University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2005

Tamara Munzner

Interpolation
Clipping

Week 7, Mon Feb 21

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2005

News

- grades for p1, h2 posted

Interpolation

Scan Conversion

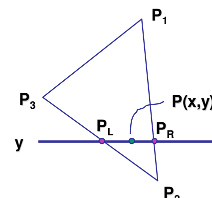
- done
  - how to determine pixels covered by a primitive
- next
  - how to assign pixel colors
    - interpolation of colors across triangles
    - interpolation of other properties

Interpolation During Scan Conversion

- interpolate values between vertices
  - z values
  - r,g,b colour components
    - use for Gouraud shading
  - u,v texture coordinates
  - N_x, N_y, N_z surface normals
- equivalent methods (for triangles)
  - bilinear interpolation
  - barycentric coordinates

Bilinear Interpolation

- interpolate quantity along L and R edges, as a function of y
  - then interpolate quantity as a function of x



Barycentric Coordinates

- weighted combination of vertices

$$\begin{cases} P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \\ \alpha + \beta + \gamma = 1 \\ 0 \le \alpha, \beta, \gamma \le 1 \end{cases}$$

"convex combination of points"

P_1 (1,0,0)

$\beta = 0$

$\beta = 0.5$

(0,0,1)

P_3

P

$\beta = 1$

P_2 (0,1,0)

7

Computing Barycentric Coordinates

- for point P on scanline

$$P_L = P_2 + \frac{d_1}{d_1+d_2}(P_3 - P_2)$$

$$= (1 - \frac{d_1}{d_1+d_2})P_2 + \frac{d_1}{d_1+d_2}P_3 =$$

$$= \frac{d_2}{d_1+d_2}P_2 + \frac{d_1}{d_1+d_2}P_3$$

8

Computing Barycentric Coordinates

- similarly

$$P_R = P_2 + \frac{b_1}{b_1+b_2}(P_1 - P_2)$$

$$= (1 - \frac{b_1}{b_1+b_2})P_2 + \frac{b_1}{b_1+b_2}P_1 =$$

$$= \frac{b_2}{b_1+b_2}P_2 + \frac{b_1}{b_1+b_2}P_1$$

9

Computing Barycentric Coordinates

- combining

$$P = \frac{c_2}{c_1+c_2} \cdot P_L + \frac{c_1}{c_1+c_2} \cdot P_R$$

$$P_L = \frac{d_2}{d_1+d_2}P_2 + \frac{d_1}{d_1+d_2}P_3$$

$$P_R = \frac{b_2}{b_1+b_2}P_2 + \frac{b_1}{b_1+b_2}P_1$$

- gives

$$P = \frac{c_2}{c_1+c_2}\left(\frac{d_2}{d_1+d_2}P_2 + \frac{d_1}{d_1+d_2}P_3\right) + \frac{c_1}{c_1+c_2}\left(\frac{b_2}{b_1+b_2}P_2 + \frac{b_1}{b_1+b_2}P_1\right)$$

10

Computing Barycentric Coordinates

$$P = \frac{c_2}{c_1+c_2}\left(\frac{d_2}{d_1+d_2}P_2 + \frac{d_1}{d_1+d_2}P_3\right) + \frac{c_1}{c_1+c_2}\left(\frac{b_2}{b_1+b_2}P_2 + \frac{b_1}{b_1+b_2}P_1\right)$$

- thus $P = a_1 \cdot P_1 + a_2 \cdot P_2 + a_3 \cdot P_3$

with

$$a_1 = \frac{c_1}{c_1+c_2}\frac{b_1}{b_1+b_2}$$

$$a_2 = \frac{c_2}{c_1+c_2}\frac{d_2}{d_1+d_2} + \frac{c_1}{c_1+c_2}\frac{b_2}{b_1+b_2}$$

$$a_3 = \frac{c_2}{c_1+c_2}\frac{d_1}{d_1+d_2}$$

11

Computing Barycentric Coordinates

- can verify barycentric properties

$$a_1 + a_2 + a_3 = 1$$

$$0 \le a_1, a_2, a_3 \le 1$$

$$a_1 = \frac{c_1}{c_1+c_2}\frac{b_1}{b_1+b_2}$$

$$a_2 = \frac{c_2}{c_1+c_2}\frac{d_2}{d_1+d_2} + \frac{c_1}{c_1+c_2}\frac{b_2}{b_1+b_2}$$

