Unity3D 塔防游戏-家园保卫

设计文档

目录

第一章 需求分析.....3
1.1 总体陈述.....3
1.2 开发背景及意义.....3
1.3 可行性分析.....3
1.3.1 市场分析.....3
1.3.2 经济可行性.....3
1.3.3 技术可行性.....4
1.3.4 用户使用可行性.....4
1.4 任务概述.....4
1.4.1 开发目标.....4
1.4.2 用户特点.....4
1.5 需求规定.....4
1.5.1 总体需求.....4
1.5.2 功能需求.....5
1.5.3 性能需求.....5
1.6 功能分析.....5
1.7 用例图.....6
第二章 概要设计.....6
2.1 系统架构设计.....6
2.2 模块划分.....6
第三章 系统实现.....7
3.1 类设计.....7
3.2 功能实现.....7
3.2.1 利用 CuBe 创建基本的地图.....7
3.2.2 创建敌人行走的路.....8
3.2.3 控制玩家的视野（移动和缩放）.....9
3.2.4 创建敌人的移动路径.....9
3.2.5 创建敌人，控制敌人移动.....10
3.2.6 创建敌人的生成器.....12
3.2.7 编写敌人的生成规律.....13
3.2.8 创建三种炮台.....13
3.2.9 创建炮台的选择 UI 界面.....15
3.2.10 创建保存炮台数据的脚本.....15
3.2.11 监听炮台的选择事件.....16
3.2.12 检测鼠标点击在哪个 cube 上.....17

3.2.13 检测是否能够创建炮台.....17
3.2.14 金钱管理，购买和 UI 显示.....17
3.2.15 使用粒子系统显示创建炮台的特效.....18
3.2.16 使用触发器检测进入攻击范围的敌人.....18
3.2.17 控制炮台发射子弹.....19
3.2.18 子弹跟敌人的碰撞处理.....20
3.2.19 添加爆炸特效.....20
3.2.20 控制敌人的爆炸销毁和特效显示.....21
3.2.21 控制炮台指向敌人攻击.....22
3.2.22 添加敌人的血条显示.....22
3.2.23 设计游戏结束的 UI.....23
3.2.24 开发游戏菜单.....24
3.2.25 游戏音效添加.....26
3.2.26 后续增加的关卡设计.....27
3.3 制作过程中遇到的部分值得记录的游戏错误及修改.....28
第四章 总结.....33

第一章 需求分析

1.1 总体陈述

本 C#大作业选题制作塔防游戏，结合个人学习经历，决定使用 Unity3D 游戏引擎 C#编写游戏运行脚本来制作一款单机 3D 塔防游戏，游戏包含三个关卡，玩家在金钱范围内通过消耗金币建造 3 种不同的炮台攻击从出生点产生的敌人来保护我方家园。

1.2 开发背景及意义

Unity3D 游戏制作通过多年的发展已经十分成熟，其主要使用的 C#编程语言也为其各种功能的实现贡献了极大的辅助。通过本课程学习 C#编程语言后，为了个人兴趣和专业知识的结合，开始自学 Unity 的各种基本实现操作，通过 C#的课程学习和 Unity 的课余学习，制作本款塔防游戏，对自己个人学习成果进行整合和考验。

1.3 可行性分析

1.3.1 市场分析

本游戏和市场上的其他塔防游戏的游戏性基本相同，本游戏以 3D 的形式展示游戏画面，相对于 2D 游戏具有更高的视觉冲击效果。现在只制作了 3 个关卡一种难度，但是整个项目具有较高的可塑性，在加入优秀的游戏剧本和游戏背景，增加游戏关卡数目和剧情，便可以进一步成为更加成熟的塔防游戏。可塑性强，可发展性高。

在后面的制作过程可以考虑联网增加数据库，实现多人同时游玩，多玩家共同操作，提高游戏的可玩性。

1.3.2 经济可行性

游戏在进一步的制作和发展后，前期积累一定玩家数量后吗，可以考虑上架到各大游戏平台例如 steam 上进行进一步的收益。

1.3.3 技术可行性

本游戏使用 C#语言来编写游戏运行的脚本，因为是单机游戏所以没有数据库相关，但是在后面的制作过程可以考虑联网增加数据库，实现多人同时游玩，多玩家共同操作，提高游戏的可玩性。

1.3.4 用户使用可行性

该游戏是一个较为传统的 3D 塔防游戏，操作简单，玩家在阅读简洁的游戏提示后即可快速上手进行畅玩。游戏后期的可塑性也很高，关卡的设计和游戏难度的调整都可进行调整，后续内容的增加较为简单。

综上所述，该游戏在技术、经济和社会效益上是完全可行的，可以进行开发。

1.4 任务概述

1.4.1 开发目标

使用 C#编程语言编写游戏运行脚本，包括炮塔建造，子弹控制，敌人属性控制，敌人生成器，炮塔属性控制，地图属性控制，炮塔数据，视角控制，敌人生成波数控制，路径点控制等功能。

1.4.2 用户特点

最终用户的特点：会基本操作电脑，有一定游戏经历，曾经玩过类似的塔防游戏的电脑游戏玩家。

1.5 需求规定

1.5.1 总体需求

1. 创建游戏基本场景，控制玩家视角
2. 创建敌人的行走路径，敌人的生成及移动
3. 创建三种炮塔的创建，控制炮塔的数据类，判断炮台的生成
4. 金钱的显示及管理
5. 控制炮台攻击目标敌人，控制敌人的销毁
6. 敌人的各种属性显示，界面的基本 UI 创建

7. 游戏开始和结束的界面显示
8. 创建游戏的菜单

1.5.2 功能需求

对游戏 UI 界面的要求：

1. 控制游戏的开始和退出
2. 游戏的结束和开始时提示
3. 显示游戏中拥有的金钱数量和炮塔指示

游戏角色的控制：

炮台：

1. 消耗金钱后在限定区域生成创建
2. 当敌人进入攻击范围后进行攻击

敌人：

1. 自动生成，拥有自己的属性
2. 不同波数产生属性和数量都不同的敌人
3. 按照预设的路径移动
4. 血量为 0 时销毁
5. 到达目的地后触发游戏结束的提示，并且后续的敌人停止生成

玩家：

1. 可以通过 wasd 进行视角的移动控制
2. 通过鼠标滚轮可以控制视角的高度

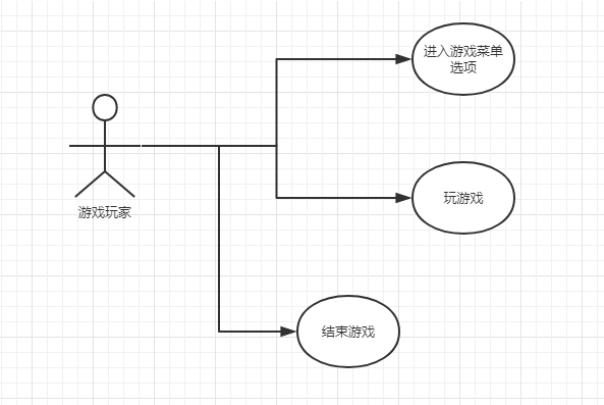
1.5.3 性能需求

1. 游戏整体能够流畅运行
2. 游戏界面设计友好，操作方便、灵活；
3. 易于维护和补充
4. 具有较高的可靠性和内存保护能力，不会对游戏的设备造成影响，不会造成游戏内存的拥挤
5. 具有一定的数据安全性，游戏的各项数据安全，并不会被外界软件应用篡改
6. 3D 画面运行流畅，不会出现明显卡顿

1.6 功能分析

1. 游戏运行后显示菜单，玩家可以通过游戏菜单进入或者退出游戏
2. 游戏关卡开始后，游戏的各个方面功能能够正常运行，玩家用户的操作不会有任何的失灵或者 bug 产生。
3. 游戏胜利或者失败，游戏界面都能显示，并提示下一步操作。

1.7 用例图



第二章 概要设计

2.1 系统架构设计

游戏程序使用者只有玩家，玩家可对游戏进行系统允许范围内的操作。游戏内角色的分类，地图，炮塔，敌人，都拥有属于自己的属性。

2.2 模块划分

炮台：

1. 消耗金钱后在限定区域生成创建
2. 当敌人进入攻击范围后进行攻击

敌人：

1. 自动生成，拥有自己的属性
2. 不同波数产生属性和数量都不同的敌人
3. 按照预设的路径移动
4. 血量为 0 时销毁
5. 到达目的地后触发游戏结束的提示，并且后续的敌人停止生成

