Computer Graphics 1

Line and Circle Drawing

by

Sean P. Strout (sps@cs.rit.edu)

Lines

Drawing Lines

- Easy in Open GL:
  
  ```
  glBegin( GL_LINES );
  glVertex2i( 10, 20 );
  glVertex2i( 50, 100 );
  glEnd();
  ```
  
  - Useful to know how it works under the hood
  - What we want in a line drawing algorithm:
    - Fast
    - Constant/uniform width & brightness regardless of slope or location
    - As straight as possible
    - As close to “ideal” as possible
    - Centered on the “ideal” line
    - Equal-length lines have comparable lengths

Scan Converting Lines

- Basic principle: compute coordinates of pixels that lie on or near an ideal, infinitely thin straight line imposed on a 2D raster grid
- For small slopes (-1<m<1), will illuminate one pixel per column; for large slopes, one pixel per row

Basic Incremental Approach

- Simplest strategy:
  - Compute slope m as Δy/Δx
  - Start at the leftmost point
  - For each x, compute yᵢ = mxᵢ + B
  - Intensify the pixel at (xᵢ, floor(yᵢ+0.5))
- This is the basic incremental algorithm
  - Also called the digital differential analyzer (DDA)
- Problem: each iteration requires a multiplication, addition, and invocation of floor()
- Can eliminate the multiplication by noting:
  
  \[
  \begin{align*}
  y_{i+1} &= mx_i + B = m(x_i + Δx) + B = mx_i + B + mΔx \\
  &= y_i + mΔx
  \end{align*}
  \]
- And, if Δx = 1, yᵢ₊₁ = yᵢ + μ

Basic Incremental Algorithm

- Implementation, for a line from p₁ to p₂,
  
  ```
  float m = (p₂.y - p₁.y) / (p₂.x - p₁.x);
  int y = p₁.y;
  for( int x = p₁.x; x <= p₂.x; ++x ) {
    drawDot( x, floor(y+0.5) );
    y += m;
  }
  ```
  
  - Problem: doing lots of floating point math
    - Slower than integer math
    - Also suffers from precision problems (roundoff error)
Midpoint Line Algorithm

- Jack Bresenham developed a better algorithm that uses only integer arithmetic
  - Called Bresenham's Algorithm
  - Designed for controlling a digital plotter
- We'll study one version of this, called the midpoint line algorithm
- For simplicity, we'll assume that \( p_1 \) is to the left of \( p_2 \), and that the slope of the line is between 0 and 1
  - If \( p_2 \) is to the right of \( p_1 \), we just swap the names of the points
  - We can deal with other slopes by reflecting around the axes

Basic Concepts

- Consider this line segment
- We have chosen the pixel at \((x_p, y_p)\) and must choose the next pixel
- Line intersects the next \( x_i \) at point \( Q \)
- Bresenham's algorithm computes the distance from each pixel to \( Q \), and selects the closest one
- Midpoint determines which side of the line \( m \) is on, and selects the appropriate pixel
- In this case, we select \( U \)

Line Equations

- We can easily verify that \( F(x,y) \) is zero for \((x,y)\) on the line, negative for \((x,y)\) above the line, and positive for \((x,y)\) below the line
- Rather than recompute \( F(x,y) \) each time, we'll figure out how to compute it incrementally
- For the midpoint \( m \), we know that \( F(m) = F(x_p+1, y_p+1/2) \)
- We compute this as \( d_{\text{old}} = a(x_p+1) + b(y_p+1/2) + c \)
- If this is > 0, we'll choose \( U \); if < 0, we'll choose \( L \)

The Next Level

- Now that we know which way to move from the current point, let's look at the next choice
- If we choose \( L \), the new midpoint is \( m' \)
- If we choose \( U \), the new midpoint is \( m'' \)

