IMAGE SPACE ALGORITHMS FOR LINE CLIPPING

by

J.D.DAY

Queensland University of Technology
School of Computing Science
GPO Box 2434
Brisbane 4001
Australia
e–mail j.day@qut.edu.au

Abstract

Algorithms for clipping lines against rectangular windows are normally implemented in object space using floating point arithmetic, and the resulting clipped lines are then scan converted to obtain their representation as a list of pixels in image space. Some algorithms avoid the use of floating point arithmetic. They use integer arithmetic to obtain the end points of the clipped line. The line is then scan converted as before. In both cases there is a side effect. This is described and discussed. An alternative approach is to scan convert the lines, then clip against the viewport. The algorithm would then use a description of the line as a list of pixel locations in image space, rather than a pair of end points in object space. The feasability of this method, in terms of side effects and efficiency, is discussed.

Keywords: clipping,line clipping,computer graphics

1. Introduction

One of the most fundamental of computer graphics algorithms is two dimensional line clipping against a rectangular window. The best known of these is the Cohen-Sutherland algorithm [6]. This algorithm assigns a four bit code to each end point of the line. Once encoded it is easy to identify lines which are completely within the window or completely outside it. Another algorithm which uses a similar model is the Kaijian-Edwards-Cooper algorithm [7] which improves the Cohen-Sutherland algorithm by means of more efficient end code regeneration. Further improvements were made in the Duvanenko-Robbins-Gyurcsik algorithm [5]. The Sobkow-Pospisil-Yang algorithm [10] combines the two end codes into an eight bit line code which permits the classification of the line as one of 81 possible cases,each of which is handled differently. The Liang-Barsky algorithm [8] has a different computational model. It uses the parametric form of the equation to a straight line. The intersections with the window edges are calculated as values of the parameter t, and the appropriate value of t is identified from this set of values. This algorithm is similar to the Cyrus-Beck algorithm [1], but is more efficient for rectangular windows. The Nicholl-Lee-Nicholl algorithm [9] uses a third computational model. It compares the slope of the line with the slopes of the lines joining an end point and each corner of the window. This enables the efficient identification of the edge which intersects the line. Day [2],[3] described two efficient algorithms, using different computational models, and compared all the algorithms described above.

All of these algorithms operate in object space. That is, they all use floating point arithmetic for their computations. Dorr's algorithm [4] was designed to avoid this by using only integer operations. The algorithm is based on the Liang-Barsky model, and includes a complex rounding scheme to ensure that the computed coordinates are the same as those of any object space algorithm whose results are rounded to the nearest integer. The same rounding scheme can be utilized by other algorithms to avoid floating point arithmetic. Although the algorithms use integer arithmetic, strictly speaking they operate in object rather than image space. Any coordinate,

such as the end point of the line (in object space) is rounded to the nearest integer. This is not necessarily a pixel location. The aim of these algorithms is to improve efficiency by doing computations using integers rather than floating point numbers.

Whether the computations are done using floating point or integer arithmetic, all algorithms essentially do the calculations in object space, obtain the end points of the clipped line (either as floating point or integer numbers), then scan convert the line into a list of pixel locations (in image space), using Bresenham's algorithm [6], or something similar. The process has an inherent side effect.

2. A Side Effect

When we scan convert a line, we take a description of that line in two dimensional object space in the form of two (x, y) coordinates corresponding to the end points of the line, and convert it to a description in image space in the form of a list of pixel locations. This is done by mapping the end points to pixel locations, then calculating which other pixels should be illuminated in order to best approximate the line with a series of dots. This is essentially done by finding the pixels whose centre points are closest to the line. Figure 1 illustrates this. Bresenham's algorithm [6] is an efficient way of doing the calculations.

If we clip a line against a window edge, we find the intersection point, and later scan convert the visible part of the line in the manner described above. The clipped line in object space is part of the original line, since the intersection calculations were done in object space. However the scan conversion process starts by mapping the end points of a line to image space, and rounding the result to pixel locations. In most cases this causes a slight perturbation to the line. Then at a series of discrete intervals of pixel size, the pixel closest to this perturbed line is determined. The point of intersection is on the line in object space. However the line is sampled in image space at discrete intervals to determine the pixel locations. The point of intersection, when mapped to image space, may not correspond to one of these sampling points. In fact, it generally will not map to a point on the perturbed line. It will map to a point on the line

in image space, but when the end points of the line are rounded to pixel locations, the line will move slightly, and the point of intersection will not. Consequently, the point of intersection could round to a pixel which is not in the pixel list of the original line. Subsequent scan conversion of the clipped line will then give a set of pixels which is not a subset of the set of pixels of the original line. Algorithms which use integer arithmetic are affected by this as well. The integer coordinates are simply the rounded floating point coordinates, and these are mapped to image space in a similar way. Figure 2 illustrates this.

