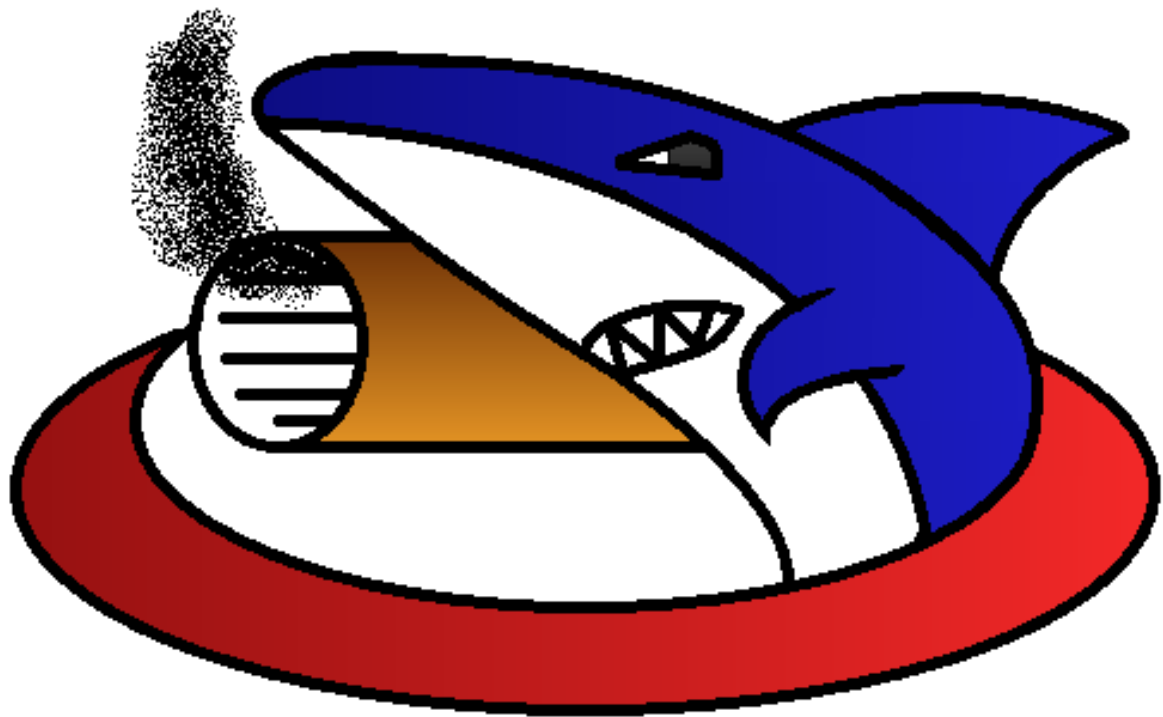Hogeschool van Amsterdam

Game Technology Assessment Report Explosive Shark Studio's

Titelblad

Studentnaam : **Jan-Willem Jozic**
Studentnummer : **500623980**
Plaats : **Hoofddorp**
Datum van uitgave : **12-4-2016**
Opleiding instituut : **Hogeschool van Amsterdam**

Table of Contents

C++

Points: 6-8
Requirements: C++ language specific concepts are used **throughout** the code, including namespaces and inheritance.

The Object class right under here was written by team member Sebastiaan van Dijk

```cpp
#pragma once
#include "Physics.h"
#include "BaseApplication.h"

class Object
{
public:
	Ogre::String mName;
	Ogre::String mTag;
	Ogre::String mMeshName;
	Ogre::SceneNode *mMainNode; // Character position and rotation
	Ogre::Entity *mEntity; // Mesh
	std::list<SphereCollider *> sphereColliders;
	std::list<BoxCollider *> boxColliders;
	Ogre::SceneManager *mSceneMgr;

	PhysicsMaterial *physicsMat;

	Object(Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String meshName);
	~Object();

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

The WorldObject class inherits functions from the Object class

```cpp
#pragma once
#include "Object.h"

class WorldObject : public Object
{
public:
	Ogre::Vector3 worldLocation;
	Ogre::Quaternion worldRotation;
	Ogre::Vector3 objectScale;

