**Unity基础**

版本问题：高版本向下兼容，但是头可能会导致资源或材质丢失

在路径：

C:\Program Files\Unity\Hub\Editor\2018.3.7f1\Editor\Data\Resources\ScriptTemplates

下修改一个文件的模板：

using System.IO;

using System.Xml;

using System.Text;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

界面

**File**文件：

-Building Settings：打包设置，Add Open Scene添加打包场景

默认第一场景为初始场景

**Edit**编辑：

-Project Settings:项目设置

-1.Audio：音频

-2.Input：输入控制器

-3.Physics：物理系统

-4.Time：时间

-5.Tags and Layers：标签和层级

-Preference：偏好设置

**Assets**资源

**Create**：创建游戏内资源

**Import**相关：导入资源及资源包

**GameObject**游戏对象：对比cocos中的Node

**Component**组件：

-Navigation：自动寻路

**Window**窗口：

-Layouts：布局切换

-General：通常窗口

-Renedering—>Lighting：取消勾选Auto Generate

-Animator:动画状态机

**Unity使用的是左手坐标系**

**四指指向Z轴**

**大拇指指向Y轴**

**掌心指向X轴**

界面内布局

**Hierarchy**资源清单窗口：搜索栏快速搜索，或者双击（shift+F）快速选择游戏对象

**Scene**场景窗口：

-Shading Mode显示模式

1.Shaded：显示物体

2.WifeFrame：线框模式，显示包围盒

3.都显示

2D显示切换：用于搭建UI

光源开关：切换是否渲染光线

**Gizmos：**辅助线

**Inspector：**属性窗口

**Transform**是所有游戏对象都具有的属性

**Position：**坐标位置

**Rotation：**旋转角度

**Scale：**缩放

材质丢失——>洋红色

3D物体游戏对象

**Cube：正方体**

**Sphere：球体**

**Capsule：胶囊体**

**Cylinder：圆柱体**

**Plane：地板**

**Quad：墙壁**

**Main Camera只能有一个，根据标签Tag来决定哪个摄像机是主摄像机**

**Unity地形编辑器**

**创建新地形：Terrain**

1. **画刷工具**

**Create Neighboor Terrains:创建一个相邻地形**

**Raise or Lower Terrain:升高或降低地形**

**Brush Size:画刷的尺寸大小**

**Opacity：画刷的疏密度**

**Paint Texture:画材质**

**Edit Terrain Layers(编辑地形材质层级)**

**Create Layer: 添加新的材质**

**Set Height:设置地形高度**

**Height:高度**

**Fight all：全设置**

**在设置完高度后回到升高或降低地形选项**

**这个时候降低地形可以达到挖地形的效果**

1. **种树工具**

**Edit Tress:添加树的预制体模型**

**Brush size:画刷大小**

**Tree Density：树的疏密程度**

1. **添加细节物体（细节所占内存比树小）**

**Edit Details：添加细节物体的2D材质图片**

1. **地形设置**

**可以修改地形的尺寸大小**

**项目文件夹归类建立**

**Scenes：场景文件夹**

**Material:材质球文件夹**

**Textures:2D纹理**

**Scripts：脚本**

**Prefabs：预制体**

**Audios:音频文件**

**Videos：视频文件**

**Animations（Unity中使用较少）**

**Animatiors:动画状态机（一般用动画状态机替代animation）**

**Sprites: 精灵图片**

**创建物体操作顺序**

1. **修改物体的位置（归位）**
2. **修改物体名字**
3. **修改物体的外观（颜色，材质，缩放比例）**
4. **修改物体的Tags标签和Layers层级**

**Unity光源**

**Directional Light:直射光**

**Sport Light:聚光灯**

**Range：照射范围**

**Sport Angel：照射角度**

**Intensity:光照强度**

**Shadow Mode：阴影效果**

1. **No shadow:没有阴影**
2. **Soft shaodow:阴影更加平滑**
3. **Hard shadow:让阴影更加棱角分明**

**Strength：阴影强度**

**Point Light:点光源**

**脚本：**

MonoBehaviour：用于调用Unity提供的生命周期函数，以及挂载到游戏物体上的功能

Debug.Log()来自于UnityEngine

Print()来自于MonoBehaviour类所继承

键盘输入

Input.GetKey/Down:前者只要按一次就会一直执行，后者按下一次执行一次

鼠标输入

Input.GetMouseButton/Down(0);

0代表鼠标左键

1代表鼠标右键

3代表鼠标中键

向量：有大小有方向的量

标量：只是大小没有方向的量（时间）

1. 世界坐标系：三维向量
2. 本体坐标系：三维向量
3. UI坐标系：二维向量
4. 屏幕坐标系：二维向量

直线：没有长度，没有大小，没有方向

射线：有方向，没有大小

线段：有方向，有大小

向量的模长：向量的长度（大小）

单位向量：模长为1的向量.normalize

通常的，以单位向量来表示方向向量

上（0,1）

下（0.-1） (x,y)模长为根号（x^2 + y^2）

左（-1，0）

右（1,0）

三维向量 unity中存储向量各坐标值按(x,y,z)存

(x,y,z)

Forward(0,0,1)

Back(0,0,-1) (x,y,z)模长为根号（x^2 + y^2+z^2）

Left(-1,0,0)

Right(1,0,1)

Up(0,1,0)

Down(0,-1)

向量的运算

V1（x1,y1,z1） V2(x2,y2,z2)

向量相加：V1+V2 = V3(x1 +x2,y1+y2,z1+z2)

减法参照法V1-V2 》》 V1+（-V2）

向量的乘法：任意向量和常数相乘

V1 \* n(n为常量) = V4（x1\*n,y1\*n,z1\*n）

除法对应乘法理解：乘法在放大向量，除法缩小向量

Unity中三维向量API

**Distance(v1,v2):返回v1向量到v2向量的直线距离**

如果对应量物体的position，则是两个物体中心点距离

向量的点乘和叉乘

V1，v2点乘表示为v1·v2，其结果是一个数值

v1的模长乘以v2的模长再乘以两向量夹角的余弦值

**点乘的作用：**

1. **点乘可用来判断两个向量是否垂直，如果点乘的结果为0，则两向量垂直，如果结果为正，则两向量夹角小于90，如果为负，则大于90；**
2. **点乘本质上是判断两个向量的相似程度，两个向量的夹角越小，他们越相似**
3. **得到两个向量的夹角：范围【0,180】，可以用来做敌人的视角内是否存在玩家，也可以用**来判断敌人攻击范围内是否存在玩家

