Dire Dawa Institute of Technology
Department of Computer Science

Chapter 6 Handout –The Viewing Pipeline (part II)
Clipping

Compiled by Gaddisa Olani, 2017

1. Introduction

In the previous chapter we introduced the concept of a *viewing pipeline* in graphics. Part of this pipeline was the *normalisation and clipping* process. In this chapter we will examine this process in more detail.

We can define clipping as removing portions of a picture that are outside of a specified region of space. The most common use of clipping is to remove (parts of) primitives that are outside of the *view volume* of the virtual camera in the graphics pipeline. The shape of the view volume will depend on the type of projection transformation used, but always the volume can be defined by a number of *clipping planes*. For each clipping plane, if the primitive lies on one side it will be saved; if it lies on the other side it will be rejected; and if it intersects with the plane it will need to be *clipped*. Recall that the projection transformation normally preserves the *z*-coordinate for later use, so clipping takes place in a normalised (and projected) 3-D coordinate system. This means that the *x*, *y* and *z* coordinates will all vary between 0 and 1 (or for some graphics packages, -1 and +1).

2. Types of Clipping Algorithm

Depending on when clipping takes place relative to *scan conversion* we can identify 3 different types of clipping algorithm. (Scan conversion refers to the digitising of picture information for storage in the frame buffer.)

2.1 Clipping Before Scan Conversion

The most common approach is to perform clipping before scan conversion. This is known as *analytical clipping*. Analytical clipping algorithms operate on object primitives such as points, lines and polygons, and this is the most efficient type of clipping for such primitives. For the rest of this chapter we will concentrate on analytical clipping algorithms.

2.2 Clipping During Scan Conversion

Clipping during scan conversion is known as *scissoring*. In this case, scan conversion is performed for each primitive, but before the pixel values are stored in the frame buffer a decision is made as to whether or not the pixel should be clipped. As such, we can view scissoring as a "brute-force" technique, since it scan converts every primitive, regardless of whether it lies

inside the view volume or not. Generally this approach is less efficient than analytical clipping. However, because we do not need to perform any processing before scan conversion, scissoring can be more efficient than analytical clipping if the primitive lies mostly inside the clipping region.

2.3 Clipping After Scan Conversion

Clipping can occur after scan conversion if we first scan convert all primitives into a temporary canvas and then copy the pixels inside the clipping region from the temporary canvas into the actual frame buffer. Again, this is a "brute force" technique because every primitive is drawn regardless of whether it lies inside the view volume. Therefore generally this is a less efficient technique for primitives such as points, lines and polygons. However, it is a simple and easy approach, and it is sometimes used when clipping text primitives.

3. 2-D Clipping

Now we will examine a number of analytical clipping techniques for different types of primitives. Clipping can be done in 2-D or 3-D, depending on whether we are using 2-D or 3-D coordinates. In this section we will examine some techniques for performing 2-D clipping, but many 2-D clipping algorithms are generalisable to 3-D. When performing clipping in 2-D we need to consider four clipping planes: left, right, bottom and top. In the following sections we will examine some algorithms for point clipping, line clipping and polygon clipping.

3.1 2-D Point Clipping

Many graphics packages include point primitives, and points can be used to construct higher-level primitives such as lines or polygons. Therefore it is important to have an efficient technique for clipping points against a clipping rectangle. This is the simplest type of clipping, and it can be performed by testing the following inequalities:

- $x_{min} \leq x \leq x_{max}$
- $y_{min} \leq y \leq y_{max}$

If any of these inequalities is not satisfied, then the point is rejected. If all are satisfied then the point is saved. This is illustrated in Figure 1.

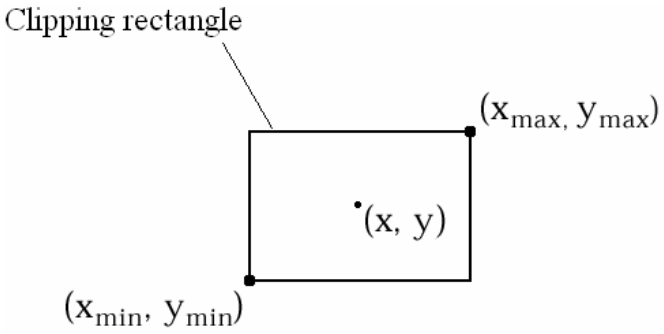


Figure 1 - 2-D Point Clipping

3.2 2-D Line Clipping

With point primitives there were only two possible outcomes of the clipping process: *accept* or *reject* the point. For other primitives we have three possible outcomes: *accept* the entire primitive, *reject* the entire primitive, or *clip* the primitive so that part of it is accepted and part rejected.

Line primitives can be represented by the coordinates of their two end-points. Depending on the positions of these end-points relative to the four clipping boundaries we may be able to make an immediate decision as to which of the three outcomes is appropriate. Consider Figure 2. This shows a number of possible positions for line primitives relative to four clipping boundaries. For some of these lines (e.g. P_1P_2) we should be able to decide straight away that the entire primitive should be accepted. For others (e.g. P_3P_4) we can decide to reject the entire primitive. To make a decision for the other lines we need to do some more work.

