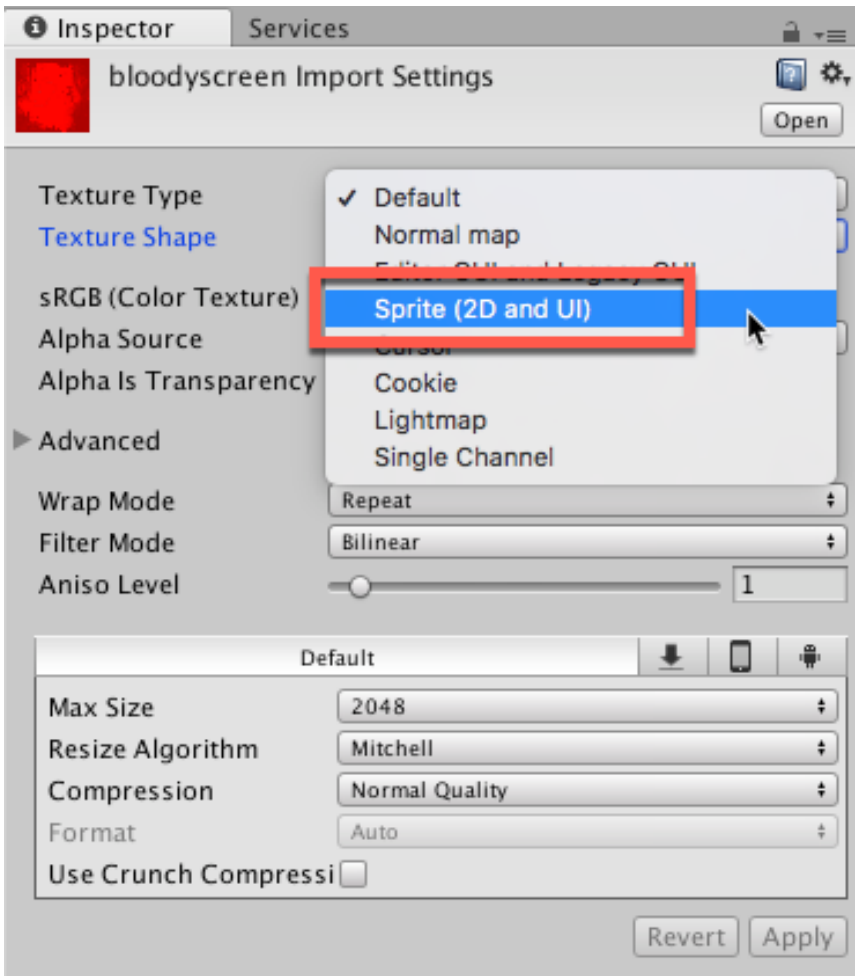
欢迎继续回到我们的学习。

在上一课的内容中，我们给僵尸敌人添加了碰撞检测机制。不过简单的攻击动画效果还是有点单独，在这一课的内容中，我们希望当敌人攻击时，让画面变红，产生一种更紧张的氛围。

