Hidden Surfaces

Clarification: Blinn-Phong Model

- only change vs Phong model is to have the specular calculation to use \((h \cdot n)\) instead of \((v \cdot r)\)

- full Blinn-Phong lighting model equation has ambient, diffuse, specular terms

\[
I_{total} = k_a I_{ambient} + \sum_{i=1}^{\text{#lights}} I_i \left( k_d (n \cdot l_i) + k_s (n \cdot h_i)^{n_{shiny}} \right)
\]

- just like full Phong model equation

\[
I_{total} = k_a I_{ambient} + \sum_{i=1}^{\text{#lights}} I_i \left( k_d (n \cdot l_i) + k_s (v \cdot r_i)^{n_{shiny}} \right)
\]
Reading for Hidden Surfaces

• FCG Sect 8.2.3 Z-Buffer
• FCG Sect 12.4 BSP Trees
  • (8.1, 8.2 2nd ed)
• FCG Sect 3.4 Alpha Compositing
  • (N/A 2nd ed)
Hidden Surface Removal
Occlusion

- for most interesting scenes, some polygons overlap

- to render the correct image, we need to determine which polygons occlude which
Painter’s Algorithm

- simple: render the polygons from back to front, “painting over” previous polygons
- draw blue, then green, then orange
- will this work in the general case?
Painter’s Algorithm: Problems

- *intersecting polygons* present a problem
- even non-intersecting polygons can form a cycle with no valid visibility order:
Analytic Visibility Algorithms

- early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:
Analytic Visibility Algorithms

- what is the minimum worst-case cost of computing the fragments for a scene composed of $n$ polygons?
- answer: $O(n^2)$
Analytic Visibility Algorithms

• so, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
• we’ll talk about one:
  • *Binary Space Partition (BSP) Trees*
Binary Space Partition Trees (1979)

- BSP Tree: partition space with binary tree of planes
  - idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
  - preprocessing: create binary tree of planes
  - runtime: correctly traversing this tree enumerates objects from back to front
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Splitting Objects

• no bunnies were harmed in previous example
• but what if a splitting plane passes through an object?
  • split the object; give half to each node
Traversing BSP Trees

• tree creation independent of viewpoint
  • preprocessing step
• tree traversal uses viewpoint
  • runtime, happens for many different viewpoints
• each plane divides world into near and far
  • for given viewpoint, decide which side is near and which is far
    • check which side of plane viewpoint is on independently for each tree vertex
    • tree traversal differs depending on viewpoint!
• recursive algorithm
  • recurse on far side
  • draw object
  • recurse on near side
Traversing BSP Trees

query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front:

renderBSP(BSPtree *T)
BSPtree *near, *far;
if (eye on left side of T->plane)
    near = T->left; far = T->right;
else
    near = T->right; far = T->left;
renderBSP(far);
if (T is a leaf node)
    renderObject(T)
renderBSP(near);
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees : Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
BSP Trees : Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees: Viewpoint B
BSP Trees: Viewpoint B
BSP Tree Traversal: Polygons

• split along the plane defined by any polygon from scene
• classify all polygons into positive or negative half-space of the plane
  • if a polygon intersects plane, split polygon into two and classify them both
• recurse down the negative half-space
• recurse down the positive half-space
BSP Demo

• useful demo:

http://symbolcraft.com/graphics/bsp
BSP Demo

• order of insertion can affect half-plane extent
Summary: BSP Trees

- **pros:**
  - simple, elegant scheme
  - correct version of painter’s algorithm back-to-front rendering approach
  - was very popular for video games (but getting less so)

- **cons:**
  - slow to construct tree: $O(n \log n)$ to split, sort
  - splitting increases polygon count: $O(n^2)$ worst-case
  - computationally intense preprocessing stage restricts algorithm to static scenes
The Z-Buffer Algorithm (mid-70’s)

• BSP trees proposed when memory was expensive
  • first 512x512 framebuffer was >$50,000!
• Ed Catmull proposed a radical new approach called z-buffering
• the big idea:
  • resolve visibility independently at each pixel
The Z-Buffer Algorithm

- we know how to rasterize polygons into an image discretized into pixels:
The Z-Buffer Algorithm

- what happens if multiple primitives occupy the same pixel on the screen?
  - which is allowed to paint the pixel?
The Z-Buffer Algorithm

- idea: retain depth after projection transform
  - each vertex maintains z coordinate
    - relative to eye point
  - can do this with canonical viewing volumes
The Z-Buffer Algorithm

• augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  • at frame beginning, initialize all pixel depths to ∞
  • when rasterizing, interpolate depth (Z) across polygon
• check Z-buffer before storing pixel color in framebuffer and storing depth in Z-buffer
• don’t write pixel if its Z value is more distant than the Z value already stored there
Interpolating Z

