6.837 F98 Lecture 16: November 10, 1998

(Thanks to John Alex for help preparing these slides.)

Administrative

Tests returned at end of class.
Project 4 assigned at end of class.

Visibility

Our trip through the graphics pipeline is nearly complete.

We now know how to transform, clip, and project polygons into NDC
(Canonical coordinates, normalized device coordinates)

But we only recently addressed what to do when scene primitive is **invisible**;
that is, when it contributes no pixels to the rendered framebuffer.

A scene primitive can be invisible for three reasons:

1. Primitive lies *outside field of view*
2. Primitive is *back-facing* (under certain model assumptions, below)
3. Primitive is *occluded* by one or more objects nearer to eyepoint

Obviously we don't want to spend a lot of work processing things which contribute nothing to the rendered image. Thus, the following
fundamental questions of **visibility computation** arise:

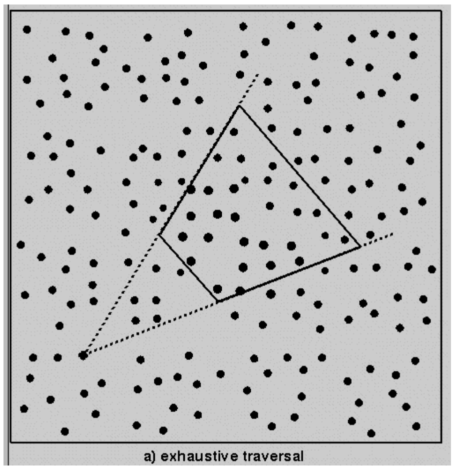
- How can invisible scene primitives be *eliminated* efficiently? Or...
- How can visible scene primitives be *identified* efficiently? Or...
- How can hybrid/alternative versions of the problem be posed? (Later.)

First example of visibility computation: **clipping to view frustum** (last time)

- Remove scene primitives entirely outside frustum (i.e., outside field of view)
    - Note that this includes primitives "behind" eye (actually those behind near plane)
- Pass through scene primitives entirely inside frustum

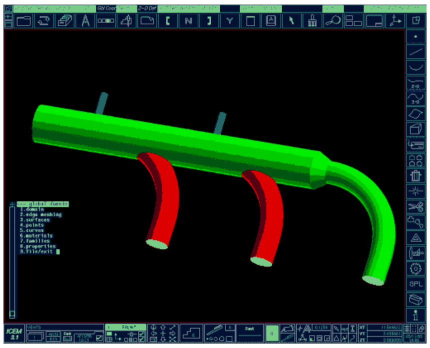
Modify remaining primitives so as to pass through only the portion inside view frustum

- Note that clipping can be done entirely in object space, i.e., in continuous coordinates with no reference to a particular viewport or rendering resolution
- What is the cost of clipping a scene of *n* primitives? Can we do better?

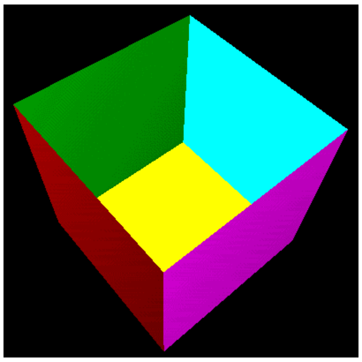


Second example of visibility computation: **back-face elimination**

- Objects typically assumed to be closed, orientable manifolds
    - Neighborhood of any point is like a disc (surface has no cracks or holes)
    - Object boundary partitions 3D space into interior and exterior regions
    - Examples of *manifold* objects: sphere, torus, well-formed CAD part:



Examples of *non-manifold* objects: single polygon; polyhedron with missing face; anything with cracks, holes in the boundary, etc.

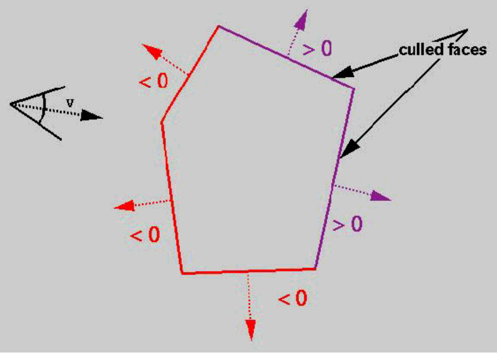


Q. How can we implement this?

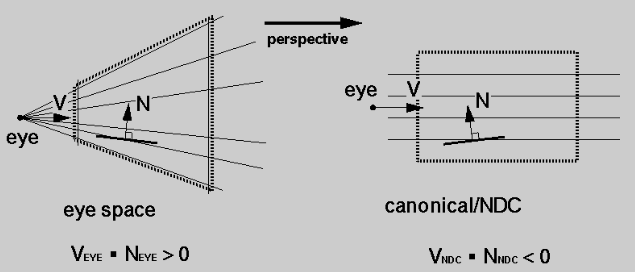
Idea: compare view direction, face

normal

```
for each face F of each object
    N = outward-pointing normal of F
    if N dot V > 0, throw away the face
```



Can we apply this "normal test" anywhere in the pipeline?



