Polygon Clipping

John E. Howland
Department of Computer Science
Trinity University
One Trinity Place
San Antonio, Texas 78212-7200
Voice: (210) 999-7364
Fax: (210) 999-7477
E-mail: jhowland@Trinity.Edu
Web: http://www.cs.trinity.edu/~jhowland/

December 20, 2003

Abstract

An efficient array algorithm, based in part on the re-entrant polygon clipping algorithm of Sutherland and Hodgman, for polygon clipping is given. The algorithm works without changes for both 2d polygon clipping and 3d polygon clipping.

Subject Areas: Computer Graphics, Array Algorithms
Keywords: Polygon Clipping, Array Algorithms.

1 Introduction

Polygon clipping was first described by Sutherland and Hodgman [Suth 1974]. Polygon clipping reduces a polygonal surface extending beyond the boundary of some three-dimensional viewing volume to a surface which does not extend beyond the boundary.

2 Polygon Clipping

To illustrate the array algorithm for polygon clipping we use the J programming language [Hui 2001] and restrict ourselves to the two-dimensional case of clipping a closed polygonal figure to a line. The array algorithm applies to three-dimensional data without changes to the J program.

Suppose we have the square:

```
   [ square =: 5 2 $ 0 0 100 0 100 100 0 100
  0   0
100   0
100 100
  0 100
```

and the line $y = x - 50$ (in homogeneous form) as illustrated in Figure 1.

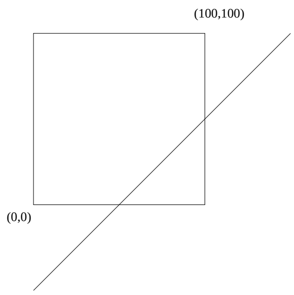

Figure 1: Clip a square to a line

```
   [ line =: _1 1 50
_1 1 50
```

The array algorithm for polygon clipping (expressed in J) is:

```
pclip =: 4 : 0
a =. (( {. $ r =. y.) , 2) $ 1 0
pic =. (r ,. 1) +/ . * x.
a =. 2 (<"1 (i =. (-. (* pic) = * 1 |. pic) # i. {. $ y.) ,. 1) } a
q =. |: ((_1 + $ x.) , $ pic) $ pic
q =. (((i { r) * i { 1 |. q) - (i { 1 |. r) * (i { q)) % (i { 1 |. q) - i { q
r =. (pic > 0) # r
a =. 0 (<"1 ((0 >: pic) # i. $ pic) ,. 0) } a
a =. (-. 0 = a) # a =. , a
r =. (/: /: a) { r , q
r , (-. (1 {. r) -: _1 {. r) # 1 {. r
)
```

Applying `pclip` we have:

```
   line pclip 0 1 2 3 0 { square
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

which is illustrated in Figure 2. We can clip to the other side of the line by:

```
   (-line) pclip 0 1 2 3 0 { square
 50  0
100  0
100 50
 50  0
```

The J code appears rather dense, but remember this array algorithm is based, in part, on a re-entrant algorithm which Sutherland and Hodgman take ten pages to explain. We explain the results formed in each of the ten lines of the program.

The first expression produces a table, named `a`, which will be used to code which side of the line/plane the points lie. A copy of the original data is also named (locally) `r` for later use. Initially, column 1 of `a` codes all of the points as being on the correct side of the clipping boundary with a value of 1. Later, points which are clipped will be coded with the value 0 in column 1 and new intersection points which must be added will be coded in column 2 with a value of 2.

```
   a =. (( {. $ r =. y.) , 2) $ 1 0
1 0
1 0
1 0
1 0
1 0
```

The second line produces a vector of values which indicate which side of the line/plane (a positive value indicates the correct side of the line/plane) by computing a matrix product of the homogeneous representation of the points and the line/plane. This result is named (locally) `pic`.

```
   pic =. (r ,. 1) +/ . * x.
50 _50 50 150 50
```

The third line modifies column 2 of the table `a` to indicate that the one point, (100,0), which lies on the wrong side of the clipping line/plane must be replaced by two points along the clipping boundary which are determined by computing the intersection with the clipping boundary.

```
   a =. 2 (<"1 (i =. (-. (* pic) = * 1 |. pic) # i. {. $ y.) ,. 1) } a
1 2
1 2
1 0
1 0
1 0
```

The fourth line builds a table, named `q`, of two columns which consist of the vector `pic`.

