|  |  |
| --- | --- |
| **Project title** | **Perihelion** |
| **Author(s)** | **Bojor Barbu** |
| **Group** | **30421** |

# Task Description

The aim of this project was to develop a relatively simple 2.5D game written in java and using the JOGL API to provide OpenGL capabilities, with possibility for easy extension. The game can be packaged in an executable .jar file with all needed dependencies for easy deployment (currently only supports 32/64-bit machines running Windows).

## 1.1 The engine

The first step before creating the actual game consisted of building the game engine to power it, allowing for the creation and management of various game entities, input handling as well as the usage of visual and audio resources such as images/textures, animations, fonts, user interface, sounds and music.

The backbone of the engine is represented by the **game loop**, implemented via a static class containing a thread running continuously while the program is running. This thread is coded to attempt 60 updates (logical update + rendering unto screen) per second on the currently relevant game entity, and sleep for the remaining time, if any. In the current implementation the constant 60 updates per second is vital to the functioning of the game since both animations and movement are calculated based on this. In case something caused an update to take more time than expected and some updates were lost, the game loop attempts to make up for the lost time by performing up to 5 consecutive updates in a single iteration.

The **game loop** can, at any moment, find itself in one of 3 different states:

* 1. The menu state, in which the entity it performs the updates on is

the main menu of the game

* 1. The game state, in which the entity receiving updates is called a game world, a class holding all the objects and information that makes up a given level in the game
  2. The exit state, an intermediate state during which the game loop stops, resulting in program termination

Other core classes of the game engine include the **Renderer**, **KeyInput** and

**MouseInput**.

The static **Renderer** class handles the creation of a game window, the initialization of several OpenGL elements required by the JOGL API, and provides several useful graphical functions such as displaying images and text, allowing for the creation of 2D graphics.

The static **KeyInput** and **MouseInput** classes provide implementations for the JOGL keyboard and mouse listener interfaces, adding extra functionality tailored to the needs of the game engine.

Once this basic framework was established, work could begin on the specific implementations required by the particular game.

The basic building block provided by the engine is the abstract **GameObject** class. This class lays the foundation for creating a functional entity in the game world by including various useful attributes like its position in the game world, its dimensions, sounds and animations and two vital methods, update and render.

Since the game opts for a 2.5D isometric perspective, the game world is split into 3 categories of objects: the **terrain**, **decorations** and **game objects** (the latter two categories are treated as one during rendering).

The **Terrain,** as the name implies, represents the ground upon which all the other game objects are placed. It is composed of a matrix of tiles with set dimension, which can be located at different heights. The coordinate system of the tiles in their matrix is also used by the other game objects for easier spatial orientation.

**Decorations** represent miscellaneous game objects that can serve as obstacles, hence an easy way to account for them is needed. They are also kept in a 2D array to help with collision detection, pathfinding.

The third category groups all the remaining game objects present in the game world that do not fit in any of the previous two categories, these include the player object and enemies.

## 1.2 The game

### 1.2.1 The Main Menu

The actual game is a small, wave-based 2.5D action game in which the player is tasked to survive for as long as possible against an increasing number of enemies.

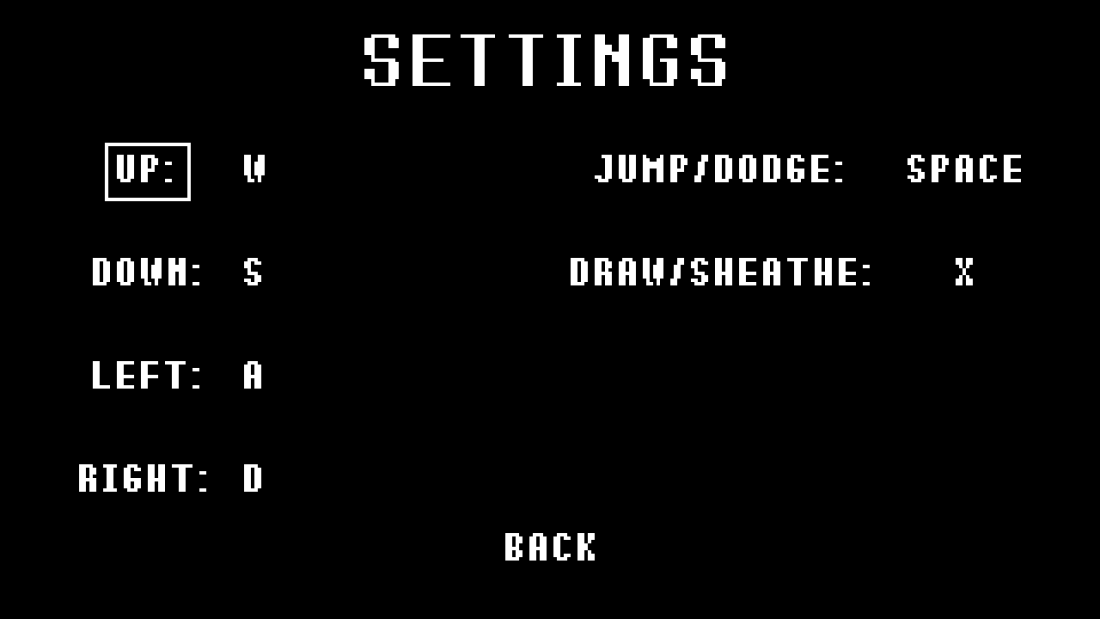
The entry point into the game, from the player’s perspective, is the **Main Menu:**



