Visibility: Z Buffering

Poll

- How far are people on project 2?
- Preferences for
  - Plan A: status quo
    - P2 stays due Tue Nov 4, stays 10% of total grade
    - P3 is "the big one" stays 15%, due Fri Nov 28
  - Plan B: P2 is "the big one"
    - P2 extension to Mon Nov 10, upgrade weight to 15%
    - P3 smaller, downgrade weight to 10%, due Fri Dec 5

Demo

- Sample program
  - Remember, download all 3 textures to run this!
  - You should also use the checkerboard image
- Questions?

Readings

- Chapter 8.8: hidden surface removal

News

- Yet more extra office hours TBD
  - Check newsgroup

Visibility recap
Invisible Primitives

- why might a polygon be invisible?
  - polygon outside the field of view / frustum
    - clipping
  - polygon is backfacing
    - backface culling
  - polygon is occluded by object(s) nearer the viewpoint
    - hidden surface removal

Back-Face Culling

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

Back-face Culling: NDCS

\[
\begin{align*}
\text{works to cull if } N_z &> 0 \\
\text{vcs} & \quad \text{NDCS} \\
\text{y} \quad \text{z} & \quad \text{eye} \\
\end{align*}
\]

Painter’s Algorithm

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

BSP Trees

- preprocess: create binary tree
  - recursive spatial partition

BSP Trees

- runtime: correctly traversing this tree enumerates objects from back to front
  - check which side of plane viewpoint is on
  - draw far, draw object in question, draw near
Summary: BSP Trees

- **pros:**
  - simple, elegant scheme
  - only writes to framebuffer (no reads to see if current polygon is in front of previously rendered polygon, i.e., painters algorithm)
  - thus very popular for video games (but getting less so)

- **cons:**
  - computationally intense preprocess stage restricts algorithm to static scenes
  - slow time to construct tree: $O(n \log n)$ to

---

Warnock’s Algorithm

- **pros:**
  - very elegant scheme
  - extends to any primitive type

- **cons:**
  - hard to embed hierarchical schemes in hardware
  - complex scenes usually have small polygons and high depth complexity
  - thus most screen regions come down to the single-pixel case

---

The Z-Buffer Algorithm

- both BSP trees and Warnock’s algorithm were proposed when memory was expensive
- example: first 512x512 framebuffer > $50,000!
- Ed Catmull (mid-70s) 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 (Z in eye coordinates) through projection transform
  – use canonical viewing volumes
  – each vertex has z coordinate (relative to eye point) intact

Interpolating Z

• edge equations: Z just another planar parameter:
  • \( z = (-D \cdot Ax - By) / C \)
  • if walking across scanline by (Dx)
    \( z_{new} = z_{old} - (A/C)(Dx) \)
  – total cost:
    • 1 more parameter to increment in inner loop
    • 3x3 matrix multiply for setup
  • edge walking: just interpolate Z along edges and across spans

Depth Test Precision

– reminder: projective transformation maps eye-space z to generic z-range (NDC)
– simple example:

\[
\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = T \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
\]

– thus:

\[
z_{NDC} = a \cdot z_{eye} + b \]

\[
z_{eye} = a + \frac{b}{z_{eye}}
\]

The Z-Buffer Algorithm

• add 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 and store in pixel of Z-buffer
  – suppress writing to a 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 8+8+8+24 bits, can be more

\[
\text{for all } i, j \{ \\
\text{Depth}[i,j] = \text{MAX_DEPTH} \\
\text{Image}[i,j] = \text{BACKGROUND_COLOUR} \\
\text{for all polygons } P \{ \\
\text{if } (x_{\text{pixel}} < \text{Depth}[i,j]) \{ \\
\quad \text{Image}[i,j] = C_{\text{pixel}} \\
\quad \text{Depth}[i,j] = x_{\text{pixel}} \\
\text{\}} \\
\text{\}}
\]

Depth Test Precision

– therefore, depth-buffer essentially stores \(1/z\), rather than \(z\)
– this yields precision problems with integer depth buffers:
Depth Test Precision

- precision of depth buffer is bad for far objects
- depth fighting: 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

Z-buffer

- hardware support in graphics cards
- poor for high-depth-complexity scenes
  - need to render all polygons, even if most are invisible
- “jaggies”: pixel staircase along edges

The A-Buffer

- antialiased, area-averaged accumulation buffer
  - z-buffer: one visible surface per pixel
  - A-buffer: linked list of surfaces
  - data for each surface includes
    - RGB, Z, area-coverage percentage, ...

The Z-Buffer Algorithm

- 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
- 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

- lots of memory (e.g. 1280x1024x32 bits)
  - with 16 bits cannot discern millimeter differences in objects at 1 km distance
  - Read-Modify-Write in inner loop requires fast memory
  - hard to do analytic antialiasing
  - we don’t know which polygon to map pixel back to
  - shared edges are handled inconsistently
    - ordering dependent
  - hard to simulate translucent polygons
  - we throw away color of polygons behind closest one
Visibility

– object space algorithms
  • explicitly compute visible portions of polygons
  • painter’s algorithm: depth-sorting, BSP trees
– image space algorithms
  • operate on pixels or scan-lines
  • visibility resolved to the precision of the display
  • Z-buffer, Warnock’s

Hidden Surface Removal

• 2 classes of methods
  – image-space algorithms
    • perform visibility test for every pixel independently
    • limited to resolution of display
    • performed late in rendering pipeline
  – object-space algorithms
    • determine visibility on a polygon level in camera coordinates
    • resolution independent
    • early in rendering pipeline (after clipping)
    • expensive

Pick up Homework 1