```
   q =. |: ((_1 + $ x.) , $ pic) $ pic
 50  50
_50 _50
 50  50
150 150
 50  50
```

The fifth line computes a table, named `q`, of the intersection points with the clipping boundary of all lines/planes which go outside the clipping boundary.

```
   q =. (((i { r) * i { 1 |. q) - (i { 1 |. r) * (i { q)) % (i { 1 |. q) - i { q
 50   0
100 50
```

The sixth line computes a table, re-named `r`, of the points which are on the correct side of the clipping boundary.

```
   r =. (pic > 0) # r
  0   0
100 100
  0 100
  0   0
```

The seventh line modifies the table, `a`, to indicate, with a zero in column one, which points are clipped from the original data.

```
   a =. 0 (<"1 ((0 >: pic) # i. $ pic) ,. 0) } a
1 2
0 2
1 0
1 0
1 0
```

The eighth line produces a vector of the non-zero elements in `a` in row major order. This array encodes with the value 1 the respective elements of `r` and uses the value 2 for the respective elements of `q` which are the newly computed boundary intersection points. This vector is the mesh vector for properly ordering the points not clipped and the boundary intersection points.

```
   a =. (-. 0 = a) # a =. , a
1 2 2 1 1 1
```

The ninth line contains the remarkable surprise that applying the grade-up function (`/:`) twice produces the proper ordering of elements taken from the combined tables `r` and `q`. Grade produces a permutation of the indices of an array which would sort the array in ascending order. So the ninth line re-arranges the elements in r and q by sorting a sort!

```
   (/: /: a) { r , q
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

This algorithm assumes that we start with a closed polygon. To ensure that the algorithm produces a closed polygon, the last line closes the polygon in the case where the first (and last) point is clipped.

```
   r , (-. (1 {. r) -: _1 {. r) # 1 {. r
  0   0
 50   0
100  50
100 100
  0 100
  0   0
```

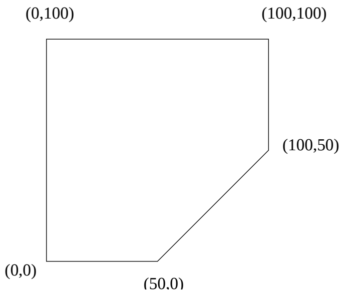

Figure 2: Square after clipping

3 Polygon Clipping of 3D Polygons

This algorithm works, without change, on 3D polygons. A 3D figure may be modeled by a collection polygon faces. To illustrate this process, we extend the 2D example of a square to a cube. Note that when modeling a cube we do not want to duplicate starting and end coordinates of each of the six polygon faces of the cube. We provide that information when clipping.

We expand our definition of a square to a cube by appending a column of zeros (the bottom of the cube is in the plane $z = 0$) followed another copy of the square after appending a column of 100's (the top of the cube is in the plane $z = 100$).

```
   [ cube =: (square ,. 0) , square ,. 100
  0   0   0
100   0   0
100 100   0
  0 100   0
  0   0 100
100   0 100
100 100 100
  0 100 100
```

We extend the line in 2-space, $y = x - 50$, (written as `_1 1 50` in homogeneous form) to the plane in 3-space, $y = x - 50$, (written as `_1 1 0 50` in homogeneous form).

```
   [ plane =: _1 1 0 50
_1 1 0 50
```

To clip the bottom of `cube` to `plane` we write:

```
   plane pclip 0 1 2 3 0 { cube
  0   0 0
 50   0 0
100  50 0
100 100 0
  0 100 0
  0   0 0
```

To clip the right-most face of `cube` to `plane` we write:

```
   plane pclip 1 2 6 5 1 { cube
100  50   0
100 100   0
100 100 100
100  50 100
100  50   0
```

The top face of the cube would be clipped by:

```
   plane pclip 4 5 6 7 4 { cube
  0   0 100
 50   0 100
100  50 100
100 100 100
  0 100 100
  0   0 100
```

References

[Hui 2001] Hui, Roger K. W., Iverson, Kenneth E., *J Dictionary*, J Software, Toronto, Canada, May 2001.

[Suth 1974] Sutherland, Ivan and Hodgman, Gary, “Re-entrant Polygon Clipping”, Communications of the ACM, 17(1):32-42, January 1974.