Graphics hardware
The problem of scan conversion
Considerations
Line equations
Scan converting algorithms
  – A very simple solution
  – The DDA algorithm, Bresenham algorithm
Open GL
Conclusion
It’s worth taking a little look at how graphics hardware works before we go any further. How do things end up on the screen?
Architecture Of A Graphics System

System Bus

CPU

Display Processor Memory

Frame Buffer

Video Controller

Monitor

System Memory
Output Devices

There are a range of output devices currently available:

– Printers/plotters
– Cathode ray tube displays
– Plasma displays
– LCD displays
– 3 dimensional viewers
– Virtual/augmented reality headsets

We will look briefly at some of the more common display devices
Basic Cathode Ray Tube (CRT)

Fire an electron beam at a phosphor coated screen

Draw one line at a time

An electron gun for each colour – red, green and blue
Applying voltages to crossing pairs of conductors causes the gas (usually a mixture including neon) to break down into a glowing plasma of electrons and ions.
Light passing through the liquid crystal is twisted so it gets through the polarizer.

A voltage is applied using the crisscrossing conductors to stop the twisting and turn pixels off.
A line segment in a scene is defined by the coordinate positions of the line end-points.
But what happens when we try to draw this on a pixel based display?

How do we choose which pixels to turn on?
Considerations to keep in mind:

– The line has to look good
  • Avoid *jaggies*

– It has to be lightening fast!
  • How many lines need to be drawn in a typical scene?
  • This is going to come back to bite us again and again
Let’s quickly review the equations involved in drawing lines

Slope-intercept line equation:

\[ y = m \cdot x + b \]

where:

\[ m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \]

\[ b = y_0 - m \cdot x_0 \]
The slope of a line \((m)\) is defined by its start and end coordinates.

The diagram below shows some examples of lines and their slopes.
We could simply work out the corresponding y coordinate for each unit x coordinate.

Let’s consider the following example:
A Very Simple Solution (cont...)

Diagram of a grid with numbers from 0 to 7 along the edges.
First work out $m$ and $b$:

$$m = \frac{5 - 2}{7 - 2} = \frac{3}{5}$$

$$b = 2 - \frac{3}{5} \times 2 = \frac{4}{5}$$

Now for each $x$ value work out the $y$ value:

$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2 \frac{3}{5}$$

$$y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3 \frac{1}{5}$$

$$y(5) = \frac{3}{5} \cdot 5 + \frac{4}{5} = 3 \frac{4}{5}$$

$$y(6) = \frac{3}{5} \cdot 6 + \frac{4}{5} = 4 \frac{2}{5}$$
Now just round off the results and turn on these pixels to draw our line

\[
\begin{align*}
y(3) &= 2 \frac{3}{5} \approx 3 \\
y(4) &= 3 \frac{1}{5} \approx 3 \\
y(5) &= 3 \frac{4}{5} \approx 4 \\
y(6) &= 4 \frac{2}{5} \approx 4
\end{align*}
\]
However, this approach is just way too slow

In particular look out for:

- The equation $y = mx + b$ requires the multiplication of $m$ by $x$
- Rounding off the resulting $y$ coordinates

We need a faster solution
In the previous example we chose to solve the parametric line equation to give us the \( y \) coordinate for each unit \( x \) coordinate.

What if we had done it the other way around?

So this gives us:

\[
x = \frac{y - b}{m}
\]

where:

\[
m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0}
\]

and

\[
b = y_0 - m \cdot x_0
\]
Leaving out the details this gives us:

\[ x(3) = 3 \frac{2}{3} \approx 4 \quad x(4) = 5 \frac{1}{3} \approx 5 \]

We can see easily that this line doesn’t look very good!

We choose which way to work out the line pixels based on the slope of the line.
A Quick Note About Slopes (cont...)

If the slope of a line is between -1 and 1 then we work out the y coordinates for a line based on it’s unit x coordinates.

Otherwise we do the opposite – x coordinates are computed based on unit y coordinates.
A Quick Note About Slopes (cont...)

Diagram of a grid with labeled coordinates.
The digital differential analyzer (DDA) algorithm takes an incremental approach in order to speed up scan conversion.

Simply calculate $y_{k+1}$ based on $y_k$.

The original differential analyzer was a physical machine developed by Vannevar Bush at MIT in the 1930’s in order to solve ordinary differential equations.

