UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

UNDERGRADUATE SCHOOL

Research and Development

BACHELOR THESIS

By

Bui Vu Huy

USTHBI7-082

Information and Communication Technology

Virtual World Development using Unity Engine

Supervisor: Dr. Nguyen Hoang Ha

Hanoi, July 10, 2019

Table of Contents

Acknowledgements

First of all, i would like to thanks Dr.Nguyen Hoang Ha for giving me 3 months for using the Unity Engine to develop the Virtual World. Also, thank you for supporting me during this internship.

I also thanks for the USTH ICT Lab for giving me a opportunities to work in a places like in a professional company.

Finally, I'd like to give special thanks to the guys, girls who made free 3d models, characters in the Unity assets store, so that i can download, use them for my project.

A. Introduction

Nowsadays, the technology is getting better and better, so the human life are becoming more and more convenient. With that, people's entertainment needs are also increasing. However, to satisfy people's needs, also to keep up with this rapid development of technology, a virtual world is also a type of entertainment that can satisfy people's need.

A virtual world is a computer-based environment that the user can interact with each other in the world. In general, the virtual world usually using 3-dimensional graphic, with 3D models, which can make people feel like they're in the real world. Virtual worlds allow for multi user to communicate, and a 3D video single player game like Elder Scroll: Blade and Skyrim can still be consider as the virtual world.

In general term, there's still no generally accepted definition for virtual world, it supports varying degrees of play and gaming. These are some uses of term: MMOGs game: large number of players within a game, and RPG game, etc...

To create a virtual world, there're some types of engine that allow you to make like Unity, Unreal Engine, blender, etc...

From one of those engines, Unity is the most popular choice for many people, from beginner to expert.

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. As of 2018, the engine had been extended to support more than 25 platforms. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences.

In my opinion, first of all, it has tons of online tutorials to be found, the document with clean format, so a good step for beginners. With it's very intuitive design, C# language make it easier to use or learn, like you can call other script within a script. Also, with great community, when you have a problem, they'll have you everywhere and every time. Unity has clean API, which is easy to use, understand, implement in C# code, the application works very well on windows, Linux, android or iOS. Lastly, unlike other engines, the unity has asset store with lots of free assets for everyone to use, while the other ones only have few free assets.

In this work, i'll focus on creating the virtual world using unity engine, in particular, it's an RPG game, where you as a player can walk around, interact with other NPC model in the world, or attack some enemy that get in your way.

To create the virtual world with Unity, first, we'll need to create the terrain by the tools Unity provides for us. After that, we can create my own model and then just simply drag and drop the model into the scene, if not, there're several websites i can get free 3d model from like turbosquid or free3d. For the Lighting and the shadow, Unity also has build-in tool just like other engines, which can help me

create the light and shadow easily, adjust it as i want. Furthermore, to make things look good, we can apply some textures to all the objects, prefabs. We can find the texture on google, choose the suitable one for a specific object, and then just simply drag and drop them on that object, or making our own texture. Finally, in order to move around or interact with the object, a camera and player controller should be added into the scene, however, Unity already has the FPScontroller prefab which can be found in the standard assets, we can use it directly without doing anything, but it's still possible to create the character controller from scratch with C# code.

B. Objective

In this project, since it's about building an RPG game, which is also a type of virtual world, I'll only focus on how to make the player move around the world, is able to interact with other objects around him. In this game, there may be some enemies that get in the way, the player should eliminate them which hurt the player.

C. Aim

The aim is not necessarily on destroying the enemies, but focus on completing the quest that villagers in the village gave the player. When finishing a quest, the player should return back to the village, talk to the villagers that you're finished. Beside that, the game should also have an online chat tab , which you can chat with other players who are connecting to the same network while playing game, they may help you finishing the quest faster.

D. Programs, Materials and Methods

1. External programs used

1.1. 3DS Max

3DS Max, a program developed by Autodesk, previously called 3D Studio Max, is a 3D computer graphics program for making 3D animations, models , images... This program was used in this project in order to edit UV mapping of some models, correct the position of the texture.

1.2. Photoshop

Photoshop is a photo editing and graphic design software. It is developed by Adobe Systems for MacOS and Windows. It's not like other graphic design or photo editing softwares, it can create normal map, height map, occulusion map from a single texture. With this, this software will help me with the technique bump mapping, so the game's texture will look more realistic.

1.3. Visual Studio

Visual studio is an IDE (integrated development environment) from Microsoft. It's used to develop computer programs, as well as websites, supports many programming languages such as : C++, C#, JavaScript, etc... This is the main software of this project, because i'm using this one to write scripts in order to make the game works. When installing unity 5, normally it's shipped with visual studio community which is free for everyone. Previously, it's shipped with the monodevelop, but now the monodevelop is discontinued, no longer support unity, replaced with visual studio community as it's more powerful than monodevelop, it support MacOS and windows, gives you a cloud storage for saving like one drive(when you project goes wrong), which monodevelop doesn't have.

2. Materials

2.1. DirectX

DirectX is the collection of API (or application programming interfaces), also made by Microsoft, for handling task related to multimedia. It contains these APIs such as Direct3D, DirectDraw, DirectMusic, DirectSound .... , especially for game programming.

