Hidden Surfaces II

Week 9, Mon Mar 15

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2010
News

• yes, I'm granting the request for course marking scheme change
  • old scheme: midterm 20%, final 25%
    • 45% of grade is exam marks
    • argument: midterm is 50 minutes, final is 150 minutes, so want 25/75% division vs 45/55%
  • new scheme: midterm 12%, final 33%
• we'll check - if you would get a better grade in course with old scheme, we'll use that instead
Correction: P1 Hall of Fame: Winner

Sung-Hoon (Nick) Kim
Further 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

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

• just like full Phong model equation

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

- FCG Sect 8.2.3 Z-Buffer
- FCG Sect 12.4 BSP Trees
- (8.1, 8.2 2nd ed)
Review: Cohen-Sutherland Line Clipping

- outcodes
  - 4 flags encoding position of a point relative to top, bottom, left, and right boundary

- \(\text{OC}(p_1) == 0 \land \text{OC}(p_2) == 0\)
  - trivial accept

- \((\text{OC}(p_1) \land \text{OC}(p_2)) \neq 0\)
  - trivial reject
Review: Polygon Clipping

• not just clipping all boundary lines
• may have to introduce new line segments
Review: Sutherland-Hodgeman Clipping

- for each viewport edge
  - clip the polygon against the edge equation for new vertex list
  - after doing all edges, the polygon is fully clipped

- for each polygon vertex
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?
Review: Sutherland-Hodgeman Clipping

- edge from $p[i-1]$ to $p[i]$ has four cases
  - decide what to add to output vertex list
Review: Painter’s Algorithm

- draw objects from back to front
- problems: no valid visibility order for
  - intersecting polygons
  - cycles of non-intersecting polygons possible
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

Diagram showing the structure of a BSP tree with viewpoints and paths indicated by arrows.
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:
  
  [link](http://symbolcraft.com/graphics/bsp)
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
Clarification: 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 \( \infty \)
  - 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

```c
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 $z$

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

• thus: \( z_{NDC} = -\left(C + \frac{D}{z_{eye}}\right) \)
Correction: Ortho Camera Projection

- camera’s back plane parallel to lens
- infinite focal length
- no perspective convergence
- just throw away z values
- x and y coordinates do not change with respect to z in this projection

\[
\begin{bmatrix}
  x_p \\
  y_p \\
  z_p \\
  1
\end{bmatrix}
= \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

\[
P' = \begin{bmatrix}
  \frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
  0 & \frac{2}{\text{top} - \text{bot}} & 0 & -\frac{\text{top} + \text{bot}}{\text{top} - \text{bot}} \\
  0 & 0 & \frac{-2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
  0 & 0 & 0 & 1
\end{bmatrix}
P
\]
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
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
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

modeling transformation
viewing transformation
projection transformation

<table>
<thead>
<tr>
<th>OCS</th>
<th>world</th>
<th>VCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>glVertex3f(x,y,z)</td>
<td>glVertex3f(x,y,z)</td>
<td>glVertex3f(x,y,z)</td>
</tr>
<tr>
<td>glTranslatef(x,y,z)</td>
<td>glTranslatef(x,y,z)</td>
<td>glTranslatef(x,y,z)</td>
</tr>
<tr>
<td>glRotatef(th,x,y,z)</td>
<td>glRotatef(th,x,y,z)</td>
<td>glRotatef(th,x,y,z)</td>
</tr>
</tbody>
</table>

gluLookAt(...) / w

glFrustum(...) alter w

glutInitWindowSize(w,h)

glViewport(x,y,a,b)

perspective division

normalized device NDCS

device DCS
Rendering Pipeline

Geometry Database

Model/View Transform.

Lighting

Perspective Transform.

Clipping

Frame-buffer

Scan Conversion

Texturing

Depth Test

Blending

object world viewing

OCS WCS VCS

normalized device

NDCS

device

DCS

screen

SCS

(2D)

(3D)

(4D)

/clw