第 4 章
塔防游戏

本章将使用 Unity 完成一款塔防游戏。我们将使用自定义的编辑器创建场景，创建路点引导敌人行动，对战斗进行配置、动画播放，还涉及摄像机控制和 UI 界面等。

4.1 策　　划

移动平台上的塔防游戏已经多得数不胜数，笔者也曾经开发过一款叫做《野人大作战》的塔防游戏（英文名为 Wild Defense）。塔防游戏的基本玩法比较类似，在场景中我方有一个基地，敌人从场景的另一侧出发，沿着相对固定的路线攻打基地。我方可以在地图上布置防守单位，攻击前来进攻的敌人，防止他们闯入基地。

本章也将制作一款塔防游戏，其具备塔防游戏的最基本要素。

4.1.1 场景

塔防游戏的场景有些固定的模式，它由一个二维的单元格组成，每个格子的用途可能都不同：

- 专用于摆放防守单位的格子。
- 专用于敌人通过的格子。
- 既无法摆放防守单位，也不允许敌人通过的格子。

4.1.2 摄像机

摄像机始终由上至下俯视游戏场景，按住鼠标左键并移动可以移动摄像机的位置。

4.1.3 胜负判定

我方基地有 10 点生命值，敌人攻入基地一次减少一点生命值，当生命值为 0，游戏失败。

敌人以波数的形式向我方基地进攻，每波由若干个敌人组成。在这个实例中，一关有 10 波，当成功击退敌人 10 波的进攻则游戏胜利。

4.1.4 敌人

敌人有两种：一种在陆地上行走；另一种可以飞行。每打倒一个敌人会奖励一些铜钱，用来购买新的防守单位。

4.1.5 防守单位

塔防游戏会有多种类型的防守单位，为了使本篇教程尽可能简单，我们只完成两种类型的防守单位：一种是近战类型；另一种是远程。每造一个防守单位需要消耗相应数量的铜钱。

4.1.6 UI 界面

游戏中的 UI 包括防守单位的按钮、敌人的进攻波数、基地的生命值和铜钱数量。

当防守单位攻击敌人时，在敌人的头上需要显示一个生命条表示剩余的生命值。

当游戏失败或胜利后显示一个按钮重新游戏。

4.2 地图编辑器

在开始正式制作游戏之前，我们有必要先完成一个塔防游戏的地图编辑器。Unity 编辑器的自定义功能非常强大，几乎可以把 Unity 编辑器扩展成任何界面。在示例中，我们将完成一个“格子”编辑系统，帮助我们输入塔防游戏的地图信息。

4.2.1 “格子”数据

新建工程，在 Hierarchy 窗口中单击鼠标右键，选择【Create Empty】创建一个空物体，然后创建脚本 TileObject.cs 指定给空物体，这里将空物体命名为 Grid Object，这个类主要用于保存场景中的“格子”数据，代码如下所示。

```
using UnityEngine;
public class TileObject : MonoBehaviour {

    public static TileObject Instance = null;

    // tile  碰撞层
    public LayerMask tileLayer;
    // tile  大小
    public float tileSize = 1;
    // x  轴方向 tile 数量
    public int xTileCount = 2;
    // z  轴方向 tile 数量
    public int zTileCount = 2;
    //  格子的数值，0 表示锁定，无法摆放任何物体。1 表示敌人通道，2 表示可摆放防守单位
    public int[] data;
    //  当前数据  id
    [HideInInspector]
    public int dataID = 0;
    [HideInInspector]
    //  是否显示数据信息
    public bool debug = false;
    void Awake()
    {
        Instance = this;
    }
    //  初始化地图数据
    public void Reset()
    {
        data = new int[xTileCount * zTileCount];
    }
```

```
        // 获得相应 tile 的数值
        public int getDataFromPosition(float pox, float poz)
        {
            int index = (int)((pox - transform.position.x)/ tileSize) * zTileCount + (int)((poz -
transform.position.z)/tileSize);
            if (index < 0 || index >= data.Length) return 0;
            return data[index];
        }
        // 设置相应 tile 的数值
        public void setDataFromPosition( float pox, float poz, int number )
        {
            int index = (int)((pox - transform.position.x) / tileSize) * zTileCount + (int)((poz -
transform.position.z) / tileSize);
            if (index < 0 || index >= data.Length) return;
            data[index] = number;
        }
        // 在编辑模式显示帮助信息
        void OnDrawGizmos()
        {
            if (!debug)
                return;
            if (data==null)
            {
                Debug.Log("Please reset data first");
                return;
            }
            Vector3 pos = transform.position;
            for (int i = 0; i < xTileCount; i++)    // 画 z 方向轴辅助线
            {
                Gizmos.color = new Color(0, 0, 1, 1);
                Gizmos.DrawLine(pos + new Vector3(tileSize * i, pos.y, 0),
                    transform.TransformPoint(tileSize * i, pos.y, tileSize * zTileCount));
                for (int k = 0; k < zTileCount; k++)    // 高亮显示当前数值的格子
                {
                    if ( (i * zTileCount + k) < data.Length && data[i * zTileCount + k] == dataID)
                    {
                        Gizmos.color = new Color(1, 0, 0, 0.3f);
                        Gizmos.DrawCube(new Vector3(pos.x + i * tileSize + tileSize * 0.5f,
                                pos.y, pos.z + k * tileSize + tileSize * 0.5f), new Vector3(tileSize, 0.2f,
tileSize));
                    }
```

```
                }
            }
            for (int k = 0; k < zTileCount; k++)   // 画 x 方向轴辅助线
            {
                Gizmos.color = new Color(0, 0, 1, 1);
                Gizmos.DrawLine(pos + new Vector3(0, pos.y, tileSize * k),
                    this.transform.TransformPoint(tileSize * xTileCount, pos.y, tileSize * k));
            }
        }
    }
```

因为 Unity 目前不支持二维数组的序列化，所以本示例使用了一维数组 data 保存地图 x、y 的信息。GetDataFromPosition 函数通过输入的坐标位置获取 data 数组的下标，其中一步计算是由输入的坐标减去当前物体 transform 的坐标值，这里要注意浮点数精度问题，比如有时 1.54～0.54 会得到 0.9999999 的结果（实际应当是 1），0.9999999 在转为整数后就会变为零，为了避免这个问题，最好将 Grid Object 的 transform 坐标值设为整数。

4.2.2 在 Inspector 窗口添加自定义 UI 控件

在 Unity 的编辑器中，当选中一个游戏体后，我们即可在【Inspector】窗口中设置它的详细属性。默认【Inspector】窗口中的选项都是预定的，Unity 提供了 API 可以扩展【Inspector】窗口中的 UI 控件。

步骤 01 以本示例的地图编辑器为例，为了扩展 TileObject 这个类的【Inspector】窗口，我们创建了脚本 TileEditor.cs，继承自 Editor。因为它是一个编辑器脚本，所以必须放到 Editor 文件夹中，代码如下：

```
using UnityEngine;
using UnityEditor;

 [CustomEditor(typeof(TileObject))]
public class TileEditor : Editor {

    // 是否处于编辑模式
    protected bool editMode = false;
    // 受编辑器影响的 tile 脚本
    protected TileObject tileObject;
    void OnEnable()
    {
        // 获得 tile 脚本
        tileObject = (TileObject)target;
    }

    // 更改场景中的操作
    public void OnSceneGUI()
    {
```

```
            if (editMode)   // 如果在编辑模式
            {
                // 取消编辑器的选择功能
                HandleUtility.AddDefaultControl(GUIUtility.GetControlID(FocusType.Passive));
                // 在编辑器中显示数据（画出辅助线）
                tileObject.debug = true;
                // 获取 Input 事件
                Event e = Event.current;

                // 如果是鼠标左键
                if ( e.button == 0 && (e.type == EventType.MouseDown || e.type == EventType.MouseDrag)
&& !e.alt)
                {
                    // 获取由鼠标位置产生的射线
                    Ray ray = HandleUtility.GUIPointToWorldRay(e.mousePosition);

                    // 计算碰撞
                    RaycastHit hitinfo;
                    if (Physics.Raycast(ray, out hitinfo, 2000, tileObject.tileLayer))
                    {
                        tileObject.setDataFromPosition(hitinfo.point.x, hitinfo.point.z, tileObject.dataID);
                    }
                }
            }
            HandleUtility.Repaint();
        }

        // 自定义 Inspector 窗口的 UI
        public override void OnInspectorGUI()
        {
            GUILayout.Label("Tile Editor"); // 显示编辑器名称
            editMode = EditorGUILayout.Toggle("Edit", editMode);   // 是否启用编辑模式
            tileObject.debug = EditorGUILayout.Toggle("Debug", tileObject.debug);   // 是否显示帮助信息

            string[] editDataStr = { "Dead", "Road", "Guard" };
            tileObject.dataID = GUILayout.Toolbar(tileObject.dataID, editDataStr);

            EditorGUILayout.Separator();   // 分隔符
            if (GUILayout.Button("Reset" ))     // 重置按钮
            {
                tileObject.Reset();// 初始化
            }
            DrawDefaultInspector();
        }
}
```

步骤 02 添加一个碰撞 Layer，这里设为 tile，然后将 Tile Layer 设为 tile，调整 Tile Count 的值即可改变地图大小；单击 Reset 按钮初始化数据，默认所有格子的值为 0，如图 4-1 所示。

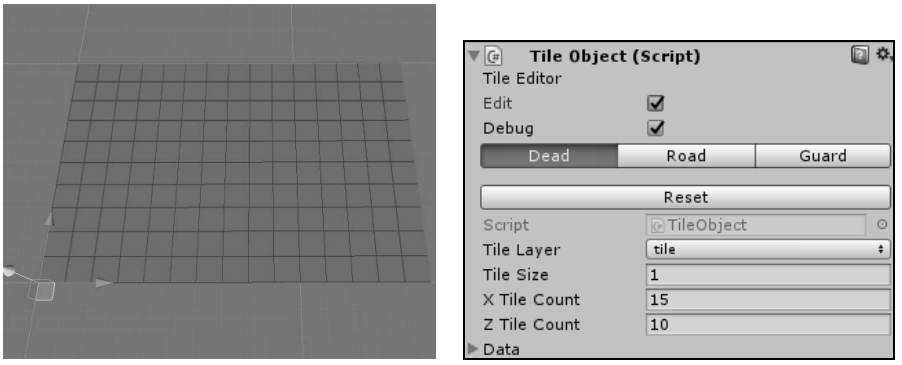

图 4-1　地图格子

步骤 03 在 Hierarchy 窗口中单击鼠标右键，选择【3D Object】→【Plane】，创建一个平面并置于 Grid Object 层级下，将它的 Layer 设为 tile，并取消选中【Mesh Renderer】复选框，我们主要使用它作为地面的碰撞层，如图 4-2 所示。

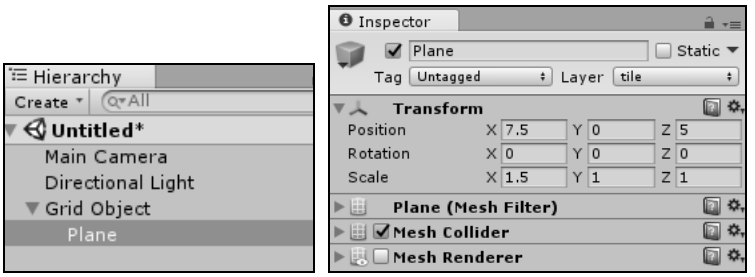

图 4-2　创建碰撞物体

步骤 04 最后，选中【Edit】复选框，单击 Dead（值 0）、Road（值 1）或 Guard（值 2）按钮就可以随意绘制地图数据了，如图 4-3 所示。

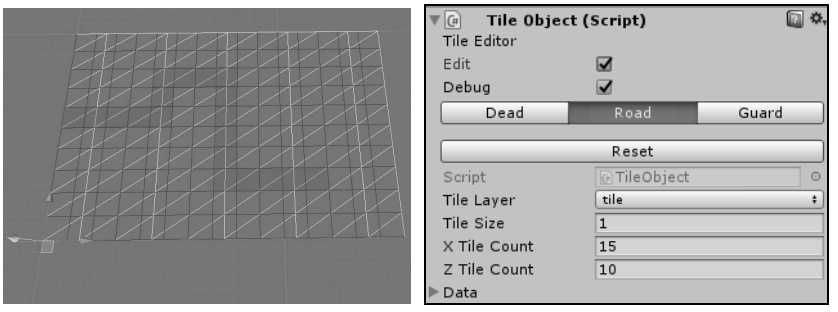

图 4-3　选中【Edit】复选框

4.2.3　创建一个自定义窗口

除了自定义 Inspector 窗口，我们还可以创建一个独立的窗口编辑游戏中的设置，在 Editor 文件夹中创建自定义窗口脚本，示例代码如下：

```
using UnityEngine;
using UnityEditor;

public class TileWnd : UnityEditor.EditorWindow   // 必须继承 EditorWindow
{
    // tile 脚本
    protected static TileObject tileObject;

    // 添加菜单栏选项
    [MenuItem("Tools/Tile Window")]
    static void Create()
    {
        EditorWindow.GetWindow(typeof(TileWnd));

        // 在场景中选中 TileObject 脚本实例
        if (Selection.activeTransform!=null)
            tileObject = Selection.activeTransform.GetComponent<TileObject>();
    }
    // 当更新选中新物体
    void OnSelectionChange()
    {
        if (Selection.activeTransform != null)
            tileObject = Selection.activeTransform.GetComponent<TileObject>();
    }

    // 显示窗口 UI，大部分 UI 函数都在 GUILayout 和 EditorGUILayout 内
    void OnGUI()
    {
        if (tileObject == null)
            return;
        // 显示编辑器名称
        GUILayout.Label("Tile Editor");
        // 在工程目录读取一张贴图
        var tex = AssetDatabase.LoadAssetAtPath<Texture2D>("Assets/GUI/butPlayer1.png");
        // 将贴图显示在窗口内
        GUILayout.Label(tex);
        // 是否显示 Tile Object 帮助信息
        tileObject.debug = EditorGUILayout.Toggle("Debug", tileObject.debug);
        // 切换 Tile Object 的数据
        string[] editDataStr = { "Dead", "Road", "Guard" };
        tileObject.dataID = GUILayout.Toolbar(tileObject.dataID, editDataStr);
        EditorGUILayout.Separator();        // 分隔符
        if (GUILayout.Button("Reset"))     // 重置按钮
        {
```

```
                tileObject.Reset();   // 初始化
            }
        }
    }
```

在场景中选择 Grid Object 物体（TileObject 实例），然后在菜单栏中选择【Tools】→【Tile Window】，即可打开自定义的窗口，如图 4-4 所示，这里只是演示了如何显示一些基本的 UI。

图 4-4　自定义窗口界面

4.3　游戏场景

如图 4-5 所示，本示例的场景地面由 Sprite 拼凑而成，注意 Sprite 的 x 轴被旋转了 90°，Sprite 刚好与 3D 视图中的 z 轴平行。使用 Sprite 制作的地面不能接收光照和投影，读者可以按自己的兴趣随意搭建游戏场景。

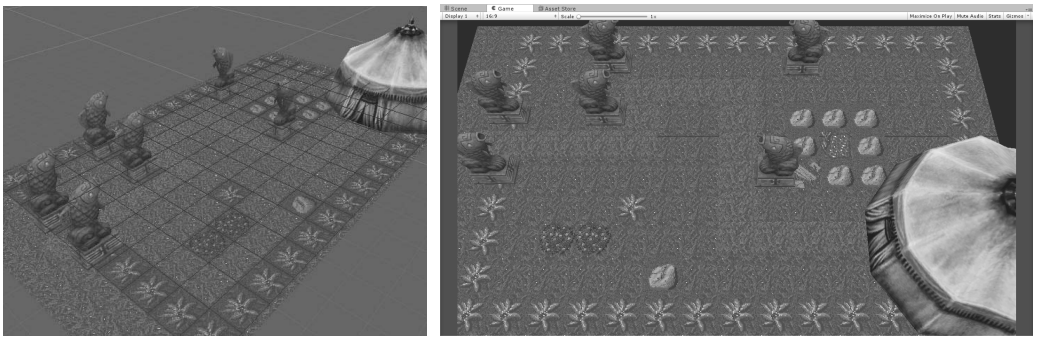

图 4-5　由 Sprite 和 3D 模型组成的示例游戏场景

本示例场景使用的部分美术资源来自 Asset Store 的免费资源 Backyard – Free，如图 4-6 所示，注意将 Pixels Per Unit 的大小设置与图片原始像素大小一致，即可使每个 Sprite 的大小与 Unity 单元格的大小一致。

本章后面还会使用到其他美术资源，它们都被保存在本书的资源文件目录 rawdata/td 内。

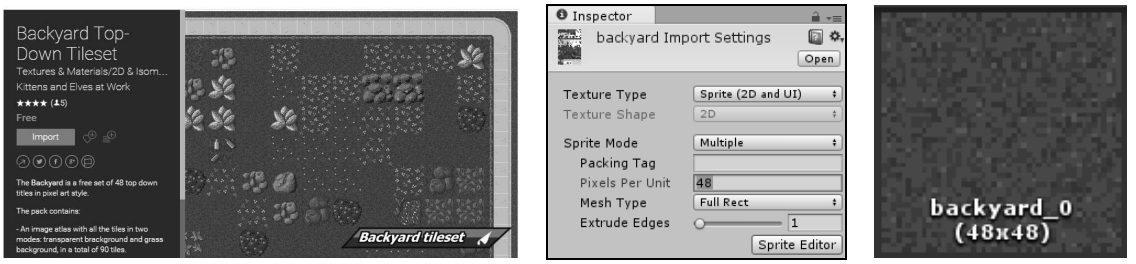

图 4-6 美术资源

4.4　制作 UI

首先创建塔防游戏的 UI 界面。

步骤 01　在本书资源文件目录 rawdata/td/GUI/中存放了所有的 UI 图片，导入图片后，注意将 Texture Type 设为 Sprite 类型，如图 4-7 所示。Unity 的 UI 系统只能使用 Sprite 类型的图片。

