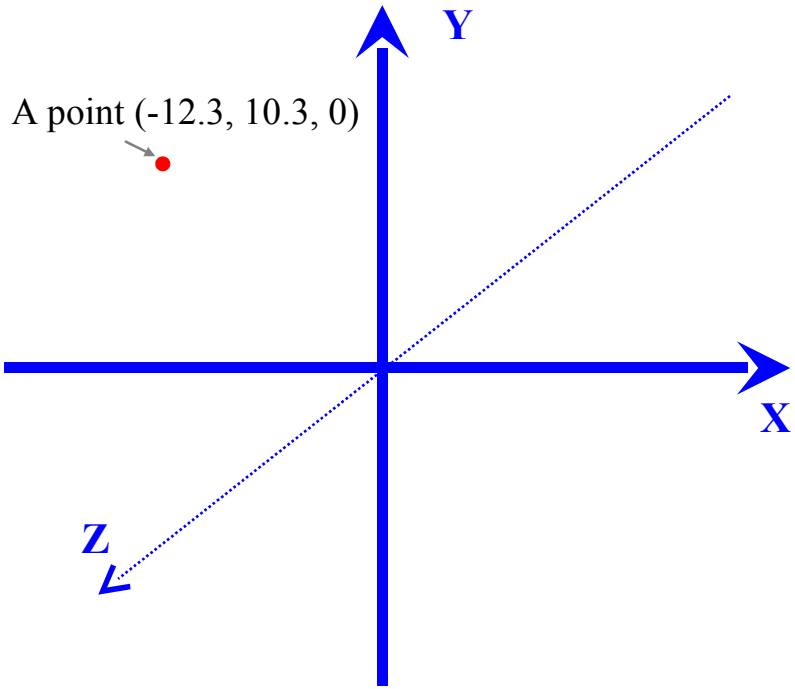
Scan Conversion / **Rasterization**

Scan Conversion

- **Rasterization** or **Scan Conversion** is required in order to convert vector data to Raster format for a scanline display device
  - convert each line to a set of regular pixels.
  - determine inside/outside areas when *filling polygons*.
  - scan-convert curves

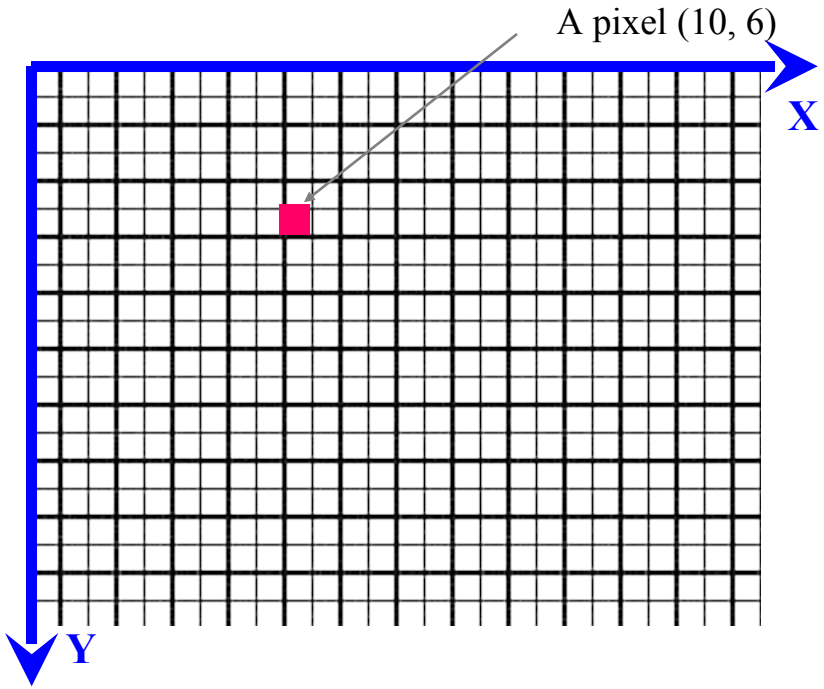
- The scan conversion process is interleaved with other processes to deliver and improve the final image, some of which are not entirely restricted to the discretization of vectors but are involved in generally determining the colour of each pixel in the raster. e.g:
  - shading & illumination,
  - hidden surface removal
  - texture mapping
  - depth testing

Pixel Co-ordinates



MODELLING CO-ORDINATES

- Mathematically vectors are defined in an infinite, “real-number” cartesian co-ordinate system

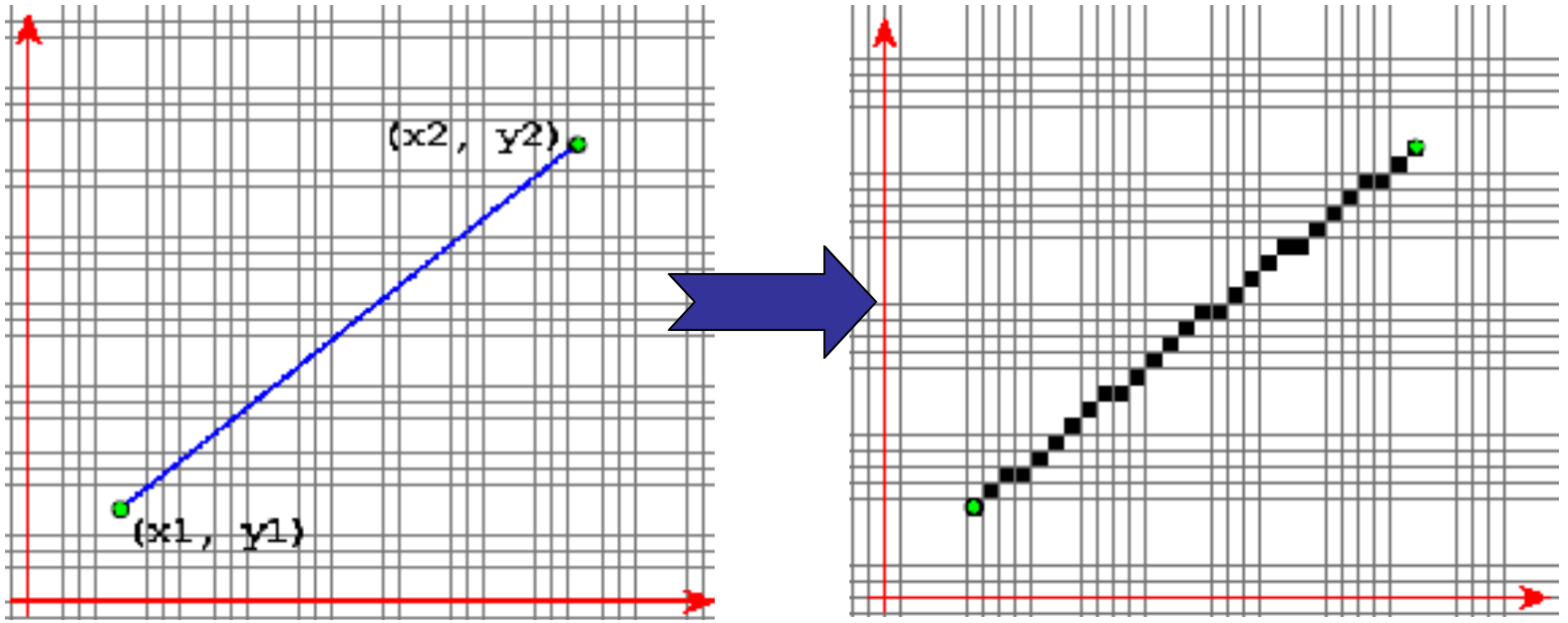


SCREEN COORDINATES

- a.k.a device co-ordinates, pixel co-ordinates
- On display hardware we deal with finite, discrete coordinates
- X, Y values in positive integers
- 0,0 is measured from top-left usually with +Y pointing down

How does a machine draw Lines?

