游戏开发设计

实训报告

作品题目：

1. 实训目的

本课程将在 Unity 3D 软件上学习游戏开发设计。掌握软件功能及各种操作技巧，深入学习从概念设计到目标用户调研；学会游戏设计制作到发布 3D 游戏中各阶段的完整应用开发流程。本课程将教授对于可运行游戏的标准需求，着重 C#程序开发和硬件设备的图形性能极限以及克服这些限制的技巧。

2. 实训内容

- 理解模型设计、类人绑定、动画和运动；
- 能创建图形用户界面、角色定制和武器系统；
- 理解通过脚本语言实现摄像机和用户输入控制；
- 使用 UV 分层和灯光控制创建具有良好视觉效果的游戏环境；
- 从不同类型和年龄的游戏受众群体中区分目标用户；
- 能充分理解专业游戏中的关卡难度设计。

3. 作品创意：

本作品是一个潜行刺杀类游戏，玩家在一个随机生成的地图中，需要找到位于角落的传送门以传送到新的场景中。每次传送的敌人随机分布，地形也随机生成。

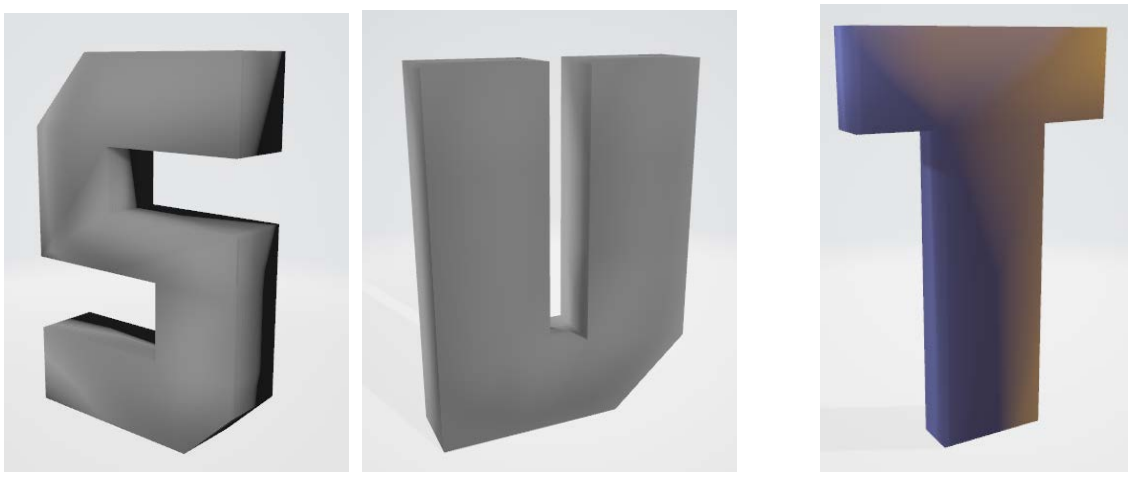
玩家拥有三种子弹，可以利用它们进行通关：空间弹用于让玩家快速在场景中进行穿梭；振动弹使玩家快速穿过一些阻碍物；时间弹用于暂停一定区域内的时间。以上子弹的灵感来源于平日所看到的影视以及游戏作品。

除此之外，游戏希望采用 Rougelike 的方式，让玩家一直游玩至结束，每次敌人攻击力会提高，但玩家的攻击力与子弹量上限也会提升，玩家可以通过击杀敌人来提升自己的能力。由于一旦死亡变会结束游戏，使得玩家必须小心地做出行动。

