Dr. Mars Caroline Wibowo, ST, M.Mm.Tech.

DASAR DESAIN GRAFIS

JILID 1

YAYASAN PRIMA AGUS TEKNIK

DASAR DESAIN GRAFIS jilid 1

Penulis :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

ISBN : 9 786235 734460

Editor :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Penyunting :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji syukur kami panjatkan atas kehadirat Tuhan karena buku yang berjudul "*Dasar Desain Grafis: Jilid 1*" dapat terselesaikan. Grafis komputer adalah bidang yang berkembang pesat, sehingga spesifikasi pengetahuan itu adalah target yang bergerak. Oleh karena itu, dalam buku ini penulis melakukan yang terbaik untuk menghindari ketergantungan pada perangkat keras atau API tertentu. Pembaca didorong untuk melengkapi teks dengan dokumentasi yang relevan untuk perangkat lunak dan lingkungan perangkat keras mereka. Untungnya, budaya grafis komputer memiliki terminologi dan konsep standar yang cukup sehingga pembahasan dalam buku ini dapat dipetakan dengan baik ke sebagian besar lingkungan.

Buku ini dibagi menjadi 2 Jilid. Jilid 1 terdiri dari 13 Bab, Bab Pertama buku jilid 1 ini memberikan definisi beberapa terminologi dasar dan memberikan beberapa latar belakang sejarah, serta sumber informasi yang berkaitan dengan komputer grafis. Bab 2 ini tidak dimaksudkan untuk membahas materi secara menyeluruh; sebaliknya intuisi dan interpretasi geometris yamh sangat ditekankan. Bab 3 Dalam bab ini, kita membahas dasar-dasar gambar dan tampilan raster, dengan memberikan perhatian khusus pada ketidaklinieran tampilan standar. Detail arti sebenarnya dari gambar. nilai piksel yang berhubungan dengan intensitas cahaya penting untuk diingat ketika kita membahas komputasi gambar di bab selanjutnya.

Bab 4 mencakup metode dasar untuk pembangkitan sinar, perpotongan sinar, dan bayangan, yang cukup untuk mengimplementasikan pelacak sinar demonstrasi sederhana. Untuk sistem yang benar-benar berguna, teknik perpotongan sinar yang lebih efisien. Sedangkan Bab 5 meninjau aljabar linier dasar dari perspektif geometris, dengan fokus pada intuisi dan algoritma yang bekerja dengan baik dalam kasus dua dan tiga dimensi. Bab 6 membahas mengenai transformasi geometris seperti rotasi, translasi, scalling, dan proyeksi dapat diselesaikan dengan perkalian matriks, dan matriks transformasi yang digunakan untuk melakukan ini adalah pokok bahasan bab ini. Bab 7 menjelaskan bagaimana menggunakan transformasi matriks untuk mengekspresikan pandangan paralel atau perspektif. Transformasi dalam bab ini memproyeksikan titik 3D dalam *scene* (ruang dunia) ke titik 2D dalam gambar (ruang gambar), dan mereka akan memproyeksikan titik mana pun pada sinar tampilan piksel tertentu kembali ke posisi piksel tersebut dalam ruang gambar.

Bab 8 akan membahas tentang fokus pada dasar-dasar umum seperti pipeline fan rasterisasi. Dalam bab 9, dimulai dengan meringkas pengambilan sampel dan rekonstruksi menggunakan contoh konkret satu dimensi dari audio digital. Kemudian, kami akan menyajikan matematika dasar dan algoritma yang mendasari pengambilan sampel dan rekonstruksi dalam satu dan dua dimensi. Bab 10 menyajikan metode bayangan heuristik yang paling umum. Diantaranya adalah difus dan phongshading, metode lain nya adalah naungan artistik, menggunakan konvensi artistik untuk memberi warna pada objek. Bab 11 membahas penggunaan tekstur untuk merepresentasikan detail permukaan, bayangan, dan pantulan.

Meskipun ide dasarnya sederhana, beberapa masalah praktis memperumit penggunaan tekstur.

Bab 12 berbicara tentang beberapa kategori struktur data dasar dan tidak terkait yang termasuk di antara yang paling umum dan berguna: struktur mesh, struktur data spasial, grafis *scene,* dan array multidimensi bersusun. Bab terakhir dalam buku jilid 1 ini membahas beberapa teknik yang lebih menarik yang dapat digunakan untuk *ray-trace* berbagai scene yang lebih luas dan untuk memasukkan lebih banyak variasi efek.

Buku Jilid 2 yang berjudul sama ini memiliki 13 bab yang akan membahas implementasi dan permodelan Dasar grafis komputer. Bab pertama dalam buku jilid 2 ini, akan membahas mesin untuk operasi probabilitas. Teknik-teknik ini juga akan terbukti berguna untuk mengevaluasi integral rumit secara numerik menggunakan integrasi Monte Carlo. Bab 2 membahas tentang Kurva, yang mencakup dengan permodelan geometris. Bab ke 3 akan memberikan gambaran tentang teknik dan algoritma yang langsung digunakan untuk membuat dan memanipulasi gerak. Bab 4 menjadi panduan pengantar untuk perangkat keras grafis dan dapat digunakan sebagai dasar untuk serangkaian praktik mingguan yang menyelidiki perangkat keras grafis.

Dalam bab 5 ini, membahas masalah praktis pengukuran cahaya, biasanya disebut radiometri. Bab 6 membahas tentang Warna, yaitu membahas teori dan matematika tentang koreksi warna. Bab 7 memberikan gambaran sebagian tentang apa yang diketahui tentang persepsi visual pada orang. Sedangan bab 8 membahas tentang pemilihan algoritma nada yang dapat digunakan untuk menghitung tingkat adaptasi lokal untuk fungsi kompresi sigmoidal.

Dalam bab 9 ini, membahas metode dan menjelaskan cara membangun model implisit kerangka secara lebih rinci. Bab ke 10 dalam buku ini membahas tentang globaliluminasi, seperti apakah yang dinamakan globaliluminati dan kegunaannya akan dibahas dalam bab ini. Bab 11 membahas beberapa aspek yang paling penting secara visual dari sifat material dan beberapa model yang cukup sederhana yang berguna dalam menangkap sifat-sifat ini. Ada banyak model BRDF yang digunakan dalam grafis, dan model yang disajikan di sini dimaksudkan untuk memberikan gambaran tentang BRDF yang tidak menyebar.

Bab 12 penulis akan merinci pertimbangan khusus yang berlaku untuk grafis dalam pengembangan game, mulai dari platform tempat game dijalankan hingga proses produksi game. Bab akhir dalam buku jilid 2 ini akan menjelaskan tentang visualisasi. Visualisasi dapat digunakan untuk menghasilkan hipotesis baru ketika menjelajahi kumpulan data yang sama sekali tidak dikenal, untuk mengkonfirmasi hipotesis yang ada dalam kumpulan data yang dipahami sebagian, atau untuk menyajikan informasi tentang kumpulan data yang diketahui kepada audiens lain. Akhir kata semoga buku ini bermanfaat bagi para pembaca.

Semarang, Februari 2022
Penulis

Dr. Mars Caroline Wibowo, M.Mm.Tech.

DAFTAR ISI

BAB 1
PENDAHULUAN

Pengantar

Istilah grafis komputer menggambarkan segala sesuatu yang menggunakan komputer untuk membuat dan memanipulasi gambar. Buku ini menyediakan algoritma dan alat matematika yang dapat Anda gunakan untu k membuat semua jenis gambar, termasuk efek visual fotorealistik, ilustrasi teknis yang informatif, dan animasi komputer yang indah. Grafis bisa dua dimensi atau tiga dimensi. Gambar dapat sepenuhnya digabungkan atau dibuat dengan memproses foto. Buku ini adalah tentang algoritma dan matematika dasar, khususnya matematika yang digunakan untuk membuat gambar komposit objek dan scene 3D.

Sebenarnya melakukan pekerjaan dalam komputer grafis membutuhkan pengetahuan tentang perangkat keras tertentu, format file, dan API grafis.
Catatan : *API: application program interface/antarmuka program aplikasi.*

1.1 AREA GRAFIS

Menerapkan kategori pada bidang apa pun berbahaya, tetapi sebagian besar praktisi grafis akan setuju pada bidang utama grafis komputer berikut:

- **Modelling** berkaitan dengan spesifikasi matematis dari bentuk dan tampilan properti dengan cara yang dapat disimpan di komputer. Misalnya, cangkir kopi dapat digambarkan sebagai sekumpulan titik 3D yang berurutan bersama dengan beberapa aturan interpolasi untuk menghubungkan titik-titik tersebut dan model pantulan yang menjelaskan interaksi cahaya pertunjukan dengan cangkir.
- **Rendering** adalah istilah yang diwarisi dari seni dan berurusan dengan pembuatan gambar berbayang dari model komputer 3D.
- **Animasi** adalah teknik untuk menciptakan ilusi gerak melalui rangkaian gambar. Animasi menggunakan pemodelan dan rendering tetapi menambahkan masalah utama pergerakan dari waktu ke waktu, yang biasanya tidak dibahas dalam pemodelan dan rendering dasar.

Ada banyak area lain yang melibatkan grafis komputer Area terkait tersebut meliputi:

- ***User Interaction*/Interaksi pengguna** berurusan dengan antarmuka antara perangkat input seperti mouse dan tablet, aplikasi, umpan balik (*feedback*) ke pengguna dalam gambar, dan umpan balik sensorik lainnya. Secara historis, area ini terkait dengan grafis sebagian besar karena peneliti grafis memiliki beberapa akses paling awal ke perangkat input/output yang sekarang ada di mana-mana.
- ***Virtual Reality*** untuk memasukkan pengguna ke dalam dunia virtual 3D. Ini biasanya membutuhkan setidaknya stereografis dan respons terhadap gerakan kepala. *Virtual reality* yang nyata, umpan balik suara dan kekuatan harus disediakan juga. Karena area ini memerlukan grafis 3D canggih dan teknologi tampilan canggih, area ini sering dikaitkan dengan grafis.
- **Visualisasi** mencoba memberi pengguna wawasan tentang informasi kompleks melalui tampilan visual. Seringkali ada masalah grafis yang harus diatasi dalam masalah visualisasi.
- **Image Processing/Pemrosesan gambar** berhubungan dengan manipulasi gambar 2D dan digunakan baik dalam bidang grafis maupun penglihatan.

- **3D Scanning/Pemindaian 3D** menggunakan teknologi pencarian jarak untuk membuat model 3D terukur. Model seperti itu berguna untuk menciptakan gambar visual yang kaya, dan pemrosesan model seperti itu sering kali membutuhkan algoritma grafis
- **Fotografi komputasional** adalah penggunaan grafis komputer, visi komputer, dan metode pemrosesan gambar untuk memungkinkan cara baru menangkap objek, scene, dan lingkungan secara fotografis

1.2 APLIKASI UTAMA

Hampir semua hal menggunakan grafis komputer, tetapi konsumen utama teknologi grafis komputer meliputi industri berikut:

- **Video Game** semakin banyak menggunakan model dan algoritma rendering 3D yang canggih.
- **Kartun** sering ditampilkan langsung dari model 3D. Banyak kartun 2D tradisional menggunakan latar belakang yang dirender dari model 3D, yang memungkinkan sudut pandang yang terus bergerak tanpa banyak waktu artis.
- **Efek visual** menggunakan hampir semua jenis teknologi grafis komputer. Hampir setiap film modern menggunakan pengomposisian digital untuk menempatkan latar belakang dengan latar depan yang difilmkan secara terpisah. Banyak film juga menggunakan pemodelan dan animasi 3D untuk menciptakan lingkungan sintetik, objek, dan bahkan karakter yang kebanyakan penonton tidak akan pernah menduga bahwa itu tidak nyata.
- **Film animasi** menggunakan banyak teknik yang sama yang digunakan untuk efek visual, tetapi tanpa harus membidik gambar yang terlihat nyata.
- **CAD/CAM** adalah singkatan dari *computer-aided-design* dan *computer-aided-manufacturing*. Bidang-bidang ini menggunakan teknologi komputer untuk merancang komponen dan produk di komputer dan kemudian, menggunakan desain virtual ini, untuk memandu proses manufaktur. Misalnya, banyak bagian mekanis dirancang dalam paket pemodelan komputer 3D dan kemudian diproduksi secara otomatis pada perangkat penggilingan yang dikendalikan komputer.
- **Simulasi** dapat dianggap sebagai permainan video yang akurat. Misalnya, simulator penerbangan menggunakan grafis 3D canggih untuk mensimulasikan pengalaman menerbangkan pesawat. Simulasi semacam itu dapat sangat berguna untuk pelatihan awal dalam domain kritis-keselamatan seperti mengemudi, dan untuk pelatihan skenario bagi pengguna berpengalaman seperti situasi khusus pemadam kebakaran yang terlalu mahal atau berbahaya untuk dibuat secara fisik.
- **Medical Imaging** menciptakan gambar yang berarti dari data pasien yang dipindai. Misalnya, kumpulan data *computed tomography* (CT) terdiri dari array persegi panjang 3D yang besar dengan nilai kepadatan. Grafis komputer digunakan untuk membuat gambar berbayang yang membantu dokter mengekstrak informasi yang paling menonjol dari data tersebut.
- **Visualisasi informasi** membuat gambar data yang tidak memiliki penggambaran visual yang "alami". Misalnya, tren harga sementara saham sering kali berbeda tidak memiliki gambaran visual yang jelas, tetapi teknik grafis yang cerdas dapat membantu manusia melihat pola dalam data tersebut.

1.3 API GRAFIS

Bagian penting dari penggunaan *graphics libraries* adalah graphicsAPI. Antarmuka program aplikasi/A*pplication program interface* (API) adalah kumpulan fungsi standar untuk

melakukan serangkaian operasi terkait, dan API grafis adalah seperangkat fungsi yang melakukan operasi dasar seperti menggambar gambar dan permukaan 3D ke dalam windows di layar.

Setiap program grafis harus dapat menggunakan dua API terkait: API grafis untuk output visual dan API antarmuka pengguna untuk mendapatkan iput dari User. Saat ini ada dua paradigma dominan untuk grafis dan API antarmuka pengguna. Yang pertama adalah pendekatan terintegrasi, yang dicontohkan oleh Java, di mana grafis dan toolkit antarmuka pengguna terintegrasi dan paket portabel yang sepenuhnya distandarisasi dan didukung sebagai bagian dari bahasa. Yang kedua diwakili oleh Direct3D dan OpenGL, di mana perintah menggambar adalah bagian dari perpustakaan perangkat lunak yang terkait dengan bahasa seperti C++, dan perangkat lunak antarmuka pengguna adalah entitas independen yang mungkin berbeda dari sistem ke sistem. Dalam pendekatan yang terakhir ini, menulis kode portabel bermasalah, meskipun untuk program sederhana dimungkinkan untuk menggunakan lapisan pustaka portabel untuk merangkum kode antarmuka pengguna khusus sistem. Apa pun pilihan API Anda, panggilan grafis dasar sebagian besar akan sama, dan konsep buku ini akan berlaku.

1.4 SALURAN GRAFIS

Setiap komputer desktop saat ini memiliki saluran grafis 3D yang kuat. Ini adalah subsistem perangkat lunak/perangkat keras khusus yang secara efisien menggambar primitif 3D dalam perspektif. Biasanya sistem ini dioptimalkan untuk memproses segitiga 3D dengan simpul bersama. Operasi dasar dalam pipeline memetakan lokasi vertex 3D ke posisi layar 2D dan menaungi segitiga sehingga keduanya terlihat realistis dan muncul dalam urutan back-to-front yang tepat.

Meskipun menggambar segitiga dengan urutan back-to-front yang valid merupakan masalah penelitian paling penting dalam grafis komputer, sekarang hampir selalu diselesaikan dengan menggunakan *buffer-z,* yang menggunakan buffer memori khusus untuk menyelesaikan masalah dengan cara yang kasar.

Ternyata manipulasi geometris yang digunakan dalam saluran grafis dapat diselesaikan hampir seluruhnya dalam ruang koordinat 4D yang terdiri dari tiga koordinat geometris tradisional dan koordinat homogen keempat yang membantu tampilan perspektif. Koordinat 4D ini dimanipulasi menggunakan matriks 4 × 4 dan 4-vektor. Oleh karena itu, pipeline grafis berisi banyak mesin untuk memproses dan menyusun matriks dan vektor tersebut secara efisien. Sistem koordinat 4D ini adalah salah satu konstruksi paling halus dan indah yang digunakan dalam ilmu komputer, dan tentu saja merupakan rintangan intelektual terbesar yang harus dilewati ketika mempelajari grafis komputer. Sebagian besar bagian pertama dari setiap buku grafis berhubungan dengan koordinat ini.

Kecepatan di mana gambar dapat dihasilkan sangat bergantung pada jumlah segitiga yang digambar. Karena interaktivitas lebih penting dalam banyak aplikasi daripada kualitas visual, sangat bermanfaat untuk meminimalkan jumlah segitiga yang digunakan untuk mewakili model. Selain itu, jika model dilihat dari jauh, segitiga yang dibutuhkan lebih sedikit daripada jika model dilihat dari jarak yang lebih dekat. Hal ini menunjukkan bahwa sangat berguna untuk merepresentasikan model dengan berbagai tingkat detail/*level of detail* (LOD).

1.5 MASALAH NUMERIK

Banyak program grafis sebenarnya hanya kode numerik 3D. Isu numerik seringkali penting dalam program semacam itu. Di "masa lalu", sangat sulit untuk menangani masalah seperti itu dengan cara yang kuat dan portabel karena mesin memiliki representasi internal

yang berbeda untuk angka, dan lebih buruk lagi, menangani pengecualian dengan cara yang berbeda dan tidak kompatibel. Untungnya, hampir semua komputer modern sesuai dengan standar titik mengambang IEEE (Asosiasi Standar IEEE, 1985). Hal ini memungkinkan programmer untuk membuat banyak asumsi nyaman tentang bagaimana kondisi numerik tertentu akan ditangani.

Catatan : IEEE floating-point memiliki dua representasi untuk nol, satu yang diperlakukan sebagai positif dan satu yang diperlakukan sebagai negatif. Perbedaan antara –0 dan +0 hanya kadang-kadang penting, tetapi perlu diingat untuk saat-saat ketika itu penting.

Meskipun IEEE *floating-point* memiliki banyak fitur yang berharga saat mengkodekan algoritma numerik, hanya ada beberapa yang penting untuk diketahui untuk sebagian besar situasi yang dihadapi dalam grafis. Pertama, dan yang paling penting, adalah memahami bahwa ada tiga nilai "khusus" untuk bilangan real di titik-mengambang IEEE:

1. **Tak terhingga (∞).** Ini adalah angka valid yang lebih besar dari semua angka valid lainnya.
2. **Minus tak terhingga (−∞).** Ini adalah nomor valid yang lebih kecil dari semua nomor valid lainnya.
3. **Not an number (NaN).** Ini adalah bilangan yang tidak valid yang muncul dari operasi dengan konsekuensi yang tidak ditentukan, seperti nol dibagi dengan nol.

Perancang IEEE *floating-point* membuat beberapa keputusan yang sangat nyaman bagi programmer. Banyak dari ini berhubungan dengan tiga nilai khusus di atas dalam menangani pengecualian seperti pembagian dengan nol. Dalam kasus ini, pengecualian dicatat, tetapi dalam banyak kasus, pemrogram dapat mengabaikannya. Secara khusus, untuk sembarang bilangan real positif a, aturan berikut yang melibatkan pembagian dengan nilai tak hingga berlaku:

$$+a/(+\infty) = +0\ ,$$
$$-a/(+\infty) = -0,$$
$$+a/(-\infty) = -0,$$
$$-a/(-\infty) = +0\ .$$

Operasi lain yang melibatkan nilai tak terhingga berperilaku seperti yang diharapkan. Sekali lagi untuk a positif, perilakunya adalah sebagai berikut:

$$\infty+\infty = +\infty,$$
$$\infty-\infty = \mathrm{NaN},$$
$$\infty\times\infty = \infty,$$
$$\infty/\infty = \mathrm{NaN},$$
$$\infty/a = \infty,$$
$$\infty/0 = \infty,$$
$$0/0 = \mathrm{NaN}.$$

Aturan dalam ekspresi Boolean yang melibatkan nilai tak hingga adalah seperti yang diharapkan:

1. Semua bilangan valid hingga kurang dari +∞.
2. Semua bilangan valid berhingga lebih besar dari -∞.
3. -∞ lebih kecil dari +∞

Aturan yang melibatkan ekspresi yang memiliki nilai NaN sederhana:

1. Ekspresi aritmatika apa pun yang menyertakan NaN menghasilkan NaN.
2. Ekspresi Boolean apa pun yang melibatkanNaN salah.

Mungkin aspek yang paling berguna dari titik IEEE adalah bagaimana pembagian dengan nol ditangani; untuk setiap bilangan real positif, aturan berikut yang melibatkan pembagian dengan nilai nol berlaku:

$$+a/+0 = +\infty,$$

$$-a/+0 = -\infty.$$

Ada banyak perhitungan numerik yang menjadi lebih sederhana jika programmer memanfaatkan aturan IEEE. Misalnya, perhatikan ekspresi berikut ini:

$$a = \frac{1}{\frac{1}{b} + \frac{1}{c}}$$

Ekspresi seperti itu muncul dengan resistor dan lensa. Jika pembagian dengan nol mengakibatkan crash program (seperti yang terjadi dalam banyak sistem sebelum titik mengambang IEEE), maka dua pernyataan if akan diperlukan untuk memeriksa nilai b atau c yang kecil atau nol. Sebaliknya, dengan IEEE floating-point, jika b dan c adalah Nol, maka kita akan mendapatkan nilai Nol juga untuk a seperti yang diinginkan. Teknik umum lainnya untuk menghindari pemeriksaan khusus adalah dengan memanfaatkan properti Boolean dari NaN. Perhatikan segmen kode berikut:

a = f(x)
Jika/if *(a>0)* ***maka/then*** *lakukan yang lain*

Di sini, fungsi f dapat mengembalikan nilai "jelek" seperti ∞ atau NaN, tetapi kondisi if masih terdefinisi dengan baik: salah untuk a = NaN atau a = dan benar untuk a = +∞. Dengan hati-hati dalam memutuskan nilai mana yang dikembalikan, seringkali if dapat membuat pilihan yang tepat, tanpa memerlukan pemeriksaan khusus. Hal ini membuat program lebih kecil, lebih kuat, dan lebih efisien.

1.6 EFESIENSI

Tidak ada aturan ajaib untuk membuat kode lebih efisien. Efisiensi dicapai melalui pengorbanan yang hati-hati, dan pengorbanan ini berbeda untuk arsitektur yang berbeda. Namun, untuk masa mendatang, heuristik yang baik adalah bahwa programmer harus lebih memperhatikan pola akses memori daripada jumlah operasi. Ini adalah kebalikan dari heuristik terbaik dua dekade lalu. Peralihan ini terjadi karena kecepatan memori tidak mengikuti kecepatan prosesor. Sejak tren itu berlanjut, pentingnya akses memori yang terbatas dan koheren untuk optimalisasi hanya akan meningkat.

Pendekatan yang masuk akal untuk membuat kode cepat adalah dengan melanjutkan dalam urutan berikut, hanya mengambil langkah-langkah yang diperlukan:

1. Tulis kode dengan cara yang paling mudah. Hitung hasil antara sesuai kebutuhan pada y daripada menyimpannya.
2. Kompilasi dalam mode yang dioptimalkan.
3. Gunakan alat profil apa pun yang ada untuk menemukan hambatan kritis.
4. Periksa struktur data untuk mencari cara untuk meningkatkan lokalitas. Jika memungkinkan, buat ukuran unit data sesuai dengan ukuran cache/halaman pada arsitektur target.
5. Jika profil mengungkapkan hambatan dalam komputasi numerik, periksa kode perakitan yang dihasilkan oleh kompilator untuk mengetahui efisiensi yang terlewat. Tulis ulang source code untuk memecahkan masalah yang Anda temukan.

Yang paling penting dari langkah-langkah ini adalah yang pertama. Sebagian besar "Optimalisasi" membuat kode lebih sulit dibaca tanpa mempercepat segalanya. Selain itu, waktu yang dihabiskan di muka untuk mengoptimalkan kode biasanya lebih baik dihabiskan untuk memperbaiki bug atau menambahkan fitur. Juga, waspadalah terhadap saran dari teks-teks lama; beberapa trik klasik seperti menggunakan bilangan bulat alih-alih nyata mungkin tidak lagi menghasilkan kecepatan karena CPU modern biasanya dapat melakukan operasi titik mengambang secepat mereka melakukan operasi bilangan bulat. Dalam semua situasi,

pembuatan profil diperlukan untuk memastikan manfaat dari setiap optimasi untuk mesin dan kompiler tertentu.

1.7 DESAIN DAN CODING PROGRAM DESAIN

Strategi umum tertentu sering berguna dalam pemrograman grafis. Di bagian ini kami memberikan beberapa saran yang mungkin berguna bagi Anda saat menerapkan metode yang Anda pelajari dalam buku ini.

Desain Kelas

Bagian utama dari setiap program grafis adalah untuk memiliki kelas atau rutinitas yang baik untuk entitas geometris seperti vektor dan matriks, serta entitas grafis seperti warna dan gambar RGB. Rutinitas ini harus dibuat sebersih dan seefisien mungkin. Sebuah pertanyaan desain universal adalah apakah lokasi dan perpindahan harus menjadi kelas yang terpisah karena mereka memiliki operasi yang berbeda, misalnya, lokasi dikalikan dengan setengah tidak masuk akal secara geometris sementara setengah dari perpindahan tidak. Ada sedikit kesepakatan tentang pertanyaan ini, yang dapat memicu perdebatan sengit selama berjam-jam di antara praktisi grafis, tetapi sebagai contoh mari kita asumsikan kita tidak akan membuat perbedaan.

Ini menyiratkan bahwa beberapa kelas dasar yang akan ditulis meliputi:

- **vektor2.** Kelas vektor 2D yang menyimpan komponen x-andy. Ini harus menyimpan komponen ini dalam array panjang-2 sehingga operator pengindeksan dapat didukung dengan baik. Anda juga harus memasukkan operasi untuk penjumlahan vektor, pengurangan vektor, perkalian titik, perkalian silang, perkalian skalar, dan pembagian skalar.
- **vektor3**. Sebuah kelas vektor 3D analog dengan vektor2.
- **vektor**. Sebuah vektor homogen dengan empat komponen .
- **rgb**. Warna RGB yang menyimpan tiga komponen. Anda juga harus menyertakan operasi untuk penambahan RGB, pengurangan RGB, perkalian RGB, perkalian skalar, dan pembagian skalar
- **transform**. Matriks 4x4 untuk transformasi. Anda harus menyertakan perkalian matriks dan fungsi anggota untuk diterapkan ke lokasi, arah, dan vektor normal permukaan. Seperti yang ditunjukkan dalam Bab 6, ini semua berbeda.
- **Image**. Array 2D piksel RGB dengan operasi output.

Selain itu, Anda mungkin tidak ingin menambahkan kelas untuk interval, basis ortonormal, dan bingkai koordinat.

Mengapung vr Ganda

Arsitektur modern menunjukkan bahwa menjaga penggunaan memori tetap rendah dan mempertahankan akses memori yang koheren adalah kunci efisiensi. Ini menyarankan menggunakan data presisi tunggal. Namun, menghindari masalah numerik menyarankan menggunakan doubleprecisionarithmetic. Pengorbanan tergantung pada programnya, tetapi bagus untuk memiliki default dalam definisi kelas Anda.

Debugging Program Grafis

Anda mungkin menemukan bahwa sebagai programmer menjadi lebih berpengalaman, mereka menggunakan debuggers tradisional dan kurang. Salah satu alasannya adalah bahwa menggunakan debugger seperti itu lebih canggung untuk program yang kompleks daripada untuk program yang sederhana. Alasan lain adalah bahwa kesalahan yang paling sulit adalah kesalahan konseptual di mana hal yang salah sedang diimplementasikan, dan mudah untuk membuang banyak waktu untuk melangkah melalui nilai-nilai variabel tanpa mendeteksi

kasus seperti itu. Kami telah menemukan beberapa strategi debugging yang sangat berguna dalam grafis.

Metode Ilmiah

Dalam program grafis ada alternatif untuk debugging tradisional yang seringkali sangat berguna. Kelemahannya adalah sangat mirip dengan apa yang diajarkan kepada programmer komputer untuk tidak dilakukan di awal karir mereka, jadi Anda mungkin merasa "nakal" jika Anda melakukannya: kami membuat gambar dan mengamati apa yang salah dengannya. Kemudian, kami mengembangkan hipotesis tentang apa yang menyebabkan masalah dan mengujinya. Misalnya, dalam program *ray-tracing,* kita mungkin memiliki banyak piksel gelap yang tampak acak. Ini adalah masalah " jerawat bayangan" klasik yang dialami kebanyakan orang saat mereka menulis pelacak.

Debug tradisional tidak membantu di sini; sebaliknya, kita harus menyadari bahwa sinar bayangan mengenai permukaan yang diarsir. Kita mungkin memperhatikan bahwa warna bintik-bintik gelap adalah warna sekitar, jadi pencahayaan langsunglah yang hilang. Pencahayaan langsung dapat dimatikan dalam bayangan, jadi Anda mungkin berhipotesis bahwa titik-titik ini salah ditandai sebagai bayangan padahal sebenarnya tidak. Untuk menguji hipotesis ini, kita bisa mematikan pemeriksaan bayangan dan mengkompilasi ulang. Ini akan menunjukkan bahwa ini adalah tes bayangan palsu, dan kami dapat melanjutkan pekerjaan detektif kami. Alasan utama mengapa metode ini terkadang dapat menjadi praktik yang baik adalah karena kita tidak pernah harus menemukan nilai yang salah atau benar-benar menentukan kesalahan konseptual kita. Sebaliknya, kami hanya mempersempit kesalahan konseptual kami secara eksperimental. Biasanya hanya beberapa percobaan yang diperlukan untuk melacak semuanya, dan jenis debugging ini menyenangkan.

Gambar sebagai Output Debugging Berkode

Dalam banyak kasus, saluran termudah untuk mendapatkan informasi debug dari program grafis adalah gambar output itu sendiri. Jika Anda ingin mengetahui nilai beberapa variabel untuk bagian komputasi yang berjalan untuk setiap piksel, Anda dapat memodifikasi program Anda untuk sementara untuk menyalin nilai tersebut secara langsung ke gambar output dan melewatkan sisa penghitungan yang biasanya dilakukan. Misalnya, jika Anda menduga masalah dengan normal permukaan menyebabkan masalah dengan bayangan, Anda dapat menyalin vektor normal langsung ke gambar (x menjadi merah, y menjadi hijau, z menjadi biru), menghasilkan ilustrasi kode warna dari vektor yang sebenarnya digunakan dalam perhitungan Anda. Atau, jika Anda menduga nilai tertentu terkadang berada di luar rentang validnya, buat program Anda menulis piksel cerah di tempat yang terjadi. Trik umum lainnya termasuk menggambar sisi belakang permukaan dengan warna yang jelas (ketika seharusnya tidak terlihat), mewarnai gambar dengan nomor ID objek, atau mewarnai piksel dengan jumlah pekerjaan yang mereka lakukan untuk menghitung.

Menggunakan Debugging

Masih ada kasus, terutama ketika metode ilmiah tampaknya telah menyebabkan kontradiksi, ketika tidak ada pengganti untuk mengamati dengan tepat apa yang sedang terjadi. Masalahnya adalah bahwa program grafis sering kali melibatkan banyak, banyak eksekusi dari kode yang sama (misalnya, sekali per piksel, atau sekali per segitiga), sehingga sama sekali tidak praktis untuk melangkah melalui debugger dari awal. Dan bug yang paling sulit biasanya hanya terjadi untuk input yang rumit.

Pendekatan yang berguna adalah "menetapkan jebakan" untuk bug. Pertama, pastikan program Anda deterministik—berjalan dalam satu utas dan pastikan bahwa semua bilangan acak dihitung dari benih tetap. Kemudian, cari tahu piksel atau segitiga mana yang menunjukkan bug dan tambahkan pernyataan sebelum kode yang Anda curigai salah yang

akan dieksekusi hanya untuk kasus yang dicurigai. Misalnya, jika Anda menemukan bahwa piksel (126.247) menunjukkan bug, tambahkan:

if *x = 126 dan y = 247* ***then***
print "gambar!"

Jika Anda menyetel *breakpoint* pada pernyataan cetak, Anda dapat memasukkan *debugger* tepat sebelum piksel yang Anda minati dihitung. Beberapadebugger memiliki fitur "*breakpoint* bersyarat" yang dapat mencapai hal yang sama tanpa mengubah kode.

Catatan: *Mode debug khusus yang menggunakan seed nomor acak tetap berguna.*

Dalam kasus di mana program mogok, debugger tradisional berguna untuk menunjukkan dengan tepat situs mogok. Anda kemudian harus mulai melacak kembali dalam program, menggunakan asserts dan mengkompilasi ulang, untuk menemukan di mana letak kesalahan program. Pernyataan ini harus dibiarkan dalam program untuk kemungkinan bug di masa mendatang yang akan Anda tambahkan. Ini sekali lagi berarti proses langkah-melalui tradisional dihindari, karena itu tidak akan menambahkan pernyataan berharga ke program Anda.

Visualisasi Data untuk Debugging

Seringkali sulit untuk memahami apa yang sedang dilakukan program Anda, karena ia menghitung banyak hasil antara sebelum akhirnya salah. Situasinya mirip dengan eksperimen ilmiah yang mengukur banyak data, dan satu solusinya sama: buat plot dan ilustrasi yang bagus untuk Anda sendiri untuk memahami apa arti data itu. Misalnya, dalam ray tracer Anda dapat menulis kode untuk memvisualisasikan pohon ray sehingga Anda dapat melihat jalur apa yang berkontribusi pada piksel, atau dalam rutinitas pengambilan sampel ulang gambar, Anda dapat membuat plot yang menunjukkan semua titik di mana sampel diambil dari input. Waktu yang dihabiskan untuk menulis kode untuk memvisualisasikan keadaan internal program Anda juga terbayar dalam pemahaman yang lebih baik tentang perilakunya ketika tiba saatnya untuk mengoptimalkannya.

BAB 2
MACAM-MACAM MATEMATIKA

Sebagian besar grafis hanya menerjemahkan matematika langsung ke dalam kode. Matematika bersih, kode hasil pembersih, sebagian besar dari buku ini berkonsentrasi pada penggunaan matematika yang tepat untuk pekerjaan itu. Bab ini mengulas berbagai alat dari matematika sekolah menengah dan perguruan tinggi dan dirancang untuk digunakan lebih sebagai referensi daripada sebagai tutorial. Ini mungkin tampak seperti campuran topik dan memang demikian; setiap topik dipilih karena agak tidak biasa dalam kurikulum matematika "standar", karena sangat penting dalam grafis, atau karena biasanya tidak diperlakukan dari sudut pandang geometris. Selain membangun ulasan tentang notasi yang digunakan dalam buku, bab ini juga menekankan beberapa poin yang terkadang dilewati dalam kurikulum standar sarjana, seperti koordinat barycentric pada segitiga.

Sebuah diskusi tentang aljabar linier ditunda sampai Bab 5 sebelum matriks transformasi dibahas. Pembaca didorong untuk membaca sekilas bab ini untuk membiasakan diri dengan topik yang dibahas dan untuk merujuk kembali jika diperlukan. Latihan di akhir bab mungkin berguna dalam menentukan topik mana yang perlu disegarkan.

2.1 MAPPING DAN PENGATURAN

Mapping, juga disebut *fungsi*, ini adalah dasar matematika dan pemrograman. Seperti fungsi dalam program, mapping dalam matematika mengambil argumen dari satu tipe dan memetakannya ke (mengembalikan) objek dari tipe tertentu. Dalam sebuah program kami mengatakan "ketik"; dalam matematika kita akan mengidentifikasi himpunan. Ketika kita memiliki objek yang merupakan anggota dari suatu himpunan, kita menggunakan simbol ∈. Sebagai contoh,

$$a \in S$$

dapat dibaca “a adalah anggota himpunan S”. Diketahui dua himpunan A dan B, kita dapat membuat himpunan ketiga dengan mengambil *produk Cartesian* dari dua himpunan, dilambangkan dengan A × B. Himpunan A × B ini terdiri dari semua kemungkinan pasangan terurut (a,b) di mana a ∈ A dan b ∈ B. Sebagai singkatan, kami menggunakan notasi A^2 untuk menunjukkan A×A. Kita dapat memperluas hasil kali Cartesian untuk membuat himpunan dari semua kemungkinan rangkap tiga terurut dari tiga himpunan dan seterusnya untuk tupel terurut panjang sewenang-wenang dari banyak himpunan sewenang-wenang. Commonset yang menarik termasuk

- R—bilangan real;
- R^+—bilangan real nonnegatif (termasuk nol);
- R^2—pasangan terurut dalam bidang 2D nyata;
- R^n—titik-titik dalam ruang Kartesius n-dimensi;
- Z—bilangan bulat;
- S^2—kumpulan titik 3D (titik di R^3) pada unit bola.

Perhatikan bahwa meskipun S^2 terdiri dari titik-titik yang tertanam dalam ruang tiga dimensi, mereka berada di permukaan yang dapat diparameterisasi dengan dua variabel, sehingga dapat dianggap sebagai himpunan 2D. Notasi untuk mapping menggunakan tanda panah dan titik dua, misalnya:

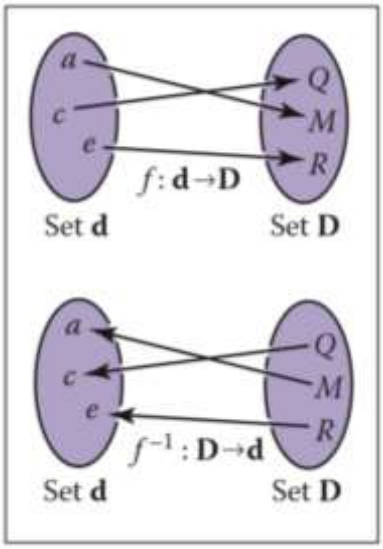
$$f : R \to Z,$$

yang dapat Anda baca sebagai "Ada fungsi yang disebut *f* yang mengambil bilangan real sebagai input dan peta untuk bilangan bulat." Di sini, himpunan yang muncul sebelum

panah disebut domain fungsi, dan himpunan di ruas kanan disebut target. Pemrogram komputer mungkin lebih nyaman dengan bahasa setara berikut: "Ada fungsi yang disebut *f* yang memiliki satu argumen nyata dan mengembalikan bilangan bulat." Dengan kata lain, notasi himpunan di atas setara dengan notasi pemrograman umum:

bilangan bulat f(real) ← ekuivalen → f : R → Z.

Jadi notasi tanda titik dua dapat dianggap sebagai sintaks pemrograman. Sesederhana itu. Titik *f(a)* disebut *image* dari a, dan bayangan himpunan A (subset dari domain) adalah himpunan bagian dari target yang memuat bayangan semua titik di A. Bayangan seluruh domain adalah disebut jangkauan fungsi.

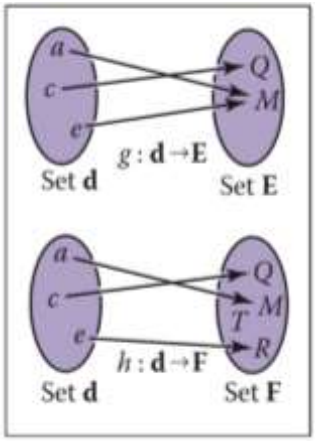


Gambar 2.1. Bijeksi dan fungsi invers f^{-1}. Perhatikan bahwa f^{-1} juga merupakan bijeksi.

Invers Mapping

Jika kita memiliki fungsi f : A → B, mungkin ada fungsi invers f^{-1} : B → A, yang didefinisikan oleh aturan $f^{-1}(b)=a$ di mana b = f(a). Definisi ini hanya berfungsi jika setiap b ∈ B adalah bayangan dari suatu titik di bawah f (yaitu, jangkauannya sama dengan target) dan jika hanya ada satu titik tersebut (yaitu, hanya ada satu di mana f(a)=b). Mapping atau fungsi seperti itu disebut bijeksi. Sebuah bijeksi memetakan setiap a ∈ A ke b ∈ B unik, dan untuk setiap b ∈ B, ada tepat satu a ∈ A sedemikian rupa sehingga f(a)=b (Gambar 2.1). Bijeksi antara sekelompok penunggang dan kuda menunjukkan bahwa setiap orang menunggangi seekor kuda, dan setiap kuda ditunggangi. Dua fungsi akan menjadi pengendara (kuda) dan kuda (penunggang). Ini adalah fungsi terbalik satu sama lain. Fungsi yang bukan bijeksi tidak memiliki invers (Gambar 2.2).

Contoh bijeksi adalah f : $\mathbb{R} \rightarrow \mathbb{R}$, dengan $f(x)=x^3$. Fungsi inversnya adalah $f^{-1}(x)=\sqrt[3]{x}$. Contoh ini menunjukkan bahwa notasi standar bisa agak aneh karena x digunakan sebagai variabel dummy di f dan f^{-1}. Terkadang lebih intuitif untuk menggunakan variabel dummy yang berbeda, dengan y = f(x) dan $x = f^{-1}(y)$. Hasil ini semakin intuitif $y= x^3$ dan $x = \sqrt[3]{y}$. Contoh fungsi yang tidak memiliki invers adalah sqr : $\mathbb{R} \rightarrow \mathbb{R}$, dimana $sqr(x)=x^2$. Ini benar karena dua alasan: pertama x2 =(−x)2, dan kedua tidak ada anggota domain yang memetakan ke bagian negatif dari target. Perhatikan bahwa kita dapat mendefinisikan invers jika kita membatasi domain dan jangkauan ke $\mathbb{R}$+. Maka x adalah invers yang valid.

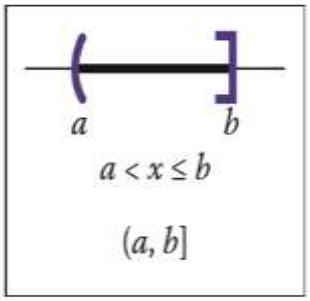


Gambar 2.2 Fungsi g tidak memiliki invers karena dua elemen d dipetakan ke elemen E yang sama. Fungsi h tidak memiliki invers karena elemen T dari F tidak memiliki elemen d yang dipetakan padanya.

Interval

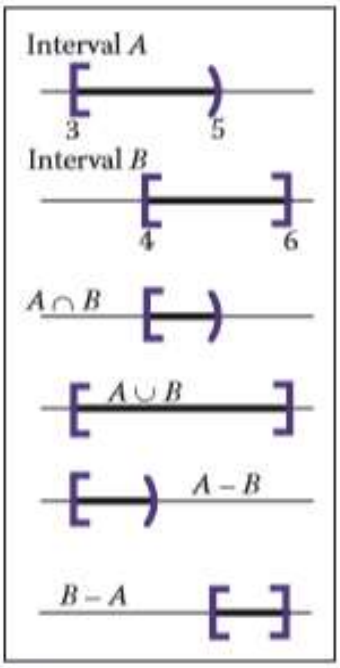
Seringkali kita ingin menentukan bahwa suatu fungsi berhubungan dengan bilangan real yang dibatasi nilainya. Salah satu kendala tersebut adalah untuk menentukan interval. Contoh interval adalah bilangan real antara nol dan satu, tidak termasuk nol atau satu. Kami menunjukkan ini (0,1). Karena tidak menyertakan titik akhirnya, ini disebut sebagai interval terbuka. Interval tertutup yang sesuai, yang memang berisi titik akhirnya, dilambangkan dengan tanda kurung siku: [0,1]. Notasi ini dapat dicampur, yaitu [0,1] termasuk nol tetapi tidak satu. Saat menulis interval [a,b], kita asumsikan bahwa a ≤ b. Tiga cara umum untuk merepresentasikan interval ditunjukkan pada Gambar 2.3. Produk interval Cartesian sering digunakan. Misalnya, untuk menunjukkan bahwa suatu titik x berada dalam kubus satuan dalam 3D, kita katakan $x \in [0,1]^3$.

Interval sangat berguna dalam hubungannya dengan operasi himpunan: *Intersection* / persimpangan, *Union* / penyatuan, *dan Different* / perbedaan. Misalnya, perpotongan dua interval adalah himpunan titik-titik yang memiliki kesamaan. Simbol digunakan untuk persimpangan. Misalnya, [3,5] ∩ [4,6] = [4,5]. *Forunions*, simbol digunakan untuk menunjukkan titik di salah satu interval. Misalnya, [3,5] ∪ [4,6] = [3,6]. Berbeda dengan dua operator pertama, operator perbedaan menghasilkan hasil yang berbeda tergantung pada urutan argumen.



Gambar 2.3 (kiri atas) Tiga cara ekuivalen untuk menyatakan interval dari a ke b yang mencakup b tetapi tidak a.

Tanda minus digunakan untuk operator selisih, yang mengembalikan titik-titik di interval kiri yang tidak juga di kanan. Misalnya, [3,5)−[4,6] = [3,4] dan [4,6]−[3,5] = [5,6]. Operasi ini sangat mudah untuk divisualisasikan menggunakan diagram interval (Gambar 2.4).



Gambar 2.4 (gambar sebelah kanan) Operasi interval pada [3,5) dan [4,6].

Logaritma

Meskipun tidak lazim hari ini seperti sebelum kalkulator, logaritma sering berguna dalam masalah di mana persamaan dengan istilah eksponensial muncul. Menurut definisi, setiap logaritma memiliki basis a. "Basis log a" dari x ditulis loga x dan didefinisikan sebagai "eksponen ke mana a harus dinaikkan untuk mendapatkan x," yaitu,

$$y = log_a x \leftrightarrow a^y = x$$

Perhatikan bahwa basis logaritma a dan fungsi yang menaikkan pangkat a adalah saling invers. Definisi dasar ini memiliki beberapa konsekuensi:

$$^a log_a(x) = x;$$
$$log_a(a^x)=x;$$
$$log_a(x^y) = log_a x + log_a y;$$
$$log_a(x/y) = log_a x - log_a y;$$
$$log_a x = log_a b \, log_b x$$

Ketika kita menerapkan kalkulus pada logaritma, angka khusus e =2.718...sering muncul. Logaritma dengan basis e disebut logaritma natural. Kami mengadopsi singkatan umum ln untuk menunjukkannya:

$$x \equiv log_e x.$$

Perhatikan bahwa simbol "≡" dapat dibaca "setara dengan definisi." Seperti π, nomor khusus e muncul dalam sejumlah konteks yang luar biasa. Banyak bidang menggunakan basis tertentu selain e untuk manipulasi dan menghilangkan basis dalam notasinya, yaitu log x. Misalnya, astronom sering menggunakan basis 10 dan ilmuwan komputer teoretis sering menggunakan basis 2. Karena grafis komputer meminjam teknologi dari banyak bidang, kita akan menghindari singkatan ini. Turunan dari logaritma dan eksponen menjelaskan mengapa logaritma natural adalah “natural”:

$$\frac{d}{dx} log_a x = \frac{1}{x \, di \, a};$$
$$\frac{d}{dx} a^x = a^x \, di \, a$$

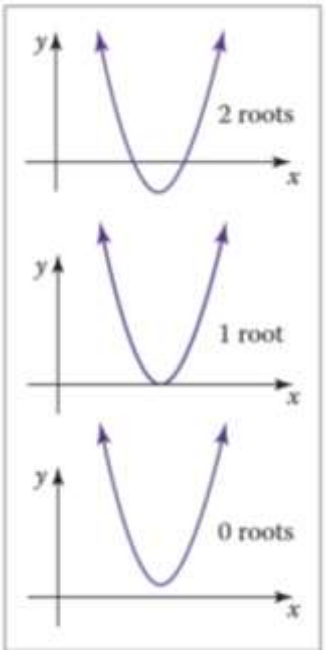
Pengganda konstanta di atas hanya satu untuk a = e.

2.2 MEMECAHKAN PERSAMAAN KUADRAT

Persamaan kuadrat memiliki bentuk

$$Ax^2 + Bx + C = 0$$

Di mana x tidak diketahui nyata, dan A, B, dan C adalah konstanta yang diketahui. Jika Anda memikirkan plot 2D *xy* dengan $y = Ax^2 + Bx + C$, solusinya adalah apa pun nilai x yang merupakan “nol persilangan” di y. Karena $y = Ax^2 + Bx + C$ adalah parabola, maka akan ada nol, satu, atau dua solusi nyata tergantung pada apakah parabola meleset, menyerempet, atau menyentuh sumbu x (Gambar 2.5).



Gambar 2.5. Interpretasi geometrik dari root (akar) persamaan kuadrat adalah titik potong parabola dengan sumbu x.

Untuk menyelesaikan persamaan kuadrat secara analitik, pertama-tama kita bagi dengan A:

$$x^2 + \frac{B}{A}x + \frac{C}{A} = 0$$

Kemudian kami "menyelesaikan kotak" ke group terms:

$$\left(x + \frac{B}{2A}\right)^2 - \frac{B^2}{4A^2} + \frac{C}{A} = 0$$

Memindahkan bagian konstan ke ruas kanan dan mengambil akar kuadrat memberikan

$$x + \frac{B}{2A} = \mp\sqrt{\frac{B^2}{4A^2} - \frac{C}{A}}$$

Mengurangi B/(2A) dari kedua sisi dan mengelompokkan suku dengan penyebut 2A menghasilkan bentuk yang sudah dikenal:[1]

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Di sini simbol "±" berarti ada dua solusi, satu dengan tanda plus dan satu lagi dengan tanda minus. Jadi 3±1 sama dengan "dua atau empat". Perhatikan bahwa suku yang menentukan banyaknya solusi nyata adalah

$$D \equiv B^2 - 4AC$$

Yang disebut diskriminan persamaan kuadrat. Jika D > 0, ada dua solusi real (disebut juga akar). Jika D = 0, ada satu solusi nyata (akar "ganda"). Jika D < 0, tidak ada solusi nyata. Misalnya, akar dari $2x^2 + 6x + 4 = 0$ adalah x = −1 dan x = 2, dan persamaan $x^2 + x + 1$ tidak memiliki solusi nyata. Diskriminan dari persamaan ini masing-masing adalah D =4 dan D = 3, jadi kita harapkan banyaknya solusi yang diberikan. Dalam program, biasanya merupakan ide yang baik

[1] Implementasi yang kuat akan menggunakan ekspresi setara $2C/(-B \mp \sqrt{B^2 - 4AC})$ untuk menghitung salah satu akar, tergantung pada tanda B (Latihan 7).

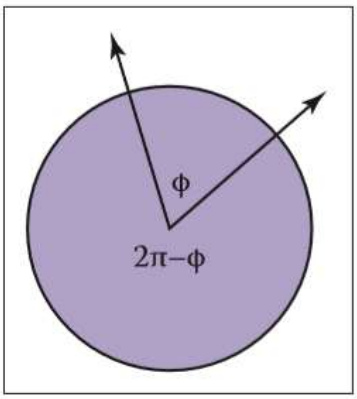
untuk mengevaluasi D terlebih dahulu dan mengembalikan "tidak ada akar" tanpa mengambil akar kuadrat jika D negatif.

2.3 TRIGONOMETRI

Dalam grafis kami menggunakan konteks dasar trigonometri inman. Biasanya, tidak ada yang terlalu mewah, dan sering kali membantu mengingat definisi dasar.

Sudut (Angel)

Meskipun kita menganggap sudut sebagai sesuatu yang biasa, kita harus kembali ke definisinya sehingga kita dapat memperluas gagasan tentang sudut ke dalam bola. Sebuah sudut terbentuk antara dua setengah garis (sinar tak hingga yang berasal dari titik asal) atau arah, dan beberapa konvensi harus digunakan untuk memutuskan antara dua kemungkinan sudut yang dibuat di antara mereka seperti yang ditunjukkan pada Gambar 2.6. Sudut didefinisikan oleh panjang segmen busur yang dipotongnya pada lingkaran satuan. Sebuah konvensi umum adalah bahwa panjang busur yang lebih kecil digunakan, dan tanda sudut ditentukan oleh urutan di mana dua setengah garis ditentukan. Menggunakan konvensi itu, semua sudut berada dalam kisaran [−π,π].



Gambar 2.6 Dua setengah garis memotong lingkaran satuan menjadi dua busur.

Panjang salah satu busur adalah sudut sah "antara" dua setengah garis. Kita dapat menggunakan konvensi bahwa panjang yang lebih kecil adalah sudutnya, atau bahwa dua setengah garis ditentukan dalam urutan tertentu dan busur yang menentukan sudut adalah busur yang ditarik berlawanan arah jarum jam dari setengah garis pertama ke kedua.

Masing-masing sudut ini adalah *panjang busur lingkaran satuan yang "dipotong" oleh dua arah*. Karena keliling lingkaran satuan adalah 2π, maka dua sudut yang mungkin berjumlah 2π. Satuan panjang busur ini adalah radian. Satuan umum lainnya adalah derajat, di mana keliling lingkaran adalah 360 derajat. Jadi, sudut radian π adalah 180°, biasanya dilambangkan 180◦. Konversi antara derajat dan radian adalah

$$derajat = \frac{180}{\pi}\ radian$$

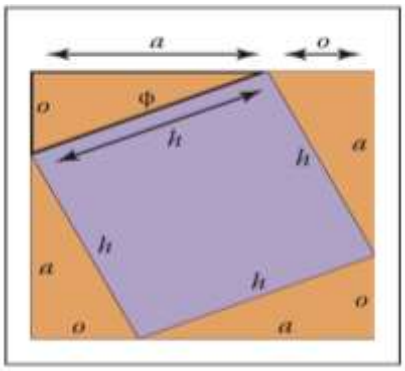
$$radian = \frac{180}{\pi}\ derajat$$

Fungsi Trigonometri

Diberikan segitiga siku-siku dengan panjang sisi a, o, dan h, di mana adalah panjang sisi terpanjang (yang selalu berlawanan dengan sudut siku-siku), atau *sisi miring*, hubungan penting dijelaskan oleh *Teorema Pythagoras*:

$$a^2 + o^2 = h^2$$

Anda dapat melihat bahwa ini benar dari Gambar 2.7, di mana kotak besar memiliki luas $(a+o)^2$, keempat segitiga memiliki luas gabungan *2ao*, dan kotak tengah memiliki luas h^2

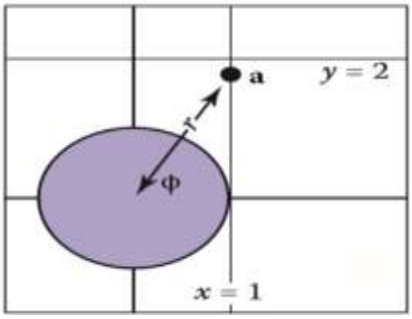


Gambar 2.7 Demonstrasi geometri dari teorema Pythagoras.

Karena segitiga dan bujur sangkar bagian dalam membagi persegi yang lebih besar secara merata, kita memiliki $2ao + h^2 = (a + o)^2$, yang mudah dimanipulasi ke bentuk di atas. Kami mendefinisikan sinus dan cosinus dari ∅, serta ekspresi trigonometri berbasis rasio lainnya:

$$\sin\phi \equiv o/h;$$
$$\csc\phi \equiv h/o;$$
$$\cos\phi \equiv a/h;$$
$$\sec\phi \equiv h/a;$$
$$\tan\phi \equiv o/a;$$
$$\cot\phi \equiv a/o.$$

Definisi ini memungkinkan kita untuk mengatur koordinat kutub, di mana sebuah titik dikodekan sebagai jarak dari titik asal dan sudut bertanda relatif terhadap sumbu x positif (Gambar 2.8). Perhatikan konvensi bahwa sudut berada dalam rentang $\emptyset \in (-\pi,\pi]$, dan bahwa sudut positif berlawanan arah jarum jam dari sumbu x positif. Konvensi ini yang berlawanan arah jarum jam memetakan ke bilangan positif arbitrer, tetapi digunakan dalam banyak konteks dalam grafissehingga perlu dilakukan Penyimpanan



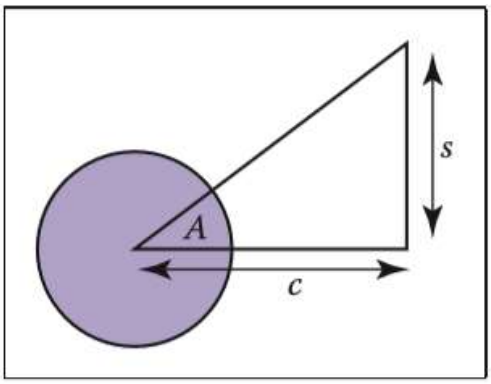
Gambar 2.8 Koordinat kutub untuk titik $(x_a, y_a) = (1,\sqrt{3})$ adalah $(r_a, \emptyset_a) = (2,\pi/3)$.

Fungsi trigonometri bersifat periodik dan dapat mengambil sudut apa pun sebagai argumen. Misalnya, sin(A) = sin(A +2π). Ini berarti fungsi-fungsi tersebut tidak dapat dibalik jika dipertimbangkan dengan domain R. Masalah ini dihindari dengan membatasi jangkauan fungsi invers standar, dan ini dilakukan dengan cara standar di hampir semua perpustakaan matematika modern (misalnya, (Plauger, 1991)). Domain dan rentang adalah:

$$\text{asin} : [-1,1] \mapsto [-\pi/2,\pi/2];$$
$$\text{acos} : [-1,1] \mapsto [0,\pi];$$
$$\text{atan} : \mathbb{R} \mapsto [-\pi/2,\pi/2];$$
$$\text{atan2} : \mathbb{R} \mapsto [-\pi,\pi].$$

Fungsi terakhir, *atan2(s,c)* seringkali sangat berguna. Dibutuhkan nilai *s* yang proporsional dengan sin *A* dan nilai *c* yang menskalakan *cosA* dengan faktor yang sama dan

mengembalikan *A*. Faktor tersebut diasumsikan positif. Salah satu cara untuk memikirkan hal ini adalah bahwa ia mengembalikan sudut titik *Cartesian 2D (s,c)* dalam koordinat kutub (Gambar 2.9).



Gambar 2.9 Fungsi *atan2(s,c)* mengembalikan sudut A dan seringkali sangat berguna dalam grafis.

Identitas yang Berguna

Bagian ini mencantumkan tanpa turunan berbagai identitas trigonometri yang berguna.

Identitas Shifting:

$$\sin(-A) = -\sin A$$
$$\cos(-A) = \cos A$$
$$\tan(-A) = -\tan A$$
$$\sin(\pi/2 - A) = \cos A$$
$$\cos(\pi/2 - A) = \sin A$$
$$\tan(\pi/2 - A) = \cot A$$

Identitas Pitagoras :

$$\sin^2 A + \cos^2 A = 1$$
$$\sec^2 A - \tan^2 A = 1$$
$$\csc^2 A - \cot^2 A = 1$$

Identitas substraksi dan tambahan :

$$\sin(A + B) = \sin A \cos B + \sin B \cos A$$
$$\sin(A-B) = \sin A \cos B - \sin B \cos A$$
$$\sin(2A) = 2 \sin A \cos A$$
$$\cos(A+B) = \cos A \cos B - \sin A \sin B$$
$$\cos(A-B) = \cos A \cos B + \sin A \sin B$$
$$\cos(2A) = \cos^2 A - \sin^2 A$$
$$\tan(A + B) = \frac{\tan A + \tan B}{1 - \tan A \tan B}$$
$$\tan(A - B) = \frac{\tan A - \tan B}{1 + \tan A \tan B}$$
$$\tan(2A) = \frac{2 \tan A}{1 - \tan^2 A}$$

Identitas Hlf-Angle (setengah sudut):

$$\sin^2(A/2) = (1-\cos A)/2$$
$$\cos^2(A/2) = (1-\cos A)/2$$

Identitas Produk :

$$\sin A \sin B = -(\cos(A+B) - \cos(A-B))/2$$
$$\sin A \cos B = (\sin(A+B) + \sin(A-B))/2$$
$$\cos A \cos B = (\cos(A+B) + \cos(A-B))/2$$

Identitas berikut adalah untuk segitiga sembarang dengan panjang sisi a, b, dan c, masing-masing dengan sudut di hadapannya diberikan oleh A, B, C, masing-masing (Gambar 2.10):

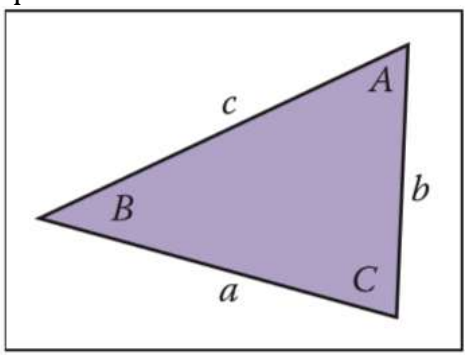
$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c} \qquad (hukum\ sin)$$

$$c^2 = a^2 + b^2 - 2ab\cos C \qquad (Hukum\ cos)$$

$$\frac{a+b}{a-b} = \frac{\tan\left(\frac{A+B}{2}\right)}{\tan\left(\frac{A-B}{2}\right)} \qquad (Hukum\ tan)$$

Luas segitiga juga dapat dihitung berdasarkan panjang sisi berikut:

$$Area\ Segitiga = \frac{1}{4}\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}$$

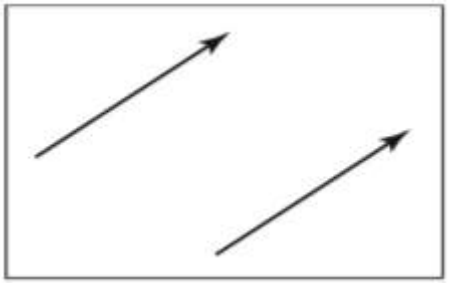


Gambar 2.10 Geometri untuk hukum segitiga.

2.4 VEKTOR

Sebuah vektor menggambarkan panjang dan arah. Hal ini dapat berguna diwakili oleh panah. Dua vektor sama jika mereka memiliki panjang dan arah yang sama bahkan jika kita menganggapnya berada di tempat yang berbeda (Gambar 2.11). Sebisa mungkin, Anda harus menganggap vektor sebagai panah dan not sebagai koordinat atau angka. Pada titik tertentu kita harus merepresentasikan vektor sebagai angka dalam program kita, tetapi bahkan dalam kode mereka harus dimanipulasi sebagai objek dan hanya operasi vektor tingkat rendah yang harus tahu tentang representasi numeriknya. Vektor akan direpresentasikan sebagai karakter tebal, mis., a. Panjang suatu vektor dilambangkan ||a||.Vektor satuan adalahvektor yang panjangnya adalah satu. Vektor nol adalah vektor dengan panjang nol. Arah vektor nol tidak terdefinisi.

Vektor dapat digunakan untuk mewakili banyak hal yang berbeda. Misalnya, mereka dapat digunakan untuk menyimpan *offset*, juga disebut *displacement*. Jika kita tahu "harta terkubur dua langkah ke timur dan tiga langkah ke utara dari tempat pertemuan rahasia," maka kita tahu *offset*nya, tapi kita tidak tahu harus mulai dari mana. Vektor juga dapat digunakan untuk menyimpan lokasi, kata lain untuk posisi atau titik. Lokasi dapat direpresentasikan sebagai perpindahan dari lokasi lain. Biasanya ada beberapa lokasi asal yang dipahami dari mana semua lokasi lain disimpan sebagai offset. Perhatikan bahwa lokasi bukan vektor. Seperti yang akan kita bahas, Anda dapat menambahkan dua vektor. Namun, biasanya tidak masuk akal untuk menambahkan dua lokasi kecuali itu adalah operasi perantara ketika menghitung rata-rata tertimbang dari suatu lokasi. Menambahkan dua *offset* memang masuk akal, jadi itulah salah satu alasan mengapa *offset* adalah vektor. Tapi ini menekankan bahwa lokasi bukanlah *offset*; itu adalah *offset* dari lokasi asal tertentu. *Offset* dengan sendirinya bukanlah lokasi.



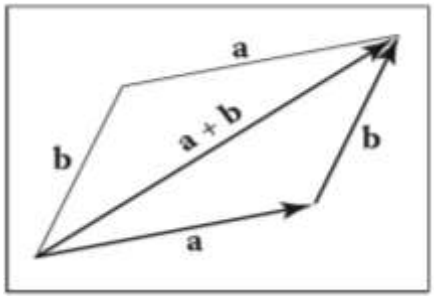
Gambar 2.11 Kedua vektor ini sama karena memiliki panjang dan arah yang sama

Operasi Vektor

Vektor memiliki sebagian besar operasi aritmatika biasa yang kita kaitkan dengan bilangan real. Dua vektor sama jika dan hanya jika mereka memiliki panjang dan arah yang sama. Dua vektor ditambahkan sesuai dengan aturan jajaran genjang. Aturan ini menyatakan bahwa jumlah dua vektor diperoleh dengan menempatkan salah satu ekor vektor terhadap kepala yang lain (Gambar 2.12). Jumlah vektor adalah vektor yang "menyelesaikan segitiga" dimulai oleh dua vektor. Jajar genjang dibentuk dengan mengambil jumlah di kedua urutan. Hal ini menekankan bahwa penjumlahan vektor bersifat komutatif:

$$\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}$$

Perhatikan bahwa aturan jajaran genjang hanya memformalkan intuisi kita tentang perpindahan. Pikirkan berjalan di sepanjang satu vektor, ekor ke kepala, dan kemudian berjalan di sepanjang yang lain.



Gambar 2.12 Dua vektor ditambahkan dengan mengaturnya dari kepala ke ekor. Ini dapat dilakukan dalam urutan apa pun.

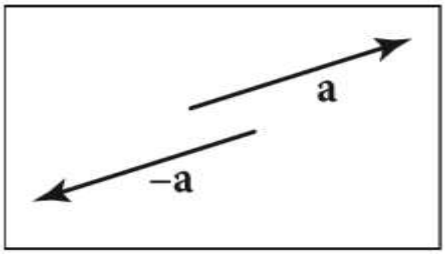
Net Displacement hanyalah diagonal jajar genjang. Anda juga dapat membuat minus unary untuk sebuah vektor: a (Gambar 2.13) adalah vektor dengan panjang yang sama dengan a tetapi arahnya berlawanan. Hal ini memungkinkan kita untuk juga mendefinisikan pengurangan:

$$\boldsymbol{b} - \boldsymbol{a} \equiv \boldsymbol{-a} + \boldsymbol{b}$$

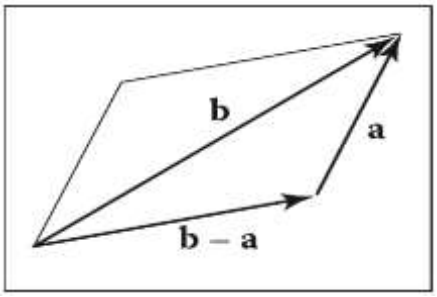
Anda dapat memvisualisasikan pengurangan vektor dengan jajaran genjang (Gambar 2.14). Kita bisa menulis

$$\boldsymbol{a} + (\boldsymbol{b} - \boldsymbol{a}) = \boldsymbol{b}$$

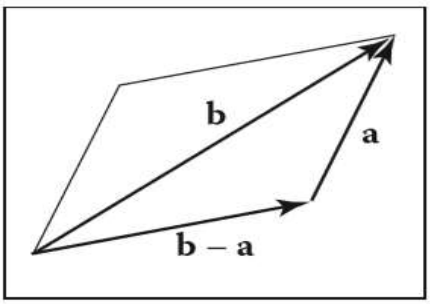
Vektor juga bisa dikalikan. Sebenarnya, ada beberapa jenis produk yang melibatkan vektor. Pertama, kita dapat menskalakan vektor dengan mengalikannya dengan bilangan real k. Ini hanya mengalikan panjang vektor tanpa mengubah arahnya. Misalnya, 3,5a adalah vektor yang arahnya sama dengan a tetapi panjangnya 3,5 kali a. Kami membahas dua produk yang melibatkan dua vektor, produk titik dan perkalian silang, nanti di bagian ini, dan produk yang melibatkan tiga vektor, determinan, di Bab 5.



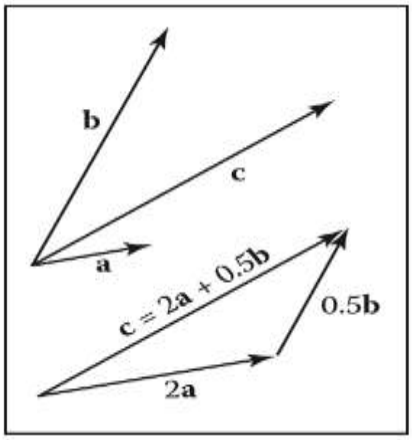
Gambar 2.13 Vektor –a memiliki panjang yang sama tetapi berlawanan arah dgn vektor a.



Gambar 2.14 Pengurangan vektor hanyalah penjumlahan vektor dengan pembalikan argumen kedua.



Gambar 2.15 Setiap vektor 2D c adalah jumlah berbobot dari dua vektor 2D tak sejajar a & b.



Gambar 2.16 Dasar Cartesian 2D untuk vektor.

Koordinat Cartesian dari Vektor

Sebuah vektor 2D dapat ditulis sebagai kombinasi dari dua vektor bukan nol yang tidak sejajar. Sifat kedua vektor ini disebut *independensi linier*. Dua vektor bebas linier membentuk basis 2D, dan vektor-vektor tersebut disebut sebagai vektor basis. Sebagai contoh, sebuah vektor c dapat dinyatakan sebagai kombinasi dari dua vektor basis a dan b (Gambar 2.15):

$$\boldsymbol{C} = \boldsymbol{a_c a} + \boldsymbol{b_c b}$$

Perhatikan bahwa bobot ac dan bc adalah unik. Basa sangat berguna jika kedua vektor ortogonal, yaitu, mereka tegak lurus satu sama lain. Bahkan lebih berguna jika mereka juga vektor satuan dalam hal ini mereka ortonormal. Jika kita menganggap dua vektor "khusus" x dan y diketahui oleh kita, maka kita dapat menggunakannya untuk mewakili semua vektor lain dalam sistem koordinat Kartesius, di mana setiap vektor direpresentasikan sebagai dua bilangan real. Sebagai contoh, sebuah vektor a dapat direpresentasikan sebagai

$$a = x_a x + y_a y$$

dimana x_a dan y_a adalah koordinat Cartesian nyata dari vektor 2D a (Gambar 2.16). Perhatikan bahwa ini secara konseptual tidak berbeda dengan Persamaan (2.3), di mana vektor basisnya tidak ortonormal. Tetapi ada beberapa keuntungan dari sistem koordinat Cartesian. Misalnya, dengan teorema Pythagoras, panjang a adalah

$$||a|| = \sqrt{x_2^a + y_2^a}$$

Menghitung perkalian titik, perkalian silang, dan koordinat vektor dalam sistem Cartesian juga mudah, seperti yang akan kita lihat di bagian berikut. Dengan konvensi kita menulis koordinat baik sebagai pasangan terurut (x_a,y_a) atau matriks kolom:

$$a = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$$

Bentuk yang kita gunakan akan tergantung pada kenyamanan tipografi. Kami juga kadang-kadang akan menulis vektor sebagai matriks baris, yang akan kami tunjukkan sebagai a^T:

$$a^T = [x_a \quad y_a]$$

Kami juga dapat mewakili 3D, 4D, dll, vektor dalam koordinat Cartesian. Untuk kasus 3D, kami menggunakan vektor basis z yang ortogonal untuk x dan y

Produk Dot

Cara paling sederhana untuk mengalikan dua vektor adalah produk titik. Hasil kali titik dari a dan b dilambangkan dengan a·b dan sering disebut hasil kali skalar karena menghasilkan skalar. Produk titik mengembalikan nilai yang terkait dengan panjang argumennya dan sudut di antara keduanya (Gambar 2.17):

$$\boldsymbol{a \,.\, b = ||a||\ ||b||\ cos\ \emptyset}$$

Penggunaan produk titik yang paling umum dalam program grafis adalah untuk menghitung kosinus sudut antara dua vektor. Produk titik juga dapat digunakan untuk menemukan proyeksi satu vektor ke vektor lainnya. Ini adalah panjang a→b dari vektor yang diproyeksikan tegak lurus pada vektor b (Gambar 2.18):

$$\boldsymbol{a \rightarrow b = ||a||\ cos\ \emptyset = a \,.\, b\ /\ ||b||}$$

Produk titik mematuhi sifat asosiatif dan distributif yang kita miliki dalam aritmatika nyata:

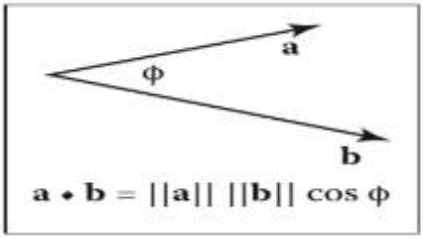
$$\boldsymbol{a{\cdot}b = b{\cdot}a,}$$
$$\boldsymbol{a{\cdot}(b + c) = a{\cdot}b + a{\cdot}c,}$$
$$\boldsymbol{(ka){\cdot}b = a{\cdot}(kb) = ka{\cdot}b.}$$

Jika vektor 2D a dan b dinyatakan dalam koordinat Cartesian, kita dapat memanfaatkan x · x = y · y =1 dan x · y = 0 untuk memperoleh hasil kali titiknya adalah

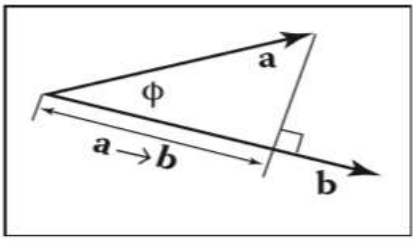
$$\begin{aligned} \boldsymbol{a{\cdot}b} &= \boldsymbol{(x_a x + y_a y){\cdot}(x_b x + y_b y)} \\ &= \boldsymbol{x_a x_b(x{\cdot}x) + x_a y_b(x{\cdot}y) + x_b y_a(y{\cdot}x) + y_a y_b(y{\cdot}y)} \\ &= \boldsymbol{x_a x_b + y_a y_b.} \end{aligned}$$

Demikian pula dalam 3D kita dapat menemukan

$$\boldsymbol{a{\cdot}b = x_a x_b + y_a y_b + z_a z_b.}$$



Gambar 2.17 Produk titik terkait dengan panjang dan sudut dan merupakan salah satu rumus terpenting dalam grafis.



Gambar 2.18 Proyeksi a ke b adalah panjang yang ditemukan oleh Persamaan (2.5).

1.1.1. Perkalian Silang

Perkalian silang a×b biasanya hanya digunakan untuk vektor tiga dimensi; perkalian silang umumdibahas dalam referensi yang diberikan dalam catatan bab. Perkalian silang mengembalikan vektor 3D yang tegak lurus terhadap dua argumen dari perkalian silang. Panjang vektor yang dihasilkan berhubungan dengan sin Ø:

$$||a \times b|| = ||a||\ ||b||\ \sin \emptyset$$

Besarnya ||a×b|| sama dengan luas jajar genjang yang dibentuk oleh vektor a dan b. Selain itu, a×b tegak lurus terhadap a dan b (Gambar 2.19). Perhatikan bahwa hanya ada dua kemungkinan arah untuk vektor semacam itu. Secara definisi, vektor-vektor pada arah sumbu x-, y- dan z diberikan oleh

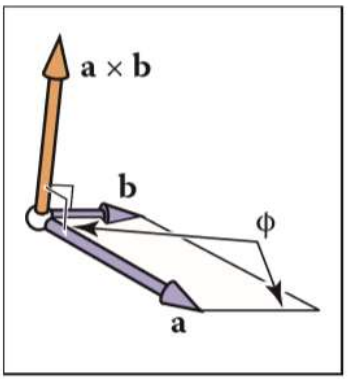
$$x = (1, 0, 0),$$
$$y = (0, 1, 0),$$
$$z = (0, 0, 1),$$

dan kami menetapkan sebagai konvensi bahwa x×y harus dalam arah plus atau minus z. Pilihannya agak sewenang-wenang, tetapi standar untuk mengasumsikan bahwa

$$z = x \times y$$

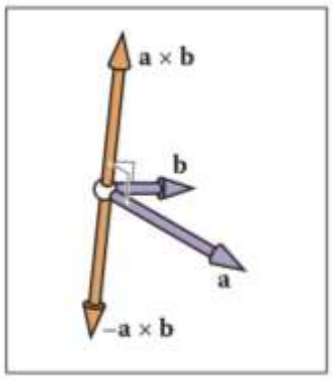
Semua permutasi yang mungkin dari ketiga vektor satuan kartesius adalah

$$x \times y = +z,$$
$$y \times x = -z,$$
$$y \times z = +x,$$
$$z \times y = -x,$$
$$z \times x = +y,$$
$$x \times z = -y.$$



Gambar 2.19 Perkalian silang a × b adalah vektor 3D yang tegak lurus terhadap kedua vektor 3D a dan b, dan panjangnya sama dengan luas jajaran genjang yang ditunjukkan.

Karena sifat sinØ, kita juga mengetahui bahwa vektor silang itu sendiri adalah vektor nol, jadi x × x = 0 dan seterusnya. Perhatikan bahwa perkalian silang tidak komutatif, yaitu x×y ≠ y×x. Pengamat yang cermat akan mencatat bahwa diskusi di atas tidak memungkinkan kita untuk menggambar gambaran yang jelas tentang bagaimana sumbu Cartesian berhubungan. Lebih khusus lagi, jika kita meletakkan x dan y di trotoar, dengan x menunjuk ke timur dan y menunjuk ke utara, maka apakah z menunjuk ke langit atau ke tanah? Konvensi yang biasa adalah memiliki titik z ke langit. Ini dikenal sebagai sistem koordinat tangan kanan. Nama ini berasal dari skema memori "meraih" x dengan telapak tangan dan jari kanan Anda dan memutarnya ke arah y. Vektor z harus sejajar dengan ibu jari Anda. Hal ini diilustrasikan pada Gambar 2.20.



Gambar 2.20 "Aturan tangan kanan" untuk perkalian silang. Bayangkan menempatkan pangkal telapak tangan kanan Anda di mana a dan b bergabung di ekornya, dan mendorong panah a ke arah b. Jempol kanan Anda yang diperpanjang harus mengarah ke a×b.

Perkalian silang memiliki properti bagus yang

$$a \times (b + c) = a \times b + a \times c,$$

dan

$$a \times (kb) = k(a \times b).$$

Namun, konsekuensi dari aturan tangan kanan adalah

$$a \times b = -(b \times a).$$

Dalam koordinat Cartesian, kita dapat menggunakan ekspansi eksplisit untuk menghitung perkalian silang:

$$\begin{aligned} a \times b &= (x_a x + y_a y + z_a z) \times (x_b x + y_b y + z_b z) \\ &= x_a x_b x \times x + x_a y_b x \times y + x_a z_b x \times z + y_a x_b y \times x + y_a y_b y \times y + y_a z_b y \times z + z_a x_b z \times x + z_a y_b z \times y + z_a z_b z \times z \\ &= (y_a z_b - z_a y_b) x + (z_a x_b - x_a z_b) y + (x_a y_b - y_a x_b) z. \end{aligned}$$

Jadi, dalam bentuk koordinat,

$$a \times b = (y_a z_b - z_a y_b,\ z_a x_b - x_a z_b,\ x_a y_b - y_a x_b).$$

1.1.2. Dasar Ortonormal dan Bingkai Koordinat

Mengelola sistem koordinat adalah salah satu tugas inti dari hampir semua program grafis; kuncinya adalah mengelola basis ortonormal. Setiap himpunan dari dua vektor 2D u dan v membentuk basis ortonormal asalkan keduanya ortogonal (bersudut siku-siku) dan masing-masing memiliki panjang satuan. Jadi,

$$||u|| = ||v|| = 1$$

dan

$$u \cdot v = 0.$$

Dalam 3D, tiga vektor u, v, dan w membentuk basis ortonormal jika

$$||u|| = ||v|| = ||w|| = 1$$

dan

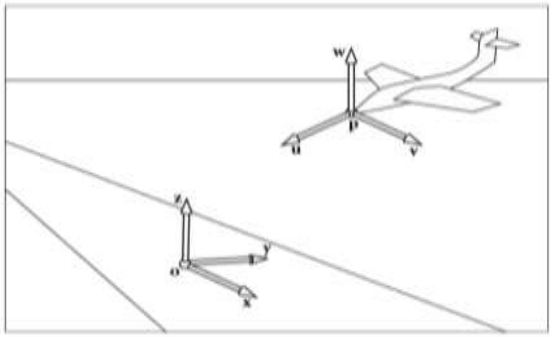
$$u \cdot v = v \cdot w = w \cdot u = 0$$

Dasar ortonormal ini adalah tangan kanan asalkan

$$w = u \times v,$$

dan selain itu kidal.

Perhatikan bahwa basis ortonormal kanonik Cartesian hanyalah salah satu dari tak terhingga banyak basis ortonormal yang mungkin. Apa yang membuatnya istimewa adalah bahwa ia dan lokasi asal implisitnya digunakan untuk representasi tingkat rendah dalam suatu program. Jadi, vektor x, y, dan z tidak pernah disimpan secara eksplisit dan juga bukan lokasi asal kanonik o. Model global biasanya disimpan dalam sistem koordinat kanonik ini, dan karena itu sering disebut sistem koordinat global. Namun, jika kita ingin menggunakan sistem koordinat lain dengan asal p dan vektor basis ortonormal u, v, dan w, maka kita menyimpan vektor-vektor tersebut secara eksplisit. Sistem seperti ini disebut kerangka acuan atau kerangka koordinat. Misalnya, dalam simulator penerbangan, kita mungkin ingin mempertahankan sistem koordinat dengan titik asal di hidung pesawat, dan basis ortonormal sejajar dengan pesawat. Secara bersamaan, kita akan memiliki sistem koordinat kanonik utama (Gambar 2.21). Sistem koordinat yang terkait dengan objek tertentu, seperti bidang, biasanya disebut sistem koordinat lokal.



Gambar 2.21 Selalu ada sistem koordinat master atau "kanonik" dengan asal dan basis ortonormal x, y, dan z. Sistem koordinat ini biasanya didefinisikan untuk disejajarkan dengan model global dan dengan demikian sering disebut sistem koordinat "global" atau "dunia". Vektor asal dan basis ini tidak pernah disimpan secara eksplisit. Semua vektor dan lokasi lainnya disimpan dengan koordinat yang menghubungkannya dengan kerangka global. Sistem koordinat yang terkait dengan bidang secara eksplisit disimpan dalam bentuk koordinat global.

Pada level rendah, bingkai lokal disimpan dalam koordinat kanonik. Misalnya, jika u memiliki koordinat (x_u, y_u, z_u),

$$u = x_u x + y_u y + z_u z.$$

Lokasi secara implisit menyertakan offset dari asal kanonik:

$$p = o + x_p x + y_p y + z_p z,$$

di mana (x_p,y_p,z_p) adalah koordinat p.

Perhatikan bahwa jika kita menyimpan sebuah vektor a terhadap kerangka u-v-w, kita menyimpan sebuah rangkap tiga (u_a,v_a,w_a) yang dapat kita tafsirkan secara geometris sebagai

$$a = u_a u + v_a v + w_a w.$$

Untuk mendapatkan koordinat kanonik dari vektor a yang disimpan dalam sistem koordinat uvw, cukup ingat bahwa u, v, dan w sendiri disimpan dalam koordinat Cartesian, sehingga ekspresi $u_a u + v_a v + w_a w$ sudah dalam koordinat Cartesian jika dievaluasi secara eksplisit . Untuk mendapatkan koordinat u-v-w dari vektorb yang disimpan dalam sistem koordinat kanonik, kita dapat menggunakan produk titik:

$$u_b = u \cdot b;\ v_b = v \cdot b;\ w_b = w \cdot b.$$

Ini berfungsi karena kita tahu bahwa untuk beberapa u_b, v_b, dan w_b,

$$u_b u + v_b v + w_b w = b,$$

dan produk titik mengisolasi koordinat u_b:

$$u \cdot b = u_b(u \cdot u) + v_b(u \cdot v) + w_b(u \cdot w) = u_b.$$

Ini bekerja karena u, v, andw ortonormal. Menggunakan matriks untuk mengelola perubahan sistem koordinat dibahas dalam Bagian 6.2.1 dan 6.5.

1.1.3. Membangun Basis dari Vektor Tunggal

Seringkali kita membutuhkan basis ortonormal yang disejajarkan dengan vektor yang diberikan. Artinya, diberikan vektor a, kami menginginkan ortonormal u, v, dan w sedemikian rupa sehingga w menunjuk ke arah yang sama dengan a (Hughes& Moller, 1999), tetapi kami tidak terlalu peduli apa u dan v itu. Satu vektor tidak cukup untuk menentukan jawaban secara unik; kita hanya perlu prosedur kuat yang akan menemukan salah satu basis yang mungkin.
Ini dapat dilakukan dengan menggunakan perkalian silang sebagai berikut. Pertama-tama buatlah w sebagai vektor satuan dengan arah a:

$$w = \frac{a}{||a||}$$

Kemudian pilih salah satu vektor t yang tidak kolinear dengan w, dan gunakan perkalian silang untuk membuat vektor satuan u tegak lurus terhadap w:

$$u = \frac{t\ x\ w}{||t\ x\ w||}$$

Catatan : Prosedur yang sama ini tentu saja dapat digunakan untuk membangun tiga vektor dalam urutan apa pun; hanya memperhatikan urutan perkalian silang untuk memastikan dasar tangan kanan.

Jika t kolinear dengan w penyebut akan hilang, dan jika hampir kolinear hasilnya akan memiliki presisi rendah. Prosedur sederhana untuk menemukan vektor yang cukup berbeda dari w adalah memulai dengan t sama dengan w dan mengubah komponen besaran terkecil dari t menjadi 1. Misalnya, jika $t = (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0)$ maka $t = (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 1)$.Setelah Anda dan Anda berada di tangan, menyelesaikan dasarnya sederhana:

$$v = w \times u$$

Contoh situasi di mana konstruksi ini digunakan adalah naungan permukaan, di mana diperlukan dasar yang sejajar dengan permukaan normal tetapi rotasi di sekitar normal sering tidak penting.

1.1.4. Membangun Basis dari Dua Vektor

Prosedur di bagian sebelumnya juga dapat digunakan dalam situasi di mana rotasi basis di sekitar vektor yang diberikan adalah penting. Contoh umum adalah membangun basis untuk kamera: penting untuk memiliki satu vektor yang disejajarkan dengan arah yang dilihat kamera, tetapi orientasi kamera di sekitar vektor itu tidak sembarang, dan entah bagaimana harus ditentukan. Setelah orientasi ditentukan, dasarnya ditentukan sepenuhnya.

Cara umum untuk menspesifikasikan frame secara penuh adalah dengan menyediakan dua vektor a (yang menspesifikasikan w) dan b (yang menspesifikasikan v). Jika kedua vektor diketahui tegak lurus, maka vektor ketiga dapat dibuat dengan mudah dengan u = b×a.

Catatan : u = a×b juga menghasilkan basis ortonormal, tetapi merupakan tangan kiri.
Untuk memastikan bahwa basis yang dihasilkan benar-benar ortonormal, bahkan jika vektor inputnya tidak cukup, disarankan untuk menggunakan prosedur seperti prosedur vektor tunggal:

$$w = \frac{a}{||a||}$$

$$u = \frac{b \times y}{||b \times w||}$$

$$v = w \times u$$

Sebenarnya, prosedur ini bekerja dengan baik ketika a dan b tidak tegak lurus. Dalam hal ini, w akan dibangun tepat pada arah a, dan v dipilih sebagai vektor terdekat ke b di antara semua vektor yang tegak lurus terhadap w.

Catatan : Jika Anda ingin metoset tongkat v ke dua arah yang tidak tegak lurus, sesuatu harus diberikan—dengan skema ini saya akan mengatur semuanya sesuai keinginan Anda, kecuali saya akan membuat perubahan terkecil ke v sehingga sebenarnya tegak lurus terhadap w. Apa yang akan salah dengan perhitungan jika a dan b paralel?

Prosedur ini tidak akan bekerja jika a dan b adalah collinear. Dalam hal ini b tidak membantu dalam memilih arah mana yang tegak lurus terhadap a yang harus kita gunakan: arahnya tegak lurus terhadap semuanya.

Dalam contoh penentuan posisi kamera (Bagian 4.3), kami ingin membuat bingkai yang memiliki w sejajar dengan arah yang dilihat kamera, dan v harus menunjuk ke bagian atas kamera. Untuk mengarahkan kamera ke atas, kami membangun basis di sekitar arah tampilan, menggunakan arah lurus ke atas sebagai vektor referensi untuk menetapkan orientasi kamera di sekitar arah tampilan. Menyetel v sedekat mungkin ke lurus sama persis dengan gagasan intuitif "memegang kamera lurus".

1.1.5. Mengkuadratkan Basis

Kadang-kadang Anda mungkin menemukan masalah yang disebabkan dalam perhitungan Anda dengan basis yang dianggap ortonormal tetapi di mana kesalahan telah merayap—karena kesalahan pembulatan dalam perhitungan, atau karena basis telah disimpan dalam file dengan presisi rendah, misalnya.

Prosedur bagian sebelumnya dapat digunakan; hanya membangun basis baru menggunakan vektor w dan v yang ada akan menghasilkan basis baru yang ortonormal dan dekat dengan yang lama.

Pendekatan ini bagus untuk banyak aplikasi, tetapi ini bukan yang terbaik yang tersedia. Ini menghasilkan vektor ortogonal yang akurat, dan untuk basis awal yang hampir ortogonal, hasilnya tidak akan menyimpang jauh dari titik awal. Namun, itu asimetris: "mendukung" w atas v dan v atas u (yang nilai awalnya dibuang). Ia memilih basis yang dekat dengan basis awal tetapi tidak memiliki jaminan untuk memilih basis ortonormal terdekat. Jika ini tidak cukup baik, SVD (Bagian 5.4.1) dapat digunakan untuk menghitung basis ortonormal yang dijamin paling dekat dengan basis aslinya.

2.5 KURVA DAN DATAR

Geometri kurva, dan terutama permukaan, memainkan peran sentral dalam grafis, dan di sini kita meninjau dasar-dasar kurva dan permukaan dalam ruang 2D dan 3D.

1.1.6. Kurva Implisit 2D

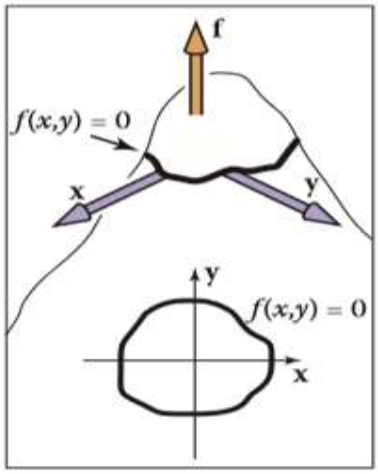
Secara intuitif, kurva adalah sekumpulan titik yang dapat digambar pada selembar kertas tanpa mengangkat pena. Cara umum untuk menggambarkan kurva adalah menggunakan persamaan implisit. Persamaan implisit dalam dua dimensi memiliki bentuk

$$f(x, y) = 0$$

Fungsi f(x, y) mengembalikan nilai riil. Titik (x, y) yang nilainya nol berada pada kurva, dan titik yang nilainya bukan nol tidak berada pada kurva. Misalkan f(x, y) adalah

$$f(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2$$

di mana (x_c, y_c) adalah titik 2D dan r adalah bilangan bukan nol. Jika kita ambil f(x, y)= 0, titik-titik di mana persamaan ini berlaku adalah pada lingkaran dengan pusat(x_c, y_c) dan jari-jari r. Alasan mengapa persamaan ini disebut persamaan “implisit” adalah karena titik (x, y) pada kurva tidak dapat langsung dihitung dari persamaan dan sebaliknya harus ditentukan dengan menyelesaikan persamaan. Dengan demikian, titik-titik pada kurva tidak dihasilkan oleh persamaan secara eksplisit, tetapi mereka terkubur di suatu tempat secara implisit dalam persamaan.



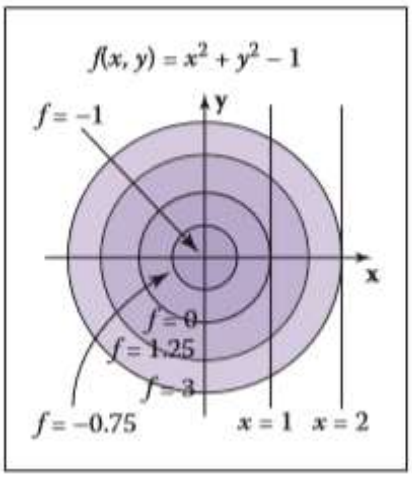
Gambar 2.22 Fungsi implisit f(x,y) = 0 dapat dianggap sebagai bidang ketinggian di mana f adalah tinggi (atas). Jalur di mana tingginya nol adalah kurva implisit (bawah).

Sangat menarik untuk dicatat bahwa f memang memiliki nilai untuk semua (x, y). Kita dapat menganggapnya sebagai medan, dengan permukaan laut pada f = 0 (Gambar2.22). Pantai adalah kurva implisit. Nilai f adalah ketinggian. Hal lain yang perlu diperhatikan adalah bahwa kurva mempartisi ruang menjadi daerah-daerah di mana f > 0, f < 0, dan f = 0. Jadi Anda mengevaluasi f untuk memutuskan apakah suatu titik "di dalam" kurva. Perhatikan bahwa f(x,

y) = c adalah kurva untuk setiap konstanta c, dan c =0 hanya digunakan sebagai konvensi. Misalnya, jika f(x, y) = $x^2 + y^2 - 1$, memvariasikan c hanya memberikan variasi lingkaran yang berpusat di titik asal (Gambar 2.23). Kita dapat memampatkan notasi kita menggunakan vektor. Jika kita mempunyai c = (x_c, y_c) dan p = (x, y), maka lingkaran kita dengan pusat c dan jari-jari r didefinisikan oleh vektor-vektor posisi yang memenuhi

$$(p - c) \cdot (p - c) - r^2 = 0 .$$

Persamaan ini, jika diperluas secara aljabar, akan menghasilkan Persamaan (2.9), tetapi lebih mudah untuk melihat bahwa ini adalah persamaan untuk lingkaran dengan "membaca" persamaan secara geometris. Bunyinya, "titik p pada lingkaran memiliki properti berikut: vektor dari c ke p ketika bertitik dengan dirinya sendiri memiliki nilai r^2." Karena sebuah vektor bertitik dengan dirinya sendiri hanya kuadrat panjangnya sendiri, kita juga dapat membaca persamaan sebagai, "titik p pada lingkaran memiliki sifat berikut: vektor dari c ke p memiliki panjang kuadrat r2."



Gambar 2.23 Fungsi implisit f(x,y) = 0 dapat dianggap sebagai bidang ketinggian di mana f adalah tinggi (atas). Jalur di mana tingginya nol adalah kurva implisit (bawah).

Bahkan lebih baik, adalah untuk mengamati bahwa panjang kuadrat hanyalah jarak kuadrat dari c ke p, yang menunjukkan bentuk yang setara

$$||p - c||^2 - r^2 = 0$$

dan, tentu saja, ini menyarankan

$$||p - c|| - r = 0$$

Di atas dapat dibaca "titik p pada lingkaran adalah yang berjarak r dari titik pusat c", yang merupakan definisi lingkaran yang baik. Ini mengilustrasikan bahwa bentuk vektor dari suatu persamaan sering kali menunjukkan lebih banyak geometri dan intuisi daripada bentuk Cartesian yang setara dengan x dan y. Untuk alasan ini, biasanya disarankan untuk menggunakan bentuk vektor jika memungkinkan. Selain itu, Anda dapat mendukung kelas vektor dalam kode Anda; kode lebih bersih ketika bentuk vektor digunakan. Persamaan berorientasi vektor juga kurang rawan kesalahan dalam implementasi: setelah Anda mengimplementasikan dan men-debug jenis vektor dalam kode Anda, kesalahan potong dan tempel yang melibatkan x, y, dan z akan hilang. Dibutuhkan sedikit waktu untuk membiasakan diri dengan vektor dalam persamaan ini, tetapi setelah Anda menguasainya, hasilnya besar.

1.1.7. Gradien2D

Jika kita menganggap fungsi f(x, y) sebagai bidang ketinggian dengan tinggi = f(x, y), vektor gradien menunjuk ke arah kemiringan maksimum, yaitu lurus menanjak. Vektor gradien ∇f(x, y) diberikan oleh

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

Vektor gradien yang dievaluasi pada suatu titik pada kurva implisit f(x, y)=0 tegak lurus terhadap vektor tangen kurva pada titik tersebut. Vektor tegak lurus ini biasanya disebut vektor normal terhadap kurva. Selain itu, karena gradiennya menanjak, ini menunjukkan arah wilayah f(x, y) > 0.

Dalam konteks medan ketinggian, makna geometris turunan parsial dan gradien lebih terlihat dari biasanya. Misalkan di dekat titik (a, b), f(x, y) adalah sebuah bidang (Gambar 2.24). Ada arah menanjak dan menurun yang spesifik. Di sudut kanan ke arah ini adalah arah yang datar terhadap pesawat. Setiap perpotongan antara bidang dan bidang f(x, y) = 0 akan berada pada arah yang datar. Dengan demikian arah menanjak/menurun akan tegak lurus dengan garis perpotongan f(x,y)=0. Untuk melihat mengapa turunan parsial ada hubungannya dengan ini, kita perlu memvisualisasikan makna geometrisnya. Ingat bahwa turunan konvensional dari fungsi 1D y = g(x) adalah

$$\frac{dy}{dx} \equiv \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(x+\Delta x) - g(x)}{\Delta x}$$

Ini mengukur kemiringan garis singgung ke g (Gambar2.25). Turunan parsial adalah generalisasi dari turunan 1D. Untuk fungsi 2D f(x,y), kita tidak dapat mengambil limit yang sama untuk x seperti pada Persamaan (2.10), karena f dapat berubah dalam banyak cara untuk perubahan x yang diberikan. Namun, jika kita menganggap y konstan, kita dapat mendefinisikan analog dari turunan, yang disebut turunan parsial (Gambar 2.26):

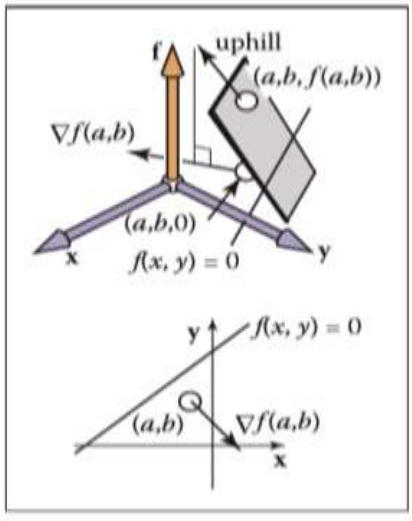
$$\frac{\partial f}{\partial x} \lim_{\Delta x \to 0} \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x}$$

Mengapa turunan parsial terhadap x dan y merupakan komponen vektor gradien? Sekali lagi, ada wawasan yang lebih jelas dalam geometri daripada di aljabar. Pada Gambar 2.27, kita melihat vektor a bergerak sepanjang jalur di mana f tidak berubah. Perhatikan bahwa ini sekali lagi pada skala yang cukup kecil sehingga tinggi permukaan (x, y) = f(x, y) dapat dianggap planar lokal. Dari gambar tersebut, kita melihat bahwa vektor a = (Δx, Δy). Karena arah menanjak tegak lurus terhadap a, kita tahu perkalian titik sama dengan nol:

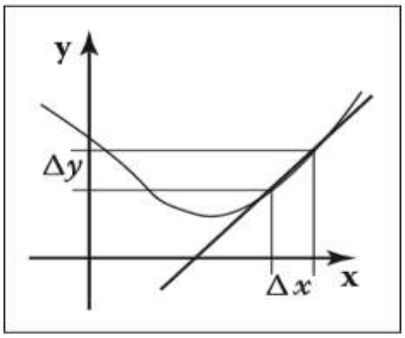
$$(\nabla f) \cdot a \equiv (x_\nabla, y_\nabla) \cdot (x_a, y_a) = x_\nabla \Delta x + y_\nabla \Delta y = 0 .$$

Kita juga tahu bahwa perubahan dalam arah f (xa,ya) sama dengan nol:

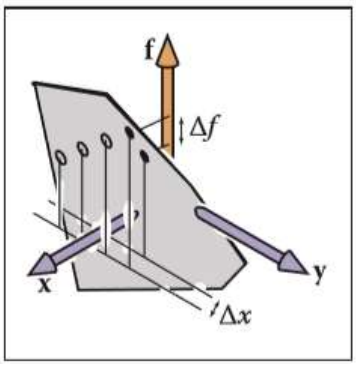
$$\Delta f = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y \equiv \frac{\partial f}{\partial x} x_a + \frac{\partial f}{\partial y} y_a = 0$$



Gambar 2.24 Tinggi permukaan = f(x, y) secara lokal planar dekat (x ,y) = (a, b). Gradien adalah proyeksi arah menanjak ke bidang ketinggian = 0.



Gambar 2.25 Turunan dari fungsi 1D mengukur kemiringan garis yang bersinggungan dengan kurva.



Gambar 2.26 Turunan parsial dari suatu fungsi f terhadap x harus memiliki konstanta y untuk memiliki nilai unik, seperti yang ditunjukkan oleh titik gelap. Titik-titik berongga menunjukkan nilai-nilai lain dari f yang tidak membuat y konstan.

Mengingat vektor (x, y) dan (x, y) yang tegak lurus, kita tahu bahwa sudut di antara mereka adalah 90 derajat, dan dengan demikian produk titik mereka sama dengan nol (ingat bahwa produk titik sebanding dengan kosinus dari sudut antara dua vektor). Jadi, kita memiliki xx' + yy' = 0. Diberikan (x, y), mudah untuk membangun vektor-vektor valid yang perkalian titiknya dengan (x, y) sama dengan nol, dua yang paling jelas adalah (y, −x) dan (−y, x); Anda dapat memverifikasi bahwa vektor-vektor ini memberikan hasil kali titik nol yang diinginkan dengan (x, y). Generalisasi dari pengamatan ini adalah bahwa (x, y) tegak lurus terhadap k(y, −x) di mana k adalah sembarang konstanta bukan nol. Ini menyiratkan bahwa

$$(x_a, y_a) = k\left(\frac{\partial f}{\partial y}, -\frac{\partial f}{\partial x}\right)$$

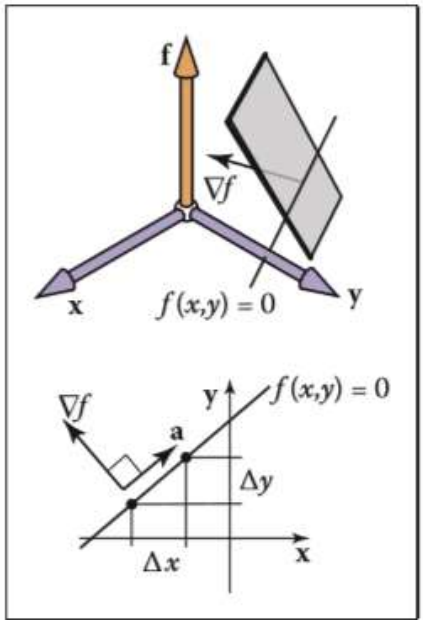
Menggabungkan Persamaan (2. 11) dan (2.12) memberikan

$$(x\nabla, \mathrm{y}\nabla) = \mathrm{k}'\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

di mana k' adalah sembarang konstanta bukan nol. Menurut definisi, "menanjak" menyiratkan perubahan positif pada f, jadi kita ingin k' > 0, dan k' =1 adalah konvensi yang sangat baik.

Sebagai contoh gradien, perhatikan lingkaran implisit $x^2 + y^2 - 1 = 0$ dengan vektor gradien (2x, 2y), yang menunjukkan bahwa bagian luar lingkaran adalah daerah positif untuk fungsi $f(x, y) = x^2 + y^2 - 1$. Perhatikan bahwa panjang vektor gradien dapat berbeda tergantung pada pengali dalam persamaan implisit. Misalnya, lingkaran satuan dapat digambarkan dengan $Ax^2 + Ay^2 - A = 0$ untuk sembarang bukan nol A. Gradien untuk kurva ini adalah (2Ax, 2Ay). Ini akan normal (tegak lurus) terhadap lingkaran, tetapi akan memiliki panjang yang

ditentukan oleh A. Untuk A > 0, normal akan mengarah ke luar dari lingkaran, dan untuk A < 0, akan mengarah ke dalam. Peralihan dari luar ke dalam ini sebagaimana mestinya, karena daerah positif beralih di dalam lingkaran. Jarak pandang tinggi-bidang, $h = Ax^2 + Ay^2 - A$, dan lingkaran berada pada ketinggian nol. Untuk A > 0, lingkaran menutupi depresi, dan untuk A < 0, lingkaran menutupi benjolan. Saat A menjadi lebih negatif, tonjolan bertambah tinggi, tetapi lingkaran h = 0 tidak berubah. Arah menanjak maksimum tidak berubah, tetapi kemiringan meningkat. Panjang gradien mencerminkan perubahan derajat kemiringan ini. Jadi secara intuitif, Anda dapat menganggap arah gradien sebagai menunjuk ke atas dan besarnya sebagai mengukur seberapa menanjak lereng itu.



Gambar 2.27 Vektor a menunjuk ke arah di mana f tidak berubah dan dengan demikian tegak lurus terhadap vektor gradien ∇f.

2.53. Garis Implisit 2D

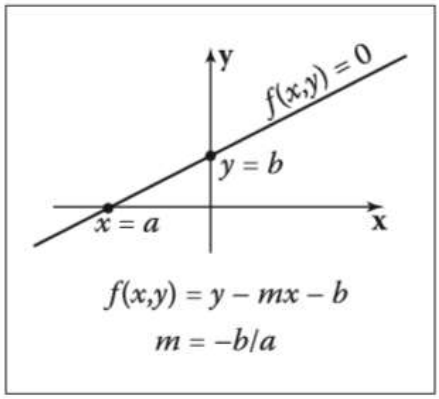
Bentuk garis "slope-intercept" yang familiar adalah

$$y \equiv mx + b$$

Ini dapat dengan mudah dikonversi ke bentuk implisit (Gambar 2.28):

$$y - mx - b = 0$$

Di sini m adalah "kemiringan" (rasio naik ke run) dan b adalah nilai y di mana garis memotong sumbu y, biasanya disebut perpotongan y. Garis juga mempartisi bidang 2D, tetapi di sini "di dalam" dan "di luar" mungkin lebih intuitif disebut "atas" dan "bawah".



Gambar 2.28 Garis 2D dapat digambarkan dengan persamaan *y – mx − b = 0.*

Karena kita dapat mengalikan persamaan implisit dengan konstanta apa pun tanpa mengubah titik di mana ia adalah nol, *kf(x, y) = 0* adalah kurva yang sama untuk sembarang k yang tidak nol. Ini memungkinkan beberapa bentuk implisit untuk baris yang sama, misalnya,

$$2y – 2mx – 2b = 0$$

Salah satu alasan bentuk perpotongan kemiringan terkadang canggung adalah karena ia tidak dapat mewakili beberapa garis seperti x = 0 karena m harus tak hingga. Untuk alasan ini, bentuk yang lebih umum sering kali berguna:

$$Ax + By + C = 0$$

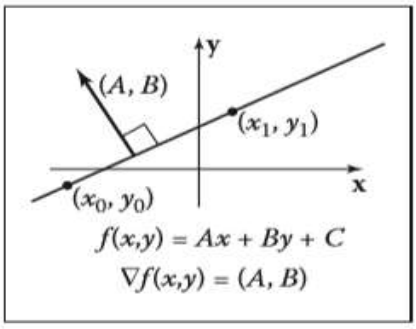
untuk bilangan real A, B, C. Misalkan kita mengetahui dua titik pada garis, (x_0, y_0) dan (x_1, y_1). Apakah A, B, dan C yang menggambarkan garis yang melalui dua titik ini? Karena titik-titik ini terletak pada garis, keduanya harus memenuhi Persamaan(2.15):

$$Ax_0 + By_0 + C = 0 ,$$
$$Ax_1 + By_1 + C = 0$$

Sayangnya kami memiliki dua persamaan dan tiga yang tidak diketahui: A, B, dan C. Masalah ini muncul karena pengali arbitrer yang dapat kita miliki dengan persamaan implisit. Kita dapat mengatur C = 1 untuk kenyamanan:

$$Ax + By + 1 = 0 ,$$

tetapi kita memiliki masalah yang serupa dengan kasus kemiringan tak hingga dalam bentuk perpotongan kemiringan: garis yang melalui titik asal harus memiliki A(0) + B(0) + 1 = 0, yang merupakan kontradiksi. Misalnya, persamaan garis 45 derajat yang melalui titik asal dapat ditulis x − y = 0, atau sama baiknya y − x = 0, atau genap 17y − 17x = 0, tetapi tidak dapat ditulis dalam bentuk Ax + By + 1=0



Gambar 2.29 Vektor gradien (A,B) tegak lurus terhadap garis implisit Ax + By + C = 0.

Setiap kali kita memiliki masalah aljabar sial seperti itu, kita mencoba untuk memecahkan masalah menggunakan intuisi geometris sebagai panduan. Salah satu alat yang kita miliki, seperti yang dibahas dalam Bagian 2.5.2, adalah gradien. Untuk garis Ax + By+ C =0, vektor gradiennya adalah (A,B). Vektor ini tegak lurus terhadap garis (Gambar 2.29), dan menunjuk ke sisi garis di mana Ax + By + C positif. Diberikan dua titik pada garis (x_0, y_0) dan (x_1, y_1), kita tahu bahwa vektor di antara keduanya menunjukkan arah yang sama dengan garis. Vektor ini hanya $(x_1 − x_0, y_1 − y_0)$, dan karena sejajar dengan garis, maka vektor tersebut juga harus tegak lurus terhadap vektor gradien (A, B). Ingatlah bahwa ada bilangan tak hingga (A, B, C) yang menggambarkan garis karena sifat scalling arbitrer dari implisit. Kami ingin salah satu dari yang valid (A, B, C).

Kita bisa mulai dengan sembarang(A, B) yang tegak lurus $(x_1 - x_0, y_1 - y_0)$. Vektor tersebut hanya (A, B) = $(y_0 - y_1, x_1 - x_0)$ dengan alasan yang sama seperti pada Bagian 2.5.2. Ini berarti persamaan garis yang melalui (x_0, y_0) dan (x_1, y_1) adalah

$$(y_0 − y_1)x + (x_1 − x_0)y + C = 0 .$$

Sekarang kita hanya perlu mencari C. Karena (x_0, y_0) dan (x_1, y_1) berada pada garis tersebut, keduanya harus memenuhi Persamaan (2.16). Kita dapat memasukkan salah satu nilai dan menyelesaikan untuk C. Melakukan ini untuk (x_0, y_0) menghasilkan $C = x_0y_1 – x_1y_0$, dan dengan demikian persamaan lengkap untuk garis adalah

$$(y_0 – y_1)x + (x_1 – x_0)y + x_0y_1 – x_1y_0 = 0 .$$

Sekali lagi, ini adalah salah satu dari banyak persamaan implisit yang valid untuk garis yang melalui dua titik, tetapi bentuk ini tidak memiliki operasi pembagian dan dengan demikian tidak ada kasus degenerasi numerik untuk titik-titik dengan koordinat Cartesian yang terbatas. Hal yang menyenangkan tentang Persamaan (2.17) adalah bahwa kita selalu dapat mengonversi ke bentuk perpotongan kemiringan (bila ada) dengan memindahkan suku non-y ke ruas kanan persamaan dan membaginya dengan pengali suku y:

$$y = \frac{y_1 - y_0}{x_1 - x_0}x + \frac{x_1y_0 - x_0y_1}{x_1 - x_0}$$

Sifat menarik dari persamaan garis implisit adalah dapat digunakan untuk mencari jarak bertanda dari suatu titik ke garis. Nilai Ax + By + C sebanding dengan jarak dari garis (Gambar 2.30). Seperti ditunjukkan pada Gambar 2.31, jarak dari suatu titik ke garis adalah panjang vektor k(A, B), yaitu

$$Jarak = k\sqrt{A^2 + B^2}$$

Untuk titik (x, y) + k(A, B), nilai pada f(x, y) = Ax + By + C adalah

$$f(x + kA, y + kB) = Ax + kA^2 + By + kB^2 + C$$
$$= k(A^2 + B^2).$$

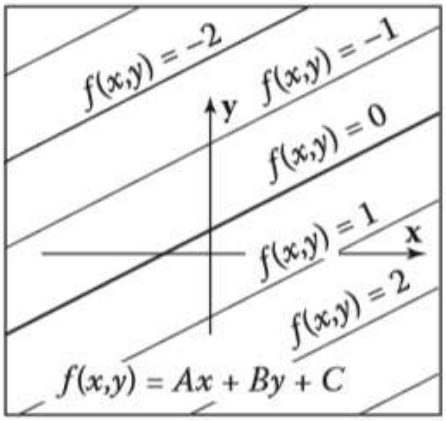
Penyederhanaan persamaan tersebut adalah hasil dari fakta bahwa kita tahu (x, y) berada pada garis, jadi Ax + By + C = 0. Dari Persamaan(2.18) dan(2.19), kita dapat melihat bahwa jarak bertanda dari garis Ax + By + C = 0 ke titik (a, b) adalah

$$jarak = \frac{f(a, b)}{\sqrt{A^2 + B^2}}$$

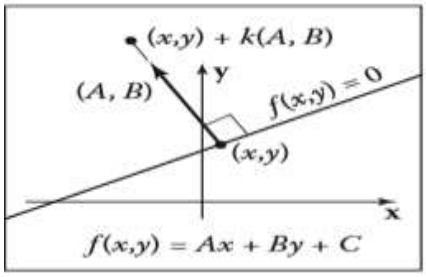
Di sini "jarak bertanda" berarti bahwa besar (nilai absolut) adalah jarak geometris, tetapi di satu sisi garis, jarak positif dan negatif. Anda dapat memilih antara representasi yang sama validnya f(x, y) = 0 dan f(x, y) = 0 jika masalah Anda memiliki alasan untuk memilih sisi tertentu yang positif. Perhatikan bahwa jika (A, B) adalah vektor satuan, maka f(a, b) adalah jarak bertanda. Kita dapat mengalikan Persamaan (2.17) dengan konstanta yang memastikan bahwa (A, B) adalah vektor satuan:

$$f(x, y) = \frac{y_0 - y_1}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}x + \frac{x_1 - x_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}y + \frac{x_0y_1 - x_1y_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}} = 0$$

Perhatikan bahwa mengevaluasi f(x, y) dalam Persamaan(2.20) secara langsung memberikan jarak bertanda, tetapi memerlukan akar kuadrat untuk menyiapkan persamaan. Garis implisit akan menjadi sangat berguna untuk asterisasi segitiga (Bagian 8.1.2). Bentuk lain untuk garis 2D dibahas dalam Bab 14.



Gambar 2.30 Nilai fungsi implisit f (x,y)= Ax+By+Cisakonstanta kali jarak bertanda dari Ax + By + C = 0.



Gambar 2.31 Vektor k(A,B) menghubungkan sebuah titik (x,y) pada garis yang terdekat dengan sebuah titik yang tidak berada pada garis tersebut. Jarak sebanding dengan k.

2.54. Kurva kuadrat implisit

Pada bagian sebelumnya kita melihat bahwa fungsi linier f(x, y) menghasilkan garis implisit f(x, y) = 0. Iff adalah fungsi kuadrat dari x dan y, dengan bentuk umum

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

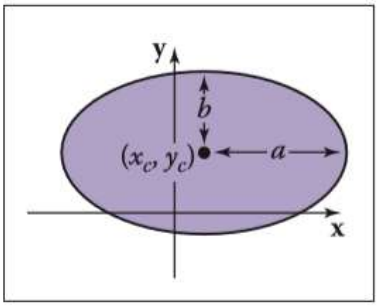
kurva implisit yang dihasilkan disebut kuadrat. Kurva kuadrat dua dimensi termasuk elips dan hiperbola, serta kasus khusus parabola, lingkaran, dan garis. Contoh kurva kuadrat meliputi lingkaran dengan pusat (x_c, y_c) dan jari-jari r,

$$(x - x_c)^2 + (y - y_c)2 - r^2 = 0,$$

dan elips sejajar sumbu dari bentuk

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} - 1 = 0$$

di mana (x_c, y_c) adalah pusat elips, dan a dan b adalah semi-sumbu minor dan mayor (Gambar 2.32)



Gambar 2.32 Elips dengan pusat (xc,y c) dan setengah sumbu dengan panjang a dan b.
Catatan : Coba atur a = b = r pada persamaan elips dan bandingkan dengan persamaan lingkaran.

2.5.5. Permukaan Implisit 3D

Sama seperti persamaan implisit yang dapat digunakan untuk mendefinisikan kurva dalam 2D, persamaan tersebut juga dapat digunakan untuk mendefinisikan permukaan dalam 3D. Asin2D, persamaan implisit secara implisit mendefinisikan sejumlah titik yang ada di permukaan:

$$f(x, y, z) = 0.$$

Setiap titik (x, y, z) yang ada di permukaan menghasilkan nol ketika diberikan sebagai argumen untuk f. Setiap titik yang tidak ada di permukaan menghasilkan beberapa angka selain nol. Anda dapat memeriksa apakah suatu titik berada di permukaan dengan mengevaluasi f, atau Anda dapat memeriksa sisi mana dari permukaan tersebut dengan melihat tanda f, tetapi Anda tidak dapat selalu secara eksplisit membangun titik di permukaan. Dengan menggunakan notasi vektor, kita akan menulis fungsi p = (x, y, z) sebagai

$$f(p) = 0.$$

2.5.6. Permukaan Normal ke Permukaan Implisit

Normal permukaan (yang diperlukan untuk perhitungan pencahayaan, antara lain) adalah vektor yang tegak lurus terhadap permukaan. Setiap titik pada permukaan mungkin memiliki vektor normal yang berbeda. Dengan cara yang sama bahwa gradien memberikan normal ke kurva implisit dalam 2D, permukaan normal pada titik p pada permukaan implisit diberikan oleh gradien dari fungsi implisit

$$n = \nabla f(p) = \left(\frac{\partial f(p)}{\partial x}, \frac{\partial f(p)}{\partial y}, \frac{\partial f(p)}{\partial z}\right)$$

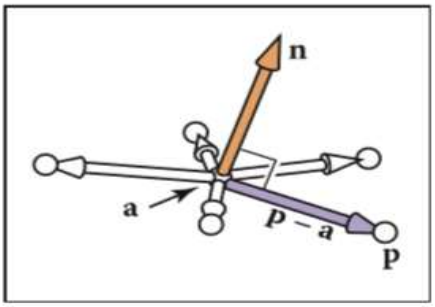
Alasan yang sama untuk kasus 2D: gradien menunjuk ke arah peningkatan tercepat dalam f, yang tegak lurus terhadap semua arah yang bersinggungan dengan permukaan, di mana f tetap konstan. Vektor gradien menunjuk ke arah sisi permukaan di mana f(p) > 0, yang mungkin kita anggap sebagai "ke dalam" permukaan atau "keluar dari" permukaan dalam konteks tertentu. Jika bentuk tertentu dari f menciptakan gradien menghadap ke dalam, dan gradien menghadap ke luar diinginkan, permukaan−f(p)= 0 sama dengan permukaan f(p)=0tetapi memiliki gradien terbalik arah, yaitu, ∇f(p) = ∇(−f(p)).

2.5.7. Pesawat Implisit

Sebagai contoh, perhatikan planet tak hingga melalui titik-titik dengan permukaan normal n. Persamaan implisit untuk menggambarkan bidang ini diberikan oleh

$$(p \cdot a) \cdot n = 0$$

Perhatikan bahwa a dan n adalah besaran yang diketahui. Titik p adalah sembarang titik yang tidak diketahui yang memenuhi persamaan. Dalam istilah geometris persamaan ini mengatakan "vektor dari a ke p tegak lurus terhadap bidang normal." Jika p tidak berada pada bidang, maka (p−a) tidak akan membentuk sudut siku-siku dengan n (Gambar2.33).



Gambar 2.33 Setiap titik p yang ditunjukkan berada pada bidang dengan vektor normal n yang mencakup titik a jika Persamaan (2.2) terpenuhi.

Terkadang kita menginginkan persamaan implisit untuk bidang yang melalui titik a, b, dan c. Normal untuk bidang ini dapat ditemukan dengan mengambil perkalian silang dari dua vektor di pesawat. Salah satu perkalian silang tersebut

$$n = (b - a) \times (c - a).$$

Ini memungkinkan kita untuk menulis persamaan bidang implisit:

$$(p - a) \cdot ((b - a) \times (c - a)) = 0.$$

Cara geometris untuk membaca persamaan ini adalah bahwa volume paralelepiped didefinisikan oleh p – a, b – a, dan c a adalah nol, yaitu, mereka adalah coplanar. Ini hanya bisa benar jika p berada pada bidang yang sama dengan a, b, dan c. Representasi Cartesian lengkap untuk ini diberikan oleh determinan:

$$\begin{vmatrix} x - x_a & y - y_a & z - z_a \\ x_b - x_a & y_b - y_a & z_b - z_a \\ x_c - x_a & y_c - y_a & z_c - z_a \end{vmatrix} = 0$$

Determinan dapat diperluas ke bentuk kembung dengan banyak istilah.

Persamaan (2.22) dan (2.23) setara, dan membandingkannya adalah pelajaran. Persamaan (2.22) mudah diinterpretasikan secara geometris dan akan menghasilkan kode yang efisien. Selain itu, relatif mudah untuk menghindari kesalahan tipografi yang dikompilasi menjadi kode yang salah jika memanfaatkan kode perkalian silang dan titik yang di-debug. Persamaan (2.23) juga mudah diinterpretasikan secara geometris dan akan efisien asalkan fungsi determinan 3 × 3 yang efisien diimplementasikan. Ini juga mudah untuk diimplementasikan tanpa tersedianya faktor penentu fungsi tipografi (a,b,c). Akan sangat mudah bagi orang lain untuk membaca kode Anda jika Anda mengganti nama volume fungsi penentu. Jadi Persamaan (2.22) dan (2.23) memetakan dengan baik ke dalam kode. Ekspansi penuh dari salah satu persamaan menjadi komponen x-, y-, dan z kemungkinan akan menghasilkan kesalahan ketik. Kesalahan ketik seperti itu cenderung dikompilasi dan, dengan demikian, menjadi sangat sial. Ini adalah contoh yang sangat baik dari matematika bersih yang menghasilkan kode bersih dan matematika kembung yang menghasilkan kode kembung.

2.5.8. Permukaan Kuadrat 3D

Sama seperti polinomial kuadrat dalam dua variabel mendefinisikan kurva kuadrat dalam 2D, polinomial kuadrat dalam x, y, dan z mendefinisikan permukaan kuadrat dalam 3D. Misalnya, bola dapat ditulis sebagai

$$f(p) = (p - c)^2 - r^2 = 0 ,$$

dan elipsoid sejajar sumbu dapat ditulis sebagai

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} + \frac{(z - z_c)^2}{c^2} - 1 = 0$$

2.5.8.1. Kurva 3D dari Permukaan Implisit

Seseorang mungkin berharap bahwa kurva 3D implisit dapat dibuat dengan bentuk f(p) = 0. Namun, semua kurva tersebut hanyalah permukaan yang merosot dan jarang berguna dalam praktik. Kurva A3D dapat dibangun dari perpotongan dua persamaan implisit simultan:

$$f(p) = 0 ,$$
$$g(p) = 0.$$

Misalnya, garis 3D dapat dibentuk dari perpotongan dua bidang implisit. Biasanya, lebih nyaman menggunakan kurva parametrik sebagai gantinya; mereka dibahas di bagian berikut.

2.5.8.2. Kurva Parametrik 2D

Kurva parametrik dikendalikan oleh satu parameter yang dapat dianggap sebagai semacam indeks yang bergerak terus menerus sepanjang kurva. Kurva tersebut memiliki bentuk

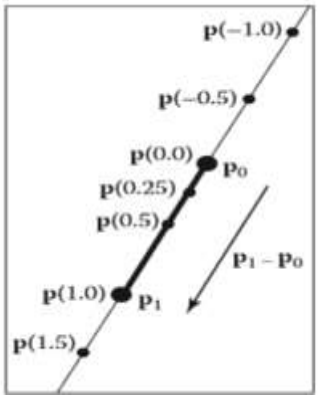
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} g(t) \\ h(t) \end{bmatrix}$$

Di sini (x, y) adalah titik pada kurva, dan t adalah parameter yang mempengaruhi kurva. Untuk t yang diberikan, akan ada beberapa titik yang ditentukan oleh fungsi g dan h. Untuk g dan h kontinu, perubahan kecil pada t akan menghasilkan perubahan kecil pada x dan y. Jadi, saat t terus berubah, titik-titik tersapu dalam kurva kontinu. Ini adalah fitur yang bagus karena kita dapat menggunakan parameter t untuk secara eksplisit membangun titik pada kurva. Seringkali kita dapat menulis kurva parametrik dalam bentuk vektor,

$$p = f(t),$$

di mana f adalah fungsi bernilai vektor, $f : \mathbb{R} \mapsto \mathbb{R}^2$. Fungsi vektor tersebut dapat menghasilkan kode yang sangat bersih, sehingga harus digunakan bila memungkinkan.

Kita dapat menganggap kurva dengan posisi sebagai fungsi waktu. Kurva bisa pergi ke mana saja dan bisa loop dan menyeberang sendiri. Kita juga dapat menganggap kurva memiliki kecepatan di sembarang titik. Misalnya, titik p(t) bergerak perlahan di dekat t = 2 dan dengan cepat antara t =2 dan t =3. Jenis kosakata "titik bergerak" ini sering digunakan ketika membahas kurva parametrik bahkan ketika kurva tidak menggambarkan titik bergerak.



Gambar 2.34 Garis parametrik 2D melalui p_0 dan p_1. Segmen garis yang didefinisikan oleh t ∈[0,1] ditampilkan dalam huruf tebal.

2.5.8.3. Garis Parametrik 2D

Garis parametrik pada 2D yang melalui titik $p_0 = (x_0, y_0)$ dan $p_1 = (x_1, y_1)$ dapat ditulis sebagai

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix}$$

Karena rumus untuk x dan y memiliki struktur yang mirip, kita dapat menggunakan bentuk vektor untuk p = (x, y) (Gambar2.34):

$$p(t) = p_0 + t(p_1 - p_0).$$

Anda dapat membaca ini dalam bentuk geometris sebagai: "mulai dari titik p_0 dan pergi beberapa jarak menuju p1 ditentukan oleh parameter t." Fitur yang bagus dari formulir ini adalah $p(0) = p_0$ dan $p(1) = p_1$. Karena titik berubah secara linier dengan t, nilai t antara p_0 dan p1 mengukur jarak fraksional antara titik-titik. Titik dengan t<0 berada di sisi “jauh” p_0, dan titik dengan t>1 berada di sisi “jauh” p_1. Garis parametrik juga dapat digambarkan hanya sebagai titik o dan vektor d:

$$p(t) = o + t(d).$$

Ketika vektor d memiliki panjang satuan, garis tersebut diparameterisasikan dengan panjang busur. Ini berarti t adalah ukuran yang tepat dari jarak di sepanjang garis. Setiap kurva parametrik dapat diparameterisasikan dengan panjang busur, yang jelas merupakan bentuk yang sangat nyaman, tetapi tidak semua dapat dikonversi secara analitis.

