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

KSOL course pages: http://bit.ly/hGvXlH / http://bit.ly/eVizrE
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
Wayback Machine archive of Brown CS123 slides: http://bit.ly/gAhJbh

Lecture Outline

- Reading for Last Class: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2 e
- 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

Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2e
1	**CG Basics: Transformation Matrices; Lab 0**	**Sections (§) 2.1, 2.2**
2	Viewing 1: Overview, Projections	§ 2.2.3 – 2.2.4, 2.8
3	Viewing 2: Viewing Transformation	§ 2.3 esp. 2.3.4; FVFH slides
4	**Lab 1a: Flash & OpenGL Basics**	**Ch. 2, 16[1], Angel *Primer***
5	Viewing 3: Graphics Pipeline	§ 2.3 esp. 2.3.7; 2.6, 2.7
6	Scan Conversion 1: Lines, Midpoint Algorithm	§ 2.5.1, 3.1; FVFH slides
7	**Viewing 4: Clipping & Culling; Lab 1b**	**§ 2.3.5, 2.4, 3.1.3**
8	Scan Conversion 2: Polygons, Clipping Intro	§ 2.4, 2.5 esp. 2.5.4, 3.1.6
9	Surface Detail 1: Illumination & Shading	§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2
10	**Lab 2a: Direct3D / DirectX Intro**	**§ 2.7, Direct3D handout**
11	Surface Detail 2: Textures; OpenGL Shading	§ 2.6.3, 20.3 – 20.4, *Primer*
12	Surface Detail 3: Mappings; OpenGL Textures	§ 20.5 – 20.13
13	**Surface Detail 4: Pixel/Vertex Shad.; Lab 2b**	**§ 3.1**
14	Surface Detail 5: Direct3D Shading; OGLSL	§ 3.2 – 3.4, Direct3D handout
15	Demos 1: CGA, Fun; Scene Graphs: State	§ 4.1 – 4.3, **CGA handout**
16	**Lab 3a: Shading & Transparency**	**§ 2.6, 20.1, *Primer***
17	**Animation 1: Basics, Keyframes; HW/Exam**	**§ 5.1 – 5.2**
	Exam 1 review; Hour Exam 1 (evening)	**Chapters 1 – 4, 20**
18	**Scene Graphs: Rendering; Lab 3b: Shader**	**§ 4.4 – 4.7**
19	**Demos 2: SFX; Skinning, Morphing**	**§ 5.3 – 5.5, CGA handout**
20	Demos 3: Surfaces; B-reps/Volume Graphics	§ 10.4, 12.7, **Mesh handout**

Lightly-shaded entries denote the due date of a written problem set; heavily-shaded entries, that of a machine problem (programming assignment); blue-shaded entries, that of a paper review; and the green-shaded entry, that of the term project.

Green, **blue** and **red** letters denote **exam review**, **exam**, and **exam solution review** dates.

Review: 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
http://bit.ly/hiSt0f Reused with permission.

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 `lab1_1.c` and include it in your lab assignment submission. Take a screen shot of the window and save it in GIMP as `lab1_1.jpg`.
2. **(20%) Polygon rasterization (scan conversion).** Follow Lesson 02 to draw a 2-D polygon and shade it using smooth (Gouraud) and constant (flat) shading. Turn in `lab1_2.c` and `lab1_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 `lab1_3.c` and `lab1_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 `lab1_4.c` and `lab1_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 `lab1_5.jpg`.

Review: Coordinate Spaces & Transformation Matrices

(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-66, especially p. 58)

1. model coordinates / object coordinates	$X_{model} \rightarrow (H_{world})$
2. world coordinates / scene coordinates	$X_{world} \rightarrow (H_{view})$
3. camera coordinates / eye coordinates	$X_{view} \rightarrow (H_{proj})$
4. (optional) view coordinates / clip coordinates	$X_{clip} \rightarrow$ (perspective division)
5. normalized device coordinates (NDC)	$X_{ndc} \rightarrow (H_{window})$
6. screen coordinates	X_{window}

H_{world}: modelview transformation

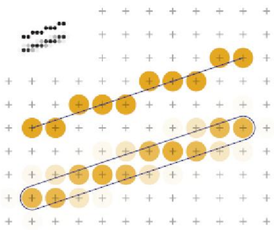
Normalizing transformation: $X_{world} \rightarrow X_{ndc}$ {
- H_{view}: "view matrix" **(really NT!)**
- H_{proj}: projection matrix
- /w: perspective division

H_{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.

