Clipping

by William Shoaff with lots of help

March 12, 2002

Overview

1. PDF version of these notes

2. Audio file of these notes (up to Blinn's algorithm)

Clipping refers to the removal of part of a scene. Internal clipping removes parts of a picture outside a given region; external clipping removes parts inside a region. We'll explore internal clipping, but external clipping can almost always be accomplished as a by-product.

There is also the question of what primitive types can we clip? We will consider *line* clipping and *polygon* clipping. A line clipping algorithms takes as input two endpoints of line segment and returns one (or more) line segments. A polygon clipper takes as input the vertices of a polygon and returns one (or more) polygons. There are several clipping algorithms. We'll study the Cohen–Sutherland line clipping algorithm to learn some basic concepts. Develop the more efficient Liang–Barsky

algorithm and us its insights to culminate with Blinn's line clipping algorithm. The Sutherland–Hodgman polygon clipping algorithm will then be covered and the Weiler-Atherton algorithm, time permitting.

There are other issues in clipping that we will not have time to cover. Some of these are:

- Text character clipping
- Scissoring — clips the primitive during scan conversion to pixels
- Bit (Pixel) block transfers (bitblts/pixblts)
  - Copy a 2D array of pixels from a large canvas to a destination window
  - Useful for text characters, pulldown menus, etc.

1 Cohen–Sutherland Line Clipping

The Cohen–Sutherland algorithm clips a line to an upright rectangular window. It is an application of *triage*, or make the simple case fast. The algorithm extended window boundaries to define 9 regions:

> top-left, top-center, top-right, center-left, center, center-right, bottom-left, bottom-center, and bottom-right.

See figure 1 below. These 9 regions can be uniquely identified using a 4 bit code, often called an *outcode*. We'll use the order: left, right, bottom, top (LRBT) for these four bits. In particular, for each point $p = (x, y)$

- Left (first) bit is set to 1 when p lies to left of window
- Right (second) bit is set to 1 when p lies to right of window
- Bottom (third) bit is set to 1 when p lies below window
- Top (fourth) bit set is set to 1 when p lies above window

The LRBT (Left, Right, Bottom, Top) order is somewhat arbitrary, but once an order is chosen we must stick with it. Note that points on the clipping window edge are considered inside (the bits are left at 0).

Given a line segment with end points $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$, here's the basic flow of the Cohen–Sutherland algorithm:

1. Compute 4-bit outcodes LRBT_0 and LRBT_1 for each end-point
2. If both outcodes are 0000, the *trivially visible* case, pass end-points to draw routine This occurs when the bitwise OR of outcodes yields 0000.
3. If both outcodes have 1's in the same bit position, the *trivially invisible* case, clip the entire line (pass nothing to the draw routine). This occurs when the bitwise AND of outcodes is <u>not</u> 0000.

1001	0001	0101
1000	0000 Clip Window	0100
1010	0010	0110

Figure 1: The nine region defined by an up-right window and their outcodes.

4. Otherwise, the *indeterminate* case, – line may be partially visible or not visible. Analytically compute the intersection of the line with the appropriate window edges

Let's explore the indeterminate case more closely. First, one of two end-points must be outside the window, pretend it is $p_0 = (x_0, y_0)$.

1. Read P_1's 4-bit code in order, say left-to-right
2. When a set bit (1) is found, compute intersection point I of corresponding window edge with line from p_0 to p_1.

As an example, pretend the right bit is set so we want to compute the intersection with the right clipping window edge, also, pretend we've already done the homogeneous divide, so the right edge is $x = 1$, and we need to find y. The y value of the intersection is found by substituting $x = 1$ into the line equation (from p_0 to p_1)

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

and solving for y

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(1 - x_1).$$

Other cases are handled similarly.

Now this may not complete the clipping of the line, so we replace p_0 by the intersection point I and repeat Cohen–Sutherland algorithm. (Clearly we can save some state to avoid some computations)

- Define window by

$$\begin{aligned} x &= 0 \quad \text{Left edge} \\ x &= 1 \quad \text{Right edge} \\ y &= 0 \quad \text{Bottom edge} \\ y &= 1 \quad \text{Top edge} \end{aligned}$$

- End-points $p_0 = (1/2, 1/2)$ and $p = (1/4, 3/4)$ both have 4-bit codes 0000. Logical bitwise OR: $0000 \vee 0000 = 0000 \Rightarrow$ line is completely visible – draw line between them.
- End-point $p_2 = (3, 3)$ has 4-bit code 0101 and end-point $p_3 = (-2, 5)$ has 4-bit code 1001. Logical bitwise AND: $0101 \wedge 1001 = 0001 \neq 0000 \Rightarrow$ both end-points to right of window. Therefore line is invisible.
- End-point $p_4 = (3, 3)$ has 4-bit code 0101 and end-point $p_5 = (0, 1/2)$ has 4-bit code 0000.
  - Logical bitwise OR $0101 \vee 0000 = 0101 \Rightarrow$ no information.
  - Logical bitwise AND $0101 \wedge 0000 = 0000 \Rightarrow$ no information.
  - $p_4 = (3, 3)$ is outside.
  - Reading p_4's 4-bit code from left-to-right, "right" bit is set.
  - Intersection with right edge is at $I = (1, 4/3)$
  - I has 4-bit code ...

