Run ‘n Gun

Developer documentation, features to be developed

Application version v0.1

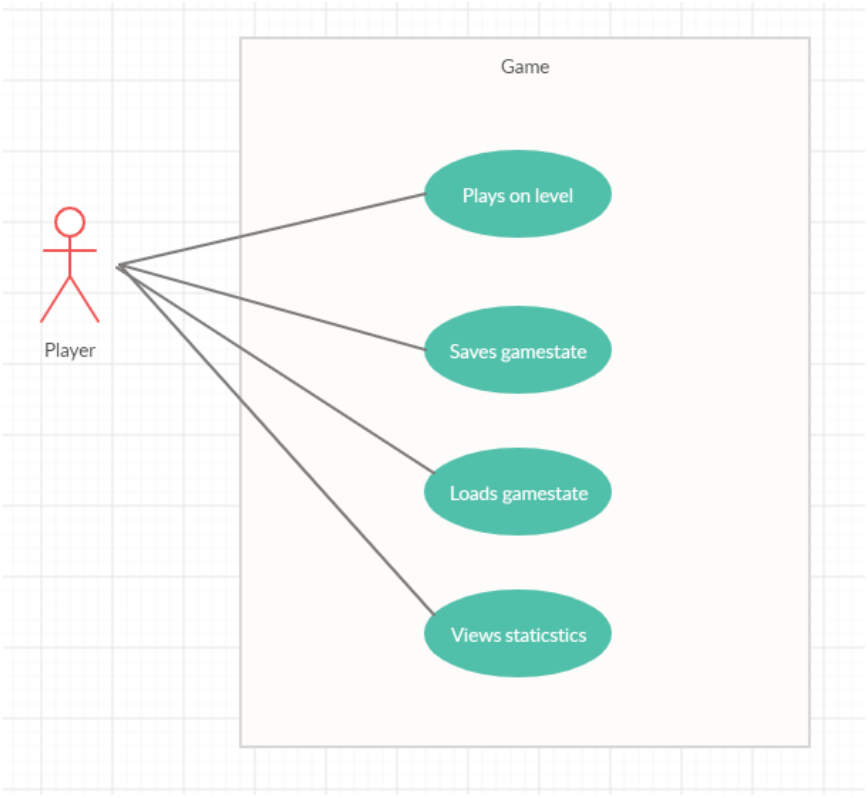
Documentation version v0.1.1

Documentation last updated: 2020.05.17.

Table of contents

Use-case diagram

The following use-case diagram represents the actions an actor can take:

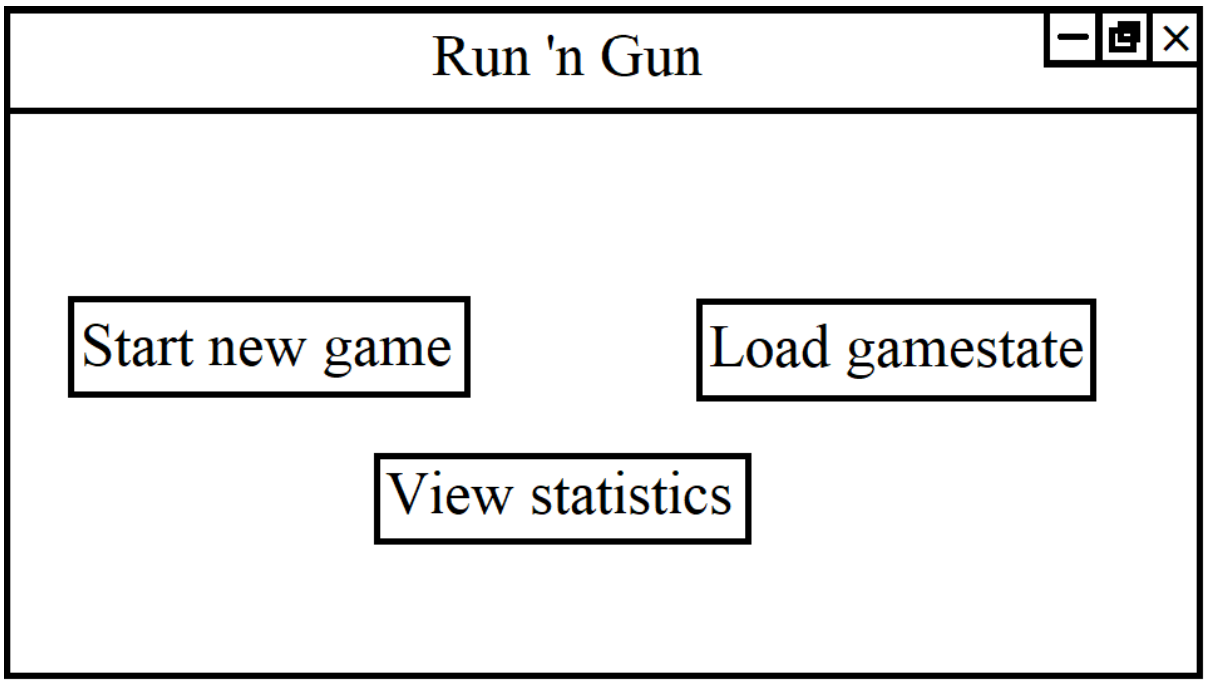


A player (which is all users of this program) is capable of:

