Assignment 3

Yue Zhang

Project Summary

- Implement five graphics primitives: Point, Line, Circle, Ellipse, and Polyline.

Task 1: Points and Lines

The task aimed to implement and manipulate 2D and 3D graphics in C, focusing on structures and functions for handling lines, points. The purpose was to create a basic graphics library capable of drawing and manipulating geometric shapes.

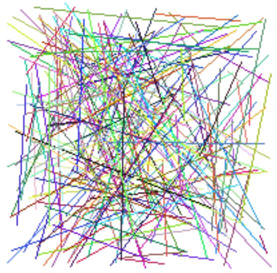
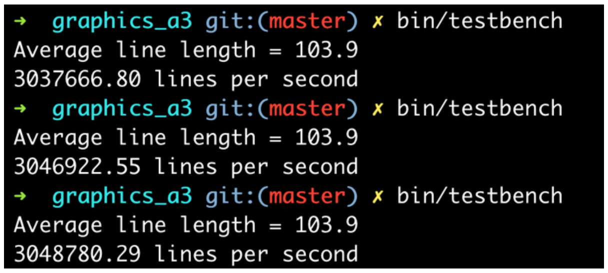


Figure 1lines.ppm



```
➜  graphics_a3 git:(master) ✗ bin/testbench
Average line length = 103.9
3037666.80 lines per second
➜  graphics_a3 git:(master) ✗ bin/testbench
Average line length = 103.9
3046922.55 lines per second
➜  graphics_a3 git:(master) ✗ bin/testbench
Average line length = 103.9
3048780.29 lines per second
```

Figure 2performance

The core of the task is the main drawing loop, where the lines are drawn onto the image. Using the previously calculated endpoints and colors, the program iterates through the lines, drawing each one onto the image using the line_draw function. This function employs Bresenham's line algorithm to efficiently render the lines pixel by pixel. The process continues for a fixed duration, ensuring that the image gradually fills with colorful lines.

After the drawing loop completes, the final image, now filled with 200 colorful lines, is written to a file named lines.ppm(Figure 1). The program calculates and outputs the average length of the lines and the drawing speed in lines per second(Figure 2) - Around 300 million lines/second.

Task 2: Lines in four quadrants (test3d)

The program generates an image file (test3d.ppm) by drawing various lines in different colors and orientations.

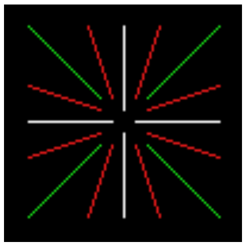


Figure 3 lines.ppm

Each line is defined by its start and end points, set using the line_set2D function, and colored using the color_set function with white, red, or green. The line_draw function employs Bresenham's line algorithm to render the lines on the image. The program draws horizontal and vertical lines to divide the image into four quadrants and various diagonal lines to cover all octants.

Task 3: Boxes, Circle and Lines (test3a)

The purpose of this project was to create a C program that demonstrates basic 2D graphics primitives by drawing various shapes such as boxes, circles, and lines. The program generates an image file (test3a.ppm) containing these shapes.

