

---

# Unity 4.x 2D

## 游戏开发基础教程

（内部资料）

大学霸



大学霸

[www.daxueba.net](http://www.daxueba.net)

## 前 言

Unity 是一款综合性的游戏开发工具，也是一款全面整合的专业游戏引擎。它可以运行在 Windows 和 Mac OS X 下，并提供交互的图形化开发环境为首要操作方式。使用 Unity 开发的游戏，可以部署到所有的主流游戏平台，而无需任何修改。这些平台包括 Windows、Linux、Mac OS X、iOS、Android、Xbox 360、PS3、WiiU 和 Web 等。开发者无需过多考虑平台之间的差异，只需把精力集中到制作高质量的游戏即可，真正做到“一次开发，到处部署”。

据权威机构统计，国内 53.1% 的人使用 Unity 进行游戏开发；有 80% 的手机游戏是使用 Unity 开发的；苹果应用商店中，有超过 1500 款游戏使用 Unity 开发。

网上有为数众多的 2D 和 3D 游戏。稍微关注一下，就会发现 2D 游戏才是主流，如植物大战僵尸、愤怒的小鸟、打飞机、2048 等。而且，问问身边的人让他们印象深刻的游戏是什么，你会惊讶的发现，大部分游戏同样是 2D 的。

基于以上不可忽略的事实，本书决定着眼于讲解使用 Unity 开发 2D 游戏的基础知识，且书中包含了两个生动的 2D 游戏示例，相信读者会喜欢它们的。

### 1. 学习所需的系统和软件

- ☐ 安装 Windows 7 操作系统
- ☐ 安装 Unity 4.5.1

### 2. 学习建议

大家学习之前，可以致信到 [unity2d@daxueba.net](mailto:unity2d@daxueba.net)，获取相关的资料和软件。如果大家在学习过程遇到问题，也可以将问题发送到该邮箱。我们尽可能给大家解决。

# 目 录

第 1 章	Unity 及其组成的介绍.....	错误!未定义书签。
1.1	Unity 概述.....	错误!未定义书签。
1.2	项目、资源和场景.....	错误!未定义书签。
1.2.1	项目.....	错误!未定义书签。
1.2.2	资源.....	错误!未定义书签。
1.2.3	场景.....	错误!未定义书签。
1.3	场景视图的操作.....	错误!未定义书签。
1.3.1	使用快捷键操作场景视图.....	错误!未定义书签。
1.3.2	使用 Gizmo 操作场景视图.....	错误!未定义书签。
1.4	游戏对象和组件.....	错误!未定义书签。
1.5	脚本与脚本编辑器.....	错误!未定义书签。
1.5.1	创建脚本.....	错误!未定义书签。
1.5.2	脚本编辑器.....	错误!未定义书签。
1.6	脚本的调试.....	错误!未定义书签。
1.6.1	调试方法一.....	错误!未定义书签。
1.6.2	调试方法二.....	错误!未定义书签。
第 2 章	材质和纹理.....	错误!未定义书签。
2.1	材质和纹理的使用.....	错误!未定义书签。
2.1.1	使用材质.....	错误!未定义书签。
2.1.2	不同的材料类型——着色器.....	错误!未定义书签。
2.1.3	使用纹理.....	错误!未定义书签。
2.2	应用于 2D 游戏的材质.....	错误!未定义书签。
2.2.1	缘由.....	错误!未定义书签。
2.2.2	技巧一：使用白色的环境光.....	错误!未定义书签。
2.2.3	技巧二：使用光不敏感着色器.....	错误!未定义书签。
2.3	纹理使用规则.....	错误!未定义书签。
2.3.1	规则 1：分辨率是 2 的次方.....	错误!未定义书签。
2.3.2	规则 2：保证“质量”.....	错误!未定义书签。
2.3.3	规则 3：增加阿尔法通道（Alpha Channel）.....	错误!未定义书签。
2.4	导入纹理.....	错误!未定义书签。
2.4.1	导入纹理时默认设置介绍.....	错误!未定义书签。
2.4.2	含有透明信息的纹理.....	错误!未定义书签。
第 3 章	着手开发一个简单的 2D 游戏.....	错误!未定义书签。
3.1	开始开发 2D 游戏.....	错误!未定义书签。
3.1.1	导入纹理资源.....	错误!未定义书签。
3.1.2	新建材质资源.....	错误!未定义书签。
3.1.3	修改场景的环境光以及游戏时的屏幕尺寸.....	错误!未定义书签。
3.2	为场景添加游戏对象.....	错误!未定义书签。

3.2.1	调整游戏对象的角度.....	错误!未定义书签。
3.2.2	改变游戏对象的位置.....	错误!未定义书签。
3.2.3	游戏对象的“碰撞”组件.....	错误!未定义书签。
3.3	让飞船动起来.....	错误!未定义书签。
3.4	让飞船发射子弹.....	错误!未定义书签。
3.4.1	在场景中添加子弹.....	错误!未定义书签。
3.4.2	游戏时，让子弹在场景中移动.....	错误!未定义书签。
3.4.3	生成子弹的预设.....	错误!未定义书签。
3.4.4	设置子弹的发射位置.....	错误!未定义书签。
3.4.5	在恰当的时机发射子弹.....	错误!未定义书签。
3.5	让外星飞船动起来.....	错误!未定义书签。
3.5.1	编写脚本.....	错误!未定义书签。
3.5.2	设置外星飞船的触发器.....	错误!未定义书签。
3.5.3	为子弹预设添加刚体组件.....	错误!未定义书签。
3.6	为游戏添加背景.....	错误!未定义书签。
第 4 章	使用编辑器类自定义编辑器.....	错误!未定义书签。
4.1	编辑器类.....	错误!未定义书签。
4.2	开始使用编辑器类编写工具.....	错误!未定义书签。
4.2.1	为项目添加脚本.....	错误!未定义书签。
4.2.2	创建指定名称的文件夹.....	错误!未定义书签。
4.3	把工具添加到菜单.....	错误!未定义书签。
4.3.1	CreateWizard函数.....	错误!未定义书签。
4.3.2	测试脚本的实现效果.....	错误!未定义书签。
4.4	读取场景中选择对象.....	错误!未定义书签。
4.4.1	在脚本中使用Selection类.....	错误!未定义书签。
4.4.2	测试脚本的实现效果.....	错误!未定义书签。
4.5	为工具窗口添加用户输入框.....	错误!未定义书签。
4.6	完成工具的所有功能.....	错误!未定义书签。
第 5 章	图片与几何图形对象.....	错误!未定义书签。
5.1	2D游戏常用的图片.....	错误!未定义书签。
5.1.1	精灵.....	错误!未定义书签。
5.1.2	图块集.....	错误!未定义书签。
5.1.3	图形绘制中的问题.....	错误!未定义书签。
5.1.4	设想.....	错误!未定义书签。
5.2	开始编写编辑器工具.....	错误!未定义书签。
5.3	设置四边形的轴点.....	错误!未定义书签。
5.4	指定四边形资源的存放路径.....	错误!未定义书签。
5.5	生成四边形.....	错误!未定义书签。
5.5.1	阶段一：创建构成四边形的顶点、UV和三角形.....	错误!未定义书签。
5.5.2	阶段二：在资源面板中生成四边形.....	错误!未定义书签。
5.5.3	阶段三：在场景中实例化一个四边形.....	错误!未定义书签。
5.6	使用四边形生成工具.....	错误!未定义书签。
第 6 章	生成纹理图集.....	错误!未定义书签。

6.1	为什么要使用纹理图集.....	错误!未定义书签。
6.1.1	降低绘制调用的次数.....	错误!未定义书签。
6.1.2	便于灵活的使用纹理.....	错误!未定义书签。
6.1.3	便于管理纹理.....	错误!未定义书签。
6.2	开始编写生成纹理图集的工具.....	错误!未定义书签。
6.3	添加组成纹理图集的纹理.....	错误!未定义书签。
6.4	UV对纹理图集的重要性.....	错误!未定义书签。
6.5	生成纹理图集.....	错误!未定义书签。
6.5.1	步骤一：优化输入的纹理.....	错误!未定义书签。
6.5.2	步骤二：构建纹理图集.....	错误!未定义书签。
6.5.3	步骤三：保存图集的预置.....	错误!未定义书签。
6.6	脚本文件TexturePacker代码汇总.....	错误!未定义书签。
6.7	测试工具的使用效果.....	错误!未定义书签。
第 7 章	UV和动画.....	错误!未定义书签。
7.1	生成一个可停靠的编辑器.....	错误!未定义书签。
7.2	编辑工具窗口的界面.....	错误!未定义书签。
7.2.1	添加预置资源选择区域.....	错误!未定义书签。
7.2.2	添加纹理选择区域.....	错误!未定义书签。
7.2.3	添加纹理选择的两种方式.....	错误!未定义书签。
7.2.4	编写用于修改网格对象UV坐标的函数.....	错误!未定义书签。
7.2.5	添加应用所有设置的按钮.....	错误!未定义书签。
7.3	工具脚本代码的汇总与使用.....	错误!未定义书签。
7.4	一个播放动画的平面对象.....	错误!未定义书签。
第 8 章	益于 2D 游戏的摄像机与场景设置.....	错误!未定义书签。
8.1	摄像机类型：透视与正交.....	错误!未定义书签。
8.2	世界单元与像素.....	错误!未定义书签。
8.3	世界单元与像素的转换.....	错误!未定义书签。
8.3.1	添加纹理和四边形对象.....	错误!未定义书签。
8.3.2	调整四边形与摄像机的位置.....	错误!未定义书签。
8.3.3	世界单元：像素 = 1: 1.....	错误!未定义书签。
8.3.4	对齐屏幕和场景坐标的原点.....	错误!未定义书签。
8.4	纹理图片的完美显示.....	错误!未定义书签。
8.5	其它有用的设置技巧.....	错误!未定义书签。
8.5.1	调节深度.....	错误!未定义书签。
8.5.2	合成视图.....	错误!未定义书签。
第 9 章	获取玩家对 2D 游戏的输入.....	错误!未定义书签。
9.1	自动检测鼠标单击事件.....	错误!未定义书签。
9.2	手动检测鼠标单击事件.....	错误!未定义书签。
9.2.1	鼠标按下的键及其位置.....	错误!未定义书签。
9.2.2	鼠标点击的第一个对象.....	错误!未定义书签。
9.2.3	鼠标点击的所有对象.....	错误!未定义书签。
9.3	修改游戏中的鼠标图标.....	错误!未定义书签。
9.3.1	准备所需的资源，并做适当设置.....	错误!未定义书签。



9.3.2	编写脚本.....	错误!未定义书签。
9.3.3	两个坐标系导致的问题.....	错误!未定义书签。
9.3.4	查看游戏视图中的效果.....	错误!未定义书签。
9.4	使用键盘控制鼠标移动.....	错误!未定义书签。
9.5	对输入的抽象——输入轴.....	错误!未定义书签。
9.5.1	了解输入轴.....	错误!未定义书签。
9.5.2	输入轴在输入过程中的应用.....	错误!未定义书签。
9.6	来自移动设备的输入.....	错误!未定义书签。
9.6.1	检测移动设备上的触摸操作.....	错误!未定义书签。
9.6.2	把触摸操作当作鼠标操作.....	错误!未定义书签。
9.6.3	有选择的编译代码.....	错误!未定义书签。
第 10 章	2D 卡片游戏——记忆大作战.....	1
10.1	游戏设计文档.....	8
10.2	开始着手创建游戏.....	9
10.2.1	在资源面板创建文件夹.....	9
10.2.2	创建一个纹理图集.....	10
10.2.3	创建四边形对象.....	12
10.2.4	修改四边形的材质和 UV.....	13
10.2.5	设置摄像机和游戏视图的分辨率.....	16
10.3	设置场景中的卡片.....	17
10.3.1	设置卡片的属性.....	18
10.3.2	定位卡片的位置.....	20
10.3.3	编写控制卡片行为的脚本.....	21
10.3.4	补全场景中其余的卡片.....	24
10.4	游戏管理类.....	24
10.4.1	重置卡片.....	25
10.4.2	处理玩家输入.....	26
10.4.3	响应玩家输入.....	27
10.4.4	游戏管理类代码汇总.....	28
10.5	完善并运行游戏.....	32
10.5.1	替换系统鼠标图标.....	33
10.5.2	游戏运行效果展示.....	33
第 11 章	可联机玩的游戏——记忆大作战.....	错误!未定义书签。
11.1	网络连接.....	错误!未定义书签。
11.2	建立服务器端.....	错误!未定义书签。
11.3	建立客户端.....	错误!未定义书签。
11.4	测试网络连接的功能.....	错误!未定义书签。
11.5	网络视图组件.....	错误!未定义书签。
11.6	构建授权服务器.....	错误!未定义书签。
11.7	建立游戏输入操作的秩序.....	错误!未定义书签。
11.7.1	游戏启动时，禁止输入操作.....	错误!未定义书签。
11.7.2	连接建立后，允许服务器端的输入操作.....	错误!未定义书签。
11.7.3	服务器端远程调用客户端上的函数.....	错误!未定义书签。