4. 详细设计：

关于美术：

由于时间原因，本作品的大多数模型与材质从网上进行了下载与使用，仅在开始菜单部分使用 3DSMAX 和 Substance Painter 制作了简单的字母模型：

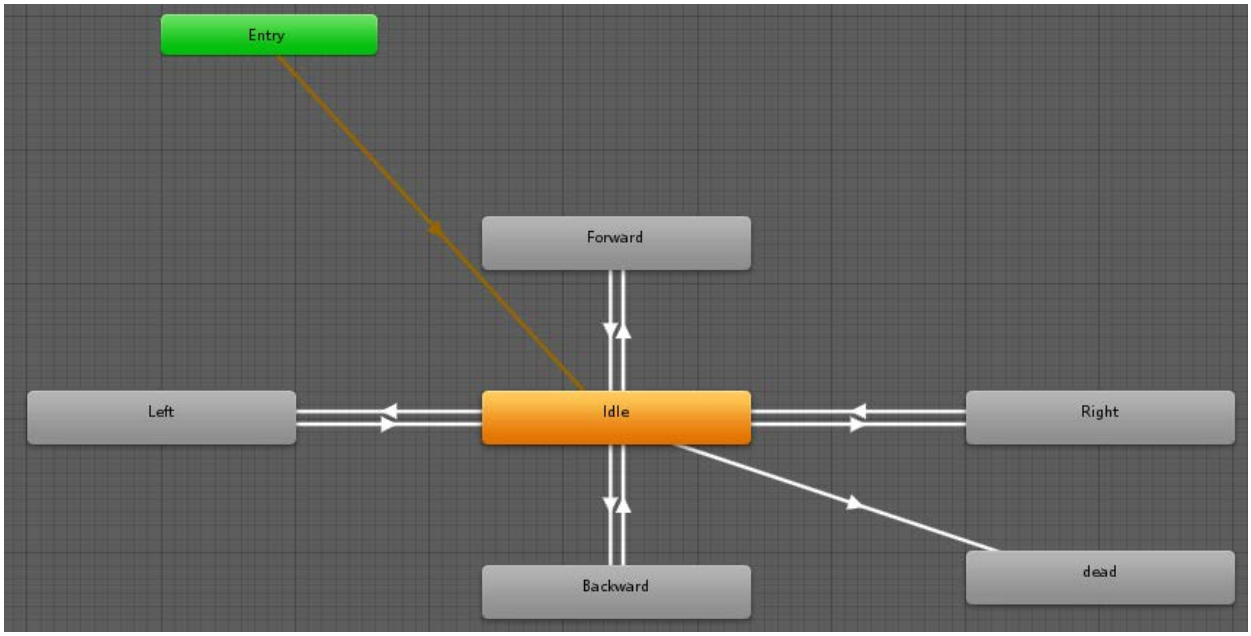


在动画方面，利用一个状态机，分层实现移动以及设计等动画

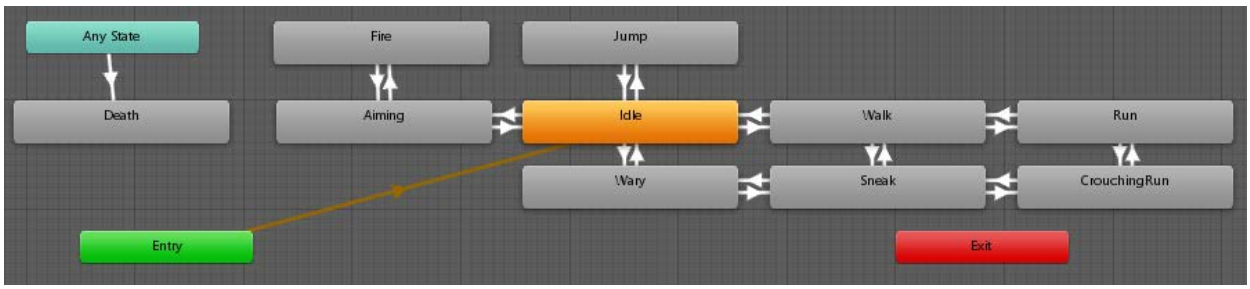
玩家：

包含行走、射击动画，不同方向的行走有不同的动画效果

同时，采用分层级混合效果以保证可以在行走时进行近战攻击



敌人：



关于主界面设计：

采用非平面的 UI 设计，并且灯光以及摄像头视角会随着鼠标进行移动，更加有代入感

关于音效与音乐：

音效与音乐均从网络下载，进行了简单的裁剪加工

关于程序：

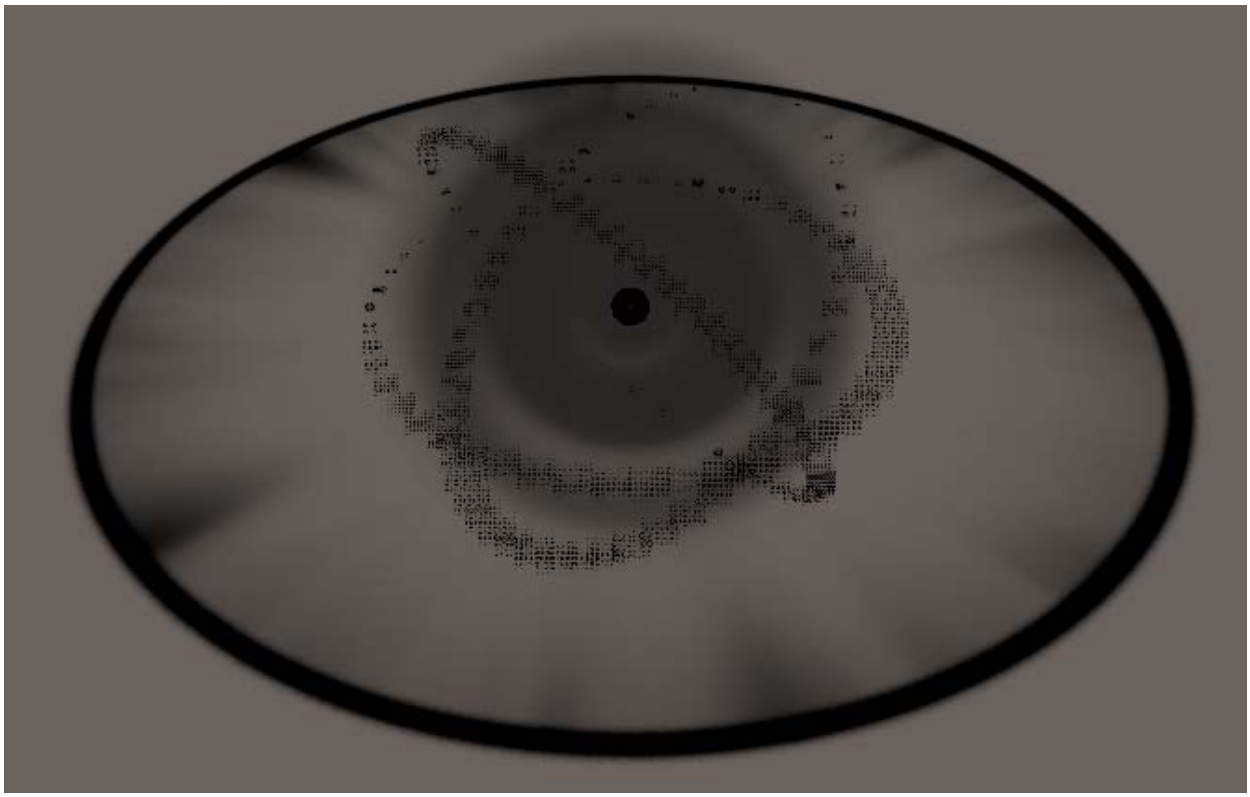
程序大部分由自己编写，因此将着重介绍这一部分

（具体代码详见附录）

首先介绍最有特色的三种子弹：

空间弹：

编写较为简单，思路是获取玩家对象，将其以很快的速度移动到子弹所在位置，当子弹与玩家之间距离足够近时销毁子弹。玩家穿越时，设定了如果撞到了任何碰撞体，则会立即停下，同时子弹也随之销毁



