**Interactive Programming Game Design**

An Unity3D Coding Game Report

11711613 王天麒 11712208 姜朋坤

**ORIGINAL OBJECTIVE**

In this experiment, we want to build a programming game base on Unity3D. As for a player, you can program the activity for all of the object which is programmable, and can pick the main character which can be handled with keyboard and mouse of yourself. We will design some new API to reduce the difficulty of programming. By doing this project, we will try to find out how to build and design a programming game and gather the experience of software development.

**WORK DESCRIPTION**

The main model of other programming game is a study platform of learning programming, or let programming be the core of this game. For the first kind of game, it will let the player know the basic programming logic and learn a new programming language with the help of the game design. It focus more on education but can not let most of the players feel fun. The second kind of game is give some puzzle to let the player solve them by programming. This type of game will be fun but may just has a little number of player.

What we want build is a open-world programming game. In this game, you can design the behavior of all object in this game by programming. We will first study the design structure of Robocode,[1] an open source programming game, and reimplement them on Unity. Then we will add our own originality into it. To reduce the modeling problem, we will use cube to represent the object in this game, which may let our scene has the similarity of game Minecraft.[2]

**CURRENT PROGRESS**

The first step of our program is to implement the interface of object’s sight and movement. Then we will find a method to add the new activity logic written by the player into the object. We first design our class diagram like below. And we implement the Sight part and Movement part.

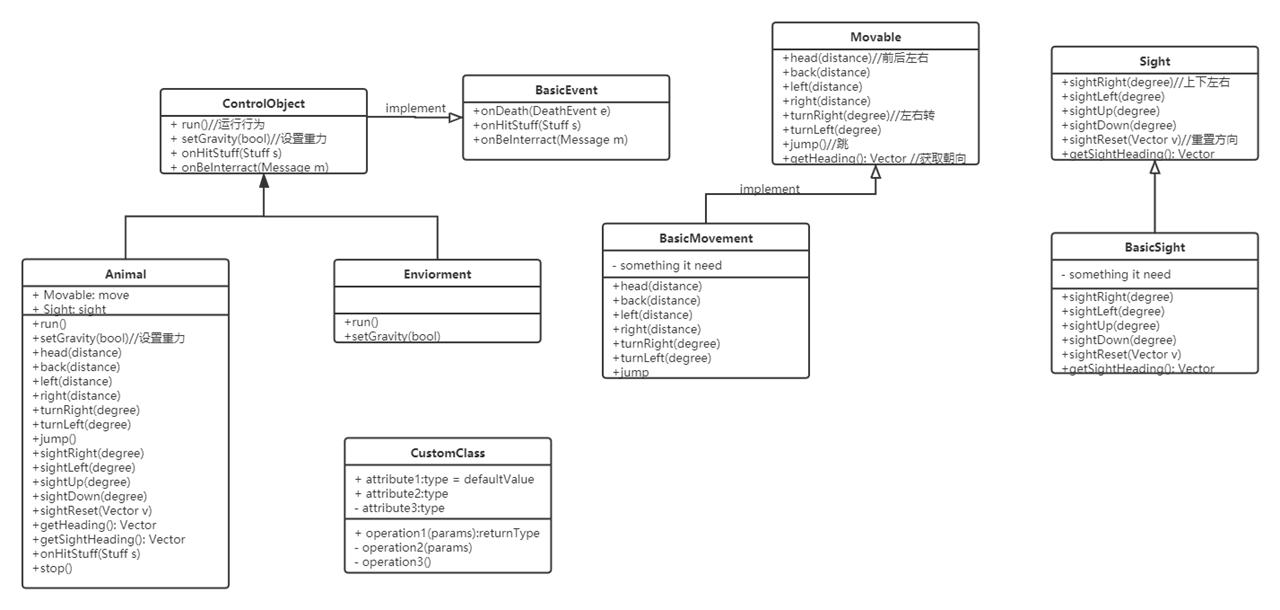


Figure 1

**Sight part**

To implement the sight part, we use some ray to simulate the sight of the object. Each object which has sight will get an extra part called eyes, and eyes will give out some ray in a limited angle (like between left 60 degree and right 60 degree). If the ray get touch with other object, sight will return some information of this touched object( can not return all information of touched object because the value that player can handle has some limitation). When player need to change the direction of the sight, we will change the angle of the eyes object. By doing this, the direction of those rays will change with the eyes.