When should back-face culling be used?
What is its cost for a collection of *n* polygons?

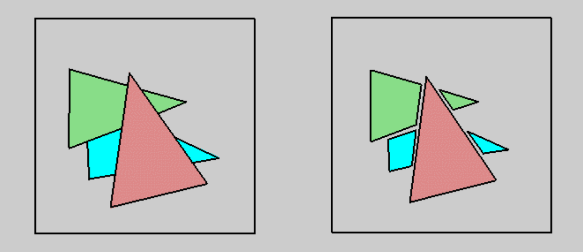
Back-face culling is an example of a criterion that can be applied anywhere along the pipeline: in world coords, eye coords, NDC, even in screen space !

Where is the best place in the pipeline to apply back-face culling?

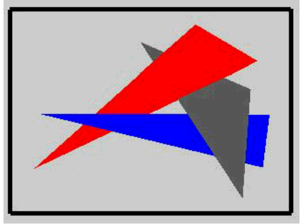
What portion of the scene will it tend to eliminate, on average?

Suppose we clip our scene to the frustum, then remove all back-facing polygons.
Are we done? That is, does this solve our visibility problem?

No! For most interesting scenes and viewpoints, some polygons will **overlap**;
somehow, we must determine which portion of each polygon is visible to eye.



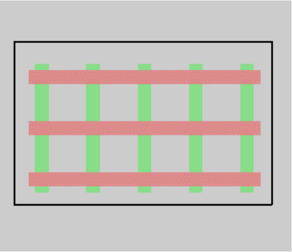
One possibility: simply draw the polygons in the "right order" so that the right picture results (example: blue, then green, then peach). So is visibility just a back-to-front sorting problem?



In 2-D, for non-intersecting line segments, yes -- visibility reduces to sorting.
However, in 3-D even non-intersecting polygons can form a **cycle** (above), which has *no valid visibility order*!

Early algorithms for visibility (now called "analytic" algorithms) computed the set of visible *fragments* directly, in object space, then rendered the fragments to a pen plotter, or to a vector display, or (a bit later on) to a framebuffer.

What is the cost of describing (and computing) the fragment map at right (above) for a scene composed of *n* polygons (say, triangles or quadrilaterals)?



... It's **quadratic** in *n* **!**

Now, as soon as renderers were invented, people started crafting geometric models.
Realism called for lots of polygons, which spelled trouble for these costly algorithms.
So, for about a decade (from the late '60s to the late '70s) there was intense interest in finding efficient algorithms for efficient **hidden surface removal**.

There's quite an extended literature about this; today I'll show just two nifty algorithms, one called **BSP traversal** from 1979/80, and one called **Warnock's algorithm**, from 1968/69.

BSP tree construction and

traversal (Fuchs et al, 1980)

BSP means "binary space partitioning"; organizes all space into a binary tree

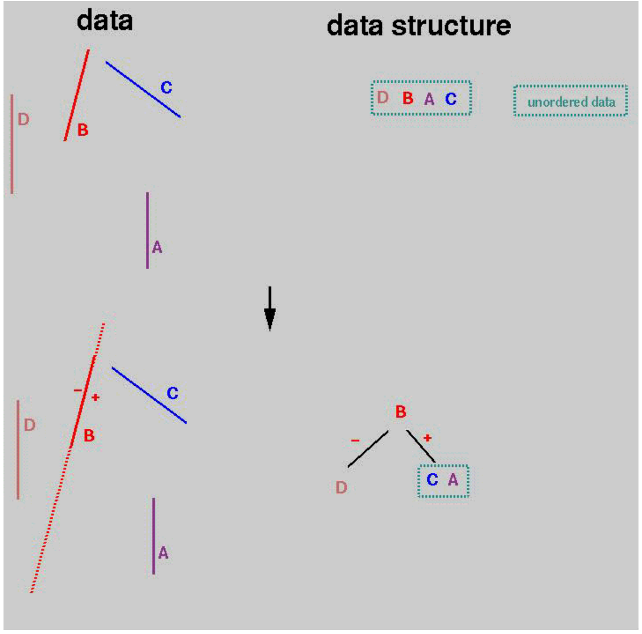
BSP traversal is one of a class of "list-priority" visibility algorithms

