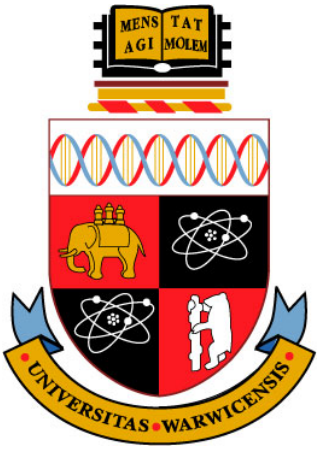

Joint Report for Coursework 2
3D Snake on LED Cube

CS132 Computer Organisation and Architecture

Wenzhou Mei and Tatsunori Ono

Contents

1 Aims and Design **1**
- 1.1 Aims of this project . . . 1
- 1.2 Overview of the 3D snake game design . . . 1
- 1.3 What the player sees . . . 1
- 1.4 How does the joystick control the snake? . . . 1
  - 1.4.1 When is input taken from the joystick . . . 1
  - 1.4.2 How are joystick signals processed . . . 2

2 Implementation **3**
- 2.1 Hardware . . . 3
  - 2.1.1 Specification of hardware . . . 3
  - 2.1.2 Range of joystick . . . 3
- 2.2 Coding style . . . 3
  - 2.2.1 Prefixes . . . 3
  - 2.2.2 Commenting . . . 3
- 2.3 Logic of code . . . 4
  - 2.3.1 Overarching logic . . . 4
  - 2.3.2 Interaction with hardware . . . 4
  - 2.3.3 Data structure of the snake . . . 4
  - 2.3.4 Data structure of the cube map . . . 5
  - 2.3.5 Mapping joystick channel inputs to useful "signals" . . . 5
  - 2.3.6 Setting direction of the snake . . . 5
  - 2.3.7 Start of program . . . 6
- 2.4 Improvements we could make . . . 6
  - 2.4.1 Apple generation . . . 6
  - 2.4.2 Checking if set bit is a snake segment or the apple . . . 6
- 2.5 How we would have implemented the extensions we didn't do given more time . . . 6

3 Results and Reflection **7**
- 3.1 Results . . . 7
- 3.2 Limited testing time . . . 7
- 3.3 Overall reflection . . . 7
- 3.4 Evaluation of the design . . . 7
  - 3.4.1 Handling of the snake death . . . 7
  - 3.4.2 Control of snake . . . 7
- 3.5 Evaluation of the implementation . . . 8
  - 3.5.1 Positives on our implementation . . . 8
  - 3.5.2 How we could further improve modularity and organisation 8
  - 3.5.3 Comments about other improvements we could've made . 8

1 Aims and Design

1.1 Aims of this project

Our primary objective was to build a functional 3D snake game using the hardware given according to the specification [2]. Any remaining time was dedicated to completing the extensions also detailed in the specification.

1.2 Overview of the 3D snake game design

The goal of the game is to, as a snake, eat as many apples as possible to reach a length of 100. The game ends if the player crashes the snake into another segment of itself, a wall (outside of the LED cube), or it reaches the score of 100 (in which the player wins). Starting at length 2, each time the snake eats an apple, the snake grows by length 1 (making it harder to avoid dying), and a new apple is generated on a random empty cell of the cube. To eat an apple, the player must control the snake using the mechanics described in section 1.4, such that the head of the snake moves to the cell where the apple is.

1.3 What the player sees

The game is rendered on the $8 \times 8 \times 8$ LED cube provided, where each LED corresponds to a cell on the map. Cells with an apple or is part of the snake are ON, while empty cells are OFF.

1.4 How does the joystick control the snake?

1.4.1 When is input taken from the joystick

Every pass (which happens every second), the program maps the current joystick input onto a signal (the joystick is tilted right: RIGHT, the joystick is tilted left: LEFT, the joystick is tilted up: UP, the joystick is tilted down: DOWN, the joystick isn't tilted in any direction: CENTRE). Note that the naming of these "signals" are from the tilt of the joystick, not the movement of the snake.

