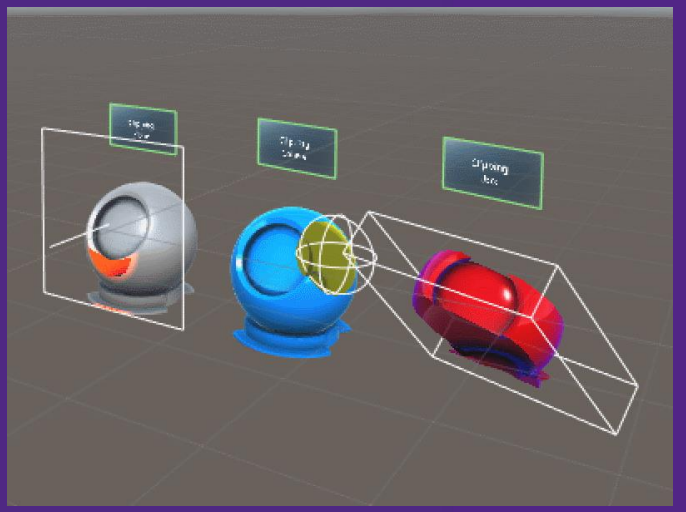
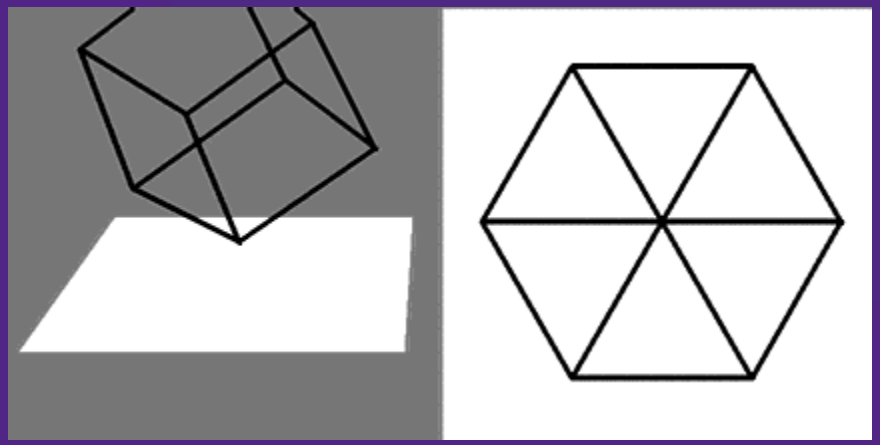
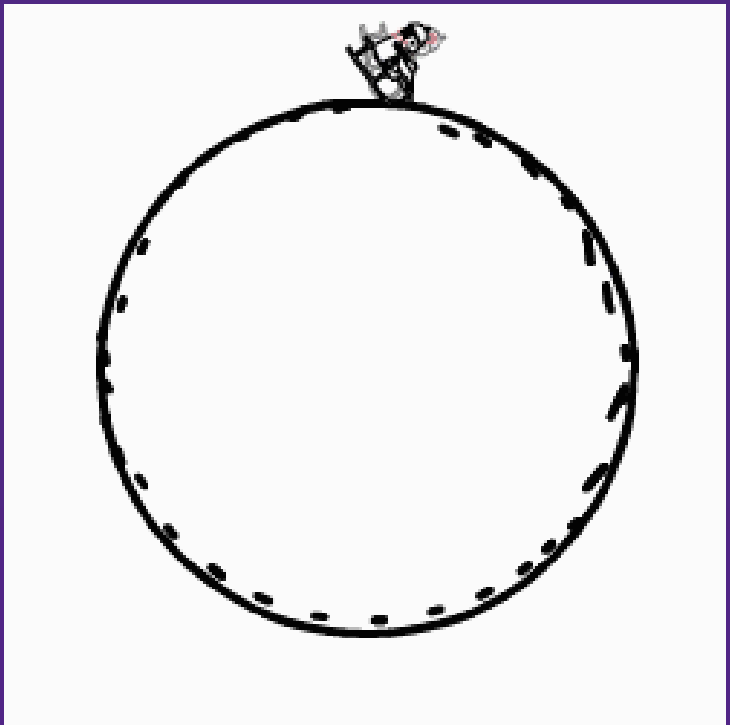
Computer Graphics I:
Lines, Circles, Ellipses, (Oh my!)





Computer Graphics I:
2D Clipping, Rasterization, Lines



```c
#include <GLFW/glfw3.h>

int main(void)
{
    //... glfw init

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 500, "Hello World", NULL, NULL);

    //... glfw stuff

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 640., 0, 500., -1, 1);

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Poll for and process events */
        glfwPollEvents();

        /* Render here */
        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_TRIANGLES);
        glVertex2f(200, 200);
        glVertex2f(300, 200);
        glVertex2f(200, 300);
        glEnd();


        /* Swap front and back buffers */
        glfwSwapBuffers(window);

    }

    glfwTerminate();
    return 0;
}
```

Rasterization

- Rasterization is the process of transforming the vector-defined (floating-point-defined) shapes and graphics primitives into a raster image that can then be drawn pixel-by-pixel.

- Recall that the viewport transformation maps NDC to the “canvas” on which we will actually draw. The viewport canvas is the subset of the window in which we are drawing.

- However, even after applying the viewport transformation, all of our primitives are still defined using vertices with floating-point numbers and symbolic formulas
- e.g. a line segment between two points is

$$y - y_0 = \frac{y_1 - y_0}{x_1 - y_0} x - x_0$$

Rasterization

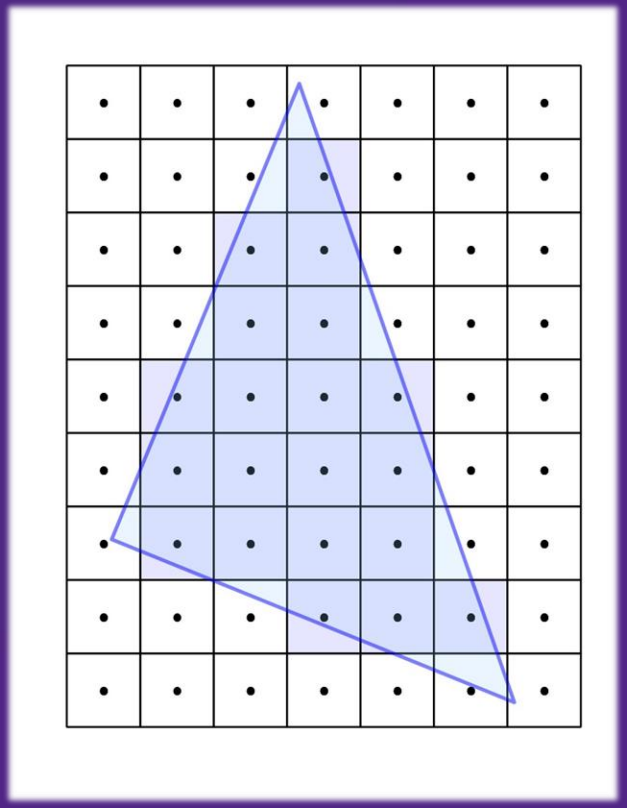
- The process of rasterizing is the transformation of converting that symbolic and continuous data to a discrete set of pixels. In fact, more than pixels. The process of rasterization produces fragments.

