6.837 Lecture 11

1. Clipping and Culling
2. Why Do We Clip?
3. What is Clipping?
4. Where do we Clip?
5. Trivial Rejection Clipping
6. Outcode Clipping
7. Outcode Clipping of Lines
8. Outcode Clipping of Triangles
9. Dealing with Crossing Cases
10. One-Plane-at-a-Time Clipping
11. Plane-at-a-Time Clipping
12. The Trick: Interpolating Parameters
13. Interpolating After Division
14. Recap of Plane-at-a-time Clipping
15. 2-D/3-D/Outcode Clipping Hybrids
16. Next Time

Clipping and Culling

Trivial Rejection

Outcode Clipping

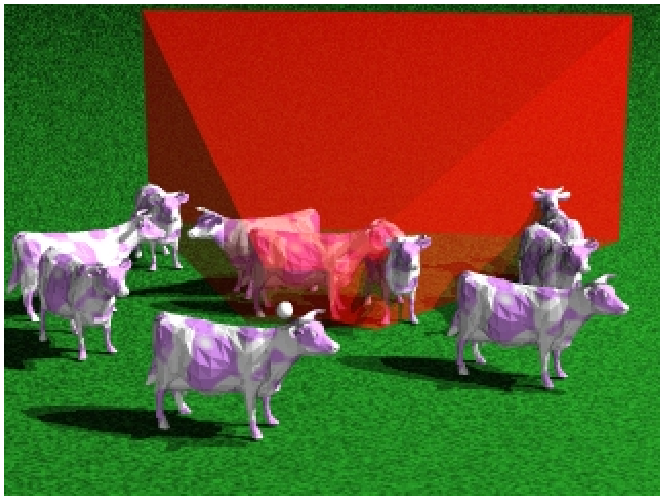
Plane-at-a-time Clipping

Culling

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?

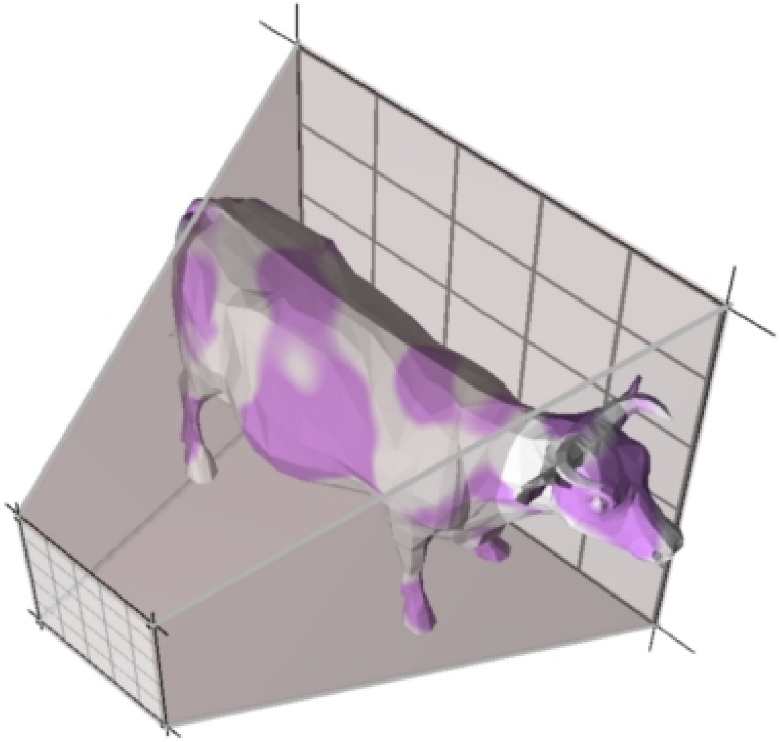


Clipping is an important optimization

What is Clipping?

