Implementation

Objectives

- Introduce basic implementation strategies
- Clipping
- Scan conversion
- Introduce clipping algorithms for polygons
- Survey hidden-surface algorithms

Implementation (ctd)

Objectives

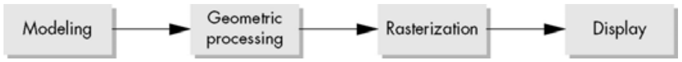
- Survey Line Drawing Algorithms
  - DDA
  - Bresenham

Meta Algorithms

- Consider two approaches to rendering a scene with opaque objects
- For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel
  - Ray tracing paradigm
- For every object, determine which pixels it covers and shade these pixels
  - Pipeline approach
  - Must keep track of depths

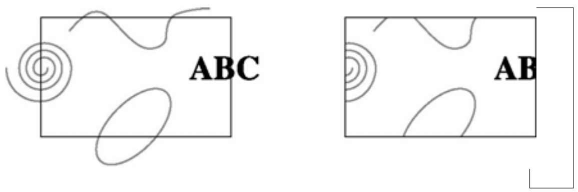
Common Tasks

- Clipping
- Rasterization or scan conversion
- Antialiasing
- Transformations
- Hidden surface removal



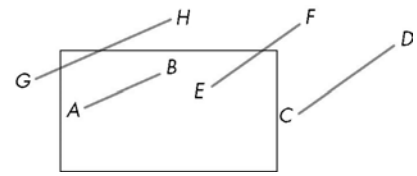
Clipping

- 2D against clipping window
- 3D against clipping volume
- Easy for line segments polygons
- Hard for curves and text
  - Convert to lines and polygons first



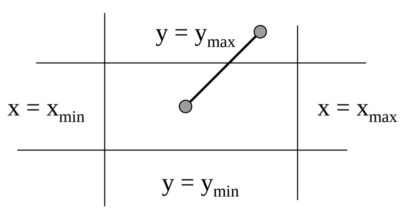
Clipping 2D Line Segments

- Brute force approach: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection



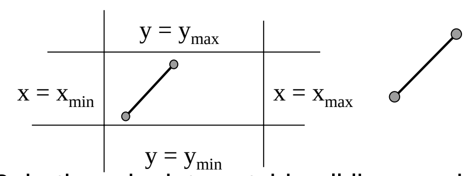
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

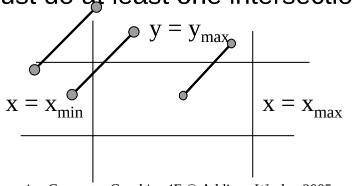
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of some line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside all lines but not on same side of any line
  - May have part inside
  - Must do at least one intersection

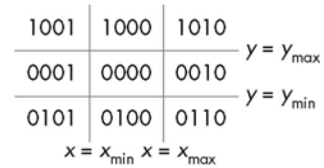


Defining Outcodes

- For each endpoint, define an outcode

$b_0b_1b_2b_3$

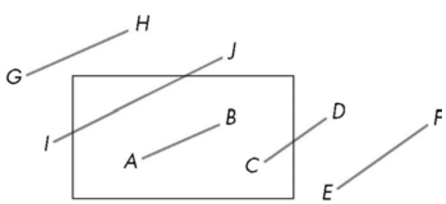
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

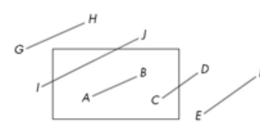
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - Accept line segment



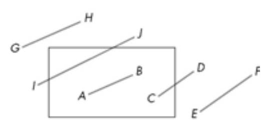
Using Outcodes

- CD: outcode (C)= 0, outcode(D)=0010$\neq$ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interesections



Using Outcodes

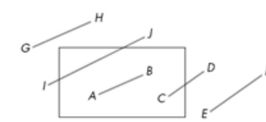
- EF: outcode(E) logically ANDed with outcode(F) (bitwise) ≠ 0
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - reject



