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	**:**	**25-5-2016**
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
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

```
#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);
```

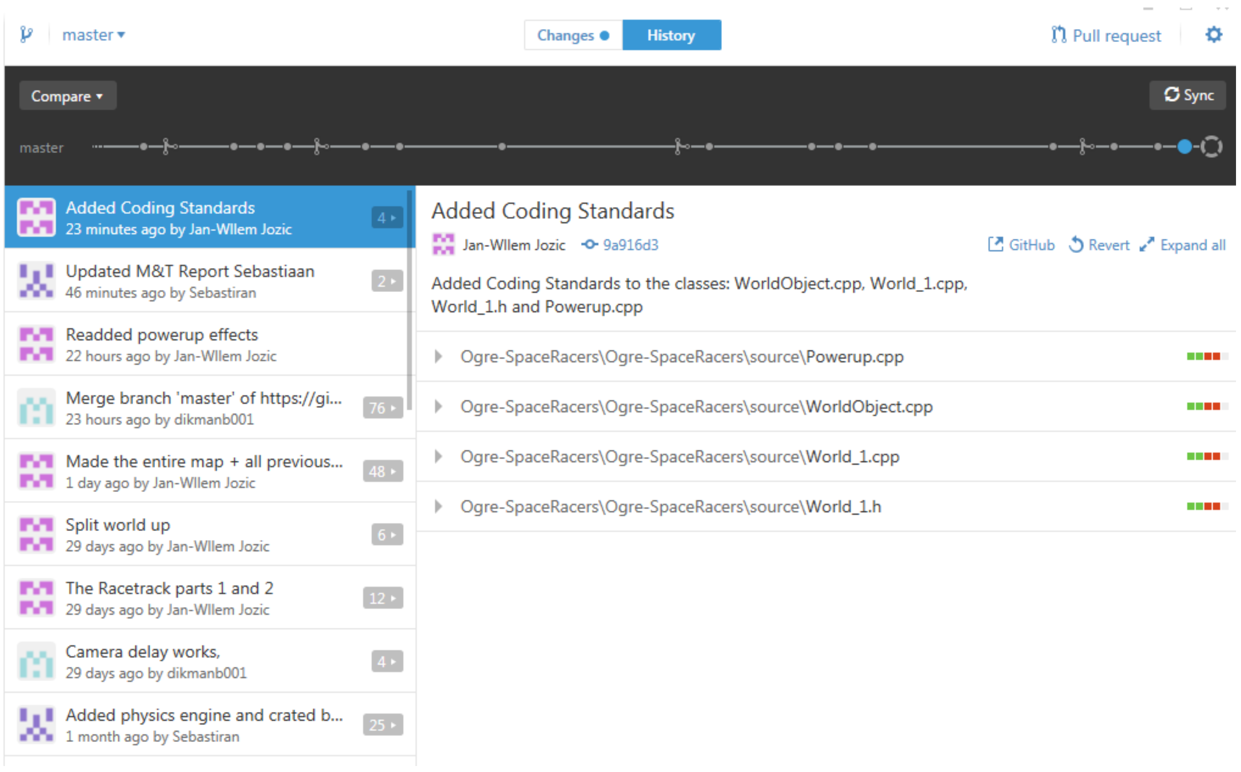
Tooling

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.