2.5.8.4. Lingkaran Parametrik 2D

Sebuah lingkaran dengan pusat (x_c, y_c) dan jari-jari r memiliki bentuk parametrik:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + r\cos\emptyset \\ y_c + r\sin\emptyset \end{bmatrix}$$

Untuk memastikan bahwa ada parameter unik untuk setiap titik pada kurva, kita dapat membatasi domainnya: [0,2π) atau (−π,π] atau interval setengah terbuka lainnya dengan panjang 2π. elips berjajar sumbu dapat dibuat dengan menskalakan persamaan parametrik x dan y secara terpisah:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_c + a\cos\emptyset \\ y_c + b\sin\emptyset \end{bmatrix}$$

2.5.8.4. Kurva Parametrik 3D

Kurva parametrik 3D beroperasi seperti kurva parametrik 2D:

$$x = f(t),$$
$$y = g(t),$$
$$z = h(t).$$

Misalnya, sebuah spiral di sekitar sumbu z ditulis sebagai:

$$x = cost,$$
$$y = sint,$$
$$z = t.$$

Seperti kurva 2D, fungsi f, g, andh didefinisikan pada domain D R jika kita ingin mengontrol di mana kurva mulai dan berakhir. Dalam bentuk vektor kita dapat menulis

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = p(t)$$

Dalam bab ini kita hanya membahas garis parametrik 3D secara detail. Kurva parametrik 3D umum dibahas lebih luas di Bab 15.

2.5.8.5. Garis Parametik 3D

Garis parametrik 3D dapat ditulis sebagai perpanjangan lurus ke depan dari garis parametrik 2D, misalnya,

$$x = 2 + 7t,$$
$$y = 1 + 2t,$$
$$z = 3 - 5t.$$

Ini rumit dan tidak diterjemahkan dengan baik ke variabel kode, jadi kami akan menulisnya dalam bentuk vektor:

$$p = o + td,$$

di mana, untuk contoh ini, o dan d diberikan oleh

$$o = (2, 1, 3),$$
$$d = (7, 2, -5).$$

Perhatikan bahwa ini sangat mirip dengan kasus 2D. Cara memvisualisasikannya adalah dengan membayangkan bahwa garis melewati o dan sejajar dengan d. Mengingat nilai

t berapa pun, Anda mendapatkan beberapa titik p(t) di telepon. Misalnya, pada t = 2 , p(t) = (2, 1, 3) + 2(7, 2, −5) = (16, 5, − 7). Konsep umum ini sama dengan dua dimensi (Gambar 2.30).

Seperti dalam 2D, segmen garis dapat digambarkan dengan garis parametrik 3D dan interval $t \in [t_a, t_b]$. Ruas garis antara dua titik a dan b diberikan oleh p(t) = a + t(b − a) dengan t ∈ [0, 1]. Di sini p(0) = a, p(1) = b, dan p(0.5) = (a + b)/2, titik tengah antara a dan b.

Sinar, garis setengah, adalah garis parametrik 3D dengan interval setengah terbuka, biasanya [0,∞). Mulai sekarang kita akan menyebut semua garis, ruas garis, dan sinar sebagai "sinar". Ini ceroboh, tetapi sesuai dengan penggunaan umum dan membuat diskusi lebih sederhana.

2.5.8.6. Permukaan Parametrik 3D

Pendekatan parametrik dapat digunakan untuk mendefinisikan permukaan dalam ruang 3D dengan cara yang sama seperti kita mendefinisikan kurva, kecuali bahwa ada dua parameter untuk menangani luas permukaan dua dimensi. Permukaan ini memiliki bentuk

x = f(u,v),
y = g(u,v),
z = h(u,v).

atau, dalam bentuk vektor,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = p(u, v)$$

Contoh:

Anggap saja Bumi benar-benar bulat, sebuah titik di permukaan Bumi dapat digambarkan dengan Pretend demi argumen bahwa Bumi itu benar-benar bulat. dua parameter garis bujur dan garis lintang. Jika kita mendefinisikan titik asal berada di pusat Bumi, dan misalkan r adalah jari-jari Bumi, maka sistem koordinat bola berpusat di titik asal (Gambar 2.35), mari kita turunkan persamaan parametrik

x = rcosϕ sinθ,
y = rsinϕ sinθ,
z = rcosθ.

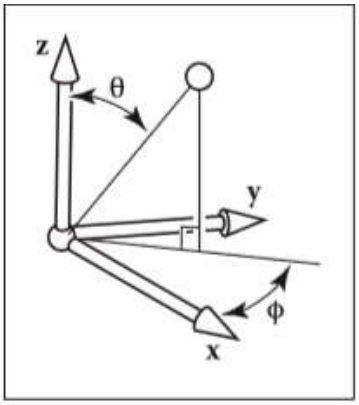
Idealnya, kami ingin menulis ini dalam bentuk vektor, tetapi tidak layak untuk bentuk parametrik khusus ini.

Kami juga ingin dapat menemukan (θ, ϕ) untuk (x, y, z) yang diberikan. Jika kita berasumsi bahwa (−π, π] ini mudah dilakukan dengan menggunakan fungsi atan2 dari Persamaan (2.2):

$$\theta = acoz\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right)$$

$$\emptyset = atan2(y, x)$$

Dengan permukaan implisit, turunan dari fungsi f memberi kita permukaan normal. Dengan permukaan parametrik, turunan dari p juga memberikan informasi tentang geometri permukaan.



Gambar 2.35 Geometri untuk koordinat bola.

Pertimbangkan fungsi q(t) = p(t, v_0). Fungsi ini mendefinisikan kurva parametrik yang diperoleh dengan memvariasikan u sambil menahan v tetap pada nilai v_0. Kurva ini, yang disebut kurva isoparametrik (atau kadang-kadang disingkat "isoparm") terletak di permukaan. Turunan dari q memberikan vektor tangen pada kurva, dan karena kurva terletak di permukaan, vektor q' juga terletak di permukaan. Karena diperoleh dengan memvariasikan satu argumen dari p, vektorq adalah turunan parsial dari p terhadap u, yang akan kita nyatakan p_u. Argumen serupa menunjukkan bahwa turunan parsial pv memberikan garis singgung ke kurva parametrik iso untuk konstanta, yang merupakan vektor tangen kedua ke permukaan.

Turunan p, kemudian, memberikan dua vektor tangen pada sembarang titik pada permukaan. Normal ke permukaan dapat ditemukan dengan mengambil perkalian silang dari vektor-vektor ini: karena keduanya bersinggungan dengan permukaan, perkalian silangnya, yang tegak lurus terhadap kedua garis singgung, adalah normal ke permukaan. Aturan tangan kanan untuk perkalian silang menyediakan cara untuk menentukan sisi mana yang merupakan bagian depan, atau bagian luar, dari permukaan; kita akan menggunakan konvensi bahwa vektor

$$n = = p_u \times p_v$$

mengarah ke luar permukaan.

2.5.8.7. Ringkasan Kurva dan Permukaan

Kurva implisit dalam 2D atau permukaan dalam 3D didefinisikan oleh fungsi bernilai skalar dari dua atau tiga variabel, $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ atau $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, dan permukaan terdiri dari semua titik di mana fungsinya adalah nol:

$$S = \{p \mid f(p) = 0\}.$$

Kurva parametrik dalam 2D atau 3D didefinisikan oleh fungsi bernilai vektor dari satu variabel, $p : D \subset \mathbb{R} \rightarrow \mathbb{R}^2$ atau $p : D \subset \mathbb{R} \rightarrow \mathbb{R}^3$, dan kurva disapu keluar karena t bervariasi pada semua D:

$$S = \{p(t) \mid t \in D\}.$$

Permukaan parametrik dalam 3D didefinisikan oleh fungsi bernilai vektor dari dua variabel, $p : D \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$, dan permukaan terdiri dari gambar semua titik (u, v) dalam domain:

$$S = \{p(t) \mid (u, v) \in D\}.$$

Untuk kurva dan permukaan implisit, vektor normal diberikan oleh turunan dari f (gradien), dan vektor tangen (untuk kurva) atau vektor (untuk permukaan) dapat diturunkan dari normal dengan membangun basis. Untuk kurva dan permukaan parametrik, turunan dari p memberikan vektor tangen (kurva fora) atau vektor (permukaan fora), dan vektor normal dapat diturunkan dari garis singgung dengan membangun basis.

2.6 INTERPOLASI LINEAR

Mungkin operasi matematika yang paling umum dalam grafis adalah interpolasi linier. Kita telah melihat contoh interpolasi linier posisi untuk membentuk segmen garis dalam 2D dan 3D, di mana dua titik a dan b diasosiasikan dengan parameter t untuk membentuk garis p = (1 − t)a + tb. Ini adalah interpolasi karena p melewati a dan b tepat pada t = 0 dan t = 1. Interpolasi linier karena suku bobot t dan 1 − t adalah polinomial linier dari t.

Interpolasi linier umum lainnya adalah di antara satu set posisi pada sumbu x: x_0, x_1, ... , x_n, dan untuk setiap x_i kita memiliki tinggi yang terkait, y_i. Kami ingin membuat fungsi kontinu y = f(x) yang menginterpolasi posisi ini, sehingga f melewati setiap titik data, yaitu, $f(x_i) = y_i$. Untuk interpolasi linier, titik-titik (xi, yi) dihubungkan oleh segmen garis lurus. Adalah wajar untuk menggunakan persamaan garis parametrik untuk segmen-segmen ini. Parameter t hanyalah jarak pecahan antara x_i dan x_i+1:

$$f(x) = y_i + \frac{x - x_i}{x_i + 1 - x_1}(y_i + 1 - y_i)$$

Karena fungsi pembobotan adalah polinomial linier dari x, ini adalah interpolasi linier. Dua contoh di atas memiliki bentuk umum dari interpolasi linier. Kami membuat variabel t yang bervariasi dari 0 hingga 1 saat kami berpindah dari item data A ke item data B. Nilai antara hanyalah fungsi (1 t)A + tB. Perhatikan bahwa Persamaan (2.26) memiliki bentuk ini dengan

$$t = \frac{x - x_i}{x_i + 1 - x_1}$$

2.7 SEGITIGA

Segitiga dalam 2D dan 3D adalah model dasar primitif di banyak program grafis. Seringkali informasi seperti warna ditandai ke simpul segitiga, dan informasi ini diinterpolasi melintasi segitiga. Sistem koordinat yang membuat interpolasi menjadi mudah disebut koordinat barycentric; kami akan mengembangkan ini dari awal. Kita juga akan membahas segitiga 2D, yang harus dipahami sebelum kita bisa menggambarnya di layar 2D.

1.1.8. Segitiga 2D

Jika kita memiliki segitiga 2D yang didefinisikan oleh titik 2D a, b, dan c, pertama-tama kita dapat mencari luasnya:

$$Area = \frac{1}{2}\begin{vmatrix} x_a - x_b & x_c - x_a \\ y_b - y_a & y_c - y_a \end{vmatrix}$$

$$= \frac{1}{2}(x_a y_b + x_b y_c + x_c y_a - x_a y_c - x_b y_a - x_c y_b)$$

Turunan dari rumus ini dapat ditemukan di Bagian 5.3. Daerah ini akan bertanda positif jika titik a, b, dan c berlawanan arah jarum jam dan bertanda negatif, sebaliknya.

Seringkali dalam grafis, kami ingin menetapkan properti, seperti warna, di setiap titik segitiga dan dengan lancar menginterpolasi nilai properti itu di segitiga. Ada berbagai cara untuk melakukan ini, tetapi yang paling sederhana adalah menggunakan koordinat barycentric. Salah satu cara untuk memikirkan koordinat barycentric adalah sebagai sistem koordinat nonorthogonal seperti yang telah dibahas secara singkat di Bagian 2.4.2. Sistem koordinat tersebut ditunjukkan pada Gambar 2.36, di mana asal koordinat adalah a dan vektor dari a ke b dan c adalah vektor basis. Dengan asal dan vektor basis tersebut, sembarang titik p dapat ditulis sebagai

$$p = a + \beta(b - a) + \gamma(c - a).$$

Perhatikan bahwa kita dapat menyusun ulang suku-suku dalam Persamaan (2.28) untuk mendapatkan

$$p = (1 - \beta - \gamma)a + \beta b + \gamma c.$$

Seringkali orang mendefinisikan variabel baruα untuk meningkatkan simetri persamaan:

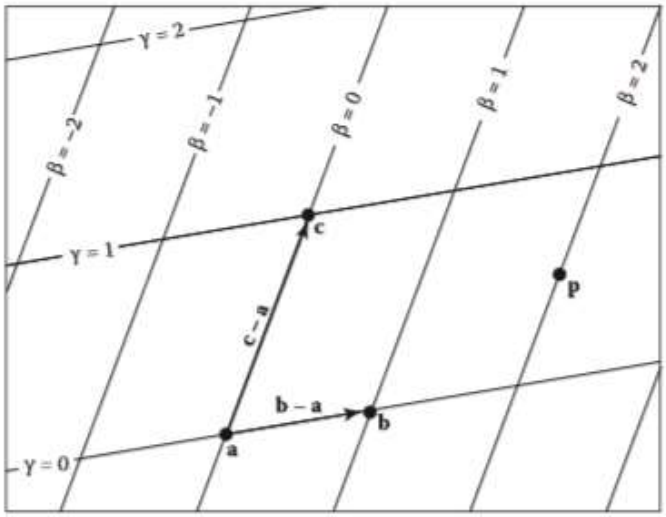
$$\alpha \equiv 1 - \beta - \gamma$$

yang menghasilkan persamaan

$$p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$$

dengan kendala bahwa

$$\alpha + \beta + \gamma = 1$$



Gambar 2.36 Segitiga 2D dengan simpul a, b, c dapat digunakan untuk membuat sistem koordinat nonortogonal dengan vektor asal a dan basis (b – a) dan (c – a). Sebuah titik kemudian diwakili oleh pasangan terurut (β, γ). Misalnya, titik p = (2.0, 0.5), yaitu p = a + 2.0 (b – a) + 0.5 (c – a).

Koordinat barycentric tampak seperti konstruksi abstrak dan tidak intuitif pada awalnya, tetapi ternyata menjadi kuat dan nyaman. Anda mungkin merasa berguna untuk memikirkan bagaimana alamat jalan akan bekerja di kota di mana ada dua set jalan paralel, tetapi di mana set tersebut tidak tegak lurus. Sistem alami pada dasarnya adalah koordinat barycentric, dan Anda akan segera terbiasa dengannya. Koordinat barycentric didefinisikan untuk semua titik pada bidang. Sebuah fitur yang sangat bagus dari koordinat barycentric adalah bahwa titik p berada di dalam segitiga yang dibentuk oleh a, b, dan c jika dan hanya jika

$$0 < \alpha < 1,$$
$$0 < \beta < 1,$$
$$0 < \gamma < 1.$$

Jika salah satu koordinatnya nol dan dua lainnya berada di antara nol dan satu, maka Anda berada di tepi. Jika dua koordinat adalah nol, maka yang lain adalah satu, dan Anda berada di sebuah titik. Properti bagus lainnya dari koordinat barycentric adalah bahwa Persamaan (2.29) pada dasarnya mencampur koordinat tiga simpul dengan cara yang mulus. Koefisien pencampuran yang sama (α, β, γ) dapat digunakan untuk mencampur sifat-sifat lain, seperti warna, seperti yang akan kita lihat di bab berikutnya.

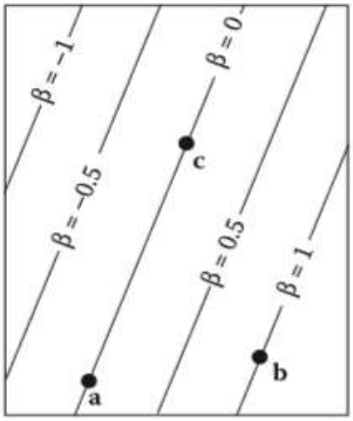
Diberikan titik p, bagaimana kita menghitung koordinat barycentric? Salah satu caranya adalah dengan menulis Persamaan (2.28) sebagai sistem linier dengan yang tidak diketahui β dan γ, selesaikan, dan tentukan α = 1 – β – γ. Sistem linier tersebut adalah

$$\begin{bmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_p - x_a \\ y_p - y_a \end{bmatrix}$$

Meskipun mudah untuk menyelesaikan Persamaan (2.31) secara aljabar, sering kali bermanfaat untuk menghitung solusi geometrik langsung. Salah satu sifat geometris dari koordinat barycentric adalah bahwa mereka adalah jarak skala bertanda dari garis melalui sisi segitiga, seperti yang ditunjukkan untuk pada Gambar 2.37. Ingat kembali dari Bagian 2.5.2bahwa menilai persamaan f(x, y) untuk garis f(x, y)=0mengembalikan jarak bertanda skala dari (x, y) ke garis. Ingat juga bahwa jika f(x, y)=0adalah persamaan untuk garis tertentu, demikian juga kf(x,y)=0 untuk sembarang k tak nol. Mengubah k menskalakan jarak dan mengontrol sisi garis mana yang memiliki jarak bertanda positif, dan mana yang negatif. Kami ingin memilih k sedemikian rupa sehingga, misalnya, kf(x, y)=β. Karena hanya satu yang tidak diketahui, kita dapat memaksa ini dengan satu kendala, yaitu bahwa pada titik b kita tahu =1. Jadi jika garis fac(x, y) = 0 melalui a dan c, maka kita dapat menghitung untuk suatu titik (x, y) sebagai berikut:

$$\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}$$

dan kita dapat menghitung γ dan α dengan cara yang sama. Efisiensi, biasanya bijaksana untuk menghitung hanya dua koordinat barycentric secara langsung dan menghitung yang ketiga menggunakan Persamaan (2.30).



Gambar 2.37 Koordinat barysentris adalah jarak skala bertanda dari garis yang melalui a dan c.

Untuk menemukan bentuk "ideal" untuk garis melalui p_0 dan p_1, pertama-tama kita dapat menggunakan teknik Bagian 2.5.2 untuk menemukan beberapa garis implisit yang valid melalui simpul. Persamaan (2.17) memberi kita

$$f_{ab}(x, y) \equiv (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0 .$$

Perhatikan bahwa $f_{ab}(x_c, y_c)$ mungkin tidak sama dengan satu, jadi ini mungkin bukan bentuk ideal yang kita cari. Dengan membagi melalui $f_{ab}(x_c, y_c)$ kita mendapatkan

$$\gamma = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}$$

Kehadiran pembagian mungkin mengkhawatirkan kita karena memperkenalkan kemungkinan pembagian dengan nol, tetapi ini tidak dapat terjadi untuk segitiga dengan area yang tidak mendekati nol. Ada rumus analog untuk dan , tetapi biasanya hanya diperlukan satu:

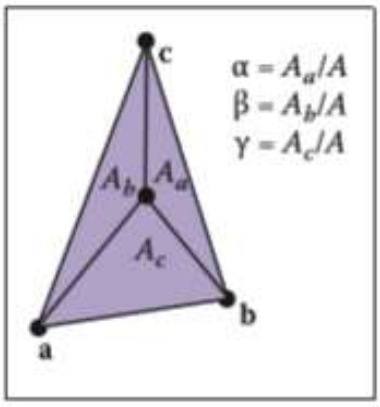
$$\gamma = \frac{(y_a - y_c)x + (x_c - x_a)y + x_a y_{cb} - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a}$$

$$\alpha = 1 - \beta - \gamma$$

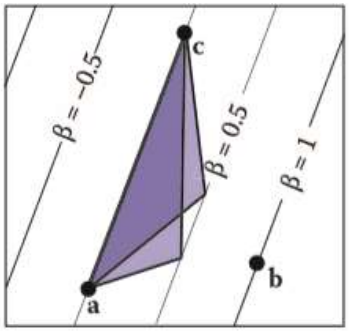
Cara lain untuk menghitung koordinat barycentric untuk menghitung daerah A_a, A_b, dan A_c, dari subtriangle seperti yang ditunjukkan pada Gambar 2.38. Koordinat barycentric mematuhi aturan

$$\alpha = A_a/A,$$
$$\beta = A_b/A,$$
$$\gamma = A_c/A,$$

dimana A adalah luas segitiga. Perhatikan bahwa $A = A_a + A_b + A_c$, sehingga dapat dihitung dengan dua penambahan daripada rumus luas penuh. Aturan ini masih berlaku untuk poin di luar segitiga jika area diizinkan untuk ditandatangani. Alasannya ditunjukkan pada Gambar 2.39. Perhatikan bahwa ini adalah daerah bertanda dan akan dihitung dengan benar selama perhitungan luas bertanda yang sama digunakan untuk A dan subsegitiga A_a, A_b, dan A_c.



Gambar 2.38 Koordinat barisentrik sebanding dengan luas tiga subsegitiga yang ditunjukkan.



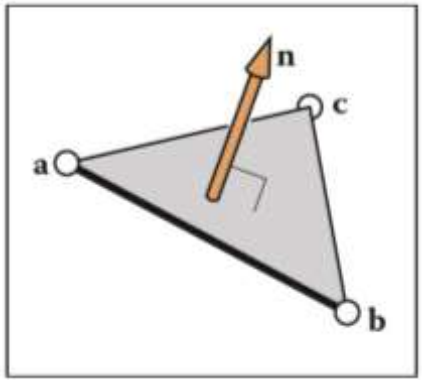
Gambar 2.39 Luas dua segitiga tanpa alas dikali tinggi dan dengan demikian sama, seperti segitiga apa pun dengan titik sudut pada garis = 0,5. Tinggi dan luasnya sebanding dengan.

1.1.9. Segitiga 3D

Satu hal yang luar biasa tentang koordinat barycentric adalah bahwa mereka meluas hampir secara transparan ke 3D. Jika kita menganggap titik a, b, dan c adalah 3D, maka kita masih dapat menggunakan representasi

$$p = (1 - \beta - \gamma)a + \beta b + \gamma c.$$

Sekarang, saat kita memvariasikan γ dan β, kita menyapu sebuah bidang.



Gambar 2.40 Vektor normal segitiga tegak lurus terhadap semua vektor pada bidang segitiga, dan dengan demikian tegak lurus terhadap tepi segitiga.

Vektor normal sebuah segitiga dapat ditemukan dengan mengambil perkalian silang dari dua vektor pada bidang segitiga (Gambar 2.40). Paling mudah untuk menggunakan dua dari tiga tepi sebagai vektor ini, misalnya,

$$n = (b - a) \times (c - a).$$

Perhatikan bahwa vektor normal ini belum tentu satuan panjang, dan mematuhi aturan tangan kanan perkalian silang.

Luas segitiga dapat ditemukan dengan mengambil panjang perkalian silang:

$$area = \frac{1}{2}||(b-a) \times (c-a)||$$

Perhatikan bahwa ini bukan area yang ditandai, sehingga tidak dapat digunakan secara langsung untuk mengevaluasi koordinat barycentric. Namun, kita dapat mengamati bahwa segitiga dengan urutan titik sudut "searah jarum jam" akan memiliki vektor normal yang menunjuk ke arah yang berlawanan dengan garis normal segitiga pada bidang yang sama dengan urutan titik sudut "berlawanan arah jarum jam". Ingat itu

$$a \cdot b = ||a||\ ||b|| \cos \emptyset$$

di mana Ø adalah sudut antara vektor. Jika dan sejajar, maka cos Ø = ±1, dan ini memberikan pengujian apakah vektor-vektor tersebut menunjuk pada arah yang sama atau berlawanan arah. Ini, bersama dengan Persamaan (2.33), (2.34), dan (2.35), menyarankan formula

$$\alpha = \frac{n \cdot n_2}{||n||^2}$$
$$\beta = \frac{n \cdot n_b}{||n||^2}$$
$$\gamma = \frac{n \cdot n_c}{||n||^2}$$

dimana n adalah Persamaan (2.34) dievaluasi dengan simpul a, b, dan c; n_a adalah Persamaan (2.34) dievaluasi dengan simpul b, c, dan p, dan seterusnya, yaitu,

$$n_a = (c - b) \times (p - b),$$
$$n_b = (a - c) \times (p - c),$$
$$n_c = (b - a) \times (p - a).$$

Pertanyaan yang Sering Diajukan

- ***Mengapa tidak ada pembagian vektor?***

Ternyata tidak ada analogi pembagian vektor yang “bagus”.

- ***Apakah ada sesuatu yang sebersih koordinat barycentric untuk poligon yang memiliki sisi lebih dari 3?***

Sayangnya tidak ada. Bahkan segi empat cembung jauh lebih rumit. Ini adalah salah satu alasan mengapa segitiga adalah primitif geometris yang umum dalam grafis.

- ***Apakah ada bentuk implisit untuk garis 3D?***

Tidak. Namun, perpotongan dua bidang 3D menentukan garis 3D, jadi garis 3D dapat dijelaskan dengan dua persamaan 3D implisit secara simultan.

Latihan

1. Kardinalitas suatu himpunan adalah jumlah elemen yang dikandungnya. Di bawah representasi titik terapung IEEE (Bagian 1.5), berapa kardinalitas pelampung?
2. Apakah mungkin untuk mengimplementasikan fungsi yang memetakan bilangan bulat 32-bit ke bilangan bulat 64-bit yang didefinisikan dengan baik sebagai invers? Apakah semua fungsi dari bilangan bulat 32-bit hingga bilangan bulat 64-bit memiliki invers yang terdefinisi dengan baik?
3. Tentukan kubus satuan (x-, y-, dan z-koordinat semua antara 0 dan 1) dalam hal produk Cartesian dari tiga interval.
4. Jika Anda memiliki akses ke fungsi log natural ln(x), tentukan bagaimana Anda dapat menggunakannya untuk mengimplementasikan fungsilog(b,x)di mana adalah dasar darilog. Apa yang harus dilakukan fungsi untuk nilai b negatif? Asumsikan implementasi titik mengambang IEEE.
5. Selesaikan persamaan kuadrat $2x^2 + 6x + 4 = 0$.
6. Implementasikan fungsi yang membutuhkan koefisien A, B, dan C untuk persamaan kuadrat $Ax^2 + Bx + C = 0$ dan menghitung dua solusi. Miliki fungsi yang mengembalikan jumlah solusi yang valid (bukan NaN) dan mengisi argumen balik sehingga yang lebih kecil dari dua solusi adalah yang pertama.
7. Tunjukkan bahwa dua bentuk rumus kuadrat pada halaman 17 adalah ekuivalen (dengan asumsi eksaktaritmatika) dan jelaskan bagaimana memilih satu untuk setiap akar untuk menghindari pengurangan bilangan titik yang hampir sama, yang menyebabkan hilangnya presisi.
8. Tunjukkan dengan contoh tandingan bahwa tidak selalu benar bahwa untuk vektor 3D a, b, dan c, $a \times (b \times c) = (a \times b) \times c$.
9. Diberikan vektor 3D nonparalel a dan b, hitunglah basis ortonormal tangan kanan sedemikian hingga u sejajar dengan a dan v pada bidang yang didefinisikan oleh a dan b.
10. Berapa gradien dari $f(x, y, z) = x^2 + y - 3z^3$?
11. Apa yang dimaksud dengan bentuk parametrik untuk elips 2D yang disejajarkan dengan sumbu?
12. Berapa persamaan implisit bidang yang melalui titik 3D (1, 0, 0), (0, 1, 0), dan (0, 0, 1)? Apa persamaan parametrik? Berapakah vektor normal bidang ini?
13. Diberikan empat titik 2D a_0, a_1, b_0, dan b_1, rancang prosedur yang kuat untuk menentukan apakah segmen garis a_0a_1 dan b_0b_1 berpotongan.

14. Rancang prosedur yang kuat untuk menghitung koordinat barycentric dari titik 2D sehubungan dengan tiga titik non-collinear 2D.

BAB 3
GAMBAR RASTER

Sebagian besar gambar grafis komputer disajikan kepada pengguna pada beberapa jenis tampilan raster. Tampilan raster menampilkan gambar sebagai susunan piksel persegi panjang. Contoh umum adalah layar komputer atau televisi panel datar, yang memiliki susunan persegi panjang dari piksel pemancar cahaya kecil yang dapat diatur secara individual ke warna berbeda untuk membuat gambar yang diinginkan. Warna yang berbeda dicapai dengan mencampur berbagai intensitas cahaya merah, hijau, dan biru. Sebagian besar printer, seperti printer laser dan printer ink-jet, juga merupakan perangkat raster. Mereka didasarkan pada pemindaian: tidak ada kisi fisik piksel, tetapi gambar diletakkan secara berurutan dengan menyimpan tinta pada titik yang dipilih pada kisi.

Raster juga lazim di perangkat input untuk gambar. Kamera digital berisi sensor gambar yang terdiri dari kisi-kisi piksel peka cahaya, yang masing-masing merekam warna dan intensitas cahaya yang jatuh di atasnya. Pemindai desktop berisi larik piksel linier yang menyapu halaman yang dipindai, membuat banyak pengukuran per detik untuk menghasilkan kisi piksel.

Karena raster sangat lazim di perangkat, gambar raster adalah cara paling umum untuk menyimpan dan memproses gambar. Gambar raster hanyalah larik 2D yang menyimpan nilai piksel untuk setiap piksel—biasanya warna yang disimpan sebagai tiga angka, untuk merah, hijau, dan biru. Gambar raster yang disimpan dalam memori dapat ditampilkan dengan menggunakan setiap piksel dalam gambar yang disimpan untuk mengontrol warna satu piksel tampilan.

Tapi kami tidak selalu ingin menampilkan gambar dengan cara ini. Kita mungkin ingin mengubah ukuran atau orientasi gambar, mengoreksi warna, atau bahkan menampilkan gambar yang ditempelkan pada permukaan tiga dimensi yang bergerak. Bahkan di televisi, layar jarang memiliki jumlah piksel yang sama dengan gambar yang ditampilkan. Pertimbangan seperti ini memutuskan hubungan langsung antara piksel gambar dan piksel tampilan. Sebaiknya pikirkan gambar raster sebagai deskripsi yang tidak bergantung pada perangkat dari gambar yang akan ditampilkan, dan perangkat tampilan sebagai cara untuk mendekati gambar ideal tersebut.

Ada cara lain untuk menggambarkan gambar selain menggunakan array piksel. Gambar vektor dideskripsikan dengan menyimpan deskripsi bentuk—area warna yang dibatasi oleh garis atau kurva—tanpa referensi ke kisi piksel tertentu. Intinya ini sama dengan menyimpan instruksi untuk menampilkan gambar daripada piksel yang diperlukan untuk menampilkannya. Keuntungan utama dari gambar vektor adalah bahwa mereka tidak bergantung pada resolusi dan dapat ditampilkan dengan baik pada perangkat dengan resolusi sangat tinggi. Kerugian yang sesuai adalah bahwa mereka harus di rasterisasi sebelum dapat ditampilkan. Gambar vektor sering digunakan untuk teks, diagram, gambar mekanik, dan aplikasi lain di mana kerenyahan dan presisi penting dan gambar fotografi dan bayangan yang rumit tidak diperlukan.

3.1 PERANGKAT RASTER

Sebelum membahas gambar raster dalam abstrak, ada baiknya untuk melihat operasi dasar dari beberapa perangkat tertentu yang menggunakan gambar ini. Beberapa perangkat raster yang sudah dikenal dapat dikategorikan ke dalam hierarki sederhana:

- Output

- Menampilkan
  - Transmisi: tampilan kristal cair (LCD)
  - Emissive: tampilan light-emitting diode (LED)
- Hardcopy
  - Biner: printer ink-jet
  - Continuoust satu: printer sublimasi pewarna

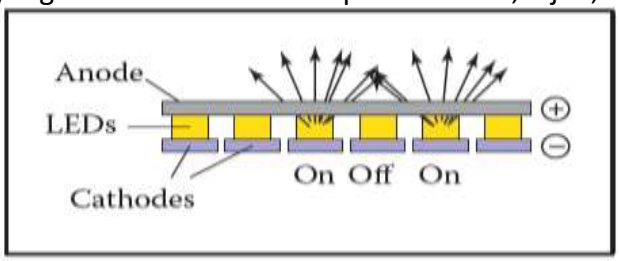
- Input
  - Sensor larik 2D: kamera digital
  - Sensor larik 1D: pemindai di tempat tidur

Display

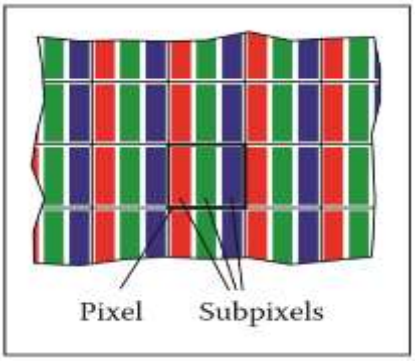
Layar saat ini, termasuk televisi dan proyektor sinematik digital serta layar dan proyektor untuk komputer, hampir secara universal didasarkan pada susunan piksel tetap. Mereka dapat dipisahkan menjadi tampilan emisif, yang menggunakan piksel yang secara langsung memancarkan jumlah cahaya yang dapat dikontrol, dan tampilan transmisif, di mana piksel itu sendiri tidak memancarkan cahaya melainkan memvariasikan jumlah cahaya yang diizinkan untuk melewatinya. Tampilan transmisif memerlukan sumber cahaya untuk meneranginya: dalam tampilan yang dilihat langsung, ini adalah lampu latar di belakang larik; di proyektor itu adalah lampu yang memancarkan cahaya yang diproyeksikan ke layar setelah melewati array. Tampilan memancarkan adalah sumber cahayanya sendiri.

Tampilan light-emitting diode (LED) adalah contoh dari tipe emisif. Setiap piksel terdiri dari satu atau lebih LED, yang merupakan perangkat semikonduktor (berdasarkan semikonduktor anorganik atau organik) yang memancarkan cahaya dengan intensitas tergantung pada arus listrik yang melewatinya (lihat Gambar 3.1).

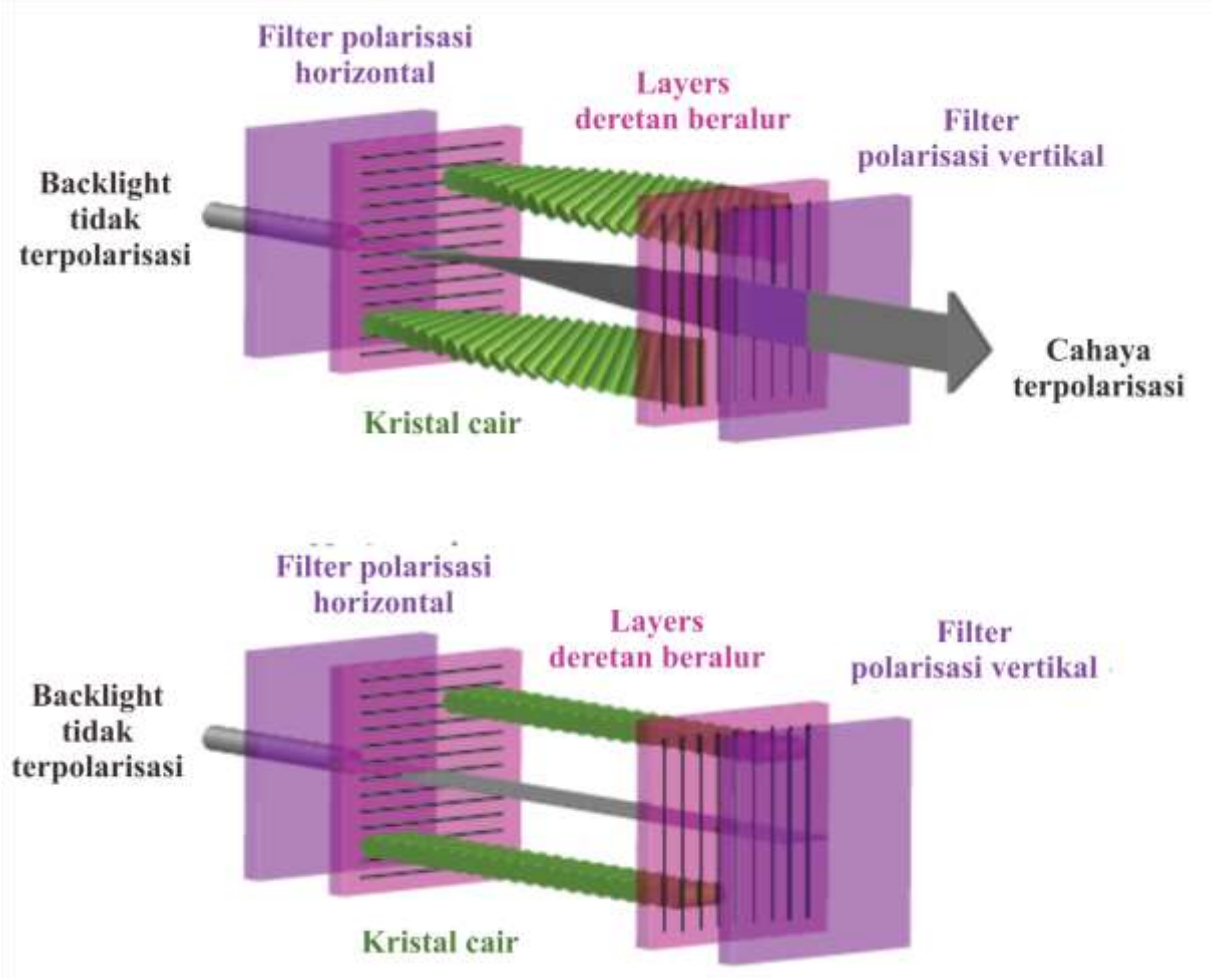
Piksel dalam tampilan warna dibagi menjadi tiga subpiksel yang dikontrol secara independen—satu merah, satu hijau, dan satu biru—masing-masing dengan LED sendiri yang dibuat menggunakan bahan berbeda sehingga memancarkan cahaya dengan warna berbeda (Gambar 3.2). Saat tampilan dilihat dari kejauhan, mata tidak dapat memisahkan subpiksel individual, dan warna yang dirasakan adalah campuran merah, hijau, dan biru.



Gambar 3.1 Pengoperasian tampilan dioda pemancar cahaya/ Light-emitting diode(LED).



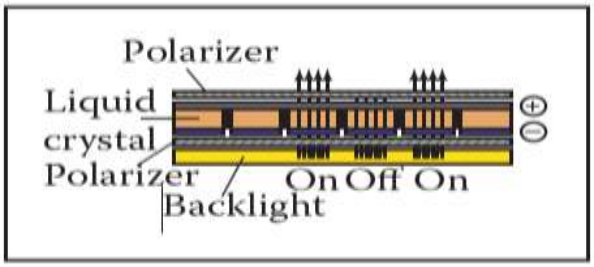
Gambar 3.2 Subpiksel merah, hijau, dan biru dalam piksel layar panel datar.



Gambar 3.3 Satu piksel layar LCD dalam keadaan mati (bawah), di mana polarizer depan memblokir semua cahaya yang melewati polarizer belakang, dan onstate (atas), di mana sel kristal cair memutar polarisasi cahaya sehingga itu dapat melewati polarizer depan. Gambar milik Erik Reinhard (Reinhard, Khan, Akyu¨z, & Johnson, 2008).

Liquid crystal display (LCD) adalah contoh dari tipe transmisif. Kristal cair adalah bahan yang struktur molekulnya memungkinkan untuk memutar polarisasi cahaya yang melewatinya, dan tingkat rotasi dapat disesuaikan dengan tegangan yang diberikan. Sebuah piksel LCD (Gambar 3.3) memiliki lapisan film polarisasi di belakangnya, sehingga disinari oleh cahaya terpolarisasi—asumsikan ia terpolarisasi secara horizontal.

Lapisan kedua dari film polarisasi di depan piksel diorientasikan untuk mentransmisikan hanya cahaya terpolarisasi vertikal. Jika tegangan yang diberikan diatur sehingga lapisan kristal cair di antaranya tidak mengubah polarisasi, semua cahaya diblokir dan piksel dalam keadaan "mati" (intensitas minimum). Jika voltase diatur sehingga kristal cair memutar polarisasi sebesar 90 derajat, maka semua cahaya yang masuk melalui bagian belakang piksel akan keluar melalui bagian depan, dan piksel sepenuhnya "aktif"—ini memiliki intensitas maksimum. Tegangan menengah akan memutar sebagian polarisasi sehingga polarizer depan sebagian menghalangi cahaya, sehingga menghasilkan intensitas antara minimum dan maksimum (Gambar 3.4). Seperti tampilan LED berwarna, LCD berwarna memiliki subpiksel merah, hijau, dan biru di dalam setiap piksel, yang merupakan tiga piksel independen dengan filter warna merah, hijau, dan biru di atasnya.

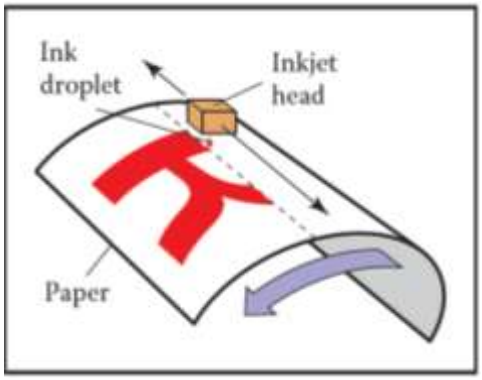


Gambar 3.4 Pengoperasian liquid crystal display (LCD).

Setiap jenis tampilan dengan kisi piksel tetap, termasuk teknologi ini dan teknologi lainnya, memiliki resolusi tetap yang secara fundamental ditentukan oleh ukuran kisi. Untuk tampilan dan gambar, resolusi hanya berarti dimensi kisi piksel: jika monitor desktop memiliki resolusi 1920 × 1200 piksel, ini berarti memiliki 2.304.000 piksel yang disusun dalam 1920 kolom dan 1200 baris. Gambar dengan resolusi berbeda, untuk memenuhi layar, harus diubah menjadi gambar 1920 x 1200 menggunakan metode Bab 9. Resolusi layar kadang-kadang disebut "resolusi asli" karena sebagian besar layar dapat menangani gambar dengan resolusi lain, melalui konversi bawaan.

Perangkat Hardcopy

Proses merekam gambar secara permanen di atas kertas memiliki kendala yang sangat berbeda dengan menampilkan gambar secara sementara di layar. Dalam pencetakan, pigmen didistribusikan di atas kertas atau media lain sehingga ketika cahaya dipantulkan dari kertas membentuk gambar yang diinginkan. Printer adalah perangkat raster seperti layar, tetapi banyak printer hanya dapat mencetak gambar biner—pigmen disimpan atau tidak pada setiap posisi grid, tanpa kemungkinan jumlah perantara.



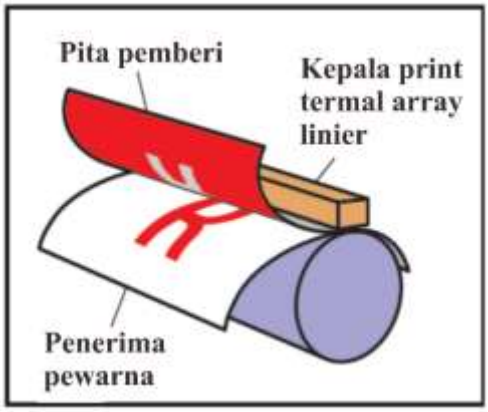
Gambar 3.5 Pengoperasian printer ink-jet.

Printer ink-jet (Gambar 3.5) adalah contoh perangkat yang membentuk gambar raster dengan memindai. Kepala cetak ink-jet mengandung pigmen pembawa tinta cair, yang dapat disemprotkan dalam tetesan yang sangat kecil di bawah kendali elektronik. Kepala bergerak melintasi kertas, dan tetesan dikeluarkan saat melewati posisi kisi yang seharusnya menerima tinta; tidak ada tinta yang terpancar di area yang dimaksudkan untuk tetap kosong. Setelah setiap sapuan, kertas dimajukan sedikit, dan kemudian baris kisi berikutnya diletakkan. Cetakan warna dibuat dengan menggunakan beberapa kepala cetak, masing-masing menyemprotkan tinta dengan pigmen yang berbeda, sehingga setiap posisi grid dapat menerima kombinasi tetesan warna yang berbeda. Karena semua tetesan sama, printer ink-jet mencetak gambar biner: pada setiap titik kisi ada setetes atau tidak ada setetes pun; tidak ada nuansa perantara. Ada juga printer ink-jet kontinu yang mencetak dalam jalur heliks

kontinu di atas kertas yang dililitkan di sekitar drum yang berputar, alih-alih menggerakkan kepala maju mundur.

Printer ink-jet tidak memiliki susunan piksel fisik; resolusi ditentukan oleh seberapa kecil tetesan dapat dibuat dan seberapa jauh kertas dimajukan setelah setiap sapuan. Banyak printer ink-jet memiliki beberapa nozel di kepala cetak, memungkinkan beberapa sapuan dilakukan dalam satu lintasan, tetapi gerak maju kertas, bukan jarak nozzle, yang pada akhirnya menentukan jarak baris.

Proses transfer pewarna termal adalah contoh dari proses pencetakan nada kontinu, yang berarti bahwa jumlah pewarna yang bervariasi dapat disimpan pada setiap piksel—tidak semuanya atau tidak sama sekali seperti printer ink-jet (Gambar 3.6). Pita donor yang mengandung pewarna berwarna ditekan di antara kertas, atau penerima pewarna, dan kepala cetak yang berisi susunan linier elemen pemanas, satu untuk setiap kolom piksel dalam gambar. Saat kertas dan pita bergerak melewati kepala, elemen pemanas hidup dan mati untuk memanaskan pita di area di mana pewarna diinginkan, menyebabkan pewarna berdifusi dari pita ke kertas. Proses ini diulang untuk masing-masing beberapa warna pewarna. Karena suhu yang lebih tinggi menyebabkan lebih banyak pewarna yang ditransfer, jumlah masing-masing pewarna yang disimpan pada setiap posisi kisi dapat dikontrol, memungkinkan rentang warna yang berkelanjutan untuk diproduksi. Jumlah elemen pemanas di kepala cetak menetapkan resolusi tetap dalam arah melintasi halaman, tetapi resolusi sepanjang halaman ditentukan oleh laju pemanasan dan pendinginan dibandingkan dengan kecepatan kertas.



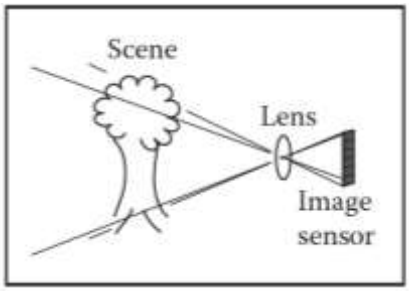
Gambar 3.6 Pengoperasian printer transfer pewarna termal.

Tidak seperti tampilan, resolusi printer dijelaskan dalam hal kerapatan piksel, bukan jumlah total piksel. Jadi printer transfer pewarna termal yang memiliki elemen spasi 300 per inci di kepala cetaknya memiliki resolusi 300 piksel per inci (ppi) di seluruh halaman. Jika resolusi sepanjang halaman yang dipilih sama, kita dapat dengan mudah mengatakan resolusi printer adalah 300 ppi. Printer ink-jet yang menempatkan titik-titik pada kisi dengan 1200 titik kisi per inci digambarkan memiliki resolusi 1200 titik per inci (dpi). Karena printer ink-jet adalah perangkat biner, printer ini membutuhkan jaringan yang jauh lebih halus setidaknya untuk dua alasan. Karena tepi adalah batas hitam/putih yang tiba-tiba, resolusi yang sangat tinggi diperlukan untuk menghindari loncatan tangga, atau aliasing, agar tidak muncul (lihat Bagian 8.3). Saat gambar nada kontinu dicetak, resolusi tinggi diperlukan untuk mensimulasikan warna menengah dengan mencetak pola titik dengan kerapatan bervariasi yang disebut halftone.

Catatan : Istilah "dpi" terlalu sering digunakan untuk mengartikan "piksel per inci", tetapi dpi harus digunakan mengacu pada perangkat biner dan ppi mengacu pada perangkat nada kontinu.

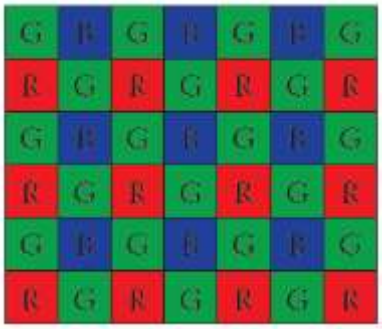
Input Device

Gambar raster harus berasal dari suatu tempat, dan gambar apa pun yang tidak dihitung oleh beberapa algoritma harus diukur oleh beberapa perangkat input raster, paling sering kamera atau pemindai. Bahkan dalam menampilkan gambar scene 3D, foto digunakan secara konstan sebagai peta tekstur (lihat Bab 11). Perangkat input raster harus melakukan pengukuran cahaya untuk setiap piksel, dan (seperti perangkat output) biasanya didasarkan pada array sensor.



Gambar 3.7 Pengoperasian kamera digital.

Kamera digital adalah contoh perangkat input array 2D. Sensor gambar dalam kamera adalah perangkat semikonduktor dengan kisi piksel peka cahaya. Dua jenis array yang umum dikenal sebagai sensor gambar CCD (charge-coupled devices) dan CMOS (complimentary metal-oxide-semiconductor). Lensa kamera memproyeksikan gambar scene untuk difoto ke sensor, dan kemudian setiap piksel mengukur energi cahaya yang jatuh padanya, yang pada akhirnya menghasilkan angka yang masuk ke gambar output (Gambar 3.7). Sama seperti tampilan warna subpiksel yang digunakan, hijau, dan biru, sebagian besar kamera warna bekerja dengan menggunakan array filter warna atau mosaik untuk memungkinkan setiap piksel hanya melihat cahaya merah, hijau, atau biru, meninggalkan perangkat lunak pengolah gambar untuk mengisi nilai-nilai yang hilang dalam proses yang dikenal sebagai demosaicking (Gambar 3.8).



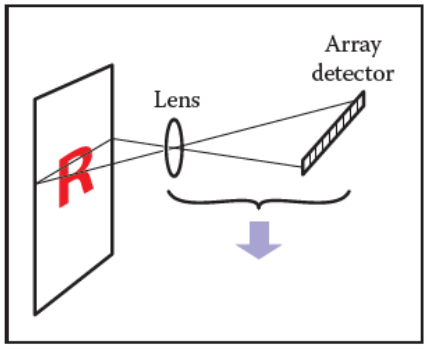
Gambar 3.8 Kebanyakan kamera digital berwarna menggunakan susunan filter warna yang mirip dengan mosaik Bayer yang ditunjukkan di sini. Setiap piksel mengukur cahaya merah, hijau, atau biru.

Kamera lain menggunakan tiga larik terpisah, atau tiga lapisan terpisah dalam larik, untuk mengukur nilai independen, hijau, dan biru dari piksel pengajaran, menghasilkan gambar berwarna yang dapat digunakan tanpa pemrosesan lebih lanjut. Resolusi kamera ditentukan oleh jumlah piksel tetap dalam larik dan biasanya dikutip menggunakan jumlah piksel total: kamera dengan larik 3000 kolom dan 2000 baris menghasilkan gambar resolusi 3000× 2000, yang memiliki 6 juta piksel, dan disebut kamera 6 megapiksel (MP). Penting untuk diingat bahwa sensor mosaik tidak mengukur gambar berwarna yang lengkap, jadi kamera yang mengukur jumlah piksel yang sama tetapi dengan pengukuran independen merah, hijau,

dan biru mencatat lebih banyak informasi tentang gambar daripada kamera dengan sensor mosaik. Orang yang menjual kamera menggunakan "mega" yang berarti 10^6, bukan 220 seperti halnya megabyte.

Pemindai flatbed juga mengukur nilai merah, hijau, dan biru untuk setiap kisi piksel, tetapi seperti printer transfer pewarna termal, pemindai ini menggunakan larik 1D yang menyapu halaman yang dipindai, membuat banyak pengukuran per detik. Resolusi di seluruh halaman ditentukan oleh ukuran larik, dan resolusi di sepanjang halaman ditentukan oleh frekuensi pengukuran dibandingkan dengan kecepatan kepala pindai bergerak. Pemindai warna memiliki larik 3xnx, di mana nx adalah jumlah piksel di seluruh halaman, dengan tiga baris ditutupi oleh filter merah, hijau, dan biru. Dengan penundaan yang tepat antara waktu di mana tiga warna diukur, ini memungkinkan tiga pengukuran warna independen di setiap titik kisi. Seperti halnya printer nada kontinu, resolusi pemindai dilaporkan dalam piksel per inci (ppi).

Resolusi pemindai kadang-kadang disebut "resolusi optik" karena sebagian besar pemindai dapat menghasilkan gambar dengan resolusi lain, melalui konversi bawaan. Dengan informasi konkret tentang dari mana gambar kita berasal dan ke mana mereka akan pergi, sekarang kita akan membahas gambar secara lebih abstrak, seperti yang akan kita gunakan dalam algoritme grafis.



Gambar 3.9 Pengoperasian pemindai flatbed.

3.2 GAMBAR, PIKSEL, DAN GEOMETRI

Kita tahu bahwa gambar raster adalah kumpulan piksel yang besar, yang masing-masing menyimpan informasi tentang warna gambar pada titik kisinya. Kami telah melihat apa yang dilakukan berbagai perangkat output dengan gambar yang kami kirimkan kepada mereka dan bagaimana perangkat input memperolehnya dari gambar yang dibentuk oleh cahaya di dunia fisik. Tetapi untuk komputasi di komputer, kita memerlukan abstraksi yang nyaman yang tidak bergantung pada spesifikasi perangkat apa pun, yang dapat kita gunakan untuk alasan tentang cara menghasilkan atau menafsirkan nilai yang disimpan dalam gambar.

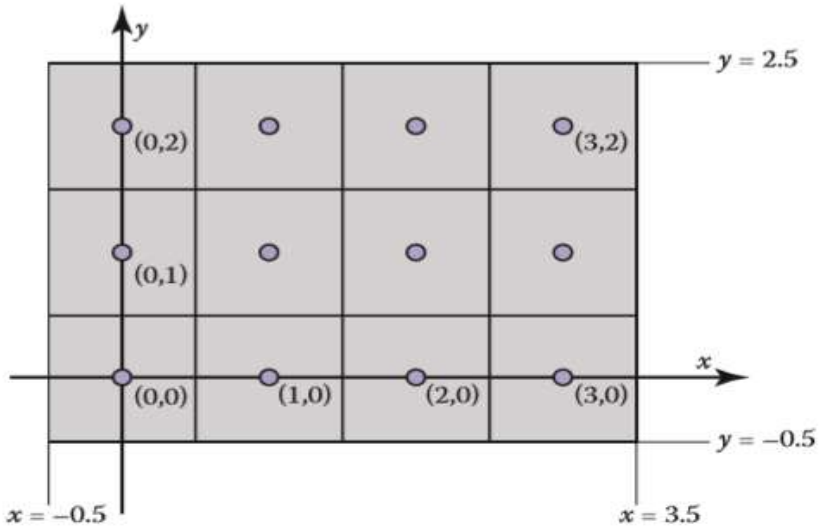
Ketika kita mengukur atau mereproduksi gambar, mereka mengambil bentuk distribusi energi cahaya dua dimensi: cahaya yang dipancarkan dari monitor sebagai fungsi posisi di muka layar; cahaya yang jatuh pada sensor gambar kamera sebagai fungsi posisi melintasi bidang sensor; pantulan, atau fraksi cahaya yang dipantulkan (sebagai lawan dari yang diserap) sebagai fungsi posisi pada selembar kertas. Jadi di dunia fisik, gambar adalah fungsi yang didefinisikan pada area dua dimensi—hampir selalu berbentuk persegi panjang. Jadi kita bisa mengabstraksikan sebuah gambar sebagai sebuah fungsi

$$I(x, y) : R \rightarrow V,$$

di mana $R \subset \mathbb{R}^2$ adalah area persegi panjang dan V adalah himpunan kemungkinan nilai piksel. Kasus paling sederhana adalah gambar skala abu-abu ideal di mana setiap titik dalam persegi panjang hanya memiliki kecerahan (tanpa warna), dan kita dapat mengatakan $V = \mathbb{R}^+$

(real non negatif). Gambar berwarna yang ideal, dengan nilai merah, hijau, dan biru pada setiap pikselnya, memiliki $V = (\mathbb{R}^+)^3$. Kami akan membahas kemungkinan lain untuk V di bagian selanjutnya.

Bagaimana gambar raster berhubungan dengan gagasan abstrak tentang gambar kontinu ini? Melihat contoh konkretnya, piksel dari kamera atau pemindai adalah pengukuran warna rata-rata gambar di beberapa area kecil di sekitar piksel. Piksel tampilan, dengan subpiksel merah, hijau, dan biru, dirancang sedemikian rupa sehingga warna rata-rata gambar di atas permukaan piksel dikendalikan oleh nilai piksel yang sesuai dalam gambar raster. Dalam kedua kasus, nilai piksel adalah rata-rata lokal dari warna gambar, dan ini disebut sampel titik gambar. Dengan kata lain, ketika kita menemukan nilai x dalam satu piksel, itu berarti "nilai gambar di sekitar titik kisi ini adalah x". Ide gambar sebagai representasi sampel dari fungsi dieksplorasi lebih lanjut dalam Bab 9.



Gambar 3.10 Koordinat layar empat piksel × tiga piksel. Perhatikan bahwa di beberapa API sumbu y akan mengarah ke bawah.

Sebuah pertanyaan biasa tapi penting adalah di mana piksel berada di ruang 2D. Ini hanya masalah kesepakatan, tetapi membangun kesepakatan yang konsisten itu penting! Dalam buku ini, gambar raster diindeks oleh pasangan (i, j) yang menunjukkan kolom (i) dan baris (j) dari piksel, dihitung dari kiri bawah. Jika sebuah gambar memiliki n_x kolom dan ny baris piksel, piksel kiri bawah adalah (0, 0) dan kanan atas adalah piksel (n_x - 1, n_y – 1). Kita membutuhkan koordinat layar nyata 2D untuk menentukan posisi piksel. Kami akan menempatkan titik sampel piksel pada koordinat bilangan bulat, seperti yang ditunjukkan oleh layar 4x3 pada Gambar 3.10.

Dalam beberapa API, dan banyak format file, baris gambar diatur dari atas ke bawah, sehingga (0, 0) berada di kiri atas. Ini karena alasan historis: baris dalam transmisi televisi analog dimulai dari atas. Domain persegi panjang gambar memiliki lebar nx dan tinggi ny dan berpusat pada kisi ini, yang berarti bahwa ia meluas setengah piksel di luar titik sampel terakhir di setiap sisi. Jadi domain persegi panjang dari gambar nx ×ny adalah

$$R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5].$$

Sekali lagi, koordinat ini hanyalah konvensi, tetapi penting untuk diingat nanti saat menerapkan kamera dan melihat transformasi. Beberapa sistem menggeser koordinat dengan setengah piksel untuk menempatkan titik sampel di tengah antara bilangan bulat tetapi menempatkan tepi gambar pada bilangan bulat.

Nilai Piksel

Sejauh ini kami telah menggambarkan nilai piksel dalam bentuk bilangan real, yang mewakili intensitas (mungkin secara terpisah untuk merah, hijau, dan biru) pada suatu titik dalam gambar. Hal ini menunjukkan bahwa gambar harus berupa larik angka titik mengambang, dengan salah satu (untuk gambar skala abu-abu, atau hitam putih) atau tiga (untuk gambar berwarna RGB) nomor titik mengambang 32-bit yang disimpan per piksel. Format ini kadang-kadang digunakan, ketika presisi dan rentang nilai diperlukan, tetapi gambar memiliki banyak piksel dan memori dan bandwidth untuk menyimpan dan mentransmisikan gambar selalu langka. Hanya satu foto sepuluh megapiksel akan menghabiskan sekitar 115 MB RAM dalam format ini.

Rentang yang lebih sedikit diperlukan untuk gambar yang dimaksudkan untuk ditampilkan secara langsung. Sementara kisaran intensitas cahaya yang mungkin tidak terbatas pada prinsipnya, perangkat apa pun yang diberikan memiliki intensitas maksimum yang jelas terbatas, jadi dalam banyak konteks itu cukup sempurna untuk piksel memiliki rentang terbatas, biasanya dianggap [0,1] untuk kesederhanaan. Misalnya, nilai yang mungkin dalam gambar 8-bit adalah 0, 1/255, 2/255, ..., 254/255, 1. Gambar yang disimpan dengan nomor floating-point, memungkinkan rentang nilai yang luas, sering disebut gambar high dynamic range (HDR) untuk membedakannya dari gambar fixed-range, atau low dynamic range (LDR) yang disimpan dengan bilangan bulat. Lihat Bab 21 untuk diskusi mendalam tentang teknik dan aplikasi untuk gambar rentang dinamis tinggi. Denominator 255, bukan 256, tetapi kemampuan untuk merepresentasikan0dan1dengan tepat adalah penting.

Berikut adalah beberapa format piksel dengan aplikasi tipikal:

- Grayscale – 1 –bit—teks dan gambar lain di mana abu-abu menengah tidak diinginkan (diperlukan resolusi tinggi);
- 8-bitRGB fixed-rangecolor (24bits total perpixel)—aplikasi email webband, foto konsumen;
- 8-atau10-bitfixed-rangeRGB (24–30bits/piksel)—antarmuka digital ke tampilan komputer;
- 12- hingga 14-bit RGB rentang tetap (36–42 bit/piksel)—gambar kamera mentah untuk fotografi profesional;
- 16-bit fixed-range RGB (48 bits/pixel)—fotografi dan pencetakan profesional;format menengah untuk pemrosesan gambar dari gambar fixed-range;
- Skala abu-abu rentang tetap 16-bit (16 bit/piksel)—radiologi dan pencitraan medis;
- 16-bit“setengah-presisi”floating-pointRGB—gambar HDR;format menengah untuk rendering waktu-nyata;
- 32-bit floating-point RGB—format menengah tujuan umum untuk rendering perangkat lunak dan pemrosesan gambar HDR.

Mengurangi jumlah bit yang digunakan untuk menyimpan setiap piksel menyebabkan dua jenis artefak yang berbeda, atau cacat yang diperkenalkan secara artifisial, dalam gambar. Pertama, pengkodean gambar dengan nilai rentang tetap menghasilkan kliping ketika piksel yang seharusnya lebih terang dari nilai maksimum diatur, atau dipotong, ke nilai representasi maksimum. Misalnya, foto scene yang cerah mungkin menyertakan pantulan yang jauh lebih terang daripada permukaan putih; ini akan terpotong (bahkan jika diukur oleh kamera) saat gambar diubah menjadi rentang tetap untuk ditampilkan. Kedua, pengkodean gambar dengan presisi terbatas mengarah ke artefak kuantisasi, atau pita, ketika kebutuhan untuk membulatkan nilai piksel ke nilai terdekat yang dapat diwakili menimbulkan lompatan yang terlihat dalam intensitas atau warna. Banding bisa sangat berbahaya dalam animasi dan video,

di mana band mungkin tidak dapat diterima dalam gambar diam, tetapi menjadi sangat terlihat ketika mereka bergerak maju mundur.

Pantau Intensitas dan Gamma

Semua monitor modern mengambil input digital untuk "nilai" piksel dan mengubahnya menjadi tingkat intensitas. Monitor asli memiliki intensitas bukan nol saat dimatikan karena layar memantulkan sedikit cahaya. Untuk tujuan kami, kami dapat menganggap ini "hitam" dan monitor menyala sepenuhnya sebagai "putih." Kami mengasumsikan deskripsi numerik warna piksel yang berkisar dari nol hingga satu. Hitam adalah nol, putih adalah satu, dan setengah abu-abu antara hitam dan putih adalah 0,5. Perhatikan bahwa di sini "setengah" mengacu pada jumlah fisik cahaya yang berasal dari piksel, bukan tampilannya. Persepsi manusia tentang intensitas adalah nonlinier dan tidak akan menjadi bagian dari diskusi ini; lihat Bab 20 untuk lebih lanjut.

Ada dua isu utama yang harus dipahami untuk menghasilkan gambar yang benar pada monitor. Yang pertama adalah bahwa monitor bersifat nonlinier sehubungan dengan input. Misalnya, jika Anda memberikan monitor 0, 0,5, dan 1,0 sebagai input untuk tiga piksel, intensitas yang ditampilkan mungkin 0, 0,25, dan 1,0 (mati, seperempat menyala penuh, dan menyala penuh). Sebagai perkiraan karakterisasi nonlinier ini, monitor biasanya dicirikan oleh nilai (“gamma”). Nilai ini adalah derajat kebebasan dalam rumus

$$\textit{intensitas yang ditampilkan} = (\textit{intensitas maksimum})\, a^{\gamma},$$

di mana a adalah nilai piksel input antara nol dan satu. Misalnya, jika monitor memiliki gamma 2.0, dan kita memasukkan nilai a =0,5, intensitas yang ditampilkan akan menjadi seperempat dari intensitas maksimum yang mungkin karena 0,52 = 0 .25. Perhatikan bahwa a = 0 peta ke intensitas nol dan a = 1 peta ke intensitas maksimum terlepas dari nilai. Menggambarkan nonlinier tampilan menggunakan 1γ hanyalah perkiraan; kita tidak membutuhkan banyak akurasi dalam memperkirakan perangkat. Cara visual yang bagus untuk mengukur ketidaklinieran adalah dengan menemukan nilai a yang memberikan intensitas di tengah antara hitam dan putih. Ini akan menjadi

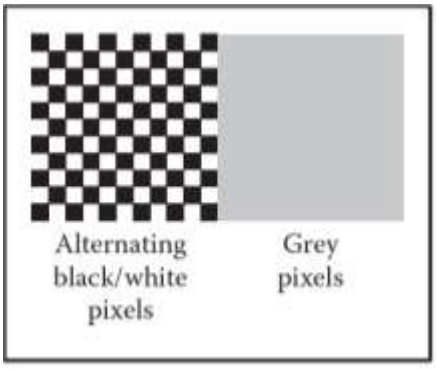
$$0.5 = a^{\gamma}$$

Jika kita dapat menemukan bahwa a, kita dapat menyimpulkan γ dengan mengambil logaritma di kedua sisi:

$$\gamma = \frac{di\ 0.5}{di\ a}$$

Untuk monitor dengan antarmuka analog, yang memiliki kesulitan mengubah intensitas dengan cepat di sepanjang arah horizontal, garis hitam dan putih horizontal bekerja lebih baik daripada papan catur.

Kita dapat menemukan ini dengan teknik standar di mana kita menampilkan pola kotak-kotak piksel hitam dan putih di sebelah kotak piksel abu-abu dengan input a (Gambar 3.11), kemudian meminta pengguna untuk menyesuaikan (dengan penggeser, misalnya) hingga kedua sisi cocok dalam kecerahan rata-rata. Ketika Anda melihat gambar ini dari kejauhan (atau tanpa kacamata jika Anda rabun jauh), kedua sisi gambar akan terlihat hampir sama ketika a menghasilkan intensitas setengah jalan antara hitam dan putih. Ini karena kotak-kotak buram mencampur jumlah piksel putih dan hitam yang genap sehingga efek keseluruhannya adalah warna yang seragam di tengah antara putih dan hitam.



Gambar 3.11 Piksel hitam dan putih bergantian dilihat dari kejauhan adalah setengah jalan antara hitam dan putih. Gamma monitor dapat disimpulkan dengan menemukan nilai abu-abu yang tampaknya memiliki intensitas yang sama dengan pola hitam putih.

Setelah mengetahui , kita dapat mengoreksi input gamma sehingga nilai a =0,5 ditampilkan dengan intensitas setengah jalan antara hitam dan putih. Ini dilakukan dengan transformasi

$$a' = a^{\frac{1}{\gamma}}$$

Ketika rumus ini dicolokkan ke Persamaan (3.1) kita dapatkan

$$intensitas\ yang\ ditampilkan = (a')^{\gamma} = \left(a^{\frac{1}{\gamma}}\right)^{\gamma}\ (intensitas\ maksimum) = a($$

Karakteristik penting lainnya dari tampilan nyata adalah bahwa mereka mengambil nilai input terkuantisasi. Jadi sementara kita dapat memanipulasi intensitas dalam rentang titik mengambang [0, 1], input detail ke monitor adalah bilangan bulat ukuran tetap. Rentang paling umum untuk bilangan bulat ini adalah 0 - 255 yang dapat disimpan dalam penyimpanan 8 bit. Ini berarti bahwa nilai yang mungkin untuk a bukanlah sembarang bilangan di [0, 1] melainkan

$$kemungkinan\ nilai\ untuk\ a = \left\{\frac{0}{255}, \frac{1}{255}, \frac{2}{255}, \ldots, \frac{254}{255}, \frac{255}{255}\right\}$$

Ini berarti kemungkinan nilai intensitas yang ditampilkan kira-kira

$$\left\{M\left(\frac{0}{255}\right)^{\gamma}, M\left(\frac{1}{255}\right)^{\gamma}, M\left(\frac{2}{255}\right)^{\gamma}, \ldots, M\left(\frac{245}{255}\right)^{\gamma}, M\left(\frac{255}{255}\right)^{\gamma}\right\}$$

dimana M adalah intensitas maksimum. Dalam aplikasi di mana intensitas yang tepat perlu dikontrol, kita harus benar-benar mengukur 256 kemungkinan intensitas, dan intensitas ini mungkin berbeda pada titik yang berbeda di layar, terutama untuk CRT. Mereka mungkin juga berbeda dengan sudut pandang. Untungnya, beberapa aplikasi memerlukan kalibrasi yang akurat.

3.3 WARNA RGB

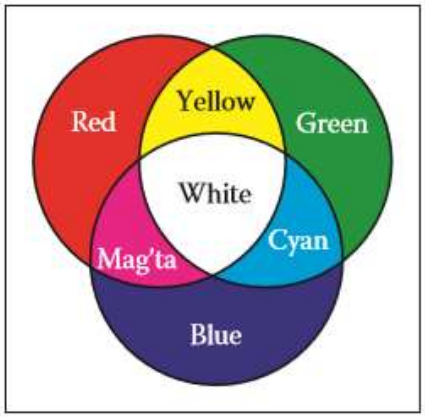
Sebagian besar gambar grafis komputer didefinisikan dalam istilah warna merah-hijau-biru (RGB). Warna RGB adalah ruang sederhana yang memungkinkan konversi langsung ke kontrol untuk sebagian besar layar komputer. Pada bagian ini, warna RGB dibahas dari sudut pandang pengguna, dan fasilitas operasional adalah tujuannya. Diskusi warna yang lebih menyeluruh diberikan dalam Bab 19, tetapi mekanisme ruang warna RGB akan memungkinkan kita untuk menulis sebagian besar program grafis. Ide dasar ruang warna RGB adalah bahwa warna ditampilkan dengan mencampurkan tiga lampu utama: satu merah, satu hijau, dan satu biru. Lampu bercampur dengan cara aditif. Di sekolah dasar Anda mungkin belajar bahwa sekolah dasar adalah merah, kuning, dan biru, dan bahwa, misalnya, kuning +

biru = hijau. Ini adalah pencampuran warna subtraktif, yang pada dasarnya berbeda dari pencampuran aditif yang lebih umum yang terjadi di layar.
Dalam pencampuran warna aditif RGB yang kami miliki (Gambar 3.12)

red + green = yellow
green + blue = cyan
blue + red = magenta
red + green + blue = white

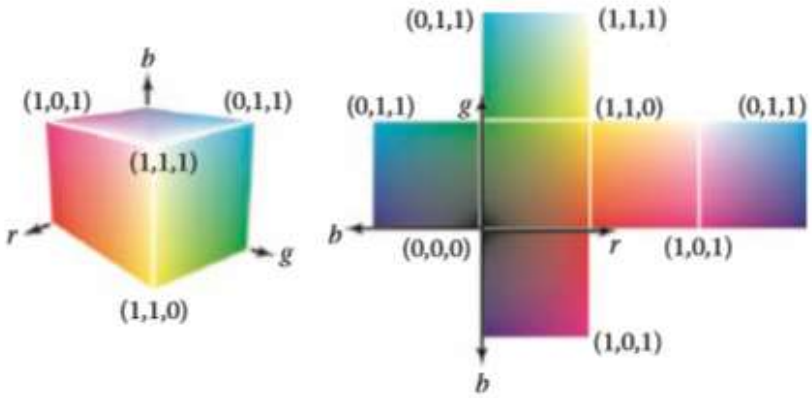
Warna "cyan" adalah biru-hijau, dan warna "magenta" adalah ungu.

black = (0 ,0,0),
red = (1 ,0,0),
green = (0 ,1,0),
blue = (0 ,0,1),
yellow = (1 ,1,0),
magenta = (1 ,0,1),



Gambar 3.12 Aturan pencampuran aditif untuk warna merah/hijau/biru.

Jika kita diizinkan untuk meredupkan lampu utama dari mati total (ditunjukkan dengan nilai piksel 0) hingga menyala penuh (ditunjukkan dengan 1), kita dapat membuat semua warna yang dapat ditampilkan pada monitor RGB. Nilai piksel merah, hijau, dan biru membuat kubus warna RGB tiga dimensi yang memiliki sumbu merah, hijau, dan biru. Koordinat yang diizinkan untuk sumbu berkisar dari nol hingga satu. Kubus warna ditunjukkan secara grafis pada Gambar 3.13. Warna pada sudut kubus adalah



Gambar 3.13 Kubus warna RGB dalam 3D dan wajahnya terbuka. Setiap warna RGB adalah titik dalam kubus.

cyan = (0 ,1,1),
white = (1 ,1,1).

Tingkat RGB sebenarnya sering diberikan dalam bentuk terkuantisasi, seperti skala abu-abu yang dibahas dalam Bagian 3.2.2. Setiap komponen ditentukan dengan bilangan bulat. Ukuran paling umum untuk bilangan bulat ini adalah satu byte masing-masing, jadi masing-masing dari tiga komponen RGB adalah bilangan bulat antara 0 dan 255. Ketiga bilangan bulat bersama-sama mengambil tiga byte, yaitu 24 bit. Jadi sistem yang memiliki "warna 24-bit" memiliki 256 kemungkinan level untuk masing-masing dari tiga warna primer. Masalah koreksi gamma yang dibahas dalam Bagian 3.2.2 juga berlaku untuk setiap komponen RGB secara terpisah.

3.4 KOMPOSISI ALPHA

Seringkali kita hanya ingin menimpa sebagian isi piksel. Contoh umum dari hal ini terjadi dalam pengomposisian, di mana kita memiliki latar belakang dan ingin menyisipkan gambar latar depan di atasnya. Untuk piksel buram di latar depan, kami hanya mengganti piksel latar belakang. Untuk piksel latar depan yang sepenuhnya transparan, kami tidak mengubah piksel latar belakang. Untuk piksel yang sebagian transparan, beberapa perhatian harus diberikan. Piksel transparan sebagian dapat terjadi ketika objek latar depan memiliki bagian transparan sebagian, seperti kaca. Tapi, kasus yang paling sering di mana latar depan dan latar belakang harus dicampur adalah ketika objek latar depan hanya menutupi sebagian piksel, baik di tepi objek latar depan, atau ketika ada lubang sub-piksel seperti di antara daun pohon yang jauh.

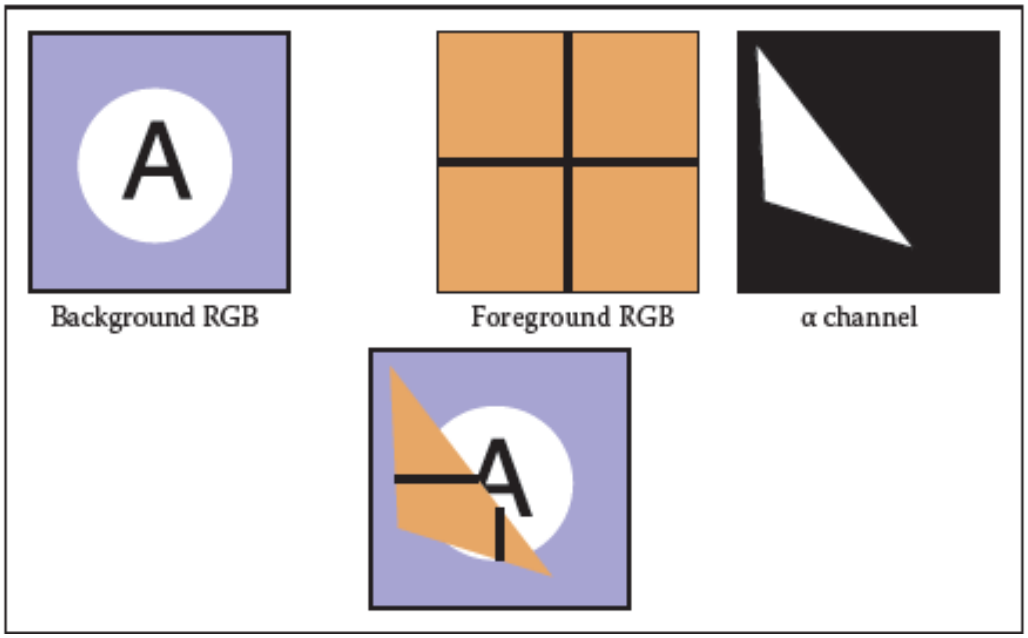
Bagian informasi terpenting yang diperlukan untuk memadukan objek latar depan di atas objek latar belakang adalah cakupan piksel, yang memberi tahu fraksi piksel yang dicakup oleh lapisan latar depan. Kita dapat menyebut pecahan ini α. Jika kita ingin menggabungkan warna latar depan c_f di atas warna latar belakang c_b, dan fraksi piksel yang dicakup oleh latar depan adalah , maka kita dapat menggunakan rumus

$$c = \alpha c_f + (1 - \alpha)c_b.$$

Untuk lapisan latar depan buram, interpretasinya adalah bahwa objek latar depan menutupi area di dalam persegi panjang piksel dan objek latar belakang menutupi area yang tersisa, yaitu (1). Untuk lapisan transparan (bayangkan gambar yang dilukis di atas kaca atau di atas kertas kalkir, menggunakan cat tembus pandang), interpretasinya adalah bahwa lapisan latar depan menghalangi fraksi (1 − α) cahaya yang masuk dari latar belakang dan menyumbang fraksi α darinya. warna sendiri untuk menggantikan apa yang telah dihapus. Contoh penggunaan Persamaan (3.2) ditunjukkan pada Gambar 3.14.

Karena bobot lapisan latar depan dan latar belakang berjumlah 1, warna tidak akan berubah jika lapisan latar depan dan latar belakang memiliki warna yang sama. Nilai untuk semua piksel dalam gambar mungkin disimpan dalam gambar skala abu-abu terpisah, yang kemudian dikenal sebagai topeng alfa atau topeng transparansi. Atau informasi dapat disimpan sebagai saluran keempat dalam gambar RGB, dalam hal ini disebut saluran alfa, dan gambar dapat disebut gambar RGBA. Dengan gambar 8-bit, setiap piksel membutuhkan 32 bit, yang merupakan potongan berukuran nyaman di banyak arsitektur komputer.

Meskipun Persamaan (3.2) adalah yang biasanya digunakan, ada berbagai situasi di mana digunakan secara berbeda (Porter & Duff, 1984).



Gambar 3.14 Contoh compositing menggunakan Persamaan (3.2). Gambar latar depan sebenarnya dipotong oleh saluran sebelum diletakkan di atas gambar latar belakang. Komposit yang dihasilkan ditampilkan di bagian bawah.

Penyimpanan Gambar

Sebagian besar format gambar RGB menggunakan delapan bit untuk setiap saluran merah, hijau, dan biru. Ini menghasilkan sekitar tiga megabita informasi mentah untuk satu juta piksel gambar. Untuk mengurangi kebutuhan penyimpanan, sebagian besar format gambar memungkinkan beberapa jenis kompresi. Pada tingkat tinggi, kompresi seperti itu bersifat lossless atau lossy. Tidak ada informasi yang dibuang dalam kompresi lossless, sementara beberapa informasi hilang tanpa dapat dipulihkan dalam sistem lossy. Format penyimpanan gambar yang populer meliputi:

- **jpeg.** Format lossy ini memampatkan blok gambar berdasarkan ambang batas dalam sistem visual manusia. Format ini bekerja dengan baik untuk gambar alami.
- **tiff.** Format ini paling sering digunakan untuk menyimpan gambar biner atau lossless dikompresi 8- atau 16-bit RGB meskipun banyak pilihan lain yang ada.
- **ppm.** Format lossless dan tidak terkompresi yang sangat sederhana ini paling sering digunakan untuk gambar RGB 8-bit meskipun ada banyak pilihan.
- **png.** Ini adalah seperangkat format lossless dengan seperangkat alat manajemen sumber terbuka yang bagus.

Karena kompresi dan varian, penulisan rutin input/output untuk gambar dapat dilibatkan. Untungnya seseorang biasanya dapat mengandalkan rutinitas perpustakaan untuk membaca dan menulis format file standar. Untuk aplikasi cepat dan kotor, di mana kesederhanaan dinilai di atas efisiensi, pilihan sederhana adalah menggunakan file ppm mentah, yang sering dapat ditulis hanya dengan membuang array yang menyimpan gambar dalam memori ke file, dengan menambahkan header yang sesuai.

Pertanyaan yang Sering Diajukan

- Mengapa mereka tidak membuat monitor linier dan menghindari semua bisnis gamma ini?

Idealnya 256 kemungkinan intensitas dari sebuah monitor harus terlihat memiliki jarak yang sama dibandingkan dengan jarak energi yang linier. Karena persepsi manusia tentang

intensitas itu sendiri non linier, gamma antara 1,5 dan 3 (tergantung pada kondisi tampilan) akan membuat intensitas kira-kira seragam dalam arti subjektif. Dengan cara ini, gamma adalah fitur. Jika tidak, pabrikan akan membuat monitor menjadi linier.

Latihan

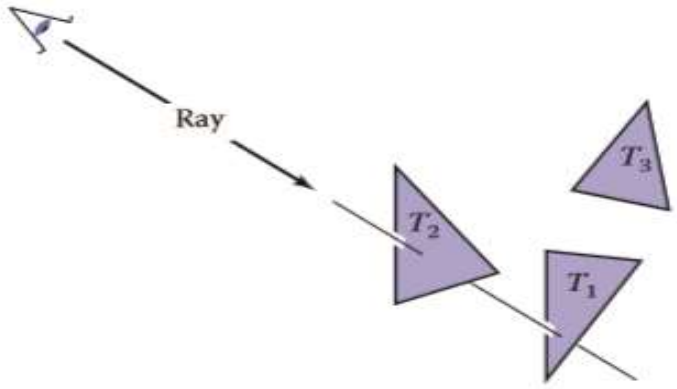
1. Simulasikan gambar yang diperoleh dari mosaik Bayer dengan mengambil gambar alami (sebaiknya foto yang dipindai daripada foto digital di mana mosaik Bayer mungkin sudah diterapkan) dan buat gambar skala abu-abu yang terdiri dari saluran merah/hijau/biru yang disisipkan. Ini mensimulasikan output mentah dari kamera digital. Sekarang buat gambar RGB yang sebenarnya dari output itu dan bandingkan dengan aslinya.

BAB 4
RAY TRACING

Salah satu tugas dasar grafisa komputer adalah merender objek tiga dimensi: mengambil scene, atau model, yang terdiri dari banyak objek geometris yang diatur dalam ruang 3D dan menghasilkan gambar 2D yang menunjukkan objek seperti yang dilihat dari sudut pandang tertentu. Ini adalah operasi yang sama yang telah dilakukan selama berabad-abad oleh arsitek dan insinyur yang membuat gambar untuk mengkomunikasikan desain mereka kepada orang lain.

Pada dasarnya, rendering adalah proses yang mengambil sebagai inputnya sekumpulan objek dan menghasilkan output berupa array piksel. Dengan satu atau lain cara, rendering melibatkan pertimbangan bagaimana setiap objek berkontribusi pada setiap piksel; itu dapat diatur dalam dua cara umum. Dalam rendering urutan objek, setiap objek dipertimbangkan secara bergantian, dan untuk setiap objek semua piksel yang dipengaruhinya ditemukan dan diperbarui. Dalam rendering urutan gambar, setiap piksel dipertimbangkan secara bergantian, dan untuk setiap piksel semua objek yang memengaruhinya ditemukan dan nilai piksel dihitung. Anda dapat memikirkan perbedaan dalam hal bersarangnya loop: dalam rendering urutan gambar, loop "untuk setiap piksel" ada di luar, sedangkan dalam rendering urutan objek, loop "untuk setiap objek" ada di luar. Jika outputnya adalah gambar vektor dan bukan gambar raster, rendering tidak harus melibatkan piksel, tetapi kami akan menganggap gambar raster dalam buku ini.

Pendekatan rendering urutan gambar dan urutan objek dapat menghitung gambar yang sama persis, tetapi mereka cocok untuk menghitung berbagai jenis efek dan memiliki karakteristik kinerja yang sangat berbeda. Kita akan mengeksplorasi kekuatan komparatif dari pendekatan-pendekatan di Bab 8 setelah kita membahas keduanya, tetapi, secara umum, pengurutan-gambaran lebih sederhana untuk bekerja dan lebih fleksibel dalam efek yang dapat dihasilkan, dan biasanya (walaupun tidak selalu) membutuhkan lebih banyak waktu eksekusi untuk menghasilkan gambar yang sebanding. Dalam ray tracer, mudah untuk menghitung bayangan dan pantulan yang akurat, yang canggung dalam kerangka urutan objek



Gambar 4.1 Sinar "dilacak" ke dalam scene dan objek pertama yang terkena adalah yang terlihat melalui piksel. Dalam hal ini, segitiga T2 dikembalikan.

Ray tracing adalah algoritme urutan gambar untuk membuat rendering scene 3D, dan kami akan mempertimbangkannya terlebih dahulu karena memungkinkan pelacak sinar bekerja tanpa mengembangkan mesin matematika apa pun yang digunakan untuk rendering urutan objek.

4.1 DASAR ALGORITMA RAY-TRACING

Pelacak sinar bekerja dengan menghitung satu piksel pada satu waktu, dan untuk setiap piksel tugas dasarnya adalah menemukan objek yang terlihat pada posisi piksel tersebut dalam gambar. Setiap piksel "memandang" ke arah yang berbeda, dan objek apa pun yang dilihat oleh piksel harus memotong sinar penglihatan, garis yang memancar dari sudut pandang ke arah yang dilihat piksel. Objek tertentu yang kita inginkan adalah objek yang memotong sinar penglihatan terdekat dengan kamera, karena objek tersebut menghalangi pandangan objek lain di belakangnya. Setelah objek ditemukan, perhitungan bayangan menggunakan titik potong, normal permukaan, dan informasi lainnya (tergantung pada jenis rendering yang diinginkan) untuk menentukan warna piksel. Hal ini ditunjukkan pada Gambar 4.1, di mana sinar memotong dua segitiga, tetapi hanya segitiga pertama yang terkena, T2, yang diarsir. Oleh karena itu, pelacak sinar dasar memiliki tiga bagian:

1. *generasi sinar,* yang menghitung asal dan arah sinar tampilan setiap piksel berdasarkan geometri kamera;
2. *perpotongan sinar,* yang menemukan objek terdekat yang memotong sinar penglihatan;
3. *shading,* yang menghitung warna piksel berdasarkan hasil perpotongan sinar.

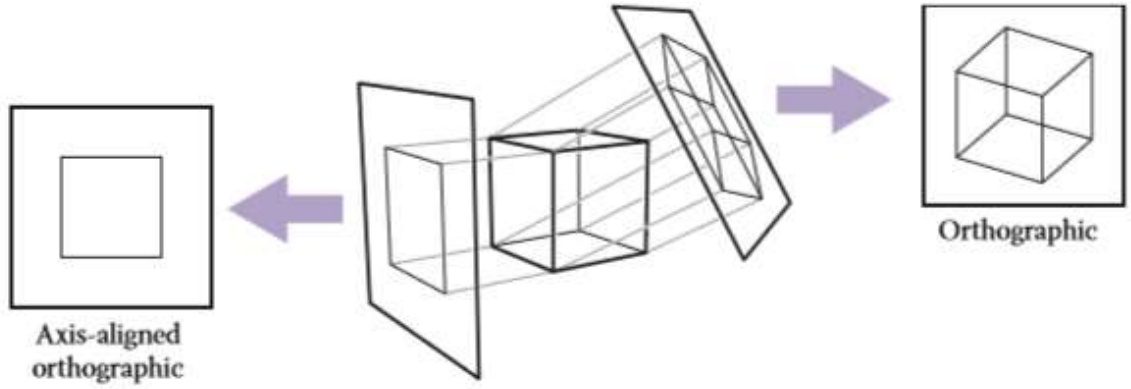
Struktur program ray tracing dasar adalah:

Untuk (for) setiap piksel **lakukan (do)**
hitung melihat sinar
temukan objek pertama yang terkena sinar dan permukaannya normal **n**
atur warna piksel ke nilai yang dihitung dari titik tekan, cahaya, dan **n**

Perspektif

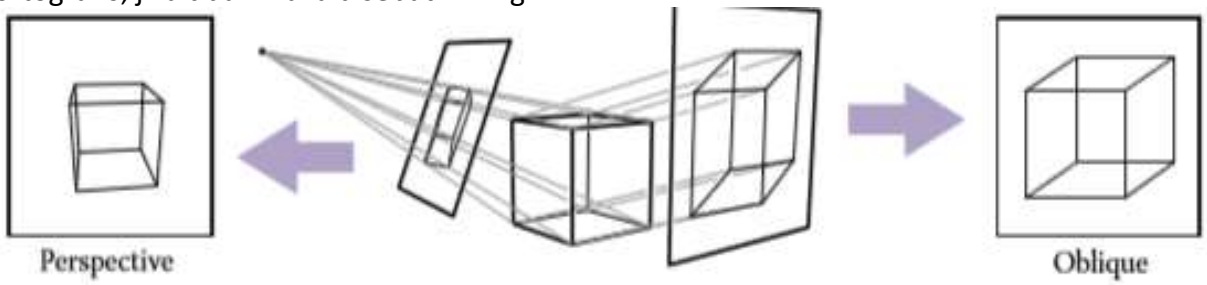
Masalah merepresentasikan objek atau scene 3D dengan gambar atau lukisan 2D dipelajari oleh seniman ratusan tahun sebelum komputer. Foto juga mewakili scene 3D dengan gambar 2D. Meskipun ada banyak cara yang tidak biasa untuk membuat gambar, dari lukisan kubisme hingga lensa mata ikan (Gambar 4.2) hingga kamera periferal, pendekatan standar untuk seni dan fotografi, serta grafis komputer, adalah perspektif linier, di mana objek 3D diproyeksikan ke bidang gambar sedemikian rupa sehingga garis lurus pada scene menjadi garis lurus pada gambar.

Gambar 4.2 Gambar yang diambil dengan lensa mata ikan bukanlah gambar perspektif linier.



Gambar 4.3 Ketika garis proyeksi sejajar dan tegak lurus dengan bidang gambar, pandangan yang dihasilkan disebut ortografi.

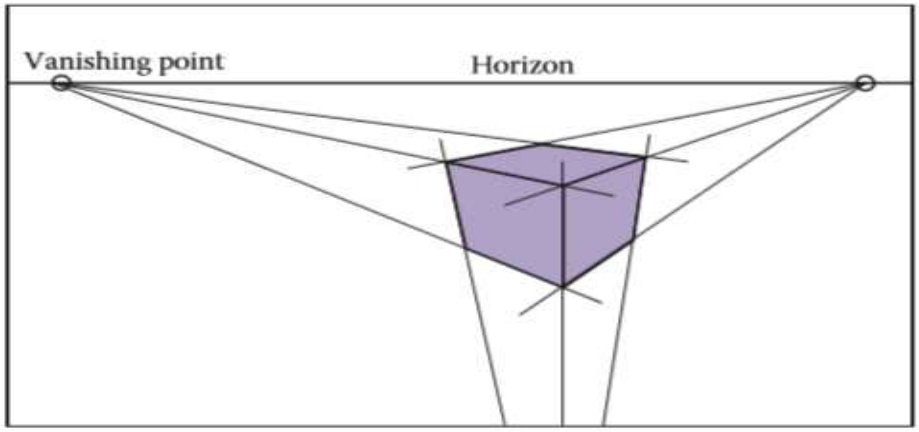
Jenis proyeksi yang paling sederhana adalah proyeksi paralel, di mana titik-titik 3D dipetakan ke 2D dengan menggerakkannya sepanjang arah proyeksi hingga mencapai bidang gambar (Gambar 4.3–4.4). Pandangan yang dihasilkan ditentukan oleh pilihan arah proyeksi dan bidang gambar. Jika bidang gambar tegak lurus dengan arah pandangan, proyeksi disebut ortografis; jika tidak maka disebut miring.



Gambar 4.4 Proyeksi sejajar yang bidang bayangannya membentuk sudut terhadap arah proyeksi disebut miring (kanan). Dalam proyeksi perspektif, semua garis proyeksi melewati sudut pandang, bukannya sejajar (kiri). Pandangan perspektif yang diilustrasikan tidak miring karena garis proyeksi yang ditarik melalui pusat gambar akan tegak lurus terhadap bidang gambar.

Proyeksi paralel sering digunakan untuk gambar mekanik dan arsitektur karena mempertahankan garis paralel sejajar dan mempertahankan ukuran dan bentuk objek planar yang sejajar dengan bidang gambar.

Keuntungan dari proyeksi paralel juga keterbatasannya. Dalam pengalaman kita sehari-hari (dan terlebih lagi dalam foto) objek terlihat lebih kecil saat semakin jauh, dan akibatnya garis paralel yang semakin menjauh tidak tampak paralel. Ini karena mata dan kamera tidak mengumpulkan cahaya dari satu arah pandang; mereka mengumpulkan cahaya yang melewati sudut pandang tertentu. Seperti yang telah diakui oleh para seniman sejak Renaisans, kita dapat menghasilkan pandangan yang tampak alami menggunakan proyeksi perspektif: kita hanya memproyeksikan sepanjang garis yang melewati satu titik, sudut pandang, daripada sepanjang garis paralel (Gambar 4.4). Dengan cara ini, objek yang lebih jauh dari sudut pandang secara alami menjadi lebih kecil ketika diproyeksikan. Pandangan perspektif ditentukan oleh pilihan sudut pandang (bukan arah proyeksi) dan bidang gambar. Seperti halnya pandangan paralel, ada pandangan perspektif miring dan tidak miring; perbedaan dibuat berdasarkan arah proyeksi di tengah gambar.



Gambar 4.5 Dalam perspektif tiga titik, seorang seniman memilih "titik hilang" di mana garis paralel bertemu. Garis horizontal sejajar akan bertemu pada satu titik di cakrawala. Setiap himpunan garis sejajar memiliki titik hilang sendiri-sendiri. Aturan-aturan ini diikuti secara otomatis jika kita menerapkan perspektif berdasarkan prinsip-prinsip geometris yang benar.

Anda mungkin telah belajar tentang konvensi artistik perspektif tiga titik, sebuah sistem untuk membangun pandangan perspektif secara manual (Gambar 4.5). Fakta mengejutkan tentang perspektif adalah bahwa semua aturan menggambar perspektif akan diikuti secara otomatis jika kita mengikuti aturan matematika sederhana yang mendasari perspektif: objek diproyeksikan langsung ke mata, dan digambar di tempat bertemu dengan bidang pandang di depan mata.

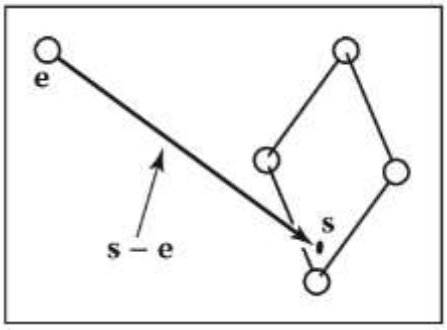
4.2 KOMPUTASI RAY

Dari bagian sebelumnya, alat dasar pembangkitan sinar adalah sudut pandang (atau arah pandang, untuk tampilan paralel) dan bidang gambar. Ada banyak cara untuk mengetahui detail geometri kamera; di bagian ini kami menjelaskan satu berdasarkan basis ortonormal yang mendukung pandangan paralel dan ortografis normal dan miring.

Untuk menghasilkan sinar, pertama-tama kita membutuhkan representasi matematis untuk sinar. Sinar sebenarnya hanyalah titik asal dan arah rambat; garis parametrik 3D sangat ideal untuk ini. Seperti dibahas dalam Bagian 2.5.7, garis parametrik 3D dari mata e ke titik s pada bidang gambar (Gambar 4.6) diberikan oleh

$$p(t) = e + t(s - e).$$

Ini harus ditafsirkan sebagai, "kita maju dari e sepanjang vektor (s−e) jarak pecahan t untuk menemukan titik p." Jadi diberikan t, kita dapat menentukan titik p. Titik e adalah asal sinar, dan se adalah arah sinar.



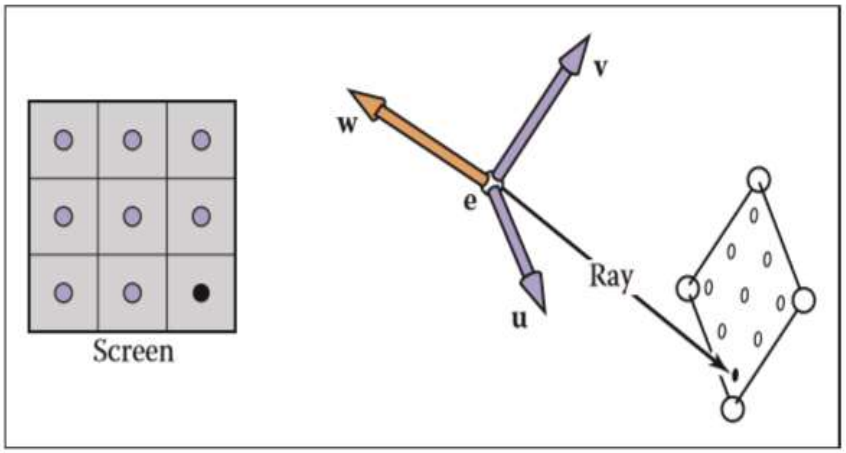
Gambar 4.6 Sinar dari mata ke suatu titik pada bidang bayangan.

Perhatian: kita membebani variabel t, yang merupakan parameter sinar dan juga koordinat v dari tepi atas gambar.

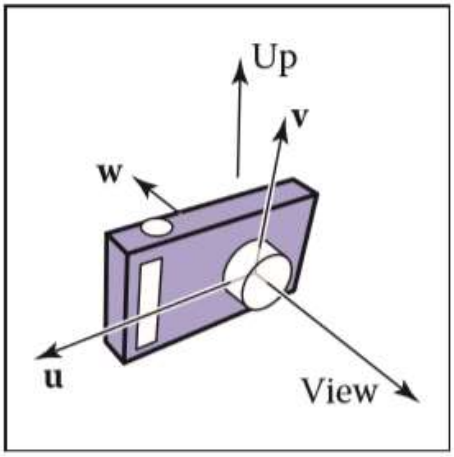
Perhatikan bahwa p(0) = e, dan p(1) = s, dan secara lebih umum, jika $0 < t_1 < t_2$, maka $p(t_1)$ lebih dekat ke mata daripada $p(t_2)$. Juga, jika t < 0, maka p(t) adalah "di belakang" mata. Fakta-fakta ini akan berguna ketika kita mencari objek terdekat yang terkena sinar yang tidak berada di belakang mata.

Untuk menghitung pancaran sinar, kita perlu mengetahui e (yang diberikan) dan s. Menemukan s mungkin tampak sulit, tetapi sebenarnya mudah jika kita melihat masalahnya pada sistem koordinat yang benar.

Metode generasi allofourray dimulai dari kerangka koordinat ortonormal yang dikenal sebagai bingkai kamera, yang akan dilambangkan dengan e, untuk titik mata, atau sudut pandang,andu, v,andw untuk tiga vektor basis, diorganisasikan dengan arah kanan upointing (dari pandangan kamera), v mengarah ke atas, dan w mengarah ke belakang, sehingga {u, v, w} membentuk sistem koordinat tangan kanan. Cara paling umum untuk membuat bingkai kamera adalah dari sudut pandang, yang menjadi e, arah pandang, yaitu w, dan vektor atas, yang digunakan untuk membangun basis yang memiliki v dan w pada bidang yang ditentukan oleh arah pandang dan arah atas, menggunakan proses untuk membangun basis ortonormal dari dua vektor yang dijelaskan dalam Bagian 2.4.7.



Gambar 4.7 Titik sampel di layar dipetakan ke larik serupa di windows 3D. Sinar penglihatan dikirim ke masing-masing lokasi ini.



Gambar 4.8 Vektor bingkai kamera, bersama dengan arah pandang dan arah atas. Vektor w berlawanan dengan arah pandangan, dan vektor v adalah coplanar dengan w dan vektor atas.

Tampilan Ortografis

Untuk tampilan ortografi, semua sinar akan memiliki arah−w. Meskipun tampilan paralel tidak memiliki sudut pandang semata, kita masih dapat menggunakan asal bingkai kamera untuk menentukan bidang di mana sinar mulai, sehingga memungkinkan objek berada di belakang kamera.

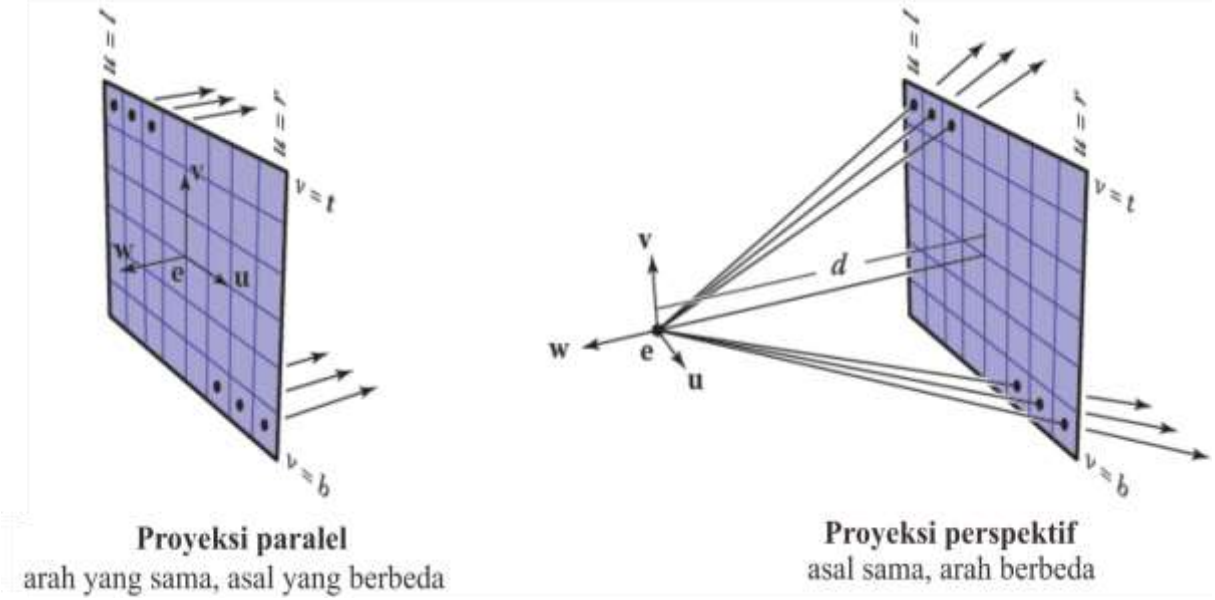
Sinar pandang harus dimulai pada bidang yang ditentukan oleh titik e dan vektor u dan v; satu-satunya informasi yang tersisa yang diperlukan adalah di mana di pesawat itu seharusnya gambar itu berada. Kita akan mendefinisikan dimensi gambar dengan empat angka, untuk keempat sisi gambar: l dan r adalah posisi tepi kiri dan kanan gambar, yang diukur dari e sepanjang arah u; dan b dan t adalah posisi tepi bawah dan tepi atas gambar, yang diukur dari e sepanjang arah v. Biasanya l < 0 < r dan b < 0 < t. (Lihat Gambar 4.9.)

Karena v dan w harus tegak lurus, vektor ke atas dan v umumnya tidak sama. Tetapi menyetel vektor atas agar mengarah lurus ke atas dalam scene akan mengarahkan kamera ke arah yang kita anggap "tegak". Tampaknya logis bahwa sinar pandang ortografis harus dimulai dari jarak yang tak terhingga, tetapi kemudian tidak mungkin untuk membuat pandangan ortografis suatu objek di dalam ruangan, misalnya. Banyak sistem berasumsi bahwa l = - r dan b = –t sehingga lebar dan tinggi cukup.

Pada Bagian 3.2 kita membahas koordinat piksel dalam sebuah gambar. Untuk memasukkan gambar dengan piksel nx × ny ke dalam persegi panjang berukuran (r – l) × (t – b), piksel diberi jarak $(r – l)/n_x$ terpisah secara horizontal dan $(t−b)/n_y$ terpisah vertikal, dengan ruang setengah piksel di sekitar tepi untuk memusatkan kisi piksel di dalam persegi panjang gambar. Artinya piksel pada posisi (i, j) pada gambar raster memiliki posisi

$$u = l +(r−l)(i +0 .5)/n_x,$$
$$v = b +(t−b)(j +0 .5)/n_y,$$

di mana (u, v) adalah koordinat posisi piksel pada bidang gambar, diukur terhadap titik asal e dan basis{u, v}.



Gambar 4.9 Generasi sinar menggunakan bingkai kamera. Kiri: Dalam tampilan ortografis, sinar mulai dari lokasi piksel pada bidang gambar, dan semuanya memiliki arah yang sama, yang sama dengan arah tampilan. Kanan: Dalam tampilan perspektif, sinar mulai dari titik pandang, dan setiap arah sinar ditentukan oleh garis yang melalui titik pandang, e, dan lokasi piksel pada bidang gambar.

Dalam tampilan ortografis, kita cukup menggunakan posisi bidang gambar piksel sebagai titik awal sinar, dan kita sudah tahu arah sinar adalah arah pandang. Prosedur untuk menghasilkan sinar pandang ortografis adalah:

Masukkan u dan v gunakan rumus (4.1)
ray.direction←−w
ray.origin←e + uu + v v

Dengan l dan r keduanya ditentukan, ada redundansi: memindahkan sudut pandang sedikit ke kanan dan dengan demikian mengurangi l dan r tidak akan mengubah tampilan (dan juga pada sumbu v). Sangat sederhana untuk membuat tampilan paralel miring: biarkan bidang gambar normal w ditentukan secara terpisah dari arah tampilan d. Prosedurnya kemudian persis sama, tetapi dengan d menggantikan w. Tentu saja w masih digunakan untuk mengkonstruksi u dan v.

Tampilan Perspektif

Untuk tampilan perspektif, semua sinar memiliki asal yang sama, pada sudut pandang; itu adalah arah yang berbeda untuk setiap piksel. Bidang gambar tidak lagi diposisikan pada e, melainkan pada jarak d di depan e; jarak ini adalah jarak bidang gambar, dari sepuluh secara longgar disebut panjang fokus, karena memilih d memainkan peran yang sama seperti memilih panjang fokus di kamera nyata. Arah setiap sinar ditentukan oleh titik pandang dan posisi piksel pada bidang gambar. Situasi ini diilustrasikan pada Gambar 4.9, dan prosedur yang dihasilkan mirip dengan prosedur ortografis:

Masukkan u dan v gunakan (4.1)
ray.direction←−dw + uu + vv
ray.origin←e

Seperti proyeksi paralel, pandangan perspektif miring dapat dicapai dengan menentukan bidang gambar normal secara terpisah dari arah proyeksi, kemudian mengganti dw dengan dd dalam ekspresi untuk arah sinar.

4.3 TITIK POTONG RAY-OBJECT

Setelah kita menghasilkan sinar e+td, selanjutnya kita perlu menemukan perpotongan pertama dengan objek di mana t > 0. Dalam praktiknya, ternyata berguna untuk memecahkan masalah yang sedikit lebih umum: temukan perpotongan pertama antara sinar dan permukaan yang terjadi pada t dalam interval $[t_0, t_1]$. Perpotongan sinar dasar adalah kasus di mana $t_0 = 0$ dan $t_1 = +\infty$. Kami memecahkan masalah ini untuk bola dan segitiga. Pada bagian berikutnya, beberapa objek dibahas.

Persimpangan Ray-Sphere

Diberikan sinar p(t) = e + td dan permukaan implisit f(p) = 0, kita ingin tahu di mana mereka berpotongan. Titik potong terjadi ketika titik-titik pada sinar memenuhi persamaan implisit, sehingga nilai t yang kita cari adalah yang menyelesaikan persamaan

$$f(p(t)) = 0 \text{ atau } f(e + td) = 0$$

Sebuah bola dengan pusat $c = (x_c, y_c, z_c)$ dan jari-jari R dapat diwakili oleh persamaan implisit

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0 .$$

Kita dapat menulis persamaan yang sama ini dalam bentuk vektor:

$$(p - c) \cdot (p - c) - R^2 = 0 .$$

Setiap titik p yang memenuhi persamaan ini berada pada bola. Jika kita memasukkan titik-titik pada sinar p(t) = e + td ke dalam persamaan ini, kita mendapatkan persamaan dalam bentuk t yang dipenuhi oleh nilai-nilai t yang menghasilkan titik-titik pada bola:

$$(e + td - c) \cdot (e + td - c) - R^2 = 0$$

Menata ulang hasil istilah

$$(d \cdot d)t^2 + 2d \cdot (e - c)t + (e - c) \cdot (e - c) - R^2 = 0 .$$

Di sini, semuanya diketahui kecuali parameter t, jadi ini adalah persamaan kuadrat klasik di t, artinya memiliki bentuk

$$At^2 + Bt + C = 0$$

Solusi untuk persamaan ini dibahas dalam Bagian 2.2. Istilah di bawah tanda akar kuadrat dalam solusi kuadrat, $B^2 - 4AC$, disebut diskriminan dan memberi tahu kita berapa banyak solusi nyata yang ada. Jika diskriminan negatif, akar kuadratnya imajiner dan garis dan bola tidak berpotongan. Jika diskriminan positif, ada dua solusi: satu solusi di mana sinar memasuki bola dan satu lagi keluar. Jika diskriminan adalah nol, sinar menyerempet bola, menyentuhnya tepat di satu titik. Memasukkan suku sebenarnya untuk bola dan membatalkan faktor dua, kita mendapatkan

$$t = \frac{-d \,.\, (e - c) \pm \sqrt{((d \,.\, (e - c))^2 - (d \,.\, d)((e - c).\,(e - c) - R^2)}}{(d \,.\, d)}$$

Dalam implementasi yang sebenarnya, Anda harus terlebih dahulu memeriksa nilai diskriminan sebelum menghitung suku-suku lain. Jika bola hanya digunakan sebagai objek pembatas untuk objek yang lebih kompleks, maka kita hanya perlu menentukan apakah kita mengenainya; memeriksa kecukupan diskriminan. Sebagaimana dibahas dalam Bagian 2.5.4, vektor normal pada titik p diberikan oleh gradien n = 2(p − c). Satuan normalis (p − c)/R.

Persimpangan Sinar-Segitiga

Ada banyak algoritma untuk menghitung perpotongan sinar-segitiga. Kami akan menyajikan bentuk yang menggunakan koordinat barycentric untuk bidang parametrik yang berisi segitiga, karena tidak memerlukan penyimpanan jangka panjang selain simpul segitiga (Snyder & Barr, 1987). Untuk memotong sinar dengan permukaan parametrik, kami membuat sistem persamaan di mana semua koordinat Cartesian cocok:

$$\left.\begin{matrix} x_e + tx_d \\ y_e + ty_d \\ z_e + tz_d \end{matrix}\right\} \quad atau, \;\; e + td = f(u, v)$$

Di sini, kami memiliki tiga persamaan dan tiga yang tidak diketahui (t, u, dan v), sehingga kami dapat menyelesaikan secara numerik untuk yang tidak diketahui. Jika kita beruntung, kita bisa menyelesaikannya secara analitis. Dalam kasus di mana permukaan parametrik adalah bidang parametrik, persamaan parametrik dapat ditulis dalam bentuk vektor seperti yang dibahas dalam Bagian 2.7.2. Jika titik sudut segitiga adalah a, b, dan c, maka perpotongan akan terjadi ketika

$$e + td = a + \beta(b - a) + \gamma(c - a),$$

Untuk *t*, β, dan γ. Persimpangan p akan berada pada e + td seperti terlihat pada Gambar 4.10. Sekali lagi, dari Bagian 2.7.2, kita mengetahui bahwa perpotongan berada di dalam segitiga jika dan hanya jika $\beta > 0$, $\gamma > 0$, dan $\beta + \gamma < 1$. Jika tidak, sinar telah mengenai bidang di luar segitiga, sehingga meleset dari segitiga. Jika tidak ada solusi, segitiga tersebut mengalami degenerasi atau sinar sejajar dengan bidang yang memuat segitiga. Untuk menyelesaikan *t, β,* dan *γ* dalam Persamaan (4.2), kami memperluasnya dari bentuk vektornya ke dalam tiga persamaan untuk tiga koordinat:

$$x_e + tx_d = x_a + \beta(x_b - x_a) + \gamma(x_c - x_a),$$
$$y_e + ty_d = y_a + \beta(y_b - y_a) + \gamma(y_c - y_a),$$
$$z_e + tz_d = z_a + \beta(z_b - z_a) + \gamma(z_c - z_a).$$

Ini dapat ditulis ulang sebagai sistem linier standar:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

Metode klasik tercepat untuk menyelesaikan sistem linier 3 x 3 ini adalah aturan Cramer. Ini memberi kita solusi

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|A|}$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|A|}$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_e \\ y_a - y_b & y_a - y_c & y_e \\ z_a - z_b & z_a - z_c & z_e \end{vmatrix}}{|A|}$$

dimana matriks A adalah

$$A = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix}$$

dan |A| menunjukkan determinan A. Determinan 3 × 3 memiliki subterm umum yang dapat dieksploitasi. Melihat sistem linier dengan variabel dummy

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix}$$

Aturan Cramer memberi kita

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg).}{M}$$

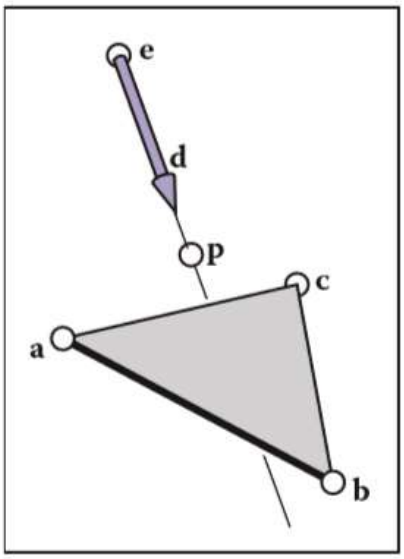
$$\gamma = \frac{i(ak - jb) + h(jc - al) + d(bl - eg).}{M}$$

$$M = \frac{j(ak - jb) + e(jc - al) + c(bl - kc).}{M}$$

di mana

M = a(ei − hf) + b(gf − di) + c(dh − eg).

Kita dapat mengurangi jumlah operasi dengan menggunakan kembali angka seperti *“ei-minus-hf.”*



Gambar 4.10 Sinar mengenai bidang yang memuat segitiga di titik p.

Algoritme untuk persimpangan segitiga-segitiga yang membutuhkan solusi linier dapat memiliki beberapa kondisi untuk terminasi dini. Jadi, fungsinya akan terlihat seperti:

boolean raytri (ray r, vector3 a, vector3 b, vector3 c, interval [t_0, t_1])
Input t
If ($t < t_0$) atau ($t > t_1$) **then**
Return salah
Input γ **if** (γ<0) atau (γ>1) **then**
return salah
input β
if (β<0) dan (β>1−γ) t**h**en
return salah
return benar

Persimpangan Ray-Poligon

Diberikan poligon planar dengan m simpul p1 melalui pm dan normal permukaan n, pertama-tama kita menghitung titik potong antara sinar e + td dan bidang yang memuat poligon dengan persamaan implisit

$$(p - p1) \cdot n = 0 .$$

Kami melakukan ini dengan menetapkan p = e + td dan memecahkan t untuk mendapatkan

$$t = \frac{(p_1 - e) \,.\, n}{d \,.\, n}$$

Hal ini memungkinkan kita untuk menghitung p. Jika p berada di dalam poligon, maka sinar itu mengenainya; jika tidak, tidak.

Kita dapat menjawab pertanyaan apakah p berada di dalam poligon dengan memproyeksikan titik dan sudut poligon ke bidang xy dan menjawabnya di sana. Cara termudah untuk melakukannya adalah dengan mengirimkan sinar 2D keluar dari atas dan menghitung jumlah perpotongan antara sinar itu dan batas poligon (Sutherland, Sproull, & Schumacker, 1974; Glassner, 1989). Jika jumlah perpotongannya ganjil, maka titik tersebut berada di dalam poligon; sebaliknya tidak. Hal ini benar karena sinar yang masuk harus keluar, sehingga menciptakan sepasang perpotongan. Hanya sinar yang dimulai di dalam tidak akan

membuat pasangan seperti itu. Untuk mempermudah perhitungan, sinar 2D mungkin juga merambat sepanjang sumbu x:

$$\begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} x_p \\ y_p \end{bmatrix} + s \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Lurus ke depan untuk menghitung perpotongan sinar itu dengan tepi seperti (x_1, y_1, x_2, y_2) untuk s ∈ (0, ∞)

Masalah muncul, bagaimanapun, untuk poligon yang proyeksi ke bidang xy adalah garis. Untuk menyiasatinya, kita dapat memilih di antara bidang *xy, yz,* atau *zx* mana yang terbaik. Jika kami menerapkan poin kami untuk memungkinkan operasi pengindeksan, misalnya, p(0) = x_p maka ini dapat dicapai sebagai berikut:

if (abs(z_n) > abs(x_n)) dan (abs(z_n) > abs(y_n)) **then**
    index0 = 0
    index1 = 1
else if (abs(y_n) > abs (x_n)) **then**
    index0 = 0
    index1= 2
else
    index0 = 1
    index1 = 2

Sekarang, semua komputasi dapat menggunakan p(index0) daripada x_p, dan seterusnya.

Pendekatan lain untuk poligon, yang sering digunakan dalam praktik, adalah menggantinya dengan beberapa segitiga.

Memotong Sekelompok Objek

Tentu saja, sebagian besar scene yang menarik terdiri dari lebih dari satu objek, dan ketika kita memotong sinar dengan scene itu, kita hanya harus menemukan perpotongan terdekat dengan kamera di sepanjang sinar itu. Cara sederhana untuk mengimplementasikan ini adalah dengan memikirkan sekelompok objek sebagai dirinya sendiri sebagai jenis objek lain. Untuk memotong sinar dengan satu grup, Anda cukup memotong sinar dengan objek dalam grup dan mengembalikan perpotongan dengan nilai t terkecil. Kode berikut menguji hit dalam interval t [t_0, t_1]:

hit = salah
untuk (for)setiap objek o dalam grup **lakukan (do)**
jika (o terkena sinar parameter t dan t ∈ [t_0, t_1]) **maka**
tekan = benar
hit objek = o
t_1 = t
pukulan balik

Gambar 4.11 Scene sederhana yang dirender dengan hanya pembangkitan sinar dan perpotongan permukaan, tetapi tanpa bayangan; setiap piksel hanya diatur ke warna tetap tergantung pada objek mana yang terkena.

4.4 SHADING

Setelah permukaan yang terlihat untuk suatu piksel diketahui, nilai piksel dihitung dengan mengevaluasi model bayangan. Bagaimana ini dilakukan sepenuhnya bergantung pada aplikasi—metode berkisar dari heuristik yang sangat sederhana hingga komputasi numerik yang rumit. Dalam bab ini kami menjelaskan dua model bayangan paling dasar; model yang lebih maju dibahas dalam Bab 10.

Sebagian besar model bayangan, satu atau lain cara, dirancang untuk menangkap proses pantulan cahaya, di mana permukaan diterangi oleh sumber cahaya dan memantulkan sebagian cahaya ke kamera. Model shading sederhana didefinisikan dalam hal iluminasi dari sumber cahaya titik. Variabel penting dalam pemantulan cahaya adalah arah cahaya l, yang merupakan vektor satuan yang mengarah ke sumber cahaya; arah pandang v, yang merupakan vektor satuan yang mengarah ke mata atau kamera; normal permukaan n, yang merupakan vektor satuan tegak lurus terhadap permukaan pada titik di mana pemantulan berlangsung; dan karakteristik permukaan—warna, kilau, atau sifat lainnya tergantung pada model tertentu.

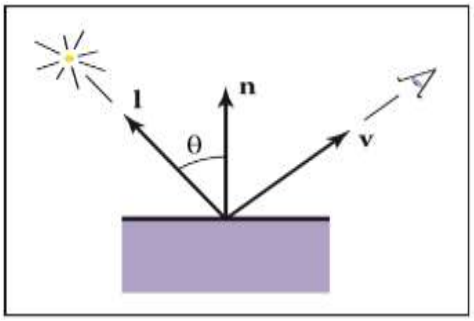
Bayangan Lambertian

Model bayangan paling sederhana didasarkan pada pengamatan yang dilakukan oleh Lambert pada abad ke-18: jumlah energi dari sumber cahaya yang jatuh pada suatu luas permukaan tergantung pada sudut permukaan terhadap cahaya. Permukaan yang menghadap langsung ke arah cahaya menerima penerangan maksimum; permukaan yang bersinggungan dengan arah cahaya (atau menghadap jauh dari cahaya) tidak menerima penerangan; dan di antara iluminasi sebanding dengan kosinus sudut antara permukaan normal dan sumber cahaya (Gambar 4.12). Ini mengarah ke model shading Lambertian:

$$L = k_d I \max(0, n \cdot l)$$

di mana L adalah warna piksel; kd adalah koefisien difus, atau warna permukaan; dan I adalah intensitas sumber cahaya. Karena n dan l adalah vektor satuan, kita dapat menggunakan $n \cdot l$ sebagai singkatan (baik di atas kertas maupun dalam kode) untuk $\cos\theta$. Persamaan ini (seperti persamaan bayangan lainnya di bagian ini) berlaku secara terpisah untuk tiga saluran warna, sehingga komponen merah dari nilai piksel adalah produk dari komponen difusi merah, intensitas sumber cahaya merah, dan produk titik; hal yang sama berlaku untuk hijau dan biru.

Penerangan dari sumber titik nyata jatuh saat jarak dikuadratkan, tetapi seringkali lebih banyak masalah daripada nilainya dalam penyaji sederhana.



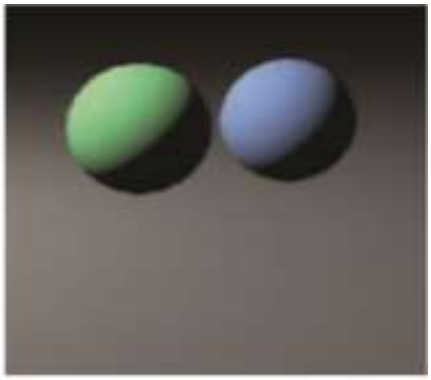
Gambar 4.12 Geometri untuk naungan Lambertian.

Jika ragu, buat sumber cahaya berwarna netral, dengan intensitas merah, hijau, dan biru yang sama. Vektor l dihitung dengan mengurangkan titik potong sinar dan permukaan dari posisi sumber cahaya. Jangan lupa bahwa v, l, dan n semuanya harus merupakan vektor satuan; kegagalan untuk menormalkan vektor-vektor ini merupakan kesalahan yang sangat umum dalam perhitungan bayangan.

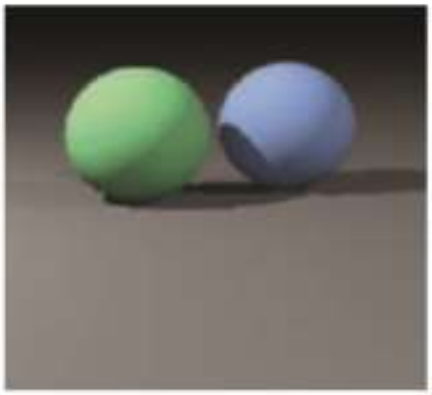
Blinn-Phong Shading

Bayangan Lambertian adalah tampilan independen: warna permukaan tidak tergantung pada arah dari mana Anda melihat. Banyak permukaan nyata menunjukkan beberapa derajat kilau, menghasilkan sorotan, atau pantulan spekular, yang tampak bergerak saat sudut pandang berubah. Bayangan Lambertian tidak menghasilkan sorotan apa pun dan mengarah ke tampilan yang sangat matte, berkapur, dan banyak model bayangan menambahkan komponen spekuler ke naungan Lambertian; Partisi Lambertian kemudian komponen difus.

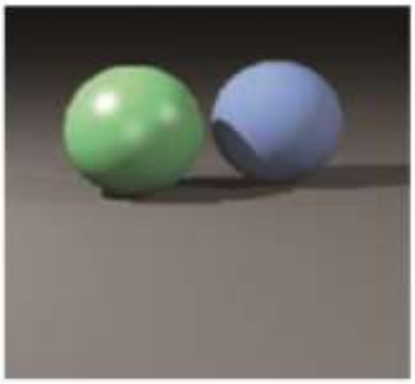
Model yang sangat sederhana dan banyak digunakan untuk sorotan spekular diusulkan oleh Phong(Phong, 1975) dan kemudian diperbarui oleh Blinn (J. F. Blinn, 1976) ke bentuk yang paling umum digunakan saat ini. Idenya adalah untuk menghasilkan refleksi yang paling terang ketika v dan l diposisikan secara simetris melintasi permukaan normal, yaitu ketika refleksi cermin akan terjadi; pantulan kemudian berkurang dengan mulus saat vektor menjauh dari konfigurasi cermin.



Gambar 4.13 Scene sederhana yang dirender dengan bayangan difus dari satu sumber cahaya.

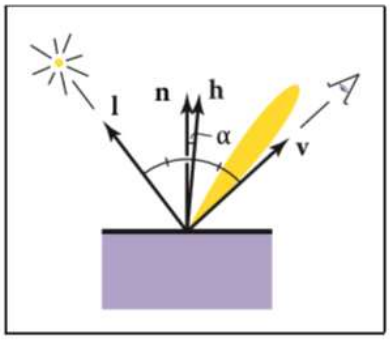


Gambar 4.14 Scene sederhana yang dirender dengan bayangan dan bayangan yang menyebar (Bagian 4.7) dari tiga sumber cahaya.



Gambar 4.15 Scene sederhana yang dirender dengan bayangan difus (bola biru), bayangan BlinnPhong (bola hijau), dan bayangan dari tiga sumber cahaya.