All menus in the game are navigated using the **w** and **s** or **up** and **down** arrow keys. The current option is highlighted by a white rectangle. In order to select the current option the player can use the **enter** or **space** keys, or **left-click** the mouse.

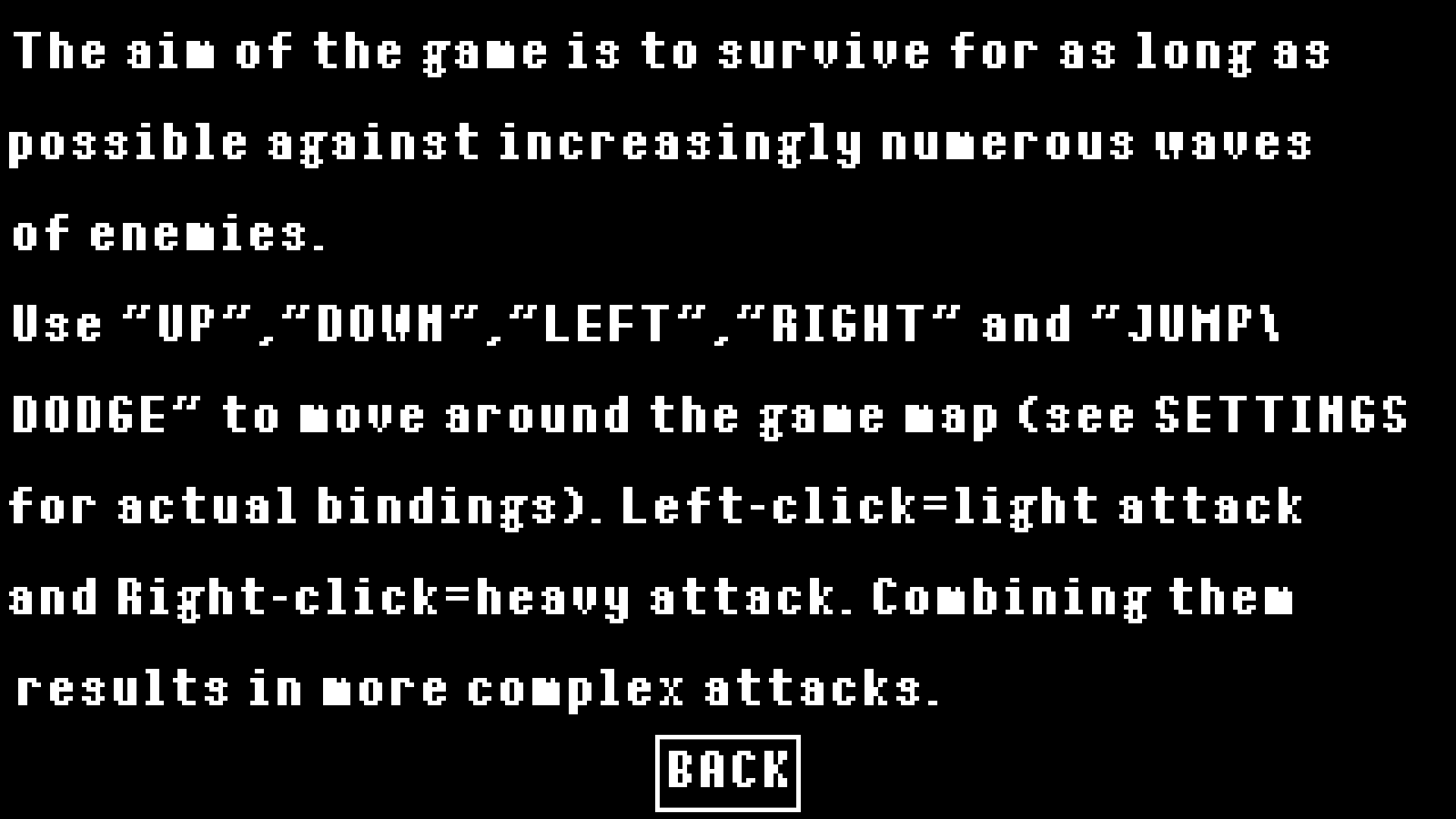
Selecting the **New Game** options will initialize a new instance of the game world and start the actual game.

