# 智能巡逻兵

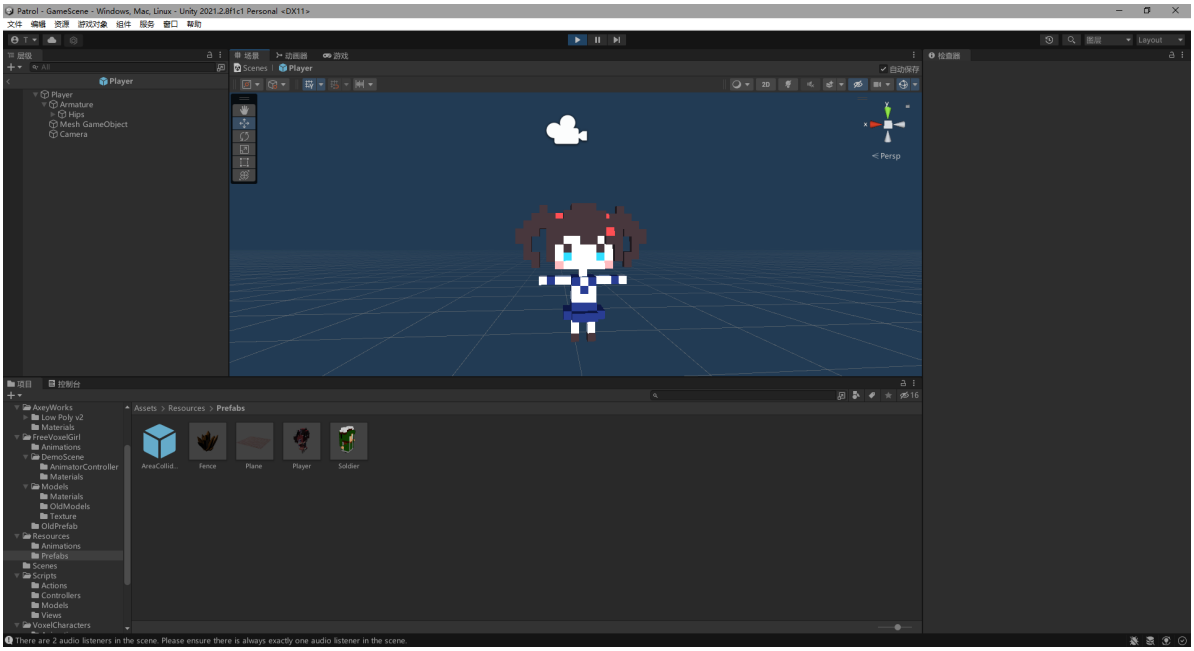
□ 游戏设计要求：
• 创建一个地图和若干巡逻兵(使用动画)；
• 每个巡逻兵走一个3~5个边的凸多边型，位置数据是相对地址。即每次确定下一个目标位置，用自己当前位置为原点计算；
• 巡逻兵碰撞到障碍物，则会自动选下一个点为目标；
• 巡逻兵在设定范围内感知到玩家，会自动追击玩家；
• 失去玩家目标后，继续巡逻；
• 计分：玩家每次甩掉一个巡逻兵计一分，与巡逻兵碰撞游戏结束；
□ 程序设计要求：
• 必须使用订阅与发布模式传消息
• 工厂模式生产巡逻兵
□ 提示1：生成 3~5个边的凸多边型
• 随机生成矩形
• 在矩形每个边上随机找点，可得到 3 - 4 的凸多边型