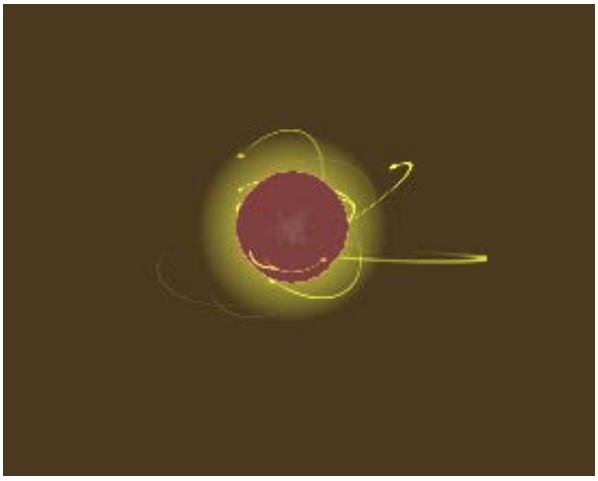
图：空间弹样式

振动弹：

编写难度较大。当振动弹刚出现时，删除本体的所有除脚本、动画器、变换组件等组件再在其两侧复制出仅含有显示功能的物体；

当振动弹碰到了玩家或者墙壁时，将会附着在上面，并销毁之前的复制体；此时会让附着上的物体进行振动。

其主要难度在于究竟要保留以及销毁哪些组件以及如何获取它们



图：振动弹本体样式

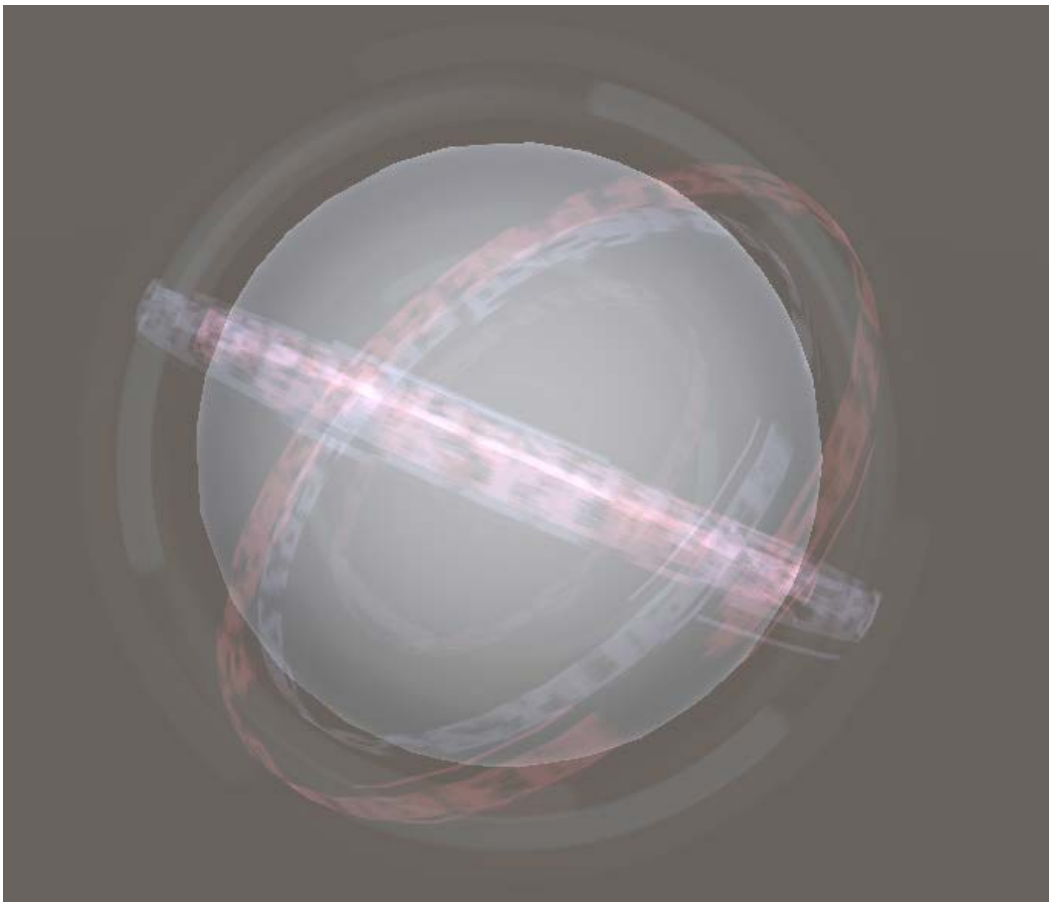
时间弹：

编写难度尚可，但是需要较为清楚认识到 Unity 脚本的一些知识点

当时间弹被激活之后，本体变大，之后将利用射线获取球内的所有碰撞体并记录。将这些碰撞体对应游戏物体的脚本全部禁用，如果含有刚体组件也将被消除

当时间子弹效果结束后，子弹缩小至消失，同时启用所记录物体的脚本以及刚体

另外，子弹的反色效果是通过让子弹的透明度超过 1 而得到的，属于无心之举但效果意外不错。



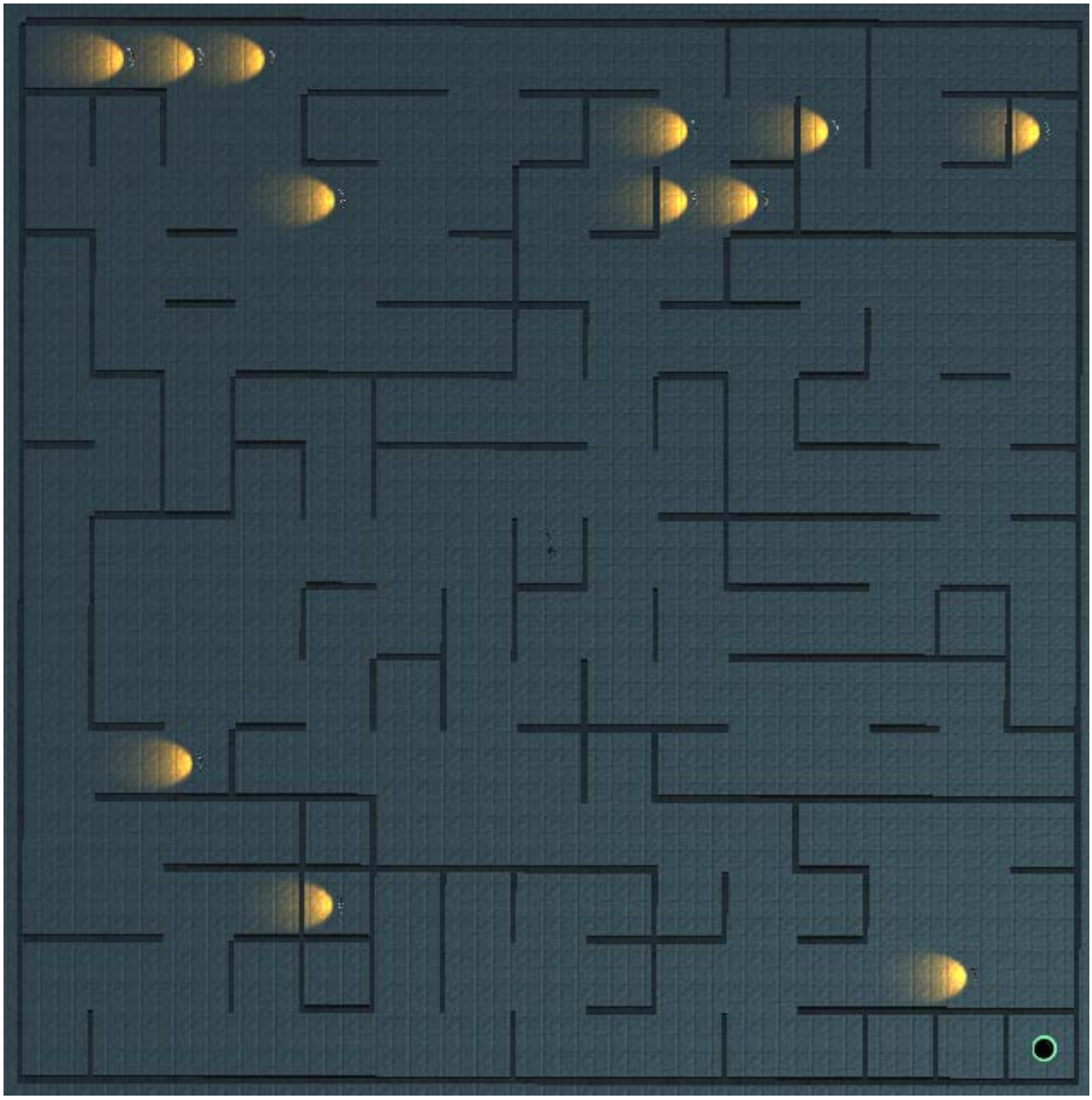
图：时间弹样式

另外，编程的难点在于地形的生成：

首先为了随机生成地形，利用矩阵进行地形的创造，利用随机数决定是否会生成中间的障碍物，而周围的障碍物则会必然生成

敌人的数量则设定为单个地形之间的间隔，仍然利用在挡板之间的空隙进行生成

传送门则生成在地图的四个角落的其中一个



