TDT4258 Low-Level Programming
Autumn 2024

Lab assignment 2

From Assembly to C: A useful abstraction

Handout: Thursday, 12th September 2024, 12:00
Deadline: Friday, 11th October 2024, 17:00

Teaching Assistants: Mikkel Svartveit and Magnus Øvre Sygard
Assignment Coordinator: Roman K. Brunner
Professor: Lasse Natvig

Pre-amble

The labs are here for you to deepen your understanding of concepts taught in the lecture. The goal is that you not only develop a theoretical understanding of the matter, but also develop the technical skills to apply it in practice.

Each lab has a **main project**, but we also provide optional exercises for those who want to go beyond the mandatory exercise. To collect points for a lab, you only need to **hand in the solution to the main project** (in addition to filling in an AI-"statement" as described at blackboard).

The optional exercises are for possible extended learning and maybe even your entertainment.

The lab exercises are compulsory activities in the course. **You need to collect 27 points in total to be allowed to take the exam**. We assess the lab assignments such that a fully approved solution will get 10 points, and we hope that will be the normal case. Submissions with significant defects might get a reduced number of points. Since there will be at least 4 labs it will be possible to reach the exam threshold without a full score on three first labs. This is a new approach compared to earlier editions of the course — and has been developed to give students more flexibility.

As the assignments are part of the evaluation, they are subject to NTNU's plagiarism rules [1]. We have tools at our disposal and will run all submissions through plagiarism checkers. Copying code from current or past students is considered plagiarism. Hence, we advise you to not share code to prevent situations where we have to find out who copied from whom.

While copying each other is disallowed, we still encourage student discussions about your solutions. This will allow you to explore alternative approaches and solutions and learn about the advantages and challenges of particular implementations. **As a rule of thumb: Sharing and discussing ideas is fine, sharing code is not.**

1 Description

In this lab we are going to raise the bar of abstraction slightly: We will not continue programming purely in Assembly, but will turn to C. While the C language still operates pretty closely to the hardware, it allows us to use many features of a high-level programming language. However, we are using C bare bone, which comes with some limitations. In a bare bone environment (such as the one simulated by CPUlator), you don't have access

[1] https://i.ntnu.no/wiki/-/wiki/English/Cheating+on+exams

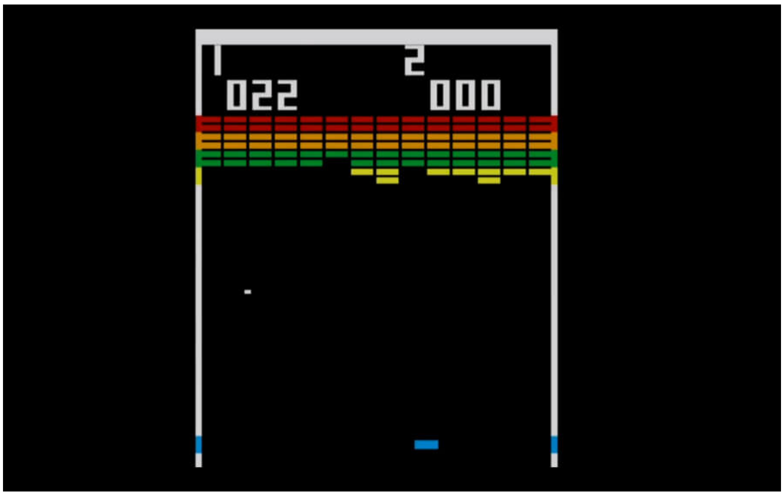

Figure 1: The original Atari Breakout game.

to OS functionality and standard library functions. This means that you can only rely on language features, anything else you need to implement yourself.

In this lab, we are going to continue where the optional tasks of lab 1 left off: we are going to use the assembly functions that write to VGA to provide graphical output for a small game that you are going to develop. The game resembles Atari's 1976 Breakout game[2]. The target of the game is to steer a bar up and down such that a ball bounces back to the blocks (instead of playing from bottom to top we are going to play from left to right, see Figure 2). Whenever the ball hits a block, it will bounce back from the block in a predefined angle, and the block that was hit will be destroyed. The game is won if the ball can escape to the right — if the player managed to create a tunnel through which the ball can reach the right end of the screen. The game is lost if the ball escapes the screen to the left. This happens when the user fails to position the bar so that it is hit by the ball.

