摘要

在二十一世纪的现在,据第一个电子游戏的出现已经过了60年以上，在这期间随着科技的迅速发展，电子游戏的技术也得到了迅速的发展，游戏的内容和类型开始变得多样。游戏的自由度越高时游戏类型就越接近类似沙盒游戏的游戏类型。沙盒游戏能够包含格斗、射击、驾驶、养成以及冒险等多种不同的游戏要素。沙盒游戏的高自由度甚至可以创造或者毁灭整个游戏世界。目前个人认为最成功的沙盒游戏分别是 3D 的《我的世界》和 2D 的《泰拉瑞亚》，二者凭借广阔的世界，丰富的游戏因素和极高的自由度吸引了大批玩家来到沙盒世界建造属于自己的世界。

本论文是基于 Unity3D 探索横版的 3D 沙盒游戏的开发与实现,既实现 2.5D 的沙盒游戏。游戏的设计方向是实现一般沙盒游戏拥有的游戏元素，着重在于沙盒世界的地图的随机生成，横版下的建造模块的探索。

关键词：Unity3D 沙盒游戏 2.5D 游戏 随机地图

Abstract

In the 21st century, more than 60 years have passed since the first video game appeared. During this period, with the rapid development of technology, the technology of video games has also been rapidly developed, and the content and genre of the game have begun. Become diverse. The higher the degree of freedom of the game, the closer the game type is to a game type similar to a sandbox game. Sandbox games can contain many different game elements such as fighting, shooting, driving, training and adventure. The high degree of freedom of sandbox games can even create or destroy the entire game world. At present, I personally think that the most successful sandbox games are 3D "My World" and 2D "Terraria". The two have attracted a large number of players with their vast world, rich game factors and extremely high degrees of freedom. Go to the sandbox world to build your own world.

This paper is based on the development and implementation of the 3D sandbox game of Unity3D exploration horizontal version, which is the realization of 2.5D sandbox game. The design direction of the game is to realize the game elements owned by general sandbox games, focusing on the automatic generation of maps in the sandbox world and the exploration of construction modules under the plane.

Key words： Unity3D Sandbox game 2.5D game Random map

目录

1 绪论

1.1 开发背景及意义

科技的日新月异，带动了游戏相关产业的技术升级，动作、冒险、休闲以及新出的沙盒和 VR 等各种各样的游戏类型迎来了进一步的发展。而在人们背负着众多压力的生活中，游戏无疑成了背负众多压力的人的一种暂时释放压力与增添生活趣味的途径之一。之所以选择沙盒游戏,是沙盒游戏足够大的世界可以保证不同的游戏元素可以实现,而且现在热门的 3D 沙盒游戏是 Minecraft 而 2D 沙盒游戏是 Terraria,这些游戏里不仅有冒险,丰富的道具,怪物以及一些电路之类的系统,大大增加了游戏的可玩性。3D 游戏比 2D 游戏在模型、代入感等上面有不可比拟的优势，但在游戏的休闲性和趣味性上我更偏向 terraria，横版的画面能让玩家清楚的知道周围的情况，能减轻紧张的情绪，而且特效效果能变得直观。

3D 横版沙盒游戏以冒险，休闲，经营为主。选择用 unity3d 做 3D 横版沙盒游戏是想要将 2D 游戏 3D 化同时保留 2D 游戏的特色。在游戏中不设置生命数量，死亡不再是游戏的终点，玩家里可以按自己现实生活中的节奏来选择游戏的退出和继续。游戏以探索未知为主并且没有日常式任务，不需要长时间进行游玩，可以随时随地保存并退出游戏。本次课题的主要的目标是为了探索 3D 横版沙盒游戏的开发提供基础理论经验,将以代码编程为主,创造一个基础的横版沙盒游戏世界。

1.2 本文内容安排

第一章说明本次课题的意义，探讨项目的开发背景以及未来前景，以及项目内容的相关介绍。

第二章简要说明项目运行的环境配置、所使用工具。

第三章说明沙盒游戏的市场前景以及项目中的游戏大致玩法介绍

第四章着重于项目中每一个功能的实现原理,代码编写以及功能的详细说明。

第五章总结项目制作过程中的收获。

1.3 课题的研究内容与重点

本课题的研究是开发一个基于 unity3D 的横版沙盒游戏，利用 C#编写包括随机的地图生成，敌人角色生成，探索，战斗，建造及保存玩家在游戏内活动的内容等相关游戏内容的实现。

课题的重点在于沙盒游戏里可编辑性地图的研究，横版玩法下角色的控制方法，物体放置的实现，以及最终的 UI 界面设计。

2 游戏开发原理

2.1 游戏引擎介绍

本次使用的游戏引擎是 Unity3D 个人版，一款非常适合新手初次使用并且免费的游戏引擎，Unity3D 对个人开发者来说是有着低廉的成本，而且相关的代码编写及操作简单，直观，容易入门。编程语言虽然支持 C#、javascript、boo 三种脚本语言，但由于官方的编程语言是 C#，建议直接使用 C#即可。同时还自带官方的 AssetStore，里面提供了大量用于开发的插件和资源，其中有不少官方提供的免费素材及插件。在界面编辑点击书本符号会跳转到当前插件的的 Unity3D 手册，Unity3D 手册提供了几乎每一个插件的介绍，参数介绍，使用条件。在 Unity3D 里进行工作的时候，工作项目的调试也非常简单，保存好编写的内容就可以在视窗界面点击 Play 进行运行，并且 Unity3D 导出的游戏能应对 Windows、Mac OS、iOS、Android、Linux 等多种平台进行兼容。