11.7.4 客户端远程调用服务器端上的函数.....	错误!未定义书签。
11.8 修改游戏管理类脚本.....	错误!未定义书签。
11.9 游戏运行效果展示.....	错误!未定义书签。
11.10 为游戏添加分数记录.....	错误!未定义书签。
第 12 章 优化游戏的方法.....	错误!未定义书签。
12.1 最优化，如你所想吗？.....	错误!未定义书签。
12.2 减少顶点的数目.....	错误!未定义书签。
12.3 减少材质.....	错误!未定义书签。
12.4 减少UV接缝.....	错误!未定义书签。
12.5 不同平台下，纹理的不同设置.....	错误!未定义书签。
12.6 对象缓存组件.....	错误!未定义书签。
12.7 避免频繁使用Update()函数.....	错误!未定义书签。
12.8 合理使用Collider组件.....	错误!未定义书签。
12.9 避免使用OnGUI()和GUI类.....	错误!未定义书签。
12.10 使用静态批处理.....	错误!未定义书签。
12.11 使用天空盒子.....	错误!未定义书签。

# 大学霸

www.daxueba.net

## 第 10 章 2D 卡片游戏——记忆大作战

前面各章节已经学习了不少有关 2D 游戏开发的技巧，相信读者也掌握的差不多了。本章打算应用这些知识，制作一个卡片类的游戏，它用于考验玩家对位置的记忆能力，如图 10-1 所示。

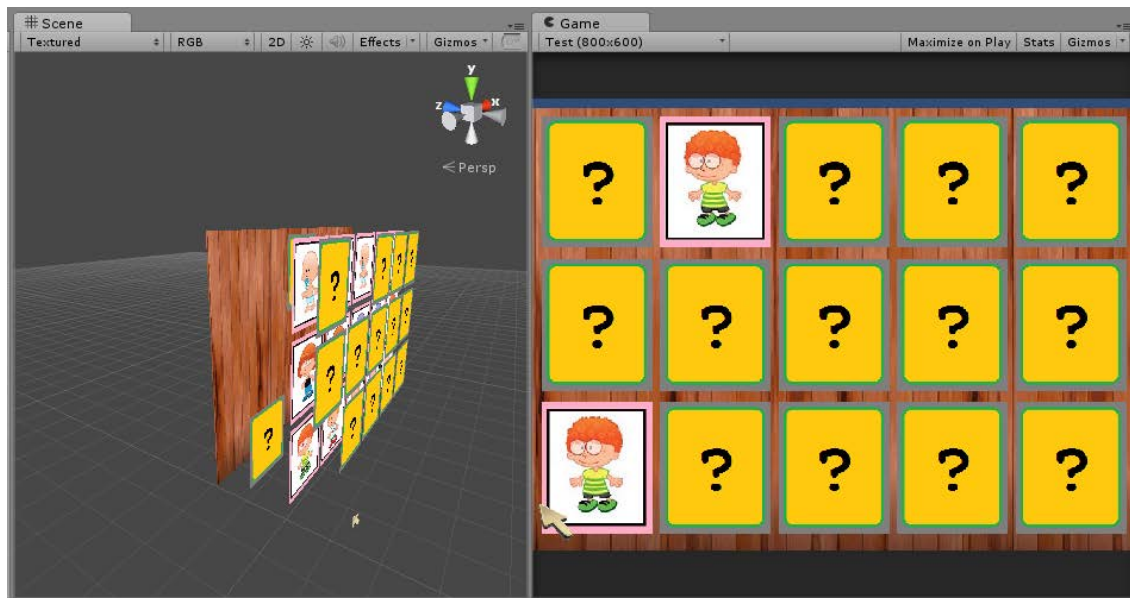


图 10-1 本章将要制作的卡片类游戏

### 10.1 游戏设计文档

无所谓是简单的游戏，还是复杂的游戏，在开发前都需要有一个明确的设计思路。有时这个设计思路只存在于你的脑袋里；有时这个思路已经形成了详细的游戏设计文档（game design document，即 GDD）。

本节打算做一个游戏设计文档，不过这是一个非常简单的文档，仅仅说明了游戏的内容以及规则，如下：

**游戏名称：**记忆大作战

**游戏内容描述：**

- ☐ 游戏的界面由 5 行 3 列，总共 15 张的卡片组成，其中包括 5 种不同类型的卡片，每种类型的卡片有 3 张。
- ☐ 卡片的背面面向玩家，玩家无法看到卡片的内容。
- ☐ 卡片的分布是随机的。
- ☐ 每轮玩家先翻开一张卡片，确定了卡片的樣子后，继续翻其它的卡片。若相同，卡片同样不再自动翻转；若不同，卡片会自动翻转。直到找到 3 张同样的卡片后，本轮结束。3 张被翻转的卡片会从游戏的界面中移除，然后进入下一轮。



**游戏胜利条件：**当游戏完成 5 轮，也就是所有相同的 3 张卡片都被找到，且被从界面移除后，游戏结束。

提示：人们习惯上，把一些设想中的 2D 游戏称为“简单的”、“小的”和“容易的”的游戏。我建议大家不要带着这样的想法马上开始创建这个游戏。这种想法会让你低估这个游戏。而导致这种想法的原因通常是大家不了解技术细节，以及涉及的其它问题。而没有前期的计划设计，即使是一个“简单的”游戏，也有可能变成“烂尾”——很容易启动一个游戏开发项目，却很难完成。

## 10.2 开始着手创建游戏

有了完整的游戏设计思路，以及设计文档，就可以马上使用 Unity 新建一个游戏项目了。本节把这个游戏项目命名为 CardMatch，且不导入任何 Unity 的资源包，如图 10-2 所示。

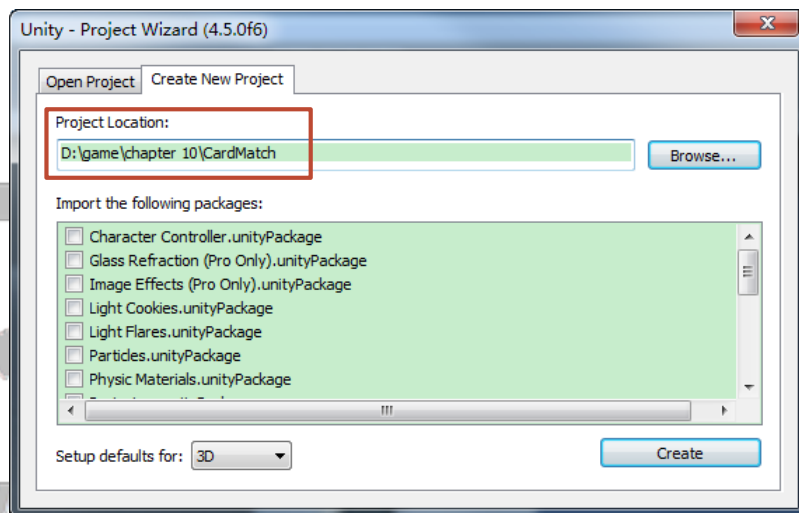


图 10-2 新建一个游戏项目

### 10.2.1 在资源面板创建文件夹

凡是游戏项目，制作的过程中一定会使用到各种资源类型，如纹理、脚本、材质等。因此在新建了游戏项目以后，最好在资源面板中提前创建好多个文件夹，以后将用于放置各种类型的资源。我们提前创建的文件夹是：Scripts, Scenes, Textures, Materials, Quads, Editor 和 Atlas\_data，如图 10-3 所示。



图 10-3 提前在游戏项目的资源面板中创建的文件夹

在以后开发游戏的过程中，随时将对应的资源放入对应的文件夹。这样做会使得资源面板更整洁，查找对应的资源也会更容易些。

## 10.2.2 创建一个纹理图集

本书第 6 章编写过一个编辑器工具，可以将多个纹理生成一个纹理图集。使用纹理图集可以减少绘制调用（draw calls），进而提升游戏的运行效果和画面质量，这点在第 6 章有详细的解释。本小节打算使用那时候创建的工具，将本游戏要用的纹理全部生成到一个图集中。

将本游戏项目要用到的纹理，导入到资源面板的 Textures 文件夹下，如图 10-4 所示。



图 10-4 导入到游戏项目中的纹理

提示：名称为 1~8 的纹理中，将有 5 个会被作为游戏时的不同卡片类型；名为 9 的纹理将用于遮挡游戏时的 15 张卡片；名为 10 的纹理将被用作游戏时的背景。

首先，需要在资源面板的 Editor 文件夹下，创建名为 TexturePacker 的 C#脚本文件，然后把第 6 章对应脚本中的代码拷贝过来。然后，在资源面板的 Scripts 文件夹下，创建名为 AtlasData 的 C#脚本文件，同样将第 6 章对应脚本中的代码拷贝过来即可，如图 10-5 所示。此时用于生成纹理图集的工具就可以用在本游戏的项目中了。

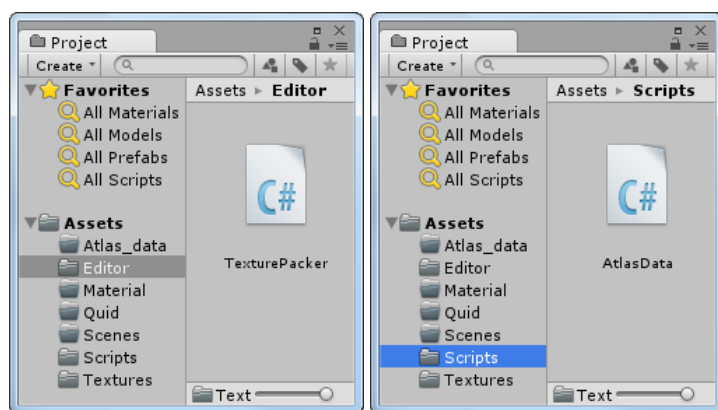


图 10-5 在对应文件夹下的两个脚本文件

先选中 Textures 文件夹下的所有纹理，然后单击 **GameObject|Create Other|Atlas Texture** 命令，打开纹理图集生成工具，如图 10-6 所示。

