Clipping Algorithms

Real world objects can be represented relative to a reference world coordinate system. It is difficult to view all the objects on computer screen at the same time in one screen shot because they usually occupy many places in the world coordinate system. As in human perception and cameras, one screen shot contains the images of some objects, parts of which are probably clipped. On determining which objects or parts of objects to display on one screen shot, one needs to run a clipping algorithm on every object in the world to determine whether it’d appear (completely or partially) on the screen or be clipped out. Clipping algorithms usually have parameters describing the ‘clipping window’ in 2D worlds or ‘clipping volume’ in 3D worlds plus the representation of the object being tested for clipping. Here, we are interested in 2D clipping algorithms only. Extension to 3D clipping is straightforward in many cases. The clipping window is usually a rectangular window although it could be of any shape (circular, polygonal or others). Rectangular windows are easy to be mapped to computer screen using a transformation called ‘window to viewport mapping’ that is simply composed of translation, scaling and reflection transformations. The parameters of the rectangular clipping window are (x_left, y_bottom) and (x_right, y_top); the lower left and upper right corner coordinates expressed in real-world units (e.g. meter, cm, inch, .. etc.). Screen viewport parameters are given in pixels. The following figure shows a part of a simple viewing pipeline in 2D graphics.

Please note that the screen’s coordinate system considers the upper-left corner to be its origin and the y-axis points downwards. Clipping window should be described relative to world coordinate system (not screen’s coordinate system); so its y-axis points upwards. This means that y_bottom of the clipping window is less than y_top. In these notes, we’ll adopt this view in algorithms, but in C++ implementation, since we haven’t yet studied the coordinate system transformation (window to view port mapping), we’ll assume that the clipping window is expressed relative to screen coordinates for simplicity.

We’ll focus on three types of clipping algorithms:

- **Point clipping**
- **Line clipping**
- **Polygon clipping**
Point Clipping

A point \((x, y)\) is considered for drawing only if \(x\) lies between \(x_{left}\) and \(x_{right}\) and \(y\) lies between \(y_{bottom}\) and \(y_{top}\). The following algorithm reflects this fact:

Algorithm PointClipping\((x,y,x_{left}, y_{bottom} ,x_{right}, y_{top})\)
If \((x\geq x_{left})\) and \((x\leq x_{right})\) and \((y\geq y_{bottom})\) and \((y\leq y_{top})\) then
DrawPoint\((x, y)\)
End if
End Algorithm

The algorithm will not draw (clip out) the point if the condition is false. Here is the Visual C++ implementation of the above algorithm

```cpp
void PointClipping(int x, int y, int xleft, int ytop, int xright, int ybottom,COLORREF color)
{
    if(x=xleft && x<= xright && y>ytop && y<=ybottom)
    SetPixel(hdc,x,y,color);
}
```

Note that the difference between the implementation and the algorithm is only in the naming of the variables: \(y_{top}\) and \(y_{bottom}\) which are swapped in the implementation because, as said before, we are supposed to run the implementation directly without the ‘window to viewport mapping’.

Line Clipping

Clipping of lines against a window needs to determine if the line has an intersection with the edges of the window and performs the intersection accordingly. The intersection between the line and an edge of the window is costly since it incorporates floating point operations. Several algorithms are introduced to clip lines against rectangular and non-rectangular windows. Efficient algorithms try to minimize the number of intersections as much as possible. The so called ‘Cohen-Sutherland algorithm’ is a famous line clipping algorithm and we’ll study it in the next section.

Cohen-Sutherland Algorithm

Given a rectangular clipping window, the algorithm divides the space into 9 regions w.r.t. the window: left, left-top, top, right-top, right, right-bottom, bottom, left-bottom and inside as in Figure.
These regions can be described as subsets of the set of basic outside sides \{left, top, right, bottom\}. So, the 9 regions could be represented as the subsets: \{left\}, \{left, top\}, \{top\}, \{right, top\}, \{right\}, \{right, bottom\}, \{bottom\}, \{left, bottom\} and \{\}. The out code of some point is one of these 9 subsets.