The DirectX version use in this project here is Direct X11 because from unity 4.x or higher, the engine tend to use more cpu cores, and with Direct3D 11, it has improved multi-threading support so it can utilize multi-core better. Without Direct X, the engine cannot simulate the light in the game or even run.

2.2. A dedicated graphic card

A graphic card with direct x11 compatible is compulsory for Unity. Because unity comes with MSAA support, to improve image, textures quality, which direct x10 or lower doesn't support. Integrated GPU is fine also as long as it's powerful enough to run unity programs, games and have direct x11 compatible.

3. Method

3.1. Overview Diagram

- In this section, i'd like to introduce my overview diagram about the concept of creating a virtual world:

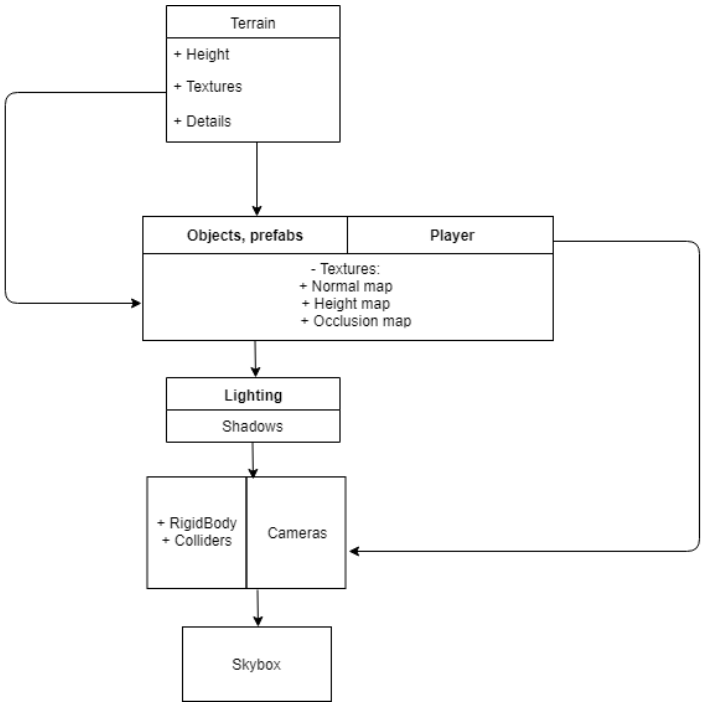


Figure 1: Overview Diagram

3.2. Details

3.2.1 Preparation

Unity can be download from the offical website. I prefer Unity personal version because it's free, for everyone. As i mentioned before, to be able to run Unity, and create a new project, you'll need at least direct x11 for simulate lighting, shadows on the scene. To create a terrain, which is the base of the virtual world, all i have to do is go to game object -> 3D Object -> Terrain. The terrain width and length is only set to 500x500, which i think is big enough for this project.

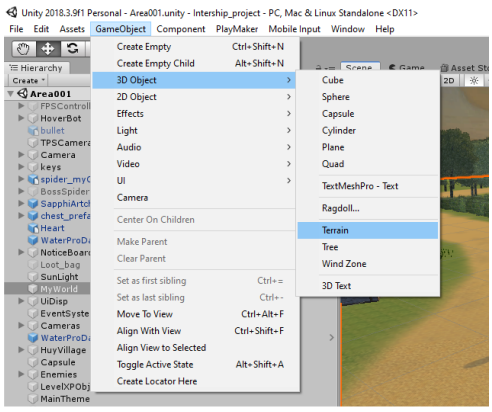


Figure 2: Create terrain

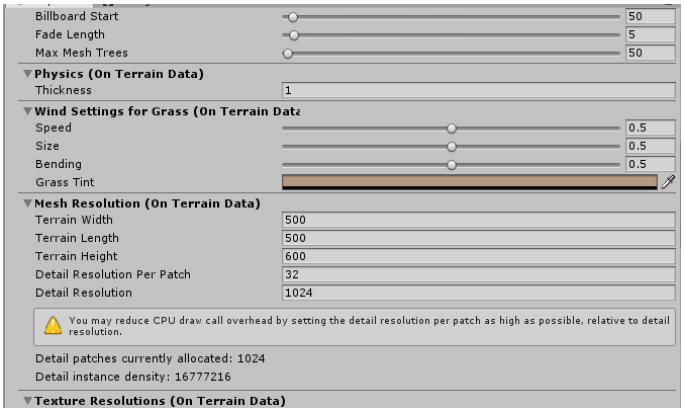


Figure 3: Properties

From the Inspector section, there're 4 sections: Paint texture, paint trees, paint details. Each of these sections has painting tool with 20 brush presets, resizable brush size, opacity. For the textures, you can use any texture you want for the terrain, for example i use this texture from this website:

https://www.textures.com/download/grass0153/48704. The same applied for painting details, but for painting trees, the material for this one is 3D model, however it can be found in the Standard Assets which comes with Unity when installed.