1. Give it a start and end position.
2. Figure out which pixels to colour in between these…
   - How do we do this?
   - Line-Drawing Algorithms: DDA, Bresenham's Algorithm



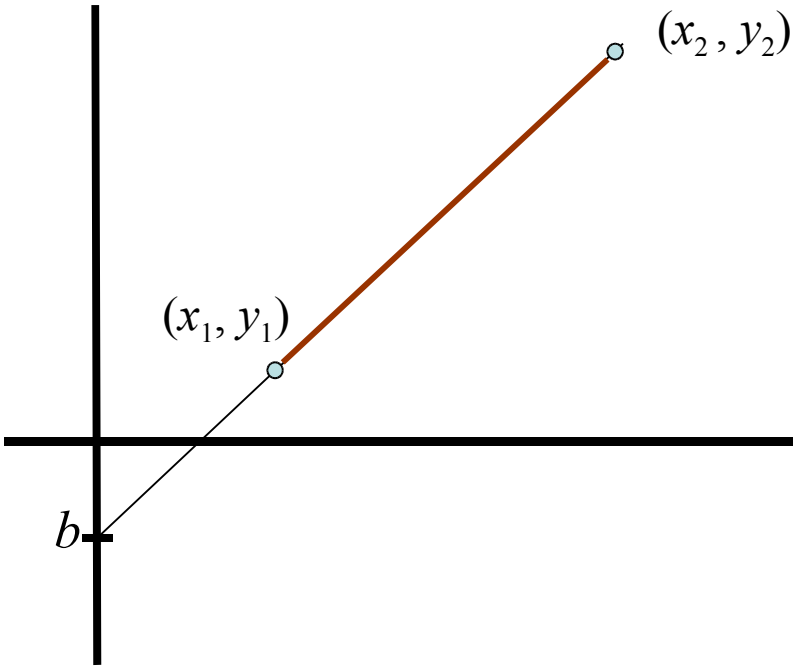
Line Equation

- The equation of an infinite line is given by

$$y = m \bullet x + b$$

m is a constant which represents the slope/gradient of the line

b is the value of x where the line intersects the y-axis



- The gradient *m* applies to the line as a whole but also to every part of the line.

- So given any two points on the line we can calculate the slope

$$m = \frac{RISE}{RUN} = \frac{y_2 - y_1}{x_2 - x_1}$$

DDA Algorithm

- Digital Differential Analyser: an algorithm for scan-converting lines
- Based on calculating the *rate of change* of x or y coordinates (Δx or Δy respectively)
- For each part of the line the following holds true:

$$m = \frac{\Delta y}{\Delta x} \quad \Rightarrow \quad \Delta y = m\Delta x$$

- If Δx = 1 i.e. 1 pixel then … $\Delta y = m$

- i.e. for each pixel we move right (along the x axis), we need to move down (along the y-axis) by *m* pixels

- In pixels, the gradient represents how many pixels we step upwards (Δy) for every step to the right (Δx)

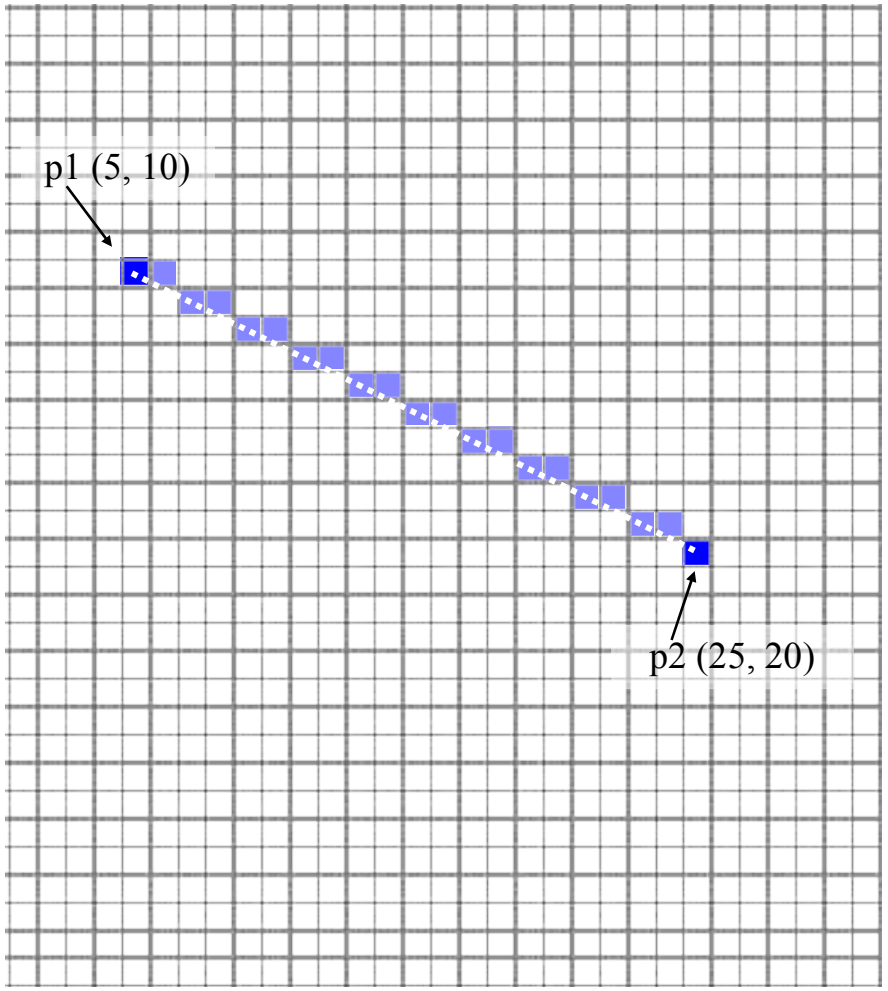
Example

- e.g. p1 = (5, 10), p2 = (25, 20)

$$m = \frac{y2 - y1}{x2 - x1} = \frac{20 - 10}{25 - 5} = 0.5$$

- Starting at p1, we move right by one pixel and down by 0.5 pixels each time.