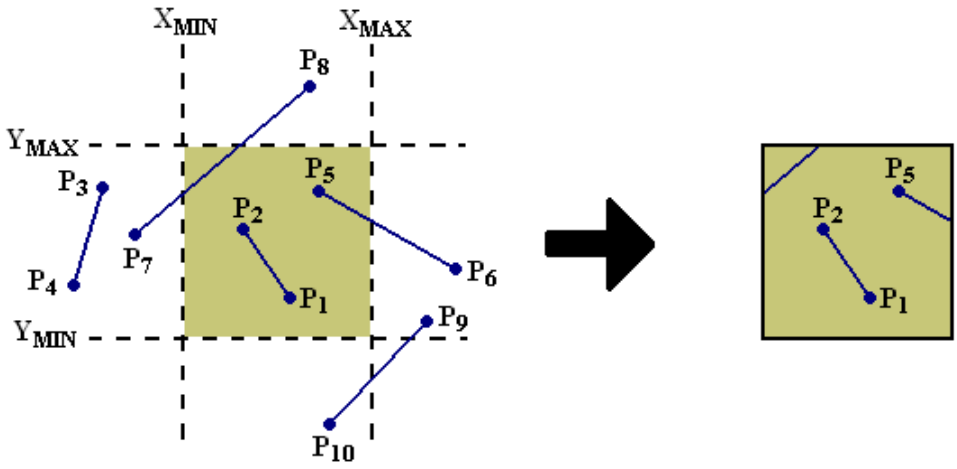


Figure 2 - 2-D Line Clipping

3.2.1 Cohen-Sutherland Line Clipping

Now we will consider a specific example of a line clipping algorithm: the *Cohen-Sutherland* algorithm. The general approach to clipping line primitives using this technique can be summarised as:

- Decide if we can accept the entire primitive without further processing (*trivial accept*)
- Decide if we can reject the entire primitive without further processing (*trivial reject*)
- *Divide-and-conquer*: for each clipping plane,
    - Compute intersection of the line with the clipping plane
    - Divide the line into two at the intersection point

- Perform a trivial reject on one part of the line
- Accept the other part of the line to be clipped by the other clipping plane(s)

How can we quickly and efficiently decide if we can perform a trivial accept or trivial reject? To help make this decision the Cohen-Sutherland algorithm divides the space around the clipping rectangle into 9 regions, based on their positions relative to each of the four clipping boundaries. Each of these regions is assigned a 4-bit binary *region code*. This is illustrated in Figure 3. The first (i.e. rightmost) bit in the code represents the position of the region relative to the left clipping boundary: we assign this bit a value of 1 if the region lies outside this boundary and a 0 if it is inside. The value of the second bit is assigned in the same way for the right clipping boundary. The third and fourth bits represent the bottom and top clipping boundaries respectively. These region codes can be very quickly computed for any given point: we simply subtract the *x* or *y* coordinate from the coordinate of each clipping boundary, and take the *sign bit* of the result. (When numbers are stored using the binary system inside a computer, one bit always represents the sign of the number.) Therefore each bit of the region code can be computed using a single subtraction.

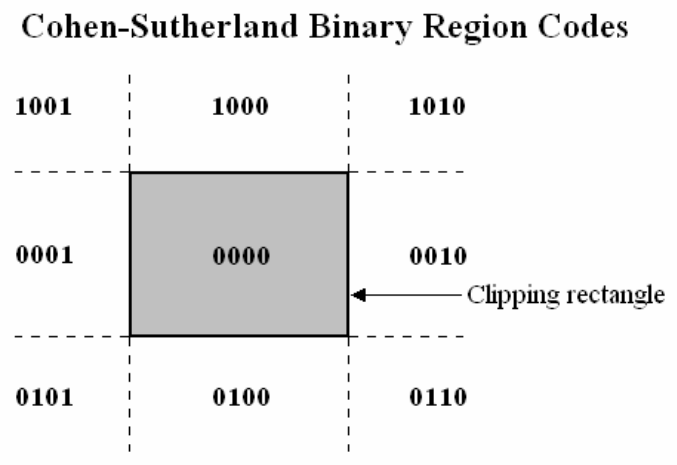


Figure 3 - Binary Region Codes

The region codes of the two end-points of a line primitive (C_{start} and C_{end}) are used to determine whether we can perform trivial accept or trivial reject operations, as follows:

- If C_{start} OR $C_{end} = 0000$, we perform *trivial accept*
- If C_{start} AND $C_{end} \neq 0000$, we perform *trivial reject*

Intuitively we can see that C_{start} OR $C_{end} = 0000$ if and only if both C_{start} and C_{end} are equal to 0000. In other words both end-points lie inside the clipping rectangle, so we can accept the entire primitive without further processing. If C_{start} AND $C_{end} \neq 0000$, this means that for at least one clipping boundary both end-points have a value of 1, and therefore lie outside that boundary. If both end-points lie outside any given boundary then the line can never intersect the clipping rectangle, so the primitive can be rejected without further processing. Note that binary operations such as OR and AND can be performed very efficiently in a computer's CPU.

