Clipping & Culling

- Trivial Rejection
- Outcode Clipping
- Plane-at-a-time Clipping
- Backface Culling
-

Lecture 11

Spring 2015

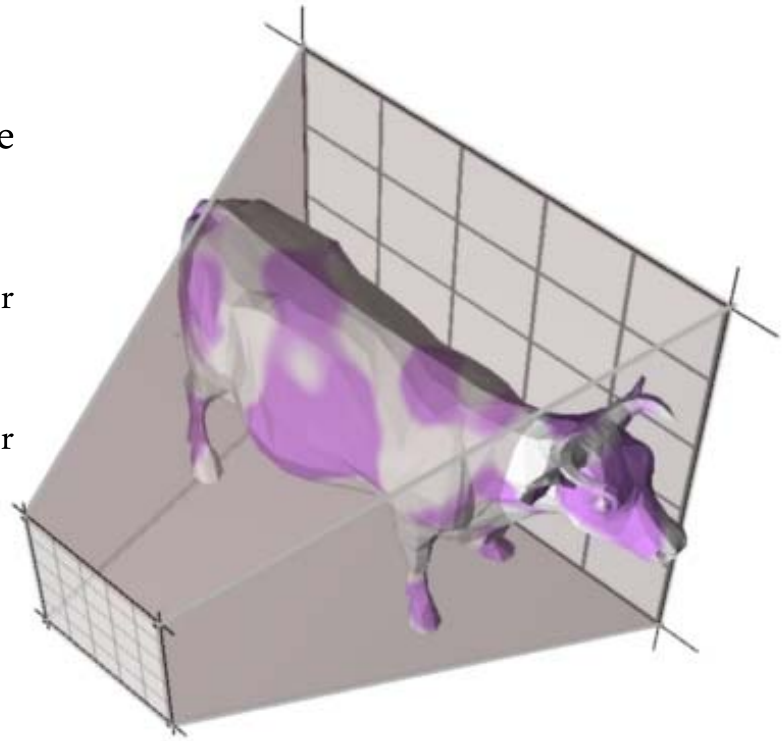
COMPUTER GRAPHICS

Department of Computer and Information Science

What is Clipping?

Clipping is a procedure for *spatially partitioning* geometric primitives, according to their containment within some region. Clipping can be used to:

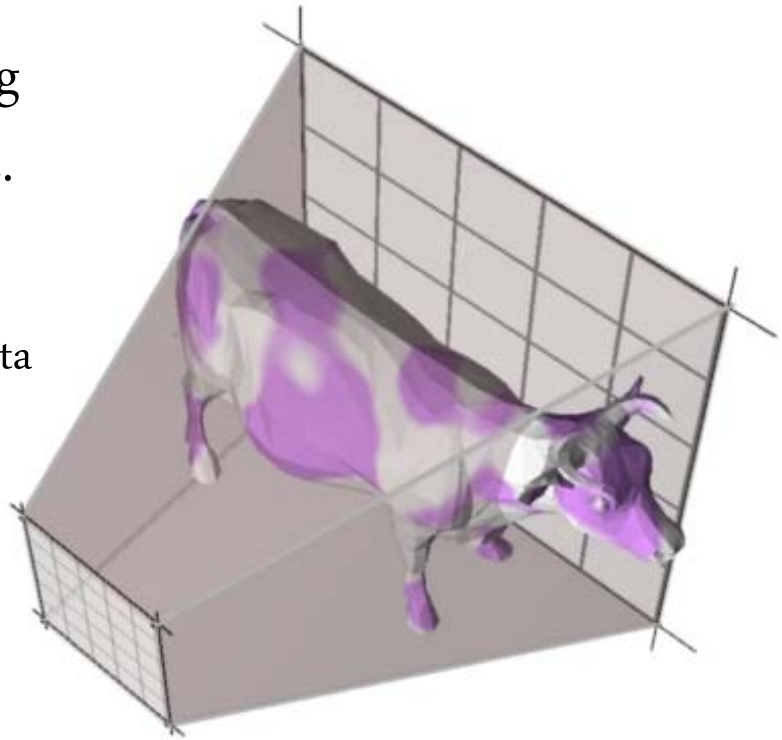
- Distinguish whether geometric primitives are inside or outside of a *viewing frustum*
- Distinguish whether geometric primitives are inside or outside of a *picking frustum*
- Detecting intersections between primitives



What is Clipping?

Clipping is a procedure for *subdividing* geometric primitives. This view of clipping admits several other potential applications.

