SOFT356 C2: Snake 3D

Student No: 10550420 / Jamie Everett

Please note: This readme is best viewed in GitHub, available here: https://github.com/SOFT356/SOFT356-C2-JamieEverett.

The video report for this project can be streamed here: https://youtu.be/4WVVP-jb2mU.

Software Used for Development:

- Visual Studio 2019
- nupengl.core v0.1.0.1 (NuGet)
- glm v0.9.9.600 (NuGet)

How To Use Snake 3D:

Key bindings

Snake3D comes with the following controls to provide player interaction with the game:

Movement Controls:

Input	Action
Up (↑)	Face snake north
Down (↓)	Face snake south
Left (←)	Face snake west
Right (→)	Face snake east

Other Game Controls:

Input	Action
P	Pause/resume game
Escape	Close the program
Mouse scroll	Move camera forward/back

How to Build

To open the project in visual studio, open the *Snake3D.sln* file. Then, select *Debug* in the solution configuration dropdown and *x64* in the solution platform dropdown. Finally, build the project by either pressing *F6* or selecting the build option in the build menu.

How to Play

To start Snake3D, launch the executable file *Snake3D.exe*, which can be found in the *executable* folder. If you're instead running the executable that you created from the previous step, you will need to first copy the *sounds* folder (found at *Snake3D/sounds*) and paste it into the build folder (*x64/Debug*), otherwise the game will play without sound effects.

Once running, the player will see a prompt in the console window asking them to choose a map size, the map sizes break down as follows:

Map Size	**Map Dimensions**	**Number of Blocks (% increase from previous size)**
Small	7x7	49
Medium	9x9	81 *(+~65%)*
Large	11x11	121 *(+~50%)*
Huge	13x13	169 *(+~40%)*

Once a valid selection has been made, the requested grid will be generated with a snake (*see figure 1.0*) and food item (*see figure 1.1*) positioned on top of it. The game will start once the player presses a directional input (shown in the key bindings section).

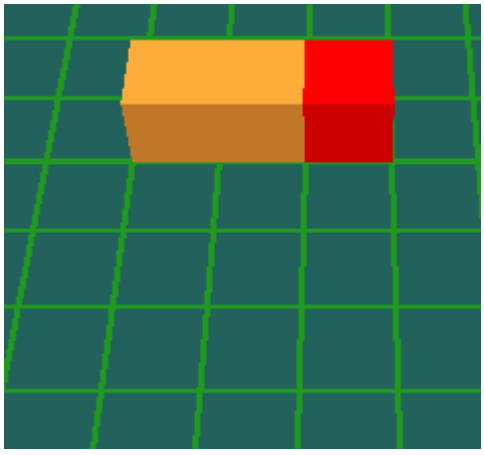

Figure 1.0: Snake movement

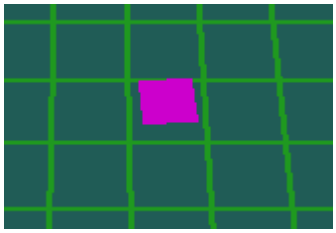

Figure 1.1: Food animation

One the game has begun, the aim is to keep eating food blocks until there are no empty spaces, this satisfies the win condition of the game. If the snake's head either tries to leave the map boundary or hits another body segment, the lose condition is met and the player receives a game over message.

At the end of a game (through either a win or loss) the players final score is displayed in the console. The player is then asked if they would like to play again and, if yes (Y), the program repeats from the map

selection prompt. If the response is no (N), the program exits.

Code Structure:

```
Snake3D
├── sounds
│   ├── death_sound.wav
│   ├── eating_sound.wav
│   └── win_sound.wav
├── Food.cpp
├── Food.h
├── Snake.cpp
├── Snake.h
├── Snake3D.cpp <-- ENTRY POINT
├── Snake3D.h
├── SnakeBody.cpp
└── SnakeBody.h
```

Snake3D.cpp

The code is made up of 4 cpp files, the main one being *Snake3D.cpp*. From this file the game logic is executed, and frame updates are coordinated. *Snake3D.cpp* is also responsible for creating and destroying instances of the *Snake.cpp* and *Food.cpp* classes, while the *Snake.cpp* class is responsible for adding new *SnakeBody.cpp* objects to the current *Snake.cpp* object.