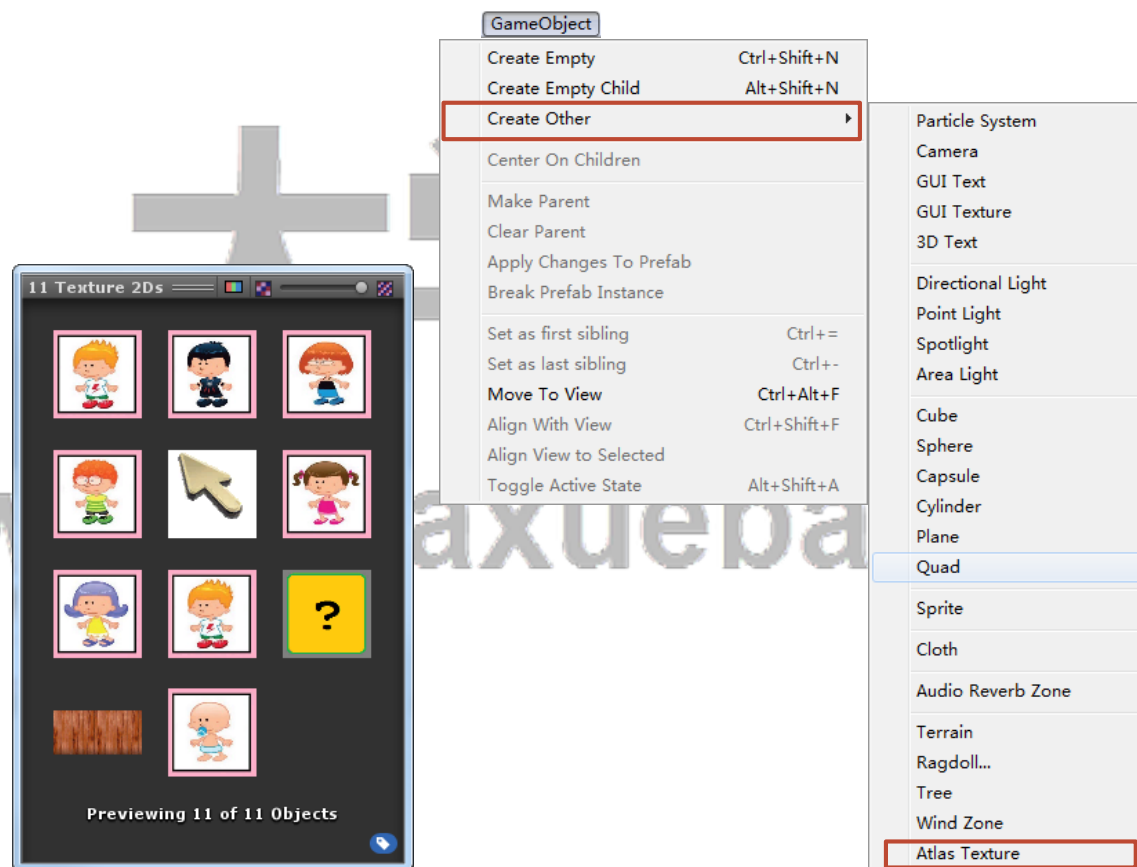


图 10-6 先选中要生成纹理图集的所有纹理，然后单击命令打开对应工具

此时纹理图集工具的窗口中就会添加上所有被选中的纹理了，还可以在工具的窗口中给即将生成的纹理图集命名，以及设置纹理图集中每个纹理的间隔（Padding），最后单击工具窗口上的 **Create** 按钮即可在资源面板中生成纹理图集和记录相应数据的预置对象，如图 10-7 所示。

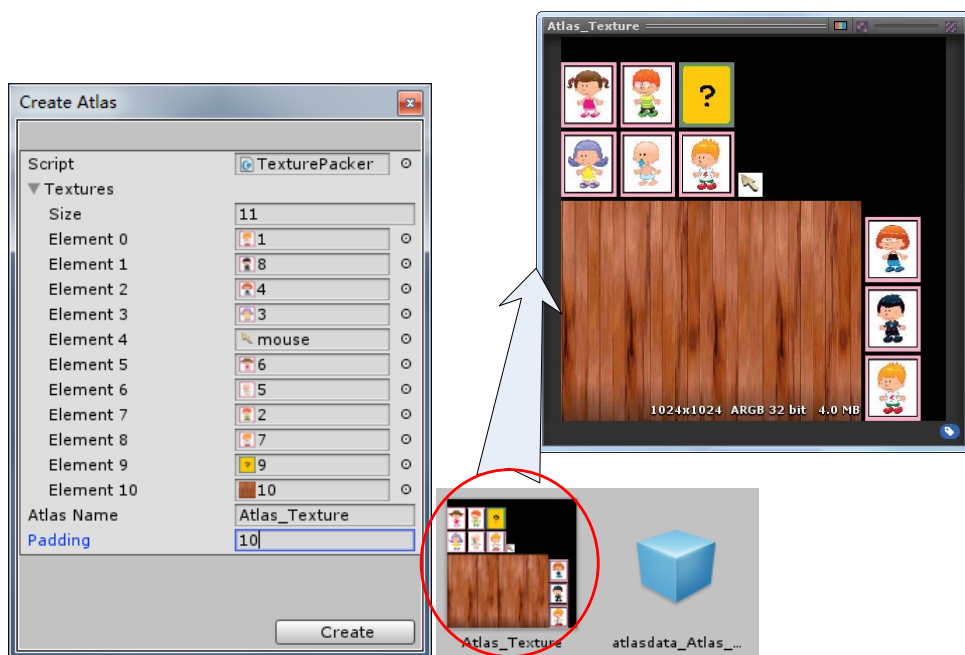


图 10-7 使用编辑器工具生成纹理图集和纹理预置对象

最后,要记得把生成的纹理图集和纹理预置对象,放到资源面板的Atlas\_data文件夹下。虽然这么做不是必须的,但是考虑到资源面板的整洁性,建议随时将资源面板中的资源归类放置。

### 10.2.3 创建四边形对象

创建的游戏将会包括 8 个重要的部分: 5 种不同的卡片类型, 1 个用于遮挡的卡片, 1 个游戏时的背景, 1 个自定义的鼠标。无论是哪个组成部分, 都需要被应用在场景中的四边形对象之上。这里就需要使用我们在第 5 章编写的四边形生成工具, 生成与纹理图片大小一致的四边形。

首先, 需要在资源面板的 Editor 文件夹下, 创建名为 CreateQuad 的 C#脚本文件, 然后把第 5 章对应脚本中的代码拷贝过来, 如图 10-8 所示。

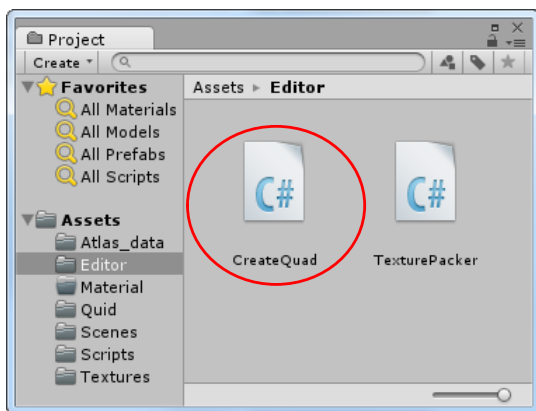


图 10-8 在对应文件夹下的脚本文件

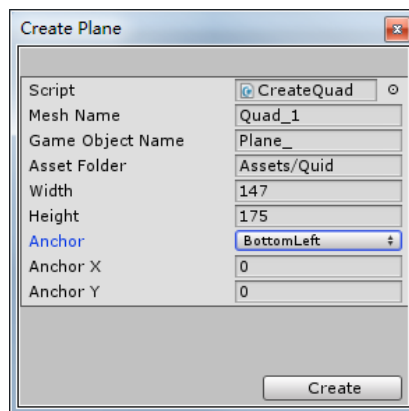


图 10-9 使用工具生成用于卡片的四边形

本游戏中: 卡片的大小是  $147 \times 175$ , 游戏背景的大小是  $800 \times 587$ , 鼠标大小是  $64 \times 64$ 。

现在需要使用工具生成 6 卡片大小、1 个游戏背景大小和 1 个鼠标大小的四边形。如图 10-9 是生成卡片所需四边形时，窗口中各项的设置。

提示：为了接下来使用的方便，可以设置四边形的轴点为四边形的左下角。

记得要将资源面板中生成的四边形放到 **Quid** 文件夹下。图 10-10 为游戏场景中生成的 8 个四边形对象，以及层次面板中各四边形的名称，还有资源面板中的 8 个四边形资源。

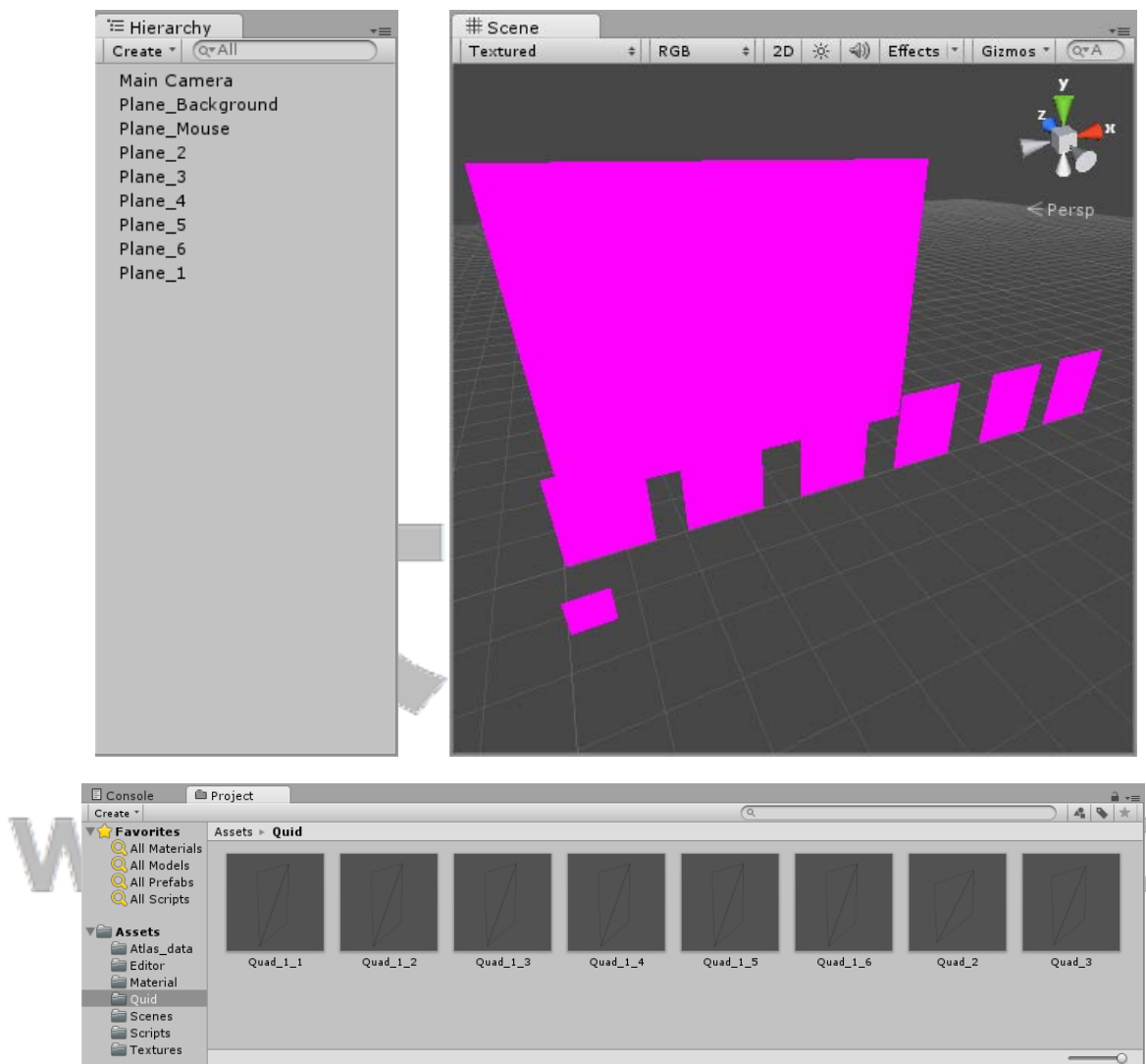


图 10-10 生成的 8 个四边形对象

从四边形对象在场景视图中的大小也可以分辨出各四边形对象以后的用途。

#### 10.2.4 修改四边形的材质和 UV

未设置材质的四边形对象，在场景视图使用粉色表示。场景中此时有 8 个四边形对象，而本小节将会让 8 个四边形对象显示合适的纹理。

首先，在资源面板的 **Material** 文件夹下创建一个材质，并命名为 **mat\_atlas**，设置此材质的着色器类型为 **Unlit/Transparent Cutout Shader**，并引用我们创建的纹理图集，如图 10-11 所示。



