CS 460

Computer Graphics

Professor Richard Eckert

March 17, 2004

A. Polygon Clipping
  – Weiler-Atherton Polygon Clipper

B. Clipping Other Primitives

C. Clipping Text

D. Modelling Complex Shapes
  – Parametric Polynomial Curves

Weiler-Atherton Polygon Clipper

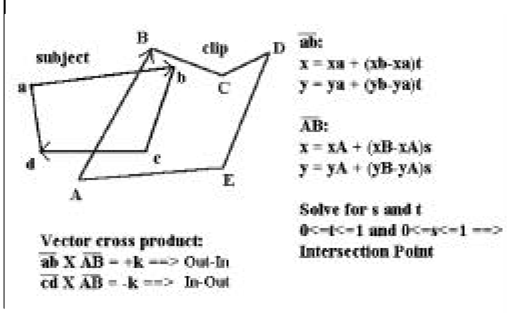
- Clips a "Subject Polygon" to a "Clip Polygon”
- Both polygons can be of any shape
- Result: one or more output polygons that lie entirely inside the clip polygon
- Basic idea:
  – Follow a path that may be a subject polygon edge or a clip polygon boundary

---

The Weiler-Atherton Algorithm

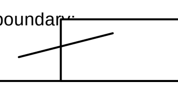
1. Set up vertex lists for <u>subject</u> and <u>clip</u> polygons
   Ordering: as you move down each list, inside of polygon is always on the right side (clockwise)
2. Compute all intersection points between subject polygon and clip polygon edges
   Insert them into each polygon's list
   Mark as intersection points
   Mark “out-in” intersection points
   (subject polygon edge moving from outside to inside of clip polygon edge)

Intersection Points & out-in Marking (General)



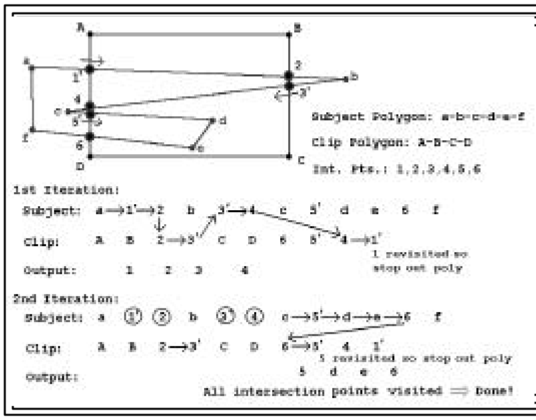
Intersection Points & Out-In Marking (Simple)

- If clip polygon is a rectangle:
  – Point in/out test
  – e.g., for intersection with left boundary:
    x<xmin means outside, x>=xmax means inside
- Intersections also easy
  – Use Cohen-Sutherland ideas
    • e.g., for intersection with left boundary:
      x = xmin
      y = m*(xmin-x2) + y2



Weiler-Atherton Algorithm, continued

3. Do until all intersection points have been visited:
   - Traverse subject polygon list until a non-visited out-in intersection point is found;
   - Output it to new output polygon list
   - Make subject polygon list be the active list
   - Do until a vertex is revisited:
     - Get next vertex from active list & output
     - If vertex is an intersection point,
       - make the other list active
   - End current output polygon list



BALSA VIDEO OF POLYGON CLIPPING ALGORITHMS

Clipping Other Curves

- Must compute intersection points between curve and clip boundaries
- In general solve nonlinear equations
- Many times approximation methods must be used
- Time consuming

Clipping Text

- Use successively more expensive tests
  1. Clip string
     Embed string in rectangle
     Clip rectangle (4 point tests)
     - entirely in ==> keep string
     - entirely out==>reject string
     - neither==>next test

2. Clip each Character
   Embed character in rectangle
   Clip rectangle (4 point tests)
   - entirely in ==> keep character
   - entirely out==>reject character
   - neither==>next test
3. Two possibilities for Character Clipping
   - Bitmapped: look at each pixel
   - Stroked: Apply line clipper to each stroke

