Hogeschool van Amsterdam

Game Technology Assessment Report Explosive Shark Studio's

Titelblad

Explosive Shark Studio's

Studentnaam	**:**	**Jan-Willem Jozic**
Studentnummer	**:**	**500623980**
Plaats	**:**	**Hoofddorp**
Datum van uitgave	**:**	**15-3-2016**
Opleiding instituut	**:**	**Hogeschool van Amsterdam**

Introduction

This report documents the C++ project of Explosive Shark Studio's member Jan-Willem Jozic

It will walk you, the reader, through the code and motivates, how each criteria is met and add images and code to prove the work.

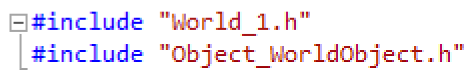
At the end of each chapter there will be a short reasoning of why a certain criteria grade is made and the grade itself.

C++

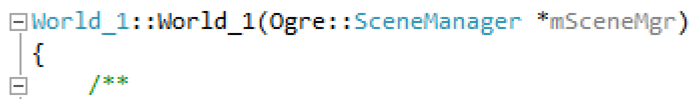
In Sprint 1 I have mainly been responsible for making the 3D models and adding them into the game world.

As such I have not yet had an opportunity to extensively write a lot of code in C++, which means the amount of language specific concepts is small.

The following segments show the usage of Scope resolution operators, Pointers,

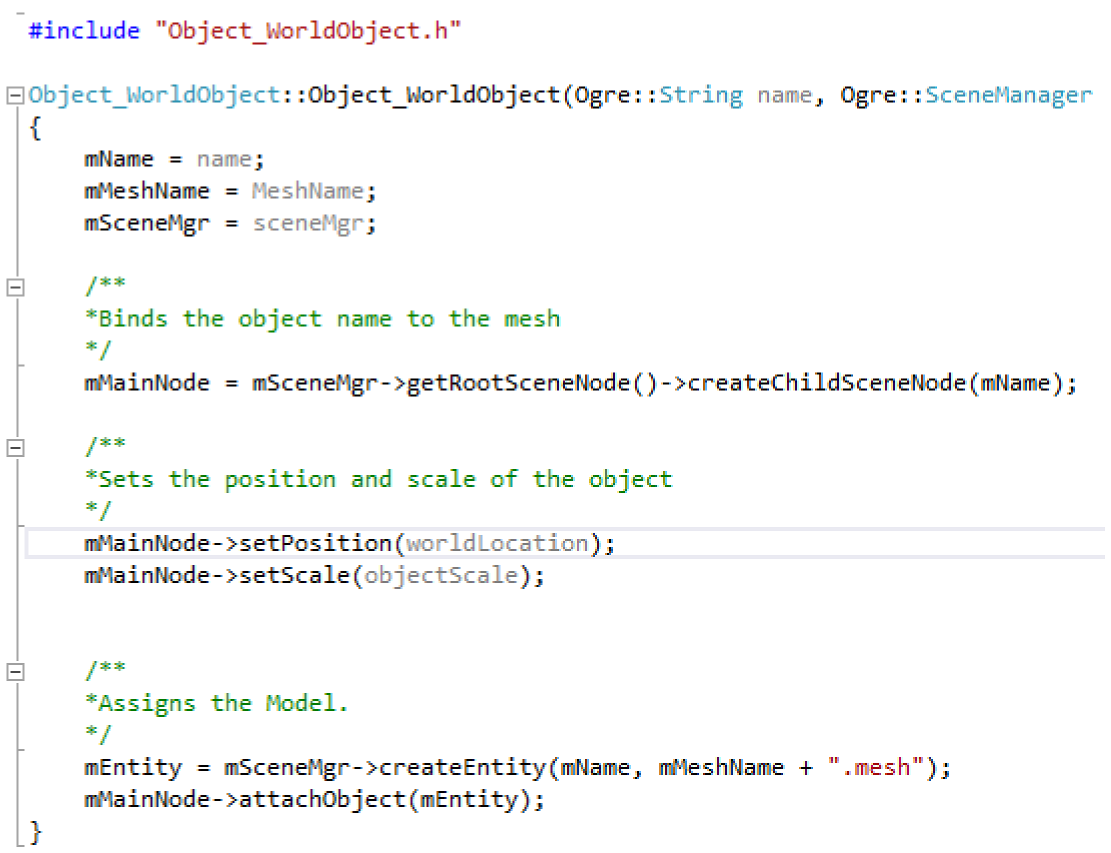


```cpp
#include "World_1.h"
#include "Object_WorldObject.h"
```



```cpp
World_1::World_1(Ogre::SceneManager *mSceneMgr)
{
    /**
```