• barycentric coordinates
  • interpolate Z like other planar parameters
Z-Buffer

• store (r,g,b,z) for each pixel
• typically 8+8+8+24 bits, can be more

```plaintext
for all i,j {
   Depth[i,j] = MAX_DEPTH
   Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
   for all pixels in P {
      if (Z_pixel < Depth[i,j]) {
         Image[i,j] = C_pixel
         Depth[i,j] = Z_pixel
      }
   }
}
```
Depth Test Precision

• reminder: perspective transformation maps eye-space (view) \( z \) to NDC \( z \)

\[
\begin{bmatrix}
E & 0 & A & 0 \\
0 & F & B & 0 \\
0 & 0 & C & D \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
= \begin{bmatrix}
Ex + Az \\
Fy + Bz \\
Cz + D \\
-1
\end{bmatrix}
= \begin{bmatrix}
-\left( \frac{Ex}{z} + Az \right) \\
-\left( \frac{Fy}{z} + Bz \right) \\
-\left( C + \frac{D}{z} \right) \\
1
\end{bmatrix}
\]

• thus: 
\[
z_{NDC} = -\left( C + \frac{D}{z_{eye}} \right)
\]
Depth Test Precision

- therefore, depth-buffer essentially stores $1/z$, rather than $z$!
- issue with integer depth buffers
  - high precision for near objects
  - low precision for far objects

![Graph showing the relationship between NDC space and eye space with a curve indicating depth precision variations.](image)
Depth Test Precision

- low precision can lead to depth fighting for far objects
  - two different depths in eye space get mapped to same depth in framebuffer
  - which object “wins” depends on drawing order and scan-conversion
- gets worse for larger ratios $f:n$
  - rule of thumb: $f:n < 1000$ for 24 bit depth buffer
- with 16 bits cannot discern millimeter differences in objects at 1 km distance
- demo: sjbaker.org/steve/omniv/love_your_z_buffer.html
More: Integer Depth Buffer

- reminder from picking discussion
  - depth lies in the NDC z range [0,1]
  - format: multiply by $2^n - 1$ then round to nearest int
    - where $n = \text{number of bits in depth buffer}$
- 24 bit depth buffer = $2^{24} = 16,777,216$ possible values
  - small numbers near, large numbers far
- consider depth from VCS: $(1<<<N) \times (a + \frac{b}{z})$
  - $N = \text{number of bits of Z precision}$
  - $a = \frac{z\text{Far}}{(z\text{Far} - z\text{Near})}$
  - $b = \frac{z\text{Far} \times z\text{Near}}{(z\text{Near} - z\text{Far})}$
  - $z = \text{distance from the eye to the object}$
Z-Buffer Algorithm Questions

• how much memory does the Z-buffer use?
• does the image rendered depend on the drawing order?
• does the time to render the image depend on the drawing order?
• how does Z-buffer load scale with visible polygons? with framebuffer resolution?
Z-Buffer Pros

- simple!!!
- easy to implement in hardware
  - hardware support in all graphics cards today
- polygons can be processed in arbitrary order
- easily handles polygon interpenetration
- enables deferred shading
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments
Z-Buffer Cons

- poor for scenes with high depth complexity
  - need to render all polygons, even if most are invisible

- shared edges are handled inconsistently
  - ordering dependent
Z-Buffer Cons

- requires lots of memory
  - (e.g. 1280x1024x32 bits)
- requires fast memory
  - Read-Modify-Write in inner loop
- hard to simulate translucent polygons
  - we throw away color of polygons behind closest one
  - works if polygons ordered back-to-front
    - extra work throws away much of the speed advantage
Hidden Surface Removal

• two kinds of visibility algorithms
  • object space methods
  • image space methods
Object Space Algorithms

- determine visibility on object or polygon level
  - using camera coordinates
- resolution independent
  - explicitly compute visible portions of polygons
- early in pipeline
  - after clipping
- requires depth-sorting
  - painter’s algorithm
  - BSP trees
Image Space Algorithms

- perform visibility test for in screen coordinates
  - limited to resolution of display
  - Z-buffer: check every pixel independently
- performed late in rendering pipeline
Projective Rendering Pipeline

- **OCS** - object coordinate system
- **WCS** - world coordinate system
- **VCS** - viewing coordinate system
- **CCS** - clipping coordinate system
- **NDCS** - normalized device coordinate system
- **DCS** - device coordinate system

**Transformations**
- **modeling transformation**
- **viewing transformation**
- **projection transformation**

**Functions**
- `glVertex3f(x, y, z)`
- `glTranslatef(x, y, z)`
- `glRotatef(th, x, y, z)`
- `gluLookAt(...)`
- `glFrustum(...)`
- `glutInitWindowSize(w, h)`
- `glViewport(x, y, a, b)`

