Snake 3D – Documentation

Table of contents:

Part I – Description and design

* Description
* Architectural decisions
* Architectural design
* Programming approach

Part II – Diagrams

* User story diagram
* Class diagram
* Activity diagram
* State diagram

Part I – Description and design

Description

Snake3D is a 3D rendition of the classic game, Snake. It’s played in a first-person view and controlled by the WASD- and/or arrow keys. The game will have different views for different parts of the game’s experience. It will also utilise files, such as JavaScript Object Notation (JSON) and Comma separated values (CSV) for saving data. The game will run in 60 frames per second (fps) to ensure a pleasant viewing experience by the user.

When selecting a new game, the user will be prompted to choose elements of the world, such as width and height. These variables will also be present in the scores-section of the game, where the player can view their high scores. A filtering method will be present in this section of the game as well.

Architectural decisions

The game will be programmed with Java, utilising JavaFX for the graphical user interface (GUI) and Maven for running the application. Java is used for its wide-spread use in software, and my personal experience in it. JavaFX is used over Swing, as it’s more recent, bringing better features for overall code quality, readability and reliability. As the program is being developed using Visual Studio Code (VS-Code), Maven is used to run the application.

Git and GitHub is used for version control, as it’s what I’m most experienced and comfortable with. In addition, VS-Code supports Git excellently.

For data storage, JSON- and CSV-files will be used. This is due to security not being too important, as the game’s produced high scores will not be recorded globally. Error handling will be present for both files, having backup copies in case corrupting occurs. In case both the original and backup files have been corrupted, both will be erased. In case the files are not detected by the application, they will be created. The files will be accessed by the application, stopping the user from too easily being able to inject corrupt files to the system.

The renderer will utilise matrix calculations for calculating vertex-locations on the GUI. Culling, i.e. the selection of what to render at any time, will likely not be sophisticated. The game will likely not have more than a few thousand vertices present at any point, meaning that calculations can be made for every vertex. If this proves not to be the case, further investigation into culling techniques will be made and implemented.

To avoid floating point errors when moving the snake, the game will use integers for movement. This will limit the size of the world due to integer overflow, but this will be addressed with a limit to world size.

Architectural design

The system will used a layered approach to ensure system functionality and increased security. The user will only be able to interact with the system’s views, which themselves will be able to communicate with other system components. The overall design philosophy will be as follows:

A diagram of a software system

Description automatically generated

The user requests a change to occur (e.g. turning the snake left). This request is received by the view, which then verifies the request. The verified request will be sent to the game, where it will be processed. This change is read by the game engine, allowing for the renderer to render the change to the view. This change is then displayed to the user.

Programming approach

The system will be produced without hurry, as for this project, both the scope and resources are fixed. All components will be versioned, including this plan. This will ensure that documentation is available for everyone accessing the application.

The code will be written using test-driven development (TDD) using JUnit as the testing tool. This will ensure that the system will have a regression test – environment, that can be run at any time. Using TDD also entails that no code will be considered “done” if there exist no tests. The tests must also pass before the component is considered done. If during a regression test a test fails, fixing this will become of highest priority, i.e. no new features will be implemented until the code is fixed.

Additionally, a piece of code is not considered done if it’s methods, classes etc. don’t have documentation notations attached to them. This is to ensure that returning to the code is easier and that refactoring could be done by other people as well. Comments inside the code will not be accepted, but instead using good variable names will be emphasized, i.e. the code should be readable.

The system will be developed using Agile methods, i.e. “working” versions of the system will be made first and additional features will be added later. Code refactoring will also be present, as even though all code will be written by myself, this doesn’t mean that I cannot improve it later.

Finally, contrasting the previous point of agile development, the software will emphasize the production of documentation. This is to better show how the code was made and what though-processes were present at that moment of time.

Part II – Diagrams

User story diagram

Class diagram

Activity diagram

State diagram