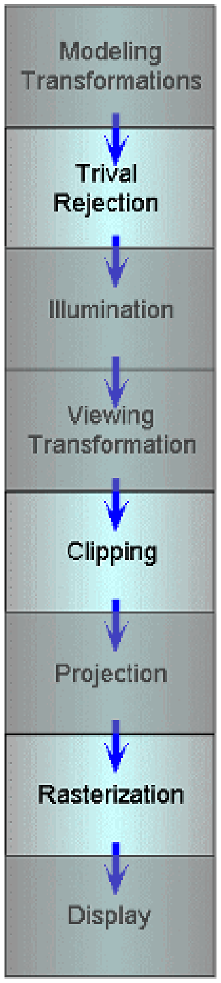
Two views of clipping:

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

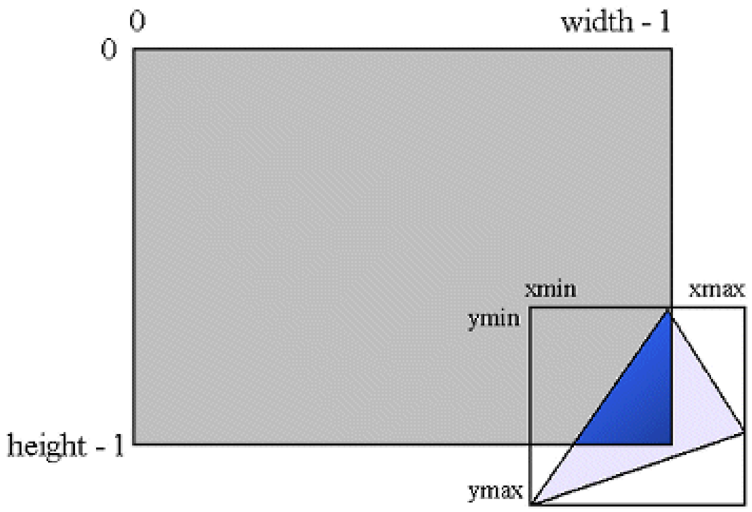


2. Clipping is a procedure for *subdividing* geometric primitives. This view of clipping admits several other potential applications.
   - Binning geometric primitives into spatial data structures.
   - Computing analytical shadows.
   - *Computing* intersections between primitives

Where do we Clip?



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 can not 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.



```
/*
   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;
```

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.

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

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

If we orient this plane so that it passes through our viewing position and our look-at direction is aligned with the normal. The 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.

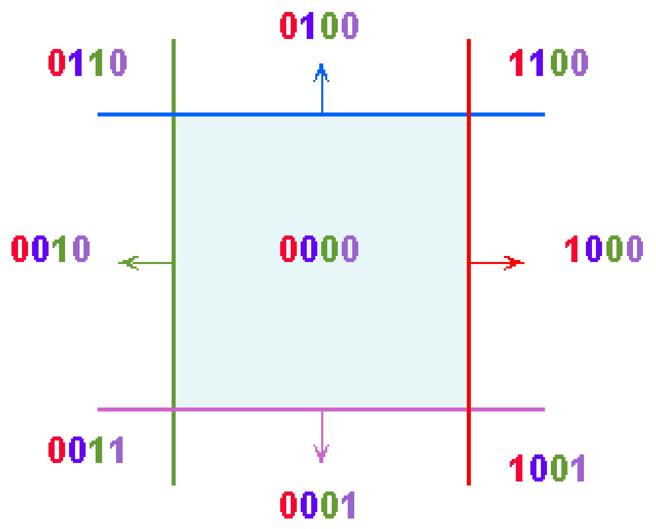
Outcode Clipping

(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 it's 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.