**Coordinate Systems**
- **OCS**
- **WCS**
- **VCS**
- **CCS**
- **NDCS**
- **DCS**
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

object world viewing

OCS WCS VCS CCS

(normalized device)

NDCS

device

DCS (3D)

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

screen

SCS

(2D)
Backface Culling
Back-Face Culling

- on the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:

note: backface culling alone doesn’t solve the hidden-surface problem!
Back-Face Culling

• not rendering back-facing polygons improves performance
  • by how much?
    • reduces by about half the number of polygons to be considered for each pixel
  • optimization when appropriate
Back-Face Culling

- most objects in scene are typically “solid”
- rigorously: orientable closed manifolds
  - orientable: must have two distinct sides
    - cannot self-intersect
    - a sphere is orientable since has two sides, 'inside' and 'outside'.
    - a Mobius strip or a Klein bottle is not orientable
  - closed: cannot “walk” from one side to the other
    - sphere is closed manifold
    - plane is not
Back-Face Culling

• examples of non-manifold objects:
  • a single polygon
  • a terrain or height field
  • polyhedron w/ missing face
  • anything with cracks or holes in boundary
  • one-polygon thick lampshade
Back-face Culling: VCS

first idea:
cull if $N_z < 0$

sometimes misses polygons that should be culled
Back-face Culling: NDCS

VCS

NDCS

eye

works to cull if $N_z > 0$
Invisible Primitives

• why might a polygon be invisible?
  • polygon outside the field of view / frustum
    • solved by clipping
  • polygon is back-facing
    • solved by backface culling
  • polygon is occluded by object(s) nearer the viewpoint
    • solved by hidden surface removal
Blending
Rendering Pipeline
Alpha and Premultiplication

• specify opacity with alpha channel $\alpha$
  • $\alpha=1$: opaque, $\alpha=.5$: translucent, $\alpha=0$: transparent
• how to express a pixel is half covered by a red object?
  • obvious way: store color independent from transparency (r,g,b,\(\alpha\))
    • intuition: alpha as transparent colored glass
      • 100% transparency can be represented with many different RGB values
    • pixel value is (1,0,0,.5)
  • upside: easy to change opacity of image, very intuitive
  • downside: compositing calculations are more difficult - not associative
• elegant way: premultiply by $\alpha$ so store ($\alpha r$, $\alpha g$, $\alpha b$, $\alpha$)
  • intuition: alpha as screen/mesh
    • RGB specifies how much color object contributes to scene
    • alpha specifies how much object obscures whatever is behind it (coverage)
    • alpha of .5 means half the pixel is covered by the color, half completely transparent
    • only one 4-tuple represents 100% transparency: (0,0,0,0)
  • pixel value is (.5, 0, 0, .5)
  • upside: compositing calculations easy (& additive blending for glowing!)
  • downside: less intuitive
Alpha and Simple Compositing

- F is foreground, B is background, F over B
- premultiply math: uniform for each component, simple, linear
  - \[ R' = R_F + (1-A_F)*R_B \]
  - \[ G' = G_F + (1-A_F)*G_B \]
  - \[ B' = B_F + (1-A_F)*B_B \]
  - \[ A' = A_F + (1-A_F)*A_B \]
  - associative: easy to chain together multiple operations
- non-premultiply math: trickier
  - \[ R' = (R_F*A_F + (1-A_F)*R_B*A_B)/A' \]
  - \[ G' = (G_F*A_F + (1-A_F)*G_B*A_B)/A' \]
  - \[ B' = (B_F*A_F + (1-A_F)*B_B*A_B)/A' \]
  - \[ A' = A_F + (1-A_F)*A_B \]
  - don't need divide if F or B is opaque. but still... oof!
  - chaining difficult, must avoid double-counting with intermediate ops
Alpha and Complex Compositing

- foreground color $A$, background color $B$
- how might you combine multiple elements?
  - Compositing Digital Images, Porter and Duff, Siggraph '84
  - pre-multiplied alpha allows all cases to be handled simply
Alpha Examples

• blend white and clear equally (50% each)
  • white is (1,1,1,1), clear is (0,0,0,0), black is (0,0,0,1)
  • premultiplied: multiply componentwise by 50% and just add together
  • (.5, .5, .5, .5) is indeed half-transparent white in premultiply format
    • 4-tuple would mean half-transparent grey in non-premultiply format
• premultiply allows both conventional blend and additive blend
  • alpha 0 and RGB nonzero: glowing/luminescent
  • (nice for particle systems, stay tuned)
• for more: see nice writeup from Alvy Ray Smith
  • technical academy award for Smith, Catmull, Porter, Duff
  • http://www.alvyray.com/Awards/AwardsAcademy96.htm