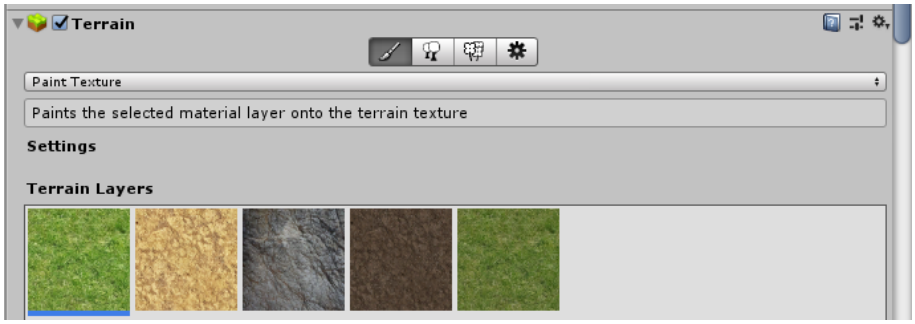


Figure 4: Sections

Besides, in order to make the terrain looks better, the user can also raise or lower the terrain to create mountains, river. After i finish creating the terrain, all i have to do is drag and drop other objects/prefabs into the scene, put the textures corresponding to each object, while these prefabs/objects can be found on unity assets store or from free3d website as i mentioned from the introduction.

3.2.2 Player

To be able to interact with the objects, NPCs, a player should be added into the scene. For example, i use this little robot for my player, this model is free to download, use: https://free3d.com/3d-model/bb8-35865.html.

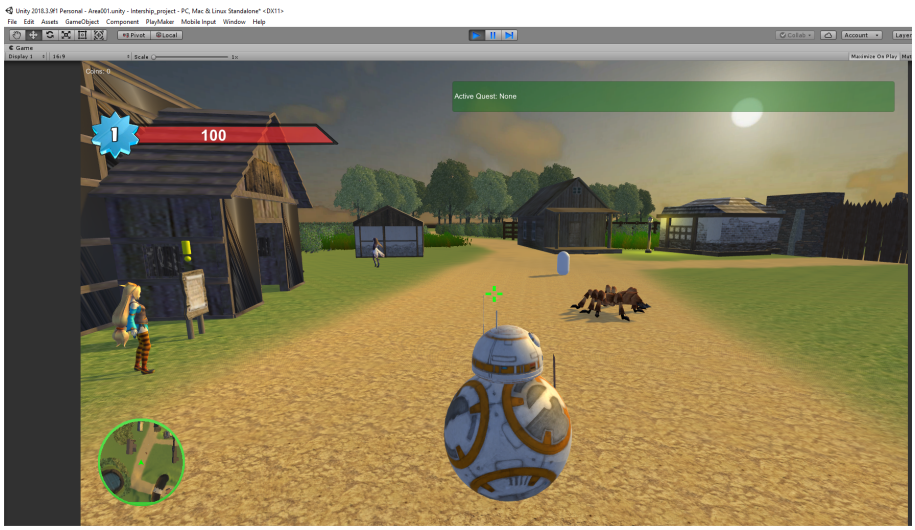


Figure 5: Character example

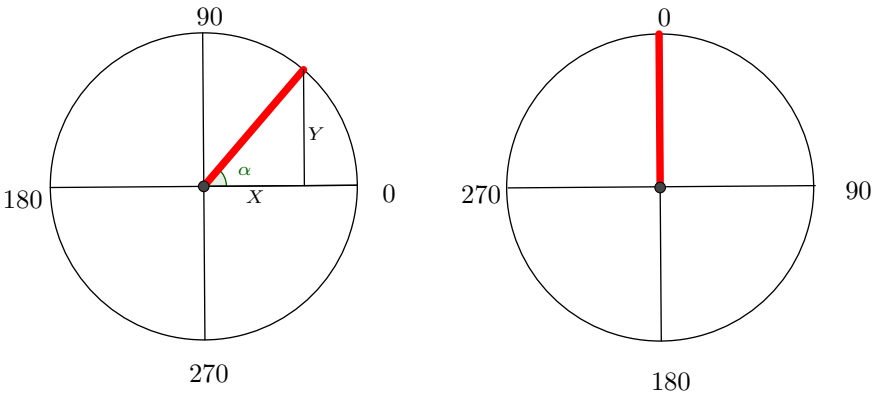
Initially, the player will not move without any scripts attached to it, so i write a simple C# script to control the player using visual studio and then attach the script to the player. Although there's a character controller prefab in the Unity's standard asset, which can be used, controlled instantly without doing anything. In this project, i'll not use character controller prefab, so that i can easily customize my character controller script later.

By default, when initializing a unity project, there's already a camera called "Main camera", which is what the user can see from the game view in the hierarchy. This camera will not follow my character as i move, but i'll explain how to make the camera follow my character later.

