Game Project Report

ALONE

Submitted by

Team Members	Student ID
Fahim Shahriar Shakkhor	*154440*
Shahriar Ivan	*154435*

CSE 4513 Software Engineering

Islamic University of Technology

21 May, 2018

Table of Contents

1.0 Introduction

The report describes the process involved in making a 3D graphical first person PC game built with Unity game engine. Using unity game engine, we created an open-world 3D game where the objective for the user is to explore the each of the different worlds and collect necessary items which will help him to get out to the final destination.

2.0 Game Overview

The primary objective of the game is for the user to search a way to the final destination. Throughout the game world, there are various items placed on different locations which the user can pick up and can be used for various purposes. This section elaborates the description of the gameplay and user controls.

2.1 Game Description

In the game, the player is a human character who is lost in a new world. His objective is to find a path to the final destination. He can travel through worlds using portals. The portals are placed somewhere in the world, which the player has to find out by exploring the world. There will be some items in some world which can be put in the player's inventory. He can take these items to other worlds and use them. We intended to create a few items for the game. But as we are at the end of this semester, we implemented only a torch which can be used as a light source. The torch needs to be used in a dark room where there is another item placed on a table.

2.1.1 Game Objects

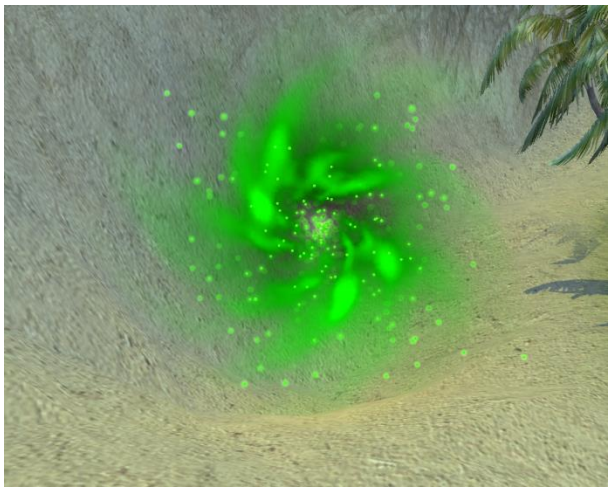
The Torch

User can pick up the torch and store in the inventory. He can use the torch pressing the L button

The Pen

In a dark room, there is a pen on the table. The player has to use the torch to find out the pen on the table.

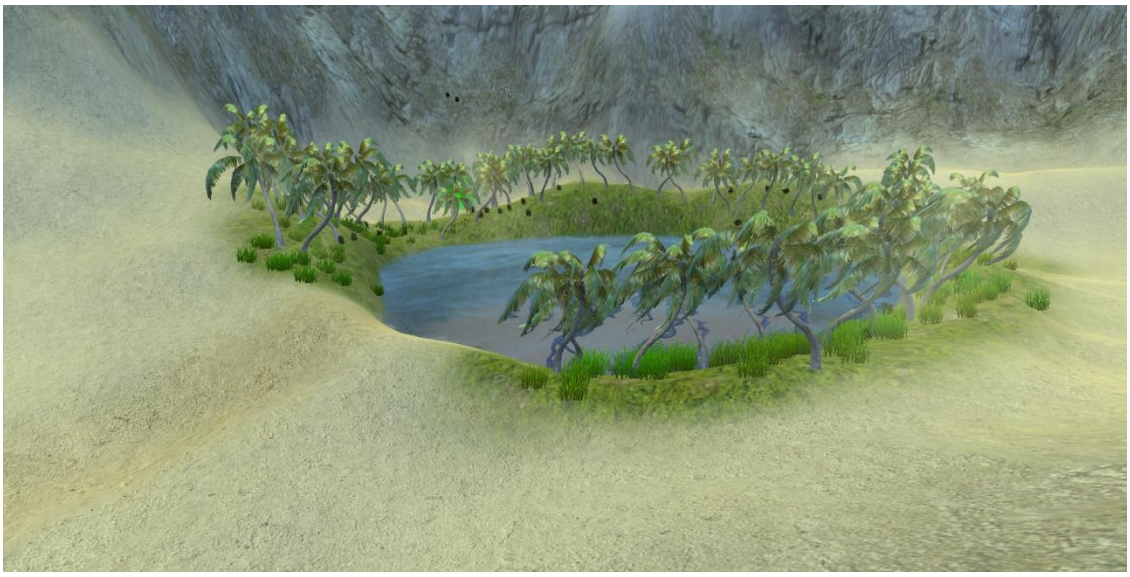
The Portal



Portals are used to travel between worlds.

2.1.2 Game Environments

The Pond



The pond is located in the very first world of our game. There is a portal behind the pond. Currently it serves no purpose, but we intended to hide an item (say, a key) underwater.