玩家：

1. 可以通过 WASD 进行视角的移动控制
2. 通过鼠标滚轮可以控制视角的高度

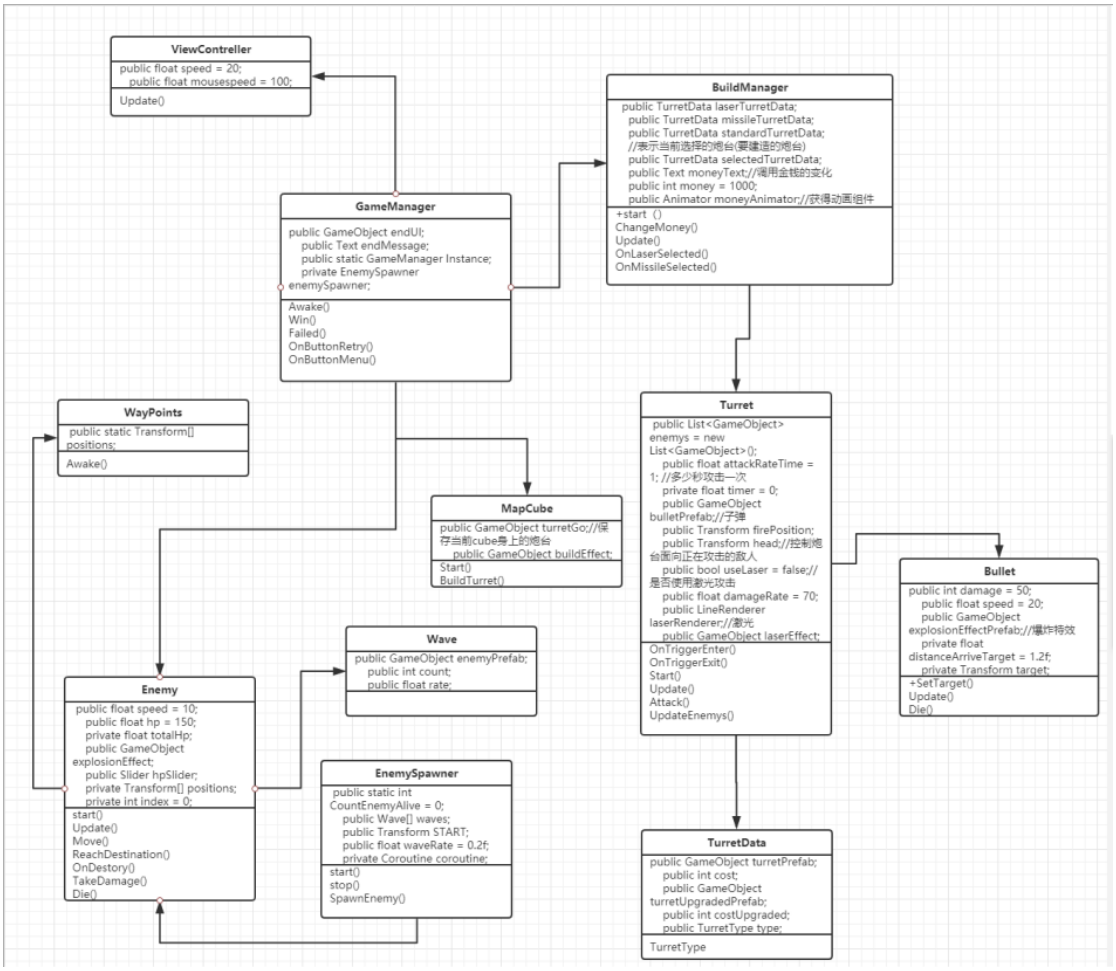
对游戏 UI 界面的要求：

1. 控制游戏的开始和退出

2. 游戏的结束和开始时提示
3. 显示游戏中拥有的金钱数量和炮塔指示

第三章 系统实现

3.1 类设计

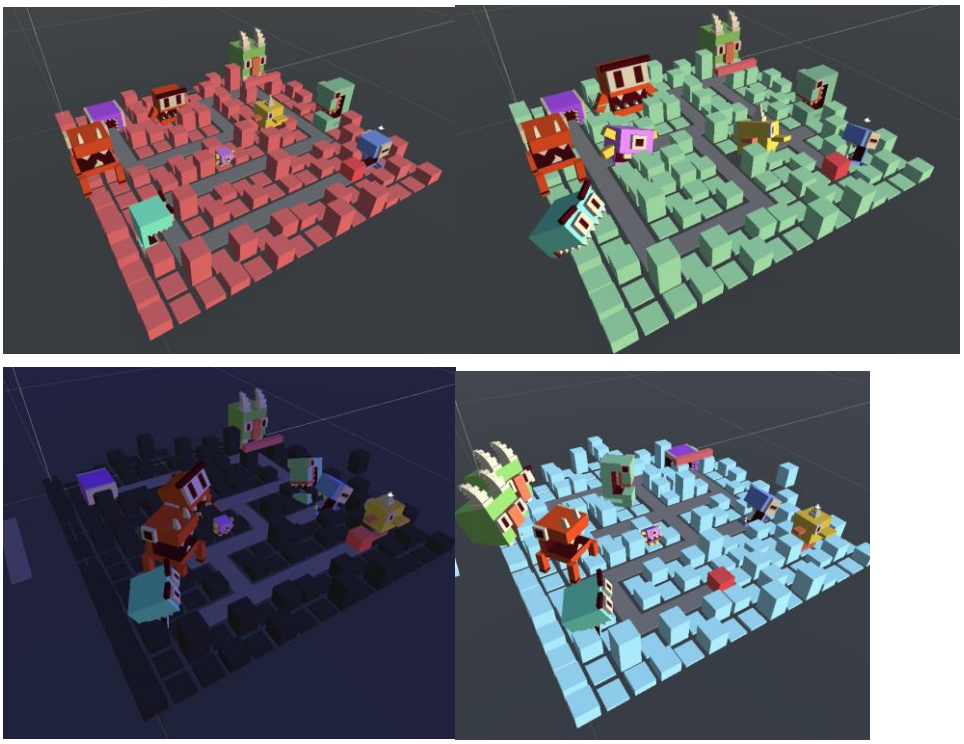


3.2 功能实现

3.2.1 利用 CuBe 创建基本的地图

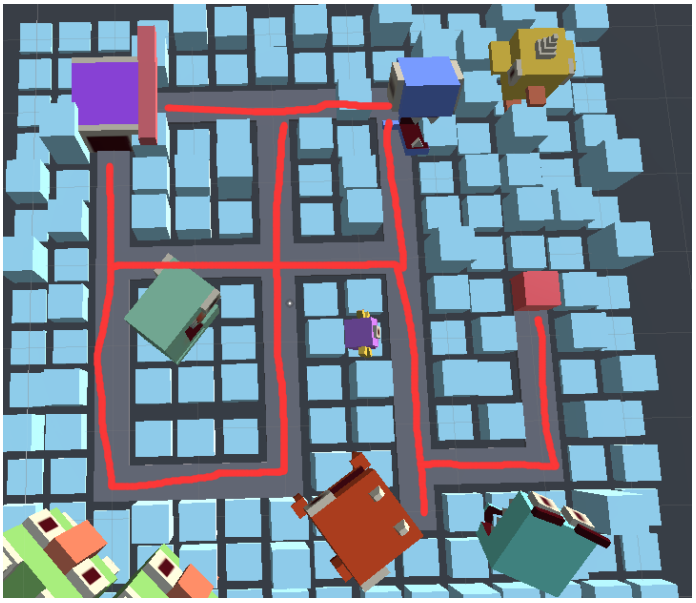
使用最基本的方体 cube 制作一个基本的单一平台作为炮台的搭建点，然后根据游戏的需要创建 30*30 的基本地图，考虑到视觉效果，每个 cube 之间相隔一点距离。

在后期给部分的 cube 提升高度，增加游戏的美观性。在导入从网上下载的怪物模型，夹杂着放在地图中，增加美观。这里一共制作了四个关卡，下面是对对地图的变动截图展示。



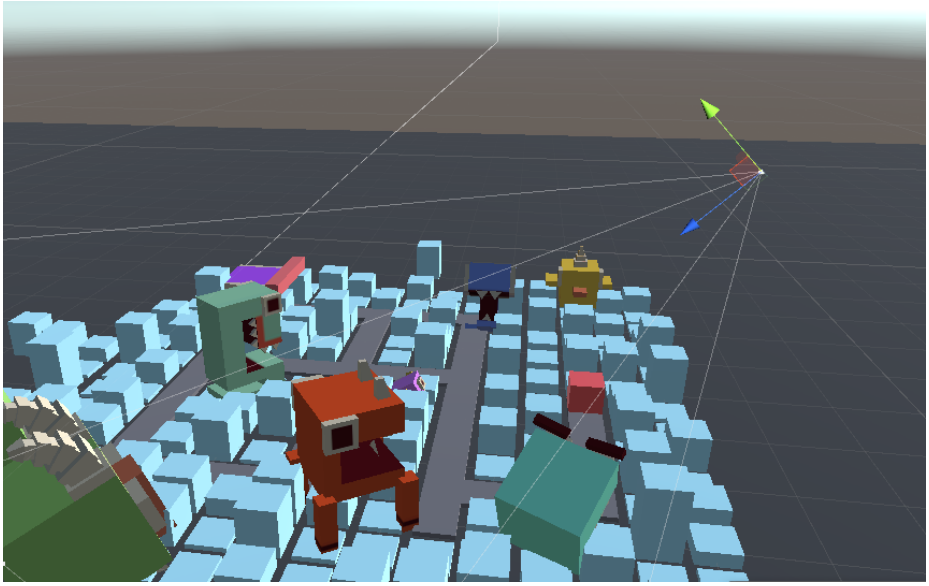
3.2.2 创建敌人行走的路

使用 cube 来制作长条的黑色的路来表示敌人的行走路径，只需要设计以后将原来的地图方块替换成路即可。（下图中红线表示敌人的基本路径）



3.2.3 控制玩家的视野（移动和缩放）

然后是玩家的视角移动，在游戏开始后，玩家使用 WASD 和鼠标的滚轮来实现玩家视角的移动变换。

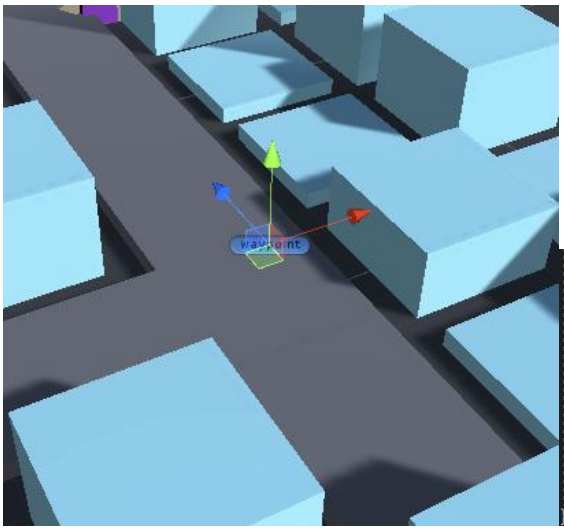


这里我们让 WASD 移动是，摄像机只在一个平面内移动，鼠标滚轮来调整是视角的高度。实现脚本如下

```
0 references
public class ViewContreller : MonoBehaviour
{
    public float speed = 20;
    public float mousespeed = 100;

    // Update is called once per frame
    0 references
    void Update()
    {
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");
        //控制前后左右视角移动
        float mouse = Input.GetAxis("Mouse ScrollWheel");
        transform.Translate(new Vector3(h * speed, mouse * mousespeed, v * speed) * Time.deltaTime, Space.World);
        //世界坐标系
    }
}
```

