Hidden Surface Removal/Visibility
CPSC 314

The Rendering Pipeline

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.

Painter's Algorithm: Problems
- Intersecting polygons present a problem.
- Even non-intersecting polygons can form a cycle with no valid visibility order.
- Draw cyan, then green, then red.

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/fragment basis after scan conversion
- Z-Buffer/Depth Buffer
- Much faster, but resolution dependent

The Z-Buffer Algorithm
- What happens if multiple primitives occupy the same pixel on the screen?
- Which is allowed to print the pixel?
- Space to depth after projection transform.

The Z-Buffer Algorithm: Ideas
- 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
- Also called depth buffer
- Stores z value at each pixel
- At frame beginning, initialize all pixel depths to
  - 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
Z-Buffer

Store \((r,g,b,z)\) for each pixel
- Typically 32-bit, 24 bits can be more
  - For all \(p\)
  - \(Depth[p] = Max\) Depth
  - \(Depth[p] = Back/Frame\) Colors
- For all polygons \(P\)
  - If \(z\) pixel < \(Depth[p]\)
    - \(Depth[p] = z\) pixel
    - \(Depth[p] = z\) pixel
  - \(z\) color

Interpolating Z

- Just interpolate Z along edges and across spans

Barycentric coordinates
- Interpolate z like other parameters
- E.g. color

The Z-Buffer Algorithm (mid-70's)

History:
- Object space algorithms were proposed when memory was expensive
- First 512x512 transform buffer 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 points to generic z-range (NDC)
- Simplest:
  - \[
    \begin{bmatrix}
      1 & 0 & 0 & z_e \\
      0 & 1 & 0 & z_e \\
      0 & 0 & 1 & z_e \\
      0 & 0 & 0 & 1 
    \end{bmatrix}
  \]
- Thus:
  - \(z_{eye} = w z_{eye} + k = w + k\)

Depth Test Precision

- Therefore, depth buffer essentially stores \(z_e\) rather than \(z\)
- Issue with integer depth buffers
  - High precision for new objects
  - Low precision for far objects
  - Eventual depth buffering

Depth Test Precision

- Depth buffering 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 \(f_e\)
  - Rule of thumb: \(f_e = 1000\) for 24 bit depth buffer
  - With 16 bits cannot discern millimeter 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 and 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 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 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 Visibility Algorithms
What is the minimum worst-case cost of computing the fragments for a scene composed of a polygon?
Answer: \(O(n^2)\)

Object Space 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
  - Still in use today for ray-tracing, and in combination with z-buffer

Binary Space Partition Trees (BSP)
BSP Tree: partition space with binary tree of planes
- Some 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

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
- Programming step
- Tree traversal uses viewpoint
- Routine, 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
  - Other side of plane viewpoint is on independently for each tree vertex
- Tree traversal differs depending on viewpoint!
- Reduces ambiguity
  - Assign near side
  - Over object
  - Traverse near side

