Graphics Primitives

Chapter 3 & 4
Intro. to Computer Graphics
Spring 2008, Y.G. Shin
Graphic Output and Input Pipeline

- **Scan conversion**
  - converts primitives such as lines, circles, etc. into pixel values
  - geometric description $\Rightarrow$ a finite scene area

- **Clipping**
  - the process of determining the portion of a primitive lying within a region called *clip region*
Graphic Output Pipeline

Output pipeline (rendering process)

- application model: descriptions of objects
  - application program: generates a sequence of functions to display a model
    - graphics package: clipping, scan conversion, shading, etc.
  - display H/W
Graphic Input Pipeline

- **Input pipeline**
  - user interaction (e.g., mouse click)
  - graphic package (by sampling or event-driven input functions)
  - application program
  - modify the model or the image on the screen
Graphic Output Pipeline

- displays with frame buffers and display controllers
  - common in plug-in graphics card
  - scan conversion by a graphic package and display processor
- displays with frame buffers only
  - scan conversion by a graphic package
When scan conversion and clipping?

- Clipping before scan conversion
  - for lines, rectangles, and polygons
  - clipping after scan converting each primitive (scissoring)

- Clipping after scan converting the entire collection of primitives into a temporary canvas
  - for text
Scan Converting Lines

A line from \((x_0,y_0)\) to \((x_1,y_1)\) \(\Rightarrow\) a series of pixels

[Criteria]
- Straight lines should appear straight
- Line end-points should be constrained - grids, snaps
- Uniform density and intensity
- Line algorithms should be fast
Why Study Scan Conversion Algorithms?

- Every high-end graphics card support this.
- You will never have to write these routines yourself, unless you become a graphics hardware designer.
- So why learn this stuff?
  - Maybe you *will* become a graphics hardware designer.
  - But seriously, the same basic tricks underlie lots of algorithms:
    - 3-D shaded polygons
    - Texture mapping
    - etc.
Simple Scan Converting Lines

Based on *slope-intercept algorithm* from algebra:

\[ y = mx + b \]

Simple approach:
increment x, solve for y

Floating point arithmetic required
Digital Differential Analyzer (DDA)

Idea

1. Go to starting end point
2. Increment \(x\) and \(y\) values by constants proportional to \(x\) and \(y\) such that one of them is 1.
3. Round to the closest raster position

\[
(x_i, y_i) \rightarrow (x_i + 1, \text{Round}(y_i + m))
\]

\[
(x_i, \text{Round}(y_i)) \rightarrow (x_i + 1, y_i + m)
\]
Digital Differential Analyzer (DDA)

- **Drawbacks**
  - rounding to an integer takes time
  - floating-point operations
- **Is there a simpler way?**
- **Can we use only integer arithmetic?**
  - Easier to implement in hardware.
Midpoint Line Algorithm (Bresenham's Line Algorithm)

- Assume a line from \((x_1,y_1)\) to \((x_2,y_2)\) that \(0<\text{slope}<1\) and \(x_1<x_2\).
- Use \textit{symmetry}
Midpoint Line Algorithm (Bresenham's Line Algorithm)

Suppose that we have just finished drawing a pixel \( P = (x_p, y_p) \) and we are interested in figuring out which pixel to draw next.

If \( \text{distance}(\text{NE},M) > \text{distance}(\text{E},M) \) then

select \( E = (x_p+1, y_p) \)

else

select \( \text{NE} = (x_p+1, y_p+1) \)
Midpoint Line Algorithm
(Bresenham's Line Algorithm)

- A line eq. in the implicit form:
  \[ F(x, y) = ax + by + c = 0 \]

- Using \( y = \frac{\Delta y}{\Delta x} \cdot x + B \),
  where \( a = \Delta y \), \( b = -\Delta x \), \( c = B \).
  \[ F(x, y) = \Delta y \cdot x - \Delta x \cdot y + B \cdot \Delta x = 0. \]

- Let's use an equivalent representation:
  \[ F(x, y) = 2ax + 2by + 2c = 0. \]
Midpoint Line Algorithm (Bresenham's Line Algorithm)

- Making slope assumptions, observe that $b < 0$, and this implies:
  - $F(x,y) < 0$ for points above the line
  - $F(x,y) > 0$ for points below the line
- To apply the midpoint criterion, we need only to compute $F(M) = F(x_p+1, y_p+\frac{1}{2})$ and to test its sign.
Midpoint Line Algorithm (Bresenham's Line Algorithm)

