Graphics and Game Technology

Assignment 1

Raster graphics: Conway's Game of Life, lines and triangles

1 Conway's Game of Life

In 1970, the British mathematician James Conway invented a game that we now call "Conway's Game of Life". It is a peculiar game in that it has no players: it is completely played by a computer from an initial playing position and using a simple set of rules. The game takes place on a two-dimensional orthogonal grid of cells, the state of which can either be "alive" or "dead". Initially, the grid is "seeded" with a random spray of dead and alive cells. After that, the game repeatedly traverses the grid and for each cell applies the following rules:

1. If the cell is alive and has fewer than two live neighbours, it dies, as if by underpopulation.
2. If the cell is alive and has two or three live neighbours, it lives on to the next generation.
3. If the cell is alive and has more than three live neighbours, it dies, as if by overpopulation.
4. If the cell is dead and has exactly three live neighbours, it becomes a live cell, as if by reproduction.

These simple rules generate patterns on the grid that are fascinating to watch: clusters of cells appear to grow and shrink in-and-out of existence, static patterns remain unchanged until they are invaded by neighbouring cells, oscillating patterns that "pulsate" and even "spaceships" that travel over the grid. Conway's Game of Life falls in the family of systems that are now commonly called "cellular automata". These systems exhibit properties that make them very useful for use in simulations and computations. In fact, Conway's Game of Life is Turing complete which means that it is capable of universal computing!

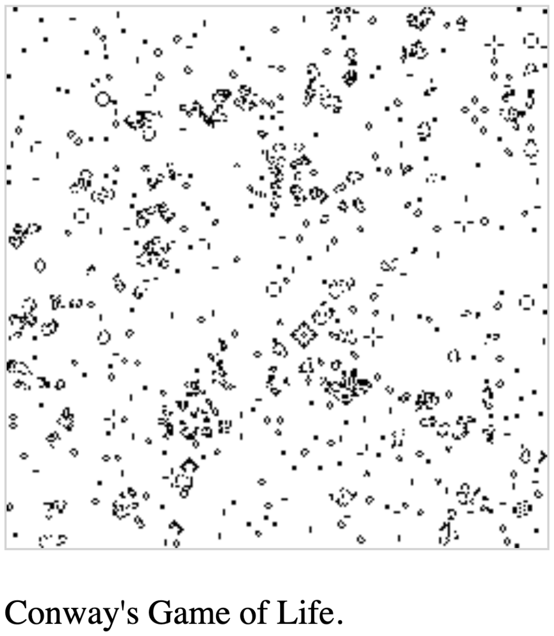

Figure 1: An example of a pattern your Conway's Game of Life could produce.

1.1 Assignment

You will implement Conway's Game of Life as a web application, using javascript. When you load `conway.html` into a web browser, the framework does little more than create a HTML canvas element of 250 by 250 pixels and sets up an interval timer to repeatedly call a function `updateGameArea()`. Your job is to fill in the blanks (`/* ... */`):

1. Begin by implementing `getPixel(x,y)` and `putPixel(x, y, val)`. Both these functions must adress the cell at position (x, y) in the grid. However, in the framework the grid is stored in a linear array. Implement the code that addresses the correct element in this array. Two things you should note:

   (a) Each pixel is represented by *four* elements in the array that represent its colour: the first for red ('r'), the second for green ('g'), the third for blue ('b') and the fourth for transparency ('a' or α). Together they are often represented as "(r,g,b,a)" while images that are stored this way are referred to as "RGBA". Note that a HTML canvas is initialized to transparent black pixels: (0,0,0,0). An opaque white pixel is represented as (255,255,255,255). In our game we only need to store two states: "alive" or "dead". Think about how you want to reflect these in the pixels.

   (b) The framework exposes *two* arrays: `data` which is the array that is visible on screen and that we suggest you use in `setPixel(x, y, val)` and `copy` which is a copy we suggest you use in `getPixel(x,y)`. Think about this for a second: why do we need two arrays?

2. Implement `countNeighbours(x, y)` which returns the number of alive neighbours of the cell at (x, y). Neighbours are cells that are horizontally, vertically or diagonally adjacent. Be careful not to address cells that are out of bounds.

3. Implement `updateGameArea()` which iterates over all cells in the grid and applies the above four rules to each cell.

We probably don't need to tell you this, but just in case: Any time you make a change to the source code, make sure you reload the web page to see the effect. Also; it helps to open a console view to see any warnings or errors (F12 in most browsers). If you program works correctly, you should end up with an animated view, a still of which is shown in Figure 1.

1.2 Grading

The grading for this part of the assignment is as follows:

- **6 points** if your program runs without errors and draws a correctly evolving Game of Life.
- **4 points** if you correctly answer the question we asked above: why do we need two arrays?
- Note that points may be deducted if your code is not clearly formatted or commented!

2 Drawing lines with the Midpoint Line Algorithm

Drawing a line can be considered one of the most elementary functions that a graphics library must perform. Nowadays, a programmer almost never has to write such a 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.

