# Unity 2D 游戏开发快速入门

## ——《狂怒坦克 RAGETANK》制作

（内部资料）


大学霸

www.daxueba.net

# 前　言

Unity 是一款综合的游戏开发工具，也是一款全面整合的专业游戏引擎。使用它开发的游戏，可以部署到所有的主流游戏平台，而无需做任何修改。这样，开发者只需把精力集中到制作高质量的游戏即可。

本书通篇介绍了一个 2D 游戏——RageTanks（狂怒坦克）的详细开发过程，包从导入游戏资源、游戏逻辑设计到最后游戏逻辑的实现。本书将这一开发过程分成了 6 个部分来讲解，力求读者在每一部分都能实现一个可见的效果，而这些效果的综合体现就是最后的 RageTanks。

没用过 Unity？没关系，这里有详细的操作步骤；

没学过 C#?没关系，这里有详细的注释和解释，更何况 C#本来就不难；

得学习很久吗？不，即时你是新手，依然可以在一个月内做出本书介绍的这个游戏！

喔~说的夸张吗？一点儿也不！为什么这么自信，因为我是作者！我精心设计了这个游戏！它简单、结构清晰，而且也很有趣！我相信你通过对这个游戏的学习，可以增进对 3 个方面的理解：Unity、2D 游戏开发流程和脚本代码的编写。

**1.学习所需的系统和软件**

- ❑ 安装 Windows 7 操作系统
- ❑ 安装 Unity 4.5.3

**2.学习建议**

大家学习之前，可以致信到 xxxxxx，获取相关的资料和软件。如果大家在学习过程遇到问题，也可以将问题发送到该邮箱。我们尽可能给大家解决。

# 目 录

# 第 2 章　为游戏精灵添加更多状态

上一章，我们创建了一个简单的 2D 游戏。此游戏中的精灵有 3 个状态：idle、left 和 right。这看起来确实很酷！但是仅有的 3 个状态却限制了精灵的能力，以及游戏逻辑的想象空间。看来有必要让精灵拥有更多的状态，而这就是本章要讲解的主要内容。

## 2.1　摄像头追踪功能

游戏里的精灵可以在游戏场景中任意移动，这没什么问题，可是这就导致了一个问题，就是精灵可能移动到我们的视野之外，或者说游戏视图之外。为了解决这个问题，很多游戏都采用了“摄像头追踪”的方法，使得摄像头的位置会随着精灵的移动而移动。例如，《超级玛丽》中，精灵始终处于视图中心的位置，如图 2-1 所示。

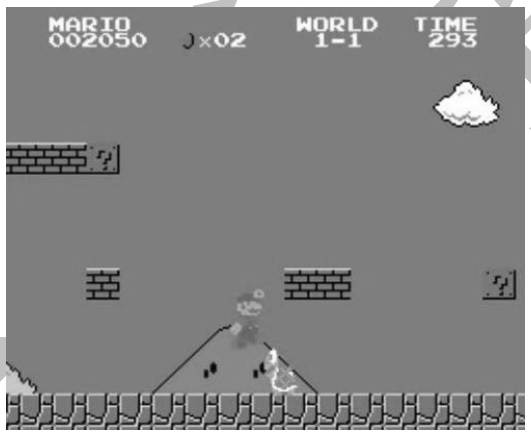



图 2-1　《超级玛丽》中，精灵始终位于游戏视图的中心

要为我们开发的游戏添加“摄像头追踪”的功能，就需要使用脚本编写这样一种逻辑。在 Project 视图的 Scripts 文件夹里，新建一个 C#脚本，命名为 CameraController，为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class CameraController : MonoBehaviour
05  {
06      //公有属性
07      //表示精灵当前的动画状态
08      public PlayerStateController.playerStates currentPlayerState =
PlayerStateController.playerStates.idle;
09      public GameObject playerObject = null;                    //表示精灵对象
10      public float cameraTrackingSpeed = 1f;                    //表示摄像机的追踪速度
11      //私有属性
12      private Vector3 lastTargetPosition = Vector3.zero;        //上一目标位置
13      private Vector3 currTargetPosition = Vector3.zero;        //下一目标位置
14      private float currLerpDistance = 0.0f;
```

```
15      //方法
16      void Start()
17      {
18          Vector3 playerPos = playerObject.transform.position;          //记录精灵的位置
19          Vector3 cameraPos = transform.position;                       //记录摄像机的位置
20          Vector3 startTargPos = playerPos;
21          lastTargetPosition = startTargPos;
22          currTargetPosition = startTargPos;
23      }
24      //加入订阅者列表
25      void OnEnable()
26      {
27          PlayerStateController.onStateChange += onPlayerStateChange;
28      }
29      //从订阅者列表中退出
30      void OnDisable()
31      {
32          PlayerStateController.onStateChange -= onPlayerStateChange;
33      }
34      //实时记录游戏精灵当前的动画状态
35      void onPlayerStateChange(PlayerStateController.playerStates newState)
36      {
37          currentPlayerState = newState;
38      }
39      void LateUpdate()
40      {
41          //依据当前精灵的动画状态，实时更新
42          onStateCycle();
43          //将摄像头移动到目标位置
44          currLerpDistance += cameraTrackingSpeed;
45          transform.position = Vector3.Lerp(lastTargetPosition, currTargetPosition,
currLerpDistance);
46      }
47      void onStateCycle()
48      {
49          switch(currentPlayerState)
50          {
51              case PlayerStateController.playerStates.idle:
52                  trackPlayer();
53              break;
54              case PlayerStateController.playerStates.left:
55                  trackPlayer();
56              break;
57              case PlayerStateController.playerStates.right:
58                  trackPlayer();
59              break;
60          }
61      }
62      void trackPlayer()
63      {
64          //获取并保存摄像机和精灵在世界坐标系的坐标
65          Vector3 currCamPos = transform.position;
66          Vector3 currPlayerPos = playerObject.transform.position;
```

