第 3 章　着手开发一个简单的 2D 游戏

本章将使用 Unity 提供的标准工具和编辑器，开发一个简单的 2D 游戏。这个游戏的主题是：使用宇宙飞船抵御外星的侵略者，如图 3-1 所示。在这个开发过程中，告诉你一些抽象的原理，以及如何快速开发一款 2D 游戏，当然这是在没有其他辅助的情况下开发的游戏。

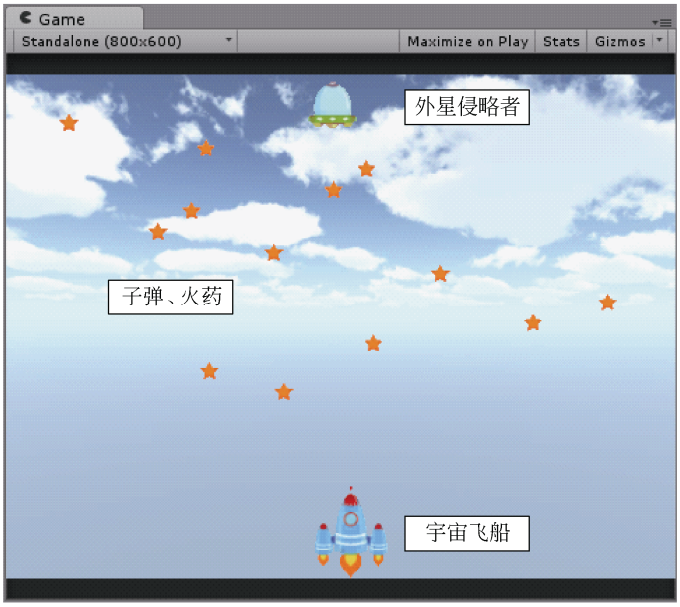


图 3-1　本章开发的游戏——展示图

3.1　开始开发 2D 游戏

要开发游戏了，第一步当然是新建一个项目，然后导入开发游戏要使用的资源。具体的操作还包括合理设置导入的资源、合理引用资源和配置游戏时的环境。

3.1.1　导入纹理资源

事先在资源视图面板下创建 3 个文件夹 materials、scripts 和 textures，分别用于存放对应的资源，如图 3-2 所示。这样操作是一个很好的习惯，方便以后对众多资源的管理。打开文件夹 textures 导入游戏要使用的 3 个纹理资源 ammo、Enemy 和 spaceship，如图 3-3

所示。

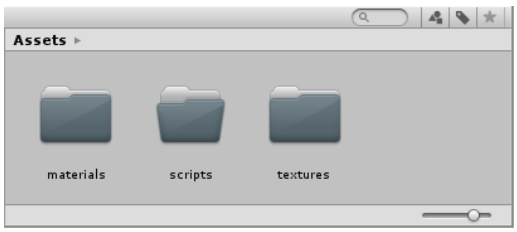


图 3-2　在资源面板创建 3 个文件夹

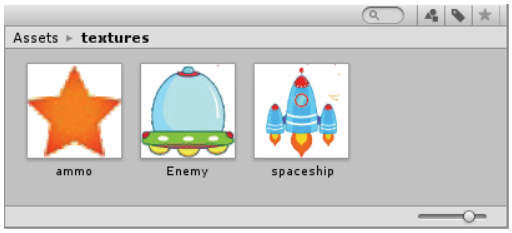


图 3-3　导入 Unity 中的 3 个纹理资源

修改 3 个纹理资源导入时的默认设置：设置它们的最大尺寸为 4096，如图 3-4 所示。方便以后游戏场景对纹理资源的缩放。

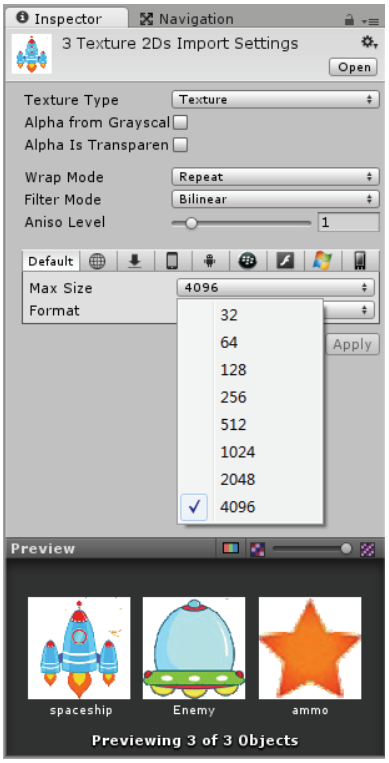


图 3-4　修改导入纹理的最大尺寸

提示：本章使用的 3 个纹理资源本身就是正方形的，包含透明的信息，且依据在第 2 章学习到的知识，它们本身的尺寸应该是 2 的次方（本示例使用的 spaceship.psd、enemy.psd 尺寸是 256 × 256，ammo.psd 尺寸是 128 × 128），导入到 Unity 以后，也设置了它们的大小为 4096 × 4096。

3.1.2　新建材质资源

纹理已经导入了，现在开始新建 3 个材质资源，用于引用导入的 3 个纹理资源。在

materials 文件夹下新建 3 个材质资源并命名为 mat_Ammo、mat_Enemy 和 mat_Player。同时选中 3 个材质，修改它们的着色器类型为：Unlit 中的 Transparent Cutout，如图 3-5 所示。

提示：Transparent Cutout 着色器类型很适合着色这样一类纹理资源：资源含有透明信息，且透明信息只分为完全透明和完全不透明这两种！

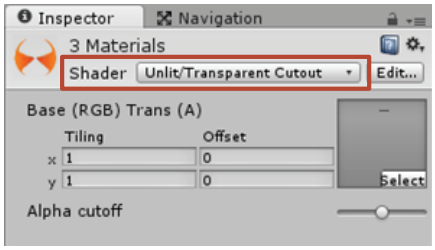


图 3-5　3 个材质要使用的着色器类型

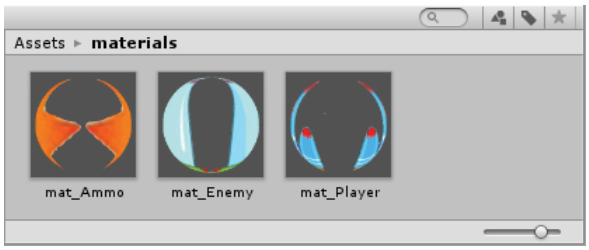


图 3-6　引用了纹理的 3 个材质资源

3 个材质分别引用对应的纹理资源后，资源面板中的材质资源如图 3-6 所示。

3.1.3　修改场景的环境光及游戏时的屏幕尺寸

在第 2 章曾介绍过 2 个让 3D 场景显示 2D 效果的技巧，本小节使用第一个技巧：修改场景的环境光（由黑色修改为白色），如图 3-7 所示。选择 Edit|Render Settings 命令，在弹出的 RenderSettings 查看器下修改 Ambient Light 属性为白色（R:255 G:255 B:255）。

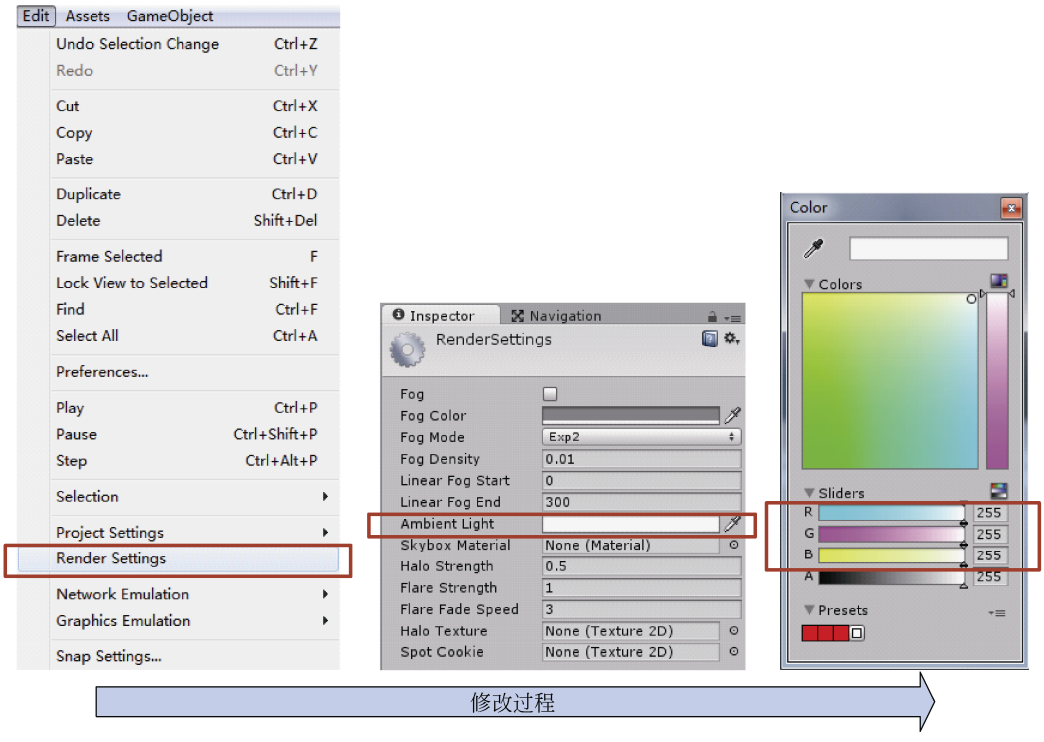


图 3-7　修改场景中的环境光

为了让游戏运行时，游戏的窗口有默认的大小，还需要设置游戏分辨率，如图 3-8 所示。本章修改游戏的分辨率为 800×600。选择 Edit|Project Settings|Player 命令，在弹出的 PlayerSettings 查看器下修改 Resolution 属性，即游戏时默认的游戏分辨率：宽 800、高 600。