- A fragment is the combination of all the data needed to color a pixel:

1. raster position (pixel coordinates)
2. color
3. texture coordinates
4. depth
5. alpha
6. etc.

Rasterization

- There are many different methods and algorithms for rasterization. Different graphics libraries use different algorithms.
- But, they all follow a simple idea: take the viewport and overlay it on a grid of pixels and test points, typically the center of each pixel.
- Then, apply some algorithm to determine whether a particular primitive “covers” a pixel, and then color that pixel appropriately.



Drawing Lines

- Pixels are referenced with a coordinate pair (x , y) . Each pixel has color and intensity attributes.
- Drawing lines is performed by turning pixels on in the frame buffer.
- The contents of the frame buffer is displayed at regular intervals (60Hz is classical – What kind of refresh rates do we have on monitors in the class?).

Simple Line Drawing Algorithm

- Given two points (x_1, y_1) and (x_2, y_2) and from the general equation of a 2D line $y=mx+b$, we can write the following:
  - $y_1 = mx_1+b$
  - $y_2 = mx_2+b$
  - $b = y_1-mx_1$
  - $b = y_2-mx_2$

Simple Line Drawing Algorithm

- If we assume the same b (note: no b_1 or b_2 used)
  - $y_1 - mx_1 = y_2 - mx_2$
  - $y_1 - y_2 = m(x_1 - x_2)$
  - $\Delta y = m\Delta x$

  - This makes perfect sense when using a pen to draw a line on an angle… Anything problematic coming to mind for graphics?

Simple Line Drawing Algorithm

- Consider the line $y = \frac{1}{3}x + \frac{2}{3}$, starting from (1,1)

x	y	round(y)
1	1	1
2	**4/3**	1
3	**5/3**	2
4	2	2
5	**7/3**	2
6	**8/3**	3
7	3	3

Simple Line Drawing Algorithm

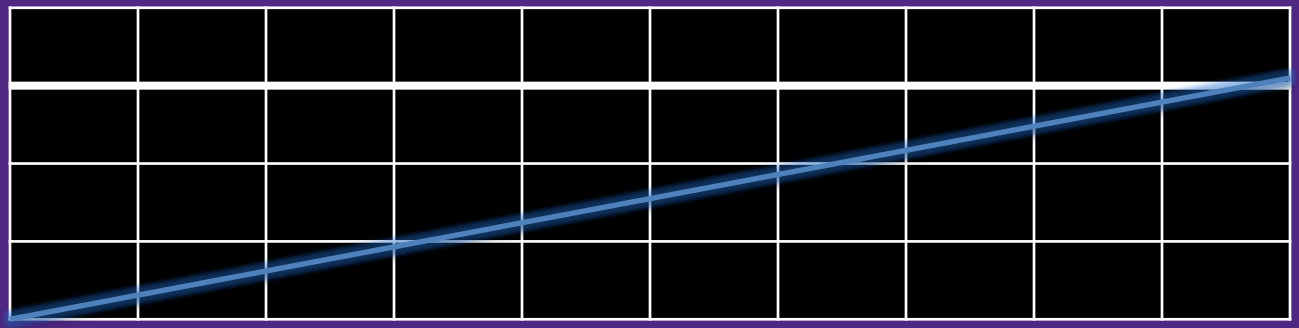
- Really, this is computing:
  - $\Delta y = m\Delta x$
  - $(y_{i+1}-y_i) = m(x_{i+1}-x_i)$

  Thus, when $\Delta x = 1$, this simplifies to…

  - $y_{i+1} = m+y_i$

Simple Line Drawing Algorithm

- So, imagine we have a grid of pixels like so, and want to render a line like so :



- What happens when we use this algorithm to plot a sline for which the slope is greater than 1?
- The line cannot be solid when going across discrete pixels, as y would vary by 1 and x would vary by m^{-1}

Bresenham's Integer Arithmetic Line Algorithm

- So, we need a way of picking which pixels to activate when drawing our line. Bresenham's algorithm here is meant to help with that challenge.

- Let $0 < m < 1$ and $\Delta x=1$. Given x_{i+1}, what is y?

$$y = m(x_{i+1}) + b$$
$$d_1 = y - y_i = m(x_i + 1) + b - y_i$$
$$d_2 = (y_i + 1) - y = y_i + 1 - m(x_i + 1) - b$$

Bresenham's Integer Arithmetic Line Algorithm

- We want to find the difference between the distances d_1 and d_2, so we write:

$$d_1 - d_2 = m(x_i+1) + b - y_i - (y_i + 1 - m(x_i + 1) - b)$$
$$= 2m(x_i+1) - 2y_i + 2b - 1 \quad \text{\#Remember m is slope}$$
$$= 2\frac{\Delta y}{\Delta x}(x_i+1) - 2y_i + 2b - 1$$

Bresenham's Integer Arithmetic Line Algorithm

- Multiply both sides by Δx

$$\Delta x(d_1-d_2) = 2\Delta y x_i + 2\Delta y - 2\Delta x y_i + 2b\Delta x - \Delta x$$

- Let $\Delta x(d_1-d_2)$ be a new measure, p_i and collect some terms:

$$p_i = 2\Delta y x_i - 2\Delta x y_i + 2\Delta y + \Delta x(2b-1)$$

Bresenham's Integer Arithmetic Line Algorithm

- If $p_i < 0$, then $d_1 < d_2$ and y_i is closer to y.
- Otherwise, $d_1 >= d_2$ and y_{i+1} is closer to y.
- If we rewrite in terms of (x_{i+1+}, y_{i+1}):

$p_{i+1} = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} + 2\Delta y + \Delta x(2b-1)$

take difference with p_i eqn to get:

$p_{i+1} - p_i = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} - (2\Delta y x_i - 2\Delta x y_i)$

Bresenham's Integer Arithmetic Line Algorithm

- Since $0 < m < 1$, we know that $x_{i+1} = x_i + 1$, allowing:

$$p_{i+1} - p_i = 2\Delta y(x_i+1) - 2\Delta x y_{i+1} - (2\Delta y x_i - 2\Delta x y_i)$$
$$= 2\Delta y - 2\Delta x (y_{i+1} - y_i)$$

- This gives us an interative mechanism to determine which pixel to turn on, as we progress on the x axis in steps of size 1

Bresenham's Integer Arithmetic Line Algorithm

- At this point, we need to determine the value of p_1 such that the iteration can start. So we write:

$$p_i = 2\Delta y x_i - 2\Delta x y_i + 2\Delta y + \Delta x(2b-1)$$

Divide by Δx…

$$\frac{p_i}{\Delta x} = 2(mx_i - y_i + b) + \frac{2\Delta y - \Delta x}{\Delta x}$$

- Simplify lastly to…

Bresenham's Integer Arithmetic Line Algorithm

- Simplify lastly to…

$p_i = 2\Delta y - \Delta x$

- If $p_i < 0$ then y_i is used for y_{i+1} and then $p_{i+1}-p_i = 2\Delta y - 2\Delta x(y_{i+1}-y_i)$
- Since $y_{i+1}-y_i = 0$, in this case, $p_{i+1} = p_i + 2\Delta y$.
- If $p_i > 0$, then y_i+1 is used as y_{i+1} and hence $y_{i+1} - y_i = 1$. (Consequently $p_{i+1}=p_i+2\Delta y-2\Delta x$)