And Member selection operators

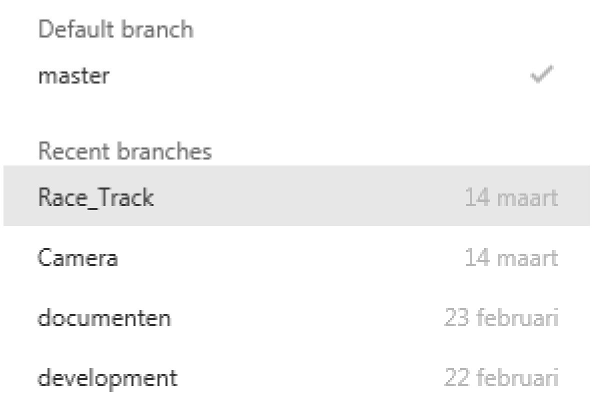


```cpp
#include "Object_WorldObject.h"

Object_WorldObject::Object_WorldObject(Ogre::String name, Ogre::SceneManager
{
    mName = name;
    mMeshName = MeshName;
    mSceneMgr = sceneMgr;

    /**
    *Binds the object name to the mesh
    */
    mMainNode = mSceneMgr->getRootSceneNode()->createChildSceneNode(mName);

    /**
    *Sets the position and scale of the object
    */
    mMainNode->setPosition(worldLocation);
    mMainNode->setScale(objectScale);


    /**
    *Assigns the Model.
    */
    mEntity = mSceneMgr->createEntity(mName, mMeshName + ".mesh");
    mMainNode->attachObject(mEntity);
}
```

Other than that I haven't written a lot of C++ in this project yet, so I would self-asses myself a 5 for this criteria.

Tooling

Our group uses GitHub to maintain version control of our Ogre3D product, inside GitHub we've created multiple sub-branches to develop the game. After a part of the branch successfully works it can get approved for merging it to the master branch project.