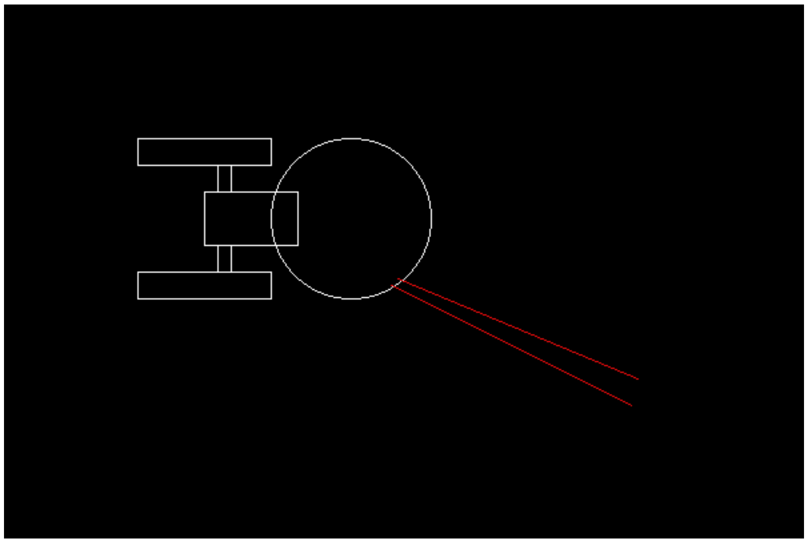


Figure 4 test3a.ppm

Boxes: The box function is defined to draw rectangles by setting the coordinates of the corners and drawing four lines to form the perimeter. Multiple boxes are drawn at different positions and dimensions using the box function and colored white.

Circle: A circle is created by setting its center point and radius using the circle_set function. The circle is then drawn in white using the circle_draw function.

Lines: Two red lines are drawn to intersect the circle and extend beyond the canvas. The lines are defined using the line_set2D function and rendered with the line_draw function.

Task 4: (test3b)

The graphics in test3b are polygons, starting from simple quadrilaterals and becoming more complex through the subdivision process.

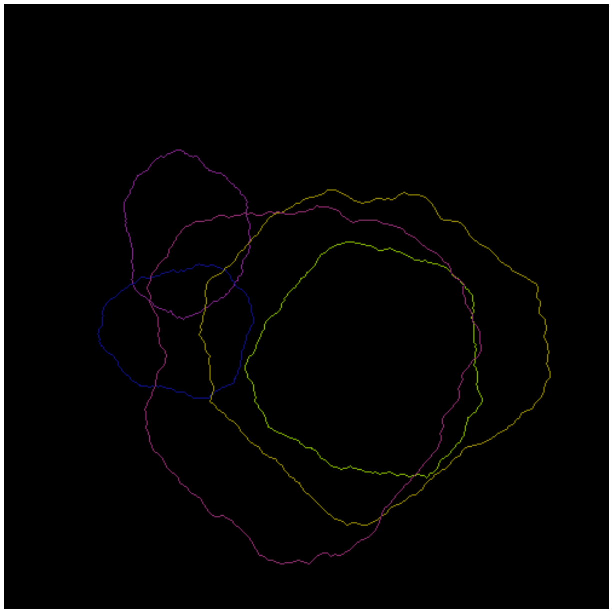


Figure 5 test3b.ppm

- Initial Polygons: The program begins by creating simple polygons, specifically quadrilaterals (boxes), using four lines to form the edges.
- Subdivision: These initial quadrilaterals are subdivided iteratively. Each edge of the quadrilateral is split into two new edges at a perturbed midpoint, effectively increasing the number of vertices and creating more complex polygonal shapes.
- Resulting Shapes: After multiple subdivisions, the initial simple polygons (quadrilaterals) are transformed into more detailed and complex polygons with many edges and vertices.

Task 5: (test3c)

In test3c, the goal was to test the polyline functions by creating and drawing various polylines, including random lines and boxes, onto an image.

