Supporting Documentation for a Game built with 'Ogre 3D' . Programmed in C#.

Candidate Number: 178492

Interactive 3D Programming

MSc Computing with Digital Media

10th May 2018

3500 Words

Contents:

1. Introduction

This was an exercise in understanding and practicing generating 3D computer graphics from the ground up using the 'Ogre 3D' engine and the C# programming language in Microsoft 'Visual Basic'. This report will document the journey taken, the problems encountered and the outcomes from the development process while using the game engine and c#. A framework of abstract classes was initially given to demonstrate a good structure of functionality for game development and the developer asked to research, understand and complete the framework while ultimately producing a fully functional and playable game in the process.

2. Background

3D graphics start out as vertex (x,y,z) points in a 3 dimensional space in computer memory and end up fully rendered and animated on a screen. The first stage of development was to create a single polygon with a surface and get it to render on screen using the game engine as a viewer.. The vertex are generated and plotted in space, their triangle direction count is created in the triangle index buffer to set the normal direction and create a polygon, add a material to the polygon and then finally attach it to the scene node to render it when running the game engine. This doesn't seem like a lot of code, only 107 lines, but you have to then imagine that there will be 100's of 1000's of polygons and this would be frightening if you had to create all the meshes in this way. The work load and design restrictions alone are immense. Obviously you would use an external package like Autodesk 3DS Max or similar and export a .MESH file to then import into the engine.

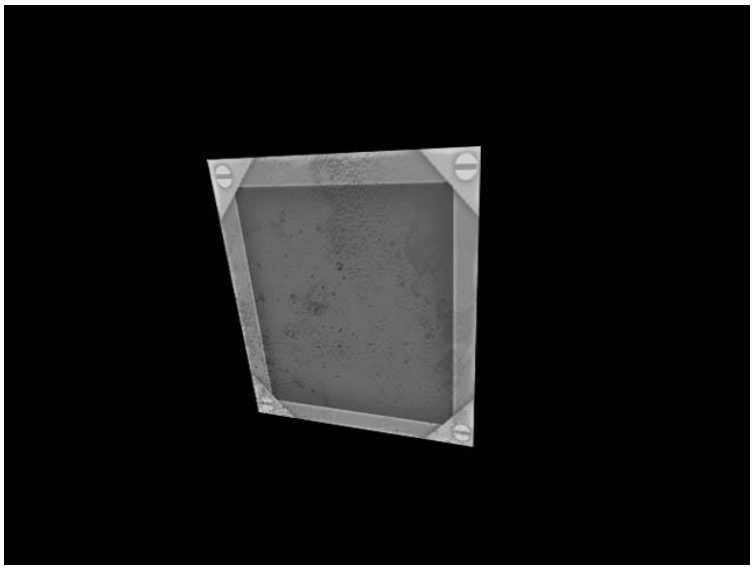


Fig. 01 - The single polygon in 3D space

The next level of development is to produce a 3D object as well as the 2D plane. Now there are 8 vertex instead of 4 (Figure 02) and once all the polygons have been created there now exist 6 polygons in 3 dimensions. These objects created are called 'ManualObjects' and are a part of the 'Ogre 3D' engine.

Here is when a problem developed, while mapping the material art correctly to all 6 sides. the left / right, forward / back facing sides would map correctly but the top / bottom sides would be stretched. After a bit of trial and error and research on the 'Ogre' forums and 'Stack Overflow' (Ref - [2], Ref - [3]) a decision was made to use 24 vertex and have all sides as separate planes, mapped correctly and rotated to form the cube. This fit the purpose and allowed the development to progress forward, but probably isn't the best solution.

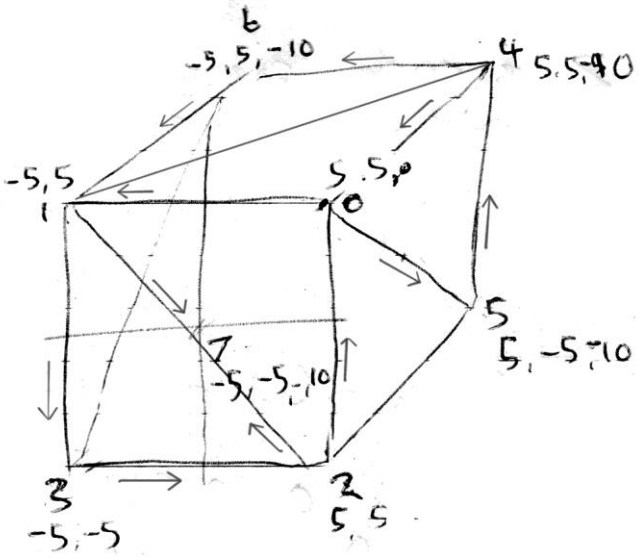


Figure 02 - Working out the cubes vertex co ordinates and UV's

This geometry is stored in the 'Leaf Node' and is attached to a 'Group node' to transform the geometry in a scene. The Group Node is then attached to the root scene node and the geometry can be passed to the 'Rasterizer' and processed for display on screen.