3.2.4 创建敌人的移动路径

这里使用 unity 中的 waypoint 进行敌人移动的参考，在脚本的作用下，敌人会不断的朝向设置的 waypoint 进行直线移动，我需要做的就是再地图的每个拐角放置一个新的 waypoint 实现敌人的转向即可。

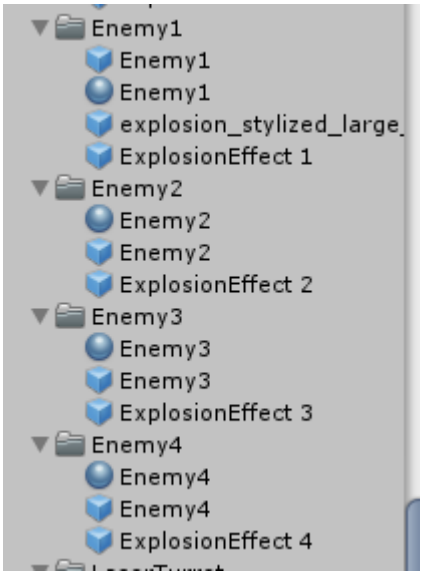


```
public static Transform[] positions;

0 references
void Awake()
{
    positions = new Transform[transform.childCount];
    for (int i = 0; i < positions.Length; i++)
    {
        positions[i] = transform.GetChild(i);
    }
}
```

这里脚本中需要让敌人在到达一个新的 waypoint 以后自动寻找下一个 point 的位置继续移动，所以这里代码使用递归实现具体的效果。实现代码如上图。

3.2.5 创建敌人，控制敌人移动

游戏设计时确定了创建四种不同的敌人，这里使用最基本的球体来代表敌人，创建了四种颜色不同的敌人来区分种类。

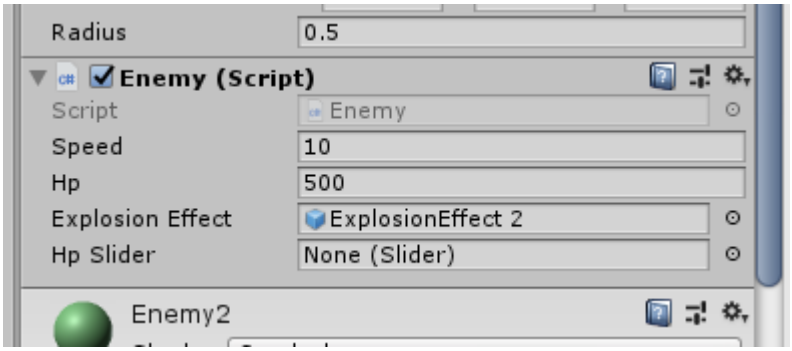


个体创建好以后，为他们每个赋属性值，移动四度和血量，还需要把他们和路径移动的脚本绑定在一起。创建 Enemy 类。

```
public class Enemy : MonoBehaviour
{
    public float speed = 10;
    public float hp = 150;
    private float totalHp;
    public GameObject explosionEffect;
```

```
    public Slider hpSlider;
    private Transform[] positions;
    private int index = 0;
    // Start is called before the first frame update
    void Start()
    {
        //GameObject go = GameObject.Find("Enemy1");
        //Debug.Log(go.name);
        //测试 ：输出Find()对象
        positions = Waypoints.positions;
        totalHp = hp;
        hpSlider = GetComponentInChildren<Slider>();
    }
    // Update is called once per frame
    void Update()
    {
        Move();
    }
    //控制Enemy移动
    void Move()
    {
        if (index > positions.Length - 1) return;
        //到达最大位置（End）则停止
        transform.Translate((positions[index].position - transform.position).normalized
* Time.deltaTime * speed);
        //位置向量，normalized取得单位向量  控制移动方向
        if (Vector3.Distance(positions[index].position, transform.position) < 0.2f)
        {
            index++;
        }
        if (index > positions.Length - 1)
        {
            ReachDestination();
        }
    }
    void ReachDestination()
    {
        GameManager.Instance.Failed();
        GameObject.Destroy(this.gameObject);
        //到达终点销毁敌人对象
    }
    void OnDestroy()
    {
        EnemySpawner.CountEnemyAlive--;
```

```
        //销毁对象
    }
    public void TakeDamage(float damage)
    {
        //控制怪物血量，承伤。
        if (hp <= 0) return;
        hp -= damage;
        hpSlider.value = (float)hp / totalHp;//设置血量，百分比显示
        if (hp <= 0)
        {
            Die();
        }
    }
    void Die()
    {
        //ChangeMoney(-selectedTurretData.cost);
        GameObject effect = GameObject.Instantiate(explosionEffect, transform.position,
transform.rotation);
        Destroy(effect, 1.5f);
        Destroy(this.gameObject);
    }

}
```



（对敌人的属性值的绑定）
里面对他们的各个事件都有了对应的函数功能负责。

3.2.6 创建敌人的生成器

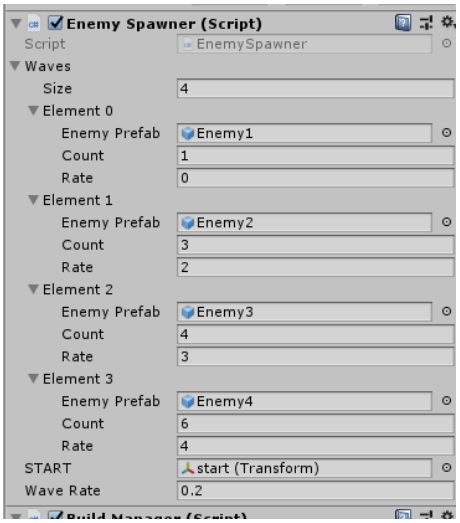
属性配置好以后需要一个统一管理生成的敌人孵化器，也就是 EnemySpawner。
代码如下

```
2 references
IEnumerator SpawnEnemy()
{
    foreach (Wave wave in waves)
    {
        for (int i = 0; i < wave.count; i++)
        {
            GameObject.Instantiate(wave.enemyPrefab, START.position, Quaternion.identity);
            CountEnemyAlive++;
            if (i != wave.count - 1)
                yield return new WaitForSeconds(wave.rate);
        }
        while (CountEnemyAlive > 0)
        {
            yield return 0;
        }
        yield return new WaitForSeconds(waveRate);
    }
    while (CountEnemyAlive > 0)
    {
        yield return 0;
    }
    GameManager.Instance.Win();
```

对敌人生成的波数和之间的时间间隔进行设置。
同时设定他们的生成起点，在我的项目中使用一个空物体来实现生成起点的设置。

3.2.7 编写敌人的生成规律

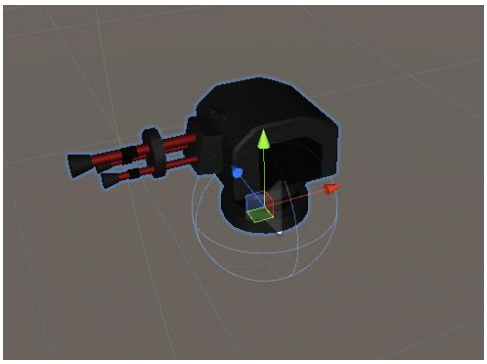
在上一章节中制作的敌人孵化器中编写生成敌人的对应代码，创建适当的类来在 unity 界面中进行简单的控制。

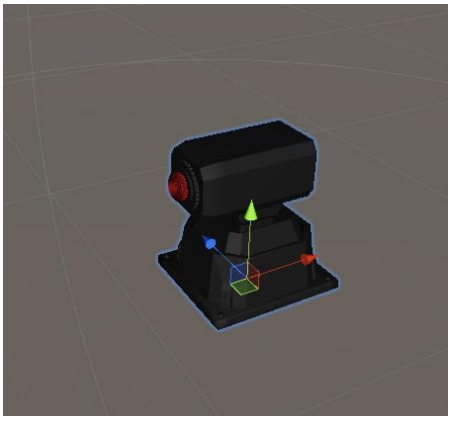


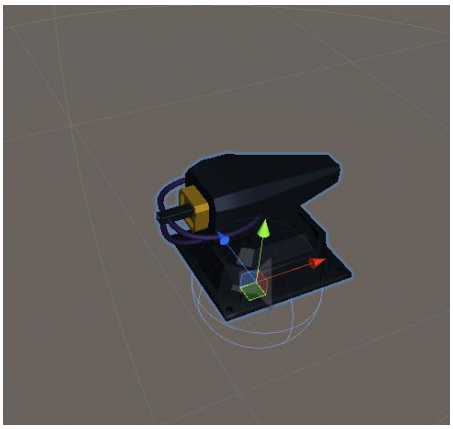
这里显示的是对敌人生成的波数和每一波生成的敌人数量的控制。

3.2.8 创建三种炮台

敌人的属性和移动，触发的事件都创建好以后是对建造炮台的功能编写。
这里首先选定好三种炮台的模型。







在资源网站上找到的三种炮台的 3D 模型，直接导入我的项目中，修改大小后直接使用。

编写 Buildmanager 脚本，控制炮台的建造。

代码省略，这里展示申明的变量

```
public class BuildManager : MonoBehaviour
{
    public TurretData laserTurretData;
    public TurretData missileTurretData;
    public TurretData standardTurretData;

    //表示当前选择的炮台(要建造的炮台)
    public TurretData selectedTurretData;
    public Text moneyText;//调用金钱的变化
    public int money = 1000;
    public Animator moneyAnimator;//获得动画组件

    0 references
    public void Start()
    {
```