```
67              lastTargetPosition = currCamPos;
68              currTargetPosition = currPlayerPos;
69              currTargetPosition.z = currCamPos.z;                    //保证摄像头 z 轴方向上的值不
变
70          }
71      }
```

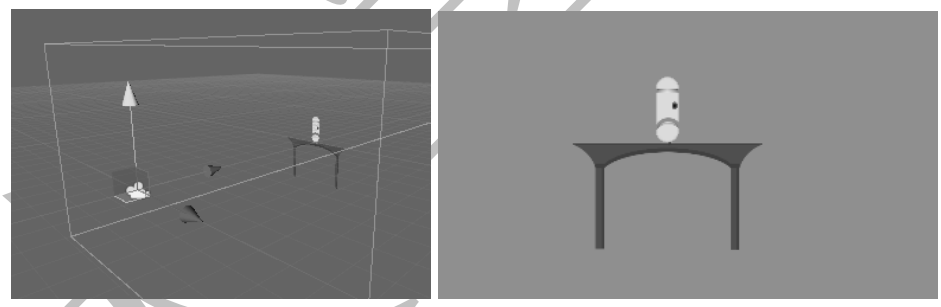
将此脚本赋予 Hierarchy 视图里的 Main Camera 对象，选中后者，然后在 Inspector 视图里设置此脚本组件的下列属性，如图 2-2 所示。

- Player Object：Player。表示摄像头要追踪的精灵对象；
- Camera Tracking Speed：1。此属性值范围是 0~1，若为 0 时，摄像头不会追踪精灵，若为 1 时，摄像头可以在瞬间追踪到精灵对象；


图 2-2　Main Camera 对象的脚本组件上各属性的设置

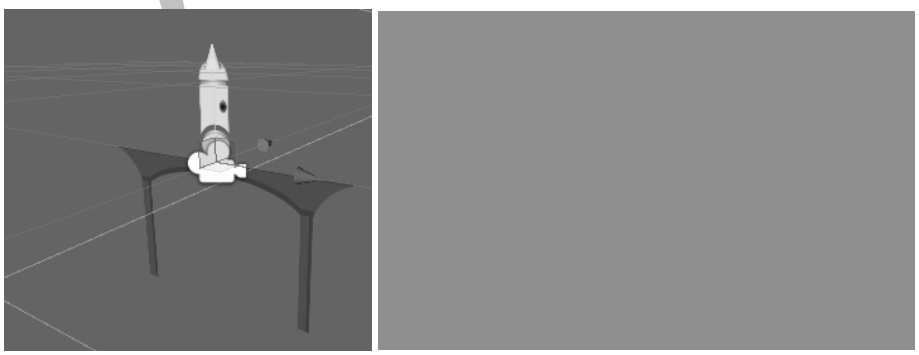
对于此脚本有以下几点需要说明：

- 脚本 69 行的代码使得摄像头与精灵对象不至于重合。要让摄像头实时追踪游戏精灵，只要实时更新摄像头的位置即可。就是要让它的位置与精灵的位置一致，但是它们在 Z 轴方向上的值不能相同，如图 2-3 所示。否则摄像头与精灵会发生重合，使得游戏视图里精灵对象消失，如图 2-4 所示。



Scene视图　　Game视图

图 2-3　摄像头与精灵在 Z 方向上的值不同



Scene视图　　Game视图

图 2-4　摄像头与精灵在 Z 方向上的值相同

- 脚本 45 行的 Vector3.Lerp()函数，是完成摄像头实时追踪功能的主要函数。这个函

数会将一个对象，以一定的速度从一个位置移动到另一个位置；

运行游戏，然后使用键盘上的方向键控制精灵左右移动，你会发现游戏视图会和精灵一同移动，甚至是精灵因为脱离地面发生坠落时，也不例外，如图 2-5 所示。

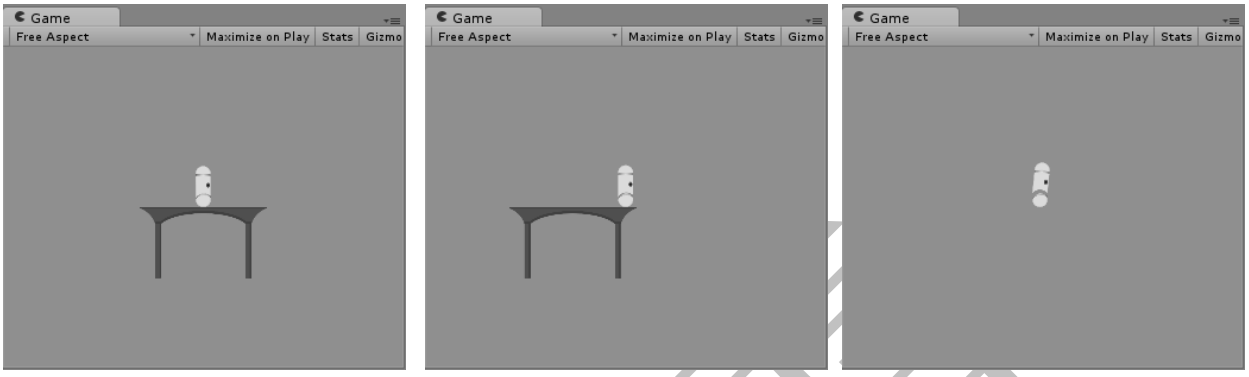



图 2-5　摄像头追踪功能，精灵始终位于游戏视图的中心

## 2.2　精灵的死亡和重生

目前为止，游戏项目里的精灵只有 Idle 和 Walking 这两种状态。也就是说，无论精灵在游戏里做什么，它都不会进入其它的状态，如死亡。于是我们发现游戏里的精灵，即使是跳入“万丈深渊”，也依然存活，显然这种游戏逻辑无法让人接受。因此，本节就来说明为精灵添加死亡和重生这两种状态的方法，并使用脚本实现这两种状态的逻辑。具体的实现步骤如下：

（1）在 Hierarchy 视图里，新建一个 Empty 对象，并命名为 Death Trigger，设置其 Position 属性为(0,0,0)。然后为此对象添加 Box Collider 2D 组件，并设置此组件的下列属性，如图 2-6 所示。

- 选中 Is Trigger 属性；
- Size：(20,1)；
- Center：(0,-2.5)；

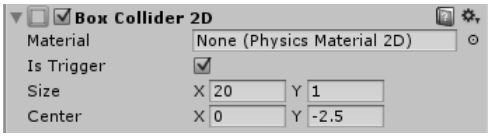



图 2-6　对象上 Box Collider 2D 组件的属性设置

回看此时的 Scene 视图，可知此步操作添加了一个绿色线框的矩形，如图 2-7 所示。我们希望当精灵与此矩形发生接触时，精灵会死亡。

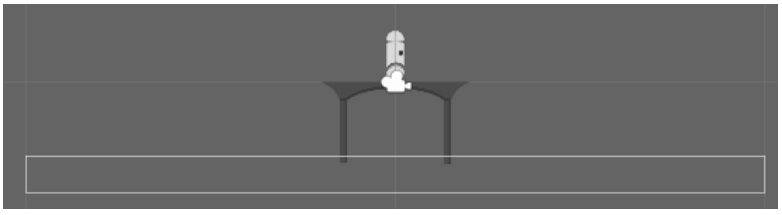


图 2-7　表示 Empty 对象范围的矩形框

（2）当精灵死亡以后，要想继续游戏，精灵必须在指定的位置重生才行，而且这个位置在精灵对象重生以后，不会让精灵被动的接触到 Death Trigger 对象。在 Hierarchy 视图里，再新建一个 Empty 对象，并命名为 Player Respawn Point，设置其 Position 属性为(0,1,5,0)，也就是说重生的点位于地面正上方的指定位置处，如图 2-8 所示。

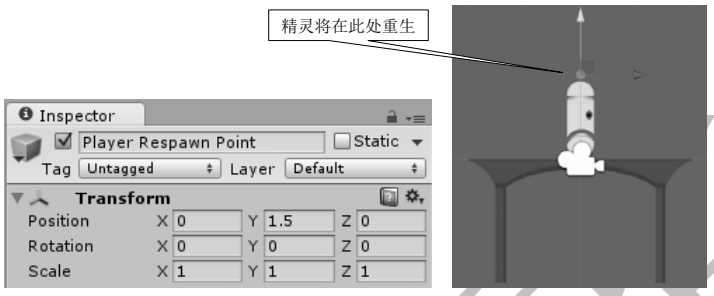



图 2-8　设置精灵重生点的位置

（3）打开 Project 视图里的 PlayerStateController 脚本，将死亡和重生这两种状态加到表示精灵状态的枚举类型中，如下代码中加粗的部分：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class PlayerStateController : MonoBehaviour
05  {
06      //定义游戏人物的各种状态
07      public enum playerStates
08      {
09          idle = 0,                    //表示空闲
10          left,                        //表示左移
11          right,                       //表示右移
12          kill,                        //表示死亡
13          resurrect                    //表示重生
14      }
15      …                                //省略
16  }
```