The mesh is given UV coordinates so when a material is added to the surface of the polygons a bitmap is displayed properly. The UV co ordinates correlate to each vertex in the mesh so to have a bitmap display correctly on the surface means that the UV co ordinates have to be mapped accordingly in the UV editor, or in this case, worked out on paper and then applied to each vertex in a line of code. The normal or cross product for each triangle in the mesh is calculated from the triangle edge vectors so the light will react properly on the mesh surface. All of this is passed to the 'Rasterizer', which assembles the polygons and converts each polygon to a set of fragments. Each fragment represents a pixel on the mesh surface and these fragments are viewed through a camera in the scene and a 2D image is produced.

The colour of each screen fragment is derived from various processes; frustum clipping, culling, lighting and texturing. Frustum is the area of view through the camera in the scene. Using a projection matrix, forshortening and perspective can be achieved on the final 2D rasterized image. Each fragment or pixel in the rasterized 2D image has a z depth buffer which contains the distance from the camera of the corresponding 3D geometry. Everything is brought together for a final output merging stage. The colour buffer, the 'Alpha' for the opacity and the Z buffer data for each screen pixel. This is all explained extensively in Chapters 1 - 4 of JungHyun Han : Programming for 3D(Ref [1])

3. Design, Development and Game Mechanics

3.a Platform and OS

This application is developed and tested on a PC running windows 7 and 10. The minimum system requirements would be at least 25 meg free hardware space, 32 bit operating system and processor speed of 2.2 Ghz.

3.b External Code

Direct3D 9 is used for rendering the 'Ogre 3D' engine. There are 'OpenGL' dll libraries included in a build. The initial framework and abstract classes provided by Course tutors.

3.c Code Objects

Main code object is **'GameElement'**. This class is the root of all the elements in the game world. It branches off as movable or static, characters, player, props, collectables and

projectile objects. There is an **'Environment'** object that builds the boundaries and sets the lighting and effects. A **'Physics Engine'** is hooked into the game objects and the environment to control how the elements react with each other and use the collision to influence game play and scoring. The **'Inputs manager'** manages all keyboard and mouse input and is sent to the **'Controller**' chain of classes. A **'HMD'** object creates the display for the 'OverlayElements' and the **'Stats'** in the **'GameInterface'**. The **'Manager'** class contains the main loop and creates and destroys the game scene and all its contents. This class persists through the entire session along with the game interface, the player. the robot enemies and the environment. Collectables are destroyed when collided with by the player character.

3.d Control Loop

The program is started by calling the 'Main' method in the 'Manager' class'. Once the scene is created, a camera created and a view port set up, the game as it stands loops around the 'UpdateScene()' method in the Manager class every frame. In this loop a list of processes are worked through;

1. The inputs from the user are checked. and the Player position and collision updated.
2. The camera LookAt() is updated to the position of the player.
3. The robot enemies are cycled through and their AI, position, collision and status is updated.
4. The pick ups are cycled through and their status and position etc updated. Any collisions are detected and responded to accordingly i.e. Collectables deleted on collision.
5. The interface is updated. Score and health adjusted.
6. Finally the physics for the entire game is updated. This includes all Phys Objects and collision lists on all game elements.

3.e Game Object Data

Player Object:	Input data - Forward, Back, Up, Down, Left, Right
	Position
	Health data Stat
	Shield data Stat
	Score data Stat
	Speed
Robot Objects:	Direction - towards player position.
	Position
	Health data Stat
	Speed Stat
	<List> for Robots in world
Gems:	<List> for Gems in world
	<List> for Gems to be removed from world
	score per Gem
HUD	Score data Stat
	Player Health Stat
	Player Shield Stat
	Game Time data
	Player Lives Stat
Physics	<List> Collided with
	<List> Forces in game world

3.f Data Flow:

All data is created and initialised at start up. A list of data types follows:
Player health, lives and shield,
All enemy health and shield,
Score

All these data types are stored in Stat classes which are used by the character and interface classes during run time and are destroyed when the game has ended.

3.g Game Physics and Statistics

The Contoller objects control all the GameElements in the game. The Player controller is instantiated from the Inputs Manager class. All Controllers use a boolean system of variables for all directions possible in the game. Up, Down, Left, Right, Forward and Back.

