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

Sebastiaan van Dijk

500691806

31 maart 2016

Contents

[C++ 2](#_Toc448179119)

[Tooling 3](#_Toc448179120)

[Coding Standards 4](#_Toc448179121)

[Testing 5](#_Toc448179122)

[Porting 5](#_Toc448179123)

[Documentation 5](#_Toc448179124)

[Refactoring 6](#_Toc448179125)

[Advanced Techniques 6](#_Toc448179126)

[Ogre’s Physics and Architecture 6](#_Toc448179127)

[What we want 7](#_Toc448179128)

[The steps taken 7](#_Toc448179129)

[Sources 8](#_Toc448179130)

# 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 architecture 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 architecture 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.





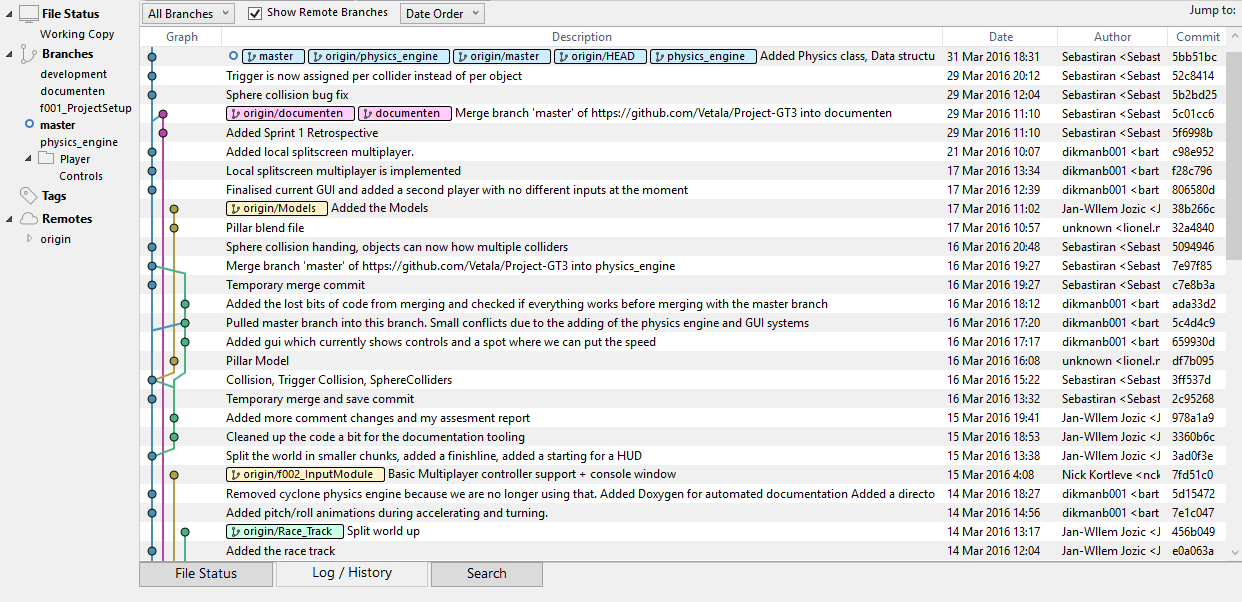
# Tooling

**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

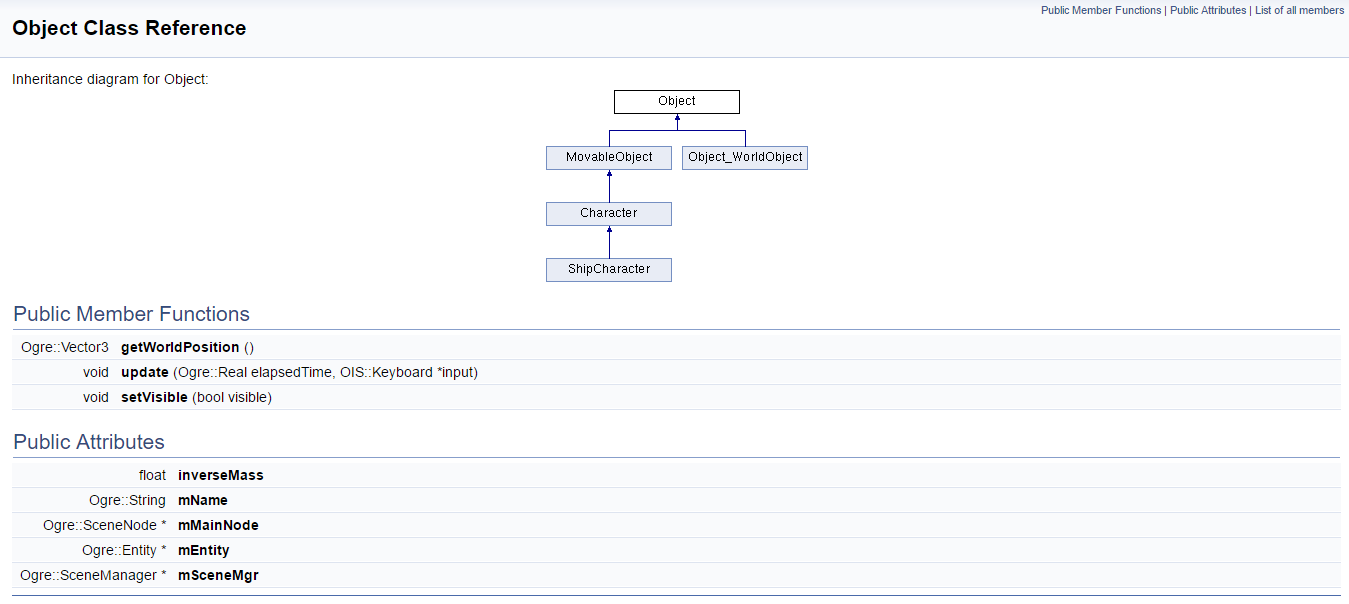
**Requires:** The game is not ported to another engine

