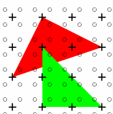
Scan Conversion



CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
June 23, 2005

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

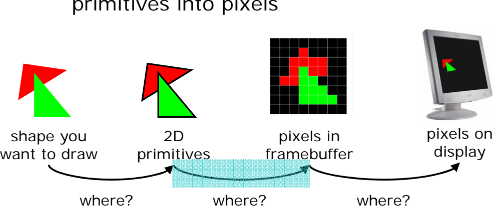
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1

Outline for today

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

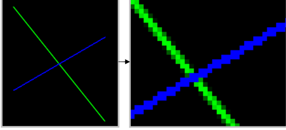
What is scan conversion?

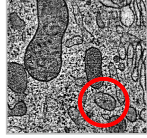
- CG objects divided into 2D primitives
- To put 2D primitives on the monitor, we need to turn on the right pixels
- Scan conversion is the process of turning primitives into pixels



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



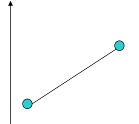


Outline for today

- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- 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);
  }
}
```

What's wrong with this approach?

```
Draw a line from (1,1) to (5,6)

dx = 4; dy = 5;
m =  1.25;  b = -0.25;
for (int x = 1; x<=5; x++) {
 double y = mx+b;
 setPixel(x, round(y), color);
}
```

(1,1)
(2,2)
(3,4)
(4,5)
(5,6)

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);
}
```

(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

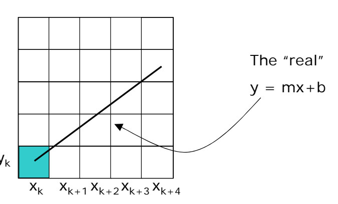
The real deal: Bresenham's Algorithm

- Uses only integer calculations
- Adapts nicely to other primitives

(unnecessary picture of Jack Bresenham to make you feel more emotionally attached to his algorithm)

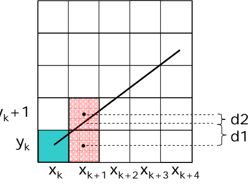
Bresenham's Algorithm: Setup

- 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



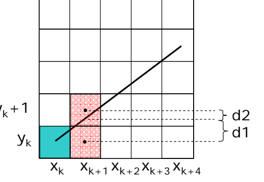
Bresenham's Algorithm: Iterating

- The "real" y value at x_{k+1} is $m(x_{k+1})+b$
- We know that 0 < 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



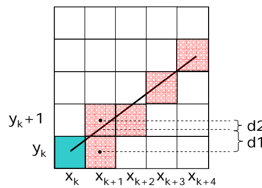
Bresenham's Algorithm: Iterating

- What does the boolean value (d1-d2>0) tell me?
- d1–d2 > 0 --> d1 > d2 -->
  y_k+1 is closer than y_k



Bresenham's Algorithm: Iterating

- So the value d1-d2 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?

$d1 = y - y_k = m(x_k+1) + b - y_k$

$d2 = (y_k+1) - y = (y_k+1) - m(x_k+1) - b$

Floating-point multiplication still kills us...

Bresenham's Algorithm: Optimizing

- Want to know whether $d1 – d2 > 0$

$d1 = y - y_k = m(x_k+1) + b - y_k$
$d2 = (y_k+1) - y = (y_k+1) - m(x_k+1) - b$
$d1 - d2 = 2m(x_k+1) - 2y_k + 2b – 1$
$m = dy/dx$
$p_k = dx(d1-d2)$ [p_k is our "decision variable"]
$p_k = dx(2m(x_k+1) - 2y_k + 2b - 1)$
$p_k = 2dy * x_k - 2dx * y_k + 2dy + dx(2b-1)$
$p_k = 2dy * x_k - 2dx * y_k + c$

if pk < 0 set the lower pixel
else set the upper pixel.

Bresenham's Algorithm: One More Step

$p_k = 2dy * x_k - 2dx * y_k + c$

- How many multiplies at each pixel?
- Can we do better?

$p_{k+1} = 2dy * x_{k+1} – 2dx * y_{k+1} + c$
$p_{k+1} – p_k = 2dy(x_{k+1} – x_k) – 2dx(y_{k+1}-y_k)$
$p_{k+1} – p_k = 2dy – 2dx(y_{k+1}-y_k)$
$p_{k+1} = p_k + 2dy – 2dx(y_{k+1}-y_k)$
$p_{k+1} = p_k + 2dy$ (if $p_k < 0$) or
$p_k + 2dy – 2dx$ (if $p_k > 0$)