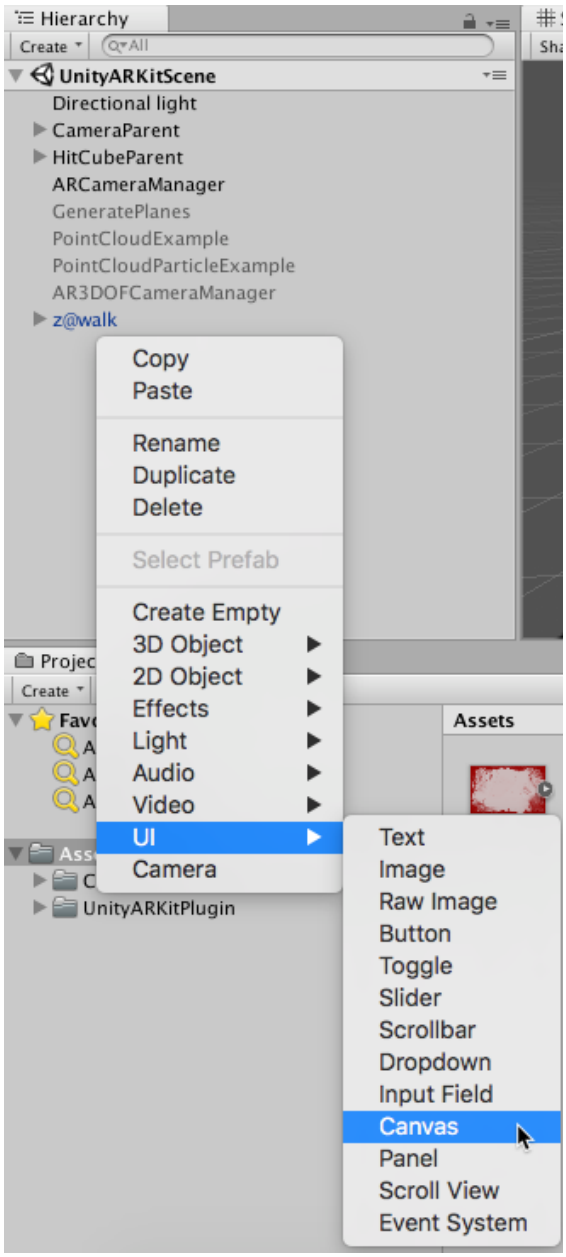
打开浏览器，在Google中搜搜bloody screen，在搜索结果上方点击Images切换到图片。从搜索的图片中选择一种你喜欢的效果就好了。当然，这里只是为了教程的展示需要，如果是开发商业项目，显然需要让美术团队成员来帮忙。

下载完成之后，把图片拖到Assets视图中。选中该图片，在Inspector中把Texture Type从Default更改为Sprite(2D and UI）。