From now, i'm going to explain how can i implement character controller to move my character. Firstly, i have to find the input by go to Edit -> Project Settings -> Input. In the input section, as you can see, the Unity already implement 2 inputs named : "Horizontal" , "Vertical", along with Negative Button and Positive Button, and i can also add my own custom input in project setting. This is where i can move my character left and right, forward and backward. After finding the input, i can get the input by using:

```
Vector2 input = new Vector2(Input.GetAxis("Horizontal"),
Input.GetAxis("Vertical"));
```

Next, i have to normalize the input vector to see which direction the player have to move , face correctly. To be more specify, let's consider two trigonometry below:



From the left one above, suppose that the red line is the input direction, X and Y are vertical and horizontal component. Now, i have to find the angle α which is the direction the character will rotate, calculated as:

$$\alpha = arctan(\frac{Y}{X})$$

In Unity, however, if my character is facing forward, then it has the rotation of 0 (as seen from the right one), when facing right it has the rotation of 90 degree and so on. Let's call the rotation of character in unity is r, when looking at the two circle, it's clearly that r is calculated as:

$$r = 90 - \alpha; \qquad or \qquad r = arctan(\frac{X}{Y})$$

By C# code:

```
float targetRotation = Mathf.Atan2(inputDir.x, inputDir.y) * Mathf.Rad2Deg;
transform.eulerAngles = Vector3.up * Mathf.SmoothDampAngle(transform.
    eulerAngles.y, targetRotation, ref turnSmoothVelocity, turnSmoothTime);
```

Where Mathf.SmoothDampAngle is the function that can gradually change an angle given in degrees towards a desired angle by turnSmoothTime (in second). The reason i use Vector3.up is because the character won't rotate up or fly straight to the sky.

After that, to be able to move, transform.forward and Controller.Move is used here to move the character by a certain amount in the world space everytime when i press a button.

Finally, to finish setting up my character controller script, i have also added gravity, ability to jump for my character. From 10th grade, the equation to calculate the falling speed is defined as:

$$v = \sqrt{2gh}$$

where g is the gravity and h is the height the character begin to fall. As there's no air resistance in the game, this equation can be applied directly to the character through the script. For the ability to jump:

```
if (controller.isGrounded)
{
        float jumpVelocity = Mathf.Sqrt(-2 * gravity * jumping);
        VelocityY = jumpVelocity;
}
Vector3 velocity = transform.forward * currentSpeed + Vector3.up * VelocityY;
controller.Move(velocity * Time.deltaTime);

if (controller.isGrounded)
{
        VelocityY = 0;
}
```

Where VelocityY is defined as the falling speed equation above. As soon the character hit the ground, the character won't be able to fall anymore, so i have to assign VelocityY = 0 when controller.isGrounded.

3.2.3 Cameras

- **TPSCamera (Third Person Camera)**

Without the camera following the character, i cannot see what happened to him when he move outside the camera view, so in this section, i'll make a script to force the camera follow the character. To do that, let's implement transform.position = Target.position - transform.forward * distFromTarget , where the Target variable is the character object. The reason why i minus with transform.forward * distFromTarget is because to make sure that the camera always behind the character, not in front of him.

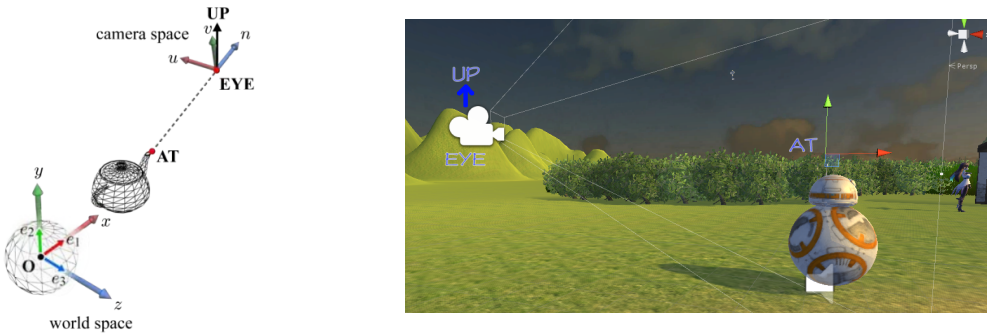


Figure 6: Camera world space

The camera is specified in term of three parameters: **EYE, AT**, and **UP**, as show in the left one. EYE

is the camera position , AT is the reference point the camera is pointing at, and UP usually is set to the y-axis of the world space. For example, in the right one, AT is set at the position a little bit above the character for not covering the center of the screen which i'm going to explain why later. After that, the camera needed to be rotated around with a mouse because i'd like to see what is happening around the character. And the same applied to the character controller script, but this time, i use Input.GetAxisRaw function instead of GetAxis is because i don't want the rotating to be smooth immiately as it can cause a short delay as GetAxisRaw will only return 0,-1 or 1 while GetAxis change gradually from 0 to 1 or 0 to -1. Moreover, the function Lerp is used here instead of SmoothDamp due to the fact that the smoothdamp function adds the curve which can cause delay, for example, when i stop moving my mouse, the camera still moving, while Lerp only behaves linear. As now, there's still one more problem, on the mouseY rotation, the camera can be rotated 360 degree which is not i wanted, so i used the function Mathf.Clamp to limit the rotation of the mouseY axis, the minimum, maximum here is -40, 85 degree for the third person camera controller. In the end, to finish my TPS camera controller, i've modified the charactercontroller script a bit by adding cameraT.eulerAngles.y in the targetRotation where cameraT is the MainCamera(the TPS camera), to make sure that the camera will following the player correctly.

- **Camera Collision and Occlusion Detection**

In some situations, like in the figure below, when the camera collide with the wall or the wall comes in between the character and the camera, they're called occlusion and collision. When the camera goes through the wall or the ground, it can cause shearing.