*Vector3.Angle(v1,v2);用来求两向量的夹角（角度）*

1. **判断游戏内目标是否在自身的前后方位**

*Vector3.Dot(玩家的正方向,要判断目标的位置);*

玩家的正方向：player.transform.Forward

要判断的目标的位置：enemy.transform.position

当这个返回值为正时，目标在自身的前方，返回值为负时，目标在自身的后方；返回值为0时，目标在自身的左右方向。

1. 模拟飞机的飞行状态

*Vector3.Dot（飞机的位置，Vector3.up）;*

结果为0时，说明飞机平行飞行；当小于0时，飞机向下飞行；当大于0时，飞机向上飞行

叉乘（仅限于三维向量）

V1,v2叉乘的结果还是一个向量v3.

V3向量的大小等于v1的模长乘以v2的模长，再乘以v1和v2夹角的正弦值

v3向量的方向：v3向量垂直于v1和v2向量所在的平面

v1（x1，y1，z1） v2（x2，y2，z2）

V1叉乘v2得出的v3坐标为：

V3(y1\*z2-z1\*y2,z1\*x2-x1\*z2,x1\*y2-y1\*x2)

叉乘的作用：

1. 用于求平面的法线向量（叉乘的几何意义:v1v2sinα）
2. 用于求两个物体之间所形成的平行四边形的面积
3. 得到两个向量的夹角：[-90,90]

V1,v2

Vector3.Cross(v1.normalize,v2.normalize).mag;

Float num = Vector.Distance(Vector3.zero,v3);

前两步是在求v1和v2夹角的正弦值

Float radius(弧度)=Mathf.Asin(num);

Float angle(角度) = radius\*Mathf.Rad2Deg;

1. 判断目标是否在自己的左右位置

Vector.Cross（玩家的正方向，被判断的目标的位置）.y;

当y值为正时，目标在自身的右方,y值为负时，目标在自身左方；y值为0时，目标在自身的前后方向

1. 将（开火口的正方向向量）与（敌人的位置 -开火口位置）进行叉乘所得到的的结果向量，就是我们开火口转轴向量，实现瞄准锁定的功能

三维向量判断两向量垂直

v1（x1，y1，z1） v2（x2，y2，z2）

X1\*x2 + y1\*y2+z1\*z2 =0:v1和v2垂直

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

namespace 三维向量

{

internal struct Vector3

{

//三维向量三个坐标

float x;

float y;

float z;

//对应的访问器

public float X

{

get

{

return x;

}

set

{

x = value;

}

}

public float Y

{

get

{

return y;

}

set

{

y = value;

}

}

public float Z

{

get

{

return z;

}

set

{

z = value;

}

}

public Vector3(float x,float y,float z)

{

this.x = x;

this.y = y;

this.z = z;

}

//在unity中的6个方向

//方向是以访问器的形式访问的

public Vector3 forward

{

get

{

return new Vector3(0, 0, 1);

}

}

//零向量

public Vector3 zero

{

get

{

return new Vector3(0, 0, 0);

}

}

//向量的模长

//模长的计算根号下（x\*x+y\*y+z\*z）

public float magnitude

{

get

{

return (float)Math.Sqrt(x \* x + y \* y + z \* z);

}

}

//cocos里叫做向量归一化。normalize

//返回该向量的单位向量，模长为1

public Vector3 normalize

{

get

{

return new Vector3(x/magnitude,y/magnitude,z/magnitude);

}

}

//运算符重载：operator

//重载+-\*使之能够在三维向量间使用

/// <summary>

/// 重载加号使两个向量相加

/// </summary>

/// <param name="v1">相加得向量1</param>

/// <param name="v2">相加得向量2</param>

/// <returns>相加后的新向量</returns>

public static Vector3 operator +(Vector3 v1,Vector3 v2)

{

return new Vector3(v1.x+v2.x,v1.y+v2.y,v1.z+v2.z);

}

public static Vector3 operator -(Vector3 v1, Vector3 v2)

{

return new Vector3(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z);

}

//乘号 向量和一个数组相乘

public static Vector3 operator \*(Vector3 v,float num)

{

return new Vector3(v.x \* num, v.y \* num, v.z \* num);

}

public static Vector3 operator \*(float num, Vector3 v)

{

return new Vector3(v.x \* num, v.y \* num, v.z \* num);

}

//重载比较运算符时，需要成对重载

//重载了<就一定要重载>号

//API部分

/// <summary>

/// 向量点乘

/// </summary>

/// <param name="v1">点乘的向量1</param>

/// <param name="v2">点乘的向量2</param>

/// <returns>向量1相对于向量2的投影长度</returns>

public static float Dot(Vector3 v1,Vector3 v2)

{

return v1.x \* v2.x + v1.y \* v2.y + v1.z \* v2.z;

}

/// <summary>

/// 求两向量的夹角角度

/// </summary>

/// <param name="v1">向量1</param>

/// <param name="v2">向量2</param>

/// <returns>求出来的夹角值</returns>

public static float Angle(Vector3 v1, Vector3 v2)

{

//求得两向量夹角的正弦值

float alpha = Dot(v1.normalize, v2.normalize);

//根据反三角函数求得夹角的弧度

float radius = (float)Math.Acos(alpha);

//将弧度转换为角度返回出去

return (float)(radius \* 180 / Math.PI);

}

/// <summary>

/// 求两向量的投影向量

/// </summary>

/// <param name="v1">向量1</param>

/// <param name="v2">向量2</param>

/// <returns>所求得的投影向量</returns>

public static Vector3 Project(Vector3 v1,Vector3 v2)

{

return v2.normalize \* (Dot(v1, v2) / v2.magnitude);

}

/// <summary>

/// 两向量叉乘

/// </summary>

/// <param name="v1">向量1</param>

/// <param name="v2">向量2</param>

/// <returns>两向量的法向量</returns>

public static Vector3 Cross(Vector3 v1,Vector3 v2)