2.2 C#介绍

C#是微软公司发布的一种运行于.NET Framework 上的计算机编程语言。C#是 Unity3D 的官方编程语言，C#本身不是一种跨平台的编程语言，但由于 Mono 的桥接使得在 Unity3D 上可以使用 C#这种语法明了，类库使用简单的语言。

2.3 系统开发运行环境

系统环境

操作系统：Windows10 家庭中文版（64 位）

开发工具：Unity3D（Unity 2018.3.8f1 Personal）

处理器：Intel(R)Core(TM) i7-6700HQ CPU@2.60GHz 2.59GHz

C#编辑器：VisualStudio2017

3 游戏需求分析

3.1 游戏基本介绍

开始游戏时，在 UI 动画结束后，玩家出现在一片废弃区域的传送通道，玩家可以自由改造初始的废弃区域，玩家可以通过传送通道去往其它随机生成的地图上，应付当地的防御机制获取在初始区域建造物品的资源。

游戏中没有真实意义上的游戏结束，在玩家的生命归零后会播放玩家倒地动画，在初始地图复活，所有的随机地图在探索结束离开后都会消失。

3.2 需求分析

当今快节奏的社会中，大多数人的吃穿住行等最基础的物质上的需求基本得到了满足，人们开始更多的寻求精神上的需求。书籍、游戏、电影和动漫这类的精神上的需求得到了大多数人的欢迎。游戏更是被人们广泛接受的文化之一。在如今快节奏的生活中，人们或许在忙碌之余想静静的放松并暂时忘记现实世界的烦恼的时候，沙盒游戏是不错的选择。在沙盒游戏中，游戏世界会基于一定的模板生成一个相对随机的世界，来到这个游戏世界的玩家拥有最大自由度去选择如何在这个虚拟的世界生活，并且可以随时保存并结束游戏投身到现实生活中去。

4 游戏设计与实现

4.1 UI 界面

游戏的菜单界面以简洁明了为主，基本上采用了黑白设计并添加动画控制消失和出现以便于在观感上变得平滑，如图 1 左右两边的形状分别对应着退出游戏和重新开始游戏，中间的齿轮可以触发菜单的关闭动画，下方的秒表图案用来保存游戏，左下方音量标志控制游戏声音的有无，另一边的是游戏的操作说明。

图 4-2 是开始游玩时的界面，红色的机器人图标代表 Player 的生命值，蓝色的代表能量，蓝齿轮表示目前已有的资源数量。右下方的齿轮组会在建造模式下从下方移动到当前位置，每个齿轮代表可以制造的物体，齿轮的右下角代表当前可造数量。

