Emergent Architecture Design

Final Version

Occlusion Studios (CG1)

TU Delft

Emergent Architecture Design

Final Version

by

Occlusion Studios (CG1)

Teacher:
Dr. ir. Rafael Bidarra

Games Context Teaching Assistant:
Jurgen van Schagen

Software Engineering Teaching Assistant:
Martijn Gribnau

Studio Members:

Raoul Bruens - Producer and Communication	4571010
Tim Huisman - SFX/Audio Designer and Interaction Designer	4591305
Mariette Schonfeld - Lead Testing	4474147
Thijmen Langendam - Lead Designer, Lead Artist and Trailer Director	4592646
Maxim Liefaard - Lead Programmer	4607171

Expertise: BSc Computer Science
Project duration: April 24, 2018 – June 28, 2018

TU Delft

Contents

1

Introduction

This document contains an overview of the architecture of the software behind the game that the studio is going to develop. First, the design goals of the architecture are explained. Then Chapter 2 details an in-depth look of the system using various diagrams to clarify how the software functions.

1.1. Design Goals

Besides all the requirements specified in the *Game Design Document (Product Vision)* the studio also has some goals with respect to the quality of the code base. These goals are explained in the following subsections.

1.1.1. Reliability

The most important aspect of the code base is reliability. The context supervisor has told the studio multiple times that this aspect is the most important because the game should be able to run for entire days without restarting. The game software will run in a public area where a large amount of people are able to play the game without any monitoring required.

In order to meet these requirements the studio decided to make use of the *Linux* operating system combined with the *C++* language which are both well known for reliability, a high level of control and speed. The provided motion tracking library *IMOVE* that is used for the location based aspects of the game is already thoroughly tested and shown to be reliable when configured correctly.

To ensure reliability of the game, it is essential to do a lot of software testing. However, due to a small variation of *IMOVE* location prediction accuracy, the studio is unable to do very precise unit tests. Most of the important testing will thus be based on higher level tests, like integration testing and functional testing. Another important aspect will be extensive stress tests in a real environment where players will try to break the game.

1.1.2. Performance

Since the software will be running on a variety of hardware that is not guaranteed to be high-end, the game has to be as flexible as possible. Furthermore, the image recognition for the location based tracking already requires a lot of processing power, so the game code should be kept as lightweight as possible. Finally, performance must not degrade over time due to memory leaks since the application could be running for an entire day.

1.1.3. Scalability

Another requirement from the context supervisor was that the application must be able to run on any *IMOVE* compatible setup. This means that the playing field size can vary and that the projector and/or camera angle can change. These are all calibrated parameters within *IMOVE*, however the game should also be compatible with these changes in parameters. The

playing field must be scalable to ensure that the game can be played on different setups than those that are used for testing.

1.1.4. Component Independence

Since most of the game will completely depend on the player locations provided by the *IMOVE* framework, it is important to keep the game component separate from the motion tracking component. The studio aims to solve this by creating a single connection between the game and the framework, this allows the framework to be interchanged with a different framework or a debug simulation. Testing will be difficult since the studio will not have access to the testing setup with real motion tracking at all times. Thus, separating these components allows the studio to replace the framework with a mock or simulation of player locations and playing field size.

1.1.5. Code Quality

The studio immediately agreed on some important rules that help increase code quality:

- The master branch is locked at all times, no one pushes directly to the master branch.
- Each pull request must have approval from at least two reviewers before the code can be merged.
- All code in a pull request must be documented.
- Pull requests preferably includes testing code if possible and reasonable.
- At least one reviewer pulls the code and runs it on his local machine to check for obvious errors.
- Each pull requests is checked by continuous integration.
- The lead programmer continuously monitors the code base and informs the team about the status of the code base. If code quality drops, then an effort is made to improve the code before continuing on new features.
- All SCRUM documentation follows the templates that were developed for this. See appendix A.

2

Software Architecture

2.1. Testing

Testing the software integrated with the framework is a very important aspect of the development process. The studio will mainly focus on functional testing, instead of unit testing. This is, because the IMOVE framework will not always be as accurate in providing data, and the software should be robust enough to handle any fluctuations in data. In other words, it should be functionally sound.

The studio looked into a variety of different testing frameworks. After some research the studio made a selection of some of the most recommended testing frameworks for C++. The candidates were: *Google Test*, *Catch* and *Boost.Test*

Google Test was chosen as a testing framework because it is the only testing framework that supports mocking out-of-the-box and the studio planned to make use of mocking in tests. Other testing frameworks do support mocking but some require additional dependencies or configurations that will only increase the complexity of the build. Unfortunately, the use of mocking was rather limited because all studio members had a lot of difficulty setting up mocking objects correctly. To prevent further loss of time the studio decided to make less use of the mocking features of *Google Test*.

To include *Google Test* into the project, a separate *testing* folder was made that generates it's own executable using *CMake*. This executable has testing code bundled with all game logic code. In this way, testing code is not included in the game executable that is used in production.

2.2. Subsystem Decomposition

This section shows how the system was originally decomposed into subsystems as shown in figure 2.1. For a more detailed look at how the final implementation is structured refer to section 2.3.

- **IMOVE** - the location tracking library that provides coordinates of players that stand in the playing area, this component takes video footage from a camera as input.
  Current Implementation: IMOVE is in a separate folder and only accessed via a single bridging class: *GameScene*.
- **IMOVE Simulation** - a testing library that is able to replace *iMove* and give testing coordinates, an idea is to allow people from the studio to test the system by providing player locations in real time using mouse pointers.
  Current Implementation: When the game is run via the game logic main the user is able to input player location using mouse, arrow keys and wasd keys.
- **Game Logic** - this subsystem handles all the game logic based on the previous state of the game and the new locations of the players. If needed this subsystem will be decomposed further into subsystems that manage the data and locations of game objects in