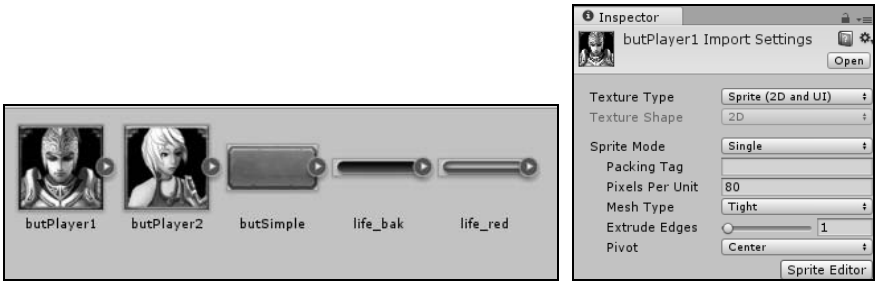

图 4-7　创建 Sprite

步骤 02　创建几个 UI 文字控件。在 Hierarchy 窗口中单击鼠标右键，选择【UI】→【Text】创建文字，在创建文字物体的同时，还会自动创建 Canvas 和 EventSystem 物体。Canvas 会自动作为文字控件的父物体，所有的 UI 控件都需要放到 Canvas 层级下。EventSystem 物体上有很多 UI 事件脚本，用来管理和响应 UI 事件，如图 4-8 所示。

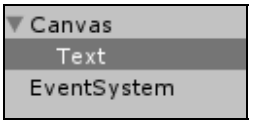

图 4-8　创建基础的 UI 控件

步骤 03　设置文字的位置。在编辑器的上方单击 UI 编辑按钮，然后选择文字控件即可改变文字的位置和尺寸，如图 4-9 所示。

步骤 04　在不同的分辨率下对齐 UI 的位置一直是件很麻烦的事情，不过使用 Unity 的新 UI 系统，一切将变得非常简单。选择前面创建的文本 UI 控件，在【Inspector】窗口的 Rect Transform 中可以快速设置控件的对齐方式，如左对齐、右对齐等，如图 4-10 所示。

步骤 05　除了对齐，我们还需要根据不同的屏幕分辨率对 UI 控件进行缩放。将 Canvas 物体的 Canvas Scaler 设为 Scale With Screen Size 模式，UI 控件将以设置的分辨率为基础，在不同的分辨率下进行缩放适配，如图 4-11 所示。

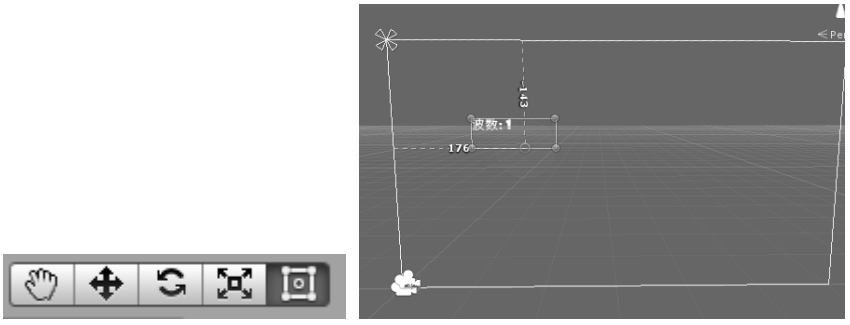

图 4-9　移动控件

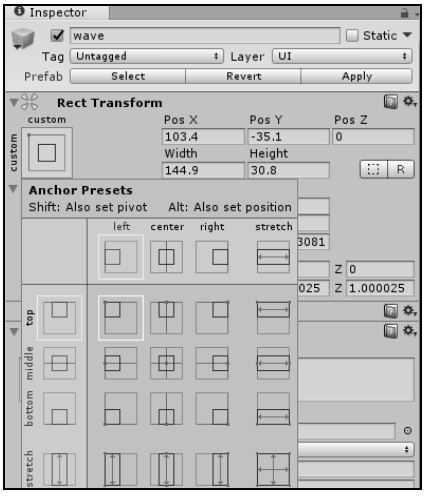

图 4-10　对齐控件

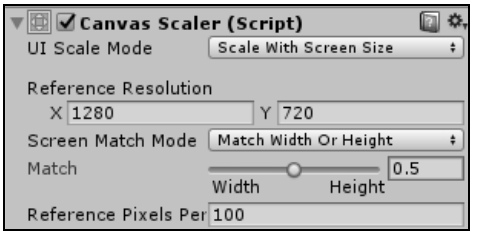

图 4-11　适配分辨率

步骤 06　这里一共需要创建三个不同的文字控件，分别用来显示敌人进攻的波数、铜钱和生命值。我们在【Inspector】窗口可以设置文字的内容、字体、大小、颜色等，使用 Unity 的 Rich Text 功能，在文本中添加 color 标记，可以使同一个文字有不同的色彩，如图 4-12 所示。

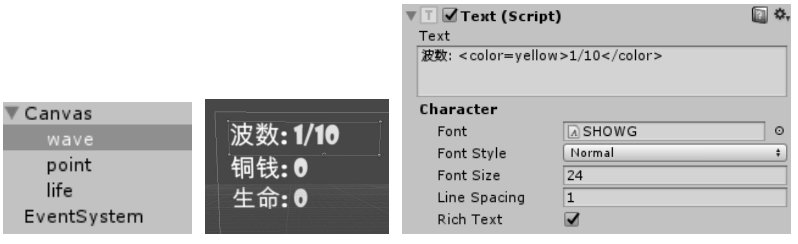

图 4-12　设置文本

步骤 07　现在很多游戏都给文字配上了描边，我们也加一个吧。选择文字，在菜单栏中选择【Component】→【UI】→【Effects】→【Outline】，如图 4-13 所示。

图 4-13　描边字

步骤 08 创建按钮，包括创建防守单位的按钮和重新游戏的按钮。在 Hierearchy 窗口中单击鼠标右键，选择【UI】→【Button】即可创建一个新的按钮控件，默认按钮下面还有一个文字控件用来显示按钮的名称。在【Inspector】窗口中找到 Image 组件下的 Source Image 设置按钮的图片，选择下面的 Set Native Size 可以使按钮的大小与图片的尺寸快速适配，如图 4-14 所示。

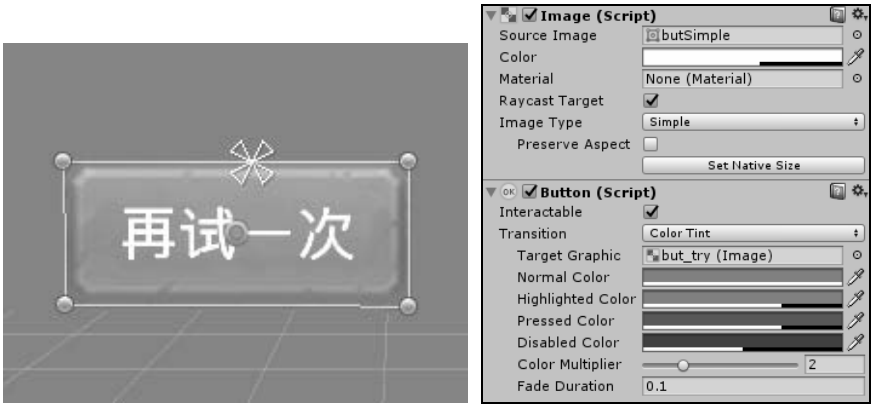

图 4-14　创建按钮

最后的 UI 效果如图 4-15 所示。注意 UI 控件的名字，我们在后面需要通过名字来查找 UI 控件。UI 控件的层级关系也比较重要，因为控件均位于 Canvas 或其他层级之下，所以它们的位置只是相对于父物体的相对位置。如果通过脚本去修改控件的位置，通常是修改 transform.localPosition，而不是 transform.position。

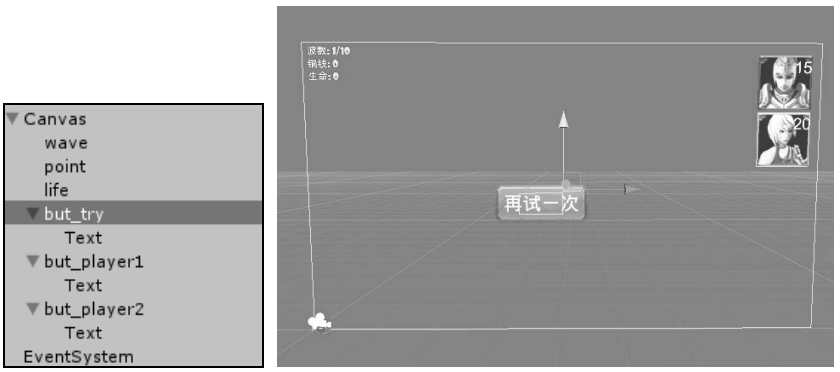

图 4-15　创建的 UI

完成 UI 设置后，可以将 UI 保存成 Prefab。

4.5　创建游戏管理器

在前一节，我们创建了 UI，但是没有功能，接下来我们将创建一个游戏管理器，它主要用来管理 UI，处理鼠标输入和逻辑事件等。

创建脚本 GameManager.cs，这里将其添加到 Canvas 物体上，代码如下所示：

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;                    // UI 控件命名空间的引用
using UnityEngine.Events;                // UI 事件命名空间的引用
using UnityEngine.EventSystems;          // UI 事件命名空间的引用

public class GameManager : MonoBehaviour {

public static GameManager Instance;          // 实例
    public LayerMask m_groundlayer;          // 地面的碰撞 Layer
    public int m_wave = 1;                   // 波数
    public int m_waveMax = 10;               // 最大波数
    public int m_life = 10;                  // 生命
    public int m_point = 30;                 // 铜钱数量

    // UI 文字控件
    Text m_txt_wave;
    Text m_txt_life;
    Text m_txt_point;
    // UI 重新游戏按钮控件
    Button m_but_try;

    // 当前是否选中的创建防守单位的按钮
    bool m_isSelectedButton =false;

    void Awake(){
        Instance = this;
    }
void Start () {
        // 创建 UnityAction，在 OnButCreateDefenderDown 函数中响应按钮按下事件
        UnityAction<BaseEventData> downAction =
                new UnityAction<BaseEventData>(OnButCreateDefenderDown);
        // 创建 UnityAction，在 OnButCreateDefenderDown 函数中响应按钮抬起事件
        UnityAction<BaseEventData> upAction =
                new UnityAction<BaseEventData>(OnButCreateDefenderUp);
        // 按钮按下事件
        EventTrigger.Entry down = new EventTrigger.Entry();
        down.eventID = EventTriggerType.PointerDown;
        down.callback.AddListener(downAction);

        // 按钮抬起事件
        EventTrigger.Entry up = new EventTrigger.Entry();
        up.eventID = EventTriggerType.PointerUp;
```

```
        up.callback.AddListener(upAction);

        // 查找所有子物体，根据名称获取 UI 控件
        foreach (Transform t in this.GetComponentsInChildren<Transform>()){
            if (t.name.CompareTo("wave") == 0)   // 找到文字控件“波数”
            {
                m_txt_wave = t.GetComponent<Text>();
                SetWave(1);   // 设为第 1 波
            }
            else if (t.name.CompareTo("life") == 0)   // 找到文字控件“生命”
            {
                m_txt_life = t.GetComponent<Text>();
                m_txt_life.text = string.Format("生命：<color=yellow>{0}</color>", m_life);
            }
            else if (t.name.CompareTo("point") == 0)   // 找到文字控件“铜钱”
            {
                m_txt_point = t.GetComponent<Text>();
                m_txt_point.text = string.Format("铜钱：<color=yellow>{0}</color>", m_point);
            }
            else if (t.name.CompareTo("but_try") == 0)   // 找到按钮控件“重新游戏”
            {
                m_but_try = t.GetComponent<Button>();
                // 重新游戏按钮
                m_but_try.onClick.AddListener( delegate(){
                    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
                });
                // 默认隐藏重新游戏按钮
                m_but_try.gameObject.SetActive(false);

            }
            else if (t.name.Contains("but_player"))      // 找到按钮控件“创建防守单位”
            {
                // 给防守单位按钮添加按钮事件
                EventTrigger trigger = t.gameObject.AddComponent<EventTrigger>();
                trigger.triggers = new List<EventTrigger.Entry>();
                trigger.triggers.Add(down);
                trigger.triggers.Add(up);
            }
        }
    }
    // 更新文字控件“波数”
    public void SetWave(int wave)
    {
        m_wave= wave;
```

```
            m_txt_wave.text = string.Format("波数:<color=yellow>{0}/{1}</color>", m_wave, m_waveMax);
        }
        // 更新文字控件“生命”
        public void SetDamage(int damage)
        {
            m_life -= damage;
            if (m_life <= 0) {
                m_life = 0;
                m_but_try.gameObject.SetActive(true);     // 显示重新游戏按钮
            }
            m_txt_life.text = string.Format("生命:<color=yellow>{0}</color>", m_life);
        }
        // 更新文字控件“铜钱”
        public bool SetPoint(int point)
        {
            if (m_point + point < 0)          // 如果铜钱数量不够
                return false;
            m_point += point;
            m_txt_point.text = string.Format("铜钱:<color=yellow>{0}</color>", m_point);
            return true;
        }
        // 按下“创建防守单位按钮”
        void OnButCreateDefenderDown(BaseEventData data)
        {
            m_isSelectedButton = true;
        }
        // 抬起“创建防守单位按钮”
        void OnButCreateDefenderUp( BaseEventData data )
        {
            GameObject go = data.selectedObject;
        // 此处代码将在后面步骤补充
        }
    }
```

在 Start 函数中，我们先定义了按钮的事件，然后通过查找所有子物体的名称找到相应的 UI 控件进行初始化处理，对 Text 文字控件，赋予初始的波数、生命和铜钱数值。

因为在游戏开始时我们不希望看到重新游戏按钮，所以调用 gameObject.SetActive(false) 方法将该按钮隐藏，当游戏结束时再显示该按钮重新游戏。

对于创建防守单位的按钮，我们分别定义了按下和抬起两个事件，当按下按钮的时候，获得要创建的对象，抬起按钮时在选定位置创建防守单位，不过当前这些按钮事件的回调函数是空的，还没做什么事情。

4.6 摄 像 机

因为游戏的场景可能会比较大，所以需要移动摄像机才能观察到场景的各个部分。接下来我们将为摄像机添加脚本，在移动鼠标的时候可以移动摄像机。

步骤 01 在为摄像机创建脚本前，首先创建一个空游戏体作为摄像机观察的目标点，并为其创建脚本 CameraPoint.cs，它只有很少的代码。注意，CameraPoint.tif 是一张图片，必须保存在工程中的 Gizmos 文件夹内。

```
using UnityEngine;
public class CameraPoint : MonoBehaviour
{
    public static CameraPoint Instance = null;
    void Awake(){
        Instance = this;
    }
    // 在编辑器中显示一个图标
    void OnDrawGizmos(){
        Gizmos.DrawIcon(transform.position, "CameraPoint.tif");
    }
}
```

步骤 02 创建脚本 GameCamera.cs，并将其指定给场景中的摄像机。

```
using UnityEngine;
public class GameCamera : MonoBehaviour {
    public static GameCamera Inst = null;
    // 摄像机距离地面的距离
    protected float m_distance = 15;
    // 摄像机的角度
    protected Vector3 m_rot = new Vector3(-55, 180, 0);
    // 摄像机的移动速度
    protected float m_moveSpeed = 60;
    // 摄像机的移动值
    protected float m_vx = 0;
    protected float m_vy = 0;
    // Transform 组件
    protected Transform m_transform;
    // 摄像机的焦点
    protected Transform m_cameraPoint;
    void Awake()
    {
```

```
            Inst = this;
            m_transform = this.transform;
        }
        void Start()
        {
            // 获得摄像机的焦点
            m_cameraPoint = CameraPoint.Instance.transform;
            Follow();
        }
        // 在 Update 之后执行
        void LateUpdate()
        {
            Follow();
        }
        // 摄像机对齐到焦点的位置和角度
        void Follow()
        {
            // 设置旋转角度
            m_cameraPoint.eulerAngles = m_rot;
            // 将摄像机移动到指定位置
            m_transform.position = m_cameraPoint.TransformPoint(new Vector3(0, 0, m_distance));
            // 将摄像机镜头对准目标点
            transform.LookAt(m_cameraPoint);
        }
        // 控制摄像机移动
        public void Control(bool mouse, float mx, float my)
        {
            if (!mouse)
                return;
            m_cameraPoint.eulerAngles = Vector3.zero;
            // 平移摄像机目标点
            m_cameraPoint.Translate(-mx, 0, -my);
        }
    }
```

在这个脚本的 Start 函数中，我们首先获得了前面创建的 CameraPoint，它将作为摄像机目标点的参考。

在 Follow 函数中，摄像机会按预设的旋转和距离始终跟随 CameraPoint 目标点。

LateUpdate 函数和 Update 函数的作用一样，不同的是它始终会在执行完 Update 后执行，我们在这个函数中调用 Follow 函数，确保在所有的操作完成后再移动摄像机。

Control 函数的作用是移动 CameraPoint 目标点，因为摄像机的角度和位置始终跟随这个目标点，所以也会随着目标点的移动而移动。

步骤 03　打开 GameManager.cs 脚本，在 Update 函数中添加代码如下：

