James Moran

This report shows certain parts of the development process for the Seek-and-Collect DirectX 11 application.

James Moran CGP600 AE2 – Individual Project Report

Contents

[Introduction 3](#_Toc503203561)

[Seek-and-Collect Overview 3](#_Toc503203562)

[Seek-and-Collect Features 3](#_Toc503203563)

[Initial Development Actions 3](#_Toc503203564)

[Additions to Satisfy the Basic Requirements 4](#_Toc503203565)

[Requirement 2 Implementation 4](#_Toc503203566)

[Requirement 3 Implementation 4](#_Toc503203567)

[Requirement 4 Implementation 4](#_Toc503203568)

[Requirement 5 Implementation 5](#_Toc503203569)

[Requirement 6/7/8/9 Implementation 5](#_Toc503203570)

[Enhancements 6](#_Toc503203571)

[Scene-Manager 6](#_Toc503203572)

[Project Conclusion 6](#_Toc503203573)

[Appendix A: Figures 8](#_Toc503203574)

[Appendix B: Class Hierarchy, Class Diagrams and Other Diagrams 16](#_Toc503203575)

[Class Hierarchy 16](#_Toc503203576)

[Enemy Finite State Machine Diagram 16](#_Toc503203577)

[Appendix C: Testing 17](#_Toc503203578)

[Feature Testing 17](#_Toc503203579)

[Evidence for Feature Testing 18](#_Toc503203580)

[Feature Test 1 18](#_Toc503203581)

[Feature Test 2 19](#_Toc503203582)

[Feature Test 3 20](#_Toc503203583)

[Feature Test 4 21](#_Toc503203584)

[Feature Test 5 22](#_Toc503203585)

[Feature Test 6 23](#_Toc503203586)

[Feature Test 7 25](#_Toc503203587)

[Feature Test 8 26](#_Toc503203588)

[Feature Test 9 26](#_Toc503203589)

[Feature Test 10 26](#_Toc503203590)

[Basic Testing 27](#_Toc503203591)

[Evidence for Basic Testing 29](#_Toc503203592)

[Basic Test 1 29](#_Toc503203593)

[Basic Test 2 29](#_Toc503203594)

[Basic Test 3 29](#_Toc503203595)

[Basic Test 4 29](#_Toc503203596)

[Basic Test 5 29](#_Toc503203597)

[Basic Test 6 29](#_Toc503203598)

[Basic Test 7 29](#_Toc503203599)

[Basic Test 8 29](#_Toc503203600)

[References 30](#_Toc503203601)

# Introduction

## Seek-and-Collect Overview

In Seek-and-Collect, you are the pilot of a hover-tank, delegated with the task of collecting Energy- Capsules. You are to push aside or avoid obstacles in your way whilst you go about collecting these capsules. You are also to watch out for any ‘Black Goons’ vehicles you see, who will attempt to destroy your hover-tank by ramming into it with their hover-tanks of a similar appearance (but with black-metal plating instead of the metallic-colour of your hover-tank’s plating).

## Seek-and-Collect Features

* The Player is to be represented by a static-mesh (as shown in Fig. 4 of Appendix A), that can move forwards, backwards, leftwards and rightwards, whilst also being able to yaw their hover-tank leftwards or rightwards (by 45° for each key-press)
* The Player and Enemies are not able to go through Static-Rocks (as represented by the static-mesh shown in Fig. 14 of Appendix A), stopping outside the bounding-sphere cast around them
* The Player and Enemies can push Wooden-Barrels (as represented by the static-mesh shown in Fig. 15 of Appendix A), moving the barrels as per their direction of movement into them
* The Player can pick-up Energy-Capsules (as represented by the static-mesh shown in Fig. 16 of Appendix A), by moving over them (removing them from the game-scene). Whilst Enemies are to simply pass through them (not removing them from the game-scene). If the Player picks-up all Energy Capsules in the game-scene, they will win that level, causing the game to freeze for 3 seconds, after which, it is terminated
* Enemies are to attempt to move towards the Player (following a bee-line), to deal damage to their hover-tank by ramming it with their own (as represented by the static-mesh shown in Fig. 17 of Appendix A). If they ram the Player 10 times, they will cause the Player to lose, removing the Player’s representation for their perspective, before sleeping for 3 seconds, then terminating the application

## Initial Development Actions

I started off by using an older Tutorial project (Tutorial 08 Exercise 01), as a basis for this assignment’s project. I went through the code, refactoring into classes, with respects to which actions these methods/functions execute. I resolved one warning error, by adding $(WindowsSDK\_IncludePath) to the Include Directories section of VC++ Directories. This resolved the occurrence of these warning messages. (gradbot, 2012)

I resolved live objects not being cleaned-up (which is what these messages indicate), by following all the steps of the tutorial, that is available in Appendix A: Fig. 1. (Master Kenneth, 2014)

# Additions to Satisfy the Basic Requirements

After the initial development actions, came the process of adding to the project, to satisfy the basic requirements (listed on the assignment brief).

First off, came that of adding functionality to the project to load and draw .obj files (assets, exported from a 3D-Modeling package, such as Autodesk’s 3D Studio-Max). After this was put in place, a basic static-mesh for the Player’s hover-tank was put together, to move onto the next requirement.

## Player Movement

For this requirement, the Player is to be given the ability to move around the level. After attempting various methods to set-up, a third-person camera, that follows the Player (iedoc, 2015), I settled on a method for an ‘Arc-Camera’ that keeps distance with the Player and always faces them, no matter the direction the Player’s hover-tank is facing. (Allen Sherrod and Wendy Jones, 2012)

## Requirement 3 Implementation

With a third-person camera now moving with the Player as they move through the level, came that of fulfilling the third requirement: Creating static-meshes to represent the object classes in the game scene.

The images used as a basis for these object’s static-meshes, can be found in Appendix A: Fig. 4, 5, 9 and 11.

## Textures and Lighting

With static and mobile obstacles, as well as collectibles (Energy Capsules) now in place in the game scene came the implementation required to meet the fourth requirement: Adding textures to the objects.

I found images to be used for the textures of these, on Textures.com. These can be found under Fig. 7, 8, 10 and 12 of Appendix A. (Copyright © 2005-2017, Textures.com)

An implementation for the lighting system (for at least Ambient-Lighting) has been put in place, but I will discuss issues with this in the Project Conclusion.

## Collision

With basic obstacles in the scene, that have lighting, came the 5th basic-requirement. For this, I went about the implementation of a collision system for all GameObjects.

I implemented a basic bounding-sphere collision system. This allows for static-obstacles (rocks) to not be moved if a controlled-object collides with them (such as the Player’s hover-tank), along with the moveable-obstacles (wooden barrels) to be pushed by the Player if they collide with them. Finally, this allows the Player to collect the energy-capsules (collectable-objects), within the level, so that the Player can win the current level. I used a method found online to handle removal of collectable-objects from the scene-objects collection. (Georg Fritzsche, 2010)

# Project Conclusion

In conclusion of this project, although a playable game has been produced (that one can win or lose at), I deem it not ready for release to any platform, as per the feature-set of Seek-and-Collect, that is detailed in the ‘Seek-and-Collect Features’ sub-section of the ‘Introduction’ section.

Looking back at the key systems and their implementation that stand out comes first, how window-resizing is handled.

For this, I consulted the Microsoft Development Network (MSDN) for two articles on window-resizing. One for a general overview of the process (Windows Dev Centre, 2018) specific documentation on the IDXGISwapChain::ResizeBuffers() method. (DXGI Documentation, 2018)

The next key system implementation (to remove Energy-Capsules from the scene, when the Player moves over them with their hover-tank), is the utilisation of the ‘Erase-Remove Idiom’. (Georg Fritzsche, 2010)

This was chosen as it makes sure that the respective item is removed from the vector completely (the vector handles resizing of itself, to account for this change). Evidence of this implementation working as expected, is shown in Feature Test 6 of the ‘Evidence for Feature Testing’ section of Appendix C: Testing.

Looking back at issues that occurred though, raises 2 issues that I encountered whilst completing the project: Texture-Mapping and Lighting.

On the issue of Texture-Mapping, although I was able to apply textures to all objects, the textures have not been applied to the object in the intended manner (either tiled or matching up appropriately to the object). I tried to apply textures to meshes in 3DSMax, before exporting the models with the textures to use in the project, but to no avail. For the future, I would want to consider how to properly set-up a texture map for a certain object, before exporting it from 3DSMax.

Regarding the issue of lighting for objects, although I have attempted to implement (at least) ambient lighting for objects, this would not show-up over the textures it seems. I also tried to export the lighting normal-directions from 3DSMax (so that lighting would be present when the objects are used in a game-scene), but once again, this still resulted in lighting not being discernible on objects. For the future, I would want to find out how to set-up lighting for models put together in 3DSMax properly, before exporting them from 3DSMax for use. If not that, I would want to make sure the lighting system is correctly set-up in a DirectX 11 project, for any .OBJ files used in it.

# Appendix A: Figures

Figure 1: A quote from masterkenth.com, as an example of D3D11 (simple) Warning messages:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | D3D11 WARNING: Process is terminating. Using simple reporting. Please call ReportLiveObjects() at runtime for standard reporting. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING: Live Producer at 0x007464B0, Refcount: 4. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x007476F8, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A07580, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A07264, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A10ACC, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A11204, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A1295C, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A11944, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A11B60, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A120C4, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A133BC, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A1373C, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A14A2C, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A14CB4, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A153AC, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A16D7C, Refcount: 1. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A22B7C, Refcount: 1. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x0C17154C, Refcount: 1. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A26904, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A2A2CC, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x0C1687F4, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x0C16BE0C, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING:  Live Object at 0x01A1ADC4, Refcount: 0. [ STATE\_CREATION WARNING #0: UNKNOWN]  D3D11 WARNING: Live                         Object :     22 [ STATE\_CREATION WARNING #0: UNKNOWN]  DXGI WARNING: Live Producer at 0x006AE3A8, Refcount: 4. [ STATE\_CREATION WARNING #0: ]  DXGI WARNING:  Live Object at 0x006AEA38, Refcount: 2. [ STATE\_CREATION WARNING #0: ]  DXGI WARNING: Live                         Object :      1 [ STATE\_CREATION WARNING #0: ] |

(Master Kennth, 2014)

Figure 2: A top-down view of a Lego interpretation of the Millennium Falcon, used as a reference image to aid in the product of the static-mesh to represent the Player’s hover-tank:



(Mike Celestino, 2015)

Figure 3: A top-down render-view from 3DSMax, of the Player’s hover-tank reference image shown side-by-side with the produced hover-tank’s static-mesh:

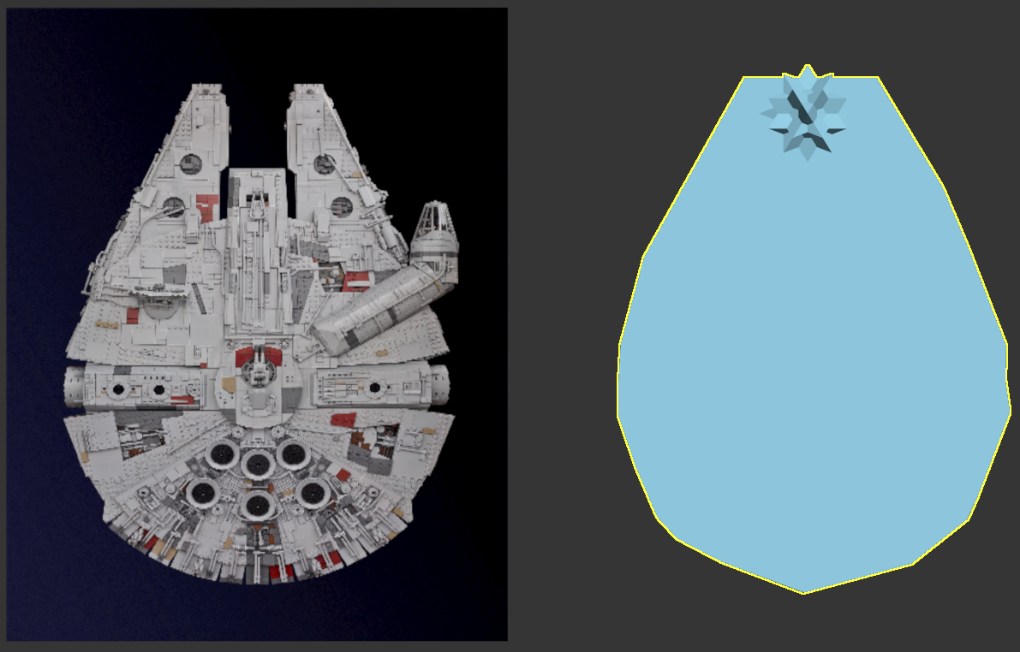


Figure 4: A perspective-view of the Player’s hover-tank’s static-mesh in the default game scene:

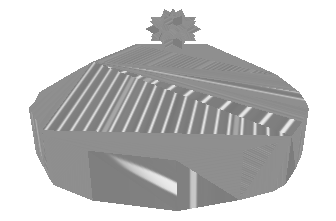


Figure 5: Side-on View of an igneous rock, used a reference image to produce static-obstacles in the default game scene:

(© 1970-2017 Analytical Scientific, LTD.)

Figure 6: The Player’s hover-tank with a few rocks, in the default game-scene:

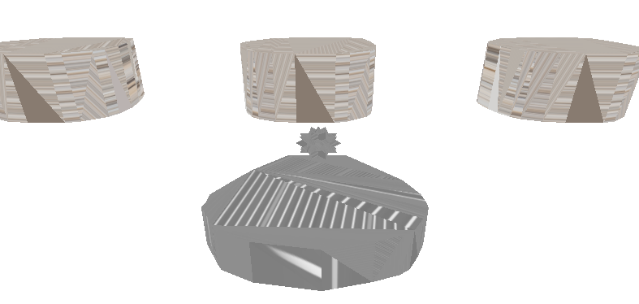


Figure 7: The image used to form a texture used for static-rocks in the default game-scene:

(Copyright © 2005-2017, Textures.com)

Figure 8: The image used to form a texture used for the Player’s hover-tank, to be applied to the Player’s hover-tank for all scenes of the game:



(Copyright © 2005-2017, Textures.com)

Figure 9: The reference image used to put together the default static-mesh to be used by moveable obstacles:



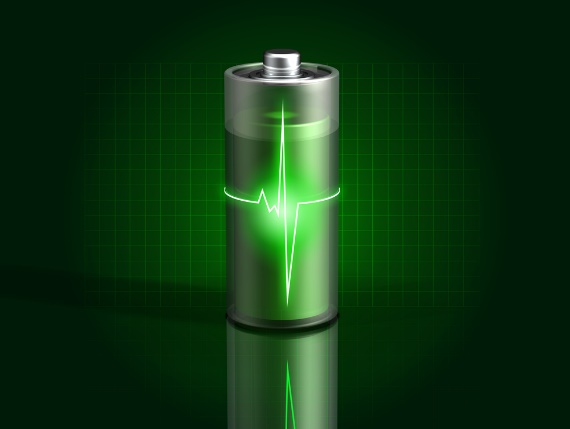
(Independent.co.uk, 2014)

Figure 10: The image used to form a texture to be used by the default moveable-obstacles (wooden barrels), in the default game-scene:



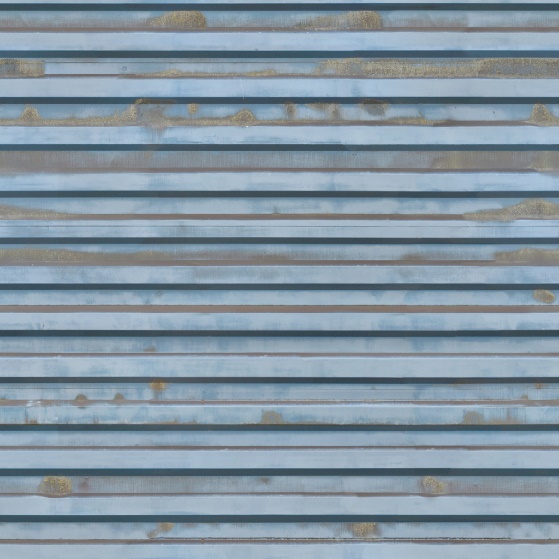
(Copyright © 2005-2017, Textures.com)

Figure 11: The image used a reference, for putting together the static-mesh to represent Energy Capsules (collectable-objects):

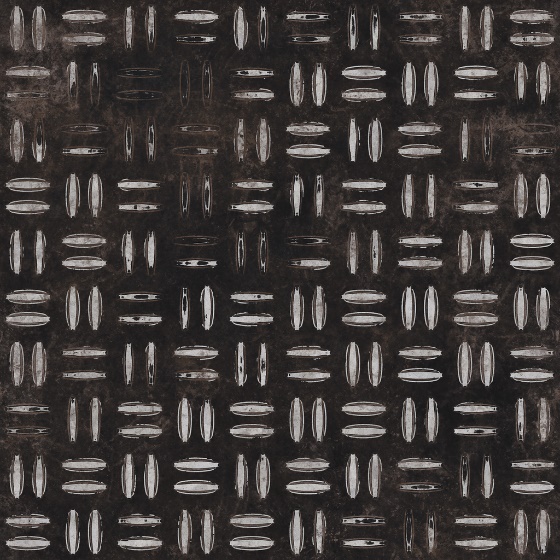


(9to5Google.com, 2013)

Figure 12: The image used to produce the Energy-Capsule’s default texture:



(Copyright © 2005-2017, Textures.com)

Figure 13: The image used to produce the Enemy hover-tank’s default texture:

(Copyright © 2005-2017, Textures.com)

Figure 14: Static-Rocks, immoveable by the Player or Enemies’ hover-tank:



Figure 15: Wooden-Barrels, moveable by the Player or Enemies’ hover-tank:



Figure 16: Energy-Capsules, the items that the Player is to collect to complete the level:

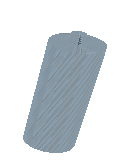
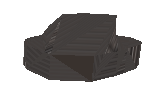


Figure 17: Enemy Hover-Tank Static-Mesh: To represent Enemy hover-tanks in the default game-scene:



# Appendix B: Class Hierarchy, Class Diagrams and Other Diagrams

## Class Hierarchy

The project’s class hierarchy:



## Enemy Finite State Machine Diagram

For the Enemy hover-tank’s AI:

# Appendix C: Testing

## Feature Testing

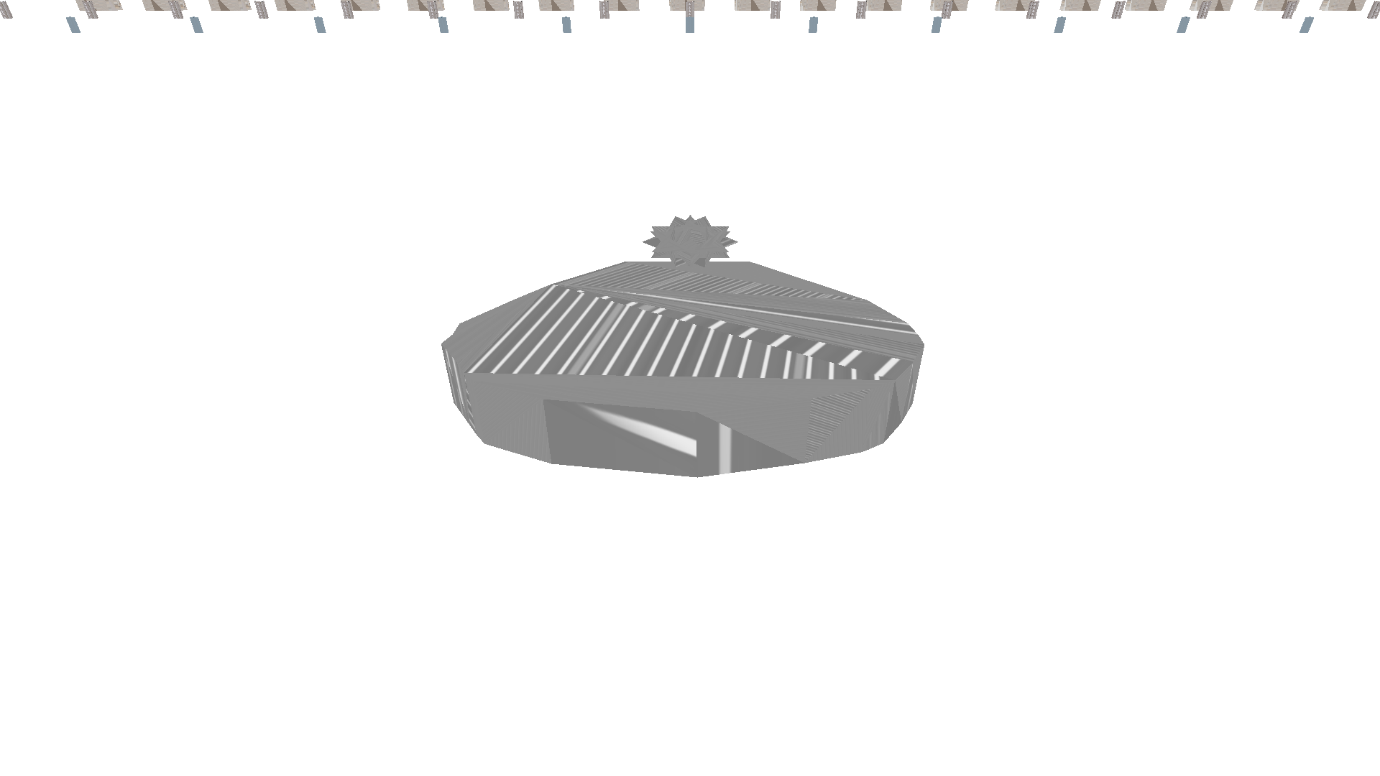
The tests of the features of Seek-and-Collect are documented here, as per the features outlined in the ‘Seek-and-Collect Features’ section of the introduction:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Test Description | Expected Result | Actual Result | Success or Failure |
| 1 | The Player’s static-mesh for their hover-tank is visible, as well as the other objects that are in their line-of-sight when the game is started. | The Player can see their hover-tank, as well as the Energy-Capsules, Wooden-Barrels and Static-Rocks of the level, when the game is initialised. | The Player can see their hover-tank, as well as the Energy-Capsules, Wooden-Barrels and Static-Rocks of the level, when the game is initialised. | Success. |
| 2 | The Player hover-tank’s movement is blocked by any Static-Rocks, disallowing their traversal through their bounds. | The Player’s hover-tank is not able to move through any given Static-Rock. | The Player’s hover-tank is not able to move through any given Static-Rock. | Success. |
| 3 | Enemy hover-tank’s movement is blocked by static-rocks, disallowing them to move through their bounds. | Enemy hover-tanks are not able to move through any given static-rock. | Enemy hover-tanks are not able to move through any given static-rock. | Success. |
| 4 | The Player can push Wooden-Barrels along with them, as per their movement direction. | Wooden-Barrels move along the Player hover-tank’s current direction of movement, when collided with. | Wooden-Barrels move along the Player hover-tank’s current direction of movement, when collided with. | Success. |
| 5 | Enemy hover-tanks can push Wooden-Barrels along with them, as per their movement direction. | Wooden-Barrels move along an Enemy hover-tank’s current direction of movement. | Wooden-Barrels move along an Enemy hover-tank’s current direction of movement. | Success. |
| 6 | The Player’s hover-tank can pick-up Energy-Capsules by colliding with them. When they collect all Energy-Capsules in the game-scene, they win the level. | The respective Energy-Capsule is removed from the game-scene when moved over by the Player.  When the Player has collected all Energy-Capsules, they win, causing the game to freeze for 3 seconds, before terminating. | The respective Energy-Capsule is removed from the game-scene when moved over by the Player.  When the Player has collected all Energy-Capsules, they win, causing the game to freeze for 3 seconds, before terminating. | Success. |
| 7 | Enemy hover-tanks are not able to pick-up Energy-Capsules by colliding with them. | The respective Energy Capsule is not removed from the game-scene when moved over by an Enemy hover-tank. | The respective Energy Capsule is not removed from the game-scene when moved over by an Enemy hover-tank. | Success. |
| 8 | Enemies are to attempt to seek out the Player, by moving towards them in a bee-line. | Enemies rotate to face the Player, before moving towards them in a bee-line, getting stopped by any Static-Rocks. | Enemies rotate to face the Player, before moving towards them in a bee-line, getting stopped by any Static-Rocks. | Success. |
| 9 | Enemies are to deal damage to the Player’s hover-tank by colliding with it. | After a collision, the Player’s hover-tank is dealt damage. | After a collision, the Player’s hover-tank is dealt damage | Success |
| 10 | If the Player receives 10 instances of collision from Enemy hover-tanks, they lose the game, with their hover-tank no longer being drawn to the game scene and the game freezing for 3 seconds before terminating. | The Player’s hover-tank’s static-mesh is no longer drawn and the game freezes for 3 seconds before terminating itself. | The Player’s hover-tank’s static-mesh is no longer drawn and the game freezes for 3 seconds before terminating itself. | Success |

## Evidence for Feature Testing

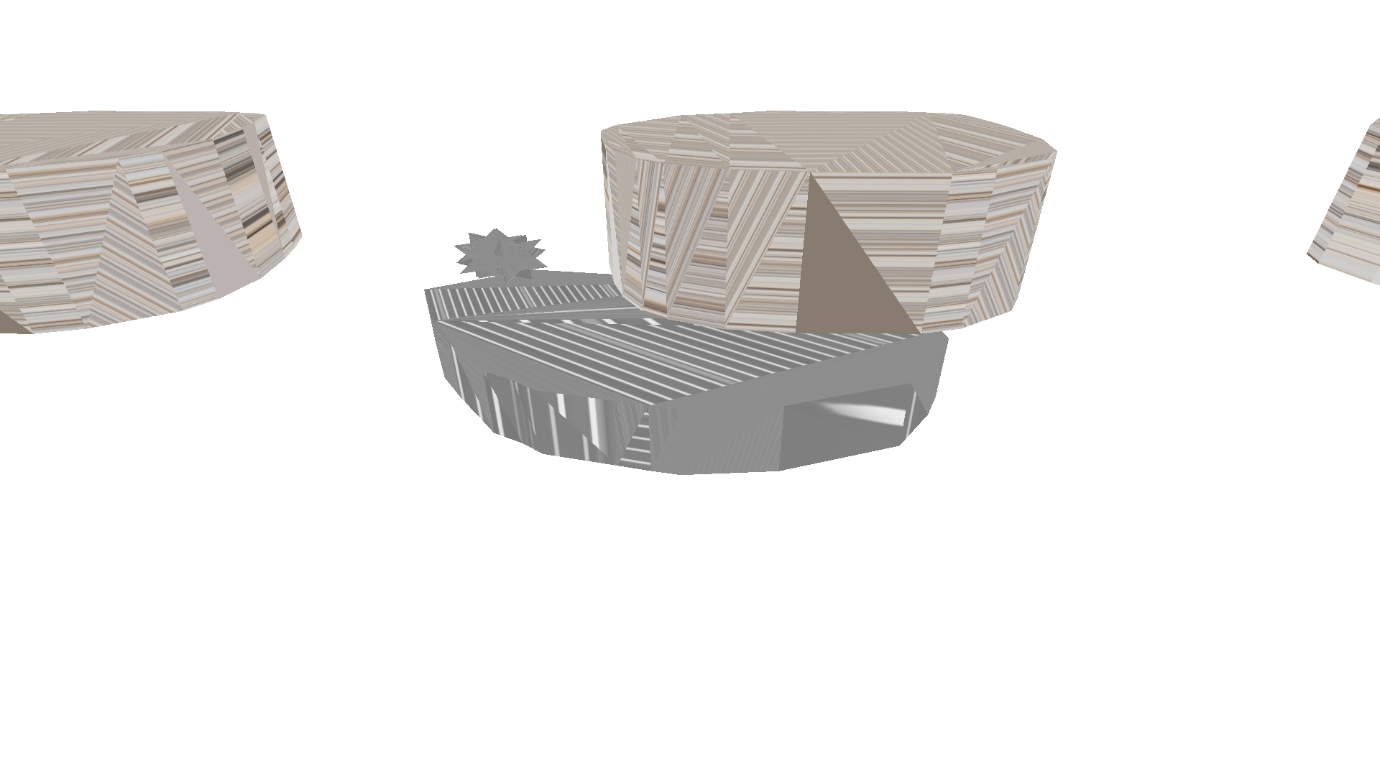
### Feature Test 1

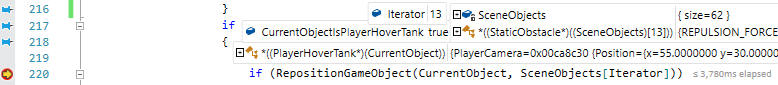
This test is simply for making sure that all the game-scene objects are shown to the Player as expected, this is the case, as the Player will see the following when the game is initialised:

Therefore, this test is successful.