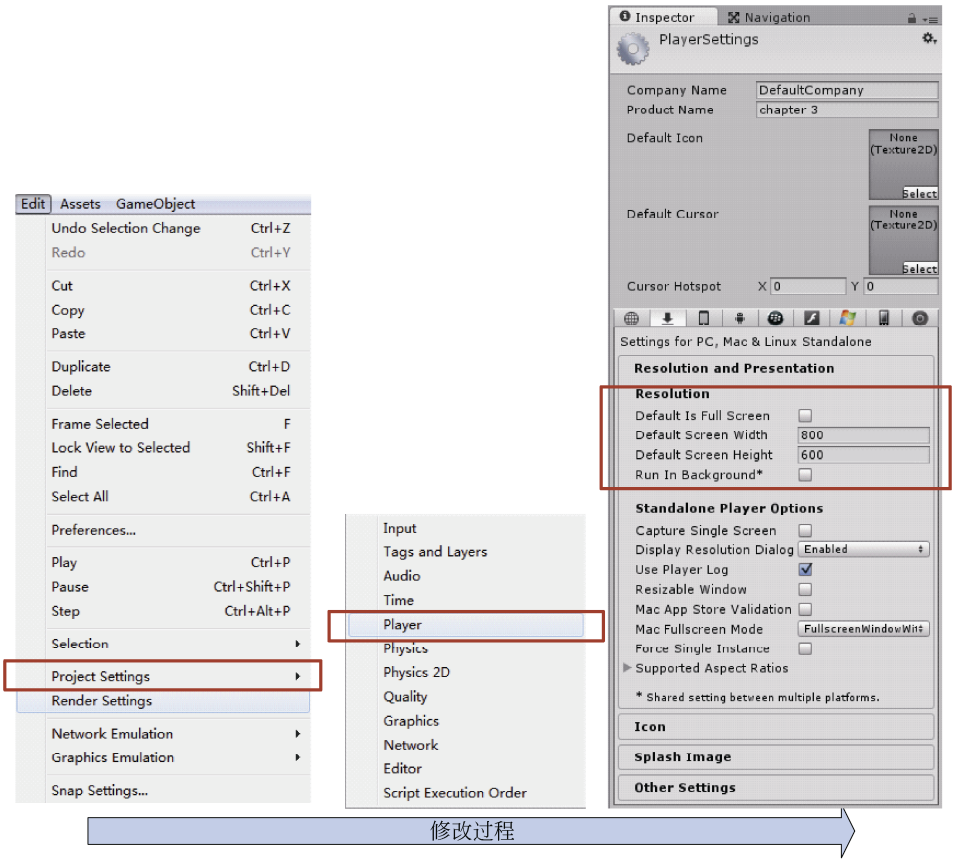


图 3-8　修改游戏的分辨率（菜单->修改值）

在运行游戏时，实时地在 Game 标签下看到特定游戏分辨率下的效果，可以在 Game 标签下做很简单的设置，如图 3-9 所示。这个标签下有一些预定义的分辨率，因为这些分辨率比较常用。

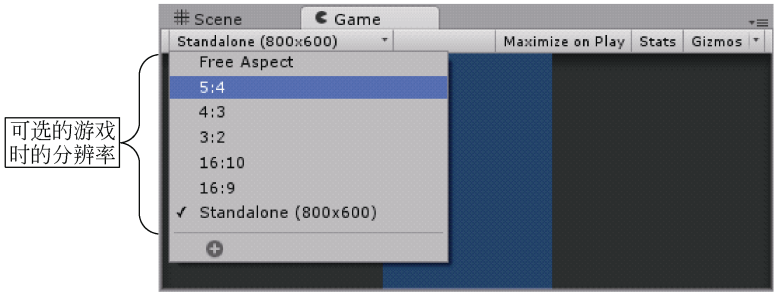


图 3-9　实时查看游戏在不同分辨率下的效果

在 Game 标签下除了选择一些预先设置好的分辨率之外，还可以设置其他的分辨率，

如图 3-10 所示。

图 3-10　使用自定义的分辨率查看游戏效果

3.2　为场景添加游戏对象

到目前为止，项目的场景里除了主摄像机对象以外什么都没有。从现在开始，本节要着手在场景中添加 2 个对象，分别用来做宇宙飞船和外星人的飞船。

3.2.1　调整游戏对象的角度

在场景中可以添加很多种对象，而本游戏要添加的对象是 Plane。生成时这个对象是水平放置的，现在要让它垂直放置，即与主摄像机的视角垂直。为此，还不得不旋转 Plane 对象，这个对象将引用名为 mat_Player 的材质。为了区分，现在把这个 Plane 对象命名为 Plane_Player。在旋转及引用了材质以后，Plane_Player 对象如图 3-11 所示。

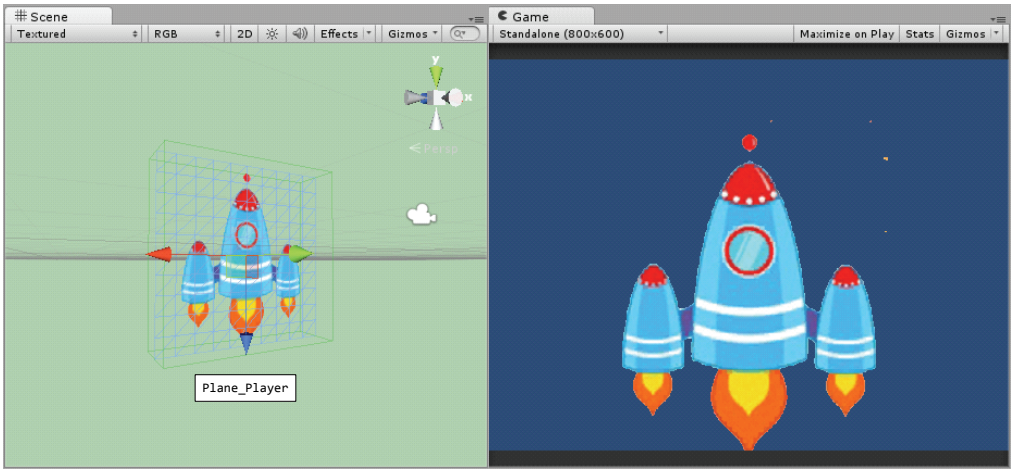


图 3-11　场景和主摄像机视角下的 Plane_Player 对象图

提示：旋转对象时，为了保证可以旋转固定的角度，可以在旋转的同时按下键盘上的 Ctrl 键。此时，每次的旋转角度将是固定的 15°。

3.2.2　改变游戏对象的位置

很显然，游戏对象相对于主摄像机的视角来说太大了，几乎占满了整个游戏视图。为此，不得不改变游戏对象的位置。对于 Plane_Player 这个游戏对象，它应该被放在远离主摄像机的位置（近大远小），而且应该处在游戏视图的底部。同理，再创建一个 Plane 对象，命名为 Plane_Enemy，引用名为 mat_Enemy 的材质，选中并调整为合适的角度及与主摄像机的距离，如图 3-12 所示。