{

return new Vector3(v1.y \* v2.z - v1.z \* v2.y, v1.z \* v2.x - v2.z \* v1.x, v2.y \* v1.x - v1.y \* v2.x);

}

/// <summary>

/// 求出两向量间的距离

/// </summary>

/// <param name="v1">向量1</param>

/// <param name="v2">向量2</param>

/// <returns>两向量间的距离值</returns>

public static float Distance(Vector3 v1,Vector3 v2)

{

return (v2 - v1).magnitude;

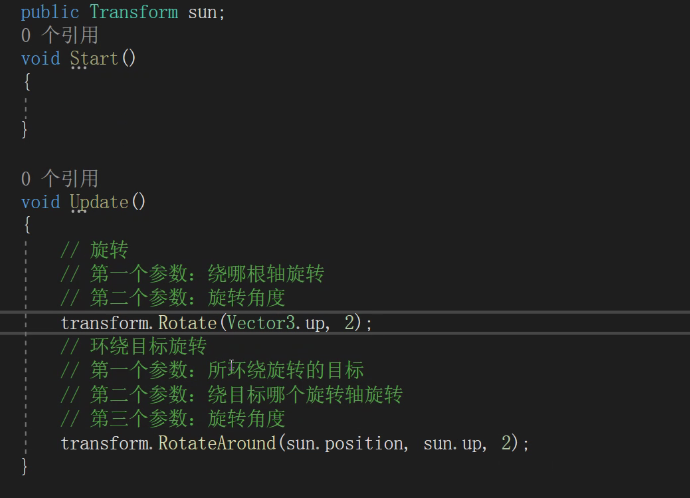
}

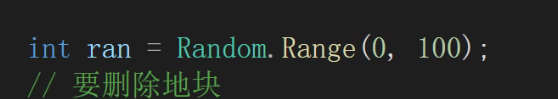
}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

旋转：



随机数：

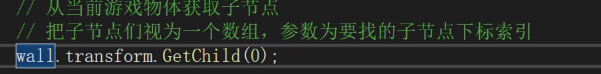
查找节点：



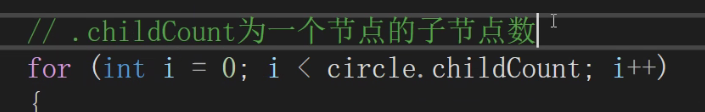
再根据父节点找下面的子节点



或者



.childCount可以拿到子节点数



预制体：

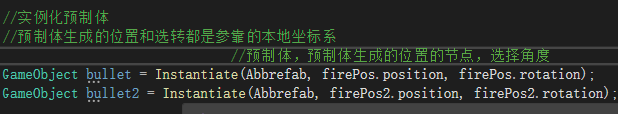
预制体做好后绑定脚本

在需要生成预制体的节点上的脚本声明预制体：



声明需要在哪个节点上生成预制体：





生成完预制体后可用SetParent函数来把预制体节点添加进某个需要的节点下

在transfrom下点出SetParent（）方法，（）里填入父节点的transform

自动瞄准API：

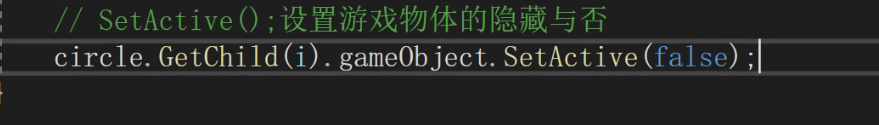
Targret：瞄准的对象



如何设置节点的active？

节点点出SetActive（）函数，（）里填入false或者true来决定该节点是否活跃

.GetChild（i）可以拿到子节点，i为下标



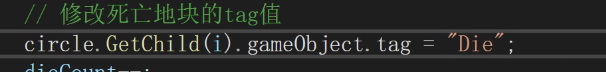
声明一个材质球：

在unity里把材质球拖到对应脚本

之后在脚本中，拿到需要修改节点的组件，之后对应的颜色赋值



修改tag标签；



停止刚体移动：

刚体.velocity = Vector3.zero

刚体和碰撞组件（RigidBody）

1. Mass：质量，影响外力对物体产生的影响效果，质量是物体的固有属性，不会随物体所在的环境改变而改变
2. Drag:阻力（包括空气阻力），摩擦力
3. Angluar Drag: 角阻力，涉及旋转
4. Use Gravity:是否开启重力
5. *Is Kinematic:是否启用动力学*
6. *Interpolate:预处理措施，判断即将碰到的物体*
7. *Colision Detection:碰撞侦测*
8. Constrauints

*Freeze Position:冻结物体的坐标位置*

*Freeze Rotation:冻结物体的坐标位置*

*冻结值物体不会因为收到外力使自身的位置或旋转发送改变，如果直接对position或rotation属性进行设置，依然会发生改变*

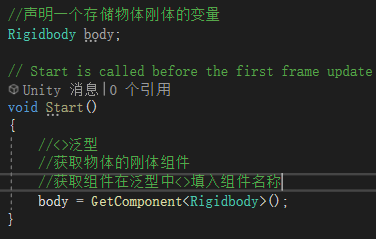
发生碰撞的条件：

1. 一个物体有碰撞体，另一个物体只有碰撞体，是不会发生碰撞的
2. 一个物体有碰撞体和刚体，另一个物体只有刚体，也不会发生碰撞
3. 一个物体有碰撞体和刚体，另一个物体只有碰撞体，会发生碰撞
4. 一个物体有碰撞体和刚体，另一个物体也有碰撞体和刚体，会发生碰撞

两个物体都必须有碰撞体，至少其中一个物体有刚体，才能发生碰撞

\*刚体组件一般是为了实现物理系统的效果才使用的。

拿到组件：刚体组件



利用刚体，给小球一个力，使其可以移动：

//小球向前移动

if (Input.GetKey(KeyCode.W))

{

//向小球添加一个外力

body.AddForce(Vector3.forward\*speed);

}

碰撞生命周期函数

刚体的碰撞函数：（加了刚体碰撞函数的才有）

OnCollisionEnter：在碰撞开始时调用一次

/// <summary>

/// 碰撞 生命周期函数

/// </summary>

/// <param name="other">发生碰撞的东西</param>

private void OnCollisionEnter(Collision other)

