游戏开发中的人工智能之《地牢逃生》

项目报告

开发者：王孙洪

目　录

一、　游戏概述　3

二、　游戏机制　4

三、　项目实现　5

1. 游戏整体操作设计　5
2. 迷宫设计　8
3. 骑士动画实现　14
4. 幽灵性格的人工智能的逻辑设计实现　17

A. 追赶型　17

B. 埋伏等待型　20

C. 包围攻击型　21

D. 随机型　23

四、　小结　25

一、 游戏概述

“处处笙歌明月夜，家家笑语绮窗花”，就在这么一个和谐的地方，谁也没有想到，整个国度会突然被黑暗包围，周遭的一切都变得死气沉沉。身为一国之主，塞顿带领着他的部下，去寻找和毁灭黑势力的根源。不幸的是，随着黑势力的逐渐扩大，他的随从也一个个随之消失，而他已置身于迷宫般的黑暗之中，孤军奋战的冒险也就此开始。

游戏的目的就是让玩家扮演游戏的主角塞顿收集藏在迷宫内所有的宝石和宝箱，并且不能被幽灵捉到。

游戏玩法：键盘方向键控制骑士移动。

关卡剧情如下：

第一关：以死对抗好不容易消灭了敌人，除了塞顿，整个军队都被团灭了。为了能够消除以后的敌人，他必须寻找充足的资源和武器来充实自己，装备自己。而他又怎料到身后还有被落下的、未消灭的敌人正向他袭来……

第二关：难缠的幽灵在被消灭瞬间发出了一声刺耳的尖叫，是惨痛的叫声，亦或是属于它们之间的什么信号？身后又有一股逼人的寒气向塞顿袭来……

第三关：随着消灭的敌人越来越多，所招来的敌人也越来越多。一个人的战争已经无法避免，这条路还有很长一段要走，只能寻找更多的武器和资源来备战了。

第四、五关剧情略。

二、 游戏机制

使用以下几种资源来模拟《地牢逃生》的机制。

① **宝石：**游戏的迷宫中散布着许多宝石，玩家必须控制骑士把它们全部收集完毕才能过关。这里的宝石是一种有形资源，宝石的数量是固定的，不会随着游戏的进行而产生，除非玩家进入下一关。

② **剑：**每个关卡中都会随机出现剑，当骑士拾取剑后，就能获得杀掉幽灵的能力。剑是一种稀少的有形资源，玩家必须合理加以利用。

③ **宝箱：**迷宫中有时候会出现宝箱，骑士获得宝箱可以获得额外分数。注意宝箱会随机出现，随时消失。

④ **幽灵：**游戏中有四个幽灵，它们会满迷宫追逐玩家控制的吃骑士。幽灵在迷宫中行走，幽灵也是一种有形资源。

⑤ **生命：**游戏开始时，骑士拥有三条命。这个游戏中的生命是无形资源，一旦玩家损失掉所有生命，游戏就会结束。

⑥ **分数：**骑士每收集一个宝石、收集宝箱或杀掉幽灵，就会将它们消耗掉，并获得一定分数。这个游戏的目标就是尽可能多地获取分数。分数是一种无形资源。

三、 项目实现

1. 游戏整体操作设计

在地牢逃生中，作为游戏舞台的迷宫的道路非常狭窄，只允许 1 个角色通过，因为砖块也是按规则排列 的，所以角色只能往上下左右四个方向移动。游戏中所有作为障碍物的砖块被整齐地排列在了网格上。因此控制骑士方向的玩家在拐弯时必须精确地把握时机，否则骑士将碰到墙壁。

本项目中，按下按键后角色并不会立即改变方向，而是首先探测是否可以沿着该方向前进。如果无法前进，则保持原方向继续运动，直到到达拐角处才改变方向。提前按下方向键就可以使角色平稳地经过拐角，但在任何时候都可以向后改变方向。

游戏画面中整个地图都被纵横切成了若干个小网格，并通过在一部分网格中排列砖块从而制作出了迷宫。为了简化处理，本游戏中将砖块的尺寸设置为 1 单位长度。这样，网格的 XZ 坐标就是 1.0、2.0、3.0……这样的整数。

