CPSC 314
21 – DEPTH TEST

Textbook: 11.1

UGRAD.CS.UBC.CA/~CS314

Mikhail Bessmeltsev
THE RENDERING PIPELINE

Vertex Shader
- Modelview transform
- Per-vertex attributes

Vertex Post-Processing
- Viewport transform
  - Clipping

Rasterization
- Scan conversion
- Interpolation

Fragment Shader
- Texturing/...
  - Lighting/shading

Per-Sample Operations
- Depth test
- Blending

Framebuffer
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
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
    • To compute z per pixel use barycentric coordinates
      • Don’t forget about perspective correction
  • Or maybe fragment shader modifies z
THE Z-BUFFER ALGORITHM

• Augment color framebuffer with Z-buffer: Z per pixel
  • Also called depth buffer
  • First 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
• Store \((r, g, b, z)\) for each pixel
  • typically 8+8+8+24 bits, can be more  
    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
      }
    }  
  }  
}
INTERPOLATING Z

- Use barycentric coordinates
  - Interpolate z like other parameters
    - E.g. color
    - Use one of three formulas
      - Plane/edge walk/barycentric
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:

$$T \left( \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

• Thus:

$$z_{\text{NDC}} = \frac{az_{\text{eye}} + b}{-z_{\text{eye}}} = -a - \frac{b}{z_{\text{eye}}}$$
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 cm differences in objects at 1 km distance
HOW NEAR AND FAR PLANES AFFECT PRECISION

\[ Z_{NDC} = \frac{az_{eye} + b}{-z_{eye}} = -a - \frac{b}{z_{eye}} \]

\[ Z_{NDC} = \frac{f + n}{f - n} + \frac{2fn}{(f - n)z_{eye}} \]

\[ \frac{dz_{NDC}}{dz_{eye}} = \frac{-2fn}{(f - n)z_{eye}^2} \]
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
  - \textit{Ordering dependent}
Z-BUFFER CONS

• Requires more memory
  • (e.g. 1280x1024x32 bits, depends on the implementation)
• 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