The **Settings** option will bring up the settings sub-menu. Currently the only preferences that can be changed are the keyboard bindings, but mouse controls and adjusting sounds levels for sound effects and music are planned. The user’s preferences are stored in text file preferences.txt stored in the same folder as the .jar. In case no file is found on launch, a new file with default settings is created.



In order to change a particular binding, the user must navigate to it and then press enter/space/left-click. The program then waits for the next key press which will be written into the respective slot, and update the control scheme. In case any of the settings has been changed in this way, the preferences.txt file gets overwritten with the new bindings.

The **Help** menu provides a brief description of the game to help the player get started.



### Selecting the Exit Game option as the name implies closes the application.

### 1.2.2 Gameplay

Gameplay consists of attempting to fight off as many waves of enemies as possible

During gameplay the player controls a character capable of moving around, jumping, dodging and attacking. Moving, jumping/dodging and sheathing/unsheathing the sword are done using the keyboard bindings set in the **Settings** menu. The orientation of the player character (left-right/forward-back) is done using the mouse cursor, the player always turns to face the position of the cursor when possible.

**Attacking** is done using the left and right mouse buttons. Combining these two inputs in various ways can allow the player to perform more complex attacks called combos.



Dodging and attacking use a resource called **stamina** (top-left, cyan)**.** This resource will start to regenerate by itself after a short amount of time in which no action was performed. If the player tries to perform an action without any stamina left, they will not be able to. If instead there is some stamina present, but not enough, the player will be able to perform the action, but will be penalized with a longer regeneration time.

The player’s **health** (top-left, magenta)is a finite resource, that gets depleted whenever the player is hit. The only way to gain back health is to attack an enemy immediately after being hit. Whenever the player takes damage, there is a small window of time in which they can regain lost health by damaging an enemy. This effect lasts for a short amount of time, and does not allow for regenerating more health than the player previously had. This previous health level is displayed in yellow behind the current health level.



The number in the middle of the screen indicates the number of enemies present at the moment in the game world. The game only progresses to the next wave after all enemies in the current wave have been destroyed.

The counter in the top-right indicates the **score** achieved by the player. Each enemy is worth 100 points.

At any point during gameplay the player can press the escape key to pause the game and open a small menu that allows the user to resume, exit to the main menu, or quit the game entirely.



Other mechanics include the **attack speed pick-up**, which increases the attack speed of the player for a short amount of time, and **picking up and throwing enemies**. Whenever an enemy has less than a third of their health level left, they can be stabbed by the player using the simple right-click attack. This allows the player to carry the enemy around and throw them. A thrown enemy will fly for a short distance or until it hits an obstacle (rock, wall, another enemy). Upon stopping, the enemy will die. In case it hit another enemy, both will be killed.



# Class Discovery

|  |  |
| --- | --- |
| **GameLoop** | |
| Update game components | World, Main Menu,KeyListener,MouseListener |
| Render required elements to screen | Renderer |

|  |  |
| --- | --- |
| **Renderer** | |
| Render shapes | |
| Render images |  |
| Render text |  |

|  |  |
| --- | --- |
| **World** | |
| Hold information about game objects | Terrain, Game Objects |
| Perform updates on the game objects |

|  |  |
| --- | --- |
| **Main Menu** | |
| Allow player to start a game | GameLoop, World |
| Allow player to change settings/preferences |  |
| Display help |  |
| Allow for quitting the game |  |

|  |  |
| --- | --- |
| **KeyListener** | |
| Keep up do date information about key presses |  |
| Store key bindings |  |

|  |  |
| --- | --- |
| **MouseListener** | |
| Store up do date information about the position of the mouse cursor and button presses |  |

|  |  |
| --- | --- |
| **World** | |
| Hold information about game objects |  |
| Perform updates on the game objects |  |

|  |  |
| --- | --- |
| **HUD** | |
| Provide in game menu functionality | GameLoop,MainMenu |
| Pause/Resume game |
| Exit to menu |
| Exit program |

|  |  |
| --- | --- |
| **Terrain** | |
| Hold tiles | Tiles |
| Information about the size of the playable area |  |

|  |  |
| --- | --- |
| **Game Object** | |
| Holds various useful information like position,  Size, animation |  |
| Provides basic rendering function | Renderer |

|  |  |
| --- | --- |
| **Tile** | |
| Holds information regarding whether it is traversable or not |  |

|  |  |
| --- | --- |
| **Player** | |
| Allows the player to interact with the environment |  |
| Move | KeyListener, World, Terrain |
| Attack | MouseListener, World, Terrain, Enemy, Weapon |

|  |  |
| --- | --- |
| **Weapon** | |
| Allows for attacks | Enemy, Players |

|  |  |
| --- | --- |
| **ComboGraph** | |
| Implements a finite state machine to handle attacks | Weapon, Player, ComboNode |

