SCUT 2019 Fall Computer Graphic Review

qixuan

December 10, 2019

Abstract

Chapter 1: Computer Graphics hot topics
Chapter 2: Raster Graphics
Chapter 3: OpenGL basic convections, Coordinate system, Geometric Primitives
Chapter 4: 2D Modeling Transformations, Matrix Representation, Homogeneous Coordinates
Chapter 5: Rendering 3D Scenes Pipeline, 3D Scene Representation, Euclidean Spaces
Chapter 6: Triangle Rasterlization, Edge Equations
Chapter 7: Cohen-Sutherland Line Clipping, Cyrus-Beck algorithm
Chapter 8: Basics of Color, Color Spaces
Chapter 9: Light Source, Ambient Light, Surface Reflection, Phong lighting model
Chapter 10: Global Illumination, Algorithm of ray tracing
Chapter 11: Three shading models, Morphing

Chapter 1: CG hot topics

Geometric modeling
Methods and algorithms for the mathematical description of shapes(3D modeling, Clothing simulation, Feather, Mesh Editing and Deformation)
Rendering
Photo-realistic rendering(Global illumination, BRDF)
Image processing
Color conversion, transfer, analogy, Deblurring, Image/Video resizing, Video Stablization...
Computer animation
Motion capture, Motion editing
GPU acceleration
GPU(Graphic Processing Unit): reducing data exchange with.
CUDA(Compute Unified Device Architecture): a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)
Virtual reality

Chapter 2: Raster Graphics

CRTs(Cathode Ray Tubes) device using raster display
Raster displays(early 70s)
Raster: A rectangular array of points or dots
Pixel: One dot or picture element of the raster
Scan line: A row of pixels
Raster Graphics
like TV, scan all pixels in regular pattern
use **frame buffer** to eliminate **sync problems**

Chapter 3: OpenGL Basic convections

Functions in OpenGL start with gl
Most functions just **gl**(glColor())
Functions start with **glu** are **utility functions**(gluLookAt())
Functions start with **glx** are for interfacing with the X Windows system(in gfx.c)
Constants: GL_2D, GL_RGB, ...
Data types: GLbyte, GLfloat, ...
Function names indicate argument type and number
Functions ending with **f** take **floats**
Functions ending with **i** take **ints**
Functions ending with **b** take **bytes**

Functions ending with **ub** take **unsigned bytes**

Functions ending with **v** take **an array**

Examples

glColor**3f**() takes 3 floats

glColor**4fv**() takes an array of 4 floats

Variables written in CAPITAL letters

eg. GLUT_SINGLE, GLUT_RGB

usually constants

use the **bitwise or** command $(x|y)$ to combine constants

OpenGL operations as an infinite loop

Put things in the scene(points, colored lines, textured polys)

Describe the camera(location, orientation, field of view)

Listen for keyboard/mouse events

Render — draw the scene

Rendering

Typically execution of OpenGL commands

Converting geometric/mathematical object descriptions into frame buffer values

OpenGL can render

Geometric primitives(Lines, Points, Polygons, etc...)

Bitmaps and Images(Images and Geometry linked through texture mapping(Graphics Pipeline))

Chapter 3: Coordinate System

the world, the local, the camera coordinate(OpenGL)

OpenGL coordinate system

right-handed(cartesian coordinate system, orthogonal)

The camera defaults to look down negative z-axis

transformation matrix

We store the transformation matrix instead of the final desired matrix.

matrix multiplication

We need Transformation Matrix to transform the vertex from the real world to the computer window

This important transformation matrix is stored as the **MODELVIEW** matrix

Note OpenGL preserves a similar matrix to describe the **camera type** and this is called the **PROJECTION_MATRIX**

Geometric Primitives

All geometric primitives are specified by vertices

glBegin(mode) and glEnd() delimit an object mode can be one of the following

GL_POINTS

GL_LINES

GL_POLYGON

GL_LINE_STRIP
GL_TRIANGLE_STRIP
GL_TRIANGLES
GL_QUADS
GL_LINE_LOOP
GL_QUAD_STRIP
GL_TRIANGLE_FAN
(eg.
Points:
glBegin(GL_POINTS);
glVertex2i(0, 0);
glVertex2i(0, 1);
glVertex2i(1, 0);
glVertex2i(1, 1);
glEnd();
Line Loop(Polyline):
glBegin(GL_LINE_LOOP);
glVertex2i(0, 0);
glVertex2i(0, 1);
glVertex2i(1, 0);
glVertex2i(1, 1);
glEnd();
)