{

//判断碰撞的物体标签：

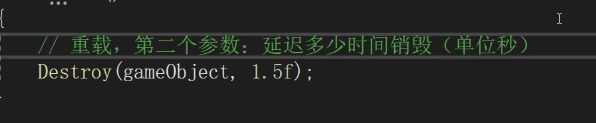
if (other.gameObject.tag == "Money")

{

//销毁游戏对象

Destroy(other.gameObject);

延迟销毁：



}

}

OnCollisionExit：在碰撞结束时调用一次

OnCollisionStay:在接触碰撞时持续调用

***Is Trigger:勾选后只调用碰撞函数不产生碰撞结果***

勾选之后刚体碰撞函数无效，改用碰撞体碰撞函数

碰撞体的碰撞函数

注意：被刚体碰撞的物体需要勾is Trigger，刚体则不需要

OnTriggerEnter：在碰撞开始时调用一次

/// <summary>

/// 碰撞体碰撞 生命周期函数

/// </summary>

/// <param name="other">发生碰撞的东西</param>

private void OnTriggerEnter(Collider other)

{

//判断碰撞的物体标签：

if (other.gameObject.tag == "Money")

{

//销毁游戏对象

Destroy(other.gameObject);

注： Destroy(gameObject);

销毁本身

}

}

OnTriggerExit：在碰撞结束时调用一次

OnTriggerStay：在接触碰撞时持续调用

切换场景

引用该集

using UnityEngine.SceneManagement;

切换场景API：

SceneManager.LoadScene(“要切换的场景”);

线性插值：

//Vector3.Lerp(移动的初始位置，移动的目标位置，移动的更新频率)线性插值

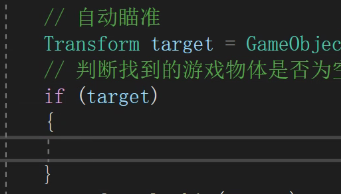
//摄像机移动的目标位置

Vector3 dePos = ball.position + offset;

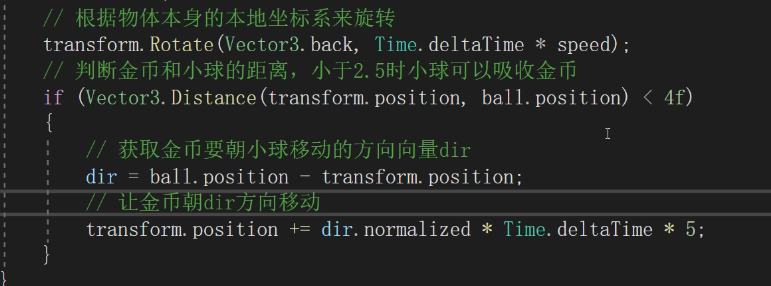
//使用线性插值让摄像机移动平滑

transform.position = Vector3.Lerp(transform.position,dePos, Time.deltaTime);

判断游戏物体是否找到：



吸金币：



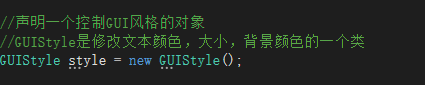
**UI相关**

1. **GUI：通过代码来实现UI**
2. **NGUI：实际上是一个Unity的插件**
3. **UGUI：Unity自带的UI组件**

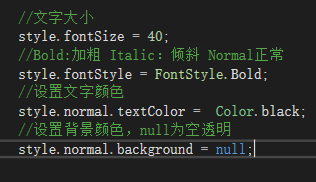
**GUI**的本质是一个生命周期函数，会由系统自动调用

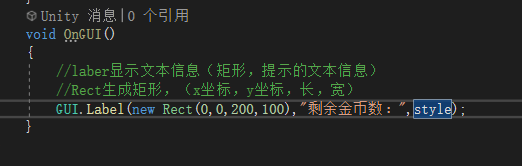
Private void OnGUI()

UI坐标系原点在左上角



想要设置字体属性，需要先声明一个style，完成设置之后在GUI.Label最后添加参数



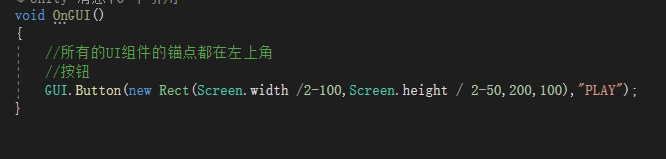


屏幕坐标系：原点在左上角

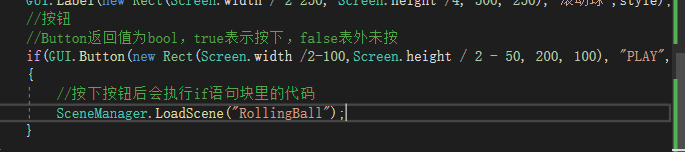
Screeen.width:屏幕宽度

Screen.height:屏幕的高度

按钮：



Button本身返回值是一个bool，所以可以直接当回调函数用



退出游戏API

Application.Quit();

音频组件

AudioSource组件

AudioClip:要播放的音频文件

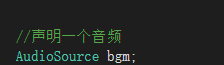
Output:输出设置

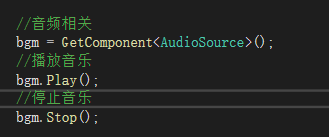
Mute：静音

Play On Awake：在开始时就播放音乐

Loop：循环

在Unity中播放音频还需要AudioListener组件用于接收音频





物理材质：

Physic Material

右键创建物理材质

把创建好的物理材质拖到物体的包围盒里



材质属性：

Dynamic Friction:动态摩擦

Static Friction:静态摩擦

Bouncinexx:弹力系数【0,1】0为没有弹力，1为不会损耗弹力

Fricition Combine:摩擦力组合效果

Bounce Combine：弹力组合效果

旋转

屏幕坐标系：原点在左下角

欧拉角

1. 章动角：X
2. 自旋角：Y
3. 旋近角：Z

四元数（x,y,z,a）

旋转根据

//鼠标按下时的位置

Vector3 firstPos;

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

//按下鼠标时执行

if (Input.GetMouseButtonDown(0))