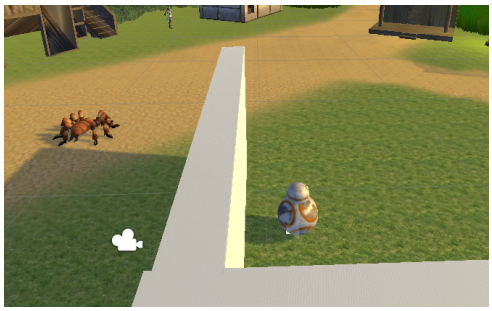


Figure 7: Occlusion example

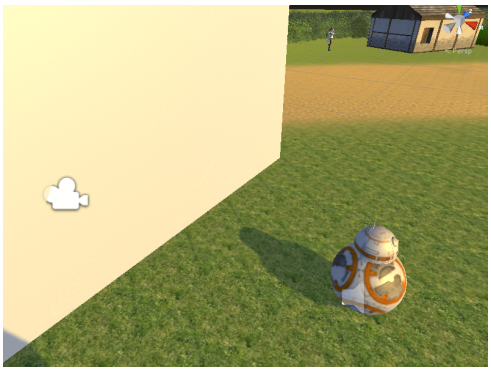


Figure 8: Collision example

So, the goal here is to handle **Collision** and **Occlusion** with as little shearing as possible. Let's defined each one of these underlined terms really means.

Occlusion is when the game is running where the pillar or the wall comes in between the character and the camera view, the character is no where to be seen, and i need to move forward if that happen. Like the figure above, if camera isn't hitting anything but it's view is being obstructed so i have to move forward, the camera can see the player.

Collision is where the camera actually does hit something and the view is still obstructed so i still need to move the camera forward.

From these, it's actually a problem that can be solved with one solution. I'm going to determine if the

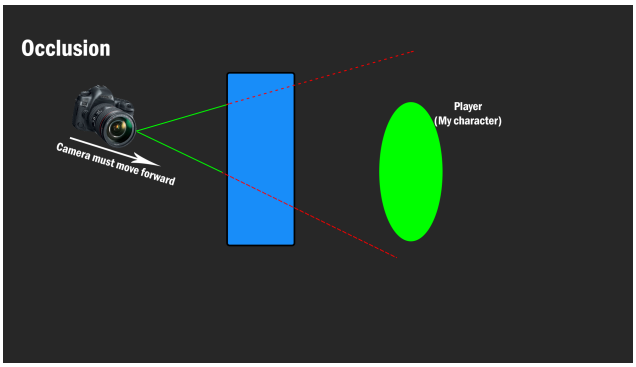


Figure 9: Occlusion

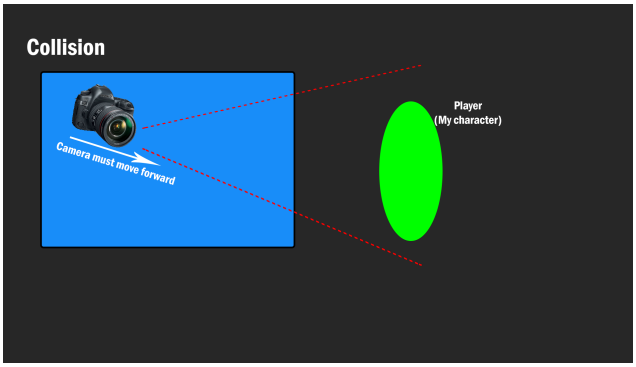


Figure 10: Collision

camera can see the player , if a point of the camera cannot see the player then the camera should be moved forward, whenever it's collision or occlusion, it's still the same issue. And shearing is when the camera go through the wall, i'm able to see through the wall too, to where i see the open space, even the skybox. It's really detract from the realism of the scene,so i have to limit the shearing as much as possible.

Before solving the problem, let's implement some definition of the Camera.

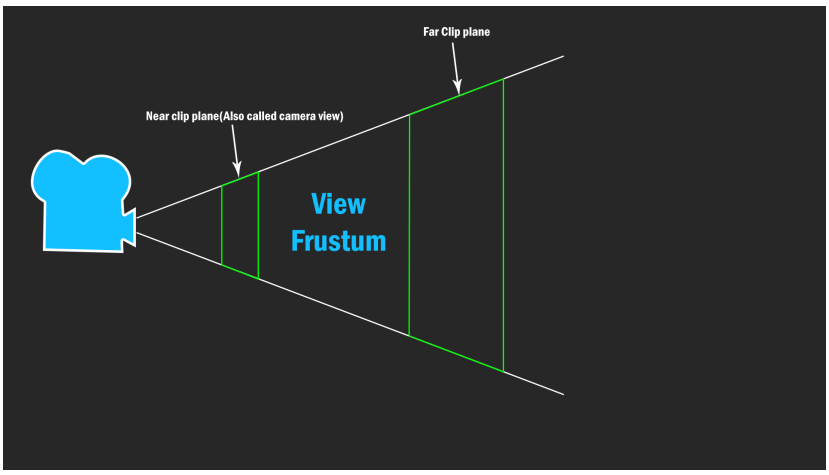


Figure 11: Occlusion

From the image above, 2 white lines are defines as a view angle. Inside the view angle, there's also a clip

plane defined by the green rectangle, which is also the camera view. And at the center of the near clip plane is where the camera is pointing at, and at that point, i've drawn the crosshair, to make sure that this is the camera position. This is the position we the character can interact with the objects, NPCs so as i mention above in the character section, that's why the character cannot be at the middle of the near clip point. The second green rectangle is defined as the far clip plane, which determines how far out into the world the camera view can see. And between the two plane is view frustum, within this shape is what i can see in the game preview.