Polygon Issues

OpenGL will only display polygons correctly that are **Simple**(edges cannot cross), **Convex**(All points on line segment between two points in a polygon are also in the polygon) and **Flat**(all vertices are in the same plane)

User program can check weather above true, OpenGL will produce output but it may not be what is desired.

Triangles satisfy all conditions, that's why we need **Triangulation algorithms**.

Attributes

Point:

Point size: **glPointSize(2.0);**

Point color: **glColor3f(0.0, 0.0, 1.0);**(blue point)

Line:

Line width: **glLineWidth(2.0);**

Line color: **glColor3f(0.0, 0.0, 1.0);**(blue line)

Face:

Front and/or back: **GL_FRONT, GL_BACK, GL_FRONT_AND_BACK**

Face color: **glColor3f(0.0, 0.0, 1.0);**(blue face)

Chapter 4: 2D Modeling Transformation

Modeling Coordinates, World Coordinate

Scale

Uniform scaling: this scalar is the same for all components

Non-uniform scaling: different scalars per component

Rotate

2-D Rotation:

$x^{'}$ is a linear combination of x and y

$y^{'}$ is a linear combination of x and y

even though $\sin(\theta)$ and $\cos(\theta)$ are both non-linear.

Translate

Transformations can be combined with simple Algebra.

Matrix representation

2D Identity

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

2D Scale around (0, 0)

$$\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

2D Rotate around (0, 0)

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

2D Shear

$$\begin{bmatrix} 1 & sh_x \\ sh_y & 1 \end{bmatrix}$$

2D Mirror about Y axis

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

2D Mirror over (0, 0)

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

2D Translation ? NO!: Only **linear** 2D transformations(Scale, Rotation, Shear, Mirror) can be represented with a 2 by 2 matrix

Properties of linear transformations

Satisfies: $T(s_1P_1 + s_2P_2) = s_1T(p_1) + s_2T(p_2))$

Origin maps to origin

Lines map to lines

Parallel lines remain parallel

Ratios are preserved

Closed under composition

Homogeneous Coordinates

$x^{'} = x + t_x$

$y^{'} = y + t_y$

Q: Since it has no 2D matrix, how can we represent translation as a 3x3 matrix?

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow^{homogeneouscoords} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

(x, y, w) represents a point at location $(\frac{x}{w}, \frac{y}{w})$

A:

$$Translation = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix Composition

$p^{'} = T \cdot R \cdot S \; p$

Chapter 5: Rendering 3D Scenes Pipeline

Input: 3D Geometric Primitives(model/camera parameters)

Modeling Transformation: Transform into 3D world coordinate system

Lighting: Illuminate according to lighting and reflectance

Viewing Transformation: Transform into 3D camera coordinate system

Projection Transformation: Transform into 2D screen coordinate system

Clipping: Clip primitive outside camera's view

Scan Conversion(Raster Graphics Algorithm): Draw pixels(includes texturing, hidden surfaces, ...)

Output: Image(Frame buffer)

Rendering: Transformations

Modeling transforms(decide the object)

Viewing transforms(Move the camera)

Projection transforms(Change the type of camera)

Transformation Matrix do transformation for us

modeling transforms are encapsulated in the OpenGL modelview matrix **GL_MODELVIEW**

projection is also represented as a matrix

Projection Matrix

To represent orthographic and perspective projection with the **projection matrix**:

In OpenGL two matrix stack to operate the transformation

glMatrixMode(GL_PROJECTION | GL_MODELVIEW)

Use **glPushMatrix(); glPopMatrix()** to separate a individual transformation, like /begin, /end.

[glPushMatrix(): copy the top matrix and push into the stack]

[glPopMatrix(): pop up the top matrix, recover to the previous one]

The rendering pipeline: Move models, Illuminate, Move camera, Clip, Project to display, Rasterize.

