Chapter 7

Polygon Clipping

revised March 20, 2013
program code revisions & assignment to follow ...

7.1 The Polygon Clipping Algorithm

When a polygon is to be written to the display, it must be clipped against the current world-coordinate window boundary so that no effort is wasted in rendering any portion of the polygon which will not be visible upon the display. So that the polygon itself is not disturbed by this process, a copy of the polygon's vertices is made and handed over to a clipping algorithm. If the algorithm determines that some portion of the polygon will in fact be visible in the window, the clipped vertices are converted to device coordinates so that a device coordinate polygon may be created from them and then written to the display. This series of events is delineated by the `write` algorithm for world coordinate polygons that follows.

```
void PolygonWC2d::write (EnvMap& map) const
{
   PointList* clippedList = new PointList (vertices);
   bool visible = clip (map, *clippedList);
   if (visible)
   {
      PolygonDC polygon (clippedList->convertTOdc(map), getColor());
      polygon.write (map);
   }
   return;
}
```

If the copied vertex list for the polygon is referred to as the *clippee*, the current world coordinate window is then the *clipper*. The clipping algorithm begins be creating a second point list corresponding to corner vertices of the clipper window. With clippee and clipper point lists in hand, the clipping process may begin in earnest.

The essence of the polygon clipping algorithm is to reduce the problem from one of clipping a polygon against another polygon to one of clipping a polygon against a series of lines, which correspond to the sides of the clipper window. Thus a loop which iterates once for each clipper side is indicated. On each pass of this loop, the clippee vertices are modified to become those representing a new polygon lying wholly on the inside of the current clipper side; see Figure 7.1. This process will continue so long as each clip results in clippee

Figure 7.1: Clippee Modification After Clip Against One Side of Clipper

vertices on the inside of the current clipper side. The final clipped set of polygon vertices will lie to the inside of all clipper sides. If this set is not empty, the polygon is visible. The creation of the clipper and the iteration for each of its sides comprises the body of the first of the polygon `clip` algorithms.

```
bool PolygonWC2d::clip (const EnvMap& map, PointList& clippee) const
{
   PointList clipper (map);

   clipper.gotoStart();
   do  // for each side of clipper
   {
      PointList currentClip;
      clip (clippee, clipper, currentClip);

      clippee = currentClip;  // refining result on each pass

      clipper.gotoNext();
   }
   while (! clipper.atEnd() && ! clippee.empty());

   return clippee.getSize() > 0; // visible?
}
```

The process of clipping the clippee against one clipper side begins by determining the inside-outside status of each clippee vertex with respect to the clipper side in question; the details of this task will be taken up in the next section. The clipping problem is now reduced further to the clipping of one side of the clippee at a time against the clipper side; each clippee side is dealt with per iteration of a loop. The handling of the single clippee side is made simple when it is observed that there are precisely four states in which the clippee side can be with respect to the clipper side.

- The first clippee side vertex lies on the inside of the clipper side, as does the second.
- The first clippee side vertex lies on the inside of the clipper side, but the second vertex lies to the outside of the clipper side.
- The first clippee side vertex lies to the outside of the clipper side, while the second vertex lies on the inside of the clipper side.
- The first clipper side vertex lies to the outside of the clipper side, as does the second.

The problem then turns to determining a course of action in each of these situations. It is clear in the second and third cases that a third point, the intersection of the clippee edge as it goes from inside to outside of the clipper side and vice versa, is also of interest. It might at first seem reasonable in each case to keep

Figure 7.2: Clipping a Polygon Against a Single Clipper Side

the inside vertices, including the intersections, while discarding those that are outside. The first case would then retain both vertices for the clipped list. Consider however the situation illustrated in Figure 7.2. Both vertices v_0 and v_1 of polygon edge e_0 in the figure lie on the inside of the clipper side, as do both of the vertices v_1 and v_2 of polygon edge e_1. Were both vertices to be kept for each edge, vertex v_1 would go onto the clipped list twice. The solution to this problem is clear when one considers an entire clippee polygon that lies on the inside of the clipper side: keeping just the first or just the second vertex for each edge would result in the desired clipped list. The decision of first or second is arbitrary and chosen here as the second; the remaining logic would need to be altered accordingly if the first were kept each time.

