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

Public mirror web site: http://www.kddresearch.org/Courses/CIS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

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

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

Lecture Outline

• Reading for Last Class: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2e
• Reading for Today: §2.5.1, 3.1 Eberly 2e
• Reading for Next Class: §2.3.5, 2.4, 3.1.3, Eberly 2e
• 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
    • Bresenham’s algorithm & midpoint line algorithm
  • Circles and Ellipses
• Next Time: More Scan Conversion & Intro to Clipping
Where We Are

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
<th>Primary Source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Course Overview</td>
<td>Chapter 1, Eberly <em>Z</em></td>
</tr>
<tr>
<td>1</td>
<td>CG Basics: Transformation Matrices; Lab 0</td>
<td>Sections (b) 2.1, 2.2</td>
</tr>
<tr>
<td>2</td>
<td>Viewing 1: Overview, Projections.</td>
<td>§ 2.2.3 – 2.2.4, 2.8</td>
</tr>
<tr>
<td>3</td>
<td>Viewing 2: Viewing Transformation</td>
<td>§ 2.3 esp. 2.3.4, PVFHT slides</td>
</tr>
<tr>
<td>4</td>
<td>Lab 1a: Flash &amp; OpenGL Basics</td>
<td>Ch. 2, 18, Angel Primer</td>
</tr>
<tr>
<td>5</td>
<td>Viewing 3: Graphics Pipeline</td>
<td>§ 3.3 esp. 3.3.7, 3.6, 3.7</td>
</tr>
<tr>
<td>6</td>
<td>Scan Conversion 1: Lines, Midpoint Algorithm</td>
<td>§ 2.5.1, 3.1, PVFHT slides</td>
</tr>
<tr>
<td>7</td>
<td>Viewing 4: Clipping &amp; Culling, Lab 1b</td>
<td>§ 2.3.8, 2.4, 3.1.3</td>
</tr>
<tr>
<td>8</td>
<td>Scan Conversion 2: Polygons, Clipping intro</td>
<td>§ 2.4, 2.5 esp. 2.5.4, 3.1.6</td>
</tr>
<tr>
<td>9</td>
<td>Surface Detail 1: Illumination &amp; Shading</td>
<td>§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2</td>
</tr>
<tr>
<td>10</td>
<td>Lab 2a: Direct3D / DirectX Intro</td>
<td>§ 2.7, Direct3D handout</td>
</tr>
<tr>
<td>11</td>
<td>Surface Detail 2: Textures, OpenGL Shading</td>
<td>§ 2.6.2, 20.3 – 20.4, Primer</td>
</tr>
<tr>
<td>12</td>
<td>Surface Detail 3: Mappings, OpenGL Textures</td>
<td>§ 20.5 – 20.19</td>
</tr>
<tr>
<td>13</td>
<td>Surface Detail 4: Pixel/Vertex Shading; Lab 2b</td>
<td>§ 3.1</td>
</tr>
<tr>
<td>14</td>
<td>Surface Detail 5: DirectX Shading; CGLSL</td>
<td>§ 3.2 – 3.4, DirectX handout</td>
</tr>
<tr>
<td>15</td>
<td>Demos 1: CGA, Fun Scene Graphs; State</td>
<td>§ 4.1 – 4.3, CGA handout</td>
</tr>
<tr>
<td>16</td>
<td>Lab 3a: Shading &amp; Transparency</td>
<td>§ 2.6, 20.1, Primer</td>
</tr>
<tr>
<td>17</td>
<td>Animation 1: Basics, Keyframes; HW/Exam</td>
<td>§ 5.1 – 5.2</td>
</tr>
<tr>
<td>18</td>
<td>Scene Graphs: Rendering; Lab 3b: Shader</td>
<td>§ 4.4 – 4.7</td>
</tr>
<tr>
<td>19</td>
<td>Demos 2: SFX, Skinning, Morphing</td>
<td>§ 8.3 – 8.6, CGA handout</td>
</tr>
<tr>
<td>20</td>
<td>Demos 3: Surfaces, B-reps/Volume Graphics</td>
<td>§ 10.4, 12.7, Mesh handout</td>
</tr>
</tbody>
</table>

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 (CMTM) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation N applied to the root/world coordinate system:
  1. $N = D_{persp}S_{far}S_{xy}M_{rot}T_{trans}$
  2. $CTM = N \cdot CMTM$
  3. $P' = CTM \cdot P$ for every vertex P defined in its own coordinate system
  4. $P_{screen} = 512 \cdot P' + 1$ for all clipped $P'$

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
Review: Lab 1a & NeHe Tutorials on GameDev

References

NeonHelium tutorials: http://nehe.gamedev.net
Mesa home page: http://www.mesa3d.org

1. (20%) Mesa setup. Log into your Gentoo Linux account in Nichols 128, the Linux Lab. Go to the NeHe site and follow the “Setting up OpenGL in MacOS” to create a GL window. As in MacOS X, Gentoo keeps its GL include files in /usr/include/GL. Name your program labi_1.c and include it in your lab assignment submission. Take a screen shot of the window and save it in GIMP as labi_1.jpg.