master ▾ | Changes ● | History | Pull request

Compare ▾ | Sync

master

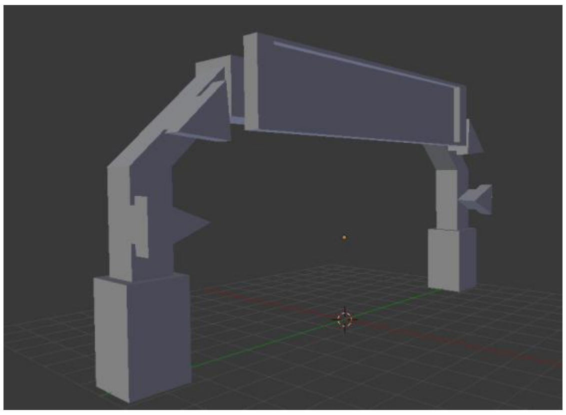
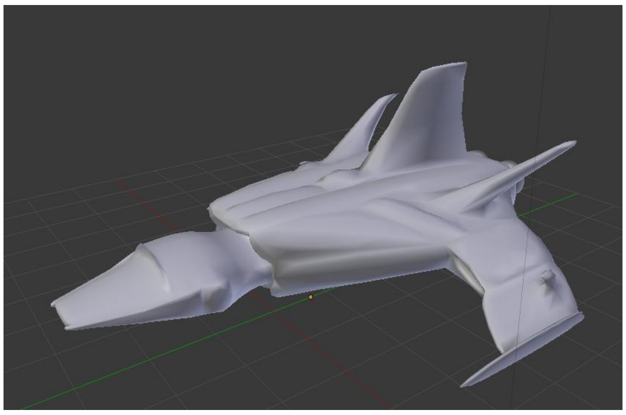
Added Coding Standards — 23 minutes ago by Jan-Wllem Jozic — 4
Updated M&T Report Sebastiaan — 46 minutes ago by Sebastiran — 2
Readded powerup effects — 22 hours ago by Jan-Wllem Jozic
Merge branch 'master' of https://gi... — 23 hours ago by dikmanb001 — 76
Made the entire map + all previous... — 1 day ago by Jan-Wllem Jozic — 48
Split world up — 29 days ago by Jan-Wllem Jozic — 6
The Racetrack parts 1 and 2 — 29 days ago by Jan-Wllem Jozic — 12
Camera delay works, — 29 days ago by dikmanb001 — 4
Added physics engine and crated b... — 1 month ago by Sebastiran — 25

Added Coding Standards
Jan-Wllem Jozic 9a916d3 | GitHub | Revert | Expand all

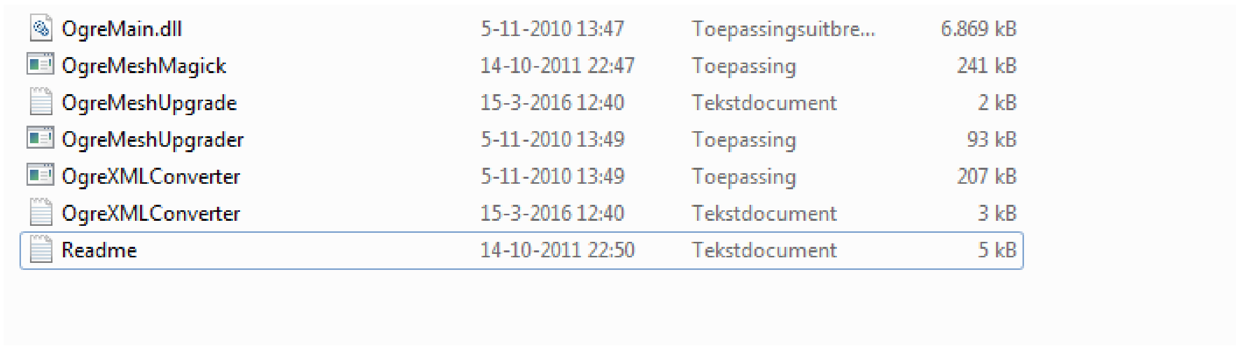
Added Coding Standards to the classes: WorldObject.cpp, World_1.cpp, World_1.h and Powerup.cpp

Ogre-SpaceRacers\Ogre-SpaceRacers\source\Powerup.cpp
Ogre-SpaceRacers\Ogre-SpaceRacers\source\WorldObject.cpp
Ogre-SpaceRacers\Ogre-SpaceRacers\source\World_1.cpp
Ogre-SpaceRacers\Ogre-SpaceRacers\source\World_1.h

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 brackets, we use the GNU statement, 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