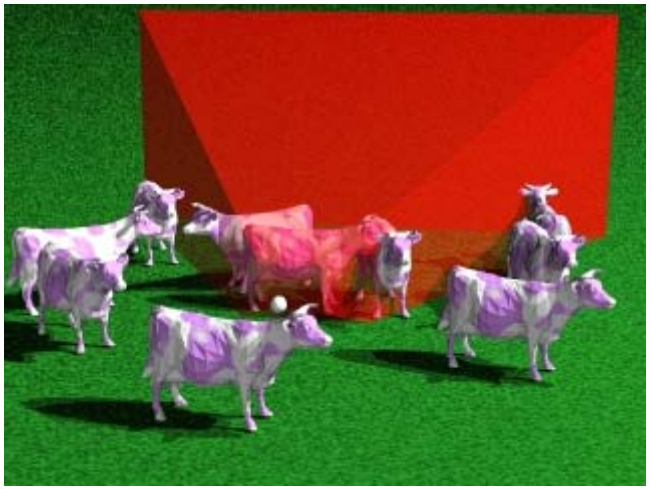
- Binning geometric primitives into spatial data structures.
- Computing analytical shadows.
- *Computing* intersections between primitives



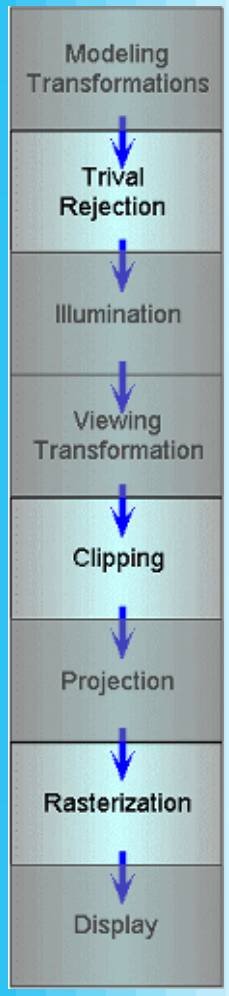
Why do we Clip?

Clipping is an important optimization

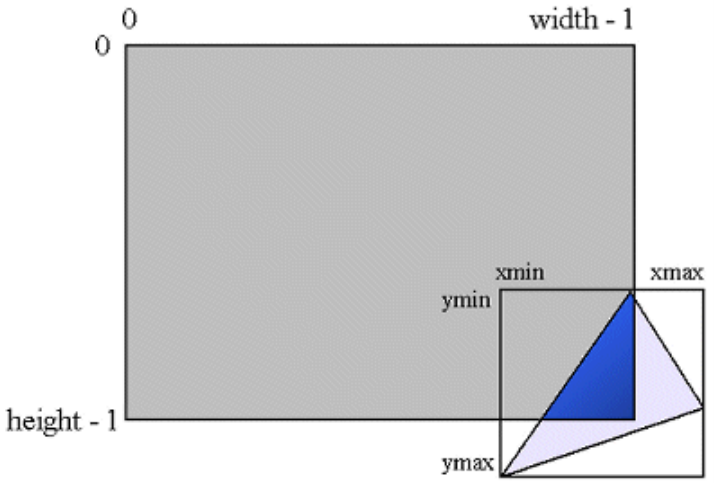
- Clipping is a visibility preprocess. In real-world scenes clipping can remove a substantial percentage of the environment from consideration.
- Assures that only potentially visible primitives are rasterized. What advantage does this have over two-dimensional clipping. Are there cases where you have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



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 view port.



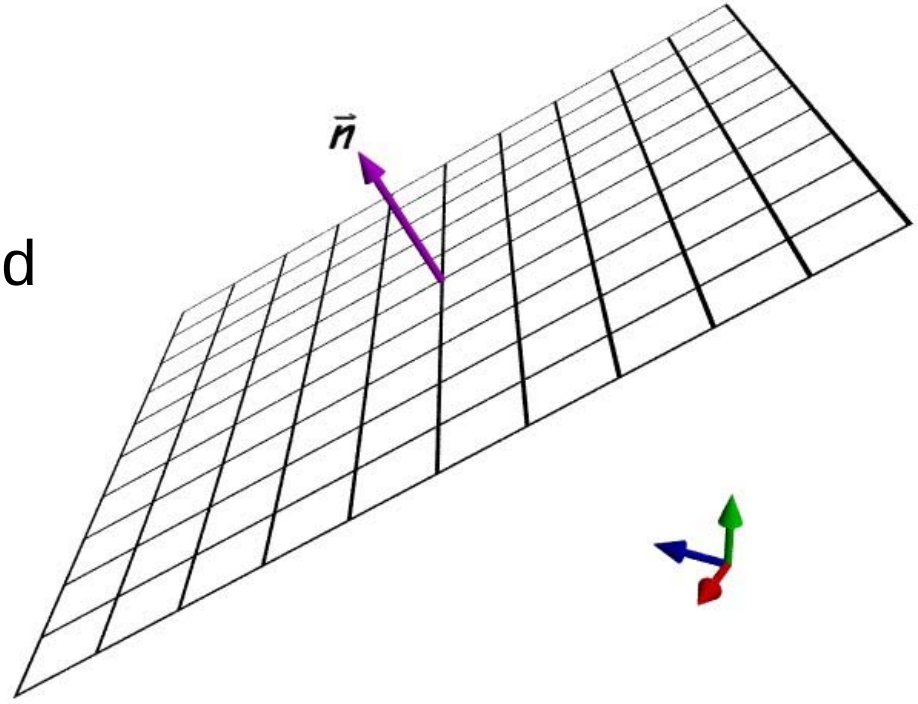

```
xMin = (int) (v[sort[xflag][0]].x);
xMax = (int) (v[sort[xflag][1]].x + 1);
yMin = (int) (v[sort[yflag][1]].y);
yMax = (int) (v[sort[yflag][0]].y + 1);

/* 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;
```

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:

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

A vector "dotted" with a point. The vector is the plane's normal. The points of the plane are a fixed distance, d, from the origin, after projection onto the normal.