Chapter 5: 3D Scene Representation

3D point

3D vector

Magnitude: $||V|| = \sqrt{(dx^2 + dy^2 + dz^2)}$

has no location

vector space

affine space = vector space + position and distance

Coordinate Systems

Point-point subtraction yields a vector

3D Line Segment

3D Ray

3D Line

3D Line-Slope Intercept

Chapter 5: Euclidean Space

Euclidean affine space = affine space + dot product

$v_1 \cdot v_2 = x_1x_2 + y_1y_2 + z_1z_2$

$u \cdot v = |u||v| \cos(\theta)$

Cross product

$||v_1 \times v_2|| = 2*$Area of triangle

Right hand rules

Determinant and Matrix

3D primitives

Chapter 6: Triangle Rasterlization

In interactive graphics, polygons rule the world:

Lowest common denominator for surfaces

(Can represent any surface with arbitrary accuracy)

(Splines, mathematical functions, volumetric isosurfaces...)

Mathematical simplicity lends itself to simple, regular rendering algorithms

Triangle is the minimal unit of a polygon

All polygons can be broken up into triangles

Triangles are guaranteed to be **Planar** and **Convex**

Triangulation

Convex polygons easily triangulated

Concave polygons present a challenge

Rasterizing Triangles

Interactive graphics hardware commonly uses **edge walking** or **edge equation** techniques for rasterizing triangles

Chapter 6: Edge Equations

Edge Walking

Draw edges vertically,namely

Fill in horizontal spans for each scanline

Algorithm 1 Edge Walking

Input: vertices T[3]
Output: Rasterized triangle

```
for each edge pair of the Triangle do
initialize x_L, x_R
compute dx_L/dy_L and dx_R/dy_R
for scanline at y do
for (int x = x_L; x <= x_R; x++) do
set_pixel(x, y)
end for
x_L += dx_L/dy_L, x_R += dx_R/dy_R, update y
end for
if x_L or x_R == T[0] or T[1] or T[2] do
continue
end if
end for
```

Advantages: Simple

Disadvantages:

very serial (one pixel at a time) can't parallelize

inner loop bottleneck if lots of computation per pixel

Special cases: horizontal edges: computing intersection causes divide by 0

Special cases: Sliver: not even a single pixel wide

Edge Equations

the implicit equation of a line

$$Ax + By + C = 0$$

Given a point (x, y)

On the line: $Ax + By + C = 0$
"Above" the line: $Ax + By + C > 0$
"Below" the line: $Ax + By + C < 0$
Edge equations thus define two **half-spaces**
And a triangle can be defined as the intersection of three positive half-spaces
So...simply turn on those pixels for which all edge equations evaluate to ¿ 0
compute min, max bounding box
we can find edge equation from two vertices.

Q: Given three corners P 0 , P 1 , P 2 of a triangle, what are our three edges?How do we make sure the half-spaces defined by the edge equations all share the same sign on the interior of the triangle?

A: **Be consistent, clockwise** (Ex: $[P_0, P_1], [P_1, P_2], [P_2, P_0]$)

Algorithm 2 Use Edge Equation to rasterize triangle

```
1 findBoundingBox(&x_min, &x_max, &y_min, &y_max);
2 setupEdges(&a_0, &b_0, &c_0, &a_1, &b_1, &c_1, &a_2, &b_2, &c_2);
3 for y from y_min to y_max do
4 for x from x_min to x_max do
5 if a_0*x+b_0*y+c_0 > 0 && a_1*x+b_1*y+c_1 > 0 && a_2*x+b_2*y+c_2 > 0
  do
6 setupPixel(x, y)
7 end if
8 end for
9 end for
```

Algorithm 3 General Polygon Rasterization

```
1 for each scanline do
2 edgeCnt = 0
3 for each pixel on scanline (left to right) do
4 if oldpixel → newpixel crosses edge do
5 edgeCnt++
6 if edgeCnt % 2 do
7 setPixel(pixel)
8 end if
9 end if
10 end for
11 end for
```