Kita dapat mengetahui seberapa dekat kita dengan konfigurasi cermin dengan membandingkan setengah vektor h (pembagi sudut antara v dan l) dengan permukaan normal (Gambar 4.16). Jika vektor setengah dekat permukaan normal, komponen specular harus cerah; jika jauh itu harus redup. Hasil ini diperoleh dengan menghitung perkalian titik antara h dan n (ingat mereka adalah vektor satuan, jadi n·h mencapai maksimum 1 jika vektornya sama), kemudian mengambil hasil pangkat p > 1 untuk membuatnya berkurang lebih cepat. Kekuatan, atau eksponen Phong, mengontrol kilau permukaan yang tampak. Setengah vektor itu sendiri mudah dihitung: karena v dan l adalah panjang yang sama, jumlah mereka adalah vektor yang membagi dua sudut di antara mereka, yang hanya perlu dinormalisasi untuk menghasilkan h.



Gambar 4.16 Geometri untuk naungan Blinn-Phong.

Menempatkan ini semua bersama-sama, model Blinn-Phongshading adalah sebagai berikut:

$$h = \frac{v + 1}{||v + 1||}$$

$$L = k_d I \max(0, n \cdot 1) + k_s I \max(0, n \cdot h)^p$$

di mana ks adalah koefisien specular, atau warna specular, dari permukaan.
Nilai khas p:
10 — “kulit telur”;
100 — sedikit mengkilat;
1000—sangat mengkilap;
10.000—hampir seperti cermin.

Jika ragu, buat warna specular menjadi abu-abu, dengan nilai merah, hijau, dan biru yang sama.

Bayangan Sekitar

Permukaan yang tidak menerima iluminasi sama sekali akan dibuat hitam pekat, yang seringkali tidak diinginkan. Heuristik kasar tetapi berguna untuk menghindari bayangan hitam adalah dengan menambahkan komponen konstan ke model bayangan, yang kontribusinya pada warna piksel hanya bergantung pada objek yang dipukul, tanpa ketergantungan pada geometri permukaan sama sekali. Ini dikenal sebagai bayangan ambien—seolah-olah permukaan diterangi oleh cahaya "ambien" yang datang secara merata dari mana-mana. Untuk kenyamanan dalam menyetel parameter, bayangan ambien biasanya dinyatakan sebagai produk dari warna permukaan dengan warna cahaya ambien, sehingga bayangan ambien dapat disetel untuk permukaan satu per satu atau untuk semua permukaan bersama-sama. Bersama dengan model Blinn-Phong lainnya, ambientshading melengkapi versi lengkap dari model shading yang sederhana dan berguna:

$$L = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^n,$$

di mana ka adalah koefisien ambien permukaan, atau "warna ambien", dan Ia adalah intensitas cahaya ambien.

Jika ragu, atur warna sekitar agar sama dengan warna difus. Di dunia nyata, permukaan yang tidak diterangi oleh sumber cahaya diterangi oleh pantulan tidak langsung dari permukaan lain.

4.5 PROGRAM RAY-TRACING

Kita sekarang tahu bagaimana menghasilkan sinar penglihatan untuk piksel tertentu, bagaimana menemukan persimpangan terdekat dengan objek, dan bagaimana membuat bayangan persimpangan yang dihasilkan. Ini semua adalah bagian yang diperlukan untuk program yang menghasilkan gambar berbayang dengan permukaan tersembunyi dihilangkan.

(for) untuk setiap piksel **lakukan (do)**
hitung sinar tampak
if (sinar menumbuk suatu benda dengan t [0,∞)) **then**
Hitung n
Evaluasi model bayangan dan atur piksel ke warna itu
else
atur warna piksel ke warna latar belakang

Di sini pernyataan "jika sinar mengenai suatu objek..." dapat diimplementasikan dengan menggunakan algoritma dari Bagian 4.4.4.

Dalam implementasi aktual, rutinitas persimpangan permukaan perlu mengembalikan referensi ke objek yang terkena, atau setidaknya vektor normal dan sifat material yang relevan dengan naungan. Hal ini sering dilakukan dengan melewatkan record/struktur dengan informasi tersebut. Dalam implementasi berorientasi objek, adalah ide yang baik untuk memiliki kelas yang disebut sesuatu seperti permukaan dengan kelas turunan segitiga, bola, grup, dll. Apa pun yang dapat berpotongan dengan sinar akan berada di bawah kelas itu. Program ray-tracing kemudian akan memiliki satu referensi ke "permukaan" untuk keseluruhan model, dan jenis objek dan struktur efisiensi baru dapat ditambahkan secara transparan.

Desain Berorientasi Objek untuk Program Pelacakan Sinar

Seperti disebutkan sebelumnya, hierarki kelas kunci dalam ray tracer adalah objek geometris yang membentuk model. Ini harus menjadi subkelas dari beberapa kelas objek geometris, dan mereka harus mendukung fungsi hit (Kirk & Arvo, 1988). Untuk menghindari kebingungan dari penggunaan kata "objek", permukaan adalah nama kelas yang sering digunakan. Dengan kelas seperti itu, Anda dapat membuat pelacak sinar yang memiliki

antarmuka umum yang mengasumsikan sedikit tentang pemodelan primitif dan men-debugnya hanya menggunakan bola. Poin penting adalah bahwa apa pun yang dapat "ditabrak" oleh sinar harus menjadi bagian dari hierarki kelas ini, misalnya, bahkan kumpulan permukaan harus dianggap sebagai subkelas dari kelas permukaan. Ini termasuk struktur efisiensi, seperti hierarki volume pembatas; mereka dapat terkena sinar, sehingga mereka berada di kelas.

Misalnya, kelas "*abstrak*" atau "*dasar*" akan menentukan fungsi hit serta fungsi kotak pembatas yang akan berguna nanti:

permukaan kelas

hit bool virtual (ray e + td, t nyata0, t nyata1, rekaman hit)

kotak pembatas-kotak virtual ()

Di sini (t_0, t_1) adalah interval pada sinar di mana hit akan dikembalikan, dan rec adalah catatan yang dilewatkan dengan referensi; ini berisi data seperti t di persimpangan ketika hit kembali benar. Jenis kotak adalah "kotak pembatas" 3D, yaitu dua titik yang mendefinisikan kotak sejajar sumbu yang menutupi permukaan. Misalnya, untuk bola, fungsinya akan diimplementasikan oleh

kotak bola :: kotak pembatas ()

vector3 min = pusat−vektor3 (jari-jari, jari-jari, jari-jari)

vector3max = pusat + vector3 (jari-jari, jari-jari, jari-jari) kotak

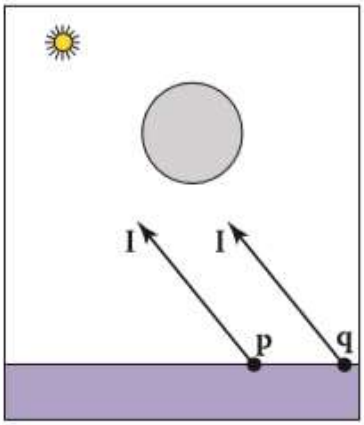
return (min, maks)

Kelas lain yang berguna adalah materi. Ini memungkinkan Anda untuk mengabstraksi perilaku material dan kemudian menambahkan materi secara transparan. Cara sederhana untuk menghubungkan objek dan material adalah dengan menambahkan pointer ke material di kelas permukaan, meskipun perilaku yang lebih dapat diprogram mungkin diinginkan. Sebuah pertanyaan besar adalah apa yang harus dilakukan dengan tekstur; apakah mereka bagian dari kelas materi atau apakah mereka tinggal di luar kelas materi? Ini akan dibahas lebih lanjut di Bab 11.

4.6 SHADOW (BAYANGAN)

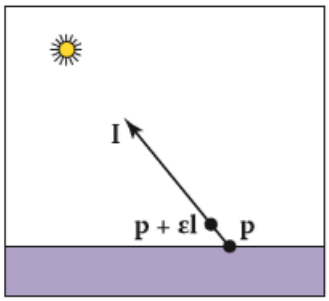
Setelah Anda memiliki program ray tracing dasar, bayangan dapat ditambahkan dengan sangat mudah. Ingat dari Bagian 4.5 bahwa cahaya datang dari beberapa arah l. Jika kita membayangkan diri kita berada di titik p pada permukaan yang diarsir, titik tersebut berada dalam bayangan jika kita “memandang” ke arah dan melihat suatu benda. Jika tidak ada objek, maka lampu tidak terhalang.

Hal ini ditunjukkan pada Gambar 4.17, di mana sinar p + tl tidak mengenai benda apapun dan dengan demikian tidak berada dalam bayangan. Titik q berada dalam bayangan karena sinar q + tl mengenai suatu benda. Vektor l adalah sama untuk kedua titik karena cahayanya “jauh”. Asumsi ini nantinya akan dilonggarkan. Sinar yang menentukan masuk atau keluar bayangan disebut sinar bayangan untuk membedakannya dari sinar pandang



Gambar 4.17 Titik p tidak berada dalam bayangan, sedangkan titik q berada dalam bayangan.

Untuk mendapatkan algoritme untuk bayangan, kami menambahkan pernyataan anif untuk menentukan apakah titik tersebut berada dalam bayangan. Dalam implementasi naif, sinar bayangan akan memeriksa t ∈ [0,∞), tetapi karena ketidaktepatan numerik, ini dapat menghasilkan perpotongan dengan permukaan di mana p terletak. Sebagai gantinya, penyesuaian yang biasa dilakukan untuk menghindari masalah tersebut adalah dengan menguji t∈ [,∞) di mana beberapa konstanta positif kecil (Gambar 4.18).



Gambar 4.18 Dengan menguji pada interval mulai dari e, kita menghindari ketidaktepatan numerik yang menyebabkan sinar mengenai permukaan p adalah on

Jika kita menerapkan sinar bayangan untuk pencahayaan Phong dengan Persamaan 4.3 maka kita memiliki yang berikut:

fungsi raycolor(ray e + td, real t0, real t1)
rekaman-rekaman, srec
if (scene→hit(e + td, t0, t1, rec)) **then**
p = e +(rek.t)d
warna c = rec.ka Ia
if (bukan scene→hit(p + sl, ,∞, srec)) **then**
vektor3 h = dinormalisasi(dinormalisasi(l)+dinormalisasi(−d))
c = c +rec.kd I max(0,rec.n·l)+(rec.ks)I (rec.n·h)rec.p
return c
else
return warna latar belakang

Perhatikan bahwa warna sekitar ditambahkan apakah p berada dalam bayangan atau tidak. Jika ada beberapa sumber cahaya, kami dapat mengirim sinar bayangan sebelum mengevaluasi model bayangan untuk setiap cahaya. Kode di atas mengasumsikan bahwa d

dan l belum tentu vektor satuan. Ini penting untuk d, khususnya, jika kita ingin menambahkan instance dengan rapi nanti (lihat Bagian 13.2).

4.7 REFLEKSI SPEKULER IDEAL

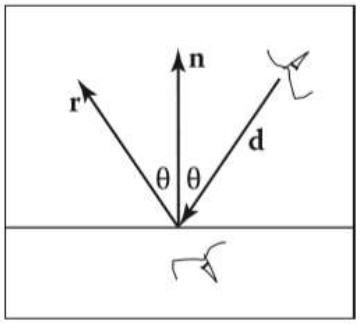
Sangat mudah untuk menambahkan pantulan spekular ideal, atau pantulan cermin, ke program penelusuran garis. Observasi kunci ditunjukkan pada Gambar 4.19 di mana penonton yang melihat dari arah e melihat apa yang ada di arah r seperti yang terlihat dari permukaan. Vektor r ditemukan dengan menggunakan varian dari Persamaan refleksi pencahayaan Phong (10.6). Ada perubahan tanda karena vektor d menunjuk ke permukaan dalam hal ini, jadi,

$$r = d - 2(d \cdot n)n.$$

Di dunia nyata, sebagian energi hilang ketika cahaya dipantulkan dari permukaan, dan kehilangan ini bisa berbeda untuk warna yang berbeda. Misalnya, emas memantulkan kuning lebih efisien daripada biru, sehingga menggeser warna objek yang dipantulkan. Ini dapat diimplementasikan dengan menambahkan panggilan rekursif di raycolor:

$$\text{Warna } c = c + k_m \text{ ray color}(p + sr, \epsilon, \infty)$$

di mana k_m (untuk "refleksi cermin") adalah warna RGB specular. Kita perlu memastikan bahwa kita menguji $s \in [\epsilon, \infty)$ untuk alasan yang sama seperti yang kita lakukan dengan sinar bayangan; kita tidak ingin sinar pantul mengenai objek yang menghasilkannya.



Gambar 4.19 Saat melihat ke cermin yang sempurna, pemirsa yang melihat ke arah d akan melihat apa pun yang dilihat pemirsa "di bawah" permukaan ke arah r.

Gambar 4.20 Scene sederhana yang dirender dengan bayangan difus dan Blinn-Phong, bayangan dari tiga sumber cahaya, dan pantulan spekular dari lantai.

Masalah dengan panggilan rekursif di atas adalah bahwa hal itu mungkin tidak akan pernah berakhir. Misalnya, jika sebuah sinar dimulai di dalam ruangan, itu akan memantul selamanya. Ini dapat diperbaiki dengan menambahkan kedalaman rekursi maksimum. Kode akan lebih efisien jika sinar pantul yang dihasilkan hanya jika km bukan nol (hitam).

4.8 CATATAN SEJARAH

Ray tracing dikembangkan pada awal sejarah komputer grafis (Appel, 1968) tetapi tidak banyak digunakan sampai daya komputasi yang cukup tersedia (Kay & Greenberg, 1979; Whitted, 1980). Ray tracing memiliki kompleksitas waktu asimtotik yang lebih rendah daripada rendering urutan objek dasar (Snyder & Barr, 1987; Muuss, 1995; S. Parker et al., 1999; Wald, Slusallek, Benthin, & Wagner, 2001). Meskipun secara tradisional dianggap sebagai metode offline, implementasi ray tracing real-time menjadi semakin umum.

Pertanyaan yang Sering Ditanyakan

- *Mengapa tidak ada matriks perspektif dalam ray tracing?*
  Matriks perspektif dalam buffer-z ada sehingga kita dapat mengubah proyeksi perspektif menjadi proyeksi paralel. Ini tidak diperlukan dalam ray tracing, karena mudah untuk melakukan proyeksi perspektif secara implisit dengan mengipasi sinar keluar dari mata.
- *Dapatkah ray tracing dibuat interaktif?*
  Untuk model dan gambar yang cukup kecil, PC modern mana pun cukup kuat untuk ray tracing menjadi interaktif. Dalam praktiknya, beberapa CPU dengan buffer bingkai bersama diperlukan untuk implementasi layar penuh. Kekuatan komputer meningkat jauh lebih cepat daripada resolusi layar, dan hanya masalah waktu sebelum PC konvensional dapat melacak scene kompleks pada resolusi layar.
- *Apakah ray tracing berguna dalam program grafis perangkat keras?*
  Ray tracing sering digunakan untuk memilih. Saat pengguna mengklik mouse pada piksel dalam program grafis 3D, program perlu menentukan objek mana yang terlihat di dalam piksel tersebut. Ray tracing adalah cara yang ideal untuk menentukan itu.

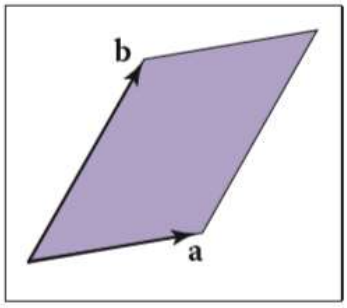
Latihan

1. Berapakah parameter sinar titik potong antara sinar(1, 1, 1) + t(−1, 1, 1) dan bola yang berpusat di titik asal dengan jari-jari 1? Catatan: ini adalah kasus debug yang bagus.
2. Apa koordinat barycentric dan parameter sinar di mana sinar (1, 1, 1) + t(−1, −1, −1) menumbuk segitiga dengan simpul (1, 0, 0), (0, 1, 0), dan (0, 0, 1)? Catatan: ini adalah kasus debug yang bagus.
3. Lakukan penghitungan belakang amplop dari perkiraan kompleksitas waktu penelusuran sinar pada model "bagus" (non-permusuhan). Bagi analisis Anda ke dalam kasus prapemrosesan dan komputasi gambar, sehingga Anda dapat memprediksi perilaku ray tracing beberapa frame untuk model statis.

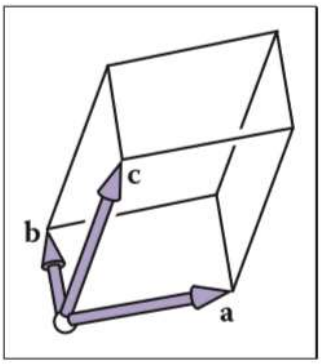
BAB 5
ALJABAR LINIER

Mungkin alat program grafis yang paling universal adalah matriks yang mengubah atau mentransformasikan titik dan vektor. Pada bab berikutnya, kita akan melihat bagaimana sebuah vektor dapat direpresentasikan sebagai matriks dengan satu kolom, dan bagaimana vektor dapat direpresentasikan dalam basis yang berbeda melalui perkalian dengan matriks persegi. Kami juga akan menjelaskan bagaimana kami dapat menggunakan perkalian tersebut untuk mencapai perubahan dalam vektor seperti scalling, rotasi, dan translasi.

Bab ini dapat dilewati oleh pembaca yang nyaman dengan araljabar garis. Namun, mungkin ada beberapa informasi penting yang mencerahkan bahkan untuk pembaca seperti itu, seperti pengembangan determinan dan pembahasan dekomposisi singular dan eigenvalue.



Gambar 5.1 Luas jajar genjang yang bertanda adalah |ab|, dan dalam hal ini luasnya positif.



Gambar 5.2 Volume bertanda dari paralelepiped yang ditunjukkan dilambangkan dengan determinan |abc|, dan dalam hal ini volumenya positif karena vektor-vektornya membentuk basis tangan kanan.

5.1 DETERMINAN

Kami biasanya menganggap determinan sebagai yang muncul dalam solusi persamaan linier. Namun, untuk tujuan kita, kita akan memikirkan determinan sebagai cara lain untuk mengalikan vektor. Untuk vektor 2D a dan b, determinannya |ab| adalah luas jajar genjang yang dibentuk oleh a dan b (Gambar 5.1). Ini adalah daerah bertanda, dan tandanya positif jika a dan b bertangan kanan dan negatif jika bertangan kiri. Ini berarti|ab| = |ba|. Dalam 2D kita dapat menafsirkan "tangan kanan" sebagai arti kita memutar vektor pertama berlawanan arah jarum jam untuk menutup sudut terkecil ke vektor kedua. Dalam 3D, determinan harus diambil dengan tiga vektor sekaligus. Untuk tiga vektor 3D, a, b, danc, determinannya|abc| adalah volume bertanda dari paralelepiped (jajaran genjang 3D; kotak 3D yang dicukur) yang dibentuk oleh tiga vektor (Gambar 5.2). Untuk menghitung determinan 2D, pertama-tama kita

perlu menetapkan beberapa propertinya. Kami mencatat bahwa scalling satu sisi jajar genjang menskalakan luasnya dengan fraksi yang sama (Gambar 5.3):

$$|(ka)b| = |a(kb)| = k|ab|.$$

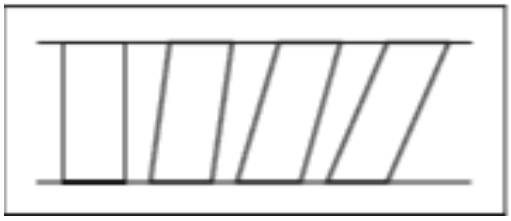
Juga, kami mencatat bahwa "memotong" jajar genjang tidak mengubah luasnya (Gambar 5.4):

$$|(a + kb)b| = |a(b + ka)| = |ab|.$$

Akhirnya, kita melihat bahwa determinan memiliki properti berikut:

$$|a(b + c)| = |ab|+|ac|,$$

karena seperti yang ditunjukkan pada Gambar 5.5 kita dapat “menggeser” tepi di antara dua jajar genjang untuk membentuk jajar genjang tunggal tanpa mengubah luas salah satu dari dua jajar genjang aslinya.

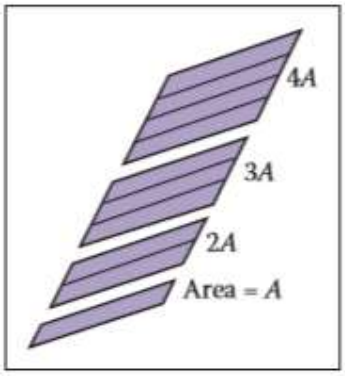


Gambar 5.4 Memotong jajar genjang tidak mengubah luasnya. Keempat jajar genjang ini memiliki alas yang sama panjang dan dengan demikian luas yang sama.

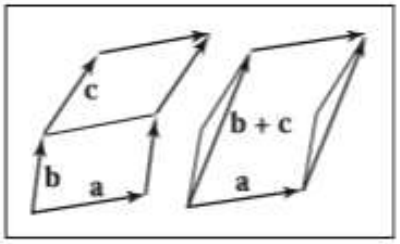
Sekarang mari kita asumsikan representasi Cartesian untuk a dan b:

$$\begin{aligned} |ab| &= |(x_a x + y_a y)\,(x_b x + y_b y)| \\ &= x_a x_b|xx| + x_a y_b|xy| + y_a x_b|yx| + y_a y_b|yy| \\ &= x_a x b(0) + x_a y_b(+1) + y_a x_b(-1) + y_a y_b(0) \\ &= x_a y_b - y_a x_b. \end{aligned}$$

Penyederhanaan ini menggunakan fakta bahwa $|vv| = 0$ untuk sembarang vektor v, karena jajar genjang semuanya akan kolinear dengan v dan dengan demikian tanpa luas.



Gambar 5.3 Menskalakan jajaran genjang di sepanjang satu arah mengubah area dalam proporsi yang sama



Gambar 5.5 Geometri dibalik Persamaan 5.1. Kedua jajar genjang di sebelah kiri dapat dicukur untuk menutupi jajar genjang tunggal di sebelah kanan.

Dalam tiga dimensi, determinan dari tiga vektor 3D a, b, dan c dinotasikan |abc|. Dengan representasi Cartesian untuk vektor, ada aturan analog untuk paralelepiped seperti halnya untuk jajaran genjang, dan kita dapat melakukan ekspansi analog seperti yang kita lakukan untuk 2D:

$$|abc| = |(x_a x + y_a y + z_a z)\,(x_b x + y_b y + z_b z)\,(x_c x + y_c y + z_c z)|$$
$$= x_a y_b z_c - x_a z_b y_c - y_a x_b z_c + y_a z_b x_c + z_a x_b y_c - z_a y_b x_c.$$

Seperti yang Anda lihat, penghitungan determinan dengan cara ini menjadi lebih buruk seiring dengan bertambahnya dimensi. Kita akan membahas cara-cara yang tidak terlalu rawan kesalahan untuk menghitung determinan di Bagian 5.3.

Contoh. Determinan muncul secara alami ketika menghitung ekspresi untuk satu vektor sebagai kombinasi linier dari dua vektor lainnya—misalnya, jika kita ingin menyatakan vektor c sebagai kombinasi vektor a dan b:

$$c = a_c a + b_c b.$$

Kita dapat melihat dari Gambar 5.6 bahwa

$$|(b_c b)a| = |ca|,$$

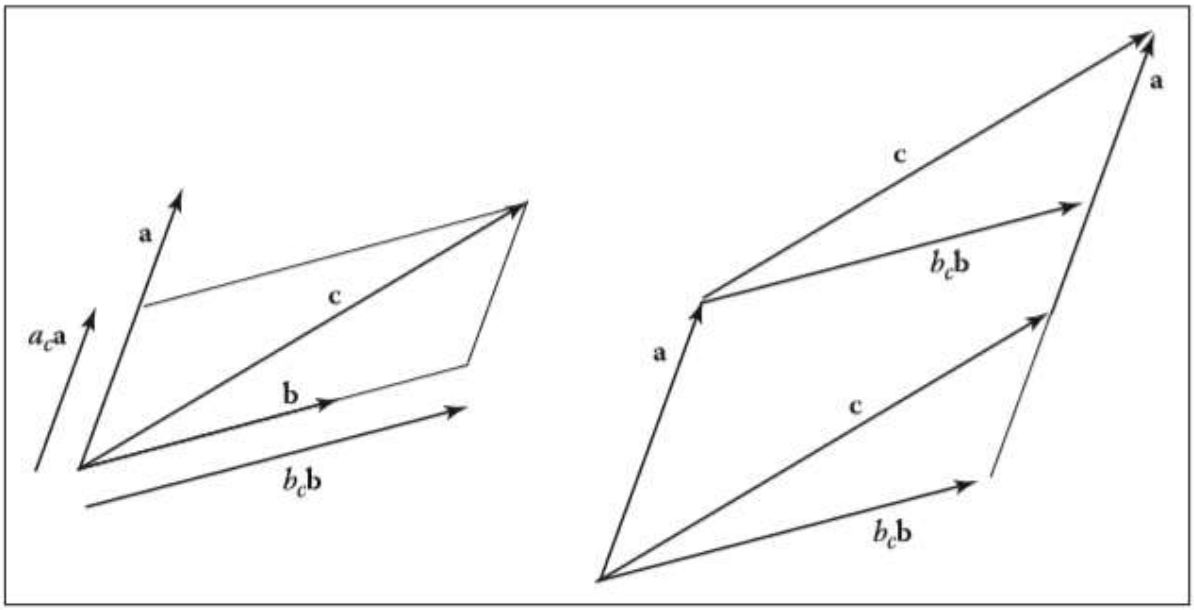
karena jajaran genjang ini hanyalah versi yang dicukur satu sama lain. Memecahkan hasil bc

$$b_c = \frac{|ca|}{|ba|}$$

Argumen analog menghasilkan

$$a_c = \frac{|bc|}{|ba|}$$

Ini adalah versi dua dimensi dari aturan Cramer.



Gambar 5.6 Di sebelah kiri, vektor c dapat direpresentasikan menggunakan dua vektor basis sebagai aca + bcb. Di sebelah kanan, kita melihat bahwa jajar genjang yang dibentuk oleh a dan c adalah versi geser dari jajar genjang yang dibentuk oleh bcb dan a.

5.2 MATRIKS

Matrik adalah susunan elemen numerik yang mengikuti aturan aritmatika tertentu. Contoh matriks dengan dua baris dan tiga kolom adalah

$$\begin{bmatrix} 1.7 & -1.2 & 4.2 \\ 3.0 & 4.5 & -7.2 \end{bmatrix}$$

Matriks sering digunakan dalam grafis komputer untuk berbagai tujuan termasuk representasi transformasi spasial. Untuk pembahasan kita, kita asumsikan elemen-elemen suatu matriks adalah semua bilangan real. Bab ini menjelaskan mekanika aritmatika matriks dan determinan matriks "persegi", yaitu matriks dengan jumlah baris yang sama dengan kolom.

Aritmatika Matriks

Matriks kali konstanta menghasilkan matriks di mana setiap elemen telah dikalikan dengan konstanta itu, mis.,

$$2\begin{bmatrix}1 & -4\\ 3 & 2\end{bmatrix} = \begin{bmatrix}2 & -8\\ 6 & 4\end{bmatrix}$$

Matriks juga menambahkan elemen demi elemen, mis.,

$$\begin{bmatrix}1 & -4\\ 3 & 2\end{bmatrix} + \begin{bmatrix}2 & 2\\ 2 & 2\end{bmatrix} = \begin{bmatrix}3 & -2\\ 5 & 4\end{bmatrix}$$

Untuk perkalian matriks, kita "mengkalikan" baris-baris matriks pertama dengan kolom-kolom matriks kedua:

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{i1} & \dots & a_{im} \\ \vdots & & \vdots \\ a_{r1} & \dots & a_{rm} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1c} \\ \vdots & & \vdots & & \vdots \\ b_{m1} & \dots & b_{mj} & \dots & b_{mc} \end{bmatrix} = \begin{bmatrix} p_{11} & \dots & p_{1j} & \dots & p_{1c} \\ \vdots & & \vdots & & \vdots \\ p_{i1} & \dots & p_{ij} & \dots & p_{ic} \\ \vdots & & \vdots & & \vdots \\ p_{r1} & \dots & p_{rj} & \dots & p_{rc} \end{bmatrix}.$$

Jadi elemen p_{ij} dari produk yang dihasilkan adalah

$$p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj}$$

Pengambilan hasil kali dua matriks hanya mungkin jika jumlah kolom matriks kiri sama dengan jumlah baris matriks kanan. Sebagai contoh,

$$\begin{bmatrix}0 & 1\\ 2 & 3\\ 4 & 5\end{bmatrix}\begin{bmatrix}6 & 7 & 8 & 9\\ 0 & 1 & 2 & 3\end{bmatrix} \begin{matrix}0 & 1 & 2 & 3\\ 12 & 17 & 22 & 27\\ 24 & 33 & 42 & 51\end{matrix}$$

Perkalian matriks tidak komutatif dalam kebanyakan kasus:

$$AB \neq BA$$

Juga, jika AB = AC, tidak selalu mengikuti bahwa B = C. Untungnya, perkalian matriks adalah asosiatif dan distributif:

$$(AB)C = A(BC),$$
$$A(B + C) = AB + AC,$$
$$(A + B)C = AC + BC.$$

Operasi Matriks

Kami ingin analog matriks dari kebalikan dari bilangan real. Kita tahu invers bilangan real x adalah 1/x dan hasil kali x dan inversnya adalah 1. Kita membutuhkan matriks I yang dapat kita anggap sebagai "matriks satu". Ini hanya ada untuk matriks persegi dan dikenal sebagai matriks identitas; itu terdiri dari satu di diagonal dan nol di tempat lain. Misalnya, matriks identitas empat kali empat adalah

$$\begin{matrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{matrix}$$

Matriks invers A^{-1} dari matriks A adalah matriks yang memastikan $AA^{-1} = I$. Misalnya,

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}^{-1} = \begin{bmatrix}-2.0 & 1.0\\ 1.5 & -0.5\end{bmatrix} \; karena \; \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}-2.0 & 1.0\\ 1.5 & -0.5\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$

Perhatikan bahwa invers dari A^{-1} adalah A. Maka AA−1 = A−1A = I. Invers dari perkalian dua matriks adalah perkalian dari invers, tetapi dengan urutan yang dibalik:

$$(AB)^{-1} = B^{-1}A^{-1}.$$

Kami akan kembali ke pertanyaan tentang menghitung kebalikannya nanti dalam bab ini. Transpos A^T dari matriks A memiliki bilangan yang sama tetapi baris-barisnya ditukar dengan kolom-kolomnya. Jika kita memberi label entri A^T sebagai a'_{ij} maka

$$a_{ij} = a'_{ij}$$

Sebagai contoh,

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}^T = \begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}$$

Transpos produk dua matriks mengikuti aturan yang mirip dengan Persamaan (5.4):

$$(AB)^T = B^T A^T.$$

Determinan matriks bujur sangkar hanyalah determinan kolom matriks, yang dianggap sebagai himpunan vektor. Determinan memiliki beberapa hubungan yang baik dengan operasi matriks yang baru saja dibahas, yang kami cantumkan di sini untuk referensi:

$$|AB| = |A||B|,$$

$$|A^{-1}| = \frac{1}{|A|}$$

$$|A^T| = |A|$$

Operasi Vektor dalam Bentuk Matriks

Dalam grafis, kami menggunakan matriks persegi untuk mengubah vektor yang direpresentasikan sebagai matriks. Misalnya, jika Anda memiliki vektor 2D a =(x_a, y_a) dan ingin memutarnya 90 derajat terhadap titik asal untuk membentuk vektor a' = (−y_a, x_a), Anda dapat menggunakan perkalian matriks 2 × 2 dan a 2 × 1 matriks, disebut vektor kolom. Operasi dalam bentuk matriks adalah

$$\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}\begin{bmatrix}x_a\\ y_a\end{bmatrix} = \begin{bmatrix}-y_a\\ x_a\end{bmatrix}$$

Kita bisa mendapatkan hasil yang sama dengan menggunakan transpos matriks ini dan mengalikan di sebelah kiri ("premultiplying") dengan vektor baris:

$$\begin{bmatrix}x_a & y_a\end{bmatrix}\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} = \begin{bmatrix}-y_a & x_a\end{bmatrix}$$

Saat ini, pascaperkalian menggunakan vektor kolom cukup standar, tetapi di banyak buku dan sistem lama Anda akan menggunakan vektor baris dan praperkalian. Satu-satunya perbedaan adalah bahwa matriks transformasi harus diganti dengan transposnya.

Kita juga dapat menggunakan formalisme matriks untuk mengkodekan operasi pada vektor saja. Jika kita menganggap hasil perkalian titik sebagai matriks 1 × 1, dapat ditulis:

$$a \cdot b = a^T b.$$

Misalnya, jika kita mengambil dua vektor 3D, kita mendapatkan

$$\begin{bmatrix}x_a & y_a & z_a\end{bmatrix}\begin{bmatrix}x_b\\ y_b\\ z_b\end{bmatrix} = \begin{bmatrix}x_a x_b + y_a y_b + z_a z_b\end{bmatrix}$$

Sebuah produk vektor terkait adalah produk luar antara dua vektor, yang dapat dinyatakan sebagai perkalian matriks dengan vektor kolom di sebelah kiri dan vektor baris di sebelah kanan: ab^T. Hasilnya adalah matriks yang terdiri dari produk dari semua pasangan entri a dengan entri b. Untuk vektor 3D, kita memiliki

$$\begin{bmatrix}x_a\\ y_a\\ z_a\end{bmatrix}\begin{bmatrix}x_b & y_b & z_b\end{bmatrix} = \begin{bmatrix}x_a x_b & x_a y_b & x_a z_b\\ y_a x_b & y_a y_b & y_a z_b\\ z_a x_b & z_a y_b & z_a z_b\end{bmatrix}$$

Seringkali berguna untuk memikirkan perkalian matriks dalam bentuk operasi vektor. Untuk mengilustrasikan penggunaan kasus tiga dimensi, kita dapat menganggap matriks 3 × 3 sebagai kumpulan tiga vektor 3D dalam dua cara: terdiri dari tiga vektor kolom berdampingan, atau terdiri dari tiga vektor kolom. vektor baris ditumpuk. Misalnya, hasil perkalian matriks-vektor y = Ax dapat diinterpretasikan sebagai vektor yang entri-entrinya adalah hasil kali titik dari x dengan baris-baris A. Penamaan vektor-vektor baris ini r_i, kita miliki

$$\begin{bmatrix} | \\ y \\ | \end{bmatrix} = \begin{bmatrix} - & r_1 & - \\ - & r_2 & - \\ - & r_3 & - \end{bmatrix} \begin{bmatrix} | \\ x \\ | \end{bmatrix}$$

$$y_i = r_i \cdot x$$

Sebagai alternatif, kita dapat menganggap hasil kali yang sama sebagai jumlah dari tiga kolom c_i dari A, yang diberi bobot oleh entri x:

$$\begin{bmatrix} | \\ y \\ | \end{bmatrix} = \begin{bmatrix} | & | & | \\ c_1 & c_2 & c_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$y = x_1c_1 + x_2c_2 + x_3c_3.$$

Dengan menggunakan ide yang sama, seseorang dapat memahami hasil kali matriks-matriks AB sebagai larik yang berisi produk titik berpasangan yang merupakan baris jatuh dari A dengan semua kolom B (lih. (5.2)); sebagai kumpulan hasil kali matriks A dengan semua vektor kolom B, disusun dari kiri ke kanan; sebagai kumpulan produk dari semua vektor baris A dengan matriks B, ditumpuk dari atas ke bawah; atau sebagai jumlah dari perkalian luar berpasangan dari semua kolom A dengan semua baris B. (Lihat Latihan 8.) Interpretasi perkalian matriks ini sering dapat menyebabkan interpretasi geometris yang berharga dari operasi yang mungkin tampak sangat abstrak.

Jenis Matriks Khusus

Matriks identitas adalah contoh matriks diagonal, di mana semua elemen bukan nol terjadi di sepanjang diagonal. Diagonal terdiri dari elemen-elemen yang indeks kolomnya sama dengan indeks baris yang dihitung dari kiri atas.

Matriks identitas juga memiliki sifat yang sama dengan transposnya. Matriks seperti ini disebut simetris. dianggap sebagai vektor memiliki panjang 1 dan kolom adalah ortogonal satu sama lain. Hal yang sama berlaku untuk baris (lihat Latihan 2). Determinan dari sembarang matriks ortogonal adalah +1 atau −1. Sifat matriks ortogonal yang sangat berguna adalah bahwa matriks tersebut hampir merupakan invers irown. Mengalikan matriks ortogonal dengan transposnya menghasilkan identitas,

$$R^TR = I = RR^T \text{ untuk ortogonal } R$$

Hal ini mudah dilihat karena entri RTR adalah produk titik antara kolom R. Entri off-diagonal adalah produk titik antara vektor ortogonal, dan entri diagonal adalah produk titik dari kolom (satuan-panjang) dengan dirinya sendiri. Gagasan matriks ortogonal sesuai dengan gagasan basis ortonormal, bukan hanya sekumpulan vektor ortogonal—kesalahan terminologi yang tidak menguntungkan.

Contoh

Matrik

$$\begin{bmatrix} 8 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

diagonal, dan karena itu simetris, tetapi tidak ortogonal (kolom-kolomnya ortogonal tetapi bukan panjang satuan).

Matrik

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 9 & 7 \\ 2 & 7 & 1 \end{bmatrix}$$

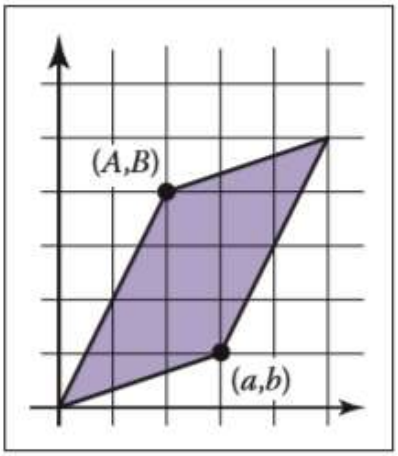
simetris, tetapi diagonalnya tidak ortogonal.
Matriks

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

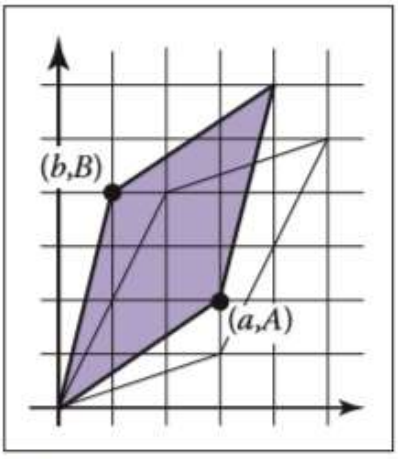
adalah ortogonal, tetapi tidak diagonal atau simetris.

5.3 KOMPUTASI DENGAN MATRIK DAN DETERMINASI

Ingat dari Bagian 5.1 bahwa determinan mengambil vektor-vektor berdimensi-n dan menggabungkannya untuk mendapatkan volume berdimensi-n bertanda dari paralelepiped berdimensi-n yang didefinisikan oleh vektor-vektor tersebut. Misalnya, determinan dalam 2Dislah luas jajaran genjang yang dibentuk oleh vektor. Kita dapat menggunakan matriks untuk menangani mekanika determinan komputasi.



Gambar 5.7 Determinan 2D dalam Persamaan 5.8 adalah luas jajaran genjang yang dibentuk oleh vektor 2D.



Gambar 5.8 Jajar genjang bersaudara memiliki luas yang sama dengan jajar genjang pada Gambar 5.7.

Jika kita memiliki vektor 2D r dan s, kita menyatakan determinannya|rs|; nilai ini adalah area bertanda jajaran genjang yang dibentuk oleh vektor. Misalkan kita memiliki dua

vektor 2D dengan koordinat Cartesian (a,b) dan (A,B) (Gambar 5.7). Determinan dapat ditulis dalam bentuk vektor kolom atau sebagai singkatan:

$$\left|\begin{bmatrix}a\\b\end{bmatrix}\begin{bmatrix}A\\B\end{bmatrix}\right| \equiv \begin{vmatrix}a & A\\b & B\end{vmatrix} = aB - Ab$$

Perhatikan bahwa determinan suatu matriks sama dengan determinan transposnya:

$$\begin{vmatrix}a & A\\b & B\end{vmatrix} = \begin{vmatrix}a & A\\b & B\end{vmatrix} = aB - Ab$$

Ini berarti bahwa untuk setiap jajar genjang dalam 2D ada jajar genjang "saudara" yang memiliki luas yang sama tetapi bentuk yang berbeda (Gambar 5.8). Misalnya, jajar genjang yang didefinisikan oleh vektor (3,1) dan (2,4) memiliki luas 10, seperti halnya jajar genjang yang didefinisikan oleh vektor (3,2) dan (1,4)

Contoh

Arti geometris dari penentu 3D sangat membantu dalam melihat mengapa rumus tertentu masuk akal. Misalnya, persamaan bidang yang melalui titik (xi,y i,z i) untuk i =0,1,2 adalah

$$\begin{vmatrix}x - x_0 & x - x_1 & x - x_2\\y - y_0 & y - y_1 & y - y_2\\z - z_0 & z - z_1 & z - z_2\end{vmatrix} = 0$$

Setiap kolom adalah vektor dari titik (x_i, y_i, z_i) ke titik (x, y, z). Volume paralelepiped dengan vektor-vektor sebagai sisi adalah nol hanya jika (x, y, z) adalah sejajar dengan tiga titik lainnya. Hampir semua persamaan yang melibatkan determinan memiliki geometri dasar yang sama sederhananya.

Seperti yang kita lihat sebelumnya, kita dapat menghitung determinan dengan ekspansi brute force di mana sebagian besar suku adalah nol, dan ada banyak pembukuan pada tanda plus dan minus. Cara standar untuk mengelola aljabar determinan komputasi adalah dengan menggunakan bentuk ekspansi Laplace. Bagian penting dari menghitung determinan dengan cara ini adalah menemukan kofaktor dari berbagai elemen matriks. Setiap elemen matriks bujur sangkar memiliki kofaktor yang merupakan determinan matriks dengan satu baris dan kolom lebih sedikit yang mungkin dikalikan dengan minus satu. Matriks yang lebih kecil diperoleh dengan menghilangkan baris dan kolom tempat elemen tersebut berada. Misalnya, untuk matriks 10 × 10, kofaktor dari 82 adalah determinan matriks 9 × 9 dengan baris ke-8 dan kolom ke-2 dihilangkan. Tanda suatu kofaktor adalah positif jika jumlah indeks baris dan kolomnya genap dan sebaliknya negatif. Ini dapat diingat dengan pola kotak-kotak:

$$\begin{bmatrix}+ & - & + & - & \cdots\\- & + & - & + & \cdots\\+ & - & + & - & \cdots\\- & + & - & + & \cdots\\\vdots & \vdots & \vdots & \vdots & \ddots\end{bmatrix}$$

Jadi, untuk matriks 4x4,

$$\mathbf{A} = \begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\a_{21} & a_{22} & a_{23} & a_{24}\\a_{31} & a_{32} & a_{33} & a_{34}\\a_{41} & a_{42} & a_{43} & a_{44}\end{bmatrix}.$$

Kofaktor baris pertama adalah

$$a_{11}^c = \begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix}, \quad a_{12}^c = - \begin{vmatrix} a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} \end{vmatrix},$$

$$a_{13}^c = \begin{vmatrix} a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix}, \quad a_{14}^c = - \begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix}.$$

Determinan suatu matriks diperoleh dengan menjumlahkan perkalian elemen-elemen baris atau kolom dengan kofaktornya. Sebagai contoh, determinan matriks 4 × 4 di atas diambil pada kolom kedua adalah

$$|A| = a_{12}a^{c12} + a_{22}a^{c}{}_{22} + a_{32}a^{c}{}_{32} + a_{42}a^{c}{}_{42}.$$

Kita bisa melakukan ekspansi serupa tentang setiap baris atau kolom dan semuanya akan menghasilkan hasil yang sama. Perhatikan sifat rekursif dari ekspansi ini.

Contoh

Contoh konkrit untuk determinan matriks 3 × 3 tertentu dengan memperluas kofaktor baris pertama adalah

$$\begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{vmatrix} = 0 \begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix} - 1 \begin{vmatrix} 3 & 5 \\ 6 & 8 \end{vmatrix} + 2 \begin{vmatrix} 3 & 4 \\ 6 & 7 \end{vmatrix}$$
$$= 0(32 - 35) - 1(24 - 30) + 2(21 - 24)$$
$$= 0$$

Kita dapat menyimpulkan bahwa volume paralelepiped yang dibentuk oleh vektor-vektor yang didefinisikan oleh kolom (atau baris karena determinan transposnya sama) adalah nol. Ini sama dengan mengatakan bahwa kolom (atau baris) tidak bebas linier. Perhatikan bahwa jumlah baris pertama dan ketiga adalah dua kali baris kedua, yang menyiratkan ketergantungan linier.

Komputasi Invers

Determinan memberi kita alat untuk menghitung invers suatu matriks. Ini adalah metode yang sangat tidak efisien untuk matriks besar, tetapi seringkali dalam grafis matriks kita kecil. Kunci untuk mengembangkan metode ini adalah bahwa determinan matriks dengan dua baris identik adalah nol. Ini harus jelas karena volume n-dimensi paralel epipedis nol jika dua sisinya sama. Misalkan kita memiliki 4x4 A dan kita ingin mencari inversnya A−1. Kebalikannya adalah:

Perhatikan bahwa ini hanyalah transpos matriks di mana elemen A diganti dengan kofaktornya masing-masing dikalikan dengan konstanta terdepan (1 atau -1). Matriks ini disebut adjoint dari A. Adjoint adalah transpos dari matriks kofaktor dari A. Kita dapat melihat mengapa ini adalah invers. Lihatlah produk AA^{-1} yang kita harapkan menjadi identitasnya. Jika kita mengalikan baris pertama dari A dengan kolom pertama dari matriks adjoint, kita perlu mendapatkan|A|(ingat konstanta terdepan di atas dibagi dengan |A|:

$$A^{-1} = \frac{1}{|A|} \begin{bmatrix} a_{11}^c & a_{21}^c & a_{31}^c & a_{41}^c \\ a_{12}^c & a_{22}^c & a_{32}^c & a_{42}^c \\ a_{13}^c & a_{23}^c & a_{33}^c & a_{43}^c \\ a_{14}^c & a_{24}^c & a_{34}^c & a_{44}^c \end{bmatrix}$$

Hal ini benar karena elemen-elemen pada baris pertama A dikalikan secara tepat dengan kofaktornya pada kolom pertama matriks adjoint yang merupakan determinan. Nilai lain sepanjang diagonal matriks yang dihasilkan adalah |A| untuk alasan analog. Angka nol mengikuti logika yang sama:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} a_{11}^c & \cdot & \cdot & \cdot \\ a_{12}^c & \cdot & \cdot & \cdot \\ a_{13}^c & \cdot & \cdot & \cdot \\ a_{14}^c & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} |\mathbf{A}| & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Perhatikan bahwa produk ini merupakan determinan dari beberapa matriks:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} a_{11}^c & \cdot & \cdot & \cdot \\ a_{12}^c & \cdot & \cdot & \cdot \\ a_{13}^c & \cdot & \cdot & \cdot \\ a_{14}^c & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Matriks sebenarnya adalah

$$a_{21}a_{11}^c + a_{22}a_{12}^c + a_{23}a_{13}^c + a_{24}a_{14}^c$$

Karena dua baris pertama identik, matriksnya singular, dan dengan demikian, determinannya adalah nol.

Argumen di atas tidak hanya berlaku untuk matriks empat kali empat; menggunakan ukuran itu hanya menyederhanakan tipografi. Untuk sembarang matriks, inversnya adalah matriks adjoint dibagi dengan determinan matriks yang dibalik. Adjoint adalah transpos matriks kofaktor, yang hanya merupakan matriks yang elemen-elemennya telah digantikan oleh kofaktornya.

Contoh

Invers dari satu matriks tiga kali tiga yang determinannya adalah 6 adalah

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 4 \\ 0 & 2 & 5 \end{bmatrix}^{-1} = \frac{1}{6} \begin{bmatrix} \begin{vmatrix} 3 & 4 \\ 2 & 5 \end{vmatrix} & -\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix} & \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \\ -\begin{vmatrix} 1 & 4 \\ 0 & 5 \end{vmatrix} & \begin{vmatrix} 1 & 2 \\ 0 & 5 \end{vmatrix} & -\begin{vmatrix} 1 & 2 \\ 1 & 4 \end{vmatrix} \\ \begin{vmatrix} 1 & 3 \\ 0 & 2 \end{vmatrix} & -\begin{vmatrix} 1 & 1 \\ 0 & 2 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 3 \end{vmatrix} \end{bmatrix}$$

$$= \frac{1}{6} \begin{bmatrix} 7 & -1 & -2 \\ -5 & 5 & -2 \\ 2 & -2 & 2 \end{bmatrix}.$$

Anda dapat memeriksanya sendiri dengan mengalikan matriks dan memastikan Anda mendapatkan identitasnya.

Sistem Linier

Kita sering kali melawan sistem linier dalam grafis dengan “n persamaan dan n tidak diketahui”, biasanya untuk n = 2 atau n = 3. Sebagai contoh,

$$3x + 7y + 2z = 4$$
$$2x - 4y - 3z = -1$$
$$5x + 2y + z = 1$$

Di sini x, y, da z adalah "tidak diketahui" yang ingin kita pecahkan. Kita dapat menulis ini dalam bentuk matriks:

$$\begin{bmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ -1 \\ 1 \end{bmatrix}$$

Singkatan umum untuk sistem tersebut adalah Ax = b di mana diasumsikan bahwa A adalah matriks persegi dengan konstanta yang diketahui, x adalah vektor kolom yang tidak

diketahui (dengan elemen x, y, andz dalam contoh kita), dan b adalah matriks kolom dari konstanta yang diketahui.

Ada banyak cara untuk menyelesaikan sistem seperti itu, dan metode yang sesuai bergantung pada sifat dan dimensi matriks A. Karena dalam grafis kita begitu sering bekerja dengan sistem berukuran n ≤ 4, kita akan membahas di sini metode yang sesuai untuk sistem ini, yang dikenal sebagai aturan Cramer, yang kita lihat sebelumnya, dari sudut pandang geometris 2D, pada contoh di halaman 90. Di sini, kami menunjukkannya secara aljabar. Solusi persamaan di atas adalah

$$x = \frac{\begin{vmatrix} 4 & 7 & 2 \\ -1 & -4 & -3 \\ 1 & 2 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}}; \quad y = \frac{\begin{vmatrix} 3 & 4 & 2 \\ 2 & -1 & -3 \\ 5 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}}; \quad z = \frac{\begin{vmatrix} 3 & 7 & 4 \\ 2 & -4 & -1 \\ 5 & 2 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 7 & 2 \\ 2 & -4 & -3 \\ 5 & 2 & 1 \end{vmatrix}}.$$

Aturan di sini adalah mengambil rasio determinan, di mana penyebut|A|dan pembilangnya adalah determinan dari matriks yang dibuat dengan mengganti kolom A dengan vektor kolom b. Kolom yang diganti sesuai dengan posisi yang tidak diketahui dalam vektor x. Misalnya, y adalah yang kedua tidak diketahui dan kolom kedua diganti. Perhatikan bahwa jika |A| =0, pembagian tidak terdefinisi dan tidak ada solusi. Ini hanyalah versi lain dari aturan bahwa jika A singular (determinan nol) maka tidak ada solusi unik untuk persamaan tersebut.

5.4 EIGENVALUE DAN DIAGONALISASI MATRIKS

Matriks persegi memiliki nilai eigen dan vektor eigen yang terkait dengannya. Vektor eigen adalah vektor bukan nol yang arahnya tidak berubah ketika dikalikan dengan matriks. Sebagai contoh, misalkan untuk matriks A dan vektor a, kita memiliki

$$Aa = \lambda a.$$

Ini berarti kita telah meregangkan atau menekan a, tetapi arahnya tidak berubah. Faktor skala λ disebut nilai eigen yang dikaitkan dengan vektor eigen a. Mengetahui nilai eigen dan vektor eigen dari matriks sangat membantu dalam berbagai aplikasi praktis. Kami akan menjelaskannya untuk mendapatkan wawasan tentang matriks transformasi geometris dan sebagai langkah menuju nilai dan vektor tunggal yang dijelaskan di bagian berikutnya.

Jika kita mengasumsikan sebuah matriks memiliki setidaknya satu vektor eigen, maka kita dapat melakukan manipulasi standar untuk menemukannya. Pertama, kami menulis kedua sisi sebagai produk dari matriks persegi dengan vektor a:

$$Aa = \lambda Ia,$$

di mana I adalah matriks identitas. Ini bisa ditulis ulang

$$Aa - \lambda Ia = 0 .$$

Karena perkalian matriks adalah distributif, kita dapat mengelompokkan matriks-matriks tersebut:

$$(A - \lambda I)a = 0 .$$

Persamaan ini hanya bisa benar jika matriks (A – λI) adalah singular, dan dengan demikian determinannya adalah nol. Unsur-unsur dalam matriks ini adalah bilangan-bilangan di A kecuali sepanjang diagonalnya. Misalnya, untuk matriks 2 × 2 nilai eigennya mematuhi

$$\begin{vmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{vmatrix} = \lambda^2 - (a_{11} + a_{22})\lambda + (a_{11}a_{22} - a_{12}a_{21}) = 0$$

Karena ini adalah persamaan kuadrat, kita tahu persis ada dua solusi untuk λ. Solusi ini mungkin atau mungkin tidak unik atau nyata. Manipulasi serupa untuk matriks n x n akan menghasilkan polinomial derajat-n di λ. Karena tidak mungkin, secara umum, untuk menemukan solusi eksplisit eksak dari persamaan polinomial dengan derajat lebih besar dari empat, kita hanya dapat menghitung nilai eigen matriks 4x4 atau lebih kecil dengan metode analitik. Untuk matriks yang lebih besar, metode numerik adalah satu-satunya pilihan.

Kasus khusus yang penting di mana nilai eigen dan vektor eigen sangat sederhana adalah matriks simetris (di mana $A = A^T$). Nilai eigen matriks simetris real selalu bilangan real, dan jika juga berbeda, vektor eigennya saling ortogonal. Matriks tersebut dapat dimasukkan ke dalam bentuk diagonal:

$$A = QDQ^T$$

di mana Q adalah matriks ortogonal dan D adalah matriks diagonal. Kolom Q adalah vektor eigen dari A dan elemen diagonal dari D adalah nilai eigen dari A. Penempatan A dalam bentuk ini juga disebut dekomposisi nilai eigen, karena ia menguraikan A menjadi produk matriks sederhana yang mengungkapkan vektor eigen dan nilai eigennya. Ingatlah bahwa matriks ortogonal memiliki baris ortonormal dan kolom ortonormal.

Contoh

Diberikan matriks

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

nilai eigen dari A adalah solusi dari

$$\lambda^2 - 3\lambda + 1 = 0$$

Kami memperkirakan nilai yang tepat untuk kekompakan notasi:

$$\lambda = \frac{3 \pm \sqrt{5}}{2}, \approx \begin{bmatrix} 2.618 \\ 0.382 \end{bmatrix}$$

Sekarang kita dapat menemukan vektor eigen terkait. Yang pertama adalah solusi nontrivial (bukan x = y = 0) untuk persamaan homogen,

$$\begin{bmatrix} 2 - 2.618 & 1 \\ 1 & 1 - 2.618 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Ini kira-kira (x,y) = (0 .8507.0.5257). Perhatikan bahwa ada tak hingga banyak solusi yang paralel dengan vektor 2D tersebut, dan kita baru saja memilih salah satu yang memiliki panjang satuan. Demikian pula vektor eigen yang terkait dengan 2 adalah (x,y)=(−0.5257.0.8507). Ini berarti bentuk diagonal dari A adalah (dalam beberapa presisi karena perkiraan numerik kami):

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} 2.618 & 0 \\ 0 & 0.382 \end{bmatrix} \begin{bmatrix} 0.8507 & 0.5257 \\ -0.5257 & 0.8507 \end{bmatrix}$$

Kami akan meninjau kembali geometri matriks ini sebagai transformasi di bab berikutnya.

Dekomposisi Nilai Tunggal

Kita telah melihat di bagian terakhir bahwa setiap matriks simetris dapat didiagonalisasi, atau didekomposisi menjadi produk matriks ortogonal dan diagonal yang sesuai. Namun, sebagian besar matriks yang kita temui dalam grafis tidak simetris, dan dekomposisi nilai eigen untuk matriks nonsimetris hampir tidak begitu nyaman atau mencerahkan, dan secara umum melibatkan nilai eigen dan vektor eigen bernilai kompleks bahkan untuk input bernilai nyata.

Kami akan merekomendasikan pembelajaran dalam urutan ini: nilai eigen/vektor simetris, nilai/vektor tunggal, dan kemudian nilai eigen nonsimetris, yang jauh lebih rumit. Kami akan merekomendasikan pembelajaran dalam urutan ini: nilai eigen/vektor simetris, nilai/vektor tunggal, dan kemudian nilai eigen nonsimetris, yang jauh lebih rumit.

Ada generalisasi lain dari dekomposisi nilai eigen simetris ke matriks nonsimetris (dan bahkan non-persegi); itu adalah dekomposisi nilai tunggal/*singular value decomposition* (SVD). Perbedaan utama antara dekomposisi nilai eigen dari matriks simetris dan SVD dari matriks nonsimetris adalah bahwa matriks ortogonal di sisi kiri dan kanan tidak harus sama dalam SVD:

$$A = USV^T$$

Di sini U dan V adalah dua matriks ortogonal yang berpotensi berbeda, yang kolomnya dikenal sebagai vektor singular kiri dan kanan dari A, dan S adalah matriks diagonal yang entrinya dikenal sebagai nilai singular A. Bila A simetris dan semua nonnegatif nilai eigen, SVD dan dekomposisi nilai eigen adalah sama.

Ada hubungan lain antara nilai singular dan nilai eigen yang dapat digunakan untuk menghitung SVD (meskipun ini bukan cara kerja implementasi SVD kekuatan industri). Pertama kita definisikan $M = AA^T$. Kami berasumsi bahwa kami dapat melakukan SVD pada M:

$$M = AA^T = (USV^T)\,(USV^T)^T = US(V^TV)SU^T = US^2U^T.$$

Substitusi didasarkan pada fakta bahwa $(BC)^T = C^TB^T$, bahwa transpos matriks ortogonal adalah kebalikannya, dan transpos matriks diagonal adalah matriks itu sendiri. Keindahan bentuk baru ini adalah M simetris dan US2UT adalah dekomposisi nilai eigennya, di mana S2 berisi nilai eigen (semua nonnegatif). Jadi, kita temukan bahwa nilai singular suatu matriks adalah akar kuadrat dari nilai eigen produk matriks dengan transposnya, dan vektor singular kiri adalah vektor eigen dari produk tersebut. Argumen serupa memungkinkan V, matriks vektor tunggal kanan, untuk dihitung dari A^TA.

Contoh

Kami sekarang membuat ini konkret dengan sebuah contoh:

$$A = \begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix};\ M = AA^T = \begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix}$$

Kami melihat dekomposisi nilai eigen untuk matriks ini di bagian sebelumnya. Kami segera mengamati

$$\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} = \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}\sqrt{2.618} & 0\\ 0 & \sqrt{0.382}\end{bmatrix}V^T$$

Kita dapat memecahkan V secara aljabar:

$$V = (S^{-1}U^TA)^T.$$

Invers dari S adalah matriks diagonal dengan kebalikan dari elemen-elemen diagonal S. Ini menghasilkan

$$\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} = U\begin{bmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{bmatrix}V^T$$

$$= \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}1.618 & 0\\ 0 & 0.618\end{bmatrix}\begin{bmatrix}0.5257 & 0.8507\\ -0.8507 & 0.5257\end{bmatrix}$$

Bentuk ini menggunakan simbol standar σ_i untuk nilai singular ke-i. Sekali lagi, untuk matriks simetris, nilai eigen dan nilai singularnya sama ($\sigma_i = \lambda_i$). Kami akan memeriksa geometri SVD lebih lanjut di Bagian 6.1.6.

Pertanyaan yang Sering Diajukan

- Mengapa perkalian matriks didefinisikan seperti itu daripada hanya elemen demi elemen?

Perkalian elemen demi elemen adalah cara yang sangat baik untuk mendefinisikan perkalian matriks, dan memang memiliki sifat yang bagus. Namun, dalam praktiknya itu tidak terlalu berguna. Pada akhirnya, sebagian besar matriks digunakan untuk mengubah vektor kolom, misalnya, dalam 3D yang mungkin Anda miliki

$$b = Ma,$$

di mana a dan b adalah vektor dan M adalah matriks 3x3. Untuk memungkinkan operasi geometri seperti rotasi, kombinasi ketiga elemen a harus masuk ke setiap elemen b. Itu mengharuskan kita untuk pergi baris demi baris atau kolom demi kolom melalui M. Pilihan itu dibuat berdasarkan komposisi matriks yang memiliki properti yang diinginkan,

$$M_2(M_1a) = (M_2M_1)a$$

yang memungkinkan kita untuk menggunakan satu matriks komposit $C = M_2M_1$ untuk mengubah vektor kita. Ini berharga ketika banyak vektor akan ditransformasikan oleh matriks komposit yang sama. Jadi, secara ringkas, aturan yang agak aneh untuk perkalian matriks direkayasa agar memiliki sifat-sifat yang diinginkan ini.

- Terkadang saya mendengar bahwa nilai eigen dan nilai singular adalah hal yang sama dan terkadang yang satu adalah kuadrat dari yang lain. Yang mana yang benar?

Jika matriks real A simetris, dan nilai eigennya nonnegatif, maka nilai eigen dan nilai singularnya sama. Jika A tidak simetris, matriks $M = AA^T$ simetris dan memiliki nilai eignen real nonnegatif. Nilai singular dari A dan A^T adalah sama dan merupakan akar kuadrat dari singular/eigenvalues M. Jadi, ketika pernyataan akar kuadrat dibuat, itu karena dua matriks yang berbeda (dengan hubungan yang sangat khusus) sedang dibicarakan : $M = AA^T$.

Latihan

1. Tulis persamaan implisit untuk garis 2D yang melalui titik (x_0, y_0) dan (x_1, y_1) menggunakan determinan 2D.
2. Tunjukkan bahwa jika kolom-kolom suatu matriks ortonormal, maka baris-barisnya juga ortonormal.
3. Buktikan sifat-sifat determinan matriks yang dinyatakan dalam Persamaan (5.5)–(5.7).
4. Tunjukkan bahwa nilai eigen suatu matriks diagonal adalah elemen-elemen diagonalnya.
5. Tunjukkan bahwa untuk matriks persegi A, AA^T adalah matriks simetris.
6. Tunjukkan bahwa untuk tiga vektor 3D,b,c,identitas berikut berlaku: $|abc| = (a \times b) \cdot c$.
7. Jelaskan mengapa volume tetrahedron dengan vektor sisi a, b, c (lihat Gambar 5.2) diberikan oleh$|abc|/6$.
8. Mendemonstrasikan keempat interpretasi perkalian matriks-matriks dengan mengambil kode perkalian matriks-matriks berikut, menyusun ulang nested loops, dan menginterpretasikan kode yang dihasilkan dalam bentuk operasi matriks dan vektor.

```
function mat-mult(in a[m][p], in b[p][n], out c[m][n]) {
        // the array c is initialized to zero
        for i = 1 to
                m for j = 1 to n
                        for k = 1 to p
                                c[i][j] += a[i][k] * b[k][j]
}
```

9. Buktikan bahwa jika A, Q, dan D memenuhi Persamaan (5.14), v adalah baris ke-i dari Q, dan λ adalah entri ke-i pada diagonal D, maka v adalah vektor eigen dari A dengan nilai eigen λ.
10. Buktikan A, Q, dan D memenuhi Persamaan (5.14), nilai eigen dari A semua berbeda, dan v adalah vektor eigen dari A dengan nilai eigen , maka untuk beberapa i, v adalah baris ke-i dari Q dan adalah entri ke-i pada diagonal D .
11. Diketahui koordinat (x, y) ketiga titik sudut pada segitiga 2D, jelaskan mengapa luas diberikan oleh

$$\frac{1}{2}\begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}$$

BAB 6
MATRIKS TRANSFORMASI

Mesin aljabar linier dapat digunakan untuk mengekspresikan banyak operasi yang diperlukan untuk mengatur objek dalam scene 3D, melihatnya dengan kamera, dan menampilkannya ke layar.

Kita akan menunjukkan bagaimana sekumpulan titik berubah jika titik-titik direpresentasikan sebagai vektor offset dari titik asal, dan kita akan menggunakan jam yang ditunjukkan pada Gambar 6.1 sebagai contoh himpunan titik. Jadi pikirkan jam sebagai sekelompok titik yang merupakan ujung vektor yang ekornya berada di titik asal. Kami juga membahas bagaimana transformasi ini beroperasi secara berbeda pada lokasi (titik), vektor perpindahan, dan vektor normal permukaan.

6.1 TRANSFORMASI LINIER 2D

Kita dapat menggunakan matriks 2×2 untuk mengubah, atau mentransformasi, sebuah vektor 2D:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}$$

Operasi semacam ini, yang mengambil 2-vektor dan menghasilkan 2-vektor lain dengan perkalian matriks sederhana, adalah transformasi linier.

Dengan rumus sederhana ini kita dapat mencapai berbagai transformasi yang berguna, tergantung pada apa yang kita masukkan ke dalam entri matriks, seperti yang akan dibahas pada bagian berikut. Untuk tujuan kita, pertimbangkan untuk bergerak sepanjang sumbu x sebagai gerakan horizontal dan sepanjang sumbu y, sebagai gerakan vertikal.

Scalling

Transformasi paling dasar adalah skala sepanjang sumbu koordinat. Transformasi ini dapat mengubah panjang dan mungkin arah:

$$scale\ (s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Perhatikan apa yang dilakukan matriks ini terhadap vektor dengan komponen Cartesian (x, y):

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$

Jadi, hanya dengan melihat matriks skala sejajar sumbu, kita dapat membaca dua faktor skala.

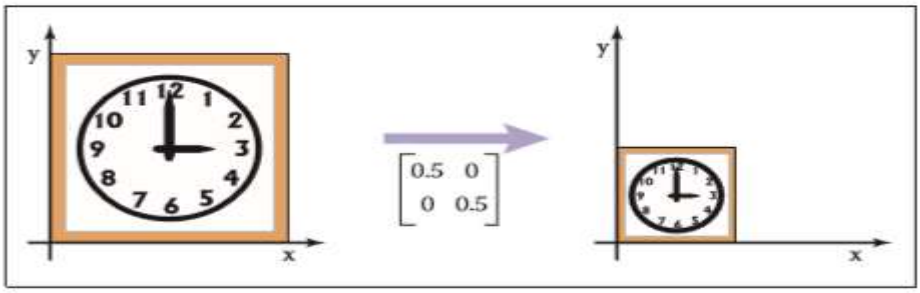
Contoh

Matriks yang mengecilkan x dan y beraturan dengan faktor dua adalah (Gambar 6.1)

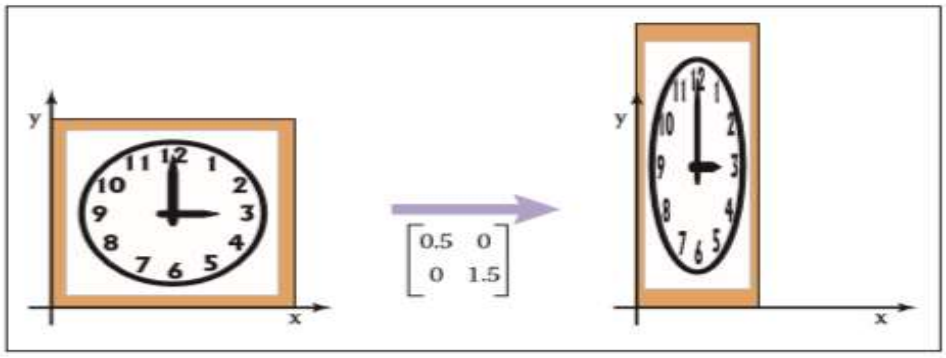
$$Scale(0.05, 0.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

Sebuah matriks yang membagi dua secara horizontal dan meningkat tiga bagian dalam vertikal adalah (lihat Gambar 6.2)

$$Scale(0.05, 1.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$



Gambar6.1 Scalling setengah secara seragam untuk setiap sumbu: Matriks skala sejajar sumbu memiliki proporsi perubahan di setiap elemen diagonal dan nol pada elemen di luar diagonal.



Gambar 6.2 Scalling tidak seragam dalam x dan y: Matriks scalling diagonal dengan elemen yang tidak sama. Perhatikan bahwa garis persegi jam menjadi persegi panjang dan permukaan lingkaran menjadi elips.

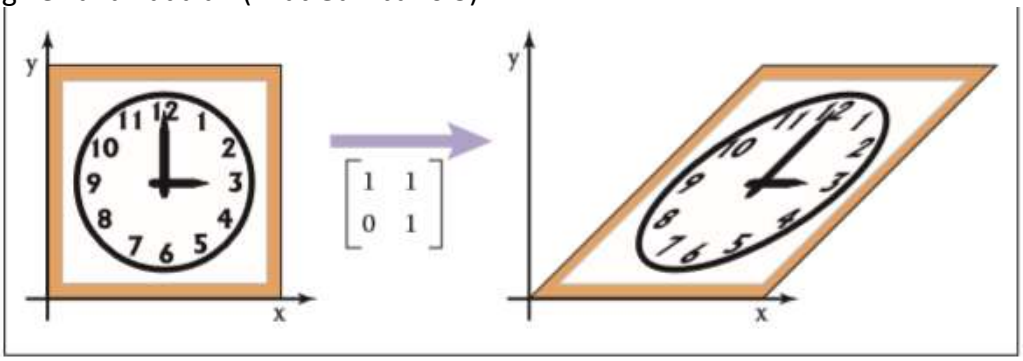
Shearing

Shear adalah sesuatu yang mendorong sesuatu ke samping, menghasilkan sesuatu seperti setumpuk kartu tempat Anda mendorong tangan Anda; kartu bawah tetap terpasang dan kartu bergerak lebih dekat ke bagian atas geladak. Matriks shearing horizontal dan vertikal adalah

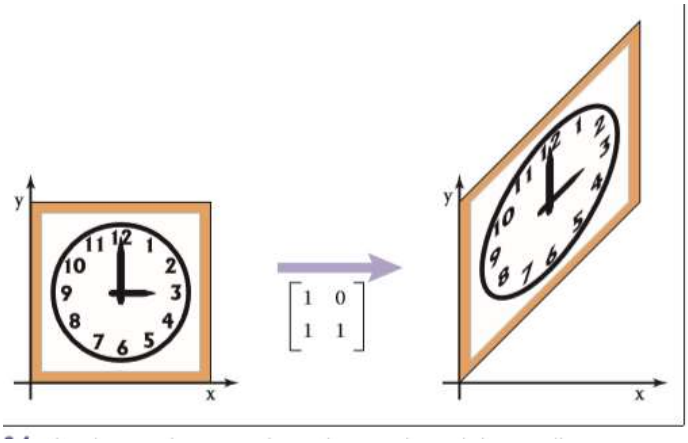
$$Shear - x(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}, Shear - y(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

Contoh

Transformasi yang memotong horizontal sehingga garis vertikal menjadi 45◦ garis condong ke kanan adalah (lihat Gambar 6.3)



Gambar 6.3 Sebuah matriks x-shear bergerak ke kanan sebanding dengan koordinat y-nya. Sekarang garis persegi jam menjadi jajar genjang dan, seperti scalling, permukaan jam yang melingkar menjadi elips.



Gambar 6.4 Sebuah matriks y-shear bergerak ke atas sebanding dengan koordinat x-nya.
Transformasi analog secara vertikal adalah (lihat Gambar 6.4)

$$shear - y(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Dalam kedua kasus, garis bujur sangkar dari jam yang digeser menjadi jajar genjang, dan permukaan melingkar dari jam yang digeser menjadi elips.

Cara lain untuk memikirkan geser adalah dalam hal rotasi hanya sumbu vertikal (atau horizontal). Transformasi geser yang mengambil sumbu vertikal dan memiringkannya searah jarum jam dengan sudut Ø adalah

$$\begin{bmatrix} 1 & \tan\emptyset \\ 0 & 1 \end{bmatrix}$$

Demikian pula matriks geser yang memutar sumbu horizontal berlawanan arah jarum jam dengan sudut adalah

$$\begin{bmatrix} 1 & 0 \\ \tan\emptyset & 1 \end{bmatrix}$$

Rotasi

Misalkan kita ingin memutar vektor a dengan sudut Ø berlawanan arah jarum jam untuk mendapatkan vektor b (Gambar 6.5). Jika a membentuk sudut dengan sumbu x dan panjangnya $r = \sqrt{x_a^2 + y_a^2}$, maka diketahui bahwa

$$x_a = r\cos\alpha$$
$$y_a = r\sin\alpha$$

Karena b adalah rotasi dari a, ia juga memiliki panjang r. Karena diputar membentuk sudut dari a, b membentuk sudut (α + Ø) dengan sumbu x. Menggunakan identitas penjumlahan trigonometri (Bagian 2.3.3):

$$x_b = r\cos(\alpha + \emptyset) = r\cos\alpha\cos\emptyset - r\sin\alpha\sin\emptyset,$$
$$y_b = r\sin(\alpha + \emptyset) = r\sin\alpha\cos\emptyset + r\cos\alpha\sin\emptyset.$$

Mengganti $x_a = r\cos\alpha$ dan $y_a = r\sin\alpha$ menghasilkan

$$x_b = x_a\cos\emptyset - y_a\sin\emptyset$$
$$y_b = y_a\cos\emptyset + x_a\sin\emptyset.$$

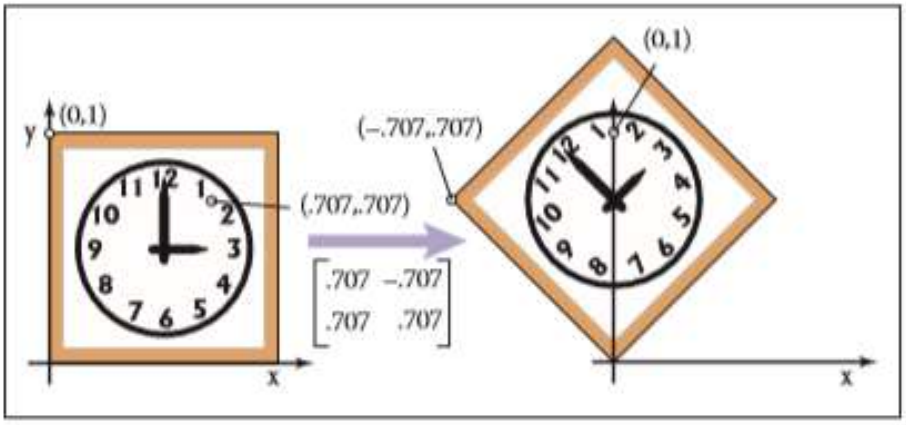
Dalam bentuk matriks, transformasi dari a ke b adalah

$$\text{rotasi}(\emptyset) = \begin{bmatrix} \cos\emptyset & -\sin\emptyset \\ \sin\emptyset & \cos\emptyset \end{bmatrix}$$

Contoh

Matriks yang memutar vektor sebesar π/4 radian (45 derajat) adalah (lihat Gambar 6.5)

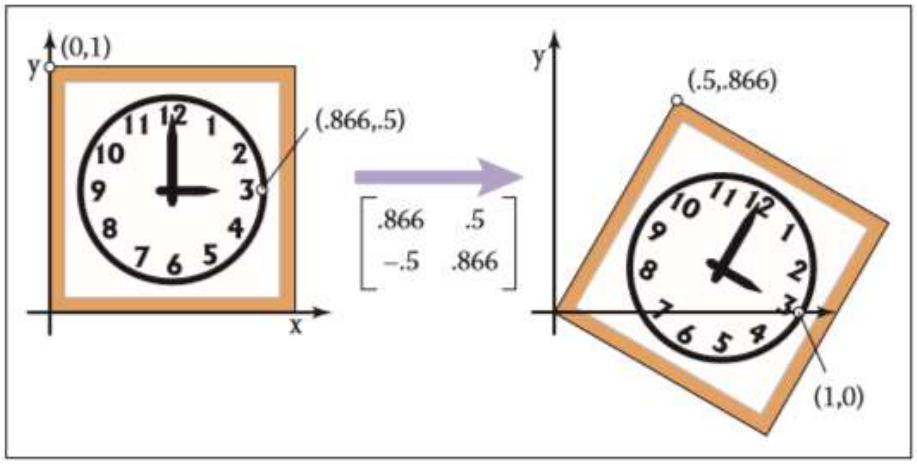
$$\begin{bmatrix} cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix}$$



Gambar 6.5 Rotasi sebesar 45◦. Perhatikan bahwa rotasi berlawanan arah jarum jam dan cos(45◦) = sin(45◦)≈.707.

Sebuah matrix yang berotasi sebesar π/6 radian (30 derajat) searah jarum jam adalah rotasi sebesar −π/6 radian dalam kerangka kerja kita (lihat Gambar 6.7):

$$\begin{bmatrix} cos\frac{-\pi}{6} & -\sin\frac{-\pi}{6} \\ \sin\frac{-\pi}{6} & \cos\frac{-\pi}{6} \end{bmatrix} = \begin{bmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{bmatrix}$$



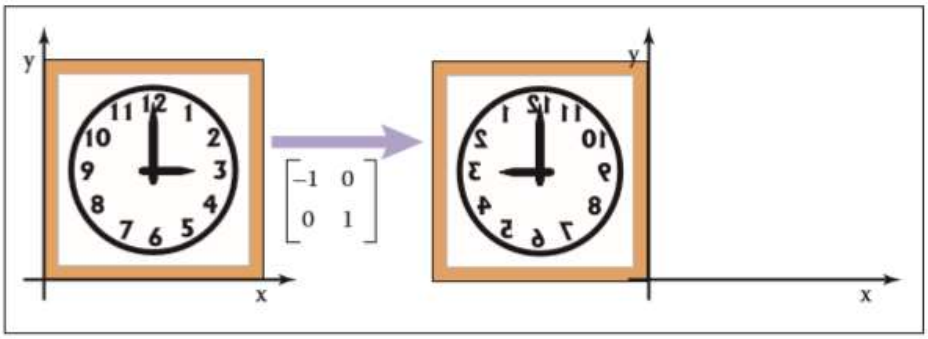
Gambar 6.6 Rotasi sebesar -30 derajat. Perhatikan bahwa rotasi searah jarum jam dan cos(–30◦) .866 dan sin(–30◦) = –.5.

Karena norma setiap baris matriks rotasi adalah satu ($\sin^2 Ø + \cos^2 = 1$), dan baris-barisnya ortogonal (cosØ(−sinØ) + sinØcosØ = 0), kita melihat bahwa matriks rotasi adalah matriks ortogonal (Bagian 5.2. 4). Dengan melihat matriks, kita dapat membaca dua pasang vektor ortonormal: dua kolom, yang merupakan vektor-vektor ke mana transformasi mengirimkan vektor basis kanonik (1,0) dan (0,1); dan baris, yang merupakan vektor yang dikirim oleh transformasi ke vektor basis kanonik.

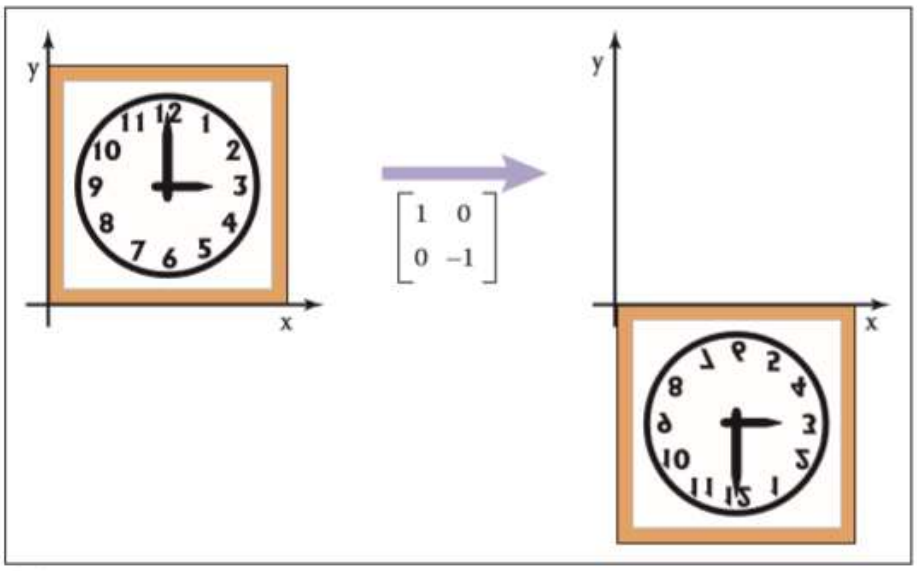
Refleksi

Kita dapat mencerminkan sebuah vektor melintasi salah satu sumbu koordinat dengan menggunakan skala dengan satu faktor skala negatif (lihat Gambar 6.7 dan 6.8):

$$refleksi - y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, refleksi - x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$



Gambar 6.7 Refleksi terhadap sumbu y diperoleh dengan mengalikan semua koordinat x dengan -1.



Gambar 6.8 Refleksi terhadap sumbu x diperoleh dengan mengalikan semua koordinat y dengan -1.

Sementara orang mungkin berharap bahwa matriks dengan 1 di kedua elemen diagonal juga merupakan refleksi, sebenarnya itu hanya rotasi oleh radian π.

Komposisi dan Dekomposisi Transformasi

Hal ini umum untuk program grafis untuk menerapkan lebih dari satu transformasi ke objek. Misalnya, pertama-tama kita mungkin ingin menerapkan skala S, dan kemudian rotasi R. Ini akan dilakukan dalam dua langkah pada vektor 2D v1:

Pertama, $v_2 = Sv_1$, kemudian, $v_3 = Rv_2$

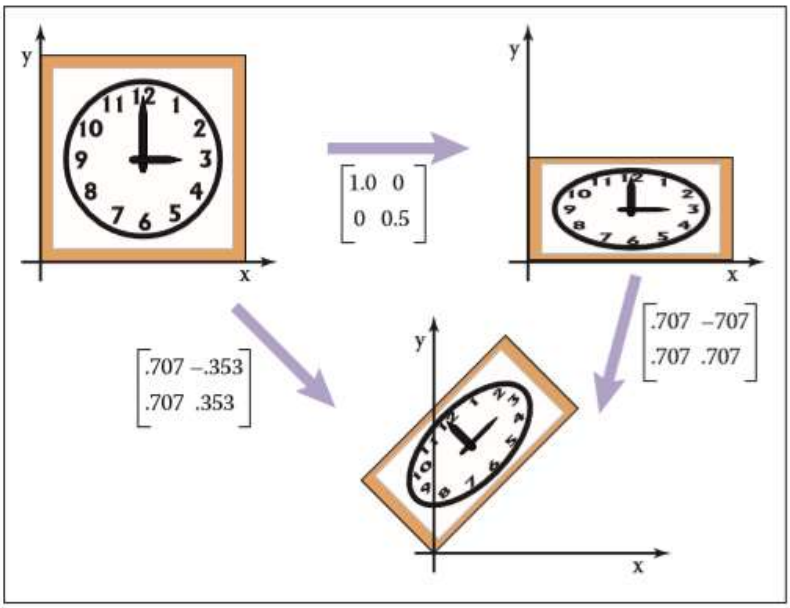
Cara lain untuk menulis ini adalah

$v_3 = R(Sv_1)$.

Karena perkalian matriks adalah asosiatif, kita juga dapat menulis

$v_3 = (RS)v_1$.

Dengan kata lain, kita dapat merepresentasikan efek transformasi vektor dengan dua matriks berurutan menggunakan matriks tunggal dengan ukuran yang sama, yang dapat kita hitung dengan mengalikan dua matriks: M = RS (Gambar 6.9). Sangat penting untuk diingat bahwa transformasi ini diterapkan dari sisi kanan terlebih dahulu. Jadi matriks M = RS pertama-tama menerapkan S dan kemudian R.



Gambar 6.9 Menerapkan dua matriks transformasi secara berurutan sama dengan menerapkan produk dari matriks-matriks tersebut satu kali. Ini adalah konsep kunci yang mendasari sebagian besar perangkat keras dan perangkat lunak grafis.

Contoh

Misalkan kita ingin menskalakan setengah dari arah vertikal dan kemudian memutarnya sebesar π/4 radian (45 derajat). Matriks yang dihasilkan adalah

$$\begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.353 \\ 0.707 & 0.353 \end{bmatrix}$$

Penting untuk selalu diingat bahwa perkalian matriks tidak komutatif. Jadi urutan transformasi itu penting. Dalam contoh ini, memutar terlebih dahulu, dan kemudian menskala, menghasilkan matriks yang berbeda (lihat Gambar 6.11):

$$\begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.353 \end{bmatrix}$$

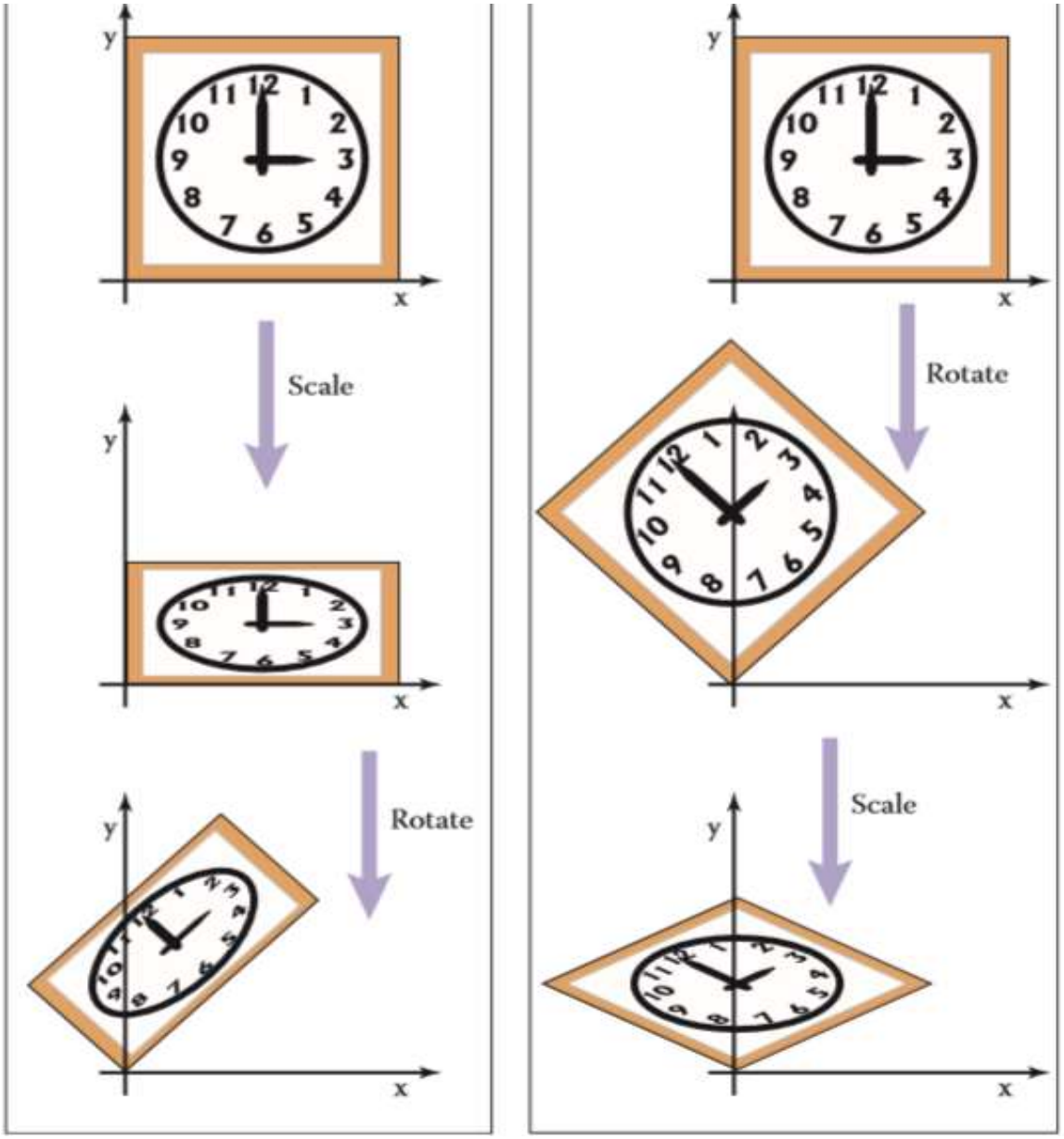
Contoh

Menggunakan matriks skala yang telah kami sajikan, scalling tidak seragam hanya dapat dilakukan di sepanjang sumbu koordinat. Jika kita ingin meregangkan jam kita sebesar 50% di sepanjang salah satu diagonalnya, sehingga 8:00 sampai 1:00 bergerak ke barat laut dan 2:00 sampai 7:00 bergerak ke tenggara, kita dapat menggunakan matriks rotasi dalam kombinasi dengan matriks scalling yang disejajarkan dengan sumbu untuk mendapatkan hasil yang kita inginkan. Idenya adalah menggunakan rotasi untuk menyelaraskan sumbu scalling dengan sumbu koordinat, lalu menskalakan sepanjang sumbu itu, lalu memutar kembali. Dalam contoh kita, sumbu scalling adalah diagonal "garis miring terbalik" dari bujur sangkar, dan kita dapat membuatnya sejajar dengan sumbu x dengan rotasi +45◦. Menempatkan operasi ini bersama-sama, transformasi penuh adalah

rotasi(−45◦) scale(1.5,1) rotasi(45◦).

Dalam notasi matematika, ini dapat ditulis RSRT. Hasil perkalian ketiga matriks tersebut adalah

$$\begin{bmatrix} 1.25 & -0.25 \\ -0.25 & 1.25 \end{bmatrix}$$

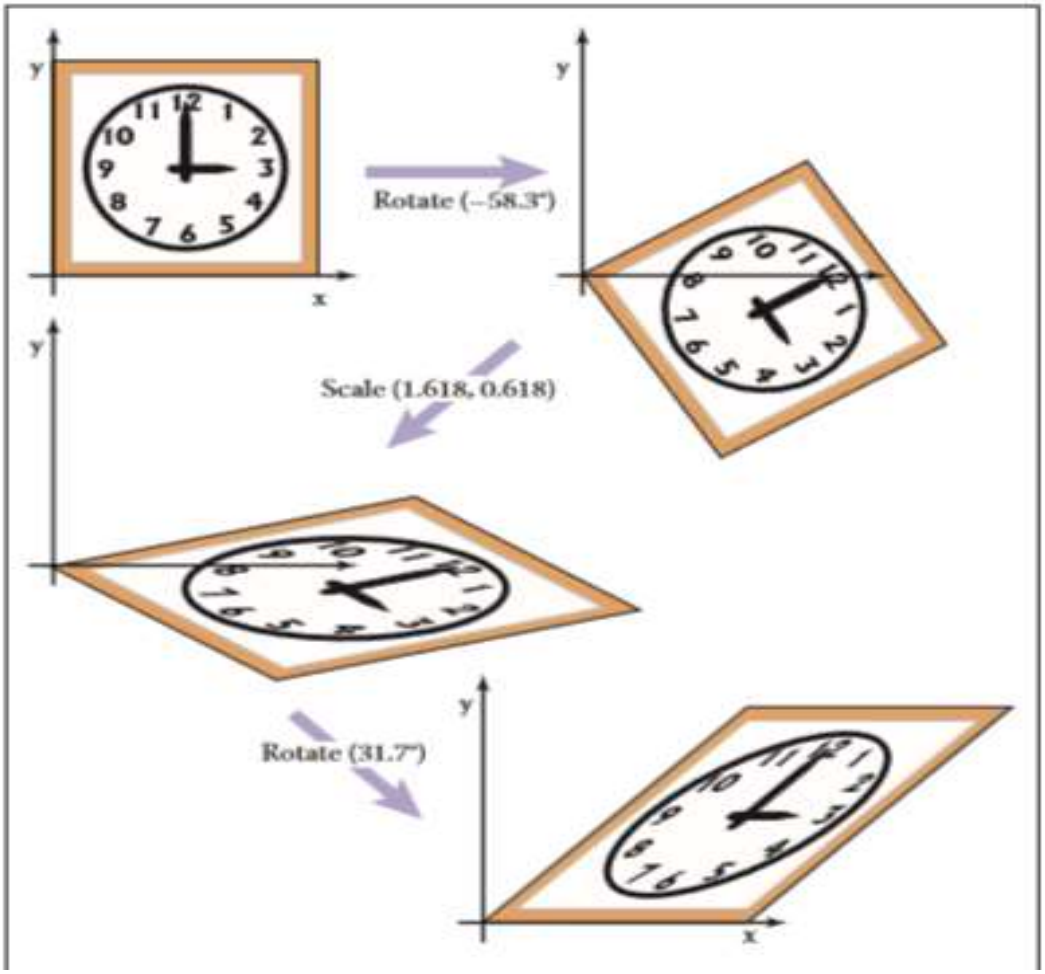


Gambar 6.10 Urutan penerapan dua transformasi biasanya penting. Dalam contoh ini, kami melakukan skala setengah dalam y dan kemudian memutar 45◦. Membalik urutan penerapan kedua transformasi ini akan menghasilkan hasil yang berbeda.

Membangun transformasi dari transformasi rotasi dan scalling benar-benar berfungsi untuk transformasi linier apa pun, dan fakta ini mengarah pada cara berpikir yang kuat tentang transformasi ini, seperti yang dieksplorasi di bagian berikutnya.

Dekomposisi Transformasi

Terkadang perlu untuk "membatalkan" komposisi transformasi, memisahkan transformasi menjadi bagian yang lebih sederhana. Misalnya, seringkali berguna untuk menyajikan transformasi kepada pengguna untuk manipulasi dalam hal rotasi dan faktor skala yang terpisah, tetapi transformasi mungkin direpresentasikan secara internal hanya sebagai matriks, dengan rotasi dan skala yang sudah dicampur bersama. Manipulasi semacam ini dapat dicapai jika matriks dapat dibongkar secara komputasi menjadi potongan-potongan yang diinginkan, potongan-potongan disesuaikan, dan matriks disusun kembali dengan mengalikan potongan-potongan itu lagi.



Gambar 6.11 Dekomposisi Nilai Singular/*Singular Value Decomposition* (SVD) untuk matriks geser. Setiap matriks 2D dapat didekomposisi menjadi produk rotasi, skala, rotasi. Perhatikan bahwa wajah lingkaran jam harus menjadi elips karena hanya lingkaran yang diputar dan diskalakan.

Ternyata dekomposisi ini, atau faktorisasi, adalah mungkin, terlepas dari entri dalam matriks—dan fakta ini memberikan cara berpikir yang bermanfaat tentang transformasi dan apa yang mereka lakukan terhadap geometri yang ditransformasikan olehnya.

Dekomposisi Nilai Eigen Simetris

Mari kita mulai dengan matriks simetris. Matriks simetris selalu dapat dipisahkan menggunakan dekomposisi nilai eigen menjadi produk berbentuk

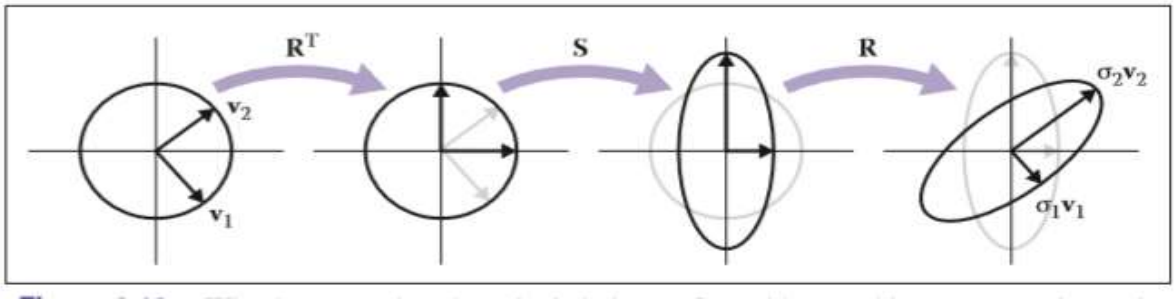
$$A = RSR^T$$

di mana R adalah matriks ortogonal dan S adalah matriks diagonal; kita akan memanggil kolom R (vektor eigen) dengan nama v_1 dan v_2, dan kita akan memanggil entri diagonal S (nilai eigen) dengan nama λ_1 dan λ_2. Dalam istilah geometris sekarang kita dapat mengenali R sebagai rotasi dan S sebagai skala, jadi ini hanyalah transformasi geometrik bertingkat (Gambar 6.12):

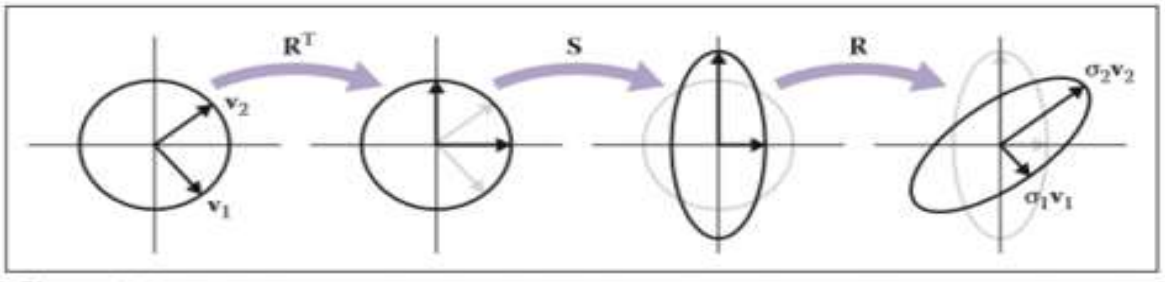
1. Putar v_1 dan v_2 ke sumbu x-dan y (transformasi oleh RT).
2. Skala dalam x dan y dengan (λ_1, λ_2) (transformasi dengan S).
3. Putar sumbu-x- dan y-balik ke v_1 dan v_2 (transformasi dengan R).

Jika Anda ingin menghitung dimensi: matriks 2 × 2 simetris memiliki 3 derajat kebebasan, dan dekomposisi nilai eigen menulis ulang sebagai sudut rotasi dan dua faktor skala. Melihat efek dari ketiga transformasi ini bersama-sama, kita dapat melihat bahwa

mereka memiliki efek skala tidak seragam sepanjang sepasang sumbu. Seperti halnya skala sejajar sumbu, sumbunya tegak lurus, tetapi bukan sumbu koordinat; sebaliknya mereka adalah vektor eigen dari A. Ini memberi tahu kita sesuatu tentang apa artinya menjadi matriks simetris: matriks simetris hanyalah operasi scalling—walaupun berpotensi tidak seragam dan tidak sejajar sumbu.



Gambar 6.12 Apa yang terjadi ketika lingkaran satuan ditransformasikan oleh matriks simetris sembarang A, juga dikenal sebagai skala tak-sejajar tak-seragam. Dua vektor tegak lurus v_1 dan v_2, yang merupakan vektor eigen dari A, tetap dalam arah tetapi diperkecil. Dalam hal transformasi dasar, ini dapat dilihat sebagai pertama-tama memutar vektor eigen ke basis kanonik, melakukan skala sejajar sumbu, dan kemudian memutar basis kanonik kembali ke vektor eigen.



Gambar 6.13 Matriks simetris selalu merupakan skala sepanjang beberapa sumbu. Dalam hal ini sepanjang arah Ø = 31,7◦ yang berarti vektor eigen real untuk matriks ini berada pada arah tersebut.

Contoh

Ingat contoh dari bab 5 :

$$\begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix} = R\begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}R^T$$

$$= \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}2.618 & 0\\ 0 & 0.382\end{bmatrix}\begin{bmatrix}0.8507 & 0.5257\\ -0.5257 & 0.8507\end{bmatrix}$$

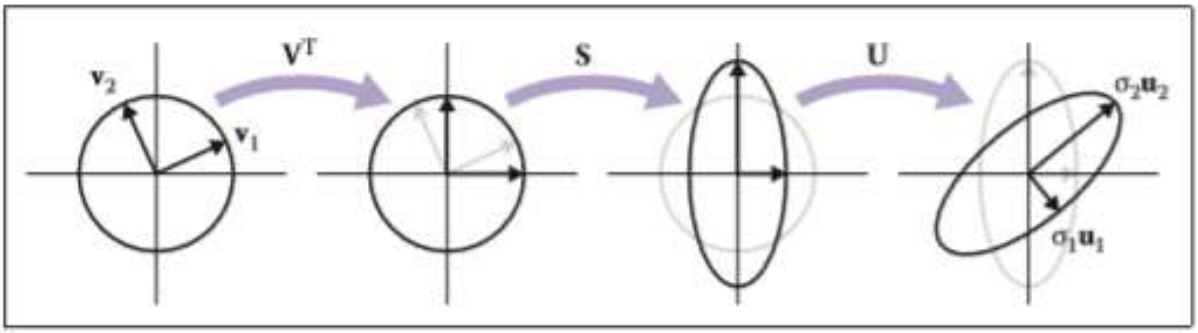
$$= rotasi(31.7°)scale(2.618, 0.382)rotasi(-31.7°)$$

Matriks di atas, kemudian, menurut dekomposisi nilai eigennya, berskala dalam arah 31,7◦ berlawanan arah jarum jam dari jam tiga (sumbu x). Ini adalah sentuhan sebelum jam 2 siang. pada permukaan jam seperti yang ditegaskan oleh Gambar 6.13. Kita juga dapat membalikkan proses diagonalisasi; untuk skala dengan (λ1,λ2) dengan arah scalling pertama sudut searah jarum jam dari sumbu x, kami memiliki

$$\begin{bmatrix}\cos\emptyset & \sin\emptyset\\ -\sin\emptyset & \cos\emptyset\end{bmatrix}\begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}\begin{bmatrix}\cos\emptyset & -\sin\emptyset\\ \sin\emptyset & \cos\emptyset\end{bmatrix}$$

$$= \begin{bmatrix}\lambda_1 cos^2\emptyset + \lambda_2 cos^2\emptyset & (\lambda_2 - \lambda_1)\cos\emptyset\, sin\,\emptyset\\ (\lambda_2 - \lambda_1)\cos\emptyset\, sin\,\emptyset & \lambda_2 cos^2\emptyset + \lambda_1 cos^2\emptyset\end{bmatrix}$$

Kita harus berhati-hati bahwa ini adalah matriks simetris seperti yang kita tahu pasti benar karena kita membangunnya dari dekomposisi nilai eigen simetris.



Gambar 6.14 Apa yang terjadi jika lingkaran satuan diubah oleh matriks sembarang A. Dua vektor tegak lurus v_1 dan v_2, yang merupakan vektor tunggal kanan A, diskalakan dan diubah arahnya agar sesuai dengan vektor tunggal kiri, u_1 dan u_2. Dalam hal transformasi dasar, ini dapat dilihat sebagai pertama-tama memutar vektor singular kanan ke basis kanonik, melakukan skala sejajar sumbu, dan kemudian memutar basis kanonik ke vektor singular kiri.

Dekomposisi Nilai Singular

Jenis dekomposisi yang sangat mirip juga dapat dilakukan dengan matriks nonsimetris: ini adalah Dekomposisi Nilai Singular (SVD), juga dibahas dalam Bagian 5.4.1. Perbedaannya adalah bahwa matriks di salah satu sisi matriks diagonal tidak lagi sama:

$$A = USV^T$$

Dua matriks ortogonal yang menggantikan rotasi tunggal R disebut U dan V, dan kolomnya masing-masing disebut ui (vektor tunggal kiri) dan vi (vektor tunggal kanan). Dalam konteks ini, entri diagonal dari S disebut nilai singular daripada nilai eigen. Interpretasi geometrinya sangat mirip dengan dekomposisi nilai eigen simetris (Gambar 6.14):

1. Putar v_1 dan v_2 ke sumbu x-andy (transformasi oleh V^T).
2. Skala dalam x dan y dengan (σ_1, σ_2) (transformasi dengan S).
3. Putar sumbu x- dan y ke u_1 dan u_2 (transformasi oleh U).

Untuk penghitung dimensi: matriks umum 2 × 2 memiliki 4 derajat kebebasan, dan SVD menulis ulangnya sebagai dua sudut rotasi dan dua faktor skala. Satu bit lagi diperlukan untuk melacak refleksi, tapi itu tidak menambah dimensi.

Perbedaan utama adalah antara satu rotasi dan dua matriks ortogonal yang berbeda. Perbedaan ini menyebabkan perbedaan lain yang kurang penting. Karena SVD memiliki vektor singular yang berbeda pada kedua sisinya, nilai singular negatif tidak diperlukan: kita selalu dapat membalik tanda nilai singular, membalikkan arah salah satu vektor singular terkait, dan berakhir dengan transformasi yang sama lagi. Untuk alasan ini, SVD selalu menghasilkan matriks diagonal dengan semua entri positif, tetapi matriks U dan V tidak dijamin rotasi—mereka juga dapat menyertakan refleksi. Dalam aplikasi geometris seperti grafis, ini adalah ketidaknyamanan, tetapi masalah kecil: mudah untuk membedakan rotasi dari pantulan dengan memeriksa determinannya, yaitu +1 untuk rotasi dan −1 untuk pantulan, dan jika diinginkan rotasi, salah satu dari nilai dapat dinegasikan, menghasilkan urutan rotasi-skala-rotasi di mana pantulan digulung dengan skala, bukan dengan salah satu rotasi.

Contoh

Contoh yang digunakan dalam Bagian 5.4.1 sebenarnya adalah matriks geser (Gambar 6.11):

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = R_2 \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} R_1$$

$$= \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} 1.618 & 0 \\ 0 & 0.618 \end{bmatrix} \begin{bmatrix} 0.5257 & 0.8507 \\ -0.8507 & 0.5257 \end{bmatrix}$$

= rotasi(31.7°) scale(1.618) rotasi(-58.3°)

Konsekuensi langsung dari keberadaan SVD adalah bahwa semua matriks transformasi 2D yang telah kita lihat dapat dibuat dari matriks rotasi dan matriks skala. Matriks geser adalah kemudahan, tetapi tidak diperlukan untuk mengekspresikan transformasi.

Singkatnya, setiap matriks dapat didekomposisi melalui SVD menjadi rotasi kali skala kali rotasi lain. Hanya matriks simetris yang dapat diuraikan melalui diagonalisasi nilai eigen ke dalam suatu rotasi dikalikan dengan skala dikali rotasi terbalik, dan matriks tersebut merupakan skala sederhana dalam arah sembarang. SVD dari matriks simetris akan menghasilkan produk rangkap tiga yang sama sebagai dekomposisi nilai eigen melalui manipulasi aljabar yang sedikit lebih kompleks.

Dekomposisi Rotasi Paeth

Dekomposisi lain menggunakan gunting untuk mewakili rotasi bukan nol (Paeth, 1990). Identitas berikut memungkinkan ini:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = R2 \begin{bmatrix} \sigma 1 & 1 \\ 0 & \sigma 2 \end{bmatrix} = R1$$

$$= \begin{bmatrix} \cos \emptyset & -\sin \emptyset \\ \sin \emptyset & \cos \emptyset \end{bmatrix} = \begin{bmatrix} 1 & \frac{\cos \emptyset - 1}{\sin \emptyset} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin \emptyset & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{\cos \emptyset - 1}{\sin \emptyset} \\ 0 & 1 \end{bmatrix}$$

= rotasi(31.7°) scale(1.618, 0.618) rotasi(-58.3°)

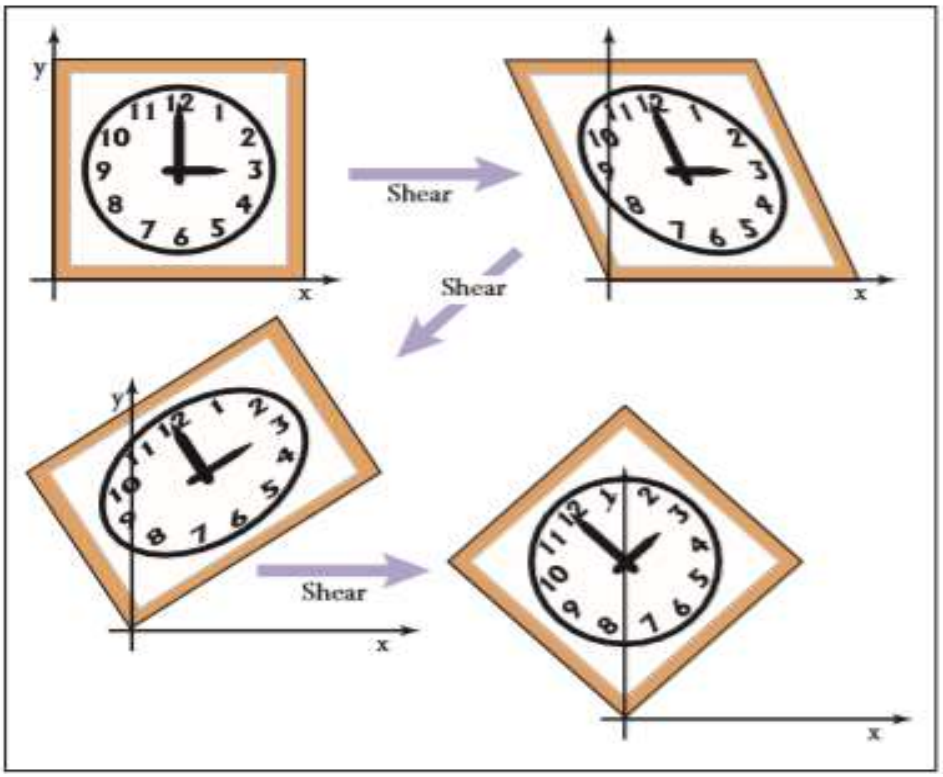
Misalnya, rotasi sebesar π/4 (45 derajat) adalah (lihat Gambar 6.15)

$$rotasi\left(\frac{\pi}{4}\right) = \begin{bmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{2}}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1-\sqrt{2} \\ 0 & 1 \end{bmatrix}$$

Transformasi khusus ini berguna untuk rotasi raster karena geser adalah operasi raster yang sangat efisien untuk gambar; itu memperkenalkan beberapa jagginess, tetapi tidak akan meninggalkan lubang. Pengamatan utama adalah bahwa jika kita mengambil posisi raster (i,j) dan menerapkan geser horizontal padanya, kita dapatkan

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i + sj \\ j \end{bmatrix}$$

Jika kita membulatkan sj ke bilangan bulat terdekat, ini berarti mengambil setiap baris dalam gambar dan memindahkannya ke samping dengan jumlah tertentu—jumlah yang berbeda untuk setiap baris. Karena ini adalah perpindahan yang sama dalam satu baris, ini memungkinkan kita untuk memutar tanpa celah pada gambar yang dihasilkan. Tindakan serupa bekerja untuk geser vertikal. Dengan demikian, kita dapat mengimplementasikan rotasi raster sederhana dengan mudah.



Gambar 6.15 Setiap rotasi 2D dapat dilakukan dengan tiga geser secara berurutan. Dalam hal ini rotasi sebesar 45◦ didekomposisi seperti yang ditunjukkan pada Persamaan 6.2.

6.2 TRANSFORMASI LINIER 3D

Transformasi 3D linier merupakan perpanjangan dari transformasi 2D. Misalnya, skala sepanjang sumbu Cartesian adalah

$$scale(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

Rotasi jauh lebih rumit dalam 3D daripada di 2D, karena ada lebih banyak sumbu rotasi yang mungkin. Namun, jika kita hanya ingin memutar sumbu z, yang hanya akan mengubah koordinat x- dan y, kita dapat menggunakan matriks rotasi 2D tanpa operasi pada z:

$$rotasi - z(\emptyset \begin{bmatrix} \cos\emptyset & -\sin\emptyset & 0 \\ \sin\emptyset & \cos\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Demikian pula kita dapat membuat matriks untuk berputar terhadap sumbu x dan sumbu y:

$$rotasi - x(\emptyset) = \begin{bmatrix} \cos\emptyset & 0 & \sin\emptyset \\ 0 & 1 & 0 \\ -\sin\emptyset & 0 & \cos\emptyset \end{bmatrix}$$

Kami akan membahas rotasi tentang sumbu sewenang-wenang di bagian selanjutnya. Seperti dalam dua dimensi, kita dapat menggeser sepanjang sumbu tertentu, misalnya,

$$shear - x(d_y, d_x) = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Seperti halnya transformasi 2D, matriks transformasi 3D apa pun dapat didekomposisi menggunakan SVD menjadi rotasi, skala, dan rotasi lainnya. Setiap matriks 3D simetris

memiliki dekomposisi nilai eigen menjadi rotasi, skala, dan rotasi terbalik. Akhirnya, rotasi 3D dapat didekomposisi menjadi produk matriks geser 3D.

Rotasi 3D Sewenang-wenang

Sebuah sin 2D, rotasi 3D adalah matriks ortogonal. Secara geometris, ini berarti bahwa ketiga baris matriks tersebut merupakan koordinat kartesius dari tiga vektor satuan yang saling ortogonal seperti yang dibahas pada Bagian 2.4.5. Kolom-kolom tersebut adalah tiga vektor satuan yang berpotensi berbeda dan saling ortogonal. Ada banyak matriks rotasi seperti itu. Mari kita tuliskan matriks seperti itu:

$$R_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

Di sini, $u = x_u x + y_u y + z_u z$ dan seterusnya untuk v dan w. Karena ketiga vektor tersebut ortonormal, kita tahu bahwa

$$u \cdot u = v \cdot v = w \cdot w = 1$$
$$u \cdot v = v \cdot w = w \cdot u = 0.$$

Kita dapat menyimpulkan beberapa perilaku matriks rotasi dengan menerapkannya pada vektor u, v dan w. Sebagai contoh,

$$R_{uvw}u = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} \begin{bmatrix} x_u \\ y_u \\ z_u \end{bmatrix} = \begin{bmatrix} x_u x_u + y_u y_u + z_u z_u \\ x_v x_u + y_v y_u + z_v z_u \\ x_w x_u + y_w y_u + z_w z_u \end{bmatrix}$$

Perhatikan bahwa ketiga baris R_{uvw} u itu semuanya adalah produk titik:

$$R_{uvw}u \begin{bmatrix} u \ . \ u \\ v \ . \ u \\ w \ . \ u \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = x$$

Demikian pula, $R_{uvw}v = y$, dan $R_{uvw}w = z$. Jadi R_{uvw} mengambil basis uvw ke sumbu Cartesian yang sesuai melalui rotasi.

Jika R_{uvw} adalah matriks rotasi dengan baris ortonormal, maka R^T_{uvw} juga merupakan matriks rotasi dengan kolom ortonormal, dan sebenarnya adalah kebalikan dari Ruvw (kebalikan dari matriks ortogonal selalu transposnya). Poin penting adalah bahwa untuk matriks transformasi, invers aljabar juga merupakan invers geometris. Jadi jika $R_{uv}w$ membawa u ke x, maka R^T_{uvw} membawa x ke u. Hal yang sama harus berlaku untuk v dan y seperti yang dapat kita konfirmasikan:

$$R^T_{uvw}y = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = v$$

Jadi kita selalu dapat membuat matriks rotasi dari basis ortonormal. Jika kita ingin memutar vektor sembarang a, kita dapat membentuk basis ortonormal dengan w = a, memutar basis tersebut ke basis kanonik xyz, memutar terhadap sumbu z, dan kemudian memutar basis kanonik kembali ke basis uvw. Dalam bentuk matriks, untuk memutar terhadap sumbu-w dengan sudut ∅:

$$\begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} \cos\emptyset & -\sin\emptyset & 0 \\ \sin\emptyset & \cos\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix}$$

Di sini kita memiliki vektor satuan dalam arah a (yaitu, a dibagi dengan panjangnya sendiri). Tapi apa kamu dan v? Sebuah metode untuk menemukan u dan v yang masuk akal diberikan dalam Bagian 2.4.6.

Jika kita memiliki matriks rotasi dan ingin memiliki rotasi dalam bentuk sumbu-sudut, kita dapat menghitung satu nilai eigen nyata (yang akan menjadi λ = 1), dan vektor eigen yang sesuai adalah sumbu rotasi. Ini adalah satu sumbu yang tidak berubah oleh rotasi. Lihat Bab

16 untuk perbandingan beberapa cara yang paling sering digunakan untuk merepresentasikan rotasi, selain matriks rotasi.

Mengubah Vektor Normal

Sementara sebagian besar vektor 3D yang kami gunakan mewakili posisi (vektor offset dari asal) atau arah, seperti dari mana cahaya berasal, beberapa vektor mewakili normal permukaan. Vektor normal permukaan tegak lurus terhadap bidang singgung suatu permukaan. Normal ini tidak berubah seperti yang kita inginkan ketika permukaan di bawahnya diubah. Misalnya, jika titik-titik suatu permukaan ditransformasikan oleh matriks M, vektor t yang bersinggungan dengan permukaan dan dikalikan dengan M akan menyinggung permukaan yang ditransformasi. Namun, vektor normal permukaan n yang ditransformasikan oleh M mungkin tidak normal terhadap permukaan yang ditransformasi (Gambar 6.16).

Kita dapat menurunkan matriks transformasi N yang membawa n ke vektor yang tegak lurus terhadap permukaan yang ditransformasi. Salah satu cara untuk mengatasi masalah ini adalah untuk mencatat bahwa vektor normal permukaan dan vektor tangen tegak lurus, sehingga produk titik mereka adalah nol, yang dinyatakan dalam bentuk matriks sebagai

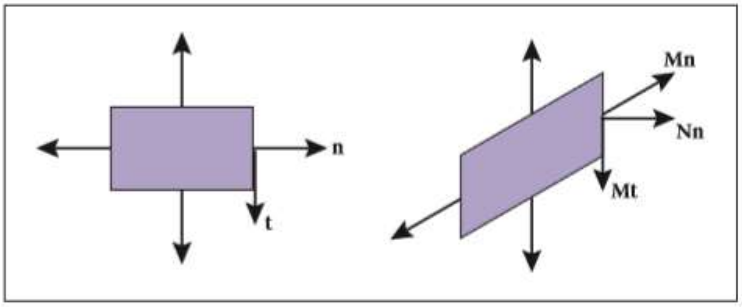
$$n^T t = 0.$$

Jika kita menyatakan vektor transformasi yang diinginkan sebagai $t_M = Mt$ dan $n_N = Nn$, tujuan kita adalah untuk menemukan N sedemikian rupa sehingga $n^T_N t_M = 0$. Kita dapat menemukan N dengan beberapa trik aljabar. Pertama, kita dapat menyelipkan matriks identitas ke dalam produk titik, dan kemudian memanfaatkan $M^{-1}M = I$:

$$n^T t = n^T I t = n^T M^{-1} M t = 0.$$

Meskipun manipulasi di atas tidak membawa kita ke mana-mana, perhatikan bahwa kita dapat menambahkan tanda kurung yang membuat ekspresi di atas lebih jelas sebagai produk titik:

$$(n^T M^{-1})\,(Mt) = (n^T M^{-1})\,t_M = 0.$$



Gambar 6.16 Ketika sebuah vektor normal ditransformasikan menggunakan matriks yang sama yang mentransformasikan titik-titik pada suatu objek, vektor yang dihasilkan mungkin tidak tegak lurus terhadap permukaan seperti yang ditunjukkan di sini untuk persegi panjang yang digeser. Namun, vektor tangen berubah menjadi vektor tangen pada permukaan yang ditransformasi.

Ini berarti bahwa vektor baris yang tegak lurus t_M adalah bagian kiri dari ekspresi di atas. Ekspresi ini berlaku untuk salah satu vektor tangen di bidang tangen. Karena hanya ada satu arah dalam 3D (dan kebalikannya) yang tegak lurus terhadap semua vektor tangen tersebut, kita tahu bahwa bagian kiri dari ekspresi di atas pasti adalah ekspresi vektor baris untuk n_N, yaitu n^T_N, jadi ini memungkinkan kita untuk menyimpulkan N:

$$n^T_N = n^T M^{-1}$$

jadi kita bisa mengambil transpos itu untuk mendapatkan

$$n_N = (n^T M^{-1})^T = (M^{-1})^T n.$$

Oleh karena itu, kita dapat melihat bahwa matriks yang mengubah vektor normal dengan benar sehingga tetap normal adalah N =$(M^{-1})^T$, yaitu transpos dari matriks terbalik. Karena matriks ini dapat mengubah panjang n, kita dapat mengalikannya dengan skalar sembarang dan akan tetap menghasilkan N dengan arah yang benar. Ingat dari Bagian 5.3 bahwa invers suatu matriks adalah transpos dari matriks kofaktor dibagi dengan determinan. Karena kita tidak peduli dengan panjang vektor normal, kita dapat melewatkan pembagian dan mencarinya untuk matriks 3x3,

$$N = \begin{bmatrix} m_{11}^c & m_{12}^c & m_{13}^c \\ m_{21}^c & m_{22}^c & m_{23}^c \\ m_{31}^c & m_{32}^c & m_{33}^c \end{bmatrix}$$

Ini mengasumsikan elemen M pada baris i dan kolom j adalah mij. Jadi ekspresi lengkap untuk N adalah

$$N = \begin{bmatrix} m_{22}m_{33} - m_{23}m_{32} & m_{23}m_{31} - m_{21}m_{33} & m_{21}m_{32} - m_{22}m_{31} \\ m_{13}m_{32} - m_{12}m_{33} & m_{11}m_{33} - m_{13}m_{31} & m_{12}m_{31} - m_{11}m_{32} \\ m_{12}m_{23} - m_{13}m_{22} & m_{13}m_{21} - m_{11}m_{23} & m_{11}m_{22} - m_{12}m_{21} \end{bmatrix}$$

6.3 TRANSLASI DAN TRANSFORMASI AFIN

Kami telah melihat metode untuk mengubah vektor menggunakan matriks M. Dalam dua dimensi, transformasi ini memiliki bentuk,

$$x' = m_{11}x + m_{12}y$$
$$y' = m_{21}x + m_{22}y.$$

Kita tidak dapat menggunakan transformasi semacam itu untuk memindahkan objek, hanya untuk menskalakan dan memutarnya. Secara khusus, titik asal (0, 0) selalu tetap di bawah transformasi linier. Untuk memindahkan, atau menerjemahkan, suatu objek dengan menggeser semua titiknya dengan jumlah yang sama, kita memerlukan transformasi bentuk,

$$x' = x + x_t$$
$$y' = y + y_t$$

Tidak ada cara untuk melakukannya dengan mengalikan (x, y) dengan matriks 2 × 2. Satu kemungkinan untuk menambahkan translasi ke sistem transformasi linier kami adalah dengan hanya mengasosiasikan vektor translasi terpisah dengan setiap matriks transformasi, membiarkan matriks menangani scalling dan rotasi dan vektor menangani translasi. Ini sangat mungkin dilakukan, tetapi pembukuannya canggung dan aturan untuk menyusun dua transformasi tidak sesederhana dan sebersih transformasi linier.

Sebagai gantinya, kita dapat menggunakan trik cerdas untuk mendapatkan perkalian matriks tunggal untuk melakukan kedua operasi bersama-sama. Idenya sederhana: nyatakan titik (x, y) dengan vektor 3D [x y 1]^T, dan gunakan matriks 3×3 berbentuk

$$\begin{bmatrix} M_{11} & M_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix}$$

Baris ketiga tetap berfungsi untuk menyalin 1 ke dalam vektor yang diubah, sehingga semua vektor memiliki 1 di tempat terakhir, dan dua baris pertama menghitung *x'* dan *y'* sebagai kombinasi linier dari x, y, dan1:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \begin{bmatrix} m_{11}x + m_{12}y + x_t \\ m_{21}x + m_{22}y + y_t \\ 1 \end{bmatrix}$$

Matriks tunggal mengimplementasikan transformasi linier diikuti dengan translasi! Transformasi semacam ini disebut transformasi affine, dan cara penerapan transformasi affine ini dengan menambahkan dimensi ekstra disebut koordinat homogen (Roberts, 1965; Riesenfeld, 1981; Penna & Patterson, 1986). Koordinat homogen tidak hanya membersihkan

kode untuk transformasi, tetapi skema ini juga memperjelas cara menyusun dua transformasi afin: cukup kalikan matriks.

Masalah dengan formalisme baru ini muncul ketika kita perlu mengubah vektor yang tidak seharusnya menjadi posisi—mereka mewakili arah, atau offset antar posisi. Vektor yang mewakili arah atau offset tidak boleh berubah ketika kita menerjemahkan suatu objek. Untungnya, kita dapat mengaturnya dengan mengatur koordinat ketiga ke nol:

$$\begin{bmatrix}1 & 0 & x_t\\ 0 & 1 & y_t\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ 0\end{bmatrix}=\begin{bmatrix}x\\ y\\ 0\end{bmatrix}$$

Jika ada transformasi scalling/rotasi pada entri matriks 2x2 kiri atas, itu akan berlaku untuk vektor, tetapi translasi masih mengalikan dengan nol dan diabaikan. Selanjutnya, nol disalin ke dalam vektor yang ditransformasikan, sehingga vektor arah tetap menjadi vektor arah setelah ditransformasikan.

Ini persis perilaku yang kita inginkan untuk vektor, sehingga mereka cocok dengan mulus ke dalam sistem: koordinat ekstra(ketiga) akan menjadi 1 atau 0 tergantung pada apakah pengkodean posisi atau arah. Kami sebenarnya perlu menyimpan koordinat homogen sehingga kami dapat membedakan antara lokasi dan vektor lainnya. Sebagai contoh,

$$\begin{bmatrix}3\\ 2\\ 1\end{bmatrix} adalah\ lokasi \quad dan \quad \begin{bmatrix}3\\ 2\\ 0\end{bmatrix} adalah\ direksi$$

Ini memberikan penjelasan untuk nama "homogen:" translasi, rotasi, dan scalling posisi dan arah semuanya masuk ke dalam satu sistem.

Kemudian, ketika kita melakukan tampilan perspektif, kita akan melihat bahwa berguna untuk membiarkan koordinat homogen mengambil nilai selain satu atau nol. Koordinat homogen digunakan hampir secara universal untuk mewakili transformasi dalam sistem grafis. Secara khusus, koordinat homogen mendasari desain dan operasi penyaji yang diimplementasikan dalam perangkat keras grafis. Kita akan melihat di Bab 7 bahwa koordinat homogen juga memudahkan menggambar scene dalam perspektif, alasan lain untuk popularitasnya. Koordinat homogen juga ada di mana-mana dalam visi komputer.

