哔哩哔哩次元游戏人生

目　录

目　录 I
C、C#、C++ 22
一、　数据类型 22
二、　运算符优先级和结合性 22
三、　C# 结构体和类-Struct and Class 23
1. Struct 23
2. Class 23
3. 类与结构体的选择 27
4. 类与结构体的区别 27
四、　构造函数 28
五、　虚函数 28
5. 虚函数 28
6. 纯虚函数 28
7. 接口(Interface)和抽象类的区别 28
六、　C#中四种访问修饰符 29
1. 属性修饰符 29
2. 存取修饰符 29
3. 类修饰符 29
4. 成员修饰符 29
七、　ref 和 out 关键字 29
1. ref 29
2. out 30
3. ref 和 out 的相同点 30
4. ref 和 out 的不同点 30
八、　分部类和方法 30
1. 分部类 30
2. 分部方法 31
九、　面向对象的三大特点 31
1. 继承 31
2. 封装 31
3. 多态性 31
十、　面向对象程序设计方法的主要优点 31
十一、　C#索引 32
十二、　运算符重载 32
十三、　值类型和引用类型 32
十四、　值类型和引用类型 32
十五、　C#索引器 32
十六、　C#反射-Reflection 32
十七、　C#常用 String 方法 32
Lua 语言 33
一、　Lua 基本语法 33
1. 单行注释 33

哔哩哔哩次元游戏人生

2. 多行注释.....33
3. 标示符.....33
4. 关键字.....33
5. 全局变量.....33
二、 数据类型.....34
1. 空-nil.....35
2. 布尔-boolean.....35
3. 数字-number.....36
4. 字符串-string.....36
5. 表-table.....37
6. 函数-function.....38
7. 线程-thread.....39
8. 自定义类型-userdata.....39
三、 Lua 的变量、赋值、索引.....39
1. 变量.....39
2. 赋值语句.....40
3. 索引.....40
四、 循环.....41
1. 循环类型.....41
2. 循环控制语句.....44
3. 无限循环.....45
五、 流程控制.....45
1. if 语句.....46
2. if…else…语句.....46
3. if...elseif...else 语句.....47
4. if 嵌套语句.....47
六、 函数.....48
1. 函数定义.....48
2. 多返回值函数.....49
3. 可变参数.....50
4. 固定参数加上可变参数.....51
5. 命名参数.....52
七、 再论函数.....53
1. 闭包.....53
2. 非全局变量.....55
3. 正确的尾调用.....57
八、 运算符.....59
1. 算术运算符.....59
2. 关系运算符.....59
3. 逻辑运算符.....60
4. 其他运算符.....60
5. 运算符优先级.....60
九、 Lua 字符串.....60
1. 转义字符.....61

哔哩哔哩次元游戏人生

2. 字符串操作.....61
3. 字符串截取.....63
4. 字符串格式化.....63
5. 字符与整数相互转换.....65
6. 匹配模式.....65
十、 Lua 数组.....66
1. 一维数组.....66
2. 多维数组.....66
十一、 Lua 迭代器.....68
1. 泛型 for 迭代器.....68
2. 无状态的迭代器.....69
3. 多状态的迭代器.....69
十二、 表-table.....70
1. 表的构造.....70
2. Table 操作.....71
十三、 数据文件与持久化.....73
1. 序列化.....75
十四、 环境.....79
1. 使用动态名字访问全局变量.....80
2. 声明全局变量.....81
3. 非全局的环境.....82
十五、 模块与包.....84
1. 模块.....84
2. require 函数.....84
3. C 包.....86
十六、 Package.....86
1. 基本方法.....86
2. 私有成员-Privacy.....88
3. 包与文件.....89
4. 使用全局表.....89
十七、 元表-Metatable.....91
1. setmetatable(table,metatable).....91
2. getmetatable(table).....91
3. __index 元方法.....91
4. __newindex 元方法.....92
5. 为表添加操作符.....93
6. __call 元方法.....94
7. __tostring 元方法.....95
十八、 协同程序-coroutine.....96
1. 什么是协同(coroutine).....96
2. 线程和协同程序区别.....96
3. 基本语法.....96
十九、 文件 I/O.....98
1. 模式值.....99

哔哩哔哩次元游戏人生

2. 简单模式.....99
3. 完全模式.....100
二十、 错误处理.....101
1. 语法错误.....101
2. 运行错误.....102
3. 错误处理.....102
二十一、 调试-Debug.....103
二十二、 垃圾回收.....106
1. 垃圾回收器函数.....106
二十三、 面向对象程序设计.....107
1. 类.....108
2. 继承.....109
3. 多重继承.....111
4. 私有性-privacy.....113
5. Single-Method 的对象实现方法.....114
二十四、 优化.....115
1. 尽可能使用局部变量.....115
二十五、 技巧.....115
二十六、 工具.....115
1. IntelliJ IDEA.....115
2. EmmyLua.....117
Unity.....121
一、 PipelineAsset.....121
1. Quality.....121
2. Lighting.....122
3. Shadows.....122
二、 Lighting 窗口.....122
1. Environment.....122
2. Other Settings.....122
三、 Navigation 窗口.....123
3. Agents（代理类型）.....123
四、 Nav Mesh Agent.....123
五、 协程与委托.....123
1. Coroutine 协同程序.....123
2. Delegate 委托.....123
六、 Event 事件.....124
七、 TextMesh Pro.....126
八、 富文本.....126
1. 标记格式.....126
2. 嵌套元素.....127
3. 标签参数.....127
九、 2D PSD 导入器.....127
1. 减少像素重叠.....128
2. 组合碰撞体.....129

哔哩哔哩次元游戏人生

3. 形状精灵烘焙.....130
4. 像素美化与锐化.....131
十、 粒子系统 Particle System.....132
1. 模拟阶段.....132
2. 渲染阶段.....133
十一、 脚本生命周期.....133
十二、 Animation.....135
1. 关于模型中的动画的位移变换问题.....135
十三、 自动布局组件.....135
1. Content Size Fitter.....135
十四、 后处理.....136
1. 为场景添加后处理效果的步骤.....136
2. 后处理效果应用顺序和效果组合.....139
3. 辉光-Bloom.....140
4. 色调映射-Tonemapping.....140
5. 白平衡-White Balance.....141
6. 胶片颗粒-Film Grain.....141
7. 景深-Depth of Field.....141
8. 帕尼尼投影-Panini Projection.....142
9. 镜头畸变-Lens Distortion.....142
10. 运动模糊-Motion Blur.....143
11. 色差-Chromatic Aberration.....143
12. 晕映-Vignette.....143
13. 颜色分级-Lift Gamma Gain.....144
14. 通道混合-Channel Mixer.....144
15. 颜色曲线- Color Curves.....145
16. 颜色调节-Color Adiustment.....145
17. 分离调色-Split Toning.....145
18. 阴影、中间调、高光-Shadow、Midtones、Highlights.....146
十五、 Unity 反射.....146
十六、 四元数.....147
十七、 其他知识.....147
1. 图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别？.....147
2. 使用 Unity3d 实现 2d 游戏，有几种方式？.....147
3. 向量的点乘、叉乘以及归一化的意义？.....147
十八、 .meta 文件.....147
1. .meta 文件的创建.....147
2. .meta 文件的作用.....147
3. 常见的错误操作.....147
Unity Script.....148
一、 GameObject.....148
1. 属性（Properties）.....148
2. 构造函数（Constructors）.....148
3. 公共方法.....148

哔哩哔哩尔元游戏人生

4. 静态方法.....149
二、 Transform.....149
1. 公共方法.....149
三、 Vector3.....150
1. 描述.....150
2. Static Properity-静态属性.....150
3. Static Methods-静态方法.....150
4. 向量归一化的区别.....151
四、 KeyCode 编码.....152
五、 数学函数库.....154
1. 绝对值-Mathf.Abs.....154
2. 反余弦-Mathf.Acos.....154
3. 近似-Mathf.Approximately.....154
4. 反正弦-Mathf.Asin.....154
5. 反正切-Mathf.Atan2.....154
6. 反正切-Mathf.Atan.....154
7. 向上最小整数- Mathf.CeilToInt.....154
8. 上限值-Mathf.Ceil.....154
9. 范围限制 0-1-Mathf.Clamp01.....154
10. 限制- Mathf.Clamp.....154
11. 最近的二次方- Mathf.ClosestPowerOfTwo.....155
12. 余弦- Mathf.Cos.....155
13. 度转弧度- Mathf.Deg2Rad.....155
14. 弧度转度- Mathf.Rad2Deg.....155
15. 增量角- Mathf.DeltaAngle.....155
16. 小正数- Mathf.Epsilon.....155
17. 指数- Mathf.Exp.....156
18. 向下最小整数- Mathf.FloorToInt.....156
19. 向下取整- Mathf.Floor.....156
20. 正无穷- Mathf.Infinity.....156
21. 反差值- Mathf.InverseLerp.....156
22. 是否为 2 的幂- Mathf.IsPowerOfTwo.....156
23. 插值角度- Mathf.LerpAngle.....156
24. 插值- Mathf.Lerp.....156
25. 基数为 10 的对数- Mathf.Log10.....156
26. 对数- Mathf.Log.....156
27. 最大值- Mathf.Max.....157
28. 最小值- Mathf.Min.....157
29. 移动角- Mathf.MoveTowardsAngle.....157
30. 移向- Mathf.MoveTowards.....157
31. 负无穷- Mathf.NegativeInfinity.....157
32. 下一个 2 的幂- Mathf.NextPowerOfTwo.....157
33. 乒乓- Mathf.PingPong.....157
34. 圆周率- Mathf.PI.....157

哔哩哔哩欢乐游戏人生

35. 次方- Mathf.Pow 157
36. 重复- Mathf.Repeat 157
37. 四舍五入到整数- Mathf.RoundToInt 158
38. 四舍五入- Mathf.Round 158
39. 有向- Mathf.Sign 158
40. 正弦- Mathf.Sin 158
41. 平方根- Mathf.Sqrt 158
42. 正切- Mathf.Tan 158
43. 平滑阻尼角度- Mathf.SmoothDampAngle 158
44. 平滑阻尼- Mathf.SmoothDamp 158
45. 平滑插值- Mathf.SmoothStep 158
六、 UGUI 事件接口 158
1. IPointEnterHandler 158
2. IPointExitHandler 159
3. IPointDownHandler 159
4. IPointUpHandler 159
5. IPointClickHandler 159
6. IDragHandler 159
7. IInitializePotentialDragHandler 159
8. IBeginDragHandler 159
9. IEndDragHanlder 159
10. IDropHandler 159
Unity 案例 160
一、 2D 角色移动 160
二、 3D 角色移动 160
三、 3D 相机控制 160
四、 输入管理-InputManager 160
五、 场景异步加载与过渡 161
六、 自定义 URP 后处理 161
七、 通用对象池 161
1. 概述 161
2. 理解 161
3. 优点 162
4. 缺点 162
5. 简单对象池 162
6. 对象池管理 167
八、 Unity 脚部 IK 171
1. 构建测试场景 171
2. 创建 Animator 并设置所需参数 171
3. 创建脚部 IK 173
4. 设置动画 IK 权重 175
5. 可能出现的问题 177
九、 射线检测应用 177
1. 添加 Layer 层 177

哔哩哔哩次元游戏人生

2. Layer 和 LayerMask 178
3. 使用单体射线检测 181
4. 旋转游戏对象 182
5. 可视化射线 182
6. 添加用于检测的对象 184
7. 使用多对象的射线检测 185
十、 射线检测的性能消耗 186
技巧 187
一、 修改 Unity 缓存位置 187
1. 修改 Unity GICache 187
2. 修改 Unity 的 Cache 缓存位置 187
3. 修改 Unity Asset Store 的默认下载路径 190
二、 优化 190
1. 全平台优化方案 190
2. 脚本性能优化 192
3. 物理性能优化 192
4. Shader 优化方案 192
三、 快捷键 196
1. Unity 196
2. Visual Studio 196
四、 脚本栏的扩展 197
1. 序列化[Serializable] 197
2. 强制序列化[SerializeField] 197
3. 属性标题[Header] 197
4. 属性提示[Tooltip] 197
5. 空行属性[Space] 197
6. 范围值属性[Range()] 197
7. 隐藏属性[HideInInSpector] 197
8. 输入域[TextArea] 197
9. [RequireComponent(typeof(xxx))] 197
10. [CreateAssetMenu(fileName = "xx", menuName = "xx", order = x)] 197
11. 永远执行-[ExecuteAlways] 197
12. 禁止多组件-[DisallowMultipleComponent] 197
13. 添加组件菜单-[AddComponentMenu] 197
14. 在编辑模式下执行-[ExecuteInEditMode] 197
五、 VisualStudio 198
六、 资源网站 198
七、 切片图集(Slice Atlas) 199
15. 下载 199
16. 切片类型 200
八、 未运行游戏的状态监视 ShaderGraph 201
九、 组件 Preset 201
十、 调整运行窗口颜色 202
十一、 修改鼠标样式和启动图像 203