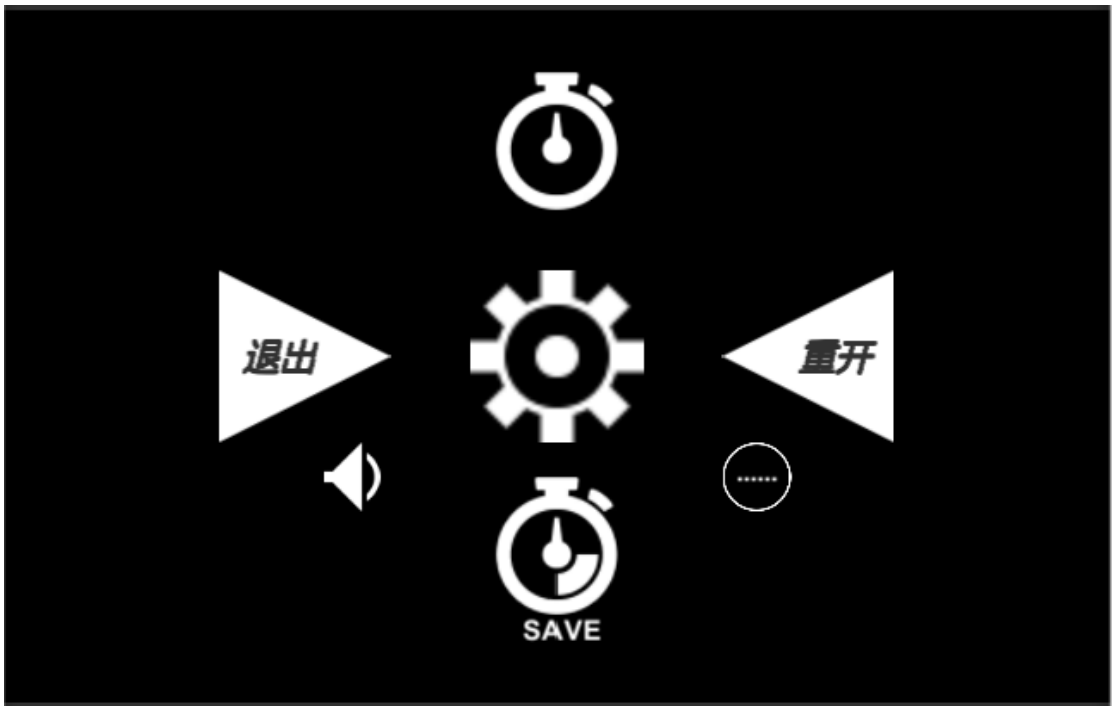

图 1 菜单界面

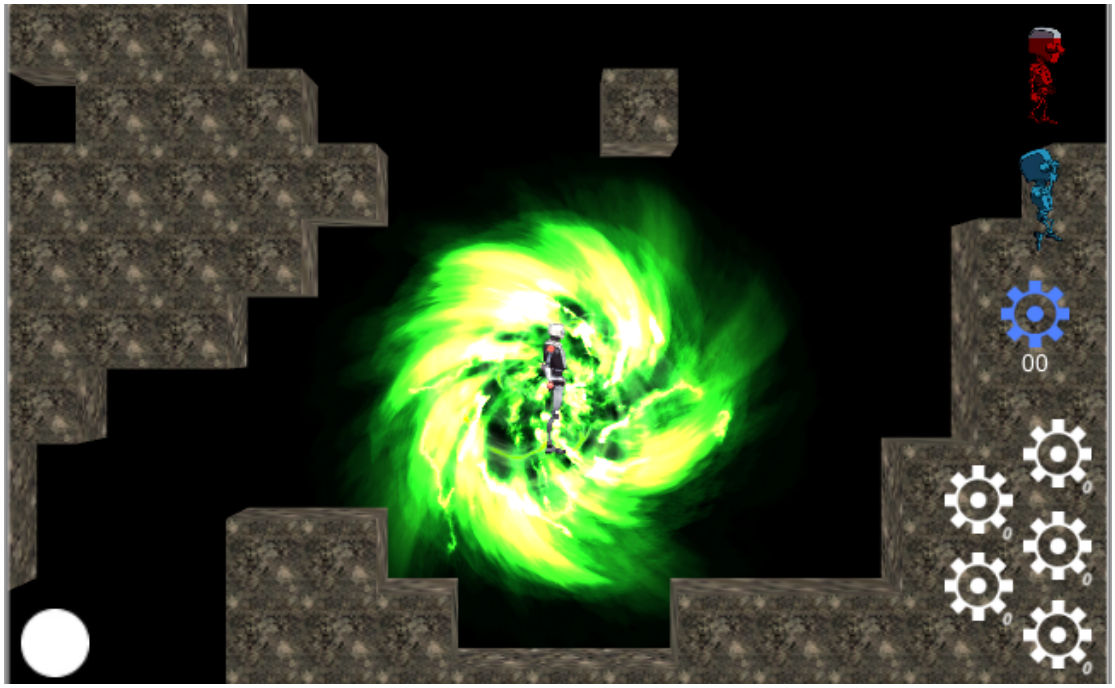

图 2 游戏界面

4.1.1 UI 介绍

Unity3D 的 UI 制作是非常方便的，在窗口 Hierarchy 空白处右键→UI→Panel 即可在 Hierarchy 窗口自动创建 Canvas 和 EventSystem 并在 Canvas 里生成 Panel，EventSystem 在 Hierarchy 窗口里基本只能有一个，创建的 UI 基本都是 Canvas 的子节点。UI 的基本类型有 image、Text、Button、Panel、Solid 等多种类型。

4.1.2 主界面 UI 制作

主界面 UI 主的制作主要使用了 Unity3D 自带的 Button 组件，Image 组件和 Animator 组件来制作。完成图 1 菜单画面的基础布置后，可以考虑加入 Animator 制作一些简单的 UI 动画。在 Canvas 里将相关的 UI 放到同一个父节点“Closing”下面，之后在父节点的组件里通过 Add Component 添加 Animator 组件。之后再根据自己的创意制作动画。全程除了 Button 的 On Click()事件要写些相关代码外都可以在编辑界面完成。

Button 按钮相关代码

```
using UnityEngine.SceneManagement;
public void RestScene1()
    {
        //当前场景重新加载
```

```
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void GameQuit()
    {
        //退出游戏
        Application.Quit();
    }
    public void ClosingEnd()
    {
        //关闭 UI
        if (!Player_control.player_Control.isDie)
        {
            anim.SetBool(ClosingStarid, false);
            anim.SetBool(ClosingEndid, true);
        }
    }
```

主要是通过将初始的保存地图的二维数组清空，遍历每一个初始场景的物体创建时的信息再保存。

```
    public void MainMapSave()
    {
        //保存初始地图
        MainMapGrid = new int[mapsize, mapsize];
        if (gamemap[5].childCount != 0)
        {
            for (int i = 0; i < gamemap[5].childCount; i++)
            {
                int _x =
gamemap[5].GetChild(i).GetComponent<BlockMessage>().x;
                int _y =
gamemap[5].GetChild(i).GetComponent<BlockMessage>().y;
                MainMapGrid[_x, _y] = 1;
            }
            for (int i = 0; i < gamemap[7].childCount; i++)
```

```
                {
                    int _x =
gamemap[7].GetChild(i).GetComponent<BlockMessage>().x;
                    int _y =
gamemap[7].GetChild(i).GetComponent<BlockMessage>().y;
                    MainMapGrid[_x, _y] = 5;
                }
            }
            Debug.Log("主地图保存");
        }
```

4.1.3 血条&蓝条

图 3 中显示的机器人图像的红蓝 UI 分别是生命值和能量值主要是应用 UI 组件 Image 完成。组件 Image 中参数设置 ImageType→Filled，FillMethod→Vertical 之后就可以通过调节 FillAmount 的值实现血蓝条的增减，如图 3。FillAmount 的值最小为 0 最大为 1，可以通过将当前的生命值除去最大的生命值来获得 FillAmount 的值。比如 FillAmount = 角色当前的数值/角色最大生命值。