Bresenham's Integer Arithmetic Line Algorithm

- Our linear algebra here allows us to condense this into the following code:

```
begin
    plot(x1,y1);
    for (i = x1 to x2, i++)
        if (i == x1)
                pi = 2Δy –Δx
        else
                if (pi < 0)
                        pi = pi + 2Δy
                else
                        pi = pi + 2Δy - 2Δx
                        y1++
                x1++
                plot(x1,y1)
    end for
end
```

Bresenham's Integer Arithmetic Line Algorithm

- Let's take the case of drawing a line of length 4, where x goes from 1 to 4 and y goes from 1 to 2

```
begin
    plot(x1,y1);
    for (i = x1 to x2, i++)
        if (i == x1)
                pi = 2Δy –Δx
        else
                if (pi < 0)
                        pi = pi + 2Δy
                else
                        pi = pi + 2Δy - 2Δx
                        y1++
                x1++
                plot(x1,y1)
    end for
end
```

i	x	Δx	y	Δy	p_i
1	1	3	1	1	-1
2	2	3	1	1	1
3	3	3	2	1	-3
4	4	3	2	1	-1
5	5	-	-	-	-

Bresenham's Integer Arithmetic Line Algorithm

- **Class Activity (10 minutes): 'Draw' line of length 6, where x goes from 1 to 6 and y goes from 1 to 4.**
- **Keep track of all intermediate values using a table, notepad, excel, etc.**

```
begin
    plot(x1,y1);
    for (i = x1 to x2, i++)
        if (i == x1)
                pi = 2Δy –Δx
        else
                if (pi < 0)
                        pi = pi + 2Δy
                else
                        pi = pi + 2Δy - 2Δx
                        y1++
                x1++
                plot(x1,y1)
    end for
end
```

i	x	Δx	y	Δy	p_i
1	1	9	1	4	-1
2	2	9	1	4	7
3	3	9	2	4	-3
4	4	9	2	4	5
5	5	9	3	4	-5
6	6	9	3	4	3
7	7	9	4	4	-7
8	8	9	4	4	1
9	9	9	5	4	-9
10	10	9	5	4	-1

Improved Bresenham's Line Algorithm

```
#Initialize values.
Δx = x2 – x1          # Calculate and reuse Δx and Δy ONCE.
Δy = y2 – y1          # This is more optimized, less error than iterative update.
pi = (2 * Δy) - Δx
y = y1
yi = 1
#Drawing loop
for x from x1 to x2
    plot(x, y)
    if pi > 0
        y = y + yi
        pi = pi + 2 (Δy - Δx)
    else
        pi = pi + 2Δ y
    end if
end for
```

Bresenham's Integer Arithmetic Line Algorithm

- All of this so far handles the case where the slope of the line to be drawn is between 0 and 1.
- You might recall from the start that the example given was one where slope was greater than 1…What do we do then?
- Thankfully, the other cases are symmetrical, in that we can exchange x and y and/or exchange Δx with Δy and derive the other cases.

Bresenham's Integer Arithmetic Line Algorithm

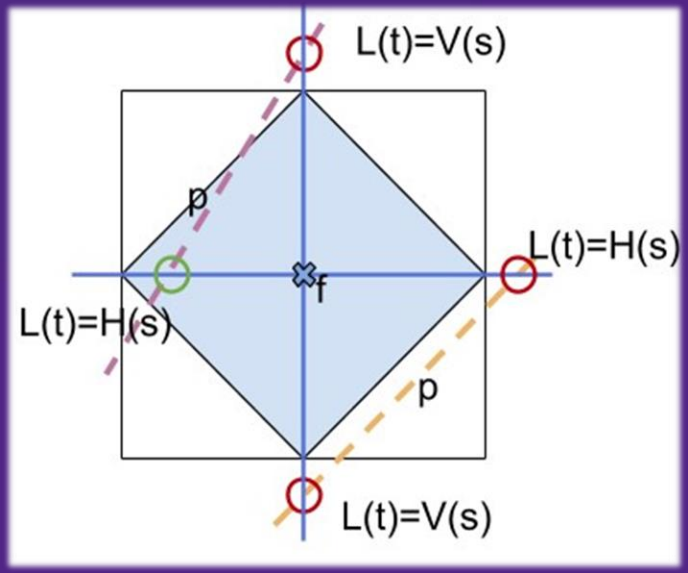
- All of this so far handles the case where the slope of the line to be drawn is between 0 and 1.
- You might recall from the start that the example given was one where slope was greater than 1…What do we do then?
- Thankfully, the other cases are symmetrical, in that we can exchange x and y and/or exchange Δx with Δy and derive the other cases.

Modern Line Algorithms

- The classic line rendering algorithm is Bresenham's line algorithm. But let's consider something more modern.

- Let's look at the line rasterization in DirectX 11 and modern OpenGL. What makes this "modern" is that it can create so-called *non-Bresenham fragments*.

- Enter the **Diamond Exit Rule:**



Modern Line Algorithms

- For a pixel centered at (x_f, y_f) define the diamond shaped region:

$$\left\{(x,y)\middle|\,|x-x_f|+|y-y_f|<\frac{1}{2}\right\}$$

- A line covers a pixel if the line exits the pixel's diamond test area when travelling along the line from the start towards the end.

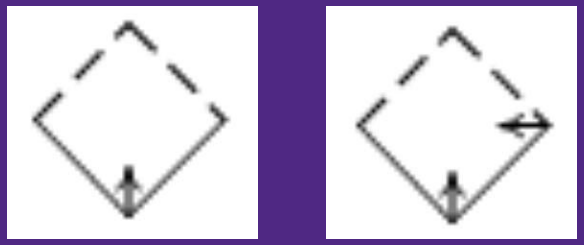
- What is a test area you ask? The test area (really test edges) depends on the line's slope m.