The Cohen–Sutherland was one of, if not, the first clipping algorithm to be implemented in hardware. Yet the intersection was not computed analytically, as we have done, but by bisection (binary search) of the line segment.

1.1 Cohen–Sutherland in 3D

The Cohen–Sutherland algorithm extends easily to 3D. Extended the 3D clipping window boundaries to define 27 regions. Assign a 6 bit code to each region, that is, for each point (x, y, z)

- Left (first) bit set (1) $\Rightarrow$ point lies to left of window
- Right (second) bit set (1) $\Rightarrow$ point lies to right of window
- Bottom (third) bit set (1) $\Rightarrow$ point lies below window
- Top (fourth) bit set (1) $\Rightarrow$ point lies above window
- Near (fifth) bit set (1) $\Rightarrow$ point lies to in front of window (near)
- Far (sixth) bit set (1) $\Rightarrow$ point lies to behind of window (far)

The Left, Right, Bottom, Top, Near, Far (LRBTNF) outcode can be used to determine segments that are trivially visible, trivially invisible, or indeterminate. In the indeterminate case we intersect the line segment with faces of clipping cube determined by the outcode of an end-point that is outside of the clipping cube. More specifically, in the indeterminant case, use parametric form of the line

$$\begin{aligned} x &= x_0 + t(x_1 - x_0) \\ y &= y_0 + t(y_1 - y_0) \\ z &= z_0 + t(z_1 - z_0) \end{aligned}$$

To clip against a face, say $y = 1$, compute

$$t = \frac{1 - y_0}{y_1 - y_0}$$

and use it to evaluate the x and z intersections

2 Liang-Barsky Line Clipping

The Liang-Barsky is optimized for clipping to an upright rectangular clip window (the Cyrus-Beck algorithms is similar but clips to a more general convex polygon). Liang-Barsky uses parametric equations, clip window edge normals, and inner products can improve the efficiency of line clipping over Cohen-Sutherland. Let

$$L(t) = p_0 + t(p_1 - p_0) = (1 - t)p_0 + tp_1, \quad 0 \leq t \leq 1$$

denote the parametric equation of the line segment from p_0 to p_1 Let $\vec{N}_e$ denote the outward pointing normal of the clip window edge e, and let p_e be an arbitrary point on edge e.

Consider the vector $L(t) - p_e$ from p_e to a point on the line $L(t)$ Figure 2 shows several of these vectors for different values of t. At the intersection of $L(t)$ and edge e the inner product of $\vec{N}_e$ and $L(t) - p_e$ is zero, see figure 2. In fact, we have

$$\begin{aligned} \vec{N}_e \cdot (L(t) - p_e) &= \vec{N}_e \cdot (p_0 + t(p_1 - p_0) - p_e) \\ &= \vec{N}_e \cdot (p_0 - p_e) + t\vec{N}_e \cdot (p_1 - p_0)) \\ &= 0 \end{aligned}$$

which if we Solve for t yields

$$t = \frac{\vec{N}_e \cdot (p_e - p_0)}{\vec{N}_e \cdot (p_1 - p_0)}.$$

(Note that checks need to be made that the denominator above is not zero.)

Using the 4 edge normals for an upright rectangular clip window and 4 points, one on each edge, we can calculate 4 parameter values where $L(t)$ intersects each edge Let's call these parameter values t_L, t_R, t_B, t_T Note any of the t's outside of the interval [0, 1] can be discarded, since they correspond to points before p_0 (when $t < 0$) and points after p_1 (when $t > 1$). The remaining t values are characterized as "potentially entering" (PE) or "potentially leaving" (PL)

- The parameter t_i is PE if when traveling along the (extended) line from p_0 to p_1 we move from the outside to the inside of the window *with respect to the edge i*.
- The parameter t_i is PL if when traveling along the (extended) line from p_0 to p_1 we move from the inside to the outside of the window *with respect to the edge i*.

See figure 3

The inner product of the outward pointing edge normal $\vec{N}_i$ with $p_1 - p_0$ can be used to classify the parameter t_i as either PE or PL.

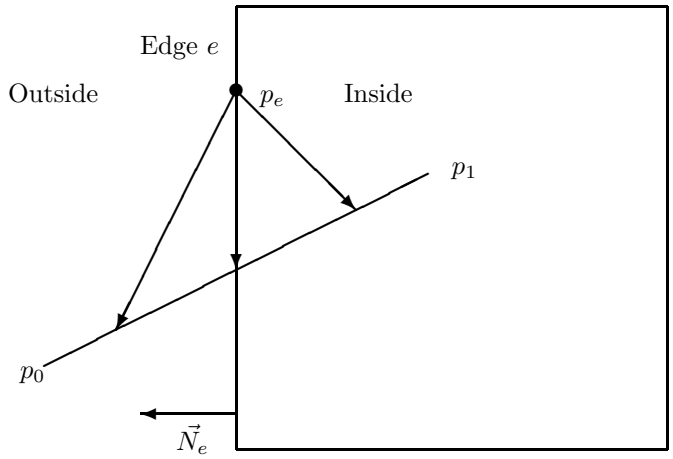


Figure 2: The setup for Liang-Barsky clipping.

