游戏架构
核心技术与面试精粹

樊松阳 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从架构与封装、艺术资源、底层核心、自定义扩展和独立游戏五个方向组织内容，每一方向均围绕一个问题展开论述，重点讲解了 Unity 3D 开发的高级技巧。这种问答的形式，既有助于读者扩展思考，又可用于面试题。本书除了涉及代码架构和引擎底层原理，还包含了美术资源管理、艺术效果制作、工作流程优化等，通过解析这些实际开发过程中遇到的问题，可以更全面地提升读者的知识储备。书中的大部分章节都附有示例代码，聚焦具体的知识点，让读者知其然并知其所以然。

本书适用于希望技术进阶的 Unity 3D 开发者、独立游戏开发者，或有初级经验的游戏从业者。

图书在版编目（CIP）数据

游戏架构：核心技术与面试精粹 / 樊松阳著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34263-9

Ⅰ. ①游… Ⅱ. ①樊… Ⅲ. ①游戏－软件设计 Ⅳ.①TP311.5

中国版本图书馆 CIP 数据核字(2018)第 109387 号

责任编辑：安　娜
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：16.25　字数：309.6 千字　彩插：2
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

这是一本面向游戏开发者的进阶指南。

本书从日常开发遇到的问题入手，以编程思想为线索，探讨合理的解决方案。在技术方面，本书会结合使用广泛的 Unity 3D 进行讲解。虽然本书中的很多内容都不会被引擎所桎梏，但从某种意义上讲，这确实是一本对 Unity 3D 使用者有更大价值的书。

为了更便于查找，本书内容组织上采用问答的形式。提出的问题通常都是开放性的，随着对底层原理认识的加深，读者自己也能答出更多有料的内容。当然，作者也会被自身能力限制，所以答案是探讨式的，以期逐步深入地挖掘原理。

本书整体通过以下五个方向组织内容：架构与封装、艺术资源、底层核心、自定义扩展和独立游戏。从层次上看是从应用层、效果层、引擎层、工具层到职能扩展层的顺序论述。大体看来，是从日常技能逐渐过渡到核心能力的讲解顺序。

每个方向围绕一个问题展开论述，有的内容会侧重概念的讲解，大多数情况下会结合实际代码来说明。通过这样的方式，尽量让读者知其然并知其所以然。很多小结后面会有延伸思考，我也希望读者能根据问题深入探讨，这些问题大多附有解答思路，希望能帮助大家养成勤于思考的习惯。

对于游戏来说，技术只是其中的重要部分，但我更将其看作技术与艺术的结合体，因此美术方面的知识、游戏玩法的设计也不可或缺。本书在编排的过程中，我尽量兼顾上述要点，以期有兴趣的新手读者，从业人员，或者是独立开发者，都能从本书中得到些收获。另外，对于希望进入游戏界的读者，本书可以从全局的角度，帮助你理解游戏开发的架构流程。

本书中会将一些网址，或者一些扩展问题的解答放到我的微信公众号中，通过在公众号中回复相应的关键字来获得自动回复。采用这种做法主要基于以下原因：

◎ 链接很容易更新或失效。
◎ 手动输入链接比较麻烦，通过微信电脑端可以直接在 PC 上打开有效的链接。
◎ 有链接形式的引用知识，方便跳转阅读。
◎ 对照问题答案时，避免来回翻页。

在本书中，我会以[:回复关键字]来标注某些特殊的内容，通过在微信中回复关键字可以得到相关信息。

下面是我的微信公众号，可扫码加入。

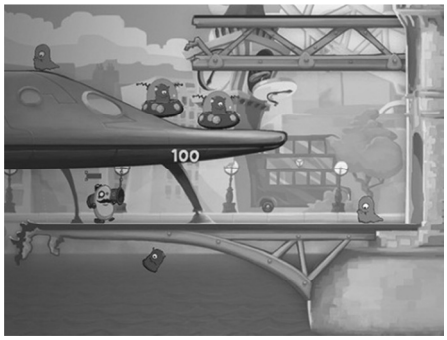
本书前四部分讨论的问题都可以用来做面试题，面试时也可以延伸出很多扩展问题。希望读者能勤于思考，动手实践，“纸上得来终觉浅，绝知此事要躬行”。

资源引用

本书引用的资源基本出自 AssetStore 中，可以免费下载完整版。我对其中的一些资源做了整合，如果希望下载这些挑拣过后的资源，可以直接下载随书的代码工程，地址为[:mygit]。前面的标记表示，在我的微信公众号中回复 mygit 即可获得相关内容，以防链接失效。下面介绍这些资源的出处。

2D platformer

Unity 官方的 2D 示例，AssetStore 的链接为 http://u3d.as/5m5。在这个项目中可以看到 2D 帧动画的制作方法，其中包含了一个可运行的游戏。



2D Roguelike

一个 2D 像素风格的项目，AssetStore 链接为 http://u3d.as/bAx。这个项目展示了如何使用序列帧制作游戏。

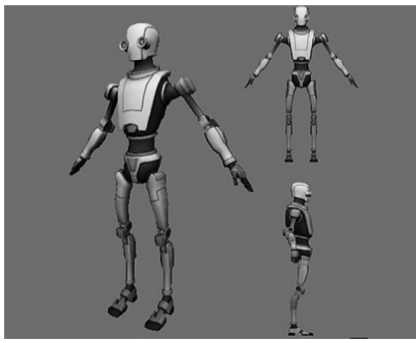
Unity-Chan

Unity 日本公司提供的 Unity 资源，AssetStore 链接为 http://u3d.as/85c。它包含一个动漫少女的模型和动画。截至本书出版，它仍是目前 AssetStore 中，品质数一数二的美术资源。资源的材质包含了目前主流的卡通渲染技术，在日渐火爆的二次元环境下，这个免费样例极具参考价值。

但需要注意的是，这个角色受到 Unity-Chan License Terms 证书的保护，如果需要在商业产品中使用该资源，需要在显眼处标记 UnityChan 公司的标志。

Space Robot Kyle

这是一个 Unity 官方推出的机器人模型，AssetStore 链接为 http://u3d.as/3t7。整个项目十分简单，只有模型、Diffuse 贴图、Normal 贴图和标准 Lambert 材质。由于它效果不错，又是很早就出现的 Unity 官方资源，因此在 AssetStore 中，大量插件都使用它作为功能演示模型。



读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

◎ **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

◎ **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34236

目　录

第一部分　架构与封装

第 1 章　UI 交互 2

1.1　绑定事件响应 2
1.2　事件传递流程 7
1.3　事件响应接口 11

第 2 章　玩法底层 16

2.1　游戏循环 16
2.2　时间记录 23
2.3　动画事件 27
2.4　游戏同步 31

第 3 章　辅助系统 36

3.1　有限状态机 36
3.2　脚本系统 43

第二部分　艺术资源

第 4 章　资源分类 50

4.1　贴图种类 50
4.2　材质效果 54
4.3　动画分类 63
4.4　流动效果 66

第 5 章　后处理效果 71
5.1　模糊效果 71
5.2　泛光效果 79
5.3　辉光效果 83
5.4　景深 94
第 6 章　资源工作流 108
6.1　图片格式更改 108
6.2　动画抽取 111
6.3　文件移动检测 116

第三部分　底层核心

第 7 章　渲染原理 120
7.1　渲染管线 120
7.2　渲染顺序 126
第 8 章　3D 数学基础 129
8.1　点和向量 129
8.2　向量的运算 130
8.3　区域检测 138
8.4　平面移动 142
第 9 章　寻路算法 147
9.1　寻路这件事 147
9.2　A*算法 150
9.3　Navigation 系统 158
9.4　任务调配 161

第四部分　自定义扩展

第 10 章　调试工具 168
10.1　GM 命令 168

10.2 绘制曲线 174
10.3 指示绘制 181

第 11 章 日志工具 190

11.1 出错暂停 190
11.2 日志接口优化 195
11.3 频道化日志 199
11.4 崩溃日志上报 204

第 12 章 快捷功能 207

12.1 自定义菜单 207
12.2 定制 UI 213
12.3 回退操作 217

第 13 章 后台服务 221

13.1 编辑器服务 221
13.2 自动注册框架 225
13.3 遍历文件 228

第五部分 独立游戏

第 14 章 角色分工 234

14.1 产品策划 234
14.2 美术设计 237
14.3 运营知识 241
14.3.1 用户规模数据 241
14.3.2 用户价值数据 244
14.4 总结 245

参考文献 247

第一部分

架构与封装

在开发游戏玩法时，我们常会遇到多变的需求，有些是显示层面的，有些则是深层的游戏机制。

在第 1 章，我们探讨 UI 事件使用方式及底层原理，也会谈到搭建框架的技巧。

玩法是游戏的灵魂。在第 2 章，我们将从游戏循环、时间线、动画事件、同步等多个方面来梳理玩法的要点。

在第 3 章，我们将讨论游戏中的辅助系统，并分析它们的作用。

第 1 章
UI交互

除去《纪念碑谷》这种凤毛麟角的无 UI 游戏，大多数游戏都需要有 UI 交互，甚至一些卡牌游戏就是 UI 游戏，因此开发 UI 的时间占比很大。

要认清一点，UI 本身并没有贵贱之分，只是太多的烂设计毁了它的一世清白。

“包围效应”就是其中一例。这种设计常见于依靠复杂系统支持的游戏：“我不需要你玩游戏，我只需要你点点点”。在屏幕四周摆满了花花绿绿的图标，使得原本就很小的屏幕变得更加捉襟见肘，配上令人抓狂的红点，足以把任何正常人的理智消磨殆尽，从而稀里糊涂地点下这些凌乱图标中最显眼的充值按钮。

在国内游戏市场中，这种游戏遍地开花。而在这些以数值为核心的游戏中，UI 肩负着信息展示与交互的重任。因此，制作 UI 是游戏从业者的必备技能。另一方面，这部分通常比较容易模块化，而且大多能跨项目复用，所以也具备深入讨论的价值。

本章依托于 UGUI，主要探讨使用 UI 控件传递事件的相关问题。

1.1 绑定事件响应

请列出为 UI 控件绑定事件响应的方法，并分析不同方法之间的优劣。

问题分析

面对问题时，最重要的是确定它的边界。对于这个问题，如何绑定当然是关键，但 UI 控件的范围也很重要。不是所有的 UI 控件都能绑定事件，但能绑定的也不是只有按钮，至少还应包含：

◎　拖动条
◎　输入框
◎　开关
◎　滚动区域

当然还有千千万万我们实际工作中用到的自扩展控件。这里不受限于按钮的点击事件，我们用按钮、开关、拖动条三种控件来举例说明。

搭建测试环境

在面临一个新问题时，我们通常会在一个独立的场景中搭建测试环境，来快速验证我们的逻辑是否正确。

在这个例子中，我们要在界面中创建三个控件，并摆放到屏幕的中心。创建方法是用右键快捷菜单添加各种控件，相信大家都知道如何操作。如果不熟悉，也可以通过网络很容易地查到。类似的基础操作不是本书的重点，因此针对此类情况，后文大多一笔带过。创建好的界面大概如图 1.1 所示。

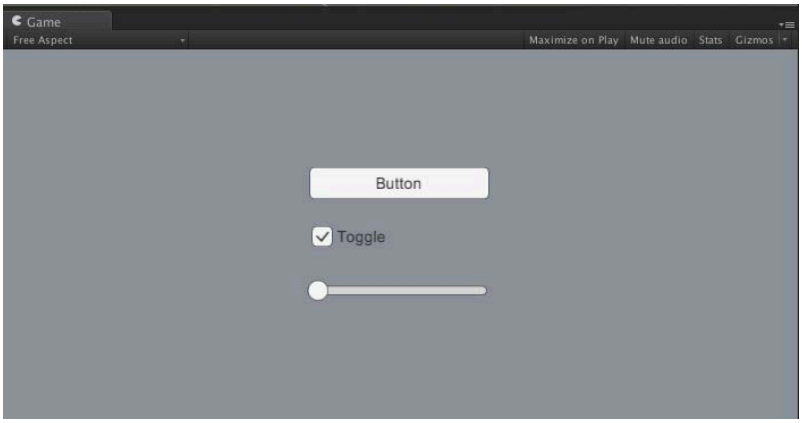


图 1.1

接着，编写处理函数。在 Unity 中，创建名为 TestEvent.cs 的代码文件，并在场景中创建一个空的 GameObject，将 TestEvent 脚本挂载到 GameObject 上。打开 TestEvent.cs，输入如下代码：

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

```
using UnityEngine.UI;

public class TestEvent : MonoBehaviour
{
    public void OnBtnClick()
    {
        Debug.Log("Btn Click");
    }
    public void OnToggleChange(bool isOn)
    {
        Debug.Log("Toggle Is "+isOn);
    }
    public void OnSliderChange(float rate)
    {
        Debug.Log("Slider cur is "+rate);
    }
}
```

现在我们已经配置好了测试环境，接下来就看看如何绑定响应。

绑定响应

从操作层面看，绑定响应的方式有两种，但它们的核心是相同的。

1）组件中添加

在 Inspector 界面中，可以直接在组件中通过拖曳的方式完成绑定。我们以 Button 控件为例，演示如何操作。

首先创建一个空的 GameObject，命名为 MyTrigger，在它上面挂载刚刚创建的 TestEvent 脚本。

完成上述步骤后，在 Hierarchy 面板中选中 Button，然后在 Inspector 面板中找到 Button 组件。不难发现在组件的下方有个区域名为 OnClick，单击这个区域下方的“＋”，就可以添加一个点击事件的处理设置，效果如图 1.2 所示。

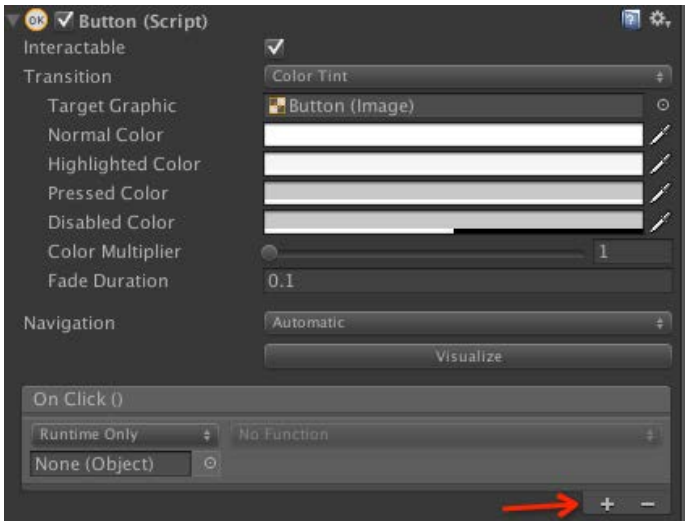


图 1.2

然后将刚刚含有 TestEvent 脚本的 GameObject 拖到下方的“None (Object)”中。操作完成后，单击右侧的“No Function”按钮，在下拉菜单中找到 TestEvent.OnBtnClick。操作成功，效果如图 1.3 所示。

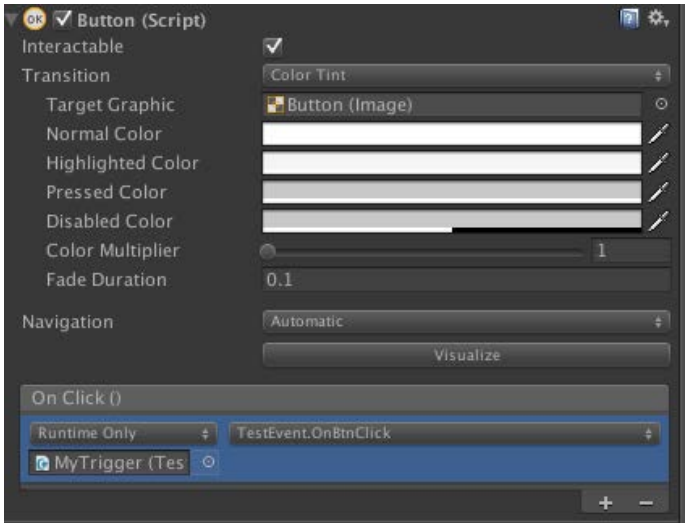


图 1.3

单击“Runtime Only”按钮进行测试，即可发现在 Console 中可以输出“Btn Click”。

用相同的方式，可以为 Toggle 和 Slider 添加相同的事件处理。添加好后，运行游戏，分别测试控件响应，控制台输出如图 1.4 所示。

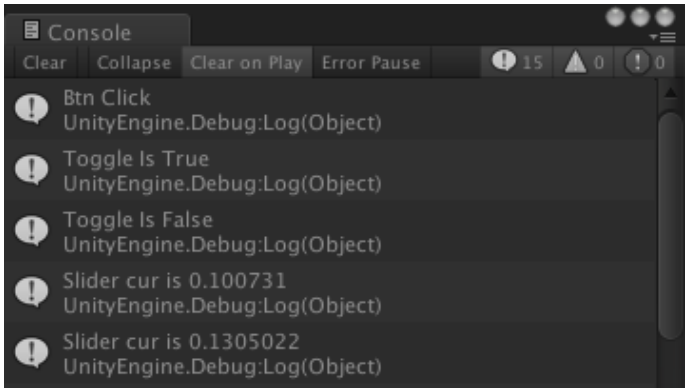


图 1.4

2）代码中添加

代码添加的方式要简单得多，打开 TestEvent 脚本，在类中添加如下代码：

```
public Button m_btn;
public Toggle m_toggle;
public Slider m_slider;

void Start()
{
    m_btn.onClick.AddListener(OnBtnClick);
    m_toggle.onValueChanged.AddListener(OnToggleChange);
    m_slider.onValueChanged.AddListener(OnSliderChange);
}
```

然后在 Inspector 中将三个控件分别拖到脚本的对应位置上即可。

优劣比较

以笔者的经验来看，绝大多数程序员喜欢用代码添加的方式。

一般来说，程序员不会轻易动别人的代码，但预设很可能被其他人更改。当 Bug 出现时，不使用代码编写绑定，开发人员心里就很没底，无法确认是否是自己的原因。

笔者认为如果有很好的框架支撑，编辑器设置绑定的方式也是可取的。就好像没人质疑在 Unity 3D 中，Start 函数是否会在一开始调用一样。如果有一目了然的绑定显示，并且有稳固的框架支撑，那么在编辑器中绑定也没什么不好。在编辑器中绑定有个优势，就是支持动态更新。由于绑定关系存储在预设中，因此意味着整个逻辑可以跟随 AssetBundle 更新。如果 C#代码逻辑写错了，只更新预设也是白费力气，但可以在基类里提供一个默认的出错处理，以便最大限度地降低影响。

总结

绑定响应的方式有两种；在组件中添加和在代码中添加。它们的核心相同，都是通过代理来回调自己的函数。组件绑定的操作更友好，但容易被人更改。代码绑定的优点是更稳定，而且能更灵活地控制监听的时间点，缺点是逻辑编译在代码中，最好有健全的框架支持，以防止出现使用问题。例如，重复绑定的处理函数，忘记移除的绑定，等等。

扩展问题

（1）有没有办法克服代码绑定的缺点？

（2）既然大家都倾向于使用代码绑定，那么组件绑定的存在有意义吗？

1.2　事件传递流程

请从结构的角度概述，控件的消息模型包含哪些部分，并解释 UGUI 中点击事件与响应调用是如何关联在一起的。

问题分析

很多 Unity 3D 项目都使用了 UGUI，但并不是所有人都研究过它的内部结构。针对事件的传递过程，会问住大多数未深入思考过的开发者。由于 UGUI 是开源的，所以我们可以通过查看源码来熟悉它的原理，它的地址在[:uisource]可以找到（前面的标记表示，在我的微信公众号中回复 uisource 可获得相关内容，主要是担心链接失效）。

事件体系

首先说明，事件体系的基础是设计模式中的观察者模式，因此按照标准的 Subject 和 Observer

来解读没有任何问题。但在这里，笔者更希望以功能为导向，将事件体系划分成更易于理解的模块。按这种划分方式，事件体系由四部分组成，分别是：

◎ 监测器（Monito）
◎ 采集器（Collector）
◎ 派发器（Dispatcher）
◎ 响应器（Receiver）

这四个模块大致的依赖关系如图 1.5 所示。

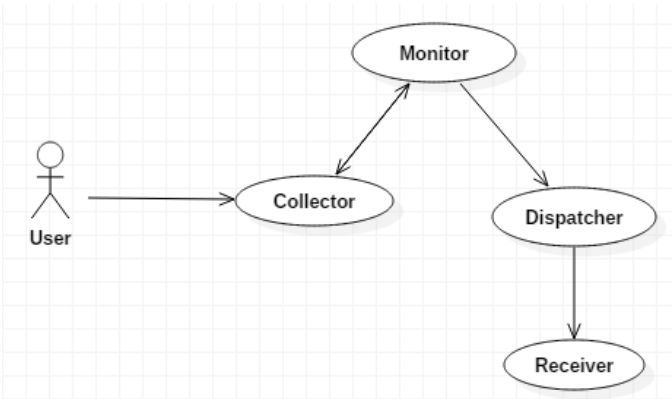


图 1.5

其中用户的操作被监测器驱动的采集器捕获，接着监测器将反馈的信息通知到派发器中，最后通过派发器将事件传播出去。

Unity 3D 实现

在 Unity 3D 中，功能模块的每个部分都有对应的实现类。

◎ 监测器（Monitor）对应的实现类为 EventSystem。它重写了 MonoBehavior 的 Update 方法，会在每一帧更新挂载在同一个 GameObject 上的采集器组件状态，并判断是否应该激活派发器。如果是，则调用各个派发器中的派发函数 Process。
◎ 采集器（Collector）由两部分组成，对应的实现类分别为 BaseInputModule 和 BaseRaycaster，在 UGUI 中默认使用的是它们的子类 StandaloneInputModule 和 GraphicRaycaster。当用户操作时，会先由 BaseInputModule 激活模块，然后发出一个射线点触，返回在 BaseRayCaster 中能点到的物体并返回信息，交由派发器进行过滤。

所以采集器有两端，连接它们要靠 EventSystem。整个过程中还有一个静态内部的管理类 RaycasterManager，用来做连接采集器和监测器的数据桥梁。

◎　派发器（Dispatcher）对应的实现类也为 BaseInputModule，最常用的是它的子类 StandaloneInputMoudle，该类的角色与采集器混在了一起。派发器完成了实际的事件生成，包括且不限于：事件类型的确定、事件内容的提取、派发对象的过滤。其中对派发对象的获取需要借助采集器，但需要通过监测器来驱动。这种设计可以带来效率上的优势，即可以合并采集操作，以达到降低事件频率的目的。

◎　响应器（Receiver）对应的实现类为 IEventSystemHandler 及其子类，例如最常用的 IPointerClickHandler，它的作用是处理点击事件。通过 ExecuteEvents 类，可以将发生事件的对象上的所有响应器都获取到并调用其响应逻辑。以点击为例，事件最终会被派发到 OnClick 的代理上，完成逻辑的执行。

类图

相关实现类有好多，大体分两部分，一部分是功能类，另一部分是编辑器类。

功能类按照前面介绍的事件体系可以清晰地找到重要的基类，它们的 UML 图如图 1.6（[:msgclass]）所示。

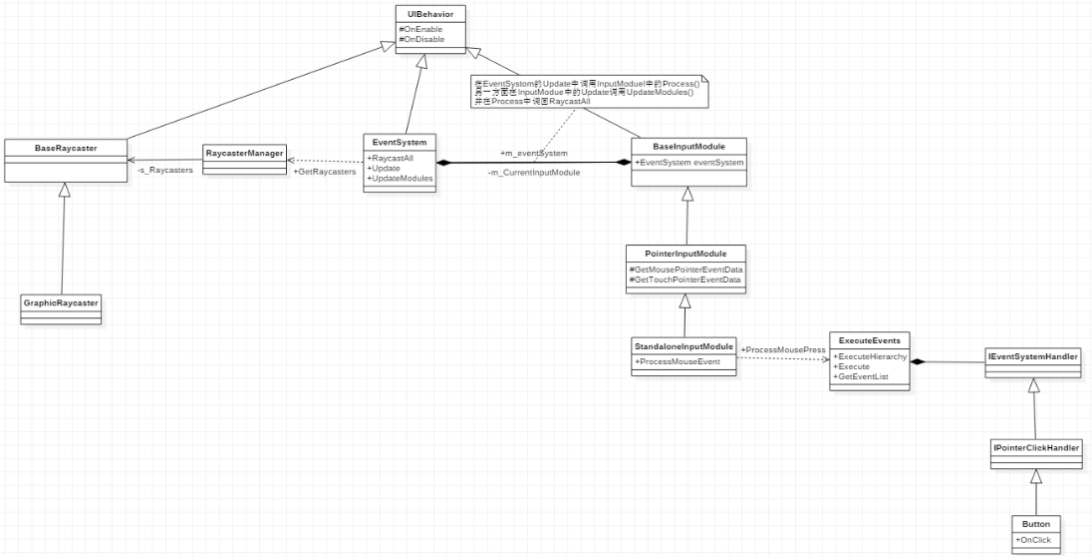


图 1.6

这个图只是和点击相关的部分，当然还有很多其他的功能，例如拖曳、滑动等。如果能找到这些重点，其他逻辑就可以结合代码自行推导了。

另一类是编辑器类，这个比较简单，本书没有单独梳理类图，直接用 VS 自动生成的类图就够用了，如图 1.7 所示。

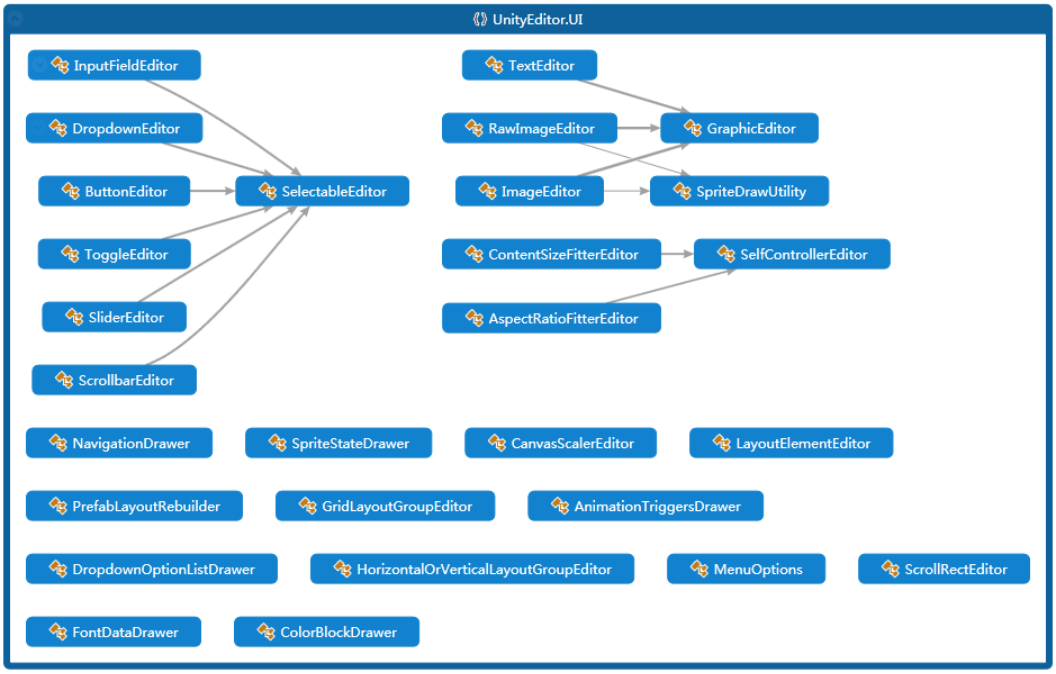


图 1.7

值得一提的是，UGUI 的编辑器界面做得很细致，用起来也很方便。当我们需要自定义 Inspector 界面时，编辑器代码中的一些功能的实现具有参考价值。

总结

如果还是对点击事件的流程不是很清楚，则可以参笔者绘制的按钮点击消息传递的调用流程，具体如图 1.8 所示（[:msgflow]）。

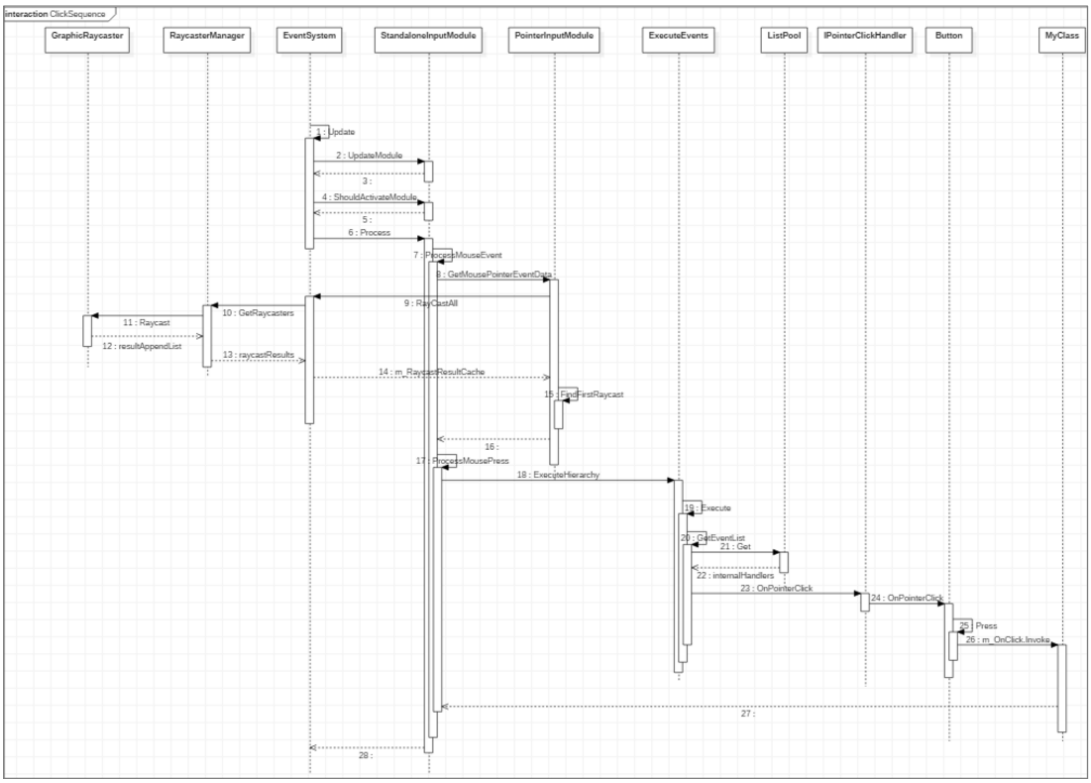


图 1.8

扩展问题

（1）请结合设计模式中的观察者模式，分析 Unity 事件框架的优劣。

（2）如何利用 EventSystem 来降低事件的频率（[:eventrate]）？

1.3　事件响应接口

通常对于事件传递，绑定消息处理时都要写很多相同的框架级别方法，例如：

```
//register
m_btn1.onClick.AddListener(OnBtnClick1);
m_btn2.onClick.AddListener(OnBtnClick2);
m_btn3.onClick.AddListener(OnBtnClick3);
```

```
//Implement
public void OnBtnClick1()
{
    //do sth in btn1
}

public void OnBtnClick2()
{
    //do sth in btn2
}

public void OnBtnClick3()
{
    //do sth in btn2
}
```

请问是否有方法对其简化，减轻开发编码量？

问题分析

这个问题并不难。但很多时候，并没有想过从框架层解决冗余代码。这个问题有时也会换个更开放的方法来问：“说说你的 UI 框架吧。”具体落实到代码上，回答它的重点跟这个问题有很大的重合性。

我们都知道，编程时有个 DRY 原则，即“Don't Repeat Yourself”。它主张不要编写重复的代码。落实到实际，就是避免在代码中无脑地复制粘贴。

在《重构：改善既有代码的设计》中，不合理的代码结构被称为“代码的坏味道”，破坏 DRY 原则就很难写出高质量的代码。如何去除代码中的“坏味道”，在这个问题中能得到很好的体现。

添加分层

很多年以前，计算机领域的先驱，David Wheeler 就提出过这样的说法：

“计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决”

针对这个问题，我们也可以将分层作为突破点。在现有的基础上添加一层，是最容易完成需求的做法。

按照这样的思路，我们需要创建一个容器类，它可以自动将可响应事件的子节点找到，并为其添加监听。代码大概是这样：

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class MyPanel : MonoBehaviour
{
    void Awake()
    {
        var btns = GetComponentsInChildren<Button>();
        for(int i = 0;i < btns.Length;++i)
        {
            btns[i].onClick.AddListener(OnBtnClick);
        }
    }

    public void OnBtnClick()
    {
        Debug.Log("btn click");
    }
}
```

然后在新场景中创建一个挂载这个脚本的节点。创建好后再为其添加三个按钮子节点。直接运行游戏，单击按钮即可看到输出，如图 1.9 所示。

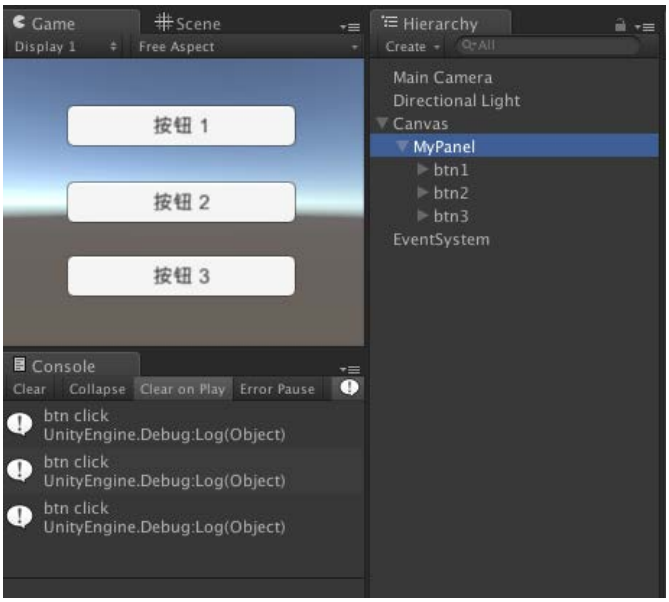


图 1.9

优化事件接口

现在点击可以自动注册了，但无法区分出单击的是哪个按钮。

所以要增加回调的标识，采用一个简化的方式：以节点被搜到的顺序作为 Key，将节点的索引值传到回调中，就可以知道单击的是哪个按钮了。

在正式的代码中，可以用节点名或属性值作为标记，以防止顺序更改出现对应的错误。也可以使用更加稳妥的方法，例如，使用 GameObject 的 GUID，或创建一套 ID 映射机制，用于标定具体对象的对应关系，实现这个功能的代码如下：

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

public class MyPanel : MonoBehaviour
{
    void Awake()
    {
        var btns = GetComponentsInChildren<Button>();
        for(int i = 0;i< btns.Length;++i)
```

```
        {
            int index = i;
            btns[i].onClick.AddListener(()=>{
                OnBtnClick(index);
            });
        }
    }

    public void OnBtnClick(int index)
    {
        Debug.Log("btn click:"+index);
    }
}
```

我们在这段函数的循环中，创建了一个 index 变量。需要注意的是，这个变量不可以移到函数外，因为它分配出的内存要跟着回调传递。这其中的内在原理可以查询 C#中匿名函数的相关知识。为了保证主题明确，此处不再展开，感兴趣的读者可以去网上自行搜索。

扩展问题

（1）有办法对各种控件添加统一的响应接口吗？

（2）有时候界面中会有大量不显示的按钮，这会影响我们的封装吗（[:hiddenbtn]）？

第 2 章
玩法底层

游戏的卖点通常是有特点的玩法，而这些玩法通常又会比较多变。因此我们在着手制作游戏玩法之前，通常要考虑很多结构上的设计，这些内容一旦确定，就不容易更改。有些结构有自己的优势，却无法兼顾某种类型的需求。

本章我们从几个常见的方面入手，看一下玩法底层有哪些需要注意的内容。

2.1 游戏循环

请简述 Unity 3D 中常用的生命周期函数，并以此为基础简述游戏循环的设计要点。

问题分析

在使用 Unity 3D 开发游戏时，总会涉及 Unity 内置的生命周期函数。弄清这些函数的调用顺序和特性十分重要，因为这影响到逻辑的执行顺序。例如，初始化要在使用之前，注册回调要在响应之前，等等。

一般来说，在大型商业项目中，我们需要自制一套游戏循环，来满足多变的游戏玩法。哪些现有的生命周期函数可以使用？哪些需要被替换掉？这些取舍贯穿了整个游戏开发过程。

生命周期

Unity 3D 的函数调用有固定的顺序，对于脚本生命周期（Script Lifecycle），官方文档中给出了一张时序图，如图 2.1 所示。（[:lifecycle]）。

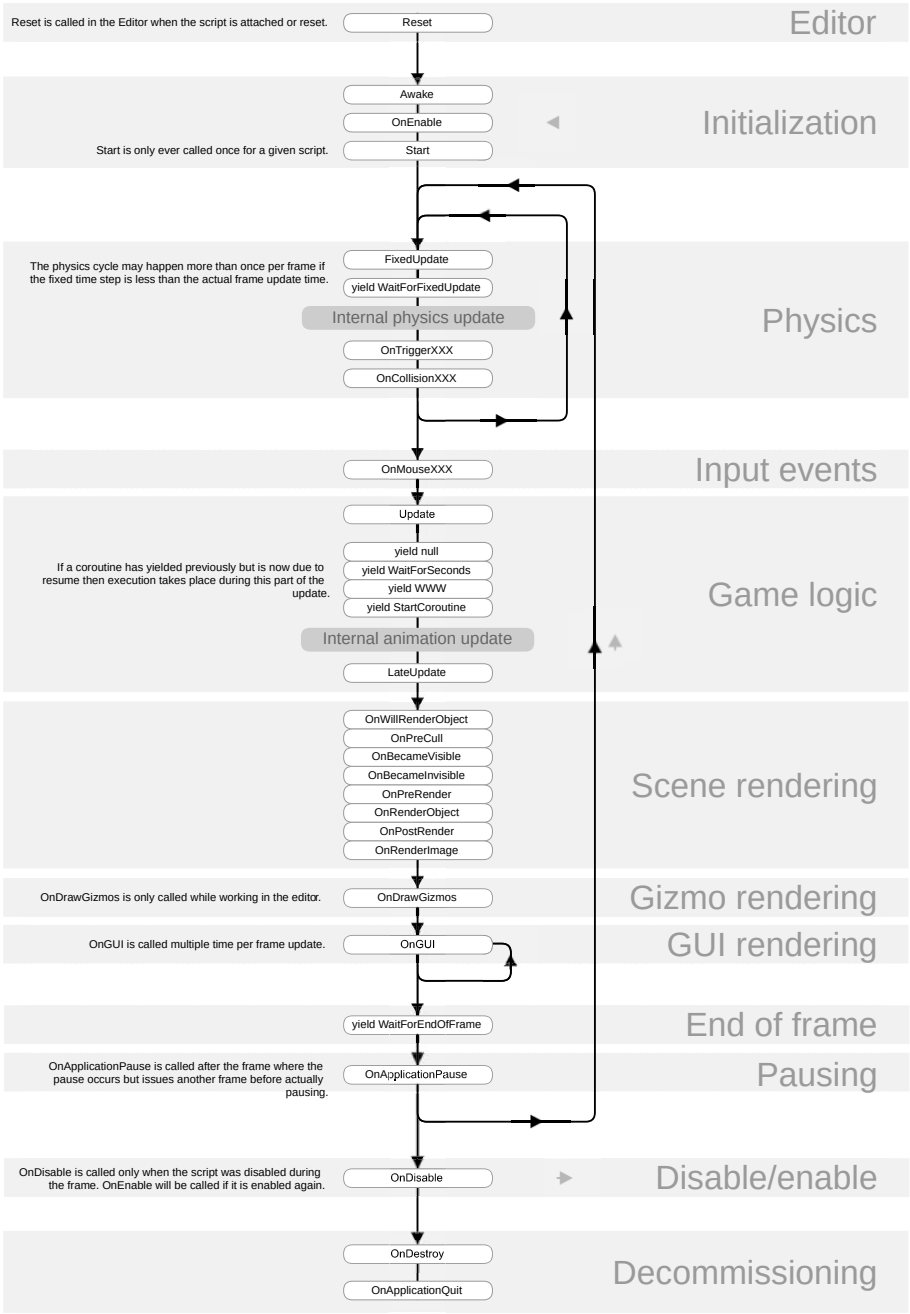

Reset is called in the Editor when the script is attached or reset.
Reset
Editor
Awake
OnEnable
Start is only ever called once for a given script.
Start
Initialization
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
yield WaitForFixedUpdate
Internal physics update
OnTriggerXXX
OnCollisionXXX
Physics
OnMouseXXX
Input events
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
LateUpdate
Game logic
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Scene rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
Gizmo rendering
OnGUI is called multiple time per frame update.
OnGUI
GUI rendering
yield WaitForEndOfFrame
End of frame
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Pausing
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
Disable/enable
OnDestroy
OnApplicationQuit
Decommissioning

图 2.1

深入理解这张图，可以应对大多数的时序问题。在 Initialization、Disable/enable、Decommissioning 区域的函数只会调用有限次数，而其他部分都是受控重复或循环调用，其中：

◎ Awake 函数在构造脚本对象时调用；
◎ OnEnable/OnDisable 在每次激活对象时调用；
◎ Start 在第一次触发脚本时调用；
◎ OnDestory 在销毁对象时调用。

主循环分为物理模拟、游戏逻辑、渲染绘制三个子循环。从 Unity 实现的方式来看，这三个循环是在同一个线程中的。不过自 Unity 3D 的 5.x 版本后，引擎加入了多线程渲染的选项，即把第三步循环放在另一个线程中进行，这样就可以在固定帧率下降低逻辑循环的压力，减少掉帧现象的发生。至于物理模拟循环，笔者觉得没必要拆出到另一个线程。毕竟线程间通信也需要性能，而物理模拟循环运行的速度一般要快于游戏逻辑，如果出现数据访问冲突时还需要加锁，就得不偿失了。

重写模板脚本

在实际项目中，可以通过更改代码的模板文件，来优化生命周期函数的使用方式。在 Unity 的安装目录中，有创建代码的模板文件。通过自定义这个文件，可以更改创建默认文件的内容。文件目录如下。

◎ Windows：Unity 安装目录/Editor/Data/Resources/ScriptTemplates/81-C# Script- NewBehaviourScript. cs.txt。
◎ Mac：Unity 安装目录/Contents/Resources/ScriptTemplates/81-C# Script-NewBehaviourScript. cs.txt。

根据团队开发人员的编程习惯，笔者自定义了一个代码模板，大家可以根据自己的需要做相应更改：

```
using UnityEngine;
using System.Collections;

public class #SCRIPTNAME# : MonoBehaviour
{
    #region Public Attributes
    #endregion
```

```
    #region Private Attributes
    #endregion
    #region Unity Messages
//    void Awake()
//    {
//
//    }
//    void OnEnable()
//    {
//
//    }
//
//    void Start()
//    {
//
//    }
//
//    void Update()
//    {
//
//    }
//
//    void OnDisable()
//    {
//
//    }
//
//    void OnDestroy()
//    {
//
//    }

    #endregion
    #region Public Methods
    #endregion
    #region Override Methods
    #endregion
    #region Private Methods
    #endregion
    #region Inner
    #endregion
}
```

代码模板中添加了常用的生命周期函数，并按照顺序进行排列。由于空函数也会产生性能消耗，因此这里采用注释的方式来规避这个弊端。另外，笔者按用途添加了几个#region 分隔函数区域。#region 是 C#的功能，可以标定折叠区域，方便查找对应函数。

［注］本节后续内容部分参考（参考书目 6）

游戏循环

了解了 Unity 自带的生命周期函数之后，再看看游戏循环应该如何设计。游戏归根结底是由交互序列组成的，因此它必然会有一个基础的结构：

```
while (true)
{
  Input();
  Update();
  Render();
}
```

从这个层面看，游戏循环由三部分组成，分别是

（1）非阻塞的用户输入；

（2）更新游戏逻辑状态；

（3）渲染游戏画面。

每次循环完成后，会更新一次画面的绘制，这个过程也被称为帧（Frame）。帧率（FPS，Frame Per Second）可以标定游戏循环的速率与真实时间的映射关系。帧率值越小，意味着游戏越“卡”。游戏在 PC 上，通常为 60 帧/秒，在手机上为 30 帧/秒。另一方面，帧率的倒数即为每帧所占用的时长，单位通常为毫秒。影响帧率的主要因素是每帧需要做的工作。例如，复杂的物理计算、游戏逻辑的处理、图形细节控制等，这些都会占据 CPU 与 GPU。如果处理操作的时长超过帧率的倒数，那么就会拖慢帧率，这种现象称为“掉帧”。

固定帧率模式

一般来说，我们有个期望的帧率，如果每帧的运行时间短，那么帧率就会超过预定的标准，因此我们通常会在循环的末尾加入延期等待。假定有个 Sleep 函数可以阻塞线程执行，即这个模式的代码结构如下：

```
while (true)
{
  double start = getCurrentTime();

  Input();
  Update();
  Render();

  sleep(start + 1/FPS - getCurrentTime());
}
```