控制骑士的程序如下：

GridMove.Move 方法

```
    public void Move(float t)
    {
        // 下次移动到的位置
        Vector3 pos = transform.position;
    // (a) 下次移动的位置
        pos += m_direction * SPEED * t;


        // 检测是否通过了网格
        bool across = false;

        // 如果和取整后的数值不同，说明跨越了网格
    // (b) 如果坐标的小数值和取整后的值不同，则穿过网格的边界
        if ((int)pos.x != (int)transform.position.x)
            across = true;
        if ((int)pos.z != (int)transform.position.z)
            across = true;

        Vector3 near_grid = new Vector3(Mathf.Round(p
```

```
os.x),pos.y,Mathf.Round(pos.z));
        m_current_grid = near_grid;
        // 是否撞到了正面墙壁
        Vector3 forward_pos = pos + m_direction*0.5f;
 // Ray 射向半个 Unit 的位置
        if (Mathf.RoundToInt(forward_pos.x) != Mathf.
RoundToInt(pos.x) ||
            Mathf.RoundToInt(forward_pos.z) != Mathf.
RoundToInt(pos.z)) {
            Vector3 tpos =pos;
            tpos.y += HITCHECK_HEIGHT;
            bool collided = Physics.Raycast (tpos,m_d
irection,1.0f,HITCHECK_LAYER_MASK);
            if (collided) {
                pos = near_grid;
                across = true;
            }
        }
        if (across || (pos-
near_grid).magnitude < 0.00005f) {
            Vector3 direction_save = m_direction;

            // （c）发送消息，调用 OnGrid() 方法
            SendMessage("OnGrid",pos);

            if (Vector3.Dot(direction_save,m_directio
n )< 0.00005f)
                pos = near_grid + m_direction * 0.001
f;  // 如果一动不动则再次执行 OnGrid
        }

        m_move_vector = (pos-transform.position)/t;
        transform.position = pos;
    }
```

程序设计思路：

- 用现在的位置加上移动方向的向量求出下次移动的位置。
- 分别对现在的位置和下次移动 的位置取整并对结果进行比较。
- 向对象发送消息并执行 OnGrid 方法。

穿过网格边界的瞬间，除了使角色改变方向，还做了拾取宝石的处理。在本游戏中，利用“穿过网格边界=拾取宝石”，这样不仅简化了处理，而且比用碰撞器检测更有优势。

PlayerController.OnGrid 方法

```
public void OnGrid(Vector3 newPos)
    {
        // 捡起宝石
        m_map.PickUpItem(newPos);
        Vector3 direction;
        if (GlobalParam.GetInstance().IsAdvertiseMode
())
            direction  = GetAdvModeMoveDirection();
        else
            direction = GetMoveDirection();

        // 没有按键输入（不转换方向）
        if (direction == Vector3.zero)
            return;

        // 先按键输入的方向移动（允许移动的情况下）
//如果按下了相反的方向键，则转换方向
        if (!m_grid_move.CheckWall(direction))
            m_grid_move.SetDirection(direction);
    }
```

向后改变方向可以随时进行，这部分处理被写在了 PlayController.Update_Normal 方法中。通过 GridMove.IsReverseDirection 方法比较用户输入的方向和目前正在行走的方向，如果二者相反，则将角色旋转 180°。

2. 迷宫设计

地牢逃生中的每一关地形都不同。本游戏利用文本文件格式制作地图，一共创建 5 个关卡。

数据格式如下：

（1）c	宝石
（2）*	砖块
（3）p	骑士
（4）1\2\3\4	幽灵（一共有 4 种）
（5）s	剑
（6）t	宝箱
（7）x	空白处

在地图文本文件（CSV）中，用逗号，分开列出：

```
*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*
*,c,c,c,c,c,c,c,c,c,c,*,c,c,c,c,c,c,c,c,c,c,*
*,c,*,*,*,c,*,*,*,x,*,x,*,*,*,c,*,*,*,c,*
*,c,*,c,c,c,c,*,*,4,s,2,*,*,c,c,c,c,*,c,*
*,c,*,c,*,*,c,*,*,*,*,*,*,*,c,*,*,c,*,c,*
*,c,c,c,*,c,c,c,c,c,c,c,c,c,c,c,*,c,c,c,*
*,c,*,c,*,c,*,c,*,*,*,*,*,c,*,c,*,c,*,c,*
*,c,*,c,*,c,*,c,*,c,p,c,*,c,*,c,*,c,*,c,*
*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*
*,c,c,c,c,c,*,c,c,c,*,c,c,c,*,c,c,c,c,c,*
*,c,*,s,*,c,*,*,*,c,*,c,*,*,*,c,*,s,*,c,*
*,c,*,c,*,c,*,c,c,c,*,c,c,c,*,c,*,c,*,c,*
*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*
*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*,c,*
*,c,c,c,c,c,*,c,*,c,t,c,*,c,*,c,c,c,c,c,*
*,c,*,c,*,c,c,c,*,c,c,c,*,c,c,c,*,c,*,c,*
*,c,*,c,*,*,*,c,*,*,*,*,*,c,*,*,*,c,*,c,*
*,c,*,c,c,c,c,c,*,3,s,1,*,c,c,c,c,c,*,c,*
*,c,*,*,*,c,*,*,*,x,*,x,*,*,*,c,*,*,*,c,*
*,c,c,c,c,c,c,c,c,c,c,*,c,c,c,c,c,c,c,c,c,*
*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*
```

