Programming for 3D 2015

The Gem Battle – race game

Coursework Report

Candidate number: 118440

Date: 15th April, 2015

TABLE OF CONTENTS

1. Introduction 4

2. 3D Graphics Rendering Background 5

2.1 Introduction 5

2.2 Input-Assemble Stage 6

2.3 Vertex Processing 7

2.4 Rasterization Stage 8

2.5 Fragment Processing 9

2.6 Output Merging 10

2.7 Conclusion 10

3 Game design document 11

3.1 Game Overview 12

3.2 Game Play 13

3.2.1 Rules 13

3.2.2 Challenge Structure 13

3.3 Game Mechanics 14

3.3.1 Game Flow 14

3.3.2 Game Loop 15

3.3.3 Game Matrix 16

3.3.4 Player Statistics 18

3.3.5 Player Rewards 18

3.3.6 Player Control 18

3.3.7 User Interface 19

3.3.8 Physics and Statistics 20

3.3.9 enemy ai (Artificial Intelligence) 20

4 advanced functionality 21

4.1 Scoring System 22

4.2 Health System 23

4.3 Level System 24

4.4 Guns 25

4.5 Environment 26

4.7 Enemies and AI 28

5 Conclusion 29

6 References 30

1. INTRODUCTION

This report aim at explaining the background knowledge of 3D graphic rendering pipeline, as well as presenting the race game coursework done in the Programming for 3D course, 2015.

The report is divided into 3 main parts: 3D Graphics Rendering Background, the Game Design Document for the game, and a detailed breakdown of the advanced functionalities in the game.

2. 3D GRAPHICS RENDERING BACKGROUND

2.1 INTRODUCTION

In this section, an introduction to the 3D graphic rendering pipeline could be found.

The 3D Graphics Pipeline describes how every single elements in a 3D computer-generated graphic is done, as well as explaining how numeric data is generated into mesh, then transformed into 3D graphics, and finally processed back into 2D pixels and display on screens.

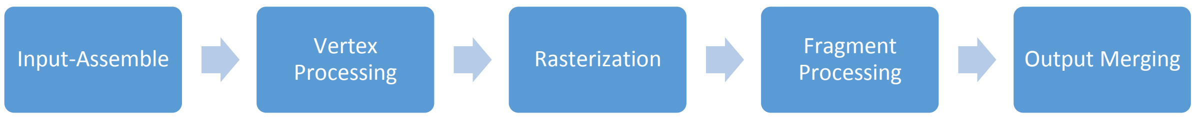
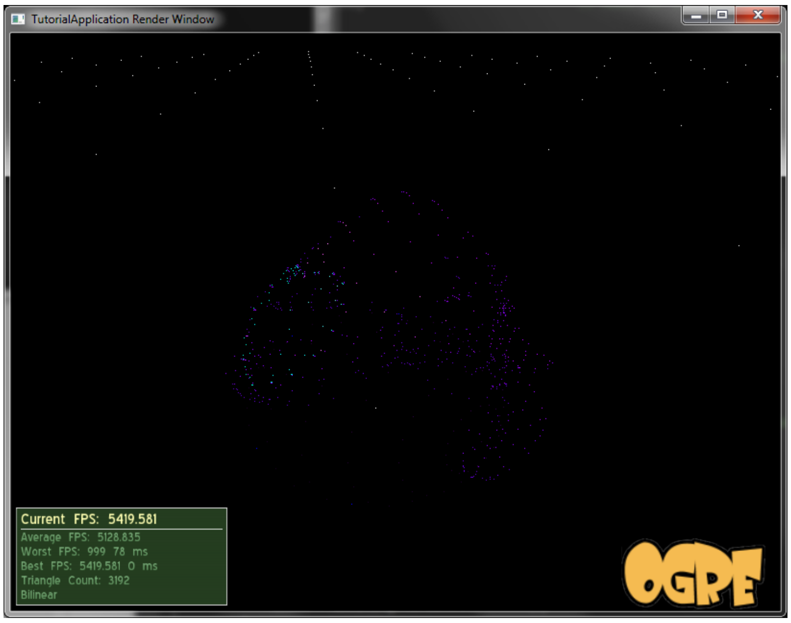


FIGURE 2.1A – RENDERING PIPELINE OUTLINE

Referring to figure 2.1a, the pipeline starts from the Input-Assemble stage. At the starting stage, the input is purely numeric data. After reaching the Geometry Processing Stage, lines and triangles are visible. Finally, colours are added in Fragment Processing stage and being merged into the final product at Output Merging Stage.

In the followings, a detailed description of every fundamental stage in the pipeline could be found,

2.2 INPUT-ASSEMBLE STAGE



```
Current  FPS:  5419.581
Average  FPS:  5128.835
Worst  FPS:  999  78  ms
Best  FPS:  5419.581  0  ms
Triangle  Count:  3192
Bilinear
```

FIGURE 2.2A – OUTPUT OF A INPUT-ASSEMBLER

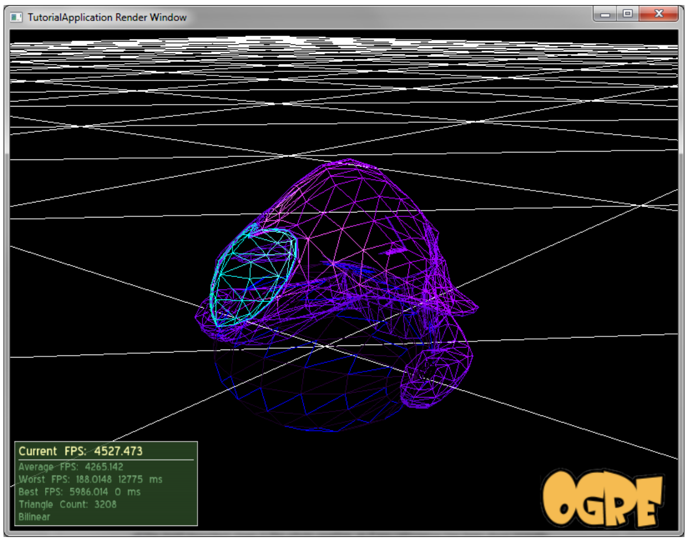
The input-assemble stage is where the whole rendering process begins.