To detect if the camera cannot see the player or the camera is collided with the objects, all i have to do is just a simple raycasting from the character to the camera, and what is raycasting, i'll explain in the next section.

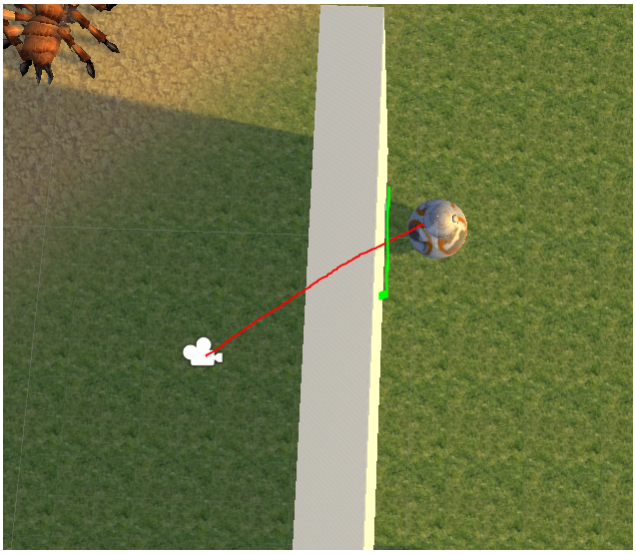


Figure 12: Occlusion

The best raycasting function to use here is Physics.LineCast, as it can return true if the camera hit something, vice versa. This function works in both Collision and Occlusion situation. Like the image above, i've drawn the LineCast between the character and the camera using the function Physics.LineCast, whenever the camera is behind the wall and the player or being collided with the wall, the line can still hit the objects, return true.

```
void CollisionCheck(Vector3 ReturnPoint)
{
        RaycastHit hit;
        if (Physics.Linecast(Target.position, ReturnPoint, out hit, collisionMask))
        {
                normal = hit.normal*wallPush;
                p = hit.point + normal;
                if (Vector3.Distance(Vector3.Lerp(transform.position, p, moveSpeed * Time.
                    deltaTime), Target.position) <= EvenCloserDistance)
                {

                }
                else
                {
                        transform.position = Vector3.Lerp(transform.position, p, moveSpeed *
                            Time.deltaTime);
                }
                return;

        }
        transform.position = Vector3.Lerp(transform.position, ReturnPoint, returnSpeed * Time.
            deltaTime);
}
```

From the code above, the hit function is to return the information where the line hit, collisionMask determine which object layer should the line hit. After the line hit something, i've defined the function hit.normal to find the normal if the line hit the wall, and if it hit, the value will be 1, and the vector p is the position the camera should move. The reason why i do the normalize, add it with the hit.point is just to prevent the camera from clipping with the object when the camera start to move. Finally, to make sure that when occlusion happen, the camera still move to the correct position, be able to see the player, i measure the distance between the camera point and the player, if it's already smaller than a specific value than stop moving the camera, otherwise, it'll lerp to the correct position to see the player.

- **FPSCamera**

This camera represent like how everybody see the world in real life, in everyday life. The controller script for this one is basically the same as the TPScamera script i've written before, but the only difference is the FoV, it's a bit narrower than the FoV of TPScamera and the clamp angle. The camera position is right at the position of the eye's character, for making the gameplay looks more realistic. Moreover, i've written the script to allow the user to change between the FPS mode and TPS mode by pressing V. One more thing, to be able to synchronized 2 camera, all i do is copy almost all the code lines from TPSCamera script and then paste to the FPS one, so that 2 camera will rotate , move extracty at the same angle, same speed.

3.2.4 RayCasting

A Raycast is conceptually like a beam that's fired from a point in space along a particular direction. Any object making contact with the beam can be detected and reported. With the ability to get information from the object that the beam hit, this is the main method for this project, as the character is able to interact with other objects in the virtual world. In Unity, raycast support both 2 dimension and 3 dimension, and for my project, i'll use 3d version for working with 3D space. There're two type of raycast that's usually used , Physics.Raycast and Physics.Linecast.

- **Physics.Raycast(Vector3 origin, Vector3 direction ,float maxDistance = Mathf.Infinity, out hit, , layerMask)**

This type of function casts a ray from the original point, which is Vector3 origin into direction point (Vector3 direction) with the length of maxDistance. Normally, the maxDistance usually set to Mathf.Infinity for casting to the infinite length. This function will return true if the ray intersects with something, otherwise false. The forth variable, the out usually passed by hit, it provides the info about what it hit, like the distance from the origin to the point or the position where it hit, which is extractly what i need for my character to be able to interact, communicate with objects in the scene.

