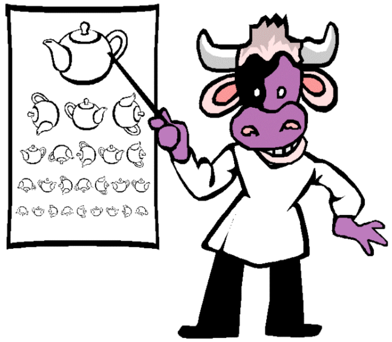
Visibility

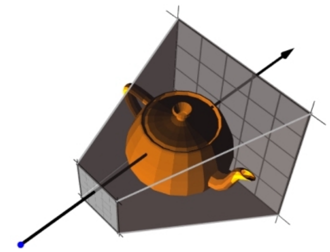
Back-Face Culling
Painter's Algorithm



Visibility Problem

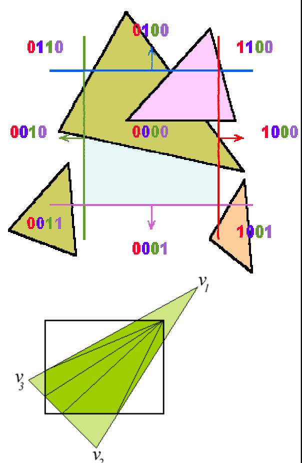
The problem of visibility is to determine which transformed, illuminated, and projected primitives contribute to pixels on the screen. In many cases, however, rather than solving the direct problem of determining what is visible, we will instead address the converse problem of eliminating those primitives that are *invisible*:

- primitives outside of the field of view
- back-facing primitives on a closed, convex object
- primitives occluded by other objects closer to the camera



Outside the Field-of-View

Clipping, as we discussed the lecture before last, addresses the problem of removing the objects outside of the field of view. Outcode clipping attempted to remove all those objects that were entirely outside of the field of view (it came up a little short of that goal, however). Frustum clipping, as demonstrated by the plane-at-a-time approach that we discussed, removed portions of objects that were partially inside of and partially outside of the field of view.



General Approaches

Image-Based Approach

```
for (each pixel in the image) {
   determine the ray through the pixel
   determine the closest object along the ray
   draw the pixel in the color of the closest object
}
```

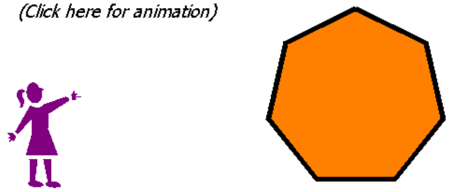
Object-Based Approach

```
for (each object in the world) {
   determine unobstructed object parts
   draw the parts in the appropriate color
}
```

Back-Face Culling

Back-face culling addressing a special case of occlusion called *convex self-occlusion*. Basically, if an object is closed (having a well defined inside and outside) then *some parts of the outer surface must be blocked by other parts of the same surface*. We'll be more precise with our definitions in a minute. On such surfaces we need only consider the normals of surface elements to determine if they are invisible.

(Click here for animation)



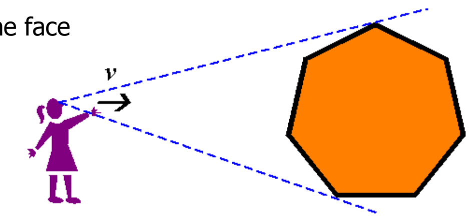
Removing Back-Faces

Idea: Compare the normal of each face with the viewing direction

Given **n**, the outward-pointing normal of F

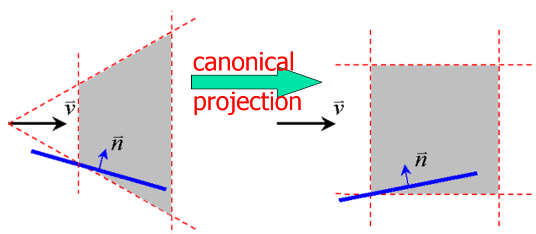
foreach face F of object
    if (**n** · **v** > 0)
        throw away the face

Does it work?



Fixing the Problem

We can't do view direction clipping just anywhere!

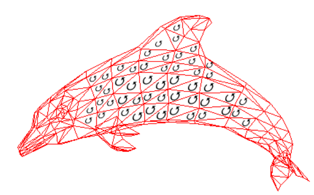


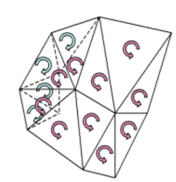
Downside: Projection comes fairly late in the pipeline. It would be nice to cull objects sooner.

Upside: Computing the dot product is simpler. You need only look at the sign of the z.