	WorldObject(Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String
meshName, Ogre::Vector3 worldLocation, Ogre::Quaternion worldRotation, Ogre::Vector3
objectScale, std::list<Ogre::Sphere *> sphereList);
	~WorldObject();
	void Update(Ogre::Real elapsedTime, OIS::Keyboard * input);
};
```

Which in turn gets implemented in the WorldObject.cpp

```cpp
#include "WorldObject.h"

WorldObject::WorldObject(Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String
meshName, Ogre::Vector3 worldLocation, Ogre::Quaternion worldRotation, Ogre::Vector3
```

```cpp
objectScale, std::list<Ogre::Sphere *> sphereList) : Object(name, sceneMgr, meshName)
{
       mMainNode->setPosition(worldLocation);        ///Sets the world location of the
Node
       mMainNode->setOrientation(worldRotation);      ///Sets the world rotation of the
Node
       mMainNode->setScale(objectScale);             ///Sets the scale of the Node
       mMainNode->attachObject(mEntity);             ///Attahes the object to its
main-node

       SphereCollider *s = new SphereCollider(false, Ogre::Sphere(Ogre::Vector3(0, 0,
0), mEntity->getBoundingRadius() * 0.75 * mMainNode->getScale().z));
       ///Initialises the sphere collision with a scale of 0.75 times the Z scale, a
scale of 0.75 was the most realistic size for current object scaling
       s->setPositionToParentPosition(mMainNode->getPosition());    ///Sets the
Spherecollider position to the center of the main node
       physicsMat->bounciness = 0.1;      ///Sets a bounce effect for when the ship
collides with the collider, knocking it back
       sphereColliders.push_back(s);      ///Adds the collider to the collision list
}

void WorldObject::Update(Ogre::Real elapsedTime, OIS::Keyboard * input)
{
       Object::Update(elapsedTime, input);
}

WorldObject::~WorldObject()
{
       mMainNode->detachAllObjects();
       delete mEntity;
       mMainNode->removeAndDestroyAllChildren();
       mSceneMgr->destroySceneNode(mName);
}
```

Which in turn can then be implemented into the World

```cpp
#include "World_1.h"


World_1::World_1(Ogre::SceneManager *mSceneMgr, std::list<Object *> &objectList,
std::list<Powerup *> &powerUpList)
{
       /**
       *The objects in it are listed for their name, mesh, location, rotation, scale
and collision-type list.
       *This class therefore creates mesh objects in the world, including the start and
powerups
       */
       std::list<Ogre::Sphere *> sphereList;
       Start = new WorldObject("Start", mSceneMgr, "Start_Line", Ogre::Vector3(0, 13,
0), Ogre::Quaternion(1.0f, 0.0f, 1.0f, 0.0f), Ogre::Vector3(5, 5, 7), sphereList);