创建炮台的脚本还要跟鼠标事件还有界面 UI 相结合，鼠标点击事件出发后，要在地图中实例化一个新的炮台对象，具有与其他炮台相同的属性值，创建炮台消耗的金钱也要在 UI 中进行显示。

3.2.9 创建炮台的选择 UI 界面

使用 2D 模式，以 world 范围创建一个 Ui 显示，其中显示当前拥有的金钱的数目，还要显示三种炮台创建的提示。这里给出截图。

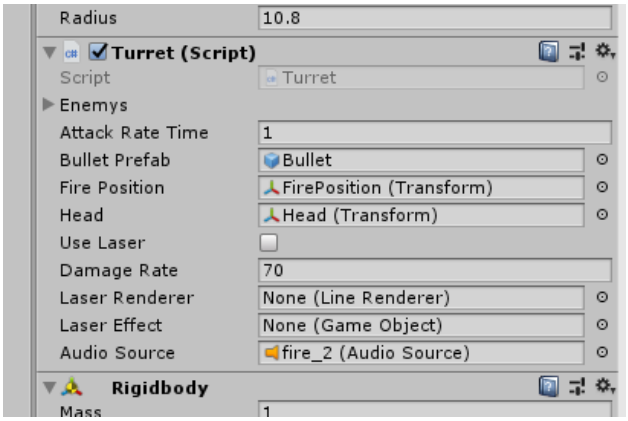
3.2.10 创建保存炮台数据的脚本

每个炮台创建都是在场景中实例化一个新的炮台对象，具有他自己的属性值。这里我使用一个数据脚本来控制所有炮台的属性值

```
5     [System.Serializable]

6     public class TurretData
7     {
8         public GameObject turretPrefab;
9         public int cost;
10        public GameObject turretUpgradedPrefab;
11        public int costUpgraded;
12        public TurretType type;
13    }

14    public enum TurretType
15    {
16        LaserTurret,
17        MissileTurret,
18        StandardTurret
19    }
20
```

通过这个我可以在 unity 的界面中直接对炮台的属性值进行调整，方便了游戏制

作中涉及到的测试问题。

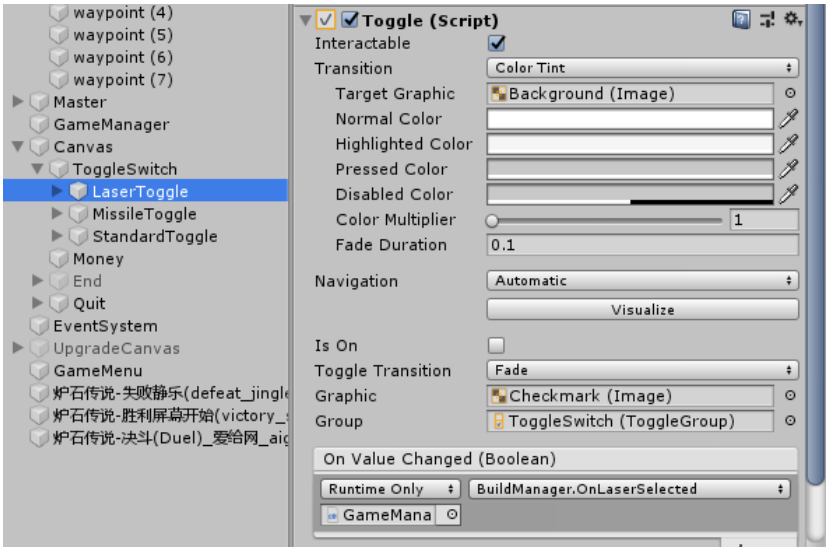


还要给炮台创建一个控制的脚本，中间涉及到了其他板块的代码事件，这里展示申明的变量。

```
public class Turret : MonoBehaviour
{
    // Start is called before the first frame update
    public List<GameObject> enemys = new List<GameObject>();
    public float attackRateTime = 1; //多少秒攻击一次
    private float timer = 0;
    public GameObject bulletPrefab;//子弹
    public Transform firePosition;
    public Transform head;//控制炮台面向正在攻击的敌人
    public bool useLaser = false;//是否使用激光攻击
    public float damageRate = 70;
    public LineRenderer laserRenderer;//激光
    public GameObject laserEffect;
    public AudioSource audioSource;
    //public AudioSource music;
    //public AudioClip fire;
```

3.2.11 监听炮台的选择事件

炮台的点击监听，这里只用了 ToggleSwitch 来控制监听事件，场景中一共放置了三种炮台的选择 UI，只需要监听他的点击事件即可，



3.2.12 检测鼠标点击在哪个 cube 上

为了让炮台创建在我的鼠标点击的 Cube 上，我需要知道当前鼠标点击在哪一块 cube 上，所以需要一个射线监听，通过对鼠标射线的监听和对地图的分层来判断到底点击在哪一个 cube 上。

3.2.13 检测是否能够创建炮台

因为一开始每个人只有 1000 块钱，建造炮台需要消耗对应的数量金钱，所以需要对金钱的数目进行监听，如果金钱能够购买所选择的炮台则建造，如果不够就进行动画提示。

```
if (EventSystem.current.IsPointerOverGameObject() == false)
{
    //开发炮台的建造
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;
    bool isCollider = Physics.Raycast(ray, out hit, 1000, LayerMask.GetMask("MapCube"));
    if (isCollider)
    {
        MapCube mapCube = hit.collider.GetComponent<MapCube>();
        if(selectedTurretData != null && mapCube.turretGo == null)//判断前者条件，防止不选择turret时点击Cube，报空指针
        {
            //可以创建
            if (money > selectedTurretData.cost)
            {
                ChangeMoney(-selectedTurretData.cost);
                mapCube.BuildTurret(selectedTurretData.turretPrefab);
            }
            else
            {
                //TODO 钱不够
                moneyAnimator.SetTrigger("Flicker");
```

这个代码是判断是否还有足够的金钱来建造炮台。不够时调用预先设定好的动画来提示玩家金钱不够了。

3.2.14 金钱管理，购买和 UI 显示

在炮台选择的界面再创建一个新的 money 空对象，为它赋一个脚本控制

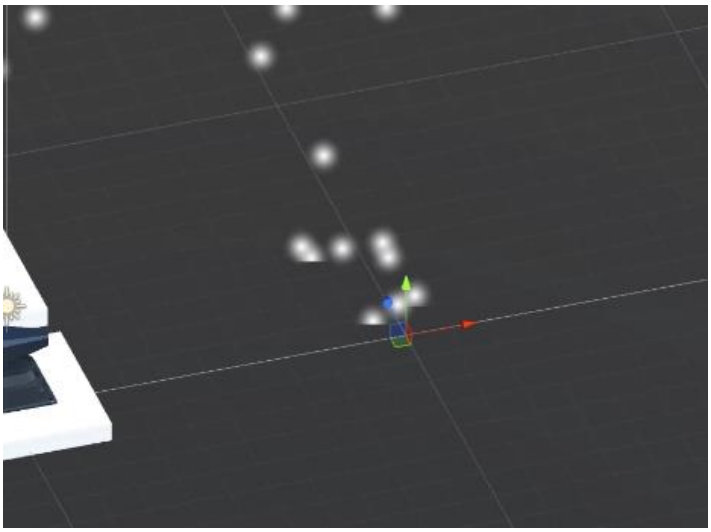
```
1 reference
void ChangeMoney(int change)//金钱的更新
{
    money += change;
    moneyText.text = "$" + money;
}
0 references
```

再炮台创建成功之前更新金钱数量的显示。

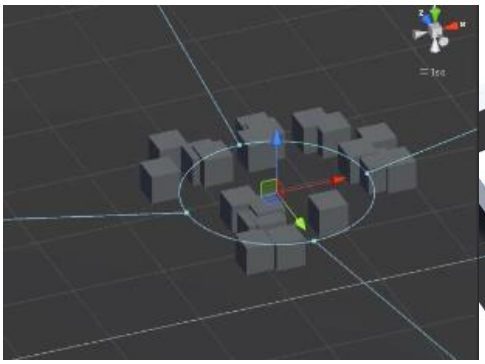
```
//可以创建
if (money > selectedTurretData.cost)
{
    ChangeMoney(-selectedTurretData.cost);
    mapCube.BuildTurret(selectedTurretData.turretPrefab);
}
else
{
    //TODO 钱不够
    moneyAnimator.SetTrigger("Flicker");
}
```