- To determine which one to pick up, we define a decision variable

\[ D = F(x_p+1, y_p+\frac{1}{2}) \]

\[ D = 2a(x_p+1) + 2b(y_p+\frac{1}{2}) + 2c \]

\[ = 2ax_p + 2by_p + (2a + b + c) \]

- If \( D > 0 \) then \( M \) is below the line, so select NE, otherwise select E.
Midpoint Line Algorithm
(Bresenham's Line Algorithm)

- How to compute \( D \) incrementally?
  - Suppose we know the current \( D \) value, and we want to determine the next \( D \).
  - If we decide on going to E next,
    - \( D_{\text{new}} = F(x_p + 2, y_p + \frac{1}{2}) \)
      - \( = 2a(x_p + 2) + 2b(y_p + \frac{1}{2}) + c \)
      - \( = D + 2a = D + 2\Delta y \)
  - If we decide on going to NE next,
    - \( D_{\text{new}} = F(x_p + 2, y_p + 1 + \frac{1}{2}) \)
      - \( = 2a(x_p + 2) + 2b(y_p + 1 + \frac{1}{2}) + c \)
      - \( = D + 2(a + b) = D + 2(\Delta y - \Delta x) \).
Midpoint Line Algorithm
(Bresenham's Line Algorithm)

- Since we start at \((x_0, y_0)\), the initial value of \(D\) can be calculated by
  \[ D_{\text{init}} = F(x_0 + 1, y_0 + \frac{1}{2}) \]
  \[ = (2ax_0 + 2by_0 + c) + (2a + b) \]
  \[ = 0 + 2a + b \]
  \[ = 2\Delta y - \Delta x \]

- Advantages
  - Only need add integers and multiply by 2
    (which can be done by shift operations)
  - Incremental algorithm
Example code

```c
void MidpointLine(int x0, int y0,
                   int x1, int y1, int value) {  
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;
    int incrE = 2 * dy;
    int incrNE = 2 * (dy - dx);
    int x = x0;
    int y = y0;

    writePixel(x, y, value);

    while (x < x1) {
        if (d <= 0) { // East Case
            d = d + incrE;
        } else { // Northeast Case
            d = d + incrNE;
            y++;
        }
        x++;
        writePixel(x, y, value);
    } /* while */
} /* MidpointLine */
```
Midpoint Line Algorithm - Example

Line end points:

\[(x_0, y_0) = (5, 8);\]
\[(x_1, y_1) = (9, 11)\]

- \(\Delta x = 4; \Delta y = 3\)
- \(D_{\text{init}} = 2\Delta y - \Delta x = 2 > 0\)
  - select NE
- \(D_{\text{new}} = D + 2(\Delta y - \Delta x) = 0\)
  - Select E
- \(D_{\text{new}} = D + 2\Delta y = 0 + 6 = 6\)
  - Select NE
Scan Converting Lines (issues)

- **Endpoint order**
  - $S_{01}$ is a set of pixels that lie on the line from $P_0$ to $P_1$
  - $S_{10}$ is a set of pixels that lie on the line from $P_1$ to $P_0$
  - $\Rightarrow S_{01}$ should be the same as $S_{10}$

- **Varying intensity of a line as a function of slope**
  - For the diagonal line, it is longer than the horizontal line but has the same number of pixels as the latter
  - $\Rightarrow$ needs antialiasing

- **Outline primitives composed of lines**
  - Care must be taken to draw shared vertices of polylines only once
Scan Converting Lines (issues)

- Starting at the edge of a clip rectangle
  - Starting point is not the intersection point of the line with clipping edge
    ⇒ Clipped line may have a different slope
Scan Converting Circles

- Eight-way symmetry

We only consider 45° of a circle
Midpoint Circle Algorithm

- Suppose that we have just finished drawing a pixel \((x_p, y_p)\) and we are interested in figuring out which pixel to draw next.
Midpoint Circle Algorithm

- \( F(x, y) = x^2 + y^2 - R^2 \)
  = 0 on the circle
  > 0 outside the circle
  < 0 inside the circle
- If \( F(\text{midpoint between } E \text{ and } SE) > 0 \) then
  select \( SE = (x_p+1, y_p-1) \)
else
  select \( E = (x_p+1, y_p); \)
Midpoint Circle Algorithm