```
void Update () {
        // 如果选中创建士兵的按钮，则取消摄像机操作
        if (m_isSelectedButton)
            return;
        // 鼠标或触屏操作，注意不同平台的 Input 代码不同
#if (UNITY_IOS || UNITY_ANDROID) && !UNITY_EDITOR
        bool press = Input.touches.Length > 0 ? true : false;   // 手指是否触屏
        float mx = 0;
        float my = 0;
        if (press)
        {
            if ( Input.GetTouch(0).phase == TouchPhase.Moved)   // 获得手指移动距离
            {
                mx = Input.GetTouch(0).deltaPosition.x * 0.01f;
                my = Input.GetTouch(0).deltaPosition.y * 0.01f;
            }
        }
#else
        bool press = Input.GetMouseButton(0);
        // 获得鼠标移动距离
        float mx = Input.GetAxis("Mouse X");
        float my = Input.GetAxis("Mouse Y");
#endif
        // 移动摄像机
        GameCamera.Inst.Control(press, mx, my);
    }
```

这段代码的作用是获取鼠标操作的各种信息并传递给摄像机，现在运行游戏，已经可以移动摄像机了。

4.7　路　　点

在前一章中，我们使用 Unity 提供的寻路功能实现敌人的行动，但在塔防游戏中，敌人通常不需要智能寻路，而是按照一条预设的路线行动。下面我们将为敌人创建一条前进路线，这条路线是预设的，敌人将从游戏场景的左侧沿着通道一直走到右侧。

步骤 01　敌人的前进路线是由若干个路点组成，首先添加路点的 Tag，这里名为 pathnode，为路点创建脚本 PathNode.cs：

```
using UnityEngine;
public class PathNode : MonoBehaviour {
    public PathNode m_parent; // 前一个节点
```

```
    public PathNode m_next; // 下一个节点
    // 设置下一个节点
    public void SetNext(PathNode node)
    {
        if (m_next != null)
            m_next.m_parent = null;
        m_next = node;
        node.m_parent = this;
    }
    // 在编辑器中显示的图标
    void OnDrawGizmos()
    {
        Gizmos.DrawIcon(this.transform.position, "Node.tif");
    }
}
```

在游戏中，敌人将从一个路点到达另一个路点，即到达当前路点的子路点。在 PathNode 脚本中，主要是通过 SetNext 函数设置它的子路点。

接下来，我们将创建路点并为每个路点设置子路点，为了设置方便，添加一个菜单功能，加速设置路点的操作。

步骤 02 在 Project 窗口中的 Assets 目录下创建一个名为 Editor 的文件夹，名称是特定的，不能改变，所有需要在编辑状态下执行的脚本都应当被存放到这里。在 Editor 文件夹内创建脚本 PathTool.cs，它将提供一个自定义的菜单，帮助我们设置路点，代码如下：

```
using UnityEngine;
using UnityEditor;
public class PathTool : ScriptableObject
{
    static PathNode m_parent=null;
    [MenuItem("PathTool/Create PathNode")]
    static void CreatePathNode()
    {
        // 创建一个新的路点
        GameObject go = new GameObject();
        go.AddComponent<PathNode>();
        go.name = "pathnode";
        // 设置 tag
        go.tag = "pathnode";
        // 使新创建的路点处于选择状态
        Selection.activeTransform = go.transform;
    }

    [MenuItem("PathTool/Set Parent %q")]
    static void SetParent()
```

```
        {
            if (!Selection.activeGameObject || Selection.GetTransforms(SelectionMode.Unfiltered).Length>1)
                return;
            if (Selection.activeGameObject.tag.CompareTo("pathnode") == 0){
                m_parent = Selection.activeGameObject.GetComponent<PathNode>();
            }
        }

        [MenuItem("PathTool/Set Next %w")]
        static void SetNextChild()
        {
            if (!Selection.activeGameObject ||
    m_parent==null || Selection.GetTransforms(SelectionMode.Unfiltered).Length>1)
                return;

            if (Selection.activeGameObject.tag.CompareTo("pathnode") == 0){
                m_parent.SetNext(Selection.activeGameObject.GetComponent<PathNode>());
                m_parent = null;
            }
        }
    }
```

这里的代码只有在编辑状态才能被执行，注意所有在这里使用的属性和函数均为 static 类型。Selection 是在编辑模式下的一个静态类，通过它可以获取到当前选择的物体。

[MenuItem("PathTool/Set Parent %q")]属性在菜单中添加名为 PathTool 的自定义菜单，并包括子菜单 Set Parent，快捷键为 Ctrl+Q。菜单 Set Parent 执行的功能即是 SetParent 函数的功能，将当前选中的节点作为父路点。

SetNextChild 函数将当前选中的路点作为父路点的子路点。

步骤 03　在菜单栏中选择【PathTool】→【Creat PathNode】创建路点。

步骤 04　复制若干个路点沿着道路摆放。按快捷键 Ctrl+Q 将其设为父路点，然后选择下一个路点，按 Ctrl+W 设为子路点，再按 Ctrl+Q 将它设为父路点，再选择子路点，反复这个操作，直到将所有路点设置完毕，效果如图 4-16 所示。注意，最后一个路点没有子路点。

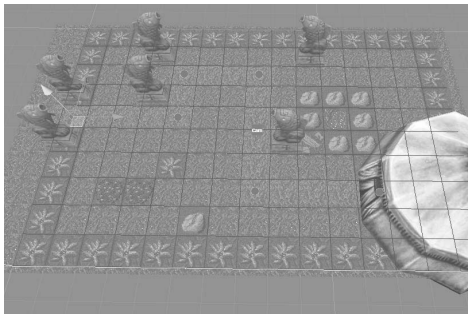

图 4-16　设置路点

虽然设置好了路点，但还是无法在场景中清楚地观察路点之间的联系，还需要在 GameManager.cs 中添加代码，使路点之间产生一条连线。

步骤 05 打开脚本 GameManager.cs，添加两个属性：m_debug 是一个开关，控制是否显示路点之间的连线；m_PathNodes 是一个 ArrayList，它用来保存所有的路点。

```
public bool m_debug = true; //  显示路点的 debug 开关
public List<PathNode> m_PathNodes; //  路点
```

步骤 06 继续在 GameManager.cs 中添加函数 BuildPath，并在 Start 函数中调用它，它的作用是将所有场景中的路点装入 m_PathNodes。

```
[ContextMenu("BuildPath")]
void BuildPath()
{
    m_PathNodes = new List<PathNode>();
    // 通过路点的 Tag 查找所有的路点
    GameObject[] objs = GameObject.FindGameObjectsWithTag("pathnode");
    for (int i = 0; i < objs.Length; i++)
    {
        m_PathNodes.Add( objs[i].GetComponent<PathNode>() );
    }
}
```

步骤 07 继续在 GameManager.cs 中添加函数 OnDrawGizmos，它的作用是当 m_debug 属性为真时，显示路点之间的连线。

```
void OnDrawGizmos(){
    if (!m_debug || m_PathNodes == null)
        return;
    Gizmos.color = Color.blue;   // 将路点连线的颜色设为蓝色
    foreach (PathNode node in m_PathNodes) // 遍历路点
    {
        if (node.m_next != null)
        {    // 在路点之间画出连接线
            Gizmos.DrawLine(node.transform.position, node.m_next.transform.position);
        }
    }
}
```

步骤 08 选择 UI Root 上的 Game Manager 脚本组件，设置 m_debug 属性为真，单击右上方的齿轮按钮，在弹出子菜单中选择【BuildPath】，这是我们自定义的菜单，如图 4-17 所示。

步骤 09 选择【BuildPath】后，将在场景中看到路点之间的连线，如图 4-18 所示。

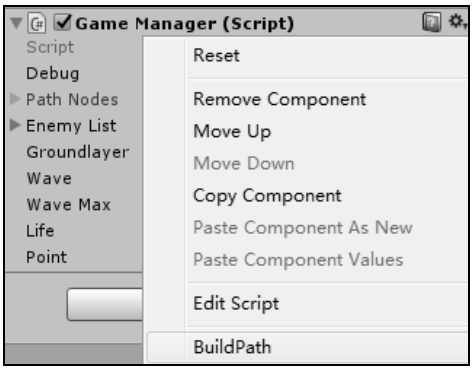

图 4-17　自定义的 BuildPath 选项

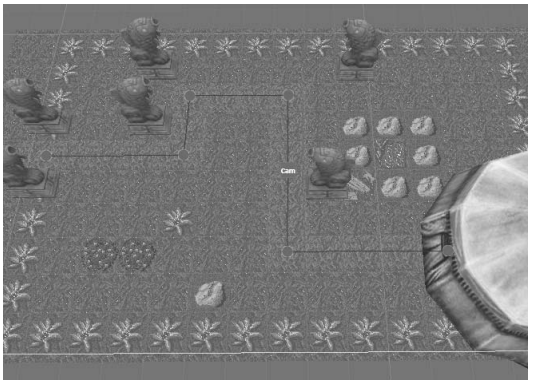

图 4-18　路点之间的连线

4.8　敌　　人

敌人一共有两种：一种在陆地上前进；另一种则会飞行。我们先创建前一种，然后继承它的大部分属性和函数，略加修改完成另一种。

步骤 01　创建敌人的脚本 Enemy.cs：

```
using UnityEngine;
public class Enemy : MonoBehaviour {
    public PathNode m_currentNode;          // 敌人的当前路点
    public int m_life = 15;                 // 敌人的生命
    public int m_maxlife = 15;              // 敌人的最大生命
    public float m_speed = 2;               // 敌人的移动速度
    public System.Action<Enemy> onDeath;    // 敌人的死亡事件
void Update () {
        RotateTo();
        MoveTo();
     }
// 转向目标
    public void RotateTo()
    {
        var position = m_currentNode.transform.position - transform.position;
        position.y = 0;                                          // 保证仅旋转 y 轴
        var targetRotation = Quaternion.LookRotation(position);  // 获得目标旋转角度
        float next = Mathf.MoveTowardsAngle(transform.eulerAngles.y, targetRotation.eulerAngles.y,
            120 * Time.deltaTime);                               // 获得中间旋转角度
        this.transform.eulerAngles = new Vector3(0, next, 0);    //旋转
    }
    // 向目标移动
```

```
public void MoveTo()
{
    Vector3 pos1 = this.transform.position;
    Vector3 pos2 = m_currentNode.transform.position;
    float dist = Vector2.Distance(new Vector2(pos1.x,pos1.z),new Vector2(pos2.x,pos2.z));
    if (dist < 1.0f)              // 如果到达路点的位置
    {
        if (m_currentNode.m_next == null)                    // 没有路点，说明已经到达我方基地
        {
            GameManager.Instance.SetDamage(1);               // 扣除一点伤害值
            DestroyMe();                                     // 销毁自身
        }
        else
            m_currentNode = m_currentNode.m_next;            // 更新到下一个路点
    }
    this.transform.Translate(new Vector3(0, 0, m_speed * Time.deltaTime));
}

public void DestroyMe()
{
    Destroy(this.gameObject);     // 注意在实际项目中一般不要直接调用 Destroy
}
```

在这个脚本中，定义了敌人的一些基本属性，如生命值、移动速度、类型等，它有一个路点属性作为出发点。

RotateTo 函数使敌人始终转向目标路点，MoveTo 函数则使其沿着当前方向前进，当距离目标路点较近时，将该路点作为当前路点，再向下一个路点前进。注意，这里计算敌人与子路点的距离时没有计算 y 轴。当敌人走到最后的路点，即是到达我方基地，销毁自身，并使基地减少一点生命值。

步骤 02 导入本书资源目录 rawdata/td/Rawdata 下的资源，找到 boar@skin.FBX 模型文件，拖入场景中。这是个野猪模型，它将作为陆地上的敌人。将 Enemy.cs 脚本指定给它，并设置起始路点，如图 4-19 所示。

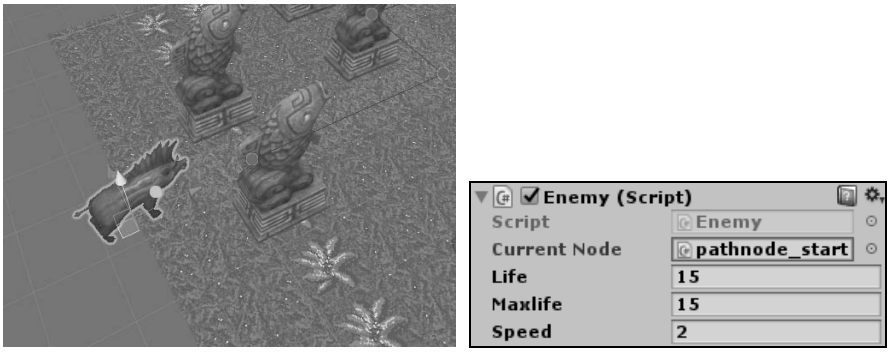

图 4-19　敌人组件

运行游戏，敌人会从起始点出发，沿着路点，一路前进到达我方基地，然后消失，我方基地将损失一点生命值。

这时我们会发现，野猪模型没有任何动画，看上去很生硬，需要为其添加动画效果。

步骤 03　带有动画的模型，被导入到 Unity 时会被自动设为 Generic。在 Project 窗口中单击鼠标右键选择【Create】→【Animator Controller】，为野猪模型创建一个动画控制器，如图 4-20 所示。前面放入场景中的野猪模型默认会带有一个 Animator 组件，将动画控制器指定给该组件。

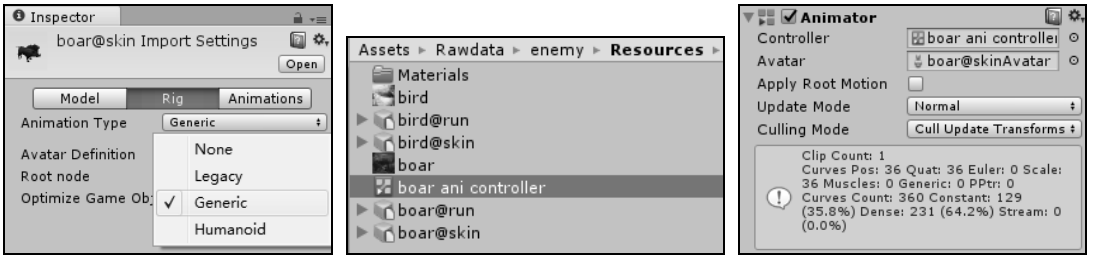

图 4-20　创建动画控制器

步骤 04　双击动画控制器打开 Animator 窗口，在 Project 窗口中选择 boar@run，在 Inspector 窗口中设置它的 Loop Time 使其循环播放，然后将动画拖入 Animator 窗口中，如图 4-21 所示。因为当前只有一个动画，所以它将作为默认动画自动播放。

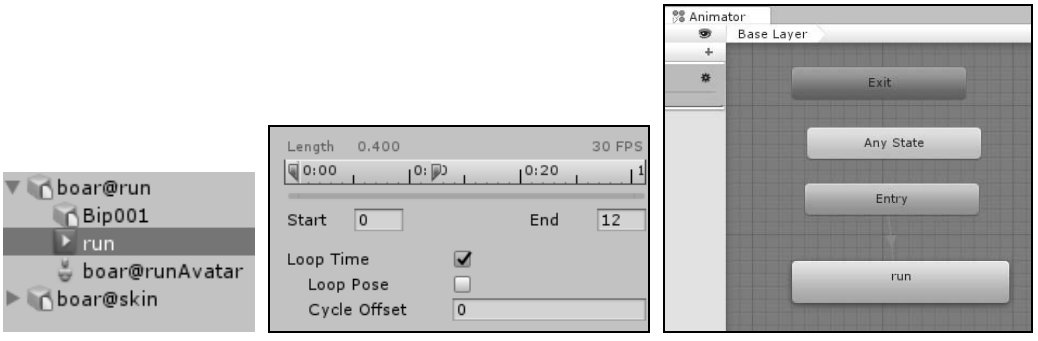

图 4-21　指定动画

再次运行游戏，即可看到模型在前进中播放了跑动的动画，将野猪模型保存为 Prefab。

步骤 05　接下来创建另一个飞行敌人的脚本 AirEnemy.cs，它继承了 Enemy 脚本的大部分功能，只添加一个 Fly 函数，作用是当高度小于 2 时向上飞行。

```
using UnityEngine;
public class AirEnemy : Enemy {
    void Update () {
        RotateTo();
        MoveTo();
        Fly();
    }
```