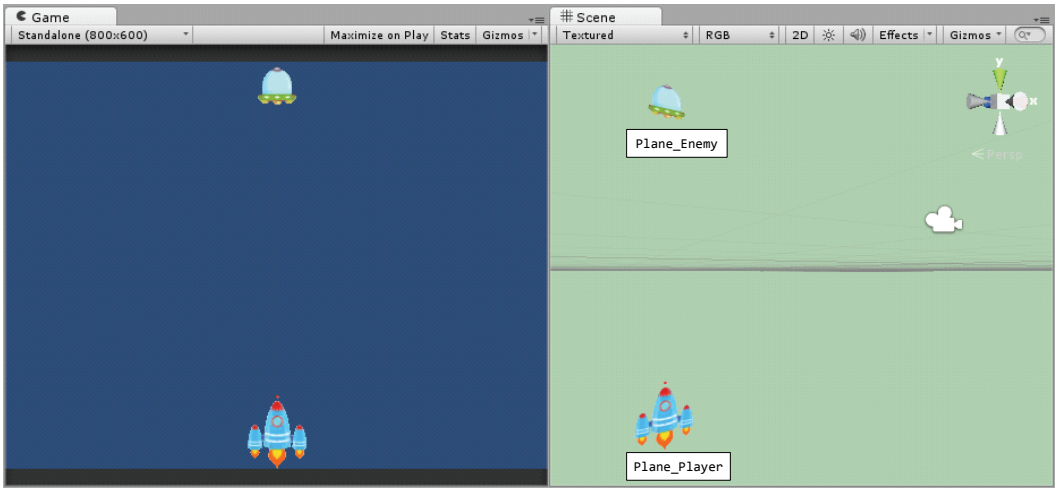


图 3-12　场景和主摄像机视角下的对象图

3.2.3　游戏对象的“碰撞”组件

选中场景中的 2 个 Plane 对象，你将会在查看器中看到名为 Mesh Collider 的组件，如图 3-13 所示，这个组件的作用是检测 2 个游戏对象是否发生“碰撞”。没有这个组件时，2 个对象在“碰撞”时会彼此穿过。

注意：网格碰撞体（Mesh Collider）会对游戏对象使用很多的计算，所以会占用很多的系统资源。对于这样一个示例的游戏这样做没有必要，所以可以使用一个简单的能实现同样功能的组件替换它，这个组件是盒碰撞体（Box Collider）。要使用盒碰撞体组件，可以先删除原来的那个组件，如图 3-14 所示，单击齿轮，在弹出的列表中选择 Remove Component 即可移除这个组件。

然后再添加组件 Box Collider，如图 3-15 所示。选择 Component|Physics|Box Collider 命令，组件就被添加到对象中了，属性 Center 表示碰撞体在对象局部坐标中的位置，Size 表示碰撞体在 X、Y 和 Z 方向上的大小。

这个组件就像是给对象加上了一个立方体的外壳一样，如图 3-16 所示，在场景视图中以绿色的线显示。

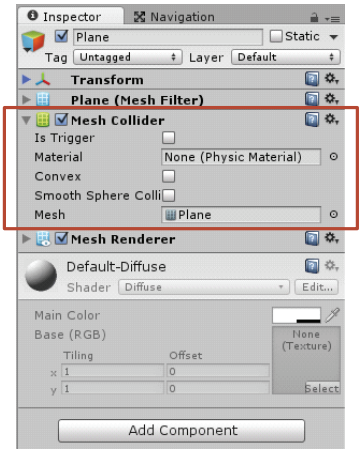


图 3-13　Plane 对象上默认添加的 Mesh Collider 组件

图 3-14　移除 Mesh Collider 组件

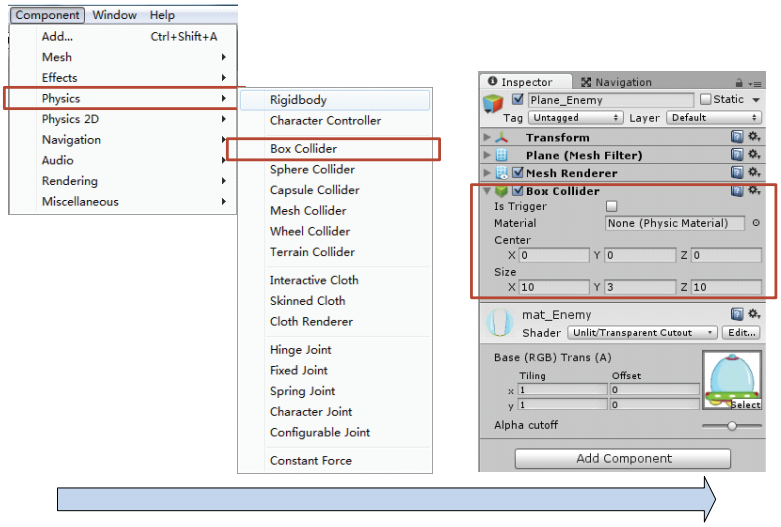


图 3-15　添加 Box Collider 组件（在图中介绍 2 个组件属性 Center 和 Size）

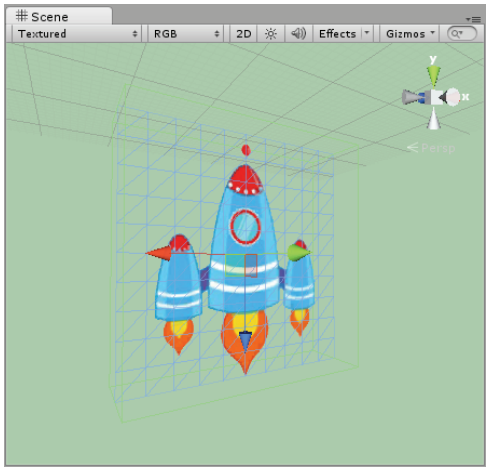


图 3-16　场景视图中游戏对象的“外壳”

在游戏视图中同样可以看到这个绿色的外壳，前提是确保选中视图工具栏上的 Gizmos 按钮，并且选中游戏对象，如图 3-17 所示。

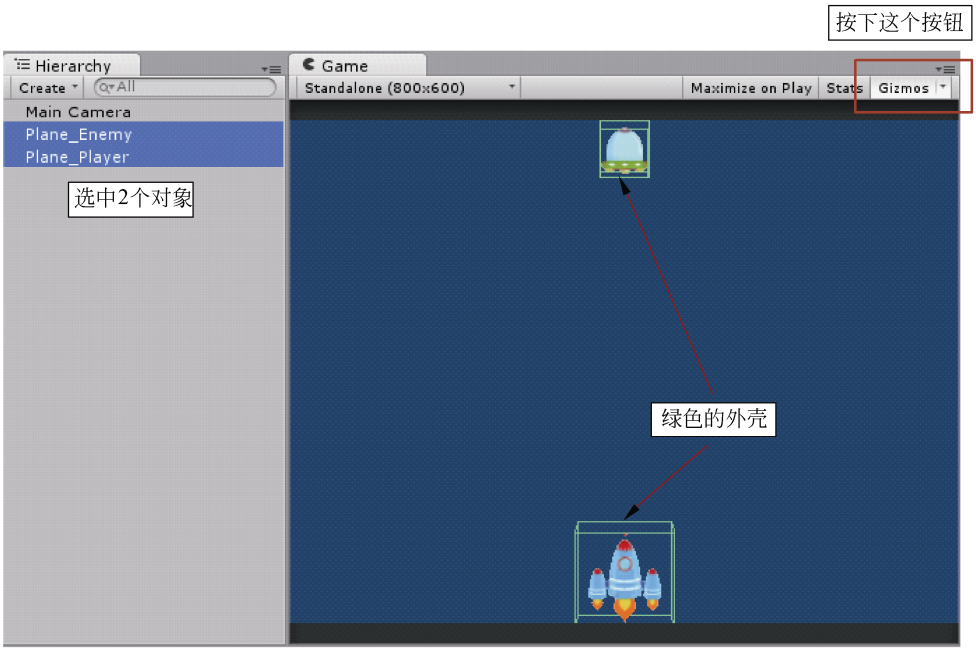


图 3-17　在游戏视图中查看绿色的边框，即 Box Collider

3.3　让飞船动起来

虽然场景中已经有了 2 个游戏对象，但是在游戏真正运行起来时，游戏视图下的对象不会有任何反应，而且也不会移动，整个游戏都死气沉沉的。为了让飞船动起来，使用键盘上的“左和右”方向键控制飞船沿着游戏视图底部左右移动，就需要让飞船引用一个脚本文件。

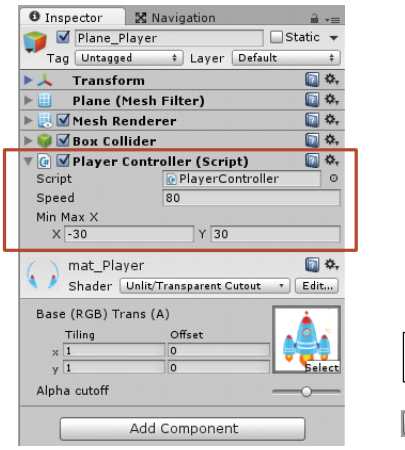
在资源面板的 scripts 文件夹下新建一个 C#脚本文件。为了对这个功能有正确的识别，命名其为 PlayerController。双击这个脚本文件，在 MonoDevelop 中编辑这个脚本，代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class PlayerController : MonoBehaviour {
05
06      public float Speed;                    //飞船每秒移动的单元的个数
07      public Vector2 MinMaxX = Vector2.zero; //保证飞船只在屏幕上左右移动
08      // Use this for initialization
09      void Start () {
10      }
11      // Update is called once per frame
12      void Update () {
13          transform.position =
14              new Vector3 (
```