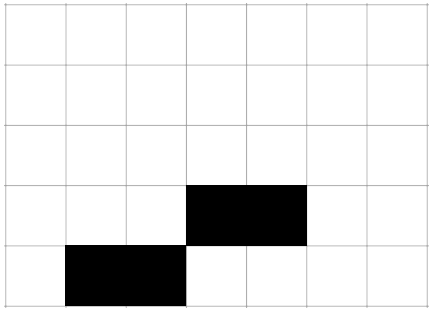
- But we can't move by "half-a pixel" so in pixel coordinates – we need to **round-off** to the nearest pixel value



Rounding Off

- Note that the actual pixel position is actually stored as a REAL number (in C/C++/java a float or a double)
- But we Round Off to the nearest whole number just before we draw the pixel.
- e.g. if m=.33 …

X	Y	Rounded { x, y }
1.0	0.33	{ 1, 0 }
2.0	0.66	{ 2, 0 }
3.0	0.99	{ 3, 1 }
4.0	1.32	{ 4, 1 }



Simple DDA

- So basically DDA draws lines as follows:

1. **Start at first endpoint.**

3. **Draw pixel.**

5. **Step right by one pixel and down by**
   `m * change_in_x`

6. **Draw pixel.**

8. **Repeat from step 3, until we reach second endpoint**

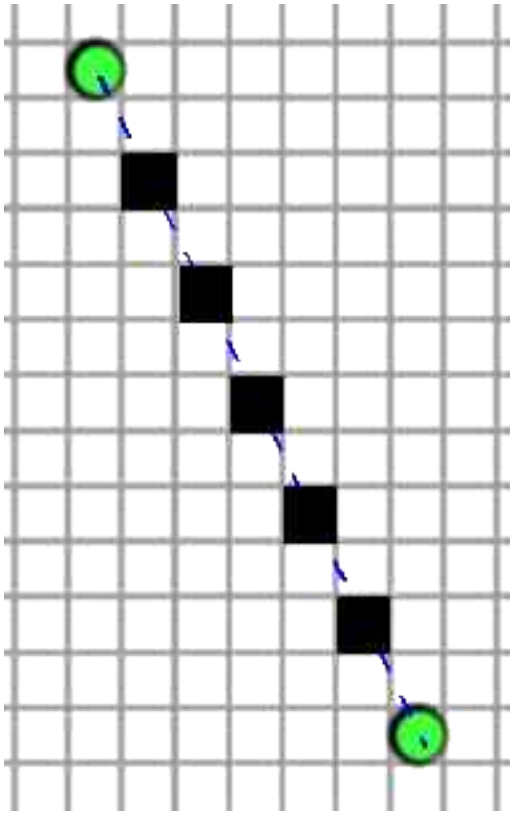
But **`change_in_x`** is 1! So, we just step down by m pixels

Pseudocode

```
1. Let x = x1; y = y1; m = (y2-y1)/(x2-
   x1);
2. Draw pixel (x, y)
3. WHILE (x < x2)  //i.e. we reached the second endpoint
   {
      x = x + 1;  //step right by one pixel
      y = y + m;  //step down by m pixels
      Draw pixel (ROUND(x), ROUND(y));
   }
```

Unfortunately…

- If the slope is very steep (i.e. Δy is much greater than Δx) then there's a problem. i.e. m is much greater than 1
- This is because we only draw one pixel for each x value (each time we step right)
- Solution: if the slope is too big, step for y instead



Y version

```
1. Let x = x1; y = y1; m* = (x2-x1)/(y2-y1);
2. Draw pixel (x, y)
3. WHILE (y < y2)  //i.e. we reached the second endpoint
   {
      y = y + 1;  //step down by one pixel
      x = x + m*;  //step right by m* pixels
      Draw pixel (ROUND(x), ROUND(y));
   }
```

Generic solution

- Of course, this is no better than the x-version as it has the same problems when the slope is too flat. i.e. if Δx is much greater than Δy we get the same “broken” line
- So we should check the steepness and call the appropriate version
- If the Gradient is high we use step along the y axis other wise we step along the x axis

```
1. Let x = x1; y = y1;
2. Draw pixel (x, y)
3. IF ((y2-y1)>(x2-x1))  //if it's steep
4. THEN
5. {      m = (x2-x1)/(y2-y1);  //y-version
          WHILE (y < y2)
          {
                y = y + 1; x = x + m;
                Draw pixel (ROUND(x), ROUND(y));
          }
   }
6. ELSE
7. {      m = (y2-y1)/(x2-x1);  //x-version
          WHILE (x < x2)
          {
                x = x + 1; y = y + m;
                Draw pixel (ROUND(x), ROUND(y));
          }
   }
```

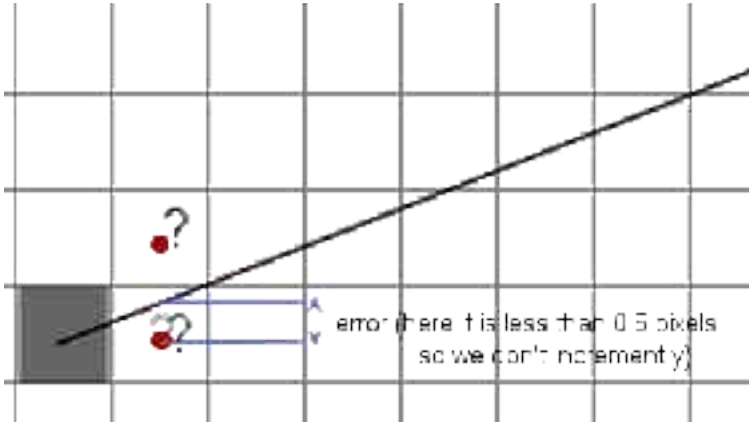
In actual code

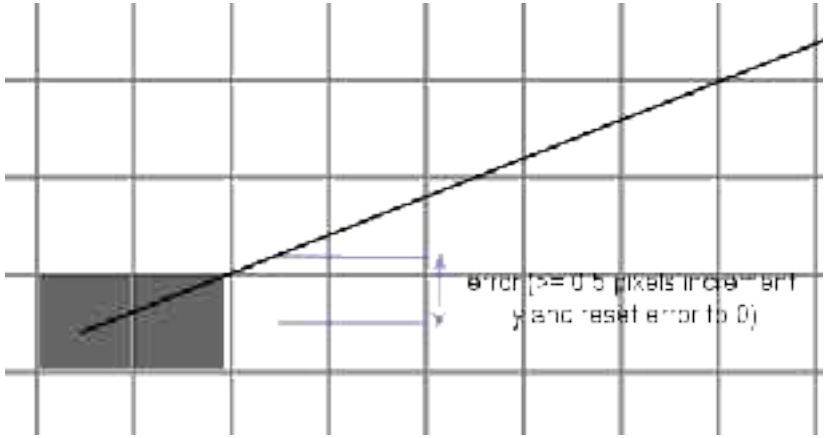
```
void lineDDA (int x1, int y1, int x2, int y2)
{
  int dx = x2 - x1, dy = y2 - y1, steps, k;
  float xIncr, yIncr, x = x1, y = y1;

  if ( abs(dx)>abs(dy) ) steps = abs(dx);
  else steps = abs(dy);
  xIncr = dx / (float) steps;
  yIncr = dy / (float) steps;

  point ((int)x, (int)y); //roundoff and drawPixel
  for (k=0; k<steps; k++)
  {
    x = x + xIncr;
    y = y + yIncr;
    point ((int)x,(int)y); //round off and draw pixel
  }
}
```