（4）在 Project 视图的 Script 文件夹里，新建一个 C#脚本，命名为 DeathTriggerScript，用于实现当精灵与 Death Trigger 接触时，精灵死亡的逻辑。为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class DeathTriggerScript : MonoBehaviour
05  {
01      //当精灵进入到 Death Trigger 的矩形范围内时，调用此函数
02      void OnTriggerEnter2D( Collider2D collidedObject )
03      {
04          //调用精灵对象上 PlayerStateListener 脚本组件里的 hitDeathTrigger()方法
05          collidedObject.SendMessage("hitDeathTrigger");
06      }
07  }
```

将此脚本赋予 Hierarchy 视图里的 Death Trigger 对象。脚本 05 行，调用的方法 hitDeathTrigger()还没有在 PlayerStateListener 脚本里定义，请将下面的方法定义添加到

PlayerStateListener 脚本里，定义如下：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Animator))]
public class PlayerStateListener : MonoBehaviour
{
    …                                   //省略
    public void hitDeathTrigger()
    {
        onStateChange(PlayerStateController.playerStates.kill);
    }
}
```

从方法的定义中可知，它所实现的功能是，修改精灵当前的状态为 Kill。

（5）继续为脚本 PlayerStateListener 添加代码，用于实现当精灵处于死亡和重生状态时，精灵应有的动作，或者说行为，部分脚本 PlayerStateListener 的代码如下：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Animator))]
public class PlayerStateListener : MonoBehaviour
{
    //公有属性
    public float playerWalkSpeed = 3f;                  //表示精灵移动的速度
    public GameObject playerRespawnPoint = null;        //表示重生的点
    //私有属性
    private Animator playerAnimator = null;             //表示对象上的 Animator 组件
    …                                                   //省略
    //用于检测当前所处的动画状态，在不同的状态下将表现出不同的行为
    void onStateCycle()
    {
        //表示当前对象的大小
        Vector3 localScale = transform.localScale;
        //判断当前处于何种状态
        switch(currentState)
        {
        …                                   //省略
        case PlayerStateController.playerStates.kill:
            onStateChange(PlayerStateController.playerStates.resurrect);
            break;

        case PlayerStateController.playerStates.resurrect:
            onStateChange(PlayerStateController.playerStates.idle);
            break;
        }
    }
    //当角色的状态发生改变的时候，调用此函数
    public void onStateChange(PlayerStateController.playerStates newState)
    {
        //如果状态没有发生变化，则无需改变状态
        if(newState == currentState)
            return;
```

```
37              //判断精灵能否由当前的动画状态，直接转换为另一个动画状态
38              if(!checkForValidStatePair(newState))
39                  return;
40              //通过修改 Parameter 中 Walking 的值，修改精灵当前的状态
41              switch(newState)
42              {
43              …                                                   //省略
44              case PlayerStateController.playerStates.kill:
45                  break;
46              //让精灵在场景重生对象的位置出现
47              case PlayerStateController.playerStates.resurrect:
48                  transform.position = playerRespawnPoint.transform.position;
49                  transform.rotation = Quaternion.identity;
50
51                  break;
52              }
53              //记录角色当前的状态
54              currentState = newState;
55          }
56
57          //用于确认当前的动画状态能否直接转换为另一动画状态的函数
58          bool checkForValidStatePair(PlayerStateController.playerStates newState)
59          {
60              bool returnVal = false;
61
62              //比较两种动画状态
63              switch(currentState)
64              {
65              …                                                       //省略
66              //精灵的 kill 状态只能转换为 resurrect 状态
67              case PlayerStateController.playerStates.kill:
68                  if(newState == PlayerStateController.playerStates.resurrect)
69                      returnVal = true;
70                  else
71                      returnVal = false;
72                  break;
73              //精灵的 resurrect 状态只能转换为 idle 状态
74              case PlayerStateController.playerStates. resurrect :
75                  if(newState == PlayerStateController.playerStates.idle)
76                      returnVal = true;
77                  else
78                      returnVal = false;
79                  break;
80              }
81              return returnVal;
82          }
83          public void hitDeathTrigger()
84          {
85              onStateChange(PlayerStateController.playerStates.kill);
86          }
87      }
```

对于此脚本，有以下几点需要说明：

- 脚本 09 行，添加了一个公有属性，用于表示游戏场景里 Player Respawn Point 对象的位置。这个属性的值需要在 Inspector 视图里设置，如图 2-9 所示。

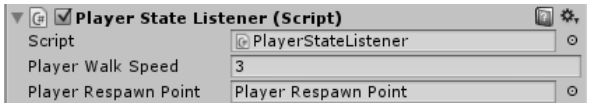



图 2-9　设置 Player State Listener 脚本组件里的 Player Respawn Point 属性值

- 脚本 14 行，方法 onStateCycle()里添加的代码，说明当精灵进入到 kill 状态以后，接着会进入 resurrect 状态；而进入 resurrect 状态的精灵会接着进入 idle 状态；
- 脚本 32 行，方法 onStateChange()里添加的代码，说明当精灵处于 resurrect 状态时，精灵将会出现在重生点的位置；
- 脚本 58 行，方法 checkForValidStatePair()里添加的代码，说明处于 kill 状态的精灵只能转换为 resurrect 状态；而处于 resurrect 状态的精灵只能转换为 idle 状态；

（6）为脚本 CameraController 添加处理精灵 kill 和 resurrect 状态的代码，如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class CameraController : MonoBehaviour
05  {
06      …                                          //省略
07      void onStateCycle()
08      {
09          switch(currentPlayerState)
10          {
11              …                                  //省略
12              case PlayerStateController.playerStates.kill:
13                  trackPlayer();
14              break;
15              case PlayerStateController.playerStates.resurrect:
16                  trackPlayer();
17              break;
18          }
19      }
20  }
```