1. If $|m| \leq 1$, then the test area is the diamond's bottom two edges and the bottom corner

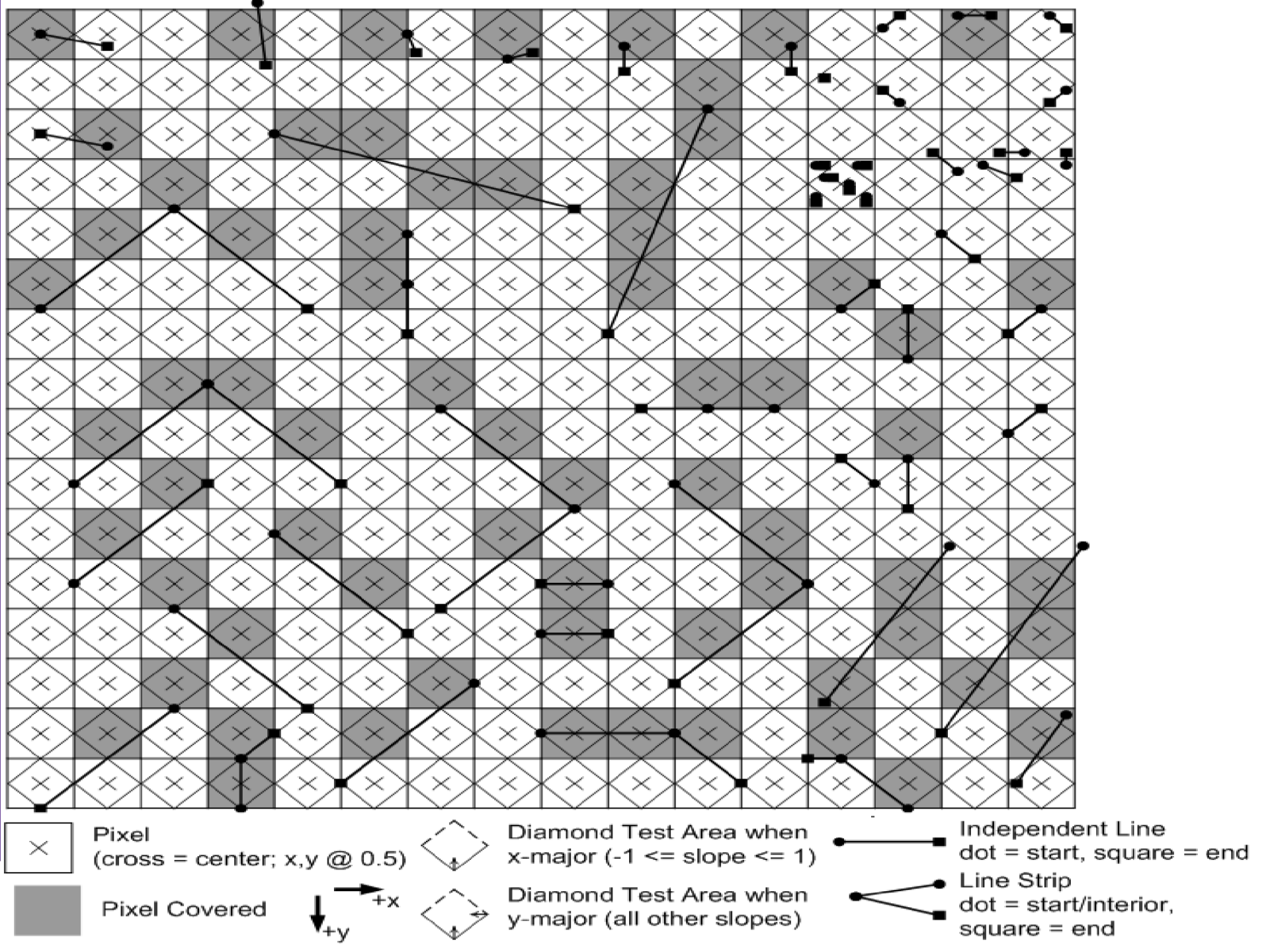
2. Otherwise, the test area is the diamond's bottom two edges and the bottom corner and the right corner.

Modern Line Algorithms

- If $|m| \leq 1$ the line is called x-major.
- Otherwise it is called y-major.



- If a line exits the diamond's test area, or is tangent to the test area, the entire pixel gets colored in.
- Notice that 3 corners in x-major (2 corners in y-major) are excluded from the test area.
- If a line merely touches one of the corners in the test area, then the pixel is colored. Otherwise, it is not.


Pixel
(cross = center; x,y @ 0.5)
Pixel Covered
+x
+y
Diamond Test Area when
x-major (-1 <= slope <= 1)
Diamond Test Area when
y-major (all other slopes)
Independent Line
dot = start, square = end
Line Strip
dot = start/interior,
square = end

Modern Line Algorithms and OpenGL

- In the land of two-dimensional graphics all our primitives are created used glVertex2f or glVertex2i.
- With just that we can draw all the primitives we saw earlier. And, more importantly, draw some neat things.
- Let’s consider a very simple idea. If we draw "enough" single dots, they will look like a complete picture. Right?
- That’s essentially what pixels are anyways. So let’s try that.

Modern Line Algorithms and OpenGL

- The most obvious use case for a dot plot is a literal dot plot: a plot of a function using dots.

```
//set the world origin at the center of the screen
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-10, 10, 8, 8, -1, 1);

/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Poll for and process events */
    glfwPollEvents();

    /* Clear here */
    glClear(GL_COLOR_BUFFER_BIT);

    //Draw the plot as points
    float x, y;
    float min_x = -10, max_x = 10;
    glColor3f(0.0f, 0.0f, 0.0f);
    glPointSize(2.0f);
    glBegin(GL_POINTS);
    for (int i = 0; i < N; ++i) {
        //calculate x-coord based on N and i.
        x = ((float) i)/N * (max_x - min_x) + min_x;
        y = my_fave_f(x);
        glVertex2f(x, y);
    }
    glEnd();

    /* Swap front and back buffers */
    glfwSwapBuffers(window);

}
```

Modern Line Algorithms and OpenGL

- Instead of dots, a piece-wise linear plot also makes sense.

Instead of dots, a piece-wise linear plot also makes sense.

```
glLineWidth(2.0f);
glBegin(GL_LINES);
for (int i = 1; i < N; ++i) {
    //calculate x-coord based on N and i.
    x = ((float) i-1)/N * (max_x - min_x) + min_x;
    y = my_fave_f(x);
    glVertex2f(x, y);

    x = ((float) i)/N * (max_x - min_x) + min_x;
    y = my_fave_f(x);
    glVertex2f(x, y);
}
glEnd();
```

Modern Line Algorithms and OpenGL

- Here we see two of our first attributes that aren't color!
- • glPointSize defines the size of a point; default 1.
- • glLineWidth defines the width of a line; default 1.
- More or less, this corresponds to pixels.

Poly-line Drawings

- A very useful kind of 2D drawing is a poly-line: connecting endpoints of multiple lines together.

- With N vertices with get N – 1 line segments.
-
- We can think of it as drawing by connecting dots, without picking up the pen from the paper.

- If we want to draw a star using the typical "opposite corners" approach, we can use poly-lines.

- It all starts with glBegin(GL_LINE_STRIP).

Poly-line Drawings

```
glBegin(GL_LINE_STRIP);
    glVertex2f(-3.5f, -3.5f);
    glVertex2f(0.0f, 7.0f);
    glVertex2f(3.5, -3.5f);
    glVertex2f(-5.0f, 3.5f);
    glVertex2f(5.0f, 3.5f);
    glVertex2f(-3.5f, -3.5f);
glEnd();
```

The nice part of polyline drawings is that they quite *portable*. Given a list of vertex positions, say, stored in a file, they can easily be loaded and rendered as a polyline drawing.

Drawing Circles

- The equation of a circle centered at the origin is given by:

$$x^2+y^2=r^2$$

- where r is the radius. Since the circle has abundant symmetry, we only need to compute the pixel coordinates of one eight of the circle to draw.