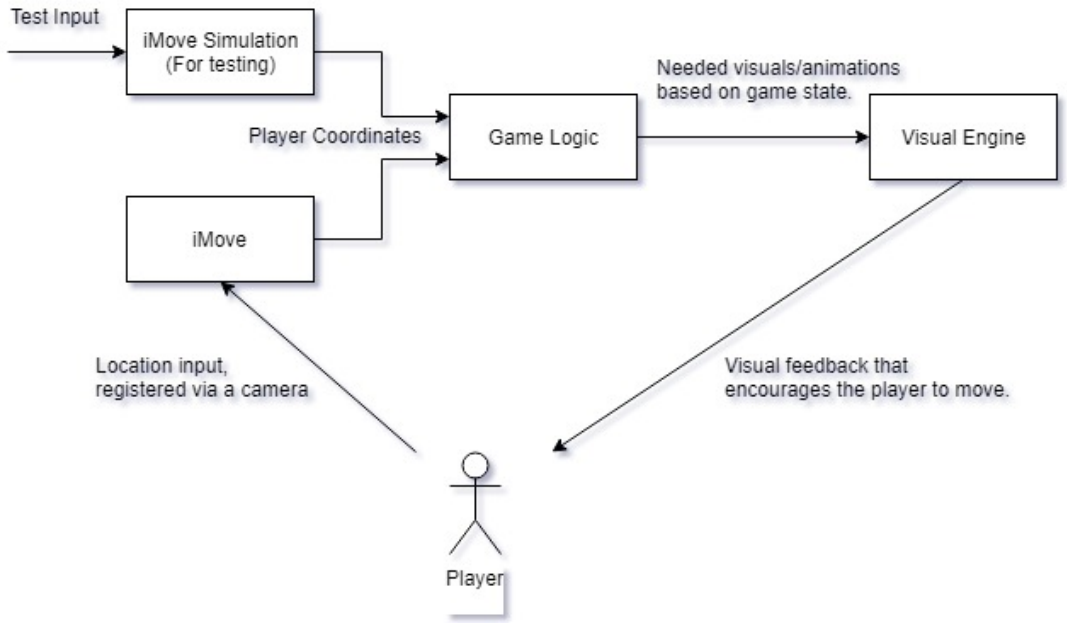


Figure 2.1: Initial concept of the subsystem decomposition.

the game scene.
Current Implementation: Game Logic is completely separate from visuals and input. Input is delivered to game logic via an *InputData* object. A *RenderWrapper* object is passed to game entities, these game entities call methods on the *RenderWrapper* to instruct it that a game entity with certain parameters should be rendered.

- **Visual Engine** - this subsystem displays visualizations and animations of game objects. The new rendered frame will be sent to the projector which then projects this frame over the playing field.
  Current Implementation: the visual engine is *SFML*, this introduces a lot of rendering and GUI code. At first this was spread throughout game logic but the team made an effort to move all of this code to the *RenderWrapper*.

2.3. IMOVE Integration Implementation

This section shows how the integration between the game logic and IMOVE was eventually implemented. The studio ensured that the game logic can function separately when provided with an SFML rendering target and input data in the correct format. Explanation of the IMOVE integration in figure 2.2:

- **GameScene** - GameScene is the class in which *IMOVE* handles its projected scene. In this class, all coordinates of the detected players are propagated as locations in the scene. GameScene has a UserManager, which keeps track of all these locations.
- **UserManager** - UserManager has the responsibility of keeping track of all players and their corresponding locations. It updates the core game logic by generating *InputData* based on the content of its attributes. This class also does the error correction of coordinates that are received from *IMOVE*.
- **RenderWrapper** - IMOVE creates its own SFML window, this is wrapped in the RenderWrapper such that it can be used in the game logic.

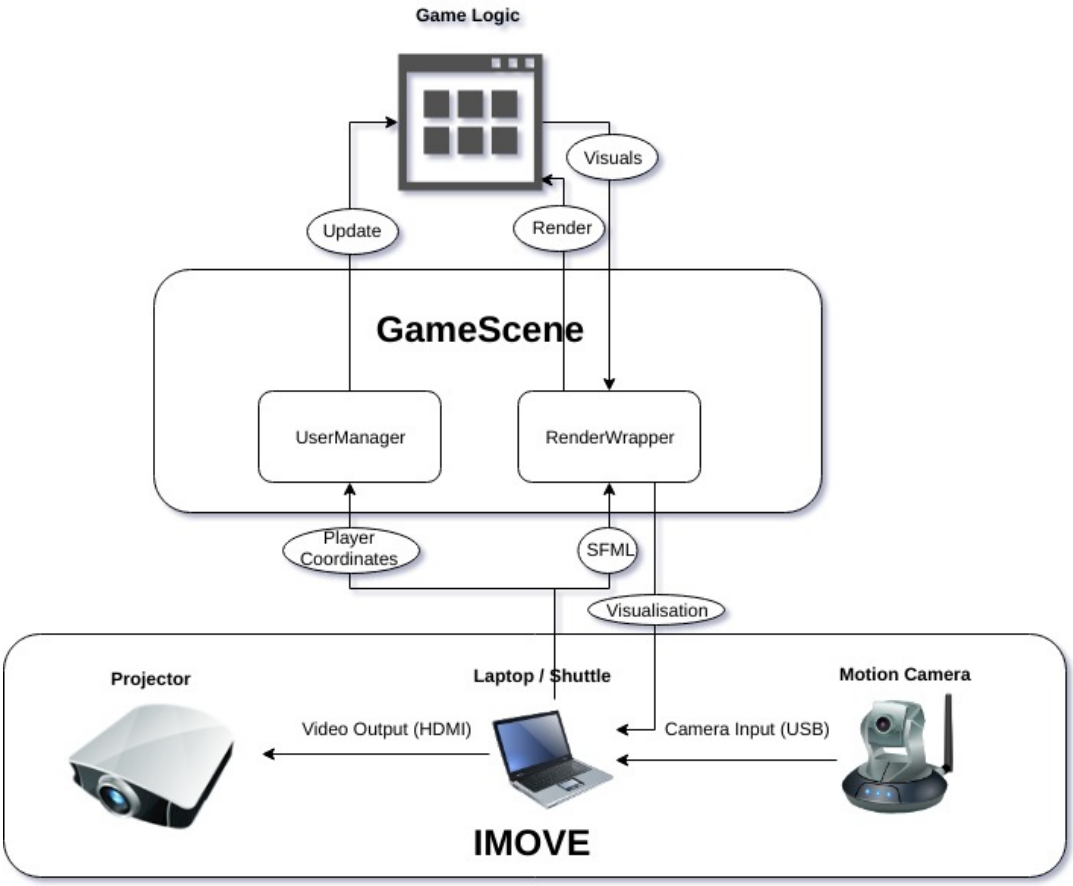


Figure 2.2: Diagram of the implementation of IMOVE integration with the Game Logic.