Bresenham's Algorithm

- One disadvantage of DDA is the ROUNDing part which can be expensive
- Bresenham's Algorithm is based on essentially the same principles but is completely based on integer variables
- One of the earliest algorithms in computer graphics

Basic Concept





```
int deltax = abs (x2-x1);
int deltay = abs (y2-y1);
float error = 0;
float gradient = deltay / deltax;

int y=y1
for (int x = x1; x<x2; x++)
{
        drawPixel(x, y);
        error = error + gradient;
        if (error >= 0.5)
        {
                y = y+1;
                error = error - 1;
        }
}
```

Optimisation

- Multiply all fractions by deltax to get rid of the floats.

```
int deltax = abs (x2-x1);
int deltay = abs (y2-y1);
int error = 0;
int Xgradient = deltay;

int y=y1
for (int x = x1; x<x2; x++)
{
        drawPixel(x, y);
        error = error + Xgradient;
        if (error >= 0.5 * deltax)
        {
                y = y+1;
                error = error - deltax;
        }
}
```

Optimisation

- Multiply all fractions by deltax to get rid of the floats.
- We might also want to get rid of the 0.5 in the IF statement. By multiplying both sides of the inequality by 2
- Multiplication by two is efficiently achieved by a bit shift

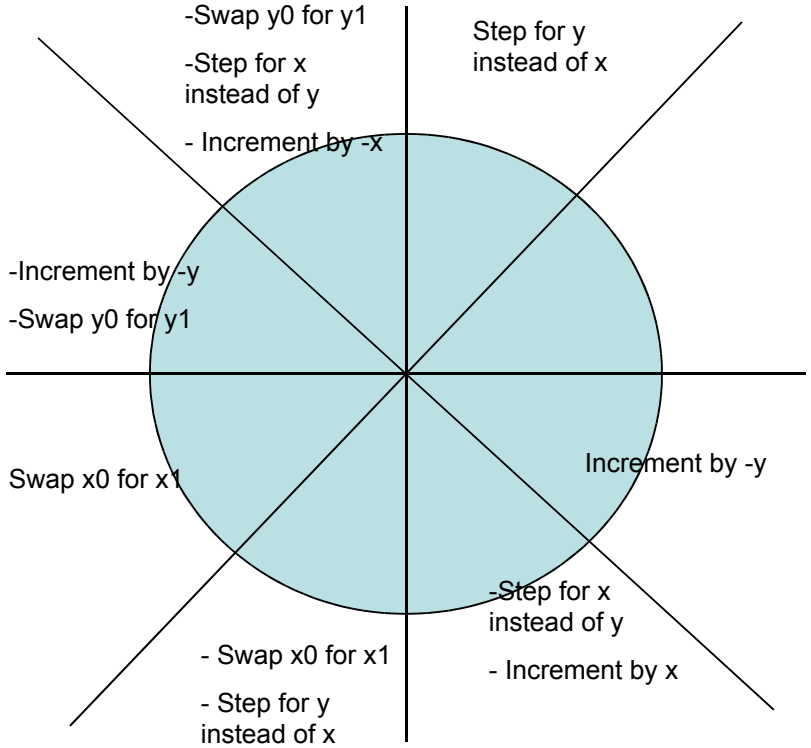
```
int deltax = abs (x2-x1);
int deltay = abs (y2-y1);
int error = 0;
int Xgradient = deltay;

int y=y1
for (int x = x1; x<x2; x++)
{
        drawPixel(x, y);
        error = error + Xgradient;
        if (2*error >= deltax)
        {
                y = y+1;
                error = error - deltax;
        }
}
```

Bit shift (side note)

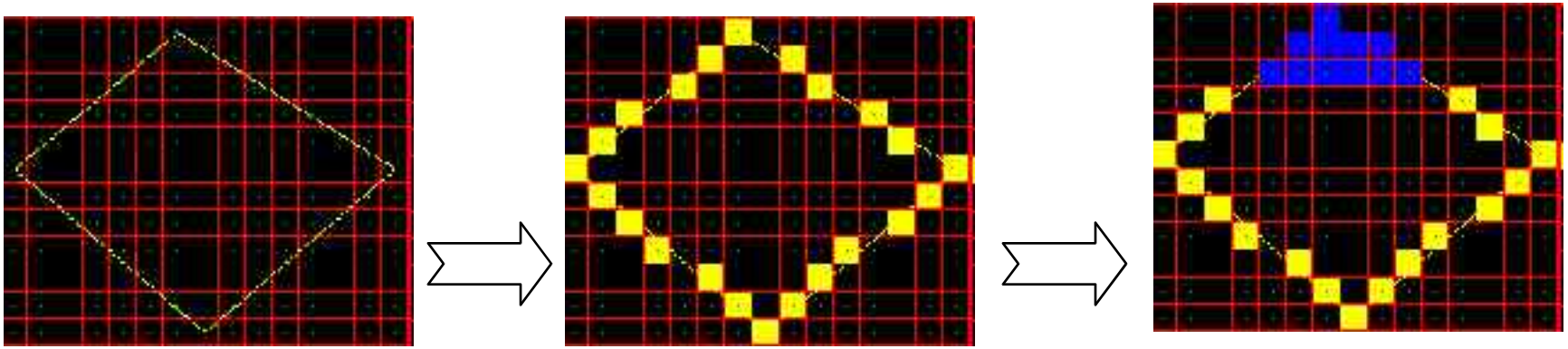
- Shifting decimal values 1 digit to the left multiplies by 10
  - e.g. 5x10 = 50x10 = 500 etc.
- Shifting 1 digit to the left in binary multiplies a number by 2
- EXAMPLE1:
  - 0001 in binary is 1 in decimal
  - Shifting the “1” one digit to the left makes it 0010 in binary, which is 2 in decimal
- EXAMPLE2:
  - 0101 in binary is 5 in decimal
  - Shifting left gives 1010, which is 10 in decimal
- The simple operation of shifting binary digits to the left is much faster to implement in hardware (circuits) than multiplication.