1.4.2 How are joystick signals processed

Snake is moving / Signal received:	In plane facing user	Perpendicular to plane facing user
LEFT	Relative left	Absolute left
RIGHT	Relative right	Absolute right
UP	Absolute inwards	No direction change
DOWN	Absolute outwards	No direction change
CENTRE	No direction change	No direction change

Figure 1: Table showing the direction the snake would travel depending on its current direction and the joystick signal received

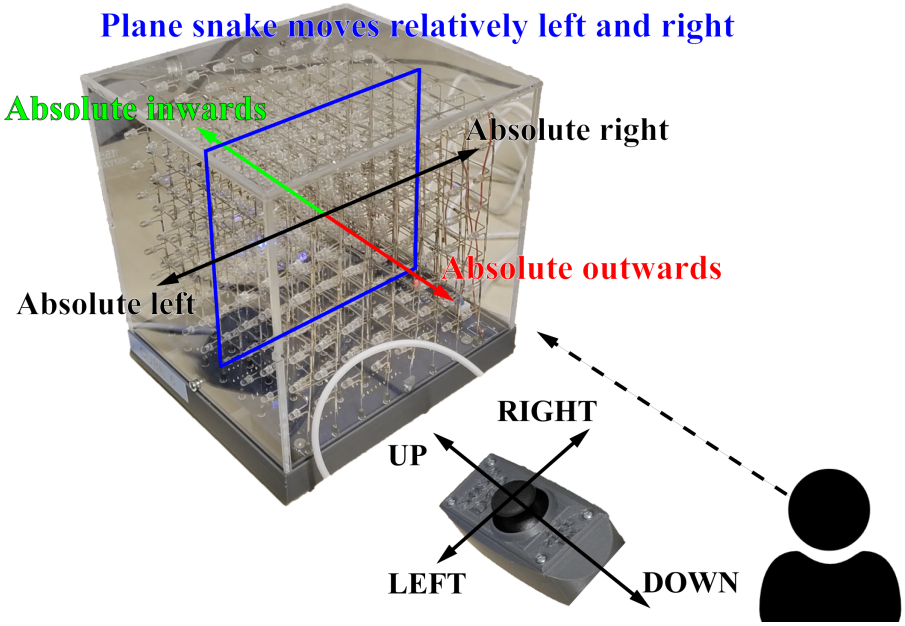

Figure 2: Labelled image of cube and joystick showing the perspective of the player for section 1.4

2 Implementation

The code is attached in the assignment. `project/ARCHIVED` is the latest version we worked on during the lab. `project/LEDCube` is the latest version, where we have added commenting and done some other minor refactorings, and is the version this report references.

2.1 Hardware

2.1.1 Specification of hardware

The code assumes it is run on a STM32 Nucleo F303RE microcontroller [1] which is connected to other hardware as specified on the coursework brief [2].

2.1.2 Range of joystick

Using GDB, it was found that the joystick channel outputs ranged from roughly 1000 to 3000.

2.2 Coding style

2.2.1 Prefixes

In order to organise our code, we have prefixed each of our variables and functions to represent what aspect of the program they are utilised for. For example, all functions that are directly used to interact with the hardware are prefixed with 'Hardware_'.

2.2.2 Commenting

Comments have been written for:

1. Sections (multiple lines) of code with an easily expressible purpose
2. A line of code which has a function that is not immediately obvious
3. Code that may be easy to understand but is logically significant

