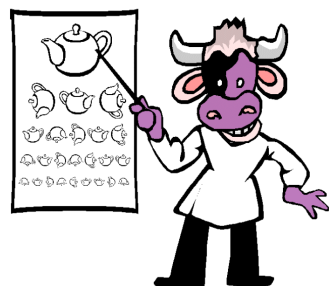


Visibility - Part I



Back-Face Culling
Painter's Algorithm

Lecture 14

Slide 1

6.837 Fall '00

Next

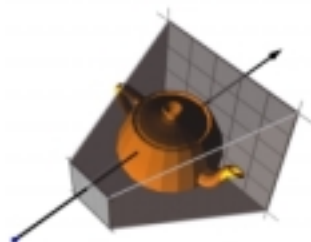
<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide01.html> [11/7/2000 4:36:53 PM]

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*.

A primitive can be invisible for one of three reasons:

1. It lies outside of the field of view
2. It is a back-facing component of a closed object
3. It is occluded by some object closer to the viewpoint



Back

Lecture 14

Slide 2

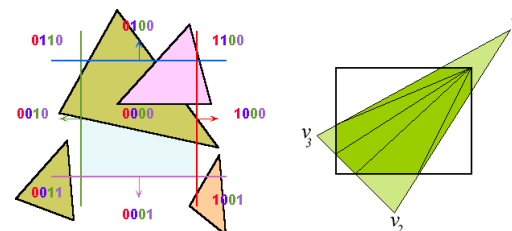
6.837 Fall '00

Next

<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide02.html> [11/7/2000 4:36:03 PM]

Outside the Field-of-View

Clipping, as we discussed the lecture before last, addresses the problem of removing those 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.



Back

Lecture 14

Slide 3

6.837 Fall '00

Next

Back-Face Culling

Back-face culling addresses 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.

We can apply back-face culling to any *orientable two-manifold*. Orientable two-manifolds have the following properties.

1. Everywhere on their surface, they are locally like a plane. They have no holes, cracks, or self-intersections.
2. Their boundary partitions 3D space into interior and exterior regions.

In our case, manifolds will be composite objects made of many primitives, generally triangles. Back-face culling eliminates a subset of these primitives & assumes that you are outside of all objects.

Back

Lecture 14

Slide 4

6.837 Fall '00

Next

<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide04.html> [11/7/2000 4:36:06 PM]

Removing Back-Faces

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

Given \mathbf{n} , the outward-pointing normal of F

```
foreach face F of object
  if ( $\mathbf{n} \cdot \mathbf{v} > 0$ )
    throw away the face
```

Does it work?


[Lecture 14](#)

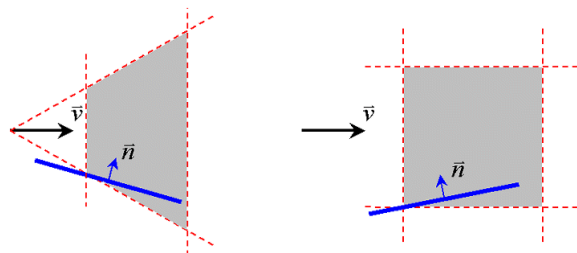
Slide 5

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide05.html> [11/7/2000 4:36:07 PM]

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 .


[Lecture 14](#)

Slide 6

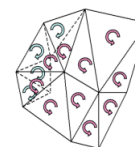
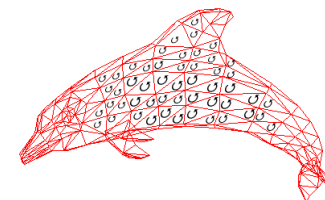
6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide06.html> [11/7/2000 4:36:09 PM]

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.



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.


[Lecture 14](#)

Slide 7

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide07.html> [11/7/2000 4:36:10 PM]

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 will 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 ?

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.