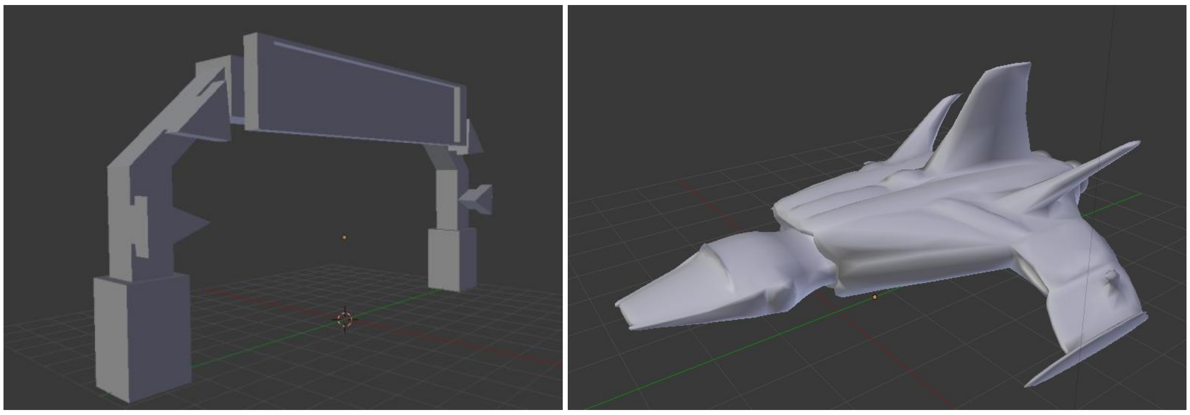
Default branch

master	✓

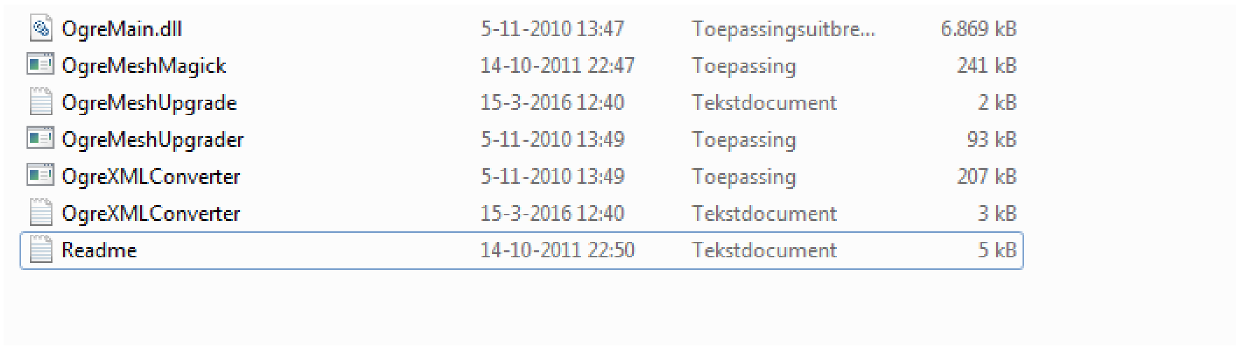
Recent branches

Race_Track	14 maart
Camera	14 maart
documenten	23 februari
development	22 februari

In addition to using Git our group utilizes tools for creating assets for our game. the 3D modeling program Blender is primarily used for our modeling.



To easily convert the blender files we have installed an Ogre Mesh Converter that can transform the meshes created with blender into a .mesh format that Ogre3D can instantly use.



OgreMain.dll	5-11-2010 13:47	Toepassingsuitbre...	6.869 kB
OgreMeshMagick	14-10-2011 22:47	Toepassing	241 kB
OgreMeshUpgrade	15-3-2016 12:40	Tekstdocument	2 kB
OgreMeshUpgrader	5-11-2010 13:49	Toepassing	93 kB
OgreXMLConverter	5-11-2010 13:49	Toepassing	207 kB
OgreXMLConverter	15-3-2016 12:40	Tekstdocument	3 kB
Readme	14-10-2011 22:50	Tekstdocument	5 kB

These tools combined fulfill the intermediate criteria for Tooling, as such I would self-asses these actions as an 8

Coding Standards

Initially we had set up a coding standard for our group which contained the method of how we'd write classes, functions and methods.

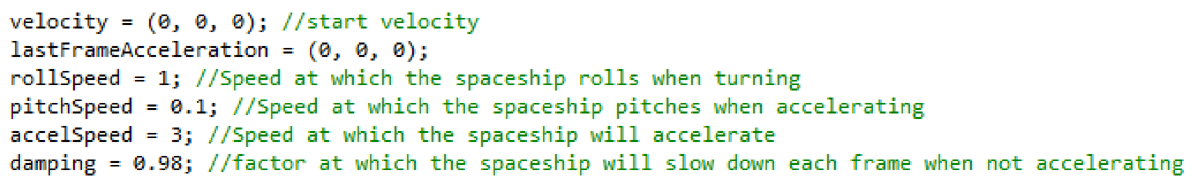
Including things such as having { and } on new lines,

In addition this coding standard strongly encouraged writing comments explaining all parts of a function right above and only adding any comment information inside when it was a very extensive or difficult function.

With the setup of our online documenting tool (which will be explained later in this report), the standard changed.
After the tool was implemented, classes needed to add information at the start

```
/**
* @class Object_WorldObject
* @author Explosive Shark Studios
* @date 15/03/2016
* @brief
*
* @section Description
* This class contains the input function to spawn world objects.
*
*/
```