Trivial Rejection Clipping

Here is a simple Example:

If we orient a plane so that it passes through our viewing position and our look-at direction is aligned with its 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.

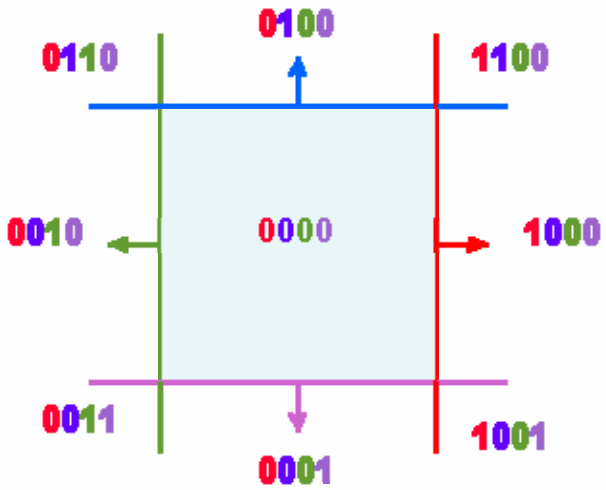
Outcode Clipping

(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.

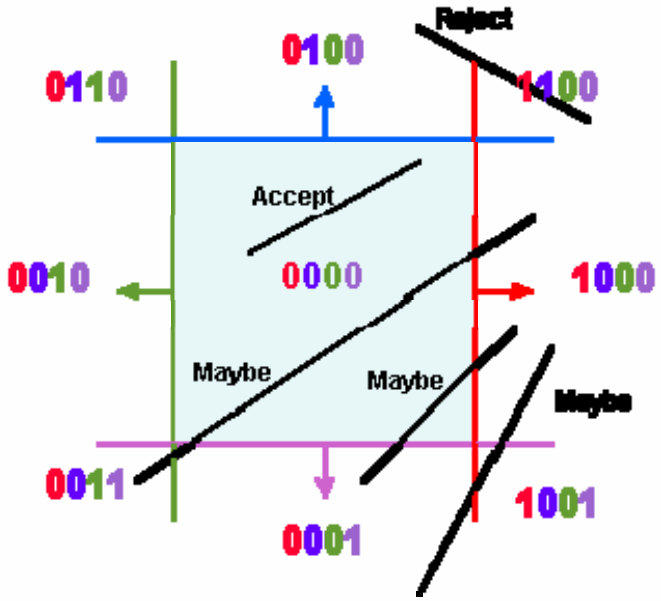
By comparing the bit vectors from all of the vertices associated with a primitive we can develop conclusions about the whole primitive.



Outcode Clipping of Lines

First, let's consider line segments.

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

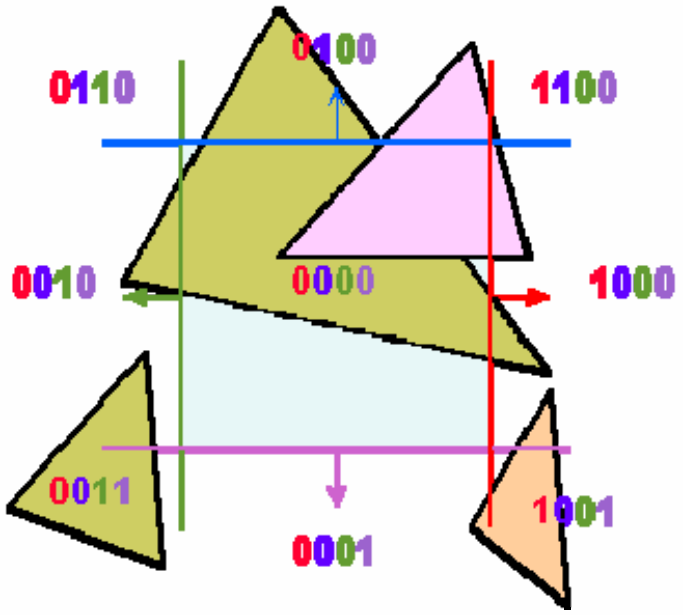


- Notice that the test cannot conclusively state whether the segment crosses the clip region. This might cause some segments that are located entirely outside of the clipping volume to be subsequently processed. Is there a way to modify this test so that it can eliminate these *false positives*?

Outcode Clipping of Triangles

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

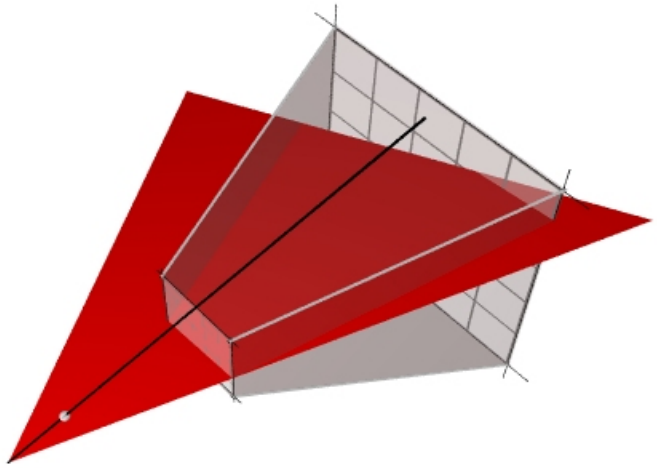
```
if (outcode1 == '0000' &&
    outcode2 == '0000' &&
    outcode3 == '0000') then
    triangle is inside
else
    if ((outcode1 & outcode2 & outcode3) == '0000') then
        line segment potentially crosses clip region
    else
        line is entirely outside of clip region
    endif
endif
```



This form of clipping is not limited to triangles or convex polygons. It 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 lies both in front of and behind the viewpoint. This complication is caused by our projection stage. It has the nasty habit of mapping objects behind the viewpoint to positions in front of it.