2.4. Game Entity Hierarchy

This section explains the hierarchy of all the entities in the game. Each entity has an *update* and a *render* method. The *update* method is used to update game logic. After all the game logic was updated the new state of the entities is rendered by calling the *render* method. Figure 2.3 shows how the method calls of *update* and *render* are propagated along all the objects. This way of propagating method calls is inspired by the *Unity3D* game engine.
Explanation of the hierarchy in figure 2.3:

- **GameManager** is the object that is central to updating and rendering the game. Depending on its active *BasicState* its behaviour changes. It propagates *update* or *render* calls to the *TurtleManager*, *PlayerManager*, *ObstacleManager*, and the list of *ObjectiveArea*s.
- **TurtleManager** is the object that maintains all turtles, creates them and removes them. It propagates *update* or *render* calls to all the *Turtle*s that is manages.
- **Turtle** is the *GameEntity* that represents a turtle its behaviour changes based on its active *TurtleState*.
- **PlayerManager** is the object that maintains all players, creates them, removes them and filters for inactive players. It propagates *update* or *render* calls to all the *Player*s that is manages.
- **Player** is the *GameEntity* that represents a player.
- **ObstacleManager** is the object that maintains all the obstacles. It removes and fades obstacles that were hit and it adds obstacles at correct location. It propagates *update* or *render* calls to all the *Obstacle*s that is manages.
- **Obstacle** is the *GameEntity* that represents an obstacle.
- **ObjectiveArea** is the *GameEntity* that represents an objectiveArea (a sea).

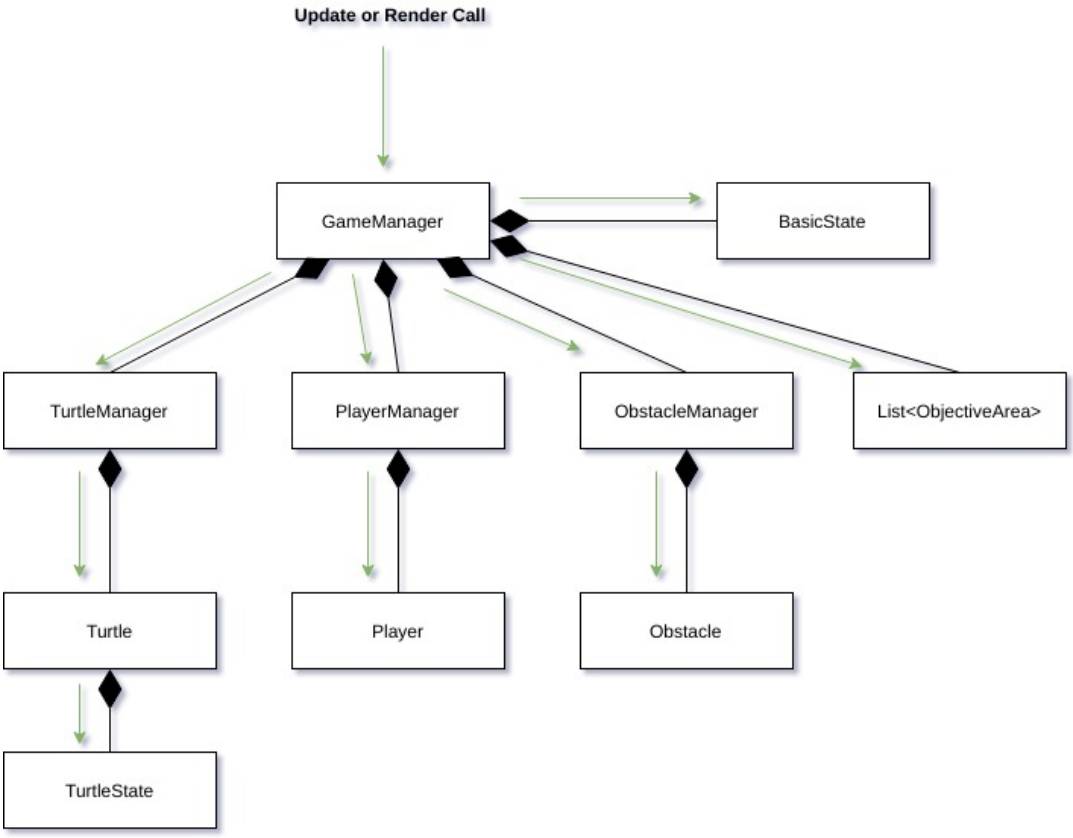


Figure 2.3: Diagram of the game entity hierarchy. The green lines show how *update* and *render* calls are propagated along the objects.

2.5. State Diagrams

2.5.1. Conceptual Diagrams

This section discusses some of the state diagrams that were designed in order to clarify some of the gameplay mechanics. Note that the studio is still in the process of completely implementing these state machines since the main focus is to build incremental prototypes in short iterations in order to achieve playable spikes. First Figure 2.4 shows the state diagram of the main gameplay loop then Figure 2.5 shows the states of a turtle entity.
Explanation of the main game states in figure 2.4:

- **Interactive Environment** is the initial state, in this state the game has not started yet but the environment does respond to the players. For example, footsteps in the sand or turtles that walk away. The mother turtle will ask players to save her turtles.
- **Game In Progress** is the state where the game is running and players have to herd the turtles towards the water.
- **Natural Reset** in the case that players leave the game, the remaining turtles will naturally walk back to their mother and the game automatically resets to an the *Interactive Environment* state.
- **Game Finished** is the state that occurs when players save all the turtles. The highscores will be displayed and the game fades back to the *Interactive Environment* state after some time.

Explanation of the turtle states in figure 2.5:

- **At Spawn** is the state in which the turtle spawns. The turtle is at the spawn location (the mother turtle) and will not move until the game starts.
- **Returning To Spawn (Natural** is the state that represents the turtles walking back to the spawn to naturally reset the game.

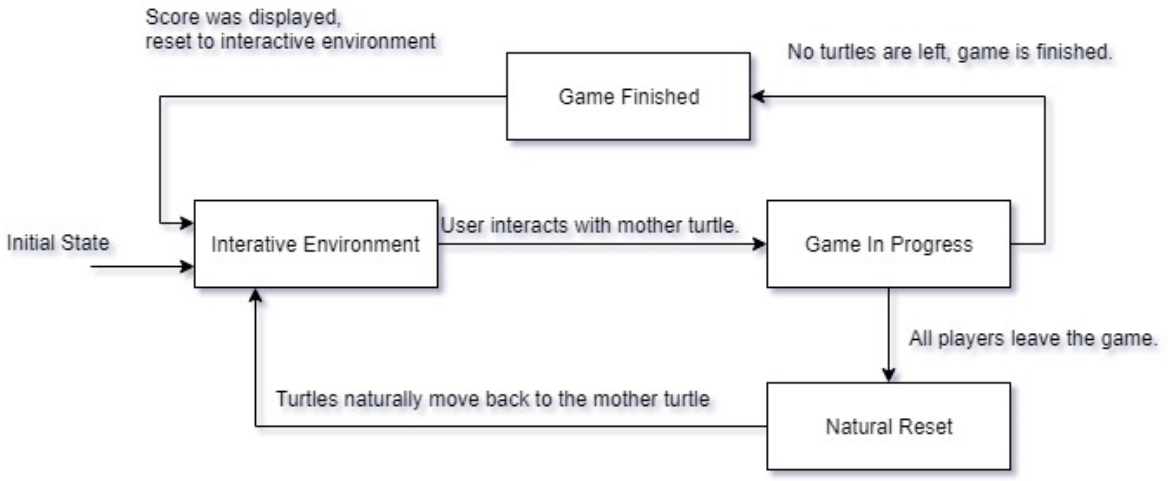


Figure 2.4: State diagram of the main game state.