Other Quadrants

- Note that this only applies to lines with a positive gradient
- But you can easily write a separate case for each other case
- Also if the gradient is too steep you need to step for x instead of y (as we saw in DDA)



Polygon Scan-conversion



- First scan convert all edges very coarsely, increment along y and don’t worry about gaps

- For each scan line, fill between start and end pixel

Area Filling

A simple recursive flood-fill algorithm: user selects a pixel S. S and all adjacent pixels of the same colour are changed, and the adjacent pixels of those etc… leads to some wasted processing

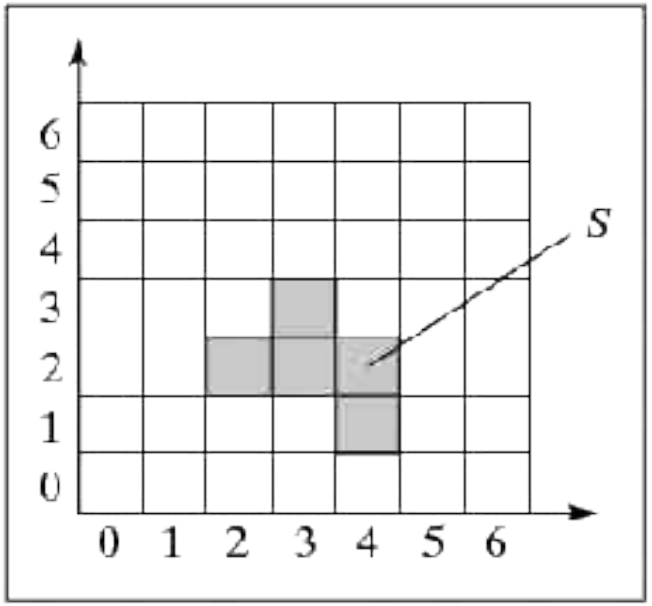
Recursive Flood Fill

2. Let orig_col = original colour of S
3. Let current_pixel = S
4. Change Colour of current_pixel to new_col
5. For all adjacent pixels p

   IF (colour == orig_col)

   THEN let p be the current_pixel

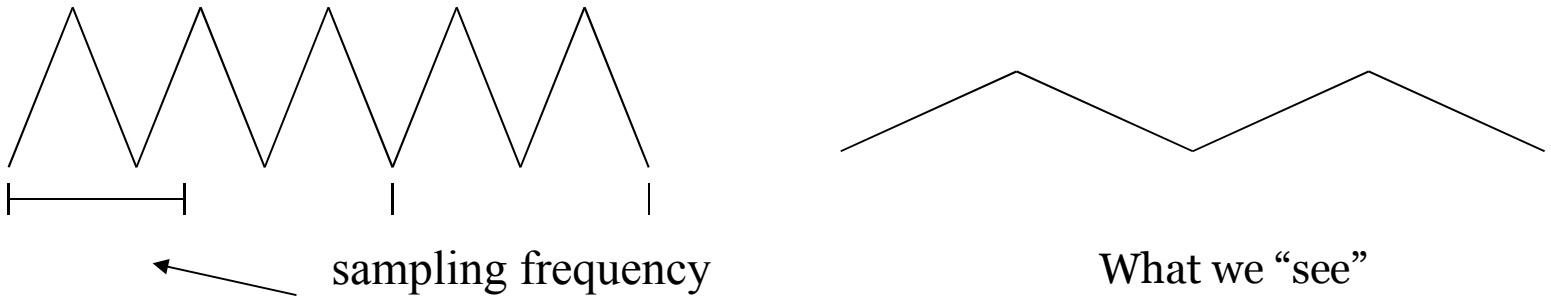
   Repeat from Step 3



Antialiasing

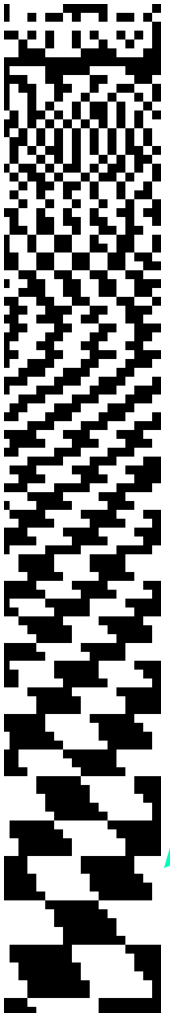
Aliasing

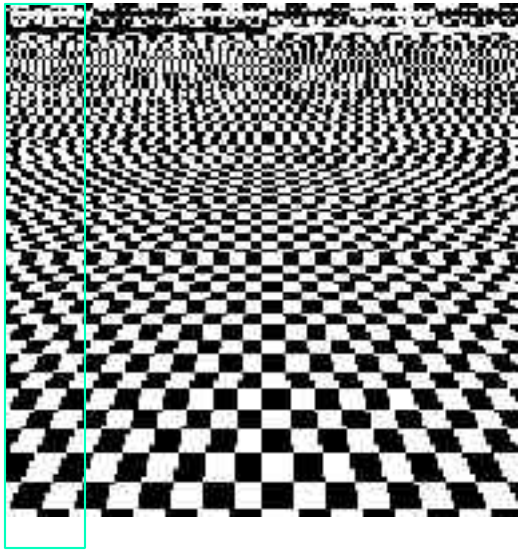
- Distortion of information due to low-frequency sampling



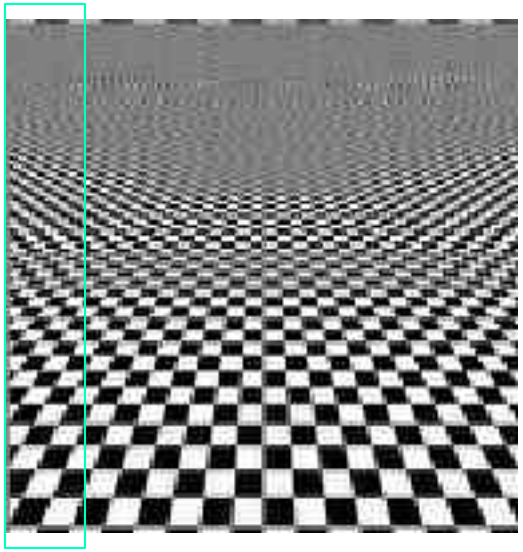
- In raster images – leads to jagged edges with staircase effect. Commonly occurs when a pattern has details that are smaller than a pixel in width
- We can reduce effects by **antialiasing** methods to compensate for undersampling

