Unity 3D游戏编程自学#4——Unity 3D进阶

原创 Zerg_Wang 于 2019-02-23 15:48:49 发布 425 收藏 2 编辑 版权

分类专栏： Game Programing

1.UI

即User Interface，用户在游戏中的操作界面，目前Unity中主要的UI系统有NGUI和UGUI，前者目前使用率较高，后者是Unity在4.6版本后自带的，版本较新，使用率渐增。

除了以上两种UI系统，还有极少使用的OnGUI和Legacy GUI，前者可用于Unity引擎的界面拓展，后者使用极为简单，可快速实现一些简单的UI界面。

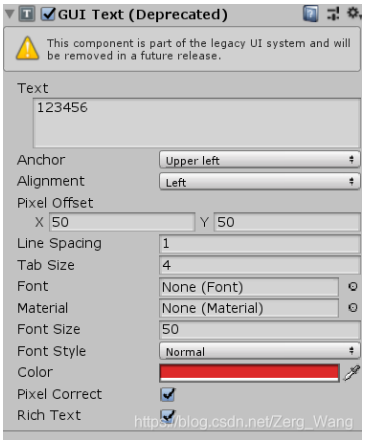
UI界面可剖析为图片和文字的组合：

文字：GUI Text

旧版的UI文字系统，使用方法：

首先创建一个空物体，然后为其添加组件：Component→Rendering→GUI Text

Text栏输入想要显示的文字，Pixel Offset调整文字位置，Font和Color调整大小和颜色。



注意：在Scene中是看不到文字的，只能在Game中面板中显示。

如果在Game中无法显示文字，问题出在以下方面：

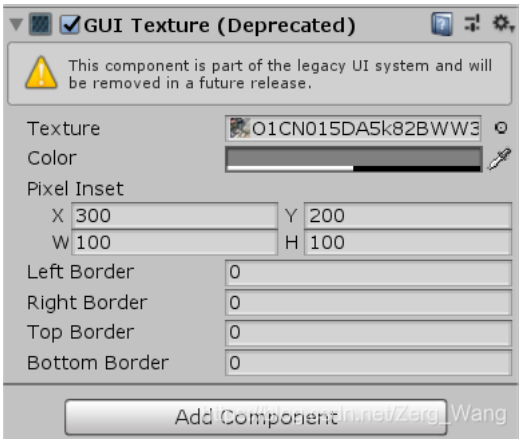
1. 文字位置在摄像机视线之外。（要解决这个问题，先把空物体的位置重置，即x，y，z均为0，然后调节Pixel Offset的x，y范围在50到500以内，这是因为这个Pixel Offset是根据屏幕分辨率确定了，x表示与屏幕左侧距离多少像素，y表示与屏幕下侧距离多少像素，假如分辨率为1920×1080并要UI显示在左上角，可能x，y的取值分别为50，1050左右）

2. 文字大小（Font）为零。

3. 没有为主摄像机添加GUI Layer组件。（Component→Rendering→GUI Layer）

图片：GUI Texture

旧版的UI图片系统，使用方法：创建空物体，为其添加组件：Component→Rendering→GUI Texture



Texture项就是我们要显示的图片，我们可以事先把图片放在Assets的Textures文件夹下。

Color是图片颜色，默认的灰色不影响原图的颜色显示。

Pixel Inset中的X、Y调整图片位置，W、H调整图片大小。

该图片同样无法在Scene中看到。

鼠标事件

不同于之前使用的获取鼠标信息的方法，这里的鼠标事件是对于某个物体说的（而不是全局），编写鼠标事件的脚本是挂载在游戏物体上的，当鼠标移动到游戏物体上进行操作时，相应的鼠标事件才会触发。

有以下鼠标事件方法：（它们与Start和update方法同级，写在这些方法之外）

```
void OnMouseEnter()
{
    ExitTransform.transform.localScale= new Vector3(0.6f, 0.6f, 0.6f);
}

void OnMouseExit()
{
    ExitTransform.transform.localScale = new Vector3(0.5f, 0.5f, 0.5f);
}

void OnMouseDown()
{
    Application.Quit();
}
```


当鼠标移动到物体中时，OnMouseEnter方法触发，当鼠标移开，OnMouseExit触发，当鼠标在物体中且点下左键时，OnMouseDown触发，以上代码的作用是：