图：地形与敌人生成样式

在 UI 方面，使用让玩家的生命值与 UI 指示结合的方式，UI 不显示数值而是以旋转方式显示当前血量以及子弹使用量

图：UI 样式

另外，作品在 AI 方面也有一定的技术含量：

AI 会在前方的一定范围内进行检测是否含有玩家，看到会进行攻击；如果看到了墙壁并发现离墙过近时则会旋转至其他角度。这里利用了大量的布尔变量以及比较巧妙的射线检测方案

5. 实训总结

自我评价：（作品特点，存在问题）

本作品基本到达了所要实现的要求，尽管由于时间等原因还有很大的进步空间，但是我还是比较满意可以到达基本的要求。

作品存在一些提示问题，尤其在振动后的效果、教程部分等等，需要在以后进行加强与改善

另外，之前试图实现的一些例如更加“聪明”的 AI 由于多种原因没能很好的实现，需要以后继续努力学习。

总体而言，这个作品比较好的实现了“技艺结合”，希望自己以后在有精力的条件下能够自己制作一个技术和艺术方面都自己完成的游戏。

实训体会：（学习到的知识、技能、方法，有待于深入学习的知识等等）

本次实训着实让我学到了很多知识。

在软件方面，我深入了解了 Unity 的渲染、声音、刚体、碰撞体、脚本等方面的知识，知晓了诸如模型设置、组件知识等方面的要点。

另外一点就是在大量的编程之后，我知晓了编写代码规范性的重要，以及如何实现“低耦合、高内聚”的代码。很大程度上提升了自己编写的水平以及 Unity 脚本的相关知识。