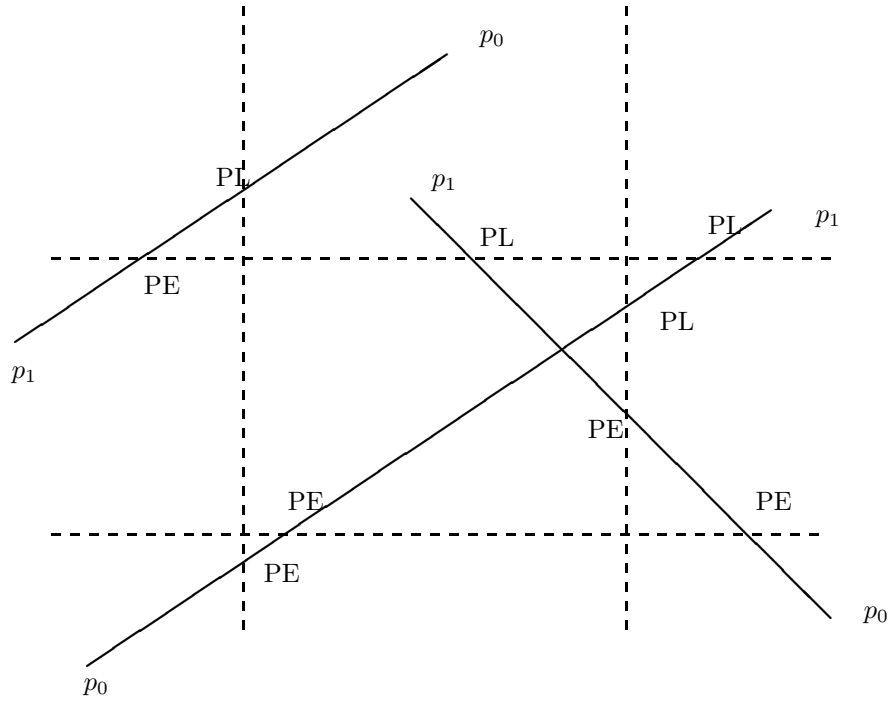


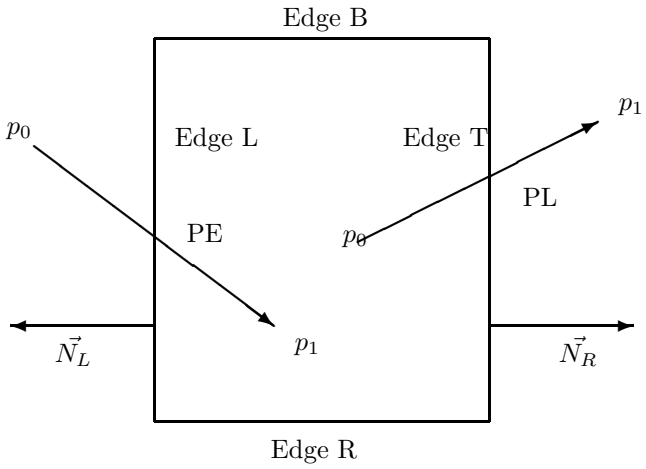
Figure 3: Potentially entering and leaving edge intersections.

1. If
$$\vec{N}_i \cdot (p_1 - p_0) < 0$$
the parameter t_i is potentially entering (PE). The vectors $\vec{N}_i$ and $p_1 - p_0$ point in opposite directions. Since $\vec{N}_i$ is outward, the vector $p_1 - p_0$ from p_0 to p_1 points inward.

2. If
$$\vec{N}_i \cdot (p_1 - p_0) > 0$$
the parameter t_i is potentially leaving (PL). The vectors $\vec{N}_i$ and $p_1 - p_0$ point in similar directions. Since $\vec{N}_i$ is outward, the vector $p_1 - p_0$ from p_0 to p_1 points outward too.

3. Let t_{pe} be the largest PE parameter value and t_{pl} the smallest PL parameter value

4. The clipped line extends from $L(t_{pe})$ to $L(t_{pl})$, where $0 \leq t_{pe} \leq t_{pl} \leq 1$



3 Blinn's Line Clipping

A clipping volume can be defined as set of bounding planes. Choosing simple planes is a good idea. We'll define the clipping volume by:

$$X = 0,\ X = 1,\ Y = 0,\ Y = 1,\ Z = 0,\ Z = 1$$

We'll call these the *real* space interpretation of the clipping volume bounding planes.

It is convenient to think of these in terms of inner products. More specifically, the $X = 0$ plane can be thought of at the column vector:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

A *homogeneous point* $(x,\ y,\ z,\ w)$ is then "in the plane" if the inner product of the point and vector is zero, that is,

$$(x,\ y,\ z,\ w) \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = x = 0$$

To obtain the *real* plane we must divide x by w (whatever its value other than 0), but we'll still get $X = x/w = 0$ since $x = 0$.

> (Recall the homogeneous divide necessary to map points after a perspective projection into real points: $X = x/w$, $Y = y/w$, $Z = z/w$.)

Using this notation, the six bounding planes become the six column vectors:

$$B_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ B_1 = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix},\ B_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$$B_3 = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix},\ B_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},\ B_5 = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

We'll call these the *homogeneous* space interpretation of the clipping volume bounding planes.
Nota bene:

1. The first three elements of each column vector form an *inward*, *unit length normal vector* to the plane in *real* space.

