Lecture 6 of 41

Scan Conversion 1 of 2: Midpoint Algorithm for Lines and Ellipses

William H. Hsu
Department of Computing and Information Sciences, KSU

Readings:
Today: Sections 2.5.1, 3.1, Eberly 2 – see http://bit.ly/ieUq45
This week: Brown CS123 slides on Scan Conversion – http://bit.ly/hfbF0D

Lecture Outline
- Reading for Last Class: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2 *
- Reading for Today: §2.5.1, 3.1 Eberly 2 *
- Reading for Next Class: §2.3.5, 2.4, 3.1.3, Eberly 2 *
- Last Time: View Volume Specification and Viewing Transformation
- CG Basics: First of Three Tutorials on OpenGL (Three Parts)
  1. OpenGL & GL Utility Toolkit (GLUT) – V. Shreiner
  2. Basic rendering – V. Shreiner
  3. 3-D viewing setup – E. Angel
- Today: Scan Conversion (aka Rasterization)
  - Lines
    ▶ Incremental algorithm
  - Circles and Ellipses
- Next Time: More Scan Conversion & Intro to Clipping

Where We Are

CTM for “Polygons-to-Pixels” Pipeline
- Entire problem can be reduced to a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite matrix (CTM) is composite of all modeling (instance) transformations (CTM) accumulated during scene graph traversal from root to leaf, composed with the final composite normalizing transformation N applied to the root/world coordinate system:

\[
N = I_{	ext{proj}} S N_{	ext{y}} M_{	ext{view}} T_{	ext{model}}
\]

\[
\begin{align*}
\text{CTM} &= N \cdot \text{CMT} \\
F &= \text{CTM} \cdot F \quad \text{for every vertex } F \text{ defined in its own coordinate system} \\
F_{\text{out}} &= \frac{1}{z_{\text{out}}} F_{\text{out}} \\
\end{align*}
\]

Review: Coordinate Spaces & Transformation Matrices
- Model coordinates (object coordinates)
- World coordinates (scene coordinates)
- Camera coordinates / eye coordinates
- Optional view coordinates / clip coordinates
- Normalized device coordinates (NDC)
- Screen coordinates

Horizontal: modelview transformation
Normalizing transformation: \(\text{model} \rightarrow \text{view}\)

Horizontal: window matrix
(normalization)

Review: Lab 1a & NeHe Tutorials on GameDev

References:
- http://nehe.gamedev.net
- Lab 1 home page: http://www.cis.ksu.edu/~n133

Adapted from slides © 1997 – 2010 van Dam et al., Brown University.
http://bit.ly/hiSt0f
Reused with permission.
Scan Converting Lines

Line Drawing
- Draw a line on a raster screen between two points
- Why is this a difficult problem?
  - What is “drawing” on a raster display?
  - What is a “line” in raster world?
  - Efficiency and appearance are both important

Problem Statement
- Given two points $P$ and $Q$ in XY plane, both with integer coordinates, determine which pixels on raster screen should be on in order to make picture of a unit-width line segment starting at $P$ and ending at $Q$.

What Is Scan Conversion?
- Final step of rasterization (process of taking geometric shapes and converting them into an array of pixels stored in the framebuffer to be displayed).
- Takes place after clipping occurs.
- All graphics packages do this at the end of the rendering pipeline.
- Takes triangles and maps them to pixels on the screen.
- Also takes into account other properties like lighting and shading, but we’ll focus first on algorithms for line scan conversion.

Finding Next Pixel

Special case:
- Horizontal Line: Draw pixel $P$ and increment $x$ coordinate value by 1 to get next pixel.
- Vertical Line: Draw pixel $P$ and increment $y$ coordinate value by 1 to get next pixel.
- Diagonal Line: Draw pixel $P$ and increment both $x$ and $y$ coordinate by 1 to get next pixel.

What should we do in general case?
- Increment $x$ coordinate by 1 and choose point closest to line.
- But how do we measure closest?

Strategy 1: Incremental Algorithm [1]

Basic Algorithm
- Find equation of line that connects two points $P$ and $Q$.
- Starting with leftmost point, increment $x_i$ by 1 to calculate $y_i = m \cdot x_i + b$ where $m = \text{slope}$, $b = \text{intercept}$.
- Draw pixel at $(x_i, \text{Round}(y_i))$ where Round$(y_i) = \text{Floor}(0.5 + y_i)$.