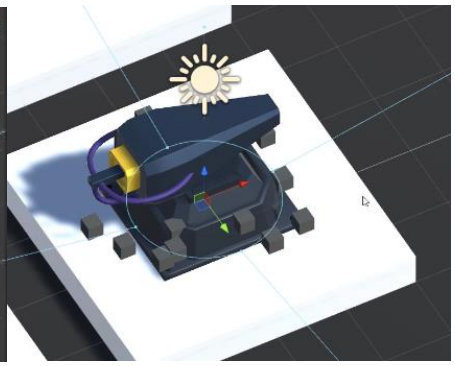
3.2.15 使用粒子系统显示创建炮台的特效

使用 unity 的粒子系统创建一个粒子动画来表示炮台的创建



逐步修改变成我要使用到的显示效果

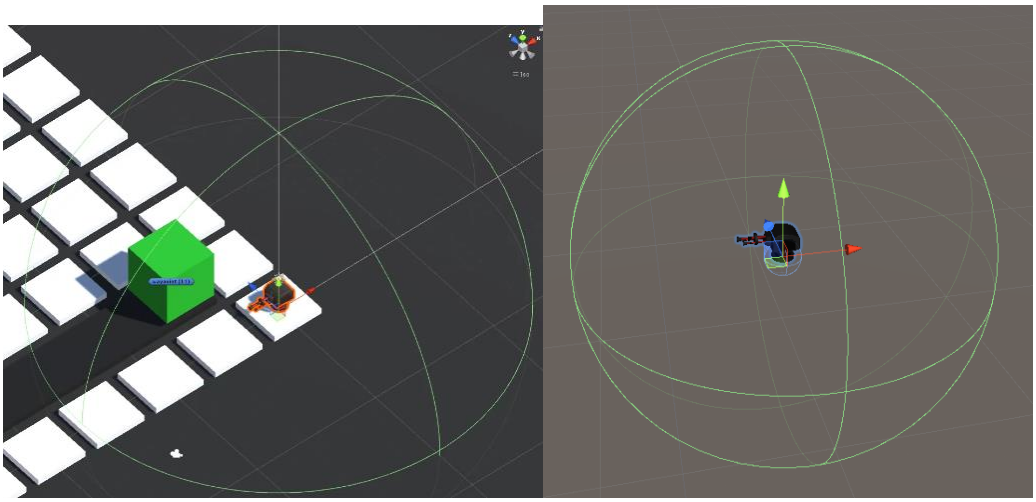




再通过脚本控制，每次创建一个炮台的时候实例化一个粒子动画的对象。

3.2.16 使用触发器检测进入攻击范围的敌人

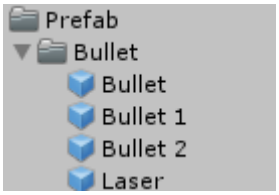
使用一个 Sphere Collider 来创建炮台的攻击范围



当敌人进入该区域时，控制炮台对敌人进行攻击

3.2.17 控制炮台发射子弹

创建 Bullet 类来集中控制所有子弹的属性，并将脚本和子弹实体进行绑定。创建三种不同子弹的模型来实现三种不同炮台的攻击



在脚本中给子弹申明若干共有的属性值，然后在 unity 界面中对每个类型的子弹属性进行相应的修改控制。

每个子弹都有自己的伤害，速度，爆炸特效等属性，然后将这些子弹拖拉到他自己的炮台实例中，使得每种炮台发射出自己对应类型的子弹。

```
public class Bullet : MonoBehaviour
{
    public int damage = 50;

    public float speed = 20;
    public GameObject explosionEffectPrefab;//爆炸特效

    private float distanceArriveTarget = 1.2f;
    private Transform target;

    public void SetTarget(Transform _target)
    {
        this.target = _target;
    }

    void Update()
    {
```

控制子弹朝向敌人移动，触及到敌人时，对敌人造成伤害，最后触发销毁动画的代码如下。

```
void Update()
{
    if (target == null)
    {
        Die();
        return;
    }

    transform.LookAt(target.position);
    transform.Translate(Vector3.forward * speed * Time.deltaTime);

    Vector3 dir = target.position - transform.position;
    if (dir.magnitude < distanceArriveTarget)
    {
        target.GetComponent<Enemy>().TakeDamage(damage);
        Die();
    }
}
```

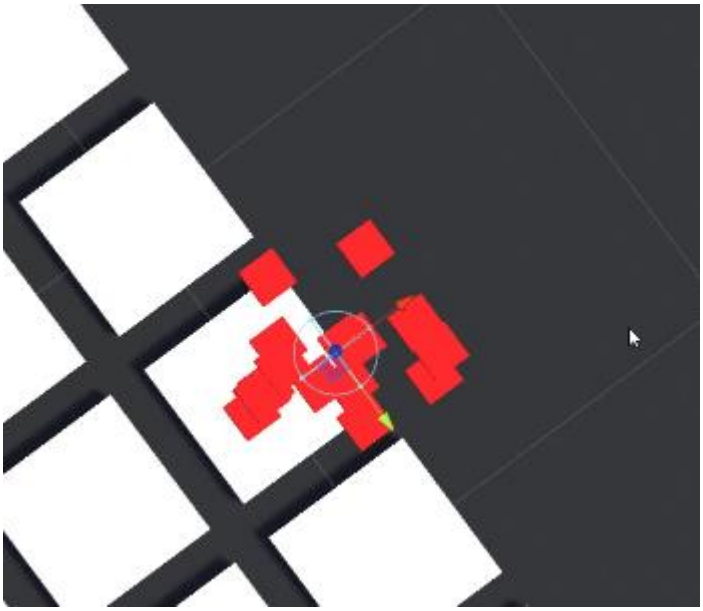
3.2.18 子弹跟敌人的碰撞处理

做子弹和敌人的碰撞检测，碰到时子弹触发自己的销毁代码，敌人则收到对应的伤害。

```
2 references
void Die()
{
    GameObject effect = GameObject.Instantiate(explosionEffectPrefab, transform.position, transform.rotation);
    Destroy(effect, 1);
    Destroy(this.gameObject);
}
```

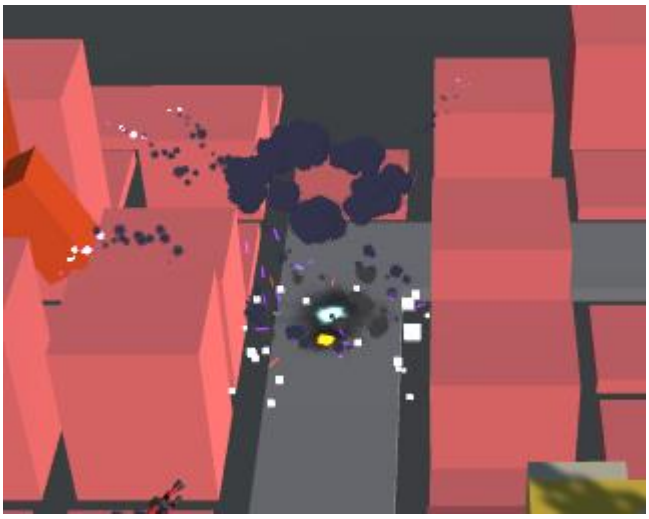
3.2.19 添加爆炸特效

类同炮台创建的特效，进行相同的操作，在创建模型时使用不同的颜色来区分



3.2.20 控制敌人的爆炸销毁和特效显示

脚本控制当敌人被销毁时，触发一个爆炸动画，这里在网上找到的爆炸特效，导入直接使用，效果如下。



3.2.21 控制炮台指向敌人攻击

这里给每个炮台创建一个 head 空对象，将炮台部分的模型放入，然后使用代码控制 head 的 Z 轴指向进入攻击范围内的第一个敌人。

```
public Transform firePosition;
public Transform head;//控制炮台面向正在攻击的敌人
public bool useLaser = false;//是否使用激光攻击
```

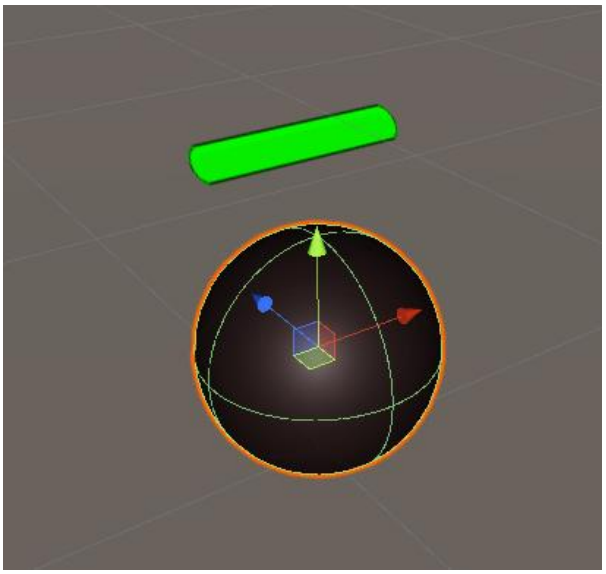
在炮台状态更新的代码中写入炮台转向的判定代码

```
if (enemys.Count > 0 && enemys[0] != null)//控制炮台面向敌人
{
    Vector3 targetPosition = enemys[0].transform.position;
    targetPosition.y = head.position.y;
    head.LookAt(targetPosition);
}//放在前面目的是，炮台先调整方向再进行攻击
```

注意这里要将控制转向的代码放在发射子弹的代码前面，达到炮台先转向再攻击的操作效果。

3.2.22 添加敌人的血条显示

给每个敌人的模型创建一个新的 UI 显示，在上方显示一个绿色的 Slider 显示



并在代码中进行赋值，将这个对象跟敌人的血量情况进行绑定显示。在 Enemy 脚本中进行相关的代码操作。

```
public float speed = 10;
public float hp = 150;
private float totalHp;
public GameObject explosionEffect;
public Slider hpSlider;
private Transform[] positions;
private int index = 0;
// Start is called before the first fr
```