将一张退出标志作为Texture，如果鼠标移动到退出标志，标志会变大，移开则变回原大小（原来的Scale均为0.5），点击会使程序退出。

2.特效

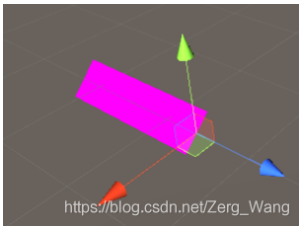
拖尾特效：Trail Renderer

为子弹、流星等高速运动的物体作出尾迹的效果。

创建尾迹：

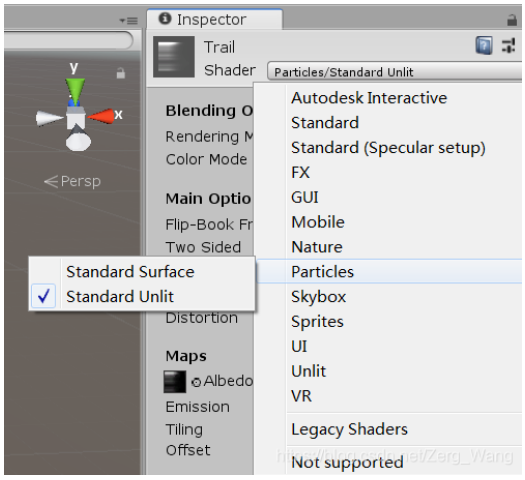
创建空物体，添加拖尾特效组件（Component→Effects→Trail Renderer）。

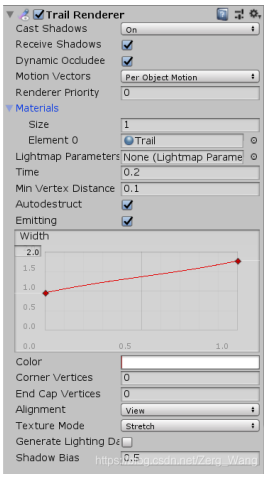
创建完毕后在Scene中拖动空物体，即可看到粉红色的尾迹（此时没有为尾迹添加材质，默认为此种粉红色尾迹）



添加尾迹材质：（注：因为软件版本原因，本人所看课程中的做法与下文不同，下文为本人认为较为正确的方式，若有错误还请指出）

将做好的尾迹贴图放于Textures文件夹，创建材质球，设置其模式为：





并设置Rendering Mode为Additive，Color Mode为Multiply，并将Textures中的尾迹贴图应用于Maps项，最后将设置好的材质球应用于尾迹中（Trail Renderer组件中的Materials项）。

在该组件下，还可以调整尾迹持续时间（一般很小，0.2左右）、尾迹扩散方式（通过调整Width项的曲线）以及颜色。

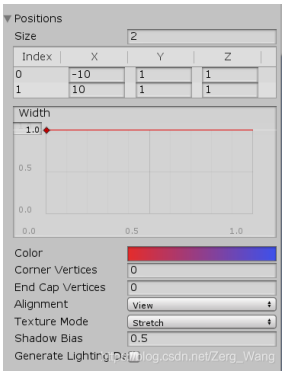
线特效：Line Renderer

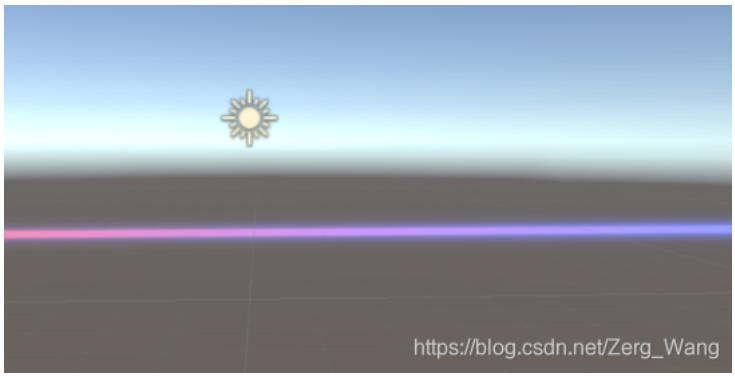
常用于射击游戏中红外线、激光等的表示。

创建空物体，添加线特效组件（Component→Effects→Line Renderer）。

然后与拖尾特效相同的方式添加材质球。

与拖尾特效不同的是，需要设置线的起点和终点，此外还可以设置其颜色：





3.声音

游戏中需要各种音效、背景音乐来配合，因此在Unity有音频编辑功能。

在Assets中创建Audios文件夹存放音频。

创建音频组件：创建空物体，添加Component→Audio→Audio Source，该音频组件在游戏中相当于一个喇叭，在Audio Clip中可设置要播放的音频。

该组件还有一个属性：Spatial Blend，可将声音设置为2D或者3D，若声音为3D，则在游戏中距离该组件越远，声音越小，若是2D则音量不随距离而改变。

有了喇叭还无法听到声音，还需要耳朵——Audio Listener，这个组件是默认挂载在摄像机上的，且没有属性可以设置。此外，上文提到的3D声音音量的变化，就是由“喇叭”与摄像机（Audio Listener）的距离所决定的。