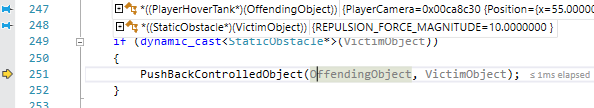
### Feature Test 2

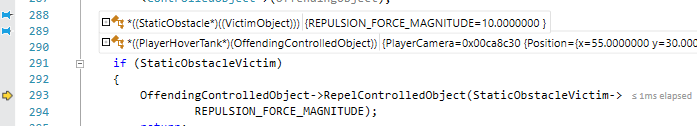
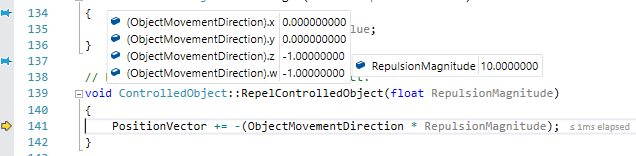
For this test, the Player’s hover-tank must be close enough to a Static-Rock, as shown here:



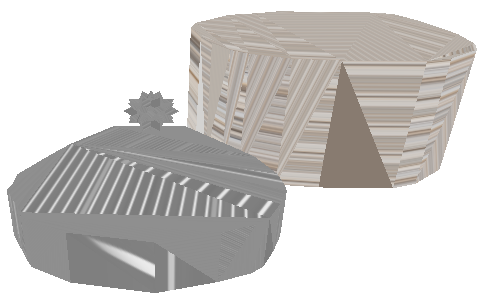
When they collide with the rock, the following breakpoint is triggered (within the GameScene’s ManageCollisionChecking() method):

From this, one can tell that the CurrentObject getting checked is the PlayerHoverTank, being checked against a GameObject at index 13 of the SceneObjects collection (that CurrentObject has collided with, after checking for bounding-sphere collision between it and this GameObject), which is a StaticRock. The program then traverses to the RepositionGameObject() method to handle repositioning CurrentObject appropriately:

In this function, the object receiving a collision (VictimObject) is a StaticObstacle whilst the object colliding with the VictimObject (OffendingObject) is a PlayerHoverTank, so in this situation, the PushBackControlledObject() method will be called:

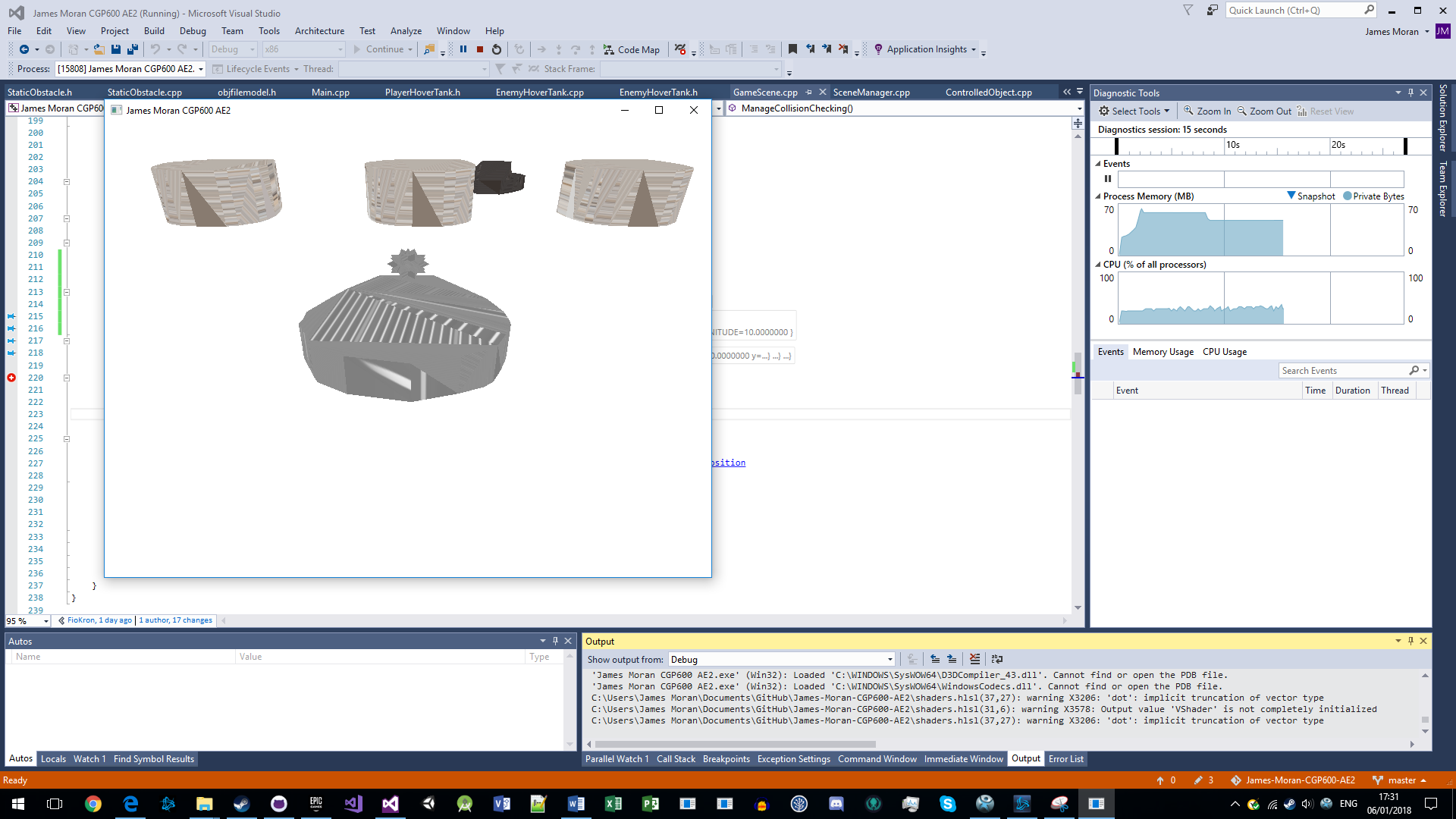
This method then makes sure which object-type is colliding with which other object type to perform the correct course of action, in this case, repel the Player’s hover-tank from this StaticObstacle via the RepelControlledObject() method:

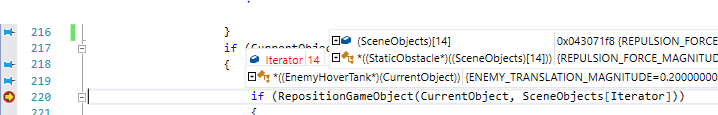
This method simply adds the (negated) movement direction of this object multiplied by the parsed-in RepulsionMagnitude, to cause the ControlledObject to bounce back appropriately, away from this StaticObstacle:

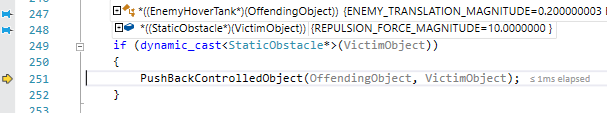
So therefore, this test is successful.

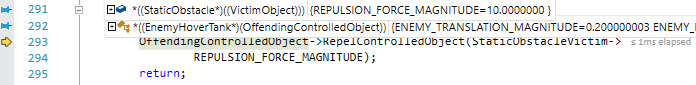
### Feature Test 3

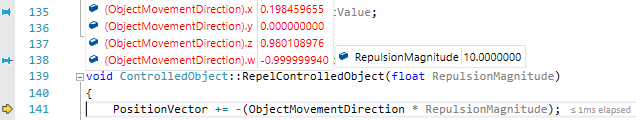
For this test, the Enemy’s hover-tank must be close enough to a StaticRock, as shown here:

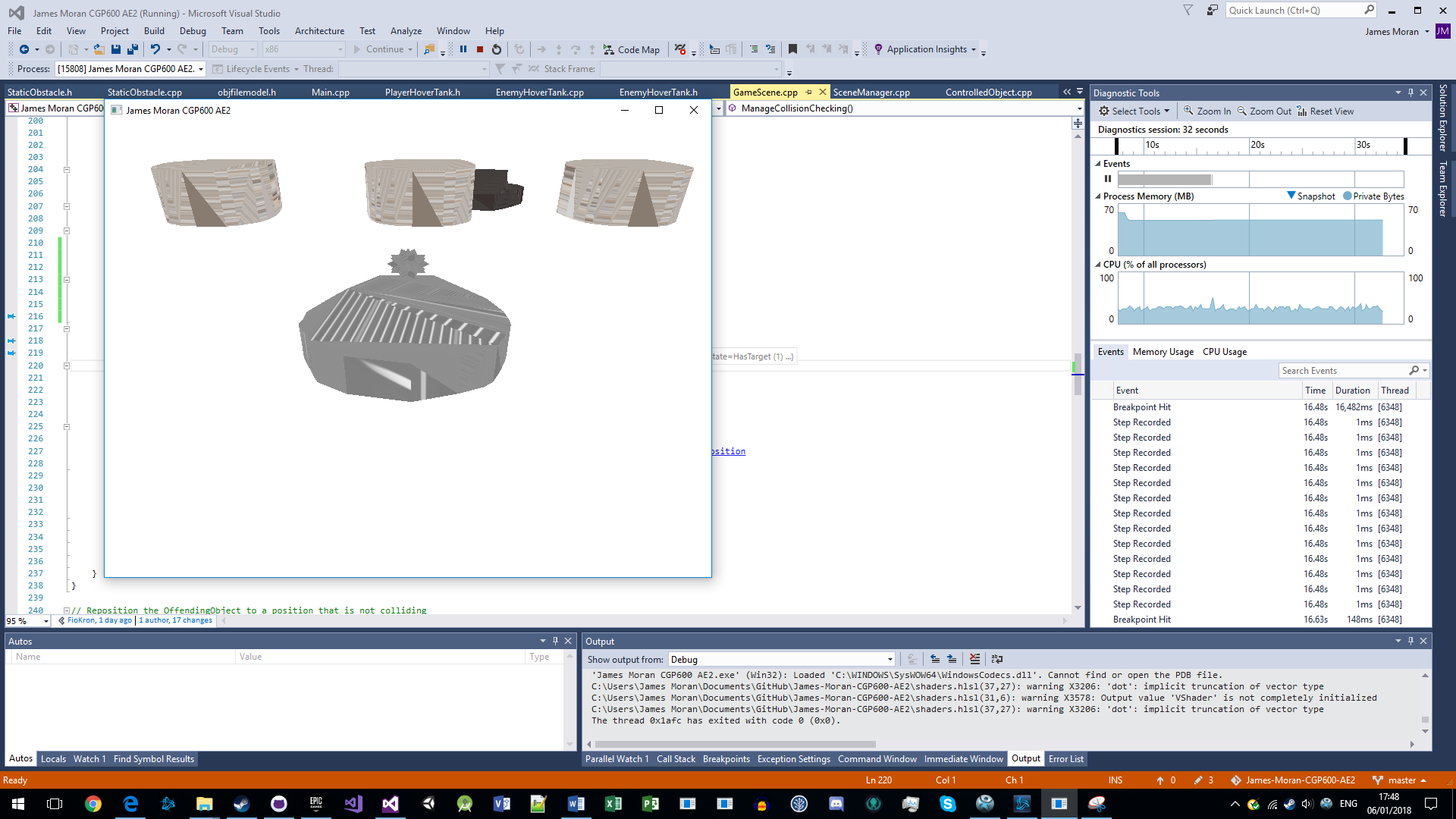
 When they collide with the rock, the breakpoint as shown on the next page is triggered:

So once again, RepositionGameObject() is called (with CurrentObject as an EnemyHoverTank this time):

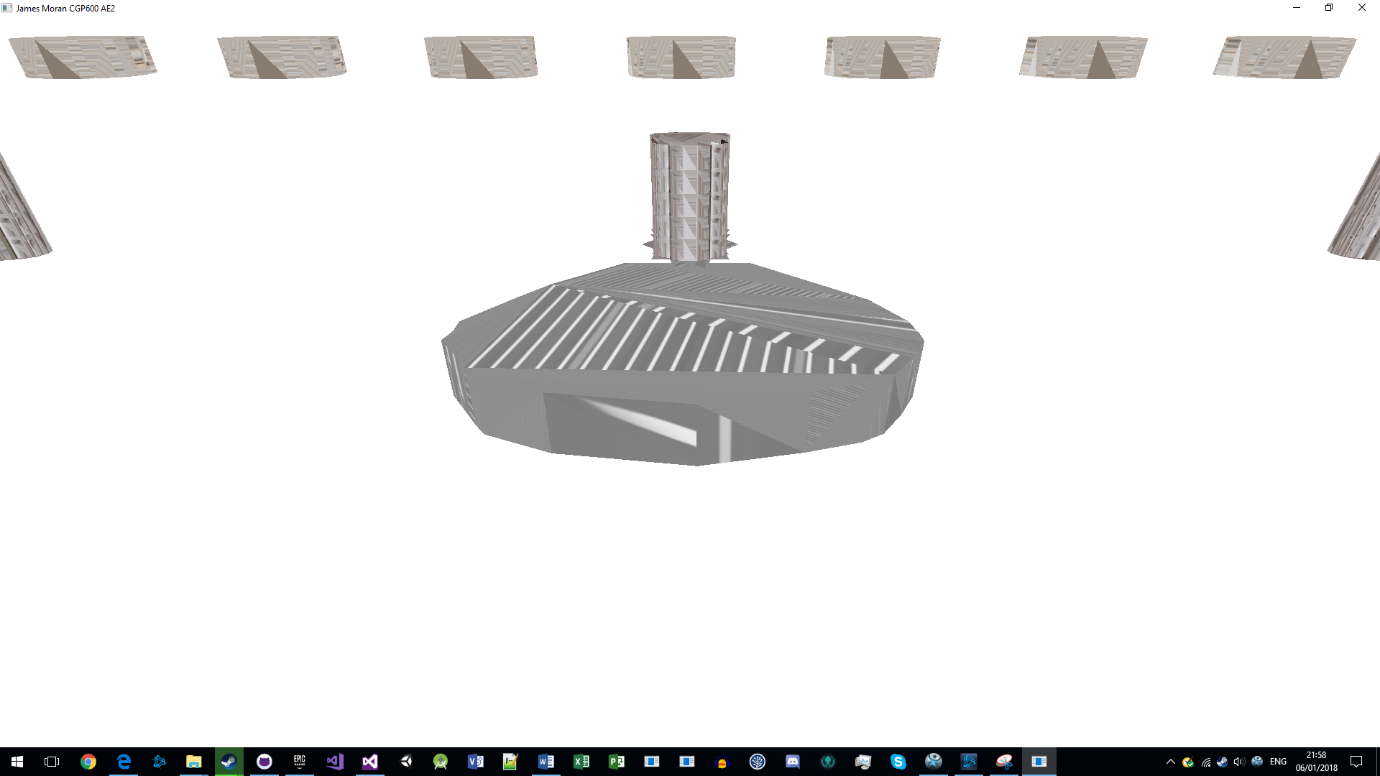
Which then calls the PushBackControlledObject() method:

Which then calls the RepelControlledObject() method (for the EnemyHoverTank):

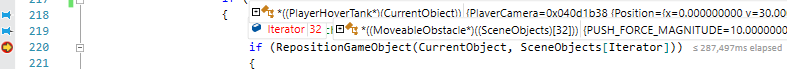
Which prevents the Enemy’s hover-tank from getting near to the Player through this StaticRock:

Therefore, this test is successful.

### Feature Test 4

For this test the Player’s hover-tank must be near a Wooden-Barrel:

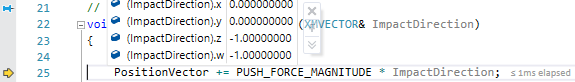
On the next page is the breakpoint triggered once again, in the GameScene’s ManageCollisionChecking() method:

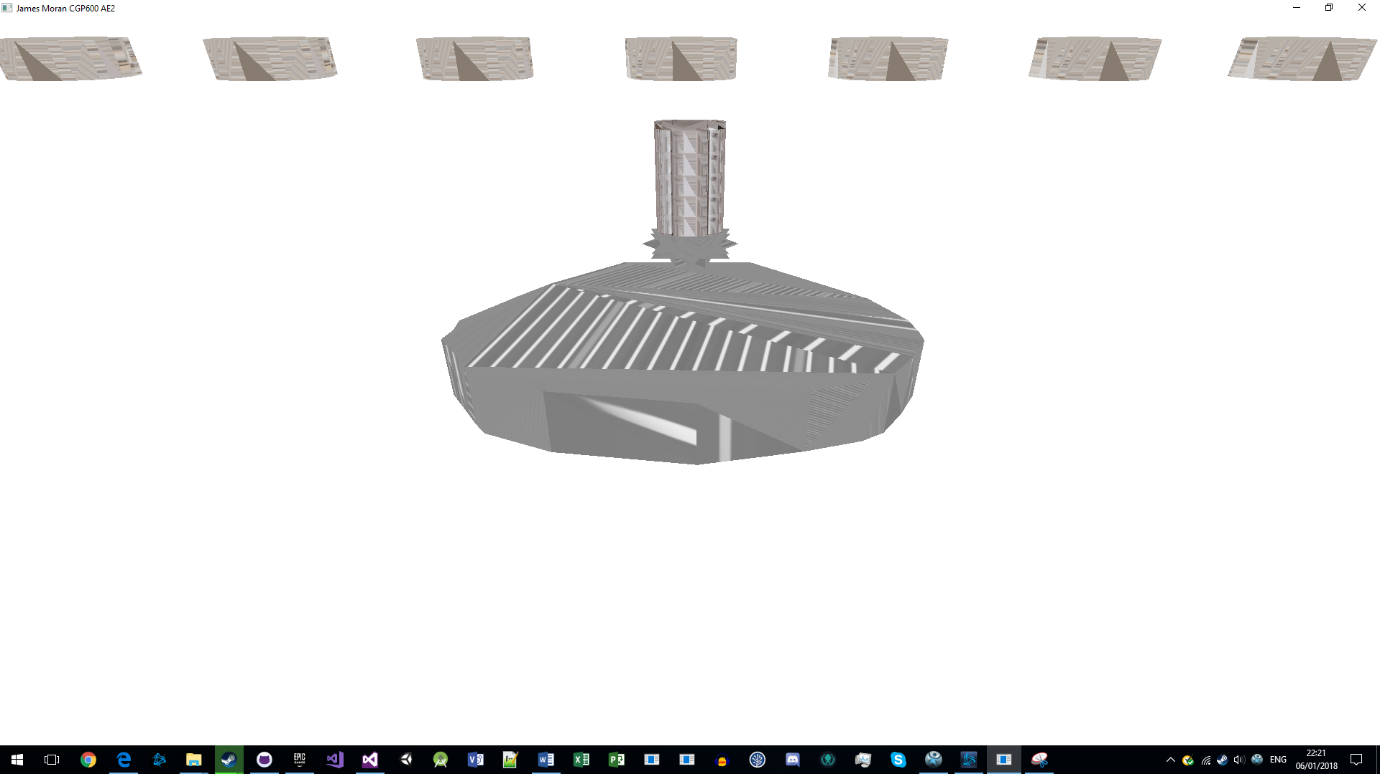
Which then calls RepositionObject():

Which then calls PushAwayMoveableObstacle (as the VictimObject in this case is a MoveableObstacle):



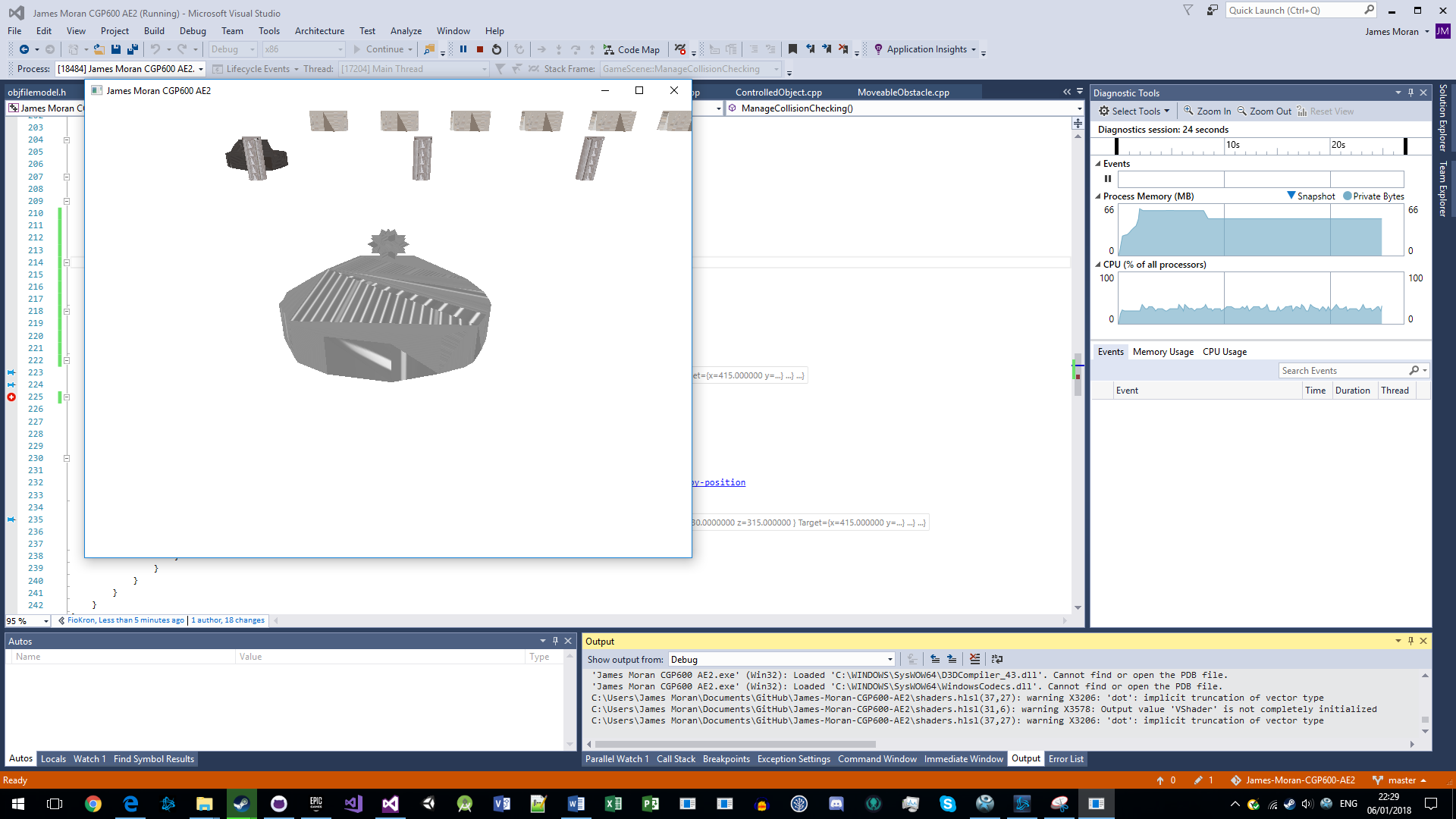
Which then calls the OnImpact() method of the MoveableObstacle (parsing in the ControlledObject’s MovementDirection):

Which results in the MoveableObstacle getting moved by the respective direction of impact:

This Wooden-Barrel is no longer in line with the other barrels, so this test is successful.

### Feature Test 5

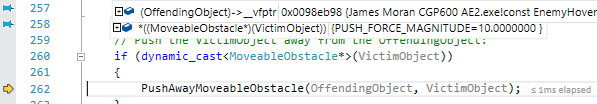
For this test, an EnemyHoverTank must be near to a MoveableObstacle:



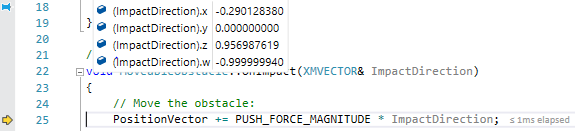
Then the same breakpoint (with different conditions) is triggered when they collide with this WoodenBarrel (as shown on the next page):

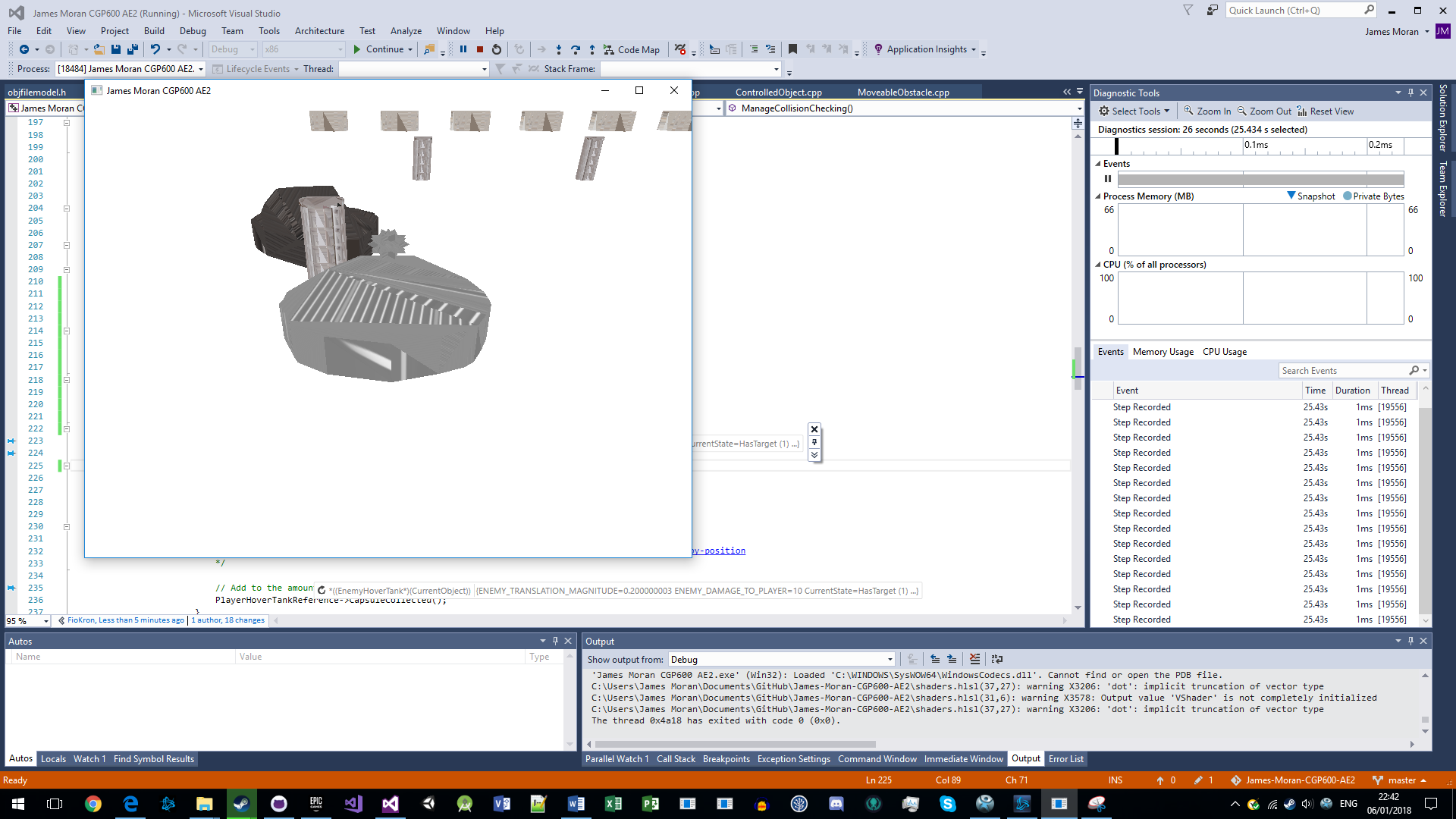


Which then calls RepositionObject():

Which then calls PushAwayMoveableObstacle():

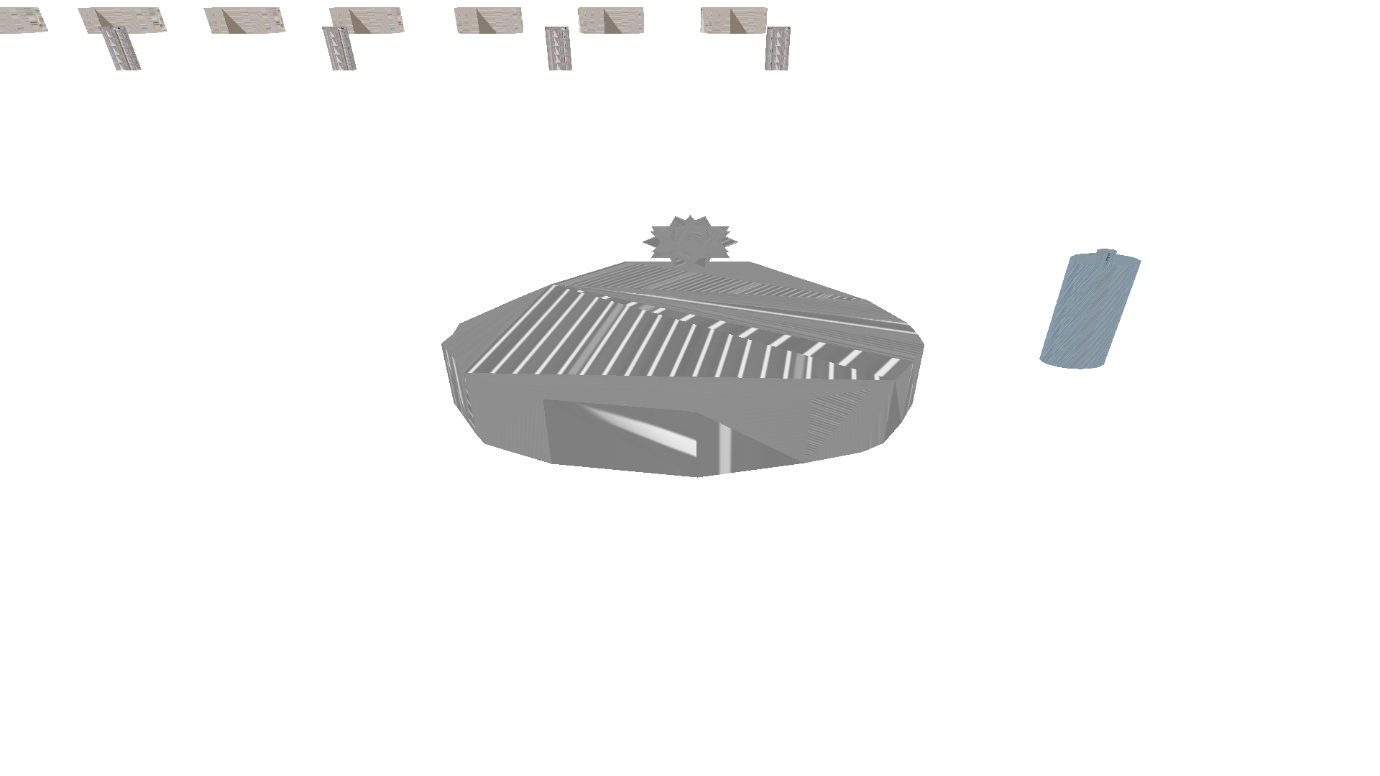
Which then calls OnImpact():

Which then results in movement of the Wooden-Barrel:

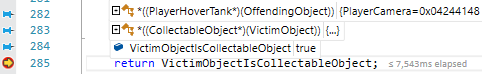
Therefore, this test is successful.