```
15                  Mathf.Clamp (
16                     transform.position.x + Input.GetAxis ("Horizontal")
                       * Speed * Time.deltaTime,
17                     MinMaxX.x,
18                     MinMaxX.y),
19                  transform.position.y,
20                  transform.position.z);
21        }
22  }
```

使用场景中的 Plane_Player 引用此脚本，Plane_Player 在查看器中就会把这个脚本看做一个组件，如图 3-18 所示。因为脚本中定义的 2 个变量 Speed 和 MinMaxX 都是公有的（Public），所以在查看器中就可以编辑这 2 个变量，在开始游戏之前设置 Speed 为 80，MinMaxX 的两个成员 x 和 y 分别为-30 和 30。

单击快捷菜单上的开始按钮，开始游戏，使用键盘上的“左右”方向键控制飞船左右移动，如图 3-19 所示。在游戏中，再次单击开始按钮，或者按下键盘上的 ESC 键就可以退出游戏。



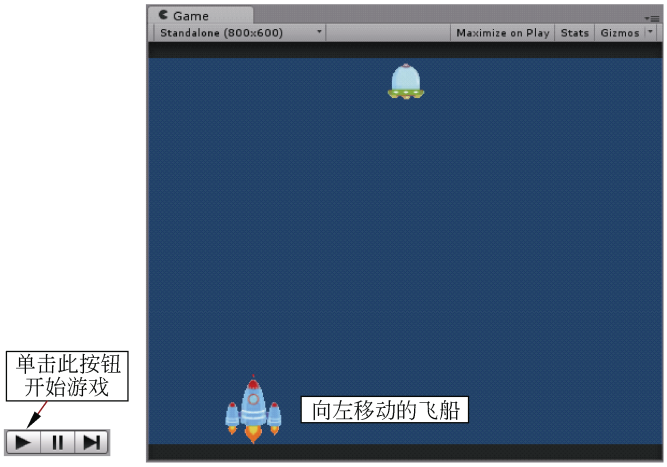


图 3-18　被对象引用后成为组件的脚本　　图 3-19　游戏时，可以移动飞船

如果觉得飞船移动得太快，可以适当减小 Speed 变量的值。

3.4　让飞船发射子弹

虽然前面编写的 C#脚本文件使得飞船可以左右移动，我们甚至可以控制飞船移动的速度和范围，但是此时的飞船却无法攻击外星人的飞船。因此本节打算给飞船搭载子弹，而飞船开火的时机是：玩家按下鼠标左键或者键盘左面的 Ctrl 键。

3.4.1　在场景中添加子弹

使用同样的方法在场景中添加子弹对象，即先添加 Plane 对象，然后让对象引用前面

准备好的名为 mat_Ammo 的材质。为了区分场景中的对象，此时最好给 Plane 起个其他的名字，如 Plane_Ammo。然后，调整子弹对象的位置和角度，必要的时候还要调整对象的大小，因为从真实性的角度来看，子弹不应该比飞船还大。在场景中添加的子弹对象，以及游戏中的视图效果如图 3-20 所示。

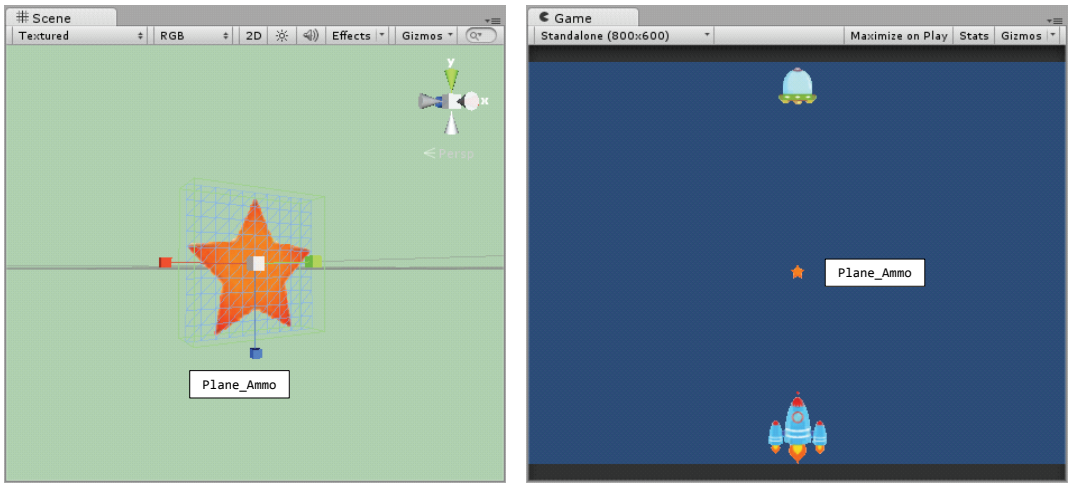


图 3-20　添加了子弹对象的场景和游戏视图

注意：虽然说，可以按照“近大远小”的原则来改变子弹对象在游戏视图上的大小，即让子弹对象远离摄像机。但这样做会让场景中的 3 个游戏对象处在不同的深度平面，那么 3 个对象在各自的平面上运动的话就永远也不可能相遇，对象上的“碰撞检测”组件就失去了意义，如果子弹不和外星人的飞船发生碰撞就让飞船爆炸是不合逻辑的。因此最好让 3 个游戏对象处在同一深度平面上，即 Z 轴方向的值应该是相等的，如图 3-21 所示。

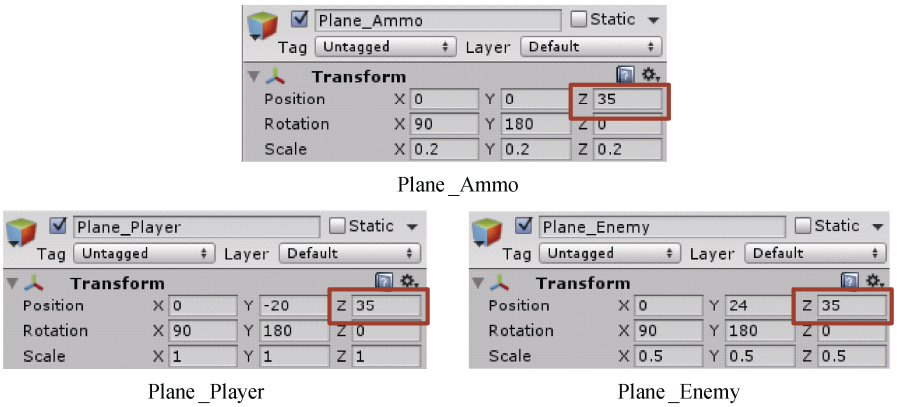


图 3-21　处于同一深度平面的 3 个游戏对象

如果飞船对象和外星人飞船对象都改变了“碰撞”组件为 Box Collider，那么子弹对象应该做同样的更改，即删除原来的 Mesh Collider 组件，然后添加 Box Collider 组件，如图 3-22 所示。

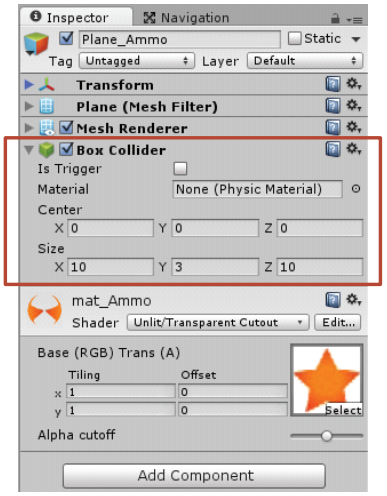


图 3-22 “碰撞”组件为 Box Collider 的子弹对象

3.4.2 游戏时，让子弹在场景中移动

同样，游戏时子弹被宇宙飞船发射后需要一直向上移动，直到击中外星人的宇宙飞船，或者移动到屏幕的外边为止。为了让子弹移动，需要使用 C#脚本来实现，我们给脚本命名为 Ammo。双击这个脚本文件，在 MonoDevelop 中编辑这个脚本，代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class Ammo : MonoBehaviour {
05      //移动的方向
06      public Vector3 Direction = Vector3.up;
07      //移动的速度
08      public float Speed = 20.0f;
09      //lifetime in seconds
10      public float Lifetime = 10.0f;
11
12      // Use this for initialization
13      void Start () {
14          //destroys ammo in lifetime
15          Invoke ("DestroyMe", Lifetime);
16
17      }
18
19      // Update is called once per frame
20      void Update () {
21          //以一定的速度改变子弹的移动位置
22          transform.position += Direction * Speed * Time.deltaTime;
23      }
24
25      void DestroyMe(){
26          //销毁场景里的子弹对象
27          Destroy (gameObject);
28      }
29  }
```

