# **Unity完全自制游戏纸箱战争项目记录（20180702）**

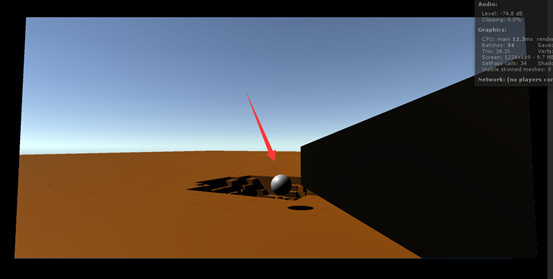
作者：[Pluto](http://gad.qq.com/user/index?id=2267380)

链接：<http://gad.qq.com/article/detail/48339>

经过周日一天懈怠的洗礼，今天又要投入到项目的构建中来，想想18年已经过去一半了，似乎什么目标都没有实现，挺让人压抑的。

本周是项目正式进行的第一天，刚开始就需要解决周六留下的遗留问题。

给老师看了坦克炮击后留下的弹坑效果，如图上所示，箭头指示的是用到了前天写的炸弹脚本后的效果。



炸弹在爆炸之后消耗地形方块的生命值，不断的分裂最后产生爆炸，在地形上留下一个空洞。

当我操纵人物进入深坑的时候，发现了一个不严重，但足以毁灭游戏的问题。

让操控角色以一定速度碰撞到任何碰撞器之后便会失去移动能力，就像是被卡主了一样，左右移动之后可以脱离，这个问题的出现使得不能从深坑中跳跃出来。

研究了半天，也没有得出个所以然来，咨询了老师得到建议是使用角色控制器来代替本身用于检测碰撞用的胶囊碰撞器，虽然问题可以得到解决，但在之后的对战设计中会存在更大的问题，只能从胶囊碰撞器入手解决。

在网上查阅了相关资料，似乎并没有找到类似的事件的解决办法，大致浏览了一遍Unity提供的第一人称控制器的代码，并没有发现有什么异常的地方。

在之前的测试中，把人体的刚体设置成了正常人的体重用来模拟碰撞效果，把质量设置为1后便没有卡住的问题了，这点还需要在之后进行详细的研究。

从项目确立的时候就有两个可以预见的难度摆在眼前，第一是可破坏物的设置，第二就是敌人的AI，我处理问题喜欢从困难的入手，如果困难的问题都能顺利解决，那么简单的问题就不再算作问题了。

可破坏物的问题算是解决了七七八八，接下来就是另一大难点敌人的AI。

经查阅资料，传统意义上的FPS的AI是要比RTS游戏简单不少的，只要打得准反应快就行了，可我觉得，如果单单是这样，玩家的游戏体验将会大大折扣。

而且传统网游中的AI智商都是相当低下，他们通过寻路系统进攻目标点或者是敌人，没有任何变数，有时候玩家只需要卡一个视野就能如同切菜一般杀死敌人，这样的游戏还有什么意义，还不如不止一个靶场来让玩家练习枪法来的实在。

因为游戏中还存在着可破坏地形以及可破坏物的限制，Unity中的寻路系统显得有些无力起来，在早期测试中，已经烘焙好的寻路网格，就算地面上出现了大型的坑洞，受导航单位走在上面如履平地，当玩家在游戏中游玩看到敌人能够在坑洞上行走，显然是很不符合逻辑的。

没办法，只能自己解决AI的寻路问题。

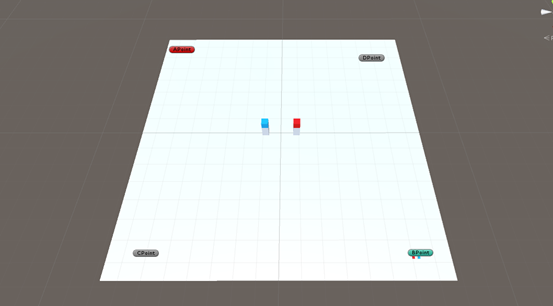
在我的设想中，应该是AI会自行选择目标点，随后朝着目标点发起攻击，期间伴随着一些突发事件，诸如驾驶载具等等...

最早我是打算通过一个AI脚本就实现全部的AI功能，随后发现这会有一定的局限性，而且就算是同阵营的NPC之间也很难产生一定的联系，在游戏中会产生诸多缺点，诸如全部的NPC都进攻同一目标点的行为。

于是我想到RTS游戏，在RTS中，玩家扮演的一般是能纵观全局的领袖，指挥手下的士兵也只是给一个大致的目标点，具体的开火和战斗还是由AI智能实现的。

通过这点，我想到把AI分为两个模块，其中较大的作为中央控制AI，负责对战局进行分析，哪些是需要进攻的目标点，哪些是需要防守的目标点，派遣多少士兵对这个据点进行防守等等。

而较小的AI脚本则挂载在NPC控制的士兵身上，他们在得到具体的目标之后则会对目标发起进攻，当任务完成之后会向中央控制AI返回“进攻成功”的指令，中央控制AI随后会发布新的命令给士兵AI。

在着手进行AI代码编写之前，首先我创建了一个新的工程对AI进行预测试，就像当时测试可破坏地形一样，这样即使修改了某些东西导致Unity工程损坏也影响不到已有的工程文件。