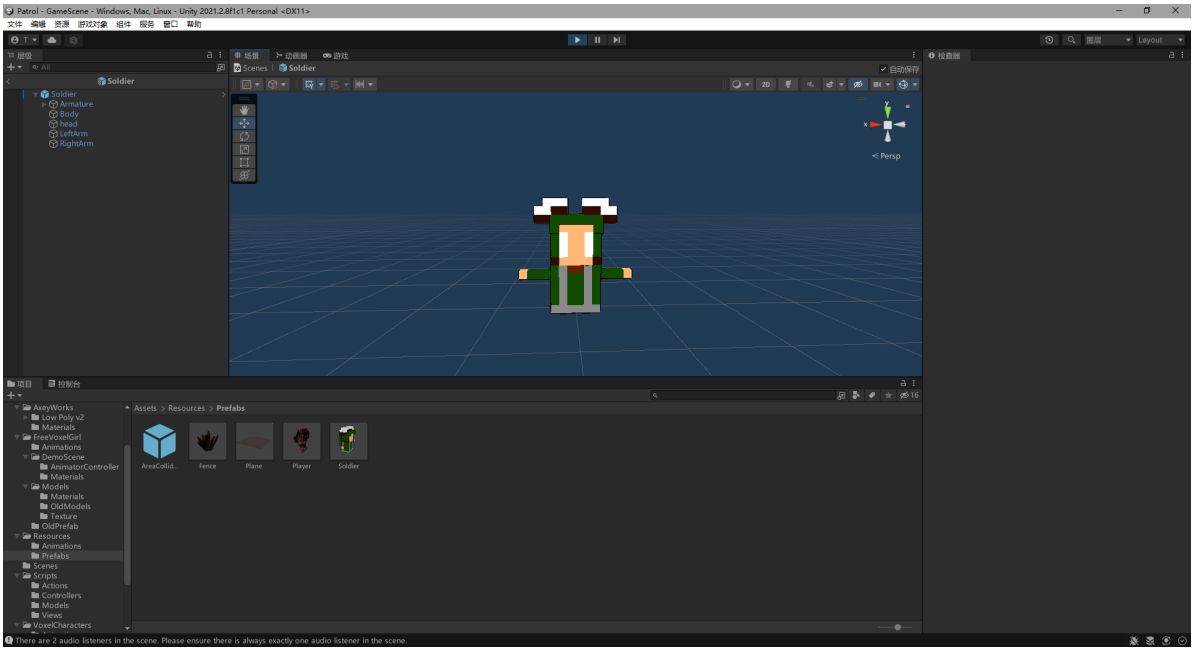
---

什么是订阅发布者模式？简单的说，比如我看见有人在公交车上偷钱包，于是大叫一声“有人偷钱包”(发送消息)，车上的人听到（接收到消息）后做出相应的反应，比如看看自己的钱包什么的。其实就两个步骤，注册消息与发送消息。

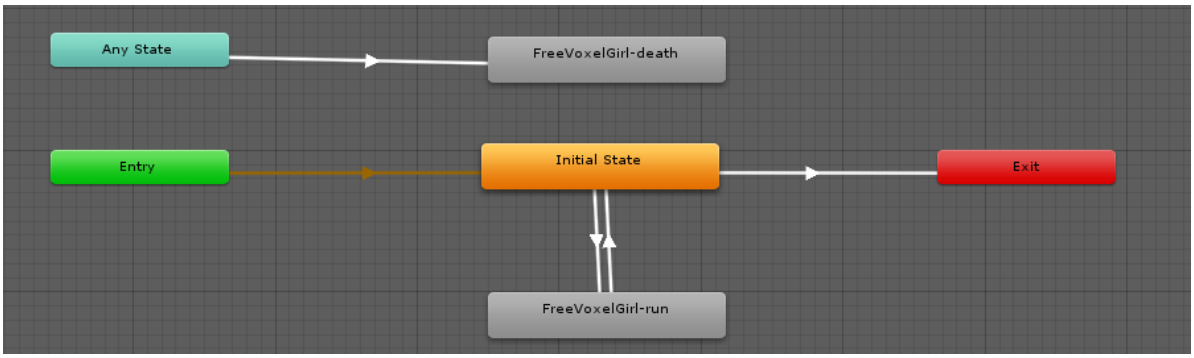
为了适应项目需要，写了一个通用订阅发布者模式的通用模块，有了这样一个模块，项目里面其他模块之间的耦合性也将大大降低。