This process takes all the data required to form a 3D graphics, which are mainly vertices. The data taken will then be translated into something useful, namely the Primitives, for future use.

A Primitive is the most basic element in 3D graphics, which includes line lists, triangle lists etc. It is the most basic element of 3D graphics and every single object in 3D world could be divided into primitives.

As seen on figure 2.2a, the the output of this stage is purely numeric data and doesn't involve any graphic presentations yet, but this is the most fundamental part of the rendering pipeline.

2.3 VERTEX PROCESSING



```
Current FPS: 4527.473
Average FPS: 4265.142
Worst FPS: 188.0148 12775 ms
Best FPS: 5986.014 0 ms
Triangle Count: 3208
Bilinear
```

FIGURE 2.3 A – OUTPUT OF A VERTEX PROCESSOR

In the vertex processing stage, several transformation operations are performed, using the information from the previous stage.

The whole process could be imagined as taking a photograph in the real world.

The first process would be Model Transforming. This is similar to placing the real object in the real world. But in 3D graphics, the object will be scaled, transformed of rotated, in order to transform the model space to the world space. A model space is where the 3D objects are created originally, while a world space is where the whole 3D scene world will be displayed.

The second stage is View Transform, which involved positioning the camera, where in world space the camera position depends the final rendering image at the end of the pipeline.

Once the camera is positioned, the model is sent to Projection Transform stage. This process depends what the camera captures, by adjusting the focus and zoom. This affect how the object will be rendered and displayed on screen.

At the end of stage, an outline of the 3D object should be visible on screen as shown on the example figure 2.3a.

2.4 RASTERIZATION STAGE

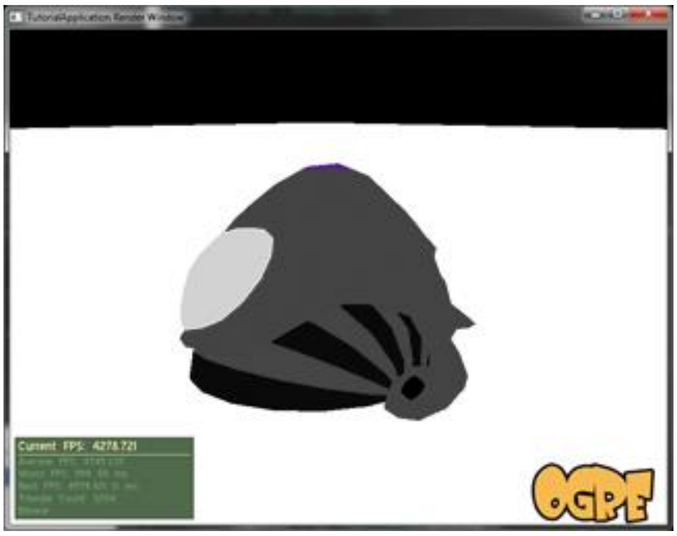
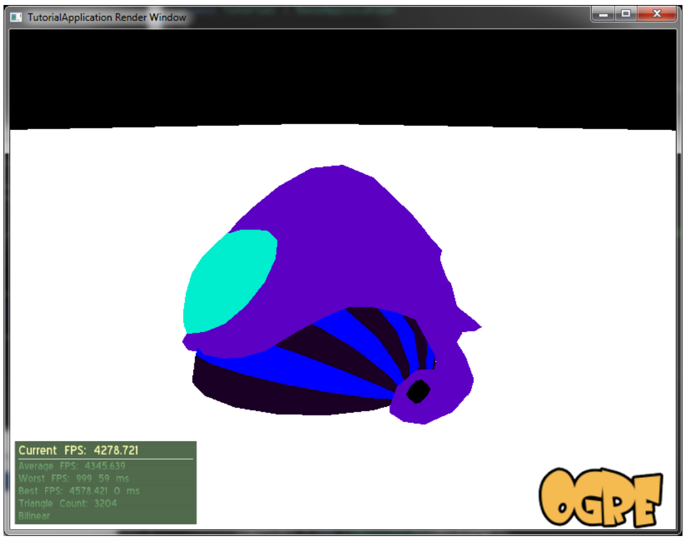


FIGURE 2.4 A – OUTPUT OF THE RASTERIZATION STAGE

In this stage, vertices are transformed into fragments. It is a hard-wired and non-programmable stage as it takes information from the vertex processor which will be aligned into pixels so it becomes more visible on screen.

By the end of this stage, the outline of the figure could be seen, but it doesn't involve any colouring display yet, since this is the job of the next stage.

2.5 FRAGMENT PROCESSING



```
Current FPS: 4278.721
Average FPS: 4345.639
Worst FPS: 999 59 ms
Best FPS: 4578.421 0 ms
Triangle Count: 3204
Bilinear
```

FIGURE 2.5 A – OUTPUT OF THE RASTERIZATION STAGE

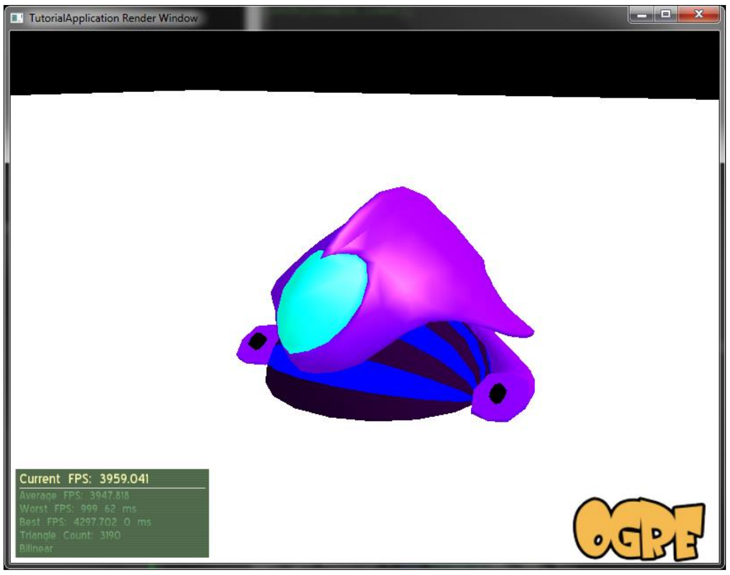
During this stage, the fragment processor make use of the fragments output from the rasterization stage. Useful information contains in each fragments of every primitives such as depth and RGB colour, to determine the colour of each fragment on the model.

