Clipping and Culling

Project #3

Trivial Rejection

Outcode Clipping

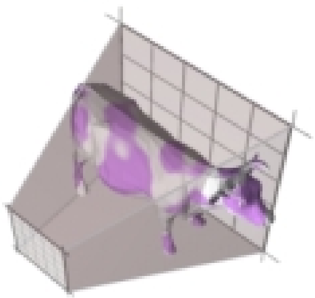
Plane-at-a-time Clipping

Culling

What is Clipping?

Two views of clipping:

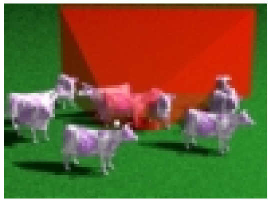
1. Clipping is a procedure for *spatially partitioning* geometric primitives, according to their containment within some region. Clipping can be used to:
   - Distinguish whether geometric primitives are inside or outside of a *viewing frustum*
   - Distinguish whether geometric primitives are inside or outside of a *picking frustum*
   - Detecting intersections between primitives



2. Clipping is a procedure for *subdividing* geometric primitives. This view of clipping admits several other potential applications.
   - Binning geometric primitives into spatial data structures.
   - Computing analytical shadows.
   - *Computing* intersections between primitives

Why Do We Clip?

- Clipping is a visibility preprocess. In real-world scenes clipping can remove a substantial percentage of the environment from consideration.
- Assures that only potentially visible primitives are rasterized. What advantage does this have over two-dimensional clipping. Are there cases where you have to clip?

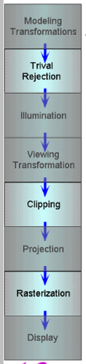


Clipping is an important optimization

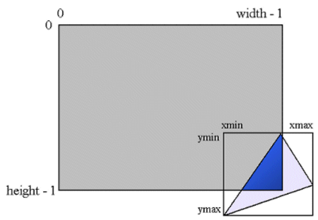
Where do we Clip?

```
xMin = (int) (v[sort[xflag][0]].x);
xMax = (int) (v[sort[xflag][1]].x + 1);
yMin = (int) (v[sort[yflag][1]].y);
yMax = (int) (v[sort[yflag][0]].y + 1);

/*
clip triangle's bounding box to raster
*/
xMin = (xMin < 0) ? 0 : xMin;
xMax = (xMax >= width) ? width - 1 : xMax;
yMin = (yMin < 0) ? 0 : yMin;
yMax = (yMax >= height) ? height - 1 : yMax;
```



There are at least 3 different stages in the rendering pipeline where we do various forms of clipping. In the trivial rejection stage we remove objects that cannot be seen. The clipping stage removes objects and parts of objects that fall outside of the viewing frustum. And, when rasterizing, clipping is used to remove parts of objects outside of the viewport.



Trivial Rejection Clipping

One of the keys to all clipping algorithms is the notion of *half-space* partitioning. We've seen this before when we discussed how edge equations partition the image plane into two regions, one negative the other non-negative. The same notion extends to 3 dimensions, but partitioning elements are *planes* rather than lines.

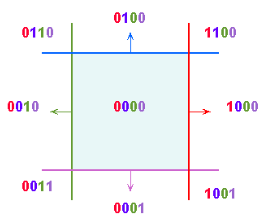
$$\begin{bmatrix} n_x & n_y & n_z & -d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

The equation of a plane in 3D is given as:

If we orient this plane so that it passes through our viewing position and our look-at direction is aligned with the normal. Then we can easily partition objects into three classes, those behind our viewing frustum, those in front, and those that are partially in both half-spaces. Click on the image above to see examples of these cases.

Outcode Clipping
(a.k.a. Cohen-Sutherland Clipping)

The extension of plane partitioning to multiple planes, gives a simple form of clipping called **Cohen-Sutherland Clipping**. This is a rough approach to clipping in that it only classifies each of its input primitives, rather than forces them to conform to the viewing window.



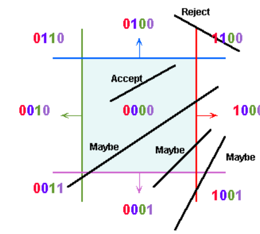
A simple 2-D example is shown on the right. This technique classifies each vertex of a primitive, by generating an *outcode*. An outcode identifies the appropriate half space location of each vertex relative to all of the clipping planes. Outcodes are usually stored as bit vectors.

By comparing the bit vectors from all of the vertices associated with a primitive we can develop conclusions about the whole primitive.