另外，本次实训教会了我如何以及管理资源的必要性：当面对繁多的资源时，一定要通过管理的

方式来保持工作效率。

最后，我认为自己一方面要继续加强代码的编写能力，另一方面要学习一定的模型建立与导入能力，目前只能做到低模的导入，显然不能满足最终需要，还需努力。

附录：

关键代码（代码要有注释）

空间弹：

```
using UnityEngine;
public class SpaceBulletController : BulletsMoveController, BulletsAbilityController/*空间弹控制器，继承 BulletsMoveController，实现 BulletsAbilityController*/
{
    private GameObject player;//玩家
    public float transfer_speed;//瞬移速度
    public static bool transfer = false;//是否进行瞬移
    public static bool effected_by_time = false;//是否受到了时间弹影响
    public GameObject follow_smoke;//跟随的烟雾
    public GameObject space_bullet_vfx;//空间弹特效

    /*脚本被启用时*/
    private void OnEnable()
    {
        effected_by_time = false;//设定没有受到时间影响
        Move();//移动
    }

    /*每帧更新的部分*/
    private void Update()
    {
        Transfer();
    }

    /*脚本被禁用时*/
    private void OnDisable()
    {
        effected_by_time = true;//设定了受到了时间弹影响
    }

    /*瞬移*/
    private void Transfer()
    {
```

```
        if (transfer)//如果确认进行瞬移
        {
            if (GetComponent<AudioSource>().clip.name != "Transfer")//如果声音名字不是 Transfer
            {
                GetComponent<AudioSource>().clip =
(AudioClip)Resources.Load("Audio/Bullets/Transfer");//将声音替换为 Transfer
                GetComponent<AudioSource>().volume = 0.05f;//降低音量
                GetComponent<AudioSource>().pitch = 1.5f;//提高音调
            }
            if (!GetComponent<AudioSource>().isPlaying && GetComponent<AudioSource>().loop)//如果声
音没有在播放并且处于循环播放状态
            {
                GetComponent<AudioSource>().loop = false;//关闭循环播放
                GetComponent<AudioSource>().Play();//播放声音
            }

            if (GetComponent<Rigidbody>().velocity != Vector3.zero)//如果子弹没有停下
            {
                GetComponent<Rigidbody>().velocity = Vector3.zero;//让子弹停下
            }
            player.GetComponent<Rigidbody>().velocity = (transform.position -
player.transform.position) * transfer_speed;//瞬移玩家

            if ((transform.position - player.transform.position).sqrMagnitude < 15 ||
player.GetComponent<PlayerMoveController>().hit_when_transfer)//如果玩家与子弹之间距离足够小或者玩
家撞到了任何碰撞体
            {
                transfer = false;//确认停止瞬移

                space_bullet_vfx.transform.parent = null;//脱离父子关系
                foreach(Transform space_bullet_vfxs in
space_bullet_vfx.GetComponentsInChildren<Transform>())//对于粒子效果
                {
                    ParticleSystem.MainModule main_module =
space_bullet_vfxs.GetComponent<ParticleSystem>().main;
                    main_module.loop = false;//停止粒子循环
                }
                Destroy(space_bullet_vfx, 4.5f);//4.5s 后销毁物体

                player.GetComponent<PlayerMoveController>().hit_when_transfer = false;//设定现在玩
家没有在瞬移
                Destroy(gameObject);//销毁子弹
            }
        }
    }
```

```
    /*激活能力*/
    void BulletsAbilityController.ActivateAbility()
    {
        transfer = true;//确认进行瞬移

        player = GameObject.FindGameObjectWithTag("Player");//找到玩家

        Instantiate(follow_smoke, player.transform.position, Quaternion.identity);//在玩家位置实例化烟雾

        GetComponent<Collider>().enabled = false;//设定空间弹无碰撞效果
        GetComponent<Rigidbody>().velocity = Vector3.zero;//让子弹停下
    }
}
```

