Programming for 3D Report

Table of Contents

Background

The Rendering Pipeline (or Graphics Pipeline) is the process taken to create a 2D rasterised representation of a 3D scene [1]. Computer Displays can only display 2D images, and so the rendering pipeline uses a sequence of steps to allow 3D models to be shown as 2D images.

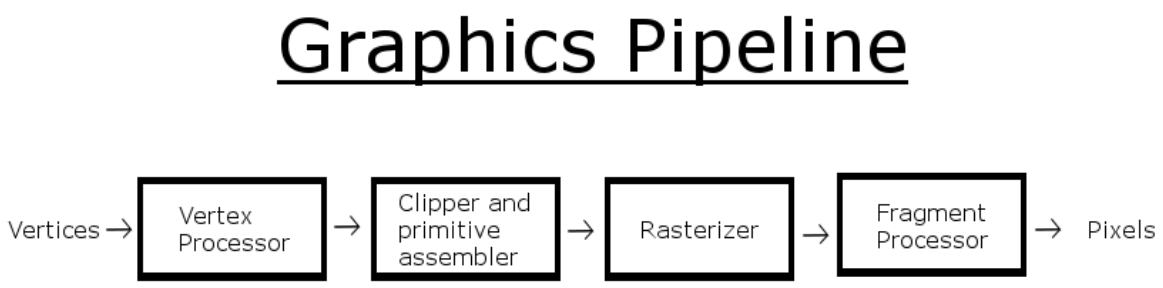

Figure 1 General Rendering Pipeline[2]

Depending on the graphics engine being used, steps in the rendering pipeline may differ but the generalised version being used by most graphics engine nowadays is as shown in Figure 1.

The first step of the Pipeline is Vertex Processing, in which the graphics engine uses a Vertex Shader to process Vertices into output Vertices and then turn said output vertices into a sequence of primitives for the next stage [3, 4].

The next stage is the Clipper and Primitive Assembler, which is in charge of dividing the sequence of primitives into individual base primitives to be rasterised by the next step [5].

Rasterizer then breaks every individual primitives into Fragments, a collection of values representing a sample-sized segment of rasterized Primitives [6,7].

Finally, the Fragment processor then processes the generated fragments into a set of colours and depth value to be displayed as pixels [8].

Game Design Document

Game Mechanics/Loop

The game loop starts with the Main method calling .Go() method. The Physics are updated, then control is given to the player using the .update(evt) method.

The Player class makes sure that its not dead, and if it isn't it updates the ammo values of its playerstats class, animates the model depending on the current event, update the player controller instance to allow for input from the user, updates the gun being actively used, and if said gun has changed, update the player model with the new gun model and checks if the model is colliding with one of the enemy robots, where if the player is not invulnerable then the player takes damage, and if the player's health depletes to 0, the player loses a life, where if the player's lives equal to zero after that then it is updated to be dead.

Once the player instance has been updated, the game interface/HUD is also updated, where it updates its elements depending on the player's stats as well as keep counting down the timer.

The Level is then updated, where all objects in it, such as gems, collectable guns, power ups and enemy robots are updated. If robots are in the current existing level, then its model is animated according to the event, then the controller is updated, which in turn updates its AI, where it figures the position of the player relative to the enemy robot's position and makes the robot model face and walk to the player; If the robot model is colliding with a projectile, it then makes the robot take damage and dispose of itself if it has no more health. The level then checks if all the gems have been collected and if all the enemy robots have been killed, where if they have it will set the level as won (a Boolean).

If the level has been won, the game then creates the next level, where if all the levels have been won, then it shows a game over screen, letting the user know they won. If the level hasn't been won and the timer has depleted to 0 or the player has died, it shows a game over screen letting the user know they lost. Else the game loop starts anew.

Data Flow

Data like player and enemy statistics is stored in instances of the CharacterStats class, which holds statistics such as the characters' health, shield and lives score. The PlayerStat implementation also holds the player's score and ammo count. Data is then constantly updated in the game loop. No data is saved or restored after each instance of a game, as there was no plans to add lasting data for functionality like a scoreboard.

Game Object Data

CharacterStats as a data structure stores the health, shield and lives statistics of a specific character, where the PlayerStats instance also stores the ammo count and score of the player, these attributes needing to be initiated when each instance is created.

The Game Interface as a data structure that displays statistics to the user, requires the PlayerStats to function, where it uses its ammo count, score, health, shield, and lives statistics. When creating the game interface, player stats need to be passed through for it to display said statistics.

Armoury is a database containing all the guns collected by the player, it stores said guns as a list and an individual gun as actively being used. Guns need to be passed through whenever needing to add them to the armoury.

Game Physics and Statistics

The game physics work by calculating the velocity at which an object travels by using the forces which have been applied to said object and then making it move at the speed of the calculated velocity.

