|  |
| --- |
| Hogeschool van amsterdam |
| Methods and Techniques |
| Sprint 2 |
|  |
| **Bart Dikmans** |
| Student Nummer: 500681415 Opleiding: Game Development Telefoon: 06-46270911 E-mail: bart.dikmans@hva.nl |

# Table of Contents

[Table of Contents 1](#_Toc448142542)

[C++ 2](#_Toc448142543)

[Tooling 3](#_Toc448142544)

[Coding Standards 5](#_Toc448142545)

[Testing 6](#_Toc448142546)

[Porting 6](#_Toc448142547)

[Documentation 6](#_Toc448142548)

[Refactoring 7](#_Toc448142549)

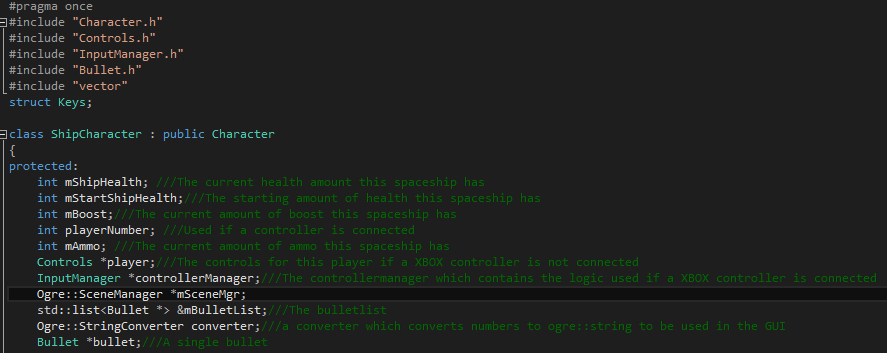
# C++

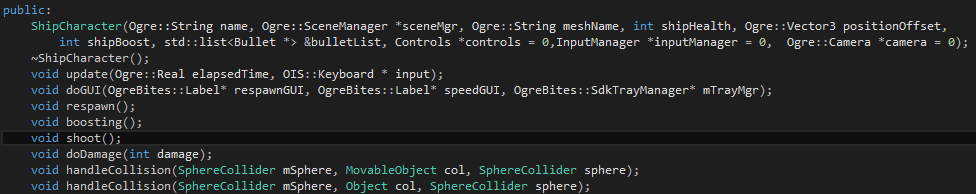
**Points:** 9-10

**Requires:** C++ language specific concepts are used throughout the code, including namespaces and inheritance, and advanced concepts are used and motivated in the report.

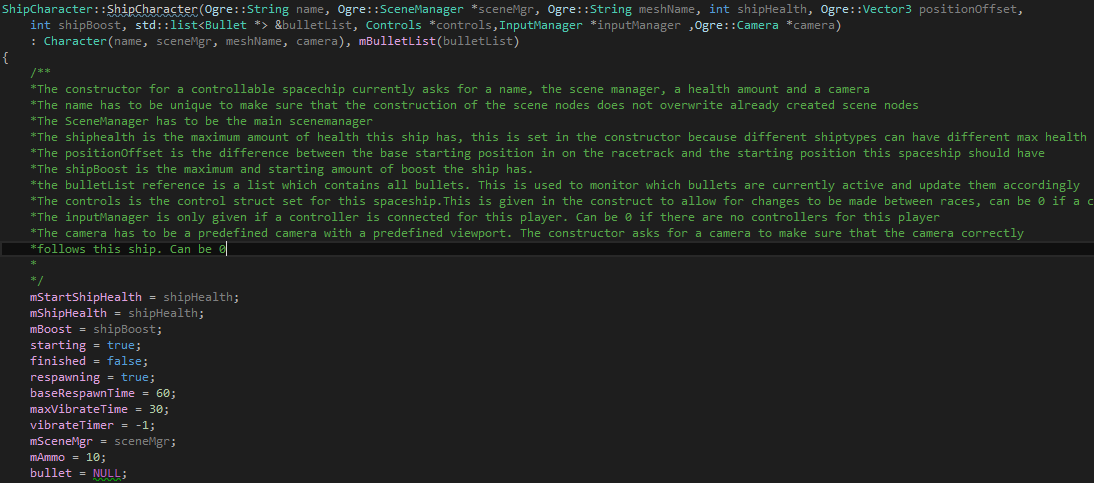
This is my first time working with C++ but I already like it a lot. The header files are great and make for clean coding. I’ve focused on creating some of the more specific classes like the ShipCharacter and Bullet classes as well as implemented multiple input devices like a XBOX controller or the regular keyboard.