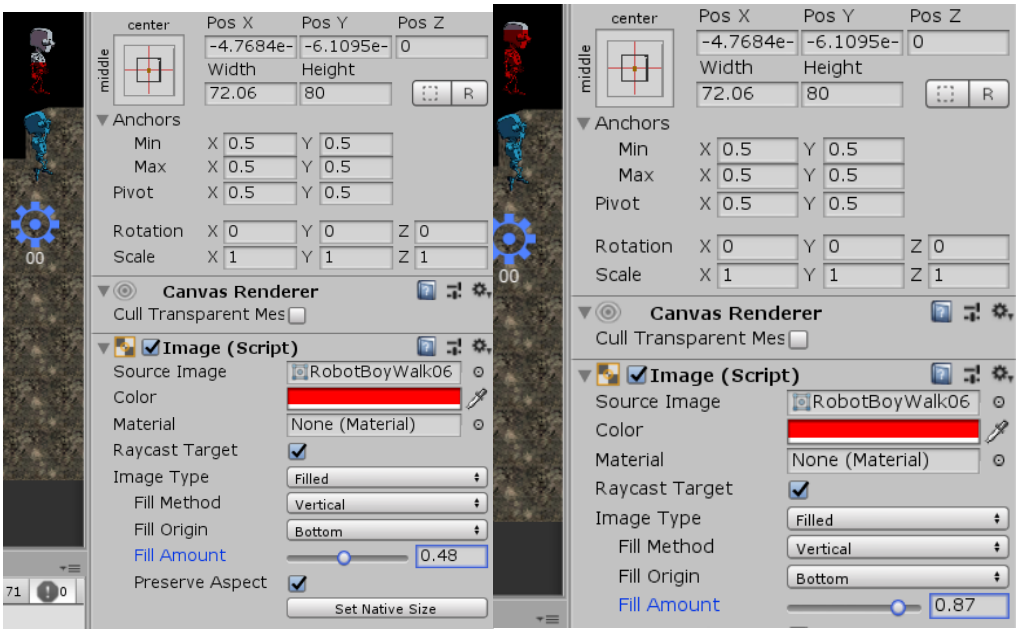

图 3

4.1.4 齿轮 UI

蓝色齿轮表示游戏中玩家获取的资源量，主要是通过击碎水晶来增加。

资源数量是通过 Text 组件里的 Text 来显示，代码上通过获取 Text 组件“BaseCount”之后再将 int 类型的资源数量“Counter”转换成类型 string，代码如下 BaseCount.text = GM.gM.Counter.ToString("0");

白色齿轮用来直观表示当前选择要建造的物体和能够建造的数量。

4.2 地图

4.2.1 地图生成

地图采用二维数组来实现，先通过 Random.Range(int min,int max)随机生成一个 40*40 的由 0 和 1 组成的二维数组 int[,]，之后再通过填补和删除逐渐得到想要的效果，之后再根据二维数组的值生成物体，并将每个物体放进物体数组 GameObject[,]，这样基本可以实现基本的物体删除和创建，在地图数组生成后经过填补，删除完成之后遍历数组查找能够添加水晶的位置接着按照二维数组创建游戏地图。如初始地图 4。图副本随机地图 5。二维地图生成代码如下

```
using UnityEngine;
public class CreateTerrain : MonoBehaviour
{
    public static CreateTerrain createTerrain;
    public GameObject rock;
    //保存生成的两种水晶类型
    public GameObject[] crystalsv;
    //生成地图位置
    public Transform[] gamemap;
    //传送通道
    public GameObject JumpTuunnel;
    //数组大小
    public int mapsize = 40;
    //副本地图二维数组
    public int[,] mapgrid;
    //主地图数组
    public int[,] MainMapGrid;
    //副本地图生成的物体池
    public GameObject[,] objgrid;
    //主地图生成的物体池
```

```
public GameObject[,] Mainobjgrid;
void Update()
{
    //判断是否需要调用主地图
    CreateMainMap();
```

检测随机副本的数组是否被清空，如果被清空则会开始销毁随机副本，并生成新的随机副本地图，一般在玩家离开随机地图后调用。

```
        if (mapgrid == null)
        {
            if (gamemap[0].childCount != 0)
            {
                for (int i = 0; i < gamemap[0].childCount; i++)
                {
                    Destroy(gamemap[0].GetChild(0));
                }
            }
            CreateMap(gamemap[0],gamemap[1]);
        }
    }
    //创建地图
```

CreateMap（）主要用来生成随机地图，两个参数分别代表生成的岩石和水晶的父节点。该函数调用的时候会检测随机副本地图是否是空数组，不是则直接调用生成地图函数 CreateMap_execute（），是的情况下则会重新生成新的随机副本地图数组，经过多次 ResetMap_del()和 ResetMap_fill()的删补后 ResetMap_delone()消除孤立的个体 CreateCrystalsv()则会寻找数组地图上满足生成水晶条件的位置并概率生成，Createtunnel()负责在数组地图基本成型的时候封边之前找出传送通道的位置，之后执行生成函数 CreateMap_execute（）。