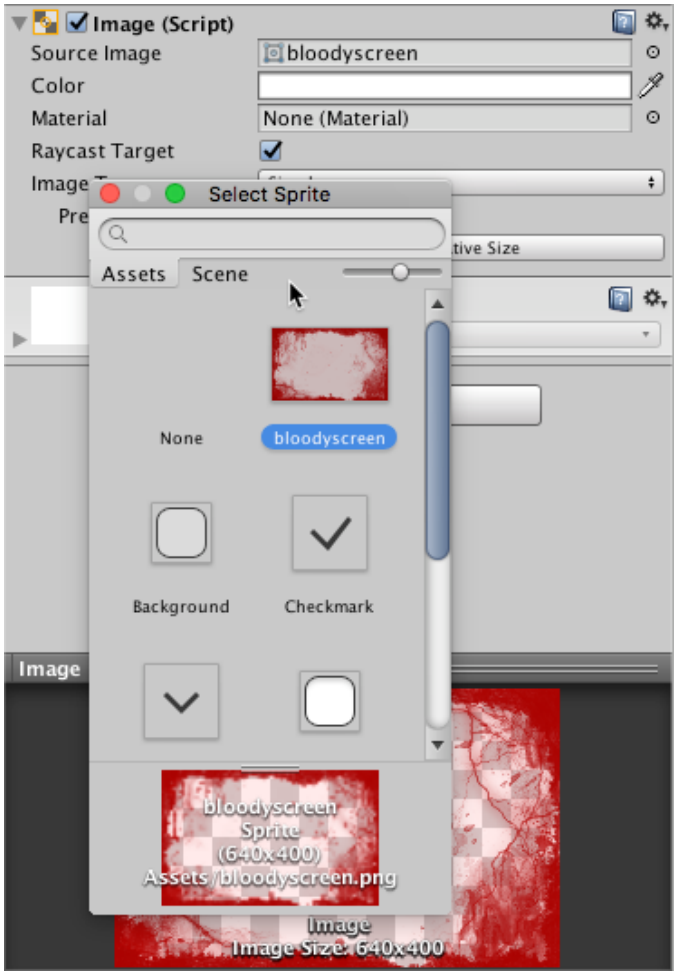
然后点击Apply更改类型。

接下来在Hierarchy视图中右键单击，选择UI-Canvas。

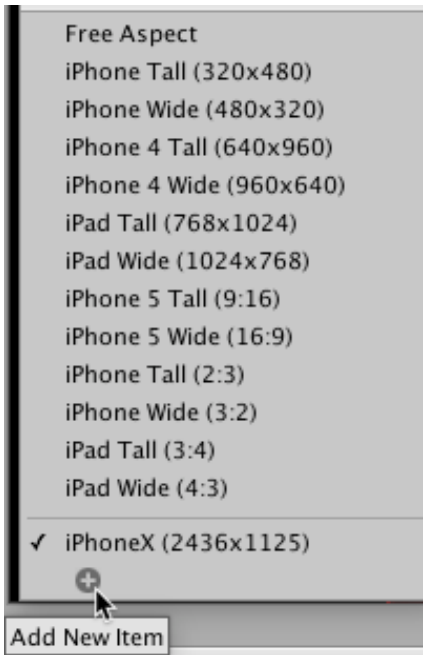


当然，还有另外一种方式，在直接添加某种UI元素的时候，Unity会自动帮我们添加一个Canvas对象。

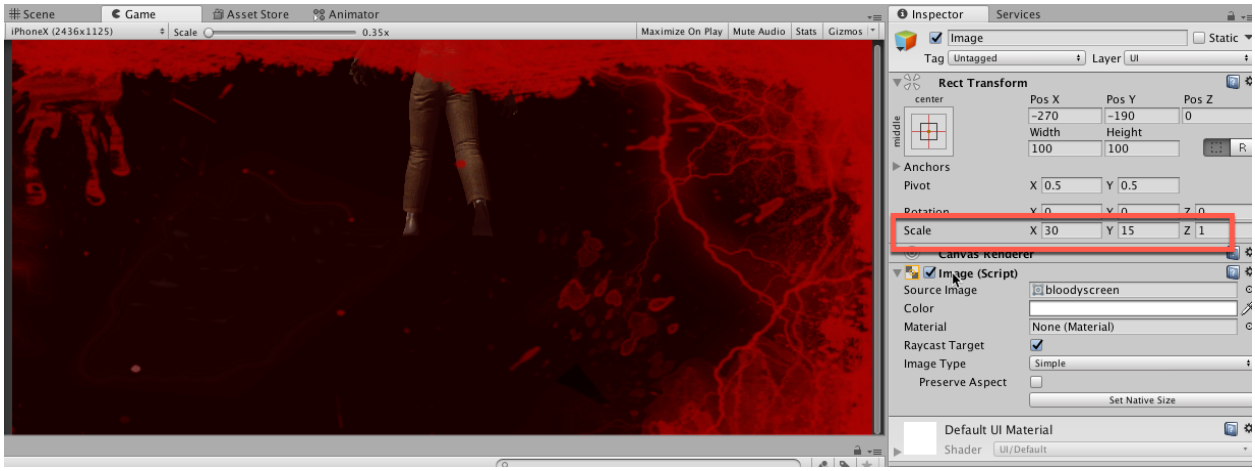
这里我们在Hierarchy视图中右键单击，选择UI-Image，
在Hierarchy视图中选择Image对象，在Inspector视图中点击Source Image，然后选择bloodyscreen。



接下来选择适配的设备分辨率，在Game视图的Free Aspect下拉列表中选择自己的设备分辨率，注意如果没有合适的选项，可以点击加号添加所需的分辨率。



继续在Game视图中，可以看到现在这个图片只覆盖了很小的范围。接下来让我们调整它的scale比例，直到覆盖整个界面。



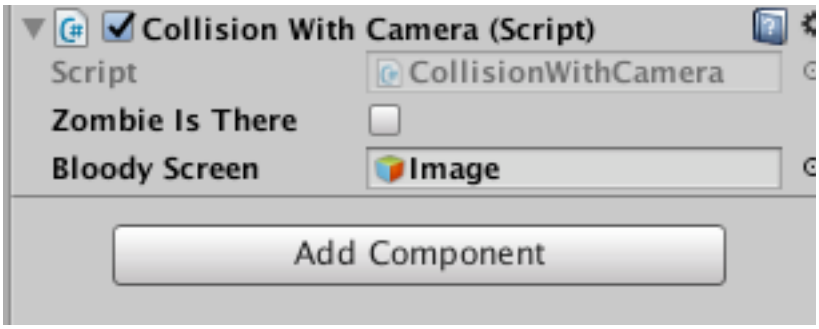
接下来我们要做的就是在合适的时机开启或禁用这个UI。

从Unity的Project视图中找到CollisionWithCamera.cs，双击将其打开。

首先创建一个到bloodyScreen对象的引用，

```
    //受到攻击的画面效果
    public GameObject bloodyScreen;
```