哔哩哔哩次元游戏人生

1. 设置鼠标样式.....203
2. 设置启动图像.....204
十二、 图片跟随文本自适应.....206
十三、 修改 Unity 的脚本模板.....207
Unity Editor 210
一、 Editor 文件夹.....210
二、 Editor Default Resources 210
三、 Gizmos 文件夹 210
Unity Editor 实践.....211
Unity 工具 213
一、 Terrain Tool.....213
1. 快捷键.....213
二、 Cinemachine.....213
三、 Recorder.....213
1. 下载.....213
2. 打开 Recorder 界面 214
3. Recording Mode(记录模式) 214
4. Frame Rate 帧率属性 214
5. Capture-捕捉设置.....216
四、 Post Processing.....217
1. 开始后处理.....217
2. 泛光-Bloom.....218
设计模式.....219
一、 单例模式.....219
1. Unity 泛型单例.....219
二、 观察者模式.....220
数据结构.....222
一、 基本概率和术语.....222
1. 数据结构三要素.....222
数据结构习题.....223
一、 数组.....223
1. 给你一个整数数组 nums 。如果任一值在数组中出现至少两次，返回 true；如果数组中每个元素互不相同，返回 false。 223
2. 数组里有{1,2,3,4,5,6,7,8,9,10}，请随机打乱顺序，生成一个新的数组。 224
3. 给你一个整数数组 nums，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。（子数组是数组中的一个连续部分）。 .224
4. 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。 226
5. 给你两个按非递减顺序排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。请你合并 nums2 到 nums1 中，使合并后的数组同样按 非递减顺序排列。 227

哔哩哔哩次元游戏人生

二、 计算机组成原理.....227
1. 写出代码判断一个整数是不是 2 的 N 次方。.....227
2. 写出下面的程序的打印结果，并解释其目的。.....228
3. 写出下面的程序的打印结果，并解释其目的。.....229
三、 递归.....229
1. 斐波那契数（通常用 F(n)表示）形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：F(0) = 0，F(1) = 1，F(n) = F(n - 1)+F(n - 2)，其中 n > 1。现给定 n 请计算 F(n)。.....229
2. 已知 28657，46368 是斐波那契数列中的两个相邻数，输出这两个数之前的斐波那契数列。.....229
算法- Algorithm.....231
一、 洗牌算法.....231
1. Fisher–Yates shuffle.....231
2. Knuth-Durstenfeld Shuffle.....231
二、 Johnson–Trotter 全排列算法.....232
1. 算法描述.....232
2. 案例.....232
3. 优势.....234
4. 代码.....234
三、 排序算法.....241
1. 冒泡排序-Bubble Sort.....241
计算机组成原理.....242
一、 常用进位计数制.....242
1. 二进制.....242
2. 八进制.....242
3. 十进制.....242
4. 十六进制.....242
5. 推广.....242
二、 各种进制数之间的转换.....242
1. 二进制数转换为十六进制数.....242
2. 十六进制数转换为二进制数.....243
3. 二进制数转换为十进制数.....243
4. 十进制数转换为二进制数.....243
三、 数的机器码表示.....243
1. 机器数与真值.....243
四、 位运算符.....245
1. 与 - &.....245
2. 或 - |.....246
3. 异或 - ^.....246
4. 取反 - ~.....247
5. 左移 - <<.....247
6. 右移 - >>.....247
物理.....248
一、 牛顿第二定律.....248

哔哩哔哩次元游戏人生

二、 抛体运动.....248
线性代数.....250
一、 点和向量.....250
1. 具体来讲.....250
2. 自由向量.....250
二、 向量在轴上的投影.....250
三、 数量积、向量积.....250
1. 数量积（点乘）.....250
2. 向量积（叉乘）.....251
四、 向量的内积（点乘）.....252
1. 内积性质.....253
五、 向量的范数（长度）.....253
1. 性质.....253
六、 行列式.....253
1. 二阶行列式.....253
2. 三阶行列式.....253
3. n 阶行列式.....254
4. 性质.....254
5. 特殊的行列式.....254
七、 全排列和对换.....255
1. 全排列（排列）.....255
2. 逆序.....255
3. 对换.....255
八、 矩阵.....255
1. 矩阵的定义.....255
2. 矩阵的计算.....257
3. 矩阵的逆.....259
4. 矩阵的秩.....259
5. 缩放矩阵.....259
6. 平移矩阵.....260
7. 旋转矩阵.....260
8. 投影矩阵.....260
9. 二维错切变换.....260
10. 齐次坐标.....260
概率论与数理统计.....261
初等数学.....262
高等数学.....263
一、 导数.....263
数据库.....264
色彩构成.....265
一、 三原色.....265
1. 光学三原色-RGB.....265
2. 颜料三原色-CMKY.....265
3. 心理原色.....266

哔哩哔哩次元游戏人生

二、 色相-Hue.....266
三、 明度.....267
四、 纯度-饱和度.....268
1. 理论.....268
2. 饱和度的应用.....269
五、 色调.....270
Computer Graphics.....271
一、 贝塞尔曲线路径.....271
1. 贝塞尔曲线.....271
2. 一次 Bezier 曲线.....271
3. 二次 Bezier 曲线.....271
4. 三次 Bezier 曲线.....272
5. 高阶 Bezier 曲线.....272
二、 程序化网格生成.....273
1. 渲染物体.....273
2. UV.....273
3. 创建网格.....274
4. 生成法线.....275
5. 生成 UV 坐标.....275
Shader Graph.....276
一、 菲涅尔效果-Fresnel Effect Node.....276
1. 端口.....276
2. 生成的代码示例.....276
二、 乘节点-Multiply Node.....276
1. 端口.....276
2. 生成的代码示例.....276
三、 加法节点- Add Node.....277
1. 端口.....277
2. 生成的代码示例.....277
四、 除法-Divide Node.....277
1. 端口.....277
2. 生成的代码示例.....277
五、 正切- Tangent Node.....277
1. 端口.....277
2. 生成的代码示例.....277
六、 步节点-Step Node.....278
1. 端口.....278
2. 生成的代码示例.....278
七、 平滑步节点-Smoothstep Node.....278
1. 端口.....278
2. 生成的代码示例.....279
八、 极坐标-Polar Coordinates Node.....279
1. 端口.....279
2. 生成的代码示例.....279

哔哩哔哩次元游戏人生

九、 重新映射-Remap Node.....279
1. 端口.....280
2. 生成的代码示例.....280
十、 限制-Clamp Node.....280
1. 端口.....281
2. 生成的代码示例.....281
十一、 时间-Time Node.....281
1. 端口.....281
2. 生成的代码示例.....281
十二、 瓦片与偏移-Tiling And Offset Node.....281
1. 端口.....281
2. 生成的代码示例.....282
十三、 取反-Neagte Node.....282
1. 端口.....282
2. 生成的代码示例.....282
十四、 简单噪声-Simple Noise Node.....283
1. 端口.....283
2. 控件.....283
3. 生成的代码示例.....283
十五、 旋转-Rotate Node.....284
1. 端口.....284
2. 控件.....284
3. 生成的代码示例.....284
十六、 幂节点-Power Node.....285
1. 端口.....285
2. 生成的代码示例.....285
十七、 饱和节点-Saturate Node.....285
1. 端口.....285
2. 生成的代码示例.....285
十八、 线性插值-Lerp Node.....286
1. 端口.....286
2. 生成的代码示例.....286
十九、 反插值- Inverse Lerp Node.....287
1. 端口.....287
2. 生成的代码示例.....287
二十、 翻书-Flipbook Node.....288
1. 端口.....288
2. 控件.....288
3. 生成的代码示例.....289
二十一、 向下取整- Floor Node.....289
1. 端口.....289
2. 生成的代码示例.....289
二十二、 向上取整- Ceiling Node.....290
1. 端口.....290

哔哩哔哩次元游戏人生

2. 生成的代码示例.....290
二十三、 取余- Modulo Node.....290
1. 端口 0.....290
2. 生成的代码示例.....290
二十四、 场景深度-Scene Depth Node.....290
1. 端口.....290
2. 深度采样模式.....290
3. 生成的代码示例.....291
二十五、 One Minus Node.....291
1. 端口.....291
2. 生成的代码示例.....291
二十六、 位置-Position Node.....291
1. 端口.....291
2. 控件.....291
二十七、 绝对值-Absolute Node.....291
1. 端口.....292
2. 生成的代码示例.....292
二十八、 变换-Transform Node.....292
1. 端口.....292
2. 控件.....292
3. World and Absolute World.....292
4. 生成的代码示例.....292
二十九、 标准化-Normalize Node.....294
1. 端口.....294
2. 生成的代码示例.....294
三十、 长度-Length Node.....294
1. 端口.....294
2. 生成的代码示例.....294
三十一、 距离-Distance Node.....295
1. 端口.....295
2. 生成的代码示例.....295
三十二、 点乘-Dot Product Node.....295
1. 端口.....295
2. 生成的代码示例.....295
三十三、 小数-Fraction Node.....295
1. 端口.....295
2. 生成的代码示例.....295
三十四、 随机范围-Random Range Node.....295
1. 端口.....296
2. 生成的代码示例.....296
三十五、 比较-Comparison Node.....296
1. 端口.....296
2. 控件.....296
3. 生成的代码示例.....296

哔哩哔哩次元游戏人生

三十六、 指数-Exponential Node.....297
1. 端口.....297
2. 控件.....297
3. 生成的代码示例.....297
三十七、 符号-Sign Node.....298
1. 端口.....298
2. 生成的代码示例.....298
三十八、 快捷键.....298
ShaderGraph 实际案例.....299
一、 2D 简易描边效果-减法版.....299
1. 创建 Shader Graph.....299
2. 创建描边轮廓.....299
3. 修改描边轮廓颜色.....300
4. 得到最终成像.....301
二、 2D 描边效果-加法版.....301
1. 创建 Shader Graph.....301
2. 放大原图像遮罩.....302
3. 获得描边.....303
4. 修改描边颜色.....304
5. 得到最终成像.....304
6. 可能出现的问题.....305
三、 2D UI 技能冷却缓动.....305
1. 创建 Shader Graph.....305
2. 创建切分效果.....306
3. 重新映射.....307
4. 扩展-旋转 UV.....308
5. 扩展-极坐标.....308
6. 在代码中控制.....309
四、 2D 图像 RGB 分离.....311
1. 创建 Shader Graph.....311
2. 添加可外部访问的参数.....312
3. 添加节点.....312
五、 2D 全息效果.....314
1. 创建 Shader Graph.....314
2. 添加可外部访问的参数.....315
3. 创建带全息效果的图像.....316
4. 让全息纹理流动起来.....316
5. 扩展-调节全息贴图显示范围.....316
六、 2D 溶解效果.....317
1. 创建 Shader Graph.....317
2. 添加可外部访问的参数.....318
3. 创建溶解效果.....319
4. 添加溶解边缘颜色.....319
七、 2D 流光效果简易版.....320

哔哩哔哩次元游戏人生

1. 创建 Shader Graph.....320
2. 添加可外部访问的参数.....321
3. 显示含有流光效果的图像.....321
4. 让流光纹理流动起来.....322
5. 反向运动.....323
6. 旋转流光.....323
7. 可能出现的问题.....323
八、 2D 流光效果进阶版.....324
1. 创建 Shader Graph.....324
九、 转场过渡效果-简易版.....325
1. 创建 Shader Graph.....325
2. 添加可外部访问的参数.....326
3. 实现硬性过渡.....327
4. 将硬性过渡更改为柔性过渡.....327
十、 2D 激光效果.....328
十一、 2D 帧动画.....328
1. 创建 Shader Graph.....328
2. 添加主纹理.....329
3. 分割主纹理.....330
4. 让图集动起来.....330
5. 设置图集的最大可显示帧.....331
6. 调整帧动画速度.....333
7. 添加额外参数.....333
8. 可能存在的问题.....333
十二、 后处理速度线.....334
1. 创建 Shader Graph.....334
2. 制作速度线.....335
3. 让速度线流动起来.....336
4. 调整速度线的 Scale 和 Tiling.....336
5. 制作蒙版.....337
6. 最终输出.....337
7. 将节点转换为参数.....338
8. 后处理运用.....338
十三、 2D 像素效果.....339
1. 创建 Shader Graph.....339
2. 原理解释.....340
3. 修改图像对应的 UV 坐标.....340
4. 使用 Slider 控制像素化效果.....341
5. 设置是否应用像素化.....343
6. 自定义像素化行列（拓展）.....343
十四、 2D 植被飘动.....344
1. 创建 Shader Graph.....344
2. 原理.....345
3. 将主纹理输出到片元着色器.....345