首先申明 Slider 的变量,然后在受到伤害的函数中接入他的函数操作

```
1 reference
public void TakeDamage(float damage)
{
    //控制怪物血量，承伤。
    if (hp <= 0) return;
    hp -= damage;
    hpSlider.value = (float)hp / totalHp;//设置血量，百分比显示
    if (hp <= 0)
    {
        Die();
    }
}
1 reference
```

控制 Slider 按照血量的百分比进行显示

3.2.23 设计游戏结束的 UI

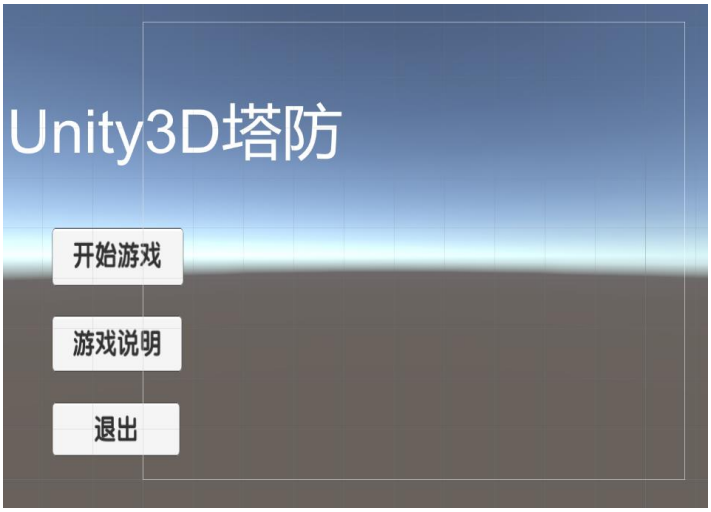
创建游戏结束时的 UI 显示，有一个文本显示和两个按钮显示，都加入移动的动画，游戏结束时从四周移动进入游戏界面，文本通过脚本控制，游戏胜利则显示胜利，失败则显示失败。点击再来一次则重新开始本关游戏，返回菜单则返回到最开始的主菜单界面。

```
    public void Win()
    {
        endUI.SetActive(true);
        audioBackSource.Stop();
        audioWinSource.Play();
        endMessage.text = "胜 利";
    }

    public void Failed()
    {
        enemySpawner.Stop();
        endUI.SetActive(true);
        audioBackSource.Stop();
        audioFailSource.Play();
        endMessage.text = "失 败";
    }
```

```
    }

    0 references
    public void OnButtonRetry()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
    0 references
    public void OnButtonMenu()
    {
        SceneManager.LoadScene(0);
    }
}
```

3.2.24 开发游戏菜单

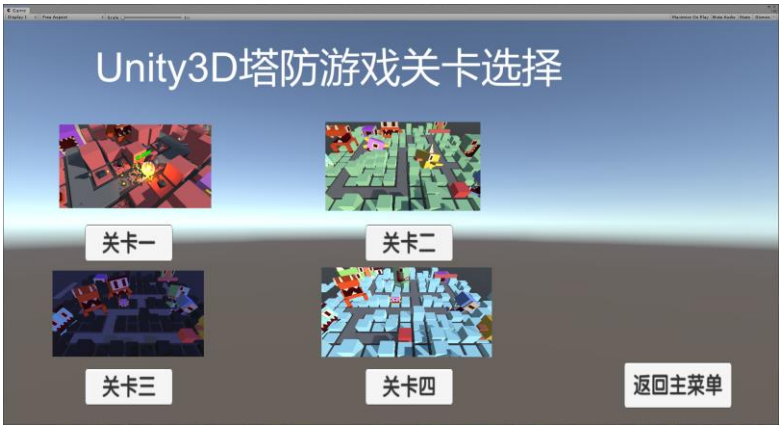
在 unity 中创建一个新的场景，用来显示主菜单



在这里放入提前制作好的视频，在开始这个场景的时候开始自动循环播放游戏视频。

菜单这里可以进入关卡选择，也可以参看游戏帮助

下面时游戏选择界面



然后是游戏说明界面

一共三个游戏场景

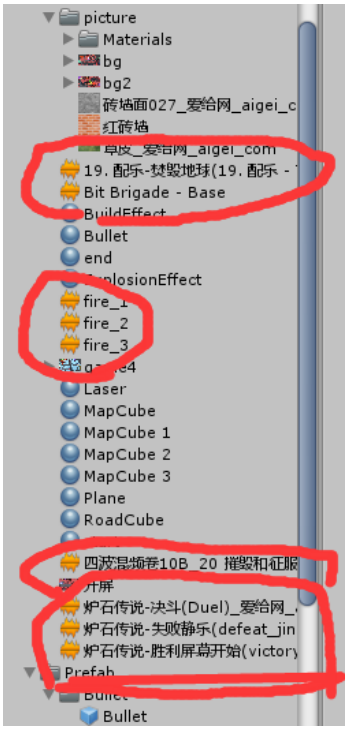
通过脚本控制场景切换

```
ublic class GameMenu : MonoBehaviour

    // Start is called before the first frame update
    0 references
    public void OnSelectGame()
    {
        SceneManager.LoadScene(3);
    }
    0 references
    public void OnStartGame1()
    {
        SceneManager.LoadScene(1);
    }
    0 references
    public void OnStartGame2()
    {
        SceneManager.LoadScene(4);
    }
    0 references
    public void OnStartGame3()
    {
        SceneManager.LoadScene(5);
    }
    0 references
    public void OnStartGame4()
    {
        SceneManager.LoadScene(6);
    }
    0 references
    public void OnExitGame()
    {
if UNITY_EDITOR
```

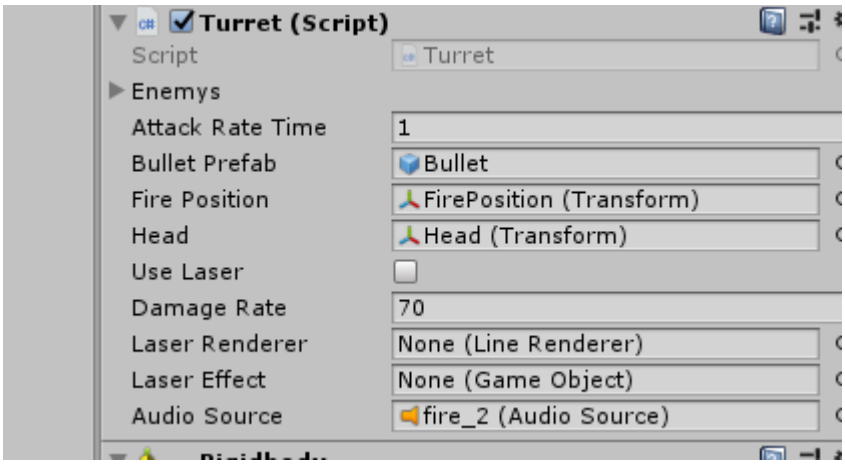
每个场景的编号在 buildsetting 中进行调整。然后通过 LoadScene（）函数进行跳转即可。

3.2.25 游戏音效添加

项目中使用到的游戏音效，主要是每个关卡的背景音乐，三种炮台的攻击音效，游戏成功结束两种不同的提示音，菜单界面的背景音乐。

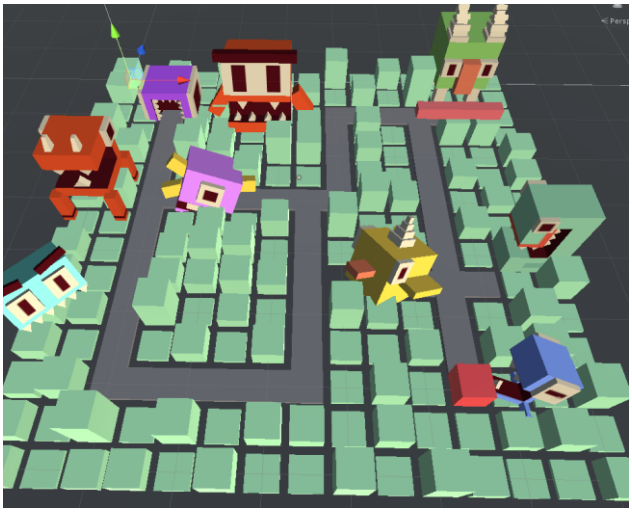


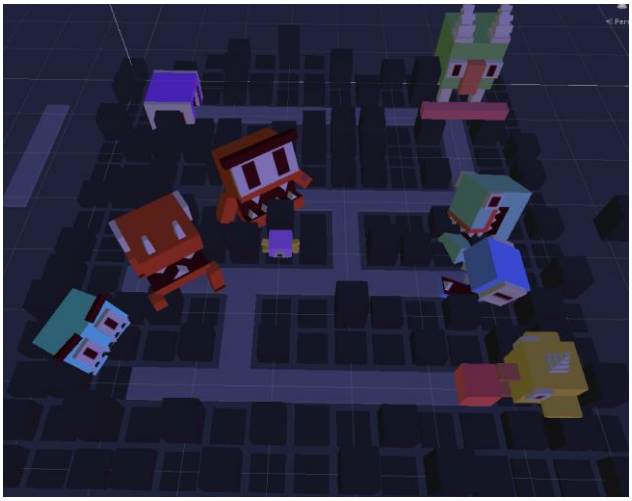
项目中使用到的音效都是网上下载的。
每个炮台的开炮音效都在 Turret 中进行申明使用。

