|  |
| --- |
| Game Technology Assessment Report |
| Explosive Shark Studio’s |
| Sebastiaan van Dijk |

Sebastiaan van Dijk

500691806

31 maart 2016

Contents

[C++ 2](#_Toc447801558)

[Tooling 3](#_Toc447801559)

[Coding Standards 4](#_Toc447801560)

[Testing 5](#_Toc447801561)

[Porting 5](#_Toc447801562)

[Documentation 5](#_Toc447801563)

[Refactoring 6](#_Toc447801564)

# 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 made the data structure for our game by having general classes going down the tree to more specific classes. At the head of the tree is a class called Object. In which the general variables and methods are written that every object in the game requires.



You could argue the update function is not a function for the root class but maybe for a node down. Since the game isn’t very complex we don’t want to make our data structure to complex with too many nodes so we placed it there. Down the ladder is MovableObject which has a RigidBody and methods that handle collision. Even further down the ladder is a Character which has a Camera. This makes for clean code and objects that don’t have unnecessary variables and methods in them.

Another thing I wrote is the physics class. For now this class only holds certain structs like RigidBody and SphereCollider, also seen above in the Object class. Both are very basic with only the most needed methods, but this file will evolve over time to do the things we need.





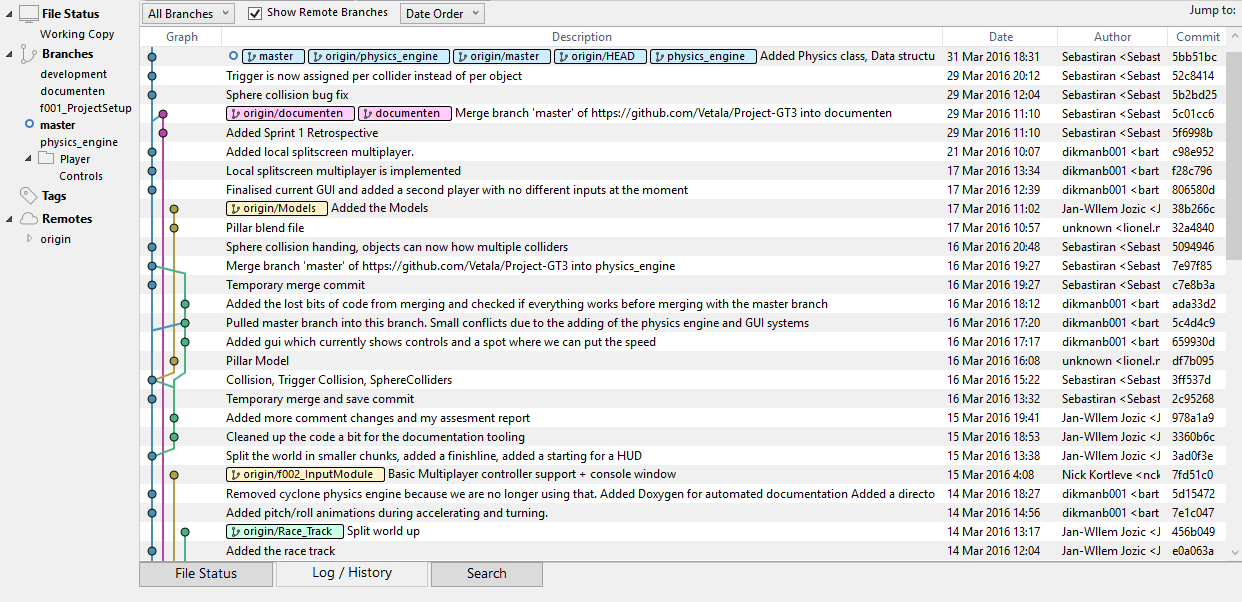
# Tooling

**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.



# 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.

# 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

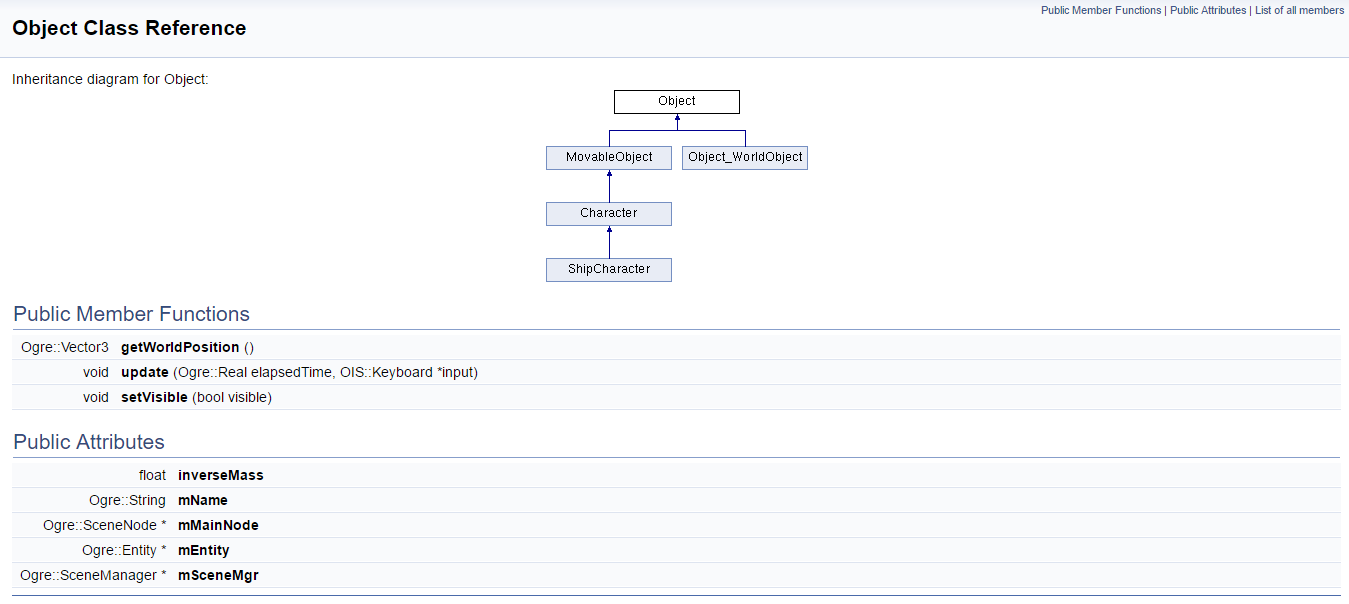
**Requires:** The game is not ported to another engine

# Documentation

**Points:** 9-10

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

To document our project code we set up an automatic documentation system that takes the comments we write in our code and puts it in an online document. The system was set up by Bart Dikmans, one of the project members. Here is a screenshot of how the system turned our comments into documentation:



# 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 done two major refactors. These can be found in the sprint 2 Trello board. <https://trello.com/b/8xBUsVSs/sprint-2>

One is about the rewriting of the collision spheres. Before, an object was able to have only one collision sphere. This was soon rewritten into a collision sphere list, so that you can have as much spheres as you like. Later we found out that we would like each sphere to be able to be a trigger collider instead of actually colliding. Before this was done per object. I rewrote the code so that, instead of per object, per collision sphere you can assign it to be a trigger. Yet this was all done in the object class and it became a rather messy class to look at. This is where the second refactoring comes in to play: The making of a physics class. Each class began to have lots of “variable groups” that were separated from other variables by a white line. Take for instance the Sphere collider and its trigger. I decided to take all of these variable out of their classes and make structs out of them in a physics class. So instead of this:



You now have this:



Which, in my opinion, is a lot nicer. I also added functions to add force and integrate in the Rigidbody struct. This makes for even less code in the object classes.