在这种结构中，帧率不会超过预定数值。在 Unity 3D 中可以通过下面的代码设置：

```
Application.targetFrameRate = FPS;
```

追赶模式

这种模式可以更好地处理掉帧引发的逻辑问题。大体思路是，当出现掉帧时，只运行逻辑，不绘制画面，用节省下来的时间追赶落后的帧。这种策略会降低图形绘制的频率，但可以保证逻辑的执行。

具体来说就是在每次 Render 执行之前，要保证累计运行时长达到阈值。如果出现卡顿，则后面的帧就会多次执行 Update，直到赶上之前的帧为止。代码结构如下：

```
double preFrameTime = getCurrentTime();
double lag = 0.0;
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - preFrameTime;
  preFrameTime = current;
  lag += elapsed;

  Input();

  while (lag >= 1/FPS)
  {
    Update();
    lag -= 1/FPS;
  }
```

```
  Render();
}
```

使用这种模式时，要注意不要将 FPS 设置得太大，否则最慢的机器将永远赶不上时间，它将卡在死循环中。对于较差的机器，Render 在逻辑循环之外，所以总体来看还是会节省一些时间。虽然看起来会比较卡，但基本能够正常运行游戏。

在 Unity 3D 中，对应这个模式的循环是 FixedUpdate。在 Unity 中设置 Fixed Timestep 可以控制 FixedUpdate 速率，其数值为时长周期。如果 FixedUpdate 在限定的时间内执行不完，则图形绘制频率就会降低，以保证物理的执行。另一方面，设置 Maximum Allowed Timestep 可以防止逻辑执行时间过长，卡死线程，如图 2.2 所示。

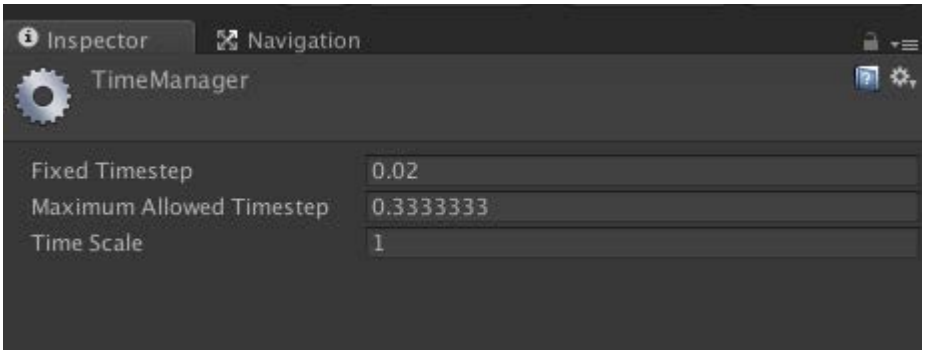


图 2.2

总结

Unity 3D 作为完整的引擎，常见的生命周期函数与游戏循环模式都已具备。但作为特定的游戏，通常有自己的特点。例如，竞速类游戏与 MMO 网游在游戏循环的设计上就有很大的差别。竞速类游戏对实时反馈的要求很高，如果采用追赶模式，则抽帧造成的体验就会很差。在掉帧方面，MMO 网游面临的则是角色在场景中漫游时，其他玩家的模型加载与位置同步造成的卡顿。在这种情况下，可能会使用分帧加载、AOI（Area Of Interest）等处理方法来保障游戏的流畅。

因此，游戏循环通常是每个项目根据自己的特点“独家定制”的。在深入理解 Unity 3D 的生命周期函数后，我们就可以在其基础上，自主独立地搭建个性化的游戏循环框架了。

扩展问题

请尝试分析多线程渲染的实现原理（[::mutirender]）。

2.2　时间记录

请问在 Unity 3D 中常见的时间分为哪几类，获取和记录它们的方法有哪些？它们的适用场景分别是什么？

问题分析

编写游戏的实时反馈型系统时，不可避免地涉及时间，比如人物行走的位移计算、战斗中的延迟产生伤害等。在编写这些功能时，我们要小心地处理对应逻辑。例如，在掉帧的低端机上，帧率会处于波动状态，这时在编写时间相关的逻辑运算时，就要防止出现与标准帧率不一致的情况。在商业项目中，通常都会编写自己的时间处理类，以得到跨设备准确无误差的时间值。

真实时间线

为了方便理解和说明，我们可以将游戏中的时间想象成一条沿着 X 轴正向的射线，这被称为时间线（TimeLine）。游戏的时间为沿着射线不断向正方向扩充的矩形区域，射线的原点与矩形的起始点为相对零点。

具体到真实时间线，它的起点为一个逻辑上的零点，它可以是设备的开机时间，也可以是游戏开始运行的时间，甚至是服务器下发的服务器时间。总之，它是一个在实际世界中真实存在的确定值。真实时间线（Real TimeLine）指的就是，以真实时间的流逝作为标准，进行时间统计的时间线。从重要程度上讲，它代表着绝对的时间流逝，是一切时间的参考，不同设备得到的结果一定是相同的，这也是整个时间记录的基石。

在 Unity 3D 中，使用 Time.realtimeSinceStartup 可以获取游戏运行了多长时间。为了测试功能，笔者创建了一个名为 TestTime.cs 的文件，挂载场景中的 GameObecjt 后，在其中输入如下代码：

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

```
using System.Collections.Generic;

public class TestTime : MonoBehaviour
{
  void OnGUI()
  {
    GUILayout.Space(20);
    GUILayout.Label("RealTime: "+Time.realtimeSinceStartup);
  }
}
```

运行游戏可以看到记录时间的文字出现在游戏视窗中。

游戏时间线

游戏时间线（Game TimeLine）是应用更广泛的时间计量方式。它以游戏世界的时间流逝速度作为标准度量时间。很多情况下，它与真实时间线不同。例如，当游戏呼出暂停菜单时，真实时间仍然在流逝，而游戏时间却应该停止。又比如，在回放战斗录像时，我们希望快进整个战斗过程，这时游戏时间的流逝速度就要快于真实时间。在有些强调打击回馈感的动作游戏中，游戏时间线速率会不停地变化，让玩家在忽快忽慢中体验战斗的乐趣。

游戏时间线也可用于辅助调试。为了追查不正常的渲染，通过停止游戏时间，开发者可以冻结所有的动作和特效。这时如果有能够控制摄像机移动的工具，开发者就可以在场景中漫游，查找问题产生的原因。更进一步，通过推动游戏时间向前移动帧时间（1/FPS），开发者可以实现逐帧调试的功能。暂停游戏的另一个好处是方便定位 Bug 现场。结合代码报错的监控模块，可以很容易地捕捉报错现场，有利于问题的解决，在后面我们还会提到这点。

在 Unity 3D 中，我们可以使用 Time.time 来获取游戏时间，通过 Time.timeScale 来控制游戏时间流逝的速度。更改 TestTime.cs 脚本中的代码如下：

```
using UnityEngine;
using System.Collections;

public class TestTime : MonoBehaviour
{
  void OnGUI()
  {
    GUILayout.Label("TIme Scale:"+Time.timeScale);
```

```
        Time.timeScale =
            GUILayout.HorizontalSlider(Time.timeScale,0,2,GUILayout.Width(200));
    GUILayout.Space(20);

    GUILayout.Label("RealTime: "+Time.realtimeSinceStartup);
    GUILayout.Label("GameTime: "+Time.time);
    GUILayout.Space(20);

  }
}
```

编辑好后，运行游戏。拖动游戏窗口中的拖动条，可以更改游戏时间的流逝速度。例如，笔者将其更改为 0.5，则真实时间每过 2s，游戏时间就过 1s。另外，当在 Editor 中暂停游戏时，游戏时间也是不计算的。程序运行效果如图 2.3 所示。

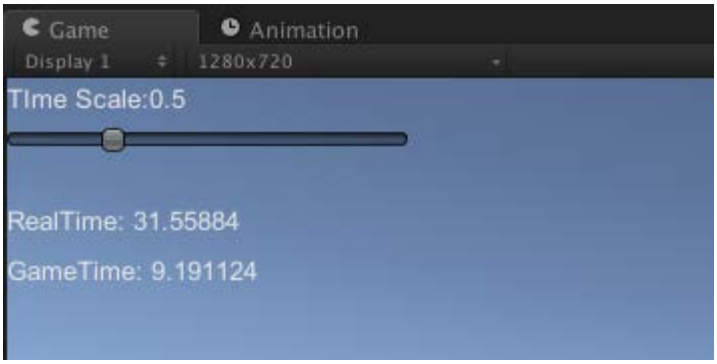


图 2.3

间隔时间

有了上面的概念，间隔时间就容易理解了。间隔时间的计算方法为计算两帧之间的差时，它也会用在很多场景中。例如，在计算距离时，我们通常会用间隔时间与速度相乘，这样得到的结果就是每秒的位移，而不是每帧的位移。

理想情况下，帧率不会发生改变，也是就说，逻辑计算的频率是固定的。如果游戏时间线缩放变小，即游戏变慢，则实际单位时间内运行的逻辑计算次数将增加。为了使最终结果相同，游戏时间线中的间隔时间就会变短，即游戏间隔时间与游戏时间流逝速率成正比。当然，更多情况是帧率会抖动。如果我们需要使用过去或当前的间隔时间，来预测物体未来的位置状态，则使用平均间隔时间会更加合理。

在 Unity 中通过 Time.unscaledDeltaTime 可以取到真实的间隔时间，Time.deltaTime 可以取到游戏间隔时间，Time.smoothDeltaTime 可以取到平均的间隔时间，这个“平均”是基于游戏时间线的。

由于 OnGUI 函数每帧会被调用多次，虽然每次调用的值都是本帧的值，但影响我们观察数据，因此笔者将对应输出写在了 Update 函数中。在 TestTime.cs 文件中加入如下代码：

```
using UnityEngine;
using System.Collections;

public class TestTime : MonoBehaviour
{
  //......

  void Update()
  {
    Debug.Log(" TimeScale:"+Time.timeScale+"\n"+
      "DeltaTime"+Time.deltaTime+"\n"+
      "Unscaled DeltaTime"+Time.unscaledDeltaTime+"\n"+
      "Smooth DeltaTime"+Time.smoothDeltaTime);

  }
}
```

编辑完成后，运行代码。当调整游戏视图中的 TimeScale 为 0.5 时，可以看到游戏真实的流逝时间是 0.008，而间隔时间为 0.004，是实际间隔时间的一半，如图 2.4 所示。

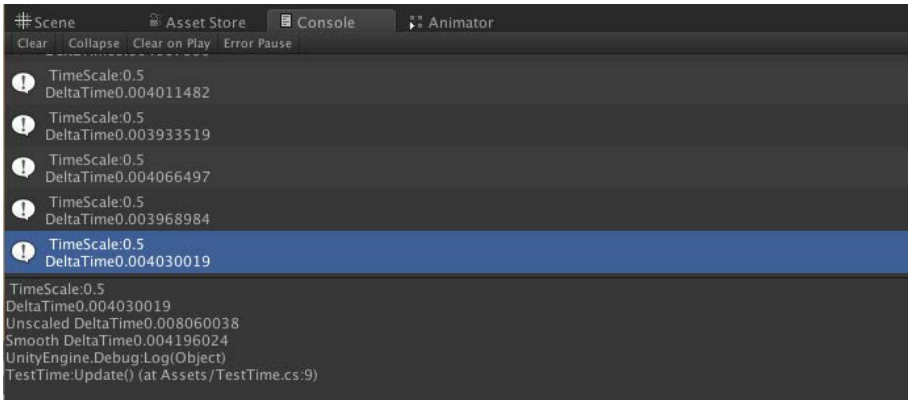


图 2.4

总结

深入了解时间统计的原理、时间线概念及 Unity 3D 对应的接口函数，我们可以更方便地设计出适合自己游戏的时间线控制类。在实际项目开发过程中，应尽早在不同性能的机器上进行时间相关的测试，以防止出现某些逻辑用错时间线，引发难以更改的问题。

扩展问题

现在有两种情况引起游戏间隔时间变长：一种是目标机器性能差，需要追赶补偿效果；另一种是断点调试，不希望影响逻辑。请问有没有办法在设计时间线时，过滤掉断点调试的情况（[:gametimelong]）？

2.3　动画事件

请简述动画事件的制作与逻辑处理方法，如果有多种备选方案，请对比它们的优劣。

问题分析

在编写战斗系统时，我们需要处理动画事件。所谓动画事件，指的是动画进行到某个时刻触发的事件。例如，当角色跳起，双脚离开地面时，要播放吹起烟尘的特效；当角色落地时，要发出撞地的声响；角色攻击，手举到最高点时，要投掷出飞镖，等等。这些与动作紧密结合的功能都属于动画事件（后面简称事件）。如何制作这些事件，并在运行时控制游戏逻辑，成为战斗逻辑不可或缺的部分。

基于动画的事件

对于显示型事件，比如播放声音或特效，通常采用基于动画的事件。尤其是当动作之间存在复杂的混合时，这种事件编辑方式尤为重要。例如，开枪动画与行走动画进行混合，以求达到边跑边开枪的动画效果。在开枪时要播放弹壳四散的特效，跑步时要播放脚步声，由于开枪动作与跑步动作的时长不同，为了使两者融合后看起来更加合理，美术人员需要反复调整资源或融合参数。在 Unreal 引擎中，这部分功能更加完善，它提供了详细的参数以帮助美术人员调整事件效果，效果如图 2.5 所示。

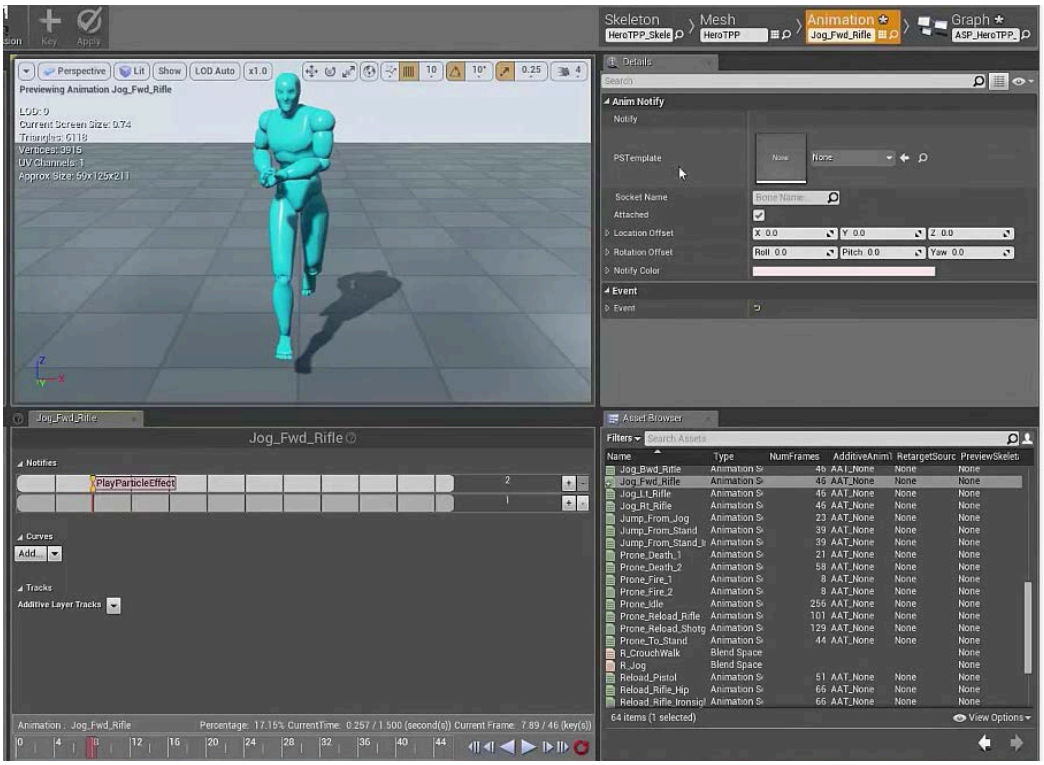


图 2.5

在 Unity 3D 中也可以制作基于动画的事件。为了详细说明这个功能，笔者创建了一个名为 AnimEvent.cs 的脚本文件，在其中编写如下代码：

```
using UnityEngine;
using System.Collections;

public class AnimEvent : MonoBehaviour
{
    public void MyEvent()
    {
        Debug.Log("My Event Triggered");
    }
}
```

编辑好代码后，选择一个动作的 FBX 文件。这里我们使用 Unity-Chain 资源包，它是 Unity 日本分部开发的一个示例资源，是不少开发者心中的 Unity 吉祥物，在后面涉及资源相关的内

容时均会使用它。

单击 Inspector 中 Animation 选项卡底端的 Events 折叠标签。标签打开后，将动画调整到合适的帧，单击左侧的“添加”按钮，即可添加帧事件的回调函数，我们这里添加刚刚写好的函数，效果如图 2.6 所示。

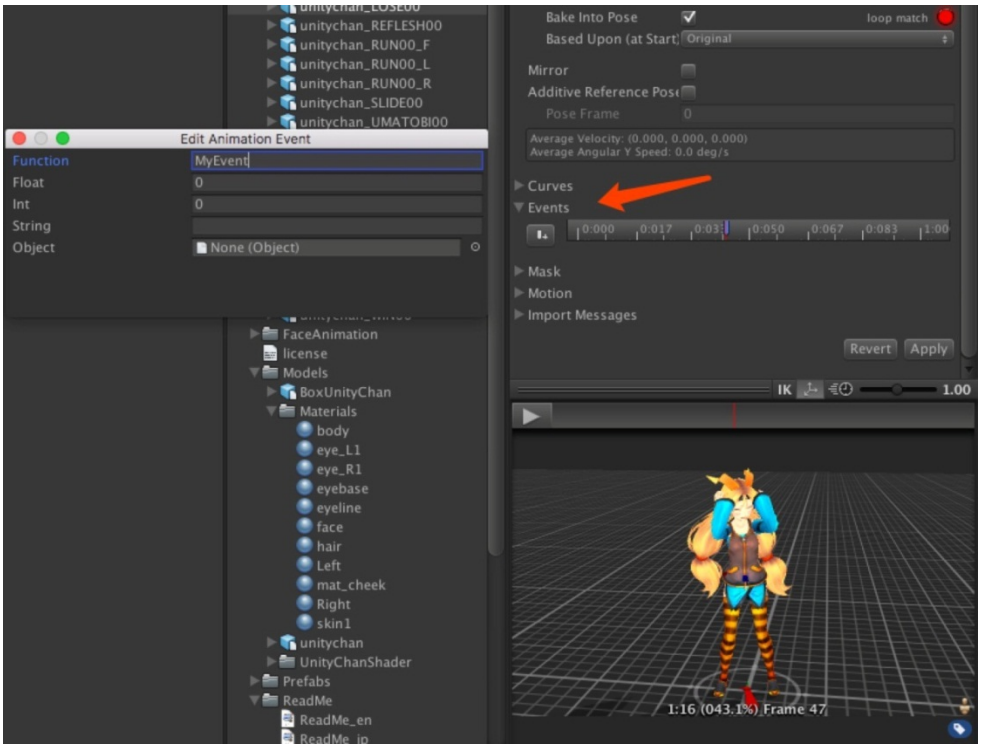


图 2.6

保存设置后，单击 Inspector 界面右下角的“Apply”按钮保存更改。

创建一个新场景，将动作对应的模型拖入场景中。接着将动画的 AnimationClip 文件拖到场景中的模型上，这会创建只有一个动画的 Animator。最后将 AnimEvent 脚本挂载到这个模型上，运行游戏，可以看到当运行到对应帧时，有自定义的日志输出，效果如图 2.7 所示。

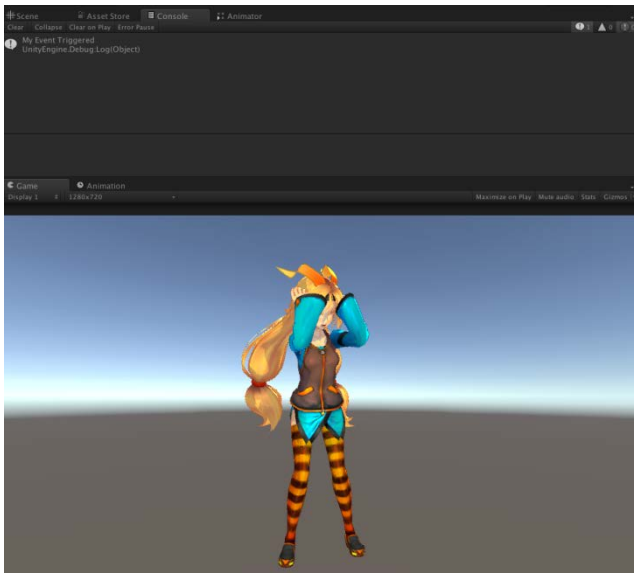


图 2.7

基于时间的事件

另一种常见的事件是逻辑型事件。虽然它也与动画的状态紧密相关，但这些事件中包含着逻辑处理。例如，角色在攻击动作的过程中触发伤害；或者在攻击出拳的动作过程中向前移动，遇到敌人就停止移动。这些事件与逻辑紧密相关，不仅取决于美术的表现效果，更多是数值与逻辑的合理性。

对于这种事件，采用基于时间的动画事件就更为合理。在制作事件时，我们记录的是局部时间线的偏移值，并将它以独立文件的形式存储。在上一节中，我们知道游戏中会存在一条游戏时间线。当动作开始时，我们会将这些事件时间偏移量并入游戏时间线，并在游戏时间线的对应时间添加触发事件的回调，这样事件触发就完全脱离了模型动画。

这样做的另一优势是显示与逻辑的分离。在同屏多人的游戏中，有时会有不播放角色动画，只产生技能触发的需求。在这种事件框架下，就可以只运行基于时间的事件而不去加载模型与动画。因为伤害触发与计算不需要依赖于动画，所以一切逻辑都可以正常运行。另外，在有异步数据影响的场景中，这种事件控制方法也有优势，它可以预先做好逻辑计算，再对动画做插值处理，数据优先于动画直接进行同步，将网络延迟对游戏的影响降到最低。

总结

动画事件的制作方法需要灵活选择。一方面需要根据自己游戏的特点，另一方面也要注意自己团队的配置人员能否习惯对应的工作流程。数据载体其实也有多种备选方案。可以通过编写插件，将基于动画的事件帧与 3D Max 中的动画制作联系在一起。也可以通过编写导出工具，将基于时间的事件配置导出成配置表等。这里要根据实际情况选择处理方式，根据经验做取舍判断，切不可生搬硬套。

扩展问题

在后期优化时发现同屏特效过多，在处理基于动画的特效事件时存在性能瓶颈，请问该如何优化（[:animeventopt]）？

2.4　游戏同步

请问在游戏同步方面有哪些分类方式？帧同步与状态同步有何优劣？

问题分析

随着竞技游戏的日渐火爆，越来越多的实时对战游戏被开发出来。我们知道，网络交互是需要时间的，如何才能在多人对战的场景中，将同步信息的延迟降低，以保证玩家的游戏体验，是游戏同步主要研究的问题。同步策略通常需要根据不同的项目类型单独制定，后期还需要经过大量玩家的检验才能完善。

游戏模式

根据游戏中玩家之间的互动方式，可以将多人交互类型的游戏大体上分为两个类别：MO 与 MMO。这两种典型的模式有着完全不同的侧重点和技术架构。

多人在线游戏（Multiplayer Online，MO）指的是少量玩家聚集在一起的竞技游戏。它的侧重点是实时对战的战斗乐趣，因此游戏追求高速的响应时间，以及频繁的交互操作。相对应的，游戏时间通常在几分钟到几十分钟不等，通常采用房间或副本的形式进行游戏，玩家数量控制在 20 人以内。常见的类型为 ARPG、FPS、格斗、RTS、竞速等。

大型多人在线游戏（Massively Multiplayer Online，MMO）则是期望大量玩家进行长期的社

交性游戏。它追求的是一种在虚拟社区中的存在感，因此游戏周期通常几个月以上。以最成功的 MMO 类型的游戏魔兽世界（World of Warcraft）为例，整个游戏运营已接近十五年。更是有新闻报道少年继承亡父的账号，足见此游戏的魅力。由于要处理大量的玩家数据，难免会牺牲响应时间，因此在 MMO 游戏中，通常不会有高速的交互。为了能长期运行，系统的稳定性与可维护性也是框架的重中之重。

MO 与 MMO 在游戏内容上有很大差别，所需的技术也截然不同，但一个游戏中同时存在两种模式并不冲突。常见的做法是，将 MO 类型的游戏以副本的形式嵌到 MMO 中，并分别采用独立结构对游戏逻辑提供支持。这样就可以兼顾 MMO 的玩法多样性与 MO 对战的乐趣性。在 WoW 中，野外战斗与大部分副本都是以 MMO 架构实现的，而少数在短时间内需要玩家紧密配合、历尽艰难才能通关的副本则是通过响应更快的 MO 架构实现的。

通信方式

1）C/S 架构

从通信方式上看，同步可以分为 C/S 与 P2P 两种。C/S 是 Client/Server 的缩写，指的是客户端与服务器连接的消息传递方式。在这种框架下，所有消息都通过服务器转发。它的优势在于可以很好地控制非法行为。服务器知道一切数据，通常也会对数据进行模拟计算，因此，作弊行为很难出现。另一方面客户端也可以只做表现，完全由服务器的数据驱动。这种方式常用在不需要，或很少需要预表现的游戏中。

这种结构的最主要弊端是网络延迟较大，通常会控制在 200ms 以下。另一方面，中央服务器上大量的运算也会增加服务器的设备成本。在商业项目上线之前，通常会进行压力测试，主要关注点在带宽、连接数、响应速度等。通过模拟大量玩家的操作，来测试服务器硬件是否能够应付预期数量的玩家。

2）P2P 架构

与 C/S 架构不同，P2P 架构是两个客户端之间直接连接。它与 C/S 结构的优劣正好完全倒置，它有低延迟的优点，也可以节省大量的服务器运算，却很难避免作弊的行为。例如，DOTA 中的全图外挂无法禁掉，根本原因就是客户端拥有其他玩家的数据。

数据模式

1）全网状数据模式

全网状数据模式指的是，每个客户端需要给网络中的每个设备广播自己的数据。因此每个设备上都有完整的数据，各个终端之间传递的只是控制设备的输入信息。可以预想到，随着网络中终端数量的增加，信息量会急剧增长。由于消息量大，因此不同客户端之间通常只会传递用户输入的信息，这就需要保证每个客户端的初始数据与每次操作的运算结果必须完全相同，否则就会表现不一致。

2）星型数据模式

另一种情况是网络中的成员并不平等，所有终端都要依从中央终端的数据。这个终端在 C/S 架构中被称为中央服务器，在 P2P 架构中被称为主机，这里为了方便说明，暂称为服务器。在标准的星型结构中，服务器在收集完所有设备发送的数据之前，一直处于等待状态，接收完成后再将数据广播出去。因此它是将输入数据暂时集中到服务器上，这一点与全网状数据模式完全不同。

这种结构避免了终端过多导致的数据暴涨，但也会引起数据传输速度的下降，有利有弊，需要根据自己项目的实际情况做取舍。

同步方式

1）状态同步

本节中提到的状态同步和帧同步都是广义上的，虽然它们的底层实现都可以基于上面提到的模式，但从广义上讲，它们的同步思路不同。

状态同步指的是将其他玩家的状态行为同步，例如，怪物的 AI 控制、角色技能释放、战斗伤害计算等。纯粹的状态同步模式下，这些内容都由服务器运算，只是将结果下发给客户端，客户端根据得到的数据驱动显示即可。这种模式的缺点是流量消耗比较大，消耗的流量取决于场景中需要转发数据的人数和内容。另外，先天的结构导致响应速度存在问题，无法做到高频交互的顺畅体验。最明显的特点就是“拉扯”现象。它表现为某个角色会突然出现在某个位置，或某个技能效果突然出现，甚至角色忽然死亡等。引发这个现象的原因是，网络波动导致数据未能及时送达，而客户端进行了某些程度的预表现或航位推测。

对于大多数实时性要求不高、交互简单的游戏，一般会使用状态同步。通常来说，常规MMO 类型的游戏只能使用状态同步。

2）帧同步

帧同步是指客户端之间只同步用户的输入指令，例如，向前走、按下哪些技能等，不同的客户端各自计算自己的结果。由于消耗的流量只取决于指令数，因此会大大减少消息。另外由于消息结构的原因，帧同步的速度要比状态同步更快，因此适用于一些高频交互的游戏。

不过由于每个客户端需要独立计算，因此需要保证计算结果的一致性。理论上讲，相同的时机，输入相同的内容，会得到相同的结果。不过从实际情况看，做到完全相同还是有些难度的。以 Unity 引擎为例，一方面，各个脚本之间 Start、Update 等生命周期函数的调用顺序不确定；另一方面，使用 Physic 物理系统也不保证是确定性模拟。比如一个单位的坐标偏差了 0.01 导致技能未能击中目标，那么这个目标的血量判断就会受到影响。如果这个目标的行为受到血量的影响，那么最终结果会完全不同。因此让每个客户端计算结果完全相同不是一件容易的事。一些常见的注意事项如下。

◎ 不使用浮点数，用整数代替。
◎ 不同客户的同步频率要保证一致。
◎ 随机种子相同，并自定义接口防止其他公用系统干扰。
◎ 使用排序容器，保证遍历顺序。
◎ 逻辑显示分离。
◎ 使用补间过渡，调整速率，掩盖卡顿。

除了一致性的难点，帧同步还需要解决流畅度的问题。由于通过网络传输过来的数据一定会慢于本地，而我们又希望在相同的时刻输入信息，因此就会引发等待，反映到用户体验就是不流畅。

优化方法有很多，例如，在帧同步游戏中，由于广播的频率非常高，因此每次广播的数据就要足够小，这样可以节省很多消息处理的时间。对于消息，可以将需要所有客户端同时发生的内容提前广播给其他用户，采用时钟同步。客户端逻辑先行，显示通过平滑追赶的方式处理。很多改进是体验优化的范畴，需要结合具体游戏进行。

在传输层，移动端的同步建议使用 UDP 作为传输协议。TCP 为了保证传输的可信性，很多机制不太适合波动较大的移动网络。在弱网络环境下，UDP 的 RTT 几乎不受影响，而 TCP 的

RTT 波动比较大，特别是在丢包重发时影响比较明显。虽然使用 UDP 会引入丢包、乱序的问题，但可以通过冗余的方式来解决这个问题。比如每帧三个数据包，实际上是包含了过去两帧的数据，也就是每次发三帧的数据来对抗丢包。

总结

本章我们从不同的侧面了解了常见的通信方式与数据同步方式。并在此基础之上，粗略地分析了状态同步与帧同步的差别。这部分内容需要大量的实践来进行测试和验证。只有经历实际上线项目的历练，才称得上是成熟的同步框架。

扩展问题

请问帧同步中的锁帧（Lockstep）是指什么（[:lockstep]）？

第 3 章
辅助系统

辅助系统在游戏开发过程中十分重要，它不是不可或缺的部分，却可以在遇到瓶颈时，提供有力的支持。下面我们就从几个方面看看常见的辅助系统及其解决的问题。

3.1 有限状态机

请简述有限状态机是什么，并尝试实现二段技能（超时会影响技能衔接）的状态机。

问题分析

有限状态机的英文缩写为 FSM（Finite State Machine），也可称为状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的模型。所谓有限，指的是状态有限，所有的状态可枚举出来。它在游戏中运用非常广泛，最常见的就是用于控制角色的状态。

在动手编写 FSM 之前，首先要明确以下两个基本概念。

◎ 状态（State）：每一个状态都是稳定的，如果不满足转移条件，则状态机会一直处于状态中。
◎ 转换（Transition）：状态之间通过转换连接，每个转换中可以包含不同的条件。

接下来就可以开始动手制作状态机了。状态机的设计是一个逻辑梳理的过程，建议拿出纸笔画图制作。先将人物所有的状态列出来，然后再用转换连接它们，一个角色的状态机可能如图 3.1 所示。

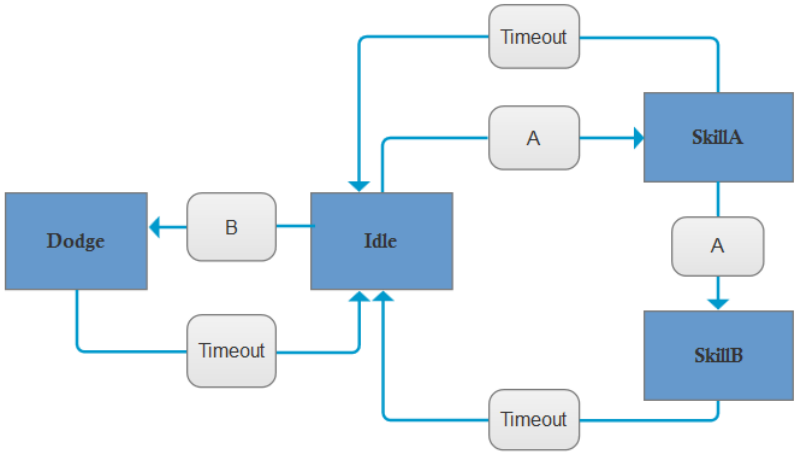


图 3.1

在图 3.1 中，角色正常会处于等待状态。当按下键盘中的 B 键时，角色进入躲闪状态，超时后返回。当按下键盘上的 A 键时，释放技能 A，如果在未超时的状态下再次按下 A 键，则触发二段技能 B。在技能超时后，自动回到等待。

代码实现

下面通过代码实现图 3.1 的核心功能。创建 FSMTest.cs 脚本，在脚本中编写如下代码：

```
public enum TRAN_INPUT
{
    BUTTON_A,
    BUTTON_B,
    TIME_OUT,
}

public interface State
{
    State HandleInput(TRAN_INPUT input);
    void EnterState();
}
```

其中，TRAN_INPUT 为转换的条件类型，HandleInput 是转换判断函数，EnterState 是进入 State 时需要触发的处理。

接下来就分别实现状态。每个状态都是一个实现 State 接口的子类。对照状态图，添加如下代码：

```
//站立状态
public class IdleState: State
{
    public State HandleInput(TRAN_INPUT input)
    {
        if(input == TRAN_INPUT.BUTTON_A)
        {
            return new SkillA();
        }
        else if (input == TRAN_INPUT.BUTTON_B)
        {
            return new DodgeState();
        }
        return null;
    }
    public void EnterState()
    {
        Debug.Log("To Idle State");
    }
}

//翻滚状态
public class DodgeState: State
{
    public State HandleInput(TRAN_INPUT input)
    {
        if(input == TRAN_INPUT.TIME_OUT)
        {
            return new IdleState();
        }
        return null;
    }
    public void EnterState()
    {
        Debug.Log("To Dodge State");
    }
}
//技能 B
```

```
public class SkillA: State
{
    public State HandleInput(TRAN_INPUT input)
    {
        if(input == TRAN_INPUT.BUTTON_A)
        {
            return new SkillB();
        }
        else if(input == TRAN_INPUT.TIME_OUT)
        {
            return new IdleState();
        }

        return null;
    }

    public void EnterState()
    {
        Debug.Log("To SkillA State");
    }
}

//技能 B
public class SkillB: State
{
    public State HandleInput(TRAN_INPUT input)
    {
        if(input == TRAN_INPUT.TIME_OUT)
        {
            return new IdleState();
        }
        return null;
    }

    public void EnterState()
    {
        Debug.Log("To SkillB State");
    }
}
```

实现了状态转换后，再编写一个 FSM 对外的接口类，将复杂的状态对外隐藏起来，添加如

下代码：

```
public class FSM
{
    State CurrentState;

    public FSM (){
        CurrentState = new IdleState();
        CurrentState.EnterState();
    }

    public void HandleInput(TRAN_INPUT input)
    {
        State newState = CurrentState.HandleInput(input);
        if(newState != null)
        {
            CurrentState = newState;
            CurrentState.EnterState();
        }
    }
}
```

通过这样的封装，外部只需调用 HandleInput，并将输入信号量传入即可，无须关心具体的跳转逻辑。

最后，我们还要编写测试类，新建 FSMTest.cs 文件，添加如下代码：

```
public class FSMTest : MonoBehaviour
{

    public FSM fsm ;

    void Start()
    {
        fsm = new FSM();
    }

    void Update()
    {
        if(Input.GetKeyUp(KeyCode.J)){
            fsm.HandleInput(TRAN_INPUT.BUTTON_A);
            StopAllCoroutines();
```

```
            StartCoroutine(autoTimeOut(2));
        }
        else if(Input.GetKeyUp(KeyCode.Space)){
            fsm.HandleInput(TRAN_INPUT.BUTTON_B);
            StopAllCoroutines();
            StartCoroutine(autoTimeOut(1));
        }
    }

    IEnumerator autoTimeOut(float sec){
        yield return new WaitForSeconds(sec);
        fsm.HandleInput(TRAN_INPUT.TIME_OUT);
    }
}
```

在这个类中，通过 Input.GetKeyUp 获取键盘的输入，控制状态机的转换。另外，笔者通过协程实现了一个简易的超时判断，在实际项目中，超时判断通常会更加复杂。

将脚本挂载到场景的 GameObject 中，运行游戏可以看到状态进入 Idle 的输出，按下对应按键即可看到状态转换的输出，如图 3.2 所示。

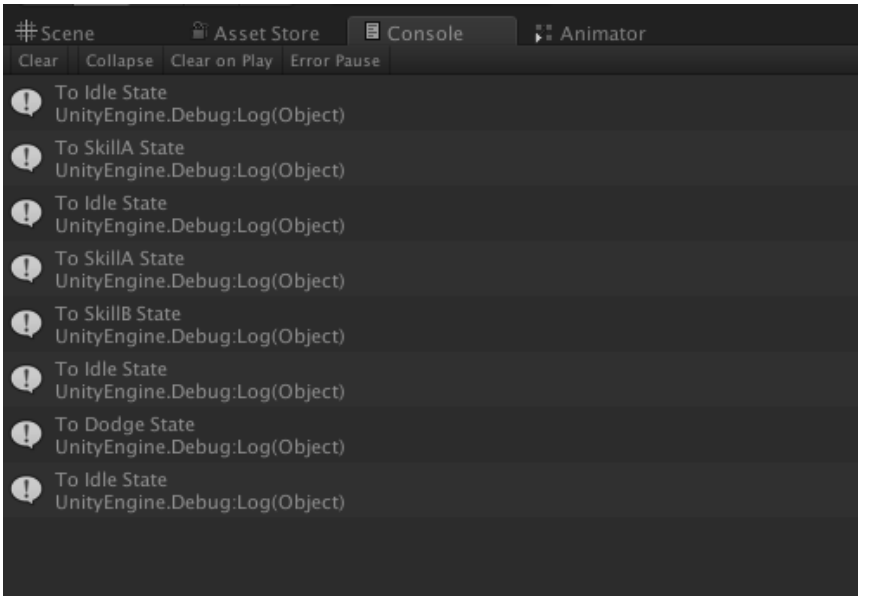


图 3.2

Animator

在 Unity 3D 中，对于动画控制可以使用 Animator。在这种编辑模式下，我们可以在图形化界面中灵活地更改状态机的连接方式，控制转换条件。Animator 还支持子状态机等整理形式，可以简化状态图的复杂程度。另外，在游戏运行的过程中，也可以看到 Animator 的运行变化，是强大的图形化工具。例如，角色的击飞、击退、击倒，以及空中连击的动画状态图等，可以按照图 3.3 的方式连接。

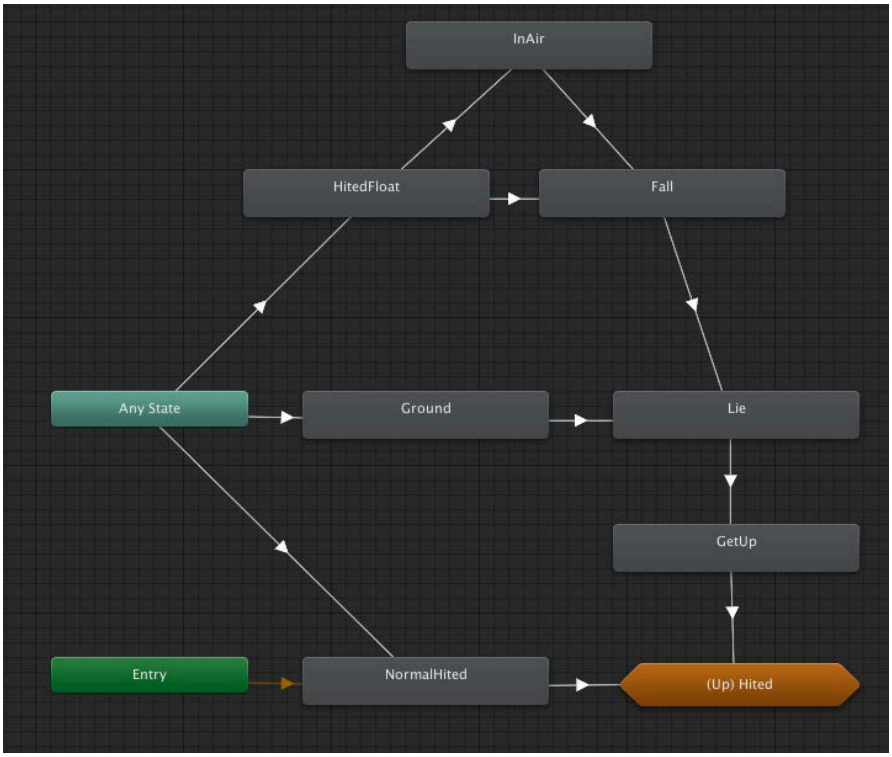


图 3.3

总结

本篇通过代码实现了一个简单的技能状态机，讲解了 FSM 的常见用法。另外，还简单介绍了其在 Animator 中的应用。由于 Animator 的使用比较简单，因此很容易找到讲解资料，另外去看官网文档也是个不错的选择（[:animatordoc]），这里就不过多介绍了。

扩展问题

既然有了 Animator，还需要实现逻辑的状态机吗（[:logicfsm]）？

3.2　脚本系统

我们都知道目前在游戏中最常用的脚本系统是 Lua，请分析除热更新支持外，使用脚本系统有何优劣？如果让你自己设计一套脚本系统，应该注意什么？

问题分析

首先明确什么是脚本。在 Unity 3D 中，我们使用的 C#脚本并不属于脚本系统中的脚本范畴。这里的脚本的使用方式与字节码（ByteCode）类似，是在运行时通过虚拟机系统（Virtual Machine）来载入文件，动态更改逻辑影响游戏行为。从本质上讲，它是把行为控制部分从编码层转向了数据层。通过脚本系统，逻辑与行为更灵活地被外部数据控制，程序只负责执行。

在脚本系统中，一条命令被可优化的底层操作定义，一系列这样的命令被编码成字节流。虚拟机通过中间层依次执行命令。它的优点是可以灵活地定义行为，并动态改变逻辑。缺点也显而易见，由于需要 VM 的支持，因此这种模式的效率不可能太高。Lua 就是这样的一种字节码，它有性能优异且轻量的 VM，并提供多个平台的支持。有大量的商业游戏使用它作为脚本引擎语言。

魔兽世界 UI 结构

很多人知道 Lua 语言是从魔兽世界开始的。在魔兽世界的聊天中输入：

```
/script print("Hello,WoW")
```

即可运行 Lua 脚本，输出“Hello WoW”。通过这样的方式，玩家几乎可以通过输入命令做游戏中的任何事。于是有很多人利用这点，制作了很多外挂，来实现自动化游戏，迫使暴雪公司不得不限制接口的功能。

与只在开发过程中使用的传统脚本系统不同，魔兽世界将脚本的控制权交给了玩家，使得每个人都可以自定义游戏。例如，通过下面的脚本可以关闭小地图：

```
-- close minimap
/script UIFrameFadeOut(Minimap,1)

-- open minimap
/script Minimap:SetAlpha(1)
```

通过几行脚本，就可以轻易地更改界面布局。我们也可以添加自己想要的功能，如监听游戏事件、弹出提醒框等。

由于 MMO 类型的游戏有着丰富多变的玩法，因此无论是游戏的开发，还是游戏的运行，引入脚本系统都可以大大提升游戏的灵活性。

Dota2 角色的 AI 结构

对于系统或 UI 界面没那么复杂的竞技类游戏，脚本系统也有它存在的意义。DOTA2 的 AI 就是这样一套脚本系统，从角色选择、装备合成、技能释放等初级的操作，到分路选择、团战时机判断、角色间配合等团队型决策，都可以通过 Lua 脚本自由定制。例如，截至本书编写时间，在 DOTA2 的 7.x 版本中，通过更改 hero_selection.lua，即可控制对战电脑的阵容：

```
function Think()
    if ( GetTeam() == TEAM_RADIANT )
    then
        print( "selecting radiant" );
        SelectHero( 0, "npc_dota_hero_antimage" );
        SelectHero( 1, "npc_dota_hero_lina" );
        SelectHero( 2, "npc_dota_hero_sven" );
        SelectHero( 3, "npc_dota_hero_bloodseeker" );
        SelectHero( 4, "npc_dota_hero_crystal_maiden" );
    elseif ( GetTeam() == TEAM_DIRE )
    then
        print( "selecting dire" );
        SelectHero( 5, "npc_dota_hero_drow_ranger" );
        SelectHero( 6, "npc_dota_hero_earthshaker" );
        SelectHero( 7, "npc_dota_hero_juggernaut" );
        SelectHero( 8, "npc_dota_hero_mirana" );
        SelectHero( 9, "npc_dota_hero_nevermore" );
    end
end
```