（7）运行游戏，控制精灵移动至地面外，精灵在下落的过程中与 Death Trigger 发生接触，精灵死亡；很快的，精灵会在 Player Respawn Point 对象的位置处重生，如图 2-10 所示。

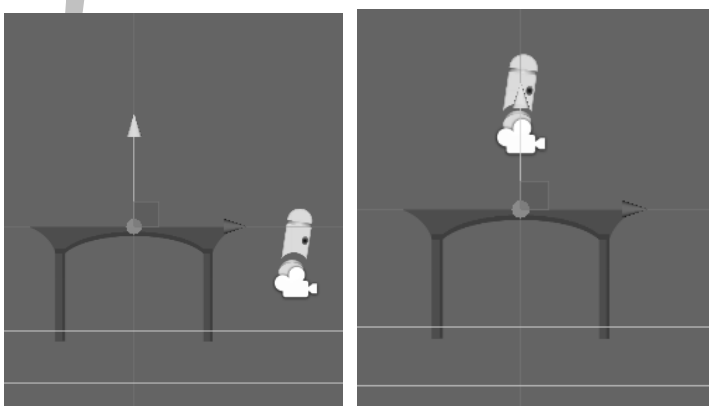


图 2-10　精灵的死亡和重生

# 2.3　添加多个地面

显然，只有一个地面的游戏场景太小了，根本不够精灵四处活动的。那么，本节就来介绍一种简单的方法，可以为游戏场景添加多个地面。具体的操作方法是：

（1）在 Project 视图里，新建一个文件夹，命名为 Prefabs。然后将 Hierarchy 视图里的 Platform 对象，拖动到 Prefabs 文件夹中，如此一来就可以生成一个同名的预置资源，如图 2-11 所示。

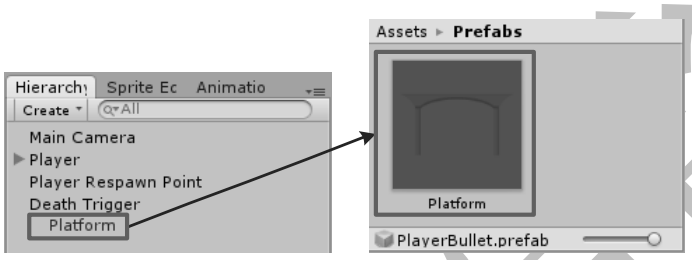



图 2-11　通过拖动对象到 Project 视图的方式，新建预置资源

（2）拖动 Project 视图 Prefabs 文件夹下的 Platform 资源到 Scene 或者 Hierarchy 视图，即可为游戏场景添加新的地面对象。在本示例中，使用此种方法为场景添加的多个地面对象，如图 2-12 所示。

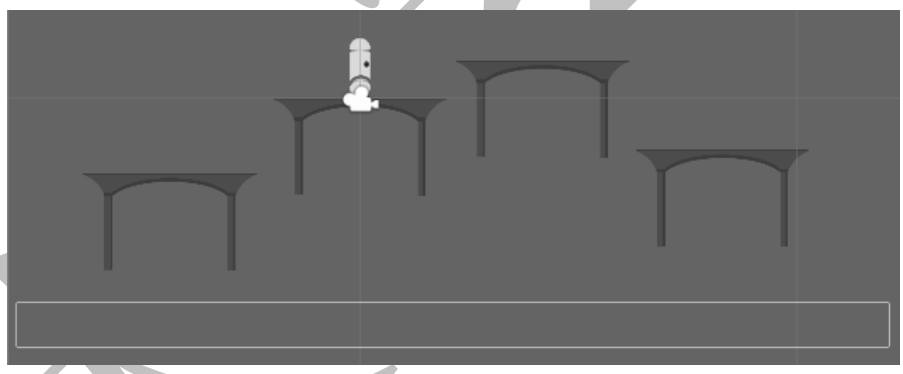


图 2-12　添加多个地面对象到游戏场景，同时修改 Death Trigger 对象的位置

（3）在本示例中，一共有 4 个地面对象，为了让 Hierarchy 视图看起来更加清晰简洁，可以再新建一个 Empty 对象，命名为 Platform Container，最后将场景中的所有地面对象都设置为此 Empty 对象的子对象，如图 2-13 所示。

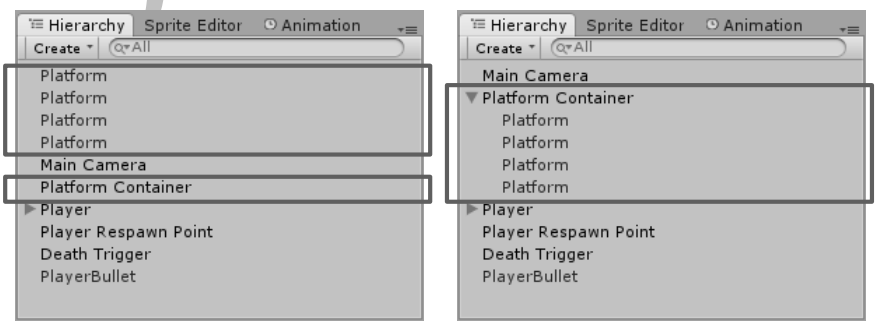



图 2-13　为了使 Hierarchy 视图更简洁，新建空对象，令其为其它对象的父对象

# 2.4 精灵的跳跃状态

为了让游戏中的精灵有更大的活动范围，上一节为游戏场景添加了多个地面，于是精灵可以从高的地面移动到低的地面处，如图 2-14 所示。但是却无法从低的地面移动到高的地面，因为当前的游戏精灵只能左右移动，即 left 和 right。为了解决这个问题，本节就来为精灵添加跳跃状态。