Antialiasing

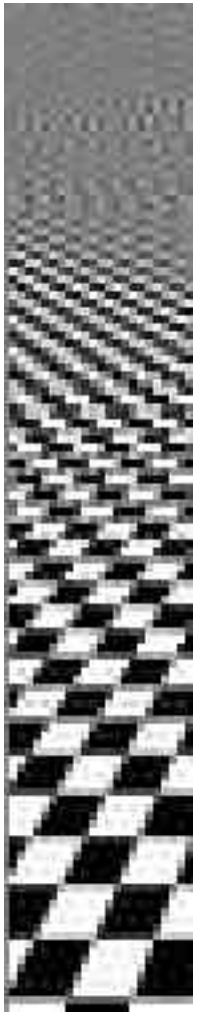




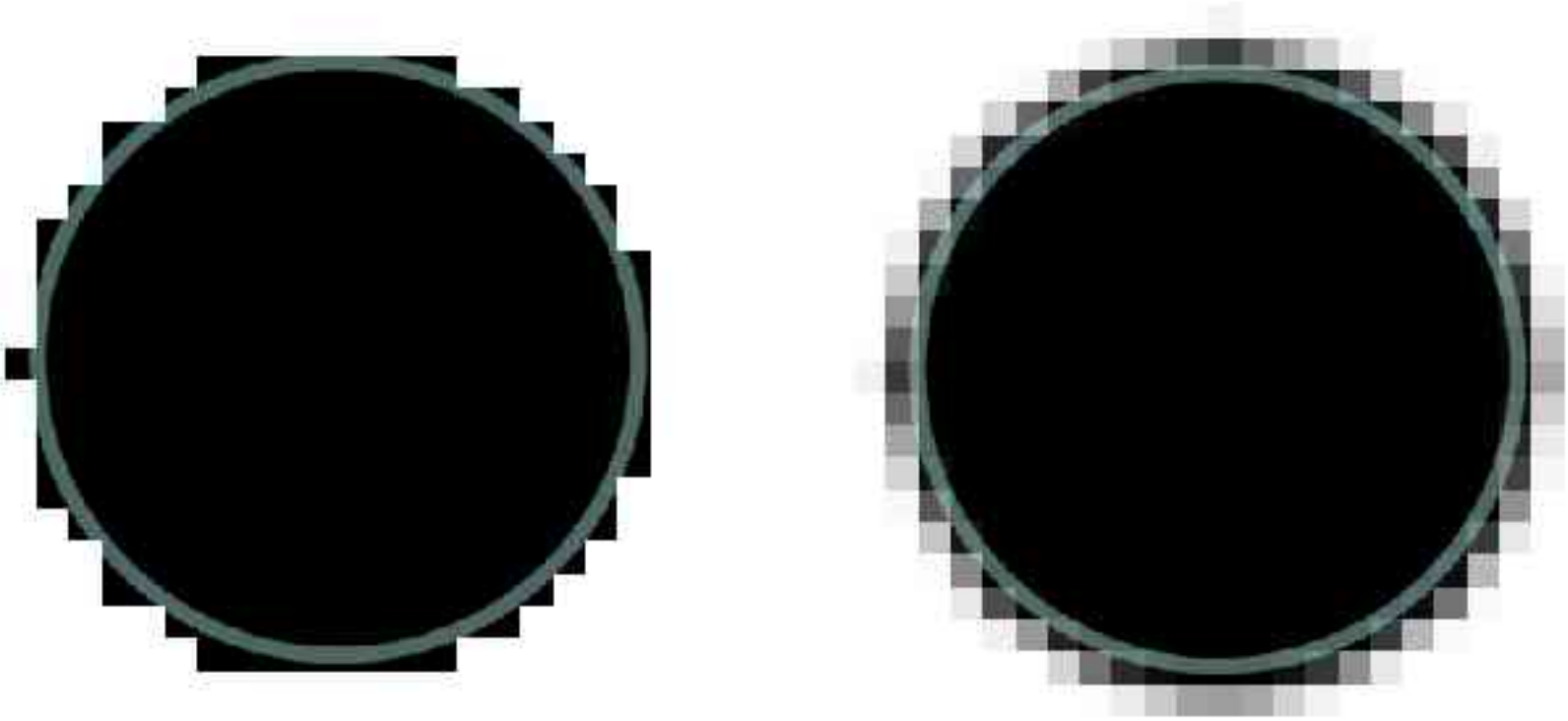
Simple
perspective
projected texture



With some
antialiasing



Antialiasing

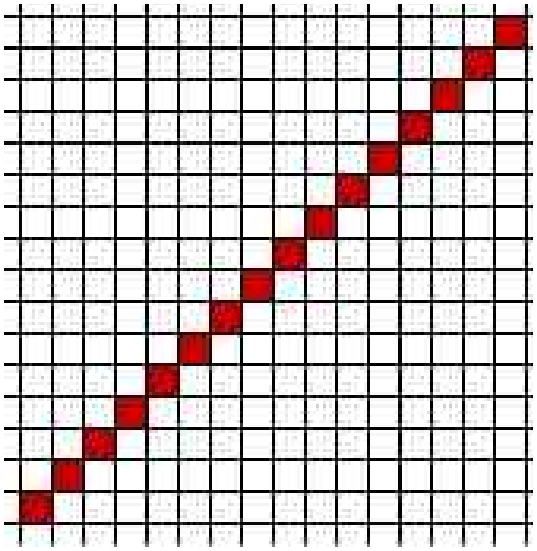


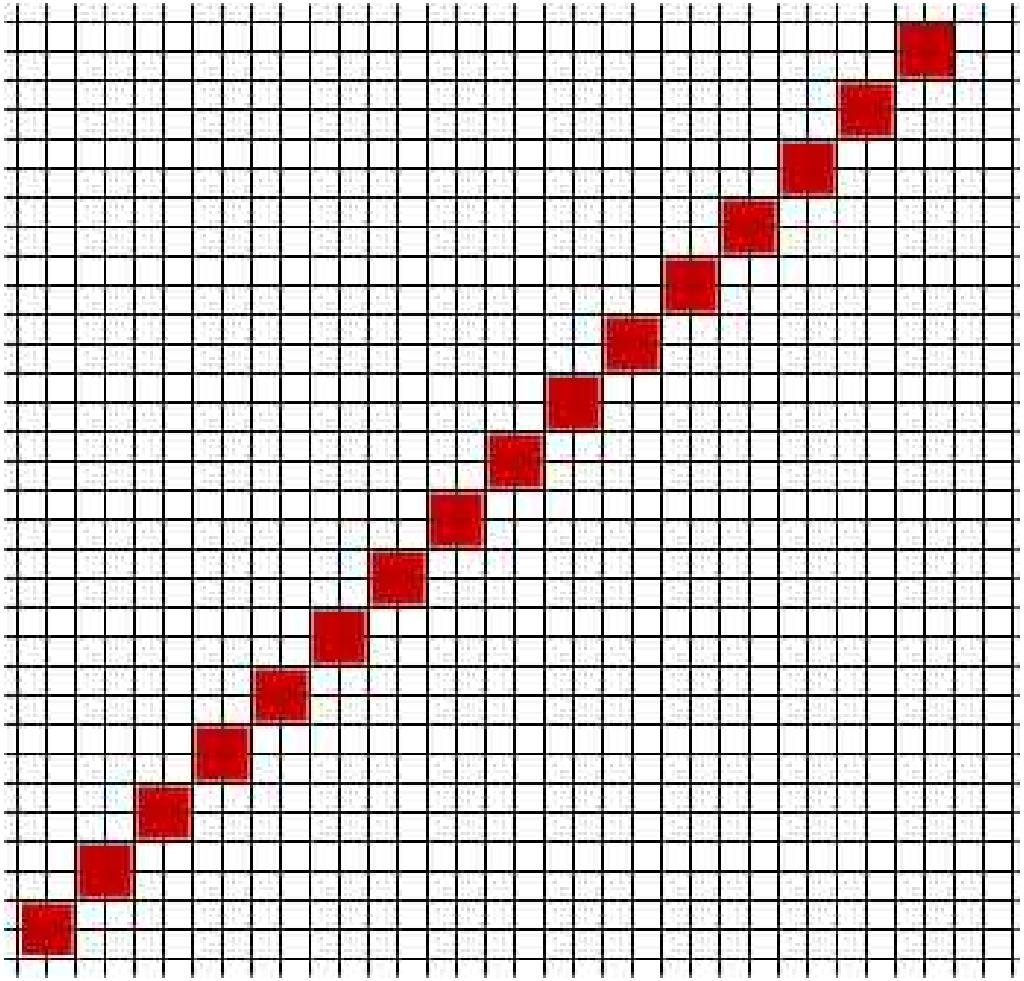
Effectively: BLUR edges to make them look smooth