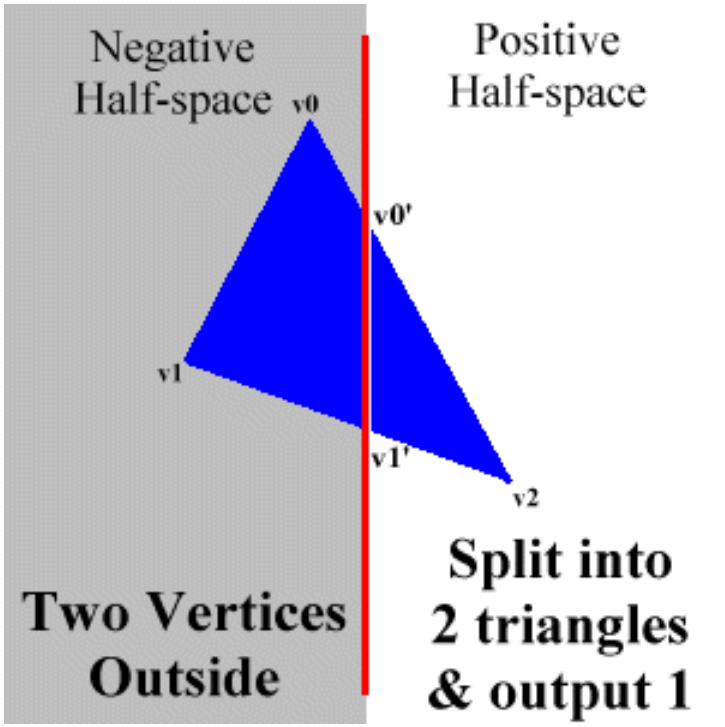
One-Plane-at-a-Time Clipping

(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.

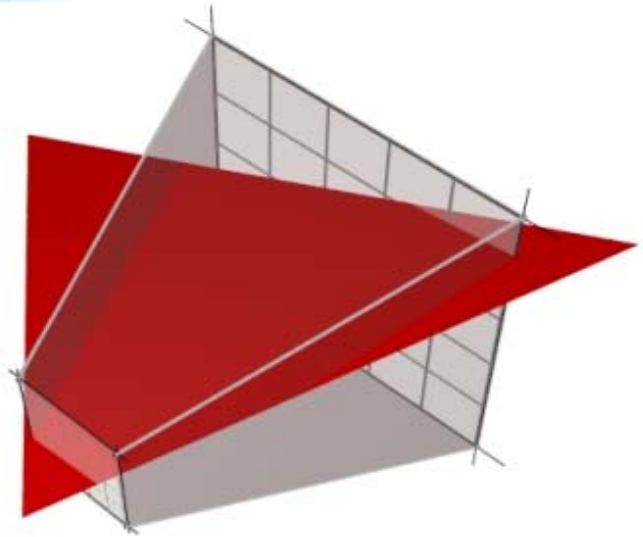
Each of the clipping planes are applied in succession to every triangle. There is minimal storage requirements for this algorithm, and it is well suited to pipelining. As a result it is often used in hardware implementations.

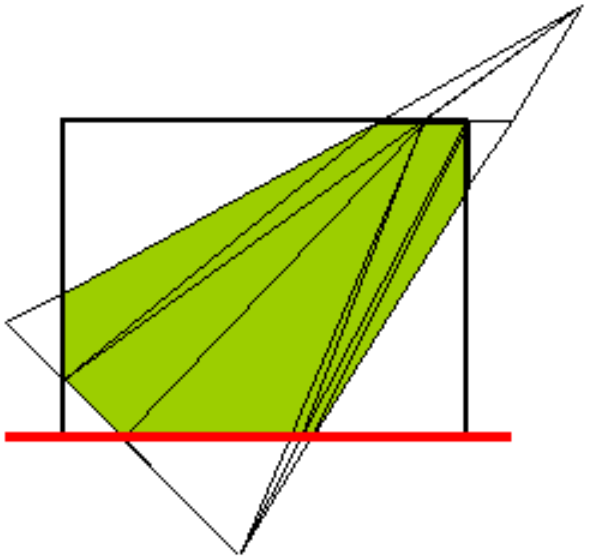


Plane-at-a-Time Clipping

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

It is straightforward to extend this algorithm to 3D.