哔哩哔哩次元游戏人生

4. 让顶点位置发生偏移.....345
5. 使用噪声调整偏移强度.....347
6. 对顶点应用噪声偏移.....347
7. 修改噪声视觉效果.....348
8. 创建影响蒙版.....348
十五、 2D 简单故障效果.....349
1. 创建 Shader Graph.....349
2. 创建色差效果.....350
3. 创建故障纹理.....352
4. 调节故障纹理密度.....354
5. 设置故障偏移方向.....354
6. 闪烁效果.....355
7. 完成最终效果.....355
十六、 UV 中心缩放.....356
1. 思路讲解.....356
2. 创建 ShaderGraph.....357
3. 虚假的中心缩放.....358
4. 真实的中心缩放.....360
十七、 2D 月食.....361
1. 思路.....361
2. 创建 ShaderGraph.....362
3. 使用圆的 SDF 去生成有向距离场纹理.....363
4. 生成蒙版.....365
5. 实现月食效果.....365
6. 可能出现的问题.....366
十八、 2D 元球-Metaball.....366
十九、 2DSDF 圆.....366
1. 圆心在原点.....366
2. 圆心不在原点上.....367
二十、 2DSDF 线段.....368
1. 位于区域一.....369
2. 位于区域二和三.....370
3. 计算端点向量和到目标点的向量.....371
4. 计算 r.....371
5. 计算aq向量.....372
6. 计算点距离线段的长度.....372
二十一、 2DSDF 矩形.....373
1. 矩形中心在原点.....373
2. 矩形中心不在原点.....375
3. 创建所需参数.....375
4. 计算向量ap1.....375
5. 计算目标点位于区域一、二、三时的模.....375
6. 计算区域四点的最短距离.....376
7. 排除目标点不位于区域四的情况.....376

哔哩哔哩次元游戏人生

8. 计算最终结果.....377
9. 缺点.....378
二十二、 2D 简易放大镜.....378
1. 核心思想.....378
2. 创建 ShaderGraph.....378
3. 添加参数.....379
4. 放大 UV.....379
5. 利用圆的 SDF 创建蒙版.....380
6. 得到最终效果.....381
7. 利用 C#脚本控制圆心坐标.....382
8. 其他方法.....383
二十三、 3D 菲尼尔边缘光.....383
1. 什么是菲尼尔效应.....383
2. 创建 LitShaderGraph.....388
3. 创建所需的基本纹理参数.....389
4. 输出基础纹理.....389
5. 使用菲涅尔节点.....391
6. 修改自发光颜色.....392
7. 重映射 FresnelValue.....392
二十四、 3D 全息投影.....392
1. 创建 Lit ShaderGraph.....392
2. 添加 PBR 纹理.....393
3. 修改自发光颜色.....394
4. 使用正切函数生成全息纹理.....394
5. 添加菲涅尔效果.....397
6. 将菲涅尔效果用于 Alpha.....401
二十五、 3D 法线膨胀描边效果.....401
二十六、 后处理描边.....401
二十七、 3D 积雪效果.....401
二十八、 3D 广告牌效果.....401
二十九、 (Billboard).....401
三十、 光学迷彩.....401
三十一、 置灰效果.....401
1. 创建 Shader Graph.....402
2. 方法一——取 RGB 的平均值.....403
3. 方法二——RGB 通道值乘对应的权重.....403
4. 方法三——使用数量积计算.....405
Shader Lab.....407
一、 Shader 概念.....407
1. 什么是 Shader.....407
2. Shader 和 Material（材质）的关系与区别.....407
3. Unity 中的 Shader.....407
二、 简单概念.....408
1. 顶点-Vertices.....408

哔哩哔哩次元游戏人生

2. 法线-Normals.....408
3. 切线-Tangents.....409
4. UV 坐标-UV coordinates.....409
三、 纹理.....410
1. 反照率颜色和透明度-Albedo.....410
2. 金属纹理-Metallic Texture.....411
3. 法线贴图（凹凸贴图）-Normal Map(Bump Map).....411
4. 高度图-Height Map.....419
5. 遮蔽贴图- Occlusion Map.....420
四、 渲染流水线.....421
1. 什么是渲染流水线.....421
2. 应用阶段(Application Stage).....421
3. 几何阶段(Geometry Stage).....422
4. 光栅化阶段(Rasterizer Stage).....422
5. 像素处理阶段(Pixel processing stage).....422
五、 渲染管线的类型.....423
1. 前向渲染-Forward rendering.....424
2. 延迟渲染- Deferred shading.....424
3. 应该使用什么渲染.....425
六、 CPU 和 GPU 之间的通信.....425
1. 把数据加载到显存中.....425
2. 设置渲染状态.....425
3. 调用 Draw Call.....426
七、 GPU 流水线.....426
1. 顶点数据.....427
2. 顶点着色器.....427
3. 曲面细分.....428
4. 几何着色器.....429
5. 裁剪.....429
6. 屏幕映射.....429
7. 三角形设置.....430
8. 三角形遍历.....431
9. 片元着色器.....431
10. 逐片元操作.....432
八、 什么是 Draw Call.....437
1. CPU 和 GPU 是如何实现并行工作的？.....437
2. 为什么 Draw Call 多了会影响帧率？.....438
3. 如何减少 Draw Call？.....438
九、 矩阵和坐标系.....439
十、 语法基础.....439
1. ShaderLab 的组织结构.....440
2. 详细解释.....453
十一、 限定语义.....468
十二、 顶点和片元着色器基础.....469

哔哩哔哩次元游戏人生

1. CGPROGRAM/ENDCG....469
2. 数据类型....470
3. 编译指令....472
4. 语义....476
5. 结构体....479
6. Unity 内置文件和变量....479
十三、 其他....486
1. 着色器和材质....486
2. 调试着色器....486
3. 添加 URP 兼容....488
4. 其他 URP 内容....495
十四、 内置数学函数....514
1. 绝对值-Abs....515
2. 上限值-Ceil....518
3. 限制-Clamp....520
4. 正弦和余弦函数-Sin and Cos....523
5. 正切-Tan....526
6. 指数- Exp, Exp2 and Pow....528
7. 向下取整-Floor....529
8. 分段和平滑分段-Step and Smoothstep....532
9. 长度- Length....534
10. 小数- Frac....536
11. 线性插值-Lerp....538
12. 最大和最小值-Min and Max....540
13. 时间和动画- Timing and animation....541
十五、 配置(Configure)输入和输出....542
十六、 Unity 中的基础光照....545
1. 我们是如何看见这个多彩世界的....545
2. 标准光照模型....547
3. 实现漫反射光照模型....550
4. 实现高光反射光照模型....550
十七、 基础纹理....550
1. 单张纹理....551
2. 凹凸映射....551
3. 渐变纹理....551
4. 遮罩纹理....551
十八、 透明效果....551
1. 透明度测试....551
2. 透明度混合....552
3. 渲染顺序的重要性....552
4. Unity Shader 的渲染顺序....555
5. 使用透明度测试....556
6. 使用透明度混合....556
7. 开启深度写入的半透明效果....556

哔哩哔哩次元游戏人生

十九、 有向距离函数-SDF-Sign Distance Function 556
1. 圆-Circle.....556
2. 线段-LineSegment 557
3. 矩形-Rectangle 561
4. 圆角矩形- Rounded Rectangle 562
5. 菱形-Rhombus 566
6. 梯形.....569
7. 月食.....569
二十、 伪随机噪声-Pseudo Random Noise 573
1. Hashing 573
二十一、 Unity 中的渲染优化技术.....575
1. 移动平台的特点.....575
2. 影响性能的因素.....576
3. 减少 DrawCall 数目 577
4. 减少需要处理的顶点数目.....582
5. 减少需要处理的片元数目.....583
6. 节省带宽.....585
7. 减少计算复杂度.....585
二十二、 额外拓展.....591
1. 什么是伽马校正.....591
2. HDR 595
3. MatCap.....595
Visual Effect Graph.....603
一、 篝火.....603
PhotoShop 604
一、 去除水印.....604
二、 快捷键.....604

哔哩哔哩次元游戏人生

C、C#、C++

一、 数据类型

数据类型	32 位字节数	64 位字节数
bool	1	1
Char	1	1
Unsigned char	1	1
Short	2	2
Unsigned short	2	2
Int	4	4
Unsigned int	4	4
Long	4	8
Unsigned long	4	8
Long long	8	8
Float	4	4
Double	8	8
Long double	8	8
*	4	8

二、 运算符优先级和结合性

优先级	运算符	结合方向	名称或含义
1	()	从左到右	圆括号
	[]		数组下标
	.		结构体成员
	->		用指针访问类或结构体成员
	::		类成员
2	+，-	从右到左	正号和负号
	++，--		自增和自减
	!		逻辑非
	~		按位取反
	(t)		类型转换
	*		由地址求内容
	&		求变量的地址
	Sizeof		求某类型变量的长度
3	*，/，%	从左到右	乘、除、求余
4	+，-		加和减
5	<<，>>		左移和右移
6	<，>，<=，>=		关系运算（比较）
7	==，！=		等于和不等于
8	&		按位与
9	^		按位异或

哔哩哔哩次元游戏人生

10	|		按位或
11	&&		逻辑与
12	||		逻辑或
13	?::		条件运算
14	=		赋值运算
	+=，-=，*=		复合赋值运算
	/=，%=，>>=		
	<<=，&=，^=		
	|=		
15	,		顺序求值运算

三、 C# 结构体和类-Struct and Class

1. Struct

在 C#中，结构体是值类型数据结构。它使得一个单一变量可以存储各种数据类型的相关数据，struct 关键字用于创建结构体。

1） 定义结构体

为了定义一个结构体，我们必须使用 struct 语句。struct 语句为程序定义了一个带有多个成员的新的数据类型。

例如，我们可以按照如下的方式声明 Book 结构体。

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public int book_id;
};
```

2） C# 结构体的特点

我们已经有了一个名为 Books 的结构体。C#中的结构体与传统的 C 或 C++ 中的结构体不同。C# 中的结构体有以下特点：

① 结构体可带有方法、字段、索引、属性、运算符方法和事件。

② 结构体可定义构造函数，但不能定义析构函数。但是，您不能为结构体定义无参构造函数。无参构造函数(默认)是自动定义的，且不能被改变。

③ 与类不同，结构体不能继承其他的结构体或类。

④ 结构体不能作为其他结构体或类的基础结构。

⑤ 结构体可实现一个或多个接口。

⑥ 结构体成员不能指定为 abstract、virtual 或 protected（成员不能添加访问修饰符）。

⑦ 当使用 New 操作符创建一个结构体对象时，会调用适当的构造函数来创建结构。与类不同，结构体可以不使用 New 操作符即可被实例化。

⑧ 如果不使用 New 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

2. Class

当定义了一个类时，就是定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，但它定义了类的名称意味着什么，也就是说，类的对象由什么组成及在这个对象上可

哔哩哔哩次元游戏人生

执行什么操作。对象是类的实例，构成类的方法和变量的称为类的成员。

1） 类的定义

类的定义是以关键字 class 开始，后跟类的名称。类的主体，包含在一对花括号内。下面是类定义的一般形式。

```
class   class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

访问标识符 <access specifier> 指定了对类及其成员的访问规则。如果没有指定，则使用默认的访问标识符。类的默认访问标识符是 internal，成员的默认访问标识符是 private。

数据类型 <data type> 指定了变量的类型，返回类型 <return type> 指定了返回的方法返回的数据类型。

如果要访问类的成员，你要使用点（.）运算符。点运算符链接了对象的名称和成员的名称。

2） 成员函数和封装

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样。作为类的一个成员，它能在类的任何对象上操作，且能访问该对象的类的所有成员。

成员变量是对象的属性（从设计角度），且它们保持私有来实现封装。这些变量只能使用公共成员函数来访问。

```
using System;
namespace BoxApplication
{
    class Box
    {
        private double length;    // 长度
```

哔哩哔哩次元游戏人生

```
    private double breadth;   //  宽度
    private double height;    //  高度
    public void setLength( double len )
    {
        length = len;
    }

    public void setBreadth( double bre )
    {
        breadth = bre;
    }

    public void setHeight( double hei )
    {
        height = hei;
    }
    public double getVolume()
    {
        return length * breadth * height;
    }
}
class Boxtester
{
    static void Main(string[] args)
    {
        Box Box1 = new Box();          //  声明 Box1，类型为 Box
        Box Box2 = new Box();                  //  声明 Box2，类型为 Box
        double volume;                                  //  体积


        // Box1  详述
        Box1.setLength(6.0);
        Box1.setBreadth(7.0);
        Box1.setHeight(5.0);

        // Box2  详述
        Box2.setLength(12.0);
        Box2.setBreadth(13.0);
        Box2.setHeight(10.0);

        // Box1  的体积
        volume = Box1.getVolume();
        Console.WriteLine("Box1  的体积：   {0}" ,volume);
```

哔哩哔哩次元游戏人生

```
                // Box2 的体积
                volume = Box2.getVolume();
                Console.WriteLine("Box2 的体积：  {0}", volume);

                Console.ReadKey();
            }
        }
    }
```

3） C# 类的静态成员

我们可以使用 static 关键字把类成员定义为静态的。当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，只会有一个该静态成员的副本。

关键字 static 意味着类中只有一个该成员的实例。静态变量用于定义常量，因为它们的值可以通过直接调用类而不需要创建类的实例来获取。静态变量可在成员函数或类的定义外部进行初始化，也可以在类的定义内部初始化静态变量。

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;
        public void count()
        {
            num++;
        }
        public int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();
            s1.count();
            s1.count();
            s1.count();
            s2.count();
            s2.count();
            s2.count();
            Console.WriteLine("s1 的变量 num：  {0}", s1.getNum());
            Console.WriteLine("s2 的变量 num：  {0}", s2.getNum());
            Console.ReadKey();
```

哔哩哔哩次元游戏人生