- Initiating a level (Playing a level)
- Saving the current gamestate
- Loading a previously saved gamestate
- Viewing game statistics

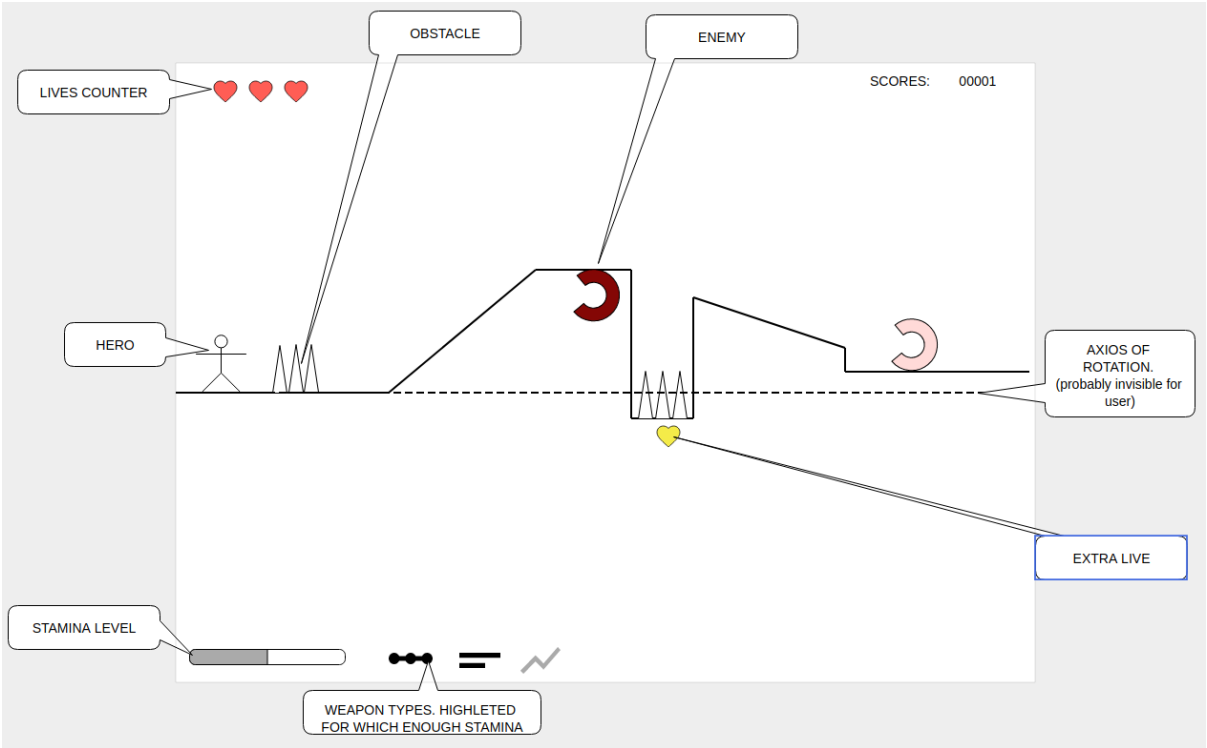
Wireframes

The wireframe for the first screen that the user is presented with:



Game mechanics

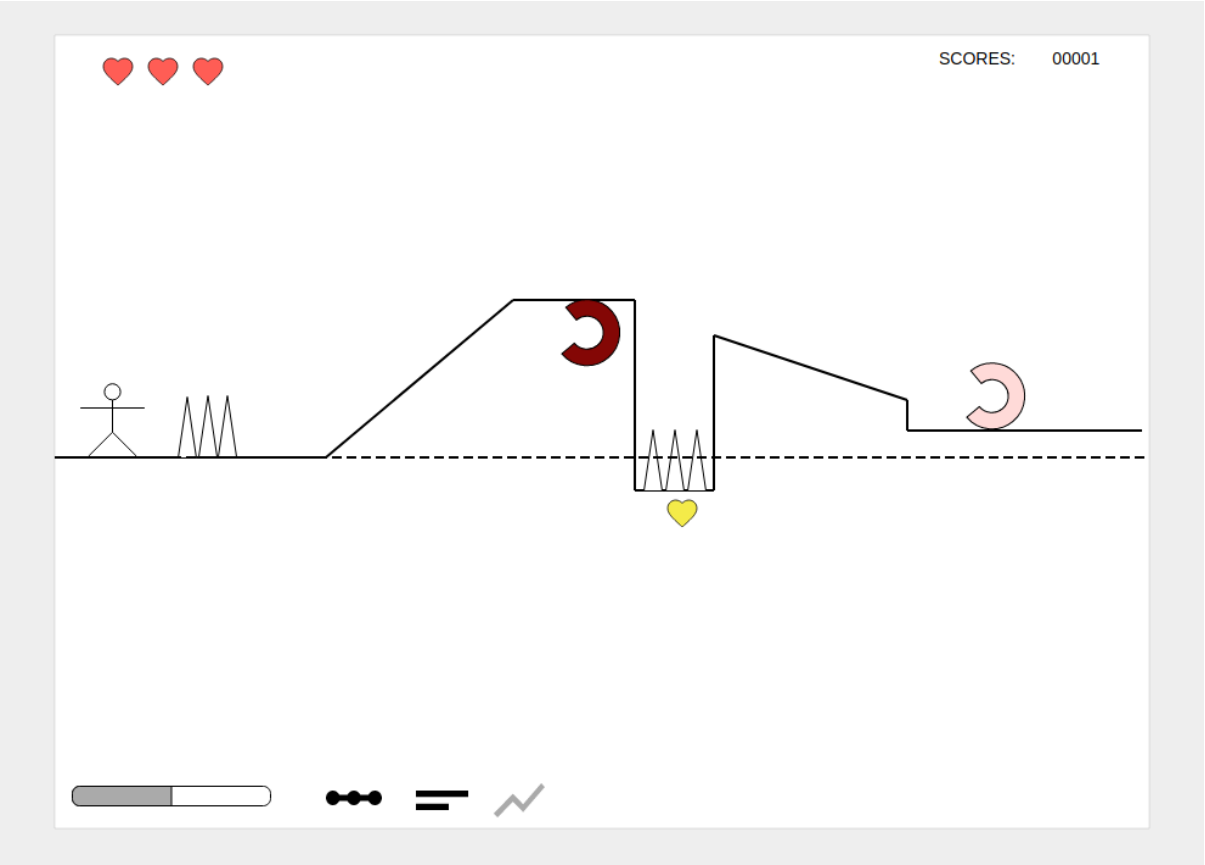
On clicking “Start new game” the user loads the first map of the game. An example for a map is the following:



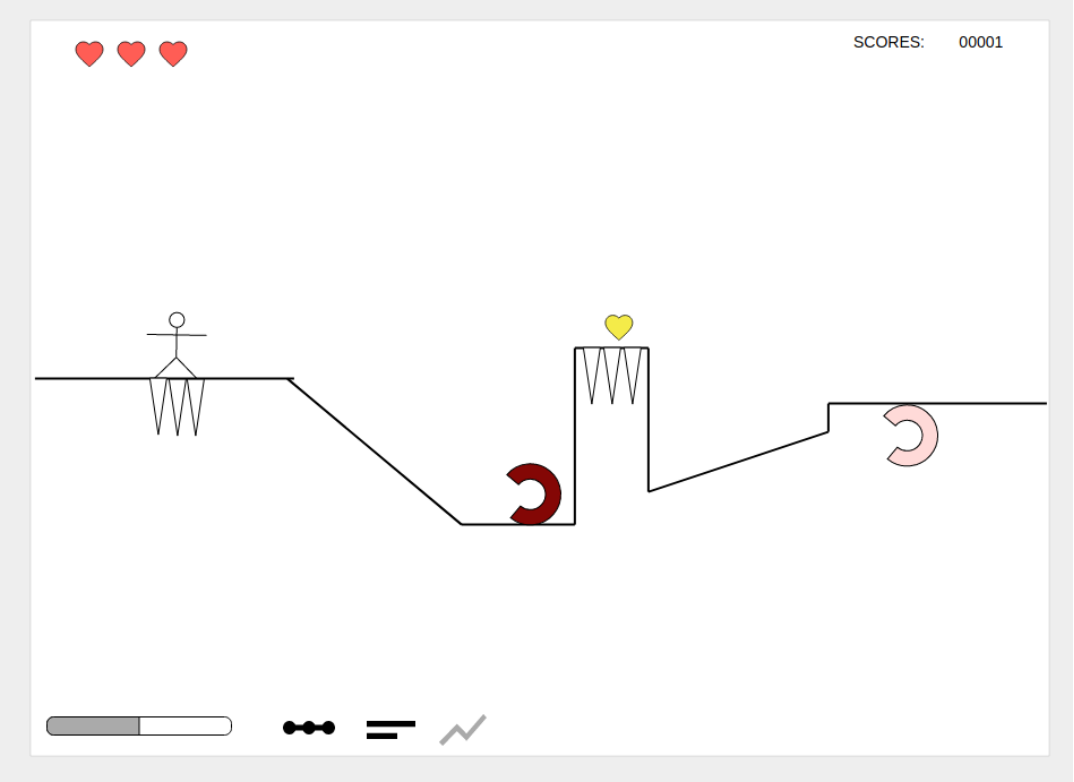
The game would implement a mechanic which “inverts” the map on a predefined axis.