（1）骑士拾取的宝石，将宝石全部收集完毕后过关；

（2）砖块。排列砖块以生成迷宫。砖块除了被用作显示迷宫的模型外，还需具备碰撞检测的功能。

（3）骑士的开始位置。关卡开始时骑士所在的地方。每个关卡中都必须有一个开始位置。

（4）幽灵，从 1 到 4，每个数值分别代表一种性格的幽灵。

（5）剑，拾起剑后能进行一次反击。

（6）宝箱，获取宝箱后将获得得分奖励。游戏开始后，每经过一定的时间将出现一次。

（7）空白区域，用于制作外墙的周围以及被墙壁包围起来的空间。

将地图数据的文本文件作为文本资源添加到预设中，并利用脚本处理数据：

Map.LoadFromAsset 方法

```
    private void LoadFromAsset(TextAsset asset)
    {
        m_mapData.offset_x = MAP_ORIGIN_X;
        m_mapData.offset_z = MAP_ORIGIN_Z;

        if (asset != null) {

            string txtMapData = asset.text;

            // Split 方法将空元素删除的选项
            System.StringSplitOptions    option = Syste
m.StringSplitOptions.RemoveEmptyEntries;

            //  (a) 用换行符将每行切割开，一个数组元素存储一行
            string[] lines = txtMapData.Split(new char[
] {'\r','\n'},option);

            // (b)用“,”将每个字符分割开
            char[] spliter = new char[1] {','};

            // (c)第一行地图的尺寸
            string[] sizewh = lines[0].Split(spliter,op
tion);
            m_mapData.width = int.Parse(sizewh[0]);
            m_mapData.length = int.Parse(sizewh[1]);

            char[,] mapdata = new char[m_mapData.length
,m_mapData.width];

            for (int lineCnt = 0; lineCnt < m_mapData.l
ength; lineCnt++) {

                // 为了保证文本文件中即时指定了特别大的值也不会
出问题
                // 进行检测
                if (lines.Length <= lineCnt+1)
```

```
                    break;

                // (d)用“,”将每个字符分割开
                string[] data = lines[m_mapData.length-
lineCnt].Split(spliter,option);

                for (int col = 0; col < m_mapData.width
; col++) {

                    if (data.Length <= col)
                        break;

                    mapdata[lineCnt,col] = data[col][0]
;
                }
            }
            m_mapData.data = mapdata;
        } else {
            Debug.LogWarning("Map data asset is null");
        }
    }
```

程序设计思路：

(a) 对文本文件整体以行为单位进行分割。只要在 Split 方法中将换行符指定为分割符，就能够得到被分割开的各行文本的 String 数组。

(b) 文件中表示对象的字符已经被逗号分隔开，继续用逗号分隔开文本并将其放入 String 数组中。这样全体文本就被分解为了处理的最小单位 Token，Token=1 个字符。