然后回到Unity，在Hierarchy视图中选择z@walk,然后在Collision With Camera组件中将Blood Screen后的属性设置为Image对象。



然后回到代码编辑器，在Attack()的方法体重添加一行代码以激活bloodyScreen对象。

```
    //攻击指令
    void Attack(){

        //恢复计时器为0
        timer = 0;
        //输出结果
//      Debug.Log ("attack");

        //播放攻击动画
        GetComponent<Animator>().Play("attack");

        //激活bloodyScreen对象
        bloodyScreen.gameObject.SetActive(true);


    }
```

但是仅仅这样还不够，我们还需要在两秒之后将bloodyScreen重新禁用。为了实现这一点，我们需要使用协程的方式来实现。

在Attack方法体中再添加一行代码：

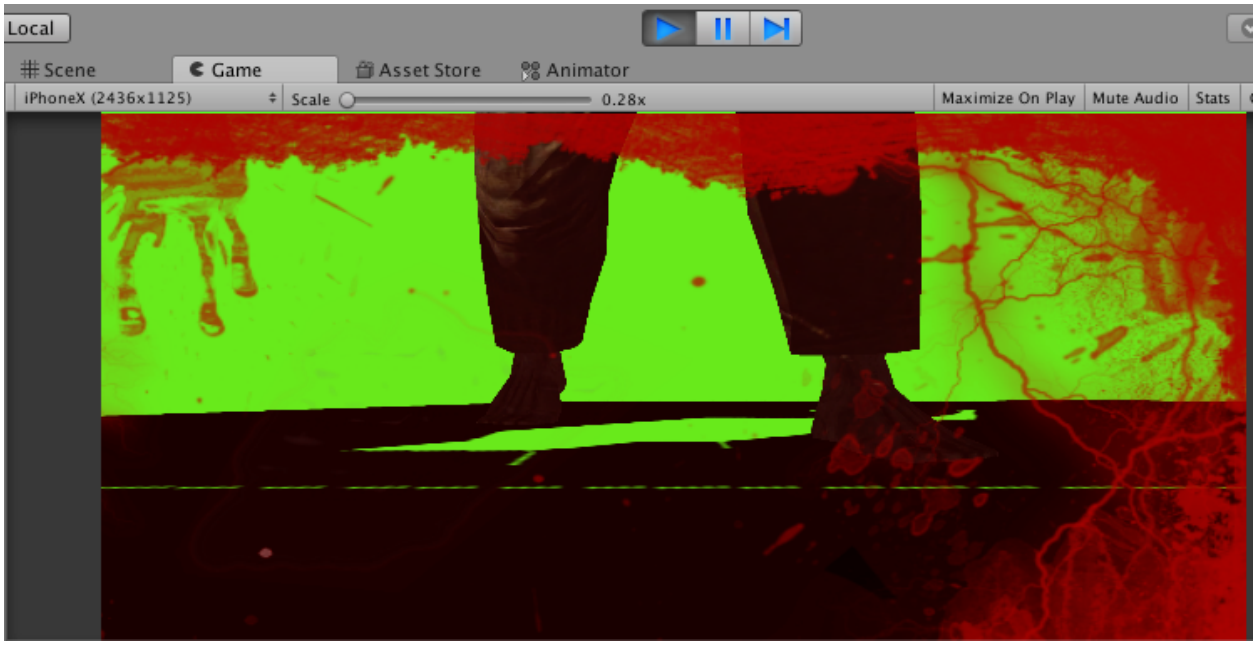
```
    //攻击指令
    void Attack(){

        //恢复计时器为0
        timer = 0;
        //输出结果
//      Debug.Log ("attack");

        //播放攻击动画
        GetComponent<Animator>().Play("attack");

        //激活bloodyScreen对象
        bloodyScreen.gameObject.SetActive(true);

        //添加协程，在两秒后禁用bloodyScreen
        StartCoroutine(WaitTwoSeconds
());
    }
```