This stage is divided into 2 sections: Texturing and lighting.

Texturing is usually done by loading the pixel information from a piece of graphic into the game element. Lighting, on the other hand, determine the realness of the object. Different algorithms deal with different level of lighting.

This process fairly affects the output quality of the graphic display.

2.6 OUTPUT MERGING



```
Current  FPS:  3959.041
Average  FPS:  3947.818
Worst  FPS:  999  62  ms
Best  FPS:  4297.702  0  ms
Triangle  Count:  3190
Bilinear
```

FIGURE 2.6 A – FINAL PRODUCT OF THE PUTPUT MERGING STAGE

This is the end of the rendering pipeline, where everything need to be merged and displayed on screen.

Again, this stage is non-programmable as the essential information for this stage, as known as the RGB-AZ value, comes from the previous stage. The A value means alpha (the capacity) and the Z means the depth. To blend everything together, the output merger needs to use the Z-buffer and the Alpha-Blender to combine the AZ value with the RBG value stored in the colour buffers.
The AZ values add realism to the final image, and enhance the graphic quality. As could be seen on the figures above, figure 2.6a contain a greater depth and more advanced shading compared to figure 2.5a.

After the process is done, the complete image should be shown on screen.

2.7 CONCLUSION

This section has introduced the main stages in the 3D graphic rendering pipeline. In the following section, the complete game design document of the game Gem battle could be found.

3 GAME DESIGN DOCUMENT

The Gem Battle

Game Design Document

3.1 GAME OVERVIEW

In this section, an overview of the game genre, description and summary could be found.

Game Description

Game	The Gem Battle
Genre	Strategy
Game Elements	Shooting Collecting Dodging
Game Sequence	3-level based linear sequence
Player	Single Player

TABLE 3.1 A – GAME DESCRIPTION

Game Technical Specifications

Graphic Display	3D graphics
Platform	C++, C#
Device Requirement	Windows 7 or above, Direct3D9 or OpenGL

TABLE 3.1.B – GAME TECHNICAL SPECIFICATIONS

3.2 GAME PLAY

In this section, a brief introduction of the Gem Battle could be found, including the rules and the level flow.

3.2.1 RULES

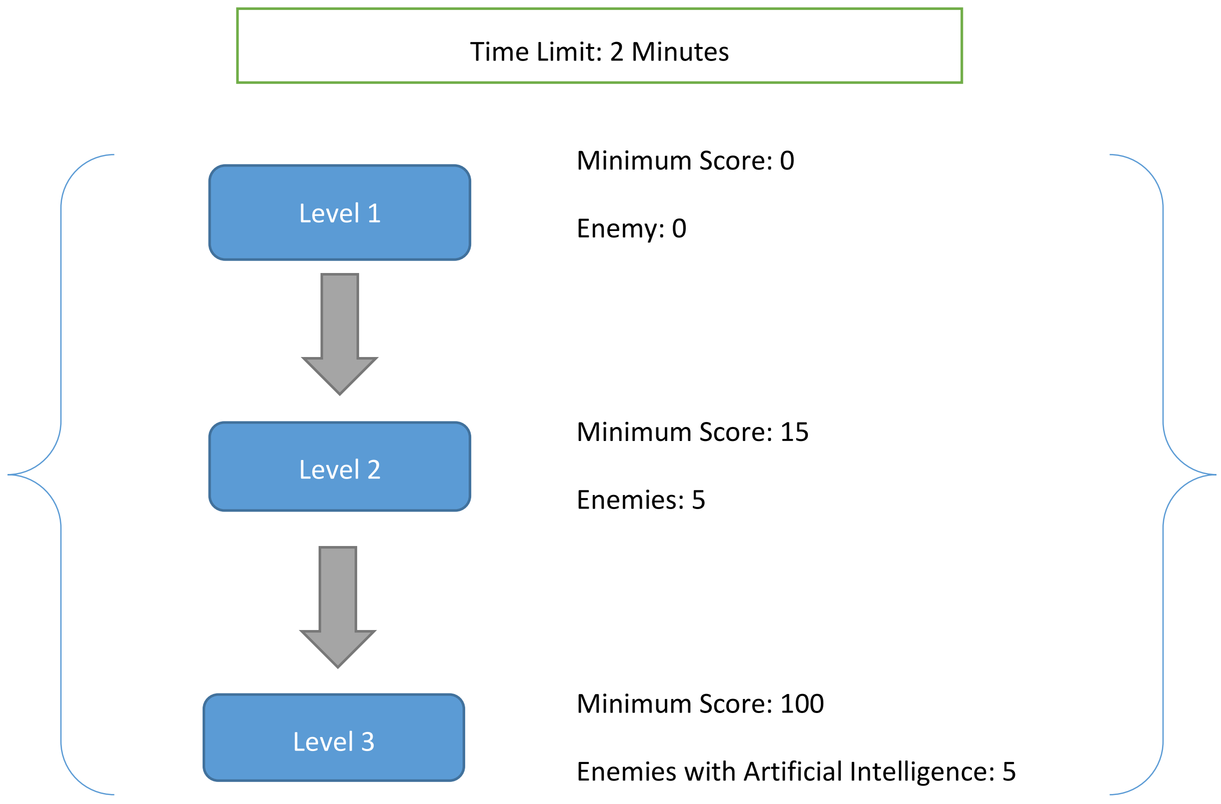
How to Win?

- Players must kill all the enemies in all levels to win the game within 2 minutes.
- However, enemies only exist in level 2 and level 3.
- Enemies in level 3 have AI so be careful
- In order to progress to the next level, players must reach a specific scores
- Players can gain scores by collecting gems, or killing an enemy.