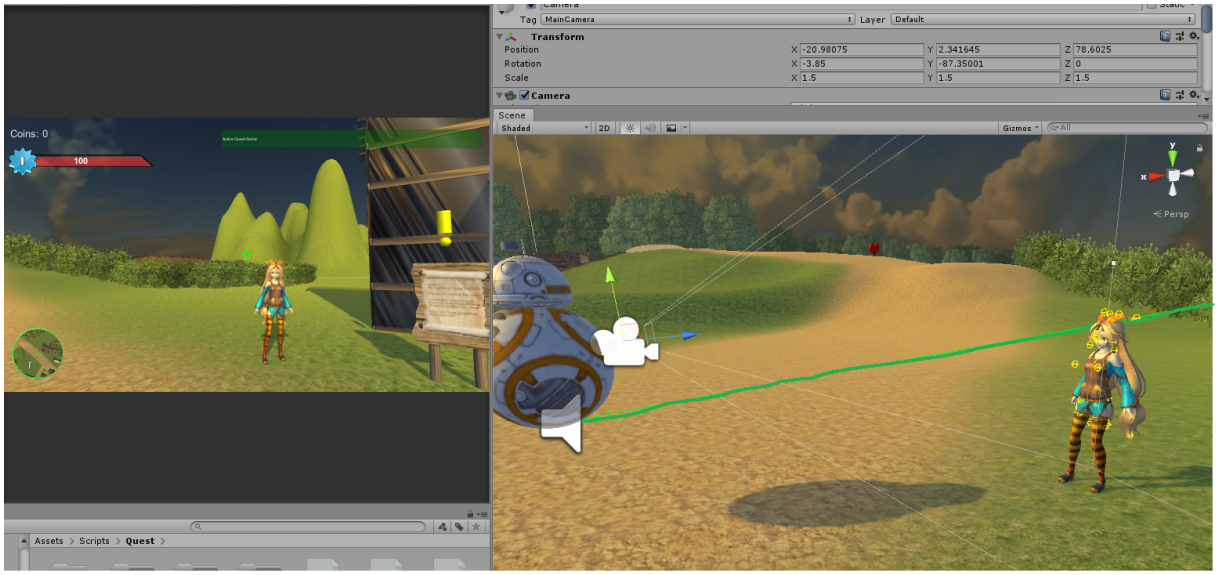


Figure 13: RayCast example

For example, from the figure 13, the greens line is actually the ray drawn from raycast. In the scene above, this ray will provide the distance between my character and the object it hit by hit.distance. In this situation, as the player is far from this NPC, the text which i have defined before will not pop to the screen, but when i move close to the NPC, the text actually pop up: "[E] talk to NPC"

Figure 14: Text popup

This is because i've declared a float variable , when hit.distance bigger than that variable, the text will not appear until hit.distance is smaller. This is the concept of communicating with objects in the scene. Besides, the out variable can be used to get the object name where the ray hit by using hit.collider.gameObject.name: Object: unitychan_dynamic_locomotion

- **Physics.Linecast(Vector3 start, Vector3 end , out hit, , layerMask)**

This function is pretty similar to the raycast function, although they both return true if the ray intersect with a collider, only recognize objects that have collider. The only difference between these two raycasting method is that the raycast only trigger the collider that starts outside of the origin object, and then intersects, while the linecast is not, it will still trigger the collider of the beginning point. That's why i didn't use physics.raycast method for my camera collision detector, as physics.linecast trigger both the collider of two points.

3.2.5 Bump mapping

In order to make the object surface looks more realistic, bump mapping is a technique in computer graphic to do that by simulating small displacements of the surface. However, the number of polygons does not increase. The modified surface heavily relied on light reflection, define how the light should shine on the surface. The method to perform bump mapping i'm going to use here will be divided into 3 stages: height map, normal map and occlusion map, the most important one in this method are height map and normal map. In this project, since i'll only generate bump map from a simple 2D texture, i'll not go too deep into this method, make it as simple as possible. This's the way how unity can perform bump mapping on the object surface.

- **Height map**

To be able to generate normal map, i'll need to create height map from the 2d texture first, to simulate the high of vertex, from that, i can simulate the dept, how the light can shine into the surface. For creating the height map, i'll use Photoshop to make it. The texture for generating the height map is this brick texture, for example:

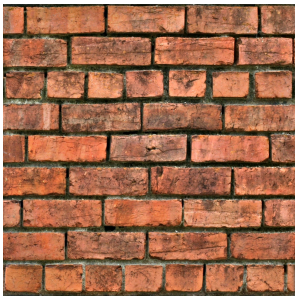


Figure 15: Original texture

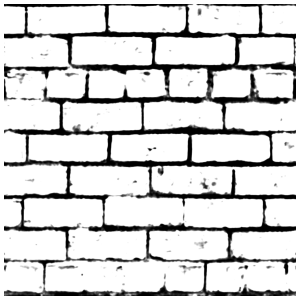
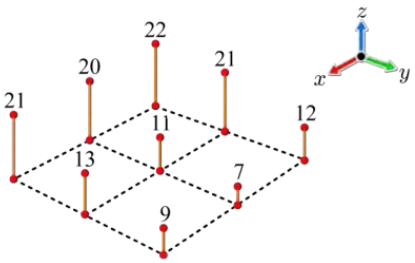
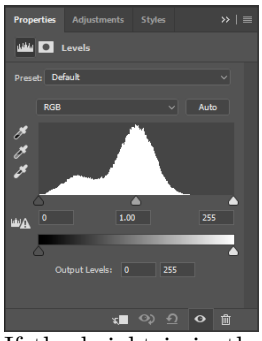
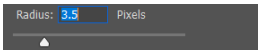