 KENT STATE

Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm



 KENT STATE

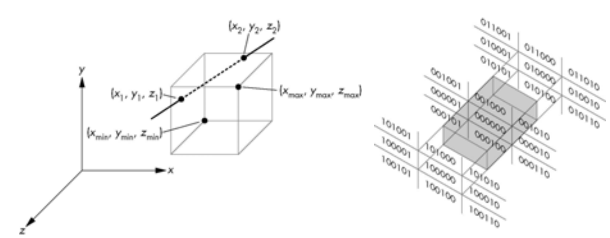
Efficiency

- In many applications, the clipping window is small relative to the size of the whole data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

 KENT STATE

Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

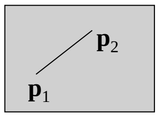


 KENT STATE

Liang-Barsky Clipping

- Consider the parametric form of a line segment

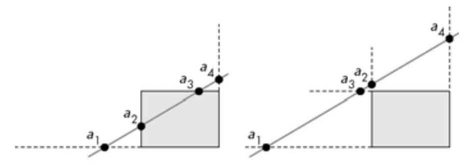
$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$



- We can distinguish between the cases by looking at the ordering of the values of α where the line determined by the line segment crosses the lines that determine the window

Liang-Barsky Clipping

- In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$
  - Intersect right, top, left, bottom: shorten
- In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$
  - Intersect right, left, top, bottom: reject

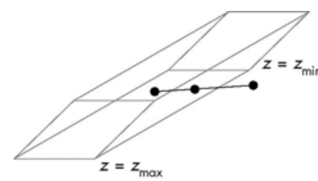


Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Decisions can be made without calculating intersections (without divisions)
- Using values of α, we do not have to use algorithm recursively as with C-S
- Extends to 3D

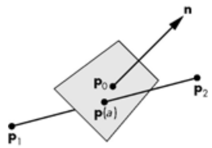
Clipping and Normalization

- General clipping in 3D requires intersection of line segments against arbitrary plane
- Example: oblique view



Plane-Line Intersections

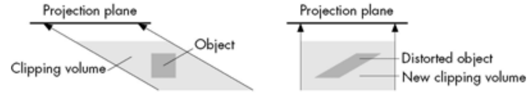
- Intersection requires 6 multiplications and a division



$$a = \frac{n \bullet (p_o - p_1)}{n \bullet (p_2 - p_1)}$$

Normalized Form

top view



before normalization after normalization

Normalization is part of viewing (pre clipping) but after normalization, we clip against sides of right parallelepiped

Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$?

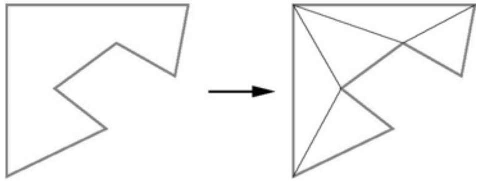
Polygon Clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons

- However, clipping a convex polygon can yield at most one other polygon

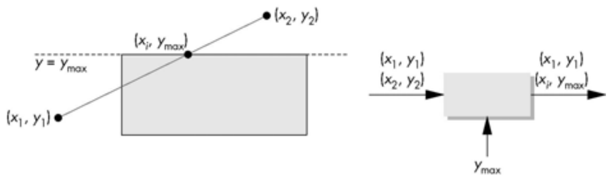
Tessellation and Convexity

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation code in GLU library



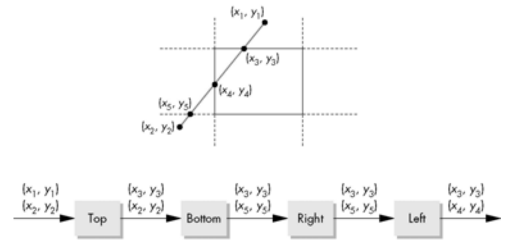
Clipping as a Black Box

- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment

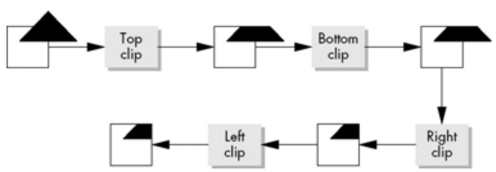


Pipeline Clipping of Line Segments

- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline



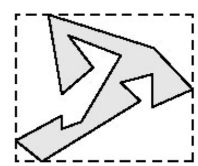
Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