And comment information for each line can be on the back of the line



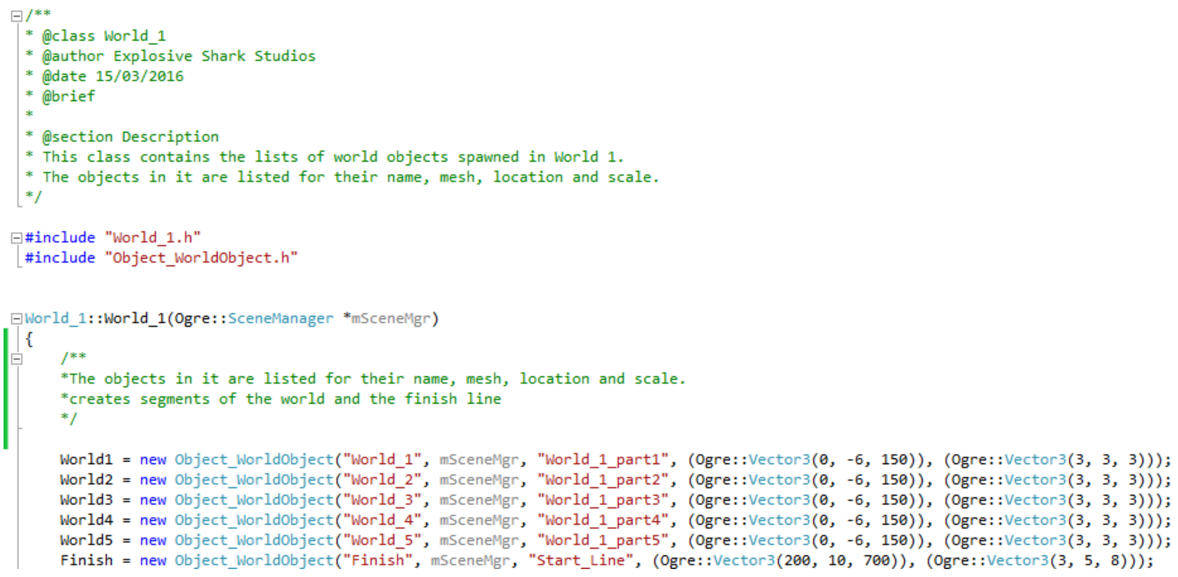
```
velocity = (0, 0, 0); //start velocity
lastFrameAcceleration = (0, 0, 0);
rollSpeed = 1; //Speed at which the spaceship rolls when turning
pitchSpeed = 0.1; //Speed at which the spaceship pitches when accelerating
accelSpeed = 3; //Speed at which the spaceship will accelerate
damping = 0.98; //factor at which the spaceship will slow down each frame when not accelerating
```

Because these coding standards contribute to the usage of an automatically documented tool it's essential they be actively used. This motivation would qualify this chapter of Coding Standards to the Advanced 10 points

Documenting

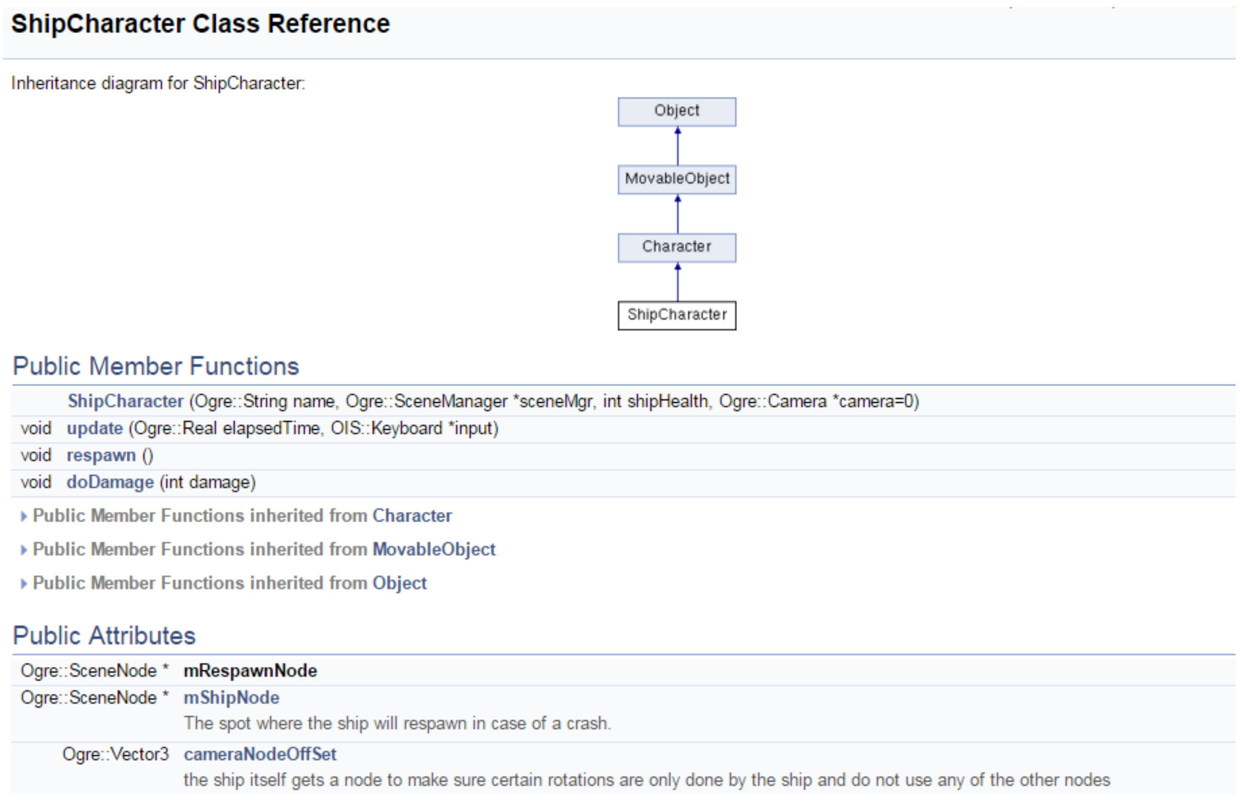
For documenting our project group has set up a system that will automatically create a document from out code through set comments.

