Lecture 6

Last lecture I introduced the view volume and mentioned how it is used to limit the set of points, lines, objects, etc that need to be drawn in an image. We might have a very complicated model of a scene, for example, an architectural model of many buildings, each of which contain many rooms. But at a given time, the view volume only requires us to consider a small part of the scene. In the next few lectures, we will discuss methods for discarding parts of the scene that we know won't appear in the image. These methods are called *clipping* or *culling*. (The two terms are sometimes used interchangably but there are distinctions which I'll mention as we go.)

Line Clipping

If we just want to know if a vertex lies in the view volume or not, then we apply the inequalities that we discussed last lecture. There's no more to be said. Today we'll look at the problem of clipping a *line*. This problem is more difficult because we need to do more than understand whether the endpoints are in the view volume; we have all the points along the line as well to consider.

There are three cases to think about this. One is if both end points of the line belong to the view volume. In this case, the whole line does too. A second case if the entire line lies outside the view volume. In this case, the line can be discarded or *culled*. The third case is that part of the line lies outside the view volume and part lies in it. In this case, we only want to keep the part that lies within. We *clip* off the parts of the line that lie outside the view volume. Imagine we take scissors and cut off parts of the line that poke outside the view volume. In the discussion below, I won't bother using different terminology for the second and third cases. I'll refer to them both as *clipping*.

As I mentioned at the end of last lecture, vertex clipping is done in clip coordinates (wx, wy, wz, w) by requiring that all the inequalities hold:

$$w > 0$$

$$-w \leq wx \leq w$$
$$-w \leq wy \leq w$$
$$-w \leq wz \leq w.$$

Indeed the same is true for line clipping and for clipping of other objects as well (triangle, etc). However, for the purposes of simplicity, we will follow tradition and *explain* a basic clipping algorithm in normalized device coordinates (x, y, z), so

$$-1 \leq x \leq 1$$

$$-1 \leq y \leq 1$$
$$-1 \leq z \leq 1.$$

Cohen-Sutherland

Several line clipping algorithms have been proposed over the years. I'll present one of the earliest which is from 1965 and was invented by Cohen and Sutherland[1]

We also start by considering the 2D case since it captures the main ideas and it is easier to illustrate. Suppose we have a line segment that is defined by two endpoints (x_0, y_0) and (x_1, y_1). We only want to draw the part of the line segment that lies within the window $[-1, 1] \times [-1, 1]$. How can we "clip" the line to this window? One idea is to check whether the line intersects each of the four edges of the square. A *parametric representation* of the line segment is

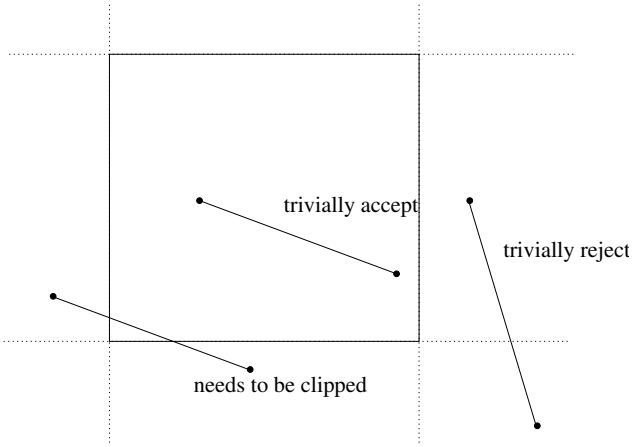
$$x(t) = x_0 + t(x_1 - x_0)$$

$$y(t) = y_0 + t(y_1 - y_0),$$

where $t \in [0, 1]$. To see if the line segment intersects an edge of the square, we could set $x(t)$ and $y(t)$ to the values at the edges of the square, e.g. we could check the intersection with the edge $x = -1$ (assuming $x_0 \neq x_1$), and solve for t which requires a division. If we find that $t \in [0, 1]$, then the line segment would intersect the edge and hence would need to be clipped. To find the clipping point, we substitute this value of t back into $(x(t), y(t))$ which requires a multiplication.