Incremental Algorithm:
- Each iteration requires a floating-point multiplication.
- Modify algorithm to use delta:
  - $(x_{i+1} - y_{i+1}) = m \cdot (x_{i+1} - x_i) + b$
  - $y_{i+1} = y_i + m \cdot (x_{i+1} - x_i)$
  - $x_{i+1} = x_i + 1$

At each step, we make incremental calculations based on preceding step to find next $y$ value.

Vertical Distance

Why can we use vertical distance as measure of which point is closer?
- Because vertical distance is proportional to actual distance.

Strategy 1: Incremental Algorithm [2]

Linear:
- $(x_i, y_i)$
- $(x_i + 1, \text{Round}(y_i + m))$

Non-linear:
- $(x_i, \text{Round}(y_i))$
- $(x_i + 1, y_i + m)$
Strategy 1: Incremental Algorithm [3]

Example Code & Problems

```c
void Line(int x0, int y0, int x1, int y1) {
    int y = y0;
    float dy = y1 - y0;
    float dx = x1 - x0;
    float m = dy / dx;
    y = y0;
    for (x = x0; x <= x1; x++) {
        LineWindow(x, round(y));
        y = y + m;
    }
}
```

Strategy 2: Midpoint Line Algorithm [2]

Example Code & Problems

```c
Strategy 2: Midpoint Line Algorithm [3]
```

- Assume that line’s slope is shallow and positive ($0 < \text{slope} < 1$); other slopes can be handled by suitable reflections about principle axes
- Call lower left endpoint $(x_0, y_0)$ and upper right endpoint $(x_1, y_1)$
- Assume that we have just selected pixel $P$ at $(x_2, y_2)$
- Next, we must choose between pixel to right (E pixel), or one right and one up (NE pixel)
- Let $Q$ be intersection point of line being scan-converted and vertical line $x = x_2 + 1$

Line Equations and Properties

- Line equation as function $f(x): y = mx + b = \frac{dy}{dx} x + b$
- Line equation as implicit function: $f(x, y) = mx - y + b = 0$
- So from above, $x = \frac{y - b}{m}$
- Properties of line $y = mx + b$:
  - $f(x_0, y_0) = 0$ when point $M$ is on line
  - $f(x_0, y_0) < 0$ when point $M$ is above line
- $f(x_0, y_0) > 0$ when point $M$ is below line
- Our decision will be based on value of function at midpoint $M$ at $y = x_2 + 1$
- Decision variable $d$
  - We only need sign of $(x_2 + 1, y_2 + 5)$ to see where line lies, and then pick nearest pixel
  - $d = f(x_2 + 1, y_2 + 5)$
    - if $d > 0$ choose pixel NE
    - if $d < 0$ choose pixel E
    - if $d = 0$ choose either one consistently
- How do we incrementally update $d$?
  - On basis of picking E or NE, figure out location of $M$ for that pixel, and corresponding value of $d$ for next grid line
  - We can derive $d$ for the next pixel based on our current decision

Decision Variable

- Adapted from slides © 1997 – 2010 van Dam et al., Brown University
- Reused with permission.
East Neighbor (E) Case

Increment $M$ by one in a $x$-direction:
\[ d_{\text{new}} = f(x + 2y + 5) = a(x + 2) + b(y + 5) + c \]
\[ d_{\text{old}} = a(x + y + 1) + b(y + 5) + c \]
\[ d_{\text{new}} - d_{\text{old}} = \Delta d_{\text{E}} = \Delta x = a - 2b \]

We can compute values of decision variable at next step incrementally without computing $F(M)$ directly.
\[ d_{\text{new}} = d_{\text{old}} + \Delta d_{\text{E}} + \Delta x \]

$\Delta x$ can be thought of as correction or update factor to take $d_{\text{old}}$ to $d_{\text{new}}$.

It is referred to as forward difference.