如果想更改角色的装备合成，也可以重写每个角色自己的 Lua 控制文件。例如，要想更改

“流浪剑客斯温”的出装，就需要重写 item_purchase_sven.lua 文件：

```
local tableItemsToBuy = {
    "item_tango",
    "item_clarity",
    "item_flask",
    ----------------------
    "item_blink",
    ----------------------
    "item_boots",
    "item_blades_of_attack",
    "item_blades_of_attack",
    ----------------------
    "item_ogre_axe",
    "item_quarterstaff",
    "item_sobi_mask",
    "item_robe",
    ----------------------
    "item_ogre_axe",
    "item_mithril_hammer",
    "item_recipe_black_king_bar",
    ----------------------
    "item_lifesteal",
    "item_mithril_hammer",
    "item_reaver",
    ----------------------
    "item_reaver",
    "item_vitality_booster",
    "item_recipe_heart",
};

// PurchaseLogic
//......
```

可以看到，通过更改角色的出装表，定制购买道具的顺序，即可个性化地控制角色出装。不做 AI 的深入讨论，有兴趣的读者可以自行上网查阅相关资料。

在 DOTA2 的套脚本架构中，最值得借鉴的是，它的主逻辑并不是使用 Lua 制作的，而是使用效果更高的 C++编写的。对于不需要定制的通用功能，采用高效的实现方式是明智之举。如果需要定制角色行为，就创建一个以角色名为后缀的 Lua 文件。例如，对于角色 Lina，可以

创建 item_purchase_Lina、ability_item_usage_Lina.lua 等。在加载游戏时，会检测相应脚本，如果存在名称匹配的脚本，就将其载入运行对应逻辑，否则使用默认的实现，最大化地实现性能的优化。

自设计脚本

既然 Lua 这么强大，是否就不用自己编写脚本系统了呢？理论上说，Lua 可以应付各种情况，没必要自己写一个解释器，重新造一个“轮子”。但这里有个脚本开发的友好性问题。使用脚本的人不一定是熟悉程序的人，因此能正确地编写 Lua 脚本是有一定门槛的。如果是自定义的脚本，就可以将语法设计得更具体一些，贴近自然语言，以降低使用门槛。值得强调的是，设计脚本语法并不是没有成本的，我们在动手之前，需要衡量工作量与工期。下面就以 CG 动画脚本为例来说明。

在游戏中制作 CG 过场动画时通常需要策划和美术多人协助完成。对于 Unity 3D 来说，可以找到制作 CG 事件的插件，但从实际推行的效果来看，学习成本并不低。对于常规的场景分镜，其实并不一定要有特殊的动画效果，因此采用有限语法的脚本完全可以满足需求。在制定语法时，完全可以使用中文，以降低英语门槛，防止出现拼写等错误。一个简单的脚本可以是下面这样的。

```
场景：3-1
描述：两人对于需求更改的争执
角色：程序员[替代标志(A),位置(2,2,3)],策划人员[替代标志(B),位置(2,2,-4)]

开始
摄像机：过肩拍[目标(B),距离(10)]
摄像机：淡入[时长(1)]
移动：B[位置(2,2,0)]
等待
转向：A[目标(B)]

下一段
摄像机：回应
对话：B[文字(已经开始做宠物模块了吗？)]
对话：A[文字(刚刚制作好界面，都快累吐血了)]
对话：B[文字(那真是不太好，由于之前的界面感觉不对，现在已经有新版了)]
对话：A[文字(没时间重做了，这版先这样吧，我先写功能)]
对话：B[文字(恩，也行。这个新版感觉也不太对)]
```

```
对话：A[文字(我就知道...)]
摄像机：淡出[时长(1)]

下一段
屏显：文字(一周之后),时长(2)
摄像机：淡入[时长(1)]
对话：B[文字(告诉你个好消息，宠物模块用处不大，这个功能可以去掉了)]
摄像机：仰拍[目标(A),距离(5)]
对话：A[文字(我刚做完！)]

结束：进入战斗
```

通过上面的脚本定义方式，能够兼顾可读性与实用性。实际应用中，可能会比这个脚本的功能参数更多，例如播放声音、特效等。但总体来讲，纯文本的内容易于修改，而简明的语法可以使得 CG 内容更直观。为了避免出错，我们甚至可以编写一个语法校验器，来检查是否在特定位置出现非关键字。VM 方面，每行是一条命令，中括号中是参数，小括号中是参数值。在运行时，每次读入一行，并转义成对应的逻辑执行。其中需要约定一些关键值，比如摄像机的回应模式，是指两个对话的人依次出现在屏幕的中心，并且脸部朝向相对，效果可以参照新闻采访。诸如此类的定义都需要结合项目实际需求制定。

总结

通过分析脚本系统的内在原理，我们分析了魔兽世界与 DOTA2 对应系统的优劣，也探讨了是否有必要设计自己的脚本语法。虽然目前大多数游戏使用脚本系统是为了能够热更新，但相信随着脚本系统的广泛应用，它会因自身的优势被用于更多的场景。

第二部分

艺术资源

从某种角度说，游戏是让美术资源“活”起来的艺术。在第 4 章，我们会看到游戏中场景的资源，以及它们的使用方式。考虑到本书的读者大多是程序员，因此这里主要概述美术基础知识。

在第 5 章，我们会进一步谈到需要程序员参与的美术效果的实现。虽然这些应由 TA（Technical Artist）来负责，但更多情况下，团队中没有能胜任的角色，因此只能由程序员侧面推动。

当游戏规模逐渐扩大后，美术资源管理带来的问题也随之出现。在第 6 章，我们将集中探讨如何实现半自动化的资源管理。

第 4 章
资源分类

本章主要介绍贴图、材质、动画等美术资源，以及它们该如何制作和使用。

4.1 贴图种类

请简要介绍一下项目中用到的材质相关的贴图种类及其作用。

问题分析

在实际项目中，为了达到千变万化的艺术效果，我们会使用图形数据——贴图。随着贴图的增加，美术可以制作出更多变的效果，但过多的贴图也会增大内存的负荷。因此，贴图的选取是一个权衡的结果，要能在程序运行效率允许的情况下，给美术最大的发挥空间。

贴图主要分为两大类：图像贴图和计算贴图。这两种类型最大的区别在于，图像贴图通常是由艺术家来绘制的，用于表现物体的性状；计算贴图则用来辅助计算，以求达到特殊效果，通常由数学算法生成。

下面以 Unity-Chain 为例，一起来看一下常用的贴图。

固有色贴图

固有色贴图（Diffuse）是物体在白色阳光下所呈现的固有颜色。固有色，可以准确地控制色相，使角色呈现一个合理的饱和度。固有色贴图也能体现模型的一部分质感，如图 4.1 所示。

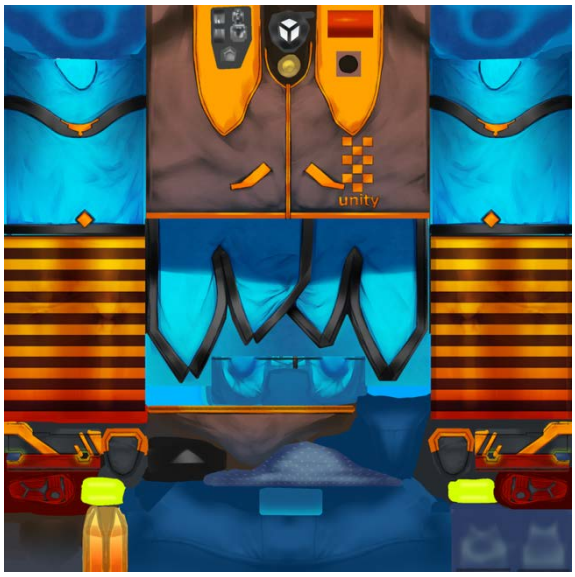


图 4.1

透明贴图

透明贴图（Alpha）是一张只由黑白像素构成的灰度图，通常使用它来实现磨砂或半透明效果。在这种图中，纯黑色代表全透明，纯白色代表不透明。

由于透明贴图是只占有一个通道的灰度图，因此有时也将固有色贴图的 A 通道用来存储透明贴图。当然如果有其他贴图的闲置通道，也可以在一起合并使用。

由于像素精度的问题，在使用透明贴图时，有时会有半个像素的白边或者是白色锯齿。常规的解决方法是调整半透明物体的边缘羽化，在材质中调整透明裁切（AlphaCutOff），只有贴图的透明度大于 AlphaCutOff 时才可以绘制。

AO 贴图

AO 贴图（Ambient Occlusion）也被称为环境光吸收贴图，主要用来制作模型的阴影。为了不受外界光照变换的影响，AO 贴图采用“吸收”光线的计算方法，模拟全局光照效果来改善阴影的细节。使用这种技术可以优化墙角阴影浅淡、缝隙阴影发虚等问题，以加强明暗对比，增加空间的层次，在灯光复杂的场景中比较有用。

因为 AO 贴图不受光照角度的影响，所以可以认为几乎是静态效果。为了节省贴图占用的

内存，在不做其他用途时，可以在 Photoshop 中将 AO 贴图与固有色贴图合并为一张。

高度贴图

高度贴图（Height Map）是一个记录物体表面高度的灰阶图，黑色代表高度的最小值，白色代表最大值。它主要用于加强法线贴图的表现效果，例如视差技术或浮雕效果。高度贴图也常用于地形的控制。在地形编辑器中，编辑出的山谷和地形起伏都存储在高度图中。当生成地形高度图时，设计者可以将不同高度的区域分通道存储以保证精度。

大多数情况下，在用作常规表面高度变化时，高度贴图都会被转化为法线贴图，以提高运行效率。

法线贴图

法线贴图（Normal Map）作为模型表面的扩展，包括了模型像素点的高度值。它是次时代游戏的标配，可以模拟起伏的或不规则的表面，如图 4.2 所示。它虽然会大幅增加渲染时间，但能够使几千面的模型拥有几万，甚至几十万面的视觉效果。另一方面，它虽然不会更改模型形状，却可以精确地表现模型表面的光照细节，大大降低了运算消耗。

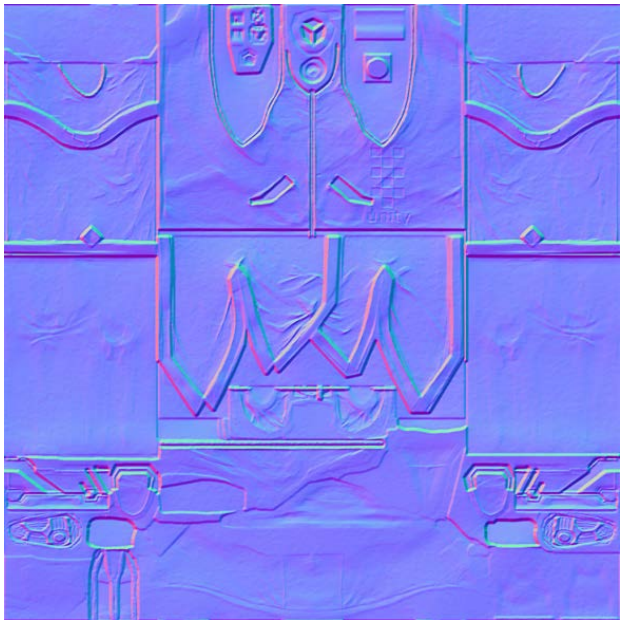


图 4.2

法线贴图是 RGB 贴图，红色通道编码法线方向的左右轴，绿色通道编码法线方向的上下轴，蓝色通道编码垂直深度。一般来说，法线贴图是在切空间下，直接把高度图（Height Map）转换成一张法线图（Normal Map）。使用常见的 3D 模型工具都可以制作法线。

高光贴图

高光贴图（Specular）主要用来控制模型的质感。皮具、布料、金属的高光表现各不相同，即使是相同的材料也会因为磨损程度的不同而呈现不同的高光效果，这些细微的差别都可以通过高光贴图体现出来。

自发光贴图

自发光贴图（Glow）用来标定模型发光的区域。贴图的白色区域渲染为完全发光，黑色区域则保持原样，灰色区域渲染为部分自发光。需要注意的是，自发光区域不应受灯光的影响和阴影的影响。

反射贴图

反射贴图（Reflection）标注在有光泽物体表面反射的效果。为了使反射贴图获得较好的渲染效果，贴图应具有较高的分辨率。

渐变贴图

渐变贴图（Ramp）常用来辅助计算光照角度引起的不同变化。通过创建高度自定义的渐变，或使用双向反射贴图，可以模拟复杂的光照效果。

噪声贴图

噪声贴图（Noise）是通过噪声函数计算出的贴图，如图 4.3 所示。通过调整混合算法，它可用于模拟云朵、火焰等自然现象，或者大理石、木材等天然材质。我们可以使用 Noise Graph 软件生成需要的噪声图。在此软件中，我们可以生成数值噪声、梯度噪声，以及数值+梯度噪声。

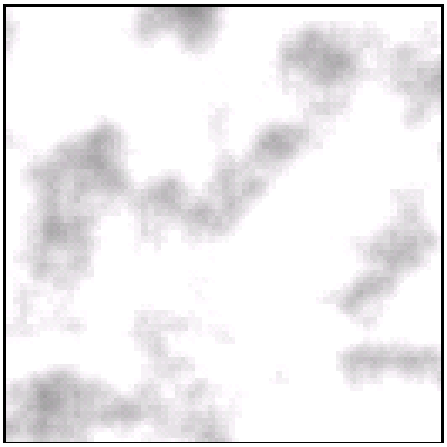


图 4.3

总结

虽然本文提及了很多贴图，但项目中并不需要全部使用。次时代模型通常会使用固有色贴图、法线贴图、高光贴图和自发光贴图。最常用的贴图主要是固有色贴图、法线贴图和渐变贴图。

扩展问题

模型的贴图是越多越好吗（[:texturemore]）？

4.2 材质效果

请问常见的材质效果有哪些，并简述如何在 Unity 3D 中实现对应效果。

问题分析

我们知道，物体的材料会影响人对物体的判断。比如有塑料感的物体就会给人很轻的感觉，而拉丝的反光效果就会给人顺滑的感觉，等等。为了达到不同的表现效果，美术人员在制作过程中会采用各种辅助方式。

从效果的实现原理的角度来看，可以按照是否使用 PBR（Physically-Based Rendering）作为标准划分。即使用 PBR 流程组织美术资源的为一类，其他的为另一类。

对于常规美术流程，即 4.1 节贴图分类中提到的方式，也是目前绝大多数游戏采用的资源组织方式。它底层基于兰伯特模型（Lambert），通过点乘的方式来估计照度，并通过功能拆分对应的贴图，功能之间互不干扰，给美术更强的控制能力。这既是优点也是缺点。优点是由于这种标准已经在游戏行业中广泛使用，因此并不难找到能够上手的美术人员。缺点是由于美术人员对效果有很强的控制能力，因此整个效果的调整对美术人员的能力要求很高。

另一方面，PBR 技术弥补了这一缺陷。它根据能量守恒的原理来计算光强，Unity 3D 中默认的 StandardShader 就是基于这种方式实现的。它的好处是只要按照材质参数表，即可配置出效果正确的材质。图 4.4 是引自 Unity 3D 官方文档中提供的参数对照图。

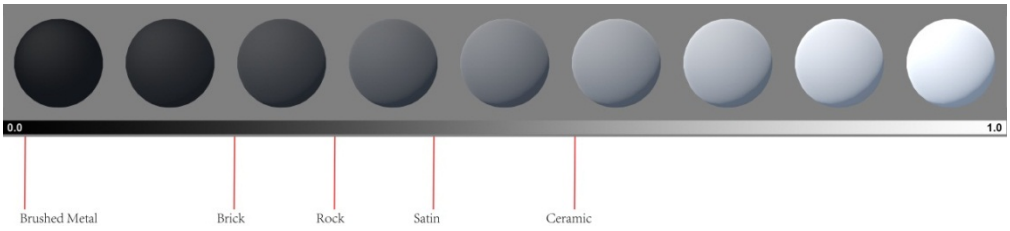


图 4.4

在这种流程下，对美术人员的能力要求略低。但由于目前使用这套流程的团队不多，并且美术人员还不太熟悉这种模式，因此反而在制作的过程中造成了困扰。另一方面，这套资源流程对引擎也有些要求。由于材质效果依靠计算，因此美术人员在 PBR 工具中制作出的效果，与导入引擎后运行的效果会有所不同。不论是 Unity 3D 还是 Unreal，都需要对 PBR 对应的 Shader 或材质参数做适当修正，以保障效果的一致性，这对技术层面也提出了更高的要求。因此，目前基于 PBR 流程研发的游戏数量增长得十分缓慢，相信随着技术的进步会逐渐成为主流。

对于额外的材质效果，不太介意是否基于 PBR。虽然乱改贴图会影响 PBR 的效果，但常规材质效果都已经包含在标准流程中了。如果在 PBR 流程中需要加入新的特殊效果，则可以在颜色计算好后再叠加。因此下面的这些效果在两种美术流程中都可以使用。

因为需要比较材质效果，所以这里选取了 Unity-Chan 和 Space Robot Kyle 的资源作为效果参考。资源的获取方式可参考序言中的相关链接，后面代码片段的完整版本也在资源包中。

边缘光

无论哪种渲染模式，光感的效果都是核心。对于卡通渲染最大的特点就是利落的边缘光或

阴影。在 Unity-Chan 中我们可以明显看到类似的效果，例如，角色袖子上的边缘光效果如图 4.5 所示。

图 4.5

把袖子上的边缘光去掉后，整个衣服的效果就完全不同了，效果如图 4.6 所示。

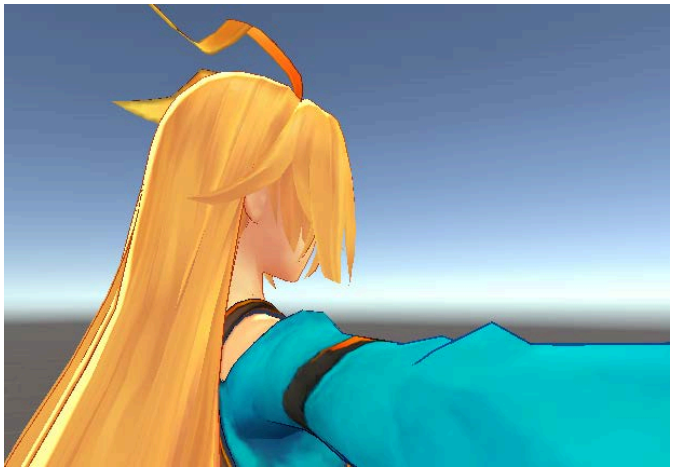


图 4.6

边缘光的计算并不复杂，核心是使用一张渐变图做控制，然后根据照度在图上做强度的采样，相关代码如下：

```
float_t rimlightDot = saturate( 0.5 * ( dot( normalVec, i.lightDir ) + 1.0 ) );
falloffU = saturate( rimlightDot * falloffU );
falloffU = tex2D( _RimLightSampler, float2( falloffU, 0.25f ) ).r;
float3_t lightColor = diffSamplerColor.rgb; // * 2.0;
combinedColor += falloffU * lightColor;
```

高光效果

另一个比较重要的光效是高光，物体的质感几乎完全取决于高光的效果。我们可以看到在 UnityChan 的袖口、胸、马甲的兜附近都有明显的高光，效果如图 4.7 所示。

图 4.7

如果去掉高光，袖子就会显得没有层次，马甲也松松垮垮，如图 4.8 所示。

图 4.8

具体代码逻辑是通过一张高光图记录反光的倍率与色值，然后再与标准高光计算出的结果相乘。核心内容如下：

```
float4_t reflectionMaskColor = tex2D( _SpecularReflectionSampler, i.uv.xy );
float_t specularDot = dot( normalVec, i.eyeDir.xyz );
float4_t lighting = lit( normalDotEye, specularDot, _SpecularPower );
float3_t specularColor = saturate( lighting.z ) * reflectionMaskColor.rgb *
diffSamplerColor.rgb;
combinedColor += specularColor;
```

渐变贴图

渐变贴图（Ramp）或者叫衰减贴图（FallOff），与前面的应用范围不太一样，它更多是以一种加强色彩控制的形式出现。例如前面提到的采样用贴图也是渐变贴图的一种，这里拿出来单独说，是因为有时候它不太明显，效果会与固有色贴图混在一起，让人以为是画在固有色中的效果。最明显的例子就是角色的皮肤。在 Unity 的皮肤效果中，使用了一张长方形的渐变图，如图 4.9 所示。

图 4.9

在这张图的影响下，皮肤的效果如图 4.10 所示。

图 4.10

如果去掉渐变效果，皮肤就缺少了层次感，如图 4.11 所示。

图 4.11

实现方面，使用视线与平面法线点乘的值作为标准，在渐变图中采样。最终，通过渐变图的透明通道作为插值标准，与固有色进行混合。核心代码如下：

```
float_t normalDotEye = dot( i.normal, i.eyeDir );
float_t falloffU = clamp( 1 - abs( normalDotEye ), 0.02, 0.98 );
float4_t falloffSamplerColor = FALLOFF_POWER * tex2D( _FalloffSampler,
float2( falloffU, 0.25f ) );
float3_t combinedColor = lerp( diffSamplerColor.rgb, falloffSamplerColor.rgb
* diffSamplerColor.rgb, falloffSamplerColor.a );
combinedColor = diffSamplerColor;
```

自发光

自发光效果（Glow）其实算是后处理效果的一种，并不属于材质层。不过它的贴图和参数配置都可以放在材质上，因此也算与材质有关，因而把它放在这里一起说。

这种效果最大的优势是，可以使画面看起来有更多的光源，这在很多特效中都会用到，当然用在角色身上的例子也比比皆是。这里使用 Kyle 做了一个简单的例子，来说明效果。

首先引入 Unity 3D 的 Effect 资源包，接着新建一个场景，在摄像机上挂载 Bloom 脚本，并做如图 4.12 所示设置。

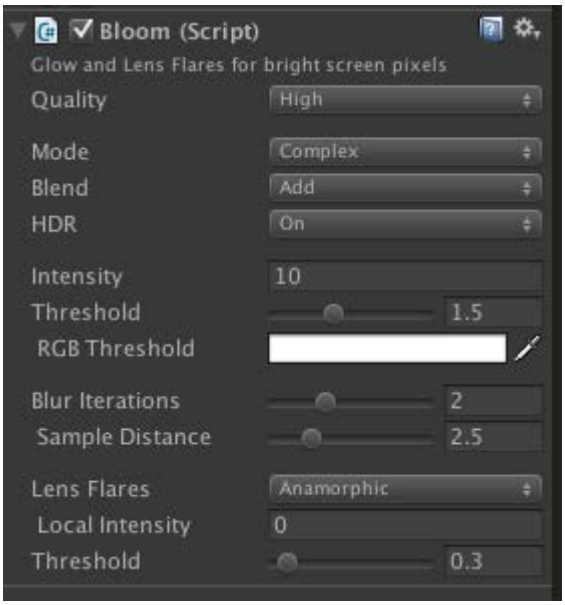


图 4.12

接着使用绘图软件，绘制一张用于加强颜色的遮罩贴图，如图 4.13 所示。

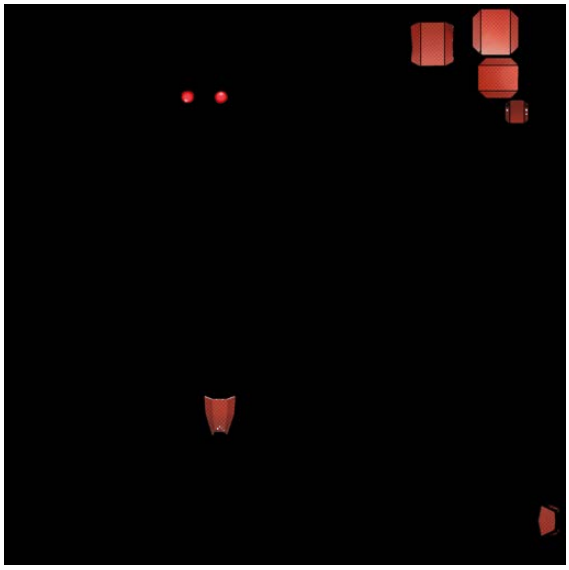


图 4.13

最后创建一个新的使用 StandardMaterial 的材质，将其自发光（Emission）参数更改为如图 4.14 所示的值。

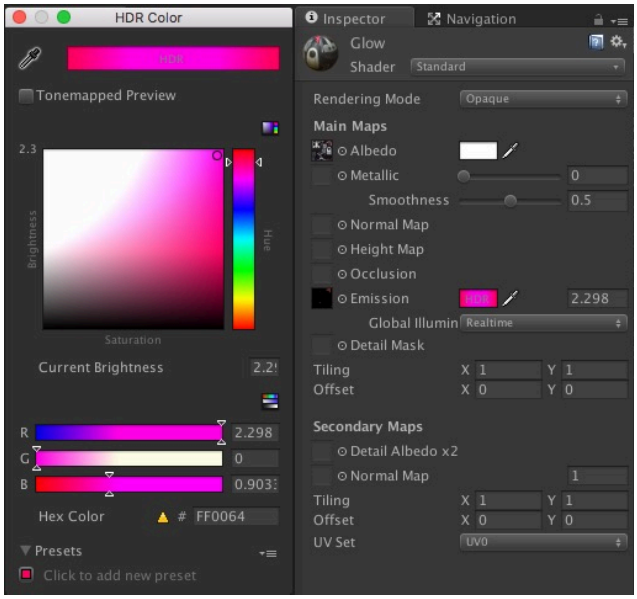


图 4.14

调整摄像机角度，即可看到如图 4.15 所示的效果。

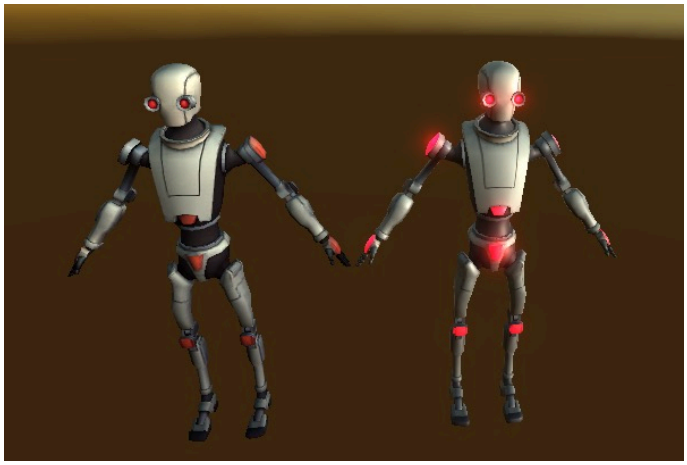


图 4.15

可以看出，使用自发光对画面效果有很大的提升。当然这些提升也是有代价的，最明显的就是资源量变大，每增加一张贴图，就代表增加了美术人员的工作量。另外，由于它的底层原理是后处理（Post-Effect），因此也会增加运行时的内存消耗。在后面的内容中，我们会介绍如何定制自己的辉光效果。

材质捕获贴图

材质捕获贴图（Material Capture），通常被称为 MatCap，在很多 3D 软件中都可以生成。它最大的特点是能真实地表现反射效果，而不需要在场景中提供对应的灯光。从本质上讲，它是将所有的光照信息都存储到了贴图中的。代码实现也不复杂，在运行时，只需将法线从模型空间转到视口空间，再将对应 UV 映射到贴图上就可以了。

在 AssetStore 中有名为“Free MatCap Shaders”的插件，它比较直观地展示了这项技术的效果，有兴趣的读者可以自行下载查看。效果如图 4.16 所示。

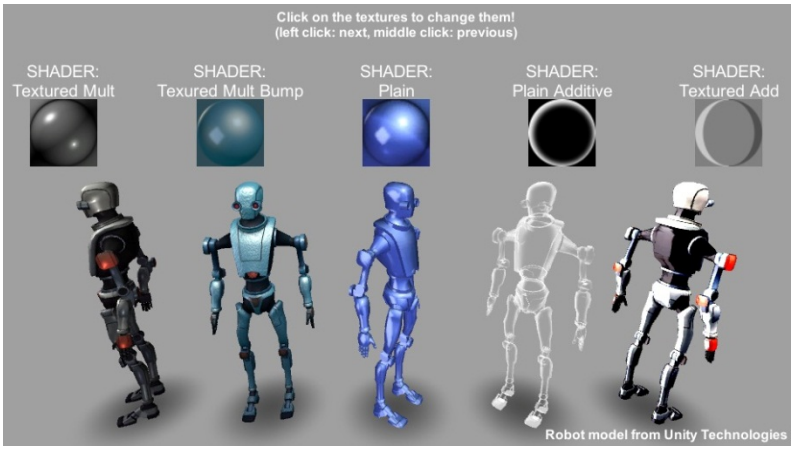


图 4.16

其中展示的模型依然为 kyle，上面的方形贴图即为 MatCap 贴图。

总结

本节主要介绍了常见的材质效果，并结合代码和实现方式进行了分析。需要注意的是，上面只是说了一些主流的材质效果，真实项目中的材质可能会更多。正是因为美术效果层出不穷，所以我们才能看到各种令人惊叹的游戏。

扩展问题

要应用多种效果势必会增加贴图的数量，有什么办法进行优化吗（[:texturepath]）？

4.3　动画分类

请问游戏中常见的动画表现方式有哪些，请简述对应的技术细节？

问题分析

游戏从本质上讲，是通过静画、文字、动画表现故事和反馈的产物。除去纯文字游戏与图片类游戏，大多数游戏都广泛使用了动画技术。

从动画的本质上讲，它利用了人的视觉暂留（Persistence of Vision）。当物体在快速运动时，人眼所看到的影像消失后，仍能继续保留影像约 0.1~0.4 秒。游戏中动画也利用相同的原理，在每帧更新时刷新图像。根据不同的表现内容，出现了多种资源的组织形式，它们的不同之处在于，如何在简化制作流程的同时，提供表现力更强的画面效果。

简化制作数据通常是使用插值技术。插值是指对离散的数据进行处理，进而得到一个连续的结果。在动画方面，通常的应用场景是，对定义边界值的中间细节内容进行填充。从广义上看，所有的关键帧应用属于这个范畴。通过更改不同的插值函数，我们可以得到不同的结果。常见的插值函数有 Hermite 曲线、Catmull-Rom 样条曲线或 Bezier 曲线等。本节主要讨论动画，与曲线相关的计算弧长、速度控制等问题已超出讨论范畴，感兴趣的读者可自行查阅相关资料。

下面我们一起来看看常见的动画表现方式有哪些。

序列帧动画

序列帧动画也被称为精灵动画（Sprite Animation）。在 FC 时代，几乎所有的游戏都是使用这样的方式制作的。简单来说，就是一系列动画的图片，按照次序播放，最终形成的动画。例如，图 4.17 引自 2D Roguelike，它就是一个序列帧图。

图 4.17

这种技术需要美术人员将动画中每帧的内容画出来，因此最大的缺点是实现成本大。另一方面，相比于骨骼或粒子的实现方式，帧动画采用了空间换时间的策略。大幅简化计算量的代价是高内存占用，因此需要注意美术资源的控制。帧动画也会用于特效的表现，例如火焰、电流等效果，这种情况下要注意贴图质量与性能瓶颈之间的权衡，适当地采用其他方式实现。

另外也有一种叫作“3D 转 2D”或者说“2.5D”的技术，也是利用了序列帧动画。它会将 3D 软件中制作的模型或动作渲染成 2D 的帧动画，依然采用建模、绑骨、蒙皮、k 动作的流程，最终导出序列帧图片。在这种工作流程下，可以将美术绘制动画帧制作流程大幅简化，也是比较主流的做法。使用这种技术最出名的游戏是《部落冲突》（Clash of Clans），其所有的动画都是这样实现的，以降低战斗过程中 CPU 的压力。

骨骼动画

3D 模型只能通过骨骼的蒙皮来做动画，因此当谈及骨骼动画时，通常指 2D 层面。这种技术的核心是将图片绑定到骨骼上。制作动画时，直接操作骨骼。只要导出骨骼数据，运行时关联图片文件即可播放动画。骨骼动画在简化了工作流程的同时，还可以支持换装系统。目前有两个主流的解决方案为这项技术提供支持，分别为 Spine 和 DragonBones。

两种工具功能很相似，Spine 是一款收费工具，DragonBones 是免费国产工具。目前来看，Spine 的工具链更加完整，可以做分层的特效混合，对美术人员的限制更少，因为是收费软件，所有有专门的团队维护。Spine 的应用非常广泛，成功作品有很多，例如《Rise & Shine》《The

Swindle》等。也有团队使用 DragonBones，《刀塔传奇》就是使用 DragonBones 的成功案例。图 4.18 摘自 Spine 官网。

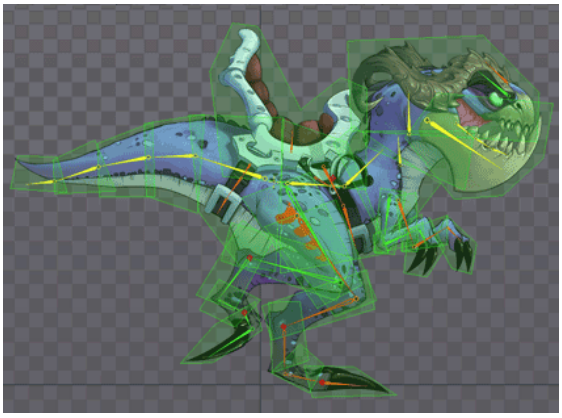


图 4.18

帧动画

通过关键帧可以控制很多元素的属性，例如 UI 移动、图片缩放、材质参数变化等。在 Unity 3D 中使用 Animation，可以指定元素的属性，然后通过插值的方式补全过渡效果。在实际项目中，这种技术被广泛地使用在 UI 动效、材质参数控制，甚至整个角色动画系统。在 2D platformer 中，角色动作就是通过帧动画实现的，如图 4.19 所示。

图 4.19

变形目标动画

变形目标动画（Morph Target Animation）是指定义了所有顶点移动方式的动画。由于数据量非常大，因此这种技术通常只用于需要控制细节的人脸表情。

蒙皮动画

通常，3D 角色都使用蒙皮动画，每个模型在绑骨之后，要为每个顶点分配对应骨头的权重，以防止动作出现撕裂模型表面的现象。因此，蒙皮的核心是权重的分配。当游戏运行时，顶点的位置是根据模型数据通过 CPU 实时计算出来的。如果是相同角色很多的场景，则可以预先将动作中顶点的每帧位置计算好，以节省运算。另一方面，蒙皮的计算最初是在 CPU 中进行的，随着技术的发展，GPU 蒙皮也流行起来。但目前不是所有平台都能很好地支持，因而使用 GPU 蒙皮需谨慎。

总结

动画大体上就分为这些种，它们之间也会混合使用。例如，在 Animation 中控制序列帧的播放，在帧动画中调整骨骼动画的材质参数等。技术是为效果服务的，为了达到表现效果，通常会权衡利弊找到最优的实现方式。

4.4 流动效果

请问 UV 滚动指的是什么？请简述如何制作一个沿模型表面流动的效果？

问题分析

在表面效果中，UV 滚动最为常见，它兼顾了多变性和低性能消耗的特点，被广泛应用于角色模型、粒子、场景中。在这项技术中，UV 使用的是纹理贴图坐标，它定义了纹理贴图上的每个点与 3D 模型的对应关系，因此，通过 UV 可以将贴图的像素点映射到模型物体的表面。滚动指的是将这种映射关系随着时间有规律的变化，可以想象到物体表面的纹理也会更改，效果有点像逐渐移动有花纹的桌布。通常我们会使用它制作一些角色身上的流光、场景中的水，或者一些特殊的粒子效果。下面就用一个简单的流动效果，演示 UV 移动的用法。

代码实现

首先我们在 3DMax 中制作一个双螺旋插片的模型，并将其导入 Unity 中。效果如图 4.20

所示。

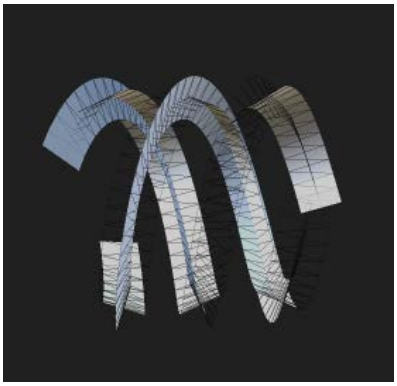


图 4.20

然后创建一个名为 UVMove 的 Shader，并编写如下代码：

```
Shader "MyShader/UVMove"
{
    Properties {
        [HDR]_TintColor ("Tint Color", Color) = (0.5,0.5,0.5,0.5)
        _MainTex ("Particle Texture", 2D) = "white" {}
        _MoveX ("MoveX",Float) = 0
        _MoveY ("MoveY",Float) = 0
    }

    Category {
        Tags { "Queue"="Transparent" "IgnoreProjector"="True"
               "RenderType"="Transparent" }

        Blend SrcAlpha One

        Cull Off Lighting Off ZWrite Off

        SubShader {
            Pass {

                CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                #include "UnityCG.cginc"
```

```
            sampler2D _MainTex;
            float4 _MainTex_ST;
            fixed4 _TintColor;
            fixed _MoveX;
            fixed _MoveY;

            struct appdata_t {
                float4 vertex : POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
            };

            struct v2f {
                float4 vertex : POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
            };



            v2f vert (appdata_t v)
            {
                v2f o;
                o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
                o.color = v.color;
                o.texcoord = TRANSFORM_TEX(v.texcoord,_MainTex) +
                             frac(float2(_MoveX,_MoveY) * _Time.y);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return 2.0f * i.color * _TintColor * tex2D(_MainTex,
                                        i.texcoord);
            }
            ENDCG
        }
    }
}
}
```

这段 Shader 中，核心部分是计算 texcoord。通过获取 UV 的滚动速度与时间相乘取小数部分，我们可以得到一个不断连续变化的 UV 值。接着创建一个使用这个 Shader 的材质，并将图 4.21 所示贴图赋值给材质。

图 4.21

最后创建一个粒子系统，调整参数如图 4.22 所示。

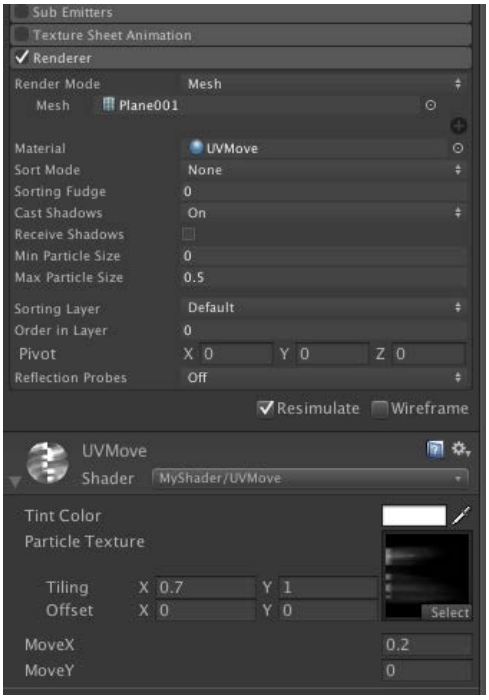


图 4.22

运行游戏，即可看到流动的效果，如图 4.23 所示。

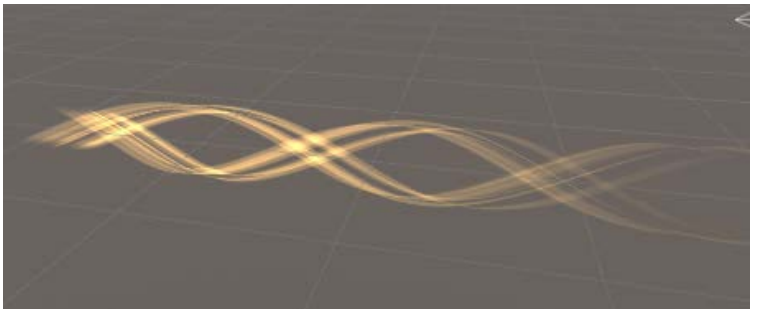


图 4.23

总结

相比于静止的 Shader，UV 滚动只是多了偏移计算的消耗，从性能上看很优秀。仅仅使用这项技术，结合不同的贴图和灵活的滚动方向，就可以做出惊艳可用的效果。

扩展问题

美术人员要想移动速度可变化，最好也能控制 Offset 属性，怎样才能快速满足他们的需求（[:uvmove]）？

第 5 章
后处理效果

本章主要讨论常见的后处理效果，我们会从需求出发，逐步实现模糊、泛光、辉光、景深效果。

内容编排上采用由浅入深的方式，建议按顺序阅读。

5.1 模糊效果

对于非全屏 UI，策划希望将背面的场景模糊掉，以突出 UI 界面的显示，请问该如何实现模糊效果？

问题分析

通常来说，覆盖全场景的效果，都会在摄像机上想办法实现。改变画面风格，通常使用后处理技术来实现。

后处理技术

后处理（Post-Process Effect）技术是一种对渲染之后的画面进行再加工的技术。具体来说，针对每一个摄像机，在绘制到用户窗口之前，我们都有机会对这个画面进行二次加工，再将装饰过的画面呈现给用户。因此，后处理能够方便地制作全局效果，但毫无疑问，这样的操作会带来性能消耗。

我们可以将摄像机照射出的内容渲染到一张图中，它被称为 RT（RenderTexture，渲染图）。而将摄像机内容绘制到渲染图的过程被称为 RTT（Render to Texture）。在 Unity 3D 中，RT 对

应的数据结构就是 RenderTexture，我们既可以动态创建内存中的 RT，也可以在工程目录下创建一个 RT 资源。

例如，我们打开 Unity 3D，创建一个新场景。在 Project 标签下空白处右击->Create->RenderTexture，更改名字为 TestRT。在 Inspector 面板中可以看到参数可预览效果。由于我们未写入任何内容，所以看到的预览为纯黑色，如图 5.1 所示。

图 5.1

在摄像机上我们将渲染目标（Render Target）赋值为刚刚创建的 TestRT，如图 5.2 所示。

如果你的场景中只有一台摄像机，则赋值过后就会发现 Game 视窗中的预览效果没有了。这是因为主摄像机的渲染目标已经是 RT，不会将内容输出到画面上。这时点击运行游戏，再选中 TestRT 就可以看到图片的内容已经被修改了，如图 5.3 所示。

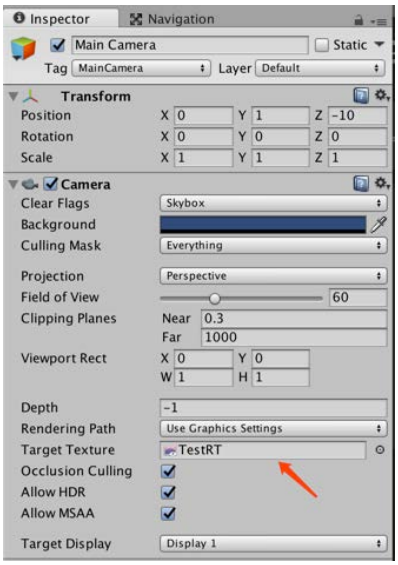


图 5.2

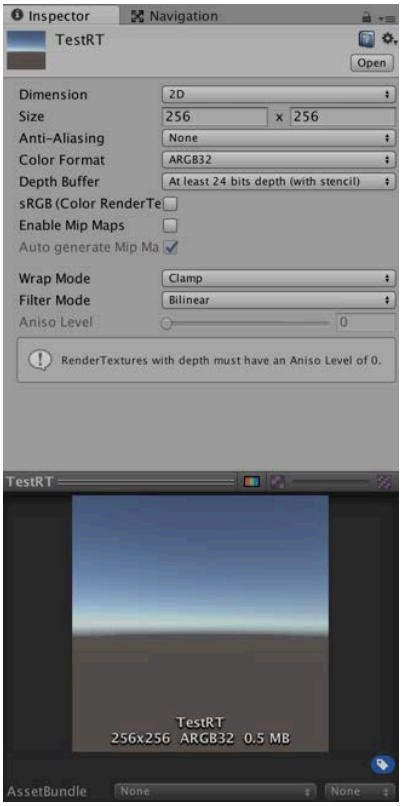


图 5.3

渲染接口

在 Unity 3D 中，编写后处理有专门的接口。在任何一个继承自 MonoBehaviour 的子类中，我们都可以重写下面的函数来改变最终的渲染效果：

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
```

每次程序在渲染之前，都会调用这个函数中，其中 sourceTexture 就是上面讲到的 RT。如果我们将 sourceTexture 的内容通过下面的接口函数直接渲染给 destTexture，则画面就是默认的渲染方式：

```
Graphics.Blit(sourceTexture,destTexture);
```

相对的，如果使用：

```
Graphics.Blit(sourceTexture, destTexture,renderMaterial);
```

则会将 sourceTexture 作为输入图片，经过 renderMaterial 渲染之后，输出到 destTexture 中。因此，为了实现模糊效果，只需在 Shader 中实现渲染算法即可。

所以我们创建一个名为 RenderPostEffect.cs 的类，并编写如下代码：

