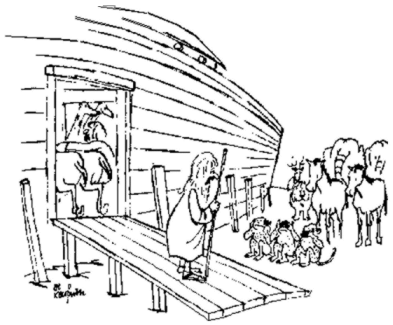
Visibility – Part 2

Binary Space Partitioning Trees

Ray Casting

Depth Buffering



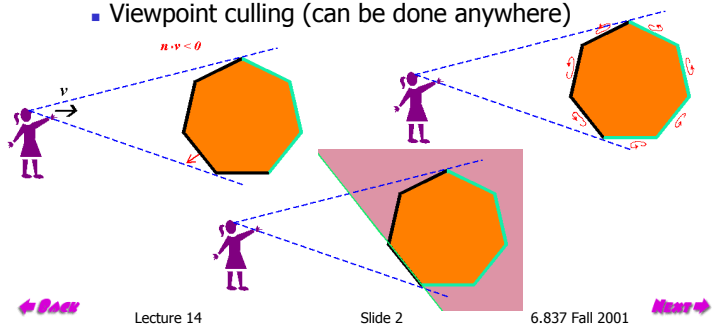
"I hate to break up the set, but one of you has to go."

Visibility – Last Time

Back-Face Culling O(n)

Simple test based on the normal of each face:

- View-direction culling (computed after projection)
- Oriented-face culling (computed at triangle set-up)
- Viewpoint culling (can be done anywhere)



Visibility – Last Time

Painters Algorithm O(n log n)

Sort triangles by their depth (min, max, centroid)

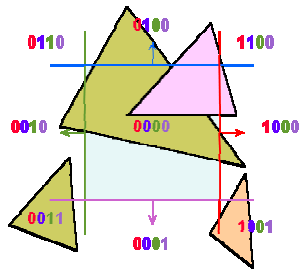
Subdivide cycle overlaps or intersecting triangles

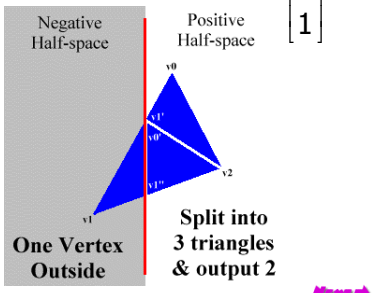
Power of Plane Equations

We've gotten a lot of mileage out of one simple equation:

- 3D outcode clipping
- plane-at-a-time clipping
- viewpoint back-face culling

$$\begin{bmatrix} n_x & n_y & n_z & -d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$



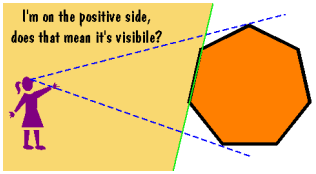


One More Trick with Planes

Consider the complement argument of the viewpoint culling algorithm:

- Any facet that contains the eye point within its negative half-space is invisible.
- Any facet that contains the eye point within its positive half-space is visible.

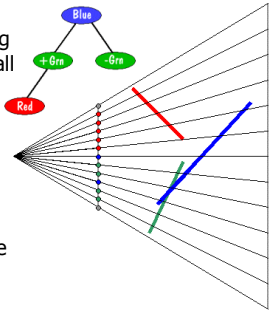
Well almost... it would work if there were no overlapping facets. However, notice how the overlapping facets partition each other. Suppose we build a tree of these partitions.



Constructing a BSP Tree

The Binary Space Partitioning (BSP) algorithm:

1. Select a partitioning plane/facet.
2. Partition the remaining planes/facets according to the side of the partitioning plane that they fall on (+ or -).
3. Repeat with each of the two new sets.



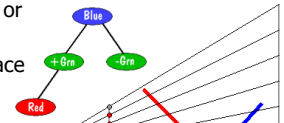
Partitioning requires testing all facets in the active set to find if they lie entirely on the positive side of the partition plane, entirely on the negative side, or if they cross it. In the case of a crossing facet we clip it into two halves (using the plane-at-a-time clipping algorithm).

BSP Visualizer Applet

Computing Visibility with BSP trees

Starting from the root of the tree.

1. Classify viewpoint as being in the positive or negative half-space of our plane
2. Call this routine with the opposite half-space
3. Draw the current partitioning plane
4. Call this routine with the same half-space



Intuitively, at each partition, we first draw the stuff further away than the current plane, then we draw the current plane, and then we draw the closer stuff. BSP traversal is called a "hidden surface elimination" algorithm, but it doesn't really "eliminate" anything; it simply orders the drawing of primitive in a back-to-front order like the Painter's algorithm.

BSP Visualizer Applet

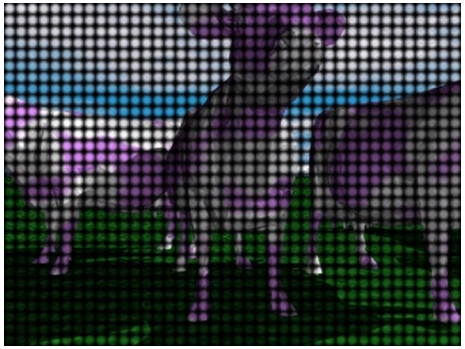
BSP Tree Example

- Computing visibility or depth-sorting with BSP trees is both simple and fast.
- It resolves visibility at the primitive level.
- Visibility computation is independent of screen size
- Requires considerable preprocessing of the scene primitives
- Primitives must be easy to subdivide along planes
- Supports CSG

BSP Visualizer Applet

Pixel-level Visibility

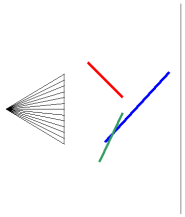
Thus far, we've considered visibility at the level of primitives. Now we will turn our attention to a class of algorithms that consider visibility at the level of each pixel.