{

//记录鼠标点击的初始位置

firstPos = Input.mousePosition;

}

//按住鼠标时执行

if (Input.GetMouseButton(0))

{

//根据按下鼠标滑动得出鼠标的终点位置

Vector3 desPos = Input.mousePosition - firstPos;

//使用物体的欧拉角进行旋转（欧拉角是三维向量可以进行加减）

//欧拉角（x,y,x）分别对应物体xyz轴的旋转角度

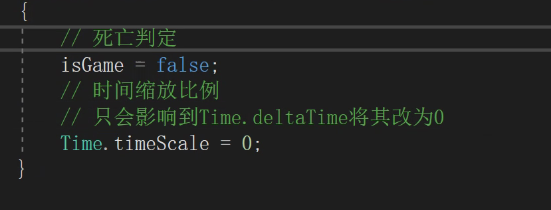
transform.eulerAngles -= Vector3.up \* desPos.x \* 1;

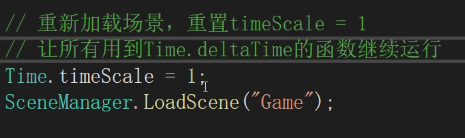
firstPos = Input.mousePosition;

}

}

游戏暂停





shift + Ctrl + F固定视角（摄像头）

预制体

Open Prefab Asset:打开预制体

Select Prefab Asset:选中预制体资源

Unpack Prefab： 解除预制体

Unpack Prefab Completely:完全解除预制体

预制体：预先制作好的游戏对象，类似于基类（父类）

1.在Unity中，不能再预制体的外部删除预制体的子节点，只能在Prefab Mode预制体模式下进行修改

2.蓝色的竖线，表示这个属性、组件被修改了，是子类独有的属性，父类不会有，称之为重写

3.嵌套预制体，修改被嵌套的父预制体，会影响到所有嵌套了该预制体的子预制体；在预制台中修改被嵌套的子预制体，不会影响到被嵌套的父预制体，同样属于重写

4.预制体变体（Prefab Variant），本体发生改变，且变化过的不是变体修改过的属性，会应用到变体上，但变体修改的属性不会应用到本体上

5.预制体上的脚本不能直接拖资源清单中的游戏物体，只能通过GameObject.Find（）方式去获取，但是预制体上的脚本可以直接拖同为预制体的其他游戏物体

射线系统

RaycastHit：检测射线碰到的物体

1.collider:物体的包围盒

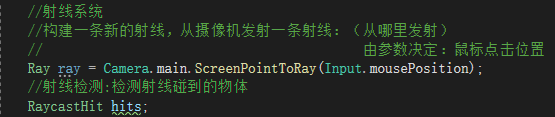
2.point:射线于物体的相交点

3.normal：射线的法向量

4.distance：射线射到物体后由起点到终点的距离

5.transform：射线所碰到的游戏物体的transform属性

6.rigidbody：射线碰到的游戏物体的刚体属性

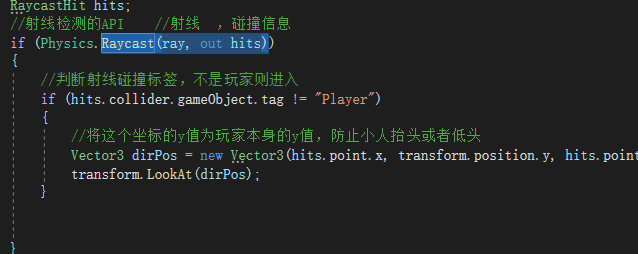


一般来说一个函数只能有一个return一个返回值

out关键字可以让函数有多个返回值

1.Raycast(ray, out hits)

一般会追加额外的if判断



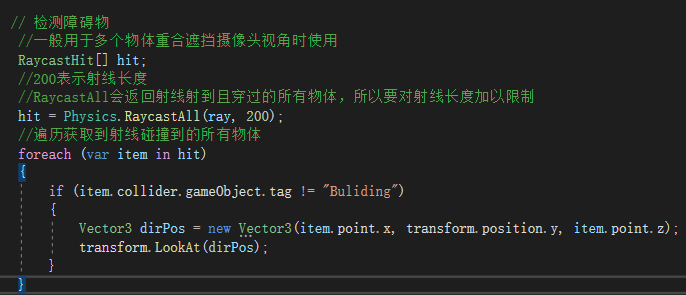
1. RaycastHit[] hit;

hit = Physics.RaycastAll(ray,200);

拿到射线碰撞且穿过的所有物体

然后遍历这个数组中物体再作筛选

一般用于摄像机和人物间的物体隐藏



3.

Raycast(ray, out hits, 200, layerMask)

int layerMask = LayerMask.GetMask("Player");（获取对应名字的层级）

一般不追加额外的if函数

通过层级遮罩去忽略特定层级的物体



计时器：（这里只阐述最常用的方法）

//声明一个时间，一个到达的时间

float timer;

float interval;

//start里赋值

timer = 0;

interval = 5f;

//updata里

//每5秒生成一波敌人：使用timer计时器来完成

timer += Time.deltaTime;

if (timer >= interval)

{

timer = 0;

}

延迟调用函数：

Invoke("函数名字", 调用间隔);

画图函数

private void OnDrawGizmos()

{

for (int i = 0; i < ways.Length-1; i++)

{

//设置辅助线的颜色

Gizmos.color = Color.white;

//在每个路径点间画一跳辅助线

//1.线的起始位置2.线的终点位置

Gizmos.DrawLine(ways[i].position,ways[i+1].position);