```
            }
        }
    }
```

我们也可以把一个成员函数声明为 static。这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在。

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s = new StaticVar();
            s.count();
            s.count();
            s.count();
            Console.WriteLine("变量 num:  {0}", StaticVar.getNum());
            Console.ReadKey();
        }
    }
}
```

将类成员函数声明为 public static，无需实例化即可调用类成员函数。反之，如果不声明为 static，即使和 Main 方法从属于同一个类，也必须经过实例化

3. 类与结构体的选择

首先明确，类的对象是存储在堆空间中，结构体存储在栈中。堆空间大，但访问速度较慢，栈空间小，访问速度相对更快。故而，当我们描述一个轻量级对象的时候，结构体可提高效率，成本更低。当然，这也得从需求出发，假如我们在传值的时候希望传递的是对象的引用地址而不是对象的拷贝，就应该使用类了。

4. 类与结构体的区别

① 结构体中声明的字段无法赋予初值，但是类可以。

② 结构体的构造函数中，必须为结构体所有字段赋值，类的构造函数无此限制。

③ 结构体是值类型，它在栈中分配空间；而类是引用类型，它在堆中分配空间，栈

哔哩哔哩次元游戏人生

中保存的只是引用。

结构体类型直接存储成员数据，而其他类的数据位于堆中，位于栈中的变量保存的是指向堆中数据对象的引用。由于结构体是值类型，并且直接存储数据，因此在一个对象的主要成员为数据且数据量不大的情况下，使用结构体会带来更好的性能。因为结构体是值类型，因此在为结构分配内存，或者当结构体超出了作用域被删除时，性能会非常好，因为他们将内联或者保存在堆栈中。当把一个结构体类型的变量赋值给另一个结构体时，对性能的影响取决于结构的大小，如果结构的数据成员非常多而且复杂，就会造成损失。

四、 构造函数

① 调用构造函数，可以在一个类通过 new 关键字实例化对象时，对这个新实例化的对象进行初始化。

② 不带参数的构造函数称为“默认构造函数”。不管在什么时候，只要是使用 new 运算符去实例化对象，并且不为其提供任何形式的参数，系统就会去调用默认的构造函数。如果类不是 static 的，即这个类不是静态的，C# 编译器将会为不具备构造函数的类提供一个公共的默认构造函数，以便该类可以实例化。

③ 要想阻止一个类被实例化，可以通过将构造函数设置为私有构造函数来实现。

④ 结构体的构造函数与类的构造函数类似却又不完全相同，结构体 struct 并不能包含显式的默认构造函数，这是由于编译器将会自动的提供一个构造函数。编译器提供的构造函数会将结构体中的每个字段都初始化为默认值。只有当 struct 结构体用 new 关键字进行实例化时，才会调用编译器提供的这个默认构造函数。

五、 虚函数

① 多态性是面向对象程序设计的重要特征之一。在 C++中，编译时的多态性主要是通过函数重载和运算符重载实现的。运行时的多态性主要是通过虚函数实现的。

② 虚函数是重载的另一种表现形式，它提供了一种更为灵活的运行时的多态性机制。虚函数允许函数调用与函数体之间的联系在运行时才建立，也就是在运行时才决定进行何种操作，即动态编译。

③ 虚函数首先是基类中的成员函数，但这个成员函数前面缀上了关键字 Virtual，并在派生类中被重载。在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。虚函数在派生类中重新定义时，其函数原型，包括其返回值类型、函数名、参数类型的顺序，都必须与基类中的原型完全一致。

5. 虚函数

6. 纯虚函数

Virtual 函数类型 函数名(参数列表）= 0;

纯虚函数没有函数体，不具有函数的功能，不能被调用。

7. 接口(Interface)和抽象类的区别

抽象类表示该类中可能已经有一些方法的具体定义，但接口就往往只能定义各个方法的界面 ，不能定义具体的实现代码在成员方法中。类是子类用来继承的，当父类已经有实际功能的方法时该方法在子类中可以不必实现，直接引用父类的方法，子类也可以重写该父类的方法。实现接口的时候必须要实现接口中所有的方法，不能遗漏任何一个。

哇哩哩哩哩源代源代人生

六、 C#中四种访问修饰符

1. 属性修饰符

Serializable 按值将对象封送到远程服务器。

STATread 是单线程套间的意思，是一种线程模型。

MATAThread 是多线程套间的意思，也是一种线程模型。

2. 存取修饰符

public：存取不受限制。

private：只有包含该成员的类可以存取。

internal：只有当前命名空间可以存取。只能在包含该类的程序集中访问该类。

protected：只有包含该成员的类以及派生类可以存取。

protected internal：protected + internal

3. 类修饰符

abstract：抽象类。指示一个类只能作为其它类的基类。

sealed：密封类。指示一个类不能被继承。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。

4. 成员修饰符

abstract：指示该方法或属性没有实现。

sealed：密封方法。可以防止在派生类中对该方法的 override（重载）。不是类的每个成员方法都可以作为密封方法密封方法，必须对基类的虚方法进行重载，提供具体的实现方法。所以，在方法的声明中，sealed 修饰符总是和 override 修饰符同时使用。

delegate：委托。用来定义一个函数指针。C#中的事件驱动是基于 delegate + event 的。

const：指定该成员的值只读不允许修改。

event：声明一个事件。

extern：指示方法在外部实现。

override：重写。对由基类继承成员的新实现。

readonly：指示一个域只能在声明时以及相同类的内部被赋值。

static：指示一个成员属于类型本身，而不是属于特定的对象。即在定义后可不经实例化，就可使用。

virtual：指示一个方法或存取器的实现可以在继承类中被覆盖。

new：在派生类中隐藏指定的基类成员，从而实现重写的功能。 若要隐藏继承类的成员，请使用相同名称在派生类中声明该成员，并用 new 修饰符修饰它。

七、 ref 和 out 关键字

ref、out 都是引用传递，而引用传递是传递的这个参数的实际地址，因此在引用传递后的参数使用都会改变参数原有的数值。引用传递都需要显示传递到方法，即在方法的参数前加上相对应的关键字。

1. ref

ref 关键字有进有出，不仅能将参数传递进去还能将参数重新传递出来，通俗的说就是在函数里对变量的改变会在函数结束时改变变量的值，并将被改变的变量值重新传递出去，因此需要在传递进入函数前初始化。

2. out

而 out 关键字是只出不进的，将参数传递进去时值是无效的，out 关键字会要求把参数的值清空，然后在函数内部对其重新赋值，所以无法将一个值从 out 传递出去。

虽然使用 ref 和 out 都能返回多个值，但是必须确保函数方法的定义及调用都是显示使用的 ref 和 out 关键字。

3. ref 和 out 的相同点

① 都能返回多个返回值。

② 若要使用 ref 和 out 参数，则方法定义和调用方法都必须显示使用 ref 和 out 关键字。

4. ref 和 out 的不同点

① ref 指定的参数在函数调用时必须初始化，不能为空的引用。Out 指定的参数在函数调用时可以不初始化。

② Out 指定的参数在进入函数时会清空自己，必须在函数内部对其赋值。Ref 指定的参数不需要内部赋值。

八、 分部类和方法

拆分一个类、一个结构、一个接口或一个方法的定义到两个或更多的文件中是可能的。 每个源文件包含类型或方法定义的一部分，编译应用程序时将把所有部分组合起来。

1. 分部类

在以下几种情况下需要拆分类定义：

① 处理大型项目时，使一个类分布于多个独立文件中可以让多位程序员同时对该类进行处理。

② 当使用自动生成的源文件时，你可以添加代码而不需要重新创建源文件。 Visual Studio 在创建 Windows 窗体、Web 服务包装器代码等时会使用这种方法。 你可以创建使用这些类的代码，这样就不需要修改由 Visual Studio 生成的文件。

③ 使用源生成器在类中生成附加功能时。

若要拆分类定义，请使用 partial 关键字修饰符，如下所示：

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

partial 关键字指示可在命名空间中定义该类、结构或接口的其他部分。所有部分都必须使用 partial 关键字。在编译时，各个部分都必须可用来形成最终的类型。各个部分必须具有相同的可访问性，如 public、private 等。

如果将任意部分声明为抽象的，则整个类型都被视为抽象的。 如果将任意部分声明为密封的，则整个类型都被视为密封的。 如果任意部分声明基类型，则整个类型都将继承该类。

指定基类的所有部分必须一致，但忽略基类的部分仍继承该基类型。 各个部分可以指定不同的基接口，最终类型将实现所有分部声明所列出的全部接口。 在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。 最终类型是所有部分在编译时的组合。

partial 修饰符不可用于委托或枚举声明中。

2. 分部方法

分部类或结构可以包含分部方法。 类的一个部分包含方法的签名。 可以在同一部分或另一部分中定义实现。 如果未提供该实现，则会在编译时删除方法以及对方法的所有调用。 根据方法签名，可能需要实现。 在以下情况下，不需要使用分部方法即可实现：

① 没有任何可访问性修饰符（包括默认的 专用）。

② 返回 void。

③ 没有任何输出参数。

④ 没有以下任何修饰符：virtual、override、sealed、new 或 extern。

任何不符合所有这些限制的方法（例如 public virtual partial void 方法）都必须提供实现。 此实现可以由源生成器提供。

九、 面向对象的三大特点

1. 继承

提高代码重用度，增强软件可维护性的重要手段，符合开闭原则。继承最主要的作用就是把子类的公共属性集合起来，便与共同管理，使用起来也更加方便。你既然使用了继承，那代表着你认同子类都有一些共同的特性，所以你把这些共同的特性提取出来设置为父类。继承的传递性：传递机制 a->b; b->c; c 具有 a 的特性 。

4） 继承的单根性

在 C#中一个类只能继承一个类，不能有多个父类。

2. 封装

封装是将数据和行为相结合，通过行为约束代码修改数据的程度，增强数据的安全性，属性是 C#封装实现的最好体现。就是将一些复杂的逻辑经过包装之后给别人使用就很方便，别人不需要了解里面是如何实现的，只要传入所需要的参数就可以得到想要的结果。封装的意义在于保护或者防止代码（数据）被我们无意中破坏。

3. 多态性

多态性是指同名的方法在不同环境下，自适应的反应出不同得表现，是方法动态展示的重要手段。多态就是一个对象多种状态，子类对象可以赋值给父类型的变量。

十、 面向对象程序设计方法的主要优点

可提高程序的重用性。

可控制程序的复杂性。

可改善程序的可维护性。

能够更好地支持大型程序设计。

哔哩哔哩次元游戏人生

增强了计算机处理信息的范围。

十一、 C#索引

十二、 运算符重载

十三、 值类型和引用类型

十四、 值类型和引用类型

十五、 C#索引器

十六、 C#反射-Reflection

十七、 C#常用 String 方法

哔哩哔哩次元游戏人生

Lua 语言

一、 Lua 基本语法

1. 单行注释

在 Lua 中使用两个减号来表示单行注释，即” --”。

PS：--这是一个单行注释

2. 多行注释

在 Lua 中使用减号与大括号的组合来实现，在使用多行注释时需要注意，多行注释是一个有始有终的组合” —[[]]--”，即有开始就一定有结束。（就像 HTML 的 一样）

PS：

--[[
这
是
一
个
多
行
注
释
--]]

3. 标示符

Lua 标示符用于定义一个变量，函数可以获取其他用户定义的项。标示符以一个字母 A 到 Z 或 a 到 z 或下划线_开头后加上 0 个或多个字母，下划线，数字（0 到 9）。

最好不要使用下划线加大写字母的标示符，因为 Lua 的保留字也是这样的。

Lua 不允许使用特殊字符如 @, $, 和 % 来定义标示符。 Lua 是一个区分大小写的编程语言。因此在 Lua 中 Runoob 与 runoob 是两个不同的标示符。

以下列出了一些正确的标示符：

mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal

4. 关键字

下面列出了 Lua 的保留关键字。保留关键字不能作为常量或变量或其他用户自定义标示符。

and	break	do	Else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while	goto		

5. 全局变量

Lua 在默认的情况下，其变量总被认为是全局的（只要不加 local 关键字修饰的变量，都是全局变量）。

哔哩哔哩次元游戏人生

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil。

PS：print(b)　结果为　nil

PS：

b=10　--这里定义了一个全局变量 b

print(b)　--输出所定义的全局变量 b

结果为 10

如果想删除一个全局变量，只需要将变量赋值为 nil。

PS：

b = nil

print(b)

结果为 nil

当我们将全局变量赋值为 nil 后，这个全局变量就好像从没被使用过一样。换句话说，当且仅当一个变量不等于 nil 时，这个变量即存在。

二、　数据类型

Lua 是动态类型语言，其变量并不需要定义数据类型，只需要为变量赋值。值可以存储在变量中，作为参数传递或结果返回。

Lua 语言中具有 8 个基本的数据类型，它们分别为：nil、boolean、number、string、userdata、function、thread 和 table。