2. When the homogeneous coordinate (w) is equal to 1 and the inner product of a point with the bounding plane column vector is set to 0, the equation reduces to the real space equation. That is, for example,

   $$(x, y, z, 1) \cdot \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = -z + 1 = 0$$

   or

   $$z = z/1 = Z = 1.$$

3. Or, better yet, dividing the equation by w produces the real equation, for example $w - z = 0$ becomes $1 - z/w = 1 - Z = 0$ or $Z = 1$.

3.1 Boundary coordinates

Following Blinn's terminology we'll call the inner products of a point with the bounding planes the *boundary coordinates* of a point. They are listed in the table below.

Boundary Number	**(Homogeneous Value) Boundary Coordinate**	**Real Plane**
0	x	$X = 0$
1	$w - x$	$X = 1$
2	y	$Y = 0$
3	$w - y$	$Y = 1$
4	z	$Z = 0$
5	$w - z$	$Z = 1$

3.2 Outcodes

The question we want to answer is:

> Given two end points (P_0 and P_1) of a line segment, how can we use the boundary coordinates to determine a points Cohen–Sutherland outcode?

Let's assume the 6-bit outcode is in the order left, right, bottom, top, near, far (LRBTNF) so it corresponds to the order of the boundary coordinates. A particular bit is set (to 1) if the point is in the designated region and unset (to 0) otherwise.

We'll assume $w > 0$. Recall that the value of w that comes out of the perspective transform is $w = z\sin(\theta)$ where θ is (half of) the field of view angle. Now both $\sin(\theta) > 0$ and $z > 0$ (provided we're looking at something in front of us) so $w > 0$. The bottom line is division by a positive quantity does not affect the sense of an inequality!

Consider the left plane $X = 0$.

- if its boundary coordinate x is negative we're in the left region and L is set.
- if its boundary coordinate x is greater than or equal to zero, we're not in the left region and L is unset.

This works with the other planes too. Consider the right plane $X = 1$.

- if its boundary coordinate $w - x < 0$, then division by w yields $1 - X < 0$ and we're in the right region ($X > 1$) and R is set.
- if its boundary coordinate $w - x \geq 0$ we're not in the right region and R is unset.

All outcodes are set by testing inequality against 0.

$x < 0$	Set L (to 1)
$w - x < 0$	Set R
$y < 0$	Set B
$w - y < 0$	Set T
$z < 0$	Set N
$w - z < 0$	Set F

For a particular bounding plane (bit in the outcode) and a pair of line segment end points there are four cases.

Bit for P_0	**Bit for P_1**	**Interpretation**
0	0	Segment visible w.r.t. this boundary
1	0	Straddles boundary, P_0 outside
0	1	Straddles boundary, P_1 outside
1	1	Segment invisible w.r.t. this boundary

Given the outcodes LRBTNF for P_0 and P_1 (set by considering the boundary coordinates x, $w - x$, etc.) we can apply the trivial accept and reject tests:

1. If `outcode(`P_0`)` | `outcode(`P_1`)` == 000000 accept line segment. (All bits in both outcodes must be 0)
2. If `outcode(`P_0`)` & `outcode(`P_1`)` != 000000 reject line segment. (Some bit is set in both outcodes for some plane)

3.3 Interpolation

When the trivial tests fail we must calculate the intersections and this is best done by parametric linear interpolation, our old friend. Let

$$P(u) = P_0 + u(P_1 - P_0)$$

be the (directed) line segment from P_0 to P_1 as parameter u increases from 0 to 1. We need to calculate u and this is done by inner products with the boundary column vectors, say B_i, that is,

$$P(u) \cdot B_i = P_0 \cdot B_i + u(P_1 - P_0) \cdot B_i$$

and since we want the point $P(u)$ to be on the boundary this inner product should be 0. Or

$$u = \frac{P_0 \cdot B_i}{(P_0 - P_1) \cdot B_i} = \frac{P_0 \cdot B_i}{P_0 \cdot B_i - P_1 \cdot B_i}$$

but these inner products are just the boundary coordinates! To be explicit,

$$P(u) = (x_0 + u(x_1 - x_0),\, y_0 + u(y_1 - y_0),\, z_0 + u(z_1 - z_0),\, w_0 + u(w_1 - w_0))$$

and let's assume the line straddles the $X = 1$ plane, so we're working with boundary column vector

$$B_1 = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

and the inner product is

$$P(u) \cdot B_1 = -[x_0 + u(x_1 - x_0)] + [w_0 + u(w_1 - w_0)]$$

Figure 4: Crossing boundaries for $u < 0$, $0 \leq u \leq 1$ and $u > 1$

which if we set to zero and solve for u yields

$$u = \frac{w_0 - x_0}{(w_0 - x_0) - (w_1 - x_1)}.$$

Note this value of u will be in the range 0 to 1 only when boundary coordinates ($w_0 - x_0$ and $w_1 - x_1$ in the above example) have opposite signs. This is easy to interpret in real space (see figure 4). But you can also establish it algebraically by considering the inequality

$$0 < \frac{a}{a - b} < 1.$$

If a is positive, then $a - b$ must be positive and greater than a, that is b is negative. Something similar happens when a is negative. The point is we only want to calculate u when the boundary coordinates differ in sign, that is the line segment straddles the boundary.