使用 FreeVoxelGirl 素材包来构建 `Player` 玩家人物模型和 `Soilder` 人物模型：

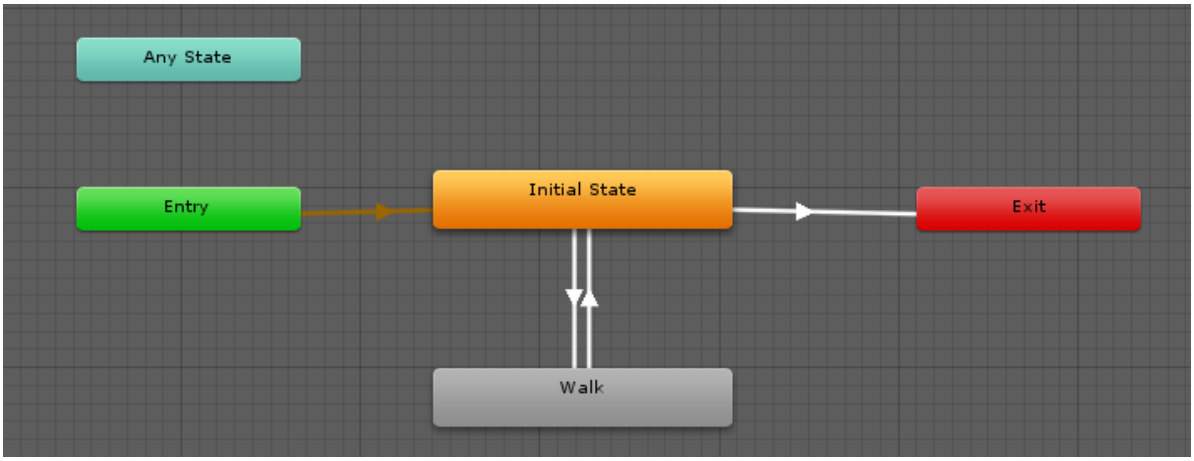




为Player其添加了 `Collider` 和 `Animator` 组件，实现与其它游戏对象的碰撞检测、动画效果。以下是 `Player` 游戏对象的 `Inspector` 栏设置：

