Computer Graphics – Lab Assignment
Triangle Rasterization

1 Introduction

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.

2 Framework

The framework for this assignment 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 3.3)

3 Assignment

3.1 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).

3.2 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.

3.3 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.

3.4 Questions

Answer the following questions (put the answers in a text file that you submit with your code):

1. What factor(s) inherent in the rasterization algorithm used here influence the speed with which a triangle is drawn?

2. The book notes on page 65 at the bottom that "[...] the test is not perfect because the line through the edge may also go through the offscreen point [...]". How would you handle a situation in which a pixel is exactly on an edge and the edge runs exactly through the offscreen point?

4 Grading

You can receive up to 3 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 2 points.

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

Correct answers to the questions are worth 0.5 point each.

You can get up to one additional point for writing clear and well-commented code.