3.4 The Algorithm

Here's the algorithm for clipping. The method `clip()` would be called twice. With the first segment end point we simply want to "move" there — it may be helpful to think in terms of a *vector* (*random scan* or *calligraphic*) display that simply moves to a location on the screen (with the electron gun turned off) then draws to a new point (with the gun on). So there are calls:

clip(P_0, "move"); // move to the first point
clip(P_1, "draw"); // draw to the second

So the first call is fairly straight forward, we calculate some boundary coordinates and outcodes, inform the next stage of the pipeline that its getting the first point of a line segment (assuming that is is visible) and save the calculated values for the next call. On the second call, we'll clip the line segment and draw the visible portion, or do nothing if it nothing is visible.

11 ⟨*Clip* 11⟩≡

```
public void clip(Hpoint p, String action) {
  ⟨Calculate boundary coordinates for p 12a⟩
  ⟨Set outcode using boundary coordinates 12b⟩
  if (action.compareTo("move") {
     ⟨Do move stuff 12c⟩
  }
  else {
     ⟨Do draw stuff 12d⟩
  }
  ⟨Copy p 14b⟩
  ⟨Copy boundary conditions 14c⟩
  ⟨Copy outcodes 14d⟩
}
```

Here we just compute the boundary coordinates using the definitions given earlier.

12a ⟨*Calculate boundary coordinates for p* 12a⟩≡ (11)

```
double[] boundaryCoord = new double[6];
boundaryCoord[0] = p.x;
boundaryCoord[1] = p.w - p.x;
boundaryCoord[2] = p.y;
boundaryCoord[3] = p.w - p.y;
boundaryCoord[4] = p.z;
boundaryCoord[5] = p.w - p.z;
```

The signs of the boundary coordinates determine the bit codes. Although not real efficient, will store them in a `boolean` array.

12b ⟨*Set outcode using boundary coordinates* 12b⟩≡ (11)

```
boolean[] outCode = new boolean[6];
if (boundaryCoord[0] < 0) outCode[0] = true;
if (boundaryCoord[1] < 0) outCode[1] = true;
if (boundaryCoord[2] < 0) outCode[2] = true;
if (boundaryCoord[3] < 0) outCode[3] = true;
if (boundaryCoord[4] < 0) outCode[4] = true;
if (boundaryCoord[5] < 0) outCode[5] = true;
```

We won't fill out this chunk of code, basically if the point is visible we just pass it through the pipe.

12c ⟨*Do move stuff* 12c⟩≡ (11)

```
if ⟨outCode is all zeros (false) (never defined)⟩ { // point is visible
  ⟨Pass point p down the pipeline as a move to point (never defined)⟩
}
```

The drawing stuff is more interesting. If we can't trivially reject the segment, we'll see if we can trivially accept it, and if not we'll do the non-trivial stuff. Of course, if we can trivially reject we do not need to do anything!

12d ⟨*Do draw stuff* 12d⟩≡ (11)

```
if ⟨Not trivial reject 13a⟩ {
   if ⟨Trivial accept 13b⟩ {
      ⟨Draw line from previous point passed down the pipeline to p (never defined)⟩
   }
   else {
      ⟨Do non-trivial stuff 13c⟩
   }
}
```

We're cheating here (when haven't I lied to you) by writing pseudo-code. We want to compute the bit-wise AND of the two outcodes. If the result is all false then we can not trivially reject the segment.

Also, we've not seen it yet but `firstOutcode` was saved from the `outCode` of the first call.

13a ⟨*Not trivial reject* 13a⟩≡ (12d)
```
firstOutCode & outCode;
```

Still cheating, the trivial accept test the bit-wise OR of two outcodes. If the result is all false then both points are in the clipping volume.

13b ⟨*Trivial accept* 13b⟩≡ (12d)
```
firstOutCode | outCode;
```

`clipCode` will tell us which boundary coordinates have opposite signs (true and false), and hence which boundaries are straddled. We'll still act as if we can just take bit-wise operations on arrays of booleans.

What we want to do is compute the last (largest) entering parameter value and the first (smallest) leaving parameter value. We'll clip against some number of clip planes so if you don't want to clip against the near and far plane set the terminating variable in the `for` loop to 4.

13c ⟨*Do non-trivial stuff* 13c⟩≡ (12d)
```
boolean[] clipCode = firstOutCode | outCode;

double uEnter = 0.0;
double uLeave = 1.0;
for (int i = 0; i < numberOfClipPlanes; i++) {
  ⟨Does segment straddle boundary? update parameters if so 14a⟩
}
if (firstOutcode != false) { // first point was outside
   Hpoint q = firstP + u*(p - firstP);
   ⟨Pass point q down the pipeline as a move to point (never defined)⟩
}
if (outCode != false) { // second point was outside
  Hpoint q = firstP + u*(p - firstP);
  ⟨Draw line from previous point passed down the pipeline to q (never defined)⟩
}
else { // second point was inside
  ⟨Draw line from previous point passed down the pipeline to p (never defined)⟩
}
```

Now if a `clipCode` element is set the corresponding boundary is straddled. So we'll compute the parameter value `u` and update the entering and leaving parameters, if appropriate.