The following code snippet is a part of the ShipCharacter.h





This code snippet shows some of the variables used by the spaceship as well as the constructor and the functions. Due to the use of Doxygen all the variables have a comment behind them but the functions don’t. This is because all the functions have their own separate comments in the .cpp file which can be seen in the next code snippet of a part of the ShipCharacter.cpp file



This code snippet shows the constructor of the ShipCharacter class with the included comments. The following code snippet shows some of the logic used in the update function if a controller is connected before refactoring was done.



For further proof of my level of c++ you can check the following classes:

* ShipCharacter
* Bullet
* InputManager
* XBOXController
* Controls

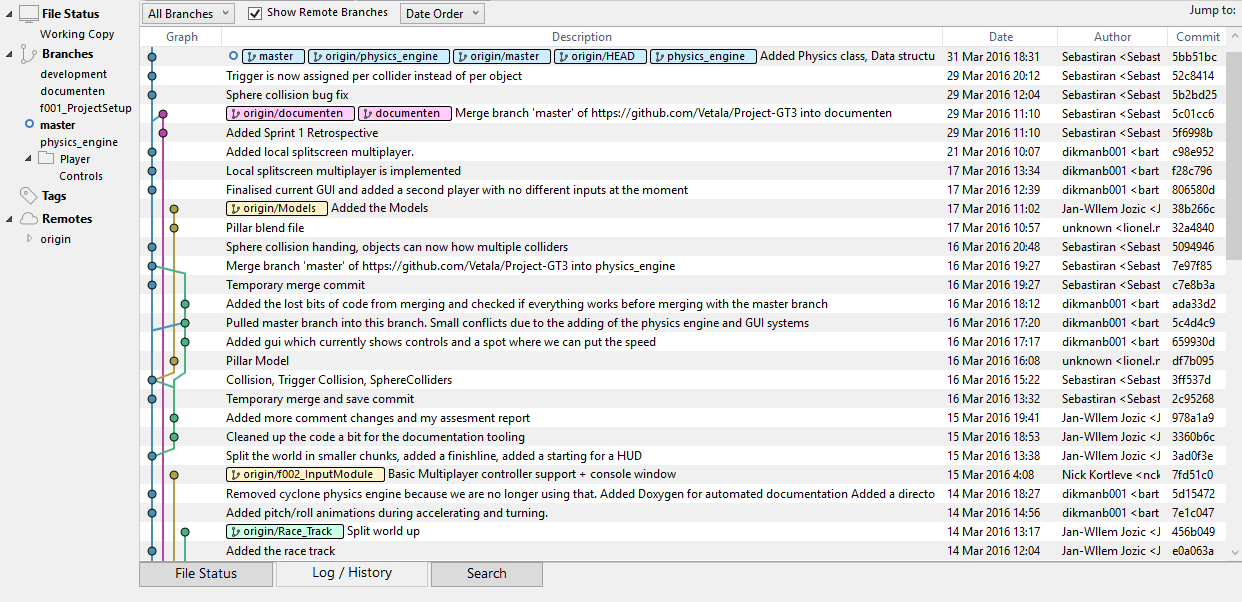
# Tooling

**Points:** 6-8

**Requires:** Student can explain basic GIT concepts and conflict resolution methods; commits are pushed to multiple branches; the branch model is motivated. The student or team has implemented a tool for creating and/or importing assets.

Our version control is done via GitHub. We use feature branches and not developer branches. The reason for this is because feature branches give a better overview of what has been added when and make for easy checkout to a version without a certain feature. A picture of our current Git branch state can be found below.

As for assets: So far we only imported models. We make them using Blender, a free modelling tool. Blender can export the model to almost all mesh types. Ogre uses .mesh.



Beside these tools we also use Doxygen to automaticly generate our documentation automaticly. This can be found at <https://oege.ie.hva.nl/~dikmanb001/> . Personally I also use ReSharper to allow for simpler code refactoring.