Midpoint Deltas

- If we choose \( L \), the new \( d \) is \( d_{\text{new}} = F(x_p+2, y_p+1/2) = a(x_p+2) + b(y_p+1/2) + c \)
- We can calculate \( d_{\text{new}} - d_{\text{old}} \) to get \( \Delta L = a = dy \)
- If we choose \( U \), the new \( d \) is \( d_{\text{new}} = F(x_p+2, y_p+3/2) = a(x_p+2) + b(y_p+3/2) + c \)
- And we calculate \( d_{\text{new}} - d_{\text{old}} \) to get \( \Delta U = a+b = dy - dx \)
- We set \( d \) initially to \( F(m) \), and add the appropriate delta value each time, depending on the pixel we selected
- For the initial midpoint, \( F(m) \) is \( F(m) = F(x_0, y_0 + 1/2) = ax_0 + by_0 + c + a + b/2 \)
  - So, the initial delta is \( a + b/2 \), which is \( dy - dx/2 \)
### Simplification

- **Problem**: This involves division!
- **Solution**: Go back to $F$, and multiply everything by 2:
  - $\Delta L = 2dy$
  - $\Delta U = 2(dy - dx)$
  - Initial delta = $2dy - dx$

- **Pseudo-code for each step**:
  - Set the pixel at $(x,y)$
  - Increment $x$ by one
  - If choosing $L$, increment $d$ by adding $2dy$
  - If choosing $U$, increment $y$, increment $d$ by adding $2dy - dx$

### Implementation

```c
void midpointLine( int x0, int y0, int x1, int y1) {
    int dl, du, x, y, d;
    dl = 2 * (y1 - y0);
    du = 2 * ((y1 - y0) - (x1 - x0));
    d = dl - (x1 - x0);
    for( x = x0, y = y0; x <= x1; ++x) {
        drawPixel( x, y );
        if( d <= 0 ) {
            d += dl;
        } else {
            d += du;
            ++y;
        }
    }
}
```

### Extending the Midpoint Line Algorithm

- We did our development for a subset of all possible lines.
- It’s worth testing for slope 0 or slope $\infty$ - we can get significant performance improvement.
- For other slopes, merely choose $m, m', m''$, and directions differently.
- For large slopes, loop on $y$ rather than $x$.

### Drawing Circles

- We describe a circle with $x^2 + y^2 = R^2$.
- Solving for $y$, we get: $y = \pm \sqrt{R^2 - x^2}$.
- However, plotting $\pm y$ for $0 \leq x \leq 17$, we get:

### Eight Way Symmetry

- We get good results for the first part of the quadrant.
- By computing for $0 \leq x \leq \frac{R}{\sqrt{2}}$, we get:
- We can mirror this across the $x$ and $y$ axes, and across both 45° lines.
- Clockwise from the second octant, we plot:
  - First quadrant: $(x,y)$
  - Second quadrant: $(y,x)$
  - Third quadrant: $(-x,-y)$
  - Fourth quadrant: $(-y,x)$
Midpoint Circle Algorithm

- Bresenham developed an incremental algorithm for circle generation
  - Originally designed for use with pen plotters
  - Generates all points around entire circumference
- We'll look at how this can be combined with our 8-way symmetry approach to do integer incremental circle drawing
- As in the previous example, we'll look only at a 45° segment, from $x = 0$ to $x = y = R / \sqrt{2}$

### Basic Concepts

- As before, we have chosen the pixel at $(x_p, y_p)$ and must choose the next pixel
- Line intersects the next $x_i$ at point $Q$
- Midpoint determines which side of the line $m$ is on, and selects the appropriate pixel
- In this case, we select $U$

### Circle Equations

- Recall the basic circle equation: $F(x, y) = x^2 + y^2 - R^2$
- $F(x, y)$ is zero for $(x, y)$ on the circle, negative for $(x, y)$ inside the circle, and positive for $(x, y)$ outside the circle
- As before, we'll figure out how to compute $F(x, y)$ incrementally
- For the midpoint $m$, we compute $d_{old} = F(x_p + 1, y_p - 1/2) = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2$
  - If this is $< 0$, we'll choose $U$; if $> 0$, we'll choose $L$