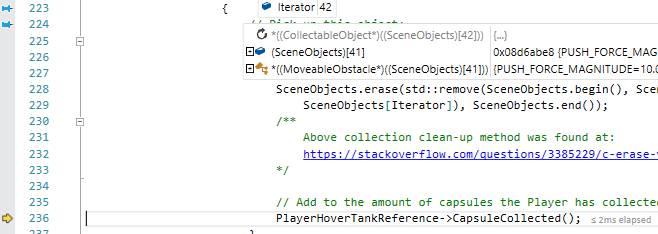
### Feature Test 6

To initialise this test, the Player is to have collected all but 1 of the Energy-Capsules in the level and the Player’s hover-tank is near to the last Energy-Capsules of the level:

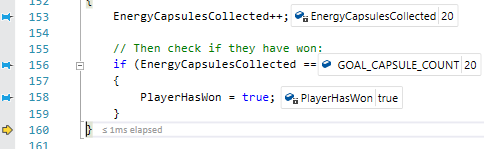


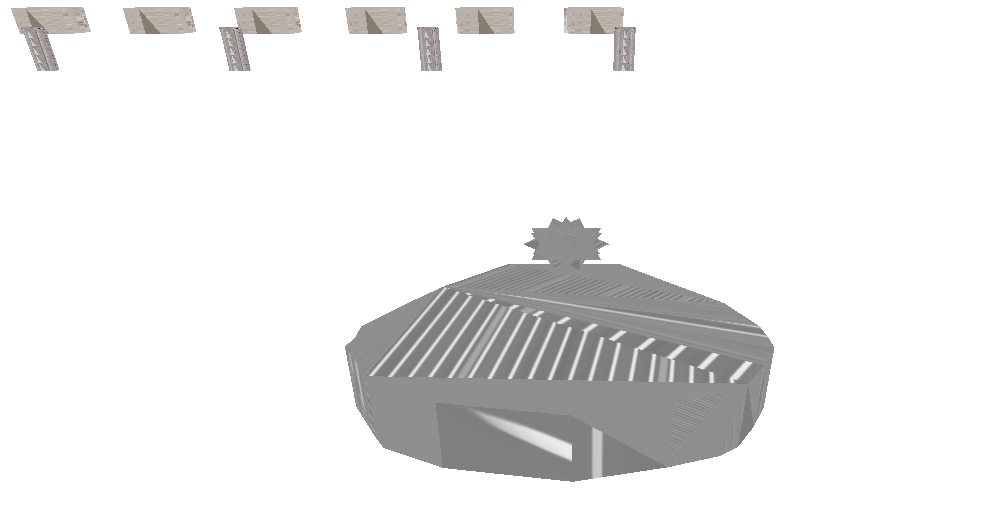
When the Player collides with the last Energy-Capsule, the following breakpoint is triggered (within RepositionGameObject):



This value is then checked in ManageCollisionChecking() and given it is true, the Energy-Capsule is removed from the collection using the methods in this screenshot:

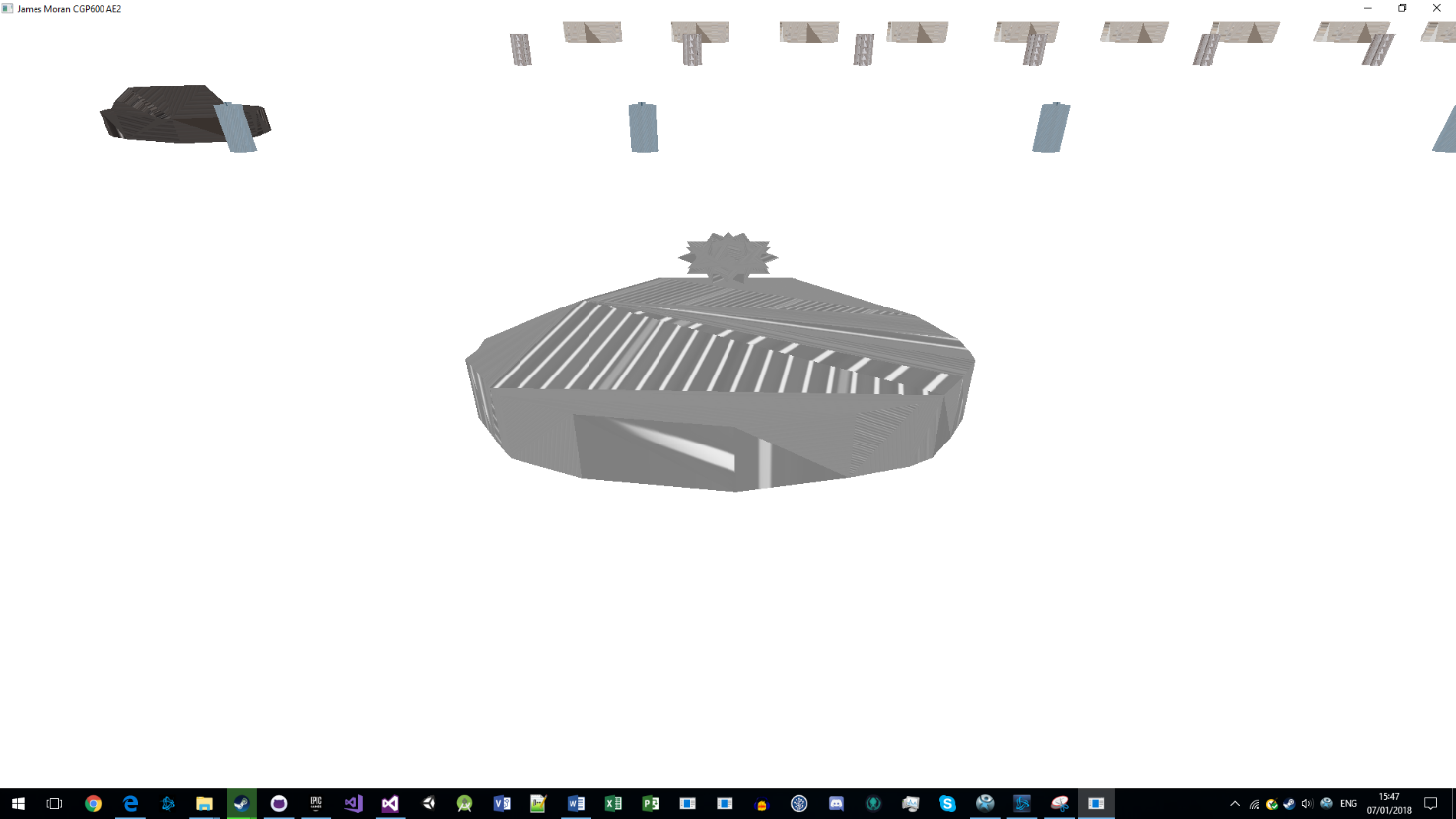
(The Energy-Capsule was at position 42 in SceneObjects, but as it has now been removed, that index of the vector no longer has a value (as the vector has a reduced size) and the last value in SceneObjects is a MoveableObstacle). Then CapsuleCollected() is called:

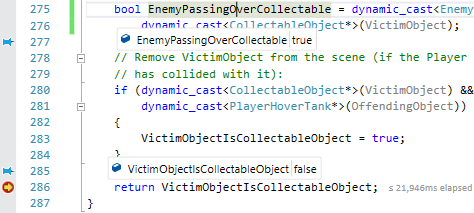
Which then causes the Player to win the game, with a freeze-frame of their hover-tank shown to the Player for 3 seconds, before the applications termination (with the last Energy-Capsule no longer present in the game-scene):

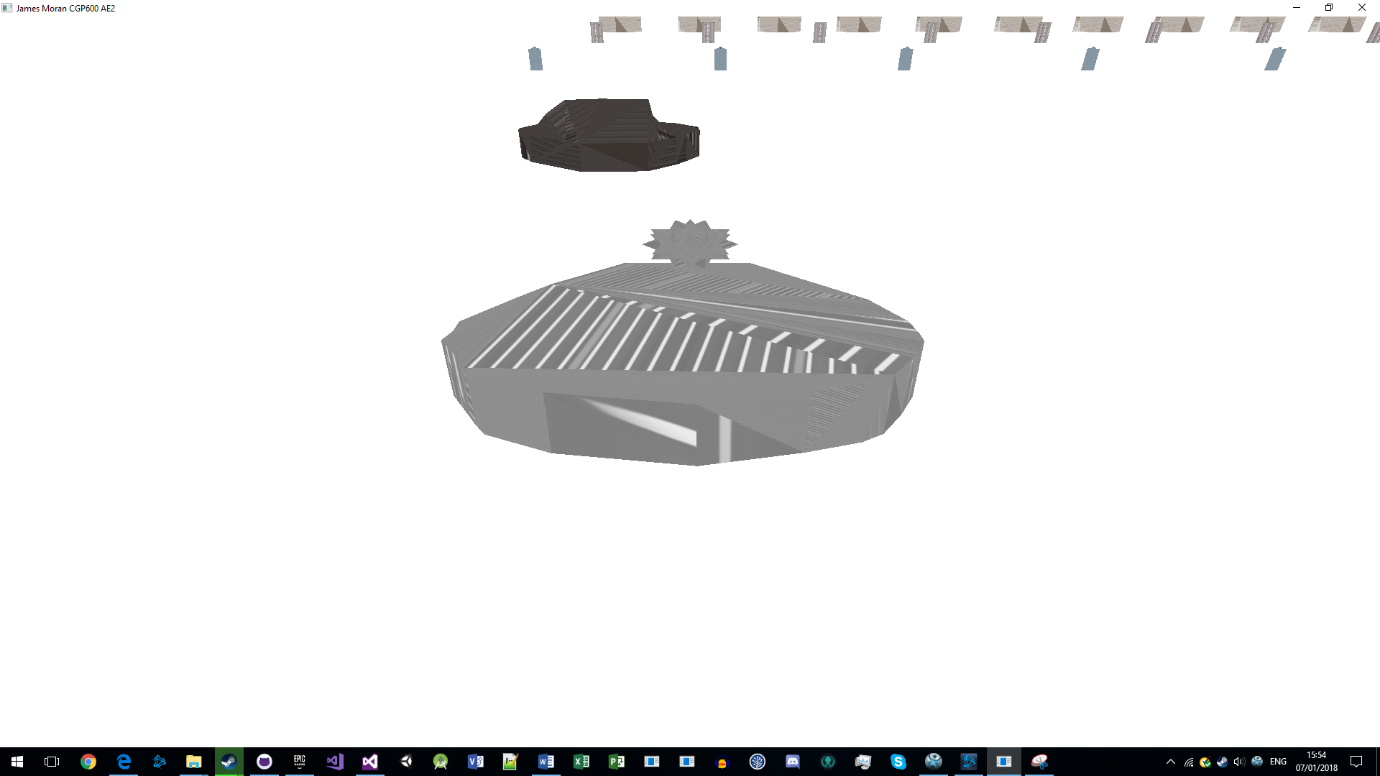
Therefore, this test is successful.

### Feature Test 7

For this test, the Enemy’s hover-tank must be near an Energy-Capsule:

Then, the same breakpoint (with different conditions), for the previous test is triggered:

 As they are an EnemyHoverTank, the Energy-Capsule will not be collected, resulting in them passing through it, instead of removing it from the collection (screenshot showing this is on the next page):

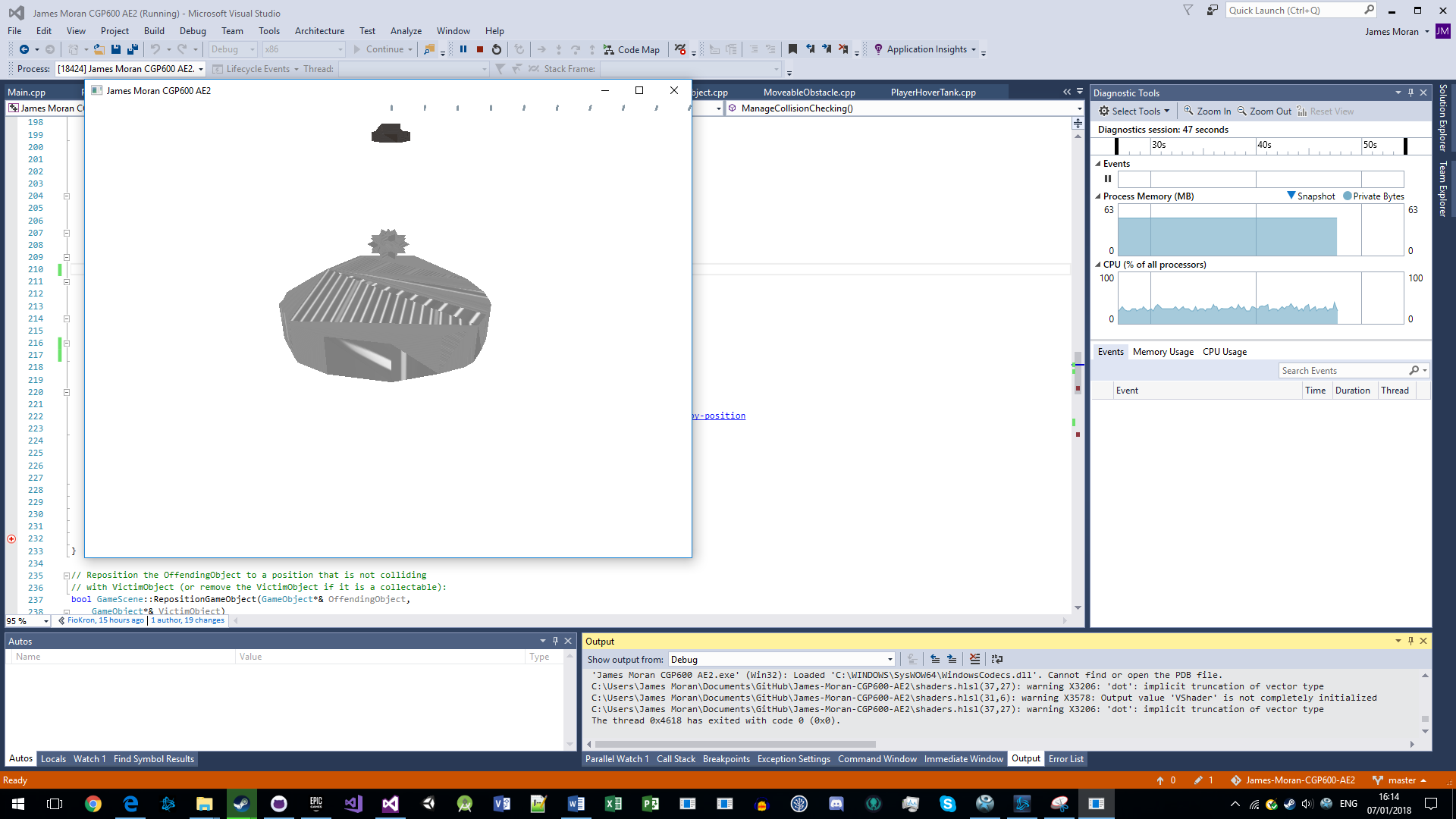
Therefore, this test is successful.

### Feature Test 8

As can be seen from the previous tests, the Enemy’s hover-tank will rotate to the face the Player’s hover-tank, then move towards them in a straight line, so this test is also successful.

### Feature Test 9

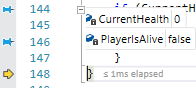
For this test, the Enemy’s hover-tank must be lined up with the Player’s hover-tank, soon to collide with it:



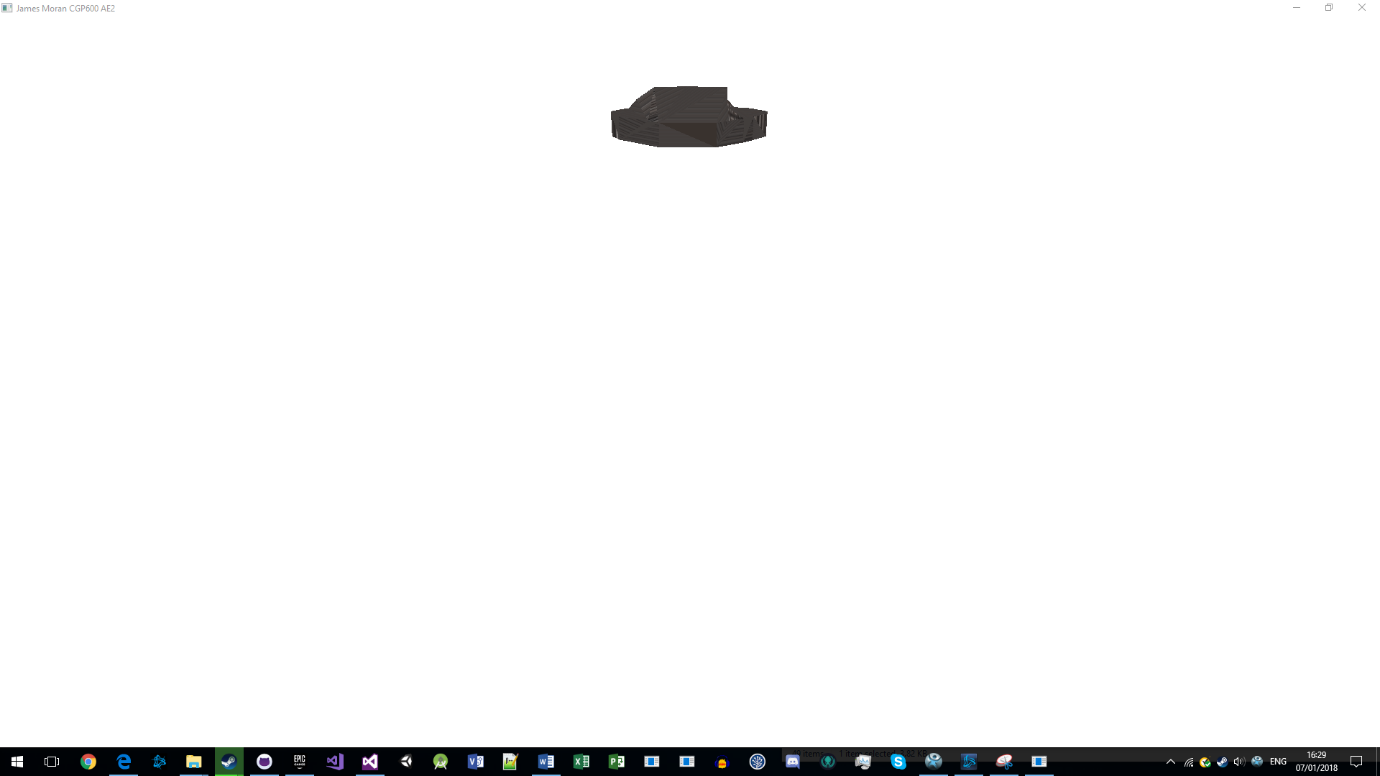
After triggering a breakpoint in HandleCollisionChecking(), program-flow transitions into the PlayerHoverTank’s ModifyHealth() method:

OAs the Player was at 10 health (before taking 10 health-points of damage), this test is successful and flows into the final test.

### Feature Test 10

Given where program execution was stopped after the previous test’s conclusion, as the Player is at 0 health-points, they are no longer alive:

So, on the next page, one can see that the Player’s hover-tank is no longer displayed:

The game will then freeze for 3 seconds, before terminating itself, as expected, so this test is successful, ending the last of the Feature Tests.

Basic Testing is in the next section of this appendix:

## Basic Testing

Testing for each of the basic requirements, is as documented in this table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Test Description | Expected Result | Actual Result | Success or Failure |
| 1 | The Player should be able to move around the environment. | The Player can move around the environment. | The Player can move around the environment. | Success. |
| 2 | The environment should be quite large and must have static and moving obstacles. These must be models, with some of these objects able to be pushed by the Player and some of these objects being collectable. | The environment has static, moveable and collectable objects, whilst being quite large (with the Player’s view distance at 1000 units) and no bounds on where the Player can move to in this environment. | The environment has static, moveable and collectable objects, whilst being quite large (with the Player’s view distance at 1000 units) and no bounds on where the Player can move to in this environment. | Success. |
| 3 | The objects in the environment must have textures and some form of lighting. | The objects have textures and lighting showing on them. | The objects have textures but no discernible lighting present. | Failure: No lighting on scene-objects |
| 4 | The Player should collide with objects, either stopping for static-objects, moving the object if it is moveable and picking up collectable objects. | The Player collides with all object types, getting knocked back slightly for static obstacles, moving the object accordingly for moveable objects and picking-up collectable objects. | The Player collides with all object types, getting knocked back slightly for static obstacles, moving the object accordingly for moveable objects and picking-up collectable objects. | Success. |
| 5 | The environment must contain non-player entities represented by a model, or at least a series of connected 3D shapes, using textures and lighting. | The Enemy’s hover-tank is visible in the environment (with a model, texture and lighting), after they get within line of sight to them/they get within the Player’s line of sight. | The Enemy’s hover-tank is visible in the environment (with a model and texture, but no discernible lighting), after they get within line of sight to them/they get within the Player’s line of sight. | Failure: No discernible lighting on Enemy-Hover-Tanks. |
| 6 | The entities should move around the environment in some fashion. | The Enemy’s hover-tank will rotate to face the Player and move towards them in a straight line, continuously. | The Enemy’s hover-tank will rotate to face the Player and move towards them in a straight line, continuously. | Success. |
| 7 | The entities should collide with objects. This should cause the entities to perform an action. | The Enemy’s hover-tank will stop for static-obstacles, push moveable-obstacles along with them and simply pass through Energy-Capsules. | The Enemy’s hover-tank will stop for static-obstacles, push moveable-obstacles along with them and simply pass through Energy-Capsules. | Success. |
| 8 | There should be some form of interaction between the non-player entities and the Player based on collision detection. | The Enemy’s hover-tank damages the Player’s hover-tank, when it collides with them. | The Enemy’s hover-tank damages the Player’s hover-tank, when it collides with them. | Success. |

## Evidence for Basic Testing

### Basic Test 1

For this test, as the Player is required to move around the environment for Feature Tests 2, 4 and 6, this test is also successful.

### Basic Test 2

For this test, as per the screenshot of Feature Test 1, one can see the Player’s hover-tank, StaticObstacles, MoveableObstacles and CollectableObjects, that are in the Player’s line of sight when they start the level, so this test is also successful.

### Basic Test 3

For this test, although objects are visible in the environment with an appropriate texture, there is no discernible lighting emanating from them, so this test has failed.

### Basic Test 4

For this test, as per Feature Tests 2, 4 and 6 (for checking the Player hover-tank’s interaction with StaticObstacles, MoveableObstacles and CollectableObjects respectively), as those tests has succeeded, this test has also succeeded.

### Basic Test 5

For this test, as per Feature Tests 3, 5, 7, 8, 9 and 10, there is a model to represent Enemy hover-tanks, with an appropriate texture, but no discernible lighting. So unfortunately, this test has failed on that basis.

### Basic Test 6

For this test, as per Feature Test 8 (referencing other feature tests), the Enemy’s hover-tank will rotate to face the Player, then move towards them in a straight line, continuously. Thus, this test is successful.

### Basic Test 7

For this test, as per Feature Tests 3, 5 and 7, the Enemy’s hover-tank will act accordingly for each type of scene-object (getting knocked back from static-obstacles, pushing moveable-obstacles as per their movement and not collecting Energy-Capsules when they pass over them). This test has therefore, succeeded.

### Basic Test 8

For this test, as per Feature Test 9 (leading into Feature Test 10), the Player will take damage when an Enemy’s hover-tank collides with them, leading to them losing the game if this is the 10th time they have been struck (as per Feature Test 10). Therefore, this final Basic Test has succeeded.

# Appendix D: Source Code

## Camera.h

#pragma once

#include <d3d11.h>

#define \_XM\_NO\_INTRINSICS\_

#define XM\_NO\_ALIGNMENT

#include <xnamath.h>

#include <math.h>

/\*\*

Reformed this class to match up to the ArcCamera class,

as detailed on p334-340 of Beginning DirectX 11 Game

Programming.

This was due to issues with the first implementation of

this class (trying to set-up the camera to act as a

third-person camera).

\*/

class Camera

{

private:

// Properties:

XMFLOAT3 Position;

XMFLOAT3 Target;

float CurrentDistanceToTarget, MinimumDistance,

MaximumDistance;

float XRotation, YRotation,

MinimumYRotation, MaximumYRotation;

// Functions:

/\*\*

Validate that a float value is

between LowerBound and UpperBound.

\*/

float ValidateFloatValue(float Value, float LowerBound,

float UpperBound);

public:

// Functions:

/\*\* Standard constructor. \*/

Camera();

/\*\* Standard destructor. \*/

~Camera();

// Set methods:

void SetDistance(float NewDistance, float

NewMinimumDistance, float NewMaximumDistance);

void SetRotation(float NewX, float NewY, float

NewMinimumY, float NewMaximumY);

void SetTarget(XMFLOAT3& NewTarget);

// Get Functions:

XMMATRIX GetViewMatrix();

// Properties:

// Constant values:

/\*\* For the default target and position values. \*/

const XMFLOAT3 ZERO\_FLOAT3 = XMFLOAT3(0.0f, 0.0f, 0.0f);

const XMFLOAT3 DEFAULT\_POSITION = XMFLOAT3(0.0f, 0.0f, 0.0f);

const float Y\_POSITION\_OFFSET = 30.0f;

/\*\*

The default distance from the Target and the

bounds of this distance.

\*/

const float DEFAULT\_DISTANCE = 50.0f;

const float DEFAULT\_MINIMUM\_DISTANCE = 1.0f;

const float DEFAULT\_MAXIMUM\_DISTANCE = 100.0f;

/\*\*

The default rotation values.

(In radians).

\*/

const float DEFAULT\_X\_ROTATION = 0.0f;

const float DEFAULT\_Y\_ROTATION = 0.0f;

};

## CollectableObject.h

#pragma once

#include "GameObject.h"

/\*\*

This is the class that Energy Capsules

use in game, for their representation

(as handled by the GameObject class).

\*/

class CollectableObject : public GameObject

{

public:

// Functions:

/\*\* Standard constructor. \*/

CollectableObject(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor. \*/

~CollectableObject();

};

## ControlledObject.h

#pragma once

#include "GameObject.h"

/\*\*

For objects controlled by the Player or AI.

\*/

class ControlledObject : public GameObject

{

protected:

/\*\*

Default directions (can be changed if

required).

\*/

XMVECTOR DefaultForwardDirection;

XMVECTOR DefaultRightDirection;

XMVECTOR DefaultUpDirection;

/\*\*

This object's directions relative to

itself.

\*/

XMVECTOR ObjectForwardDirection;

XMVECTOR ObjectRightDirection;

XMVECTOR ObjectUpDirection;

/\*\*

The forward direction vector

from this object's perspective.

\*/

XMVECTOR ObjectForwardTarget;

/\*\*

For the object's current

direction of movement in

the game-world.

\*/

XMVECTOR ObjectMovementDirection;

/\*\*

To be used in movement (in respects to

the orientation).

\*/

float MovementLeftRight;

float MovementForwardBackwards;

float MovementUpDown;

public:

/\*\* Standard constructor \*/

ControlledObject(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor \*/

virtual ~ControlledObject();

/\*\*

Override the Draw method to draw this object,

given movement.

\*/

void Draw(XMMATRIX\* View, XMMATRIX\* Projection)override;

// Movement methods:

/\*\*

These methods are overrideable,

to allow the Player's camera to

move with the Player's hover-tank.

\*/

/\*\* Translation using trig. \*/

void MoveForward(float DisplacementValue);

void MoveRight(float DisplacementValue);

/\*\*

When a static-obstacle, or

another controlled-object is

hit.

\*/

void RepelControlledObject(float RepulsionMagnitude);

// Get methods/functions:

XMVECTOR& GetObjectMovementDirection();

};

## DirectXSystem.h

#pragma once

#include <Windows.h>

#include <windows.h>

#include "Camera.h"

#include "PlayerHoverTank.h"

#include "EnemyHoverTank.h"

#include "StaticObstacle.h"

#include "MoveableObstacle.h"

#include "CollectableObject.h"

class DirectXSystem

{

public:

// Structures:

/\*\* To manage the camera \*/

struct CONSTANT\_BUFFER0

{

XMMATRIX WorldViewProjectionMatrix; // '64 bytes'

float Scale; // 4 bytes

XMFLOAT3 ScalePacking; // 3x4 bytes = 12 bytes

XMFLOAT4 NewPosition; // 4x4 bytes = 16 bytes

bool InputProvided; // 1 byte

bool PackingInputBytes[3]; // 3 bytes (1x3)

XMFLOAT3 AdditionalPackingInputBytes; // 3x4 bytes = 12 bytes

XMVECTOR DirectionalLightVector; // '16 bytes'

XMVECTOR DirectionalLightColour; // '16 bytes'

XMVECTOR AmbientLightColour; // '16 bytes'

/\*\* Standard 'constructor' \*/

void Initialise()

{

WorldViewProjectionMatrix = XMMATRIX();

Scale = 0.0f;

ScalePacking = XMFLOAT3(0.0f, 0.0f, 0.0f);

NewPosition = XMFLOAT4(0.0f, 0.0f, 0.0f, 0.0f);

InputProvided = false;

for each (bool CurrentValue in PackingInputBytes)

{

CurrentValue = false;

}

AdditionalPackingInputBytes = XMFLOAT3(0.0f, 0.0f, 0.0f);

DirectionalLightVector = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

DirectionalLightColour = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

AmbientLightColour = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

}

}; /\*\* Total size: 160 bytes. \*/

/\*\* 'Define vertex structure'. \*/

struct POS\_COL\_TEX\_NORM\_VERTEX

{

XMFLOAT3 Position;

XMFLOAT3 Color;

XMFLOAT2 TextureUVCoordinates;

XMFLOAT3 Normal;

};

/\*\*

From Main.cpp (for obstacles

in the scene).