数据类型	描述
nil	只有值 nil 属于该类，表示一个无效值（在条件表达式中相当于 false）。
boolean	包含两个值：false 和 true。
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
userdata	表示任意存储在变量中的 C 数据结构
function	由 C 或 Lua 编写的函数
thread	表示执行的独立线路，用于执行协同程序
table	Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单的构造表达式是{}，用来创建一个空表。

我们可以使用 type 函数测试给定变量或者值的类型：

PS：

```
print(type("Hello world"))      --> string 类型
print(type(10.4*3))             --> number 类型
print(type(print))              --> function 类型
print(type(type))               --> function 类型
print(type(true))               --> boolean 类型
print(type(nil))                --> nil 类型
print(type(type(X)))            --> string 类型
```

哔哩哔哩次元游戏人生

1. 空-nil

nil 类型表示没有任何有效值，它只有一个值就是 nil，如果打印一个没有赋值的变量，便会输出一个 nil 值：

PS：

```
print(type(a))
```

输出结果为 nil

对于全局变量和 table，nil 还有一个"删除"作用，给全局变量或者 table 表里的变量赋一个 nil 值，等同于把它们删掉。

PS：

```
tab1 = { key1 = "value_1", key2 = "value_2", "value_3" }
for key, element in pairs(tab1) do
    print(key .. " - " .. element)
end

tab1.key1 = nil
for key, element in pairs(tab1) do
    print(key .. " - " .. element)
end
```

1） nil 作比较时应该加上双引号"

PS：

```
type(x)     -- 输出 nil
type(x) == nil    -- 输出 false
type(x) == "nil"     --输出 true
```

type(x) == nil 的结果为 false 的原因是 type(x)实质是返回的 "nil" 字符串，是一个 string 类型。

print(type(type(x)))输出的结果为 string

2. 布尔-boolean

boolean 类型只有两个可选值：true（真）和 false（假），Lua 把 false 和 nil 看作是 false，其他的都为 true，数字 0 也是 true。

PS：

```
print(type(true))
print(type(false))
print(type(nil))

if false or nil
then
    print("至少有一个是 true")
else
    print("false 和 nil 都为 false")
end

if 0
then
```

哔哩哔哩次元游戏人生

```
    print("数字 0 是 true")
else
    print("数字 0 为 false")
end
```

上面的结果为

```
boolean
boolean
nil
false 和 nil 都为 false
数字 0 是 true
```

3. 数字-number

Lua 默认只有一种 number 类型，即 double（双精度）类型（默认类型可以修改 luaconf.h 里的定义），以下几种写法都被看作是 number 类型。

PS：

```
print(type(2))
print(type(2.2))
print(type(0.2))
print(type(2e+1))
print(type(0.2e-1))
print(type(7.8263692594256e-06))
```

4. 字符串-string

字符串由一对双引号或单引号来表示。

PS：

```
string1 = "this is string1"
string2 = 'this is string2'
```

也可以用两个方括号 "[[]]" 来表示"一块"字符串。

PS：

```
html = [[
<html>
<head></head>
<body>
    <a href="https://space.bilibili.com/7399222?spm_id_from=333.1007.0.0">教程</a>
</body>
</html>
]]
print(html)
```

上面的内容输出的结果就为：

```
<html>
<head></head>
<body>
    <a href="https://space.bilibili.com/7399222?spm_id_from=333.1007.0.0">教程</a>
</body>
</html>
```

哔哩哔哩次元游戏人生

1） 对数字字符串进行算术操作

在对一个数字字符串上进行算术操作时，Lua 会尝试将这个数字字符串转换成一个数字。

PS：

```
print("2" + 6)     --结果为 8.0
print("2" + "6")   --结果为 8.0
print("2 + 6")   --结果为 2 + 6，因为这就是一个完整的字符串
print("-2e2" * "6")   --结果为-1200.0
```

2） 字符串连接

要将字符串连接起来，需要使用的是“..”。

PS：print("a" .. 'b') --结果为 ab

print(157 .. 428) --结果为 157428

3） 计算字符串长度

使用#来计算字符串的长度，#需要放在字符串的前面。

PS：

```
len = "www.runoob.com"
print(#len)
print(#"www.runoob.com")
```

上面的输出结果为 14，即字符的个数。

5. 表-table

在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表。

1） 创建一个空的 table

```
local tbl1 = {}
```

2） 直接初始表

```
local tbl2 = {"apple", "pear", "orange", "grape"}
```

3） 关联数组

Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串，就相当于数据字典(dictionary)。

PS：

```
a = {}
a["key"] = "value"
key = 10
a[key] = 22
a[key] = a[key] + 11
for k, value in pairs(a)
do
    print(k .. " : " .. value)
end
```

上面的结果为 key : value 10 : 33

不同于其他语言的数组，会把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。

哔哩哔哩次元游戏人生

```
local tbl = {"apple", "pear", "orange", "grape"}
for key, value in pairs(tbl)
do
    print("Key", key)
end
```

输出结果为

```
Key     1
Key     2
Key     3
Key     4
```

table 不会固定长度大小，有新数据添加时 table 长度会自动增长，没初始的 table 都是 nil。

PS：

```
a3 = {}
for i = 1, 10
do
    a3[i] = i
end
a3["key"] = "val"
print(a3["key"])
print(a3["none"])
```

从上面的代码可以知道，字符串"key"在 table 中所对应的值为"val"，而字符串"none"在 table 中并没有对应的值，因此它的输出结果为 nil。

6. 函数-function

在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，函数可以存放在变量里。

PS：

```
function factorial1(n)
    if n == 0
    then
        return 1
    else
        return n * factorial1(n - 1)   --递归，最终结果为 n!（数 n 的阶乘）
    end
end
print(factorial1(5))
factorial2 = factorial1   --用全局变量 factorial2 去存储 factorial1 函数
print(factorial2(5))
```

上面代码的输出都是为 120。

1） 匿名函数

函数(function)可以以匿名函数（anonymous function）的方式通过参数传递。

PS：

```
function testFun(tab,fun)
    for key ,value in pairs(tab)
```


哔哩哔哩次元游戏人生

```
    do
        print(fun(key,value));
    end
end


tab={key1="val1",key2="val2"};
testFun(tab,
        function(key,value)   --这是一个匿名函数
            return key.."="..value;     -- ".."表示连接
        end
);
```

上面的执行结果为

key1 = val1

key2 = val2

7. 线程-thread

在 Lua 里，最主要的线程是协同程序（coroutine）。它跟线程（thread）差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分东西。

1） 线程跟协程的区别

线程可以同时多个运行，而协程任意时刻只能运行一个，并且处于运行状态的协程只有被挂起（suspend）时才会暂停。

8. 自定义类型-userdata

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++语言库所创建的类型，可以将任意 C/C++的任意数据类型的数据（通常是 struct 和指针）存储到 Lua 变量中调用。

三、 Lua 的变量、赋值、索引

1. 变量

变量在使用前，需要在代码中进行声明，即创建该变量。

编译程序执行代码之前，编译器需要知道如何给语句变量开辟存储区，用于存储变量的值。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

Lua 中的变量默认全是全局变量，哪怕是语句块或是函数里，除非用 local 显式声明为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束。

变量的默认值均为 nil。

PS：

```
a = 5                   -- 这是一个默认的全局变量
local b = 5             -- 这是我们显示声明的局部变量

function joke()
    c = 5               -- 这是全局变量
    local d = 6         -- 这是函数中的局部变量
```

```
end

joke()
print(c,d)              -- 输出结果为 5 nil，因为变量 d 是一个局部变量

do
    local a = 6         -- 局部变量
    b = 6               -- 对局部变量重新赋值
    print(a,b);         -- 这里的 a 使用的是这个域中的局部变量，输出结果为 6 6
end

print(a,b)          --这里使用的 a 是最初声明的全局变量 a ，输出结果为 5 6
```

2. 赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法。

PS：

```
a = "hello" .. "world"
t.n = t.n + 1
```

Lua 可对多个变量同时赋值，变量列表和值列表的各个元素用逗号分开，赋值语句右边的值会依次赋给左边的变量。

PS： a, b = 10, 2*x -- 这条语句相当于 a=10; b=2*x

遇到赋值语句 Lua 会先计算右边所有的值然后再执行赋值操作，利用这一特性我们就可以交换变量的值。

PS：

```
x,y = 1,2    --为全局变量 x 和 y 赋值
x, y = y, x     --利用 lua 赋值特性来交换 x 和 y 的值
print(x,y)
```

在上面的代码中，我们利用 lua 的赋值特性来交换了 x 和 y 的值，因此最后的输出结果为 2 1。

当变量个数和值的个数不一致时，Lua 会一直以变量个数为基础采取不同的策略。

当 变量个数 > 值的个数　　按变量个数补足 nil

当 变量个数 < 值的个数　　多余的值会被忽略

PS：

```
a, b, c = 0, 1--因为值的数量小于参数的个数，因此后续的参数值用 nil 补足
print(a,b,c) --输出结果为 0    1    nil

a, b = a+1, b+1, b+2 --因为值的数量大于参数的个数，多余的值被忽略
print(a,b)    --输出结果为 1     2

a, b, c = 0--因为值的数量小于参数的个数，因此后续的参数值用 nil 补足
print(a,b,c)    --输出结果为 0     nil     nil
```

注意：如果要对多个变量赋值必须依次对每个变量赋值。

多值赋值经常用来交换变量，或将函数调用返回给变量。

3. 索引

对 table 的索引使用方括号[]。Lua 也提供了 . 操作。

哔哩哔哩次元游戏人生

```
t[i]
t.i                    -- 当索引为字符串类型时的一种简化写法
gettable_event(t,i) -- 采用索引访问本质上是一个类似这样的函数调用
```

PS：

```
site = {}   --创建一个空的 table，其相当于一个数据字典
site["key"] = "this" --为这个表赋值
print(site["key"])
print(site.key)
```

上面代码的最终输出都是 this，只是采用的索引方式不同。

四、 循环

很多情况下我们需要做一些有规律性的重复操作，因此在程序中就需要重复执行某些语句。一组被重复执行的语句称之为循环体，终止条件决定能否继续重复。

循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。循环语句是由循环体及循环的终止条件两部分组成的。

1. 循环类型

1） while

Lua 编程语言中 while 循环语句在判断条件为 true 时会重复执行循环体语句，这和 C、C#、C++中的 while 循环一致。

```
while(condition)
do
    statements
end
```

statements(循环体语句) 可以是一条或多条语句，condition(条件) 可以是任意表达式，在 condition(条件) 为 true 时执行循环体语句。

PS：

```
a=10
while( a < 20 )
do
    print("a 的值为:", a)
    a = a+1
end
```

2） for

Lua 编程语言中 for 循环语句可以重复执行指定语句，重复次数可在 for 语句中控制。

a. 数值 for 循环

```
for var=exp1,exp2,exp3
do
    <执行体>
end
```

上述表示 var 从 exp1 变化到 exp2，每次变化以 exp3 为步长递增 var，并执行一次 "执行体"。exp3 是可选的，如果不指定，默认为 1。

PS：

哔哩哔哩次元游戏人生

```
for i=1,3--没有设置可选参数 3，将默认每次执行完成后加 1
do
    print(i)
end

for i=10,1,-1--设置了可选参数，每次执行完成后数值减 1
do
    print(i)
end
```

b. 泛型 for 循环

泛型 for 循环通过一个迭代器函数来遍历所有值，类似 java、C#中的 foreach 语句。

Lua 编程语言中泛型 for 循环语法格式为：

```
a = {"one", "two", "three"}--这是一个直接初始表
for index, value in ipairs(a)
do
    print(index, value)
end
```

index 是数组索引值，value 是对应索引的数组元素值。ipairs 是 Lua 提供的一个迭代器函数，用来迭代数组。

PS：

```
days = {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}
for index,value in ipairs(days)
do
    print(value)
end
```

上面代码，通过迭代器函数遍历数组的索引，来输出它所对应的 value 值。

3） repeat…until

Lua 编程语言中 repeat...until 循环语句不同于 for 和 while 循环，for 和 while 循环的条件语句在当前循环执行开始时判断，而 repeat...until 循环的条件语句在当前循环结束后判断。（repeat...until 相当于 do while）

Lua 编程语言中 repeat...until 循环语法格式:

```
repeat
   statements
until( condition )
```

我们注意到循环条件判断语句（condition）在循环体末尾部分，所以在条件进行判断前循环体都会执行一次。如果条件判断语句（condition）为 false，循环会重新开始执行，直到条件判断语句（condition）为 true 才会停止执行。

PS：

```
a = 10--定义一个全局变量
repeat
    print("a 的值为:", a)
```

哔哩哔哩次元游戏人生

```
    a = a + 1
until( a > 15 )--当 a>15 时，跳出循环
```

4） 循环嵌套

Lua 编程语言中允许循环中嵌入循环。

a. for 循环嵌套语法格式

```
for init,max/min value, increment
do
    for init,max/min value, increment
    do
    statements
    end
statements
end
```

b. while 循环嵌套语法格式

```
while(condition)--condition 为 true 执行循环体
do
    while(condition)--condition 为 true 执行循环体
    do
        statements--循环体
    end
    statements--循环体
end
```

c. repeat...until 循环嵌套语法格式

```
repeat
    statements--循环体
    repeat
        statements--循环体
    until( condition )
until( condition )--当 condition 为 false 时执行循环体，为 true 时跳出循环
```

