**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

**BSP Trees (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:
  - 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

Depth Test Precision

- reminder: projective transformation maps eye-space z to generic z-range (NDC)
- simple example:
  \[
  \begin{bmatrix}
    x' \\ y' \\ z' \\
  \end{bmatrix} =
  \begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & -1 & 1 \\
  \end{bmatrix}
  \begin{bmatrix}
    x \\ y \\ z \\
  \end{bmatrix}
  \]
  - thus:
  \[
  z_{\text{NDC}} = \frac{z - a}{b} = \frac{z_{\text{eye}} - a}{b_{\text{eye}}}
  \]

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 ratio \(b/a\)
- rule of thumb: \(b < 1000\) 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

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

Interpolating Z

- barycentric coordinates
  - interpolate Z like other planar parameters

Render Pipeline

- store (r,g,b,z) for each pixel
- typically 8+8+8+24 bits, can be more

Z-Buffer

- 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

Projective Rendering Pipeline

- performs (4D) perspective division
- object world viewing
  - 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

Rendering Pipeline

- performs (4D) projection transformation
- object world viewing
  - 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

Clipping

- performs (4D) view transformation
- object world viewing
  - 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
**Back-face Culling**

- on the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:
  
  ![Diagram of back-face culling](image)

- most objects in scene are typically "solid"
  - rigorously: orientable closed manifolds
    - manifold: local neighborhood of all points isomorphic to disc
    - boundary partitions space into interior & exterior

**Back-Face Culling**

- not rendering backfacing polygons improves performance
  - by how much?
    - reduces by about half the number of polygons to be considered for each pixel
    - optimization when appropriate

**Manifold**

- examples of *manifold* objects:
  - sphere
  - torus
  - well-formed CAD part

**Back-Face Culling**

- examples of non-manifold objects:
  - a single polygon
  - a terrain or height field
  - polyhedron w/ missing face
  - anything with cracks or holes in boundary
    - one-polygon thick lampshade

**Invisible Primitives**

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