The tool was set up by project member: Bart Dikmans



```
/**
 * @class World_1
 * @author Explosive Shark Studios
 * @date 15/03/2016
 * @brief
 *
 * @section Description
 * This class contains the lists of world objects spawned in World 1.
 * The objects in it are listed for their name, mesh, location and scale.
 */

#include "World_1.h"
#include "Object_WorldObject.h"


World_1::World_1(Ogre::SceneManager *mSceneMgr)
{
    /**
    *The objects in it are listed for their name, mesh, location and scale.
    *creates segments of the world and the finish line
    */

    World1 = new Object_WorldObject("World_1", mSceneMgr, "World_1_part1", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World2 = new Object_WorldObject("World_2", mSceneMgr, "World_1_part2", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World3 = new Object_WorldObject("World_3", mSceneMgr, "World_1_part3", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World4 = new Object_WorldObject("World_4", mSceneMgr, "World_1_part4", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World5 = new Object_WorldObject("World_5", mSceneMgr, "World_1_part5", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    Finish = new Object_WorldObject("Finish", mSceneMgr, "Start_Line", (Ogre::Vector3(200, 10, 700)), (Ogre::Vector3(3, 5, 8)));
```

The resulting documentation looks like the ShipCharacter class (written by Bart Dikmans)



ShipCharacter Class Reference

Inheritance diagram for ShipCharacter:

Public Member Functions

	ShipCharacter (Ogre::String name, Ogre::SceneManager *sceneMgr, int shipHealth, Ogre::Camera *camera=0)
void	**update** (Ogre::Real elapsedTime, OIS::Keyboard *input)
void	**respawn** ()
void	**doDamage** (int damage)

▸ Public Member Functions inherited from **Character**

▸ Public Member Functions inherited from **MovableObject**

▸ Public Member Functions inherited from **Object**

Public Attributes

Ogre::SceneNode *	**mRespawnNode**
Ogre::SceneNode *	**mShipNode** The spot where the ship will respawn in case of a crash.
Ogre::Vector3	**cameraNodeOffSet** the ship itself gets a node to make sure certain rotations are only done by the ship and do not use any of the other nodes

Although an Individual grade is given for the Documentation criteria, the Advanced grade requirement to setup an automatic tool would only require one person to set it up.

But after that and using that tool would give a self-assesment of the full 10 points.

Refactoring

Originally the object class contains variables that would help identify it, give it a location and mesh amongst other things.

