CS3388B: Lecture 5

January 26, 2023

5 Drawing in 2D using Open GL: Part 1

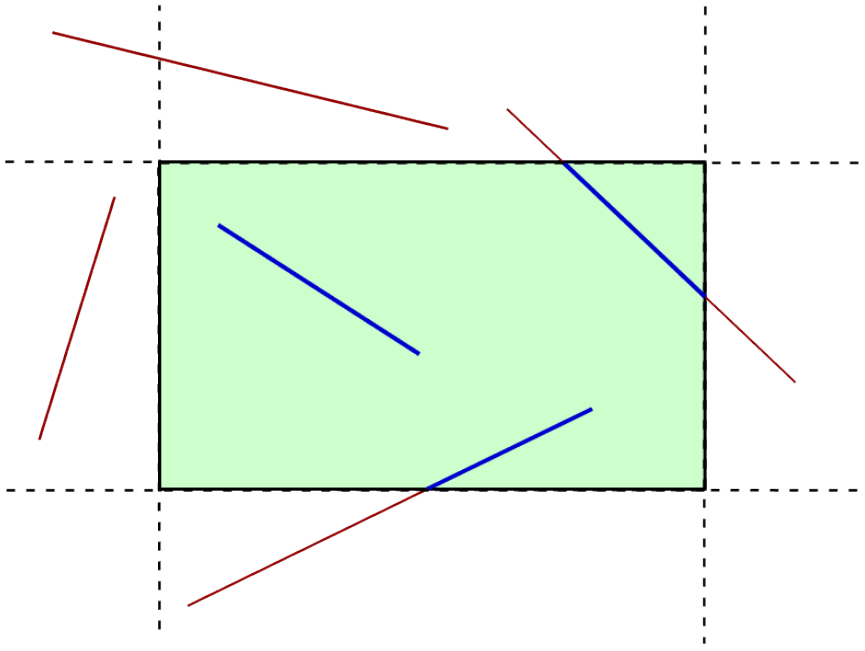
5.1 Review: Establish a coordinate system

We specify the positions of vertices in **world coordinates**. It just so happens that, by default, world coordinates coincide with **Normalized Device Coordinates** because the **projection matrix** is the identity matrix.

Recall we use `glOrtho` to specify the *viewing volume*. It is the projection matrix used to transfer *from* the viewing volume *to* NDC). For example, if we specify a viewing volume where x ranges between 0 and 100, and y ranges between 0 and 100, then anything we draw at within those coordinates will be displayed on the screen. Anything outside will be *clipped*.

5.2 Line Clipping

Clipping is the process of *excluding* data from being drawn. It is a fundamental operation in graphics. Clipping is more than *just* excluding data. As the name suggests there is some "cutting" going on. If only part of primitive is outside the viewing volume (or viewport) we still want that part that is visible to be seen.



The Cohen-Sutherland algorithm is a classic *line clipping* algorithm developed in the 1960s. The key is to divide screen space into 9 areas, where only the center area is visible.

The algorithm has three simple steps:

1. If the line is entirely inside the viewport, draw it completely.
2. If the line is entirely outside the viewport, draw nothing.
3. Otherwise, *cut* the line at one of the edges of viewport, keep the "inside" part of the cut line, and repeat the above with the new shorter line.

Let's consider the first two cases. How do we determine if a line is *completely outside* or *completely inside* the viewing area?

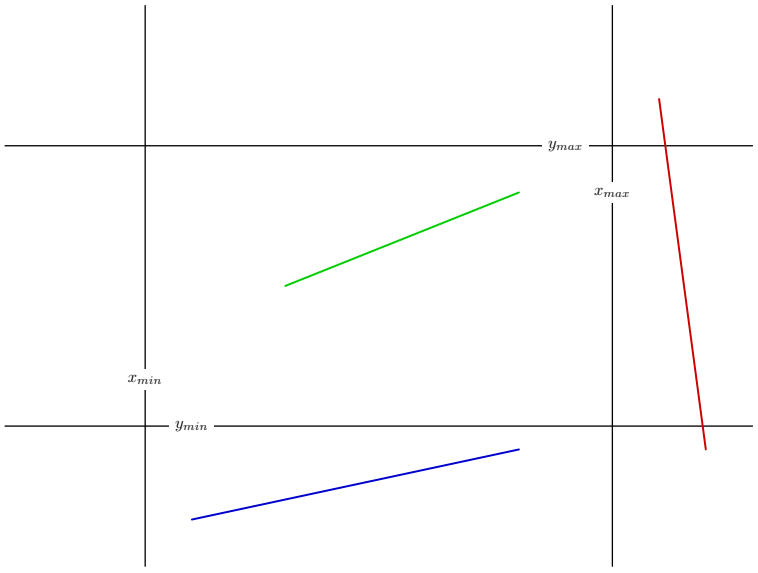
Line (segments) are described by two endpoints, say (x_0, y_0) and (x_1, y_1). The viewport can be described using its two diagonal corners: (x_{min}, y_{min}), (x_{max}, y_{max}).