(c) 第一行中记录了地图的横向和纵向长度，分别取得横向和纵向的尺寸，创建用于存储地图数据的数组。

再利用脚本按照数据生成地图模型：

Map.CreateMap 方法

```
    void CreateMap(bool collisionMode,string mapName,bo
ol modelOnly = false)  // todo: 废除碰撞模式
    {

        m_mapObjects = new GameObject(mapName);
        m_spawnPositions = new Vector3[SPAWN_POINT_TYPE
_NUM];
```

```
        if (m_items != null)
            Destroy(m_items);
        m_items = new GameObject("Item Folder");

        for (int x = 0; x < m_mapData.width; x++) {
            for (int z = 0; z < m_mapData.length; z++)
{
                // (a) 根据地图数据配置游戏对象
                // 配置方块模型
                switch (m_mapData.data[z,x]) {

                // (a1) 墙壁
                case WALL:
                    if (collisionMode) {
                        GameObject o = Instantiate(m_wa
llForCollision,
                                                   new Vect
or3(x+m_mapData.offset_x,0.5f,z+m_mapData.offset_z),
                                                   Quaterni
on.identity) as GameObject;
                        o.transform.parent = m_mapObjec
ts.transform;
                    } else {

                        GameObject o = Instantiate(m_wa
llObject[0],
                                                   new
Vector3(x+m_mapData.offset_x,WALL_Y,z+m_mapData.offset_
z),
                                                   Quat
ernion.identity) as GameObject;
                        o.transform.parent = m_mapObjec
ts.transform;
                    }
                    break;

                // (a2) 骑士
                case PLAYER_SPAWN_POINT:
                    m_spawnPositions[(int)SPAWN_POINT_T
YPE.BLOCK_SPAWN_POINT_PLAYER] = new Vector3(x+m_mapData
.offset_x,0.0f,z+m_mapData.offset_z);
                    break;

                // (a3) 宝箱
                case TERASURE_SPAWN_POINT:
                    m_spawnPositions[(int)SPAWN_POINT_T
YPE.BLOCK_SPAWN_TREASURE] = new Vector3(x+m_mapData.off
set_x,0.0f,z+m_mapData.offset_z);
                    break;
```

```
                //  (a4)幽灵
                case '1':
                case '2':
                case '3':
                case '4':
                    int enemyType = int.Parse(m_mapData
.data[z,x].ToString());
                    m_spawnPositions[enemyType] = new V
ector3(x+m_mapData.offset_x,0.0f,z+m_mapData.offset_z);
                    break;

                // 用于检测怪物的AI
                // 在相同位置创建“追赶型”和“伏击型”
                case '5':
                    m_spawnPositions[1] = new Vector3(x
+m_mapData.offset_x,0.0f,z+m_mapData.offset_z);
                    m_spawnPositions[2] = new Vector3(x
+m_mapData.offset_x,0.0f,z+m_mapData.offset_z);
                    break;

                default:

                    break;
                }
            }
        }

        // 只生成地图数据
        if (modelOnly)
            return;

        Transform[] children = m_mapObjects.GetComponen
tsInChildren<Transform>();
        m_mapObjects.AddComponent<CombineChildren_Custo
m>();
        m_mapObjects.GetComponent<CombineChildren_Custo
m>().Combine();
        Destroy(m_mapObjects.GetComponent<CombineChildr
en_Custom>());

        for (int i = 1; i < children.Length; i++)
            MyDestroy(children[i].gameObject,collisionM
ode);

        if (collisionMode) {
            m_mapObjects.AddComponent<MeshCollider>();
            m_mapObjects.GetComponent<MeshCollider>().s
haredMesh = m_mapObjects.GetComponent<MeshFilter>().mes
h;
            MyDestroy(m_mapObjects.GetComponent<MeshRen
```

```
derer>(),collisionMode);
            m_mapCollision = m_mapObjects;
        }

        if (!collisionMode)
            SetupGemsAndItems();
    }
```

程序设计思路：

（a）根据文本资源生成的地图数据 m_mapData.data[]是一个二维数组，可以直接使用 XZ 的坐标值作为数组的索引来访问数据。

（a1）“墙壁”部分用于生成墙壁对象

（a2）“骑士”部分用于在游戏启动时生成对象，这里提前记录下坐标，（a3）宝箱也做同样处理。

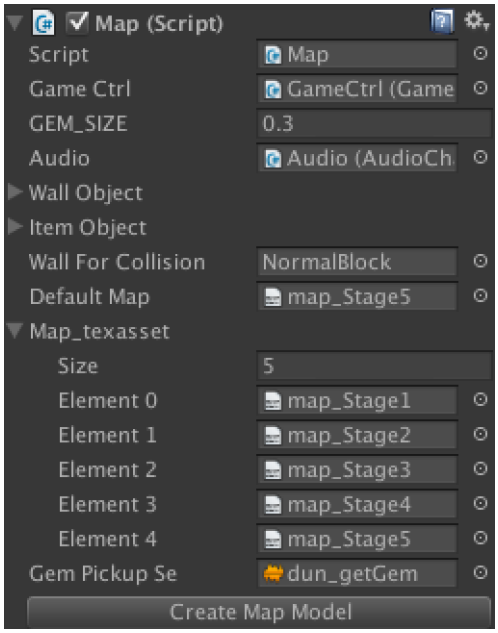
（a4）地图数据为数字时生成幽灵。生成的幽灵的性格因数字而异。各种性格的幽灵只生成一只。

并且可以利用定制编辑器的功能生成迷宫：

MapModelCreator.cs

```
using UnityEngine;
using System.Collections;
//(a)声明使用 UnityEditor 名称空间
using UnityEditor;
//（b）表明是 map 类的定制编辑器
[CustomEditor(typeof(Map))]
//（c）继承 Editor 类生成定制编辑器使用类
public class MapModelCreator : Editor {
//（d）重载 OnInspectorGUI 方法
    public override void OnInspectorGUI() {
//（e）执行标准功能
        DrawDefaultInspector ();
        if (GUILayout.Button("Create Map Model")) {
            Map map = target as Map;
            map.CreateModel();
        }
    }
}
```