# 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 variables out of their classes and make structs out of them in a physics class. So instead of this:



You now have this:



In my opinion, the latter 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.

# Advanced Techniques

**Points:** 6-8

**Requires:** The report describes one or more advanced techniques; successfully applied in the game; the report shows how this technique works, and how it was applied, with references to relevant literature.

My advanced technique is Physics and Architecture. I will talk about the following subjects:

* The state of Ogre’s Physics and Architecture
* What I want to accomplish within Ogre
* The steps taken to accomplish what I want to accomplish

I will try to keep this as short as possible without leaving any valuable information out. Personally I wouldn’t call this a research report. Rather a summary of my research and thought process.

## Ogre’s Physics and Architecture

When we picked Ogre as the base for our game we knew Ogre didn’t have much physics build in. Ogre is mostly a rendering engine. What it does have is built in raycasts and AABB’s (Axis Aligned Bounding Box). With these you can check for intersection with other objects, but what happens when objects intersect is all up to you. There is nothing that handles the collision for you or a method that is called when objects collide. AABB’s are very simple shapes and thus easy and fast when it comes to checking for collision, but when an object rotates the AABB needs to be recalculated which isn’t ideal when you have objects that constantly rotate. Ogre does have spheres and boxes but before I can talk about them you need to know something about how Ogre’s architecture works.

Ogre is, in a way, similar to Unity3D. In Ogre you have something called a node. A node is nothing more than an empty game object that has a position, a scale and a rotation. Nodes can have child nodes that have a relative position to their parent node. All nodes exist is a scene, which is managed by something called a scenemanager. This scenemanager has access to all nodes and their attached objects. Nodes can have things attached to them called objects. An object can be an entity, something to actually display some shape or mesh on the screen. But it can also be something like a camera, or a light, or a particle system. There is just one requirement when it comes to attaching an object: It needs be a subclass of Ogre’s MovableObject class. Unfortunately Ogre’s spheres and boxes are not subclasses from MovableObject. They are simply primary shapes. Thus there is no way to attach collision shapes to objects.

The newest version of Ogre does not have any physics whatsoever implemented. On the internet people are split into two groups. They either use the raycasts and AABB’s Ogre provides and make their own very simple collision handler or they import a physics engine. There are several physics engine that can be used with Ogre. Unfortunately most of them are either outdated or require a manual to install, but they don’t have a manual which makes it rather difficult.

## What we want

Our game is a space racer. There is no need for advanced physics and we don’t need it to be very realistic (Talking about you RK4). We don’t need crazy matrices and angular rotation physics. The project is only fifteen weeks long. With only ten weeks to get a grade. We won’t have time to implement all that, and it is not that high on the priority list. What we want is basic particle physics and colliders that we can put on our objects. Sphere colliders are enough, and if we want to expand we can even implement box colliders. We want sphere colliders that work properly, that can be attached to objects with ease and have customizable sizes and positions. We also want objects to be able to have multiple colliders and triggers. Triggers are colliders that don’t actually apply physics to objects when they collide but give a signal that they collide.

Another thing we want is custom collision checks (Similar to Unity3D’s layers). For example, we don’t want bullets to check collision with powerups. We also don’t want unmovable objects to check for collision with other unmovable objects. But we do want the spaceships to be able to collide with practically everything. Objects are required to have a unique name, and can have a tag if assigned to make for easy grouping of objects.