Object.h

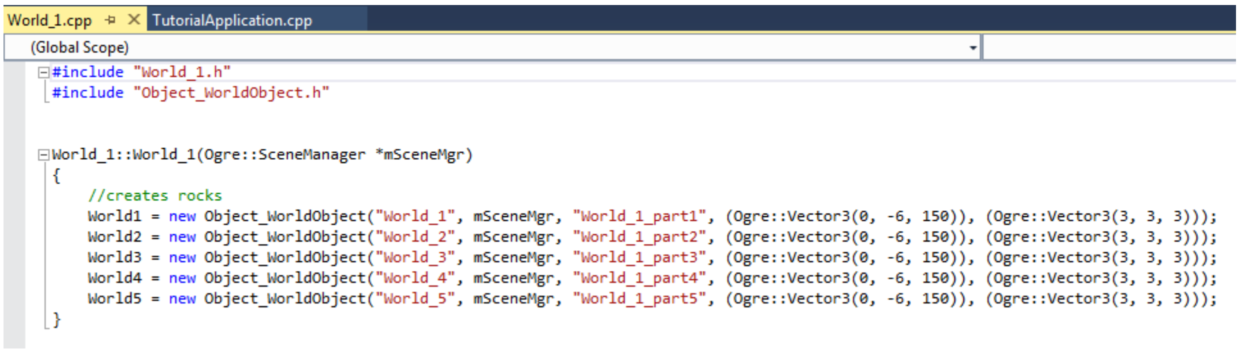
```
class Object
{
public:
    float inverseMass;

    Ogre::String mName;
    Ogre::SceneNode *mMainNode; // Character position and rotation
    Ogre::Entity *mEntity; // Mesh
    Ogre::SceneManager *mSceneMgr;

    Object();
    Ogre::Vector3 getWorldPosition() {
        return mMainNode->_getDerivedPosition();
    }
    void update(Ogre::Real elapsedTime, OIS::Keyboard * input);
    void setVisible(bool visible);
```

However it was initially written in a way that each object would have to be called in one class and then defined in another.

```
    // Give this character a shape
    mEntity = mSceneMgr->createEntity(mName, "Ship2.mesh");
    mMainNode->attachObject(mEntity);
    respawnTimer = baseRespawnTime;
```

This has then been refactored so that when an object gets called, all information can be added, Not just a name and mesh type, but also its location and scale.

```
Object_WorldObject::Object_WorldObject(Ogre::String name, Ogre::SceneManager *sceneMgr,
 Ogre::String MeshName, Ogre::Vector3 worldLocation, Ogre::Vector3 objectScale)

    mEntity = mSceneMgr->createEntity(mName, mMeshName + ".mesh");
    mMainNode->attachObject(mEntity);
```



```
#include "World_1.h"
#include "Object_WorldObject.h"


World_1::World_1(Ogre::SceneManager *mSceneMgr)
{
    //creates rocks
    World1 = new Object_WorldObject("World_1", mSceneMgr, "World_1_part1", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World2 = new Object_WorldObject("World_2", mSceneMgr, "World_1_part2", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World3 = new Object_WorldObject("World_3", mSceneMgr, "World_1_part3", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World4 = new Object_WorldObject("World_4", mSceneMgr, "World_1_part4", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World5 = new Object_WorldObject("World_5", mSceneMgr, "World_1_part5", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
}
```

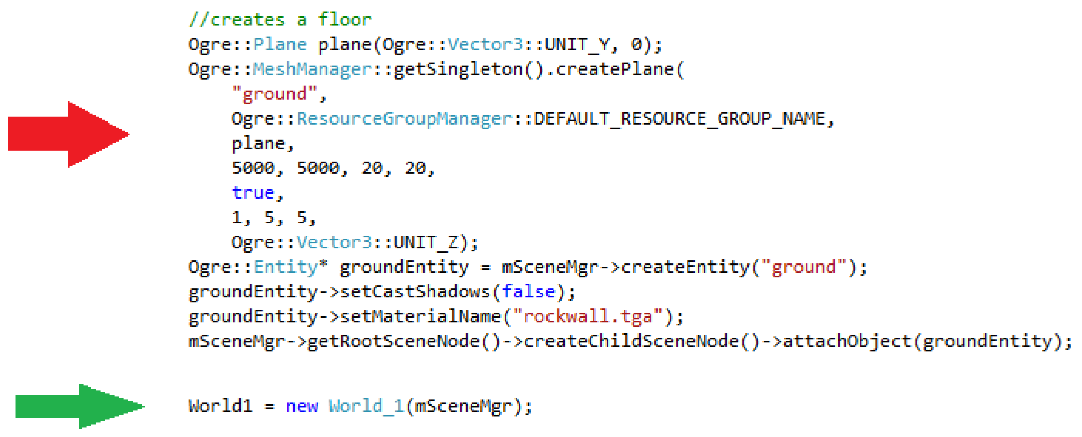
After that I made a similar refactoring to the way the world was generated. This was first done manually by adding objects in the TutorialApplication.cpp and defining them there.

