

Computer Animation Guest Lecture

Reflection Models

Definitions

Surface Normal - points away from the surface

View Vector - vector from the surface point to the viewer

Light Vector - vector from the surface point to a/the light source (might be a global direction)

If you know these three vectors, then you know enough about the illumination geometry to create a realistic reflection of the local illumination.

Modeling Illumination Sources

Ambient light

- Ambient light is everywhere
- It is a simple attempt to model diffuse interreflection

Directional light source

- Assumption is that the light source is very far away
- The L direction vector does not change across the surface

Point light sources

- Intensity
- Color
- Position
- Have to explicitly calculate L each time you make an appearance determination

Spot light sources

- A point light source with limitations
- Specify a cone using an angle (store the cosine of that angle)
- Specify a spot direction which provides the center of the cone
- When you make an appearance determination
 - Calculate the light source to surface direction (-L)
 - Take the dot product with the spot direction
 - If result is $< \cos$ of spot angle, the light source is not visible at that surface
- To create penumbra effects, have the intensity fall off as $\cos^n$ instead of making a hard decision at the edge of the light cone.
- Area light sources
 - More difficult to compute
 - Need to sample the light source area from the surface (ray casting)
 - Calculate multiple L vectors using random points on the light source surface
 - Average the light source intensity values
 - Use the average values to make the appearance determination

Modeling Reflection

We know the intensity and color of the light falling on a surface

We can calculate the L vector(s)

What does the surface look like?

- Surface color & intensity
 - Specify as an RGB tuple (0..1, or 0..255)
- Surface geometry
 - Surface normal
 - View vector

Inhomogeneous Dielectrics: paint, plastic, paper, ceramics, cloth, ...

- Any material that is a substrate with pigment particles
- Most light is absorbed into the surface and re-emitted as the body color of the object
- Some light reflects off the surface and is not modified (in color) by the surface

Metals/Conductors

- Light that penetrates the surface is absorbed by the material and re-emitted as black-body radiation (typically not in the visible range)
- Most light is reflected at the surface (up to 80%)
- Surface reflection of metals can modify the reflected color (gold, copper, bronze, etc)

Ambient reflection

- Helps to assume that there is some light just bouncing around the world
- No particular direction or source
- Constant value across the surface
- Define ambient illumination $L_a = (R_a, G_a, B_a)$
- Define ambient surface $C_a = (R_{ca}, G_{ca}, B_{ca})$
- Define ambient reflection $I_a = (R_a * R_{ca}, G_a * G_{ca}, B_a * B_{ca})$
- Let the elements of L_a and C_a be in the range [0..1]

Body reflection

- Light is absorbed by the material
- Light is re-emitted at specific wavelengths (gives color of the material)
- Light is re-emitted in random directions
- Intensity of re-emitted light is dependent upon the amount of energy incident on the surface
- Energy is dependent upon the angle of incidence
- Perfect diffuser assumption makes a Lambertian surface
 - Appearance of the surface is only dependent upon the angle of incidence
 - $I = LC_d \cos \theta$
 - $R = L_r C_{dr} \cos \theta$, and so on for each color band
 - L and C_d need to be in the range [0..1] for this representation to work correctly

Surface reflection

- Light is reflected at the surface
- Illumination color does not change for most man-made materials
- Reflection is a spike that depends upon the viewing direction as well as the angle of incidence

- Look at angle (ϕ) between perfect specular direction (R) and the surface normal (N)
- Phong model: $I_s = LC_s \cos^n(\phi)$
- n is the specularity coefficient
- Perfect reflection direction: $R = (2N \cdot L)N - L$
 - When the view vector $V = R$ the viewer is at the maximum reflection direction
- Can approximate R with the halfway vector: $H = (L + V) / |L + V|$
 - When $H = N$ then the viewer is at the maximum reflection direction

Overall lighting equations:

$$I = L_a C_a + LC_d(L \cdot N) + LC_s(H \cdot N)^n$$

Multiple light sources: $I = L_a C_a + \sum_L L_i [C_d(L \cdot N) + C_s(H \cdot N)^n]$

Distance attenuation: $f(d) = \min\left(1, \frac{1}{a_0 + a_1 d + a_2 d^2}\right)$

Multiple light sources with attenuation: $I = L_a C_a + \sum_L f(d) L_i [C_d(L \cdot N) + C_s(H \cdot N)^n]$

Real Surface Reflection

Microfacet model: Cook & Torrance

$$R_s = \frac{F}{\pi} \frac{DG}{(N \cdot L)(N \cdot V)}$$

F = Fresnel Reflectance

D = distribution of normals (slopes) = $ce^{-\left(\frac{\alpha}{m}\right)^2}$

G = geometric attenuation factor = $\min\left(1, \frac{2(N \cdot H)(N \cdot V)}{(V \cdot H)}, \frac{2(N \cdot H)(N \cdot L)}{(V \cdot H)}\right)$

N = macro surface normal

L = light source direction

V = viewer direction

H = angular bisector of V and L = $\frac{V + L}{2}$

Hidden Surface Removal

Backface Culling

Use the polygon equation and the definition of “out” to cull back-facing polygons.

Take dot product of view vector and polygon surface normal

- If product is positive, draw it
- else, ignore it
- Works perfectly for unoccluded convex objects

Backface culling works well as a precursor step to other hidden-surface removal algorithms because it reduces the number of polygons you have to deal with.

