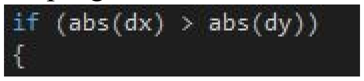
Computer Graphics HW3

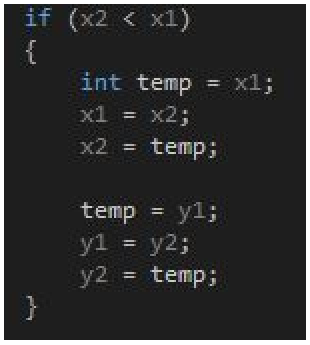
Cameron Jackson u0927597

Task 1:
The most difficult part of the line drawing algorithm was handling all the different cases. First the program has to determine which axis to iterate over by comparing |dx| and |dy|:



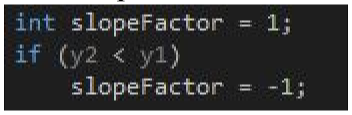
```
if (abs(dx) > abs(dy))
{
```

Then it needs to decide which vertex to start from, swapping the two if necessary:



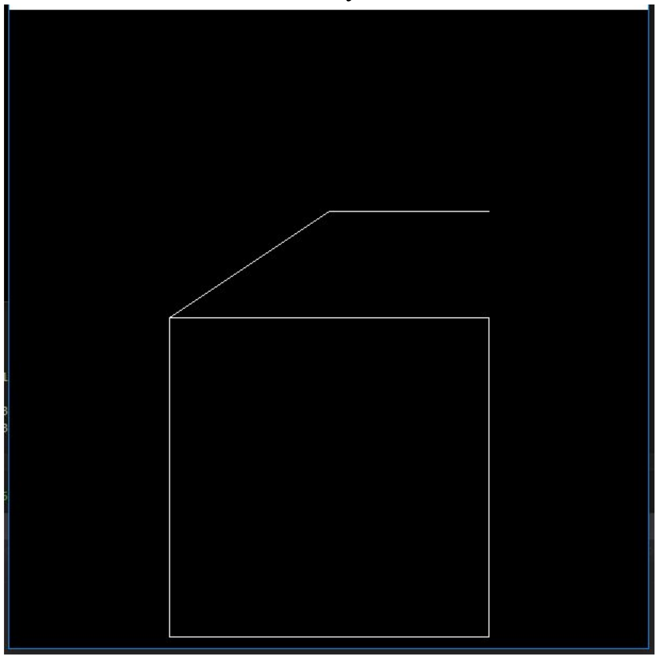
```
if (x2 < x1)
{
    int temp = x1;
    x1 = x2;
    x2 = temp;

    temp = y1;
    y1 = y2;
    y2 = temp;
}
```

Then it needs to decide whether the slope of the line is positive or negative. I handled this with an int slopeFactor that is either 1 or -1 depending on the slope of the line.

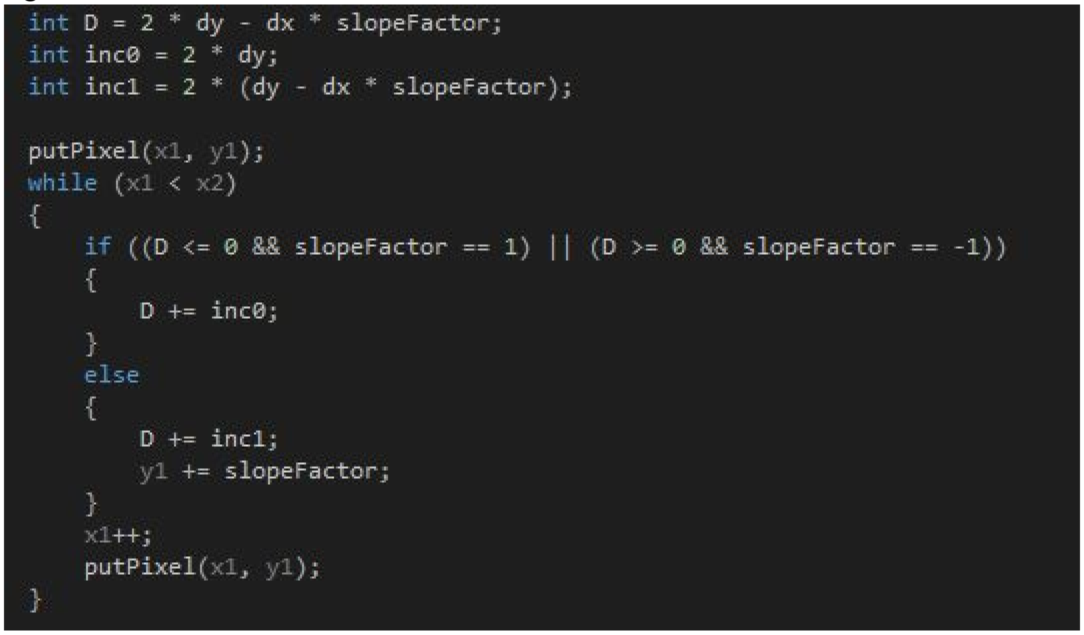


```
int slopeFactor = 1;
if (y2 < y1)
    slopeFactor = -1;
```