代码中一共定义了 3 个公有的变量：Direction、Speed 和 Lifetime。分别用于控制子弹的移动方向、移动速度和子弹被销毁的时间。Lifetime 指定了子弹在多少秒以后销毁，不管这个子弹处于何种状态。因为 3 个变量都被 public 修饰，所以能在 Unity 的查看器中动态修改这些变量的值，如图 3-23 所示。

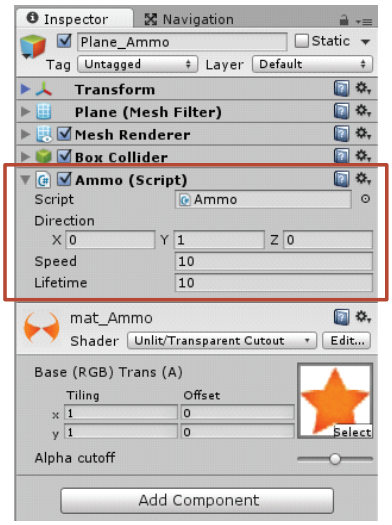


图 3-23　子弹对象查看器脚本文件中可以被修改的变量

提示：销毁子弹是为了避免计算机计算资源的浪费，因为当子弹飞出屏幕以后就再也没有存在的必要了。不销毁子弹，计算机会继续计算并保存子弹对象在空间中的位置，而这个数据毫无意义。

运行这个场景，子弹会向上（Y 轴正向）移动，并在 10 秒后自动销毁，如图 3-24 所示。

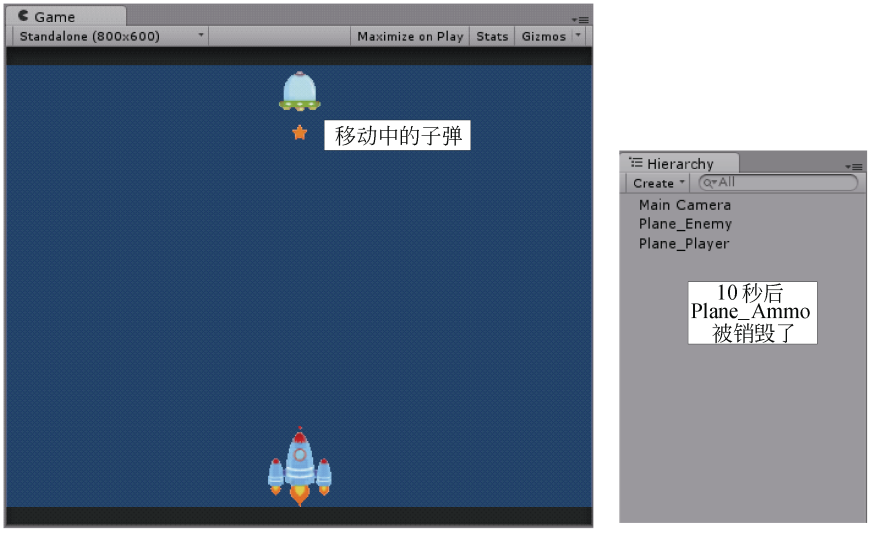


图 3-24　引用了脚本的子弹对象

提示：游戏处于运行状态时，Unity 的面板还会动态显示一些参数，例如：场景中都有哪些对象，对象在空间中的位置，图 3-25 是子弹对象在游戏的 2 个时刻显示在查看器中的位置。从 Y 值的变化（从-4.3803 到 13.0713），也可以看出子弹一直在向 Y 轴的正方向移动。

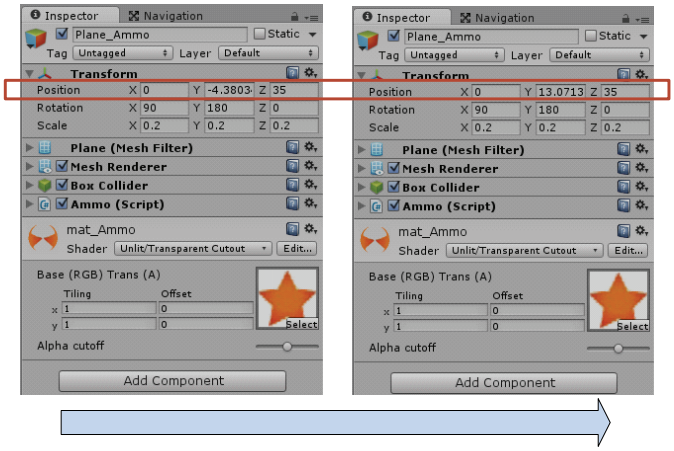


图 3-25　游戏中子弹对象的位置变化

3.4.3　生成子弹的预设

子弹是可以移动了，但我们希望子弹是在按下“发射”按钮以后才出现并移动的。为了实现这个功能，可以使用“预设”（Prefab）。预设是一种资源，它可以变成场景中任何其他的对象的一个副本。本游戏打算使用预设生成子弹的副本。

选择 Assets|Create|Prefab 命令即可生成预设，如图 3-26 所示，给其命名为 Prefab_Ammo。要让预设生成子弹的副本，就需要拖动子弹对象到预设上，预设与子弹对象建立关联后的预设如图 3-27 所示。

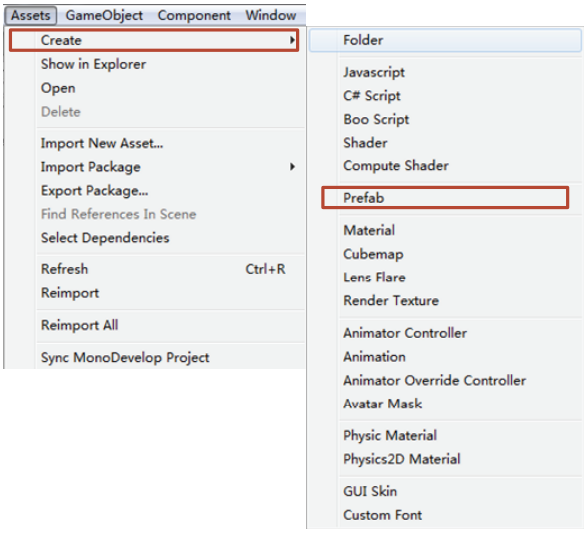


图 3-26　生成预设

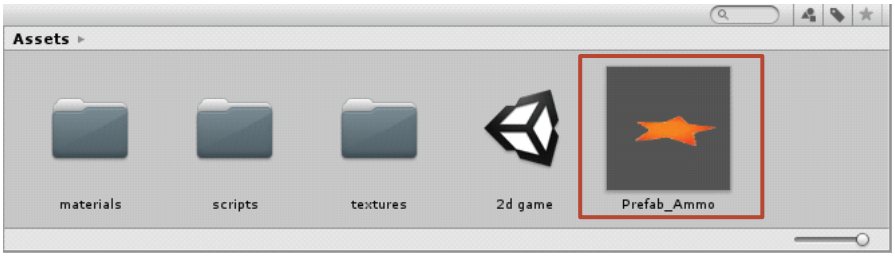


图 3-27　与子弹对象建立关联后的预设

3.4.4　设置子弹的发射位置

发射子弹时，子弹一定有一个起始的位置。子弹在这个位置出现然后移动，很显然这个位置应该是在飞船的顶端，或者其他发射子弹的地方。本游戏将使用虚拟游戏对象（Dummy GameObject）来标记这个位置，创建这个对象的方法是：选择 GameObject|Create Empty 命令，如图 3-28 所示。我们把这个对象命名为 Canno_dummy。

之所以说这个对象是虚拟的，是因为它不可见，且此对象只有一个组件：Transform，控制对象的位置、角度和大小。但没关系，因为在游戏中这个对象只是用来标记子弹发射位置的。如果非要在场景中找寻这个对象的位置，可以单击层次面板中这个对象的名字，然后场景视图会自动定位到这个对象。

注意：创建一个空的对象也可以使用快捷键 Ctrl+Shift+N。有快捷键也间接的暗示了这个对象的重要性，以及使用频率。

子弹发射的位置应该随着飞船的移动而移动，而发射位置与飞船间也有相对位置，为了让子弹与飞船的相对位置固定，且能随着飞船的移动而移动，就需要把前面创建的 Canno_dummy 对象设置为飞船的子对象。具体的实现操作是：拖动 Canno_dummy 对象到 Plane_Player 对象上，然后在层次视图上会展示出这个“父子”对象关系，如图 3-29 所示。