[Lecture 14](#)

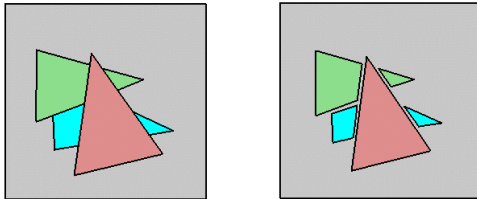
Slide 8

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide08.html> [11/7/2000 4:36:11 PM]

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.



Solving the occlusion problem used to be one of the **BIG PROBLEMS** in computer graphics.


[Lecture 14](#)

Slide 9

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide09.html> [11/7/2000 4:36:12 PM]

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 it 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.


[Lecture 14](#)

Slide 10

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide10.html> [11/7/2000 4:36:13 PM]

Implementation

```
import Raster;

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

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

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


[Lecture 14](#)

Slide 11

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide11.html> [11/7/2000 4:36:14 PM]

Sorting Code

```
//
// Use QuickSort to order the vertices from near to far
//
private void sort(int lo0, int hi0) {
    int lo = lo0;
    int hi = hi0;
    if (lo >= hi)
        return;
    float mid = triList[(lo + hi) / 2].zCentroid();
    while ((lo < hi) && (triList[lo].zCentroid() < mid)) {
        lo++;
    }
    while ((lo < hi) && (triList[hi].zCentroid() >= mid)) {
        hi--;
    }
    if (lo < hi) {
        FlatTri T = (FlatTri) triList[lo];
        triList[lo] = triList[hi];
        triList[hi] = T;
    }
    if (hi < lo) {
        int T = hi;
        hi = lo;
        lo = T;
    }
    sort(lo0, lo);
    sort((lo == lo0) ? lo + 1 : lo, hi0);
}
```


[Lecture 14](#)

Slide 12

6.837 Fall '00


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide12.html> [11/7/2000 4:36:15 PM]

Rendering Code

```
void DrawScene() {
    view.transform(vertList, tranList, vertices);
    ((FlatTri) triList[0]).setVertexList(tranList);
    raster.fill(getBackground());

    sort(0, triangles-1);
    for (int i = triangles-1; i >= 0; i--) {
        triList[i].Draw(raster);
    }
}
```

Here is a rendering method that we can add to any applet.


[Lecture 14](#)

Slide 13

6.837 Fall '00

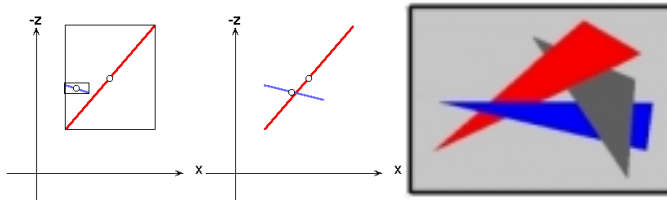


<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide13.html> [11/7/2000 4:36:17 PM]

Problems with Painters

1. Big triangles and little triangles. This problem can usually be resolved using further tests (i.e. considering the bounding boxes as well). Suggest some.
2. 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).
3. Cycles among primitives

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



Usually the painter algorithm is used only when the database does not include interpenetrating primitives. Examples of this type of model include meshes and height fields.


[Lecture 14](#)

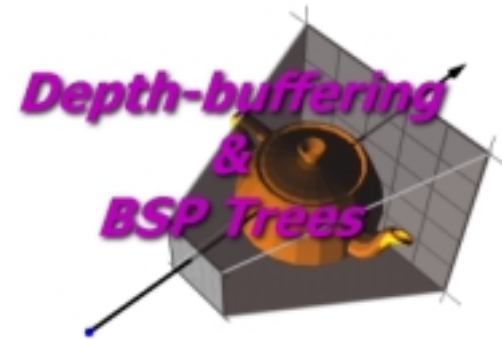
Slide 14

6.837 Fall '00



<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide14.html> [11/7/2000 4:36:19 PM]

Next Time


[Lecture 14](#)

Slide 15

6.837 Fall '00

<http://graphics.lcs.mit.edu/classes/6.837/F00/Lecture14/Slide15.html> [11/7/2000 4:36:20 PM]