Observe how an edge such as e_2 in Figure 7.2 illustrates the second case above of starting on the inside and going to the outside. Since the decision was made to keep the second inside vertex, neither vertex from edge e_2 goes on the clipped list; the edge's intersection with the clipper side however must.

Edge e_3 in Figure 7.2 depicts the third case above as it starts on the outside and continues to the inside. Because of the decision made to keep the second inside vertex, vertex v_4 of edge e_3 will be kept, but not until the edge's preceding intersection with the clipper side is added to the clipped list.

Edge e_5 in Figure 7.2 illustrates the fourth and final case, wherein both edge vertices lie to the outside of the clipper side. Clearly, this side contributes nothing to the final clipped list.

```
void PolygonWC2d::clip (PointList& clippee, PointList& clipper, PointList& currentClip) const
{
   clippee.insideOutside (clipper); // set flags for each clippee vertex

   if (! clippee.empty())
   {
      clippee.gotoStart();
      do  // for each edge of current clippee
      {
         if // both inside
            // keep second
         else if // first inside, second outside
            // keep intersection
         else if // first outside, second inside
            // keep intersection & second
         // else both outside
            // keep nothing
         clippee.gotoNext();
      }  // end do-while
      while (! clippee.atEnd());
   }  // end if clippee not empty
   return;
}
```

Figure 7.3: Inside-Outside Cross Product Test

7.2 Inside-Outside Test for a Convex Polygon

As the clip of the polygon proceeds from one window side to the next, the inside-or-outside status of all of the polygon's vertices must be determined with respect to the current window side. A series of vector cross products is used to determine the status of each. The first operand of each of these cross products is a vector along a window side in the counter-clockwise direction. The second operand is a vector from the beginning of the window side to the polygon vertex in question. When the cross product of the window side and the vector to the polygon vertex is formed, the fingers of the right hand curl in the direction of the turn from the window side to the polygon vertex vector; the thumb of the right hand will point up when inside and down when outside. See Figure 7.3 for an illustration. //If the current window side is defined by points w_i and w_{i+1},

```
void PointList::insideOutside (const PointList& clipper)
{
   PolyPoint w1 = clipper.getCurrentPoint();
   PolyPoint w2 = clipper.getNextPoint();

   Vector clipperEdge = w2 - w1;

   gotoStart ();
   do // for each clippee point
   {
      Point p = getCurrentPoint(); // clippee point

      // cross vector w1->w2 (clipperEdge) with w1->p (operator *)

      getCurrent()->setInsideClipper (true/false cross product test);

      gotoNext();
   }
   while (! atEnd()); // for each clippee point
   return;
}
```

7.3 Intersection of a Polygon Side with a Clipping Window Edge

The slope m of a line segment has the same value between its endpoints (x_1, y_1) and (x_2, y_2) as it does between any other pair of points on the line, say (x_1, y_1) and some arbitrary point (x, y).

$$\begin{aligned} m &= \frac{\Delta y}{\Delta x} && (7.1)\\ &= \frac{y_2 - y_1}{x_2 - x_1} && (7.2)\\ &= \frac{y - y_1}{x - x_1} && (7.3) \end{aligned}$$

Equating the last and the first yields an equation for a line which can be rewritten in standard form.