除了以上同类型循环嵌套外，我们还可以使用不同的循环类型来嵌套，如 for 循环体中嵌套 while 循环。

PS：

```
j =2   --这是一个全局变量
for i=2,10 --for 循环未使用第三个参数，默认每次加 1
do
    for j=2,(i/j) , 2
```

哔哩哔哩次元游戏人生

```
    do
        if(not(i%j))--求余
        then
            break
        end
        if(j > (i/j))
        then
            print("i 的值为：",i)
        end
    end
end
```

2. 循环控制语句

循环控制语句用于控制程序的流程， 以实现程序的各种结构方式。

1） break 语句

Lua 编程语言 break 语句插入在循环体中，用于退出当前循环或语句，并开始脚本执行紧接着的语句。

如果你使用循环嵌套，break 语句将停止最内层循环的执行，并开始执行的外层的循环语句。

PS：

```
a = 10--定义一个全局变量
while( a < 20 )--当 a<20 时，执行循环体内容
do
    print("a 的值为:", a)
    a=a+1;
    if( a > 15)
    then
        break--使用 break 语句终止循环
    end
end
```

上述代码执行 while 循环，在变量 a 小于 20 时输出 a 的值，并在 a 大于 15 时终止执行循环。

2） goto 语句

goto 语句允许将控制流程无条件地转到被标记的语句处。

语法格式如下：

```
goto Label
```

Label 的格式为：

```
:: Label ::
```

PS：

```
local a = 1   --这是一个局部变量
::label:: print("--- goto label ---")

a = a+1   --这是一个全局变量
if(a < 3)
then
```


哔哩哔哩次元游戏人生

```
    goto label     --当 a < 3  的时候跳转到标签  label
end
```

上述代码的最终输出会有两个--- goto label ---。

我们可以在 lable 中设置多个语句：

```
i = 0   --这是一个全局变量
::s1::
do
    print(i)
    i = i+1
end
if(i>3)
then
    os.exit()     -- i  大于  3  时退出
end
goto s1   --跳转到标签 s1 的位置
```

有了 goto 语句，我们就可以实现 continue 的功能。

PS：

```
for i=1, 3   --for 循环未使用第三个参数，将默认每次执行循环体后 i 加 1
do
    if (i <= 2)
    then
        print(i, "yes continue")
        goto continue   --跳转到 continue 标签
    end
    print(i, "no continue")
    ::continue::
    print([[i'm end]])
end
```

3. 无限循环

在循环体中如果终止条件永远为 true，循环语句就会永远执行下去。

PS：

```
while( true )
do
   print("循环将永远执行下去")
end
```

五、 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码，在条件为 false 时执行其他指定代码。

控制结构的条件表达式结果可以是任何值，Lua 认为 false 和 nil 为假，true 和非 nil 为真。

注意：Lua 中 0 为 true，而 C#中的 0 为假(false)。

PS：

```
if(0) --在 Lua 中，0 代表 true，这一点和 c 系列语言不一样
then
```


哔哩哔哩次元游戏人生

```
    print("0 为 true")
end
```

1. if 语句

if 语句由一个布尔表达式作为条件判断，其后紧跟其他语句组成。

语句语法格式如下：

```
if(布尔表达式)   --当布尔表达式结果为 true 时才执行条件语句
then
   --[ 在布尔表达式为 true 时执行的语句 --]
End
```

PS：

```
a = 10;   --定义了一个全局变量

if( a < 20 )   --使用 if 语句判断全局变量 a
then
    --[ if 条件为 true 时打印以下信息 --]
    print("a 小于 20" );
end
print("a 的值为:", a);
```

上述代码用于判断变量 a 的值是否小于 20，根据条件判断执行不同的输出。

2. if…else…语句

if 语句可以与 else 语句搭配使用，在 if 条件表达式为 false 时执行 else 语句代码块。

语句语法格式如下：

```
if(布尔表达式)
then
   --[ 布尔表达式为 true 时执行该语句块 --]
else
   --[ 布尔表达式为 false 时执行该语句块 --]
end
```

当布尔表达式为 true 时会 if 中的代码块会被执行，当布尔表达式为 false 时，else 的代码块会被执行。

Lua 认为 false 和 nil 为假，true 和非 nil 为真。

注意：Lua 中 0 为 true，而 C#中的 0 为假(false)。

PS：

```
a = 100;   --定义了一个全局变量
if( a < 20 )   --条件判断语句
then
    --[ if 条件为 true 时执行该语句块 --]
    print("a 小于 20" );
else
    --[ if 条件为 false 时执行该语句块 --]
    print("a 大于 20" );
end
print("a 的值为 :", a);
```

哔哩哔哩次元游戏人生

3. if...elseif...else 语句

if 语句可以与 elseif...else 语句搭配使用，在 if 条件表达式为 false 时执行 elseif...else 语句代码块，用于检测多个条件语句。

语句语法格式如下：

```
if( 布尔表达式 1)
then
--[ 在布尔表达式 1 为 true 时执行该语句块 --]

    elseif( 布尔表达式 2)
    then
    --[ 在布尔表达式 2 为 true 时执行该语句块 --]

        elseif( 布尔表达式 3)
        then
        --[ 在布尔表达式 3 为 true 时执行该语句块 --]
            else
            --[ 如果以上布尔表达式都不为 true 则执行该语句块 --]
end
```

PS：

```
a = 100   --这是一个全局变量
if( a == 10 )   --这是条件判断语句
then
    --[ 如果条件为 true 打印以下信息 --]
    print("a 的值为 10" )
    elseif( a == 20 )
    then
        --[ if else if 条件为 true 时打印以下信息 --]
        print("a 的值为 20" )
        elseif( a == 30 )
        then
            --[ if else if condition 条件为 true 时打印以下信息 --]
            print("a 的值为 30" )
            else
                --[ 以上条件语句没有一个为 true 时打印以下信息 --]
                print("没有匹配 a 的值" )
end
print("a 的真实值为: ", a )
```

上述代码对对变量 a 的值进行判断。

4. if 嵌套语句

if 语句允许嵌套，这就意味着你可以在一个 if 或 else if 语句中插入其他的 if 或 else if 语句。

语句语法格式如下：

```
if( 布尔表达式 1)
then
```

哔哩哔哩次元游戏人生

```
    --[ 布尔表达式 1 为 true 时执行该语句块 --]
    if(布尔表达式 2)
    then
        --[ 布尔表达式 2 为 true 时执行该语句块 --]
    end
end
```

我们可以用同样的方式嵌套 else if...else 语句。

PS：

```
--下面定义了两个全局变量
a = 100;
b = 200;

if( a == 100 )--条件判断语句
then
    --[ if 条件为 true 时执行以下 if 条件判断 --]
    if( b == 200 )
    then
        --[ if 条件为 true 时执行该语句块 --]
        print("a 的值为 100 b 的值为 200" );
    end
end
print("a 的值为 :", a );
print("b 的值为 :", b );
```

上述代码判断变量 a 和 b 的值，根据条件判断语句的结果输出不同的值。

六、 函数

在 Lua 中，函数是对语句和表达式进行抽象的主要方法。既可以用来处理一些特殊的工作，也可以用来计算一些值。

Lua 提供了许多的内建函数，你可以很方便的在程序中调用它们，如 print()函数可以将传入的参数打印在控制台上。

函数可以完成指定的任务，这种情况下函数作为调用语句使用；也可以计算并返回值，这种情况下函数作为赋值语句的表达式使用。

1. 函数定义

定义格式如下：

```
optional_function_scope function function_name( argument1, argument2, argument3..., argumentn)
    function_body    --函数体
    return result_params_comma_separated    --返回值
end
```

① optional_function_scope

该参数是可选的制定函数是全局函数还是局部函数，未设置该参数默认为全局函数，如果你需要设置函数为局部函数需要使用关键字 local。

② function_name

指定函数名称。

悟空聊理发元游戏人生

③ argument1, argument2, argument3..., argumentn
函数参数，多个参数以逗号隔开，函数也可以不带参数。

④ function_bod
函数体，函数中需要执行的代码语句块。

⑤ result_params_comma_separated
函数返回值，Lua 语言函数可以返回多个值，每个值以逗号隔开。
PS：

```
function max(num1, num2)--函数返回两个值的最大值
    if (num1 > num2)
    then
        result = num1;
    else
        result = num2;
    end

    return result;
end
-- 调用函数
print("两值比较最大值为 ",max(10,4))
print("两值比较最大值为 ",max(5,6))
```

上面代码定义了函数 max()，参数为 num1, num2，用于比较两值的大小，并返回最大值。

```
myprint = function(param)
    print("这是打印函数 -  ##",param,"##")
end

function add(num1,num2,functionPrint)
    result = num1 + num2
    functionPrint(result)-- 调用传递的函数参数
end
myprint(10)
-- myprint 函数作为参数传递
add(2,5,myprint)
```

在上述代码中，我们可以将函数作为参数传递给函数。

2. 多返回值函数

Lua 函数可以返回多个结果值，比如 string.find，其返回匹配串"开始和结束的下标"（如果不存在匹配串返回 nil）。
PS：

```
function maximum (a)--最大值
    local maxIndex = 1                  -- 最大值索引
    local maxValue = a[maxIndex]            -- 最大值
```

哔哩哔哩次元游戏人生

```
    for index,value in ipairs(a) do
        if value > maxValue
        then
            maxIndex = index
            maxValue = value
        end
    end
    return m, maxIndex
end

print(maximum({8,10,23,12,5}))
```

在上述代码中，我们的函数 maximum 返回了多个结果，即最大值和它所对应的索引。

3. 可变参数

Lua 函数可以接受可变数目的参数，和 C 语言类似，在函数参数列表中使用三点 ... 表示函数有可变的参数。

PS：

```
function add(...)
    local s = 0   --这是一个局部变量
    for i, v in ipairs{...} -- {...} 表示一个由所有变长参数构成的数组
    do
        s = s + v
    end
    return s
end
print(add(3,4,5,6,7))   --输出结果为 25
```

PS：

```
function average(...)--计算平均数
    result = 0   --定义一个全局变量 result
    local arg={...}      -- arg 为一个表，是局部变量
    for index,value in ipairs(arg)
    do
        result = result + value
    end
    print("总共传入 " .. #arg .. " 个数")
    return result/#arg
end

print("平均值为",average(10,5,3,4,5,6))
```

1） 获取可变参数的数量

可以通过 select("#",...) 来获取可变参数的数量。

PS：

```
function average(...)
    result = 0
```


哔哩哔哩次元游戏人生

```
    local arg={...}
    for index,value in ipairs(arg)
    do
        result = result + value
    end
    --使用 select("#",...) 来获取可变参数的数量
    print("总共传入 " .. select("#",...) .. " 个数")
    return result/select("#",...)
end

print("平均值为",average(10,5,3,4,5,6))
```

4. 固定参数加上可变参数

有时候我们可能需要几个固定参数加上可变参数，固定参数必须放在变长参数之前。

```
function fwrite(fmt, ...)   ---> 固定的参数 fmt
return io.write(string.format(fmt, ...))
end

fwrite("runoob\n")          --->fmt = "runoob", 没有变长参数。
fwrite("%d%d\n", 1, 2)      --->fmt = "%d%d", 变长参数为 1 和 2
```

通常在遍历变长参数的时候只需要使用 {…}，然而变长参数可能会包含一些 nil，那么就可以用 select 函数来访问变长参数了：select('#', …) 或者 select(n, …)。

select('#', …) ：返回可变参数的长度。

select(n, …) ：用于返回从起点 n 开始到结束位置的所有参数列表。

调用 select 时，必须传入一个固定实参 selector(选择开关) 和一系列变长参数。如果 selector 为数字 n，那么 select 返回参数列表中从索引 n 开始到结束位置的所有参数列表，否则只能为字符串#，这样 select 返回变长参数的总数。

PS：

```
function f(...)
    a = select(3,...)   -->从第三个位置开始，变量 a 对应右边变量列表的第一个参数
    print (a)
    print (select(3,...)) -->打印第 3 个位置后的所有列表参数
end
f(0,1,2,3,4,5) –调用函数 f()
```

PS：

```
do
    function foo(...)
        for i = 1, select('#', ...) do   -->获取参数总数
            local arg = select(i, ...); -->读取参数，arg 对应的是右边变量列表的第一个参数
            print("arg", arg);
        end
    end
    foo(1, 2, 3, 4);
end
```

5. 命名参数

Lua 的函数参数是和位置相关的，调用时实参会按顺序依次传给形参。有时候用名字指定参数是很有用的，比如 rename 函数用来给一个文件重命名，有时候我们我们记不清命名前后两个参数的顺序了：

```
rename(old="temp.lua", new="temp1.lua")
```

上面这段代码是无效的，Lua 可以通过将所有的参数放在一个表中，把表作为函数的唯一参数来实现上面这段伪代码的功能。因为 Lua 语法支持函数调用时实参可以是表的构造。

根据这个想法我们重定义了 rename：

```
function rename (arg)
    return os.rename(arg.old, arg.new)
end
```