Although having to compute several divisions or multiplications per line segment might not seem like a problem (since computers these days are very fast), it could slow us down unnecessarily if we have many line segments. Let's look at a classical method (by Cohen and Sutherland) for how to clip line segments in a more efficient way.

To motivate, note that if x_0 and x_1 are both greater than 1 or both less than -1, or if y_0 and y_1 are both greater than 1 or less than -1 then the line segment from (x_0,y_0) to (x_1,y_1) cannot intersect the square. This is called *trivially rejecting* the line segment. It is possible to *trivially accept* the line segment as well: if x_0 and x_1 are both in $[-1, 1]$ and y_0 and y_1 are both in $[-1, 1]$, the line segment lies entirely within the square and does not need to be clipped.



[1]Ivan Sutherland, in particular, has made many enormous contributions to computer science `http://en.wikipedia.org/wiki/Ivan_Sutherland`.

Cohen and Sutherland's method is a way to organize the test for trivial reject/accept. Let (x, y) be a point in the plane, namely an endpoint of a line segment. Define a 4 bit binary string - called an "outcode" - to represent (x, y) relative to the clipping square.

$$\begin{aligned} b_3 &:= (y > 1) \\ b_2 &:= (y < -1) \\ b_1 &:= (x > 1) \\ b_0 &:= (x < -1) \end{aligned}$$

Note the outcode $b_3\ b_2\ b_1\ b_0$ denotes the position in the order "top, bottom, right, left."

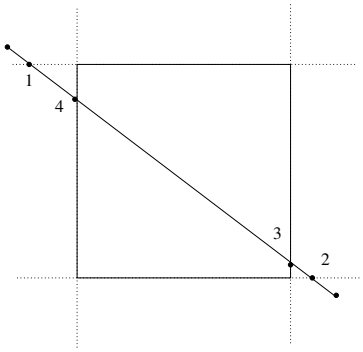
The 3×3 grid of regions showed in the sketch on the previous page have a corresponding 3×3 grid of outcodes:

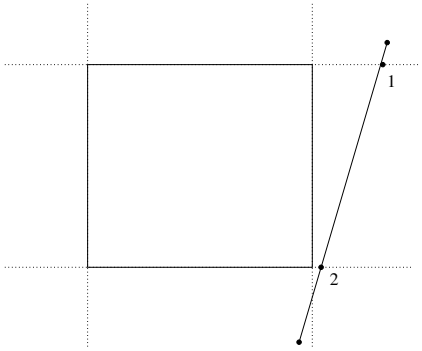
1001	1000	1010
0001	0000	0010
0101	0100	0110

We "trivially accept" a line segment if the *bitwise or* of the outcodes of the endpoints (x_0, y_0) and (x_1, y_1) is 0000. We "trivially reject" a line segment if the *bitwise and* of the two outcodes of the two points is other than 0000.

If we can neither trivially reject nor trivially accept the line segment, then we need to carry out further computation to determine whether the line overlaps the square and, if so, to clip it to the square. We sequentially clip the line segment according to the bits $b_3\ b_2\ b_1\ b_0$. If we cannot trivially reject or accept, then at least one of these bits must have the value 0 for one endpoint and 1 for the other endpoint, which means that the line segment crosses the boundary edge that corresponds to that bit. We proceed arbitrarily from b_3 to b_0 and examine the bits, clipping the line segment to the edge for that bit. Each such clip requires computing t and the new (x, y) endpoint.

The example below on the left shows that four clips might be needed using this method. The initial outcodes are 1001 and 0110. At each clip, exactly one of the 1 bits is flipped to a 0. For the example on the right, the outcodes are 1010 for the upper and 0100 for the lower vertex. After the first clip, the outcode of the upper vertex switches from 1010 to 0010. After the second clip, the bottom vertex switches from 0100 to 0010 (note the flip in the b_1 bit). Then, the line segment can be trivially rejected since the b_1 bit is 1 for both endpoints.