The algorithm first computes the out codes of the end points of the line: \((x_s, y_s)\) and \((x_e, y_e)\). It then does the following tests:

1. If both out codes are empty sets (meaning that both end points are inside) the line is completely inside the window and has to be drawn with no clipping. The algorithm then terminates.
2. If both out codes have a non-empty intersection (meaning that they have a common outside) the line is completely outside the window and completely clipped out (not drawn). The algorithm then terminates.
3. If the above conditions are not true (meaning that at least one out code is non-empty and no intersection between the two out codes), the algorithm picks one side from the non-empty out code sets and intersects the line with this side, replaces the corresponding end point with the point of intersection, and re-computes the out code of this end point. The algorithm then repeats the tests until termination.

In the last test, the algorithm defines the order of picking one side from the non-empty sets. One possibility is:

- Start with the out code of \((x_s, y_s)\) if not empty; otherwise start with that of \((x_e, y_e)\)
- Pick the edges in the following (priority) order: left, top, right, bottom.

Of course we can change the order rules without any effect on the final results. Although the algorithm can trivially accept or reject lines without doing any intersection, there is also a possibility of doing unnecessary intersections before trivially rejecting or accepting a line.

**Implementation of the algorithm**

To implement the algorithm, first we define a data structure for the out code sets. One possible way in visual C++ is to define union containing two fields: the first field is a 4-bit field named ‘All’ and the second is a structure containing four one-bit fields which are left, top, right and bottom. According to the definition of unions in C++, both fields are stored in the same memory location so they are aliases to each other; updating one field leads to updating the other. The definition of the union is shown below:

```c++
Union OutCode
{
    unsigned All:4;
    struct{
        unsigned left:1,top:1,right:1,bottom:1; 
    };
};
```

Second, we write the utility function that gets the side code of a given point \((x, y)\) with respect to a window \((x_{left}, y_{top}, x_{right}, y_{bottom})\) as shown below:

```c++
OutCode GetOutCode(double x, double y, int xleft, int ytop, int xright, int ybottom)
{
    OutCode out;
    out.All=0;
    if(x<xleft)out.left=1;
    if(x>xright)out.right=1;
    if(y<ytop)out.top=1;
    if(y>ybottom)out.bottom=1;
    return out;
}
```

Third we define the intersection utilities. The intersection point between the line and some edge is the solution of the line and edge equations. The equations of the edges are as follows:
The equation of the line is:

\[ \frac{y - y_s}{x - x_s} = \frac{y_e - y_s}{x_e - x_s} \]

Clearly, for the left and right edges the intersection point has its x component equals the x component of the edge \( x_{edge} \) (which is either \( x_{left} \) or \( x_{right} \)). The y component is obtained by the substitution \( x = x_{edge} \) in the line equation; i.e.

\[ y_{in} = y_s + (x_{edge} - x_s)(y_e - y_s)/(x_e - x_s) \]

Similarly, for the top and bottom edges:

\[ x_{in} = x_s + (y_{edge} - y_s)(x_e - x_s)/(y_e - y_s) \]

Below is the implementation of the two intersection utilities:

```c
void VIntersect(double xs, double ys, double xe, double ye, int x, double *xi, double *yi) {
    *xi=x;
    *yi=ys+(x-xs)*(ye-ys)/(xe-xs);
}
void HIntersect(double xs, double ys, double xe, double ye, int y, double *xi, double *yi) {
    *yi=y;
    *xi=xs+(y-ys)*(xe-xs)/(ye-ys);
}
```