For a point to be *completely inside the viewport* (e.g. green below):

1. $x_{min} \leq x_0 \leq x_{max}$, and
2. $y_{min} \leq y_0 \leq y_{max}$, and
3. $x_{min} \leq x_1 \leq x_{max}$, and
4. $y_{min} \leq y_2 \leq y_{max}$.

For a point to *necessarily* be *completely outside the viewport* (e.g. red or blue below):

1. ($x_0 < x_{min}$ and $x_1 < x_{min}$) or
2. ($x_0 > x_{max}$ and $x_1 > x_{max}$) or
3. ($y_0 < y_{min}$ and $y_1 < y_{min}$) or
4. ($y_0 > y_{max}$ and $y_1 < y_{max}$) or



Notice that the above test for testing "completely outside" may miss finding some lines which *are* completely outside. In particular, when one endpoint's x-coordinate is "inside" but its y-coordinate is "outside", and vice versa for the other end point. That's fine. It'll get chopped like any other line and then eventually found to be "trivially outside".

The main idea to to chop once, then check using the above conditions if the chopped line is "trivially inside" or "trivially outside". Otherwise, chop it again and repeat the trivial checks.

To do the chopping and the cutting, we have several choices. A classic choice is **TBRL**. which uses the "top" y_{max} line as the knife, then uses the bottom y_{min} line as the knife, then x_{max}, then x_{min}.

We can use classic $y = mx + b$ to determine the new endpoints after cutting. The key is that, when being cut by "top", the endpoint's new y coordinate is y_{max}. When cut by "bottom" the endpoint's new y is y_{min}. Same goes for left and right, but for x.

Then, we can use the line's slope to determine the corresponding x coordinate (when cut by top or bottom) or y coordinate (when cut by right or left).

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Then, from (x_0, y_0), following along a straight line of slope m, and ending at y_{max} must have corresponding x coordinate x_{new}:

$$\frac{y_1 - y_0}{x_1 - x_0} = m = \frac{y_{max} - y_0}{x_{new} - x_0}$$

$$(x_{new} - x_0)(y_1 - y_0) = (y_{max} - y_0)(x_1 - x_0)$$

$$x_{new} - x_0 = (y_{max} - y_0)(x_1 - x_0)/(y_1 - y_0)$$

$$x_{new} = x_0 + (y_{max} - y_0)(x_1 - x_0)/(y_1 - y_0)$$

The the almost same formula can be used to find x_{new} at y_{min} by just replacing y_{max} with y_{min}.

When we are cutting along right or left, the new x-coordinate will be x_{max} or x_{min}, respectively. We can then find y_{new} using the same slope formula:

$$\frac{y_1 - y_0}{x_1 - x_0} = m = \frac{y_{new} - y_0}{x_{max} - x_0}$$

$$(x_{max} - x_0)(y_1 - y_0) = (y_{new} - y_0)(x_1 - x_0)$$

$$y_{new} - y_0 = (x_{max} - x_0)(y_1 - y_0)/(x_1 - x_0)$$

$$y_{new} = y_0 + (x_{max} - x_0)(y_1 - y_0)/(x_1 - x_0)$$

So now we know how to do a cut. Let's make it an algorithm. Remember the main idea is to cut, check, cut, check, cut, check, etc. until the chopped line is "trivially inside" or "trivially outside".

```
//let xmin xmax, ymin, ymax be defined as globals
bool CS_clip (float& x0, float& y0, float& x1, float &y1) {
    while(1) {
        if (line totally inside viewport) {
            return true;
        } else if (line totally outside) {
            return false;
        } else if ((x0,y0) is outside) {

            if (y0 > ymax)  { // top
                x0 = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y0 = ymax;
            } else if (y0 < ymin) { // bottom
                x0 = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y0 = ymin;
            } else if (x0 > xmax) { //right
                y0 = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x0 = xmax;
            } else { //left
                x0 = xmin;
                y0 = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
            }

        } else ((x1,y1) is outside) {
            //repeat above but for (x1,y1)
        }
    }
}
```