If neither of these two conditions is met, we need to do some more work. In particular we need to compute intersection points of the line with the clipping boundaries. However, we don't necessarily need to compute intersections with every clipping boundary, only those boundaries

where the end-points lie on different sides. Therefore we can decide which boundaries need to have intersections computed by performing an XOR operation on the binary region codes. Any clipping boundary that has a 1 in the XOR of the two end-point region codes may need to have an intersection calculated. Otherwise no intersection need be computed. Note that again XOR operations are performed very quickly in the computer's CPU.

It is normal to compute line intersections using the *parametric equation* for a straight line. The alternative is to use the *slope-intercept* equation (i.e. $y = mx + c$), but this equation can have problems when dealing with vertical lines (the gradient becomes infinite). The parametric form is:

$$x = x_{start} + u(x_{end} - x_{start}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

$$y = y_{start} + u(y_{end} - y_{start}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

These equation are illustrated in Figure 4: the parameter u indicates how far the point is along the line from (x_{start},y_{start}) to (x_{end},y_{end}). Given that we know the coordinates of the two end-points and the coordinate of the clipping boundary, we can compute the intersection point with the left or right clipping boundaries as follows:

- Substitute x_{clip} for x in Eq. (1) and solve for u
- If u < 0 or u > 1 then the line does not intersect the clipping boundary
- Substitute u into Eq. (2) and solve for y

For the bottom or top clipping boundaries we switch Eqs. (1) and (2) in the above calculation. See the Exercises section at the end of this chapter for examples of this calculation.

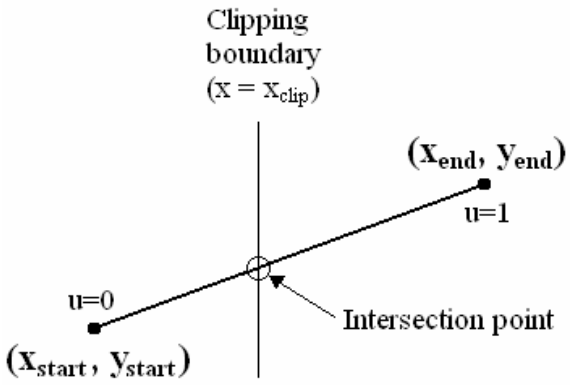


Figure 4 - Computing Line Intersections Using the Parametric Line Equation

Now we know how to compute intersections, and we know which boundaries we need to compute intersections for. The final stage is to *divide-and-conquer*: we consider each clipping boundary in turn, if necessary we compute an intersection point of the line with the boundary, then we divide the line into two parts. One part can be trivially rejected, whilst the other part is accepted and processed against the other clipping boundaries.

For example, consider Figure 5. We start with the line P_1P_2 and perform the following steps:

- Compute region codes: $C_1 = 0100$ and $C_2 = 1001$
- Test for trivial accept: C_1 OR $C_2 = 1101$, so we cannot do trivial accept
- Test for trivial reject: C_1 AND $C_2 = 0000$, so we cannot do trivial reject
- Decide which boundaries need intersections computed: C_1 XOR $C_2 = 1101$, so left, bottom and top boundaries may need intersections computed
- Clip against left clipping boundary:
  - Compute intersection: P_2'
  - Reject P_2P_2'
  - Accept P_2'P_1
- Clip against right clipping boundary:
  - We do not need to compute an intersection, so we can do either trivial accept or trivial reject: in this case we accept the entire line P_2'P_1
- Clip against bottom clipping boundary:
  - Compute intersection: P_1'
  - Reject P_1P_1'
  - Accept P_2'P_1'
- Clip against top clipping boundary:
  - Compute intersection: P_2''
  - Reject P_2''P_2'
  - Accept P_2''P_1'
- So the final clipped line is P_2''P_1'.

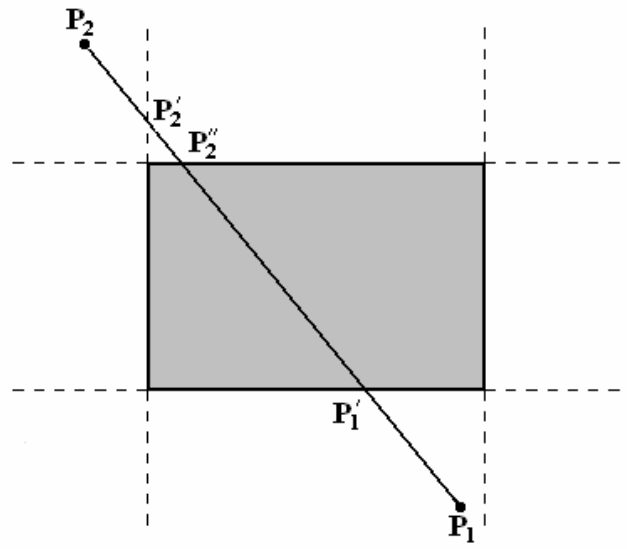


Figure 5 - The Cohen-Sutherland Line Clipping Algorithm

3.3 2-D Polygon Clipping

Just as with line clipping the first stage in polygon clipping is to detect the trivial accept and trivial reject cases. We could do this by performing the Cohen-Sutherland line test for each edge in the polygon individually – if all are accepted or all rejected then we can accept/reject the entire polygon. However, this would be time-consuming, so the normal approach is to use a *bounding box* that encloses the polygon. For example, Figure 6 shows the detection of trivial accept and trivial reject cases for a polygon. This technique is preferred because calculating a bounding box and testing its coordinate extents against those of the clipping rectangle is fast and easy.