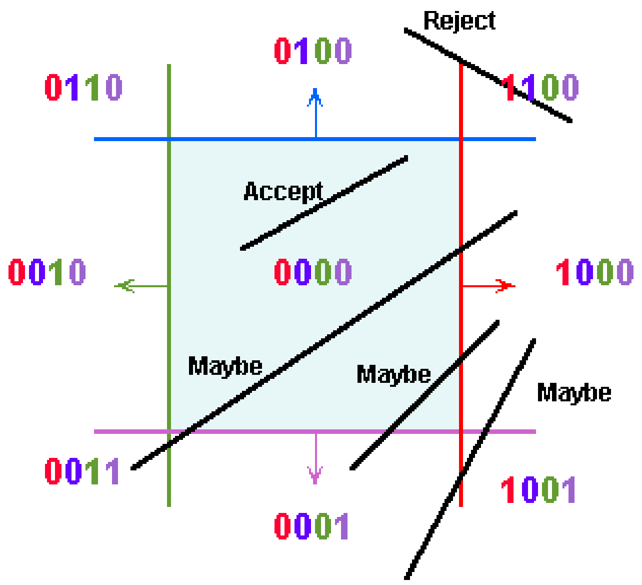


Outcode Clipping of Lines

First, let's consider line segments.

```
if (outcode1 == 0 && outcode2 == 0) then
  line segment is inside
else

if
  (outcode1 & outcode2 ! = 0) then
        line segment is outside
  else
        don't know
endif
```

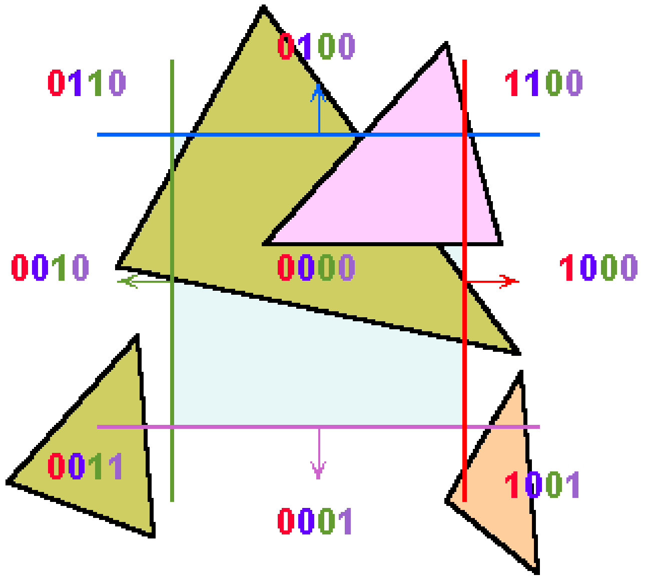


Notice that the test cannot conclusively state whether the segment crosses the clip region. This might cause some segments that are located entirely outside of the clipping volume to be subssequently processed. Is there a way to modify this test so that it can eliminate these *false positives*?