Modeling Complex Shapes

- Can use line/polygon primitives to approximate
- But complex objects-->huge number of primitives
- Better to use more complex primitives
- Use curves (2-D) or surfaces (3-D)

Curves in Space

- Three forms:
  - Explicit
  - Implicit
  - Parametric

Explicit Form

- y = f(x)
- example--line:

  y = m*x + b

  But this is not a finite line segment
- Not all curves can be put into this form

Implicit Form

- f(x,y)=0
- Example--circle:

  $(x-h)^2 + (y-k)^2 - R^2 = 0$
- Indicates if a point x,y is on the curve
- Can be difficult to plot
- In some cases can be cast into explicit form

Parametric Form

- x and y expressed as explicit functions of a parameter, t

  x = f(t)

  y = g(t)
- Range of parameter also given
  - Delimits the extent of the curve
- To plot, let t vary over its range
  - Points on curve are generated
- Easily extended to curves in 3-D

  z = h(t)

Parametric Equations for a Line Segment

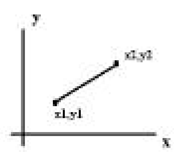
- Given endpoints P1(x1,y1), P2(x2,y2)
- Assume:

  t=0: endpoint P1

  t=1: endpoint P2
- Linear equation ==>

  x = a*t + b

  y = c*t + d
- Need to get constants a,b,c,d



x = a*t + b, y = c*t + d

- Apply boundary conditions:
  - t=0 ==> x=x1, y=y1
    - x1 = a*0 + b, so b=x1
    - y1 = c*0 + d, so d=y1
  - t=1 ==> x=x2, y=y2
    - x2 = a*1 + b, so a = x2 - b, or a = x2 - x1
    - y2 = c*1 + d, so c = y2 - d, or c = y2 - y1
- Resulting Parametric equations:
  - x = (x2-x1)*t + x1   **0<=t<=1**
  - y = (y2-y1)*t + y1

Polynomials

- Explicit Form of n-degree polynomial:
  - $y = a_0 + a_1{*}x + a_2{*}x^2 + \ldots\ a_n{*}x^n$
- Assume we have a set of n+1 known control points: (xi,yi)
- Get polynomial coefficients a_i from the control points
- Two Methods:
  - Interpolation
  - Approximation

Interpolating Polynomial, degree n

- Curve passes through all n+1 control points (xi,yi)
- Given (x0,y0), (x1,y1), (x2,y2) ... (xn,yn):

$y0 = a_0 + a_1{*}x0 + a_2{*}x0^2 \ldots\ a_n{*}x0^n$

$y1 = a_0 + a_1{*}x1 + a_2{*}x_1^{\,2} \ldots\ a_n{*}x1^n$

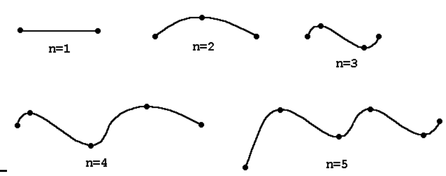
…

$yn = a_0 + a_1{*}xn + a_2{*}xn^2 \ldots\ a_n{*}xn^n$

- n+1 equations in n+1 unknown constants: $a_0, a_1, a_2, \ldots\ a_n$

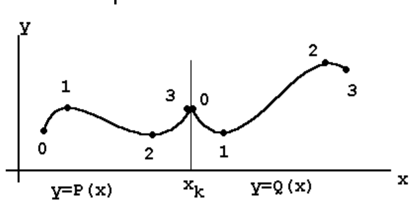
May not be good in graphics

- Many control points==>high degree polynomial
- Many calculations
- Polynomial "wiggle"



Segmented Interpolating Polynomials

- Break curve into segments
- Each with different low-degree polynomial
- Easier computations



Joining Segmented Curves

- Join points called knots
- kth knot at $x=x_k$
- Level-0 continuity: $P(x_k)=Q(x_k)$
  - Continuous, but not smooth (kinks)
- Level-1 continuity: $P'(x_k)=Q'(x_k)$
  - First derivative-->smoother curve
- Level-2 conitnuity: $P''(x_k)=Q''(x_k)$
  - Second derivative-->still smoother