The vector of the Player position is altered by adding or subtracting the boolean variables from the current position vector. These boolean values are set true (1) when the corresponding key is pressed. The enemy movement uses the same system but instead of being triggered by player input the movement booleans are set by the AI of the game. This all happens in the MovementsControl() function of the controller object and is where the speed and acceleration can be adjusted.

The collision objects are attached to each game element when they are created at the 'ModelElement' level of the 'GameElement' class tree. The Physics object is inherited from the base class 'GameElement' so is applied to every element. The radius and mass variables are passed as parameters on instantiation of each Physics object applied to the top sceneNode of all element groups. This 'sceneNode' is the node used to move the group. For example, the Player Model contains a hierarchy of 7 'sceneNode's, the Robot Model has 2 'sceneNodes'.

Every frame the Physics engine is updated and collision lists created for every object in the scene. For collectables, the collision list in the corresponding physics object is used to identify and delete all objects that are flagged as colliding with the Player Model. The enemies will have the same removal flag applied to destroy the models when shot, but this hasn't been implemented yet.

Force objects are created in the physics engine which the physics engine uses to affect all model objects in the game scene. A 'WeightForce' object is created and applied to all objects in the game scene and acts like gravity in the scene. A private constant float in this class sets the amount of gravity force applied to all models. This force is calculated in the compute() function.

There isn't any real combat currently in the application, but enemies do decrease the health Stat of the Player on collision.

In the Manger class the amount of enemies, collectables and obstacles can be adjusted in the 'robotNum', 'cubeNum' and 'gemNum' variables. This slows down the frame rate the more game elements added because of the physics calculated on every model.

3.h Artificial Intelligence (AI)

The robot enemies have a simple seek and destroy AI. Its not really intelligence at all. The first thing each enemy instantiated into the game world does on each new frame is turn to face in the direction of the player. Then the robot is forced to move along its forward vector. This produces amusing behaviour, but instead of walking determined toward the player, the enemies bounce and fly towards the player sometimes at skew angles. The X and Z axis of the robots need to be locked so only the Y axis is used, preventing this crazy pitch and roll.

As soon as the player collision list contains a robots collision / physics object id, health is deducted from the players health Stat. This takes place in the IsCollidingWith() method in the 'PlayerMode'l.

3.i Multiplayer

No Multiplayer code.

3.j User Interface

The User Interface is created from the HMD base class and creates an Overlay manager to control Overlay objects built into the Mogre engine. The elements in the interface are connected with the 'Player Stats' so as to be updated during run time. In game HUD is the only interface at this stage. The game is started by running the .EXE and ended by escaping the window and re running the .EXE, but this would at least, on another pass, be improved by the addition of a 'Play Again' button on game over and at boot up. Colour of interface elements is easily changed by changing parameters in the GameInterface materials file as well as bitmap images.

3.k Game Shell

There are no game shell screens, but there should be at least a game lobby screen with a 'Play' button and a display of the previous games score created in the next development pass.

3.l Main Play Screen

Currently there is only a Play Screen. It consists of the 3D scene and game area with the camera attached to the Player gameNode and the game interface overlay on top.

3.m Art and Video

The 3D meshes are imported .MESH files apart from the Cube obstacles which are 'ManualObjects' made from creating vertex and triangle lists and converted into a mesh in the CubeModel class. The robot animations are stored in the .MESH file and accessed when the .MESH file is loaded into a gameEntity AnimationStateIterator.

3.n Graphics engine

The graphics are produced in the Mogre 3D engine. A scene manager and viewport are set up in relation to a camera and all objects are attached to the scene manager so as to be rendered in the output window.

3.o Artist Instructions

All mesh objects need to be exported as .MESH files from the external 3D package used. All bitmaps should be no more than 1024 x 1024. A pixel density hasn't been established, but maybe 128 pixels per metre would be a good guide.

3.p Sound and Music

Currently there is no sound or music in the application.

3.q Level Specific Code

There is no level specific code currently in the application apart from a simple timer that stops the game, in the same way as in game over, when the max time has been reached. Max time is set in the Update() method in the Manager() class. This can be used to time levels, but currently as the application stands this is a simple 'Win' outcome as opposed to the 'Lose' outcome when the payer runs out of lives.