Koordinat homogen dapat dianggap sebagai cara cerdas untuk menangani pembukuan untuk terjemahan, tetapi ada juga interpretasi geometris yang berbeda. Pengamatan utama adalah bahwa ketika kita melakukan geser 3D berdasarkan koordinat z, kita mendapatkan transformasi ini:

$$\begin{bmatrix}1 & 0 & x_t\\ 0 & 1 & y_t\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\end{bmatrix}=\begin{bmatrix}x+x_tz\\ y+y_tz\\ z\end{bmatrix}$$

Perhatikan bahwa ini hampir memiliki bentuk yang kita inginkan dalam x dan y untuk terjemahan 2D, tetapi memiliki z berkeliaran yang tidak memiliki arti dalam 2D. Sekarang sampai pada keputusan kunci: kita akan menambahkan koordinat z = 1 ke semua lokasi 2D. Ini memberi kita

$$\begin{bmatrix}1 & 0 & x_t\\ 0 & 1 & y_t\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ 1\end{bmatrix}=\begin{bmatrix}x+x_tz\\ y+y_tz\\ 1\end{bmatrix}$$

Dengan mengasosiasikan a(z = 1)-koordinat dengan semua titik 2D, kita sekarang dapat mengkodekan terjemahan ke dalam bentuk matriks. Misalnya, untuk pertama menerjemahkan dalam 2D oleh (x_t, y_t) dan kemudian memutar dengan sudut kita akan menggunakan matriks

$$M=\begin{bmatrix}\cos\emptyset & -\sin\emptyset & 0\\ \sin\emptyset & \cos\emptyset & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & x_t\\ 0 & 1 & y_t\\ 0 & 0 & 1\end{bmatrix}$$

Perhatikan bahwa matriks rotasi 2D sekarang menjadi 3 × 3 dengan nol dalam "slot terjemahan". Dengan jenis formalisme ini, yang menggunakan geser sepanjang z =1 untuk mengkodekan translasi, kita dapat merepresentasikan sejumlah geser 2D, rotasi 2D, dan translasi 2D sebagai satu matriks 3D komposit. Baris bawah matriks itu akan selalu (0,0,1), jadi kita tidak perlu menyimpannya. Kita hanya perlu mengingat itu ada ketika kita mengalikan dua matriks bersama-sama.

Dalam 3D, teknik yang sama bekerja: kita dapat menambahkan koordinat keempat, koordinat homogen, dan kemudian kita memiliki terjemahan:

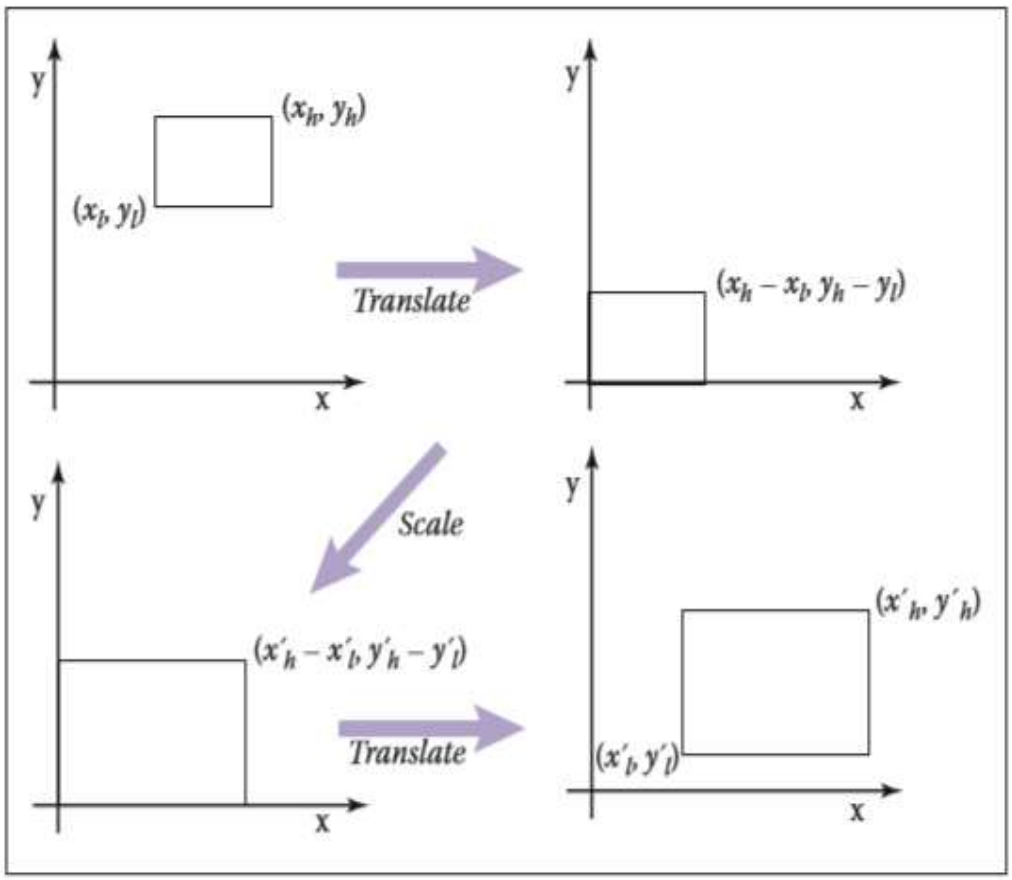
$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{bmatrix}$$

Sekali lagi, untuk vektor arah, koordinat keempat adalah nol dan vektor dengan demikian tidak terpengaruh oleh translasi.

Contoh (Transformasi Window).

Seringkali dalam grafisa kita perlu membuat matriks transformasi yang mengambil titik-titik dalam persegi panjang $[x'_l, x'_h] \times [y'_l, y'_h]$ ke persegi panjang $[x'_l, x'_h] \times [y'_l, y'_h]$. Hal ini dapat dicapai dengan skala tunggal dan menerjemahkan secara berurutan. Namun, lebih intuitif untuk membuat transformasi dari urutan tiga operasi (Gambar 6.17):

1. Pindahkan titik (x_l, y_l) ke titik asal.
2. Skala persegi panjang menjadi ukuran yang sama dengan persegi panjang target.
3. Pindahkan titik asal ke titik (x'_l, y'_l).



Gambar 6.17 Untuk mengambil satu persegi panjang (windows) ke yang lain, pertama-tama kita menggeser sudut kiri bawah ke titik asal, kemudian menskalakannya ke ukuran baru, dan kemudian memindahkan titik asal ke sudut kiri bawah persegi panjang target.

Mengingat bahwa matriks tangan kanan diterapkan terlebih dahulu, kita dapat menulis

$$\text{Window} = \text{translate } (x'_l, y'_1)\left(\frac{x'_h - x'_t}{x_h - x_t}, \frac{y'_h - y'_t}{y_h - y_t}\right) \; translate\,(-x_y, -y_t)$$

$$\begin{bmatrix} 1 & 0 & x'_t \\ 0 & 1 & y'_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x'_h - x'_t}{x_h - x_t} & 0 & 0 \\ - & \frac{y'_h - y'_t}{y_h - y_t} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_t \\ 0 & 1 & -y_t \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{x'_h - x'_t}{x_h - x_t} & 0 & \frac{y'_h x_h - y'_t y_t}{x_h - x_t} \\ 0 & \frac{y'_h - y'_t}{y_h - y_t} & \frac{y'_t y_h - y'_h y_t}{y_h - y_t} \\ 0 & 0 & 1 \end{bmatrix}$$

Mungkin tidak mengherankan bagi beberapa pembaca bahwa matriks yang dihasilkan memiliki bentuk seperti itu, tetapi proses konstruktif dengan tiga matriks tidak meninggalkan keraguan tentang kebenaran hasilnya.

Konstruksi analog yang tepat dapat digunakan untuk mendefinisikan transformasi windowing 3D, yang memetakan kotak $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ ke kotak $[x'_l, x'_h] \times [y'_l, y'_h] \times [z'_l, z'_h]$

$$\begin{bmatrix} \frac{x'_h - x'_t}{x_h - x_t} & 0 & 0 & \frac{y'_h x_h - y'_t y_t}{x_h - x_t} \\ 0 & \frac{y'_h - y'_t}{y_h - y_t} & 0 & \frac{y'_t y_h - y'_h y_t}{y_h - y_t} \\ 0 & 0 & \frac{z'_h - z'_t}{z_h - z_t} & \frac{z'_t z_h - z'_h z_t}{z_h - z_t} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sangat menarik untuk dicatat bahwa jika kita mengalikan matriks arbitrer yang terdiri dari skala, geser, dan rotasi dengan terjemahan sederhana (translasi berada di urutan kedua), kita mendapatkan

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dengan demikian, kita dapat melihat matriks apa pun dan menganggapnya sebagai bagian scalling/rotasi dan bagian translasi karena komponen-komponennya terpisah satu sama lain dengan baik. Kelas transformasi yang penting adalah transformasi benda tegar. Ini hanya terdiri dari translasi dan rotasi, sehingga mereka tidak memiliki peregangan atau penyusutan objek. Transformasi tersebut akan memiliki rotasi murni untuk a_{ij} di atas.

6.4 INVERS DARI TRANSFORMASI MATRIKS

Meskipun kita selalu dapat membalikkan matriks secara aljabar, kita dapat menggunakan geometri jika kita mengetahui apa yang dilakukan transformasi tersebut. Misalnya, kebalikan dari skala(s_x, s_y, s_z) adalah scale($1/s_x$, $1/s_y$, $1/s_z$). Kebalikan dari rotasi adalah rotasi yang sama dengan tanda yang berlawanan pada sudut. Invers dari suatu terjemahan adalah terjemahan yang berlawanan arah. Jika kita memiliki deret matriks $M = M_1M_2 \cdots M_n$ maka $M^{-1} = M_N^{-1}, \ldots M_2^{-1}M_1^{-1},$.

Juga, beberapa jenis matriks transformasi mudah dibalik. Kami telah menyebutkan skala, yang merupakan matriks diagonal; contoh penting kedua adalah rotasi, yang merupakan matriks ortogonal. Ingat (Bagian 5.2.4) bahwa invers dari matriks ortogonal adalah transposnya. Hal ini memudahkan untuk membalikkan rotasi dan transformasi benda tegar (lihat Latihan 6). Juga, berguna untuk mengetahui bahwa matriks dengan [0 0 0 1] di baris bawah memiliki invers yang juga memiliki [0 0 0 1] di baris bawah (lihat Latihan 7).

Menariknya, kita juga bisa menggunakan SVD untuk membalikkan matriks. Karena kita tahu bahwa matriks apa pun dapat didekomposisi menjadi rotasi kali skala dikalikan rotasi, inversi adalah mudah. Misalnya, dalam 3D kita memiliki

$$M = R_1 \text{scale}(\sigma_1, \sigma_2, \sigma_3) R_2,$$

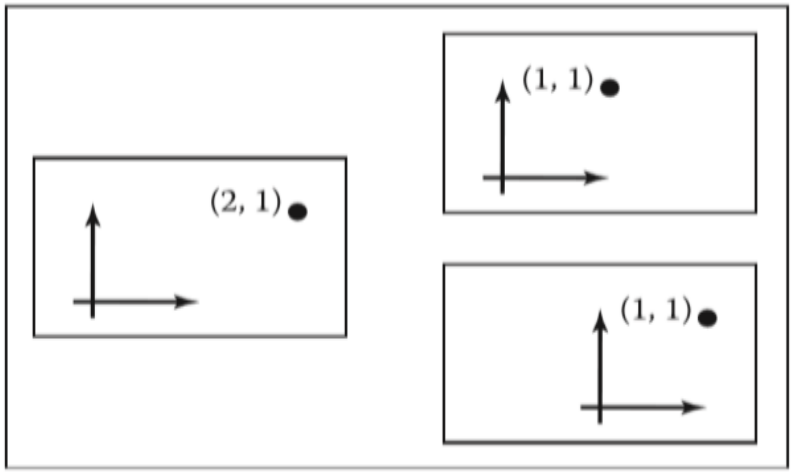
dan dari aturan di atas dengan mudah mengikuti bahwa

$$M^{-1} = R^T_2 \, \text{scale}(1/\sigma_1, 1/\sigma_2, 1/\sigma_3) R^T_1 .$$

6.5 TRANSFORMASI KOORDINAT

Semua diskusi sebelumnya telah menggunakan matriks transformasi untuk memindahkan poin. Kita juga dapat menganggapnya sebagai sekadar mengubah sistem koordinat di mana titik diwakili. Misalnya, pada Gambar 6.18, kita melihat dua cara untuk memvisualisasikan suatu gerakan. Dalam konteks yang berbeda, salah satu interpretasi mungkin lebih cocok.

Misalnya, permainan mengemudi mungkin memiliki model kota dan model mobil. Jika pemain disuguhi scene di luar kaca depan, objek di dalam mobil selalu digambar di tempat yang sama di layar, sementara jalan dan bangunan tampak bergerak mundur saat pemain mengemudi. Pada setiap bingkai, kami menerapkan transformasi ke objek-objek ini yang memindahkannya lebih jauh ke belakang daripada pada bingkai sebelumnya. Salah satu cara untuk memikirkan operasi ini adalah dengan memindahkan bangunan ke belakang; cara lain untuk memikirkannya adalah bahwa bangunan itu tetap berdiri tetapi sistem koordinat di mana kita ingin menggambarnya — yang melekat pada mobil — sedang bergerak. Pada interpretasi kedua, transformasi adalah mengubah koordinat geometri kota, dinyatakan sebagai koordinat dalam sistem koordinat mobil. Kedua cara akan mengarah ke matriks yang sama persis yang diterapkan pada geometri di luar mobil.



Gambar 6.18 Poin (2,1) memiliki transformasi "terjemahkan oleh (-1,0)" yang diterapkan padanya. Di kanan atas adalah gambar mental kita jika kita melihat transformasi ini sebagai gerakan fisik, dan di kanan bawah adalah gambar mental kita jika kita melihatnya sebagai perubahan koordinat (gerakan asal dalam hal ini). Batas buatan hanyalah sebuah artifisial, dan posisi relatif dari sumbu dan titik adalah sama dalam kedua kasus.

Jika permainan juga mendukung tampilan atas untuk menunjukkan di mana mobil berada di kota, bangunan dan jalan perlu digambar dalam posisi tetap sementara mobil perlu bergerak dari bingkai ke bingkai. Dua interpretasi yang sama berlaku: kita dapat menganggap transformasi yang berubah sebagai memindahkan mobil dari posisi kanoniknya ke lokasi saat ini di dunia; atau kita dapat menganggap transformasi sebagai sekadar mengubah koordinat geometri mobil, yang awalnya dinyatakan dalam hal sistem koordinat yang melekat pada mobil, untuk menyatakannya sebagai gantinya dalam sistem koordinat tetap relatif terhadap kota. Interpretasi perubahan koordinat memperjelas bahwa matriks yang digunakan dalam dua mode ini (perubahan koordinat kota-ke-mobil vs. perubahan koordinat mobil-ke-kota) adalah kebalikan satu sama lain.

Gagasan untuk mengubah sistem koordinat sangat mirip dengan gagasan tentang konversi tipe dalam pemrograman. Sebelum kita dapat menambahkan bilangan titik-mengambang ke bilangan bulat, kita perlu mengonversi bilangan bulat ke titik mengambang atau bilangan titik-mengambang menjadi bilangan bulat, tergantung pada kebutuhan kita, sehingga jenisnya cocok. Dan sebelum kita dapat menggambar kota dan mobil bersama-sama, kita perlu mengubah koordinat kota ke mobil atau koordinat mobil ke kota, tergantung pada kebutuhan kita, agar koordinatnya cocok.

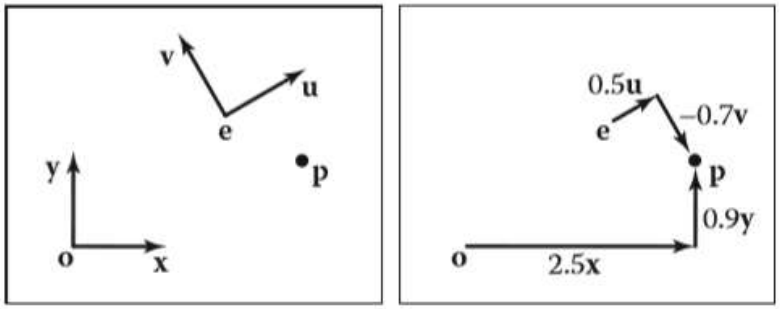
Saat mengelola beberapa sistem koordinat, mudah bingung dan berakhir dengan objek di koordinat yang salah, menyebabkannya muncul di tempat yang tidak terduga. Tetapi dengan pemikiran sistematis tentang transformasi antara sistem koordinat, Anda bisa mendapatkan transformasi yang benar dengan andal.

Secara geometris, sistem koordinat, atau kerangka koordinat, terdiri dari asal dan basis—satu set tiga vektor. Basis ortonormal begitu nyaman sehingga kita biasanya akan menganggap frame ortonormal kecuali ditentukan lain. Dalam bingkai dengan asal p dan basis {u,v,w}, koordinat (u,v,w) menggambarkan titik

$$p + uu + vv + ww.$$

Ketika kita menyimpan vektor-vektor ini di komputer, vektor-vektor tersebut perlu direpresentasikan dalam beberapa sistem koordinat. Untuk memulai, kita harus menetapkan beberapa sistem koordinat kanonik, yang sering disebut koordinat "global" atau "dunia", yang digunakan untuk menggambarkan semua sistem lainnya. Dalam contoh kota, kita mungkin mengadopsi grid jalan dan menggunakan konvensi bahwa sumbu x menunjuk sepanjang Jalan Utama, sumbu y menunjuk ke atas, dan sumbu z menunjuk sepanjang Central Avenue. Kemudian ketika kita menulis asal dan dasar kerangka mobil dalam hal koordinat ini, jelas apa yang kita maksud.

Dalam 2D konvensi kita adalah menggunakan titik o untuk titik asal, dan x dan y untuk vektor basis ortonormal tangan kanan x dan y (Gambar 6.19).



Gambar 6.19 Titik p dapat direpresentasikan dalam salah satu sistem koordinat.

Sistem koordinat lain mungkin memiliki vektor basis ortonormal asal dan tangan kanan u dan v. Perhatikan bahwa biasanya data kanonik o, x, andy tidak pernah disimpan secara eksplisit. Mereka adalah kerangka acuan untuk semua sistem koordinat lainnya. Dalam sistem koordinat itu, kita sering menuliskan lokasi p sebagai pasangan terurut, yang merupakan singkatan dari ekspresi vektor penuh:

$$p = (x_p, y_p) \equiv o + x_p x + y_p y.$$

Misalnya, pada Gambar 6.19, $(x_p, y_p) = (2.5, 0.9)$. Perhatikan bahwa pasangan (x_p, y_p) secara implisit mengasumsikan asal o. Demikian pula, kita dapat menyatakan p dalam persamaan lain:

$$p = u_p, v_p) \equiv e + u_p u + v_p v.$$

Pada Gambar 6.19, ini memiliki $(u_p, v_p) = (0.5, -0.7)$. Sekali lagi, asal e dibiarkan sebagai bagian implisit dari sistem koordinat yang terkait dengan u dan v. Kita dapat menyatakan hubungan yang sama ini menggunakan mesin matriks, seperti ini:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

Perhatikan bahwa ini mengasumsikan kita memiliki titik e dan vektor u dan v disimpan dalam koordinat kanonik; sistem koordinat (x, y) adalah yang pertama di antara yang sederajat. Dalam hal tipe dasar transformasi yang telah kita bahas dalam bab ini, ini adalah rotasi (melibatkan u dan v) diikuti dengan translasi (melibatkan e). Melihat matriks untuk rotasi dan translasi bersama-sama, Anda dapat melihat sangat mudah untuk menuliskannya: kita hanya menempatkan u, v, ande ke dalam kolom-kolom matriks, dengan [0 0 1] biasa di baris ketiga. Untuk membuatnya lebih jelas, kita dapat menulis matriks seperti ini:

$$p_{xy} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} p_{uv}$$

Kami menyebut matriks ini sebagai matriks bingkai-ke-kanonik untuk bingkai (u, v). Dibutuhkan poin yang dinyatakan dalam bingkai (u,v) dan mengubahnya menjadi titik yang sama yang dinyatakan dalam bingkai kanonik.

Catatan : Nama "frame-tocanonical" didasarkan pada pemikiran tentang mengubah koordinat vektor dari satu sistem ke sistem lainnya. Berpikir dalam hal vektor bergerak di sekitar, matriks bingkai-ke-kanonik memetakan bingkai kanonik ke bingkai (u,v).

Untuk pergi ke arah lain yang kita miliki

$$\begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_e \\ 0 & 1 & -y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}$$

Ini adalah terjemahan yang diikuti oleh rotasi; mereka adalah kebalikan dari rotasi dan translasi yang kami gunakan untuk membangun matriks bingkai-ke-kanonik, dan ketika dikalikan bersama, mereka menghasilkan kebalikan dari matriks bingkai-ke-kanonik, yang (tidak mengherankan) disebut matriks kanonik-ke- matriks bingkai:

$$p_{uv} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix}^{-1} p_{xy}$$

Matriks kanonik-ke-bingkai mengambil titik yang dinyatakan dalam bingkai kanonik dan mengubahnya menjadi titik yang sama yang dinyatakan dalam bingkai (u,v). Kami telah menulis matriks ini sebagai kebalikan dari matriks bingkai-ke-kanonik karena tidak dapat segera ditulis menggunakan koordinat kanonik dari e, u, danv. Tetapi ingat bahwa semua sistem koordinat adalah ekuivalen; hanya konvensi kami untuk menyimpan vektor dalam bentuk koordinat x-andy yang menciptakan asimetri ini. Matriks kanonik-ke-bingkai dapat dinyatakan secara sederhana dalam bentuk koordinat (u, v) dari o, x, dan y:

$$p_{uv} \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} p_{xy}$$

Semua ide ini bekerja secara analog dalam 3D, di mana kita memiliki

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix}$$

$$p_{uvw} \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix} p_{xyz}$$

Dan

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

$$p_{uvw} \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} p_{xyz}$$

Pertanyaan yang Sering Diajukan

- *Tidak bisakah saya mengubah hardcode saja daripada menggunakan formalisme matriks?*

Jawaban : Ya, tetapi dalam praktiknya lebih sulit untuk diturunkan, lebih sulit untuk di-debug, dan tidak lagi efisien. Juga, semua API grafis saat ini menggunakan formalisme matriks ini sehingga harus dipahami bahkan untuk menggunakan perpustakaan grafis.

- *Baris paling bawah dari matriks selalu (0,0,0,1). Apakah saya harus menyimpannya?*

Jawaban : Anda tidak perlu menyimpannya kecuali jika Anda menyertakan transformasi perspektif (Bab 7).

Latihan

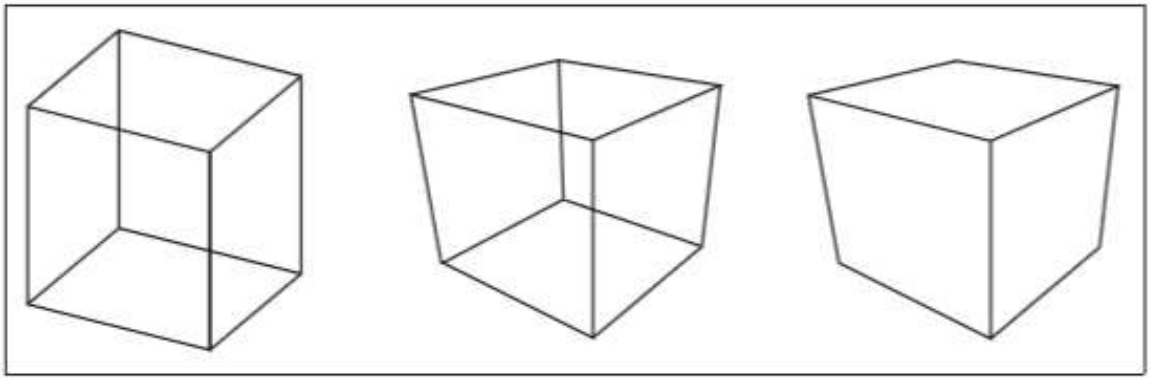
1. Tuliskan matriks 3D 4 × 4 yang akan dipindah (x_m,y_m,z_m).
2. Tuliskan matriks 3D 4 × 4 yang berotasi dengan sudut terhadap sumbu y.
3. Tulislah matriks 4 x 4 3D untuk menskalakan suatu benda sebesar 50% ke segala arah.
4. Tulis matriks rotasi 2D yang berputar 90 derajat searah jarum jam.
5. Tulis matriks dari Latihan 4 sebagai produk dari tiga matriks geser.
6. Temukan kebalikan dari transformasi benda tegar:
$$\begin{bmatrix} R & t \\ 000 & 1 \end{bmatrix}$$
di mana R adalah matriks rotasi 3×3 dan t adalah 3-vektor
7. Tunjukkan bahwa invers matriks untuk transformasi afin (yang memiliki semua nol di baris bawah kecuali satu di entri kanan bawah) juga memiliki bentuk yang sama.
8. Jelaskan dengan kata-kata apa yang dilakukan matriks transformasi 2D ini:
$$\begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$
9. Tulislah matriks 3 × 3 yang memutar titik 2D dengan sudutθ terhadap titik p = (x_p, y_p).
10. Tuliskan matriks rotasi 4 × 4 yang mengambil vektor 3D ortonormal u = (x_u, y_u, z_u), v = (x_v, y_v, z_v), dan w = (x_w, y_w, z_w), ke 3D ortonormal vektor a = (x_a, y_a, z_a), b = (x_b, y_b, z_b), and c = (x_c, y_c, z_c), Jadi Mu = a, Mv = b, dan Mw = c.
11. Apa matriks invers untuk jawaban soal sebelumnya?

BAB 7
VIEWING

Pada bab sebelumnya, kita telah melihat bagaimana menggunakan transformasi matriks sebagai alat untuk mengatur objek geometris dalam ruang 2D atau 3D. Penggunaan penting kedua dari transformasi geometri adalah dalam memindahkan objek antara lokasi 3D dan posisinya dalam tampilan 2D dari dunia 3D. Mapping 3D ke 2D ini disebut transformasi tampilan, dan ini memainkan peran penting dalam rendering urutan objek, di mana kita perlu menemukan lokasi ruang-gambar setiap objek dalam scene dengan cepat.

Ketika kita mempelajari ray tracing di Bab 4, kita membahas berbagai jenis perspektif dan pandangan ortografis dan bagaimana menghasilkan sinar pandang menurut pandangan yang diberikan. Jika Anda belum melihatnya baru-baru ini, disarankan untuk meninjau kembali pembahasan perspektif dan pembangkitan sinar di Bab 4 sebelum membaca bab ini.

Dengan sendirinya, kemampuan untuk memproyeksikan titik-titik dari dunia ke gambar hanya bagus untuk menghasilkan rendering wireframe—rendering di mana hanya tepi objek yang digambar, dan permukaan yang lebih dekat tidak menutup permukaan yang lebih jauh (Gambar 7.1). Sama seperti pelacak sinar yang perlu menemukan persimpangan permukaan terdekat di sepanjang setiap sinar pandang, penyaji urutan objek yang menampilkan objek tampak padat harus mencari tahu mana dari (mungkin banyak) permukaan yang digambar pada titik tertentu di layar yang paling dekat dan hanya menampilkan yang itu. Dalam bab ini, diasumsikan bahwa kita sedang menggambar model yang hanya terdiri dari segmen garis 3D yang ditentukan oleh koordinat (x, y, z) dari dua titik ujungnya. Bab selanjutnya akan membahas mesin yang dibutuhkan untuk menghasilkan rendering permukaan padat.



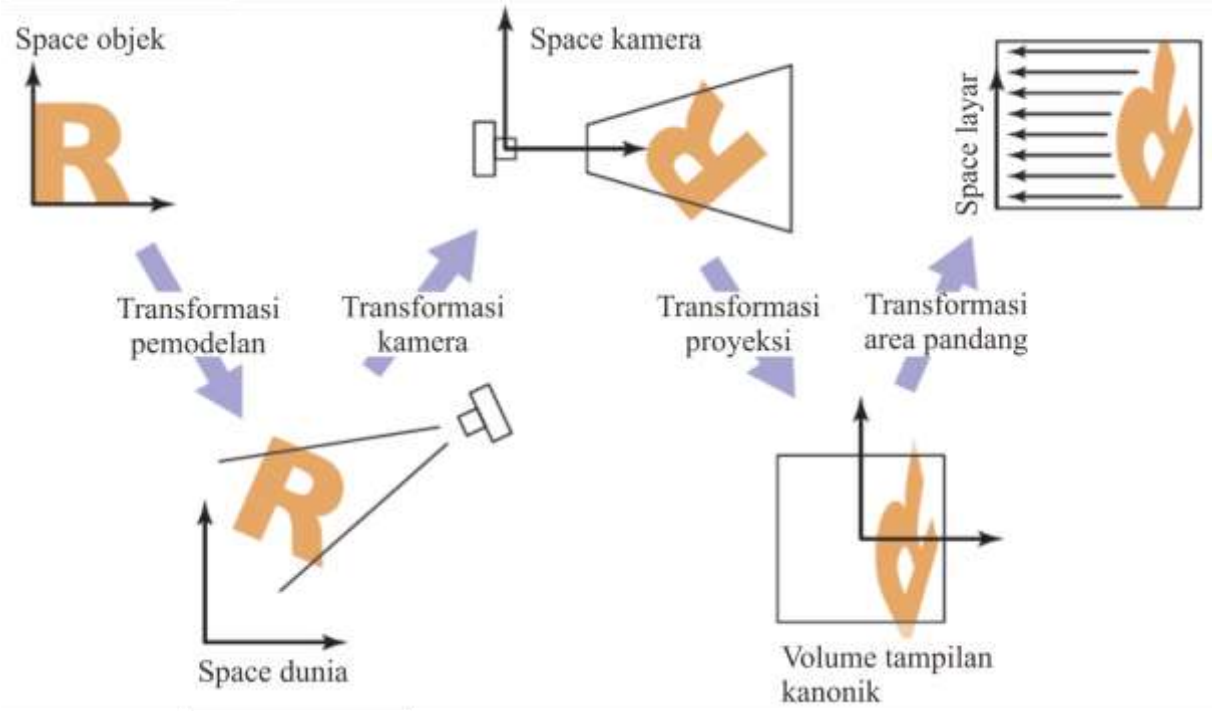
Gambar 7.1 Kiri: kubus gambar rangka dalam proyeksi ortografis. Tengah: kubus gambar rangka dalam proyeksi perspektif. Kanan: proyeksi perspektif dengan garis tersembunyi dihilangkan.

7.1 MELIHAT TRANSFORMASI

Transformasi tampilan memiliki tugas memetakan lokasi 3D, direpresentasikan sebagai koordinat (x,y,z) dalam sistem koordinat kanonik, ke koordinat dalam gambar, yang dinyatakan dalam satuan piksel. Ini adalah binatang yang rumit yang bergantung pada banyak hal yang berbeda, termasuk posisi dan orientasi kamera, jenis proyeksi, bidang pandang, dan resolusi gambar. Seperti halnya semua transformasi yang rumit, pendekatan terbaik adalah dengan memecahnya menjadi produk dari beberapa transformasi yang lebih sederhana. Sebagian besar sistem grafis melakukan ini dengan menggunakan urutan tiga transformasi:

- Transformasi kamera atau transformasi mata, yang merupakan transformasi tubuh kaku yang menempatkan kamera pada titik asal dalam orientasi yang nyaman. Itu hanya bergantung pada posisi dan orientasi, atau pose, kamera.
- Transformasi proyeksi, yang memproyeksikan titik-titik dari ruang kamera sehingga semua titik yang terlihat berada dalam kisaran−1 hingga 1 dalam x dan y. Itu hanya tergantung pada jenis proyeksi yang diinginkan.
- Transformasi viewport atau transformasi windowing, yang memetakan persegi panjang gambar unit ini ke persegi panjang yang diinginkan dalam koordinat piksel. Itu hanya tergantung pada ukuran dan posisi gambar output.

Untuk memudahkan dalam mendeskripsikan tahapan proses (Gambar 7.2), kami memberikan nama pada sistem koordinat yang merupakan input dan output dari transformasi tersebut. Transformasi kamera mengubah titik dalam koordinat kanonik (atau ruang dunia) menjadi koordinat kamera atau menempatkannya di ruang kamera. Transformasi proyeksi memindahkan titik dari ruang kamera ke volume tampilan kanonik. Terakhir, transformasi viewport memetakan volume tampilan kanonik ke ruang layar.



Gambar 7.2 Urutan ruang dan transformasi yang mendapatkan objek dari koordinat aslinya ke ruang layar.

Masing-masing transformasi ini secara individual cukup sederhana. Kami akan membahasnya secara rinci untuk kasus ortografis yang dimulai dengan transformasi viewport, kemudian mencakup perubahan yang diperlukan untuk mendukung proyeksi perspektif.
Catatan : Nama lain: ruang kamera juga merupakan “ruang mata” dan transformasi kamera terkadang merupakan “transformasi tampilan;” volume tampilan kanonik juga merupakan "ruang klip" atau "koordinat perangkat yang dinormalisasi;" ruang layar juga "koordinat piksel."

Transformasi Viewport

Kami mulai dengan masalah yang solusinya akan digunakan kembali untuk kondisi tampilan apa pun. Kami berasumsi bahwa geometri yang ingin kami lihat berada dalam volume tampilan kanonik, dan kami ingin melihatnya dengan kamera ortografis yang melihat ke arah z. Volume tampilan kanonik adalah kubus yang berisi semua titik 3D yang koordinat

Cartesiannya antara 1 dan +1—yaitu, (x, y, z) ∈ [−1, 1]^3 (Gambar 7.3). Kami memproyeksikan x = 1 ke sisi kiri layar, x = +1 ke sisi kanan layar, y = 1 ke bagian bawah layar, dan y = +1 ke bagian atas layar. Kata "kanonik" muncul lagi—artinya sesuatu yang dipilih secara sewenang-wenang demi kenyamanan. Misalnya, lingkaran satuan bisa disebut "lingkaran kanonik." Ingat konvensi untuk koordinat piksel dari Bab 3: setiap piksel “memiliki” unit persegi yang berpusat pada koordinat bilangan bulat; batas gambar memiliki setengah unit overshoot dari pusat piksel; dan koordinat pusat piksel terkecil adalah (0,0). Jika kita menggambar ke dalam gambar (atau windows di layar) yang memiliki piksel nx kali ny, kita perlu memetakan persegi [−1, 1]^2 ke persegi panjang [−0.5, n_x − 0.5] × [−0.5, n_y – 0,5].

Untuk saat ini, kita akan mengasumsikan bahwa semua segmen garis yang akan digambar sepenuhnya berada di dalam volume tampilan kanonik. Nanti kita akan mengendurkan asumsi itu ketika kita membahas kliping. Karena transformasi view port memetakan satu persegi panjang sejajar sumbu ke yang lain, ini adalah kasus transformasi windowing yang diberikan oleh Persamaan (6.6):

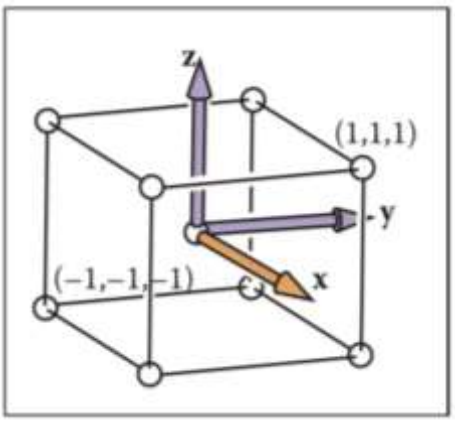
Persamaan 7.1

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ 1 \end{bmatrix} \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ 1 \end{bmatrix}$$

Perhatikan bahwa matriks ini mengabaikan koordinat z dari titik-titik dalam volume tampilan kanonik, karena jarak titik di sepanjang arah proyeksi tidak memengaruhi tempat titik itu memproyeksikan dalam gambar. Tetapi sebelum kita secara resmi menyebutnya matriks viewport, kita menambahkan baris dan kolom untuk membawa koordinat z tanpa mengubahnya. Kita tidak membutuhkannya dalam bab ini, tetapi pada akhirnya kita akan membutuhkan nilai z karena nilai tersebut dapat digunakan untuk membuat permukaan yang lebih dekat menyembunyikan permukaan yang lebih jauh (lihat Bagian 8.2.3).

$$M_{vp} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_2 - 1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Memetakan persegi ke persegi panjang yang berpotensi non-persegi tidak menjadi masalah; x dan y hanya berakhir dengan faktor skala yang berbeda dari koordinat kanonik ke piksel.



Gambar 7.3 Volume tampilan kanonik adalah kubus dengan panjang sisi dua berpusat di titik asal.

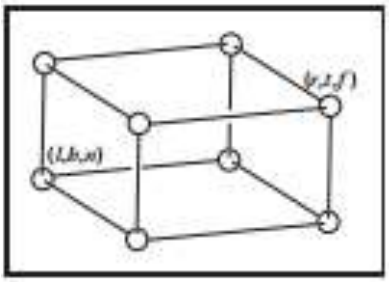
Transformasi Proyeksi Ortografis

Tentu saja, kita biasanya ingin menampilkan geometri di beberapa wilayah ruang selain volume tampilan kanonik. Langkah pertama kita dalam menggeneralisasi tampilan akan menjaga arah dan orientasi tampilan tetap terlihat sepanjang z dengan +y ke atas, tetapi akan memungkinkan sembarang persegi panjang untuk dilihat. Daripada mengganti matriks viewport, kita akan menambahnya dengan mengalikannya dengan matriks lain di sebelah kanan.

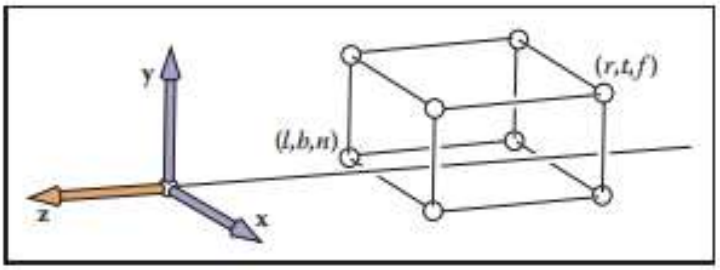
Di bawah batasan ini, volume tampilan adalah kotak sejajar sumbu, dan kami akan memberi nama koordinat sisi-sisinya sehingga volume tampilan adalah [l,r]×[b,t]×[f,n] yang ditunjukkan pada Gambar 7.4. Kami menyebut kotak ini volume tampilan ortografis dan mengacu pada bidang pembatas sebagai berikut:

$x = l \equiv$ left plane,
$x = r \equiv$ right plane,
$y = b \equiv$ bottom plane,
$y = t \equiv$ top plane,
$z = n \equiv$ near plane,
$z = f \equiv$ far plane

Kosakata tersebut mengasumsikan seorang penonton yang melihat sepanjang sumbu z minus dengan kepala mengarah ke arah y.[2] Ini menyiratkan bahwa n>f, yang mungkin tidak intuitif, tetapi jika Anda mengasumsikan seluruh volume tampilan ortografis memiliki nilai z negatif maka z = n bidang "dekat" lebih dekat ke pengamat jika dan hanya jika n>f ; di sini adalah bilangan yang lebih kecil dari n, yaitu bilangan negatif dari nilai zat terlarut yang lebih besar dari n.



Gambar 7.4 Volume tampilan ortografi.



Gambar 7.5 Volume tampilan ortografis berada di sepanjang sumbu z negatif, jadi f adalah angka yang lebih negatif daripada n, sehingga n > f.

[2] Sebagian besar programmer merasa intuitif untuk memiliki sumbu x yang menunjuk ke kanan dan sumbu y yang mengarah ke atas. Dalam sistem koordinat tangan kanan, ini menyiratkan bahwa kita melihat ke arah z. Beberapa sistem menggunakan sistem koordinat tangan kiri untuk melihat sehingga arah pandangan sepanjang +z. Mana yang terbaik adalah masalah selera, dan teks ini mengasumsikan sistem koordinat tangan kanan. Sebuah referensi yang mendukung sistem tangan kiri diberikan dalam catatan di akhir bab ini.

Konsep ini ditunjukkan pada Gambar 7.5. Transformasi dari volume tampilan ortografis ke volume tampilan kanonik adalah transformasi windows lainnya, jadi kita dapat dengan mudah mengganti volume tampilan ortografis dan kanonik yang terikat ke dalam Persamaan (6.7) untuk mendapatkan matriks untuk transformasi ini:

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriks ini sangat dekat dengan matriks yang digunakan secara tradisional di OpenGL, kecuali bahwa n, f, dan zkanonik semuanya memiliki tanda yang berlawanan.

Untuk menggambar segmen garis 3D dalam volume tampilan ortografis, kami memproyeksikannya ke koordinat x-andy layar dan mengabaikan koordinat z. Kami melakukan ini dengan menggabungkan Persamaan(7.2)dan (7.3). Perhatikan bahwa dalam sebuah program kita mengalikan matriks bersama-sama untuk membentuk satu matriks dan kemudian memanipulasi titik-titik sebagai berikut:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = (\mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Koordinat z sekarang akan berada di [−1,1]. Kami tidak mengambil keuntungan dari ini sekarang, tetapi akan berguna ketika kami memeriksa algoritma buffer-z. Kode untuk menggambar banyak garis 3D dengan endpointsai dan bi menjadi sederhana dan efisien:

construct **M**vp
construct **M**orth
M = **M**vp**M**orth
for each line segment (**a***i*, **b***i*) **do**
p = **M**a*i*
q = **M**b*i*
drawline(*xp, yp, xq, yq*)

Transformasi Kamera

Kami ingin dapat mengubah sudut pandang dalam 3D dan melihat ke segala arah. Ada banyak konvensi untuk menentukan posisi dan orientasi pemirsa. Kami akan menggunakan yang berikut ini (lihat Gambar 7.6):

- posisi mata e,
- arah tatapan g,
- vektor tampilan t.

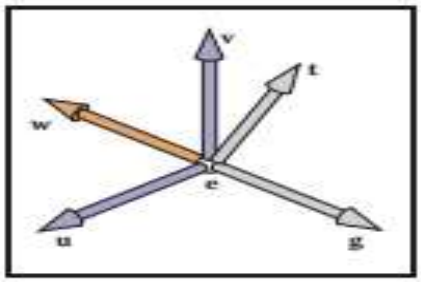
Posisi mata adalah lokasi yang "dilihat dari" mata. Jika Anda menganggap grafis sebagai proses fotografi, itu adalah pusat lensa. Arah pandangan adalah vektor apa pun ke arah yang dilihat oleh pemirsa. Vektor view-up adalah vektor apa pun di pesawat yang membagi kepala pemirsa ke kanan dan kiri dan menunjuk "ke langit" untuk seseorang yang berdiri di tanah. Vektor-vektor ini memberi kita informasi yang cukup untuk membuat sistem koordinat dengan asal e dan basis uvw, menggunakan konstruksi Bagian 2.4.7:

$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|},$$
$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

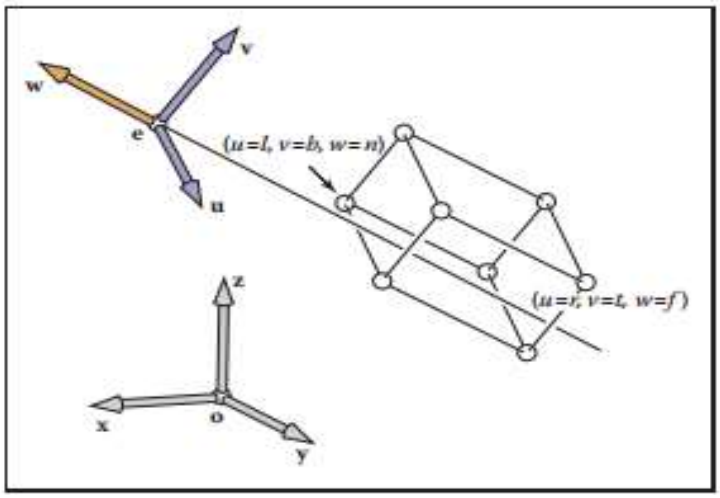
Tugas kita akan selesai jika semua titik yang ingin kita ubah disimpan dalam koordinat dengan vektor asal dan basis su, v, dan w. Tetapi seperti yang ditunjukkan pada Gambar 7.7, koordinat model disimpan dalam bentuk sumbu kanonik (atau dunia) asal dan sumbu x-,y-,danz. Untuk menggunakan mesin yang telah kita kembangkan, kita hanya perlu mengubah koordinat titik akhir segmen garis yang ingin kita gambar dari koordinat xyz menjadi koordinat uvw. Transformasi semacam ini telah dibahas di Bagian 6.5, dan matriks yang memberlakukan transformasi ini adalah matriks kanonik-ke-basismater dari bingkai koordinat kamera:

$$\mathbf{M}_{\text{cam}} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Sebagai alternatif, kita dapat memikirkan transformasi yang sama ini sebagai pertama-tama memindahkan e ke titik asal, kemudian menyelaraskan,v,w ke x,y,z.



Gambar 7.6 Pengguna menentukan tampilan sebagai posisi mata e, arah tatapan g, dan vektor ke atas t. Kami membangun basis tangan kanan dengan w menunjuk berlawanan dengan pandangan dan v berada pada bidang yang sama dengan g dan t.



Gambar 7.7 Untuk tampilan sewenang-wenang, kita perlu mengubah titik yang akan disimpan dalam sistem koordinat yang "sesuai". Dalam hal ini memiliki koordinat asal e dan offset dalam bentuk uvw.

Untuk membuat algoritme tampilan hanya sumbu-z kita sebelumnya berfungsi untuk kamera dengan lokasi dan orientasi apa pun, kita hanya perlu menambahkan transformasi kamera ini ke produk transformasi viewport dan proyeksi, sehingga mengubah titik masuk dari dunia ke koordinat kamera sebelum diproyeksikan:

construct **M**vp
construct **M**orth
construct **M**cam
M = **M**vp**M**orth**M**cam
for each line segment (**a***i*, **b***i*) **do**
p = **M**a*i*
q = **M**b*i*
drawline(*xp, yp, xq, yq*)

Sekali lagi, hampir tidak ada kode yang diperlukan setelah infrastruktur matriks siap.]

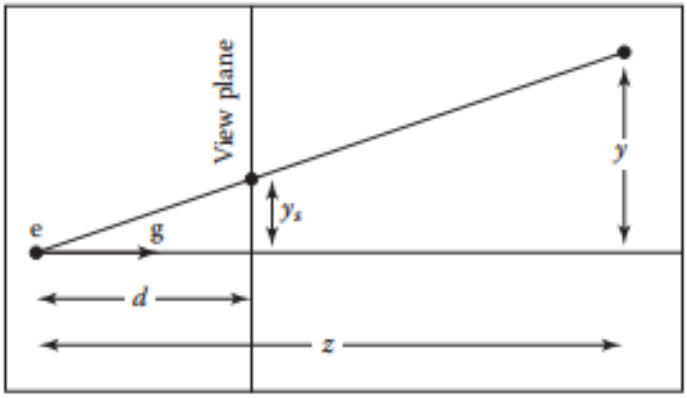
7.2 TRANSFORMASI PROYEKTIF

Kami telah meninggalkan perspektif untuk yang terakhir karena dibutuhkan sedikit kepandaian untuk membuatnya cocok dengan sistem vektor dan transformasi matriks yang telah melayani kami dengan sangat baik hingga sekarang. Untuk melihat apa yang perlu kita lakukan, mari kita lihat apa yang perlu dilakukan transformasi proyeksi perspektif dengan titik di ruang kamera. Ingatlah bahwa sudut pandang diposisikan pada titik asal dan kamera melihat sepanjang sumbu z.

Properti utama dari perspektif adalah bahwa ukuran objek di layar sebanding dengan 1/z untuk mata di titik asal yang melihat ke atas sumbu z negatif. Ini dapat dinyatakan lebih tepat dalam persamaan untuk geometri pada Gambar 7.8:

$$y_s = \frac{d}{z} y,$$

di mana y adalah jarak titik di sepanjang sumbu y, dan ys adalah titik yang harus digambarkan di layar.



Gambar 7.8 Geometri untuk Persamaan (7.5). Mata pemirsa berada di e dan arah pandangan adalah g (sumbu z minus). Bidang pandang berjarak d dari mata. Sebuah titik diproyeksikan ke arah e dan di mana ia memotong bidang pandang adalah di mana ia ditarik.

Kami benar-benar ingin menggunakan mesin matriks yang kami kembangkan untuk proyeksi ortografis untuk menggambar gambar perspektif; kita kemudian dapat mengalikan matriks lain ke dalam matriks komposit kita dan menggunakan algoritme yang sudah kita miliki.

Namun, jenis transformasi ini, di mana salah satu koordinat vektor input muncul di penyebut, tidak dapat dicapai dengan menggunakan transformasi afin.

Kita dapat mengizinkan pembagian dengan generalisasi sederhana dari mekanisme koordinat homogen yang telah kita gunakan untuk transformasi affine. Kita telah setuju untuk merepresentasikan titik (x, y, z) menggunakan vektor homogen $[x\ y\ z\ 1]^T$; koordinat ekstra, w, selalu sama dengan 1, dan ini dipastikan dengan selalu menggunakan $[0\ 0\ 0\ 1]^T$ sebagai baris keempat dari matriks transformasi afin.

Daripada hanya memikirkan 1 sebagai bagian tambahan yang dibautkan untuk memaksa perkalian matriks untuk mengimplementasikan terjemahan, sekarang kita mendefinisikannya sebagai penyebut dari koordinat x-, y-, dan z: vektor homogen $[x\ y\ z\ w\ ]^T$ mewakili titik(x/w, y/w, z/w). Ini membuat perbedaan ketika w = 1, tetapi memungkinkan rentang transformasi yang lebih luas untuk diterapkan jika kita mengizinkan nilai apa pun di baris bawah matriks transformasi, menyebabkan w mengambil nilai selain 1. Secara konkret, transformasi linier memungkinkan kita untuk menghitung ekspresi seperti

$$x = ax + by + cz$$

dan transformasi afine memperluas ini ke

$$x = ax + by + cz + d.$$

Memperlakukan w sebagai penyebut semakin memperluas kemungkinan, memungkinkan kita untuk menghitung fungsi seperti

$$x' = \frac{ax + by + cz + d}{ex + fy + gz + h};$$

ini bisa disebut "fungsi rasional linier" dari x, y, dan z. Tetapi ada kendala tambahan—penyebutnya sama untuk semua koordinat titik yang ditransformasikan:

$$x' = \frac{a_1x + b_1y + c_1z + d_1}{ex + fy + gz + h},$$

$$y' = \frac{a_2x + b_2y + c_2z + d_2}{ex + fy + gz + h},$$

$$z' = \frac{a_3x + b_3y + c_3z + d_3}{ex + fy + gz + h}.$$

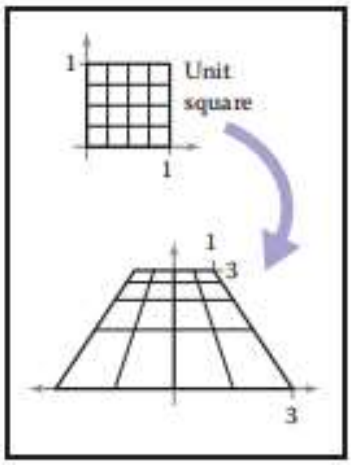
Dinyatakan sebagai transformasi matriks

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

dan

$$(x', y', z') = (\tilde{x}/\tilde{w}, \tilde{y}/\tilde{w}, \tilde{z}/\tilde{w}).$$

Transformasi seperti ini dikenal sebagai transformasi proyektif atau homografi.



Gambar 7.9 Transformasi proyektif memetakan persegi ke segi empat, mempertahankan garis lurus tetapi tidak garis sejajar.

Contoh

Matriks

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 3 & 0 \\ 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

merepresentasikan transformasi proyektif 2D yang mengubah kuadrat satuan([0,1]× [0,1]) menjadi segi empat yang ditunjukkan pada Gambar 7.9. Misalnya, sudut kanan bawah bujur sangkar di (1,0) diwakili oleh vektor homogen $[1\ 0\ 1]^T$ dan transformasi sebagai berikut:

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 3 & 0 \\ 0 & \frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \frac{1}{3} \end{bmatrix}$$

yang mewakili titik (1/1 3,0/1 3), atau(3,0). Perhatikan bahwa jika kita menggunakan matriks

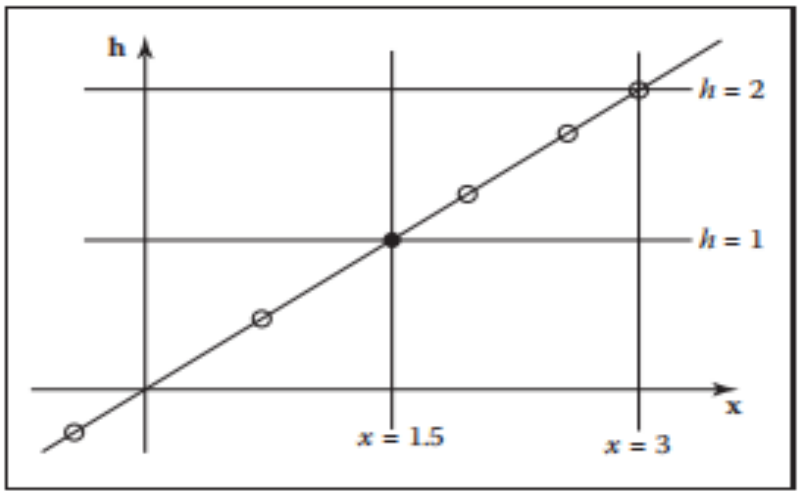
$$3M = \begin{bmatrix} 6 & 0 & -3 \\ 0 & 9 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

sebaliknya, hasilnya adalah $[3\ 0\ 1]^T$, yang juga mewakili (3,0). Faktanya, setiap skalar kelipatan cM adalah ekuivalen: pembilang dan penyebutnya sama-sama diskalakan dengan c, yang tidak mengubah hasilnya.

Ada cara yang lebih elegan untuk mengekspresikan ide yang sama, yang menghindari memperlakukan koordinat w secara khusus. Dalam pandangan ini, transformasi proyektif 3D hanyalah transformasi linier 4D, dengan ketentuan tambahan bahwa semua kelipatan skalar dari suatu vektor mengacu pada titik yang sama:

$$x \sim \alpha x \text{ untuk semua } \alpha \neq 0.$$

Simbol ~ dibaca “setara dengan” dan berarti bahwa dua vektor homogen menggambarkan titik yang sama dalam ruang.

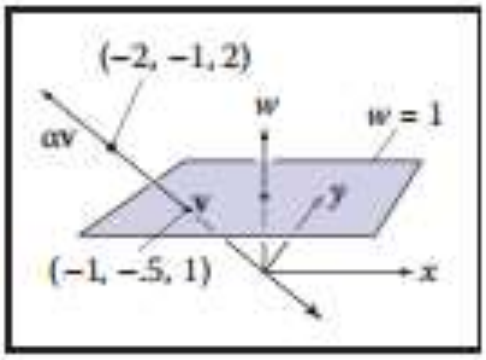


Gambar 7.10 Titik x = 1,5 diwakili oleh sembarang titik pada garis x = 1,5 jam, seperti titik-titik pada lingkaran berongga. Namun, sebelum kita menginterpretasikan x sebagai koordinat Cartesian konvensional, pertama-tama kita bagi dengan h untuk mendapatkan (x, h) = (1.5, 1) seperti yang ditunjukkan oleh titik hitam.

Contoh

Dalam koordinat homogen 1D, di mana kita menggunakan 2-vektor untuk mewakili titik pada garis nyata, kita dapat mewakili titik (1, 5) menggunakan vektor homogen $[1, 5\ 1]^T$, atau titik lain pada garis x = 1,5h dalam ruang homogen. (Lihat Gambar 7.10.)

Dalam koordinat homogen 2D,dimana kita menggunakan3-vektor yang menyimpan titik-titik pada bidang, kita dapat merepresentasikan titik (−1, −0.5) menggunakan vektor homogen $[-2; -1; 2]^T$, atau titik lain pada garis x = $[-1\ 0.5\ 1]^T$. Setiap vektor homogen pada garis dapat dipetakan ke perpotongan garis dengan bidang w =1 untuk mendapatkan koordinat kartesiusnya. (Lihat Gambar 7.11.)



Gambar 7.11 Suatu titik dalam koordinat homogen adalah ekuivalen dengan titik lain pada garis yang melaluinya dan titik asal, dan menormalkan titik tersebut sama dengan memotong garis ini dengan bidang w = 1.

Tidak masalah untuk mentransformasikan vektor-vektor homogen sebanyak yang diperlukan, tanpa mengkhawatirkan nilai koordinat-w—bahkan, tidak masalah jika koordinat-w adalah nol pada beberapa fase peralihan. Hanya ketika kita menginginkan koordinat Cartesian biasa dari suatu titik, kita perlu menormalkan ke titik ekivalen yang memiliki w = 1, yang berarti membagi semua koordinat dengan w. Setelah kita melakukan ini, kita diperbolehkan untuk membaca koordinat (x, y, z) dari tiga komponen pertama dari vektor homogen.

7.3 PROYEKSI PERSPEKTIF

Mekanisme transformasi proyektif membuatnya mudah untuk mengimplementasikan pembagian yang diperlukan untuk mengimplementasikan perspektif. Pada contoh 2D yang ditunjukkan pada Gambar 7.8, kita dapat mengimplementasikan proyeksi perspektif dengan transformasi matriks sebagai berikut:

$$\begin{bmatrix} y_s \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}.$$

Ini mentransformasikan vektor homogen 2D [y; z; 1]^T menjadi vektor homogen 1D [dy z]^T, yang merepresentasikan titik 1D (dy/z) (karena ekivalen dengan vektor homogen 1D [dy/z 1]^T Ini cocok dengan Persamaan (7.5).

Untuk matriks proyeksi perspektif "resmi" dalam 3D, kami akan mengadopsi konvensi biasa kami tentang kamera di titik asal menghadap ke arah z, jadi jarak titik (x, y, z) adalah z. Seperti proyeksi ortografi, kami juga mengadopsi gagasan bidang dekat dan jauh yang membatasi jangkauan jarak yang dapat dilihat. Dalam konteks ini, kita akan menggunakan bidang dekat sebagai bidang proyeksi, sehingga jarak bidang gambar adalah n.

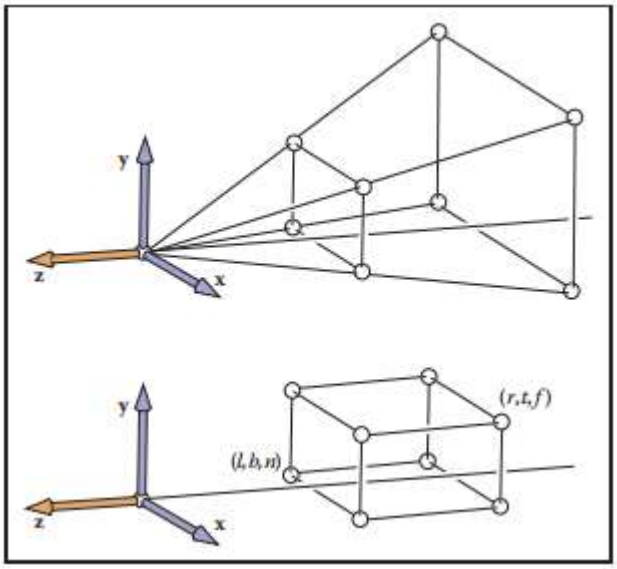
Mapping yang diinginkan adalah ys =(n/z)y, dan juga untuk x. Transformasi ini dapat diimplementasikan dengan matriks perspektif:

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

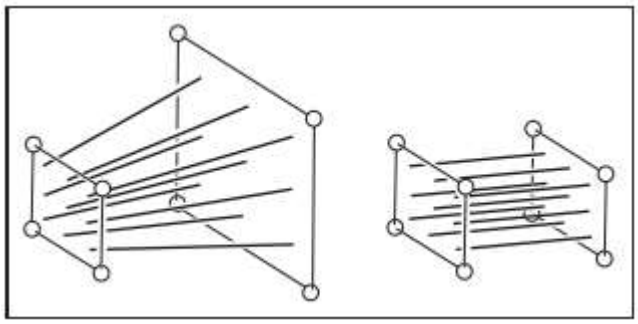
Baris pertama, kedua, dan keempat hanya menerapkan persamaan perspektif. Baris ketiga, seperti pada matriks ortografis dan viewport, dirancang untuk membawa koordinat z "sepanjang perjalanan" sehingga kita dapat menggunakannya nanti untuk menghilangkan permukaan tersembunyi. Namun, dalam proyeksi perspektif, penambahan penyebut tidak konstan mencegah kita untuk benar-benar mempertahankan nilai z—sebenarnya tidak mungkin untuk mencegah z berubah sambil membuat x dan y melakukan apa yang kita perlukan. Alih-alih, kami memilih untuk mempertahankan z tidak berubah untuk poin pada bidang dekat atau jauh.

$$\mathbf{P}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \sim \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

Ada banyak matriks yang dapat berfungsi sebagai matriks perspektif, dan semuanya secara nonlinier mendistorsi koordinat-z. Matriks khusus ini memiliki properti bagus yang ditunjukkan pada Gambar 7.12 dan 7.13; ia meninggalkan titik-titik pada bidang (z = n) seluruhnya sendirian, dan ia meninggalkan titik-titik pada bidang (z = f) sambil "memencet" mereka dalam x dan y dengan jumlah yang sesuai. Pengaruh matriks pada suatu titik (x, y, z) adalah



Gambar 7.12 Proyeksi perspektif membiarkan titik-titik pada bidang z=n tidak berubah dan memetakan z=persegi panjang yang besar di bagian belakang volume perspektif ke z=persegi panjang kecil di bagian belakang volume ortografi.



Gambar 7.13 Proyeksi perspektif memetakan setiap garis melalui titik asal/mata ke garis yang sejajar dengan sumbu z dan tanpa memindahkan titik pada garis di z=n

Seperti yang Anda lihat, x dan y diskalakan dan, yang lebih penting, dibagi dengan z. Karena keduanya n dan z (di dalam volume tampilan) adalah negatif, tidak ada "slip" di x dan y. Meskipun tidak jelas (lihat latihan di akhir bab), transformasi juga mempertahankan urutan relatif nilai z antara z = n dan z = f, memungkinkan kita untuk melakukan urutan kedalaman setelah matriks ini diterapkan. Ini akan menjadi penting nanti ketika kita melakukan eliminasi permukaan tersembunyi.

Terkadang kita ingin mengambil kebalikan dari P, misalnya, untuk mengembalikan koordinat layar plus z ke ruang semula, seperti yang mungkin ingin kita lakukan untuk memilih. Kebalikannya adalah

$$\mathbf{P}^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}.$$

Karena mengalikan vektor homogen dengan skalar tidak mengubah artinya, hal yang sama berlaku untuk matriks yang beroperasi pada vektor homogen. Jadi kita dapat menulis matriks invers dalam bentuk yang lebih indah dengan mengalikannya dengan nf:

$$\mathbf{P}^{-1} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & fn \\ 0 & 0 & -1 & n+f \end{bmatrix}.$$

Matriks ini secara harfiah bukan kebalikan dari matriks P, tetapi transformasi yang digambarkannya adalah kebalikan dari transformasi yang dijelaskan oleh P.

Diambil dalam konteks matriks proyeksi ortografi Morth dalam Persamaan (7.3), matriks perspektif hanya memetakan volume tampilan perspektif (yang berbentuk seperti irisan, atau frustum, piramida) ke volume tampilan ortografis (yang merupakan kotak yang disejajarkan dengan sumbu). Keindahan matriks perspektif adalah setelah kita menerapkannya, kita dapat menggunakan transformasi ortografis untuk mendapatkan volume tampilan kanonik. Jadi, semua mesin ortografis berlaku, dan semua yang telah kita tambahkan adalah satu matriks dan pembagian dengan w. Hal ini juga membesarkan hati bahwa kita tidak "membuang-buang" baris terbawah dari matriks empat kali empat kita! Menggabungkan P dengan M_{orth} menghasilkan matriks proyeksi perspektif,

Mper = **M**orth**P**.

Namun, satu masalah adalah: Bagaimana l,r,b,t ditentukan untuk perspektif? Mereka mengidentifikasi "windows" yang melaluinya kita melihat. Karena matriks perspektif tidak mengubah nilai x dan y pada bidang (z = n), kita dapat menentukan (l, r, b, t) pada bidang tersebut.

Untuk mengintegrasikan matriks perspektif ke dalam infrastruktur ortografis kami, kami cukup mengganti M_{orth} dengan M_{per}, yang memasukkan matriks perspektif P setelah matriks kamera M_{cam} diterapkan tetapi sebelum proyeksi ortografis. Jadi, matriks lengkap untuk melihat perspektif adalah

M = **M**vp**M**orth**PM**cam.

Algoritma yang dihasilkan adalah:

hitung **M**vp
hitung **M**per
hitung **M**cam
M = **M**vp**M**per**M**cam
for setiap segmen garis (**a***i*, **b***i*) **do**
p = **M**a*i*
q = **M**b*i*
drawline(*xp/wp, yp/wp, xq/wq, yq/wq*)

Perhatikan bahwa satu-satunya perubahan selain matriks tambahan adalah pembagian dengan koordinat homogen. Dikalikan, matriks M_{per} terlihat seperti ini:

$$\mathbf{M}_{\text{per}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Matriks atau matriks serupa ini muncul dalam dokumentasi, dan tidak terlalu misterius ketika disadari bahwa mereka biasanya merupakan hasil dari beberapa matriks sederhana.

Contoh

Banyak API seperti OpenGL (Shreiner, Neider, Woo, & Davis, 2004) menggunakan volume tampilan kanonik yang sama seperti yang disajikan di sini. Mereka juga biasanya meminta pengguna menentukan nilai absolut dari n dan f. Matriks proyeksi untuk OpenGL adalah

$$\mathbf{M}_{\text{OpenGL}} = \begin{bmatrix} \frac{2|n|}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2|n|}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{|n|+|f|}{|n|-|f|} & \frac{2|f||n|}{|n|-|f|} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

API lain mengirim n dan f ke 0 dan 1, masing-masing. Blinn (J. Blinn, 1996) merekomendasikan untuk membuat volume tampilan kanonik[0,1]3 untuk efisiensi. Semua keputusan tersebut akan sedikit mengubah matriks proyeksi.'

7.4 BEBERAPA SIFAT TRANSFORMASI PERSPEKTIF

Properti penting dari transformasi perspektif adalah dibutuhkan garis ke garis dan bidang datar. Selain itu, dibutuhkan segmen garis dalam volume tampilan ke segmen garis dalam volume kanonik. Untuk melihat ini, pertimbangkan segmen garis

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}).$$

Ketika ditransformasikan oleh matriks 4 × 4 M, itu adalah titik dengan koordinat homogen yang mungkin bervariasi:

$$\mathbf{Mq} + t(\mathbf{MQ} - \mathbf{Mq}) \equiv \mathbf{r} + t(\mathbf{R} - \mathbf{r}).$$

Segmentasi garis 3D yang homogen adalah

$$\frac{r + t(R - r)}{w_r + t(WR - w_r)}$$

Jika Persamaan (7.6) dapat ditulis ulang dalam bentuk

$$\frac{r}{w_r} + f(t\left(\frac{R}{WR} - \frac{r}{w_r}\right)$$

maka semua titik homogen terletak pada garis 3D. Manipulasi brute force dari Persamaan (7.6) menghasilkan bentuk seperti itu dengan

$$f(t) = \frac{w_R t}{w_r + t(w_R - w_r)}$$

Ternyata juga bahwa segmen garis melakukan mapping ke segmen garis yang mempertahankan urutan titik (Latihan 8), yaitu, mereka tidak menyusun ulang atau "sobek." Sebuah produk sampingan dari transformasi mengambil segmen garis ke segmen garis adalah bahwa ia mengambil tepi dan simpul dari segitiga ke tepi dan simpul dari segitiga lain. Jadi, dibutuhkan segitiga ke segitiga dan pesawat ke pesawat.

7.5 FOV (FIELD-OF-VIEW)

Meskipun kita dapat menentukan windows apa pun menggunakan nilai (l, r, b, t) dan n, terkadang kita ingin memiliki sistem yang lebih sederhana di mana kita melihat melalui bagian tengah windows. Ini menyiratkan kendala bahwa

$$l = -r,$$
$$b = -t.$$

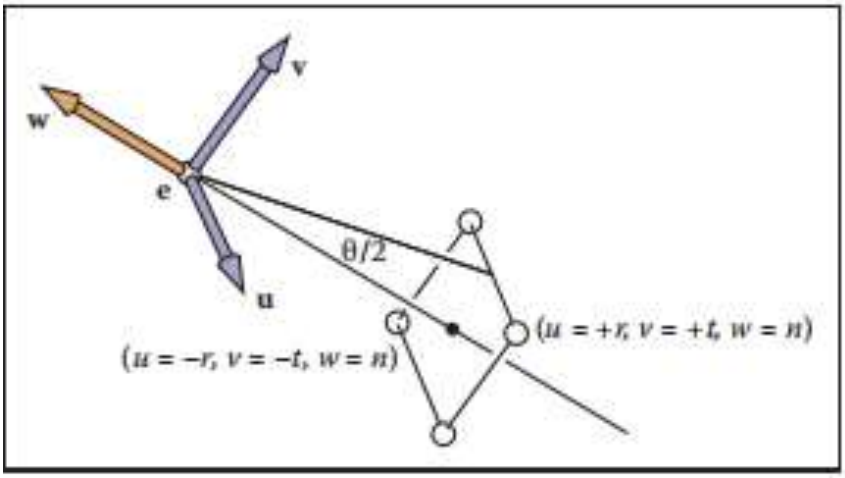
Jika kita juga menambahkan batasan bahwa pikselnya persegi, yaitu, tidak ada distorsi bentuk pada gambar, maka rasio r terhadap t harus sama dengan rasio jumlah piksel horizontal dengan jumlah piksel vertikal:

$$\frac{n_x}{n_y} = \frac{r}{t}$$

Setelah nx dan ny ditentukan, ini hanya menyisakan satu derajat kebebasan. Itu sering diatur menggunakan bidang pandang yang ditunjukkan sebagai pada Gambar 7.14. Ini kadang-kadang disebut bidang pandang vertikal untuk membedakannya dari sudut antara sisi kiri dan kanan atau dari sudut antara sudut diagonal. Dari gambar tersebut kita dapat melihat bahwa

$$tan\frac{\theta}{2} = \frac{t}{|n|}$$

Jika n dan ditentukan, maka kita dapat menurunkan t dan menggunakan kode untuk sistem tampilan yang lebih umum. Dalam beberapa sistem, nilai dari nishard-coded ke beberapa nilai yang masuk akal, dan dengan demikian kita memiliki satu derajat kebebasan yang lebih sedikit.



Gambar 7.14 Bidang pandang adalah sudut dari bagian bawah layar ke bagian atas layar yang diukur dari mata.

Pertanyaan yang Sering Diajukan

- Apakah proyeksi ortografi pernah berguna dalam praktik?

Ini berguna dalam aplikasi di mana penilaian panjang relatif penting. Ini juga dapat menghasilkan penyederhanaan di mana perspektif akan terlalu mahal seperti yang terjadi pada beberapa aplikasi visualisasi medis.

- Bola terselubung yang saya gambar dalam perspektif terlihat seperti oval. Apakah ini bug?

Tidak. Ini adalah perilaku yang benar. Jika Anda menempatkan mata Anda pada posisi relatif yang sama ke layar seperti yang dimiliki penampil virtual sehubungan dengan area pandang, maka oval ini akan terlihat seperti lingkaran karena mereka sendiri dilihat dari sudut.

- Apakah matriks perspektif mengambil nilai z negatif ke nilai z positif dengan urutan terbalik? Bukankah itu menyebabkan masalah?

Ya. Persamaan untuk transformz adalah

$$z' = n + f - \frac{fn}{z}$$

Jadi z =+∈ ditransformasikan menjadi z'= -∞ dan z = -∞ ditransformasikan menjadi z = ∞. Jadi setiap segmen garis yang merentang z = 0 akan "terkoyak" meskipun semua titik akan diproyeksikan ke lokasi layar yang sesuai. Robekan ini tidak relevan jika semua objek berada

dalam volume tampilan. Ini biasanya dijamin dengan kliping ke volume tampilan. Namun, kliping itu sendiri dibuat lebih rumit oleh fenomena robek seperti yang dibahas dalam Bab 8.

- Matriks perspektif mengubah nilai koordinat homogen. Bukankah itu membuat perpindahan dan transformasi skala tidak lagi berfungsi dengan baik?

Menerapkan terjemahan ke titik homogen yang kita miliki

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} hx \\ hy \\ hz \\ h \end{bmatrix} = \begin{bmatrix} hx + ht_x \\ hy + ht_y \\ hz + ht_z \\ h \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}.$$

Efek serupa berlaku untuk transformasi lainnya (lihat Latihan 5).

Latihan

1. Bangun matriks viewport yang diperlukan untuk sistem di mana koordinat piksel menghitung mundur dari atas gambar, bukan naik dari bawah.
2. Kalikan viewport dan matriks proyeksi ortografis, dan tunjukkan bahwa hasilnya juga dapat diperoleh dengan satu aplikasi Persamaan (6.7).
3. Turunkan baris ketiga Persamaan (7.3) dari kendala yang dipertahankan untuk titik-titik pada bidang dekat dan jauh.
4. Tunjukkan secara aljabar bahwa matriks perspektif mempertahankan urutan nilai z dalam volume tampilan.
5. Untuk matriks 4×4 yang tiga baris atasnya merupakan bilangan biner dan baris bawahnya (0, 0, 0, 1), tunjukkan bahwa titik-titik(x, y, z, 1) dan (hx, hy, hz, h) berubah menjadi titik yang sama setelah homogenisasi.
6. Pastikan bentuk $M^{-1}{}_{p}$ yang diberikan dalam teks sudah benar. 7.
7. Pastikan bahwa proyeksi perspektif penuh ke matriks kanonik M membutuhkan (r, t, n) hingga (1, 1, 1).
8. Tuliskan matriks perspektif untuk n = 1 , f = 2.
9. Untuk titik p = (x, y, z, 1), berapakah hasil homogen dan tidak homogen untuk titik tersebut yang ditransformasikan oleh matriks perspektif pada Latihan 6?
10. Untuk posisi mata e = (0 ,1,0), vektor pandangan g = (0 ,−1,0), dan vektor pandangan ke atas t = (1 ,1,0), berapakah *uvw* ortonormal yang dihasilkan dasar yang digunakan untuk rotasi koordinat?
11. Tunjukkan, bahwa untuk transformasi perspektif, segmen garis yang dimulai dalam tampilan volumedomaptolinesegmen dalam volume kanonik setelah homogenisasi. Selanjutnya, tunjukkan bahwa pengurutan relatif titik-titik pada kedua ruas adalah sama. Petunjuk: Tunjukkan bahwa f(t) pada Persamaan (7.8) memiliki sifat-sifat f(0) = 0, f(1) = 1, turunan dari f positif untuk semua t [0,1], dan koordinat homogen tidak tanda tidak berubah.

BAB 8
SALURAN GRAFIS

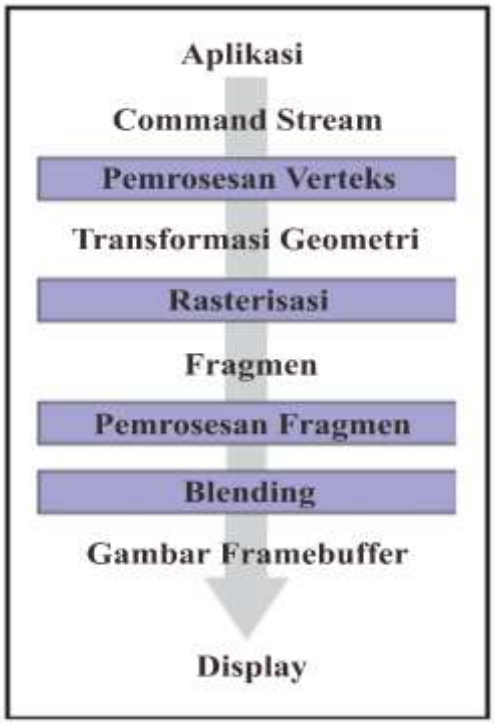
Beberapa bab sebelumnya telah menetapkan perancah matematika. Kita perlu melihat pendekatan utama kedua untuk rendering: menggambar objek satu per satu ke layar, atau rendering urutan objek. Tidak seperti dalam ray tracing, di mana kita mempertimbangkan setiap piksel secara bergantian dan menemukan objek yang memengaruhi warnanya, sekarang kita akan mempertimbangkan untuk mencapai objek geometris secara bergantian dan menemukan piksel yang dapat berpengaruh. Proses menemukan semua piksel dalam gambar yang ditempati oleh primitif geometris disebut rasterisasi, jadi rendering urutan objek juga bisa disebut rendering dengan rasterisasi. Urutan operasi yang diperlukan, dimulai dengan objek dan diakhiri dengan memperbarui piksel pada gambar, dikenal sebagai jalur grafis.

Setiap sistem grafis memiliki satu atau lebih jenis "objek primitif" yang dapat ditangani secara langsung, dan objek yang lebih kompleks diubah menjadi "primitif" ini. Segitiga adalah primitif yang paling sering digunakan. Sistem berbasis rasterisasi juga disebut perender scanline.

Rendering urutan objek telah menikmati kesuksesan besar karena efisiensinya. Untuk scene besar, pengelolaan pola akses data sangat penting untuk kinerja, dan membuat satu lintasan di atas scene mengunjungi setiap bit geometri sekali memiliki keuntungan yang signifikan dibandingkan pencarian berulang kali scene untuk mengambil objek yang diperlukan untuk menaungi setiap piksel.

Judul bab ini menunjukkan bahwa hanya ada satu cara untuk melakukan objectorderrendering. Tentu saja ini tidak benar—dua contoh pipeline grafis yang sangat berbeda dengan tujuan yang sangat berbeda adalah pipeline perangkat keras yang digunakan untuk mendukung rendering interaktif melalui API seperti OpenGL dan Direct3D dan pipeline perangkat lunak yang digunakan dalam produksi film, mendukung API seperti RenderMan. Pipeline perangkat keras harus berjalan cukup cepat untuk bereaksi secara real time untuk game, visualisasi, dan antarmuka pengguna. Pipeline produksi harus membuat animasi dan efek visual dengan kualitas terbaik dan menskalakan ke scene yang sangat besar, tetapi mungkin membutuhkan lebih banyak waktu untuk melakukannya.

Pekerjaan yang perlu dilakukan dalam rendering urutan objek dapat diatur ke dalam tugas rasterisasi itu sendiri, operasi yang dilakukan pada geometri sebelum rasterisasi, dan operasi yang dilakukan pada piksel setelah rasterisasi. Operasi geometris yang paling umum adalah menerapkan transformasi matriks, seperti yang dibahas dalam dua bab sebelumnya, untuk memetakan titik-titik yang mendefinisikan geometri dari ruang objek ke ruang layar, sehingga input ke rasterizer dinyatakan dalam koordinat piksel, atau ruang layar. Operasi pixelwise yang paling umum adalah penghilangan permukaan tersembunyi yang mengatur agar permukaan lebih dekat ke penampil untuk muncul di depan permukaan yang lebih jauh dari penampil. Banyak operasi lain juga dapat dimasukkan dalam setiap tahap, sehingga mencapai berbagai efek rendering yang berbeda menggunakan proses umum yang sama.



Gambar 8.1 Tahapan pipeline grafis.

Untuk tujuan bab ini, kita akan membahas grafis pipa dalam empat tahap (Gambar 8.1). Objek geometris dimasukkan ke dalam saluran pipa dari aplikasi interaktif atau dari file deskripsi scene, dan objek tersebut selalu dideskripsikan oleh himpunan simpul. Vertikal dioperasikan pada tahap pemrosesan simpul, kemudian primitif yang menggunakan simpul tersebut dikirim ke tahap rasterisasi. Rasterizer memecah setiap primitif menjadi sejumlah fragmen, satu untuk setiap piksel yang dicakup oleh primitif. Fragmen diproses dalam tahap pemrosesan fragmen, dan kemudian berbagai fragmen yang sesuai dengan setiap piksel digabungkan dalam tahap pencampuran fragmen. Kita akan mulai dengan membahas rasterisasi, kemudian mengilustrasikan tujuan tahap geometris dan piksel dengan serangkaian contoh.

8.1 RASTERISASI

Rasterisasi adalah operasi sentral dalam grafis urutan objek, dan rasterizer adalah pusat untuk setiap saluran grafis. Untuk setiap primitif yang datang, rasterizer memiliki dua tugas: menghitung piksel yang dicakup oleh primitif dan menginterpolasi nilai, yang disebut atribut, di seluruh primitif—tujuan atribut ini akan dijelaskan dengan contoh selanjutnya. Output dari rasterizer adalah satu set fragmen, satu untuk setiap piksel yang dicakup oleh primitif. Setiap fragmen "hidup" pada piksel tertentu dan membawa kumpulan nilai atributnya sendiri.

Dalam bab ini, kami akan menyajikan rasterisasi dengan tujuan menggunakannya untuk membuat scene tiga dimensi. Metode esterisasi yang sama juga digunakan untuk menggambar garis dan bentuk dalam 2D—walaupun semakin umum menggunakan sistem grafis 3D "di bawah penutup" untuk melakukan semua gambar 2D.

Menggambar garis

Sebagian besar paket grafis berisi perintah menggambar garis yang mengambil dua titik akhir dalam koordinat layar (lihat Gambar 3.10) dan menarik garis di antara keduanya. Misalnya, panggilan untuk titik akhir(1,1)dan (3,2) akan mengaktifkan piksel (1,1) dan (3,2) dan

mengisi satu piksel di antara keduanya. Untuk titik akhir koordinat layar umum (x0,y0) dan(x1,y1), rutin harus menggambar beberapa kumpulan piksel yang "masuk akal" yang mendekati garis di antara mereka. Menggambar garis tersebut didasarkan pada persamaan garis, dan kami memiliki dua jenis persamaan untuk dipilih: implisit dan parametrik. Bagian ini menjelaskan pendekatan menggunakan garis implisit. Meskipun kami sering menggunakan titik akhir bernilai integer sebagai contoh, penting untuk mendukung titik akhir arbitrer dengan benar

Menggambar Garis Menggunakan Persamaan Garis Implisit Cara yang paling umum untuk menggambar garis menggunakan persamaan implisit adalah algoritma titik tengah (Pitteway (1967);van Aken dan Novak(1985)). Algoritma titik tengah akhirnya menggambar garis yang sama dengan algoritma Bresenham (Bresenham, 1965) tetapi agak lebih mudah. Hal pertama yang harus dilakukan adalah menemukan persamaan implisit untuk garis seperti yang dibahas dalam Bagian 2.5.2:

Persamaan 8.1

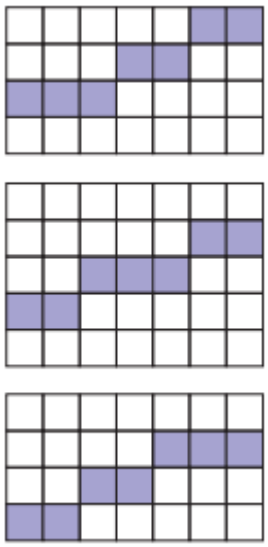
$$f(x, y) \equiv (y0 - y1)x + (x1 - x0)y + x0y1 - x1y0 = 0.$$

Kita asumsikan bahwa $x_0 \geq x_1$. Jika itu tidak benar, kami menukar poin sehingga itu benar. Kemiringan garis m diberikan oleh

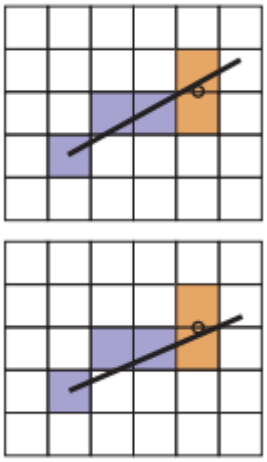
$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Diskusi berikut mengasumsikan m ∈ (0, 1]. Diskusi analog dapat diturunkan untuk m ∈ (−∞, −1], m ∈ (−1,0], dan m ∈ (1, ∞).Keempat kasus mencakup semua kemungkinan.

Untuk kasus m ∈ (0, 1], ada lebih banyak "lari" daripada "naik", yaitu, garis bergerak lebih cepat di x daripada di y. Jika kita memiliki API di mana sumbunya mengarah ke bawah, kita mungkin memiliki kekhawatiran tentang apakah ini membuat proses lebih sulit, tetapi, pada kenyataannya, kita dapat mengabaikan detail itu. Kita dapat mengabaikan gagasan geometris "atas" dan "bawah", karena aljabarnya persis sama untuk dua kasus. Pembaca yang cermat dapat memastikan bahwa ada algoritma yang bekerja untuk sumbu y ke bawah. Asumsi kunci dari algoritma titik tengah adalah bahwa kita menggambar yang paling tipis garis mungkin yang tidak memiliki celah.Hubungan diagonal antara dua piksel tidak dianggap sebagai celah.



Gambar 8.2 Tiga garis "masuk akal" yang menghasilkan tujuh piksel secara horizontal dan tiga piksel secara vertikal.



Gambar 8.3 Atas: garis berada di atas titik tengah sehingga piksel teratas digambar. Bawah: garis berada di bawah titik tengah sehingga piksel bawah digambar.

Saat garis bergerak dari titik akhir kiri ke kanan, hanya ada dua kemungkinan: menggambar piksel dengan ketinggian yang sama dengan piksel yang ditarik ke kiri, atau menggambar piksel satu lebih tinggi. Akan selalu ada tepat satu piksel di setiap kolom piksel di antara titik akhir. Nol akan menyiratkan celah, dan dua akan menjadi alkali gigi. Mungkin ada dua piksel dalam baris yang sama untuk kasus yang sedang kita pertimbangkan; garisnya lebih horizontal daripada vertikal sehingga kadang-kadang akan lurus, dan kadang-kadang naik. Konsep ini ditunjukkan pada Gambar 8.2, di mana tiga garis "masuk akal" ditampilkan, masing-masing lebih maju ke arah horizontal daripada ke arah vertikal.

Algoritme titik tengah untuk m (0,1] pertama-tama menetapkan piksel paling kiri dan nomor kolom (nilai-x) dari piksel paling kanan dan kemudian mengulang secara horizontal membentuk baris (nilai-y) dari setiap piksel. algoritma adalah:

y = *y*0
for *x* = *x*0 ke *x*1 **do**
draw(*x, y*)
if (beberapa kondisi) **then**
y = *y* + 1

Perhatikan bahwa x dan y adalah bilangan bulat. Dengan kata-kata ini mengatakan, "terus menggambar piksel dari kiri ke kanan dan kadang-kadang bergerak ke atas dalam arah y saat melakukannya." Kuncinya adalah untuk menetapkan cara yang efisien untuk membuat keputusan dalam pernyataan **if**.

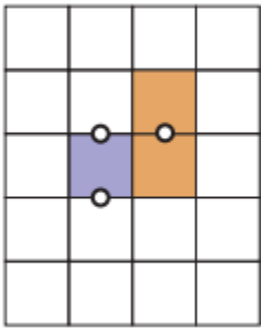
Cara efektif untuk menentukan pilihan adalah dengan melihat titik tengah garis antara dua pusat piksel potensial. Lebih khusus lagi, piksel yang baru saja digambar adalah piksel (x, y) yang pusatnya dalam koordinat layar sebenarnya adalah di (x,y). Kandidat piksel yang akan digambar ke kanan adalah piksel (x + 1, y) dan (x + 1, y + 1). Titik tengah antara pusat dari dua kandidat piksel adalah (x + 1, y + 0.5). Jika garis melewati di bawah titik tengah ini kita menggambar piksel bawah, dan sebaliknya kita menggambar piksel atas (Gambar 8.3).

Untuk memutuskan apakah garis melewati di atas atau di bawah (x + 1, y + 0.5), kita mengevaluasi f(x, y + 0.5) pada Persamaan (8.1). Ingat kembali dari Bagian 2.5.1 bahwa f(x, y)=0 untuk titik (x, y) pada garis, f(x, y) > 0 untuk titik pada salah satu sisi garis, dan f(x, y) < 0 untuk titik-titik di sisi lain garis. Karena f(x, y)=0 dan f(x, y) = 0 keduanya merupakan persamaan yang sangat baik untuk garis, persamaan tersebut tidak langsung jelas apakah f(x, y) menjadi positif menunjukkan bahwa (x, y) berada di atas garis, atau apakah itu di bawah. Namun, kita bisa mengetahuinya; istilah kunci dalam Persamaan (8.1) adalah suku y (x1 x0)y. Perhatikan

bahwa (x1 x0) pasti positif karena $x_1 > x_0$. Ini berarti bahwa dengan bertambahnya y, suku ($x_1 - x_0$)y semakin besar (yaitu, lebih banyak positif atau kurang negatif). Jadi, kasus f(x, +∞) pasti positif, dan pasti di atas garis, menyiratkan poin di atas garis semuanya positif. Cara lain untuk melihatnya adalah bahwa komponen y dari vektor gradien adalah positif. Jadi di atas garis, di mana y dapat meningkat secara sembarang, f(x, y) harus positif. Ini berarti kita dapat membuat kode kita lebih spesifik dengan mengisi pernyataan if:

if $f(x + 1, y + 0.5) < 0$ **then**
$y = y + 1$

Kode di atas akan bekerja dengan baik untuk garis kemiringan yang sesuai (yaitu, antara nol dan satu). Pembaca dapat mengerjakan tiga kasus lain yang hanya berbeda dalam detail-detail kecil.



Gambar 8.4 Saat menggunakan titik keputusan yang ditunjukkan di antara dua piksel oranye, kami hanya menggambar piksel biru, jadi kami mengevaluasi f di salah satu dari dua titik kiri yang ditampilkan

Jika efisiensi yang lebih besar diinginkan, menggunakan metode tambahan dapat membantu. Metode inkremental mencoba membuat loop lebih efisien dengan menggunakan kembali komputasi dari langkah sebelumnya. Dalam algoritma titik tengah seperti yang disajikan, perhitungan utama adalah evaluasi f(x + 1, y + 0.5). Perhatikan bahwa di dalam loop, setelah iterasi pertama, kita telah mengevaluasi f(x − 1, y + 0.5) atau f(x − 1, y − 0.5) (Gambar 8.4). Perhatikan juga hubungan ini:

$$f(x + 1, y) = f(x, y)+(y0 -- y1)$$
$$f(x + 1, y + 1) = f(x, y)+(y0 - - y1)+(x1- x0).$$

Ini memungkinkan kita untuk menulis versi inkremental dari kode:

```
y = y0
d = f(x0 + 1, y0 + 0.5)
for x = x0 to x1 do
        draw(x, y)
        if d < 0 then
        y = y + 1
        d = d + (x1 − x0)+(y0 − y1)
else
        d = d + (y0 − y1)
```

Kode ini harus berjalan lebih cepat karena memiliki sedikit biaya penyiapan tambahan dibandingkan dengan versi non-incremental (yang tidak selalu benar untuk algoritma inkremental), tetapi mungkin mengakumulasi lebih banyak kesalahan numerik karena evaluasi f(x, y + 0.5) dapat terdiri dari banyak tambahan untuk antrean panjang. Namun, mengingat bahwa garis jarang lebih panjang dari beberapa ribu piksel, kesalahan seperti itu tidak mungkin menjadi kritis. Biaya setup yang sedikit lebih lama, tetapi eksekusi loop yang lebih

cepat, dapat dicapai dengan menyimpan $(x_1 - x_0) + (y_0 - y_1)$ dan $(y_0 - y_1)$ sebagai variabel. Kami mungkin berharap goodcompiler akan melakukannya untuk kami, tetapi jika kodenya penting, sebaiknya periksa hasil kompilasi untuk memastikannya.

Rasterisasi Segitiga

Kita sering ingin menggambar segitiga 2D dengan pon 2D $p_0 = (x_0, y_0)$, $p_1 = (x_1, y_1)$, dan $p_2 = (x_2, y_2)$ dalam koordinat layar. Ini mirip dengan masalah menggambar garis, tetapi memiliki beberapa kehalusan tersendiri. Seperti menggambar garis, kita mungkin ingin menginterpolasi warna atau properti lain dari nilai pada simpul. Ini mudah jika kita memiliki koordinat barycentric (Bagian 2.7). Misalnya, jika titik-titik memiliki warna c_0, c_1, dan c_2, warna pada suatu titik dalam segitiga dengan koordinat barycentric (α, β, γ) adalah

$$\mathbf{c} = \alpha\mathbf{c}0 + \beta\mathbf{c}1 + \gamma\mathbf{c}2.$$

Jenis interpolasi warna ini dikenal dalam grafis sebagai interpolasi Gouraud setelah penemunya (Gouraud, 1971).

Kehalusan lain dari rasterisasi segitiga adalah bahwa kita biasanya rasterisasi segitiga yang berbagi simpul dan tepi. Ini berarti kita ingin melakukan rasterisasi pada segitiga yang berdekatan sehingga tidak ada lubang. Kita bisa melakukan ini dengan menggunakan algoritma titik tengah untuk menggambar garis luar setiap segitiga dan kemudian mengisi piksel interior. Ini berarti segitiga yang berdekatan menggambar piksel yang sama di sepanjang setiap tepi. Jika segitiga yang berdekatan memiliki warna yang berbeda, gambar akan tergantung pada urutan di mana dua segitiga digambar. Cara paling umum untuk meraster segitiga yang menghindari masalah urutan dan menghilangkan lubang adalah dengan menggunakan konvensi bahwa piksel digambar jika dan hanya jika pusatnya berada di dalam segitiga, yaitu, koordinat barysentris daripusat piksel berada di dalam interval (0,1). Ini menimbulkan masalah tentang apa yang harus dilakukan jika pusatnya tepat di tepi segitiga. Ada beberapa cara untuk menangani ini seperti yang akan dibahas nanti di bagian ini. Pengamatan utama adalah bahwa koordinat barycentric memungkinkan kita untuk memutuskan apakah akan menggambar piksel dan warna apa yang seharusnya menjadi piksel jika kita menginterpolasi warna dari simpul. Jadi masalah rasterisasi segitiga kita bermuara pada pencarian koordinat barycentric dari pusat piksel secara efisien (Pineda, 1988). Algoritma rasterisasi brute force adalah:

```
for all x do
    for all y do
        compute (α, β, γ) for (x, y)
        if (α [∈ 0, 1] dan β ∈ [0, 1] dan γ ∈ [0, 1]) then
            c = αc0 + βc1 + γc2
            drawpixel (x, y) dengan warna c
```

Algoritma lainnya membatasi loop luar menjadi kumpulan piksel kandidat yang lebih kecil dan membuat komputasi barycentric menjadi efisien.

Kita dapat menambahkan efisiensi sederhana dengan menemukan persegi panjang pembatas dari tiga simpul dan hanya mengulang persegi panjang ini untuk menggambar kandidat piksel. Kita dapat menghitung koordinat barycentric menggunakan Persamaan (2.32). Ini menghasilkan algoritma:

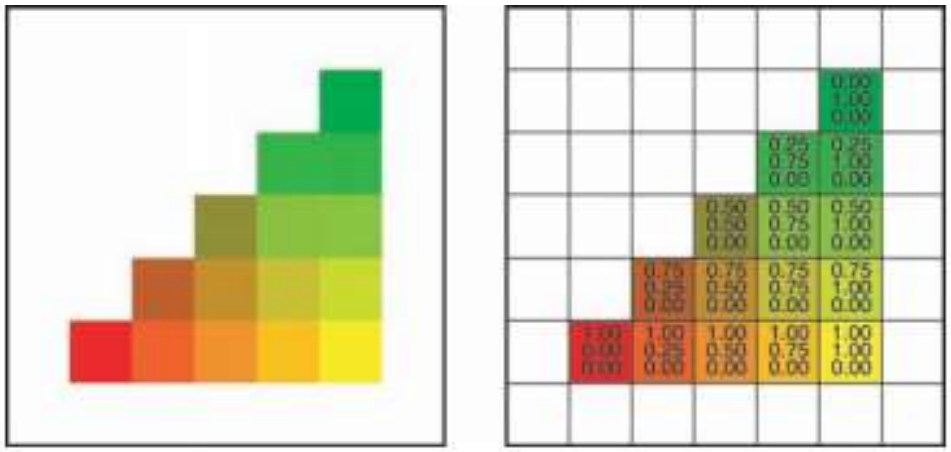
```
xmin = flfloor(xi)
xmax = ceiling(xi)
ymin = flfloor(yi)
ymax = ceiling(yi)
for y = ymin to ymax do
    for x = xmin to xmax do
```

$\alpha = f12(x, y)/f12(x0, y0)$
$\beta = f20(x, y)/f20(x1, y1)$
$\gamma = f01(x, y)/f01(x2, y2)$
if ($\alpha > 0$ dan $\beta > 0$ dan $\gamma > 0$) **then**
c = α**c**0 + β**c**1 + γ**c**2
drawpixel (*x, y*) dengan warna **c**

Di sini f_{ij} adalah garis yang diberikan oleh Persamaan (8.1) dengan simpul yang sesuai:

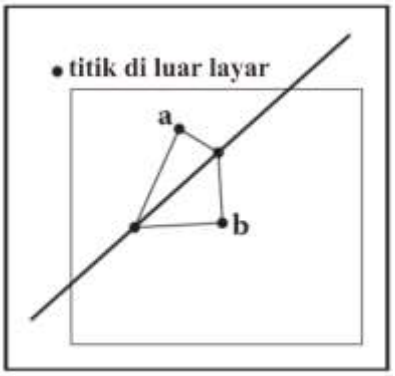
$$f01(x, y)=(y0 - y1)x + (x1 - x0)y + x0y1 - x1y0,$$
$$f12(x, y)=(y1 - y2)x + (x2 - x1)y + x1y2 - x2y1,$$
$$f20(x, y)=(y2 - y0)x + (x0 - x2)y + x2y0 - x0y2.$$

Perhatikan bahwa kita telah menukar uji $\alpha \in (0, 1)$ dengan $\alpha > 0$ dst., karena jika semua α, β, γ positif, maka kita tahu semuanya kurang dari satu karena α+β+γ = 1. Kita juga dapat menghitung hanya dua dari tiga variabel barycentric dan mendapatkan yang ketiga dari hubungan itu, tetapi tidak jelas bahwa ini menghemat perhitungan setelah algoritma dibuat inkremental, yang mungkin seperti pada algoritma menggambar garis; masing-masing perhitungan α, β, dan γ melakukan evaluasi bentuk f(x, y) = Ax + By + C. Dalam loop dalam, hanya x yang berubah, dan itu berubah satu. Perhatikan bahwa f(x + 1, y) = f(x, y) + A. Ini adalah dasar dari algoritma inkremental. Pada loop luar, evaluasi berubah untuk f(x, y) menjadi f(x, y + 1) , sehingga efisiensi yang sama dapat dicapai. Karena α, β dan γ berubah dengan kenaikan konstan dalam loop, begitu juga warna c. Jadi ini bisa dibuat inkremental juga. Misalnya, nilai merah untuk piksel (x + 1, y) berbeda dari nilai merah untuk piksel (x, y) dengan jumlah konstan yang dapat dihitung sebelumnya. Contoh segitiga dengan interpolasi warna ditunjukkan pada Gambar 8.5.



Gambar 8.5 Segitiga berwarna dengan interpolasi barycentric. Perhatikan bahwa perubahan komponen warna adalah linier di setiap baris dan kolom serta di sepanjang setiap tepi. Bahkan konstan sepanjang setiap garis, seperti diagonal, juga.

Menangani Piksel pada Tepi Segitiga Kita masih belum membahas apa yang harus dilakukan untuk piksel yang pusatnya tepat berada di tepi segitiga. Jika sebuah piksel tepat berada di tepi segitiga, maka piksel tersebut juga berada di tepi segitiga yang berdekatan jika ada. Tidak ada cara yang jelas untuk memberikan piksel ke satu segitiga atau yang lain. Keputusan terburuk adalah tidak menggambar piksel karena lubang akan terjadi di antara dua segitiga. Lebih baik, tapi tetap tidak bagus, adalah membuat kedua segitiga menggambar piksel. Jika segitiga transparan, ini akan menghasilkan pewarnaan ganda. Kami benar-benar ingin memberikan piksel ke salah satu segitiga, dan kami ingin proses ini sederhana; segitiga mana yang dipilih tidak masalah selama pilihan itu didefinisikan dengan baik.



Gambar 8.6 Titik di luar layar akan berada di satu sisi tepi segitiga atau sisi lainnya. Tepat satu dari simpul tak-berbagi a dan b akan berada pada sisi yang sama.

Salah satu pendekatan adalah untuk mencatat bahwa setiap titik di luar layar pasti berada tepat di satu sisi tepi bersama dan itulah tepi yang akan kita gambar. Untuk dua segitiga yang tidak tumpang tindih, simpul-simpul yang tidak berada di tepi berada pada sisi yang berlawanan dari tepi satu sama lain. Tepat satu dari simpul ini akan berada di sisi tepi yang sama dengan titik di luar layar (Gambar 8.6). Ini adalah dasar dari tes. Pengujian jika angka p dan q memiliki tanda yang sama dapat diimplementasikan sebagai pengujian pq > 0, yang sangat efisien di sebagian besar lingkungan.

Perhatikan bahwa pengujian ini tidak sempurna karena garis yang melalui tepi mungkin juga melewati titik di luar layar, tetapi setidaknya kami telah sangat mengurangi jumlah kasus yang bermasalah. Titik di luar layar mana yang digunakan adalah sembarang, dan (x, y) = (−1, −1) merupakan pilihan yang baik. Kita perlu menambahkan tanda centang untuk kasus titik yang tepat di tepi. Kami ingin pemeriksaan ini tidak dilakukan untuk kasus-kasus umum, yang merupakan pengujian sepenuhnya di dalam atau di luar. Ini menyarankan:

```
xmin = flfloor(xi)
xmax = ceiling(xi)
ymin = flfloor(yi)
ymax = ceiling(yi)
fα = f12(x0, y0)
fβ = f20(x1, y1)
fγ = f01(x2, y2)
for y = ymin to ymax do
    for x = xmin to xmax do
        α = f12(x, y)/fα
        β = f20(x, y)/fβ
        γ = f01(x, y)/fγ
        if (α ≥ 0 and β ≥ 0 and γ ≥ 0) then
            if (α > 0 or fαf12(−1, −1) > 0) and
            (β > 0 or fβf20(−1, −1) > 0) and
            (γ > 0 or fγf01(−1, −1) > 0) then
            c = αc0 + βc1 + γc2
            drawpixel (x, y) dengan warna c
```

Kita mungkin berharap bahwa kode di atas akan bekerja untuk menghilangkan lubang dan penarikan ganda hanya jika kita menggunakan persamaan garis yang persis sama untuk kedua segitiga. Faktanya, persamaan garis adalah sama hanya jika dua simpul bersama memiliki orde yang sama dalam panggilan undian untuk setiap segitiga. Jika tidak, persamaan

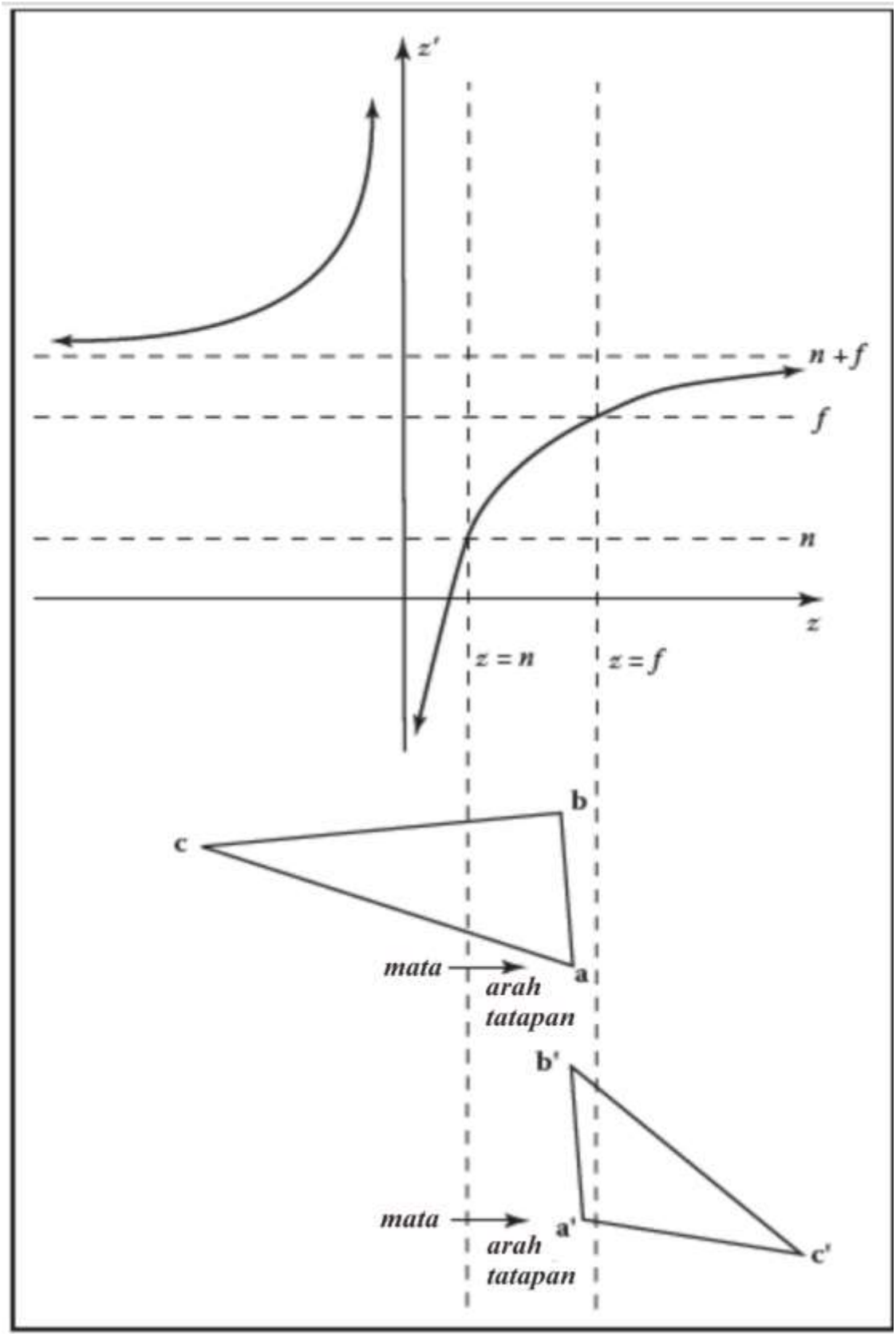
mungkin terbalik. Ini bisa menjadi masalah tergantung pada apakah kompiler mengubah urutan operasi. Jadi jika implementasi yang kuat diperlukan, detail kompiler dan unit aritmatika mungkin perlu diperiksa. Empat baris pertama dalam pseudocode di atas harus dikodekan dengan hati-hati untuk menangani kasus di mana tepi tepat mengenai pusat piksel.

Selain dapat menerima implementasi inkremental, ada beberapa titik keluar awal yang potensial. Misalnya, jika negatif, tidak perlu menghitung β atau γ. Meskipun ini mungkin menghasilkan peningkatan kecepatan, pembuatan profil selalu merupakan ide yang baik; cabang tambahan dapat mengurangi saluran pipa atau konkurensi dan mungkin memperlambat kode. Jadi selalu, ujilah optimalisasi yang tampak menarik jika kode adalah bagian yang kritis.

Detail lain dari kode di atas adalah bahwa pembagian dapat berupa pembagian dengan nol untuk segitiga yang mengalami degenerasi, yaitu jika $f_\gamma = 0$. Entah kondisi kesalahan titik mengambang harus diperhitungkan dengan benar, atau tes lain akan diperlukan

Guntingan

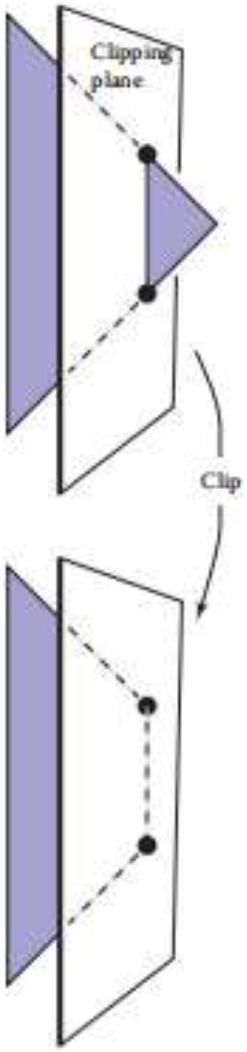
Cukup mengubah primitif menjadi ruang layar dan rasterisasi mereka tidak cukup bekerja dengan sendirinya. Ini karena primitif yang berada di luar volume tampilan—khususnya, primitif yang ada di belakang mata—dapat menjadi raster, yang mengarah ke hasil yang salah. Misalnya, perhatikan segitiga yang ditunjukkan pada Gambar 8.7.



Gambar8.7 Kedalaman z ditransformasikan ke kedalaman z’ dengan transformasi perspektif. Perhatikan bahwa ketika z bergerak dari positif ke negatif, z’ beralih dari negatif ke positif. Dengan demikian simpul di belakang mata dipindahkan di depan mata melampaui z = n + f. Ini akan menyebabkan hasil yang salah, itulah sebabnya segitiga terlebih dahulu dipotong untuk memastikan semua simpul berada di depan mata.

Dua simpul berada dalam volume tampilan, tetapi yang ketiga berada di belakang mata. Transformasi proyeksi memetakan teks vertikal ini pada lokasi yang tidak masuk akal di belakang bidang jauh, dan jika ini dibiarkan terjadi, segitiga akan diraster dengan tidak benar. Untuk itu, rasterisasi harus didahului dengan operasi pemotongan yang menghilangkan bagian-bagian primitif yang dapat memanjang ke belakang mata.

Kliping adalah operasi umum dalam grafis, diperlukan setiap kali satu entitas geometris "memotong" yang lain. Misalnya, jika Anda menjepit segitiga pada bidang x =0 , bidang tersebut memotong segitiga menjadi dua bagian jika tanda-tanda koordinat x dari simpul-simpulnya tidak sama semua. Dalam sebagian besar aplikasi kliping, bagian segitiga di sisi yang "salah" dari bidang dibuang. Operasi ini untuk satu bidang ditunjukkan pada Gambar 8.8.



Gambar 8.8 Sebuah poligon terpotong pada bidang kliping. Bagian "di dalam" pesawat dipertahankan.

Dalam kliping untuk mempersiapkan rasterisasi, sisi yang "salah" adalah sisi di luar volume tampilan. Selalu aman untuk memotong semua geometri di luar volume tampilan—yaitu, memotong keenam sisi volume—tetapi banyak sistem berhasil lolos hanya dengan memotong pada bidang dekat.

Bagian ini membahas implementasi dasar modul kliping. Mereka yang tertarik untuk menerapkan clipper kecepatan industri harus melihat buku oleh Blinn yang disebutkan dalam

catatan di akhir bab ini. Dua pendekatan yang paling umum untuk mengimplementasikan kliping adalah

1. di koordinat dunia menggunakan enam bidang yang mengikat piramida penglihatan terpotong,
2. di ruang transformasi 4D sebelum pembagian homogen

Kemungkinan dapat diterapkan secara efektif (J. Blinn, 1996) dengan menggunakan pendekatan berikut untuk setiap segitiga:

for setiap six plane **do**
if (segitiga seluruhnya di luar bidang) **then**
break (segitiga tidak terlihat)
else if bidang bentang segitiga **then**
clip segitiga
if (segi empat kiri) **then**
break ke kedua segitiga

Kliping Sebelum Transform (Opsi 1)

Opsi 1 memiliki implementasi langsung. Satu-satunya pertanyaan adalah, "Apa persamaan enam bidang?" Karena persamaan ini sama untuk semua segitiga yang dirender dalam gambar tunggal, kita tidak perlu menghitungnya dengan sangat efisien. Untuk alasan ini, kita bisa membalikkan transformasi yang ditunjukkan pada Gambar 5.11 dan menerapkannya ke delapan simpul dari volume tampilan yang diubah:

$$(x, y, z) = (l, b, n)$$
$$(r, b, n)$$
$$(l, t, n)$$
$$(r, t, n)$$
$$(l, b, f)$$
$$(r, b, f)$$
$$(l, t, f)$$
$$(r, t, f).$$

Persamaan bidang dapat disimpulkan dari sini. Atau, kita dapat menggunakan geometri vektor untuk mendapatkan bidang langsung dari parameter tampilan.

Memotong dalam Koordinat Homogen (Opsi 2)

Anehnya, opsi yang biasanya diterapkan adalah kliping dalam koordinat homogen sebelum membagi. Di sini volume tampilan adalah 4D, dan dibatasi oleh volume 3D (hyperplanes). Ini adalah

$$-x + lw = 0,$$
$$x - rw = 0,$$
$$-y + bw = 0,$$
$$y - tw = 0,$$
$$-z + nw = 0,$$
$$z - fw = 0.$$

Bidang-bidang ini cukup sederhana, sehingga efisiensinya lebih baik daripada Opsi 1. Bidang ini masih dapat ditingkatkan dengan mengubah volume tampilan [l, r] × [b, t] × [f, n] menjadi $[0, 1]^3$. Ternyata kliping segitiga tidak lebih rumit daripada di 3D.

Kliping melawan Pesawat

Tidak peduli pilihan mana yang kita pilih, kita harus memotong pesawat. Ingat dari Bagian 2.5.5 bahwa persamaan implisit untuk bidang yang melalui titik q dengan n normal adalah

$$f(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{q}) = 0.$$

Ini sering ditulis : (Persamaan 8.2)

$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + D = 0.$$

Menariknya, persamaan ini tidak hanya menggambarkan bidang 3D, tetapi juga menggambarkan garis pada 2 Dan dan volume analog bidang 4D. Semua entitas ini biasanya disebut bidang dalam dimensi yang sesuai.

Jika kita memiliki segmen garis antara titik a dan b, kita dapat "memotongnya" pada bidang menggunakan teknik untuk memotong tepi segitiga 3D dalam program pohon BSP yang dijelaskan dalam Bagian 12.4.3. Di sini, titik a dan b diuji untuk menentukan apakah mereka berada pada sisi yang berlawanan dari bidang f(p) = 0dengan memeriksa apakah f(a) dan f(b) memiliki tanda yang berbeda. Biasanya f(p) < 0 didefinisikan sebagai “di dalam” bidang, dan f(p) > 0 adalah “di luar” bidang. Jika bidang membelah garis, maka kita dapat menyelesaikan titik potong dengan mensubstitusi persamaan garis parametrik,

$$P = a + t(b - a)$$

ke dalam f(p) = 0 bidang Persamaan(8.2). Ini menghasilkan

$$n \ . \ (a + t(b - a)) + D = 0$$

Memecahkan untuk t memberi

$$t = \frac{n \ . \ a \ + D}{n \ . (a - b)}$$

Kami kemudian dapat menemukan titik persimpangan dan "memperpendek" garis. Untuk memotong segitiga, kita kembali dapat mengikuti Bagian 12.4.3 untuk menghasilkan satu atau dua segitiga.

8.2 OPERASI SEBELUM DAN SETELAH RASTERISASI

Sebelum primitif dapat dirasterisasi, simpul yang mendefinisikannya harus dalam koordinat layar, dan warna atau atribut lain yang seharusnya diinterpolasi melintasi primitif harus diketahui. Mempersiapkan data ini adalah tugas dari tahap vertexprocessing dari pipeline. Pada tahap ini, simpul masuk ditransformasikan oleh pemodelan, tampilan, dan transformasi proyeksi, memetakannya dari koordinat aslinya ke ruang layar (di mana, ingat, posisi diukur dalam piksel). Pada saat yang sama, informasi lain, seperti warna, normal permukaan, atau koordinat tekstur, diubah sesuai kebutuhan; kita akan membahas atribut tambahan ini dalam contoh di bawah ini.

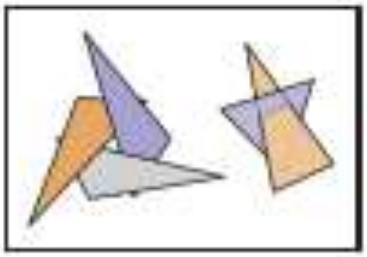
Setelah rasterisasi, pemrosesan lebih lanjut dilakukan untuk menghitung warna dan kedalaman untuk setiap fragmen. Pemrosesan ini dapat menjadi sangat sederhana hanya dengan melewatkan warna yang diinterpolasi dan menggunakan kedalaman yang dihitung oleh rasterizer; atau dapat melibatkan operasi bayangan yang kompleks. Akhirnya, fase pencampuran menggabungkan fragmen yang dihasilkan oleh (mungkin beberapa) primitif yang tumpang tindih setiap piksel untuk menghitung warna akhir. Pendekatan pencampuran yang paling umum adalah memilih warna fragmen dengan kedalaman terkecil (paling dekat dengan mata). Tujuan dari tahapan yang berbeda paling baik diilustrasikan dengan contoh.

Gambar 2D Sederhana

Pipeline paling sederhana yang mungkin tidak melakukan apa pun dalam tahap verteks atau fragmen, dan dalam tahap pencampuran, warna setiap fragmen hanya menimpa nilai dari yang sebelumnya. Aplikasi menyediakan primitif langsung dalam koordinat piksel, dan rasterizer melakukan semua pekerjaan. Pengaturan dasar ini adalah inti dari banyak API sederhana yang lebih tua untuk menggambar antarmuka pengguna, plot, grafis, dan konten 2D lainnya. Bentuk warna solid dapat digambar dengan menentukan warna yang sama untuk semua simpul dari setiap primitif, dan model pipa kami juga mendukung variasi warna yang mulus menggunakan interpolasi.

Pipa 3D Minimal

Untuk menggambar objek dalam 3D, satu-satunya perubahan yang diperlukan untuk jalur gambar 2D adalah transformasi matriks tunggal: tahap pemrosesan titik mengalikan posisi titik masuk dengan produk dari matriks pemodelan, kamera, proyeksi, dan viewport, menghasilkan segitiga ruang layar yang kemudian digambar dengan cara yang sama seperti jika mereka menggambar objek dalam 3D. telah ditentukan secara langsung dalam 2D.



Gambar 8.9 Dua siklus oklusi, yang tidak dapat ditarik dalam urutan back-to-front.

Gambar 8.10 Hasil menggambar dua bola dengan ukuran yang sama menggunakan pipa minimal. Bola yang tampak lebih kecil lebih jauh tetapi ditarik terakhir, sehingga salah menimpa bola yang lebih dekat.

Satu masalah dengan pipeline 3D minimal adalah untuk mendapatkan hubungan oklusi yang benar—untuk mendapatkan objek yang lebih dekat di depan objek yang lebih jauh—primitif harus digambar dalam urutan belakang-ke-depan. Ini dikenal sebagai algoritme pelukis untuk menghilangkan permukaan tersembunyi, dengan analogi melukis latar belakang lukisan terlebih dahulu, lalu melukis latar depan di atasnya. Algoritme pelukis adalah cara yang benar-benar valid untuk menghilangkan permukaan tersembunyi, tetapi memiliki beberapa kelemahan. Itu tidak dapat menangani segitiga yang berpotongan satu sama lain, karena tidak ada urutan yang benar untuk menggambarnya. Demikian pula, beberapa segitiga, bahkan jika mereka tidak berpotongan, masih dapat diatur dalam siklus oklusi, seperti yang ditunjukkan pada Gambar 8.9, kasus lain di mana urutan back-to-front tidak ada. Dan yang paling penting, pengurutan primitif menurut kedalaman lambat, terutama untuk scene besar, dan mengganggu aliran data yang efisien yang membuat rendering urutan objek menjadi sangat cepat. Gambar8.10 menunjukkan hasil dari proses ini ketika objek tidak diurutkan berdasarkan kedalaman.

Menggunakan z-Buffer untuk Permukaan Tersembunyi

Dalam praktiknya, algoritma pelukis jarang digunakan; sebagai gantinya digunakan algoritma penghilangan permukaan tersembunyi yang sederhana dan efektif yang dikenal sebagai algoritma z-buffer. Metodenya sangat sederhana: pada setiap piksel kami melacak jarak ke permukaan terdekat yang telah digambar sejauh ini, dan kami membuang fragmen yang lebih jauh dari jarak tersebut. Jarak terdekat disimpan dengan mengalokasikan nilai

tambahan untuk setiap piksel, selain nilai warna hijau, dan biru, yang dikenal sebagai kedalaman, atau nilai z. Buffer kedalaman, atau buffer-z, adalah nama untuk kisi nilai kedalaman.

Algoritme buffer-z diimplementasikan dalam fase pencampuran fragmen, dengan membandingkan kedalaman setiap fragmen dengan nilai saat ini yang tersimpan di buffer-z. Jika kedalaman fragmen lebih dekat, baik warna maupun nilai kedalamannya akan menimpa nilai yang saat ini ada dalam buffer warna dan kedalaman. Jika kedalaman fragmen lebih jauh, itu dibuang. Untuk memastikan bahwa fragmen pertama akan lulus uji kedalaman, buffer diinisialisasi ke kedalaman maksimum (kedalaman bidang jauh). Terlepas dari urutan permukaan yang digambar, fragmen yang sama akan memenangkan uji kedalaman, dan gambar akan sama.

Algoritma z-buffer membutuhkan setiap fragmen untuk membawa kedalaman. Hal ini dilakukan hanya dengan menginterpolasi koordinat z sebagai atribut verteks, dengan cara yang sama bahwa warna atau atribut lainnya diinterpolasi.

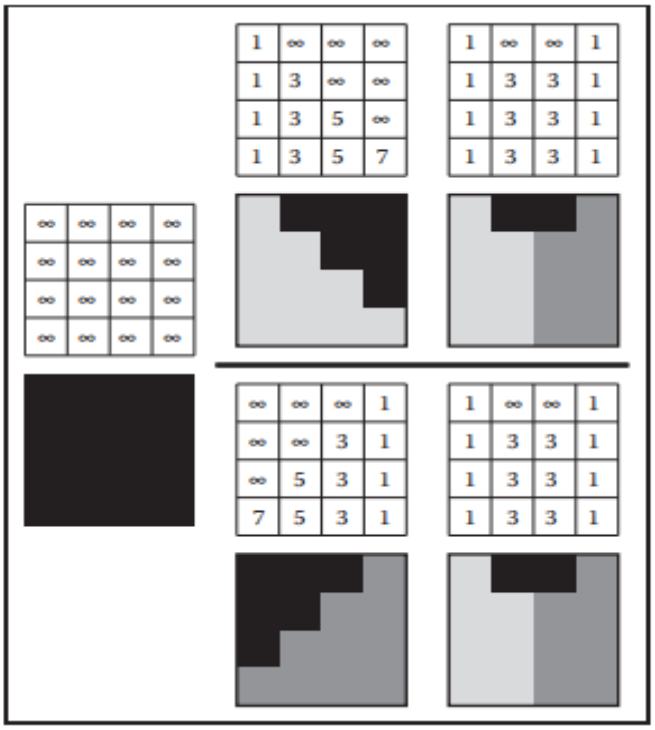
Buffer-z adalah cara yang sederhana dan praktis untuk menangani permukaan tersembunyi dalam urutan objek yang sejauh ini merupakan pendekatan yang dominan. Ini jauh lebih sederhana daripada metode geometris yang memotong permukaan menjadi potongan-potongan yang dapat diurutkan berdasarkan kedalaman, karena menghindari pemecahan masalah yang tidak perlu dipecahkan. Urutan kedalaman hanya perlu ditentukan di lokasi piksel, dan hanya itu yang dilakukan buffer-z. Ini didukung secara universal oleh saluran grafis perangkat keras dan juga merupakan metode yang paling umum digunakan untuk saluran perangkat lunak. Gambar 8.11 menunjukkan hasil contoh.

Gambar 8.11 Hasil menggambar dua bola yang sama menggunakan buffer-z.

Masalah Presisi

Dalam praktiknya, nilai-z yang disimpan dalam buffer adalah bilangan bulat non-negatif. Ini lebih disukai daripada float yang sebenarnya karena memori cepat yang dibutuhkan untuk buffer-z agak mahal dan harus dijaga seminimal mungkin.

Penggunaan bilangan bulat dapat menyebabkan beberapa masalah presisi. Jika kita menggunakan rentang bilangan bulat yang memiliki nilai B{0, 1, ..., B−1}, kita dapat memetakan 0 ke bidang kliping dekat z = n dan B−1 ke bidang kliping jauh z = f. Perhatikan bahwa untuk pembahasan ini, kita asumsikan z, n, dan f adalah positif. Ini akan menghasilkan hasil yang sama dengan kasus negatif, tetapi detail argumen lebih mudah diikuti. Kami mengirim setiap nilai-z ke sebuah “ember” dengan kedalaman $\Delta z = (f - n)/B$. Kami tidak akan menggunakan integerz-buffer jika memori nota premium, jadi berguna untuk membuat B sekecil mungkin.



Gambar 8.12 Sebuah z-buffer rasterisasi dua segitiga di masing-masing dari dua urutan yang mungkin. Segitiga pertama sepenuhnya raster. Segitiga kedua memiliki setiap piksel yang dihitung, tetapi untuk tiga piksel kontes kedalaman hilang, dan piksel tersebut tidak digambar. Gambar akhirnya tetap sama.

Jika kita mengalokasikan b bit untuk menyimpan nilai z, maka B =2^b. Kami membutuhkan bit yang cukup untuk memastikan setiap segitiga di depan segitiga lain akan memiliki kedalaman yang dipetakan ke tempat kedalaman yang berbeda.

Misalnya, jika Anda membuat scene di mana segitiga memiliki pemisahan setidaknya satu meter, maka Δz < 1 harus menghasilkan gambar tanpa artefak. Ada dua cara untuk membuat Δz lebih kecil: memindahkan n dan f lebih dekat atau meningkatkan b. Jika b telah diperbaiki, seperti dalam API atau pada platform perangkat keras tertentu, menyesuaikan n dan f adalah satu-satunya pilihan.

Ketepatan buffer-z harus ditangani dengan sangat hati-hati saat gambar perspektif dibuat. Nilai Δz di atas digunakan setelah pembagian perspektif. Ingat dari Bagian 7.3 bahwa hasil dari pembagian perspektif adalah

$$z = n + f = \frac{f_n}{z_w}$$

Kedalaman bin yang sebenarnya terkait dengan z_w, kedalaman dunia, bukan z, kedalaman pembagian pascaperspektif. Kita dapat memperkirakan ukuran nampan dengan membedakan kedua sisinya:

$$\Delta z \approx \frac{f_n \Delta z_w}{z_w^2}$$

Ukuran bin bervariasi dalam kedalaman. Ukuran tempat sampah di ruang dunia adalah

$$\Delta z_w \approx \frac{z_w^2 \Delta z}{fn}$$

Perhatikan bahwa besaran Δz adalah seperti yang telah didiskusikan sebelumnya. Bin terbesar adalah untuk z’ = f, di mana

$$\Delta z_w^{max} \approx \frac{f \Delta z}{n}$$

Perhatikan bahwa memilih n =0, pilihan yang wajar jika kita tidak ingin kehilangan objek tepat di depan mata, akan menghasilkan tempat sampah yang sangat besar—kondisi yang sangat buruk. Untuk membuat Δz_w^{max}sekecil mungkin, kita ingin meminimalkan f dan memaksimalkan n. Jadi, selalu penting untuk memilih n dan f dengan hati-hati.