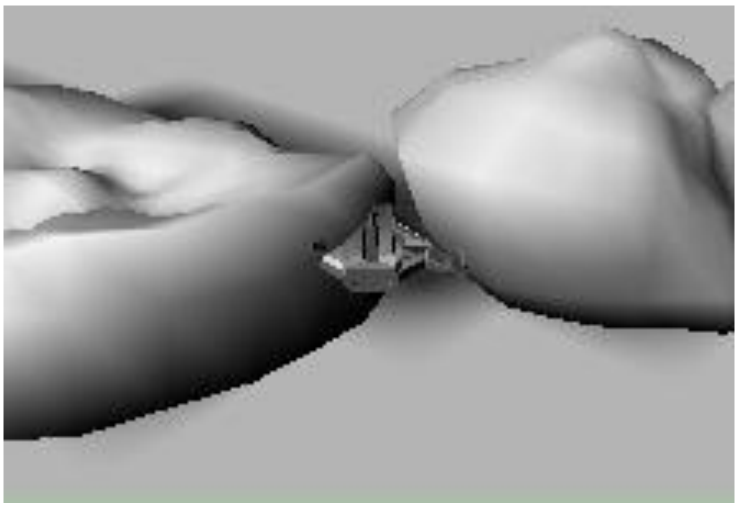
```
/**
* @class Powerup
* @author Explosive Shark Studios
* @date 12/04/2016
* @brief
*
* @section Description
* This class contains the input function to spawn world objects.
*
*/
#include "Powerup.h"

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)
{
	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
}
```

The coding standards are enforced by team member Bart Dikmans who runs the tool Resharper on the code at every build.