二维数组地图生成函数 CreateMapGrid(int[,] mapgrid)，该函数通过 for 循环和随机数生成大致的二维数组模型，之后通过函数 RestMap_del()删除,函数 RestMap_fill()填充,函数 RestMap_delone 删除孤立点，完成基本框架，之后遍历数组添加水晶敌人和传送通道位置。

```
    void CreateMapGrid(int[,] mapgrid)
    {
```

```
for (int i = 0; i < mapsize; i++)
{
	for (int j = 0; j < mapsize; j++)
	{
		if (Random.Range(1, 100) >= 40)
		{
			mapgrid[i, j] = 1;
		}
		else
		{
			mapgrid[i, j] = 0;
		}
	}
}
//优化地图
ResetMap_del(mapgrid);
ResetMap_fill(mapgrid);
ResetMap_del(mapgrid);
ResetMap_delone(mapgrid);
//生成水晶
CreateCrystalsv(mapgrid);
//生成传送门
Createtunnel(mapgrid);
//封边
for (int i = 0; i < mapsize; i++)
{
	if (mapgrid[0, i] == 0)
	{
		mapgrid[0, i] = 1;
	}
	if (mapgrid[i, 0] == 0)
	{
		mapgrid[i, 0] = 1;
```

```
                }
                if (mapgrid[i,mapsize - 1] == 0)
                {
                    mapgrid[i, mapsize - 1] = 1;
                }
                if (mapgrid[mapsize - 1,i] == 0)
                {
                    mapgrid[mapsize - 1, i] = 1;
                }
            }
        }
```

传送通道的位置生成是通过不断随机出地图数组中一个当前位子没有其它物体的坐标（x，y）符合条件后开始判断在数组中(x-2,y)+(x+2,y)+(x,y-2)+(x,y+2)值的和为零的情况下生成传送通道并将坐标(x,y)为中心的 3*3 的九宫格，除坐标外其他坐标的值变更为-1，变成不可使用区域，不等于零的情况下在开始下一轮的随机，直到找到符合条件的位置循环结束。

```
//挑选位置生成传送门
void Createtunnel()
{
    int _x;
    int _y;
    for (int i = 0; i < i + 1; i++)
    {
        _x = Random.Range(2, 38);
        _y = Random.Range(2, 38);
        if (mapgrid[_x, _y] == 0)
        {
            int _sum = mapgrid[_x - 2, _y] + mapgrid[_x + 2, _y]
+ mapgrid[_x, _y - 2] + mapgrid[_x, _y + 2];
            if (_sum == 0)
            {
                for (int k = _x - 2; k < _x + 3; k++)
                {
```

```
                        for (int l = _y - 2; l < _y + 3; l++)
                        {
                            mapgrid[k, l] = -1;
                        }
                    }
                    mapgrid[_x, _y] = 4;
                    break;
                }
            }
        }
    }
    //生成主地图
```

函数 CreateMainMap()主要负责在场景里按照主地图的二维数组创建对应的主地图，由于主地图和副本地图同样在同一个 Scene 里，在生成完物体后变动主地图父节点位置以防和副本地图位置重合。

```
    public void CreateMainMap()
    {
        //如果未生成主地图
        if (gamemap[5].childCount == 0)
        {
            //判断主地图数组是否为空
            if (MainMapGrid.Length != 0)
            {
                //生成地图
                CreateMap_execute(MainMapGrid, Mainobjgrid,
gamemap[5],gamemap[6]);
                gamemap[5].position = new Vector3(0, 70, 0);
                gamemap[6].position = new Vector3(0, 70, 0);
                gamemap[7].position = new Vector3(0, 70, 0);
            }
        }
    }
    //地形生成
```

CreateMap_execute()负责通过数组遍历数组里的值在场景里生成数组地

形。这里是后期要重点改进的地方，如果游戏中的可生成物体越多这里编写的代码便会越越多越显的臃肿，不利于后期的代码维护。

```
for (int i = 0; i < mapsize; i++)
{
    for (int j = 0; j < mapsize; j++)
    {
        switch (_mapgrid[i, j])
        {
            //地形框架
            case 1:
                _objgrid[i, j] = Instantiate(rock, new Vector3(i, j, 0), Quaternion.identity, _rocktr);
                _objgrid[i, j].GetComponent<BlockMessage>().x = i;
                _objgrid[i, j].GetComponent<BlockMessage>().y = j;
                break;
            //攻击型水晶
            case 2:
                //向右
                if (_mapgrid[i + 1, j] == 1)
                {
                    _objgrid[i, j] = Instantiate(crystalsv[0], new Vector3(i, j, 0), Quaternion.Euler(0f, 0f, 90f), _crytr);
                    _objgrid[i, j].GetComponent<CryNews>().cryX = i;
                    _objgrid[i, j].GetComponent<CryNews>().cryY = j;
                    continue;
                }
                //向左
                if (_mapgrid[i - 1, j] == 1)
                {
                    _objgrid[i, j] = Instantiate(crystalsv[0],
```

```
    new Vector3(i, j, 0), Quaternion.Euler(0f, 0f, -90f), _crytr);
                                        _objgrid[i,
j].GetComponent<CryNews>().cryX = i;
                                        _objgrid[i,
j].GetComponent<CryNews>().cryY = j;
                                        continue;
                                    }
                                    //向上
                                    if (_mapgrid[i, j + 1] == 1)
                                    {
                                        _objgrid[i, j] = Instantiate(crystalsv[0],
new Vector3(i, j, 0), Quaternion.Euler(0f,0f,180f), _crytr);
                                        _objgrid[i,
j].GetComponent<CryNews>().cryX = i;
                                        _objgrid[i,
j].GetComponent<CryNews>().cryY = j;
                                        continue;
                                    }
                                    //向下
                                    if (_mapgrid[i, j - 1] == 1)
                                    {
                                        _objgrid[i, j] = Instantiate(crystalsv[0],
new Vector3(i, j, 0), Quaternion.Euler(0f, 0f, 0f), _crytr);
                                        _objgrid[i,
j].GetComponent<CryNews>().cryX = i;
                                        _objgrid[i,
j].GetComponent<CryNews>().cryY = j;
                                        continue;
                                    }
                                    break;
                                ……
                                ……
                                ……
                            default:
```

```
                    break;
                }
            }
        }
```

case 2:这里的四个 if 语句是利用数组来判断生成的水晶方向。

数组的删补操作基本都要遍历数组，找到每个数组成员的(x，y)值获得上下左右的值，判断是否满足条件。比如这里有一个 5*5 的二维数组

Y+2	1	1	0	1	0
y+1	0	0	0	1	0
y	1	1	1	0	1
y-1	0	0	0	1	1
y-2	0	0	1	1	0
y/x	x-2	x-1	x	x+1	x+2