Bayangan Per-verteks

Sejauh ini aplikasi yang mengirim segitiga ke dalam pipa bertanggung jawab untuk mengatur warna; rasterizer hanya menginterpolasi warna dan mereka ditulis langsung ke dalam gambar output. Untuk beberapa aplikasi ini sudah cukup, tetapi dalam banyak kasus kita ingin objek 3D digambar dengan bayangan, menggunakan persamaan iluminasi yang sama yang kita gunakan untuk rendering urutan gambar dalam Bab 4. Ingat bahwa persamaan memerlukan arah cahaya, arah mata, dan permukaan normal untuk menghitung warna permukaan.

Salah satu cara untuk menangani komputasi shading adalah dengan melakukannya di tahap vertex. Aplikasi menyediakan vektor normal pada simpul, dan posisi serta warna cahaya disediakan secara terpisah (tidak berbeda di permukaan, sehingga tidak perlu ditentukan untuk setiap simpul). Untuk setiap titik, arah ke penampil dan arah ke setiap cahaya dihitung berdasarkan posisi kamera, lampu, dan titik. Persamaan bayangan yang diinginkan dievaluasi untuk menghitung warna, yang kemudian diteruskan ke rasterizer sebagai warna titik. Bayangan pervertex kadang-kadang disebut Gouraudshading.

Salah satu keputusan yang harus dibuat adalah sistem koordinat di mana perhitungan bayangan dilakukan. Ruang dunia atau ruang mata adalah pilihan yang baik. Penting untuk memilih sistem koordinat yang ortonormal jika dilihat dalam ruang dunia, karena persamaan bayangan bergantung pada sudut antara vektor, yang tidak dipertahankan oleh operasi seperti skala tidak seragam yang sering digunakan dalam transformasi pemodelan, atau proyeksi perspektif, yang sering digunakan dalam proyeksi ke kanonik. melihat volume. Bayangan dalam ruang mata memiliki kelebihan yaitu kita tidak perlu melacak posisi kamera, karena kamera selalu berada di titik asal dalam ruang mata, dalam proyeksi perspektif, atau arah pandang selalu +z dalam proyeksi ortografis.

Gambar 8.13 Dua bola digambar menggunakan bayangan per titik (Gouraud). Karena segitiganya besar, artefak interpolasi terlihat.

Bayangan per titik memiliki kelemahan yaitu tidak dapat menghasilkan detail apapun dalam bayangan yang lebih kecil dari primitif yang digunakan untuk menggambar permukaan, karena hanya menghitung bayangan sekali untuk setiap titik dan tidak pernah di antara simpul. Misalnya, pada ruangan dengan lantai yang digambar menggunakan dua segitiga besar dan diterangi oleh sumber cahaya di tengah ruangan, bayangan hanya akan dievaluasi di sudut ruangan, dan nilai interpolasi kemungkinan akan terlalu gelap di tengahnya. Juga, permukaan melengkung yang diarsir dengan sorotan spekular harus digambar menggunakan primitif yang

cukup kecil sehingga sorotan dapat diselesaikan. Gambar 8.13 menunjukkan dua bola kita yang digambar dengan per-vertexshading.

Bayangan per-fragmen

Untuk menghindari artefak interpolasi yang terkait dengan shading per-vertex, kita dapat menghindari interpolasi warna dengan melakukan perhitungan shading setelah interpolasi, dalam tahap fragmen. Dalam naungan per-fragmen, persamaan naungan yang sama dievaluasi, tetapi persamaan tersebut dievaluasi untuk setiap fragmen menggunakan vektor interpolasi, bukan untuk setiap simpul menggunakan vektor dari aplikasi.

Bayangan per-fragmen kadang-kadang disebut bayangan Phong, yang membingungkan karena nama yang sama melekat pada model penerangan Phong. Dalam shading per-fragment, informasi geometrik yang diperlukan untuk shading dilewatkan melalui rasterizer sebagai atribut, sehingga tahap vertex harus berkoordinasi dengan tahap fragmen untuk menyiapkan data dengan tepat. Salah satu pendekatannya adalah dengan menginterpolasi normal permukaan ruang mata dan posisi titik sudut ruang mata, yang kemudian dapat digunakan seperti pada per-vertexshading. Gambar 8.14 menunjukkan dua bola kita yang digambar dengan per-fragmentshading.

Gambar 8.14 Dua bola digambar menggunakan bayangan per-fragmen. Karena segitiganya besar, artefak interpolasi terlihat.

Mapping Tekstur

Tekstur (dibahas dalam Bab 11) adalah gambar yang digunakan untuk menambahkan detail ekstra pada bayangan permukaan yang jika tidak akan terlihat terlalu homogen dan buatan. Idenya sederhana: setiap kali bayangan dihitung, kita membaca salah satu nilai yang digunakan dalam perhitungan bayangan—warna yang menyebar, misalnya—dari tekstur alih-alih menggunakan nilai atribut yang dilampirkan ke geometri yang dirender. Operasi ini dikenal sebagai pencarian tekstur: kode bayangan menentukan koordinat tekstur, sebuah titik dalam domain tekstur, dan sistem mapping tekstur menemukan nilai pada titik itu dalam gambar tekstur dan kembali. Nilai tekstur tersebut kemudian digunakan dalam perhitungan shading. Cara yang paling umum untuk mendefinisikan koordinat tekstur adalah dengan membuat koordinat tekstur atribut vertex lain. Setiap primitif kemudian tahu di mana ia tinggal dalam tekstur.

Frekuensi Bayangan

Keputusan tentang tempat untuk melakukan komputasi bergantung pada seberapa cepat perubahan warna—skala detail yang dihitung. Bayangan dengan fitur skala besar, seperti bayangan difus pada permukaan melengkung, dapat dievaluasi cukup jarang dan kemudian diinterpolasi: dapat dihitung dengan frekuensi bayangan rendah. Bayangan yang menghasilkan fitur skala kecil, seperti sorotan tajam atau tekstur detail, perlu dievaluasi pada frekuensi bayangan tinggi. Untuk detail yang perlu terlihat tajam dan tajam pada gambar, frekuensi bayangan harus setidaknya satu sampel bayangan per piksel.

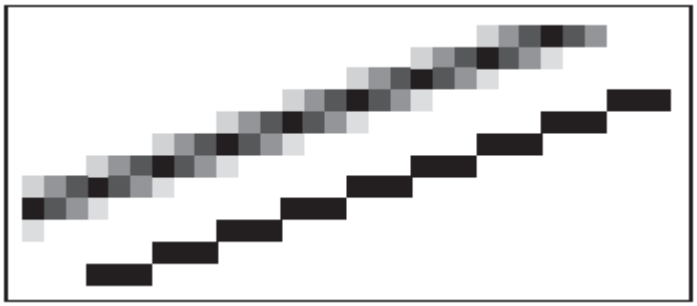
Jadi efek skala besar dapat dengan aman dihitung dalam tahap vertex, bahkan ketika vertex yang mendefinisikan primitif terpisah banyak piksel. Efek yang memerlukan frekuensi

bayangan yang tinggi juga dapat dihitung pada tahap verteks, selama simpul-simpul tersebut berdekatan satu sama lain dalam gambar; sebagai alternatif, mereka dapat dihitung pada tahap fragmen ketika primitif lebih besar dari piksel.

Misalnya, saluran perangkat keras seperti yang digunakan dalam permainan komputer, umumnya menggunakan primitif yang mencakup beberapa piksel untuk memastikan efisiensi tinggi, biasanya melakukan sebagian besar perhitungan bayangan per fragmen. Di sisi lain, sistem PhotoRealistic RenderMan melakukan semua perhitungan bayangan per titik, setelah terlebih dahulu membagi, atau memotong, semua permukaan menjadi segi empat kecil yang disebut mikropoligon yang berukuran kira-kira piksel. Karena primitifnya kecil, shading per-vertex dalam sistem ini mencapai frekuensi shading yang tinggi yang cocok untuk shading detail.

8.3 ANTI ALIASING SEDERHANA

Seperti halnya ray tracing, rasterisasi akan menghasilkan garis bergerigi dan tepi segitiga jika kita membuat penentuan semua atau tidak sama sekali apakah setiap piksel berada di dalam primitif atau tidak. Faktanya, himpunan fragmen yang dihasilkan oleh algoritme rasterisasi segitiga sederhana yang dijelaskan dalam bab ini, terkadang disebut rasterisasi standar atau alias, persis sama dengan kumpulan piksel yang akan dipetakan ke segitiga tersebut oleh pelacak sinar yang mengirimkan satu sinar melalui pusat setiap piksel.



Gambar 8.15 Garis antialias dan bergerigi dilihat dari jarak dekat sehingga piksel individual terlihat.

Juga seperti dalam ray tracing, solusinya adalah membiarkan piksel sebagian tertutup oleh primitif (Crow, 1978). Dalam praktiknya, bentuk blurring ini membantu kualitas visual, terutama dalam animasi. Ini ditunjukkan sebagai garis atas Gambar 8.15.

Ada sejumlah pendekatan berbeda untuk antialiasing dalam aplikasi rasterisasi. Seperti halnya ray tracer, kita dapat menghasilkan gambar antialias dengan menyetel setiap nilai piksel ke warna rata-rata gambar di atas area persegi yang dimiliki piksel, suatu pendekatan yang dikenal sebagai pemfilteran kotak. Ini berarti kita harus berpikir bahwa entitas yang dapat ditarik jatuh memiliki area yang terdefinisi dengan baik. Misalnya, garis pada Gambar 8.15 dapat dianggap sebagai mendekati persegi panjang satu piksel. Ada filter yang lebih baik daripada kotaknya, tapiboxfilter akan cukup untuk semua aplikasi kecuali yang paling menuntut.

Cara termudah untuk mengimplementasikan filter antialiasing kotak adalah dengan supersampling: buat gambar dengan resolusi sangat tinggi lalu turunkan sampel. Misalnya, jika tujuan kita adalah gambar garis 256×256 piksel dengan lebar 1,2 piksel, kita dapat mensterilkan versi persegi panjang dari garis dengan lebar 4,8 piksel pada layar 1024 × 1024, lalu rata-rata 4 × 4 kelompok piksel untuk mendapatkan warna untuk masing-masing 256 × 256 piksel dalam gambar "menyusut". Ini adalah perkiraan dari gambar kotak yang disaring

sebenarnya, tetapi bekerja dengan baik ketika objek tidak memiliki teks yang sangat kecil dibandingkan dengan jarak antar piksel.

Namun, supersampling cukup mahal. Karena tepi yang sangat tajam yang menyebabkan aliasing biasanya disebabkan oleh tepi primitif, daripada variasi tiba-tiba dalam bayangan dalam primitif, optimalisasi yang banyak digunakan adalah untuk mengambil sampel visibilitas pada tingkat yang lebih tinggi daripada bayangan. Jika informasi tentang cakupan dan kedalaman disimpan untuk beberapa titik dalam setiap piksel, antialiasing yang sangat baik dapat dicapai bahkan jika hanya satu warna yang dihitung. Dalam sistem seperti RenderMan yang menggunakan per-vertexshading, ini dicapai dengan rasterisasi pada resolusi tinggi: tidak mahal untuk melakukannya karena shading hanya diinterpolasi untuk menghasilkan warna untuk banyak fragmen, atau sampel visibilitas. Dalam sistem dengan per-fragmentshading, seperti pipeline perangkat keras, antialiasing multisampel dicapai dengan menyimpan untuk setiap fragmena satu warna ditambah coveragemask dan satu set nilai kedalaman.

8.4 MEMUSNAHKAN PRIMITIFISASI DEMI EFISIENSI

Kekuatan rendering urutan objek, yang membutuhkan satu lintasan untuk semua geometri dalam scene, juga merupakan kelemahan untuk scene yang kompleks. Misalnya, dalam model seluruh kota, hanya beberapa bangunan yang mungkin terlihat pada waktu tertentu. Gambar yang benar dapat diperoleh dengan menggambar semua primitif dalam scene, tetapi banyak usaha akan terbuang sia-sia untuk memproses geometri yang ada di belakang bangunan yang terlihat, atau di belakang penampil, dan oleh karena itu tidak berkontribusi pada gambar akhir.

Mengidentifikasi dan membuang geometri tak kasat mata untuk menghemat waktu yang akan dihabiskan untuk memprosesnya dikenal sebagai pemusnahan. Tiga strategi pemusnahan yang umum diterapkan (sering digunakan bersama-sama) adalah

- **view volume culling** — penghapusan geometri yang berada di luar volume tampilan;
- **occlusion culling** — penghapusan geometri yang mungkin berada dalam volume tampilan tetapi dikaburkan, atau dihalangi, oleh geometri lain yang lebih dekat ke kamera;
- **backface culling** — penghapusan primitif yang menghadap jauh dari kamera.

Kami akan membahas secara singkat pemusnahan volume tampilan dan pemusnahan muka belakang, tetapi pemusnahan dalam topik kompleks sistem kinerja tinggi, lihat (Akenine-Mo¨lleretal., 2008) untuk diskusi lengkap dan untuk informasi tentang pemusnahan oklusi.

View Volume Culling

Ketika seluruh primitif terletak di luar volume tampilan, itu dapat dimusnahkan, karena tidak akan menghasilkan fragmen saat diraster. Jika kita dapat menyisihkan banyak primitif dengan tes cepat, kita mungkin dapat mempercepat menggambar secara signifikan. Di sisi lain, menguji primitif secara individual untuk memutuskan dengan tepat mana yang perlu digambar mungkin lebih mahal daripada membiarkan rasterizer menghilangkannya.

Pemusnahan volume tampilan, juga dikenal sebagai pemusnahan tampilan frustum, sangat membantu ketika banyak segitiga dikelompokkan ke dalam objek dengan volume pembatas terkait. Jika volume pembatas terletak di luar volume tampilan, maka lakukan juga semua segitiga yang membentuk objek. Misalnya, jika kita memiliki 1000 segitiga yang dibatasi oleh satu bola dengan pusat c dan jari-jari r, kita dapat memeriksa apakah bola terletak di luar bidang kliping,

$$(p - a) \cdot n = 0$$

di mana sebuah titik pada bidang, dan p adalah variabel. Ini setara dengan memeriksa apakah jarak bertanda dari pusat bola c ke bidang lebih besar dari +r. Ini sama dengan cek bahwa

$$\frac{(c-a)}{||n||} > r$$

Perhatikan bahwa bola mungkin tumpang tindih dengan bidang bahkan dalam kasus di mana semua segitiga terletak di luar bidang. Jadi, ini adalah tes konservatif. Seberapa konservatif tes ini tergantung pada seberapa baik bola membatasi objek. Ide yang sama dapat diterapkan secara hierarkis jika scene diatur dalam salah satu struktur data spasial yang dijelaskan dalam Bab 12.

Backface Culling

Ketika model poligonal tertutup, yaitu, mereka terikat pada ruang tertutup tanpa lubang, maka mereka sering dianggap memiliki vektor normal yang menghadap ke luar seperti yang dibahas dalam Bab 10. Untuk model seperti itu, poligon yang menghadap jauh dari mata pasti ditarik oleh poligon yang menghadap mata. Dengan demikian, poligon tersebut dapat dimusnahkan bahkan sebelum pipa dimulai. Pengujian untuk kondisi ini sama dengan yang digunakan untuk menggambar siluet yang diberikan dalam Bagian 10.3.1.

Pertanyaan yang Sering Diajukan

- Saya sering melihat kliping dibahas panjang lebar, dan ini adalah proses yang jauh lebih terlibat daripada yang dijelaskan dalam bab ini. Apa yang terjadi disini?

Kliping yang dijelaskan dalam bab ini berfungsi, tetapi tidak memiliki optimalisasi yang dimiliki oleh pemotong berkekuatan industri. Optimalisasi ini dibahas secara rinci dalam karya definitif Blinn yang tercantum dalam catatan bab.

- Bagaimana poligon yang bukan segitiga diraster?

Ini dapat dilakukan secara langsung garis pindai demi garis pindai, atau dapat dipecah menjadi segitiga. Yang terakhir tampaknya menjadi teknik yang lebih populer.

- Apakah antialias selalu lebih baik?

Tidak. Beberapa gambar terlihat lebih tajam tanpa antialiasing. Banyak program menggunakan "font layar" yang tidak dikenal karena lebih mudah dibaca.

- Dokumentasi untuk API saya berbicara tentang "grafis scene" dan "tumpukan matriks". Apakah ini bagian dari jalur grafis?

Pipa grafis tentu saja dirancang dengan pemikiran ini, dan apakah kita mendefinisikannya sebagai bagian dari pipa adalah masalah selera. Buku ini menunda diskusi mereka sampai Bab 12.

- Apakah buffer z jarak seragam lebih baik daripada buffer standar yang mencakup nonlinier matriks perspektif?

Tergantung. Salah satu "fitur" dari nonlinier adalah bahwa buffer-z memiliki lebih banyak resolusi di dekat mata dan lebih sedikit di kejauhan. Jika sistem level-of-detail digunakan, maka geometri di kejauhan lebih kasar dan "ketidakadilan" buffer-z bisa menjadi hal yang baik.

- Apakah perangkat lunak z-buffer pernah berguna?

Ya. Sebagian besar film yang menggunakan grafis komputer 3D telah menggunakan varian dari perangkat lunak z-buffer yang dikembangkan oleh Pixar (Cook, Carpenter, & Catmull, 1987).

Catatan

Buku yang bagus tentang mendesain saluran grafis adalah Jim Blinn's Corner: A Trip Down the Graphics Pipeline (J. Blinn, 1996). Banyak detail bagus dari pipeline dan pemusnahan

yang ada di 3DGameEngineDesign(Eberly,2000) dan Rendering Real-Time (Akenine-Mo¨lleretal.,2008).

Latihan

1. Misalkan dalam transformasi perspektif kita memiliki n = 1 dan f = 2. Dalam keadaan apa kita akan memiliki "pembalikan" di mana sebuah titik sebelum dan sesudah perspektif mengubah ips dari depan ke belakang mata atau sebaliknya?
2. Apakah ada alasan untuk tidak memotong x dan y setelah pembagian perspektif (lihat Gambar 11.2, tahap 3)?
3. Turunkan bentuk inkremental dari algoritma menggambar garis titik tengah dengan warna pada titik akhir untuk $0 < m \leq 1$.
4. Memodifikasi algoritma gambar segitiga sehingga akan menggambar tepat satu piksel untuk titik-titik pada sisi segitiga yang melalui$(x, y) = (-1, -1)$.
5. Misalkan Anda sedang merancang buffer-z bilangan bulat untuk simulasi penerbangan di mana semua objek setidaknya setebal satu meter, tidak pernah lebih dekat ke pemirsa dari 4 meter, dan mungkin sejauh 100 km. Berapa banyak bit yang dibutuhkan dalam z-buffer untuk memastikan tidak ada kesalahan visibilitas? Misalkan kesalahan visibilitas hanya penting di dekat penampil, yaitu untuk jarak kurang dari 100 meter. Berapa banyak bit yang dibutuhkan dalam kasus itu?

BAB 9
PEMROSESAN SINYAL

Dalam grafis, kita sering berurusan dengan fungsi variabel kontinu: gambar adalah contoh pertama yang Anda lihat, tetapi Anda akan menemukan lebih banyak lagi saat Anda melanjutkan penjelajahan grafis. Sesuai dengan sifatnya, fungsi kontinu tidak dapat direpresentasikan secara langsung di komputer; kita harus merepresentasikannya dengan menggunakan jumlah bit yang terbatas. Salah satu pendekatan yang paling berguna untuk merepresentasikan fungsi kontinu adalah dengan menggunakan sampel fungsi: simpan saja nilai fungsi pada banyak titik berbeda dan rekonstruksi nilai di antara saat dan jika diperlukan.

Anda sekarang sudah familiar dengan gagasan untuk merepresentasikan sebuah gambar menggunakan grid piksel dua dimensi—jadi Anda telah melihat representasi sampel! Pikirkan gambar yang ditangkap oleh kamera digital: gambar sebenarnya dari scene yang dibentuk oleh lensa kamera adalah fungsi kontinu dari posisi pada bidang gambar, dan kamera mengubah fungsi itu menjadi kisi sampel dua dimensi. Secara matematis, kamera mengonversi fungsi tipe $\mathbb{R}^2 \rightarrow$ C (di mana C adalah himpunan warna) ke array sampel warna dua dimensi, atau fungsi tipe $\mathbb{Z}^2 \rightarrow$ C.

Contoh lain dari representasi sampel adalah tablet digitalisasi 2D, seperti layar komputer tablet atau komputer terpisah yang digunakan oleh seniman. Dalam hal ini, fungsi aslinya adalah gerakan stylus, yang merupakan posisi 2D yang berubah-ubah terhadap waktu, atau fungsi tipe $\mathbb{R} \rightarrow \mathbb{R}^2$. Digitizer mengukur posisi stylus di banyak titik waktu, menghasilkan urutan koordinat 2D, atau fungsi tipe $\mathbb{Z} \rightarrow \mathbb{R}^2$.Sistem penangkap gerak melakukan hal yang persis sama untuk penanda khusus yang dipasang pada tubuh aktor: itu mengambil posisi 3D penanda dari waktu ke waktu ($\mathbb{R} \rightarrow \mathbb{R}^3$) dan membuatnya menjadi serangkaian pengukuran posisi seketika ($\mathbb{Z} \rightarrow \mathbb{R}^3$).

Naikindimensi,pemindai CT medis,digunakan secara invasifmemeriksa bagian dalam tubuh seseorang, mengukur kepadatan sebagai fungsi posisi di dalam tubuh. Output pemindai adalah kisi 3D nilai kerapatan: ia mengubah kerapatan benda ($\mathbb{R}^3 \rightarrow \mathbb{R}$) menjadi larik 3D bilangan real ($\mathbb{Z}^3 \rightarrow \mathbb{R}$).

Contoh-contoh ini tampak berbeda, tetapi sebenarnya semuanya dapat ditangani menggunakan matematika yang persis sama. Dalam semua kasus, suatu fungsi disampel pada titik-titik kisi dalam satu atau lebih dimensi, dan dalam semua kasus kita harus mampu merekonstruksi fungsi kontinu asli itu dari larik sampel.

Dari contoh gambar 2D, tampaknya pikselnya cukup, dan kita tidak perlu memikirkan fungsi kontinu lagi setelah kamera telah mendiskritisasi gambar. Tetapi bagaimana jika kita ingin membuat gambar lebih besar atau lebih kecil di layar, terutama dengan faktor skala non-bilangan bulat? Ternyata algoritma paling sederhana untuk melakukan ini berkinerja buruk, memperkenalkan artefak visual yang jelas yang dikenal sebagai aliasing. Menjelaskan mengapa aliasing terjadi dan memahami cara mencegahnya memerlukan teori-teori matematis sampling. Algoritma yang dihasilkan agak sederhana, tetapi alasan di baliknya, dan detail membuatnya berkinerja baik, bisa jadi tidak kentara.

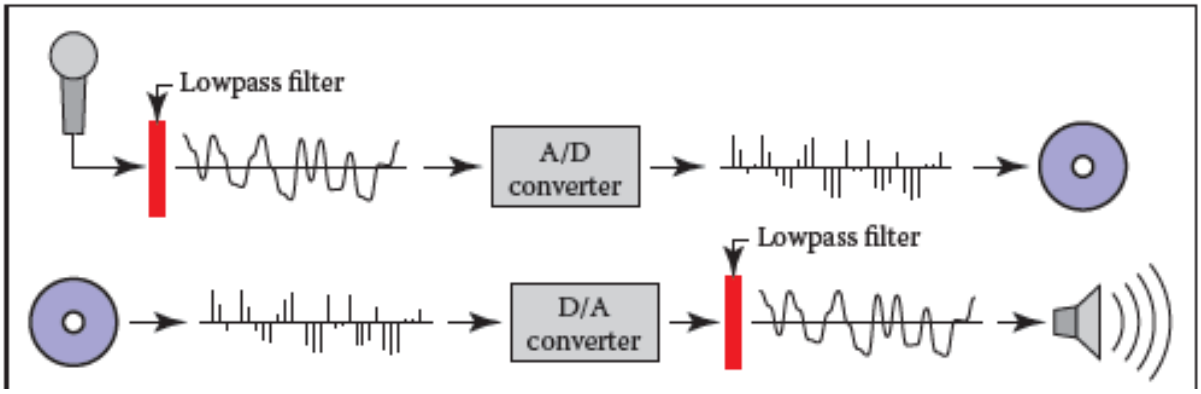
Mewakili fungsi kontinu di komputer, tentu saja, tidak unik untuk grafis; juga ide pengambilan sampel dan rekonstruksi. Representasi sampel digunakan dalam aplikasi dari audioto digital fisika komputasi, dan grafis hanyalah salah satu (dan tidak berarti pengguna pertama) dari algoritma dan matematika terkait. Fakta mendasar tentang bagaimana melakukan sampling dan rekonstruksi telah dikenal di bidang komunikasi sejak tahun 1920-an

dan dinyatakan dalam bentuk yang kita gunakan pada tahun 1940-an (Shannon & Weaver, 1964).

Terakhir, kita masuk ke detail sudut pandang domain frekuensi, yang memberikan banyak wawasan tentang perilaku algoritme ini.

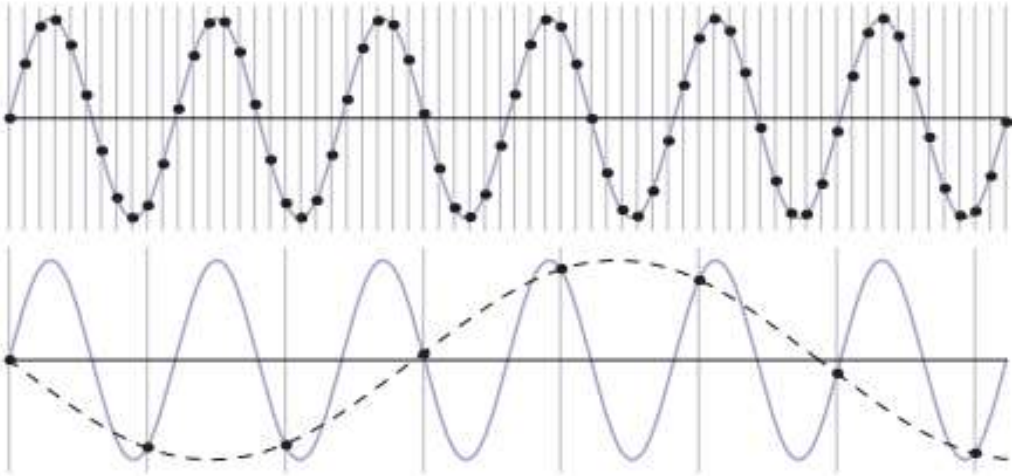
9.1 AUDIO DIGITAL : SAMPLING DI 1D

Meskipun representasi sampel telah digunakan selama bertahun-tahun di bidang telekomunikasi, pengenalan compact disc 1982, menyusul peningkatan penggunaan rekaman digital untuk audio pada dekade sebelumnya, adalah aplikasi pengambilan sampel pertama yang sangat terlihat oleh konsumen.



Gambar 9.1 Pengambilan sampel dan rekonstruksi dalam audio digital.

Dalam perekaman audio, mikrofon mengubah suara, yang ada sebagai gelombang tekanan di udara, menjadi tegangan yang berubah-ubah terhadap waktu menjadi ukuran perubahan tekanan udara pada titik di mana mikrofon berada. Sinyal listrik ini perlu disimpan sedemikian rupa sehingga dapat diputar ulang di lain waktu dan dikirim ke pengeras suara yang mengubah tegangan kembali menjadi gelombang tekanan dengan menggerakkan diafragma pada sinkronisasi dengan tegangan.



Gambar 9.2 Gelombang sinus (kurva biru) sampel pada dua tingkat yang berbeda. Atas: pada laju sampel yang tinggi, sampel yang dihasilkan (titik hitam) mewakili sinyal dengan baik. Bawah: laju sampel yang lebih rendah menghasilkan hasil yang ambigu: sampel persis sama dengan yang dihasilkan dari pengambilan sampel gelombang dengan frekuensi yang jauh lebih rendah (kurva putus-putus).

Pendekatan digital untuk merekam sinyal audio (Gambar 9.1) menggunakan sampel: konverter analog-ke-digital/*analog-to-digital converter* (A/D converter, atau ADC) mengukur tegangan ribuan kali per detik, menghasilkan aliran bilangan bulat yang dapat dengan mudah disimpan di sejumlah media, katakanlah disk di komputer di studio rekaman, atau ditransmisikan ke lokasi lain, katakanlah memori di pemutar audio portabel. Pada waktu pemutaran, data dibacakan pada kecepatan yang sesuai dan dikirim ke konverter digital-ke-analog/ *digital-to-analog converter* (konverter D/A, atau DAC). DAC menghasilkan tegangan sesuai dengan angka yang diterimanya, dan, asalkan kita mengambil sampel yang cukup untuk mewakili variasi tegangan, sinyal listrik yang dihasilkan, untuk semua tujuan praktis, identik dengan input.

Ternyata jumlah sampel per detik yang diperlukan untuk menghasilkan reproduksi yang baik bergantung pada seberapa tinggi nada suara yang digunakan untuk mencoba merekam. Sample rate yang bekerja dengan baik untuk mereproduksi string bass atau kick drum menghasilkan hasil yang terdengar aneh jika kita mencoba merekam piccolo atau cymbal; tetapi suara tersebut diproduksi ulang dengan baik dengan sampel yang lebih tinggi. Untuk menghindari *artefak undersampling* ini, perekam audio digital menyaring input ke ADC untuk menghilangkan frekuensi tinggi yang dapat menyebabkan masalah.

Jenis masalah lain muncul di sisi output. DAC menghasilkan tegangan yang berubah setiap kali sampel baru masuk, tetapi tetap konstan sampai sampel berikutnya, menghasilkan grafis berbentuk tangga. Tangga ini bertindak seperti kebisingan, menambahkan suara dengung frekuensi tinggi yang bergantung pada sinyal. Untuk menghilangkan artefak rekonstruksi ini, pemutar audio digital menyaring output dari DAC untuk menghaluskan bentuk gelombang.

Pengambilan Sampel Artefak dan Aliasing

Rantai perekaman audio digital dapat berfungsi sebagai model konkret untuk proses pengambilan sampel dan rekonstruksi yang terjadi dalam grafis. Jenis undersampling dan artefak rekonstruksi yang sama juga terjadi dengan gambar atau sinyal sampel lainnya dalam grafis, dan solusinya sama: penyaringan sebelum pengambilan sampel dan penyaringan lagi selama rekonstruksi.

Contoh konkret dari jenis artefak yang dapat muncul dari frekuensi sampel yang terlalu rendah ditunjukkan pada Gambar 9.2. Di sini kami mengambil sampel gelombang sinus sederhana menggunakan dua frekuensi sampel yang berbeda: 10,8 sampel per siklus di bagian atas dan 1,2 sampel per siklus di bagian bawah. Laju yang lebih tinggi menghasilkan satu set sampel yang jelas menangkap sinyal dengan baik, tetapi sampel yang dihasilkan dari laju sampel yang lebih rendah tidak dapat dibedakan dari sampel gelombang sinus berfrekuensi rendah—sebenarnya, menghadapi rangkaian sampel ini, sinusoid berfrekuensi rendah tampaknya lebih memungkinkan interpretasi.

Setelah pengambilan sampel dilakukan, tidak mungkin untuk mengetahui yang mana dari dua sinyal—gelombang sinus cepat atau lambat—yang asli, dan oleh karena itu tidak ada metode tunggal yang dapat merekonstruksi sinyal dengan benar dalam kedua kasus. Karena sinyal frekuensi tinggi "berpura-pura menjadi" sinyal frekuensi rendah, fenomena ini dikenal sebagai aliasing.

Aliasing muncul setiap kali kesalahan dalam pengambilan sampel dan rekonstruksi menyebabkan artefak pada frekuensi yang mengejutkan. Dalam audio, aliasing mengambil bentuk nada tambahan yang terdengar aneh—bel berbunyi pada 10KHz, setelah diambil sampelnya pada 8KHz, berubah menjadi nada 6KHz. Dalam citra, aliasing sering mengambil bentuk pola moir´e yang dihasilkan dari interaksi grid sampel dengan fitur reguler dalam citra, misalnya tirai windows pada Gambar 9.34.

Contoh lain dari aliasing dalam gambar sintetis adalah undakan tangga yang familiar pada garis lurus yang ditampilkan hanya dengan piksel hitam dan putih (Gambar 9.34). Ini adalah contoh fitur skala kecil (tepi tajam dari garis) menciptakan artefak pada skala yang berbeda (untuk garis lereng dangkal tangga sangat panjang).

Masalah dasar pengambilan sampel dan rekonstruksi dapat dipahami hanya berdasarkan fitur yang terlalu kecil atau terlalu besar, tetapi beberapa pertanyaan kuantitatif lebih sulit untuk dijawab:

- Berapa jumlah sampel yang cukup tinggi untuk memastikan hasil yang baik?
- Jenis filter apa yang cocok untuk pengambilan sampel dan rekonstruksi?
- Tingkat smoothing apa yang diperlukan untuk menghindari aliasing?

Jawaban yang kuat untuk pertanyaan-pertanyaan ini harus menunggu sampai kita mengembangkan teori sepenuhnya di Bagian 9.5

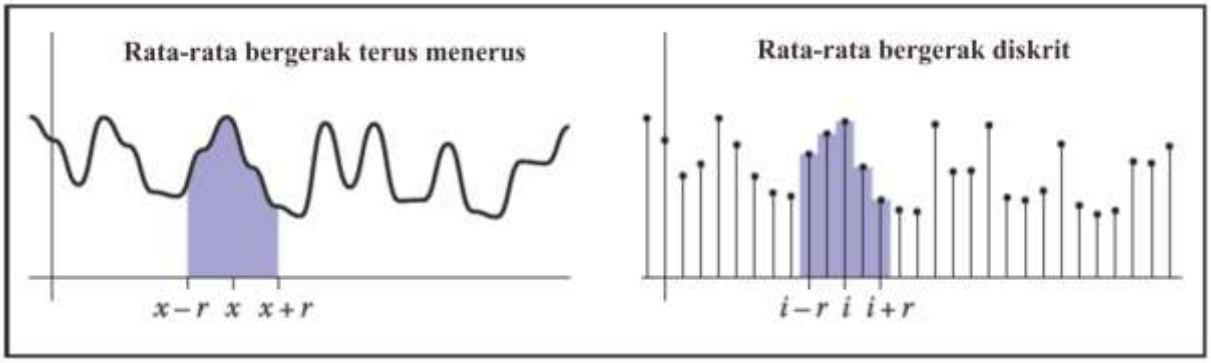
9.2 KONVOLUSI

Sebelum kita membahas algoritma untuk sampling dan rekonstruksi, pertama-tama kita akan menguji konsep matematika yang menjadi dasarnya—konvolusi. Konvolusi adalah konsep matematika sederhana yang mendasari algoritma yang digunakan untuk pengambilan sampel, penyaringan, dan rekonstruksi. Ini juga merupakan dasar bagaimana kita akan menganalisis algoritme ini nanti dalam bab ini.

Konvolusi adalah operasi pada fungsi: dibutuhkan dua fungsi dan menggabungkannya untuk menghasilkan fungsi baru. Dalam buku ini, operator konvolusi dilambangkan dengan bintang: hasil penerapan konvolusi pada fungsi f dan g adalah f * g. Kita katakan bahwa f adalah konvolusi dengan g, dan f * g adalah konvolusi dari f dan g.

Konvolusi dapat diterapkan baik ke fungsi kontinu (fungsi f(x) yang didefinisikan untuk argumen real x) atau ke barisan diskrit (fungsi a[i] yang didefinisikan hanya untuk argumen bilangan bulat i). Ini juga dapat diterapkan ke fungsi yang didefinisikan pada domain satu dimensi, dua dimensi, atau lebih tinggi (yaitu, fungsi satu, dua, atau lebih argumen). Kita akan mulai dengan kasus diskrit satu dimensi terlebih dahulu, kemudian melanjutkan ke fungsi kontinu dan fungsi dua dan tiga dimensi.

Untuk memudahkan definisi, kita umumnya berasumsi bahwa domain fungsi berlangsung selamanya, meskipun tentu saja dalam praktiknya mereka harus berhenti di suatu tempat, dan kita harus menangani titik akhir dengan cara khusus.



Gambar 9.3 Smoothing menggunakan Pergerakan rata-rata.

Pergerakan Rata-rata

Untuk mendapatkan gambaran dasar konvolusi, perhatikan contoh pemulusan fungsi 1D menggunakan Pergerakan rata-rata (Gambar 9.3). Untuk mendapatkan nilai yang dihaluskan di sembarang titik, kami menghitung rata-rata fungsi pada rentang yang

memperpanjang jarak r di setiap arah. Jarak r, yang disebut radius operasi pemulusan, adalah parameter yang menunjukkan banyak pemulusan yang terjadi.

Kita dapat menyatakan ide ini secara matematis untuk fungsi diskrit atau kontinu. Jika kita memuluskan fungsi kontinu g(x), rata-rata berarti mengintegrasikan g pada suatu interval dan kemudian membaginya dengan panjang interval:

$$h(x) = \frac{1}{2r}\int_{x-r}^{x+r} g(t)dt$$

Di sisi lain, jika kita menghaluskan fungsi diskrit a[i], rata-rata berarti menjumlahkan a untuk rentang indeks dan membaginya dengan jumlah nilai:

$$c(i) = \frac{1}{2r+1}\sum_{j=1-r}^{i+r} a[j]$$

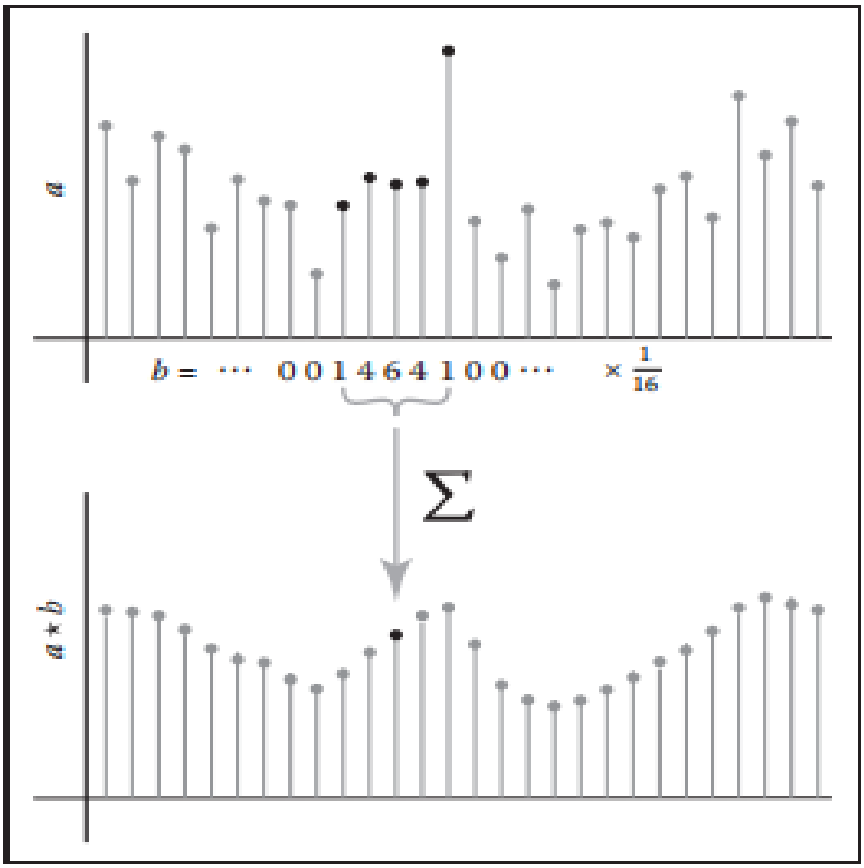
Dalam setiap kasus, konstanta normalisasi dipilih sehingga jika kita memuluskan fungsi konstan, hasilnya akan menjadi fungsi yang sama. Gagasan tentang Pergerakan rata-rata ini adalah inti dari konvolusi; satu-satunya perbedaan adalah bahwa dalam konvolusi Pergerakan rata-rata adalah rata-rata tertimbang.

Konvolusi Diskrit

Kita akan mulai dengan kasus konvolusi yang paling konkrit: menggulung barisan diskrita[i]dengan barisandiskrit lainb[i]. Hasilnya adalah urutan yang tidak jelas (a b)[i]. Prosesnya seperti pemulusan a dengan Pergerakan rata-rata, tetapi kali ini alih-alih menimbang semua sampel dalam jarak yang sama, kami menggunakan urutan kedua b untuk memberikan bobot pada setiap sampel (Gambar 9.4). Nilai b[I − j] memberikan bobot untuk sampel pada posisi j, yang berada pada jarak i j dari indeks i tempat kita mengevaluasi konvolusi. Berikut adalah definisi dari (a * b), dinyatakan sebagai rumus:

$$(a * b)[i] = \sum_{j} a[j]b[i-j]$$

Dengan menghilangkan batas pada j, kami menunjukkan bahwa jumlah ini berjalan di atas semua bilangan bulat (yaitu, dari -∞ ke +∞). Gambar 9.4 mengilustrasikan bagaimana satu sampel output dihitung, menggunakan contoh b = 1/16[..., 0, 1, 4, 6, 4, 1, 0, ...]—yaitu, b[0] = 6/16, a[±1] = 4/16, dst.



Gambar 9.4 Menghitung satu nilai dalam konvolusi diskrit deret a dengan filter b yang mendukung lebar lima sampel. Setiap sampel di a b adalah rata-rata sampel terdekat di a, dibobot dengan nilai b.

Dalam grafis, salah satu dari dua fungsi biasanya akan memiliki dukungan terbatas (seperti contoh pada Gambar 9.4), yang berarti bahwa itu bukan nol hanya pada interval nilai argumen yang terbatas. Jika kita berasumsi bahwa b memiliki tumpuan berhingga, ada radius r sedemikian rupa sehingga b[k] = 0 kapan pun |k| > r. Dalam hal ini, kita dapat menulis jumlah di atas sebagai

$$(a * b)[i] \sum_{j=i-r}^{i+r} a[j]b[i-j]$$

dan kita dapat mengungkapkan definisi dalam kode sebagai

```
function convolve(sequence a, filter b, int i)
s = 0
r = b.radius
for j = i − r to i + r do
s = s + a[j]b[i − j]
return s
```

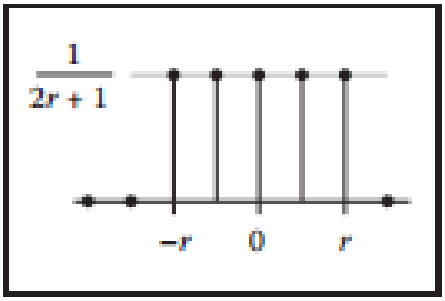
9.3 FILTER KONVOLUSI

Konvolusi penting karena kita dapat menggunakannya untuk melakukan pemfilteran. Melihat kembali contoh pemfilteran pertama kita, Pergerakan rata-rata, sekarang kita dapat menafsirkan kembali operasi pemulusan itu sebagai konvolusi dengan urutan tertentu. Ketika kita menghitung rata-rata pada beberapa rentang indeks yang terbatas, itu sama dengan pembobotan titik-titik dalam kisaran semuanya secara identik dan pembobotan sisa poin

dengan nol. Filter jenis ini, yang memiliki nilai konstan selama interval di mana bukan nol, dikenal sebagai filter kotak (karena terlihat seperti persegi panjang jika Anda menggambar grafisnya—lihat Gambar 9.5). Untuk filter kotak berjari-jari r bobotnya adalah 1/(2r + 1) :

$$b[k] = \begin{cases} \dfrac{1}{2r+1} & -r \le k \le r \\ 0 & jika\ tidak \end{cases}$$

Jika Anda mensubstitusikan filter ini ke dalam Persamaan (9.2), Anda akan menemukan bahwa filter tersebut tereduksi menjadi Pergerakan rata-rata dalam Persamaan (9.1). Seperti dalam contoh ini, filter konvolusi biasanya dirancang sedemikian rupa sehingga jumlahnya menjadi 1. Dengan cara itu, filter tersebut tidak mempengaruhi tingkat sinyal secara keseluruhan.



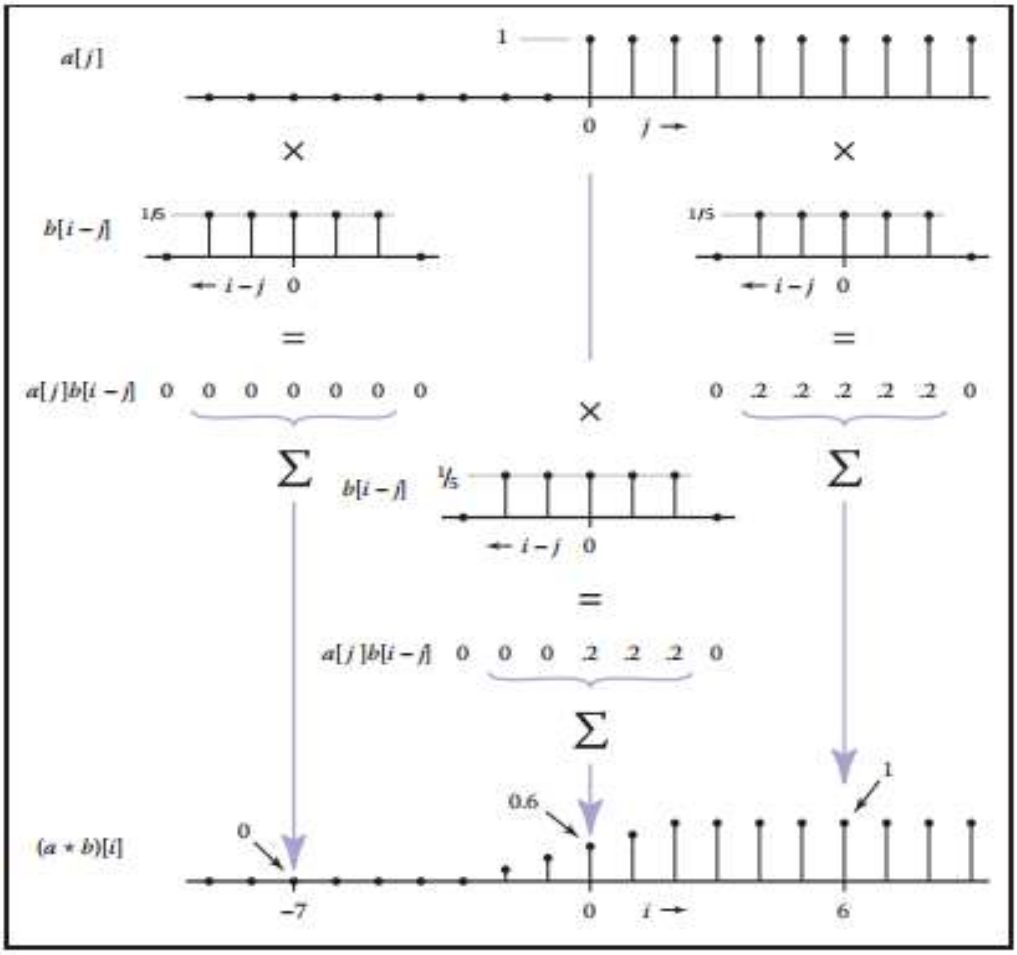
Gambar 9.5 Filter kotak diskrit.

Contoh (Konvolusi kotak dan langkah).
Untuk contoh sederhana penyaringan, biarkan sinyal menjadi fungsi langkah

$$a[i] = \begin{cases} 1 & i \ge 0 \\ 0 & i < 0 \end{cases}$$

dan filter menjadi filter kotak lima titik yang berpusat di nol,

$$b[k] = \frac{1}{5}\begin{cases} 1 & -2 \le k \le 2 \\ 0 & jika\ tidak \end{cases}$$



Gambar 9.6 Konvolusi diskrit fungsi kotak dengan fungsi langkah.

Berapakah hasil perkalian a dan b? Pada indeks tertentu i, seperti yang ditunjukkan pada Gambar 9.6, hasilnya adalah rata-rata dari fungsi langkah pada rentang dari i−2 hingga i +2. If i < −2, kita rata-ratakan semua nol dan hasilnya nol. Jika i ≥ 2, kita rata-ratakan semua dan hasilnya adalah satu. Di antara ada i +3, menghasilkan nilai $\frac{1+3}{5}$. Outputnya adalah jalur linier yang bergerak dari 0 ke 1 selama lima sampel: 1/5[..., 0, 0, 1, 2, 3, 4, 5, 5, ...].

Sifat Konvolusi

Cara kita menulisnya sejauh ini, konvolusi tampak seperti operasi asimetris: ini adalah urutan kita menghaluskan, dan memberikan bobot. Tetapi salah satu sifat dari konvolusi adalah bahwa itu sebenarnya tidak membuat perbedaan yang mana: filter dan sinyal dapat dipertukarkan. Untuk melihat ini, pikirkan kembali jumlah dalam Persamaan (9.2) dengan indeks dihitung dari asal filter b, bukan dari asal a. Artinya, kita mengganti j dengan i−k. Hasil dari perubahan variabel ini adalah

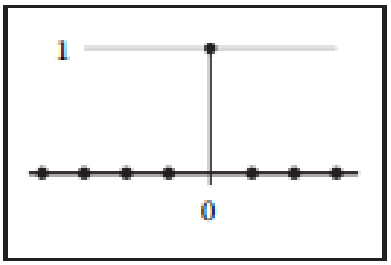
$$(a * b)[i] = \sum_k a[i-k]b[i-k]$$
$$= \sum_k b[k]a[i-k]$$

Ini persis sama dengan Persamaan (9.2) tetapi dengan a bertindak sebagai filter dan b bertindak sebagai sinyal. Jadi untuk sembarang barisan a dan b, (a * b) = (b * a), dan kita katakan bahwa konvolusi adalah operasi komutatif.[3]

Lebih umum, konvolusi adalah operasi "seperti perkalian". Seperti perkalian atau penambahan angka atau fungsi, baik urutan argumen maupun penempatan tanda kurung tidak mempengaruhi hasil. Juga, konvolusi berhubungan dengan penjumlahan dengan cara yang sama seperti perkalian. Tepatnya, konvolusi adalah komutatif dan asosiatif, dan itu adalah penjumlahan distributif.

Kumulatif *(a*b)[i] = (b*a)[i]*
Asosiatif *(a*b(b*c))[i] = ((a*b)*c)[i]*
Distributig *(a*(b*c))[i] = (a*b + a*c)[i]*

Sifat-sifat ini sangat alami jika kita menganggap konvolusi sebagai perkalian, dan sangat berguna untuk diketahui karena dapat membantu kita menghemat pekerjaan dengan menyederhanakan lilitan sebelum kita benar-benar menghitungnya. Misalnya, misalkan kita ingin mengambil barisan a dan mengonvolusikannya dengan tiga filter, b_1, b_2, dan b_3 — yaitu, kita menginginkan $((a * b_1)*b_2)*b_3$. Jika urutannya panjang dan filternya pendek (yaitu, mereka memiliki jari-jari kecil), itu jauh lebih cepat untuk menggabungkan tiga filter bersama-sama (menghitung $b_1*b_2*b_3$) dan akhirnya mengonvolusikan hasil dengan sinyal, menghitung $a*(b_1*b_2*b_3)$, yang kita ketahui dari asosiasi memberikan hasil yang sama.



Gambar 9.7 Filter identitas diskrit.

Filter yang sangat sederhana berfungsi sebagai identitas untuk konvolusi diskrit: filter diskrit berjari-jari nol, atau barisan d[i] = ..., 0, 0, 1, 0, 0, ... (Gambar 9.7). Jika kita dibelokkan dengan sinyal a, hanya akan ada satu suku bukan nol dalam jumlah:

$$(a * d)[i] = \sum_{j=i}^{j=i} a[j]d[i-j] = a[i]$$

Jadi jelas, convolvinga dengan d hanya memberikan kembali a lagi. Barisan d dikenal sebagai implus diskrit. Kadang-kadang berguna dalam mengekspresikan filter: misalnya, proses menghaluskan sinyal a dengan filter b dan kemudian mengurangkannya dari aslinya dapat dinyatakan sebagai konvolusi tunggal dengan filter d−b:

c = a – a * b = a * d – a * b = a * (d – b)

Konvolusi sebagai Jumlah Filter yang Digeser

Ada cara kedua, yang sepenuhnya setara, untuk menafsirkan Persamaan (9.2). Melihat sampel sofa tulang pada waktu mengarah ke interpretasi rata-rata tertimbang yang telah kita lihat. Tetapi jika kita menghilangkan [i], kita dapat menganggap penjumlahan sebagai

[3] Anda mungkin telah memperhatikan bahwa salah satu fungsi dalam jumlah konvolusi tampak terbalik— yaitu, b[k] memberikan bobot untuk unit sampel k sebelumnya dalam barisan, sementara b[−k] memberikan bobot untuk unit sampel k kemudian dalam urutan. Alasan untuk ini berkaitan dengan memastikan asosiatif; lihat Latihan 4. Sebagian besar filter yang kami gunakan simetris, jadi Anda hampir tidak perlu mengkhawatirkan hal ini.

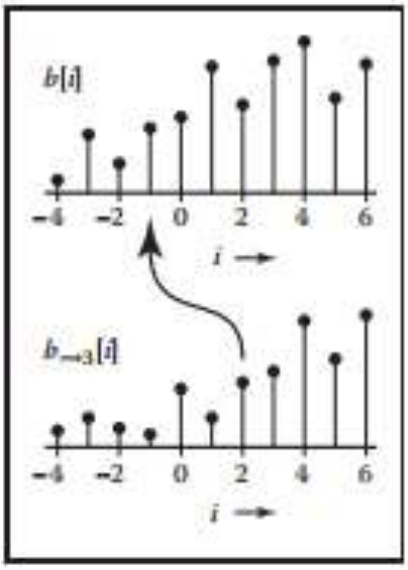
penjumlahan seluruh barisan. Satu notasi diperlukan untuk membuat ini bekerja: jika b adalah barisan, maka barisan yang sama digeser ke kanan sebanyak j tempat disebut b→j (Gambar 9.8):

$$b \rightarrow j\,[i] = b[i - j].$$

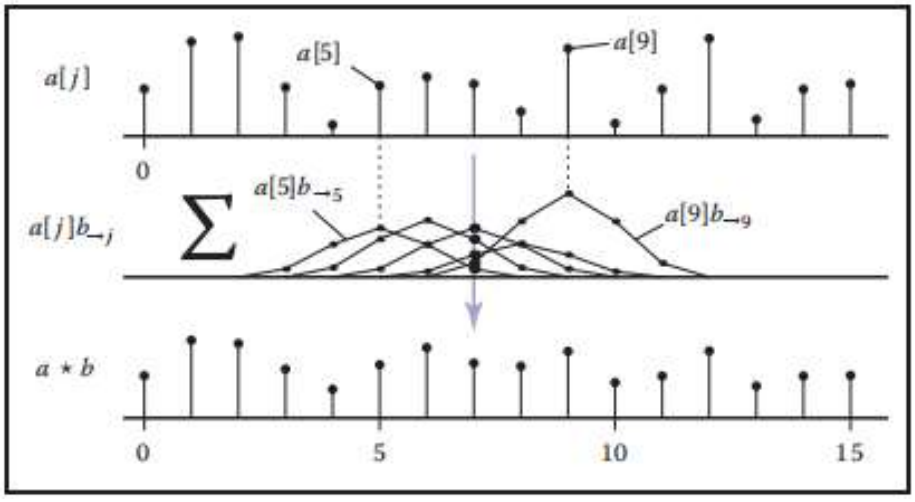
Kemudian, kita dapat menulis Persamaan (9.2) sebagai pernyataan tentang seluruh barisan (a b) daripada elemen demi elemen:

$$(a * b) = \sum_j a[j]b \rightarrow j$$

Melihat cara ini, konvolusi adalah jumlah salinan bergeser dari b, dibobot dengan entri a (Gambar 9.9). Karena komutatif, kita dapat memilih a atau b sebagai filter; jika kita memilih b, maka kita menjumlahkan satu salinan filter untuk setiap sampel dalam input.



Gambar 9.8 Menggeser barisan b untuk mendapatkan b→j.



Gambar 9.9 Konvolusi diskrit sebagai jumlah salinan filter yang digeser.

Konvolusi dengan Fungsi Berkelanjutan

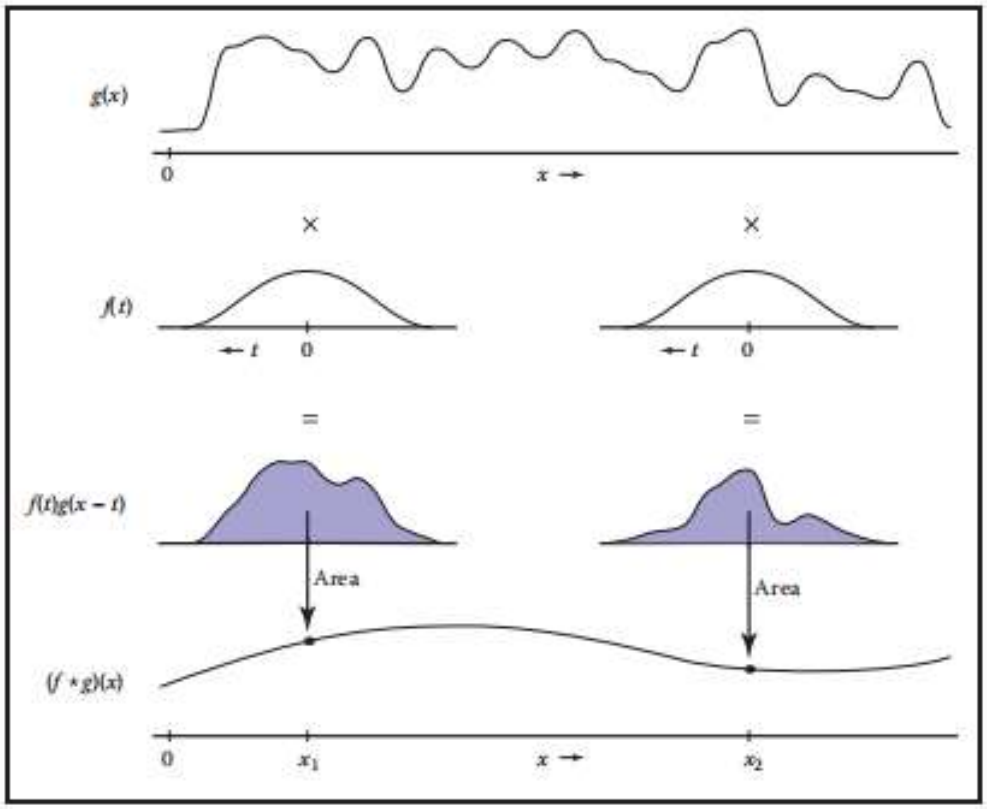
Meskipun benar bahwa urutan diskrit adalah apa yang sebenarnya kita kerjakan dalam program komputer, urutan sampel ini seharusnya mewakili fungsi kontinu, dan sering kali kita perlu alasan secara matematis tentang fungsi kontinu untuk mengetahui apa yang harus

dilakukan. Untuk alasan ini, berguna untuk mendefinisikan konvolusi antara fungsi kontinu dan juga antara fungsi kontinu dan diskrit.

Konvolusi dua fungsi kontinu adalah generalisasi yang jelas dari Persamaan (9.2), dengan integral yang menggantikan jumlah:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)dt$$

Salah satu cara untuk menafsirkan definisi ini adalah bahwa konvolusi off dan g, yang dievaluasi pada argumen x, adalah luas di bawah kurva produk dari dua fungsi setelah kita menggeser g sehingga g(0) sejajar dengan f(t). Sama seperti dalam kasus diskrit, konvolusi adalah Pergerakan rata-rata, dengan filter memberikan bobot rata-rata (lihat Gambar 9.10).



Gambar 9.10 Konvolusi terus menerus.

Seperti konvolusi diskrit, konvolusi fungsi kontinu adalah komutatif dan asosiatif, dan distributif terhadap penjumlahan. Juga seperti kasus diskrit, konvolusi kontinu dapat dilihat sebagai jumlah salinan dari filter daripada perhitungan rata-rata tertimbang. Kecuali, dalam hal ini, ada tak terhingga banyak salinan dari filter g:

$$(f * g) = \int_{-\infty}^{+\infty} f(t)g \to t\, dt.$$

Contoh (Konvolusi dua fungsi kotak).
Misalkan f adalah fungsi kotak:

$$f(x) = \begin{cases} 1 & -\frac{1}{2} \le x < \frac{1}{2} \\ 0 & jika\ tidak \end{cases}$$

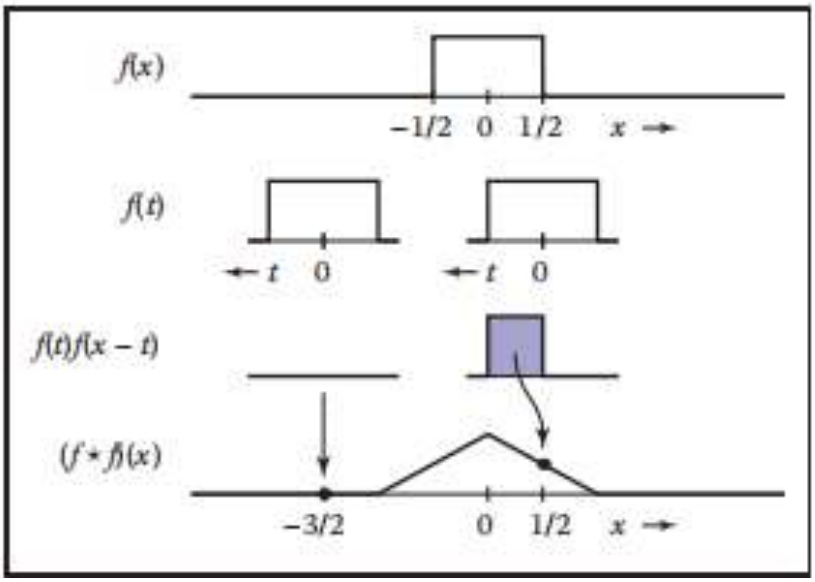
Lalu apa itu f * f? Definisi (Persamaan 9.3) memberikan

$$(f * f)(x) = \int_{-\infty}^{+\infty} f(t)f(x-t)dt$$

Gambar 9.11 menunjukkan dua kasus integral ini. Kedua kotak mungkin memiliki tumpang tindih nol, yang terjadi ketika x ≤ 1 atau x ≥ 1; dalam hal ini hasilnya adalah nol. Ketika −1 < x < 1, tumpang tindih tergantung pada pemisahan antara dua kotak, yaitu|x|; hasilnya adalah 1−|x|. Jadi

$$(f * x)(x) = \begin{cases} 1 - |x| & -1 < x < 1 \\ 0 & jika\ tidak \end{cases}$$

Fungsi ini, yang dikenal sebagai fungsi tent, adalah filter umum lainnya (lihat Bagian 9.3.1).



Gambar 9.11 Menggandakan dua kotak menghasilkan fungsi tent.

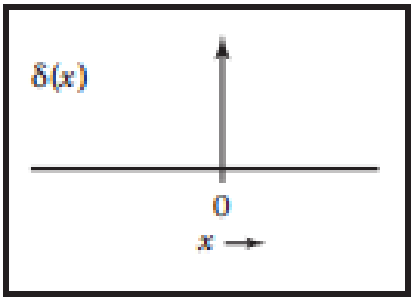
Fungsi Delta Dirac

Dalam konvolusi diskrit, kita melihat bahwa impuls diskrit d bertindak sebagai identitas: d * a = a. Dalam kasus kontinu, ada juga fungsi identitas, yang disebut impuls Dirac atau fungsi delta Dirac, yang dilambangkan δ(x).

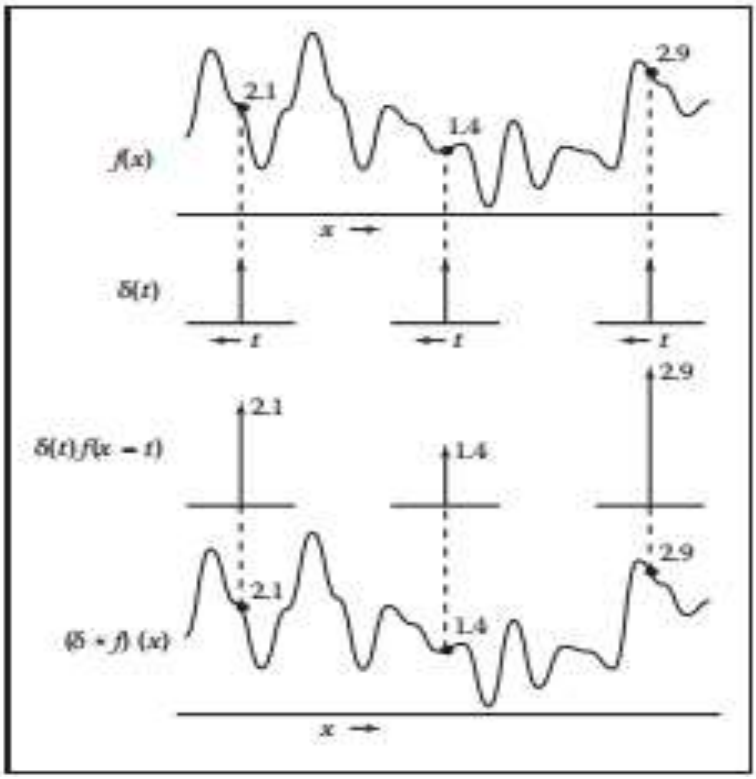
Secara intuitif, fungsi delta adalah paku sangat sempit, sangat tinggi yang memiliki lebar terbatas tetapi masih memiliki luas sama dengan 1 (Gambar 9.12). Sifat kunci dari fungsi delta adalah mengalikannya dengan fungsi akan memilih nilai tepat pada nol:

$$\int_{-\infty}^{+\infty} \delta(x) f(x) dx = f(0)$$

Fungsi delta tidak memiliki nilai yang terdefinisi dengan baik pada 0 (Anda dapat menganggap nilainya secara longgar sebagai +∞), tetapi memiliki nilai (x)=0untuk semua x =0



Gambar 9.12 Fungsi delta Dirac (x).



Gambar 9.13 Mengonversi fungsi dengan δ(x) mengembalikan salinan fungsi yang sama.

Dari sifat pemilihan nilai tunggal ini, maka fungsi delta adalah identitas untuk konvolusi kontinu (Gambar 9.13), karena konvolusi dengan sembarang fungsi f menghasilkan

$$(\delta * \mathrm{f})(\mathrm{x}) = \int_{-\infty}^{\infty} \delta(\mathrm{t})\mathrm{f}(\mathrm{x}-\mathrm{t})\mathrm{dt} = \mathrm{f}(\mathrm{x})$$

Jadi δ * f = f (dan karena komutatifitas δ * f = f juga).

Konvolusi Berkelanjutan Diskrit

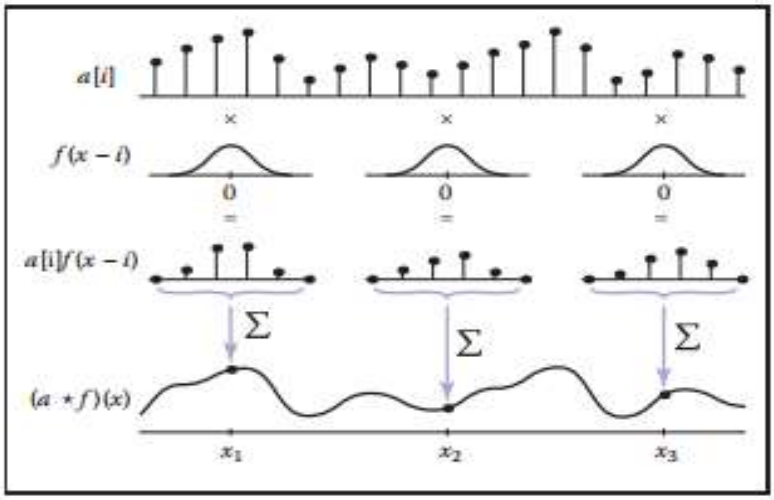
Ada dua cara untuk menghubungkan dunia diskrit dan kontinu. Salah satunya adalah pengambilan sampel: kami mengubah fungsi kontinu menjadi fungsi diskrit dengan menuliskan nilai fungsi pada semua argumen bilangan bulat dan melupakan sisanya. Diberikan fungsi kontinu f(x), kita dapat mengambil sampel untuk mengubahnya menjadi barisan diskrita[i]:

$$a[i] = f(i).$$

Pergi ke arah lain, dari fungsi diskrit, atau urutan, ke fungsi kontinu, disebut rekonstruksi. Ini dicapai dengan menggunakan bentuk lain dari konvolusi, bentuk kontinu-diskrit. Dalam hal ini, kami memfilter barisan diskrit a[i] dengan filter kontinu f(x):

$$(a * f)(x) = \sum_i a[i] f(x-1)$$

Nilai dari fungsi yang dibangun di atas merupakan penjumlahan terbobot dari sampel a[i]untuk nilai yang mendekati murni(Gambar 9.14). Bobot berasal dari filter, yang dievaluasi pada sekumpulan titik yang berjarak satu unit. Misalnya, jika x = 5 .3 dan f memiliki jari-jari 2, f dievaluasi pada 1.3, 0.3, −0.7, dan −1.7. Perhatikan bahwa untuk konvolusi kontinu diskrit kita biasanya menulis barisan ini terlebih dahulu dengan filter sekon, sehingga jumlahnya melebihi bilangan bulat.



Gambar 9.14 Konvolusi diskrit-kontinyu.

Seperti halnya konvolusi diskrit, kita dapat memberi batas pada jumlah jika kita mengetahui jari-jari filter, r, menghilangkan semua titik di mana perbedaan antara x dan i setidaknya r:

$$(a * f)(x) = \sum_{i=\lceil x-r \rceil}^{\lceil x-r \rceil} a[i]f(x-i)$$

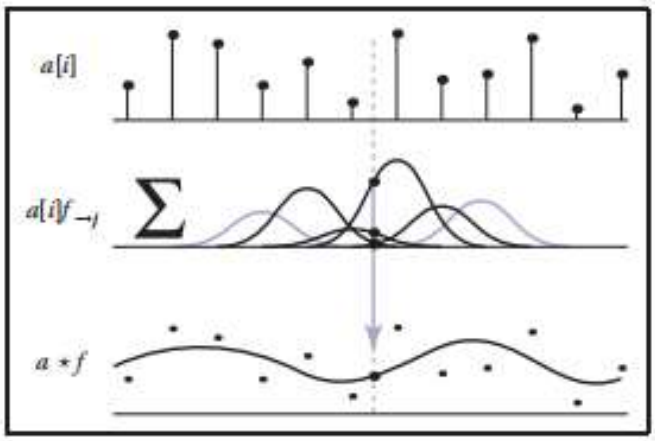
Perhatikan, bahwa jika sebuah titik jatuh tepat pada jarak r dari x (yaitu, jika x − r ternyata bilangan bulat), itu akan ditinggalkan dari jumlah. Ini berbeda dengan kasus diskrit, di mana kami memasukkan titik di I − r.

Dinyatakan dalam kode, ini adalah:

```
function reconstruct(sequence a, filter f, real x)
s = 0
r = f.radius
for i = x − r to x + r do
s = s + a[i]f(x − i)
return s
```

Seperti bentuk konvolusi lainnya, konvolusi kontinu diskrit dapat dilihat sebagai penjumlahan salinan bergeser dari filter (Gambar 9.15):

$$(a * f) = \sum_{i} a[i]f \to i$$



Gambar 9.15 Rekonstruksi (konvolusi kontinu-diskrit) sebagai jumlah salinan filter yang digeser.

Konvolusi kontinu-diskrit terkait erat dengan splines. Untuk spline yang seragam (misalnya, B-spline yang seragam), kurva parameter untuk spline adalah persis konvolusi dari fungsi dasar spline dengan urutan titik kontrol (lihat Bagian 15.6.2).

Konvolusi di Lebih dari Satu Dimensi

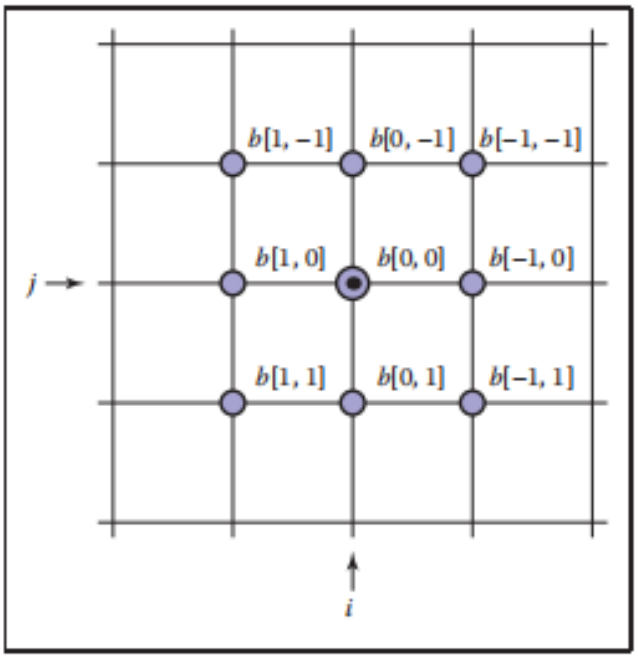
Sejauh ini, semua yang telah kami katakan tentang pengambilan sampel dan rekonstruksi adalah satu dimensi: ada variabel tunggal x atau indeks urutan tunggal i. Namun, banyak aplikasi penting dari pengambilan sampel dan rekonstruksi dalam grafis, diterapkan pada fungsi dua dimensi—khususnya, pada gambar 2D. Untungnya, generalisasi dari algoritma dan teori sampling dari 1D ke 2D, 3D, dan seterusnya secara konseptual sangat sederhana. Dimulai dengan definisi konvolusi diskrit, kita dapat menggeneralisasikannya ke dua dimensi dengan membuat jumlah menjadi jumlah ganda:

$$(a * b)[i,j] = \sum_{i'} \sum_{j'} a[i',j']b[i' - j', \quad j - j']$$

Jika b adalah filter berjari-jari r yang didukung terbatas (yaitu, memiliki $(2r + 1)^2$ nilai), maka kita dapat menulis jumlah ini dengan batas (Gambar 9.16):

$$(a * b)[i,j] = \sum_{i'=i-r}^{i+r} \sum_{j'=j-r}^{j+r} a[i',j']b[i - i', j - j']$$

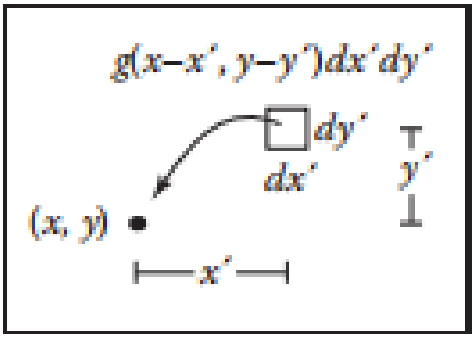
dan ekspresikan dalam kode:



Gambar 9.16 Bobot untuk sembilan sampel input yang berkontribusi pada konvolusi diskrit di titik (i, j) dengan filter b berjari-jari 1.

```
function convolve2d(sequence2d a, fifilter2d b, int i, int j)
s = 0
r = b.radius
for i = i − r to i + r do
for j = j − r to j + r do
s = s + a[i ][j ]b[i − i ][j − j ]
return s
```

Definisi ini dapat diinterpretasikan dengan cara yang sama seperti dalam kasus 1D: setiap sampel output adalah rata-rata tertimbang dari suatu area di input, menggunakan filter 2D sebagai "topeng" untuk menentukan bobot setiap sampel dalam rata-rata.



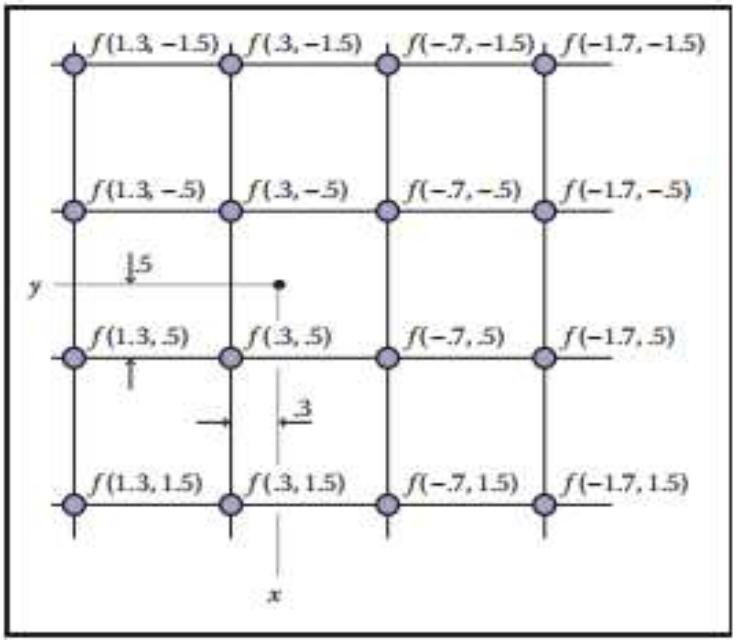
Gambar 9.17 Bobot untuk area kecil tak terhingga dalam sinyal input yang dihasilkan dari konvolusi kontinu di (x, y).

Melanjutkan generalisasi, kita juga dapat menulis konvolusi kontinu-kontinyu (Gambar 9.17) dan konvolusi diskrit-kontinu (Gambar 9.18) dalam 2D:

$$(f * g)(x,y) = \int\int (x',y')\, g(x-x', y-y')dx'dy'$$

$$(a * f)(x,y) = \sum_i \sum_j a[i,j] f(x-i, y-j)$$

Dalam setiap kasus, terdapat titik tertentu yang merupakan rata-rata tertimbang dari input yang mendekati titik tersebut. Untuk kasus kontinu-kontinu, ini adalah integral berbobot di atas daerah yang berpusat di titik itu, dan dalam kasus kontinu-diskrit itu adalah rata-rata tertimbang dari semua sampel yang jatuh di dekat titik tersebut.



Gambar 9.18 Bobot untuk 16 sampel input yang berkontribusi pada konvolusi kontinu diskrit di titik (x, y) untuk filter rekonstruksi radius 2.

Setelah kita beralih dari 1D ke 2D, seharusnya cukup jelas bagaimana menggeneralisasi lebih jauh ke 3D atau bahkan ke dimensi yang lebih tinggi.

9.4 PEMROSESAN SINYAL UNTUK GAMBAR

Sekarang setelah kita memiliki mesin konvolusi, mari kita periksa beberapa filter tertentu yang biasa digunakan dalam grafis

Masing-masing filter berikut memiliki radius alami, yang merupakan ukuran default yang akan digunakan untuk pengambilan sampel atau rekonstruksi ketika sampel dipisahkan satu unit. Dalam bagian ini, filter didefinisikan pada ukuran alami ini: misalnya, filter kotak memiliki radius alami 1/2, dan filter kubik memiliki radius alami 2. Kami juga mengatur agar setiap filter berintegrasi ke $\int_{x=0}^{\infty} f(x)dx = 1$, seperti yang diperlukan untuk pengambilan sampel dan rekonstruksi tanpa mengubah nilai rata-rata sinyal. Seperti yang akan kita lihat di Bagian 9.4.3, beberapa aplikasi memerlukan filter dengan ukuran berbeda, yang dapat diperoleh dengan menskalakan filter dasar. Untuk filter f(x), kita dapat mendefinisikan versi skala s:

$$f_s(x) = \frac{f(x)/s)}{s}$$

Saringan diregangkan secara horizontal dengan faktor s, dan kemudian diremas secara vertikal oleh faktor 1 s sehingga luasnya tidak berubah. Sebuah filter yang memiliki jari-jari alami r dan digunakan pada skala s memiliki jari-jari tumpuan sr (lihat Gambar 9.20 di bawah).

Galeri Filter Konvolusi

Filter Kotak

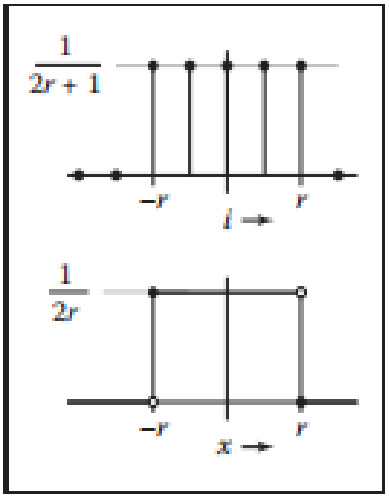
Filter kotak (Gambar 9.19) adalah fungsi konstanta sepotong-sepotong yang integralnya sama dengan satu. Sebagai filter diskrit, dapat ditulis sebagai

$$a_{box} = \begin{cases} \dfrac{1}{(2r+1)} & |i| \le r \\ 0 & Jika\ tidak \end{cases}$$

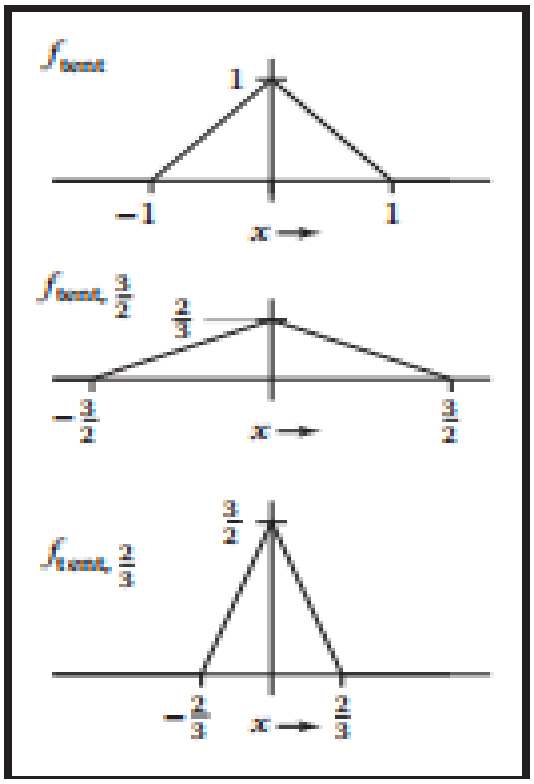
Perhatikan bahwa untuk simetri kami menyertakan kedua titik akhir. Sebagai filter kontinu, kami menulis

$$f_{box}(x) = \begin{cases} \dfrac{1}{2r} & -r \le x < r \\ 0 & Jika\ tidak \end{cases}$$

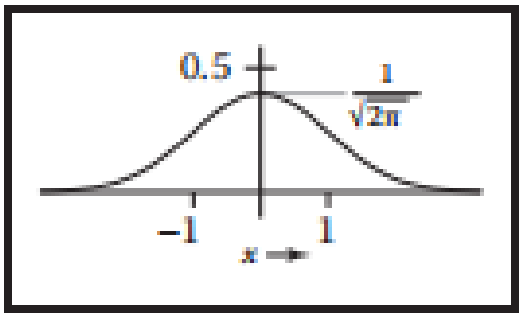
Dalam hal ini, kami mengecualikan satu titik akhir, yang membuat kotak dengan radius 0,5 dapat digunakan sebagai filter rekonstruksi. Karena filter kotak tidak kontinu maka kasus batas ini penting, dan untuk filter khusus ini kita perlu memperhatikannya. Kami menulis hanya fbox untuk jari-jari alami r = 1 2.



Gambar 9.19 diskrit



Gambar 9.20 Filter tent dan dua versi skala.



Gambar 9.21 Filter Gauss.

Filter Tent

Tent, atau filter linier (Gambar 9.20), adalah fungsi linier yang kontinu dan sepotong-sepotong:

$$f^{tent}(x)\begin{cases}1-|x| & |x|<1\\ 0 & jika\ tidak\end{cases}$$

Jari-jari alaminya adalah 1. Untuk filter, seperti ini, yang paling sedikit C^0 (yaitu, tidak ada lompatan nilai yang tiba-tiba, seperti halnya dengan kotak), kita tidak perlu lagi memisahkan definisi diskrit dan filter kontinu: filter diskrit hanyalah filter kontinu yang diambil sampelnya pada bilangan bulat.

Filter Gaussian

Fungsi Gaussian (Gambar 9.21), juga dikenal sebagai distribusi normal, merupakan filter penting secara teoritis dan praktis. Kita akan melihat lebih banyak properti khususnya saat bab ini berlanjut:

$$f_g,\sigma(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-x^2/-\sigma^2}$$

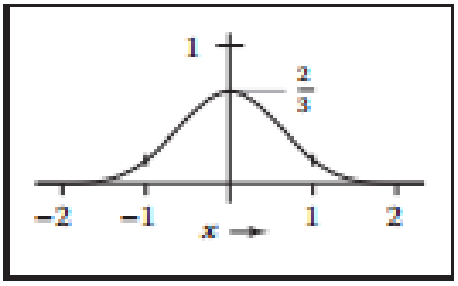
Parameter σ disebut simpangan baku. Gaussian membuat filter pengambilan sampel yang baik karena sangat halus; kami akan membuat pernyataan ini lebih tepat nanti di bab ini. Filter Gaussian tidak memiliki radius natural tertentu; ini adalah filter pengambilan sampel yang berguna untuk rentang σ. Gaussian juga tidak memiliki radius dukungan yang terbatas, meskipun karena peluruhan eksponensial, nilainya dengan cepat menjadi cukup kecil untuk diabaikan. Bila perlu, kita dapat memangkas ekor dari fungsi dengan menyetelnya ke nol di luar radius r, menghasilkan Gaussian yang dipangkas. Ini berarti bahwa lebar filter dan jari-jari alami dapat bervariasi tergantung pada aplikasinya, dan Gaussian terpangkas yang diskalakan oleh s sama dengan Gaussian terpangkas tanpa skala dengan deviasi standar sσ dan radius sr. Cara terbaik untuk menangani ini dalam praktiknya adalah membiarkan σ dan r menjadi properti dari filter, tetap ketika filter ditentukan, dan kemudian menskalakan filter seperti yang lain ketika diterapkan.

Filter Kubik B-Spline

Banyak filter didefinisikan sebagai polinomial sepotong-sepotong, dan filter kubik dengan empat bagian (radius natural2) sering digunakan sebagai filter rekonstruksi. Salah satu filter tersebut dikenal sebagai filter B-spline (Gambar 9.22) karena asalnya sebagai fungsi pencampuran untuk kurva spline (lihat Bab 15):

$$F_B(x) = \frac{1}{6}\begin{cases} -3(1-|x|)^3 3(1|x|)^2 + 3(1-|x|) + 1 & -1 \le x \le 1 \\ (2-|x|)^2 & 1 \le |x| \le 2 \\ 0 & jika\ tidak \end{cases}$$

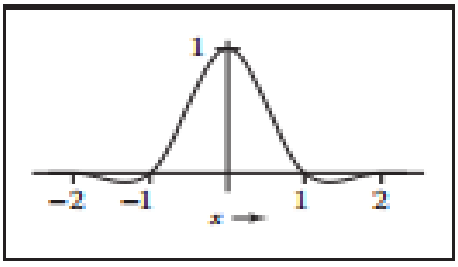
Di antara kubik piecewise, B-spline istimewa karena memiliki turunan pertama dan kedua yang kontinu—yaitu, C^2. Cara yang lebih ringkas untuk mendefinisikan filter ini adalah $f_B = f_{box} * f_{box} * f_{box} * f_{box}$; membuktikan bahwa bentuk yang lebih panjang di atas setara adalah latihan yang bagus dalam konvolusi (lihat Latihan 3).



Gambar 9.22 Filter B-spline.

Filter Kubik Catmull-Rom

Filter kubik per potong lainnya yang dinamai untuk spline, Catmull-Romfilter (Gambar 9.23), memiliki nilai nol pada x =−2,−1, 1, dan2, yang berarti akan menginterpolasi sampel saat digunakan sebagai filter rekonstruksi (Bagian 9.3.2):



Gambar 9.23 Filter CatmullRom.

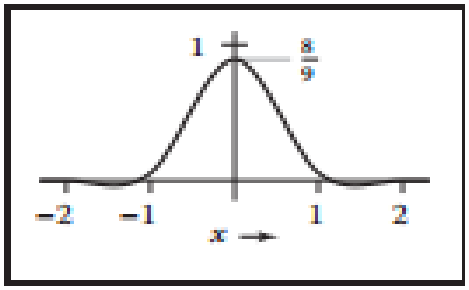
$$F_C(x) = \frac{1}{2}\begin{cases} -3(1-|x|)^3 4(1|x|)^2 + (1-|x|) + 1 & -1 \le x \le 1 \\ (2-|x|)^3 - (2-|x|)^2 & 1 \le |x| \le 2 \\ 0 & jika\ tidak \end{cases}$$

Filter Mitchell-Netravali Cubic

Untuk aplikasi yang sangat penting dari gambar resampling, Mitchell dan Netravali (Mitchell & Netravali, 1988) membuat studi tentang filter kubik dan merekomendasikan satu bagian antara dua filter sebelumnya sebagai pilihan terbaik (Gambar 9.24). Ini hanyalah kombinasi berbobot dari dua filter sebelumnya:

$$F_M(x) = \frac{1}{3}F_B(x) + \frac{2}{3}F_C(x)$$

$$= \frac{1}{8}\begin{cases} -21(1-|x|)^3 + 27(1|x|)^2 + 9(1-|x|) + 1 & -1 \le x \le 1 \\ 7(2-|x|)^3 - 6(2-|x|)^2 & 1 \le |x| \le 2 \\ 0 & jika\ tidak \end{cases}$$

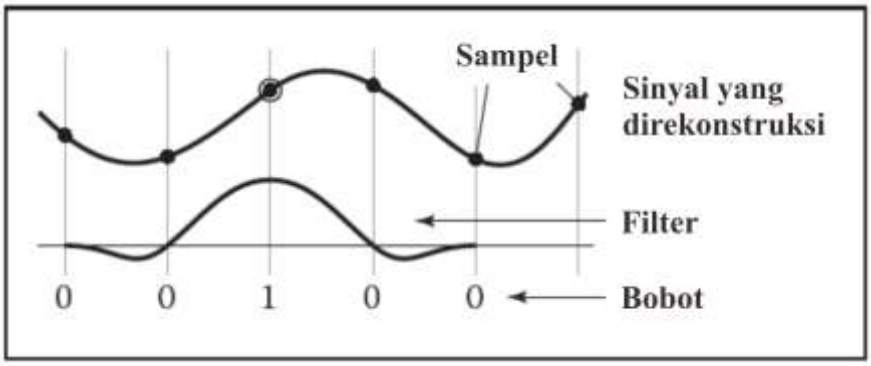


Gambar 9.24 Filter Mitchell Netravali.

Sifat Filter

Filter memiliki beberapa terminologi tradisional yang menyertainya, yang kami gunakan untuk menggambarkan filter dan membandingkannya satu sama lain. Respon impuls dari filter hanyalah nama lain untuk fungsi: itu adalah respons filter terhadap sinyal yang hanya berisi impuls (dan ingat bahwa menggulung dengan impuls hanya mengembalikan filter).

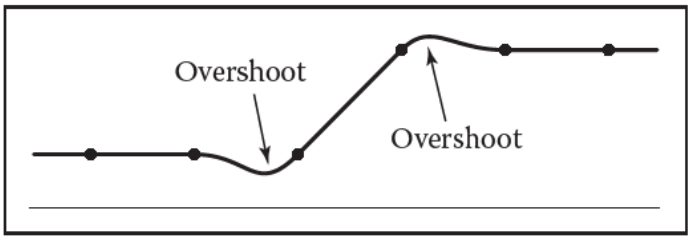
Filter kontinu diinterpolasi jika, ketika digunakan untuk merekonstruksi fungsi kontinu dari urutan diskrit, fungsi yang dihasilkan mengambil nilai sampel secara tepat pada titik sampel—yaitu, "menghubungkan titik-titik" daripada menghasilkan fungsi yang hanya mendekati titik-titik. Filter interpolasi adalah filter f yang f(0) = 1 dan f(i)=0untuk semua bilangan bulat bukan nol i (Gambar 9.25).



Gambar 9.25 Filter interpolasi merekonstruksi titik sampel tepat karena memiliki nilai nol pada semua offset bilangan bulat bukan nol dari pusat.

Filter yang mengambil nilai negatif memiliki dering atau overshoot: filter akan menghasilkan osilasi ekstra dalam nilai di sekitar perubahan tajam dalam nilai fungsi yang difilter. Misalnya, filter Catmull-Rom memiliki lobus negatif di kedua sisi, dan jika Anda

memfilter fungsi langkah dengannya, itu akan sedikit membesar-besarkan langkah, menghasilkan nilai fungsi yang undershoot0 dan overshoot1 (Gambar 9.26).



Gambar 9.26 Filter dengan lobus negatif akan selalu menghasilkan beberapa overshoot saat menyaring atau merekonstruksi diskontinuitas yang tajam.

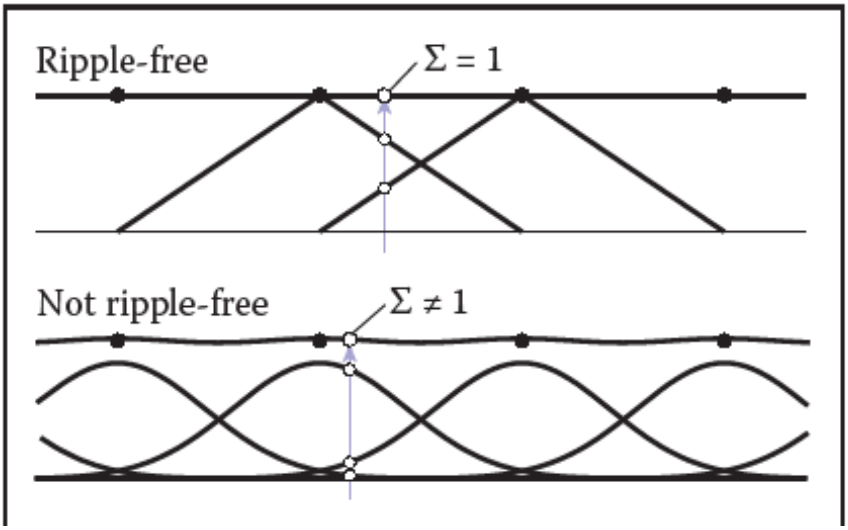
Filter kontinu adalah bebas riak jika, ketika digunakan sebagai filter rekonstruksi, filter tersebut akan merekonstruksi urutan konstan sebagai fungsi konstan (Gambar 9.27). Ini ekuivalen dengan persyaratan bahwa filter sumto satu pada setiap kotak bilangan bulat:

$$\sum_i f(x+i) = 1 \quad untuk\ semua\ x$$

Semua filter dalam Bagian 9.3.1 bebas riak pada jari-jari alaminya, kecuali Gaussian, tetapi tidak satupun dari filter tersebut harus bebas riak ketika digunakan pada skala noninteger. Jika perlu untuk menghilangkan riak dalam konvolusi kontinu diskrit, mudah untuk melakukannya: bagi setiap sampel yang dihitung dengan jumlah bobot yang digunakan untuk menghitungnya:

$$\frac{}{(a * f)(x)} = \frac{\sum_i a[i] f[x-1]}{\sum_i a[i]}$$

Ekspresi ini masih dapat diinterpretasikan sebagai konvolusi antara a dan filter f (lihat Latihan 6).



Gambar 9.27 Filter tent radius 1 adalah filter rekonstruksi bebas riak; filter Gaussian dengan standar deviasi 1/2 tidak.

Filter kontinu memiliki derajat kontinuitas, yang merupakan turunan orde tertinggi yang didefinisikan di mana-mana. Filter, seperti filter kotak, yang nilainya melonjak tiba-tiba tidak kontinu sama sekali. Sebuah filter yang kontinu tetapi memiliki sudut tajam (diskontinuitas pada turunan pertama), seperti filter tent, memiliki urutan kontinuitas nol, dan kami katakan itu adalah C^0. Filter yang memiliki turunan kontinu (tidak ada sudut tajam), seperti filter kubik sepotong-sepotong di bagian sebelumnya, adalah C^1; jika turunan kedua

juga kontinu, seperti halnya filter B-spline, itu adalah C^2. Urutan kontinuitas filter sangat penting untuk filter rekonstruksi karena fungsi yang direkonstruksi mewarisi kontinuitas filter.

Filter yang Dapat Dipisahkan

Sejauh ini kita hanya membahas filter untuk konvolusi 1D, tetapi untuk gambar dan sinyal multidimensi lainnya kita juga membutuhkan filter. Secara umum, setiap fungsi 2D bisa menjadi filter 2D, dan kadang-kadang berguna untuk mendefinisikannya dengan cara ini. Namun, dalam banyak kasus, kita dapat membuat filter 2D (atau dimensi lebih tinggi) yang sesuai dari filter 1D yang telah kita lihat.

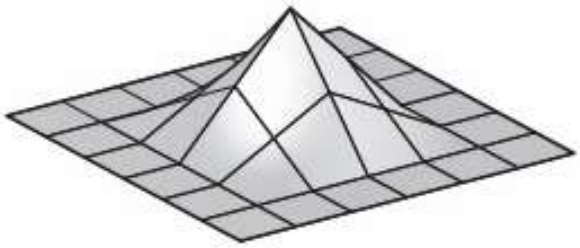
Cara yang paling berguna untuk melakukan ini adalah dengan menggunakan filter yang dapat dipisahkan. Nilai filter yang dapat dipisahkan $f_2(x, y)$ pada x dan y tertentu hanyalah produk dari f_1 (filter 1D) yang dievaluasi pada x dan pada y:

$$f2(x, y) = f1(x)f1(y)$$

Demikian pula, untuk filter diskrit,

$$b2[i, j] = b1[i]b1[j]$$

Setiap irisan horizontal atau vertikal melalui f_2 adalah salinan skala dari f_1. Integral f_2 adalah kuadrat dari integral f_1, jadi khususnya jika f_1 dinormalisasi, maka f_2 juga demikian.



Gambar 9.28 Filter tent 2D yang dapat dipisahkan.

Contoh (Filter tent yang dapat dipisahkan).

Jika kita memilih fungsi tent untuk f1, hasil fungsi linier sepotong-sepotong (Gambar 9.28) adalah

$$f_{2,tent}(x,y) = \begin{cases} 1-|x|)(1-|y|) & |x|<1 \quad dan\ |y|<1 \\ 0 & jika\ tidak \end{cases}$$

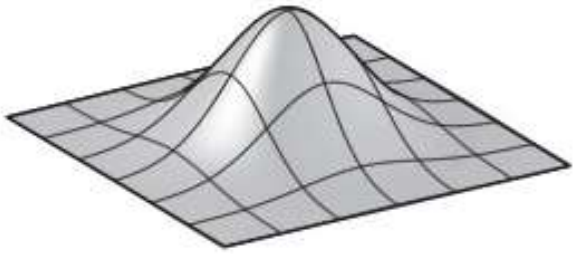
Profil sepanjang sumbu koordinat adalah fungsi tent, tetapi profil sepanjang diagonal adalah kuadrat (misalnya, sepanjang garis x = y di kuadran positif, kita melihat fungsi kuadrat $(1 - x)^2$).

Contoh (Filter Gaussian 2D).

Jika kita memilih fungsi Gaussian untuk f_1, hasil fungsi 2D (Gambar 9.29) adalah

$$\begin{aligned} f_{2,g}(x,y) &= \frac{1}{2\pi}(e^{\frac{-x^2}{2}}e^{\frac{-y^2}{2}}) \\ &= \frac{1}{2\pi}(e^{\frac{-(x^2+y^2}{2}}) \\ &= \frac{1}{2\pi}e^{-r^2/2} \end{aligned}$$

Perhatikan bahwa ini (hingga faktor skala) fungsi yang sama yang akan kita dapatkan jika kita memutar Gaussian 1D di sekitar titik asal untuk menghasilkan fungsi simetris sirkular. Sifat simetri sirkular dan dapat dipisahkan pada saat yang sama adalah unik untuk fungsi Gaussian. Profil di sepanjang sumbu koordinat adalah Gauss, tetapi begitu juga profil di sepanjang arah mana pun pada offset mana pun dari pusat.



Gambar 9.29 Filter Gaussian 2D, yang dapat dipisahkan dan simetri radial.

Keuntungan utama dari filter yang dapat dipisahkan dibandingkan filter 2D lainnya berkaitan dengan efisiensi dalam implementasi. Mari kita substitusikan definisi a_2 ke dalam definisi konvolusi diskrit:

$$(a * b_2)[i,j] = \sum_{i\prime} \sum_{j\prime} a[i',j']b_1[i-i']b_1[j-j']$$

Perhatikan bahwa b_1[i- i'] tidak bergantung pada j' dan dapat difaktorkan dari jumlah dalam:

$$= \sum_{i\prime} b_1[i-i'] \sum_{i\prime} a[i'-j'']b_1[j-j']$$

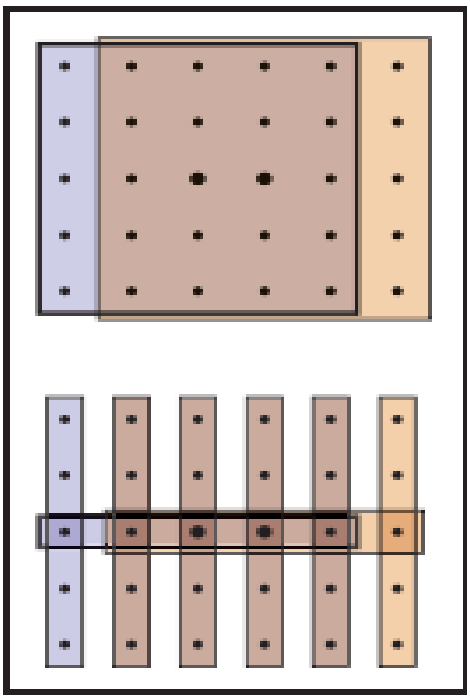
Mari kita menyingkat jumlah dalam sebagai S[i]:

$$S[i'] = \sum_{i\prime} a\big[i'-j''\big]b_1[j-j']$$

$$(a * b_1)[i,j] = \sum_{i\prime} b_1[i-i']S[i']$$

Dengan persamaan dalam formulir ini, pertama-tama kita dapat menghitung dan menyimpan S[i] untuk setiap nilai dari i , dan kemudian menghitung penjumlahan luar menggunakan nilai-nilai yang tersimpan. Sepintas hal ini tidak tampak luar biasa, karena kita masih harus melakukan pekerjaan proporsional $(2r + 1)^2$ untuk menghitung semua jumlah bagian dalam. Namun, agak berbeda jika kita ingin menghitung nilai di banyak titik [i, j].

Misalkan kita perlu menghitung a*b_2 pada [2, 2] dan [3, 2], dan b_1 memiliki radius 2. Dengan memeriksa Persamaan 9.5, kita dapat melihat bahwa kita membutuhkan S[0], ..., S[4] untuk menghitung hasil pada [2,2], dan kita akan membutuhkan S[1], ..., S[5] untuk menghitung hasil pada [3, 2]. Jadi, dalam formulasi yang dapat dipisahkan ini, kita hanya dapat menghitung enam nilai S dan berbagi S[1], ..., S[4] (Gambar 9.30).



Gambar 9.30 Menghitung dua titik output menggunakan larik 2D terpisah dari 25 sampel (di atas) vs. pemfilteran satu kali di sepanjang kolom, kemudian menggunakan 1Darraysoffivesampel terpisah (di bawah).

Penghematan ini sangat penting untuk filter besar. Memfilter gambar dengan filter berjari-jari r dalam kasus umum membutuhkan perhitungan $(2r + 1)^2$ produk per piksel, sedangkan memfilter gambar dengan filter yang dapat dipisahkan dari ukuran yang sama membutuhkan 2(2r + 1) produk (dengan mengorbankan beberapa penyimpanan perantara). Perubahan kompleksitas asimtotik dari $O(r^2)$ ke O(r) memungkinkan penggunaan filter yang jauh lebih besar.

Algoritmanya adalah:

```
function fifilterImage(image I, fifilter b)
r = b.radius
nx = I.width
ny = I.height
mengalokasikan array penyimpanan S[0…nx – 1
allocate image Iout[r ... nx − r – 1, r…ny – r – 1]
 initialize S and Iout to all zero
for j = r to ny − r − 1 do
for i = 0 to nx − 1 do
S[i ]=0
for j = j − r to j + r do
S[i ] = S[i ] + I[i , j ]b[j − j ]
for i = r to nx − r − 1 do
for i = i − r ke i + r do
Iout[i, j] = Iout[i, j] + S[i ]b[i − i ]
return Iout
```

Untuk kesederhanaan, fungsi ini menghindari semua pertanyaan tentang batas dengan memangkas r piksel dari keempat sisi gambar output. Dalam prakteknya ada berbagai cara untuk menangani perbatasan; lihat Bagian 9.4.3.

Pemrosesan Sinyal untuk Gambar

Kita telah membahas pengambilan sampel, penyaringan, dan rekonstruksi secara abstrak sejauh ini, menggunakan sebagian besar sinyal 1D sebagai contoh. Tetapi seperti yang kami amati di awal bab, aplikasi pemrosesan sinyal yang paling penting dan paling umum dalam grafis adalah untuk gambar sampel. Mari kita perhatikan baik-baik bagaimana semua ini berlaku untuk gambar.

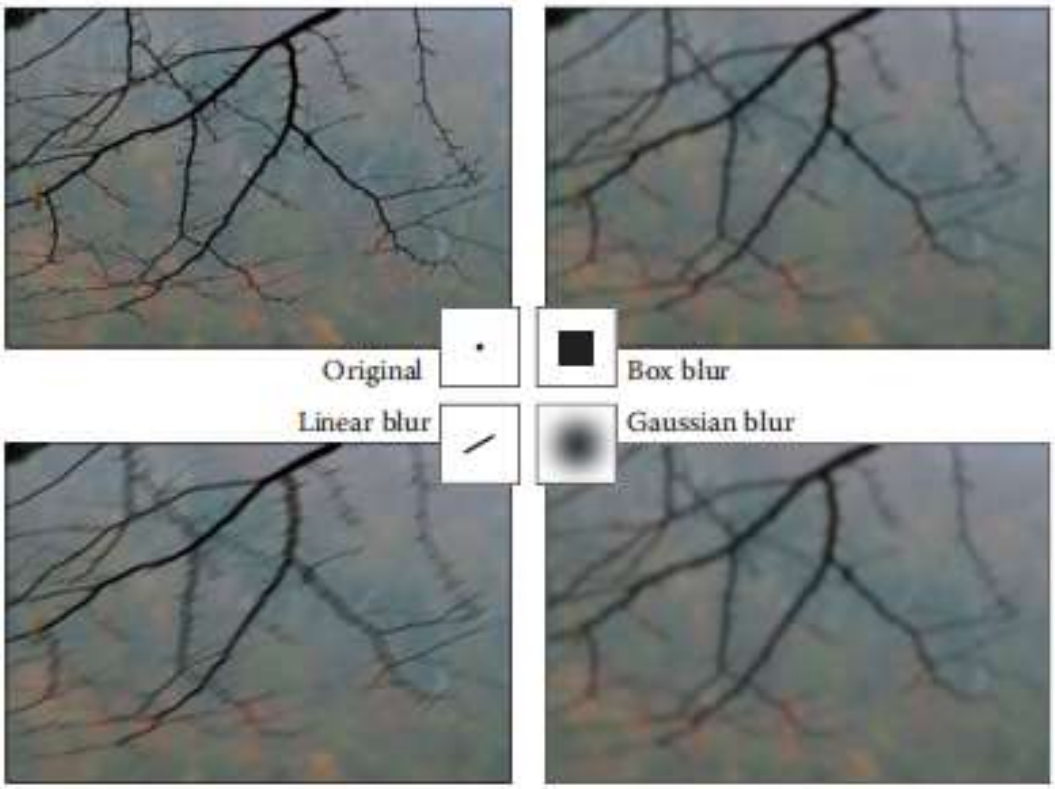
Pemfilteran Gambar Menggunakan Filter Diskrit

Mungkin aplikasi konvolusi yang paling sederhana adalah memproses gambar menggunakan konvolusi diskrit. Beberapa fitur program manipulasi gambar yang paling banyak digunakan adalah filter konvolusi sederhana. Kekaburan gambar dapat dicapai dengan menggabungkan banyak filter lowpass yang umum, mulai dari kotak hingga Gaussian (Gambar 9.31). Filter Gaussian menciptakan keburaman yang tampak sangat halus dan biasanya digunakan untuk tujuan ini.

Kebalikan dari blurring adalah sharpening, dan salah satu cara untuk melakukannya adalah dengan menggunakan prosedur "unsharp mask": kurangi sebagian α gambar buram dari aslinya. Dengan scalling ulang untuk menghindari perubahan kecerahan secara keseluruhan, kami memiliki

$$I\text{sharp} = (1 + \alpha)I - \alpha(Ifg,\sigma)$$
$$= I * (1 + \alpha)d - \alpha fg,\sigma$$
$$= I * \text{sharp}(\sigma, \alpha),$$

di mana $f_{g,\sigma}$ adalah filter Gaussian dengan lebar . Dengan menggunakan dorongan diskrit d dan sifat distributif konvolusi, kami dapat menulis seluruh proses ini sebagai filter tunggal yang bergantung pada lebar keburaman dan derajat sharpening (Gambar 9.32).



Gambar 9.31 Mengaburkan gambar dengan konvolusi dengan masing-masing dari tiga filter yang berbeda.

Gambar 9.32 Mempertajam gambar menggunakan filter konvolusi.

Gambar 9.33 Sebuah bayangan yang lembut.

Contoh lain dari menggabungkan dua buah drop shadow diskrit. Biasanya mengambil salinan garis objek yang kabur dan digeser untuk membuat bayangan jatuh yang lembut (Gambar 9.33). Kita dapat menyatakan operasi shifting sebagai konvolusi dengan impuls di luar pusat:

$$d_{m,n}(i,j) = \begin{cases} 1 & i = m \; dan \; j = n \\ 0 & jika \; tidak \end{cases}$$

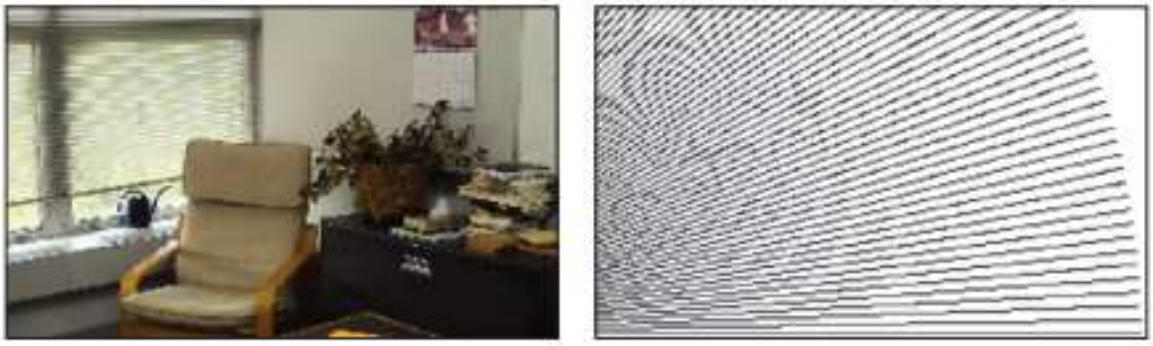
Shifting, lalu blurring, dicapai dengan menggabungkan dengan kedua filter:

Ishadow = (I*dm,n)* fg,σ
= I* (dm,n * fg,σ)
= I*$f$shadow(m, n, σ).

Di sini kita telah menggunakan associativity untuk mengelompokkan dua operasi menjadi satu filter dengan tiga parameter.

Antialiasing dalam Pengambilan Sampel Gambar

Dalam sintesis gambar, kita sering memiliki tugas untuk menghasilkan representasi sampel dari gambar yang memiliki rumus matematika kontinu (atau setidaknya prosedur yang dapat kita gunakan untuk menghitung warna pada titik mana pun, tidak hanya pada posisi piksel bilangan bulat). Penelusuran sinar adalah contoh umum; lebih lanjut tentang ray tracing dan metode khusus untuk antialiasing ada di Bab 4. Dalam bahasa pemrosesan sinyal, kami memiliki sinyal 2D kontinu (gambar) yang perlu kami sampel pada kisi 2D biasa. Jika kita melanjutkan dan mengambil sampel gambar tanpa tindakan khusus, hasilnya akan menunjukkan berbagai artifak aliasing (Gambar 9.34). Pada bagian tepi yang tajam pada gambar, kita melihat artefak anak tangga yang dikenal sebagai "jaggies". Di daerah di mana ada pola berulang, kita melihat pita lebar yang dikenal sebagai pola moir´e.



Gambar 9.34 Dua artefak aliasing dalam gambar: pola moir dalam tekstur periodik (kiri), dan "jaggies" pada garis lurus (kanan).

Gambar 9.35 Perbandingan tiga filter pengambilan sampel berbeda yang digunakan untuk antialias sebagai gambar uji sulit yang berisi lingkaran yang diberi jarak semakin dekat saat semakin besar.

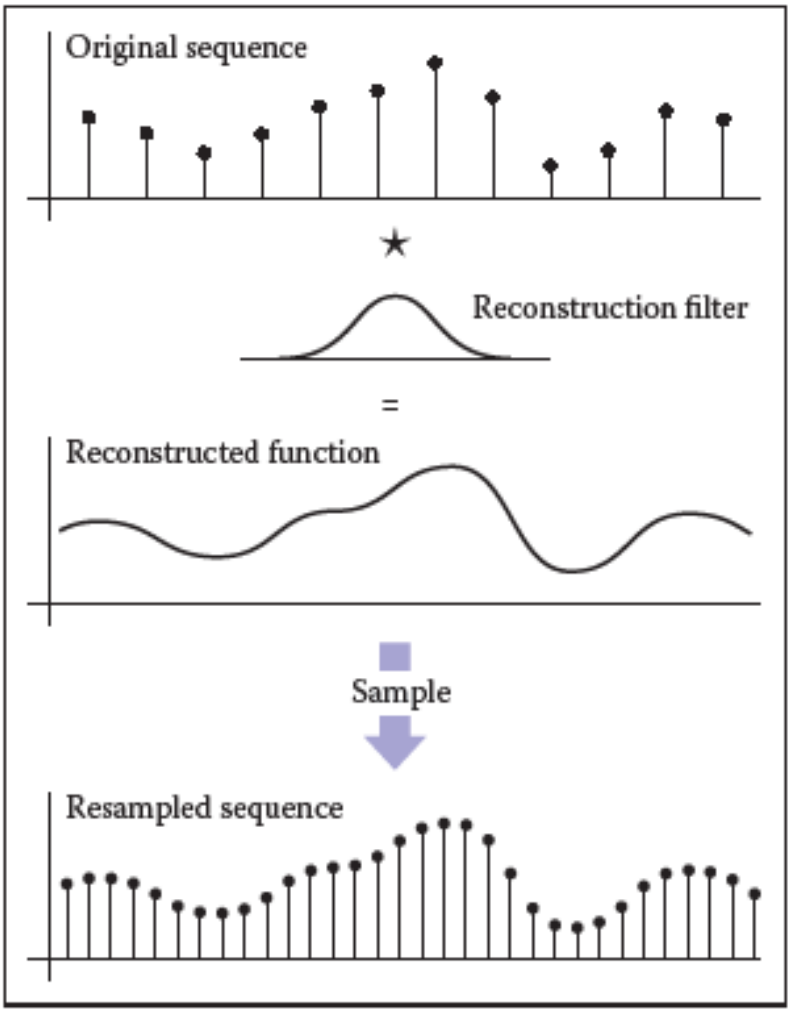
Masalahnya di sini adalah bahwa gambar mengandung terlalu banyak fitur skala kecil; kita perlu menghaluskannya dengan menyaringnya sebelum pengambilan sampel. Melihat kembali definisi konvolusi kontinu dalam Persamaan (9.3), kita perlu merata-ratakan gambar pada area di sekitar lokasi piksel, daripada hanya mengambil nilai pada satu titik. Metode khusus untuk melakukan ini dibahas dalam Bab 4. Filter sederhana seperti kotak akan meningkatkan tampilan tepi yang tajam, tetapi masih menghasilkan beberapa pola moire´ (Gambar 9.35). Filter Gaussian, yang sangat halus, jauh lebih efektif melawan pola moire´, dengan mengorbankan keseluruhan yang agak lebih kabur. Dua contoh ini menggambarkan pertukaran antara ketajaman dan aliasing yang mendasar untuk memilih filter antialiasing.

Rekonstruksi dan Pengambilan Sampel Ulang

Salah satu operasi gambar yang paling umum di mana pemfilteran yang cermat sangat penting adalah pengambilan sampel ulang—mengubah laju sampel, atau mengubah ukuran gambar. Misalkan kita telah mengambil gambar dengan kamera digital yang berukuran 3000 x 2000 piksel, dan kita ingin menampilkannya pada monitor yang hanya berukuran 1280 x 1024 piksel. Untuk membuatnya pas, sambil mempertahankan rasio aspek 3:2, kita perlu membuat sampel ulang menjadi 1278 kali 852 piksel. Bagaimana kita harus pergi tentang ini? Salah satu cara untuk mengatasi masalah ini adalah dengan menganggap proses sebagai piksel yang jatuh: rasio ukuran antara 2 dan 3, jadi kita harus menghapus satu atau dua piksel di antara piksel yang kita simpan. Anda dapat mengecilkan gambar dengan cara ini, tetapi kualitas hasilnya rendah—gambar pada Gambar 9.34 dibuat menggunakan penurunan piksel.

Namun, penurunan piksel sangat cepat, dan merupakan pilihan yang masuk akal untuk membuat pratinjau gambar yang diubah ukurannya selama manipulasi interaktif.

Cara berpikir tentang mengubah ukuran gambar adalah sebagai operasi resampling: kami ingin satu set sampel gambar pada kisi tertentu yang ditentukan oleh dimensi gambar baru, dan kami mendapatkannya dengan mengambil sampel fungsi kontinu yang direkonstruksi dari input sampel (Gambar 9.36). Melihatnya seperti ini, itu hanya urutan operasi pemrosesan gambar standar: pertama kami merekonstruksi fungsi kontinu dari sampel input, dan kemudian kami mengambil sampel fungsi itu sama seperti kami akan mengambil sampel gambar kontinu lainnya. Untuk menghindari artifak aliasing, filter yang sesuai perlu digunakan pada setiap tahap.



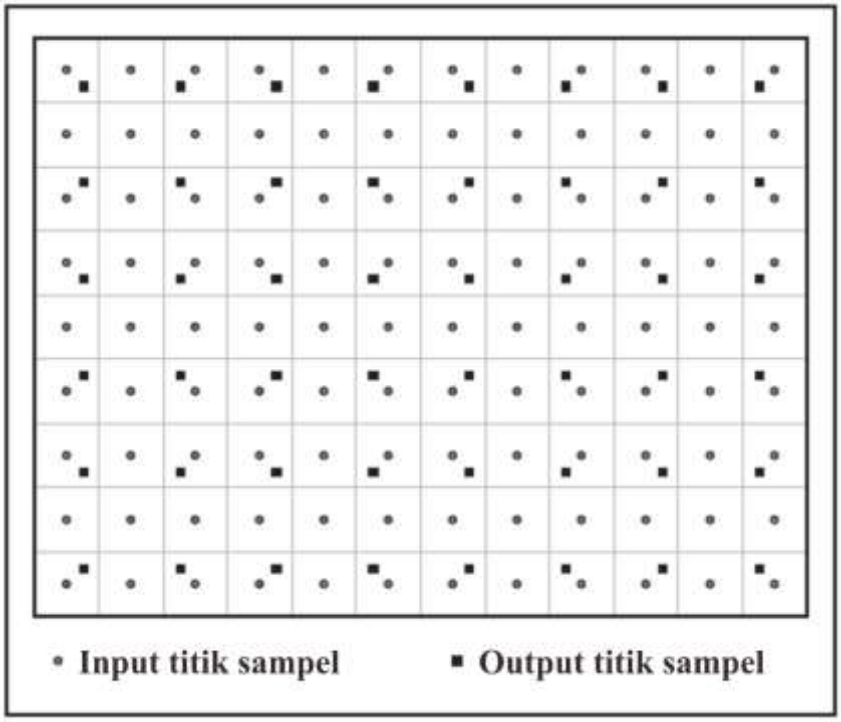
Gambar 9.36 Resampling gambar terdiri dari dua langkah logis yang digabungkan menjadi satu operasi dalam kode. Pertama, kami menggunakan filter rekonstruksi untuk mendefinisikan fungsi kontinu yang mulus dari sampel input. Kemudian, kami mengambil sampel fungsi tersebut pada kisi baru untuk mendapatkan sampel output.

Sebuah contoh kecil ditunjukkan pada Gambar 9.37: jika gambar asli adalah 12 × 9 piksel dan yang baru adalah 8 × 6 piksel, ada 2/3 piksel output pada piksel input di setiap dimensi, jadi jaraknya di seluruh gambar adalah 3/2 jarak dari sampel asli.

Untuk mendapatkan nilai untuk setiap sampel output, kita perlu menghitung nilai untuk gambar di antara sampel. Algoritma pixeldropping memberi kita satu cara untuk melakukan ini: ambil saja nilai sampel terdekat pada gambar input dan jadikan itu nilai output.

Ini persis sama dengan merekonstruksi gambar dengan filter kotak selebar 1 piksel (radius satu setengah) dan kemudian titik pengambilan sampel.

Tentu saja, jika alasan utama untuk memilih penurunan piksel atau pemfilteran yang sangat sederhana lainnya adalah kinerja, orang tidak akan pernah menerapkan metode itu sebagai kasus khusus dari prosedur umum rekonstruksi dan pengambilan sampel ulang. Faktanya, karena diskontinuitas, sulit untuk membuat filter kotak bekerja dalam kerangka umum. Namun, untuk pengambilan sampel ulang berkualitas tinggi, kerangka kerja rekonstruksi/pengambilan sampel memberikan fleksibilitas yang berharga.



Gambar 9.37 Lokasi sampel untuk grid input dan output dalam resampling gambar 12 x 9 untuk membuat 8 x 6.

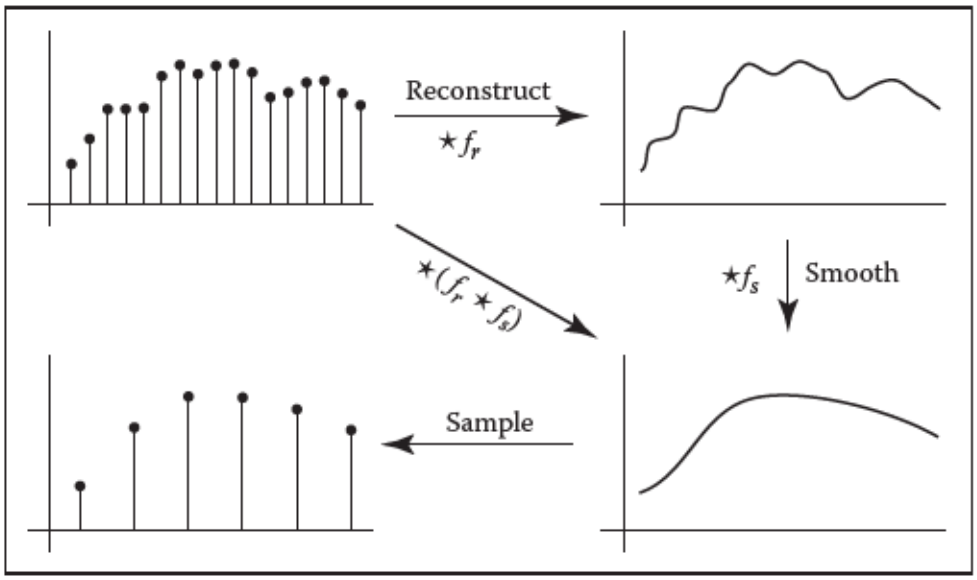
Untuk mengetahui detail algoritme, paling sederhana adalah turun ke 1D dan mendiskusikan pengambilan sampel ulang suatu urutan. Cara paling sederhana untuk menulis implementasi adalah dengan fungsi rekonstruksi yang telah kita definisikan di Bagian 9.2.5.

```
function resample(sequence a, flfloat x0, flfloat Δx, int n, fifilter f)
create sequence b of length n
for i = 0 to n − 1 do
b[i] = reconstruct
(a, f, x0
+ iΔx)
return b
```

Parameter x_0 memberikan posisi sampel pertama dari barisan baru dalam hal sampel dari barisan lama. Artinya, jika sampel output pertama berada di tengah-tengah antara sampel 3 dan 4 dalam urutan input, x_0 adalah 3,5.

Prosedur ini merekonstruksi gambar kontinu dengan memutar urutan input dengan filter kontinu dan kemudian mengarahkan sampelnya. Itu tidak berarti bahwa kedua operasi ini secara kebetulan—fungsi kontinu hanya ada pada prinsipnya dan nilainya dihitung hanya pada titik sampel. Tetapi secara matematis, fungsi ini menghitung himpunan sampel titik dari fungsi a f.

Pengambilan sampel titik ini tampaknya salah, karena kita baru saja selesai mengatakan bahwa sinyal harus diambil sampelnya dengan filter pemulusan yang sesuai untuk menghindari aliasing. Kita harus menggulung fungsi yang direkonstruksi dengan filter pengambilan sampel g dan pengambilan sampel titik g*(f*a). Tetapi karena ini sama dengan (g*f)*a, kita dapat menggulung filter pengambilan sampel bersama dengan filter rekonstruksi; hanya satu operasi konvolusi yang kita butuhkan (Gambar 9.38). Gabungan rekonstruksi dan filter pengambilan sampel ini dikenal sebagai filter pengambilan sampel ulang.



Gambar 9.38 Resampling melibatkan penyaringan untuk rekonstruksi dan untuk pengambilan sampel. Karena dua filter konvolusi yang diterapkan secara berurutan dapat diganti dengan satu filter, kita hanya memerlukan satu filter resampling, yang berfungsi untuk rekonstruksi dan pengambilan sampel.