Snake.cpp

The main functions of this class are to draw the snake (looping though each *SnakeBody* child), update the snake's body position, move the snake in a specified direction and detect collisions (with either food, the boundary or the snake itself).

SnakeBody.cpp

Apart for the getters and setters for the XYZ positions and RGB values, the only function in this class is to draw the respective *SnakeBody* part. Each body part is constructed of a simple cube, the colour of which is either orange (default) or overridden to a different colour through the use of *setColour()*. An example of this is in *Snake.cpp*, where the first SnakeBody element in the body vector is set to red (to distinguish the head from the rest of the body).

Food.cpp

The final class is the food class. Like the *SnakeBody.cpp* class, apart from the getters and setters, the only function in the class is to draw the food object. Inside this draw method, the animation for the up and down floating motion and rotation is performed by modifying the values passed in to *glTranslatef()* and *glRotatef()*, respectively.

The display loop

display(GLFWwindow window)* <-- (inside *Snake3D.cpp*)
The display loop is where most of the program's runtime is spent as it renders the snake, food and grid elements as well as checking for collisions (*snake->detectCollisions*). It is also responsible for linking callbacks, handling frame timing/update frequency and updating the camera position.
Most of this function is repeatedly called until either *Escape* is pressed (which calls *exit(0)*) or a win/loss condition is met (which ends the current game and returns the player to the map size selection prompt).

Below is a high-level flow diagram of the program execution logic:

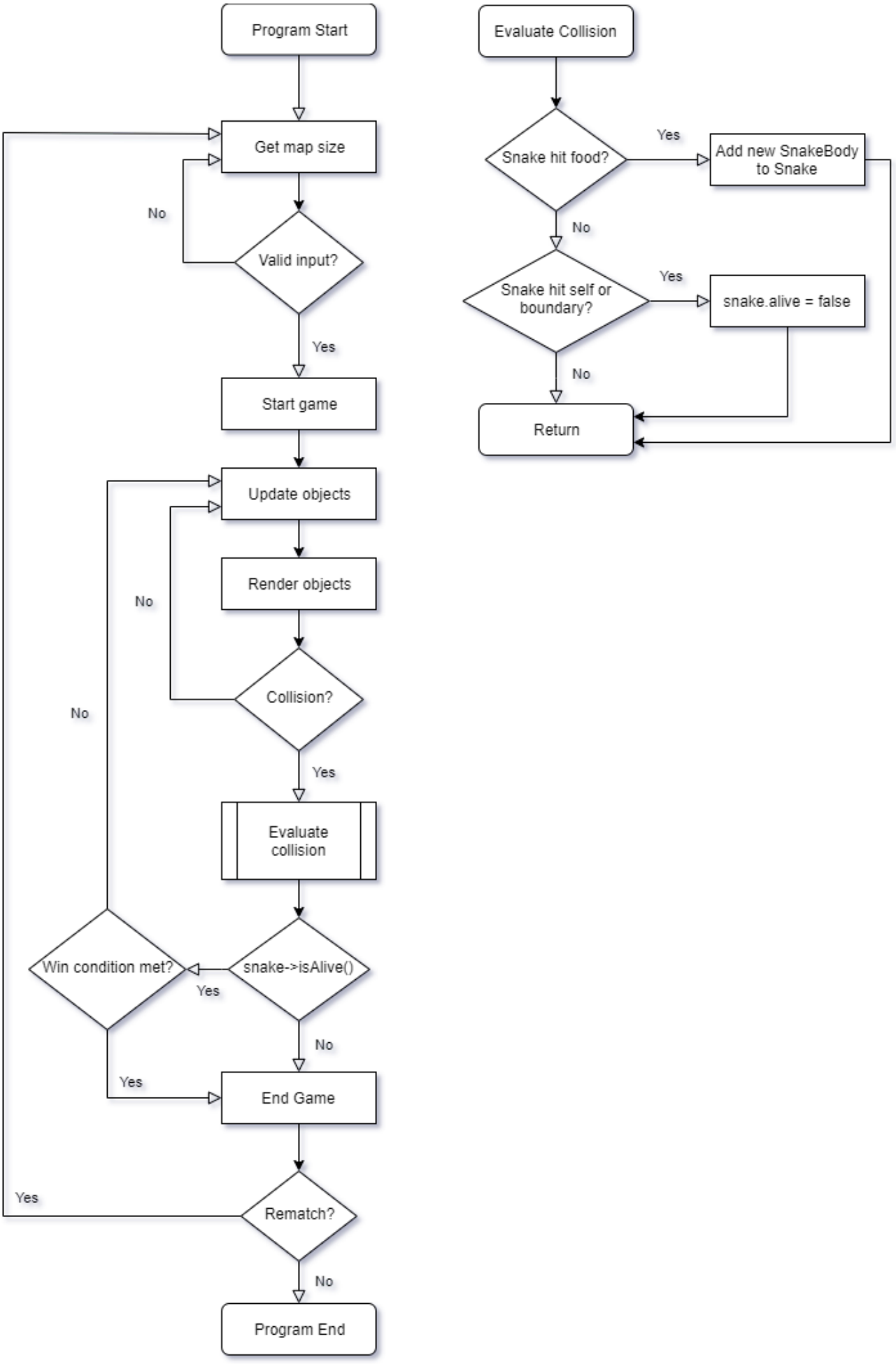

Figure 2.0: High-level flow diagram

Optimisations and Other Features:

The snake and food objects are created at the start of each new game using the *new* keyword (and therefore created on the heap). To prevent memory leaks, after each game is finished the objects are destroyed using *delete*.

As an optimisation, to prevent the food object from being destroyed and a new one created every time the snake eats it, the same food object is used throughout an entire game. Once the food is eaten, it's X and Z coordinates are randomised (and checked to ensure they are not underneath a snake section), which takes less processing time than destroying and recreating the object.

I have added sounds to my game as it provides another dimension of feedback, which in turn makes for a greater user experience. Different sounds are played if a snake eats a food block, if the snake dies and if the player wins.

All inputs in my game have validators which ensure there is no undocumented behaviour. If an input fails to pass a validator, the prompt is repeated.

In order to create the clunky 'retro' snake movement, I added an update limiter (using the delta time between frames). This limiter is set to 5 updates per second, which I found mimics the original 2D game well. However, to prevent the food animation from also appearing clunky, I isolated the update limiter to just the snake updates, so the rendering happens as fast as the machine can run (typically around 2.5-3k fps) which keeps the food animation looking smooth.

Finally, I added a zoom option which is controlled by the scroll wheel. When the camera is fully zoomed in, the player appears to be standing at the same level as the grid, however when zoomed out, the camera is positioned to look top-down onto the grid. This is to recreate a 'faux-2D' perspective as a reference to the original 2D game.

Program Background:

I chose to create a 3D version of the popular 2D snake game because I wanted to build something that could be played, with objectives and scores.

I got this idea when thinking about what I could create in OpenGL that would be fun to create and play whilst still being feasible with my current knowledge of OpenGL and C++.

I created this project from scratch, building it up as I went; starting with a grid, adding the snake, adding the controls, adding the food and finally implementing the game logic.

Future Improvements:

I believe this project can be improved further with the addition of several features.

One improvement would be to add a difficulty mechanic, where the snake speeds up as the total number of food collected increases. This would be fairly simple to implement, as I already have a *limitUpdates* variable, so combining this with the *snake.body.size()* function would be straight forward.

I would also like to add more camera options, such as key bindings for *A* and *D* which rotate either the camera or the grid. I'd also like to add first and third person viewpoints which would follow the snake as it moves around the 3D environment.