Player can’t jump, can move forward and back, hit enemies and invert axis.

Let’s suppose that this next image is how the map is first represented to the player:

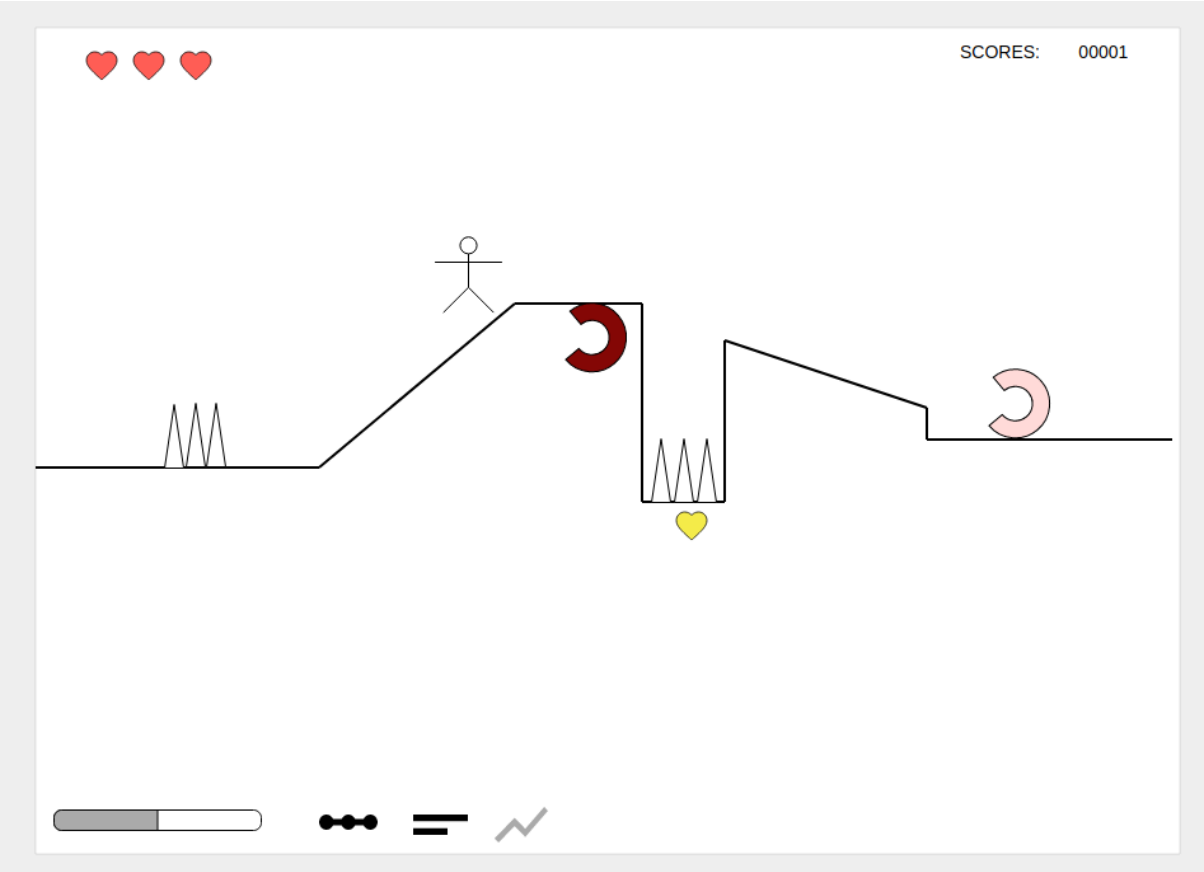


When the player hits the key for “inverting” the map on the invisible axis, the terrain is “flipped” on the line of the axis:

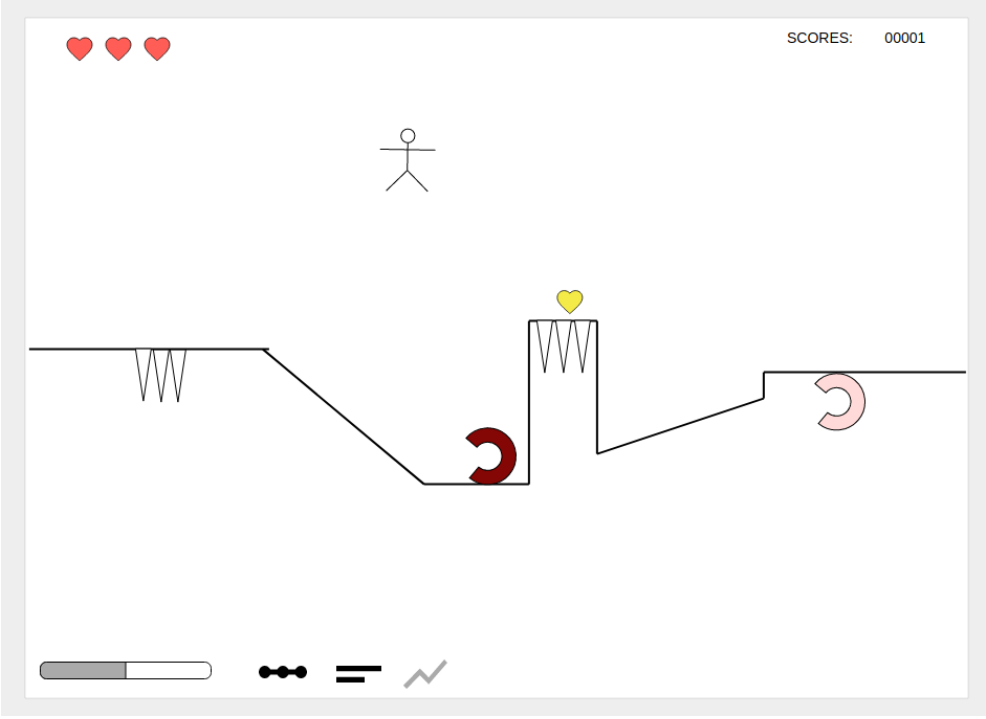


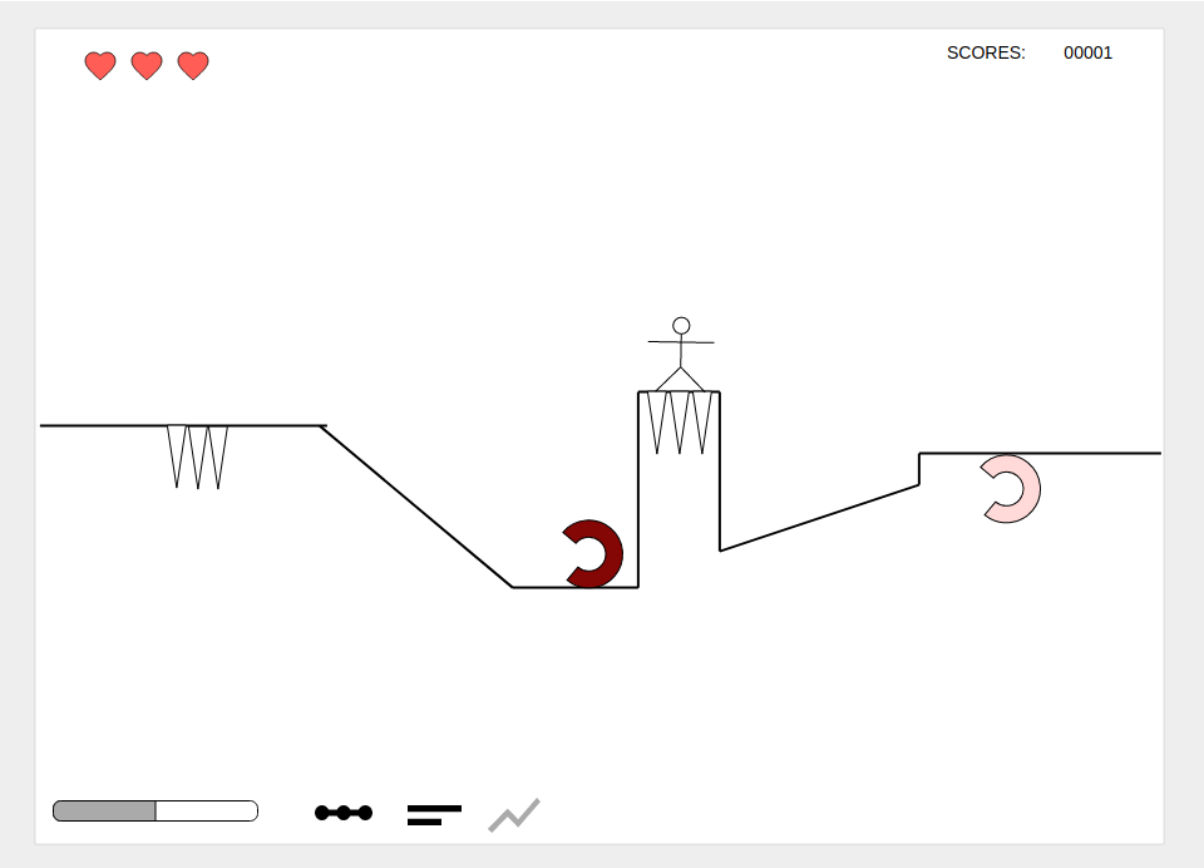
This way, the player can reach the “extra life” heart but will have to fight off an enemy.