Ray Casting

Cast a ray from the viewpoint through each pixel to find the closest surface

```
for (each pixel in image) {
   compute ray for pixel
   set depth = ZMAX
   for (each primitive in scene) {
      if (ray intersects primitive and
          distance < depth) {
          pixel = object color
          depth = distance to object
      }
   }
}
```



Ray Casting

Pros:

- Conceptually simple
- Can take advantage of spatial coherence in scene
- Can be extended to handle global illumination effects (ex: shadows and reflectance)

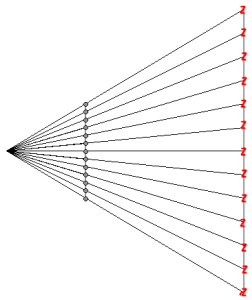
Cons:

- Renderer must have access to entire model
- Hard to map to special-purpose hardware
- Visibility determination is coupled to sampling
  - Subject to aliasing
  - Visibility computation is a function of resolution

Depth Buffering

Project all primitives and update depth of each pixel

```
set depth of all pixels to ZMAX
for (each primitive in scene)  {
   determine pixels touched
   for (each pixel in primitive) {
      compute z at pixel
      if (z < depth) {
         pixel = object color
         depth = z
      }
   }
}
```



Depth Buffer

Pros:

- Primitives can be processed immediately
- Primitives can be processed in any order
  - Exception: primitives at same depth
- Well suited to H/W implementation
- Spatial coherence
  - Incremental evaluation of loops

Cons:

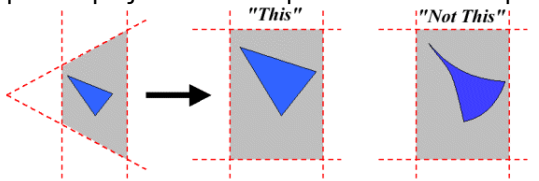
- Visibility determination is coupled to sampling (aliasing)
- Requires a Raster-sized array to store depth
- Excessive over-drawing

What Exactly Gets Stored in a Depth Buffer?

Recall that we augmented our projection matrix to include a mapping for z values:

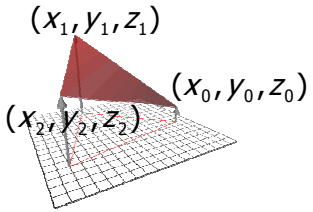
$$\begin{bmatrix} x'w \\ y'w \\ z'w \\ w \end{bmatrix} = \begin{bmatrix} \frac{\text{width}\cdot\text{near}}{\text{right}-\text{left}} & 0 & \frac{-\text{left}\cdot\text{width}}{\text{right}-\text{left}} & 0 \\ 0 & \frac{\text{height}\cdot\text{near}}{\text{bottom}-\text{top}} & \frac{-\text{top}\cdot\text{height}}{\text{bottom}-\text{top}} & 0 \\ 0 & 0 & \frac{z_{\max}\cdot\text{far}}{\text{far}-\text{near}} & \frac{-\text{near}\cdot z_{\max}\cdot\text{far}}{\text{far}-\text{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The perspective projection matrix preserves lines and planes:



Interpolating Depth

Projection that preserves planes allows us to use the plane equation for interpolation.



$$Ax' + By' + C = z'$$

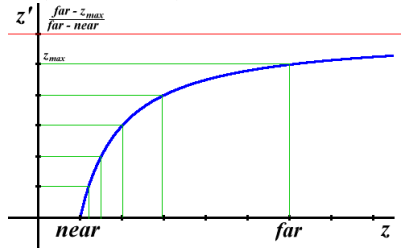
$$\begin{bmatrix} z'_0 \\ z'_1 \\ z'_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix}$$

- Solve the linear system for A, B, and C and use the values to interpolate the depth z' at any point x', y'.
- Similar computations are also used for rasterization and color interpolation

Monotonic Depth Values

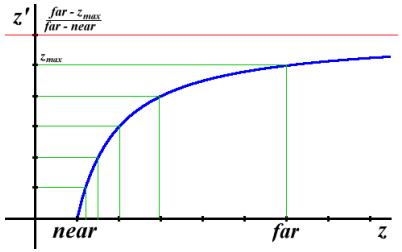
We need to be careful when reading the values out of a depth-buffer and interpolating them. Even though, our interpolated values of z lie on a plane, uniform differences in depth-buffer values do no correspond to a uniform differences in space:

$$z' = \frac{\text{far}\cdot z_{\max}\cdot(z-\text{near})}{z\cdot(\text{far}-\text{near})} = \frac{\text{far}\cdot z_{\max}}{\text{far}-\text{near}}\left(1-\frac{\text{near}}{z}\right)$$



Monotonic Depth Values

However, our z-comparisons will still work because this parameter mapping, while not linear, is monotonic.
Note that when the z values are uniformly quantized the number of discrete discernable depths is greater closer to the near plane than near the far plane. Is this good?



Next Time