Graphics and Game Technology

Assignment 1

Line and Triangle Rasterization

1 Part 1 - drawing lines: The Midpoint Line Algorithm

Drawing a line can be considered one of the most elementary functions that a graphics library should be able to perform. Nowadays, a programmer almost never has to write a line function himself. Most computers have graphics hardware that draw lines through specialised hardware. In fact, most basic drawing operations are executed directly by the graphics hardware (drawing lines, triangles, rectangles, copying parts of the screen, etc.). However, to get some appreciation of the complexity of such a simple function, this first assignment introduces you to a commonly used algorithm for drawing lines. You will implement the classic Midpoint Line Algorithm (MLA) that was developed by Bresenham in the mid 1960s.

1.1 SDL

For this assignment you will use a graphical library that is called “Simple DirectMedia Layer” (SDL). This is a multimedia library designed to provide fast

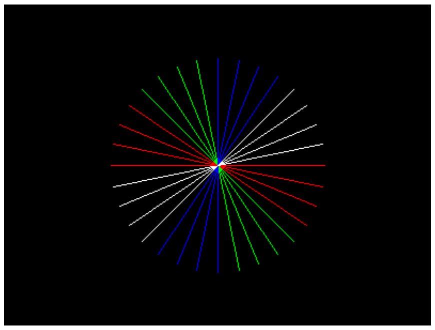

Figure 1: The final result.

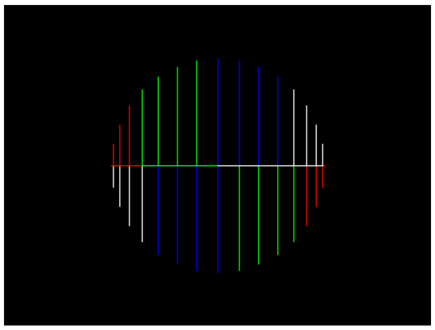

Figure 2: The output of the framework.

access to the graphics hardware (and also the audio device). SDL provides functions to open windows, draw pixels, lines and other objects, even play and record sounds and music. Furthermore there are extensions to read out joystick positions, etc. The only thing you will use is the ability to open graphical windows and change single pixels in them. Sounds archaic, right? True, but you will gain an appreciation of the complexity of modern graphics subsystems.

1.2 Getting started

Compile and run the framework in `basic_midpoint`. You should see the output as shown in Figure 2. If this works, open the file `mla.c` in your favorite editor. There, you will find a function called `mla()` that receives the coordinates of two points. The framework connects the two points by both a horizontal and a vertical line that have a right angle between them, as shown in Figure 2.

Although this is not what we want, this dummy function shows you how to draw pixels on the screen. After you have studied this, remove the body of the function and write your own, following the explanation of the mid-point algorithm in the lecture and in Shirley's book.

1.3 First step: the Midpoint Line Algorithm

To split up the problem, first only consider lines with a slope between 0 and 1 and whose starting point is on the left side. This means that only the lines in the first octant[1] will be drawn (the white lines in the upper right of Figure 1).

Modify `DrawFigure` (in `init.c`) so that only one line is drawn, for example by temporarily removing the for-loop and adding a single `mla()` function call that draws a line between (100,100) and (200,90).

Only if this works should you proceed with step 2.

[1]We will split up the coordinate system in 8 octants relative to the orgin. The first octant lies between 0 and 45 degrees (starting from a horizontal line to the right, the counter-clockwise to the left).

1.4 Second step: reflections

Next, you need to modify your algorithm to work for all octants. To begin with this extension, first determine in which octant the line will fall. This can be done by looking at the sign of $x_1 - x_0$ and $y_1 - y_0$ as well as the sign of $(x_1 - x_0) - (y_1 - y_0)$. Write a function that maps the resulting 8 cases to the corresponding octants (the usual ordering of octants is to begin with octant 1 from the previous step, and to continue counter-clockwise).

If you examine the properties of the second octant, you will notice that it is actually only mirrored on a line with slope 1. Therefore, you can copy your algorithm and carefully replace some of the y's by x's and vice versa, in such a way that we move each step to the north and north east, respectively instead of moving to the east and north east direction as we did for the first octant. Try this to see if it works.

Similar modifications apply for the other octants. If you want, you can work out the idea of reflections for the other six remaining octants.

1.5 Third step: refactoring your code

As you can probably imagine, this method blows up the size of your program considerably. Although it will probably work, it is not considered good software engineering practice to copy the same piece of code eight times with only slight modifications such as flipping x's and y's or changing the signs in some expressions. Look for a more general solution that is non-redundant, i.e. avoid code repetitions whenever possible. This can be done by refactoring your program by means of introducing new variables and functions.