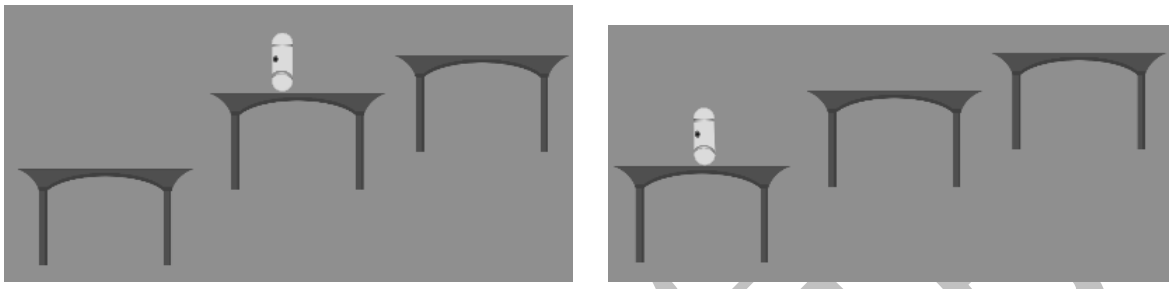


图 2-14 精灵从一个地面移动到另一个地面

（1）如果要为精灵添加跳跃状态，即 jump，就不得不再引入其它状态：

- landing：用于表示精灵接触到地面的这种状态。为了阻止精灵在跳跃到空中的时候再次跳跃，就需要在精灵做跳跃动作之前，确认其接触到了地面。

提示：有些游戏允许精灵在空中的时候再跳跃一次，也就是所谓的二级跳。例如，《天天酷跑》中，精灵就需要有二级跳的能力，因为有些场景光是跳跃一次无法跳过悬崖，或者吃到金币，如图 2-15 所示。


图 2-15 《天天酷跑》中，需要连续跳跃来越过断崖的精灵

- falling：用于表示精灵在空中的状态，处于此种状态下的精灵只可能过渡到 landing 和 kill 状态，也就是说精灵在空中的时候，无法产生其它动作行为，如 left 和 right。

（2）为脚本 PlayerStateController 中定义的动画状态枚举类型添加 3 种新的状态：jump、landing 和 falling，并且设置当玩家按下键盘上的空格键时，精灵会进入跳跃状态。脚本 PlayerStateController 中的部分代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class PlayerStateController : MonoBehaviour
```

```
{
    //定义游戏人物的各种状态
    public enum playerStates
    {
        …                                   //省略
        jump,                               //表示跳跃
        landing,                            //表示落地
        falling,                            //表示降落过程
    }
    …                                       //省略
    void LateUpdate ()
    {
        …                                   //省略
        //当玩家按下键盘上的空格键时，进入跳跃状态
        float jump = Input.GetAxis("Jump");
        if(jump > 0.0f)
        {
            if(onStateChange != null)
                onStateChange(PlayerStateController.playerStates.jump);
        }
    }
}
```

（3）在 Hierarchy 视图里，新建一个 Empty 对象，命名为 SceneryToggler，并拖动它到 Player 对象下，使其成为 Player 对象的子对象。为其添加 Box Collider 2D 组件，并在 Inspector 视图里，设置此组件的属性，如图 2-16 所示。

- 选中 Is Trigger 复选框；
- Size：x:1,y:2；
- Center：x:0,y:1；

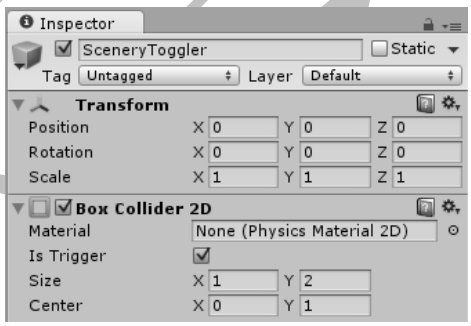



图 2-16　SceneryToggler 对象上，组件属性的设置

如此这般设置，相当于为 Player 对象添加了一个 Collider 检测框，如图 2-17 所示，可以看出除了精灵自身的 Polygon Collider 2D 的线框外，外层还包裹了矩形的线框。它在后面将用于检测精灵与场景中其它对象的碰撞。

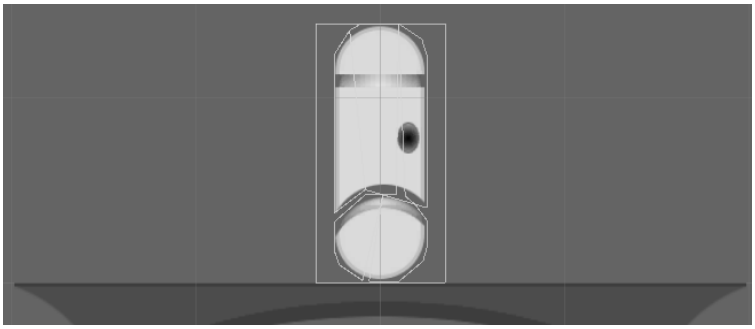


图 2-17　精灵对象外层包裹的矩形线框

读者此时心中一定在疑惑，既然精灵本身就有 Collider，为什么还要添加一个 Collider 呢？答案是，精灵是有实体的，因此即使其它对象与精灵发生了接触，它们也不会“交叉”，因此就永远不会触发我们期望的 landing 状态。但是对于 Empty 对象而言，就没有这种担心了，因为它根本没有实体，因此会与其它对象发生“交叉”，然后触发 landing 状态。

（4）在 Unity 里，单击 Edit|Project Settings|Tags and Layers 命令，打开 Tag&Layers 对话框，为 Tags 属性添加两个元素：Platform 和 DeathTrigger，如图 2-18 所示。

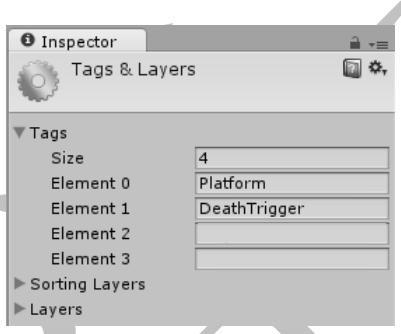



图 2-18　在 Tag&Layers 对话框里，为 Tags 属性添加两个元素

然后修改场景里地面对象的 Tag 属性为 Platform，Death Trigger 对象的 Tag 属性为 DeathTrigger。

（5）在 Project 视图里的 Scripts 文件夹下，新建一个 C#脚本，命名为 PlayerColliderListener，为此脚本添加下面的代码：