Supersampling Example

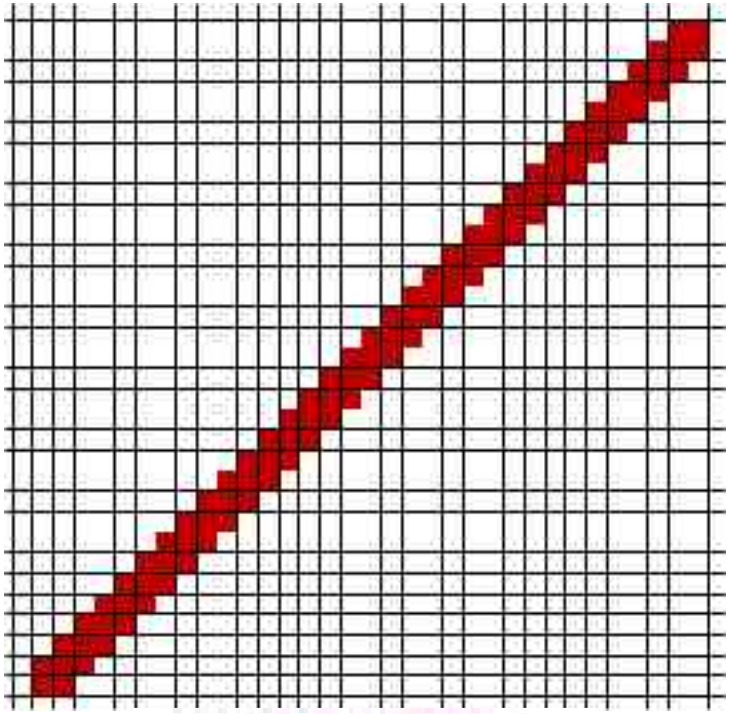
Use twice the available resolution

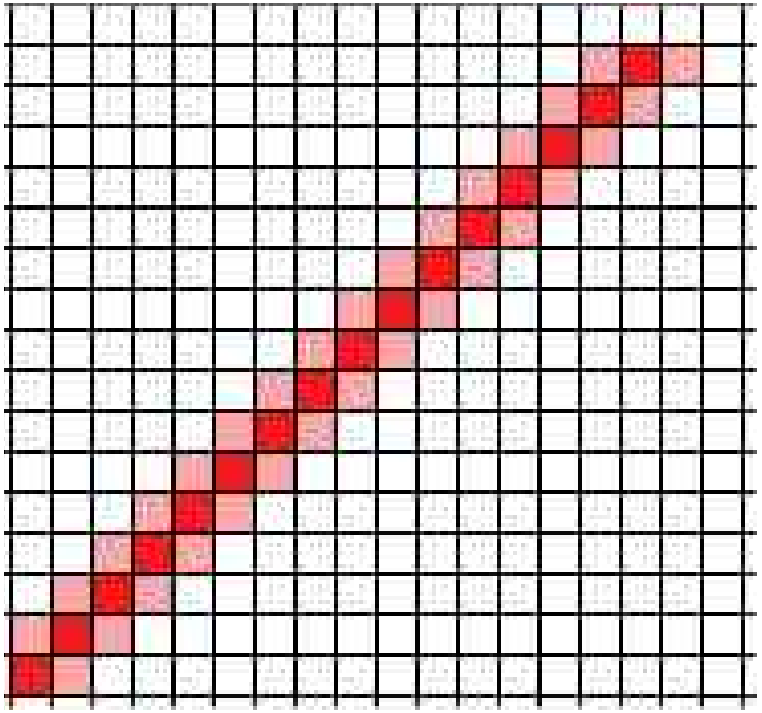
4 pixels (2x2 square) for every 1 in the original

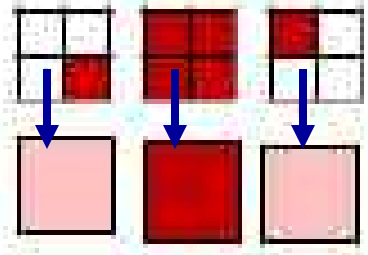


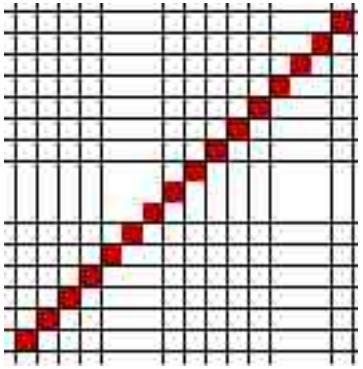


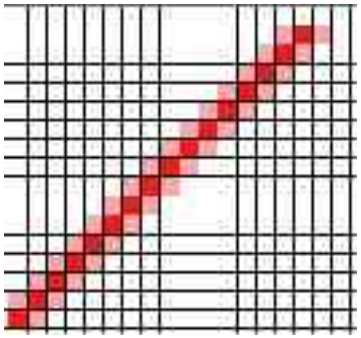
But we don't actually render this hi-res line. Instead we *downsample* it using a "trick"