Thus we have a situation where the clipping process itself can affect the image. It will not affect the model in object space, merely the display. This will not always be a problem. It depends on what is being done, and even so, it may be possible to avoid the consequences by a suitable adjustment to any algorithm which produces this as a side effect. Nevertheless it is interesting to look for ways of avoiding this artifact.

3. An Image Space Algorithm

The two dimensional viewing pipeline is:

1. Build the model in object space using whatever primitives and attributes are available.
2. Specify a window in object space, and clip the model to it.
3. Specify a viewport in image space, and map the contents of the window to it. If the model consists of lines, for example, then the end points of the lines are mapped from object to image space.
4. Scan convert the model. For lines, the pixel locations of the end points are known, and the scan conversion algorithm will find all other pixels needed to approximate the line.

Consider an alternative viewing pipeline:

1. Build the model in object space.
2. Specify a window in object space, but do not clip the model to it.

3. Specify a viewport in image space, and map the unclipped model to it. This means that the part of the model which is in the window will be mapped to the interior of the viewport. However there will be part of the model which maps to the exterior of the viewport, and must be clipped. If parts of this unclipped model are mapped to image space which does not correspond to any pixel location, it does not matter. Ultimately, only the interior of the viewport will be displayed.

4. Scan convert the unclipped model.

5. Clip the scan converted model against the viewport.

This will certainly eliminate the side effect described above, but at a cost. There will be more effort required to do the scan conversion, since we are now using the whole model, rather than that portion inside the window. There is also the question of the clipping algorithm itself. This will take a line described by a list of pixel locations and clip it against a rectangular viewport. It will return a list of pixels corresponding to that part of the (scan converted) line which is inside the viewport. Some object space algorithms can be used as a basis for this algorithm. For example the well known Cohen–Sutherland algorithm [6] is easily adapted. This algorithm assigns a four bit endcode to each of the two endpoints of the line, depending on their position with respect to the window (Figure 3). In pseudocode, the algorithm is:

```
repeat
  {Find the end codes for P1 and P2 in the form
  ( bit[top],bit[bottom],bit[right],bit[left]) eg (1001) }
  if totally invisible then
    rejected
    finished
  else if totally visible then
    accepted
    finished
  else
    if P1 inside window then
```

```
      exchange P1 and P2
    end if
    {end codes below refer to P1 }
    if bit[left] = 1 then
      clip against left edge
    else if  bit[right] = 1 then
      clip against right edge
    else if  bit[bottom] = 1 then
      clip against bottom edge
    else if bit[top] = 1 then
      clip against top edge
    end if
  end if
until finished
if accepted then draw the line
```

Clearly, all that is needed is an algorithm for finding the point of intersection between a line (described by a list of pixel locations) and a vertical or horizontal viewport edge. The pixel list is ordered in x and y so a binary search [11] suggests itself. In pseudocode this is:

```
while low < high do
  mid ← (low + high +1) div 2
  if key < list[mid] then
    high ← mid - 1
  else
    low ← mid
  end if
end while
```

The algorithm has complexity $O(log(n))$, and the operations within the loop are a few integer additions, subtractions and comparisons, together with an integer division by 2. All of these are fast operations, and the $log(n)$ complexity ensures that even if n is quite large, there will not be many passes through the loop. For example,

device resolution of approximately 1000×1000 is considered good, but $log(1000)$ is approximately 10. As mentioned previously the line can extend beyond the display space. However, even if there were 10^6 elements in the pixel list, only approximately 20 passes through the loop would be required. The average situation could be expected to be more reasonable than this, and so the approach seems competitive.

A side effect, however, is that the 'point' of intersection may be more than one pixel, due to aliasing (Figure 4). This is not much of a problem since they are adjacent on the pixel list, and a slight adjustment to the searching algorithm will enable all pixels coincident with an edge to be identified.

4. Summary

An undesirable side effect of the traditional two dimensional viewing pipeline has been described. A different viewing pipeline has been described, which requires an algorithm for clipping a scan converted line against a viewport in image space. Such an algorithm has been described, and an analysis indicates that the new process is feasible.