当函数的参数很多的时候，这种函数参数的传递方式很方便的。例如 GUI 库中创建窗体的函数有很多参数并且大部分参数是可选的，可以用下面这种方式：

```
w = Window
{
    x=0, y=0, width=300, height=200,
    title = "Lua", background="blue",
    border = true
}
function Window (options)
    -- check mandatory options
    if type(options.title) ~= "string"
    then
        error("no title")
    elseif type(options.width) ~= "number"
    then
        error("no width")
    elseif type(options.height) ~= "number"
    then
        error("no height")
    end
    -- everything else is optional
    _Window(options.title,
            options.x or 0, -- default value
            options.y or 0, -- default value
            options.width, options.height,
            options.background or "white", -- default
            options.border -- default is false (nil)
    )
end
```

哔哩哔哩次元游戏人生

七、 再论函数

1. 闭包

当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征我们称作词法定界。虽然这看起来很清楚，事实并非如此，词法定界加上第一类函数在编程语言里是一个功能强大的概念，很少语言提供这种支持。

下面看一个简单的例子，假定有一个学生姓名的列表和一个学生名和成绩对应的表；现在想根据学生的成绩从高到低对学生进行排序。

PS：

```
names = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]    -- compare the grades
end)
```

假定创建一个函数实现此功能。

PS：

```
function sortbygrade (names, grades)
    table.sort(names, function (n1, n2)
        return grades[n1]  >  grades[n2]    -- compare the grades
    end)
end
```

上述例子中，包含在 sortbygrade 函数内部的 sort，其匿名函数可以访问 sortbygrade 的参数 grades，在匿名函数内部 grades 不是全局变量也不是局部变量，我们称作外部的局部变量（external local variable）或者 upvalue。（upvalue 意思有些误导，然而在 Lua 中他的存在有历史的根源，还有他比起 external local variable 简短）。

PS：

```
function newCounter()
    local i = 0
    return function()    -- anonymous function
        i = i + 1
        return i
    end
end
c1 = newCounter()
print(c1())      --> 1
print(c1())      --> 2
```

匿名函数使用 upvalue i 保存他的计数，当我们调用匿名函数的时候 i 已经超出了作用范围，因为创建 i 的函数 newCounter 已经返回了。然而 Lua 用闭包的思想正确处理了这种情况。简单的说，闭包是一个函数以及它的 upvalues。如果我们再次调用 newCounter，将创建一个新的局部变量 i，因此我们得到了一个作用在新的变量 i 上的新闭包。

PS：

```
c2 = newCounter()
print(c2())      --> 1
print(c1())      --> 3
print(c2())      --> 2
```

哔哩哔哩次元游戏人生

c1、c2 是建立在同一个函数上，但作用在同一个局部变量的不同实例上的两个不同的闭包。

技术上来讲，闭包指值而不是指函数，函数仅仅是闭包的一个原型声明；尽管如此，在不会导致混淆的情况下我们继续使用术语函数代指闭包。

闭包在上下文环境中提供很有用的功能，如前面我们见到的可以作为高级函数（sort）的参数；作为函数嵌套的函数（newCounter）。这一机制使得我们可以在 Lua 的函数世界里组合出奇幻的编程技术。闭包也可用在回调函数中，比如在 GUI 环境中你需要创建一系列 button，但用户按下 button 时回调函数被调用，可能不同的按钮被按下时需要处理的任务有点区别。具体来讲，一个十进制计算器需要 10 个相似的按钮，每个按钮对应一个数字，可以使用下面的函数创建他们。

PS：

```
function digitButton (digit)
    return Button{ label = digit,
        action = function ()
            add_to_display(digit)
        end
    }
end
```

上面的例子我们假定 Button 是一个用来创建新按钮的工具，label 是按钮的标签，action 是按钮被按下时调用的回调函数。（实际上是一个闭包，因为他访问 upvalue digit）。digitButton 完成任务返回后，局部变量 digit 超出范围，回调函数仍然可以被调用并且可以访问局部变量 digit。

闭包在完全不同的上下文中也是很有用途的。因为函数被存储在普通的变量内我们可以很方便的重定义或者预定义函数。通常当你需要原始函数有一个新的实现时可以重定义函数。例如你可以重定义 sin 使其接受一个度数而不是弧度作为参数。

PS：

```
oldSin = math.sin
math.sin = function (x)
    return oldSin(x*math.pi/180)
end
do
    local oldSin = math.sin
    local k = math.pi/180
    math.sin = function (x)
        return oldSin(x*k)
    end
end
```

这样我们把原始版本放在一个局部变量内，访问 sin 的唯一方式是通过新版本的函数。

利用同样的特征我们可以创建一个安全的环境（也称作沙箱，和 java 里的沙箱一样），当我们运行一段不信任的代码（比如我们运行网络服务器上获取的代码）时安全的环境是需要的，比如我们可以使用闭包重定义 io 库的 open 函数来限制程序打开的文件。

PS：

```
do
```

哔哩哔哩次元游戏人生

```
    local oldOpen = io.open
    io.open = function (filename, mode)
        if access_OK(filename, mode) then
            return oldOpen(filename, mode)
        else
            return nil, "access denied"
        end
    end
end
```

2. 非全局变量

Lua 中函数可以作为全局变量也可以作为局部变量，我们已经看到一些例子，比如函数作为 table 的域（大部分 Lua 标准库使用这种机制来实现的比如 io.read、math.sin）。这种情况下，必须注意函数和表语法。

1） 表和函数放在一起

PS：

```
Lib = {}
Lib.foo = function (x,y) return x + y end
Lib.goo = function (x,y) return x - y end
```

2） 使用表构造函数

PS：

```
Lib = {
    foo = function (x,y) return x + y end,
    goo = function (x,y) return x - y end
}
```

3） Lua 提供另一种语法方式

PS：

```
Lib = {}
function Lib.foo (x,y)
    return x + y
end
function Lib.goo (x,y)
    return x - y
end
```

当我们将函数保存在一个局部变量内时，我们得到一个局部函数，也就是说局部函数像局部变量一样在一定范围内有效。这种定义在包中是非常有用的，因为 Lua 把 chunk 当作函数处理，在 chunk 内可以声明局部函数（仅仅在 chunk 内可见），词法定界保证了包内的其他函数可以调用此函数。（下面是声明局部函数的两种方式）

a. 方式一

PS：

```
local f = function (...)
    ...
```

哔哩哔哩次元游戏人生

```
end
local g = function (...)
    ...
    f() -- external local `f' is visible here
    ...
end
```

b. **方式二**

PS：

```
local function f (...)
    ...
End
```

有一点需要注意的是声明递归局部函数的方式。

PS：

```
local fact = function (n)
    if n == 0 then
        return 1
    else
        return n*fact(n-1) -- buggy
    end
end
```

上面这种方式导致 Lua 编译时遇到 fact(n-1)并不知道他是局部函数 fact，Lua 会去查找是否有这样的全局函数 fact。为了解决这个问题我们必须在定义函数以前先声明。

PS：

```
local fact
fact = function (n)
    if n == 0 then
        return 1
    else
        return n*fact(n-1)
    end
end
```

这样在 fact 内部 fact(n-1)调用是一个局部函数调用，运行时 fact 就可以获取正确的值了。

但是 Lua 扩展了他的语法，使得其可以直接在递归函数定义时使用两种方式。在定义非直接递归局部函数时要先声明然后定义才可以。

PS：

```
local f, g        -- `forward' declarations
function g ()
    ... f() ...
end
function f ()
    ... g() ...
```

哔哩哔哩次元游戏人生

```
end
```

3. 正确的尾调用

Lua 中函数的另一个有趣的特征是可以正确的处理尾调用（proper tail recursion，一些书使用术语“尾递归”，虽然并未涉及到递归的概念）。

尾调用是一种类似在函数结尾的 goto 调用，当函数最后一个动作是调用另外一个函数时，我们称这种调用尾调用。

PS：

```
function f(x)
    return g(x)
end
```

g 的调用是尾调用。

例子中 f 调用 g 后不会再做任何事情，这种情况下当被调用函数 g 结束时程序不需要返回到调用者 f；所以尾调用之后程序不需要在栈中保留关于调用者的任何信息。一些编译器比如 Lua 解释器利用这种特性在处理尾调用时不使用额外的栈，我们称这种语言支持正确的尾调用。

由于尾调用不需要使用栈空间，那么尾调用递归的层次可以无限制的。例如下面调用不论 n 为何值不会导致栈溢出。

PS：

```
function foo (n)
if n > 0 then return foo(n - 1) end
end
```

注意：必须明确什么是尾调用，一些调用者函数调用其他函数后，没有做其他的事情这不属于尾调用。

PS：

```
function f (x)
    g(x)
    return
end
```

上面这个例子中 f 在调用 g 后，不得不丢弃 g 地返回值，所以不是尾调用，同样的下面几个例子也不时尾调用。

PS：

```
return g(x) + 1        -- must do the addition
return x or g(x)       -- must adjust to 1 result
return (g(x))      -- must adjust to 1 result
```

Lua 中类似 return g(...)这种格式的调用是尾调用。但是 g 和 g 的参数都可以是复杂表达式，因为 Lua 会在调用之前计算表达式的值。

```
return x[i].foo(x[j] + a*b, i + j)
```

可以将尾调用理解成一种 goto，在状态机的编程领域尾调用是非常有用的。状态机的应用要求函数记住每一个状态，改变状态只需要 goto(or call)一个特定的函数。我们考虑一个迷宫游戏作为例子，迷宫有很多个房间，每个房间有东西南北四个门，每一步输入一个移动的方向，如果该方向存在即到达该方向对应的房间，否则程序打印警告信息。目标是：从开始的房间到达目的房间。

这个迷宫游戏是典型的状态机，每个当前的房间是一个状态。我们可以对每个房间写一个函数实现这个迷宫游戏，我们使用尾调用从一个房间移动到另外一个房间。一个四个

哔哩哔哩次元游戏人生

房间的迷宫代码如下：

PS：

```
function room1 ()
    local move = io.read()
    if move == "south" then
        return room3()
    elseif move == "east" then
        return room2()
    else
        print("invalid move")
        return room1() -- stay in the same room
    end
end

function room2 ()
    local move = io.read()
    if move == "south" then
        return room4()
    elseif move == "west" then
        return room1()
    else
        print("invalid move")
        return room2()
    end
end

function room3 ()
    local move = io.read()
    if move == "north" then
        return room1()
    elseif move == "east" then
        return room4()
    else
        print("invalid move")
        return room3()
    end
end

function room4 ()
    print("congratilations!")
end
```

我们可以调用 room1()开始这个游戏。

如果没有正确的尾调用，每次移动都要创建一个栈，多次移动后可能导致栈溢出。但正确的尾调用可以无限制的尾调用，因为每次尾调用只是一个 goto 到另外一个函数并不是

哔哩哔哩次元游戏人生

传统的函数调用。

八、 运算符

1. 算术运算符

下面表格列出了 Lua 语言常用算术运算符，设定 A 的值为 10，B 的值为 20。

操作符	描述	示例
+	加法	A + B 输出结果 30
-	减法	A - B 输出结果 -10
*	乘法	A * B 输出结果 200
/	除法	B / A 输出结果 2
%	取余	B % A 输出结果 0
^	乘幂	A^2 输出结果 100
-	负号	-A 输出结果 -10
//	整除运算符（Lua 版本大于 5.3 可用）	5//2 输出结果 2

在 Lua 中，/ 用作除法运算，计算结果包含小数部分，// 用作整除运算，计算结果不包含小数部分。

```
a = 5
b = 2
print("除法运算 - a/b 的值为 ", a / b )
print("整除运算 - a//b 的值为 ", a // b )
```

上述代码运行结果为

```
除法运算 - a/b 的值为    2.5
整除运算 - a//b 的值为   2
```

2. 关系运算符

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为 10，B 的值为 20。

操作符	描述	示例
==	等于，检测两个值是否相等，相等返回 true，否则返回 false	(A == B) 为 false。
~=	不等于，检测两个值是否相等，不相等返回 true，否则返回 false	(A ~= B) 为 true。
>	大于，如果左边的值大于右边的值，返回 true，否则返回 false	(A > B) 为 false。
<	小于，如果左边的值大于右边的值，返回 false，否则返回 true	(A < B) 为 true。
>=	大于等于，如果左边的值大于等于右边的值，返回 true，否则返回 false	(A >= B) 返回 false。
<=	小于等于， 如果左边的值小于等于右边的值，返回	(A <= B) 返回 true。

哔哩哔哩次元游戏人生

	true，否则返回 false	

这些操作符返回结果为 false 或者 true；==和~=比较两个值，如果两个值类型不同,Lua 认为两者不同；nil 只和自己相等。Lua 通过引用比较 tables、userdata、functions。

也就是说当且仅当两者表示同一个对象时相等。

3. 逻辑运算符

操作符	描述	示例
and	逻辑与操作符。 若 A 为 false，则返回 A，否则返回 B。	(A and B) 为 false。
or	逻辑或操作符。 若 A 为 true，则返回 A，否则返回 B。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	not(A and B) 为 true。

