Hidden Surface Algorithms

Reading

Reading:

- Watt, 6.6 (esp. intro and subsections 1, 4, and 8–10), 12.1.4.

Optional reading:

- Foley, van Dam, Feiner, Hughes, Chapter 15
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

Introduction

So far we know how to construct a hierarchical 3D model and map points from 3D to 2D. Is that all?

Not every surface of an object is visible from a given camera viewpoint. We need an algorithm to determine which parts get drawn.

Known as the **hidden surface elimination problem** or the **visible surface determination problem**..

Hidden surface algorithms can be characterized in at lease three ways:

- Object-space vs. image-space
- Object order vs. image order
- Sort first vs. sort last

Object-space algorithms

Basic idea: operate on 3D objects

- For each object (3D primitive) in the scene, compute which part is visible, then draw
- Objects typically intersected against each other
- Tests performed to high precision
- Resulting list of visible objects can be drawn at any resolution

Complexity:

- May have to compare every pair of objects, so for n objects, can take $O(n^2)$ time
- For an *mxm* display, have to fill in colors for m^2 pixels.
- Overall complexity can be $O(k_{obj}\, n^2 + k_{disp}\, m^2)$.

Implementation:

- Difficult to implement
- Can get numerical problems

Image-space algorithms

<u>Basic idea:</u> operate on pixels

- Find the closest point as seen through each pixel
- Calculations performed at display resolution
- Precision requirements typically not high

<u>Complexity:</u>

- Naïve approach checks all n objects at every pixel. Then, *O(   )*.
- Better approaches check only the objects that *could* be visible at each pixel. Let's say, on average, *d* objects are visible at each pixel (a.k.a., depth complexity). Then, *O(   )*.

<u>Implementation:</u>

- Very simple to implement.
  - Used a lot in practice.

Object order vs. image order

<u>Object order:</u>

- Consider each object only once, draw its pixels, and move on to the next object.
- Might draw the same pixel multiple times.

<u>Image order:</u>

- Consider each pixel only once, find nearest object, and move on to the next pixel.
- Might compute relationships between objects multiple times.

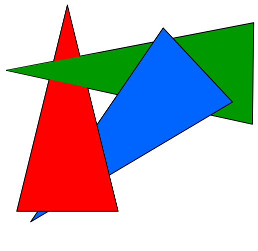
Sort first vs. sort last

<u>Sort first:</u>

- Find some depth-based ordering of the objects relative to the camera, then draw back to front.
- Build an ordered data structure to avoid duplicating work.

<u>Sort last:</u>

- Sort implicitly as more information becomes available.



Outline of Lecture

- Z-buffer
- Ray casting
- Binary space partitioning (BSP) trees

Z-buffer

Idea: along with a pixel's red, green and blue values, maintain some notion of its *depth*

- An additional channel in memory, like alpha
- Called the depth buffer or Z-buffer

```
void draw_mode_setup( void ) {
  …
  GlEnable( GL_DEPTH_TEST );
  …
}
```

When the time comes to draw a pixel, compare its depth with the depth of what's already in the framebuffer. Replace only if it's closer

Very widely used

History

- Originally described as "brute-force image space algorithm", mentioned in an appendix
- Written off as totally impractical algorithm (for huge memories)
- Today, done easily in hardware

Z-buffer

The **Z-buffer**' or **depth buffer** algorithm [Catmull, 1974] is probably the simplest and most widely used.

Here is pseudocode for the Z-buffer hidden surface algorithm:

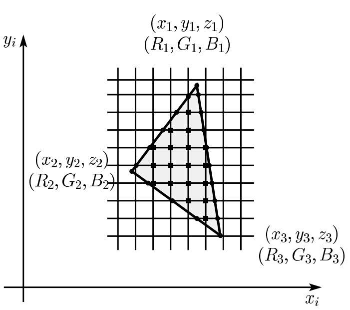
for each pixel *(i,j)* **do**
  Z-buffer *[i,j]* ← *FAR*
  Framebuffer[i,j] ← <background color>
end for
for each polygon *A* **do**
  for each pixel in *A* **do**
    Compute depth *z* and shade *s* of *A* at *(i,j)*
    if z > *Z-buffer [i,j]* **then**
      Z-buffer [i,j] ← *z*
      Framebuffer[i,j] ← *s*
    end if
  end for
end for

Q: What should FAR be set to?

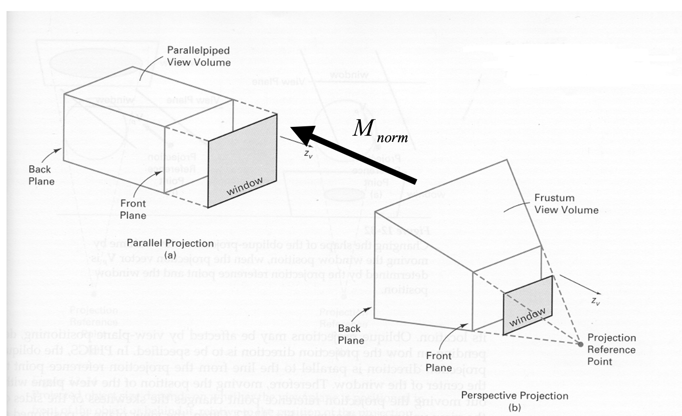
Z-buffer, cont'd

The process of filling in the pixels inside of a polygon is called **rasterization**.