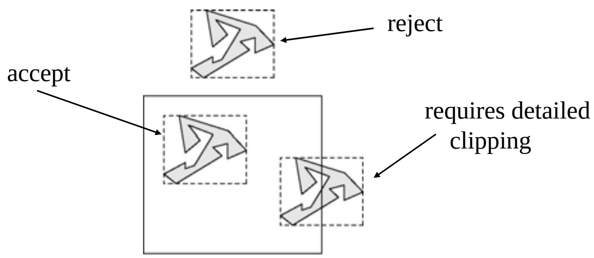
Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - Smallest rectangle aligned with axes that encloses the polygon
  - Simple to compute: max and min of x and y



Bounding boxes

Can usually determine accept/reject based only on bounding box

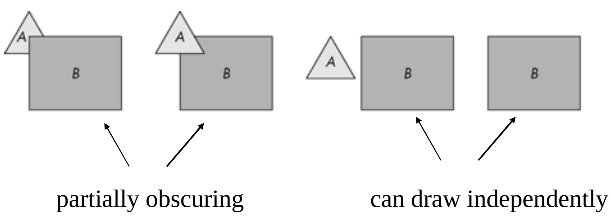


Clipping and Visibility

- Clipping has much in common with hidden-surface removal
- In both cases, we are trying to remove objects that are not visible to the camera
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

Hidden Surface Removal

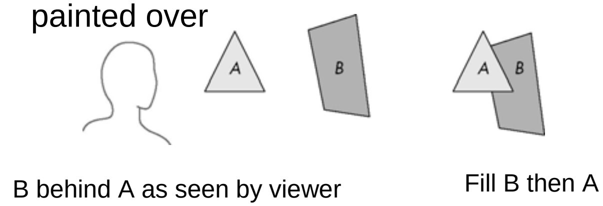
- Object-space approach: use pairwise testing between polygons (objects)



- Worst case complexity $O(n^2)$ for n polygons

Painter's Algorithm

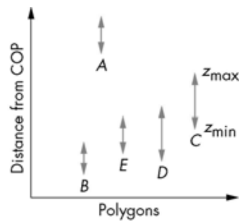
- Render polygons in back to front order so that polygons behind others are simply painted over



Depth Sort

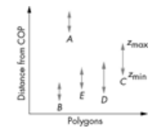
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP



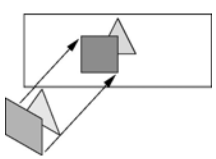
Easy Cases

- A lies behind all other polygons
  - Can render



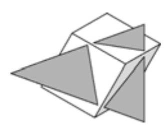
- Polygons overlap in z but not in either x or y
  - Can render independently

Hard Cases



Overlap in all directions but one is fully on one side of the other

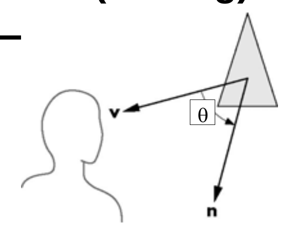
cyclic overlap



penetration

Back-Face Removal (Culling)

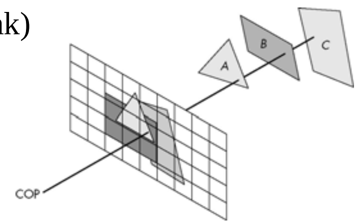
- face is visible iff $90 \geq \theta \geq -90$ equivalently $\cos\theta \geq 0$ or $\mathbf{v} \cdot \mathbf{n} \geq 0$



- plane of face has form $ax + by + cz + d = 0$ but after normalization $\mathbf{n} = (0\ 0\ 1\ 0)^T$
- need only test the sign of c
- In OpenGL we can simply enable culling but may not work correctly if we have nonconvex objects