- The technique here is to start at the top (0,r) and proceed until x=y (assumes circle is centered at the origin)

Bresenham's Circle Drawing Algorithm

- Assume $(x_c,y_c) = (0,0)$ and start at the top of the circle $(0,r)$
- The current pixel is (x_i,y_i)
- Next pixel to turn on will be (x_i+1,y_i) or (x_i+1,y_i-1)

Bresenham's Circle Drawing Algorithm

- We can obtain the exact value for y as $y^2=r^2-(x_i+1)^2$
- And define the distances d_1 and d_2 as follows:

$d_1 = y_i^2-y^2 = y_i^2 - (r^2 - (x_i+1)^2)$

$d_2 = y^2-(y_i-1)^2 = r^2 - (x_i +1)^2 - (y_i - 1)^2$

- Which allows for us to determine p_i as follows:

$p_i = d_1 - d_2 = 2(x_i+1)^2+y_i^2+(y_i-1)^2-2r^2$

Bresenham's Circle Drawing Algorithm

- If $p_i < 0$, then we choose y_i for y_{i+1} .
- Otherwise, we choose y_i-1

- Just like with the line algorithm, we determine the next p_i, p_{i+1}, in terms of p_i :

$$p_{i+1} = 2((x_i+1)+1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2r^2$$
$$= p_i + 4x_i + 6 + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i)$$

- Where y_{i+1} is either y_i or y_i-1, depending on the sign of p_i

Bresenham's Circle Drawing Algorithm

- To initialize p_i, or to find p_1, we set (x_1,y_1) to $(0,r)$, so starting with:

$$p_i = d_1 - d_2 = 2(x_i+1)^2+y_i^2+(y_i-1)^2-2r^2$$
$$= 2(0+1)^2 + r^2 + (r-1)^2 - 2r^2$$
$$= 2 + r^2 + r^2 -2r +1 - 2r^2$$
$$= 3 - 2r$$

Bresenham's Circle Drawing Algorithm

- Suppose $p_i < 0$
  - Then, $y_{i+1} = y_i$, and we have:
  - $p_{i+1} = p_i+4x_i+6+2(y_i^2 -y_i^2)-2(y_i-y_i) = p_i+4x_i+6$
- Conversely, if $p_i >= 0$, then $y_{i+1} = y_i - 1$, and
  - $p_{i+1}= p_i+4x_i+6+2((y_i-1)^2 - y_i^2)-2(y_i-1-y_i) = pi+4(x_i- y_i)+10$

Bresenham's Circle Drawing Algorithm

- This algorithm can be expressed as:

```
begin
    x=0 ;
    y=r ;
    plot ( x , y) ;
    d=3−2 r ;
    while ( x< y ) { #Remember that we're only progressing until x = y and then using symmetry.
        x ++ ;
        if ( d<0 ) {
                d=d+4x+6 ;
        }
        else {
                y -- ;
                d=d+4(x− y)+10 ;
        }
        plot ( x , y) ;
    } #End while
end
```

Bresenham's Circle Drawing Algorithm

- Symmetrical cases:
  - This algorithm only draws one eighth of the circle
  - (It may be tempting to think it's $1/4^{th}$ – try to imagine where x = y occurs and what angle in the circle that would occur at!)
  - So, how do we get the rest of the circle?
  - For every pixel (x,y) activated, we activate:
    (y,x), (x, -y), (y, -x), (-y, -x), (-x, -y), (-x,y), (-y,x)

Bad Attempt at Humor #1

- What if we don't want to draw

?

Drawing Ellipses

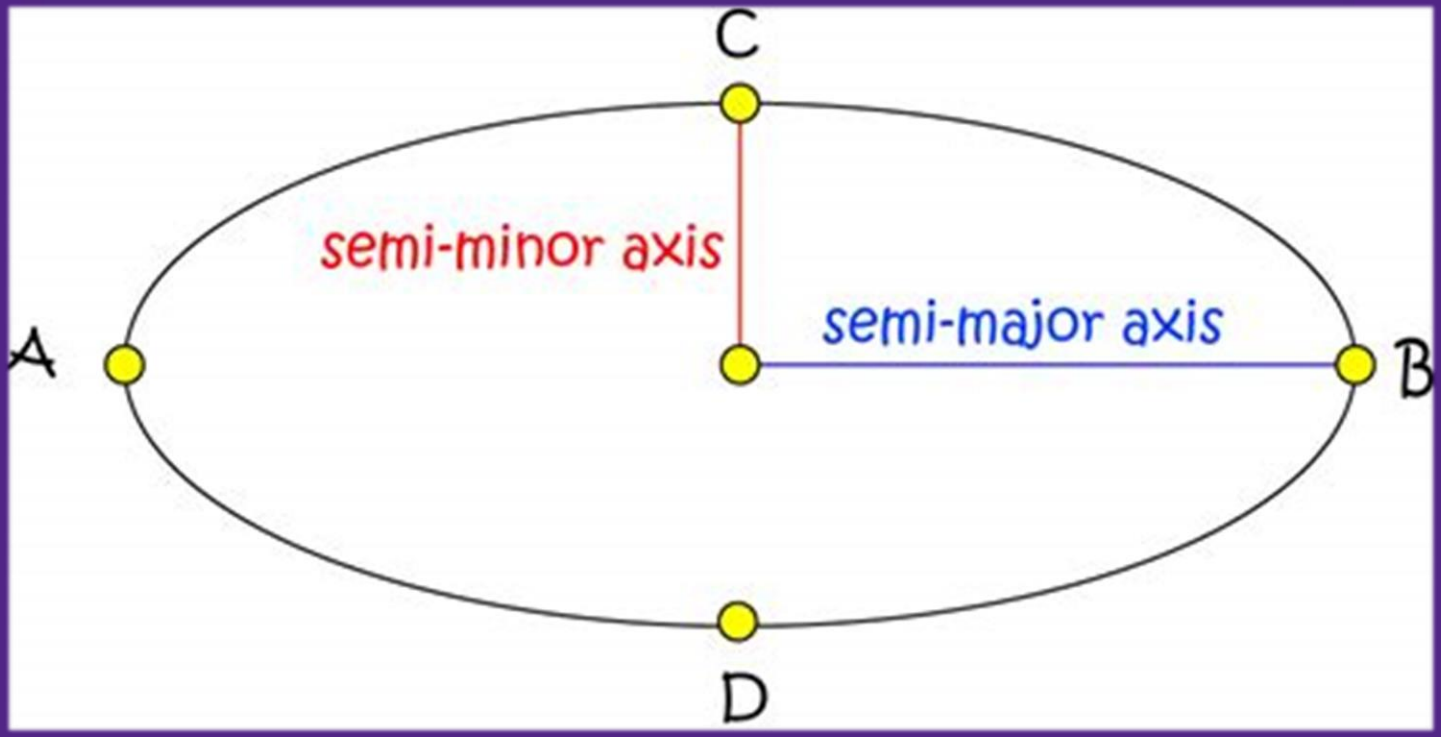
- The previous circle drawing algorithm can be extended to plot ellipses. We can write the ellipse equation as:

$$\left(\frac{x}{r_1}\right)^2 + \left(\frac{y}{r_2}\right)^2 = 1$$

- Where r_1 and r_2 are the semi-major and semi-minor axes, respectively.