```
using UnityEngine;
using System.Collections;

public class PlayerColliderListener : MonoBehaviour
{
    public PlayerStateListener targetStateListener = null;          //表示精灵对象
    //进入碰撞检测区域时，调用此函数
    void OnTriggerEnter2D( Collider2D collidedObject )
    {
        switch(collidedObject.tag)
        {
            //当精灵落到地面上时，触发 landing 动画状态
            case "Platform":

targetStateListener.onStateChange(PlayerStateController.playerStates.landing);
            break;
        }
```

```
17        }
18        //离开碰撞检测区域时，调用此函数
19        void OnTriggerExit2D( Collider2D collidedObject)
20        {
21              switch(collidedObject.tag)
22              {
23                    //当精灵离开地面是，触发 falling 动画状态
24                    case "Platform":
25
    targetStateListener.onStateChange(PlayerStateController.playerStates.falling);
26                    break;
27                    //当精灵离开的是 Death Trigger 对象，则触发 kill 动画状态
28                    case "DeathTrigger":
29
    targetStateListener.onStateChange(PlayerStateController.playerStates.kill);
30                    break;
31              }
32        }
33  }
```

将此脚本赋予 Player 对象的子对象 SceneryToggler，选中后者，然后在 Inspector 视图里设置此脚本组件的 Target State Listener 属性为 Player 对象，如图 2-19 所示。

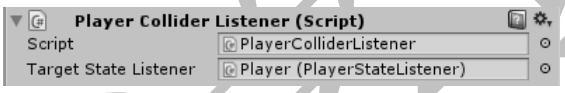



图 2-19　设置脚本组件里的属性

对于此脚本，有以下几点需要说明：

- 脚本 08 行定义的 OnTriggerEnter2D()，说明当与 SceneryToggler 对象发生碰撞的对象的 Tag 属性是 Platform 的时候（精灵与地面接触），精灵就会进入 landing 状态；
- 脚本 19 行定义的 OnTriggerExit2D()，说明当离开 SceneryToggler 对象检测范围的对象的 Tag 属性是 Platform 的时候（精灵离开地面），精灵就会进入 falling 状态；Tag 属性为 DeathTrigger 的时候（精灵离开场景中的 Death Trigger 对象），精灵就会进入 kill 状态；

提示：如果读者再回头细细的查看精灵的死亡逻辑的话，就会发现精灵的死亡只是表象！本质上是精灵在一瞬间转移到了重生点所在的位置。正是因为游戏中是如此实现精灵死亡逻辑的，所以才需要检测碰撞对象的 Tag 属性是否为 Death Trigger。

（6）为脚本 PlayerStateListener 添加对 jump、landing 和 falling 这些状态的处理，如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  [RequireComponent(typeof(Animator))]
05  public class PlayerStateListener : MonoBehaviour
06  {
07        public float playerWalkSpeed = 3f;                  //表示精灵移动的速度
08        public GameObject playerRespawnPoint = null;        //表示重生的点
09        public float playerJumpForceVertical = 300f;        //表示跳跃时，水平方向上，力的大
小
10        public float playerJumpForceHorizontal = 200f;      //表示跳跃时，垂直方向上，力的大
小
11        private bool playerHasLanded = true;                //表示精灵是否落地
```

```
12          …                                                    //省略
13          //当角色的状态发生改变的时候，调用此函数
14          public void onStateChange(PlayerStateController.playerStates newState)
15          {
16              //如果状态没有发生变化，则无需改变状态
17              if(newState == currentState)
18                  return;
19              //判断精灵能否由当前的动画状态，直接转换为另一个动画状态
20              if(!checkForValidStatePair(newState))
21                  return;
22              //通过修改 Parameter 中 Walking 的值，修改精灵当前的状态
23              switch(newState)
24              {
25              …                                        //省略
26              case PlayerStateController.playerStates.jump:
27                  if(playerHasLanded)
28                  {
29                      //确定精灵的跳跃方向
30                      float jumpDirection = 0.0f;
31                      if(currentState == PlayerStateController.playerStates.left)
32                          jumpDirection = -1.0f;
33                      else if(currentState == PlayerStateController.playerStates.right)
34                          jumpDirection = 1.0f;
35                      else
36                          jumpDirection = 0.0f;
37                      //给精灵施加一个特定方向的力
38                      rigidbody2D.AddForce(new          Vector2(jumpDirection          *
playerJumpForceHorizontal,
39                                                  playerJumpForceVertical));
40                      playerHasLanded = false;
41                  }
42                  break;
43              case PlayerStateController.playerStates.landing:
44                  playerHasLanded = true;
45                  break;
46              }
47              //记录角色当前的状态
48              currentState = newState;
49          }
50          //用于确认当前的动画状态能否直接转换为另一动画状态的函数
51          bool checkForValidStatePair(PlayerStateController.playerStates newState)
52          {
53              bool returnVal = false;
54              //比较两种动画状态
55              switch(currentState)
56              {
57              …                                    //省略
58              case PlayerStateController.playerStates.jump:
59                  if(     newState == PlayerStateController.playerStates.landing
60                      || newState == PlayerStateController.playerStates.kill
61                      )
62                      returnVal = true;
63                  else
```

```
64                          returnVal = false;
65                  break;
66              case PlayerStateController.playerStates.landing:
67                  if(     newState == PlayerStateController.playerStates.left
68                          || newState == PlayerStateController.playerStates.right
69                          || newState == PlayerStateController.playerStates.idle
70                          )
71                          returnVal = true;
72                  else
73                          returnVal = false;
74                  break;
75              case PlayerStateController.playerStates.falling:
76                  if(     newState == PlayerStateController.playerStates.landing
77                          || newState == PlayerStateController.playerStates.kill
78                          )
79                          returnVal = true;
80                  else
81                          returnVal = false;
82                  break;
83              }
84              return returnVal;
85          }
86          …                                   //省略
87  }
```