生成定制编辑器如下（Create Map Model）：



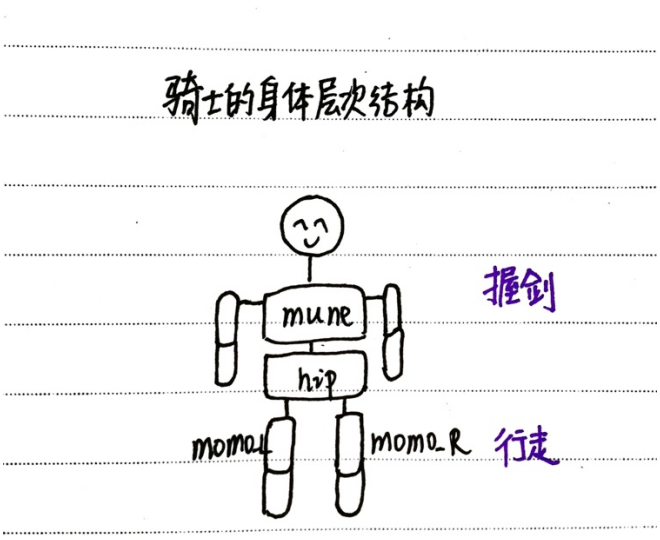
3. 骑士动画实现

Unity 中能够对角色动画进行简单的处理。本游戏采用两个方法：

① 让身体各个部位播放不同动画；

② 结合动作播放音效（SE）；

骑士的身体层次结构如下：



对应层级视图中 Player 对象：

各个对象对应手脚头等身体各个部位，像人体骨骼一样，通过移动这些对象，让角色整体产生动画效果。

持剑动画代码如下：

Weapon.Start 方法

```
    void Start () {
        m_equiped = false;
        m_sword.renderer.enabled = false;

        // 开始合成动画的结点
    // (a) 寻找 mune 节点
        Transform mixTransform = transform.Find("root/hip/mune");

        // 挥动宝剑
        animation["up_sword_action"].layer = 1;
    // (b) 指定开始合成动画节点
        animation["up_sword_action"].AddMixingTransform(mixTransform);

        // 把剑举到胸前
        animation["up_sword"].layer = 1;
    // (b) 指定开始合成动画节点
        animation["up_sword"].AddMixingTransform(mixTransform);

        m_audio = FindObjectOfType(typeof(AudioChannels)) as AudioChannels;
```

```
        m_combo = 0;
    }
```

程序设计思路：

（a）寻找开始合成动画的 mune 节点。

（b）指定合成动画的最上层节点，这里指定了 mune，因此 mune 和它所有子对象将合成动画。对骑士模型而言，mune 及其下的所有节点构成了角色的上半身完成持剑动画效果。

在游戏中，当骑士行走时，将配合着动画播放脚步声的音效。在特定的时刻设定事件，并调用事件的方法。实现注册好关键帧和回调方法，当动画播放到该帧时就会执行指定的方法。播放脚步声的事件代码如下：

CharaAnimator.InitializeAnimation 方法

```
    protected virtual void InitializeAnimations()
    {
        animation["run"].speed = 2.0f;

        // 脚步声事件
        //（a）创建事件
        AnimationEvent ev = new AnimationEvent();
        ev.time = 0.0f;
        ev.functionName = "PlayStepSound";
        ev.floatParameter = 1.0f;
        //（b）注册事件
        animation["run"].clip.AddEvent(ev);

        AnimationEvent ev2 = new AnimationEvent();
        ev2.time = animation["run"].clip.length / 2.0f;
        ev2.functionName = "PlayStepSound";
        ev2.floatParameter = 1.06f;
        animation["run"].clip.AddEvent(ev2);
    }
```

程序设计思路：

（a）创建 AnimationEvent 事件，指定触发事件的关键帧和回调函数。

（b）将事件注册到 AnimationClip。

骑士行走的动画就是左右脚交替迈出这一动作的循环播放，每只脚着地的瞬间都将激活事件。

PlayerController.PlayStepSound 方法

```
    // 播放脚步声
    public void PlayStepSound(AnimationEvent ev)
    {
        (FindObjectOfType(typeof(AudioChannels)) as Aud
ioChannels).PlayOneShot(m_stepSE,1.0f,0.0f,ev.floatPara
meter);

    }
```