振动弹：

```
using UnityEngine;
public class VibrationBulletController : BulletsMoveController, BulletsAbilityController/*振动弹控制器，继承 BulletsMoveController，实现 BulletsAbilityController*/
{
    private GameObject copy_left;//本体的左复制体
    private GameObject copy_right;//本体的右复制体
    private bool copied = false;//是否已经进行了复制
    private bool copy_left_active;//做复制体是否被激活

    public float vibrate_range;//振动幅度
    public float vibrate_frequence;//振动频率
    private float timer;//计时器
    public GameObject attach_place;//附着点

    private bool activable = false;//是否可被激活
    private bool actived = false;//是否被激活

    private GameObject vibrate_object;//需要振动的物体

    public float vibrate_time;//可以振动的最长时间

    /*初始化*/
    private void Start()
    {
        timer = vibrate_frequence;//设定计时器
    }

    /*脚本被启用时*/
```

```
    private void OnEnable()
    {
        Move();//移动
    }

    /*每帧更新的部分*/
    private void Update()
    {
        if (!actived)//如果没有被激活
        {
            Vibrate(gameObject);//自己进行振动
        }
        else//如果被激活
        {
            Vibrate(vibrate_object);//振动附着上的物体的根物体
            transform.position = vibrate_object.transform.position;//始终跟随附着上的物体
            if(vibrate_time > 0)//若计时未结束
            {
                vibrate_time -= Time.deltaTime;//计时 vibrate_times
            }
            else//若计时结束
            {
                foreach(Renderer renders in vibrate_object.GetComponentsInChildren<Renderer>())//
对于振动的物体及其子物体的渲染器
                {
                    renders.enabled = true;//设定可见
                }
                foreach (Collider colliders in vibrate_object.GetComponentsInChildren<Collider>())//
对于振动的物体及其子物体的碰撞器
                {
                    colliders.isTrigger = false;//关闭触发器
                }
                Destroy(copy_left);//销毁左复制体
                Destroy(copy_right);//销毁右复制体
                Destroy(attach_place);//销毁附着点
                Destroy(gameObject);//销毁子弹
            }
        }
    }

    /*振动弹与碰撞体相遇*/
    private void OnCollisionEnter(Collision collision)
    {
        if(GetComponent<MonoBehaviour>().enabled && collision.gameObject.tag != "UnthroughableBlock"
&& collision.gameObject.tag != "EnermyBullet" && collision.gameObject.tag != "Enermy")//如果振动弹
```

```
脚本存在，除去不可穿越的阻碍物以及敌人的子弹，以及玩家和爹
        {
            Destroy(GetComponent<Rigidbody>());//销毁刚体组件，使子弹停下
            GetComponent<Collider>().enabled = false;//子弹碰撞器禁用
            vibrate_object = collision.transform.root.gameObject;//获取被碰撞的对象的根物体
            attach_place = Instantiate(attach_place, collision.contacts[0].point,
Quaternion.identity);//在碰撞点实例化一个附着点
            attach_place.transform.parent = vibrate_object.transform;//设定附着点为被碰撞对象根物体
的子物体
            transform.parent = attach_place.transform;//设定子弹是附着点的子物体
            activable = true;//设定可以被激活
        }
    }

    /*振动*/
    private void Vibrate(GameObject game_object)//game_object 为要振动的物体
    {
        if (!copied)//如果没有进行过复制
        {
            Copy(game_object);//复制物体
            copied = true;//设定已经复制了物体
        }
        if (!copy_left_active)//如果左复制体不可见
        {
            if (Timing())//进行计时，如果计时结束
            {
                foreach (Renderer renderer in copy_left.GetComponentsInChildren<Renderer>())//对于
左复制体及其子物体
                {
                    renderer.enabled = true;//设置为可见
                }
                foreach (Renderer renderer in copy_right.GetComponentsInChildren<Renderer>())//对于
右复制体及其子物体
                {
                    renderer.enabled = false;//设置为不可见
                }
                copy_left_active = true;//设定左复制体可见
            }
        }
        else//如果右复制体不可见
        {
            if (Timing())//进行计时，如果计时结束
            {
                foreach (Renderer renderer in copy_left.GetComponentsInChildren<Renderer>())//对于
左复制体及其子物体
```

```
                {
                    renderer.enabled = false;//设置为不可见
                }
                foreach (Renderer renderer in copy_right.GetComponentsInChildren<Renderer>())//对于右复制体及其子物体
                {
                    renderer.enabled = true;//设置为可见
                }
                copy_left_active = false;//设定左复制体不可见
            }
        }
    }

    /*复制本体*/
    private void Copy(GameObject game_object)
    {
        copy_left = Instantiate(game_object);//复制本体到左复制体
        copy_right = Instantiate(game_object);//复制本体到右复制体
        if (game_object.tag == "Enermy")//如果复制对象是敌人
        {
            copy_left.GetComponent<PlayerController>().SetArsenal("Rifle");//设置左复制体玩家的枪
            copy_right.GetComponent<PlayerController>().SetArsenal("Rifle");//设置右复制体玩家的枪
        }
        if (game_object == gameObject)//对于子弹本身
        {
            GetComponent<Renderer>().enabled = false;//不进行渲染
        }
        else//对于要振动的物体
        {
            foreach (Renderer renderer in game_object.GetComponentsInChildren<Renderer>())//对于本体及其子物体
            {
                renderer.enabled = false;//设置为不可见
            }
        }
        RemoveComponents(copy_left);//移除左复制体组件
        RemoveComponents(copy_right);//移除右复制体组件
        copy_left.transform.position += new Vector3(vibrate_range, 0, 0);//移动左复制体到正确位置
        copy_right.transform.position -= new Vector3(vibrate_range, 0, 0);//移动右复制体到正确位置
        copy_right.transform.parent = copy_left.transform.parent = game_object.transform;//设定左、右复制体为本体子对象
        foreach (Renderer renderer in copy_left.GetComponentsInChildren<Renderer>())//对于左复制体及其子物体
        {
            renderer.enabled = false;//设置为不可见
```

```
        }
        copy_left_active = false;//设定左复制体没有被激活
    }

    /*移除组件*/
    private void RemoveComponents(GameObject game_object_copy)//game_object_copy 是复制体
    {
        foreach (Component components in game_object_copy.GetComponentsInChildren<Component>())//
对于复制体及其在同一个树分支下的所有组件
        {
            if (!(components.GetType().ToString() == "UnityEngine.Transform")
&& !(components.GetType().ToString() == "UnityEngine.MeshRenderer")
&& !(components.GetType().ToString() == "UnityEngine.SkinnedMeshRenderer")
&& !(components.GetType().ToString() == "UnityEngine.MeshFilter")
&& !(components.GetType().ToString() == "UnityEngine.Animator") && !(components.GetType().ToString()
== "PlayerAnimatorController"))//除去变换组件、网格过滤器与渲染器以及人物动画控制器
            {
                Destroy(components);//销毁这个组件
            }
        }
    }

    /*计时*/
    private bool Timing()
    {
        if (timer > 0)//进行计时
        {
            timer -= Time.deltaTime;//计时 vibrate_frequence 秒
            return false;
        }
        else//计时结束
        {
            timer = vibrate_frequence;//重置计时器
            return true;
        }
    }

    /*激活能力*/
    void BulletsAbilityController.ActivateAbility()
    {
        if (activable)//如果可被激活
        {
            foreach (Component components in GetComponentsInChildren<Component>())//对于子弹本体及
其子物体
            {
```

```
                if (!(components.GetType().ToString() == "UnityEngine.Transform")
&& !(components.GetType().ToString() == "VibrationBulletController"))//除去变换组件与
VibrationBulletController 脚本
                {
                    Destroy(components);//销毁这个组件
                }
            }
            transform.tag = "Untagged";//设定标签为 Untagged
            transform.parent = null;//解除与物体的父子关系
            Destroy(copy_left);//销毁左复制体
            Destroy(copy_right);//销毁右复制体
            copied = false;//设定还没有复制物体
            actived = true;//设定已经激活
            activable = false;//设定子弹不可再被激活
            foreach (Collider colliders in vibrate_object.GetComponentsInChildren<Collider>())//对
于被附着物体以及其子物体的碰撞体
            {
                colliders.isTrigger = true;//使碰撞体变为触发器
            }
        }
    }
}
```