       Powerup1 = new Powerup("Powerup1", mSceneMgr, "PowerUp", Ogre::Vector3(210, 2,
280), Ogre::Quaternion(1.0f, 0.0f, 1.8f, 0.0f), Ogre::Vector3(3, 3, 3), sphereList);
       Powerup2 = new Powerup("Powerup2", mSceneMgr, "PowerUp", Ogre::Vector3(225, 2,
300), Ogre::Quaternion(1.0f, 0.0f, 1.8f, 0.0f), Ogre::Vector3(3, 3, 3), sphereList);
```

Tooling

Points: 6-8
Requirements: 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.

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.

Our group uses GitHub to maintain version control of our Ogre3D product, inside GitHub we've created multiple sub-branches to develop the game, using feature branches instead of developer branches, this gives a better overview of game functions being made and we only have 3 developers now, meaning the branches would be very broad otherwise.

After a part of the branch successfully works it can get approved for merging it to the master branch project.
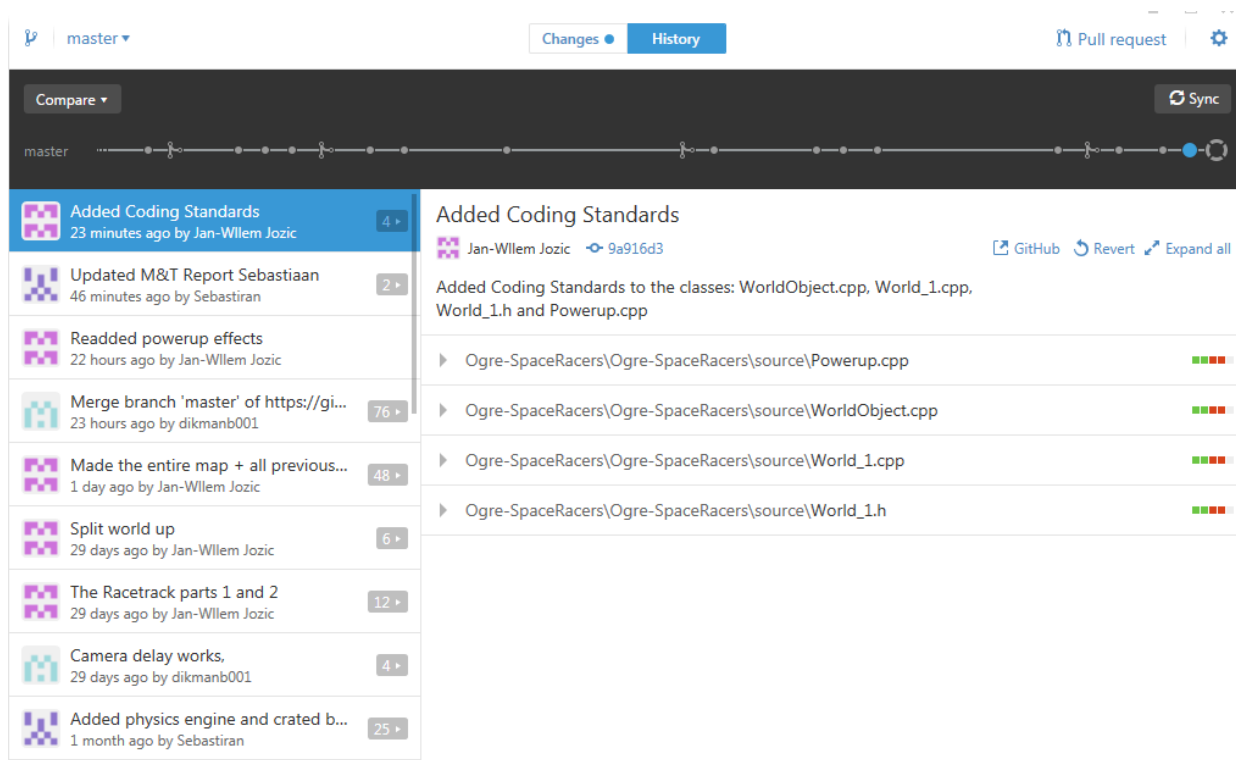
In addition to using Git our group utilizes tools for creating assets for our game. the 3D modeling program Blender is primarily used for our modeling.
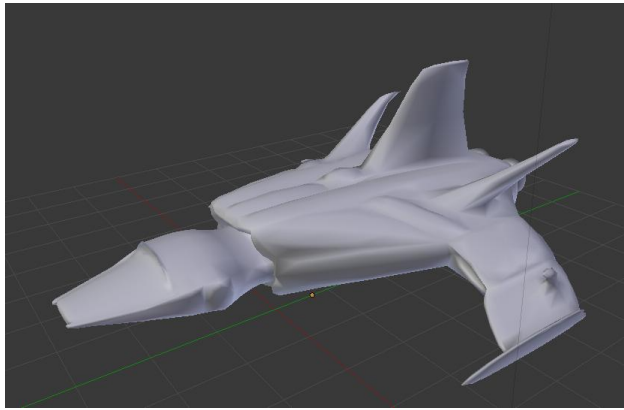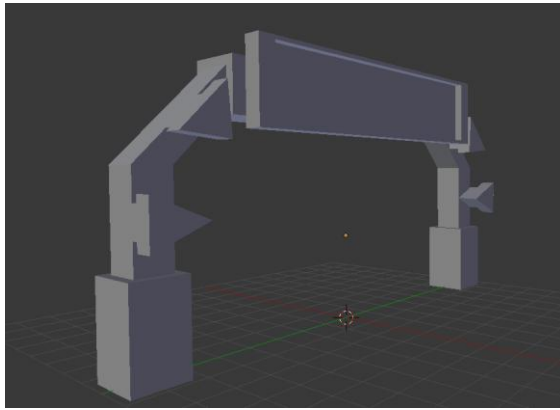
To easily convert the blender files we have installed an Ogre Mesh Converter that can transform the meshes created with blender into a .mesh format that Ogre3D can instantly use.

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	

Coding Standards

Points: 9-10
Requirements: 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.

The Documented coding standards:

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.

Examples of my application of coding standards

Example 1