```
using UnityEngine;
using System.Collections;

public class RenderPostEffect : MonoBehaviour
{
    #region Public Attributes
    public Material renderMaterial;
    #endregion

    #region Unity Messages

    void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
    {
        if(renderMaterial != null)
        {
            Graphics.Blit(sourceTexture, destTexture,renderMaterial);
        }
        else
        {
            Graphics.Blit(sourceTexture,destTexture);
        }
    }
    #endregion
}
```

编辑完成后，将脚本挂载到摄像机上。

模糊原理

在了解了 RT 之后，我们接着探讨最初的问题：实现模糊效果。

多年近视的经历告诉我们，当颜色的边缘不清楚时，就会呈现出模糊效果。因此模糊算法的核心在于，如何处理当前颜色受周围颜色的影响。

最容易想到的是均匀采集周围的像素值与原值混合，这样自身色值就不那么突出了，从而达到模糊效果。这种模糊的策略叫作均值模糊，示意图如图 5.4 所示。

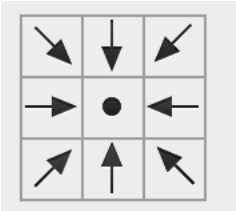


图 5.4

代码实现也非常简单，用两层 for 循环分别取得一个方形区域的点，算出这些颜色的平均值，这个计算过程叫作卷积。我们创建一个名为 Blur 的 Shader，并编写如下内容代码：

```
Shader "Custom/Blur"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _BlurRadius ("_BlurRadius",Range(1,10) ) = 5
        _TextureSizeX ("_TextureSizeX",Float) = 256
        _TextureSizeY ("_TextureSizeY",Float) = 256
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
```

```
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            UNITY_FOG_COORDS(1)
            float4 vertex : SV_POSITION;
        };

        sampler2D _MainTex;
        float4 _MainTex_ST;
        int _BlurRadius;
        float _TextureSizeX;
        float _TextureSizeY;

         fixed4 BlurTexture( float2 uv,float blurRadius,float
                             textureSizeX,float textureSizeY)
        {
            float pixelDisX = 1.0/textureSizeX;  //像素间距X
            float pixelDisY = 1.0/textureSizeY;  //像素间距Y
            int count = blurRadius * 2 +1; //每行的像素数量
            count *= count;

            float4 tmpColor = float4(0,0,0,0);
            for( int x = -blurRadius ; x <= blurRadius ; x++ )
            {
                for( int y = -blurRadius ; y <= blurRadius ; y++ )
                {
                    float4 color = tex2D(_MainTex,uv + float2(x * pixelDisX,y
                                         * pixelDisY));
                    tmpColor += color;
                }
            }
            return tmpColor/count;
        }

        v2f vert (appdata v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
```

```
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the texture
                fixed4 col = BlurTexture(i.uv,_BlurRadius,_TextureSizeX,_
                                         TextureSizeY);
                // apply fog
                UNITY_APPLY_FOG(i.fogCoord, col);
                return col;
            }

            ENDCG
        }
    }
}
```

编写好后，创建一个名为 Blur.mat 的材质。由于输出窗口的分辨率为 1136×640，因此在材质上也调整为对应的长宽，如图 5.5 所示。

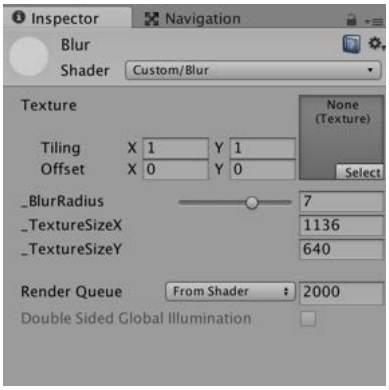


图 5.5

将材质拖到摄像机 RenderPostEffect 组件的 RenderMaterial 上，运行程序，效果如图 5.6 所示。

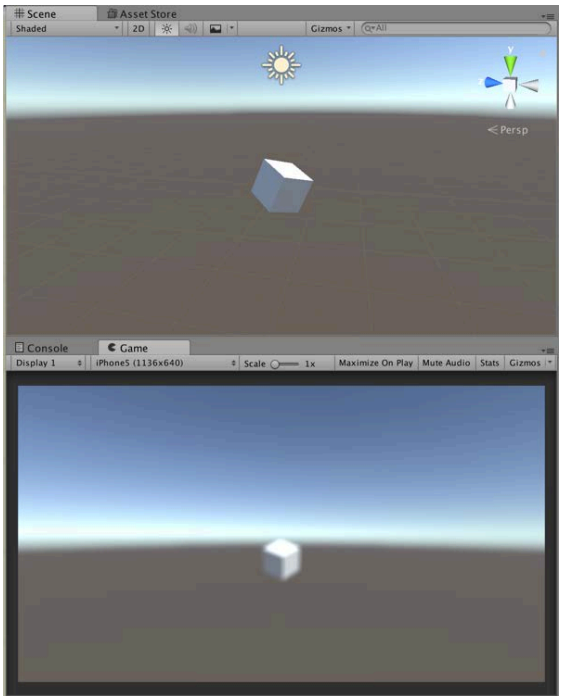


图 5.6

可以明显地看到输出图像中的模糊效果。

总结

本节从最基础的 RT 讲起，介绍了如何在 Unity 3D 中实现后渲染，并利用 Shader 实现了均值模糊。

很多情况下，模糊效果需要调整算法。在上面的算法中，我们平均地取到一个区域内的颜色，但其实这是不合理的。因为距离目标点越远的点，影响应该越小。因此在计算卷积的过程中，我们会引入权重，这些权重也被称为模板（Mask）或卷积核，不同的模板对应了不同的模糊策略。例如，应用了正态分布的加权卷积，就被称为高斯模糊，这种模糊会使模糊后的图像保留更多的色彩细节。相关实现也不复杂，只需在卷积计算时乘上权值即可。通过更改卷积核，这套渲染流程也被广泛用于边缘检测、径向模糊、浮雕、锐化等屏幕效果。

扩展问题

（1）请实现高斯模糊（[:gaussian]）。

（2）请实现边缘检测（[:edgecheck]）。

5.2　泛光效果

现在想制作一个全屏泛光的效果，请简述如何实现。

问题分析

全屏泛光（Bloom）是一种在实际项目中常用的技术，用于模拟强光下的效果。它的技术实现并不复杂，简单来说，全屏泛光只是在模糊后的渲染结果基础上，再叠加原场景效果。在上一节中，我们介绍了模糊的实现方式，现在就在之前的基础上看看如何实现全屏泛光。

实现 Shader

虽说是全屏泛光，但在实现时还是要限制泛光区域，否则就会过亮。通常的做法是通过阈值颜色来控制。当颜色大于设置的阈值时，才被认为是有效颜色，而只有有效颜色才可以计入模糊的采样色中。

通过对上节的学习，我们已经掌握了如何获取一个模糊后的色值。在这个基础上，我们只需加入混色的逻辑，将原图与模糊图按照一个参数叠加即可。

说过了原理，我们再来创建一个名为 Bloom 的 Shader，并编写如下内容：

```
Shader "Custom/Bloom"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _BlurRadius ("_BlurRadius",Range(1,20) ) = 5
        _BloomFactor ("_BloomFactor",Range(0,1))=0.5
        _ColorThreshold ("_ColorThreshold",Color) = (0.5,0.5,0.5,1)
        _TextureSizeX ("_TextureSizeX",Float) = 256
        _TextureSizeY ("_TextureSizeY",Float) = 256
    }
```

```
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;
            int _BlurRadius;
            float _TextureSizeX;
            float _TextureSizeY;
            float _BloomFactor;
            fixed4 _ColorThreshold;

            fixed4 BlurTexture( float2 uv,float blurRadius,float
                                textureSizeX,float textureSizeY)
            {
                float pixelDisX = 1.0/textureSizeX;  //像素间距
                float pixelDisY = 1.0/textureSizeY;  //像素间距
                int count = blurRadius * 2 +1; //每行的像素数量
```

```
                count *= count;

                float4 tmpColor = float4(0,0,0,0);
                for( int x = -blurRadius ; x <= blurRadius ; x++ )
                {
                    for( int y = -blurRadius ; y <= blurRadius ; y++ )
                    {
                        float4 color = tex2D(_MainTex,uv + float2(x * pixelDisX,y
                                                                  * pixelDisY));
                        color = saturate(color - _ColorThreshold);
                        tmpColor += color;
                    }
                }
                return tmpColor/count;
            }

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the texture
                fixed4 orgColor = tex2D(_MainTex,i.uv);
                fixed4 blurColor =
BlurTexture(i.uv,_BlurRadius,_TextureSizeX,_TextureSizeY);
                fixed4 final = orgColor + blurColor * _BloomFactor;
                // apply fog
                UNITY_APPLY_FOG(i.fogCoord, final);
                return final;
            }

            ENDCG
        }
    }
}
```

不难发现，大多数内容与模糊 Shader 相同，在计算阈值时，我们使用了 saturate 函数：

```
color = saturate(color - _ColorThreshold);
```

这个函数会返回参数的值，并将其限制在[0,1]区间，这样就得到了裁剪后的合法的颜色。

最后，我们通过_BloomFactor 这个参数，来调整模糊图与原图的混色程度，得到最终的效果：

```
fixed4 final = orgColor + blurColor * _BloomFactor;
```

测试效果

编写完 Shader 之后，我们创建一个名为 Bloom 的材质，调整参数如图 5.7 所示。

图 5.7

在主摄像机上，挂载前面编写的 RenderPostEffect 脚本，将 Bloom 材质拖到 RenderMaterial 处，运行程序，即可看到泛光效果，如图 5.8 所示。

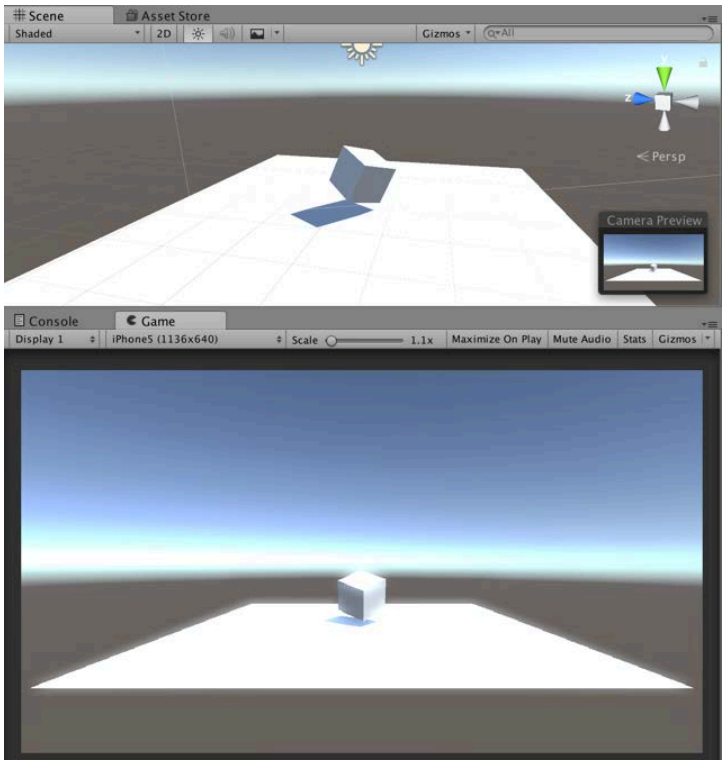


图 5.8

总结

本节介绍了如何通过 Shader 与后处理实现全屏泛光效果。在实际使用过程中，全屏泛光效果会可通过定制化来调整和优化，这就需要我们了解每一部分效果都是如何制作出来的。比如，泛光的范围不够大、泛光的强度不够亮，等等。又或者为了支持某些低端机型，可以舍弃一些效果，该如何优化？这都是很实际的问题，只有掌握原理才能给出最有效的方案。

5.3　辉光效果

在看过全屏泛光效果之后，组内讨论一致认为，赛博朋克那种风格会更好一些。现在想做场景有些辉光的物体，但由于性能和效果等因素，不可以使用额外光源。请实现这种效果。

问题分析

简单来说，辉光效果（Glow）是全屏泛光效果的升级版本。它们的不同之处在于，辉光效果要求场景中只有部分物体泛光。因此，我们需要在制作全屏泛光之前，区分出忽略哪些物体。在 Unity 3D 中，可以用下面的函数，动态替换摄像机下物体的 Shader：

```
GetComponent<Camera>().RenderWithShader(m_renderShader, "RenderType");
```

有了这个函数，我们就可以利用 RenderTexture 来制作辉光效果了。大体思路是，用一个额外的摄像机，在这个摄像机下，所有的非泛光物体都是黑色的，而泛光物体则保持原样。将这个结果渲染到一个 RT 上，然后对它进行模糊。最后将模糊后的泛光图叠加到主摄像机的渲染图像上，即可达成效果。

厘清了主体思路，下面我们一步一步实现这个过程。

目标物体 Shader

首先，我们制作目标物体的 Shader。总体来说，发光物体与原物体没有太大区别。我们只需更改它的 `RenderType` 即可，因此我们创建一个名为 LightGeometry 的 StanderShader，并编写如下内容：

```
Shader "Custom/LightGeometry" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Glow" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all
        // light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0
```

```
        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        // #pragma instancing_options assumeuniformscaling
        UNITY_INSTANCING_CBUFFER_START(Props)
            // put more per-instance properties here
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

不难发现，我们只更改了 RenderType：

```
Tags { "RenderType"="Glow" }
```

接着创建一个使用这个 Shader 的材质，名为 Light，并将其指定给场景中的物体。

替换 Shader

在之前的 RenderPostEffect 脚本中，我们能完成按传入材质渲染画面，现在我们为其加入替换 shader 的扩展功能。更改 RenderPostEffect 脚本如下：

```
using UnityEngine;
```

```
using System.Collections;

public class RenderPostEffect : MonoBehaviour
{
    #region Public Attributes
    public Material renderMaterial;
    public Shader replaceShader;
    #endregion

    #region Unity Messages
    void LateUpdate()
    {
        if(GetComponent<Camera>() != null && replaceShader != null)
        {
            GetComponent<Camera>().RenderWithShader(replaceShader,"RenderTyp
e");
        }
    }

    void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
    {
        if(renderMaterial != null)
        {
            Graphics.Blit(sourceTexture, destTexture,renderMaterial);
        }
        else
        {
            Graphics.Blit(sourceTexture,destTexture);
        }
    }
    #endregion
}
```

当调用 RenderWithShader 函数时，会检查摄像机下的每个物体的 RenderTag。通常选取的 Tag 条件都是 RenderType。替换过程我们可以想象成：遍历所有要显示的物体，如果物体的 Shader 与传入的 replaceShader 有相同的 RenderType，则用 replaceShader 的相应 subshader 替换。如果找不到，就不绘制。通常来说，我们依然要绘制不发光的物体为黑色，以免遮挡关系会出错。另外，在使用 RenderWithShader 实现特效时，应该将这个相机设为 disable，以防止影响原图的绘制。

我们创建一个用来渲染 RT 的 Shader，命名为 RenderGlowRT，编写内容如下：

```
Shader "Custom/RenderGlowRT" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Glow" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all
        // light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        // #pragma instancing_options assumeuniformscaling
        UNITY_INSTANCING_CBUFFER_START(Props)
            // put more per-instance properties here
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
```

```
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
```

```
            {
                // sample the texture
                fixed4 col = fixed4(0,0,0,1);
                return col;
            }

            ENDCG
        }
    }
    FallBack "Diffuse"
}
```

不难看出，在这个 Shader 中增加了一个 subshader，用来将 RenderType 为 Opaque 的物体替换为黑色，而 RenderType 为 Glow 的内容则保持不变。

控制脚本

在主摄像机上，我们要动态创建一个子摄像机，并将渲染结果保存到一张 RT 上，最后传递给一个混色 Shader。创建一个名为 CustomGlow 的脚本，编写如下代码：

```
using UnityEngine;
using System.Collections;

public class CustomGlow : MonoBehaviour
{
    #region Public Attributes
    public Shader m_renderGlowShader;
    public Material m_blurMaterial;
    public Material m_mixMaterial;
    #endregion

    #region Private Attributes
    private RenderTexture m_rt;
    #endregion

    #region Unity Messages

    void Start()
    {
        var mainCam = GetComponent<Camera>();
```

```
        m_rt = new RenderTexture(Screen.width, Screen.height,(int)mainCam.depth);

        GameObject obj = new GameObject("CameraRT");
        obj.transform.SetParent(transform,false);

        var cam = obj.AddComponent<Camera>();
        cam.enabled = false;
        cam.clearFlags = CameraClearFlags.SolidColor;
        cam.backgroundColor = Color.black;
        cam.orthographic = mainCam.orthographic;
        cam.orthographicSize = mainCam.orthographicSize;
        cam.nearClipPlane = mainCam.nearClipPlane;
        cam.farClipPlane = mainCam.farClipPlane;
        cam.fieldOfView = mainCam.fieldOfView;
        cam.targetTexture = m_rt;

        var bloomTex = obj.AddComponent<RenderPostEffect>();
        bloomTex.replaceShader = m_renderGlowShader;
        bloomTex.renderMaterial = m_blurMaterial;


        m_mixMaterial.SetTexture("_BlurTex", m_rt);

    }

    void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
    {
        Graphics.Blit(sourceTexture, destTexture, m_mixMaterial);
    }

    #endregion
}
```

从代码中可以看出，我们在主摄像机下创建了参数相同的子摄像机，并将这个摄像机的渲染目标设置为内存中的一个 RenderTexture。在渲染时，通过 OnRenderImage 将其与主摄像机的内容进行混合。编写完成后，将这个脚本挂在主摄像机上。

将刚刚创建的 RenderGlowRT 赋值给脚本上的 RenderGlowShader 属性。再创建一个名为 BlurGlow 的材质，使用 Blur 着色器，将材质赋值给 BlurMaterial 属性，并设置参数如图 5.9 所示。

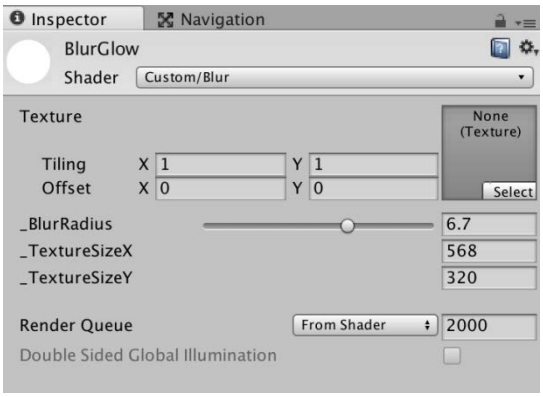


图 5.9

最后再新建名为 MixTexture 的 Shader，编写如下内容：

```
Shader "Custom/MixTexture"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _BlurTex ("Blur Texture", 2D) = "white" {}
        _BlurFactor("Blur Factor",Range(0,1)) = 0.5
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
```

```
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            UNITY_FOG_COORDS(1)
            float4 vertex : SV_POSITION;
        };

        sampler2D _MainTex;
        float4 _MainTex_ST;

        sampler2D _BlurTex;
        float4 _BlurTex_ST;

        float _BlurFactor;

        v2f vert (appdata v)
        {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _MainTex);
            UNITY_TRANSFER_FOG(o,o.vertex);
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
            // sample the texture
            fixed4 col = tex2D(_MainTex, i.uv);
            fixed4 blur = tex2D(_BlurTex,i.uv);
            fixed4 final = col + blur * _BlurFactor;
            // apply fog
            UNITY_APPLY_FOG(i.fogCoord, final);
            return final;
        }
        ENDCG
    }
  }
}
```

创建一个使用这个 Shader 的材质，并命名为 MixTexture，然后将它赋值给 Custom Glow 脚本的 Mix Material 属性。脚本挂载完成后应如图 5.10 所示。

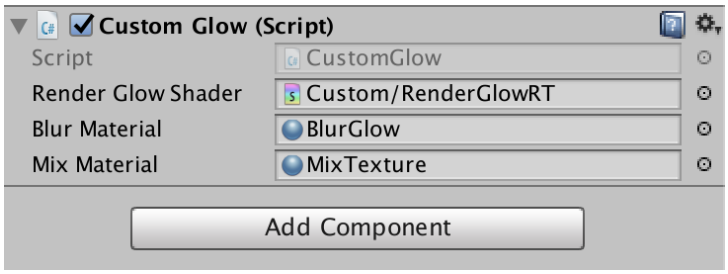


图 5.10

编辑完成后，运行程序效果如图 5.11 所示。

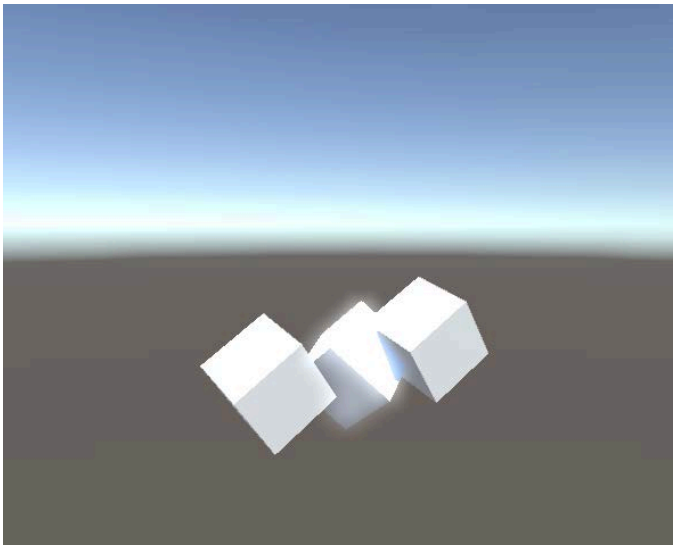


图 5.11

总结

本节我们延续 Bloom 的知识，利用 Unity 3D 的替换 Shader 接口，自定义了 RenderTexture 的内容，最终通过混色 Shader 达到想要的效果。通过这样的流程，还可以制作更多风格化的效果，如自定义透明效果或者更改遮挡关系等，希望大家能掌握这项技术。

扩展问题

（1）在了解了辉光的实现原理之后，请谈谈辉光与多光源的优劣对比（[:glowvslight]）。

（2）由于使用辉光效果后内存占用过多，请问有无优化的方法（[:glowmem]）？

（3）请实现场景中的人物描边（[:edgepost]）。

5.4 景深

随着游戏的开发，场景中的物件越来越多，会导致玩家注意力分散。为了改变这种情况，现希望制作景深效果，使画面更有层次感。具体来说，当物体距离较远时可以自动模糊，可以参考单反摄像的失焦效果。

问题分析

在准备做一个未知效果时，我们需要将其拆分成已知的效果，再找到技术难点并给出解决方案。对于景深效果（Depth Of Field，简称 DOF），我们可以将一张图拆分为模糊区域与清晰区域两部分。那么哪部分可以模糊呢？显然这里需要模糊的是远处的物体，因此我们要解决的问题就演化为如何区分场景中物体与摄像机的距离。

在 Unity 3D 中，可以通过

```
GetComponent<Camera>().depthTextureMode |= DepthTextureMode.Depth;
```

来开启摄像机的深度检测模式。开启后，就可以在 Shader 中通过：

```
float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
depth = Linear01Depth(depth);
```

来取到深度值。当获取深度后，就可以根据景深与焦距来判断需要模糊哪些像素。根据焦距，将原图与远景图进行混色，即可完成景深的效果。

深度图

深度图其实与前面辉光效果中用到的 RT 很类似，只不过它特指用来记录图像深度的灰度图。在 Unity 3D 中我们可以直接开启摄像机的相关模式来获取深度。因为会产生性能消耗，所

以默认情况下它是关闭的，这些消耗在延迟渲染（Deferred Rendering）中可以忽略，因为这个模式下的几何缓冲（G-Buffer）也需要深度。这里我们依然采用扩展 RenderPostEffect 脚本的形式，为脚本添加深度图的支持：

```
using UnityEngine;
using System.Collections;

public class RenderPostEffect : MonoBehaviour
{
    #region Public Attributes
    public bool isDepthEnable = false;
    public Material renderMaterial;
    public Shader replaceShader;
    #endregion

    #region Unity Messages
    void Start()
    {
        if(GetComponent<Camera>() != null )
        {
            if(isDepthEnable)
            {
                GetComponent<Camera>().depthTextureMode |=
DepthTextureMode.Depth;
            }
        }
    }

    void LateUpdate()
    {
        if(GetComponent<Camera>() != null && replaceShader != null)
        {
            GetComponent<Camera>().RenderWithShader(replaceShader,"RenderType");
        }
    }

    void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
    {
        if(renderMaterial != null)
        {
            Graphics.Blit(sourceTexture, destTexture,renderMaterial);
```

```
        }
        else
        {
            Graphics.Blit(sourceTexture,destTexture);
        }
    }
    #endregion
}
```

在 Start 函数中，我们加入根据标志位开启获取深度图的逻辑。

首先新建一个场景，并在场景中创建几个立方体。

其次在主摄像机上挂载这个脚本，并在脚本上勾选 IsDepthEnable。

接着再创建一个绘制深度图的 Shader，并命名为 DepthTex，编写内容如下：

```
Shader "Custom/DepthTex"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
```

```
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _CameraDepthTexture;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
                depth = Linear01Depth(depth);
                return float4(depth, depth, depth, 1);
            }
            ENDCG
        }
    }
}
```

我们通过内置变量_CameraDepthTexture 可以获取引擎内部的深度图，并将其绘制出来。最后我们创建一个使用此 Shader 的材质，并将其拖到对应属性，如图 5.12 所示。

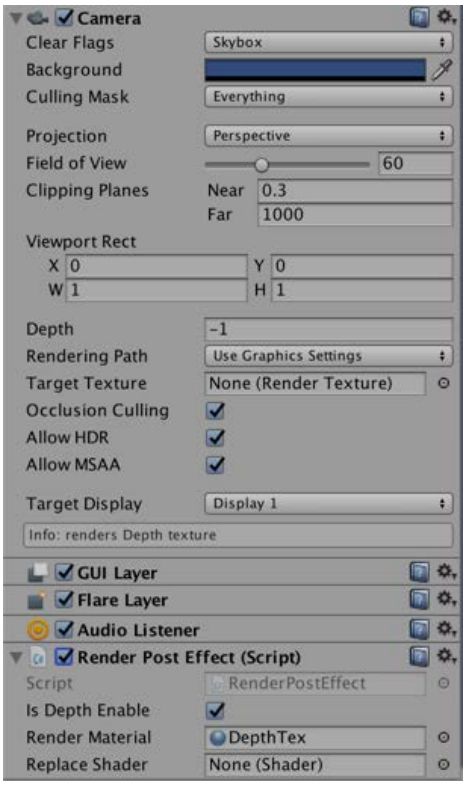


图 5.12

运行游戏后我们发现，越远的地方越白，越近越黑，但立方体为纯黑色，并不能分辨出自身的深度。出现这个错误的原因是，摄像机的视锥体太高，导致深度变化无法反映在深度图中。更改摄像机上的 `Clipping Planes` 值，直到合适范围，笔者这里的取值为 0.3~15。

运行效果如图 5.13 所示。

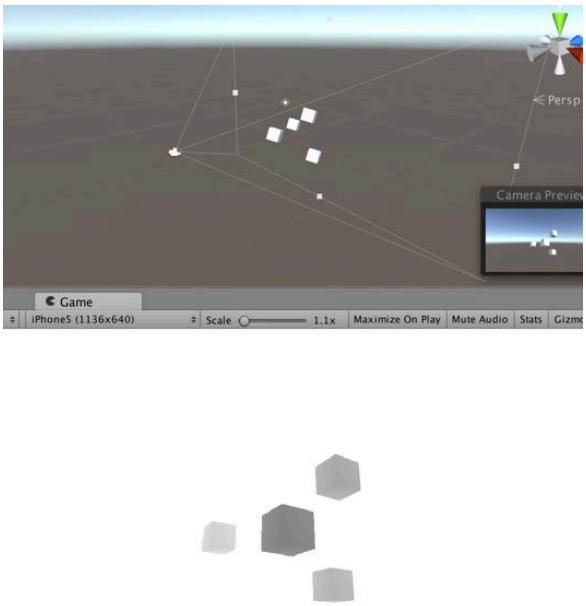


图 5.13

制作模糊

在能够获取深度之后，我们就可以着手通过深度判断来绘制模糊图。首先创建一个名为 DOF 的 Shader，用来混合模糊图与原图，编写如下内容：

```
Shader "Custom/DOF" {

    Properties{
        _MainTex("Base (RGB)", 2D) = "white" {}
        _BlurTex("Blur", 2D) = "white"{}
    }

    SubShader
    {
        Pass
        {

            Cull Off
            ZTest Off
```

```
            ZWrite Off
            Fog{ Mode Off }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float2 uv  : TEXCOORD0;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BlurTex;
            sampler2D_float _CameraDepthTexture;
            float _focalDistance;

            v2f vert(appdata v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                fixed4 ori = tex2D(_MainTex, i.uv);
                fixed4 blur = tex2D(_BlurTex, i.uv);

                float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
                depth = Linear01Depth(depth);
```

```
                fixed4 final = (depth <= _focalDistance) ? ori : blur;

                return final;
            }
            ENDCG
        }
    }
}
```

在这个 Shader 中，我们通过比较图像深度与传入的焦距_focalDistance，决定采用模糊图还是原图。创建一个使用此 Shader 的材质，重命名为 DOF。

接着新建一个名为 CustDOF 的脚本，编写如下内容：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Camera))]
public class CustDOF : MonoBehaviour
{
    #region Public Attributes
    public Material _BlurMaterial;
    public Material _DOFMaterial;

    [Range(0,1)]
    public float focalDistance = 0.5f;

    public int downSample = 1;

    #endregion

    #region Unity Messages

    void OnEnable()
    {
        GetComponent<Camera>().depthTextureMode |= DepthTextureMode.Depth;
    }

    void OnDisable()
    {
        GetComponent<Camera>().depthTextureMode &= ~DepthTextureMode.Depth;
```

```
    }

    void OnRenderImage(RenderTexture source, RenderTexture destination)
    {

        RenderTexture blurTex = RenderTexture.GetTemporary(source.width >>
downSample, source.height >> downSample, 0, source.format);

        Graphics.Blit(source, blurTex, _BlurMaterial);

        _DOFMaterial.SetTexture("_BlurTex",blurTex);
        _DOFMaterial.SetFloat("_focalDistance", focalDistance);

        Graphics.Blit(source, destination,_DOFMaterial);

        RenderTexture.ReleaseTemporary(blurTex);
    }
    #endregion

}
```

在这个脚本中，我们开启了摄像机的深度图。在 OnRenderImage 中，我们根据屏幕分辨率与采样率动态创建一张 RT。将原图模糊后的效果渲染到这张 RT 上。接着将这张图与焦距作为参数传入景深的 Shader 中，最后将原图通过景深 Shader 渲染出来。在函数结束之前记得释放 RenderTexture。

将脚本挂在摄像机上（记得将刚才挂在摄像机上的 RenderPostEffect 删掉），再把 DOF 材质拖到 `DOF Material` 上。接着创建一个使用 `Custom/Blur` 的材质，调整成合适的参数，拖到 Blur Material 上。最终效果如图 5.14 所示。

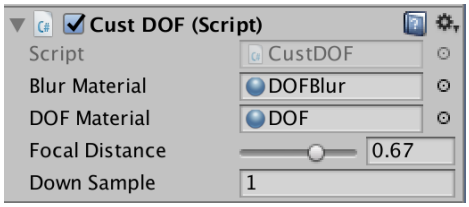


图 5.14

运行程序，可以看到如图 5.15 所示的景深效果。

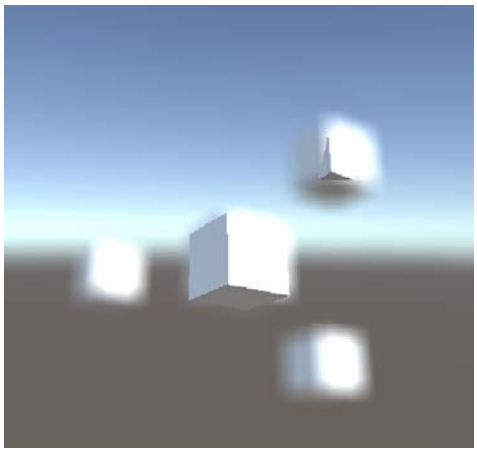


图 5.15

平滑过渡

仔细观察效果，就会发现在模糊与原图过渡的区域存在比较明显的切边，如图 5.16 所示。这显然不可接受，需要加入平滑过渡。

图 5.16

过渡通常的做法是使用 lerp 函数进行插值。这里我们加入一个可调整的系数来解决此问题。

更改 CustDOF 脚本如下：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Camera))]
public class CustDOF : MonoBehaviour
{
    #region Public Attributes
    public Material _BlurMaterial;
    public Material _DOFMaterial;

    [Range(0,1)]
    public float focalDistance = 0.5f;
    [Range(0.0f, 100.0f)]
    public float scaleParam = 30.0f;

    public int downSample = 1;

    #endregion

    #region Unity Messages

    void OnEnable()
    {
        GetComponent<Camera>().depthTextureMode |= DepthTextureMode.Depth;
    }

    void OnDisable()
    {
        GetComponent<Camera>().depthTextureMode &= ~DepthTextureMode.Depth;
    }

    void OnRenderImage(RenderTexture source, RenderTexture destination)
    {

        RenderTexture blurTex = RenderTexture.GetTemporary(source.width >>
               downSample, source.height >> downSample, 0, source.format);

        Graphics.Blit(source, blurTex, _BlurMaterial);

        _DOFMaterial.SetTexture("_BlurTex",blurTex);
        _DOFMaterial.SetFloat("_focalDistance", focalDistance);
        _DOFMaterial.SetFloat("_scaleParam", scaleParam);
```

```
        Graphics.Blit(source, destination,_DOFMaterial);

        RenderTexture.ReleaseTemporary(blurTex);
    }
    #endregion

}
```

更改 DOF.shader 如下：

```
Shader "Custom/DOF" {

    Properties{
        _MainTex("Base (RGB)", 2D) = "white" {}
        _BlurTex("Blur", 2D) = "white"{}
    }

    SubShader
    {
        Pass
        {

            Cull Off
            ZTest Off
            ZWrite Off
            Fog{ Mode Off }
            ColorMask RGBA

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
```

```
            {
                float4 pos : SV_POSITION;
                float2 uv  : TEXCOORD0;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BlurTex;
            sampler2D_float _CameraDepthTexture;
            float _focalDistance;
            float _scaleParam;

            v2f vert(appdata v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                fixed4 ori = tex2D(_MainTex, i.uv);
                fixed4 blur = tex2D(_BlurTex, i.uv);

                float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
                depth = Linear01Depth(depth);

                fixed4 final = (depth <= _focalDistance) ? ori : lerp(ori, blur,
                        clamp((depth - _focalDistance) * _scaleParam, 0, 1));

                return final;
            }
            ENDCG
        }

    }
}
```

可以看到，我们根据景深，通过缩放系数来平滑插值。调整参数后再次运行，即可看到较为平滑的过渡效果，如图 5.17 所示。

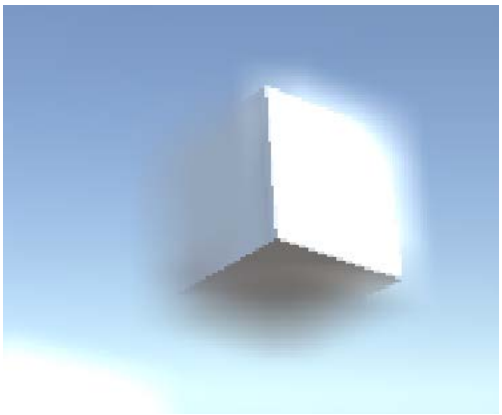


图 5.17

总结

本节讲述了如何通过深度图与 Shader 混色实现景深的效果。深度图是全屏抓取，存在较大的性能消耗，通常会采用降低采样率或定制裁剪的方式进行优化。另一方面，深度图只能绘制 RenderType 在 2500 或以下的材质，因此半透明物体不会写入深度图中，如果需要兼容这些物体，则需要用 RenderWithShader 的方法，重写 RenderType 为 Transparent 的 Shader，定制更改深度图输出效果，最终合成正确的效果。

扩展问题

如何才能同时实现近处失焦与远处失焦（[:neardof]）。

第 6 章
资源工作流

在实际工作过程中，美术资源并不总是按照预想的方式提交的。导入工作本质上处于程序和美术工作链中的灰色区域：美术人员认为自己只负责在 PS、3D Max 或 Maya 中制作出对应的资源，只要评审通过，其他与自己无关；而程序员则认为自己只负责编写读取逻辑，不需要操心资源的验收。按照正规流程，这其中的过渡工作应由项目中的 TA（Tech Artist）来负责，但坦白讲，国内这方面人才比较匮乏，有丰富经验 TA 把控的项目更是凤毛麟角。

随着项目规模的增大，资源管理混乱产生的影响会成倍地增加。有些团队的做法是不给美术提交权限，而是将资源提交给相关负责人或执行策划，检查无误后再提交到项目中。笔者更倾向于在导入时加入自动化处理，虽然需要编写一部分处理代码，但总体来看，这种策略可以省下不少时间。

6.1 图片格式更改

项目中有很多 UI 图片，默认都开启了 MipMap 属性，现在希望将其关闭。一般来说直接选中一批图片更改就好了，但现在 UI 目录的层级比较复杂，分目录查找比较耗费精力。另一方面，图片还会源源不断地提交到各个文件夹中，每有更新都检查资源也不方便，请问有没有好的解决方案。

问题分析

图片导入项目时，默认是 Texture 类型。对于 UI 来说，要想将其应用到 UGUI 中，就需要手动更改其类型为 Sprite。这步操作本质上不需要人工干预，最优的处理方式是自动完成。同

理，UI 图片也不需要开启 MipMap，它们应该在统一的图片处理流水线中完成。

在 Unity 中，AssetPostprocessor 类能够在导入资源时，或导入资源后，捕获到相应的信息。

在纹理图片导入的过程中，钩子函数的调用顺序如下。

◎ OnPreporcessTexture：在进入导入阶段前，会调用这个函数。我们能够重写 TextureImporter 的设置来更改导入资源的通用信息。

◎ OnPostprocessTexture：这是导入之后执行的函数，当函数运行完成时对象也创建完成。这个函数有一个参数，就是这个实例对象自己。由于是引用对象，因此改变属性的操作可以带出函数作用域，保存到最终的预设中。

因此，要想完格式转换的任务，可以在当有图片需要导入项目中时，触发回调函数，在回调函数中设置参数并保存对应属性。

文件导入回调

首先创建一个名为 TextureChange.cs 的文件，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class TextureChange :  AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        Debug.Log(importer.assetPath);
    }
}
```

代码编辑保存后，导入新图片，或在右键快捷菜单中单击 Reimport，即可输出图片的路径。如图 6.1 所示。

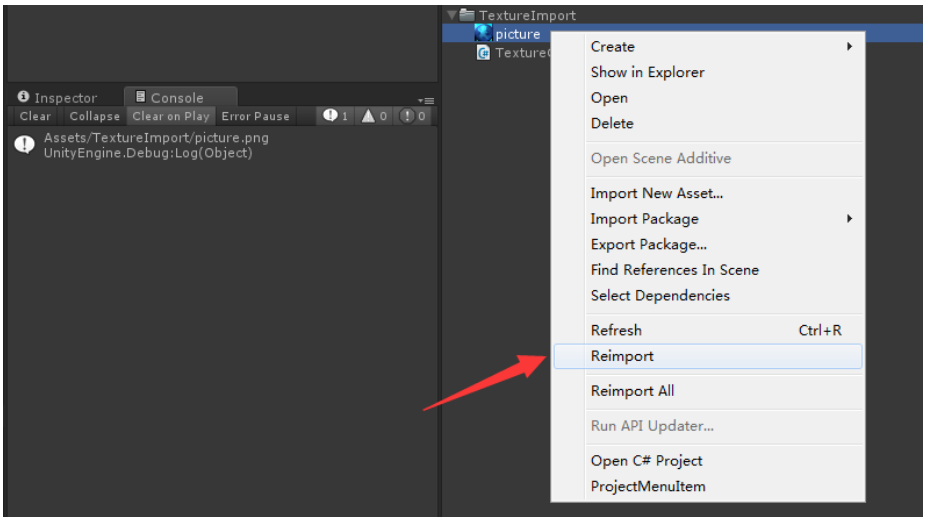


图 6.1

更改属性

接着更改对应的属性，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class TextureChange :  AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        Debug.Log(importer.assetPath);
        //Set mipmap disable
        importer.mipmapEnabled = false;
        importer.textureType = TextureImporterType.Sprite;
        //Save the changes
        EditorUtility.SetDirty(importer);
    }
}
```

在上面的代码中，通过 TextureImporter 更改相关属性，通过 SetDirty 将对象标记为需要保存更改。

总结

通过 AssetPostprocessor 类，可以整合导入的资源，定制流程化的处理。一旦完成了资源预处理的配置，就可以从源头降低项目中资源出错的现象。

扩展问题

（1）什么情况下需要开启 MipMap，什么情况需要关闭（[:mipmap]）？

（2）项目中有些资源还是需要使用默认的 Texture 类型，所以不能完全更改为 Sprite，请问应如何改进（[:spritecfg]）？

6.2　动画抽取

模型美术上传了一批角色的动作文件，经过检查，发现很多动作都有问题。有些动作导入项目时，并未使用快捷键 Ctrl+D 提取其中的 Animation 文件，只是提交了动作的 FBX；有些动作应该是循环的，但美术并未勾选循环，如图 6.2 所示。

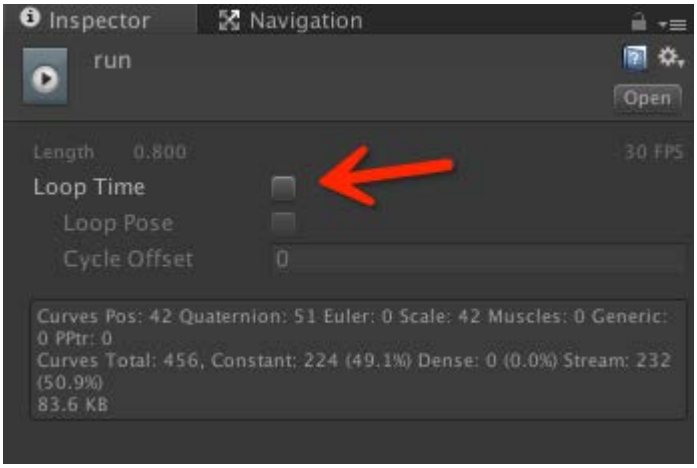


图 6.2

动作文件总数很多，而且目录层级复杂，检查需要花费不少时间。有什么办法能解决并避免这样的情况出现呢？

问题分析

在很多项目中，模型的动画文件都是单独导出 FBX，而这些文件中，有价值的信息都保存在对应的 Animation 内。当使用这种资源组织形式时，动画都需要从 FBX 中提取存放。由于 FBX 文件中既可以包含模型，也可以包含动画，因此常规做法是采用命名规范，或者文件路径做区分标记。

本节以路径为例讲解，这次我们需要借助 `ModelImporter` 类来监听模型资源导入事件。当有模型导入，且在动画文件夹中时，就提取其中的动画文件保存，并将这个 FBX 文件本身删除。

本节的资源使用 Unity 3D 官方吉祥物 Unity-chan 的动作。整套资源可以在 AssetStore 上免费下载（[:unitychan]）。笔者以 unitychan_WIN00 动作为例，配合 unitychan.fbx 讲解如何抽取动画。

文件导入回调

创建一个名为 AnimExtract.cs 的文件，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class AnimExtract :  AssetPostprocessor
{
    void OnPostprocessModel(GameObject obj)
    {
        ModelImporter importer = assetImporter as ModelImporter;
        Debug.Log(importer.assetPath);
    }
}
```

与前面的更改图片格式一样，当导入新资源或 Reimport 时，会输出文件的路径日志。

文件判断

成功收到回调信息后，可以着手做文件类型区分了。整理文件目录为如图 6.3 所示的结构。

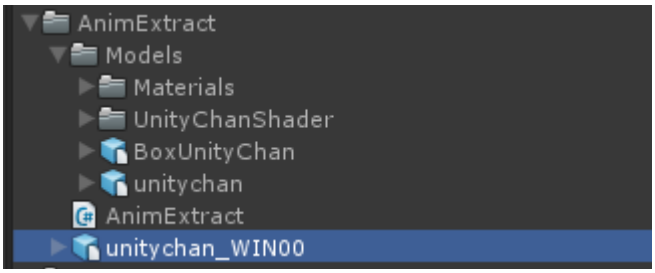


图 6.3

更改函数代码如下：

```
void OnPostprocessModel(GameObject obj)
{
    ModelImporter importer = assetImporter as ModelImporter;

    if (!importer.assetPath.Contains("Models"))
    {
        Debug.Log(importer.assetPath);
    }
}
```

编写完成后，即可完成模型过滤：只有目录中不包含 Models 字符串时才被认为是动画。

提取动画

能确定导入的是动画文件后，就可以将其中的动画抽取保存，然后将导入的文件删除：