**Movement part**

To implement the movement part, we use a asset package called iTween provided by the asset store.[3] For those movement, we will first calculate the position of the destination by the distance and angle. Then use function iTween.MoveTo() to go to that position automatically. The powerful iTween provide a series of arguments to show the animation of the movement including speed, loop, time, face direction and so on. By fill the arguments, we can get the movement we need.

To detect the ground, we use a ray and try to touch the ground. If the ray has collision with the ground, we can get the distance to the ground by the length of this ray. Using this way we can detect the ground and do some special dealing.

**Player programming part**

To let the player has the similar experience when programming, we need to remove the influence of the Update() function in unity, which will do all code in this function each frame. Which means if we write a loop in the Update() function, we will execute all operation immediately. And we choose coroutine to fix this situation. In the coroutine, we can use yield return to return a IEnumerator to the main thread, and the coroutine will stop here and wait for the return condition. When reach the return condition, it will continue from last stop position. By using coroutine, we can use loop in it and add some wait instruction to let the animation finish before the next animation start amd without stop the whole thread.

Now we have a function Task(), player can program in this part with the interface we provided. And we provided another function ScanOn(), when the sight ray touch some object(which means the object see something), an event will be called, then ScanOn() function will be called. So the player only need to overwrite 2 function to create the activity logic.

**Dynamic link code to game object**

At first, we output player written code to a C# script and then connect it with the game object. But when we output the whole project to a real game program, the script we write all become .dll files. And we can not add a connection between a .cs script and our running program.

Then we realize that the .dll files is used to dynamic link to a program. So if we can compile our player written script to a .dll file, then we can use those code in our program. C# has its own dynamic compile library called Microsoft.CSharp.dll, and we use this library to make our simple dynamic compiler. To make our .dll file run in our Unity program successfully, we must use same library with our Unity program. We need UnityEngine.dll to provide the Unity environment and Assembly-CSharp.dll (all script we write in Unity will be compiled into this file) to let the compiler can use our API.

We try to use this way directly in our Unity program, but we failed. And that may because that Unity does not support all functions in Microsoft.CSharp.dll. So we use another way: write a independent compiler and let two program communicate by socket.[4] We combine the server and the compiler together and put it into the Managed folder of our program which contain all dll file this program will use. When the player finish his/her writing, we send the string player write to the server, server compile it and output the library file in a special name. Then we use this file when we need add the logic to our game object. The result is that it really works.

The structure of our dynamic link part is as this picture:

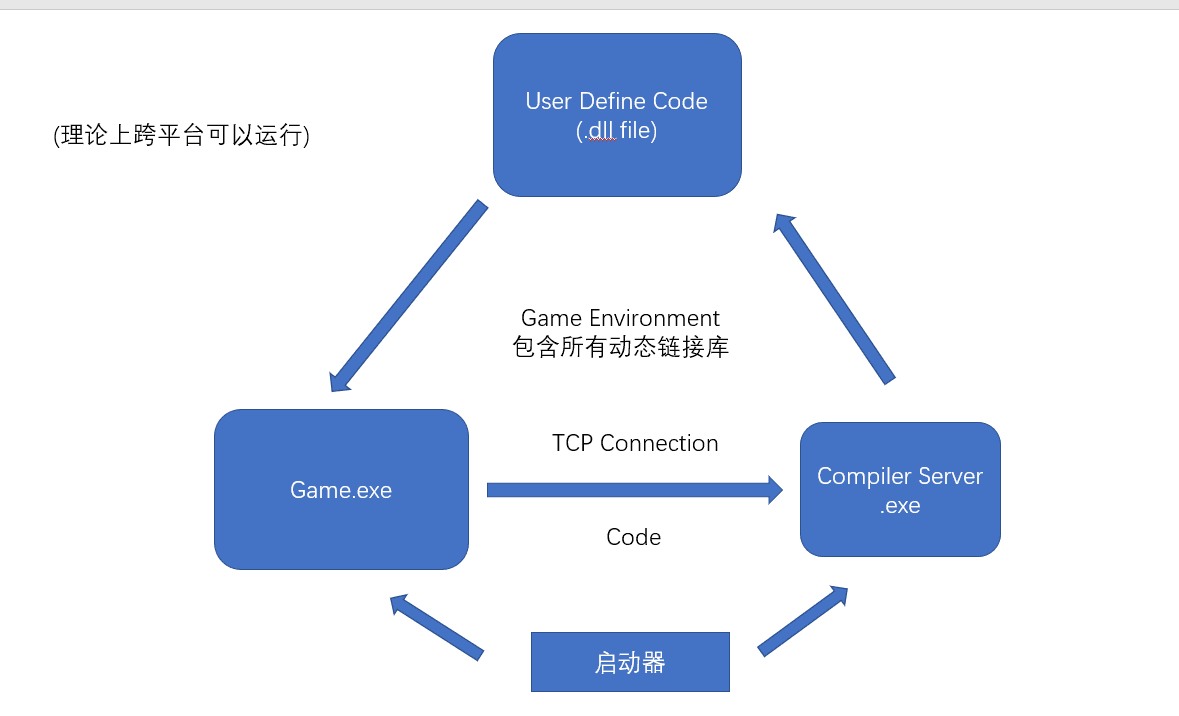


Figure 2

**World Generation**

In order to produce the effect of the infinite simulation world, we need to create terrain and match player’s position.

For this proposal we choose a noise to generate height of terrain, to ensure consistency and smoothness, Perlin noise is chosen[5]. Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. The function has a pseudo-random appearance, yet all of its visual details are the same size.

It means the noise result is the same when P(x,y) and noise seed are determined( pseudo-randomness).

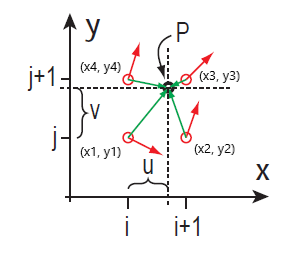


Figure 3

We have P(x, y) as input, and as the output we get a value between 0.0 and 1.0. First, we divide the x, y coordinates into unit cubes to find the coordinate’s location within the square(red dot in figure3). On each 4 unit coordinates, we generate what’s called a pseudorandom gradient vector(red vector in figure 3). This gradient defines a positive direction (in the direction that it points to) and of course a negative direction (in the direction opposite that it points to). Pseudorandom means that, for any set of integers inputted into the gradient vector equation, the same result will always come out. Thus, it seems random, but it isn't in reality. Additionally this means that each integral coordinate has its "own" gradient that will never change if the gradient function doesn't change. Next, we need to calculate the 4 vectors from the given point to the 4 surrounding points on the grid(green vector in figure 3). Next, we take the dot product between the two vectors (the gradient vector and the distance vector). So now all we need to do is interpolate between these 4 values so that we get a sort of weighted average in between the 4 grid points.



Figure 4

Figure 4 is the terrain generated by Perlin noise in Unity, in this case we make height as noise result product max height.

**Refinement**

**Generation Area**

In general game, player always has a view distance which means if the scenery are out of indicate distance, they are invisible. So we can only generate blocks within range.

**Face rendering**

Drawing all blocks in game is inefficiency and resource-intensive. But it is not necessary to draw blocks which is invisible to player. So we just need to draw the faces which player can see.

