Scan-conversion and Clipping

• Review
  – Graphics commands specify straight lines or other geometric primitives that are scan-converted into a set of discrete intensity pixels in the framebuffer (back buffer)
  – Color points in the frame buffer are sent to the corresponding pixels in the display device by a video controller

• The purpose of this section is to understand how the drawing primitives and clipping are implemented

• A graphics library function may be implemented quite differently
SCAN CONVERTING LINES

• For now, we consider only 1-pixel width lines.

• The screen/pixel may be represented in different forms
The Basic Incremental Algorithm

\[ \text{slope } m = \frac{\Delta y}{\Delta X}; \quad y_i = mx_i + B; \quad y_{i+1} = y_i + m \]

Intensify the pixel at \((x_i, \text{Round}(y_i))\).

**Example: J1_2_Line**

```c
void line(int x0, int y0, int xn, int yn)
{
    // for \(-1 \leq m \leq 1\)
    int x; float m, y;

    m = (float) (yn-y0)/(xn-x0);
    x=x0; y=y0;

    while (x<nxn+1) {
        // write a pixel into the framebuffer
        glBegin(GL_POINTS);
            glVertex2i (x, (int) (y+0.5));
        glEnd();
        x++; y+=m; /* next pixel's position */
    }
}
```

Copyright @ 2011 by Jim X. Chen:

jchen@cs.gmu.edu
The Midpoint Line Algorithm

• The shortcoming of the previous algorithm is that rounding $y$ to an integer takes time.

$y = \frac{dy}{dx}x + B$, i.e. $F(x, y) = dyx - dx^2 + Bdx = 0$; that is: $F(x, y) = ax + by + c = 0$ when $(x, y)$ on the line, where $a = dy > 0$, $b = -dx < 0$, and $c = Bdx > 0$.

• Let $M1 = (x, y')$, $Q = (x, y)$, $F(x, y') > 0$ when $(x, y')$ below the line, and $F(x, y') < 0$ when $(x, y')$ above the line.
If \( F(x_p + 1, y_p + 1/2) > 0 \), \( Q \) above \( M \), that is, the line is above middle point), we choose \( NE \); else we choose \( E \). Therefore we have a decision variable:

\[
d_{\text{old}} = F(x_p + 1, y_p + 1/2) = F(x_p, y_p) + a + b/2 = a + b/2
\]

If \( d_{\text{old}} \leq 0 \), \( E \) is chosen: \( d_{\text{new}} = F(x_p + 2, y_p + 1/2) = d_{\text{old}} + a \)
If \( d_{\text{old}} > 0 \), \( NE \) is chosen: \( d_{\text{new}} = F(x_p + 2, y_p + 3/2) = d_{\text{old}} + a + b \)

“\( a \)” and “\( b \)” are integers. Multiply the decision equations by 2 for integer operation:

\[
d_{\text{old}} = 2F(x_p + 1, y_p + 1/2) = 2F(x_p, y_p) + 2a + b = 2a + b
\]
If \( d_{\text{old}} \leq 0 \), \( E \) is chosen: \( d_{\text{new}} = 2F(x_p + 2, y_p + 1/2) = d_{\text{old}} + 2a \)
If \( d_{\text{old}} > 0 \), \( NE \) is chosen: \( d_{\text{new}} = 2F(x_p + 2, y_p + 3/2) = d_{\text{old}} + 2a + 2b \)

Let’s call \( d_E = 2a = 2dy \), \( d_{NE} = 2(a + b) = 2(dy - dx) \), then
\[
\begin{align*}
\text{If } d_{\text{old}} \leq 0: & \quad d_{\text{new}} = d_{\text{old}} + d_E \\
\text{If } d_{\text{old}} > 0: & \quad d_{\text{new}} = d_{\text{old}} + d_{NE}
\end{align*}
\]
We only need to decide if \(d_{\text{old}} > 0\) to choose NE or E, and update \(d_{\text{new}}\) accordingly with an integer addition.

**Example: J1 3 Line**

```c
void line(int x0, int y0, int xn, int yn)
{
    // Bresenham's midpoint line algorithm
    int dx, dy, dE, dNE, d, x, y, flag = 0;

    ...; // taking care of all slopes

    x=x0; y=y0; d=2*dy-dx;
    dE=2*dy; dNE=2*(dy-dx);

    while (x<xn+1) {
        writepixel(x,y,flag);
        x++; // next pixel
        if (d<=0) d+=dE;
        else { y++; d+=dNE; };
    }
}
```

*Bresenham Line Alg.*
SCAN CONVERTING CIRCLES

\[ x^2 + y^2 = R^2, \text{ therefore } y = (R^2 - x^2)^{1/2}. \] We can increase \( x \) from 0 to \( R/2 \) to draw the corresponding points at \((x, y), (y, x), (y, -x), (x, -y), (-x, -y), (-y, -x), (-y, x), (-x, y)\).

```c
void circlePixel(float x, float y)
{
    WritePixel(x, y); WritePixel(y, x);
    WritePixel(-x, y); WritePixel(-y, x);
    WritePixel(x, -y); WritePixel(y, -x);
    WritePixel(-x, -y); WritePixel(-y, -x);
}
```