4. 幽灵性格的人工智能的逻辑设计实现

《地牢逃生》中有 4 中幽灵，各个幽灵的行为各不相同。在游戏中控制好进退非常重要，在游戏过程中都要开动脑筋，揣测对方的习惯，分析对方的行为，以避免在狭窄的迷宫内被幽灵追上。

四种幽灵的性格如下：

①追赶型：幽灵一直在骑士后面追赶。

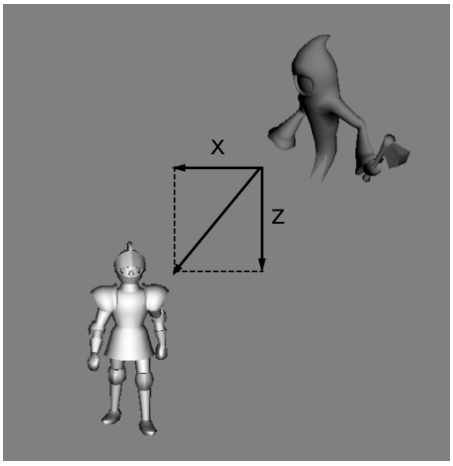
②埋伏等待型：幽灵提前在骑士将要到达的地方埋伏等待。

③包围攻击型：配合追赶型幽灵夹击骑士。

④随机型：和骑士的位置无关，幽灵随机改变方向，其行为毫无规律无法预测。

A. 追赶型

幽灵首先计算出从自己的位置指向骑士的位置的向量：



程序如下：

MonsterCtrl.Tracer 方法

```
    private void Tracer(Vector3 newPos)
    {
        Vector3 newDirection1st,newDirection2nd;

        //  (a) 从自身位置指向玩家位置的向量
        Vector3 diff = m_player.position - newPos;

        //  (b) 选择X, Z 中绝对值更大的一个
        if (Mathf.Abs(diff.x) > Mathf.Abs(diff.z)) {
            newDirection1st = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection2nd = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        } else {
            newDirection2nd = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection1st = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        }

        //  (c) 从两个候选值中选择可能移动的方向
        Vector3 newDir = DirectionChoice(newDirection1s
t,newDirection2nd);

        if (newDir == Vector3.zero)
            m_grid_move.SetDirection(-
m_grid_move.GetDirection());
        else
            m_grid_move.SetDirection(newDir);
    }
```

程序设计思路：

(a) 求出从自己的位置指向骑士的位置的向量。

(b) 选择 X 分量和 Z 分量中绝对值较大的一个。

X、Z 两个分量中，将绝对值较大的一个代入 newDirection1st，较小的一个代入 newDirection2nd。绝对值较小的那个值将其作为优先度较低的候补值存了起来。因为根据迷宫的形状变化，无法保证一定能够朝期望的方向移动。

(c) 调用 DirectionChoice 方法的实现。

MonsterCtrl.DirectionChoice 方法

```
    //  从候选值中获取可能的移动方向
    //  参数：
    //  first    第一候选值
    //  second   第二候选值
    //  返回值：
    //  移动方向／ 不允许移动时为Vector3.zero
    private Vector3 DirectionChoice(Vector3 first, Vect
or3 second)
    {
        // 按照下列顺序
        // 第一候选值
        // 第二候选值
        // 第二候选值的反方向
        // 第一候选值的反方向
        //
        // 进行探测，返回允许移动的方向

        // 第一候选值
        if (!m_grid_move.IsReverseDirection(first) &&
            !m_grid_move.CheckWall(first))
            return first;

        // 第二候选值
        if (!m_grid_move.IsReverseDirection(second) &&
            !m_grid_move.CheckWall(second))
            return second;

        first *= -1.0f;
        second *= -1.0f;
        // 第二候选值的反方向
        if (!m_grid_move.IsReverseDirection(second) &&
            !m_grid_move.CheckWall(second))
            return second;
```

```
        // 第一候选值的反方向
        if (!m_grid_move.IsReverseDirection(first) &&
            !m_grid_move.CheckWall(first))
            return first;

        return Vector3.zero;
    }
```

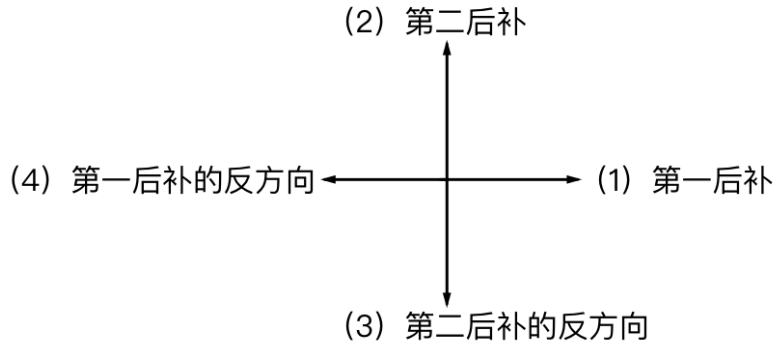
这个方法会按照下列顺序检查各个方向是否为墙壁，若允许移动就将其作为移动方向返回，这里就有测试迷宫通道：