Outcode Clipping of Triangles

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

```
if (outcode1 == 0 &&
    outcode2 == 0 &&
    outcode3 == 0) then
  triangle is inside
else
  if (outcode1 & outcode2 & outcode3 != 0) then
        triangle is outside
  else
        don't know
 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 andle most cases. However, there is one case in general that cannot be handled this way. This is the case when parts of a primitive lies both in front of and behind the viewpoint. This complication is caused by our projection stage. It has the nasty habit of mapping object in behind the viewpoint to positions in front of it.

(Click above to see projection)

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.)

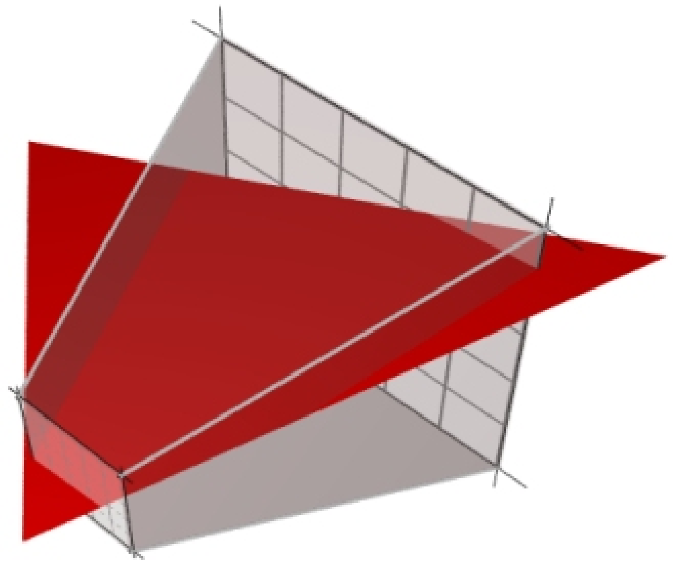
Each of the clipping planes are applied in succession to every triangle. There is minimal storage requirements for this algorithm, and it is well suited to pipelining. As a result it is often used in hardware implementations.

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 traingle is treated by on of the four cases mentioned previously.

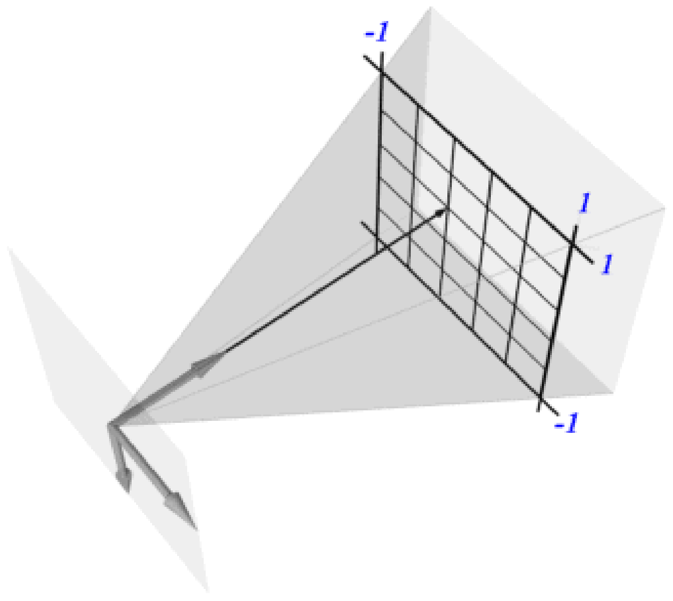
It is straightforward to extend this algorithm to 3D.

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



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 in the last lecture that it is often desireable to introduce an intermediate coordinate frame in-between the eye space and the viewport space. In the canonical space the viewable region is projected into a volume that ranges from -1 to +1 in all dimensions. Therefore, in homogeneous coordinates, before the division by w, coordinates in all dimensions range from -w to +w.



After division	Before Division (Homogenous)
$-1 \leq x \leq 1$	$-w \leq x \leq w$