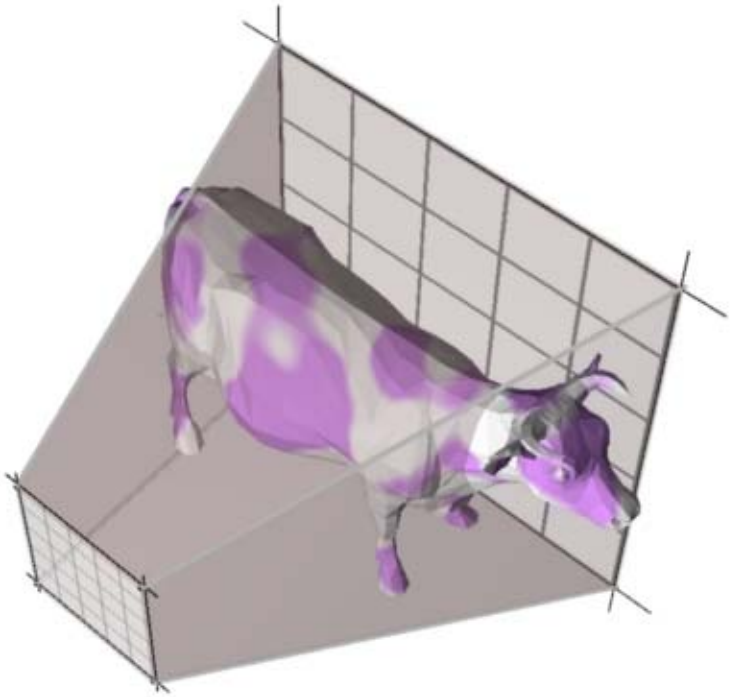
Recap of Plane-at-a-time Clipping

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 clipped edge



Alternatives to Plane-at-a-time

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

- Clipping against concave volumes (Wieler-Atherton clipping)

  Can clip arbitrary polygons against arbitrary polygons

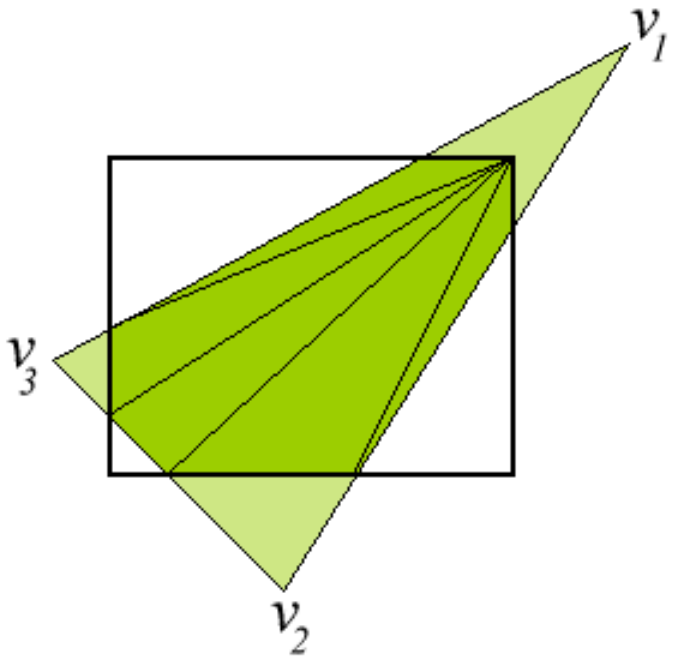
  Maintains more state than plane-at-a-time clipping

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

  It waits before generating triangles to reduce the number of clip sections generated.

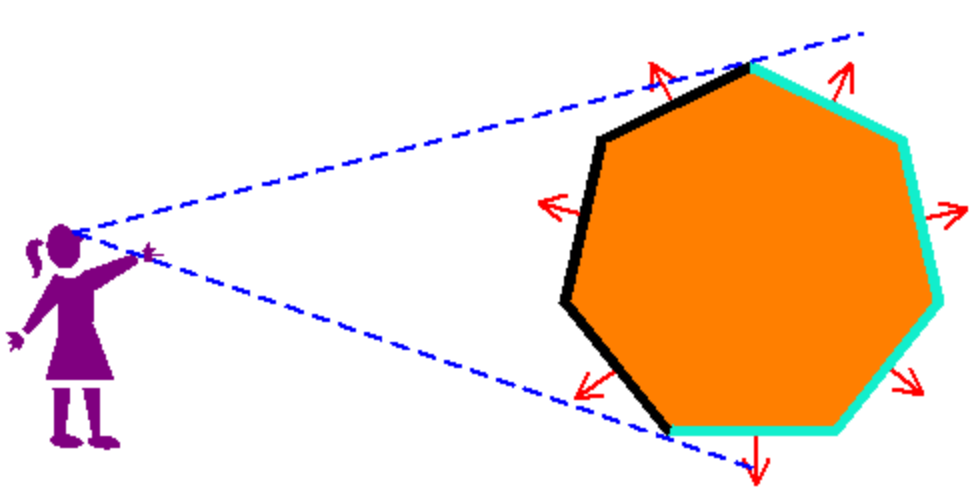
  Tracks polygon through the 27 sub-regions relative to the clip volume

  Might need to generate a "corner vertex"



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 visible.