2.3 Logic of code

2.3.1 Overarching logic

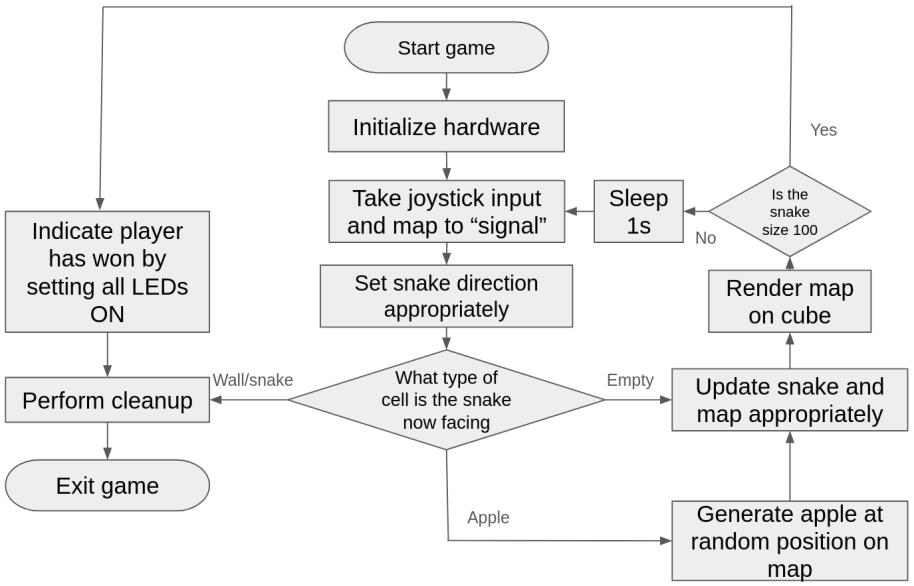

Figure 3: Flowchart showing the overarching logic of the program

2.3.2 Interaction with hardware

There are 3 functions defined in our implementation that directly interact with the hardware - `Hardware_Setup`, `Hardware_ReadChannel` and `Hardware_RenderCube` - which all do so using the `libopencm3` library.

1. `Hardware_Setup` - Runs at the start of the program performing set up necessary to interact with hardware
2. `Hardware_ReadChannel` - Reads raw values from a channel, which is used for getting inputs from the joystick
3. `Hardware_RenderCube` - Renders the cube using the global array `Cube_map`

2.3.3 Data structure of the snake

The game needs to store the positions of each of the segments of the snake, and which segment is joined with which. We chose to use a linked list to do so, as it makes moving the snake a relatively computationally inexpensive operation.

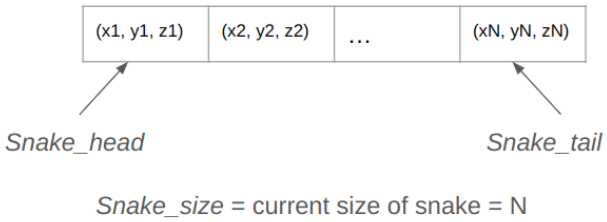

Figure 4: Representation of the snake list labelled with variables used

2.3.4 Data structure of the cube map

The state of the cube is stored in a 64 (8-bit) character array - `Cube_map` - where each character represents a row, each bit represents a cell and the rows are ordered in the same order they are sent as the messages would be sent in a frame. This makes it simple for the function `Hardware_RenderCube`. However, it means it is slightly more complicated to manipulate and read. The logic to manipulate and read it is implemented through the functions `Cube_SetBitAt`, `Cube_ClearBitAt` and `Cube_IsBitOnAt`.

2.3.5 Mapping joystick channel inputs to useful "signals"

The return value of `Hardware_ReadChannel` is still meaningless data in the context of the game logic. As specified in section 1.4, it should be mapped into signals `LEFT`, `RIGHT`, `UP`, `DOWN` or `CENTRE`. The enum `DirectionChange` is defined in order to represent these signals, and the `Controller_GetDirection` function does the mapping using the information in 2.1.2. Due to the arbituary order of the conditionals, the `LEFT` and `RIGHT` signals get priority.

	channel1	*channel2*
> 2500	UP	**LEFT**
< 1500	**RIGHT**	DOWN
> 1500 and < 2500	CENTRE	CENTRE

Control signals in **bold** take priority if multiple conditions are satisfied

Figure 5: Table showing channel input to direction signal mappings

2.3.6 Setting direction of the snake

The global 3-length array `Snake_currentDirection` stores the current direction of the snake, where each element represents how much the snake steps in the x, y and z directions respectively in the next pass. The `Snake_Turn` function implements the logic described in section 1.4, using the current `Snake_currentDirection` and a `DirectionChange` argument to set the appropriate new `Snake_currentDirection`.

2.3.7 Start of program

The snake always starts with its tail at (0, 5, 5), its head at (1, 5, 5), a length of 2 and a `Snake_currentDirection` of (1, 0, 0).