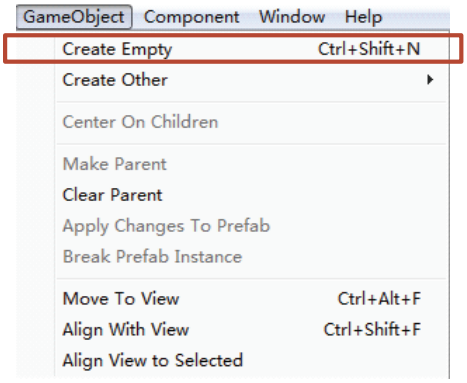


图 3-28　生成虚拟游戏对象

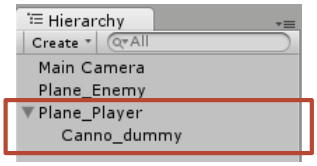


图 3-29　层次视图中的“父子”对象

此时，Canno_dummy 的坐标就是相对于飞船的位置了，而不是飞船所在坐标系的位置，如图 3-30 所示。

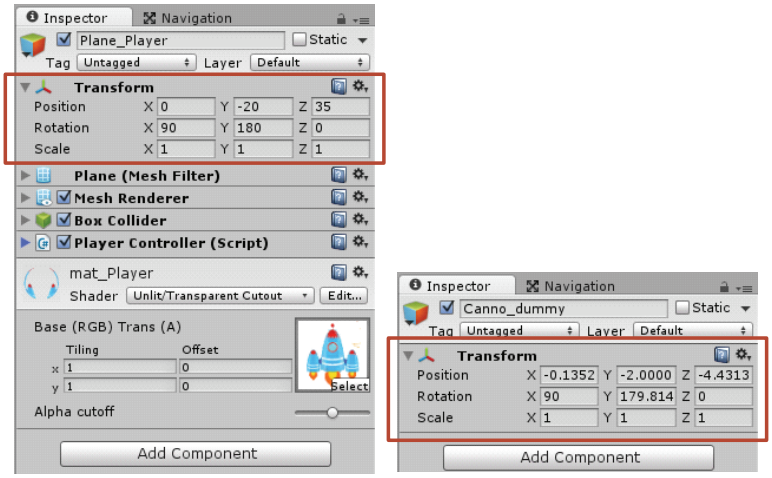


图 3-30　处于 2 个不同坐标系的 2 个游戏对象

移动 Canno_dummy 对象到子弹发射的位置，本游戏设定子弹的发生位置是飞船的顶端，如图 3-31 所示。

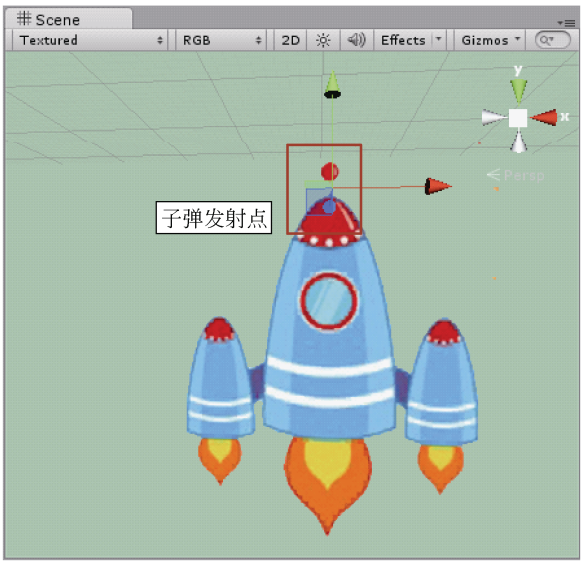


图 3-31　飞船发射子弹的位置，即对象 Canno_dummy 的位置

3.4.5　在恰当的时机发射子弹

此时，子弹的预设有了，子弹的发射位置也设置好了。当玩家按下发射按钮（即鼠标左键，或者键盘上左边的 Ctrl 键）时，我们将在子弹的发射位置初始化一个子弹预设的实例。具体的操作是：修改之前编写的 PlayerController 脚本文件，代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
```

```
04  public class PlayerController : MonoBehaviour {
05
06      //飞船的生命值
07      public int Health = 100;
08      public float Speed;
09      //子弹两发的间隔
10      public float ReloadDelay = 0.2f;
11      public Vector2 MinMaxX = Vector2.zero;
12      public GameObject PrefabAmmo = null;
13      public GameObject GunPosition = null;
14      private bool WeaponsActivated = true;
15      // Use this for initialization
16      void Start () {
17
18      }
19      // Update is called once per frame
20      void Update () {
21          transform.position =
22          new Vector3 (
23              Mathf.Clamp (
24              transform.position.x + Input.GetAxis ("Horizontal") * Speed
                * Time.deltaTime,
25              MinMaxX.x, MinMaxX.y),transform.position.y, transform.
                position.z
26                  );
27      }
28      void LateUpdate(){
29          //当按下发射按钮
30          if (Input.GetButton ("Fire1") && WeaponsActivated) {
31              //创建一个新的子弹
32              Instantiate (PrefabAmmo,GunPosition.transform.position,
33              PrefabAmmo.transform.rotation);
34              WeaponsActivated = false;
35              Invoke ("ActivateWeapons",ReloadDelay);
36          }
37      }
38      //允许发射子弹
39      void ActivateWeapons(){
40          WeaponsActivated = true;
41      }
42  }
```

脚本中又添加了 4 个公有变量和 1 个私有变量，公有变量会显示在查看器中，如图 3-32 所示。

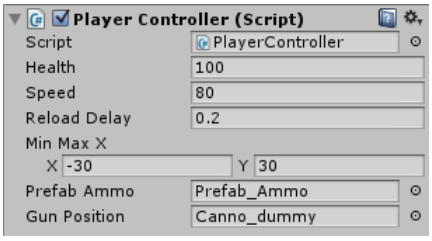


图 3-32　查看器中脚本的公有变量

其中，PrefabAmmo 变量应该被赋予子弹预设对象，这表明预设对象应该被初始化为子弹对象。GunPosition 变量应该被赋予 Canno_dummy 对象，表明子弹的发射位置。给

PrefabAmmo 和 GunPosition 变量赋值的方式是：单击文本框右面的⊙，在弹出的对话框中选择相应的对象即可，如图 3-33 所示。

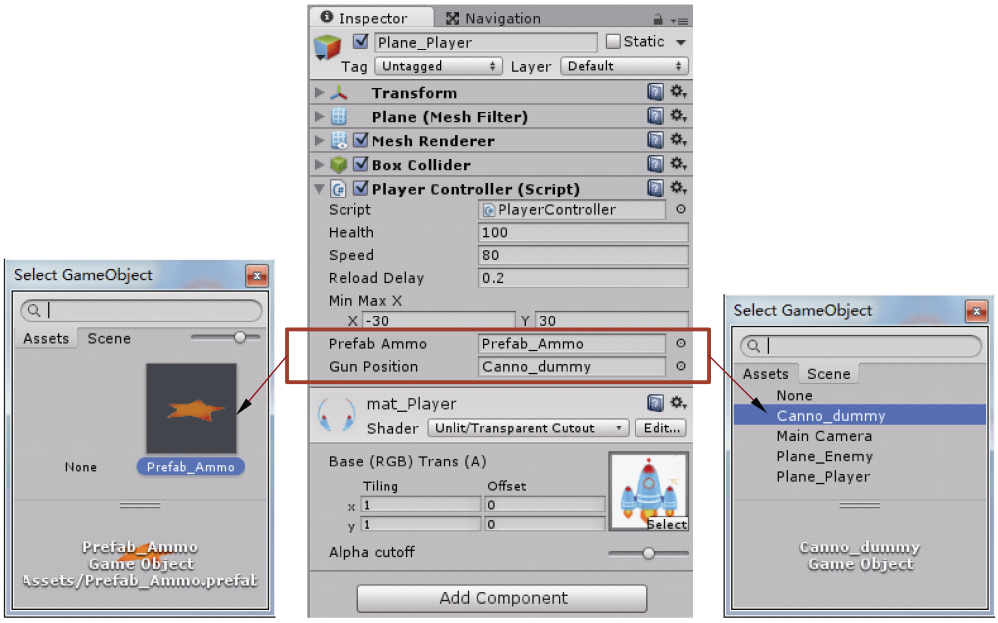


图 3-33　为脚本中的 2 个公有变量赋值

还记得之前在场景中创建的 Plane_Ammo 对象吗？因为现在打算在游戏运行时动态地创建这个对象，所以这个对象应该被删掉了。现在，再次运行游戏，飞船就可以在任何水平位置，任何时间发射子弹了，如图 3-34 所示，当然，此时外星人的飞船被子弹击中时并不会消失，因为还没有实现那部分功能。

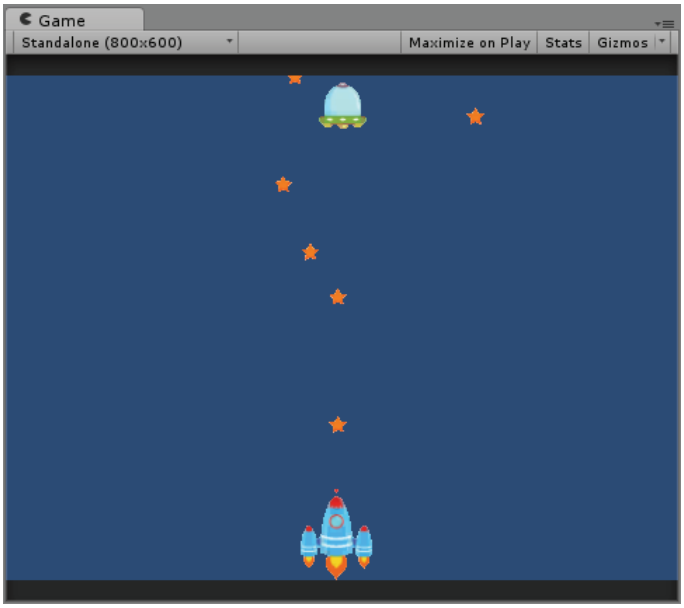


图 3-34　游戏时，自由发射子弹的飞船

3.5　让外星飞船动起来

在本游戏中，外星飞船可以自动左右移动，而且在外星飞船与子弹发生碰撞的时候，外星飞船应该爆炸。

3.5.1　编写脚本

为了让外星飞船动起来，同样需要为外星飞船编写脚本文件，我们给它命名为 EnemyController，代码如下：