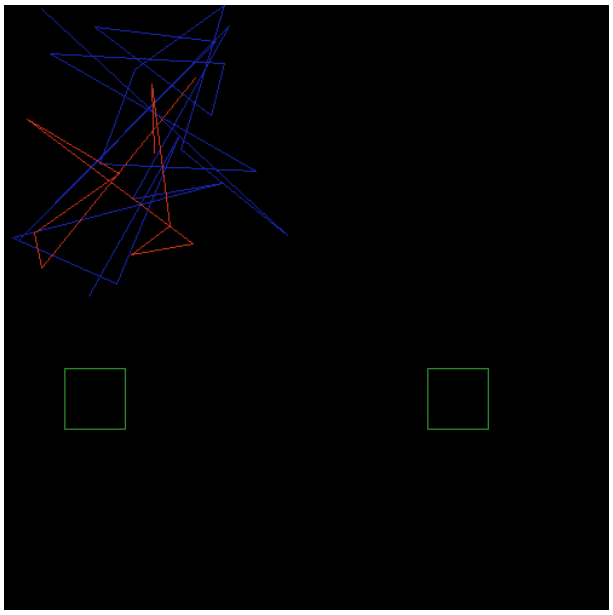


Figure 6 test3c.ppm

The program tests polyline functions by creating and drawing various polylines, including random lines and boxes. Random points are generated and used to initialize two polylines (thing1 and thing2). thing1 is created with 20 points, while thing2 is created with 10 points. The polylines are drawn in blue and red, respectively. The program then draws two green boxes: one counter-clockwise and one clockwise, demonstrating the effect of point order on rendering. The final image is saved as test3c.ppm, showcasing the different polylines and their colors.

Task 6: Create a 3D image

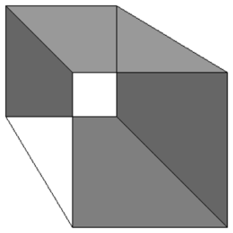


Figure 7 box.ppm

To draw this 3D box, I first define the eight corners of the box in 3D space.

Then we project these points to 2D. We use a perspective projection to convert the 3D coordinates of each point to 2D coordinates. This projection helps give the illusion of depth. The project_point_perspective function scales the x and y coordinates based on the z-coordinate and a distance parameter d.

Lastly, we draw and fill the faces. We define the six faces of the box using the projected 2D points. Each face is drawn and filled using the draw_filled_parallelogram function. This function draws the outline of the face using Line primitives and then fills the face using the flood_fill function in my extension task.

Extension Task1: Test Elipse (test 3e)

In this program, a red ellipse with white boundary and a while circle with red boundary were drew.

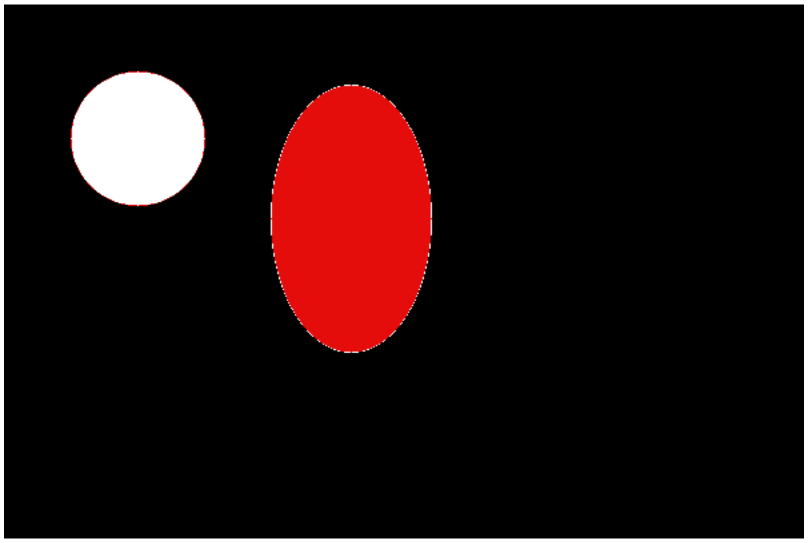


Figure 8 test3e.ppm

In this task, I demonstrate the ability to draw and fill ellipses and circles using 2D graphics operations, using ellipse_draw, ellipse_drawFill, circle_draw, circle_drawFill.

Extension Task2: Flood-fill algorithm

Create a flood-fill algorithm so that we can fill in polygons and circles.

The flood-fill algorithm I implemented uses Breadth-First Search (BFS).