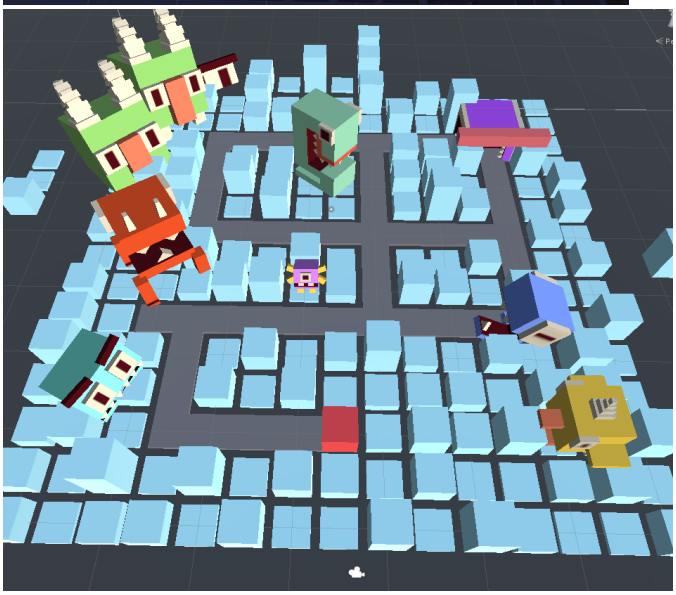


3.2.26 后续增加的关卡设计

后来只需要对原来的关卡地图进行修改，颜色改变一下即可。







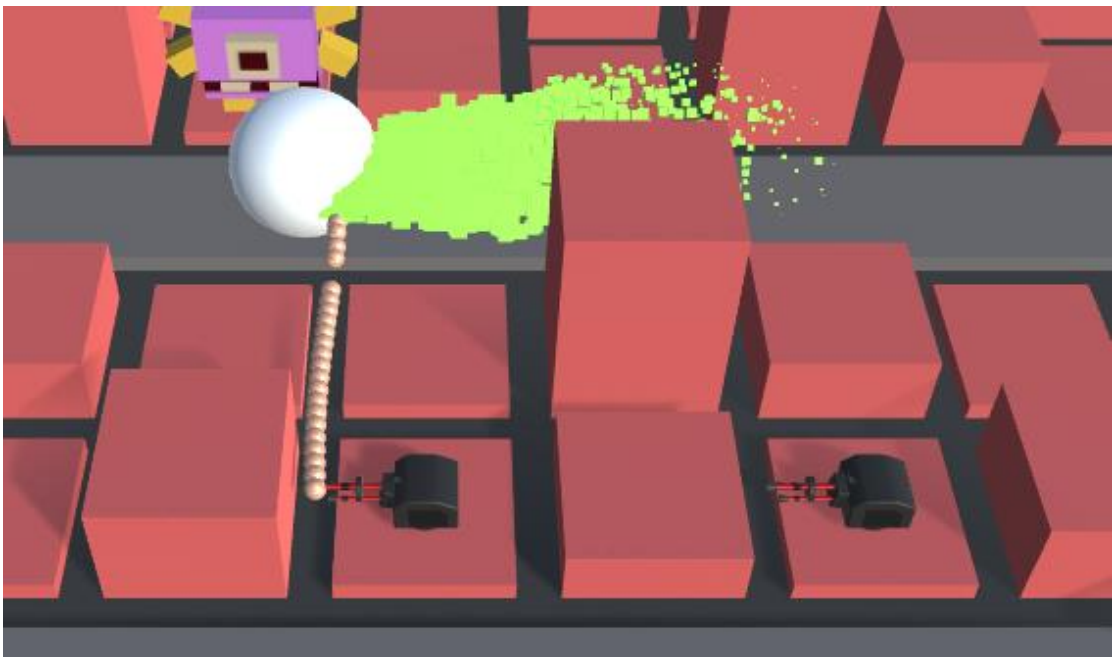
之后又制作的三个关卡，本游戏一共四个关卡。

3.3 制作过程中遇到的部分值得记录的游戏错误及修改

进行过程中遇到的 bug 及解决办法：

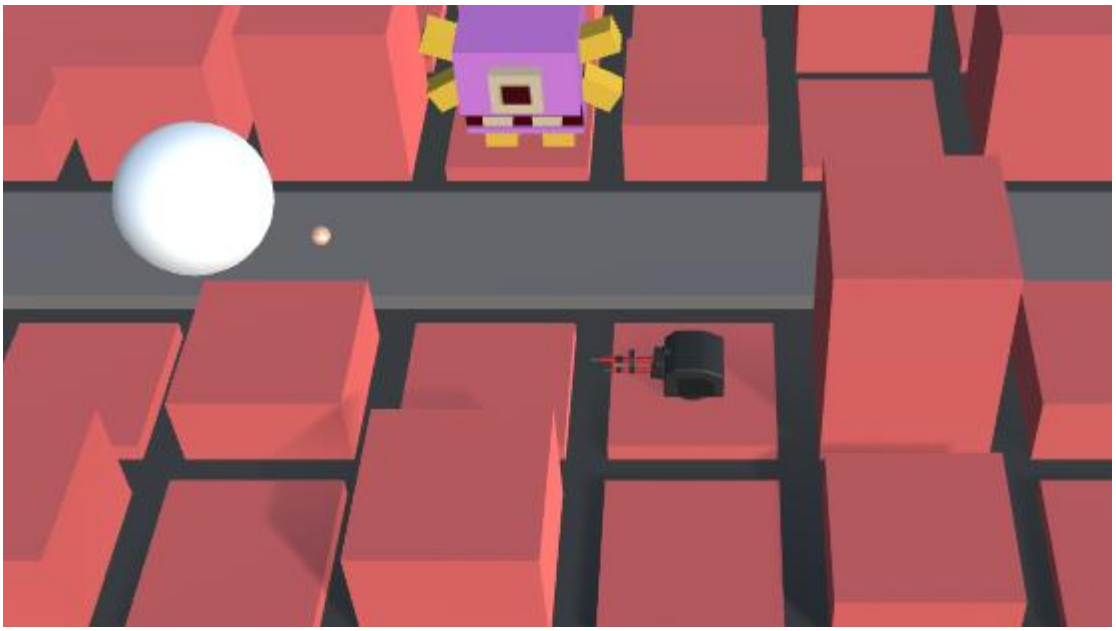
问题一：

Turret.cs 中 39 行设置子弹发射的倒计时时，使用 timer -= Time.deltaTime;导致发射的子弹一连串，而不是前面一颗消除后发射第二颗。