这里编辑一个脚本，用空格键控制声音的播放与暂停：

```
private AudioSource musicPlay;
bool f = false;

void Start()
{
    musicPlay = gameObject.GetComponent<AudioSource>();
}

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        if (f)
        {
            musicPlay.Play();
            f = false;
        } else
        {
            musicPlay.Pause();
            f = true;
        }
    }
}
```

如果是想为游戏添加循环播放的背景音乐，可直接在Main Camera处添加，然后勾选Play On Awake和Loop即可。

4.常用API

1.游戏物体的实例化与销毁

有时我们需要即时生成或销毁一些游戏物体，此时要用到以下方法：

例如，在白色地板上，每按下以此空格，则随机生成一个红色方块：

首先创建地板，然后创建红色方块并设置为预制体，接下来编写脚本：

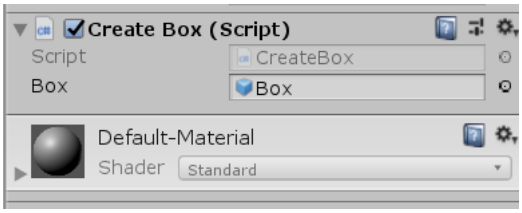
```
public class CreateBox : MonoBehaviour
{
    public GameObject box;           //指定要操作的预制体
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Vector3 position = new Vector3(Random.Range(-4f, 4f), 3, Random.Range(-4f, 4f));
            //在地板范围内生成随机坐标，Random.Range(min,max)用于生成min到max范围内的一个随机数
            GameObject.Instantiate(box, position, Quaternion.identity);
            //该方法的三个参数分别是要实例化的预制体、预制体的位置、预制体的旋转情况，这里的写法表示无旋转
        }
    }
}
```


实际上实例化方法还可以进行赋值：

```
public GameObject myBox;
myBox = GameObject.Instantiate(box, position, Quaternion.identity) as GameObject;
```

此处GameObject.Instantiate返回的是Object类型，而myBox是GameObject类型，所以要as GameObject进行强制转换。

然后我们将该脚本挂载到游戏物体上，这里可以挂载在地板上：



注意：这里还需要手动指定要实例化的预制体：要将perfabs文件夹中做好的预制体拖拽至上图的Box项。（注意：在定义预制体时一定要用public，否则此处无法拖拽）

在实例化后还可以给这些物体不通过材质球而进行“上色”：

```
myBox.GetComponent<MeshRenderer>().material.color = new Color(0.2f, 0.7f, 0.4f);
```

物体的颜色由MeshRenderer组件确定，因此要通过这个组件下的方法来确定物体颜色，颜色由三个范围是[0, 1]的数组成，代表RGB。

接下来是游戏物体的销毁，这个脚本需要挂载到预制体上：

```
void Start()
{
    GameObject.Destroy(gameObject, Random.Range(2, 5));
}
```

前一个参数为游戏物体，后一个参数为一个实数，代表实例化后多久销毁该物体（也可以不填，表示立即销毁）。

2.Invoke函数

是一个调用其他函数的函数（仅限同一脚本中），用法举例：每隔2秒自动生成一个游戏物体，直接在上面的CreateBox脚本中修改：

```
public class CreateBox : MonoBehaviour
{
    public GameObject box;
    void AutoCreateBox()
    {
        Vector3 position = new Vector3(Random.Range(-4f, 4f), 3, Random.Range(-4f, 4f));
        GameObject.Instantiate(box, position, Quaternion.identity);
    }
    void Start()
    {
        InvokeRepeating("AutoCreateBox", 0, 1);
    }
}
```


首先将要反复使用的代码编写成函数，然后使用Invoke函数调用：

Invoke(string, float)，前一个为要执行的函数名称，后一个为多少秒后执行该函数。（因此调用函数时无法传递参数）

InvokeRepeating(string, float, float)，前一个为要执行的函数名称，后一个为多少秒后执行该函数，最后一个表示以后每隔多少秒执行该函数。然而，InvokeRepeating函数相当于一个开关，执行一次，之后每隔数秒就会重复执行，因此不能将该函数写在Update中。

两种Invoke函数的区别是，前者只执行一次，后者执行无限次。

注意：如上图所示，要调用的函数名应用双引号引起来，且没有括号。

若要取消Invoke以及InvokeRepeating函数的重复调用，有：CancelInvoke()