If the first point was outside the boundary we must be entering, so if we have a larger `u` than the current entering parameter save this larger value. On the other hand, if the first point was inside, the second must be outside (after all the boundary is straddled). Thus the intersection must be a leaving one, so we'll update the leaving value if we've computed a smaller one.

If at any time we discover we've left the clipping volume before we've entered it we'll simple return.

14a ⟨*Does segment straddle boundary? update parameters if so* 14a⟩≡ (13c)

```
if (clipCode[i]) {
  u = firstBoundryCoord[i]/(firstBoundaryCoord[i] - boundaryCoord[i]);
  if ((firstOutcode[i] == true) { // first point outside this boundary
      uEnter = max (uEnter, u);
  }
  else { // first point inside, so second point outside
    uLeave = min (uLeave, u);
  }
  if (uLeave < uEnter) { // segment is invisible
    return;
  }
```

Here we just save the values of the point, boundary coordinates, and outcodes between calls to the clipper.

14b ⟨*Copy p* 14b⟩≡ (11)

```
static Hpoint firstP;
firstP = p;
```

14c ⟨*Copy boundary conditions* 14c⟩≡ (11)

```
static double[] firstBoundaryCoord;
firstBoundaryCoord = boundaryCoord;
```

14d ⟨*Copy outcodes* 14d⟩≡ (11)

```
static boolean[] firstOutcode;
firstOutcode = outCode;
```

Jim Blinn presents this material better than I can. See [2] and, in particular, [1].

4 Polygon Clipping

Polygon clipping differs from line clipping in several respects.

1. The input to the clipper is a polygon, which for simplicity we will view as a list of $n \geq 3$ vertices $(v_0, v_1, \ldots, v_{n-1})$.
2. The output from the clipper is one or more polygons.
3. The clipping process may generate vertices that do not lie on any of the edges of the orginal polygon.
4. Complex polygons (that is, non-convex) may lead to strange artifacts.

5 Sutherland–Hodgman Polygon Clipping

Since polygons are basic primitives, algorithms have been developed for clipping them directly. The Sutherland–Hodgman algorithm is a polygon clipper. It was a basic component in James Clark's "Geometry Engine," which was the precursor to the first Silicon Graphics machines. This algorithm clips any *subject polygon* (convex or concave) against any *convex clipping window*, but we will usually pretend the clipping window is an upright rectangle.

Given a *subject* polygon with an ordered sequence of vertices

$$v_1, v_2, \ldots, v_{n-1}, n \geq 3,$$

Sutherland–Hodgman compares each subject polygon edge against a single clip window edge, saving the vertices on the in-side of the edge and the intersection points when edges are crossed. The clipper is then re-entered with this intermediate polygon and another clip window edge.

Given a clip window edge and a subject polygon edge, there are four cases to consider:

1. The subject polygon edge goes from outside clip window edge to outside clip window edge. In this case we output nothing.
2. The subject polygon edge goes from outside clip window edge to inside clip window edge. In this case we save intersection and inside vertex.
3. The subject polygon edge goes from inside clip window edge to outside clip window edge. In this case we save intersection point.
4. The subject polygon edge goes from inside clip window edge to inside clip window edge. In this case we save second inside point (the first was saved previously).

To complete the description, we need to consider the first vertex of the subject polygon and its last edge. If the first vertex is inside the current edge we save it to the list of vertices in the intermediate polygon, otherwise we drop it out. For the last edge, note that if nothing has yet been saved in the intermediate polygon, the entire subject must not be visible in the clip window, so we can quit.

Otherwise, if the last subject edge crosses clip window edge, the intersection point must be appended to the intermediate polygon.

Figure 5 shows an example of the Sutherland–Hodgman clipping process. The clip window edge currently be used is solid, the others are dashed.

First Clip Window Edge: Starting with the original triangle (subject polygon), we set the intermdiate polygon to `null` and find

1. Start vertex: p_0 is outside the edge, so not saved in the intermediate polygon vertex list.
2. Subject edge p_0p_1 crosses the clip edge and so the intersection i_{01} is saved — intermediate list (i_{01}).
3. p_1 is inside the edge, so it is saved — intermediate list (i_{01}, p_1).
4. Subject edge p_1p_2 does not crosses the clip edge and so no intersection is computed.
5. p_2 is inside the edge, so it is saved — intermediate list (i_{01}, p_1, p_2).
6. Last edge p_2p_0: We have output some data, so we'll continue.
   (a) Subject edge p_2p_0 does crosses the clip edge and so the intersection i_{01} is saved — intermediate list $(i_{01}, p_1, p_2, i_{20})$.

Second Clip Window Edge:

Start vertex: i_{01} is inside the edge, so saved in the intermediate polygon vertex list — intermediate list (i_{01}).

Subject edge $i_{01}p_1$ crosses the clip edge and so the intersection i_{011} is saved — intermediate list (i_{01}, i_{011}).

p_1 is outside the edge, so it is not saved.

Subject edge p_1p_2 does crosses the clip edge and so the intersection i_{12} is saved — intermediate list $(i_{01}, i_{011}, i_{12})$.

p_2 is inside the edge, so it is saved — intermediate list $(i_{01}, i_{011}, i_{12}, p_2)$.