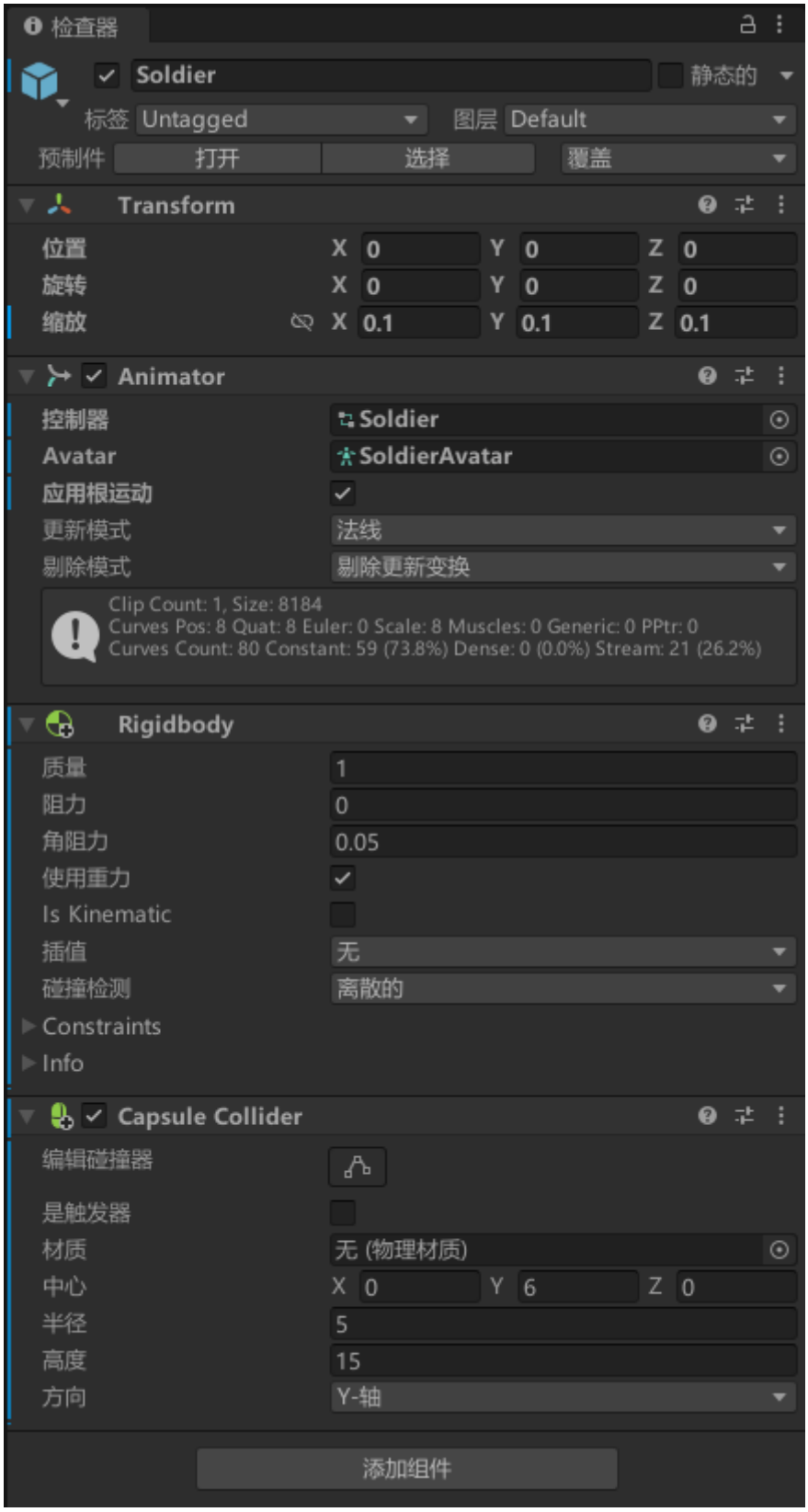



和 `Player` 类似，`Soldier` 也需要进行动画的设置，具体如下图所示。




Player和Soldier属性设置：


检查器
Player
静态的
标签 Player
图层 Default
Transform
位置 X 0 Y 0.3 Z 0
旋转 X 0 Y 0 Z 0
缩放 X 1 Y 1 Z 1
Animator
控制器 Player
Avatar FreeVoxelGirlBlackhairAvatar
应用根运动
更新模式 法线
剔除模式 剔除更新变换
Clip Count: 3, Size: 70136
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 390 Generic: 0 PPtr: 0
Curves Count: 390 Constant: 205 (52.6%) Dense: 135 (34.6%) Stream: 50 (12.8%)
Rigidbody
质量 1
阻力 0
角阻力 0.05
使用重力
Is Kinematic
插值 无
碰撞检测 离散的
Constraints
Info
Capsule Collider
编辑碰撞器
是触发器
材质 无 (物理材质)
中心 X 0 Y 0.65 Z 0
半径 0.5
高度 1.3
方向 Y-轴
添加组件