```
    public void Fly()
    {
        float flyspeed = 0;
        if (this.transform.position.y < 2.0f){
            flyspeed = 1.0f;
        }
        this.transform.Translate(new Vector3(0, flyspeed * Time.deltaTime,0));
    }
}
```

步骤 06 在资源文件中找到 bird@skin.FBX 模型，拖入场景中进行设置，步骤与野猪模型的设置一样，最后保存为 Prefab。运行游戏，效果如图 4-22 所示。

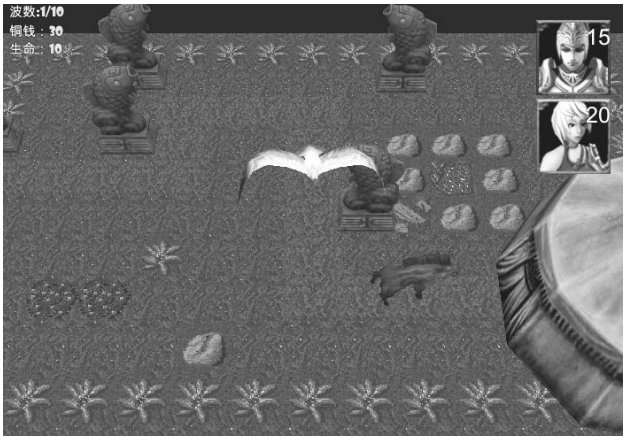

图 4-22　沿着路点前进的敌人

4.9　敌人生成器

塔防游戏的敌人通常是成批出现，一波接着一波，因为敌人的数量众多，所以需要一个生成器按预先设置的顺序生成不同的敌人。

4.9.1　创建敌人生成器

步骤 01 创建 WaveData.cs，它定义了战斗时每波敌人的配置。

```
using UnityEngine;
using System.Collections.Generic;
[System.Serializable] // 一定要添加该属性这个类才能序列化
public class WaveData {
    public int wave = 0;
    public List<GameObject> enemyPrefab;   // 每波敌人的 Prefab
    public int level = 1;   // 敌人的等级
```

```
    public float interval = 3; // 每 3 秒创建一个敌人
}
```

步骤 02 创建 EnemySpawner.cs，代码如下所示。因为是每隔一定时间根据配置生成敌人，这里使用协程完成了这个功能。

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class EnemySpawner : MonoBehaviour {
    public PathNode m_startNode;            // 起始节点
    private int m_liveEnemy = 0;            // 存活的敌人数量
    public List<WaveData> waves;            // 战斗波数配置数组
    int enemyIndex = 0;                     // 生成敌人数组的下标
    int waveIndex = 0;                      // 战斗波数数组的下标

    void Start () {
        StartCoroutine(SpawnEnemies());                     // 开始生成敌人
    }

    IEnumerator SpawnEnemies()
    {
        yield return new WaitForEndOfFrame();               // 保证在 Start 函数后执行
        GameManager.Instance.SetWave((waveIndex + 1));      // 设置 UI 上的波数显示

        WaveData wave = waves[waveIndex];                   // 获得当前波的配置
        yield return new WaitForSeconds(wave.interval);     // 生成敌人时间间隔
        while (enemyIndex < wave.enemyPrefab.Count)         // 如果没有生成全部敌人
        {
            Vector3 dir = m_startNode.transform.position - this.transform.position;          // 初始方向
            GameObject enmeyObj = (GameObject)Instantiate(wave.enemyPrefab[enemyIndex],
                                  transform.position, Quaternion.LookRotation(dir));   // 创建敌人
            Enemy enemy = enmeyObj.GetComponent<Enemy>();           // 获得敌人的脚本
            enemy.m_currentNode = m_startNode;                      // 设置敌人的第一个路点

            // 设置敌人数值，这里只是简单示范
            // 数值配置适合放到一个专用的数据库（SQLite 数据库或 JSON、XML 格式的配置）中读取
            enemy.m_life = wave.level * 3;
            enemy.m_maxlife = enemy.m_life;

            m_liveEnemy++;          // 增加敌人数量
            enemy.onDeath= new System.Action<Enemy>((Enemy e) =>{ m_liveEnemy--; });
                                                                    // 当敌人死掉时减少敌人数量
            enemyIndex++;                                           // 更新敌人数组下标
```

```
                yield return new WaitForSeconds(wave.interval);          // 生成敌人时间间隔
            }
            // 创建完全部敌人，等待敌人全部被消灭
            while(m_liveEnemy>0)
            {
                yield return 0;
            }

            enemyIndex = 0;                                  // 重置敌人数组下标
            waveIndex++;                                     // 更新战斗波数
            if (waveIndex< waves.Count)                      // 如果不是最后一波
            {
                StartCoroutine(SpawnEnemies());              // 继续生成后面的敌人
            }
            else
            {
                // 通知胜利
            }
        }
        // 在编辑器中显示一个图标
        void OnDrawGizmos()
        {
            Gizmos.DrawIcon(transform.position, "spawner.tif");
        }
    }
```

步骤 03 创建一个空游戏体作为敌人生成器放置到场景中，为其指定 EnemySpawner.cs 脚本。在 m_startNode 中设置起始路点，在 Waves 中配置敌人的生成，这里配置了 10 波，如图 4-23 所示。

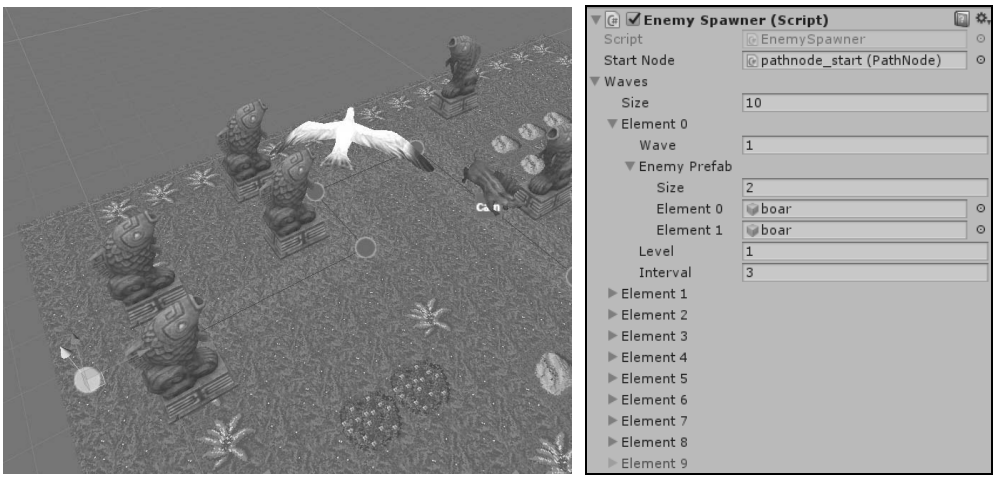

图 4-23　设置敌人 prefab 和起始路点

运行游戏，敌人会按照配置逐个生成。

4.9.2　遍历敌人

现在，游戏中有很多敌人，为了能方便地遍历游戏中的所有敌人，查看它们的情况，我们可以准备一个容器，将所有生成的敌人都装进去。

步骤 01　打开 GameManager.cs 脚本，添加一个 List 用来存放所有的敌人。

```
public List<Enemy> m_EnemyList = new List<Enemy>();
```

步骤 02　打开 Enemy.cs 脚本，在 Start 和 DestroyMe 函数中分别更新 List 中的敌人。添加一个 SetDamage 函数更新敌人生命值，当生命值为 0 时销毁自身并增加些铜钱。

```
 void Start () {
     GameManager.Instance.m_EnemyList.Add(this);
     // …
 }
public void DestroyMe()
{
     GameManager.Instance.m_EnemyList.Remove(this);
     onDeath(this);                    // 发布死亡消息
     Destroy(this.gameObject);     // 注意在实际项目中一般不要直接调用 Destroy
}
public void SetDamage(int damage)
{
     m_life -= damage;
     if (m_life <= 0)
     {
          m_life = 0;
          // 每消灭一个敌人增加一些铜钱
          GameManager.Instance.SetPoint(5);
          DestroyMe();
     }
}
```

现在所有的敌人都被保存到 List 中，当我们创建出防守单位后，他们可以通过遍历 List 中的敌人查找并攻击敌人，通过 SetDamage 函数更新敌人的生命值。

4.10　防守单位

本游戏中有两种防守单位：一种是近战类型；另一种是远程类型。我们先创建近战类型的防守单位，然后通过它派生出远程类型的防守单位。

步骤 01 使用资源文件目录 rawdata/td/Rawdata/players 中提供的模型和动画资源创建防守单位的 Prefab，将 Prefab 放到 Resources 文件夹内。这里主要是设置动画控制器并添加动画，如图 4-24 所示，包括 idle 和 attack 动画。

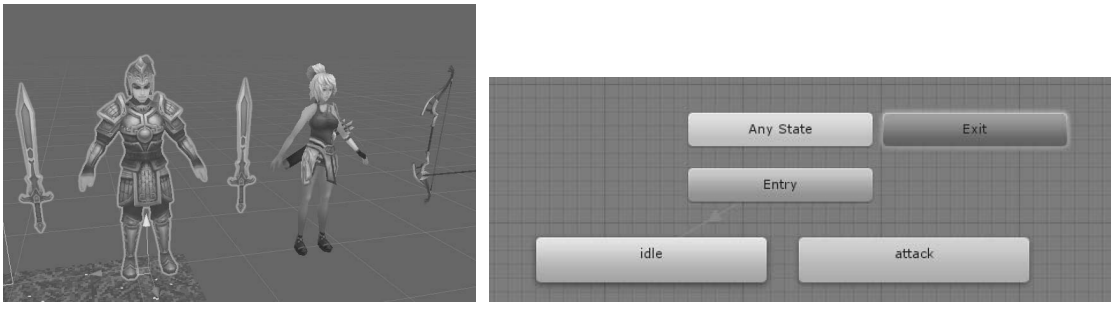

图 4-24　创建防守单位 Prefab

步骤 02 创建脚本 Defender.cs：

```
using UnityEngine;
using System.Collections;

public class Defender : MonoBehaviour {
    // 格子的状态
    public enum TileStatus
    {
        DEAD = 0,        // 不能在上面做任何事
        ROAD =1,         // 专用于敌人行走
        GUARD =2,        // 专用于创建防守单位的格子
    }

    // 攻击范围
    public float m_attackArea = 2.0f;
    // 攻击力
    public int m_power = 1;
    // 攻击时间间隔
    public float m_attackInterval = 2.0f;
    // 目标敌人
    protected Enemy m_targetEnemy;
    // 是否已经面向敌人
    protected bool m_isFaceEnemy;
    // 模型 Prefab
    protected GameObject m_model;
    // 动画播放器
    protected Animator m_ani;

    // 静态函数 创建防守单位实例
```

```
public static T Create<T>( Vector3 pos, Vector3 angle ) where T : Defender
{
    GameObject go = new GameObject("defender");
    go.transform.position = pos;
    go.transform.eulerAngles = angle;
    T d = go.AddComponent<T>();
    d.Init();

    // 将自己所占格子的信息设为占用
    TileObject.Instance.setDataFromPosition(d.transform.position.x, d.transform.position.z,
                                            (int)TileStatus.DEAD);
    return d;
}

// 初始化数值
protected virtual void Init()
{
    // 这里只是简单示范，在实际项目中，数值通常会从数据库或配置文件中读取
    m_attackArea = 2.0f;
    m_power = 2;
    m_attackInterval = 2.0f;
    // 创建模型，这里的资源名称是写死的，实际的项目通常会从配置中读取
    CreateModel("swordman");
    StartCoroutine(Attack()); // 执行攻击逻辑

}

// 创建模型
protected virtual void CreateModel(string myname)
{
    GameObject model = Resources.Load<GameObject>(myname);
    m_model = (GameObject)Instantiate(model, this.transform.position,
                                                this.transform.rotation, this.transform);
    m_ani = m_model.GetComponent<Animator>();
}

 void Update () {
    FindEnemy();
    RotateTo();
    Attack();
 }

public void RotateTo()
```

```
{
    if (m_targetEnemy == null)
        return;

    var targetdir = m_targetEnemy.transform.position - transform.position;
    targetdir.y = 0;            // 保证仅旋转 y 轴
    // 获取旋转方向
    Vector3 rot_delta = Vector3.RotateTowards(this.transform.forward, targetdir,
                        20.0f * Time.deltaTime, 0.0F);
    Quaternion targetrotation = Quaternion.LookRotation(rot_delta);

    // 计算当前方向与目标之间的角度
    float angle = Vector3.Angle(targetdir, transform.forward);
    // 如果已经面向敌人
    if (angle < 1.0f)
    {
        m_isFaceEnemy = true;
    }
    else
        m_isFaceEnemy = false;

    transform.rotation = targetrotation;
}

// 查找目标敌人
void FindEnemy()
{
    if (m_targetEnemy != null)
        return;
    m_targetEnemy = null;
    int minlife = 0;            // 最低的生命值
    foreach (Enemy enemy in GameManager.Instance.m_EnemyList)   // 遍历敌人
    {
        if (enemy.m_life == 0)
            continue;
        Vector3 pos1 = this.transform.position; pos1.y = 0;
        Vector3 pos2 = enemy.transform.position; pos2.y = 0;
        // 计算与敌人的距离
        float dist = Vector3.Distance(pos1, pos2);
        // 如果距离超过攻击范围
        if (dist > m_attackArea)
            continue;
        // 查找生命值最低的敌人
```

```
                if (minlife == 0 || minlife > enemy.m_life)
                {
                    m_targetEnemy = enemy;
                    minlife = enemy.m_life;
                }
            }
        }
        // 攻击逻辑
        protected virtual IEnumerator Attack()
        {
            while (m_targetEnemy == null || !m_isFaceEnemy)          // 如果没有目标一直等待
                yield return 0;
            m_ani.CrossFade("attack", 0.1f);                         // 播放攻击动画

            while (!m_ani.GetCurrentAnimatorStateInfo(0).IsName("attack"))      // 等待进入攻击动画
                yield return 0;
            float ani_lenght = m_ani.GetCurrentAnimatorStateInfo(0).length;     // 获得攻击动画长度
            yield return new WaitForSeconds(ani_lenght * 0.5f);                 // 等待完成攻击动作
            if (m_targetEnemy != null)
                m_targetEnemy.SetDamage(m_power);                    // 攻击
            yield return new WaitForSeconds(ani_lenght * 0.5f);      // 等待播放剩余的攻击动画
            m_ani.CrossFade("idle", 0.1f); // 播放待机动画

            yield return new WaitForSeconds(m_attackInterval);       // 间隔一定时间

            StartCoroutine(Attack()); // 下一轮攻击
        }
    }
```

① Create 函数是一个静态函数，我们可以使用它直接在代码中创建防守单位的游戏体。

② TileStatus 是一个枚举，用来表示场景中格子的状态，在 Create 函数中创建防守单位时，我们会更新格子的状态，使原本空闲的格子变为占有状态，这样便不能在这个格子中创建新的防守单位。

③ Init 函数是一个初始化函数，初始化了一些数值。注意，这是一个虚函数，当我们派生出其他类后，可以重载 Init 函数，赋予不一样的数值。在实际的项目中，我们也可以将所有的防守单位数值记录到配置文件中，创建不同的角色，读入不同的数值即可。

④ CreateModel 函数创建了所有的模型和动画。我们使用 Resources.Load 函数读入了模型和动画资源。注意，在实际项目中不建议在 Update 中直接调用 Resources.Load。

⑤ FindEnemy 函数会遍历所有的敌人，找出处于攻击范围内的敌人，并选择生命值最低的。

⑥ Attack 函数是一个协程函数，实现了攻击的逻辑。

步骤 03　创建远程防守单位。我们需要它的 Prefab 添加一个空物体作为“攻击点”，也就是发射弓箭的位置，如图 4-25 所示，这里“攻击点”的命名为 atkpoint。

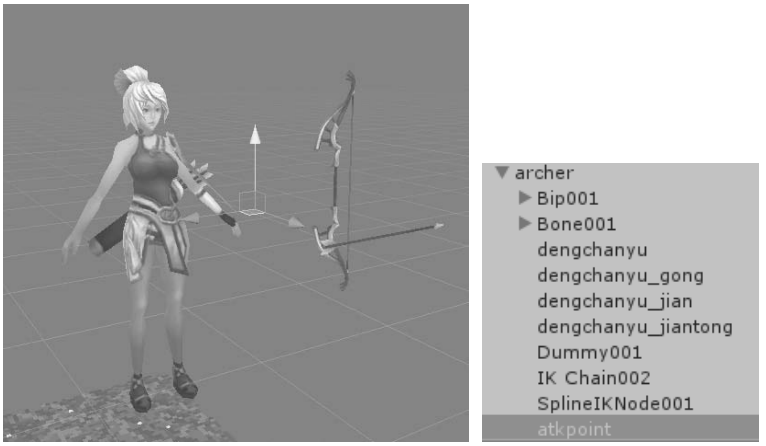

图 4-25　设置攻击点

步骤 04　使用资源文件目录 rawdata/td/Rawdata/players，创建一个新的脚本 Archer.cs，主要是在攻击时创建了弓箭模型。

```
using UnityEngine;
using System.Collections;

// 远程防守单位
public class Archer : Defender {

    // 初始化数值
    protected override void Init()
    {
        // 这里只是简单示范，在实际项目中，数值通常会从数据库或配置文件中读取
        m_attackArea = 5.0f;
        m_power = 1;
        m_attackInterval = 1.0f;
        // 创建模型，这里的资源名称是写死的，实际的项目通常会从配置中读取
        CreateModel("archer");
        // 获得模型的边框
        StartCoroutine(Attack());   // 执行攻击逻辑
    }

    // 攻击逻辑
    protected override IEnumerator Attack()
    {
        while (m_targetEnemy == null || !m_isFaceEnemy)   // 如果没有目标一直等待
            yield return 0;
        m_ani.CrossFade("attack", 0.1f);                  // 播放攻击动画

        while (!m_ani.GetCurrentAnimatorStateInfo(0).IsName("attack"))      // 等待进入攻击动画
```

```
            yield return 0;
        float ani_lenght = m_ani.GetCurrentAnimatorStateInfo(0).length;      // 获得攻击动画长度
        yield return new WaitForSeconds(ani_lenght * 0.5f);                  // 等待完成攻击动作
        if (m_targetEnemy != null)                                           // 向敌人发射弓箭
        {
            // 查找攻击点位置
            Vector3 pos = this.m_model.transform.FindChild("atkpoint").position;

            // 创建弓箭
            Projectile.Create(m_targetEnemy.transform, pos, (Enemy enemy) =>
            {
                enemy.SetDamage(m_power);
                m_targetEnemy = null;
            });
        }
        yield return new WaitForSeconds(ani_lenght * 0.5f);        // 等待播放剩余的攻击动画
        m_ani.CrossFade("idle", 0.1f);                             // 播放待机动画

        yield return new WaitForSeconds(m_attackInterval);         // 间隔一定时间

        StartCoroutine(Attack());                                  // 下一轮攻击
    }
}
```