Let’s see a second example. Suppose that the current player position is such as:



With this position, when the player “flips” the terrain, on the new terrain he is afloat in the air for a moment, but gravity immediately affects the player, and he starts falling down. This is a “challenge” that the player has to overcome for fun:





The “axis” that the map is flipped on would be presented with gray dashed lines ------- to the player, so that this mechanic would be easily understood by the player.

As the player progresses onto new levels, this axis would become invisible to the player, making it a new challenge to the player to experiment with the flips, and find out where the new axis is currently located in the current level.

Classes structure overview for modders

Notable classes from which modders are able to implement their own new features are the following:

SpecialItem, ISpecialItem

A special item's constructor has the following signature:
`public SpecialItem(Brush brush, Pen pen, Geometry area)`

`This class has one function only: void OnPlayerPickUp(GameModel model)`

`This function is to be called whenever the player encounters a pickupable specialitem ont he map. This function called in the Logic layer, using: void OnPlayerPickUpItem(SpecialItem item)`

Enemy, IEnemy

This class is responsible for representing the enemies the player encounters, and is able to kill. These enemies are able to shoot bullets, which can hurt the player or other enemies.

An enemy has health, and a bullet. This bullettype is the one that the enemy will continously shoot at the horizontal direction of the player.

Using `void EnemyShoot(double playerCX), the bullet is shot from the enemy into the direction of the player.`

Bullet, IBullet

This class is the basis for the bullets that are shot, both by players and enemies.

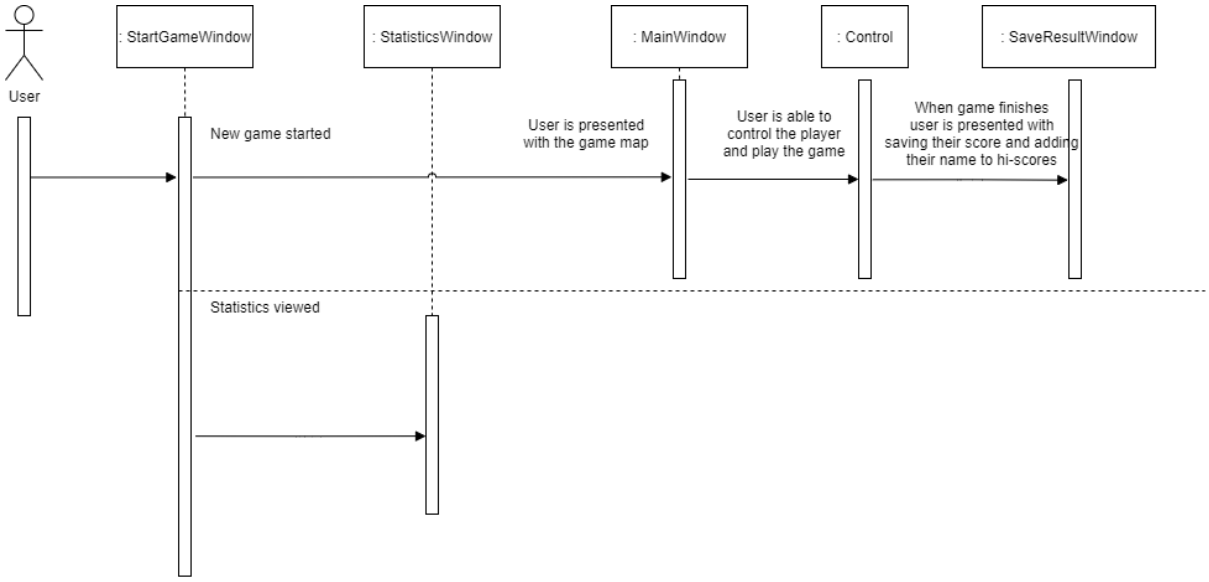
Every bullet has an `int damage, which determines the amount of health that is substracted from the enemy. However, when the bullet is shot at the player, the player loses one of their lives even by a single bullet.`

`The bullet constructor is as follows: Bullet(double cx, double cy, double dx, double dy, int damage)`

`Every bullet has a void Move() function, which increments the bullet's position by 1.`

Application overview, additional diagrams

Sequence diagram representation of a game cycle



When the user runs the application, the **StartGameWindow** is presented. Using this window, the user is able to view the **StatisticsWindow**, which presents all currently saved scores of currently saved games.