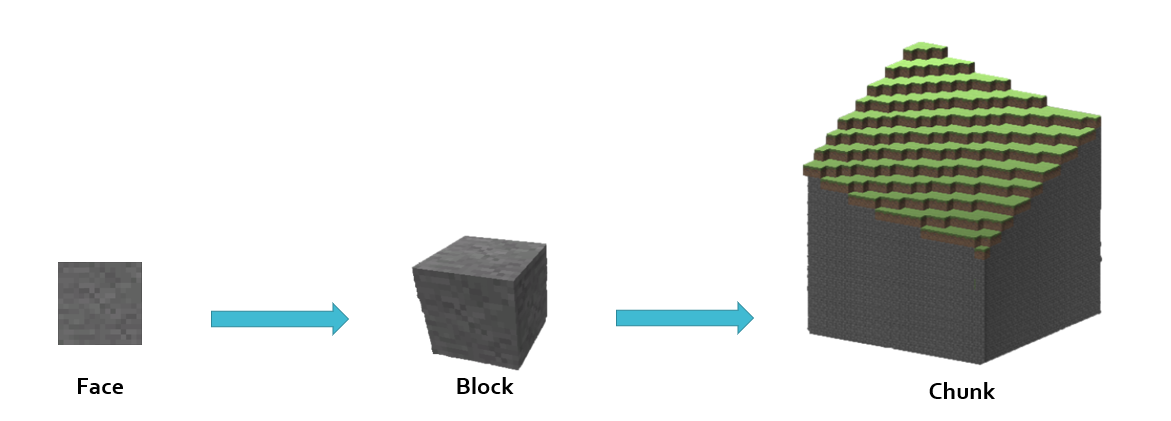


Figure 5

As the figure 5 shows that, 6 faces form a block, many blocks form a chunk, and finally many chunks form the world. We only draw the outer face of chunks.

**PROBLEMS AND DIFFICULTIES**

1. We only has one kind of event when see an object. But we need an AppearEvent and a DisappearEvent to handle the condition when the object appear and disappear in one’s sight. We need to fix those two condition to complete our movement and sight interface.

2.When the object drop out of the cliff because of the horizontal movement, it will go to the place with the same height as the origin position, then drop down. If we change the destination to the ground and the object does not on the edge of this cliff, it will go into the cliff. So we need a new movement for this type of activity.

3.

**WORK DIVISION**

11711613 王天麒：world generation, sight interface design and implementation

11712208 姜朋坤：movement part, dynamic link part

**FUTURE WORK**

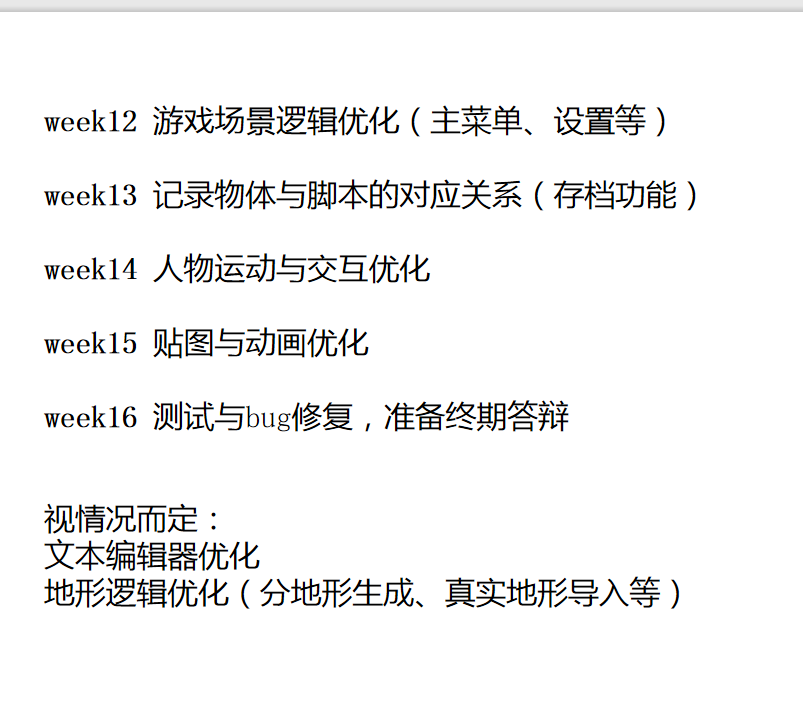
1. fix the bug and problem we have already found, and do more test to find bug we have not found.

2. finish saving part.

3. upgrade the interaction part.

4. a better UI for programming.

**Future Timeline**



**Reference**

[1] Robocode, code, <http://robowiki.net/wiki/>

[2] Minecraft, game, https://my.minecraft.net/

[3] iTween,code, <http://itween.pixelplacement.com/>

[4] Unity动态脚本挂载，zxy2847225301，article， https://blog.csdn.net/zxy13826134783/article/details/80983542

<https://blog.csdn.net/zxy13826134783/article/details/80987081>

[5] Ken Perlin's SIGGRAPH 2002 paper: Improving Noise