Others

- Player must stay inside the playground
- Players can collect power ups to upgrade their health and shield values.
- Guns could be collected and used to fight enemies
- Once hitting with an enemy, player will lose health.

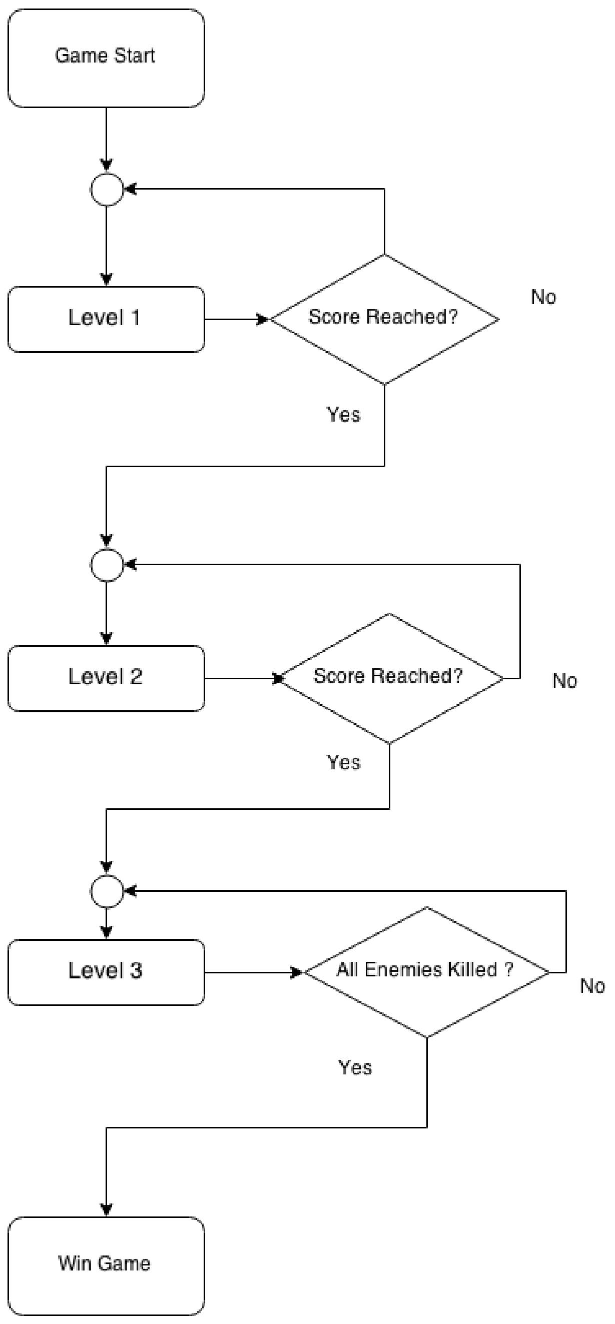
3.2.2 CHALLENGE STRUCTURE



3.3 GAME MECHANICS

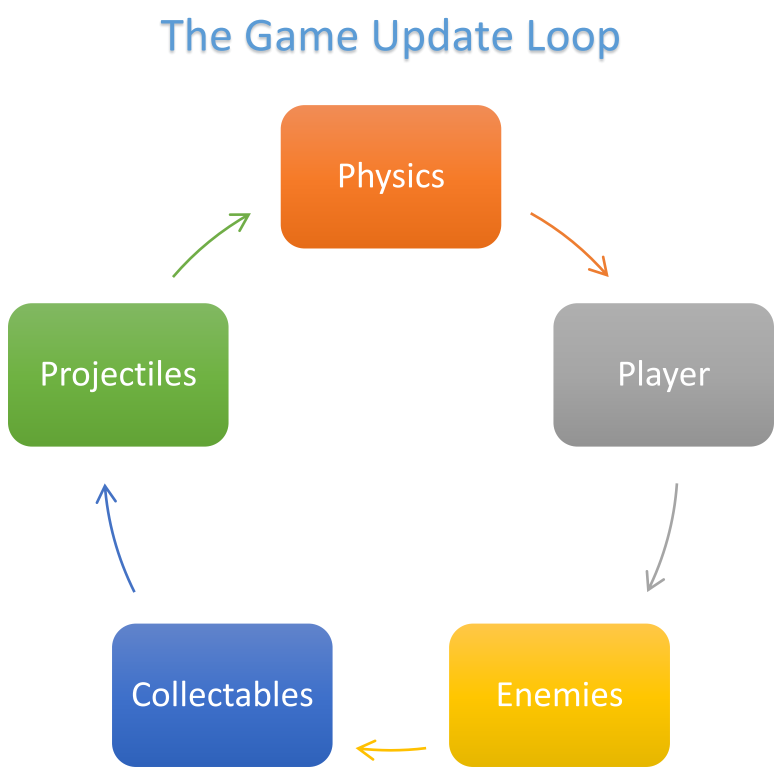
3.3.1 GAME FLOW

In this flow chart, the game flow of the game could be found.



3.3.2 GAME LOOP

This section describe what are the elements updated in the game loop.



Player Update

- Model Animation
- Player Stat
- player Controller
- player Aromoury
- Collision with Enemie

Enemies Update

- Model Animation
- Collision Detection with Projectiles

Collectables Update

- Collision Detection with Player

Projectile Update

- Collision with Enemy
- Timer

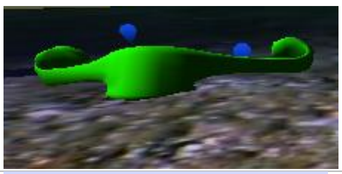
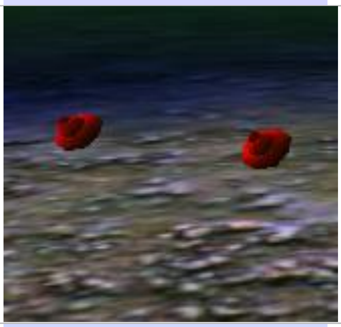
Physics Update

- Model Animation
- Collision Detection with Projectiles

3.3.3 GAME MATRIX

In the game matrix, a general description of every game object could be found.