From the **StartGameWindow** the user is also able to navigate to the **MainWindow**, which is the main window for the gameplay. In this window, the gamemap is shown, that contains:

- Terrain and obstacles
- The player
- The enemies
- Any bullet that is created by a player or enemy

After this window has been loaded, the **Control** component takes over, which is responsible for interaction between the logic, model and the user. While the user is playing the game, **Control** is indeed in control, and will stay in control until the player reaches its goal or dies, as in runs out of lives to be spared.

When the current game is over, a dialog appears that informs the user of its inevitable death, and the user is able to accept their players' death by clicking the „OK" button.

Upon clicking OK, the user is presented with the **SaveResultsWindow** in which they are able to input their name, and save their score that is now attached with their given input name. This score is saved, and is open for viewing using the **StatisticsWindow**, which is reachable from the **StartGameWindow**.

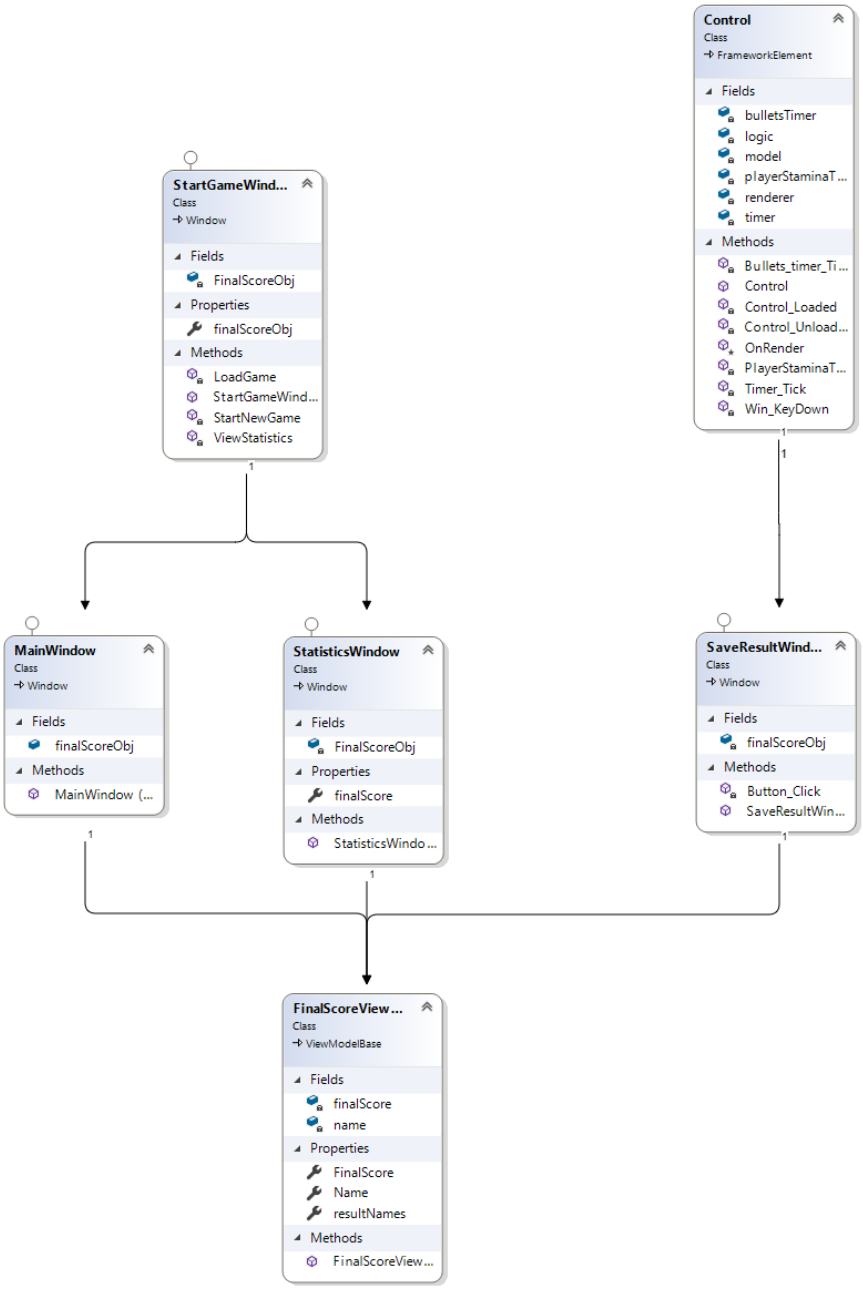
Component diagrams of classes

The application contains 5 projects:

- Program
- Renderer
- Logic
- Model
- Tests

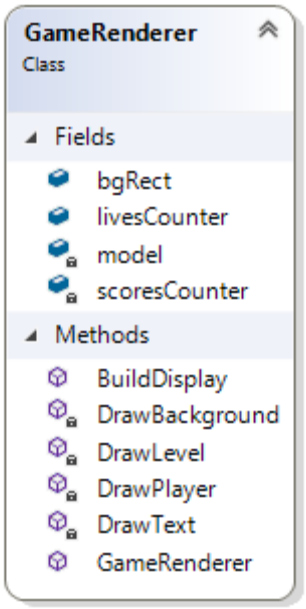
Program project

This project contains the components that are explained at „Sequence diagram representation of a game cycle„ section of the documentation.