Chapter 7: Cohen-Sutherland Line Clipping

Divide view window into regions defined by window edges
Assign each region a 4-bit outcode

Algorithm 4 int computeOutcode(x, y)

Input: Y_{max}, Y_{min}, X_{max}, X_{min}, (x, y)
Output: the 4-bit outcode of point (x, y)

```
 1 code = 0
 2 if y > Y_max do
 3 code = 8
 4 else if y < Y_min do
 5 code = 4
 6 end if
 7 if x > X_max do
 8 code = code + 2
 9 else if x < X_min do
10 code = code + 1
11 end if
12 return code
```

Cohen-Sutherland Line Clipping

Algorithm 5 Cohen-Sutherland Line Clipping

```
 1 for each line segment do
 2 Assign an outcode to each endpoint according to the area
 3 if bitwise OR == 0 do
 4 trivial accept
 5 else if bitwise AND != 0 do
 6 trivial reject
 7 else
 8 split line segment
 9 end if
10 end if
11 end for
```

Use outcodes to quickly eliminate/include lines

Is best algorithm when trivial accepts/rejects are common

Must compute viewing window clipping of the remaining lines

Non-trivial clipping cost

Redundant clipping of some lines

More efficient algorithms exist

Chapter 7: Cyrus-Beck Algorithm

We wish to optimize line/line intersection

Start with parametric equation of line:

$$P(t) = P_0 + (P_1 - P_0)t$$

And **a point** and **normal** for **each edge**:

$$P_L, N_L$$

Find t such that

$$N_L \cdot [P(t) - P_L] = 0$$

$P(t)$ is the intersection point of a line segment and a view window edge
Solve for **t**:

$$t = \frac{N_L[P_L - P_0]}{N_L[P_1 - P_0]}$$

Here we store $N_L[P_1 - P_0]$

Algorithm 6 Cyrus-Beck Algorithm

1 Compute t for line intersection with all four edges, store flag = $N_L[P_1 - P_0]$
2 Discard all $t < 0$ and $t > 1$
3 **for** each line in scene **do**
4 Compute PE with largest t(flag < 0)
5 Compute PL with smallest t(flag > 0)
6 Clip to these two points

Cyrus-Beck
Computation of t-intersections is cheap
Computation of (x,y) clip points is only done once
Algorithm doesn't consider trivial accepts/rejects
Best when many lines must be clipped
Liang-Barsky: Optimized Cyrus-Beck

Chapter 8: Basic of Color

Elements of color: Illumination, Reflectance, Perception
Physics:
Illumination: Electromagnetic spectra
Reflection: Material properties and Surface geometry and microgeometry (i.e., polished versus matte versus brushed)
Perception:
Physiology and neurophysiology
Perceptual psychology
Trichromacy:
RGB
visible electromagnetic spectrum: **400-700 nm**

Most of the light we see is reflected

The Retina:

Strangely, **rods** and **cones** are at the **back** of the retina, behind a mostly-transparent neural structure that collects their response.

The center of the retina is a densely packed region called the **fovea**.

Cones much denser here than the **periphery**

Photopic:

Light intensities that are bright enough to stimulate the cone receptors and bright enough to "saturate" the rod receptors(Sunlight and bright indoor lighting are both photopic lighting conditions)

Scotopic:

Light intensities that are bright enough to stimulate the rod receptors but too dim to stimulate the cone receptors(Moonlight and extremely dim indoor lighting are both scotopic lighting conditions)

Rods are sensitive to scotopic light levels

All rods contain the same photopigment molecule: Rhodopsin

All rods have the **same sensitivity** to various wavelengths of light

Therefore, rods suffer from the problem of univariance and **cannot sense differences in color**

Under scotopic conditions, only rods are active, which is why the world seems drained of color

Cone photoreceptors: Three varieties:

L-cones: Cones that are preferentially sensitive to **long** wavelengths (**red cones**)

M-cones: Cones that are preferentially sensitive to **middle** wavelengths (**green cones**)

S-cones: Cones that are preferentially sensitive to **short** wavelengths (**blue cones**)

Artists often specify color as tints, shades, and tones of saturated (pure) pigments