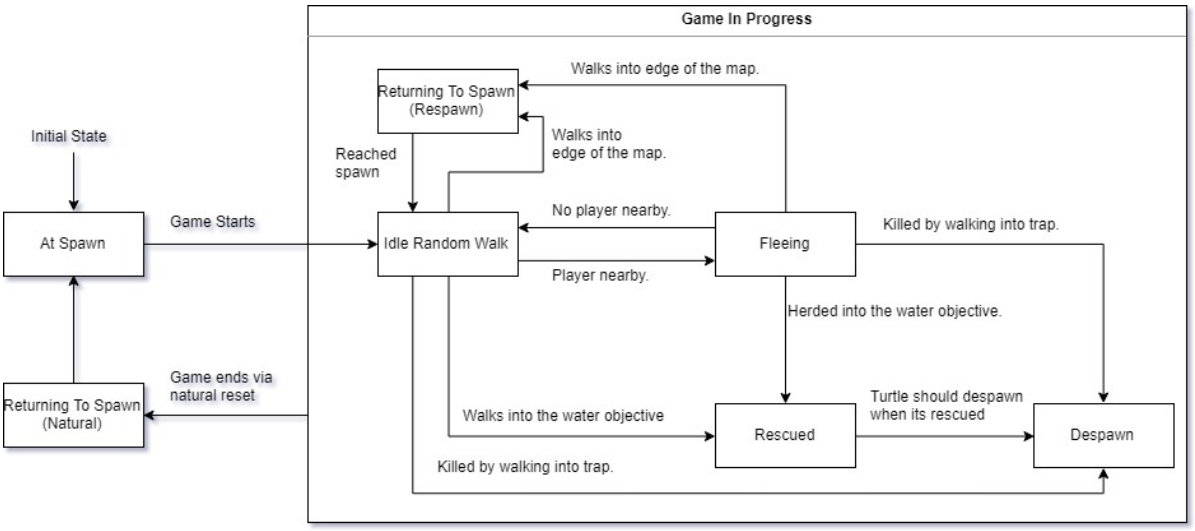


Figure 2.5: State diagram of a turtle.

- **Idle Random Walk**, in this state the turtle will randomly walk around the map, players should watch out that the turtles do not kill themselves by walking into traps.
- **Fleeing**, when a player approaches the turtle will flee from the player. By approaching the turtle from different sides the player can influence the direction that the turtle walks in.
- **Returning To Spawn (Respawn)**, when a turtle walks to the edge of the map it will quickly flee back to the spawn location to reset itself. In this state it is not affected by traps or players.
- **Rescued**, if the turtle walks into water it has reached its objective it will automatically transition to the despawn state. The score is incremented and the turtle entity is destroyed.

2.5.2. Current Implementation of States

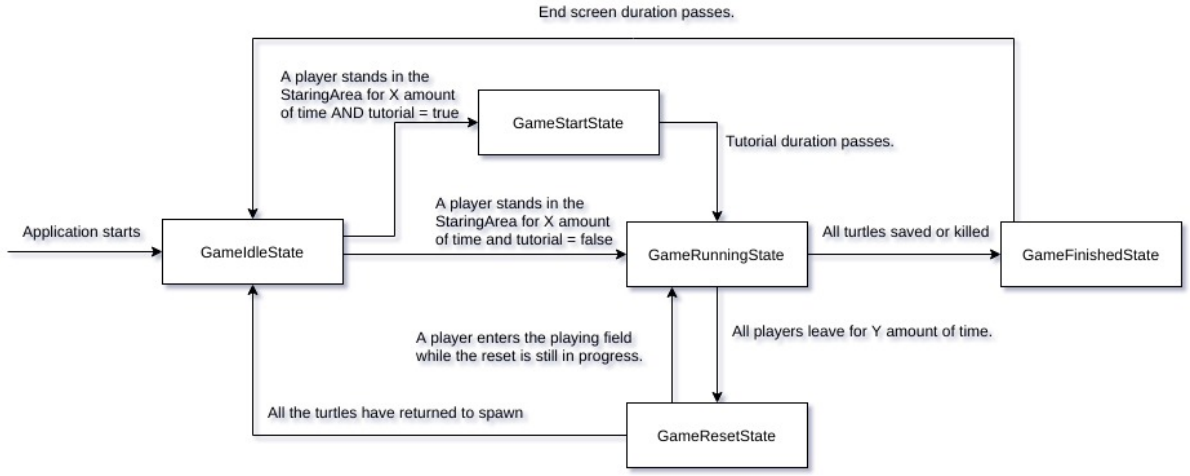


Figure 2.6: Implementation of the main game states at the end of sprint 4.

Explanation of the main game states in figure 2.6:

- **GameIdleState** is the initial state, in this state the game has not started yet. Players can interact with the turtles but there are no traps and there is no scoring system.
- **GameStartState** is the state in which the tutorial is displayed, this takes a configurable amount of seconds and it automatically transitions to the GameRunningState.
- **GameRunningState** is the state where the game is running and players have to herd the turtles towards the water. Traps are spawned and the game ends once all turtles are either killed or saved.
- **GameResetState** is the state where the game is resetting in the case that all players leave a game in progress. Turtles will start to move back to the spawn area. Once all turtles have returned the game is reset and it transitions back to the GameIdleState. The reset is cancelled when someone enters the field during the reset, then the game that was in progress before the reset can still be finished.
- **GameFinishedState** is the state that occurs when players save or kill all the turtles. The highscores will be displayed. The game fades back to the GameIdleState after some time.

Note: The transition between the GameRunningState and the GameFinishedState is delayed. Once all turtles have been saved, a screen transition starts. Once the transition is done, the game enters the GameFinishedState.