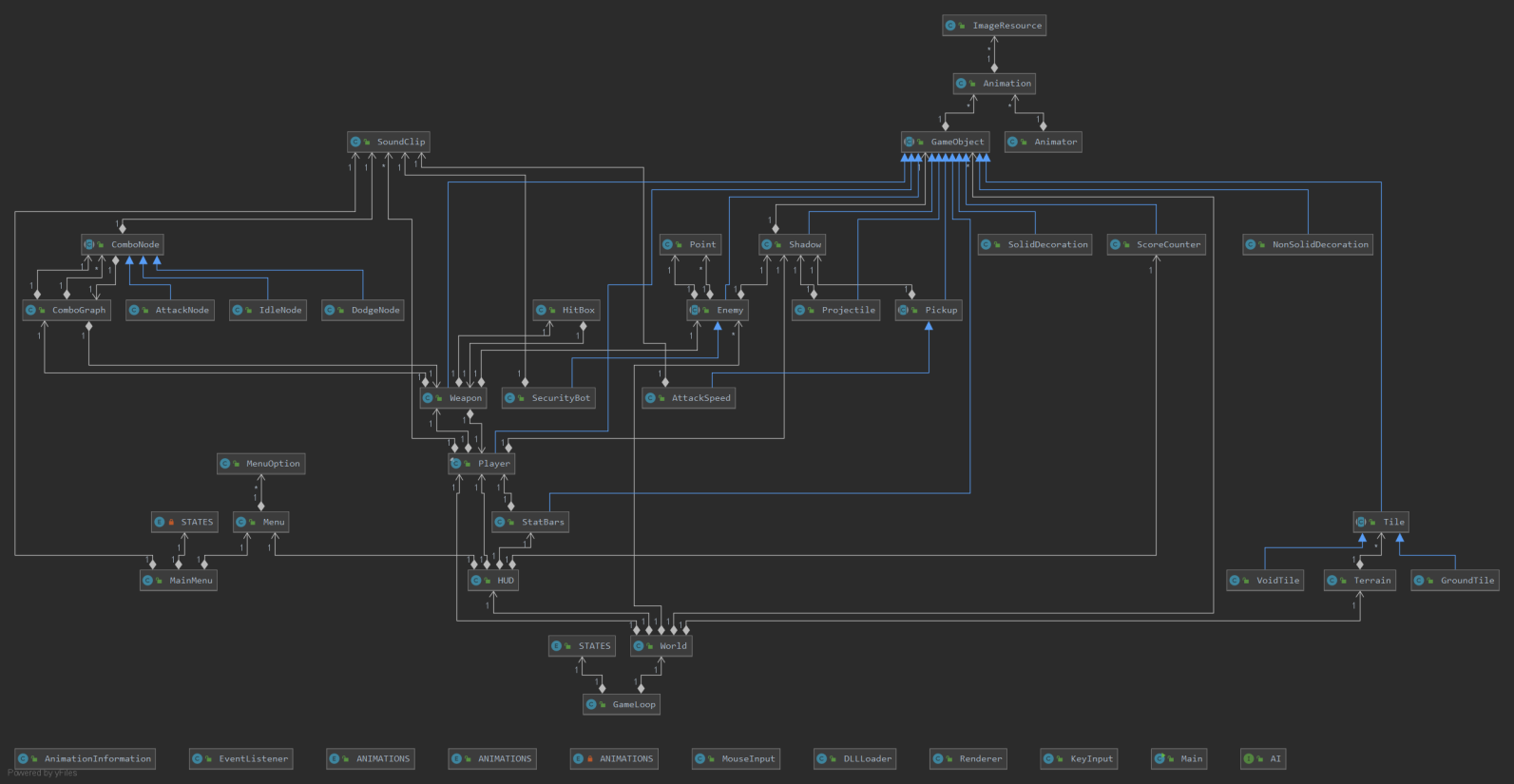
|  |  |
| --- | --- |
| **ComboNode** | |
| Holds information about the current action being performed and next possible actions | Weapon, Player, ComboGraph |

|  |  |
| --- | --- |
| **Enemy** | |
| Follows player, attacks them | World,Terrain,Player,AI |

|  |  |
| --- | --- |
| **AI** | |
| Provides functions for enemy behaviour | Enemy |

# Class Diagram

The top-level class diagram is the following



Individual class diagrams (for more details regarding the fields and method of each class refer to the javadoc):