# Coding Standards

**Points:** 9-10

**Requires:** A set of coding standards, developed by the student and/or team, is documented, and generally enforced throughout the code; the motivation for using these particular coding standards is documented in the report

We use the following coding standards:

Every page of code will have a small summary at the top which accurately describes the code.

All functions have to start with a capital letter. For example: Update() or Draw().

All variables will use standard camel case rules.

Every unclear function should have some comments to further describe the function. Rule of thumb to determine an unclear function is:

\*Is the function large (more than 15 lines of code)

\*Are there many variables that look alike/function almost similar

\*A function that cannot be understood if you read it like a person that doesn't know coding

Try to place big comments at the top of the function.

In the case of Egyptian brackets all statements will look like the following example:

if(test)

{

insert stuff here;

}

Try to avoid using cout. Use puts or printf instead.

The reason we have these coding standards is because they make the code more consistent and readable. Rules like “Give the variable a reasonable name” are left out because this is to be automatically expected and is not an actual rule. Aside from these rules we also agreed to review each other’s code and refactor the stuff that can be done better.

The reason we avoid using cout is minor, but it is not as fast as puts and printf and we like to be uniform.

There are multiple functions that do not start with an uppercase. This is because these functions are all part of the ogreSDK which means modifying those takes a long time and has a chance of breaking the entire SDK if done incorrectly. It also shows us which functions are made by us and which are made by Ogre.

# Testing

**Points:** 6-8

**Requires:** A testing framework is implemented throughout the team’s code, and the choice for this framework is motivated

Whenever we make changes to the project, before pushing it to the master branch it undergoes some tests. We made these tests up ourselves and test them manually. Every sprint the tests are updated to the new features implemented. For this sprint we test if:

* The Collision is working with Objects and Movable Objects.
* Power ups are working (For every power up, check if it does what is should do and nothing more)
* Controls are working
* GUI is displaying everything correct (Speed, health and boost are being updated correctly)
* Multiplayer is working
* The level is working (You can finish and respawn + objects have their collision boxes and are displayed on the correct position)

The reason we test these things is because theses make up the entire game’s functionality. If these things work the game is working.

# Porting

**Points:** 0-2

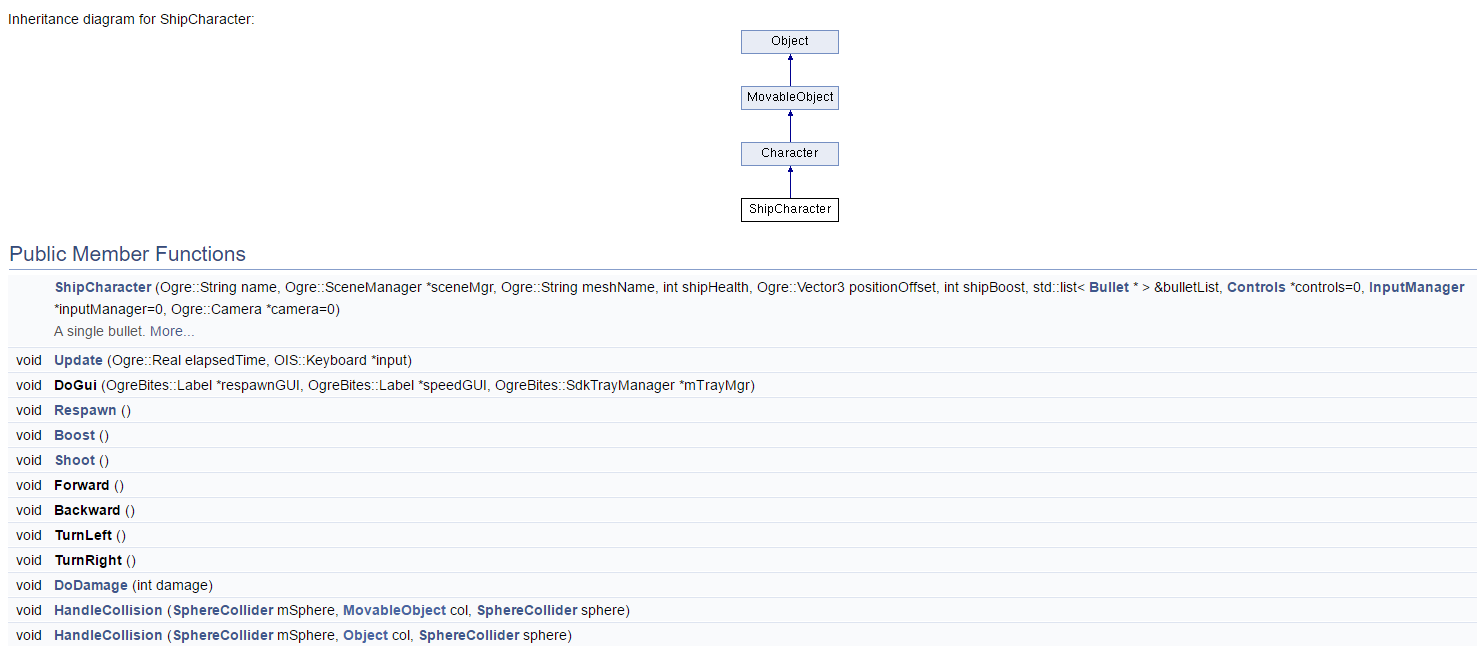
**Requires:** The game is not ported to another engine