Explanation of the main game states in figure 2.7:

- **TurtleSpawnState** the turtle is spawned as an egg at the center and does not move. If a player collides with the egg, the egg hatches.
- **TurtleWalkState** the turtle enters this state upon interaction with a player. This state encompasses the fleeing of the turtle.
- **TurtleIdleState** the turtle enters this state if it is not interacted with and if it is at the spawn area. In this state it will roam around randomly but it will never leave the spawn area by itself.
- **TurtleRoamingState** the turtle enters this state if it is not interacted with and if it is outside the spawn area. In this state it will roam around randomly. It can never walk into the objective area by itself but it can kill itself by walking into a trap.
- **TurtleReturningState** this state is used during the GameResetState to return the turtles to spawn. Turtles remain in this state until they are back at spawn.
- **TurtleObjectiveState** the turtle enters this state upon reaching an objective area. The turtle then picks the closest corner of the map and moves towards it. Once it has reached this corner, it is saved and it disappears.
- **Killing Turtles** turtles can die during any state when colliding with an obstacle.

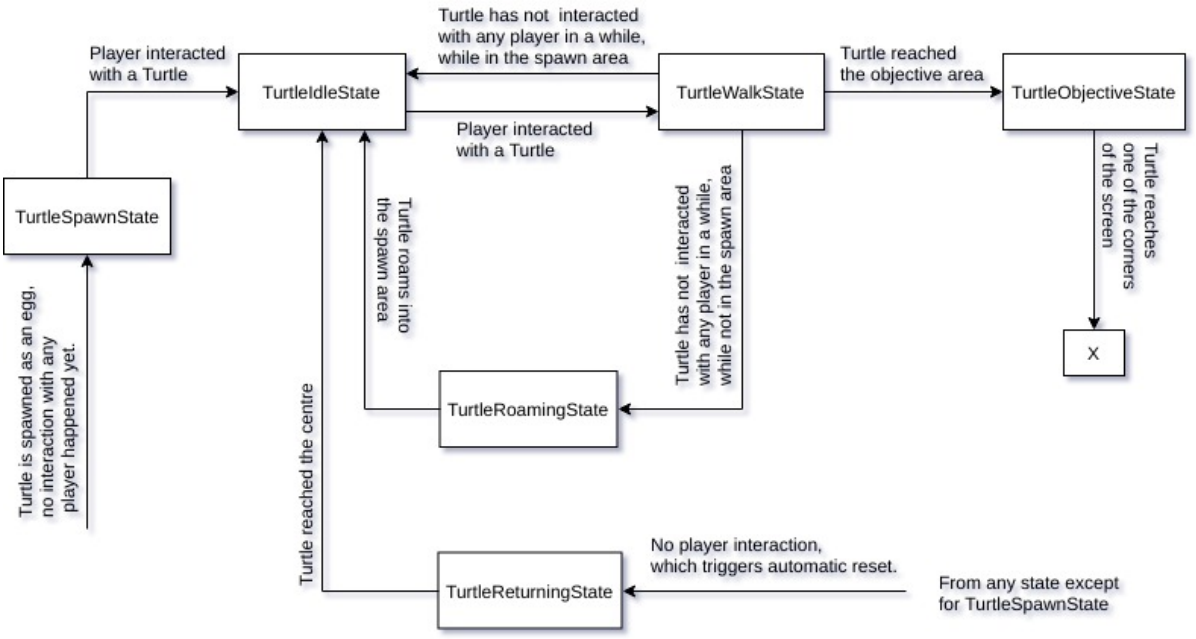


Figure 2.7: Implementation of the main turtle states at the end of sprint 4.

2.6. Hardware/Software Mapping

The setup consists out of a computer that uses camera input for motion tracking people on a ground. The projector projects the playing field onto the ground. Projector and camera are pointed to the same direction because the camera must be able to see the people that are in the playing field.

The software uses camera input for object tracking and produces video output to give feedback to people playing the game. The IMOVE framework processes the video input to determine where persons are in the playing field. The Game code runs on a separate thread within IMOVE. IMOVE propagates the people that it detected to the game logic, which instantiates player objects for each person.

After updating the game logic, IMOVE requests a render from the game which is then output in an SFML window such that it can be displayed on the projection.

See figure 2.8 for a visualization of the mapping.

2.7. Concurrency

Concurrency is handled by the *IMOVE* framework. The framework automatically starts seperate threads for image processing jobs. Thus this is not a part of the game implementation. *IMOVE* offers a dedicated thread for updating the window and the logic behind the game. The update and render methods are called from this thread.

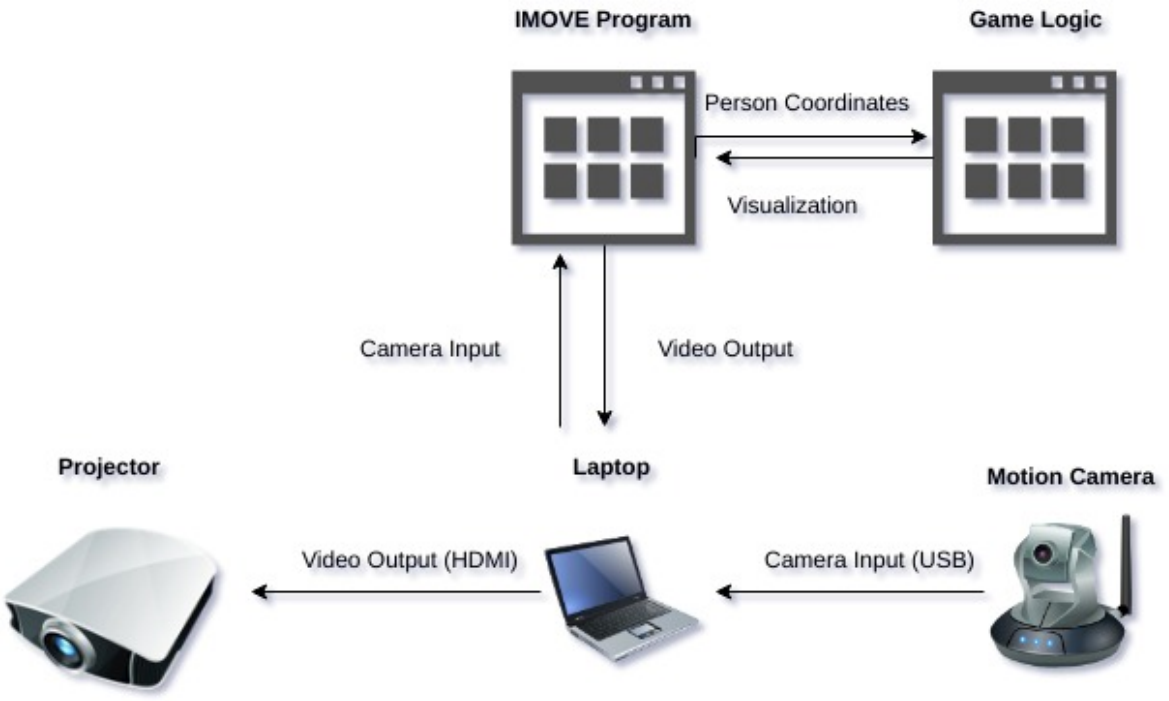


Figure 2.8: Diagram of the hardware and software mapping in the current state of the system.