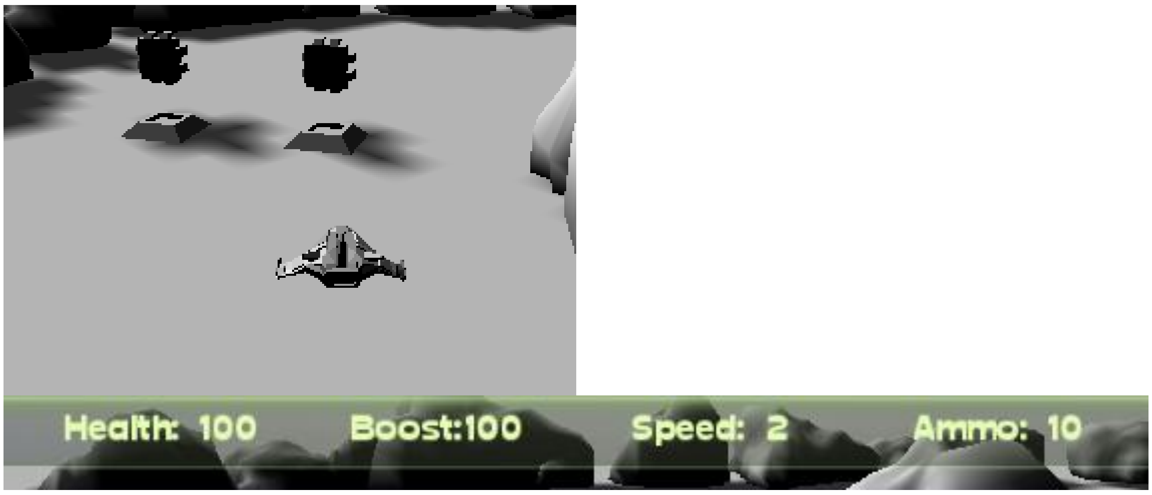
Testing

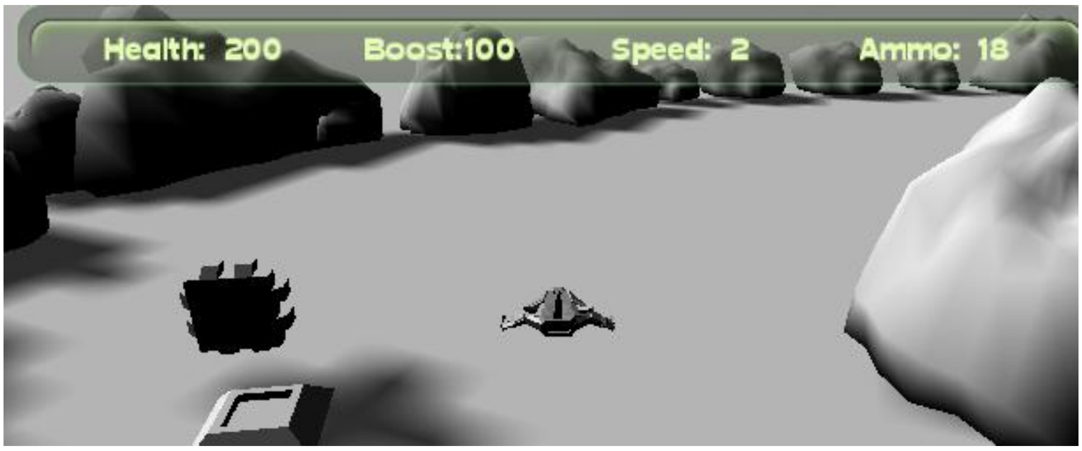
Points: 3-5
Requirements: Code is **tested**, test are **documented** in the report, but no testing framework (or otherwise consistent testing method) is used throughout the team

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: 3-5
Requirements: Parts (e.g. libraries) of the game are **ported** to another game engine, but not the entire game

The engine uses many functions specific for the Ogre engine, such as Ogre::String, Ogre::SceneManager and Ogre::Vector3.

This makes it difficult to port classes and libraries from Ogre to another engine without being forced to bring along the entire Ogre engine library.

Our game uses the Irrklang audio library, which can be ported to other engines such as the Irrlicht engine or any other engine that is compatible with C++, C# and .NET.

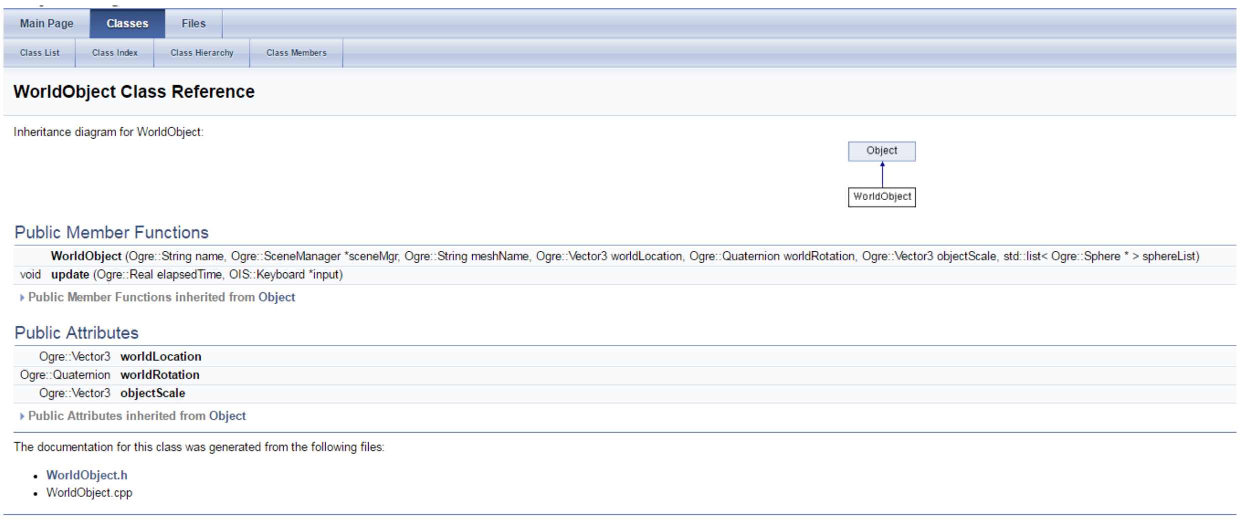
Documenting

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



Main Page | Classes | Files

Class List | Class Index | Class Hierarchy | Class Members

WorldObject Class Reference

Inheritance diagram for WorldObject:

Public Member Functions

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

▸ Public Member Functions inherited from Object

Public Attributes

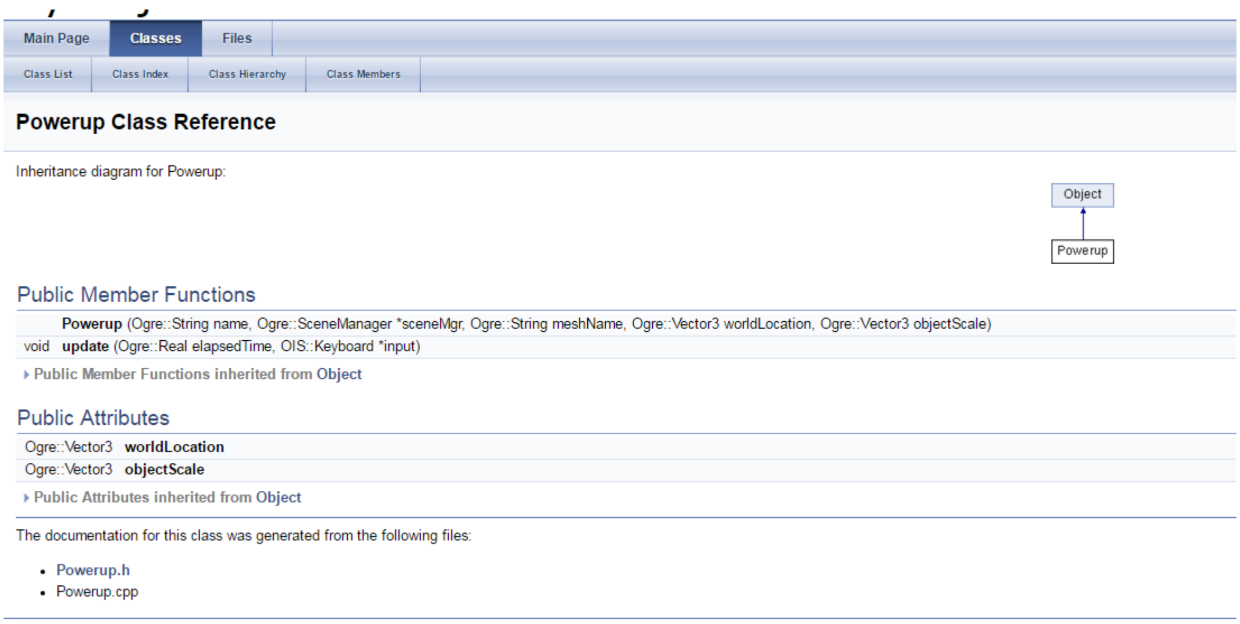
Ogre::Vector3	**worldLocation**
Ogre::Quaternion	**worldRotation**
Ogre::Vector3	**objectScale**

▸ Public Attributes inherited from Object

The documentation for this class was generated from the following files:

- WorldObject.h
- WorldObject.cpp

Example 2



Main Page | Classes | Files

Class List | Class Index | Class Hierarchy | Class Members

Powerup Class Reference

Inheritance diagram for Powerup:

Public Member Functions

	Powerup (Ogre::String name, Ogre::SceneManager *sceneMgr, Ogre::String meshName, Ogre::Vector3 worldLocation, Ogre::Vector3 objectScale)
void	**update** (Ogre::Real elapsedTime, OIS::Keyboard *input)

▸ Public Member Functions inherited from Object

Public Attributes

Ogre::Vector3	**worldLocation**
Ogre::Vector3	**objectScale**

▸ Public Attributes inherited from Object

The documentation for this class was generated from the following files:

- Powerup.h
- Powerup.cpp

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

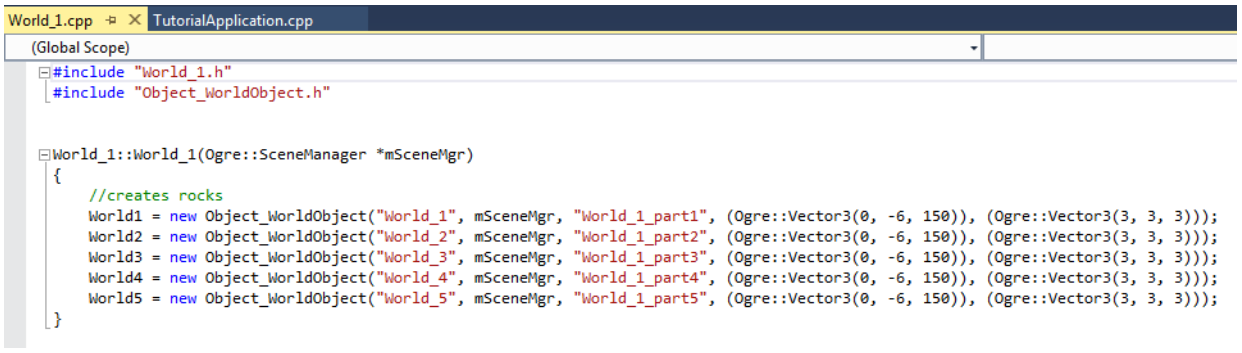
```
class Object
{
public:
    float inverseMass;

    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)));
}
```

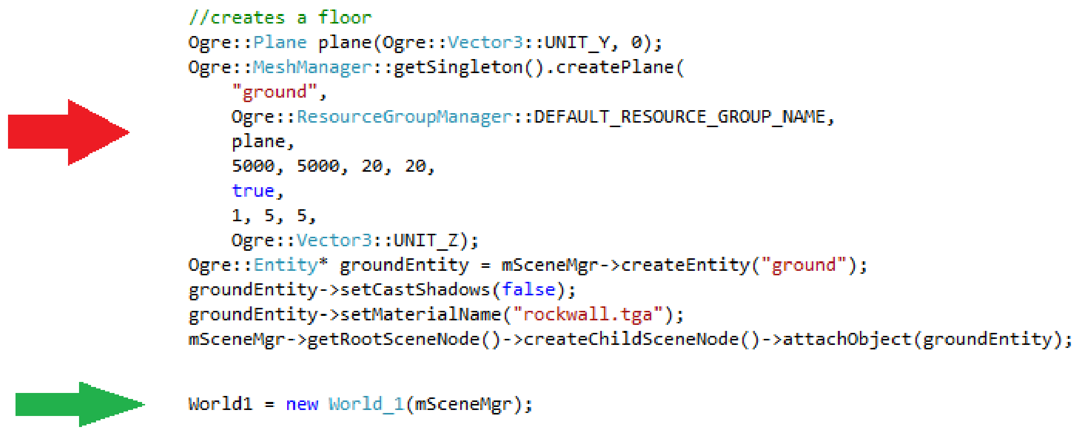
After that I made a similar refactoring to the way the world was generated. This was first done manually by adding objects in the TutorialApplication.cpp and defining them there.

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.