① 第一后补

② 第二后补

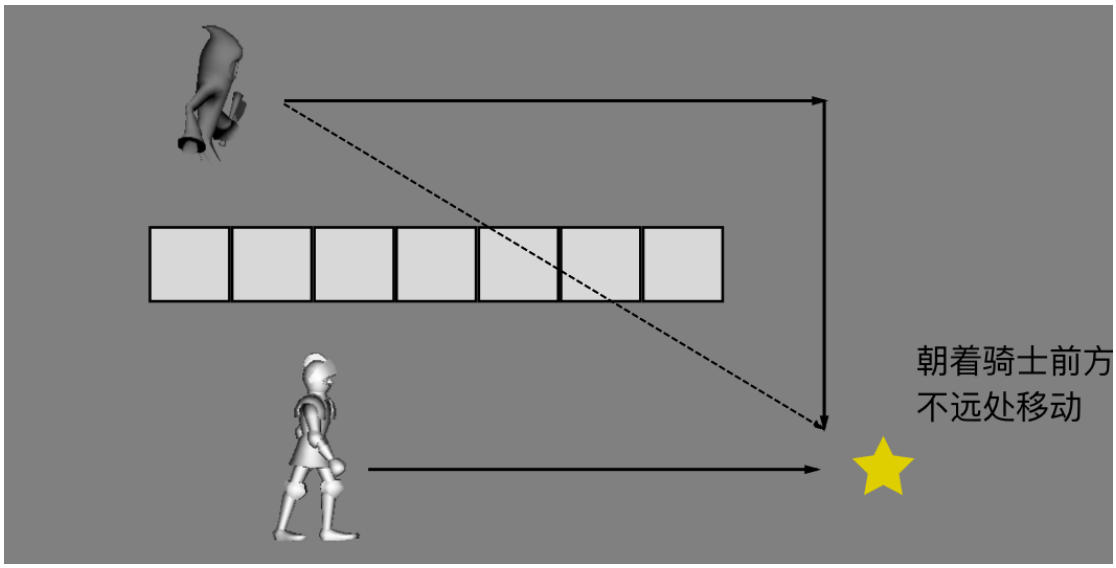
③ 第二后补的反方向

④ 第一后补的反方向



B. 埋伏等待型

埋伏等待型的路线示意图：



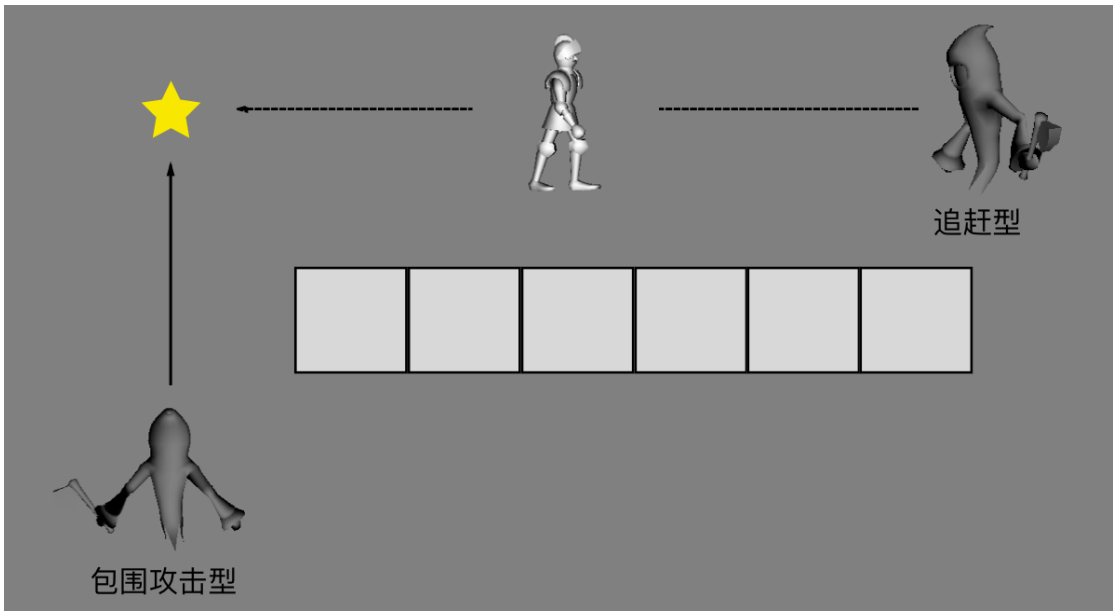
和追赶型幽灵相比，埋伏等待型的目标是骑士前方不远处的位置，而追赶型的目标是骑士的位置，实现程序如下：