以坐标为（x,y）的点来算，点（x,y）的周围的 8 个点合计值为 3，小于 5 之后（x,y）的值会从 1 变为 0。水晶的生成则是判断点(x,y)的值是否为 1，之后遍历上下左右，有值为 0 的情况下就有几率将 0 替换成 3 或 4。

```
    //填补
    private void ResetMap_fill()
    {
        for (int i = 0; i < mapsize; i++)
        {
            for (int j = 0; j < mapsize; j++)
            {
                //如果空物体位置周围的方块数大于 5 则填补,物块周围方块数在 4 以内则消去
                if (mapgrid[i, j] == 0)
                {
                    //Debug.Log(sun_b);
                    if (GetObjSideNumber(i, j) >= 5)
                    {
                        mapgrid[i, j] = 1;
                    }
                }
            }
```

```
            }
        }

        //在地图数组生成水晶的位置 2/3 生成攻击水晶，1/3 生成能量水晶
        void CreateCrystalsv()
        {
            for (int i = 1; i < mapsize - 1; i++)
            {
                for (int j = 1; j < mapsize - 1; j++)
                {
                    if (mapgrid[i, j] == 1)
                    {
                        if (mapgrid[i + 1, j] == 0 &&
Random.Range(0,10) <= 1)
                        {
                            if (Random.Range(0, 3) > 0)
                                mapgrid[i + 1, j] = 2;
                            else
                                mapgrid[i + 1, j] = 3;
                        }
                        if (mapgrid[i - 1, j] == 0 && Random.Range(0,
10) <= 1)
                        {
                            if (Random.Range(0, 3) > 0)
                                mapgrid[i - 1, j] = 2;
                            else
                                mapgrid[i - 1, j] = 3;
                        }
                        if (mapgrid[i, j + 1] == 0 && Random.Range(0,
10) <= 1)
                        {
                            if (Random.Range(0, 3) > 0)
                                mapgrid[i, j + 1] = 2;
```

```
                    else
                        mapgrid[i, j + 1] = 3;
                }
                if (mapgrid[i, j - 1] == 0 && Random.Range(0,
10) <= 1)
                {
                    if (Random.Range(0, 3) > 0)
                        mapgrid[i, j - 1] = 2;
                    else
                        mapgrid[i, j - 1] = 3;
                }
            }
            else
                continue;
        }
    }
}
```