2.8. Project Structure

The codebase is currently structured as follows:

- .gitlab - This folder contains templates used for GitLab issues and merge requests.
- changelogs - This folder contains a script to generate a retrospective.
- deliverables - This folder contains all PDFs: meeting documentation, retrospectives, coverage reports, etc.
- game - This folder contains all code related to the game, excluding the object tracking feature.
  - assets - This folder contains sprites and fonts used in the game.
  - src - This folder contains all the code of the game.
    - control - This package contains controllers for managing the different game states and managers for collections of player and turtle entities.
    - gameEntities - This package contains code for all game entities: player and turtle. Turtle has a series of states as described by the state diagrams.
    - storage - This package contains classes that only serve as storage objects for data. These objects are passed around to support propagation of more complex datasets like InputData.
    - utility - This class contains a series of utility classes like: classes that contain mathematical functionality and a wrapper for SFML components.
  - testing - This folder contains all testing code.
    - googletest - The GTest library.
    - test-src - Contains all the testing folders for specific test types.
      - unit-tests - This folder contains the same folder structure as the src folder with unit testing files for each code file in src.

  - description-based-tests - This contains tests for visual aspects that have to be checked manually.
  - feature-tests - This folder contains broader tests that test integration between multiple components by following the user story scenarios that were defined in earlier documentation. These tests are used to verify acceptance criteria.

- IMOVE - This folder contains the original IMOVE framework with some adaptations to facilitate a bridge between IMOVE and the game.
  - imove_scene - This folder contains code that supports creating an SFML game scene, the bridge to the game logic is made here and some changes were made to create a better link.
  - All other unmodified IMOVE code folders.
  - Build/run scripts for IMOVE, these were modified to include code from the Game folder. These scripts are used when running the Game for an IMOVE setup.

- Various build/run scripts, tool runners and continuous integration configurations ...

2.9. Deployment Guide

This section contains guide for setting up the project and using the different runnables and build scripts included in the repository.

2.9.1. Dependency Setup

The project was based upon Ubuntu 16:04. If you choose to use a later iteration of Ubuntu please note that you could get more updated versions of our dependencies. This can cause unwanted behaviour like compilation issue or different coverage/static analysis results. The studio does not take this into account and we assume that you use Ubuntu 16:04 because ensuring compatability with multiple operating systems is beyond the scope of this project.

Once you have ensured that you use the correct operating system you can automatically install dependencies by running the *installation_script.sh* script. This took our computers 20 to 30 minutes.

Note that code coverage and Cppcheck have more specific dependencies, refer to their subsections for more details. Clion is needed for running description-based tests.

2.9.2. Running the game standalone

The game can be run in standalone mode by running the *buildAndRunGame.sh* script.

To change aspects of the game please refer to the Variables.cpp class and the section that documents all these variables.

Note: We plan on using a dedicated config file on release instead of this setup.

2.9.3. Running the game with IMOVE

The game can be run with IMOVE by running the *buildAndRunIMOVE.sh* script.

To change aspects of the game please refer to the Variables.cpp class and the section that documents all these variables.

To change aspects of IMOVE please refer to the official IMOVE documentation. The config files are located in IMOVE/config. The currently used config can be found inside the *buildAndRunIMOVE.sh* script. An immediate crash is most often caused by an incorrect *Camera_device* setting.

2.9.4. Running automated tests

Automated tests can be run using the *buildAndRunTests.sh* script. We use GoogleTest as a testing framework.

2.9.5. Running description-based tests

Based on feedback we added description-based tests of our rendering aspects in the game. These are located in *testing/test-src/description-based-tests/*. There is a markdown file that describes all test cases. To run the tests look into the testMain.cpp class for instructions and run testMain.cpp using the play button in CLion.

2.9.6. Generating code coverage

Code coverage is generated using *lcov*. Please refer to the contents of *generateCodeCoverage.sh* for instructions on how to install this dependency.

Once the dependencies are installed you can run the script, it will only generate coverage if all tests pass and it will automatically open the coverage report in chrome (it assumes you have chrome installed).

The generated report will be located in *game/CodeCoverageReport*

2.9.7. Running CppCheck

CppCheck is the tool that we use for static analysis. Please refer to the contents of *static-analysis/runCppCheck.sh* for instructions on how to install CppCheck. The tool runs fairly quickly and prints all warnings in the console.

False positives or useless warnings were suppressed from the *static-analysis/CppCheckSupression.txt* files. If you need motivation for any of the suppression the studio can always be contacted.

2.9.8. Verifying userstory acceptance criteria

At the start of the project the studio created a lot of user stories, each with standard scenarios and edge case scenarios. In the feature-tests folder these user stories are verified by either reasoning, description-based tests or integration testing. Markdown documentation in those files shows which user story is verified in what way.

2.10. Configuration

2.10.1. SFML Window

- *GAME_NAME* - The name that is displayed at the top of the window.
- *PIXEL_WIDTH* - The SFML window pixel width, this should match the projector resolution.
- *PIXEL_HEIGHT* - The SFML window pixel height, this should match the projector resolution.
- *FRAMERATE_LIMIT* - The maximum number of frames per second, framerate is capped when it manages to produce more frames than the limit;
- *PIXELS_PER_METER* - The amount of pixels per real life meter, this is also used in the IMOVE configuration.
- *IMOVE_ACTIVE* - A boolean to denote whether the game is launched in IMOVE mode, this is automatically set.

2.10.2. Debug

- *DEBUG_UPDATE_RATE* - The amount of seconds after which the debug visual should be updated, this is needed to make the statistics readable.
- *DEBUG_MODE* - A boolean to denote whether the game is launched in debug mode, in this mode fps indicators are displayed.
- *DEBUG_SLOWMO_MODE* - A boolean to denote whether the slow-motion debug mode is active. This reduces (or increases) the game speed by the value of DEBUG_SLOWMO_FACTOR. It is useful when debugging specific interactions that happen quickly.
- *DEBUG_SLOWMO_FACTOR* - The value with which the time that passes during each frame is multiplied. Choose a value in the range <0.0; 1.0> for slow-motion and choose a value larger than 1.0 for a speedup.