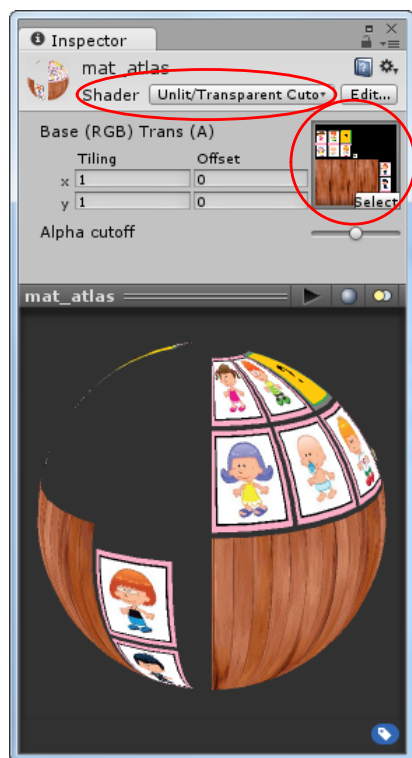


图 10-11 创建并设置材质资源

图 10-12 将材质应用到场景中的所有四边形对象上

接下来，把此材质应用到场景中的所有四边形对象上，如图 10-12 所示，可以看到所有四边形对象都在显示整个纹理图集，而不是特定的纹理，因为四边形的 UV 值被设置成了显示整个纹理图集。此时就需要使用第 7 章编写的工具，修改每个四边形对象的 UV 值，让它们显示指定的纹理。

在资源面板的 Editor 文件夹下，创建名为 UVEditor 的 C#脚本文件，并将第 7 章对应脚本文件中的代码拷贝进来。保存以后，此工具就可通过单击 Window|Atlas UV Editor 命令打开，如图 10-13 所示。

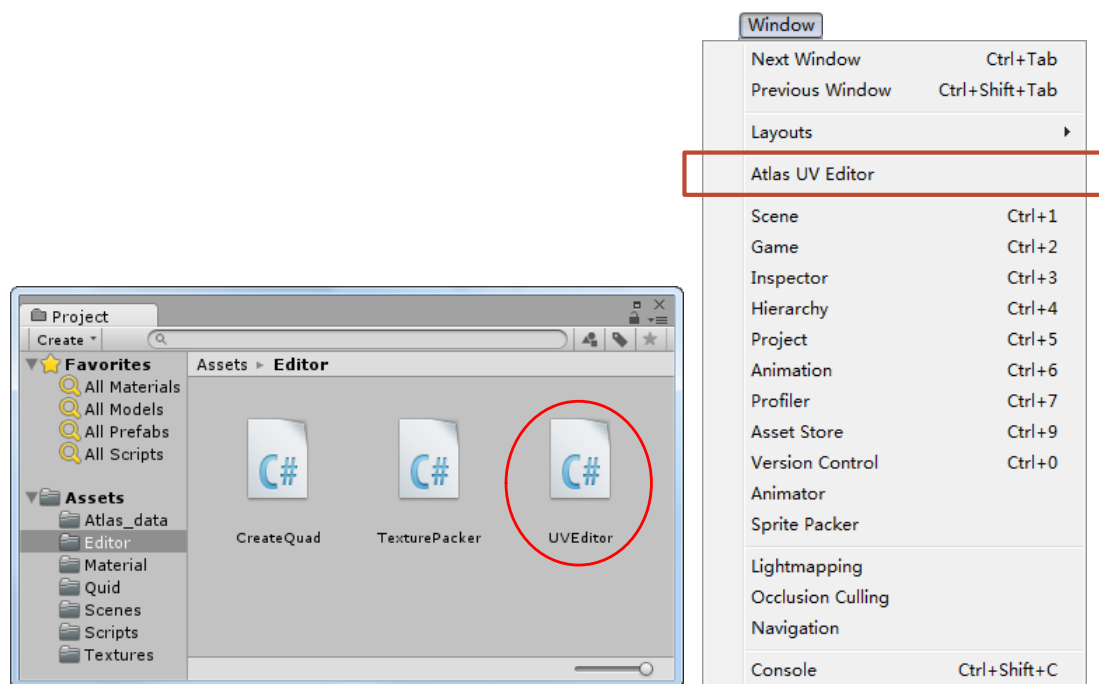


图 10-13 将第 7 章编写的工具，引入到本游戏项目中

打开此工具，设置工具窗口中 Atlas Object 属性为纹理图集预置对象，然后再从场景中选中要修改 UV 值的四边形对象，在工具窗口中选择特定的纹理，如图 10-14 所示。

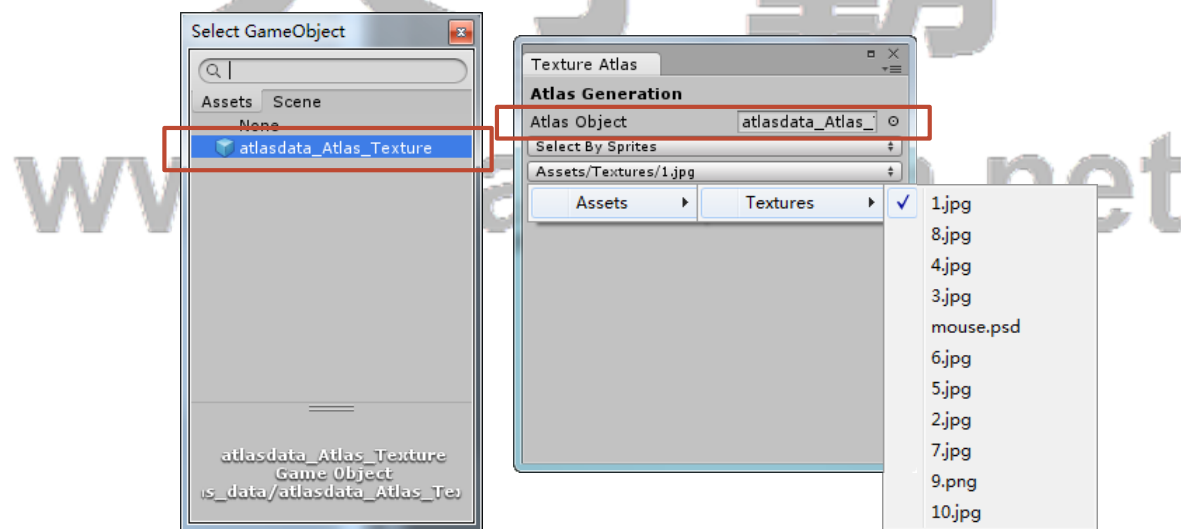


图 10-14 设置工具窗口中的 Atlas Object 属性，以及纹理图集的纹理

场景中所有四边形对象的 UV 值都被设置了以后的效果如图 10-15 所示。

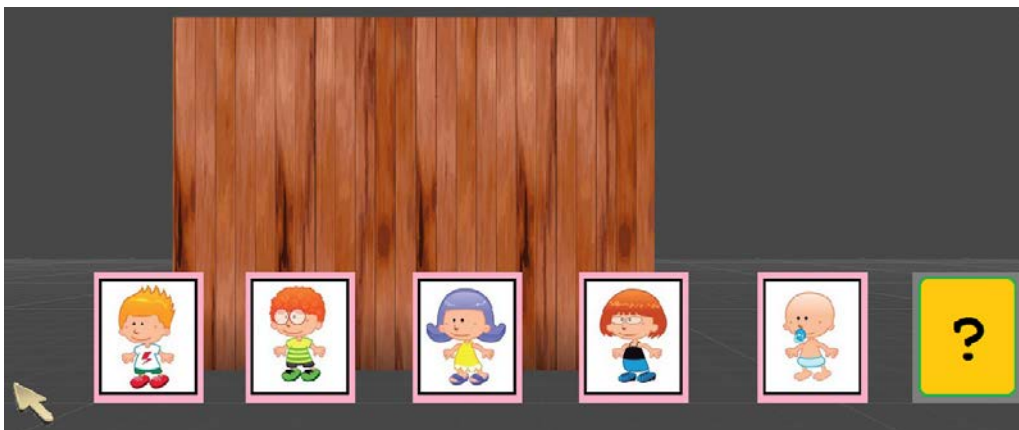


图 10-15 场景中的 8 个游戏对象

### 10.2.5 设置摄像机和游戏视图的分辨率

在第 9 章学习过如何设置摄像机的类型和游戏视图的分辨率，现在是时候应用这些所学的知识了。要求是：

- ❑ 修改游戏视图的分辨率为  $800 \times 600$ ；
- ❑ 修改摄像机的类型为正交摄像机；
- ❑ 调节场景中个游戏对象的位置；

要修改游戏视图分辨率，可以直接单击游戏视图左上角的按钮，然后设置并使用  $800 \times 600$  的分辨率，如图 10-16 所示。

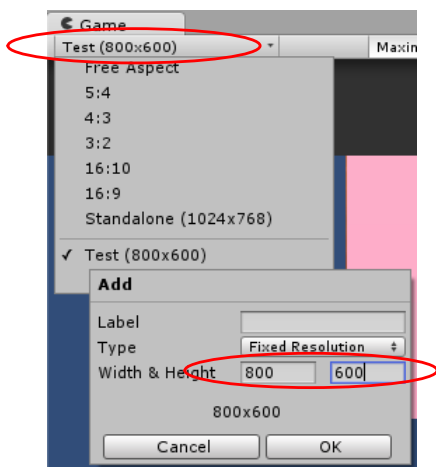


图 10-16 修改游戏视图的分辨率

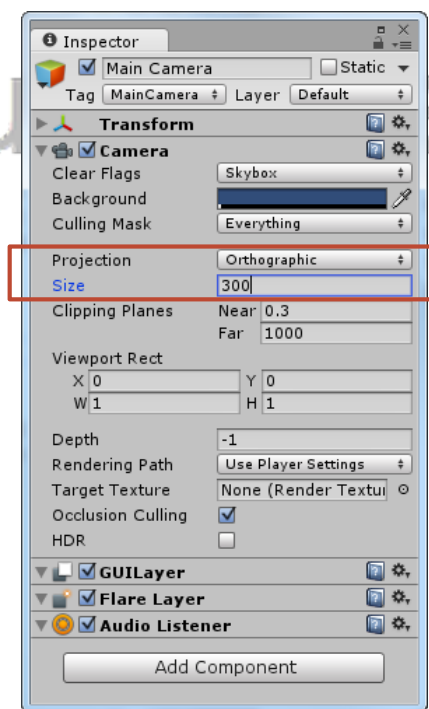


图 10-17 修改摄像机的类型，以及视角的大小

至于摄像机的类型，只需要先选中摄像机，然后在查看器中修改 Camera 组件下的

Projection 属性为 Orthographic 即可。为了让游戏场景三维坐标的长度和屏幕二维坐标的长度有 1:1 的对应关系，还需要设置 Size 属性值为 300（即分辨率中较小值的一半），如图 10-17 所示。

为了保证三维坐标和二维坐标数值上的一致，还需要将两个坐标的原点置于同一位置，如图 10-18 所示，因此需要设置摄像机的位置为(400, 300, Z)（Z 的值随意，但要保证将来可以看到场景中的所有游戏对象）。

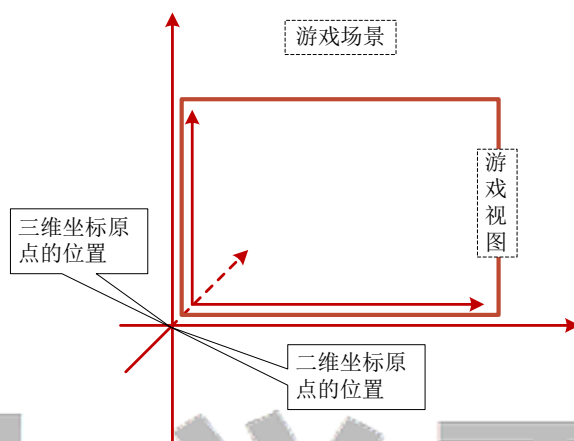


图 10-18 对齐三维坐标和二维坐标的原点

接着调整场景中各四边形对象的位置（至少要让游戏背景四边形与游戏视图对齐），如图 10-19 所示。



图 10-19 调整场景中各游戏对象的位置，以及得到的游戏视图效果

### 10.3 设置场景中的卡片

开发一个 2D 游戏的过程，可以明确的分为两个阶段：

□ 阶段一：创建游戏项目，配置 2D 开发环境，并导入游戏中将要使用的资源；

#### □ 阶段二：实现游戏的各部分功能：

目前，阶段一的任务已经完成了。而阶段二的任务，将由本章剩余的部分，以及本书剩余的章节来逐步完成。此时游戏场景中只有 8 个游戏对象，以后会使用复制的方式添加更多，但是在复制之前最好先对卡片对象做简单的设置。

