Rendering approaches

1. image-oriented
   foreach pixel ...

2. object-oriented
   foreach object ...

geometry → 3D rendering pipeline → image
**3D graphics pipeline**

**Vertex processing:** coordinate transformations and color

**Clipping and primitive assembly:** output is a set of primitives

**Rasterization:** output is a set of fragments for each primitive

**Fragment processing:** update pixels in the frame buffer
Graphics Pipeline
(slides courtesy K. Fatahalian)
Vertex processing

Vertices are transformed into “screen space”
Vertex processing

Vertices are transformed into “screen space”

Vertices

EACH VERTEX IS TRANSFORMED INDEPENDENTLY
Primitive processing

Then organized into primitives that are clipped and culled…
Rasterization

Primitives are rasterized into “pixel fragments”
Rasterization

Primitives are rasterized into “pixel fragments”

EACH PRIMITIVE IS RASTERIZED INDEPENDENTLY
Fragment processing

Fragments are shaded to compute a color at each pixel
Fragment processing

Fragments are shaded to compute a color at each pixel

EACH FRAGMENT IS PROCESSED INDEPENDENTLY
Pixel operations

Fragments are blended into the frame buffer at their pixel locations (z-buffer determines visibility)
Pipeline entities

Vertices

Primitives

Fragments

Fragments (shaded)

Pixels
Rasterization
What is rasterization?

Rasterization is the process of determining which pixels are “covered” by the primitive.
What is rasterization?

input: primitives   output: fragments
enumerate the pixels covered by a primitive
interpolate attributes across the primitive
Rasterization

Compute integer coordinates for pixels covered by the 2D primitives

Algorithms are invoked many, many times and so must be efficient

Output should be visually pleasing, for example, lines should have constant density

Obviously, should be able to draw all possible 2D primitives
Screen coordinates
Line Representation
Math Review

• 2D math for lines

How do we determine the equation of the line?
Math Review

• 2D math for lines

Slope-Intercept formula for a line

\[
\text{Slope} = \frac{(Y2 - Y1)}{(X2 - X1)} \quad \frac{(Y - Y1)}{(X - X1)}
\]

Solving For Y

\[
Y = \left[\frac{(Y2 - Y1)}{(X2 - X1)}\right]X \\
+ \left[-\frac{(Y2 - Y1)}{(X2 - X1)}\right]X1 + Y1 \text{ or }
\]

\[
Y = mX + b
\]
Math Review

• Explicit (functional) representation
  \[ y = f(x) \]

  \[ y \] is the dependent, \[ x \] independent variable

Find value of \( y \) from value of \( x \)

Example, for a line: for a circle:
\[ y = mx + b \] \[ x^2 + y^2 = r^2 \]
Math Review

• Parametric Representation

\[ x = x(u), \quad y = y(u) \]

where new parameter \( u \) (or often \( t \)) determines the value of \( x \) and \( y \) (and possibly \( z \)) for each point

\( x, y \) treated the same, axis invariant
Math Review

Parametric formula for a line

\[ X = X_1 + t(X_2 - X_1) \]
\[ Y = Y_1 + t(Y_2 - Y_1) \]

for parameter \( t \) from 0 to 1

Therefore, when

\[ t = 0 \] we get \((X_1, Y_1)\)
\[ t = 1 \] we get \((X_2, Y_2)\)

Varying \( t \) gives the points along the line segment
Implicit Line Equation

\[ f(X) = N \cdot (X - X_0) = 0 \]

\[ X_0 = (x_0, y_0) \]
Implicit Line Equation

$$f(X) = N \cdot (X - X_0) = d$$

decision variable, \(d\)

- \(d > 0\)
- \(d < 0\)
- \(d = 0\)

\(X_0 = (x_0, y_0)\)
Implicit Line Equation

Decision variable, \( d \)

\[
f(X) = N \cdot (X - X_0) = d
\]

- \( d > 0 \)
- \( d < 0 \)
- \( d = 0 \)
Implicit Line Equation

\[ f(X) = N \cdot (X - X_0) = d \]

decision variable, \( d \)

\[ d > 0 \]
\[ d < 0 \]
\[ d = 0 \]
Implicit Line Equation

Decision variable, $d$

$$f(X) = N \cdot (X - X_0) = d$$

- $d > 0$
- $d < 0$
- $d = 0$
Line Drawing
Which pixels should be used to approximate a line?