MonsterCtrl.Ambush 方法

```
    private void Ambush(Vector3 newPos)
    {
        Vector3 newDirection1st,newDirection2nd;

        // 将玩家的前方设置为目标位置
        Vector3 diff = m_player.position + m_player.for
ward*AMBUSH_DISTANCE - newPos;

        if (Mathf.Abs(diff.x) > Mathf.Abs(diff.z)) {
            newDirection1st = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection2nd = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        } else {
            newDirection2nd = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection1st = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        }

        Vector3 newDir = DirectionChoice(newDirection1s
t,newDirection2nd);
        if (newDir == Vector3.zero)
            m_grid_move.SetDirection(-
m_grid_move.GetDirection());
        else
            m_grid_move.SetDirection(newDir);

    }
```

C. 包围攻击型

包围攻击型幽灵向着以骑士为中心和追赶型幽灵对称的位置移动。移动的方法与追赶型和埋伏等待型相同。



实现程序如下：

MonsterCtrl.Pincer 方法

```
    private void Pincer(Vector3 newPos)
    {
        Vector3 newDirection1st,newDirection2nd;

        Vector3 diff;
        if (m_tracer == null)
            diff = m_player.position - newPos;
        else
            diff = m_player.position * 2 - m_tracer.pos
ition - newPos;


        if (Mathf.Abs(diff.x) > Mathf.Abs(diff.z)) {
            newDirection1st = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection2nd = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        } else {
            newDirection2nd = new Vector3(1,0,0) * Math
f.Sign(diff.x);
            newDirection1st = new Vector3(0,0,1) * Math
f.Sign(diff.z);
        }

        Vector3 newDir = DirectionChoice(newDirection1s
t,newDirection2nd);
        if (newDir == Vector3.zero)
            m_grid_move.SetDirection(-
```

```
m_grid_move.GetDirection());
        else
            m_grid_move.SetDirection(newDir);

    }
```

D. 随机型

随机型的幽灵采用随机数来决定它的前进方向。

MonsterCtrl.RandomAI 方法

```
    private void RandomAI(Vector3 newPos)
    {
        Vector3 newDirection1st,newDirection2nd;
        Vector3 diff = m_player.position - newPos;
        if (diff.magnitude < DISTANCE_RANDOM_MOVE) {
            int r = Random.Range(0,4);

            // (a)  按照纵横成对选择
            newDirection1st = DIRECTION_VEC[r];
            newDirection2nd = DIRECTION_VEC[(r+2)%4];
            // (b) 通过随机将第 2 后补取反方向
            if (Random.value > 0.5f)
                newDirection2nd *= -1.0f;

            Vector3 newDir = DirectionChoice(newDirecti
on1st,newDirection2nd);
            if (newDir == Vector3.zero)
                m_grid_move.SetDirection(-
m_grid_move.GetDirection());
            else
                m_grid_move.SetDirection(newDir);
        } else
            Tracer(newPos);

    }
```

程序设计思路：

DirectionChoice 方法中必须选择两个后补项。当两个后补方向都无法前进时，则将选择它们的反方向。第一后补和第二后补正好是相反方向时，有：

第三后补=第二后补的反方向=第一后补；

第四后补=第一后补的反方向=第二后补

这样就变得只有纵向或横向可选。为了让四个后补项分别为上下左右四个方向，就必须另第一后补和第二候补中有一个是纵向有一个是横向：

(a) 随机选择第一个后补项后，取出“表内位于其后两位的元素”作为第二后补，保证了两个后补项一横一纵。

(b) 如果仅做（a）的处理，将只能使用“右和上”、“左和下”的组合，因此将第二后补通过随机数取反方向，这样，每个组合会以相同的概率出现。

四、 小结

通过本学期学习的《游戏开发中的人工智能》知识，利用Unity 游戏引擎开发本项目。《地牢逃生》中的幽灵就是人工智能中自动寻路算法、遵循路径走、有限状态自动机等知识点的实践。

未来发展规划：

- 移植至 iOS 平台，通过 GameCenter 集成分数排行榜；
- 增加更多道具，上线道具商城系统及角色选择；
- 与虚拟现实技术相结合，给玩家更多刺激。