```
void OnPostprocessModel(GameObject obj)
{
    ModelImporter importer = assetImporter as ModelImporter;

    if (!importer.assetPath.Contains("Models"))
    {
        var assets = AssetDatabase.LoadAllAssetsAtPath(importer.assetPath);
        foreach (var clipObj in assets)
        {
            if (clipObj is AnimationClip && !clipObj.name.Contains("__preview__"))
            {
                string outputPath = Path.GetDirectoryName(importer.assetPath)
                    + Path.DirectorySeparatorChar + clipObj.name + ".anim";
                //Keep the meta connection
```

```
                var currentAsset = AssetDatabase.LoadAssetAtPath<AnimationClip>
                                   (outputPath);
                if (currentAsset != null)
                {
                    EditorUtility.CopySerialized(clipObj, currentAsset);
                    EditorUtility.SetDirty(currentAsset);
                }
                else
                {
                    var newAnim = new AnimationClip();
                    EditorUtility.CopySerialized(clipObj, newAnim);
                    AssetDatabase.CreateAsset(newAnim, outputPath);
                }
            }
        }
    }
}
```

由于 FBX 有时会保存预览动作的动画，所以这里需要通过判断名称将其过滤掉。使用 GetDirectorName 获取父级路径，我们可以将动画保存在 FBX 的同级目录中。最后借助 CopySerialized 函数，可以在不影响文件关联的情况下更新文件内容。

导入与删除

现在有个小问题，在第一次导入时不会运行我们的提取函数。仔细研究不难发现，在 OnPosprocessModel 函数执行时，资源信息无法从 AssetDataBase 中获取，所以需要延迟一点执行，等文件先存储，然后再分离。此逻辑同样适用于删除源 FBX。这里我们使用代理 delayCall，在下一帧执行资源处理逻辑。更改代码逻辑如下：

```
void OnPostprocessModel(GameObject obj)
{
    if (!assetImporter.assetPath.Contains("Models"))
    {
        EditorApplication.delayCall += RemoveFBX;
    }
}

void RemoveFBX()
{
    EditorApplication.delayCall -= RemoveFBX;
```

```
    //... code in OnPostprocessModel

    AssetDatabase.DeleteAsset(assetImporter.assetPath);
}
```

编写完成后，再导入动画文件就会直接将 FBX 替换成对应的 Animation 了。

另外，当所有文件都导入完成时，会调用 OnPostprocessAllAssets。它的参数之中会有出乎意料的对象参与进来，比如脚本文件，因此不推荐使用这个接口。

总结

本节内容是 FBX 动画的自动脱壳流程，在这个基础上，我们可以做很多定制化的检测操作。比如，动画文件占用内存超过某个限定值时，可以输出提醒，或者不允许导入项目等。最终生成的动画直接拖到 Animator 上会自动生成播放逻辑，效果如图 6.4 所示。

图 6.4

扩展问题

（1）如何在本文的基础上自动设置循环动画（[:isloop]）？

（2）如果资源是通过移动的方式进入指定文件夹，这种情况下会自动化处理吗？如果不能

应该怎么办？

6.3 文件移动检测

代码中有些资源加载路径是硬编码的，例如，加载某个预设需要用到的路径为：

```
private string prefabPath = "Assets/FileMove/Res/TestPrefab.prefab";
```

这个文件如果被移动就会引起加载不到预设的问题。虽然可以在加载逻辑中做保护，但大家更倾向于在编辑阶段解决这个问题。请问是否有办法让这个文件无法被移走？

问题分析

在项目中，有些文件的更改需要同步进行。前面讲到的动画抽取功能，如果一开始导入的目录不正确，那么改变目录后并不会触发对应的导入逻辑。再比如，一些自动设置图集标签的功能，也通常在导入时设置，更改目录就需要自动重新导入。

Unity 中的 AssetModificationProcessor 类能处理类似的问题。通过重写它的 OnWillMoveAsset 函数，可以阻止文件的移动。

禁止移动文件

创建一个名为 FileMove.cs 的文件，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class FileMove : AssetModificationProcessor
{
    private static string prefabPath = "Assets/FileMove/Res/TestPrefab.prefab";

    public static AssetMoveResult OnWillMoveAsset(string oldPath,string newPath)
    {
        AssetMoveResult result = AssetMoveResult.DidNotMove;
        if (oldPath == prefabPath)
        {
            result = AssetMoveResult.FailedMove;
```

```
            EditorUtility.DisplayDialog("Attention", "You shouldn't move this
                                        asset","Ok");
        }
        return result;
    }
}
```

编写完成后，当试图移动这个预设时，就会弹出一个提示对话框，阻止文件的移动，效果如图 6.5 所示。

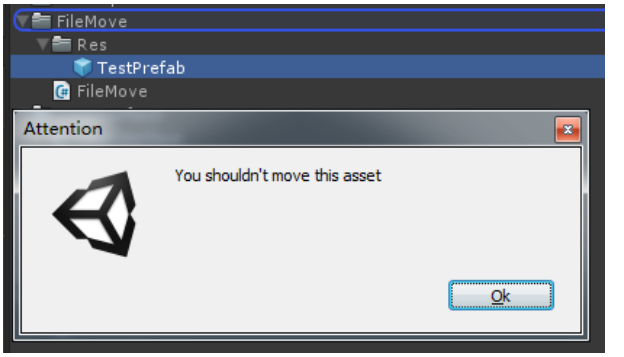


图 6.5

总结

通过定制文件操作的处理方式，我们能做到很多强检测或辅助操作。虽然这些操作不难，但容易忘记，这很容易引起某些设置的错误。因此有必要对移动文件的后续处理进行详细的规划。

扩展问题

如何实现移动到动画文件目录时，对 FBX 文件进行提取动画的操作？

第三部分

底层核心

我们使用 Unity 3D 引擎做游戏开发时，有时会遇到引擎相关的问题，这就需要我们懂得一部分引擎内部的实现原理。

在第 7 章，我们会一起看一下最常见的问题：渲染。从渲染管线和渲染顺序这两个部分来讨论。

为了简化物理处理模型，有时我们会自己搭建更为简单的物理系统，这就需要我们掌握 3D 数学的知识。在第 8 章，我们会探讨数学相关的内容。

第 9 章则从寻路这个实际案例出发，深入探讨相关知识。

第 7 章
渲染原理

游戏图像的绘制也被称为渲染。当讨论渲染问题时，我们需要将注意力集中在渲染管线的某个阶段。例如，光栅化阶段，或者曲面细分阶段等。因此，我们首先从管线入手，看看如何描述一个渲染过程。然后再来看看常见的渲染处理方式是怎样的。

7.1 渲染管线

请简述在绘制到游戏视图的过程中，一个模型文件的数据经过了哪些转换步骤。或者，请简述渲染管线的整个流程。

问题分析

只要是一本谈及图形学的书，必然会有渲染流水线的内容，本书也不能免俗。究其原因，如果没有这个知识基础，一切都无从谈起。由于本书并不是一本图形学的专著，因此不可能将渲染讲得面面俱到，只希望大家能对此有大概的认识，以便更深入的学习。

回到问题本身，通常所说的渲染流水线指的是 GPU 流水线，不包含 CPU 部分，但其实 CPU 部分也很重要。笔者觉得把渲染流水线比作烹饪会更容易理解一些。做菜可不只是炒，还有洗菜、切菜、腌制等准备工作。GPU 部分就是做这些准备工作的。我们将数据从硬盘读取到内存中，转化为对应的数据流，最后加载到缓存里。

这部分准备工作与平台密切相关，不同引擎的实现也有所差异。通常这部分工作有稳定的标准库。例如，在 OpenGL 中，加载模型使用的是 Assimp（Open Asset Import Library），加载

贴图使用的是 SOIL（Simple OpenGL Image Library）等。因此在讨论时，这部分通常不是重点。就好像大家都在讨论米其林大厨烹调火候掌握得好，但很少有人说这个切墩师傅的刀工不错。

CPU 准备好顶点、贴图、法线、灯光方向、颜色等各种杂七杂八的材料之后，就会把它们交给我们的主厨。GPU 拿到这些材料后，就进到渲染的过程。我们的主厨可不一般，材料全下锅，却井井有条，不忙不乱。主厨技术虽高，但锅并不总是称心如意。锅的讲究可就大了，锅的容量为显存大小，锅口大小为显卡带宽，若想炒得一手好菜，不下点血本怕是痴人说梦。厨师手中锃明瓦亮的锅铲，那是图形 API，每每翻飞一下，便是一次 Drawcall。经过精心的设计翻炒之后，多种食材在一双巧手下，融洽地交错在一起，这种高超的技艺被称为 Blend。最终，出锅的食材，经过屏幕后渲染的勾芡，化作一桌色香味俱全的佳肴，完美地呈现在食客的面前。

言归正传，这只是个直观的比喻，便于理解一些概念。若从流水线上讲，上面的说法并没有深入到本质。下面我们就认真研究一下，这位主厨究竟是如何炼成的。

两大阶段

GPU 接收数据之后，就对其做处理，并最终显示到屏幕上。由于屏幕是 2D 画面，而模型是 3D 资源，因此其中必然有一步操作，将 3D 转化为 2D。我们就将这个转化点作为关键点，将 GPU 流水线分为两个阶段，之前的阶段被称为顶点阶段（或几何阶段），处理 3D 模型数据；之后的阶段被称为片元阶段（或光栅化阶段），处理 2D 像素数据。在这两大阶段中，有很多小流程，下面分别简述它们的作用。

顶点着色器（Vertex Shader）

这是可编程管线中的顶点部分。在 Unity 3D 的 Shader 中可以通过下面的代码进行编辑:

```
#pragma vertex vert
struct v2f {
    float4 pos : SV_POSITION;
    fixed4 color : COLOR;
};
v2f vert (appdata_base v)
{
    //vertex operation
}
```

每个顶点都会调用一次这个函数，并转化为这个函数的输出，这些输出被称为图元

（Primitive ）。如果我们希望更改顶点或法线相关的信息，通常会在这里进行。

曲面细分（Tessellation Shader)

这是可编程管线的一个可选部分，但由于是 OpenGL 4.0 新加入的特性，因此很多设备还不支持。它由 Tessellation Control Shader 和 Tessellation Evaluation Shader 组成，最主要的作用是可以优化曲面，例如使曲面更平滑。

几何着色器（Gemoetry Shader）

这也是可编程管线的一个可选部分。几何着色器的输入输出都是图元（Primitive），不是顶点。这个 Shader 可以用来将简单的图元扩展成更为复杂的形式。例如，在一个 GL_POINT 的基础上，可以扩充成一个六芒星阵。

变换回执（Transform Feedback)

它是在 OpenGL 3.0 之后加入的特性。我们可以将图元存放到 Transform Feedback Buffer 中，并可以决定是否按照先前的流程光栅化。它最主要的特性在于，当下一帧渲染时，我们可以读取到上一帧图元的数据，因此对于定量变化的情况，可以节省掉流水线中之前的步骤。利用这样的特性，很多出众的效果可以不经过 CPU 实现。

这种特性常被用来制作粒子系统。实际实现过程中，模拟计算的逻辑会被移到 GPU 中，这可以极大地降低 CPU 与 GPU 通信带来的消耗。

另一个常见的应用是角色的头发。由于头发经常通过样条曲线控制，在 GPU 中更改样条曲线的控制点，就可以改变头发的形状，进行物理模拟。

裁剪（Clipping）

在这个阶段中，图元会根据视域的平截头体（Frustums）做可见性判断。区域外的顶点会被舍弃，与这个顶点连线的三角形的边会与平截头体求交点，这些交点会成为裁剪后三角形的新顶点。

屏幕映射（Screen Transform）

在这个阶段中，图元的坐标从齐次裁剪空间变换到屏幕空间。

图元装配（Primitive Assembly）

在这个阶段，顶点会被转化为基础图元，供后续步骤使用。如果开启了 TES，或者 GS，则此处输入会变成图元，因此有时也会先将图元拆解，然后再做转化。值得注意的是，三角形的朝向剔除（Face Culling）是在这个阶段完成的，在 Unity 中对应的功能为 Cull。例如，在 Shader 代码中可以这样写：

```
Cull Back
```

即为剔除背面。

光栅化（Rasterization）

在这个阶段中三角形会经历两个步骤。首先，三角形会被转化成片段，接着会遍历三角形，获取顶点属性进行插值，在每个像素点上产生一个片元（Framgent）供后续使用。如果开启了多重采样抗锯齿（Multisample Antialiasing，MSAA），此处就会对每个像素多次采样，产生多个片元，最后再进行混合，以达到抗锯齿的效果。

提前深度测试（Early Depth Test）

在这个阶段会检查片元的深度，如果不需要显示则丢弃它。经过这个流程，可以在片元着色器（Fragment Shader）运行之前去掉不合理的值，以减少运算的消耗。这个流程合情合理，但却与后续的透明测试有些冲突。如果这个片元在后续的透明测试中失败，那么这步深度测试也不应将其通过。因此，有些 GPU 会判断是否有此类冲突，如果有就跳过这个流程。但这个检测是有消耗的，例如，在 PowerVR 的 GPU 中会出现 AlphaTest 拖慢性能的情况。因为有利有弊，所以这个流程不是所有的显卡都支持。

片元着色器（Fragment Shader）

从这里往后，是可编程管线中的片元部分，也是我们自定义 Shader 时，主要更改的部分。在这个阶段，我们可以自定义光照计算、读取贴图颜色、设置透明度等。这个阶段的输入值是光栅化得到的片元数据，输出值也为片元数据，但在 Unity 中，我们会将函数返回值写为运算后的颜色。在 Shader 文件中，我们可以做如下编辑：

```
#pragma fragment frag
```

```
fixed4 frag (v2f i) : SV_Target
{
   //do framgent operation
}
```

逐片元操作（Pre-Sample Operation）

这个阶段做了很多事情，简单来说就是先测试后混合。 先说测试。这个阶段共经历了下面的测试：

◎ 裁切测试（Scissor Test）
◎ 透明测试（Alpha Test）
◎ 模板测试（Stencil Test）
◎ 深度测试（Depth Test）

它们的作用都是确定一个像素是否应该显示，但判断逻辑却各不相同。

裁切测试是通过区域来判断的，因此它与基于顶点的 Clipping 阶段不同。

透明测试。意如其名，如果透明度达不到，就不绘制。在 Unity 中对应的方法为 AlphaTest，例如：

```
AlphaTest Greater 0.5
```

需要注意的是，在 Unity 中，如果重写了片元着色器，那么这个关键字是失效的，我们需要在着色器代码中调用 clip 函数实现透明测试，因此更多时候还是要从全局来规划透明的显示规则。

模板测试是通过缓冲区来进行像素比较。与 ColorBuffer 或 DepthBuffer 相似，StencilBuffer 是一个 8 位图像，每个像素保存一个无符号整型。在渲染时，计算出的值会与预先设定的模板值比较，按照预定的规则决定显示与否。这个过程在 Unity 中，对于 Shader 的关键字 Stencil，用法如下：

```
Stencil {
    Ref 2
    Comp equal
    Pass keep
    ZFail decrWrap
}
```

它的判断逻辑是：

```
if((RefValue & readMask) ComparisonFunction (StencilBufferValue & readMask))
{
    pass;
}
else
{
    discard;
}
```

通过按位与的操作比较两侧的结果。左侧 RefValue 为预定义的值，StencilBufferValue 为缓冲区中的值，ComparisonFunction 为比较方法。在上例中，ComparisonFunction 为 equal，即左右相等。

深度测试。通过深度缓冲区的比较，判断是否应该绘制像素。为了做深度测试，需要先制定深度写入规则，然后再制定比较规则。在 Unity 中，对应的功能分别是 ZWrite 与 ZTest。深度写入不透明物体默认打开，透明物体默认关闭，测试方法默认是小于等于。另外还可以通过 Offset 关键字自定义深度。

说完了测试，再来看看混合。混合的主要步骤如下：

◎　混合（Blending）；
◎　写入遮罩（Write Mask）。

混合是指通过测试的颜色与 ColorBuffer 中颜色的叠加方式。混合有多种叠加方式，一般会使用 RGBA 的值作为计算因子，采用插值的方式进行线性运算。在 Unity 的 Shader 中，使用 Blend 关键字可以开启混合模式，例如：

```
Blend SrcAlpha OneMinusSrcAlpha
```

写入遮罩，在这里主要是颜色遮罩（Color Mask）。它的功能是在最终绘制时，对颜色进行过滤。在 Unity 中，可以使用 ColorMask 关键字开启此功能，例如：

```
ColorMask RG
```

经过上面的所有步骤后，最终这个像素会被绘制到颜色缓冲区中，并被显示到屏幕上。

总结

经历了漫长的步骤，终于讲通了如何把一个模型绘制到屏幕上。我们再简单回顾一下绘制流程。在 CPU 中准备数据，传入 GPU，进入顶点着色器，改变曲面，转变图元数量，裁剪，转换到屏幕中，装配图元，光栅化，片元着色器改变颜色，最终逐片元操作进行混合。

扩展问题

上面的总结是最完整的流程，但有些步骤是可以省略的，请给出最简单的流水线（[:samplepipeline]）。

7.2 渲染顺序

请问一个不透明或半透明物体的显示状态应该怎样确定？

问题分析

首先，我们要认识到这不是一个简单的问题。而且就目前来看，并没有完美的解决方案。下面我们从画家算法说起，再谈谈深度缓冲区，以及它的局限性。

画家算法

画家算法是最简单的方式。使用这种策略时，首先将场景中的多边形根据深度进行排序，然后按照顺序进行描绘。就好比“绿树村边合，青山郭外斜”，先画山水，后画村庄。那么村庄自然会遮挡在山水前面。

但事情远没有这么简单。所谓“山外青山楼外楼”，远景近景有时并不那么规则。如果是两者互相穿插，仅仅通过对模型位置的排序基本不可能处理。如果是“白云生处有人家”呢？当半透明与混合也参与进来，事情会变得更加复杂。

深度缓冲区

为了解决这个问题，便有了深度缓冲的方案。我们开辟一块缓冲区（DepthBuffer）记录像素的深度信息。它存储的是像素到摄像机的深度。每一帧开始前，深度缓冲区都会清空。当需要渲染新像素时，像素的深度会预先被计算出来，比较这个像素的深度值与缓冲区中的值，通过算法判断是否要通过该像素，或需要在深度缓冲中写入该像素的深度。

通过深度缓冲，可以解决“山外青山楼外楼”的问题。在场景中没有透明物体的情况下，我们按照缓冲区比较，无须对场景中的物件进行排序，也避免了重复绘制，既能解决重叠的问题，又可以带来性能的提升。

渲染队列

深度缓冲可以完美解决不透明的情况，但当出现“白云生处有人家”这种半透明的情况时，就会显得力不从心。下面使用云中的农舍来举例说明。

云相对于摄像机的距离一定比农舍更近，因此如果云写入了深度值，农舍就不会被绘制。自然也就看不到半透明效果。为了解决这个问题，我们引入渲染队列的概念。我们依然使用深度缓冲，但将不透明与半透明的物体分开。先渲染不透明的物体，对比其深度值，并将其深度写入缓冲。然后再渲染半透明物体，只对比深度，不写入缓冲。由于半透明物体不能写入缓冲区，因此在半透明物体之间，依然使用画家算法进行排序。

在 Unity 中，可以使用预定义的渲染队列，通过它将渲染顺序大致划分出来。队列索引对于一个整数而言，索引号越小越早被渲染。具体定义如表 7-1 所示。

表 7-1

渲染队列	渲染队列索引	渲染队列描述
Background	1000	这个队列最先被渲染，一般用于背景物体，例如 Skyboxes
Geometry	2000	这是默认不透明物体的渲染队列
AlphaTest	2450	需要透明测试几何体时使用该队列
Transparent	3000	任何需要有半透明的物体都应使用此队列。它默认不写入深度缓冲区
Overlay	4000	任何最后被渲染的对象都使用该队列，例如屏幕后特效

使用时，在 Shader 的 Tag 中通过 Queue 进行设置，例如：

```
Shader "my/test"
{
    SubShader
    {
        //Tags { "Queue"="Geometry" }
        Tags { "Queue"="Geometry+1" } //After Geometry
    }
}
```

依然存在的问题

使用了渲染队列后，我们即可应对半透明的情况。不过由于半透明使用了画家算法，当遇到“云青青兮欲雨，水澹澹兮生烟”的效果时，我们依然无法处理。雨滴与云雾如果相互穿插，我们就无法得到正确的排序，进而写入错误的颜色值。

所幸半透明的错误不会很明显，权衡之下，大多数引擎采用上述方式作为渲染逻辑。为了正确地显示半透明，我们可以拆分或优化模型的形状，以降低穿插。也可以调整混合参数，使穿插看起来不那么明显。

如果还是不行，就需要采用预先深度 Pass（Pre Depth Pass）的方法，将其作为特例，写入深度缓冲。这个操作由于影响了缓冲区，因此也就影响了后续的混合流程，所以使用时需要小心。另外，它还增加了 Drawcall，建议酌情使用。

在 Unity 中可以通过下面的 Shader 代码实现：

```
Shader "pre/depth/path" {
    SubShader {
        Tags {"Queue"="Transparent" "IgnoreProjector"="True"
"RenderType"="Transparent"}

        // extra pass that renders to depth buffer only
        Pass {
            ZWrite On
            ColorMask 0
        }

        //Real Pass Below
    }
}
```

总结

如何写入一个像素的颜色有很大的学问，对于半透明，更是需要根据需求随机应变，了解这其中的技术也就变得尤为重要。

第 8 章
3D数学基础

数学很重要。这是很多行业的准则，游戏也不例外。本章主要探讨解析几何在游戏中的常见用法。

掌握了这些知识后，我们就可以更加灵活地定制物理检测、渲染、轨迹等常见需求了。

8.1 点和向量

我们知道通过一个 Vector3 可以表示一个点，例如：

```
transform.position = new Vector3(1,0,0);
```

另一方面，用 Vector3 也可以表示一个方向，例如：

```
transform.position += Vector3.forward;
```

从直观上看，点和向量都是三个数字组成的有序列表，它们之间究竟有什么区别和联系呢？

问题分析

点和向量有什么区别？点和向量虽然用相同的数据结构表示，含义却完全不同，它们之间的关系也是密不可分的。

动与静

在三维空间中，如果希望描述一个确切的位置，只需将这个位置在坐标轴上的投影记下来，

这就是坐标。它是对应方向上点相对于原点的偏移量，因此是静态的概念。

向量更多情况下用来描述过程，是动态变化的数字表示。向量是有大小和方向的有向线段，因此可以表示位移、速度、方向等信息。

运算

向量主要定义物体间位移和相对差异。可以将其想象成一个没有根的箭头，它没有起点，只能描述相对位置。

点可以与向量相加。相加意味着下面的操作：

（1）移动向量，将箭尾与点重合。

（2）点沿着向量的箭头移动到箭头处。

因此，点与向量做和依然是点。当然，点与向量做差时，结果也一样。可以将其看作是与一个反向的向量做和。

另一方面，我们也可以将点的坐标，看作从原点出发指向该点的向量。这时，点和向量从数值上看完全相同，但意义大相径庭。如果很难理解这种转换关系，不妨将其想象成：把原点沿着向量移动到目标处。

由于有这些灵活多变的性质，所以我们在使用的过程中，需要根据自己的需要灵活变换用法。

总结

点和向量是 3D 数学的基础，我们要理解它们在概念上的不同和联系。当我们说点的时候，想象一个静态的位置；当说向量时，想象沿方向滑动的动作。

扩展问题

为什么两个点做差可以得到一个指向被减数点向量（[:vectoropt]）？

8.2 向量的运算

请列举向量间可以进行的运算，并说明它们的实际意义。

问题分析

向量间的运算几乎是图形学的基石，在各个方面都有着广泛的应用。只有清楚透彻地理解它，才能灵活运用。

下面我们将分析向量的运算方法，以及其几何意义。

搭建测试

为了方便说明我们创建了一个绘制向量的组件。每个向量的数据结构由偏移值、方向值、线色三个属性构成。

创建文件 VecUtil.cs，编写代码如下：

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

[System.Serializable]
public class Vec
{
    public Vector3 offset = Vector3.zero;
    public Vector3 vec = Vector3.right;
    public Color lineColor = Color.cyan;
}

[ExecuteInEditMode]
public class VecUtil : MonoBehaviour
{
    public List<Vec> vecList = new List<Vec>();

    void Update()
    {
        for (int i = 0; i < vecList.Count; ++i)
        {
            ForDebug(transform.position+vecList[i].offset,vecList[i].vec,vec
List[i].lineColor);
        }
    }
```

```
    public void ForDebug(Vector3 pos, Vector3 direction,Color lineColor)
    {
        float arrowHeadLength = 0.25f;
        float arrowHeadAngle = 20.0f;
        Debug.DrawRay(pos, direction, lineColor);

        Vector3 right = Quaternion.LookRotation(direction) * Quaternion.Euler(0,
180 + arrowHeadAngle, 0) * new Vector3(0, 0, 1);
        Vector3 left = Quaternion.LookRotation(direction) * Quaternion.Euler(0,
180 - arrowHeadAngle, 0) * new Vector3(0, 0, 1);
        Debug.DrawRay(pos + direction, right * arrowHeadLength, lineColor);
        Debug.DrawRay(pos + direction, left * arrowHeadLength, lineColor);
    }
}
```

在上面的代码中，我们在 Update 里遍历了数组中所有的向量信息，并提取信息绘制对应的线条。例如，创建空的 GameObject，将它命名为 Test，添加脚本，设置信息如图 8.1 所示。

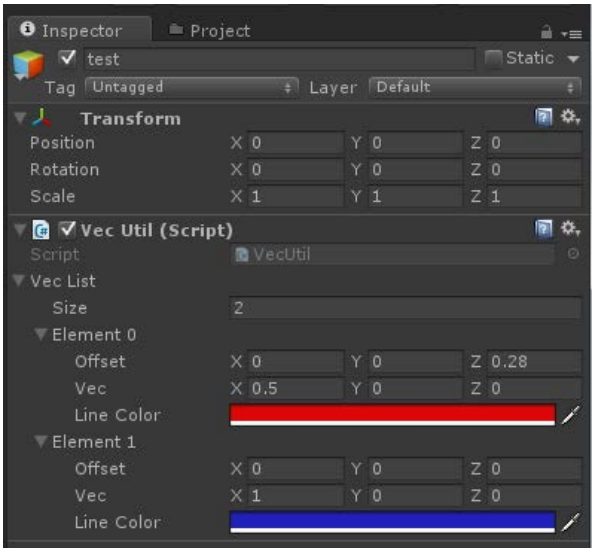


图 8.1

在 SceneView 中的效果如图 8.2 所示。

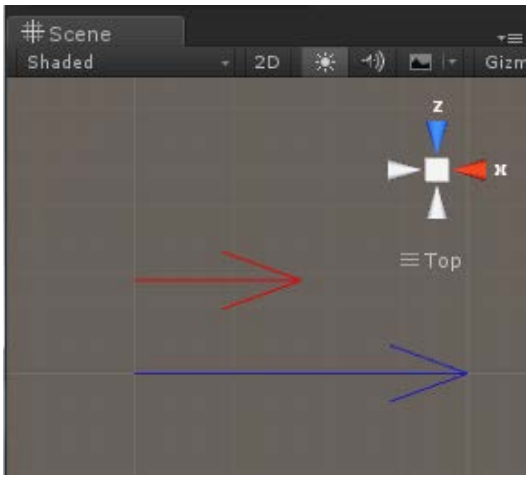


图 8.2

这样就搭建好了环境，能可视化地验证运算规则。

取反

在向量前加上负号，即乘以–1，意味着将向量 180 度转向，如图 8.3 所示。

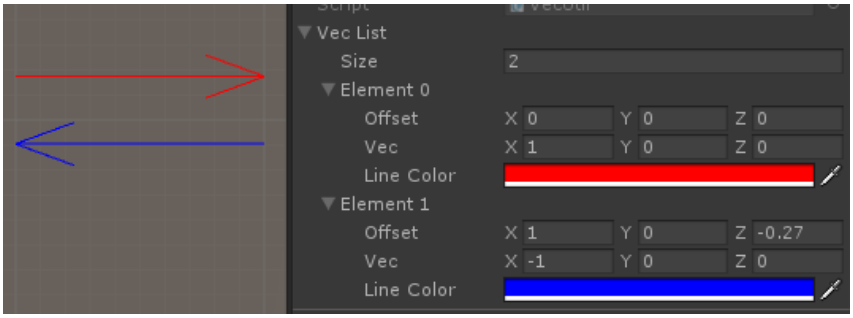


图 8.3

求模

向量有方向，有大小，向量大小的求法遵循公式：

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

它的几何意义就是向量线段的长度。

标量乘以向量

标量与向量的乘法，是将向量的每个分量都与标量相乘。例如：

$$K\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = \begin{bmatrix} ka_1 \\ ka_2 \\ \vdots \\ ka_{n-1} \\ ka_n \end{bmatrix}$$

在几何意义上相当于对向量的长度进行缩放，如果 k 小于 0，则方向倒转。因此前面提到的负向量，可以认为是向量乘以标量－1。

标准化

有时对于向量，我们不关心它的长度，而只关心方向。这种情况下，最好的选择是将向量标准化为单位向量，它的运算法则为：

$$V_{normal} = \frac{V}{\|V\|}$$

在几何上，标准化的向量可以认为是箭尾在圆心，箭头落在单位圆上的向量。

向量的加减

向量之间加减时，它的每个分量各自加减：

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_{n-1} + b_{n-1} \\ a_n + b_n \end{bmatrix}$$

减法也相同，只不过是将其更改为负数。

它的几何解释为：

（1）平移向量，使得尾与头相连接。

（2）从尾向头画一个向量。

（3）这个新向量即为所求。

如图 8.4 所示。

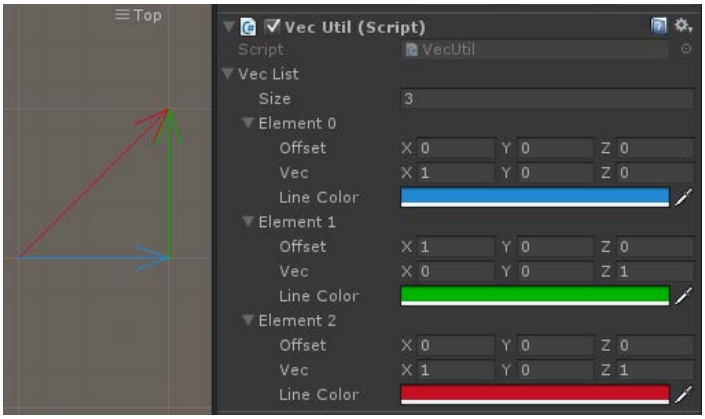


图 8.4

这个几何意义也有一个扩展应用：为了计算两点之间的向量，我们可以将两点做差。这个行为实际上是将点扩展为两个从原点出发的向量。为了求得它们连线的向量，我们将其中一个向量取反，然后做和。

点乘

点乘（Dot）又叫内积，是两个向量之间的一种特殊乘法。它的运算法则如下：

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} = a_1b_1 + a_2b_2 + \cdots + a_{n-1}b_{n-1} + a_nb_n$$

从几何的角度讲，点乘的结果描述了两个向量的相似程度，两个向量越相似，点乘的结果越大。点乘的结果等于向量大小与向量夹角 cos 值的积。

$$A \cdot B = \|A\| \|B\| \cos\theta$$

当向量为单位向量时，点乘的结果完全取决于两向量之间的夹角。因此通过点乘计算夹角也是最常用的做法。

投影

通过点乘我们可以很容易地计算出一个向量在另一个向量上的投影。例如，我们已知和，现在想求在某个方向上的投影，推导过程如下：

$$\because \|V\| = \|A\| \cos\theta$$

$$\because V = B\frac{\|V\|}{\|B\|}$$

$$\therefore V = B\frac{\|A\| \cos\theta}{\|B\|}$$

$$\therefore V = B\frac{\|A\| \|B\| \cos\theta}{\|B\|^2}$$

$$\therefore V = B\frac{A \cdot B}{\|B\|^2}$$

当 ***B*** 为单位向量时，除法部分可以省略。效果如图 8.5 所示。

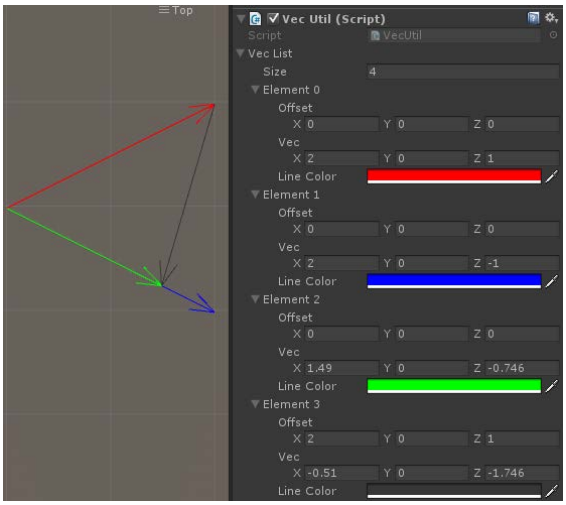


图 8.5

反射向量

在投影的基础上，我们也可以计算反射向量。例如，现在已知入射向量 ***A*** 和法向量 ***H***，求反射向量 ***V***。首先我们先求出向量 ***A*** 在向量 ***n*** 方向上的投影，然后通过向量的和来获得与向量 ***n*** 垂直的向量 ***S***：

$$S = A + (n(-A)n$$

然后根据角平分线的性质求得反射向量 ***H***：

$$\vec{V} = 2\vec{S} - \vec{A}$$
$$\vec{V} = \vec{A} - 2\vec{n}(\vec{A} \cdot \vec{n})$$

反射向量的求解应用了向量的投影和向量加减法，是点乘知识点的绝佳应用，也是常见的面试题。效果图如图 8.6 所示。

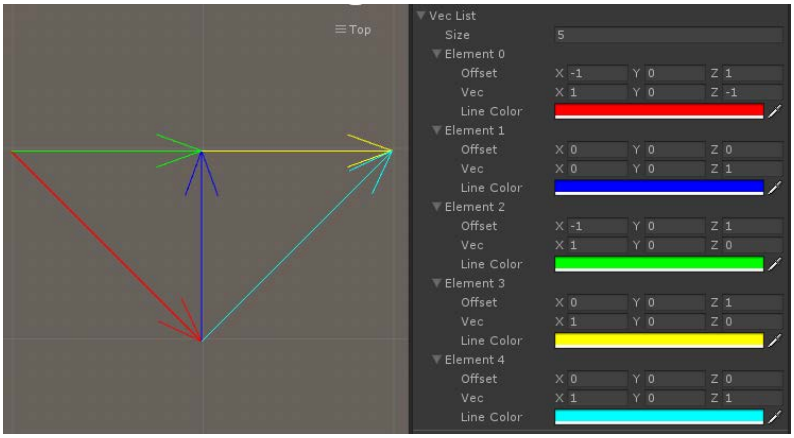


图 8.6

叉乘

点乘（Cross）又叫外积，是两个向量之间的另一种乘法，相关的运算公式如下：

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{bmatrix}$$

从几何角度看，叉乘得到的是垂直于两个向量的向量。如果将这两个向量构成一个平面，叉乘的结果就是这个平面的垂向量。垂向量的方向可以有两个，这取决于我们的坐标系。Unity 3D 的世界坐标系是左手坐标系，因此判断垂线的方法遵循左手定则。即手指方向与第一个向量对齐，四指卷向第二个向量方向，则拇指方向为外积方向。效果如图 8.7 所示。

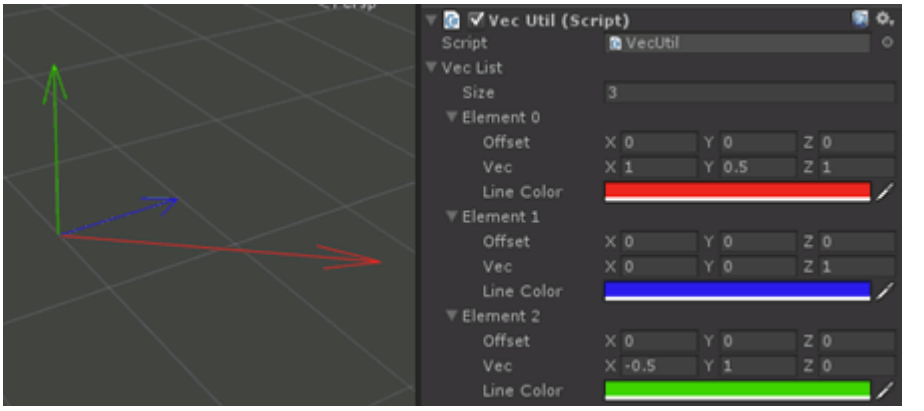


图 8.7

面积公式

从长度上看，有下面的公式：

$$\|a\| \times \|b\| = \|a\| \|b\| \sin\theta$$

对于一个平行四边形来说，两条邻边的叉乘的模就是它的面积。

扩展问题

为什么平行四边形邻边叉乘的模是它的面积（[:rectarea]）？

8.3 区域检测

请问如何检测某点是否在扇形区域内？例如图 8.8 所示的情况。

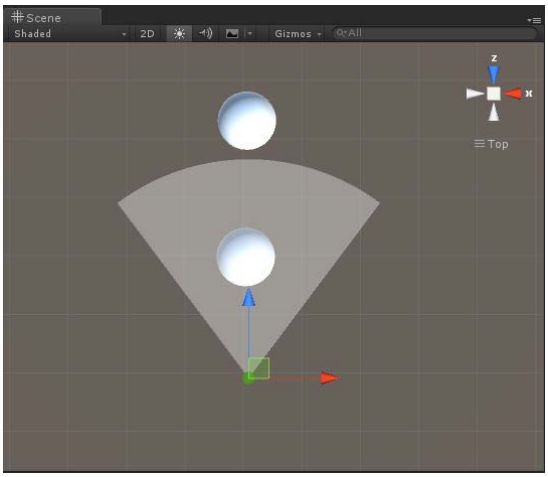


图 8.8

为了简化问题，可以认为检查时，所有点都在同一平面，不受高度 *Y* 的影响。

问题分析

扇形区域是游戏中最常见的范围判断。由于扇形是角色攻击范围的常见形状，因此扇形区域的判断也是常规逻辑。由于扇形的张角可大可小，半径可长可短，因此编写算法时应将它们都作为输入参数。

大体思路是利用向量的点乘。在 8.2 节向量运算中我们了解到，单位向量点乘的结果是它们夹角的 cos 值。由于 cos 在 0~π 上单调调减，因此可以比较 cos 值来达到比较夹角的效果。

搭建测试环境

创建名为 SectorCheck.cs 的类，编写如下代码：

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

[ExecuteInEditMode]
public class SectorCheck : MonoBehaviour
{
    #region Public Attributes
```

```
    public float m_radius;
    public float m_angle;
    public List<GameObject> m_list = new List<GameObject>();
    #endregion

    void Update()
    {
        for (int i = 0; i < m_list.Count; ++i)
        {
            if (checkInArea(m_list[i].transform.position,m_angle, m_radius))
            {
                m_list[i].SetActive(true);
            }
            else
            {
                m_list[i].SetActive(false);
            }
        }
    }

    bool checkInArea(Vector3 tarPos, float angle, float radius)
    {
        return true;
   }
}
```

在新场景中放置 9 个 Cube，按照一定间距摆放。接着创建一个空对象，挂载上面的脚本。将 9 个 Cube 赋值到脚本上，效果如图 8.9 所示。

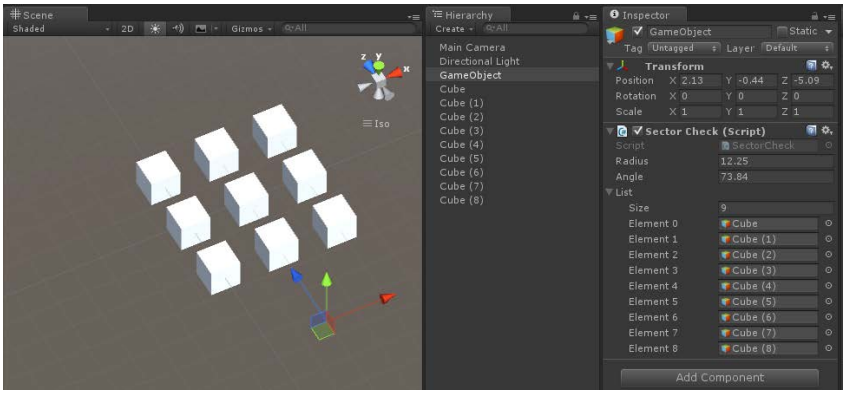


图 8.9

检测算法实现

下面来实现检测逻辑，编写函数 checkInArea，代码如下：

```
bool checkInArea(Vector3 tarPos, float angle, float radius)
{
    var cosAngle = Mathf.Cos(Mathf.Deg2Rad * angle * 0.5f);
    Vector3 circleCenter = transform.position;
    Vector3 disV = tarPos - circleCenter;
    float dis2 = disV.sqrMagnitude;
    if (dis2 < radius * radius)
    {
        disV.y = 0.0f;
        disV = disV.normalized;
        float cos = Vector3.Dot(transform.forward, disV);
        if (cos - cosAngle >= 1e-5 )
        {
            return true;
        }
    }
    return false;
}
```

其中第一个参数为目标点的位置，第二个参数为扇形的夹角，第三个参数为扇形的半径。当目标点在扇形区域内时，函数返回真，否则返回假。

辅助线框

完成上述代码检测后，还需要一个辅助的基准线框。我们需要知道这个扇形的真实范围，以此为依据来检查我们的逻辑判断是否准确。添加一个 Gizmos 函数，绘制一个半透明的扇形，编码如下：

```
[DrawGizmo(GizmoType.Selected | GizmoType.NonSelected)]
static void DrawGiz(SectorCheck scr, GizmoType gizType)
{
    Handles.color = new Color(1, 1, 1, 0.2f);
    var newStart = Quaternion.Euler(new Vector3(0, -scr.m_angle / 2, 0)) *
scr.transform.forward;
    Handles.DrawSolidArc(scr.transform.position, scr.transform.up, newStart,
scr.m_angle, scr.m_radius);
}
```

编辑好后，就可以在界面中看到扇形区域了。我们同时使用 GizmoType.Selected 和 GizmoType.NotSelected 两个参数，保证无论是否选中对象，这个 Gizmo 都会绘制。调整区域到方阵附近，即可验证检测功能，当区域变小时，未覆盖的部分就会隐藏，效果如图 8.10 所示。

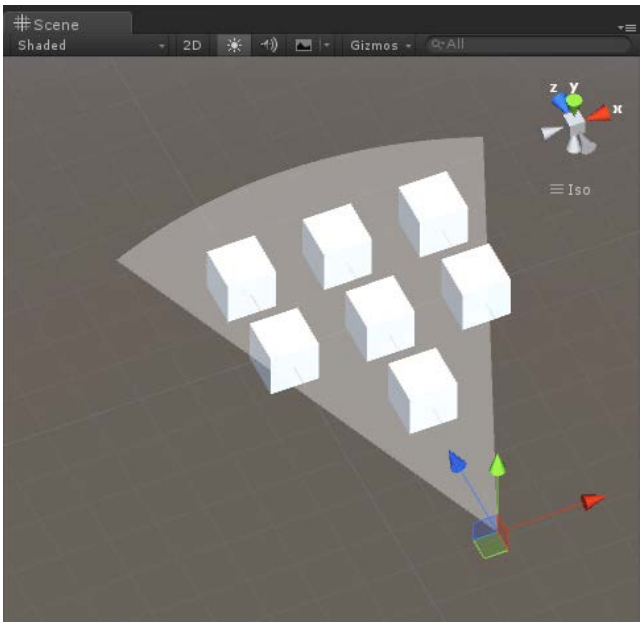


图 8.10

总结

借助点乘的性质可以方便地进行扇形区域的检测。需要注意的是，当角度等于 360 时，即为圆形，建议做特殊判断，直接计算两点距离，以优化检测的效率。

8.4 平面移动

现有一个 3D 卡牌项目，希望实现这样的效果：单击屏幕能将卡牌拾起，拖动时沿着一个平行于桌面的平面移动，松开时将卡牌放下，示意图如图 8.11 所示。

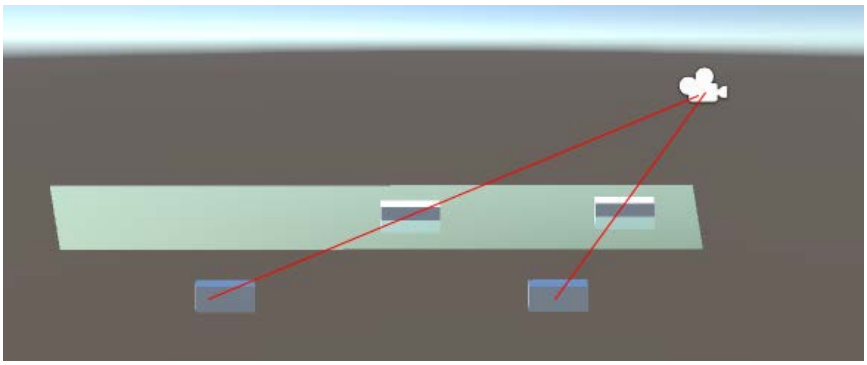


图 8.11

拾取左侧的卡牌，它会沿着视觉射线移动到平面上。当拖动时，它沿着上层的平面移动。释放时，它会沿着视线方向下落到桌面上。

请简要说明如何实现位置计算，并用代码实现。

问题分析

在 3D 卡牌或 SLG 类型的游戏中，常常会有拾起、平移、放置这样的操作。这些移动通常要根据物体的不同位置计算修正点。仔细分析可以发现，这个问题的核心是求一条直线与平面的交点。