The Camp

The green portal takes the player to the jungle. He spawns before this camp.

Blood Trail Scene

In the second world, the player finds out a place with blood and a torch near it.

The House

The player will find this lonely house in the second world. There are some items of interest here, but their functionalities have not yet been implemented.

2.2 User Manual

The user controls the game using particular keys on the keyboard and mouse.

Direction Keys:

- **Key 'W' :** Player moves forward
- **Key 'S' :** Player moves backward
- **Key 'A' :** Player moves left
- **Key 'D' :** Player moves right
- **Key 'Shift' :** Player sprints
- **Key 'Space' :** Player jumps

Other:

- **Key 'ESC' :** Displays the Pause menu
- **Mouse :** Changes the view angle
- **Key 'E' :** Interacts with portals and objects
- **Key 'L' :** Light a torch

3.0 Game Design

In creating the game, the primary tool used was Unity, although different assets were imported from different sources, namely: Blender, 3D Studio Max, Maya etc. For sound effects, we had to download audio clips from different audio sharing websites. Some of the resources were directly implemented by ourselves, such as the Portal Flare particle system. The following section describes how different aspects of the game was implemented.

3.1 Camera and Character Controller

Throughout the main part of the game, the player explores the game world in the first-person view. The player is able to look around from his current position using the mouse. To achieve this, a First Person Character Controller was built and a Camera component was attached

with it. Different scripts were written to handle the input and the corresponding behavior of the controller. Some scripts dedicated to the character controller as used in our project was:

- FirstPersonController.cs for the Character movement
- HeadBob.cs for handling the camera bob while moving
- MouseLook.cs for associating mouse movement with camera rotation.
- RigidbodyFirstPersonController.cs for handling physics calculations on the character.

3.2 Lighting

For generating light in the game world, we customized the built-in Directional Light GameObject in Unity to create our very own light source for each scene. We also have ambient lights separately defined for each scene. All lights are generated prior to scene loading and applied as Texture.

3.3 Terrain generation

Each of the game world holds a terrain built using Procedural Generation and Procedural Texturing technique. A terrain is basically a heightmap holding opacity values for each point of a two dimensional grid. Unity provides built-in functionality to heighten or lower different parts of the terrain and renders the 3D terrain in realtime. Different environmental details such as: Trees, Bushes, Rocks etc were obtained free of cost from the Unity Asset Store.

3.4 Collision Detection and Triggering

Collision detection has been implemented using the built-in Collider component in Unity. For making collision detection computationally light, we used simple 3D shapes as colliders for the solid objects. Unity automatically handles movement obstruction for solid colliders. For detecting entry of an object inside a specific location, we used colliders that we defined as “Triggers”. These triggers are special colliders that do not obstruct movement but instead run a script when being collided with. This was especially necessary to detect whether the character was within the range of a portal.