$$a_3 = \frac{c_2}{c_1+c_2}\frac{d_1}{d_1+d_2}$$

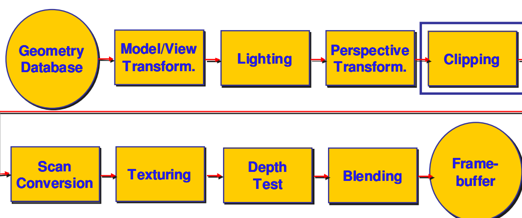
12

Clipping

Reading

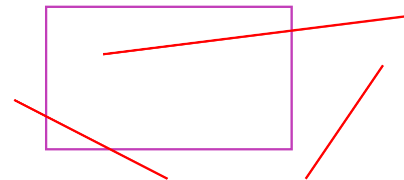
- FCG Chapter 11
  - pp 209-214 only: clipping

Rendering Pipeline



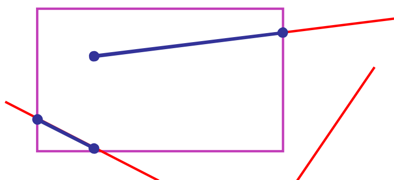
Next Topic: Clipping

- we've been assuming that all primitives (lines, triangles, polygons) lie entirely within the *viewport*
  - in general, this assumption will not hold:



Clipping

- analytically calculating the portions of primitives within the viewport



Why Clip?

- bad idea to rasterize outside of framebuffer bounds
- also, don't waste time scan converting pixels outside window
  - could be billions of pixels for very close objects!

Line Clipping

- 2D
  - determine portion of line inside an axis-aligned rectangle (screen or window)
- 3D
  - determine portion of line inside axis-aligned parallelpiped (viewing frustum in NDC)
  - simple extension to 2D algorithms

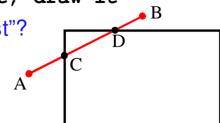
19

Clipping

- naïve approach to clipping lines:

```
for each line segment
    for each edge of viewport
        find intersection point
        pick "nearest" point
    if anything is left, draw it
```

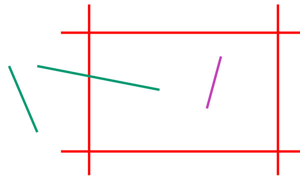
- what do we mean by "nearest"?
- how can we optimize this?



20

Trivial Accepts

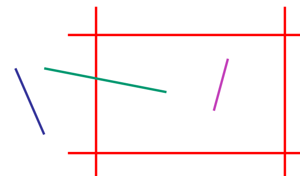
- big optimization: trivial accept/rejects
  - Q: how can we quickly determine whether a line segment is entirely inside the viewport?
  - A: test both endpoints



21

Trivial Rejects

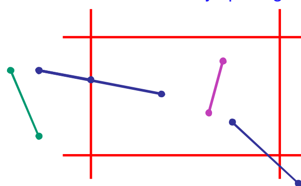
- Q: how can we know a line is outside viewport?
- A: if both endpoints on wrong side of same edge, can trivially reject line



22

Clipping Lines To Viewport

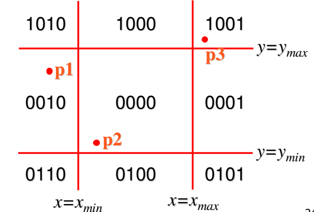
- combining trivial accepts/rejects
  - trivially accept lines with both endpoints inside all edges of the viewport
  - trivially reject lines with both endpoints outside the same edge of the viewport
  - otherwise, reduce to trivial cases by splitting into two segments



23

Cohen-Sutherland Line Clipping

- outcodes
  - 4 flags encoding position of a point relative to top, bottom, left, and right boundary
    - $OC(p1)$=0010
    - $OC(p2)$=0000
    - $OC(p3)$=1001



24

Cohen-Sutherland Line Clipping

- assign outcode to each vertex of line to test
  - line segment: (**p1**,**p2**)
- trivial cases
  - OC(**p1**)== 0 && OC(**p2**)==0
    - both points inside window, thus line segment completely visible (trivial accept)
  - (OC(**p1**) & OC(**p2**))!= 0
    - there is (at least) one boundary for which both points are outside (same flag set in both outcodes)
    - thus line segment completely outside window (trivial reject)