Subject edge p_2i_{20} does not cross the clip edge and so no interesection is computed.

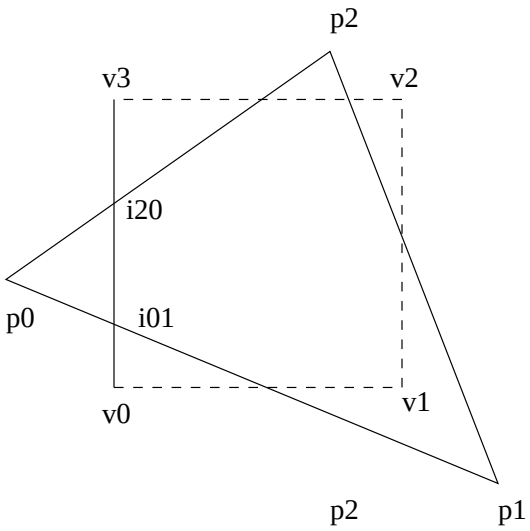
i_{20} is inside the edge, so it is saved — intermediate list $(i_{01}, i_{011}, i_{12}, p_2, i_{20})$.

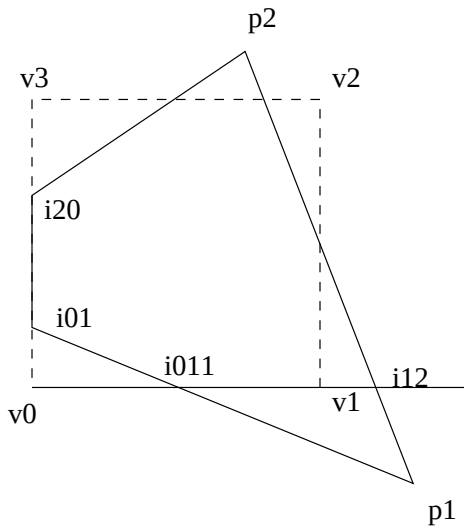
Last edge $i_{20}i_{01}$: We have output some data, so we'll continue.

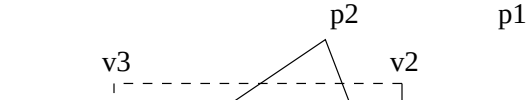
1. Subject edge $i_{20}i_{01}$ does not crosses the clip edge and so no intersection is saved.

As an exercise, you can complete the clipping process.

The Sutherland–Hodgman algorithm is not too difficult to code either.


p2
v3
v2
i20
p0
i01
v0
v1
p1


p2
v3
v2
i20
i01
i011
i12
v0
v1
p1


p2
v3
v2

5.1 Inside/Outside Testing

The Sutherland–Hodgman algorithm depends on our ability to determine that a point (vertex) is inside or outside of a given edge (line). This is a fairly common decision problem in computer graphics, yet due to the build up of floating point arithmetic errors, it may be difficult to answer the question exactly. We will declare that vertices that lie on the edge are *inside*.

There are several ways to answer the question: Is point $p = (x, y)$ on the *in* or *out* side of a line determined by vertices $v_0 = (x_0, y_0)$ and $v_1 = (x_1, y_1)$. The one we present is based on inner products, but before we can begin, we must know what is meant by *inside* and *outside*. A complete description of this topic and related one is included in the chapter on basic concepts. Recall, that our polygon vertices are listed in counter-clockwise when viewed from the front side, and this implies that the inside of the polygon is on our left as we traverse its vertices.

These assumption lead to the mathematical result that the outward edge normal from vertex v_0 to v_1 is given by

$$\vec{n_{01}} = \langle y_1 - y_0, -(x_1 - x_0) \rangle.$$

Now, consider the vector

$$p \vec{-} v_0 = \langle x - x_0, y - y_0 \rangle$$

from v_0 to p. If the inner product of $\vec{n_{01}}$ and $p \vec{-} v_0$ is

1. positive, p is on the *out* side.
2. zero, p is on the edge and consider *inside*.
3. negative, p is on the *in* side.

5.2 Crossings

We can use the inside/outside decision algorithm to determine whether or not two edges cross.

The edge crossing algorithm is called by one edge with a second edge as an input parameter. We assume an instance of `Edge` knows its first and second vertices. So we simply ask: are the first and second vertices inside the argument `edge`. If both are inside or both or outside, the edges do not cross. Put differently, if one vertex is inside and the other outside, then the edges cross.

18 ⟨*Edge crossing algorithm* 18⟩≡

```
public boolean crosses(Edge edge) {
  boolean isFirstInside = firstvertex.inside(edge);
  boolean isSecondInside = secondvertex.inside(edge);
  if ((isFirstInside && !isSecondInside) || (!isFirstInside && isSecondInside)) {
    return true;
  }
  else {
    return false;
  }
}
```

