Rendering approaches

1. **image-oriented**
   - foreach pixel ...

2. **object-oriented**
   - foreach object ...

**object-oriented rendering**
- e.g., hardware: OpenGL graphics pipeline, Direct3D
- software: Renderman (REYES)
  - task: figure out where a point in the geometry will land on the final image pixels
3D graphics pipeline

1. **Geometry**: objects – made of primitives – made of vertices
2. **Vertex processing**: coordinate transformations and color
3. **Clipping and primitive assembly**: use clipping volume. must be primitive by primitive rather than vertex by vertex. therefore vertices must be assembled into primitives before clipping can take place. Output is a set of primitives.
4. **Rasterization**: primitives are still in terms of vertices -- must be converted to pixels. E.g., for a triangle specified by 3 vertices, the rasterizer must figure out which pixels in the frame buffer fill the triangle. Output is a set of fragments for each primitive. A fragment is like a potential pixel. Fragments can carry depth information used to figure out if they lie behind other fragments for a given pixel.
5. **Fragment processing**: update pixels in the frame buffer. some fragments may not be visible. texture mapping and bump mapping. blending.
Graphics Pipeline
(slides courtesy K. Fatahalian)
Vertex processing

Vertices are transformed into “screen space”
Vertex processing

Vertices are transformed into “screen space”

EACH VERTEX IS TRANSFORMED INDEPENDENTLY
Primitive processing

Then organized into primitives that are clipped and culled...
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

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels

Shaded fragments
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

- **output** 1 fragment per pixel covered by 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
we’ll assume stuff has been converted to screen coordinates
Line Representation
Math Review

• 2D math for lines

How do we determine the equation of the line?

(X1, Y1)

(X2, Y2)
Math Review

• 2D math for lines
  
  Slope-Intercept formula for a line

  \[
  \text{Slope} = \frac{(Y_2 - Y_1)}{(X_2 - X_1)} \quad \text{or} \quad \frac{(Y - Y_1)}{(X - X_1)}
  \]

  Solving For Y

  \[
  Y = \left[\frac{(Y_2 - Y_1)}{(X_2 - X_1)}\right]X \\
  + \left[\frac{-(Y_2 - Y_1)}{(X_2 - X_1)}\right]X_1 + Y_1 \quad \text{or} \quad Y = mX + b
  \]
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:
\[ y = mx + b \quad \text{for} \quad x^2 + y^2 = r^2 \]
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

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

\[ X_0 = (x_0, y_0) \]

<whiteboard>: work out the implicit line equation in terms of X0 and X1
Implicit Line Equation

decision variable, \( d \)

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

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

\( X_0 = (x_0, y_0) \)

<whiteboard>: work out the implicit line equation in terms of \( X_0 \) and \( X_1 \)
Implicit Line Equation

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

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

\[ \mathbf{X}_0 = (x_0, y_0) \]

<whiteboard>: work out the implicit line equation in terms of \( \mathbf{X}_0 \) and \( \mathbf{X}_1 \)
Implicit Line Equation

Decision variable, \( d \)

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

\[
\begin{align*}
&\text{if } d > 0 \\
&\text{if } d < 0 \\
&\text{if } d = 0
\end{align*}
\]

\( X_0 = (x_0, y_0) \)

<whiteboard>: work out the implicit line equation in terms of \( X_0 \) and \( X_1 \)
Implicit Line Equation

decision variable, $d$

$$f(\mathbf{X}) = \mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = d$$

- $d > 0$
- $d < 0$
- $d = 0$

<whiteboard>: work out the implicit line equation in terms of $X_0$ and $X_1$
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)

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

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

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

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:

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

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

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

• move from left to right
• choose between
  $(x+1,y)$ and $(x+1,y+1)$

draw pixels from left to right, occasionally move up
Line drawing algorithm

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

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

- move from left to right
- choose between $(x + 1, y)$ and $(x + 1, y + 1)$

draw pixels from left to right, occasionally move up
Use the midpoint between the two pixels to choose

If the line falls below the midpoint, use the bottom pixel
If the line falls above the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

If the line falls **below** the midpoint, use the bottom pixel
if the line falls **above** the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

If the line falls below the midpoint, use the bottom pixel
if the line falls above the midpoint, use the top pixel
Use the midpoint between the two pixels to choose

\[ \mathbf{X}_0 = (x_0, y_0) \]

Implicit line equation:

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

Evaluate \( f \) at midpoint:

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

\(<\text{whiteboard}>\): work out the implicit line equation in terms of \( \mathbf{X}_0 \) and \( \mathbf{X}_1 \)

Question: will \( f(x, y + 1/2) \) be > 0 or < 0?
Use the midpoint between the two pixels to choose

Implicit line equation:

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

Evaluate \( f \) at midpoint:

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

This means midpoint is above the line \( \Rightarrow \) line is closer to bottom pixel
Line drawing algorithm
(case: $0 < m \leq 1$)

```
y = y0
for x = x0 to x1 do
    draw(x,y)
    if $(f(x + 1, y + \frac{1}{2}) < 0)$ then
        y = y + 1
```

can now fill in the **condition**
We can make the Midpoint Algorithm more efficient

\[
y = y_0 \\
\text{for } x = x_0 \text{ to } x_1 \text{ do} \\
\quad \text{draw}(x,y) \\
\quad \text{if } (f(x + 1, y + \frac{1}{2}) < 0) \text{ then} \\
\quad \quad y = y + 1
\]

in each iteration we draw the current pixel and we evaluate the line equation at the next midpoint halfway above the current pixel
We can make the Midpoint Algorithm more efficient by making it incremental!

Assume we have drawn the last red pixel and evaluated the line equation at the next (Red) midpoint. There are two possible outcomes:
1. we will choose the bottom pixel. In this case the next midpoint will be at the same level \((x + 1, y)\)
2. we will choose the top pixel. In this case the next midpoint will be one level up \((x + 1, y + 1)\)

The line equation at these next midpoints can be evaluated incrementally using the update formulas shown.

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

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

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

As we move over one pixel to the right, we will choose either \((x+1,y)\) (yellow) or \((x+1,y+1)\) (pink) and the next midpoint we will evaluate will be either

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

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

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

\[ f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0) \]

As we move over one pixel to the right, we will choose either \((x+1,y)\) (yellow) or \((x+1,y+1)\) (pink) and the next midpoint we will evaluate will be either
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

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

The algorithm is incremental and uses only integer arithmetic.
Adapt Midpoint Algorithm for other cases

case: $0 < m \leq 1$
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

\[ \gamma = 2 \]
\[ \gamma = 1 \]
\[ \gamma = 0 \]
\[ \gamma = -1 \]
barycentric coordinates

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

\[ c - a \quad b - a \]

\[ \text{a} \quad \text{b} \quad \text{c} \]
barycentric coordinates
barycentric coordinates

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

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

<whiteboard>