Game Objects	Display	Rewards for Player	Damage towards Player	No. of Items
Player		n/a	n/a	1
Enemy		n/a	-1 Shield	5
Red Gem		+ 4 Scores	n/a	10
Green Gem		+ 2 Scores	n/a	10
Health Up		+ 1 Health	n/a	5

Shield Up		+ 1 Shield	n/a	5
Bomb Dropper		n/a	n/a	1
Bomb		n/a	n/a	10
Cannon		n/a	n/a	1
Cannon Ball		n/a	n/a	10

3.3.4 PLAYER STATISTICS

Properties	Initial Value	Maximum Value
Health	5	10
Shield	5	10
Live	1	3

3.3.5 PLAYER REWARDS

Action	Reward
Collecting Green Gems	+ 2 Scores
Collecting Red Gems	+ 4 Scores
Killing an Enemy	+ 20 Scores

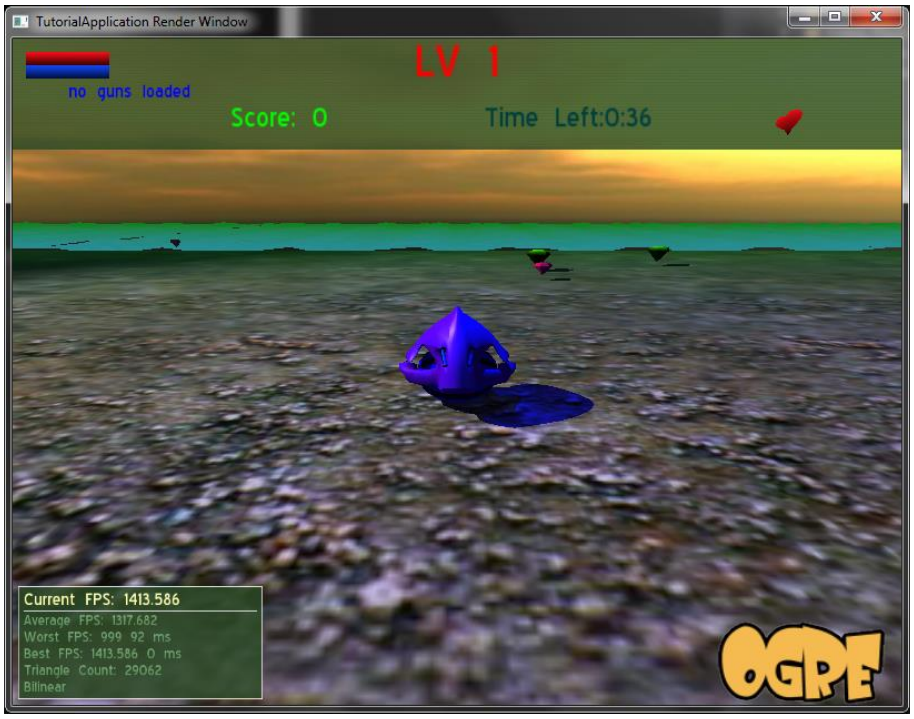
3.3.6 PLAYER CONTROL

Key Board Input	Affected Game Object	Action On Screen
A	Player Movement	Left
D	Player Movement	Right
W	Player Movement	Front
S	Player Movement	Back
E	Gun	Swap Gun
Space bar	Gun and Bullet	Shoot

Mouse Input	Affected Game Object	Action On Screen
Left Mouse Button	Player Viewpoint	Drag to the left or right

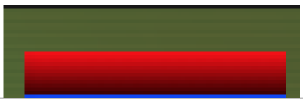
3.3.7 USER INTERFACE

In this section an overview of the user interface could be found, including the detailed explanation of the functionalities of each element.



```
Current FPS: 1413.586
Average FPS: 1317.682
Worst FPS: 999 92 ms
Best FPS: 1413.586 0 ms
Triangle Count: 29062
Bilinear
```

FIGURE 2.6 A – FULL DISPLAY OF THE USER INTERFACE

Item name	Display	Function
Level Display		Show the level that the player is in
Score Display		Show the score of the player
Time Left Display		Show the time left before game over
Health Bar		Show the value of health of the player
Shield Bar		Show the value of shield of the player

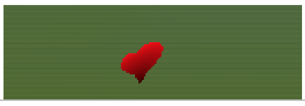
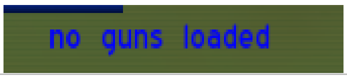
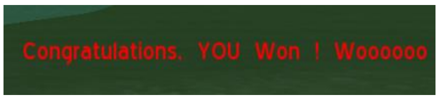
Life Display		Show the number of life left of the player
Ammo Load Display	 no guns loaded	Show the number of ammo left when there is a gun loaded
Game Over / Winning Display	 Congratulations. YOU Won ! Woooooo	Show the result of the game, game over of winning.

TABLE 2.6 A – DETAILED EXPLAINATION OF THE UI

3.3.8 PHYSICS AND STATISTICS

Object Name	Radius	Weight Force	Friction Force	Elastic Force
Player	10	50	0.8	10
Enemy	15	50	0.8	10
Red Gem	5	100	0.5	0.1
Green Gem	5	100	0.5	0.1
Health Up	5	100	0.5	0.1
Shield Up	5	100	0.5	0.1
Bomb Dropper	10	50	0.1	0.5
Bomb	10	33.3	0.1	0.5
Cannon	10	50	0.1	0.5
Cannon Ball	10	33.3	0.1	0.5

3.3.9 ENEMY AI (ARTIFICIAL INTELLIGENCE)

Condition	Action
Player is far	Walk in circles
Player is near	Walk towards player

4 ADVANCED FUNCTIONALITY

This section will be a detailed introduction with explanation of what advanced functionality have been done in the Gem Battle game. A brief methodology will be mentioned as well regarding who these functionalities are being implemented.