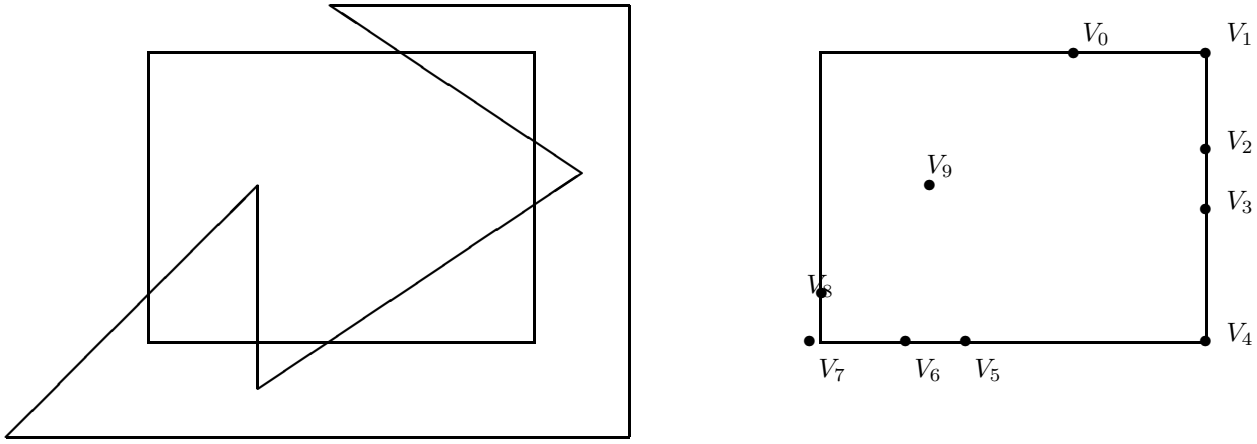
5.3 Intersections

The last task we must complete is developing the code to compute an intersection.

5.4 Sutherland–Hodgman Summary

The Sutherland–Hodgman polygon clipping algorithm clips polygons against convex clipping windows. It does so by clipping the subject polygon against each clip edge producing intermediate subject polygons. Although we have not done so, the Sutherland–Hodgman algorithm easily extends to 3 dimensions.

The Sutherland–Hodgman may produce connecting lines that were not in the original polygon. When the subject polygon is concave (not convex) these connecting lines may be undesirable artifacts.

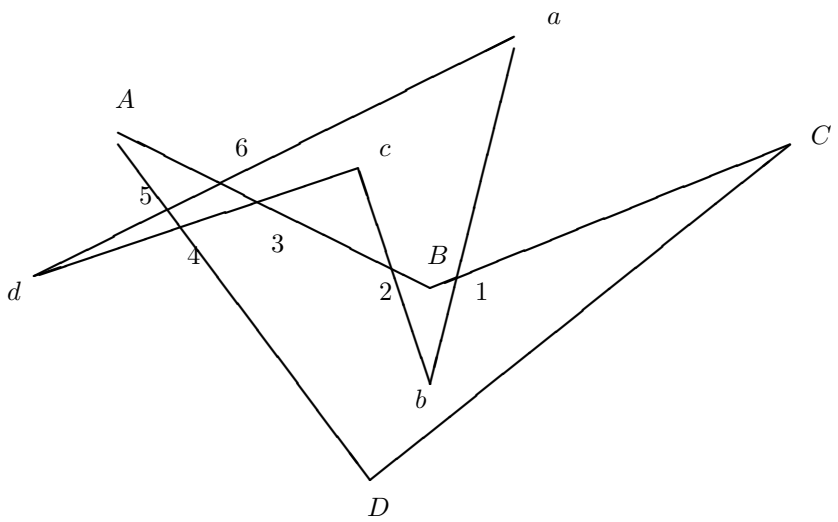


The Weiler–Atherton, which we will consider next, clips arbitrary polygons against arbitrary clipping windows, the price we pay for this generally is that Weiler–Atherton is only a 2D clipper.

6 Weiler–Atherton Polygon Clipper

- Assume the vertices of the subject polygon are listed in clockwise order (interior is on the right)
- Start at an entering intersection
- Follow the edge(s) of the polygon being clipped until an exiting intersection is encountered
- Turn right at the exiting intersection and following clip window edge until intersection is found
- Turn right and follow the subject polygon
- Continue until vertex already visited is reached
- If entire polygon has not be processed, repeat
- Consider the subject polygon with vertices a, b, c, d and the clip polygon with vertices A, B, C, D
- Insert the intersections in both vertex lists
  - Subject list: a, 1, b, 2, c, 3, 4, d, 5, 6

– Clip list: A, 6, 3, 2, B, 1, C, D, 4, 5



- Starting at vertex a of the clip polygon, find 1 is first entering intersection
- Traversing the subject, find 2 is exiting intersection
- "Jump" to vertex 2 in clip polygon, follow until vertex 1 (which has be visited)
- Output clipped list 1, b, 2, B
- Jump back to subject list, restarting at c, find 3 is entering intersection
- Traversing the subject, find 4 is exiting intersection
- Jump to vertex 4 in clip polygon, follow until vertex 5 (which is entering)
- Jump to subject, at vertex 5, find 6 is exiting
- Jump to clip, at vertex 6, find 3 is visited
- Output clipped list 3, 4, 5, 6
- All entering intersections have been visited

References

[1] J. Blinn, *A trip down the graphics pipeline: Line clipping*, IEEE Computer Graphics and Applications, 11 (1991), pp. 98 – 105.

[2] ——, *Jim Blinn's Corner: a trip down the graphics pipeline*, Morgan Kaufmann Publishers, Inc., 1996. 1-55860-387-5.