Saat mengambil sampel gambar, kami biasanya menentukan sumber persegi panjang di unit gambar lama yang menentukan bagian yang ingin kami simpan di gambar baru. Misalnya, menggunakan konvensi penentuan posisi sampel piksel dari Bab 3, persegi panjang yang akan kita gunakan untuk mengambil sampel ulang seluruh gambar adalah $(-0.5, n^{old}_x - 0.5) \times (-0.5, n_y - 0.5)$. Diberikan persegi panjang sumber $(x_l, x_h) \times (y_l, y_h)$, jarak sampel untuk gambar baru adalah $\Delta x = (x_h - x_l)/n_x$ baru dalam x dan $y = (y_h - y_l)/n^{old}_y$ dalam y. Sampel kiri bawah diposisikan di $(x_l + \Delta x/2, y_l + \Delta y/2)$

Memodifikasi pseudocode 1D untuk menggunakan konvensi ini, dan memperluas panggilan ke fungsi rekonstruksi ke dalam loop ganda yang tersirat, kita sampai pada:

```
function resample(sequence a, flfloat xl, flfloat xh, int n, fifilter f)
create sequence b of length n
r = f.radius
x0 = xl + Δx/2
for i = 0 to n − 1 do
s = 0
x = x0 + iΔx
for j = [x – r] ke [ x + r]  do
s = s + a[j]f(x − j)
b[i] = s
return b
```

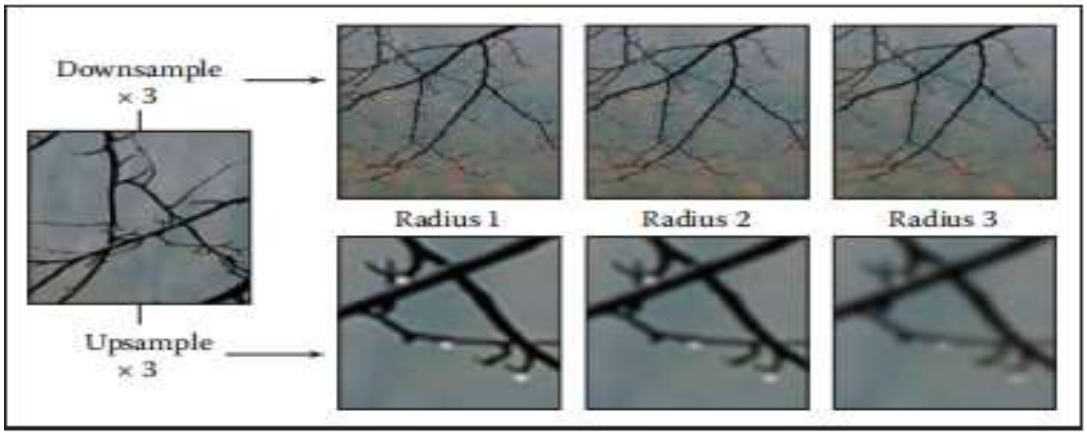
Rutin ini berisi semua dasar-dasar resampling gambar. Satu masalah terakhir yang masih harus ditangani adalah apa yang harus dilakukan di tepi gambar, di mana versi sederhana di sini akan mengakses di luar batas urutan input. Ada beberapa hal yang mungkin kita lakukan:

- Hentikan saja loop di ujung urutan. Ini setara dengan mengisi gambar dengan nol di semua sisi.
- Klip semua akses ke akhir urutan—yaitu, kembalikan a[0] ketika kita ingin mengakses a[−1]. Ini sama dengan mengisi tepi gambar dengan memperpanjang baris atau kolom terakhir.
- Ubah filter saat kita mendekati tepi sehingga tidak melampaui batas urutan.

Opsi pertama mengarah ke tepi yang redup saat kami mengambil sampel ulang seluruh gambar, yang sebenarnya tidak memuaskan. Opsi kedua mudah diterapkan; yang ketiga mungkin berkinerja terbaik. Cara paling sederhana untuk memodifikasi filter di dekat tepi gambar adalah dengan menormalkannya: bagi filter dengan jumlah bagian filter yang ada di dalam gambar. Dengan cara ini, filter selalu menambahkan hingga 1 pada sampel gambar yang sebenarnya, sehingga mempertahankan intensitas gambar. Untuk kinerja, diinginkan untuk menangani pita piksel dalam radius filter tepi (yang memerlukan renormalisasi ini) secara terpisah dari pusat (yang berisi lebih banyak piksel dan tidak memerlukan renormalisasi).

Pemilihan filter untuk pengambilan sampel ulang adalah penting. Ada dua masalah terpisah: bentuk filter dan ukuran (jari-jari). Karena filter berfungsi sebagai filter rekonstruksi dan filter pengambilan sampel, persyaratan dari kedua peran mempengaruhi pilihan filter. Untuk rekonstruksi, kami ingin filter yang cukup halus untuk menghindari aliasing artefak ketika kami memperbesar gambar, dan filter harus bebas dari apel. Untuk pengambilan sampel, filter harus cukup besar untuk menghindari undersampling dan cukup halus untuk menghindari beberapa artefak. Gambar 9.39 mengilustrasikan dua kebutuhan yang berbeda.

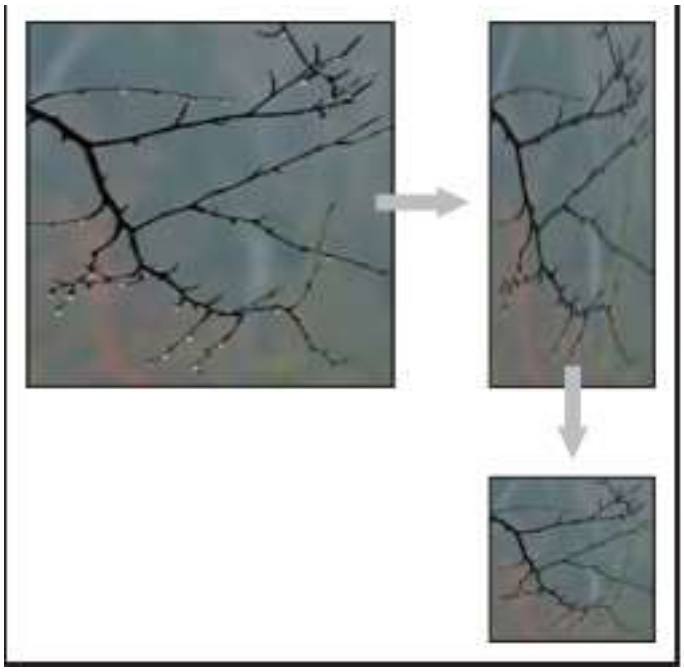
Umumnya, kita akan memilih salah satu bentuk filter dan skala menurut resolusi relatif input dan output. Lebih rendah dari dua resolusi menentukan ukuran filter: ketika output sampel lebih kasar daripada input (downsampling, atau mengecilkan gambar), penghalusan yang diperlukan untuk pengambilan sampel yang tepat lebih besar daripada penghalusan yang diperlukan untuk rekonstruksi, jadi kami mengukur filter menurut jarak sampel output (radius3 pada Gambar 9.39). Di sisi lain, ketika output sampel lebih halus (upsampling, atau memperbesar gambar) maka penghalusan yang diperlukan untuk rekonstruksi mendominasi (fungsi yang direkonstruksi sudah cukup halus untuk sampel pada tingkat yang lebih tinggi daripada yang dimulai), sehingga ukuran filter ditentukan oleh jarak sampel input (radius 1 pada Gambar 9.39).



Gambar 9.39 Efek penggunaan ukuran filter yang berbeda untuk upsampling (memperbesar) atau downsampling (mengurangi) gambar.

Memilih filter itu sendiri merupakan kompromi antara kecepatan dan kualitas. Pilihan umum adalah filter kotak (ketika kecepatan adalah yang terpenting), filter tent (kualitas sedang), atau kubus sepotong (kualitas sangat baik). Dalam kasus kubik piecewise, tingkat smoothing dapat disesuaikan dengan interpolasi antara fB dan fC; filter Mitchell-Netravali adalah pilihan yang baik.

Sama seperti pemfilteran gambar, filter yang dapat dipisahkan dapat memberikan percepatan yang signifikan. Ide dasarnya adalah membuat sampel ulang semua baris terlebih dahulu, menghasilkan gambar dengan lebar yang diubah tetapi tidak tinggi, kemudian sampel ulang kolom dari gambar tersebut untuk menghasilkan hasil akhir (Gambar 9.40). Memodifikasi kodesemu yang diberikan sebelumnya sehingga memanfaatkan optimalisasi ini cukup mudah.



Gambar 9.40 Resampling gambar menggunakan pendekatan yang dapat dipisahkan.

9.5 TEORI SAMPLING

Jika Anda hanya tertarik pada implementasi, Anda dapat berhenti membaca di sini; algoritme dan rekomendasi di bagian sebelumnya akan memungkinkan Anda mengimplementasikan program yang melakukan pengambilan sampel dan rekonstruksi serta mencapai hasil yang sangat baik. Namun, ada teori matematis yang lebih dalam tentang pengambilan sampel dengan sejarah yang mencapai kembali ke tujuan pertama dari representasi sampel di bidang telekomunikasi. Teori pengambilan sampel menjawab banyak pertanyaan yang sulit dijawab dengan penalaran yang hanya didasarkan pada argumen skala.

Tapi yang paling penting, teori sampling memberikan wawasan yang berharga tentang cara kerja sampling dan rekonstruksi. Ini memberi siswa yang mempelajarinya seperangkat alat intelektual tambahan untuk penalaran tentang bagaimana mencapai hasil terbaik dengan kode yang paling efisien.

Transformasi Fourier

Transformasi Fourier, bersama dengan konvolusi, adalah konsep matematika utama yang mendasari teori sampling. Anda dapat membaca tentang transformasi Fourier di banyak buku matematika tentang analisis, serta dalam pemrosesan sinyal buku.

Ide dasar di balik transformasi Fourier adalah untuk mengekspresikan fungsi apa pun dengan menjumlahkan gelombang sinus (sinusoid) dari semua frekuensi. Dengan menggunakan bobot

yang sesuai untuk frekuensi yang berbeda, kita dapat mengatur sinusoida untuk menambahkan hingga fungsi (masuk akal) yang kita inginkan. Sebagai contoh, gelombang persegi pada Gambar 9.41 dapat dinyatakan dengan barisan gelombang sinus:

$$\sum_{n=1,3,5,\ldots}^{\infty} \frac{4}{\pi n} \sin 2\pi n x$$

Deret Empat ini dimulai dengan gelombang sinus(sin2πx)yang memiliki frekuensi1.0—sama seperti gelombang persegi—dan suku lainnya memiliki koreksi yang lebih kecil dan lebih kecil untuk mengurangi riak dan, pada limitnya, mereproduksi gelombang persegi dengan tepat. Perhatikan bahwa semua istilah dalam jumlah memiliki frekuensi yang merupakan kelipatan bilangan bulat dari frekuensi gelombang persegi. Ini karena frekuensi lain akan menghasilkan hasil yang tidak memiliki periode yang sama dengan gelombang persegi.

Fakta yang mengejutkan adalah bahwa sinyal tidak harus periodik untuk dinyatakan sebagai jumlah sinusoidal dengan cara ini: sinyal non-periodik hanya membutuhkan lebih banyak sinusoid. Alih-alih menjumlahkan urutan sinusoidal yang diskrit, kami akan mengintegrasikan lebih dari keluarga sinusoidal yang kontinu. Misalnya, fungsi kotak dapat ditulis sebagai integral dari keluarga gelombang kosinus:

Persamaan 9.6

$$\int_{-\infty}^{\infty} \frac{\sin \pi u}{\pi u} \cos 2\pi u x \, du$$

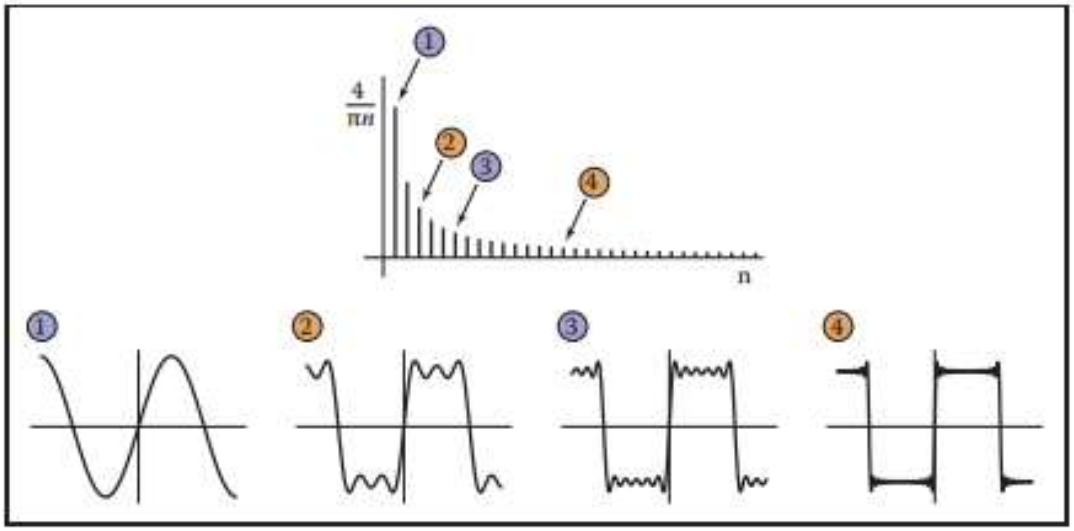
Persamaan integral ini (9.6) menambahkan hingga tak terhingga banyak cosinus, pembobotan kosinus frekuensi u dengan bobot (sinπu)/πu. Hasilnya, saat kami memasukkan frekuensi yang lebih tinggi dan lebih tinggi, konvergen ke fungsi kotak (lihat Gambar 9.42). Ketika suatu fungsi f dinyatakan dengan cara ini, bobot ini, yang merupakan fungsi dari frekuensi u, disebut Transformasi Fourier dari f, dinotasikan f. Fungsi f memberitahu kita bagaimana membangun f dengan mengintegrasikan keluarga sinusoida:

Persamaan 9.7

$$f(x) = \int_{-\infty}^{\infty} f'(u) u^{2\pi i u x} du$$

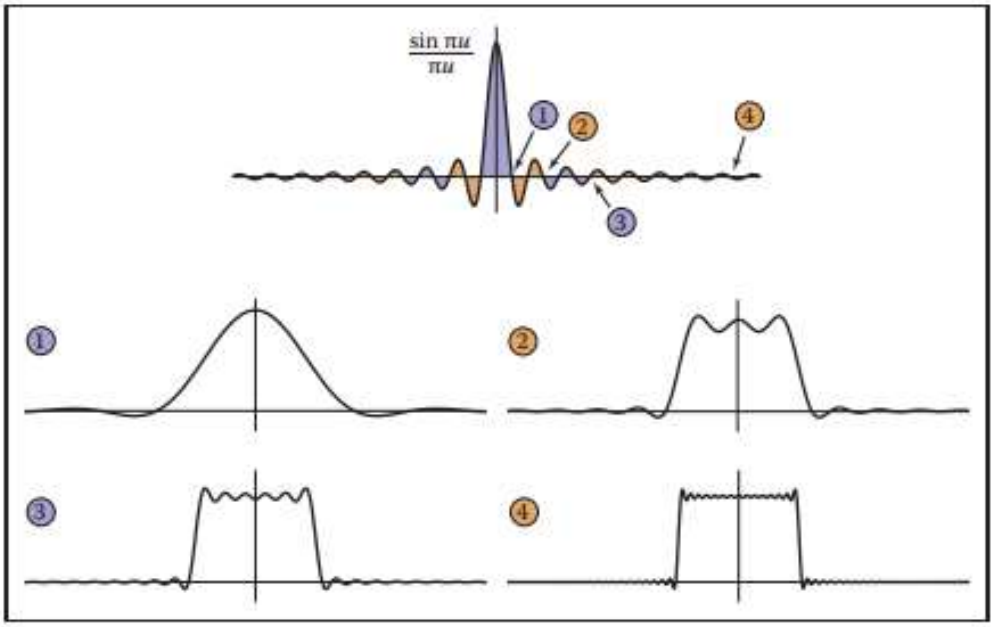
Persamaan (9.7) dikenal sebagai Invers Fourier Transform (IFT) karena dimulai dengan Transformasi Fourier dari f dan diakhiri dengan f.[4]

[4] Perhatikan bahwa istilah "Transformasi Fourier" digunakan baik untuk fungsi f maupun untuk operasi yang menghitung f dari f. Sayangnya, penggunaan yang agak ambigu ini adalah standar.



Gambar 9.41 Mendekati gelombang persegi dengan jumlah sinus yang terbatas.

Perhatikan bahwa dalam Persamaan (9.7) eksponensial kompleks e2πiux telah diganti dengan kosinus dalam persamaan sebelumnya. Juga, f adalah fungsi bernilai kompleks. Mesin bilangan kompleks diperlukan untuk memungkinkan fase, serta frekuensi, dari sinusoida untuk dikontrol; ini diperlukan untuk mewakili setiap fungsi yang tidak simetris di nol. Besarnya f dikenal sebagai spektrum Fourier, dan, untuk tujuan kita, ini sudah cukup—kita tidak perlu mengkhawatirkan fase atau menggunakan bilangan kompleks apa pun secara langsung.



Gambar 9.42 Mendekati fungsi kotak dengan integral kosinus hingga masing-masing dari empat frekuensi cutoff.

Ternyata menghitung f’ dari f sangat mirip dengan menghitung f dari f’:

$$f(x) = \int_{-\infty}^{\infty} f'(u) e^{-2\pi i u x} dx$$

Persamaan (9.8) dikenal sebagai (maju) *Fourier transform* (FT). Tanda dalam eksponensial adalah satu-satunya perbedaan antara transformasi Fourier maju dan terbalik,

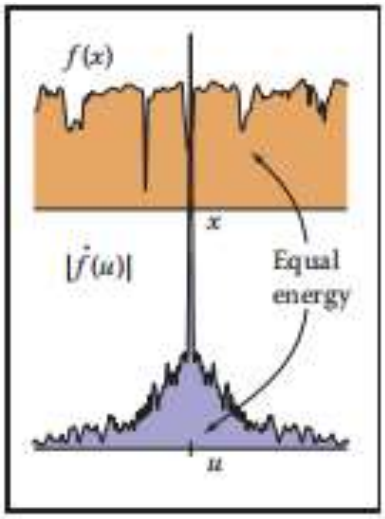
dan itu benar-benar hanya detail teknis. Untuk tujuan kita, kita dapat menganggap FT dan IFT sebagai operasi yang sama.

Kadang-kadang notasi f – f' tidak tepat, dan kemudian kita akan menyatakan transformasi Fourier dengan F{f}dan transformasi Fourier invers dari f oleh F^{-1}{f'}. Suatu fungsi dan transformasi Fouriernya terkait dalam banyak cara yang berguna. Beberapa fakta (kebanyakan mudah diverifikasi) yang akan kita gunakan nanti di bab ini adalah:

- Suatu fungsi dan transformasi Fouriernya memiliki integral kuadrat yang sama:

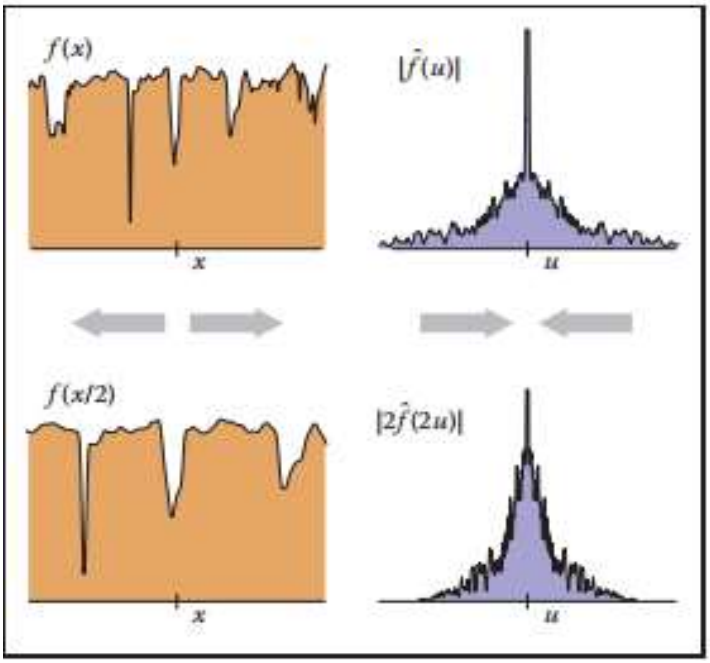
$$\int (f(x))^2 dx = \int (f'(u))^2 du$$

Interpretasi fisiknya adalah bahwa keduanya memiliki energi yang sama (Gambar 9.43).



Gambar 9.43 Transformasi Fourier mempertahankan integral kuadrat dari sinyal.

Secara khusus, menskalakan fungsi dengan a juga menskalakan transformasi Fouriernya dengan a. Yaitu,F{af}= aF{f}.



Gambar 9.44 Scalling sinyal di sepanjang sumbu x dalam domain ruang menyebabkan skala terbalik di sepanjang sumbu u dalam domain frekuensi.

- Meregangkan suatu fungsi pada sumbu x akan menekan transformasi Fouriernya sepanjang sumbu u dengan faktor yang sama (Gambar 9.44):

$$F\{f(x/b)\} = bf\hat{}(bx).$$

(Ada normalisasi oleh b untuk menjaga energi tetap sama.) Ini berarti bahwa jika kita tertarik pada keluarga fungsi dengan lebar dan tinggi yang berbeda (katakanlah semua fungsi kotak berpusat pada nol), maka kita hanya perlu mengetahui transformasi Fourier dari satu fungsi kanonik (katakanlah fungsi kotak dengan lebar dan tinggi sama dengan satu), dan kita dapat dengan mudah mengetahui transformasi Fourier dari semua versi yang diperbesar dan diperkecil dari fungsi tersebut. Sebagai contoh, kita dapat langsung menggeneralisasi Persamaan (9.6) untuk menghasilkan transformasi Fourier dari sebuah kotak dengan lebar b dan tinggi a:

$$ab\frac{\sin \pi bu}{\pi bu}$$

- Nilai rata-rata off sama dengan f(0). Ini masuk akal karena f(0) dianggap sebagai komponen frekuensi nol dari sinyal (komponen DC jika kita memikirkan tegangan listrik).
- Jika f adalah real(yang selalu menguntungkan), f adalah fungsi genap—yaitu, f(u)= f(−u). Demikian juga, jika f adalah fungsi genap maka f akan nyata (ini biasanya tidak terjadi di domain kita, tetapi ingat bahwa kita benar-benar hanya akan peduli dengan besarnya f).

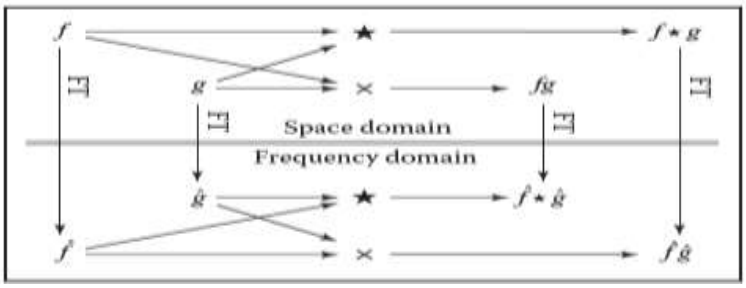
Konvolusi dan Transformasi Fourier

Salah satu sifat terakhir dari transformasi Fourier yang perlu disebutkan secara khusus adalah hubungannya dengan konvolusi (Gambar 9.45). Singkat,

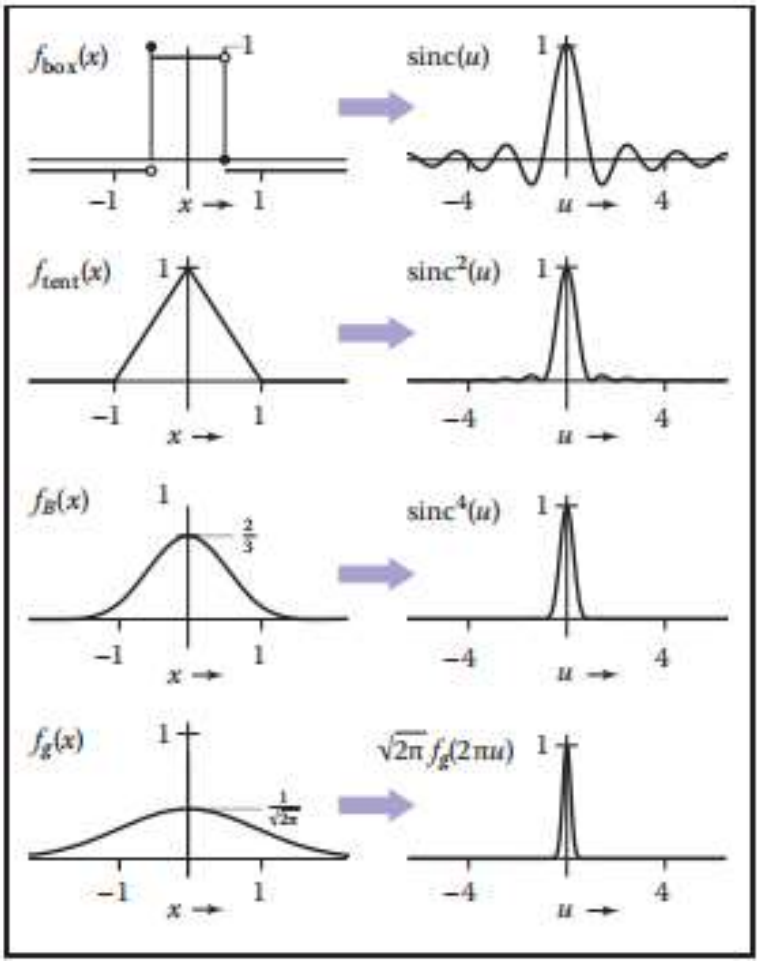
$$F\{f * g\} = f\hat{}\hat{}g.$$

Transformasi Fourier dari konvolusi dua fungsi adalah produk dari transformasi Fourier. Mengikuti simetri yang sekarang sudah dikenal,

Konvolusi dua transformasi Fourier adalah transformasi Fourier dari produk dari dua fungsi. Fakta-fakta ini cukup mudah untuk diturunkan dari definisi. Hubungan ini adalah alasan utama transformasi Fourier berguna dalam mempelajari efek sampling dan rekonstruksi. Kita telah melihat bagaimana pengambilan sampel, pemfilteran, dan rekonstruksi dapat dilihat dalam bentuk konvolusi; sekarang transformasi Fourier memberi kita domain baru—domain frekuensi—di mana operasi ini hanyalah produk.



Gambar 9.45 Diagram komutatif untuk menunjukkan secara visual hubungan antara konvolusi dan perkalian. Jika kita mengalikan f dan g dalam ruang, kemudian mengubahnya menjadi frekuensi, kita berakhir di tempat yang sama seolah-olah kita mengubah f dan g menjadi frekuensi dan kemudian mengonversinya. Demikian juga, jika kita mengonversi f dan g dalam ruang dan kemudian mengubahnya menjadi frekuensi, kita berakhir di tempat yang sama seolah-olah kita mengubah f dan g menjadi frekuensi, kemudian mengalikannya.



Gambar 9.46 Transformasi Fourier dari kotak, tent, B-spline, dan filter Gaussian.

Galeri Transformasi Fourier

Sekarang kita memiliki beberapa fakta tentang transformasi Fourier, mari kita lihat beberapa contoh fungsi individual. Secara khusus, kita akan melihat beberapa filter dari Bagian 9.3.1, yang ditampilkan dengan transformasi Fourier mereka pada Gambar 9.46. Kita telah melihat fungsi kotak:

$$F\{f_{box}\} = \frac{sin\pi u}{\pi u} = \sin \pi u$$

Fungsi[5] sinx/x cukup penting untuk memiliki nama sendiri, sinc x.

Fungsi tent adalah konvolusi kotak dengan dirinya sendiri, sehingga transformasi Fouriernya hanyalah kuadrat dari transformasi Fourier dari fungsi kotak:

$$F\{f_{tent}\} = \frac{sin^2\,\pi u}{\pi^2 u^2} = sin^2\,\pi u$$

Kita dapat melanjutkan proses ini untuk mendapatkan transformasi Fourier dari filter B-spline (lihat Latihan 3):

$$F\{f_B\} = \frac{sin^4\,\pi u}{\pi^4 u^4} = sin^4 \pi u$$

[5] Anda mungkin memperhatikan bahwa sinπu/πu tidak terdefinisi untuk u =0. Namun, terus menerus melintasi nol, dan kami menganggapnya sebagai pemahaman bahwa kami menggunakan nilai pembatas rasio ini, 1, pada u = 0

Gaussian memiliki transformasi Fourier yang sangat bagus:

$$F\{fG\} = e^{-(2\pi u)2/2}.$$

ItisanotherGaussian! Gaussian dengan standar deviasi 1.0 menjadi Gaussian dengan standar deviasi 1/2π.

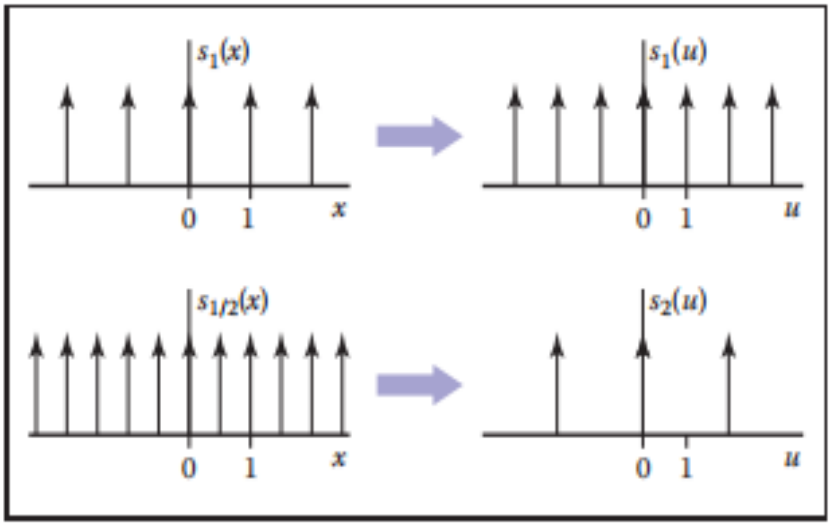
Impuls Dirac dalam Teori Sampling

Alasan mengapa impuls berguna dalam teori sampling adalah bahwa kita dapat menggunakannya untuk membicarakan sampel dalam konteks fungsi kontinu dan transformasi Fourier. Kami mewakili sampel, yang memiliki posisi dan nilai, dengan dorongan yang diterjemahkan ke posisi itu dan diskalakan dengan nilai itu. Sampel pada posisi a dengan nilai b diwakili oleh bδ(x−a). Dengan cara ini kita dapat menyatakan operasi pengambilan sampel fungsi f(x) pada a sebagai mengalikan f dengan (x−a). Hasilnya adalah f(a)δ(x−a).

Oleh karena itu, pengambilan sampel fungsi atas deret titik yang berjarak sama dinyatakan sebagai perkalian fungsi dengan jumlah deret impuls yang berjarak sama, yang disebut rangkaian impuls (Gambar 9.47). Kereta impuls dengan periode T, artinya impuls berjarak T terpisah adalah

$$S_T \sum_{i=-\infty}^{\infty} \delta(T_i)$$

Transformasi Fourier dari s1 sama dengan s1: urutan impuls pada semua frekuensi bilangan bulat. Anda dapat melihat mengapa hal ini harus benar dengan memikirkan tentang apa yang terjadi ketika kita mengalikan impuls yang disusun dengan sinusoid dan terintegrasi. Kami akhirnya menjumlahkan nilai sinusoidal di semua bilangan bulat. Jumlah ini akan benar-benar membatalkan ke nol untuk frekuensi non-bilangan bulat, dan itu akan menyimpang ke +∞untuk frekuensi bilangan bulat.



Gambar 9.47 Kereta impuls. Transformasi Fourier dari kereta impuls adalah kereta impuls lain. Mengubah periode kereta impuls di ruang angkasa menyebabkan perubahan terbalik dalam periode frekuensi.

Karena sifat dilatasi dari transformasi Fourier, kita dapat menebak bahwa transformasi Fourier dari rangkaian impuls dengan periode T (yang seperti dilatasi s_1) adalah rangkaian impuls dengan periode 1/T. Membuat sampel lebih halus dalam domain ruang membuat impuls lebih jauh terpisah dalam domain frekuensi.

Sampling dan Aliasing

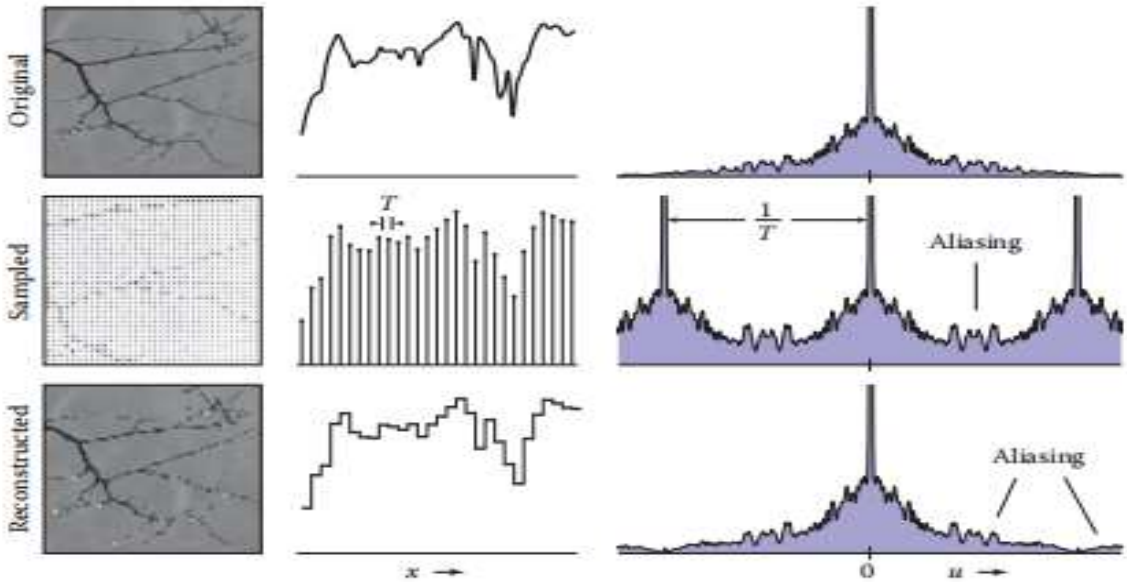
Sekarang setelah kita membangun mesin matematika, kita perlu memahami proses pengambilan sampel dan rekonstruksi dari sudut pandang domain frekuensi. Keuntungan utama memperkenalkan transformasi Fourier adalah bahwa hal itu membuat efek penyaringan konvolusi pada sinyal menjadi lebih jelas, dan memberikan penjelasan yang lebih tepat tentang mengapa kita perlu memfilter saat pengambilan sampel dan rekonstruksi.

Kami memulai proses dengan sinyal asli dan berkelanjutan. Secara umum, Transformasi Fourier dapat mencakup komponen frekuensi satan, meskipun sebagian besar jenis sinyal (terutama gambar), kami berharap konten berkurang seiring frekuensi yang semakin tinggi. Gambar juga cenderung memiliki komponen besar pada frekuensi nol—ingat bahwa komponen frekuensi nol, atau DC, adalah integral dari keseluruhan gambar, dan karena semua gambar bernilai positif, ini cenderung menjadi jumlah yang besar.

Mari kita lihat apa yang terjadi pada transformasi Fourier jika kita mengambil sampel dan merekonstruksi tanpa melakukan penyaringan khusus (Gambar 9.48). Ketika kami mengambil sampel sinyal, kami memodelkan operasi sebagai perkalian dengan rangkaian impuls; sinyal sampel adalah fsT. Karena sifat perkalian-konvolusi, FT dari sinyal sampel adalah $f * s_T = f * s_{1/T}$. Ingatlah bahwa adalah identitas untuk konvolusi. Ini berarti bahwa

$$\left(f * s_{\frac{1}{T}}\right)(u) = \sum_{i=-\infty}^{\infty} f\left(u - \frac{1}{T}\right)$$

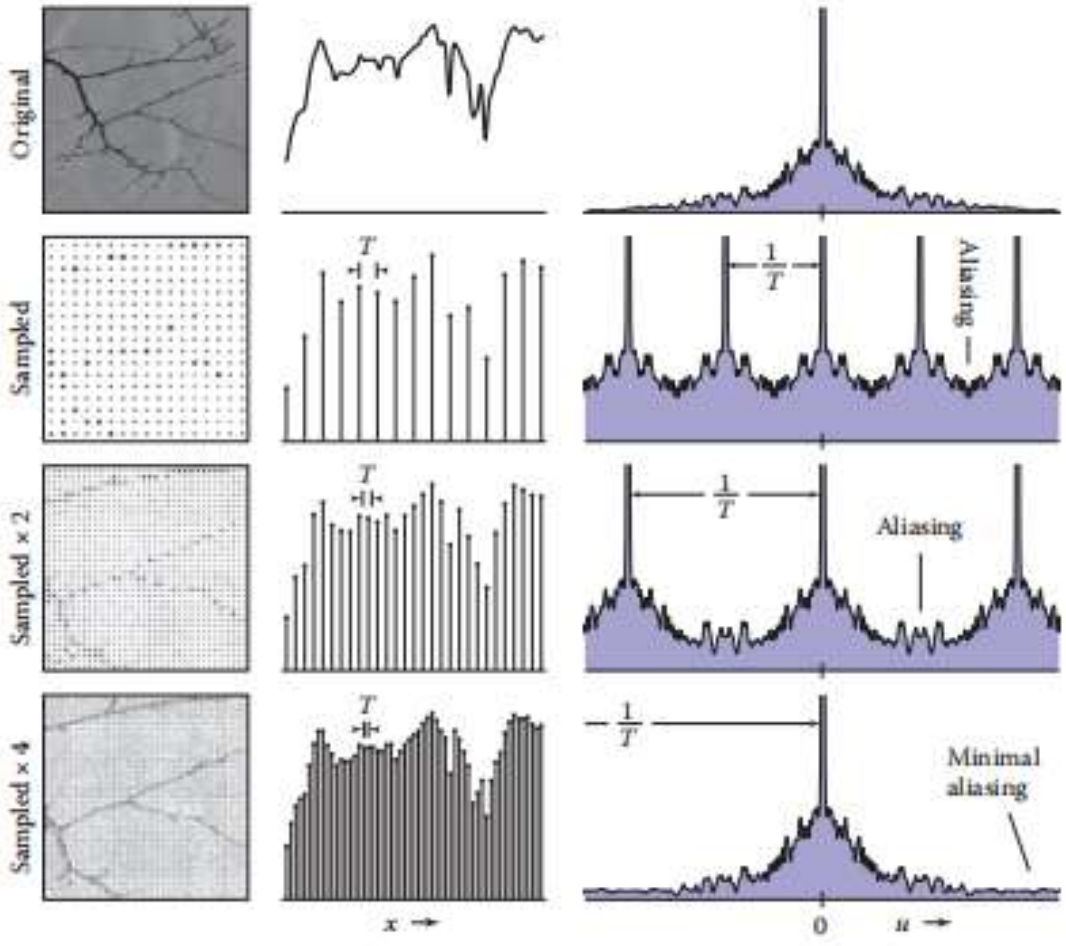
yaitu, berevolusi dengan kereta impuls membuat seluruh rangkaian salinan spektrum f yang berjarak sama. Interpretasi intuitif yang baik dari hasil yang tampaknya aneh ini adalah bahwa semua salinan hanya mengungkapkan fakta (seperti yang kita lihat kembali di Bagian 9.1.1) bahwa frekuensi yang berbeda dengan kelipatan bilangan bulat dari frekuensi pengambilan sampel tidak dapat dibedakan setelah kita mengambil sampel—mereka akan menghasilkan kumpulan sampel yang persis sama. Spektrum asli disebut spektrum dasar dan salinannya disebut spektrum alias.



Gambar 9.48 Pengambilan sampel dan rekonstruksi tanpa penyaringan. Pengambilan sampel menghasilkan spektrum alias yang tumpang tindih dan bercampur dengan spektrum dasar. Rekonstruksi dengan filter kotak mengumpulkan lebih banyak informasi dari spektrum alias. Hasilnya adalah sinyal yang memiliki artifak aliasing yang serius.

Masalah dimulai jika salinan spektrum sinyal tumpang tindih, yang akan terjadi jika sinyal mengandung konten yang signifikan melebihi setengah frekuensi sampel. Ketika ini terjadi, spektra bertambah, dan informasi tentang frekuensi yang berbeda bercampur secara ireversibel. Ini adalah placealiasing pertama yang dapat terjadi, dan jika itu terjadi di sini, itu karena undersampling—menggunakan frekuensi sampel yang terlalu rendah untuk sinyal.

Misalkan kita merekonstruksi sinyal menggunakan teknik tetangga terdekat. Ini ekuivalen dengan konvolusi dengan kotak dengan lebar 1. (Konvolusi kontinu diskrit yang digunakan untuk melakukan ini sama dengan konvolusi kontinu dengan rangkaian impuls yang mewakili sampel.) Sifat konvolusi-perkalian berarti bahwa spektrum sinyal yang direkonstruksi akan menjadi produk dari spektrum sinyal sampel dan spektrum kotak. Hasil transformasi Fourier yang direkonstruksi berisi spektrum dasar (meskipun agak dilemahkan pada frekuensi yang lebih tinggi), ditambah salinan yang dilemahkan dari spektrum jatuh yang lebih rendah. Karena kotak memiliki transformasi Fourier yang cukup luas, bit spektrum alias yang dilemahkan ini signifikan, dan merupakan bentuk aliasing kedua, karena filter rekonstruksi yang tidak memadai, alias komponen-komponen ini memanifestasikan dirinya dalam gambar sebagai pola kotak yang merupakan karakteristik dari rekonstruksi tetangga terdekat.

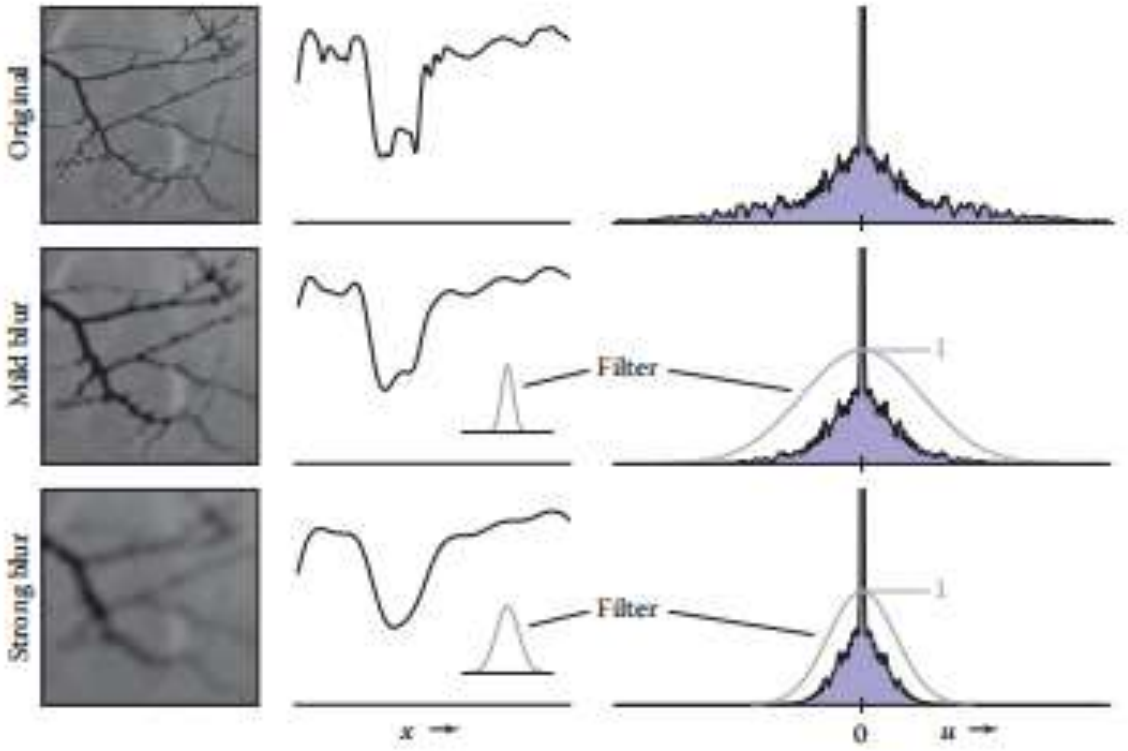


Gambar 9.49 Pengaruh laju sampel pada spektrum frekuensi sinyal sampel. Tingkat sampel yang lebih tinggi mendorong salinan spektrum terpisah, mengurangi masalah yang disebabkan oleh tumpang tindih.

Mencegah Aliasing dalam Pengambilan Sampel

Untuk melakukan pengambilan sampel dan rekonstruksi berkualitas tinggi, kita telah melihat bahwa kita perlu memilih filter pengambilan sampel dan rekonstruksi dengan tepat. Dari sudut domain frekuensi, tujuan penyaringan lowpass saat pengambilan sampel adalah untuk membatasi rentang frekuensi sinyal sehingga spektrum alias tidak tumpang tindih dengan spektrum dasar. Gambar 9.49 menunjukkan pengaruh laju sampel pada transformasi Fourier dari sinyal sampel. Tingkat sampel yang lebih tinggi memindahkan spektrum alias lebih jauh, dan akhirnya overlapis apa pun yang tersisa tidak menjadi masalah.

Kriteria kuncinya adalah lebar spektrum harus lebih kecil dari jarak antara salinan—yaitu, frekuensi tertinggi yang ada dalam sinyal harus kurang dari setengah frekuensi sampel. Ini dikenal sebagai kriteria Nyquist, dan frekuensi tertinggi yang diizinkan dikenal sebagai frekuensi Nyquist atau batas Nyquist. Teorema pengambilan sampel Nyquist-Shannon menyatakan bahwa sinyal yang frekuensinya tidak melebihi batas Nyquist (atau, dengan kata lain, sinyal yang dibatasi pita frekuensinya) dapat, pada prinsipnya, direkonstruksi secara tepat dari sampel.



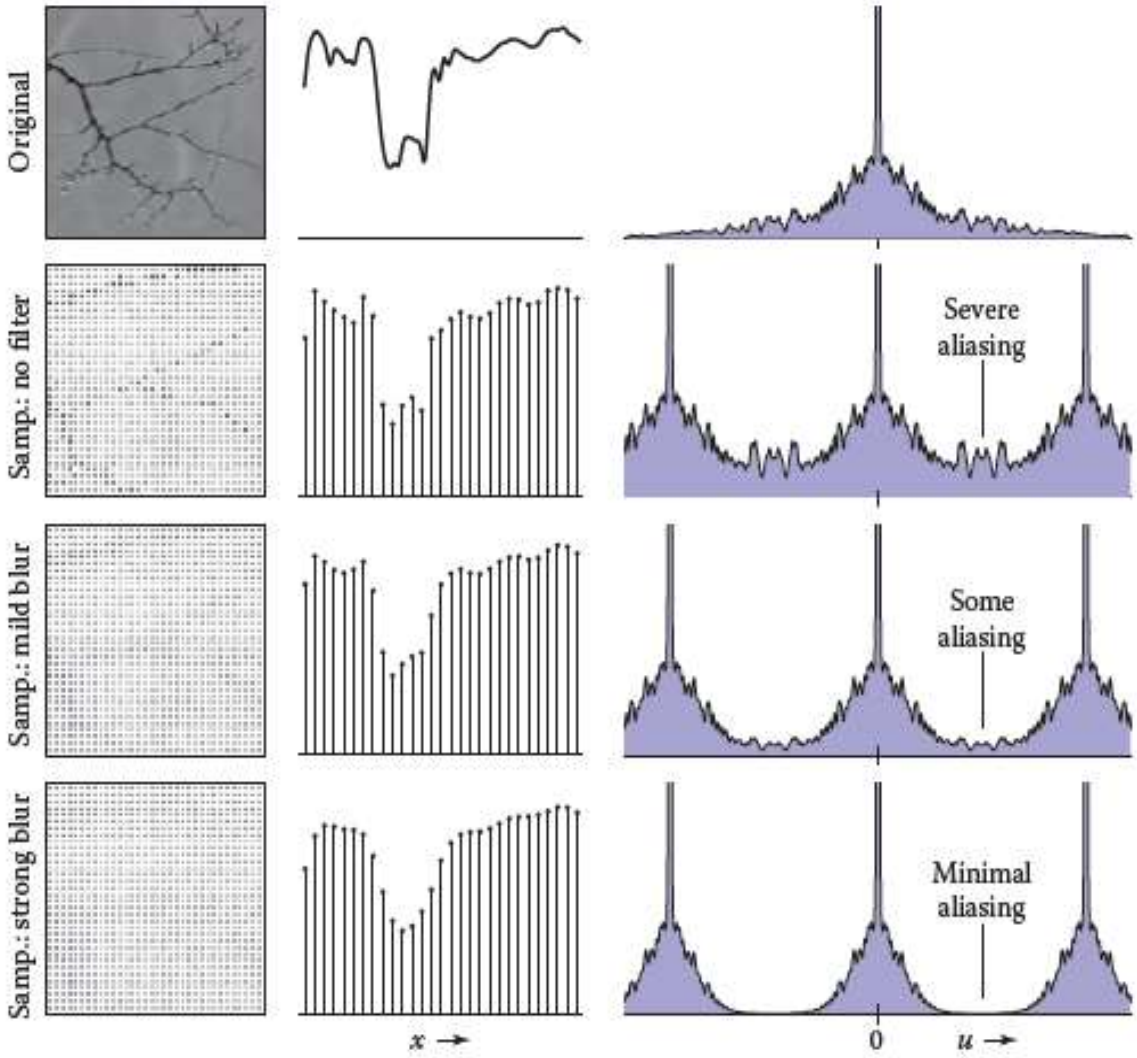
Gambar 9.50 Menerapkan filter lowpass (smoothing) mempersempit spektrum frekuensi sinyal.

Dengan sample rate yang cukup tinggi untuk sinyal tertentu, kita tidak perlu menggunakan filter sampling. Tetapi jika kita terjebak dengan sinyal yang berisi rentang frekuensi yang luas (seperti gambar dengan tepi tajam di dalamnya), kita harus menggunakan filter sampling untuk membatasi pita sinyal sebelum kita dapat mengambil sampelnya. Gambar 9.50 menunjukkan efek dari tiga filter lowpass (smoothing) dalam domain frekuensi, dan Gambar 9.51 menunjukkan efek penggunaan filter yang sama ini saat pengambilan sampel. Bahkan jika spektrum tumpang tindih tanpa penyaringan, menggulung sinyal dengan filter lowpass dapat mempersempit spektrum yang cukup untuk menghilangkan tumpang

tindih dan menghasilkan representasi sinyal yang disaring dengan baik. Tentu saja, kami telah kehilangan frekuensi tinggi, tetapi itu lebih baik daripada membuatnya diacak dengan sinyal dan berubah menjadi artefak.

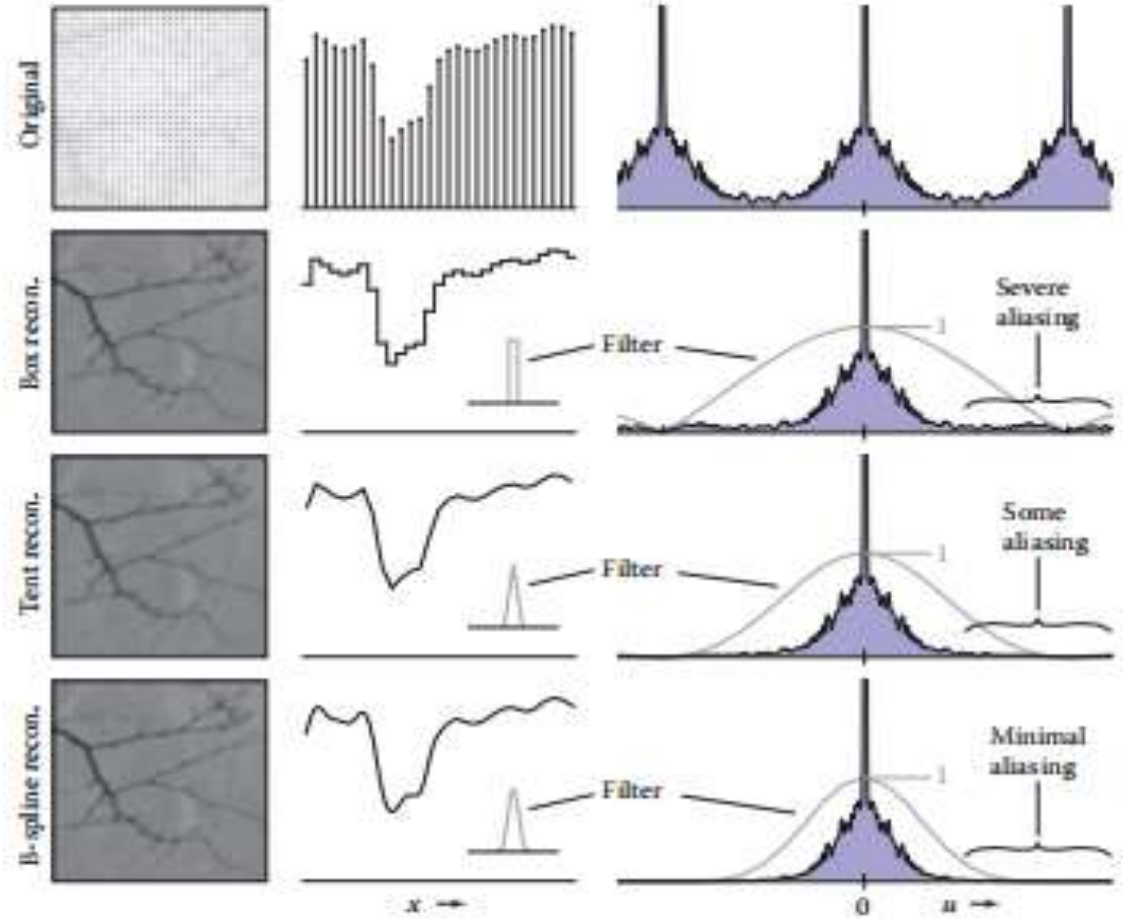
Mencegah Aliasing dalam Rekonstruksi

Dari perspektif domain frekuensi, tugas filter rekonstruksi adalah menghilangkan spektrum alias sambil mempertahankan spektrum dasar. Pada Gambar 9.48, kita dapat melihat bahwa filter rekonstruksi paling kasar, kotak, memang melemahkan spektrum alias. Yang paling penting, itu benar-benar memblokir lonjakan DC untuk semua spektrum alias. Ini adalah karakteristik dari semua filter rekonstruksi yang masuk akal: mereka memiliki nol dalam ruang frekuensi di semua kelipatan frekuensi sampel. Ini ternyata setara dengan properti bebas riak di domain ruang angkasa. Jadi filter rekonstruksi yang baik perlu filter lowpass yang baik, dengan persyaratan tambahan untuk sepenuhnya memblokir semua kelipatan frekuensi sampel.



Gambar 9.51 Bagaimana filter lowpass dari Gambar 9.50 mencegah aliasing selama pengambilan sampel. Penyaringan lowpass mempersempit spektrum sehingga salinan tumpang tindih lebih sedikit, dan frekuensi tinggi dari spektrum alias kurang mengganggu spektrum dasar.

Tujuan penggunaan filter konstruksi yang berbeda dari filter kotak untuk menghilangkan spektrum alias, mengurangi kebocoran artefak frekuensi tinggi ke dalam sinyal yang direkonstruksi, sambil mengganggu spektrum dasar sesedikit mungkin. Gambar 9.52 mengilustrasikan efek dari filter yang berbeda ketika digunakan selama rekonstruksi. Seperti yang telah kita lihat, filter kotak cukup "bocor" dan menghasilkan banyak artefak bahkan jika laju sampel cukup tinggi. Filter tent, menghasilkan interpolasi linier, melemahkan frekuensi tinggi lebih banyak, menghasilkan artefak yang lebih ringan, dan filter B-spline sangat halus, mengendalikan spektrum alias dengan sangat efektif. Ini juga menghaluskan beberapa spektrum dasar—ini adalah tradeoff antara smoothing dan aliasing yang kita lihat sebelumnya.



Gambar 9.52 Efek dari filter rekonstruksi yang berbeda dalam domain frekuensi. Filter rekonstruksi yang baik melemahkan spektrum alias secara efektif sambil mempertahankan spektrum dasar.

Mencegah Aliasing dalam Resampling

Ketika operasi rekonstruksi dan pengambilan sampel digabungkan dalam pengambilan sampel ulang, prinsip yang sama berlaku, tetapi dengan satu filter melakukan pekerjaan konstruksi dan pengambilan sampel. Gambar 9.53 mengilustrasikan bagaimana filter resampling harus menghilangkan spektrum alias dan membiarkan spektrum cukup sempit untuk diambil sampel pada laju sampel baru.

Filter Ideal vs. Filter Berguna

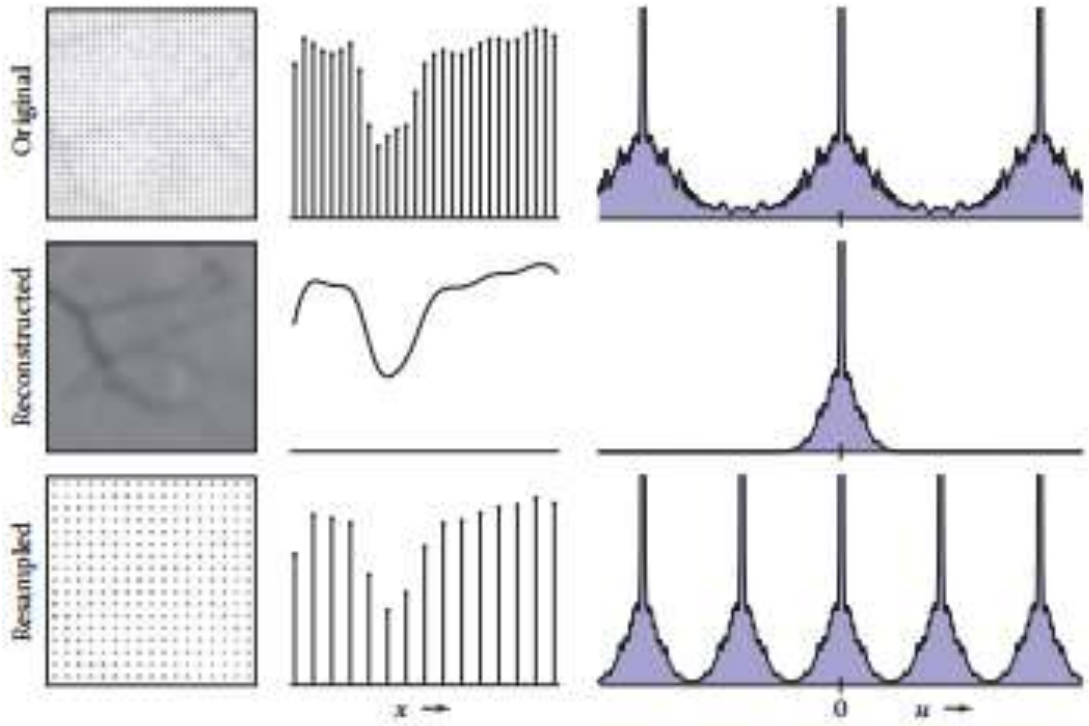
Mengikuti analisis domain frekuensi hingga kesimpulan logisnya, filter yang persis kotak di domain frekuensi sangat ideal untuk pengambilan sampel dan rekonstruksi. Filter

seperti itu akan mencegah aliasing pada kedua tahap tanpa mengurangi frekuensi di bawah frekuensi Nyquist sama sekali.

Ingatlah bahwa transformasi Fourier terbalik dan maju pada dasarnya identik, sehingga filter domain spasial yang memiliki kotak sebagai transformasi Fouriernya adalah fungsi sinπx/πx = sinc x.

Namun, filter ini tidak umum digunakan dalam praktik, baik untuk sampel atau untuk rekonstruksi, karena tidak praktis dan karena, meskipun optimal menurut kriteria domain frekuensi, tidak menghasilkan hasil terbaik untuk banyak aplikasi.

Untuk pengambilan sampel, luasnya filter sinc yang tak terhingga, dan laju penurunan yang relatif lambat dengan jarak dari pusat, merupakan suatu kewajiban. Juga, untuk beberapa jenis pengambilan sampel, lobus negatif bermasalah. Filter Gaussian membuat filter sampel yang sangat baik bahkan untuk kasus-kasus sulit di mana pola frekuensi tinggi harus dihilangkan dari sinyal input, karena transformasi Fouriernya turun secara eksponensial, tanpa benjolan yang cenderung membiarkan alias bocor. Untuk kasus yang tidak terlalu sulit, biasanya cukup dengan menggunakan penyaring tent.



Gambar 9.53 Resampling dilihat dalam domain frekuensi. Filter resampling keduanya merekonstruksi sinyal (menghilangkan spektrum alias) dan membatasi pita (mengurangi lebarnya) untuk pengambilan sampel pada kecepatan baru.

Untuk rekonstruksi, ukuran fungsi sinc kembali menimbulkan masalah, tetapi yang lebih penting, banyak riak menciptakan artefak "berdering" dalam sinyal yang direkonstruksi.

Latihan

1. Tunjukkan bahwa konvolusi diskrit bersifat komutatif dan asosiatif. Lakukan hal yang sama untuk konvolusi kontinu.
2. Konvolusi kontinu-diskrit tidak dapat bersifat komutatif, karena argumennya memiliki dua tipe yang berbeda. Tunjukkan bahwa itu asosiatif.

3. Buktikan bahwa B-spline adalah konvolusi dari empat fungsi kotak.
4. Tunjukkan bahwa definisi konvolusi "terbalik" diperlukan dengan mencoba menunjukkan bahwa konvolusi adalah komutatif dan asosiatif menggunakan definisi (salah) ini (lihat catatan kaki di halaman 192):

$$(a * b)[i] = \sum_j a[j]b[i+j]$$

5. Buktikan bahwaF{f * g} = fˆg’ dan f * g = F{fg}.
6. Persamaan 9.4 dapat diartikan sebagai konvolusi dari sebuah filter. Tulis ekspresi matematika untuk filter "de-rippled" f. Plot filter yang dihasilkan dari de-riak kotak, tent, dan filter B-spline yang diskalakan ke s = 1,25.

BAB 10
SHADING PERMUKAAN

Untuk membuat objek tampak lebih bervolume, penggunaan bayangan dapat membantu, yaitu permukaan "dilukis" dengan cahaya. Ini menciptakan gambar yang mengingatkan pada gambar teknis, yang diinginkan dalam banyak aplikasi.

10.1 SHADING DIFFUSE

Banyak objek di dunia memiliki tampilan permukaan yang secara longgar digambarkan sebagai “matte”, yang menunjukkan bahwa objek tersebut sama sekali tidak mengkilap. Contohnya termasuk kertas, kayu yang belum selesai, dan batu kering yang tidak dipoles. Untuk sebagian besar, objek tersebut tidak memiliki perubahan warna dengan perubahan sudut pandang. Misalnya, jika Anda menatap titik tertentu pada selembar kertas dan bergerak sambil menjaga pandangan Anda tetap pada titik itu, warna pada titik itu akan tetap relatif konstan. Objek matte seperti itu dapat dianggap berperilaku sebagai objek Lambertian. Bagian ini membahas bagaimana menerapkan shading objek tersebut. Poin kuncinya adalah bahwa semua rumus dalam bab ini harus dievaluasi dalam koordinat dunia dan bukan dalam koordinat melengkung setelah transformasi perspektif diterapkan. Jika tidak, sudut antara normal akan berubah dan bayangan akan menjadi tidak akurat.

Model Bayangan Lambertian

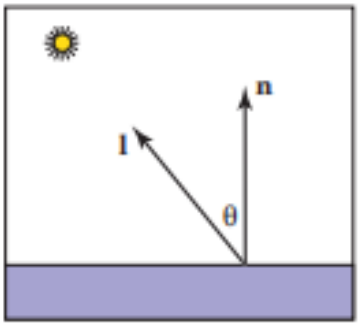
Sebuah objek Lambertian mematuhi hukum kosinus Lambert, yang menyatakan bahwa warna c permukaan sebanding dengan kosinus sudut antara permukaan normal dan arah ke sumber cahaya (Gouraud, 1971):

$$C \propto \cos \vartheta,$$

atau dalam bentuk vektor,

$$c \propto \mathbf{n} \cdot \mathbf{l},$$

dimana n dan l ditunjukkan pada Gambar 10.1. Dengan demikian, warna pada permukaan akan bervariasi sesuai dengan kosinus sudut antara permukaan normal dan arah cahaya. Perhatikan bahwa vektor l biasanya diasumsikan tidak bergantung pada lokasi objek. Asumsi itu setara dengan asumsi bahwa cahaya itu "jauh" relatif terhadap ukuran benda. Cahaya "jauh" seperti itu sering disebut cahaya terarah, karena posisinya hanya ditentukan oleh arah.



Gambar 10.1 Geometri untuk hukum Lambert. Baik n dan l adalah vektor satuan.

Sebuah permukaan dapat dibuat lebih terang atau lebih gelap dengan mengubah intensitas sumber cahaya atau pantulan permukaan. Refleksi difus cr adalah fraksi cahaya yang dipantulkan oleh permukaan. Pecahan ini akan berbeda untuk komponen warna yang berbeda. Misalnya, sebuah permukaan berwarna merah jika memantulkan fraksi yang lebih

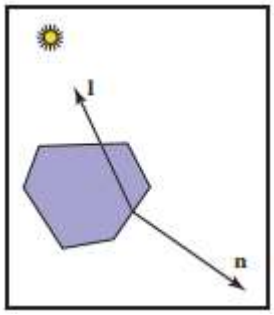
tinggi dari cahaya datang merah daripada biru. Jika kita menganggap warna permukaan sebanding dengan cahaya yang dipantulkan dari permukaan, maka reflektansi difus cr—warna RGB—juga harus disertakan: (Persamaan 10.1)

$$c \propto cr\mathbf{n} \cdot \mathbf{l}.$$

Ruas kanan Persamaan (10.1) adalah warna RGB dengan semua komponen RGB dalam rentang [0,1]. Kami ingin menambahkan efek intensitas cahaya sambil menjaga komponen RGB dalam kisaran [0,1]. Ini menyarankan untuk menambahkan istilah intensitas RGB cl yang memiliki komponen dalam kisaran [0,1]: (Persamaan 10.2)

$$c = crcl\mathbf{n} \cdot \mathbf{l}.$$

Ini adalah bentuk yang sangat nyaman, tetapi dapat menghasilkan komponen RGB untuk c yang berada di luar kisaran [0,1], karena hasil kali titik dapat negatif. Hasil kali titik negatif ketika permukaan menjauhi cahaya seperti yang ditunjukkan pada Gambar 10.2.



Gambar 10.2 Ketika suatu permukaan menjauhi cahaya, seharusnya tidak menerima cahaya. Kasus ini dapat diverifikasi dengan memeriksa apakah hasil kali titik dari l dan n adalah negatif.

Fungsi "maks" dapat ditambahkan ke Persamaan (10.2) untuk menguji kasus itu: (Persamaan 10.30)

$$c = crcl\max(0, \mathbf{n} \cdot \mathbf{l}).$$

Cara lain untuk menangani cahaya "negatif" adalah dengan menggunakan nilai absolut: (Persamaan `0.4)

$$c = crcl|\mathbf{n} \cdot \mathbf{l}|.$$

Meskipun Persamaan (10.4) mungkin tampak tidak masuk akal secara fisik, persamaan tersebut sebenarnya sesuai dengan Persamaan (10.3) dengan dua cahaya dalam arah yang berlawanan. Untuk alasan ini sering disebut pencahayaan dua sisi (Gambar 10.3).

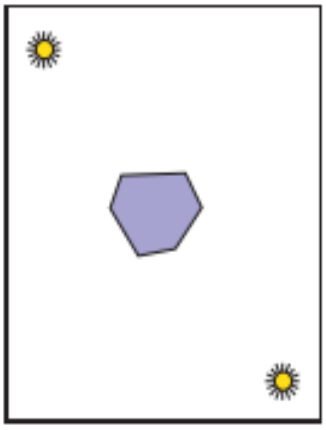
Bayangan Sekitar

Satu masalah dengan bayangan difus dari Persamaan (10.3) adalah bahwa setiap titik yang wajah normalnya menjauhi cahaya akan menjadi hitam. Dalam kehidupan nyata, cahaya dipantulkan ke mana-mana, dan beberapa cahaya datang dari segala arah. Selain itu, sering ada skylight yang memberikan pencahayaan “ambient”. Salah satu cara untuk mengatasinya adalah dengan menggunakan beberapa sumber cahaya. Trik umum adalah selalu menempatkan sumber redup di mata sehingga semua titik yang terlihat akan menerima sedikit cahaya. Cara lain adalah dengan menggunakan pencahayaan dua sisi seperti yang dijelaskan oleh Persamaan (10.4). Pendekatan yang lebih umum adalah dengan menambahkan istilah ambient (Gouraud, 1971). Ini hanya warna konstan yang ditambahkan ke Persamaan (10.3):

$$c = cr\,(ca + cl\max(0, \mathbf{n} \cdot \mathbf{l})).$$

Secara intuitif, Anda dapat menganggap warna sekitar ca sebagai warna rata-rata semua permukaan dalam scene. Jika Anda ingin memastikan bahwa warna RGB yang dihitung

tetap dalam kisaran $[0,1]^3$, maka $c_a + c_l \leq (1, 1, 1)$. Jika tidak, kode Anda harus "menjepit" nilai RGB di atas satu untuk memiliki nilai satu.

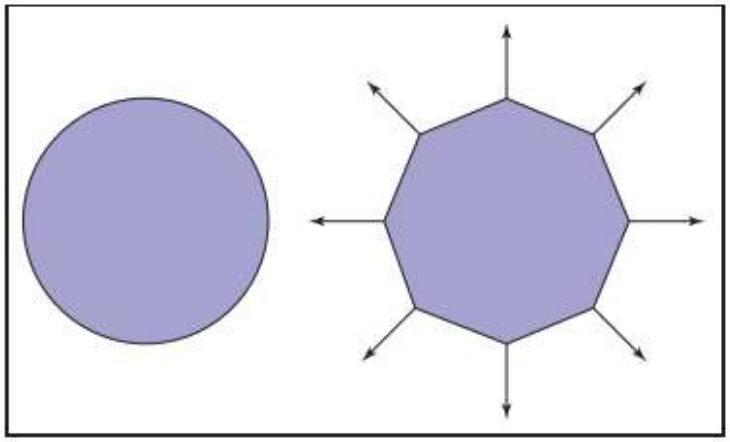


Gambar 10.3 Menggunakan Persamaan (10.4), rumus pencahayaan dua sisi, setara dengan mengasumsikan dua sumber cahaya berlawanan warna sama.

Bayangan Difusi Berbasis Vertex

Jika kita menerapkan Persamaan (10.1) pada objek yang terdiri dari segitiga, objek tersebut biasanya memiliki tampilan segi. Seringkali, segitiga adalah pendekatan untuk permukaan yang halus. Untuk menghindari munculnya segi, kita dapat menempatkan vektor normal permukaan pada simpul segitiga (Phong, 1975), dan menerapkan Persamaan (10.3) pada setiap simpul menggunakan vektor normal pada simpul (lihat Gambar 10.4). Ini akan memberikan warna pada setiap simpul segitiga, dan warna ini dapat diinterpolasi menggunakan dibagian sebelumnya.

Satu masalah dengan bayangan di simpul segitiga adalah bahwa kita perlu mendapatkan normal dari suatu tempat. Banyak model akan datang dengan normal yang disediakan. Jika Anda menguji model mulus Anda sendiri, Anda dapat membuat normal saat Anda membuat segitiga. Jika Anda disajikan dengan model poligonal yang tidak memiliki normal di simpul dan Anda ingin menaungi dengan mulus, Anda dapat menghitung normal dengan berbagai metode heuristik. Yang paling sederhana adalah dengan rata-rata normal segitiga yang berbagi setiap simpul dan menggunakan rata-rata normal ini pada simpul. Normal rata-rata ini tidak akan secara otomatis menjadi satuan panjang, jadi Anda harus mengubahnya menjadi vektor satuan sebelum menggunakannya untuk bayangan.



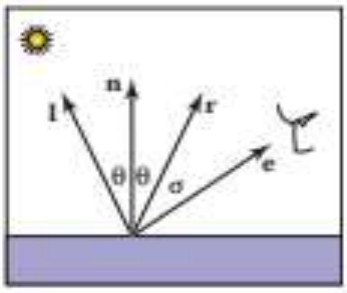
Gambar 10.4 Lingkaran (kiri) didekati dengan segi delapan (kanan). Titik normalsmerekam normal permukaan dari kurva asli.

10.2 SHADING PHONG

Beberapa permukaan pada dasarnya seperti permukaan matte, tetapi mereka memiliki highlight. Contoh permukaan tersebut meliputi lantai tiling yang dipoles, cat kilap, dan papan tulis. Sorotan bergerak melintasi permukaan saat sudut pandang bergerak. Ini berarti bahwa kita harus menambahkan vektor satuan e ke arah mata ke dalam persamaan kita. Jika Anda melihat dengan cermat pada sorotan, Anda akan melihat bahwa itu benar-benar pantulan cahaya; terkadang refleksi ini kabur. Warna sorotan ini adalah warna cahaya—warna permukaan tampaknya tidak banyak berpengaruh. Hal ini karena pemantulan terjadi pada permukaan objek, dan cahaya yang menembus permukaan dan mengambil warna objek tersebar secara difus.

Model Pencahayaan Phong

Kami ingin menambahkan "titik" kabur dengan warna yang sama dengan sumber cahaya di tempat yang tepat. Pusat titik harus digambar di mana arah e ke mata “bergaris” dengan arah alami pemantulan r seperti yang ditunjukkan pada Gambar 10.5. Di sini "berbaris" secara matematis setara dengan "di mana adalah nol." Kami ingin sorotan memiliki beberapa area bukan nol, sehingga mata dapat melihat beberapa sorotan di mana punσ kecil.



Gambar 10.5 Geometri untuk model iluminasi Phong. Mata akan melihat sorotan jika kecil.

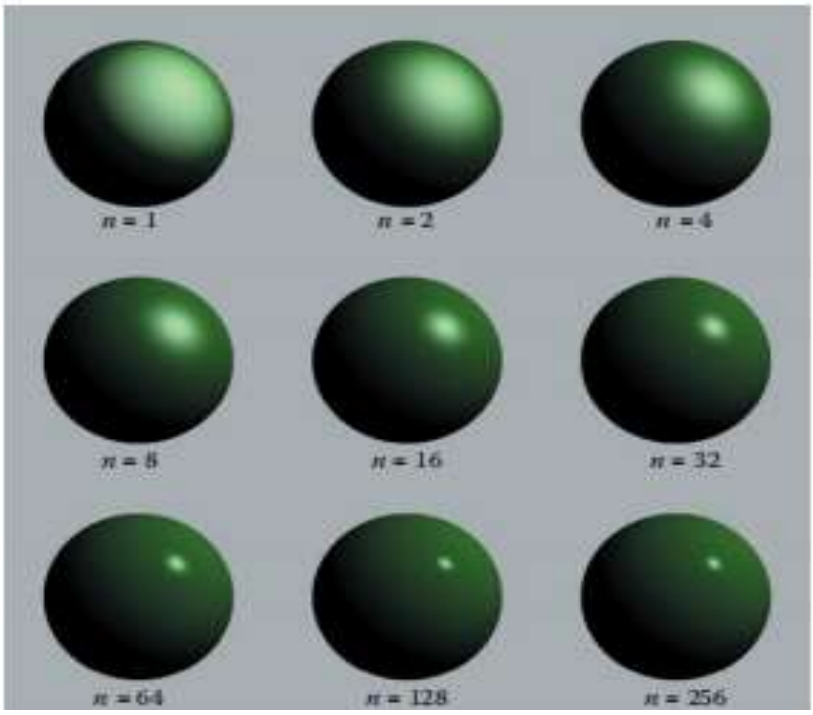
Mengingat r, kami ingin fungsi heuristik yang cerah ketika e = r dan turun secara bertahap ketika e menjauh dari r. Kandidat yang jelas adalah kosinus sudut di antara mereka:

$$c = cl(\mathbf{e} \cdot \mathbf{r}).$$

Ada dua masalah dengan menggunakan persamaan ini. Pertama, hasil kali titik bisa negatif. Ini dapat diselesaikan secara komputasi dengan pernyataan "jika" yang menetapkan warna ke nol ketika produk titik negatif. Masalah yang lebih serius adalah bahwa sorotan yang dihasilkan oleh persamaan ini jauh lebih luas daripada yang terlihat dalam kehidupan nyata. Maksimum berada di tempat yang tepat dan warnanya tepat, tetapi terlalu besar. Kita dapat mempersempitnya tanpa mengurangi warna maksimumnya dengan menaikkan pangkat: (Persamaan 10.5)

$$c = cl\max(0, \mathbf{e} \cdot \mathbf{r})p.$$

Di sini p disebut eksponen Phong; itu adalah bilangan real positif (Phong, 1975). Efek yang mengubah eksponen Phong pada sorotan dapat dilihat pada Gambar 10.6.

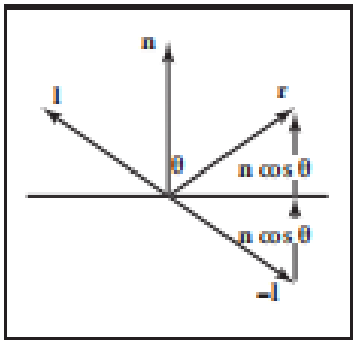


Gambar 10.6 Pengaruh eksponen Phong pada karakteristik sorotan. Ini menggunakan Persamaan (10.5) untuk sorotan. Ada juga komponen difus, memberikan objek tampilan yang mengkilap tetapi bukan logam. Gambar milik Nate Robins.

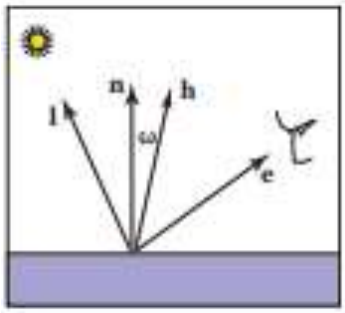
Untuk mengimplementasikan Persamaan (10.5), pertama-tama kita perlu menghitung vektor satuan r. Diketahui vektor satuan l dan n, r adalah vektor l yang direfleksikan terhadap n. Gambar 10.7 menunjukkan bahwa vektor ini dapat dihitung (Persamaan 10.6)

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n},$$

di mana produk titik digunakan untuk menghitung cos θ.



Gambar 10.7 Geometri untuk menghitung vektor r.



Gambar 10.8 Vektor satuan h berada di tengah antara l dan e.

Model heuristik alternatif berdasarkan Persamaan (10.5) menghilangkan kebutuhan untuk memeriksa nilai negatif dari bilangan yang digunakan sebagai basis untuk eksponensial (Warn, 1983). Alih-alih r, kita menghitung h, vektor satuan di tengah antara l dan e (Gambar 10.8):

$$h = \frac{e + 1}{||e + 1||}$$

Sorotan terjadi ketika h dekat n, yaitu, ketika cosω = h·n dekat 1. Ini menunjukkan aturan: (Persamaan 10.7)

$$c = cl(\mathbf{h} \cdot \mathbf{n})p.$$

Eksponen p di sini akan memiliki perilaku kontrol yang analog dengan eksponen dalam Persamaan (10.5), tetapi sudut antara h dan n adalah setengah ukuran sudut antara e dan r, sehingga detailnya akan sedikit berbeda. Keuntungan menggunakan kosinus antara n dan h adalah selalu positif untuk mata dan cahaya di atas bidang. Kerugiannya adalah bahwa akar kuadrat dan pembagian diperlukan untuk menghitung h.

Dalam praktiknya, kami ingin sebagian besar bahan memiliki tampilan yang menyebar selain sorotan. Kita dapat menggabungkan Persamaan (10.3) dan (10.7) untuk mendapatkan (Persamaan 10.8)

$$c = cr\,(ca + cl\max(0, \mathbf{n} \cdot \mathbf{l})) + cl(\mathbf{h} \cdot \mathbf{n})p.$$

Jika kita ingin mengizinkan pengguna untuk meredupkan sorotan, kita dapat menambahkan controlterm cp: (Persamaan 10.9)

$$c = cr\,(ca + cl\max(0, \mathbf{n} \cdot \mathbf{l})) + clcp(\mathbf{h} \cdot \mathbf{n})p.$$

Istilah cp adalah warna RGB, yang memungkinkan kita untuk mengubah warna sorotan. Ini berguna untuk logam di mana cp = cr, karena sorotan pada logam menghasilkan warna metalik. Selain itu, sering kali berguna untuk membuat cp nilai netral kurang dari satu, sehingga warna tetap di bawah satu. Misalnya, pengaturan cp =1− M di mana M adalah komponen maksimum cr akan menjaga warna di bawah satu untuk satu sumber cahaya dan tidak ada ambientterm.

Interpolasi Vektor Normal Permukaan

Permukaan halus dengan highlight cenderung cepat berubah warna dibandingkan permukaan Lambertian dengan geometri yang sama. Dengan demikian, bayangan pada vektor normal dapat menghasilkan artefak yang mengganggu.

Masalah-masalah ini dapat diatasi dengan menginterpolasi vektor normal di seluruh poligon dan kemudian menerapkan bayangan Phong pada setiap piksel. Ini memungkinkan Anda mendapatkan gambar yang bagus tanpa membuat ukuran segitiga menjadi sangat kecil. Ingat dari Bab 3, bahwa ketika rasterisasi sebuah segitiga, kita menghitung koordinat barycentric (α, β, γ) untuk menginterpolasi warna titik c_0, c_1, c_2: (Persamaan 10.10)

$$c = \alpha c0 + \beta c1 + \gamma c2.$$

Kita dapat menggunakan persamaan yang sama untuk menginterpolasi normal permukaan n_0, n_1, dan n_2: (Persamaan 10.11)

$$\mathbf{n} = \alpha\mathbf{n}0 + \beta\mathbf{n}1 + \gamma\mathbf{n}2$$

Dan Persamaan (10.9) kemudian dapat dievaluasi untuk kemudian menghitung tanggal setiap piksel. Perhatikan bahwa n yang dihasilkan dari Persamaan (10.11) biasanya bukan satuan normal. Hasil visual yang lebih baik akan dicapai jika dikonversi ke vektor satuan sebelum digunakan dalam perhitungan bayangan. Interpolasi normal jenis ini sering disebut interpolasi normal Phong (Phong, 1975).

10.3 SHADING ARTISTIK

Metode shading Lambertian dan Phong didasarkan pada heuristik yang dirancang untuk meniru penampilan objek di dunia nyata. Bayangan artistik dirancang untuk meniru gambar yang dibuat oleh seniman manusia (Yessios, 1979; Dooley & Cohen, 1990; Saito & Takahashi, 1990; L.Williams, 1991). Bayangan seperti itu tampaknya memiliki keuntungan dalam banyak aplikasi. Misalnya, produsen mobil mempekerjakan seniman untuk menggambar diagram untuk manual pemilik mobil. Ini lebih mahal daripada menggunakan foto-foto yang jauh lebih "realistis", jadi mungkin ada beberapa keuntungan intrinsik bagi teknik seniman ketika jenis komunikasi tertentu diperlukan. Di bagian ini, kami menunjukkan cara membuat gambar garis berbayang halus yang mengingatkan pada gambar yang digambar manusia. Membuat gambar seperti itu sering disebut rendering non-fotorealistik, tetapi kami akan menghindari istilah itu karena banyak teknik non-fotorealistik digunakan untuk efisiensi yang tidak terkait dengan praktik artistik apa pun.

Menggambar garis

Hal paling jelas yang kita lihat dalam gambar manusia yang tidak kita lihat di kehidupan nyata adalah siluet. Ketika kita memiliki satu set segitiga dengan sisi yang sama, kita harus menggambar sebuah sisi sebagai siluet ketika salah satu dari dua sisi tanpa segitiga menghadap ke arah penampil, dan segitiga lainnya menghadap ke arah penampil. Kondisi ini dapat diuji untuk dua normal n_0 dan n_1 dengan

$$\text{draw silhouette if } (\mathbf{e} \cdot \mathbf{n}0)(\mathbf{e} \cdot \mathbf{n}1) \leq 0.$$

Di sini e adalah vektor dari tepi ke mata. Ini bisa berupa titik mana saja di tepi atau salah satu segitiga. Atau, jika $f_i(p) = 0$ adalah persamaan bidang implisit untuk dua segitiga, pengujian dapat ditulis

$$\text{draw silhouette if } f0(\mathbf{e})f1(\mathbf{e}) \leq 0.$$

Kami juga ingin menggambar tepi yang terlihat dari model poligonal. Untuk melakukan ini, kita dapat menggunakan salah satu metode permukaan tersembunyi dari Bab 12 untuk menggambar dalam warna latar belakang dan kemudian menggambar garis luar setiap segitiga dalam warna hitam. Ini, sebenarnya, juga akan menangkap siluet. Sayangnya, jika poligon mewakili permukaan yang halus, kami benar-benar tidak ingin menggambar sebagian besar tepi tersebut. Namun, kita mungkin ingin menggambar semua lipatan di mana benar-benar ada sudut dalam geometri. Kami dapat menguji lipatan dengan menggunakan ambang heuristik:

$$\text{draw crease if } (\mathbf{n}0 \cdot \mathbf{n}1) \leq \text{threshold}$$

Ini dikombinasikan dengan tes siluet akan memberikan gambar garis yang tampak bagus.

Bayangan Dingin-ke-Hangat

Ketika seniman menaungi gambar garis, mereka sering menggunakan bayangan intensitas rendah untuk memberi kesan melengkung ke permukaan dan memberi warna pada objek (Gooch, Gooch, Shirley, & Cohen, 1998). Permukaan yang menghadap ke satu arah diarsir dengan warna dingin, seperti biru, dan permukaan yang menghadap ke arah yang berlawanan diarsir dengan warna hangat, seperti oranye. Biasanya warna-warna ini tidak terlalu jenuh dan juga tidak gelap. Dengan begitu, siluet hitam muncul dengan apik. Secara keseluruhan ini memberikan efek seperti kartun. Ini dapat dicapai dengan mengatur arah ke cahaya "hangat" l dan menggunakan kosinus untuk memodulasi warna, di mana konstanta kehangatan kw didefinisikan pada [0, 1]:

$$k_w = \frac{1 + n \,.\, 1}{2}$$

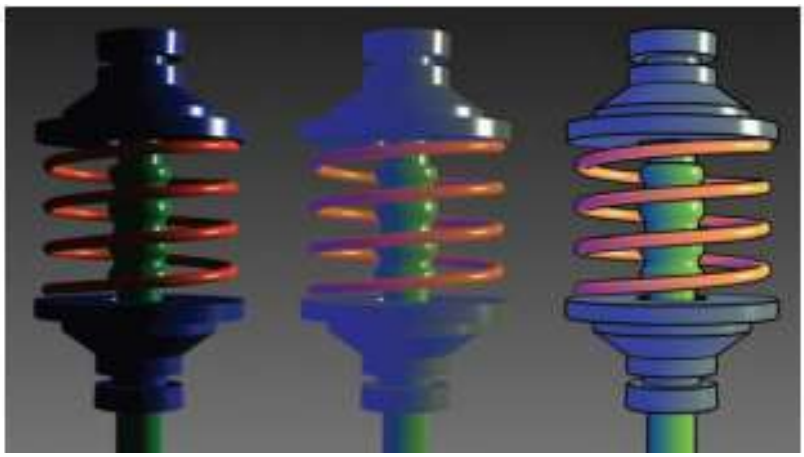
Warna c kemudian hanyalah perpaduan linier dari warna dingin cc dan warna hangat cw:

$$c = kwcw + (1 - kw)cc$$

Ada banyak kemungkinan c_w dan c_b yang akan menghasilkan hasil yang terlihat masuk akal. Tempat awal yang bagus untuk menebak adalah

$$cc = (0.4, 0.4, 0.7),$$
$$cc = (0.8, 0.6, 0.6).$$

Gambar 10.9 menunjukkan perbandingan antara pencahayaan tradisional Phong dan jenis naungan artistik ini.



Gambar 10.9 Kiri: gambar bercahaya Phong. Tengah: bayangan dingin-ke-hangat tidak berguna tanpa siluet. Kanan: bayangan dingin-ke-hangat plus siluet. Gambar milik Amy Gooch.

Pertanyaan yang Sering Diajukan

- Semua bayangan dalam bab ini tampak seperti peretasan besar. Benarkah?

Ya. Namun, mereka adalah peretasan yang dirancang dengan cermat yang telah terbukti berguna dalam praktiknya. Dalam jangka panjang, kita mungkin akan memiliki algoritma dengan motivasi yang lebih baik yang mencakup fisika, psikologi, dan mapping nada. Namun, peningkatan kualitas gambar mungkin akan bertahap.

- Saya benci memanggil pow(). Apakah ada cara untuk menghindarinya saat melakukan pencahayaan Phong?

Cara sederhana adalah dengan hanya memiliki eksponen yang merupakan pangkat dua, yaitu 2, 4, 8, 16, ... Dalam praktiknya, ini bukan pembatasan bermasalah untuk sebagian besar aplikasi. Tabel pencarian juga dimungkinkan, tetapi seringkali tidak memberikan percepatan yang besar.

Latihan

1. Bulan didekati dengan buruk oleh bayangan difus atau Phong. Pengamatan apa yang memberi tahu Anda bahwa ini benar?
2. Velvet kurang didekati dengan difus atau naungan Phong. Pengamatan apa yang memberi tahu Anda bahwa ini benar?
3. Mengapa sebagian besar highlight pada benda plastik terlihat putih, sedangkan pada logam emas terlihat emas?

BAB 11
MAPPING TEKSTUR

Ketika mencoba untuk meniru tampilan dunia nyata, seseorang dengan cepat menyadari bahwa hampir tidak ada permukaan yang tidak berbentuk. Kayu tumbuh dengan biji-bijian; kulit tumbuh dengan kerutan; kain menunjukkan struktur tenunannya; cat menunjukkan bekas kuas atau rol yang meletakkannya. Plastik halus rata dibuat dengan tonjolan yang dibentuk ke dalamnya, dan logam halus menunjukkan tanda dari proses pemesinan yang membuatnya. Bahan-bahan yang tadinya tidak memiliki sifat dengan cepat menjadi tertutup oleh bekas, penyok, noda, goresan, sidik jari, dan kotoran.

Dalam grafis komputer, kita lihat fenomena ini di bawah judul "sifat permukaan yang bervariasi secara spasial"—atribut permukaan yang bervariasi dari satu tempat ke tempat lain tetapi tidak benar-benar mengubah bentuk permukaan dengan cara yang berarti. Untuk memungkinkan efek ini, semua jenis sistem pemodelan dan rendering menyediakan beberapa cara untuk mapping tekstur: menggunakan gambar, yang disebut peta tekstur, gambar tekstur, atau hanya tekstur, untuk menyimpan detail yang ingin Anda tuju di permukaan, kemudian secara matematis "memetakan" gambar ke permukaan.

Ternyata, setelah mekanisme untuk memetakan gambar ke permukaan ada, ada banyak cara yang kurang jelas dapat digunakan yang melampaui tujuan dasar memperkenalkan detail permukaan. Tekstur dapat digunakan untuk membuat bayangan dan pantulan, memberikan penerangan, bahkan untuk menentukan bentuk permukaan. Dalam program interaktif yang canggih, tekstur digunakan untuk menyimpan semua jenis data yang bahkan tidak ada hubungannya dengan gambar!

Pertama-tama, tekstur mudah terdistorsi, dan mendesain fungsi yang memetakan tekstur ke permukaan merupakan tantangan. Juga, mapping tekstur adalah proses resampling, sama seperti rescaling gambar, dan seperti yang kita lihat di Bab 9, resampling dapat dengan mudah memperkenalkan artefak alias. Penggunaan mapping tekstur dan animasi bersama-sama dengan mudah menghasilkan aliasing yang benar-benar dramatis, dan sebagian besar kompleksitas sistem mapping tekstur dibuat oleh tindakan antialiasing yang digunakan untuk menjinakkan artefak ini.

11.1 MENCARI NILAI TEKSTUR

Untuk memulai, mari pertimbangkan aplikasi sederhana dari mapping tekstur. Kami memiliki scene dengan lantai kayu, dan kami ingin warna lantai yang menyebar dikontrol oleh gambar yang menunjukkan papan lantai dengan serat kayu. Terlepas dari apakah kita menggunakan ray tracing atau rasterization, kode shading yang menghitung warna untuk titik perpotongan permukaan sinar atau untuk fragmen yang dihasilkan oleh rasterizer perlu mengetahui warna tekstur pada titik shading, untuk menggunakan sebagai warna difus dalam model Lambertianshading dari Bab 10.

Untuk mendapatkan warna ini, shader melakukan pencarian tekstur: menentukan lokasi, dalam sistem koordinat gambar tekstur, yang sesuai dengan titik bayangan, dan membacakan warna pada titik itu dalam gambar, menghasilkan sampel tekstur. Warna itu kemudian digunakan dalam bayangan, dan karena pencarian tekstur terjadi di tempat berbeda dalam tekstur untuk setiap piksel yang melihat lantai, pola warna berbeda muncul di gambar. Kodenya mungkin terlihat seperti ini:

```
Color texture_lookup(Texture t, float u, float v) {
int i = round(u * t.width() - 0.5)
```

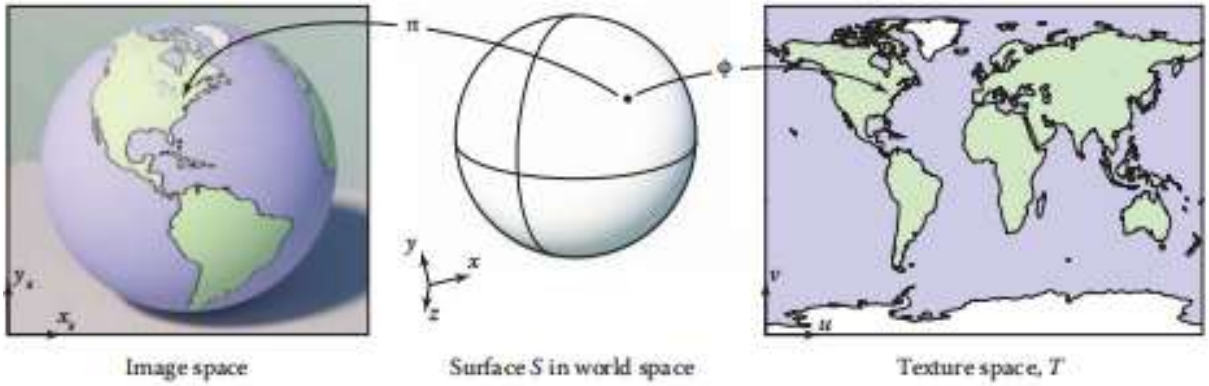
```
int j = round(v * t.height() - 0.5)
return t.get_pixel(i,j)
}
Color shade_surface_point(Surface s, Point p, Texture t) {
Vector normal = s.get_normal(p)
(u,v) = s.get_texcoord(p)
Color diffuse_color = texture_lookup(u,v)
// compute shading using diffuse_color and normal
// return shading result
}
```

Dalam kode ini, bayangan akan menanyakan permukaan tempat untuk melihat tekstur, dan bagaimanapun juga setiap permukaan yang ingin kita arsir menggunakan tekstur harus dapat menjawab pertanyaan ini. Ini membawa kita ke bahan utama pertama dari mapping tekstur: kita membutuhkan fungsi yang memetakan dari permukaan ke tekstur yang dapat kita hitung dengan mudah untuk setiap piksel. Ini adalah fungsi koordinat tekstur (Gambar 11.1) dan kita katakan bahwa ini memberikan koordinat tekstur ke setiap titik di permukaan. Secara matematis adalah mapping dari permukaan S ke domain tekstur, T:

$$\emptyset: S \rightarrow T$$

$$: (x, y, z) \rightarrow (u, v).$$

Himpunan T, sering disebut "ruang tekstur," biasanya hanya persegi panjang yang berisi gambar; biasanya menggunakan satuan kuadrat $(u,v) \in [0, 1]^2$ (dalam buku ini kita akan menggunakan nama u dan v untuk dua koordinat tekstur). Dalam banyak hal ini mirip dengan proyeksi tampilan yang dibahas dalam Bab 7, yang disebut π dalam bab ini, yang memetakan titik pada permukaan dalam scene ke titik dalam gambar; keduanya merupakan mapping 3D-ke2D, dan keduanya diperlukan untuk rendering—satu untuk mengetahui dari mana mendapatkan nilai tekstur, dan satu untuk mengetahui dari mana untuk menempatkan hasil bayangan pada gambar. Tetapi ada beberapa perbedaan penting juga: π hampir selalu merupakan perspektif atau proyeksi ortografis, sedangkan Ø dapat mengambil banyak bentuk; dan hanya ada satu proyeksi tampilan untuk sebuah gambar, sedangkan setiap objek dalam scene kemungkinan memiliki fungsi koordinat tekstur yang benar-benar terpisah.



Gambar 11.1 Sama seperti proyeksi tampilan memetakan setiap titik pada permukaan objek, S, ke suatu titik dalam gambar, fungsi koordinat tekstur memetakan setiap titik pada permukaan objek ke suatu titik di peta tekstur, T. Mendefinisikan fungsi ini dengan tepat merupakan dasar untuk semua aplikasi mapping tekstur.

Mungkin tampak mengejutkan bahwa Ø adalah mapping dari permukaan ke tekstur, ketika tujuan kita adalah menempatkan tekstur ke permukaan, tetapi ini adalah fungsi yang

kita butuhkan. Untuk kasus lantai kayu, jika lantai berada pada konstanta z dan sejajar dengan sumbu x dan y, kita dapat menggunakan mapping

$$u = ax; v = by,$$

untuk beberapa faktor skala a dan b yang dipilih dengan tepat, untuk menetapkan koordinat tekstur(s, t) ke titik $(x, y, z)_{floor}$, dan kemudian menggunakan nilai piksel tekstur, atau texel, yang paling dekat dengan (u,v) sebagai nilai tekstur pada (x,y). Dengan cara ini kami membuat gambar pada Gambar 11.2.

Gambar 11.2 Lantai kayu, bertekstur menggunakan fungsi koordinat tekstur yang hanya menggunakan koordinat titik x dan y secara langsung.

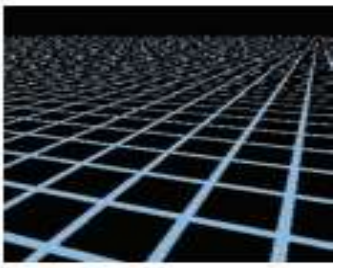
Namun, ini cukup membatasi: bagaimana jika ruangan dimodelkan pada sudut sumbu x dan y, atau bagaimana jika kita menginginkan tekstur kayu di bagian belakang kursi yang melengkung? Kita akan membutuhkan beberapa cara yang lebih baik untuk menghitung koordinat tekstur untuk titik-titik di permukaan.

Masalah lain yang muncul dari bentuk paling sederhana dari mapping tekstur diilustrasikan secara dramatis dengan merendering tekstur kontras tinggi dari sudut yang sangat merumput ke dalam gambar beresolusi rendah. Gambar 11.3 menunjukkan bidang yang lebih besar bertekstur menggunakan pendekatan yang sama tetapi dengan pola grid kontras tinggi dan pandangan ke arah cakrawala. Anda dapat melihatnya berisi artifak aliasing (tangga di latar depan, pola bergelombang dan berkilauan di kejauhan) serupa dengan yang muncul dalam pengambilan sampel ulang gambar (Bab 9) bila filter yang sesuai tidak digunakan. Meskipun dibutuhkan kasus ekstrim untuk membuat artefak ini begitu jelas dalam gambar diam kecil yang dicetak dalam sebuah buku, dalam animasi pola-pola ini bergerak dan sangat mengganggu bahkan ketika mereka jauh lebih halus.

Kami sekarang telah melihat dua masalah utama dalam mapping tekstur dasar:

- mendefinisikan fungsi koordinat tekstur, dan
- mencari nilai tekstur tanpa memasukkan terlalu banyak aliasing.

Kedua perhatian ini mendasar untuk semua jenis aplikasi mapping tekstur dan dibahas dalam Bagian 11.2 dan 11.3. Setelah Anda memahaminya dan beberapa solusi untuknya, maka Anda memahami mapping tekstur. Selebihnya hanyalah bagaimana menerapkan mesin tekstur dasar untuk berbagai tujuan yang berbeda, yang dibahas dalam Bagian 11.4.



Gambar 11.3 Bidang horizontal besar, bertekstur dengan cara yang sama seperti pada Gambar 11.2 dan menampilkan artifak aliasing yang parah.

11.2 FUNGSI KOORDINAT TEKSTUR

Merancang fungsi koordinat tekstur sumur merupakan persyaratan utama untuk mendapatkan hasil yang baik dengan mapping tekstur. Anda dapat menganggap ini sebagai memutuskan bagaimana Anda akan mengubah bentuk, gambar persegi panjang sehingga sesuai dengan permukaan 3D yang ingin Anda gambar. Atau sebagai alternatif, Anda mengambil permukaan dan meratakannya dengan lembut, tanpa membuatnya kusut, sobek, atau terlipat, sehingga menempel pada gambar. Terkadang ini mudah: mungkin permukaan 3D sudah berbentuk persegi panjang datar! Dalam kasus lain sangat rumit: bentuk 3D mungkin sangat rumit, seperti permukaan tubuh karakter.

Masalah dalam mendefinisikan fungsi koordinat tekstur bukanlah hal baru dalam grafis komputer. Masalah yang persis sama dihadapi oleh kartografer ketika merancang peta yang mencakup area yang luas di permukaan bumi: mapping dari globe melengkung ke peta tentu saja menyebabkan distorsi area, sudut, dan/atau jarak yang dapat dengan mudah membuat peta menjadi sangat menyesatkan. Banyak proyeksi peta telah diusulkan selama berabad-abad, semua menyeimbangkan masalah persaingan yang sama — meminimalkan berbagai jenis distorsi sambil menutupi area yang luas dalam satu bagian yang berdekatan — yang dihadapi dalam mapping tekstur.

Dalam beberapa aplikasi (seperti yang akan kita lihat nanti di bab ini) ada alasan yang jelas untuk menggunakan peta tertentu. Namun dalam kebanyakan kasus, merancang peta koordinat tekstur adalah tugas yang rumit untuk menyeimbangkan kekhawatiran yang bersaing, yang dilakukan oleh pemodel yang terampil.

"Mapping UV" atau "parameterisasi permukaan" adalah nama lain yang mungkin Anda temui untuk fungsi koordinat tekstur.

Anda dapat mendefinisikan Ø dengan cara apa pun yang Anda impikan. Tetapi ada beberapa tujuan yang bersaing untuk dipertimbangkan:

- **Bijektivitas**. Dalam kebanyakan kasus, Anda ingin menjadi bijektif (lihat Bagian 2.1.1), sehingga setiap titik pada permukaan dipetakan ke titik yang berbeda dalam ruang tekstur. Jika beberapa titik dipetakan ke titik ruang tekstur yang sama, nilai pada satu titik di tekstur akan mempengaruhi beberapa titik di permukaan. Dalam kasus di mana Anda ingin tekstur berulang di atas permukaan (pikirkan wallpaper atau karpet dengan pola berulangnya), masuk akal untuk secara sengaja memperkenalkan mapping banyak ke satu dari titik permukaan ke titik tekstur, tetapi Anda tidak ingin ini terjadi secara tidak sengaja.
- **Distorsi ukuran**. Skala tekstur harus kira-kira konstan di seluruh permukaan. Artinya, titik-titik yang berdekatan di mana saja di permukaan yang jaraknya kira-kira sama harus dipetakan ke titik-titik dengan jarak yang sama dalam tekstur. Dalam hal fungsi , besaran turunan dari tidak boleh terlalu bervariasi.
- **Distorsi bentuk.** Teksturnya tidak boleh terlalu terdistorsi. Artinya, lingkaran kecil yang digambar di permukaan harus dipetakan ke bentuk yang cukup melingkar di ruang tekstur, daripada bentuk yang sangat terjepit atau memanjang. Dalam hal , turunan dari tidak boleh terlalu berbeda dalam arah yang berbeda.
- **Kontinuitas**. Seharusnya tidak ada terlalu banyak jahitan: titik-titik tetangga di permukaan harus memetakan titik-titik tetangga dalam tekstur. Artinya, harus kontinu, atau memiliki diskontinuitas sesedikit mungkin. Dalam kebanyakan kasus, beberapa diskontinuitas tidak dapat dihindari, dan kami ingin menempatkannya di lokasi yang tidak mencolok.

Permukaan yang didefinisikan oleh persamaan parametrik (Bagian 2.5.8) hadir dengan pilihan bawaan untuk fungsi koordinat tekstur: cukup balikkan fungsi yang mendefinisikan

permukaan, dan gunakan dua parameter permukaan sebagai koordinat tekstur. Koordinat tekstur ini mungkin atau mungkin tidak memiliki sifat yang diinginkan, tergantung pada permukaannya, tetapi mereka menyediakan mapping.

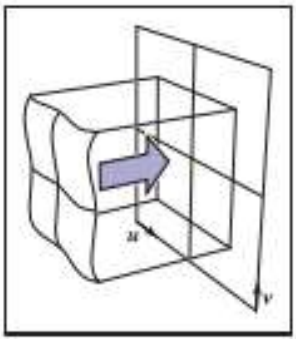
Tetapi untuk permukaan yang didefinisikan secara implisit, atau hanya didefinisikan oleh mesh segitiga, kita memerlukan beberapa cara lain untuk mendefinisikan koordinat tekstur, tanpa bergantung pada parameterisasi yang ada. Secara garis besar, dua cara untuk menentukan koordinat tekstur adalah dengan menghitungnya secara geometris, dari koordinat spasial titik permukaan, atau, untuk permukaan mesh, untuk menyimpan nilai koordinat tekstur pada titik-titik dan menginterpolasinya di seluruh permukaan. Mari kita lihat opsi-opsi ini satu per satu.

Koordinat yang Ditentukan Secara Geometris

Koordinat tekstur yang ditentukan secara geometris digunakan untuk bentuk sederhana atau situasi khusus, sebagai solusi cepat, atau sebagai titik awal untuk merancang peta koordinat tekstur yang disesuaikan dengan tangan.

Kami akan mengilustrasikan berbagai fungsi koordinat tekstur dengan memetakan gambar uji pada Gambar 11.4 ke permukaan. Angka-angka dalam gambar memungkinkan Anda membaca perkiraan (u,v) koordinat dari gambar yang dirender, dan kisi memungkinkan Anda melihat seberapa terdistorsi mappingnya.

Gambar 11.4 Gambar uji.



Gambar 11.5 Proyeksi planar membuat parameterisasi yang berguna untuk objek atau bagian objek yang hampir datar untuk memulai, jika arah proyeksi dipilih secara kasar di sepanjang garis normal keseluruhan.

Proyeksi Planar

Mungkin mapping paling sederhana dari 3D ke 2D adalah proyeksi paralel—mapping yang sama seperti yang digunakan untuk tampilan ortografis (Gambar 11.5). Mesin yang telah kami kembangkan untuk melihat (Bagian 7.1) dapat digunakan kembali secara langsung untuk menentukan koordinat tekstur: seperti halnya tampilan ortografi yang bermuara pada

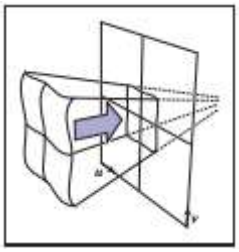
perkalian dengan matriks dan membuang komponen z, menghasilkan koordinat tekstur dengan proyeksi planar dapat dilakukan dengan matriks sederhana berkembang biak:

$$\emptyset(x, y, z) = (u, v) dimana \begin{bmatrix} u \\ v \\ * \\ 1 \end{bmatrix} = M_t \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

di mana matriks tekstur Mt mewakili transformasi yang halus, dan tanda bintang menunjukkan bahwa kita tidak peduli apa yang berakhir di koordinat ketiga.

Ini bekerja cukup baik untuk permukaan yang sebagian besar datar, tanpa terlalu banyak variasi normal permukaan, dan arah proyeksi yang baik dapat ditemukan dengan mengambil rata-rata normal. Namun, untuk semua jenis bentuk tertutup, proyeksi bidang tidak akan menjadi injektif: titik-titik di bagian depan dan belakang akan dipetakan ke titik yang sama dalam ruang tekstur (Gambar 11.6).

Gambar 11.6 Menggunakan proyeksi planar pada objek tertutup akan selalu menghasilkan mapping satu-ke-banyak noninjektif, dan distorsi ekstrem di dekat titik-titik di mana arah proyeksi bersinggungan dengan permukaan.



Gambar 11.7 Transformasi tekstur proyektif menggunakan transformasi seperti tampilan yang memproyeksikan ke suatu titik.

Dengan hanya mengganti proyeksi perspektif untuk ortografi, weget koordinat tekstur proyek (Gambar 11.7):

$$\emptyset(x, y, z) = (u, v) dimana \begin{bmatrix} u \\ v \\ * \\ 1 \end{bmatrix} = M_t \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Sekarang, matriks 4 × 4 P_t merepresentasikan transformasi proyektif (tidak harus halus)—yaitu, baris terakhir mungkin bukan [0, 0, 0, 1]. Koordinat tekstur proyektif penting dalam teknik mapping bayangan, dibahas dalam Bagian 11.4.4.

Koordinat Bulat

Untuk bola, parameterisasi lintang/bujur sudah dikenal dan digunakan secara luas. Ia memiliki banyak distorsi di dekat kutub, yang dapat menyebabkan kesulitan, tetapi ia menutupi seluruh bola dengan diskontinuitas hanya di sepanjang satu garis lintang.

Permukaan yang kira-kira berbentuk bola dapat diparameterisasi menggunakan fungsi koordinat tekstur yang memetakan titik pada permukaan ke suatu titik pada bola menggunakan proyeksi radial: ambil garis dari pusat bola melalui titik pada permukaan, dan

temukan perpotongan dengan bola. Koordinat bola dari titik perpotongan ini adalah koordinat tekstur dari titik yang Anda mulai di permukaan.

Cara lain untuk mengatakan ini adalah Anda menyatakan titik permukaan dalam koordinat bola (ρ, θ, ϕ) dan kemudian membuang koordinat dan memetakan dan masing-masing ke kisaran [0, 1]. Rumusnya tergantung pada konvensi koordinat bola; menggunakan konvensi Bagian 2.5.8,

$$\phi(x, y, z) = ([\pi + \text{atan2}(y, x)]/2\pi, [\pi - \text{acos}(z/x)]/\pi).$$

Peta koordinat asferis akan bijektif di mana-mana kecuali di kutub jika seluruh permukaan terlihat dari titik pusat. Ini mewarisi distorsi yang sama di dekat kutub sebagai peta lintang-bujur. Gambar 11.8 menunjukkan objek yang koordinat bolanya menyediakan fungsi koordinat tekstur yang sesuai.

Koordinat Silinder

Untuk objek yang lebih berbentuk kolom daripada bola, proyeksi ke luar dari sumbu ke silinder dapat bekerja lebih baik daripada proyeksi dari titik ke bola (Gambar 11.9). Analog dengan proyeksi bola, ini sama dengan mengkonversi ke koordinat silinder dan membuang jari-jari:

$$\phi(x, y, z) = 21\pi [\pi + \text{atan2}(y, x)]/2\pi, 12 [1 + z]$$

Gambar 11.8 Untuk objek seperti bola yang samar-samar ini, memproyeksikan setiap titik ke bola yang berpusat di tengah objek memberikan mapping injektif, yang di sini digunakan untuk menempatkan tekstur peta yang sama seperti yang digunakan untuk gambar globe. Perhatikan bahwa area diperbesar (titik permukaan berkumpul bersama dalam ruang tekstur) di mana permukaan jauh dari pusat, dan area menyusut di mana permukaan lebih dekat ke pusat.

Gambar 11.9 Vas jauh dari bola yang proyeksi bolanya menghasilkan banyak distorsi (kiri) dan proyeksi silinder menghasilkan hasil yang sangat baik di permukaan luar.

Cubemaps

Menggunakan koordinat bola untuk parameterisasi sebagai bentuk bulat atau seperti bola menyebabkan distorsi tinggi pada bentuk dan area di dekat kutub, yang sering mengarah ke artefak yang terlihat yang mengungkapkan bahwa ada dua titik khusus di mana ada sesuatu yang salah dengan tekstur. Sebuah alternatif yang populer jauh lebih seragam dengan biaya diskontinuitas yang lebih banyak. Idenya adalah untuk memproyeksikan ke sebuah kubus, bukan bola, dan kemudian menggunakan enam tekstur persegi terpisah untuk enam wajah

dari kubus. Kumpulan enam tekstur persegi disebut peta kubus. Ini menimbulkan diskontinuitas di sepanjang tepi kubus, tetapi distorsi bentuk dan area tetap rendah.

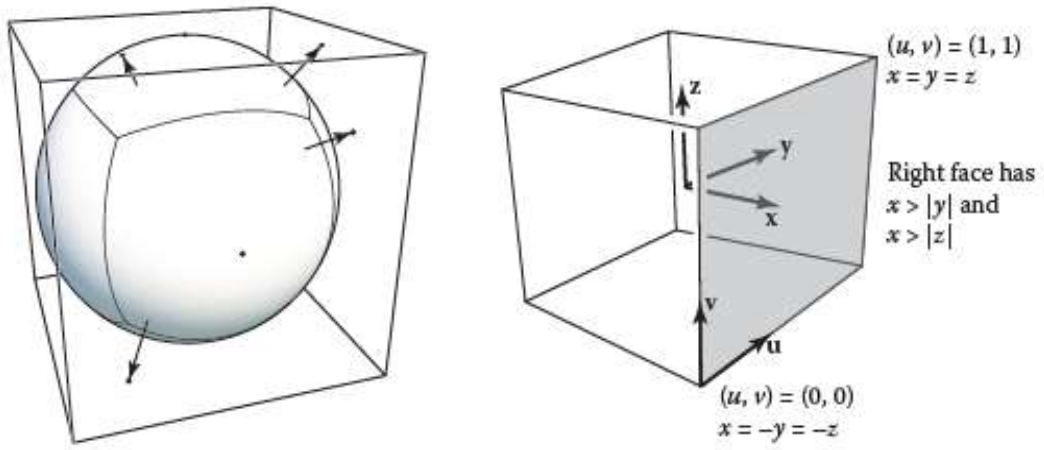
Menghitung koordinat tekstur peta kubus juga lebih murah daripada koordinat bola, karena memproyeksikan ke bidang hanya memerlukan pembagian—pada dasarnya sama dengan proyeksi perspektif untuk dilihat. Misalnya, untuk titik yang diproyeksikan ke permukaan +z kubus:

$$(x, y, z) \rightarrow \left(\frac{x}{z}, \frac{y}{z}\right)$$

Aspek yang membingungkan dari peta kubus adalah menetapkan konvensi tentang bagaimana arah dan arah ditentukan pada enam wajah. Konvensi apa pun baik-baik saja, tetapi konvensi yang dipilih memengaruhi isi tekstur, jadi standarisasi itu penting. Karena peta kubus sangat sering digunakan untuk tekstur yang dilihat dari bagian dalam kubus (lihat mapping lingkungan di Bagian 11.4.5), konvensi biasanya memiliki sumbu u dan v yang berorientasi sehingga u searah jarum jam dari v seperti yang dilihat dari dalam. Konvensi yang digunakan oleh OpenGL adalah

$$\phi{-}x(x, y, z) = 1/2[1 + (+z, -y) / |x|],$$
$$\phi{+}x(x, y, z) = 1/2\ [1 + (-z, -y) / |x|],$$
$$\phi{-}y(x, y, z) = ½\ [1 + (+x, -z) / |y|],$$
$$\phi{+}y(x, y, z) = ½\ [1 + (+x, +z) / |y|],$$
$$\phi{-}z(x, y, z) = ½\ [1 + (-x, -y) / |z|],$$
$$\phi{+}z(x, y, z) = ½\ [1 + (+x, -y) / |z|].$$

Subskrip menunjukkan wajah kubus mana yang sesuai dengan setiap proyeksi. Misalnya, ϕ_{-x} digunakan untuk titik-titik yang diproyeksikan ke muka kubikx = +1 . Anda dapat mengetahui wajah mana yang diproyeksikan oleh titik dengan melihat koordinat dengan nilai absolut terbesar: misalnya, jika |x| > |y| dan |x| > |z|, titik diproyeksikan ke muka +x or−x, tergantung pada tanda x.



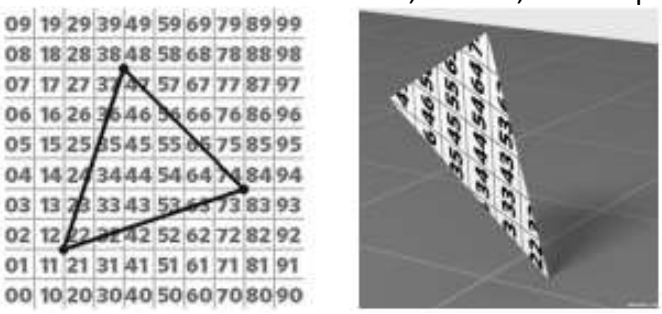
Gambar 11.10 Sebuah permukaan diproyeksikan ke dalam peta kubus. Titik-titik di permukaan diproyeksikan keluar dari pusat, masing-masing memetakan ke satu titik di salah satu dari enam wajah.

Tekstur untuk digunakan dengan peta kubus enam buah persegi. (Lihat Gambar 11.10.) Seringkali mereka dikemas bersama dalam satu gambar untuk penyimpanan, diatur seolah-olah kubus itu dibuka.

Koordinat Tekstur Interpolasi

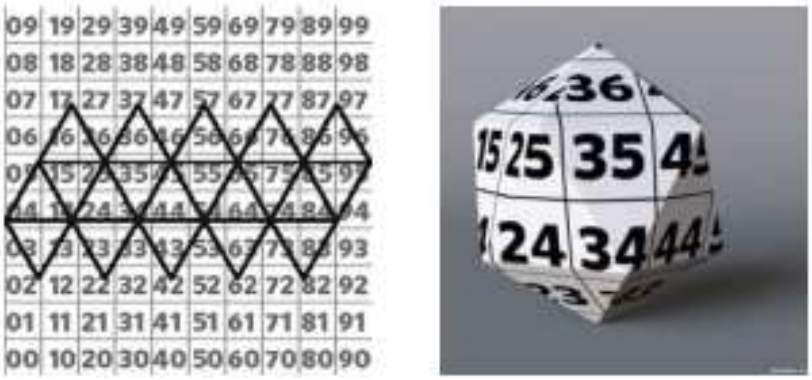
Untuk kontrol yang lebih halus atas fungsi koordinat tekstur pada permukaan mesh segitiga, Anda dapat secara eksplisit menyimpan koordinat tekstur di setiap titik, dan

menginterpolasinya melintasi segitiga menggunakan interpolasi barycentric (Bagian 8.1.2). Ini bekerja dengan cara yang persis sama seperti kuantitas lain yang bervariasi dengan lancar yang mungkin Anda definisikan melalui mesh: warna, normal, bahkan posisi 3D itu sendiri.



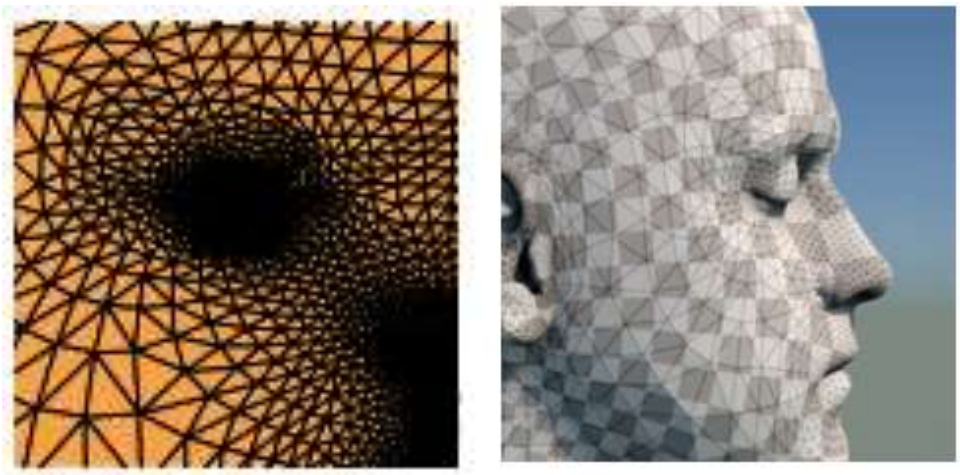
Gambar 11.11 Segitiga tunggal menggunakan koordinat tekstur interpolasi linier. Kiri: segitiga yang digambar dalam ruang tekstur; kanan: segitiga yang ditampilkan dalam scene 3D.

Mari kita lihat contoh dengan segitiga tunggal. Gambar 11.11 menunjukkan tekstur segitiga yang dipetakan dengan bagian dari pola uji yang sekarang sudah dikenal. Dengan melihat pola yang muncul pada segitiga yang dirender, Anda dapat menyimpulkan bahwa koordinat tekstur dari ketiga simpul tersebut adalah (0.2, 0.2), (0.8, 0.2), dan (0.2, 0.8), karena itulah titik-titik dalam tekstur yang muncul di tiga sudut segitiga. Sama seperti mapping yang ditentukan secara geometris di bagian sebelumnya, kami mengontrol di mana tekstur pergi ke permukaan dengan memberikan mapping dari permukaan ke domain tekstur, dalam hal ini dengan menentukan di mana setiap titik harus masuk dalam ruang tekstur. Setelah Anda memposisikan simpul, interpolasi linier (barycentric) melintasi segitiga menangani sisanya.



Gambar 11.12 Sebuah icosahedron dengan segitiganya ditata di ruang tekstur untuk memberikan distorsi nol tetapi dengan banyak jahitan.

Pada Gambar 11.12 kami menunjukkan cara umum untuk memvisualisasikan koordinat tekstur pada keseluruhan mesh: cukup menggambar segitiga di ruang tekstur dengan posisi simpul pada koordinat teksturnya. Visualisasi ini menunjukkan kepada Anda bagian tekstur mana yang digunakan oleh segitiga mana, dan ini adalah alat yang berguna untuk mengevaluasi koordinat tekstur dan untuk men-debug semua jenis kode mapping tekstur.



Gambar 11.13 Model wajah, dengan koordinat tekstur yang ditetapkan untuk mencapai distorsi bentuk yang cukup rendah, tetapi masih menunjukkan distorsi area sedang.

Kualitas mapping koordinat tekstur yang ditentukan oleh koordinat tekstur titik bergantung pada koordinat apa yang ditetapkan ke titik—yaitu, bagaimana mesh diletakkan di ruang tekstur. Tidak peduli koordinat apa yang ditetapkan, selama segitiga di simpul berbagi mesh (Bagian 12.1), mapping koordinat tekstur selalu kontinu, karena segitiga tetangga akan setuju pada koordinat tekstur pada titik-titik di tepi bersama mereka. Tetapi kualitas-kualitas lain yang diinginkan yang dijelaskan di atas tidak begitu otomatis. Injektivitas berarti segitiga tidak tumpang tindih dalam ruang tekstur—jika demikian, berarti ada beberapa titik dalam tekstur yang akan muncul di lebih dari satu tempat di permukaan.

Distorsi ukuran rendah ketika area segitiga dalam ruang tekstur sebanding dengan areanya dalam 3D. Misalnya, jika wajah karakter dipetakan dengan fungsi koordinat tekstur kontinu, salah satu sering berakhir dengan kemudian dijepit ke dalam area yang relatif kecil dalam ruang tekstur, seperti yang ditunjukkan pada Gambar 11.13. Meskipun segitiga di hidung lebih kecil daripada di pipi, rasio ukurannya lebih ekstrem di ruang tekstur. Hasilnya adalah tekstur pada hidung membesar, karena area tekstur yang kecil harus menutupi area permukaan yang luas. Demikian pula, membandingkan dahi dengan pelipis, segitiga memiliki ukuran yang serupa dalam 3D, tetapi segitiga di sekitar pelipis lebih besar dalam ruang tekstur, menyebabkan tekstur tampak lebih kecil di sana.

Demikian pula, distorsi bentuk rendah ketika bentuk segitiga serupa dalam 3D dan dalam ruang tekstur. Contoh wajah memiliki distorsi bentuk yang cukup rendah, tetapi, misalnya, bola pada Gambar 11.17 memiliki distorsi bentuk yang sangat besar di dekat kutub.

Tiling, Mode wrapping, dan Transformasi Tekstur

Seringkali berguna untuk mengizinkan koordinat tekstur keluar dari batas gambar tekstur. Terkadang ini adalah detailnya: kesalahan pembulatan dalam perhitungan koordinat tekstur dapat menyebabkan titik yang mendarat tepat pada batas tekstur menjadi sedikit di luar, dan mesin mapping tekstur tidak boleh gagal dalam kasus itu. Tapi juga bisa menjadi alat pemodelan.

Jika tekstur seharusnya hanya menutupi sebagian permukaan, tetapi koordinat tekstur sudah diatur untuk memetakan seluruh permukaan ke unit persegi, salah satu opsi adalah menyiapkan gambar tekstur yang sebagian besar kosong dengan konten di area kecil. Tapi itu mungkin memerlukan gambar tekstur resolusi sangat tinggi untuk mendapatkan detail yang cukup di area yang relevan. Alternatif lain adalah memperbesar semua koordinat tekstur sehingga mencakup rentang yang lebih besar—[−4.5, 5.5] × [−4.5, 5.5] misalnya, untuk memposisikan unit persegi pada ukuran sepersepuluh di tengah permukaan.

Untuk kasus seperti ini, pencarian tekstur di luar area satuan persegi yang dicakup oleh gambar tekstur harus mengembalikan warna latar belakang yang konstan. Salah satu cara untuk melakukannya adalah dengan mengatur warna latar belakang yang akan dikembalikan oleh pencarian tekstur di luar unit persegi. Jika gambar tekstur sudah memiliki warna latar belakang yang konstan (misalnya, logo di atas latar belakang putih), cara lain untuk memperluas latar belakang ini secara otomatis di atas bidang adalah dengan mengatur pencarian di luar satuan persegi untuk mengembalikan warna gambar tekstur di titik terdekat di tepi, dicapai dengan menjepit koordinat u dan v ke kisaran dari piksel pertama hingga piksel terakhir dalam gambar.

Terkadang kita menginginkan pola yang berulang, seperti papan catur, lantai keramik, atau dinding bata. Jika pola berulang pada kotak persegi panjang, akan sia-sia jika membuat gambar dengan banyak salinan dari data yang sama. Sebagai gantinya, kita dapat menangani pencarian tekstur di luar gambar tekstur menggunakan pengindeksan sampul—ketika titik pencarian keluar dari tepi kanan gambar tekstur, titik pencarian itu membungkus ke tepi kiri. Ini ditangani dengan sangat sederhana menggunakan operasi sisa bilangan bulat pada koordinat piksel.