Archer 类继承自 Defender 类，重载了 Init、Attack 函数，赋予不同的数值并做出不同的攻击行为。因为是该类表现为远程攻击，所以在攻击的时候创建了一个弓箭实例，弓箭的脚本我们将在下一步创建。

步骤 05　导入本书资源文件目录 rawdata/td/Fx_effect.unitypackage，它包括弓箭的 Prefab，创建一个新的脚本 Projectile.cs：

```
using UnityEngine;

public class Projectile : MonoBehaviour
{
    // 当打击到目标时执行的动作
    System.Action<Enemy> onAttack;
    // 目标对象
    Transform m_target;
    // 目标对象模型的边框
    Bounds m_targetCenter;
    // 使用静态函数创建弓箭
    public static void Create(Transform target, Vector3 spawnPos, System.Action<Enemy> onAttack)
    {
        // 读取弓箭模型
```

```
            GameObject prefab = Resources.Load<GameObject>("arrow");
            GameObject go = (GameObject)Instantiate(prefab, spawnPos,
                                Quaternion.LookRotation(target.position - spawnPos));
            // 添加弓箭脚本组件
            Projectile arrowmodel = go.AddComponent<Projectile>();
            // 设置弓箭的目标
            arrowmodel.m_target = target;
            // 获得目标模型的边框
            arrowmodel.m_targetCenter = target.GetComponentInChildren<SkinnedMeshRenderer>().bounds;
            // 取得 Action
            arrowmodel.onAttack = onAttack;
            // 3 秒之后自动销毁
            Destroy(go, 3.0f);
        }
        void Update()
        {
            // 瞄准目标中心位置
            if (m_target != null)
                this.transform.LookAt(m_targetCenter.center);

            // 向目标前进
            this.transform.Translate(new Vector3(0, 0, 10 * Time.deltaTime));
            if (m_target != null)
            {
                // 简单通过距离检测是否打击到目标
                if (Vector3.Distance(this.transform.position, m_targetCenter.center) < 0.5f)
                {
                    // 通知弓箭发射者
                    onAttack(m_target.GetComponent<Enemy>());
                    // 销毁
                    Destroy(this.gameObject);
                }
            }
        }
    }
```

弓箭类的功能很简单，创建一个弓箭模型，然后朝目标点前进，当距离目标小于 0.5 个单位时，触发 Action 通知弓箭发射者已经打击到目标，然后销毁自己。再次提醒，在实际项目中不要在 Update 中直接调用 Resources.Load、Instantiate 和 Destroy。

最后，我们需要修改 GameManager.cs 脚本，在按钮事件中添加创建防守单位的代码。

步骤 06 打开 GameManager.cs 脚本，修改 OnButCreateDefenderUp 函数，在抬起鼠标时用射线测试是否与格子地面碰撞，如果符合条件，则创建防守单位，代码如下：

```
// 抬起 "创建防守单位按钮" 创建防守单位
    void OnButCreateDefenderUp( BaseEventData data )
    {
        // 创建射线
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hitinfo;
        // 检测是否与地面相碰撞
        if (Physics.Raycast(ray, out hitinfo, 1000, m_groundlayer))
        {
            // 如果选中的是一个可用的格子
            if (TileObject.Instance.getDataFromPosition(hitinfo.point.x, hitinfo.point.z) ==
                                                                    (int)Defender.TileStatus.GUARD)
            {
                // 获得碰撞点位置
                Vector3 hitpos = new Vector3(hitinfo.point.x, 0, hitinfo.point.z);
                // 获得 Grid Object 坐位位置
                Vector3 gridPos = TileObject.Instance.transform.position;
                // 获得格子大小
                float tilesize = TileObject.Instance.tileSize;
                // 计算出所点击格子的中心位置
                hitpos.x = gridPos.x + (int)((hitpos.x - gridPos.x) / tilesize) * tilesize + tilesize * 0.5f;
                hitpos.z = gridPos.z + (int)((hitpos.z - gridPos.z) / tilesize) * tilesize + tilesize * 0.5f;

                // 获得选择的按钮 GameObject，将简单通过按钮名字判断选择了哪个按钮
                GameObject go = data.selectedObject;
                if (go.name.Contains("1"))   //如果按钮名字包括“1”
                {
                    if (SetPoint(-15)) // 减 15 个铜钱，然后创建近战防守单位
                        Defender.Create<Defender>(hitpos, new Vector3(0, 180, 0));
                }
                else if (go.name.Contains("2"))// 如果按钮名字包括“2”
                {
                    if (SetPoint(-20)) // 减 20 个铜钱，然后创建远程防守单位
                        Defender.Create<Archer>(hitpos, new Vector3(0, 180, 0));
                }
            }
        }
        m_isSelectedButton = false;
    }
```

运行游戏，单击右侧的按钮，然后拖动鼠标到场景中并释放，即可创建一个相应的防守单位，如图 4-26 所示。

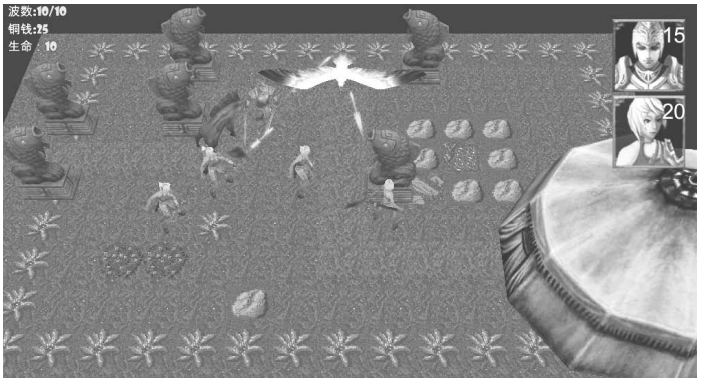

图 4-26　防守单位

4.11　生　命　条

敌人在受到攻击的时候，我们并不知道它受到了多少伤害，为了能够显示它的剩余生命值，我们需要为它制作一个生命条，显示在敌人身体的上方。因为这是一款 3D 游戏，所以我们将使用 3D UI 功能创建这个生命条。

步骤 01　在 Hierarchy 窗口中单击鼠标右键，选择【UI】→【Canvas】，在场景中创建一个新的 Canvas，并命名为 Canvas3D，将【Render Mode】设为【World Space】，使这个 UI 成为一个 3D UI，如图 4-27 所示。

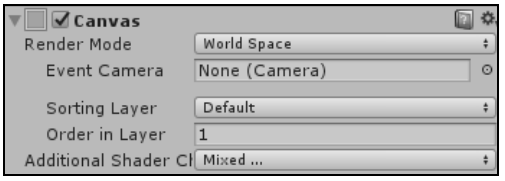

图 4-27　设置为 3D UI

步骤 02　在 Hierarchy 窗口中单击鼠标右键，选择【UI】→【Slider】创建一个滑动条控件，在 Source Image 中指定生命条的背景，因为生命条并不需要滑块，所以将 Handle Slider Area 隐藏或删除，如图 4-28 所示。

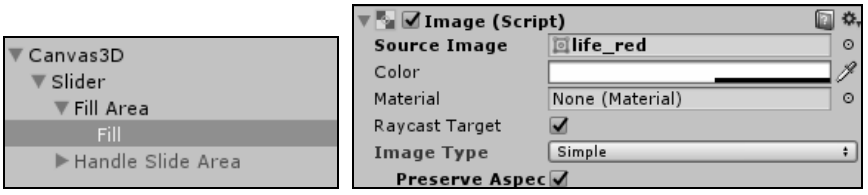

图 4-28　生命条背景图

步骤 03　默认 Silder 层级下的 Fill 即是生命条的前景，在 Source Image 中设置前景图片，【Image Type】选择【Filled】，【Fill Method】选择【Horizontal】，如图 4-29 所示。

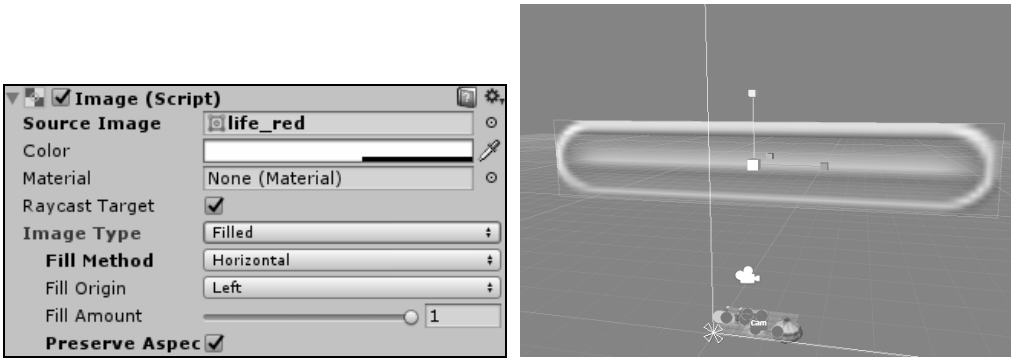

图 4-29　生命条设置

将 UI 保存为 Prefab 并放置在 Resource 目录下，命名为 Canvas3D，然后删除场景中的 3D UI。

步骤 04　打开 Enemy.cs 脚本，添加创建、更新生命条的代码如下：

```
    Transform m_lifebarObj;     // 敌人的 UI 生命条 GameObject
    UnityEngine.UI.Slider m_lifebar;     // 控制生命条显示的 Slider

void Start () {

        GameManager.Instance.m_EnemyList.Add(this);

        // 读取生命条 prefab
        GameObject prefab = (GameObject)Resources.Load("Canvas3D");
        // 创建生命条，将当前 Transform 设为父物体
        m_lifebarObj = ((GameObject)Instantiate(prefab, Vector3.zero,
                        Camera.main.transform.rotation, this.transform )).transform;
        m_lifebarObj.localPosition = new Vector3(0, 2.0f, 0);     // 将生命条放到角色头上
        m_lifebarObj.localScale = new Vector3(0.02f, 0.02f, 0.02f);
        m_lifebar = m_lifebarObj.GetComponentInChildren<UnityEngine.UI.Slider>();
        // 更新生命条位置和角度
        StartCoroutine(UpdateLifebar());
    }

    IEnumerator UpdateLifebar()
    {
        // 更新生命条的值
        m_lifebar.value = (float)m_life / (float)m_maxlife;
        // 更新角度，如终面向摄像机
        m_lifebarObj.transform.eulerAngles = Camera.main.transform.eulerAngles;
        yield return 0;   // 没有任何等待
        StartCoroutine(UpdateLifebar());     // 循环执行
    }
```

运行游戏，在敌人的上方会出现一个生命条，当敌人受到攻击且生命值下降时，生命条状态会改变，如图 4-30 所示。如果需要精确地放置生命条在角色头上的位置，可以在创建 3D 模型时专门创建一个节点用来参考生命条的位置。

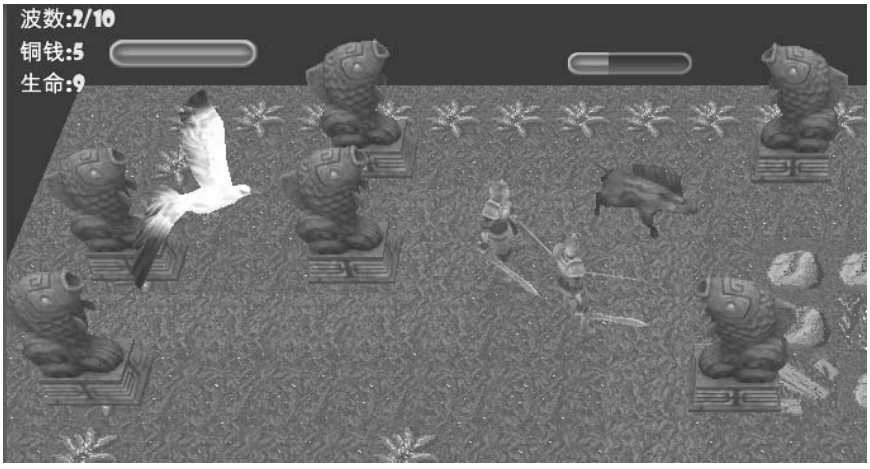

图 4-30　生命条

这个塔防游戏到这里就结束了，它还比较简单，但已具备了塔防游戏的基本要素，如果添加更多的细节和更好的画面，相信它可以变成一款不错的游戏。

本章的最终示例工程保存在资源文件目录 c04_TD 中。

4.12　小　　结

本章完成了一个塔防游戏的实例，我们使用数组定义场景中的单元格数据并制作了一个地图编辑器，创建敌人行动的路点，还涉及动画的播放、出生点的创建等。

第 5 章
2D游戏

本章将介绍如何使用 Unity 开发 2D 游戏，包括 Sprite 的创建、动画的制作、2D 物理的使用，以及一个 2D 捕鱼游戏的示例。

5.1 Unity 2D 系统简介

在 Unity 4.x 版本之前，使用 Unity 制作 2D 游戏通常要依赖插件。Unity 4.x 后，终于发布了内置的 2D 游戏制作功能，核心模块包括 Sprite（精灵）的创建、动画的制作和 2D 游戏专用的物理模块。现在，Unity 不但是一个 3D 的游戏引擎，也是一个专业的 2D 游戏引擎了，暴雪推出的 2D 卡牌游戏大作《Heart Stone》（炉石传说）就是用的 Unity。

5.2 创建 Sprite

2D 游戏的图像部分主要是图片的处理，通常称图片为 Sprite（精灵）。为了提高效率，2D 游戏会将动画帧或不同的小图片拼成一张大图，在游戏运行的时候，再将这张大图的某一部分读出来作为 Sprite 显示在屏幕上。

使用 Unity 制作 Sprite 有两种选择：一种是可以将不同的图片在图像软件中拼接为一张大图，然后导入到 Unity 中，使用 Sprite Editor 工具将这张大图分割成若干个 Sprite 使用；另一种是直接将原始图片导入到 Unity 中，使用 Unity 的 Sprite Packer 工具将这些零散的图片打包，再从中读取 Sprite 使用。接下来，将分别介绍这两种创建 Sprite 的方式及动画的制作。

5.2.1 使用 SpriteEditor 创建 Sprite

步骤 01 启动 Unity 编辑器，在菜单栏中选择【Edit】→【Project Settings】→【Editor】，打开编辑器设置窗口，将【Default Behavior Mode】设为【2D】，Unity 编辑器将会进入 2D 编辑模式，在这种模式下，导入到工程中的图片默认会被自动设为 Sprite 格式。将【Sprite Packer】设为【Always Enabled】，在任何情况下都能启动 Sprite Packer，如图 5-1 所示。

步骤 02 在资源文件目录“rawdata/2d”提供了示例图片 envtile.png，将它导入到 Unity 工程中。这是一张由若干个小图组成的图片，如图 5-2 所示。

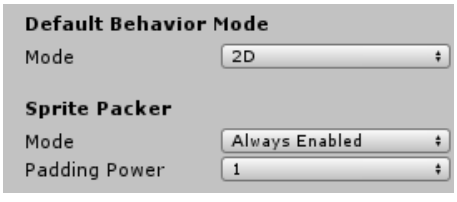

图 5-1 将编辑器设置为 2D 模式

图 5-2 导入的 2D 图片

步骤 03 在 Project 窗口中选择导入的图片，在 Inspector 窗口中将 Sprite Mode 设为 Multiple，这种模式可以将一张大图分割为若干个 Sprite，如图 5-3 所示。注意，Pixels Per Unit（单位像素）的值越大，Sprite 图像显示的默认尺寸越小。

步骤 04 在 Inspector 窗口中选择 Sprite Editor 打开编辑窗口，选择左上角的【Slice】，默认的切割方式为 Automatic（自动的），将它改为 Grid，然后在 Pixel Size 中输入每个 Sprite 的尺寸大小，本例为 32×32，默认 Sprite 的轴点心为 Center（中心），单击【Slice】按钮开始分割图片，如图 5-4 所示。