2 Main Task: Breakout Game

2.1 Game specification

The game has to follow very specific requirements:

1. The ball is represented by a black, 7×7 pixels square.

[2]The Breakout video game was released in May 1976, and was designed by Steve Wozniak. In the same year, Steve Wozniak co-founded Apple Computer together with Steve Jobs.

2. The blocks are of size 15×15 pixels. The blocks can be of any colour, but they must remain the same throughout the game. Choose the colour so adjacent blocks aren't the same colour (or add a border to the blocks as in Figure 2). If you opt to add a border, make sure that the size of your blocks including the border are not bigger than 15×15 pixels.

3. User input is read from UART, where the letter w moves the playing bar up by 15 pixels and the letter s moves the playing bar down by 15 pixels. Any other letters should be consumed from the buffer, but be ignored. The first letter (w or s) will start the game if the game is not yet running (e.g. after loading, winning or losing a game). Pressing enter should terminate the game, which in C means that the main function should return.

4. For the direction of the ball we define a vertical direction upwards as 0 degrees, going horizontally to the right as 90 degrees and vertically down as 180 degrees. The bar is of size 7×45 pixels, where as for the central 15 pixels (in y direction), the ball will be reflected in a $90°$ angle to the bar. In contrast, if it hits the left (upper) or right (lower) 15 pixels, it will be reflected in a $45°$ angle to the left, respectively in $135°$ angle to the right.

5. In the template code you find the variable `n_cols` that specifies how many columns of blocks the game should have. Your game should support anywhere between 1 and 18 columns. If the number lies outside that range, your program should not crash, but inform the user that this is not a playable configuration. The blocks should always fill the full height of the screen with blocks. The blocks should be positioned to the very right of the playing field, which means that the right-most column's x coordinates start from 306 and end at 320.

6. After the game has finished (winning: the ball escapes the playing field out to the right / loosing: the ball escapes the playing field out to the left), print a corresponding message to the UART.

In Figure 2 you see an example of a breakout implementation as specified above. Some details to notice: The ball cannot leave the playing field on the upper and lower edge, but it will be reflected according to the rules of physics. This means that hitting the upper edge at angle 45 degrees will bounce back at angle 135 degrees, and similarly an angle 135 degrees to the bottom edge will bounce back with direction 45 degrees. The same rules

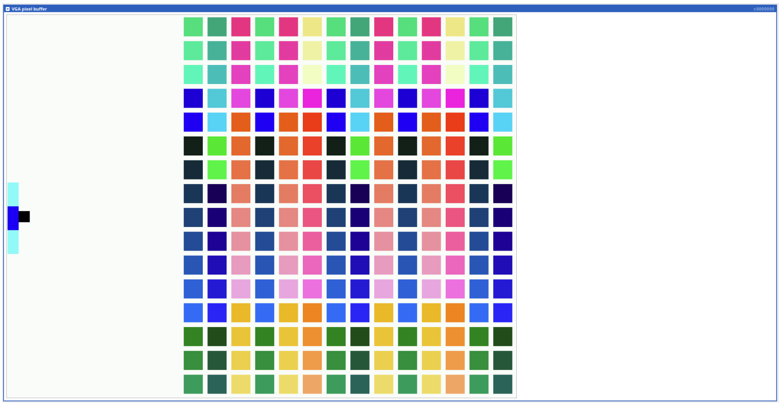

Figure 2: A sample of the breakout game with the above mentioned parameters.

apply for the ball hitting the blocks. Again, it is only the center pixel of the ball that defines what is hit.

2.2 Hints

1. Use the given code in `breakout.c` as a starting point for your coding. It contains an assembler routine for `SetPixel()` and shows how to use it from C to program the CPUlator VGA.