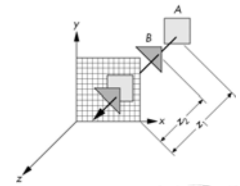
Image Space Approach

- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity O(nmk)
- Variations
  - Ray tracing
  - z-buffer



Z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Efficiency

- If we work scan line by scan line as we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$

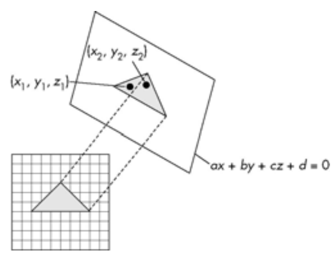
Along scan line

$\Delta y = 0$

$\Delta z = - \frac{a}{c} \Delta x$

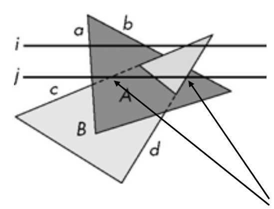
In screen space $\Delta x = 1$

So only need $\frac{a}{c}$ (a const)



Scan-Line Algorithm

- Can combine shading and hidden surface removal through scan line algorithm



scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

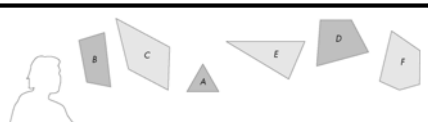
Implementation

- Need a data structure to store
  - Flag for each polygon (inside/outside)
  - Incremental structure for scan lines that stores which edges are encountered
  - Parameters for planes

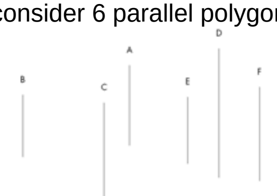
Visibility Testing

- In many realtime applications, such as games, we want to eliminate as many objects as possible within the application
  - Reduce burden on pipeline
  - Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree

Simple Example



consider 6 parallel polygons

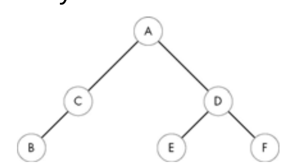


top view

The plane of A separates B and C from D, E and F

BSP Tree

- Can continue recursively
  - Plane of C separates B from A
  - Plane of D separates E and F
- Can put this information in a BSP tree
  - Use for visibility and occlusion testing



Rasterization

- Rasterization (scan conversion)
  - Shade pixels that are inside object specified by a set of vertices
    - Line segments
    - Polygons: scan conversion = fill
- Shades determined by color, texture, shading model
- Here we study algorithms for determining the correct pixels starting with the vertices

Rasterization or Scan Conversion of Lines

- Such a line should ideally have the following properties.
  - Straight,
  - pass through endpoints
  - smooth
  - independent of endpoint order
  - uniform brightness
  - brightness independent of slope
  - efficient

Line Drawing - Algorithm 1

A Straightforward Implementation

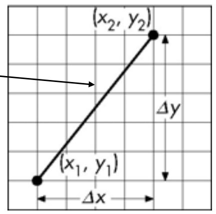
```
Drawline(x1,y1,x2,y2)
int x1,y1,x2,y2;
{
  float y;
  int x;

  for (x=x1; x<=x2; x++) {
    y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
    SetPixel(x, Round(y) );
  }
}
```


Scan Conversion of Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function

y = mx + h



$$m = \frac{\Delta y}{\Delta x}$$

DDA Algorithm

- <u>D</u>igital <u>D</u>ifferential <u>A</u>nalyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line y=mx+ h satisfies differential equation

    $dy/dx = m = \Delta y/\Delta x = y_2\text{-}y_1/x_2\text{-}x_1$
- Along scan line $\Delta x = 1$

```
For(x=x1; x<=x2, x++) {
    y+=m;
  write_pixel(x, round(y), line_color)
}
```


Line Drawing - Algorithm 2

A Better Implementation

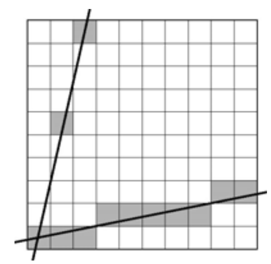
```
DrawLine(x1,y1,x2,y2)
int x1,y1,x2,y2;
{
  float m,y;
  int dx,dy,x;
  dx = x2 - x1;
  dy = y2 - y1;
  m = dy/dx;
  y = y1 + 0.5;
  for (x=x1; x<=x2; x++) {
    SetPixel(x, Floor(y) );
    y = y + m;
  }
}
```