\*/

struct NonControlledSceneObjects

{

/\*\*

To refer to the assets in this scene.

\*/

std::vector<StaticObstacle\*>

StaticObstaclesReference;

std::vector<MoveableObstacle\*>

MoveableObstaclesReference;

std::vector<CollectableObject\*>

CollectableObjectsReference;

/\*\*

For the quantity of each type of

obstacle.

\*/

int StaticObstacleCount;

int MoveableObstacleCount;

int CollectableObjectCount;

/\*\* Standard constructor. \*/

NonControlledSceneObjects::NonControlledSceneObjects(

int NewStaticObstacleCount,

int NewMoveableObstacleCount,

int NewCollectableObjectCount)

{

StaticObstacleCount = NewStaticObstacleCount;

MoveableObstacleCount = NewMoveableObstacleCount;

CollectableObjectCount = NewCollectableObjectCount;

}

};

// Functions:

/\*\* Standard constructor. \*/

DirectXSystem();

/\*\* Standard destructor. \*/

~DirectXSystem();

/\*\* Initialise the Direct3D system. \*/

HRESULT InitialiseD3D(HWND& WindowHandle);

/\*\*

Initialise the graphics system,

used by Direct3D, in this application.

\*/

HRESULT InitialiseGraphics(PlayerHoverTank\*&

PlayerHoverTankReference,

NonControlledSceneObjects\*&

Scene0ObjectsReference,

std::vector<EnemyHoverTank\*>&

EnemyHoverTanksReference);

/\*\* If ever the window is resized. \*/

HRESULT HandleWindowResizing(HWND& WindowHandle);

/\*\* Update the constant buffer's values, for the current transformation. \*/

void UpdateConstantBuffer(XMMATRIX& Projection, XMMATRIX& World,

XMMATRIX& View, XMVECTOR& DirectionalLightVector,

XMVECTOR& DirectionalLightColour, XMVECTOR& AmbientLightColour);

// Get Functions:

ID3D11DeviceContext\*& GetImmediateContextReference();

IDXGISwapChain\*& GetSwapChainReference();

ID3D11RenderTargetView\*& GetBackBufferRenderTargetViewReference();

/\*\* For getting references to the different types of buffers. \*/

ID3D11Buffer\*& GetConstantBuffer0Reference();

ID3D11Buffer\*& GetVertexBufferReference();

ID3D11Buffer\*& GetIndexBufferReference();

ID3D11DepthStencilView\*& GetZBufferReference();

/\*\* For shaders. \*/

ID3D11ShaderResourceView\*& GetPlayerHoverTankTextureReference();

ID3D11ShaderResourceView\*& GetStaticRockTextureReference();

ID3D11ShaderResourceView\*& GetWoodenBarrelTextureReference();

ID3D11ShaderResourceView\*& GetEnergyCapsuleTextureReference();

ID3D11ShaderResourceView\*& GetEnemyHoverTankTextureReference();

ID3D11ShaderResourceView\*& GetPlayerLossSplashScreenTexture();

ID3D11ShaderResourceView\*& GetPlayerVictorySplashScreenTexture();

ID3D11SamplerState\*& GetDefaultTextureSamplerReference();

/\*\* To refer to the D3DDevice. \*/

ID3D11Device\*& GetD3DDeviceReference();

// Constant values:

/\*\*

The initial position for each object type, is where the first

object of that type will be placed in the level.

\*/

const int SCENE0\_STATIC\_OBSTACLE\_COUNT = 20;

const XMFLOAT3 STATIC\_OBSTACLE\_INITIAL\_POSITION = XMFLOAT3(-500.0f, 0.0f, 0.0f);

const int SCENE0\_MOVEABLE\_OBSTACLE\_COUNT = 20;

const XMFLOAT3 MOVEABLE\_OBSTACLE\_INITIAL\_POSITION = XMFLOAT3(-500.0f, 0.0f, 250.0f);

const int SCENE0\_COLLECTABLE\_OBJECT\_COUNT = 20;

const XMFLOAT3 COLLECTABLE\_OBJECT\_INITIAL\_POSITION = XMFLOAT3(-500.0f, 0.0f, 500.0f);

const XMFLOAT3 DEFAULT\_POSITION\_INCREMENTOR = XMFLOAT3(50.0f, 0.0f, 0.0f);

/\*\*

For game-scene assets.

\*/

const std::string DEFAULT\_PLAYER\_MODEL\_FILE\_PATH = "Assets/PlayerShip.obj";

const std::string DEFAULT\_ENEMY\_MODEL\_FILE\_PATH = "Assets/EnemyHoverTank.obj";

const std::string DEFAULT\_STATIC\_OBSTACLE\_FILE\_PATH = "Assets/StaticRock.obj";

const std::string DEFAULT\_MOVEABLE\_OBSTACLE\_FILE\_PATH = "Assets/WoodenBarrel.obj";

const std::string DEFAULT\_COLLECTABLE\_OBJECT\_FILE\_PATH = "Assets/EnergyCapsule.obj";

const std::string DEFAULT\_PLAYER\_MODEL\_TEXTURE\_FILE\_PATH = "Assets/PlayerHoverTankTexture.tiff";

const std::string DEFAULT\_STATIC\_ROCK\_TEXTURE\_FILE\_PATH = "Assets/StaticRockTexture.jpg";

const std::string DEFAULT\_WOODEN\_BARREL\_TEXTURE\_FILE\_PATH = "Assets/WoodenBarrelTexture.jpg";

const std::string DEFAULT\_ENERGY\_CAPSULE\_TEXTURE\_FILE\_PATH = "Assets/EnergyCapsuleTexture.jpg";

const std::string DEFAULT\_ENEMY\_MODEL\_TEXTURE\_FILE\_PATH = "Assets/EnemyHoverTankTextureImage.tiff";

const std::string PLAYER\_VICTORY\_SPLASH\_SCREEN\_TEXTURE\_FILE\_PATH =

"Assets/PlayerVictoryTexture.png";

const std::string PLAYER\_LOSS\_SPLASH\_SCREEN\_TEXTURE\_FILE\_PATH = "Assets/PlayerLossTexture.png";

/\*\* For 'random' placement. \*/

const int MINIMUM\_XZ\_POSITION = -100;

const int MAXIMUM\_XZ\_POSITION = 100;

/\*\*

(These two values were constant,

but changed for parsing into con-

structors):

Default position values for the

Player's hover-tank.

\*/

XMFLOAT3 DEFAULT\_PLAYER\_MODEL\_LOCATION = XMFLOAT3(0.0f, 0.0f, 1000.0f);

/\*\* For the Enemy hover-tank. \*/

XMFLOAT3 DEFAULT\_FIRST\_ENEMY\_HOVER\_TANK\_LOCATION = XMFLOAT3(0.0f, 0.0f, -1000.0f);

const int ENEMY\_HOVER\_TANK\_QUANTITY = 1;

/\*\*

For the number of vertices present in the shape to draw.

For a certain reason, this has to be (at least) 1 greater than the

number of vertices to be drawn, to draw them...

This is for a cube that uses vertices.

\*/

const UINT VERTEX\_COUNT = 17;

/\*\*

Default value to clear the viewport to,

before drawing to it.

\*/

const float DEFAULT\_CLEAR\_COLOUR[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

private:

// Properties:

/\*\*

For the types of drivers and respective DirectX 11

features to use.

\*/

D3D\_DRIVER\_TYPE DriverTypeReference = D3D\_DRIVER\_TYPE\_NULL;

D3D\_FEATURE\_LEVEL FeatureLevelReference = D3D\_FEATURE\_LEVEL\_11\_0;

/\*\*

A reference to the device for DirectX, as well as the

hardware context along with a reference to the swap-chain

(important for swapping between the front and back buffers,

as one draws to the back buffer, then swaps this buffer with

the front buffer to become the new front buffer; showing what

has been drawn).

\*/

ID3D11Device\* D3DDeviceReference = nullptr;

ID3D11DeviceContext\* ImmediateContextReference = nullptr;

IDXGISwapChain\* SwapChainReference = nullptr;

/\*\*

To refer to the shaders and their input-layout.

\*/

ID3D11VertexShader\* VertexShader = nullptr;

ID3D11PixelShader\* PixelShader = nullptr;

ID3D11InputLayout\* InputLayout = nullptr;

ID3D11ShaderResourceView\* PlayerHoverTankTexture = nullptr;

ID3D11ShaderResourceView\* StaticRockTexture = nullptr;

ID3D11ShaderResourceView\* WoodenBarrelTexture = nullptr;

ID3D11ShaderResourceView\* EnergyCapsuleTexture = nullptr;

ID3D11ShaderResourceView\* EnemyHoverTankTexture = nullptr;

ID3D11ShaderResourceView\* PlayerVictorySplashScreenTexture = nullptr;

ID3D11ShaderResourceView\* PlayerLossSplashScreenTexture = nullptr;

ID3D11SamplerState\* DefaultTextureSampler = nullptr;

/\*\* To show objects in front of and behind each other \*/

ID3D11DepthStencilView\* ZBuffer = nullptr;

/\*\* The one and only constant buffer \*/

ID3D11Buffer\* ConstantBuffer0 = nullptr;

CONSTANT\_BUFFER0 ConstantBuffer0Values;

/\*\* For indices \*/

ID3D11Buffer\* IndexBuffer = nullptr;

/\*\* To hold onto the vertices to draw \*/

ID3D11Buffer\* VertexBuffer = nullptr;

/\*\* For whatever has been drawn, to then be swapped with the front buffer \*/

ID3D11RenderTargetView\* BackBufferRenderTargetViewReference = nullptr;

/\*\* For the placement of non-controlled objects. \*/

XMFLOAT3 CurrentStaticObstaclePosition;

XMFLOAT3 CurrentMoveableObstaclePosition;

XMFLOAT3 CurrentCollectableObjectPosition;

// Constant Values:

/\*\* The first and only constant buffer that is used \*/

const UINT FIRST\_CONSTANT\_BUFFER\_BYTE\_WIDTH = 160u;

// Functions:

// Initialisers:

/\*\*

For the non-controlled

scene-objects of each type.

\*/

HRESULT InitialiseStaticObstacles(

NonControlledSceneObjects\*&

Scene0ObjectsReference, HRESULT&

ResultHandleReference);

HRESULT InitialiseMoveableObstacles(

NonControlledSceneObjects\*&

Scene0ObjectsReference, HRESULT&

ResultHandleReference);

HRESULT InitialiseCollectableObjects(

NonControlledSceneObjects\*&

Scene0ObjectsReference, HRESULT&

ResultHandleReference);

/\*\*

For any texture of a GameObject.

\*/

void InitialiseTexture(std::string TextureFilePath,

ID3D11ShaderResourceView\*& TextureReference);

/\*\* Set-up the shaders for use in drawing graphics \*/

HRESULT SetUpShaders(HRESULT ResultHandle);

/\*\* For varience in object placement \*/

XMFLOAT3 GetNewObjectPosition();

// Debugging:

/\*\* For checking on D3D11 Warnings \*/

void ReportLiveObjects();

};

## EnemyHoverTank.h

#pragma once

#include "ControlledObject.h"

#include "PlayerHoverTank.h"

class EnemyHoverTank : public ControlledObject

{

public:

// Enumerations:

/\*\*

For the current state this tank is in

(on the Finite State Machine).

\*/

enum AIState

{

Initial,

HasTarget,

MovingToTarget

};

// Functions:

/\*\* Standard constructor. \*/

EnemyHoverTank(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor. \*/

~EnemyHoverTank();

/\*\* For state mangement. \*/

void ManageAIState(PlayerHoverTank\*&

PlayerHoverTankReference);

// Properties:

// Constant Values:

const float ENEMY\_TRANSLATION\_MAGNITUDE = 0.020f;

const int ENEMY\_DAMAGE\_TO\_PLAYER = 10;

private:

// Functions/methods:

/\*\* Rotate to face a vector-position. \*/

void RotateToFace();

/\*\* Get a target to move to. \*/

void FindTarget(PlayerHoverTank\*&

PlayerHoverTankReference);

/\*\* Move to the current target. \*/

void MoveToTarget();

// Properties:

/\*\* For the current state of this tank. \*/

AIState CurrentState;

/\*\*

The Enemy's default target is

the Player's hover-tank.

\*/

PlayerHoverTank\* CurrentTarget;

};

## GameObject.h

#pragma once

#include "objfilemodel.h"

/\*\*

This class is used to wrap objfilemodel.h's

functionality, in a format that one can

use to load .obj files into the project.

This is to be used for all objects that are

visible to the Player in the level (scene),

such as their tank, enemy tanks and objects

they can interact with (such as Energy-

Capsules).

\*/

class GameObject

{

protected:

// Functions:

/\*\* Get the centre point of this GameObject's model. \*/

void CalculateModelCentrePoint();

/\*\* Get the radius of this bounding sphere. \*/

void CalculateBoundingSphereRadius();

float CalculateCentreToVertexDistanceSquared(int VertexIndex);

// Properties:

/\*\* Reference pointers. \*/

ID3D11Device\* D3DDeviceReference;

ID3D11DeviceContext\* ImmediateContextReference;

ObjFileModel\* ObjectModelReference;

ID3D11VertexShader\* GameObjectVertexShaderReference;

ID3D11PixelShader\* GameObjectPixelShaderReference;

ID3D11InputLayout\* GameObjectInputLayoutReference;

ID3D11Buffer\* GameObjectConstantBufferReference;

/\*\* For transformation of this object. \*/

float GameObjectScale;

XMVECTOR ObjectScaleVector;

XMVECTOR UpDirectionVector;

XMVECTOR PositionVector;

/\*\*

For the PlayerHoverTank to parse

this value to the Camera class.

\*/

XMFLOAT3 PositionFloat3;

XMVECTOR RotationVector;

XMMATRIX RotationMatrix;

XMMATRIX World;

/\*\* For basic bounding-sphere collision. \*/

XMFLOAT3 BoundingSphereCentre;

float BoundingSphereRadius;

/\*\* For the lighting this object. \*/

XMVECTOR DirectionalLightShinesFrom;

XMVECTOR DirectionalLightColour;

XMVECTOR AmbientLightColour;

// Constant values:

/\*\*

To use instead of having to type

out "FILE NOT LOADED".

\*/

const std::string FILE\_NOT\_LOADED = "FILE NOT LOADED";

const UINT FIRST\_CONSTANT\_BUFFER\_BYTE\_WIDTH = 112u;

// Functions:

/\*\* For initialising the shaders \*/

HRESULT InitialiseShaders(HRESULT& ResultHandleReference);

/\*\* For initialising the constant-buffer(s) \*/

HRESULT InitialiseConstantBuffers(HRESULT& ResultHandleReference);

/\*\* Keep rotation values within certain bounds \*/

void MaintainPitchRange();

void MaintainYawRange();

public:

// Functions:

/\*\* Standard constructor. \*/

GameObject(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\*

Standard destructor (can be overriden by

sub-classes).

\*/

virtual ~GameObject();

/\*\* Load a model .obj. \*/

int LoadObjectModel(std::string FileName);

/\*\* Check to see if this is valid. \*/

bool ObjectModelFileReferenceValid();

/\*\*

To draw the .obj model (can be overriden if

additional processing is required).

\*/

virtual void Draw(XMMATRIX\* View, XMMATRIX\* Projection);

// Set methods:

/\*\* For if a collision occurs. \*/

void SetPositionVector(XMVECTOR NewPosition);

/\*\* For rotation \*/

void Yaw(float RotationValue);

void Pitch(float RotationValue);

// Get functions:

/\*\*

Get the world space coordinates of the

bounding sphere's centre point.

\*/

XMVECTOR GetBoundingSphereWorldSpacePosition();

/\*\* Check against another GameObject for collision. \*/

bool CheckCollision(GameObject\*& TargetObject);

float CalculateDistanceSquaredBetweenBoundingSpheres(XMVECTOR

ThisGameObjectBoundingSphereWorldSpacePosition, XMVECTOR

TargetBoundingSphereWorldSpacePosition);

float GetBoundingSphereRadius();

float GetScale();

/\*\* For getting the position and rotation. \*/

XMVECTOR& GetVectorPosition();

XMFLOAT3& GetPositionFloat3();

XMVECTOR& GetVectorRotation();

XMMATRIX& GetRotationMatrix();

/\*\* Get the file path of the object model \*/

std::string& GetObjectModelFileName();

XMVECTOR& GetDirectionalLightShinesFrom();

XMVECTOR& GetDirectionalLightColour();

XMVECTOR& GetAmbientLightColour();

// Properties:

// Constant values:

const XMFLOAT3 ZERO\_FLOAT3 = XMFLOAT3(0.0f, 0.0f, 0.0f);

const float MINIMUM\_PITCH = -90.0f;

const float MAXIMUM\_PITCH = 90.0f;

const float MINIMUM\_YAW = 0.0f;

const float MAXIMUM\_YAW = 360.0f;

};

## GameScene.h

#pragma once

#include "GameObject.h"

#include "DirectXSystem.h"

#include "WindowClass.h"

class GameScene

{

public:

// Functions:

/\*\* Standard constructor. \*/

GameScene(std::vector<GameObject\*>& NewSceneObjects,

int NewSceneID, bool SceneIsActive,

ID3D11DeviceContext\* NewImmediateContextReference);

/\*\* Standard destructor. \*/

~GameScene();

/\*\*

For when this GameScene

is to be updated.

\*/

HRESULT UpdateGameScene(ID3D11RenderTargetView\* BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference, ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference, ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference, DirectXSystem\*& DirectXSystemHandleReference);

// Get Functions:

bool GetIsSceneActive();

bool GetPlayerIsAlive();

bool GetPlayerHasWon();

int GetUniqueSceneID();

XMMATRIX& GetViewMatrix();

XMMATRIX& GetProjectionMatrix();

private:

// Functions/methods:

/\*\* Draw objects to the back-buffer. \*/

void DrawObjects(XMMATRIX\* View, XMMATRIX\* Projection,

DirectXSystem\*& DirectXSystemHandleReference);

/\*\* Render the frame for certain scenes \*/

HRESULT RenderFrame(ID3D11RenderTargetView\*

BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference,

ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference,

ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference,

DirectXSystem \*& DirectXSystemHandleReference);

/\*\*

Check for collision between any objects (before

rendering the frame).

\*/

void ManageCollisionChecking();

/\*\*

Manage which object is to move where.

\*/

bool RepositionGameObject(GameObject\*& OffendingObject,

GameObject\*& VictimObject);

void PushBackControlledObject(GameObject\*& OffendingObject,

GameObject\*& VictimObject);

void PushAwayMoveableObstacle(GameObject\*& OffendingObject,

GameObject\*& VictimObject);

/\*\* Set the transformation for the camera. \*/

void SetupCustomWorldViewProjectionMatrix(XMMATRIX& Projection,

XMMATRIX& World, XMMATRIX& View, const XMVECTOR& NewTranslation,

const XMMATRIX & NewRotation, WindowClass \*& WindowClassHandleReference,

DirectXSystem \*& DirectXSystemHandleReference);

void UpdateEnemyHoverTanks();

// Get Functions:

/\*\*

Look through all the game objects to find the

Player's hover-tank (the Player).

\*/

void GetPlayerHoverTankReference();

/\*\*

Perform similar actions to the above

method, to find references to Enemy

hover-tanks.

\*/

void GetEnemyHoverTankReferences();

// Properties:

/\*\*

For refering to the immediate context of the

device to which, the user can see.

\*/

ID3D11DeviceContext\* ImmediateContextReference;

/\*\*

For when certain members of the

PlayerHoverTank are required.

\*/

PlayerHoverTank\* PlayerHoverTankReference;

/\*\*

To allow updating of the EnemyHoverTanks.

\*/

std::vector<EnemyHoverTank\*> EnemyHoverTanksReference;

/\*\* For all the objects in the scene. \*/

std::vector<GameObject\*> SceneObjects;

/\*\* The scene's unique ID. \*/

int UniqueSceneID;

/\*\* For the WorldViewProjection matrix: \*/

XMMATRIX Projection, World, View;

// Flags:

/\*\* For the scene's active state. \*/

bool IsSceneActive;

bool PlayerHasWon;

bool PlayerIsAlive;

// Constant values:

/\*\* The field of view for the camera to use. \*/

const FLOAT DEFAULT\_FIELD\_OF\_VIEW = 60.0f; // Angle noted in degrees

/\*\*

For finding any controllable objects in the

SceneObjects (or other checking).

\*/

const std::string PLAYER\_FILE\_PATH = "Assets/PlayerShip.obj";

const std::string ENEMY\_FILE\_PATH = "Assets/EnemyHoverTank.obj";

const std::string STATIC\_ROCK\_FILE\_PATH = "Assets/StaticRock.obj";

const std::string MOVEABLE\_WOODEN\_BARREL\_FILE\_PATH = "Assets/WoodenBarrel.obj";

const std::string COLLECTABLE\_ENERGY\_CAPSULE\_FILE\_PATH = "Assets/EnergyCapsule.obj";

const std::string GENERIC\_SPLASH\_SCREEN\_FILE\_PATH = "Assets/GenericSplashScreen.obj";

};

## GlobalReferences.h

#pragma once

#include <string>

// Global Properties:

// Constant values:

/\*\* For the respective Virtual-Key codes \*/

// Numbers (not from the numpad):

const UINT VK\_0 = 0x30u;

const UINT VK\_1 = 0x31u;

const UINT VK\_2 = 0x32u;

const UINT VK\_3 = 0x33u;

const UINT VK\_4 = 0x34u;

const UINT VK\_5 = 0x35u;

const UINT VK\_6 = 0x36u;

const UINT VK\_7 = 0x37u;

const UINT VK\_8 = 0x38u;

const UINT VK\_9 = 0x39u;

// Leters:

const UINT VK\_AKEY = 0x41u;

const UINT VK\_BKEY = 0x42u;

const UINT VK\_CKEY = 0x43u;

const UINT VK\_DKEY = 0x44u;

const UINT VK\_EKEY = 0x45u;

const UINT VK\_FKEY = 0x46u;

const UINT VK\_GKEY = 0x47u;

const UINT VK\_HKEY = 0x48u;

const UINT VK\_IKEY = 0x49u;

const UINT VK\_JKEY = 0x4Au;

const UINT VK\_KKEY = 0x4Bu;

const UINT VK\_LKEY = 0x4Cu;

const UINT VK\_MKEY = 0x4Du;

const UINT VK\_NKEY = 0x4Eu;

const UINT VK\_OKEY = 0x4Fu;

const UINT VK\_PKEY = 0x50u;

const UINT VK\_QKEY = 0x51u;

const UINT VK\_RKEY = 0x52u;

const UINT VK\_SKEY = 0x53u;

const UINT VK\_TKEY = 0x54u;

const UINT VK\_UKEY = 0x55u;

const UINT VK\_VKEY = 0x56u;

const UINT VK\_WKEY = 0x57u;

const UINT VK\_XKEY = 0x58u;

const UINT VK\_YKEY = 0x59u;

const UINT VK\_ZKEY = 0x5Au;

// To load the splash screen's default asset:

const std::string GENERIC\_SPLASH\_SCREEN\_FILE\_PATH = "Assets/GenericSplashScreen.obj";

/\*\* How long to sleep for before terminating the application. \*/

const DWORD VICTORY\_LOSS\_FREEZE\_TIME = 3000ul;

//////////////////////////////////////////////////////////////////////////////////////

// Forward declarations (functions/methods):

//////////////////////////////////////////////////////////////////////////////////////

LRESULT CALLBACK WndProc(HWND WindowHandle, UINT Message, WPARAM

WParam, LPARAM LParam);

/\*\*

This function initialises the pointers declared in

Main.cpp.

\*/

void DefaultProgramConstructor();

void ShutdownD3D();

## MoveableObstacle.h

#pragma once

#include "GameObject.h"

class MoveableObstacle : public GameObject

{

public:

/\*\* Standard constructor. \*/

MoveableObstacle(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor. \*/

~MoveableObstacle();

/\*\* If ever an impact occurs with this obstacle. \*/

void OnImpact(XMVECTOR& ImpactDirection);

// Properties:

// Constant values:

const float PUSH\_FORCE\_MAGNITUDE = 10.0f;

};

## objfilemodel.h

#pragma once

#define \_XM\_NO\_INTRINSICS\_

#define XM\_NO\_ALIGNMENT

#include <d3d11.h>

#include <d3dx11.h>

#include <dxerr.h>

#include <xnamath.h>

#include <stdio.h>

#include <string>

#include <vector>

using namespace std;

class ObjFileModel

{

private:

ID3D11Device\* pD3DDevice;

ID3D11DeviceContext\* pImmediateContext;

//////////////////////////////////////////////////

int loadfile(char\* fname);

char\* fbuffer;

long fbuffersize; // filesize

size\_t actualsize; // actual size of loaded data (can be less if loading as text files as ASCII CR (0d) are stripped out)

//////////////////////////////////////////////////

void parsefile();

bool getnextline() ;

bool getnexttoken(int& tokenstart, int& tokenlength);

unsigned int tokenptr;

//////////////////////////////////////////////////

bool createVB();

ID3D11Buffer\* pVertexBuffer;

public:

struct xyz { float x, y, z; }; //used for vertices and normals during file parse

struct xy { float x, y; }; //used for texture coordinates during file parse

// Define model vertex structure

struct MODEL\_POS\_TEX\_NORM\_VERTEX

{

XMFLOAT3 Pos;

XMFLOAT2 TexCoord;

XMFLOAT3 Normal;

};

string filename;

ObjFileModel(char\* filename, ID3D11Device\* device, ID3D11DeviceContext\* context);

~ObjFileModel();

void Draw(void);

vector <xyz> position\_list; // list of parsed positions

vector <xyz> normal\_list; // list of parsed normals

vector <xy> texcoord\_list; // list of parsed texture coordinates

vector <int> pindices, tindices, nindices; // lists of indicies into above lists derived from faces

MODEL\_POS\_TEX\_NORM\_VERTEX\* vertices;

unsigned int numverts;

};

## PlayerHoverTank.h

#pragma once

/\*\*

This class is used to represent the Player

(so not only a GameObject, but this class

uses a Camera as well).

\*/

#include "ControlledObject.h"

#include "Camera.h"

class PlayerHoverTank : public ControlledObject

{

private:

// Properties:

/\*\* This Camera will follow the Player around \*/

Camera\* PlayerCamera;

/\*\* For refering to the result-handle. \*/

HRESULT PlayerResultHandleReference;

/\*\* Their important statistics. \*/

int CurrentHealth;

int EnergyCapsulesCollected;

// Flags:

bool PlayerIsAlive;

bool PlayerHasWon;

public:

// Functions:

/\*\* Standard constructor. \*/

PlayerHoverTank(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor. \*/

~PlayerHoverTank();

// Get Functions:

Camera\*& GetPlayerCameraReference();

bool GetPlayerHasWon();

bool GetPlayerIsAlive();

/\*\*

For the Player's hover-tank to handle camera positioning

relative to the tank.

\*/

void Draw(XMMATRIX\* View, XMMATRIX\* Projection)override;

/\*\*

Handle damage/healing to the Player's hover-tank

(parse in a positive value to damage them, or a

negative value to heal them).

\*/

void ModifyHealth(int HealthModificationValue);

/\*\* For when they collect a capsule. \*/

void CapsuleCollected();

// Properties:

const XMVECTOR DEFAULT\_CAMERA\_LOCATION = XMVectorSet(0.0f, 30.0f, 30.0f, 0.0f);

const XMVECTOR DEFAULT\_CAMERA\_ORIENTATION = XMVectorSet(45.0f, 180.0f, 0.0f, 0.0f);

/\*\* For use in repelling EnemyHoverTanks. \*/

const float REPULSION\_FORCE\_MAGNITUDE = 100.0f;

const int DEFAULT\_INITIAL\_HEALTH = 100;

const int GOAL\_CAPSULE\_COUNT = 20;

};

## SceneManager.h

#pragma once

#include <Windows.h>

#include <windows.h>

#include <d3d11.h>

#include <d3dx11.h>

#include <dxerr.h>

#include "WindowClass.h"

#include "Camera.h"

#include "GlobalReferences.h"

#include "GameScene.h"

#define \_XM\_NO\_INTRINSICS\_

#define XM\_NO\_ALIGNMENT

#include <xnamath.h>

class SceneManager

{

public:

// Structures:

/\*\* To refer to any GameScene \*/

struct SceneComponents

{

/\*\*

Initialise this member of this structure

before it is used.