修改后的脚本，在脚本组件里就会多出两个属性，分别用于决定精灵跳跃时水平和垂直方向受到的力，如图 2-20 所示。

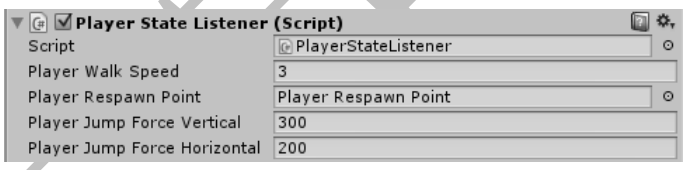



图 2-20　修改后的脚本组件

对于此脚本里新加入的代码，有以下几点需要说明：

- 脚本 09~11 行，定义了 3 个变量：playerJumpForceVertical、playerJumpForceHorizontal 和 playerHasLanded，用于辅助新加入状态的逻辑实现；
- 脚本 14 行，onStateChange()里添加的代码说明，精灵只有在落地的时候才能继续跳跃，而且脚本会依据精灵当前的运动朝向，来决定是朝左跳还是朝右跳，具体的方法是，给精灵的刚体施加一个力，精灵将在此力的作用下产生类似于跳跃的行为。
- 脚本 51 行，checkForValidStatePair()里添加的代码说明，jump 状态只能转换为 landing 和 kill 状态；landing 状态只能转换为 left、right 和 idle 状态；falling 状态只能转换为 landing 和 kill 状态；

（7）运行游戏，按下空格的时候，精灵就会进入 jump 状态了。当精灵处于 idle 状态时，精灵会原地起跳；处于 left 状态时，会朝左跳，同理 right 状态；最终，精灵可以在多个地面间上窜下跳，如图 2-21 所示。

图 2-21 精灵的跳跃状态（原地起跳、左跳和右跳）

提示：当精灵与 Death Trigger 对象接触，并进入死亡状态时，Unity 里的 Console 视图里会弹出一条警告信息：hitDeathTrigger has no receiver!，如图 2-22 所示。


图 2-22 Console 视图里弹出的警告信息

这是由于脚本 PlayerColliderListener 里包含了方法 OnTriggerExit2D()，代码如下：

```
01        //离开碰撞检测区域时，调用此函数
02        void OnTriggerExit2D( Collider2D collidedObject)
03        {
04              switch(collidedObject.tag)
05              {
06                    //当精灵离开地面是，触发 falling 动画状态
07                    case "Platform":
08
    targetStateListener.onStateChange(PlayerStateController.playerStates.falling);
09                    break;
10                    //当精灵离开的是 Death Trigger 对象，则触发 kill 动画状态
11                    case "DeathTrigger":
12
    targetStateListener.onStateChange(PlayerStateController.playerStates.kill);
13                    break;
14              }
15        }
```

代码中加粗的部分直接激活了精灵的 kill 状态，因此不再需要 hitDeathTrigger()方法去激活这个状态。因此，读者可以做两步操作，来解决 Console 视图里提示的警告信息。首先，移除 Death Trigger 对象上的 Death Trigger Script 组件，然后，移除脚本 PlayerStateListener 里定义的 hitDeathTrigger()方法：

```
01        public void hitDeathTrigger()
02        {
03              onStateChange(PlayerStateController.playerStates.kill);
04        }
```

# 2.5　精灵的开火状态

“开火”就是发射子弹的意思，在战争类型的电影或者电视剧中，主角们就爱这么说！本节打算为精灵添加发射子弹的能力。因为本游戏在后面会引入敌人，而精灵最好具备开火的能力，否则会被敌人轻易干掉！具体的实现方法是：

（1）导入一个表示子弹的图片到 Unity，本示例中选用的子弹图片，名为 PlayerBullet，如图 2-23 所示。

图 2-23　导入到游戏项目的表示子弹的图片

（2）拖动此图片到 Scene 视图，即可在当前的游戏场景中添加此对象，为此对象添加 RigidBody 2D 组件，并设置组件的下列属性，如图 2-24 所示。

- Gravity Scale：0。表示子弹对象将不受重力的影响，所以不会偏离预定的发射轨道；

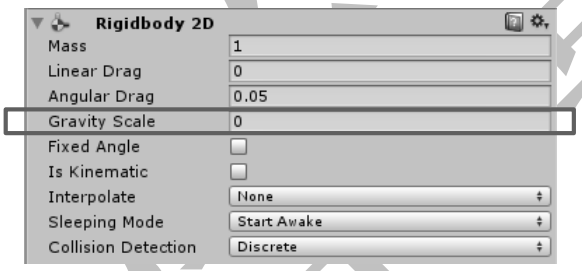



图 2-24　设置子弹对象上各组件的属性

（3）在 Project 视图里，新建一个 C#脚本，命名为 PlayerBulletController，为此脚本添加下面的代码：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class PlayerBulletController : MonoBehaviour
05  {
06      public GameObject playerObject = null;          //表示精灵对象
07      public float bulletSpeed = 15.0f;               //表示子弹的速度
08      private float selfDestructTimer = 0.0f;         //表示子弹自我销毁的时间
09      void Update()
10      {
11          //在一段时间以后销毁子弹对象
12          if(selfDestructTimer > 0.0f)
13          {
14              if(selfDestructTimer < Time.time)
15                  Destroy(gameObject);
16          }
17      }
18      //调用此函数发射子弹
19      public void launchBullet()
20      {
```

```
21              //确定精灵对象的朝向
22              float mainXScale = playerObject.transform.localScale.x;
23              Vector2 bulletForce;                                    //对子弹施加的力的方向
24              //如果精灵对象面朝左，则向左发射子弹
25              if(mainXScale < 0.0f)
26              {
27                  bulletForce = new Vector2(bulletSpeed * -1.0f,0.0f);
28              }
29              //如果精灵对象面朝右，则向右发射子弹
30              else
31              {
32                  bulletForce = new Vector2(bulletSpeed,0.0f);
33              }
34              //施加给子弹对象一个指定方向的力
35              rigidbody2D.velocity = bulletForce;
36              //在 1 秒后销毁子弹对象
37              selfDestructTimer = Time.time + 1.0f;
38          }
39      }
```

将此脚本赋予场景中的子弹对象，选中后者，然后在 Inspector 视图里可以设置此脚本组件的 Bullet Speed 属性，用于修改子弹的发射速度，如图 2-25 所示。


图 2-25　脚本组件的属性设置

将 Hierarchy 视图里的子弹对象，拖动到 Project 视图，即可创建一个子弹对象的预置资源，以备后面脚本中的代码使用。

（4）在 PlayerStateController 脚本中，添加精灵开火的状态，且指定当玩家按下鼠标左键的时候，精灵将进入开火状态，要添加的代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class PlayerStateController : MonoBehaviour
05  {
06      //定义游戏人物的各种状态
07      public enum playerStates
08      {
09          …                                   //省略
10          falling,                            //表示降落过程
11          firingWeapon                        //表示开火
12      }
13      //定义委托和事件
14      public delegate void playerStateHandler(PlayerStateController.playerStates newState);
15      public static event playerStateHandler onStateChange;
16      void LateUpdate ()
17      {
18          …                                   //省略
19          //当玩家按下鼠标的左键时，进入开火状态
20          float firing = Input.GetAxis("Fire1");
21          if(firing > 0.0f)
```

```
22              {
23                  if(onStateChange != null)
24                      onStateChange(PlayerStateController.playerStates.firingWeapon);
25              }
26          }
27      }
```