This is the is what the non-dominant axis gets incremented by every time it needs to change. It is also used to initially calculate D and inc1. Originally, I didn't have this which caused the negative line to be horizontal because D was always less than 0:

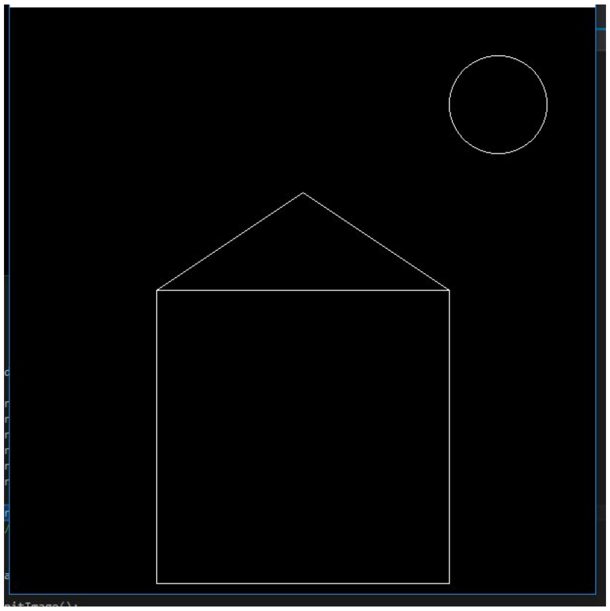


Once all of this is determined, my code runs a slightly modified version of the Bresenham algorithm from the slides.