Culling Technique #2

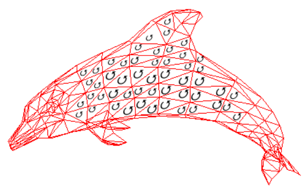
Detect a change in screen-space orientation. If all face vertices are ordered in a consistent way, back-facing primitives can be found by detecting a reversal in this order. One choice is a counterclockwise ordering when viewed from outside of the manifold. This is consistent with computing face normals (Why?). If, after projection, we ever see a clockwise face, it must be back facing.

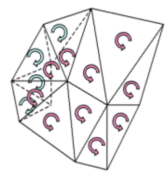




Culling Technique #2

This approach will work for all cases, but it comes even later in the pipe, at triangle setup. We already do this calculation in our triangle rasterizer. It is equivalent to determining a triangle with negative area.





Culling Plane-Test

Here is a culling test that will work anywhere in the pipeline. Remove faces that have the eye in their negative half-space. This requires computing a plane equation for each face considered.

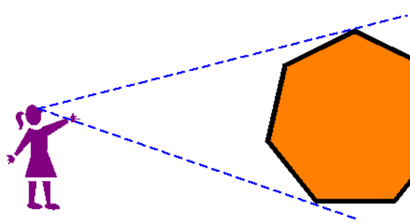
$$\begin{bmatrix} n_x & n_y & n_z & -d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

We still need to compute the normal (How?). But, we don't have to normalize it.

How do we go about computing a value for *d*?

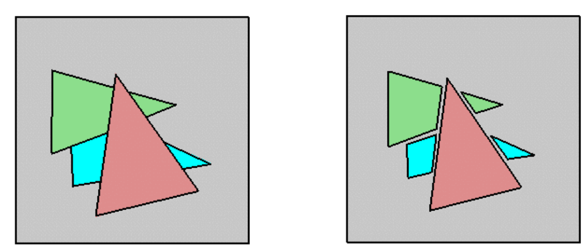
Culling Plane-Test

Once we have the plane equation, we substitute the coordinate of the viewing point (the eye coordinate in our viewing matrix). If it is negative, then the surface is back-facing.



Handling Occlusion

For most interesting scenes and viewpoints, some polygons will overlap; somehow, we must determine which portion of each polygon is visible to eye.



A Painter's Algorithm

The painter's algorithm, sometimes called depth-sorting, gets its name from the process which an artist renders a scene using oil paints. First, the artist will paint the background colors of the sky and ground. Next, the most distant objects are painted, then the nearer objects, and so forth. Note that oil paints are basically opaque, thus each sequential layer completely obscures the layer that its covers.

A very similar technique can be used for rendering objects in a three-dimensional scene. First, the list of surfaces are sorted according to their distance from the viewpoint. The objects are then painted from back-to-front.

While this algorithm seems simple there are many subtleties. The first issue is which depth-value do you sort by? In general a primitive is not entirely at a single depth. Therefore, we must choose some point on the primitive to sort by.

Implementation

The algorithm can be implemented very easily. First we extend the drawable interface so that any object that might be drawn is capable of supplying a z value for sorting.

```
import Raster;
public abstract interface Drawable {
   public abstract void Draw(Raster r);
   public abstract float zCentroid();
}
```

Next, we add the required method to our triangle routine:

```
public final float zCentroid() {
   return (1f/3f) * (vlist[v[0]].z + vlist[v[1]].z + vlist[v[2]].z);
}
```


Rendering Code

Here is a painter's method that we can add to any rendering applet:

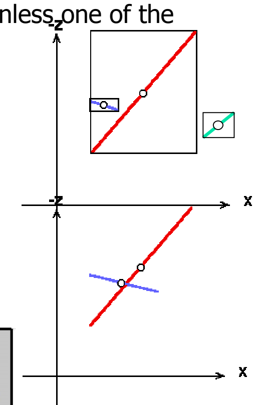
```
void DrawScene() {
   view.transform(vertList, tranList, vertices);
   ((FlatTri) riList[0]).setVertexList(tranList);
   raster.fill(getBackground());
   sort(0, triangles-1);
   for (int i = triangles-1; i >= 0; i--) {
      triList[i].Draw(raster);
   }
}
```

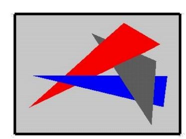
<click here for a sample applet>

Problems with Painters

The painter's algorithm works great... unless one of the following happens:

- Big triangles and little triangles. This problem can usually be resolved using further tests. Suggest some.
- Another problem occurs when the triangle from a model interpenetrate as shown below. This problem is a lot more difficult to handle. generally it requires that primitive be subdivided (which requires clipping).
- Cycles among primitives





Next Time