Scan Conversion

Warning: Algebra Ahead

- This will be more pleasant if you follow along
- Following along will make your first homework easier
- Tips for following along:
  - Participate
  - Stop me if I go too fast

Outline for today
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1

Why study scan conversion?
- Hardware does this for me, why should I care?
- Need to understand the whole graphics pipeline for effective GL coding/debugging
- Scan conversion is fundamental to many image-processing algorithms
- Math demonstrates important optimization concepts

What is scan conversion?
- CG objects divided into 2D primitives
- To put 2D primitives on the monitor, we need to turn on the right pixels
- Scan conversion is the process of turning primitives into pixels

Outline for today
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1

Shape you want to draw
2D primitives
Pixels in framebuffer
Pixels on display

Outline for today
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Anti-aliasing
- Project 1

Scan converting lines
- All we have to work with is:
  setPixel(int x, int y, int color);
- Implement the routine:
  void draw_me_a_pretty_line(int x1, int y1, int x2, int y2, int color);

Scan converting lines: take one
void scLine(int x1, int y1, int x2, int y2, int color) {
  // compute the y=mx+b equation for the line
  double dy, dx, y, m, b;
  int x;
  dx = x2 - x1;
  dy = y2 - y1;
  m = dy / dx;
  b = y1 – m*x1;
  // loop over each x value
  for (x = x1; x <= x2; x++) {
    // compute the corresponding y value
    y = m*x+b;
    setPixel(x, round(y), color);
  }
}

A quick fix
Draw a line from (1,1) to (5,6)
dx = 4; dy = 5;
m = 1.25; b = -0.25;
for (int y = 1; y<=6; y++) {
  double x = (y-b)/m;
  setPixel(round(x), y, color);
}

What’s wrong with this approach?
(1,1)
(2,2)
(3,3)
(3,4)
(4,5)
(5,6)

What’s still wrong with this approach?
- Performance: floating-point multiplies are a graphics programmer’s worst enemy
- Doesn’t generalize well to other shapes
- Also has subtle roundoff problems if you happen to walk exactly between two rows or columns

What’s still wrong with this approach?
(1,1)
(2,2)
(3,3)
(3,4)
(4,5)
(5,6)
The real deal: Bresenham’s Algorithm
- Uses only integer calculations
- Adapts nicely to other primitives

Bresenham’s Algorithm: Setup
- Assume we’re drawing a line with positive slope less than 1
- Assume we’ve decided to draw the kth pixel on our line at \((x_k, y_k)\)
- We’re going to step horizontally

Bresenham’s Algorithm: Iterating
- The “real” y value at \(x_{k+1}\) is \(m(x_{k+1}) + b\)
- We know that \(0 < \text{slope} < 1\), so our only choices are \(y_k\) and \(y_k+1\)
- Compute the distance from the “real line” to each of our two choices

Bresenham’s Algorithm: Iterating
- What does the boolean value \((d_1 - d_2 > 0)\) tell me?
  - \(d_1 - d_2 > 0 \rightarrow d_1 > d_2 \rightarrow y_{k+1}\) is closer than \(y_k\)

Bresenham’s Algorithm: Iterating
- So the value \(d_1 - d_2\) tells me whether to pick \(y_k\) or \(y_{k+1}\)
- Then I can just move on to the next pixel, starting the algorithm again with \((x_{k+1}, y_k)\) or \((x_{k+1}, y_{k+1})\)

What’s wrong with this approach?
\[
d_1 = y - y_k = m(x_{k+1}) + b - y_k
\]
\[
d_2 = (y_k + 1) - y = (y_{k+1}) - m(x_{k+1}) - b
\]
Floating-point multiplication still kills us...
Bresenham’s Algorithm: Optimizing

- Want to know whether \( d_1 - d_2 > 0 \)
- \( d_1 = y - y_k = m(x_k+1) + b - y_k \)
- \( d_2 = (y_{k+1}) - y = (y_{k+1}) - m(x_{k+1}) - b \)
- \( d_1 - d_2 = 2m(x_{k+1}) - 2y_k + 2b - 1 \)
- \( m = \frac{dy}{dx} \)
- \( p_k = dx(d_1-d_2) \) \([p_k \text{ is our “decision variable”}]\)
- \( p_k = dx(2m(x_{k+1}) - 2y_k + 2b - 1) \)
- \( p_k = 2dy \times x_k - 2dx \times y_k + 2dy + dx(2b-1) \)

if \( p_k < 0 \) set the lower pixel
else set the upper pixel.