Here we fill the circles and polygons:

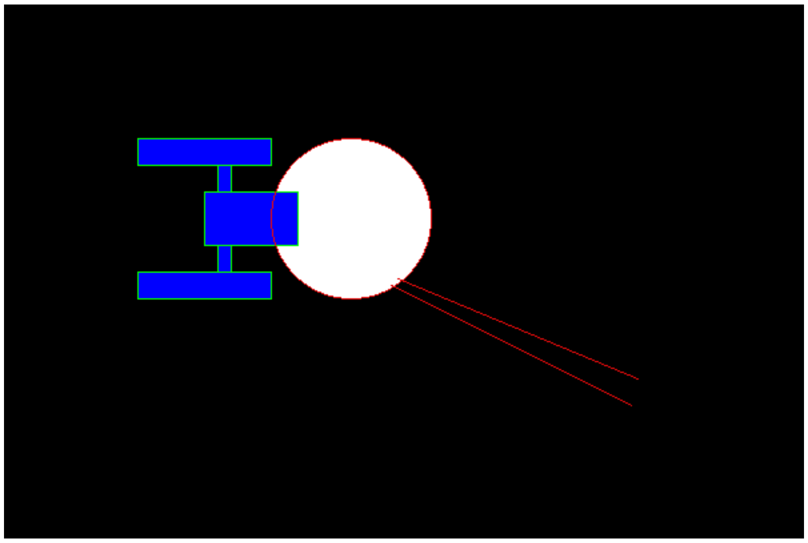


Figure 9 test3a_flood_fill.ppm

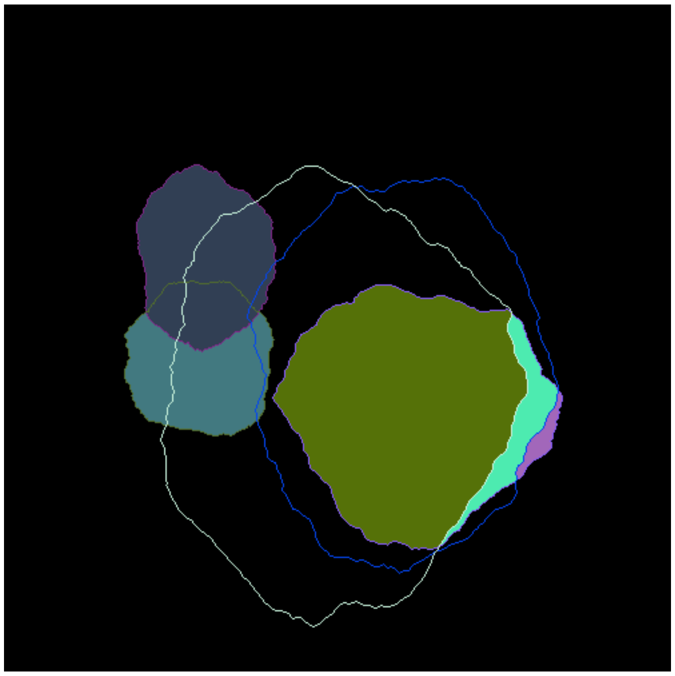


Figure 10 test3b_flood_fill.ppm

The flood_fill.c file implements a flood-fill algorithm to fill contiguous regions of an image with a specified color.

- Color Comparison: The is_valid_pixel function checks if a pixel matches the target color and is not already the fill color.

- Queue Initialization: A queue is used to manage the points to be filled. Functions are provided to create the queue, enqueue points, dequeue points, and check if the queue is empty.
- Flood-Fill Algorithm: The flood_fill function starts by getting the target color at the initial point (x, y). If the target color is the same as the fill color, it returns immediately.
- Queue Processing: The initial point is enqueued. The algorithm processes each point in the queue by checking its validity using is_valid_pixel. Valid points are colored with the fill color, and their neighboring points (up, down, left, right) are enqueued.
- Queue Depletion: The algorithm continues until the queue is empty, ensuring all connected pixels of the target color are filled with the new color.

Reflection

Implementing the flood-fill algorithm using BFS showed me how different search methods can change how tasks are done in image processing. I also learned that the order of points is very important when drawing polygons correctly. Working with linked lists and breaking down lines into smaller parts helped me understand how to use dynamic data structures and improve shapes step by step.

Acknowledgements

The Bresenham line drawing algorithm using symmetry and multiple: https://zingl.github.io/Bresenham.pdf

Examples of midpoint algorithm for drawing circles and ellipse provided in homework: circle_midpoint.c and ellipse_midpoint.c.

Flood_fill algorithm: https://en.wikipedia.org/wiki/Flood_fill