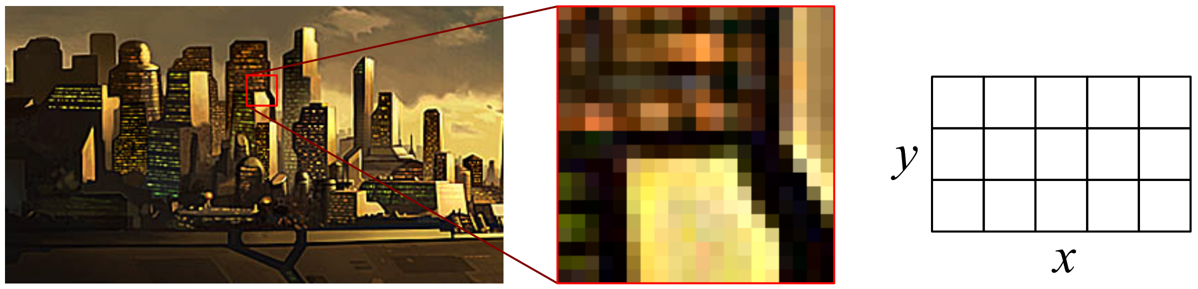
1 Introduction to Graphics

1.1 Raster Displays

The screen is represented by a 2D array of locations called **pixels**.



Zooming in on an image made up of pixels

The convention in these notes will follow that of OpenGL, placing the origin in the lower left corner, with that pixel being at location $(0, 0)$. Be aware that placing the origin in the upper left is another common convention.

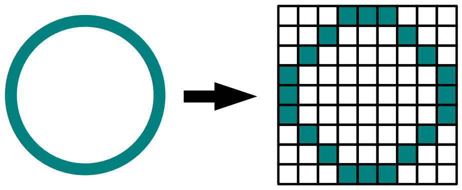
One of 2^N intensities or colors are associated with each pixel, where N is the number of bits per pixel. Greyscale typically has one byte per pixel, for $2^8 = 256$ intensities. Color often requires one byte per channel, with three color channels per pixel: red, green, and blue.

Color data is stored in a **frame buffer**. This is sometimes called an image map or bitmap.

Primitive operations:

- `setpixel(x, y, color)`
  Sets the pixel at position (x, y) to the given color.
- `getpixel(x, y)`
  Gets the color at the pixel at position (x, y).

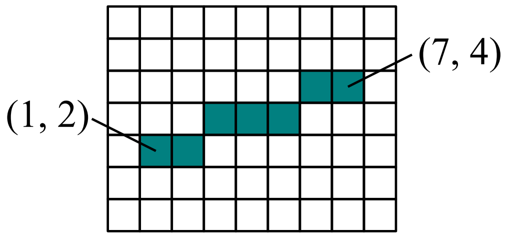
Scan conversion is the process of converting basic, low level objects into their corresponding pixel map representations. This is often an approximation to the object, since the frame buffer is a discrete grid.



Scan conversion of a circle

1.2 Basic Line Drawing

Set the color of pixels to approximate the appearance of a line from (x_0, y_0) to (x_1, y_1).



It should be

- "straight" and pass through the end points.
- independent of point order.
- uniformly bright, independent of slope.

The explicit equation for a line is $y = mx + b$.

Note:
Given two points (x_0, y_0) and (x_1, y_1) that lie on a line, we can solve for m and b for the line. Consider $y_0 = mx_0 + b$ and $y_1 = mx_1 + b$.
Subtract y_0 from y_1 to solve for $m = \frac{y_1 - y_0}{x_1 - x_0}$ and $b = y_0 - mx_0$.
Substituting in the value for b, this equation can be written as $y = m(x - x_0) + y_0$.

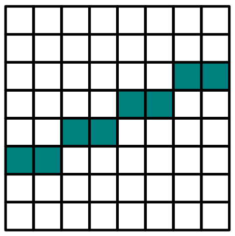
Consider this simple line drawing algorithm:

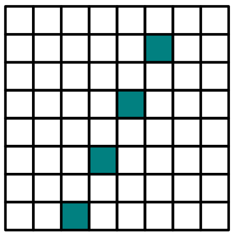
```
int x
float m, y
m = (y1 - y0) / (x1 - x0)
for (x = x0; x <= x1; ++x) {
  y = m * (x - x0) + y0
  setpixel(x, round(y), linecolor)
}
```

Problems with this algorithm:

- If $x_1 < x_0$ nothing is drawn.
  Solution: Switch the order of the points if $x_1 < x_0$.