\*/

std::vector<GameObject\*>& SceneObjects = std::vector<GameObject\*>();

int SceneID;

bool SceneIsActive;

SceneComponents::SceneComponents(std::vector<GameObject\*>&

NewSceneObjects, int NewSceneID, bool SceneActive)

{

SceneObjects = NewSceneObjects;

SceneID = NewSceneID;

SceneIsActive = SceneActive;

}

};

// Functions:

/\*\* Standard constructor \*/

SceneManager(ID3D11DeviceContext\* NewImmediateContextReference,

std::vector<SceneComponents\*>& NewGameSceneValues);

/\*\* Standard destructor \*/

~SceneManager();

/\*\* Update the active scene(s) in the game. \*/

HRESULT UpdateGameScenes(ID3D11RenderTargetView\* BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference, ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference, ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference, DirectXSystem\*& DirectXSystemHandleReference);

// Get functions:

// To check on its values:

GameScene\*& GetDefaultGameScene();

private:

/\*\*

To store all the game scenes of the game

(whether they are active or not).

\*/

std::vector<GameScene\*> GameScenes;

// Constant values:

const UINT FIRST\_CONSTANT\_BUFFER\_BYTE\_WIDTH = 144;

/\*\* For the constant buffer used in the vertex shader. \*/

const UINT VERTEX\_SHADER\_BUFFER\_COUNT = 1;

};

## StaticObstacle.h

#pragma once

#include "GameObject.h"

class StaticObstacle : public GameObject

{

public:

// Functions/methods:

/\*\* Standard constructor \*/

StaticObstacle(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition);

/\*\* Standard destructor \*/

~StaticObstacle();

// Properties:

// Constant values:

const float REPULSION\_FORCE\_MAGNITUDE = 10.0f;

};

## WindowClass.h

#pragma once

#include <Windows.h>

#include <windows.h>

#include <string>

#include "Camera.h"

#include "DirectXSystem.h"

/\*\*

Class name: WindowClass.

Purpose: To encapsulate the functionality

of window creation (for a win32 application),

in order to a show a window that DirectX can

use.

\*/

class WindowClass

{

private:

// Properties:

HINSTANCE InstanceWindowHandle = nullptr;

HWND WindowHandle;

// Constant values:

/\*\*

The window's title (left as my name plus

the unit and assignment for now).

\*/

const std::string WINDOW\_TITLE = "James Moran CGP600 AE2\0";

/\*\*

The author's name (me, James Moran).

\*/

const std::string AUTHOR\_NAME = "James Moran\0";

/\*\*

For the window's extents. (Changed from FLOAT to LONG

given narrowing conversion other-wise being required,

as well as possible loss of data).

\*/

const LONG DEFAULT\_WINDOW\_WIDTH = 800L; // Pixels

const LONG DEFAULT\_WINDOW\_HEIGHT = 600L; // Pixels

/\*\* For the all transformation functions (scale, rotation and translation) \*/

const float TRANSLATION\_MAGNITUDE = 5.0f;

const float ROTATION\_MAGNITUDE = 45.0f;

// Functions:

/\*\*

Handle the action(s) taken when certain keys are pressed

(This function would not account shift or ctrl pressed

whilst another key is being pressed):

\*/

void ManageKeyPressed(int VirtualKeyCode, Camera\*& DefaultCameraReference,

PlayerHoverTank\*& PlayerHoverTankReference);

public:

// Functions/Methods:

/\*\* Standard constructor. \*/

WindowClass();

/\*\* Standard destructor. \*/

~WindowClass();

/\*\*

Initialise the window for DirectX to use (as well

as registering the class).

\*/

HRESULT InitialiseWindow(HINSTANCE InstanceHandle, int nCmdShow, WNDPROC WindowsProcedure);

/\*\*

Called in Main.cpp, to handle WndProc behavior

(W used to equate to 'word' and L used to equate

to 'long', but this is not the case in x64. This

was how they were initially defined in 16-bit

windows).

\*/

HRESULT WindowsProcedureLogic(HWND WindowHandle, UINT Message, WPARAM WParam,

LPARAM LParam, DirectXSystem\*& DirectXSystemHandleReference,

PlayerHoverTank\*& PlayerHoverTankReference);

// Get Functions:

HWND& GetWindowHandle();

FLOAT GetFOVAspectRatio();

};

## Camera.cpp

#include "Camera.h"

// Initialise:

Camera::Camera()

{

Position = DEFAULT\_POSITION;

Target = ZERO\_FLOAT3;

SetDistance(DEFAULT\_DISTANCE, DEFAULT\_MINIMUM\_DISTANCE,

DEFAULT\_MAXIMUM\_DISTANCE);

SetRotation(DEFAULT\_X\_ROTATION, DEFAULT\_Y\_ROTATION,

-XM\_PIDIV2, XM\_PIDIV2);

}

// Clean-up:

Camera::~Camera()

{

}

// Set methods:

void Camera::SetDistance(float NewDistance, float NewMinimumDistance, float NewMaximumDistance)

{

CurrentDistanceToTarget = NewDistance;

MinimumDistance = NewMinimumDistance;

MaximumDistance = NewMaximumDistance;

// Keep CurrentDistanceToTarget within its bounds:

CurrentDistanceToTarget = ValidateFloatValue(

CurrentDistanceToTarget, MinimumDistance,

MaximumDistance);

}

void Camera::SetRotation(float NewX, float NewY, float NewMinimumY, float NewMaximumY)

{

XRotation = NewX;

YRotation = NewY;

MinimumYRotation = NewMinimumY;

MaximumYRotation = NewMaximumY;

// Keep YRotation within its bounds:

YRotation = ValidateFloatValue(YRotation, MinimumYRotation,

MaximumYRotation);

}

void Camera::SetTarget(XMFLOAT3& NewTarget)

{

Target = NewTarget;

}

// Get functions:

// For the WorldViewProjection matrix:

XMMATRIX Camera::GetViewMatrix()

{

XMVECTOR Zoom = XMVectorSet(0.0f, 0.0f,

CurrentDistanceToTarget, 1.0f);

XMMATRIX RotationMatrix = XMMatrixRotationRollPitchYaw(

XRotation, YRotation, 0.0f);

Zoom = XMVector3Transform(Zoom, RotationMatrix);

XMVECTOR PositionVector = XMLoadFloat3(&Position);

XMVECTOR LookAt = XMLoadFloat3(&Target);

PositionVector = LookAt + Zoom;

PositionVector.y += Y\_POSITION\_OFFSET;

XMStoreFloat3(&Position, PositionVector);

XMVECTOR Up = XMVectorSet(0.0f, 1.0f, 0.0f, 1.0f);

Up = XMVector3Transform(Up, RotationMatrix);

return XMMatrixLookAtLH(PositionVector, LookAt, Up);

}

// Validation functions:

// For float values:

float Camera::ValidateFloatValue(float Value, float LowerBound, float UpperBound)

{

if (Value < LowerBound)

{

Value = LowerBound;

return Value;

}

if (Value > UpperBound)

{

Value = UpperBound;

return Value;

}

return Value;

}

## CollectableObject.cpp

#include "CollectableObject.h"

// Initialise (using an initialiser list):

CollectableObject::CollectableObject(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition) : GameObject(NewD3DDeviceReference, NewD3DDeviceContextReference,

ResultHandleReference, InitialPosition)

{

}

// Clean-up:

CollectableObject::~CollectableObject()

{

}

## ControlledObject.cpp

#include "ControlledObject.h"

// Structures:

struct MODEL\_CONSTANT\_BUFFER

{

XMMATRIX WorldViewProjectionMatrix; // '64 bytes ( 4 x 4 = 16 floats x 4 bytes)'

XMVECTOR DirectionalLightVector; // '16 bytes'

XMVECTOR DirectionalLightColour; // '16 bytes'

XMVECTOR AmbientLightColour; // '16 bytes'

}; // 'TOTAL SIZE = 112 bytes'

// Initialise (using an initialiser list):

ControlledObject::ControlledObject(ID3D11Device\*& NewD3DDeviceReference,

ID3D11DeviceContext\*& NewD3DDeviceContextReference,

HRESULT& ResultHandleReference, XMFLOAT3& InitialPosition)

: GameObject(NewD3DDeviceReference, NewD3DDeviceContextReference,

ResultHandleReference, InitialPosition)

{

// Set-up the default directions...

DefaultForwardDirection = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f);

DefaultRightDirection = XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f);

DefaultUpDirection = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

// ...then initialise the object's directions...

ObjectForwardDirection = DefaultForwardDirection;

ObjectRightDirection = DefaultRightDirection;

ObjectUpDirection = DefaultUpDirection;

// ...before initialising the translation and rotation values:

MovementLeftRight = 0.0f;

MovementForwardBackwards = 0.0f;

MovementUpDown = 0.0f;

ObjectMovementDirection = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

}

// Clean-up:

ControlledObject::~ControlledObject()

{

}

void ControlledObject::Draw(XMMATRIX\* View, XMMATRIX\* Projection)

{

// Also taken from the tutorial mentioned in the header file

// (of the Camera class, as parts of this function was taken

// from the Camera.cpp):

// Perform degrees to radians conversions first though:

float PitchRotationRadians = XMConvertToRadians(RotationVector.x);

float YawRotationRadians = XMConvertToRadians(RotationVector.y);

float RollRotationRadians = XMConvertToRadians(RotationVector.z);

RotationMatrix = XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

ObjectForwardTarget = XMVector3TransformCoord(DefaultForwardDirection,

RotationMatrix);

ObjectForwardTarget = XMVector3Normalize(ObjectForwardTarget);

XMMATRIX RotateYTempMatrix = XMMatrixRotationY(YawRotationRadians);

ObjectRightDirection = XMVector3TransformCoord(DefaultRightDirection, RotateYTempMatrix);

ObjectUpDirection = XMVector3TransformCoord(DefaultUpDirection, RotateYTempMatrix);

ObjectForwardDirection = XMVector3TransformCoord(DefaultForwardDirection, RotateYTempMatrix);

// Update the position (and current movement direction), for movement:

XMVECTOR HorizontalMovementVelocity = MovementLeftRight \* ObjectRightDirection;

XMVECTOR VerticalMovementVelocity = MovementUpDown \* ObjectUpDirection;

XMVECTOR DepthMovementVelocity = MovementForwardBackwards \* ObjectForwardDirection;

PositionVector += HorizontalMovementVelocity;

PositionVector += DepthMovementVelocity;

// Set the movement-direction to HorizontalMovementVelocity

// before incrementing it by depth and height:

ObjectMovementDirection = HorizontalMovementVelocity;

ObjectMovementDirection += DepthMovementVelocity;

ObjectMovementDirection += VerticalMovementVelocity;

ObjectMovementDirection = XMVector3Normalize(ObjectMovementDirection);

MovementLeftRight = 0.0f;

MovementForwardBackwards = 0.0f;

MovementUpDown = 0.0f;

ObjectForwardTarget += PositionVector;

// Update the World transform matrix and the buffers now:

// Scale first, then rotation, with translation as the last of

// the transformation actions to take place:

World = XMMatrixScalingFromVector(ObjectScaleVector);

World \*= XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

World \*= XMMatrixTranslationFromVector(PositionVector);

// For lighting on this object:

XMMATRIX TransposeMatrix = XMMATRIX();

MODEL\_CONSTANT\_BUFFER ObjectConstantBufferValues;

ObjectConstantBufferValues.WorldViewProjectionMatrix = World \* (\*View) \* (\*Projection);

ObjectConstantBufferValues.DirectionalLightColour = DirectionalLightColour;

ObjectConstantBufferValues.AmbientLightColour = AmbientLightColour;

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Transform(DirectionalLightShinesFrom,

TransposeMatrix);

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Normalize(ObjectConstantBufferValues.

DirectionalLightVector);

ImmediateContextReference->VSSetConstantBuffers(0u, 1u, &GameObjectConstantBufferReference);

ImmediateContextReference->UpdateSubresource(GameObjectConstantBufferReference, 0u, nullptr,

&ObjectConstantBufferValues, 0u, 0u);

// Set this model's shaders and input layout as active:

ImmediateContextReference->VSSetShader(GameObjectVertexShaderReference, nullptr, 0);

ImmediateContextReference->PSSetShader(GameObjectPixelShaderReference, nullptr, 0);

ImmediateContextReference->IASetInputLayout(GameObjectInputLayoutReference);

ObjectModelReference->Draw();

}

/\*\*

Parse in a positive value for the direction as

per the function name, or a negative value for the

opposite direction.

\*/

// Add Z-Axis offset to the object:

void ControlledObject::MoveForward(float DisplacementValue)

{

MovementForwardBackwards -= DisplacementValue;

}

// Add X-Axis offset to the object:

void ControlledObject::MoveRight(float DisplacementValue)

{

MovementLeftRight -= DisplacementValue;

}

// Repel this object from a static-object:

void ControlledObject::RepelControlledObject(float RepulsionMagnitude)

{

PositionVector += -(ObjectMovementDirection \* RepulsionMagnitude);

}

XMVECTOR& ControlledObject::GetObjectMovementDirection()

{

return ObjectMovementDirection;

}

## DirectXSystem.cpp

#include "DirectXSystem.h"

#include <d3d11.h>

#include <d3dx11.h>

#include <dxerr.h>

#include "WindowClass.h"

// For 'random' positioning of objects:

#include <random>

#include <time.h>

#include <functional>

// Initialise:

DirectXSystem::DirectXSystem()

{

ConstantBuffer0Values.Initialise();

CurrentStaticObstaclePosition = STATIC\_OBSTACLE\_INITIAL\_POSITION;

CurrentMoveableObstaclePosition = MOVEABLE\_OBSTACLE\_INITIAL\_POSITION;

CurrentCollectableObjectPosition = COLLECTABLE\_OBJECT\_INITIAL\_POSITION;

}

// Clean-up:

DirectXSystem::~DirectXSystem()

{

if (BackBufferRenderTargetViewReference)

{

BackBufferRenderTargetViewReference->Release();

BackBufferRenderTargetViewReference = nullptr;

}

if (ZBuffer) ZBuffer->Release(); ZBuffer = nullptr;

if (SwapChainReference) SwapChainReference->Release();

SwapChainReference = nullptr;

if (ImmediateContextReference) ImmediateContextReference->Release();

ImmediateContextReference = nullptr;

if (PlayerHoverTankTexture) PlayerHoverTankTexture->Release(); PlayerHoverTankTexture = nullptr;

if (StaticRockTexture) StaticRockTexture->Release(); StaticRockTexture = nullptr;

if (WoodenBarrelTexture) WoodenBarrelTexture->Release(); WoodenBarrelTexture = nullptr;

if (PlayerVictorySplashScreenTexture) PlayerVictorySplashScreenTexture->Release();

PlayerVictorySplashScreenTexture = nullptr;

if (PlayerLossSplashScreenTexture) PlayerLossSplashScreenTexture->Release();

PlayerLossSplashScreenTexture = nullptr;

if (DefaultTextureSampler) DefaultTextureSampler->Release(); DefaultTextureSampler = nullptr;

if (InputLayout) InputLayout->Release(); InputLayout = nullptr;

if (VertexShader) VertexShader->Release(); VertexShader = nullptr;

if (PixelShader) PixelShader->Release(); PixelShader = nullptr;

if (VertexBuffer) VertexBuffer->Release(); VertexBuffer = nullptr;

if (IndexBuffer) IndexBuffer->Release(); IndexBuffer = nullptr;

if (ConstantBuffer0) ConstantBuffer0->Release(); ConstantBuffer0 = nullptr;

// Report on any objects that are still live (before releasing the device).

// (Only for debug build solutions):

ReportLiveObjects();

if (D3DDeviceReference) D3DDeviceReference->Release();

D3DDeviceReference = nullptr;

}

// Create D3D device and swap chain

HRESULT DirectXSystem::InitialiseD3D(HWND& WindowHandle)

{

HRESULT ResultHandle = S\_OK;

RECT WindowRectangle;

GetClientRect(WindowHandle, &WindowRectangle);

UINT Width = WindowRectangle.right - WindowRectangle.left;

UINT Height = WindowRectangle.bottom - WindowRectangle.top;

UINT CreateDeviceFlags = 0;

#ifdef \_DEBUG

CreateDeviceFlags |= D3D11\_CREATE\_DEVICE\_DEBUG;

#endif

D3D\_DRIVER\_TYPE DriverTypes[] =

{

D3D\_DRIVER\_TYPE\_HARDWARE, // comment out this line if you need to test D3D 11.0 functionality on hardware that doesn't support it

D3D\_DRIVER\_TYPE\_WARP, // comment this out also to use reference device

D3D\_DRIVER\_TYPE\_REFERENCE,

};

UINT numDriverTypeReferences = ARRAYSIZE(DriverTypes);

D3D\_FEATURE\_LEVEL FeatureLevels[] =

{

D3D\_FEATURE\_LEVEL\_11\_0,

D3D\_FEATURE\_LEVEL\_10\_1,

D3D\_FEATURE\_LEVEL\_10\_0,

};

UINT numFeatureLevels = ARRAYSIZE(FeatureLevels);

DXGI\_SWAP\_CHAIN\_DESC DefaultSwapChainDescription;

ZeroMemory(&DefaultSwapChainDescription, sizeof(DefaultSwapChainDescription));

DefaultSwapChainDescription.BufferCount = 1;

DefaultSwapChainDescription.BufferDesc.Width = Width;

DefaultSwapChainDescription.BufferDesc.Height = Height;

DefaultSwapChainDescription.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

DefaultSwapChainDescription.BufferDesc.RefreshRate.Numerator = 60;

DefaultSwapChainDescription.BufferDesc.RefreshRate.Denominator = 1;

DefaultSwapChainDescription.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

DefaultSwapChainDescription.OutputWindow = WindowHandle;

DefaultSwapChainDescription.SampleDesc.Count = 1;

DefaultSwapChainDescription.SampleDesc.Quality = 0;

DefaultSwapChainDescription.Windowed = true;

for (UINT DriverTypeIndex = 0; DriverTypeIndex < numDriverTypeReferences; DriverTypeIndex++)

{

DriverTypeReference = DriverTypes[DriverTypeIndex];

ResultHandle = D3D11CreateDeviceAndSwapChain(nullptr, DriverTypeReference, nullptr,

CreateDeviceFlags, FeatureLevels, numFeatureLevels,

D3D11\_SDK\_VERSION, &DefaultSwapChainDescription, &SwapChainReference,

&D3DDeviceReference, &FeatureLevelReference, &ImmediateContextReference);

if (SUCCEEDED(ResultHandle))

break;

}

if (FAILED(ResultHandle))

return ResultHandle;

// Get pointer to back buffer texture

ID3D11Texture2D \*pBackBufferTexture;

ResultHandle = SwapChainReference->GetBuffer(0, \_\_uuidof(ID3D11Texture2D),

(LPVOID\*)&pBackBufferTexture);

if (FAILED(ResultHandle)) return ResultHandle;

// Use the back buffer texture pointer to create the render target view

ResultHandle = D3DDeviceReference->CreateRenderTargetView(pBackBufferTexture, NULL,

&BackBufferRenderTargetViewReference);

pBackBufferTexture->Release();

if (FAILED(ResultHandle)) return ResultHandle;

// 'Create a Z buffer texture'

D3D11\_TEXTURE2D\_DESC ZBufferDescription;

ZeroMemory(&ZBufferDescription, sizeof(ZBufferDescription));

ZBufferDescription.Width = Width;

ZBufferDescription.Height = Height;

ZBufferDescription.ArraySize = 1;

ZBufferDescription.MipLevels = 1;

ZBufferDescription.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

ZBufferDescription.SampleDesc.Count = 1;

ZBufferDescription.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

ZBufferDescription.Usage = D3D11\_USAGE\_DEFAULT;

ID3D11Texture2D\* ZBufferTexture = nullptr;

ResultHandle = D3DDeviceReference->CreateTexture2D(&ZBufferDescription, NULL, &ZBufferTexture);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

// 'Create the Z buffer'

D3D11\_DEPTH\_STENCIL\_VIEW\_DESC DepthStencilViewDescription;

ZeroMemory(&DepthStencilViewDescription, sizeof(DepthStencilViewDescription));

DepthStencilViewDescription.Format = ZBufferDescription.Format;

DepthStencilViewDescription.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;

D3DDeviceReference->CreateDepthStencilView(ZBufferTexture, &DepthStencilViewDescription,

&ZBuffer);

// No longer required:

ZBufferTexture->Release();

// Set the render target view

ImmediateContextReference->OMSetRenderTargets(1, &BackBufferRenderTargetViewReference, ZBuffer);

// Set the viewport

D3D11\_VIEWPORT Viewport;

Viewport.TopLeftX = 0;

Viewport.TopLeftY = 0;

Viewport.Width = (FLOAT) Width;

Viewport.Height = (FLOAT) Height;

/\*\*

It seems as though the window is created in a diffrent on-screen location

(changing between 0 and the current value):

\*/

Viewport.MinDepth = 0.0f; // For Tutorial 06 Extra 01 (Now changed back to zero)

Viewport.MaxDepth = 1.0f;

ImmediateContextReference->RSSetViewports(1, &Viewport);

return S\_OK;

}

// Initialise graphics:

HRESULT DirectXSystem::InitialiseGraphics(PlayerHoverTank\*&

PlayerHoverTankReference,

NonControlledSceneObjects\*&

Scene0ObjectsReference,

std::vector<EnemyHoverTank\*>&

EnemyHoverTanksReference)//'03-01'

{

HRESULT ResultHandle = S\_OK;

// Set-up the Player's hover-tank:

PlayerHoverTankReference = new PlayerHoverTank(D3DDeviceReference, ImmediateContextReference,

ResultHandle, DEFAULT\_PLAYER\_MODEL\_LOCATION);

if (PlayerHoverTankReference)

{

ResultHandle = PlayerHoverTankReference->

LoadObjectModel(DEFAULT\_PLAYER\_MODEL\_FILE\_PATH);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

}

// Set-up the Enemy hover-tanks as well:

for (int Iterator = 0; Iterator < ENEMY\_HOVER\_TANK\_QUANTITY;

Iterator++)

{

EnemyHoverTanksReference.push\_back(new EnemyHoverTank(D3DDeviceReference,

ImmediateContextReference, ResultHandle,

DEFAULT\_FIRST\_ENEMY\_HOVER\_TANK\_LOCATION));

if (EnemyHoverTanksReference[Iterator])

{

ResultHandle = EnemyHoverTanksReference[Iterator]->

LoadObjectModel(DEFAULT\_ENEMY\_MODEL\_FILE\_PATH);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

}

}

// Set-up the scene's non-controlled objects:

Scene0ObjectsReference = new NonControlledSceneObjects(SCENE0\_STATIC\_OBSTACLE\_COUNT,

SCENE0\_MOVEABLE\_OBSTACLE\_COUNT, SCENE0\_COLLECTABLE\_OBJECT\_COUNT);

if (FAILED(InitialiseStaticObstacles(Scene0ObjectsReference,

ResultHandle)))

{

return ResultHandle;

}

if (FAILED(InitialiseMoveableObstacles(Scene0ObjectsReference,

ResultHandle)))

{

return ResultHandle;

}

if (FAILED(InitialiseCollectableObjects(Scene0ObjectsReference,

ResultHandle)))

{

return ResultHandle;

}

// Initialise all the textures:

InitialiseTexture(DEFAULT\_PLAYER\_MODEL\_TEXTURE\_FILE\_PATH, PlayerHoverTankTexture);

InitialiseTexture(DEFAULT\_STATIC\_ROCK\_TEXTURE\_FILE\_PATH, StaticRockTexture);

InitialiseTexture(DEFAULT\_WOODEN\_BARREL\_TEXTURE\_FILE\_PATH, WoodenBarrelTexture);

InitialiseTexture(DEFAULT\_ENERGY\_CAPSULE\_TEXTURE\_FILE\_PATH, EnergyCapsuleTexture);

InitialiseTexture(DEFAULT\_ENEMY\_MODEL\_TEXTURE\_FILE\_PATH, EnemyHoverTankTexture);

InitialiseTexture(PLAYER\_VICTORY\_SPLASH\_SCREEN\_TEXTURE\_FILE\_PATH,

PlayerVictorySplashScreenTexture);

InitialiseTexture(PLAYER\_LOSS\_SPLASH\_SCREEN\_TEXTURE\_FILE\_PATH,

PlayerLossSplashScreenTexture);

// Create the second constant buffer:

D3D11\_BUFFER\_DESC Constant\_Buffer0\_Description;

ZeroMemory(&Constant\_Buffer0\_Description, sizeof(Constant\_Buffer0\_Description));

Constant\_Buffer0\_Description.Usage = D3D11\_USAGE\_DEFAULT; // 'Can use UpdateSubresource() to update'

Constant\_Buffer0\_Description.ByteWidth = FIRST\_CONSTANT\_BUFFER\_BYTE\_WIDTH; // 'MUST be a multiple of 16, calculate from CB struct'

Constant\_Buffer0\_Description.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER; // 'Use as a constant buffer'

ResultHandle = D3DDeviceReference->CreateBuffer(&Constant\_Buffer0\_Description, NULL, &ConstantBuffer0);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

// 'Set up and create vertex buffer'

D3D11\_BUFFER\_DESC VertexBufferDescription;

ZeroMemory(&VertexBufferDescription, sizeof(VertexBufferDescription));

VertexBufferDescription.Usage = D3D11\_USAGE\_DYNAMIC; // 'Used by CPU and GPU'

VertexBufferDescription.ByteWidth = sizeof(POS\_COL\_TEX\_NORM\_VERTEX) \* VERTEX\_COUNT; // Total size of buffer, VERTEX\_COUNT vertices

VertexBufferDescription.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER; // 'Use as a vertex buffer'

VertexBufferDescription.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE; // 'Allow CPU access'

ResultHandle = D3DDeviceReference->CreateBuffer(&VertexBufferDescription, nullptr, &VertexBuffer); // 'Create the buffer'

if (FAILED(ResultHandle)) // return error code on failure

{

return ResultHandle;

}

if (FAILED(SetUpShaders(ResultHandle)))

{

return ResultHandle;

}

D3D11\_SAMPLER\_DESC DefaultTextureSamplerDescription;

ZeroMemory(&DefaultTextureSamplerDescription, sizeof(DefaultTextureSamplerDescription));

DefaultTextureSamplerDescription.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

DefaultTextureSamplerDescription.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

DefaultTextureSamplerDescription.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

DefaultTextureSamplerDescription.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

DefaultTextureSamplerDescription.MaxLOD = D3D11\_FLOAT32\_MAX;

D3DDeviceReference->CreateSamplerState(&DefaultTextureSamplerDescription, &DefaultTextureSampler);

/\*\*

To resolve:

D3D11 ERROR: ID3D11DeviceContext::Draw: Current Primitive Topology value (0)

is not valid. [EXECUTION ERROR #365: DEVICE\_DRAW\_INVALID\_PRIMITIVETOPOLOGY]

USE D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST HERE, NOT TRIANGLESTRIP!!DA:wA

\*/

ImmediateContextReference->IASetPrimitiveTopology(D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return S\_OK;

}

// Initialiser functions:

HRESULT DirectXSystem::InitialiseStaticObstacles(NonControlledSceneObjects\*& Scene0ObjectsReference,

HRESULT& ResultHandleReference)

{

int CollectionIterator = 0;

// Only to the limit:

for (CollectionIterator; CollectionIterator < SCENE0\_STATIC\_OBSTACLE\_COUNT;

CollectionIterator++)

{

Scene0ObjectsReference->StaticObstaclesReference.push\_back(

new StaticObstacle(D3DDeviceReference, ImmediateContextReference,

ResultHandleReference, CurrentStaticObstaclePosition));

// Positioning the objects in lines for now:

CurrentStaticObstaclePosition.x += DEFAULT\_POSITION\_INCREMENTOR.x;

if (Scene0ObjectsReference->

StaticObstaclesReference[CollectionIterator])

{

ResultHandleReference = Scene0ObjectsReference->

StaticObstaclesReference[CollectionIterator]

->LoadObjectModel(DEFAULT\_STATIC\_OBSTACLE\_FILE\_PATH);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

}

}

return ResultHandleReference;

}

HRESULT DirectXSystem::InitialiseMoveableObstacles(NonControlledSceneObjects\*& Scene0ObjectsReference,

HRESULT& ResultHandleReference)

{

int CollectionIterator = 0;

// Only to the limit:

for (CollectionIterator; CollectionIterator < SCENE0\_MOVEABLE\_OBSTACLE\_COUNT;

CollectionIterator++)

{

Scene0ObjectsReference->MoveableObstaclesReference.push\_back(

new MoveableObstacle(D3DDeviceReference, ImmediateContextReference,

ResultHandleReference, CurrentMoveableObstaclePosition));

CurrentMoveableObstaclePosition.x += DEFAULT\_POSITION\_INCREMENTOR.x;

if (Scene0ObjectsReference->

MoveableObstaclesReference[CollectionIterator])

{

ResultHandleReference = Scene0ObjectsReference->

MoveableObstaclesReference[CollectionIterator]

->LoadObjectModel(DEFAULT\_MOVEABLE\_OBSTACLE\_FILE\_PATH);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

}

}

return ResultHandleReference;

}

HRESULT DirectXSystem::InitialiseCollectableObjects(NonControlledSceneObjects\*& Scene0ObjectsReference,

HRESULT& ResultHandleReference)

{

int CollectionIterator = 0;

// Only to the limit:

for (CollectionIterator; CollectionIterator < SCENE0\_COLLECTABLE\_OBJECT\_COUNT;

CollectionIterator++)

{

Scene0ObjectsReference->CollectableObjectsReference.push\_back(

new CollectableObject(D3DDeviceReference, ImmediateContextReference,

ResultHandleReference, CurrentCollectableObjectPosition));

CurrentCollectableObjectPosition.x += DEFAULT\_POSITION\_INCREMENTOR.x;

if (Scene0ObjectsReference->

CollectableObjectsReference[CollectionIterator])

{

ResultHandleReference = Scene0ObjectsReference->

CollectableObjectsReference[CollectionIterator]

->LoadObjectModel(DEFAULT\_COLLECTABLE\_OBJECT\_FILE\_PATH);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

}

}

return ResultHandleReference;

}

// Given a file-path and a ID3D11ShaderResourceView reference...

void DirectXSystem::InitialiseTexture(std::string TextureFilePath, ID3D11ShaderResourceView \*& TextureReference)

{

D3DX11CreateShaderResourceViewFromFile(D3DDeviceReference,

TextureFilePath.c\_str(), NULL,

NULL, &TextureReference, NULL);

}

// Initialise the shaders:

HRESULT DirectXSystem::SetUpShaders(HRESULT ResultHandle)

{

// 'Load and compile pixel and vertex shaders - use vs\_5\_0 to target DX11 hardware only'

ID3DBlob \*VertexShaderBlob, \*PixelShaderBlob, \*Error;

ResultHandle = D3DX11CompileFromFile("shaders.hlsl", 0, 0, "VShader", "vs\_4\_0", 0, 0, 0,

&VertexShaderBlob, &Error, 0);

if (Error != 0) // 'check for shader compilation error'

{

OutputDebugStringA((char\*)Error->GetBufferPointer());

Error->Release();

if (FAILED(ResultHandle)) // 'don't fail is error is just a warning'

{

return ResultHandle;

};

}

ResultHandle = D3DX11CompileFromFile("shaders.hlsl", 0, 0, "PShader", "ps\_4\_0", 0, 0, 0,

&PixelShaderBlob, &Error, 0);

if (Error != 0) // 'check for shader compilation error'

{

OutputDebugStringA((char\*)Error->GetBufferPointer());

Error->Release();

if (FAILED(ResultHandle)) // 'don't fail is error is just a warning'

{

return ResultHandle;

};

}

// 'Create shader objects'

ResultHandle = D3DDeviceReference->CreateVertexShader(VertexShaderBlob->GetBufferPointer(),

VertexShaderBlob->GetBufferSize(), NULL, &VertexShader);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

ResultHandle = D3DDeviceReference->CreatePixelShader(PixelShaderBlob->GetBufferPointer(),

PixelShaderBlob->GetBufferSize(), NULL, &PixelShader);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

// 'Set the shader objects as active'

ImmediateContextReference->VSSetShader(VertexShader, 0, 0);

ImmediateContextReference->PSSetShader(PixelShader, 0, 0);

// 'Create and set the input layout object'

D3D11\_INPUT\_ELEMENT\_DESC iedesc[] =

{

{ "POSITION", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "COLOR", 0, DXGI\_FORMAT\_R32G32B32A32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0},

{ "TEXCOORD", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0},

{ "NORMAL", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0}

};

ResultHandle = D3DDeviceReference->CreateInputLayout(iedesc, ARRAYSIZE(iedesc), VertexShaderBlob->GetBufferPointer(),

VertexShaderBlob->GetBufferSize(), &InputLayout);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

ImmediateContextReference->IASetInputLayout(InputLayout);

/\*\*

Removed due to a breakpoint sometimes getting triggered here.