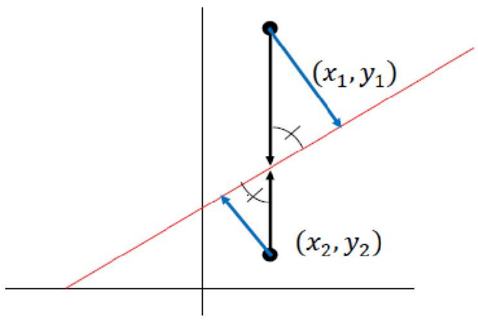
- Vertical Line:
  Draw pixel P and increment y coordinate value by 1 to get next pixel.

- 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

11

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 $y_i = m * x_i + \mathrm{B}$
  where m = slope, B = y intercept
- Draw pixel at $(x_i, \mathrm{Round}(y_i))$ where
  Round (y_i) = Floor $(0.5 + y_i)$

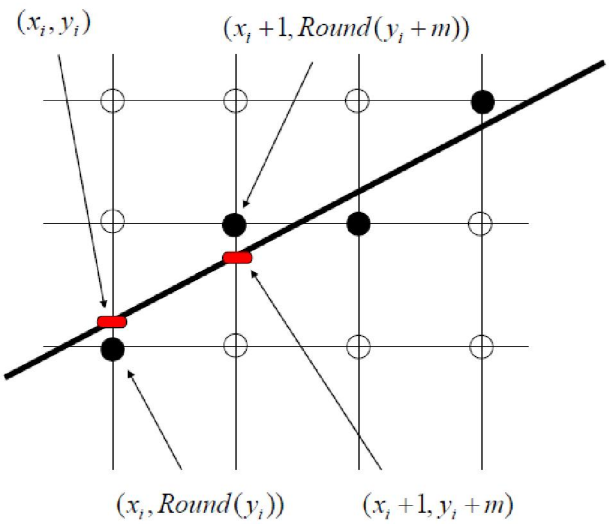
Incremental Algorithm:

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

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

12

Strategy 1: Incremental Algorithm [2]



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

Strategy 1: Incremental Algorithm [3] Example Code & Problems

```
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;
        }

}
```

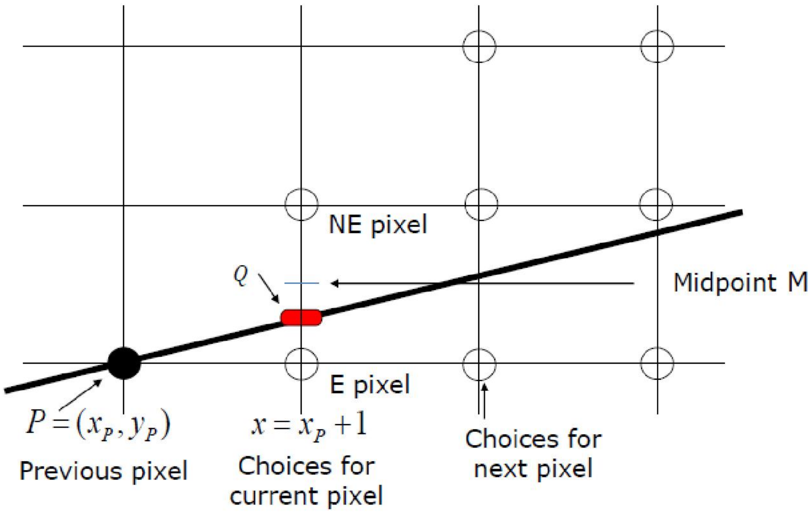
Since slope is fractional, need special case for vertical lines (dx = 0)

Rounding takes time

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 $x = x_P + 1$

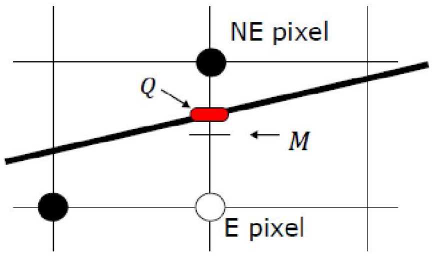
Strategy 2: Midpoint Line Algorithm [2]



Strategy 2: Midpoint Line Algorithm [3]

- Line passes between E and NE
- Point that is closer to intersection point Q must be chosen
- 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
- 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: $f(x, y) = ax + by + c = 0$

for coefficients a, b, c, where $a, b \neq 0$

So from above,

$$y \cdot dx = dy \cdot x + B \cdot dx$$
$$dy \cdot x - y \cdot dx + B \cdot dx = 0$$
$$\therefore a = dy, b = -dx, c = B \cdot dx$$