Approximating Polynomials

- Curve determined by control points
- But does NOT go through all of them
- Control Points act as magnets
- Better for many graphics applications
- Most commonly used:
  - Bezier curves
  - B-spline curves

- Bezier Curves
  - See CS-460/560 Notes:
  - Bezier Polynomial Curves
  - http://www.cs.binghamton.edu/~reckert/460/bezier.htm
- B-Spline Curves
  - See CS-460/560 Notes:
  - B-spline Polynomial Curves
  - http://www.cs.binghamton.edu/~reckert/460/bspline.htm

Bezier Polynomial Curves

- Parametric equations for a 2-D cubic polynomial curve:

  $x = ax*t^3 + bx*t^2 + cx*t + dx$

  $y = ay*t^3 + by*t^2 + cy*t + dy$

  $0<=t<=1$
- Shape of curve determined by polynomial coefficients:
  - (ax,bx,cx,dx, ay,by,cy,dy)

Easily extended to 3-D

- Just add a third parametric equation:

  $z = az*t^3 + bz*t^2 + cz*t + dz$

Control Points

- Want to easily determine shape of curve
- Specify four control points:
  - P0 (x0,y0,z0), P1(x1,y1,z1), P2(x2,y2,z2), P3(x3,y3,z3)
- Could use interpolating polynomial
- More useful: approximating polynomial
  - Doesn't interpolate all control points
  - Many ways to do the approximating

Uniform Cubic Bezier Polynomial

- Important kind of approximating polynomial
- Assume a generic parametric cubic polynomial:

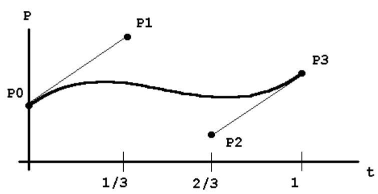
  $P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$
- Determined by control points P0, P1, P2, P3
  - P could be x, y, or z
  - a could be ax, ay, or az
  - P0 could be x0, y0, z0, etc.

<u>Uniform</u> Bezier Polynomial

$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

- Control points uniformly separated in t

  P0 at t=0, P1 at t=1/3, P2 at t=2/3, P3 at t=1

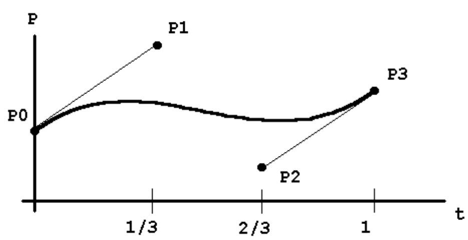


Boundary conditions:

$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

1. Curve must interpolate control point P0
   - P=P0 when t=0
   - So P0 = d
2. Curve must interpolate control point P3
   - P=P3 when t=1
   - so P3 = a + b + c + d

Uniform Cubic Bezier Curve



$P = a*t^3 + b*t^2 + c*t + d, \quad 0 <= t <= 1$

3. Slope of curve at t=0 must be equal to that of the line that joins control points P0 and P1
   - dP/dt(at t=0) = slope of P0-P1
   - $dP/dt = 3*a*t^2 + 2*b*t + c$
   - slope of P0-P1 = (P1-P0)/(1/3-0)
   - So: c = 3*(P1-P0)
4. Slope of curve at t=1 must be equal to that of the line that joins control points P2 and P3
   - dP/dt(at t=1) = slope of P2-P3
   - 3*a + 2*b + c = (P3-P2)/(1 - 2/3)
   - 3*a + 2*b + c = 3*(P3-P2)

Solving for Polynomial Coefficients

- Equations:

```
0   +  0   +  0   +  d   = P0
a   +  b   +  c   +  d   = P3
0   +  0   +  c   +  0   = 3*(P1-P0)
3*a +  2*b +  c   +  0   = 3*(P3-P2)
```