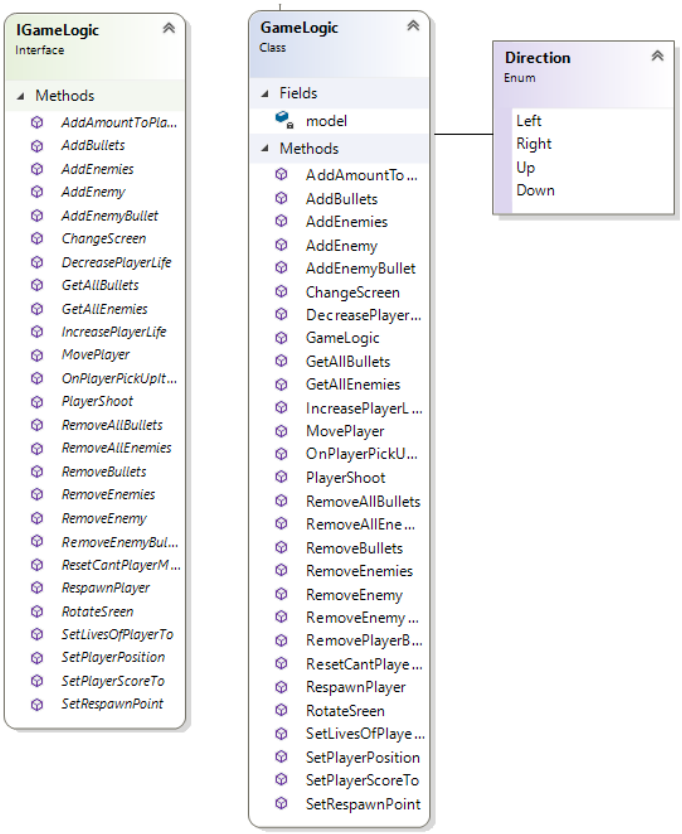
Renderer project

Contains one class, called the **GameRenderer**:

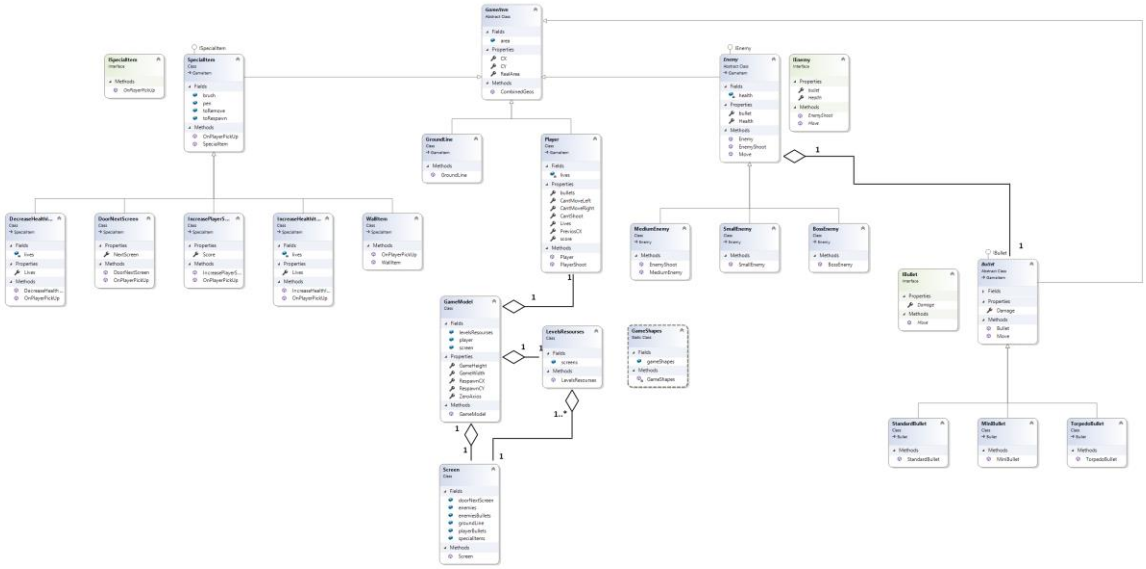


Logic project

Contains the main GameLogic which is a library of functions for the mechanics of the game.



Model project



Please zoom in, in order to view the diagram in its full glory, as image compression is turned *off* for this document. The model projects contains all the classes that are handled, and are acted upon by the other projects.

Tests project

The tests are purely for helping the developers and modders of the game to carry out and run automated tests on the application.