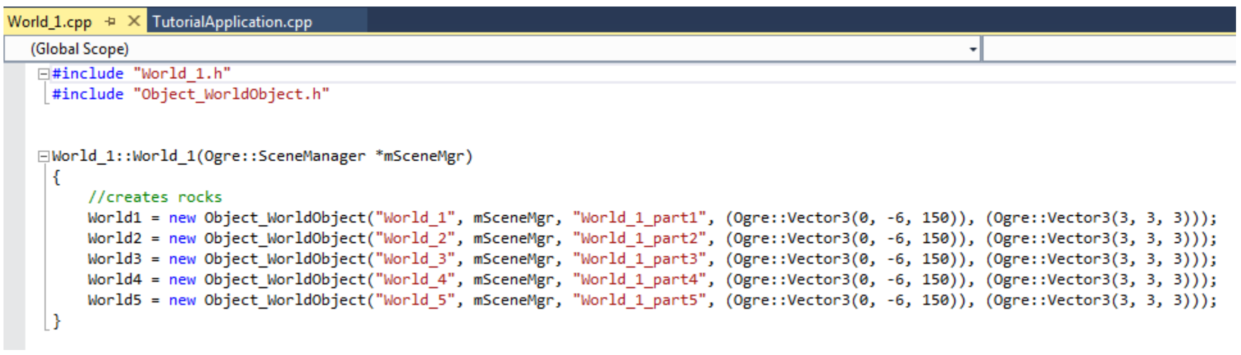
By doing this refactor there would only be one line of code needed in the TutorialApplication.cpp keeping it clean.

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.

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

```
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: 3-5
Requirements: Student can show that he/she **attempted** one or more advanced techniques, but failed; some **documentation** on the technique used is found in the report

My advanced techniques are very basic Artificial Intelligence attempts and particles.

However due to time constraints we haven't been able to implement a lot of advanced techniques, our team was halved by people being forced to leave and our production dropped considerably as a result.

Artificial Intelligence

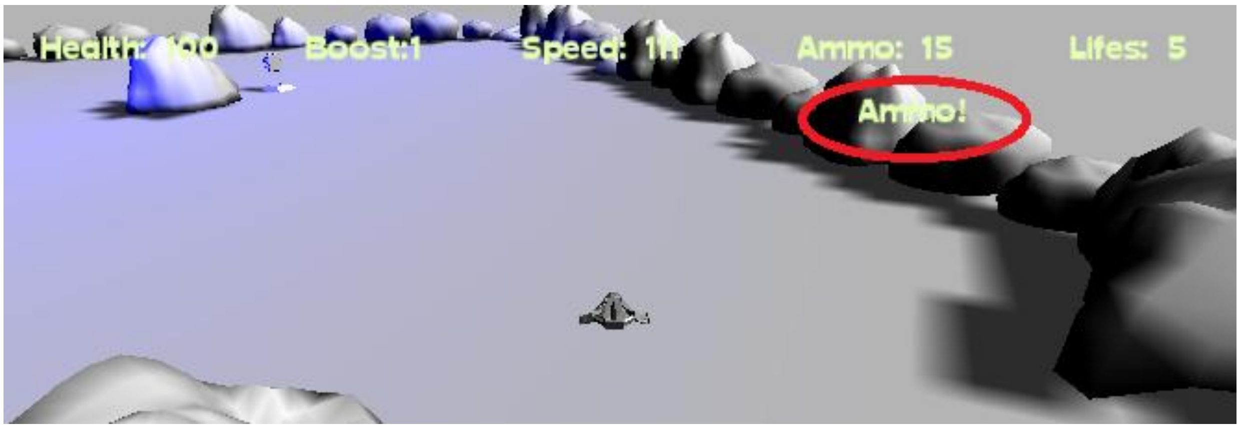
In video games, artificial intelligence is used to generate intelligent behaviors primarily in non-player characters (NPCs), often simulating human-like intelligence. The techniques used typically draw upon existing methods from the field of AI. However, the term game AI is often used to refer to a broad set of algorithms that also include techniques from control theory, robotics, computer graphics and computer science in general.

Power ups

An extremely simple and basic form of A.I. was built in the power ups, using the collision code written by team member Sebastiaan van Dijk, the ship will check collision with other objects. If the other object contains the tag "Powerup" the ship will generate a random number between 0 and 100 and will give a random power up based on the value of the random number.

Although this could hardly be considered an Artificial Intelligence, the code does evaluate between the types of collision it receives if it contains a specific tag, thus there is a very basic form of reasoning.