- Decision variable $d_{old} = F(x_p+1, y_p^{-1/2})$
  
  $= (x_p+1)^2 + (y_p^{-1/2})^2 - R^2$

- If $d_{old} < 0$, select E.
  $d_{new} = F(x_p+2, y_p^{-1/2}) = d_{old} + (2x_p + 3)$

- If $d_{old} \geq 0$, select SE.
  $d_{new} = F(x_p+2, y_p^{-1/2}-1) = d_{old} + (2x_p - 2y_p + 5)$

- We have to calculate $d_{new}$ based on the point of evaluation $P=(x_p, y_p)$, but this is not expensive computationally.
Midpoint Circle Algorithm

- Since we start at (0,R), the initial value of d can be calculated by
  \[ d_{\text{init}} = F(1, R - \frac{1}{2}) = \frac{5}{4} - R. \]

- By substituting \( d - \frac{1}{4} \) by \( h \), we can get the integer midpoint circle scan-conversion algorithm.
Scan Converting Ellipses

- $F(x,y) = b^2x^2 + a^2y^2 - a^2b^2$
- Divide the quadrant into two regions; the boundary of two regions is the point at which the curve has a slope of -1.
- And then apply any midpoint algorithm.
Area Filling

How to generate a solid color/patterned polygon area

- Which pixels?
- What value?

Scan line approach
Area Filling (Scan line Approach)

- Take advantage of
  - span coherence - all pixels on a span are set to the same value
  - scan-line coherence - consecutive scan lines are identical
  - edge coherence - edges intersected by scan line \( i \) are also intersected by scan line \( i+1 \)
Area Filling (Scan line Approach)

- For each scan line
  1. Find intersections (the extrema of spans)
     - Use Bresenham's line-scan algorithm
     - Note that in a line drawing algorithm there is no difference between interior and exterior pixels
     - BUT it is better to draw interior only
  2. Sort intersections (increasing x order)
  3. Fill in between pair of intersections
Find intersections

- \( x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \)
  - example (left edge)
    - \( m = \frac{5}{2} \)
    - \( x_{\text{min}} = 3 \)
    - the sequence of x values
      - 3, 3+2/5, 3+4/5, 3+5/6=4+1/5

<table>
<thead>
<tr>
<th>y</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
<td>3+2/5</td>
<td>3+4/5</td>
<td>4+1/5</td>
</tr>
<tr>
<td>pixel</td>
<td>(3,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(5,3)</td>
</tr>
</tbody>
</table>
How to decide interior

Parity Fill Approach

- For each pixel, determine if it is inside or outside of a given polygon.

- **Approach**
  - from the point being tested cast a ray in an arbitrary direction
  - if the number of crossings is odd then the point is inside
  - if the number of crossings is even then the point is outside
**Parity Fill Approach**

- **Edge Crossing Rules**
  - an upward edge includes its starting endpoint, and excludes its final endpoint;
  - a downward edge excludes its starting endpoint, and includes its final endpoint;
  - horizontal edges are excluded;
  - the edge-ray intersection point must be strictly right of the point P.
Parity Fill Approach

- Very fragile algorithm
  - Ray crosses a vertex
  - Ray is coincident with an edge
- Commonly used in ECAD
- Suitable for H/W
Winding Number Approach

A winding number is an attribute of a point with respect to a polygon that tells us how many times the polygon encloses (or wraps around) the point. It is an integer, greater than or equal to 0. Regions of winding number 0 (unenclosed) are obviously outside the polygon, and regions of winding number 1 (simply enclosed) are obviously inside the polygon.

- Initially 0
  - +1: edge crossing the line from right to left
  - -1: left to right
- Use the sign of the cross product of the line and edge vectors
- The line does not cross any vertex
How to decide interior

Vertices are numbered: 0 1 2 3 4 5 6 7 8 9
Span Rules

- intersection at integer coordinate
  - leftmost: interior
  - rightmost: exterior
- shared vertices
  - count parity at \( y_{\text{min}} \) vertices only
  - shorten edges
- horizontal edges
  - do not count vertices

A standard convention is to say that a point on a left or bottom edge is inside, and a point on a right or top edge is outside.
Span Rules
Area Filling

- Use edge coherence and the scan-line algorithm
  
  - ET
    - Contains all the non-horizontal edges.
    - Edges are sorted by their smaller y coordinates.
  