}

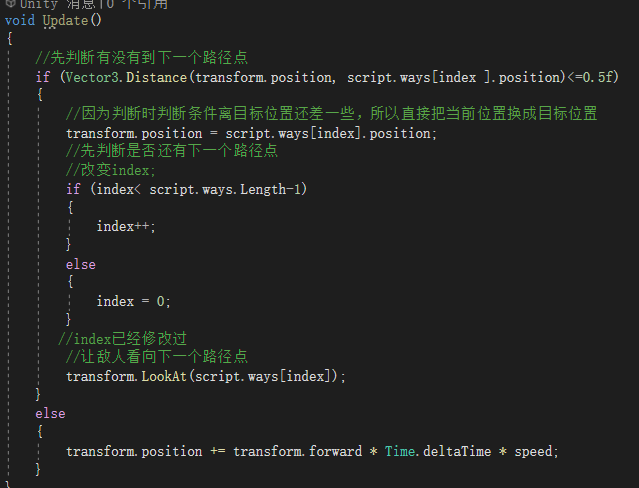
}

路点寻路：

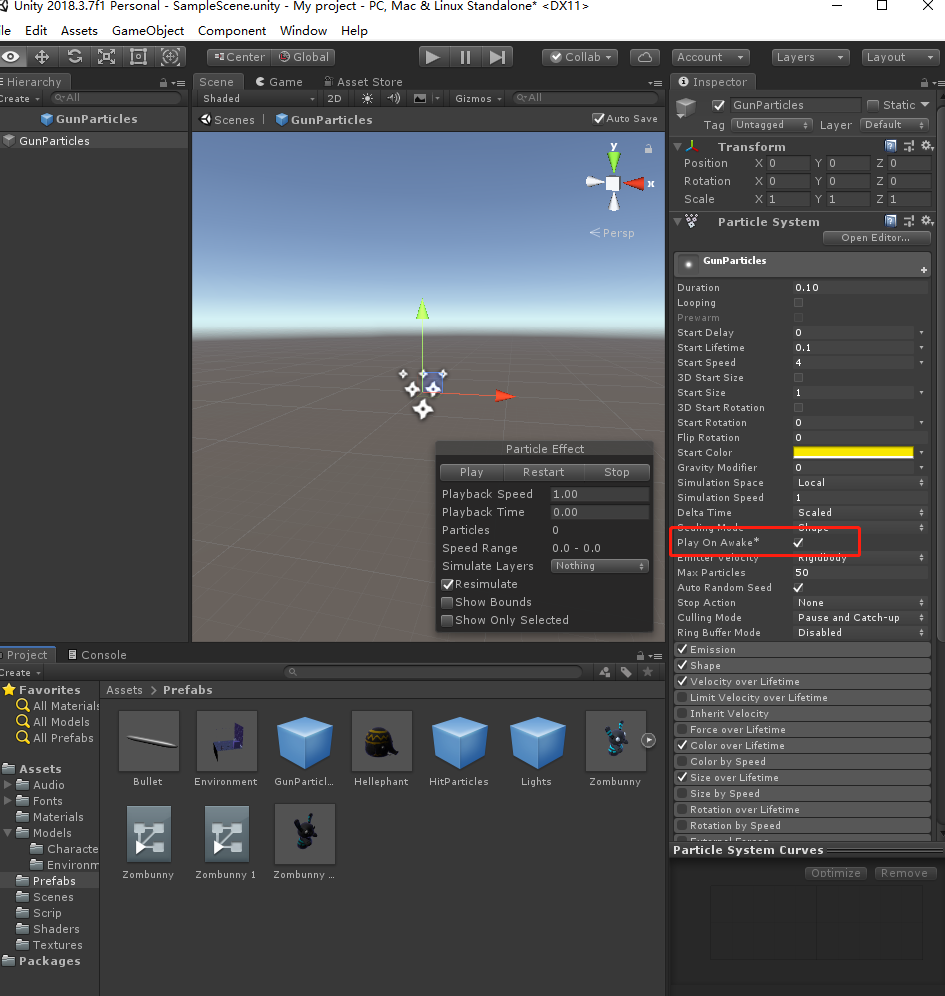
先在游戏中生成空节点，空节点内创建多个路点，之后脚本中声明坐标数组，把路点信息赋值进去