- Bresenham's algorithm can trace ellipses by using this equation instead of that of a circle in the evaluation of the parameter p_i.

Drawing Ellipses



Drawing Ellipses

- For an ellipse centered at the origin, we can express y as:

$$y^2 = r_2^2\left(1 - \frac{x^2}{r_1^2}\right)$$

- And modify the computation of p_i based on this.
- For reasons that will become clearer when we talk about advantages of Bresenham's algorithms at the end, we want to limit our mathematics for tracing the ellipses to integer arithmetic only.

Drawing Ellipses

- In the previous equation, r_1 and r_2 can be referred to as the width and height of the ellipse respectively. As with the circle, we still need an expression that gives an exact y value at x_i+1. We can do this with:

$$y^2 = r_2^2\left(1 - \frac{(x_i + 1)^2}{r_1^2}\right)$$

- And modify the computation of p_i based on this.
- For reasons that will become clearer when we talk about advantages of Bresenham's algorithms at the end, we want to limit our mathematics for tracing the ellipses to integer arithmetic only.

Drawing Ellipses

- Like the circle, we need to define distance (using the squares):

$$d_1 = y_i^2 - y^2 = y_i^2 - r_2^2\left(1 - \frac{(x_i + 1)^2}{r_1^2}\right)$$

- Let's multiply by r_1^2 on both sides to eliminate the denominator

$r_1^2 d_1 = r_1^2(y_i^2 - y^2) = r_1^2 y_i^2 - r_2^2(r_1^2 - (x_i+1)^2)$

$= r_1^2 y_i^2 - r_1^2 r_2^2 + r_2^2 (x_i+1)^2$

Drawing Ellipses

- We can similarly define the second distance as:

$$d_2 = y^2 - (y_i - 1)^2 = r_2^2 \left(1 - \frac{(x_i+1)^2}{r_1^2}\right) - (y_i - 1)^2$$

- Let's multiply by r_1^2 again to eliminate the denominator

$$r_1^2 d_2 = r_2^2 (r_1^2 - (x_i + 1)^2) - r_1^2 (y_i - 1)^2$$

Drawing Ellipses

- Like we've defined it before, we can determine p_i as the difference between $r_1^2 d_1 - r_1^2 d_2$
- (Warning – some of the equations are about to get *long*).

$$p_i = r_1^2 y_i^2 - r_1^2 r_2^2 + r_2^2 (x_i+1)^2 - [\, r_2^2 (\, r_1^2 - (\, x_i + 1\,)^2\,) - r_1^2 (\, y_i - 1)^2\,]$$

- Which simplifies down to

$$p_i = r_1^2 y_i^2 - 2 r_1^2 r_2^2 + 2 r_2^2 (x_i+1)^2 + r_1^2 (y_i-1)^2$$

Drawing Ellipses

- Like before for circles, since we want an iterative process, we want to be able to express p_{i+1} in terms of p_i

$$p_{i+1} = r_1^2 y_{i+1}^2 - 2r_1^2 r_2^2 + 2r_2^2((x_i+1)+1)^2 + r_1^2(y_{i+1}-1)^2$$

Then, we want to take the difference between p_i and p_{i+1}

$$\begin{aligned} p_{i+1}-p_i &= r_1^2 y_{i+1}^2 - 2r_1^2 r_2^2 + 2r_2^2((x_i+1)+1)^2 + r_1^2(y_{i+1}-1)^2 \\ &\quad - [\, r_1^2 y_i^2 - 2r_1^2 r_2^2 + 2r_2^2(x_i+1)^2 + r_1^2(y_i-1)^2 \,] \\ &= p_1 + r_1^2(y_{i+1}^2 - y_i^2) + r_2^2(4x_i+6) + r_1^2(y_{i+1}-1)^2 - r_1^2(y_i-1)^2 \end{aligned}$$

Drawing Ellipses

- If $p_i < 0$, then we choose y_i for y_{i+1} and p_{i+1} becomes

$$p_{i+1} = p_i + r_2^2(4x_i+6)$$

- If $p_i >= 0$, then we choose y_i -1 for y_{i+1} and p_{i+1} becomes

$$p_{i+1} = p_i + r_2^2(4x_i+6) + 4r_1^2(1-y_i)$$

Drawing Ellipses

- We begin drawing at $(0,r_2)$ and use this for p_1 like so:

$$p_1 = 2r_2^2+r_1^2(1-2r_2)$$

- And we draw until:

$$\frac{x^2}{r_1^2} = \frac{y^2}{r_2^2}$$

Drawing Ellipses

- This point is where the slope of the tangent to the ellipse is m=-1
- To keep the stop condition in integer form, we multiply both sides with $r_1^2 r_2^2$ to obtain $(r_2 x)^2=(r_1 y)^2$, which we take as equivalent to $r_2 x=r_1 y$ for the purpose of the stop condition.
- The next step is to trace the remaining part of the ellipse after $r_2 x=r_1 y$; we do this by interchanging x and y and trace from $(r_1,0)$ in steps along y to get the rest of the first quadrant. p_1 here is:

$$p_1 = 2r_1^2+r_2^2(1-2r_1)$$

Drawing Ellipses

- The next step is to trace the remaining part of the ellipse after $r_2x=r_1y$; we do this by interchanging x and y and trace from $(r_1,0)$ in steps along y to get the rest of the first quadrant. p_1 here is:

$$p_1 = 2r_1^2+r_2^2(1-2r_1)$$

- And p_{i+1} becomes

$$p_{i+1} = p_i + r_1^2(4y_i+6) + 4r_2^2(1-x_i)$$

Bresenham's Ellipse Drawing Algorithm

Too long to put into slides without breaking it up into multiple components.

PDF will be posted for the algorithm itself.

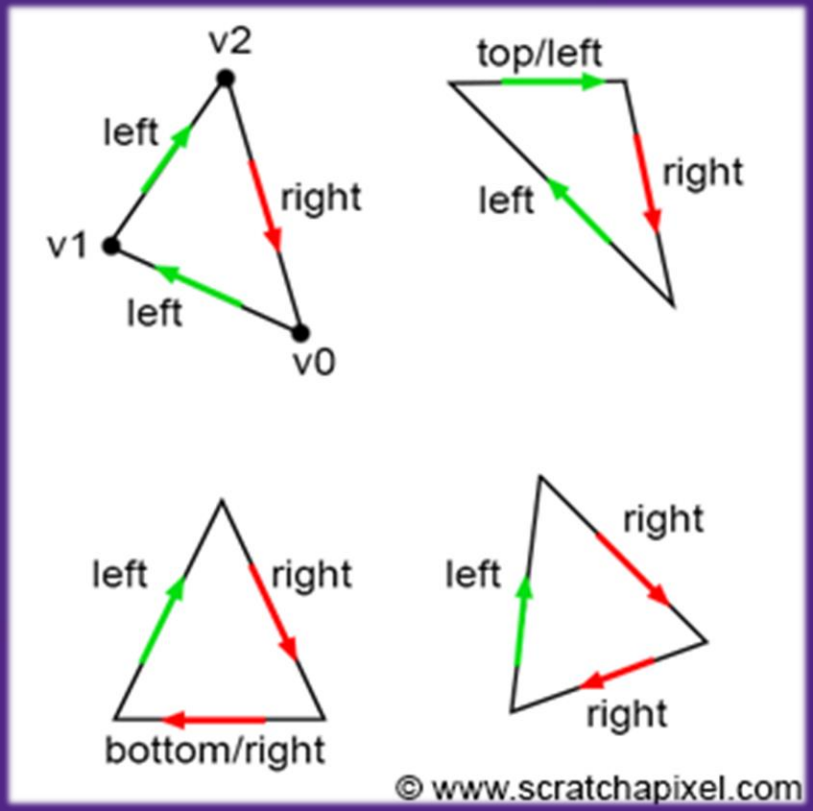
Why Bresenham's Algorithms?