Outcode Clipping of Lines

First, let's consider line segments.

```
if (outcode1 == '0000' and outcode2 == 0000) then
  line segment is inside
else
  if (outcode1 and outcode2 == 0000) then
        line segment potentially crosses clip region
  else
        line is entirely outside of clip region
 endif
endif
```

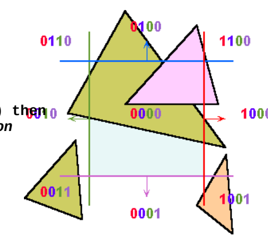


Notice that the test cannot conclusively state whether the segment crosses the clip region. This might cause some segments that are located entirely outside of the clipping volume to be subssequently processed. Is there a way to modify this test so that it can eliminate these *false positives*?

Outcode Clipping of Triangles

For triangles we need merely modify the tests so that all vertices are considered:

```
if (outcode1 == '0000' and
    outcode2 == '0000' and
    outcode3 == '0000') then
  triangle is inside
else
  if (outcode1 and outcode2 and outcode3 == '0000') then
        line segment potentially crosses clip region
  else
        line is entirely outside of clip region
 endif
endif
```



This form of clipping is not limited to triangles or convex polygons. Is is simple to implement. But it won't handle all of our problems...

Dealing with Crossing Cases

The hard part of clipping is handling objects and primitives that straddle clipping planes. In some cases we can ignore these problems because the combination of screen-space clipping and outcode clipping will handle most cases. However, there is one case in general that cannot be handled this way. This is the case when parts of a primitive lie both in front of and behind the viewpoint. This complication is caused by our projection stage. It has the nasty habit of mapping objects in behind the viewpoint to positions in front of it.

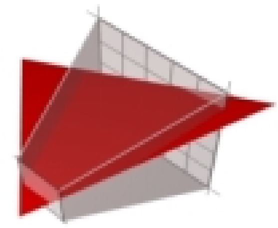
(Click above to see projection)

Plane-at-a-Time Clipping

The results of Sutherland-Hodgeman clipping can get complicated very quickly once multiple clip-planes are considered. However, the algorithm is still very simple. Each clip plane is treated independently, and each triangle is treated by one of the four cases mentioned previously.

It is straightforward to extend this algorithm to 3D.

(Click on the image below to see clipping against multiple planes.)



One-Plane-at-a-Time Clipping

(a.k.a. Sutherland-Hodgeman Clipping)

The Sutherland-Hodgeman triangle clipping algorithm uses a *divide-and-conquer* strategy. It first solves the simple problem of clipping a triangle against a single plane.

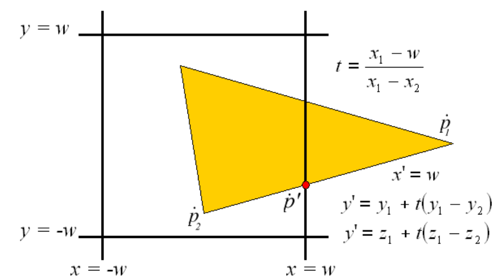
There are four possible relationships that a triangle can have relative to a clipping plane as shown in the figures on the right.

(Click on the image below to see the various clipping cases for a single plane.)

Each of the clipping planes are applied in succession to every triangle. There is minimal storage requirements for this algorithm, and it is well suited to pipelining. As a result it is often used in hardware implementations.

The Trick: Interpolating Parameters

The complication of clipping is computing the new vertices. This process is greatly simplified by using a *canonical clipping volume.*



We mentioned last lecture that it is often desireable to introduce an intermediate coordinate frame in-between eyespace and the final projection stage (i.e. the dividing through by w). In this space the viewable region is mapped into a volume that ranges from -1 to +1 in all dimensions *after projection*.

This space has several advantages. It simplifies the clipping test (all dimensions are compared against the *w* component of the vertex) and it is the perfect place in the pipeline to transistion from a floating-point to a fixed-point representation. It also simplifes the interpolation of the new vertex positions and triangle parameters as shown in the figure.

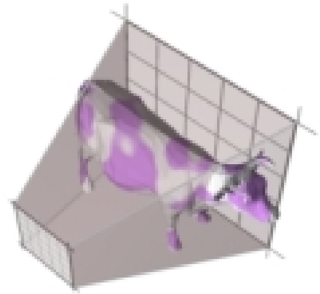
Recap of Plane-at-a-time Clipping

Advantages:

- Elegant (few special cases)
- Robust (handles boundary and edge conditions well)
- Well suited to hardware
- Canonical clipping makes fixed-point implementations manageable

Disadvantages:

- Hard to fix into O-O paradigm (Reentrant, objects spawn new short-lived objects)
- Only works for convex clipping volumes
- Often generates more than the minimum number of triangles needed
- Requires a divide per edge