然后添加WaitTwoSeconds方法的实现代码如下：

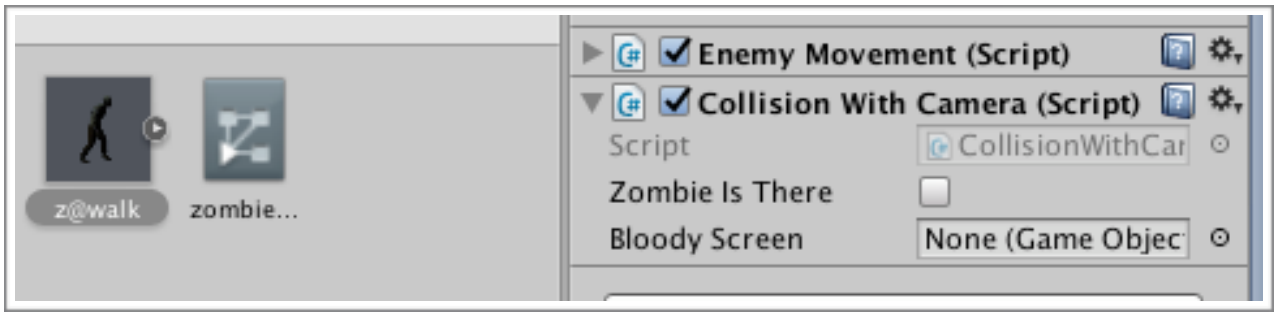
```
IEnumerator WaitTwoSeconds(){

      yield return new WaitForSeconds (2f);
      bloodyScreen.gameObject.SetActive (false);

}
```

接下来我们在Unity中先默认禁用Image游戏对象，只有当敌人攻击的时候才会显示出来。

点击Play按钮预览游戏效果，基本上已经实现了。

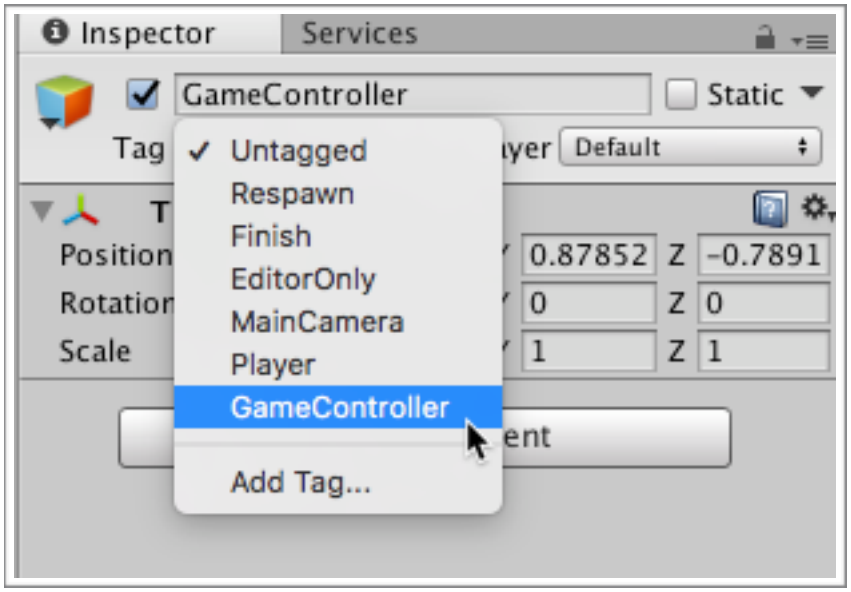


但是现在还有一个问题。
当我们把z@walk这个对象拖到Project视图的Assets中后，所生成的预设体中，Collision With Camera组件的Bloody Screen属性已经消失了。



而且此时如果我们想要把Image对象拖到Collision With Camera的Bloody Screen属性处，会发现根本无法完成该操作。
这是因为在Unity中，预设体只能以其它预设体作为引用对象。
这个问题是需要修复的，因为之后我们希望让敌人以预设体的方式出现在场景中，而非现在这样直接拖动到固定的地点。

我们将会换一种方式来实现。
在Hierarchy视图中创建一个新的空对象，将其命名为GameController，然后给它设置一个Tag，选择GameController这个类型。如果我们想设置其它名称的Tag，那么只需点击Add Tag即可。