Tint: Gotten by adding white to a pure pigment, decreasing saturation

Shade: Gotten by adding black to a pure pigment, decreasing lightness

Tone: Gotten by adding white and black to a pure pigment

Chapter 8: Color Spaces

Color space: A three-dimensional space that describes all colors. There are several possible color spaces(**RGB**, **HSB**(defined by hue, saturation, and brightness))

Taking linear combinations of R, G and B defines the RGB color space

the range of perceptible colors generated by adding some part each of R, G and B

If R, G and B correspond to a monitor's phosphors (monitor RGB), then the space is the range of colors displayable on the monitor

Chapter 9: Light Source

Modeling Light Sources $I_L(x, y, z, q, f, \lambda)$...
describes the intensity of energy, leaving a light source, ...
arriving at location(x,y,z), ...
from direction (q,f), ...
with wavelength λ

OpenGL Light Source Models
Simple mathematical models: Point, Directional, Spot.

Point Light Sources: A point light source emits light equally in all directions from a single point. The direction to the light from a point on a surface thus differs for different points. So we need to calculate a normalized vector to the light source for every point we light:

$$\bar{d} = \frac{\bar{p} - \bar{l}}{||\bar{p} - \bar{l}||}$$

Directional Light Sources: For a directional light source (e.g., sun) we make simplifying assumptions
Direction is constant for all surfaces in the scene
All rays of light from the source are **parallel**
The direction from a surface to the light source is important in lighting the surface
Models point light source at infinity (e.g., sun)[intensity $I_L = I_0$(No attenuation with distance), direction (dx, dy, dz)]

Spot Light Sources: Spot lights are point sources whose intensity falls off directionally
Requires color, point direction, falloff parameters
Supported by OpenGL
Other Light Sources: Area light sources define a 2-D emissive surface (usually a disc or polygon)

Chapter 9: Ambient Light

Direct (Local) Illumination
Emission at light sources
Scattering at surfaces
Global illumination
Shadows, Refractions, Inter-object reflections.
Ambient Term
Represents reflection of all indirect illumination
Ambient Light

Objects not directly lit are typically still visible(e.g., the ceiling in this room, undersides of desks)

This is the result of indirect illumination from emitters, bouncing off intermediate surfaces

Too expensive to calculate (in real time), so we use a hack called an ambient light source

No spatial or directional characteristics; illuminates all surfaces equally

Amount reflected depends on surface properties

Chapter 9: Surface Reflection

Surface reflectance:

$R_s(\theta, \phi, \gamma, \pi, \lambda)$...

describes the amount of incident energy,

arriving from direction (θ, ϕ)...

leaving in direction (γ, π), ...

with wavelength λ

Diffuse reflectance: An ideal diffuse reflector, at the microscopic level, is a rough surface (real-world example: chalk), Because of these microscopic variations, an incoming ray of light is equally likely to be reflected in any direction over the hemisphere.

What does the reflected intensity depend on? How much light is reflected?

Depends on angle of incident light

Depends on angle of incident light

Lambert's Law

$$dL = dA\cos(\theta)$$

Ideal diffuse surfaces reflect according to Lambert's cosine law: The energy reflected by a small portion of a surface from a light source in a given direction is proportional to the cosine of the angle between that direction and the surface normal. Note that the reflected intensity is independent of the viewing direction, but does depend on the surface orientation with regard to the light source

Specular reflection

Shiny surfaces exhibit specular reflection(eg. Polished metal, Glossy car finish)

A light shining on a specular surface causes a **bright spot** known as a specular highlight

Where these highlights appear is a function of the **viewer's position**, so specular reflectance is **view dependent**

Specular Reflection follows **Snell's Laws**

$$\theta_{(l)ight} = \theta_{(r)eflect}$$

Chapter 9: Phong Lighting Model

Non-Ideal Specular Reflectance: An Empirical Approximation

Snell's law applies to **perfect mirror-like surfaces**, but aside from mirrors, few surfaces exhibit perfect specularity. How can we capture the "softer" reflections of surface that are glossy rather than mirror-like?