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

$p_0 = 2dy - dx$

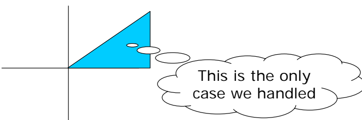
Bresenham's Algorithm: Summary

Bresenham's Line-Drawing Algorithm for 0 <= m < 1

1. Input two endpoints, store left endpoint as (x0 , y0).
2. Turn on initial point: setPixel(x0,y0,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<=x1-x0; k++)
    if (pk < 0)
        setPixel(xk + 1, yk, color)
        pk+1 = pk + 2dy
        yk+1 = yk
    else
        setPixel(xk + 1, yk + 1, color)
        pk+1 = pk + 2dy - 2dx
        yk+1 = yk + 1
```

Bresenham's Algorithm: Other Cases

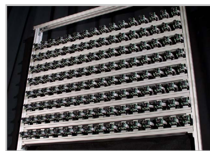


- Negative slope: $0 > m > -1$
  - Change one sign on the previous slide
- x0 > x1
  - Swap _______ and _______
- |dy| > |dx|
  - Iterate ______ ______ instead of ______

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

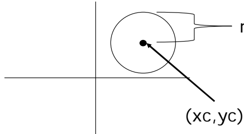


Outline for today

- 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) ;



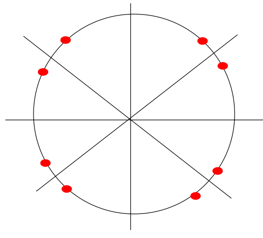
Scan-converting circles, take one

- Pythagorean theorem tells us:
  $(x-x_c)^2 + (y-y_c)^2 = r^2$
- We can solve this for y:
  $y = y_c \pm \mathrm{sqrt}(r^2 - (x_c - x)^2)$

```
for(int x = xc - r; x <= xc + r;
  x++) {
  int dy = sqrt(r^2 - (xc - 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?



Scan-converting circles, take two

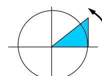
- Hint: pretend center is at the origin, except when you call setPixel(…) (**today's notation assumes this**)

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

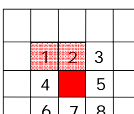
- What's *still* wrong with this approach?

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?



Which point is closer?

- 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}?

$d1 = x_{true}^2 - (x_k-1)^2 = r^2 - (y_k + 1)^2 - (x_k - 1)^2$

$d2 = x_k^2 - x_{true}^2 = x_k^2 - r^2 + (y_k + 1)^2$

d2

d1

Which point is closer?

- Define a decision parameter: p_k = d2 – d1
- If $p_k > 0$, point 1 (the left point) is closer
- Terminology: p_k helps us choose x_{k+1}

$p_k = d2–d1 = 2(y_k+1)^2 + x_k^2 + (x_k-1)^2 – 2r^2$

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

Moving right along

- What's p_{k+1}?

$p_{k+1} = 2(y_{k+1}+1)^2 + (x_{k+1})^2 + (x_{k+1}-1)^2 – 2r^2$
$= 2((y_k+1)+1)^2 + (x_{k+1})^2 + (x_{k+1}-1)^2 – 2r^2$

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

$p_{k+1} = p_k + 4y_k + 6 + 2(x_{k+1}^2 - x_k^2) - 2(x_{k+1}-x_k)$

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

Moving right along

$p_{k+1} = p_k + 4y_k + 6 + 2(x_{k+1}^2 - x_k^2) - 2(x_{k+1}-x_k)$

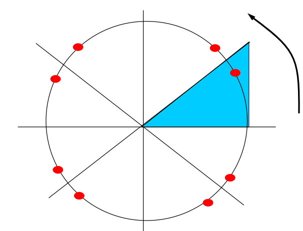
- if we chose the pixel on the right ($p_k < 0$)
  $x_{k+1} == x_k$
  $p_{k+1} = p_k + 4y_k + 6$

- if we chose the pixel on the left ($p_k > 0$)
  $x_{k+1} == x_k – 1$
  $p_{k+1} = p_k + 4(y_k – x_k) + 10$

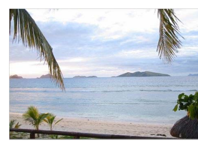
Bresenham's algorithm for circles (just the first octant)

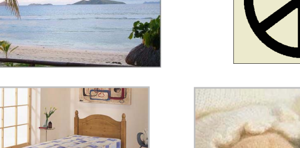
```
drawCircle(int xc, int yc, int r, int color)
   x = __;
   y = __;
   p = __;
   for (int y = __; y < __; y++)
      setPixel(x+xc,y+yc,color)
      if (p < 0)
            do what?
      else
            do what?
```

What about the other seven quadrants?