生成地图

```
// 地图平面预制。
private static GameObject planePrefab = Resources.Load<GameObject>
("Prefabs/Plane");
// 篱笆预制。
private static GameObject fencePrefab = Resources.Load<GameObject>
("Prefabs/Fence");
// 区域Collider预制。
```

```
6  private static GameObject areaColliderPrefab = Resources.Load<GameObject>
   ("Prefabs/AreaCollider");
7  // 地图 9 个区域的中心点位置。
8  public static Vector3[] center = new Vector3[] { new Vector3(-10, 0, -10),
   new Vector3(0, 0, -10), new Vector3(10, 0, -10), new Vector3(-10, 0, 0), new
   Vector3(0, 0, 0), new Vector3(10, 0, 0), new Vector3(-10, 0, 10), new
   Vector3(0, 0, 10), new Vector3(10, 0, 10) };
9
10 // 构造地图边界篱笆。
11 public static void LoadBoundaries()
12 {
13     for (int i = 0; i < 12; ++i)
14     {
15         GameObject fence = Instantiate(fencePrefab);
16         fence.transform.position = new Vector3(-12.5f + 2.5f * i, 0, -15);
17     }
18     for (int i = 0; i < 12; ++i)
19     {
20         GameObject fence = Instantiate(fencePrefab);
21         fence.transform.position = new Vector3(-12.5f + 2.5f * i, 0, 15);
22     }
23     for (int i = 0; i < 12; ++i)
24     {
25         GameObject fence = Instantiate(fencePrefab);
26         fence.transform.rotation = Quaternion.AngleAxis(90, Vector3.up);
27         fence.transform.position = new Vector3(-15, 0, -15 + 2.5f * i);
28     }
29     for (int i = 0; i < 12; ++i)
30     {
31         GameObject fence = Instantiate(fencePrefab);
32         fence.transform.rotation = Quaternion.AngleAxis(90, Vector3.up);
33         fence.transform.position = new Vector3(15, 0, -15 + 2.5f * i);
34     }
35 }
36
37 // 构造内部篱笆。
38 public static void LoadFences()
39 {
40     //  为 0 表示通道，为 1 表示篱笆。
41     var row = new int[2, 12] { { 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1 }, { 0,
   1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0 } };
42     var col = new int[2, 12] { { 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1 }, { 0,
   1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0 } };
43     for (int i = 0; i < 2; ++i)
44     {
45         for (int j = 0; j < 12; ++j)
46         {
47             if (row[i, j] == 1)
48             {
49                 GameObject fence = Instantiate(fencePrefab);
50                 fence.transform.position = new Vector3(-12.5f + 2.5f * j, 0,
   -5 + 10 * i);
51             }
52         }
53     }
54     for (int i = 0; i < 2; ++i)
55     {
56         for (int j = 0; j < 12; ++j)
```

```
        {
            if (col[i, j] == 1)
            {
                GameObject fence = Instantiate(fencePrefab);
                fence.transform.rotation = Quaternion.AngleAxis(90,
Vector3.up);
                fence.transform.position = new Vector3(-5 + 10 * i, 0, -15 +
2.5f * j);
            }
        }
    }
}
```

为了探测玩家的所在区域号，我们需要为 9 个区域分别添加 `AreaCollider` 检测脚本。

```
// 构造区域Collider。
public static void LoadAreaColliders()
{
    for (int i = 0; i < 9; ++i)
    {
        GameObject collider = Instantiate(areaColliderPrefab);
        collider.name = "AreaCollider" + i;
        collider.transform.position = center[i];
        // 添加区域检测脚本。
        collider.AddComponent<AreaCollider>().area = i;
    }
}
```

在区域检测中，我们使用了**订阅与发布模式**，对游戏逻辑进行了解耦。在 `GameController` 中，我们实现了 `OnPlayerEnterArea` 方法用于订阅**玩家进入区域**的事件，该方法在 `OnTriggerEnter` 触发时被调用，即玩家摆脱一位巡逻兵，进入新区域时。在 `GameController` 的 `Awake` 函数中，我们注册了对应事件的处理函数。

```
public class GameEventManager
{
    // Singleton instance.
    private static GameEventManager instance;

    public delegate void OnPlayerEnterArea(int area);
    public static event OnPlayerEnterArea onPlayerEnterArea;

    public delegate void OnSoldierCollideWithPlayer();
    public static event OnSoldierCollideWithPlayer
onSoldierCollideWithPlayer;

    // 使用单例模式。
    public static GameEventManager GetInstance()
    {
        return instance ?? (instance = new GameEventManager());
    }

    // 当玩家进入区域。
    public void PlayerEnterArea(int area)
    {
        onPlayerEnterArea?.Invoke(area);
    }
```

```
23
24      // 当巡逻兵与玩家碰撞。
25      public void SoldierCollideWithPlayer()
26      {
27          onSoldierCollideWithPlayer?.Invoke();
28      }
29  }
30
31  // 设置游戏事件及其处理函数。
32  GameEventManager.onPlayerEnterArea += OnPlayerEnterArea;
33  GameEventManager.onSoldierCollideWithPlayer += OnSoldierCollideWithPlayer;
```