# Documentation

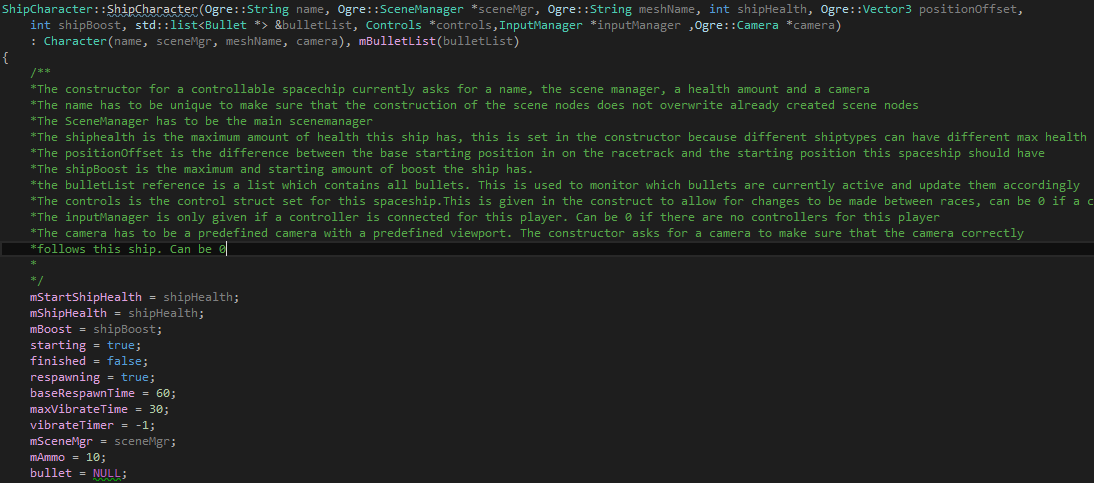
**Points:** 9-10

**Requires:** Tooling is used for automatic documentation generation

To document our project code we set up an automatic documentation tool called doxygen that takes the comments we write in our code and puts it on the following webpage: <https://oege.ie.hva.nl/~dikmanb001/> . The following screenshot is a part of the documentation written for the ShipCharacter class.



An example of the documentation in the code can be seen in the following code snippet



# Refactoring

**Points:** 6-8

**Requires:** Multiple good examples of refactoring are shown, each applicable to the situation, and motivated in the report

So far I have made multiple small refactors of the ways the ship can be controlled. For example at first the controls were set for the spaceship in the spaceship itself but now due to the addition of a controls struct we are able to modify the control scheme outside of the spaceship which allows for the existence of multiple modifiable control schemes.

I have also made a big refactoring attempt at the way the input is handled in the spaceship. Before the refactoring the update method had a major flaw which resulted in a possibility of discrepancy between the way the controller spaceship would accelerate and the way a spaceship controlled by keyboard would be controlled.

To solve this I made all the ways the spaceship behaves when a button is pressed into different functions using ReSharper resulting in similar behaviour for both control methods. The following code snippet shows the current code.