时间弹：

```
using UnityEngine;
using System.Collections.Generic;
public class TimeBulletController : BulletsMoveController, BulletsAbilityController/*时间弹控制器，
继承 BulletsMoveController，实现 BulletsAbilityController*/
{
    private bool active = false;//子弹是否被激活
    public float scale_speed;//子弹的放大速度
    public float max_size;//子弹的最大放大倍数
    public float color_speed;//子弹的颜色改变速度
    public float gray_level;//灰度
    public float alpha_level;//透明度
    private List<GameObject> game_objects_in_sphere = new List<GameObject>();//在场景中的所有物体
    private GameObject game_object_in_sphere;//在场景中的某个物体
    public float pause_time;//子弹效果持续时间
    private bool audio_changed = false;//是否改变了声音
    public GameObject time_bullet_vfx;//时间弹特效

    /*初始化*/
    private void Start()
    {
        GetComponent<Renderer>().material.color = new Color(gray_level, gray_level, gray_level,
```

```
alpha_level);//设定初始颜色
    }

    /*脚本启用时*/
    private void OnEnable()
    {
        Move();//移动
    }

    /*每帧更新的部分*/
    private void Update()
    {
        if (active && pause_time > 0)//如果子弹被激活并且效果未结束
        {
            if (transform.position.y > -2.3f)//如果子弹高度大于 2.3f
            {
                transform.position -= new Vector3(0, Time.deltaTime, 0);//降低子弹高度
            }
            if (tag != "Untagged")//如果子弹标签不是"Untagged"
            {
                tag = "Untagged";//设定子弹标签是"Untagged"
            }
            if (transform.localScale.y < max_size)//子弹放大未结束
            {
                transform.localScale += Vector3.one * scale_speed;//以每帧 scale_speed 的速度放大子
弹
            }
            else//放大结束
            {
                foreach (Collider colliders in Physics.OverlapSphere(transform.position, max_size /
2))//找到球内所有碰撞体
                {
                    if (!game_objects_in_sphere.Contains(game_object_in_sphere =
colliders.gameObject) && game_object_in_sphere != gameObject && game_object_in_sphere.tag !=
"Player")//除去时间弹、玩家，获取碰撞体对应的物体，如果没有被记录
                    {
                        game_objects_in_sphere.Add(game_object_in_sphere);//记录这个物体
                        if (game_object_in_sphere.GetComponent<Rigidbody>()
&& !game_object_in_sphere.GetComponent<Rigidbody>().isKinematic)//如果物体包含刚体且物体不是运动学
的
                        {
                            game_object_in_sphere.GetComponent<Rigidbody>().isKinematic = true;//
让物体停下来
                        }
                        if (game_object_in_sphere.GetComponent<Animator>())//如果物体包含动画组件
```

```
                        {
                            game_object_in_sphere.GetComponent<Animator>().SetFloat("speed", 0);//
停止播放动画
                        }
                        foreach (MonoBehaviour scripts in
game_object_in_sphere.GetComponents<MonoBehaviour>())//对于其中物体，找到它的所有脚本
                        {
                            scripts.enabled = false;//禁用这些脚本
                        }
                    }
                }
                pause_time -= Time.deltaTime;//子弹进行计时
            }
        }
        if(pause_time < 0)//如果子弹计时结束
        {
            if (!audio_changed)//如果音频没有改变
            {
                GetComponent<AudioSource>().clip =
(AudioClip)Resources.Load("Audio/Bullets/TimeBulletReverse");//切换音频
                audio_changed = true;//设定改变完成
            }
            if (!GetComponent<AudioSource>().isPlaying)//如果音效没有在播放
            {
                GetComponent<AudioSource>().Play();//播放音效
            }

            if (transform.localScale.y > 0)//子弹缩小未结束
            {
                transform.localScale -= Vector3.one * scale_speed;//以每帧 scale_speed 的速度缩小子
弹
            }
            else//子弹缩小结束
            {
                foreach (GameObject game_object_in_sphere in game_objects_in_sphere)//对于记录过的
物体
                {
                    if (game_object_in_sphere.GetComponent<Rigidbody>())//如果物体包含刚体
                    {
                        game_object_in_sphere.GetComponent<Rigidbody>().isKinematic = false;//禁用
物体的运动学
                    }
                    if (game_object_in_sphere.GetComponent<Animator>())//如果物体包含动画组件
                    {
                        game_object_in_sphere.GetComponent<Animator>().SetFloat("speed", 1);//继续
```

```
播放动画
                    }
                    foreach (MonoBehaviour scripts in
game_object_in_sphere.GetComponents<MonoBehaviour>())//对于其中物体，找到它的所有脚本
                    {
                        scripts.enabled = true;//开启这些脚本
                    }
                }
                Destroy(gameObject);//销毁子弹本身
            }
        }
    }

    /*激活能力*/
    void BulletsAbilityController.ActivateAbility()
    {
        if (!time_bullet_vfx.activeSelf)//如果时间弹特效没有被激活
        {
            time_bullet_vfx.SetActive(true);//激活时间弹特效
        }
        if (!active)//如果子弹没有被激活
        {
            active = true;//设定子弹被激活
            Destroy(GetComponent<Rigidbody>());//移除子弹的刚体组件
            Destroy(GetComponent<Collider>());//移除子弹的碰撞体组件
            if (!GetComponent<AudioSource>().isPlaying)//如果音频没在播放
            {
                GetComponent<AudioSource>().Play();//播放时停音频
            }
        }
    }
}
```

敌人 AI：

```
using UnityEngine;
public class DetectObjects : MonoBehaviour/*检测物体*/
{
    public Transform detect_start;//检测起点
    public float block_max_distance;//检测阻碍物的最大长度
    public float player_max_distance;//检测玩家的最大长度
    public float max_degree;//检测的最大角度
    [HideInInspector] public bool detected_block = false;//是否检测到障碍物
    [HideInInspector] public bool detected_player = false;//是否检测到玩家
    private RaycastHit hit_info;//射线击中信息
    private bool player_hit = false;//玩家是否撞上来
```

```
    /*每帧更新的部分*/
    private void Update()
    {
        if (!player_hit)//如果玩家没有主动撞上来
        {
            detected_player = detected_block = false;//假定什么都没有检测到
        }
        for (detect_start.localEulerAngles = new Vector3(0, 360 - max_degree, 0);
detect_start.localEulerAngles.y >= (360 - max_degree) || detect_start.localEulerAngles.y <= max_degree;
detect_start.localEulerAngles += new Vector3(0, 1, 0))//在敌人左右 70 度范围内
        {
            if (Physics.Raycast(detect_start.position, detect_start.forward, out hit_info,
player_max_distance))//向检测方向前方 player_max_distance 长度发射射线，如果检测到任何碰撞体
            {
                if (hit_info.collider.tag == "Player")//如果检测到的是玩家
                {
                    if (!GetComponent<EnermyMoveController>().enabled)//如果自己无法移动
                    {
                        GetComponent<EnermyMoveController>().enabled = true;//设定自己可以移动
                    }
                    if (!GetComponent<EnermyRotateController>().enabled)//如果自己无法旋转
                    {
                        GetComponent<EnermyRotateController>().enabled = true;//设定自己可以旋转
                    }
                    detected_player = true;//设定检测到玩家
                    player_hit = false;//设定玩家没有碰到敌人
                    GetComponent<AlarmController>().saw_player = true;//设定见过玩家
                    break;
                }
            }
            if (Physics.Raycast(detect_start.position, detect_start.forward, out hit_info,
block_max_distance))//向检测方向前方 block_max_distance 长度发射射线，如果检测到任何碰撞体
            {
                if (hit_info.collider.tag == "Block" || hit_info.collider.tag ==
"UnthroughableBlock")//如果检测到的是障碍物
                {
                    detected_block = true;//设定检测到障碍物
                    break;
                }
            }
        }
    }

    /*如果与任何碰撞体碰撞*/
```

```
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.transform.root.tag == "Player")//如果玩家撞上来
        {
            detected_player = true;//设定检测到玩家
            player_hit = true;//设定玩家撞了上来
        }
    }
}
```