There are 3 different types of Forces:

- WeightForce: Force which applies gravity to an object by applying the formula: -(mass * gravity). All objects except the cannonballs have this weight applied to them to give them gravity.
- FrictionForce: Force which applies friction to an object by applying the formula –Friction Coefficient * (Velocity / Normalised Velocity). By applying friction to an object, it makes it move according to friction, I applied this force to the player, robots, gems and powerups.
- ElasticForce: Adds an elastic effect to the force of an object, by applying the formula –(k * (extension – RestLenght) + d * (dv * direction)) *direction. By applying this force to an object, it would make it act like an elastic, where it'd try to stabilise back to the original

point. I didn't apply this to any object in the game because I deemed it unnecessary, but I have implemented the formula correctly.

AI

The Artificial Intelligence that I have applied to the enemy robots takes into account the position of the player as well as the position of itself, calculates the distance and moves towards the player according to that distance. By also rotating according to its own x and z axis position and the player's y axis position, it makes itself look at the player as they move.

Player Controls

Mouse Movement – Control camera

W – Move Forward

A – Move Left

D – Move Right

S – Move Backward

Q – Reload Ammo (If a gun is equipped)

E – Swap Gun (If more than 1 gun is collected)

Spacebar – Shoot (If a gun is equipped)

Advanced Functionality

Power Ups

I have added 3 different types of collectable Power Ups: collectable health, shield and lives pickups. When creating a power up, the specific stat that the power up updates when picked up. For the health and shield power ups, I used the sphere model used in the player's model and added red and blue gradients to them correspondingly, as well as adding vertex shaders and rotating them for their animation. The lives power up uses the same heart model used in the game interface and rotates for its animation.

Collectable Guns

I've implemented 2 types of collectable guns, the cannon and bomb dropper. The cannon spawns cannonballs which fly straight in front of the player and disappear after 2 seconds or if they hit an enemy of if the player swaps guns or reloads. The Bomb dropper drops a bomb behind the player, which have weight and friction forces applied to them so that they do no move too fast after being spawned. The bombs disappear after 5 seconds, or if they hit a robot or if the player reloads or swaps guns.

Basic AI

The Basic AI implemented on the enemy robots, passes through the player's position so that they move towards it and its own position with the y axis changed to the player's position's y axis so that it rotates towards it, but the model doesn't actually face forward so I rotate it 90 Degrees.

Vertex Shaders

I've implemented 2 versions of vertex Shaders and applied them to the health and shield power ups. The health power up's vertex Shader creates vertical waves by calculating the y axis position of each

vertex using the formula wave = (cos(20*Pos.x+5*time)+sin(20*Pos.x-Pos.z+10*time)) and then doing the y axis = wave * 0.8 to reduce the length of the waves a bit. Similarly, the shield power up's creates horizontal waves using the formula wave = (cos(20*Pos.x+3*time)+sin(20*Pos.x-Pos.z+5*time)) and then changing the x axis to wave * 1.11 to increase the length of the wave a bit.

Randomly Generated Levels

I've created a LevelGen class which takes in an instance of Level Stats, which contains the number of gems, health, shield, life power ups, enemies and guns in the level, and then for each of them randomly generate their position on the map, while also randomly choosing the amount of red to blue gems in the game. If there are 15 gems in a level, the amount of red or blue gems change each time.

Game over Screen(s)

I've added another interface which disposes and hides the previous game interface and displays a message telling the user if they've lost or won, as well as showing their score and the amount of time that is or isn't left.

UML Class Diagram

References

[1] Wikipedia, (2016), Graphics Pipeline [online], Last Accessed 4th May 2016. Available at:

https://en.wikipedia.org/wiki/Graphics_pipeline

[2] Wikipedia, (2016), Graphic Pipeline New Tech [online], Last Accessed 4th May 2016. Available at: https://en.wikipedia.org/wiki/File:Graphic_Pipeline_New_Tech.png

[3] OpenGL.org, (2016), Vertex Processing [online], Last Accessed 4th May 2016. Available at: https://www.opengl.org/wiki/Vertex_Processing

[4] OpenGL.org, (2016), Vertex Post-Processing [online], Last Accessed 4th May 2016. Available at: https://www.opengl.org/wiki/Vertex_Post-Processing

[5] OpenGL.org, (2016), Primitive Assembly [online], Last Accessed 4th May 2016. Available at: https://www.opengl.org/wiki/Primitive_Assembly

[6] OpenGL.org, (2016), Rasterization [online], Last Accessed 4th May 2016. Available at:

https://www.opengl.org/wiki/Rasterization

[7] OpenGL.org, (2016), Fragment [online], Last Accessed 4th May 2016. Available at:

https://www.opengl.org/wiki/Fragment

[8] OpenGL.org (2016), Fragment Shader [online], Last Accessed 4th May 2016. Available at:

https://www.opengl.org/wiki/Fragment_Shader