Figure 16: Height map after creation



Basically, the height image is the grey scale image with black and white range, so i'll convert the image to greyscale in Photoshop by selecting Image -> Adjustment -> Black and white. After creating the greyscale image, i'll try to modify the contrast by selecting a small circle at the bottom left, and choose level to adjust the black, white range, adjust the white area to become whiter, dark area to become even darker.



If the height is in the range of [0,255], the lowest height 0 is colored in black, and the highest 255 is colored in white. As of now, there're still noises in the height map, i'll filter the noise by going to Filter -> Blur -> Gaussian Blur: 

Depends on each textures, but for this one, 3.5 pixels is enough to filter the noises, for not losing the detail like in the figure 16, so that i'll get better result when creating the normal map.

- **Normal map**

A normal map is a special kind of texture that allow you to add surface detail such as bumps, grooves, and scratches to a model which catch the light as if they are represented by real geometry, it contains surface normals.

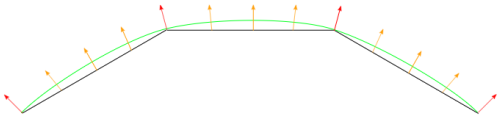


Figure 17: Surface normal, across 3 polygons

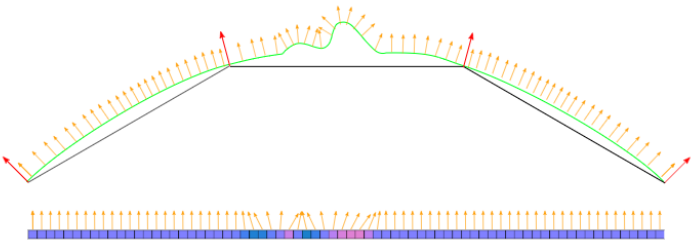
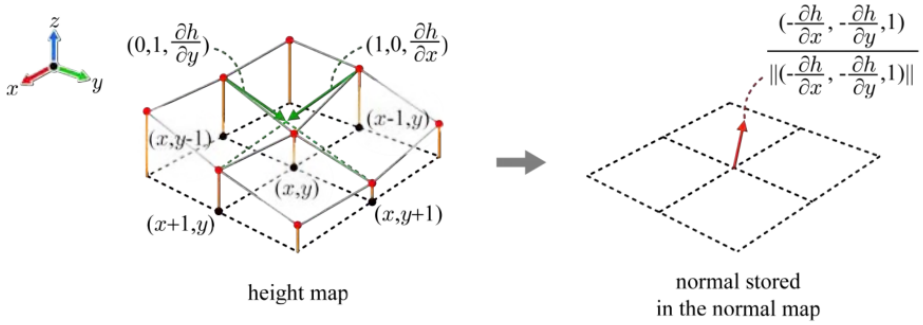


Figure 18: Normal mapping across three polygons

The simplest way is to compute the surface normal from the height map as i created above at the point (x,y,h(x,y)). Here, h represents the discrete height function, and x,y represent as the pixel coordinate.



In here, the partial derivatives are need for computing the normal: $\frac{\partial h}{\partial x}$ and $\frac{\partial h}{\partial y}$. They are defined by the height store at four neighbors of the pixel coordinate: {(x + 1,y),(x −1,y),(x,y + 1),(x,y −1)}.

The equation of these partial derivatives are:

$$\frac{\partial h}{\partial x} = \frac{h(x+1,y) - h(x-1,y)}{2}$$
$$\frac{\partial h}{\partial y} = \frac{h(x,y+1) - h(x,y-1)}{2}$$

After calculating these derivatives, the cross product is $(-\frac{\partial h}{\partial x}, -\frac{\partial h}{\partial y}, 1)$, which is normalized and store at (x,y). In the C# function, this can be implemented as:

```
        private Texture2D GenerateNormal(Texture2D source)
        {
                normalTexture = new Texture2D(source.width, source.height, TextureFormat.ARGB32
                    , true);
                for (int y = 0; y < normalTexture.height; y++)
                {
                        for (int x = 0; x < normalTexture.width; x++)
                        {
                                xHeight = (GetColor((source.GetPixel(x+1, y) - source.GetPixel(
                                    x - 1, y)))*2f-1)*0.5f;
                                yHeight = (GetColor((source.GetPixel(x, y + 1) - source.
                                    GetPixel(x, y - 1)))*2f-1)*0.5f;
                                normalTexture.SetPixel(x, y, new Color(-xHeight, -yHeight, 1.0f
                                    , 1.0f));

                        }
                }
                normalTexture.Apply();
                System.IO.File.WriteAllBytes("Assets/Sample001_n.png", normalTexture.
                    EncodeToPNG());
                return normalTexture;
        }
```