3.SendMessage函数

gameObject.SendMessage(string)：通知该游戏物体挂载的脚本文件中的某个方法执行（参数为方法名）。

与Invoke不同，Sendmessage可以跨脚本调用函数，也可以带参数。

用法举例：

以下脚本实现的功能：控制方块WASD移动，在移动的时候使用GUIText显示其移动方向：

挂载在GUIText的脚本：（为了之后可在别的脚本调用ShowDirection，要将其改成public形式）

```
private GUIText Score;

void Start()
{
    Score = gameObject.GetComponent<GUIText>();
}

public void ShowDirection(string dir)
{
    Score.text = dir;
}
```


挂载在方块上的脚本：

```
private Rigidbody cubeRigidbody;
private Transform cubeTransform;
private GUIText Score;
void Start()
{
    cubeRigidbody = gameObject.GetComponent<Rigidbody>();
    cubeTransform = gameObject.GetComponent<Transform>();
    Score = GameObject.Find("text").GetComponent<GUIText>();
}
```


首先要调用游戏物体的组件，因为该脚本挂载的是方块，所以要用GameObject.Find方法来找到GUIText来挂载脚本中的Score组件。

```
if (Input.GetKey(KeyCode.W))
{
    cubeRigidbody.MovePosition(cubeTransform.position + Vector3.forward * 0.1f);
    Score.SendMessage("ShowDirection", "上");
}
```

用法：某游戏物体.SendMessage，第一个参数为游戏物体挂载的脚本中要调用的函数名，如果要调用的函数只有一个参数，可直接接在后面，若有多个，则要转换成GameObject类型进行传递。也就是说，SendMessage的第二个参数是GameObject类型的，但如果要调用的函数参数只有一个，可直接写上该参数。

更改如下：

```
public GameObject obj;

public void ShowDirection(object[] obj)
{
    Score.text = obj[0]+obj[1].ToString();
}
```

```
object[] message = new object[2];
message[0] = "上";
message[1] = 2;
Score.SendMessage("ShowDirection",message);
```

4. 协同程序

一般而言，脚本中的代码都是顺序执行的，但有时有些代码需要同时执行。除了多线程，在Unity中还可以使用协同程序来实现这一功能：

```
void Start()
{
    Debug.Log("1");
    Debug.Log("2");
    StartCoroutine("DelayNoMore");
    Debug.Log("4");
}

IEnumerator DelayNoMore()
{
    yield return new WaitForSeconds(2);
    Debug.Log("3");
}
```


如图所示，正常的执行顺序是1234，然而DelayNoMore函数采用了协同，因此程序的执行顺序是124，然后等待2秒执行3。

IEnumerator是协同函数的返回值类型，yield return 这一行是协同函数的返回格式，同时实现了隔两秒执行3的功能。

编写完协同程序，还需要StartCoroutine()来调用它（如果直接调用DelayNoMore，则函数内部代码将一直不会执行）

在协同程序执行的时候，可使用StopCoroutine()命令停止协同程序，与StartCoroutine一样，括号中填入协同函数名。

（实际上该方法有多个重载，这里先学习这一种）

5. Screen屏幕类

用于查看游戏屏幕的相关信息。

```
Debug.Log(Screen.width + " " + Screen.height);
```

可直接输出查看，Screen.width是屏幕宽度，Screen.height是长度，返回的是int型数值。

在游戏编写中，输出值为Game窗口的尺寸，而在游戏打包发布后输出的就是实际运行窗口的尺寸。

6. Time时间类

Time.time，表示游戏运行到当前过去的时间（秒数）。

Time.deltaTime，表示渲染完上一帧画面所用时间（秒数），可用于实现倒计时的效果。

Time.timeScale，表示时间缩放，值为1时游戏正常运行，值为0时游戏暂停，值为0.5表示游戏慢放一倍。

（实际上，即使更改了timeScale，Update的执行速度并不会受影响，它影响的是游戏中所有与时间有关的函数，例如游戏物体的自由落体）

以上函数返回值均为float类型。

7. Mathf数学类

Mathf.Abs() 绝对值　　Mathf.Max() 最大值　　Mathf.Min() 最小值　　Mathf.Round() 四舍五入

重点：插值运算（可用于平滑过渡）

Mathf.Lerp(float a, float b, float c)，返回值为a+(b-a)*c，其中c的范围为[0, 1]，超过1按1算，小于0按0算。

```
float a=0;
void Update()
{
    a = Mathf.Lerp(0, 10, 0.1f);
    Debug.Log(a);
}
```