碰撞检测

我们在生成巡逻兵实例时，为其添加了 `SoldierCollider` 碰撞检测脚本，用于判定游戏的胜负：当巡逻兵与玩家碰撞时，游戏失败。

```
1   public class SoldierCollider : MonoBehaviour
2   {
3       // 当巡逻兵与玩家碰撞时。
4   private void OnSoldierCollideWithPlayer()
5   {
6       view.state = model.state = GameState.LOSE;
7       // 设置玩家的“死亡”动画。
8       player.GetComponent<Animator>().SetTrigger("isDead");
9       player.GetComponent<Rigidbody>().isKinematic = true;
10      soldiers[currentArea].GetComponent<Soldier>().isFollowing = false;
11      // 取消所有巡逻兵的动画。
12      actionManager.Stop();
13      for (int i = 0; i < 9; ++i)
14      {
15          soldiers[i].GetComponent<Animator>().SetBool("isRunning", false);
16      }
17  }
18  }
```

动作分离

在游戏中，巡逻兵有两种动作可以展现：**自主巡逻**和**追随玩家**。为此，我们使用了**动作分离**的技术，具体代码参照 `GameActionManager` 类的实现。

在自主巡逻中，确定巡逻目的地，是一个核心问题。我们在 `GetGoAroundTarget` 方法中，通过随机生成目的地，并对其进行合法性判断，确定巡逻目的地。我们在 `MoveToAction` 类中实现了巡逻兵的自主巡逻。

```
1   // 存储自主巡逻动作。
2   Dictionary<int, MoveToAction> moveToActions = new Dictionary<int,
    MoveToAction>();
3
4   // 巡逻兵自主巡逻。
5   public void GoAround(GameObject patrol)
6   {
7       var area = patrol.GetComponent<Soldier>().area;
8       // 防止重入。
9       if (moveToActions.ContainsKey(area))
10      {
11          return;
```

```
    }
    // 计算下一巡逻目的地。
    var target = GetGoAroundTarget(patrol);
    MoveToAction action = MoveToAction.GetAction(patrol, this, target, 1.5f,
area);
    moveToActions.Add(area, action);
    AddAction(action);
}

// 计算下一巡逻目的地。
private Vector3 GetGoAroundTarget(GameObject patrol)
{
    Vector3 pos = patrol.transform.position;
    var area = patrol.GetComponent<Soldier>().area;
    // 计算当前区域的边界。
    float x_down = -15 + (area % 3) * 10;
    float x_up = x_down + 10;
    float z_down = -15 + (area / 3) * 10;
    float z_up = z_down + 10;
    // 随机生成运动。
    var move = new Vector3(Random.Range(-3, 3), 0, Random.Range(-3, 3));
    var next = pos + move;
    int tryCount = 0;
    // 边界判断。
    while (!(next.x > x_down + 0.1f && next.x < x_up - 0.1f && next.z >
z_down + 0.1f && next.z < z_up - 0.1f) || next == pos)
    {
        move = new Vector3(Random.Range(-1.5f, 1.5f), 0, Random.Range(-1.5f,
1.5f));
        next = pos + move;
        // 当无法获取到符合要求的 target 时，抛出异常。
        if ((++tryCount) > 100)
        {
            Debug.LogFormat("point {0}, area({1}, {2}, {3}, {4}, {5})", pos,
area, x_down, x_up, z_down, z_up);
            throw new System.Exception("Too many loops for finding a
target");
        }
    }
    return next;
}
```