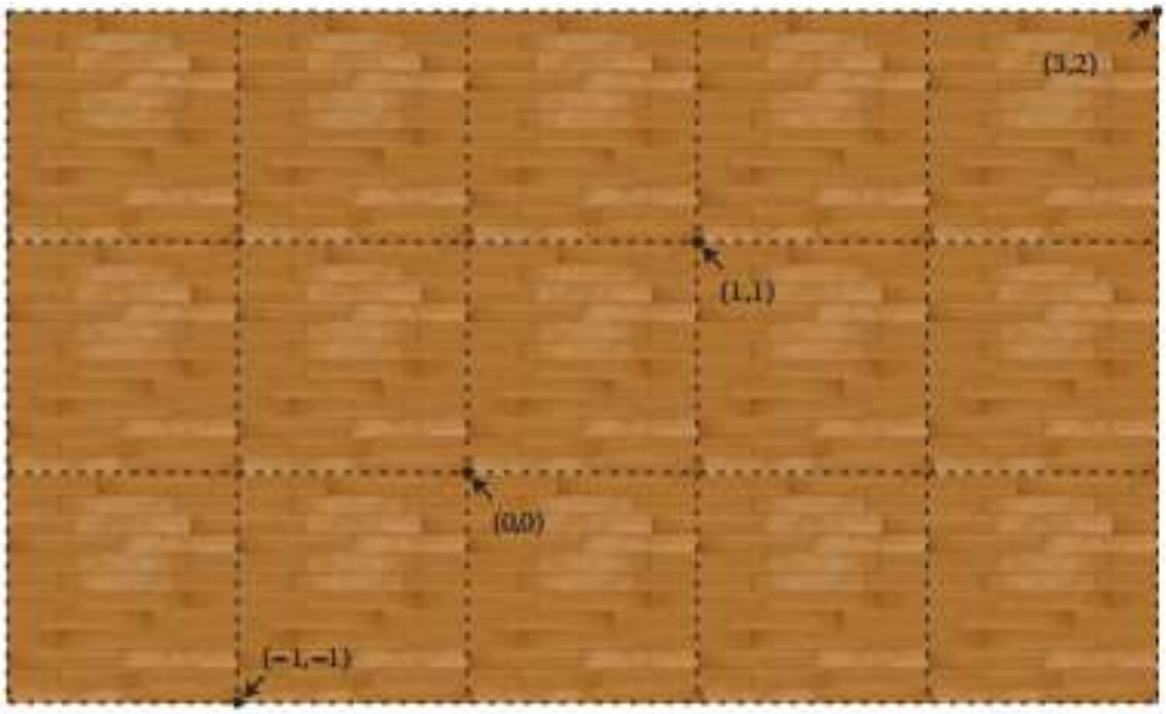
```
Color texture_lookup_wrap(Texture t, float u, float v) {
        int i = round(u * t.width() - 0.5)
        int j = round(v * t.height() - 0.5)
        return t.get_pixel(i % t.width(), j % t.height())
}
Color texture_lookup_wrap(Texture t, float u, float v) {
        int i = round(u * t.width() - 0.5)
        int j = round(v * t.height() - 0.5)
        return t.get_pixel(max(0, min(i, t.width()-1)),
                                                (max(0, min(j, t.height()-1))))
}
```

Pilihan antara dua cara menangani pencarian di luar batas ini ditentukan dengan memilih mode wrappingan dari daftar yang mencakup tiling, penjepitan, dan seringkali kombinasi atau varian dari keduanya. Dengan mode wrapping, kita dapat dengan bebas memikirkan tekstur sebagai fungsi yang mengembalikan warna untuk setiap titik di bidang2D tak berhingga(Gambar 11.14). Ketika kita menentukan tekstur menggunakan gambar, mode ini menjelaskan bagaimana data gambar hingga seharusnya digunakan untuk mendefinisikan fungsi ini. Dalam Bagian 11.5, kita akan melihat bahwa tekstur prosedural secara alami dapat meluas melintasi bidang tak berhingga, karena teksturnya tidak dibatasi oleh data gambar berhingga. Karena keduanya secara logika tidak terbatas luasnya, kedua jenis tekstur tersebut dapat dipertukarkan.

Saat menyesuaikan skala dan penempatan tekstur, sebaiknya hindari benar-benar mengubah fungsi yang menghasilkan koordinat tekstur, atau nilai koordinat tekstur yang tersimpan di simpul mesh, dengan menerapkan transformasi matriks ke koordinat tekstur sebelum menggunakannya untuk sampel tekstur:

$$\phi(\mathbf{x}) = \mathbf{M}T\,\phi\text{model}(\mathbf{x}),$$

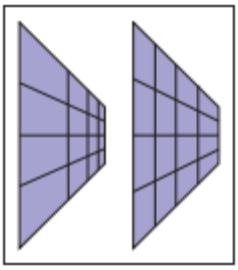
di mana model adalah fungsi koordinat tekstur yang disediakan dengan model, dan MT adalah matriks 3 oleh 3 yang mewakili transformasi proyektif atau afin dari koordinat tekstur 2 D menggunakan koordinat homogen. Transformasi seperti itu, terkadang terbatas hanya pada scalling dan/atau terjemahan, didukung oleh sebagian besar perender yang menggunakan mapping tekstur.



Gambar 11.14 Tekstur lantai kayu ditata di atas ruang tekstur dengan membungkus koordinat texel.

Perspektif Interpolasi yang Benar

Ada beberapa seluk-beluk dalam mencapai perspektif yang tampak benar dengan menginterpolasi koordinat tekstur di seluruh segitiga, tetapi kita dapat mengatasi ini pada tahap asterisasi. Alasan mengapa hal-hal ini tidak mudah adalah bahwa hanya dengan menginterpolasi koordinat tekstur dalam ruang layar akan menghasilkan gambar yang salah, seperti yang ditunjukkan untuk tekstur grid pada Gambar 11.15. Karena hal-hal dalam perspektif menjadi lebih kecil seiring dengan bertambahnya jarak ke penampil, garis-garis yang berjarak sama dalam 3D harus dikompresi dalam ruang gambar 2D. Interpolasi koordinat tekstur yang lebih hati-hati diperlukan untuk mencapai hal ini.



Gambar 11.15 Kiri: perspektif yang benar. Kanan: interpolasi dalam ruang layar.

Kita dapat mengimplementasikan mapping tekstur pada segitiga dengan menginterpolasi koordinat (u,v), memodifikasi metode rasterisasi dari Bagian 8.1.2, tetapi ini menghasilkan masalah yang ditunjukkan di sebelah kanan Gambar 11.15. Masalah serupa terjadi untuk segitiga jika koordinat barycentric ruang layar digunakan seperti dalam kode rasterisasi berikut:

for all *x* **do**
for all *y* **do**
compute (α, β, γ) for (*x*, *y*)

if $\alpha \in (0, 1)$ and $\beta \in (0, 1)$ dan $\gamma \in (0, 1)$ **then**
t = α**t**0 + β**t**1 + γ**t**2
drawpixel (*x, y*) with color texture(**t**) untuk solid texture
or with texture(β, γ) untuk 2D texture.

Kode ini akan menghasilkan gambar, tetapi ada masalah. Untuk memecahkan masalah dasar, mari kita pertimbangkan perkembangan dari ruang dunia q ke titik homogen ke titik homogen s:

$$\begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix} \xrightarrow{\text{transform}} \begin{bmatrix} x_r \\ y_r \\ z_r \\ h_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x_r/h_r \\ y_r/h_r \\ z_r/h_r \\ 1 \end{bmatrix} \equiv \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}$$

Bentuk paling sederhana dari masalah interpolasi koordinat tekstur adalah ketika kita memiliki koordinat tekstur (u,v) yang terkait dengan dua titik, q dan Q, dan kita perlu membuat koordinat tekstur pada gambar sepanjang garis antara s dan S. Jika titik ruang-dunia q yang berada di garis antara q dan Q memproyeksikan ke titik screenspace s pada garis antara s dan S, maka kedua titik tersebut harus memiliki koordinat tekstur yang sama.

Kemudian pendekatan vescreen-space, yang diwujudkan oleh algoritma di atas, mengatakan bahwa pada titik s'= s + α(S − s) kita harus menggunakan koordinat tekstur u_s + α(u_S − u_s) dan v_s + (v_S – v_s). Ini tidak bekerja dengan benar karena titik-dunia q yang berubah menjadi s bukanlah q + α (Q − q). Namun, kita tahu dari Bagian 7.4 bahwa titik-titik pada ruas garis antara q dan Q memang berakhir di suatu tempat pada ruas garis antara s dan S; sebenarnya, di bagian itu kami menunjukkan bahwa

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}) \rightarrow \mathbf{s} + \alpha(\mathbf{S} + \mathbf{s}).$$

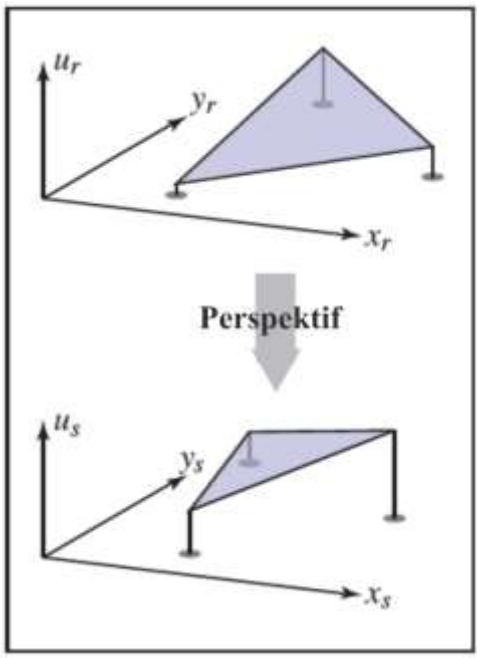
Parameter interpolasi t dan tidak sama, tetapi kita dapat menghitung satu dari yang lain:[6]

$$t(\alpha) = \frac{w_r \alpha}{w_R + \alpha(w_r - w_R)} \, dan \alpha(t) = \frac{w_R t}{w_r + t(w_R - w_r)}$$

Persamaan ini memberikan satu kemungkinan perbaikan untuk ide interpolasi ruang-layar. Untuk mendapatkan koordinat tekstur untuk titik-layar s' = s + α(S − s), hitung u's = us + t(α) (uS − us) dan v's = vs + t(α) (vS − vs) . Ini adalah koordinat titik q yang dipetakan ke s , jadi ini akan berhasil. Namun, lambat untuk mengevaluasi t(α) untuk setiap fragmen, dan ada cara yang lebih sederhana.

[6] Penting untuk menurunkan sendiri fungsi-fungsi ini dari Persamaan (7.6); dalam notasi bab tersebut, = f(t).

Pengamatan kuncinya adalah karena, seperti yang kita ketahui, transformasi perspektif mempertahankan garis dan bidang, aman untuk menginterpolasi atribut apa pun yang kita inginkan di seluruh segitiga secara linier, tetapi hanya selama atribut tersebut melalui transformasi perspektif bersama dengan titiknya. Untuk mendapatkan intuisi geometrik untuk ini, kurangi dimensi sehingga kita memiliki titik homogen (x_r, y_r, w_r) dan satu atribut yang diinterpolasi. Atribut tersebut dianggap sebagai fungsi linier dari x_r dan y_r, jadi jika kita memplotuasa delapan bidang di atas (x_r, y_r) ada bidang datar. Sekarang, jika kita menganggap u sebagai koordinat spasial ketiga (sebut saja ur untuk menekankan bahwa itu diperlakukan sama seperti yang lain) dan mengirim seluruh3D titik homogen (x_r, y_r, u_r, w_r) melalui transformasi perspektif, hasilnya (x_s, y_s, u_s) masih menghasilkan titik-titik yang terletak pada bidang. Akan ada beberapa lengkungan di dalam pesawat, tetapi pesawat tetap datar. Ini berarti bahwa kita adalah fungsi linier dari (x_s, y_s)—artinya, kita dapat menghitung kita di mana saja dengan menggunakan interpolasi linier berdasarkan koordinat (x_s, y_s).



Gambar 11.16 Penalaran geometris untuk interpolasi ruang layar. Atas: ur akan diinterpolasi sebagai fungsi linier dari (x_r,y_r). Bawah: setelah transformasi perspektif dari (x_r, y_r, u_r, w_r) menjadi (x_s, y_s, u_s, 1), u_s adalah fungsi linier dari (x_s, y_s).

Kembali ke masalah penuh, kita perlu menginterpolasi koordinat tekstur (u, v) yang merupakan fungsi linier dari koordinat ruang dunia (x_q, y_q, z_q). Setelah mengubah titik menjadi ruang layar, dan menambahkan koordinat tekstur seolah-olah itu adalah koordinat tambahan, kita memiliki (Persamaan 11.2)

$$\begin{bmatrix} u \\ v \\ 1 \\ x_r \\ y_r \\ z_r \\ w_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} u/w_r \\ v/w_r \\ 1/w_r \\ x_r/w_r = x_s \\ y_r/w_r = y_s \\ z_r/w_r = z_s \\ 1 \end{bmatrix}.$$

Implikasi praktis dari paragraf sebelumnya adalah bahwa kita dapat melanjutkan dan menginterpolasi semua besaran ini berdasarkan nilai (x_s, y_s)—termasuk nilai z_s, yang digunakan dalam buffer-z. Masalah dengan pendekatan naïve adalah bahwa kita menginterpolasi komponen yang dipilih secara tidak konsisten—selama kuantitas yang terlibat berasal dari sebelum atau semua dari setelah pembagian perspektif, semuanya akan baik-baik saja.

Satu masalah yang tersisa adalah bahwa (u/w_r, v/w_r) tidak secara langsung berguna untuk mencari data tekstur; kita membutuhkan (u,v). Ini menjelaskan tujuan dari parameter tambahan yang kita masukkan (11.2), yang nilainya selalu 1: setelah kita memiliki u/w_r, v/w_r, dan $1/w_r$, kita dapat dengan mudah memulihkan (u, v) dengan membagi.

Untuk memverifikasi bahwa ini semua benar, mari kita periksa bahwa interpolasi kuantitas 1/wr di ruang layar memang menghasilkan kebalikan dari interpolasi wr di ruang dunia. Untuk melihat ini benar, konfirmasikan (Latihan 2):

$$\frac{1}{w_r} + a(t)\left(\frac{1}{w_R} - \frac{1}{w_r}\right) = \frac{1}{w'_r} = \frac{1}{w_r + t(w_R - w_r)}$$

Kemampuan untuk menginterpolasi $1/w_r$ secara linier tanpa kesalahan dalam ruang yang diubah memungkinkan kita untuk membuat tekstur segitiga dengan benar. Kita dapat menggunakan fakta ini untuk memodifikasi kode scan-konversi untuk tiga titik $t_i = (x_i, y_i, z_i, w_i)$ yang telah melewati matriks tampilan, tetapi belum dihomogenisasi, lengkap dengan koordinat tekstur $t_i = (u_i, v)$:

for all *xs* **do**
for all *ys* **do**
compute (*α, β, γ*) for (*xs, ys*)
if (*α* ∈ [0, 1] dan *β* ∈ [0, 1] dan *γ* ∈ [0, 1]) **then**
us = *α*(*u*0/*w*0) + *β*(*u*1/*w*1) + *γ*(*u*2/*w*2)
vs = *α*(*v*0/*w*0) + *β*(*v*1/*w*1) + *γ*(*v*2/*w*2)
1*s* = *α*(1/*w*0) + *β*(1/*w*1) + *γ*(2/*w*2)
u = *us*/1*s*
v = *vs*/1*s*
drawpixel (*xs, ys*) with color texture(*u, v*)

Tentu saja, banyak ekspresi yang muncul dalam kodesemu ini akan dikomputasi di luar loop untuk kecepatan. Untuk tekstur padat, cukup sederhana untuk memasukkan koordinat ruang dunia asli xq,yq,zq dalam daftar atribut, diperlakukan sama seperti u dan v, dan koordinat ruang dunia interpolasi yang benar akan diperoleh, yang dapat diteruskan ke fungsi tekstur padat.

Kontinuitas dan Jahitan

Meskipun distorsi dan kontinuitas rendah adalah sifat yang bagus untuk memiliki fungsi koordinat tekstur, diskontinuitas seringkali tidak dapat dihindari. Untuk permukaan 3D tertutup apa pun, ini adalah hasil dasar topologi bahwa tidak ada fungsi bijektif kontinu yang memetakan seluruh permukaan menjadi gambar tekstur. Sesuatu harus diberikan, dan dengan memperkenalkan jahitan—kurva pada permukaan di mana koordinat tekstur berubah secara tiba-tiba—kita dapat memiliki distorsi rendah di tempat lain. Banyak mapping yang ditentukan secara geometris yang dibahas di atas sudah mengandung jahitan: dalam koordinat bola dan silinder, jahitannya adalah di mana sudut yang dihitung oleh atan2 melingkari dari π ke - π, dan dalam peta kubus, jahitannya berada di sepanjang tepi kubus, di mana mapping beralih di antara enam tekstur persegi.

Gambar 11.17 Bola poligonal: di sebelah kiri, dengan semua simpul bersama, fungsi koordinat tekstur kontinu, tetapi selalu memiliki masalah dengan segitiga yang melintasi meridian ke-180, karena koordinat tekstur diinterpolasi dari garis bujur di dekat 180 hingga garis bujur di dekat 180. Di sebelah kanan, beberapa simpul diduplikasi, dengan posisi 3D yang identik tetapi koordinat tekstur berbeda persis 360 derajat dalam garis bujur, sehingga koordinat tekstur diinterpolasi melintasi meridian daripada di sepanjang peta.

Dengan koordinat tekstur yang diinterpolasi, jahitan memerlukan pertimbangan khusus, karena tidak terjadi secara alami. Kami mengamati sebelumnya bahwa koordinat tekstur yang diinterpolasi secara otomatis kontinu pada mesh simpul bersama — jaminan berbagi koordinat tekstur. Tetapi ini berarti bahwa segitiga lemak merentang, dengan beberapa simpul di satu sisi dan yang lain, mesin interpolasi akan dengan senang hati menyediakan mapping terus menerus, tetapi kemungkinan akan sangat terdistorsi atau terlipat sehingga tidak injektif. Gambar 11.17 mengilustrasikan masalah ini pada globe yang dipetakan dengan koordinat bola.

Satu-satunya cara untuk membuat transisi yang jelas adalah untuk menghindari pembagian koordinat tekstur pada jahitan: segitiga yang melintasi Selandia Baru perlu diinterpolasi ke bujur +181, dan segitiga berikutnya di Pasifik perlu melanjutkan mulai dari bujur−179. Untuk melakukan ini, kami menduplikasi simpul pada jahitan: untuk setiap simpul kami menambahkan simpul kedua dengan bujur yang setara, berbeda 360◦, dan segitiga di sisi-sisi yang berlawanan dari laut menggunakan simpul yang berbeda. Solusi ini ditunjukkan di bagian kanan Gambar 11.17, di mana simpul di paling kiri dan kanan dari ruang tekstur adalah duplikat, dengan posisi 3D yang sama.

11.3 PENCARIAN TEKSTUR ANTIALIASING

Masalah mendasar kedua dari texturemapping adalah antialiasing. Rendering gambar yang dipetakan tekstur adalah proses pengambilan sampel: memetakan tekstur ke permukaan dan kemudian memproyeksikan permukaan ke dalam gambar menghasilkan fungsi 2D melintasi bidang gambar, dan kami mengambil sampelnya dalam piksel. Seperti yang kita lihat di Bab 9, melakukan ini dengan menggunakan sampel titik akan menghasilkan aliasing artefak ketika gambar mengandung detail atau tepi yang tajam—dan karena seluruh inti tekstur adalah untuk memperkenalkan detail, tekstur menjadi sumber utama masalah alias seperti yang kita lihat di Gambar 11.3.

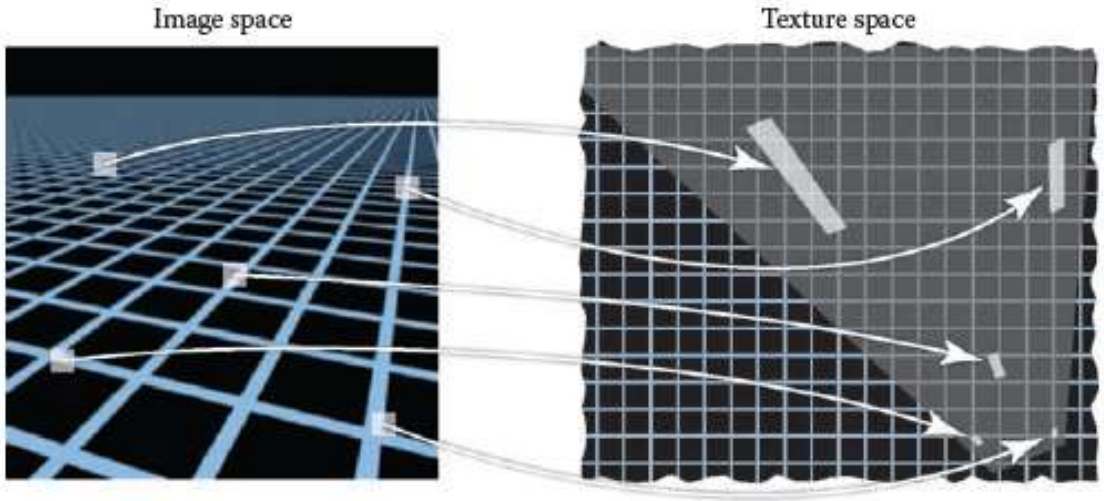
Sama seperti rasterisasi antialias dari garis atau segitiga, ray tracing antialias (Bagian 13.4), atau downsampling gambar (Bagian 9.4), solusinya adalah membuat setiap piksel tidak titik sampel tetapi area rata-rata gambar, lebih dari ukuran yang sama dengan piksel. Menggunakan pendekatan supersampling yang sama yang digunakan untuk rasterisasi antialias dan penelusuran sinar, dengan sampel yang cukup, hasil yang sangat baik dapat diperoleh tanpa perubahan pada mesin mapping tekstur: banyak sampel dalam area piksel akan mendarat di tempat berbeda di peta tekstur, dan rata-rata hasil bayangan dihitung dengan menggunakan pencarian tekstur yang berbeda dan cara yang akurat untuk memperkirakan warna rata-rata gambar di atas piksel. Namun, dengan tekstur yang detail dibutuhkan sangat banyak sampel untuk mendapatkan hasil yang baik, yang lambat. Menghitung rata-rata area ini secara efisien dengan adanya tekstur di permukaan adalah topik kunci pertama dalam antialiasing tekstur.

Gambar tekstur biasanya didefinisikan oleh gambar raster, jadi ada juga masalah rekonstruksi yang harus dipertimbangkan, seperti halnya dengan gambar upsampling (Bagian 9.4). Solusinya sama untuk tekstur: gunakan filter rekonstruksi untuk menginterpolasi antar teks. Kami memperluas masing-masing topik ini di bagian berikut.

Jejak Piksel

Apa yang membuat tekstur antialiasing lebih kompleks daripada jenis antialiasing lainnya adalah bahwa hubungan antara gambar yang dirender dan tekstur terus berubah. Setiap nilai piksel harus dihitung sebagai warna rata-rata di atas area yang termasuk dalam piksel dalam gambar, dan dalam kasus umum bahwa piksel melihat satu permukaan, ini sesuai dengan rata-rata di atas area di permukaan. Jika warna permukaan berasal dari tekstur, ini

pada gilirannya berarti rata-rata pada bagian tekstur yang sesuai, yang dikenal sebagai jejak ruang tekstur piksel. Gambar 11.18 mengilustrasikan bagaimana jejak area persegi (yang dapat berupa area piksel dalam gambar beresolusi lebih rendah) dipetakan ke area dengan ukuran dan bentuk yang sangat berbeda dalam ruang tekstur lantai.



Gambar 11.18 Jejak kaki di ruang tekstur area persegi berukuran identik dalam gambar bervariasi dalam ukuran dan bentuk di seluruh gambar.

Ingat tiga ruang yang terlibat dalam rendering dengan tekstur: proyeksi π yang memetakan titik 3D ke dalam gambar dan fungsi koordinat tekstur ∅ yang memetakan titik 3D ke dalam ruang tekstur. Untuk bekerja dengan pixel footprints kita perlu memahami komposisi kedua mapping ini: pertama-tama ikuti mundur π untuk mendapatkan dari gambar ke permukaan, kemudian ikuti maju. Komposisi ini $\psi=\varphi \circ \pi^{-1}$ yang menentukan jejak piksel: jejak piksel adalah gambar area persegi piksel itu dari gambar di bawah mapping ψ.

Masalah inti dalam antialiasing tekstur adalah menghitung nilai rata-rata tekstur di atas tapak piksel. Untuk melakukan ini secara umum bisa menjadi pekerjaan yang cukup rumit: untuk objek yang jauh dengan bentuk permukaan yang rumit, tapak dapat berupa bentuk rumit yang mencakup area yang luas, atau mungkin beberapa area yang tidak terhubung, dalam ruang tekstur. Namun dalam kasus tipikal, sebuah piksel mendarat di area permukaan yang halus yang dipetakan ke satu area di tekstur.

Karena ψ berisi mapping dari gambar ke permukaan dan mapping dari permukaan ke tekstur, ukuran dan bentuk tapak bergantung pada situasi tampilan dan fungsi koordinat tekstur. Saat permukaan lebih dekat ke kamera, jejak piksel akan lebih kecil; ketika permukaan yang sama bergerak lebih jauh, jejaknya semakin besar. Ketika permukaan dilihat pada sudut miring, tapak piksel pada permukaan memanjang, yang biasanya berarti juga akan memanjang di ruang tekstur. Bahkan dengan tampilan tetap, fungsi koordinat tekstur dapat menyebabkan variasi pada footprint: jika mendistorsi area, ukuran footprint akan bervariasi, dan jika mendistorsi bentuk, mereka dapat memanjang bahkan untuk tampilan permukaan.

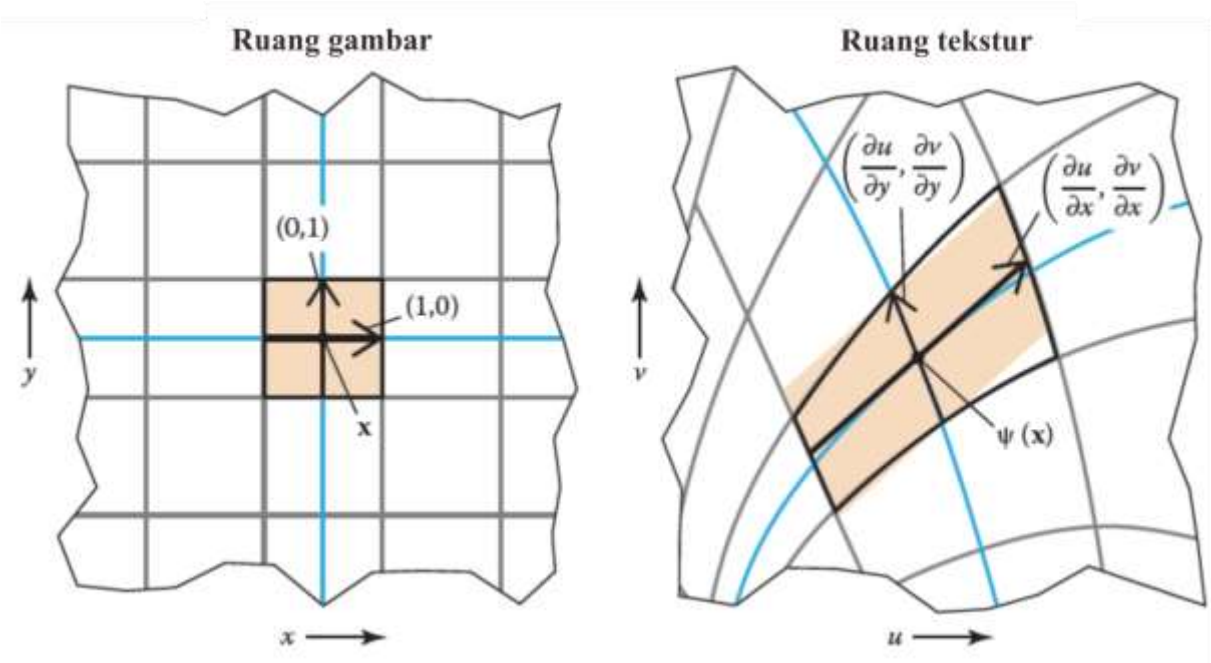
Namun, untuk menemukan algoritma yang efisien untuk menghitung pencarian antialias, beberapa pendekatan substansial akan diperlukan. Ketika suatu fungsi mulus, pendekatan linier sering berguna. Dalam kasus antialiasing tekstur, ini berarti mendekati mapping ψ dari ruang gambar ke ruang tekstur sebagai mapping linier dari 2D ke 2D:

$$\psi(\mathbf{x}) = \psi(\mathbf{x}0) + \mathbf{J}(\mathbf{x} - \mathbf{x}0),$$

di mana matriks 2-kali-2 J adalah beberapa aproksimasi terhadap turunan dari ψ. Ini memiliki empat entri, dan jika kita menyatakan posisi ruang-gambar sebagai x = (x, y) dan posisi ruang-tekstur sebagai u = (u, v) maka

$$M = \begin{bmatrix} \frac{du}{dx} & \frac{du}{dy} \\ \frac{dv}{dx} & \frac{dv}{dy} \end{bmatrix}$$

dimana keempat turunan tersebut menggambarkan bagaimana titik tekstur (u, v) yang terlihat pada titik (x, y) pada gambar berubah ketika kita mengubah x dan y.



Gambar 11.19 Perkiraan jejak tekstur-ruang dari suatu piksel dapat dibuat menggunakan turunan mapping dari (x, y) ke (u, v). Turunan parsial terhadap x dan y sejajar dengan gambar isoline x dan y (berwarna biru) dan merentang jajar genjang (diarsir dalam warna oranye) yang mendekati bentuk lengkung dari tapak kaki yang tepat (diuraikan dalam warna hitam).

Interpretasi geometris dari aproksimasi ini adalah bahwa ia mengatakan piksel persegi berukuran satuan yang berpusat di x pada gambar akan dipetakan kira-kira ke jajar genjang di ruang tekstur, berpusat di ψ(x) dan dengan tepinya sejajar dengan vektor u_x = (du/dx, dv/dx) dan u_y = (du/dy, dv/dy).

Matriks turunan J berguna karena menceritakan keseluruhan cerita tentang variasi dalam (perkiraan) jejak tekstur-ruang di seluruh gambar. Derivatif yang lebih besar besarnya menunjukkan jejak tekstur-ruang yang lebih besar, dan hubungan antara vektor turunan ux dan uy menunjukkan bentuknya. Ketika mereka ortogonal dan sama panjang, tapaknya persegi, dan saat mereka menjadi miring dan/atau sangat berbeda panjangnya, tapak menjadi memanjang.

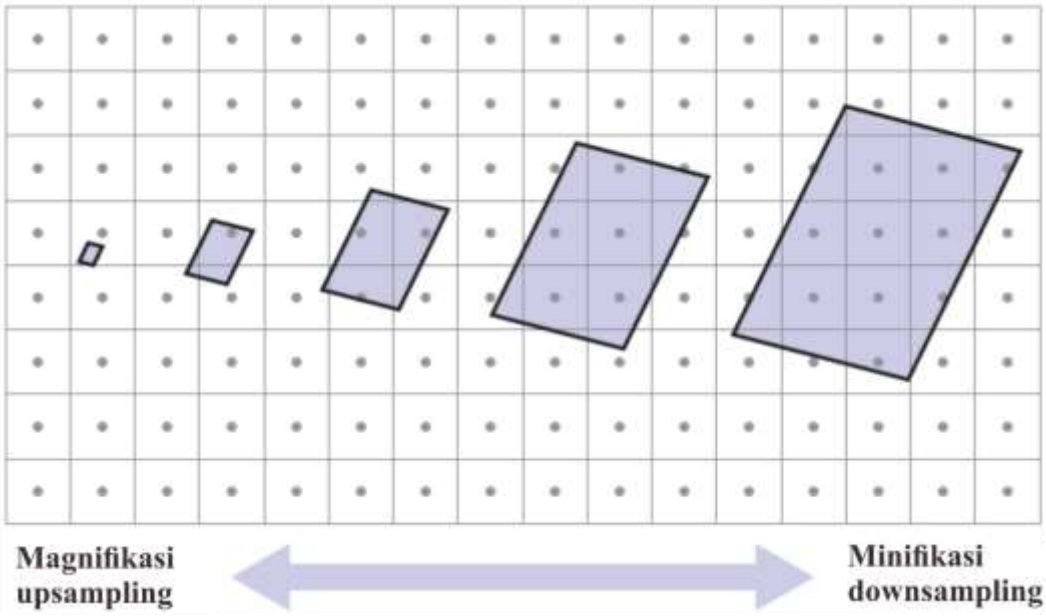
Kami sekarang telah mencapai bentuk masalah yang biasanya dianggap sebagai "jawaban yang benar": sampel tekstur yang disaring pada posisi ruang gambar tertentu harus menjadi nilai rata-rata peta tekstur di atas jejak berbentuk jajaran genjang yang ditentukan oleh tekstur koordinat turunan pada titik itu. Ini sudah memiliki beberapa asumsi—yaitu, bahwa mapping dari gambar ke tekstur halus—tetapi cukup akurat untuk kualitas gambar yang sangat baik. Namun, rata-rata luas jajaran genjang ini sudah terlalu mahal untuk dihitung

secara tepat, sehingga berbagai pendekatan digunakan. Pendekatan untuk tekstur antialiasing berbeda dalam kecepatan/kualitas pengorbanan yang mereka buat dalam mendekati pencarian ini. Kami membahas ini di bagian berikut.

Catatan : Pendekatan di sini menggunakan filter kotak untuk mengambil sampel gambar. Beberapa sistem malah menggunakan filter piksel Gaussian, yang menjadi Gaussian elips dalam ruang tekstur; ini adalah rata-rata tertimbang elips (EWA).

Rekonstruksi

Saat footprint lebih kecil dari texel, kami memperbesar tekstur saat dipetakan ke dalam gambar. Kasus ini analog dengan upsampling gambar, dan pertimbangan utama adalah interpolasi antara texel untuk menghasilkan gambar yang halus di mana grid texel tidak jelas. Sama seperti pada image upsampling, proses smoothing ini ditentukan oleh filter konstruksi yang digunakan untuk menghitung sampel tekstur pada lokasi sembarang dalam ruang tekstur. (Lihat Gambar 11.20.)



Gambar 11.20 Isu dominan dalam pemfilteran tekstur berubah dengan ukuran tapak. Untuk footprint kecil (kiri) diperlukan interpolasi antar piksel untuk menghindari artefak yang tidak jelas; untuk footprint yang besar, tantangannya adalah menemukan rata-rata banyak piksel secara efisien.

Pertimbangannya hampir sama seperti pada pengambilan sampel ulang gambar, dengan satu perbedaan penting. Dalam pengambilan sampel ulang gambar, tugasnya adalah menghitung sampel output pada kisi biasa, dan keteraturan itu memungkinkan optimalisasi penting dalam kasus filter rekonstruksi yang dapat dipisahkan. Dalam penyaringan tekstur, pola pencarian tidak teratur, dan sampel harus dihitung secara terpisah. Ini berarti filter rekonstruksi besar dan berkualitas tinggi sangat mahal untuk digunakan, dan untuk alasan ini filter kualitas tertinggi yang biasanya digunakan untuk tekstur adalah interpolasi bilinear.

Perhitungan sampel tekstur terinterpolasi abilinear adalah sama dengan menghitung satu piksel dalam sebuah gambar yang diambil sampelnya dengan interpolasi bilinear. Pertama kita nyatakan titik sampel ruang tekstur dalam bentuk koordinat texel (bernilai nyata), kemudian kita membaca nilai dari empat texel tetangga dan rata-ratanya. Tekstur biasanya diparameterisasi di atas persegi satuan, dan teksel ditempatkan dengan cara yang sama seperti piksel dalam gambar apa pun, berjarak $1/n_u$ terpisah dalam arah u dan $1/n_v$ dalam v,

dengan texel (0,0) diposisikan setengah texel dari tepi untuk simetri. (Lihat Bab 9 untuk penjelasan lengkapnya.)

```
Color tex_sample_bilinear(Texture t, float u, float v) {
        u_p = u * t.width - 0.5
        v_p = v * t.height - 0.5
        iu0 = floor(u_p); iu1 = iu0 + 1
        iv0 = floor(v_p); iv1 = iv0 + 1
        a_u = (iu1 - u_p); b_u = 1 - a_u
        a_v = (iv1 - v_p); b_v = 1 - a_v
        return a_u * a_v * t[iu0][iv0] + a_u * b_v * t[iu0][iv1] + b_u
        * a_v * t[iu1][iv0] + b_u * b_v * t[iu1][iv1]
}
```

Dalam banyak sistem, operasi ini menjadi hambatan kinerja yang penting, terutama karena latensi memori yang terlibat dalam pengambilan empat nilai texel dari data tekstur. Pola titik sampel untuk tekstur tidak beraturan, karena mapping dari gambar ke ruang tekstur bersifat arbitrer, tetapi sering kali koheren, karena titik gambar yang berdekatan cenderung memetakan ke titik tekstur terdekat yang dapat membaca teks yang sama. Untuk alasan ini, sistem berkinerja tinggi memiliki perangkat keras khusus yang dikhususkan untuk pengambilan sampel tekstur yang menangani interpolasi dan mengelola cache data tekstur yang baru saja digunakan untuk meminimalkan jumlah pengambilan data yang lambat dari memori tempat data tekstur disimpan.

Setelah membaca Bab 9 Anda mungkin mengeluh bahwa interpolasi linier mungkin tidak mulus dengan rekonstruksi yang cukup untuk beberapa aplikasi yang menuntut. Namun, itu selalu dapat dibuat cukup baik dengan meresampling tekstur ke resolusi yang agak lebih tinggi menggunakan filter yang lebih baik, sehingga teksturnya cukup halus sehingga interpolasi bilinear bekerja dengan baik.

Mipmapping

Melakukan pekerjaan interpolasi yang baik hanya cukup dalam situasi di mana tekstur diperbesar: di mana jejak piksel kecil dibandingkan dengan jarak texel. Ketika jejak piksel mencakup banyak texel, antialiasing yang baik memerlukan komputasi rata-rata banyak texel untuk menghaluskan sinyal sehingga dapat disampel dengan aman.

Salah satu cara yang sangat akurat untuk menghitung nilai tekstur rata-rata di atas jejak adalah dengan menemukan semua teks di dalam tapak dan menjumlahkannya. Namun, ini berpotensi sangat mahal jika footprint-nya besar—mungkin memerlukan membaca ribuan texel hanya untuk satu pencarian. Pendekatan yang lebih baik adalah menghitung dan menyimpan rata-rata tekstur di berbagai area dengan ukuran dan posisi yang berbeda.

Nama "mip" adalah singkatan dari frasa Latin multim in parvo yang berarti "banyak dalam ruang kecil." Versi yang sangat populer dari ide ini dikenal sebagai “MIP mapping” atau hanya mipmapping. Mipmap adalah urutan tekstur yang semuanya berisi gambar yang sama tetapi dengan resolusi yang lebih rendah dan lebih rendah. Gambar tekstur resolusi penuh asli disebut tingkat dasar, atau level0, dari mipmap, dan level1 dihasilkan dengan mengambil gambar itu dan menurunkannya dengan faktor 2 di setiap dimensi, menghasilkan gambar dengan seperempat lebih banyak teksel. Texel dalam gambar ini, secara kasar, rata-rata dari area persegi berukuran 2 kali 2 texel pada gambar level-0.

Proses ini dapat dilanjutkan untuk menentukan level mipmap sebanyak yang diinginkan: gambar pada level k dihitung dengan menurunkan sampel gambar pada level k 1 sebanyak dua. Sebuah texel pada tingkat k sesuai dengan area persegi berukuran 2^k kali 2^k texel dalam tekstur aslinya. Misalnya, dimulai dengan gambar tekstur 1024× 1024, kita dapat

menghasilkan mipmap dengan 11 level: level 0 adalah 1024 × 1024; level 1 adalah 512 × 512, dan seterusnya hingga level 10, yang hanya memiliki satu texel. Struktur semacam ini, dengan gambar yang mewakili konten yang sama pada serangkaian tingkat pengambilan sampel yang lebih rendah dan lebih rendah, disebut piramida gambar, berdasarkan metafora visual untuk menumpuk semua gambar yang lebih kecil di atas aslinya.

Penyaringan Tekstur Dasar dengan Mipmaps

Dengan mipmap, atau piramida gambar, di tangan, pemfilteran tekstur dapat dilakukan jauh lebih efisien daripada dengan mengakses banyak texel satu per satu. Saat kita membutuhkan rata-rata nilai tekstur pada area yang luas, kita cukup menggunakan nilai dari tingkat mipmap yang lebih tinggi, yang sudah menjadi rata-rata pada area gambar yang luas. Cara termudah dan tercepat untuk melakukannya adalah dengan mencari satu nilai dari mipmap, memilih level sehingga ukuran yang dicakup oleh teks pada level tersebut kira-kira sama dengan ukuran keseluruhan piksel jejak kaki. Tentu saja, jejak piksel mungkin sangat berbeda bentuknya dari area (selalu persegi) yang diwakili oleh teks, dan kita dapat mengharapkannya untuk menghasilkan beberapa artefak.

Mengesampingkan sejenak pertanyaan tentang apa yang harus dilakukan ketika tapak piksel memiliki bentuk memanjang, misalkan tapak adalah persegi dengan lebar D, diukur dalam bentuk teks dalam tekstur resolusi penuh. Berapa tingkat mipmap yang sesuai untuk sampel? Karena teks pada level k menutupi kotak dengan lebar 2k, tampaknya tepat untuk memilih k sehingga

$$2k \approx D$$

jadi kita misalkan $k = \log_2 D$. Tentu saja ini akan memberikan nilai non-integer k sebagian besar waktu, dan kami hanya menyimpan gambar mipmap untuk level integer. Dua solusi yang mungkin adalah mencari nilai hanya untuk bilangan bulat terdekat ke k (efisien tetapi menghasilkan jahitan pada transisi mendadak antar level) atau mencari nilai untuk dua bilangan bulat terdekat ke k dan menginterpolasi nilai secara linier (dua kali kerja, tapi lebih halus).

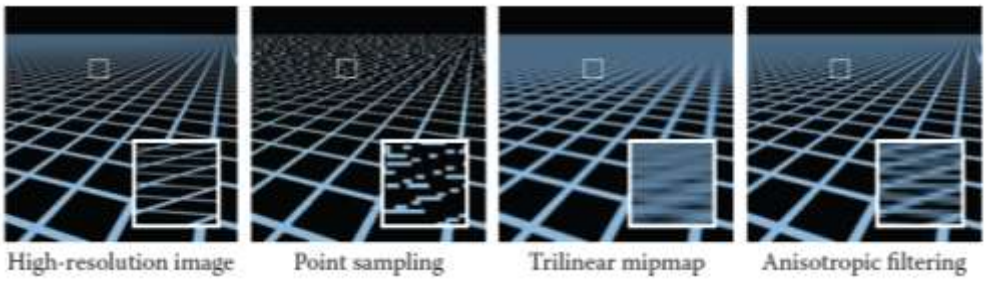
Sebelum kita benar-benar dapat menuliskan algoritme untuk pengambilan sampel peta mip, kita harus memutuskan bagaimana kita akan memilih "lebar" D ketika jejak kaki tidak persegi. Beberapa kemungkinan mungkin antara lain akar kuadrat dari luas atau untuk menemukan sumbu terpanjang dari tapak dan menyebutnya lebar. Kompromi praktis yang mudah dihitung adalah dengan menggunakan panjang sisi terpanjang:

$$D = \max\{||u_x||, ||u_y||\}$$

```
Color mipmap_sample_trilinear(Texture mip[], float u, float v,
matrix J) {
D = max_column_norm(J)
k = log2(D)
k0 = floor(k); k1 = k0 + 1
a = k1 - k; b = 1 - a
c0 = tex_sample_bilinear(mip[k0], u, v)
c1 = tex_sample_bilinear(mip[k1], u, v)
return a * c0 + b * c1
}
```

Mapping mip dasar melakukan pekerjaan yang baik untuk menghilangkan alias, tetapi karena tidak dapat menangani jejak piksel yang memanjang, atau anisotropik, itu tidak bekerja dengan baik saat permukaan dilihat di sudut penggembalaan. Ini paling sering terlihat pada bidang besar yang mewakili permukaan tempat penonton berdiri. Titik-titik di lantai yang jauh dilihat pada sudut yang sangat curam, menghasilkan jejak kaki yang sangat anisotropik yang

mendekati mapping mip dengan area persegi yang jauh lebih besar. Gambar yang dihasilkan akan tampak kabur dalam arah horizontal.



Gambar 11.21 Hasil antialiasing scene uji yang menantang (gambar referensi menunjukkan struktur detail, di sebelah kiri) menggunakan tiga strategi berbeda: cukup mengambil sampel titik tunggal dengan interpolasi tetangga terdekat; menggunakan piramida mipmap untuk rata-rata area persegi dalam tekstur untuk setiap piksel; menggunakan beberapa sampel dari mipmap untuk rata-rata wilayah anisotropik dalam tekstur.

Penyaringan Anisotropik

Sebuah mipmap dapat digunakan dengan beberapa pencarian untuk memperkirakan tapak memanjang dengan lebih baik. Idenya adalah untuk memilih level mipmap berdasarkan sumbu terpendek dari footprint daripada yang terbesar, kemudian rata-rata bersama beberapa lookupspasi sepanjang sumbu panjang. (Lihat Gambar 11.21.)

11.4 APLIKASI MAPPING TEKSTUR

Setelah Anda memahami ide untuk mendefinisikan koordinat tekstur untuk permukaan dan mesin untuk mencari nilai tekstur, mesin ini memiliki banyak kegunaan. Di bagian ini kami meninjau beberapa teknik paling penting dalam mapping tekstur, tetapi tekstur adalah alat yang sangat umum dengan aplikasi yang hanya dibatasi oleh apa yang dapat dipikirkan oleh programmer.

Mengontrol Parameter Shading

Dasar paling dasar dari mapping tekstur adalah untuk memperkenalkan variasi warna dengan membuat warna difus yang digunakan dalam perhitungan bayangan—apakah dalam ray tracer atau dalam shader fragmen—bergantung pada pencarian nilai dari tekstur. Komponen difus bertekstur dapat digunakan untuk menempelkan stiker, dekorasi cat, atau teks cetak pada permukaan, dan juga dapat mensimulasikan variasi warna material, misalnya untuk kayu atau batu.

Tidak ada yang membatasi kita untuk memvariasikan hanya warna difus. Parameter lain, seperti pantulan spekular atau kekasaran spekular, juga dapat diteksturkan. Misalnya, kotak kardus dengan pita pembungkus transparan yang menempel di dalamnya mungkin memiliki warna difus yang sama di mana-mana tetapi lebih berkilau, dengan pantulan spekuler yang lebih tinggi dan kekasaran yang lebih rendah, di tempat pita itu berada daripada di tempat lain. Dalam banyak kasus, peta untuk parameter yang berbeda dikorelasikan: misalnya, cangkir keramik putih mengilap dengan logo yang tercetak di atasnya mungkin lebih kasar dan lebih gelap tempat cetakannya (Gambar 11.22), dan buku dengan judulnya yang dicetak tinta metalik mungkin berubah warna tidak merata, warna spekular, dan kekasaran, semuanya sekaligus.

Gambar 11.22 Mug keramik dengan kekasaran specular yang dikendalikan oleh salinan terbalik dari tekstur warna yang menyebar.

Peta Normal dan Peta Bump

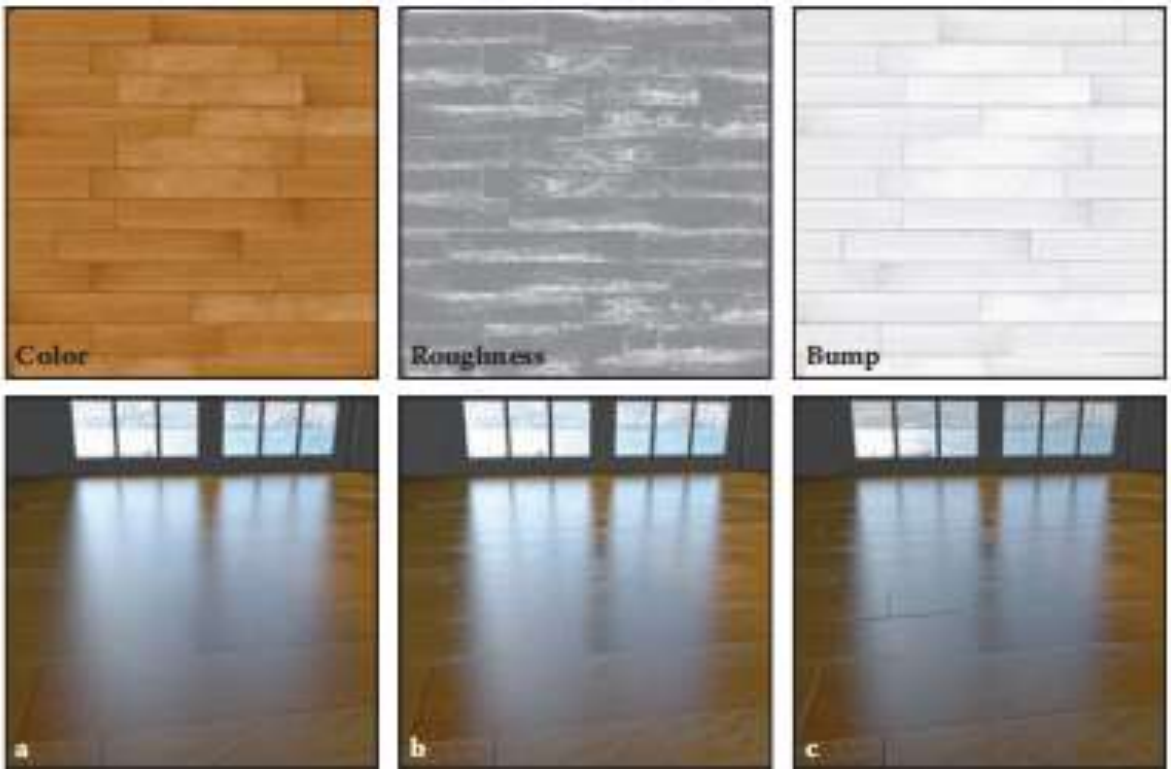
Kuantitas lain yang penting untuk shading adalah permukaan normal. Dengan normal terinterpolasi (Bagian 8.2), kita tahu bahwa normal bayangan tidak harus sama dengan normal geometrik dari permukaan di bawahnya. Mapping normal mengambil keuntungan dari fakta ini dengan membuat bayangan nilai ketergantungan normal dibaca dari peta tekstur. Cara paling sederhana untuk melakukannya adalah dengan menyimpan normal dalam tekstur, dengan tiga angka tersimpan di setiap teksel yang ditafsirkan, bukan sebagai tiga komponen warna, sebagai koordinat 3D dari vektor normal.

Sebelum peta normal dapat digunakan, kita perlu mengetahui sistem koordinat apa yang diwakili oleh normal yang dibaca dari peta. Menyimpan normal secara langsung di ruang objek, dalam sistem koordinat yang sama yang digunakan untuk mewakili geometri permukaan itu sendiri, adalah yang paling sederhana: pembacaan normal dari peta dapat digunakan dengan cara yang persis sama seperti normal yang dilaporkan oleh permukaan itu sendiri: dalam banyak kasus itu perlu diubah menjadi ruang dunia untuk perhitungan pencahayaan, seperti normal yang datang dengan geometri.

Namun, peta normal yang disimpan dalam ruang objek secara inheren terkait dengan geometri permukaan—bahkan agar peta normal tidak berpengaruh, untuk mereproduksi hasil dengan normal geometrik, isi peta normal harus melacak orientasi permukaan. Selanjutnya, jika permukaan akan mengalami deformasi, sehingga perubahan normal geometrik, peta normal objek-ruang tidak dapat digunakan lagi, karena akan tetap memberikan normal bayangan yang sama.

Solusinya adalah menentukan sistem koordinat untuk yang normal yang melekat pada permukaan. Sistem koordinat seperti itu dapat didefinisikan berdasarkan ruang singgung permukaan (lihat Bagian 2.5): pilih sepasang vektor tangen dan gunakan untuk mendefinisikan basis ortonormal (Bagian2.4.5). Fungsi koordinat tekstur itu sendiri menyediakan cara yang berguna untuk memilih vektor-vektor tangen: gunakan arah-arah yang bersinggungan dengan garis-garis konstanta u dan v. Garis singgung ini umumnya tidak ortogonal, tetapi kita dapat menggunakan prosedur dari Bagian 2.4.7 untuk “mengkuadratkan” basis ortonormal, atau dapat didefinisikan menggunakan normal permukaan dan hanya satu vektor tangen.

Ketika normal diekspresikan dalam dasar ini, variasinya jauh lebih sedikit; karena sebagian besar menunjuk dekat arah normal ke permukaan halus, mereka akan berada di dekat vektor $(0, 0, 1)^T$ di peta normal.



Gambar 11.23 Lantai kayu dirender menggunakan peta tekstur untuk mengontrol bayangan. (a) Hanya warna semu yang dimodulasi oleh peta tekstur. (b) Kekasaran specular juga dimodulasi oleh peta tekstur kedua. (c) Permukaan normal dimodifikasi oleh bump map.

Dari mana peta normal berasal? Seringkali mereka dihitung dari model yang lebih rinci di mana permukaan halus merupakan perkiraan; kadang-kadang mereka dapat diukur langsung dari permukaan nyata. Mereka juga dapat ditulis sebagai bagian dari proses pemodelan; dalam hal ini sering kali baik menggunakan peta benjolan untuk menentukan normal secara tidak langsung. Idenya adalah bahwa bump map adalah bidang ketinggian: fungsi yang memberikan ketinggian lokal dari permukaan detail di atas permukaan halus. Di mana nilainya tinggi (di mana peta terlihat cerah, jika Anda menampilkannya sebagai gambar) permukaan menonjol di luar permukaan halus; di mana nilainya di bawah (di mana peta terlihat gelap) di bawah permukaannya. Misalnya, garis gelap sempit di peta benjolan adalah goresan, atau titik putih kecil adalah benjolan.

Menurunkan peta normal dari peta benjolan sederhana: peta normal (dinyatakan dalam bingkai tangen) adalah turunan dari peta benjolan. Gambar 11.23 menunjukkan peta tekstur yang digunakan untuk membuat warna butiran kayu dan untuk mensimulasikan peningkatan kekasaran permukaan karena lapisan akhir yang meresap ke bagian kayu yang lebih berpori, bersama dengan peta benjolan untuk membuat hasil akhir yang tidak sempurna dan celah antar papan, untuk membuat lantai kayu yang realistis.

Peta Perpindahan

Masalah dengan peta normal adalah mereka tidak benar-benar mengubah permukaan sama sekali; mereka hanya trik bayangan. Ini menjadi jelas ketika geometri yang ditunjukkan oleh peta normal seharusnya menyebabkan efek yang nyata dalam 3D. Dalam gambar diam, masalah pertama yang diamati biasanya adalah siluet objek tetap mulus meskipun ada tonjolan di bagian dalam. Inanimasi, thelackofparallax menunjukkan bahwa tonjolan-tonjolan itu, betapapun meyakinkannya, sebenarnya hanya "dilukis" di permukaan.

Tekstur dapat digunakan untuk lebih dari sekadar bayangan: tekstur dapat digunakan untuk mengubah geometri. Peta perpindahan adalah salah satu versi paling sederhana dari ide ini. Konsepnya sama dengan bump map: peta skalar (satu saluran) yang memberikan ketinggian di atas "rata-rata medan". Tapi efeknya berbeda. Alih-alih menurunkan bayangan normal dari peta ketinggian saat menggunakan geometri mulus, peta perpindahan sebenarnya mengubah permukaan, memindahkan setiap titik sepanjang normal permukaan halus ke lokasi baru. Normalnya kira-kira sama dalam setiap kasus, tetapi permukaannya berbeda.

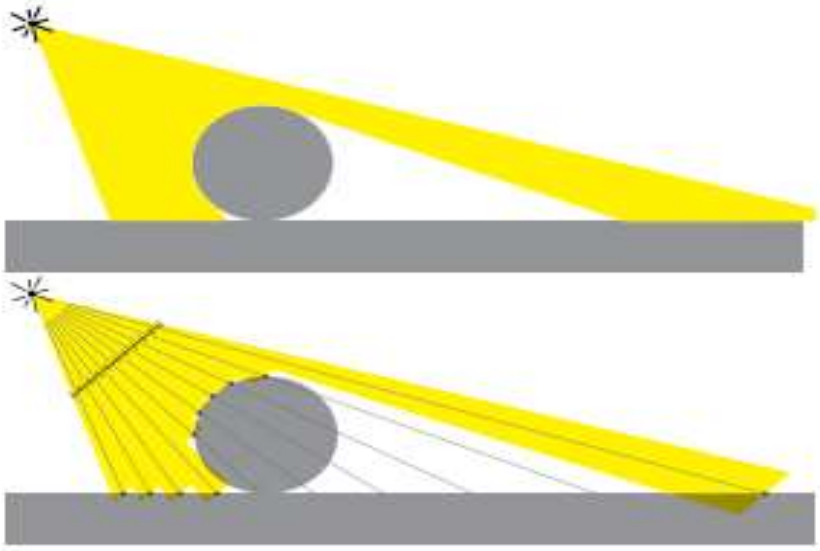
Cara paling umum untuk menerapkan peta perpindahan adalah dengan menguji permukaan halus dengan sejumlah besar segitiga kecil, dan kemudian memindahkan simpul dari mesh yang dihasilkan menggunakan peta perpindahan. Dalam pipeline grafis, ini dapat dilakukan dengan menggunakan pencarian tekstur pada tahap vertex, dan sangat berguna untuk medan.

Peta Bayangan

Bayangan adalah isyarat penting untuk hubungan objek dalam sebuah scene, dan seperti yang telah kita lihat, bayangan itu mudah disertakan dalam gambar ray-trace. Namun, tidak jelas bagaimana cara mendapatkan bayangan dalam rendering raster, karena permukaan dianggap satu per satu, secara terpisah. Peta bayangan adalah teknik untuk menggunakan mesin mapping tekstur untuk mendapatkan bayangan dari sumber cahaya titik.

Ide peta bayangan adalah untuk mewakili volume ruang yang diterangi oleh sumber cahaya titik. Pikirkan sumber seperti lampu sorot atau proyektor video, yang memancarkan cahaya dari suatu titik ke berbagai arah yang terbatas. Volume yang diterangi—kumpulan titik di mana Anda akan melihat cahaya di tangan Anda jika Anda memegangnya di sana—adalah gabungan segmen garis yang menghubungkan sumber cahaya ke titik permukaan terdekat di sepanjang setiap sinar yang meninggalkan titik itu.

Menariknya, volume ini sama dengan volume yang terlihat oleh kamera perspektif yang terletak di titik yang sama dengan sumber cahaya: sebuah titik disinari oleh sumber jika dan hanya jika terlihat dari lokasi sumber cahaya. Dalam kedua kasus, ada kebutuhan untuk mengevaluasi visibilitas untuk titik di scene: visibilitas, kami perlu tahu apakah sebuah fragmen terlihat oleh kamera, untuk mengetahui apakah akan menggambarnya di gambar; dan untuk bayangan, kita perlu tahu apakah sebuah fragmen terlihat oleh sumber cahaya, untuk mengetahui apakah itu diterangi oleh sumber itu atau tidak. (Lihat Gambar 11.24.)



Gambar 11.24 Atas: wilayah ruang yang diterangi oleh titik cahaya. Bawah: wilayah itu seperti yang diperkirakan oleh peta bayangan selebar 10 piksel.

Dalam kedua kasus, solusinya sama: peta kedalaman yang memberi tahu jarak ke permukaan terdekat di sepanjang sekelompok sinar. Dalam kasus visibilitas, ini adalah zbuffer (Bagian 8.2.3), dan untuk kasus bayangan, ini disebut peta bayangan. Dalam kedua kasus, visibilitas dievaluasi dengan membandingkan kedalaman fragmen baru dengan kedalaman yang tersimpan di peta, dan permukaan disembunyikan dari titik proyeksi (oklusi atau dibayangi) jika kedalamannya lebih besar dari kedalaman permukaan terdekat yang terlihat. Perbedaannya adalah bahwa buffer z digunakan untuk melacak permukaan terdekat yang terlihat sejauh ini dan diperbarui selama rendering, sedangkan peta bayangan memberi tahu jarak ke permukaan terdekat di seluruh scene.

Peta bayangandihitung dalamseparaterenderingpass sebelumnya: cukup rasterisasi seluruh scene seperti biasa, dan pertahankan peta kedalaman yang dihasilkan (tidak perlu repot menghitung nilai piksel). Kemudian, dengan peta bayangan di tangan, Anda melakukan rendering pass biasa, dan ketika Anda perlu mengetahui apakah sebuah fragmen terlihat oleh sumber, Anda memproyeksikan lokasinya di peta bayangan (menggunakan proyeksi perspektif yang sama yang digunakan untuk membuat peta bayangan di tempat pertama) dan membandingkan nilai yang dicari dmap dengan jarak sebenarnya d ke sumber. Jika jaraknya sama, titik fragmen akan menyala; jika $d > d_{map}$, itu menyiratkan ada permukaan yang berbeda lebih dekat ke sumbernya, sehingga dibayangi.

Ungkapan "jika jaraknya sama" seharusnya menimbulkan tanda bahaya di benak Anda: karena semua besaran yang terlibat adalah aproksimasi dengan presisi terbatas, kita tidak bisa mengharapkannya persis sama. Untuk titik yang terlihat, $d \approx d_{map}$ tetapi terkadang d akan sedikit lebih besar dan terkadang sedikit lebih kecil. Untuk alasan ini, toleransi diperlukan: suatu titik dianggap menyala jika $d - d_{map} < \in$. Toleransi ini dikenal sebagai bias bayangan.

Saat mencari di peta bayangan, tidak masuk akal untuk menginterpolasi antara nilai kedalaman yang direkam di peta. Ini mungkin menghasilkan kedalaman yang lebih akurat (membutuhkan lebih sedikit bias bayangan) di area yang halus, tetapi akan menyebabkan masalah yang lebih besar di dekat batas bayangan, di mana nilai kedalaman berubah secara tiba-tiba. Oleh karena itu, pencarian tekstur pada peta bayangan dilakukan dengan menggunakan rekonstruksi tetangga terdekat. Untuk mengurangi aliasing, beberapa sampel dapat digunakan, dengan hasil bayangan 1-atau-0 (bukan kedalaman) dirata-ratakan; ini dikenal sebagai penyaringan persentase lebih dekat.

Peta Lingkungan

Sama seperti tekstur yang berguna untuk memasukkan detail ke dalam bayangan pada permukaan tanpa harus menambahkan lebih banyak detail ke model, tekstur juga dapat digunakan untuk memasukkan detail ke dalam iluminasi tanpa harus memodelkan geometri sumber cahaya yang rumit. Ketika cahaya datang dari jauh dibandingkan dengan ukuran objek yang terlihat, iluminasi berubah sangat sedikit dari titik ke titik dalam scene. Hal ini berguna untuk membuat asumsi bahwa iluminasi bergantung hanya pada arah yang Anda lihat, dan sama untuk semua titik dalam scene, dan kemudian mengekspresikan ketergantungan iluminasi pada arah ini menggunakan peta lingkungan.

Ide dari peta lingkungan adalah bahwa suatu fungsi yang didefinisikan pada arah dalam 3Disfungsi pada bola satuan, sehingga dapat direpresentasikan menggunakan peta tekstur dalam cara yang persis sama seperti kita mungkin merepresentasikan variasi warna pada objek bola. Alih-alih menghitung koordinat tekstur dari koordinat 3D titik permukaan, kita menggunakan rumus yang persis sama untuk menghitung koordinat tekstur dari koordinat 3D dari vektor satuan yang mewakili arah dari mana kita ingin mengetahui iluminasi.

Aplikasi paling sederhana dari peta lingkungan adalah memberi warna pada sinar dalam pelacak sinar yang tidak mengenai objek apa pun:

```
trace_ray(ray, scene) {
        if (surface = scene.intersect(ray)) {
        return surface.shade(ray)
    } else {
        u, v = spheremap_coords(r.direction)
        return texture_lookup(scene.env_map, u, v)
    }
}
```

Dengan perubahan pada ray tracer ini, objek mengkilap yang mencerminkan objek scene lainnya sekarang juga akan mencerminkan lingkungan latar belakang.

Efek asimilar dapat dicapai dalam konteks asterisasi dengan menambahkan pantulan cermin ke perhitungan bayangan, yang dihitung dengan cara yang sama seperti dalam ray tracer, tetapi hanya mencari langsung di peta lingkungan tanpa memperhatikan objek lain di tempat kejadian:

```
shade_fragment(view_dir, normal) {
    out_color = diffuse_shading(k_d, normal)
    out_color += specular_shading(k_s, view_dir, normal)
    u, v = spheremap_coords(reflect(view_dir, normal))
    out_color += k_m * texture_lookup(environment_map, u, v)
}
```

Teknik ini dikenal sebagai mapping refleksi

Penggunaan peta lingkungan yang lebih maju menghitung semua penerangan dari peta lingkungan, bukan hanya pantulan cermin. Ini adalah pencahayaan lingkungan, dan dapat dihitung dalam pelacak sinar menggunakan integrasi Monte Carlo atau dalam rasterisasi dengan memperkirakan lingkungan dengan kumpulan sumber titik dan menghitung banyak peta bayangan.

Peta lingkungan dapat disimpan dalam koordinat apa pun yang dapat digunakan untuk memetakan bola. Koordinat sferis (bujur–lintang) adalah salah satu opsi yang populer, meskipun kompresi tekstur di kutub menyia-nyiakan resolusi tekstur dan dapat membuat artefak di kutub. Cubemapsmerupakan pilihan yang lebih efisien, banyak digunakan dalam aplikasi interaktif (Gambar 11.25).

11.5 TEKSTUR 3D PROSEDURAL

Dalam bab-bab sebelumnya, kami menggunakan cr sebagai pantulan difus pada suatu titik pada suatu objek. Untuk objek yang tidak memiliki warna solid, kita dapat menggantinya dengan fungsi $c_r(p)$ yang memetakan titik 3D ke warna RGB (Peachey, 1985; Perlin, 1985). Fungsi ini mungkin hanya mengembalikan pantulan objek yang berisi p. Tetapi untuk objek dengan tekstur, kita harus mengharapkan $c_r(p)$ bervariasi saat p bergerak melintasi permukaan.

Sebuah alternatif untuk mendefinisikan fungsi mapping tekstur yang memetakan dari permukaan 3D ke domain tekstur 2D adalah dengan membuat tekstur 3D yang mendefinisikan nilai RGB pada setiap titik dalam ruang 3D. Kami hanya akan menyebutnya untuk titik p di permukaan, tetapi biasanya lebih mudah untuk mendefinisikannya untuk semua titik 3D daripada subset titik 2D yang berpotensi aneh yang berada di permukaan yang berubah-ubah. Hal yang baik tentang mapping tekstur 3D adalah mudah untuk mendefinisikan fungsi mapping, karena permukaan sudah tertanam dalam ruang 3D, dan tidak ada distorsi dalam

mapping dari 3D ke ruang tekstur. Strategi seperti itu jelas cocok untuk permukaan yang "diukir" dari media padat, seperti patung marmer.

Gambar 11.25 Sebuah peta kubus Basilika Santo Petrus, dengan enam wajah yang tersimpan dalam gambar dalam susunan "salib horizontal" yang belum dibuka. (tekstur: Emil Persson)

Kelemahan dari tekstur 3D adalah menyimpannya sebagai gambar atau volume raster 3D menghabiskan banyak memori. Untuk alasan ini, koordinat tekstur 3D paling sering digunakan dengan tekstur prosedural di mana nilai tekstur dihitung menggunakan prosedur matematika daripada dengan melihatnya dari gambar tekstur. Di bagian ini, kita melihat beberapa alat dasar yang digunakan untuk mendefinisikan tekstur prosedural. Ini juga dapat digunakan untuk mendefinisikan tekstur prosedural 2D, meskipun dalam 2D lebih umum menggunakan gambar tekstur raster.

Tekstur Garis 3D

Ada beberapa cara yang mengejutkan untuk membuat tekstur bergaris. Mari kita asumsikan kita memiliki dua warna c_0 dan c_1 yang ingin kita gunakan untuk membuat warna garis. Kita membutuhkan beberapa fungsi osilasi untuk beralih di antara dua warna. Aneasyoneisasi:

```
RGB stripe( point p )
if (sin(xp) > 0) then
return c0
else
return c1
```

Kami juga dapat membuat lebar strip dapat dikontrol:

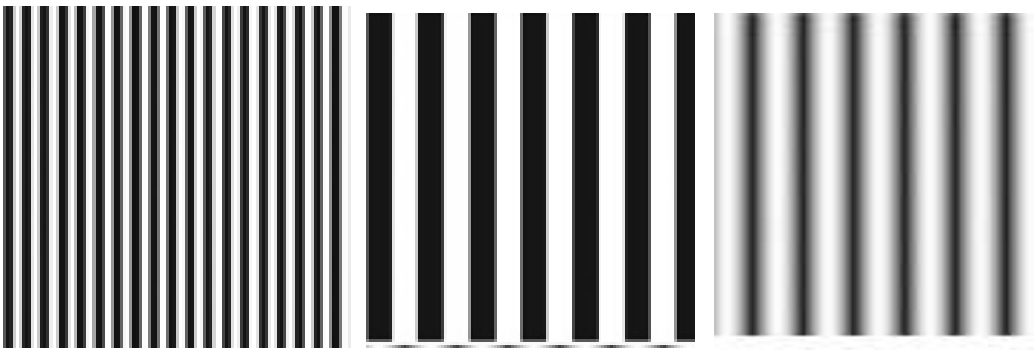
```
RGB stripe( point p, real w)
if (sin(πxp/w) > 0) then
return c0
else
return c1
```

Jika kita ingin menginterpolasi secara halus antara warna garis, kita dapat menggunakan parameter t untuk memvariasikan warna secara linier:

```
RGB stripe( point p, real w )
t = (1 + sin(πpx/w))/2
```

return (1 − *t*)*c*0 + *tc*1

Ketiga kemungkinan ini ditunjukkan pada Gambar 11.26



Gambar 11.26 Berbagai tekstur garis dihasilkan dari menggambar larik teratur titik xy sambil menjaga z tetap konstan.

Solid noise

Meskipun tekstur biasa seperti garis-garis sering berguna, kami ingin dapat membuat tekstur "berbintik-bintik" seperti yang kita lihat pada telur burung. Ini biasanya dilakukan dengan menggunakan semacam "suara keras", biasanya disebut kebisingan Perlin setelah penemunya, yang menerima Academy Award teknis untuk dampaknya dalam industri film (Perlin, 1985).

Mendapatkan tampilan yang tidak jelas dengan memanggil nomor acak sebelum setiap titik tidak akan sesuai, karena itu hanya akan seperti "white noise" di TV statis. Kami ingin membuatnya mulus tanpa kehilangan kualitas acak. Salah satu kemungkinannya adalah mengaburkan white noise, tetapi tidak ada implementasi praktis untuk ini. Kemungkinan lain adalah membuat kisi besar dengan nomor acak di setiap titik kisi, dan kemudian menginterpolasi titik acak ini untuk titik baru di antara simpul kisi; ini hanya 3Dteksturarrayyang dijelaskan di bagian terakhir dengan angka acak dalam array. Teknik ini membuat kisi terlalu jelas. Perlin menggunakan berbagai trik untuk meningkatkan teknik dasar kisi ini sehingga kisi tidak begitu jelas. Ini menghasilkan serangkaian langkah yang tampak seperti barok, tetapi pada dasarnya hanya ada tiga perubahan dari interpolasi linier array 3D nilai acak. Perubahan pertama adalah menggunakan interpolasi Hermite untuk menghindari mach band, seperti yang dapat dilakukan dengan tekstur biasa. Perubahan kedua adalah penggunaan vektor acak daripada nilai, dengan produk titik untuk menurunkan angka acak; ini membuat struktur grid yang mendasarinya menjadi kurang jelas secara visual dengan memindahkan minimum lokal dan maxima dari simpul grid. Perubahan ketiga adalah menggunakan array 1D dan hashing untuk membuat virtual3Darray dari vektor acak. Ini menambahkan komputasi untuk menurunkan penggunaan memori. Inilah metode dasarnya:

$$n(x,y,z) = \sum_{i=[x]}^{[x]+1} \sum_{j=[y]}^{[y]+1} \sum_{k=[z]}^{[z]+1} \Omega_{ijk}(x-i, y-j, z-k)$$

di mana (x, y, z) adalah koordinat Cartesian dari x, dan

$$\Omega ijk(u, v, w) = \omega(u)\omega(v)\omega(w)\,(\Gamma ijk \cdot (u, v, w)),$$

dan ω(t) adalah fungsi pembobotan kubik:

$$\omega(\mathrm{t}) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & jika\ |t| < 1 \\ 0 & jika\ tidak \end{cases}$$

Bagian akhirnya adalah bahwa Γ_{ijk} adalah vektor satuan acak untuk titik kisi (x, y, z) = (i, j, k). Karena kami menginginkan *ijk* potensial, kami menggunakan pseudorandomtable:

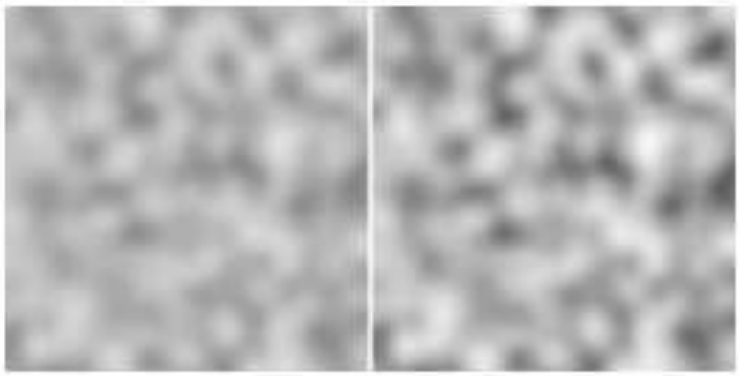
$$\Gamma ijk = \mathbf{G}\,(\phi(i + \phi(j + \varphi(k))))),$$

di mana G adalah larik terkomputasi dari n vektor satuan acak, dan φ(i) = P[$_i$ mod *n*] di mana P adalah larik dengan panjang n yang berisi permutasi bilangan bulat 0 hingga n – 1. Dalam praktiknya, Perlin melaporkan n = 256 bekerja dengan baik. Untuk memilih vektor unit acak (v_x, v_y, v_z) set pertama

$$vx = 2\xi - 1,$$
$$vy = 2\xi - 1,$$
$$vz = 2\xi - 1,$$

di mana ξ,ξ' ,ξ'' adalah bilangan acak kanonik (seragam dalam interval [0, 1)). Kemudian, jika ($v^2_x + v^2_y + v^2_z$) < 1, buatlahvektoraunitvektor. Jika tidak, tetap atur secara acak hingga panjangnya kurang dari satu, lalu buatlah menjadi vektor satuan. Ini adalah contoh metode penolakan, yang akan dibahas lebih lanjut di Bab 14. Pada dasarnya, uji "kurang dari" mendapatkan titik acak dalam bola satuan, dan vektor untuk titik asal ke titik tersebut adalah acak seragam. Itu tidak berlaku untuk titik acak dalam kubus, jadi kami "menyingkirkan" sudut dengan tes.

Gambar 11.27 Nilai absolut dari solid noise, dan kebisingan untuk nilai x dan y yang diskalakan.



Gambar 11.28 Menggunakan 0,5(noise+1) (atas) dan 0,8(noise+1) (bawah) untuk intensitas.

Karena solidnoise bisa positif atau negatif, maka harus diubah sebelum diubah menjadi warna. Nilai absolut dari noise pada persegi 10x10 ditunjukkan pada Gambar 11.27, bersama dengan versi yang diregangkan. Versi ini diregangkan dengan menskalakan titik-titik yang dimasukkan ke fungsi noise.

Kurva gelap adalah tempat fungsi noise asli berubah dari positif menjadi negatif. Karena noise bervariasi dari 1 hingga 1, gambar yang lebih halus dapat diperoleh dengan menggunakan(noise+1)/2untuk warna. Namun, karena nilai noise yang mendekati 1 atau 1 jarang terjadi, ini akan menjadi gambar yang cukup halus. Scalling yang lebih besar dapat meningkatkan kontras (Gambar 11.28).

Pergolakan

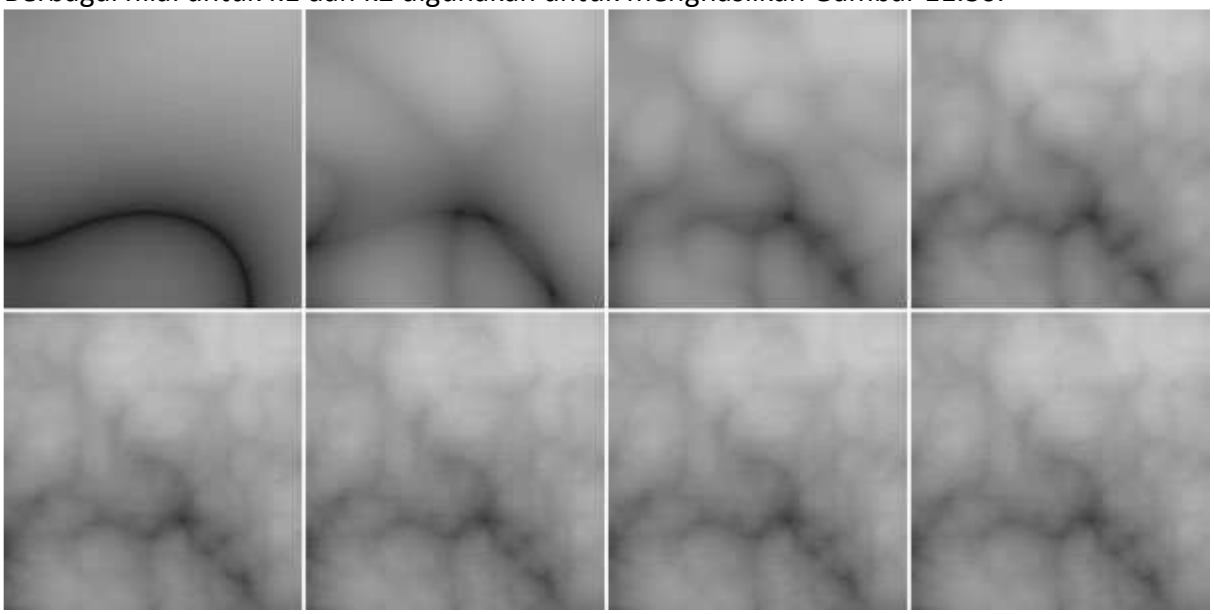
Banyak tekstur alami mengandung berbagai ukuran fitur dalam tekstur yang sama. Perlin menggunakan fungsi "turbulensi" pseudofractal:

$$n_t = \sum_i \frac{|n(2^i x)|}{2^i}$$

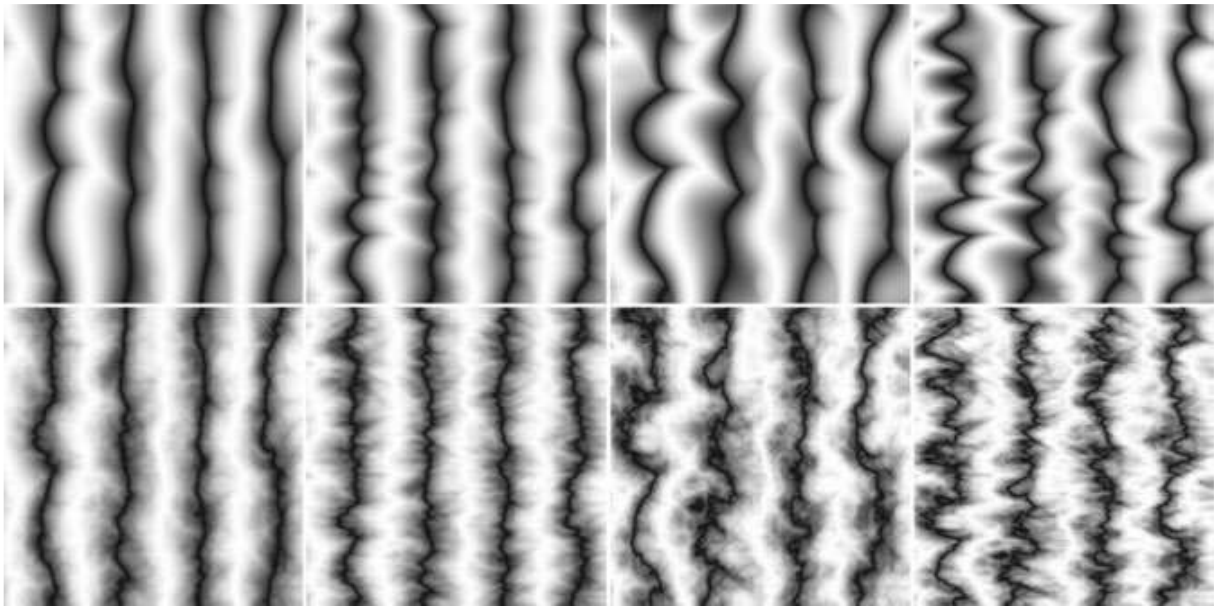
Ini secara efektif berulang kali menambahkan salinan skala dari fungsi kebisingan pada diri sendiri seperti yang ditunjukkan pada Gambar 11.29. Turbulensi dapat digunakan untuk mendistorsi fungsi garis:

RGB turbstripe(point **p**, double w)
double *t* = (1 + sin(*k*1*zp* + turbulence(*k*2**p**))*/w*)*/*2
return *t* * *s*0+ (1 − *t*) * *s*1

Berbagai nilai untuk k1 dan k2 digunakan untuk menghasilkan Gambar 11.30.



Gambar 11.29 Fungsi turbulensi dengan (dari kiri atas ke kanan bawah) satu hingga delapan suku dalam penjumlahan.



Gambar 11.30 Berbagai tekstur garis turbulen dengan k1, k2 yang berbeda. Baris atas hanya memiliki suku pertama dari deret turbulensi.

Pertanyaan yang Sering Diajukan

- Bagaimana cara menerapkan mapping perpindahan dalam ray tracing?

Tidak ada cara yang ideal untuk melakukannya. Membangkitkan semua segitiga dan menyimpan geometri bila diperlukan akan mencegah kelebihan memori (Pharr & Hanrahan, 1996; Pharr, Kolb, Gershbein, & Hanrahan, 1997). Mencoba untuk memotong permukaan yang dipindahkan secara langsung dimungkinkan ketika fungsi perpindahan dibatasi (Patterson, Hoggar, & Logie, 1991; Heidrich & Seidel, 1998; Smits, Shirley, & Stark, 2000).

- Mengapa gambar saya dengan tekstur tidak terlihat realistis?

Manusia pandai melihat ketidaksempurnaan kecil di permukaan. Ketidaksempurnaan geometris biasanya tidak ada dalam gambar yang dihasilkan komputer yang menggunakan peta tekstur untuk detailnya, sehingga terlihat "terlalu halus".

Catatan

Pembahasan tekstur perspektif-benar didasarkan pada Bayangan Cepat dan Efek Pencahayaan Menggunakan Mapping Tekstur (Segal, Korobkin, van Widenfelt, Foran, & Haeberli, 1992) dan pada Desain Mesin Game 3D (Eberly, 2000).

Latihan

1. Temukan beberapa cara untuk menerapkan papan catur 2D tak terbatas menggunakan teknik permukaan dan padat. Mana yang terbaik?
2. Pastikan bahwa Persamaan (11.3) adalah persamaan yang valid dengan menggunakan aljabar brute-force.
3. Bagaimana Anda bisa menerapkan tekstur padat dengan menggunakan kedalaman buffer-z dan transformasi matriks?
4. Memperluas fungsi mipmap sample trilinear menjadi satu fungsi.

BAB 12
STRUKTUR DATA GRAFIS

Struktur data tertentu tampaknya muncul berulang kali dalam aplikasi grafis, mungkin karena mereka membahas ide-ide mendasar seperti permukaan, ruang, dan struktur scene.

Untuk mesh, kami membahas skema penyimpanan dasar yang digunakan untuk menyimpan mesh statis dan untuk mentransfer API meshestografis. Kami juga membahas struktur data edge bersayap (Baumgart, 1974) dan struktur setengah sisi terkait, yang berguna untuk mengelola model di mana tessellation berubah, seperti dalam subdivisi atau penyederhanaan model. Meskipun metode ini digeneralisasi ke poligon mesh sewenang-wenang, kami fokus pada kasus mesh segitiga yang lebih sederhana di sini.

Selanjutnya disajikan struktur data scene-graph. Berbagai bentuk struktur data ini ada di mana-mana dalam aplikasi grafis karena sangat berguna dalam mengelola objek dan transformasi. Semua API grafis baru dirancang untuk mendukung grafis scene dengan baik.

Untuk struktur data spasial, kami membahas tiga pendekatan untuk mengorganisasikan model dalam ruang 3D—hierarki volume pembatas, subdivisi ruang hierarkis, dan subdivisi ruang seragam—dan penggunaan subdivisi ruang hierarkis (pohon BSP) untuk menghilangkan permukaan tersembunyi. Metode yang sama juga digunakan untuk tujuan lain, termasuk pemeriksaan geometri dan deteksi tumbukan.

Akhirnya, array multidimensi tiling disajikan. Awalnya dikembangkan untuk membantu kinerja paging dalam aplikasi di mana data grafis perlu ditukar dari disk, struktur seperti itu sekarang penting untuk lokalitas memori pada mesin terlepas dari apakah array cocok di memori utama.

12.1 TRIANGLE MESHES

Kebanyakan model dunia nyata terdiri dari kompleks segitiga dengan simpul bersama. Ini biasanya dikenal sebagai triangular meshes, triangle meshes, ortriangular irregular networks (TINs), dan penanganannya secara efisien sangat penting untuk kinerja banyak program grafis. Jenis efisiensi yang penting tergantung pada aplikasinya. Mesh disimpan di disk dan di memori, dan kami ingin meminimalkan jumlah penyimpanan yang digunakan. Ketika mesh ditransmisikan melalui jaringan atau dari CPU ke sistem grafis, mesh tersebut menghabiskan bandwidth, yang seringkali bahkan lebih berharga daripada penyimpanan. Dalam aplikasi yang melakukan operasi pada mesh, selain hanya menyimpan dan menggambarnya—seperti subdivisi, pengeditan mesh, kompresi mesh, atau operasi lainnya—akses yang efisien ke informasi kedekatan sangat penting.

Mata triangle meshes umumnya digunakan untuk merepresentasikan permukaan, sehingga mata jaring bukan hanya kumpulan segitiga yang tidak berhubungan, tetapi merupakan jaringan kerja segitiga yang terhubung satu sama lain melalui simpul dan tepi bersama untuk masing-masing permukaan kontinu. Ini adalah wawasan utama tentang mesh: mesh dapat ditangani dengan lebih efisien daripada kumpulan segitiga yang tidak berhubungan dengan jumlah yang sama.

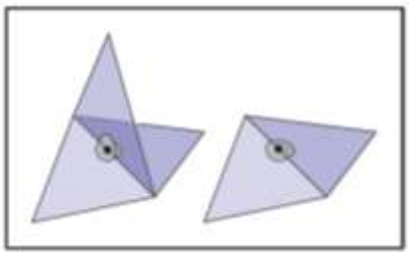
Informasi minimum yang diperlukan untuk mesh segitiga adalah himpunan segitiga (tiga kali lipat dari simpul) dan posisi (dalam ruang 3D) dari simpulnya. Tetapi banyak, jika tidak sebagian besar, program memerlukan kemampuan untuk menyimpan data tambahan di simpul, tepi, atau wajah untuk mendukung mapping tekstur, bayangan, animasi, dan operasi lainnya. Data simpul adalah yang paling umum: setiap simpul dapat memiliki parameter material, koordinat tekstur, radiasi—parameter apa pun yang nilainya berubah di seluruh

permukaan. Parameter-parameter ini kemudian diinterpolasi secara linier di setiap segitiga untuk menentukan fungsi kontinu di seluruh permukaan mesh. Namun, terkadang juga penting untuk dapat menyimpan data per edge atau per face.

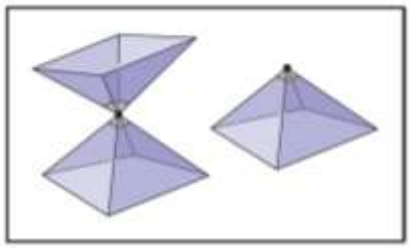
Topologi Mesh

Gagasan bahwa mesh adalah seperti permukaan dapat diformalkan sebagai batasan pada topologi mesh — cara segitiga terhubung bersama, tanpa memperhatikan posisi simpul. Banyak algoritma hanya akan bekerja, atau lebih mudah untuk diterapkan, pada mesh dengan konektivitas yang dapat diprediksi. Persyaratan paling sederhana dan paling ketat pada topologi mesh adalah permukaan menjadi manifold. Manifoldmesh adalah "kedap air"—tidak memiliki celah dan memisahkan ruang di bagian dalam permukaan dari ruang di luar. Itu juga terlihat seperti permukaan di mana-mana di jala.

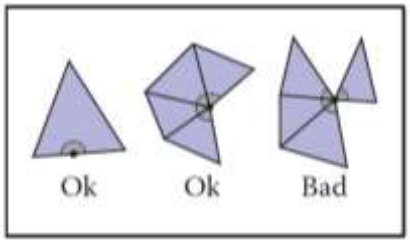
Istilah manifold berasal dari bidang matematika topologi: secara kasar, manifold (khususnya manifold dua dimensi, atau manifold 2) adalah permukaan di mana lingkungan kecil di sekitar titik mana pun dapat dihaluskan menjadi sedikit permukaan datar. Ide ini paling jelas dijelaskan dengan contoh tandingan: jika tepi pada jaring memiliki tiga segitiga yang terhubung padanya, lingkungan titik di tepi berbeda dari lingkungan salah satu titik di bagian dalam salah satu segitiga, karena ia memiliki "sirip" ekstra yang mencuat (Gambar 12.1). Jika tepi memiliki tepat dua segitiga yang melekat padanya, titik-titik di tepi memiliki lingkungan seperti titik di bagian dalam, hanya dengan lipatan di tengahnya. Demikian pula, jika segitiga yang berbagi simpul berada dalam konfigurasi seperti kiri pada Gambar 12.2, lingkungan itu seperti dua potong permukaan yang direkatkan di tengah, yang tidak dapat diratakan tanpa melipatgandakannya. Simpul dengan lingkungan yang lebih sederhana yang ditunjukkan di sebelah kanan baik-baik saja.



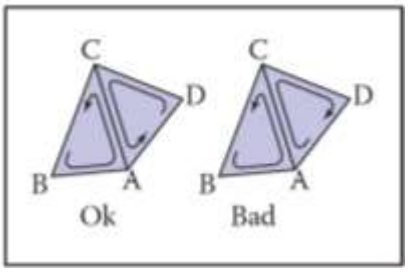
Gambar 12.1 Tepi interior non-manifold (kiri) dan manifold (kanan).



Gambar 12.2 Vertek interior non-manifold (kiri) dan manifold (kanan).



Gambar 12.3 Kondisi di tepi manifold dengan batas.



Gambar 12.4 Segitiga (B,A,C) dan (D,C,A) berorientasi konsisten, sedangkan (B,A,C) dan (A,C,D) berorientasi tidak konsisten.

Banyak algoritme berasumsi bahwa mesh berlipat ganda, dan selalu merupakan ide yang baik untuk memverifikasi properti ini untuk mencegah crash atau loop tak terbatas jika Anda diberikan mesh yang cacat sebagai input. Verifikasi ini bermuara pada pengecekan bahwa semua sisi berlipat ganda dan memeriksa bahwa semua simpul berlipat ganda dengan memverifikasi kondisi berikut:

- Setiap sisi dibagi oleh tepat dua segitiga.
- Setiap simpul memiliki satu lingkaran segitiga lengkap di sekitarnya.

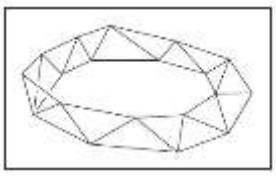
Manifold mesh memang nyaman, tetapi terkadang perlu untuk memungkinkan mesh memiliki tepi, atau batas. Jaring-jaring tersebut bukanlah manifold—suatu titik pada batas memiliki lingkungan yang terputus di satu sisi. Mereka tidak selalu kedap air. Namun, kita dapat mengendurkan persyaratan dari manifold mesh menjadi manifold dengan batas tanpa menyebabkan masalah dari kebanyakan algoritma pemrosesan mesh. Kondisi santai adalah:

- Setiap sisi digunakan oleh satu atau dua segitiga.
- Setiap simpul terhubung ke satu himpunan segitiga yang terhubung sisi.

Gambar 12.3 mengilustrasikan kondisi ini: dari kiri ke kanan, terdapat sebuah sisi dengan satu segitiga, sebuah titik yang segitiga-segitiga tetangganya berada dalam satu himpunan yang terhubung-sisi, dan sebuah titik dengan dua himpunan segitiga yang tidak terhubung yang melekat padanya.

Akhirnya, dalam aplikasinya penting untuk dapat membedakan "depan" atau "luar" dari permukaan dari "belakang" atau "dalam"—ini dikenal sebagai orientasi permukaan. Untuk sebuah segitiga tunggal, kita mendefinisikan orientasi berdasarkan urutan di mana simpul-simpulnya terdaftar: bagian depan adalah sisi dari mana tiga simpul segitiga disusun dalam urutan berlawanan arah jarum jam. Jala yang terhubung diorientasikan secara konsisten jika segitiga-segitiganya setuju pada sisi mana yang merupakan bagian depan—dan ini benar jika dan hanya jika setiap pasangan segitiga yang berdekatan berorientasi secara konsisten.

Dalam pasangan segitiga yang berorientasi konsisten, dua simpul bersama muncul dalam urutan yang berlawanan dalam daftar simpul dua segitiga (Gambar 12.4). Yang penting adalah konsistensi orientasi—beberapa sistem mendefinisikan bagian depan menggunakan urutan searah jarum jam daripada berlawanan arah jarum jam.



Gambar 12.5 Mobiusband berbentuk segitiga, yang tidak dapat diorientasikan.

Setiap mesh yang memiliki tepi non-manifold tidak dapat diorientasikan secara konsisten. Tapi mungkin juga mesh menjadi manifold yang valid dengan batas (atau bahkan manifold), namun tidak memiliki cara yang konsisten untuk mengarahkan segitiga—mereka bukan permukaan yang dapat diorientasikan. Contohnya adalah pita Mo¨bius yang ditunjukkan pada Gambar 12.5. Ini jarang menjadi masalah dalam praktiknya.

Penyimpanan Mesh Terindeks

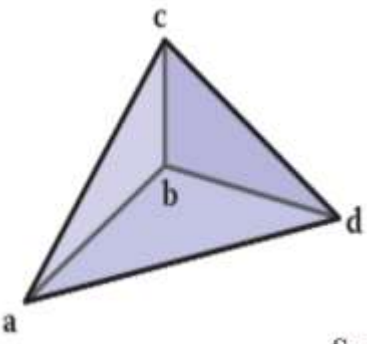
Jaring segitiga sederhana ditunjukkan pada Gambar 12.6. Anda dapat menyimpan ketiga segitiga ini sebagai entitas independen, masing-masing dalam bentuk ini:

```
Triangle {
vector3 vertexPosition[3]
}
```

Ini akan menghasilkan penyimpanan simpul tiga kali dan simpul lainnya masing-masing dua kali untuk total sembilan titik tersimpan (tiga simpul untuk masing-masing dari tiga segitiga). Atau Anda bisa mengatur untuk berbagi simpul umum dan menyimpan hanya empat, menghasilkan mesh simpul bersama. Logikanya, struktur data ini memiliki segitiga yang menunjuk ke simpul yang berisi data simpul:

```
Triangle {
Vertex v[3]
}
Vertex {
vector3 position // or other vertex data
}
```

Perhatikan bahwa entri dalam varrayareferensi, atau pointer, ke objek Vertex; simpul tidak terkandung dalam segitiga.



Segitiga yang terpisah:

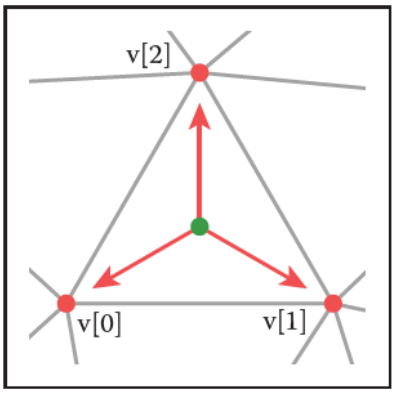
#	*Puncak 0*	*Puncak 1*	*Puncak 2*
0	(a_x, a_y, a_z)	(b_x, b_y, b_z)	(c_x, c_y, c_z)
1	(b_x, b_y, b_z)	(d_x, d_y, d_z)	(c_x, c_y, c_z)
2	(a_x, a_y, a_z)	(d_x, d_y, d_z)	(b_x, b_y, b_z)

Sudut yang digabung

Segitiga	
#	*Sudut*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Sudut	
#	*Posisi*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Gambar 12.6 Jala tiga segitiga dengan empat simpul, diwakili dengan segitiga terpisah (kiri) dan dengan simpul bersama (kanan).

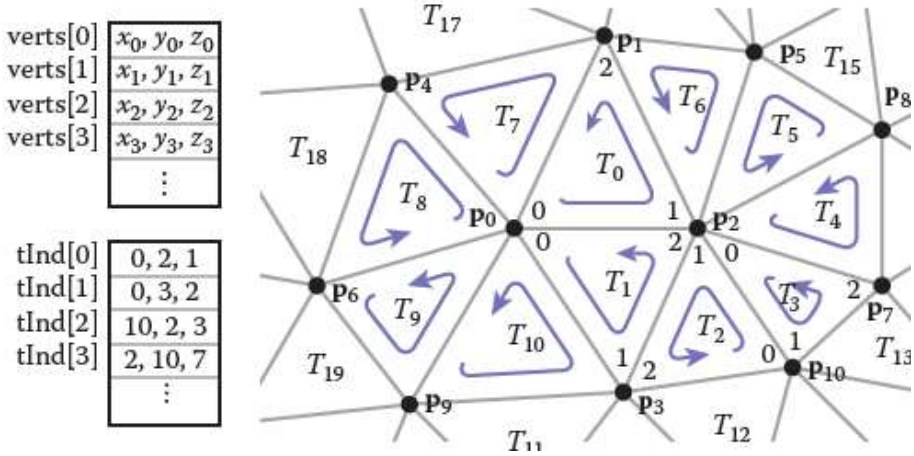


Gambar 12.7 Referensi segitiga ke simpul dalam mesh simpul bersama.

Dalam implementasinya, simpul dan segitiga biasanya disimpan dalam array, dengan referensi segitiga ke simpul ditangani dengan menyimpan indeks array:

```
IndexedMesh {
int tInd[nt][3]
vector3 verts[nv]
}
```

Indeks simpul ke-k dari segitiga ke-i ditemukan di tInd[i][k], dan posisi simpul tersebut disimpan dalam baris yang sesuai dari larik simpul; lihat Gambar 12.8 sebagai contoh. Cara menyimpan mesh simpul bersama ini adalah mesh segitiga terindeks.



Gambar 12.8 Jala segitiga yang lebih besar, dengan bagian representasinya sebagai jaring segitiga berindeks.

Segitiga terpisah atau simpul bersama akan bekerja dengan baik. Apakah ada keuntungan ruang untuk berbagi simpul? Jika jala kita memiliki nv simpul dan nt segitiga, dan jika diasumsikan bahwa data untuk float, pointer, dan semua memerlukan penyimpanan yang sama (asumsi yang meragukan), persyaratan ruang adalah sebagai berikut:

- **Triangle** Tiga vektor per segitiga, untuk unit penyimpanan 9nt;
- **IndexedMesh** Terindeks. Satu vektor per titik dan tiga int per segitiga, untuk unit penyimpanan 3nv + 3nt.

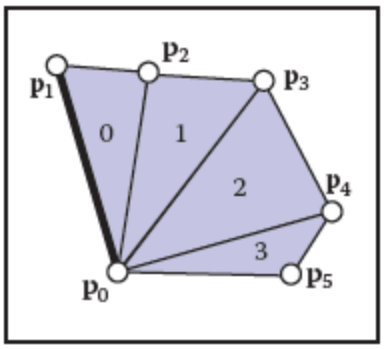
  Persyaratan penyimpanan relatif bergantung pada rasio nt terhadap nv.

Sebagai aturan praktis, jaring besar memiliki setiap simpul yang terhubung ke sekitar enam segitiga (walaupun bisa ada angka berapa pun untuk kasus ekstrem). Karena setiap segitiga terhubung ke tiga simpul, ini berarti bahwa secara umum ada segitiga dua kali lebih

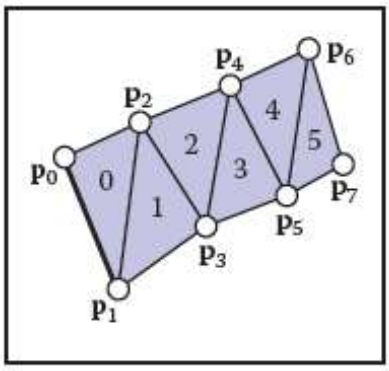
banyak dari simpul dalam mesh besar: nt 2nv. Dengan melakukan substitusi ini, kita dapat menyimpulkan bahwa kebutuhan penyimpanan adalah 18 nv untuk struktur Segitiga dan 9 nv untuk Indexed Mesh. Menggunakan simpul bersama mengurangi kebutuhan penyimpanan sekitar dua faktor; dan ini tampaknya bertahan dalam bentuk praktik di sebagian besar implementasi. Apakah faktor dua ini sepadan dengan komplikasinya? Saya pikir jawabannya adalah ya, dan itu menjadi kemenangan yang lebih besar segera setelah Anda mulai menambahkan "properti" ke simpul

Strip Segitiga dan Kipas

Jaring berindeks adalah representasi paling umum dalam memori dari jerat segitiga, karena mereka mencapai keseimbangan yang baik antara kesederhanaan, kenyamanan, dan kekompakan. Mereka juga biasanya digunakan untuk mentransfer mesh melalui jaringan dan antara aplikasi dan pipa grafis. Dalam aplikasi di mana lebih banyak kekompakan diinginkan, indeks simpul segitiga (yang mengambil dua pertiga dari ruang di mesh berindeks dengan hanya posisi di simpul) dapat diekspresikan lebih efisien menggunakan strip segitiga dan kipas segitiga.



Gambar 12.9 Penggemar segitiga

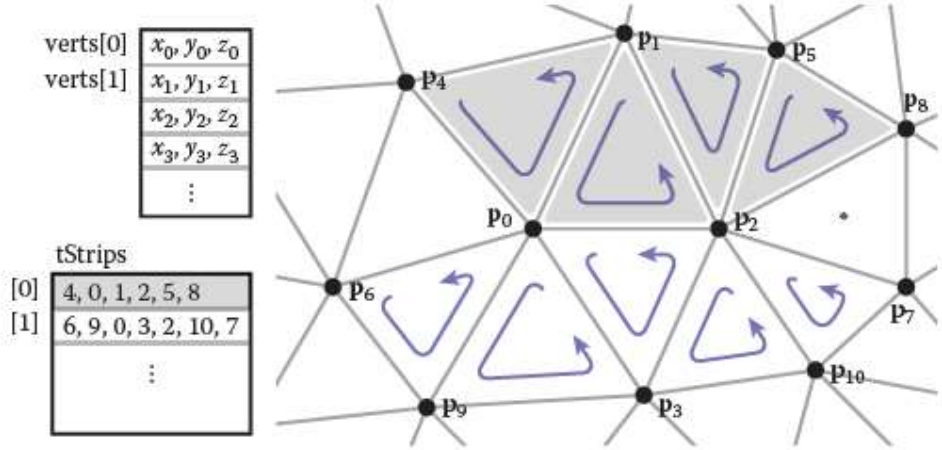


Gambar 12.10 Sebuah strip segitiga.

Sebuah kipas segitiga ditunjukkan pada Gambar 12.9. Dalam mesh yang diindeks, array segitiga akan berisi [(0, 1, 2), (0, 2, 3), (0, 3, 4), (0, 4, 5)]. Kami menyimpan 12 indeks simpul, meskipun hanya ada enam simpul yang berbeda. Dalam kipas segitiga, semua segitiga berbagi satu simpul yang sama, dan simpul lainnya menghasilkan satu set segitiga seperti baling-baling kipas yang dapat dilipat. Kipas pada gambar dapat ditentukan dengan urutan berikut [0,1, 2,3, 4, 5]: simpul pertama membentuk pusat, dan selanjutnya setiap pasangan simpul yang berdekatan (1-2, 2-3, dll.) menciptakan segitiga.

Strip segitiga adalah konsep yang serupa, tetapi berguna untuk rentang mata jaring yang lebih luas. Di sini, simpul ditambahkan bergantian atas dan bawah dalam strip linier seperti yang ditunjukkan pada Gambar 12.10. Garis segitiga pada gambar dapat ditentukan

dengan barisan [0 1 2 3 4 5 6 7], dan setiap barisan dari tiga simpul yang berdekatan (01-2, 1-2-3, dst.) membentuk sebuah segitiga. Untuk orientasi yang konsisten, setiap segitiga lainnya harus dibalik urutannya. Dalam contoh, ini menghasilkan segitiga (0, 1, 2), (2, 1, 3), (2, 3, 4), (4, 3, 5), dll. Untuk setiap simpul baru yang masuk , simpul tertua dilupakan dan urutan dua simpul yang tersisa ditukar. Lihat Gambar 12.11 untuk contoh yang lebih besar.



Gambar 12.11 Dua strip segitiga dalam konteks jaring yang lebih besar. Perhatikan bahwa tidak ada strip yang dapat diperluas untuk menyertakan segitiga yang ditandai dengan tanda bintang.

Baik dalam strip maupun fan, n +2 simpul cukup untuk menggambarkan n segitiga—penghematan substansial atas 3n simpul yang dibutuhkan oleh mesh berindeks standar. Strip segitiga panjang akan menghemat kira-kira tiga kali lipat jika programnya vertexbound.

Tampaknya strip segitiga hanya berguna jika strip sangat panjang, tetapi bahkan strip yang relatif pendek sudah mendapatkan sebagian besar manfaat. Penghematan ruang penyimpanan (hanya untuk indeks simpul) adalah sebagai berikut:

strip length	1	2	3	4	5	6	7	8	16	100	∞
relative size	1.00	0.67	0.56	0.50	0.47	0.44	0.43	0.42	0.38	0.34	0.33

Jadi, sebenarnya, ada penurunan yang agak cepat seiring dengan semakin panjangnya potongan-potongan itu. Jadi, bahkan untuk mesh yang tidak terstruktur, sangat bermanfaat untuk menggunakan beberapa algoritma serakah untuk mengumpulkannya menjadi shortstrip.

Struktur Data untuk Konektivitas Mesh

Jaring, strip, dan kipas yang diindeks semuanya bagus, representasi ringkas untuk jerat statis. Namun, mereka tidak dengan mudah mengizinkannya untuk dimodifikasi. Untuk mengedit mesh secara efisien, diperlukan struktur data yang lebih rumit untuk menjawab pertanyaan seperti:

- Diberikan sebuah segitiga, berapakah tiga segitiga yang berdekatan?
- Diberikan sebuah tepi, manakah dua segitiga yang membaginya?
- Diberikan sebuah simpul, wajah mana yang membaginya?
- Diberikan sebuah simpul, tepi mana yang membaginya?

Ada banyak struktur data untuk mesh segitiga, mesh poligonal, dan mesh poligonal berlubang (lihat catatan di akhir bab untuk referensi). Dalam banyak aplikasi, mesh sangat besar, sehingga representasi yang efisien dapat menjadi sangat penting.

Implementasi yang paling mudah, meskipun membengkak, adalah memiliki tiga jenis, Vertex, Edge, dan Triangle, dan hanya menyimpan semua hubungan secara langsung:

```
Triangle {
Vertex v[3]
Edge e[3]
}
Edge {
Vertex v[2]
Triangle t[2]
}
Vertex {
Triangle t[]
Edge e[]
}
```

Ini memungkinkan kita untuk langsung mencari jawaban atas pertanyaan konektivitas di atas, tetapi karena semua informasi ini saling terkait, ia menyimpan lebih dari yang sebenarnya dibutuhkan. Selain itu, menyimpan konektivitas dalam simpul membuat struktur data dengan panjang variabel (karena simpul dapat memiliki jumlah tetangga yang acak), yang umumnya kurang efisien untuk diterapkan. Daripada berkomitmen untuk menyimpan semua hubungan ini secara eksplisit, yang terbaik adalah mendefinisikan antarmuka kelas untuk menjawab pertanyaan-pertanyaan ini, di balik struktur data yang lebih efisien dapat bersembunyi. Ternyata kita hanya dapat menyimpan beberapa konektivitas dan secara efisien memulihkan informasi lain bila diperlukan.

Array ukuran tetap di kelas Edge dan Triangle menunjukkan bahwa akan lebih efisien untuk menyimpan informasi konektivitas di sana. Faktanya, untuk mesh poligon, di mana poligon memiliki jumlah edge dan vertex yang berubah-ubah, hanya edge yang memiliki informasi konektivitas ukuran tetap, yang mengarah ke banyak struktur data mesh tradisional yang didasarkan pada edge. Tetapi untuk jerat segitiga saja, menyimpan konektivitas di wajah (lebih sedikit) menarik.

Struktur data yang baik harus cukup ringkas dan memungkinkan jawaban yang efisien untuk semua kueri kedekatan. Efisien berarti waktu konstan: waktu untuk menemukan tetangga tidak boleh bergantung pada ukuran mata jaring. Kita akan melihat tiga struktur data untuk mesh, satu berdasarkan segitiga dan dua berdasarkan tepi.

Struktur Segitiga-Tetangga

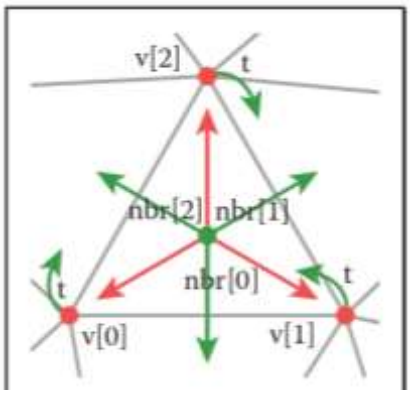
Kita dapat membuat struktur data mesh padat berdasarkan segitiga dengan menambah mesh simpul bersama dasar dengan pointer dari segitiga ke tiga segitiga yang bertetangga, dan pointer dari setiap sudut teks tepat di atas kepala segitiga (tidak masalah yang mana); lihat Gambar 12.12:

```
Triangle {
Triangle nbr[3];
Vertex v[3];
}
Vertex {
// ... per-vertex data ...
Triangle t; // any adjacent tri
}
```

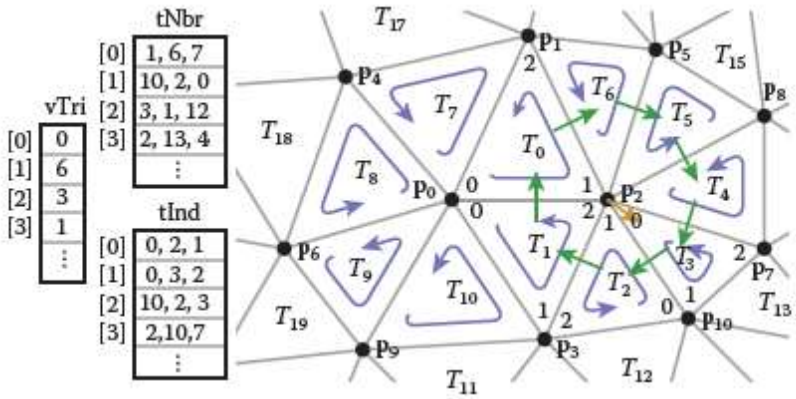
Dalam array Triangle.nbr, entri ke-k menunjuk ke segitiga tetangga yang berbagi simpul k dan k +1. Kami menyebut struktur ini sebagai struktur segitiga-tetangga. Mulai dari

array mesh berindeks standar, dapat diimplementasikan dengan dua array tambahan: satu yang menyimpan tiga tetangga dari setiap segitiga, dan satu yang menyimpan satu segitiga tetangga untuk setiap simpul (lihat Gambar 12.13 sebagai contoh):

```
Mesh {
// ... per-vertex data ...
int tInd[nt][3]; // vertex indices
int tNbr[nt][3]; // indices of neighbor triangles
int vTri[nv]; // index of any adjacent triangle
}
```



Gambar 12.12 Referensi antara segitiga dan simpul dalam struktur segitigatetangga.



Gambar 12.13 Struktur segitiga-tetangga seperti yang dikodekan dalam array, dan urutan yang diikuti dalam melintasi segitiga tetangga dari simpul 2.

Jelas segitiga bertetangga dan simpul dari sebuah segitiga dapat ditemukan secara langsung dalam struktur data, tetapi dengan menggunakan informasi ketetanggaan segitiga ini dengan hati-hati juga dimungkinkan untuk menjawab pertanyaan konektivitas tentang simpul dalam waktu yang konstan. Idenya adalah untuk berpindah dari segitiga ke segitiga, hanya mengunjungi segitiga yang berdekatan dengan simpul yang relevan. Jika segitiga t memiliki simpul v sebagai simpul ke-knya, maka segitiga t.nbr[k] adalah segitiga berikutnya yang mengelilingiv searah jarum jam. Pengamatan ini mengarah pada algoritma berikut untuk melintasi semua segitiga yang berdekatan dengan simpul yang diberikan:

```
TrianglesOfVertex(v) {
t = v.t
do {
find i such that (t.v[i] == v)
t = t.nbr[i]
```

```
} while (t != v.t)
}
```

Operasi ini menemukan setiap segitiga berikutnya dalam waktu yang konstan—walaupun pencarian diperlukan untuk menemukan posisi titik pusat dalam setiap daftar titik segitiga, daftar titik memiliki ukuran konstan sehingga pencarian membutuhkan waktu yang konstan. Namun, pencarian itu canggung dan membutuhkan percabangan ekstra.

Perbaikan kecil dapat menghindari pencarian ini. Masalahnya adalah begitu kita mengikuti penunjuk dari satu segitiga ke segitiga berikutnya, kita tidak tahu dari mana asalnya: kita harus mencari simpul segitiga untuk menemukan simpul yang menghubungkan kembali ke segitiga sebelumnya. Untuk mengatasi ini, alih-alih menyimpan pointer ke segitiga tetangga, kita dapat menyimpan pointer ke segitiga tertentu dengan menyimpan indeks dengan pointer:

```
Triangle {
Edge nbr[3];
Vertex v[3];
}
Edge { // the i-th edge of triangle t
Triangle t;
int i; // in {0,1,2}
}
Vertex {
// ... per-vertex data ...
Edge e; // any edge leaving vertex
}
```

Dalam prakteknya Edge disimpan dengan meminjam dua bit penyimpanan dari indeks segitiga t untuk menyimpan indeks edge i, sehingga total kebutuhan penyimpanan tetap sama.

Dalam struktur ini, array tetangga untuk segitiga menunjukkan sisi mana dari segitiga tetangga yang dibagi dengan tiga tepi segitiga itu. Dengan informasi tambahan ini, kita selalu tahu di mana menemukan segitiga asli, yang mengarah ke sebuah invarian dari struktur data: untuk setiap tepi ke-j dari sembarang segitiga t,

$$t.\text{nbr}[j].t.\text{nbr}[t.\text{nbr}[j].i] \;.\; t == t$$

Mengetahui tepi mana yang kita masuki memungkinkan kita mengetahui dengan segera tepi mana yang harus dilalui untuk melanjutkan perjalanan di sekitar simpul, yang mengarah ke algoritme yang disederhanakan:

```
TrianglesOfVertex(v) {
{t, i} = v.e;
do {
{t, i} = t.nbr[i];
i = (i+1) mod 3;
} while (t != v.e.t);
}
```

Struktur segitiga-tetangga cukup kompak. Untuk mesh dengan hanya posisi titik, kami menyimpan empat angka(tiga koordinat dan satu sisi) per titik dan enam (tiga indeks titik dan tiga sisi) per wajah, dengan total $4n_v + 6n_t = 16n_v$ unit penyimpanan per titik, dibandingkan dengan $9n_v$ untuk jaring indeks dasar.

Struktur tetangga segitiga seperti yang disajikan di sini hanya berfungsi untuk manifold mesh, karena bergantung pada kembali ke segitiga awal untuk mengakhiri traversal tetangga simpul, yang tidak akan terjadi pada simpul batas yang tidak memiliki siklus segitiga penuh. Namun, tidak sulit untuk menggeneralisasikannya ke manifold dengan batas, dengan

memperkenalkan nilai sentinel yang sesuai (seperti 1) untuk tetangga segitiga batas dan berhati-hati bahwa simpul batas menunjuk ke segitiga tetangga yang paling berlawanan arah jarum jam, daripada ke sembarang segitiga.

Struktur Tepi Bersayap

Salah satu struktur data mesh yang banyak digunakan yang menyimpan informasi konektivitas di tepi dan bukan di permukaan adalah struktur data bersayap-tepi. Struktur data ini menjadikan edge sebagai warga kelas satu dari struktur data, seperti yang diilustrasikan pada Gambar 12.14 dan 12.15.

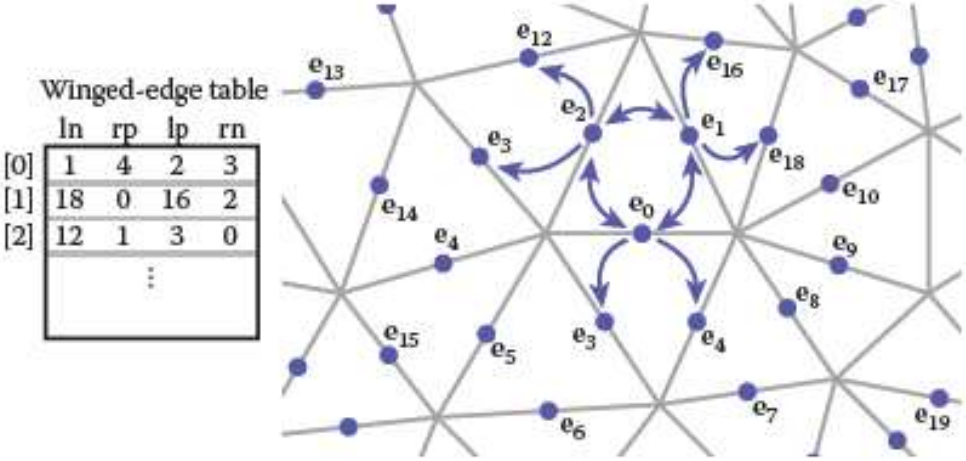
Dalam jaring tepi bersayap, setiap tepi menyimpan penunjuk ke dua simpul yang dihubungkannya (simpul kepala dan ekor), dua wajah yang menjadi bagiannya (sisi kiri dan kanan), dan yang terpenting, tepi berikutnya dan sebelumnya dalam traversal berlawanan arah jarum jam dari wajah kiri dan kanannya (Gambar 12.16). Setiap vertex dan face juga menyimpan pointer ke satu edge arbitrer yang menghubungkannya:

```
Edge {
Edge lprev, lnext, rprev, rnext;
Vertex head, tail;
Face left, right;
}
Face {
// ... per-face data ...
Edge e; // any adjacent edge
}
Vertex {
// ... per-vertex data ...
Edge e; // any incident edge
}
```

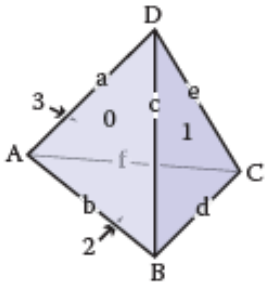
Struktur winged-edgedata mendukung akses waktu-konstan ke tepi face atau vertex, dan dari edge tersebut vertex atau face yang berdampingan dapat ditemukan:

```
EdgesOfVertex(v) {
e = v.e;
do {
if (e.tail == v)
e = e.lprev;
else
e = e.rprev;
} while (e != v.e);
}
EdgesOfFace(f) {
e = f.e;
do {
if (e.left == f)
e = e.lnext;
else
e = e.rnext;
} while (e != f.e);
}
```

Algoritma dan struktur data yang sama ini akan bekerja dengan baik pada poligon mesh yang tidak terbatas pada segitiga; ini adalah salah satu keuntungan penting dari struktur berbasis tepi.



Gambar 12.14 Contoh struktur mesh bersayap, disimpan dalam array.

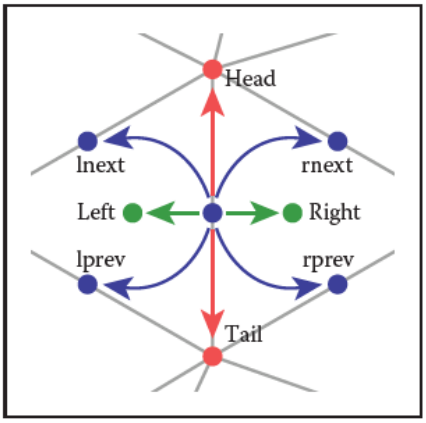


Edge	*Vertex 1*	*Vertex 2*	*Face left*	*Face right*	*Pred left*	*Succ left*	*Pred right*	*Succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	e	b	d

Vertex	*Edge*
A	a
B	d
C	d
D	e

Face	*Edge*
0	a
1	c
2	d
3	a

Gambar 12.15 Tetrahedron dan elemen terkait untuk struktur data bersayap. Dua meja kecil itu tidak unik; setiap vertex dan face menyimpan salah satu edge yang diasosiasikan.



Gambar 12.16 Referensi dari tepi ke tepi tetangga, wajah, dan simpul dalam struktur tepi bersayap.

Seperti halnya struktur data lainnya, struktur winged-edgedata membuat berbagai pengorbanan waktu/ruang. Misalnya, kita dapat menghilangkan referensi sebelumnya. Hal ini membuat lebih sulit untuk melintasi searah jarum jam di sekitar wajah atau berlawanan arah jarum jam di sekitar simpul, tetapi ketika kita perlu mengetahui tepi sebelumnya, kita selalu dapat mengikuti tepi penerus dalam lingkaran sampai kita kembali ke tepi aslinya. Ini menghemat ruang, tetapi membuat beberapa operasi lebih lambat. (Lihat catatan bab untuk informasi lebih lanjut tentang pengorbanan ini).

Struktur Setengah Tepi

Struktur tepi bersayap cukup elegan, tetapi memiliki satu kecanggungan yang tersisa—harus terus-menerus memeriksa ke arah mana tepi diorientasikan sebelum pindah ke tepi berikutnya. Pemeriksaan ini secara langsung analog dengan pencarian yang kita lihat dalam versi dasar dari struktur tetangga segitiga: kita mencari untuk mengetahui apakah kita memasuki tepi sekarang dari kepala atau dari ekor. Solusinya juga hampir tidak dapat dibedakan: daripada menyimpan data untuk setiap tepi, kami menyimpan data untuk setiap setengah tepi. Ada satu setengah sisi untuk masing-masing dari dua segitiga yang berbagi sisi, dan dua setengah sisi berorientasi berlawanan, masing-masing berorientasi secara konsisten dengan segitiganya sendiri.

Data biasanya disimpan di tepi dibagi antara dua setengah tepi. Setiap setengah tepi menunjuk ke wajah di sisi tepi dan ke kepala tegak, dan masing-masing berisi penunjuk tepi untuk wajahnya. Itu juga menunjuk ke tetangganya di sisi lain tepi, dari mana separuh informasi lainnya dapat ditemukan. Seperti tepi-bersayap, tepi-setengah dapat berisi penunjuk ke tepi-setengah sebelumnya dan berikutnya di sekitar wajahnya, atau hanya ke tepi-setengah berikutnya. Kami akan menunjukkan contoh yang menggunakan satu pointer.

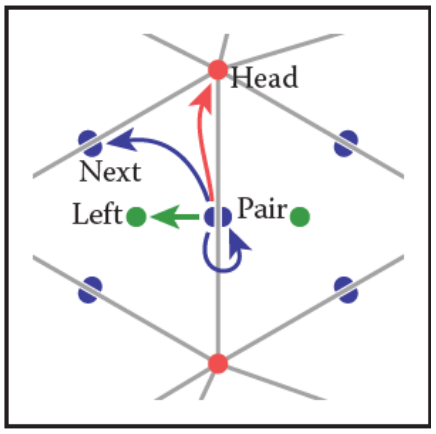
```
HEdge {
HEdge pair, next;
Vertex v;
Face f;
}
Face {
// ... per-face data ...
HEdge h; // any h-edge of this face
}
Vertex {
// ... per-vertex data ...
```

HEdge h; // any h-edge pointing toward this vertex
}

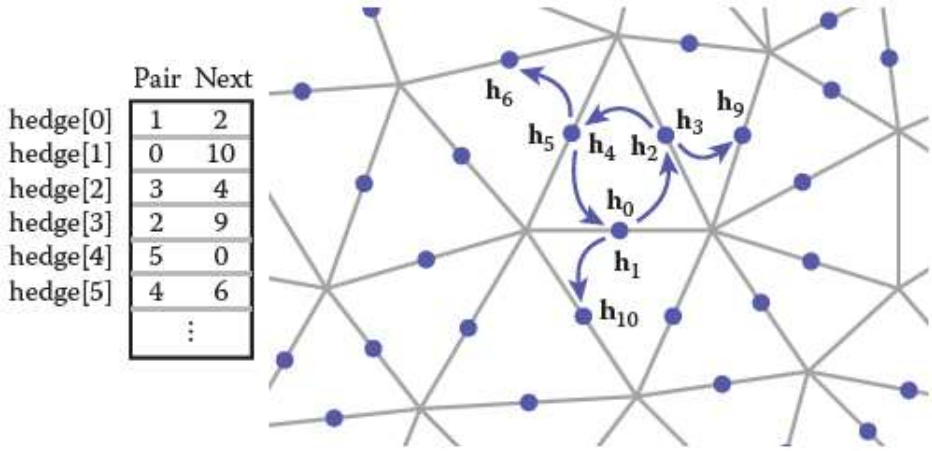
Struktur tepi-setengah traversiga sama seperti struktur tepi bersayap traversi, kecuali bahwa kita tidak perlu lagi memeriksa orientasi, dan kita mengikuti penunjuk pasangan untuk mengakses tepi-tepi pada muka yang berlawanan.

```
EdgesOfVertex(v) {
h = v.h;
do {
h = h.pair.next;
} while (h != v.h);
}
EdgesOfFace(f) {
h = f.h;
do {
h = h.next;
} while (h != f.h);
}
```

Verteks melintang di sini searah jarum jam, yang diperlukan karena menghilangkan penunjuk sebelumnya dari struktur.



Gambar 12.17 Referensi dari setengah tepi ke komponen mesh tetangganya.