- Remember: Every operation in a computer takes some amount of time, even if it's very fast.
- Addition, subtraction, often faster than multiplication, division.
- Other algorithms like Digital Differential Analyser (DDA) use floating point numbers – Bresenham's works with integers, which is more sensible for pixels, and has memory and speed advantages.
- Float operations lose precision or accuracy over time, integer operations do not.

Triangle Rasterization

- Triangle rasterization is a lot easier than line rasterization. We define a region enclosed by the 3 edges of a triangle. If the pixel's test point falls inside this region, color in that pixel.

- Revisiting the previous rasterized triangle, we see that the test points are the pixel's center and each pixel which is colored has its test point fall fully within the triangle's region.



Triangle Rasterization

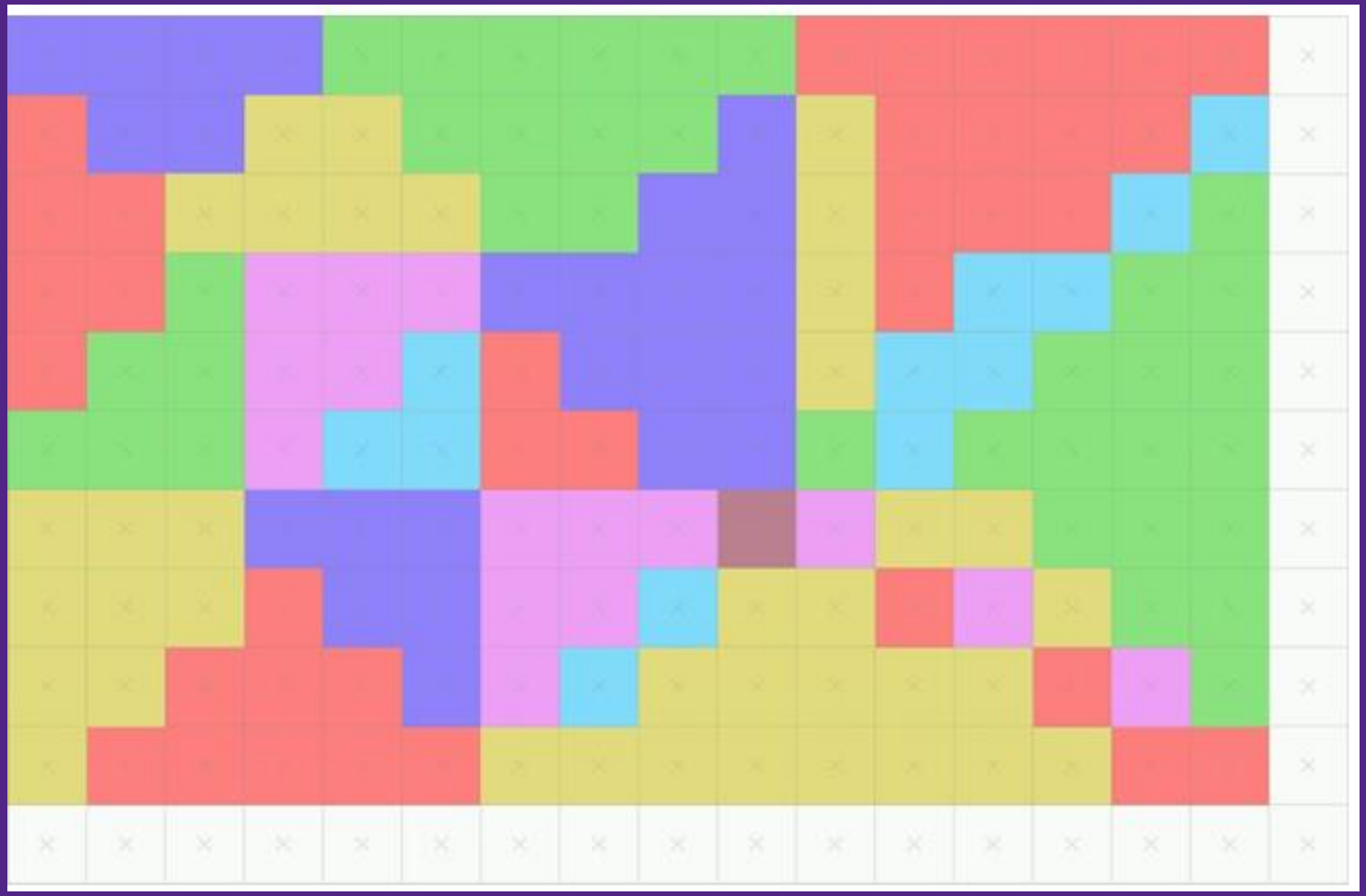
- But what if a test point falls on a triangle. Not within?
- In particular, what if two triangles share and edge and that edge goes directly through the test point?

- Introduce the top-left rule.
- A pixel's test point is considered to fall within a triangle's region if it lies on the top edge or the left edge of a triangle.
  - A triangle's top edge is a horizontal edge which is above the other two edges.
  - A triangle's left edge is a non-horizontal edge that is on the left side of the triangle.