之后在预制体脚本中拿到刚刚赋值的节点位置

面朝下一节点位置，判断自身位置是否到达节点位置，到达的话再看下一节点位置



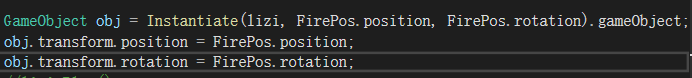
粒子特效：

首先进入粒子特效的预制体把awake勾上，自主调用：

在玩家脚本内拿到粒子组件



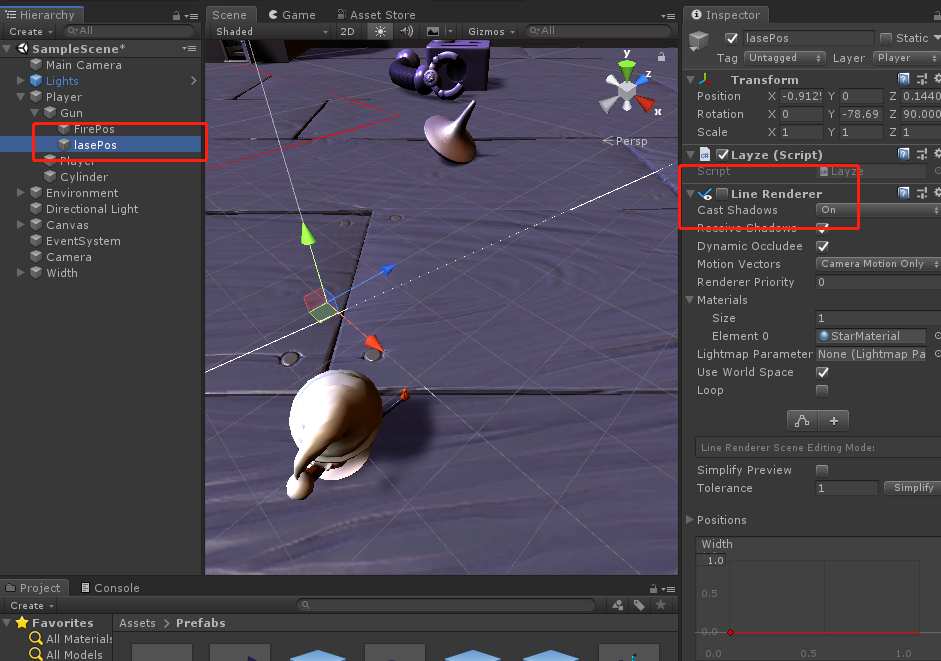
实例化粒子预制体：并设置位置



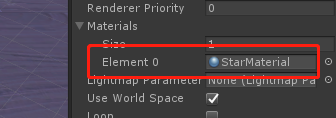
激光：

利用射线系统：

先在相应节点添加激光组件：

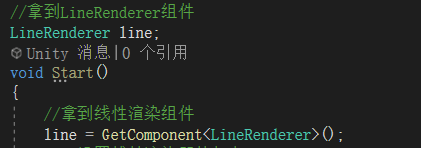


注意，因为一开始不需要激光，所以吧这个组件的active为false；

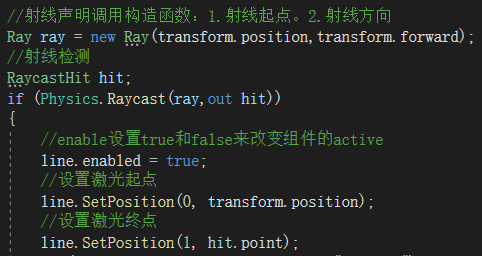
激光添加材质：

给这个激光所在的节点添加脚本：

并在脚本内拿到激光组件：

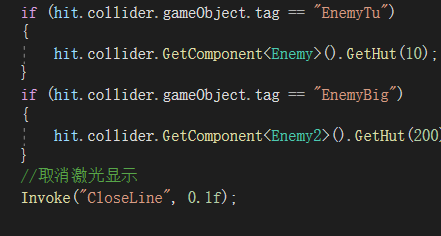


利用射线系统，由射线的起始位置和终点位置赋值给激光的起点位置和终点位置：

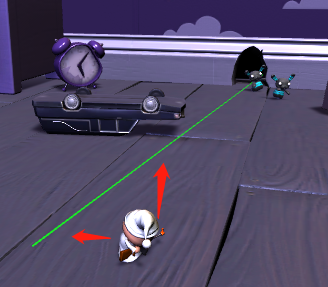


注：可用enabled来打开和关闭组件的active

可以利用射线的碰撞检测，来做到给激光赋值伤害



注意：如果不销毁的话，没有第二道激光，则第一道激光一直存在

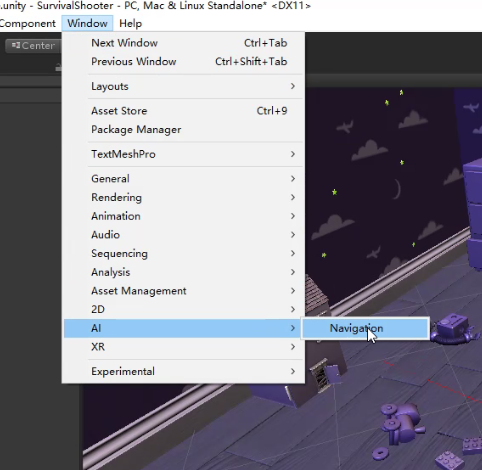
直到发射第二道激光为止，所以要销毁激光。

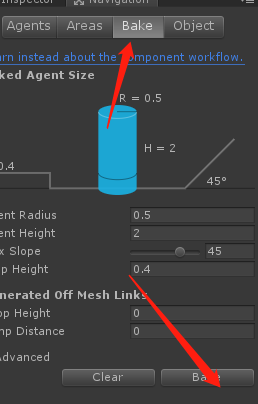
自动寻路Navigation

以A\*算法为底层的AI导航

注意：Window→Ai→Navigation

在自动寻路前必须烘焙场景，但只有静态的物体才能烘焙，烘焙完成后会创建一个和场景名相同的文件夹，里面放的就是烘焙好的地图数据





1. Agents：自动导航代理

NavMeshAgent

\*StepGHeight：步高

1. Areas：区域花费

-Name代表区域名字，cost代表区域花费

1. Bake：烘焙属性

-AgentRadius：可通行区域到物体边框的距离

-AgentHeight：可通行区域的高度

-MaxSlope：可通行区域的最大坡度

StepGHeight：步高，可通行区域最大台阶高度

步高不能大于AgentHeight属性；

-DropHeight:最大下落高度

-JumpDistance：最远跳跃距离

1. Object对象，选择带有static静态的物体
2. 在object里修改

NavMeshAgent组件

AgentType:寻路代理类型

BaseOffest：包围盒偏移量

Speed：移动速度

AngularSpeed：角速度：涉及旋转

Acceleration：加速度

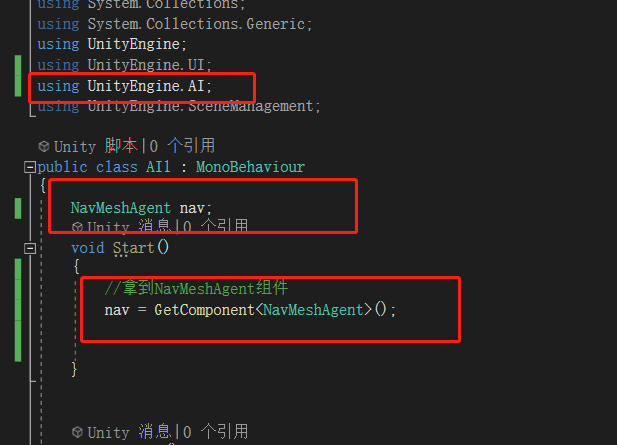
StoppingDistance：停下时距离目标距离0

Quality：寻路质量

Priority：寻路优先级（0-99）:值越低优先级越高

创建需要移动物体的脚本并绑定

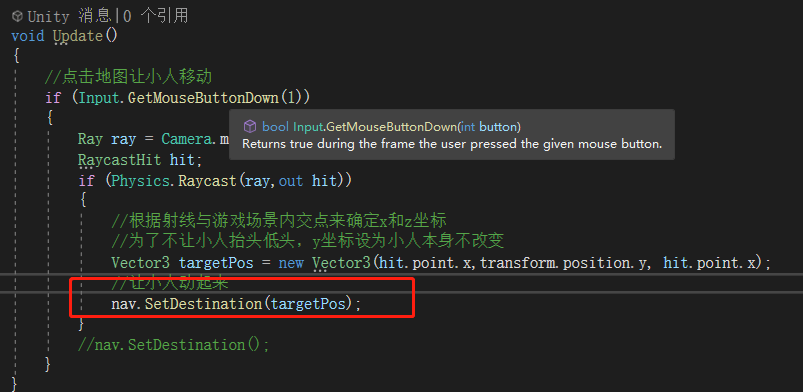
添加using UnityEngine.AI;引用集



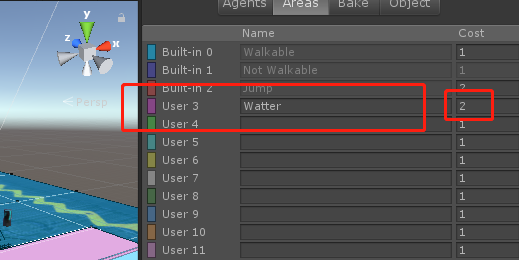
最常用API“SetDestination（参数）”