```
/**
* @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"


World_1::World_1(Ogre::SceneManager *mSceneMgr, std::list<Object *> &objectList,
```

```
std::list<Powerup *> &powerUpList)
{
      /**
      *The objects in it are listed for their name, mesh, location, rotation, scale
and collision-type list.
      *This class therefore creates mesh objects in the world, including the start and
powerups
      */
```

Example 2

```
/**
* @class WorldObject
* @author Explosive Shark Studios
* @date 12/04/2016
* @brief
*
* @section Description
* This class contains the input function to spawn world objects.
*
*/
#include "WorldObject.h"

WorldObject::WorldObject(Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String
meshName, Ogre::Vector3 worldLocation, Ogre::Quaternion worldRotation, Ogre::Vector3
objectScale, std::list<Ogre::Sphere *> sphereList) : Object(name, sceneMgr, meshName)
{
      mMainNode->setPosition(worldLocation);          ///Sets the world location of the
Node
      mMainNode->setOrientation(worldRotation);       ///Sets the world rotation of the
Node
      mMainNode->setScale(objectScale);               ///Sets the scale of the Node
      mMainNode->attachObject(mEntity);               ///Attahes the object to its
main-node

      SphereCollider *s = new SphereCollider(false, Ogre::Sphere(Ogre::Vector3(0, 0,
0), mEntity->getBoundingRadius() * 0.75 * mMainNode->getScale().z));
      ///Initialises the sphere collision with a scale of 0.75 times the Z scale, a
scale of 0.75 was the most realistic size for current object scaling
      s->setPositionToParentPosition(mMainNode->getPosition());    ///Sets the
Spherecollider position to the center of the main node
      physicsMat->bounciness = 0.1;     ///Sets a bounce effect for when the ship
collides with the collider, knocking it back
      sphereColliders.push_back(s);     ///Adds the collider to the collision list
```

Example 3

```
/**
* @class Powerup
* @author Explosive Shark Studios
* @date 12/04/2016
* @brief
*
* @section Description
* This class contains the input function to spawn world objects.
*
*/
#include "Powerup.h"

Powerup::Powerup(Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String
meshName, Ogre::Vector3 worldLocation, Ogre::Quaternion worldRotation, Ogre::Vector3
objectScale, std::list<Ogre::Sphere *> sphereList) : Object(name, sceneMgr, meshName)
{
```

```cpp
        mMainNode->setPosition(worldLocation);          ///Sets the world location of the
Node
        mMainNode->setOrientation(worldRotation);       ///Sets the world rotation of the
Node
        mMainNode->setScale(objectScale);               ///Sets the scale of the Node
        mMainNode->attachObject(mEntity);               ///Attahes the object to its
main-node

        SphereCollider *s = new SphereCollider(false, Ogre::Sphere(Ogre::Vector3(0, 0,
0), mEntity->getBoundingRadius() * 1));  ///Initialises the sphere collision with a
scale of 1
        s->trigger = true;   ///Sets the collision radius to act as a trigger-zone
instead of blocking objects
        s->setPositionToParentPosition(mMainNode->getPosition());    ///Sets the
Spherecollider position to the center of the main node
        sphereColliders.push_back(s);      ///Adds the collider to the collision list

        Inactive = false;    ///Initialises the powerup to be active and ready for being
picked up the moment it is created in the world
}

void Powerup::Update(Ogre::Real elapsedTime, OIS::Keyboard * input)
{
        Object::Update(elapsedTime, input);

        if (Inactive == true)
        {
                Cooldown--;          ///Whilst the object is inactive, lower the cooldown
timer at the update rate
                if (Cooldown <= 1)   ///When the cooldown reaches below 1 the powerup must
be made active again
                {
                        Inactive = false;    ///Re-activates the powerup so it can be picked
up again
                        mMainNode->attachObject(mEntity); ///Re-attaches the mesh and
collision objects to the node so they can make the object visible again and get picked
up again
                }