### 10.3.1 设置卡片的属性

现在的场景中有 5 种卡片类型，每种卡片类型以后会再复制 2 份，达到 15 张卡片！因此现在最好组织一下层次面板中的游戏对象名，以后就不致于太混乱。具体做法是，在场景中创建两个新的游戏对象：root 和 cards。它们都是空对象（Empty），需要单击 GameObject|Create Empty 命令来创建，然后命名，如图 10-20 所示。

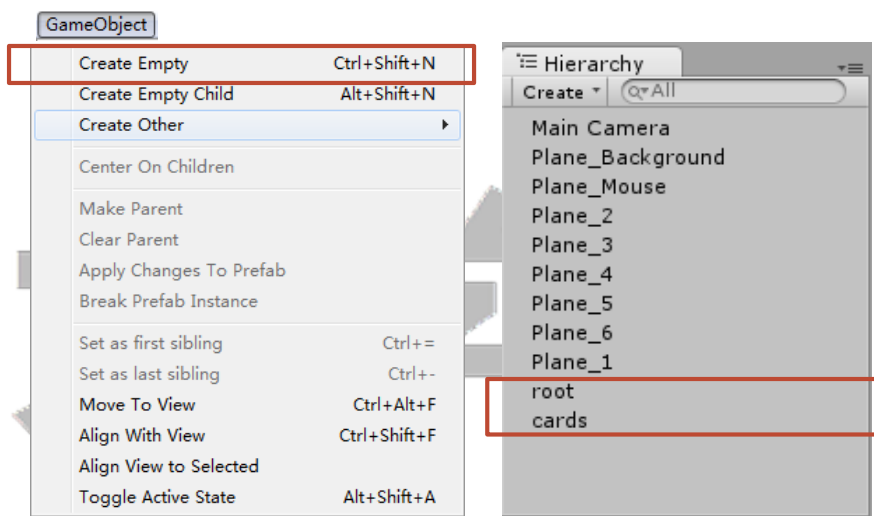


图 10-20 在场景中创建两个空对象：root 和 cards

设置这两个空对象的位置为(0,0,0)，然后让 root 成为其它所有游戏对象的父对象，让 cards 成为其它卡片对象（包括用于遮挡的卡片对象）的父对象。方法很简单，在层次面板中选中相应的游戏对象，然后拖动它们到指定的对象下即可，如图 10-21 所示。

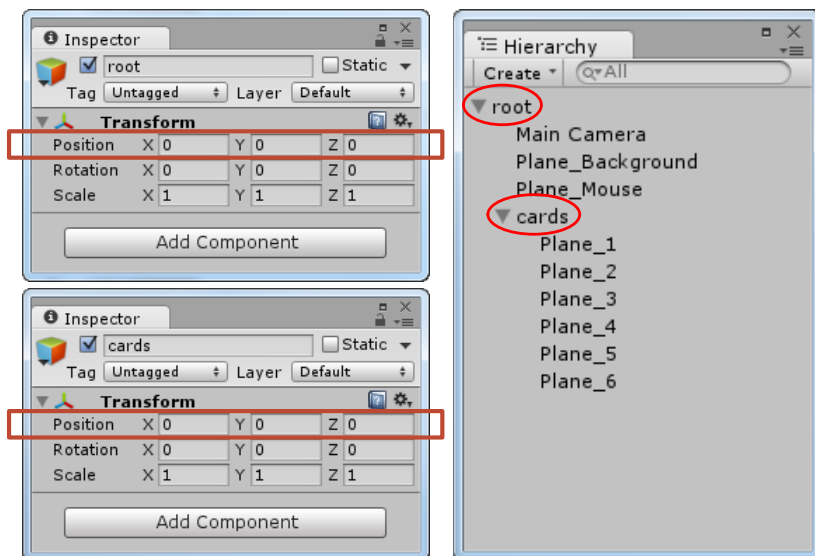




图 10-21 空对象 root 和 card 的位置，及其子对象

提示：建立了父对象与子对象的关系以后，移动父对象时，会同时导致子对象的移动。

为了以后在脚本代码中，可以很方便的访问每个卡片对象，现在最好给 5 种卡片类型起个统一的别名 Card。Unity 允许脚本代码使用别名来得到对象名列表。现在任意选中一个游戏对象，然后在查看器中单击 Tag 下拉列表，选择 Add Tag... 选项，最后在出现的 Tags&Layers 面板中填写一个别名 Card，如图 10-22 所示。

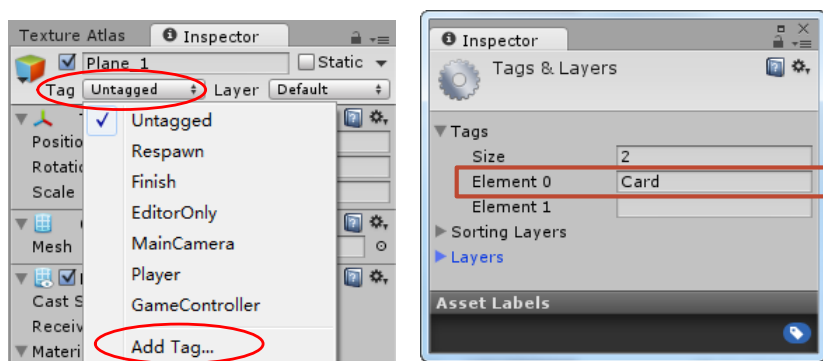


图 10-22 在游戏项目中创建一个别名

接着选中层次面板中的 5 种卡片对象，在查看器中同时设置它们的 Tag 属性为 Card，如图 10-23 所示。

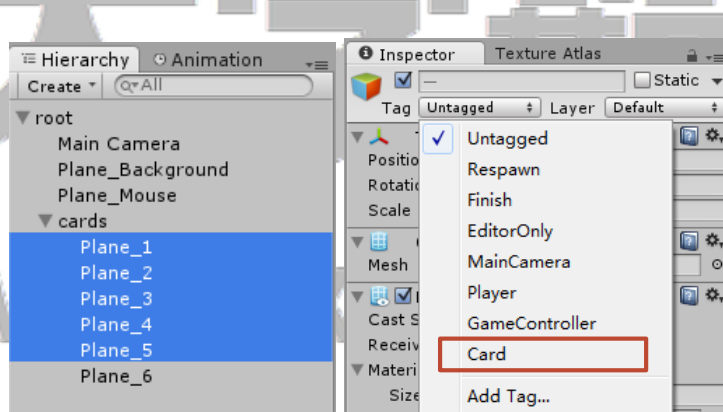


图 10-23 指定 5 种卡片对象的别名

除了已经处理的 5 个卡片对象，还有 1 个卡片对象，它是用于遮挡其它卡片的，现在需要移除这个卡片对象中的 Box Collider 组件，如图 10-24 所示。

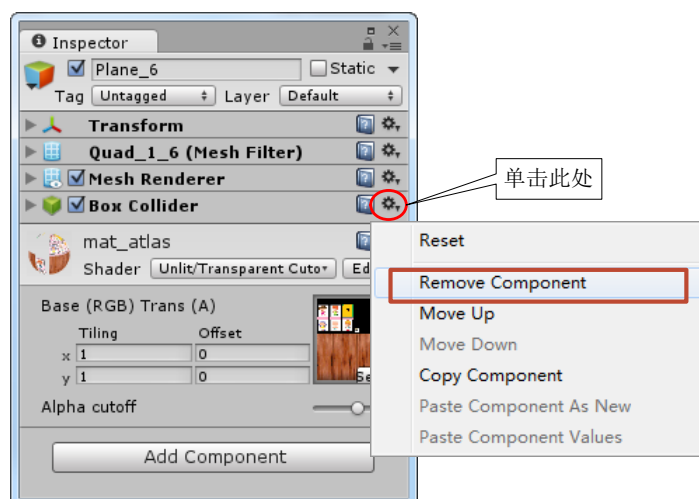


图 10-24 移除 Box Collider 组件

### 10.3.2 定位卡片的位置

将 5 种卡片对象，移动对齐（Y 值相同）到游戏视图的上方，且保证它们处于同一深度（Z 值相同）。各卡片间可以适当留一些空隙，如图 10-25 所示。

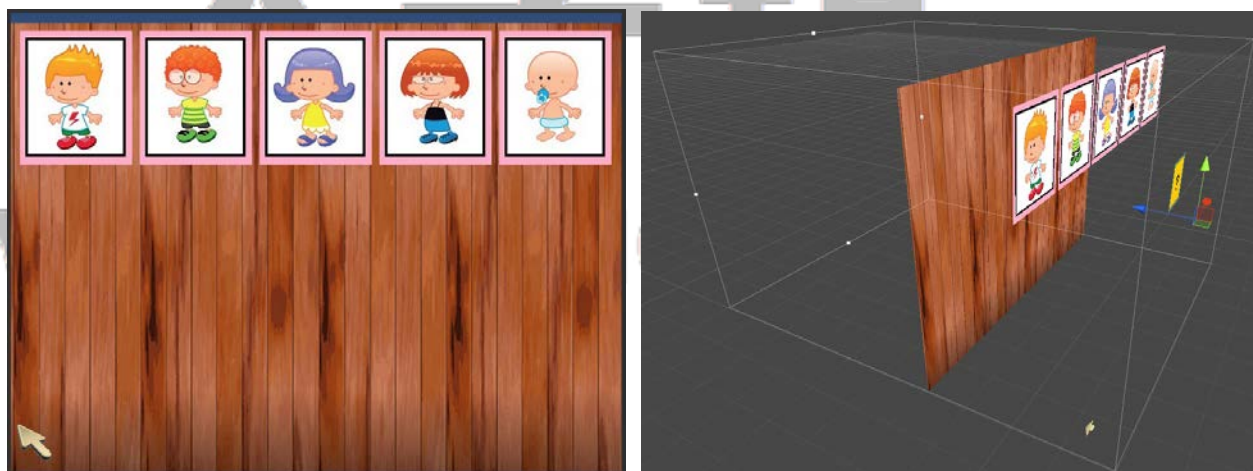


图 10-25 移动对齐 5 种卡片到游戏视图的上方

现在将用于遮挡的卡片拷贝 4 份，也就是总共有 5 个游戏对象，将被用于遮挡 5 种类型的卡片。然后设置它们为 5 种卡片类型的子对象，如图 10-26 所示。

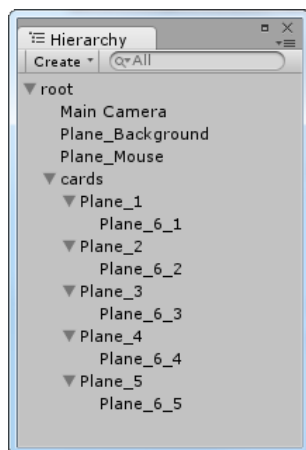


图 10-26 设置遮挡卡片为各卡片的子对象，并重新命名

设置遮挡卡片的 Z 值，使其更靠近于摄像机，如图 10-27 所示。

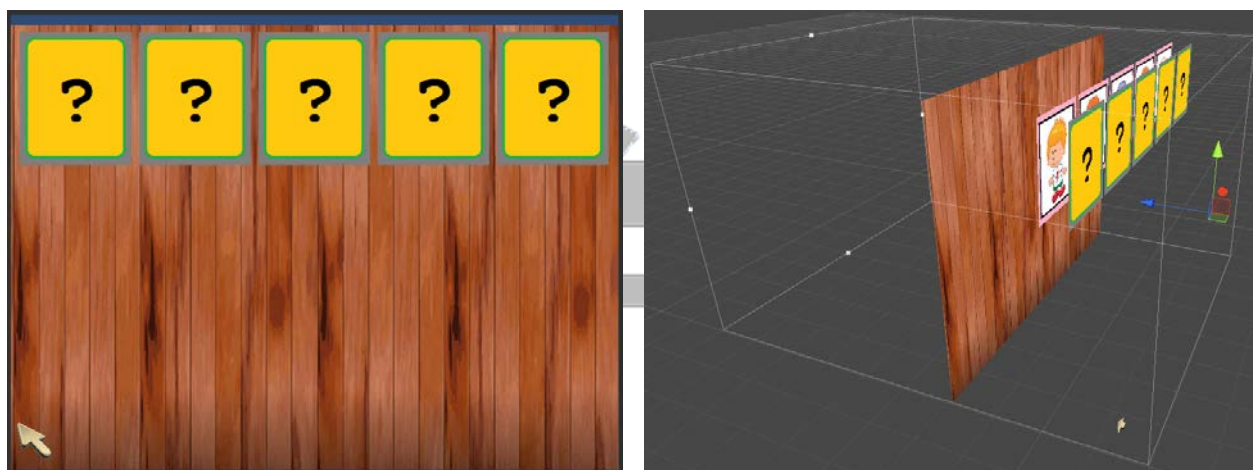


图 10-27 5 个子卡片对象将分别用于遮挡各自的父对象

### 10.3.3 编写控制卡片行为的脚本

很明显，游戏的雏形已经有了，接下来是要拷贝两行卡片到游戏视图的下方吗？在此之前最好先为每个卡片添加脚本代码，用于控制它们在游戏时的行为，然后再拷贝。否则就不得不手动为每个卡片添加脚本组件了。