4. Advanced Functionality

4.a Controls

These were changed to make the player movement feel more natural. The mouse left and right movement steers the player avatar while the 'W' and 'S' keys accelerate and decelerate the avatar. The 'A' and 'D' keys still work, straifing the avatar left and right but the movement is in relation to the forward and backward movement. If there is no forward / backward movement there can be no straifing. Inertia is added to the movement to prevent unrealistic sudden stops and provide acceleration. This has a more natural feel and even though the player doesn't have a maximum speed so can go very fast if forward is held down for a long period of time, the speed is easily controlled. This inertia functionality is added in the MovementControls() method in the Player Controller and is an addition to the original boolean direction system. It took a bit of thinking through and a few passes of tweaking to get the movement just right. When the player avatar is moving forward and the backward button is pressed, the backward button acts like a brake until the players movement becomes 0 then a reverse variable is set true and the back button increases its negative speed. In this circumstance the forward button is like a brake, slowing down negative movement and then increasing positive movement when the player avatar speed reaches 0. The end result definitely adds to the feel of the movement, but the maths may not be the best and most efficient.

The space bar acts as a jump button (or jet pack) and sets 'true' the boolean variable that controls the 'Up' movement in the 'Player Controller' class. The physics takes care of the rest of the movement ie. the falling and bouncing when the space bar is released.

4.b Interface

The textures and colours were changed in the material for the game interface to produce a more co-ordinated colour scheme and theme. The map applied to the hearts in the interface has been worked into and they needed to be positioned correctly within the 'Overlay3D' overlay.

4.c Cubes

These are used as obstacles and appear randomly over the game area. The original cube with 8 vertexes wouldn't map properly. 4 sides would map the bitmap correctly, but the top and bottom sides were always stretched. Instead 24 vertex were used so each vertex coordinate has 3 normal directions and the bitmap can be mapped properly on every side.

The obstacles are stored in a <List> collection and the number of cubes can be varied using the 'cubeNum' variable. Each cube is positioned randomly around the game world giving each run time a different feel as the cubes act like walls.

4.d Gems and Power ups

The 2 abstract classes seem similar so a blend of the two was created by having gems as power ups effecting the health and the ammo and score. The Gems are stored in a <List> collection similar to the cubes. These inherit from the Collectables class, have a spinning animation and the amount of gems instantiated in the game world is stored in a variable in the Manager object. Each collectable object is given a randomly generated position. This is fine for this stage of development but may cause problems if an important pick up is instantiated inside a cube and cannot be collected. This may depend on the reactions of the physics.

Guns to be collected in the game world should be instantiated from the Collectables Class and the corresponding Gun class object added to the Armoury. Sadly no projectile objects are being created or used in this application.

4.e Environment

Custom texture maps have been created and the original 'Mogre' materials have been modified to produce an individual look. This look ended up looking a bit retro 80's, but the process demonstrates the ease in which styling can be changed without affecting the other areas of code. A similar themed texture map has been used in the interface to tie it in with the game world graphics. Only diffuse colouring has been utilised with point lights.

No bump or specular has been applied to the materials at this stage, The shadows are set up to work and all the 'ModelElement' Entitie's mesh have had BuildEdgeList() applied to them, but the shadows didn't work properly and, at certain times, seriously impaired the frame rate and game play. Because of this at this stage of development they have been designed out of the look of the game.

The environment contains an ambient light, a light that is attached to the player and acts like a headlight and red lights were attached to all robot models. These robot lights caused strange lighting effects so were switched off untill a better solution can be coded.

4.f Physics

The physics has been added to 'gameNodes' in the constructor of the 'ModelElement' class. A physics object inside a 'ModelElement' is created when passed the id name 'Phys_'. All physics objects have a unique name so each and every object can be identified by the collision of other objects.

Big problems arose with aligning the physics object element with the mesh element and with calculating the weight force for each physics object. According to Newton's laws of physics; Force = mass * acceleration. This logic has to be converted to vectors and more precisely the difference between the current vector and the proposed new vector of an object.

