Computation Structures
Project 2: Simple Fractal in β-assembly

November 20, 2024

General information

- **Hard Deadline: 14th December 2024** (**0** for late submission).
- Project must be done **individually**.
- Questions will no longer be answered 24 hours before the deadline.
- Contact: `bknott@uliege.be`, Office 1.9 (B37).

1 Introduction

The goal of this project is to make you more familiar with the β-assembly language by getting your hands dirty and writing code. In this project, you will implement algorithms to draw a simple fractal[1] using circles and squares.

The β-simulator (βSim), available on eCampus, provides a graphical interface which will be useful to visualize the fractal you will draw.

2 Memory view of the β-simulator

The β-simulator (βsim) presented during the tutorials features a memory view window allowing you to visualize the 1024 first words[2] of the memory. This portion of the memory can be reffered to as the video memory (see Figure 1). In this view, each bit corresponds to 1 pixel: a bit set to 1 will result in a black pixel while a bit set to 0 will result in a white pixel. Each line of pixels corresponds to 8 consecutive words of the memory. This window is where you will draw the fractal.

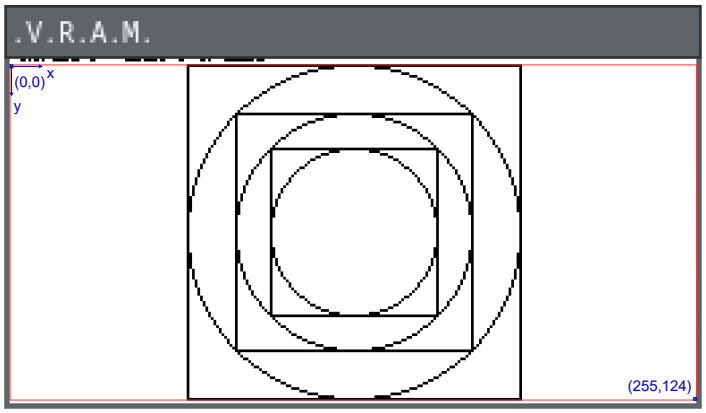

Figure 1: The memory view displaying the video memory (=first 1024 words of memory). The fractal you will draw is shown here with a recursion depth of 3. Red outlines indicate the canvas boundaries. The canvas's last pixel is located at $x = 255$ (8 words of 32 bits) and $y = 124$ (128 lines minus 1 line for bootstrap code[3]and 2 empty lines for visual clarity).

[1] `https://en.wikipedia.org/wiki/Fractal`

[2] Note that in this statement, words of 32 bits are considered, just like in the theorethical course. This is not the same meaning as in `beta.uasm`, where a `WORD` is 16 bits and a `LONG` is 32 bits.

[3] See section 4.1, in `main.asm`.

You will not have to deal with the memory view in details in this project as a canvas to draw your fractal has been defined for you and the function `canvas_set_to_1` is provided to set a bit to 1 in the canvas.

This function can be found in the `util.asm` file provided with the project.

Tip: the graphical view of the memory can sometimes be buggy. If you encounter issues, simply clicking on the memory view window should fix it.

3 Simple fractal

This section explains the algorithm you will implement to draw the fractal. Even though understanding the algorithm is not necessary to implement it (you simply need to translate the C code given with the project), it is advised to at least get a basic understanding of it.

The challenge when implementing a fractal using β-assembly is that the language does not provide easy support for floating-point arithmetic. This is why you are going to implement a simple fractal that only uses integer arithmetic.

3.1 Note on the axes

There are two sets of axes used in the project:

- The **canvas axes** serve as the reference axes for the entire canvas. The top left corner of the canvas is $(0, 0)$, the `x` axis goes from left to right and the `y` axis goes from top to bottom. These are the axes used for most of the project.
- The **circle axes** are the axes used to draw a circle. The center of the circle is $(0, 0)$, the `x` axis goes from left to right and the `y` axis goes from bottom to top. These axes are only used in the function to draw the circle and coordinates are always named `circleX` and `circleY`.

3.2 Drawing a square

The square will be defined by the coordinates of its top-left corner (x, y) and its side length. To draw it, iterate over all the pixels within the square's boundaries, setting the value of only the pixels along the outer edges (i.e., the outline) to 1.

3.3 Drawing a circle

Drawing a circle is a bit more complex. The algorithm used, illustrated in Figure 2, in is the Bresenham's circle algorithm.

The circle is divided into 8 octants, and you only need to compute the pixels in one of them, the other pixels being symmetrical. The overall logic is as follows:

1. The pixel at coordinates $(0, r)$ is chosen as the starting point [4]. The initial values are thus `circleX = 0`, `circleY = r`. A decision variable `decisionVar` is also initialized.
2. At each step:
   - a pixel is drawn at `(circleX, circleY)` and at the symmetrical points in the other octants.
   - the focus is moved one pixel to the right (`circleX++`);

[4] **Be careful:** $(0, 0)$ now represents the center of the circle

- the value of the decision variable `decisionVar` is checked to decide if the focus should be moved one pixel down (`circleY--`) or not;
- the decision variable `decisionVar` is updated.

3. Once `circleY` is less than `circleX`, it means the end of the octant has been reached and you can stop.

	circleX	circleY	decisionVar
iteration 0	0	8	-13
iteration 1	1	8	-7
iteration 2	2	8	3
iteration 3	3	7	-11
iteration 4	...	...	...