Hypothesis: most light reflects according to Snell's Law But because of microscopic surface variations, some light may be reflected in a direction slightly off the ideal reflected ray. As we move from the ideal reflected ray, some light is still reflected.

Phong lighting model: The most common lighting model in computer graphics was suggested by **Phong:**

$$I_{specular} = k_s I_{light}(\cos(\phi))^{n_{shiny}}$$

k_s: Material surface reflectance
I_{light}: Intensity of the light source
θ: The angle between the ideal reflectance direction and viewer
n_{shiny}: purely empirical constant that varies the rate of falloff

Though this model has no physical basis, it works (sort of) in practice
The cos term of Phong lighting can be computed using vector arithmetic:

$$I_{specular} = k_s I_{light}(\cos(\bar{v} \cdot \bar{r}))^{n_{shiny}}$$

v is the unit vector towards the viewer
r is the ideal reflectance direction

The Final Combined Equation

Single light source:

$$I = I_E + K_A I_{AL} + K_D(N \cdot L)I_L + K_s(V \cdot R)^n I_L$$

Multiple light sources:

$$I = I_E + K_A I_{AL} + \sum_i (K_D(N \cdot L_i)I_i + K_s(V \cdot R_i)^n I_i)$$

(I = emission + ambient + diffuse + specular)

OpenGL Function, Light properties

void glMaterialfv(GLenum face, GLenum pname, const GLfloat * params);

face: Specifies which face or faces are being updated. Must be one of GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK

pname: Specifies the material parameter of the face or faces that is being updated. Must be one of GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, GL_AMBIENT_AND_DIFFUSE, or GL_COLOR_INDEXES.

Params: parameter values.

void glNormalfv(normal[]);

void glLightfv(GLenum light, GLenum pname, GLfloat *params);
light: Number ID of light
pname: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION
GL_SPOT_DIRECTION, GL_SPOT_EXPONENT
GL_SPOT_CUTOFF
GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION

Chapter 10: Global Illumination

Global illumination

Shadows, Refractions, Inter-object reflections.

The notion that a point is illuminated by more than light from local lights; it is illuminated by all the emitters and reflectors in the global scene

Ray Tracing

Radiosity

Local vs. Global Illumination

- Local illumination: Phong model (OpenGL)
  - Light to surface to viewer
  - No shadows, inter-reflections
  - Fast enough for interactive graphics
- Global illumination: Ray tracing
  - Multiple specular reflections and transmissions
  - Only one step of diffuse reflection
- Global illumination: Radiosity
  - All diffuse interreflections; shadows

Image vs. Object Space

- Image space: Ray tracing
  - Trace backwards from viewer
  - View-dependent calculation
  - Result: rasterized image (pixel by pixel)
- Object space: Radiosity
  - Assume only diffuse-diffuse interactions
  - View-independent calculation
  - Result: 3D model, color for each surface patch
  - Can render with OpenGL

Chapter 10: Algorithm of Ray Tracing

Idea of ray tracing: Inversely tracing the process of light being reflected and refracted multiple time and finally casted to the eye.

Ray tree of a ray
Recursion termination:

Condition 1: The ray does not intersect with any object, or intersects with pure diffusion plane

Condition 2: The contribution of the ray is small enough

Condition 3: The recursive depth reaches maximum

Algorithm 7 Algorithm of Ray Tracing

1 **for** each pixel p in the image **do**
2 Shoot a ray R from viewpoint to pixel p
3 Compute all intersections between R and the scene, and find the visible one, P
4 Compute the color lc of P using Phong model
5 Cast rays from the directions of reflection and refraction from P(The surface is opaque, stop)
6 Recursive compute Ir and It(contribution from the environment)
7 $p = lc + lr + lt$
8 **end for**

Pseudo Code

Recursive Ray Tracing

Computing all shadow and feeler rays is slow

- Stop after fixed number of iterations
- Stop when energy contributed is below threshold

Most work is spent testing ray/plane intersections

- Use bounding boxes to reduce comparisons
- Use bounding volumes to group objects
- Parallel computation (on shared-memory machines)

Summary

Ray casting (direct Illumination)

- Usually use simple analytic approximations for light source emission and surface reflectance

Recursive ray tracing (global illumination)

- Incorporate shadows, mirror reflections, and pure refractions

All of this is an approximation so that it is practical to compute

Radiosity

- Ray tracing models specular reflection and refractive transparency, but still uses an **ambient term** to account for other lighting effects
- Radiosity is the rate at which **energy is emitted or reflected by a surface**
- By conserving light energy in a volume, these radiosity effects can be traced

All surfaces are assumed perfectly diffuse

- What does that mean about property of lighting in scene?