在资源面板的 **Scripts** 文件夹下，创建一个 C# 脚本，并命名为 **Card**。在编写脚本前需要明确，每张卡片在游戏过程中，一定是处于下面三种状态中的一种的：

- ☐ 被遮挡（Concealed）。卡片在游戏时的默认状态。处于这种状态的原因是，它被遮挡卡片遮挡住了，而玩家也还没有点击这张卡片。
- ☐ 显示（Revealed）。处于这种状态的卡片，玩家可以看到它上面的图片。玩家在游戏过程中点击了这张卡片，于是遮挡卡片被移开了。
- ☐ 隐藏（Hidden）。玩家成功的在一轮中连续找到 3 张相同的卡片，于是被找到的卡片会在游戏中移除，或者说是被隐藏了。

这些卡片状态将使用 Card 脚本代码实现，如下：

```

01 using UnityEngine;
02 using System.Collections;
03
04 public class Card : MonoBehaviour
05 {
06     //卡片的三种状态
07     public enum CARD_STATE {CARD_CONCEALED = 0,
08                             CARD_REVEALED = 1, CARD_HIDDEN = 2};
09     //卡片的默认状态
10     public CARD_STATE ActiveState = CARD_STATE.CARD_CONCEALED;
11     //卡片和遮挡卡片的深度 - (X = 卡片的 Z 值, Y = 遮挡卡片的 Z 值)
12     public Vector2[] ZPosStates = new Vector2[3];
13     //用于移动卡片的变量
14     private Transform CardTransform = null;
15     //用于移动遮挡卡片的变量
16     private Transform CoverTransform = null;
17     //在游戏对象被初始化时调用，调用时机早于 Start()函数
18     void Awake()
19     {
20         //获取卡片的位置
21         CardTransform = transform;
22         //获取遮挡卡片的位置
23         CoverTransform = CardTransform.GetChild(0).transform;
24     }
25     //用于设置不同状态下，卡片的位置
26     public void SetCardState(CARD_STATE State = CARD_STATE.CARD_CONCEALED)
27     {
28         //改变卡片的状态
29         ActiveState = State;
30         //改变卡片和遮挡卡片的位置
31         CardTransform.localPosition = new Vector3(CardTransform.localPosition.x,
32                                                     CardTransform.localPosition.y,
33                                                     ZposStates[(int)ActiveState].x);
34         CoverTransform.localPosition = new Vector3(CoverTransform.localPosition.x,
35                                                     CoverTransform.localPosition.y,
36                                                     ZPosStates[(int)ActiveState].y);
37     }
38 }

```

脚本使用 10 行定义的 ActiveState 枚举变量，表示当前卡片的状态；26 行定义的函数 SetCardState()用于设置不同状态下卡片在 Z 轴方向上的位置。确保将 C#脚本 Card 应用于场景中的 5 类卡片对象。选中其中任意卡片对象，在查看器中打开对应的脚本组件，如图 10-28 所示。

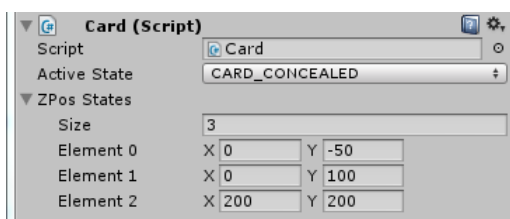


图 10-28 应用于卡片对象上的脚本组件

脚本组件的 **Active State** 属性用于设置卡片当前的状态，默认选中的是遮挡状态；**ZPos States** 属性用于设置卡片和遮挡卡片在 3 种状态时，Z 轴上的值。**Element 0** 表示遮挡状态，**Element 1** 表示显示状态，**Element 2** 表示隐藏状态；**X** 的值表示卡片在 Z 轴方向上的值，**Z** 的值表示遮挡卡片在 Z 轴方向上的值，如图 10-29，图 10-30 和图 10-31 所示。

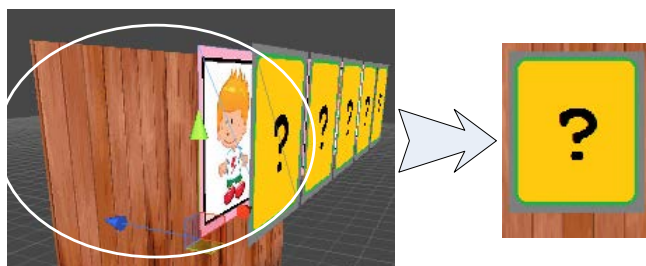


图 10-29 卡片和遮挡卡片的状态——遮挡

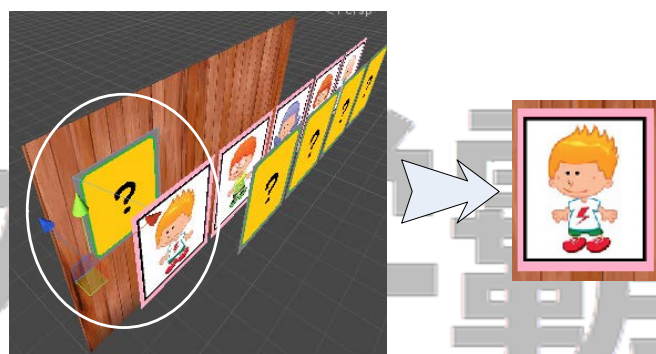


图 10-30 卡片和遮挡卡片的状态——显示

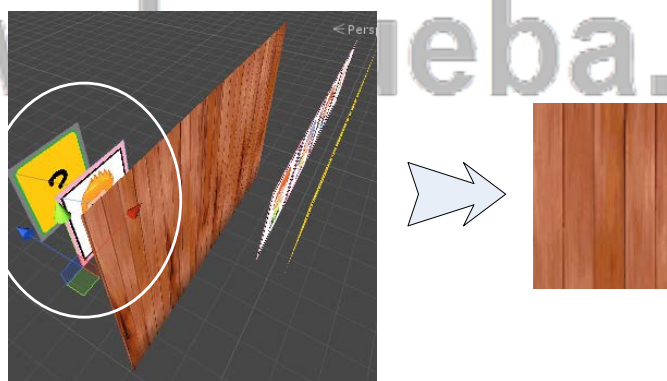




图 10-31 卡片和遮挡卡片的状态——隐藏

要做到以上的效果，需要读者依据游戏背景、卡片和遮挡卡片的位置，自己指定 **ZPos States** 属性的 3 个状态值。其余 4 组卡片和遮挡卡片也需要同相同的设置。

提示：如果觉着为其余 4 组卡片和遮挡卡片，设置脚本中的属性为相同的值太麻烦，可以单击已经设置好值的组件右边的  选项，选择 **Copy Component**。然后，单击还未设置好值的组件右边的  选项，选择 **Paste Component Values**，即完成数据的复制，如图 10-32 所示。



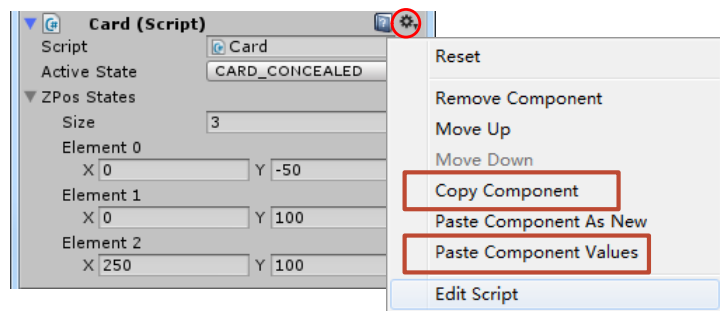


图 10-32 复制、粘贴组件各属性的值

### 10.3.4 补全场景中其余的卡片

现在就可以补全游戏视图中的其余两行卡片了。可以先在层次面板中，选中 5 个卡片对象，然后复制两次，设置它们的位置，并合理命名，如图 10-33 所示。



图 10-33 补全游戏视图中的卡片

## 10.4 游戏管理类

在软件设计领域，游戏管理类（Game Manager class）在游戏中被广泛使用。它本身只是一个游戏对象，却管理着游戏的规则；它真实存在于场景中，但是却不可见、不可触摸；它使得场景中的对象相互协作。

而在本游戏中，它管理的规则涉及到：

- ❑ 重置卡片。在游戏开始时，将卡片的顺序整乱，确保玩家每次玩游戏时，卡片的顺序都不一样。
- ❑ 处理玩家输入。检测玩家的点击是否有效，即检测玩家所点击的卡片是否匹配或者

一致。

- ❑ 响应玩家输入。玩家点击操作后的响应，如果玩家点击的 3 张卡片图像一致，则将这 3 张卡片从游戏视图中移除；如果卡片图像不一致，重新开始这一轮，并恢复卡片的遮挡状态。

要在游戏项目中添加游戏管理类，需要在资源面板的 **Scripts** 文件夹下创建名为 **GameManager** 的 C# 脚本文件，然后将此脚本文件赋予场景中的 **root** 对象，如图 10-34 所示。

