Okay, you have learned ...

- OpenGL drawing
- Viewport and World Window setup

```c
main()
{
    glViewport(0, 0, 300, 200);
    gluOrtho2D(-1, 1, -1, 1);
    glBegin(GL_QUADS);
    glColor3f(1, 1, 0);
    glVertex2i(-0.5, -0.5);
    glVertex2i(+0.5, 0);
    glVertex2i(+0.5, +0.5);
    glVertex2i(-0.5, +0.5);
    glEnd();
}
```

**2D Graphics Pipeline**

Graphics processing consists of many stages:

1. **Object Local Coordinates**
2. **Modeling transformation**
3. **Object World Coordinates**

**Clipping and Rasterization**

- OpenGL does these for you – no explicit OpenGL functions needed for doing clipping and rasterization
- **Clipping** - Remove objects that are outside the world window
- **Rasterization (scan conversion)** - Convert high level object descriptions to pixel colors in the frame buffer
2D Point Clipping

Determine whether a point \((x,y)\) is inside or outside of the world window?

If \((xmin <= x <= xmax)\) and \((ymin <= y <= ymax)\) then the point \((x,y)\) is inside else the point is outside.

2D Line Clipping

Determine whether a line is inside, outside, or partially inside.

If a line is partially inside, we need to display the inside segment.

Trivial Accept Case

Lines that are clearly inside the world window - what are they?

\[ X_{min} <= P1.x, P2.x <= xmax \]
\[ Y_{min} <= P1.y, P2.y <= ymax \]

Trivial Reject Case

Lines that are clearly outside the world window - what are they?

\[ p1.x, p2.x <= X_{min} \text{ OR} \]
\[ p1.x, p2.x >= X_{max} \text{ OR} \]
\[ p1.y, p2.y <= y_{min} \text{ OR} \]
\[ p1.y, p2.y >= y_{max} \]
Non-Trivial Cases

- Lines that cannot be trivially rejected or accepted
  - One point inside, one point outside
  - Both points are outside, but not “trivially” outside
- Need to find the line segments that are inside

Non-trivial case clipping

- Compute the line/window boundary edges intersection
- There will be four intersections, but only one or two are on the window edges
- These two points are the end points of the desired line segment

Rasterization (Scan Conversion)

- Convert high-level geometry description to pixel colors in the frame buffer

Rasterization Algorithms

- A fundamental computer graphics function
- Determine the pixels’ colors, illuminations, textures, etc.
- Implemented by graphics hardware
- Rasterization algorithms
  - Lines
  - Circles
  - Triangles
  - Polygons
**Rasterize Lines**

- Why learn this?
  - Understand the discrete nature of computer graphics
  - Write pure device independent graphics programs (Palm graphics)
  - Become a graphics system developer

**Line Drawing Algorithm (1)**

- Given two end points \((x_0, y_0), (x_1, y_1)\), how to compute \(m\) and \(b\)?

\[
m = \frac{y_1 - y_0}{x_1 - x_0} = \frac{dy}{dx}
\]

\[
b = y_1 - m \cdot x_1
\]

**Line Drawing Algorithm (2)**

- Slope-intercept line equation
  - \(Y = mx + b\)

**Line Drawing Algorithm (3)**

- Given the line equation \(y = mx + b\), and end points \((x_0, y_0)\), \((x_1, y_1)\)
- Walk through the line: starting at \((x_0, y_0)\)
- If we choose the next point in the line as \(X = x_0 + \Delta x\)
  
  \[
  Y = y_0 + \Delta x \cdot m = y_0 + \Delta x \cdot \left(\frac{dy}{dx}\right)
  \]
Line Drawing Algorithm (4)

\[ X = x_0 \quad Y = y_0 \]
Illuminate pixel \((x, \text{int}(Y))\)
\[ X = x_0 + 1 \quad Y = y_0 + 1 \times m \]
Illuminate pixel \((x, \text{int}(Y))\)
\[ X = X + 1 \quad Y = Y + 1 \times m \]
Illuminate pixel \((x, \text{int}(Y))\)

\[ \text{Until } X = x_1 \]

Line Drawing Algorithm (5)

- How about a line like this?

  Can we still increment \(X\) by 1 at each Step?
  The answer is No. Why?
  We don't get enough samples
  How to fix it?
  Increment \(Y\)

Line Drawing Algorithm (6)

\[ X = x_0 \quad Y = y_0 \]
Illuminate pixel \((x, \text{int}(Y))\)
\[ Y = y_0 + 1 \quad X = x_0 + 1 \times \frac{1}{m} \]
Illuminate pixel \((x, \text{int}(Y))\)
\[ Y = Y + 1 \quad X = X + 1 \times \frac{1}{m} \]
Illuminate pixel \((x, \text{int}(Y))\)

\[ \text{Until } Y = y_1 \]

Line Drawing Algorithm (7)

- The above is the simplest line drawing algorithm
- Not very efficient
- Optimized algorithms such integer DDA and Bresenhan algorithm (section 8.10) are typically used
- Not the focus of this course