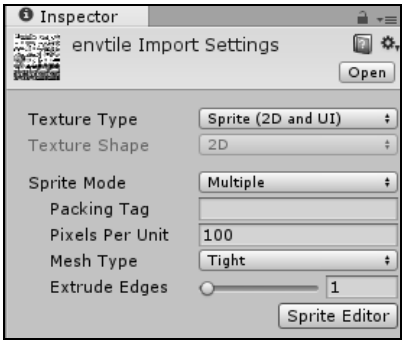

图 5-3 设置 Sprite

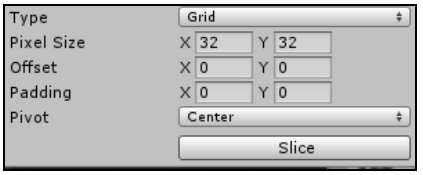

图 5-4 设置 Sprite 尺寸

步骤 05 分割好图片后，会发现有些 Sprite 的尺寸大于 32×32，可以用鼠标直接选择分割出来的 Sprite，并将无用的删除，重新设置需要放大尺寸的 Sprite，选择 Trim 可以清除无用的 Alpha，最后单击【Apply】按钮完成修改，如图 5-5 所示。

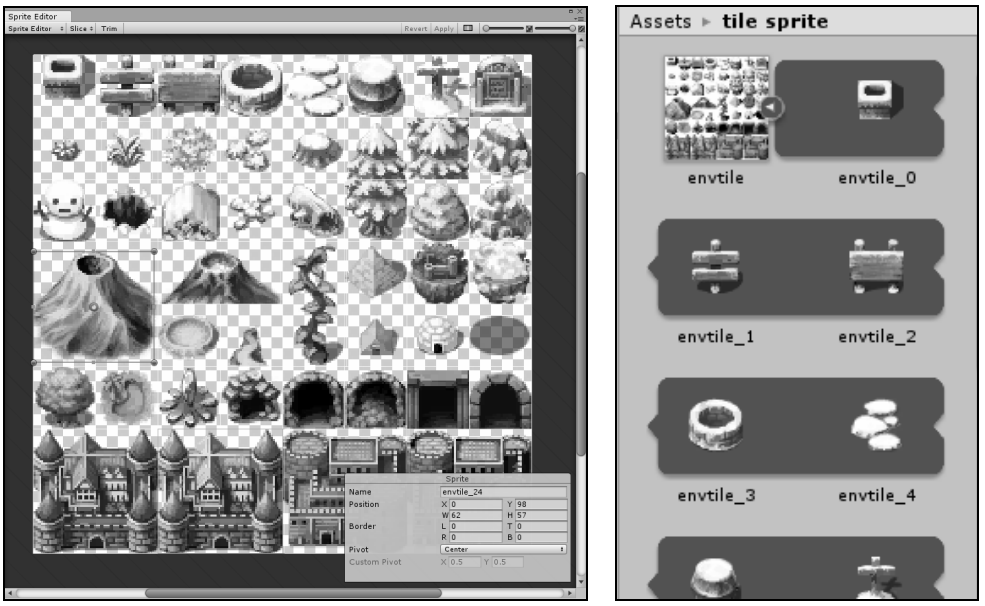

图 5-5 设置单个 Sprite 尺寸

步骤 06 在 Hierarchy 窗口中单击鼠标右键，选择【2D Object】→【Sprite】，在场景中创建一个空的 Sprite 实例，在 Sprite 选项中引用 Sprite 图像即可显示 2D 图像，如图 5-6 所示。

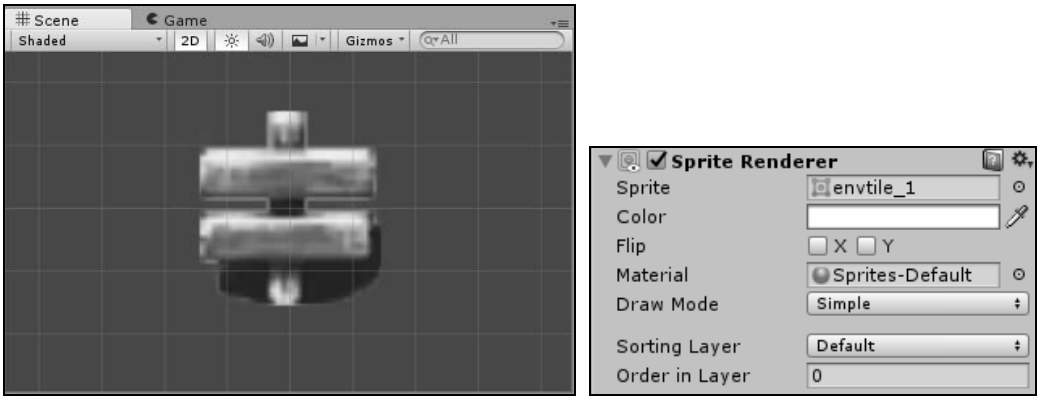

图 5-6　创建 Sprite 实例

步骤 07　在场景中复制多个 Sprite，并指定不同的贴图，示例效果如图 5-7 所示。

尽管我们在场景中创建了很多 Sprite 图像，但是如果在 Game 窗口中选择 Stats 查看运行信息，会发现 Batches（Draw Calls）很少，这是因为这些 Sprite 图像的原始图片是同一张图片，如图 5-8 所示。如果使用的 Sprite 图像都来自不同的图片素材，Batches 的数量会明显提升。

图 5-7　2D Sprite 示例

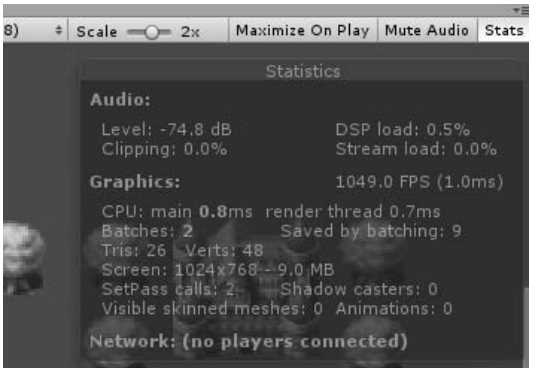

图 5-8　查看 Draw Calls

本节的示例工程文件保存在资源文件目录 c05_2dgame 的 tiled sprite 场景中。

5.2.2　使用 SpritePacker 创建 Sprite

手动将琐碎的图片拼凑到一张大图上需要做额外的工作，后期修改也不是很方便。Unity 提供的另一个 Sprite 工具 SpritePacker 可以动态地将零散的图片自动合成，省去了手动设置的麻烦。下面用一个简单的例子介绍这个工具的使用。

步骤 01　首先要确认在 Editor 设置中将 Sprite Packer 设为 Always Enabled。在资源文件目录 rawdata\2d 复制 player_n.png 等共 8 张图片到 Unity 工程目录内。

步骤 02　在 Project 窗口中右键选择【Create】→【Sprite Atlas】创建一个图集文件，如图 5-9 所示，将需要打包到图集中的图片拖拽到 Objects for Packing 中。

步骤 03　最后选择 Pack Preview 就可以预览打包后的效果了，如图 5-10 所示。

图 5-9　创建 Atlas

图 5-10　打包图片

前面我们在 Editor 中设置过 Sprite Packer 的模式，Always Enabled 允许在任何情况下使用 Sprite Packer，Enabled For Builds 只有在导出游戏时才会使用 Sprite Packer，Disabled 则取消使用 Sprite Packer 的功能，如图 5-11 所示。

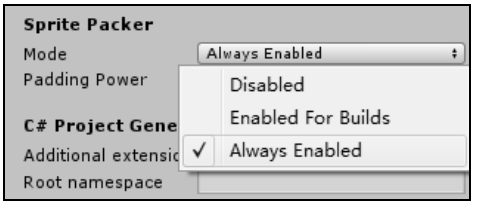

图 5-11　Sprite packer 模式

5.2.3　图层排序

2D 游戏中的图片之间没有深度关系，都处在一个平面上，但在实际游戏中仍需要为它们排序，按视觉上的空间先后顺序显示。在 Sprite 的 Sorting Layer 中可以为不同的 Sprite 创建不同的层，每个层有一个名称，层的顺序即是 Sprite 的显示顺序，值越大越显示在前面。在同一个层中的 Sprite 可以通过 Order in Layer 来排列显示顺序，如图 5-12 所示。

图 5-12　图层排序

Sorting Layer 和 Order in Layer 的值在脚本中都可以获取，通过脚本可以动态地改变图层的顺序。下面的代码改变了 Sprite 的层和排序顺序：

```
SpriteRenderer r = this.GetComponent<SpriteRenderer>();
r.sortingLayerName = "Layer Name";
r.sortingOrder = 100;
```

5.2.4 Sprite 边框和重复显示

默认的 Sprite 显示方式，一种是 Sprite 只能显示一张图像，如果需要显示大片重复的图像，就需要复制很多 Sprite。

Sprite 的另一种显示方式是 Tiled（单元格重复）模式，允许重复显示图像，如图 5-13 所示。使用这种方式，要将 Sprite 的 Mesh Type 设为 Full Rect。

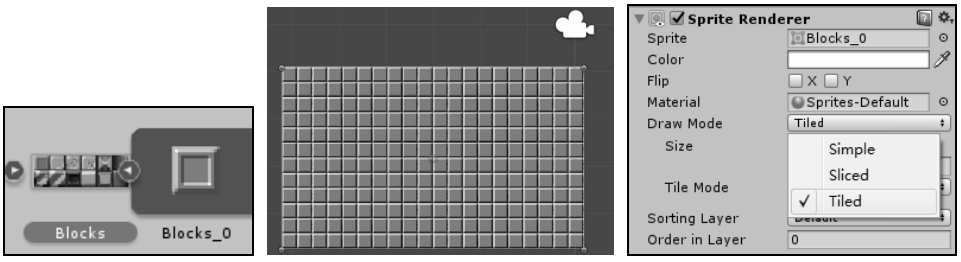

图 5-13　重复显示

在 Sprite Editor 编辑 Sprite 时，可以编辑边框区域（绿色的框），将带有边框的 Sprite 设为 Sliced（切片）或 Tiled 模式，改变 Sprite 图像大小时不会拉伸边框像素，如图 5-14 所示。

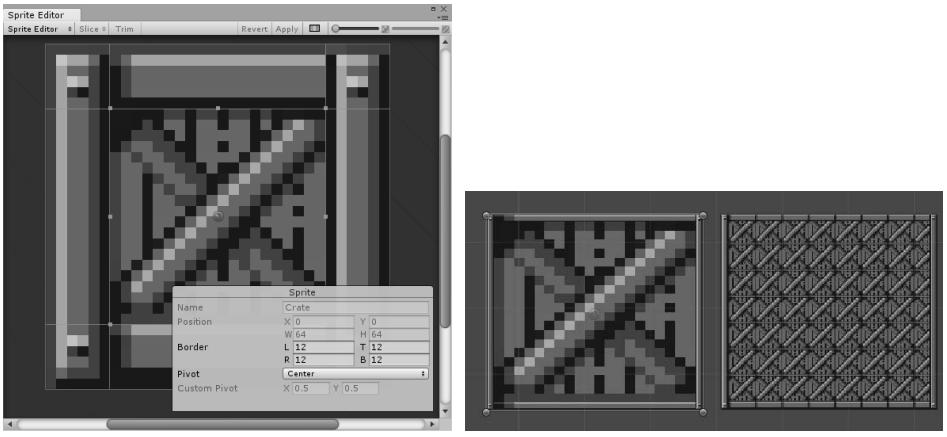

图 5-14　Sliced 和 Tiled 显示模式

5.3 动画制作

5.3.1 序列帧动画

传统的 2D 动画主要是由若干张表现有连续动作的图片组成，在 Unity 中制作这种类型的 2D 动画非常简单，可以将动画序列帧直接保存为动画文件，使用起来和普通的 3D 动画一样。

步骤 01 在 Project 窗口中按住 Shift 键选中需要的序列帧图片，将它们拖向 Hierarchy 窗口，这时 Unity 会自动弹出一个窗口保存动画文件，为这个动画文件命名，单击【保存】按钮，将动画文件保存在工程内，如图 5-15 所示。

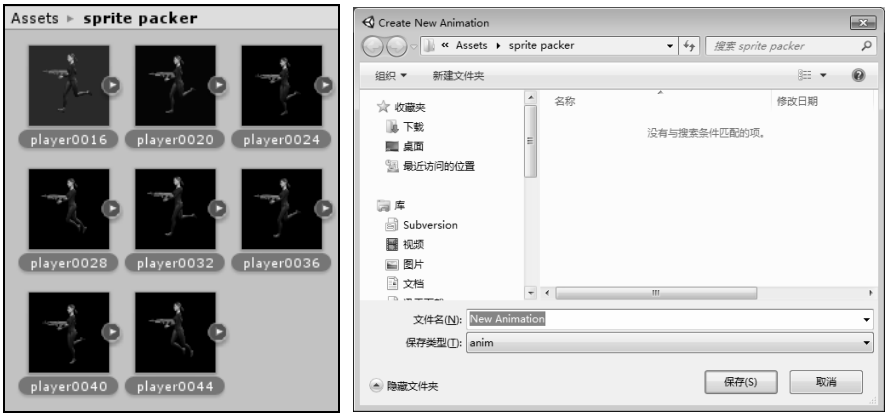

图 5-15　保存序列帧动画文件

步骤 02 上一步在创建动画的同时，也在场景中创建了一个 Sprite 文件。运行游戏，即可看到 2D 的序列帧动画效果，如图 5-16 所示。

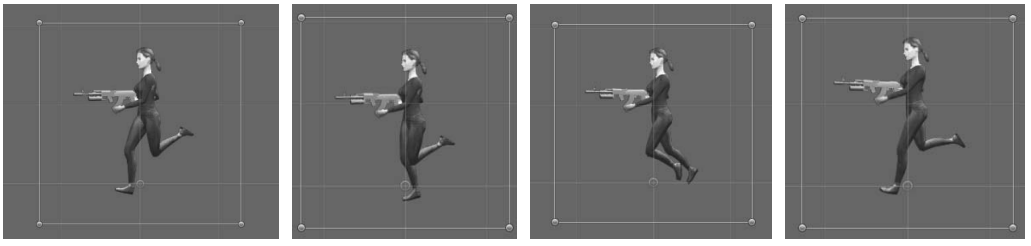

图 5-16　序列帧动画

步骤 03 保存动画后，在 Project 窗口中多出了两个文件：一个是动画控制器（Animator Controller）；另一个是动画文件。生成动画的时候，在场景中创建的 Sprite 会被自动添加一个 Animator 组件。双击动画控制器打开 Animator 窗口，可以像设置 3D 动画一样设置 2D 动画，如图 5-17 所示。

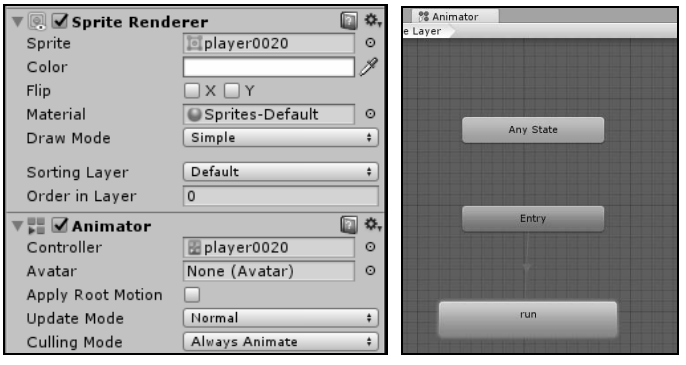

图 5-17　序列帧动画

5.3.2 使用脚本实现序列帧动画

因为 2D 的序列帧动画只是将图片一张张地按顺序显示出来，我们也可以使用脚本实现序列帧动画，下面是一个简单的示例。

步骤 01 在场景中创建一个 Sprite，为它创建一个脚本，这里命名为 SpriteAnimator.cs，添加代码如下

```
using UnityEngine;
public class AnimationTest : MonoBehaviour
{
    /// Sprite 渲染器
    protected SpriteRenderer m_sprite;
    /// Sprite 动画帧
    public Sprite[] m_clips;
    /// 动画计时器（默认每隔 0.1 秒更新一帧）
    protected float timer = 0.1f;
    /// 当前的帧数
    protected int m_frame = 0;

    void Start()
    {
        // 为当前 GameObject 添加一个 Sprite 渲染器
        m_sprite = this.gameObject.GetComponent<SpriteRenderer>();
        // 设置第 1 帧的 Sprite
        m_sprite.sprite = m_clips[m_frame];
    }

    void Update()
    {
        // 更新时间
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            timer = 0.1f;
            // 更新帧
            m_frame++;
            if (m_frame >= m_clips.Length)
                m_frame = 0;
            // 更新 Sprite 动画帧
            m_sprite.sprite = m_clips[m_frame];
        }
    }
}
```