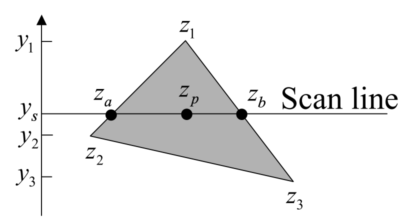
During rasterization, the *z* value and shade *s* can be computed incrementally (fast!).



Z value interpolation



After projective normalization, the z values may be linearly interpolated within the image



$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

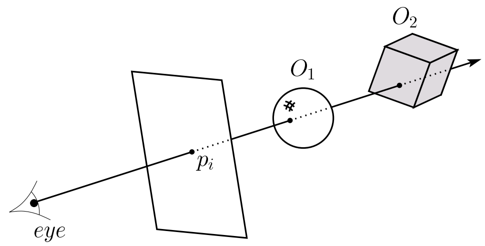
$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Z-buffer: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn't need all objects before drawing begins)?
- If objects move, does it take extra work than normal to draw the frame?
- If the viewer moves, does it take extra work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

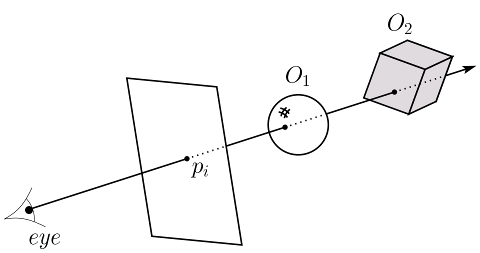
Ray casting



Idea: For each pixel center $\boldsymbol{P}_{ij}$

- Send ray from eye point (COP), **c**, through $\boldsymbol{P}_{ij}$ into scene.
- Intersect ray with each object.
- Select nearest intersection.

Ray casting, cont.



Implementation:

- Might parameterize each ray:

  $\mathbf{r}(t) = \mathbf{c} + t\,(\boldsymbol{P}_{ij} - \mathbf{c})$

- Each object O_k returns $t_k > 1$ such that first intersection with O_k occurs at $\mathbf{r}(t_k)$.

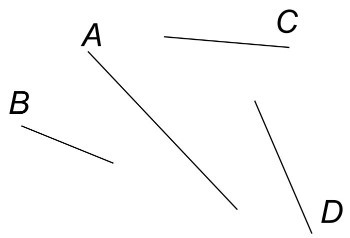
Q: Given the set $\{t_k\}$ what is the first intersection point?

Note: these calculations generally happen in world coordinates.

Ray casting: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn't need all objects before drawing begins)?
- If objects move, does it take extra work than normal to draw the frame?
- If the viewer moves, does it take extra work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

Binary-space partitioning (BSP) trees



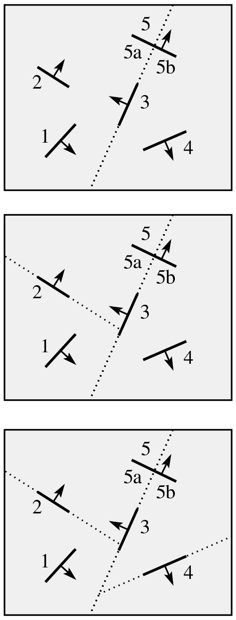
Idea:

- Do extra preprocessing to allow quick display from any viewpoint.

Key observation: A polygon *A* is painted in correct order if

- Polygons on far side of *A* are painted first
- P is painted next
- Polygons in front of *A* are painted last.

BSP tree creation



BSP tree creation (cont'd)

procedure *MakeBSPTree*:

takes *PolygonList L*

returns *BSPTree*

Choose polygon *A* from *L* to serve as root

Split all polygons in *L* according to *A*

node ← *A*

node.neg ← *MakeBSPTree*(Polys on - side of A)

node.pos ← *MakeBSPTree*(Polys on + side of A)

return node

end procedure

Note: Performance is improved when fewer polygons are split --- in practice, best of ~ 5 random splitting polygons are chosen.

Note: BSP is created in *world* coordinates.

BSP tree display

procedure *DisplayBSPTree:*

Takes *BSPTree T*

if *T* is empty **then return**

if viewer is in front (on pos. side) of *T.node*

DisplayBSPTree(T. ______)

Draw T.node

DisplayBSPTree|(T.______)

else

DisplayBSPTree(T. ______)

Draw T.node

DisplayBSPTree(T. ______)

end if

end procedure

BSP trees: Analysis

- Classification?
- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn't need all objects before drawing begins)?
- If objects move, does it take extra work than normal to draw the frame?
- If the viewer moves, does it take extra work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

Visibility tricks for Z-buffers

Z-buffering is ***the*** algorithm of choice for hardware rendering, so let's think about how to make it run as fast as possible…

What is the complexity of the Z-buffer algorithm?

What can we do to decrease the constants?

Summary

What to take home from this lecture:

- Classification of hidden surface algorithms
- Understanding of Z-buffer and ray casting hidden
- surface algorithms
- Familiarity with BSP trees