2.10.3. Gameplay

- *TOTAL_TURTLES* - The total amount of turtles that will have to be saved.
- *MAX_VISIBLE_TURTLES* - The maximum amount of turtles that can be in the playing field at one point
- *NUMBER_OF_PLAYERS* - The number of players that is spawned by default when running in standalone mode. Becomes redundant when using the interactive environment loop.
- *DEAD_TURTLES* - The amount of turtles that have died in the current game loop.
- *ID_COUNTER* - The counter of the last global id that was assigned to the last entity that requested an id. This is used to make sure that the id's of entities are unique.
- *SCORE_WIN_THRESHOLD* - How high the the score should be for the participants to receive the *win screen* instead of the *lose screen*.

2.10.4. GameLoop

- *NO_ACTIVITY_TIMEOUT* - The amount of seconds before the GameManager goes into the GameResetState when all players leave the field during the GameRunningState.
- *START_SCREEN_TIME* - Amount of seconds that the GameStartState is active before transitioning to the GameRunningState. This is the duration of the tutorial screen.
- *END_SCREEN_TIME* - Amount of seconds that the GameFinishState is active before fading back to GameIdleState.

- *PERCENTAGE_OF_HATCHED_TURTLES* - Percentage of turtles that should be hatched automatically during the GameIdleState. This makes the interactive environment more appealing to bystanders.
- *SCREEN_FADE_TIME* - The amount of time (seconds) that the fade takes when the game is finished.
- *SHOW_SPONSOR* - Enables the visibility of the sponsor logo in the end screen.

2.10.5. Player

- *PLAYER_FORCE_RADIUS* - The radius of the force field of the player. (meters)
- *PLAYER_WALL_RADIUS* - The radius of the wall of the player which the turtles cannot cross. (meters)
- *PLAYER_TIMEOUT* - The amount of seconds it takes before a player that received no new input in those seconds is removed. This is used in IMOVE to remove glitched players that are not coupled to a real person.

2.10.6. Turtle

- *TURTLE_BORDER_OFFSET* - The offset in meters that the turtles cannot walk into. This is to prevent that turtles can move to the borders of the playing field.
- *EGG_SPAWN_TIME* - The time (seconds) that needs to pass before an egg can hatch. In this time the player cannot interact with the turtle.
- *TURTLE_WALK_SPEED* - The walking speed of the turtle in meters/second while it is in the TurtleWalkState.
- *TURTLE_FORCE_SPEED* - The walking speed of the turtle in meters/second while it is colliding with the inner circle of the player. This should be a large value to prevent a turtle from being under the persons feet.
- *TURTLE_IDLE_SPEED* - The walking speed of the turtle in meters/second while it is in the TurtleIdleState.
- *TURTLE_ROAMING_SPEED* - The walking speed of the turtle in meters/second while it is roaming randomly in the TurtleRoamingState.
- *TURTLE_RESET_SPEED* - The walking speed of the turtle in meters/second while it is resetting, this happens during the GameResetState. *TURTLE_OBJECTIVE_SPEED* - The walking speed of the turtle in meters/second while it is in the ObjectiveArea and leaving the screen.
- *TURTLE_RADIUS* - The radius of the hitbox of the turtle. (meters)
- *WIGGLE_DURATION_IDLE* - The amount of time (seconds) in between direction changes when a turtle is wiggling in the TurtleIdleState.
- *WIGGLE_OFFSET_IDLE* - The sharpness of the turn that a turtle makes when wiggling in the TurtleIdleState, a larger value results in a sharper turn.
- *WIGGLE_DURATION_OBJECTIVE* - The amount of time (seconds) in between direction changes when a turtle is wiggling in the TurtleObjectiveState.
- *WIGGLE_OFFSET_IDLE* - The sharpness of the turn that a turtle makes when wiggling in the TurtleObjectiveState, a larger value results in a sharper turn.
- *WIGGLE_DURATION_RETURNING* - The amount of time (seconds) in between direction changes when a turtle is wiggling in the TurtleReturningState.

- *WIGGLE_OFFSET_RETURNING* - The sharpness of the turn that a turtle makes when wiggling in the TurtleReturningState, a larger value results in a sharper turn.
- *MOVE_DURATION_ROAMING* - The amount of time (seconds) the turtle moves after selecting a goal location while in the TurtleRoamingState.
- *WIGGLE_DURATION_ROAMING* - The amount of time (seconds) in between direction changes when a turtle is wiggling in the TurtleRoamingState.
- *WIGGLE_OFFSET_ROAMING* - The sharpness of the turn that a turtle makes when wiggling in the TurtleRoamingState, a larger value results in a sharper turn.
- *MOVE_DURATION_WALK* - The amount of time (seconds) the turtle moves after an interaction with a player while in the TurtleWalkState.
- *WIGGLE_DURATION_WALK* - The amount of time (seconds) in between direction changes when a turtle is wiggling in the TurtleWalkState.
- *WIGGLE_OFFSET_WALK* - The sharpness of the turn that a turtle makes when wiggling in the TurtleWalkState, a larger value results in a sharper turn.
- *INTERACTION_ROTATE_TIME* - The time it takes for a turtle to rotate when being interacted with, for instance when it collides with a player. This should be a small value to make the game feel responsive.
- *ROAMING_BUFFER* - The distance the turtles will roam before changing direction.

2.10.7. Objective

- *OBJECTIVE_RADIUS* - The radius of the objective - the place where the turtles meet the sea. (meters)
- *VISIBILITY_MARGIN* - The amount of pixels the turtle has to move away from the screen to make their sprite completely vanish from the game screen this is used to determine how fast to fade turtles that leave the screen.
- *DESPAWN_MARGIN* - The amount of pixels the turtle has to move away from the screen in order to be despawned.
- *ROAMING_NOGO_RADIUS* - The radius from the center of the four objective areas, in which the turtles will return back to the center of the playing field to prevent them from going into the objective radius. (meters)

2.10.8. Starting Area

- *START_GAME_TIME* - The amount of seconds needed for a player to stand in the StartingArea in order to start the game.
- *START_AREA_WIDTH* - The width in meters for the StartingArea.
- *START_AREA_HEIGHT* - The height in meters for the StartingArea.
- *START_AREA_VISIBILITY_TIME* - The amount of seconds needed for a player to be in the playing field for the StartingArea to become activated and visible.

2.10.9. Obstacle

- *TOTAL_OBSTACLES* - The total amount of obstacles that will be placed in the game.
- *OBSTACLE_RADIUS* - The size of an individual obstacle in meters.
- *OBSTACLE_SPACING* - The minimum distance between two obstacles.
- *OBSTACLE_PLACING* - The size of the ellipse on which obstacles can be placed. By increasing this the obstacles will be placed closer to the edges of the screen.