步骤 02 在编辑器中设置用于动画的 Sprite，如图 5-18 所示。

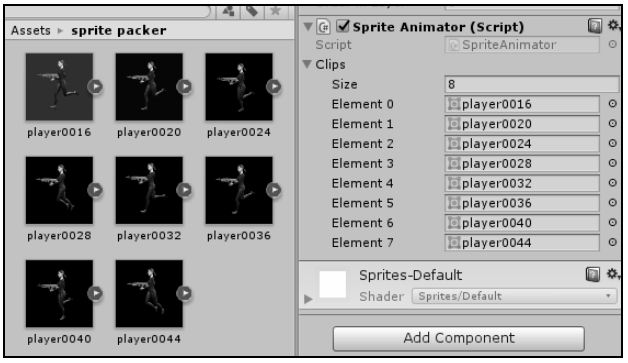

图 5-18　设置动画帧

运行程序，可以看到与 5.3.1 节类似的动画效果。

5.3.3　骨骼动画

如果需要表现细腻的序列帧动画效果，就需要很多图片，相当消耗资源，因此游戏中的序列帧动画往往是按一定程度跳帧的，但跳帧后动画流畅度会大幅下降。现在也有很多 2D 游戏，将 2D 的角色拆分为若干个 Sprite，用骨骼绑定起来，利用制作 3D 动画的方式制作 2D 动画，在更加节约资源的同时又加强了动画的流畅性，下面是一个简单的示例。

步骤 01 新建一个工程，复制资源文件目录 rawdata/2d/下的 warrior.png 到当前工程目录内，这是一个角色图片，角色的不同位置是分开的，如图 5-19 所示。

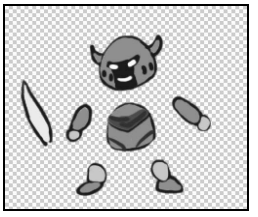

图 5-19　原始图片资源

步骤 02 将角色图片的 Sprite Mode 设为 Multiple，打开 Sprite Editor，选择左上角的【Slice】，选择【Automatic】模式，然后选择【Slice】自动切分图片，选择每个切分部分，移动上面的圆点设置轴心位置，如图 5-20 所示。

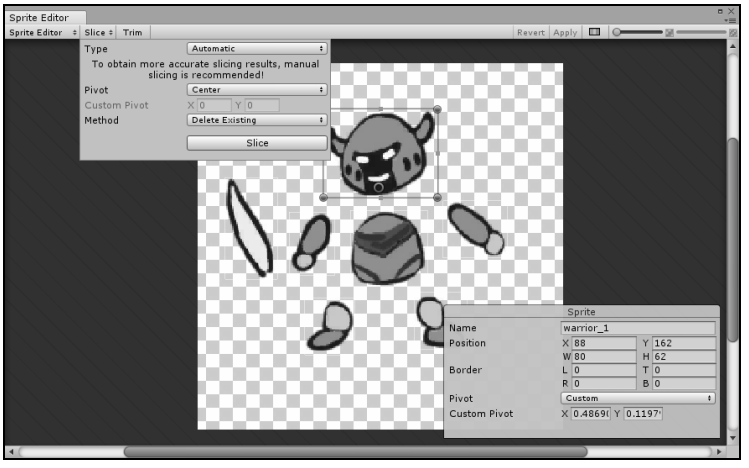

图 5-20　自动切分图片

步骤 03 在场景中新建一个空的 GameObject，将角色不同位置的图片放到它的层级下面作为子物体，拼接成完整的角色形象，注意不同位置的层级关系，如图 5-21 所示。

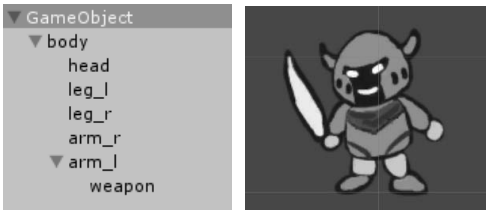

图 5-21　拼接角色

步骤 04 确定当前选择是角色的最顶级物体，在菜单栏中选择【Window】→【Animation】打开动画编辑窗口，单击左上方的“录制”按钮开始为角色录制动画，滑动时间轴到不同的时间，然后在场景中位移或旋转角色的不同部分，即可完成动画的制作，如图 5-22 所示。

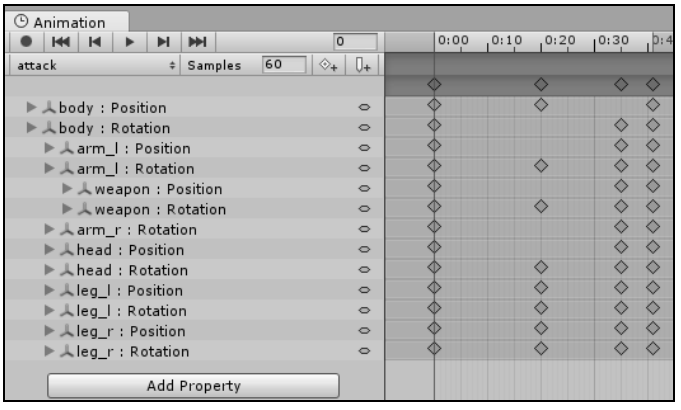

图 5-22　动画帧

提示　Ctrl+C 组合键和 Ctrl+V 组合键可以复制、粘贴关键帧。

最后的动画效果如图 5-23 所示。本节的示例工程文件保存在资源文件目录 c05_2dgame 的 bone animation 场景中。

图 5-23　动画

5.4　2D 物理

针对 2D 系统，Unity 提供了专门的 2D 物理功能。下面是一个示例，移动鼠标滑动一个球体，使它向相反的方向弹出去，在场景中有一些方块，如果球体弹到方块，则会将方块撞飞。

步骤 01 新建一个 Unity 的 2D 工程，复制资源文件目录 rawdata/2d/中的 box.png 图片到当前工程中。

步骤 02 将 box.png 的 Sprite Mode 设为 Multiple，然后自动分割，最后单击【Apply】按钮确定，如图 5-24 所示。

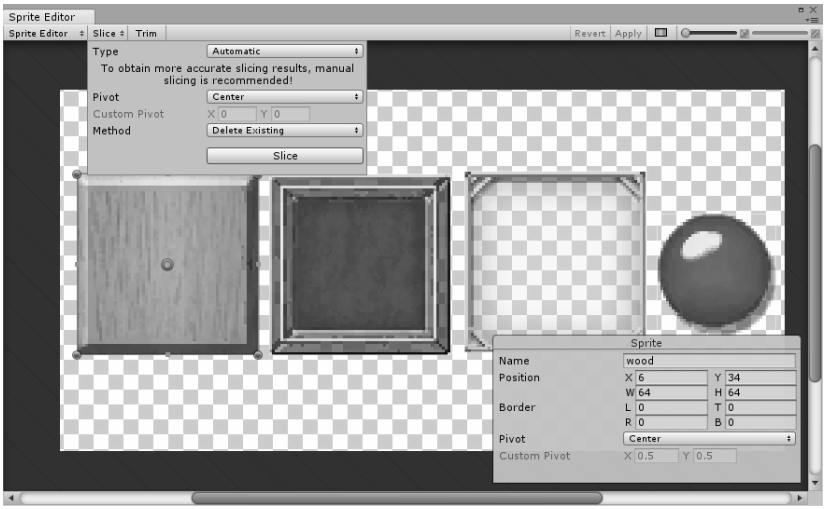

图 5-24　自动分割图片

步骤 03 制作场景的地面。创建一个 Sprite 并指定贴图，然后在菜单栏中选择【Component】→【Physics 2D】→【Rigidbody 2D】，添加一个 2D 刚体组件，将 Body Type 设为 Kinematic 使其不受物理碰撞或重力影响。在菜单栏中选择【Component】→【Physics 2D】→【Box Collider 2D】，添加一个 2D 矩形碰撞体组件，最后将 Sprite 复制多个以组成地面，如图 5-25 所示。

步骤 04 重复步骤（2）的操作，创建不同的 Sprite 作为物理碰撞的方块，唯一与步骤（2）不同的是不要将 Body Type 设为 Kinematic，如图 5-26 所示。

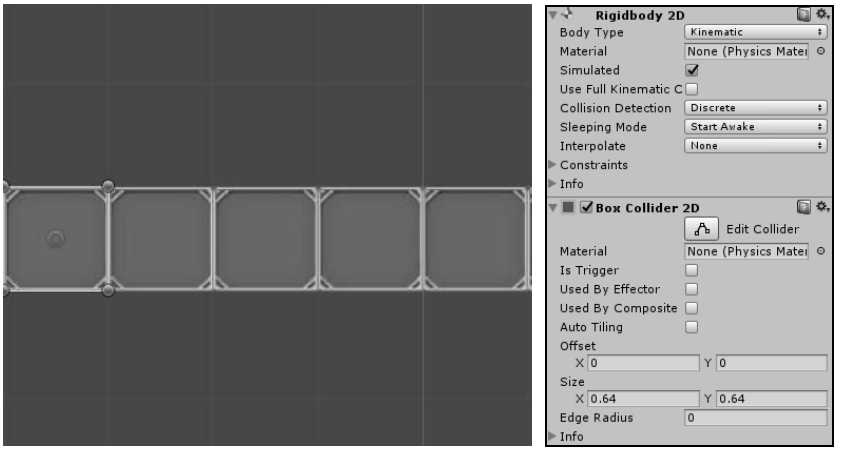

图 5-25　设置地面

步骤 05 创建一个 Sprite 作为球，添加 Rigidbody 2D 组件，将【Body Type】设为【Kinematic】，我们将在后面使用脚本打开这个选项。在菜单栏中选择【Component】→【Physics 2D】→【Circle Collider 2D】，添加一个 2D 圆形碰撞体组件，如图 5-27 所示。

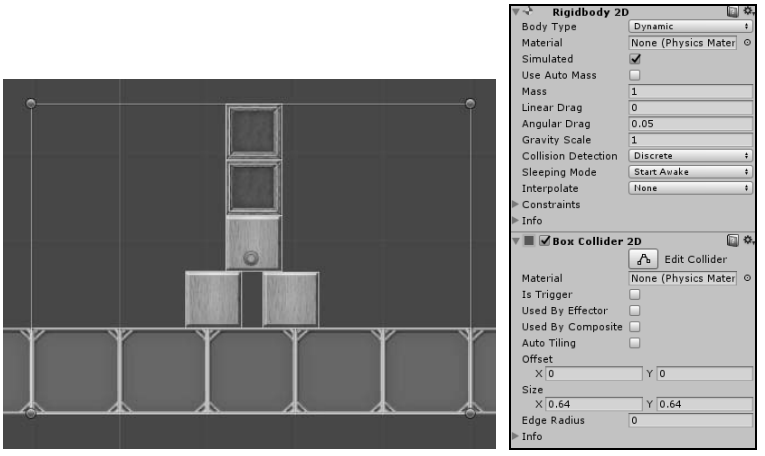

图 5-26　设置方块

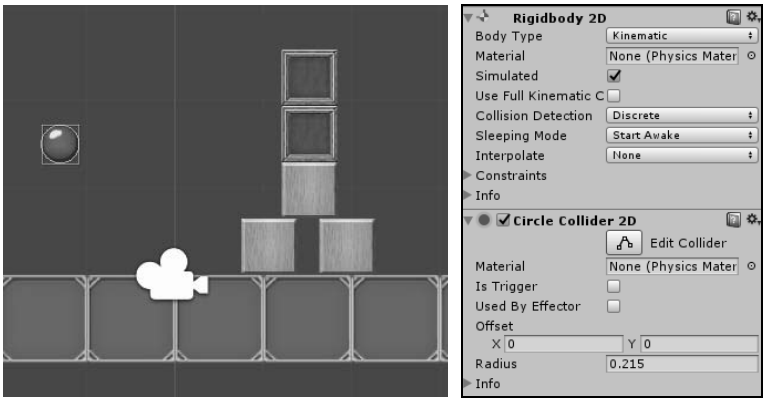

图 5-27　设置球

步骤 06　选中球的 Sprite，添加脚本 Ball.cs，添加代码如下：

```
public class Ball : MonoBehaviour {

    // 是否单击到球
    bool m_isHit = false;

    // 单击球时的位置
    Vector3 m_startPos;
    // Sprite 渲染器
    SpriteRenderer m_spriteRenderer;

     void Start () {
         m_spriteRenderer = this.GetComponent<SpriteRenderer>();
     }
}
```

步骤 07　添加 IsHit 函数，判断当前鼠标位置（在手机上就是手指的位置）是否碰到了球体的 Sprite，代码如下：

```
bool IsHit()
{
    m_isHit = false;
    // 获得鼠标位置
    Vector3 ms = Input.mousePosition;
    // 将鼠标位置由屏幕坐标转为世界坐标
    ms = Camera.main.ScreenToWorldPoint(ms);
    // 获得球的位置
    Vector3 pos = this.transform.position;
    // 获得球 Sprite 的宽和高（注意宽和高不是图片像素值的宽和高）
    float w = m_spriteRenderer.bounds.extents.x;
    float h = m_spriteRenderer.bounds.extents.y;
    // 判断鼠标的位置是否在 Sprite 的矩形范围内
    if (ms.x > pos.x - w && ms.x < pos.x + w &&
        ms.y > pos.y - h && ms.y < pos.y + h)
    {
        m_isHit = true;
        return true;
    }
    return m_isHit;
}
```

步骤 08 在 Update 函数中添加代码如下，先通过按住鼠标记录鼠标起始位置，再通过释放鼠标记录结束位置，通过两个位置算出矢量方向，给球加一个力，发射出去。

```
void Update () {
    // 如果单击鼠标左键并且碰到球
    if (Input.GetMouseButtonDown(0) && IsHit() )
    {
        // 记录位置
        m_startPos = Input.mousePosition;
    }
    // 当释放鼠标
    if (Input.GetMouseButtonUp(0) && m_isHit)
    {
        Vector3 endPos = Input.mousePosition;

        Vector3 v = (m_startPos - endPos) * 3.0f;
        // 将 body type 设为 Dynamic
        this.GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Dynamic;
        // 向球加一个力
        this.GetComponent<Rigidbody2D>().AddForce(v);
    }
}
```

运行程序，选中球向后滑动鼠标，释放鼠标会将球射出，如果碰到方块，会将方块弹飞，如图 5-28 所示。本节的示例工程文件保存在资源文件目录 c05_2dgame 的 2d physics 场景中。

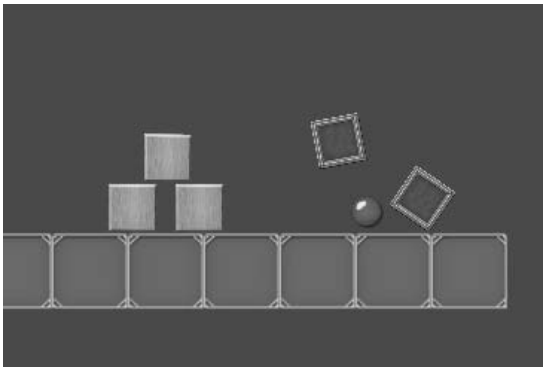

图 5-28　物理碰撞

5.5　捕鱼游戏

5.5.1　游戏玩法

接下来，我们将使用 Unity 的 2D 功能完成一个相对完整的 2D 游戏实例。游戏的玩法比较简单，在屏幕上有很多不同的鱼来回游动，我们的任务则是操作屏幕下方的一门大炮向鱼群开火，将鱼消灭（或称捕获）。

5.5.2　准备 2D 资源

步骤 01　新建一个工程，复制资源文件目录 rawdata\2d\fishgame 下的所有图片到当前工程目录中，这里包括背景、鱼、大炮、特效等图片，如图 5-29 所示。

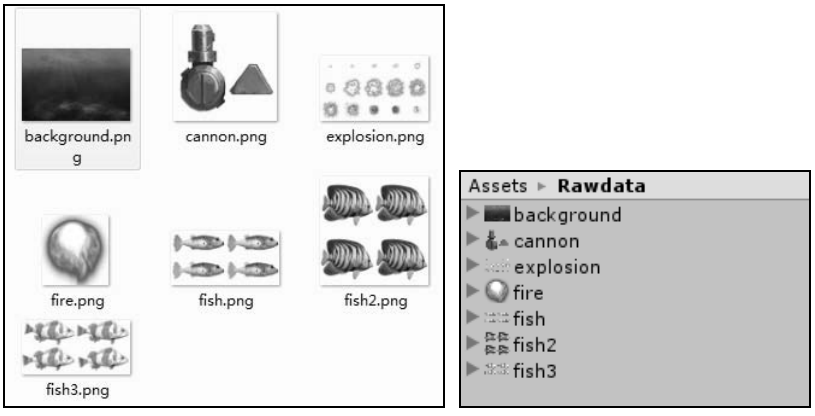

图 5-29　图片资源

步骤 02　将鱼、大炮和爆炸效果的 Sprite 设为 Multiple，使用 Sprite Editor 进行切割，注意大炮的 Sprite 需要将轴心点设置到炮的圆轴中心，如图 5-30 所示。