- Time consuming
- Space between pixels is not uniform

Copyright @ 2006 by Jim X. Chen:
jchen@cs.gmu.edu
Polar Coordinates

- \( x = x_c + r \cos(\alpha); \ y = y_c + r \sin(\alpha); \)

```java
double cx = WIDTH/2, cy = HEIGHT/2, //center of the circle
            r = WIDTH/3, // radius of the circle
delta = 1/r; // angle for one unit apart: 2PI/2PI*r

public void drawCircle(double x0, double y0, double r) {
    double th = 0;

    while (th <= Math.PI/4) {
        th = th + delta;
        double x = r*Math.cos(th); double y = r*Math.sin(th);
        drawPoint(x+x0, y+y0); drawPoint(x+x0, -y+y0);
        drawPoint(-x+x0, y+y0); drawPoint(-x+x0, -y+y0);
        drawPoint(y+x0, x+y0); drawPoint(y+x0, -x+y0);
        drawPoint(-y+x0, x+y0); drawPoint(-y+x0, -x+y0);
    }
}
```

Example: J1_3_CircleLine

- We often draw line segments instead of real curve

```java
void circle(double cx, double cy, double r) {
    double xn, yn, theta = 0, delta = 0.1; // the delta angle for a line segment
    double x0 = r*Math.cos(theta)+cx;
    double y0 = r*Math.sin(theta)+cy;

    while (theta<2*Math.PI) {
        theta = theta + delta;
        xn = r*Math.cos(theta)+cx;
        yn = r*Math.sin(theta)+cy;
        bresenhamLine((int)x0, (int)y0, (int)xn, (int)yn);
        x0 = xn;
        y0 = yn;
    }
}
```
Midpoint Circle Algorithm

\[ P(x_p, y_p) \]

Copyright @ 2006 by Jim X. Chen:
jchen@cs.gmu.edu
Scan-converting Other Curves

- Conic Sections (2nd degree)
  \[ Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \]
  \[ B^2 - 4AC < 0 \text{ ellipse or circle}; \quad = 0 \text{ parabola}; \quad >0 \text{ hyperbola} \]

- Parabolic path: (parametric equation)
  \[ x = x_0 + v_{x_0} t \]
  \[ y = y_0 + v_{y_0} t - gt^2/2 \]

- Hyperbolic pathes:
  \[ (x/r_x)^2 - (y/r_y)^2 = 1 \]
HW1_2: due before next class

• Draw a pentagon in a circle that rotates. You cannot use glBegin(GL_LINES). You should use glBegin(GL_POINTS) instead. You should implement your own line function that can be used to draw the pentagon. You may inherit my provided samples.
3D Spline: two adjacent curve sections have the same coordinate position at the boundary and the slopes at the boundary are equal.

\[ x = a_x + b_x t + c_x t^2 + d_x t^3 \]
\[ y = a_y + b_y t + c_y t^2 + d_y t^3 \]
\[ z = a_z + b_z t + c_z t^2 + d_z t^3 \]

where \( 0 \leq t \leq 1 \)

Polynomials: useful in the design of object shapes, the specification of animation paths, and the graphing of data trends in a discrete set of data points.
**FILLING RECTANGLES**

for (y=ymin; y<ymax; y++)
for (x=xmin; x<xmax; x++)
    WritePixel(x,y,value);

**SCAN-CONVERTING POLYGONS**

- Find the intersections of the scan-line with all edges of the polygon.
- Sort the intersections by increasing the x coordinate.
- Fill in all pixels between pairs of intersections that lie interior to the polygon, using the odd-parity rule to determine that a point is inside a region.
Scan-line algorithm for filling polygons

**ET** (edge table) -- all edges sorted by the smaller y coords;

**AET** (active-edge table) corresponds to the current scan-line.
Set $y$ to the smallest $y$ that has an entry in the ET. Initialize the AET to be empty. Repeat until the AET and ET are empty:

1. Move from ET bucket $y$ to AET those edges whose $y_{min} = y$.
2. Sort the AET on $x$.
3. Fill-in desired pixel values on scan-line $y$ by using pairs of $x$ coord. From the AET.
4. Increment $y$ by 1.
5. Remove from the AET those entries that $y = y_{max}$.
6. For each non-vertical edge remaining in the AET, update $x$ for the new $y$: $x = x + 1/m$.

**Example: J1 3 Triangle**
OpenGL Implementation

• A *convex* polygon means that all the angles inside the polygon formed by the edges are smaller than 180 degrees.

• If a polygon is not convex, it is *concave*. Convex polygons can be scan-converted faster than concave polygons.

• OpenGL draws a *convex* polygon with the following commands:

```c
    glBegin(GL_POLYGON);
        // a list of vertices
    glEnd();
```
1. Implement functions
myTriangle(double V[3][2]), myTriangle(double X[3], double Y[3]) and myTriangle(double x1, double y1, double x2, double y2, double x3, double y3) for J1_3_Triangle.java. You cannot use OpenGL functions such as glBegin(GL_TRIANGLES) to draw the triangle directly.