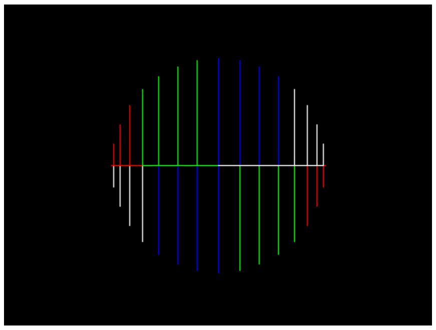

Figure 2: The initial output of the framework.

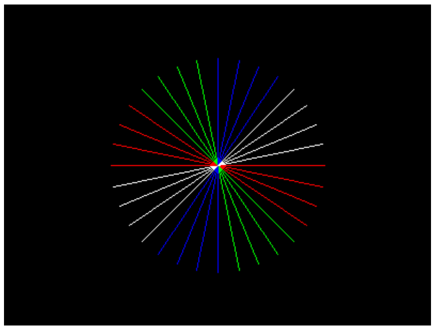

Figure 3: What we expect your program to do.

2.1 The SDL library

For this assignment you will use a graphical library that is called "Simple DirectMedia Layer" (SDL)[1]. This is a multimedia library designed to provide fast access to the graphics hardware (amongst others). 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? True, but you will gain an appreciation of the complexity of modern graphics subsystems.

2.2 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 demonstrates 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 the "Fundamentals of Computer Graphics" (FCG) text book.

2.3 First step: the Midpoint Line Algorithm

To solve our problem in small, easier steps, we break up the coordinate system in 8 octants relative to the orgin. The first octant lies between 0 and 45 degrees: the section between a horizontal line to the right and the line rotated around

[1]Make sure you install *development* version 2.0.x. We assume you know how to do this.

the origin 45 degrees counter-clockwise (the white lines in the upper right of Figure 3).

First 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 will be drawn.

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.

2.4 Second step: reflections

Next, you need to modify your algorithm so that it works 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: work out the idea of reflections for the other six remaining octants.

2.5 Third step: refactoring your code

As you should have noticed, this method blows up the size of your program considerably. Although it will 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.

2.6 Grading

The grading for this part of the assignment is as follows:

- **2 points** if your program compiles, runs without errors and then draws lines in one octant correctly (e.g. the first octant).
- **4 points** if your program paints the complete figure as shown in Figure 3 correctly (i.e. the line segments are at the right position and with the right colour).

- **4 points** if you refactored your program so that it is short and non-repetitive. To be clear: if your program has 8 while-loops (this is the main loop of the original algorithm) it is *not* optimally refactored.
- Note that points may be deducted if your code is not clearly formatted or commented!

3 Triangle Rasterization

Modern graphics hardware is capable of drawing millions of triangles per second. The process by which this is done is called *rasterization* and the exact details of the algorithms are mostly kept secret, so as not to give the competition an advantage.

In this assignment you will implement the triangle rasterization method as described in the FCG book (see the PDF file that comes with this assignment).

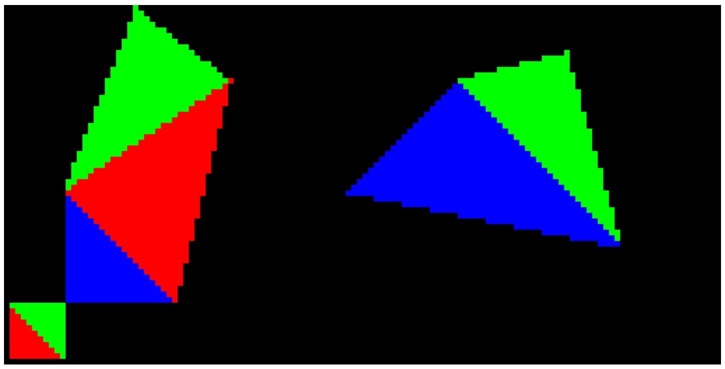

Figure 4: This is what your triangle rasterization should look like.

3.1 Framework

The framework in `tiangle_rasterization` opens a window that will show the result of the rasterization once you have implemented it. It provides a method `PutPixel(x, y, r, g, b)` that you can use to set the colour of a single pixel. 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 colour (in the range 0 up to 255 per colour).

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, 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 the performance of the rasterization, which is used in the last part of the assignment. 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 section 3.4)

3.2 Basic rasterization

Implement the basic triangle rasterization algorithm described on page 165 by filling in the function `draw_triangle()` in `trirast.c`. At this point, do not worry about pixels that happen to be exactly on triangle edges.

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

3.3 Dealing with shared edges

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

We are going to "abuse" the frame buffer for this purpose. Instead of storing the new pixel colour when `PutPixel()` is executed, we're going to store the number of times a given pixel was set using certain colours. 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 colour was set. To keep things simple you only have to distinguish three different cases for a pixel:

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

Note that initially all pixels in the frame buffer are 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 the FCG book to `draw_triangle()` and verify that the shared edges are now handled correctly.

3.4 Optimizations

As the FCG book notes; there is 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 terminate 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.

3.5 Grading

The grading for this part of the assignment is as follows:

- **4 points** for a correct implementation of the basic rasterization algorithm.
- **3 points** for the addition of the debug mode and correct handling of pixels on triangle edges.
- **3 points** for a correctly working optimized version of the algorithm.
- Again: points may be deducted if your code is not clearly formatted or commented!