Chapter 10
Hidden Surface Removal/Depth Test

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 cyan, then green, then red
- Will this work in general?

Painter's Algorithm: Problems

- Intersecting polygons present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

Hidden Surface Removal

- Object Space Methods:
  - Work in 3D before scan conversion
    - E.g. Painter's algorithm
  - Usually independent of resolution
    - Important to maintain independence of output device (screen/printer etc.)
- Image Space Methods:
  - Work on per-pixel/per fragment basis after scan conversion
  - Z-Buffer/Depth Buffer
  - Much faster, but resolution dependent

Copyright Alla Sheffer
UBC 2012
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
    - To compute z per pixel use barycentric coordinates (next week)

The Z-Buffer Algorithm
- Augment color framebuffer with Z-buffer
  - Also called depth buffer
  - Stores z value at each pixel
  - At frame beginning, initialize all pixel depths to \( \infty \) (depth = far)
- When scan converting: 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
- Use barycentric coordinates
  - Interpolate z like other parameters
    - E.g. color
    - Use one of three formulas shown
      - Plane/edge walk/barycentric

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
    }
  }
}
```

The Z-Buffer Algorithm (mid-70’s)
- History:
  - Object space algorithms were proposed when memory was expensive
  - First 512x512 framebuffer was >$50,000!
- Radical new approach at the time
  - The big idea:
    - Resolve visibility independently at each pixel
**Depth Test Precision**

- Reminder: projective transformation maps eye-space z to generic z-range (NDC).
- Simple example:
  \[
  \begin{bmatrix}
    x \\
    y \\
    z \\
    1
  \end{bmatrix}
  =
  \begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & z \\
    0 & 0 & -1 & 1
  \end{bmatrix}
  \begin{bmatrix}
    x' \\
    y' \\
    z'
  \end{bmatrix}
  \]
- Thus:
  \[
  z_{\text{NDC}} = \frac{a \cdot z_{\text{eye}} + b}{z_{\text{eye}}} = \frac{a}{z_{\text{eye}}} + \frac{b}{z_{\text{eye}}}
  \]

Therefore, depth-buffer essentially stores 1/z, rather than z!

**Depth Test Precision**

- 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 cm differences in objects at 1 km distance

**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

**Z-Buffer Cons**

- Poor for scenes with high depth complexity
  - Need to render all polygons, even if most are invisible
  - Shared edges/overlaps handled inconsistently
    - Ordering dependent

---

Copyright Alla Sheffer
UBC 2012
Z-Buffer Cons

- Requires “lots” of memory
  - (e.g. 1280x1024x32 bits)
- Requires fast memory
  - Read-Modify-Write in inner loop
- Hard to simulate transparent 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

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

Object Space Algorithms

- Object Space Algorithms

BSP Trees: Idea

- For any plane (3D) objects on the same side of plane as viewer CANNOT be occluded by objects on other side
- Idea:
  - Recursively split space by planes
  - Traverse resulting tree to establish rendering order
    - Test eye location w.r.t. each plane

Creating BSP Trees: Objects

Creating BSP Trees: Objects

Creating BSP Trees: Objects

Creating BSP Trees: Objects

Copyright Alla Sheffer
UBC 2012
Creating BSP Trees: 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

Splitting Objects

Tree creation independent of viewpoint
- Preprocessing step
- Tree traversal uses viewpoint
- Runtime, happens for many different viewpoints

Traversing BSP Trees

BSP Trees: Viewpoint A

Copyright Alla Sheffer
UBC 2012
BSP Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
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

Copyright Alla Sheffer
UBC 2012
Traversing BSP Trees

```c
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 Demo

- Useful demo: [http://symbolcraft.com/graphics/bsp](http://symbolcraft.com/graphics/bsp)

Summary: BSP Trees

- **Pros:**
  - Simple, elegant scheme
  - Correct version of painter's algorithm back-to-front rendering approach
  - Still very popular for video games
- **Cons:**
  - Slow(ish) 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
Back Face Culling (object space)

- In closed polyhedron you don't see object “back” faces
- Assumption:
  - Normals of faces point out from the object

hidden surfaces problem

Back Face Culling

- Determine back & front faces using sign of inner product $n \cdot v$
  
  $n \cdot v = n_x v_x + n_y v_y + n_z v_z = |n| |v| \cos \theta$

- In a convex object:
  - Invisible back faces
  - All front faces entirely visible $\Rightarrow$ solves hidden surfaces problem

- In non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible