Lecture 3
Rasterization:
Lines and Circles
Outline

2D rendering pipeline
Graphics primitives

Drawing straight lines
  Digital differential analyzer (DDA)
  Midpoint line algorithm
  Bresenham’s algorithm

Drawing circles
  Uniform angular sampling
  Midpoint circle algorithm
Rendering Pipeline

Model definition
Represent each graphical object in digital form

Model transform
Transform the object’s coordinates to place it in the scene

View transform
Place a virtual camera in the scene

Render
Draw the scene to the camera image

Display the 2D image
Set the voltages that control each pixel in the display
Represent each graphical object in digital form

Points

Lines

Curves

Filled 2D shapes
Transform object to place it in the scene
  e.g., transform a unit sphere to a sphere with radius r at (x, y, z)

Transformations: scale, rotation, translation, …
Place a virtual camera in the scene

Transform the scene coordinates to camera coordinates to determine what is seen through the camera lens
Set the pixel values of the camera image according to what objects are seen by the camera at each pixel.

Object-order rendering

- Render each object one at a time
- For overlapping objects
  - Overwrite pixels with front object if opaque objects overlap
  - Blend pixel colors if transparent objects overlap
Set the pixel values of the camera image according to what objects are seen by the camera at each pixel

**Image-order rendering**
- Render each pixel in one at a time (e.g., in raster-scan order)
- For each pixel
  - Determining what objects affect the pixel
  - Determine the pixel’s color from the objects
Use the pixel color to control voltages

Voltages control the brightness of locations on the display device
  e.g., Strength of the electron beam in CRTs
  e.g., How much light passes through the liquid crystal material in LCDs
Graphics primitives

Common shapes that can be combined to build complex objects or a complex scene

2D examples
• Lines
• Circles
• Images

3D examples
• Spheres
• Cubes
• 3D triangles
Drawing graphics primitives

1. Determine a digital representation of the shape
2. Determine which image pixels are affected by the shape
3. Set the pixel colors accordingly
Representing graphics primitives

Possible digital representations:

Pictures
• A 2D array of values
• e.g. a bitmap letter, an icon, an image

Equations
• e.g. \( y = mx + b \) (a straight line with slope \( m \) and y-intercept \( b \))
• e.g. \( y^2 + x^2 = R^2 \) (a circle centered at \((0, 0)\) with radius \( R \))

Procedures
• e.g. \( \text{image}(i, j) = \text{rand(sin}(i + 0.1 \times \text{rand}(j))) \)
Rendering graphics primitives

Determine an intensity or color for each pixel affected by the object

e.g., Render a red line from pixel (1, 1) to pixel (1,9)

e.g., Render an image of size 300x300 pixels into a 100x100 window
Line Drawing Algorithms
A line can be defined by...

... its slope $m$ and its y-intercept $b$.

Points on the line satisfy the equation $y = mx + b$
A line can be defined by...

... two points \((x_0, y_0)\) and \((x_1, y_1)\).

To convert to the previous representation, substitute the two points into \(y = mx + b\) and solve for \(m\) and \(b\).
A line can be defined by...

... a point \((x_0, y_0)\) and a vector \(\mathbf{v}\).

\[(x, y) = (x_0, y_0) + k\mathbf{v}\] is on the line, where \(k\) is a scalar value.
Explicit form

One variable is expressed as a function of the other variable(s)

In 2D, y is expressed as an explicit function of x:

- Line: \( y = mx + b \)
- Circle: \( y = \sqrt{R^2 - x^2} \)
Implicit form

Points on the shape satisfy an implicit function

In 2D, a point \((x, y)\) is on the shape if \(f(x, y) = 0\):

- Line: \(f(x, y) = mx + b - y\)
- Circle: \(f(x, y) = R^2 - x^2 - y^2\)
Parametric form

Points on the shape are expressed in terms of a separate parameter (not $x$ or $y$)

Line segment:
\[ x = (1-t) x_0 + t x_1 \quad t \in [0, 1] \]
\[ y = (1-t) y_0 + t y_1 \]

Points on the line are a linear combination of the endpoint positions $(x_0, y_0)$ and $(x_1, y_1)$

Circle:
\[ x = R \cos \Theta \quad \Theta \in [0, 2\pi] \]
\[ y = R \sin \Theta \]

Points on the circle are swept out as $\Theta$ ranges from 0 to $2\pi$
Drawing straight lines

A line segment is continuous
  All points between the two endpoints belong to the line

A digital image is discrete
  Can only set pixels at discrete locations
  Pixels have a finite size
  Which pixels between the two endpoints should be set?
Exercise

1. On the handout, draw the following line segments

   A. (1, 15) to (8, 15)
   B. (3.5, 13.5) to (11.5, 13.5)
   C. (1, 6) to (6, 11)
   D. (1, 5) to (8, 7)
   E. (1, 0) to (7, 4)

2. Fill in the pixels you think best render each line

   Think about how you are deciding which pixels to color
   Can you describe your algorithm?
Exercise

Hold your paper at arms length and look at your lines

What do you think about the quality of your lines?

Is the thickness consistent along the line?
Do some regions appear thicker than others?
Are there any breaks or gaps in your line?
How does the thickness compare between lines?
Drawing straight lines

Drawing vertical and horizontal lines is straightforward

Set pixels nearest the line endpoints and all the pixels in between
Drawing straight lines

What about non-vertical, non-horizontal lines?
How do we determine which pixels to set?
Many approaches are possible, most of these are problematic
Naïve approach

Set every pixel traversed by the line

Can create thick, chunky lines
Naïve approach

Set every pixel traversed by the line

Apparent line thickness depends on slope
Line as a thin rectangle

Set each pixel whose center falls inside the rectangle

What is the ideal width of the rectangle?
Line as a thin rectangle

Set each pixel whose center falls inside the rectangle

What is the ideal width of the rectangle?

Too thick: Line looks chunky
Line as a thin rectangle

Set each pixel whose center falls inside the rectangle

What is the ideal width of the rectangle?

Too thin: Gaps appear in line
Line as a thin rectangle

The ideal rectangle width depends on the slope of the line
Line drawing algorithms

Uniform sampling
Digital Differential Analyzer DDA)
Midpoint line algorithm
Bresenham’s algorithm
Uniform sampling

Sample the line at equal intervals

Set the pixel closest to each sample point
Uniform sampling

Use a parametric expression of the line:
- $x = (1-t)x_0 + tx_1 \quad t \in [0, 1]$
- $y = (1-t)y_0 + ty_1$

Sample the line at equal intervals of $t$
- Sample at $t = 0, 1/N, 2/N, \ldots 1$ to get $N+1$ samples
Uniform sampling

What is the ideal value of $N$?

$N$ too big: Line looks chunky and some pixels set more than once

Artifacts if the line is transparent
Uniform sampling

What is the ideal value of $N$?

$N$ too small: Gaps appear in line
Uniform sampling

The best N depends on the slope of the line

Ideal spacing = $\sqrt{2}$ pixel

Ideal spacing = 1 pixel
Digital Differential Analyzer (DDA)

Guarantees the thinnest possible line with no gaps

Sample the line at equal intervals
  • Determine the optimal intervals from the slope of the line
Digital Differential Analyzer (DDA)

Guarantees the thinnest possible line with no gaps

For lines that are more horizontal than vertical:
   Sample once for each column of pixels between the endpoints
Digital Differential Analyzer (DDA)

Guarantees the thinnest possible line with no gaps

For lines that are more vertical than horizontal:
Sample once for each row of pixels between the endpoints
Digital Differential Analyzer (DDA)

Four cases to consider:

1) Positive slope \( m \in [0, 1] \)

2) Positive slope \( m \in [1, \infty] \)

3) Negative slope \( m \in [0, -1] \)

4) Negative slope \( m \in [-1, -\infty] \)
Digital Differential Analyzer (DDA)

Consider the first case: Positive slope $m \in [0, 1]$

For now, assume integer endpoints $(x_0, y_0)$ and $(x_1, y_1)$
Order the endpoints from left to right
Digital Differential Analyzer (DDA)

Choose the sampling interval to sample once for each column between the endpoints.

The number of pixels to be set is \((N + 1)\), where \(N = x_1 - x_0\)

![Diagram illustrating the DDA process with points \(P_0(0, 0)\) and \(P_1(7, 3)\) and the calculation of \(N = 7\) and \(#\) pixels to set = 8.]