4.1 SCORING SYSTEM

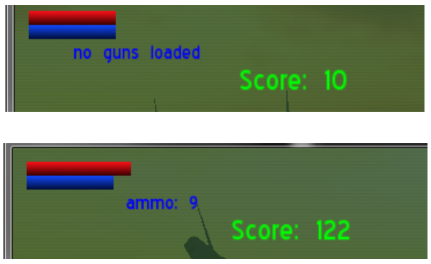
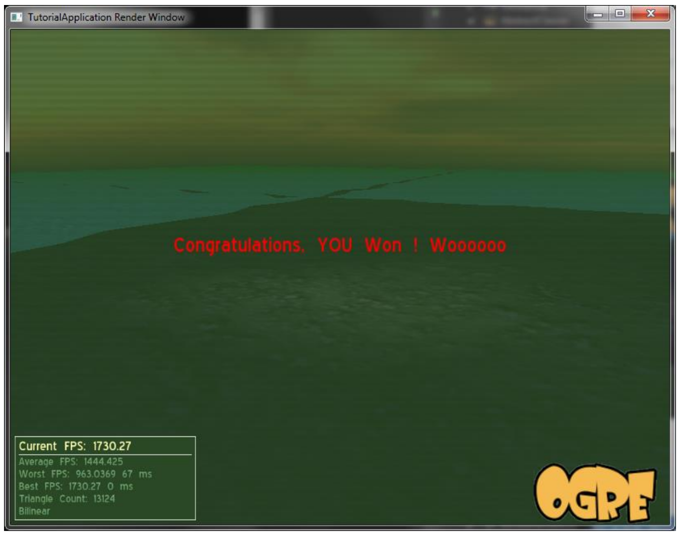


FIGURE 4.1 A – SCORE DISPLAY CHANGES

The scoring system is the main element to keep the player motivated as the level system depends on the player scores. Once an event that trigger the score to go up and down, the user interface will be updated as well as score. (See figure 4.1a)

There are mainly two ways to gain score in the game: one is by collecting gems, two is by shooting an enemy. Either way is done by collision detection in the physics engine.



Congratulations. YOU Won ! Woooooo

```
Current FPS: 1730.27
Average FPS: 1444.425
Worst FPS: 963.0369 67 ms
Best FPS: 1730.27 0 ms
Triangle Count: 13124
Bilinear
```

FIGURE 4.1B – WINNING MESSAGE DISPLAY

Other than the score that displays on the UI, another secret scoring system that runs inside the program is the enemy killing count. Since the aim of the whole game is to kill every single enemy, the player stat keep track of the number of robot killed. Once all the robot are being killed, the winning message will pop up and the game will end. (See figure 4.1b)

4.2 HEALTH SYSTEM

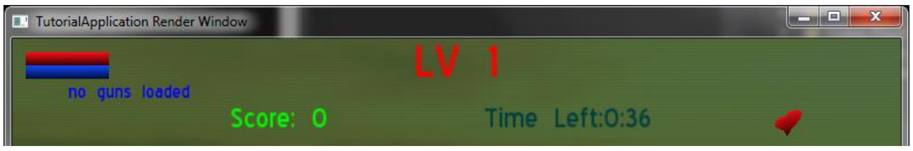


FIGURE 4.2 A – INITIAL HEALTH SYSTM DISPLAY

Players have an initial stat once the game started, which could be seen as figure 4.2a.

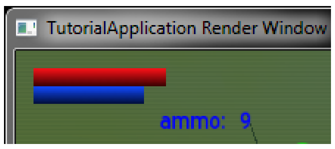


FIGURE 4.2 B – HEALTH VALUE INCREASED DISPLAY

These health value could be increased by collecting power ups on the ground. To enable this functionality, collision detection is applied to the gem which detects whether a player have crashed into a specific power up. Once it does, the corresponding health value will be upgraded and the graphic interface will be updated as well depending on the state of the player. (See figure 4.2b)

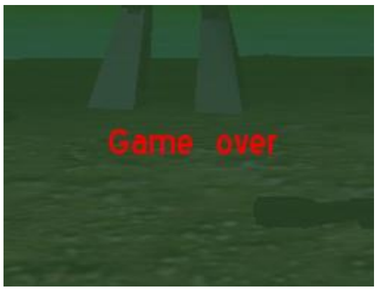


FIGURE 4.2 C – GAME OVER MESSAGE

On the other hand, the health stat could be decreased as well. The only way to decrease the health value is by colliding with an enemy. This is done with a similar way of collision detection as mentioned above. When a player's life reaches zero, the game interface will be updated and the game over sign will pop up on the screen. After that, the player model will be disposed so the user can no longer play the game. (See figure 4.2c).