25

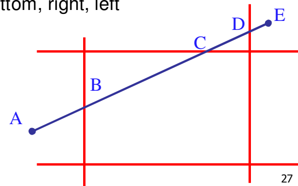
Cohen-Sutherland Line Clipping

- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses (*how?*)
- intersect line with edge (*how?*)
- discard portion on wrong side of edge and assign outcode to new vertex
- apply trivial accept/reject tests; repeat if necessary

26

Cohen-Sutherland Line Clipping

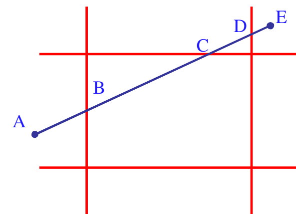
- if line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- pick an edge that the line crosses
  - check against edges in same order each time
    - for example: top, bottom, right, left



27

Cohen-Sutherland Line Clipping

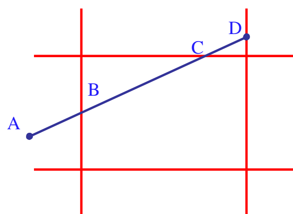
- intersect line with edge (how?)



28

Cohen-Sutherland Line Clipping

- discard portion on wrong side of edge and assign outcode to new vertex

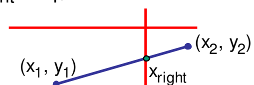


- apply trivial accept/reject tests and repeat if necessary

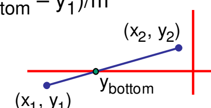
29

Viewport Intersection Code

- (x_1, y_1), (x_2, y_2) intersect vertical edge at x_{right}
  - $y_{intersect} = y_1 + m(x_{right} - x_1)$
  - $m=(y_2-y_1)/(x_2-x_1)$



- (x_1, y_1), (x_2, y_2) intersect horiz edge at y_{bottom}
  - $x_{intersect} = x_1 + (y_{bottom} - y_1)/m$
  - $m=(y_2-y_1)/(x_2-x_1)$



30

Cohen-Sutherland Discussion

- use opcodes to quickly eliminate/include lines
  - best algorithm when trivial accepts/rejects are common
- must compute viewport clipping of remaining lines
  - non-trivial clipping cost
  - redundant clipping of some lines
- more efficient algorithms exist

31

Line Clipping in 3D

- approach
  - clip against parallelpiped in NDC
    - after perspective transform
  - means that clipping volume always the same
    - xmin=ymin= -1, xmax=ymax= 1 in OpenGL
  - boundary lines become boundary planes
    - but outcodes still work the same way
    - additional front and back clipping plane
      - zmin = -1, zmax = 1 in OpenGL

32

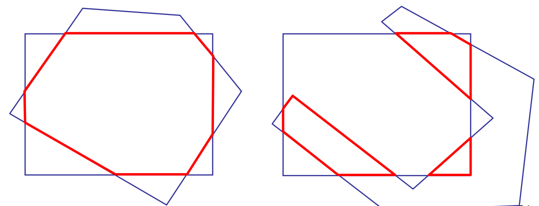
Polygon Clipping

- objective
  - 2D: clip polygon against rectangular window
    - or general convex polygons
    - extensions for non-convex or general polygons
  - 3D: clip polygon against parallelpiped

33

Polygon Clipping

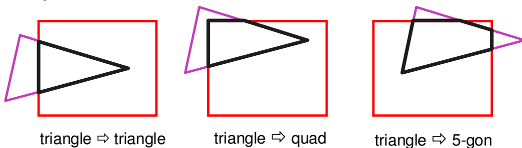
- not just clipping all boundary lines
  - may have to introduce new line segments



34

Why Is Clipping Hard?

- what happens to a triangle during clipping?
- possible outcomes:

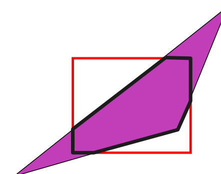


- how many sides can a clipped triangle have?

35

How Many Sides?

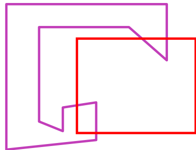
- seven…



36

Why Is Clipping Hard?

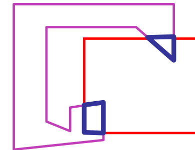
- a really tough case:



37

Why Is Clipping Hard?