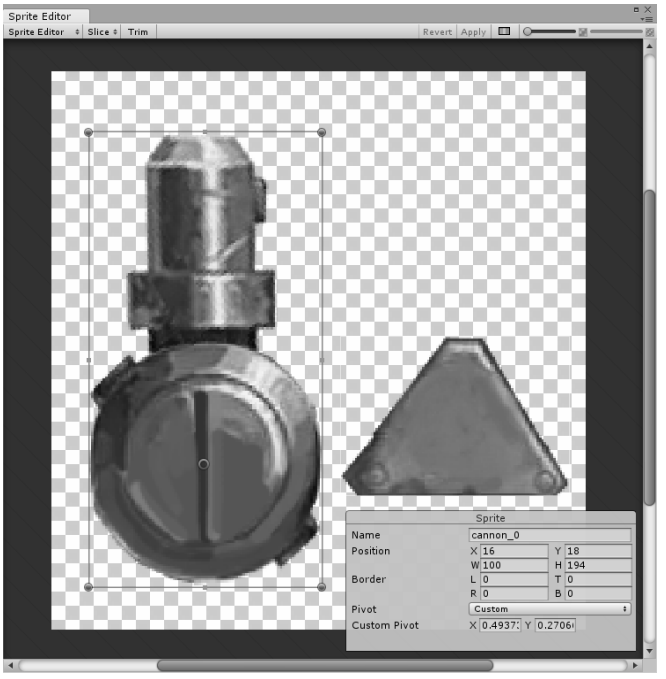

图 5-30　设置大炮的轴心点

步骤 03 将背景和大炮的 Sprite 放到场景中，在菜单栏中选择【Edit】→【Project Settings】→【Tags And Layers】，在 Sorting Layers 创建三个新层，然后选择背景和大炮的 Sprite，将背景设到 background 层，将大炮设到 weapon 层，如图 5-31 所示。

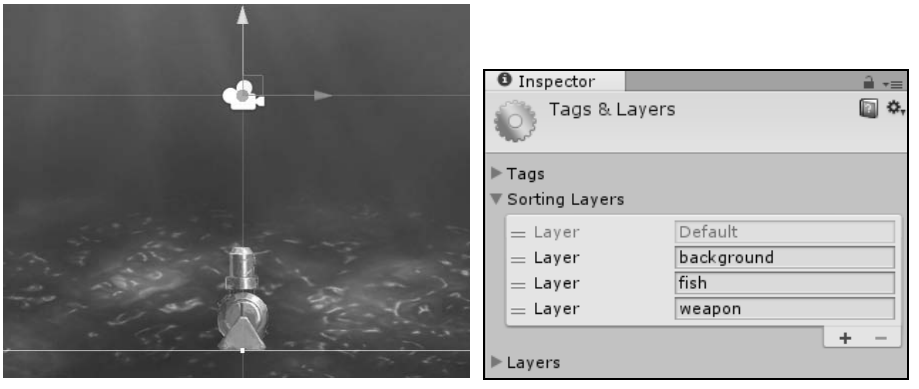

图 5-31　设置背景和层

步骤 04 同时选择鱼的 Sprite，将其拖放到场景中，这时会自动创建序列帧动画。使用相同的方式为所有的鱼和爆炸制作动画，如图 5-32 所示。

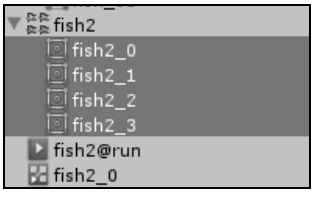

图 5-32　创建动画

步骤 05 为开火、带有动画的鱼和爆炸 Sprite 制作 Prefab，并存放到 Resources 文件夹内，如图 5-33 所示。将鱼 Sprite 的 Sorting Layer 设为 fish，将开火和爆炸设为 weapon。创建完成 Prefab 后，可以删除场景中的 Sprite。

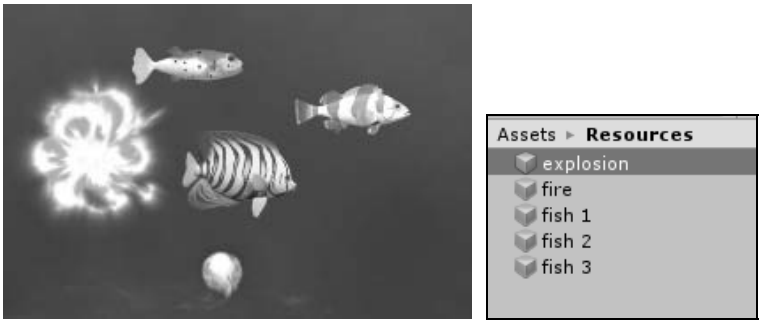

图 5-33　制作 Prefab

5.5.3　创建鱼

在准备好美术资源后，我们将使用脚本创建一条会游来游去的鱼。

步骤 01　创建脚本 Fish.cs，代码如下：

```
public class Fish : MonoBehaviour {

    protected float m_moveSpeed = 2.0f; //  鱼的移动速度
    protected int m_life = 10;   //  生命值

    public enum Target   //  移动方向
    {
        Left=0,
        Right=1
    }
    public Target m_target = Target.Right;          //  当前移动目标（方向）
    public Vector3 m_targetPosition;                //  目标位置

    public delegate void VoidDelegate( Fish fish );
    public VoidDelegate OnDeath;                    //  鱼死亡回调

    //  静态函数，创建一个 Fish 实例
    public static Fish Create(GameObject prefab, Target target, Vector3 pos )
    {
        GameObject go = (GameObject)Instantiate(prefab, pos, Quaternion.identity);
        Fish fish = go.AddComponent<Fish>();
        fish.m_target = target;

        return fish;
    }
    //  受到伤害
    public void SetDamage( int damage )
    {
        m_life -= damage;
        if (m_life <= 0)
        {
```

```
                GameObject prefab = Resources.Load<GameObject>("explosion");
                GameObject explosion = (GameObject)Instantiate(prefab, this.transform.position,
                                                this.transform.rotation);   // 创建鱼死亡时的爆炸效果
                Destroy(explosion, 1.0f);   // 1 秒后自动删除爆炸效果
                OnDeath(this);   // 发布死亡消息
                Destroy(this.gameObject);   // 删除自身
            }
        }

        void Start () {
            SetTarget();
        }
        // 设置移动目标
        void SetTarget()
        {
            // 随机值
            float rand = Random.value;

            // 设置 Sprite 翻转方向
            Vector3 scale = this.transform.localScale;
            scale.x = Mathf.Abs(scale.x) * (m_target == Target.Right ? 1 : -1);
            this.transform.localScale = scale;

            float cameraz = Camera.main.transform.position.z;
            // 设置目标位置
            m_targetPosition = Camera.main.ViewportToWorldPoint(new Vector3((int)m_target,
                                    1 * rand, -cameraz));
        }

        void Update () {
            UpdatePosition();
        }
        // 更新当前位置
        void UpdatePosition()
        {
            Vector3 pos = Vector3.MoveTowards(this.transform.position, m_targetPosition,
                                                m_moveSpeed*Time.deltaTime);
            if (Vector3.Distance(pos, m_targetPosition) < 0.1f) // 如果移动到目标位置
            {
                m_target = m_target==Target.Left ? Target.Right : Target.Left;
                SetTarget();
            }
            this.transform.position = pos;
        }
    }
```

在实际项目中，不能在 Update 中使用 Resources.Load: Instantiate 和 Destroy 函数，通常要使用缓存池减少游戏运行中的内存请求和释放的开销。

步骤 02 为了观察一下脚本的效果，可以将 Resources 下鱼的 Prefab 拖入场景，指定 Fish.cs 脚本。运行游戏，则会看到鱼在场景中来回游动。

5.5.4 创建鱼群生成器

步骤 01 创建脚本 FishSpawn.cs，用来生成鱼群，代码如下：

```
public class FishSpawn : MonoBehaviour {
    // 生成计时器
    public float timer = 0;
    // 最大生成数量
    public int max_fish = 30;
    // 当前鱼的数量
    public int fish_count = 0;

    void Update () {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            // 重新计时
            timer = 2.0f;

            // 如果鱼的数量达到最大数量则返回
            if (fish_count >= max_fish)
                return;

            // 随机 1、2、3 产生不同的鱼
            int index = 1 + (int)(Random.value * 3.0f);
            if (index > 3)
                index = 3;
            // 更新鱼的数量
            fish_count++;
            // 读取鱼的 prefab
            GameObject fishprefab = (GameObject)Resources.Load("fish " + index);

            float cameraz = Camera.main.transform.position.z;
            // 鱼的初始随机位置
            Vector3 randpos = new Vector3(Random.value, Random.value, -cameraz);
            randpos = Camera.main.ViewportToWorldPoint(randpos);

            // 鱼的随机初始方向
            Fish.Target target = Random.value > 0.5f ? Fish.Target.Right : Fish.Target.Left;
            Fish f = Fish.Create(fishprefab, target, randpos);
```

```
                // 注册鱼的死亡消息
                f.OnDeath+=OnDeath;
            }
             }
        void OnDeath( Fish fish )
        {
            // 更新鱼的数量
            fish_count--;
        }
}
```

这里的代码主要集中在 Update 函数中，每隔 2 秒创建一条鱼，在创建的过程中，使用随机数随机鱼的初始位置、方向和鱼的 Prefab。

步骤 02 在菜单栏中选择【GameObject】→【Create Empty】，创建一个空游戏体，指定 FishSpawn.cs 脚本。运行游戏，会不断地出现新的鱼游来游去，如图 5-34 所示。

图 5-34 鱼群

5.5.5 创建子弹和大炮

现在，游戏中已经有足够的鱼供我们捕猎了，接下来将创建子弹和大炮的功能。

步骤 01 在创建大炮之前，先要完成子弹的脚本。创建脚本 Fire.cs，添加代码如下：

```
public class Fire : MonoBehaviour {
    // 移动速度
    float m_moveSpeed = 10.0f;

    // 创建子弹实例，注意，在实际项目的 Update 中不建议使用 Resources.Load, Instantiate 和 Destroy
    public static Fire Create( Vector3 pos, Vector3 angle )
    {
        // 读取子弹 Sprite prefab
        GameObject prefab = Resources.Load<GameObject>("fire");
```

```
        // 创建子弹 Sprite 实例
    GameObject fireSprite = (GameObject)Instantiate(prefab, pos, Quaternion.Euler(angle));
        Fire f = fireSprite.AddComponent<Fire>();
        Destroy(fireSprite, 2.0f);
        return f;
    }

     void Update () {
        // 更新子弹位置
        this.transform.Translate(new Vector3(0, m_moveSpeed * Time.deltaTime, 0));
     }

    void OnTriggerEnter2D(Collider2D other)
    {
        Fish f =other.GetComponent<Fish>();
        if (f == null)
            return;
        else    // 如果击中鱼
            f.SetDamage(1);
        Destroy(this.gameObject);
    }
}
```

步骤 02 创建脚本 Canon.cs 实现大炮的逻辑，添加代码如下：

```
public class Cannon : MonoBehaviour {

    // 射击计时器
    float m_shootTimer = 0;

     void Update () {
          UpdateInput();
     }

    void UpdateInput()
    {
        m_shootTimer -= Time.deltaTime; // 更新射击间隔时间计时

        // 获得鼠标位置
        Vector3 ms = Input.mousePosition;
        ms = Camera.main.ScreenToWorldPoint(ms);

        // 大炮的位置
        Vector3 mypos = this.transform.position;

        // 单击鼠标左键开火
        if (Input.GetMouseButton(0))
        {
            // 计算鼠标位置与大炮位置之间的角度
```

```
                Vector2 targetDir = ms - mypos;
                float angle = Vector2.Angle(targetDir, Vector3.up);
                if (ms.x > mypos.x)
                    angle = -angle;
                this.transform.eulerAngles = new Vector3(0, 0, angle);

                if (m_shootTimer <= 0)
                {
                    m_shootTimer = 0.1f;  // 每隔 0.1 秒可以射击一次

                    // 开火，创建子弹实例
                    Fire.Create(this.transform.TransformPoint(0, 1, 0), new Vector3(0, 0, angle));
                }
            }
        }
    }
```

在 UpdateInput 函数中，计算出大炮到鼠标位置的角度，旋转大炮并发射子弹。运行游戏，单击鼠标左键发射子弹，如图 5-35 所示。

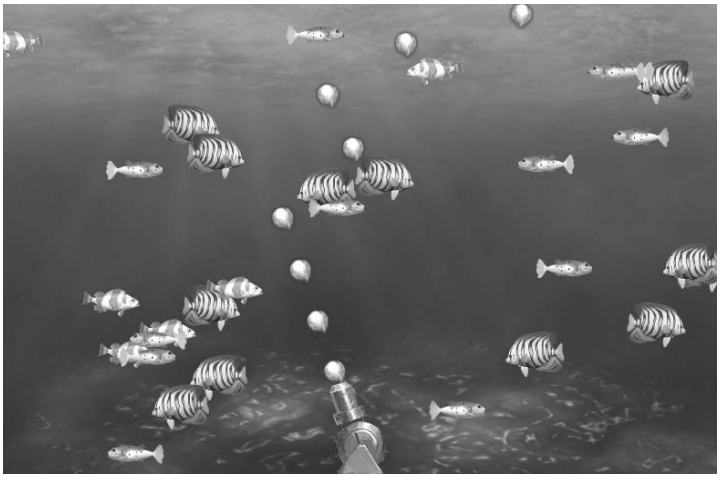

图 5-35　发射子弹

5.5.6　物理碰撞

现在虽然大炮可以发射子弹，但还不能消灭鱼。接下来我们将添加碰撞功能，使子弹可以碰撞到鱼。

步骤 01　在 Resources 文件夹中选择鱼的 Prefab，为它们添加 Rigidbody 2D 和 Polygon Collider 2D 组件，将【Body Type】设为【Kinematic】，将【Is Trigger】复选框选中，单击【Edit Collider】按钮可以改变多边形碰撞体顶点位置或增加顶点，按住 Ctrl 键可以删除顶点，如图 5-36 所示。

步骤 02　重复步骤（1）的操作对子弹 Prefab 做同样的处理。

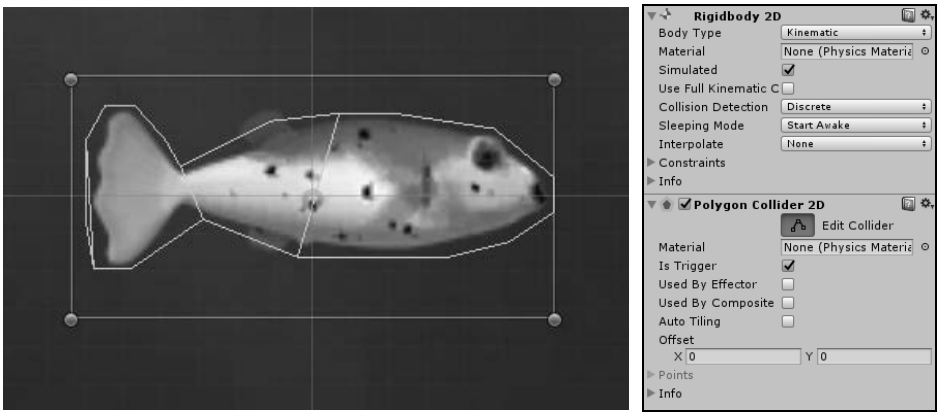

图 5-36　设置鱼的碰撞

运行游戏，子弹现在可以打到鱼了，最后的效果如图 5-37 所示。

图 5-37　2D 的爆炸效果

这个示例到这里就结束了，如果深入做下去，还可以添加不同的大炮、不同的鱼、得分系统等。本节的示例工程文件保存在资源文件目录 c05_fish2d。

5.6　2D 材质

5.6.1　修改 Sprite 颜色

尽管 Sprite 多用于 2D 游戏，但它也有自己的材质和 Shader。下面的代码获取了 SpriteRenderer 的默认 material，并修改了默认的颜色，使 Sprite 的颜色变为红色。

```
SpriteRenderer render = this.GetComponent<SpriteRenderer>();
render.material.color = new Color(1, 0, 0, 1);
```

5.6.2 自定义的黑白效果材质

2D 游戏中一种常见的效果就是将图片变为黑白色。为了实现这种效果，需要修改默认的 Sprite 材质，步骤如下：

步骤 01 在 http://unity3d.com/unity/download/archive/站点下载最新的 Unity 官方 shader 源代码，在本示例工程中提供的 builtin_shaders.zip 包括 Unity5.5.2 版本的 Shader 源代码。

步骤 02 解压 shader 压缩包，找到 Sprites-Default.shader 文件并导入到 Unity 工程中。

步骤 03 双击 Sprites-Default.shader 修改它的代码，先修改第一行，将 Shader 的名字重命名，这里改为 Sprite/Gray，然后修改 fixed4 frag(v2f IN)函数，添加两行代码，使输出的颜色变成灰度显示，如下所示：

```
Shader "Sprites/Gray"
fixed4 frag(v2f IN) : SV_Target
{
    fixed4 c = SampleSpriteTexture (IN.texcoord) * IN.color;
    c.rgb *= c.a;
    // 添加的代码
    float gray = dot(c.xyz, float3(0.299, 0.587, 0.114));
    c.xyz = float3(gray, gray, gray);

    return c;
}
```

步骤 04 创建一个新的 Material 指定给 Sprite，并使用自定义的黑白效果 Shader，即可使图片变为黑白色。也可以通过代码动态地改变 shader 实现该效果，代码如下：

```
SpriteRenderer render = this.GetComponent<SpriteRenderer>();
render.material.shader = Shader.Find("Sprites/Gray");
```

本节的示例工程文件保存在资源文件目录 c05_2dgame 的 change shader 场景中。

5.7 小　　结

本章介绍了 Unity 在 2D 游戏中的应用，包括创建 Sprite、不同的动画方式、2D 物理的应用，最后还通过一个较为完整的实例，介绍了如何创建一个 2D 捕鱼游戏。尽管我们在这里是开发 2D 游戏，但因为 Unity 也是一个 3D 游戏引擎，我们也可以在 3D 的模式下使用 2D 功能，创造出 3D、2D 混合的图形效果。