4.3 LEVEL SYSTEM

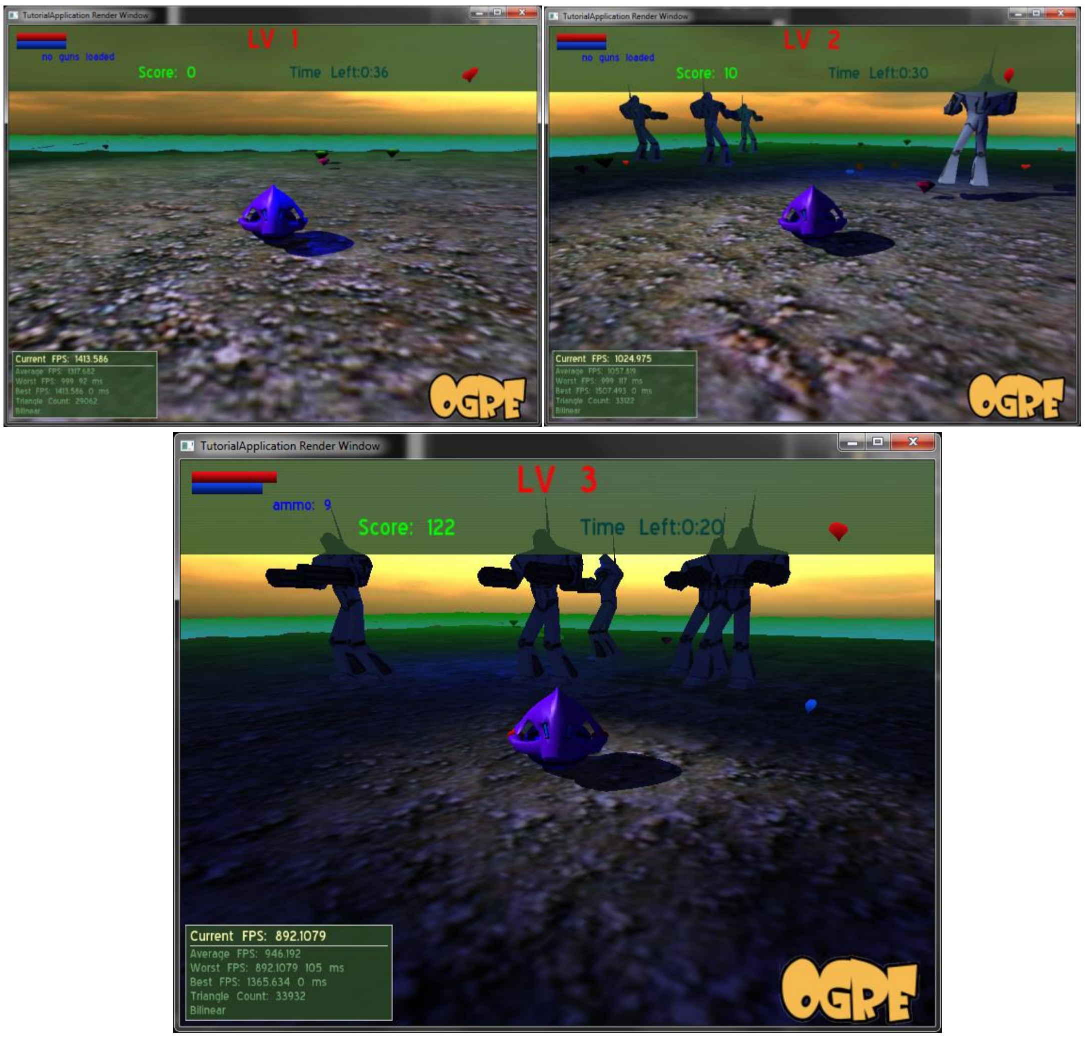


FIGURE 4.3 A – DIFFERENTR LEVEL SCENECHANGES

There are a total of 3 levels in the game. The first level only involves collecting gem; the second involves power-ups and dumb enemies, and the third level involved clever enemies.

When the level goes up depends on the score of the player. When the player reach a minimum score of that level, the UI updated and the scene is filled with elements that is in that level.
How different elements are added to the scene with different levels are done with the update loop in the main class. When a certain level is reached, the create level method is called once only to initialize the scene objects. In addition of game objects, environment of the scene is changed as well, to aware the player that the level state has been changed. (See figure 4.3a).

4.4 GUNS

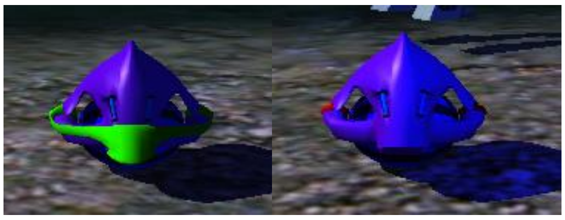


FIGURE 4.4 A – TWO KINDS OF GUNS ON THE PLAYER

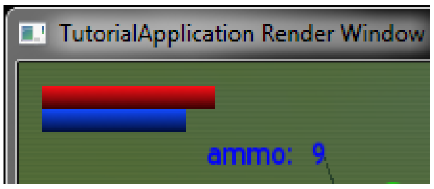


FIGURE 4.4 B – AMMO DISPLAY

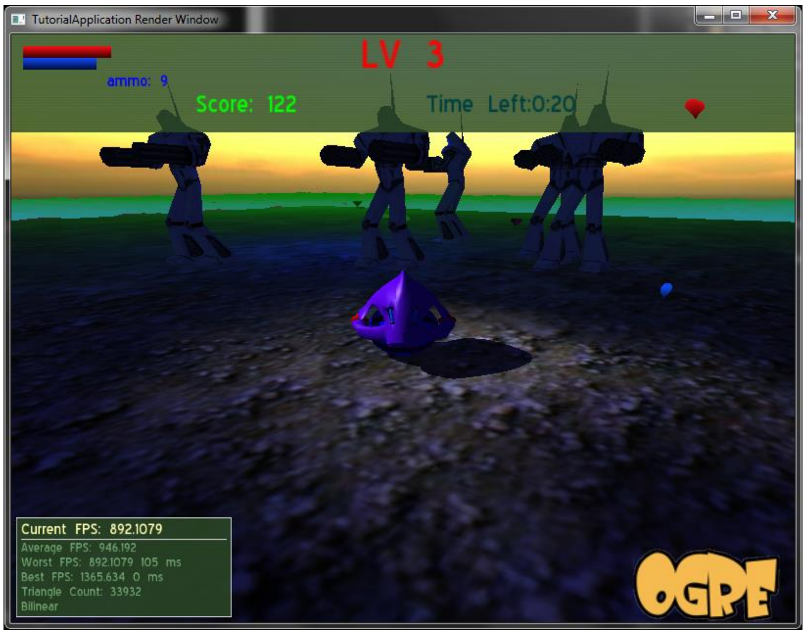
There are two type of guns where player can collect: the Bomb Dropper and the Cannon. They both have 10 ammos loaded in.

In the game, player can collect multiple guns from the world, and will be stored in the player Armoury object through the Collectable gun List with in the class. Once the gun is collected, the gun will be loaded on to the gun node of the player model so it is visible on screen.

To swap a gun, players can simple press the “R” key on the keyboard and the corresponding gun and ammo value will be visible instantly, as seen on Figure 4.4a.

Players can shoot a projectile once they got a gun, by pressing the space bar on the keyboard. A corresponding projectile type will be generated from the Gun class and will be projected on the screen. After the player shoot once, the ammo value will be decrease. Players will be disabled to shoot once there are no ammo left on the gun. (See figure 4.4b).