参数为自动寻路的目标点

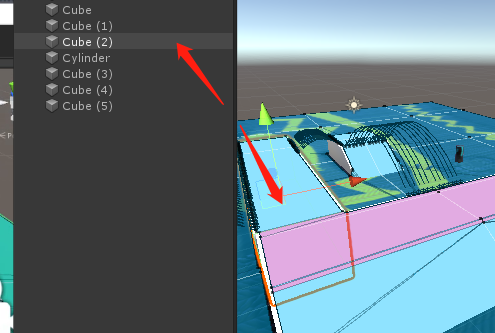
利用右键点击，摄像头发射一条射线，判断射线的终点位置为人物需要达到的目标地点



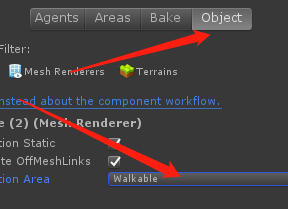
如果想要修改地图移动的类型，与时间，则：



在Areas下添加类型以及通过这个类型的对象要多少时间



之后选中需要修改的游戏对象

点击object，下面选择你刚刚添加的类型，最后重新烘焙

烘焙完后，在需要移动的对象上添加组件

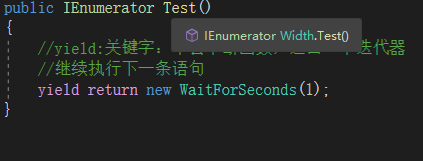
**协程：在主程序运行时，再开启一段新的程序**

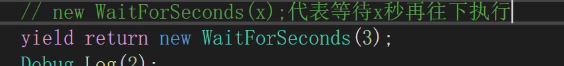
**注意：开启协程不等于开启线程**

**进程:正在执行的程序就是一个程序**

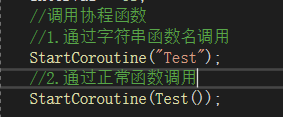
**线程：进程中一个单一的连续的控制流程。**

**一个进程可以同时有多个线程**

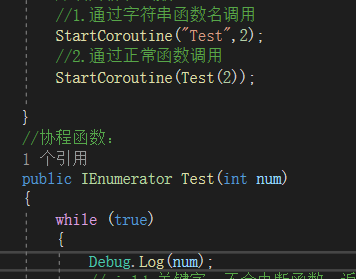




调用：

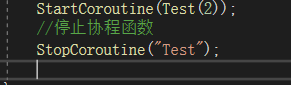


比延迟函数好用的是，可以传参：



停止**停止协程函数要做到对应，什么形式开启，什么形式停止**

**//StopCoroutine只能停止以字符串开启的协程**

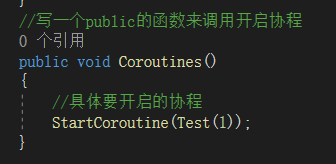


或者停止所有的协程函数：

StopAllCoroutines();

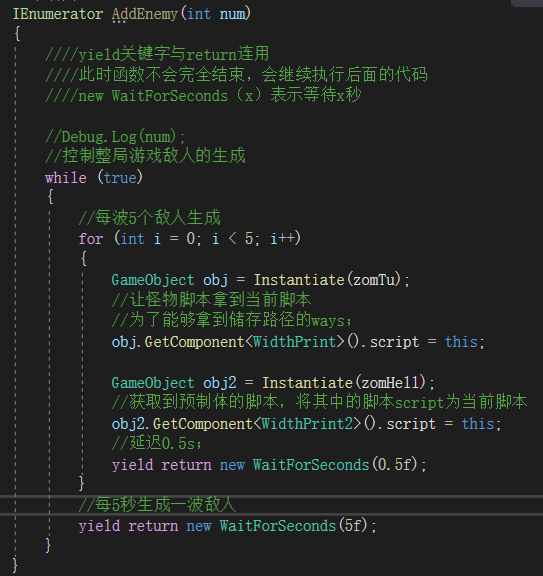
协程的特点：

1. 多个协程（无论是否同名）是可以同时执行的，根据协程开启的顺序先后执行
2. StopCoroutine（）函数只能停止由字符串形式启动的协程函数，并且会关闭所有同名函数（只能关闭由字符串形式开启的函数）
3. StopAllCoroutines()函数会停止所有已开启的协程函数，无论是什么方式开启的，都会被停止
4. 协程可以嵌套
5. 协程函数传参时不能使用ref和out关键字，但是可以传引用类型
6. 协程不是多线程
7. 多个脚本可以调动同一协程，通过public关键字实现



1. 禁用协程所在的脚本（脚本.enable = false）不会停止该脚本所开启的协程，但可以通过关闭脚本所在的游戏对象（游戏对象.SetActive(false)）来停止协程
2. 就目前来说。在Unity中没有一个简易的办法可以检测到已开启的协程个数，以及协程的作用对象和协程的状态
3. 把运算量大的计算（算法），放到一个随时间变化的协程中

用协程生成敌人



在切换场景时，使用StopAllCoroutines（）停止本场景开启的所有协程

Awake（）生命周期函数

游戏场景编译脚本时执行一次，速度最快

Start();脚本运行时调用一次，速度仅慢于Awake

OnEnable()脚本启用时调用（反复调用）

OnDisable()脚本禁用时调用（反复调用）

Updata():每帧执行，，，帧（Tiame.deltaTime）;

OnDestroy():当前脚本游戏对象销毁时调用一次。一定要使用Destroy销毁，才会调用