3.3.1 Sutherland-Hodgman Polygon Clipping

If we cannot accept or reject the entire polygon, we must clip it. A common technique for polygon clipping is the Sutherland-Hodgman algorithm. The first stage in this algorithm is to categorise each of the edges in the polygon according to whether it is entirely inside, entirely outside, entering or leaving the clipping rectangle. This should be done separately for each clipping boundary. Referring to Figure 7, we can see that for the top clipping boundary 4 types of edge can be identified:

- OUT-OUT: The edge from vertex 1 to 2 in Figure 7 is entirely outside the clipping rectangle.
- OUT-IN: The edge from vertex 2 to vertex 3 in Figure 7 starts outside the clipping rectangle and finishes inside it.
- IN-IN: The edge from vertex 3 to 4 in Figure 7 is entirely inside the clipping rectangle.
- IN-OUT: The edge from vertex 4 to 1 in Figure 7 starts inside the clipping rectangle and finishes outside it.

Next, for each clipping boundary, we process each edge in turn. We categorise it according to the 4 categories listed above, and produce a set of output vertices for each edge according to its category. The input for each edge is the two vertices that define the edge. Figure 8 illustrates the output that is generated for each of the 4 categories of edge. For OUT-OUT edges no output is produced. For OUT-IN edges we compute the intersection of the edge with the clipping boundary and then output the intersection point followed by the second vertex. For IN-IN edges we output the second of the input vertices only. For IN-OUT edges we compute and return only the intersection point. For the example shown in Figure 8 the final output is the sequence of vertices 2', 3, 4, 4', which represents the correct polygon clipped against the top clipping boundary. This process should be repeated for each of the four boundaries.

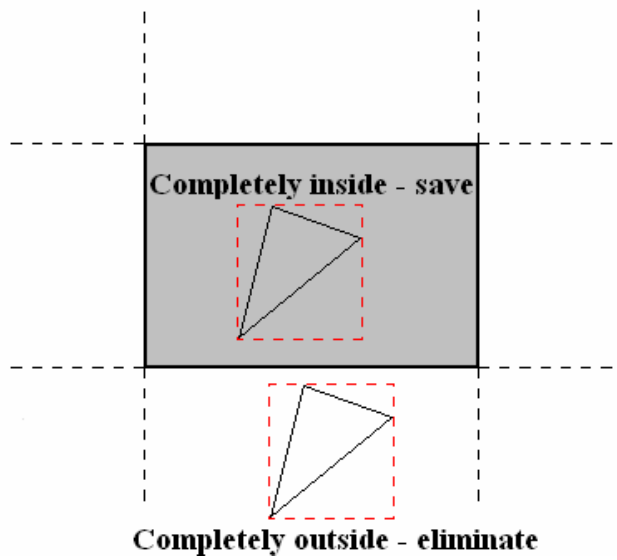


Figure 6 - Determining Trivial Accept and Trivial Reject Cases for Polygon Clipping

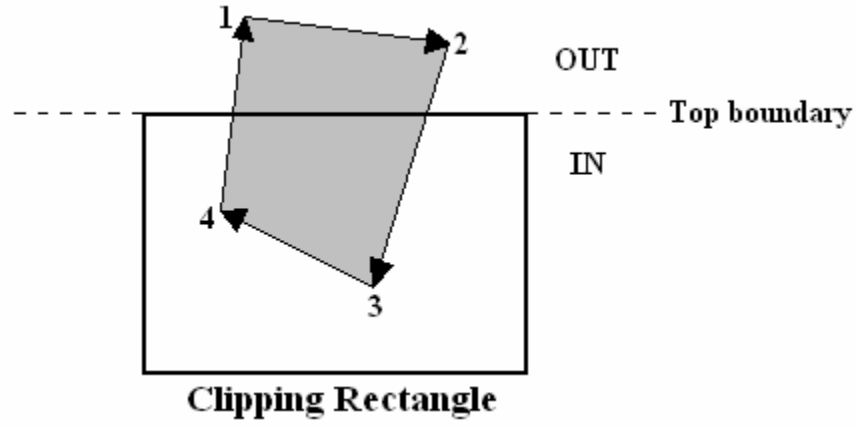
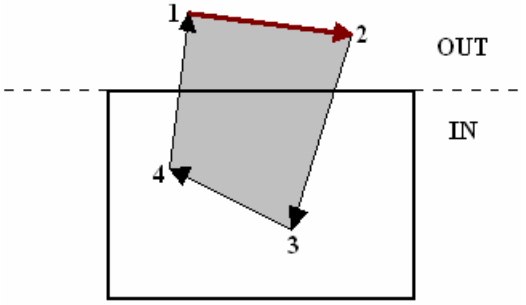
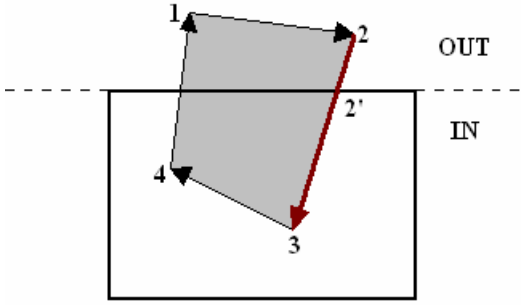