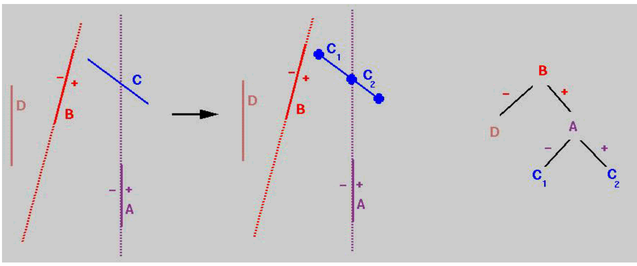
after a "static pre-processing stage", constructs a data structure which
for a specified query eyepoint returns an *ordered list* of polygon (fragments)
that can be drawn one on top of the next, via scan conversion for example.

Preprocess: Building a BSP tree out of polygons

1. split along the plane of any polygon
2. classify all polygons into positive & negative halfspaces of the plane
   * if a polygon is on the plane, split it in half
3. recurse down the positive halfspace
4. recurse down the negative halfspace

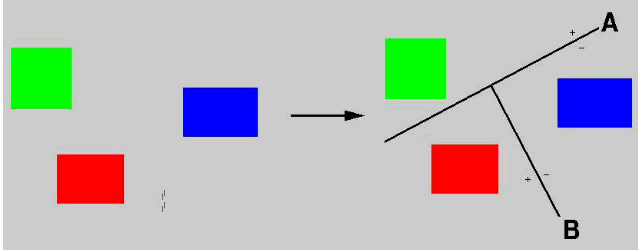
each node stores a plane, the polygon on that plane, and two child pointers,
one each for the positive & negative halfspaces.



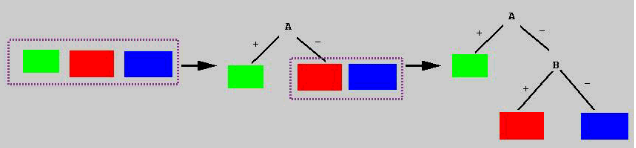


Building a tree from non-polygonal objects

choose split planes that "well-separate" objects from one another:



1. if only one object left, we're done.
2. split along *any* plane, preferably one that divides the objects well
3. classify all polygons into positive & negative halfspaces
4. if any objects intersect the plane, split them in half *(may be costly!)*
5. recurse to construct subtrees for the two children.



each node stores a plane and two child pointers

no object *on* the plane

Dynamic query: Produce ordered polygon list from the tree, instantaneous eyepoint

node::draw (viewpoint)

- classify viewpoint as being in the positive or negative halfspace of our plane

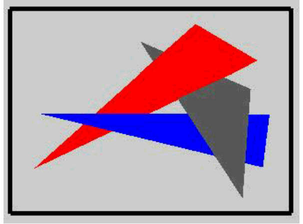
  call that the "near" halfspace, and the associated child node the "near child"

  1. farchild->draw (viewpoint)
  2. draw the polygon on the plane
  3. nearchild->draw (viewpoint)

Intuitively: at each partition, draw the "farther side", then the "nearer side"

BSP traversal is called a "hidden surface elimination" algorithm, but it doesn't really "eliminate" anything; it simply orders polygons so the resulting picture is correct.

Can we always find a polygon ordering (without fragments) that gives the right rendered image?



... Nope !

What are the time and space requirements for a BSP tree of *n* polygons?
What is the cost of a query for a single eyepoint?

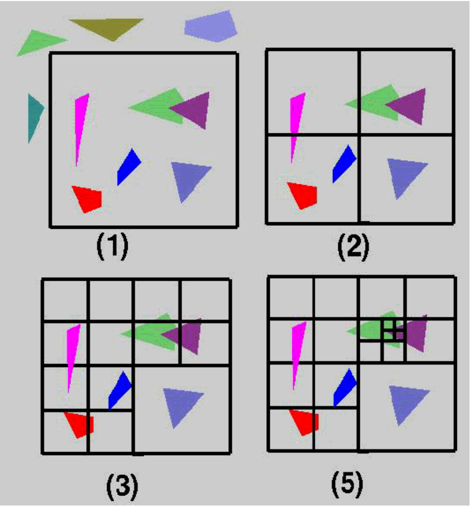
Next: **Warnock's algorithm (1969).** This is an elegant hybrid of object-space and image-space computations, based on a simple idea that has recurred over and over again in computer graphics: *If the situation is too complex,* **subdivide**.