平面的表示

我们都知道在 3D 空间中，任意一个平面可以记为：

$$ax + by + cz = \boldsymbol{H}$$

如果我们将其转化为两个向量的点乘则可以写成：

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{H}$$

因此，我们得出了 3D 空间中平面的表达式：

$$\boldsymbol{p} \cdot \boldsymbol{n} = \boldsymbol{d}$$

从原点出发到（x,y,z）的向量，在向量 $\boldsymbol{n}$ 方向上的投影等于 $\boldsymbol{H}$。因此反过来说：这些点是在向量 $\boldsymbol{H}$ 方向上投影为 d 的点的集合。其中也称为平面的法向量，因为它垂直于平面。

交点公式

相比于平面，直线的函数表达式要简单得多：

$$\boldsymbol{P}（t）=Po+tH$$

可以将其看成，从已知点出发向两侧沿着向量无限延伸的点的集合。由此我们很容易地推导出交点的等式，进而求得变量 t。

$$（\boldsymbol{P}o+t\boldsymbol{v}）\cdot\boldsymbol{n}=\boldsymbol{d}$$

$$\boldsymbol{P}o\cdot\boldsymbol{n}+t\boldsymbol{d}\cdot\boldsymbol{n}=\boldsymbol{d}$$

$$t\boldsymbol{d}\cdot\boldsymbol{n}=\boldsymbol{d}-\boldsymbol{p}o\cdot\boldsymbol{n}$$

$$t=\frac{\boldsymbol{d}-\boldsymbol{p}o\cdot\boldsymbol{n}}{\boldsymbol{u}\cdot\boldsymbol{n}}$$

场景搭建

下面依照公式实现需求。创建一个测试场景，场景中放置一个 Cube，调整缩放，使它看起来像张卡牌。然后创建 RayPick 脚本：

```
using UnityEngine;
using System.Collections;

public class RayPick : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKey("mouse 0"))
        {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit))
            {
                Debug.Log("Pick " + hit.collider.gameObject.name);
            }
```

```
        }
    }
}
```

编写好后，在场景中创建一个空对象，将脚本挂在上面。调整摄像机和物体，将它们放在合适的位置上。运行游戏，若单击物体能够看到输出，则说明证明环境搭建正确。

引入公式

接着就实现核心功能，创建函数 calculatePos，编辑代码如下：

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

public class RayPick : MonoBehaviour
{
    public Vector3 m_normal = Vector3.up;
    public float m_d = 1;

    void Update()
    {
        if (Input.GetKey(KeyCode.Mouse0))
        {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit))
            {
                //Debug.Log("Pick " + hit.collider.gameObject.name);
                if (Vector3.Dot(ray.direction.normalized,m_normal) == 0)
                {
                    return;
                }
                hit.collider.transform.position =
calculatePos(Camera.main.transform.position,ray.direction.normalized,m_norm
al,m_d);
            }
        }
    }

    Vector3 calculatePos(Vector3 p0,Vector3 direction,Vector3 normal,float d)
```

```
    {
        float t = (d - Vector3.Dot(p0, normal)) / Vector3.Dot(direction , normal);
        return p0 + t * direction;
    }
}
```

运行游戏，即可看到物体被拾起到平面的效果。完整功能涉及交互控制，这不是本书的重点，所以不做延伸，感兴趣的读者可以自行实现完整功能。

总结

这个问题的难点是如何将几何理论与程序实现相结合。首先，我们将需求中的平面和直线用数学模型表示出来，进而求得计算公式，接着将公式转化成代码实现。整个过程中，解题的思路最重要。因为需求可能千变万化，只有掌握思考方法才能以不变应万变。

扩展问题

1．向量 ***n*** 为什么垂直于平面（[：nisvertical]）？

第9章
寻路算法

在本章中，我们主要从算法层探讨寻路功能的实现。首先，我们要了解常见寻路算法的分类，然后实现最常见的 A*算法，最后看看 Unity 3D 中的 Navigation 系统底层原理是什么。另外，我们还从算法层探讨多任务结构的算法实现，它会被用于 MMO 类型的复杂任务，或自动跨地图寻路，等等。

9.1 寻路这件事

请大致描述 A*（AStar）寻路算法是什么，它与 Dijkstra 算法有什么区别和联系？

问题分析

简单来说，寻路问题就是在给定起点终点的情况下规划出一条合理的路。算法中最关键的地方在于合理。它有很多定义方式，例如，路径最短、转角要平滑、有些区域有更大的权值等。这里我们简化问题，只考虑路径最短的情况。在这个前提下，有个应用广泛的算法，它就是 A*算法。

启发式

广义上讲，A*是一种基于启发式函数（Heuristic）改变节点选择规则的搜索。一般来说，启发式函数算出的值必须要小于等于真实开销，否则就有可能出现其他的最优解。

常用的启发函数是曼哈顿距离和欧几里得距离。曼哈顿距离是标明两个点在标准坐标系上的绝对轴距总和。在 2D 坐标系下算法如下：

$$h(x)=|\text{Start}.x-\text{End}.x|+|\text{Start}.y-\text{End}.y|$$

曼哈顿距离适用于不可以走斜线的条件。它的好处是计算中不会有浮点数的参与，但如果允许斜着走，那么该方法会导致估算距离大于实际值。因此，我们会采用欧几里得距离来计算这种复杂的情况，虽然会产生更大的性能开销，但会获得较好的移动体验。2D 的欧几里得距离算法如下：

$$h(x) = \sqrt{(\text{Start}\cdot x - \text{End}\cdot x)^2 + (\text{Start}\cdot y - \text{End}\cdot y)^2}$$

现在，我们定义 A*使用的启发式函数为 $f(x)$。在运行算法之前，我们需要一些辅助数据，笔者将其称为“两弹一星”。“两弹”是指两个集合，用以存储我们搜索的结果。由于集合的图形表示是圆圈，所以就为它起了这个外号。其中一个为开集（OpenSet），它存储尚未探索周围区域的节点；另一个为闭集（CloseSet），它用于存储已经找过周围的点。“一星”是指我们需要存储每个节点的父节点，当我们找到目标节点时，反向遍历链表就可以找到最短路径。由于存储结构是链表，在 C++中使用星号表示指针，所以将其称为“一星”。

算法开始时，我们先将初始点添加到 CloseSet 中，并将其设置为当前节点，然后开始循环，每次循环要找到当前节点周围相邻的点。

- ◎ 如果它在 CloseSet 中，就跳过本次判断。
- ◎ 如果它们已经在 OpenSet 中，则判断并更新父节点为当前节点，并更改启发式结果值。
- ◎ 如果它们不在 OpenSet 中，就更新它们的父节点为当前节点，计算它的启发式函数值，并加到 OpenSet 中。当 OpenSet 为空时，或者找到目标节点时，循环结束；否则找到 OpenSet 中启发函数最小的值，将它从 OpenSet 中移除，加到 CloseSet 中。最终转置链表，找到路径。

贪心算法

如果编写一个纯粹贪心算法的路径搜索，则启发函数为：

$$f(x) = h(x)$$

可以猜到这种算法下，局部最优解并不是全局最优解，因此这样的算法不能求得最短路径。如图 9.1 所示。

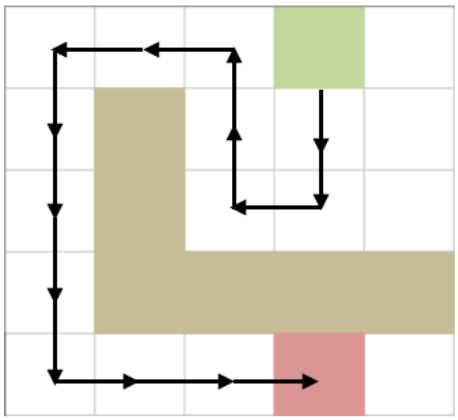


图 9.1

Dijkstra 算法

如果更改启发函数，计算从开始点到当前点经历的步数 $g(x)$，我们就得到了另一个启发式：

$$f(x)=g(x)$$

使用这个启发式可以获得最优解，它被称为 Dijkstra 算法。它的缺点在于搜索范围过大。因为没有距离考量，所以它的搜索范围是圆形的。如果两点相距较远，则几乎会搜索整张地图，效果如图 9.2 所示。

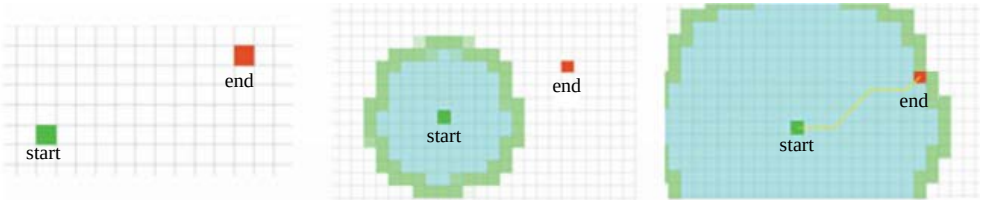


图 9.2

A*算法

为了优化上述算法，我们将贪心算法和 Dijkstra 算法结合起来，更改启发函数为：

$$f(x)=g(x)+h(x)$$

这就是 A*算法，它既能减少搜索的范围，又可以得到最优解。需要注意的是，在执行循环时，需要在 $g(x)$更小的解时更新节点数据，运行效果如图 9.3 所示。

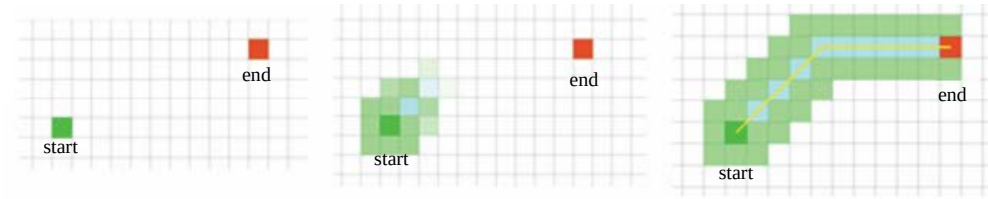


图 9.3

IDA*算法

A*算法基本能满足大多数需求，但有时我们还希望能够进一步优化。在能预估出深度的情况下，我们可以使用 IDA*。它是迭代加深算法与 A*算法的结合。这个算法不是在启发函数上做文章，而是使用迭代加深更改了遍历的逻辑。

迭代加深算法是指在深度优先（DFS）搜索算法的基础上逐步加深搜索的深度。它避免了广度优先搜索占用搜索空间太大的缺点，也就是不需要“两弹”了。

总结

A*算法作为常见的寻路解决方案，用在几乎任何游戏中。另一方面，知道它的演变过程也尤为重要，毕竟实现代码花些时间都可以找到，但优化逻辑就必须了解其内在原理。

扩展问题

请用代码实现 A*算法。

9.2 A*算法

请实现 A*（AStar）寻路算法。为了明确目标，将提供如下实现环境。

已知地图定义：

```
public static int wide = 6;
public static int height = 5;
```

```
public static int[,] pathTable = new int[,]{
    { 0, 0, 0, 0,-1, 0},
    { 0, 0, 0, 0,-1, 0},
    { 0, 0, 0, 0,-1, 0},
    { 0, 0,-1,-1,-1, 0},
    { 0, 0, 0, 0, 0, 0},
};
```

其中 0 为可行走区域，–1 为不可行走区域。wide 为地图的宽，height 为地图的高。

另外，提供了搜索节点结构体：

```
public class SearchNode
{
    public Vector2 pos;
    public float f;
    public float g;
    public float h;
    public SearchNode parent;

    public SearchNode(Vector2 pos)
    {
        this.pos = pos;
        parent = null;
    }
};
```

现在希望实现 FindWay 函数：

```
void find(SearchNode startNode, SearchNode endNode)
```

它可以输出从起始点到终止点的路径。

问题分析

有了 9.1 节的基础，这个问题就会相对容易处理。这部分可以看成是它的延续，是一个程序验证的过程。

搭建循环

根据逻辑，我们首先定义初始化变量，并搭建循环逻辑：

```
void FindWay(SearchNode startNode, SearchNode endNode)
{
    List<SearchNode> openset = new List<SearchNode>();
    List<SearchNode> closeset = new List<SearchNode>();
    SearchNode currentNode = startNode;
    while (!isEqualNode(currentNode,endNode))
    {
        //获取 currentNode 节点周围的节点
        List<SearchNode> adjacentNode = getAdjacent(currentNode);
        for (int i = 0; i < adjacentNode.Count; ++i)
        {
            //计算 f(x)，并做集合操作
        }

        if (openset.Count == 0)
        {
            break;
        }

        //找到 f(x)最小的节点
        currentNode = getMinFNode(openset);
        RemoveNode(openset,currentNode);
        closeset.Add(currentNode);
    }

    //输出结果逻辑
}
```

在上面的代码中，可以看到大体流程，我们查找每个点的四周点，并将 OpenSet 中最小的值放到 CloseSet 中。

启发式计算

接着我们着重实现 *f*(*x*)的计算。编写如下代码：

```
void FindWay(SearchNode startNode, SearchNode endNode)
{
    List<SearchNode> openset = new List<SearchNode>();
       List<SearchNode> closeset = new List<SearchNode>();
       SearchNode currentNode = startNode;
    while (!isEqualNode(currentNode,endNode))
```

```
{
    //获取 currentNode 节点周围的节点
    List<SearchNode> adjacentNode = getAdjacent(currentNode);
    for (int i = 0; i < adjacentNode.Count; ++i)
    {
        //计算 f(x), 并做集合操作
        SearchNode tryNode = adjacentNode[i];
        if (ContainNode(closeset,tryNode))
        {
            continue;
     }
     //OpenSet 中包含节点
       if (ContainNode(openset,tryNode))
       {
           //以当前节点为基础, 计算 g
           var cur_g = currentNode.g+1;
           //有更优的解
           if (cur_g < tryNode.g)
           {
               tryNode.parent = currentNode;
               tryNode.g = cur_g;
               tryNode.f = tryNode.g + tryNode.h;
           }
       }
       else
     {
          //OpenSet 中不含节点
           tryNode.parent = currentNode;
           tryNode.h = Mathf.Abs(tryNode.pos.x - endNode.pos.x) +
                       Mathf.Abs(tryNode.pos.y - endNode.pos.y);
           tryNode.g = currentNode.g+1;
           tryNode.f = tryNode.g + tryNode.h;
           openset.Add(tryNode);
       }
    }

    if (openset.Count == 0)
    {
        break;
    }

    //找到 f(x)最小的节点
```

```
        currentNode = getMinFNode(openset);
        RemoveNode(openset,currentNode);
        closeset.Add(currentNode);
    }

    //输出结果逻辑
}
```

上面这段函数中有些未实现的方法，会在后面实现。为了不干扰逻辑理解，请先认为它是能够正确运行的函数。

输出结果

最终我们需要将链表反转，得到整个寻路过程：

```
void FindWay(SearchNode startNode, SearchNode endNode)
{
    List<SearchNode> openset = new List<SearchNode>();
    List<SearchNode> closeset = new List<SearchNode>();
    SearchNode currentNode = startNode;
    //......
    //输出结果逻辑
    if (isEqualNode(currentNode, endNode))
    {
        //使用栈反转链表
        Stack<SearchNode> path = new Stack<SearchNode>();
        SearchNode node = GetNode(closeset, endNode);
        while (node != null)
        {
            path.Push(node);
            node = node.parent;
        }

        //输出结果
        string str = "Path is: ";
        node = path.Pop();
        while (path.Count > 0)
        {
            str += " [" + (int)node.pos.x + "," + (int)node.pos.y + "] ";
            node = path.Pop();
        }
```

```
        //在输出中增加最后的节点
        str += " [" + (int)endNode.pos.x + "," + (int)endNode.pos.y + "] ";
        Debug.Log(str);
    }
    else
    {
        Debug.Log("Can't find the way!");
    }
}
```

集合函数的实现

最后再说说上文未实现的集合相关函数。由于我们比较的基础是 PathTable，因此集合的相关操作也都应基于坐标的运算。

```
SearchNode GetNode(List<SearchNode> set, SearchNode node)
{
    for (int i = 0; i < set.Count; ++i)
    {
        if (isEqualNode(set[i],node))
        {
            return set[i];
        }
    }
    return null;
}

bool ContainNode(List<SearchNode> set,SearchNode node)
{
    for (int i = 0; i < set.Count; ++i)
    {
        if (isEqualNode(set[i],node))
        {
            return true;
        }
    }
    return false;
}

void RemoveNode(List<SearchNode> set, SearchNode node)
{
```

```
    for (int i = 0; i < set.Count; ++i)
    {
        if (isEqualNode(set[i],node))
        {
            set.RemoveAt(i);
            return;
        }
    }
}

SearchNode getMinFNode(List<SearchNode> openset)
{
    int index = 0;
    float min = wide+height;
    for (int i = 0; i < openset.Count; ++i)
    {
        if (min > openset[i].f)
        {
            min = openset[i].f;
            index = i;
        }
    }
    return openset[index];
}

List<SearchNode> getAdjacent(SearchNode centerNode)
{
    Vector2 up = new Vector2(centerNode.pos.x, centerNode.pos.y - 1);
    Vector2 down = new Vector2(centerNode.pos.x, centerNode.pos.y + 1);
    Vector2 left = new Vector2(centerNode.pos.x - 1, centerNode.pos.y);
    Vector2 right = new Vector2(centerNode.pos.x + 1, centerNode.pos.y);

    List<SearchNode> nodeList = new List<SearchNode>();
    if (isValidPos(up))
    {
        nodeList.Add(new SearchNode(up));
    }

    if (isValidPos(down))
    {
        nodeList.Add(new SearchNode(down));
    }
```

```
    if (isValidPos(left))
    {
        nodeList.Add(new SearchNode(left));
    }

    if (isValidPos(right))
    {
        nodeList.Add(new SearchNode(right));
    }

    return nodeList;
}


bool isEqualNode(SearchNode a, SearchNode b)
{
    return ((int)a.pos.x == (int)b.pos.x &&
        (int)a.pos.y == (int)b.pos.y);
}

bool isValidPos(Vector2 pos)
{
    if (pos.x < 0 || pos.x > wide-1 ||
        pos.y < 0 || pos.y > height-1)
    {
        return false;
    }
    return pathTable[(int)pos.y,(int)pos.x] == 0 ? true:false;
}
```

调用测试

实现了上述代码后，添加一个调用入口：

```
void Start()
{
    SearchNode start = new SearchNode(new Vector2(0, 1));
    SearchNode end = new SearchNode(new Vector2(5, 1));
    FindWay(start, end);
}
```

将脚本挂载到空的 GameObject 上，运行游戏，可看到如下输出：

```
Path is:  [0,1]  [1,1]  [1,2]  [1,3]  [1,4]  [2,4]  [3,4]  [4,4]  [5,4]  [5,3]
[5,2]  [5,1]
```

总结

本节的主要内容是以最简单的方式实现 A*的逻辑。依然是那句话，代码不重要，重要的是理解过程。

扩展问题

（1）集合的所有操作不同之处仅在于比较两个节点的坐标，有什么办法简化编程方式吗（[:astarpos]）？

（2）两个集合都是基于 List 的，是否有办法进行优化呢（[:astarset]）？

9.3 Navigation 系统

在 Unity 3D 中，我们通常会使用导航网格（Navigation）做寻路，但有时也会对场景直接划分格子，并自己实现 A*算法。请问 Navigation 系统与传统的寻路方式有联系吗，它们的优劣又该如何取舍呢？

问题分析

角色移动时，通常不是在网格上一格格走，因此在非 SGL 游戏中，会采用寻路节点或导航网格的形式。

寻路节点是指，关卡设计师在游戏世界中摆放的角色可到达的位置。这些点会被作为图的节点载入内存中。边则通过点与点之间的组合自动生成。它最大的问题在于，角色只能在节点和对应边缘移动。因此地图上会有很多不能走的镂空区域。为了更精确地描述这些区域，又需要添加大量的寻路节点，进而导致寻路算法花费的时间变长。

导航网格则不存在上述问题。它通过凸多边形来表示节点，临近节点就是相邻的凸多边形。一般情况下，用远小于寻路节点数量的凸多边形就可以将游戏世界区域表示出来。而且由于封闭区域可行走，因此返回的路径会更加自然。

另一方面，A*是启发式优化寻路算法的统称，导航网格与其并不在同一层级。无论是通过凸多边形还是网格，说到底都是对地图信息的一种描述，拓扑这些信息之后，可以使用相同的算法运算。因此我们可以通过 A*来实现 Navigation。接下来就看看 Navmesh 是如何与 A*结合在一起的。

与 A*结合

下面通过 Navmesh 示意图来说明，如图 9.4 所示。在实际工作中，这个结构可以通过 Unity 引擎生成。

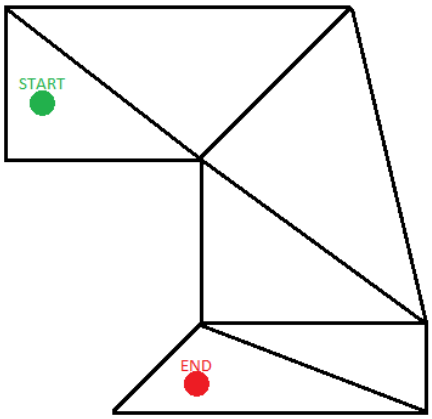


图 9.4[1]

在图 9.4 中我们可以看到起始点和终止点，现在希望寻找出一条最近的路径。

首先，找到起始点和终止点所在的三角形。我们可以通过点乘等方式实现这样的算法。如果两点在相同的三角形中，直接走过去即可。如果两点中有任何一点不在任何三角形中，就需要做不可达判断。对于 Unity 3D 来说，会在一定范围内找到 Navmesh 上最接近的三角形。更多的情况是两点不在同一三角形上，这时就需要采用寻路策略。

以 A*算法为例，我们找到三角形的三个顶点，使用欧几里得距离作为启发式函数，找到 $f(x)$ 最小值，将对应的点加到 CloseSet 中，之后就与上 9.2 节中谈到的寻路算法相同了，如图 9.5 所示。

1　图片源于：https://medium.com/@mscansian/a-with-navigation-meshes-246fd9e72424

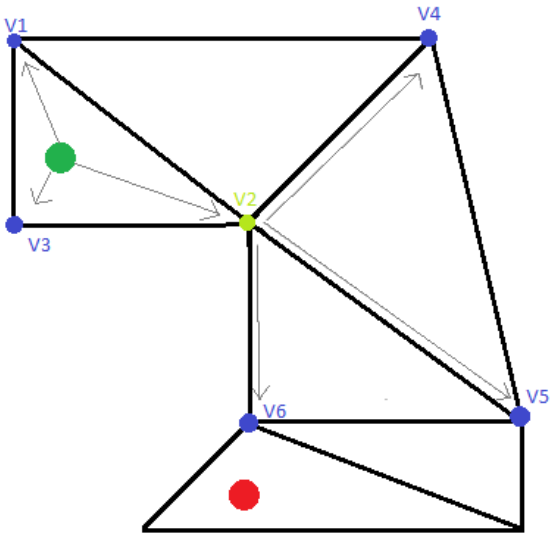


图 9.5

重复这个步骤直到找到目标三角形，如图 9.6 所示。

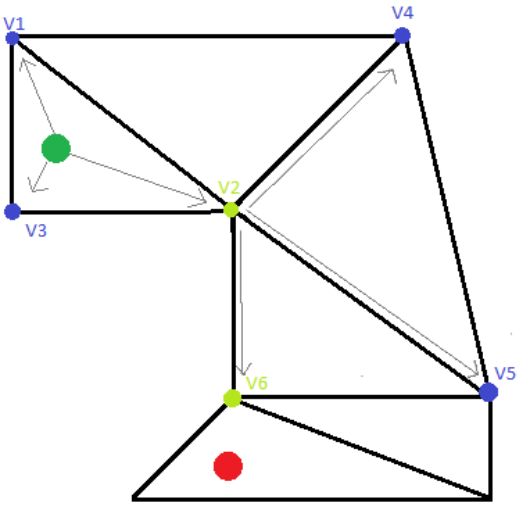


图 9.6

最终在目标三角形中做两点间移动，如图 9.7 所示。

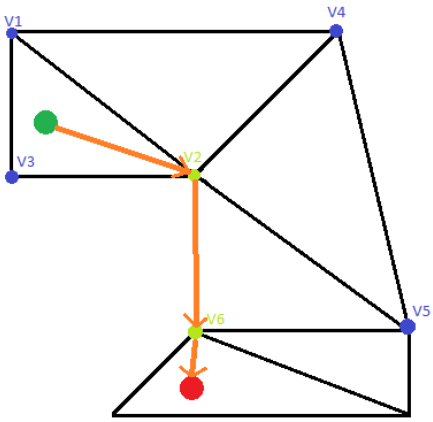


图 9.7

上面描述的是最简单的实现方式。在实际过程中，移动路径通常不会沿着顶点进行，所以这里会对一些算法做优化，使得路径更加平滑。

传统寻路的优势

如果不使用导航网格，而是使用格子划分或寻路节点，自己实现 A*也并非不可取。有些情况下，我们对角色移动的精度与流畅度要求不是很高，为了节省运算效率，也会权衡做出选择。

总结

虽然 Unity 3D 引擎已经封装好了 Navigation，但对其内在原理的研究依然具有价值。当需要服务器端计算或验证导航网格寻路，或想要对整套系统优化时，熟悉它的原理就事半功倍了。

扩展问题

请问为什么说导航网格比较适用于地形复杂的场景（[:nav]）？

9.4　任务调配

策划人员为了增加游戏的趣味性，通常会配置很多的分枝任务，比如：

```
接取吃饭任务->找碗->盛饭……
            ->找到筷子……吃饭
```

不过随着配置变得复杂，策划有些担心自己配置错了导致任务无法完成，所以他希望你能帮忙写一个验证程序检查一下是否存在这种情况。

具体来说，一共有 *n* 个任务，它们的 ID 从 0 到 *n*-1。有些任务有前置任务，例如，如果你要接任务 1，必须先完成任务 0，则表示为一个键值对[1,0]。如果恰好出现循环任务，即要想接任务 0，则必须完成任务 1，可任务集合表示为：[[1,0],[0,1]]。那么这样的任务队列就是不可能完成的。

现在已知任务数量 *n* 和它们的关系集合，求证是否可以完成任务，请实现下面的函数：

```
public class Solution {
    public bool CanFinish(int numCourses, int[,] prerequisites) {

    }
}
```

其中参数 numCourses 为课程数量，prerequistites 为任务的边列表，请返回任务是否能完成。

问题分析

分析任务节点之间的关系，我们需要从图的角度入手。任务之间存在前置任务这种单向关系，因此图是有向图。因此这个问题就变成：判断有向图能否覆盖所有节点，且不存在环。

拓扑排序

对于一个有向无环图（Directed Acyclic Graph），可以使用拓扑排序来计算任务完成的顺序。这个算法的核心在于，只关注一个节点的入度。所谓入度，指的是在图中每存在一条指向节点的边，就称为一个入度。不难想象，如果一个任务要想从某个点开始，那么它的入度一定为 0。如图 9.8 所示。

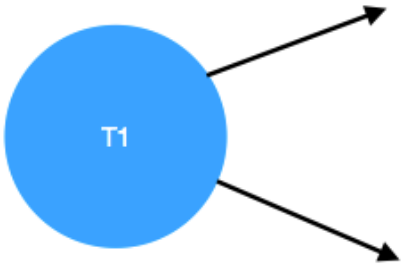


图 9.8

找到这个点后，我们将与它关联的点的入度减 1，并将这个点从整个网络中删掉。删掉后，我们发现整个图又回到了上一步的状态，我们可以继续在图中找到入度为 0 的节点并再次判断。最终如果入度表中没有元素，则说明所有任务都完成了。逻辑并不复杂，下面我们看一下如何通过代码实现。

初始化数据

由于输入数据为边的数组，所以我们需要读取数据，将其转化为邻接矩阵和入度数组。

```
public bool CanFinish(int numCourses, int[,] prerequisites) {
    //初始化数据
    int[] list = new int[numCourses];
    int[,] matrix = new int[numCourses,numCourses];
    for(int i = 0;i<prerequisites.GetLength(0);++i)
    {
        list[prerequisites[i,0]]++;
        matrix[prerequisites[i,1],prerequisites[i,0]]=1;
    }
    //计算入度

    return  true;
}
```

通过遍历数组，将对应的信息存入数据结构中。

循环删除

接着我们需要编写一个循环。找到入度为 0 的点，将它关联的点的入度都减 1，并将它从入度列表中删掉。这里删掉这个元素的方式是将它的入度标记为异常值–1。如果在队列中再也

找不到任何入度为 0 的元素，那么证明已没有需要完成的任务，就退出循环。最后检查入度队列是否有可用元素，如果有，则证明任务无法完成。编辑代码如下：

```
public bool CanFinish(int numCourses, int[,] prerequisites) {
    //初始化数据
    int[] list = new int[numCourses];
    int[,] matrix = new int[numCourses,numCourses];
    for(int i = 0;i<prerequisites.GetLength(0);++i)
    {
        list[prerequisites[i,0]]++;
        matrix[prerequisites[i,1],prerequisites[i,0]]=1;
    }

    //计算入度
    while(true){
        //找到入度为 0 的标记
        int start = -1;
        for(int i = 0;i<list.Length;++i)
        {
            if(list[i] == 0){
                start = i;
                list[i] = -1;
                break;
            }
        }
        //未找到入度为 0 元素，无可完成任务
        if(start == -1)
        {
            break;
        }

        //减少关联节点的入度
        for(int i = 0;i<numCourses;++i)
        {
            if(matrix[start,i] !=0)
            {
                list[i]--;
            }
        }
    }
```

```
    //检查任务完成情况
    for(int i = 0;i<numCourses;++i){
        if(list[i] != -1){
            return false;
        }
    }
    return  true;
}
```

运行代码即可输出任务的可行性。

总结

拓扑排序作为图的基础算法，并不是难点。这里提出这个问题，是希望大家能多多思考，如何通过算法优化我们的日常工作。配置有可能出错，但不一定会被测试到，写个保障性的工具，利人利己，可以提升项目组的整体工作效率。

扩展问题

如果希望在函数运行结束后，返回任务完成的顺序数组，该如何实现呢（[:taskcalc]）？

第四部分

自定义扩展

程序员大多是天生喜欢做工具辅助自己工作的人。很多经验丰富的大牛都有随身携带自制工具集的习惯。在第四部分，我们将从工具集的不同方向深入分析，探讨制作工具的注意事项，并指导大家扩充自己的工具箱。

眼见为实。面对未知问题，能直观地控制和看到信息非常重要。在第 10 章，我们将在指令与绘制方面探寻可能。

看到更多后，就希望将其记录下来。在第 11 章，会分析日志工具应该包含的功能。

在第 12 章，我们将聚焦如何提升日常操作的便利性。

第 13 章，我们会制作一些底层逻辑，来实现更强的功能。

需要注意的是，编辑器扩展无穷无尽，不能指望在短期内完成各种工具链的开发。而且，所有的编辑器功能都应与项目相关。不要指望开发出一个覆盖各种功能的 ToolBox，只有适合项目的，才是最好的。

第 10 章
调试工具

虽然在 Unity 3D 中，调试功能已经很强大了，但更多时候，我们还是希望有切合自己项目的调试工具。本章将探讨 GM 指令的框架如何实现，以及对曲线和区域绘制展开相关讨论。

10.1 GM 命令

随着游戏的开发，越来越多的功能需要测试。在界面上添加按钮已经无法覆盖到所有功能了。这就需要有一种更方便的方式来调用一个接口来完成操作或服务器请求。常见的方式是在聊天中输入一串特殊命令，触发一个逻辑函数。请设计一个代码框架结构，方便在频繁添加功能调用的同时，兼顾代码的整洁性。

问题分析

GM 指令与菜单操作很像，都是调用几行代码。此处采用面向过程的方式，将逻辑封装到函数层最为方便。但功能函数的数量很大，常常要加入很多功能函数。如果只是通过 if-else 来做区分，则需要写很多判断条件，通过工厂模式又要写很多无用代码。这里我们可以借助 C# 的属性（Attribute），将这些重复的结构代码抽出来，从而保持功能和信息的完整性。

属性类设计

明确需求后，我们抽象出的必要属性为以下变量：

◎ 命令名称
◎ 参数数量

◎　用法

我们创建一个 GMCommond.cs 的文件，在其中编写一个 MCommondAttribute 的属性类：

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

[AttributeUsage(AttributeTargets.Method)]
public class GMCommondAttribute : Attribute
{
    public string Cmd;
    public int ParamNum;
    public string Usage;
    public GMCommondAttribute(string cmd,int paramNum, string usage)
    {
        Cmd = cmd;
        ParamNum = paramNum;
        Usage = usage;
    }
}
```

可以看到属性有三个参数，第一个字符串为命令名称，第二个整数为参数个数，第三个字符串为使用说明。下面将这些信息分别存到函数的属性中。

命令实现

有了属性我们就可以编写修饰函数了。再创建一个名为 GMModule 的类，在其中添加两个命令：

```
public class GMModule
{
    [GMCommond("userId", 0, "userId | 显示玩家 ID")]
    public string showId(string[] args)
    {
        int userId = 666;

        //Query logic to get ID
```

```
        return "User id is:" + userId;
    }

    [GMCommond("lvUp", 1, "lvUp 80 | 升级到 xx ")]
    public string levelUp(string[] args)
    {
        //ask server to level up

        return "level up to " + args[0];
    }
}
```

使用 GMCommond 修饰函数 showId 与 levelUp，GM 指令的信息会被存储起来供后面调取。

调用封装

那么如何调用这些函数呢，这也是整个模块运转的关键。简单说是利用反射，将有属性修饰的函数提取出来，根据其中的信息，保存它们的 MethodInfo。我们在 GMModule 函数中添加扩展逻辑：

```
public class GMModule
{

    #region Private Attributes
    private static GMModule _instance;
    public static GMModule Instance
    {
        get
        {
            if(_instance == null)
            {
                _instance = new GMModule();
                _instance.Init();
            }
            return _instance;
        }
    }

    private Dictionary<string, MethodInfo> m_methods = new Dictionary<string,
                                                        MethodInfo>();
    #endregion
```

```
#region Public Methods
public void Init()
{
    m_methods.Clear();

    System.Type type = typeof(GMModule);
    var methods =type.GetMethods();
    foreach(var each in methods)
    {
        var attribute =
           each.GetCustomAttributes(typeof(GMCommondAttribute), false);
        if(attribute != null && attribute.Length>0)
        {
            GMCommondAttribute gmc = attribute[0] as GMCommondAttribute;
            m_methods.Add(gmc.Cmd, each);
        }
    }

}

public string Call(string input)
{
    var tmpStr = input.Split(' ');
    if (m_methods.ContainsKey(tmpStr[0]))
    {
        List<string> param = new List<string>();
        for (int i = 1; i < tmpStr.Length; ++i)
        {
            param.Add(tmpStr[i]);
        }

        var method = m_methods[tmpStr[0]];
        var info = method.GetCustomAttributes(typeof(GMCommondAttribute),
                   false)[0] as GMCommondAttribute;

        if (param.Count != info.ParamNum)
        {
            return "Usage: "+info.Usage;
        }
        else
        {
```

```
                return m_methods[tmpStr[0]].Invoke(this, new object[]
{ param.ToArray() }) as string;
            }
        }
        else
        {
            return "Commond Not Found!";
        }
    }
    #endregion

    //......
}
```

可以看到，在上面这段代码中，我们将 GMModule 扩展为单例，并在初始化函数中注册所有的 GMCommond。在调用时，根据不同的 Key 来分别找到对应的 MethodInfo，通过反射调用对应函数。

测试类

最后，验证功能，创建测试文件 GMTest.cs，在其中添加测试类 GMTest：

```
public class GMTest : MonoBehaviour
{
    string inputText = "";
    void OnGUI()
    {
        inputText = GUILayout.TextField(inputText, GUILayout.Height(30),
GUILayout.Width(200));


        if (GUILayout.Button("Submit", GUILayout.Width(100),
GUILayout.Height(50)))
        {
            Debug.Log(GMModule.Instance.Call(inputText));
        }
    }
}
```

运行项目，在输入框中分别输入下面每行内容，并分别单击“Submit”按钮。

```
userId 30001
userId
lvUp
lvUp 60
```

效果如图 10.1 所示。

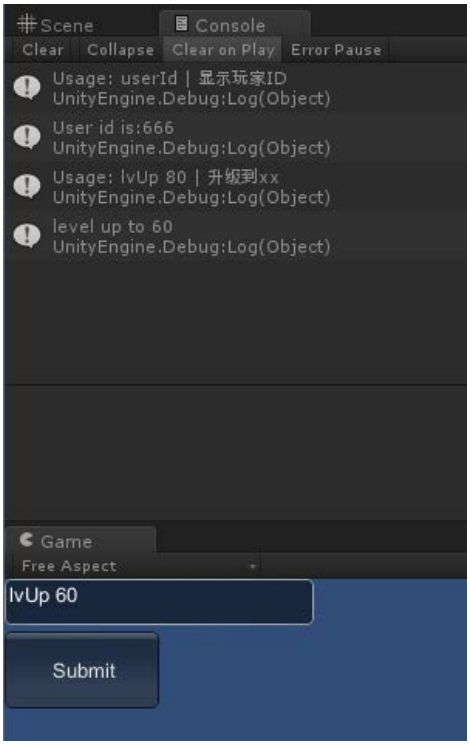


图 10.1

总结

通过反射遍历类函数属性，然后自动添加有属性修饰的类成员，完成调用信息注册。通过这样的形式，可以让我们在添加 GM 指令时专注于内容的编写，而不至于分散精力到其他无关紧要的细枝末节中。

10.2 绘制曲线

现游戏中有个飞行子弹轨迹出现了问题，理论上应该照正弦曲线运动，但实际情况似乎有些偏离。请给出一个方案，将现有轨迹绘制在场景中，确认算法是否存在问题。

问题分析

轨迹运动是游戏很重要的组成部分。通常来说，轨迹可以用函数来表示，即通过一个公式 $y=f(x)$，当变量 x 不断增加时，能得到对应的 y。在这个例子中，每经过一帧，x 都增加一帧的值，再通过公式计算 y 的值，将物体的坐标设定到 y 值对应的位置。从动态效果来看，即完成了沿轨迹移动。因此，只要保证 $y=f(x)$是正确的，即可保证曲线轨迹正确。

能够直观地看到每个点的位置，是大体确认算法是否正确的必经之路，那么如何绘制轨迹的曲线呢？

初中的函数知识告诉我们，是将每个 x 对应的 y 都绘制到场景中，就能看到函数曲线。这里我们可以加个阈值，比如，x 以 0.1 的增量变大，来得到 y 的离散点。然后将这些 y 值连接起来，就是我们需要的曲线了。连接这些点可以使用 Unity 的画线 API:

```
Debug.DrawLine
```

通常来说，在直观看到曲线后，就能够定位问题。如果依然无法确认，就需要根据对曲线的估计，缩小测试范围，在可疑的区间段内，通过在日志中输出可预期的值来排查问题。

画线示例

下面就动手写一个绘制正弦曲线的功能。首先创建一个名为 DrawCustLine.cs 的文件，在其中编写如下代码：

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

[ExecuteInEditMode]
public class DrawCustLine : MonoBehaviour
{
    #region Public Attributes
    public Transform target;
```

```
    public float Height = 1;
    public float Length = 2;
    public float Offset = 0;
    public float TotalLength = 8;

    [Range(0.1f,1)]
    public float delat = 0.1f;
    #endregion

    #region Unity Messages
    void Update()
    {
        makeSin(target);
    }
    #endregion

    #region Public Methods
    public void makeSin(Transform tarTrans)
    {
        if(tarTrans== null)
        {
            return;
        }
        Debug.DrawLine(tarTrans.position,
tarTrans.TransformPoint(Vector3.forward * TotalLength));

        Vector3 PerPos = tarTrans.position;
        for (float i = 0; i < TotalLength; i = i + delat)
        {
            float h = Mathf.Sin(i * Length + Offset) * Height;
            Vector3 localPos = new Vector3(0, h, i);
            Vector3 worldPos = tarTrans.TransformPoint(localPos);
            Debug.DrawLine(PerPos, worldPos, Color.green);
            PerPos = worldPos;
        }
    }
    #endregion
}
```

首先，我们将函数的相位、振幅、波长、总长，开放到 Inspector 中。然后绘制部分，我们

为曲线绘制一条基准线，来做 *X* 轴，接着分别计算出增量 delat 的 *x* 值对应的点的位置，最后在先前点与当前点之间画线即可绘制出曲线。

脚本编辑完成后，我们在场景中创建一个空的 GameObject，然后将脚本关联到上面，即可看到曲线，如图 10.2 所示。

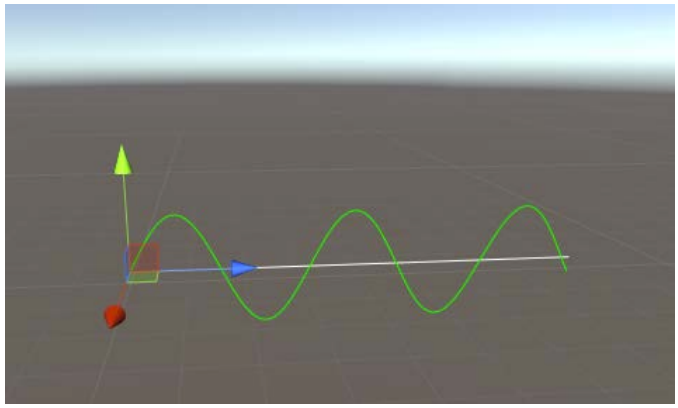


图 10.2

封装接口

目前我们成功绘制了正弦曲线，接下来要做的是封装接口，抽离出曲线算法。具体重构方式如下：

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

[ExecuteInEditMode]
public class TestLine : MonoBehaviour
{
    #region Public Attributes
    public Transform target;

    public float Height = 1;
    public float Length = 2;
    public float Offset = 0;
    public float TotalLength = 8;
```

```
    [Range(0.1f, 1)]
    public float delat = 0.1f;
    #endregion

    #region Private Attributes

    #endregion

    #region Unity Messages

    void Update()
    {
        //makeSin(target);

        makeLine(target, (float x) =>
        {
            float h = Mathf.Sin(Length * x + Offset) * Height;
            Vector3 localPos = new Vector3(0, h, x);
            return localPos;
        });
    }

    #endregion

    #region Public Methods

    public void makeLine(Transform tarTrans, Func<float, Vector3> calcPos)
    {
        if (tarTrans == null)
        {
            return;
        }
        Debug.DrawLine(tarTrans.position,
tarTrans.TransformPoint(Vector3.forward * TotalLength));

        Vector3 PerPos = tarTrans.position;
        for (float i = 0; i < TotalLength; i = i + delat)
        {
            Vector3 localPos = calcPos(i);
            Vector3 worldPos = tarTrans.TransformPoint(localPos);
            if (i > 0)
            {
```

```
                Debug.DrawLine(PerPos, worldPos, Color.green);
            }
            PerPos = worldPos;
        }
    }
    #endregion
}
```

这里我们用到匿名函数的知识，将点的计算封装成匿名函数，传入绘制方法中。函数的参数为 x，返回值为对应点的位置。这样，我们就可以在匿名函数中，调用现有的算法绘制它的曲线了。

扩展性

这里我们可以更进一步，做成更通用的功能：绘制任意算法的曲线。为了达成这个目的，我们需要将函数的参数抽象出来，作为匿名函数的参数，这样就可以应对不同的算法了。在下面的代码中，添加了一个分段函数演示：

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

[System.Serializable]
public class LineParam
{
    public float Height = 1;
    public float Length = 2;
    public float Offset = 0;
    public float TotalLength = 8;
}

[ExecuteInEditMode]
public class TestLine : MonoBehaviour
{
    #region Public Attributes
    public Transform target;
    public LineParam lineParam1;

    public Transform target2;
    public LineParam lineParam2;
```

```
[Range(0.1f, 1)]
public float delat = 0.1f;
#endregion

#region Private Attributes

#endregion

#region Unity Messages

void Update()
{
    //makeSin(target);

    makeLine(target, lineParam1, (float x, LineParam param) =>
    {
        float h = Mathf.Sin(param.Length * x + param.Offset) * param.Height;
        Vector3 localPos = new Vector3(0, h, x);
        return localPos;
    });

    makeLine(target2, lineParam2, (float x, LineParam param) =>
    {
        if (((int)x) % 2 == 0)
        {
            Vector3 localPos = new Vector3(0, -param.Height, x + param.Offset);
            return localPos;
        }
        else
        {
            Vector3 localPos = new Vector3(0, param.Height, x + param.Offset);
            return localPos;