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

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

	1	2	3	
	4		5	
	6	7	8	

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

Pixel 1 is 8-connected to the red pixel

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"

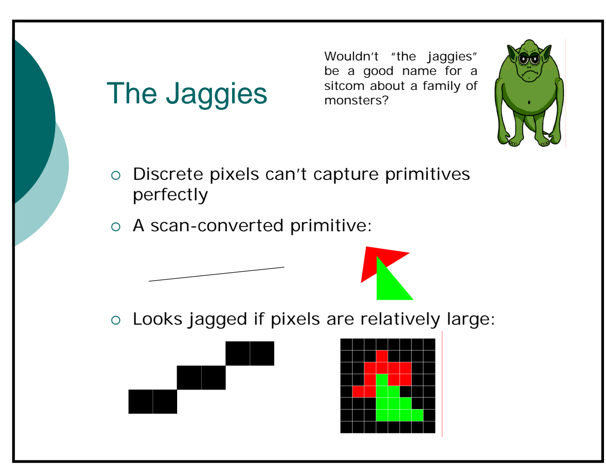
This is the seed

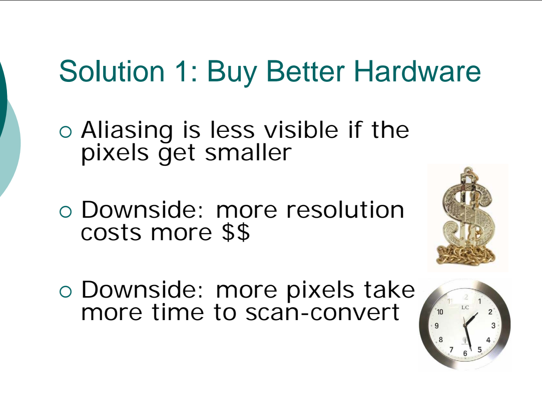
c b s a i e d h g f j

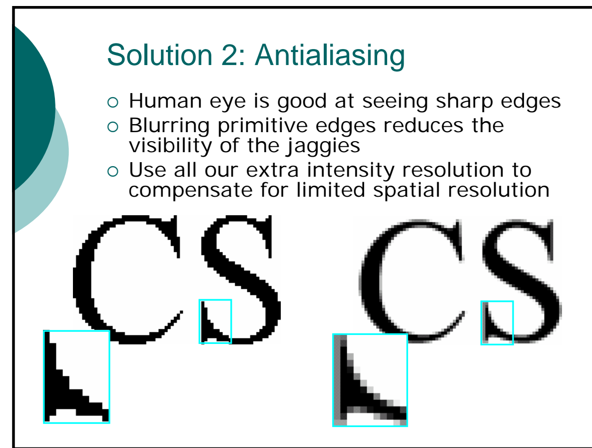
- Turn on the seed
- Fill as far left and right as you can from the seed
- Scan the row above and below to look for runs
- Queue up the rightmost pixel of each new run

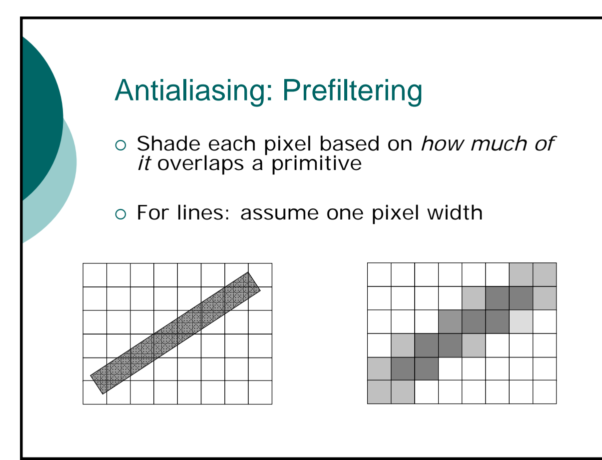
Outline for today

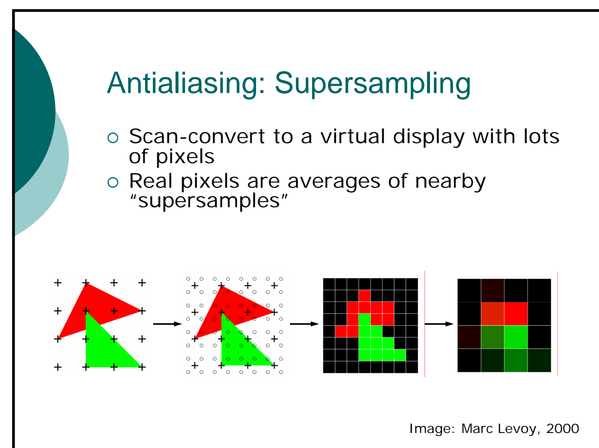
- What is scan conversion?
- Scan-converting lines
- SIGGRAPH video break
- Scan-converting circles
- Floodfilling
- Antialiasing
- Project 1











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

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