Northeast Neighbor (NE) Case

Increment $M$ by one in both $x$ and $y$ directions:
\[ d_{\text{new}} = f(x + 2y + 1.5) = a(x + 2) + b(y + 1.5) + c \]
\[ d_{\text{new}} - d_{\text{old}} = \Delta d_{\text{NE}} = \Delta x + a + b \]
\[ \Delta d_{\text{NE}} = \Delta x = a + b = dx - dy \]

Thus, incrementally,
\[ d_{\text{new}} = d_{\text{old}} + \Delta d_{\text{NE}} = d_{\text{old}} + dx - dy \]

Midpoint Algorithm [1]: Forward Differences

At each step, algorithm chooses between two pixels based on sign of decision variable calculated in previous iteration.

It then updates decision variable by adding either $\Delta d_{\text{E}}$ or $\Delta d_{\text{NE}}$ to old value depending on choice of pixel. Simple additions only!

First pixel is first endpoint $(x_0, y_0)$, so we can directly calculate initial values of $d$ for choosing between E and NE.

Midpoint Algorithm [2]: Initialization and Normalization

First midpoint for first $d = d_{\text{start}} = at(x_0, y_0 + 0.5)$
\[ f(x + 1, y + 0.5) \]
\[ = a(x + 1) + b(y + 0.5) + c \]
\[ = a + b + y + 0.5 + c \]
\[ = f(x, y) + 0.5 \]

But $(x_0, y_0)$ is point on line and $f(x_0, y_0) = 0$.

Therefore, $d_{\text{start}} = a + 0.5 = dy - \frac{dx}{2}$

Use $d_{\text{start}}$ to choose initial pixel, etc.

To eliminate fraction in $d_{\text{start}}$:

- Multiply by multiplying it by $2$: $f(x, y) = 2(ax + by + c)$
- This multiplies each constant and decision variable by 2, but does not change sign.

Bresenham’s Midpoint Line Algorithm: Pseudocode

```c
void MidpointLine(int x0, int y0, int x1, int y1) { 
    int dx = (x1 - x0), dy = (y1 - y0);
    int d = 2 * dy - dx;
    int incx = 2 * dy;
    int incy = 2 * dx;
    int x = x0, y = y0;
    writePixel(x, y);

    while (x < x1) {
        if (d < 0) { d += incx; // East Case
                     ++x;
        } else { d += incy; // Northeast Case
                  ++x;
                  ++y;
        }
    }
}
```

Preview: Drawing Circles, Versions 1 & 2

Version 1: really bad
For $x$ from $-R$ to $R$:
\[ y = \sqrt{R^2 - x^2} \]
Pixel $(\text{round}(x), \text{round}(y))$;
Pixel $(\text{round}(x), \text{round}(y))$;

Version 2: slightly less bad
For $x$ from 0 to 360:
\[ y = \sqrt{R^2 - x^2} \]
Pixel $(\text{round}(R \cdot \cos(x)), \text{round}(R \cdot \sin(x)))$;
### Summary

- Lab 1: Based on First of Three Tutorials on OpenGL (Three Parts)
- Lecture 5: Viewing 3 of 4 – Graphics Pipeline (§2.3.2 - 2.3.7, pp. 48-66)
- See Also: CG Basics 1-2
  - CG Basics 1: Mathematical Foundations
  - CG Basics 2: OpenGL Primer 1 of 3 (in greater detail)
- Today: Scan Conversion (aka Rasterization)
  - Lines
    - Incremental algorithm
  - Symmetries (8) and reduction to two-case analysis: $E$ vs. $NE$
    - Decision variable and method of forward differences
    - (Bresenham’s) midpoint line algorithm
- Circles and Ellipses
- Next Time: More Scan Conversion & Intro to Clipping
  - Polygons: scan line interpolation
    - Clipping basics: 2-D problem definition and examples

### Terminology

- Picture elements (pixels)
- Scan Conversion (aka Rasterization)
  - Given: geometric object (e.g., line segment, projected polygon)
  - Decide: what pixels to light (turn on; later, color/shade)
  - Basis: what part of pixels crossed by object
  - Issues (Reasons why Scan Conversion is Nontrivial Problem)
    - Aliasing (e.g., jaggies) – discontinuities in lines
    - Cracks: discontinuities in “polygon” mesh
- Line Drawing
  - Incremental algorithm – uses rounding, floating point arithmetic
  - Forward differences – precalculated amounts to add to running total
    - Midpoint line algorithm – uses forward differences
      - For lines: Bresenham’s algorithm
      - For circles and ellipses