```
if (col.mTag == "Powerup")
       {
              int v1 = rand() % 100;

              if (v1 <= 24)
              {
                     mBoost += 100;
              }

              if (v1 >= 25 && v1 <= 49)
              {
                     mAmmo += 10;
              }

              if (v1 >= 50 && v1 <= 74)
              {
                     mShipHealth += 100;
              }
              if (v1 >= 75)
              {
                     shield = true;
              }
       }
```



Smart Mines

There was also an idea of smart-mines, proximity bombs that would seek out enemy players.

The idea behind it would be that as players have the tag of their ship number (Ship1 and Ship2) the bomb would be reminded not to attack the ship that placed it, but instead only seek out the enemy ship once it gets near you.

The principles behind the smart mines would be that it contains 2 collision spheres. One with a big radius and one with a small one.

The big sphere would be used as a proximity detector and checker if the ship that enters is an enemy or an ally.

The ships contain tags that give their own name, so if a collision takes place between a ship and the mine the ship would notify the mine if its Ship1 or Ship2. The mine would contain a function upon being spawned that it knows which ship created it, Ship1 or Ship2.

If the ship that enters its proximity isn't the same that created it, the mine will know it is an enemy ship and will proceed to chase the enemy ship at a set speed until one of two conditions happen.

Either the ship will escape the mine by leaving the radius of the bigger collision sphere again, therefore deactivating the tracking. Or the mine will get close enough that the ship enters the radius of the smaller collision sphere.

The smaller collision sphere will then trigger the mine to explode and cause damage to the nearby ship.

This is a basic A.I. which several states and a goal it seeks to complete.

But this was never implemented due to time constraints.

Particles

A particle system is a technique in game physics, motion graphics and computer graphics that uses a large number of very small sprites, 3D models, or other graphic objects to simulate certain kinds of phenomena, which are otherwise very hard to reproduce with conventional rendering techniques.

Now although Ogre has a built in particle manager system, the use of it still requires knowledge about how particles work.

Particles aren't just defined by "I want a particle <BOOM> Particle works" it needs a series of parameters and definitions in order to work.

Particles need to be defined both in size, speed and color, as well as any extra functions like when they need to be visible or how they need to be presented and even if they need a specific shape (like a sprite) so that it's not specifically cubes or spherical in appearance.

In addition to the particle definition itself, the particle needs a location in which it can spawn, which can also be written out. for example a specific boundary in which particles can spawn, or roam.

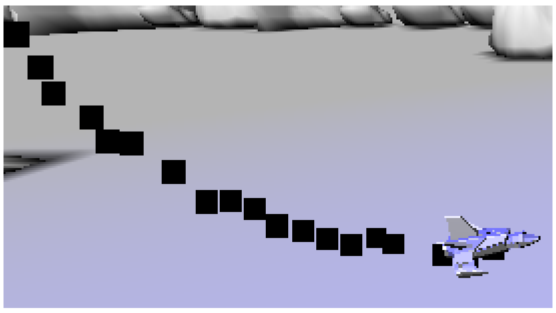
Below is an example of a particle system we use in our game:

```
particle_system Examples/ShipSmoke
{
	particle_width		2
	particle_height		2
	cull_each			false
	cull_each			false
	quota				5000
	billboard_type		point

	 // Area emitter
	emitter Box
	{
		angle				30
		emission_rate		30
		time_to_live		1
		direction			0 1 0
		velocity			0
		colour_range_start	0 0 0
		colour_range_end	0.3 0.3 0.3
		width				2
		height				1
		depth				2
	}
	// Make them float upwards axis X Y Z
	affector LinearForce
	{
		force_vector		0 100 0
		force_application	add
	}

	// Fader
	affector ColourFader
	{
        }
}
```

Implementing a particle system will create an entity that can also be attached to object nodes in Ogre

```
ps = mSceneMgr->createParticleSystem("Particle" + mName, "Examples/ShipSmoke");
        mShipNode->attachObject(ps);
```

The resulting particle effect will be like this:



In this case the particle doesn't have a sprite to represent it so it shows a square box.