地形生成：

```
using UnityEngine;
using System.Collections.Generic;
public class GeneratorController : MonoBehaviour/*生成控制器*/
{
    private enum BlockType { vertical, horizontal };//阻碍种类的枚举

    public float interval;//间隔长度

    public int min__number;//障碍物生成的最小数(行或列)
    public GameObject block;//障碍物预设

    private GameObject bound_block;//边界阻碍物
    public Transform vertical_start;//纵向阻碍的起始生成点
    public Transform horizontal_start;//横向阻碍的起始生成点

    public GameObject enermy;//敌人预设
    public Transform enermy_start;//敌人生成启动位置
    private Vector2 possible_position;//敌人可以生成的位置
    private List<Vector2> exist_position = new List<Vector2>();//已经记录生成的坐标位置
    private int enermy_number;//敌人生成的数量

    public GameObject portal;//传送门
    private Vector2 portal_position;//传送门生成位置

    public GameObject player;//玩家

    /*初始化*/
    void Start()
    {
        InstantiateBlocks(BlockType.vertical, vertical_start.position);//生成纵向阻碍
        InstantiateBlocks(BlockType.horizontal, horizontal_start.position);//生成横向阻碍
        InstantiateEnermies();//生成敌人
        InstantiatePortal();//生成传送门
        InstantiatePlayer();//生成玩家
```

```
    }

    /*生成矩阵*/
    private void InstantiateBlocks(BlockType block_type, Vector3 generate_position)//BlockType 为阻碍的种类，min_blocktype_number 为阻碍的最小个数(行或列),generate_position 为生成点
    {
        if (block_type == BlockType.vertical)//如果要生成纵向阻碍
        {
            for (int i = 0; i < min__number + 1; i++)//遍历矩阵行数
            {
                for (int j = 0; j < min__number; j++)//遍历矩阵纵数
                {
                    if (i == 0 || i == min__number)//如果是边界
                    {
                        bound_block = Instantiate(block, generate_position, Quaternion.identity);//生成纵向阻碍
                        bound_block.tag = "UnthroughableBlock";//改变障碍物标签为"UnthroughableBlock"
                    }
                    else if (Random.Range(0, 2) == 0)//如果不是边界，约 50%几率生成阻碍物
                    {
                        Instantiate(block, generate_position, Quaternion.identity);//生成纵向阻碍
                    }
                    generate_position += new Vector3(0, 0, interval);//生成下一个阻碍之前更新生成点纵坐标
                }
                generate_position += new Vector3(interval, 0, 0);//生成下一个阻碍之前更新生成点横坐标
                generate_position = new Vector3(generate_position.x, generate_position.y, vertical_start.position.z);//重置纵坐标
            }
        }
        else//如果要生成横向阻碍
        {
            for (int i = 0; i < min__number; i++)//遍历矩阵行数
            {
                for (int j = 0; j < min__number + 1; j++)//遍历矩阵纵数
                {
                    if (j == 0 || j == min__number)//如果是边界
                    {
                        bound_block = Instantiate(block, generate_position, Quaternion.Euler(0, 90, 0));//生成横向阻碍
                        bound_block.tag = "UnthroughableBlock";//改变障碍物标签为"UnthroughableBlock"
                    }
```

```
                else if (Random.Range(0, 3) == 0)//如果不是边界，约33%几率生成阻碍物
                {
                    Instantiate(block, generate_position, Quaternion.Euler(0, 90, 0));//生成横向阻碍
                }
                generate_position += new Vector3(0, 0, interval);//生成下一个阻碍之前更新生成点纵坐标
            }
            generate_position += new Vector3(interval, 0, 0);//生成下一个阻碍之前更新生成点横坐标
            generate_position = new Vector3(generate_position.x, generate_position.y, horizontal_start.position.z);//重置纵坐标
        }
    }
}

/*生成敌人*/
private void InstantiateEnermies()//generate_position为生成点
{
    for (enermy_number = 0; enermy_number != interval; enermy_number++)//一个个生成敌人直到规定数量
    {
        while (true)//循环以下过程
        {
            possible_position = new Vector2(Random.Range(-(min__number / 2), min__number / 2 + 1), Random.Range(-(min__number / 2), min__number / 2 + 1));//随机生成一个可能的位置
            if (!exist_position.Contains(possible_position) && !(possible_position.x == 0 && possible_position.y == 0) && !(Mathf.Abs(possible_position.x) == 1) && !(Mathf.Abs(possible_position.y) == 1))//如果这个位置没有被记录过并且不是生成起点
            {
                exist_position.Add(possible_position);//记录该位置
                break;//跳出循环
            }
        }
        Instantiate(enermy, new Vector3(enermy_start.position.x + (possible_position.x * interval), enermy_start.position.y, enermy_start.position.z + (possible_position.y * interval)), Quaternion.identity);//生成敌人
    }
}

/*生成传送门*/
private void InstantiatePortal()
{
    /*在地图的四个角生成传送门*/
    if (Random.Range(0, 2) == 0)
```

```
        {
            if (Random.Range(0, 2) == 0)
            {
                portal_position = new Vector2(-88, -82.5f);
            }
            else
            {
                portal_position = new Vector2(80, -82.5f);
            }
        }
        else
        {
            if (Random.Range(0, 2) == 0)
            {
                portal_position = new Vector2(-88, 85.5f);
            }
            else
            {
                portal_position = new Vector2(80, 85.5f);
            }
        }
        Instantiate(portal, new Vector3(portal_position.x, 0, portal_position.y),
Quaternion.identity);//生成传送门
    }

    /*生成玩家*/
    private void InstantiatePlayer()
    {
        Instantiate(player);//实例化玩家
    }
}
```