$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{\Delta y}{\Delta x} && (7.4)\\ (y - y_1)\Delta x &= (x - x_1)\Delta y && (7.5)\\ y\Delta x - y_1\Delta x &= x\Delta y - x_1\Delta y && (7.6)\\ (-\Delta y)x + (\Delta x)y &= -x_1\Delta y + y_1\Delta x && (7.7)\\ Ax + By &= D && (7.8) \end{aligned}$$

When intersecting an edge of the polygon clippee with a side of the window clipper, a solution may be obtained from the system of equations formed from their line equations.

$$\begin{aligned} A_{clippee}\, x + B_{clippee}\, y &= D_{clippee} && (7.9)\\ A_{clipper}\, x + B_{clipper}\, y &= D_{clipper} && (7.10) \end{aligned}$$

The solution may be stated by way of the determinants in Cramer's rule.

$$\begin{aligned} x_{intersection} &= \frac{\begin{vmatrix} D_{clippee} & B_{clippee} \\ D_{clipper} & B_{clipper} \end{vmatrix}}{\begin{vmatrix} A_{clippee} & B_{clippee} \\ A_{clipper} & B_{clipper} \end{vmatrix}} && (7.11)\\ y_{intersection} &= \frac{\begin{vmatrix} A_{clippee} & D_{clippee} \\ A_{clipper} & D_{clipper} \end{vmatrix}}{\begin{vmatrix} A_{clippee} & B_{clippee} \\ A_{clipper} & B_{clipper} \end{vmatrix}} && (7.12) \end{aligned}$$

Of course, no solution can exist when the denominator is zero; consider when this occurs.

$$\begin{aligned} \begin{vmatrix} A_{clippee} & B_{clippee} \\ A_{clipper} & B_{clipper} \end{vmatrix} &= 0 && (7.13)\\ A_{clippee}\, B_{clipper} - B_{clippee}\, A_{clipper} &= 0 && (7.14)\\ -\Delta y_{clippee}(\Delta x_{clipper}) - \Delta x_{clippee}(-\Delta y_{clipper}) &= 0 && (7.15)\\ -\Delta y_{clippee}\Delta x_{clipper} &= -\Delta x_{clippee}\Delta y_{clipper} && (7.16)\\ \frac{\Delta y_{clippee}}{\Delta x_{clippee}} &= \frac{\Delta y_{clipper}}{\Delta x_{clipper}} && (7.17) \end{aligned}$$

This conclusion states that there can be no intersection when the slope of the polygon clippee edge is equal to the slope of the window clipper side.

The position (x, y) of the polygon edge/window side intersection is of course of primary interest, but there is another object-oriented consideration here. If the point class used to represent the endpoints of the line segments should have descendant classes which add supplementary information (e.g., lighting), the "intersection" value of this information must be calculated, too. Since there is no way to anticipate what data future descendant classes may introduce, a data-independent fraction or percentage may be calculated here for use with any endpoint data. If the polygon edge is not vertical and thus has a non-zero Δx, the fraction may be calculated as a percentage of the x-distance from polygon side endpoint p_1 to the intersection.

$$fraction = \frac{x_{intersection} - p_{1_x}}{\Delta x} \tag{7.18}$$

If the line were vertical, the percentage could be calculated as a percentage of the y-distance on the polygon side (as long as it has vertical length).

$$fraction = \frac{y_{intersection} - p_{1_y}}{\Delta y} \tag{7.19}$$

If the polygon were somehow defined with two adjacent points as the same, both Δx and Δy would be zero. Note that the application of Cramer's rule above would have already made objection to this situation.

```
double PointList::intersection (const PointList& clipper) const
{
   // The object of this method is the point list of the polygon being clipped;
   // the "current" and "next" points of the list define the polygon edge being
   // clipped.

   Point p1 = getCurrentPoint(); // clippee = polygon edge
   Point p2 = getNextPoint();
   // now can use p1.get(X) and .get(Y), p2.get(X) and .get(Y)

   // The "current" and "next" points of the clipper (clipping window) define
   // the window edge being clipped against.

   Point w1 = clipper.getCurrentPoint(); // clipper = window side
   Point w2 = clipper.getNextPoint();
   // now can use w1.get(X) and .get(Y), w2.get(X) and .get(Y)

   // Goal: Find the fraction of the distance from p1 to p2 to the intersection
   //       of this edge (p1-p2) with the line segment w1-w2.

   double fraction;

   // solve for fraction

   return fraction;
}
```