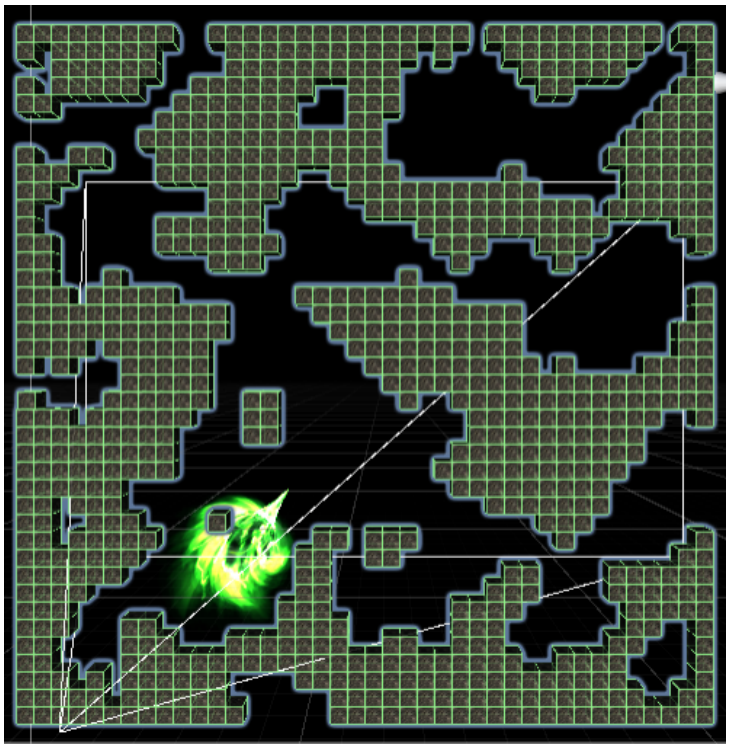

图 4 初始地图

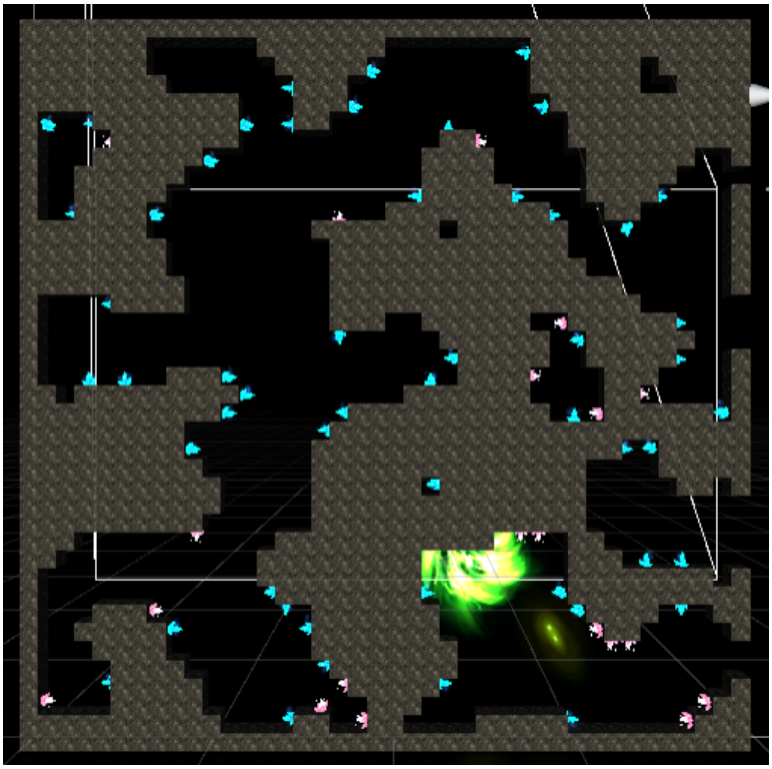

图 5 随机地图

4.3 玩家

4.3.1 玩家设计

玩家模型来源于 AssetStore 资源包 Robot Kyle 里的机器人模型，模型可以通过 www.mixamo.com 绑定骨骼，如图 6。玩家模型绑好骨骼后导入项目前最好在 Hierarchy 里新建一个空物体 Player，之后将玩家模型放到空物体里，调整参数后保存为预制体，方便后期调整。Mixamo 同样可以寻找并下载角色动画。

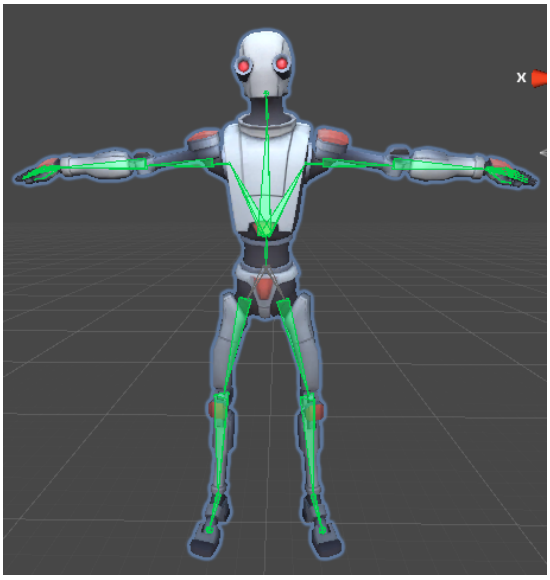

图 6 玩家和骨骼

4.3.2 玩家物理效果

玩家的物理效果的实现没有使用碰撞器和刚体的组合，而用的是 Unity3D 自带的玩家控制器组件 CharacterController。本次课题曾尝试用过胶囊、正方形的碰撞器，虽然原理上面差不多，但是在实际的场景里有些麻烦的问题，胶囊形刚体在场景里移动的时候会有细小的抖动，换成方块形的物体则是会有被地面卡住的现象，如果追求更好的交互效果，比较适合用网格碰撞器 GenerateColliders。网格碰撞器是一种复杂的、贴合于模型结构的碰撞器，能够 达到很好的交互效果，但是同时消耗大量的计算机资源，影响场景 表现性能 Error! Reference source not found.。

组件 CharacterController 能够解决移动的时候的卡顿问题，但是 Unity3D 自带的物理下落效果有些不尽人意，而且 CharacterController 组件上默认启用了刚体效果，虽然一般可以通过 Edit→ProjectSettings→Physics 里的 Gravity 调整刚体的重力参数来决定玩家的下落速度，但是当视野宽广的时候，下落的效果没达到预期效果。本课题解决办法是在判断玩家离开地面的时候添加一个不断增长的垂直向下的速度 velocity.y += gravity * Time.deltaTime，在跳跃的时候用物理的跳跃公式，速度 V=$\sqrt{(jumpHeight * -2f * grivaty)}$即 velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity)。由于玩家在空中的时候，向下的速度就会不断增大，所以在玩家接触地面时强制让向下的速度变为 0.0f。

```
public CharacterController CC;
velocity.y += 重力 * Time.deltaTime;
CC.Move(velocity * Time.deltaTime);
```

4.3.3 玩家动画

玩家动画和 UI 动画一样是通过 Unity3D 组件 Animator 来实现的，但 AssetStore 导入的人物模型不附带人物动画，获取人物动画一个方法是自己在 Unity3D 中通过 Animation 窗口来自己制作，另一个方法是找其他带动作的模型绑到玩家上使用，由于基本是人类动画，可以选择将人物模型放到网站 Mixamo 下载当前人物模型的其他动作。下载后的人物模型导入后将其中的一个模型的骨骼 Avatar 放到自己玩家模型的 animator 组件上，之后把在 Mixamo 下载的模型在 Inspector 窗口里找到 Rig 之后把 Animation Type 改为 Humanoid，Avatar Definition 选为 Copy From Other Avatar，之后就可以将其他的模型动画共用一套骨骼了。玩家动画齐全后可以在 Project 新建一个动画控制器和 C#脚本来控制玩家动画的播放，动画控制器如图 4-2-2-3-1。C#脚本主要通过检查玩家当前处于的状态来改变动画控制器左侧参数的变化。相关代码如下