Can use the same concept when calculating a surface's appearance, because if the dot product of the light vector and the surface normal is negative, the light is not shining on that surface.

Painter's Algorithm

Oil painters draw the background first, then the stuff in front.

Concept: sort polygons by depth and draw them in that order

- for polygons that don't overlap in depth, this is easy
- for polygons that overlap in depth there are special cases that don't require splitting
- for polygons that intersect, divide them along the intersection line
- Alternatively, recursively divide crossing polygons in half by depth until things are ok

Works well for quick and dirty pictures of scenes

- Cull out the polygons that don't face the viewer
- Calculate color for each polygon
- Transform the vertices & clip them to the screen
- Draw the polygons in order from back to front after sorting by depth

Scan conversion of a single polygon

Calculate depth of each vertex of the polygon

If we assume orthographic projection, then depth values vary linearly over the polygon in x and y

- Ok assumption with perspective projection if $dz \ll \text{distance to camera}$
- Otherwise, you want to do some additional math to get the interpolation correct

Calculate the dz/scanline value when you generate each edge

- Store the zIntersect and dzperscanline with each edge structure
- Also store yStart, yEnd, xIntersect, and dxperscanline

Start at the top of the polygon and go scanline by scanline down the image

- Interpolate depth from left to right across the polygon for each row
- Increment zIntersect using dzperscanline after each row
- Increment xIntersect using dxperscanline after each row
- Stop when you get to yEnd

Z-buffer Algorithm

Create a depth buffer in addition to an image buffer

- Initialize the depth buffer to the maximum distance
- Initialize the image buffer to the background color

Clip polygons to the screen

Scan-convert polygons individually

- Calculate z-values for each pixel using the depth-conversion algorithm

- For each pixel
 - If the new depth value is closer than the current value in the buffer
 - Draw the pixel color in the image buffer
 - Update the depth buffer with the new value
 - Else
 - Don't do anything, it's behind another polygon

This is fast, but it doesn't do transparency or shadows

- You can do transparency if you store pointers to polygons at each pixel in the Z-buffer instead of just depth values (A-buffer method)

Shading Methods

Flat Shading

- Make the calculation of the color of a polygon once
- Use the surface normal for the defining plane (appropriate direction)
- Use the average position of the vertices as the point to do the calculations

Graud Shading

- Make the calculation of the color of a polygon at each vertex
 - Define a surface normal at each vertex
 - Define a color at each vertex
- Interpolate colors across the polygon
 - Add a dcPerScan and cIntersect to your Edge record in the scanline fill algorithm
- OpenGL does this, most games do this
 - Minimizes the number of shading calculations
 - Gives you reasonable shading effects
 - Can approximate smooth surfaces with polygons
- Graud shading is subject to Mach banding
 - Discontinuous first derivatives cause Mach banding
 - Our eyes "overshoot" at the edges and we perceive lines

Phong Shading

- Make the calculation of color/intensity at each point on a polygon
 - Need to have positions
 - Interpolate position as you move across the polygon surface
 - dpPerScan, pIntersect (vectors)
 - Need to have surface normals
 - Define surface normals at vertices
- Interpolate surface normals across the polygon surface
 - dnPerScan, nIntersect (vectors)
- Use the interpolated values to calculate the shading/color at each point on the surface
- Better graphics systems do this
- Phong shading reduces or eliminates Mach banding

Implementation with a Z-buffer algorithm

Flat shading

- Easy to implement
- Calculate the color when you create the polygon
- No need to change your z-buffer scanline algorithm

Graud shading

- Easy to implement
- Calculate the colors at the vertices when you create the polygon
- Modify your z-buffer scanline algorithm
 - Takes an array of colors as well as an array of vertices
 - Add cIntersect and dcPerScan to your edge record
 - Modify MakeEdgeRec to calculate the cIntersect and dcPerScan values
 - Modify FillScan so that it interpolates the colors across a span

Phong Shading

- Moderately easy to implement
- Need to significantly modify the z-buffer algorithm
 - Pass in surface normal for each vertex
 - Pass in a true CVV coordinate for each vertex (not the screen + depth as above)
 - Pass in a color for the polygon (or a color for each vertex)
 - Modify MakeEdgeRec to set up the edge record
 - Set up true position interpolation
 - Set up surface normal interpolation
 - Modify FillScan to interpolate positions and normals
 - For loop interpolates positions and normals
 - Calculate shading at each position
- Speedups
 - Use directional lighting and parallel projection (viewer at infinity)
 - don't have to recalculate the light vector
 - don't need to know the true position of the point
 - Calculate shading at every nth pixel
 - Interpolate the ones in between unless adjacent values are fairly different
 - This catches most highlights but saves a lot of computation

Shadows in a Z-buffer Rendering Algorithm

One thing you can do is create shadow polygons or shadow volumes by intersecting all of the polygons with each light source.

- Shadow volumes are a common solution in OpenGL type applications
- Shadow volumes are made up of shadow polygons that turn light sources on or off
- Render them just like regular polygons, but just flip a flag for each light source

Global Illumination Models

The basic problem with z-buffer methods is that you don't get shadows or any kind of interreflection between surfaces (which makes up a big part of the world of light).

Ray Tracing: send a ray through each pixel and bounce it around the scene

Radiosity: solve for the mutual reflection from each surface to every other surface (big matrix)