如图所示，地面上有两个阵营的士兵各一个，还有四个目标点，其中A点为红色方占有，B点为蓝色方占有。

再进行AI编写前还需要先能让AI知道什么时候该进攻目标点，什么时候该防守目标点。

这里我把目标点设想为一个环形区域，只要在此区域内的阵营物体都可以参与目标点的争夺。

理论如下：

当目标点为红色方所有，蓝色士兵进入区域，对目标点执行占领，需要先消耗掉目前红方的已有占领条，一个单位即记为消耗速度为1，多个单位则进行累加，最高上限速度为3。

当占领条消耗完毕，目标区域则变为蓝色，同时对占领条进行增加，一个单位记为增加速度1，多个单位累加，最高上限速度为3。

当占领条重新增加到满为止，这个据点才算被成功拿下，可以作为重生点使用。

如果在占领过程中同时有敌方士兵进行防守，每个敌方士兵会减少一点的占领速度，如果双方在占领圈中人数相同，则占领区域既不增加也不减少。

using UnityEngine;

using System.Collections;

public class Point\_Test : MonoBehaviour

｛

    //占领圈的大小

    public float PointRad = 20.0f;

    //当前据点的归属

    public int PointCamp = 2;

    //目标滑向阵营

    public int PointWantCamp;

    //占领值

    public float PointValue = 1.0f;

    //占领速度,根据圈内人数的不同变化

    public int OccupyNumber = 0;

    public int OccupySpeed = 0;

    void Start()

    ｛

        InvokeRepeating("Occupy", 0, 1);

    ｝

    void Update()

    ｛

        Collider[] InSphereCollider = Physics.OverlapSphere(transform.position, PointRad);

        if (OccupyNumber != InSphereCollider.Length)

        ｛

            OccupyNumber = InSphereCollider.Length;

            OccupySpeed = 0;

            int OutSpeedNumber = 0;

            int SetSpeedNumber = 0;

            foreach (var hitCollider in InSphereCollider)

            ｛

                if (InSphereCollider.Length == 0)

                ｛

                    PointWantCamp = 3;

                ｝

                if (hitCollider.GetComponent<Camp>() != null)

                ｛

                    if (hitCollider.GetComponent<Camp>().CampNumber == PointCamp)

                    ｛

                        SetSpeedNumber += 1;

                    ｝

                    if (hitCollider.GetComponent<Camp>().CampNumber != PointCamp)

                    ｛

                        OutSpeedNumber += 1;

                    ｝

                ｝

            ｝

            if (SetSpeedNumber != OutSpeedNumber)

            ｛

                OccupySpeed = SetSpeedNumber - OutSpeedNumber;

                if (OccupySpeed > 3)

                ｛

                    OccupySpeed = 3;

                ｝

                if (OccupySpeed < -3)

                ｛

                    OccupySpeed = -3;

                ｝

            ｝

        ｝

    ｝

    void Occupy()

    ｛

        if (PointValue <= 1 || PointValue >= 0)

        ｛

            PointValue = PointValue + OccupySpeed \* 0.01f;

        ｝

        if (PointValue < 0)

        ｛

            if (PointWantCamp != PointCamp)

            ｛

                PointCamp = PointWantCamp;

                PointValue = 0;

            ｝

        ｝

        if (PointValue > 1)

        ｛

            PointValue = 1;

        ｝

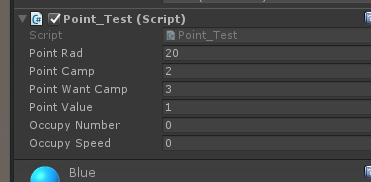
    ｝

｝

在代码中，我定义了一个函数叫“目标滑向阵营”，不知道该怎么形容，大概意思就是判断该区域内成员较多的一方势力，这就说明目标点的占领方向是向着该方向倾斜，主要用来修正当占领条不满时，据点内双方都不存在人数的情况。

说句实在话，今天的状态实在是不好，可能是因昨晚没有休息好缘故，在处理逻辑的时候总感觉意识有点模糊似的。

代码写完之后，感觉像是写错了，Debug一下居然输出了正确结果，也让我自己有点小意外。



Point Rad为目标点的范围；

Point Camp为目标点的阵营编号；

Point Want Camp为目标点滑向阵营；

Point Value为目标点的占领值；

Occupy Number为占领区域内的总人数

Occupy Speed为占领的速度

这样是在面板中可以定义的属性，在测试阶段看上去数目繁杂，有些很乱，当测试通过之后，就可以把不需要外部调用的变量从公有改成私有，虽然不知道这么做还有什么好处，最起码在设置参数的时候看上去能舒服一点。

完成了目标点的占领测试之后，项目近战又前进了一步。

接下来是AI总控制器和个体AI的编写，因为两者之间存在着很多练习，所以在这个程序中，我选择同时编写两个AI。

在编写个体AI的时候，我想到战争时期，通讯不便的情况下是如何发布战斗指令的，从烽火狼烟联想到飞鸽传书。

两者都是用来传递信号的，但都有一个通病，传送的信息量太小，飞鸽传说或许还能好一些，狼烟兴许能通过改变烟雾的节奏？这个说不清楚。

接着还联想到了Unity中的动画状态机，似乎动画状态机的运作方式跟我设想的很相似，已经既定好了若干个行动目标，如果达成了某项条件，就执行预先设定好的动作。

照着这个思想，我想到了AI总控制器给个体AI传递信息的时候可以只传递坐标信息，动作制订则由预先编写好的方法体来实现，通过代号来判断具体指令的目的。

比如“0000”是朝向目标点行进，“0001”是朝着目标点行进，在目标点中环形运动巡逻等等...

在AI总控制器中我也把目标点的可重生性和可攻击性以及攻击的先后顺序进行的方法预编辑，在使用的时候方便调用。

预计整个AI总控制写下来在两千行左右，同时还要处理大量的逻辑问题，对我这样一个初学者来说算是个不小的挑战。