```
01  using UnityEngine;
02  using System.Collections;
03
04  public class EnemyController : MonoBehaviour {
05      //外星飞船的生命值
06      public int Health = 100;
07      //外星飞船每秒移动的单元个数
08      public float Speed = 1.0f;
09      //外星飞船的移动范围
10      public Vector2 MinMaxX = Vector2.zero;
11      // Use this for initialization
12      void Start () {
13
14      }
15      // Update is called once per frame
16      void Update () {
17          transform.position = new Vector3 (
18              MinMaxX.x + Mathf.PingPong (Time.time * Speed, 1.0f) *
                (MinMaxX.y - MinMaxX.x),
19              transform.position.y, transform.position.z);
20      }
21      void OnTriggerEnter(Collider other)            //触发器的入口
22      {
23          Destroy (gameObject);
24          Destroy (other.gameObject);
25      }
26  }
```

从代码中来看，外星飞船和我们的宇宙飞船有很多类似的地方：同样多的声明值（即 100），同样的移动范围，同样可以在查看器中修改对象的速度、范围。当然也有不同的地方：外星飞船是自己来回移动的，而宇宙飞船是由玩家控制移动的，且可以发射子弹。

让外星飞船引用这个脚本文件以后，在查看器中手动设置外星飞船的移动范围，设置的变量是 MinMaxX 的 X 和 Y 值，如图 3-35 所示，设置 X 为-30，Y 为 30。

设置好了外星飞船的移动范围以后，直接运行游戏，可以看到外星飞船开始持续左右移动，如图 3-36 所示。但是当子弹击中外星飞船的时候，并不摧毁，这是因为脚本文件中的触发器 OnTriggerEnter 没有被调用。原因有 2 个：

第一：外星飞船的 Box Collider 组件应该被设置为触发器。这使得外星飞船被标准的 Unity 物理系统忽略，且仅能收到一种消息：其他对象与外星飞船对象发生了交叉，或者

说重叠。

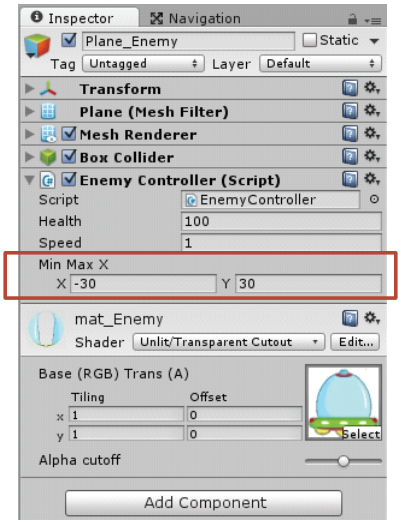


图 3-35　在查看器中设置外星飞船的移动范围

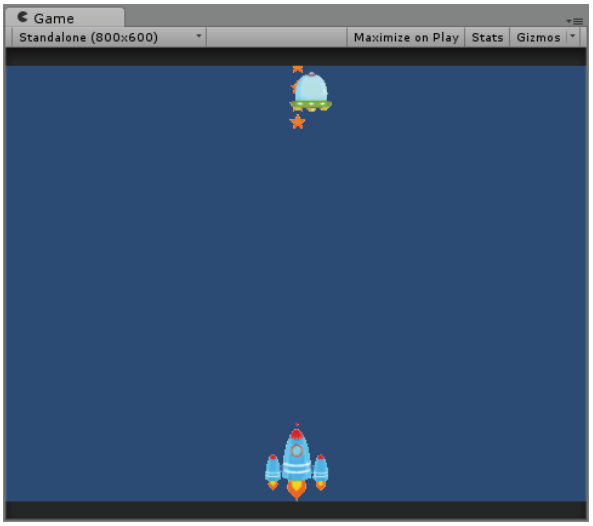


图 3-36　游戏时，左右来回移动的外星飞船

第二：子弹对象必须有刚体（RigidBody）组件。这个组件是必要的，因为 OnTriggerEnter 事件只有在刚体进入到外星飞船的范围内才会被触发。

3.5.2　设置外星飞船的触发器

添加到外星飞船上的 Box Collider 组件，有一个属性复选框，名为 Is Trigger，默认情况下这个复选框是未被选中的，现在将其复选即可，如图 3-37 所示。