1.6 Grading for Part 1

The grading of this assignment will be performed as follows:

- If your program compiles, runs without errors and then draws lines in one octant correctly (preferably the first octant) you will get 5.0 points.
- If your program paints the complete figure as shown in Figure 1 correctly (i.e. the line segments are at the right position and with the right color) and it is short and non-repetitive you will get 5.0 points. To be clear: if your program has 8 while-loops (this is the main loop of the original algorithm), you can be sure that it is *not* optimally refactored.
- Note that points may be deducted if your code is not well outlined or commented!

2 Part 2 - Triangle Rasterization

Modern graphics hardware is capable of drawing millions of triangles per second. The process in which this is done is called *rasterization* and the exact details of the algorithms used by companies like NVIDIA or ATI for rasterization and other 3D rendering operations are mostly kept secret, so as not to give the competition an advantage.

In this assignment you'll implement the triangle rasterization method as described in the book of Shirley *et al*, Chapter 3 and specifically Section 3.6 (this section comes with the assignment).

2.1 Framework

The framework in `tiangle_rasterization` will open a window that shows the output of the rasterization (or at least, after you have implemented it...).

It provides a method `PutPixel(x, y, r, g, b)` with which the color of a single pixel can be set. The `x` and `y` values are the (integer) coordinates of the pixel to be set, while the `r`, `g` and `b` values determine the pixel's color (in the range 0 up to 255).

Note that the origin – the pixel with coordinates (0,0) – is at the *lower-left* of the screen, the X axis points to the right and Y axis points up.

Also note that coordinates of triangle vertices are specified in floating-point format.

As a single pixel is usually pretty small on the screen the framework by default draws an enlarged version of the rasterized triangles, where each "triangle pixel" is drawn using a block of 7x7 screen pixels.

The framework provides two different scenes. The first (default) scene draws a number of triangles as specified in the file `triangles.h`. The second mode draws triangles with random vertex coordinates. This second mode is meant to test rasterization speed, which is used in the assigment. You can switch between the two scenes with the `'1'` and `'2'` keys.

Other keys available are:

- `q` – Exit
- `z` – Toggle zoom
- `o` – Switch between unoptimized and optimization rasterization (see 2.4)

2.2 Basic rasterization

Implement the basic triangle rasterization algorithm of page 64 by filling in the function `draw_triangle()` in `trirast.c`. Do not worry about pixels that happen to be exactly on triangle edges yet.

A useful trick when implementing your algorithm can be to initially ignore the color values passed to the function and instead color the pixels drawn using the barycentric coordinates for the pixel (suitably scaled to cover the range 0 to 255 per color channel).

2.3 Dealing with shared edges

When you're sure your implementation behaves as it should the next step would be to add the method described in section 3.6.1 for dealing with pixels exactly on triangle edges. As noted in the book, the off-screen point method should ensure that pixels on an edge shared by two triangles are drawn for just one of the triangles. But as the output image only shows the final pixel color we have no way of knowing if a pixel's color was actually set more than once.

We are going to "abuse" the frame buffer for this purpose. Instead of storing the new pixel color when `PutPixel()` is executed, we're going to store the number of times a given pixel was set using certain colors. This way we can literally see how many times a pixel was set and detect shared edges where pixels are not set exactly once.

The framework has a boolean flag `color_by_putpixel_count`, which you can toggle with the `'d'` key ('d' for double and/or debug). Alter the function `PutPixel()` so that when this flag is set the function uses the frame buffer to keep track of how many times a pixel's color was set. To keep things simple you only have to distinguish three different cases for a pixel:

- No `PutPixel()` operations yet; color: (0, 0, 0)
- 1 `PutPixel()`; color: (128, 0, 0)
- 2 or more `PutPixel()`'s: (255, 0, 0)

Note that the frame buffer's pixels are initially all black, i.e. (0,0,0).

When you test this "debug mode" you should see shared edges being drawn twice in your current rasterization implementation.

Add the method described in Section 3.6.1 to `draw_triangle()` and verify that the shared edges are now handled correctly.

2.4 Optimizations

As the book notes on pages 64 and 66 there's a lot of potential to optimize the algorithm. We can incrementally compute the values of α, β and γ, instead of fully computing them for each pixel. We can also skip the innermost loop early depending on the tests on the α, β and γ values.

Copy the triangle rasterization function you have made so far into the (empty) function `draw_triangle_optimized()`. Then alter this function to add the optimizations described in the previous paragraph and any others you can think of.

Check your optimized version against the original one, to make sure the output of the optimized version is not different from the unoptimized one. You can switch between triangle rasterization using the unoptimized and optimized versions using the `'o'` key.

3 Grading for Part 2

You can receive up to 4 points for a correct implementation of the basic rasterization algorithm.

Addition of the debug mode and correct handling of pixels on triangle edges can give you up to 3 points.

For a correctly working optimized version of the algorithm you can receive up to 3 points.

Again: points may be deducted if your code is not well outlined or commented!