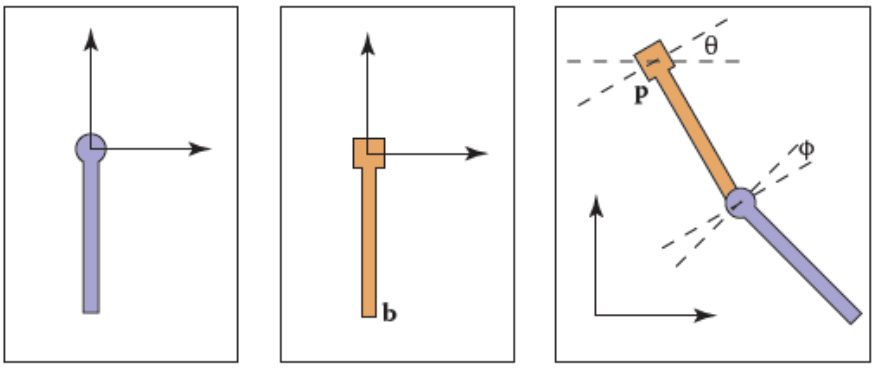
Gambar 12.18 Contoh struktur mesh setengah tepi, disimpan dalam array.

Karena half-edge umumnya dialokasikan berpasangan (setidaknya dalam mesh tanpa batas), banyak implementasi dapat menghilangkan pointer pasangan. Misalnya, dalam implementasi berdasarkan pengindeksan array (seperti yang ditunjukkan pada Gambar 12.18), array dapat diatur sedemikian rupa sehingga tepi genap i selalu berpasangan dengan tepi i +1 dan tepi bernomor ganjil j selalu berpasangan dengan tepi j 1.

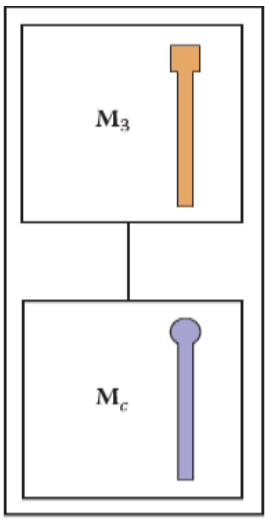
Selain algoritma traversal sederhana yang ditunjukkan dalam bab ini, ketiga struktur topologi mesh ini dapat mendukung operasi "operasi mesh" dari berbagai jenis, seperti membelah atau menciutkan simpul, menukar tepi, menambah atau menghilangkan segitiga, dll.

12.2 GRAFIS SCENE

Segitiga mesh mengelola kumpulan segitiga yang membentuk objek dalam sebuah scene, tetapi masalah universal lainnya dalam aplikasi grafis adalah mengatur objek pada posisi yang diinginkan. Seperti yang kita lihat di Bab 6, ini dilakukan dengan menggunakan transformasi, tetapi scene kompleks dapat berisi banyak sekali transformasi dan mengaturnya dengan baik membuat scene lebih mudah untuk dimanipulasi. Sebagian besar scene mengakui organisasi hierarkis, dan transformasi dapat diatur menurut hierarki ini menggunakan grafis scene.



Gambar 12.19 Sebuah bandul berengsel. Di sebelah kiri adalah dua bagian dalam sistem koordinat "lokal" mereka. Engsel bagian bawah berada di titik b dan sambungan untuk bagian bawah berada di titik asalnya. Derajat kebebasan benda rakitan adalah sudut (θ,ϕ) dan letak p dari engsel atas.



Gambar 12.20 Grafis scene untuk bandul berengsel Gambar 12.19.

Untuk memotivasi struktur data grafis scene, kita akan menggunakan pendulum berengsel yang ditunjukkan pada Gambar 12.19. Pertimbangkan bagaimana kita akan menggambar bagian atas bandul:

M1 = rotate(ϑ)

M2 = translate(**p**)

M3 =**M**2**M**1

Apply**M**3 ke semua titik yang berada di atas bandul

Bagian bawah lebih rumit, tetapi kita dapat mengambil keuntungan dari fakta bahwa bandul tersebut melekat pada bagian bawah bandul atas pada titik b dalam sistem koordinat lokal. Pertama, kita memutar bandul bawah sehingga membentuk sudut relatif terhadap posisi awalnya. Kemudian, kita gerakkan sehingga engsel atasnya berada di titik b. Sekarang ia berada pada posisi yang sesuai di koordinat lokal bandul atas, dan kemudian dapat dipindahkan bersama dengan sistem koordinat itu. Transformasi komposit untuk bandul bawah adalah:

M*a* = rotate(φ)

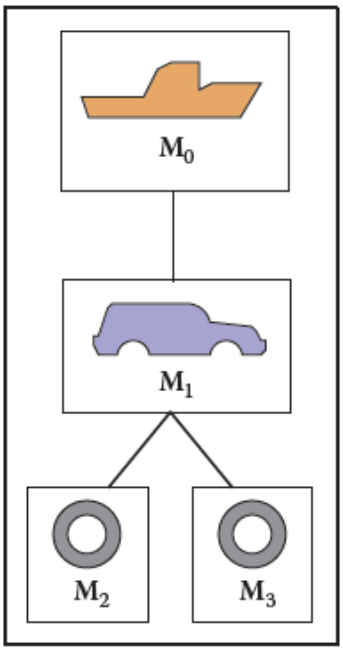
M*b* = translate(**b**)

M*c* = **M***b***M***a*

M*d* = **M**3**M***c*

Apply**M***d* ke semua titik yang ada dibawah bandul

Jadi, kita melihat bahwa bandul bawah tidak hanya hidup dalam sistem koordinat lokalnya sendiri, tetapi juga sistem koordinat itu sendiri bergerak bersama dengan bandul atas. Kami dapat mengkodekan pendulum dalam struktur data yang membuat pengelolaan masalah sistem koordinat ini lebih mudah, seperti yang ditunjukkan pada Gambar 12.20. Matriks yang sesuai untuk diterapkan pada suatu objek hanyalah produk dari jatuhnya matriks dalam rantai dari objek ke akar struktur data. Sebagai contoh, perhatikan model feri yang memiliki mobil yang dapat bergerak bebas di geladak feri, dan roda yang masing-masing bergerak relatif terhadap mobil seperti yang ditunjukkan pada Gambar 12.21.



Gambar 12.21 Feri, mobil di feri, dan roda mobil (hanya dua yang ditampilkan) disimpan dalam grafis scene.

Seperti halnya bandul, setiap benda harus ditransformasikan oleh perkalian matriks-matriks pada lintasan dari akar ke benda:

- transformasi feri menggunakan M0;
- transformasi bodi mobilmenggunakan M0M1;
- transformasi roda kirimenggunakan M0M1M2;
- transformasi roda kiri menggunakan M0M1M3.

Implementasi yang efisien dapat dicapai dengan menggunakan amatrixstack, struktur data yang didukung oleh banyak API. Tumpukan matriks dimanipulasi menggunakan operasi push dan pop yang menambahkan dan menghapus matriks dari sisi kanan produk matriks. Misalnya, menelepon:

push(**M**0)
push(**M**1)
push(**M**2)

membuat matriks aktif M = M0M1M2. Panggilan berikutnya ke pop() menghapus matriks terakhir yang ditambahkan sehingga matriks aktif menjadi M = M_0M_1. Menggabungkan tumpukan matriks dengan traversal rekursif dari grafis scene memberi kita:

function traverse(node)
push(**M**local)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()

Ada banyak variasi pada grafis scene tetapi semuanya mengikuti ide dasar di atas.

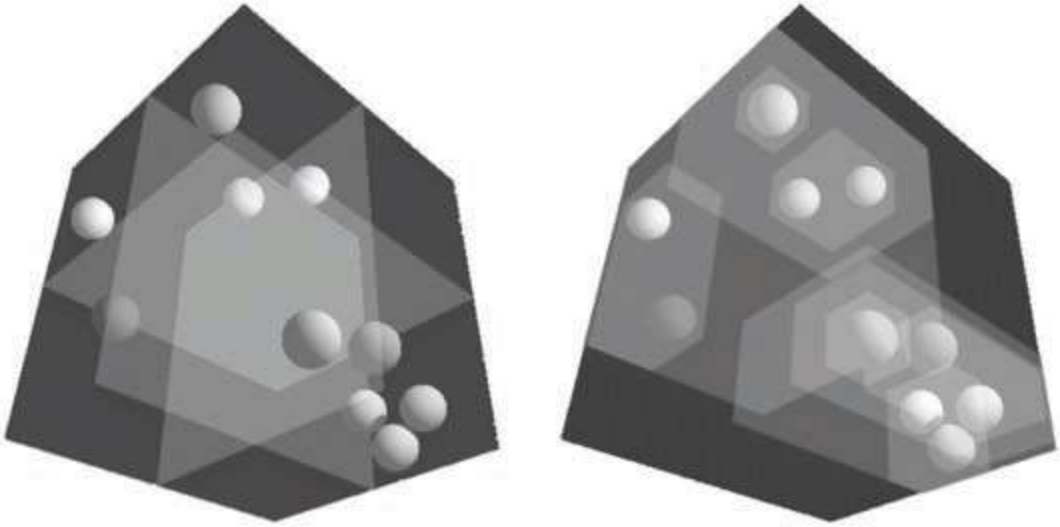
12.3 STRUKTUR DATA SPASIAL

Dalam banyak, jika tidak semua, aplikasi grafis, kemampuan untuk dengan cepat menemukan objek geometris di wilayah ruang tertentu adalah penting. Pelacak sinar perlu menemukan objek yang bersinggungan; aplikasi interaktif yang menavigasi lingkungan perlu menemukan objek yang terlihat dari sudut pandang tertentu; permainan dan simulasi fisik memerlukan pendeteksian kapan dan di mana objek bertabrakan. Semua kebutuhan ini dapat didukung oleh berbagai struktur data spasial yang dirancang untuk mengorganisasikan objek-objek dalam ruang sehingga dapat dicari secara efisien.

Pada bagian ini kita akan membahas contoh dari tiga kelas umum struktur data spasial. Struktur yang mengelompokkan objek ke dalam hierarki adalah skema partisi objek: objek dibagi menjadi grup yang terpisah, tetapi grup tersebut mungkin berakhir tumpang tindih dalam ruang. Struktur yang membagi ruang menjadi wilayah yang terpisah-pisah adalah skema partisi ruang: ruang dibagi menjadi partisi yang terpisah, tetapi satu objek mungkin harus memotong lebih dari satu partisi. Skema partisi ruang bisa teratur, di mana ruang dibagi menjadi bagian-bagian yang berbentuk seragam, atau tidak beraturan, di mana ruang dibagi secara adaptif menjadi bagian-bagian yang tidak beraturan, dengan bagian-bagian yang lebih kecil di mana ada objek yang lebih banyak dan lebih kecil.

Kami akan menggunakan raytracing sebagai motivasi utama saat mendiskusikan struktur ini, meskipun mereka semua juga dapat digunakan untuk melihat pemusnahan atau deteksi tabrakan. Dalam Bab 4, semua objek dilingkarkan saat memeriksa persimpangan. Untuk N objek, ini adalah pencarian linier O(N) dan dengan demikian lambat untuk scene besar. Seperti kebanyakan masalah pencarian, perpotongan objek sinar dapat dihitung dalam waktu sub-linier menggunakan teknik "membagi dan menaklukkan", asalkan kita dapat membuat struktur data terurut sebagai praproses. Ada banyak teknik untuk melakukan ini.

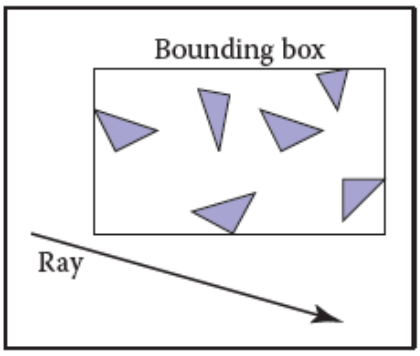
Bagian ini membahas tiga teknik ini secara rinci: hierarki volume pembatas (Rtiling&Whitted,1980;Whitted,1980;Goldsmith&Salmon,1987), subdivisi spasial seragam (Cleary, Wyvill, Birtwistle, & Vatti, 1983; Fujimoto, Tanaka, & Iwata, 1986 ; Amanatides & Woo, 1987), dan partisi ruang biner (Glassner, 1984; Jansen, 1986; Havran, 2000). Contoh dari dua strategi pertama ditunjukkan pada Gambar 12.22.



Gambar 12.22 Kiri: partisi ruang yang seragam. Kanan: hierarki kotak pembatas adaptif. Gambar milik David DeMarle.

Kotak pembatas

Operasi kunci dalam kebanyakan skema percepatan-persimpangan adalah menghitung perpotongan sebuah sinar dengan kotak pembatas (Gambar 12.23). Ini berbeda dari tes persimpangan konvensional di mana kita tidak perlu tahu di mana sinar mengenai kotak; kita hanya perlu tahu apakah itu mengenai kotak.



Gambar 12.23 Sinar hanya diuji untuk perpotongan dengan permukaan jika mengenai kotak pembatas.

Untuk membangun algoritma perpotongan kotak sinar, kita mulai dengan mempertimbangkan sinar 2D yang vektor arahnya memiliki komponen x dan y positif. Kita dapat menggeneralisasi ini ke sinar 3D sewenang-wenang nanti. Kotak pembatas 2D didefinisikan oleh dua garis horizontal dan dua garis vertikal:

$$x = x\text{min},$$
$$x = x\text{max},$$
$$y = y\text{min},$$
$$y = y\text{max}.$$

Titik-titik yang dibatasi oleh garis-garis ini dapat digambarkan dalam notasi interval:

$(x, y) \in [x\text{min}, x\text{max}] \times [y\text{min}, y\text{max}]$.

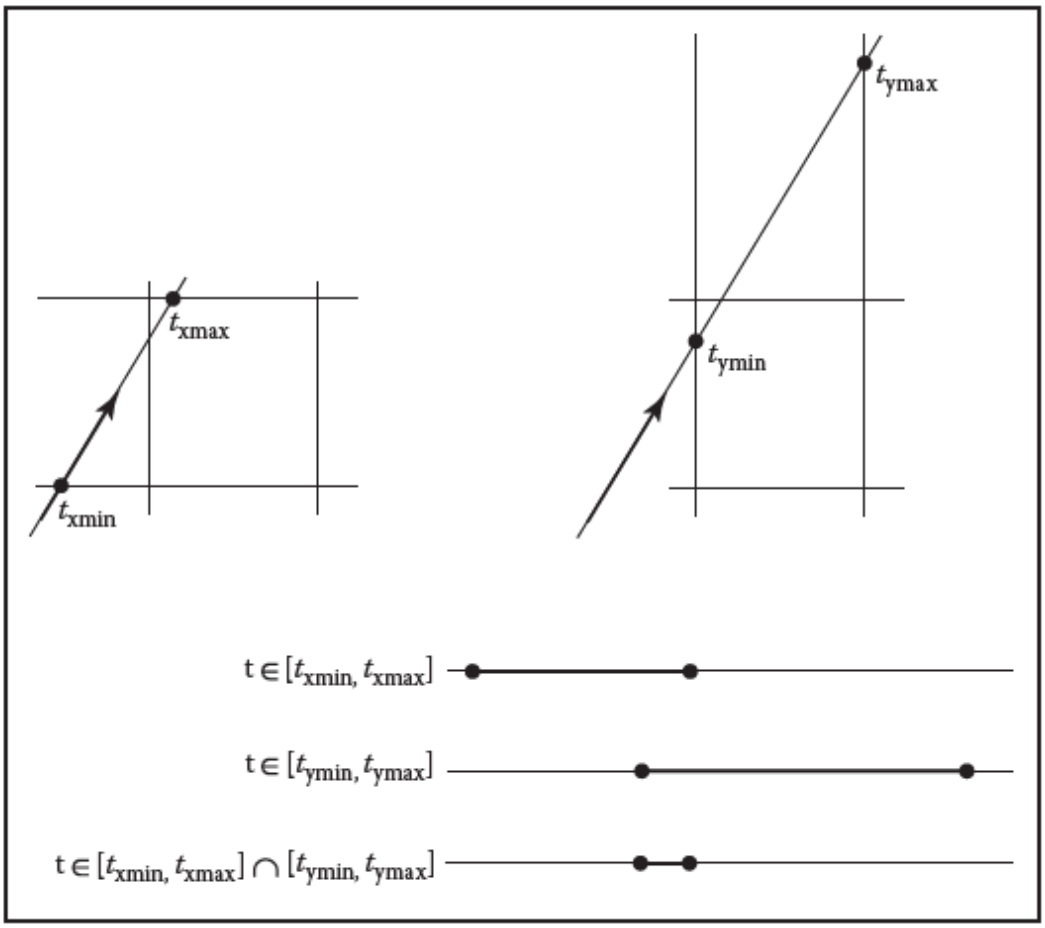
Seperti yang ditunjukkan pada Gambar 12.24, uji persimpangan dapat diutarakan dalam istilah interval ini. Pertama, kita menghitung parameter sinar di mana sinar mengenai garis $x = x_{min}$:

$$t_{min} = \frac{x_{min} - x_e}{x_d}$$

Kami kemudian membuat perhitungan serupa untuk txmax, tymin, andtymax. Sinar mengenai kotak jika dan hanya jika interval $[t_{xmin}, t_{xmax}]$ dan $[t_{ymin}, t_{ymax}]$ tumpang tindih, yaitu perpotongannya tidak kosong. Dalam pseudocode algoritma ini adalah:

*t*xmin = (*x*min − *xe*)*/xd*
*t*xmax = (*x*max − *xe*)*/xd*
*t*ymin = (*y*min − *ye*)*/yd*
*t*ymax = (*y*max − *ye*)*/yd*
if (*t*xmin > *t*ymax) or (*t*ymin > *t*xmax) **then**
return false
else
return true

Pernyataan if mungkin tampak tidak jelas. Untuk melihat logikanya, perhatikan bahwa tidak ada tumpang tindih jika interval pertama seluruhnya ke kanan atau seluruhnya ke kiri interval kedua.



Gambar 12.24 Sinar akan berada di dalam interval $x \in [x_{min}, x_{max}]$ untuk beberapa interval dalam ruang parameternya $t [t_{xmin}, t_{xmax}]$. Interval yang sama ada untuk interval y. Sinar memotong kotak jika berada pada interval x dan interval y pada saat yang bersamaan, yaitu perpotongan dua interval satu dimensi tidak kosong.

Hal pertama yang harus kita bahas adalah kasus ketika xd atau yd negatif. Jika xd negatif, maka sinar akan mencapai x max sebelum x min. Jadi kode untuk menghitung t_{xmin} dan t_{xmax} diperluas menjadi:

if (*xd* ≥ 0) **then**
*t*xmin = (*x*min − *xe*)*/xd*
*t*xmax = (*x*max − *xe*)*/xd*
else
*t*xmin = (*x*max − *xe*)*/xd*
*t*xmax = (*x*min − *xe*)*/xd*

Perluasan kode serupa harus dilakukan untuk kasus y. Perhatian utama adalah bahwa sinar horizontal dan vertikal memiliki nilai nol untuk y_d dan x_d, masing-masing. Ini akan menyebabkan pembagian dengan nol yang mungkin menjadi masalah. Namun, sebelum menangani ini secara langsung, kami memeriksa apakah komputasi titik mengambang IEEE menangani kasus ini dengan baik untuk kami. Ingat kembali dari Bagian 1.5 aturan untuk membagi dengan nol: untuk setiap bilangan real positif a,

$+a/0 = +\infty$;
$-a/0 = -\infty$.

Pertimbangkan kasus sinar vertikal di mana $x_d = 0$ dan $y_d > 0$. Kemudian kita dapat menghitung

$$t_{min} = \frac{x_{min} - x_e}{0}$$
$$t_{max} = \frac{x_{max} - x_e}{0}$$

Ada tiga kemungkinan yang menarik:

1. *xe* ≤ *x*min (no hit);
2. *x*min < *xe* < *x*max (hit);
3. *x*max ≤ *xe* (no hit).

Untuk kasus pertama yang kita miliki

$$t_{min} = \frac{jumlah\ positif}{0}$$
$$t_{max} = \frac{jumlah\ positif}{0}$$

Ini menghasilkan interval (txmin,txmin)=(∞,∞). Interval itu tidak akan tumpang tindih dengan interval apa pun, jadi tidak akan ada pukulan, seperti yang diinginkan. Untuk kasus kedua, kita punya

$$t_{min} = \frac{jumlah\ negatif}{0}$$
$$t_{max} = \frac{jumlah\ negatif}{0}$$

Ini menghasilkan interval (t_{xmin}, t_{xmin}) = (−∞, ∞) yang akan tumpang tindih dengan semua interval dan dengan demikian akan menghasilkan hit yang diinginkan. Kasus ketiga menghasilkan interval (−∞, −∞) yang tidak menghasilkan pukulan, seperti yang diinginkan. Karena kasus ini bekerja seperti yang diinginkan, kami tidak memerlukan pemeriksaan khusus untuk mereka. Seperti yang sering terjadi, konvensi floating point IEEE adalah sekutu kita. Namun, masih ada masalah dengan pendekatan ini. Pertimbangkan segmen kode:

if (*xd* ≥ 0) **then**
*t*min = (*x*min − *xe*)*/xd*
*t*max = (*x*max − *xe*)*/xd*
else

*t*min = (*x*max − *xe*)*/xd*
*t*max = (*x*min − *xe*)*/xd*

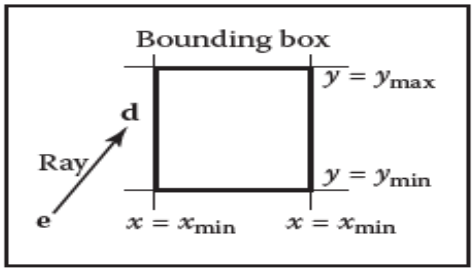
Kode ini rusak ketika xd = 0. Hal ini dapat diatasi dengan pengujian pada resiprokal xd (A. Williams, Barrus, Morley, & Shirley, 2005):

a = 1*/xd*
if (*a* ≥ 0) **then**
*t*min = *a*(*x*min − *xe*)
*t*max = *a*(*x*max − *xe*)
else
*t*min = *a*(*x*max − *xe*)
*t*max = *a*(*x*min − *xe*)

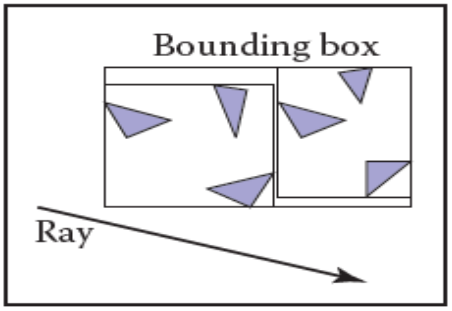
Kotak Pembatas Hirarki

Ide dasar kotak pembatas hierarkis dapat dilihat dengan taktik umum menempatkan kotak pembatas 3D sejajar sumbu di sekitar semua objek seperti yang ditunjukkan pada Gambar 12.25. Sinar yang mengenai kotak pembatas sebenarnya akan lebih mahal untuk dihitung dibandingkan dengan pencarian brute force, karena pengujian perpotongan dengan kotak tidak gratis. Namun, sinar yang ketinggalan kotak lebih murah daripada pencarian brute force. Kotak pembatas tersebut dapat dibuat hierarkis dengan mempartisi himpunan objek dalam kotak dan menempatkan kotak di sekitar setiap partisi seperti yang ditunjukkan pada Gambar 12.26. Struktur data untuk hierarki yang ditunjukkan pada Gambar 12.27 mungkin berupa pohon dengan kotak pembatas besar di akarnya dan dua kotak pembatas yang lebih kecil sebagai subpohon kiri dan kanan. Ini pada gilirannya akan menunjukkan setiap titik ke daftar tiga segitiga. Perpotongan sebuah sinar dengan pohon hard-coded tertentu ini akan menjadi:

Beberapa pengamatan yang berhubungan dengan algoritma ini adalah bahwa tidak ada urutan geometris antara dua subpohon, dan tidak ada sinar alasan yang mungkin tidak ada pada kedua subpohon. Memang, tidak ada alasan bahwa dua subpohon mungkin tidak tumpang tindih.

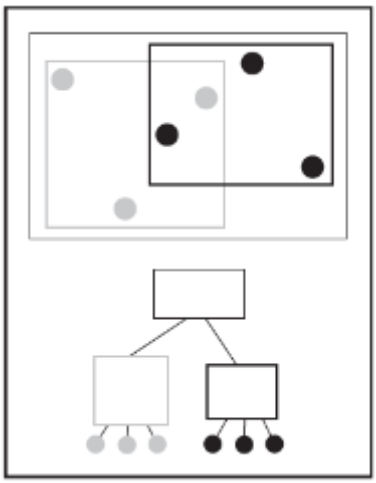


Gambar 12.25 Sinar 2D e + td diuji terhadap kotak pembatas 2D.



Gambar 12.26 Kotak pembatas dapat disarangkan dengan membuat kotak di sekitar himpunan bagian dari model.

if (ray hits root box) **then**
if (ray hits left subtree box) **then**
check three triangles for intersection
if (ray intersects right subtree box) **then**
check other three triangles for intersection
if (an intersections returned from each subtree) **then**
return the closest of the two hits
else if (a intersection is returned from exactly one subtree) **then**
return that intersection
else
return false
else
return false



Gambar 12.27 Kotak abu-abu adalah simpul pohon yang menunjuk ke tiga bola abu-abu, dan kotak hitam tebal menunjuk ke tiga bola hitam. Perhatikan bahwa tidak semua bola yang diapit oleh kotak dijamin ditunjuk oleh simpul pohon yang sesuai.

Poin kunci dari hierarki data tersebut adalah bahwa sebuah kotak dijamin untuk mengikat semua objek yang berada di bawahnya dalam hierarki, tetapi kotak tersebut tidak dijamin berisi semua objek yang tumpang tindih secara spasial, seperti yang ditunjukkan pada Gambar 12.27. Hal ini membuat pencarian geometris ini agak lebih rumit daripada pencarian biner tradisional pada data satu dimensi yang dipesan secara ketat. Pembaca mungkin memperhatikan bahwa beberapa kemungkinan optimalisasi muncul dengan sendirinya. Kami menunda optimalisasi sampai kami memiliki algoritma hierarki penuh.

Jika kita membatasi pohon menjadi biner dan mengharuskan setiap simpul di pohon memiliki kotak pembatas, maka kode traversal ini meluas secara alami. Selanjutnya, asumsikan bahwa semua node adalah salah satu daun dalam pohon dan berisi primitif, atau mereka berisi satu atau dua subpohon.

Kelas bvh-node harus bertipe surface, jadi harus mengimplementasikan surface::hit. Data yang dikandungnya harus sederhana:

The bvh-node class should be of type surface, so it should implement
surface::hit. The data it contains should be simple:
class bvh-node subclass of surface
virtual bool hit(ray **e** + *t***d**, real *t*0, real *t*1, hit-record rec)
virtual box bounding-box()
surface-pointer left

surface-pointer right
box bbox

Kode traversal kemudian dapat dipanggil secara rekursif dalam gaya berorientasi objek:

function bool bvh-node::hit(ray **a** + *t***b**, real *t*0, real *t*1,
hit-record rec)
if (bbox.hitbox(**a**+ *t***b**, *t*0, *t*1)) **then**
hit-record lrec, rrec
left-hit = (left _≠ NULL) and (left →hit(**a** + *t***b**, *t*0, *t*1, lrec))
right-hit = (right _≠ NULL) and (right→hit(**a**+*t***b**, *t*0, *t*1, rrec))
if (left-hit and right-hit) **then**
if (lrec.t < rrec.t) **then**
rec = lrec
else
rec = rrec
return true
else if (left-hit) **then**
rec = lrec
return true
else if (right-hit) **then**
rec = rrec
return true
else
return false
else
return false

Perhatikan bahwa karena *left* dan *right* menunjuk ke permukaan daripada *bvh-node* secara khusus, kita dapat membiarkan fungsi virtual menangani pembedaan antara node internal dan leaf; fungsi hit yang sesuai akan dipanggil. Perhatikan bahwa jika pohon dibangun dengan benar, kita dapat menghilangkan tanda centang untuk dibiarkan menjadi NULL. Jika kita ingin menghilangkan tanda centang untuk kanan menjadi NULL, kita dapat mengganti pointer kanan NULL dengan pointer yang berlebihan ke kiri. Ini akan mengakhiri pemeriksaan kiri dua kali, tetapi akan menghilangkan pemeriksaan di seluruh pohon. Apakah itu layak akan tergantung pada detail konstruksi pohon.

Ada banyak cara untuk membangun pohon untuk hierarki volume pembatas. Lebih mudah untuk membuat pohon biner, kira-kira seimbang, dan agar kotak-kotak subpohon bersaudara tidak terlalu tumpang tindih. Sebuah heuristik untuk mencapai hal ini adalah untuk mengurutkan permukaan sepanjang sumbu sebelum membaginya menjadi dua sublist. Jika sumbu didefinisikan oleh bilangan bulat dengan x =0 , y =1, dan z =2 kita memiliki:

function bvh-node::create(object-arrayA, int AXIS)
N = A.length
if (N= 1) **then**
left = A[0]
right = NULL
bbox = bounding-box(A[0])
else if (N= 2) **then**
left-node = A[0]
right-node = A[1]
bbox = combine(bounding-box(A[0]), bounding-box(A[1]))

else
sort A by the object center along AXIS
left= new bvh-node(A[0..N/2− 1], (AXIS +1) mod 3)
right = new bvh-node(A[N/2..N−1], (AXIS +1) mod 3)
bbox = combine(left→bbox, right→bbox)

Kualitas pohon dapat ditingkatkan dengan memilih AXIS dengan hati-hati setiap saat. Salah satu cara untuk melakukannya adalah dengan memilih sumbu sedemikian rupa sehingga jumlah volume kotak pembatas dari dua subpohon diminimalkan. Perubahan ini dibandingkan dengan memutar melalui sumbu akan membuat sedikit perbedaan untuk scene yang terdiri dari objek kecil yang didistribusikan secara isotopik, tetapi dapat membantu secara signifikan dalam scene yang berperilaku kurang baik. Kode ini juga dapat dibuat lebih efisien dengan hanya melakukan partisi daripada pengurutan penuh.

Cara lain, dan mungkin lebih baik, untuk membangun pohon adalah dengan membuat subpohon berisi jumlah ruang yang sama daripada jumlah objek yang sama. Untuk melakukan ini, kami mempartisi daftar berdasarkan ruang:

function bvh-node::create(object-arrayA, int AXIS)
N = A.length
if (N = 1) **then**
left = A[0]
right = NULL
bbox = bounding-box(A[0])
else if (N = 2) **then**
left = A[0]
right = A[1]
bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
else
find the midpoint*m* of the bounding box of A along AXIS
partition A into lists with lengths *k* and (*N* −*k*) surrounding*m*
left = new bvh-node(A[0..*k*], (AXIS +1) mod 3)
right = new bvh-node(A[*k*+ 1..*N* − 1], (AXIS +1) mod 3)
bbox = combine(left→bbox, right→bbox)

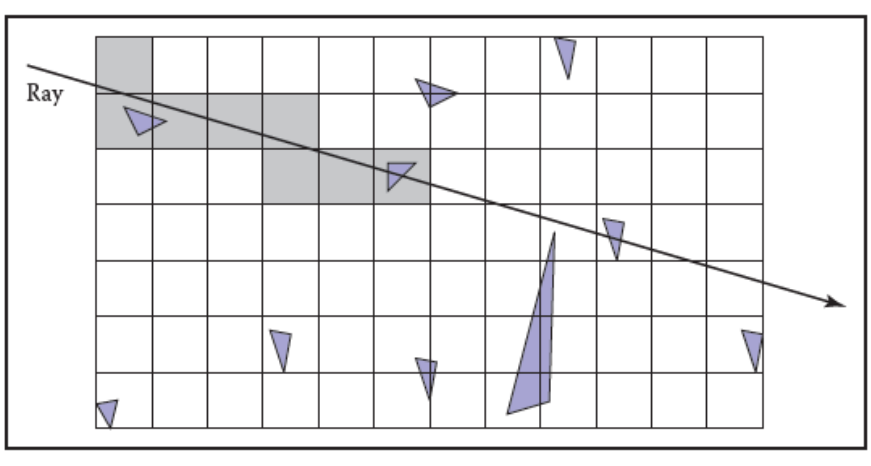
Meskipun hal ini menghasilkan pohon yang tidak seimbang, ini memungkinkan penjelajahan ruang kosong dengan mudah dan lebih murah untuk dibangun karena mempartisi lebih murah daripada menyortir.

Subdivisi Tata Ruang Seragam

Strategi lain untuk mengurangi uji simpang adalah membagi ruang. Ini pada dasarnya berbeda dari membagi objek seperti yang dilakukan dengan volume pembatas hierarkis:

- Volume terikat hierarkis, setiap objek termasuk ke dalam satu dari dua simpul bersaudara, sedangkan sebuah titik dalam ruang mungkin berada di dalam kedua simpul bersaudara.
- Dalam subdivisi spasial, setiap titik dalam ruang milik tepat satu node, sedangkan objek mungkin milik banyak node.

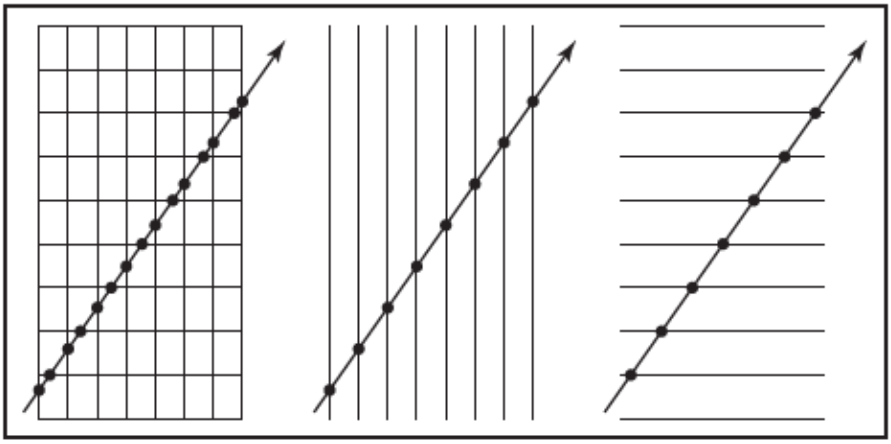
Dalam subdivisi spasial yang seragam, scene dipartisi ke dalam kotak yang disejajarkan dengan sumbu. Semua kotak ini berukuran sama, meskipun belum tentu kubus. Sinar melintasi kotak-kotak ini seperti yang ditunjukkan pada Gambar 12.28. Ketika sebuah objek terkena, traversal berakhir.



Gambar 12.28 Subdivisi spasial yang tidak seragam, dilacak ke depan melalui sel sampai objek di salah satu sel tersebut terkena. Dalam contoh ini, hanya objek dalam sel yang diarsir yang diperiksa.

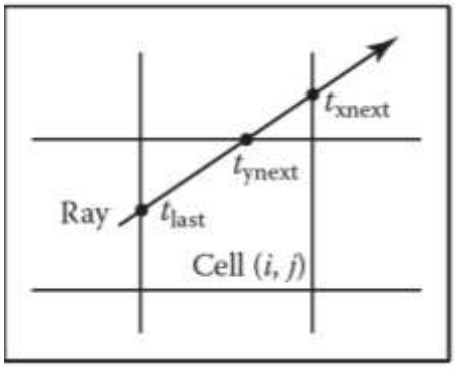
Grid itu sendiri harus menjadi subclass dari permukaan dan harus diimplementasikan sebagai array 3D dari pointer ke permukaan. Untuk sel kosong, pointer ini adalah NULL. Untuk sel dengan satu objek, penunjuk menunjuk ke objek itu. Untuk sel dengan lebih dari satu objek, penunjuk bisa mengarah ke daftar, kisi lain, atau struktur data lain, seperti hierarki volume pembatas.

Traversal ini dilakukan secara bertahap. Keteraturan berasal dari cara sinar menumbuk setiap set bidang sejajar, seperti yang ditunjukkan pada Gambar 12.29. Untuk melihat bagaimana traversal ini bekerja, pertama-tama pertimbangkan kasus 2D di mana arah sinar memiliki komponen x dan y positif dan dimulai di luar grid. Asumsikan grid dibatasi oleh titik (x_{min}, y_{min}) dan (x_{max}, y_{max}). Grid memiliki $n_x \times n_y$ sel.

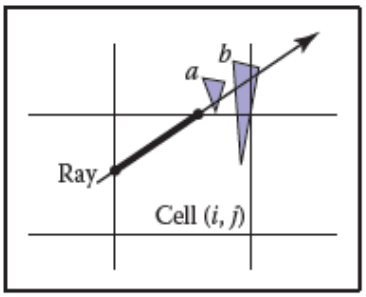


Gambar 12.29 Meskipun pola pukulan sel tampak tidak beraturan (kiri), pukulan pada set bidang paralel sangat merata.

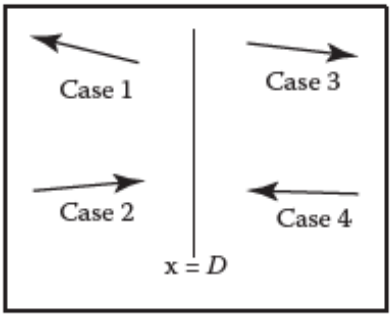
Urutan bisnis pertama kita adalah menemukan indeks (i,j) dari sel pertama yang terkena sinar e + td. Kemudian, kita perlu melintasi sel dalam urutan yang sesuai. Bagian kunci dari algoritma ini adalah menemukan sel awal (i,j) dan memutuskan apakah akan menaikkan i atau j (Gambar 12.30). Perhatikan bahwa ketika kita memeriksa persimpangan dengan objek di dalam sel, batasi rentangnya hingga berada di dalam sel (Gambar 12.31). Kebanyakan implementasi membuat3Darrayoftype"pointertosurface."Untuk meningkatkan lokalitas traversal, array dapat diberi tiling seperti yang dibahas dalam Bagian 12.5



Gambar 12.30 Untuk memutuskan apakah kita maju ke kanan atau ke atas, kita melacak perpotongan dengan batas vertikal dan horizontal sel berikutnya.



Gambar 12.31 Hanya hit di dalam sel yang harus dilaporkan. Jika tidak, kasus di atas akan menyebabkan ustoreport mengenai objek b daripada objek a.



Gambar 12.32 Empat kasus bagaimana sinar berhubungan dengan bidang potong BSP x=D.

Partisi Ruang Biner Sejajar Sumbu

Kita juga dapat mempartisi ruang dalam struktur data hierarkis seperti pohon partisi ruang biner (pohon BSP). Ini mirip dengan pohon BSP yang digunakan untuk penyortiran visibilitas di Bagian 12.4, tetapi yang paling umum adalah menggunakan bidang potong berjajar sumbu, bukan berjajar poligon, untuk perpotongan sinar.

Sebuah node dalam struktur ini berisi satu bidang pemotongan dan subpohon kiri dan kanan. Setiap subpohon berisi semua objek di satu sisi bidang pemotongan. Objek yang melewati bidang disimpan di dalam kedua subpohon. Jika kita asumsikan bidang potong sejajar dengan bidang y_z pada x = D, maka kelas simpulnya adalah:

class bsp-node subclass of surface
virtual bool hit(ray **e** + *t***d**, real *t*0, real *t*1, hit-record rec)
virtual box bounding-box()
surface-pointer left
surface-pointer right
real *D*

Kami menggeneralisasi ini ke y dan z memotong pesawat nanti. Kode persimpangan kemudian dapat dipanggil secara rekursif dalam gaya berorientasi objek. Kode mempertimbangkan empat kasus yang ditunjukkan pada Gambar 12.32. Untuk tujuan kita, asal sinar ini adalah titik pada parameter t0:

$$\mathbf{p} = \mathbf{a} + t0\mathbf{b}.$$

Keempat kasus tersebut adalah:

1. Sinar hanya berinteraksi dengan subpohon kiri, dan kita tidak perlu mengujinya untuk perpotongan dengan bidang potong. Itu terjadi untuk xp <Dan xb < 0.
2. Sinar diuji terhadap subpohon kiri, dan jika tidak ada hit, maka diuji terhadap subpohon kanan. Kita perlu menemukan parameterparameteratx = D, sehingga kita dapat memastikan bahwa kita hanya menguji perpotongan di dalam subpohon. Kasus ini terjadi untuk $x_p < D$ dan $x_b > 0$.
3. Kasus ini analog dengan kasus 1 dan terjadi untuk $x_p > D$ dan $x_b > 0$.
4. Kasus ini analog dengan kasus 2 dan terjadi untuk $x_p > D$ dan $x_b < 0$.

Kode traversal yang dihasilkan menangani kasus-kasus ini secara berurutan adalah:

```
function bool bsp-node::hit(ray a + tb, real t0, real t1,
hit-record rec)
xp = xa + t0xb
if (xp < D) then
if (xb < 0) then
return (left _= NULL) and (left→hit(a + tb, t0, t1, rec))
t = (D − xa)/xb
if (t > t1) then
return (left _= NULL) and (left→hit(a + tb, t0, t1, rec))
if (left _= NULL) and (left→hit(a + tb, t0, t, rec)) then
return true
return (right _= NULL) and (right→hit(a + tb, t, t1, rec))
else
analogous code for cases 3 and 4
```

kode analog untuk kasus 3 dan 4

Ini adalah kode yang sangat bersih. Namun, untuk memulai, kita perlu mencapai beberapa objek root yang berisi kotak pembatas sehingga kita dapat menginisialisasi traversal, t0 dan t1. Masalah yang harus kita atasi adalah bahwa bidang pemotongan mungkin berada di sepanjang sumbu apa pun. Kita dapat menambahkan sumbu indeks integer ke kelas bsp-node. Jika kita mengizinkan operator pengindeksan untuk poin, ini akan menghasilkan beberapa modifikasi sederhana pada kode di atas, misalnya,

$$xp = xa + t0xb$$

akan menjadi

$$up = a[\text{axis}] + t0b[\text{axis}]$$

yang akan menghasilkan beberapa pengindeksan array tambahan, tetapi tidak akan menghasilkan lebih banyak cabang.

Sementara pemrosesan satu lebsp-node lebih cepat daripada pemrosesan abvh-node, fakta bahwa satu permukaan mungkin ada di lebih dari satu subtree berarti ada lebih banyak node dan, berpotensi, penggunaan memori yang lebih tinggi. Seberapa "baik" pohon-pohon itu dibangun menentukan mana yang lebih cepat. Membangun pohon mirip dengan membangun pohon BVH. Kita dapat memilih sumbu untuk dibagi dalam satu siklus, dan kita dapat membagi menjadi dua setiap kali, atau kita dapat mencoba lebih canggih dalam cara kita membagi.

12.4 POHON BSP UNTUK VISIBILITAS

Masalah geometris lain di mana struktur data spasial dapat digunakan adalah menentukan urutan visibilitas objek dalam scene dengan sudut pandang yang berubah. Jika kita membuat banyak gambar scene tetap yang tersusun dari poligon planar, dari sudut pandang yang berbeda—seperti yang sering terjadi pada aplikasi seperti game—kita dapat menggunakan skema partisi ruang biner yang terkait erat dengan metode perpotongan sinar yang dibahas di bagian sebelumnya. Perbedaannya adalah bahwa untuk pengurutan visibilitas kami menggunakan bidang pemisah yang tidak sejajar sumbu, sehingga bidang dapat dibuat berhimpitan dengan poligon. Ini mengarah pada algoritma elegan yang dikenal sebagai algoritma pohon BSP untuk mengurutkan permukaan dari depan ke belakang. Aspek kunci dari pohon BSP adalah bahwa ia menggunakan preprocess untuk membuat struktur data yang berguna untuk sudut pandang apapun. Jadi, saat sudut pandang berubah, struktur data yang sama digunakan tanpa perubahan.

Ikhtisar Algoritma Pohon BSP

Algoritma pohon BSP adalah contoh dari algoritma pelukis. Algoritme pelukis menggambar setiap objek dari belakang ke depan, dengan setiap poligon baru berpotensi menggambar lebih banyak poligon sebelumnya, seperti yang ditunjukkan pada Gambar 12.33. Itu dapat diimplementasikan sebagai berikut:

```
sort objects back to front relative to viewpoint
for each object do
draw object on screen
```

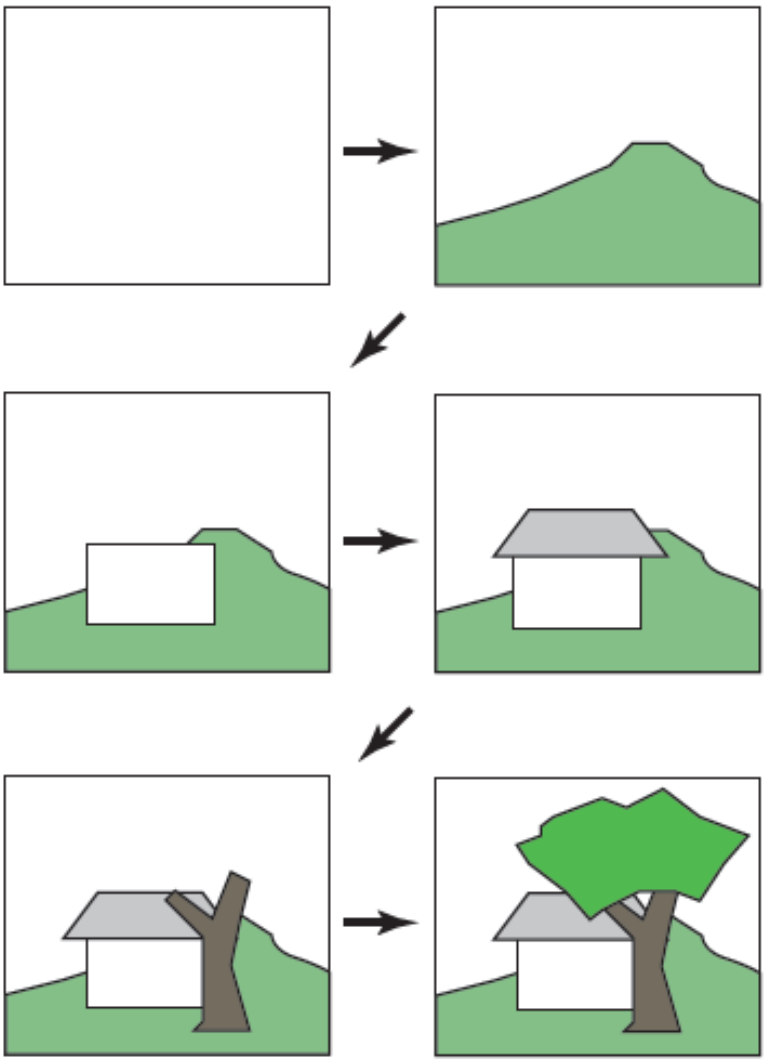
Masalah dengan langkah pertama (pengurutan) adalah bahwa urutan relatif beberapa objek tidak selalu terdefinisi dengan baik, bahkan jika urutan setiap pasangan objek adalah. Masalah ini diilustrasikan pada Gambar 12.34 di mana tiga segitiga membentuk siklus.

Algoritma pohon BSP bekerja pada setiap scene yang terdiri dari poligon di mana tidak ada poligon yang melintasi bidang yang ditentukan oleh poligon lainnya. Pembatasan ini kemudian dilonggarkan dengan langkah preprocessing. Untuk sisa diskusi ini, segitiga dianggap sebagai satu-satunya primitif, tetapi gagasan meluas ke poligon sewenang-wenang.

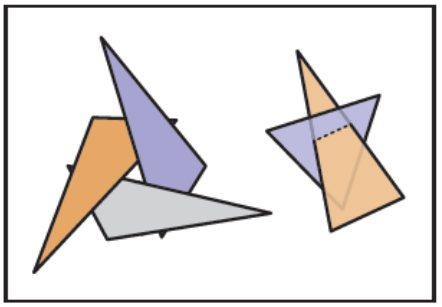
Ide dasar pohon BSP dapat digambarkan dengan dua segitiga, T_1 dan T_2. Ingat pertama (lihat Bagian 2.5.3) persamaan bidang implisit dari bidang yang mengandung T_1 : $f_1(p) = 0$. Properti kunci dari bidang implisit yang ingin kita manfaatkan adalah bahwa untuk semua titik p+ pada satu sisi bidang, $f_1(p^+) > 0$; dan untuk semua titik p− pada sisi lain bidang, $f_1(p^-) < 0$. Dengan menggunakan sifat ini, kita dapat mengetahui di sisi mana bidang T_2 terletak. Sekali lagi, ini mengasumsikan ketiga simpul dari T_2 berada pada sisi yang sama pada bidang. Untuk diskusi, asumsikan bahwa T_2 berada di $f_1(p) < 0$ sisi bidang. Kemudian, kita dapat menggambar T_1 dan T_2 dalam urutan yang benar untuk setiap titik mata:

```
if (f1(e) < 0) then
draw T1
draw T2
else
draw T2
draw T1
```

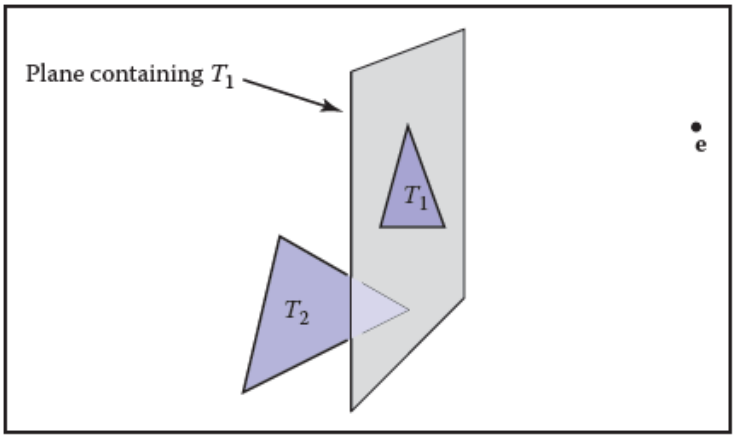
Alasan cara ini berhasil adalah jika T_2 dan e berada pada sisi yang sama dari bidang yang memuat T_1, tidak mungkin T_2 terhalangi sebagian atau seluruhnya oleh T_1 seperti yang terlihat dari e, jadi aman untuk menggambar T_1 terlebih dahulu. Jika e dan T_2 berada pada sisi berlawanan dari bidang yang memuat T_1, maka T_2 tidak dapat sepenuhnya atau sebagian memblokir T_1, dan urutan gambar yang berlawanan aman (Gambar 12.35).



Gambar 12.33 Algoritme pelukis dimulai dengan gambar kosong dan kemudian menggambar scene satu objek pada satu waktu dari belakang ke depan, menggambar berlebihan apa pun yang sudah ada. Ini secara otomatis menghilangkan permukaan tersembunyi.



Gambar 12.34 Siklus terjadi jika pemesanan back-to-front global tidak memungkinkan untuk posisi mata tertentu.



Gambar 12.35 Ketika e dan T_2 berada pada sisi berlawanan dari bidang yang memuat T_1, maka aman untuk menggambar T_2 terlebih dahulu dan T_1 sekon. Jika e dan T_2 berada pada sisi bidang yang sama, maka T_1 harus digambar sebelum T_2. Ini adalah ide inti dari algoritma pohon BSP.

Pengamatan ini dapat digeneralisasikan ke banyak objek asalkan tidak satupun dari mereka menjangkau bidang yang ditentukan oleh T_1. Jika kita menggunakan struktur data pohon biner dengan T_1 sebagai root, cabang negatif dari pohon berisi semua segitiga yang simpulnya memiliki $f_i(p) < 0$, dan cabang positif dari pohon berisi semua segitiga yang simpulnya memiliki $f_i(p) > 0$. Kita dapat menggambar dengan urutan yang benar sebagai berikut:

```
function draw(bsptree tree, point e)
if (tree.empty) then
return
if (ftree.root(e) < 0) then
draw(tree.plus, e)
rasterize tree.triangle
draw(tree.minus, e)
else
draw(tree.minus, e)
rasterize tree.triangle
draw(tree.plus, e)
```

Hal yang menyenangkan tentang kode itu adalah kode itu akan berfungsi untuk sudut pandang apa pun e, sehingga pohon dapat dihitung sebelumnya. Perhatikan bahwa, jika setiap subpohon itu sendiri adalah sebuah pohon, di mana akar segitiga membagi segitiga lainnya menjadi dua kelompok relatif terhadap bidang yang memuatnya, kode akan bekerja apa adanya. Ini dapat dibuat sedikit lebih efisien dengan mengakhiri panggilan rekursif satu tingkat lebih tinggi, tetapi kodenya akan tetap sederhana. Sebuah pohon yang menggambarkan kode ini ditunjukkan pada Gambar 12.36. Sebagaimana dibahas dalam Bagian 2.5.5, persamaan implisit untuk titik p pada bidang yang memuat tiga titik non-kolinear a, b, danc adalah

$$f(\mathbf{p}) = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) \cdot (\mathbf{p} - \mathbf{a}) = 0.$$

Lebih cepat untuk menyimpan (A,B,C,D) dari persamaan implisit dari bentuk

$$f(x, y, z) = Ax + By + Cz + D = 0.$$

Persamaan (12.1) dan (12.2) adalah ekuivalen, jelas terlihat bahwa gradien persamaan implisit adalah normal segitiga. Gradien Persamaan (12.2) adalah n =(A,B,C) yang merupakan vektor normal

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}).$$

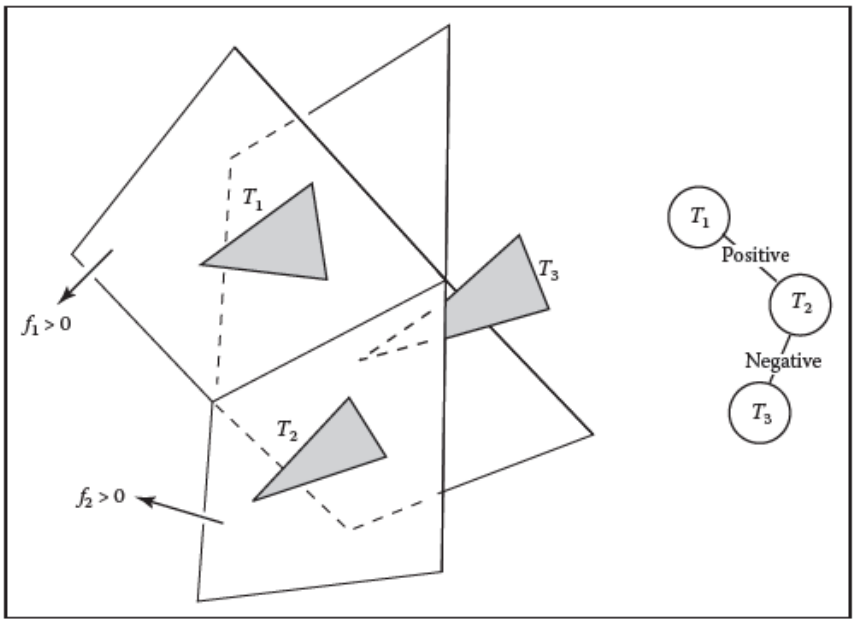
Kita dapat menyelesaikan D dengan memasukkan sembarang titik pada bidang, misal, a:

$$D = -Axa - Bya - Cza$$
$$= -\mathbf{n} \cdot \mathbf{a}.$$

Ini menyarankan bentuk:

$$f(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{a}$$
$$= \mathbf{n} \cdot (\mathbf{p} - \mathbf{a})$$
$$= 0,$$

yang merupakan persamaan (12.1) setelah Anda mengingatnya, kemudian dihitung dengan menggunakan perkalian silang. Bentuk persamaan bidang mana yang Anda gunakan dan apakah Anda hanya menyimpan simpul, simpul, orn, D, dan simpul, mungkin adalah soal selera—pertukaran penyimpanan waktu klasik yang akan diselesaikan paling baik dengan pembuatan profil. Untuk debugging, menggunakan Persamaan (12.1) mungkin yang terbaik.



Gambar 12.36 Tiga segitiga dan pohon BSP yang valid untuk mereka. "Positif" dan "negatif" masing-masing dikodekan oleh posisi subpohon kanan dan kiri.

Satu-satunya masalah yang mencegah kode di atas bekerja secara umum adalah bahwa seseorang tidak dapat menjamin bahwa sebuah segitiga dapat diklasifikasikan secara unik pada satu sisi bidang atau sisi lainnya. Itu dapat memiliki dua simpul di satu sisi pesawat dan yang ketiga di sisi lain. Atau dapat memiliki simpul di pesawat. Ini ditangani dengan membagi segitiga menjadi segitiga yang lebih kecil menggunakan pesawat untuk "memotong" mereka.

Membangun Pohon

Jika tidak ada segitiga dalam kumpulan data yang melintasi bidang satu sama lain, sehingga semua segitiga berada di satu sisi dari semua segitiga lainnya, pohon BSP yang dapat dilalui menggunakan kode di atas dapat dibangun menggunakan algoritma berikut:

```
tree-root = node(T1)
for i ∈{2, . . .,N} do
tree-root.add(Ti)
function add ( triangle T )
if (f(a) < 0 and f(b) < 0 and f(c) < 0) then
```

if (negative subtree is empty) **then**
negative-subtree = node(*T*)
else
negative-subtree.add (*T*)
else if (*f*(**a**) > 0 and *f*(**b**) > 0 and *f*(**c**) > 0) **then**
if positive subtree is empty **then**
positive-subtree = node(*T*)
else
positive-subtree.add (*T*)
else

kami berasumsi bahwa kasus ini tidak mungkin

Satu-satunya hal yang perlu kita perbaiki adalah kasus di mana segitiga memotong bidang pembagi, seperti yang ditunjukkan pada Gambar 12.37. Asumsikan, untuk sederhananya, bahwa segitiga memiliki simpul a dan b di satu sisi bidang, dan simpul di sisi lain. Dalam hal ini, kita dapat menemukan titik potong A dan B dan memotong segitiga menjadi tiga segitiga baru dengan simpul

*T*1 = (**a**, **b**,**A**),
*T*2 = (**b**,**B**,**A**),
*T*3 = (**A**,**B**, **c**),

seperti yang ditunjukkan pada Gambar 12.38. Urutan titik sudut ini penting agar arah garis normal tetap sama dengan arah segitiga semula. Jika kita berasumsi bahwa f(c) < 0, kode berikut dapat menambahkan ketiga segitiga ini ke pohon dengan asumsi subpohon positif dan negatif tidak kosong:

positive-subtree = node (*T*1)
positive-subtree = node (*T*2)
negative-subtree = node (*T*3)

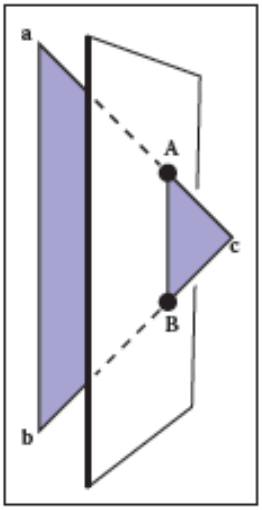
Masalah presisi yang akan mengganggu implementasi anaif terjadi ketika verteks sangat dekat dengan bidang pemisah. Misalnya, jika kita memiliki dua simpul di satu sisi bidang pemisah dan simpul lainnya hanya berjarak sangat kecil di sisi lain, kita akan membuat segitiga baru yang hampir sama dengan yang lama, segitiga yang merupakan sliver, dan segitiga dengan ukuran hampir nol. Akan lebih baik untuk mendeteksi ini sebagai kasus khusus dan tidak dibagi menjadi tiga segitiga baru. Orang mungkin berharap kasus ini jarang terjadi, tetapi karena banyak model memiliki bidang dan segitiga yang terselubung dengan simpul yang sama, hal itu sering terjadi, dan karenanya harus ditangani dengan hati-hati. Beberapa manipulasi sederhana yang mencapai ini adalah:

function add(triangle *T*)
fa = *f*(**a**)
fb = *f*(**b**)
fc = *f*(**c**)
if (*abs*(*fa*) <) **then**
fa = 0
if (*abs*(*fb*) <) **then**
fb = 0
if (*abs*(*fc*) <) **then**
fc = 0
if (*fa* ≤ 0 and fb ≤ 0 and fc ≤ 0) **then**
if (negative subtree is empty) **then**
negative-subtree = node(*T*)

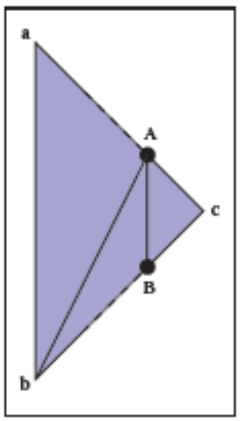
else
negative-subtree.add(*T*)
else if (*fa* ≥ 0 and fb ≥ 0 and fc ≥ 0) **then**
if (positive subtree is empty) **then**
positive-subtree = node(*T*)
else
positive-subtree.add(*T*)
else
potong segitiga menjadi tiga segitiga dan tambahkan disetiap sisi.

Ini mengambil setiap simpul yang nilai f-nya berada di dalam bidang ∈ dan menghitungnya sebagai positif atau negatif. Konstanta adalah real positif kecil yang dipilih oleh pengguna. Teknik di atas adalah contoh langka di mana pengujian untuk titik-mengambang berguna dan berhasil karena nilai nol ditetapkan daripada dihitung. Membandingkan kualitas depan dengan nilai titik mengambang yang dihitung hampir tidak pernah disarankan, tetapi kami tidak melakukan itu.

if (*fa* ∗ *fc* ≥ 0) **then**
swap(*fb, fc*)
swap(**b**, **c**)
swap(*fa, fb*)
swap(**a**, **b**)
else if (*fb* ∗ *fc* ≥ 0) **then**
swap(*fa, fc*)
swap(**a**, **c**)
swap(*fa, fb*)
swap(**a**, **b**)
compute **A**
compute **B**
*T*1 = (**a**, **b**,**A**)
*T*2 = (**b**,**B**,**A**)
*T*3 = (**A**,**B**, **c**)
if (*fc* ≥ 0) **then**
negative-subtree.add(*T*1)
negative-subtree.add(*T*2)
positive-subtree.add(*T*3)
else
positive-subtree.add(*T*1)
positive-subtree.add(*T*2)
negative-subtree.add(*T*3)



Gambar 12.37 Ketika sebuah segitiga merentang pada sebuah bidang, akan ada satu simpul di satu sisi dan dua di sisi lainnya.



Gambar 12.38 Ketika sebuah segitiga dipotong, kami memecahnya menjadi tiga segitiga, tidak ada yang merentang pada bidang pemotongan.

Memotong Segitiga

Mengisi detail dari kasus terakhir "potong segitiga menjadi tiga segitiga dan tambahkan ke setiap sisi" sangat mudah, tetapi membosankan. Kita harus memanfaatkan konstruksi pohon BSP sebagai praproses di mana efisiensi tertinggi bukanlah kuncinya. Sebagai gantinya, kita harus mencoba untuk memiliki kode kompak yang bersih. Trik yang bagus adalah memaksa banyak kasus menjadi satu dengan memastikan bahwa c berada di satu sisi bidang dan dua simpul lainnya berada di sisi lain. Ini mudah dilakukan dengan swap. Mengisi rincian dalam pernyataan akhir (dengan asumsi subpohon tidak kosong untuk kesederhanaan) memberikan:

if (*fa* * *fc* ≥ 0) **then**
swap(*fb*, *fc*)
swap(**b**, **c**)
swap(*fa*, *fb*)
swap(**a**, **b**)
else if (*fb* * *fc* ≥ 0) **then**
swap(*fa*, *fc*)
swap(**a**, **c**)
swap(*fa*, *fb*)

```
swap(a, b)
compute A
compute B
T1 = (a, b,A)
T2 = (b,B,A)
T3 = (A,B, c)
if (fc ≥ 0) then
        negative-subtree.add(T1)
        negative-subtree.add(T2)
        positive-subtree.add(T3)
else
        positive-subtree.add(T1)
        positive-subtree.add(T2)
        negative-subtree.add(T3)
```

Kode ini mengambil keuntungan dari fakta bahwa produk dari sebuah dan positif kosong jika mereka memiliki tanda yang sama—dengan demikian, pernyataan if pertama. Jika simpul ditukar, kita harus melakukan dua pertukaran untuk menjaga agar simpul terurut berlawanan arah jarum jam. Perhatikan bahwa tepat satu simpul mungkin terletak tepat pada bidang, dalam hal ini kode di atas akan bekerja, tetapi salah satu segitiga yang dihasilkan akan memiliki luas nol. Ini dapat ditangani dengan mengabaikan kemungkinan, yang tidak terlalu berisiko, karena kode rasterisasi harus menangani segitiga area-nol di ruang layar (yaitu, segitiga edge-on). Anda juga dapat menambahkan tanda centang yang tidak menambahkan segitiga area-nol ke pohon. Akhirnya, Anda dapat menempatkan dalam kasus khusus ketika tepat satu dari a, fb, dan fcis nol yang memotong segitiga menjadi dua segitiga.

Untuk menghitung A dan B, segmen alinea dan persimpangan bidang implisit diperlukan. Misalnya, garis parametrik yang menghubungkan a dan c adalah

$$\mathbf{p}(t) = \mathbf{a} + t(\mathbf{c} - \mathbf{a}).$$

Titik perpotongan dengan bidang n·p + D = 0 ditemukan dengan memasukkan p(t) ke dalam persamaan bidang:

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{c} - \mathbf{a})) + D = 0,$$

dan penyelesaian untuk t:

$$t = \frac{n\,.\,a.\,+D}{n\,.\,(c-a)}$$

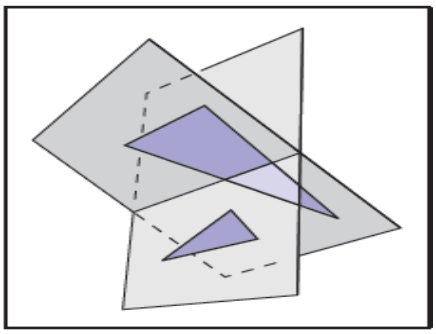
Memanggil solusi ini t_A, kita dapat menulis ekspresi untuk A:

$$\mathbf{A} = \mathbf{a} + tA(\mathbf{c} - \mathbf{a}).$$

Perhitungan serupa akan memberikan B.

Mengoptimalkan Pohon

Efisiensi penciptaan pohon tidak terlalu mengkhawatirkan dibandingkan dengan lintasan pohon karena ini adalah sebuah proses awal. Penjelajahan pohon BSP membutuhkan waktu yang sebanding dengan jumlah node dalam pohon. (Seberapa seimbang pohon itu tidak masalah.) Akan ada satu simpul untuk setiap segitiga, termasuk segitiga yang dibuat sebagai hasil pemisahan. Jumlah ini dapat bergantung pada urutan segitiga yang ditambahkan ke pohon. Sebagai contoh, pada Gambar 12.39, jika T_1 adalah root, akan ada dua node di pohon, tetapi jika T_2 adalah root, akan ada lebih banyak node, karena T_1 akan dipecah.



Gambar 12.39 Menggunakan T_1 sebagai akar pohon BSP akan menghasilkan pohon dengan dua simpul. Menggunakan T_2 sebagai akar akan membutuhkan pemotongan dan dengan demikian membuat pohon yang lebih besar.

Sulit untuk menemukan urutan segitiga "terbaik" untuk ditambahkan ke pohon. Untuk N segitiga, ada N! pemesanan yang mungkin. Jadi mencoba semua pemesanan biasanya tidak memungkinkan. Sebagai alternatif, beberapa urutan yang telah ditentukan sebelumnya dapat dicoba dari kumpulan permutasi acak, dan yang terbaik dapat disimpan untuk pohon terakhir.

Algoritma pemisahan yang dijelaskan di atas membagi satu segitiga menjadi tiga segitiga. Akan lebih efisien untuk membagi segitiga menjadi segitiga dan segi empat cembung. Ini mungkin tidak sepadan jika semua model input hanya memiliki segitiga, tetapi akan mudah untuk mendukung implementasi yang mengakomodasi poligon arbitrer.

12.5 TILING MULTIDIMENSI ARRAY

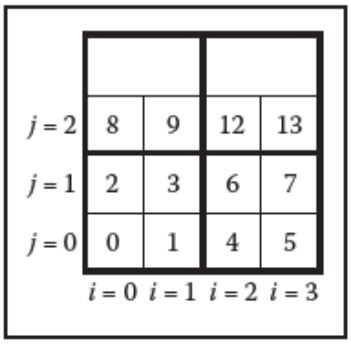
Memanfaatkan hierarki memori secara efektif adalah tugas penting dalam merancang algoritme untuk arsitektur modern. Memastikan bahwa array multidimensi memiliki data dalam pengaturan yang "bagus" dilakukan dengan memasang tiling, terkadang juga disebut bricking. Array 2D tradisional disimpan sebagai array 1D bersama dengan mekanisme pengindeksan; misalnya, larik N_x oleh Ny disimpan dalam larik 1D dengan panjang N_xN_y dan 2D index(x, y) (yang berjalan dari (0, 0) ke (N_x – 1, N_y − 1)) dipetakan ke indeks 1D (berjalan dari 0 hingga N_xN_y-1) menggunakan rumus

$$\text{index} = x + Nxy.$$

Contoh bagaimana memori itu diletakkan ditunjukkan pada Gambar 12.40. Masalah dengan tata letak ini adalah bahwa meskipun dua elemen array yang berdekatan yang berada di baris yang sama bersebelahan di memori, dua elemen yang berdekatan di kolom yang sama akan dipisahkan oleh elemen N_x di memori. Ini dapat menyebabkan lokalitas memori yang buruk untuk N_x besar. Solusi standar untuk ini adalah dengan menggunakan tiling untuk membuat lokalitas memori untuk baris dan kolom lebih setara. Sebuah contoh ditunjukkan pada Gambar 12.41 di mana tiling 2 x 2 digunakan. Rincian pengindeksan array seperti itu dibahas di bagian berikutnya. Contoh yang lebih rumit, dengan dua tingkat tiling pada larik 3D, dibahas setelah itu.

$j = 2$	8	9	10	11
$j = 1$	4	5	6	7
$j = 0$	0	1	2	3
	$i = 0$	$i = 1$	$i = 2$	$i = 3$

Gambar 12.40 Tata letak memori untuk larik 2D dengan N_x = 4 dan N_y = 3.



Gambar 12.41 Tata letak memori untuk 2Darray dengan N_x = 4 dan Ny = 3 dan 2 × 2 tiling. Perhatikan bahwa padding di bagian atas larik diperlukan karena N_y bukan kelipatan dari tiling ukuran dua.

Pertanyaan kuncinya adalah ukuran tiling yang akan dibuat. Dalam praktiknya, mereka harus serupa dengan ukuran unit memori pada mesin. Sebagai contoh, jika kita menggunakan nilai data 16-bit(2-byte) pada mesin dengan baris cache 128-byte, tiling 8x8 cocok persis di baris cache. Namun, dengan menggunakan nomor titik-mengambang 32-bit, yang memuat 32 elemen ke saluran cache, tiling 5 × 5 agak terlalu kecil dan tiling 6 × 6 agak terlalu besar. Karena ada juga unit memori berukuran lebih kasar seperti halaman, hierarki dengan logika serupa dapat berguna.

Tiling Satu Tiling untuk Array 2D

Jika kita asumsikan larik N_x×N_y didekomposisi menjadi persegi×n tiling(Gambar 12.42), maka jumlah tiling yang dibutuhkan adalah

$$Bx = Nx/n,$$
$$By = Ny/n.$$

Di sini, kita asumsikan bahwa n membagi N_x dan N_y dengan tepat. Jika ini tidak benar, array harus diisi. Misalnya, jika N_x = 15 dan n =4, maka N_x harus diubah menjadi 16. Untuk menyusun rumus pengindeksan larik seperti itu, pertama-tama kita cari indeks tiling (b_x, b_y) yang memberikan baris/kolom untuk tiling (tiling itu sendiri membentuk larik 2D):

$$bx = x \div n,$$
$$by = y \div n,$$

di mana÷adalah pembagian bilangan bulat, misalnya, 12 ÷ 5 = 2. Jika kita mengurutkan tiling di sepanjang baris seperti yang ditunjukkan pada Gambar 12.40, maka indeks elemen pertama tiling (b_x,b_y) adalah

$$\text{index} = n2(Bxby + bx).$$

Memori di tiling itu diatur seperti array 2D tradisional seperti yang ditunjukkan pada Gambar 12.41. Offset parsial (x' ,y') di dalam petak adalah

$$X' = x \bmod n,$$
$$Y' = y \bmod n,$$

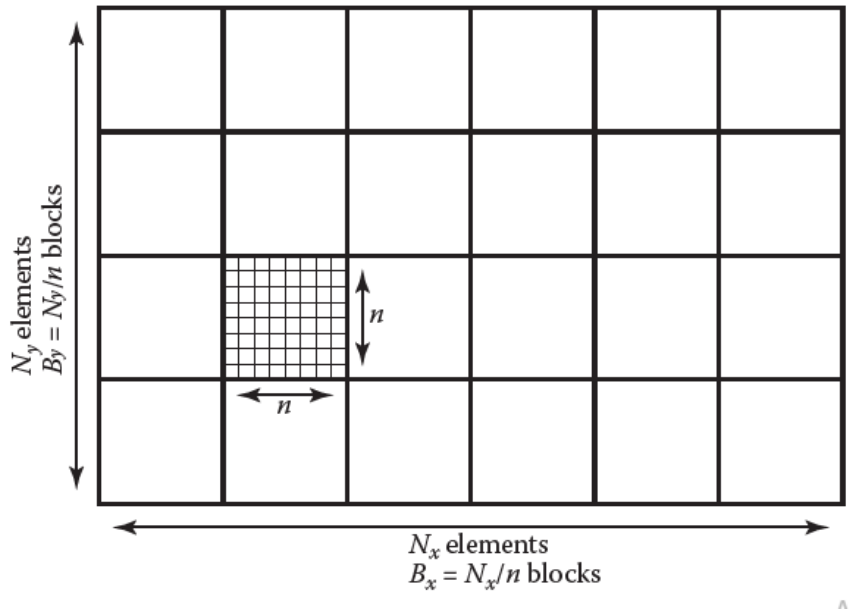
di mana mod adalah operator sisa, mis., 12 mod 5 = 2. Oleh karena itu, offset di dalam petak adalah

$$\text{offset} = y'n + x'$$

Jadi, rumus lengkap untuk menemukan 1 elemen indeks (x, y) dalam larik N_x × N_y dengan n × n tiling adalah

$$\text{index} = n2(Bxby + bx) + yn + x' = n2((Nx \div n)(y \div n) + x \div n) + (y \bmod n)n + (x \bmod n).$$

Ekspresi ini berisi banyak operasi perkalian, pembagian dan modulus bilangan bulat, yang mahal pada beberapa prosesor. Ketika n adalah pangkat dua, operasi ini dapat diubah menjadi bitshift dan operasi logika bitwise. Namun, seperti disebutkan di atas, ukuran ideal tidak selalu merupakan pangkat dua. Beberapa perkalian dapat diubah menjadi operasi shift/tambah, tetapi operasi pembagian dan modulus lebih bermasalah. Indeks dapat dihitung secara bertahap, tetapi ini akan membutuhkan penghitung pelacakan, dengan banyak perbandingan dan kinerja prediksi cabang yang buruk.



Gambar 12.42 Larik 2D bersusun yang terdiri dari Bx ×Dengan tiling masing-masing berukuran n kali n.

Namun, ada solusi sederhana; perhatikan bahwa ekspresi indeks dapat ditulis sebagai

$$\text{index} = Fx(x) + Fy(y),$$

di mana

$$Fx(x) = n2(x \div n) + (x \bmod n),$$
$$Fy(y) = n2(Nx \div n)(y \div n) + (y \bmod n)n.$$

Kami membuat tabulasi Fx dan Fy, dan menggunakan x dan y untuk menemukan indeks ke dalam larik data. Tabel ini akan terdiri dari elemen Nx dan Ny, masing-masing. Ukuran total tabel akan muat dalam cache data utama prosesor, bahkan untuk ukuran kumpulan data yang sangat besar.

Contoh: Tiling Dua Tingkat untuk Array 3D

Pemanfaatan TLB yang efektif juga menjadi faktor penting dalam kinerja algoritma. Teknik yang sama dapat digunakan untuk meningkatkan tingkat hit TLB dalam larik 3D dengan membuat bata m×m×m dari sel n×n×n. Sebagai contoh, volume 40×20×19 dapat didekomposisi menjadi 4×2×2 makrobata dari 2×2×2 bata berukuran 5×5×5 sel. Ini sesuai dengan m =2 dan n =5. Karena 19 tidak dapat difaktorkan dengan mn = 10, diperlukan satu tingkat padding. Ukuran yang berguna secara empiris adalah m =5untuk kumpulan data 16-bit dan m =6untuk kumpulan data float.

Indeks yang dihasilkan ke dalam array data dapat dihitung untuk setiap (x,y,z)tiga kali lipat dengan ekspresi

$$\begin{aligned}\text{index} = {} & ((x \div n) \div m)n3m3((Nz \div n) \div m)((Ny \div n) \div m) \\ & +((y \div n) \div m)n3m3((Nz \div n) \div m) \\ & +((z \div n) \div m)n3m3\end{aligned}$$

$$+((x \div n) \bmod m)n3m2$$
$$+((y \div n) \bmod m)n3m$$
$$+((z \div n) \bmod m)n3$$
$$+(x \bmod (n2))n2$$
$$+(y \bmod n)n$$
$$+(z \bmod n),$$

di mana N_x, N_y dan N_z adalah ukuran masing-masing dari dataset. Perhatikan bahwa, dalam kasus tingkat 2 yang lebih sederhana, ekspresi ini dapat ditulis sebagai:

$$\text{index} = Fx(x) + Fy(y) + Fz(z),$$

di mana

$$Fx(x) = ((x \div n) \div m)n3m3((Nz \div n) \div m)((Ny \div n) \div m)$$
$$+((x \div n) \bmod m)n3m2$$
$$+(x \bmod n)n2,$$
$$Fy(y) = ((y \div n) \div m)n3m3((Nz \div n) \div m)$$
$$+((y \div n) \bmod m)n3m +$$
$$+(y \bmod n)n,$$
$$Fz(z) = ((z \div n) \div m)n3m3$$
$$+((z \div n) \bmod m)n3$$
$$+(z \bmod n).$$

Pertanyaan yang Sering Diajukan

- Apakah pemasangan tiling benar-benar menghasilkan banyak perbedaan dalam kinerja?

Pada beberapa aplikasi rendering volume, strategi tiling dua tingkat dibuat sebagai faktor perbedaan kinerja sepuluh. Ketika array tidak muat di memori utama, array dapat secara efektif mencegah thrashing di beberapa aplikasi seperti pengeditan gambar.

- Bagaimana cara menyimpan daftar dalam struktur bersayap?

Untuk sebagian besar aplikasi, adalah layak untuk menggunakan array dan indeks untuk referensi. Namun, jika banyak operasi penghapusan yang akan dilakukan, maka adalah bijaksana untuk menggunakan daftar dan pointer yang ditautkan.

Catatan

Pembahasan struktur data bersayap-tepi didasarkan pada catatan saja dari Ching-Kuang Shene (Shene, 2003). Ada struktur data mesh yang lebih kecil daripada winged-edge. Pengorbanan dalam menggunakan struktur seperti itu dibahas dalam Directed Edges—A Representasi Skalabel untuk Segitiga Jerat (Campagna, Kobbelt, & Seidel, 1998). Diskusi tile-array didasarkan pada Interactive Ray Tracing untuk Visualisasi Volume (S. Parker et al., 1999). Struktur yang mirip dengan struktur tetangga segitiga dibahas dalam laporan teknis oleh Charles Loop (Loop, 2000). Sebuah diskusi tentang manifold dapat ditemukan dalam teks pengantar topologi (Munkres, 2000).

Latihan

1. Apa perbedaan memori untuk tetrahedron sederhana yang disimpan sebagai empat segitiga independen dan satu disimpan dalam struktur data tepi bersayap?
2. Buatlah diagram grafis scene untuk sepeda.
3. Berapa banyak tabel pencarian yang diperlukan untuk tiling tingkat tunggal dari array ndimensional?
4. Diketahui N segitiga, berapa jumlah minimum segitiga yang dapat ditambahkan ke pohon BSP yang dihasilkan? Berapa jumlah maksimumnya?

BAB 13
RAY TRACING LEBIH DALAM LAGI

Pelacak sinar adalah substrat hebat untuk membangun semua jenis efek rendering tingkat lanjut. Banyak efek yang membutuhkan usaha yang signifikan untuk dimasukkan ke dalam kerangka rasterisasi urutan objek, termasuk dasar-dasar seperti bayangan dan pantulan yang sudah disajikan di Bab 4, sederhana dan elegan dalam ray tracer. Beberapa ekstensi memungkinkan geometri yang lebih umum: instanting dan geometri padat konstruktif (CSG) adalah dua cara untuk membuat model lebih kompleks dengan kompleksitas minimal yang ditambahkan ke program. Ekstensi lain menambah jangkauan bahan yang dapat kami tangani: pembiasan melalui bahan transparan, seperti kaca dan air, dan pantulan mengkilap pada berbagai permukaan sangat penting untuk realisme di banyak scene.

Bab ini juga membahas kerangka umum penjaluran sinar distribusi (Cook, Porter, & Carpenter, 1984), perluasan yang kuat untuk gagasan penelusuran sinar dasar di mana beberapa sinar acak dikirim melalui setiap piksel dalam suatu gambar untuk menghasilkan gambar dengan tepi yang halus dan untuk sederhana dan elegan (jika lambat) menghasilkan berbagai efek dari bayangan lembut hingga kedalaman bidang kamera.

Harga keanggunan ray tracing ditentukan dalam hal waktu komputer: sebagian besar ekstensi ini akan melacak sejumlah besar sinar untuk setiap scene nontrivial. Karena itu, sangat penting untuk menggunakan metode yang dijelaskan dalam Bab 12 untuk mempercepat penelusuran sinar.

Catatan : Jika Anda memulai dengan loop perpotongan sinar bruteforce, Anda akan memiliki cukup waktu untuk menerapkan struktur akselerasi sambil menunggu gambar direnderer.

13.1 REFRAKSI DAN TRANSPARANSI

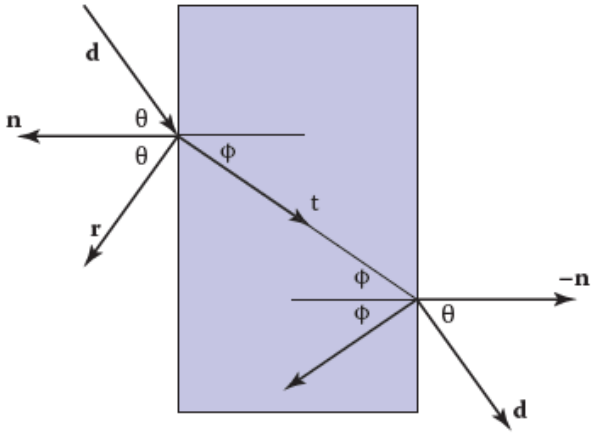
Dalam Bab 4, kita membahas penggunaan penelusuran sinar rekursif untuk menghitung pantulan spekular, atau cermin, dari permukaan. Jenis objek spekular lainnya adalah dielektrik—bahan transparan yang memantulkan cahaya. Berlian, kaca, air, dielektrik dairare. Dielektrik juga menyaring cahaya; beberapa kaca menyaring lebih banyak cahaya merah dan biru daripada cahaya hijau, sehingga kaca mengambil warna hijau. Ketika sinar merambat dari medium dengan indeks bias n ke medium dengan indeks bias n_t, sebagian cahaya ditransmisikan dan dibelokkan. Ini ditunjukkan untuk n_t > n dalam Gambar 13.1. Hukum Snell memberi tahu kita bahwa

$$n \sin \vartheta = nt \sin \varphi.$$

Menghitung sinus sudut antara dua vektor biasanya tidak semudah menghitung kosinus, yang merupakan produk titik sederhana untuk vektor satuan seperti yang kita miliki di sini. Dengan menggunakan identitas trigonometri sin2 + cos 2 =1, kita dapat menurunkan hubungan refraksi untuk cosinus:

$$cos^2\emptyset = 1\frac{n^2(1 - cos^2\theta)}{n_t^2}$$

Perhatikan bahwa jika n dan n_t dibalik, maka dan juga demikian seperti yang ditunjukkan di sebelah kanan Gambar 13.1.



Gambar 13.1 Hukum Snell menjelaskan bagaimana sudut Ø tergantung pada sudut θ dan indeks bias benda dan media sekitarnya.

Untuk mengubah sin ϕ dan cos ϕ menjadi vektor 3D, kita dapat menetapkan basis ortonormal 2D pada bidang normal permukaan, n, dan arah sinar, d. Dari Gambar 13.2, kita dapat melihat bahwa n dan b membentuk basis ortonormal untuk bidang bias. Dengan definisi, kita dapat menggambarkan arah sinar yang diubah, t, dalam hal dasar ini:

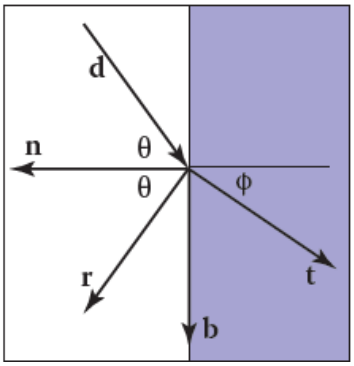
$$\mathbf{t} = \sin\varphi\mathbf{b} - \cos\varphi\mathbf{n}.$$

Karena kita dapat mendeskripsikan d dengan dasar yang sama, dan d diketahui, kita dapat menyelesaikan untuk b:

$$\mathbf{d} = \sin\vartheta\mathbf{b} - \cos\vartheta\mathbf{n},$$

$$b = \frac{d + n\cos\theta}{\sin\theta}$$

Perhatikan bahwa persamaan ini bekerja terlepas dari n dan nt mana yang lebih besar. Pertanyaan langsungnya adalah, “Apa yang harus Anda lakukan jika angka di bawah akar kuadrat negatif?” Dalam hal ini, tidak ada sinar bias dan semua energi dipantulkan. Ini dikenal sebagai pemantulan internal total, dan bertanggung jawab atas sebagian besar tampilan objek kaca yang kaya.



Gambar 13.2 Vektor n dan b membentuk basis ortonormal 2D yang sejajar dengan vektor transmisi t.

Reflektivitas dielektrik bervariasi dengan sudut datang menurut persamaan Fresnel. Cara lain untuk mengimplementasikan sesuatu yang dekat dengan persamaan Fresne adalah dengan menggunakan pendekatan Schlick (Schlick, 1994a),

$$R(\vartheta) = R0 + (1 - R0)\,(1 - \cos\vartheta)^5\,,$$

di mana R_0 adalah pemantulan pada kejadian normal:

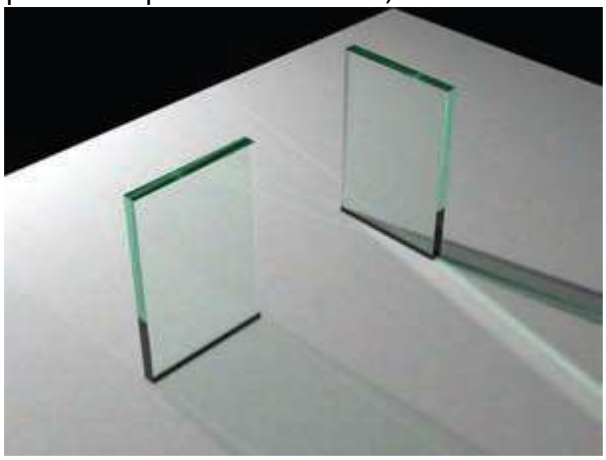
$$R_0 = \left(\frac{n_t - 1}{n_t - 1}\right)^2$$

Perhatikan bahwa suku cosθ di atas selalu untuk sudut di udara (yang lebih besar dari sudut internal dan eksternal relatif terhadap normal).

Untuk pengotor homogen, seperti yang ditemukan pada kaca berwarna biasa, intensitas sinar pembawa cahaya akan dilemahkan menurut Hukum Beer. Saat sinar merambat melalui medium, ia kehilangan intensitas menurut dI = CIdx, di mana dx adalah jarak. Jadi, dI/dx = CI. Kita dapat menyelesaikan persamaan ini dan mendapatkan eksponensial I = k exp(−Cx). Derajat redaman digambarkan oleh konstanta redaman RGB a, yang merupakan jumlah redaman setelah satu satuan jarak. Menempatkan dalam kondisi batas, kita tahu bahwa I(0) = I0, dan I(1) = aI(0). Yang pertama menyiratkan I(x)=I0 exp(−Cx). Yang terakhir menyiratkan I0a = I0 exp(−C), jadi−C = ln(a). Dengan demikian, rumus akhirnya adalah

aI(0). Di implikasikan *I*(*x*) = *I*0 exp(−*Cx*). Diimplikasikan setelahnya *I*0*a* = *I*0 exp(−*C*), so −*C* = ln(*a*). Sehingga rumus terakhirnya adalah *I*(*s*) = *I*(0)*e*ln(*a*)*s,*

di mana I(s) adalah intensitas sinar pada jarak dari antarmuka. Dalam praktiknya, kami melakukan rekayasa balik secara kasat mata, karena data seperti itu jarang mudah ditemukan. Pengaruh Hukum Bir dapat dilihat pada Gambar 13.3, di mana kaca berwarna hijau.



Gambar 13.3 Warna kaca dipengaruhi oleh refleksi internal total dan Hukum Beer. Jumlah cahaya yang ditransmisikan dan dipantulkan ditentukan oleh persamaan Fresne. Pencahayaan kompleks pada bidang dasar dihitung menggunakan penelusuran partikel seperti yang dijelaskan dalam Bab 23.

Untuk menambahkan bahan transparan ke kode kita, kita membutuhkan cara untuk menentukan kapan sinar akan "menjadi" objek. Cara paling sederhana untuk melakukannya adalah dengan mengasumsikan bahwa semua objek tertanam di udara dengan indeks bias sangat dekat dengan 1,0, dan garis normal permukaan menunjukkan "keluar" (menuju udara). Segmen kode untuk sinar dan dielektrik dengan asumsi ini adalah:

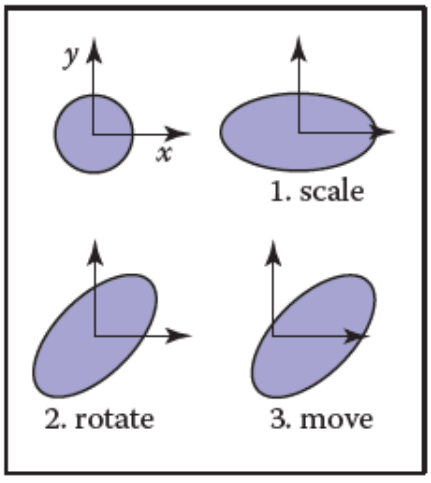
if (**p** is on a dielectric) **then**
r = reflect(**d**, **n**)
if (**d** · **n** < 0) **then**
refract(**d**, **n**, *n,* **t**)
c = −**d** · **n**
kr = *kg* = *kb* = 1
else

```
kr = exp(−art)
kg = exp(−agt)
kb = exp(−abt)
if refract(d,−n, 1/n, t) then
c = t · n
else
return k * color(p + tr)
R0 = (n − 1)2/(n + 1)2
R = R0 + (1 − R0)(1 − c)5
return k(R color(p + tr) + (1 − R) color(p + tt))
```

Kode di atas mengasumsikan bahwa log natural telah dilipat menjadi konstanta (a_r, a_g, a_b). Fungsi therefract mengembalikan false jika ada refleksi internal total, dan jika tidak, ia mengisi argumen terakhir dari daftar argumen.

13.2 INSTANCING

Sifat elegan dari ray tracing adalah memungkinkan pembuatan contoh yang sangat alami. Ide dasar pembuatan instance adalah untuk mendistorsi semua titik pada suatu objek dengan matriks transformasi sebelum objek ditampilkan. Misalnya, jika kita mengubah lingkaran satuan (dalam 2D) dengan faktor skala (2,1) berturut-turut di x dan y, kemudian memutarnya sebesar 45◦, dan memindahkan satu satuan ke arah x, hasilnya adalah elips dengan eksentrisitas 2 dan sumbu panjang sepanjang (x = y)-arah berpusat di (0,1) (Gambar 13.4). Hal utama yang membuat entitas tersebut menjadi "instance" adalah bahwa kita menyimpan lingkaran dan matriks transformasi komposit. Dengan demikian, konstruksi eksplisit elips dibiarkan sebagai operasi masa depan pada waktu render.



Gambar 13.4 Contoh lingkaran dengan deret tiga transformasi adalah elips.

Keuntungan instancing dalam ray tracing adalah kita dapat memilih ruang untuk melakukan perpotongan. Jika objek dasar terdiri dari sekumpulan titik, salah satunya adalah p, maka objek yang ditransformasi terdiri dari himpunan titik yang ditransformasikan oleh matriks M, di mana titik contoh ditransformasikan ke Mp. Jika kita memiliki raya+tb yang ingin kita potong dengan objek yang diubah, kita dapat memotong sinar transformasi terbalik dengan objek yang tidak diubah (Gambar 13.5). Ada dua keuntungan potensial untuk komputasi dalam ruang yang tidak diubah (yaitu, sisi kanan Gambar 13.5):

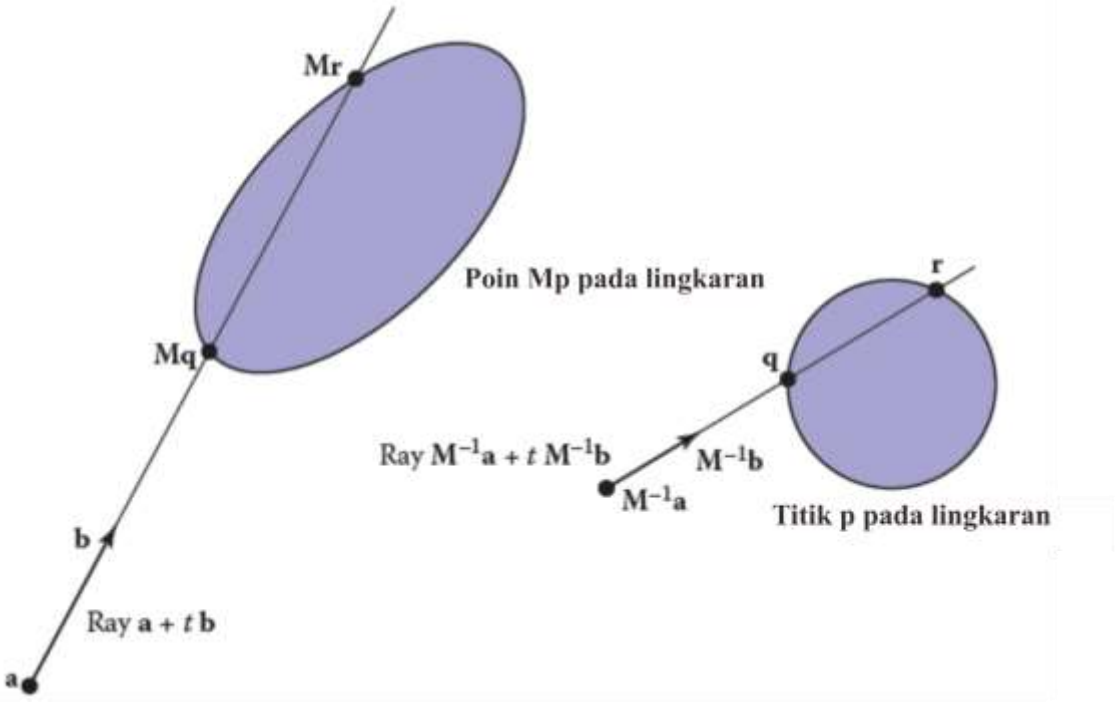
1. Objek yang tidak diubah mungkin memiliki rutinitas persimpangan yang lebih sederhana, misalnya, bola versus ellipsoid.

2. Banyak objek yang diubah dapat berbagi objek yang tidak diubah yang sama sehingga mengurangi penyimpanan, misalnya, kemacetan lalu lintas mobil, di mana masing-masing mobil hanyalah transformasi dari beberapa model dasar (tidak diubah).

Seperti yang dibahas dalam Bagian 6.2.2, vektor normal permukaan berubah secara berbeda. Dengan mengingat hal ini dan menggunakan konsep yang diilustrasikan pada Gambar 13.5, kita dapat menentukan perpotongan sinar dan objek yang ditransformasikan oleh matriks M. Jika kita membuat kelas instance dari tipe permukaan, kita perlu membuat fungsi hit:

```
instance::hit(ray a + tb, real t0, real t1, hit-record rec)
ray r_ = M^-1a + tM^-1b
if (base-object→hit(r_, t0, t1, rec)) then
rec.n = (M
−1)Trec.n
return true
else
        return false
```

Hal yang elegan tentang fungsi ini adalah bahwa parameter rec.t tidak perlu diubah, karena sama di kedua ruang. Perhatikan juga bahwa kita tidak perlu menghitung atau menyimpan matriks M.



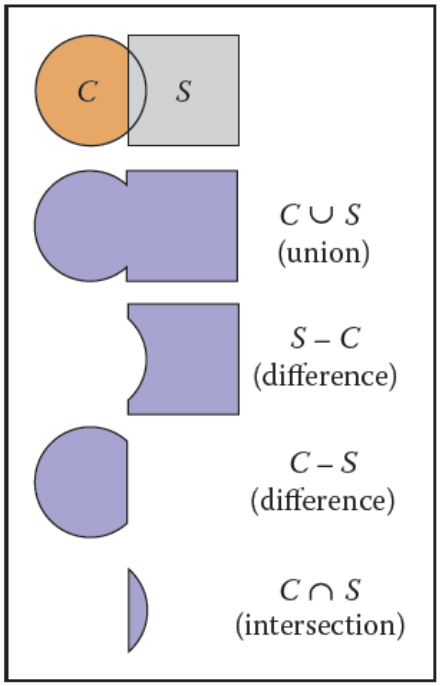
Gambar 13.5 Masalah perpotongan pada dua ruang adalah transformasi sederhana satu sama lain. Objek ditentukan sebagai matriks bulat plus M. Sinar ditentukan dalam ruang (dunia) yang diubah berdasarkan lokasi a dan arah b.

Ini memunculkan poin yang sangat penting: arah sinar b tidak boleh dibatasi pada vektor satuan panjang, atau tidak satu pun dari infrastruktur di atas pekerjaan. Untuk alasan ini, berguna untuk tidak membatasi arah sinar ke vektor satuan.

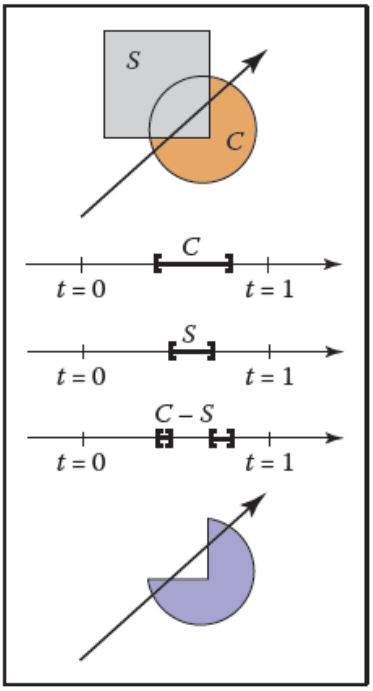
13.3 GEOMETRI SOLID KONSTRUKTIF

Satu hal yang menarik tentang raytracing adalah bahwa setiap geometri primitif yang perpotongannya dengan garis 3D dapat dihitung dapat ditambahkan dengan mulus ke ray

tracer. Ternyata juga mudah untuk menambahkan geometri padat konstruktif (CSG) ke pelacak sinar (Roth, 1982). Ide dasar CSG adalah menggunakan operasi himpunan untuk menggabungkan bentuk padat. Operasi dasar ini ditunjukkan pada Gambar 13.6. Operasi dapat dilihat sebagai operasi yang ditetapkan. Sebagai contoh, kita dapat menganggap C himpunan semua titik dalam lingkaran dan S himpunan semua titik dalam bujur sangkar. Operasi perpotongan C S adalah himpunan semua titik yang merupakan anggota C dan S. Operasi lainnya analog.



Gambar 13.6 Operasi CSG dasar pada lingkaran dan persegi 2D.



Gambar 13.7 Interval diproses untuk menunjukkan bagaimana sinar mengenai objek komposit.

Meskipun seseorang dapat melakukan CSG secara langsung pada model, jika semua yang diinginkan adalah gambar, kita tidak perlu mengubah model secara eksplisit. Sebagai gantinya, kami melakukan operasi yang ditetapkan secara langsung pada sinar ketika mereka berinteraksi dengan model. Untuk membuat ini alami, kita menemukan semua perpotongan sinar dengan model, bukan hanya yang terdekat. Misalnya, sinar a + tb mungkin mengenai bola pada t =1 dan t = 2. Dalam konteks CSG, kami menganggap ini sebagai sinar yang berada di dalam bola untuk t ∈ [1, 2]. Kita dapat menghitung "interval dalam" ini untuk semua permukaan dan melakukan operasi himpunan pada interval tersebut (ingat Bagian 2.1.2). Hal ini diilustrasikan pada Gambar 13.7, di mana interval hit diproses untuk menunjukkan bahwa ada dua interval di dalam objek perbedaan. Pukulan pertama untuk t > 0 adalah sinar yang sebenarnya berpotongan.

Dalam praktiknya, rutinitas persimpangan CSG harus mempertahankan daftar interval. Ketika hitpoint pertama ditentukan, properti material dan normal permukaan berhubungan dengan hitpoint. Selain itu, Anda harus memperhatikan masalah presisi karena tidak ada yang menghalangi pengguna untuk mengambil dua objek yang berbatasan dan mengambil persimpangan. Ini dapat dibuat kuat dengan menghilangkan interval yang ketebalannya di bawah toleransi tertentu.

13.4 DISTRIBUSI RAY TRACING

Untuk beberapa aplikasi, gambar yang dilacak dengan sinar terlalu "bersih". Efek ini dapat dikurangi dengan menggunakan ray tracing distribusi (Cook et al., 1984). Gambar ray-trace konvensional terlihat bersih, karena semuanya tajam; bayangannya sangat tajam, pantulannya tidak kabur, dan semuanya dalam fokus yang sempurna. Terkadang kita ingin memiliki bayangan yang lembut (seperti di kehidupan nyata), pantulannya kabur seperti dengan logam yang disikat, dan gambar memiliki derajat fokus yang bervariasi seperti pada foto dengan bukaan diafragma yang besar. Sementara menyelesaikan hal-hal ini dari prinsip pertama agak terlibat (seperti yang dikembangkan dalam Bab 23), kita bisa mendapatkan sebagian besar dampak visual dengan beberapa perubahan sederhana pada algoritma ray tracing dasar. Selain itu, kerangka kerja memberi kita cara yang relatif sederhana untuk antialias (ingat Bagian 8.3) gambar.

Antialiasing

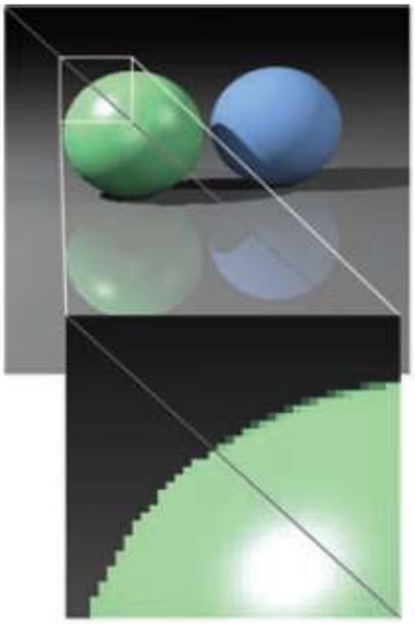
Ingatlah bahwa cara sederhana untuk antialias suatu gambar adalah dengan menghitung warna rata-rata untuk area piksel daripada warna di titik tengah. Dalam ray tracing, komputasi primitif kami adalah menghitung warna pada suatu titik di layar. Jika kita rata-rata banyak dari titik-titik ini di seluruh piksel, kita mendekati rata-rata yang sebenarnya. Jika koordinat layar yang membatasi piksel adalah [i,i + 1] × [j,j + 1] , maka kita dapat mengganti loop:

for each pixel (*i, j*) **do**
cij = ray-color(*i*+ 0.5*, j* + 0.5)

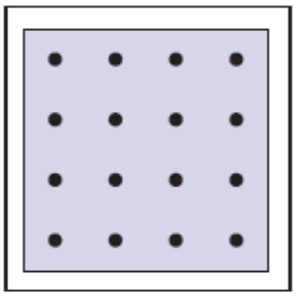
dengan kode yang mengambil sampel pada kisi sampel n × n reguler dalam setiap piksel:

for each pixel (*i, j*) **do**
c = 0
for *p* = 0 to *n* − 1 **do**
for *q* = 0 to *n* − 1 **do**
c = *c* + ray-color(*i*+ (*p* + 0.5)*/n*, *j* + (*q* + 0.5)*/n*)
cij = *c*/n^2

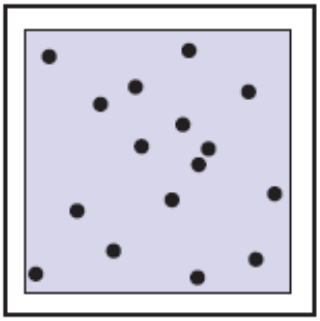
Ini biasanya disebut pengambilan sampel reguler. 16 lokasi sampel dalam piksel untuk n =4 ditunjukkan pada Gambar 13.9. Perhatikan bahwa ini menghasilkan jawaban yang sama seperti merender gambar ray-traced tradisional dengan satu sampel per piksel pada resolusi nxn kali nyn dan kemudian rata-rata blok n kali n piksel untuk mendapatkan gambar nx kali n_y.



Gambar 13.8 Scene sederhana yang dirender dengan satu sampel per piksel (setengah kiri bawah) dan sembilan sampel per piksel (setengah kanan atas)



Gambar 13.9 Enam belas sampel reguler untuk satu piksel.



Gambar 13.10 Enam belas sampel acak untuk satu piksel.

Salah satu masalah potensial dengan mengambil sampel dalam pola reguler dalam piksel adalah artefak reguler seperti pola moire´ dapat muncul. Artefak ini dapat diubah menjadi noise dengan mengambil sampel dalam pola acak dalam setiap piksel seperti yang ditunjukkan pada Gambar 13.10. Ini biasanya disebut pengambilan sampel acak dan hanya melibatkan sedikit perubahan pada kode:

for each pixel (*i, j*) **do**
c = 0
for *p* = 1 to *n*2 **do**
c = *c*+ ray-color(*i*+ *ξ*, *j* + *ξ*)
cij = *c*/*n*^{2}

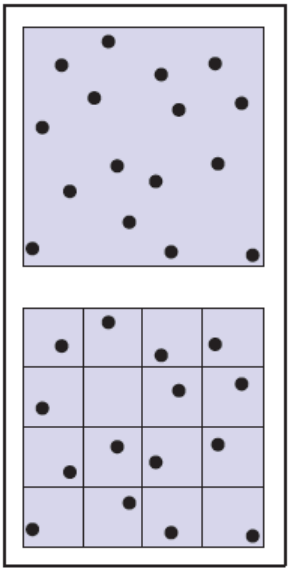
Di sini ξ adalah panggilan yang mengembalikan nomor acak seragam dalam kisaran [0,1). Sayangnya, kebisingan bisa sangat tidak menyenangkan kecuali banyak sampel yang diambil. Kompromi adalah membuat strategi hibrid yang secara acak mengganggu jaringan biasa:

for each pixel (*i, j*) **do**
c = 0
for *p* = 0 to *n* − 1 **do**
for *q* = 0 to *n* − 1 **do**
c = *c* + ray-color(*i*+ (*p* + *ξ*)/*n*, *j* + (*q* + *ξ*)/*n*)
cij = *c*/*n*^{2}

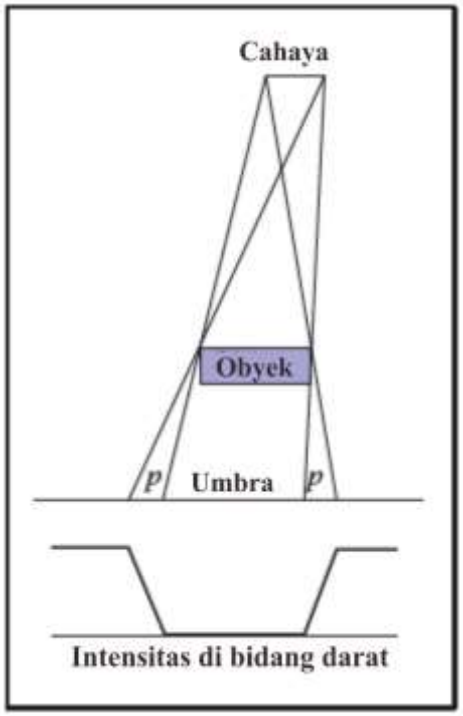
Metode tersebut biasanya disebut jittering atau stratified sampling (Gambar 13.11).

Bayangan Lembut

Alasan mengapa bayangan sulit ditangani dalam ray tracing standar adalah karena cahaya adalah titik atau arah yang sangat kecil sehingga dapat terlihat atau tidak terlihat. Dalam kehidupan nyata, lampu memiliki area bukan nol dan dengan demikian dapat terlihat sebagian. Ide ini ditunjukkan dalam 2D pada Gambar 13.12. Daerah di mana cahaya sama sekali tidak terlihat disebut umbra. Daerah yang terlihat sebagian disebut penumbra. Tidak ada istilah yang umum digunakan untuk daerah yang tidak berada dalam bayangan, tetapi kadang-kadang disebut anti-umbra.



Gambar 13.11 Enam belas sampel bertingkat (bergetar) untuk satu piksel yang ditampilkan dengan dan tanpa bin yang disorot. Ada tepat satu sampel acak yang diambil dalam setiap bin.



Gambar 13.12 Bayangan lembut memiliki transisi bertahap dari daerah yang tidak dibayangi ke daerah yang dibayangi. Zona transisi adalah "penumbra" yang dilambangkan dengan p pada gambar.

Kunci untuk menerapkan bayangan lembut adalah entah bagaimana memperhitungkan cahaya sebagai area daripada titik. Cara mudah untuk melakukan ini adalah dengan memperkirakan cahaya dengan set N lampu titik terdistribusi masing-masing dengan satu N intensitas cahaya dasar. Konsep ini diilustrasikan di sebelah kiri Gambar 13.13 di mana sembilan lampu digunakan. Anda dapat melakukan ini dalam ray tracer standar, dan ini adalah trik umum untuk mendapatkan softshadows di off-the-shelfrenderer. Ada dua potensi masalah dengan teknik ini. Pertama, biasanya lusinan lampu titik diperlukan untuk mencapai hasil visual yang mulus, yang sangat memperlambat program. Masalah kedua adalah bahwa bayangan memiliki transisi yang tajam di dalam penumbra.

Penelusuran sinar distribusi memperkenalkan perubahan kecil pada kode bayangan. Alih-alih merepresentasikan cahaya area pada sejumlah sumber titik yang terpisah, kita merepresentasikannya sebagai bilangan tak berhingga dan memilih satu secara acak untuk setiap sinar pandang. Ini sama dengan memilih titik acak pada lampu untuk setiap titik permukaan yang menyala seperti yang ditunjukkan di sebelah kanan Gambar 13.13.

Jika cahaya adalah jajar genjang yang ditentukan oleh titik sudut c dan dua vektor sisi dan b (Gambar 13.14), maka pilihlah titik dominan adalah lurus ke depan

$$\mathbf{r} = \mathbf{c} + \xi_1\mathbf{a} + \xi_2\mathbf{b},$$

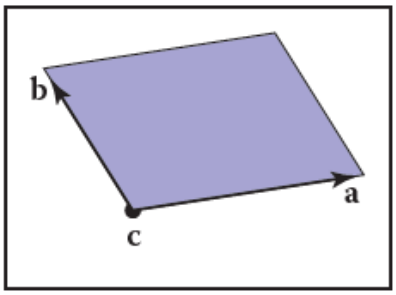
di mana $\xi 1$ dan $\xi 2$ adalah bilangan acak seragam dalam rentang [0,1). Kami kemudian mengirim sinar bayangan ke titik ini seperti yang ditunjukkan di sebelah kanan pada Gambar 13.13. Perhatikan bahwa arah sinar ini bukanlah satuan panjang, yang mungkin memerlukan beberapa modifikasi pada pelacak sinar dasar Anda tergantung pada asumsinya. Kami sangat ingin titik jitter di atas cahaya. Namun, bisa berbeda untuk menerapkan ini tanpa berpikir. Kami tidak ingin selalu sinar di sudut kiri atas piksel menghasilkan sinar bayangan ke sudut kiri atas cahaya. Sebagai gantinya kami ingin mengacak sampel, sehingga sampel piksel dan sampel cahaya masing-masing gelisah, tetapi agar tidak ada korelasi antara sampel piksel dan

sampel lampu. Cara yang baik untuk mencapai ini adalah dengan menghasilkan dua set berbeda dari n^2 sampel jitter dan meneruskan sampel ke rutinitas sumber cahaya:

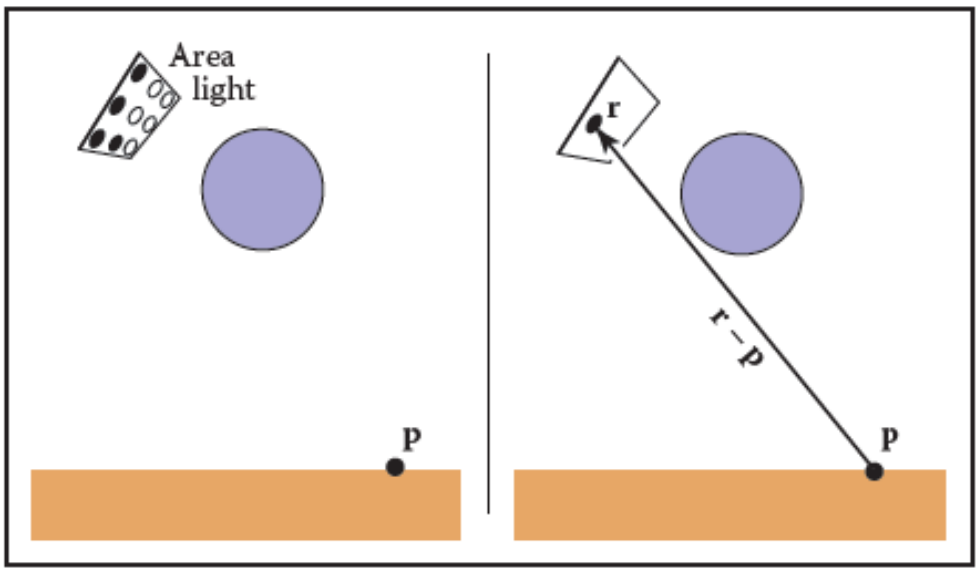
for each pixel (*i, j*) **do**
c = 0
generate *N* = *n*2 jittered 2D points and store in array r[ ]
generate *N* = *n*2 jittered 2D points and store in array s[ ]
shuffle the points in array s[ ]
for *p* = 0 to *N* − 1 **do**
c = *c* + ray-color(*i* + r[*p*].x(), *j* + r[*p*].y(), s[*p*])
cij = *c*/*N*

Rutinitas ini akan menghilangkan koherensi antara array dan. Rutinitas bayangan hanya akan menggunakan titik acak 2D yang disimpan di s[p] daripada memanggil generator angka acak. Sebuah shuffleroutineforanarray yang diindeks dari 0 hingga N−1 adalah:

for *i* = *N* − 1 downto 1 **do**
choose random integer *j* between 0 and *i* inclusive
swap array elements *i* and *j*



Gambar 13.13 Kiri: lampu area dapat didekati dengan sejumlah lampu titik; empat dari sembilan titik terlihat p sehingga di penumbra. Kanan: titik acak pada cahaya dipilih untuk sinar bayangan, dan memiliki kemungkinan mengenai cahaya atau tidak.

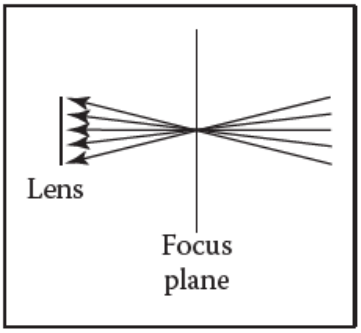


Gambar 13.14 Geometri cahaya jajaran genjang ditentukan oleh titik sudut dan dua vektor tepi.

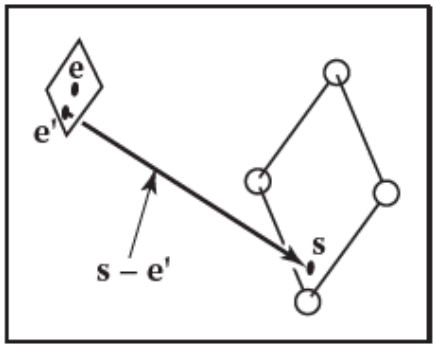
Depth-of-Field (DoF)

Efek fokus lembut yang terlihat di sebagian besar foto dapat disimulasikan dengan mengumpulkan cahaya pada "lensa" ukuran bukan nol daripada pada satu titik. Ini disebut

kedalaman bidang. Lensa mengumpulkan cahaya dari kerucut arah yang memiliki puncaknya pada jarak di mana semuanya tidak fokus (Gambar 13.15). Kita dapat menempatkan "windows" memakai sampel di pesawat di mana segala sesuatu dalam fokus (bukan di pesawat = n seperti sebelumnya) dan lensa di mata. Jarak ke bidang di mana segala sesuatu berada dalam fokuskita sebut bidang fokus, dan jarak yang ditentukan oleh pengguna, sama seperti jarak ke bidang fokus di kamera nyata yang diatur oleh pengguna atau pencari jarak.



Gambar 13.15 Rata-rata lensa di atas kerucut arah yang mengenai lokasi piksel yang dijadikan sampel.



Gambar 13.16 Untuk menciptakan efek depth-of-field, mata dipilih secara acak dari daerah persegi.

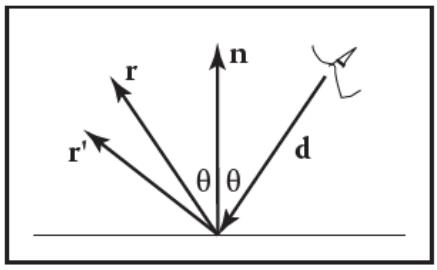
Untuk menjadi yang paling setia pada kamera nyata, kita harus membuatlensa disk. Namun, kita akan mendapatkan efek yang sangat mirip dengan lensa persegi (Gambar 13.16). Jadi kami memilih panjang sisi lensa dan mengambil sampel acak di atasnya. Asal usul sinar pandang adalah posisi terganggu ini daripada posisi mata. Sekali lagi, rutinitas perpindahan digunakan untuk mencegah korelasi dengan posisi sampel piksel. Contoh menggunakan 25 sampel per piksel dan lensa disk besar ditunjukkan pada Gambar 13.17.

Refleksi Glossy

Beberapa permukaan, seperti logam yang disikat, berada di antara cermin yang ideal dan permukaan yang menyebar. Beberapa gambar yang dapat dilihat terlihat dalam pantulan, tetapi kabur. Kita dapat mensimulasikan ini dengan secara acak mengganggu sinar pantul spekular ideal seperti yang ditunjukkan pada Gambar 13.18.

Gambar 13.17 Contoh depthoffield. Kaustik dalam bayangan gelas anggur dihitung menggunakan penelusuran partikel seperti yang dijelaskan dalam Bab 23.

Hanya dua detail yang perlu dikerjakan: bagaimana memilih vektor dan apa yang harus dilakukan ketika sinar terganggu yang dihasilkan berada di bawah permukaan tempat sinar dipantulkan. Detail terakhir biasanya diselesaikan dengan mengembalikan warna nol ketika sinar berada di bawah permukaan.



Gambar 13.18 Sinar pantul terganggu ke vektor acak r′.

Untuk memilih r′, kita kembali mengambil sampel persegi acak. Kotak ini tegak lurus terhadap r dan memiliki lebar a yang mengontrol derajat keburaman. Kita dapat mengatur orientasi persegi dengan membuat basis ortonormal dengan w = r menggunakan teknik di Bagian 2.4.6. Kemudian, kami membuat titik acak di kotak 2D dengan panjang sisi a berpusat di titik asal. Jika kita memiliki titik sampel 2D $(\xi, \xi') \in [0, 1]^2$, maka titik analog pada persegi yang diinginkan adalah

$$u = -\frac{a}{2} + \xi a$$
$$v = -\frac{a}{2} + \xi' a$$

Karena bujur sangkar yang akan kita ganggu sejajar dengan vektor u dan v, sinar r hanya

$$\mathbf{r'} = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

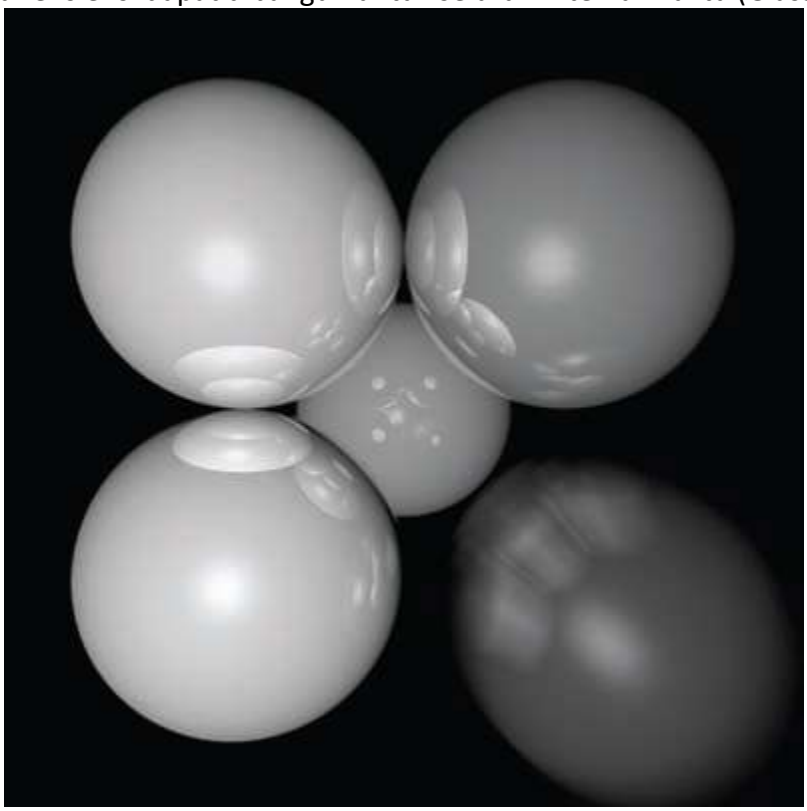
Perhatikan bahwa r′ tidak perlu merupakan vektor satuan dan harus dinormalisasi jika kode kita memerlukan itu untuk arah sinar.

Motion Blur

Kita dapat menambahkan tampilan kabur ke objek seperti yang ditunjukkan pada Gambar 13.19. Ini disebut gerakan kabur dan merupakan hasil dari gambar yang terbentuk selama rentang waktu yang tidak nol. Dalam kamera nyata, aperture terbuka untuk beberapa interval waktu selama objek bergerak. Kita dapat mensimulasikan bukaan bukaan dengan menyetel variabel waktu mulai dari T_0 hingga T_1. Untuk setiap sinar pandang, kami memilih waktu acak,

$$T = T_0 + \xi(T_1 - T_0).$$

Kita mungkin juga perlu membuat beberapa objek untuk bergerak seiring waktu. Misalnya, kita mungkin memiliki bola bergerak yang pusatnya bergerak dari c_0 ke c_1 selama interval. Diberikan T, kita dapat menghitung pusat sebenarnya dan melakukan perpotongan sinar dengan bola itu. Karena setiap sinar berada pada waktu yang berbeda, masing-masing akan menghadapi bola pada posisi yang berbeda, dan tampilan akhirnya akan kabur. Perhatikan bahwa kotak pembatas untuk bola bergerak harus membatasi seluruh jalurnya sehingga struktur efisiensi dapat dibangun untuk seluruh interval waktu (Glassner, 1988).



Gambar 13.19 Bola kanan bawah sedang bergerak, dan hasil tampilan kabur. Gambar milik Chad Barb.

Catatan

Ada banyak, banyak metode lanjutan lainnya yang dapat diimplementasikan dalam kerangka kerja penelusuran sinar. Beberapa sumber untuk informasi lebih lanjut adalah Glassner's An Introduction to Ray Tracing dan Principles of Digital Image Synthesis, Shirley's Realistic Ray Tracing, dan Pharr and Humphreys's Physically Based Rendering: From Theory to Implementation.

Pertanyaan yang Sering Diajukan

- Apa struktur efisiensi perpotongan sinar terbaik?

Struktur yang paling populer adalah pohon partisi ruang biner (pohon BSP), grid subdivisi seragam, dan hierarki volume pembatas. Kebanyakan orang yang menggunakan pohon BSP membuat bidang pemisah sejajar sumbu, dan pohon seperti itu biasa disebut pohon k-d. Tidak ada jawaban yang jelas untuk mana yang terbaik, tetapi semuanya jauh, jauh lebih baik daripada pencarian brute-force dalam praktiknya. Jika saya menerapkan hanya satu, itu akan menjadi hierarki volume yang terbatas karena kesederhanaan dan kekuatannya.

- Mengapa orang menggunakan kotak pembatas daripada bola atau ellipsoid?

Terkadang bola atau ellipsoid lebih baik. Namun, banyak model memiliki elemen poligonal yang dibatasi oleh kotak, tetapi akan sulit untuk mengikatnya dengan ellipsoid.

Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

DASAR DESAIN GRAFIS

JILID 1

Bio Data Penulis

Penulis lahir di Semarang pada tanggal 1 Maret 1983. Penulis menempuh pendidikan Sarjana Teknik Elektro di Universitas Kristen Satya Wacana (UKSW), lulus tahun 2004, kemudian tahun 2005 melanjutkan studi pada Magister Desain pada Fakultas Seni Rupa dan Desain, Institut Teknologi Bandung (ITB), dan melanjutkan studi pada program studi Teknologi Multimedia pada Swinburne University of Technology Australia.

Penulis sejak tahun 2010, menjadi dosen pada program studi Desain Grafis Universitas Sains dan Teknologi Komputer (Universitas STEKOM), memiliki jabatan fungsional Lektor 300 dan sedang proses mengajukan kenaikan jabatan fungsional menjadi Lektor Kepala. Penulis juga seorang wirausaha di bidang toko online yang berhasil di kota Semarang dan juga aktif sebagai freelancer dalam bidang fotografi, web design dan multimedia.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-623-5734-46-0 (jil.1 PDF)
9 786235 734460

Dr. Mars Caroline Wibowo, ST, M.Mm.Tech.

DASAR DESAIN GRAFIS

JILID 1

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id