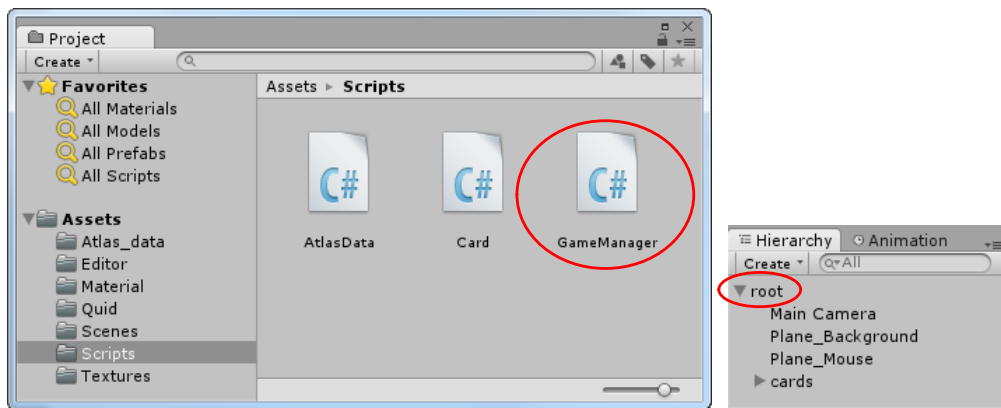


图 10-34 创建 C# 脚本，写完代码后赋予 root 对象

提示：此脚本可以赋予任何对象，并不一定非得是 **root** 对象。但被赋予的对象一定是在游戏开始的时候就被创建了，在场景中始终保持激活状态，直到场景被终止。

#### 10.4.1 重置卡片

重置过程涉及生成随机数的问题。在本游戏中就是改变游戏视图中卡片的排序，玩家将无法预测特定卡片的位置。要生成随机数，可以使用 Unity 的 **Random** 类。下面的脚本代码是使用 **Random** 类，编写的函数 **Shuffle()**：

```
01 public void Shuffle()
02 {
03     //公有成员 CardObjects 是场景中所有卡片对象的列表
04     //此列表是通过下面的方式获取到的：
05     //CardObjects = GameObject.FindGameObjectsWithTag("Card") as GameObject[];
06     foreach(GameObject CardObject in CardObjects)
07         CardObject.SendMessage("SetCardState",
Card.CARD_STATE.CARD_CONCEALED,
08             SendMessageOptions.DontRequireReceiver);
09     //遍历所有的卡片对象，并交换它们的位置
10     foreach(GameObject Card in CardObjects)
11     {
12         //获取一个卡片的位置
13         Transform CardTransform = Card.transform;
14         //获取另一个随机取得的卡片的位置
15         Transform ExchangeTransform = CardObjects[Mathf.FloorToInt(Random.Range(0,
CardObjects.Length-1))].transform;
16
17         //交换两张卡片的位置
18         Vector3 TmpPosition = CardTransform.localPosition;
19         CardTransform.localPosition = ExchangeTransform.localPosition;
```

```
20         ExchangeTransform.localPosition = TmpPosition;
21     }
22 }
```

Shuffle()函数将遍历游戏中的 15 张卡片，并与随机找到的另一张卡片交换位置。随机找到的另一张卡片，使用了 Random 类的 Random.Range()函数，如下：

```
Mathf.FloorToInt(Random.Range(0, CardObjects.Length-1));
```

此函数会从 0~ CardObjects.Length-1 这个范围内随机选择一个浮点数，再经由 Mathf.FloorToInt()函数转换成整数，就得到了随机选择的另一张卡片的位置。

注意：“另一张卡片”也可能是卡片本身，也就是说卡片将和自己交换位置，卡片的位置将不会发生改变。

程序的 18~20 行，是使用 localPosition 变量来设置卡片的位置的；其实还有一个变量 position 也可以用于设置卡片的位置；那么它们有什么不同呢？简单来说就是，后者设置的是对象在世界坐标系中的位置（即**绝对位置**），而前者设置的是对象相对于父对象的位置（即**相对位置**）。

## 10.4.2 处理玩家输入

这里所说的玩家输入，只包括玩家点击卡片时的输入。此输入，既可以是玩家鼠标的点击过程，又可以是玩家触摸移动设备屏幕的过程。下面是游戏管理 C#脚本中处理玩家输入的代码：

```
01 public void HandleInput()
02 {
03     //忽略无效的输入操作
04     if(!InputEnabled) return;
05     //当前玩家没有任何输入
06     if(!Input.GetMouseButtonDown(0) && Input.touchCount <= 0)
07         return;
08     //玩家触摸屏幕时的位置
09     Vector3 TapPosition = Vector3.zero;
10     //处理电脑上的鼠标输入
11     #if UNITY_STANDALONE || UNITY_WEBPLAYER
12         TapPosition = Input.mousePosition;
13     #endif
14     //处理移动设备上的输入
15     #if UNITY_IPHONE || UNITY_ANDROID || UNITY_WP8
16         Touch T = Input.GetTouch(0);
17         if(T.phase == TouchPhase.Began)
18             TapPosition = T.position;
19     else
20         return;
21     #endif
22     //在点击或者触摸的位置生成一束激光
23     Ray R = Camera.main.ScreenPointToRay(TapPosition);
24     //检测与激光相交的游戏对象，进而得到玩家此时点击的卡片
25     RaycastHit[] Hits;
26     Hits = Physics.RaycastAll(R);
27     //遍历所有与激光相交的对象
28     foreach(RaycastHit H in Hits)
```

```

29     {
30         PickCard (H.collider.gameObject);
31         UpdateTurn(H.collider.gameObject);
32         return;
33     }
34 }

```

此段代码中的大部分涉及玩家输入的部分，已经在第 9 章接触过了，不再重复说明。代码 31 行，UpdateTurn()函数将用于检测，玩家此时选中的卡片与前面选中的卡片是否匹配。

### 10.4.3 响应玩家输入

玩家在游戏过程中，每一轮的第一次可以随机选择一张卡片，第二、三次就要检测选中的卡片的图像是否与第一次选中的卡片一致。一轮结束可能是由于匹配失败，也可能是由于匹配成功。游戏管理类主要使用两个函数：StartTurn()和 UpdateTurn()，管理每轮对玩家输入的响应。脚本代码如下：

```

01 //游戏开始了新一轮时调用此函数
02 public void StartTurn()
03 {
04     //表示此轮第几次点击的变量
05     TurnStep = 0;
06     //清空数组
07     for(int i = 0; i<SelectedCards.Length; i++)
08         SelectedCards[i]=null;
09 }
10 public void UpdateTurn(GameObject PickedCard)
11 {
12     //添加卡片到“已被选中”数组中
13     SelectedCards[TurnStep] = PickedCard;
14     //进行下一次点击
15     ++TurnStep;
16     //判断是否应该进入下一轮
17     if(TurnStep <= 1)
18         return;
19     //如果此轮已经选中了两张卡片，就判断这两种卡片的图片是否一致
20     if(SelectedCards[TurnStep-1].GetComponent<MeshFilter>().mesh.name !=
21        SelectedCards[TurnStep-2].GetComponent<MeshFilter>().mesh.name)
22     {
23         //如果不一致，重新开始下一轮
24         StartCoroutine(LoseTurn());
25         return;
26     }
27     //玩家当前选择的两种卡片图片一致，则判断是否进入下一轮，即是否当前已选中了 3 张卡片
28     if(TurnStep >= NumMatches)
29     {
30         //玩家已经选择了 3 张卡片，图片一致，移除这 3 张卡片，进入下一轮
31         StartCoroutine(WinTurn());
32     }
33 }

```

函数 `StartTurn()`，在游戏新一轮开始时，用于重置内部的两个变量：`TurnStep` 和 `SelectedCards`。前者的数值范围是 0~3，0 表示当前玩家没有选择任何卡片，1 表示玩家当前已经选择了一张卡片，2 和 3 依次类推。数组 `SelectedCards` 包含 3 个元素，用于存放玩家选中的 3 张卡片。判断 2 张卡片的图像是否一致的方法是，检测选中的两张卡片是否是基于同样的四边形资源生成的。

#### 10.4.4 游戏管理类代码汇总

C#脚本 `GameManager` 在游戏启动时，需要获取当前游戏场景中的所有卡片对象，以及场景中的鼠标对象，依据的是场景中对象的别名。卡片对象的别名是 `Card`，鼠标对象的别名还没有设置，可以用同样的方法设置为 `Cursor`，如图 10-35 所示。

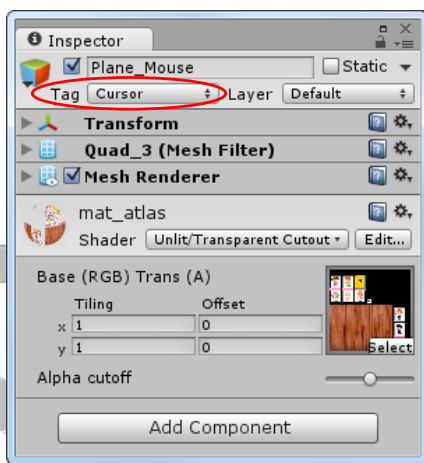


图 10-35 场景中鼠标对象的别名

在游戏运行的过程中，游戏管理类应该在每帧都检测一下，玩家是否点击了卡片，也就是说响应玩家的输入过程，应该放在 `Update()` 函数中。前面各小节的函数中，还涉及到很多未定义的变量，而它们应该根据需求被定义为脚本类的公有或者私有成员。综上所述，脚本 `GameManager` 中的代码编写如下：

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class GameManager : MonoBehaviour
05 {
06     //-----公有成员-----
07     //一轮失败，开始新一轮的时间间隔
08     public float LoseInterval = 1.0f;
09     //一轮获胜，开始新一轮的时间间隔
10     public float WinInterval = 1.0f;
11
12     //-----私有成员-----
13     //每种类型的卡片数目
14     private int NumMatches = 3;
15     //存储场景中的所有卡片对象
16     private GameObject[] CardObjects = null;
17     //记录每轮游戏，玩家已经点击了多少卡片
```



```

18     private int TurnStep = 0;
19     //每轮玩家选中的卡片
20     private GameObject[] SelectedCards = null;
21     private bool InputEnabled = true;
22     private MeshRenderer CursorRender = null;
23
24     void Start()
25     {
26         //获取场景中的所有卡片对象
27         CardObjects = GameObject.FindGameObjectsWithTag("Card") as
28         GameObject[];
29         //获取鼠标绘制组件
30         CursorRender =
31         GameObject.FindGameObjectWithTag("Cursor").GetComponent<MeshRenderer>();
32         SelectedCards = new GameObject[NumMatches];
33         Shuffle();
34     }
35     void Update()
36     {
37         HandleInput();
38     }
39     //重置卡片
40     public void Shuffle()
41     {
42         foreach(GameObject CardObject in CardObjects)
43             CardObject.SendMessage("SetCardState",
44             Card.CARD_STATE.CARD_CONCEALED,
45             SendMessageOptions.DontRequireReceiver);
46         //遍历所有的卡片对象，并交换它们的位置
47         foreach(GameObject Card in CardObjects)
48         {
49             //获取一个卡片的位置
50             Transform CardTransform = Card.transform;
51             //获取另一个随机取得的卡片的位置
52             Transform ExchangeTransform =
53             CardObjects[Mathf.FloorToInt(Random.Range(0,
54             CardObjects.Length-1))].transform;
55             //交换两张卡片的位置
56             Vector3 TmpPosition = CardTransform.localPosition;
57             CardTransform.localPosition = ExchangeTransform.localPosition;
58             ExchangeTransform.localPosition = TmpPosition;
59         }
60     }
61     //处理玩家输入
62     public void HandleInput()
63     {
64         //忽略无效的输入操作
65         if(!InputEnabled)
66             return;

```

```
65         //当前玩家没有任何输入
66         if(!Input.GetMouseButtonDown(0) && Input.touchCount <= 0)
67             return;
68         //玩家触摸屏幕时的位置
69         Vector3 TapPosition = Vector3.zero;
70         //处理电脑上的鼠标输入
71         #if UNITY_STANDALONE || UNITY_WEBPLAYER
72             TapPosition = Input.mousePosition;
73         #endif
74         //处理移动设备上的输入
75         #if UNITY_IPHONE || UNITY_ANDROID || UNITY_WP8
76             Touch T = Input.GetTouch(0);
77             if(T.phase == TouchPhase.Began)
78                 TapPosition = T.position;
79             else
80                 return;
81         #endif
82         //在点击或者触摸的位置生成一束激光
83         Ray R = Camera.main.ScreenPointToRay(TapPosition);
84         //检测与激光相交的游戏对象，进而得到玩家此时点击的卡片
85         RaycastHit[] Hits;
86         Hits = Physics.RaycastAll(R);
87         //遍历所有与激光相交的对象
88         foreach(RaycastHit H in Hits)
89         {
90             PickCard (H.collider.gameObject);
91             UpdateTurn(H.collider.gameObject);
92             return;
93         }
94     }
95     public void EnableInput(bool bEnabled = true)
96     {
97         CursorRender.enabled = InputEnabled = bEnabled;
98     }
99     public void PickCard(GameObject SelectedCard)
100    {
101        SelectedCard.SendMessage("SetCardState",
102        Card.CARD_STATE.CARD_REVEALED,
103        SendMessageOptions.DontRequireReceiver);
104    }
105    //游戏开始了新一轮时调用此函数
106    public void StartTurn()
107    {
108        //表示此轮第几次点击的变量
109        TurnStep = 0;
110        //清空数组
111        for(int i = 0; i<SelectedCards.Length; i++)
112            SelectedCards[i]=null;
113    }
114    public void UpdateTurn(GameObject PickedCard)
115    {
116        //添加卡片到“已被选中”数组中
```