Line Drawing Algorithm Comparison

- Advantages over Algorithm 1
  - eliminates multiplication
  - improves speed
- Disadvantages
  - round-off error builds up
  - get pixel drift
  - rounding and fp arithmetic still time consuming
  - works well only for |m| < 1
  - need to loop in y for |m| > 1
  - need to handle special cases

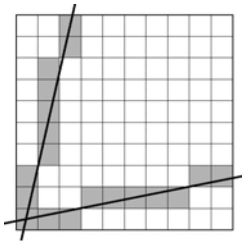
Problem

- DDA = for each x plot pixel at closest y
  - Problems for steep lines



Using Symmetry

- Use for $1 \geq m \geq 0$
- For $m > 1$, swap role of x and y
  - For each y, plot closest x

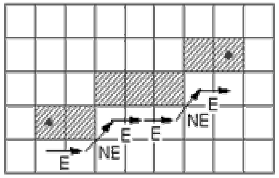


Based on Implicit Representation

- Explicit: $y = f(x)$
  - $y = m(x - x0) + y0$ where $m = dy/dx$
- Implicit: $f(x,y) = 0$
  - $F(x,y) = (x-x0)dy - (y-y0)dx$
  - if $F(x,y) = 0$ then (x,y) is on line
  - $F(x,y) > 0$ then (x,y) is below line
  - $F(x,y) < 0$ then (x,y) is above line

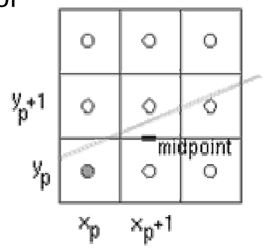
Line Drawing - Midpoint Algorithm

- The Midpoint or Bresenham's Algorithm
  - Uses only integer calculations. It treats line drawing as a sequence of decisions. For each pixel that is drawn the next pixel will be either E or NE, as shown below.



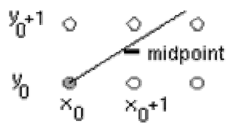
Midpoint Algorithm

- The midpoint algorithm makes use of the the implicit definition of the line, $F(x,y) = 0$. The N/NE decisions are made as follows.
- $d = F(x_p + 1, y_p + 0.5)$
  - if $d < 0$ line below midpt choose E
  - if $d > 0$ line above midpt choose NE
- if E is chosen
  - $d_{new} = F(x_p + 2, y_p + 0.5)$
  - $d_{new} - d_{old} = F(x_p + 2, y_p + 0.5) -$
    - $F(x_p + 1, y_p + 0.5)$
  - $\Delta d = d_{new} - d_{old} = dy$



Midpoint Algorithm

- If NE is chosen
  - $d_{new} = F(x_p + 2, y_p + 1.5)$
  - $\Delta d = dy - dx$
- Initialization
  - $d_{start} = F(x_0 + 1, y_0 + 0.5) = (x_0 + 1 - x_0)\,dy - (y_0 + 0.5 - y_0)dx = dy - dx/2$
- Integer only algorithm
  - $F'(x,y) = 2\,F(x,y)$; $d' = 2d$
  - $d'_{start} = 2dy - dx$
  - $\Delta d' = 2\Delta d$



Midpoint Algorithm for x1 < x2 and slope <= 1

```
drawline(x1, y1, x2, y2, colour)
int x1, y1, x2, y2, colour;
{
  int dx, dy, d, incE, incNE, x, y;

  dx = x2 - x1;
  dy = y2 - y1;
  d = 2*dy - dx;
  incE = 2*dy;
  incNE = 2*(dy - dx);
  y = y1;
  for (x=x1; x<=x2; x++) {
    setpixel(x, y, colour);
    if (d>0) {
      d = d + incNE;
      y = y + 1;
    } else {
      d = d + incE;
    } } }
```