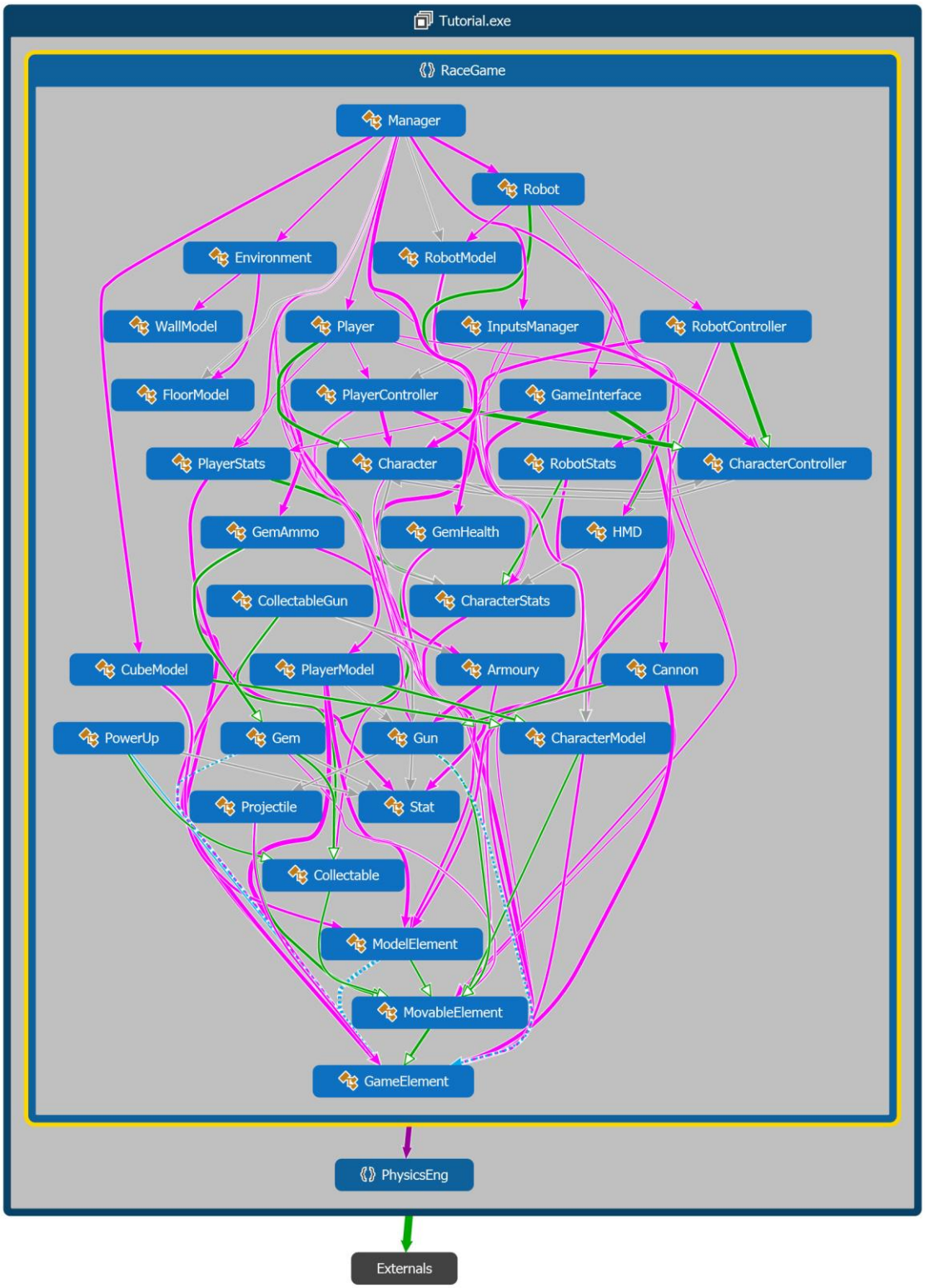
Ultimately, according to the original UML class design, the physics object is inherited from the 'GameElement' class which is the top parent class for creating most of the game elements, so through the 'ModelElement' class each model in the game will be given a 'PhysObj' automatically when created. When first setting up the physics, the mistake was made of not overriding the base physics object properly and when removing objects from the scene, the physics objects remained even though the mesh had disappeared. So the wrong physics object was being referenced. This was remedied by not overriding the base 'PhysObj' in the 'ModelElement' class.

Objects are prevented from falling through the floor by setting the floor as a 'Boundary' the same as the walls surrounding the game area. These boundaries are 'Plane' objects that are a part of the 'Mogre' library. The 'FloorModel' worked fine once it was set up as a boundary and included into the physics engine, but the walls didn't work as well. Once implemented they were definitely present in the game world, but the results were not what was expected and due to time restrictions a simple set up in the 'PlayerController' was established that simply cancels out the vector move of the PlayerModel if the X and Z reach the set world limits. In this case the world is 2000 to -2000 units square.

4.g Enemies

Robot enemy movement is controlled using a 'RobotController' class. This is basically a copy of the 'PlayerController' Class and initially the robot enemy faces the player and moves forward untill the physics collides with the player physics object. This is the basic AI that all enemies will have at this stage in development. When alternate enemies are added each one would be a chance to work on the AI more added behaviours.

5. UML Diagram



6. References

[1] JungHyun Han: 3D Graphics for Programming: CRC Press, Taylor and Francis Group (2011)
[2] Ogre Forums : https://forums.ogre3d.org
[3] Stack Overflow: https://stackoverflow.com