Traversing BSP Trees

```
renderBSP(bsptree, T)
BSPtree near, *far
if tree 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

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

BSP Trees: Viewpoint A

BSP Trees: Viewpoint A
**BSP Trees: Viewpoint B**

- 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://sumbirdart.com/practices.html

---

**Summary: BSP Trees**

**Pros:**
- Simple, elegant scheme.
- Correct version of painter's algorithm back-to-front rendering approach.
- Still very popular in video games (but getting less so).

**Cons:**
- Slow to construct tree. O(n log n) to split, sort, and rebuild.
- Splitting increases polygon count. O(n^2) worst case.
- Computationally intense preprocessing stage makes the algorithm inapplicable.

---

**BSP Tree Traversal: Polygons**

- Not rendering back-facing polygons improves performance.
  - 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 one side is inside and the other side is outside.
  - A Möbius strip or a Klein bottle is not orientable.
- **Closed:** surface encloses a volume.
  - Sphere is closed manifold.
  - Plane is not.

---

**Optimization using Visibility:**

**Back-Face Culling**

- On the surface of a closed orientable manifold, polygons whose normal points away from the camera are always stripped.

---

**Back-Face Culling**

Most objects in scene are typically "solid" rigorously: orientable closed manifolds.

- Identifying and removing all points homeomorphic to disc.
  - Boundary partitions space into interior & exterior.
**Manifold**

Examples of manifold objects:
- Spheres
- Torus
- Old-fashioned CAD part

**Back-Face Culling**

Examples of non-manifold objects:
- A single polygon
- A terrain or height field
- Polyhedron with missing face
- Anything with cracks or holes in boundary
- One-polygon thick lampshade

**Back-face Culling: VCS**

buffer idea: call if eye is below polygon plane

**Back-face Culling: NDCS**

VCS

NDCS

eye

works to call if $N_x > 0$

**Blending**

How might you combine multiple elements?

- New color A, old color B

**Premultiplying Colors**

Specify opacity with alpha channel ($r_{_A}, g_{_A}, b_{_A}, a_{_A}$)
- $A \times B$ is $r_{_A} \times B, g_{_A} \times B, b_{_A} \times B, a_{_A} \times B$

But what if B is also partially transparent?
- $C = (r_{_B}, g_{_B}, b_{_B}, 1) \times B, A = (r_{_A}, g_{_A}, b_{_A}, a_{_A})$

A multiply, effective equvalent for alpha vs. 0.5

**OpenGL Blending**

In OpenGL:
- Enable blending:
  - glEnable(GL_BLEND)
- Specify blending function:
  - glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
  - $C = a_{_A} \times r_{_A} + (1-a_{_A}) \times a_{_B}$

**OpenGL Blending**

Caveats:
- Alpha blending works on order-dependent operation
- It matters which object is drawn first and last
- Which surface is in front
- For 3D scenes, this makes it necessary to keep track of rendering order explicitly
- Possibly also viewpoint-dependent
- E.g., always draw "back" surface first
- Also note: interaction with z-buffer
Double Buffering

Framebuffer:
- Plane of memory where the final image is written
- Problem
  - The display needs to read the contents, synchronously, while the GPU is already working on the next frame
  - Could result in display of partially rendered images on screen
- Solution
  - Have two buffers
  - One is currently displayed (front buffer)
  - One is rendered into for the next frame (back buffer)

Switching buffers:
- At end of rendering one frame, simply exchange the pointers to the front and back buffer
- GLUT toolkit: glutSwapBuffers() function
- Different functions under windows/KT if not using GLUT

Interactive Object Selection

Move cursor over object, click
- How to decide which is below?

Ambiguity
- Many 3D world objects map to same 2D point

Common approaches
- Manual ray intersection
- Bounding extents
- Bisection region with hit list (OpenGL support)

Manual Ray Intersection

Do all computation at application level
- Map selection point to a ray
- Intersect ray with all objects in scene

Advantages
- No library dependence

Disadvantages
- Difficult to program
- Slow, work to do depends on total number and complexity of objects in scene

Bounding Extents

Keep track of axis-aligned bounding rectangles

Advantages
- Conceptually simple
- Easy to keep track of boxes in world space

Disadvantages
- Low precision
- Must keep track of object-rectangle relationship

Extensions
- Do more sophisticated bound bookkeeping
  - First level: box check, second level: object check
OpenGL Picking

"Render" image in picking mode
- Pixels are never written to framebuffer
- Only store IDs of objects that would have been drawn

Procedure
- Set unique ID for each pickable object
- Call the regular sequence of `gIDDrawVertExtEnd` commands
  - If possible, use `gIDColor`, `gIDNormal`, `gIDTexCoord` etc. for performance

Viewport

Small rectangle around cursor
- Change cursor to a flicker cursor
- `F10` key to acquire a target pixel
- `F11` key to acquire a target area
- Allow several modes of view

Render Modes

```c
for(int i = 0; i < 2; i++) {
    glPushMatrix();
    for(int j = 0; j < 2; j++) {
        glTranslatef(i*10.0, 0, j * 10.0);
        glPushMatrix();
        glTranslatef(i*10.0, 0, j * 10.0);
        glCallList(snowManHeadDL);
        glCallList(snowManBodyDL);
        glPopMatrix();
    }
    glPopMatrix();
}
```

Why rectangle instead of point?
- Practical: largest target is a function of the distance to and size of the target
- Allow several modes of view

Hit List

```c
for(int i = 0; i < 2; i++) {
    glPushMatrix();
    for(int j = 0; j < 2; j++) {
        glTranslatef(i*10.0, 0, j * 10.0);
        glPushMatrix();
        glTranslatef(i*10.0, 0, j * 10.0);
        glCallList(snowManHeadDL);
        glCallList(snowManBodyDL);
        glPopMatrix();
    }
    glPopMatrix();
}
```

Hierarchical Names Example

```c
if (g_selected[0].data[0] == GL_RENDER) {
    g_selected[0].data[1] = GL_SELECT;
    if (g_selected[0].data[2] == GL_SELECT) {
        g_selected[0].data[3] = GL_FEEDBACK;
    }
}
```

Using OpenGL Picking

Example code:
```c
def handleEvents():
    gSelectedBuffer(1000, buffer);
    gRenderMode(GL_SELECT);
    drawObject();
    // Extract data from the selection buffer
    gSelectedFootprint(GL_SELECT);
    // Test analysis with multiple different footprint records
```

Name Stack

- "names" are just integers
  - `glPushMatrix(name)`
  - `glPopMatrix(name)`
  - `gIDMatrix(name)`
- Can have multiple names per object
- Helpful for identifying objects in a hierarchy

Tricky to compute
- Invert viewport matrix, set up new orthogonal projection

Simple utility command
- `glDrawBuffers(x, y, w, v, viewport)`
  - `x,y` is viewport
  - `w,v` is sensitivity (in pixels)
- Push old setup first, so can pop it later

http://www.lightwea.com/opengl/picking/
### Integrated vs. Separate Pick Function

**Integrate:** use same function to draw and pick
- Simpler to code
- Name stack commands ignored in render mode

**Separate:** customize functions for each
- Potentially more efficient
- Can avoid drawing unprintable objects

---

### Select/Hit

**Advantages**
- Faster
  - OpenGL support means hardware accel
  - Only do clipping work, no shading or normalization
  - Flexible precision
  - Size of region controllable
  - Flexible architectures
  - Custom code possible, e.g., guaranteed frame rate

**Disadvantages**
- More complex