Draw the thinnest possible line that has no gaps
DDA algorithm for lines

Parametric Lines: the DDA algorithm
(digital differential analyzer)

\[ Y_{i+1} = m \times x_{i+1} + B \]

\[ = m(x_i + \Delta x) + B \quad \Delta x = (x_{i+1} - x_i) \]

\[ = y_i + m(\Delta x) \quad \text{<- must round to find int} \]

If we increment by 1 pixel in X, we turn on
[\[x_i, \text{Round}(y_i)\]] or same for Y if \( m > 1 \)
Scan conversion for lines

DDA includes Round(); and this is fairly slow

For Fast Lines, we want to do only integer math +,-

We do this using the **Midpoint Algorithm**

To do this, let's look at lines with y-intercept B and with slope between 0 and 1:

\[ y = \frac{dy}{dx}x + B \quad \Rightarrow \quad f(x,y) = \left( \frac{dy}{dx} \right)x - (dx)y + B(dx) = 0 \]

Removes the division => slope treated as 2 integers
Line drawing algorithm
(case: $0 < m \leq 1$)

\[ y = y_0 \]
for \( x = x_0 \) to \( x_1 \) do
\[ \text{draw}(x,y) \]
\[ \text{if } (<\text{condition}> \text{)} \text{ then } y = y + 1 \]

- move from left to right
- choose between \((x+1,y)\) and \((x+1,y+1)\)
Line drawing algorithm

(case: $0 < m \leq 1$)

$y = y_0$
for $x = x_0$ to $x_1$ do
draw($x, y$)
if ($<$condition$>$) then
  $y = y + 1$

• move from left to right
• choose between
  $(x+1, y)$ and $(x+1, y+1)$
Use the midpoint between the two pixels to choose
Use the midpoint between the two pixels to choose
Use the midpoint between the two pixels to choose
Use the midpoint between the two pixels to choose.

Implicit line equation:

\[ f(X) = N \cdot (X - X_0) = 0 \]

Evaluate \( f \) at midpoint:

\[ f(x, y + \frac{1}{2}) \neq 0 \]
Use the midpoint between the two pixels to choose

Implicit line equation:

\[ f(\mathbf{X}) = \mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = 0 \]

Evaluate \( f \) at midpoint:

\[ f(x, y + \frac{1}{2}) > 0 \]
Line drawing algorithm

(case: $0 < m \leq 1$)

$$y = y_0$$

for $x = x_0$ to $x_1$ do

\[
draw(x, y)
\]

if \( f(x + 1, y + \frac{1}{2}) < 0 \) then

$$y = y + 1$$
We can make the Midpoint Algorithm more efficient

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x,y) \\
\quad \text{if } \left( f(x + 1, y + \frac{1}{2}) < 0 \right) \text{ then} \\
\quad \quad y = y + 1
\]
We can make the Midpoint Algorithm more efficient by making it incremental!

\[ f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0 \]

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
We can make the Midpoint Algorithm more efficient

\[ f(x + 1, y + \frac{1}{2}) > 0 \]

\[ f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0 \]

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
We can make the Midpoint Algorithm more efficient

\[ f(x + 1, y + \frac{1}{2}) < 0 \]

\[ f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0 \]

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
We can make the Midpoint Algorithm more efficient

\[ y = y_0 \]
\[ d = f(x_0+1, y_0+1/2) \]
\[ \text{for } x = x_0 \text{ to } x_1 \text{ do} \]
\[ \text{draw}(x, y) \]
\[ \text{if } (d < 0) \text{ then} \]
\[ y = y + 1 \]
\[ d = d + (y_0 - y_1) + (x_1 - x_0) \]
\[ \text{else} \]
\[ d = d + (y_0 - y_1) \]

\[ f(x + 1, y) = f(x, y) + (y_0 - y_1) \]
\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]
Adapt Midpoint Algorithm for other cases

\[ 0 < m \leq 1 \]
Adapt Midpoint Algorithm for other cases

case: \(-1 \leq m < 0\)
Adapt Midpoint Algorithm for other cases

Case: \( l \leq m \) or \( m \leq -l \)
Line drawing references

• the algorithm we just described is the *Midpoint Algorithm* (Pitteway, 1967), (van Aken and Novak, 1985)

• draws the same lines as the *Bresenham Line Algorithm* (Bresenham, 1965)
Triangles
barycentric coordinates
barycentric coordinates
barycentric coordinates

\[ \gamma = 0 \]

\[ \gamma = 1 \]

\[ \gamma = 2 \]

\[ \gamma = -1 \]
barycentric coordinates

\[ \beta = -1 \quad \beta = 0 \quad \beta = 1 \]
barycentric coordinates
barycentric coordinates

\[ p = \alpha a + \beta b + \gamma c \]

What are \((\alpha, \beta, \gamma)\) ?

<whiteboard>