### The Next Level

- Now that we know which way to move from the current point, let's look at the next choice
- If we choose $U$, the new midpoint is $m'$
- If we choose $L$, the new midpoint is $m''$

### Midpoint Deltas

- If we choose $U$, the new $d$ is $d_{new} = F(x_p + 2, y_p - 1/2) = (x_p + 2)^2 + (y_p - 1/2)^2 - R^2$
- We can calculate $d_{new} - d_{old}$ to get $\Delta U = 2x_p + 3$
- If we choose $U$, the new $d$ is $d_{new} = F(x_p + 2, y_p - 1/2) = (x_p + 2)^2 + (y_p - 1/2)^2 - R^2$
- And we calculate $d_{new} - d_{old}$ to get $\Delta L = 2x_p - 2y_p + 5$
- Note that the deltas this time vary according to the position of the previous pixel chosen at $(x_p, y_p)$
  - We call this point $P$, the **point of evaluation**

### First Implementation

- We know that the starting pixel is at $(0, R)$
- The first midpoint is at $(1, R - 1/2)$, so $F(1, 1) = 0 + (R - 1/2)^2 - R^2 = 1 + (R^2 - R + 1/4) - R^2 = 5/4 - R$
- Implementation:

```c
void midpointCircle( int radius) {
    int x, y;
    float d;
    x = 0;
    y = radius;
    d = 5.0 / 4 - radius;
    drawAllEightPoints( x, y );
    while( y > x ) {
        if( d < 0 ) {
            d += x * 2.0 + 3;
            ++x;
        } else {
            d += (x – y) * 2.0 + 5;
            ++x;
            --y;
        }
        drawAllEightPoints( x, y );
    }
}
```
Simplification

- Problem: this involves floating point arithmetic
- Solution: transform the program
- Define a new decision variable, \( h \), as \( h = d - \frac{1}{4} \)
- Substitute \( h + \frac{1}{4} \) for \( d \) in the code
- Initialization is now \( h = 1 - R \), and the comparison \( d < 0 \)
  becomes \( h < 0 \)
- However, \( h \) starts out with an integer value, and is incremented by integer values, so the comparison can be simplified to \( h < 0 \)

Integer Implementation

```c
void midpointCircle( int radius) {
    int x, y, h;
    x = 0;
    y = radius;
    h = 1 – radius;
    drawAllEightPoints( x, y );
    while( y > x ) {
        if( h < 0 ) {
            h += x * 2 + 3;
            ++x;
        } else {
            h += (x – y) * 2 + 5;
            ++x;
            --y;
        }
        drawAllEightPoints( x, y );
    }
}
```

Further Simplification

- We can further improve the algorithm's performance by using second-order differences
- We noted that the deltas are linear equations, and computed them directly
- We can apply the same technique to allow us to compute the deltas incrementally
- If we choose the \( U \) pixel, \( \Delta U = 2x_p + 3 \)
  - Call this \( \Delta U_{old} = 2x_p + 3 \)
- Next, calculate \( \Delta U_{new} \) at \((x_p+1, y_p)\) as \( 2(x_p+1) + 3 \)
- From this, \( \Delta U_{new} - \Delta U_{old} = 2 \)
- Similarly, we calculate \( \Delta L_{new} - \Delta L_{old} = 2 \)
- If we choose \( L \) instead of \( U \), the deltas are 2 and 4, respectively

Second-Order Implementation

```c
void midpointCircle( int radius) {
    int x, y, h, du, dl;
    x = 0;
    y = radius;
    h = 1 – radius;
    du = 3;  dl = 5 – radius * 2;  // initial deltas
    drawAllEightPoints( x, y );
    while( y > x ) {
        if( h < 0 ) {
            h += du; du += 2; dl += 2;
            ++x;
        } else {
            h += dl; du += 2; dl += 4;
            ++x;
            --y;
        }
        drawAllEightPoints( x, y );
    }
}
```

Revision History

- v0.00, 9/21/2003 11:04 AM, sps
  Initial revision.