Fourth, the algorithm implementation is as shown below:

```c
void CohenSuth(HDC hdc, int xs, int ys, int xe, int ye, int xleft, int ytop, int xright, int ybottom) {
    double x1=xs, y1=ys, x2=xe, y2=ye;
    OutCode out1=GetOutCode(x1,y1,xleft,ytop,xright,ybottom);
    OutCode out2=GetOutCode(x2,y2,xleft,ytop,xright,ybottom);
    while( (out1.All || out2.All) && !(out1.All & out2.All) )
    {
        double x1,y1;
        if(out1.All)
        {
            if(out1.left)VIntersect(x1,y1,x2,y2,xleft,&xi,&yi);
            else if(out1.top)HIntersect(x1,y1,x2,y2,ytop,&xi,&yi);
            else if(out1.right)VIntersect(x1,y1,x2,y2,xright,&xi,&yi);
            else HIntersect(x1,y1,x2,y2,ybottom,&xi,&yi);
            x1=xi;
            y1=yi;
            out1=GetOutCode(x1,y1,xleft,ytop,xright,ybottom);
        }
        else
        {
            if(out2.left)VIntersect(x1,y1,x2,y2,xleft,&xi,&yi);
            else if(out2.top)HIntersect(x1,y1,x2,y2,ytop,&xi,&yi);
            else if(out2.right)VIntersect(x1,y1,x2,y2,xright,&xi,&yi);
            else HIntersect(x1,y1,x2,y2,ybottom,&xi,&yi);
            x2=xi;
            y2=yi;
            out2=GetOutCode(x2,y2,xleft,ytop,xright,ybottom);
        }
    }
    if(!out1.All && !out2.All) {
        MoveToEx(hdc,Round(x1),Round(y1),NULL);
        LineTo(hdc,Round(x2),Round(y2));
    }
}
```
Polygon clipping algorithms

Here, we’ll study the so-called Sutherland-Hodgman algorithm for polygon clipping. The algorithm clips the polygon by the window side by side. With rectangular windows, the algorithm has four main iterations: clipping with the left side, clipping with the top side, clipping with the right side and clipping with the bottom side. The four iterations are similar; so, it is sufficient to describe the one ‘generic’ iteration. For this purpose, we need two utility functions:

The first utility function is ‘\( \text{In}(v, \text{side}) \)’ that takes two parameters: polygon vertex \( (v) \) and window edge \( (\text{side}) \). The function tests if the vertex is ‘in’ or ‘out’ relative to the side. The window edge line divides the space into two regions; the first is in the same side of the window and the second is in the other side. A vertex is considered ‘in’ if it is in the same side of the window and considered ‘out’ otherwise. The following figure shows the ‘in’ and ‘out’ regions relative to the left window edge.

![Diagram showing 'in' and 'out' regions for left window edge]

The second utility is ‘\( \text{Intersect}(v1, v2, \text{side}) \)’ that takes three parameters: Two vertexes of a polygon edge \( (v1 \text{ and } v2) \) and the window edge \( (\text{side}) \). The function computes the intersection between the polygon edge and the window edge and returns the intersection point.

The basic iteration of the Sutherland-Hodgman algorithm is given below:

```plaintext
Algorithm ClipWithEdge(polygon, side)
Define an empty list; name it ‘Clipped’
V1=polygon[n-1]
For i=0 to n-1
V2=polygon[i]
If not In(V1, side) and In(V2, side) then
V=Intersect(V1, V2, side)
Append V1 to Clipped
Append V2 to Clipped
Else if In(V1, side) and In(V2, side) then
Append V2 to Clipped
Else if In(V1, side) then
V=Intersect(V1, V2, side)
Append v1 to Clipped
End if
V1=V2
End For
Return clipped
End Algorithm
```
Implementation of the Sutherland-Hodgman algorithm

To implement the algorithm, first define the data structures and type definitions used by the algorithm implementation like the following:

```cpp
#include <vector>
using namespace std;
struct Vertex {
    double x, y;
    Vertex(int x1=0, int y1=0) {
        x = x1;
        y = y1;
    }
};
typedef vector<Vertex> VertexList;
typedef bool (*IsInFunc)(Vertex& v, int edge);
typedef Vertex (*IntersectFunc)(Vertex& v1, Vertex& v2, int edge);
```