Figure 7 - Categorising Edges for Polygon Clipping



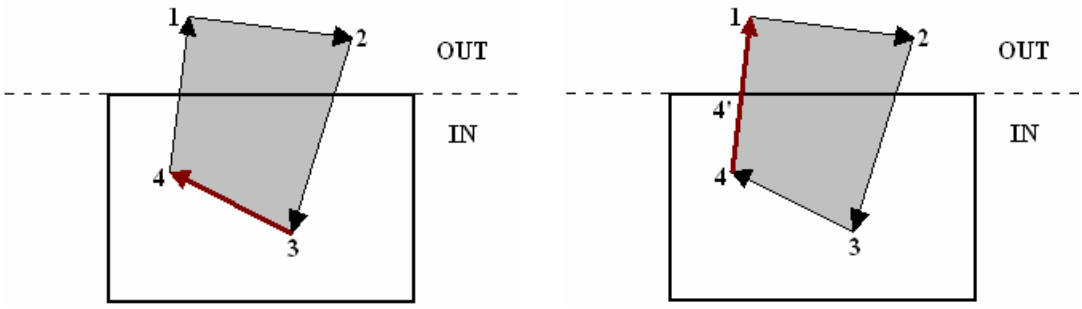
OUT-OUT: Output = none

(a)



OUT-IN: Output = 2', 3

(b)

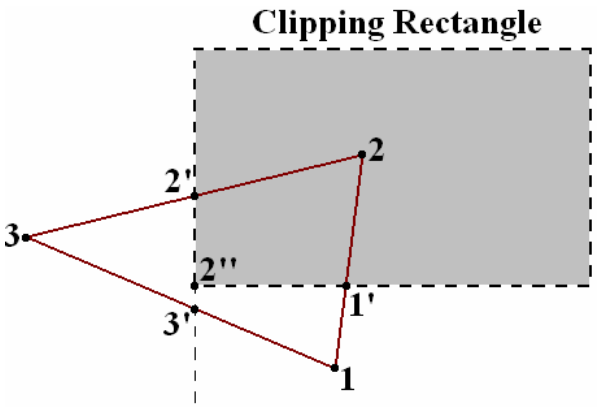


IN-IN: Output = 4
(c)

IN-OUT: Output = 4'
(d)

Figure 8 - Processing Polygon Edges for Clipping

One appealing factor about the Sutherland-Hodgman algorithm is its potential for parallelism. We can think of the algorithm as consisting of four clipping processes, one for each clipping boundary, operating in a pipeline. As soon as two vertices have been produced as the output from the first clipping process, they can be fed in to the next clipping process, without waiting for the full set of output vertices to be produced. Figure 9 shows an example application of Sutherland-Hodgman polygon clipping that illustrates this pipeline. After the first two edges have been processed by the left clipper, we have a current output of [2, 2']: this forms the first edge fed into the right clipper. When another vertex is available from the left clipper (3'), we form a new edge ([2', 3']) and feed it into the right clipper. As soon as two edges are available from the output of the right clipper, we form a new edge [2', 3'] and feed it into the bottom clipper, and so on. For polygons with many edges this can produce a significant improvement in efficiency, because all four clippers can be operating at the same time.



(a)

Left Clipper →	Right Clipper →	Bottom Clipper →	Top Clipper
[1,2] - IN-IN - [2]			
[2,3] - IN-OUT - [2']	[2,2'] - IN-IN - [2']		
[3,1] - OUT-IN - [3',1]	[2',3'] - IN-IN - [3']	[2',3'] - IN-OUT - [2'']	

	[3’,1] - IN-IN - [1]	[3’,1] - OUT-OUT - []	
	[1,2] - IN-IN - [2]	[1,2] - OUT-IN - [1’,2]	[2’’,1’] - IN-IN - [1’]
		[2,2’] - IN-IN - [2’]	[1’,2] - IN-IN - [2]
			[2,2’] - IN-IN - [2’]
			[2’,2’’] - IN-IN - [2’’]

(b)

Figure 9 - Sutherland-Hodgman Polygon Clipping

One potential limitation with the Sutherland-Hodgman algorithm is that it always produces a single polygon as its output. This can cause problems if we are trying to clip a *concave* polygon. Figure 10 shows what can happen if concave polygons have two areas of overlap with the clipping rectangle: the output should consist of two separate polygons but the Sutherland-Hodgman algorithm joins them together to form a single linked polygon, which is not the desired result. Such problems do not occur with convex polygons. Although this is a weakness of the technique, we should remember that most graphics packages do not permit concave polygons for reasons related to rendering speed. Therefore, the Sutherland-Hodgman algorithm (or variations of it) is still commonly used by graphics packages.