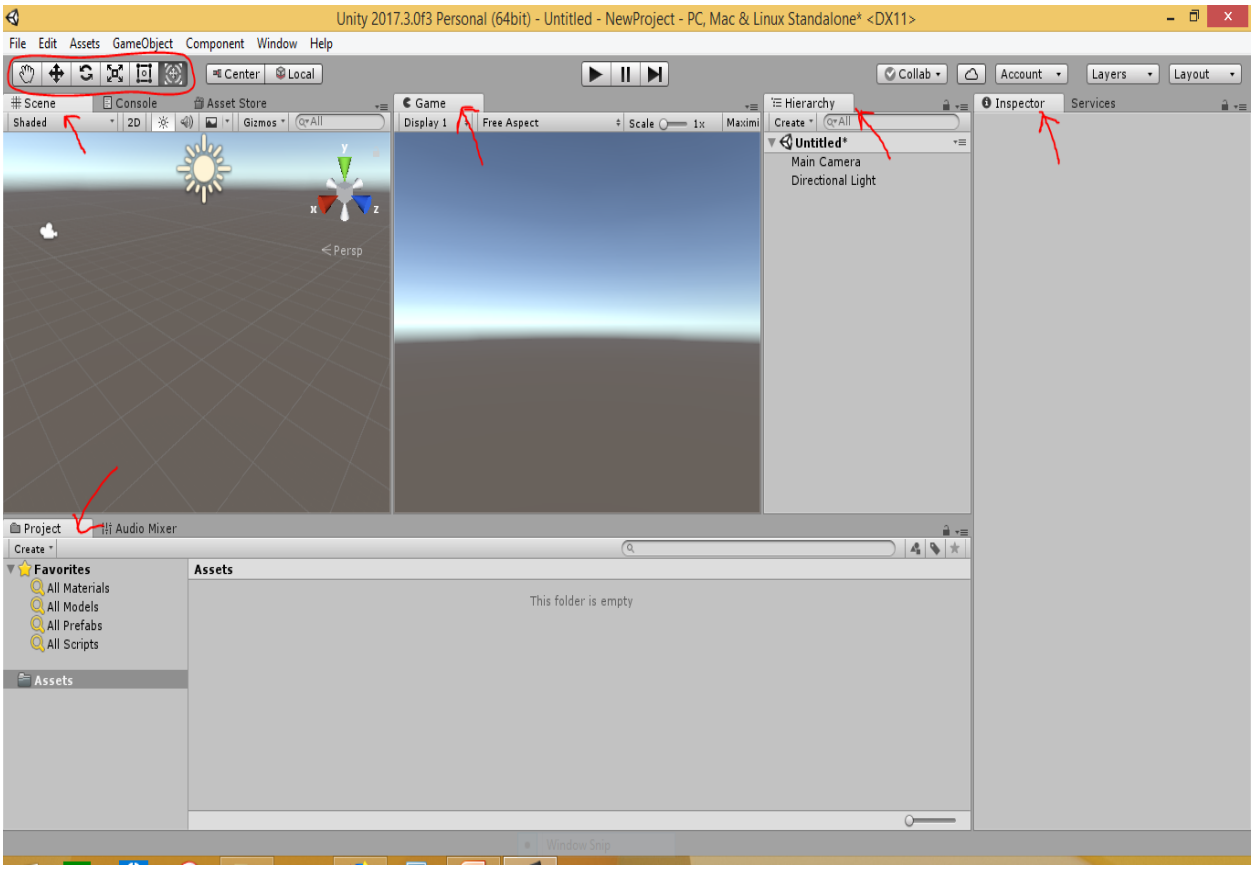
3.5 Particle Systems and Object Pooling

Different environmental effects like: Sandstorm, Snow, Fire, etc were generated using the Unity Particle System. One specific particle system, namely the Portal Flare, was custom developed using the Object Pooling technique.

By Object Pooling, we maintain a Dictionary of prefabricated objects and instantiate them into the game world and add them to a Queue Data Structure in order. Then in the update thread, we simply dequeue each of the objects from the queue and provide them with a random motion and enqueue them into the queue again. Thus, the objects are being infinitely recycled without exhausting the system resources.

4.0 Game Engine Overview

4.1 Unity User Interface



- The Tools section lies in the top-left portion of the window. Its main purpose is to allow movement, rotation and scaling of objects in the Scene View.

- The Tools section lies in the top-left portion of the window. Its main purpose is to allow movement, rotation and scaling of objects in the Scene View.

- The Game View shows the objects in action when we press the Play button. The View is based on the location of the camera with the tag “Main Camera”.

- The Hierarchy view shows all the objects (together with their inheritance relations) that are currently on the scene that we are working on.

- The Inspector tab shows the details of an object that we have currently selected.

- The Project tab shows the Assets currently linked to our project, such as Scenes, Prefabs, Audio, Materials, Textures, Scripts etc.

4.2 Unity Threads and Co-routine

Threads: Threads are used to run multiple process in parallel. Each thread execute sequence of programming instructions. In a game, there are many computations need to be done simultaneously. If the script follows a linear flow of control, then it is not possible to handle parallel computations. That’s where threads are used.

```
bool goingLeft;
private void Update()
{
    MoveObject();
}
void MoveObject()
```

```
{
   transform.localPosition += Vector3.right * (goingLeft ? -.01f : .01f);

   if ((goingLeft && transform.localPosition.x < -3) ||
      (!goingLeft && transform.localPosition.x > 3))
            goingLeft = !goingLeft;
}
```

Here **Update()** is a built-in thread in unity. We have created a method **MoveObject()**. If we call the method in the thread, the method will be executed once every frame for the object which has this script as a component.

Thread is an expensive operation. Whenever we use a thread, the game engine needs to use different resources. Although it is resource hungry, threads are must for synchronization in a game.

Co-routine: Unity has a functionality called co-routines that can be a substitution for threads in some cases. Unity coroutines use concurrency and threads use parallelism. Coroutine methods can be executed piece by piece over time, but all processes are still done by a single main Thread. If a Coroutine attempts to execute time-consuming operation, the whole application freezes for the time being. Here is an example.

```
using UnityEngine;

using System.Collections;

public class ExampleClass : MonoBehaviour

{

    IEnumerator WaitAndPrint()

    {

        // suspend execution for 5 seconds
```