        }
    });
}

#endregion

#region Public Methods
```

```
    public void makeLine(Transform tarTrans, LineParam param, Func<float,
                         LineParam, Vector3> calcPos)
    {
        if (tarTrans == null)
        {
            return;
        }
        Debug.DrawLine(tarTrans.position, tarTrans.TransformPoint
                       (Vector3.forward * param.TotalLength));

        Vector3 PerPos = tarTrans.position;
        for (float i = 0; i < param.TotalLength; i = i + delat)
        {
            Vector3 localPos = calcPos(i, param);
            Vector3 worldPos = tarTrans.TransformPoint(localPos);
            if (i > 0)
            {
                Debug.DrawLine(PerPos, worldPos, Color.green);
            }
            PerPos = worldPos;
        }
    }
    #endregion
}
```

我们定义了 `LineParam` 结构，用来包装各种函数的参数。从目前的需求出发，四个参数已经足够，如果有需要更多参数的情况，就需要灵活扩充了。运行效果如图 10.3 所示。

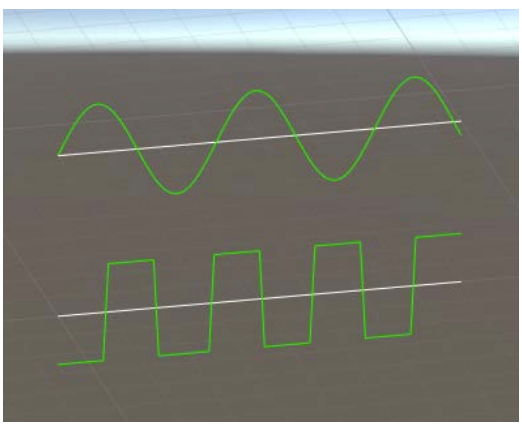


图 10.3

总结

本节我们介绍了如何通过可视化的方法，排查曲线算法的问题。这种方法在实际项目中十分常用，大家可以根据自己的需要对绘制函数做深入的定制，例如，更改颜色、扩展参数、调整步长等。

10.3　指示绘制

游戏角色常常需要绘制一些与自身相关的范围标志。例如，巡逻的视野范围、攻击的伤害范围、机关的预警范围等。现在希望制作一个范围绘制器，能选择方形、扇形或者环形，并配置相应的参数，完成精确的范围绘制。

问题分析

指示绘制最大的特点是范围精确。例如，策划人员配置了夹角 120 度、攻击半径为 5 的扇形，那么角色稍稍站在这个范围之外，就不应该受到伤害。通常的做法是通过代码控制模型面片的形状，进而控制绘制范围。另一方面，面片也要支持自定义材质，这样才能满足美术的表现需求。

编写框架类

明确了想法后，我们首先创建 RangeShower.cs，并创建对应的类 RangeShower，在其中添加 updateMesh 函数，作为更新 Mesh 的函数入口，并在 Update 中调用它：

```
using UnityEngine;
using System.Collections;


[ExecuteInEditMode]
[RequireComponent( typeof( MeshRenderer ) )]
[RequireComponent( typeof( MeshFilter ) )]
public class RangeShower : MonoBehaviour
{

    public enum ShapeType
    {
        none,
```

```
        sector,
        obb,
        ring,
    }

    #region Public Attributes
    public ShapeType m_type = ShapeType.none;
    public float m_degree = 120;
    public float intervalDegree = 10;
    public float m_radius = 5;
    public float m_InnerOff= 2;
    public Material m_circleIndicator;
    public Material m_RectIndicator;
    #endregion

    #if UNITY_EDITOR
    void Update()
    {
        if (!Application.isPlaying)
        {
            updateMesh(m_type, m_degree, m_radius, m_InnerOff);
        }
    }
    #endif

    public void updateMesh(ShapeType shape, float degree,float radius,float
innerOff) {
        //logic to make Mesh
    }
}
```

这里重写了 Update 函数，在编辑器非运行模式下，会每帧刷新 Mesh，做 Application.isPlaying 判断主要是出于性能的考虑。updateMesh 函数主要用于根据参数创建对应的面片，参数分别为形状类型、角度、半径和内切半径。最后一个参数用于环形，即中空的圆形或扇形，内切半径即为小圆半径。

方形面片

我们从简单的方形面片开始。定义几个变量后，在 updateMesh 中编写方形部分：

```
public class RangeShower : MonoBehaviour
{

    #region Private Attributes
    Mesh mesh = null;
    MeshFilter meshFilter = null;
    MeshRenderer meshRender = null;

    Vector3[] vertices;
    int[] triangles;
    Vector2[] uvs;

    int lastCount ;
    #endregion

    public void updateMesh(ShapeType shape, float degree,float radius,float
innerOff) {
        if (shape == ShapeType.obb)
        {
            if (mesh == null)
            {
                mesh = new Mesh();
            }
            meshFilter = GetComponent<MeshFilter>();
            meshRender = GetComponent<MeshRenderer>();

            Vector2 uvCenter = new Vector2(0.5f, 0.5f);
            mesh.Clear();
            vertices = new Vector3[4];
            triangles = new int[6];
            uvs = new Vector2[4];
            vertices[0] = Vector3.zero;
            uvs[0] = uvCenter;

            vertices[0] = new Vector3(degree, 0, 0); //1,0
            vertices[1] = new Vector3(degree, 0, radius); //1,1
            vertices[2] = new Vector3(-degree, 0, radius); //-1,-1
            vertices[3] = new Vector3(-degree, 0, 0); //-1,0

            triangles[0] = 3;
            triangles[1] = 2;
            triangles[2] = 0;
```

```
            triangles[3] = 0;
            triangles[4] = 2;
            triangles[5] = 1;

            uvs[0] = new Vector2(0, 1);
            uvs[1] = new Vector2(0, 0);
            uvs[2] = new Vector2(1, 0);
            uvs[3] = new Vector2(1, 1);

            mesh.vertices = vertices;
            mesh.triangles = triangles;
            mesh.uv = uvs;
            meshFilter.sharedMesh = mesh;
            meshFilter.sharedMesh.name = "CircularSectorMesh"; ;
            meshRender.sharedMaterial = m_RectIndicator;

            m_degree = degree;
            m_radius = radius;
        }
        //...
    }

}
```

可以看到，我们编写了顶点的位置、三角形的绘制方式、UV 的映射方式，然后把这个 Mesh 赋值给了 MeshFilter。

扇形区域

接着写扇形区域。扇形区域有分隔精细度的概念，比如，分隔精度为 36 度，则 360 度的圆，就可以分成 10 份。显而易见这个分隔角度越小，划分的区域就越细，圆形的边缘就越平滑。但面数过多，会对性能造成负担，因此这个值需要根据效果调整，这里使用 10 度作为默认值：

```
//...
public void updateMesh(ShapeType shape, float degree,float radius,float
innerOff)
{
    if (shape == ShapeType.sector)
    {
        if (mesh == null)
        {
```

```
        mesh = new Mesh();
    }
    meshFilter = GetComponent<MeshFilter>();
    meshRender = GetComponent<MeshRenderer>();

    int i;
    float beginDegree;
    float endDegree;
    float beginRadian;
    float endRadian;
    float uvRadius = 0.5f;
    Vector2 uvCenter = new Vector2(0.5f, 0.5f);
    float currentIntervalDegree = 0;
    float limitDegree;
    int count;

    float beginCos;
    float beginSin;
    float endCos;
    float endSin;

    int beginNumber;
    int endNumber;
    int triangleNumber;

    currentIntervalDegree = Mathf.Abs(m_intervalDegree);

    count = (int)(Mathf.Abs(degree) / currentIntervalDegree);
    if (degree % m_intervalDegree != 0)
    {
        ++count;
    }
    if (degree < 0)
    {
        currentIntervalDegree = -currentIntervalDegree;
    }

    if (lastCount != count || shape != m_type)
    {
        mesh.Clear();
        vertices = new Vector3[count * 2 + 1];
        triangles = new int[count * 3];
```

```
            uvs = new Vector2[count * 2 + 1];
            vertices[0] = Vector3.zero;
            uvs[0] = uvCenter;
            lastCount = count;
        }

        i = 0;

        beginDegree = 90 - degree / 2;
        limitDegree = degree + beginDegree;

        while (i < count)
        {
            endDegree = beginDegree + currentIntervalDegree;

            if (degree > 0)
            {
                if (endDegree > limitDegree)
                {
                    endDegree = limitDegree;
                }
            }
            else
            {
                if (endDegree < limitDegree)
                {
                    endDegree = limitDegree;
                }
            }

            beginRadian = Mathf.Deg2Rad * beginDegree;
            endRadian = Mathf.Deg2Rad * endDegree;

            beginCos = Mathf.Cos(beginRadian);
            beginSin = Mathf.Sin(beginRadian);
            endCos = Mathf.Cos(endRadian);
            endSin = Mathf.Sin(endRadian);

            beginNumber = i * 2 + 1;
            endNumber = i * 2 + 2;
            triangleNumber = i * 3;
```

```
        vertices[beginNumber].x = beginCos * radius;
        vertices[beginNumber].y = 0;
        vertices[beginNumber].z = beginSin * radius;
        vertices[endNumber].x = endCos * radius;
        vertices[endNumber].y = 0;
        vertices[endNumber].z = endSin * radius;

        triangles[triangleNumber] = 0;
        if (degree > 0)
        {
            triangles[triangleNumber + 1] = endNumber;
            triangles[triangleNumber + 2] = beginNumber;
        }
        else
        {
            triangles[triangleNumber + 1] = beginNumber;
            triangles[triangleNumber + 2] = endNumber;
        }

        if (radius > 0)
        {
            uvs[beginNumber].x = beginCos * uvRadius + uvCenter.x;
            uvs[beginNumber].y = beginSin * uvRadius + uvCenter.y;
            uvs[endNumber].x = endCos * uvRadius + uvCenter.x;
            uvs[endNumber].y = endSin * uvRadius + uvCenter.y;
        }
        else
        {
            uvs[beginNumber].x = -beginCos * uvRadius + uvCenter.x;
            uvs[beginNumber].y = -beginSin * uvRadius + uvCenter.y;
            uvs[endNumber].x = -endCos * uvRadius + uvCenter.x;
            uvs[endNumber].y = -endSin * uvRadius + uvCenter.y;
        }

        beginDegree += currentIntervalDegree;
        ++i;
    }

    mesh.vertices = vertices;
    mesh.triangles = triangles;
    mesh.uv = uvs;
```

```
        mesh.RecalculateNormals();
        mesh.RecalculateBounds();

        meshFilter.sharedMesh = mesh;
        meshFilter.sharedMesh.name = "CircularSectorMesh";

        m_degree = degree;
        m_radius = radius;

        meshRender.sharedMaterial = m_circleIndicator;
    }
    //...
}
```

上面代码的核心依然是根据角度计算顶点的位置，按顺序围成三角形，需要注意顶点的顺序，编程时可以在纸上演练一下。环形与扇形类似，只是在近点处计算了偏移，这里就不做代码示例了。

测试

新建两个材质，分别用于方形和圆形的区域，这里选择“LegacyShader/Transparent/Diffuse”。这里找到两种贴图放到材质上，以避免效果太单调。在场景中创建几个对象，设置参数，即可看到如图 10.4 所示效果。

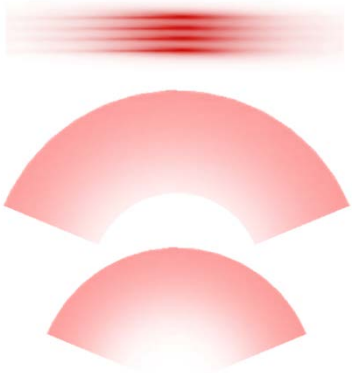


图 10.4

总结

通过改变模型面片形状，我们可以做很多定制化的效果。上面展示了如何通过更改顶点位置来更改区域的样式，我们也可以自己编写一些面片生成规则，制作出特殊的特效。

扩展问题

（1）由于目前方形与圆形混用在一起，所以参数和材质有些混乱，请问有办法清晰化吗（[:rangedrawinfo]）？

（2）如何实现环形绘制功能（[:rangedraw]）。

第 11 章
日志工具

在笔者看来，日志是程序员最后的防线。如果一个问题通过增加日志无法解决，那么其他手段更是几乎不可能。因此，最后的防线是否稳固，直接关系到我们解决问题的效率。

在本章中，我们将看到日志相关的常见问题，并探讨更好的使用方法。

11.1　出错暂停

在 Unity 3D 的编辑器的 Console 中，单击“Error Pause”按钮，可以在报错时暂停游戏，如图 11.1 所示。

图 11.1

这个功能的方便之处在于：每当遇到报错时，暂停游戏可极大地方便开发人员分析产生问题的原因。

现在这个功能被项目组中的测试人员知道了，他们希望在手机上也实现类似的效果。他们希望在程序报错时将游戏停下来，以方便截图提 Bug。请给出这个需求的解决方案。

问题分析

在手机上暂停并显示错误有以下优点

（1）它可以通过侦测到报错的出现，来防止无用的测试。Unity 3D 程序报错不一定会导致游戏崩溃，但可能会引起其他功能的异常。因此一旦出现报错，后续的测试就没有意义了，应及时停止测试，避免在错误的基础上浪费时间。

（2）它可以在停止游戏的时候将错误日志显示在屏幕上，这就相当于找到了与报错日志相吻合的现场，可以方便后续的 Bug 定位。

基础环境搭建

为了测试功能，需要搭建测试环境。大致流程是，在场景中创建一个按钮和一个文本区域。左侧是交互按钮，右侧是显示文字的文本区域。接着创建代码 LogListener.cs：

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class LogListener : MonoBehaviour
{
    public Text logText;
    public Button btn;

    void Start ()
    {
        btn.onClick.AddListener(() =>
        {
            Debug.Log("TestLog");
            Debug.LogError("Test Error");
        });
    }
}
```

将按钮和文字的对应节点拖到脚本节点上，效果如图 11.2 所示。

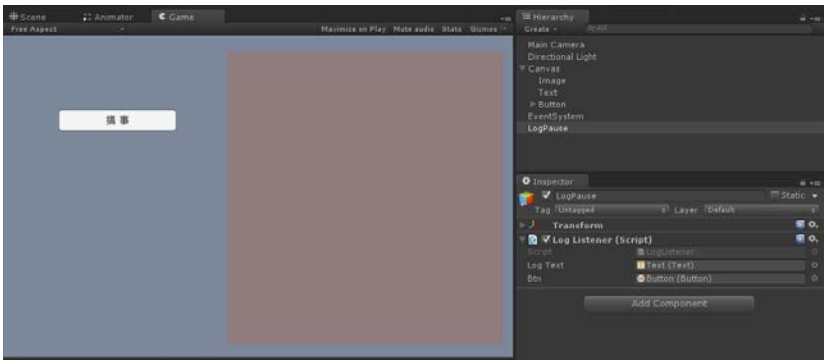


图 11.2

监听错误日志

对于输出日志的监听，可以使用为 Application.logMessageReceived 添加回调函数。更改代码如下：

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class LogListener : MonoBehaviour
{
    public Text logText;
    public Button btn;

    void Start ()
    {
        btn.onClick.AddListener(() =>
        {
            Debug.Log("TestLog");
            Debug.LogError("Test Error");
        });

        Application.logMessageReceived += logMessageReceivedFunc;
    }
    void OnDestroy()
    {
        Application.logMessageReceived -= logMessageReceivedFunc;
```

```
    }

    public void logMessageReceivedFunc(string logValue, string stackTrace,
LogType type)
    {
        if (type == LogType.Error || type == LogType.Exception)
        {
            logText.text +="<color=red>"+ logValue + "\n" +
stackTrace+"</color>";
        }
        else
        {
            logText.text +=  logValue + "\n" + stackTrace;
        }
    }
}
```

我们添加了 logMessageReceivedFunc 回调函数，在其中将截获的错误日志写到文本区域中。如果是报错日志就用红色标记出来。单击“搞事”按钮后，可以看到日志输出到文本区域，效果如图 11.3 所示。

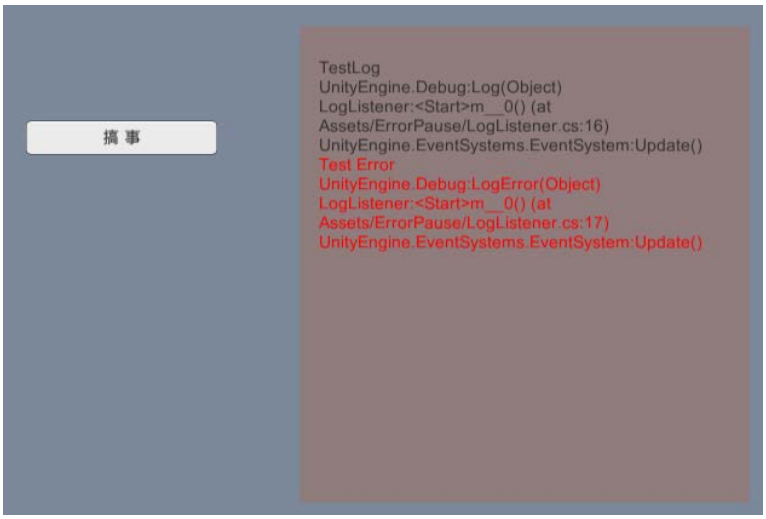


图 11.3

暂停游戏

暂停游戏的核心是将游戏循环停掉。最简单的做法是将 TimeScale 调整为 0，这样可以停止几乎所有的动画与特效，并且可以停下 FixedUpdate 的调用。通常来说，这种级别的暂停已经足够了。当然如果项目规划了游戏主循环，那么最好加入对应的暂停逻辑。这里更改函数代码如下：

```
public void logMessageReceivedFunc(string logValue, string stackTrace, LogType
type)
{
    if (type == LogType.Error || type == LogType.Exception)
    {
        Time.timeScale = 0f; //Pause the game
        logText.text +="<color=red>"+ logValue + "\n" + stackTrace+"</color>";
    }
    else
    {
        logText.text +=  logValue + "\n" + stackTrace;
    }
}
```

总结

为了实际测试功能，笔者在按钮单击时，故意写错一行代码：

```
void Start ()
{
    btn.onClick.AddListener(() =>
    {
        List<int> a = new List<int>();
        a[0].ToString();
    });

    Application.logMessageReceived += logMessageReceivedFunc;
}
```

运行后，单击按钮结果如图 11.4 所示。

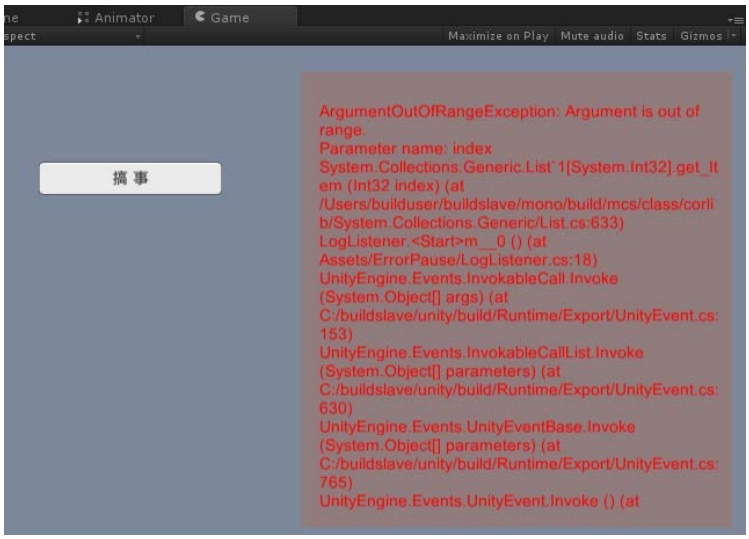


图 11.4

实现了这个功能后，测试极大地降低了哑巴吃黄连的窘境：即使很难复现，也可以根据截图理直气壮地提 Bug。另一方面，有了报错现场，程序员可以更高效地修复问题。

扩展问题

如果出错会引起程序闪退，这样做依然有效吗？有办法进行优化吗？

11.2　日志接口优化

使用 Unity 3D 的日志输出会影响游戏性能，通常会产生大量的 GC 并占用 CPU 资源。如果不小心可能会出现如图 11.5 所示的情况。

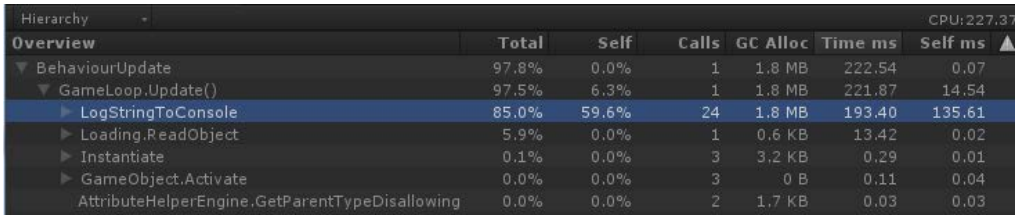


图 11.5

现在希望在开发过程中依然保留日志，在打包到手机上时，将大部分日志过滤掉，以达到

性能优化的效果。请设计相关的解决方法。

问题分析

做过 Unity 3D 性能优化的人都知道，在程序运行过程中，日志的输出量会对游戏的效率产生很大的影响。具体原因在于，输出日志时需要占用 CPU 来获取堆栈信息，然后拼接这些信息时，还会产生内存垃圾，并触发程序的垃圾回收。因此总的来说，最好的解决办法就是不调用日志输出。

另一方面，如果只是为了关闭输出，那么可以更改 Unity 3D 内置的日志输出参数：

```
Debug.logger.logEnabled = false;
```

这样就可以将所有日志都关闭了。但一般来说，我们还是有需要输出的信息，所以这种做法并不可取。

接口封装

Unity 3D 日志输出功能，是依照 String.Format 拼接规则的接口：

```
public static void LogFormat(string format, params object[] args);
public static void LogFormat(Object context, string format, params object[]
args);
```

但通常大家都不这样使用，而直接是使用加号进行字符串拼接。这样做会产生额外的内存分配，并且无法在封装的函数内优化掉。因此更好的做法是，在 Log 接口中直接支持 format 形式的可变参数，通过分析参数来选择输出形式。

创建一个名为 LogManager 的类，输入如下代码：

```
using UnityEngine;
using System.Collections;

public class LogManager
{
    public static bool Enable = true;
    public static void Log(string format, params object[] args)
    {
        if (Enable)
        {
```

```
            if (format.Length == 0)
            {
                return;
            }

            if (args.Length == 0)
            {
                Debug.Log(format);
            }
            else
            {
                Debug.LogFormat(format, args);
            }
        }
    }
}
```

为了测试这个类，我们创建一个 LogTest，并将其挂载到场景对象中，代码如下：

```
using UnityEngine;
using System.Collections;

public class LogTest : MonoBehaviour
{
    void Start()
    {
        //LogManager.Enable = false;
        LogManager.Log("测试 {0} 的 Log", "动态参数");
    }
}
```

运行游戏即可看到日志输出，如果将注释去掉则可以关闭日志。

DLL 封装

功能虽然封装完成，但有个小缺陷：双击日志后不能跳转到 Start 函数中，而是跳到了 LogManager 的调用位置。这会增加我们的调试难度，因此必须解决这个问题。为了修复它，我们需要将 LogManager 类打包成 DLL。

对于 Windows 平台，首先要找到编译器。因为是 Unity 3D 使用的是 C#2.0 版本，所以需要找到 2.0 版本的目录：

```
C:\Windows\Microsoft.NET\Framework\v2.xxxxx
```

如果未能找到，则可以参考微软的官方文档（[:csc]）。从命令行中进入对应目录，可输入以下命令：

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727>csc
-r:C:\YourProjectDir\Library\UnityAssemblies\UnityEngine.dll -target:library
C:\YourProjectDir\Assets\LogInt
erface\LogManager.cs
```

这样就会在同级目录中生成一个名为 LogManager 的 DLL 了，把它移到项目中并将 LogManager.cs 删除，再次运行项目，当出现日志时双击日志即可定位到调用点。

更改标准日志

现在有了新版的日志接口，但可能依然会有人使用 Debug.Log 作为日志输出。另外，如果有很多历史代码，整体替换可能不是最优的选择。这里我们需要创建一个 ILoggerHandle 的子类，将 Debug.Log 的默认输出替换成可控的函数。在 LogManager.cs 中添加 LogReplace 类，并且在 LogManager 类中创建一个静态成员来触发构造函数，代码如下：

```
public class LogReplace : ILogHandler
{
    private ILogHandler m_DefaultLogHandler = Debug.logger.logHandler;

    public LogReplace()
    {
        //替换默认的 logHandler
        Debug.logger.logHandler = this;
    }

    //重写日志输出
   public void LogFormat(LogType logType, UnityEngine.Object context, string
                         format, params object[] args)
    {
        if (LogManager.Enable)
        {
            m_DefaultLogHandler.LogFormat(logType, context, format, args);
        }
    }
```

```
    public void LogException (Exception exception, UnityEngine.Object context)
    {
        m_DefaultLogHandler.LogException (exception, context);
    }
}

public class LogManager
{
    private static LogReplace replace = new LogReplace();
    //......
}
```

这样在运行游戏时，如果将日志开关关闭，那么 Debug.Log 也无法输出任何信息。这个功能对于会产生日志的一些第三方插件会很有用。

总结

采用上文的封装可以通过 Enable 开关动态控制日志的输出，最后，建议过滤日志时网开一面，例如，通过类型判断放行 Debug.LogError，毕竟有时候可能还是需要输出一些信息。

扩展问题

对于一定不会输出的日志有办法节省它的运行效率吗（[:debuglogcull]）？

11.3　频道化日志

经过开关封装后，日志模块已经告别了性能的问题。没有了这个忧虑，所有的开发人员都很开心，大家都开心地写了很多的日志。但很快，大家就发现自己的日志被淹没在海量的输出中，查找起来非常困难。请设计一套方法控制日志的输出。

问题分析

随着项目规模的扩大，日志被淹没是常见现象。因此有很多开发者会把日志写到一些“干净”的地方。比如使用 Debug.LogError 将日志写到错误输出窗口中。但当有复杂的情况出现时，备用窗口就不够用了。

这个问题其实和我们使用 QQ 聊天十分类似。聊天时，各种群信息应该选择性可见，单聊

的小窗信息也应该选择性可见，小窗信息和群信息不应混合在一起。基于这样的考虑，笔者认为日志频道最好分为两个维度：

◎ 公共频道
◎ 个人频道

每个维度的信息单独显示，并分别设定不同的信息过滤规则。为了不影响 Unity 3D 的 ErrorPause 功能，可以将个人频道信息放置在 warning 窗口中，这对于消除运行时的 warning 也可以间接地起到推进作用。

频道算法

频道开关的判断规则，推荐采用 bitmask 来实现。一方面，便于表示多个逻辑兼容的情况；另一方面，只用一个数字方便数据逻辑的迁移。创建一个名为 BitMask.cs 的文件，编写如下代码：

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

[Flags]
public enum MaskDef
{
    Fsy         = 1<<0, //User Channel
    SL          = 1<<1,
    DJ          = 1<<2,
    JC          = 1<<3,
    //...
    Module      = 1<<6, //Group Channel
    NetWork     = 1<<7,
    Battle      = 1<<8,
    EventLine   = 1<<9,
    AllPlatform = 1<<10,
    Process     = 1<<11,
    Alert       = 1<<12,
}


public class BitMask
{
```

```
    public static bool IsEnabled(MaskDef flags, MaskDef flag)
    {
        return (flags & flag) != 0;
    }

    public static void Enable(ref MaskDef flags, MaskDef flag)
    {
        flags = (flags | flag);
    }

    public static void Disable(ref MaskDef flags, MaskDef flag)
    {
        flags = (flags & (~flag));
    }
}
```

编辑好文件后，再创建一个测试脚本来验证。创建 MaskTest.cs，将其添加到场景对象上。代码内容如下：

```
using UnityEngine;
using System.Collections;

public class MaskTest : MonoBehaviour
{
    void Start()
    {
        LogMaskDef mask = LogMaskDef.Fsy|LogMaskDef.Process;
        Debug.LogFormat("Mask Process is: {0}",  BitMask.IsEnabled(mask,
LogMaskDef.Process));
        Debug.LogFormat("Mask Module is: {0}", BitMask.IsEnabled(mask,
LogMaskDef.Module));
        BitMask.Enable(ref mask, LogMaskDef.Module);
        Debug.LogFormat("Enable Mask Set: {0}", mask);
    }
}
```

运行游戏，输出如图 11.6 所示。

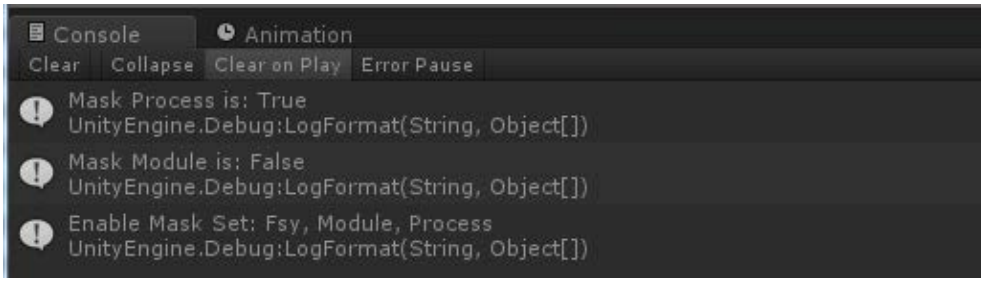


图 11.6

融入模块

基础逻辑写好后，就可以将它作为功能添加到日志控制中了。基于之前日志接口优化中的 LogManager，我们需要将 BitMask.cs 的逻辑一并编入 DLL 中。由于代码不多，推荐直接将其复制到 LogManager.cs 中。另一方面，LogManager 类需要扩展 Mask 变量，替换掉单纯的开关输出的逻辑，并根据不同的频道设计接口。因此 LogManager 类应更改为下面的代码：

```
public class LogManager
{
    public static MaskDef Mask = MaskDef.Fsy|MaskDef.Process;
    private static LogReplace replace = new LogReplace();

    public static bool IsEnabled(MaskDef flag)
    {
        return BitMask.IsEnabled(Mask, flag);
    }

    protected static void StandardLog(string format, params object[] args)
    {
        if (format.Length == 0)
        {
            return;
        }

        if (args.Length == 0)
        {
            Debug.Log(format);
        }
        else
        {
```

```
            Debug.LogFormat(format, args);
        }
    }

    //For Old Interface
    public static void Log(string format, params object[] args)
    {
        LogManager.Process(format, args);
    }

    public static void Process(string format, params object[] args)
    {
        if (IsEnabled(MaskDef.Process))
        {
            StandardLog(format, args);
        }
    }

    public static void NetWork(string format, params object[] args)
    {
        if (IsEnabled(MaskDef.NetWork))
        {
            StandardLog(format, args);
        }
    }
    //......
}
```

更改测试类为：

```
public class MaskTest : MonoBehaviour
{
    void Start()
    {
        //...
        Debug.Log(LogManager.Mask);
        LogManager.Process("Process is working");
        LogManager.NetWork("NetWork is working");
    }
}
```

运行游戏可见输出如图 11.7 所示。

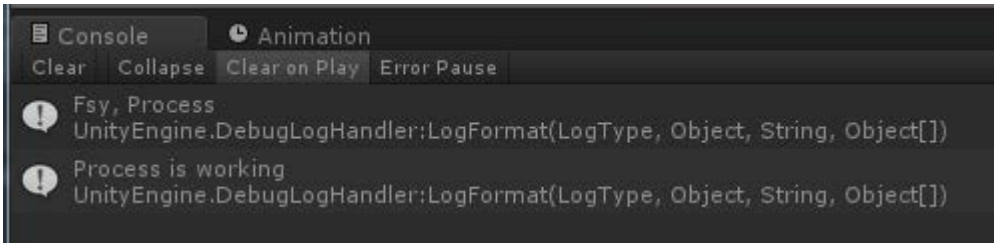


图 11.7

在这个例子中，我们将 LogManager.Mask 设置为 Fsy 和 Process，可以发现 Process 级别的日志可以正常输出，但 NetWork 级别的日志就不会显示。

总结

通过频道控制日志可以灵活地控制日志的输出，不必担心在海量的日志中抓狂。另外，也可以根据不同的情景制定不同的策略，以应对不同的情况。

扩展问题

有的开发人员不喜欢默认的日志输出样式，希望自定义显示格式，请问如何优化日志输出的架构呢？

11.4 崩溃日志上报

游戏在手机上崩溃了，真是个令人无奈的事情。由于整个函数堆栈挂掉了，程序员无法借助之前做好的出错暂停来追查问题。偏巧测试人员只能在某台设备上复现问题，而现在必须定位并解决问题。在事情变得更棘手之前，有什么好的方案来处理这种情况吗？

问题分析

回想探案类影视剧中，死者如何告诉侦探谁是真凶的？最常见的做法就是死亡信息（Dying Message）。死者在死前留下的有助于案件解决的信息，往往对案件的侦破起着巨大的指向性的作用。这里也可以采用相同的策略，在程序崩溃之前，将“致死”的信息留下来。由于 Unity 3D 在程序的最外层有异常捕捉函数，因此崩溃时也可以捕获到错误日志。我们可以将出错的日志记录到文件中，当程序下次启动时，从文件中读出错误日志，将它发送给收集错误日志的“侦探”。日志收集者可以是客户端程序，也可以是专门的日志服务器。

搭建环境

之前在出错暂停一篇中编写过如何监测报错日志，本节的逻辑与之类似，可以把之前的场景复制一份出来。创建新的脚本 Upload.cs，编写如下代码：

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

public class Upload : MonoBehaviour
{
    public Text logText;
    public Button btn;

    void Start()
    {
        btn.onClick.AddListener(() =>
        {
            List<int> a = new List<int>();
            a[0].ToString();
        });
        Application.logMessageReceived += logMessageReceivedFunc;
    }

    void OnDestroy()
    {
        Application.logMessageReceived -= logMessageReceivedFunc;
    }

    public void logMessageReceivedFunc(string logValue, string stackTrace,
                                       LogType type)
    {
        if (type == LogType.Error || type == LogType.Exception)
        {
            logText.text += "<color=red>" + logValue + "\n" + stackTrace + "</color>";
        }
    }
}
```

运行游戏，单击“搞事”按钮，如果在文本区域输出错误，则环境配置正确。

读写文件

环境配置好之后，接下来实现读写文件。首先需要将错误写到文件中，然后在游戏运行时将内容读出来。这里借用 Unity 3D 的本地存储数据接口 PlayerPrefs。相关代码如下：

```
private readonly string ErrorTag = "LogErrorTag";

public void logMessageReceivedFunc(string logValue, string stackTrace, LogType
type)
{
    //...
    PlayerPrefs.SetString(ErrorTag, text);
}

void Start()
{
    logText.text = PlayerPrefs.GetString(ErrorTag);
    if (PlayerPrefs.HasKey(ErrorTag))
    {
        PlayerPrefs.DeleteKey(ErrorTag);
    }

    //...
}
```

当这样实现后，每当游戏运行时，就会检测本地是否存有报错，如果有就将其读取出来并显示到界面的对应位置上。

总结

在实际使用时，错误日志通常会在游戏启动时上传给服务器，这样就可以在产品上线后获得大量的反馈，从而推进产品质量的提升。

扩展问题

通过崩溃日志上报可以获取崩溃堆栈，但很多情况下，没法复现问题，查起来没什么头绪，能想办法优化上报功能吗（[:erroruploadext]）？

第 12 章
快捷功能

对于很多日常性的操作，我们要动脑去简化，这才是一个程序员该做的事情。

本章我们将讨论简化操作的实现方式，为大家定制自己的策略提供思路。

12.1　自定义菜单

最近项目中加入了不少功能，但功能太多总是找不到启动入口。经过商量，大家决定把这些功能都加到菜单中，请问 Unity 3D 中的菜单一共有多少种呢？它们都可以添加自定义菜单项吗？如果想在其中加入内容，该如何做？

问题分析

在 Unity 中常用的菜单区域有以下几个：

◎　顶部下拉菜单
◎　Hierarchy 面板
◎　Project 面板

如果可以选择，我们通常都更倾向于使用后两个。

菜单创建

使用 Unity 3D 的内置属性 MenuItem 可以创建一个菜单。不同的菜单区域，可通过属性的参数区分。例如下面的代码，可用于创建顶部菜单：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class Menu : MonoBehaviour
{
    [MenuItem("MyMenu/Do Something")]
    static void DoSomething()
    {
        Debug.Log("Doing Something...");
    }
}
```

参数是个分级路径，可以定义出菜单的父子关系，效果如图 12.1 所示。

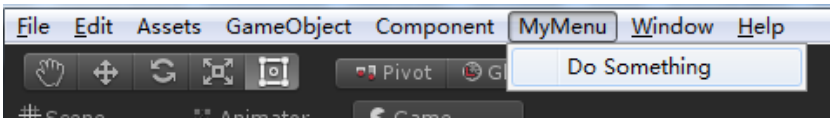


图 12.1

如果希望加到 Hierarchy 面板的右键菜单中，则可以使用路径 GameObject/开头。如果希望加到 Project 面板，则使用 Assets/开头。代码如下：

```
[MenuItem("GameObject/HierarchyMenuItem",false,10)]
static void HierarchyMenu()
{
    Debug.Log("Hierarchy MenuItem Click");
}

[MenuItem("Assets/ProjectMenuItem")]
static void ProjectMenu()
{
    Debug.Log("Project MenuItem Click");
}
```

编辑完成后，在不同区域右击，效果如图 12.2 和图 12.3 所示。

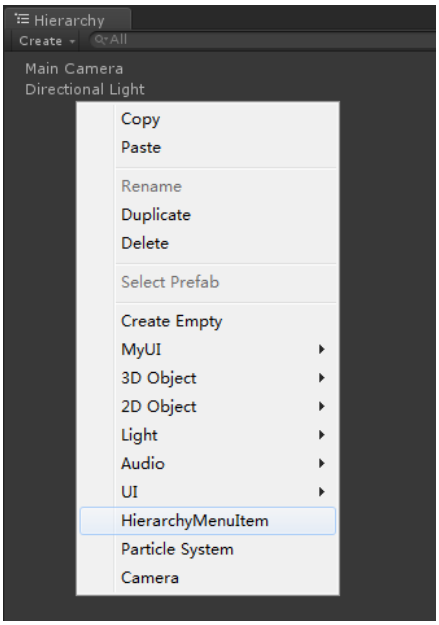


图 12.2

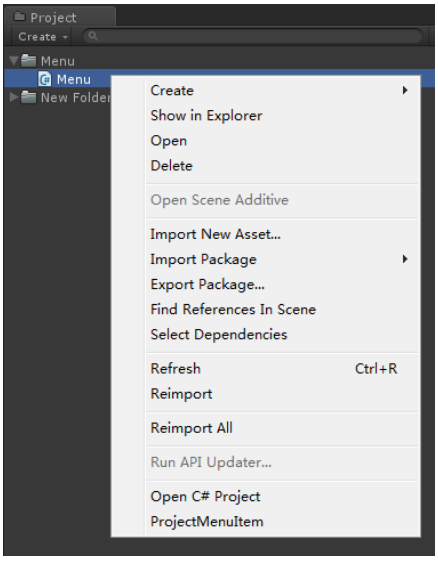


图 12.3

其中 MenuItem 的第三个参数是菜单显示位置的优先级。第二个用来标记菜单项的可用判断函数。例如，若加入下面的代码，则在 Project 区域有选中对象时，ProjectMenuItem 才可点击：

```
 [MenuItem("Assets/ProjectMenuItem", true)]
static bool ProjectMenu2()
{
    return Selection.objects.Length>0;
}
```

编辑完成后，效果如图 12.4 所示。

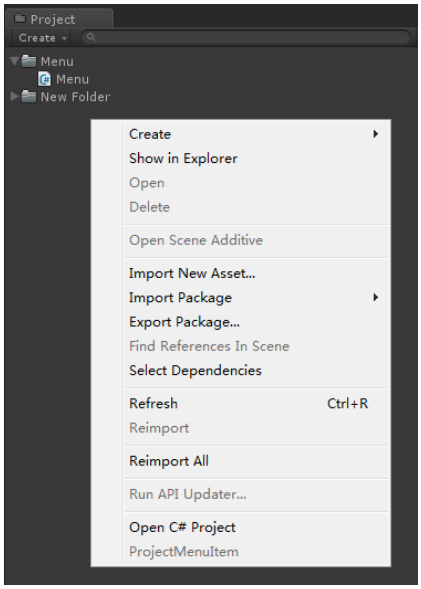


图 12.4

快捷键

MenuItem 不但可以制作菜单，还可以制作热键功能，例如下面的代码：

```
[MenuItem("HotKey/ShowLog %1")]
static void DoSomething()
{
    Debug.Log("Ctrl + 1 is Pressed!");
}
```

当按下快捷键 Ctrl＋1 时，这个函数就会被触发。%是 Ctrl 的对应关键字。热键的关键字有很多，常用的如表 12.1 所示。

热键标记	对应键
%	Ctrl（Windows）／cmd（Mac）
#	Shift
&	Alt
LEFT	左箭头
RIGHT	右箭头
UP	上箭头
DOWN	下箭头
F1~F12	F1~F12

热键也可以多个组合，例如，shift＋alt＋6，可以表示为#&6。

菜单标记

菜单有选中标记，我们可以利用它做状态的更改操作。例如，在频道化日志一篇中提到的日志掩码开关，完全可以使用菜单来进行设置。在 Unity 3D 中，使用 Menu.SetChecked 可以更改选中状态。下面以实现掩码功能为例，介绍菜单标记的用法。创建一个名为 MenuMask.cs 的脚本，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MenuMask : MonoBehaviour
{
    private const string channelFsy = "MaskMenu/Channel/Fsy";
    private const string channelProcess = "MaskMenu/Channel/Process";
    private const string channelNetWork = "MaskMenu/Channel/NetWork";

    private static void setFlag(string path, MaskDef channel)
    {
        bool flag = UnityEditor.Menu.GetChecked(path);
        if (flag)
        {
            BitMask.Disable(ref LogManager.Mask, channel);
            UnityEditor.Menu.SetChecked(path, false);
        }
        else
        {
```

```
            BitMask.Enable(ref LogManager.Mask, channel);
            UnityEditor.Menu.SetChecked(path, true);
        }
    }

    [MenuItem(channelFsy)]
    static void ChannelFsy()
    {
        setFlag(channelFsy, MaskDef.Fsy);
    }

    [MenuItem(channelFsy, true)]
    static bool ChannelFsyCheck()
    {
        UnityEditor.Menu.SetChecked(channelFsy,
LogManager.IsEnabled(MaskDef.Fsy));
        return true;
    }

    [MenuItem(channelProcess)]
    static void ChannelProcess()
    {
        setFlag(channelProcess, MaskDef.Process);
    }

    [MenuItem(channelProcess, true)]
    static bool ChannelProcessCheck()
    {
        UnityEditor.Menu.SetChecked(channelProcess,
LogManager.IsEnabled(MaskDef.Process));
        return true;
    }

    //......
}
```

编辑完成后，效果如图 12.5 所示。

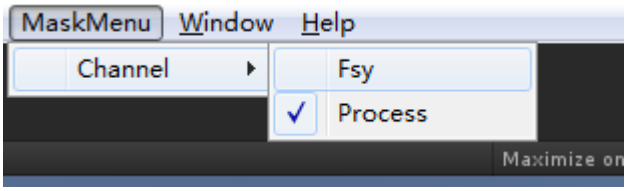


图 12.5

总结

有了多种使用菜单的方法，记得好好规划一下，否则菜单项会杂乱无章，快捷键也很难记清。制定热键规则时需要注意冲突，例如，如果有 Shift 和数字键的组合，不要使用常用的特殊字符，否则会造成无法输入对应字符的现象。

扩展问题

在什么情况下使用顶部菜单项比较好（[:menutop]）？

12.2　定制 UI

在制作 UI 的过程中，大多数资源是相同的，例如：

◎　确定按钮会使用相同的图片。
◎　文字会使用相同的字体。

针对这些情况，有办法简化开发 UI 的流程吗？

问题分析

在开发过程中，如果能简化项目中的 UI 制作流程，则能节省大量的时间。但很多开发者只从程序运行的角度架构化代码，而忽略了从制作流程上节省开发的精力。这个问题的重点在于设计出一套简单可行的流程，这需要有对编辑器开发的相关经验。

创建工具类

首先要确定创建的入口，这方面最好的策略是与日常操作相似，所以这里将其添加在 Hierarchy 的右键菜单中。新建一个名为 CreateTool 的脚本，编写如下代码：

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

public class CreateTool : MonoBehaviour
{
    [MenuItem("GameObject/MyUI/Button",false,0)]
    public static void MyBtn()
    {
        Debug.Log("I Will Create Btn");
    }