General Bresenham's Algorithm

- To generalize to lines with arbitrary slope
  - consider symmetry between various octants and quadrants
  - for m > 1, interchange roles of x and y, that is step in y direction, and decide whether x value is above or below line
  - if m > 1, and right endpoint is the first point, both x and y decrease. To ensure uniqueness, independent of direction, always choose upper (or lower) point if the line goes through the mid-point
  - handle special cases without invoking algorithm: horizontal, vertical and diagonal lines

Additional Issues

- End-point order
  - cannot just interchange end-points
  - does not work when we use line styles since we need the pattern to go the same way on all segments of a polygon
- varying the intensity of a line with the slope
  - consider horizontal line and diagonal line
  - both have same number of pixels
  - diagonal √2 times horizontal line in length
  - intensity per unit length less for diagonal

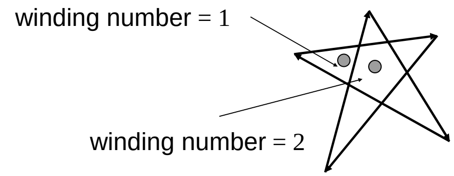
Polygon Scan Conversion

- Scan Conversion = Fill
- How to tell inside from outside
  - Convex easy
  - Nonsimple difficult
  - Odd even test
    - Count edge crossings
  - Winding number

odd-even fill

Winding Number

- Count clockwise encirclements of point



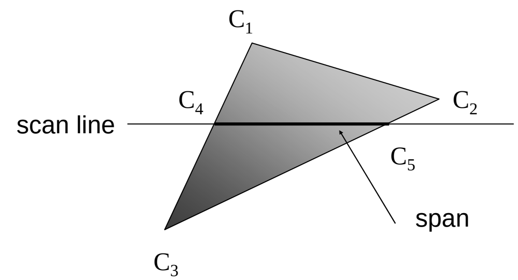
- Alternate definition of inside: inside if winding number $\neq 0$

Filling in the Frame Buffer

- Fill at end of pipeline
  - Convex Polygons only
  - Nonconvex polygons assumed to have been tessellated
  - Shades (colors) have been computed for vertices (Gouraud shading)
  - Combine with z-buffer algorithm
    - March across scan lines interpolating shades
    - Incremental work small

Using Interpolation

C_1 C_2 C_3 specified by **`glColor`** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



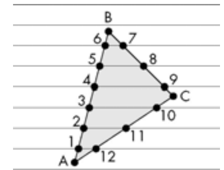
Flood Fill

- Fill can be done recursively if we know a seed point located inside, currently background color (WHITE)
- Scan convert edges into buffer in edge/fill color (BLACK)

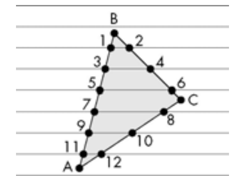
```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
       write_pixel(x,y,BLACK);
       flood_fill(x-1, y);
       flood_fill(x+1, y);
       flood_fill(x, y+1);
       flood_fill(x, y-1);
}   }
```


Scan Line Fill

- Can also fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort by scan line
  - Fill each span



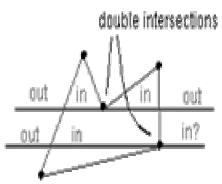
vertex order generated by vertex list



desired order

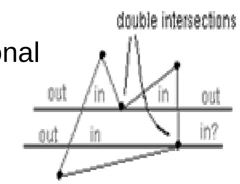
Scan Conversion Algorithm

- intersect each scan-line with all edges
- sort intersections by increasing x coordinate
- calculate parity of intersections to determine in/out
  - parity starts even - each intersection inverts
- fill the 'in' pixels - those with odd parity
- General issues - how to handle intersection at integer and fractional