More information here.
Consider the list of points that we determined for the line in our previous example:

\[(2, 2), (3, \frac{2^3}{5}), (4, \frac{3^1}{5}), (5, \frac{3^4}{5}), (6, \frac{4^2}{5}), (7, 5)\]

Notice that as the $x$ coordinates go up by one, the $y$ coordinates simply go up by the slope of the line.

This is the key insight in the DDA algorithm.
The DDA Algorithm (cont...)

When the slope of the line is between -1 and 1 begin at the first point in the line and, by incrementing the $x$ coordinate by 1, calculate the corresponding $y$ coordinates as follows:

$$y_{k+1} = y_k + m$$

When the slope is outside these limits, increment the $y$ coordinate by 1 and calculate the corresponding $x$ coordinates as follows:

$$x_{k+1} = x_k + \frac{1}{m}$$
Again the values calculated by the equations used by the DDA algorithm must be rounded to match pixel values.

\[
\begin{align*}
(x, y) & \rightarrow (x, \text{round}(y)) \\
(x, y) & \rightarrow (x+1, \text{round}(y+m)) \\
(x+1, \text{round}(y)) & \rightarrow (x+1, y+m) \\
(\text{round}(x, y)) & \rightarrow (\text{round}(x, y) + \frac{1}{m}, y) \\
(\text{round}(x, y)) & \rightarrow (\text{round}(x+\frac{1}{m}, y), y+1) \\
\end{align*}
\]
Let’s try out the following examples:
DDA Algorithm Example (cont...)

Diagram of DDA Algorithm Example.
The DDA Algorithm Summary

The DDA algorithm is much faster than our previous attempt

- In particular, there are no longer any multiplications involved

However, there are still two big issues:

- Accumulation of round-off errors can make the pixelated line drift away from what was intended
- The rounding operations and floating point arithmetic involved are time consuming
Open Graphics Library (OpenGL) is cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU). Explore: https://www.opengl.org/
OpenGL (GLUT) on Linux (Ubuntu)

Installation

Install the following packages from the ubuntu repository:
1. freeglut3-dev
2. mesa-common-dev

```
sudo apt-get install freeglut3 freeglut3-dev mesa-common-dev
```

Check your /usr/include/GL folder to verify the installation of the openGL headers that you intend to use.

Note: sudo add-apt-repository main universe restricted multiverse; sudo apt-get update
// Header Files
#include <math.h>
#include <GL/gl.h>
#include <GL/glut.h>
int main(int argc, char ** argv)
{
    glutInit(&argc, argv);  // Initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // set Display mode
    glutInitWindowSize(640, 480);  // set window size
    glutInitWindowPosition(100, 150);  // set window position on screen
    glutCreateWindow("Dot Plot of a Line");  // open the screen window
    glutDisplayFunc(myDisplay);  // register redrtaw function
    myInit();
    glutMainLoop();  // go into a perpetual loop
    return 0;
}
void myInit(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0); // set white background color
    glColor3f(0.0f, 0.0f, 0.0f); // set the drawing color (black)
    glPointSize(4.0); // a 'dot' is 4 by 4 pixels
    glMatrixMode(GL_PROJECTION); // set camera shape
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble)screenWidth, 0.0, (GLdouble)screenHeight);
}

void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT); // clear the screen
    lineDDA(10, 10, 400, 400);
    lineDDA(300, 100, 300, 400);
    glFlush(); // send all output to display
}
void drawDot(GLint x, GLint y)
{
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}
void lineDDA(float x1, float y1, float x2, float y2)
{
    float x, y, dx, dy, pixel;
    int i;
    dx = fabs(x2 - x1);
    dy = fabs(y2 - y1);

    if(dx >= dy)
        pixel = dx;
    else
        pixel = dy;

    dx = dx / pixel;
    dy = dy / pixel;
x = x1;
y = y1;

i=1;
while(i <= pixel)
{
    drawDot((GLint)(x+0.5), (GLint)(y+0.5));
    x = x + dx;
    y = y + dy;
    i = i + 1;
}
}
Compiling and Linking

You will have to use the -lglut linker option with gcc/g++ to compile a program with glut library. For example, to compile the program test.cpp that uses GLUT type, use

gcc -o test test.cpp -lglut -lGL -lGLU -lm

to get the binary executable.
Open GL: execution
The Bresenham Line Algorithm

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that it uses only integer calculations.

Jack Bresenham worked for 27 years at IBM before entering academia. Bresenham developed his famous algorithms at IBM in the early 1960s.
The Big Idea

Move across the $x$ axis in unit intervals and at each step choose between two different $y$ coordinates.

For example, from position $(2, 3)$ we have to choose between $(3, 3)$ and $(3, 4)$.