Bresenham’s Algorithm: One More Step

\[ p_k = 2dy \times x_k - 2dx \times y_k + c \]
- How many multiplies at each pixel?
- Can we do better?

\[ p_{k+1} = 2dy \times x_{k+1} - 2dx \times y_{k+1} + c \]

\[ p_{k+1} - p_k = 2dy(x_{k+1} - x_k) - 2dx(y_{k+1} - y_k) \]

\[ p_{k+1} = p_k + 2dy - 2dx(y_{k+1} - y_k) \]

\[ p_{k+1} = p_k + 2dy \text{ (if } p_k < 0 \text{) or } p_k + 2dy - 2dx \text{ (if } p_k > 0 \text{)} \]

- If we plug \((x_0,y_0)\) into the \( p_k \) equation, we get our starting value:

\[ p_0 = 2dy - dx \]

Bresenham’s Algorithm: Summary

*Bresenham’s Line-Drawing Algorithm for 0 <= m < 1*
1. Input two endpoints, store left endpoint as \((x_0, y_0)\).
2. Turn on initial point: setPixel\((x_0,y_0,\text{color})\).
3. Calculate constants \(dx, dy, 2dy\) and \(2dy - 2dx\).
4. Calculate starting value of decision parameter:

\[ p_0 = 2dy - dx \]

for\((k=0; k<=x_1-x_0; k++)\)

- if \(p_k < 0\)
  - setPixel\((x_{k+1}, y_{k+1}, \text{color})\)
  - \(y_{k+1} = y_k\)
- else
  - setPixel\((x_{k+1}, y_{k+1} + 1, \text{color})\)
  - \(y_{k+1} = y_k + 1\)

Bresenham’s Algorithm: Other Cases

- Negative slope: 0 > m > -1
  - Change one sign on the previous slide
  - \(x_0 > x_1\)
    - Swap \(\text{______ and ______}\)
  - \(|dy| > |dx|\)
    - Iterate \(\text{______ ______} \) instead of \(\text{______ ______} \)

SIGGRAPH video break

- Five-minute break to introduce you to what’s going on in research graphics in 2005
- Slight bias toward Stanford projects
- This week:
  - High Performance Imaging Using Large Camera Arrays, Wilburn et al, SIGGRAPH 2005

Outline for today

- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1
Scan-converting Circles
- All we have to work with is:
  `setPixel(int x, int y, int color);`
- Implement the routine:
  `void draw_me_a_circle(int xc, int yc, int radius, int color);`

### Pythagorean theorem
- 
  \[(x-x)^2 + (y-y)^2 = r^2\]

### Solving for \(y\)
- 
  \[y = y_c \pm \sqrt{r^2 - (x_c - x)^2}\]

```c
for(int x = xc - r; x <= xc + r; x++) {
    int dy = sqrt(r^2 - (x_c - x)^2);
    setPixel(x,y+dy,color);
    setPixel(x,y-dy,color);
}
```

### Taking advantage of circular symmetry
- If point \((x,y)\) is on the circle, what other points must be on the circle?

### Bresenham's Algorithm for Circles
- Start in the octant just above the x-axis, walk ccwise around the circle
- Red pixel is turned on, which pixels **could** be turned on next?

### Scan-converting circles, take one
- Pythagorean theorem tells us:
  \[(x-x)^2 + (y-y)^2 = r^2\]
- We can solve this for \(y\):
  \[y = y_c \pm \sqrt{r^2 - (x_c - x)^2}\]

```c
for(int x = xc - r; x <= xc + r; x++) {
    int dy = sqrt(r^2 - (x_c - x)^2);
    setPixel(x,y+dy,color);
    setPixel(x,y-dy,color);
}
```

### Scan-converting circles, take two
- Hint: pretend center is at the origin, except when you call `setPixel(…)`
  (today's notation assumes this)

```c
for(int x = ___; x <= ___ ; x++) {
    int y = sqrt(r^2 - x^2);
    setPixel(__,__,color);
    setPixel(__,__,color);
    setPixel(__,__,color);
    setPixel(__,__,color);
    ...
}
```

### What's still wrong with this approach?

### Scan-converting circles, take three
- Just like we did for lines, let’s compute the distance from each point to the “real” circle
- Call the red point \((x_k,y_k)\)
- We want to find \(x_{k+1}\) and \(y_{k+1}\)
- What’s \(y_{k+1}\)?

\[
d_1 = x_{true}^2 - (x_k+1)^2 = r^2 - (y_k + 1)^2 - (x_k - 1)^2
\]
\[
d_2 = x_k^2 - x_{true}^2 = x_k^2 - r^2 + (y_k + 1)^2
\]
Which point is closer?

- Define a decision parameter: \( p_k = d_2 - d_1 \)
- If \( p_k > 0 \), point 1 (the left point) is closer
- Terminology: \( p_k \) helps us choose \( x_{k+1} \)

\[ p_k = d_2 - d_1 = 2(y_{k+1})^2 + x_k^2 + (x_{k+1} - 1)^2 - 2r^2 \]

- This is still a little nasty... what trick did we do next for lines?