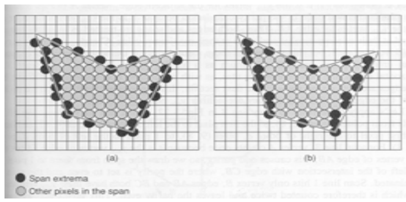
Scan Conversion Algorithm

- General issues - how to handle intersection at integer and fractional x values
- Special cases:
  - shared vertices lying on scan-lines - double intersections
    - count y_{min} vertices but not y_{max} vertices in parity count
  - do NOT count vertices of horizontal edges



Fractional and Integer Intersections

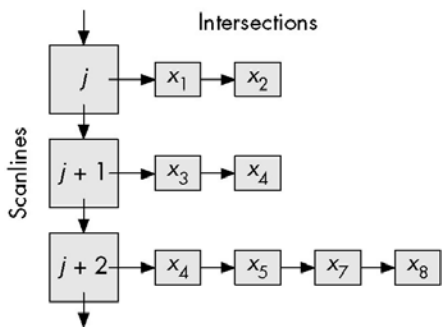
- Fractional intersections
  - if approaching intersection to the right to determine inside pixel
    - take floor if inside, ceil if outside
- Integer intersections
  - if leftmost pixel
  - make interior,
  - rightmost exterior



Using Spatial Coherence

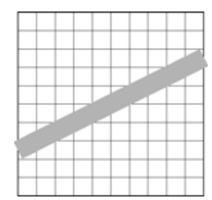
- Efficiency can be improved by using *spatial coherence*
- Edges that intersect scan-line i are likely to intersect i+1
- x_i changes predictably from scan-line i to i+1
- use an incremental algorithm that calculates the scan-line extrema from extrema of previous scan line by using
  - $x_{i+1} = x_i + 1/m$ where m is slope

Data Structure – Edge Table



Aliasing

- Ideal rasterized line should be 1 pixel wide



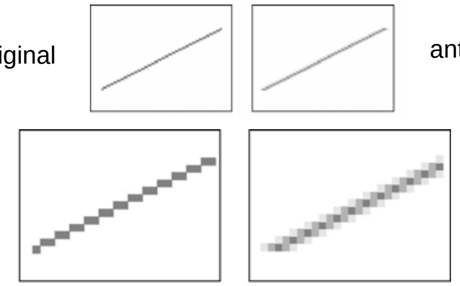
- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

- Color multiple pixels for each x depending on coverage by ideal line

original

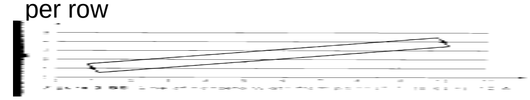
antialiased



magnified

Unweighted Area Sampling

- Assume background white - lines black
- Recognize that primitive has non-zero width
  - even thinnest line is 1 pixel thick
- Consider line as (thin) rectangle
  - covers different (square) pixels to different extent
- In most cases should not set a single pixel to black
  - Set intensity of pixel differently for each pixel covered
  - Only horizontal and vertical lines effect only 1 pixel per row



Unweighted Area Sampling

- Simplest assumption on geometry of pixels
  - nonoverlapping square tiles - grey scale display
  - line contributes to intensity proportional to area of pixel's tile covered
  - pixel (2,1) is 70% black, (2,2) is 25% black
  - makes line appear better at a distance

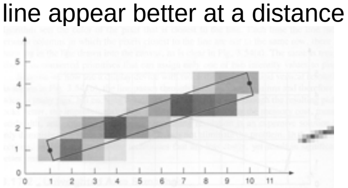


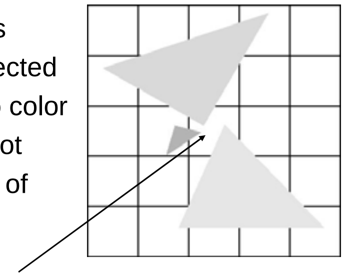
Fig. 3.56 Intensity proportional to area covered.

Properties of Unweighted Area Sampling

- 1. Intensity decreases with increasing distance from pixel to edge
- 2. Primitives do not influence pixel they do not intersect
- 3. Equal areas contribute equal intensity
  - distance from pixel center to area overlapped
  - small area in corner contributes same as equal-sized area in center

Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color