- Consider the cases when $m < 1$ and $m > 1$:


(a) $m < 1$

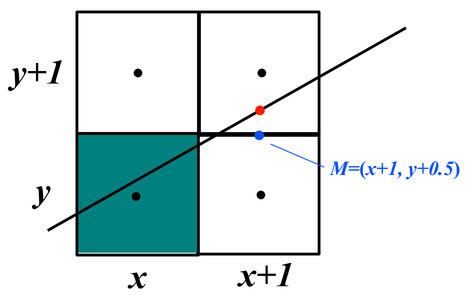

(b) $m > 1$

  A different number of pixels are on, which implies different brightness between the two.
  Solution: When $m > 1$, loop over $y = y_0 \dots y_1$ instead of x, then $x = \frac{1}{m}(y - y_0) + x_0$.

- Inefficient because of the number of operations and the use of floating point numbers.
  Solution: A more advanced algorithm, called Bresenham's Line Drawing Algorithm.

1.3 Bresenham's Algorithm

Bresenham's algorithm is an efficient incremental integer algorithm for line rasterization. It is based on the **midpoint rule**. Midpoint rule states that if true line at $x_i + 1$ is above midpoint $M = (x_i + 1, y_i + \frac{1}{2})$ (closer to $y_i + 1$) then $(x_i + 1, y_i + 1)$ should be drawn next, otherwise if true line at $x_i + 1$ is bellow midpoint M (closer to y_i) then $(x_i + 1, y_i)$ should be drawn next.



How do we check if the true line is above or bellow the midpoint using only integer operations? We use implicit equation for the line,

$$y = \frac{H}{W}(x - x_0) + y_0 \quad (1)$$
$$Wy = H(x - x_0) + Wy_0 \quad (2)$$
$$f(x, y) = 0 = 2H(x - x_0) - 2W(y - y_0) \quad (3)$$

where $H = y_1 - y_0$ and $W = x_1 - x_0$ is the numerator and denominator of the slope ($W \geq H > 0$). All we need to do is evaluate $f(x_i, y_i)$ at the midpoint (i.e. $f(x_i + 1, y_i + 0.5)$).

Note:
if $f(x, y) = 0$ then (x, y) on the line
if $f(x, y) < 0$ then (x, y) above the line
if $f(x, y) > 0$ then (x, y) below the line

This can be checked by holding the x fixed and changing y up or down.

Since all we need to do is keep track of $f(x, y)$ at the midpoints, this can be done incrementally:

- $f(x + 1, y) = f(x, y) + 2H$ (when y stays the same)
- $f(x + 1, y) = f(x, y) + 2(H - W)$ (when y changes)

The algorithm for the line drawing in the first quadrant and with slope < 1 can then be written:

```
int y, H, W, f
y = y0
```

```
H = (y1 - y0)
W = (x1 - x0)
f = 2 * H - W
for (x = x0; x <= x1; ++x) {
  setpixel(x, y, linecolor)
  if (f < 0) {
     f += 2 * H      // y stays the same
  } else {
     y++             // y increases
     f += 2 * (H - W)
  }
}
```

Note:
Initially $f(x_0, y_0) = 0$, so first test is at $f(x_0 + 1, y_0 + \frac{1}{2})$.

1.4 Triangle Rasterization

Find pixels on inside of the triangle $P = \{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$ and fill them with a given color.

Solution: For each horizontal scanline L find intersection of L with P, store the sorted (by increasing values of x) intersections in the active edge list (AEL), and fill in pixel runs between pairs of intersections.

```
for each edge [(x0, y0), (x1, y1)]
  double x, m
  x = x0
  m = (y1 - y0) / (x1 - x0)
  for (y = y0; y <= y1; ++y) {
     x += 1 / m
     place (round(x), y) in AEL
  }
}
```

Problems with this algorithm:

- Horizontal edges are not handled properly.
  Solution: Simply ignore horizontal edges.

Note:
This algorithm will also work for general polygons, except for a few special cases that only happen at vertices of the polygon. All the special cases are about the ownership convention and can be handled by only resterizing the edges (i.e. by looping over y from y_0 to $y_1 - 1$ (instead of to y_1)).