Produces the thinnest possible line without gaps.
DDA basic algorithm

Recall the parametric form of the line
\[ x = (1-t) x_0 + t x_1 \quad t \in [0, 1] \]
\[ y = (1-t) y_0 + t y_1 \]

Sample the line at \( t = 0, 1/N, 2/N, \ldots 1 \)

\[ x_i = x_0 + i \]
\[ y_i = (1 - t_i) y_0 + t_i y_1 \]
DDA basic algorithm

\[
N = (x_1 - x_0)
\]
for (i = 0; i <= N; i++)
{
    t = i / N
    setPixel (x_0 + i, round((1 – t) y_0 + t y_1))
}

Cost per new pixel: 1 divide, 3 additions, 2 multiplies, and 1 round
DDA more efficiently

DDA is an incremental method

\[ x_i = x_0 + i \]

\[ y_i = (1 - t_i) y_0 + t_i y_1 \]
\[ y_{i+1} = (1 - t_{i+1}) y_0 + t_{i+1} y_1 \]

\[ y_{i+1} - y_i = (-t_{i+1} + t_i) y_0 + (t_{i+1} - t_i) y_1 \]
\[ = ((-i - 1)/N + i/N) y_0 + (((i + 1)/N - i/N) y_1 \]
\[ = (y_1 - y_0) / N \]
\[ = (y_1 - y_0) / (x_1 - x_0) \]
\[ = m \]

For each sample, x increases by 1, and y increases by m
DDA more efficiently

Modified algorithm:

\[
i = x_0 \\
m = (y_1 - y_0) / (x_1 - x_0) \\
y = y_0 \\
while (i <= x_1) \\
\{
    setPixel (i, round(y)) \\
    i = i + 1 \\
    y = y + m \\
\}
\]

Cost per new pixel: 2 additions and 1 round
DDA with floating point endpoints

Final algorithm:

\[
i = \text{round}(x_0) \\
m = \frac{(y_1 - y_0)}{(x_1 - x_0)} \\
y = y_0 + m \cdot (i - x_0) \\
\text{while } (i \leq x_1) \\
\{ \\
    \text{setPixel } (i, \text{round}(y)) \\
    i = i + 1 \\
    y = y + m \\
\}\]

Have eliminated the multiplications and division
Round is still required
y and m are both floats – problematic for fixed point processors
Midpoint line algorithm

Extends the DDA to avoid floating point calculation
Each new pixel requires one or two integer adds and a sign test
Midpoint line algorithm

Consider the case where \( m \in [0,1] \)

\( x \) increases by 1 for each sample along the line
- The next pixel will be in the next column
Midpoint line algorithm

y increases along the line, but more slowly than x

- For each unit change in x, y will change by less than one
- Sometimes the next pixel will be on the same row (to the right)
- Sometimes it will be on the row above (to the right and up)
Midpoint line algorithm

Sample once along the line at every column

At each sample, determine if the next pixel is to the right or to the right and up from the current pixel.
Midpoint line algorithm

Basic algorithm:

\[
\begin{align*}
    i &= x_0 \\
    j &= y_0 \\
    \text{while} \ (i \leq x_1) \\
    \{ \\
        \text{setPixel} \ (i, j) \\
        i &= i + 1 \\
        \text{if} \ (\text{Condition}) \\
        \quad j &= j + 1 \\
    \}
\end{align*}
\]

Condition() determines which of the two candidates is the next pixel.
The key is to determine an efficient Condition().
Midpoint line algorithm

Assume that for the column $i$, the pixel $(i, j)$ was set.
Need to determine if the next pixel is pixel $(i+1, j)$ or $(i+1, j+1)$.
Midpoint line algorithm

Consider the midpoint between these two pixel centers: \((i+1, j+\frac{1}{2})\)

- If the midpoint is above the line, the pixel \((i+1, j)\) is closer to the line
- If the midpoint is below the line, the pixel \((i+1, j+1)\) is closer to the line
Midpoint line algorithm

Consider the midpoint between these two pixel centers: \((i+1, j^{+1/2})\)

- If the midpoint is above the line, the pixel \((i+1, j)\) is closer to the line
- If the midpoint is below the line, the pixel \((i+1, j+1)\) is closer to the line
Midpoint line algorithm

Set this pixel \((i+1, j)\)
Advance to the next sample point
Consider the midpoint between the two candidate pixels

\[ \text{Midpoint between the centers of pixel } (i+1, j) \text{ and pixel } (i+1, j+1) \]
Midpoint line algorithm

Set this pixel
Advance to the next sample point

\[ i \quad i + 1 \]

\[ j \quad j + 1 \]
Midpoint line algorithm

Recall the basic algorithm:

\[
i = x_0 \\
j = y_0 \\
while (i <= x_1) \\
\{ \\
\quad \text{setPixel (i, j)} \\
\quad i = i + 1 \\
\quad \text{if (Condition)} \\
\quad \quad j = j+1 \\
\}
\]

Condition() tests if the midpoint \((i+1, j+\frac{1}{2})\) is above or below the line