- a really tough case:



concave polygon ⇨ multiple polygons

38

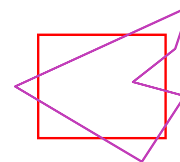
Polygon Clipping

- classes of polygons
  - triangles
  - convex
  - concave
  - holes and self-intersection

39

Sutherland-Hodgeman Clipping

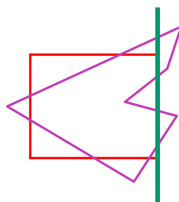
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



40

Sutherland-Hodgeman Clipping

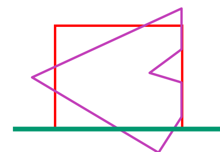
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



41

Sutherland-Hodgeman Clipping

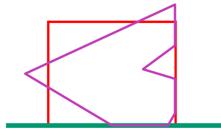
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



42

Sutherland-Hodgeman Clipping

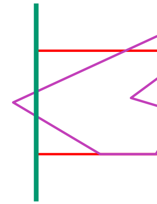
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



43

Sutherland-Hodgeman Clipping

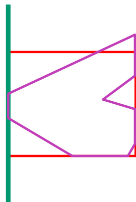
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



44

Sutherland-Hodgeman Clipping

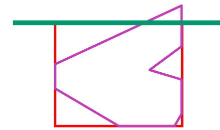
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



45

Sutherland-Hodgeman Clipping

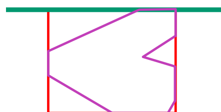
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



46

Sutherland-Hodgeman Clipping

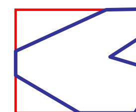
- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



47

Sutherland-Hodgeman Clipping

- basic idea:
  - consider each edge of the viewport individually
  - clip the polygon against the edge equation
  - after doing all edges, the polygon is fully clipped



48

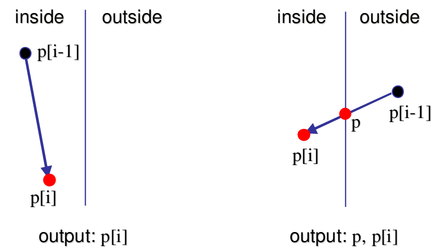
Sutherland-Hodgeman Algorithm

- input/output for algorithm
  - input: list of polygon vertices in order
  - output: list of clipped poygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- basic routine
  - go around polygon one vertex at a time
  - decide what to do based on 4 possibilities
    - is vertex inside or outside?
    - is previous vertex inside or outside?

49

Clipping Against One Edge

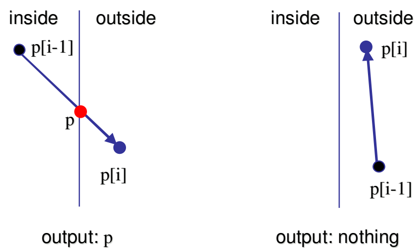
- p[i] inside: 2 cases



50

Clipping Against One Edge

- p[i] outside: 2 cases



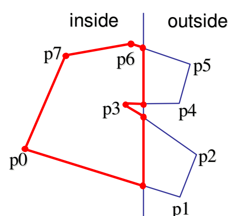
51

Clipping Against One Edge

```
clipPolygonToEdge( p[n], edge ) {
    for( i= 0 ; i< n ; i++ ) {
        if( p[i] inside edge ) {
            if( p[i-1] inside edge ) output p[i];      // p[-1]= p[n-1]
            else {
                p= intersect( p[i-1], p[i], edge ); output p, p[i];
            }
        } else {                                       // p[i] is outside edge
            if( p[i-1] inside edge ) {
                p= intersect(p[i-1], p[I], edge ); output p;
            }
        }
    }
}
```

52

Sutherland-Hodgeman Example



53

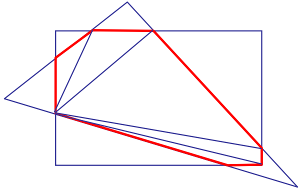
Sutherland-Hodgeman Discussion

- similar to Cohen/Sutherland line clipping
  - inside/outside tests: outcodes
  - intersection of line segment with edge: window-edge coordinates
- clipping against individual edges independent
  - great for hardware (pipelining)
  - all vertices required in memory at same time
    - not so good, but unavoidable
    - another reason for using triangles only in hardware rendering

54

Sutherland/Hodgeman Discussion

- for rendering pipeline:
  - re-triangulate resulting polygon
    (can be done for every individual clipping edge)



55