Moving right along

- What's \( p_{k+1} \)?

\[ p_{k+1} = 2(y_{k+1} + 1)^2 + (x_{k+1})^2 + (x_{k+1} - 1)^2 - 2r^2 \]

- How can \( p_k \) help us find \( p_{k+1} \)?
- Let's compute \( p_{k+1} - p_k \)
- Skipping all the algebra, we get:

\[ p_{k+1} = p_k + 4\cdot y_k + 6 + 2(x_{k+1}^2 - x_k^2) - 2(x_{k+1} - x_k) \]

- What are the possible values for the terms in parentheses?

Moving right along

\[ p_{k+1} = p_k + 4\cdot y_k + 6 + 2(x_{k+1}^2 - x_k^2) - 2(x_{k+1} - x_k) \]

- if we chose the pixel on the right (\( p_k < 0 \))

\[ x_{k+1} = x_k \]

\[ p_{k+1} = p_k + 4\cdot y_k + 6 \]

- if we chose the pixel on the left (\( p_k > 0 \))

\[ x_{k+1} = x_k - 1 \]

\[ p_{k+1} = p_k + 4(y_k - x_k) + 10 \]

What about the other seven quadrants?

Bresenham's algorithm for circles (just the first octant)

\[ \text{drawCircle}(\text{int } x_c, \text{ int } y_c, \text{ int } r, \text{ int } \text{color}) \]

\[ x = \_\_\_ \_; \]

\[ y = \_\_\_\_; \]

\[ p = \_\_\_\_; \]

for (int \( y = \_\_\_\_; y < \_\_\_\_; y++)

setPixel(x+x_c, y+y_c, color)

if (p < 0)

\[ \_\_\_\_\_ \]

do what?

else

\[ \_\_\_\_\_ \]

do what?

Relax, that's all the algebra for today…
Outline for today
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1

Floodfilling
- Change the color of all the pixels that are the same color as some “seed” pixel
- Like the “paint bucket” tool

Floodfill, take one
- The color the user clicked on is inside_color
- The color the user is “dumping” is new_color

void FloodFill(int x, int y, int inside_color, int new_color) {
  if (GetPixel(x, y) == inside_color) {
    SetPixel(x, y, new_color);
    FloodFill(x-1, y, inside_color, new_color);
    FloodFill(x+1, y, inside_color, new_color);
    FloodFill(x, y+1, inside_color, new_color);
    FloodFill(x, y-1, inside_color, new_color);
  }
}

What’s wrong with this approach?

Floodfill, take two: fill in “runs”
- Turn on the seed
- Fill as far left and right as you can from the seed
- Scan the row above and below to look for runs
- Queue up the rightmost pixel of each new run

Outline for today
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1

Terminology
- 2 pixels are 4-connected if they are adjacent horizontally or vertically
- 2 pixels are 8-connected if they are adjacent horizontally, vertically, or diagonally

Pixel 2 is 4-connected and 8-connected to the red pixel
Pixel 1 is 8-connected to the red pixel

This is the seed

What’s wrong with this approach?
The Jaggies

Wouldn’t "the jaggies" be a good name for a sitcom about a family of monsters?

- Discrete pixels can’t capture primitives perfectly
- A scan-converted primitive:

  Looks jagged if pixels are relatively large:

Solution 1: Buy Better Hardware

- Aliasing is less visible if the pixels get smaller
- Downside: more resolution costs more $$
- Downside: more pixels take more time to scan-convert

Solution 2: Antialiasing

- Human eye is good at seeing sharp edges
- Blurring primitive edges reduces the visibility of the jaggies
- Use all our extra intensity resolution to compensate for limited spatial resolution

Antialiasing: Prefiltering

- Shade each pixel based on how much of it overlaps a primitive
- For lines: assume one pixel width

Antialiasing: Supersampling

- Scan-convert to a virtual display with lots of pixels
- Real pixels are averages of nearby “supersamples”

Antialiasing: What does this mean to me?

- OpenGL supports antialiasing, but it’s not free
  - Can be slow
  - Can require complicated sorting of your primitives
- One major task of any graphics programmer is to evaluate beauty vs. performance

Image: Marc Levoy, 2000
Project 1 Overview: MiniPaint
- You’re the rasterizer
- You’re given setPixel(…) and getPixel(…) routines
- You need to implement:
  - Bresenham for lines
  - Bresenham for circles
  - Filled axis-aligned rectangles
  - Floodfill
  - "Airbrush"
- Also an intro to GLUT

Next time
- Intro to OpenGL
- Windows and clipping