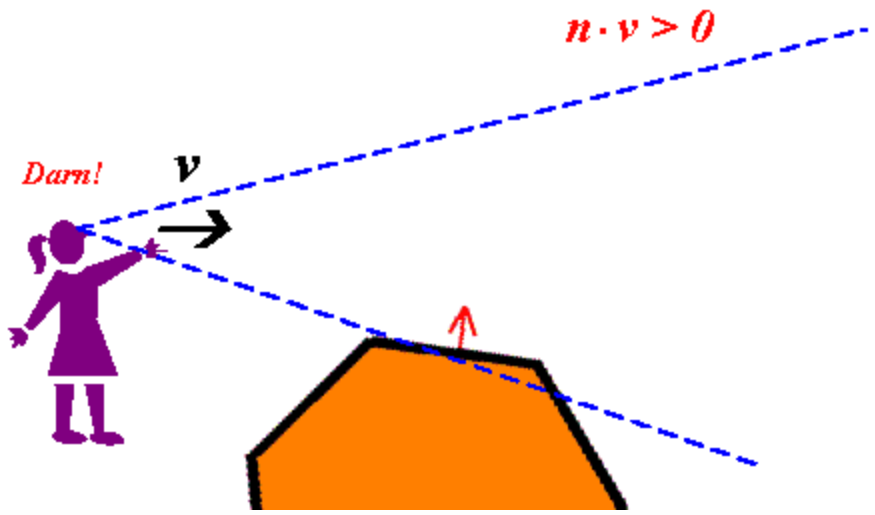
Removing Back-Faces

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

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

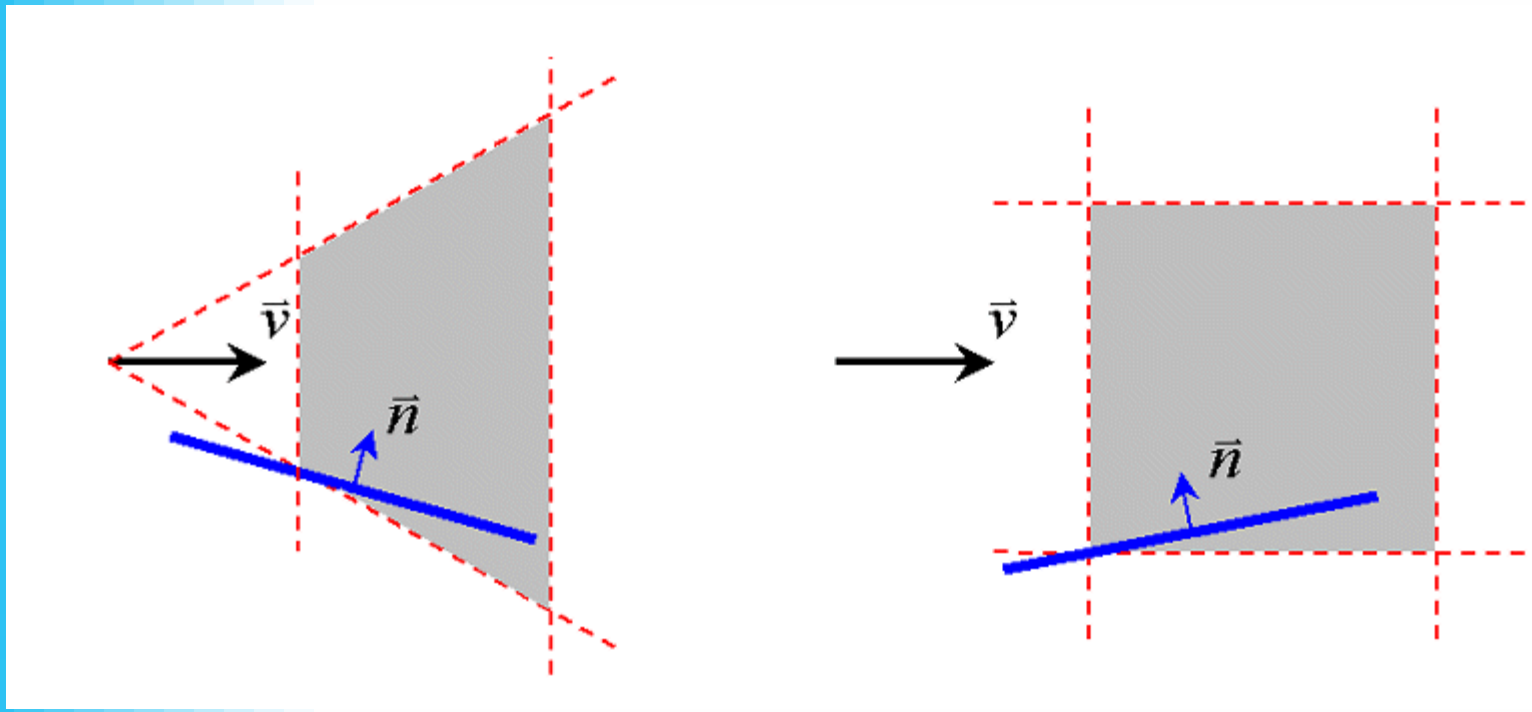
```
foreach face F of object
    if (n · v > 0)
        throw away the face
```

Does it work?



Fixing the Problem

We can't do view direction clipping just anywhere!