## The steps taken

With Ogre’s sphere not being a subclass of Ogre’s MovableObject and thus not able to attach to a node as a collider, I had to start thinking. I want my nodes to have collision spheres. But they cannot be attached. This leaves me with two options. I either dig really deep into Ogre code and make it so that nodes can have collision spheres that are subclasses of MovableObject or I just don’t attach the shapes to the node. Looking back it might not have been that hard to make collision shapes that are subclasses of MovableObject, but I decided to do the latter.

First we need a decent structure for our objects. We decided to use the structure we learned on school during Game Programming and Architecture. At the root we have a class called Object. All static objects are a direct child from this class. Objects only have the required variables to display it on the screen and a list of SphereColliders. Optionally it has a tag and a PhysicsMaterial to change the object’s bounciness.

Objects that are able to move are a child class of MovableObject. This is not the same MovableObject I talked about earlier. This one is our own MovableObject class and doesn’t have the namespace Ogre. This class is a subclass of Object and thus has the same variables Object has plus a RigidBody. I’ve dropped the terms SphereCollider and Rigidbody. These are not Ogre variable. These come from the physics.h file and are made by me. For a good overview see our project’s documentation: <https://oege.ie.hva.nl/~dikmanb001/class_object.html>

So we now have the base for creating objects. We can now easily assign colliders and triggers to objects and change their properties. We can have as many as we want on an object and require only three lines of code to add. But we do nothing with the colliders. We haven’t integrated collision when objects collide. We don’t even check if objects collide at all. Let’s change that.

The TutorialApplication class is a default class created by Ogre. It contains the scenemanager and is used to create all objects required for a scene. In here we also create the in Unity3D so called “layers”, but in Ogre they are technically just lists. Currently we use four layers: Ships, Bullets, Powerups and Objects. When an object is created we push it to one of these lists. Every frame we call a method that checks for collision but only between certain layers. Ships on ships, ships on bullets, ships on powerups and ships on objects. All collision between other layers is not being check for and thus saves a lot of calculation time. Whenever an object collides, that object’s handleCollision method is called. This method is completely customizable per object. For colliders on objects that are not triggers, collision is handled. This is done with simple particle physics I won’t be explaining in this report. If you are interested in how this physics works you can read any of the books I mention at the end of this report, or the article Nilson Souto wrote about Collision Detection for Solid Object also mentioned at the end of this report. Aside from handling collision, one can do other things in the handleCollision method depending on the object you collided with. For example, if a ship collides with a bullet it takes damage.

And that’s it. The most basic physics and architecture you need to make a space racer. Personally I’m really happy with the result. But while writing this report I’ve already come up with some new ideas to make the architecture even better. This is an ongoing advanced technique with much more to learn. For a better understanding of the architecture I can recommend checking our project’s documentation mentioned earlier or downloading our source code from github.

## Sources

I read a lot of articles about this subject. I won’t drop the entire list here, just the ones that helped me the most with my research and are being used in this summary. I’ll start with the literature I read.

Ian Millington, Game Physics Engine Development, Series in interactive 3D technology

This book I think was the most useful. It describes really well what kind of physics engine there are out there, what the difference between them is, and what kind of engine would be suitable for your game. And this is only the first chapter. The other chapters are also really useful when it comes to understanding and implementing physics.

Christer Ericson, Real time Collision Detection, Series in interactive 3D technology

Another great book within the series in interactive 3D technology.

Two articles I read that are worth checking out are the following, one is about physics and the other about simple collision detection in Ogre.

Nilson Souto, Toptal, Video Game Physics part II: Collision Detection for Solid Objects,

[https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects](https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects%20)

Cordinc, Ogre Beginner for Beginner Turotial: Opponent & simple collision detection,

<http://www.cordinc.com/blender/tutorials/mech/6_ogre_part2/>

A lot of my research comes from coding and learning about Ogre code. I can recommend checking out the Ogre documentation <http://www.ogre3d.org/docs/api/1.9/>

Some of the references used in my summary:

Ogre3D, Ogre::SceneNode Class Reference: attachObject,

[http://www.ogre3d.org/docs/api/1.9/class\_ogre\_1\_1\_scene\_node.html#a51052c58747a77a6cf7451b700e7eb87](http://www.ogre3d.org/docs/api/1.9/class_ogre_1_1_scene_node.html%23a51052c58747a77a6cf7451b700e7eb87)

Ogre3D, Ogre::MovableObject Class Reference,

<http://www.ogre3d.org/docs/api/1.9/class_ogre_1_1_movable_object.html>