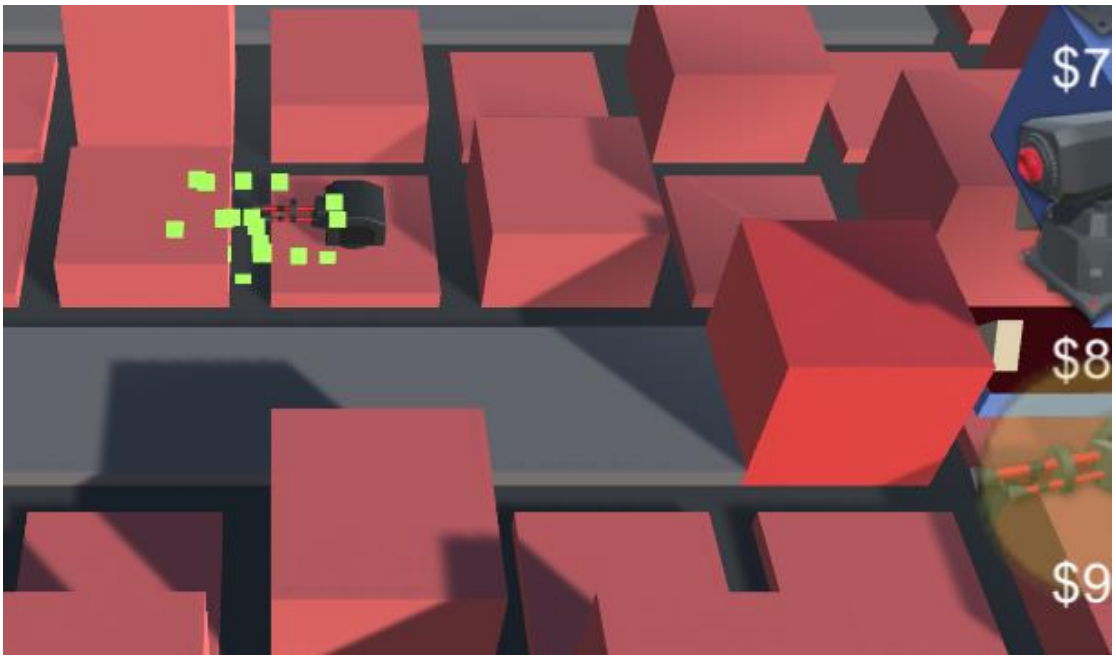
修改办法：

改为 timer =0;直接重置倒计时。



问题二：

已经放置的炮台，在目标物体已经到达终点后，终点也在攻击范围内，炮台仍然会发射子弹，因为没有目标物而直接在炮口爆炸并消除。



解决办法：

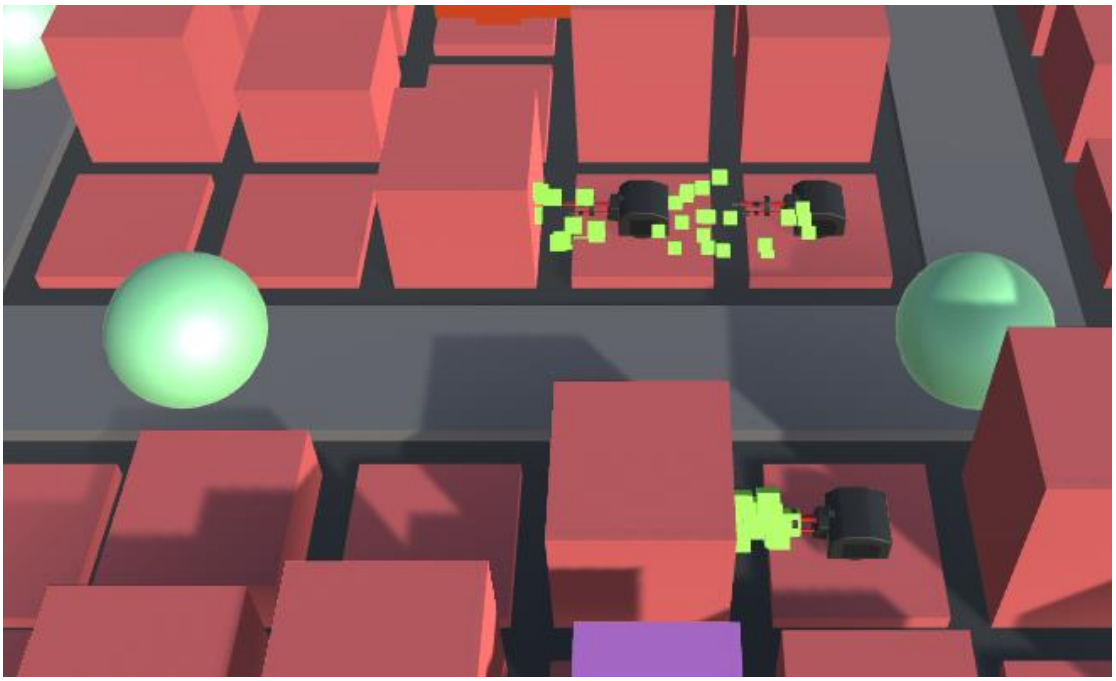
在 Turret.cs 的 Attack 函数中加入对攻击范围内目标物是否存在的判断

```
void Attack()
{
    if (enemys[0] == null)
    {
        UpdateEnemys();
    }
    if (enemys.Count > 0)
    {
        GameObject bullet = GameObject.Instantiate(bulletPrefab, firePosition.position, firePosition.rotation);
        bullet.GetComponent<Bullet>().SetTarget(enemys[0].transform);
    }
    else
    {
        timer = attackRateTime;//没有敌人，重置时间并等待新的敌人
    }
}
```

成功消除该问题。

问题三：

炮台攻击消灭敌人后，会继续开炮，但是子弹在炮口处爆炸，并且不会攻击新的敌人。



在 Turret.cs 的 Attack 函数中加入对攻击范围内目标物是否存在的判断，同时增加 UpdateEnemys 函数，对攻击范围内的敌人进行为空判断。

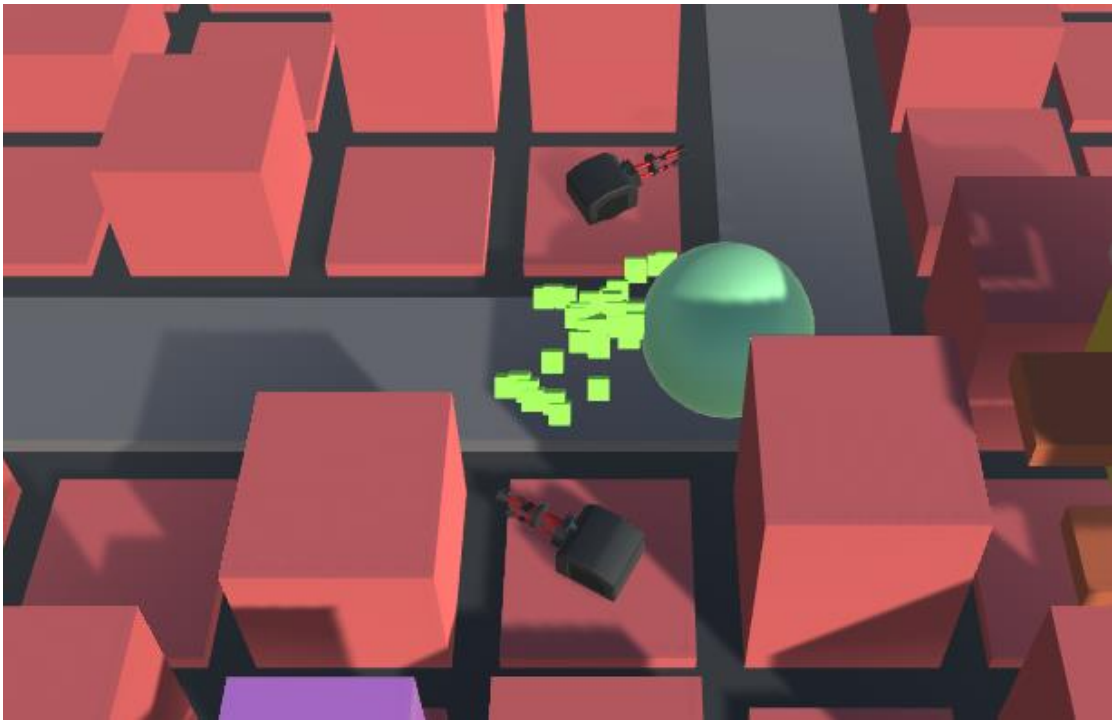
```
1 reference
void UpdateEnemys()
{
    //enemys.RemoveAll(null);
    //需要找到哪些元素时空的，才能对敌人的集合进行清空。
    List<int> emptyIndex = new List<int>();//保存所有的空元素
    for (int index = 0; index < enemys.Count; index++)
    {
        if (enemys[index] == null)
        {
            emptyIndex.Add(index);
        }
    }

    for (int i = 0; i < emptyIndex.Count; i++)
    {
        enemys.RemoveAt(emptyIndex[i] - i);
    }
}
```

成功消除问题。

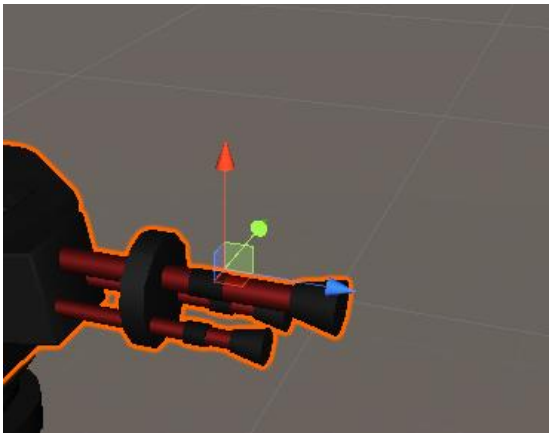
问题四：

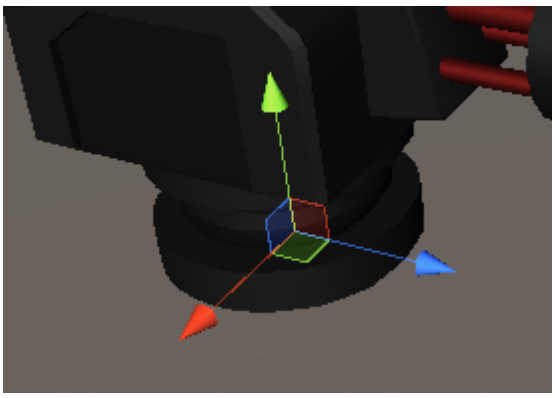
控制炮台攻击时面对被攻击者，在代码中增加 head 的同时，还要在场景中加入炮台 head 的判断，在炮台的模型中创建空物体命名为 Head，将炮台的头部放入

其中，代替为头部的判断。但是实际应用中，炮台并不能面对攻击物体。



解决办法：
这里实际是将 Head 的 Z 轴来面对攻击物体，所以这里只需要将炮台头部的 Z 轴与 Head 的 Z 轴统一方向即可。





第四章 总结

开学以后经常玩塔防游戏，所以也想自己制作一款.

最开始的选题过程，基本的游戏机制就按传统塔防游戏那样制作，至于控制脚本当然是基于游戏功能，当时最大的问题就是游戏模型问题，自己制作需要花费太多的时间，所以就在网上浏览下载适合游戏主题的模型备用，精选一部分可用的游戏模型后才开始考虑游戏场景的事，毕竟相对于可以自己搭建的游戏场景，游戏模型的考虑优先级就在考虑制作的最高级。

前期相当的准备后，游戏的实体制作就轻松许多，想到好的游戏场景就搭建，好的游戏机制就用脚本实现，在模型和脚本的结合的过程中则遇到很多的问题，这些都子啊上文列举说过了，这里不赘述，总之这里消耗的时间大概占据实际制作时间的 50%到 60%，大部分时间在改 BUG，见笑。

模型场景的搭建在之前的项目都接触到了许多，这次游戏的制作主要是学习脚本在游戏机制中的实际作用，还有控制模型做多种动作上，还有插入游戏音效和开屏的游戏视频。过程中看了很多网课和教程，项目完成的时候也学到了很多新的知识，能在以后的项目操作中得到应用，这一点是很值得高兴的。

游戏的整体在实现之后，后头看看制作过程，更能体会到平时玩到的好的游戏制作不易，很多地方不是能靠代码脚本就能机器实现的，还是需要自己手动花费大量的时间来进行实际操作检测，比如游戏难度的设计，炮台的攻击能力，敌人的血量和移动速度，这些都需要通过实际的游戏体验才能进行合理的修改和设置。