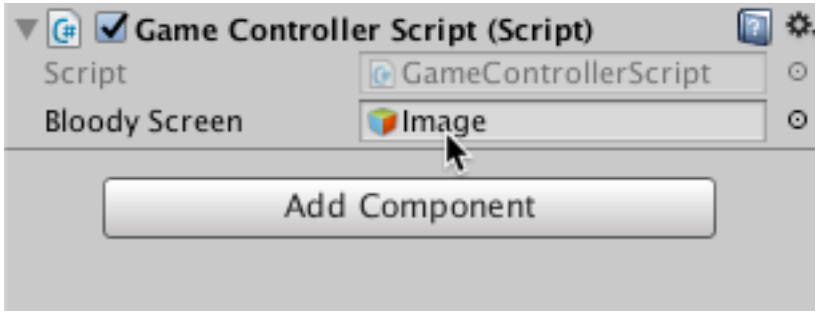
接下来给GameController对象添加一个新的脚本，将其命名为GameControllerScript。

接下来我们打开该文件，并更改代码如下：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameControllerScript : MonoBehaviour {

    //添加到bloodyScreen对象的引用
```

```
    public GameObject bloodyScreen;


    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

然后回到Unity编辑器，并在Game Controller Script脚本的Bloody Screen属性处设置到Image的引用。



接下来我们将产生画面效果的方法从CollisionWithCamera中移到GameControllerScript中，修改GameControllerScript的代码如下。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameControllerScript : MonoBehaviour {

    //添加到bloodyScreen对象的引用
    public GameObject bloodyScreen;


    // Use this for initialization
    void Start () {

    }
```

```
	// Update is called once per frame
	void Update () {

	}

	public void zombieAttack(bool zombieIsThere){

		//激活bloodyScreen对象
		bloodyScreen.gameObject.SetActive (true);

		//添加协程，在两秒后禁用bloodyScreen
		StartCoroutine(WaitTwoSeconds());
	}

	IEnumerator WaitTwoSeconds(){

		yield return new WaitForSeconds (2f);
		bloodyScreen.gameObject.SetActive (false);

	}
}
```

接下来我们需要创建从CollisionWithCamera到GameControllerScript之间的关联。
打开CollisionWithCamera.cs，并更改其中的代码如下：

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CollisionWithCamera : MonoBehaviour {

	//敌人是否在场
	public bool zombieIsThere;
	//计时器
	float timer;
	//两次攻击之间的间隔
	int timeBetweenAttack;

	//1.到GameControllerScript的引用
	private GameControllerScript gameController;
```

```
	// Use this for initialization
	void Start () {

		//定义初始数值
		timeBetweenAttack = 2;

		//2.创建到GameController对象的引用
		GameObject gameControllerObject =
GameObject.FindWithTag("GameController");

		if (gameControllerObject != null) {

			gameController =
gameControllerObject.GetComponent<GameControllerScript>();
		}
	}

	// Update is called once per frame
	void Update () {

		//获取系统时间
		timer += Time.deltaTime;
//		print (timer);

		//判断敌人是否在场，而且攻击间隔大于2秒
		if (zombieIsThere && timer >= timeBetweenAttack) {

			//开始攻击动作
			Attack ();
		}
	}
	//碰撞开始

	void OnCollisionEnter (Collision col)
	{
		//判断碰撞体中是否有主摄像机
		if (col.gameObject.tag == "MainCamera") {

//			Debug.Log ("enter");
			//确认敌人在现场
			zombieIsThere = true;
		}
	}
```

```
    //碰撞结束
    void OnCollisionExit(Collision col){

        //判断碰撞体中是否有主摄像机
        if (col.gameObject.tag == "MainCamera") {

//          Debug.Log ("exit");
            //设置敌人不在现场
            zombieIsThere = false;
        }

    }

    //攻击指令
    void Attack(){

        //恢复计时器为0
        timer = 0;
        //输出结果
//      Debug.Log ("attack");

        //播放攻击动画
        GetComponent<Animator>().Play("attack");

        //3.执行GameControllerScript中的方法
        gameController.zombieAttack(zombieIsThere);
    }


}
```

其中最关键的三处是数字编号1，2，3的部分。
其中编号1的代码处，我们创建了一个到GameControllerScript的引用。

在编号2的代码处，我们创建了到GameController对象的引用，并且当该对象不为空时，获取其GameControllerScript组件，并赋予在编号1处所创建的引用。

在编号3的代码处，我们调用了GameControllerScript中的方法。

回到Unity主编辑器，点击Play按钮预览游戏效果，发现和刚才的没有什么区别，只是实现机制已经得到了优化。