2. Generate random vertices and draw triangles.
CHARACTERS

- Characters are polygons. However, they are used very often.
- Type-face, font: the overall design style for a set of characters, Courier, Helvetica, Times, etc.
- Can represent the character shapes by rectangular grid patterns (bitmap font)
- Or describe character shapes by lines and curves, as in Postscript (outline font)
- Bitmap fonts require more space, because each variation (size or format) must be stored.
-Outline fonts require more time to produce, because they must be scan converted

-Small bitmaps do not scale well. Different bitmaps (sizes and typefaces) are defined

-accessing fonts is platform dependent. GLUT provides a simple platform independent subset of font functions.

-GLX provides font functions and interfaces between OpenGL and the X window system. WGL is the equivalent of GLX on the Microsoft Windows platform.

**Example: J1_3_xFont**
Example: J1_3_xFont

import javax.media.opengl.*;
import com.sun.opengl.util.*;

public class J1_3_xFont extends J1_3_Triangle {
    static GLUT glut = new GLUT();

    public void display(GLAutoDrawable drawable) {
        // generate a random line;

        // bitmap fonts
        gl.glWindowPos3f(x0, y0, 0); // start poistion glRasterpos or glWindowPos
        glut.glutBitmapCharacter(GLUT.BITMAP_HELVETICA_18, 'S');
        glut.glutBitmapString(GLUT.BITMAP_HELVETICA_18, "tart");

        // stroke fonts
        gl.glPushMatrix();
        gl.glTranslatef(xn, yn, 0); // end of line position
        gl.glScalef(0.2f, 0.2f, 0.2f); // size
        glut.glutStrokeCharacter(GLUT.STROKE_ROMAN, 'E');
        glut.glutStrokeString(GLUT.STROKE_ROMAN, "nd");
        gl.glPopMatrix();
    }
}
The Cohen-Sutherland Line-Clipping Algorithm

Iterate the following for l, r, b, t (left, right, bottom, top)

If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected.
Implementation Example: Line Scan-conversion with Clipping

// 2D clipping against a fixed window
drawClippedLine (x0, y0, x1, y1) {
    clip(x0, y0, x1, y1, x00, y00, x11, y11);
    // (l, b, r, t)
    // (x00, y00) and (x11, y11) are the two
    // endpoints after clipping
    line(x00, y00, x11, y11);
}
public void drawPoint(double x, double y) {

    // clip against the window
    if (x < lLeft[0] || x > uRight[0]) {
        return;
    }

    if (y < lLeft[1] || y > uRight[1]) {
        return;
    }

    super.drawPoint(x, y);
}
Line Clipping against a 3D Plane

// how about a line against a plane?

GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};

glClipPlane (GL_CLIP_PLANE0, eqn);
glEnable (GL_CLIP_PLANE0);

• A plane equation
  \[ Ax + By + Cz + D = 0 \]

• A 3D line equation
  \[ x = x_0 + t(x_1 - x_0); \ y = y_0 + t(y_1 - y_0); \ z = z_0 + t(z_1 - z_0); \]

The intersection of the line with the plane is at a specific \( t \)
The Sutherland Hodgman Polygon-Clipping Algorithm

The alg moves around the polygon from \( v_1 \) to \( v_n \) and the back to \( v_1 \) against the edges of the clipping boundary.

At each step, 0, 1, or 2 vertices are added to the output list of vertices that defines the clipped polygon edge. Four possible cases must be analyzed:

- starting & ending vertices inside (add ending vertex \( p \) to the output list)
- starting inside, ending outside (add intersection vertex \( i \))
- starting outside, ending inside (add \( i, p \))
- starting outside, ending outside (continue)
Polygon Clipping against a Plane

• A plane equation
  \[ Ax + By + Cz + D = 0 \]
• A polygon is a list of vertices
• A polygon is a list of line segments (pairs of vertices)

essentially line clipping against a plane

Example: J1 3 windowClipping
HW2: 2013 Fall Class

1. Draw a line that bounces between the circle and rectangle. The line can be two moving points, so the length changes; (40%)

2. Draw a triangle that bounces. The triangle can be three moving points, so the length changes; (35%)

3. Using mouse to move the circle around. It is used to reflect objects (points and endpoints of lines & triangles); (20%)

4. Using character strings to report (live) point counts or some other information you like to inform users. (5%)
HW2: 2012 Fall Class

1. Draw a line that bounces in a circle. The line can be two moving points, so the length changes; (40%)

2. Draw a triangle that bounces in a circle. The triangle can be three moving points, so the length changes; (35%)

3. Draw a horizontal bar that can be moved using mouse horizontally, and keyboard vertically. It is used to reflect objects (points and endpoints of lines & triangles) so they don’t fall below the bar; (20%)

4. Using character strings to report (live) point counts or some other information you like to inform users. (5%)