```

116     SelectedCards[TurnStep] = PickedCard;
117     //进行下一次点击
118     ++TurnStep;
119     //判断是否应该进入下一轮
120     if(TurnStep <= 1)
121         return;
122     //如果此轮已经选中了两张卡片，就判断这两种卡片的图片是否一致
123     if(SelectedCards[TurnStep-1].GetComponent<MeshFilter>().mesh.name !=
124         SelectedCards[TurnStep-2].GetComponent<MeshFilter>().mesh.name)
125     {
126         //如果不一致，重新开始下一轮
127         StartCoroutine(LoseTurn());
128         return;
129     }
130     //玩家当前选择的两种卡片图片一致，则判断是否进入下一轮，即是否当前已选中了 3
张卡片
131     if(TurnStep >= NumMatches)
132     {
133         //玩家已经选择了 3 张卡片，图片一致，移除这 3 张卡片，进入下一轮
134         StartCoroutine(WinTurn());
135     }
136 }
137 //本轮点击的卡片图像不一致时，调用此函数
138 public IEnumerator LoseTurn()
139 {
140     //禁止输入操作
141     EnableInput(false);
142     //等待一定的时间后，开始新一轮
143     yield return new WaitForSeconds(LoseInterval);
144     //重新遮挡当前轮中显示的卡片
145     for(int i=0; i<SelectedCards.Length; i++)
146     {
147         //遍历每个被选中的卡片，设置为遮挡状态
148         if(SelectedCards[i])
149             SelectedCards[i].SendMessage(
150                 "SetCardState", Card.CARD_STATE.CARD_CONCEALED,
151                 SendMessageOptions.DontRequireReceiver);
152     }
153     //开始新一轮
154     StartTurn();
155     //输入操作可用
156     EnableInput(true);
157 }
158 //本轮点击的 3 张卡片图像一致时，调用此函数
159 public IEnumerator WinTurn()
160 {
161     //禁止输入操作
162     EnableInput(false);
163     //等待一定的时间后，开始新一轮
164     yield return new WaitForSeconds(WinInterval);
165     //设置选中的 3 张卡片为隐藏模式
166     for(int i=0; i<SelectedCards.Length; i++)

```

```

167     {
168         if(SelectedCards[i])
169             SelectedCards[i].SendMessage(
170                 "SetCardState", Card.CARD_STATE.CARD_HIDDEN,
171                 SendMessageOptions.DontRequireReceiver);
172     }
173     //检查当前游戏视图中，是否还有卡片
174     bool CardsRemaining = false;
175     foreach(GameObject CardObj in CardObjects)
176     {
177         if(CardObj.GetComponent<Card>().ActiveState !=
178             Card.CARD_STATE.CARD_HIDDEN)
179         {
180             CardsRemaining = true;
181             break;
182         }
183     }
184     if(CardsRemaining)
185     {
186         //进入下一轮
187         StartTurn();
188     }
189     else
190     {
191         Shuffle();
192         StartTurn();
193     }
194     EnableInput(true);
195 }

```

将此脚本赋予 root 对象，称为后者的组件，如图 10-36 所示。2 个属性分别用于设置玩家赢一轮，或者输一轮时，进入下一轮时的时间间隔。

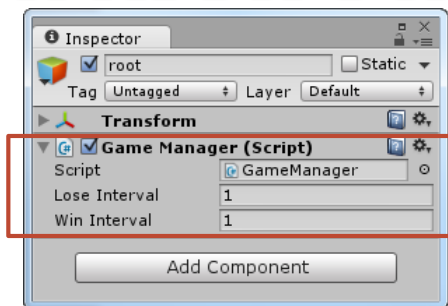


图 10-36 root 对象的组件

## 10.5 完善并运行游戏

目前为止，游戏的全部行为逻辑已经完成了。美中不足的是，我们准备了用于替换系统鼠标的图标，但现在还没有使用。第 9 章学习过，如何使用自定义的鼠标图标修改系统的鼠

标图标，现在就是应用所学的知识的时候了。完成鼠标图标的替换，就可以运行体验游戏了。

### 10.5.1 替换系统鼠标图标

在第 9 章中，图标图标的移动控制可以使用键盘或者鼠标来完成，而本游戏将设置为只能用鼠标移动。在游戏项目的资源面板的 Scripts 文件夹下，添加名为 Cursor 的 C#脚本，脚本中的代码如下：

```

01 using UnityEngine;
02 using System.Collections;
03
04 public class Cursor : MonoBehaviour
05 {
06     //自定义鼠标纹理位置的偏移量
07     public Vector2 CursorOffset = Vector2.zero;
08     //是否显示系统的鼠标
09     public bool ShowSystemCursor = false;
10     //用于实时改变纹理位置的对象
11     private Transform ThisTransform = null;
12     void Start ()
13     {
14         //Cache transform
15         ThisTransform = transform;
16     }
17     void Update ()
18     {
19         #if UNITY_STANDALONE || UNITY_EDITOR || UNITY_WEBPLAYER ||
UNITY_DASHBOARD_WIDGET
20
21         //更新显示鼠标的位置
22         ThisTransform.position = new Vector3( Input.mousePosition.x + CursorOffset.x,
23                                             Input.mousePosition.y + CursorOffset.y,
24                                             ThisTransform.position.z);
25     #endif
26     //显示还是隐藏系统鼠标
27     Screen.showCursor = ShowSystemCursor;
28 }
29 }

```

将此脚本赋予到场景中的鼠标对象上，即可使用自定义的鼠标图标，替换系统的鼠标图标。

提示：要记得移除鼠标对象上的 Box Collider 组件，避免触发鼠标与激光的相交事件，详细内容见第 9 章。同理，还要移除游戏背景上的 Box Collider 组件。

### 10.5.2 游戏运行效果展示

启动游戏后，视图中的所有卡片都处在被遮挡的状态，如图 10-37 所示。



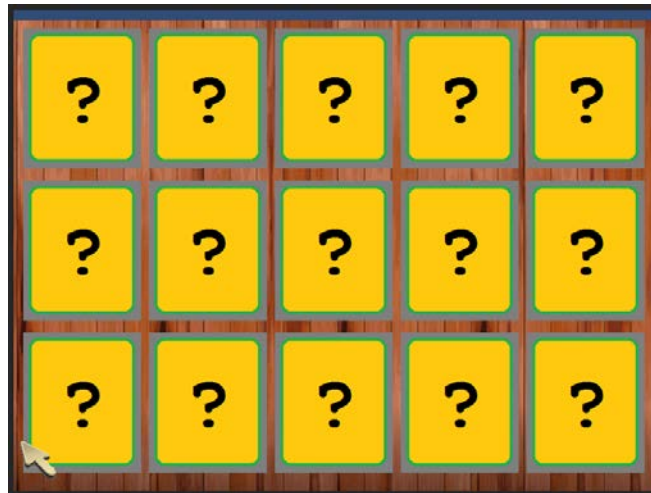


图 10-37 游戏的起始视图

当用鼠标点击一张卡片时，卡片的图片会显示出来，而一轮游戏也就开始了，点击第二张卡片的图像与第一张不一致时，两张卡片会恢复被遮挡的状态，一轮结束，如图 10-38 所示。

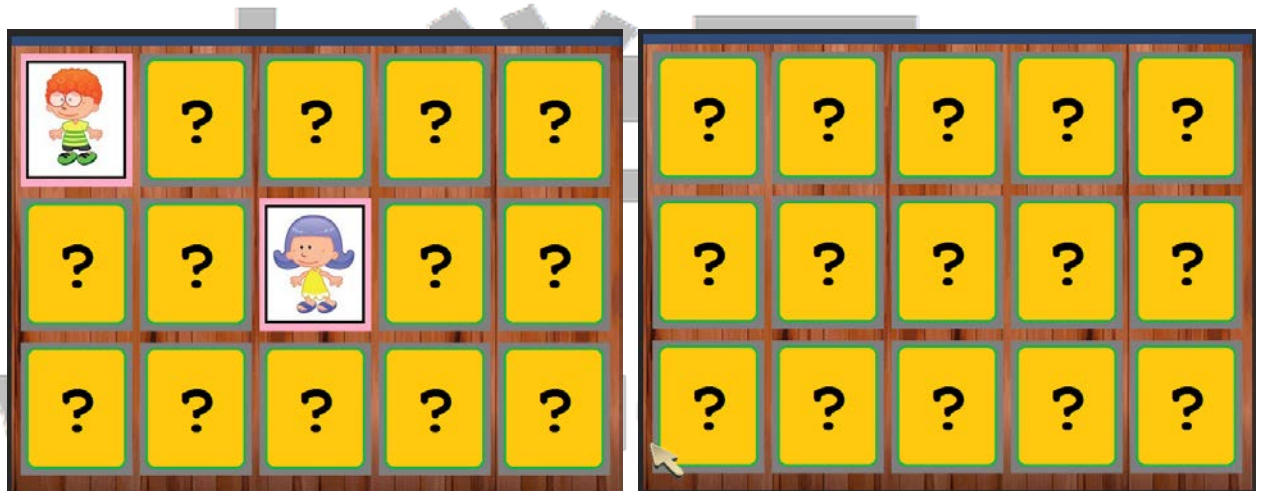


图 10-38 一轮游戏中，点击的卡片图像不一致，两张卡片都恢复被遮挡的状态

当一轮中，点击的三张卡片全部一致时，三张卡片将被移除，一轮结束，如图 10-39 所示。

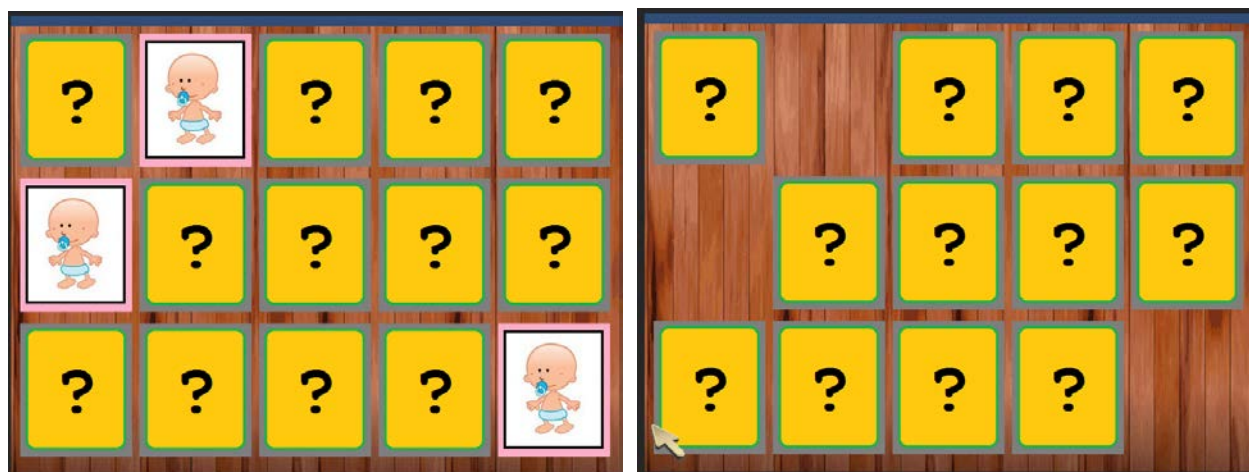


图 10-39 一轮游戏中，点击的卡片图像一致，三张卡片被从游戏视图中移除

看起来游戏运行的不错！确实，看起来是这样没错，但是由于代码的逻辑不是很完善，所以不可避免的会有一些 Bug。例如，当玩家始终点击一张卡片时，游戏会以为点击的是多张相同的卡片，于是在卡片被点击 3 次以后，也会消失！消除这个 Bug 的方法是，禁用被点击的卡片的 Box Collider 组件，这样的话，玩家单击相同的卡片时，游戏系统是不会检测到这张卡片的。这个 Bug 的修改希望读者自己来完成。

www.daxueba.net