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

$$d_{new} = f(x_P + 2, y_P + .5)$$
$$= a(x_P + 2) + b(y_P + .5) + c$$

$$d_{old} = a(x_P + 1) + b(y_P + .5) + c$$

- $d_{new} - d_{old}$ is the incremental difference ΔE

  $d_{new} = d_{old} + a$
  $\Delta E = a = dy$ (2 slides back)

- We can compute value of decision variable at next step incrementally without computing *F(M)* directly
  $d_{new} = d_{old} + \Delta E = d_{old} + dy$

- ΔE can be thought of as correction or update factor to take d_{old} to d_{new}
- It is referred to as forward difference

Northeast Neighbor (NE) Case

Increment M by one in both *x* and *y* directions

$$d_{new} = f(x_P + 2, y_P + 1.5)$$
$$= a(x_P + 2) + b(y_P + 1.5) + c$$

- $\Delta NE = d_{new} - d_{old}$

  $d_{new} = d_{old} + a + b$

  $\Delta NE = a + b = dx - dy$

- Thus, incrementally,

$$d_{new} = d_{old} + \Delta NE = d_{old} + dx - dy$$

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 $d = d_{start}$ is at $(x_0 + 1, y_0 + .5)$
- $f(x_0 + 1, y_0 + .5)$
- $= a(x_0 + 1) + b(y_0 + .5) + c$
- $= a * x_0 + b * y_0 + a + \frac{b}{2} + c$
- $= f(x_0, y_0) + a + \frac{b}{2}$
- But (x_0, y_0) is point on line and $f(x_0, y_0) = 0$
- Therefore, $d_{start} = a + \frac{b}{2} = dy - \frac{dx}{2}$
  - use d_{start} to choose second pixel, etc.
- To eliminate fraction in d_{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

23

Bresenham's Midpoint Line Algorithm: Pseudocode

```
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);
    }
}
```


24

Preview: Drawing Circles, Versions1 & 2

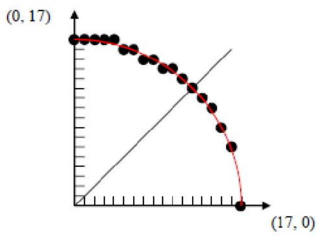
Version 1: really bad

For x from $-R$ *to* R:

$y = \sqrt{R * R \ - x * x}$;

Pixel (round(x), round(y));

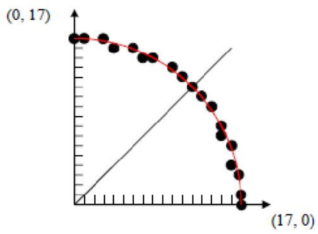
Pixel (round(x), round($-y$));



Version 2: slightly less bad

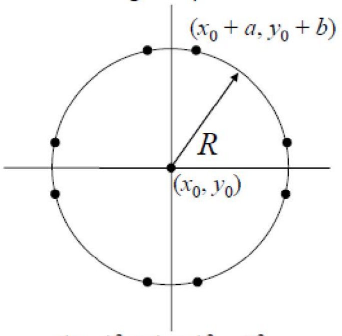
For x from 0 *to* 360:

Pixel (round ($R * \cos(x)$),
round ($R * \sin(x)$));



Preview: Drawing Circles, Version 3

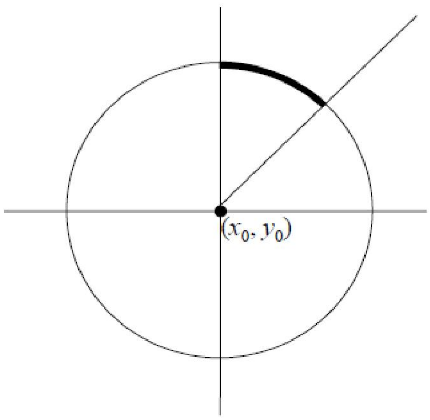
- Symmetry: If $(x_0 + a, y_0 + b)$ is on circle
  - also $(x_0 \pm a, y_0 \pm b)$ and $(x_0 \pm b, y_0 \pm a)$, hence 8-way symmetry.
- Reduce the problem to finding the pixels for 1/8 of the circle



$(x\text{-}x_0)^2 + (y\text{-}y_0)^2 = R^2$

Preview: 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