As you see, we used a built-in dynamic array data structure called 'vector' in the standard template library (STL) of C++ since the polygon size changes dynamically with clipping. Template data types are generic in the sense that they allow to use variable element types. For example, we can use the vector template to define a vector of integers as `vector<int>`, of float as `vector<float>` and etc. The definition (`typedef vector<Vertex> VertexList;`) declares a new data type called 'VertexList' that is a equivalent to a vector of 'Vertex' type. The two definitions (`typedef bool (*IsInFunc)(Vertex& v, int edge);`) and (`typedef Vertex (*IntersectFunc)(Vertex& v1, Vertex& v2, int edge);`) declare new data types called 'IsInFunc' and 'IntersectFunc' that are pointer to functions. The first is supposed to point to a Boolean function taking a vertex parameter and an edge parameter (and returns true if the vertex is in relative to the edge). The second is supposed to point to a function that computes the intersection between the line connecting vertex v1 and vertex v2 and the edge.

Second, given the above data structures, we write the implementation Sutherland-Hodgman generic iteration as below:

```cpp
VertexList ClipWithEdge(VertexList p, int edge, IsInFunc In, IntersectFunc Intersect) {
    VertexList OutList;
    Vertex v1 = p[p.size()-1];
    bool v1_in = In(v1, edge);
    for(int i=0; i<p.size(); i++) {
        Vertex v2 = p[i];
        bool v2_in = In(v2, edge);
        if(!v1_in && v2_in) {
            OutList.push_back(Intersect(v1, v2, edge));
            OutList.push_back(v2);
        } else if(v1_in && v2_in) OutList.push_back(v2);
        else if(v1_in) OutList.push_back(Intersect(v1, v2, edge));
        v1 = v2;
        v1_in = v2_in;
    }
    return OutList;
}
```

Note that the function has the parameters: ‘p’ which is the input polygon (of type `VertexList`), ‘edge’ which gives the position of the generic window side, ‘In’ which is a pointer to the function that tests if a vertex is in or out, and ‘Intersect’ which is a pointer to the function that computes the intersection. The
two function pointers (and in fact all the parameters) will be passed by the main algorithm as will be shown.

Third, define the following utilities to test if a vertex is in or out relative to the four window edges. These are the functions that will be passed to the generic iteration of the algorithm

```c
bool InLeft(Vertex& v, int edge) {
    return v.x>=edge;
}
bool InRight(Vertex& v, int edge) {
    return v.x<=edge;
}
bool InTop(Vertex& v, int edge) {
    return v.y>=edge;
}
bool InBottom(Vertex& v, int edge) {
    return v.y<=edge;
}
```

**Fourth** define the following two intersection functions

```c
Vertex VIntersect(Vertex& v1, Vertex& v2, int xedge) {
    Vertex res;
    res.x=xedge;
    res.y=v1.y+(xedge-v1.x)*(v2.y-v1.y)/(v2.x-v1.x);
    return res;
}
Vertex HIntersect(Vertex& v1, Vertex& v2, int yedge) {
    Vertex res;
    res.y=yedge;
    res.x=v1.x+(yedge-v1.y)*(v2.x-v1.x)/(v2.y-v1.y);
    return res;
}
```

And last, define the main function of the algorithm as follows:

```c
void PolygonClip(HDC hdc, POINT *p, int n, int xleft, int ytop, int xright, int ybottom) {
    VertexList vlist;
    for(int i=0; i<n; i++) vlist.push_back(Vertex(p[i].x, p[i].y));
    vlist=ClipWithEdge(vlist, xleft, InLeft, VIntersect);
    vlist=ClipWithEdge(vlist, ytop, InTop, HIntersect);
    vlist=ClipWithEdge(vlist, xright, InRight, VIntersect);
    vlist=ClipWithEdge(vlist, ybottom, InBottom, HIntersect);
    Vertex v1=vlist[vlist.size()-1];
    for(int i=0; i< (int)vlist.size(); i++) {
        Vertex v2=vlist[i];
        MoveToEx(hdc, Round(v1.x), Round(v1.y), NULL);
        LineTo(hdc, Round(v2.x), Round(v2.y));
        v1=v2;
    }
}
```