\*/

// Clean up the local pointers:

//if (VertexShaderBlob) VertexShaderBlob->Release(); VertexShaderBlob = nullptr;

//if (PixelShaderBlob) PixelShaderBlob->Release(); PixelShaderBlob = nullptr;

//if (Error) Error->Release(); Error = nullptr;

return ResultHandle;

}

// Report on live objects:

void DirectXSystem::ReportLiveObjects()

{

// Only if this is a debug build:

#ifdef \_DEBUG

ID3D11Debug\* DebugDevice = nullptr;

HRESULT ResultHandle = D3DDeviceReference->QueryInterface(\_\_uuidof(ID3D11Debug),

reinterpret\_cast<void\*\*>(&DebugDevice));

if (FAILED(ResultHandle))

{

return;

}

ResultHandle = DebugDevice->ReportLiveDeviceObjects(D3D11\_RLDO\_DETAIL);

if (FAILED(ResultHandle))

{

return;

}

if (DebugDevice)

{

DebugDevice->Release();

}

#endif

}

// Get the new 'random' position incrementor,

// to add on, for the new position of an object

// (in the XZ plane):

XMFLOAT3 DirectXSystem::GetNewObjectPosition()

{

std::default\_random\_engine Generator;

std::uniform\_int\_distribution<int> UniformDistribution(MINIMUM\_XZ\_POSITION,

MAXIMUM\_XZ\_POSITION);

// Seed the generator before binding:

Generator.seed((unsigned int) time(nullptr));

// Bound together for repeated uses:

// auto used here to skip having to set-up std::\_Binder:

auto GetRandomInteger = std::bind(UniformDistribution, Generator);

return XMFLOAT3((float) GetRandomInteger(), 0.0f, (float) GetRandomInteger());

}

HRESULT DirectXSystem::HandleWindowResizing(HWND& WindowHandle)

{

// The default value to return (if SwapChainReference is invalid):

const HRESULT SWAP\_CHAIN\_REFERENCE\_INVALID = HRESULT\_FROM\_WIN32(ERROR\_NOT\_FOUND);

// For checking the results of functions:

HRESULT ErrorResultHandle = SWAP\_CHAIN\_REFERENCE\_INVALID;

if (SwapChainReference)

{

/\*\*

Solution found at:

https://msdn.microsoft.com/en-us/library/windows/desktop/bb205075(v=vs.85).aspx#Handling\_Window\_Resizing

\*/

/\*\*

This result has been set to S\_OK in execution so far

(when calling ResizeBuffers):

\*/

ImmediateContextReference->OMSetRenderTargets(0, 0, 0);

// Release all outstanding references to the swap chain's buffers:

BackBufferRenderTargetViewReference->Release();

/\*\*

Then it is possible to resize the buffers correctly

(Using DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH for the

last parameter to allowing switching between full-screen

and windowed mode. I'm also using DXGI\_FORMAT\_UNKNOWN, to

preserve the existing format. Finally, I have set the first 3

parameters as 0, to preserve the exitsting number of buffers,

as well as having DXGI use the Width and Height of the client

area of the target window, for then new Width and Height).

SOURCE: https://msdn.microsoft.com/en-us/library/bb174577(v=vs.85).aspx

\*/

ErrorResultHandle = SwapChainReference->ResizeBuffers(0, 0, 0, DXGI\_FORMAT\_UNKNOWN,

DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH);

if (ErrorResultHandle != S\_OK)

{

return ErrorResultHandle;

}

// Get buffer and create a render-target-view.

ID3D11Texture2D\* pBuffer;

ErrorResultHandle = SwapChainReference->GetBuffer(0, \_\_uuidof(ID3D11Texture2D),

(void\*\*)&pBuffer);

if (ErrorResultHandle != S\_OK)

{

return ErrorResultHandle;

}

ErrorResultHandle = D3DDeviceReference->CreateRenderTargetView(pBuffer, NULL,

&BackBufferRenderTargetViewReference);

if (ErrorResultHandle != S\_OK)

{

return ErrorResultHandle;

}

pBuffer->Release();

ImmediateContextReference->OMSetRenderTargets(1, &BackBufferRenderTargetViewReference, NULL);

// Get the new bounds of the window:

RECT ClientBounds;

GetClientRect(WindowHandle, &ClientBounds);

FLOAT NewWidth = (FLOAT) (ClientBounds.right - ClientBounds.left);

FLOAT NewHeight = (FLOAT) (ClientBounds.bottom - ClientBounds.top);

// Set up the viewport.

D3D11\_VIEWPORT NewViewport;

NewViewport.Width = NewWidth;

NewViewport.Height = NewHeight;

NewViewport.MinDepth = 0.0f;

NewViewport.MaxDepth = 1.0f;

NewViewport.TopLeftX = 0;

NewViewport.TopLeftY = 0;

ImmediateContextReference->RSSetViewports(1, &NewViewport);

}

return ErrorResultHandle;

}

// Get method implementation:

ID3D11DeviceContext\*& DirectXSystem::GetImmediateContextReference()

{

return ImmediateContextReference;

}

IDXGISwapChain\*& DirectXSystem::GetSwapChainReference()

{

return SwapChainReference;

}

ID3D11RenderTargetView\*& DirectXSystem::GetBackBufferRenderTargetViewReference()

{

return BackBufferRenderTargetViewReference;

}

// The SceneManager will call this function:

void DirectXSystem::UpdateConstantBuffer(XMMATRIX& Projection, XMMATRIX& World,

XMMATRIX& View, XMVECTOR& DirectionalLightVector,

XMVECTOR& DirectionalLightColour, XMVECTOR& AmbientLightColour)

{

// For lighting:

XMMATRIX TransposeMatrix = XMMATRIX();

// Keep in mind the transformation order: Scale, Rotation, Translation:

ConstantBuffer0Values.WorldViewProjectionMatrix = World \* View \* Projection;

// Update the light properties now:

TransposeMatrix = XMMatrixTranspose(World);

ConstantBuffer0Values.DirectionalLightColour = DirectionalLightColour;

ConstantBuffer0Values.AmbientLightColour = AmbientLightColour;

ConstantBuffer0Values.DirectionalLightVector = XMVector3Transform(DirectionalLightVector,

TransposeMatrix);

ConstantBuffer0Values.DirectionalLightVector = XMVector3Normalize(

ConstantBuffer0Values.DirectionalLightVector);

ImmediateContextReference->UpdateSubresource(ConstantBuffer0, 0u, 0u, &ConstantBuffer0Values, 0u, 0u);

}

// Get Functions:

ID3D11Buffer\*& DirectXSystem::GetConstantBuffer0Reference()

{

return ConstantBuffer0;

}

ID3D11Buffer\*& DirectXSystem::GetVertexBufferReference()

{

return VertexBuffer;

}

ID3D11Buffer\*& DirectXSystem::GetIndexBufferReference()

{

return IndexBuffer;

}

ID3D11DepthStencilView\*& DirectXSystem::GetZBufferReference()

{

return ZBuffer;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetPlayerHoverTankTextureReference()

{

return PlayerHoverTankTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetStaticRockTextureReference()

{

return StaticRockTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetWoodenBarrelTextureReference()

{

return WoodenBarrelTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetEnergyCapsuleTextureReference()

{

return EnergyCapsuleTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetEnemyHoverTankTextureReference()

{

return EnemyHoverTankTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetPlayerLossSplashScreenTexture()

{

return PlayerLossSplashScreenTexture;

}

ID3D11ShaderResourceView\*& DirectXSystem::GetPlayerVictorySplashScreenTexture()

{

return PlayerVictorySplashScreenTexture;

}

ID3D11SamplerState\*& DirectXSystem::GetDefaultTextureSamplerReference()

{

return DefaultTextureSampler;

}

ID3D11Device\*& DirectXSystem::GetD3DDeviceReference()

{

return D3DDeviceReference;

}

## EnemyHoverTank.cpp

#include "EnemyHoverTank.h"

// Initialise (using an initialiser list):

EnemyHoverTank::EnemyHoverTank(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition) : ControlledObject(NewD3DDeviceReference, NewD3DDeviceContextReference,

ResultHandleReference, InitialPosition)

{

CurrentState = Initial;

}

// Clean-up:

EnemyHoverTank::~EnemyHoverTank()

{

}

// Handle management of the current AI-State:

void EnemyHoverTank::ManageAIState(PlayerHoverTank\*&

PlayerHoverTankReference)

{

switch (CurrentState)

{

case Initial:

FindTarget(PlayerHoverTankReference);

break;

case HasTarget:

MoveToTarget();

break;

// Keep moving to the target, even after

// one has been found:

case MovingToTarget:

MoveToTarget();

break;

default:

break;

}

}

// Get a target...

void EnemyHoverTank::FindTarget(PlayerHoverTank\*&

PlayerHoverTankReference)

{

CurrentTarget = PlayerHoverTankReference;

CurrentState = HasTarget;

}

// ...to turn-to-face towards...

void EnemyHoverTank::RotateToFace()

{

RotationVector.y = atan2(PositionVector.x -

CurrentTarget->GetVectorPosition().x, PositionVector.z

- CurrentTarget->GetVectorPosition().z) \* (180.0f / XM\_PI);

// Update the rotation matrix now:

float PitchRotationRadians = XMConvertToRadians(RotationVector.x);

float YawRotationRadians = XMConvertToRadians(RotationVector.y);

float RollRotationRadians = XMConvertToRadians(RotationVector.z);

RotationMatrix = XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

}

// ...to then move towards:

void EnemyHoverTank::MoveToTarget()

{

RotateToFace();

MoveForward(ENEMY\_TRANSLATION\_MAGNITUDE);

}

## GameObject.cpp

#include "GameObject.h"

// Structures:

struct MODEL\_CONSTANT\_BUFFER

{

XMMATRIX WorldViewProjectionMatrix; // '64 bytes ( 4 x 4 = 16 floats x 4 bytes)'

XMVECTOR DirectionalLightVector; // '16 bytes'

XMVECTOR DirectionalLightColour; // '16 bytes'

XMVECTOR AmbientLightColour; // '16 bytes'

}; // 'TOTAL SIZE = 112 bytes'

// Initialise:

GameObject::GameObject(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*& NewD3DDeviceContextReference,

HRESULT& ResultHandleReference, XMFLOAT3& InitialPosition)

{

D3DDeviceReference = NewD3DDeviceReference;

ImmediateContextReference = NewD3DDeviceContextReference;

PositionVector = XMVECTOR();

RotationVector = XMVECTOR();

PositionFloat3 = InitialPosition;

PositionVector = XMLoadFloat3(&PositionFloat3);

BoundingSphereRadius = 0.0f;

GameObjectScale = 1.0f;

ObjectScaleVector = XMVectorSet(GameObjectScale, GameObjectScale, GameObjectScale, 0.0f);

UpDirectionVector = XMVECTOR();

UpDirectionVector.x = 0.0f;

UpDirectionVector.y = 1.0f;

UpDirectionVector.z = 0.0f;

InitialiseShaders(ResultHandleReference);

InitialiseConstantBuffers(ResultHandleReference);

RotationMatrix = XMMatrixRotationRollPitchYawFromVector(RotationVector);

World = XMMATRIX();

DirectionalLightShinesFrom = XMVectorSet(0.0f, 0.0f, -1.0f, 0.0f);

DirectionalLightColour = XMVectorSet(1.0f, 1.0f, 1.0f, 0.0f);

AmbientLightColour = XMVectorSet(0.10f, 1.10f, 1.10f, 1.0f);

}

// Clean-up:

GameObject::~GameObject()

{

if (ObjectModelReference)

{

delete ObjectModelReference;

ObjectModelReference = nullptr;

}

if (GameObjectVertexShaderReference)

{

GameObjectVertexShaderReference->Release();

GameObjectVertexShaderReference = nullptr;

}

if (GameObjectPixelShaderReference)

{

GameObjectPixelShaderReference->Release();

GameObjectPixelShaderReference = nullptr;

}

if (GameObjectInputLayoutReference)

{

GameObjectInputLayoutReference->Release();

GameObjectInputLayoutReference = nullptr;

}

if (GameObjectConstantBufferReference)

{

GameObjectConstantBufferReference->Release();

GameObjectConstantBufferReference = nullptr;

}

}

// Find the model's centre point:

void GameObject::CalculateModelCentrePoint()

{

XMFLOAT3 MaximumXYZValues = ZERO\_FLOAT3;

XMFLOAT3 MinimumXYZValues = ZERO\_FLOAT3;

for (UINT VerticesIterator = 0u; VerticesIterator <

ObjectModelReference->numverts; VerticesIterator++)

{

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.x > MaximumXYZValues.x)

{

MaximumXYZValues.x = ObjectModelReference->

vertices[VerticesIterator].Pos.x;

}

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.x < MinimumXYZValues.x)

{

MinimumXYZValues.x = ObjectModelReference->

vertices[VerticesIterator].Pos.x;

}

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.y > MaximumXYZValues.y)

{

MaximumXYZValues.y = ObjectModelReference->

vertices[VerticesIterator].Pos.y;

}

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.y < MinimumXYZValues.y)

{

MinimumXYZValues.y = ObjectModelReference->

vertices[VerticesIterator].Pos.y;

}

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.z > MaximumXYZValues.z)

{

MaximumXYZValues.z = ObjectModelReference->

vertices[VerticesIterator].Pos.z;

}

if (ObjectModelReference->vertices[VerticesIterator]

.Pos.z < MinimumXYZValues.z)

{

MinimumXYZValues.z = ObjectModelReference->

vertices[VerticesIterator].Pos.z;

}

}

// Store the centre point of the minimum and maximumn X, Y and Z values

// in BoundingSphereCentre:

BoundingSphereCentre.x = (MaximumXYZValues.x - abs(MinimumXYZValues.x)) / 2.0f;

BoundingSphereCentre.y = (MaximumXYZValues.y - abs(MinimumXYZValues.y)) / 2.0f;

BoundingSphereCentre.z = (MaximumXYZValues.z - abs(MinimumXYZValues.z)) / 2.0f;

}

// Find the radius of the sphere cast from the object's centre point,

// to the furthest out vertex:

void GameObject::CalculateBoundingSphereRadius()

{

float GreatestDistance = 0.0f;

for (UINT VerticesIterator = 0u; VerticesIterator <

ObjectModelReference->numverts; VerticesIterator++)

{

float CurrentDistance = CalculateCentreToVertexDistanceSquared(

VerticesIterator);

if (CurrentDistance > GreatestDistance)

{

GreatestDistance = CurrentDistance;

}

}

BoundingSphereRadius = GreatestDistance;

}

// To help in calculating the distance between the centre point

// and a particular vertex:

float GameObject::CalculateCentreToVertexDistanceSquared(int VertexIndex)

{

return (pow(BoundingSphereCentre.x - ObjectModelReference->

vertices[VertexIndex].Pos.x, 2) + pow(BoundingSphereCentre.y -

ObjectModelReference->vertices[VertexIndex].Pos.y, 2)

+ pow(BoundingSphereCentre.z - ObjectModelReference->

vertices[VertexIndex].Pos.z, 2));

}

// Initialise all shaders:

HRESULT GameObject::InitialiseShaders(HRESULT& ResultHandleReference)

{

// 'Load and compile pixel and vertex shaders - use vs\_5\_0 to target DX11 hardware only'

ID3DBlob \*VertexShaderBlob, \*PixelShaderBlob, \*Error;

ResultHandleReference = D3DX11CompileFromFile("Model\_Shaders.hlsl", 0, 0, "ModelVS", "vs\_4\_0", 0, 0, 0,

&VertexShaderBlob, &Error, 0);

if (Error != 0) // 'check for shader compilation error'

{

OutputDebugStringA((char\*)Error->GetBufferPointer());

Error->Release();

if (FAILED(ResultHandleReference)) // 'don't fail is error is just a warning'

{

return ResultHandleReference;

};

}

ResultHandleReference = D3DX11CompileFromFile("Model\_Shaders.hlsl", 0, 0, "ModelPS", "ps\_4\_0", 0, 0, 0,

&PixelShaderBlob, &Error, 0);

if (Error != 0) // 'check for shader compilation error'

{

OutputDebugStringA((char\*)Error->GetBufferPointer());

Error->Release();

if (FAILED(ResultHandleReference)) // 'don't fail is error is just a warning'

{

return ResultHandleReference;

};

}

// 'Create shader objects'

ResultHandleReference = D3DDeviceReference->CreateVertexShader(VertexShaderBlob->GetBufferPointer(),

VertexShaderBlob->GetBufferSize(), NULL, &GameObjectVertexShaderReference);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

ResultHandleReference = D3DDeviceReference->CreatePixelShader(PixelShaderBlob->GetBufferPointer(),

PixelShaderBlob->GetBufferSize(), NULL, &GameObjectPixelShaderReference);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

// 'Set the shader objects as active'

ImmediateContextReference->VSSetShader(GameObjectVertexShaderReference, nullptr, 0);

ImmediateContextReference->PSSetShader(GameObjectPixelShaderReference, nullptr, 0);

// 'Create and set the input layout object'

D3D11\_INPUT\_ELEMENT\_DESC iedesc[] =

{

{ "POSITION", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "COLOR", 0, DXGI\_FORMAT\_R32G32B32A32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "TEXCOORD", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "NORMAL", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, D3D11\_APPEND\_ALIGNED\_ELEMENT, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 }

};

ResultHandleReference = D3DDeviceReference->CreateInputLayout(iedesc, ARRAYSIZE(iedesc), VertexShaderBlob->GetBufferPointer(),

VertexShaderBlob->GetBufferSize(), &GameObjectInputLayoutReference);

if (FAILED(ResultHandleReference))

{

return ResultHandleReference;

}

ImmediateContextReference->IASetInputLayout(GameObjectInputLayoutReference);

return ResultHandleReference;

}

// Initialise all constant buffer(s):

HRESULT GameObject::InitialiseConstantBuffers(HRESULT & ResultHandleReference)

{

D3D11\_BUFFER\_DESC ModelConstantBufferDescription;

ZeroMemory(&ModelConstantBufferDescription, sizeof(ModelConstantBufferDescription));

ModelConstantBufferDescription.Usage = D3D11\_USAGE\_DEFAULT; // 'Can use UpdateSubresource() to update'

ModelConstantBufferDescription.ByteWidth = FIRST\_CONSTANT\_BUFFER\_BYTE\_WIDTH; // 'MUST be a multiple of 16, calculate from CB struct'

ModelConstantBufferDescription.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER; // 'Use as a constant buffer'

ResultHandleReference = D3DDeviceReference->CreateBuffer(&ModelConstantBufferDescription, nullptr,

&GameObjectConstantBufferReference);

return ResultHandleReference;

}

// Load an object:

int GameObject::LoadObjectModel(std::string FileName)

{

// C-Style cast from const char\* to char\* used here (from std::string):

if (!ObjectModelReference)

{

ObjectModelReference = new ObjFileModel((char\*)FileName.c\_str(), D3DDeviceReference,

ImmediateContextReference);

}

if (ObjectModelReference->filename == FILE\_NOT\_LOADED)

{

return S\_FALSE;

}

// The centre point can now be calculated for this model:

CalculateModelCentrePoint();

CalculateBoundingSphereRadius();

return S\_OK;

}

bool GameObject::ObjectModelFileReferenceValid()

{

bool ReferenceValid = false;

if (ObjectModelReference)

{

ReferenceValid = true;

}

return ReferenceValid;

}

// To draw this model:

void GameObject::Draw(XMMATRIX\* View, XMMATRIX\* Projection)

{

// Scale first, then rotation, with translation as the last of

// the transformation actions to take place:

World = XMMatrixScalingFromVector(ObjectScaleVector);

// Get the rotation values in radians first though:

float PitchRotationRadians = XMConvertToRadians(RotationVector.x);

float YawRotationRadians = XMConvertToRadians(RotationVector.y);

float RollRotationRadians = XMConvertToRadians(RotationVector.z);

World \*= XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

World \*= XMMatrixTranslationFromVector(PositionVector);

// For lighting on this object:

XMMATRIX TransposeMatrix = XMMATRIX();

MODEL\_CONSTANT\_BUFFER ObjectConstantBufferValues;

ObjectConstantBufferValues.WorldViewProjectionMatrix = World \* (\*View) \* (\*Projection);

ObjectConstantBufferValues.DirectionalLightColour = DirectionalLightColour;

ObjectConstantBufferValues.AmbientLightColour = AmbientLightColour;

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Transform(DirectionalLightShinesFrom,

TransposeMatrix);

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Normalize(ObjectConstantBufferValues.

DirectionalLightVector);

ImmediateContextReference->VSSetConstantBuffers(0u, 1u, &GameObjectConstantBufferReference);

ImmediateContextReference->UpdateSubresource(GameObjectConstantBufferReference, 0u, nullptr,

&ObjectConstantBufferValues, 0u, 0u);

// Set this model's shaders and input layout as active:

ImmediateContextReference->VSSetShader(GameObjectVertexShaderReference, nullptr, 0);

ImmediateContextReference->PSSetShader(GameObjectPixelShaderReference, nullptr, 0);

ImmediateContextReference->IASetInputLayout(GameObjectInputLayoutReference);

// Make sure this is valid before using it:

if (ObjectModelReference)

{

ObjectModelReference->Draw();

}

}

void GameObject::SetPositionVector(XMVECTOR NewPosition)

{

PositionVector = NewPosition;

}

// Get Functions:

// For collision detection:

XMVECTOR GameObject::GetBoundingSphereWorldSpacePosition()

{

XMMATRIX WorldSphere = XMMATRIX();

WorldSphere = XMMatrixScalingFromVector(ObjectScaleVector);

// Get the rotation values in radians first though:

float PitchRotationRadians = XMConvertToRadians(RotationVector.x);

float YawRotationRadians = XMConvertToRadians(RotationVector.y);

float RollRotationRadians = XMConvertToRadians(RotationVector.z);

WorldSphere \*= XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

WorldSphere \*= XMMatrixTranslationFromVector(PositionVector);

XMVECTOR Offset = XMVectorSet(BoundingSphereCentre.x, BoundingSphereCentre.y,

BoundingSphereCentre.z, 0.0f);

Offset = XMVector3Transform(Offset, WorldSphere);

return Offset;

}

bool GameObject::CheckCollision(GameObject\*& TargetObject)

{

if (TargetObject == this)

{

return false;

}

XMVECTOR ThisGameObjectBoundingSphereWorldSpacePosition =

GetBoundingSphereWorldSpacePosition();

XMVECTOR TargetObjectBoundingSphereWorldSpacePosition =

TargetObject->GetBoundingSphereWorldSpacePosition();

float DistanceBetweenWorldSpacePositionsSquared =

CalculateDistanceSquaredBetweenBoundingSpheres(

ThisGameObjectBoundingSphereWorldSpacePosition,

TargetObjectBoundingSphereWorldSpacePosition);

float BoundingSphereRadiiSum = GetBoundingSphereRadius() +

TargetObject->GetBoundingSphereRadius();

bool Collision = DistanceBetweenWorldSpacePositionsSquared <=

BoundingSphereRadiiSum;

return DistanceBetweenWorldSpacePositionsSquared <=

BoundingSphereRadiiSum;

}

// Find the square distance between this object's bounding-sphere's

// centre point and TargetVector:

float GameObject::CalculateDistanceSquaredBetweenBoundingSpheres(

XMVECTOR ThisGameObjectBoundingSphereWorldSpacePosition,

XMVECTOR TargetBoundingSphereWorldSpacePosition)

{

return pow(ThisGameObjectBoundingSphereWorldSpacePosition.x -

TargetBoundingSphereWorldSpacePosition.x, 2) +

pow(ThisGameObjectBoundingSphereWorldSpacePosition.y -

TargetBoundingSphereWorldSpacePosition.y, 2) +

pow(ThisGameObjectBoundingSphereWorldSpacePosition.z -

TargetBoundingSphereWorldSpacePosition.z, 2);

}

void GameObject::Yaw(float RotationValue)

{

RotationVector.y += RotationValue;

MaintainYawRange();

}

void GameObject::Pitch(float RotationValue)

{

RotationVector.x += RotationValue;

MaintainPitchRange();

}

void GameObject::MaintainPitchRange()

{

// Keep this rotation value in the -90 to 90 range;

if (RotationVector.x > MAXIMUM\_PITCH)

{

RotationVector.x = MAXIMUM\_PITCH;

}

if (RotationVector.x < MINIMUM\_PITCH)

{

RotationVector.x = MINIMUM\_PITCH;

}

}

void GameObject::MaintainYawRange()

{

// Keep this rotation value in the 0 to 360 range:

if (RotationVector.y > MAXIMUM\_YAW)

{

RotationVector.y -= MAXIMUM\_YAW;

}

if (RotationVector.y < MINIMUM\_YAW)

{

RotationVector.y += MAXIMUM\_YAW;

}

}

// Get functions:

float GameObject::GetBoundingSphereRadius()

{

// One can use x, y or z for this calculation:

return BoundingSphereRadius \* ObjectScaleVector.x;

}

float GameObject::GetScale()

{

return GameObjectScale;

}

XMVECTOR& GameObject::GetVectorPosition()

{

return PositionVector;

}

XMFLOAT3& GameObject::GetPositionFloat3()

{

return PositionFloat3;

}

XMVECTOR& GameObject::GetVectorRotation()

{

return RotationVector;

}

XMMATRIX& GameObject::GetRotationMatrix()

{

return RotationMatrix;

}

std::string& GameObject::GetObjectModelFileName()

{

return ObjectModelReference->filename;

}

XMVECTOR& GameObject::GetDirectionalLightShinesFrom()

{

return DirectionalLightShinesFrom;

}

XMVECTOR& GameObject::GetDirectionalLightColour()

{

return DirectionalLightColour;

}

XMVECTOR& GameObject::GetAmbientLightColour()

{

return AmbientLightColour;

}

## GameScene.cpp

#include "GameScene.h"

#include "MoveableObstacle.h"

#include <algorithm>

// Initialise:

GameScene::GameScene(std::vector<GameObject\*>& NewSceneObjects,

int NewSceneID, bool SceneIsActive,

ID3D11DeviceContext\* NewImmediateContextReference)

{

SceneObjects = NewSceneObjects;

UniqueSceneID = NewSceneID;

IsSceneActive = SceneIsActive;

ImmediateContextReference = NewImmediateContextReference;

PlayerHasWon = false;

PlayerIsAlive = true;

}

// Clean-up:

GameScene::~GameScene()

{

}

// Update this scene:

HRESULT GameScene::UpdateGameScene(ID3D11RenderTargetView\* BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference, ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference,

ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference,

DirectXSystem\*& DirectXSystemHandleReference)

{

ManageCollisionChecking();

HRESULT ResultHandle = RenderFrame(

BackBufferRenderTargetViewReference,

SwapChainReference, VertexBufferReference,

ZBufferReference,

ConstantBuffer0Reference, WindowClassHandleReference,

DirectXSystemHandleReference);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

return ResultHandle;

}

bool GameScene::GetIsSceneActive()

{

return IsSceneActive;

}

bool GameScene::GetPlayerIsAlive()

{

return PlayerIsAlive;

}

bool GameScene::GetPlayerHasWon()

{

return PlayerHasWon;

}

int GameScene::GetUniqueSceneID()

{

return UniqueSceneID;

}

XMMATRIX& GameScene::GetViewMatrix()

{

return View;

}

XMMATRIX& GameScene::GetProjectionMatrix()

{

return Projection;

}

void GameScene::DrawObjects(XMMATRIX\* View, XMMATRIX\* Projection,

DirectXSystem\*& DirectXSystemHandleReference)

{

HRESULT ResultHandle = S\_OK;

for each (GameObject\* CurrentObject in SceneObjects)

{

// Check what GameObject CurrentObject is...

if (CurrentObject->GetObjectModelFileName() ==

PLAYER\_FILE\_PATH)

{

ImmediateContextReference->PSSetShaderResources(0, 1, &DirectXSystemHandleReference

->GetPlayerHoverTankTextureReference());

}

if (CurrentObject->GetObjectModelFileName() ==

STATIC\_ROCK\_FILE\_PATH)

{

ImmediateContextReference->PSSetShaderResources(0, 1, &DirectXSystemHandleReference

->GetStaticRockTextureReference());

}

if (CurrentObject->GetObjectModelFileName() ==

MOVEABLE\_WOODEN\_BARREL\_FILE\_PATH)

{

ImmediateContextReference->PSSetShaderResources(0, 1, &DirectXSystemHandleReference

->GetWoodenBarrelTextureReference());

}

if (CurrentObject->GetObjectModelFileName() ==

COLLECTABLE\_ENERGY\_CAPSULE\_FILE\_PATH)

{

ImmediateContextReference->PSSetShaderResources(0, 1, &DirectXSystemHandleReference->

GetEnergyCapsuleTextureReference());

}

if (CurrentObject->GetObjectModelFileName() ==

ENEMY\_FILE\_PATH)

{

ImmediateContextReference->PSSetShaderResources(0, 1, &DirectXSystemHandleReference->

GetEnemyHoverTankTextureReference());

}

if (!PlayerHoverTankReference->GetPlayerIsAlive())

{

PlayerIsAlive = false;

}

if (PlayerHoverTankReference->GetPlayerHasWon())

{

PlayerHasWon = true;

}

// ...before drawing it:

CurrentObject->Draw(View, Projection);

}

}

HRESULT GameScene::RenderFrame(ID3D11RenderTargetView\*

BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference,

ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference,

ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference,

DirectXSystem\*& DirectXSystemHandleReference)

{

HRESULT ResultHandle = S\_OK;

if (!PlayerHoverTankReference)

{

GetPlayerHoverTankReference();

}

if (EnemyHoverTanksReference.size() ==

0u)

{

GetEnemyHoverTankReferences();

}

UpdateEnemyHoverTanks();

ImmediateContextReference = DirectXSystemHandleReference->GetImmediateContextReference();

if (BackBufferRenderTargetViewReference && ImmediateContextReference)

{

ImmediateContextReference->ClearRenderTargetView(BackBufferRenderTargetViewReference, DirectXSystemHandleReference->DEFAULT\_CLEAR\_COLOUR);

}

// 'Set the render target view'

ImmediateContextReference->ClearDepthStencilView(ZBufferReference, D3D11\_CLEAR\_DEPTH | D3D11\_CLEAR\_STENCIL, 1.0f, 0);

ImmediateContextReference->PSSetSamplers(0, 1, &DirectXSystemHandleReference->GetDefaultTextureSamplerReference());

XMMATRIX CameraRotationMatrix = XMMATRIX();

SetupCustomWorldViewProjectionMatrix(Projection, World, View,

PlayerHoverTankReference->GetVectorPosition(), CameraRotationMatrix,

WindowClassHandleReference, DirectXSystemHandleReference);

DrawObjects(&View, &Projection, DirectXSystemHandleReference);

// Display what has just been rendered

SwapChainReference->Present(0, 0);

return ResultHandle;

}

/\*\*

NOTE: Resolved issues with collision by using dynamic\_cast,

instead of static\_cast (to get the right type of obstacle).