$-1 \leq y \leq 1$	$-w \leq y \leq w$
$-1 \leq z \leq 1$	$-w \leq z \leq w$

The canonical space in homogenous coordinates 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.

Interpolating After Division

Suppose a line between two points (x0, y0, z0) and (x1, y1, z1) crosses the left clipping plane (x = -1). To clip the line, we have to find a point at the intersection between the clipping plane and the line. A parametric representation of the x-coordinate is $x = x_0 + t(x_1 - x_0)$. As t varies from 0 to 1 the parametric equation computes the x-coordinate of all points on the line. The calculate the parameter t at the intersection with the clipping plane we solve the equation for t such that x=-1:

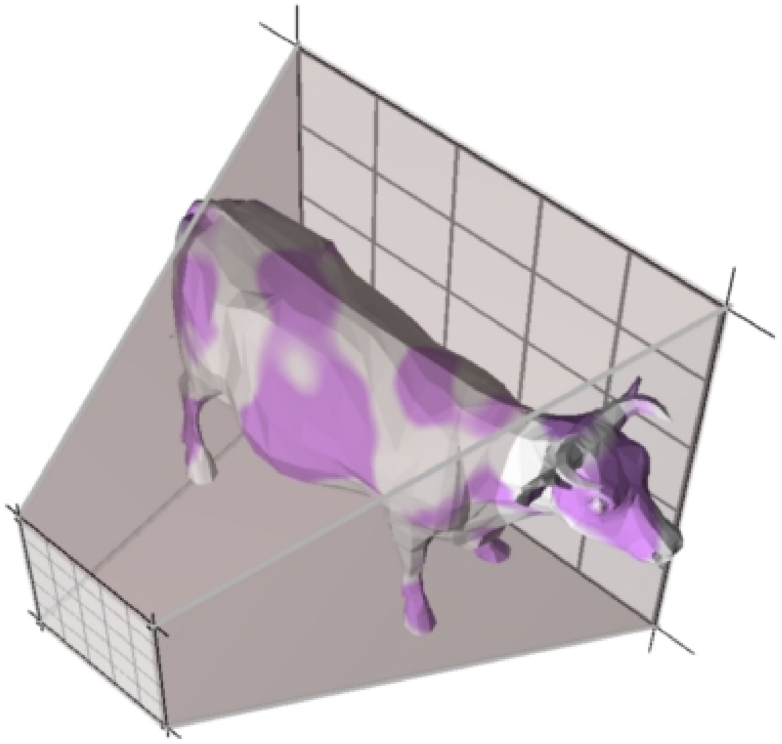
$$t = \frac{-1 - x_0}{x_1 - x_0}$$

The coordinates y and z of the intersection point are computed by substituing this value of the parameter t into the corresponding parametric equations:

$$y = y_0 + \frac{-1 - x_0}{x_1 - x_0}(y_1 - y_0), \qquad z = z_0 + \frac{-1 - x_0}{x_1 - x_0}(z_1 - z_0)$$

Recap of Plane-at-a-time Clipping

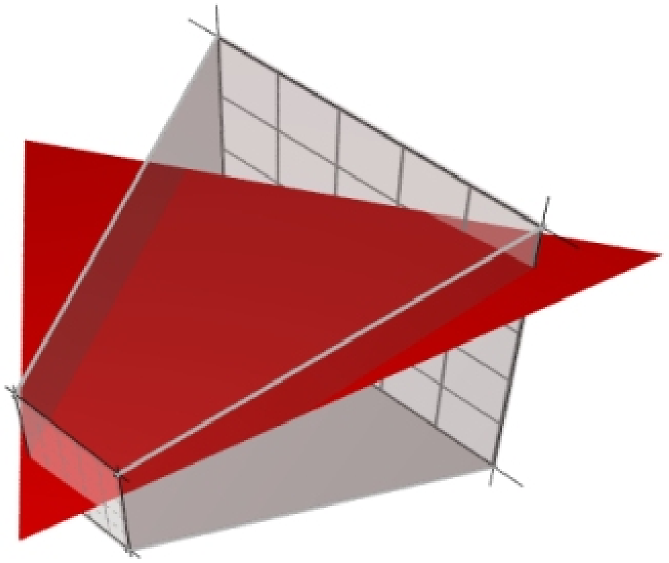
- Elegant (few special cases)
- Robust (handles boundary and edge conditions well)
- Well suited to hardware
- Canonical clipping makes fixed-point implementations manageable
- Hard to fix into O-O paradigm (Reentrant, objects spawn new short-lived objects)
- Only works for convex clippping volumes
- Often generates more than the minimum number of triangles needed
- Requires a divide per edge

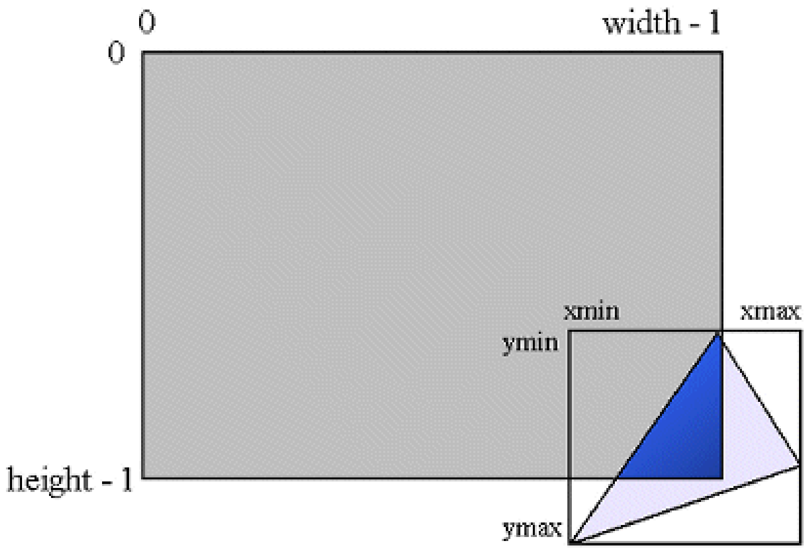


Advantages: Disadvantages:

2-D/3-D/Outcode Clipping Hybrids

Do we really need to implement 6 clipping planes?





Next Time

Surface Modeling

Types:

Polygon surfaces

Curved surfaces

Volumes

Generating models:

Interactive

Procedural