2. (20%) Polygon rastorization (scan conversion). Follow Lesson 02 to draw a 2-D polygon and shade it using smooth (Gouraud) and constant (flat) shading. Turn in labi_2.c and labi_2.jpg.

3. (20%) Modelview transformation: 3-D Rotation of 2-D objects. Follow Lesson 03 to rotate the flat polygons and then render them. Turn in labi_3.c and labi_3.jpg.

4. (20%) Modelview transformation: 3-D Rotation of 3-D objects. Follow Lesson 04 to draw 3-D polyhedra and rotate them. Turn in labi_4.c and labi_4.jpg.

5. (20%) XWindows. Repeat Lesson 04 from a notebook computer or PC running Mac OS X, Windows XP or Windows Vista. Turn in labi_5.jpg.

Review: Coordinate Spaces & Transformation Matrices

(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-86, especially p. 58)

1. model coordinates / object coordinates
2. world coordinates / scene coordinates
3. camera coordinates / eye coordinates
4. (optional) view coordinates / clip coordinates
5. normalized device coordinates (NDC)
6. screen coordinates

$$X_{\text{model}} \rightarrow (H_{\text{world}})$$
$$X_{\text{world}} \rightarrow (H_{\text{view}})$$
$$X_{\text{view}} \rightarrow (H_{\text{proj}})$$
$$X_{\text{clip}} \rightarrow (\text{perspective division})$$
$$X_{\text{ndc}} \rightarrow (H_{\text{window}})$$
$$X_{\text{window}}$$

$$H_{\text{world}}$$: modelview transformation

Normalizing transformation: $$X_{\text{world}} \rightarrow X_{\text{ndc}}$$

$$H_{\text{view}}$$: “view matrix” (really NT!)

$$H_{\text{proj}}$$: projection matrix

$$H_{\text{window}}$$: perspective division

$$H_{\text{window}}$$: window matrix

(aka viewport transformation)
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 rasterisation (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”?

Vertical Distance

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

- Similar triangles show that true distances to line (in blue) are directly proportional to vertical distances to line (in black) for each point
- Therefore, point with smaller vertical distance to line is closest to line

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
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 * x_i + B
  where m = slope, B = y intercept
- Draw pixel at (x_i, Round(y_i)) where
  Round(y_i) = Floor(0.5 + y_i)

Incremental Algorithm:
- Each iteration requires a floating-point multiplication
- Modify algorithm to use deltas
  - (y_{i+1} - y_i) = m * (x_{i+1} - x_i) + B
  - y_{i+1} = y_i + m * (x_{i+1} - x_i)
  - If Δx = 1, then y_{i+1} = y_i + m
- At each step, we make incremental calculations based on preceding step to find next y value

Strategy 1: Incremental Algorithm [2]
Strategy 1: Incremental Algorithm [3]

Example Code & Problems

```c
void Line(int x0, int y0, int x1, int y1) {
    int x, y;
    float dy = y1 - y0;
    float dx = x1 - x0;
    float m = dy / dx;

    y = y0;
    for (x = x0; x < x1; ++x) {
        WritePixel(x, Round(y));
        y = y + m;
    }
}
```

Since slope is fractional, need special case for vertical lines (dx = 0)
Rounding takes time

Adapted from slides © 1997 – 2010 van Dam et al., Brown University

---

Strategy 2: Midpoint Line Algorithm [1]

- Assume that line's slope is shallow and positive (0 < 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 \((x_p, y_p)\)
- 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 \(x = x_p + 1\)

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
Strategy 2: Midpoint Line Algorithm [2]

1. Line passes between E and NE
2. Point that is closer to intersection point Q must be chosen
3. Observe on which side of line midpoint $M$ lies:
   - E is closer to line if midpoint $M$ lies above line, i.e., line crosses bottom half
   - NE is closer to line if midpoint $M$ lies below line, i.e., line crosses top half
4. Error (vertical distance between chosen pixel and actual line) is always $\leq .5$

- Algorithm chooses NE as next pixel for line shown
- Now, need to find a way to calculate on which side of line midpoint lies
Line Equations and Properties

Line equation as function \( f(x) \):
\[
y = mx + B = \frac{dy}{dx} x + B
\]

Line equation as implicit function:
\[
f(x, y) = ax + by + c = 0
\]