        }
}

void Powerup::SetInactive()
{
        Cooldown = 100;      ///Sets the time it takes until the powerup can be picked up
again
        Inactive = true;     ///Turns the powerup inactive so it can not be picked up
again during the cooldown time
        mMainNode->detachAllObjects();     ///Detaches the mesh and collision objects from
the node, making them disappear from view so it appears the object has been picked up
}
```
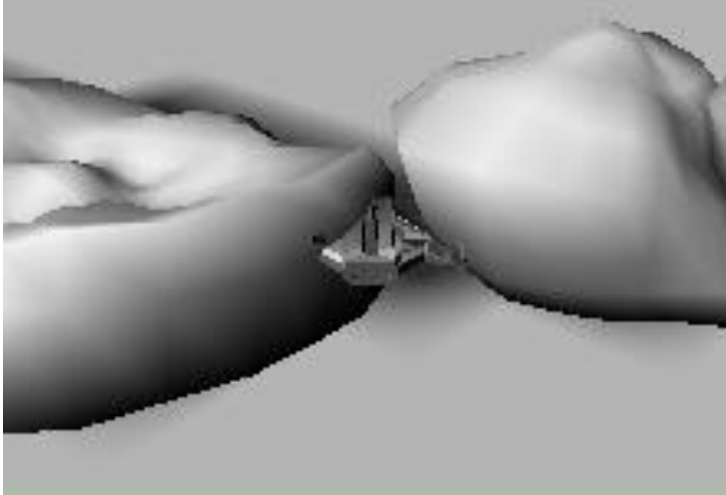
Testing

Points: 4-5
Requirements: Code is **tested**, test are **documented** in the report, but no testing framework (or otherwise consistent testing method) is used throughout the team

There have been several tests ran on making sure the collision on objects is realistic

Tests to see if the powerups can be picked up and disappear

Porting

Points: 0-2
Requirements: The game is not ported to another engine

For now there have been no attempts yet made to port the game into another engine

Documenting

Points: 9-10
Requirements: Tooling is used for automatic documentation generation

For our documentation, project member Bart Dikmans set up an automated system that takes comments that we write in our code and writes online documentation.

Heres several examples of the work:

Example 1

Example 2

Refactoring

Points: 6-8
Requires: Multiple good examples of refactoring are shown, each applicable to the situation, and motivated in the report

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

Object.h

```cpp
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.

```cpp
// 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.

```cpp
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);
```

```cpp
World_1.cpp  ⇄ ✕  TutorialApplication.cpp
(Global Scope)
#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

```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.

```cpp
World_1.cpp    TutorialApplication.cpp
(Global Scope)

#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.

Another refactoring took place after the world was split up into single objects which needed custom rotations as well as collisions per object.

```cpp
World_1::World_1(Ogre::SceneManager *mSceneMgr, std::list<Object *> &objectList,
std::list<Powerup *> &powerUpList)
{
        /**
        *The objects in it are listed for their name, mesh, location, rotation, scale
and collision-type list.
        *This class therefore creates mesh objects in the world, including the start and
powerups
        */
        std::list<Ogre::Sphere *> sphereList;
        Start = new WorldObject("Start", mSceneMgr, "Start_Line", Ogre::Vector3(0, 13,
0), Ogre::Quaternion(1.0f, 0.0f, 1.0f, 0.0f), Ogre::Vector3(5, 5, 7), sphereList);


        Powerup1 = new Powerup("Powerup1", mSceneMgr, "PowerUp", Ogre::Vector3(210, 2,
280), Ogre::Quaternion(1.0f, 0.0f, 1.8f, 0.0f), Ogre::Vector3(3, 3, 3), sphereList);

        Powerup2 = new Powerup("Powerup2", mSceneMgr, "PowerUp", Ogre::Vector3(225, 2,
300), Ogre::Quaternion(1.0f, 0.0f, 1.8f, 0.0f), Ogre::Vector3(3, 3, 3), sphereList);
```

Advanced Techniques

Points: 0-2
Requirements: Report contains **no** record of using advanced programming techniques, such as artificial intelligence, procedural content generation, optimization using advanced datastructures (e.g. hash tables, redblack trees), etc.