Need a mathematical expression for Condition()
**Midpoint line algorithm**

Use the implicit representation for the line

- \( F(x, y) = mx + b - y \)

Verify that for positive \( m \), at a given \( x \),

- If \( y \) is on the line, then \( F(x, y) = 0 \)
- If \( y \) is above the line, then \( F(x, y) < 0 \)
- If \( y \) is below the line, then \( F(x, y) > 0 \)

\[ m = \text{slope} = \frac{\Delta y}{\Delta x} \]
Midpoint line algorithm

To determine if the next pixel is (i+1, j) or (i+1, j+1):

- Evaluate
  
  \[ F(i+1, j + \frac{1}{2}) = m(i + 1) + b - (j + \frac{1}{2}) \]

- The midpoint test, i.e., the desired Condition(i, j), is

  \[ F(i+1, j + \frac{1}{2}) < 0 \]
Midpoint line algorithm

The final algorithm:

\[
i = x_0 \\
j = y_0 \\
\text{while } (i \leq x_1) \\
\{
\quad \text{setPixel } (i, j) \\
\quad i = i + 1 \\
\quad F(i, j+ \frac{1}{2}) = m*i + b - (j+ \frac{1}{2}) \\
\quad \text{if } (F(i, j+ \frac{1}{2}) < 0) \\
\quad \quad j = j+1 \\
\}
Midpoint line algorithm

What if line passes exactly through the midpoint?

Convention in most graphics systems:

- If $F(i+1, j + \frac{1}{2}) = 0$, choose the pixel to the right
- Be careful: This will set different pixels for the four different cases or if the line is drawn left-to-right or right-to-left
Bresenham’s Algorithm

An efficient form of the midpoint algorithm

Uses integer math
Assumes both endpoints are at integer pixel locations
Uses incremental arithmetic for each new pixel
Bresenham’s Algorithm

Midpoint test:
\[ F(i, j + \frac{1}{2}) < 0 \]
\[ m*i + b - (j + \frac{1}{2}) < 0 \]

where
\[ m = \frac{y_1 - y_0}{x_1 - x_0} \]
\[ b = \frac{x_1y_0 - x_0y_1}{x_1 - x_0} \]

Multiply through by \(2(x_1 - x_0)\) to get rid of all fractions

Modified midpoint test:
\[ 2(y_1 - y_0)i + 2(x_1y_0 - x_0y_1) - 2(x_1 - x_0)j - (x_1 - x_0) < 0 \]

As long as \(x_1 > x_0\), the modified midpoint test is equivalent to the midpoint test
Bresenham’s Algorithm

Use incremental arithmetic

\[ F(i, j) = 2(y_1 - y_0)i + 2(x_1y_0 - x_0y_1) - 2(x_1 - x_0)j - (x_1 - x_0) \]

If \( F \geq 0 \), the next pixel is \((i+1, j)\)

\[ F(i+1, j) = 2(y_1 - y_0)(i+1) + 2(x_1y_0 - x_0y_1) - 2(x_1 - x_0)j - (x_1 - x_0) \]
\[ = F(i, j) + 2(y_1 - y_0) \]

Else if \( F < 0 \), the next pixel is \((i+1, j+1)\)

\[ F(i+1, j+1) = 2(y_1 - y_0)(i+1) + 2(x_1y_0 - x_0y_1) - 2(x_1 - x_0)(j+1) - (x_1 - x_0) \]
\[ = F(i, j) + 2(y_1 - y_0) - 2(x_1 - x_0) \]
Bresenham’s Algorithm

setPixel \( (x_0, y_0) \)

\[ i = x_0 \]

\[ j = y_0 \]

\[ dy = y_1 - y_0 \]

\[ dx = x_1 - x_0 \]

\[ F = 2*dy*i + 2(x_1y_0 - x_0y_1) - 2*dx*(j +1) \]

while \((i <= x_1)\) {

    setPixel \((i, j)\)

    \[ i = i + 1 \]

    if \((F >= 0)\) {

        \[ F = F + 2dy \]

    } else {

        \[ F = F + 2dy - 2dx \]

        \[ j = j + 1 \]

    }

}
Circle Drawing Algorithms
A circle is...

... the set of all points that are a distance $R$ from a center point $(c_x, c_y)$
Representing circles

Explicit equation
\[ y = c_y + \sqrt{R^2 - (x - c_x)^2} \]

Implicit equation
\[ F(x, y) = R^2 - (x - c_x)^2 - (y - c_y)^2 \]

Parametric equation
\[ x = c_x + R \cos \Theta \]
\[ y = c_y + R \sin \Theta \]
\[ \Theta \in [0, 2\pi] \]
Eight-way symmetry

If the circle is centered at the origin, we can exploit 8-way symmetry.

