图形学实验PA3： 光栅图形学

实验目的

1. become familiar with the concept of Raster Graphics
2. become familiar with content of Chapter 2 of 《计算机图形学基础教程》
3. realize algorithms to draw a straight line, circle, and area filling algorithm

实验逻辑

There were three TODO's that needed to be completed to realize the three functionality. We had to implement a line drawing algorithm, circle drawing algorithm, and an area filling algorithm.

The line drawing algorithm was quite simple, it is based on Bresenham's line algorithm that determines the points of the raster that should be selected in order to form as close of an approximation to a straight line between two points. It is also known as an incremental algorithm. The algorithm into two different cases, one where the slope is negative (from high to low) and one where the slope is positive (low to high) listed as

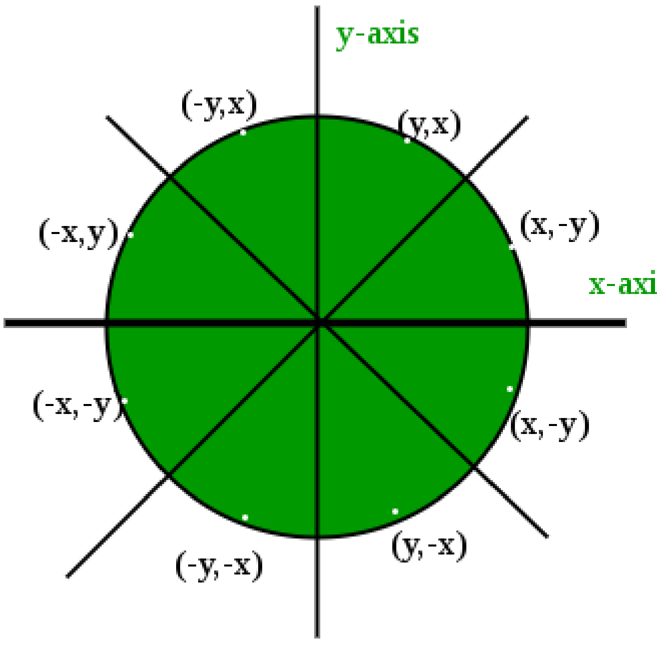
```
plotLineLow(int x0, int y0, int x1, int y1, Image &img, Vector3f color);
plotLineHigh(int x0, int y0, int x1, int y1, Image &img, Vector3f color);
```

However, these two functions alone only consider cases when

$$x0 < x1, y0 < y1$$

In order to consider for cases where this isn't true, we just have to reverse the coordinates before drawing.

The algorithm used for the circle is known as Bresenham's circle algorithm. It uses the key feature that a circle is symmetric. So for a whole 360 degrees, we can divide it into 8 parts, with each octant being 45 degrees. Thus, for every pixel it calculates, we draw a pixel in each of the 8 octant of the circle. This is shown below



where the center of the circle in this case would be (0,0).

The fill algorithm was the hardest out of the three to complete. At first I utilized a recursive algorithm that would continue to check a pixels neighboring 4 pixels (up, right, left, down) and check whether or not it was the same color as the spot where the fill was placed. The code is as shown

```
void floodFill(int x, int y, img, oldcolor, newcolor) {
bool outOfBounds = false;
        if (x < 0|| y < 0 || y > img.Height() || x > img.Width()) outOfBounds = 
true;
        if(!outOfBounds){
            if(img.GetPixel(x,y) == oldcolor) {
                img.SetPixel(x,y,newcolor);
                floodFill(x+1,y,img, oldcolor, newcolor);
                floodFill(x,y+1,img, oldcolor, newcolor);
                floodFill(x-1,y,img, oldcolor, newcolor);
                floodFill(x,y-1,img, oldcolor, newcolor);
            }
        }
      }
```

This method worked for the first drawing, `canvas01` , however, it caused a `segmentation fault (core dumped)` error. This is due to recursive nature of the algorithm, it become to memory expensive, thus causing the error. This error occurred at the second command in the canvs02_emoji.txt input,

```
Fill 255 255 0.95 0.95 0.95
```

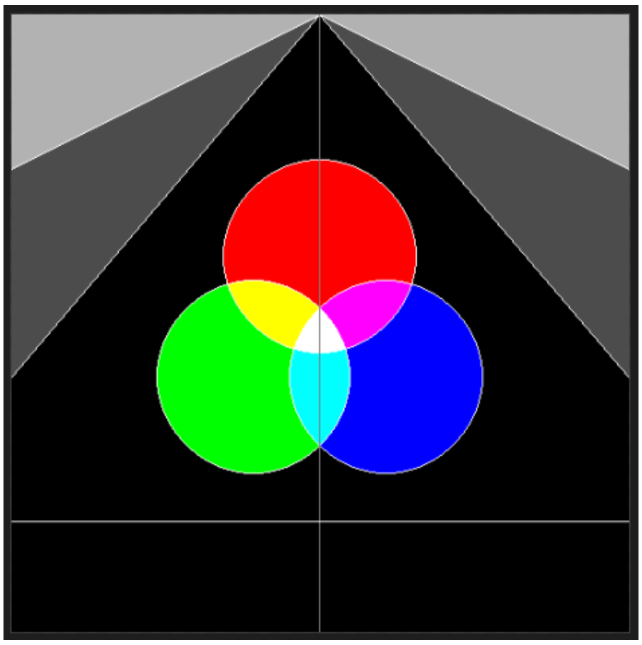
which tells the program to completely fill the canvas with a white background. Removing this line works fine.

In order to get the white background, I used a simple BFS algorithm to keep track of nodes visited and nodes unvisited. It utilized `queue` to keep track of nodes that are yet to be visited, and only add nodes to the `queue` that have not been yet visited, and that fits the correct criteria:

- x and y are both greater than zero
- x and y are both less than the max width and height of the canvas
- the current color of the pixel equals the original color of fill's origin.

This method worked, and the results are shown below.

实验结果



实验建议

没有建议，文档详尽