2. The handout code contains the main game loop. Note that quite often when you program graphics you might need to slow down the program to be able to observe the changes on the VGA screen, at least during the debugging phase. The need depends on the efficiency of your code. This can be done by a breakpoint during debugging, or a delay inserted at an appropriate place in the code.

3 TODO: Optional Tasks

- **A [EASY]:** Implement the `BigPixel()` function described in optional task D for Lab 1. You can use `VGAmini.s` from that Lab, learning material from lectures 1 and 2 and the code handed out in this Lab to help you.

- **B [MEDIUM]:** Write a C function `RasterChar(char* font, unsigned int x, unsigned int y, unsigned int color)` that draws a picture of 8 x 8 "BigPixels" (see optional task A) on the CPUlator VGA display where the upper left corner is placed at position (x, y). The parameter `font` is a pointer to exact 8 bytes in memory, and the values of the bits in each byte from Least Significant Bit (LSB) to Meast Significant Bit (MSB) defines the pixels from left to right in one row of pixels. The first byte represents the upper row. You find an open-source font[3] in the handout code.

- **C [MEDIUM]:** (generative AI)
  Try to use generative AI to develop a C function that reads an 8x8 character matrix where each position is either blank (space) or one of the eight uppercase letters R, G, Y, B, P, C, O and W representing the following color-values on CPUlator: 0xF800 = Red, 0x07E0 = Green, 0xFFE0 = Yellow, 0x001E = Blue, 0xF81F = Pink, 0x07FF = Cyan, 0xFD20 = Orange, 0xFFFF = White. The function should read the matrix and display it as 8 x 8 "BigPixels" (see optional task A) on the CPUlator VGA display. Write a loop that displays a few such colored 8 x 8 images on the VGA-display. Remember hint no. 2 about delay mentioned above.

- **D [DIFFICULT]:** (generative AI)
  Try to use generative AI to develop a stand alone C program that is a simple text-editor specialized for editing an 8 x 8 character "color-matrix" as described in the task above. You are only allowed to write one of the 8 characters or blank (space) in each of the 64 positions. Solving this should make it easy to extend the 8 x 8 bit font with the fantastic Norwegian letters Æ, Ø and Å.

- **E [DIFFICULT]:** Extend the game to a two-player game, where one player is an "AI-player" that you can play against. The player that breaks out in shortest time wins. You are probably capable of calculating the trajectory of the ball, making the "AI-player" too good for you. Then you can adjust its "intelligence" down by increasing the probability of imprecise moves of the bar. In other words — reduce its "intelligence" by a parameter so that you are able to beat it.

4 Submission and assessment

Submit your **commented C code file** `breakout.c` before the deadline on Blackboard. Aside from comments, make sure your code is well readable, variables are sensibly named and that your code is well structured.

[3]Source: Basic font taken from `https://github.com/dhepper/font8x8`

We expect all submissions to meet the following requirements:

1. The submission is on time. We provide enough time to solve the labs, but we expect you to start early.
2. You submit *a single C file* ***`breakout.c`***. The file should contain the two assembler functions `drawBlock()` and `drawBar()` as inline assembly. To be able to get a full score (10 points) these two functions must follow the ARM Register usage conventions as taught in Lecture 2.
3. The submitted file compiles and does not crash when running on the specified system on CPUlator[4].
4. The program follows the outlined requirements.
5. You have submitted an *"AI-statement"* for this lab exercise. It will be anonymized and will not influence on the assessment — but will help us all to gather experience with the use of generative AI in the course. This is further explained in lecture 0 and on blackboard.

Failing to meet any of these requirements will result in the number of points collected by the submission to be reduced, depending on the seriousness of the defect(s). No submission gives zero points.

Commenting on your code and keeping it tidy is very important. Helping us understand what you did, supports us in assessing your work – we can only give points for what we understand.

5 Similarity Checking and Plagiarism

You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates or any other sources. We check all submitted code for similarities to other submissions using automated tools.

6 Questions

If you have any questions about this assignment, we encourage you to post it on the discussion forum provided by the course. By that, you also help other students who have the same questions in the future.

[4]https://cpulator.01xz.net/?sys=arm-de1soc

Figure 3: Source: https://xkcd.com/505/