If a point \((x, y)\) is on the circle, then so are \((y, x)\), \((y, -x)\), \((x, -y)\), \((-x, y)\), \((-y, x)\), \((-y, -x)\), and \((-x, -y)\).
Eight-way symmetry

If the circle is centered at an integer point \((i, j)\)

1. Shift the circle to the origin
2. Determine which points to set using eight-way symmetry
3. Shift the determined pixels back to \((i, j)\)
Polygonal approximation

Approximate the circle with straight lines
Generate points at equal angular increments around the circle and draw straight lines between the points

This general strategy is often used for more complex curves
Tradeoff between accuracy and speed (more lines → more accurate)
Uniform angular sampling

Sample circle at equal angular increments

- Set pixels beneath the sample points
- Use the parametric expression of the circle
- Sample at equal intervals of $\Theta$, $\Theta \in [0, 2\pi]$
  \[
  \begin{align*}
  x &= c_x + R \cos \Theta \\
  y &= c_y + R \sin \Theta
  \end{align*}
  \]

What increments of $\Theta$ should be used?

- If increments are too big, the circle may contain gaps
- If too small, the circle may be chunky and blending artifacts may occur
- Thickness may be uneven
Midpoint circle algorithm

The circle can be divided into some regions where $x$ changes faster than $y$ ...
Midpoint circle algorithm

... and regions where y changes faster than x
Midpoint circle algorithm

We will consider one of these regions
  • The others can be determined by symmetry

We will assume the circle is centered at the origin
  • More general circles are left as an exercise
Midpoint circle algorithm

Consider the explicit equation for this region of the circle:

\[ y = \sqrt{R^2 - x^2}, \ x < y \]

Note that y changes more slowly than x in this region
Midpoint circle algorithm

Approach:

- Set one pixel for each column that the section intersects
- If the current pixel is \((i, j)\), then the pixel in the next column will either be pixel \((i+1, j)\) or \((i+1, j-1)\)
- Similar to the midpoint line algorithm, need to determine which one
Midpoint circle algorithm

Approach:
- Use the implicit form of a circle to determine which point to choose
  \[ F(x, y) = R^2 - x^2 - y^2 \]
- Verify that
  \[ F(x, y) = 0, \text{ for points on the circle} \]
  \[ F(x, y) < 0, \text{ for points outside the circle} \]
  \[ F(x, y) > 0, \text{ for points inside the circle} \]
Midpoint circle algorithm

Consider the point \((i+1, j-\frac{1}{2})\) midway between the candidate pixels

- If the midpoint is inside the circle, choose pixel \((i+1, j)\)
- If the midpoint is outside the circle, choose pixel \((i+1, j-1)\)
Midpoint circle algorithm

Consider the point \((i+1, j-\frac{1}{2})\) midway between the candidate pixels

- If the midpoint is inside the circle, choose pixel \((i+1, j)\)
- If the midpoint is outside the circle, choose pixel \((i+1, j-1)\)
Midpoint circle algorithm

Advance to the next point and consider the midpoint
Midpoint circle algorithm
Midpoint circle algorithm

Continue while $x < y$
Midpoint circle algorithm
Midpoint circle algorithm

The midpoint test is $F(i, j - \frac{1}{2}) < 0$, where $F(x, y) = R^2 - x^2 - y^2$

- If the test is false, the midpoint is inside the circle and the next pixel is $(i+1, j)$
- If the test is true, the midpoint is outside the circle and the next pixel is $(i+1, j-1)$

Basic algorithm:

```plaintext
i = 0
j = round (R)
while (i <= j)
{
    setPixel (i, j)
    i = i + 1
    $F(i, j - \frac{1}{2}) = R^2 - i^2 - (j + \frac{1}{2})^2$
    if ($F(i, j + \frac{1}{2}) < 0$)
        j = j-1
}
```
More general 2D drawing

Midpoint algorithms can be used for drawing ellipses

General curves
- Represent the curve parametrically and sample at equally spaced intervals
- Approximate the curve by a set of straight lines and draw the lines
Summary

2D rendering pipeline

Line drawing algorithms, esp.
  Digital differential analyzer (DDA)
  Midpoint line algorithm
  Bresenham’s algorithm

Circle drawing algorithms, esp.
  Uniform angular sampling
  Midpoint circle algorithm