Figure 10 - Clipping Concave Polygons Using Sutherland-Hodgman Polygon Clipping

4. 3-D Clipping

All of the techniques described in this chapter are easily generalisable to 3-D. Recall that in 3-D graphics we have an extra two clipping planes to process: the near and far clipping planes. For clipping techniques that use the 4-bit binary region code we simply add an extra 2 bits for the extra clipping planes. Figure 11 illustrates the 27 regions that are produced using this extension: 9 in front of the near clipping plane, 9 between the near and far clipping planes and 9 beyond the far clipping plane.

4.1 3-D Point Clipping

Now 3-D point clipping is just a matter of testing the region code of each point to see if it is equal to 000000. If it is not equal to 000000 then the point should be rejected, otherwise it will be accepted.

4.2 3-D Line Clipping

The Cohen-Sutherland line clipping algorithm can be easily extended to 3-D. The trivial accept and trivial reject cases can be detected in a similar way to the 2-D case:

- If C_{start} OR C_{end} = 000000, we perform *trivial accept*
- If C_{start} AND $C_{end} \neq$ 000000, we perform *trivial reject*

Then the algorithm proceeds with the divide-and-conquer technique, dividing the line at its intersection points with each of six clipping boundaries (instead of four). As in the 2-D case, we only compute intersections for planes where the appropriate bit in the XOR of the regions codes is equal to 1.

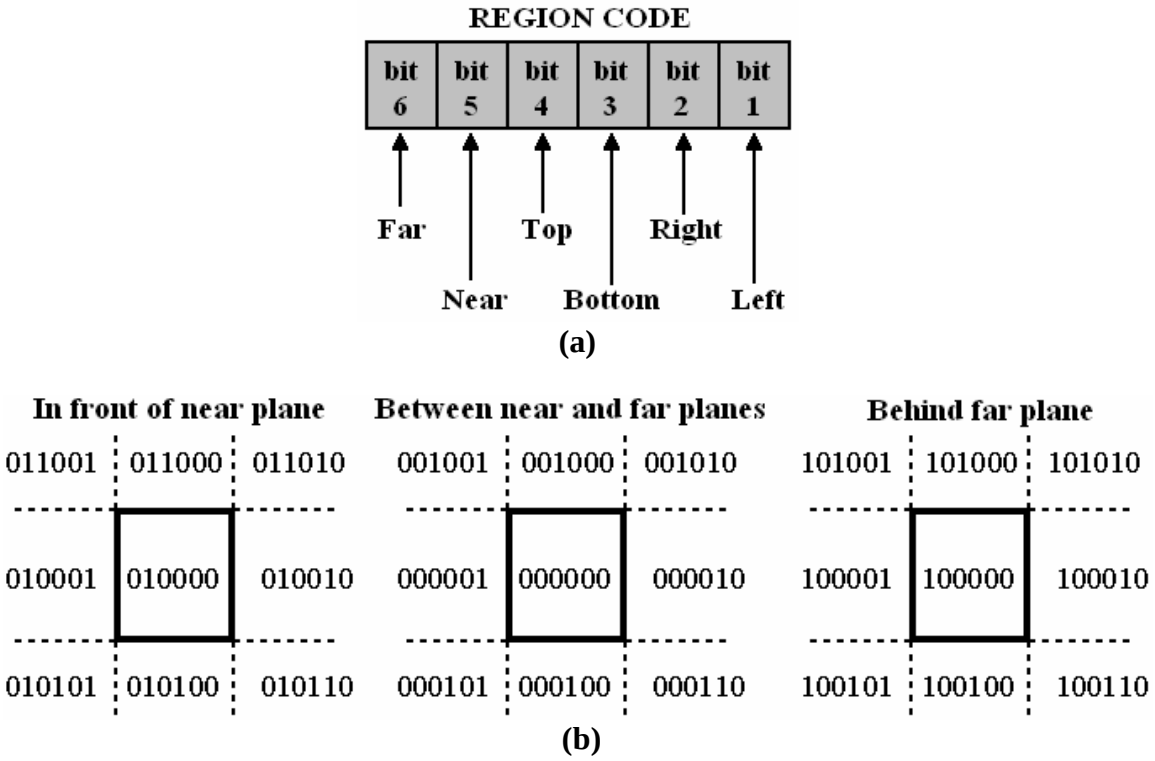


Figure 11 - Binary Region Codes for 3-D Clipping

4.3 3-D Polyhedron Clipping

Most 3-D graphics objects are *polyhedra* – that is, they are formed from a mesh of planar polygons. To clip such 3-D polyhedra, we can apply the Sutherland-Hodgman algorithm to each polygon in the mesh in turn. The combination of all clipped polygons will form a correct clipped polyhedron. For example, Figure 12 illustrates a polygonal mesh being clipped against a cuboid view volume. The parts of the polyhedron shown with solid lines are clipped, whereas the portion shown using dashed lines is saved.