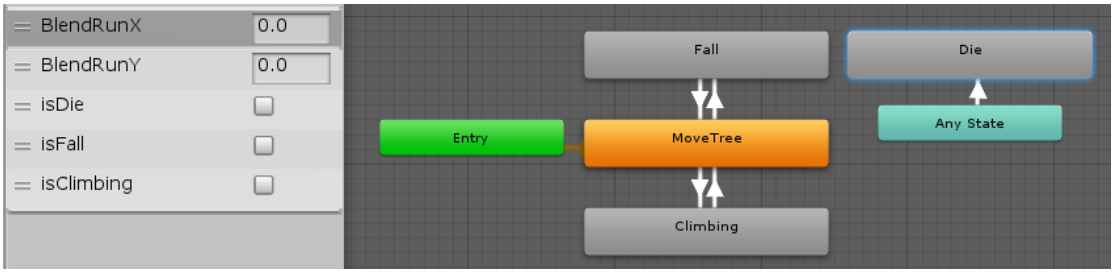

图 7 动画控制器

代码中的 Animator.StringToHash(String)是将动画控制器里的参数转换成数字序号，提前获取参数的数字序号就不用输入可能输错 string 类型的名称，而且有些平台是不支持 SetBool(string,true),这里用 SetBool(int,true)更保险。

```
public Animator Anim;
    //动画参数序号
    int  下落参数序号;
      ……
    void Start()
    {
        下落参数序号  = Animator.StringToHash("isFall");
       ……
    }
```

```
void PlayerAnimation()
{
    if (角色是否处于下落状态)
        Anim.SetBool(下落参数序号, true);
    else
        Anim.SetBool(下落参数序号, false);
    ……
}
```

PhysicsCheck()通过半球形检测判断玩家是否在地面上，是否正在跳跃。

```
void PhysicsCheck()
{
Physics.CheckSphere 是球形物理检测
        是否在地上= Physics.CheckSphere(玩家脚步的点, 检测半径, 检测的图层);
……
}
```

4.3.4 玩家移动

玩家的移动利用的是 CharacterController 组件里的 Move(Vector3 motion)，Move()可以提供一个方向的恒定推力，通过 Horizontal 的值来实现玩家的走动。给物体添加一个 Physic Material 并将其的摩擦力调为零，这样移动的时候当物体紧贴其他物体下落的时候就不会有摩擦力使玩家不能平滑的下落。

玩家的主要移动是通过“Horizontal”的值来实现的，在编辑里的项目设置中可以找到。当玩家按下 A 键时“Horizontal”的值从 0 变为-1，按下 D 键时从 0 变为 1，没有按下时为 0。通过按下 A 或 D 键时“Horizontal”的值来判断玩家运动的方向，再给玩家一个推力就可以实现玩家的移动。相关代码如下

```
playerX = Input.GetAxis("Horizontal");
Vector3 移动方向 = new Vector3(playerX, 0f, 0f);
//转向
if (按下 A 键)
    向左旋转玩家
if (按下 D 键)
```

向右旋转玩家

CC.Move(移动方向*移动速度*Time.deltaTime);

4.4 助手

助手的功能有四个，分别是攻击、短距离传送、建造物体和销毁物体。通过 Tap 键来启用助手，键盘上的 1,2,3,4 分别对应一个模式。按 1 时处于攻击模式时，助手会从方块变为攻击模式，长按鼠标左键放出激光束，如图 8。按 2 时处于短距离传送模式的时候会在 10 个单位内生成小型传送通道，单机左键瞬移。按 3 或 4 时是单击左键时建造或销毁物体。在玩家身上设置一个跟随点通过Vector3.Lerp（助手位置,玩家位置,speed）来实现，虽然速度不是恒定的但效果不错。

函数 AssisterChange()负责对应的 1,2,3,4 的模式切换并在切换时隐藏其他模式时产生的物体。模式的切换是用键盘上的 Alpha1、Alpha2、Alpha3 和 Alpha4 来实现的

```
if (按下对应的数字键)
{
    启动对应的模式
    关闭其它模式
    if (有其它模式下对象在场景里)
    {
        隐藏或销毁对应的对象
    }
}
```

当切换到模式 2 下时通过射线判定传送通道位置

```
Ray _ray = new Ray(射线原点, 射线方向);
RaycastHit _hit;
bool _rayCheck = Physics.Raycast(_ray, out _hit, 射线长度, 检测图层);
if (未生成传送通道)
    生成传送通道
else
{
    if (_rayCheck)
```

```
                    {
                        Vector3 _vp =  接触点的坐标  +  接触点的法向量
                        传送通道位置= new Vector3(_vp.x, _vp.y, 0);
                    }
                    else
                        传送通道位置=射线的原点+射线的方向*射线长度
                }
```

函数 Shoot()负责启动对应模式时的鼠标事件和助手旋转

```
void Shoot()
    {
        //炮塔旋转
        if (isAttack || isLongTransmit)
        {
            //获取鼠标坐标
            Vector3 mouse = Input.mousePosition;
            //获取物体中心点坐标,并转换成屏幕坐标
            Vector3 obj =
Camera.main.WorldToScreenPoint(transform.position);
            //向量相减
            Vector3 direction = mouse - obj;
            //z 轴为 0
            direction.z = 0f;
            //将目标向量长度变成单位向量,获得方位向量
            direction = direction.normalized;
            //物体自身 Y 轴和目标向量保持一致
            transform.right = direction;
        }
        //按下鼠标左键时
        if (Input.GetButtonDown("Fire1"))
        {
            if (攻击模式)
            {
                if (bullets.childCount < 1)
                {
```