However these were very extensive and if you need to generate 50 objects that each need around 13 lines of code that would cause a massive clutter of code.

In order to keep the TutorialApplication.cpp clean of massive clutter of spawning dozens of objects inside the scene I refactored the spawning of objects into a separate class for this scene called World_1.

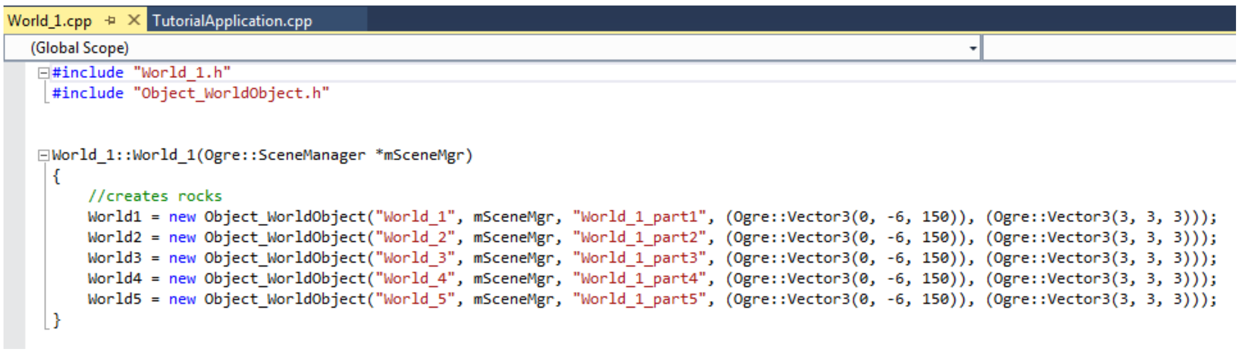
By doing this refactor there would only be one line of code needed in the TutorialApplication.cpp keeping it clean.

TutorialApplication.cpp



```
//creates a floor
Ogre::Plane plane(Ogre::Vector3::UNIT_Y, 0);
Ogre::MeshManager::getSingleton().createPlane(
    "ground",
    Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
    plane,
    5000, 5000, 20, 20,
    true,
    1, 5, 5,
    Ogre::Vector3::UNIT_Z);
Ogre::Entity* groundEntity = mSceneMgr->createEntity("ground");
groundEntity->setCastShadows(false);
groundEntity->setMaterialName("rockwall.tga");
mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(groundEntity);


World1 = new World_1(mSceneMgr);
```

And on the other hand, inside the World_1 class a massive amount of objects can be spawned in an orderly fashion as seen earlier in the way world objects were called.



```
#include "World_1.h"
#include "Object_WorldObject.h"


World_1::World_1(Ogre::SceneManager *mSceneMgr)
{
    //creates rocks
    World1 = new Object_WorldObject("World_1", mSceneMgr, "World_1_part1", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World2 = new Object_WorldObject("World_2", mSceneMgr, "World_1_part2", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World3 = new Object_WorldObject("World_3", mSceneMgr, "World_1_part3", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World4 = new Object_WorldObject("World_4", mSceneMgr, "World_1_part4", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
    World5 = new Object_WorldObject("World_5", mSceneMgr, "World_1_part5", (Ogre::Vector3(0, -6, 150)), (Ogre::Vector3(3, 3, 3)));
}
```

The TutorialApplication class is a very important one because it's the core of the game, as such many different codes are being called and handled in there. My refactoring in that class was a very important one because it ensures that the application is kept clean at the core. Which for coding purposes is essential.

In addition, the refactoring made in the generating of world objects gives an easy way to both create and define world objects in a simple easy to change format, and all collected in one file.

These changes are clean and add utility and simple overview, I would self-asses these actions as an 8 (Intermediate) for the subject of Refactoring.