追随玩家

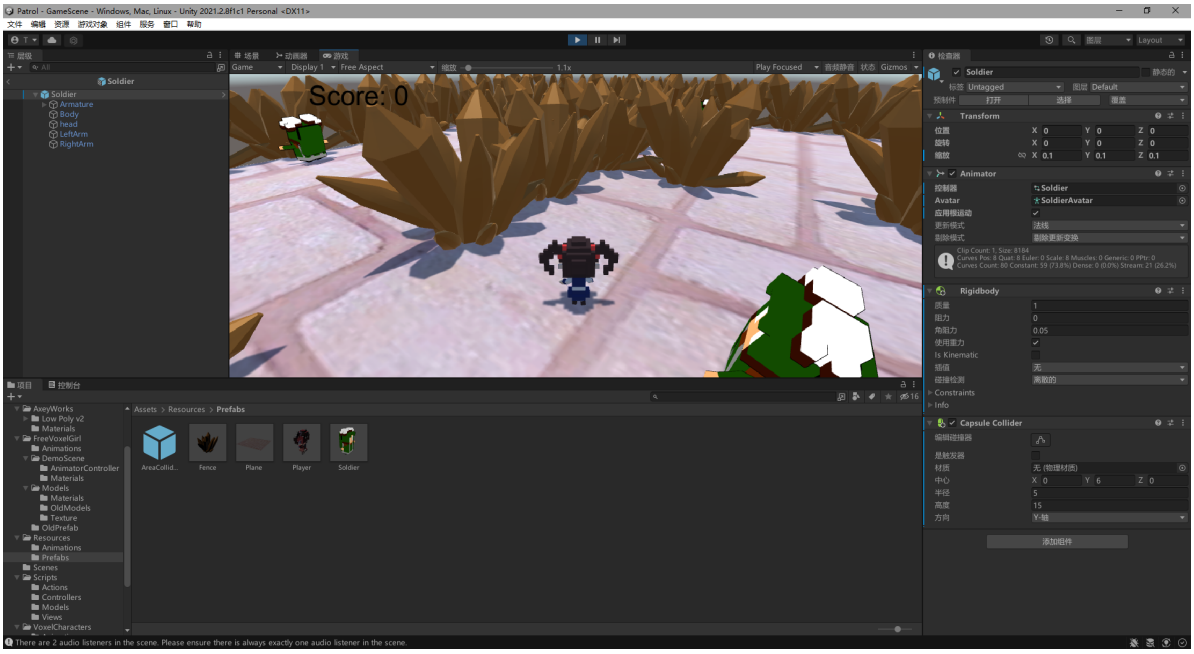
我们在 `TraceAction` 类中，实现了巡逻兵追随玩家的动作，具体方式是调用 `Vector3.MoveTowards` 方法。我们在 `GameController` 的 `update` 方法中，根据不同的区域，设置巡逻兵的动作类型。

```
// 巡逻兵追随玩家。
public void Trace(GameObject patrol, GameObject player)
{
    var area = patrol.GetComponent<Soldier>().area;
    // 防止重入。
    if (area == currentArea)
    {
        return;
    }
    currentArea = area;
```

```
    if (moveToActions.ContainsKey(area))
    {
        moveToActions[area].destroy = true;
    }
    TraceAction action = TraceAction.GetAction(patrol, this, player, 1.5f);
    AddAction(action);
}

// 设置巡逻兵动作类型。
for (int i = 0; i < 9; ++i)
{
    // 不在当前区域的巡逻兵进行自主巡逻。
    if (i != currentArea)
    {
        actionManager.GoAround(soldiers[i]);
    }
    else // 在当前区域的巡逻兵对玩家进行追随。
    {
        soldiers[i].GetComponent<Soldier>().isFollowing = true;
        actionManager.Trace(soldiers[i], player);
    }
}
```

最终呈现效果：




参考：https://github.com/Jiahonzheng/Unity-3D-Learning/tree/master/HW6