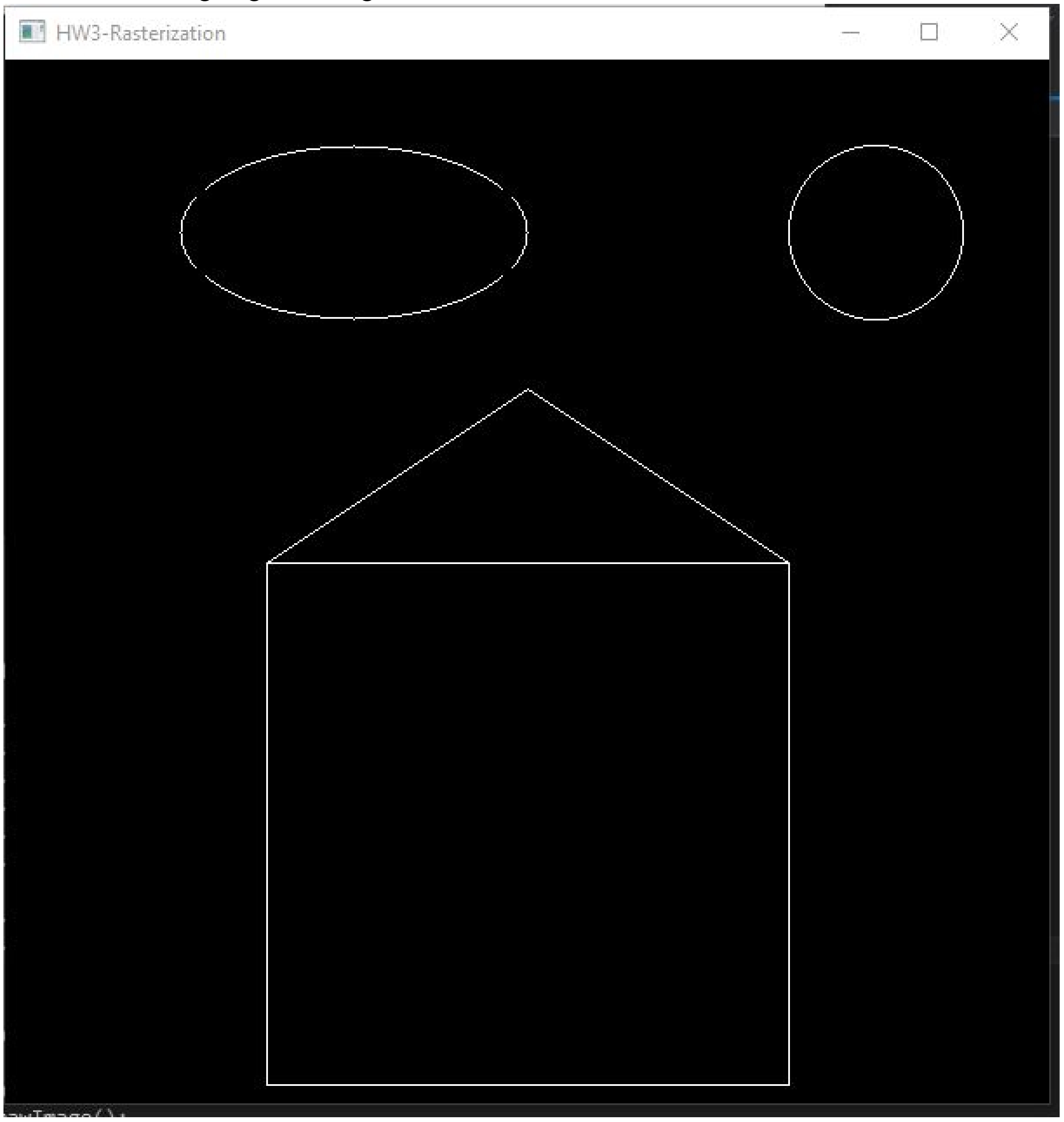


```
int D = 2 * dy - dx * slopeFactor;
int inc0 = 2 * dy;
int inc1 = 2 * (dy - dx * slopeFactor);

putPixel(x1, y1);
while (x1 < x2)
{
    if ((D <= 0 && slopeFactor == 1) || (D >= 0 && slopeFactor == -1))
    {
        D += inc0;
    }
    else
    {
        D += inc1;
        y1 += slopeFactor;
    }
    x1++;
    putPixel(x1, y1);
}
```

Task2:
Rasterizing the circle was much easier because there is only one case to worry about, since most of the circle can be constructed through symmetry. I basically just adapted the algorithm straight from the slides:

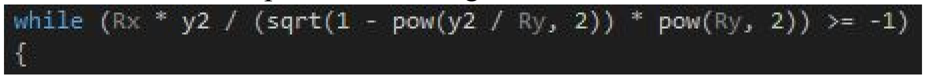


Extra:
I also tried to rasterize an ellipse in the same manner, which is substantially more difficult than a circle because symmetry only works for quarters of the circle, not eighths. This means I had to separately rasterize the two halves of each quadrant. The problem is that I couldn't find a good way to determine when to stop and begin rasterizing the other half. For example, I tried having an int that counts the number of consecutive iterations where D > 0 and stopping when it reached a certain amount. This worked for a few cases, but a lot of times it would either leave gaps because it wasn't going far enough:

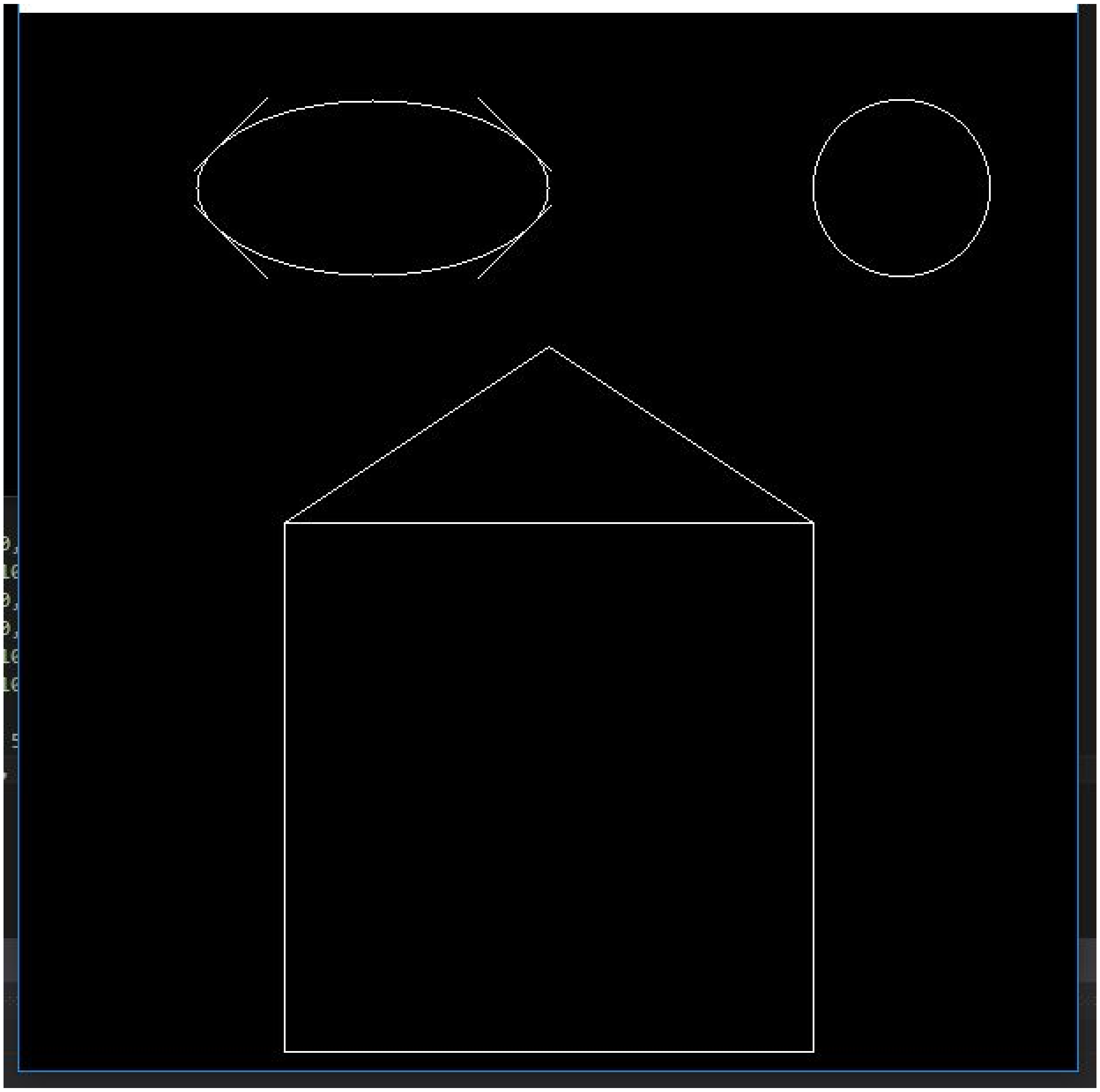


Or have diagonal lines coming off because it was going too far.

The closest I got was by taking the derivative of the curve of my ellipse and comparing that to -1, which should be the point that the algorithm should switch over:



```
while (Rx * y2 / (sqrt(1 - pow(y2 / Ry, 2)) * pow(Ry, 2)) >= -1)
{
```



I feel like there must be a simpler way of doing this, but I haven't been able to come up with one.