\*/

// Check to see if any objects are colliding:

void GameScene::ManageCollisionChecking()

{

for (UINT ObjectIterator = 0u; ObjectIterator <

SceneObjects.size(); ObjectIterator++)

{

// Only check for collision between controlled objects/moveable obstacles

// and any other objects (as static obstacles will not collide with other

// objects):

GameObject\* CurrentObject = SceneObjects[ObjectIterator];

if (CurrentObject && CurrentObject->ObjectModelFileReferenceValid() &&

(CurrentObject->GetObjectModelFileName() == PLAYER\_FILE\_PATH ||

CurrentObject->GetObjectModelFileName() == ENEMY\_FILE\_PATH))

{

for (UINT Iterator = 0u; Iterator < SceneObjects.size();

Iterator++)

{

if (SceneObjects[Iterator] != CurrentObject)

{

if (CurrentObject->CheckCollision(SceneObjects[Iterator]))

{

// Pick-up this object:

if (RepositionGameObject(CurrentObject, SceneObjects[Iterator]))

{

// The object at Iterator position in SceneObjects is a CollectableObject,

// so clean it up (erase the pointer at Iterator position):

SceneObjects.erase(std::remove(SceneObjects.begin(), SceneObjects.end(),

SceneObjects[Iterator]), SceneObjects.end());

/\*\*

Above collection clean-up method was found at:

https://stackoverflow.com/questions/3385229/c-erase-vector-element-by-value-rather-than-by-position

\*/

// Add to the amount of capsules the Player has collected:

PlayerHoverTankReference->CapsuleCollected();

}

}

}

}

}

}

}

// Reposition the OffendingObject to a position that is not colliding

// with VictimObject (or remove the VictimObject if it is a collectable):

bool GameScene::RepositionGameObject(GameObject\*& OffendingObject,

GameObject\*& VictimObject)

{

// The return value:

bool VictimObjectIsCollectableObject = false;

// Push the OffendingObject away from the VictimObject:

if (dynamic\_cast<StaticObstacle\*>(VictimObject))

{

PushBackControlledObject(OffendingObject, VictimObject);

}

// Push the VictimObject away from the OffendingObject:

if (dynamic\_cast<MoveableObstacle\*>(VictimObject))

{

PushAwayMoveableObstacle(OffendingObject, VictimObject);

}

// Push the OffendingObject away from the VictimObject

// (EnemyHoverTank away from the PlayerHoverTank):

if (dynamic\_cast<PlayerHoverTank\*>(VictimObject))

{

PushBackControlledObject(OffendingObject, VictimObject);

// Damage the Player:

PlayerHoverTankReference->ModifyHealth(dynamic\_cast<EnemyHoverTank\*>

(OffendingObject)->ENEMY\_DAMAGE\_TO\_PLAYER);

}

// Remove VictimObject from the scene (if the Player

// has collided with it):

if (dynamic\_cast<CollectableObject\*>(VictimObject) &&

dynamic\_cast<PlayerHoverTank\*>(OffendingObject))

{

VictimObjectIsCollectableObject = true;

}

return VictimObjectIsCollectableObject;

}

// Push back the controlled-object (the OffendingObject has collided

// with a static-object):

void GameScene::PushBackControlledObject(GameObject\*& OffendingObject,

GameObject\*& VictimObject)

{

ControlledObject\* OffendingControlledObject = dynamic\_cast

<ControlledObject\*>(OffendingObject);

// Repel the offending object:

StaticObstacle\* StaticObstacleVictim = dynamic\_cast<StaticObstacle\*>(VictimObject);

if (StaticObstacleVictim)

{

OffendingControlledObject->RepelControlledObject(StaticObstacleVictim->

REPULSION\_FORCE\_MAGNITUDE);

return;

}

if (VictimObject == PlayerHoverTankReference)

{

OffendingControlledObject->RepelControlledObject(PlayerHoverTankReference

->REPULSION\_FORCE\_MAGNITUDE);

}

}

// Push back the VictimObject (the OffendingObject has collided with a

// MoveableObstacle:

void GameScene::PushAwayMoveableObstacle(GameObject\*& OffendingObject,

GameObject\*& VictimObject)

{

dynamic\_cast<MoveableObstacle\*>(VictimObject)->

OnImpact(dynamic\_cast<ControlledObject\*>(OffendingObject)

->GetObjectMovementDirection());

}

void GameScene::SetupCustomWorldViewProjectionMatrix(XMMATRIX& Projection,

XMMATRIX& World, XMMATRIX& View, const XMVECTOR& NewTranslation,

const XMMATRIX & NewRotation, WindowClass \*& WindowClassHandleReference,

DirectXSystem \*& DirectXSystemHandleReference)

{

FLOAT SineFov;

FLOAT CoSineFov;

FLOAT Height;

FLOAT Width;

FLOAT HalfDefaultFOV = XMConvertToRadians(DEFAULT\_FIELD\_OF\_VIEW) \* 0.50f;

FLOAT FOVAspectRatio = WindowClassHandleReference->GetFOVAspectRatio();

const FLOAT NEAR\_Z = 1.0f;

const FLOAT FAR\_Z = 1000.0f;

XMMATRIX PerspectiveProjectionMatrix;

// Modifiy the rotation first, before translating the object:

World \*= XMMatrixTranslationFromVector(NewTranslation);

World \*= NewRotation;

// Get the cosine and sine of the FOV first...

XMScalarSinCos(&SineFov, &CoSineFov, HalfDefaultFOV);

Height = CoSineFov / SineFov;

Width = Height / FOVAspectRatio;

PerspectiveProjectionMatrix.r[0] = XMVectorSet(Width, 0.0f, 0.0f, 0.0f);

PerspectiveProjectionMatrix.r[1] = XMVectorSet(0.0f, Height, 0.0f, 0.0f);

PerspectiveProjectionMatrix.r[2] = XMVectorSet(0.0f, 0.0f, FAR\_Z / (FAR\_Z - NEAR\_Z), 1.0f);

PerspectiveProjectionMatrix.r[3] = XMVectorSet(0.0f, 0.0f, -(PerspectiveProjectionMatrix.r[2].vector4\_f32[2] \* NEAR\_Z), 0.0f);

Projection = PerspectiveProjectionMatrix;

View = PlayerHoverTankReference->GetPlayerCameraReference()->GetViewMatrix();

DirectXSystemHandleReference->UpdateConstantBuffer(Projection, World,

View, PlayerHoverTankReference->GetDirectionalLightShinesFrom(),

PlayerHoverTankReference->GetDirectionalLightColour(),

PlayerHoverTankReference->GetAmbientLightColour());

}

// Set-up a reference to the Player's hover-tank:

void GameScene::GetPlayerHoverTankReference()

{

for each (GameObject\* CurrentObject in SceneObjects)

{

if (CurrentObject->GetObjectModelFileName() ==

PLAYER\_FILE\_PATH)

{

PlayerHoverTankReference =

dynamic\_cast<PlayerHoverTank\*>(CurrentObject);

return;

}

}

}

// Set-up references to Enemy hover-tanks:

void GameScene::GetEnemyHoverTankReferences()

{

for each (GameObject\* CurrentObject in SceneObjects)

{

if (CurrentObject->GetObjectModelFileName() ==

ENEMY\_FILE\_PATH)

{

EnemyHoverTanksReference.push\_back(

dynamic\_cast<EnemyHoverTank\*>(CurrentObject));

}

}

}

void GameScene::UpdateEnemyHoverTanks()

{

for each (EnemyHoverTank\* ThisEnemy in EnemyHoverTanksReference)

{

if (ThisEnemy)

{

ThisEnemy->ManageAIState(PlayerHoverTankReference);

}

}

}

## Main.cpp

#include <windows.h>

#include <Windows.h>

#include <d3d11.h>

#include <d3dx11.h>

#include <dxerr.h>

/\*\*

RESOLUTIONS NOTED HERE:

1. To resolve macro redefinition warnings, add $(WindowsSDK\_IncludePath)

as the first include directory in VC++ Include Directories, within the

project settings.

2. For 'Warning C4838 conversion from 'unsigned int' to 'INT' requires a narrowing conversion'.

Philip says to ignore these warnings if they are pointing to a DirectX related file (in respects

to an outdated library), but if these warnings point to your code, cast any values setting INT

values to unsigned (using 'u')).

3. Resolved Live Object/Producer warnings, by following the tutorial here:

http://masterkenth.com/directx-leak-debugging/.

4. Resolved models not being rendered as they appear in 3DSMax,

by changing the primitive topology from TRIANGLESTRIP to

TRIANGLELIST.

\*/

/\*\*

TASK LIST:

1. Make sure this project runs on multiple workstations, given different settings.

2. Note process for resolving issue with model not facing the camera.

3. Note my use of pointer references in the report (why, how etc.).

4. Attempt to resolve issues with translation depending on orientation

by determining the forward vector given an object's oreintation, then using

that to work out the other directions

\*/

#define \_XM\_NO\_INTRINSICS\_

#define XM\_NO\_ALIGNMENT

#include <xnamath.h>

/\*\* To use instead of an array of characters. \*/

#include <string>

/\*\*

A legacy file for references, that still holds on to certain values (e.g. for

virtual key-codes).

\*/

#include "GlobalReferences.h"

/\*\* For the Player's perspective. \*/

#include "Camera.h"

/\*\* This class encapsulates the win32 behavior. \*/

#include "WindowClass.h"

/\*\* To handle setting-up and rendering via DirectX 11. \*/

#include "DirectXSystem.h"

/\*\* For scene mangement. \*/

#include "SceneManager.h"

/\*\* For the Player. \*/

#include "PlayerHoverTank.h"

/\*\* For enemies. \*/

#include "EnemyHoverTank.h"

/\*\* For static-obstacles. \*/

#include "StaticObstacle.h"

/\*\*

For assertions (add '#define NDEBUG' above this include

line, to disable the 'assert' macro).

Reference source:

http://www.cplusplus.com/reference/cassert/assert/

\*/

#include <assert.h>

/\*\*

DirectX related pointers are declared here.

\*/

/\*\* For window handling. \*/

WindowClass\* WindowClassHandle;

/\*\* For handling DirectX. \*/

DirectXSystem\* DirectXSystemHandle;

/\*\*

To manage the scenes of objects that

are shown to the Player.

\*/

SceneManager\* SceneManagerHandle;

/\*\* For the Player's in-game representation. \*/

PlayerHoverTank\* PlayerHoverTankReference;

/\*\* For the in-game representation of enemies. \*/

std::vector<EnemyHoverTank\*> EnemyHoverTanksReference;

DirectXSystem::NonControlledSceneObjects\* NonControlledScene0Objects;

/\*\* To initialise the SceneManager. \*/

std::vector<SceneManager::SceneComponents\*> GameSceneValues;

/\*\* Incremented by 1 for each scene. \*/

int SceneIDCounter = 0;

/\*\*

End of DirectX related pointer delcaration.

\*/

//////////////////////////////////////////////////////////////////////////////////////

// Entry point to the program. Initializes everything and goes into a message processing

// loop. Idle time is used to render the scene.

//////////////////////////////////////////////////////////////////////////////////////

int WINAPI WinMain(HINSTANCE InstanceHandle, HINSTANCE PreviousInstanceHandle, LPSTR lpCmdLine, int nCmdShow)

{

// For certain function calls:

HRESULT ResultHandle = S\_OK;

UNREFERENCED\_PARAMETER(PreviousInstanceHandle);

UNREFERENCED\_PARAMETER(lpCmdLine);

// Initialise bespoke class pointers:

DefaultProgramConstructor();

// Stop here if any of the pointers

// are invalid:

assert(WindowClassHandle);

assert(DirectXSystemHandle);

if (FAILED(WindowClassHandle->InitialiseWindow(InstanceHandle, nCmdShow, WndProc)))

{

DXTRACE\_MSG("Failed to create Window");

return 0;

}

if (FAILED(DirectXSystemHandle->InitialiseD3D(WindowClassHandle->

GetWindowHandle())))

{

DXTRACE\_MSG("Failed to create Device");

return 0;

}

if (FAILED(DirectXSystemHandle->InitialiseGraphics(

PlayerHoverTankReference, NonControlledScene0Objects,

EnemyHoverTanksReference)))

{

DXTRACE\_MSG("Failed to initialise graphics");

return 0;

}

// As these are initialised via DirectXSystem::InitialiseGraphics():

assert(PlayerHoverTankReference->GetPlayerCameraReference());

assert(PlayerHoverTankReference);

for each (EnemyHoverTank\* CurrentTank in EnemyHoverTanksReference)

{

assert(CurrentTank);

}

// For one scene at the moment:

std::vector<GameObject\*> DefaultSceneObjects = { PlayerHoverTankReference };

// Add of the Enemy hover-tanks into this collection as well:

for each (EnemyHoverTank\* CurrentTank in EnemyHoverTanksReference)

{

DefaultSceneObjects.push\_back(CurrentTank);

}

// Static obstacles:

for (UINT Iterator = 0u; Iterator < NonControlledScene0Objects->

StaticObstaclesReference.size(); Iterator++)

{

if (NonControlledScene0Objects->StaticObstaclesReference[Iterator])

{

DefaultSceneObjects.push\_back(NonControlledScene0Objects->

StaticObstaclesReference[Iterator]);

}

}

// Moveable obstacles:

for (UINT Iterator = 0u; Iterator < NonControlledScene0Objects->

MoveableObstaclesReference.size(); Iterator++)

{

if (NonControlledScene0Objects->MoveableObstaclesReference[Iterator])

{

DefaultSceneObjects.push\_back(NonControlledScene0Objects->

MoveableObstaclesReference[Iterator]);

}

}

// Collectable objects:

for (UINT Iterator = 0u; Iterator < NonControlledScene0Objects->

CollectableObjectsReference.size(); Iterator++)

{

if (NonControlledScene0Objects->CollectableObjectsReference[Iterator])

{

DefaultSceneObjects.push\_back(NonControlledScene0Objects->

CollectableObjectsReference[Iterator]);

}

}

GameSceneValues.push\_back(new SceneManager::

SceneComponents(DefaultSceneObjects, SceneIDCounter,

true));

SceneManagerHandle = new SceneManager(

DirectXSystemHandle->GetImmediateContextReference(),

GameSceneValues);

assert(SceneManagerHandle);

// Main message loop

MSG PrimaryMessage = { 0 };

while (PrimaryMessage.message != WM\_QUIT)

{

if (PeekMessage(&PrimaryMessage, NULL, 0, 0, PM\_REMOVE))

{

TranslateMessage(&PrimaryMessage);

DispatchMessage(&PrimaryMessage);

}

else

{

if (FAILED(SceneManagerHandle->UpdateGameScenes(DirectXSystemHandle->

GetBackBufferRenderTargetViewReference(),

DirectXSystemHandle->GetSwapChainReference(), DirectXSystemHandle->

GetVertexBufferReference(),

DirectXSystemHandle->GetZBufferReference(),

DirectXSystemHandle->GetConstantBuffer0Reference(),

WindowClassHandle, DirectXSystemHandle)))

{

DXTRACE\_MSG("Failed to update scenes");

return 0;

}

// Player wins:

if (SceneManagerHandle->GetDefaultGameScene()->GetPlayerHasWon())

{

// For the splash screen to use:

const XMVECTOR TARGET\_POSITION = PlayerHoverTankReference->GetVectorPosition() + XMVectorSet(0.0f, 0.0f, 40.0f, 0.0f);

XMFLOAT3 TargetPositionFloat3 = XMFLOAT3();

XMStoreFloat3(&TargetPositionFloat3, TARGET\_POSITION);

// Show the Loss indicator to the Player:

GameObject\* PlayerVictorySplashScreen = new GameObject(DirectXSystemHandle->

GetD3DDeviceReference(), DirectXSystemHandle->GetImmediateContextReference(),

ResultHandle, TargetPositionFloat3);

// Leave the function now:

if (PlayerVictorySplashScreen)

{

ResultHandle = PlayerVictorySplashScreen->LoadObjectModel(

GENERIC\_SPLASH\_SCREEN\_FILE\_PATH);

PlayerVictorySplashScreen->Draw(&SceneManagerHandle->GetDefaultGameScene()->GetViewMatrix(),

&SceneManagerHandle->GetDefaultGameScene()->GetProjectionMatrix());

Sleep(VICTORY\_LOSS\_FREEZE\_TIME);

PrimaryMessage.message = WM\_QUIT;

// No longer required:

delete PlayerVictorySplashScreen;

PlayerVictorySplashScreen = nullptr;

}

}

// Player loses:

if (!SceneManagerHandle->GetDefaultGameScene()->GetPlayerIsAlive())

{

// For the splash screen to use:

const XMVECTOR TARGET\_POSITION = PlayerHoverTankReference->GetVectorPosition() + XMVectorSet(0.0f, 0.0f, 40.0f, 0.0f);

XMFLOAT3 TargetPositionFloat3 = XMFLOAT3();

XMStoreFloat3(&TargetPositionFloat3, TARGET\_POSITION);

// Show the Loss indicator to the Player:

GameObject\* PlayerLossSplashScreen = new GameObject(DirectXSystemHandle->

GetD3DDeviceReference(), DirectXSystemHandle->GetImmediateContextReference(),

ResultHandle, TargetPositionFloat3);

// Leave the function now:

if (PlayerLossSplashScreen)

{

ResultHandle = PlayerLossSplashScreen->LoadObjectModel(

GENERIC\_SPLASH\_SCREEN\_FILE\_PATH);

PlayerLossSplashScreen->Draw(&SceneManagerHandle->GetDefaultGameScene()->GetViewMatrix(),

&SceneManagerHandle->GetDefaultGameScene()->GetProjectionMatrix());

Sleep(VICTORY\_LOSS\_FREEZE\_TIME);

PrimaryMessage.message = WM\_QUIT;

// No longer required:

delete PlayerLossSplashScreen;

PlayerLossSplashScreen = nullptr;

}

}

}

}

// For closing Direct3D correctly:

ShutdownD3D();

return (int) PrimaryMessage.wParam;

}

// Initialise this application:

void DefaultProgramConstructor()

{

WindowClassHandle = new WindowClass();

DirectXSystemHandle = new DirectXSystem();

}

//////////////////////////////////////////////////////////////////////////////////////

// Called every time the application receives a message

//////////////////////////////////////////////////////////////////////////////////////

LRESULT CALLBACK WndProc(HWND WindowHandle, UINT Message, WPARAM WParam, LPARAM LParam)

{

// For checking to see if the message is not equal to the values accounted for

// in the switch statement of WindowsProcedureLogic():

HRESULT ResultHandle = WindowClassHandle->WindowsProcedureLogic

(WindowHandle, Message, WParam, LParam, DirectXSystemHandle,

PlayerHoverTankReference);

if (ResultHandle == E\_FAIL)

{

return DefWindowProc(WindowHandle, Message, WParam, LParam);

}

return 0;

}

//////////////////////////////////////////////////////////////////////////////////////

// Clean up D3D objects:

//////////////////////////////////////////////////////////////////////////////////////

void ShutdownD3D()

{

if (PlayerHoverTankReference) delete PlayerHoverTankReference; PlayerHoverTankReference = nullptr;

for each (EnemyHoverTank\* CurrentEnemy in EnemyHoverTanksReference)

{

if (CurrentEnemy)

{

delete CurrentEnemy;

CurrentEnemy = nullptr;

}

}

for each (StaticObstacle\* CurrentObstacle in NonControlledScene0Objects->StaticObstaclesReference)

{

if (CurrentObstacle)

{

delete CurrentObstacle;

CurrentObstacle = nullptr;

}

}

for each(MoveableObstacle\* CurrentObstacle in NonControlledScene0Objects->MoveableObstaclesReference)

{

if (CurrentObstacle)

{

delete CurrentObstacle;

CurrentObstacle = nullptr;

}

}

for each (CollectableObject\* CurrentCollectable in NonControlledScene0Objects->CollectableObjectsReference)

{

if (CurrentCollectable)

{

delete CurrentCollectable;

CurrentCollectable = nullptr;

}

}

if (NonControlledScene0Objects) delete NonControlledScene0Objects; NonControlledScene0Objects = nullptr;

if (SceneManagerHandle) delete SceneManagerHandle; SceneManagerHandle = nullptr;

if (DirectXSystemHandle) delete DirectXSystemHandle; DirectXSystemHandle = nullptr;

if (WindowClassHandle) delete WindowClassHandle; WindowClassHandle = nullptr;

}

## MoveableObstacle.cpp

#include "MoveableObstacle.h"

// Initialise (using an initialiser list)

MoveableObstacle::MoveableObstacle(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference,

XMFLOAT3& InitialPosition)

: GameObject(NewD3DDeviceReference, NewD3DDeviceContextReference, ResultHandleReference,

InitialPosition)

{

}

// Clean-up:

MoveableObstacle::~MoveableObstacle()

{

}

// When this obstacle is hit:

void MoveableObstacle::OnImpact(XMVECTOR& ImpactDirection)

{

// Move the obstacle:

PositionVector += PUSH\_FORCE\_MAGNITUDE \* ImpactDirection;

}

## objfilemodel.cpp

// turn off fopen warnings

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "ObjFileModel.h"

// draw object

void ObjFileModel::Draw(void)

{

UINT stride = sizeof(MODEL\_POS\_TEX\_NORM\_VERTEX);

UINT offset = 0;

pImmediateContext->IASetVertexBuffers(0, 1, &pVertexBuffer, &stride, &offset);

pImmediateContext->Draw(numverts, 0);

}

// load object from obj file in constructor

ObjFileModel::ObjFileModel(char\* fname, ID3D11Device\* device, ID3D11DeviceContext\* context)

{

pD3DDevice = device;

pImmediateContext = context;

if(loadfile(fname)==0)

{

// file not loaded, check debug output;

filename="FILE NOT LOADED";

return;

}

filename = fname;

parsefile();

createVB();

delete[] fbuffer; // delete file buffer created in loadfile()

}

// load wavefront object file. adds terminating \n so last line of file can be correctly parsed as a 'line' later

// basic loader - only deals with vertices v, texcoords vt, normals vn

// - only copes with triangular meshes (no quads)

// - doesn't deal with textures or materials

int ObjFileModel::loadfile(char\* fname)

{

FILE\* pFile;

// Failed to open model file ERROR OCCURS HERE, RESOLVE THIS:

pFile = fopen(fname , "r"); // if changed to bin format will read carriage return \r (0d) as well as \n (0a) into fbuffer, may need to add \r checks(but seemed to work with basic test)

if (pFile==NULL) { DXTRACE\_MSG("Failed to open model file");DXTRACE\_MSG(fname); return 0 ;}

// get file size

fseek(pFile, 0, SEEK\_END);

fbuffersize = ftell(pFile);

rewind(pFile);

// allocate memory for entire file size

fbuffer = new char[fbuffersize+1]; // 1 added to cope with adding a \n later in case file doesn't end with \n

if (fbuffer == NULL) {fclose(pFile); DXTRACE\_MSG("Failed allocate memory for model file");DXTRACE\_MSG(fname); return 0 ;}

// copy file into memory

actualsize = fread(fbuffer,1,fbuffersize,pFile); // actualsize may be less than fbuffersize in text mode as \r are stripped

if (actualsize == 0) {fclose(pFile); DXTRACE\_MSG("Failed to read model file");DXTRACE\_MSG(fname); return 0 ;}

// add a newline at end in case file does not, so can deal with whole buffer as a set of lines of text

fbuffer[actualsize] = '\n'; fclose(pFile);

return 1;

}

// uses concept of getting parsable tokens seperated by whitespace and '/'

// one line of file is parsed at a time, lines seperated by '\n'

void ObjFileModel::parsefile()

{

tokenptr=0; // token pointer points to first element of buffer

int tokenstart, tokenlength;

xyz tempxyz;

xy tempxy;

bool success;

int line=0;

do

{

line++; // keep track of current line number for error reporting

if(!getnexttoken(tokenstart, tokenlength)) continue; // get first token on line, go to next line if first token is \n

// ADD FURTHER KEYWORDS HERE TO EXTEND CAPABILITIES

if(strncmp(&fbuffer[tokenstart], "v ", 2)==0) // VERTEX POSITION - note the space in the string is needed (see vt, etc)

{

success=true; // used to see if correct number of tokens left on line for this type of attribute

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.x = (float) atof(&fbuffer[tokenstart]);

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.y = (float) atof(&fbuffer[tokenstart]);

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.z = (float) atof(&fbuffer[tokenstart]);

// if not correct number of tokens, display error in debug output

if(!success) {char s[100] = "ERROR: Badly formatted vertex, line : "; \_itoa(line, &s[strlen(s)], 10); strcat(s, " : "); strcat(s, filename.c\_str()); DXTRACE\_MSG(s); }

position\_list.push\_back(tempxyz); // add a new element to the list

}

else if(strncmp(&fbuffer[tokenstart], "vt", 2)==0) // TEXTURE COORDINATES

{

success=true;

success = success && getnexttoken(tokenstart, tokenlength);

tempxy.x = (float) atof(&fbuffer[tokenstart]);

success = success && getnexttoken(tokenstart, tokenlength);

tempxy.y = (float) atof(&fbuffer[tokenstart]);

if(!success) {char s[100] = "ERROR: Badly formatted texture coordinate, line : "; \_itoa(line, &s[strlen(s)], 10); strcat(s, " : "); strcat(s, filename.c\_str()); DXTRACE\_MSG(s); }

texcoord\_list.push\_back(tempxy);

}

else if(strncmp(&fbuffer[tokenstart], "vn", 2)==0) // NORMALS

{

success=true;

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.x = (float) atof(&fbuffer[tokenstart]);

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.y = (float) atof(&fbuffer[tokenstart]);

success = success && getnexttoken(tokenstart, tokenlength);

tempxyz.z = (float) atof(&fbuffer[tokenstart]);

if(!success) {char s[100] = "ERROR: Badly formatted normal, line : "; \_itoa(line, &s[strlen(s)], 10); strcat(s, " : "); strcat(s, filename.c\_str()); DXTRACE\_MSG(s); }

normal\_list.push\_back(tempxyz);

}

else if(strncmp(&fbuffer[tokenstart], "f ", 2)==0) // FACE - only deals with triangles so far

{

int tempptr = tokenstart + 2; // skip "f "

int forwardslashcount=0;

bool adjacentslash = false;

// this works out how many elements are specified for a face, e.g.