}
```

这步编辑器操作使用了一个小技巧，在创建菜单项时，如果使用 GameObject/UI 开头，就会将其创建到 UI 的子菜单中去；如果以 Assets/开头；就会创建到 Project 右键快捷菜单中。

编写完成后，在 Hierarchy 中右击，调出右键快捷菜单即可看到对应的选项，效果如图 12.6 所示。

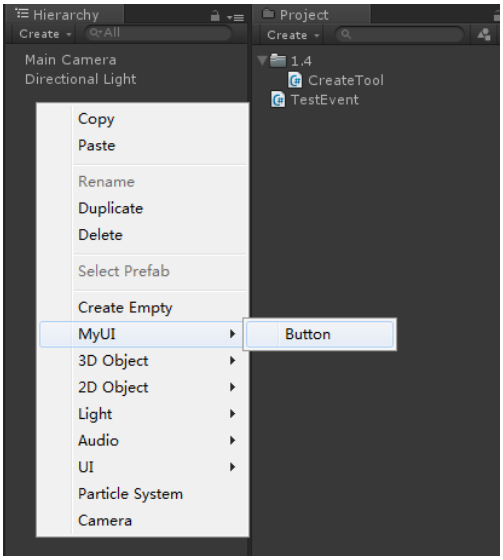


图 12.6

单击 Button 后会输出日志，证明一切功能正常运行。

创建标准按钮

菜单入口创建好后，接下来实现功能。大体思路是先创建一个标准按钮，然后换掉图 12.6 中的图片。创建按钮可以参考下面的代码：

```
 [MenuItem("GameObject/MyUI/Button",false,0)]
public static void MyBtn()
{
    GameObject buttonRoot = new GameObject("Button");

    GameObject childText = new GameObject("Text");
    childText.transform.SetParent(buttonRoot.transform,false);

    Image image = buttonRoot.AddComponent<Image>();

    Button bt = buttonRoot.AddComponent<Button>();

    Text text = childText.AddComponent<Text>();
    text.text = "Button";
    text.alignment = TextAnchor.MiddleCenter;
    text.color = new Color(0, 0, 0);
    text.transform.SetParent(buttonRoot.transform,text);
    text.raycastTarget = false;

    RectTransform textRectTransform = childText.GetComponent<RectTransform>();
    textRectTransform.anchorMin = Vector2.zero;
    textRectTransform.anchorMax = Vector2.one;
    textRectTransform.sizeDelta = Vector2.zero;
}
```

再次单击右键快捷菜单中的 Button 后就会在 Hierarchy 中创建出一个按钮了。需要注意的是，由于它不在 Canvas 中，所以需要手动将其拖到合适的位置才可以在 Game 视图中预览。

替换资源

按钮创建好后，就可以进行真正的替换工作了。这步操作会用到一个名为 `AssetDatabase` 的类，它是编辑器对文件操作的核心，几乎所有对资源文件的 CRUD 都有它的参与。CRUD 是在做计算处理时增加（Create）、读取查询（Retrieve）、更新（Update）和删除（Delete）几个

单词的首字母简写。相关的使用方法在官方文档上有详细介绍（[:assetdatabasedoc]）。接下来依照上面所说，一步步将按钮的底图换掉。

首先，准备好资源图，将其放置到项目中，并更改图片类型为 Sprite，接着编写如下代码：

```
[MenuItem("GameObject/MyUI/Button",false,0)]
    public static void MyBtn()
    {
        //......
        Image image = buttonRoot.AddComponent<Image>();
        image.sprite = AssetDatabase.LoadAssetAtPath<Sprite>
                      ("Assets/1.4/BtnTexture.png");
        image.SetNativeSize();

        Button bt = buttonRoot.AddComponent<Button>();
        //......
    }
```

编辑完成后单击菜单中的 MyButton 选项，即可直接创建出一个自定义背景的按钮，效果如图 12.7 所示。

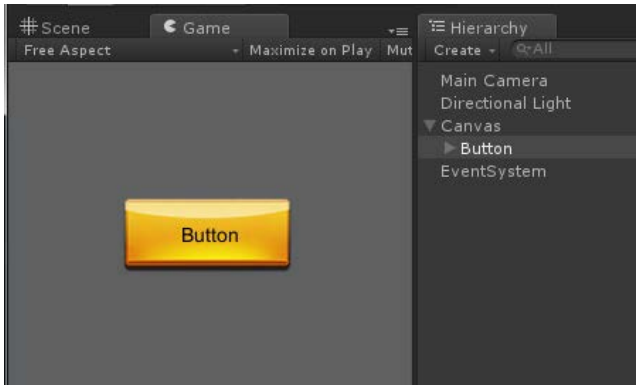


图 12.7

优化体验

按钮背景的替换就这样完成了，但在操作的过程中，我们发现整个流程有个很麻烦的地方：创建出的按钮需要手动拖动并更改位置。不合乎操作习惯就要优化，所以这里将其添加到选中对象的子节点下：

```
[MenuItem("GameObject/MyUI/Button", false, 0)]
public static void MyBtn()
{

    //......
    RectTransform textRectTransform = childText.GetComponent<RectTransform>();
    textRectTransform.anchorMin = Vector2.zero;
    textRectTransform.anchorMax = Vector2.one;
    textRectTransform.sizeDelta = Vector2.zero;

    if (Selection.gameObjects.Length > 0)
    {
        buttonRoot.transform.SetParent(Selection.gameObjects[0].transform,
false);
    }

    // Make Button Selected
    Selection.objects = new Object[] { buttonRoot };
}
```

在函数的末尾添加上面的代码，就可以在选中目标下创建子按钮了。另外这里还增加了一个更改选中焦点的功能，一般来说，添加了按钮之后需要对它进行编辑，自动选中可以减少一步单击的操作。

总结

使用同样的思路可以创建使用指定字体的文字、指定资源的进度条等。工作流程的优化会节省大量的编辑时间，这其中的技巧值得花时间研究。

扩展问题

（1）现在图片资源的位置是固定的，很不灵活，有没有办法优化呢？

（2）采用下拉菜单项创建有时找起来很麻烦，有办法变得更方便吗？[:forquick]

12.3　回退操作

在 Unity 3D 编辑器扩展时，经常有影响很大的属性操作，比如更改坐标位置，或增删

GameObject。当误操作时，常规的回退是 Undo 操作，即 Cmd/Ctrl + z。那么如何让一个编辑器扩展功能支持回退操作呢？

现在有一批预设中有很多无用且 Disable 的 GameObject，但有些 Disable 的节点不能删除。这些节点没有规律，因此需要尝试性地将子节点中所有 Disable 的对象删掉，如果删错，就需要有回退操作。

这种情况下该如何完成任务呢？

问题分析

编辑器实现功能时，通常会忘记编写回退操作。大多数情况下，添加操作不会有太多问题，但更改和删除有时就需要回退功能以防止误操作了。Unity 中的 Undo 类专门用来回退各种操作。下面分别以删除对象和更改 Transform 属性为例，介绍如何使用 Undo 回退操作。

删除对象

Undo 类中的 DestroyObjectImmediate 函数可以将删除的对象“记住”，如果使用了回退操作，那么它是可以复原出删掉的对象的。创建 UndoOperation.cs，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class UndoOperation : MonoBehaviour
{
    [MenuItem("MyMenu/DelDisableObject")]
    static void DelObject()
    {
        var objs = Selection.gameObjects;
        foreach (var go in objs)
        {
            for (int i = 0; i < go.transform.childCount; ++i)
            {
                var child = go.transform.GetChild(i).gameObject;
                if (child.activeInHierarchy == false)
                {
                    Undo.DestroyObjectImmediate(child);
                }
```

```
            }
        }
    }
}
```

编写完成后，即可将隐藏的直接子节点删除，使用 Undo 操作则可将删除的对象重新创建出来。

更改 Transform 属性

更改属性也是常见操作。例如，将一组对象的坐标按照第一个对象的 y 值对齐。这时就需要更改坐标值，如果误选了其他元素，则需要回退操作，实现这个功能可借助 RecordObject 函数，在刚刚的文件中加入下面的代码：

```
 [MenuItem("MyMenu/TransformChange")]
static void TransformChange()
{
    var objs = Selection.gameObjects;
    float lineY = 0;
    if (objs.Length > 0)
    {
        lineY = objs[0].transform.position.y;
    }

    foreach (var go in objs)
    {
        var trans = go.transform;
        Undo.RecordObject(trans, "Change Transform");
        trans.position = new Vector3(trans.position.x, lineY, trans.position.z);
    }
}
```

编写完成后，选中一组对象即可将它们水平对齐，如果使用 Undo 操作，则可以将位置退回到之前的状态。需要注意的是，RecordObject 只能检测当前对象自己的属性变化，因此不要指望只传入一个 GameObject，就可以回退所有挂载组件上的属性。

总结

通过两个实例介绍了如何使编辑器扩展支持回退操作。在实际工作中，这项技巧几乎是不

可或缺的，也希望大家能仔细研究一下官方文档中的其他接口[:undodoc]。

扩展问题

是否应该在编辑器代码更改值的附近都加入 Undo 支持（[:undo]）？

第 13 章
后台服务

在 Unity 3D 中，有很多自动运行的服务。本章我们将搭建一个自己的服务，用于在编辑器状态变化时产生回调。本章也会讨论文件遍历的方式。在资源管理方面，这种遍历通常会结合后台服务，完成资源扫描更新的任务。

13.1　编辑器服务

最近项目进入到比较忙的阶段，组里的人经常忙到忘记订饭，所以现在准备开发个 Unity 编辑器小功能，在每天固定的时间弹出一个订饭的提示窗口，提醒大家去订饭。

如果不打开项目，岂不是看不到提醒了？

不干活的人根本不用管嘛。

问题分析

做编辑器插件时，如果能借助自定义服务，就可以做很多监测型的操作。除了订饭，当然还有更多用处，确实比较实用。另外，如果能获取到编辑器的状态改变，也可以做些特殊的操作。比如我希望在游戏退出运行模式之前，把一些编辑的东西缓存出来，然后对这些数据做自动化处理，那么我就需要退出运行模式的事件。

自动运行

使用属性 InitializeOnLoad，可以在脚本编译过后自动调用类的构造函数。利用这点，可以制作一个自动启动的服务。创建文件 AutoServer.cs，编写如下代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[InitializeOnLoad]
public class AutoServer : MonoBehaviour
{
    static AutoServer()
    {
        EditorApplication.update += ScriptLoaded;
    }

    static void ScriptLoaded()
    {
        EditorApplication.update -= ScriptLoaded;
        EditorApplication.update += IdleUpdate;
    }

    static void IdleUpdate()
    {
       //do sth
    }
}
```

编辑完成后，IdelUpdate 中的代码逻辑会在编辑状态的每一帧执行。

时间检测

有了自动的服务，一切都变得简单了，创建一个名为 CheckAndShow 的函数，并在 IdleUpdate 中调用它，修改代码如下：

```
static void IdleUpdate()
{
       CheckAndShow();
    if (EditorApplication.isPlayingOrWillChangePlaymode)
    {
        EditorApplication.update -= IdleUpdate;
    }
}

static void CheckAndShow()
```

```
{
    if (System.DateTime.Now.Hour == 17 && System.DateTime.Now.Minute == 0 &&
System.DateTime.Now.Second == 0)
    {
        if (EditorUtility.DisplayDialog("Order The Dinner!", "It's time to order
the dinner", "Ok", "Cancel"))
        {
            //replace to the dinner webset
            Application.OpenURL("http://www.baidu.com");
        }
    }
}
```

这样每天下午 5 点，都会有弹窗提示，效果如图 13.1 所示。

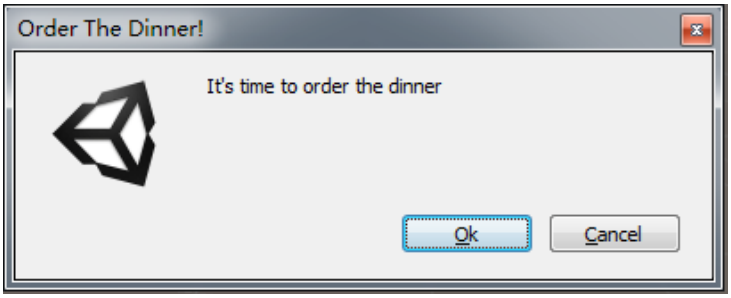


图 13.1

单击“OK”按钮即可跳转到指定订餐网站，这里配置的是百度首页，大家可根据实际情况进行更改。

运行状态

现在还有个小瑕疵，就是游戏状态下，并不会有服务启动。因为 Unity 在运行游戏时，会清除之前内存中的缓存，所以需要添加运行阶段的服务启动逻辑，更改代码如下：

```
 [InitializeOnLoad]
public class AutoServer : MonoBehaviour
{

    static AutoServer()
    {
        if (!EditorApplication.isPlayingOrWillChangePlaymode)
```

```
        {
            EditorApplication.update += ScriptLoaded;
        }
        else
        {
            EditorApplication.update += ScriptRunning;
        }
    }

    //...

    static void ScriptRunning()
    {
        EditorApplication.update -= ScriptRunning;
        EditorApplication.update += RunningUpdate;
    }

    static void RunningUpdate()
    {
        CheckAndShow();
        if (!EditorApplication.isPlayingOrWillChangePlaymode)
        {
            EditorApplication.update -= RunningUpdate;
            EditorApplication.update += ScriptLoaded;
        }
    }
}
```

在运行脚本初始化中，开启一个不同的更新逻辑。在退出此状态时，将 ScriptLoaded 的代理添加到 update 上。这样就完成了自定义服务的闭环。

总结

通过制定自定义的后台服务，我们可以在编辑器后台运行很多自定义逻辑。比如制定自己的消息队列、监控游戏运行状态的参数变化等。

扩展问题

众所周知，当停止运行时，Unity 会自动复原对象的属性。现在希望在停止之后，能够保留选中的 GameObject 属性，该如何实现呢？（[:savego]）

13.2　自动注册框架

使用了导入资源的 OnPreprocessModel 和移动文件夹的 OnWillMoveAsset 之后，大家发现这种子类重写方法的编程方式非常灵活方便，所以现在希望实现一套类似的机制，来自动注册函数。当子类实现某些接口后，就可以在恰当的时机（例如游戏刚刚开始运行），自动调用对应的逻辑，请问该如何设计呢？

问题分析

在 13.1 节，我们介绍了如何获取编辑器的状态变化。一旦有了这些状态量，我们就可以自定义许多功能。例如，在启动编辑器时载入配表信息，在游戏运行时更改日志输出状态等。但随着功能的增加，框架中的代码会趋于臃肿，变得混乱和难以维护。

由于这是在编辑器中的逻辑，因而不必太在意性能。出于对开发效率和维护成本的考虑，利用 C#反射自动注册逻辑实现是更优的选择。

创建接口

首先创建接口类 EditorState.cs，它定义了四个状态的调用接口，编辑代码如下：

```
using UnityEngine;
using System.Collections;

public interface EditorState
{
    void StartEditor();
    void PreRun();
    void RunGame();
    void WillEndRun();
}
```

这个接口类中分别定义了四个状态，即开启编辑器、开始游戏之前、开始游戏之后和结束游戏之前。

创建子类对象

如何获取一个类的所有子类呢？通过 C#的 Assemblies 类中的接口函数，我们能找到想要的

类，接着动态创建类的对象即可。这里我们创建一个名为 AutoRegister.cs 的文件，在其中编写如下代码：

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

[InitializeOnLoad]
public class AutoRegister : MonoBehaviour
{
    public static T[] GetAllImplementTypes<T>(System.AppDomain aAppDomain)
where T : class
    {
        var result = new List<T>();
        var assemblies = aAppDomain.GetAssemblies();
        foreach (var assembly in assemblies)
        {
            var types = assembly.GetTypes();
            foreach (var type in types)
            {
                if (typeof(T).IsAssignableFrom(type))
                {
                    if (!type.IsAbstract)
                    {
                        var tar = assembly.CreateInstance(type.FullName) as T;
                        result.Add(tar);
                    }
                }
            }
        }
        return result.ToArray();
    }
}
```

这里通过 IsAssignableFrom 来判断类的类型，进而区分出想要的子类。通过 CreateInstance 创建一个类的对象。

自动检测

注册的时间点的相关代码可以参考编辑器服务一篇中的实现。这里复制 AutoServer.cs 中的

代码并做些改动，如下所示：

```
private static List<EditorState> m_operations = new List<EditorState>()

static void ScriptLoaded()
{
    m_operations.Clear();
    var ops =
GetAllImplementTypes<EditorState>(System.AppDomain.CurrentDomain);
    foreach (var operation in ops)
    {
        operation.StartEditor();
        m_operations.Add(operation);
    }

    EditorApplication.update -= ScriptLoaded;
    EditorApplication.update += IdleUpdate;
}

static void IdleUpdate()
{
    if (EditorApplication.isPlayingOrWillChangePlaymode)
    {
        foreach (var operation in m_operations)
        {
            operation.PreRun();
        }
        EditorApplication.update -= IdleUpdate;
    }
}

// Run implement is same way
//......
```

从上面的代码可以看出，我们将类的实例对象存储到了一个数组中，并在运行之前，分别调用其中的 PreRun 函数。

编写子类

完成了自动注册的逻辑后，我们编辑一个测试类。创建 EditorOptTest.cs，编写如下代码：

```
using UnityEngine;
using System.Collections;

public class EditorOptTest : EditorState
{
    public void StartEditor()
    {
        Debug.Log("Init Editor Script");
    }
    public void PreRun()
    {
        Debug.Log("Will Run Game");
    }
    public void RunGame()
    {
        Debug.Log("Run Game Now");
    }
    public void WillEndRun()
    {
        Debug.Log("Will End Game");
    }
}
```

编辑完成后，单击“运行”按钮，可以看到对应的输出按期望显示。

总结

有了自动注册的框架，就可以制定很多定制服务了。另外，采用相同的思路，也可以不局限于编辑器的状态变化，自定义出更多符合需求的事件。

扩展问题

（1）请补全 RunGame 和 WillEndRun 的相关代码实现。

（2）请问有必要为两种运行状态添加 Update 接口吗（[:autoregupd]）？

13.3 遍历文件

最近大家都觉得字体颜色不太对，准备统一更改文字颜色。项目中的各个界面都存成了预

设，且分布在不同层级的子文件夹中，有什么办法统一更换这些文字的颜色吗？

问题分析

项目进行到中后期时，很多界面和功能都已经趋于完善，整体的改动也变得越来越多。有时我们会期望改动一些特定的资源，比如更改预设的内容，更改模型、贴图的设置，检查某些错误的资源类型等。

比较灵活的办法是：遍历某个路径下的所有文件，按照过滤类型返回相应的文件路径，然后再回外层的处理逻辑。另外，过滤规则也应当开放给外层，以提高不同需求的通用性。

遍历算法

遍历文件夹的实现方式有很多种，这里提供一种递归的实现。创建 ResCheck.cs 文件，编辑代码如下：

```
using UnityEngine;
using System.Collections;
using System;
using System.IO;
using UnityEditor;

public class ResCheck : MonoBehaviour
{

    public static void WalkDir(string path, string extRule, Action<string>
callback)
    {
        DirectoryInfo baseDir = new DirectoryInfo(path);

        if (baseDir == null)
        {
            Debug.Log("You should start from a dir :" + path);
        }
        else
        {
            if (baseDir.Attributes == FileAttributes.Directory)
            {
                foreach (var dir in baseDir.GetDirectories())
                {
```

```
                WalkDir(dir.FullName, extRule, callback);
            }
        }
        FileInfo[] files = baseDir.GetFiles(extRule);
        foreach (var file in files)
        {
            callback(ToAssetPath(file.FullName));
        }
    }
}

private static string ToAssetPath(string fullPath)
{
    string[] splited2 = fullPath.Split(new string[] { "Assets" },
     System.StringSplitOptions.RemoveEmptyEntries);
    string nameShort = "Assets" + splited2[1];
    return nameShort;
}
}
```

值得一提的是，由于使用 System.IO 来遍历目录，所以在返回路径信息时，需要将路径转换为以“Assets/”开头的项目路径。在 ToAssetPath 函数中，通过对字符串拆分的方式，将路径进行转化。

替换颜色实现

完成上述编码之后，就可以编写菜单功能函数了，在 ResCheck.cs 中添加如下代码：

```
 [MenuItem("MyMenu/WalkDir")]
static void MyWalk()
{
    //获取选中对象
    var objs = Selection.objects;

    if(objs != null && objs.Length>0)
   {
        //遍历对象
        foreach (var eachDir in objs)
        {
            WalkDir(AssetDatabase.GetAssetPath(eachDir),"*.prefab", (string
                                                                    path) =>
```

```
            {
                var prefab = AssetDatabase.LoadAssetAtPath<GameObject>(path);
                var texts = prefab.GetComponentsInChildren<Text>();
                foreach (var t in texts)
                {
                    t.color = new Color(1, 0, 0);
                }
                AssetDatabase.SaveAssets();
            });
        }
    }
}
```

编写完成后，选中对应的文件夹，单击菜单项，即可更改组件中的颜色属性，效果如图 13.2 所示。

图 13.2

按类型查找

有时我们也会按类型查找一些元素。如果不考虑目录，而是在所有资源中进行遍历，那么可以使用另一个接口函数 FindObjectOfType，它可以返回整个项目中所有某个类型的资源。

例如，在场景编辑的过程中，有时美术会误用一些材质，比如 Unity 默认的 Standard 材质。现在希望检查所有的材质，如果有使用错误的，就将其定位出来。在 ResCheck.cs 中添加如下代码：

```
[MenuItem("MyMenu/FindMaterial")]
static void MyMat()
{
    var mats = Resources.FindObjectsOfTypeAll(typeof(Material)) ;
    foreach (var mat in mats)
    {
        var shader = ((Material)mat).shader;
        if (shader.name.Contains("Standard"))
```

```
        {
            Debug.LogWarning("Use Wrong Shader: Standard, Path is :"
                + AssetDatabase.GetAssetPath(mat), mat);
        }
    }
}
```

编辑完成后，当单击菜单项 FindMaterial 时，就会在日志中输出项目中有问题的材质。

总结

通过遍历文件或类型查找的方式，可以灵活地管理项目中的资源进行。例如，可与错误检测、资源替换、属性更改等需求灵活组合，希望大家能够熟练掌握。

第五部分

独立游戏

独立游戏是很多游戏人的梦想，也被一度认为是游戏创意的源泉。虽然这种孤胆英雄式的探索，很难成就庞大的商业帝国，但它的意义应该被肯定，它的精神应该被传承。

作为独立游戏开发者，最大的难点就是要全能，即可以从事其他角色的工作。一名做程序出身的开发者，不仅要能编写前后端的代码，有时还要兼任策划的部分责任。美术出身的开发者，在完成自己本职工作之后，可能也会参与运营相关的工作中来。因此，全面的技术成为制约独立游戏的瓶颈。

在本书的前面部分，笔者有意的将美术相关内容渗透在书中，但只有这些知识还是不太够用。鉴于笔者曾有过创业经历，从事过部分其他角色工作，因此在这里分享出来，希望大家在这条路上能够少走弯路。

第 14 章
角色分工

大多数游戏人都想做自己的游戏，九分努力加一分热爱有时候还真能铸成不朽的作品。

其实做独立游戏需要的能力一点也不比商业游戏少，有时还要面对困难的抉择。例如，为了自己的那份坚持，游戏的付费设计可能无法带来大量的收入。“痛，并快乐着”，成了独立游戏最大的特点。风雨过后未必有彩虹，但梦想就是这样，总有人为之献出青春，只求无悔此生。

对于独立游戏来说，理想的团队是 4 人，分别为开发、策划、美术、商务运营。不过通常很难有这种规模，因此独立游戏开发者通常需要身兼数职，甚至要担负全部角色的工作。单人团队通常是程序员出身，因为这是最不可或缺的角色。作为开发者，有必要了解其他角色的工作技巧，以保证项目能顺利进行下去。

值得注意的是，当角色转换时，应按照目标角色行为习惯思考问题，不被自己的习惯干扰。例如，当作为策划考虑游戏系统时，就不要衡量系统是否容易实现。当作为美术设计效果时，也不要思考是否会对性能产生影响。

本章主要介绍编程之外的技巧，为大家指出常见的弯路，以期避开可能遇到的风险。

14.1　产品策划

对于独立游戏开发者来说，策划技巧会稍微有些宏观。通常游戏的核心玩法已经确定，而独立游戏的外围系统注定不会太复杂，系统或数值会简化很多。因此策划会关注一些对团队很重要却容易忽视的问题。

不忘初心

时间就是生命，在人生的高速路上狂奔时，常常忘了自己要去往何方。为什么要做独立游戏？不同的人有不同的答案，这些理由将伴随游戏的整个开发过程。将它写在最前面，是因为它尤为重要。当面临压力时，它就是强心针；当众人迷茫时，它就是灯塔。

确定边界

即使是独立游戏，也不可能用无限的时间做下去。人力有限，我们必须合理安排工作量。例如，服务器开发压力很大，可不可以只做单机玩法？美术资源产能不足，是否需要更改角色数量？关卡数目太多，策划配置时间太短，能否暂时关闭对应关卡？

一旦将工作量化，就涉及能力的边界。通常情况下，我们都要舍弃一些功能，以保障项目按计划完成。一个清醒的开发者应该知道自己的强项和弱项，避免做超出能力范围的事情。虽然冒险有可能成功，但从概率上讲，这种行为对项目有害无益。

鸡肋：策划案

策划案的存在是否有必要？这个问题很容易引起争执。作为独立游戏开发者，一切理应从简，抛开低效的形式主义。曾经有一句话在游戏圈中广为流传：“收起你那装腔作势的策划案，直接告诉我抄的是什么游戏”。很多时候，洋洋洒洒数千言的文档，只是在用语言描述其他游戏的相同功能，这中间耗费的人力物力不言自明。

反过来说，是不是不需要策划案呢？也不应如此绝对。对于相同品类的游戏来说，功能雷同是家常便饭。但这就像抄作业，两份答案不一样怎么办？我们需要一个取舍标准，这才是应该记录到策划案中的信息。几个月过后，产品定位很可能已经发生变化。做与不做需要重新判断时，当初走上这条路的原因就尤为重要。

毕竟知道来路才不会迷失自己。

平衡性

是否好玩是个很虚的概念，但其实是在讨论游戏的平衡性。太难的游戏，玩家会受到折磨而放弃，太简单又会失去兴趣。通常说来，将游戏设计为“易于上手，难于精通”，可以最大限度地让玩家接受挑战。

平衡性也体现在游戏要素的分配上。例如，一场战斗的胜利有多少因素取决于玩家的操作？如果操作因素影响过小，会使游戏过于休闲；操作因素过大，又会使玩家觉得太累。设计核心玩法时，我们要综合考虑操作、数值、成长、运气等因素对战斗过程和结果的影响，调配出合适的战斗体验。

系统设计

系统不会孤立存在，它是游戏内资源流动的表象。从资源产出，到成长体系，再到资源回收，这是一个完整的链条。系统可大可小，全看我们的目的，但每个系统都应为资源的流转提供支持。价值高的资源通常会以活动的形式出现。这样既方便控制数量，也可以拉动玩家的积极性。

如果要增加系统，最简单的方法是添加平行体系。即产出新的资源，玩家可以获得新的能力。由于不干扰先前的资源产出，所以交叉系统的复杂度会大幅下降，对应的数值控制也会简化很多。但要注意，不要一次性给玩家太多的内容，很多人无法在短时间内适应大量全新的玩法。

活动安排

逢年过节推出活动是游戏拉动活跃和充值最主要的方式。不过对于独立游戏来说，每个节假日都安排活动是比较困难的。处理突发的 Bug 与玩家的反馈，会耗掉大量的时间，为节日安排特殊的玩法几乎是不可能的任务。因此独立游戏应该简化活动，最好可以通过简单的更改配置调整产出，再通过公告、邮件或是活动页的方式发布活动。一方面我们应尽早设计灵活的架构，另一方面也要提前安排工期，预留足够的时间制作相关的配置和美术资源。

总结

独立游戏通常源于一个创意，但我们需要认识到，好创意并不等于畅销的创意。认为自己能代表大部分玩家，是产品设计的死穴。在确定核心玩法时，策划要能预想出它最终的样子，以保证它能被普通大众玩家所接受，而不是卖不出去的诡异想法。在合理的玩法下，设计出有趣且平衡稳定的游戏才是策划追求的目标。

14.2　美术设计

独立开发者通常需要在没有美术人员的参与下，完成资源的整理改造甚至制作。这对于非专业人士还是有不小的难度的。总体来说，2D 资源所见即所得，大家通常都能够处理，这里就略过，不做展开。本篇主要介绍 3D 资源的相关知识。

［注］本节内容部分参考（参考书目 2）

3D 模型制作流程

对于独立游戏来说，3D 资源的制作很可能不会经历完整周期，更多时候，只是一部分美术步骤。例如，在现有模型上更改固有色贴图、法线贴图等。熟悉制作 3D 模型的整套流程，有助于我们在合适的阶段做定制修改。

制作 3D 模型的步骤大致如下：

（1）制作中模；

（2）雕刻高模；

（3）拓扑低模；

（4）划分 UV；

（5）烘焙法线贴图、AO 贴图；

（6）制作固有色贴图、高光贴图等其他贴图。

下面具体分析每个步骤。

1）中模

中模是为后续雕刻准备的底板模型。通常在 3D Max 或 Maya 中制作，然后导入到雕刻软件中。一般来说，中模应尽量使用四边面，不要出现大于五边公用同一顶点的情况。另外网格应尽量均匀分布，不要过疏或过密，这可以防止雕刻时出现不光滑的情况。

2）高模

高模不会限制面数，通常在专业的雕刻软件中完成。ZBrush、Mudbox、3Dcoat 都是这个领

域的佼佼者。建模布线的要点在于使用最少的面来表现模型的形体。建模时主要分为点和面两个方向来检查。对于点，不可以有空点，也应尽量避免在尖端处只用一个点，例如弓箭的箭头。对于面，不应有开放的几何体，同时要避免多余、重合或穿插的面。

3）低模

在游戏引擎中运行的模型需要严格限制面数，因此需要将高模拓扑成低模。通过 ZBrush 的拓扑功能自然可以实现，但更多时候会使用更加方便的软件，如 Topogun 或 3Dcoat。

拓扑的首要目的是减面，因此最重要的自然是面数。另外，还要考虑划分之后动画制作对布线的需要。例如，关节上建议使用三根线划分，这样方便动画后续蒙皮时设置权重。拓扑时从能活动的区域入手，再兼顾起伏较大的区域，最后自动补全剩余部分。一般来说，模型尽量使用三角面，在减面的过程中，把所有不产生形变的点都合并掉。理论上讲，表面有大量平行线或四边面的模型，都有优化的空间。

为了降低性能消耗，对于场景中玩家无法靠近的远景的植物，通常采用插片的做法。插片常见的有十字插片和田字插片。

◎ 十字插片，就是用一组相互垂直的面片承载植物的透明贴图。例如草丛、树叶、花瓣等。

◎ 田字插片，就是将模型分成四部分，像田字一样，然后将正中心的点拉出平面，与其他四点不共面。这样从侧面看也会有厚度。

插片需要双面材质，但为了降低性能消耗，通常会复制出一个面片贴在反面。虽然增加了面数，但总的来说还是节省了性能。另外，这些植物会在场景中大量使用，制作时也需要注意面数不要太高。

4）划分 UV

为模型划分 UV 也是比较重要的工作。使用 Maya 的 UV Texture Editor 和 3D Max 的 UVW 工具都可以对其进行精确的控制。也可以使用 UVlayout 或 Unfold3D 这种更自动化的工具完整划分。在划分 UV 时，应尽量做到以下几点。

◎ 合理摆放 UV 分块。尽量保持横平竖直，或将相关的部分放在一起。一方面方便后续绘制细节，另一方面也可以一定程度上减少锯齿。

◎ 利用 UV 重叠。如果是纹理重复的物体就可以利用相同的 UV 来节省空间。相对应的，

如果不是需要，就不应该有重叠的现象。

◎ 减少 UV 十分接缝。可以适当地少切几刀。

◎ 把 UV 接缝安排在不明显处。例如，隐藏到两种颜色的交界处，或者是布料的缝合线上。如果实在不行，就要在 Bodypaint 3D 或 MudBox 中，处理贴图接缝处的像素了。

◎ 按主次分配 UV 摆放比例。主要展现的地方应占比较大的区域，次要的则可以适当缩小。

◎ 尽量有规律地划分 UV。如果后续需要做 UV 动画，那么 UV 的排布就比较重要了，最好提前沟通。

5）制作贴图

只有模型，没有贴图是没法使用的。贴图种类繁多，在制作贴图时，有些技巧需要注意。

（1）固有色贴图

在制作固有色贴图时，我们需要将模型的 UV 线条导出到一张 PNG 图中。将这张底图导入 Photoshop，在上层依次添加 AO 贴图、填充色、纹理材质、磨损污迹等需要的效果层，并调整好混合关系。

（2）高光贴图

通常来说，金属的亮度最高，布料的亮度最低。高光贴图的制作通常是在固有色贴图的基础上进行调整。将固有色贴图去色，并手动调整色值。凸起的部分较亮，凹陷的部分较暗，但不要出现纯黑色或纯白色。最后整体调整图片的灰度，以达到合理的曝光效果。如果有需要，还应再做细节调整。

（3）法线贴图

法线贴图与其说是一张图，不如说是一个储存了数组信息的配置文件。因此在 Photoshop 中直接更改图片毫无意义，也几乎不可能得到正确的效果。法线贴图一旦生成，就无法将其转为灰阶的凹凸贴图。

对模型而言，从内到外的法线是正方向，反过来就是负方向。烘焙法线时，如果法线方向是反的，则生成的表面凹凸就会出错。因为 Maya 与 3D Max 的法线贴图绿色通道是反的，如果法线方向错了，那么生成的法线贴图会包含黄色区域，这在正常的法线贴图中绝不会出现。

在模型软件中，每根法线都应为绿色，如果出现黄色，就需要翻转法线。修正所有法线后，

才可以开始烘焙法线贴图。通常来说，使用 xNormal 烘焙软边模型法线贴图效果更好一些。

由于部分细节通过布线做模型，或者用 ZBrush 雕刻的方式制作会耗费很大精力，因此很多时候会采用平面转法线的方式制作法线。平面转法线要求 UV 拆分时要十分小心，不允许有拉抻的现象。如果准备好合适的贴图，就可以通过 NVIDIA 官网下载 Photoshop 的插件对其进行转换（[:nvidiatools]）。打开滤镜菜单 NVIDIA ToolsNormalMapFilter，更改相应设置即可。由于不可以修改转换之后的贴图，因此建议分层绘制法线效果，做一层转一层，以降低最终效果调整的难度。

上面说的是精细雕刻的处理方法，如果是制作整体的凹凸效果，推荐使用更易于调节的工具 CrazyBump。它可以自动产生一个合适的贴图，并在调整参数的同时，可实时预览法线效果。

（4）DDS 贴图

DDS（DirectDraw Surface）是一种图片格式，它是图片经过微软 DirectX 纹理压缩（DirectX Texture Compression，简称 DXTC）后得到的资源。它可以减少图片加载的时长，根据用途的不同，有以下三种格式：

◎ DXT1 只保留 RGB 颜色，因此一般用作非透明图片等压缩方式。
◎ DXT3 使用 4 位的 Alpha 值，可用作无渐变的透明图。
◎ DXT5 使用线性插值的 4 位 Alpha 值，可认为与原图效果相同。

由于是压缩格式，所以我们需要使用特殊的方法打开它。如果使用 Photoshop，则可以在 NVIDIA 官网下载 Photoshop 的插件（[:nvidiatools]）。打开后就可以对它进行编辑了，保存时也可存为 DDS 格式。如果不想打开 Photoshop，也可以使用其他工具预览，例如 WTV 或者 DXBmp。

（5）无缝贴图

无缝贴图是指在与自身贴图连接时不产生接缝的贴图。通常分为二方连续和四方连续两种。二方是指水平或竖直拼接，四方是指上下左右都可以拼接。因此二方可以拼接出无限长的贴图，四方可以拼出无限大的贴图。

制作无缝贴图通常使用 PixPlant，它可以像 Photoshop 一样，分层制作无缝贴图，并创建更真实的视觉效果。另外，它可以矫正歪斜的图片，便于倾斜图片的制作。最后，它还可以将文件导出成法线，省去了在其他软件中转法线的操作。

动画

对于游戏中的角色或某些特定物件，我们要让它在游戏中动起来。这些动画通常在 3D Max 或 Maya 中制作。大体要经过以下步骤。

（1）模型自检：主要检查位置是否归零、顶点是否缝合、关节布线是否合理等。

（2）绑骨：为模型绑定尽量少的骨骼数量。

（3）蒙皮：为顶点设置权重，使动作看起来平滑自然。

（4）制作序列帧动画：根据需求制作多个动作。

动画制作属于易学难精的技巧。在有限的时间内做出优质的动画，需要多年的摸索和经验积累。如果有条件，建议找专业的动画设计师完成这项工作。

总结

抛开时间和精力，如果仅折算金钱成本，占预算中最大比例的就是美术资源花费。独立游戏开发者通常要在品质和支出之间做权衡，如果不具备相关知识，就很难取舍，也容易被以次充好。因此，希望大家重视美术的质量。现在，越来越多的玩家已把游戏的美术质量放在了影响评价因素的首位。

14.3　运营知识

玩家的数据对游戏有着很强的指导作用，通过数据我们可以从各个层面来衡量游戏的健康状态，供我们进行分析，发现游戏存在的问题。那么究竟该留意哪些数据呢？或者说该从何入手了解这些数据？本节介绍常见的运营概念，以帮助大家了解究竟该注意些什么。

［注］本节内容部分参考（参考书目 13）

14.3.1　用户规模数据

DNU

日新增用户（Daily New User，DNU）可以用来衡量：

◎ 渠道贡献的用户份额；
◎ 宏观走势，确定投放策略；
◎ 是否存在大量垃圾用户；
◎ 注册转化率分析。

DOSU

日一次会话用户数（Daily One Session Users，DOSU）是 DNU 的引申指标，重点关注首次登录之日后 7 天内或 14 天内再未打开游戏的用户。了解非留存用户首日行为及比例，有助于优化产品导入用户的流程。

◎ 推广渠道的质量评估。
◎ 用户导入是否存在障碍点，如网络状况、加载时间、客户端崩溃等。
◎ 游戏新手引导设计分析点之一。

DAU

日活跃用户（Daily Active User，DAV）是比较重要的指标，它用于：

◎ 衡量核心用户规模；
◎ 分析产品生命周期；
◎ 分析产品活跃用户流失情况，分解活跃用户；
◎ 判断用户活跃率，包括活跃用户/累计用户。

WAU

周活跃用户（Week Active User，WAU）用来跟踪：

◎ 周期性用户规模；
◎ 周期性变化趋势，主要是推广期和非推广期比较。

MAU

月活跃用户（Mouth Active User，MAV）变化幅度较小，是产品用户规模和稳定性的风向标。但在推广时期数据波动较大，可用于：

◎ 用户规模稳定性评估；
◎ 推广效果评估；

◎　分析总体游戏用户规模变化。

DEC

日参与次数（Daily Engagement Count，DEC）主要用于记录用户参与的积极性，它可以：

◎　参与频率分析，尤其是在上线、版本更新和运营活动等期间，监控该数据，了解用户对产品的反馈；
◎　衡量用户黏性，可对不同用户群进行分析（活跃、新增、付费）。

DAOT

日均使用时长（Daily Average Online Time，DAOT）可用于：

◎　分析产品质量问题；
◎　观察不同时间维度的评价使用时长，了解不同用户群的习惯；
◎　渠道质量的衡量标准之一；
◎　流失用户分析依据之一。

DAU / MAU

用户活跃程度，理论不低于 0.2。0.2×30=6 天，即每个用户每个月至少有 6 天登录游戏，这个比例也用于衡量用户规模。它用于考量：

◎　游戏人气变化的风向标；
◎　用户活跃天数的评估。

Retention Ratio

留存率（Retention Ratio）通常考察+1 日、+3 日、+7 日。以次日为例其算法为：DNU 在+1 日登录的用户数占当日 DNU 的比例。新增当日不计入天数。这个值为衡量游戏质量的最重要标准。在对比时，统一的计算标准是要重点考虑和关心的，比如是账号还是设备，是具体某日还是在某日内，是新增账号还是活跃账号。这可以导致 8 种不同的结果。

◎　以账号为维度统计，在活跃账号计算方法中，次日留存率方面，活跃账号留存率大概是新增账号留存率的 1.7 倍左右，而七日留存大概在 3.6 倍左右。
◎　以设备为维度计算，在次日留存率中，活跃设备留存率大概是新增率设备留存率的 1.8 倍，而七日留存率则为 3.5 倍。

Churn Ratio

日流失率（Churn Ratio）是指，统计日登录游戏，但随后 x 天未登录游戏的用户数占当日活跃用户数的比例。其中 x 可以根据需要变化。周流失是指，上周登录过游戏但本周未登录的用户数占上周活跃用户数的比例。月流失是指上月登录过游戏，但本月未登录游戏的用户数占上月活跃用户数的比例。

流失率是产品进入稳定期时需要重点关注的指标。稳定期的收益和活跃都很稳定，如果存在较大的流失率，则需要通过该指标分析，逐步查找究竟是哪些用户离开了游戏，特别要注意付费用户的反馈。

14.3.2 用户价值数据

PA

付费用户（Payment Account，PA）是盈利计算的基础值，它用于考量：

◎ 游戏产品的收益转化能力标准；
◎ 用户付费关键点和转化周期；
◎ 付费转化效果的评估。

APA

由于活跃付费用户数（Active Payment Account，APA）通常按月计，因此它的计算公式可简单地认为：

APA = MAU × MPR

ARPU

平均每用户收入（Average Revenue Per Uer，ARPU）的计算公式为：

ARPU = Revenue/Players

即总收入除以总人数，一般按月计。ARPU 用于预估不同规模下的收入，也是 LTV 的重要参考，它可以考量：

◎ 不同渠道的用户质量判断；

◎　产品收益贡献；
◎　活跃用户人均收入与投放成本的关系。

ARPPU

平均每付费用户收入（Average Revenue per Paying User，ARPPU）的计算公式为：

ARPPU = Revenue / PaymentUser

如果按月为单位，则为 Revenue / APA。它用于考察：

◎　付费用户的付费能力和梯度变化；
◎　付费用户的整体付费趋势和不同付费阶层差异；
◎　大 R、小 R 对整体收入的影响。

LTV

生命周期价值（Life Time Value，LTV）可以简单地看成长期的 ARPU。它的计算公式为:

$$LTV = ARUP \times LT$$

其中 LT 为生命周期，一般以月计算平均值。

跟踪某日或某周的新增用户，计算该批用户在随后的 7 日、14 日、30 日的累积收入贡献，然后除以新增数量，就可以算出本批用户的 ARPU 值，进而算出本批用户的 LTV。可以根据不同阶段的 LTV 绘制曲线，了解整体推广的效果。它用于衡量：

◎　用户收益贡献周期；
◎　用户群与渠道的利润贡献，LTV 与 CAC 的衡量；
◎　LTV 不区分付费与非付费用户，因而衡量的是整体价值。

14.4　总结

运营的工作核心是在数据的指引下做出有效的决策，使游戏向着期望的方向发展。我们需要根据实际情况关注不同的数据。例如，缺少用户的原因有很多，从源头上讲，可能是宣传不到位，也可能是安装包有些设备不兼容，也可能是有很多渠道导入了垃圾用户（非真实玩家）。从流失上讲，可能是客户端不稳定，可能是有些新手设计使玩家无法通过，可能是某些关卡设

计不合理等。各种情况都有可能，而真相只能依靠数据告诉我们。

相对于商业游戏，独立开发者可以稍微淡化运营，但也需要知道相关概念，只有这样才能在出现问题时，快速解决。

参考文献

[1] 游艺网教育部. 次世代游戏开发基础. 北京：清华大学出版社，2015.

[2] 游艺网教育部. 次世代游戏角色制作. 北京：清华大学出版社，2015.

[3] 李瑞森，张卫亮，王星儒. 网络游戏场景设计与制作实战. 北京：电子工业出版社，2015.

[4] Mike Bailey, Steve Cunningham. Graph shaders(Second Edition). 刘鹏，译. 图形着色器——理论与实践（第二版）. 北京：清华大学出版社，2013.

[5] Rick Parent. Computer Animation algorithms and Techniques, Second Edition. 刘祎，译. 计算机动画算法与技术（第二版）. 北京：清华大学出版社，2012.

[6] Robert Nystrom. Game Programming Patterns. GPP 翻译组，译. 游戏设计与开发. 北京：人民邮电出版社，2016.

[7] 中嶋谦互. オンラインゲームを支える技術 壮大なプレイ空間の舞台裏. 毛姝雯，田剑，译. 网络游戏核心技术与实战. 北京：人民邮电出版社，2014.

[8] Scott Rogers. The Guide To Great Video Game Design(Second Edition). 孙懿，高济润，译. 通关！游戏设计之道（第二版）. 北京：人民邮电出版社，2017.

[9] Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics, Third Edition. 詹海生，译. 3D 游戏与计算机图形学中的数学方法（第三版）. 北京：清华大学出版社，2016

[10] 冯乐乐. Unity Shader 入门精要. 北京：人民邮电出版社，2016.

[11] Sanjay Madhav. Game Programming Algorithms and Techniques. 刘瀚阳，译. 游戏编程算法与技巧. 北京：电子工业出版社，2016

[12] 王睿杰等. 创世学说：游戏系统设计指南. 北京：电子工业出版社，2016.

[13] 于洋，余敏雄，吴娜，师胜柱.游戏数据分析的艺术. 北京：机械工业出版社，2015.