- *ELIPSE_DENSITY* - The amount of possible obstacle points calculated on the ellipse.
- *OBSTACLE_FADE_SPEED* - The amount of seconds it takes for an obstacle to fade in or out.

2.10.10. Trail

- *TRAIL_DURATION* - The amount of seconds that a TrailPoint stays on the scene before fading.
- *TRAIL_FREQUENCY* - The interval of the TrailPoints, in points per second.
- *TRAIL_FADE_SPEED* - The speed that a TrailPoint fades. This decrement happens every frame and starts at 1 until it has reached 0.

2.10.11. GUI

- *GUI_SCORE_GAME* - The font size of the score display during gameplay.
- *GUI_SCORE_FINISH* - The font size of the score display when the game has finished.

2.10.12. Animation

- *TURTLE_ANIMATION_SPEED* - The speed of the turtle default movement animation, each integer increment multiplies the base speed of the animation.
- *OBSTACLE_ANIMATION_SPEED* - The speed of the obstacle default movement animation, each integer increment multiplies the base speed of the animation.
- *DEATH_ANIMATION_SPEED* - The speed of the turtle death animation, each integer increment multiplies the base speed of the animation.
- *DEATH_ANIMATION_SIZE* - The size of the turtle death animation.
- *SCORE_ANIMATION_SPEED* - The speed of the turtle default movement animation, each integer increment multiplies the base speed of the animation.
- *SCORE_ANIMATION_SIZE* - The size of the score animation.

2.10.13. Debug Variables

Note that these variables are not configurable and are changed at runtime to increase the accessibility of certain values.

- *DEBUG_PLAYER_COUNT* - Debug variable used to display the number of players in the active PlayerManager, only used when DEBUG_ACTIVE is true.
- *DEBUG_TURTLE_COUNT* - Debug variable used to display the number of turtles in the active TurtleManager, only used when DEBUG_ACTIVE is true.
- *DEBUG_GAME_STATE* - Debug variable used to display the current state of the active GameManager, only used when DEBUG_ACTIVE is true.
- *DEBUG_TURTLES_LEFT_TO_SPAWN* - Debug variable used to display the number of turtles that are left to spawn in the active TurtleManager, only used when DEBUG_ACTIVE is true.

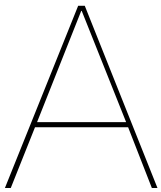
3

Glossary

This chapter contains definitions of all the terms and abbreviations used in this report. Feel free to contact the authors if anything remains unclear after consulting this chapter.

IMOVE	A library that provides location tracking of people based that walk around in a designated area using a camera.
SFML	A game engine for C++ which was used as a basis for the game.

Appendices



SCRUM Templates

This chapter shows some of the templates that were used to improve the SCRUM methodology of the studio. Templates were used for merge requests, user stories, tasks and bugs. The templates helped to make sure that studio members did not forget any documentation steps and it enabled to studio to have consistency for documentation.

A.1. Merge Request Template

```
<!-- You can erase any parts of this template not applicable to your Pull Request. -->
<!--- Provide a general summary of your feature or change in the Title above -->

## Description
<!--- Describe your changes or additions in detail -->

## Related Issue (link to bug issue)
<!--- If fixing a major bug or parts of the code, there should be an issue
describing it with steps to reproduce. -->
<!--- Please link to the issue here: -->

## Motivation and Context
<!--- Why is this change required? What problem does it solve? -->
<!--- If it addresses an open issue, just link to the issue here. -->

## Checklist

* [ ] Have you updated the issues on the board?
* [ ] Have you assigned assignee, labels and milestone to this PR?
* [ ] Have you updated your time spent on the task / bug?
* [ ] Have you written new tests for your core changes according to the definition of done?
* [ ] Have you documented your code according to the guidelines in the FAQ?
* [ ] Have you created new issues of feature improvements / bugs introduced by this PR?
* [ ] Have you run the CLION code formatter on all the code?
* [ ] Have you merged your branch with master before opening this PR?
* [ ] Have you successfully ran tests with your changes locally?
* [ ] Have you successfully ran Cppcheck and fixed all warnings?
```

A.2. User Story Template

```
<!--- Provide a general summary of the user story in the Title above -->

## Description
```

```
<!--Provide a general description of the task, and what it will add to the game
 in context of the according user story -->

## Tasks
<!-- Link all tasks that must be defined with this user story here -->

## Acceptance criteria
<!--- Describe the steps in how one could identify this user story to be implemented-->
* [ ] Criteria 1
* [ ] Criteria 2
* [ ] Criteria 3
* [ ] Criteria 4

## Possible Implementation
<!--- Not obligatory, but suggest an idea for implementing addition or change -->
```

A.3. Task Template

```
<!--- Provide a general summary of the task in the Title above -->
# Assigned to:
<!-- Mention person responsible -->

<!--Please estimate time spend by writing /estimate x hours,
and update this by /spend x hours -->

## Description
<!--Provide a general description of the task, and what it will add to the game
in context of the according user story -->

## Definition of Done
<!--- Describe the steps in how one could identify this task to be implemented-->
* [ ]
* [ ]
* [ ]
* [ ]

## Possible Implementation
<!--- Not obligatory, but suggest an idea for implementing addition or change -->

## Motivation for prioritization
<!--- Specify the priority set and describe why it is this way -->
Priority level: High / Medium / Low

# Checklist

* [ ] Assign milestone
* [ ] Assign time estimation
* [ ] Add due date (informal deadline, if applicable)
* [ ] Assign member responsible
* [ ] Assign prioritization
* [ ] Assign additional labels
```

A.4. Bug Template

```
<!-- Please have a look at the FAQ on the wiki before submitting a bug issue -->
```

```
<!--- Provide a general summary of the issue in the Title above -->

<!--Please estimate time spend by writing /estimate x hours, and update this by
/spend x hours -->

## Expected Behavior
<!--- Tell us what should happen -->

## Current Behavior
<!--- Tell us what happens instead of the expected behavior -->

## Possible cause
<!--- Not obligatory, but suggest a cause of the bug, -->

## Steps to Reproduce
<!--- Provide a link to a live example, or an unambiguous set of steps to -->
<!--- reproduce this bug. Include code to reproduce, if relevant -->
1.
2.
3.
4.

## Possible Implementation
<!--- Not obligatory, but suggest an idea for implementing addition or change -->

## Checklist

* [ ] Assign milestone
* [ ] Assign time estimation
* [ ] Add due date (informal deadline, if applicable)
* [ ] Assign member responsible
* [ ] Assign prioritization
* [ ] Assign additional labels
```