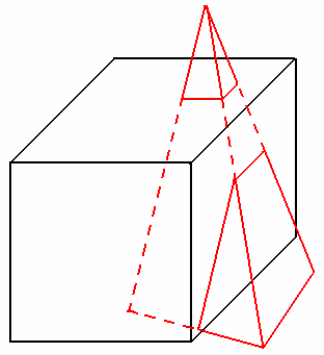


Figure 12 - 3-D Polyhedron Clipping

Summary

The following points summarise the key concepts of this chapter:

- *Clipping* is the process of removing portions of a picture that are outside of a specified region of space
- The most common reason for clipping is to remove (parts of) primitives that lie outside of the view volume in the graphics pipeline.
- Clipping can take place:
    - *Before scan conversion* – known as *analytical clipping*, the most common approach.
    - *During scan conversion* – known as *scissoring*, generally less efficient than analytical clipping, but can be more efficient if the primitive lies mostly inside the clipping region.
    - *After scan conversion* – the primitive is drawn on a temporary canvas, and then the clipping region is copied to the frame buffer. Common for drawing text primitives
- Many clipping algorithms use *binary region codes* to efficiently determine the positions of points
- *Point clipping* can be performed by testing if the binary region code is equal to 0000 (in 2-D, or 000000 in 3-D).
- The *Cohen-Sutherland* algorithm can be used to clip lines in either 2-D or 3-D:
    - If C_{start} OR $C_{end} = 0000$, we perform *trivial accept*
    - If C_{start} AND $C_{end} \neq 0000$, we perform *trivial reject*
    - *Divide-and-conquer*: for each clipping plane:
        - Compute intersection of the line with the clipping plane
        - Divide the line into two at the intersection point
        - Perform a trivial reject on one part of the line
        - Accept the other part of the line to be clipped by the other clipping plane(s)
- The *Sutherland-Hodgman* algorithm can be used to clip planar polygons:
    - Perform a test for trivial reject and trivial accept using a bounding box
    - For each clipping boundary:
        - Categorise each edge as either IN-IN, IN-OUT, OUT-IN or OUT-OUT
        - Produce output vertices for each edge according to its category
- 3-D *polyhedra* can be clipped by applying the *Sutherland-Hodgman* algorithm to each of its constituent polygons in turn.

Exercises

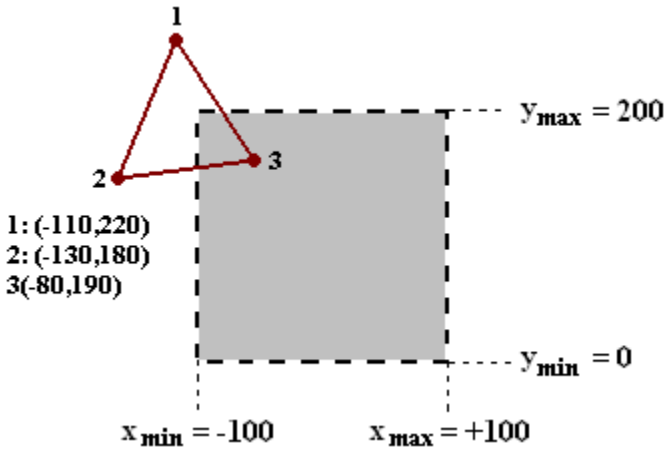
1) Using a 2-D clipping rectangle defined by the coordinates (x_{min},y_{min}) = (-100,0) and (x_{max},y_{max}) = (100,200), calculate 4-bit binary region codes for the following line end-points. For each line, state whether the Cohen-Sutherland line clipping algorithm would be able to perform a *trivial accept*, *trivial reject*, or if further processing would be required. If further processing is required, state which clipping boundaries require intersection points to be computed.

   a. (-110,250) and (-120,-50)
   b. (0,90) and (150,210)
   c. (50,-50) and (150,-50)
   d. (-90,10) and (90,190)
   e. (0,-100) and (200,10)

2) Using the same 2-D clipping rectangle, (x_{min},y_{min}) = (-100,0) and (x_{max},y_{max}) = (100,200), state whether each of the lines specified below intersect with the boundaries specified. If they do intersect, compute the intersection point.

   a. The intersection of the line defined by the end-points (0,90) and (150,210) with the *right* and *left* clipping boundaries.
   b. The intersection of the line defined by the end-points (0,-100) and (200,10) with the *right* and *bottom* clipping boundaries.

3) Again using the same 2-D clipping rectangle, (x_{min},y_{min}) = (-100,0) and (x_{max},y_{max}) = (100,200), use the Sutherland-Hodgman polygon clipping algorithm to clip the polygon shown in the figure below. Show what would be the output for each edge as it passes through each clipping process. Process the edges against the clipping boundaries in the order *left*, *right*, *bottom*, *top*.



Note: Remember that when these algorithms are implemented by most graphics packages they will operate on normalised coordinates (i.e. in the range 0 to 1). However, the details of the algorithm are unchanged.