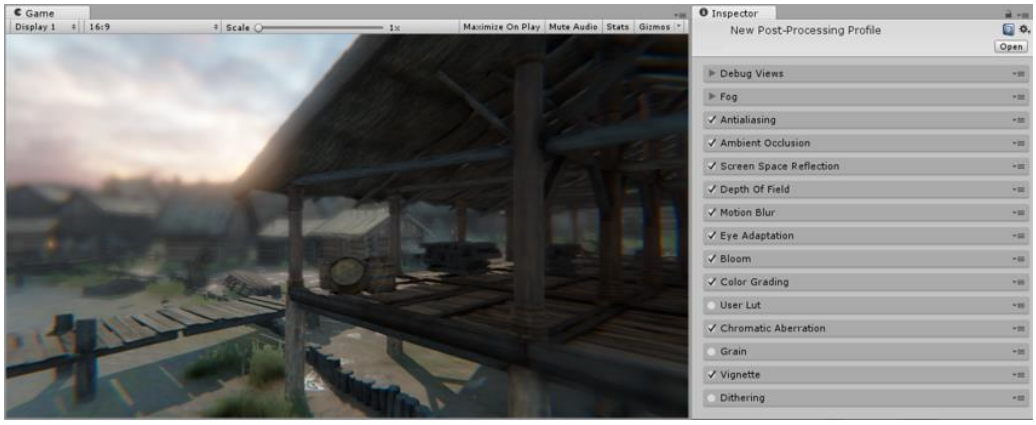
```
        yield return new WaitForSeconds(5);

        print("WaitAndPrint " + Time.time);

    }


    IEnumerator Start()

    {

        print("Starting " + Time.time);
        // Start function WaitAndPrint as a coroutine

        yield return StartCoroutine("WaitAndPrint");

        print("Done " + Time.time);

    }

}
```

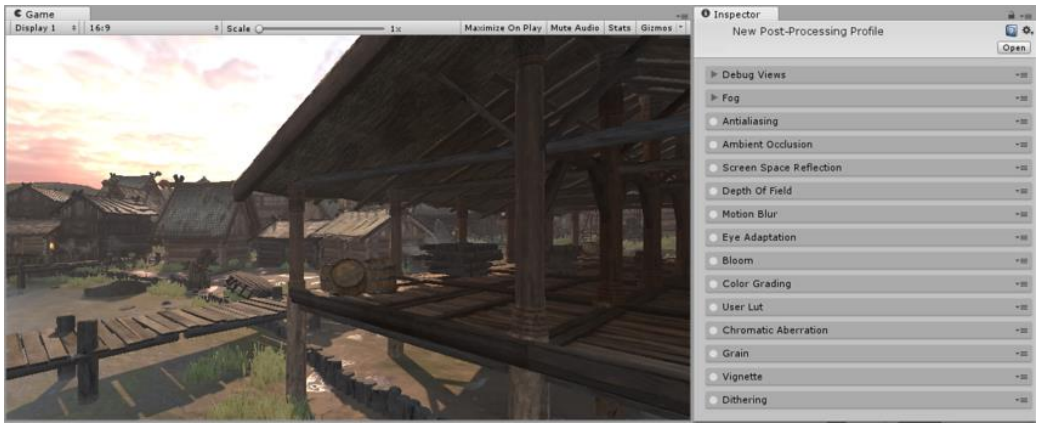
4.3 Post Processing

Post-processing is the process of applying full-screen filters and effects to a camera's image buffer before it is displayed to screen. It can drastically improve the visuals of your product with little setup time.

You can use post-processing effects to simulate physical camera and film properties; for example; **Bloom, Depth of Field, Chromatic Aberration** or **Color Grading.**

The images below demonstrate a Scene with and without post-processing.





4.4 Collision Detection

Collision detection is the computational problem of detecting the intersection of two or more objects. In addition to determining whether two objects have collided, collision detection systems may also calculate time of impact. Solving collision detection problems requires extensive use of concepts from linear algebra and computational geometry.

In the a posteriori case, we advance the physical simulation by a small time step, then check if any objects are intersecting, or are somehow so close to each other that we deem them to be intersecting. At each simulation step, a list of all intersecting bodies is created, and the positions and trajectories of these objects are somehow "fixed" to account for the collision. We say that this method is a posteriori because we typically miss the actual instant of collision, and only catch the collision after it has actually happened.

In the *a priori* methods, we write a collision detection algorithm which will be able to predict very precisely the trajectories of the physical bodies. The instants of collision are calculated with high precision, and the physical bodies never actually interpenetrate. We call this *a priori* because we calculate the instants of collision before we update the configuration of the physical bodies.

5.0 Conclusion

In the game **ALONE**, we have implemented a game environment that includes 3D viewing and objects with a moving camera, lighting and material variations and texture mapping. We have also used features like gravity, collision detection. In this game, we have implemented object pooling technique to make the portals more realistic. We have also used sound effects . Moreover, the game is challenging, interesting and enjoyable to play with.