5.3 Polygon Clipping

So we can clip lines. Cool. What about *two-dimensional* primitives? What about triangles? Sutherland strikes again. But, honestly, graphics libraries are sophisticated enough now that you never need to know how this works. If you ever need to implement this algorithm yourself, you're certainly doing low-level graphics programming and can figure out this algorithm for yourself. For reference, see this wiki article.

5.4 Dot and Line Plots

Now that we know how to set up a world coordinate system, and potentially clip geometry which is outside our viewport, it's time to actually start drawing stuff (yay!)

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.

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);

}
```

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();
```

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.

5.5 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)`.

```
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.

5.6 Transforms!

We know about the math of affine transforms. Now let's see them in practice.

Recall that one can view affine transforms as **transforming the coordinate system**, and then vertices are drawn with respect to this modified coordinate system. (The opposite view is that the transforms manipulate the positions of the vertices themselves).

Consider a simple triangle with vertices $(0, 1)$, $(1, 0)$, $(-1, 0)$. With no other modifications, this will be drawn in NDC and thus span the width of the entire window and the top-half.

We saw previously that we can use `glOrtho` to define the viewing volume in world coordinates which then get mapped to NDC. With a viewing volume spanning from -10 to 10 in x and y, the same triangle is not just a small figure with width 2 in a viewing volume of width 20.

But how do we move the triangle around the viewing volume? We could manually compute transformation matrices to get the desired result and then multiply these matrices against each vertex's position. But, this is rather silly.

What makes more sense is to instead keep the vertices in exactly the same place: $(0, 1)$, $(1, 0)$,

$(-1, 0)$, and instead move the coordinate system around.

Say we want to draw the triangle so that it's upper point is at $(5, 6)$ in the world. Since this point is originally at $(0, 1)$, the corresponding displacement vector is $(5, 5)$. We **translate** the coordinate system by $(5, 5)$, draw the triangle with its pre-defined vertex positions, and then undo the translation to get the original coordinate system back.

In code we can do this using `glTranslatef(x,y,z)`

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-10, 10, -10, 10, -1, 1);

glTranslatef(5, 5, 0);
glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 1.0f);
    glVertex2f(-1.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);
glEnd();
glTranslatef(-5, -5, 0);
```

Now, remember two things:

1. OpenGL is a global state machine
2. `glOrtho`, and all transforms, *multiply* the current matrix by their corresponding transformation matrix

So, in the above code we are modifying the projection matrix to translate the triangle... not a great (although it will get you the correct result.. for now).

Instead, let's use a new matrix: `GL_MODELVIEW`.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-10, 10, -10, 10, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(5, 5, 0);
glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 1.0f);
    glVertex2f(-1.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);
glEnd();
glTranslatef(-5, -5, 0);
```

In OpenGL the *modelview matrix* is applied *before* the projection matrix.

$$v_{ndc} = PV\,v_{local}$$

The modelview matrix transforms an object's *local coordinates* to *world coordinates*. Or, another way of thinking about it, it (temporarily) moves around world coordinate system.

To see the power of the modelview matrix, let's try drawing two triangles. One whose tip is at $(6, 5)$ and another whose tip is at $(-4, -5)$.

```
void drawTriangle() {
    glBegin(GL_TRIANGLES);
        glVertex2f(0.0f, 1.0f);
        glVertex2f(-1.0f, 0.0f);
        glVertex2f(1.0f, 0.0f);
    glEnd();
}

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-10, 10, -10, 10, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(5, 5, 0);
drawTriangle();

glLoadIdentity();
glTranslatef(-5, -5, 0);
drawTriangle();
```

So, we can always move around the coordinate system and draw vertices with respect to the moved coordinate system. We don't have to modify any of the object's vertices directly.

We can apply many transformations in this way:

- `glTranslatef`
- `glRotatef`
- `glScalef`