0) start with a "root viewport", and list of all scene primitives
(transformed to screen-space)

Then, recursively:

1) classify each object as incident, or outside of, viewport

2) if number of objects is zero or one,
visibility is established for this portion of the screen
else
subdivide into smaller frusta, distribute incident objects to them, and recurse.



What is the termination condition ?

And how is the correct surface identified in this case?

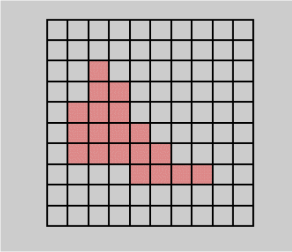
Third example of a visibility

operation is **z-buffering** (or **depth buffering**).

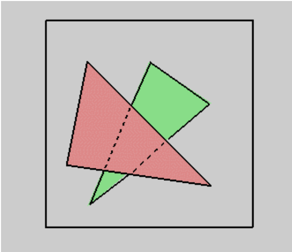
Both of the above algorithms were proposed when memory was very expensive
(the first 512 x 512 framebuffer cost more than fifty thousand dollars).

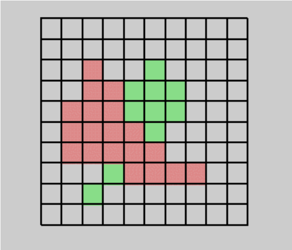
In the mid-70's, Ed Catmull had access to a framebuffer and proposed a
radically new visibility algorithm: **z-buffering**. The idea of z-buffering is
to resolve visibility *independently at each pixel.*

- We have seen how to rasterize polygons into a discrete 2D viewport:



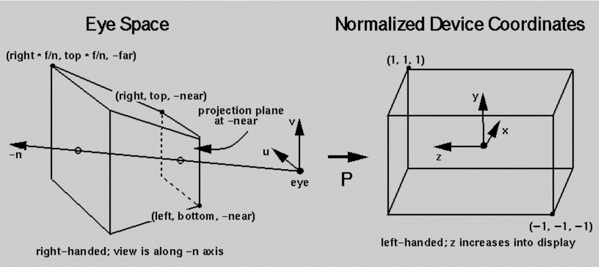
- Question arises: what if multiple primitives occupy the same screen pixel?
  Which one should be allowed to "paint" the pixel?





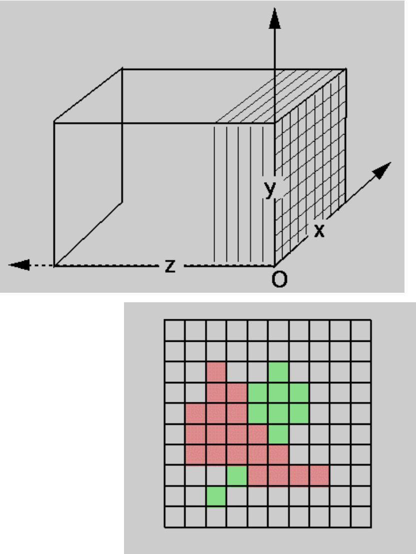
Retain depth through transformation to screen space (simply transform z_NDC)
- Note that both perspective and viewport transformations preserve depth order
- In (admittedly) somewhat confusing terminology, an *x,y* pixel with
  associated color and depth *z* has come to be known as a *fragment*

Augment framebuffer with depth value at each pixel (*b* bits, represent 0 .. 2^b - 1)



At beginning of frame, initialize all pixel depths to largest integer (distant background)

- Now, when rasterizing, simply suppress pixel write if z_fragment > z_pixel !
- In example below, peach triangle is rendered first, then green triangle



Properties of depth-buffer:

- How much memory does depth-buffer use?
- Does the rendered image depend on the drawing order?
- Does the *time* to produce the rendered image depend on drawing order?
- How much of pipeline is traversed by in-frustum, occluded polygons?
  - What are implications for front of pipeline (traversal, matrix transformation)?
  - How does z-buffering load scale with ...
    ... the number of in-frustum polygons?
    ... the framebuffer resolution?

After initial resistance, the world adopted z-buffers in the mid-1980's.

- Then, history repeated itself: the model-crafters demanded more and more polygons (and later, texture), and not only z-buffers but
  earlier pipe stages became significant computational drains.
- Next time -- we'll talk about **conservative visibility algorithms**
  which *shed load* from the rendering pipeline.

And now a word from our sponsor (LM) about exam, next project.

Previous Meeting .... Next Meeting ... Course Page

Last modified: Nov 1998

Prof. Seth Teller, MIT Computer Graphics Group, seth@graphics.lcs.mit.edu