Alternatives to Plane-at-a-time

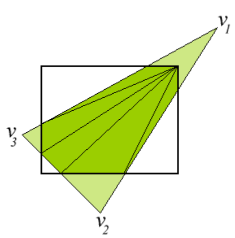
- Clipping againist concave volumes (Wieler-Atherton clipping)

Can clip abitrary polygons against abitrary polygons
Maintains more state than plane-at-a-time clipping

- Handle all planes at once (Nicholle-Lee-Nicholle clipping)

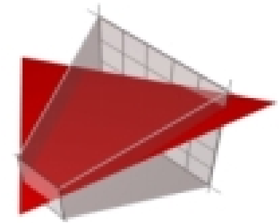
It waits before geneating triangles to reduce the number of clip sections generated.
Tracks polygon through the 27 sub-regions relative to the clip volume
Might need to generate a "corner vertex"

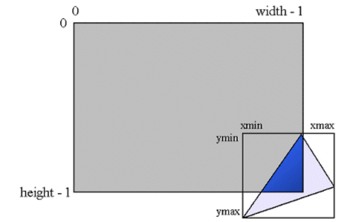
Over the years there have been several improvements to plane-at-a-time clipping.



2-D/3-D/Outcode Clipping Hybrids

Do we really need to implement 6 clipping planes?





An Example Clipper

One of the difficult aspects of clipping is fitting it into a clean conceptual model. Here we'll consider how I added clipping to our Triangle class.

```
public void Draw(Raster raster) {
  int flag = 0;

  v0 = vlist[v[0]];
  v1 = vlist[v[1]];
  v2 = vlist[v[2]];

  float near = Vertex3D.getNear();
  if (v0.z > Raster.MAXZ) flag += 1;
  if (v1.z > Raster.MAXZ) flag += 2;
  if (v2.z > Raster.MAXZ) flag += 4;
```

This first segment of code computes our outcodes.

More Clipping

```
if (flag == 0) {          // entirely inside clipping volume
    ScanConvert(raster);
  } else
  if (flag == 7) {          // entirely outside clipping volume
    return;
  } else {
    // permute triangle vertices to one of two canonical forms:
    //    1. Just v2 outside of clipping volume (flag == 4)
    //    2. v1 and v2 outside of clippling volume (flag == 6)
    if (flag == 1) {
      Vertex3D tv = v0;
      v0 = v1;
      v1 = v2;
      v2 = tv;
      flag = 4;
    } else
    if (flag == 2) {
      Vertex3D tv = v0;
      v0 = v2;
      v2 = v1;
      v1 = tv;
      flag = 4;
    } else
    if (flag == 3) {
      Vertex3D tv = v0;
      v0 = v2;
      v2 = v1;
      v1 = tv;
    } else
    if (flag == 5) {
      Vertex3D tv = v0;
      v0 = v1;
      v1 = v2;
      v2 = tv;
    }
```


Actual Clipping Code

```
float t;
v0 = normalize(v0);
v1 = normalize(v1);
v2 = normalize(v2);
if (flag == 4) {    // hither clipping yields 2 triangles
  float tx = v2.x, ty = v2.y, tz = v2.z, tw = v2.w;
  int trgb = rgb[2];

  t = (near - v1.w)/(tw - v1.w);
  Vertex3D.lerp(v2, v1, v2, t);
  rgb[2] = rgbLerp(rgb[1], trgb, t);
  v0 = normalize(v0);
  v1 = normalize(v1);
  v2 = normalize(v2);
  ClipYon(raster);

  v0 = normalize(v0);
  v1.x = v2.x; v1.y = v2.y; v1.z = v2.z; v1.w = v2.w; rgb[1] = rgb[2];
  v2.x = tx;  v2.y = ty;  v2.z = tz;  v2.w = tw;

  t = (near - v0.w)/(tw - v0.w);
  Vertex3D.lerp(v2, v0, v2, t);
  rgb[2] = rgbLerp(rgb[0], trgb, t);
  v0 = normalize(v0);
  v2 = normalize(v2);
  ClipYon(raster);
} else
```


The Other Case

```
  } else {    // hither clipping yields one triangle
     t = (near - v0.w)/(v2.w - v0.w);
     Vertex3D.lerp(v2, v0, v2, t);
     rgb[2] = rgbLerp(rgb[0], rgb[2], t);
     t = (near - v0.w)/(v1.w - v0.w);
     Vertex3D.lerp(v1, v0, v1, t);
     rgb[1] = rgbLerp(rgb[0], rgb[1], t);
     v0 = normalize(v0);
     v1 = normalize(v1);
     v2 = normalize(v2);
     ClipYon(raster);
   }
  }
}
```


Culling

Argh, I ran out of time!!!!

So we'll get back to this later when we discuss visibility.