4.5 ENVIRONMENT



```
Current FPS: 892.1079
Average FPS: 946.192
Worst FPS: 892.1079 105 ms
Best FPS: 1365.634 0 ms
Triangle Count: 33932
Bilinear
```

FIGURE 4.5 A – ENVIRONMENT

The environment of the main scene in the game is composed of many parts, manly the ground, water, sky, shadow and fog.

The ground and water are made by creating 9 plane objects and put them together. The ground is made by applying a single PNG texture, while the water is done using vertex fragment shader to create an animated effect of the water.

For the sky, the Skybox is used to create a natural looking sky. As the Skybox enables setting the six faces to different image, while the player is facing different direction, the sky would look different and that creates a natural feel comparing to the Sky dome which only have a repetitive surface of the sky.

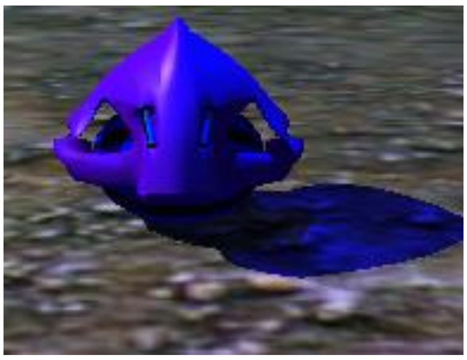


FIGURE 4.5 B – SHADOW OF THE PLAYER

Regarding the shadow, the additive type shadow has been applied to the scene as this type of shadow blend nicely with the texture on the ground so it wouldn't cast a shadow which is purely black, compared to the modulative shadows. (See figure 4.5 b).

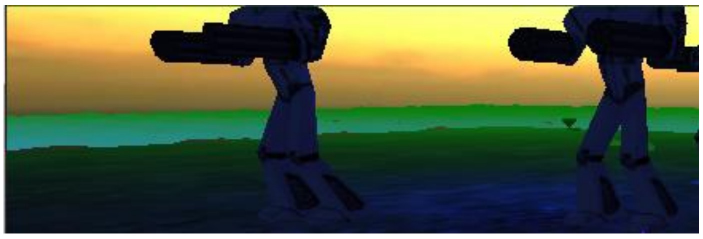


FIGURE 4.5C – FOG IN THE ENVIRONMENT

The fog for the scene is slightly coloured green to create a mysterious feeling. It is done with an exponential square operation as this make the fog grow thicker and thicker depends on the distance from the camera. (See figure 4.5c).

On the other hand, several lightings are applied to the scene.

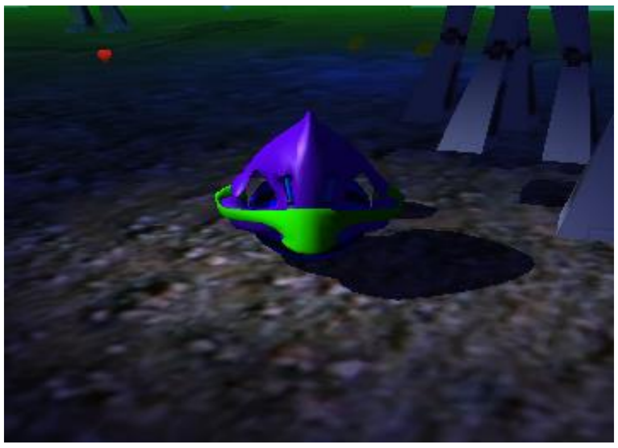


FIGURE 4.6C – SPOT LIGHT IN THE ENVIRONMENT

An ambient light of a colour blue is applied to the whole scene to create a night time atmosphere.

Another spot light is shed and attached on the player so wherever the player moves, the spot light follows. (See figure 4.6c).

This increase the challenge for the player as player could not see the objects far away from the camera. This is done by simply attached the spot light to the player game node.

4.7 ENEMIES AND AI

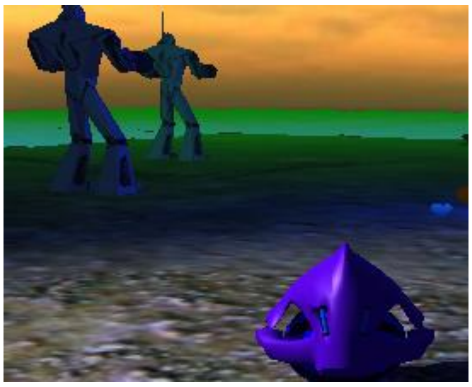


FIGURE 4.7 A – ENEMIES WITHOUT AI

There are two kind of enemies in the game, the first one is dumb enemies.

This kind of enemy is being created on the scene once the player entered level 2. The dumb enemies only do a circular motion in a limited area so the player can shoot them with little challenge. (See figure 4.7a).

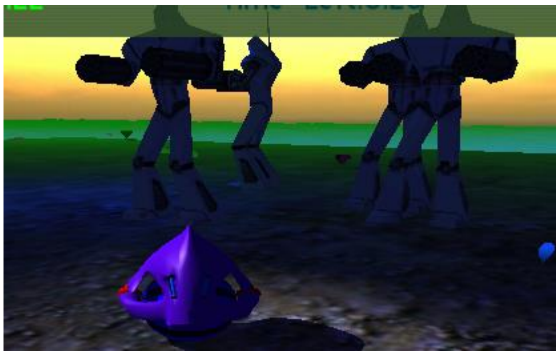


FIGURE 4.7 B – ENEMIES WITH AI

The second type of enemy id the clever enemies – the ones with AI.

This kind of enemy have two kind of activities: one is doing the circular motion as the dumb enemies do, and second is follows the player. Once the player come closer to an enemy, the enemy sense the existence of the player and follows it. To do this, the player object is passed in to the enemy object so the enemy can know the position of the player and hence calculated the direction it needs to go. (See figure 4.7b).

5 CONCLUSION

In this report, the background knowledge of 3D graphic rendering pipeline is introduced, as well as presenting the race game coursework.

In the future, hopefully the Gem Battle game could be development further and more functionalities could be added in.

6 REFERENCES

6.1 The framework, demo codes, model and texture files comes from Marco Gilardi, 2015, Programming for 3D (2015), the University of Sussex.