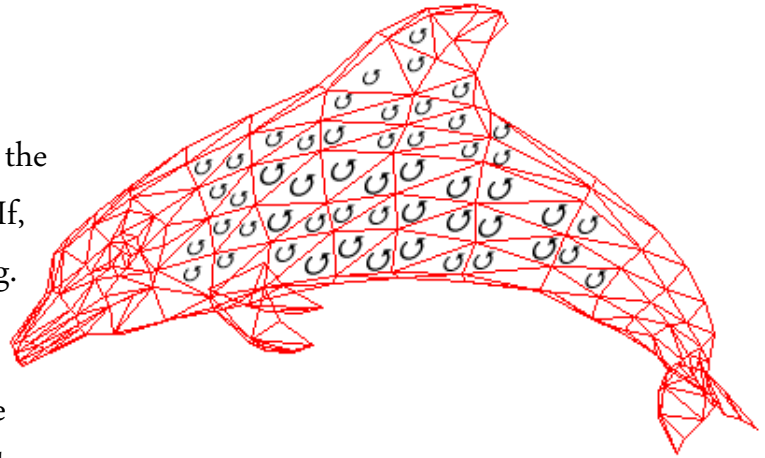
Downside: Projection comes fairly late in the pipeline. It would be nice to cull objects sooner.

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

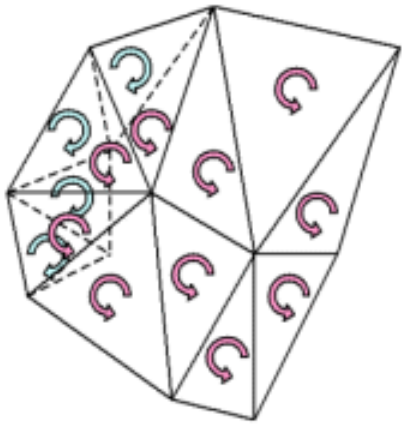
Culling Technique #2

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.



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.

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

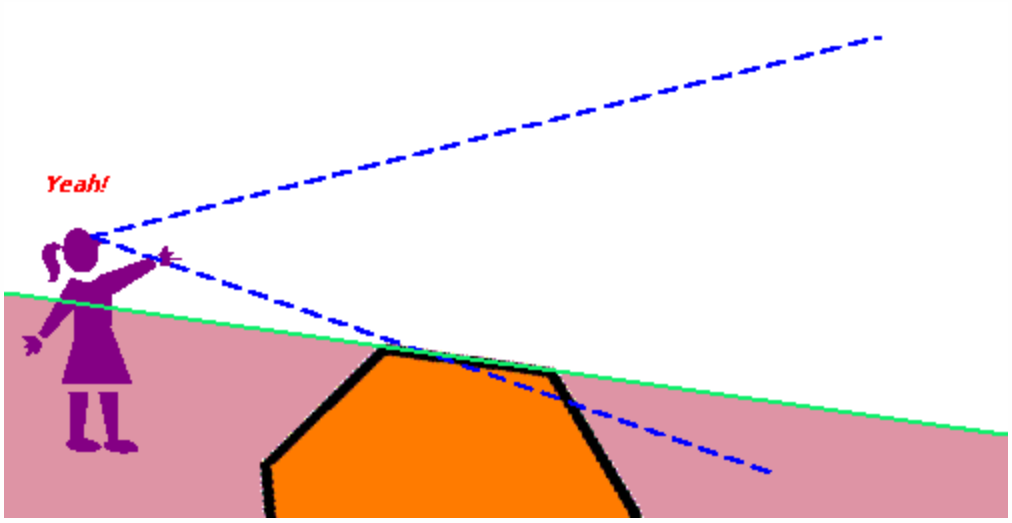
We will still need to compute the normal (How?).

But, we don't have to normalize it. (How do we go about computing a value for *d*?)

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.

Example:

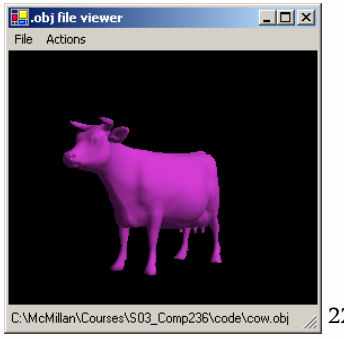


Back Face Culling in OpenGL

Back Face Culling is available as a mode setting in OpenGL.

- very flexible (can cull fronts as well as backs)
- it can double your performance!
- depends on "consistent" models