  - AET
    - Contains edges which intersect the current scan line.
    - Edges are sorted on their x intersection values.
Area Filling (Scan line method)

Scan line 9

Scan line 10
Area Filling (Filling Methods)

- **Pixel Adjacency**

- **Boundary-Fill Algorithm**
  - starting a point inside the figure and painting the interior in a specified color or intensity.
Boundary Filling

procedure boundary_fill4(
    x,y : integer  starting point in region
    boundaryValue : value that defines boundary
    newValue : color); replacement value
var
    c : color
begin
    c := readPixel(x,y);
    if c <> boundaryValue and
    c <> newValue then
    begin
        writePixel(x,y,newValue);
        boundary_fill4(x,y-1,boundaryValue,newValue);
        boundary_fill4(x,y+1,boundaryValue,newValue);
        boundary_fill4(x-1,y,boundaryValue,newValue);
        boundary_fill4(x+1,y,boundaryValue,newValue);
    end
end;
Boundary Filling

- There is the following problem with `boundary_fill4`:

- Involve heavy duty recursion which may consume memory and time

  ![Diagram of boundary fill problem]

- Solve with 8-connected
Boundary Filling

Efficiency in space!

- finish the scan line containing the starting position
- process all lines below the start line
- process all lines above the start line
Flood Filling

: Start a point inside the figure, replace a specified interior color only.

procedure flood_fill4(
  x,y : integer  starting point in region
  oldValue  value that defines interior
  newValue : color); replacement value
begin
  if readPixel(x,y) = oldValue then
    begin
      writePixel(x,y,newValue);
      flood_fill4(x,y-1,oldValue,newValue);
      flood_fill4(x,y+1,oldValue,newValue);
      flood_fill4(x-1,y,oldValue,newValue);
      flood_fill4(x+1,y,oldValue,newValue);
    end
  end;
end;
Problems of Filling Algorithm

- What happens if a vertex is shared by more than one polygon, e.g. three triangles?
- What happens if the polygon intersects itself?
- What happens for a “sliver”?

Solutions?
- Redefine what it means to be inside of a triangle
- Different routines for nasty little triangles
Patterned Lines

- Patterned line from $P$ to $Q$ is not the same as patterned line from $Q$ to $P$.

- Patterns can be geometric or cosmetic
  - Cosmetic: Texture applied after transformations
  - Geometric: Pattern subject to transformations
Character, Symbols

- **Stroke tables**: a set of vectors which are scan converted as lines

  (Example) outline font

  move 0 0
  draw 1 1
  move 0 1
  draw 1 0
  move 1 1
  ....

- **Bitmaps**: array of 0's and 1's, scan converted as points
## Comparison of Methods

<table>
<thead>
<tr>
<th>Stroke table</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>easy to rotate</td>
<td>rotate by multiples of 90°</td>
</tr>
<tr>
<td>easy to scale</td>
<td>scale by powers of 2</td>
</tr>
<tr>
<td>variable length storage</td>
<td>fixed length storage</td>
</tr>
<tr>
<td>Scan convert lines</td>
<td>scan convert points</td>
</tr>
<tr>
<td>fill if polygons</td>
<td>draw as filled or outline</td>
</tr>
<tr>
<td>may be anti-aliased or smoothed via</td>
<td>may be pre-anti-aliased</td>
</tr>
<tr>
<td>curve fitting</td>
<td></td>
</tr>
<tr>
<td>best for linear designs</td>
<td>arbitrary patterns with many colors</td>
</tr>
</tbody>
</table>
Line Attributes

- **Butt cap**
- **Round cap**
- **Projecting square cap**
- **Miter join**
- **Round Join**
- **Bevel join**
Aliasing in CG

Which is the better?
Digital technology can only approximate analog signals through a process known as sampling.

Aliasing: the distortion of information due to low-frequency sampling (undersampling).

Choosing an appropriate sampling rate depends on data size restraints, need for accuracy, the cost per sample...

Errors caused by aliasing are called artifacts. Common aliasing artifacts in computer graphics include jagged profiles, disappearing or improperly rendered fine detail, and disintegrating textures.