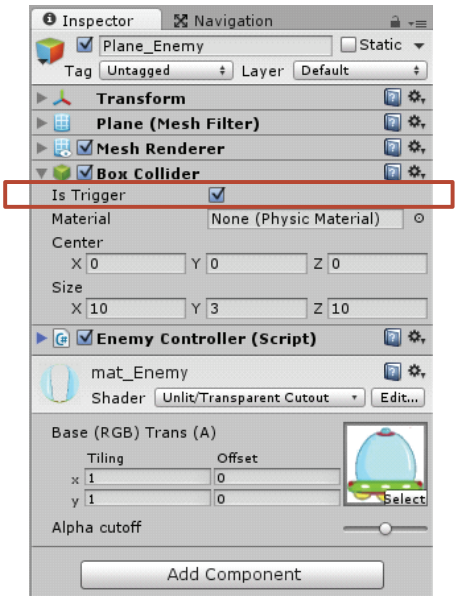


图 3-37　复选外星飞船 Box Collider 组件中的 Is Trigger 属性

3.5.3　为子弹预设添加刚体组件

选中资源面板中的预设子弹资源，然后在查看器中单击此预设资源的 Add Component 按钮，然后依次选择 Physics 中的 Rigidbody，如图 3-38 所示。

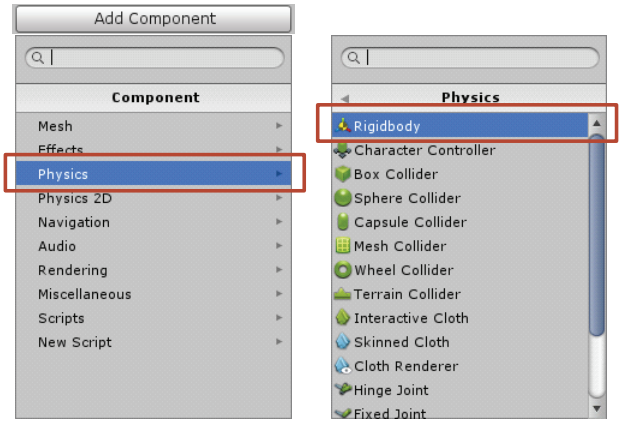


图 3-38　为预设子弹资源添加 Rigidbody 组件

如果不修改刚体组件的默认属性设置，预设子弹资源会受到重力的影响，即当宇宙飞船向上发射子弹以后，子弹会受到重力的影响，向屏幕内运动，如图 3-39 所示。

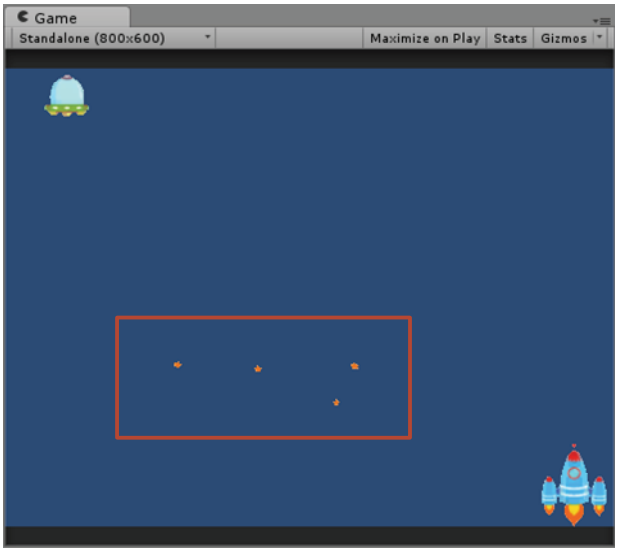


图 3-39　受到重力影响的子弹，在向下坠落

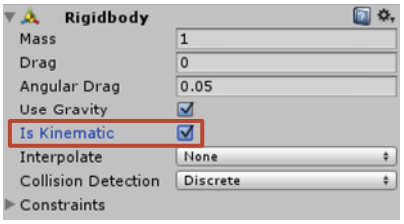


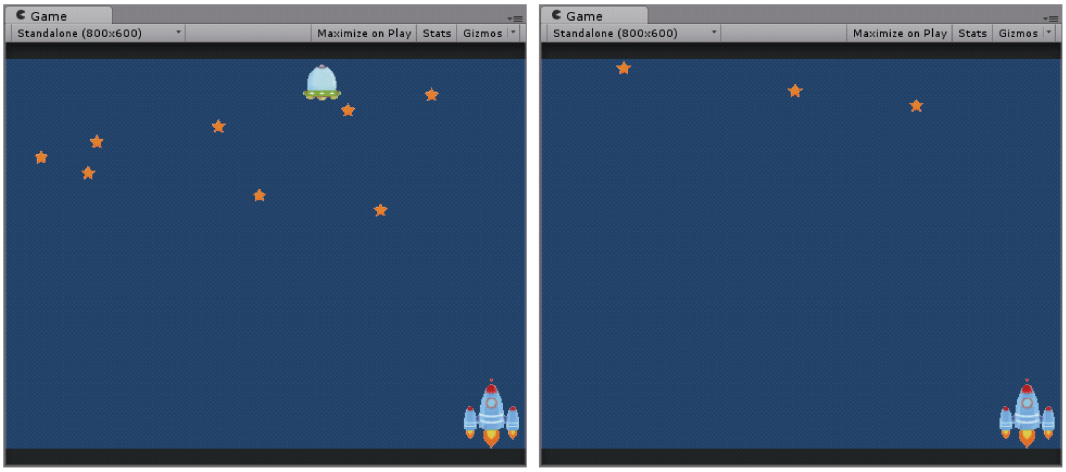
图 3-40　复选刚体组件的一个复选框

所以在为预设子弹添加了刚体组件以后，还需要做一个简单的属性设置：复选刚体组件的 Is Kinematic 属性复选框，如图 3-40 所示。这个复选框相当于一个开关，即是否开启动力学的开关。选中它之后游戏对象将不再受物理引擎（包括重力）的影响，而只能通过 Transform 属性来改变这个对象的位置。

现在再次运行游戏，本游戏的基本功能就全部完成了，即：

- ❑ 宇宙飞船可以在玩家的控制下左右移动，以及发射子弹。
- ❑ 外星飞船可以自动地左右持续运动，当被子弹击中时会销毁。

游戏的运行效果如图 3-41 所示。



击中前的外星飞船　　击中后的外星飞船

图 3-41　运行游戏时，击中外星飞船的前后

3.6　为游戏添加背景

为游戏添加背景可以增强我们游戏场景的真实性。本节打算导入 Unity 自带的包，即天空盒子（Skyboxes），里面提供了一系列的天气纹理。导入天空盒子包的方法如图 3-42 所示。选择 Assets|Import Package|Skyboxes 命令，然后在接下来弹出的 Importing package 中保留默认选项，然后单击 Import 按钮导入包中的纹理资源。

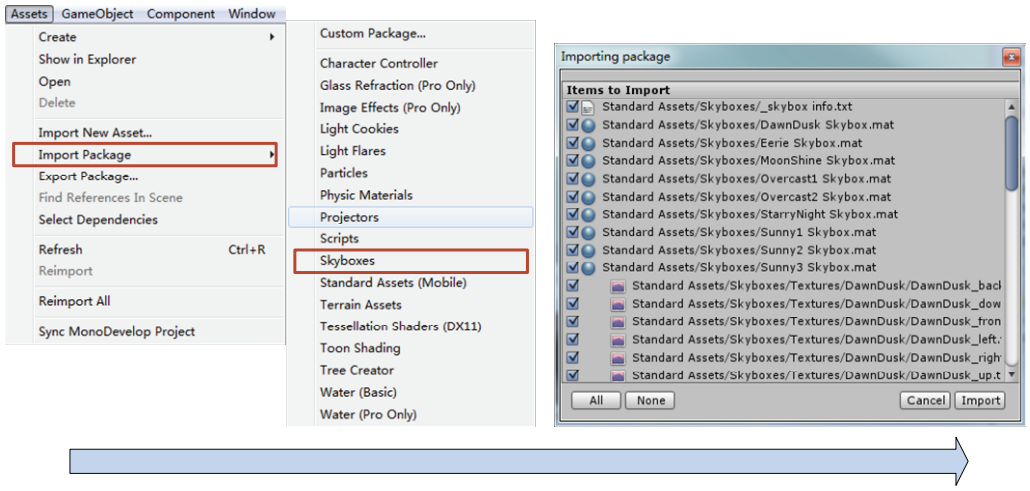


图 3-42　为项目导入天空盒子包

补充：Skybox 实际上是一种特殊着色器类型的材质，此种材质可以笼罩在整个游戏场景上。

接下来要为场景中的摄像机添加名为 Skybox 的组件。具体操作是：单击摄像机查看器中的 Add Component 按钮，然后依次选择 Rendering 和 Skybox 即可，如图 3-43 所示。

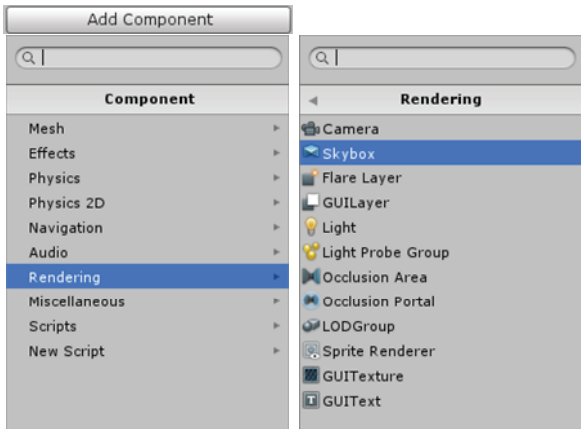


图 3-43　为摄像机添加 Skybox 组件

这个组件只有一个自定义天空盒子（Custom Skybox）属性，给这个属性赋予一个导入的包中的纹理，本示例选择的纹理是 Sunny2 Skybox，效果会即时的显示在游戏视图中，此时游戏的运行效果如图 3-44 所示。

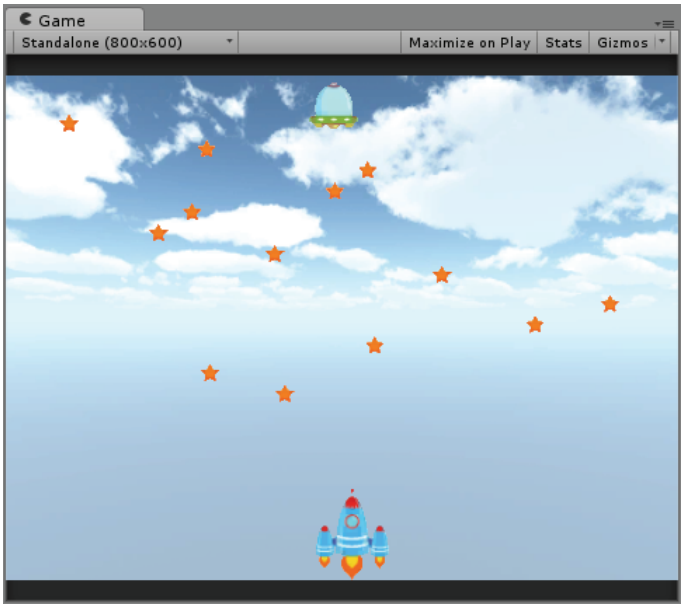


图 3-44　添加了背景的游戏运行效果