Snake 3D – Documentation

Table of contents:

Description and design

* Description
* Architectural decisions
* Architectural design
* Programming approach
* Algorithms & data structures
  + Rendering
* User story diagram
* Class diagram

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 application

Description automatically generated

The user requests a change to occur (e.g. turning the snake left). This request is received by the view, which calls the corresponding method of its controller. The controller can now either produce the change itself, ask another service to provide the required information, or ask the application itself for the required data. If a service is called upon, it also might access data from the application. Finally, the controller with its calculations will produce the rendering, sending this data to the view that displays it to the user. In the background, the application is being asked by the game engine to update the display at a fixed framerate. In the case of this application, it’s 60 times/second.

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.

Algorithms & data structures

Most of the game will be utilising matrix calculations. To get good rotations, the game will also use a class called DoubleVector3D. This will, instead of integers, use doubles to represent its coordinates. This leads this class vulnerable to floating point errors, hence it will not be used for other than rendering, i.e. only the camera will be utilising it. This fact will also mean that the class will not be “cluttered” with many functions, as it will never actually move.

The matrices used to calculate the rotations will be the following:

These are well known rotation matrices, where given the angle – given in radians – the calculation will produce the correct rotation for the 3D vector. Using these for the camera will allow us to perform a “smooth” turn instead of it occurring immediately.

The snake will be represented with a collection of segments. The application will use LinkedLists to represent these, as they have constant time updates to their lengths and have a constant time access to the first element. The size of the list is also a constant time operator. These are primarily the only two functionalities needed by the application from the lists. It’s also good to remark that each element needs to be accessed every time the scene is rendered, which will be accounted for to make sure it only accesses each list-element once. Finally, the snake will realistically never grow very large, meaning that performance will likely not be affected by small blunders.

The game will utilise a new data structure – vector tuples – for a more lightweight collection for a pair of 3D vectors. As tuples don’t exist in Java, these will be added manually. This will also enable the addition of custom functions, making the code more readable.

When calculating edges, it’s easiest to get them with three collections of the world’s vertices. The first will go in the order: x, y, z. The second: y, x, z. The third and final: z, x, y. With these orderings of the vertices, producing the edges for rendering a 3D grid on the world’s walls will be easier.

The vertices of a cube entity will be represented with an ArrayList of double-precision 3D vectors. These will be in the order (looking head on at the cube):

* Top front left, top front right, bottom front left, bottom front right, top rear left, top rear right, bottom rear left, bottom rear right.

Each game will have a random seed generated from the system’s nano time method. This will be in a long format, as to maximize randomness and the number of seeds available.

The world will have ‘eatable’ entities, i.e. entities that when collided with will cause an effect. One such entity will be the apples, whose effect is increasing the player’s score. These apples will spawn randomly, based on the game’s seed. Entities can only spawn on non-occupied spots, so if there are no more spots left, no entity will spawn. The limit to the number of apples is calculated based on the world’s size. The equation is:

This equation will give a reasonable number of apples present at the world, regardless of its size. It also ensures that apples will spawn even if the equation would produce a value less than 1.

Rendering

The rendering process will rely on normal 3D rendering algorithms. The matrix multiplication for getting the non-perspective corrected coordinates will be the following:

The individual matrices are the following:

* Starting vector: The 3D vector that calculations are applied to.
* Translation matrix: The position of the matrix relative to the camera’s position. The matrix is: (4x4 matrices are used for translations).
* Rotation matrix: The multiplication of the previously mentioned rotation matrices, expanded to 4x4 matrices with 0s in all new slots except the bottom right corner. The order of multiplication is , i.e. first application is the rotation matrix of the x-axis.
* Camera matrix: The camera transform matrix applies the focal length transform – *f*, screen pixel-size correction – *p* and sensor size adjustments – *s*. The matrix is as follows: . The exact values of the focal length and sensor size will be experimented with to get a good idea.

Once the non-corrected rendered vector has been calculated, a final calculation for perspective correction is needed. This is also represented with a matrix multiplication, giving the following equation:

These calculations must be done separately, as the perspective correction matrix relies on the z-value of the non-corrected vector. The perspective correction matrix is:

These calculations are applied to all position vectors of the world and its inhabitants.

The rendering of cubical entities will be done utilising the *fillPolygon* method of JavaFX’s GraphicsContext2D. This method takes in three parameters – an array of x-coordinates, an array of y-coordinates, and the number of vertices of the shape. A cube can be represented with 6 polygons, one for each face. The vertices must be ordered correctly, such that the drawing succeeds correctly, i.e. no diagonal jumps. The faces’ vertex sets are the following:

* Front face: 0 – 1 – 3 – 2
* Rear face: 4 – 5 – 7 – 6
* Top face: 0 – 1 – 5 – 4
* Bottom face: 2 – 3 – 7 – 6
* Left face: 0 – 2 – 6 – 4
* Right face: 1 – 3 – 7 – 5

Each number represents the index of the vertex in the CubeEntity’s *getVertices* method.

User story diagram

Class diagram