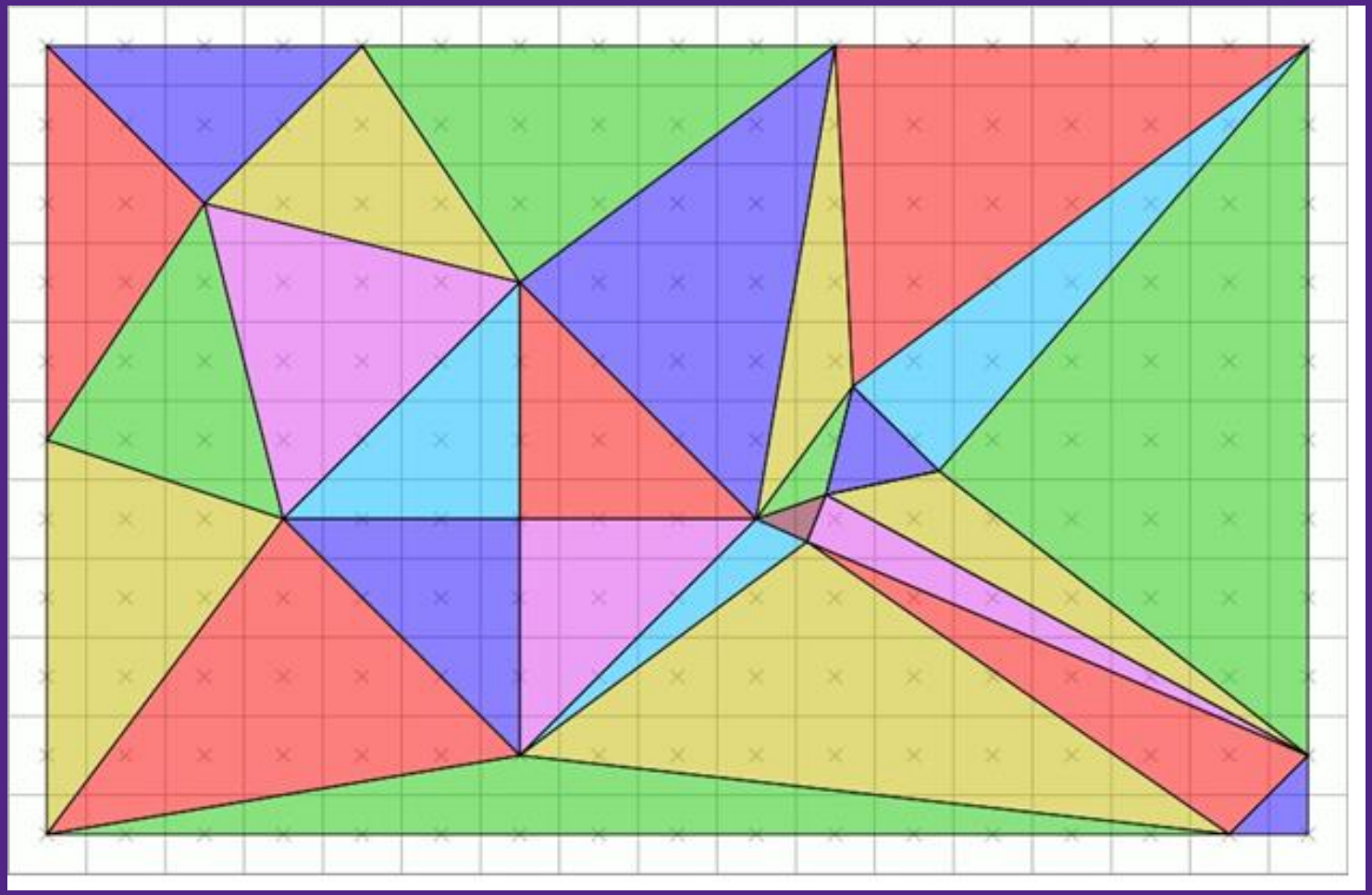
Triangle Rasterization

- Clearly, a triangle can either 0 or 1 top edges.
- A triangle can have 0 or 1 left edges.
- When we draw triangles side by side, we often implicitly want them to be viewed as ‘connected’ as a continuous surface.
- The top-left rule ensures that there is a uniform color applied to edges which are shared between two triangles.
- It also ensures no gaps between triangles.

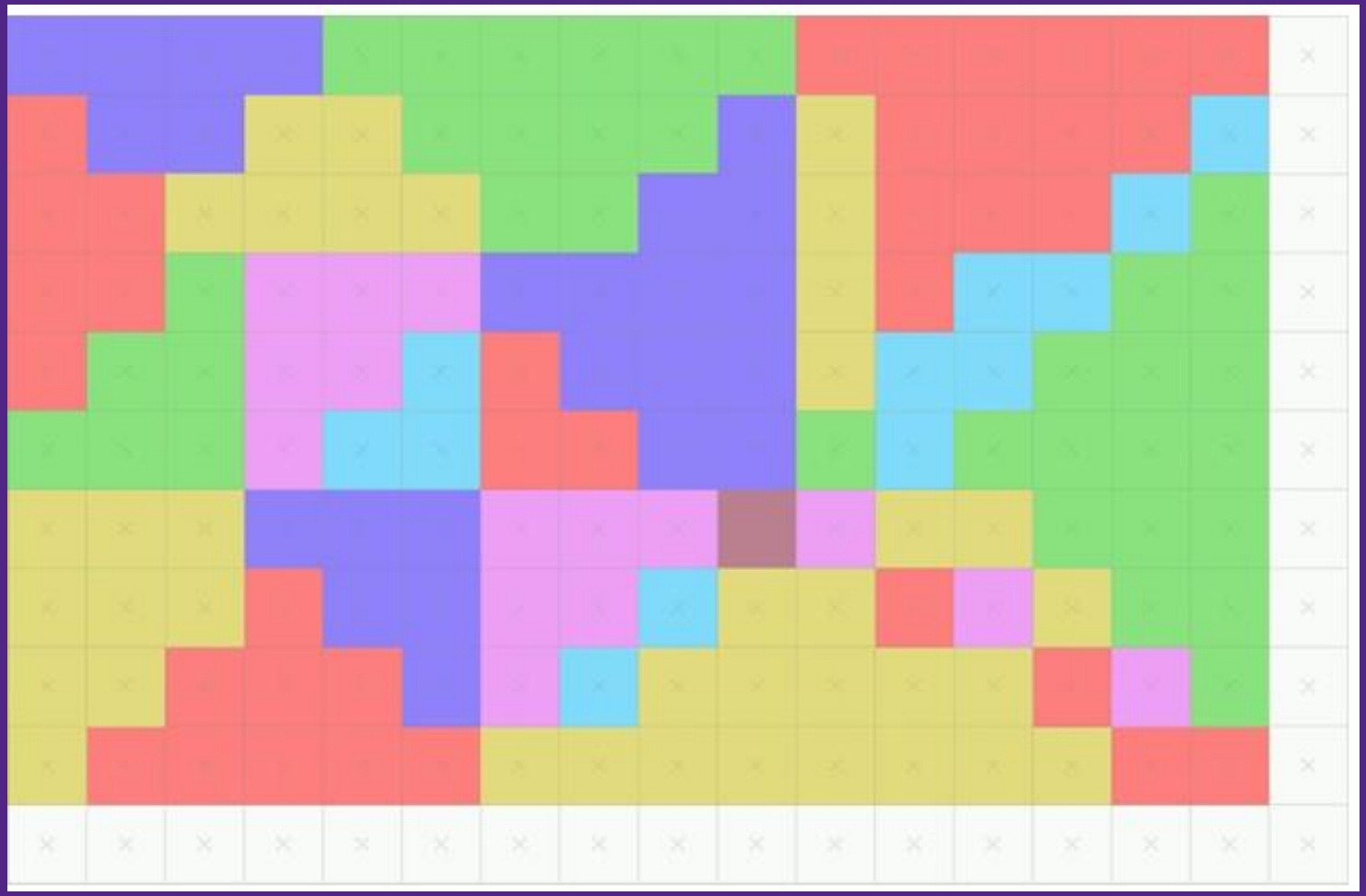
Triangle Rasterization



Triangle Rasterization



Triangle Rasterization



Triangle Rasterization

- Let's make the top-left rule algorithmic.
- 1. If a test point is fully inside of a triangle, then it is covered by that triangle.
- 2. If a test point is on two edges of a triangle, and both edges are either top or left, then it is covered by that triangle.
- 3. If a test point is on exactly one edge of a triangle, and that edge is a top or left edge, then it is covered by that triangle.