References

[1] Cyrus,M. and Beck,J.: Generalized two- and three-dimensional clipping. *Computers and Graphics* **3**,23–28(1978)

[2] Day,J.D.: An algorithm for clipping lines in object and image space. *Computers and Graphics* **16**,421–426(1992)

[3] Day,J.D. A new two dimensional line clipping algorithm for small windows. *Computer Graphics Forum* **11**,241–245(1992)

[4] Dorr,M. A new approach to parametric line clipping. *Computers and Graphics* **14**,449–464(1990)

[5] Duvanenko,V.J.,Robbins,W.E., and Gyurcsik,R.S.: Improving line segment clipping. *Dr.Dobb's Journal* **15,7**,36–45,98–100(1990)

[6] Foley,J.D.,van Dam,A.,Feiner,S.K. and Hughes,J.F.: *Computer Graphics Principles and Practice*, second edition,Addison-Wesley Reading,Mass.(1990)

[7] Kaijian,S.,Edwards,J.A. and Cooper,D.C.: An efficient line clipping algorithm. *Computers and Graphics* **14**,297–301(1990)

[8] Liang,Y-D. and Barsky,B.A.: A new concept and method for line clipping. *ACM Transactions on Graphics* **3**,1–22(1984)

[9] Nicholl,T.M.,Lee,D.T. and Nicholl,R.A.: An efficient new algorithm for 2-D line clipping:It's development and analysis. *Computer Graphics* **21**,253–262(1987)

[10] Sobkow,M.S.,Pospisil,P. and Yang,Y-H.: A fast two-dimensional line clipping algorithm via line encoding. *Computers and Graphics* **11**,459–467(1987)

[11] Stubbs,D.F. and Webre,N.W.: *Data Structures with Abstract Data Types and Pascal*, Brooks–Cole Publishing Company, Monterey,Calif. (1985)

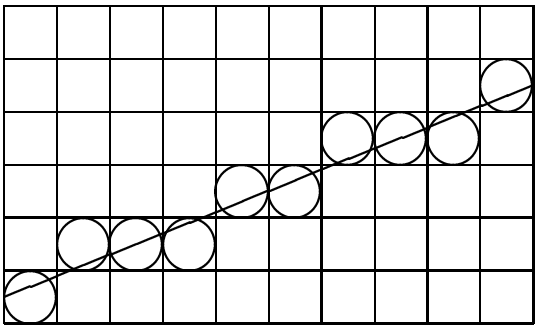

Figure 1: Scan Conversion of a Line

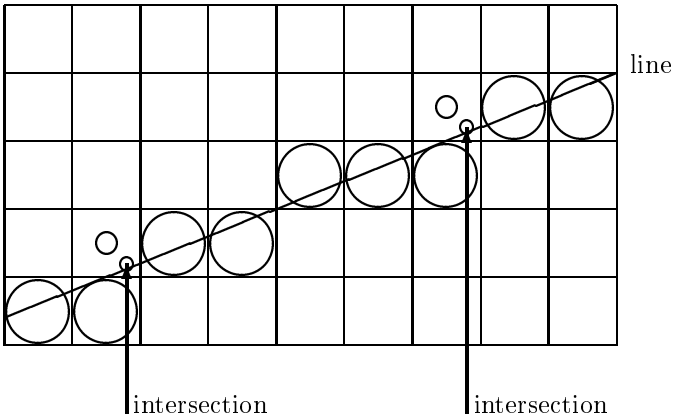

Figure 2: Scan Conversion of a Line and the
Visible Portion of that Line.
(small circle indicates end points of visible line)

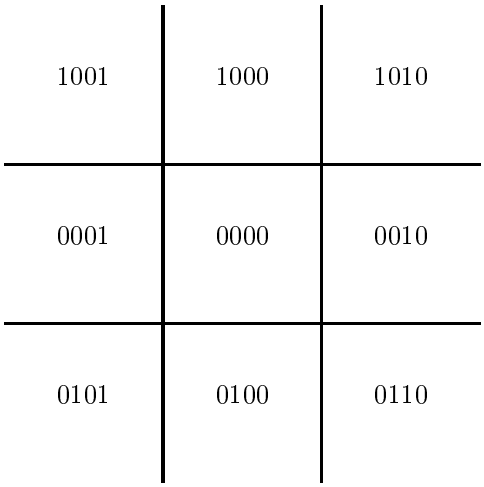

Figure 3: Cohen-Sutherland End Codes

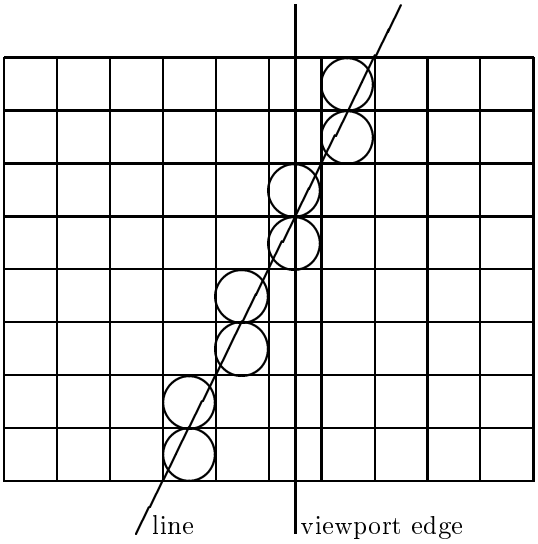

Figure 4: Intersection of a Line and a Viewport Edge