The Cohen Sutherland method can be applied in 3D to clip a line segment to a 3D view volume in normalized projection coordinates. One uses six bits instead of four, i.e. one needs the two additional bits

$$\begin{aligned} b_5 &:= (z > 1) \\ b_4 &:= (z < -1) \end{aligned}$$

A few technical issues are worth noting. First, recall that in OpenGL one doesn't clip in normalized device coordinates but rather one clips in clip coordinates (wx, wy, wz, w). If $w > 0$, then assigning the bitcodes is done as follows:

$$\begin{aligned} b_5 &:= (wz > w) \\ b_4 &:= (wz < -w) \\ b_3 &:= (wy > w) \\ b_2 &:= (wy < -w) \\ b_1 &:= (wx > w) \\ b_0 &:= (wx < -w) \end{aligned}$$

What about if $w < 0$? Think about it, and then see the Exercises.

Another issue arises when using clip coordinates for clipping. While the tests for outcodes are straightforward, it is not obvious how to do the clipping itself. Recall the parametric equation of the line from page 2 which you need to solve on page 2. In clipping coordinates, you don't have the x or y values but rather you have the wx or wy values. To solve for the intersection with the boundary of the view volume, you might think that you need to divide by the w coordinate, which essentially takes you to normalized device coordinates. It turns out you don't need to do this division. You can find the intersection in clip coordinates. Think about it, and see the Exercises.

Windows and viewports

We are almost done with our transformations. The last step concerns mapping to pixels on the screen i.e. in pixel space. That is, we need to talk about *screen coordinates*, also known as *display coordinates*.

A few points to mention before we discuss these coordinates. First, the display coordinates of a point are computed by mapping the normalized device coordinates (x, y, z) to a pixel position. However, only the (x, y) components are used in this mapping. The z component matters, as we'll see in the next few lectures when we discuss which points are visible. But for the discussion here, the z values don't matter.

Second, there is some inevitable confusion about terminology here, so heads up! The term *window* will be used in a number of different ways. The most familiar usage for you is the rectangular area of the screen where your image will be drawn. We'll call this the *screen window* or *display window*. This is the same usage as you are familiar with from other applications: a web browser, an X terminal, MS Office, etc. You can resize this window, drag it around, etc. Because this is most familar, let's consider it first.

The OpenGL glut commands

```
glutInitWindowPosition(x,y)
glutInitWindowSize(width,height)
```

specify this display window where your image will be drawn.[2] Typically, you can drag this window around. One often defines code to allow a user to resize the window (using the mouse), which is why the function has `Init` as part of its name. Note that this is not core OpenGL, but rather it is `glut`.

Note that a "display window" is different from the "viewing window" which we discussed in the previous two lectures, namely a window defined by `(left, right, bottom, top)` boundaries in the plane z = `-near`. We have to use the term window in both cases and keep in mind they are not the same thing.

OpenGL distinguishes a display window from a viewport. A *viewport* is a 2D region *within* a display window. At any given time, OpenGL draws images within the current viewport. You can define a viewport using

`glViewport(left,bottom,width,height).`

The parameters `left` and `bottom` are offsets relative to the bottom left corner of the display window. Of course, they are different values from the parameters with the same name that are used to define the display window in a `glFrustum` call.

The default for the viewport (i.e. no call) is the entire display window and is equivalent to

`glViewport(0,0,width,height).`

where width and height are the parameters of the display window.

Why would you want a viewport that is different from the default? You might want to define multiple non-overlapping viewports within the display window. For example each viewport might show the same scene from different viewer positions. That is what we will do in Assignment 1.