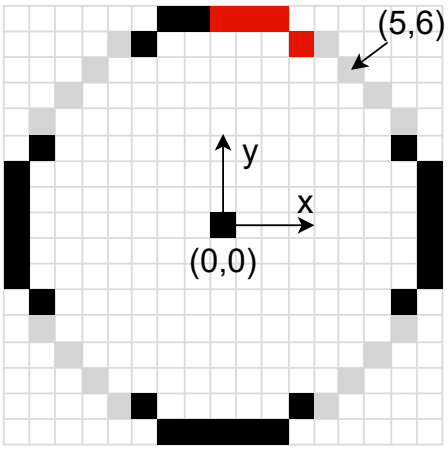

Figure 2: The Bresenham's circle algorithm on a circle with a radius of 8 pixels[5]. In red, the pixels computed until iteration 3 (included). In black, the symmetrical pixels (in the other octants). In grey, the pixels that will be computed in the next iterations. When the value of `decisionVar` is positive, the value of `circleY` is decremented for the next iteration.

The value of the decision variable `decisionVar` is inferred from the equation of the circle, you do not have to understand its meaning.

3.4 Drawing the fractal

The fractal to draw is shown in Figure 1. It is composed of squares and circles interleaved.

At each iteration, a square and a circle are drawn. A depth of 3 would thus draw 3 squares and 3 circles. They are organized in a way that they always fit perfectly in the previously drawn shape.

To fit a circle inside a square is not very difficult: from the top left corner and the side length of the square, you can easily deduce the radius and the center of the circle.

Fitting a square inside a circle is a bit more complex as it would typically require trigonometric functions. However, an easier method is to leverage the Bresenham's circle algorithm, since the last pixel drawn is exactly where one of the corners of the square should be (on Figure 2, the last pixel would be $(5, 6)$). The Bresenham's function will thus be implemented in a way that it returns the coordinates of the last pixel drawn.

[5]Technically, the radius is 8.5 pixels, but we will consider it is 8.

4 Project & implementation

4.1 Given files

You are provided with `simple_fractal.c`, a full *C* implementation of the algorithm (with comments). You will have to implement the `drawSquare`, `drawCircleBres` and `drawFractal` functions in β-assembly. The remaining functions in `simple_fractal.c` are either provided or are not required for the β-assembly implementation.

You are also provided with a file `util.asm` containing the definition of `canvas_set_to_1` (see Section 2), a file `circle_bresenham.asm` that already contains the implementation of the `placeCirclePixels` function from `simple_fractal.c`, and a file `main.asm` that you can use to test your implementation.

The file `main.asm` contains the bootstrap code at the start of the memory. This is why, on Figure 1, the first line is not blank.

Just afterward, the canvas to draw the fractal on is initialized to 0. Note that "`canvas`" is a **global** label that contains the address of the first word of the canvas. You will not have to use it directly in your implementation as the `canvas_set_to_1` function directly uses the address of the canvas.

The `maxDepth` variable is also defined in this file, you can change its value to test your implementation. You can assume that the value of `maxDepth` is always greater than 0.

Some debug code is also provided (commented) to help you test your square and circle drawing functions.

Finally, the `drawFractal` function is called with the appropriate arguments.

4.2 Your task

First, you have to implement the `drawSquare` function in a file called `square.asm`. This function takes 3 arguments: the x and y coordinates of the top left corner of the square, and the side length of the square. The function must draw the square in the memory. You will have to use the function `canvas_set_to_1` to set the pixels to 1.

Then, you have to implement the `drawCircleBres` function in the file `circle_bresenham.asm`. This function takes 3 arguments: the x and y coordinates of the center of the circle, and the radius of the circle. The function must draw the circle in the memory.

Finally, you have to implement the `drawFractal` function in a file called `fractal.asm`. This function takes 4 arguments: the x and y coordinates of the top left corner of the first square, its side length, and the depth of the fractal.

> **Important note:** you are required to implement the *drawFractal* function in a **recursive** way, as in the *C* implementation.

4.3 Additional guidelines

Regarding procedure calls, you **must** use the "*last-argument-pushed-first*" (LAPF) convention throughout the whole project.

Since you will only submit the three files containing your implementation, any additional macro or procedure needed for the implementation of the fractal must be implemented in these files (and **not** in `util.asm` or `main.asm`).

You are advised to comment extensively your assembly code. It is good practice to document what your code is doing on a per-block basis (usually, a line of C translates into several instructions) and to state which registers are assigned to which variables.

Your primary task is to translate the C code into β-assembly, so you won't be penalized for omitting aspects, such as argument verification or optimizations unrelated to β-assembly, if they are not present in the original C code.

For this project, using registers instead of the stack for local variables is preferred. You should also try to minimize the number of registers used by reusing them where practical.

5 Practical information

In order to learn β-assembly effectively, **this assignment will be done individually**. Plagiarism is of course not allowed and will be severely punished. Asking questions on the Forum (on eCampus), and helping your fellow students on the Forum is allowed (to a reasonable extent).

You will include your completed `fractal.asm`, `square.asm` and `circle_bresenham.asm` files in a ZIP named {`student_id`}`.zip` (for example `s123456.zip`). You can also include a `feedback.txt` file in the ZIP if you want to provide feedback on the project.

Naming your files differently or submitting other files **will result in a penalty**.

Submit your archive to the **Montefiore Submission Platform**[6] (course INFO0012-2). If you encounter any problem with the platform, contact the assistant (bknott@uliege.be) as soon as possible.

Good luck and have fun!

[6] `http://submit.montefiore.ulg.ac.be/`