（5）在 Hierarchy 视图里，新建一个 Empty 对象，命名为 BulletSpawnPoint。拖动它到 Player 对象下，使其成为 Player 对象的子对象。然后调整 BulletSpawnPoint 对象的位置，使其位于精灵发射子弹时，子弹的出现位置。在本示例中，它们的相对位置如图 2-26 所示。

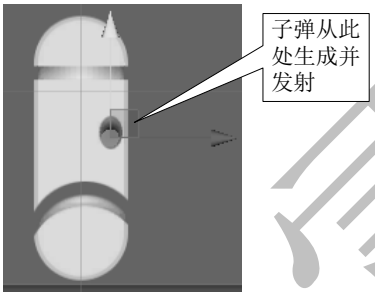



图 2-26　设置开火时，子弹的发射位置

（6）在 PlayerStateListener 脚本中，添加代码用于处理精灵进入开火状态时，开火的游戏逻辑，要添加的代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  [RequireComponent(typeof(Animator))]
05  public class PlayerStateListener : MonoBehaviour
06  {
07      …                                                   //省略
08      public GameObject bulletPrefab = null;              //表示子弹对象
09      public Transform bulletSpawnTransform;              //表示创建子弹的位置
10      …                                                   //省略
11      //当角色的状态发生改变的时候，调用此函数
12      public void onStateChange(PlayerStateController.playerStates newState)
13      {
14          //如果状态没有发生变化，则无需改变状态
15          if(newState == currentState)
16              return;
17          //判断精灵能否由当前的动画状态，直接转换为另一个动画状态
18          if(!checkForValidStatePair(newState))
19              return;
20          //通过修改 Parameter 中 Walking 的值，修改精灵当前的状态
21          switch(newState)
22          {
23          …                                               //省略
24          case PlayerStateController.playerStates.firingWeapon:
25              //实例化一个子弹对象
26              GameObject newBullet = (GameObject)Instantiate(bulletPrefab);
27              //设置子弹的起始位置
28              newBullet.transform.position = bulletSpawnTransform.position;
29              //建立与子弹对象上 PlayerBulletController 组件的联系
```

```
30                  PlayerBulletController bullCon =
newBullet.GetComponent<PlayerBulletController>();
31                  //设定子弹对象上的 PlayerBulletController 组件的 player object 属性
32                  bullCon.playerObject = gameObject;
33                  //发射子弹
34                  bullCon.launchBullet();
35                  break;
36              }
37              //记录角色当前的状态
38              currentState = newState;
39          }
40          //用于确认当前的动画状态能否直接转换为另一动画状态的函数
41          bool checkForValidStatePair(PlayerStateController.playerStates newState)
42          {
43              bool returnVal = false;
44              //比较两种动画状态
45              switch(currentState)
46              {
47              …                                           //省略
48              case PlayerStateController.playerStates.jump:
49                  if(     newState == PlayerStateController.playerStates.landing
50                      || newState == PlayerStateController.playerStates.kill
51                      || newState == PlayerStateController.playerStates.firingWeapon
52                      )
53                      returnVal = true;
54                  else
55                      returnVal = false;
56                  break;
57              case PlayerStateController.playerStates.landing:
58                  if(     newState == PlayerStateController.playerStates.left
59                      || newState == PlayerStateController.playerStates.right
60                      || newState == PlayerStateController.playerStates.idle
61                      || newState == PlayerStateController.playerStates.firingWeapon
62                      )
63                      returnVal = true;
64                  else
65                      returnVal = false;
66                  break;
67              case PlayerStateController.playerStates.falling:
68                  if(
69                      newState == PlayerStateController.playerStates.landing
70                      || newState == PlayerStateController.playerStates.kill
71                      || newState == PlayerStateController.playerStates.firingWeapon
72                      )
73                      returnVal = true;
74                  else
75                      returnVal = false;
76                  break;
77              case PlayerStateController.playerStates.firingWeapon:
78                  returnVal = true;
79                  break;
80              }
81              return returnVal;
```

```
82        }
83    }
```

对于此脚本，有以下几点需要说明：

- 脚本 08、09 行的代码，使得脚本组件中多出了两个属性设置框，如图 2-27 所示。需要依次设置为子弹预置对象和子弹的生成位置。

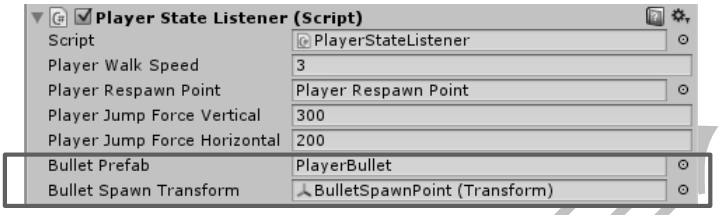
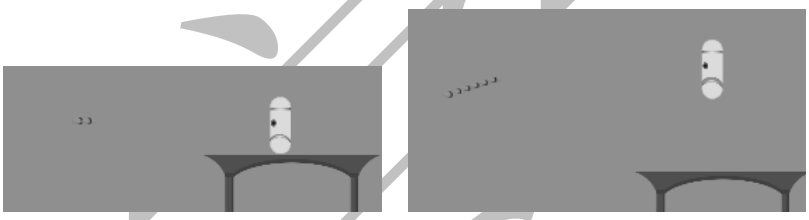



图 2-27　脚本组件上增加的两个属性，需要手动设置属性值

- 脚本 12 行，定义的方法 onStateChange()里面添加的代码，表示精灵处于开火状态时，将在指定位置上实例化一个子弹的预置对象，然后通过给子弹施加一个力来将它发射出去。
- 脚本 41 行，定义的方法 checkForValidStatePair()里面添加的代码，表示处于 jump、landing 和 falling 状态的精灵，可以转换为 firingWeapon，即开火状态。

（7）运行游戏，当按下鼠标的左键时，精灵将发射子弹，而且精灵可以在任何状态下发射子弹，如图 2-28 所示。



idle状态下开火　　jump状态下开火

图 2-28　精灵开火状态的运行效果