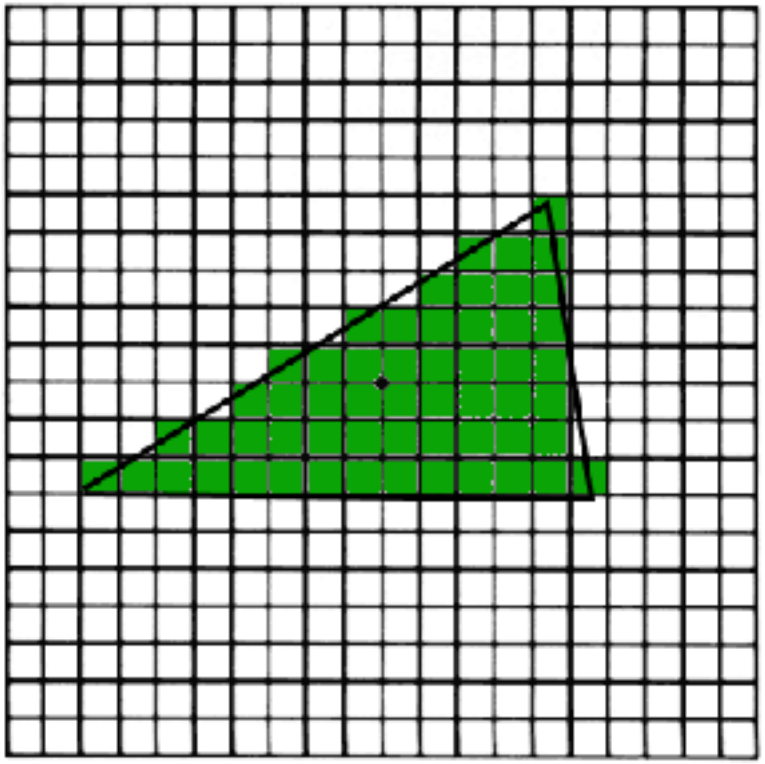


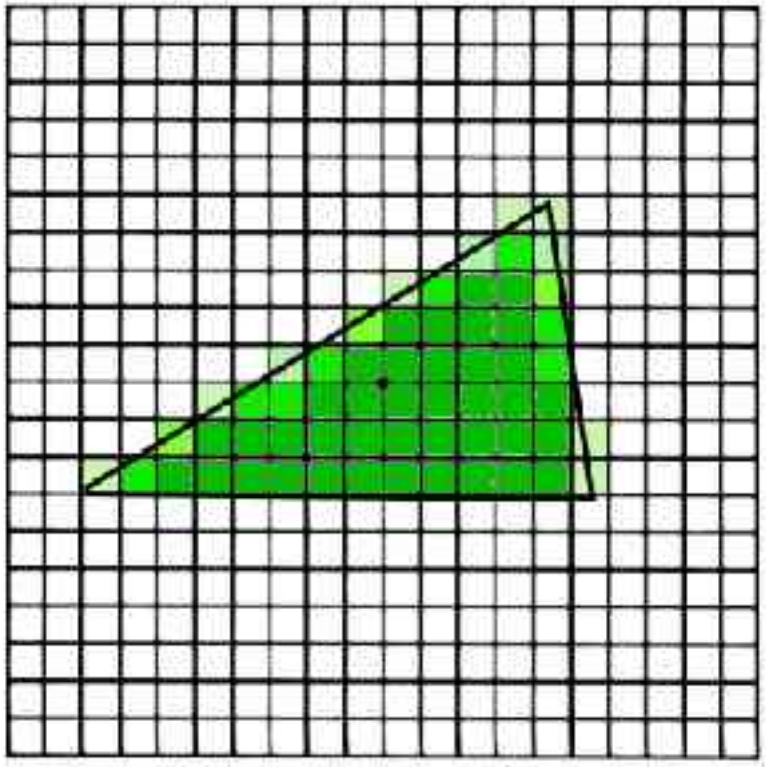




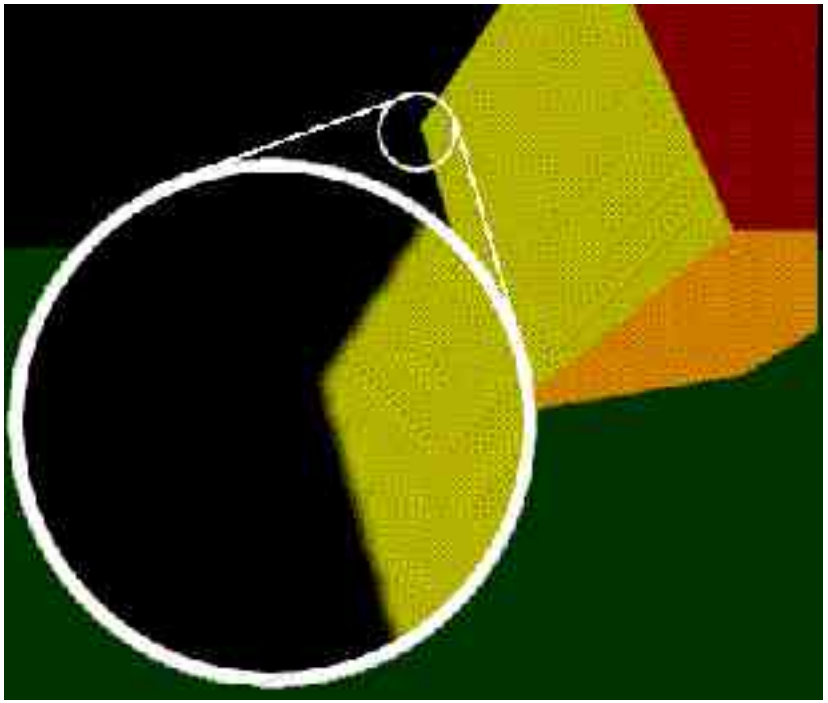
Avarage out the colours in the 4 pixel squares

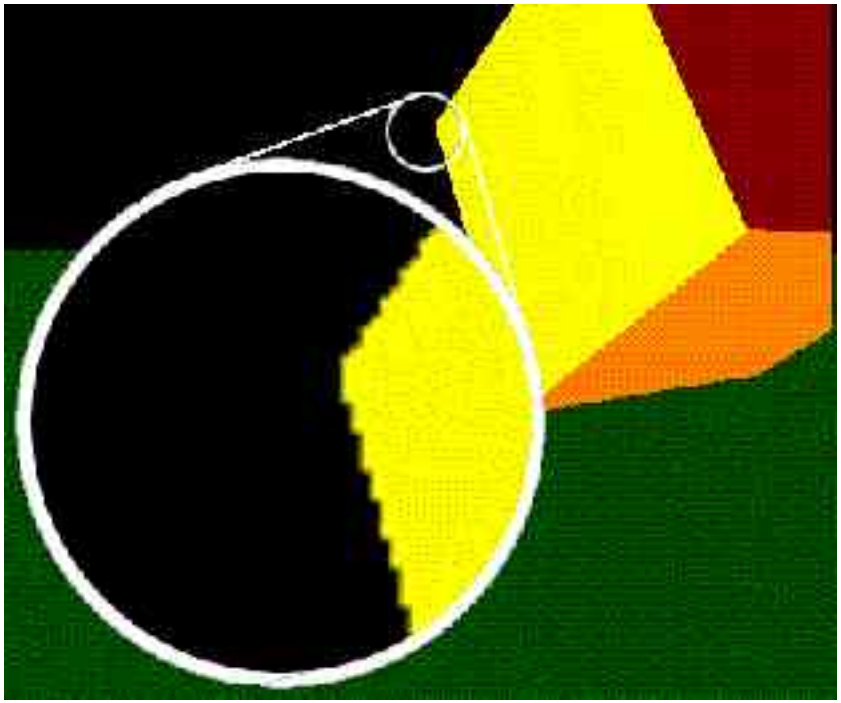
Antialiasing by polygon boundary





Anti-aliasing with Accumulation





FSAA Example