We would like the point that is closer to the original line.
At sample position $x_k + 1$ the vertical separations from the mathematical line are labelled $d_{\text{upper}}$ and $d_{\text{lower}}$.

The $y$ coordinate on the mathematical line at $x_k + 1$ is:

$$y = m(x_k + 1) + b$$
So, $d_{upper}$ and $d_{lower}$ are given as follows:

$$d_{lower} = y - y_k$$

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

and:

$$d_{upper} = (y_k + 1) - y$$

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

We can use these to make a simple decision about which pixel is closer to the mathematical line.
This simple decision is based on the difference between the two pixel positions:

\[ d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1 \]

Let’s substitute \( m \) with \( \Delta y/\Delta x \) where \( \Delta x \) and \( \Delta y \) are the differences between the endpoints:

\[ \Delta x(d_{lower} - d_{upper}) = \Delta x(2 \frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1) \]

\[ = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1) \]

\[ = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \]
So, a decision parameter \( p_k \) for the \( k \)th step along a line is given by:

\[
p_k = \Delta x (d_{lower} - d_{upper})
\]

\[
= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c
\]

The sign of the decision parameter \( p_k \) is the same as that of \( d_{lower} - d_{upper} \)

If \( p_k \) is negative, then we choose the lower pixel, otherwise we choose the upper pixel.
Remember coordinate changes occur along the $x$ axis in unit steps so we can do everything with integer calculations.

At step $k+1$ the decision parameter is given as:

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

Subtracting $p_k$ from this we get:

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$
But, \( x_{k+1} \) is the same as \( x_k + 1 \) so:

\[
p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)
\]

where \( y_{k+1} - y_k \) is either 0 or 1 depending on the sign of \( p_k \).

The first decision parameter \( p_0 \) is evaluated at \((x_0, y_0)\) is given as:

\[
p_0 = 2\Delta y - \Delta x
\]
BRESENHAM’S LINE DRAWING ALGORITHM
(for $|m| < 1.0$)

1. Input the two line end-points, storing the left end-point in $(x_0, y_0)$
2. Plot the point $(x_0, y_0)$
3. Calculate the constants $\Delta x$, $\Delta y$, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and get the first value for the decision parameter as:
   $$p_0 = 2\Delta y - \Delta x$$
4. At each $x_k$ along the line, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and:
   $$p_{k+1} = p_k + 2\Delta y$$
Otherwise, the next point to plot is \((x_k + 1, y_k + 1)\) and:

\[
p_{k+1} = p_k + 2\Delta y - 2\Delta x
\]

5. Repeat step 4 \((\Delta x - 1)\) times

**ACHTUNG!** The algorithm and derivation above assumes slopes are less than 1. For other slopes we need to adjust the algorithm slightly
Let's have a go at this
Let's plot the line from (20, 10) to (30, 18)
First off calculate all of the constants:

- $\Delta x$: 10
- $\Delta y$: 8
- $2\Delta y$: 16
- $2\Delta y - 2\Delta x$: -4

Calculate the initial decision parameter $p_0$:

- $p_0 = 2\Delta y - \Delta x = 6$
Bresenham Example (cont...)

<table>
<thead>
<tr>
<th>k</th>
<th>p_k</th>
<th>(x_{k+1},y_{k+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram illustrates the Bresenham algorithm for drawing lines on a digital display. The table tracks the decision variable $p_k$ and the coordinates $(x_{k+1},y_{k+1})$ for each step of the algorithm.
Go through the steps of the Bresenham line drawing algorithm for a line going from (21,12) to (29,16)
Bresenham Exercise (cont...)

\[
k \quad p_k \quad (x_{k+1}, y_{k+1})
\]

\[
\begin{array}{|c|c|c|}
\hline
0 & & \\
1 & & \\
2 & & \\
3 & & \\
4 & & \\
5 & & \\
6 & & \\
7 & & \\
8 & & \\
\hline
\end{array}
\]
The Bresenham line algorithm has the following advantages:

- An fast incremental algorithm
- Uses only integer calculations

Comparing this to the DDA algorithm, DDA has the following problems:

- Accumulation of round-off errors can make the pixelated line drift away from what was intended
- The rounding operations and floating point arithmetic involved are time consuming
Conclusion

In this lecture we took a very brief look at how graphics hardware works. Drawing lines to pixel based displays is time consuming so we need good ways to do it. The DDA algorithm is pretty good – but we can do better. Next time we’ll like at the Bresenham line algorithm.