Snake_currentDirection	List of segments (*Snake_head -> ... -> Snake_tail*)	*Snake_size*
(1, 0, 0)	(1, 5, 5) -> (0, 5, 5)	2
(0, 1, 0)	(2, 5, 5) -> (1, 5, 5)	2
...	(1, 6, 5) [APPLE WAS HERE] -> (2, 5, 5) -> (1, 5, 5)	3

Figure 6: Table showing potential start of program

2.4 Improvements we could make

2.4.1 Apple generation

Currently, the generation of apples is done by randomly picking a cell until an empty cell is picked. When the snake covers a large portion of the cube, this will start to get computationally costly and potentially cause noticeable delays. One way to reduce this is to pick a random cell, and if it is not empty, loop through the following cells (circularly) until an empty cell is found. This way, the maximum number of checks of whether cells are empty is limited to 63.

2.4.2 Checking if set bit is a snake segment or the apple

Currently, to differentiate between whether an ON cell is a snake or an apple, the program must loop through the list of segments of the snake to check if there is a match. An easy fix to this would be to have a global variable containing the current position of the apple, meaning the program only has to perform a single check to decide.

2.5 How we would have implemented the extensions we didn't do given more time

1. **Extend the snake solution to include wrap-around** - In `Snake_Step`, set the new position using modular arithmetic (and remove all now unnecessary code that relates to the WALL cell type)

2. **When the snake dies, display the length of the snake before resuming the game** - Create a new function that is run at the start of the function `Game_Over` that maps each possible `Snake_size` to the corresponding appropriate state of `Cube_map` and renders it

3. **Make the congratulations for completing the game as extravagant as possible!** - When the game is won, instead of setting all LEDs ON pass control to a function that displays an "extravagant" animation

3 Results and Reflection

3.1 Results

By the end, we believe (see section 3.2) we successfully implemented a working snake game on the LED cube that incorporates extension 3. Demo gameplay is attached in `report/LEDCubeDemo.mp4`.

3.2 Limited testing time

We only had 10 minutes of testing, meaning that there could be unidentified bugs in the program, especially for a longer snake (we only managed to get a highscore of 7).

3.3 Overall reflection

We were very rushed for time in this project, especially as we missed 1 hour of a lab due to it conflicting with an urgent house viewing. Therefore, we were very happy to have done as much as we did. However, this meant that there are certain aspects in our design and implementation which we could have improved were we given more time.

3.4 Evaluation of the design

3.4.1 Handling of the snake death

The program currently exits after the snake dies, which means player has to manually restart the program to play again. A better way to have done this is to restart the game after a set time period or even better, after the joystick takes in a non-centred input (so the player can decide when to restart).

3.4.2 Control of snake

Our design for how our snake is controlled was not as intuitive as we initially thought it would be, although it was not too hard to get used to after a few goes.

One way we thought we could improve our snake control design is to also make the UP and DOWN controller signals rotate the snake in the same way that LEFT and RIGHT controller signals currently do, except in the other vertical plane. This allows the player to always be able to control the snake in all possible directions (the current controls do not allow you to go up and down when the snake is travelling inwards or outwards), and also improves consistency of control.

Snake is moving / Signal received:	In plane facing user	In the other vertical plane
LEFT	Relative left	Absolute left
RIGHT	Relative right	Absolute right
UP	Absolute inwards	Relative inwards (turn clockwise)
DOWN	Absolute outwards	Relative inwards (turn anticlockwise)
CENTRE	No direction change	No direction change

Figure 7: Table showing the direction the snake would travel depending on its current direction and the joystick signal received when following the suggested design in section 3.4.2

3.5 Evaluation of the implementation

3.5.1 Positives on our implementation

Overall, we believe that our implementation logically and modularly organised the data and logic used into different structures and functions such that we can easily refactor the code.

3.5.2 How we could further improve modularity and organisation

Currently, our implementation groups statements and functions through prefixes. One way we could further improve our modularity would be to separate different groups of code into separate files instead.

3.5.3 Comments about other improvements we could've made

Of course, there are also many specific areas of our implementation that worked but could be improved, some of them we have listed in section 2.4. However, most of them were compromises to save time, and due to the modularity of the code, should be easy to fix through refactoring. For example, the first improvement listed there could be realised by modifying just the `Snake_Turn` function.

References

[1] R. Kirk. Cs132 lab 5 - software control. `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs132/labs/lab5/`, October 2023.

[2] R. Kirk. Cs132 lab project - led cube. `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs132/labs/ledcube/`, September 2023.