// f 1 2 3 -> 0 forward slashes = just position

// f 1/1 2/2 3/3 -> 3 slashes = position and texcoords

// f 1/1/1 2/2/2 3/3/3 -> 6 slashes = position, texcoords, normals

// f 1//1 2//2 3//3 -> 6 slashes with adjacent = position, normals

while(fbuffer[tempptr] != '\n')

{

if(fbuffer[tempptr] == '/')

{

forwardslashcount++;

if(fbuffer[tempptr-1] == '/') adjacentslash=true;

}

tempptr++;

}

success=true;

// Get 3 sets of indices per face

for(int i=0; i<3; i++)

{

// get vertex index

success = success && getnexttoken(tokenstart, tokenlength);

pindices.push\_back(atoi(&fbuffer[tokenstart]));

if(forwardslashcount>=3&& adjacentslash==false) // get texcoord index if required

{

success = success && getnexttoken(tokenstart, tokenlength);

tindices.push\_back(atoi(&fbuffer[tokenstart]));

}

if(forwardslashcount==6 || adjacentslash==true) // get normal index if required

{

success = success && getnexttoken(tokenstart, tokenlength);

nindices.push\_back(atoi(&fbuffer[tokenstart]));

}

}

if(!success) {char s[100] = "ERROR: Badly formatted face, line : "; \_itoa(line, &s[strlen(s)], 10); strcat(s, " : "); strcat(s, filename.c\_str()); DXTRACE\_MSG(s); }

}

} while(getnextline() == true);

}

// get next token. if \n is next token do not proceed, use getnextline() to resume

bool ObjFileModel::getnexttoken(int& tokenstart, int& tokenlength)

{

tokenstart = tokenptr;

tokenlength=1;

int tokenend;

while(fbuffer[tokenptr] == ' ' || fbuffer[tokenptr] == '\t' || fbuffer[tokenptr] == '/') tokenptr++; //skip whitespace and '/'

if(fbuffer[tokenptr] == '\n') { return false; } // keeps tokenptr pointing to \n as a token to indicate end of line

// doesn't point to next token, dealt with in getnextline()

tokenend=tokenptr+1;

while(fbuffer[tokenend] != ' ' && fbuffer[tokenend] != '\t' && fbuffer[tokenend] != '\n' && fbuffer[tokenend] != '/') tokenend++; // find length of token by finding next whitespace or '\n' or '/'

tokenlength = tokenend - tokenptr;

tokenstart = tokenptr;

tokenptr+=tokenlength; //ready for next token

return true;

}

// gets next line of buffer by skipping to next element after end of current line, returns false when end of buffer exceeded

bool ObjFileModel::getnextline()

{

// relies on getnexttoken()leaving tokenptr pointing to \n if encountered

while(fbuffer[tokenptr] != '\n' && tokenptr < actualsize) tokenptr++; // skip to end of line

tokenptr++; // point to start of next line

if (tokenptr >= actualsize) return false;

else return true;

}

// create Vertex buffer from parsed file data

bool ObjFileModel::createVB()

{

// create vertex array to pass to vertex buffer from parsed data

numverts = pindices.size();

vertices = new MODEL\_POS\_TEX\_NORM\_VERTEX[numverts]; // create big enough vertex array

for(unsigned int i = 0; i< numverts; i++)

{

int vindex = pindices[i]-1; // use -1 for indices as .obj files indices begin at 1

// set position data

vertices[i].Pos.x = position\_list[vindex].x;

vertices[i].Pos.y = position\_list[vindex].y;

vertices[i].Pos.z = position\_list[vindex].z;

if(tindices.size() > 0)

{

// if there are any, set texture coord data

int tindex = tindices[i]-1;

vertices[i].TexCoord.x = texcoord\_list[tindex].x;

vertices[i].TexCoord.y = texcoord\_list[tindex].y;

}

if(nindices.size() > 0)

{

// if there are any, set normal data

int nindex = nindices[i]-1;

vertices[i].Normal.x = normal\_list[nindex].x;

vertices[i].Normal.y = normal\_list[nindex].y;

vertices[i].Normal.z = normal\_list[nindex].z;

}

}

// Set up and create vertex buffer

D3D11\_BUFFER\_DESC bufferDesc;

ZeroMemory(&bufferDesc, sizeof(bufferDesc));

bufferDesc.Usage = D3D11\_USAGE\_DYNAMIC; // Used by CPU and GPU

bufferDesc.ByteWidth = sizeof(vertices[0])\*numverts; // Total size of buffer

bufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER; // Use as a vertex buffer

bufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE; // Allow CPU access

HRESULT hr = pD3DDevice->CreateBuffer(&bufferDesc, NULL, &pVertexBuffer); // Create the buffer

if(FAILED(hr))

{

return false;

}

// Copy the vertices into the buffer

D3D11\_MAPPED\_SUBRESOURCE ms;

pImmediateContext->Map(pVertexBuffer, NULL, D3D11\_MAP\_WRITE\_DISCARD, NULL, &ms); // Lock the buffer to allow writing

memcpy(ms.pData, vertices, sizeof(vertices[0])\*numverts); // Copy the data

pImmediateContext->Unmap(pVertexBuffer, NULL); // Unlock the buffer

return true;

}

ObjFileModel::~ObjFileModel()

{

// clean up memory used by object

if(pVertexBuffer) pVertexBuffer->Release();

delete [] vertices;

vertices = nullptr;

position\_list.clear();

normal\_list.clear();

texcoord\_list.clear();

}

## PlayerHoverTank.cpp

#include "PlayerHoverTank.h"

// Structures:

struct MODEL\_CONSTANT\_BUFFER

{

XMMATRIX WorldViewProjectionMatrix; // '64 bytes ( 4 x 4 = 16 floats x 4 bytes)'

XMVECTOR DirectionalLightVector; // '16 bytes'

XMVECTOR DirectionalLightColour; // '16 bytes'

XMVECTOR AmbientLightColour; // '16 bytes'

}; // 'TOTAL SIZE = 112 bytes'

// Initialise (for default values, also use an initialiser list for such):

PlayerHoverTank::PlayerHoverTank(ID3D11Device\*& NewD3DDeviceReference, ID3D11DeviceContext\*&

NewD3DDeviceContextReference, HRESULT& ResultHandleReference, XMFLOAT3& InitialPosition)

: ControlledObject(NewD3DDeviceReference, NewD3DDeviceContextReference, ResultHandleReference,

InitialPosition)

{

PlayerCamera = new Camera();

PlayerResultHandleReference = ResultHandleReference;

EnergyCapsulesCollected = 0;

CurrentHealth = DEFAULT\_INITIAL\_HEALTH;

PlayerIsAlive = true;

PlayerHasWon = false;

}

// Clean-up:

PlayerHoverTank::~PlayerHoverTank()

{

if (PlayerCamera) delete PlayerCamera;

PlayerCamera = nullptr;

}

Camera\*& PlayerHoverTank::GetPlayerCameraReference()

{

return PlayerCamera;

}

bool PlayerHoverTank::GetPlayerHasWon()

{

return PlayerHasWon;

}

bool PlayerHoverTank::GetPlayerIsAlive()

{

return PlayerIsAlive;

}

// Handle the camera's offset as well:

void PlayerHoverTank::Draw(XMMATRIX\* View, XMMATRIX\* Projection)

{

// Also taken from the tutorial mentioned in the header file

// (of the Camera class, as parts of this function was taken

// from Camera.cpp):

// Perform degrees to radians conversions first though:

float PitchRotationRadians = XMConvertToRadians(RotationVector.x);

float YawRotationRadians = XMConvertToRadians(RotationVector.y);

float RollRotationRadians = XMConvertToRadians(RotationVector.z);

// Only handle the procedure for drawing the Player, if they

// are still alive:

if (PlayerIsAlive)

{

RotationMatrix = XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

ObjectForwardTarget = XMVector3TransformCoord(DefaultForwardDirection,

RotationMatrix);

ObjectForwardTarget = XMVector3Normalize(ObjectForwardTarget);

XMMATRIX RotateYTempMatrix = XMMatrixRotationY(YawRotationRadians);

ObjectRightDirection = XMVector3TransformCoord(DefaultRightDirection, RotateYTempMatrix);

ObjectUpDirection = XMVector3TransformCoord(DefaultUpDirection, RotateYTempMatrix);

ObjectForwardDirection = XMVector3TransformCoord(DefaultForwardDirection, RotateYTempMatrix);

// Update the position (and current movement direction), for movement:

XMVECTOR HorizontalMovementVelocity = MovementLeftRight \* ObjectRightDirection;

XMVECTOR VerticalMovementVelocity = MovementUpDown \* ObjectUpDirection;

XMVECTOR DepthMovementVelocity = MovementForwardBackwards \* ObjectForwardDirection;

PositionVector += HorizontalMovementVelocity;

PositionVector += DepthMovementVelocity;

// Set the movement-direction to HorizontalMovementVelocity

// before incrementing it by depth and height:

ObjectMovementDirection = HorizontalMovementVelocity;

ObjectMovementDirection += DepthMovementVelocity;

ObjectMovementDirection += VerticalMovementVelocity;

ObjectMovementDirection = XMVector3Normalize(ObjectMovementDirection);

// Convert this vector to FLOAT3, to store in PositionFloat3:

XMStoreFloat3(&PositionFloat3, PositionVector);

// Then set the new camera target with this:

PlayerCamera->SetTarget(PositionFloat3);

MovementLeftRight = 0.0f;

MovementForwardBackwards = 0.0f;

MovementUpDown = 0.0f;

ObjectForwardTarget += PositionVector;

// Update the World transform matrix and the buffers now:

// Scale first, then rotation, with translation as the last of

// the transformation actions to take place:

World = XMMatrixScalingFromVector(ObjectScaleVector);

World \*= XMMatrixRotationRollPitchYaw(PitchRotationRadians,

YawRotationRadians, RollRotationRadians);

World \*= XMMatrixTranslationFromVector(PositionVector);

// For lighting on this object:

XMMATRIX TransposeMatrix = XMMATRIX();

MODEL\_CONSTANT\_BUFFER ObjectConstantBufferValues;

ObjectConstantBufferValues.WorldViewProjectionMatrix = World \* (\*View) \* (\*Projection);

ObjectConstantBufferValues.DirectionalLightColour = DirectionalLightColour;

ObjectConstantBufferValues.AmbientLightColour = AmbientLightColour;

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Transform(DirectionalLightShinesFrom,

TransposeMatrix);

ObjectConstantBufferValues.DirectionalLightVector = XMVector3Normalize(ObjectConstantBufferValues.

DirectionalLightVector);

ImmediateContextReference->VSSetConstantBuffers(0u, 1u, &GameObjectConstantBufferReference);

ImmediateContextReference->UpdateSubresource(GameObjectConstantBufferReference, 0u, nullptr,

&ObjectConstantBufferValues, 0u, 0u);

// Set this model's shaders and input layout as active:

ImmediateContextReference->VSSetShader(GameObjectVertexShaderReference, nullptr, 0);

ImmediateContextReference->PSSetShader(GameObjectPixelShaderReference, nullptr, 0);

ImmediateContextReference->IASetInputLayout(GameObjectInputLayoutReference);

ObjectModelReference->Draw();

}

}

// Heal or damage the Player:

void PlayerHoverTank::ModifyHealth(int HealthModificationValue)

{

CurrentHealth -= HealthModificationValue;

// Then check if they are still alive:

if (CurrentHealth <= 0)

{

PlayerIsAlive = false;

}

}

// Increment their EnergyCapsulesCollected member:

void PlayerHoverTank::CapsuleCollected()

{

EnergyCapsulesCollected++;

// Then check if they have won:

if (EnergyCapsulesCollected == GOAL\_CAPSULE\_COUNT)

{

PlayerHasWon = true;

}

}

## SceneManager.cpp

#include "SceneManager.h"

#include "DirectXSystem.h"

#include "PlayerHoverTank.h"

// Initialise:

SceneManager::SceneManager(ID3D11DeviceContext\* NewImmediateContextReference,

std::vector<SceneComponents\*>& NewGameSceneValues)

{

// Make a new scene for each of the values in NewGameSceneValues:

for each (SceneComponents\* CurrentSceneValues in NewGameSceneValues)

{

GameScenes.push\_back(new GameScene(CurrentSceneValues->SceneObjects,

CurrentSceneValues->SceneID, CurrentSceneValues->SceneIsActive,

NewImmediateContextReference));

}

}

// Shut-Down:

SceneManager::~SceneManager()

{

for each (GameScene\* CurrentScene in GameScenes)

{

delete CurrentScene;

CurrentScene = nullptr;

}

}

HRESULT SceneManager::UpdateGameScenes(ID3D11RenderTargetView\* BackBufferRenderTargetViewReference,

IDXGISwapChain\* SwapChainReference, ID3D11Buffer\*& VertexBufferReference,

ID3D11DepthStencilView\*& ZBufferReference, ID3D11Buffer\*& ConstantBuffer0Reference,

WindowClass\*& WindowClassHandleReference, DirectXSystem\*& DirectXSystemHandleReference)

{

HRESULT ResultHandle = S\_OK;

for each (GameScene\* CurrentGameScene in GameScenes)

{

// Only update active scenes:

if (CurrentGameScene->GetIsSceneActive())

{

ResultHandle =

CurrentGameScene->UpdateGameScene(

BackBufferRenderTargetViewReference,

SwapChainReference, VertexBufferReference,

ZBufferReference, ConstantBuffer0Reference,

WindowClassHandleReference, DirectXSystemHandleReference);

if (FAILED(ResultHandle))

{

return ResultHandle;

}

}

}

return ResultHandle;

}

// Get the default game-scene (where ID = 0):

GameScene\*& SceneManager::GetDefaultGameScene()

{

for (UINT CollectionIterator = 0u; CollectionIterator <

GameScenes.size(); CollectionIterator++)

{

// 0 is the default game-scene's ID:

if (GameScenes[CollectionIterator]->

GetUniqueSceneID() == 0)

{

return GameScenes[CollectionIterator];

}

}

// Default to returning the first scene

// in the collection, if one was not found

// in the above for loop:

return GameScenes[0];

}

## StaticObstacle.cpp

#include "StaticObstacle.h"

// Initialise (using an initialiser list):

StaticObstacle::StaticObstacle(ID3D11Device\*& NewD3DDeviceReference,

ID3D11DeviceContext\*& NewD3DDeviceContextReference,

HRESULT& ResultHandleReference, XMFLOAT3& InitialPosition)

: GameObject(NewD3DDeviceReference, NewD3DDeviceContextReference,

ResultHandleReference, InitialPosition)

{

}

// Clean-up:

StaticObstacle::~StaticObstacle()

{

}

## WindowClass.cpp

#include "WindowClass.h"

#include "GlobalReferences.h"

#include "PlayerHoverTank.h"

// Initialise an instance:

WindowClass::WindowClass()

{

}

// Clean-up an instance:

WindowClass::~WindowClass()

{

if (InstanceWindowHandle)

{

delete InstanceWindowHandle;

InstanceWindowHandle = nullptr;

}

}

// Register class and create the window for DirectX to use:

HRESULT WindowClass::InitialiseWindow(HINSTANCE InstanceHandle, int nCmdShow, WNDPROC WindowsProcedure)

{

// Register class

WNDCLASSEX wcex = { 0 };

wcex.cbSize = sizeof(WNDCLASSEX);

wcex.style = CS\_HREDRAW | CS\_VREDRAW;

wcex.lpfnWndProc = WindowsProcedure;

wcex.hInstance = InstanceHandle;

wcex.hCursor = LoadCursor(NULL, IDC\_ARROW);

// wcex.hbrBackground = (HBRUSH )( COLOR\_WINDOW + 1); // Needed for non-D3D apps

// const char\* used once again:

wcex.lpszClassName = AUTHOR\_NAME.c\_str();

if (!RegisterClassEx(&wcex)) return E\_FAIL;

// Create window

InstanceHandle = InstanceHandle;

/\*\*

1>c:\users\2moraj05\downloads\james-moran-cgp600-ae2-master\windowclass.cpp(41): warning C4838: conversion from 'const FLOAT' to 'LONG' requires a narrowing conversion

1>c:\users\2moraj05\downloads\james-moran-cgp600-ae2-master\windowclass.cpp(41): warning C4244: 'initializing': conversion from 'const FLOAT' to 'LONG', possible loss of data

\*/

RECT WindowRectangle = { 0, 0, DEFAULT\_WINDOW\_WIDTH, DEFAULT\_WINDOW\_HEIGHT };

AdjustWindowRect(&WindowRectangle, WS\_OVERLAPPEDWINDOW, FALSE);

// A const char\* can be parsed in as an LPCCHAR:

WindowHandle = CreateWindow(AUTHOR\_NAME.c\_str(), WINDOW\_TITLE.c\_str(), WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, WindowRectangle.right - WindowRectangle.left,

WindowRectangle.bottom - WindowRectangle.top, NULL, NULL, InstanceHandle, NULL);

if (!WindowHandle)

{

return E\_FAIL;

}

ShowWindow(WindowHandle, nCmdShow);

return S\_OK;

}

HRESULT WindowClass::WindowsProcedureLogic(HWND WindowHandle,

UINT Message, WPARAM WParam, LPARAM LParam,

DirectXSystem\*& DirectXSystemHandleReference,

PlayerHoverTank\*& PlayerHoverTankReference)

{

PAINTSTRUCT PaintStructure;

HDC DeviceContextHandle;

switch (Message)

{

case WM\_PAINT:

DeviceContextHandle = BeginPaint(WindowHandle, &PaintStructure);

EndPaint(WindowHandle, &PaintStructure);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

case WM\_KEYDOWN:

ManageKeyPressed(WParam, PlayerHoverTankReference->

GetPlayerCameraReference(), PlayerHoverTankReference);

break;

// CHECK TO SEE IF THIS IS THE OPTIMAL WAY TO GO ABOUT RESIZING THE BUFFERS!!OGea

case WM\_SIZE:

DirectXSystemHandleReference->HandleWindowResizing(WindowHandle);

break;

default:

// A break-point was tResultHandleown on the return line, if hWnd is not valid:

// BREAKPOINT STILL THROWN AT TIMES GESU(GES[HJAD2!

if (WindowHandle)

{

// Return this result for WndProc in Main to handle:

return E\_FAIL;

}

}

return S\_OK;

}

HWND& WindowClass::GetWindowHandle()

{

return WindowHandle;

}

FLOAT WindowClass::GetFOVAspectRatio()

{

return (FLOAT) (DEFAULT\_WINDOW\_WIDTH / DEFAULT\_WINDOW\_HEIGHT);

}

void WindowClass::ManageKeyPressed(int VirtualKeyCode, Camera\*& DefaultCameraReference,

PlayerHoverTank\*& PlayerHoverTankReference)

{

switch (VirtualKeyCode)

{

case VK\_ESCAPE:

DestroyWindow(WindowHandle);

break;

/\*\*

For movement forwards, backwards, upwards

and downwards, as well as rotation leftwards,

rightwards, upwards, downwards and rolling.

\*/

case VK\_WKEY:

PlayerHoverTankReference->MoveForward(TRANSLATION\_MAGNITUDE);

break;

case VK\_AKEY:

PlayerHoverTankReference->MoveRight(-TRANSLATION\_MAGNITUDE);

break;

case VK\_SKEY:

PlayerHoverTankReference->MoveForward(-TRANSLATION\_MAGNITUDE);

break;

case VK\_DKEY:

PlayerHoverTankReference->MoveRight(TRANSLATION\_MAGNITUDE);

break;

case VK\_QKEY:

PlayerHoverTankReference->Yaw(-ROTATION\_MAGNITUDE);

break;

case VK\_EKEY:

PlayerHoverTankReference->Yaw(ROTATION\_MAGNITUDE);

break;

default:

break;

}

}

## Model\_Shaders.hlsl

/\*\*

This is the alternate shader that I put

together, to allow the GameObjects of

the game-scene, to be shown correctly.

\*/

cbuffer CB0

{

matrix WorldViewProjectionMatrix; // '64 bytes'

}; // 'TOTAL SIZE = 64 bytes'

texture2D PlayerHoverTankTexture;

SamplerState DefaultTextureSampler;

struct VOut

{

float4 Position : SV\_POSITION;

float4 Color : COLOR;

float2 UVTextureCoordinates : TEXCOORD;

};

VOut ModelVS(float4 Position : Position, float2 UVTextureCoordinates : TEXCOORD,

float3 Normal : NORMAL)

{

VOut Output;

float4 Default\_Colour = { 1.0f, 1.0f, 1.0f, 1.0f };

Output.Position = mul(WorldViewProjectionMatrix, Position);

Output.UVTextureCoordinates = UVTextureCoordinates;

Output.Color = Default\_Colour;

return Output;

}

float4 ModelPS(float4 Position : SV\_POSITION, float4 Color : COLOR,

float2 UVTextureCoordinates : TEXCOORD) : SV\_TARGET

{

return PlayerHoverTankTexture.Sample(DefaultTextureSampler, UVTextureCoordinates) \* Color;

}

## Shaders.hlsl

/\*\*

This is the original shader file I used, before changing to

Model\_Shaders.hlsl (due to lighting colour issues with this file).

\*/

texture2D PlayerHoverTankTexture;

SamplerState DefaultTextureSampler;

cbuffer CBuffer0

{

matrix WorldViewProjectionMatrix; // '64 bytes'

float Scale; // (also 4 bytes)

float3 ScalePacking; // 3x4 bytes = 12 bytes

float4 NewPosition; // 4x4 bytes = 16 bytes

bool InputProvided; // 1 byte

bool PackingInputBytes[3]; // 3 bytes (1x3)

float3 AdditionalPackingInputBytes; // 3x4 bytes = 12 bytes

float4 DirectionalLightVector; // 4x4 bytes = 16 bytes

float4 DirectionalLightColour; // 4x4 bytes = 16 bytes

float4 AmbientLightColour; // 4x4 bytes = 16 bytes

}; /\*\* Total size: 160 bytes. \*/

struct VOut

{

float4 Position : SV\_POSITION;

float4 Color : COLOR;

float2 TextureUVCoordinates : TEXCOORD;

float3 Normal : NORMAL;

};

VOut VShader(float4 Position : POSITION, float4 Color : COLOR, float2 TextureUVCoordinates : TEXCOORD, float3 Normal : NORMAL)

{

VOut Output;

Output.Position = mul(WorldViewProjectionMatrix, Position);

float DiffuseMagnitude = dot(DirectionalLightVector, Normal);

DiffuseMagnitude = saturate(DiffuseMagnitude);

Output.Color = AmbientLightColour + (DirectionalLightColour \*

DiffuseMagnitude);

Output.TextureUVCoordinates = TextureUVCoordinates;

return Output;

}

float4 PShader (float4 Position : SV\_POSITION, float4 Color : COLOR, float2 TextureUVCoordinates : TEXCOORD) : SV\_TARGET

{

return Color \* PlayerHoverTankTexture.Sample(DefaultTextureSampler, TextureUVCoordinates);

}

# Appendix E: AI for Seek-and-Collect

I have implemented a class for these requirements (EnemyHoverTank), I found a texture to distinguish Enemy hover-tanks from the Player’s at Textures.com. This can be found in Appendix A: Under Fig. 13. (Copyright © 2005-2017, Textures.com)

For the AI of the Enemy hover-tanks, a Finite State Machine (FSM) will be used (as they have simple behaviour, that can be aptly described in an FSM). This is available under Appendix B: Class Hierarchy, Class Diagrams and Other Diagrams: Enemy Finite State Machine Diagram.

The Player is also able to take damage from an enemy and if they take enough damage (from 10 collisions between them and the Enemy hover-tanks), they will lose. By extension though, the Player can win if they collect all the Energy capsules in the level (by default, there are 20 capsules to collect). Unfortunately, this is not explicitly mentioned to the Player (unable to set-up textures for a plane to show to the Player and/or texture to be drawn to the Player’s viewport).

# Appendix F: Enhancements to the Basic Requirements

## Scene-Manager

After setting-up a third-person camera to follow the Player as they move, I went about the implementation of a GameScene class, for the SceneManager to manage the active scenes of the game (not only certain GameObjects), as per the design laid out for the SceneManager, by John McGrath. (John McGrath and James Moran, 2017)

The GameScene would then manage any GameObjects that are part of that scene, but only when it is active.

There is a global function (called in the scope of Main.cpp), that initialises the classes. When this function is called in the WinMain() function, assert() macro-calls are used to make sure the pointers to these classes, are valid. (cplusplus.com, 2000-2017)

For referring to pointers (not copying them), by certain classes, of which the pointer is that of another class, \*& is used for such (instead of \*\*), to call the functions of that pointer’s class. (Matthew Hoggan, 2017)

After refactoring all the components of the project into their own classes, came the process of cleaning-up the project, by resolving as many warnings as possible.

# References

1. © cplusplus.com, 2000-2017. *assert* [viewed on the 29/11/2017]. Available from: <http://www.cplusplus.com/reference/cassert/assert/>
2. Matthew Hoggan, 2012. *Reason to Pass a Pointer by Reference in C++?* [viewed on the 30/11/2017]. Available from: <https://stackoverflow.com/questions/10240161/reason-to-pass-a-pointer-by-reference-in-c>
3. gradbot, 2012, *Macro definition clash between directx headers and winerror.h* [viewed on the 07/12/2017]. Available from: https://stackoverflow.com/questions/12663091/macro-definition-clash-between-directx-headers-and-winerror-h
4. © Microsoft 2018. *DXGI Overview* [Viewed on the 08/01/2018]. Available from: <https://msdn.microsoft.com/en-us/library/windows/desktop/bb205075%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396#Handling_Window_Resizing>
5. © Microsoft 2018. *IDXGISwapChain::ResizeBuffers method* [Viewed on the 08/01/2018]. Available from: <https://msdn.microsoft.com/en-us/library/bb174577(v=vs.85).aspx>
6. Master Kenneth, 2014, DirectX Leak Debugging [viewed on the 07/12/2017]. Available from: <http://masterkenth.com/directx-leak-debugging/>
7. Mike Celestino, 2015. *Lego Millennium Falcon Top-Down View* [digital image] [viewed on the 09/12/2017]. Available from: http://www.insidethemagic.net/2015/12/enormous-custom-lego-millennium-falcon-proves-star-wars-fans-dedication-attention-to-detail/
8. iedoc, 2015. 32. Simple 3rd Person Camera [Viewed on the 24/12/2017]. Available from: <https://www.braynzarsoft.net/viewtutorial/q16390-32-simple-3rd-person-camera>
9. Allen Sherrod and Wendy Jones, 2012. *Beginning DirectX® 11 Game Programming.* Boston, MA: Course Technology PTR
10. John McGrath and James Moran, 2017. *CGP600 Advanced Games Programming AE1 – Group Project.* Southampton: Southampton Solent University.
11. © 1970-2017 Analytical Scientific, LTD, 2017. *Pumice Rock* [Digital Image] [Viewed on the 26/12/2017]. Available from: <https://www.analyticalsci.com/store/p432/Pumice_Rock.html>
12. Copyright © 2005-2017, Textures.com. *RockSmoothErosion0027* [Digital Image] [Viewed on the 28/12/2017]. Available from: <https://www.textures.com/download/rocksmootherosion0027/68006?q=rock&filter=seamless> (TexturesCom\_RockSmoothErosion0027\_1\_seamless\_S.jpg)
13. Copyright © 2005-2017, Textures.com. *Bare Metal Threadplate Floor (S0070)* [Digital Image] [Viewed on the 29/12/2017]. Available from: <https://www.textures.com/preview/substance0070/126684?q=plate>

(TexturesCom\_BareThreadplateFloor\_1024\_height.TIFF)

1. Independent.co.uk, 2014. *Jack Daniels Whiskey Barrel* [Digital Image] [Viewed on the 30/12/2017]. Available from: <https://static.independent.co.uk/s3fs-public/thumbnails/image/2014/08/28/12/whiskey%20barrel.jpg>
2. Copyright © 2005-2017, Textures.com. *WoodStudded0044* [Digital Image] [Viewed on the 30/12/2017]. Available from: <https://www.textures.com/download/woodstudded0044/46382?q=wood&filter=seamless> (TexturesCom\_WoodStudded0044\_1\_seamless\_S.jpg)
3. 9to5Google.com, 2013. *Huge Improvements to batter-life expected in Android 4.3?* [Digital Image] [Viewed on the 31/12/2017]. Available from: <https://9to5google.files.wordpress.com/2013/07/glowing-green-battery-charging.jpg>
4. Copyright © 2005-2017, Textures.com. MetalPlatesPainted0022 [Digital Image] [Viewed on the 01/01/2018]. Available from: <https://www.textures.com/download/metalplatespainted0022/45883?q=can> (Textures.com.TexturesCom\_MetalPlatesPainted0022\_1\_seamless\_S.jpg)
5. Georg Fritzsche, 2010. *C++ Erase vector element by value rather than by position? [duplicate]* [viewed on the 01/01/2018]. Available from: <https://stackoverflow.com/questions/3385229/c-erase-vector-element-by-value-rather-than-by-position>
6. Anthony Carmona, Copyright © 2005-2018, Textures.com. Metal Threadplate Floor (S0035) [Digital Image] [Viewed on the 02/01/2018]. Available from: <https://www.textures.com/download/substance0035/127550?q=black+metal>

(TexturesCom\_ThreadplateFloor\_1024\_albedo.tiff)