To map the normalized device coordinates coordinates to pixel coordinates on the display, OpenGL performs a *window-to-viewport* transformation. I emphasize: the term "window" here does *not* refer to the display window. Rather, it refers to the $(x, y) = [-1, 1] \times [-1, 1]$ range of the normalized device coordinates, which corresponded to the viewer's "window" on the 3D world as specified by the `glFrustum()` call.) The *window-to-viewport* mapping involves a scaling and translation. See the Exercises.

Scan conversion (rasterization)

By mapping to a viewport (or more generally, to a display window), we have finally arrived at the pixel representation. There will be alot to say about pixels later in the course – and we will need to use a more general term *fragments* at that time. For now, let's just stick to pixels and deal with a few basic problems.

[2] This is only a recommendation. You have no guarentee that your window ends up exactly there.

One basic prolem is how to represent a continuous position or set of positions – a point, a line segment, a curve – on a rectangular pixel grid. This problem is known as *rasterization* or *scan conversion*.

The term "scan" comes from the physical mechanism by which old CRT (cathode ray tube) televisions and computer screens used to display an image, namely they scanned the image from left to right, one row at a time. The term "scan conversion" should be interpreted as converting a continuous (or floating point) description of object image so that it is defined on a pixel grid (raster) – so that it can be scanned.

Scan converting a line

Consider a line segment joining two positions (x_0, y_0) and (x_1, y_1) in the viewport. The slope of the line is $m = \frac{y_1 - y_0}{x_1 - x_0}$. The line may be represented as

$$y = y_0 + m(x - x_0).$$

If the absolute value of the slope is less than 1, then we draw the line by sampling one pixel per column (one pixel per x value). Although the equation of the line (and in particular, the slope) doesn't change if we swap the two points, let's just assume we have labelled the two points such that $x_0 \leq x_1$. We draw the line segment as follows.

```
m = (y1 - y0)/(x1 - x0)
y = y0

for x = round(x0) to round(x1)
   writepixel(x,  Round(y), rgbValue)
   y = y + m
```

Notice that this method still requires a floating point operation for each y assignment since `m` is a float. In fact, there is a more clever way to draw the line which do not require floating point operations, which takes advantage of the fact that the x and y coordinates are integers. This algorithm[3] was introduced by Bresenham in the mid 1960s.

The above algorithm assumes that we write one pixel on the line for each x value. This makes sense if $|m| < 1$. However, if $|m| > 1$ then it can lead to gaps where no pixel is written in some rows. For example, if $m = 4$ then we would only write a pixel every fourth row! We can avoid these gaps by changing the algorithm so that we loop over the y variable. We assume (or relabel the points so) that $y_0 \leq y_1$. The line can be written as $x = (y - y_0)/m + x_0$. It is easy to see that the line may be drawn as follows:

```
x = x0
y = y0;
for y = y0 to y1 {
```

[3] I am not covering the details because the tricks there (although clever and subtle) are not so interesting when you first see them and they don't generalize to anything else we are doing in the course (and take some time to explain properly). If you are interested, see `http://www.cim.mcgill.ca/~langer/557/Bresenham.pdf`

```
    writepixel(Round(x),  y).   //  RGB color to be determined elsewhere...
    x  = x  +  1/m;
}
```

Scan converting a triangle (more generally, a polygon)

Suppose we want to scan convert a simple geometric shape such as a triangle. Obviously we can scan convert each of the edges using the method described above. But suppose we want to fill in the interior of the shape too. In the case of a triangle, this is relatively simple to do. Below is a sketch of the algorithm. For more complicated polygons, the two steps within the loop require a bit of work to do properly. For example, a data structure needs to be used that keeps a list of the edges, sorted by position with the current row. I'm omitting the details for brevity's sake. Please see the sketches drawn in the slides so you have a basic understanding of what is required here. We will return to this problem later in the course when we will say *much* more about what it means to fill a polygon.

```
examine vertices of the polygon and find ymin and ymax
//  if they don't have integer coordinates, then round them

for y = ymin to ymax
    compute intersection of polygon edges with row y
    fill in between adjacent pairs of edge intersections
```