Exercise Solutions

1) Binary region codes can be calculated by determining the positions of the points relative to the clipping boundaries according to Figure 3. In the following answers we assume that the origin of the coordinate system is at the bottom-left. To determine trivial accept we compute the logical OR of the two region codes; to determine trivial reject we take the logical AND; and to determine which clipping planes require an intersection calculation we take the logical XOR.

   a. (-110,250) has the region code 1001
   (-120,-50) has the region code 0101
   1001 OR 0101 = 1101, so we cannot perform trivial accept
   1001 AND 0101 = 0001, so we <u>can</u> perform trivial reject

   b. (0,90) has the region code 0000
   (150,210) has the region code 1010
   0000 OR 1010 = 1010, so we cannot perform trivial accept
   0000 AND 1010 = 0000, so we cannot perform trivial reject
   0000 XOR 1010 = 1010, so we need to compute intersections for the right and top boundaries.

   c. (50,-50) has the region code 0100
   (150,-50) has the region code 0110
   0100 OR 0110 = 0110, so we cannot perform trivial accept
   0100 AND 0110 = 0100, so we <u>can</u> perform trivial reject

   d. (-90,10) has the region code 0000
   (90,190) has the region code 0000
   0000 OR 0000 = 0000, we <u>can</u> perform trivial accept

   e. (0,-100) has the region code 0100
   (200,10) has the region code 0010
   0100 OR 0010 = 0110, so we cannot perform trivial accept
   0100 AND 0010 = 0000, so we cannot perform trivial reject
   0100 XOR 0010 = 0110, so we need to compute intersections for the right and bottom boundaries.

2) Here we use the parametric equation of the straight line, as given in Eqs. (1) and (2). First we substitute the clipping boundary coordinate into either Eq. (1) or (2) (depending on whether it is an x or y coordinate) and compute a value for u. If $0 \leq u \leq 1$ then an intercept exists, otherwise it doesn't. If it exists, we substitute the value of u back into the other equation and find a value for the other coordinate.

   a. The coordinate of the left clipping boundary is $x_{clip} = -100$
   Substituting $x_{clip} = -100$ into Eq. (1), we get $-100 = 0 + u(150 - 0)$
   Therefore u = -0.67, so there is no intercept with this line

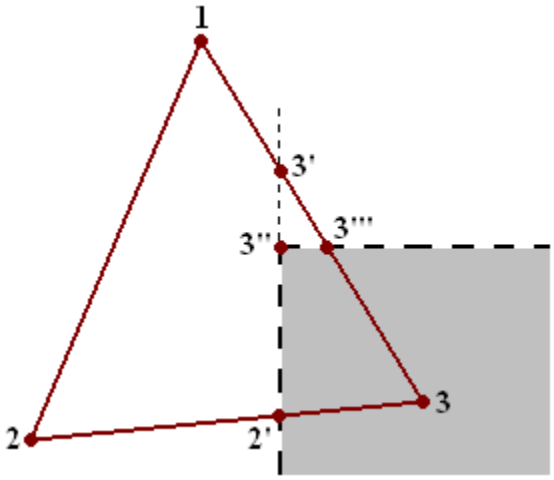
   The coordinate of the right clipping boundary is $x_{clip} = 100$
   Substituting $x_{clip} = 100$ into Eq. (1), we get $100 = 0 + u(150 - 0)$
   Therefore $u = 0.67$, so there is an intercept with this line
   The x-coordinate of the intercept point is equal to 100 (x_{clip})
   The y-coordinate of the intercept point is calculated by substituting $u = 0.67$ into Eq. (2):
   $y = 90 + 0.67(210 - 90) = 170$
   So the intercept point is at (100,170)

   b. The coordinate of the right clipping boundary is $x_{clip} = 100$
   Substituting $x_{clip} = 100$ into Eq. (1), we get $100 = 0 + u(200 - 0)$
   Therefore $u = 0.5$, so there is an intercept with this line
   The x-coordinate of the intercept point is equal to 100 (x_{clip})
   The y-coordinate of the intercept point is calculated by substituting $u = 0.5$ into Eq. (2):
   $y = -100 + 0.5(10 - (-100)) = -45$
   So the intercept point is at (100,-45)

   The coordinate of the bottom clipping boundary is $y_{clip} = 0$
   Substituting $y_{clip} = 0$ into Eq. (2), we get $0 = -100 + u(10 - (-100))$
   Therefore $u = 0.909$, so there is an intercept with this line
   The y-coordinate of the intercept point is equal to 0 (y_{clip})
   The x-coordinate of the intercept point is calculated by substituting $u = 0.909$ into Eq. (1):
   $x = 0 + 0.909(200 - 0) = 181.81$
   So the intercept point is at (181.81,0)

3) The actions of each clipper in the pipeline are shown below. Refer to the figure to see where the extra intersection points are located.

Left Clipper →	Right Clipper →	Bottom Clipper →	Top Clipper
[1,2] – OUT-OUT - []			
[2,3] – OUT-IN - [2’,3]	[2’,3] - IN-IN - [3]		
[3,1] – IN-OUT - [3’]	[3,3’] - IN-IN - [3’]	[3,3’] - IN-IN - [3’]	
	[3’,2’] - IN-IN - [2’]	[3’,2’] – IN-IN - [2’]	[3’,2’] - OUT-IN - [3’’,2’]
		[2’,3] - IN-IN - [3]	[2’,3] - IN-IN - [3]
			[3,3’] - IN-OUT - [3’’’]



Source: Computer Graphics with OpenGL,
Donald Hearn and M. Pauline Baker,
Prentice-Hall, 2004 (Third Edition)