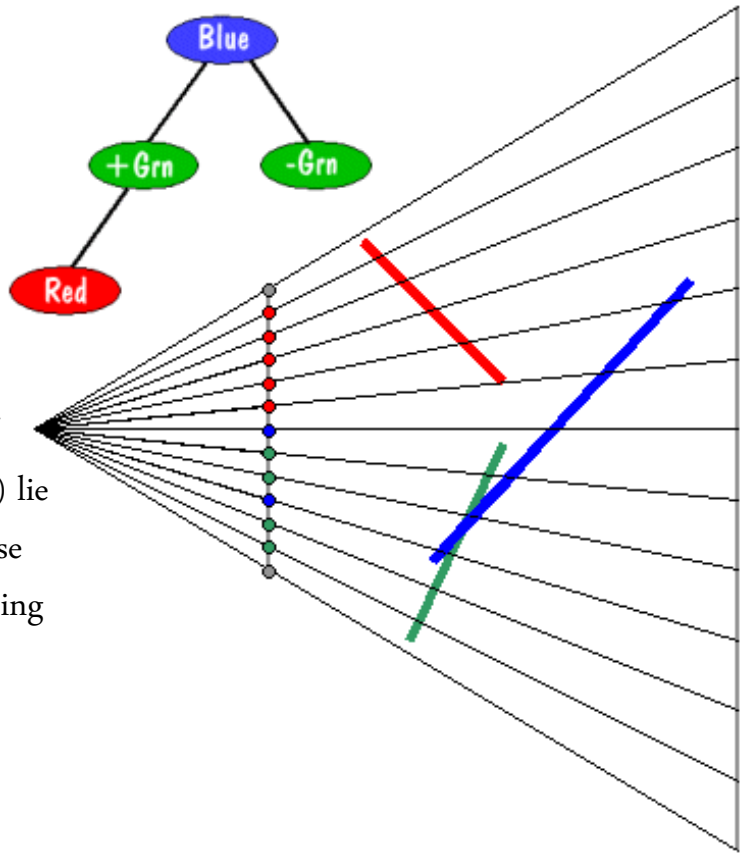
```
public void backFaceCull(bool turnon) {
        if (turnon) {
            glFrontFace(GL_CCW);          // define the orientation of front faces
            glEnable(GL_CULL_FACE);       // enable Culling
            glCullFace(GL_BACK);          // tell which faces to cull
        } else {
            glDisable(GL_CULL_FACE);      // disable Culling
        }
    }
```



I think that I shall never see...

A Binary Space Partition (BSP) tree is a simple spatial data structure:

1. Select a partitioning plane/face.
2. Partition the remaining planes/faces according to the side of the partitioning plane that they fall on (+ or -).
3. Repeat with each of the two new sets.



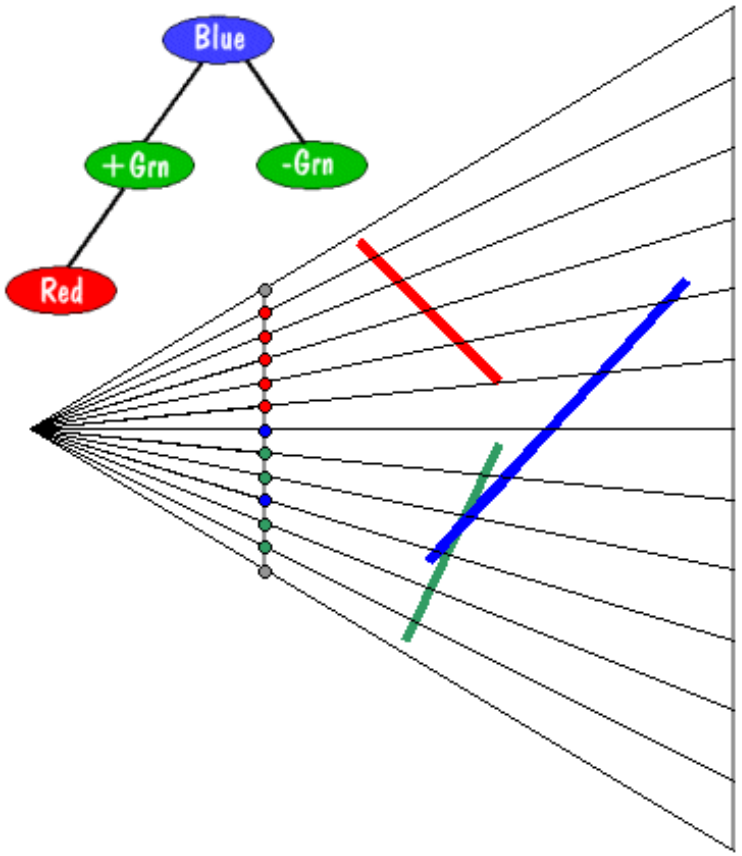
Partitioning facets:

Partitioning requires testing all facets in the active set to find if they 1) lie entirely on the positive side of the partition plane, 2) lie entirely on the negative side, or 3) if they cross it. In the 3rd case of a crossing face we clip the offending face into two halves (using our plane-at-a-time clipping algorithm).

Computing Visibility with BSP trees

Starting at the root of the tree.

1. Classify viewpoint as being in the positive or negative halfspace of our plane
2. Call this routine with the negative child (if it exists)
3. Draw the current partitioning plane
4. Call this routine with the positive child (if it exists)



Intuitively, at each partition, we first draw the stuff further away than the current plane, then we draw the current plane, and then we draw the closer stuff. BSP traversal is called a "hidden surface elimination" algorithm, but it doesn't really "eliminate" anything; it simply orders the drawing of primitive in a back-to-front order.

BSP Tree Example

Computing visibility or depth-sorting with BSP trees is both simple and fast.

It resolves visibility at the primitive level.

Visibility computation is independent of screen size

Requires considerable preprocessing of the scene primitives

Primitives must be easy to subdivide along planes

Supports Constructive Solid Geometry (CSG) modeling