So from above,
\[
y' dx = dy' x + B' dx
\]
\[
dy' x - y' dx + B' dx = 0
\]
\[
a = dy, b = -dx, c = B' dx
\]

Properties (proof by case analysis):
- \( f(x_m, y_m) = 0 \) when any point \( M \) is on line
- \( f(x_m, y_m) < 0 \) when any point \( M \) is above line
- \( f(x_m, y_m) > 0 \) when any point \( M \) is below line
- Our decision will be based on value of function at midpoint \( M \) at \((x_p + 1, y_p + .5)\)

Decision Variable

Decision Variable \( d \):
- We only need sign of \( f(x_p + 1, y_p + .5) \) to see where line lies, and then pick nearest pixel
- \( d = f(x_p + 1, y_p + .5) \)
  - 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 \( d \) for next grid line
- We can derive \( d \) for the next pixel based on our current decision
East Neighbor (E) Case

Increment \( M \) by one in \( x \) direction

\[
d_{\text{new}} = f(x_p + 2, y_p + .5)
= a(x_p + 2) + b(y_p + .5) + c
\]

\[
d_{\text{old}} = a(x_p + 1) + b(y_p + .5) + c
\]

\[d_{\text{new}} - d_{\text{old}}\] is the incremental difference \( \Delta E \)

\[
d_{\text{new}} = d_{\text{old}} + \Delta E = d_{\text{old}} + a
\]

\( \Delta E = a = dy \) (2 slides back)

We can compute value of decision variable at next step incrementally without computing \( F(M) \) directly

\[
d_{\text{new}} = d_{\text{old}} + \Delta E - d_{\text{old}} + dy
\]

\( \Delta E \) 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

\[
d_{\text{new}} = f(x_p + 2, y_p + 1.5)
= a(x_p + 2) + b(y_p + 1.5) + c
\]

\( \Delta NE = d_{\text{new}} - d_{\text{old}} \)

\[
d_{\text{new}} = d_{\text{old}} + a + b
\]

\( \Delta NE = a + b = dx - dy \)

Thus, incrementally,

\[
d_{\text{new}} = d_{\text{old}} + \Delta NE = d_{\text{old}} + dx - dy
\]
Midpoint Algorithm [1]: Forward Differences

- At each step, algorithm chooses between 2 pixels based on sign of decision variable calculated in previous iteration.
- It then updates decision variable by adding either ΔE or ΔNE 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 value of \(d\) for choosing between \(E\) and \(NE\).

Midpoint Algorithm [2]: Initialization and Normalization

- First midpoint for first \(d = d_{\text{start}}\) is at \((x_0 + 1, y_0 + .5)\)
- \[ f(x_0 + 1, y_0 + .5) = a(x_0 + 1) + b(y_0 + .5) + c \]
- \[ = a \cdot x_0 + b \cdot y_0 + a + \frac{b}{2} + c \]
- \[ = f(x_0, y_0) + a + \frac{b}{2} \]
- But \((x_0, y_0)\) is point on line and \(f(x_0, y_0) = 0\)
- Therefore, \(d_{\text{start}} = a + \frac{b}{2} = dy - \frac{dx}{2}\)
  - use \(d_{\text{start}}\) to choose second pixel, etc.
- To eliminate fraction in \(d_{\text{start}}\):
  - redefine \(f\) 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 incrE = 2 * dy;
    int incrNE = 2 * (dy - dx);
    int x = x0, y = y0;
    writePixel(x, y);

    while (x < x1) {
        if (d <= 0)  d = d + incrE; // East Case
        else        d = d + incrNE, ++y; // Northeast Case
        ++x;
        writePixel(x, y);
    }
}
```

Adapted from slides © 1997 – 2010 van Dam et al., Brown University

Preview: Drawing Circles, Versions 1 & 2

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

Version 2: slightly less bad
For x from 0 to 360:
Pixel (round ($R \cos(x)$)),
round ($R \sin(x)$));

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
### Drawing Circles, Version 3

- Symmetry: If \((x_0 + a, y_0 + b)\) is on circle
  - also \((x_0 \pm a, y_0 \pm b)\) and \((x_0 \pm b, y_0 \pm a)\), hence 8-way symmetry.

- Reduce the problem to finding the pixels for 1/8 of the circle

\[
\begin{align*}
(x_0 + a, y_0 + b)^2 &= (x_0, y_0)^2 + R^2 \\
(x_0 - x_0)^2 + (y_0 - y_0)^2 &= R^2
\end{align*}
\]

### Using The Symmetry

- Scan top right 1/8 of circle of radius \(R\)
- Circle starts at \((x_0, y_0 + R)\)
- Let’s use another incremental algorithm with decision variable evaluated at midpoint
Summary

- Lab 1a: 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