如图，即可实现从0开始向10递增，递增速度会越来越慢，最后a会无限逼近10。

5. 脚本生命周期

C#脚本一般挂载在游戏物体上，当游戏物体被实例化后脚本的生命周期就开始了，直至游戏物体被销毁。

脚本的生命周期以以下顺序进行：

```
void Awake() {}         //唤醒事件，仅执行一次

void OnEnable() {}      //启用事件，当脚本组件被启用的时候执行一次（游戏物体被实例化时
                                                    //脚本组件会被调用，因此会执行一次）
void Start() {}         //开始事件，仅执行一次

void FixedUpdate() { } //固定更新事件（0.02秒执行一次），会反复执行直到游戏物体销毁或组件停用
                                                                //所有物理组件的更新会在此进行

void Update() { }       //更新事件，与上面的FixUpdate一同反复进行，每帧执行一次

void LateUpdate() {}    //稍后更新事件，反复执行，但每次会在Update之后执行

void OnGUI() {}         //GUI渲染事件，执行的次数是Update的两倍

void OnDisable() { }    //禁用事件，仅执行一次，在物体销毁或组件停用的时候执行

void OnDestroy() { }    //销毁事件，仅执行一次，在物体销毁时执行          https://blog.csdn.net/Zerg_Wang
```

在脚本中这些函数都是默认编写好的，用户省略不写。此外，系统调用这些函数的顺序已定义好，与用户书写顺序无关。

6. 物理射线

1. 射线

Unity 3D中自带一种类：射线，从一点发射出去并无限延伸，且可以与物体发生物理碰撞。首先看以下代码：

```
private Ray ray;
private RaycastHit hit;

void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (Physics.Raycast(ray, out hit))
        {
            GameObject.Destroy(hit.collider.gameObject);
        }
    }
}
```

Camera.main表示tag设置为MainCamera的摄像机组件的引用
ScreenPointToRay(Vector3)为其下的一个方法，用于创建一条射线，从摄像机处发射至Vector3处。Input.mousePosition为鼠标当前位置，参数中填此可实现向鼠标点击处发射射线。

接下来编写射线碰撞到游戏物体的效果：

RaycastHit：一个结构体，用于存储射线的碰撞信息。

Physics.Raycast()，有多种重载，脚本中所用的参数，前一个为射线，后一个为RaycastHit结构体（编写时一定要带out关键字）,该方法的作用是：检测射线是否与其他游戏物体有物理碰撞，若有返回真，并将碰撞信息存入RaycastHit中。

被射线碰撞到的游戏物体用GameObject.Destroy(hit.collider.gameObject)方法销毁。

2. 子弹效果

实现了上述功能，我们可以进一步作出子弹的效果：

```
private Ray ray;
private RaycastHit hit;
public GameObject prefabBullet;
private GameObject bullet;
private Transform CameraTransform;

void Start()
{
    CameraTransform = gameObject.GetComponent<Transform>();
```

```
}11
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (Physics.Raycast(ray, out hit))
        {
            bullet = GameObject.Instantiate(prefabBullet, CameraTransform.position, Quaternion.identity) as GameOb
            Vector3 dir = hit.point - CameraTransform.position;
            bullet.GetComponent<Rigidbody>().AddForce(dir * 200);
        }
    }
}
```

在摄像机处实例化出子弹，然后朝向点击方向发射（AddForce），这个方向其实就是射线与物体碰撞处的坐标减去摄像机处的坐标。

因此，要得到射线与物体碰撞处的坐标，可直接使用：RaycastHit.point

脚本中有一个较为简单的写法来获取游戏物体的组件：直接对游戏物体GetComponent，不需要再额外定义游戏组件的接受对象。因此，CameraTransitform.position也可以写成：Camera.main.GetComponent<Transform>().position，这样Start就可以整个省却。

另外，在测试以上代码时，我们可以看到射线的效果，但一直看不到射线本身，可通过：

Debug.DrawRay(Vector3, Vector3, color)即可，第一个参数为射线起点位置，第二个为射线发射方向，最后为射线颜色。因此可在原脚本中加入：（注意：这个效果只能在Scene面板中看见）

Debug.DrawRay(cameraPosition, dir, Color.red);

7. 资源导入

游戏编写需要大量的素材（音效、贴图、场景模型等等），在Unity 3D中这些资源若是独立于项目之外的（类似于扩充包），则会以.unitypackage的后缀名存储（类似于rar，zip）。

若要导入这些资源包到当前项目中，可以鼠标选中其然后拖拽到Project面板，也可以在Assets文件夹中右键Import Package，选中Custom Package即可。

同样的，若自己有整理好的资源包，也可右键Export Package导出。

本文部分内容来自擅码网（http://www.mkcode.net）Unity 3D课程，经本人学习、整理得来，若有错漏，欢迎指正！