– Light is reflected equally in all directions

Diffuse-diffuse surface lighting effects possible

Basic Idea

• We can accurately model diffuse reflections from a surface by considering the radiant energy transfers between surfaces, subject to conservation of energy laws.

• Divide surfaces into patches (elements)

• Model light transfer between patches as system of linear equations

Ray Tracing vs. Radiosity

Radiosity captures the sum of light transfer well

• But it models all surfaces as diffuse reflectors

• Can't model specular reflections or refraction

– Images are viewpoint independent

Ray tracing captures the complex behavior of light rays as they reflect and refract

• Works best with specular surfaces.

– Diffuse surface converts light ray into many. Ray tracing follows one ray and does not capture the full effect of the diffusion.

– Must use ambient term to replace absent diffusion

Chapter 11: Three Shading Models

Flat Shading

The simplest approach, flat shading, **calculates illumination at a single point for each polygon**

If an object really is faceted, is this accurate? Is flat shading realistic for faceted object?

No:

• For point sources, the direction to light varies across the facet

– For specular reflectance, direction to eye varies across the facet

Vertex Normals

To get smoother-looking surfaces,we introduce vertex normals at each vertex

• Usually different from facet normal

• Used **only** for shading

• Think of as a better approximation of the **real** surface that the polygons approximate

Vertex normals may be

• Provided with the model

• Approximated by averaging the normals of the facets that share the vertex

Gouraud Shading

This is the most common approach

• Perform Phong lighting at the vertices

• Linearly interpolate the resulting colors over faces

– Along edges

– Along scanlines

Does this eliminate the facets?

Artifacts

• Often appears dull, chalky

• Lacks accurate specular component

– If included, will be averaged over entire polygon

Phong Shading

Phong shading is not the same as Phong lighting, though they are sometimes mixed up

• **Phong lighting**: the empirical model we've been discussing to calculate illumination at a point on a surface

• **Phong shading**: linearly interpolating the surface normal across the facet, **applying the Phong lighting model at every pixel**

– Same input as Gouraud shading

– Usually very smooth-looking results

– But, considerably more calculation

Interpolate the vertex normals over the surface

• Normal of each edge point is linearly interpolated by that of two endpoints

• normal of interior points is linearly interpolate by the normals of the triangle vertices

Shading Models

Flat Shading

• Compute Phong lighting once for entire polygon

Gouraud Shading

• Compute Phong lighting at the vertices and interpolate lighting values across polygon

Phong Shading

• Interpolate normals across polygon and perform Phong lighting across polygon

Chapter 11: Morphing

What is morphing?

• Combination of warping and blending

– warp = image distortion

Map image to a coke can

Ripple effect

– blend = cross dissolve

Film cut effect

Ways to morph

3D Techniques

• Interpolate between corresponding vertices

– Models must align somehow

– Polygon transformations may be difficult

2D Techniques

• Cross-dissolve

–Difficult to align

•Pixelize the images and move the "tiles"

–Tile paths must be determined somehow

Beier-Neely Morphing

Transformation with One Pair of Lines

How would start image morph?

Build Common Coordinate System

Find projection

Find projection on perpendicular

Sample color from initial image

Repeat for all pixels in image

Now do the same thing for the final image

Now do a 50/50 blend of two warped images

Blends between Two Images

• We define corresponding lines in I 0 and I 1 .

• Each intermediate frame I of the metamorphosis is defined by creating a new set of line segments by interpolating the lines from their positions in I_0 to the positions in I_1 .

• Both images I_0 and I_1 are distorted toward the position of the lines in I. These two resulting images are cross-dissolved throughout the metamorphosis