4. 其他运算符

操作符	描述	示例
…	连接两个字符串	a..b ，其中 a 为 "Hello " ， b 为 "World", 输出结果为 "Hello World"。
#	一元运算符，返回字符串或表的长度	#"Hello" 返回 5

5. 运算符优先级

下表是优先级从高到低的顺序

^					
not	-				
*	/	%			
+	-				
..					
<	>	<=	>=	~=	==
And					
or					

除了^和..外所有的二元运算符都是左连接的

a+i < b/2+1	<——>	(a+i) < ((b/2)+1)
5+x^2*8	<——>	5+((x^2)*8)
a < y and y <= z	<——>	(a < y) and (y <= z)
-x^2	<——>	-(x^2)
x^y^z	<——>	x^(y^z)

九、 Lua 字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

哔哩哔哩次元游戏人生

① 单引号间的一串字符。
② 双引号间的一串字符。
③ [[与]] 间的一串字符。

PS：

```
string1 = "Lua"
print("\"字符串 1 是\"",string1)
string2 = '666'
print("字符串 2 是",string2)

string3 = [["Lua 语言"]]
print("字符串 3 是",string3)
```

1. 转义字符

转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。如在字符串转换双引号可以使用 "\""。

转义字符	意义	ASCII 码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS) ，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF) ，将当前位置移到下一行开头	010
\r	回车(CR) ，将当前位置移到本行开头	013
\t	水平制表(HT) （跳到下一个 TAB 位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符"\'	092
\'	代表一个单引号（撇号）字符	039
\"	代表一个双引号字符	034
\0	空字符(NULL)	000
\ddd	1 到 3 位八进制数所代表的任意字符	三位八进制
\xhh	1 到 2 位十六进制所代表的任意字符	二位十六进制

2. 字符串操作

序号	方法&用途	示例
1	string.upper(argument): 字符串全部转为大写字母。	string1 = "Lua"; print(string.upper(string1))

哔哩哔哩次元游戏人生

		结果为：LUA
2	string.lower(argument): 字符串全部转为小写字母。	string1 = "Lua"; print(string.lower(string1)) 结果为：lua
3	string.gsub(mainString,findString,replaceString,num) 在字符串中替换。 mainString 为要操作的字符串， findString 为被替换的字符，replaceString 要替换的字符，num 替换次数（可以忽略，则全部替换）。	string.gsub("aaaa","a","z",3);
4	string.find (str, substr, [init, [end]]) 在一个指定的目标字符串 str 中搜索指定的内容 substr，如果找到了一个匹配的子串，就会返回这个子串的起始索引和结束索引，不存在则返回 nil。	查找"Lua"的起始索引和结束索引位置 string.find("Hello Lua user", "Lua", 1) 结果为：7 9
5	string.reverse(arg) 将字符串反转	
6	string.format(...) 返回一个类似 printf 的格式化字符串	string.format("the value is:%d",4) 结果：the value is:4
7	string.char(arg) 和 string.byte(arg[,int]) char 将整型数字转成字符并连接， byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。	
8	string.len(arg) 计算字符串长度。	string.len("abc") 结果为：3
9	string.rep(string, n) 返回字符串 string 的 n 个拷贝	string.rep("abcd",2) 结果为：abcdabcd
10	.. 链接两个字符串	print("www.runoob.."com") 结果为：www.runoob.com
11	string.gmatch(str, pattern) 返回一个迭代器函数，每一次调用这个函数，返回一个在字符串 str 找到的下一个符合 pattern 描述的子串。如果参数 pattern 描述的字符串没有找到，迭代函数返回 nil。	for word in string.gmatch("Hello Lua user", "%a+") do print(word) end 结果：Hello Lua user
12	string.match(str, pattern, init) string.match()只寻找源字串 str 中的第一个配对。参数 init 可选，指定搜寻过程的起点，默认为 1。在成功配对时，函数将返回配对表达式中的所有捕获结果；如果没有设置捕获标记，则返回整个配对字符串．当没有成功的配对时，返回 nil。	print(string.match("I have 2 questions for you.", "%d+ %a+")) 结果为：2 questions print(string.format("%d, %q", string.match("I have 2

哔哩哔哩次元游戏人生

		questions for you.", "(%d+)(%a+)"))) 结果为：, "questions"

3. 字符串截取

字符串截取使用 sub()方法，string.sub() 用于截取字符串。

string.sub(s, i [, j])

① s：要截取的字符串。

② i：截取开始位置。

③ j：截取结束位置，默认为 -1，最后一个字符。

```
-- 字符串
local sourcestr = "prefix--runoobgoogletaobao--suffix"
print("\n 原始字符串", string.format("%q", sourcestr))

-- 截取部分，第 4 个到第 15 个
local first_sub = string.sub(sourcestr, 4, 15)
print("\n 第一次截取", string.format("%q", first_sub))

-- 取字符串前缀，第 1 个到第 8 个
local second_sub = string.sub(sourcestr, 1, 8)
print("\n 第二次截取", string.format("%q", second_sub))

-- 截取最后 10 个
local third_sub = string.sub(sourcestr, -10)
print("\n 第三次截取", string.format("%q", third_sub))

-- 索引越界，输出原始字符串
local fourth_sub = string.sub(sourcestr, -100)
print("\n 第四次截取", string.format("%q", fourth_sub))
```

上述代码执行结果为：

```
原始字符串      "prefix--runoobgoogletaobao--suffix"
第一次截取      "fix--runoobg"
第二次截取      "prefix--"
第三次截取      "ao--suffix"
第四次截取      "prefix--runoobgoogletaobao--suffix"
```

4. 字符串格式化

Lua 提供了 string.format() 函数来生成具有特定格式的字符串，函数的第一个参数是格式，之后是对应格式中每个代号的各种数据。

由于格式字符串的存在，使得产生的长字符串可读性大大提高了。这个函数的格式很像 C 语言中的 printf()。

格式字符串可能包含以下的转义码:

转义码	描述
%c	接受一个数字，并将其转化为 ASCII 码表

哔哩哔哩次元游戏人生

	中对应的字符
%d,%i	接受一个数字并将其转化为有符号的整数格式
%o	接受一个数字并将其转化为八进制数格式
%x	接受一个数字并将其转化为无符号整数格式
%X	接受一个数字并将其转化为十六进制数格式，使用小写字母
%e	接受一个数字并将其转化为科学记数法格式，使用小写字母 e
%E	接受一个数字并将其转化为科学记数法格式，使用大写字母 E
%f	接受一个数字并将其转化为浮点数格式
%g(%G)	接受一个数字并将其转化为%e(%E，对应%G)及%f 中较短的一种格式
%q	接受一个字符串并将其转化为可安全被 Lua 编译器读入的格式
%s	接受一个字符串并按照给定的参数格式化该字符串

为进一步细化格式，可以在%号后添加参数。参数将以如下的顺序读入:

① 符号

一个+号表示其后的数字转义符将让正数显示正号。默认情况下只有负数显示符号。

② 占位符

一个 0，在后面指定了字串宽度时占位用。不填时的默认占位符是空格。

③ 对齐标识

在指定了字串宽度时，默认为右对齐，增加-号可以改为左对齐。

④ 宽度数值

⑤ 小数位数/字串裁切

在宽度数值后增加的小数部分 n，若后接 f(浮点数转义符，如%6.3f)则设定该浮点数的小数只保留 n 位，若后接 s(字符串转义符，如%5.3s)则设定该字符串只显示前 n 位。

PS：

```
--下面是定义的全局变量，分别是 string 和 number 类型
string1 = "Lua"
string2 = "Tutorial"
number1 = 10
number2 = 20
-- 基本字符串格式化
print(string.format("基本格式化 %s %s",string1,string2))--%s 接受一个字符串并按照给定的参数格式化该字符串
-- 日期格式化
date = 2; month = 1; year = 2014
print(string.format("日期格式化 %02d/%02d/%03d", date, month, year))--%d 接受一个数字并将其转化为有符号的整数格式
-- 十进制格式化
```

print(string.format("%.4f",1/3))--接受一个数字并将其转化为浮点数格式

5. 字符与整数相互转换

--byte 转换字符为整数值(可以指定某个字符，默认第一个字符)

print(string.byte("Lua")) --结果为 76

-- 转换第三个字符

print(string.byte("Lua",3)) --结果为 97

-- 转换末尾第一个字符

print(string.byte("Lua",-1)) --结果为 97

-- 第二个字符

print(string.byte("Lua",2)) --结果为 117

-- 转换末尾第二个字符

print(string.byte("Lua",-2)) --结果为 117

-- 整数 ASCII 码转换为字符

print(string.char(97))--char 将整型数字转成字符并连接 --结果为 a

6. 匹配模式

Lua 中的匹配模式直接用常规的字符串来描述。 它用于模式匹配函数 string.find, string.gmatch, string.gsub, string.match。

你还可以在模式串中使用字符类。

字符类指可以匹配一个特定字符集合内任何字符的模式项。比如，字符类 %d 匹配任意数字。所以你可以使用模式串 %d%d/%d%d/%d%d%d%d 搜索 dd/mm/yyyy 格式的日期：

s = "Deadline is 30/05/1999, firm"

date = "%d%d/%d%d/%d%d%d%d"

print(string.sub(s, string.find(s, date))) --输出为 30/05/1999

下面列表列出了 Lua 支持的所有字符类。

单个字符(除 ^$()%.[]*+-? 外): 与该字符自身配对。

.（点）	与任何字符配对
%a	与任何字符配对
%c	与任何控制符配对(例如\n)
%d	与任何数字配对
%l	与任何小写字母配对
%p	与任何标点(punctuation)配对
%s	与空白字符配对
%u	与任何大写字母配对
%w	与任何字母/数字配对
%x	与任何十六进制数配对
%z	与任何代表 0 的字符配对
%x(此处 x 是非字母非数字字符)	与字符 x 配对. 主要用来处理表达式中有功能的字符(^$()%.[]*+-?)的配对问题, 例如%%与%配对
[数个字符类]	与任何[]中包含的字符类配对. 例如[%w_]与任何字母/数字, 或下划线符号(_)配对

哔哩哔哩次元游戏人生

[^数个字符类]	与任何不包含在[]中的字符类配对. 例如[^%s]与任何非空白字符配对

当上述的字符类用大写书写时，表示与非此字符类的任何字符配对。例如，%S 表示与任何非空白字符配对。例如，'%A'非字母的字符。

PS：print(string.gsub("hello, up-down!", "%A", "."))

结果为 hello..up.down.　　4；数字 4 不是字符串结果的一部分，他是 gsub 返回的第二个结果，代表发生替换的次数。

在模式匹配中有一些特殊字符，他们有特殊的意义，Lua 中的特殊字符如下：

(	)	.	%	+	-	*	?	[	^	$

'%' 用作特殊字符的转义字符，因此 '%.' 匹配点；'%%' 匹配字符 '%'。转义字符 '%' 不仅可以用来转义特殊字符，还可以用于所有的非字母的字符。

十、 Lua 数组

数组，就是相同数据类型的元素按一定顺序排列的集合，可以是一维数组和多维数组。

Lua 数组的索引键值可以使用整数表示，数组的大小不是固定的。

1. 一维数组

一维数组是最简单的数组，其逻辑结构是线性表。一维数组可以用 for 循环出数组中的元素。

PS：

```
array = {"Lua", "Tutorial"}    —这是一个直接初始表
for i= 0, 2
do
    print(array[i])
end
```

上述代码的输出结果为，nil；Lua；Tutorial。

正如我们所看到的，我们可以使用整数索引来访问数组元素，如果知道的索引没有值则返回 nil。

在 Lua 索引值是以 1 为起始，但你也可以指定 0 开始。

除此外我们还可以以负数为数组索引值。

PS：

```
array = {}--这是一个空表
for i= -2, 2    --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
    array[i] = i *2
end
for i = -2,2 do
    print(array[i])
end
```

上述代码结果为-4，-2，0，2，4。

2. 多维数组

多维数组即数组中包含数组或一维数组的索引键对应一个数组。

1） 数组的数组

以下是一个三行三列的阵列多维数组（矩阵）：

哔哩哔哩次元游戏人生

PS：

```
-- 初始化数组
array = {}    --这是一个空表
for i=1,3    --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
    array[i] = {}
    for j=1,3    --该 for 循环没有设置第三个参数，因此将采用默认值 1
    do
        array[i][j] = i*j
    end
end

-- 访问数组
for i=1,3    --该 for 循环没有设置第三个参数，因此将采用默认值 1
do
    for j=1,3
    do
        print(array[i][j])
    end
end
```

2） 不同索引键的一维数组

表示矩阵的另一方法，是将行和列组合起来。如果索引下标都是整数，通过第一个索引乘于一个常量（列）再加上第二个索引。

PS：

```
-- 初始化数组
array = {}
maxRows = 3    --最大的行数
maxColumns = 3    --最大的列数
for row=1,maxRows
do
    for col=1,maxColumns
    do
        array[row*maxColumns +col] = row*col
    end
end

-- 访问数组
for row=1,maxRows
do
    for col=1,maxColumns
    do
        print(array[row*maxColumns +col])
    end
end
```

哔哩哔哩次元游戏人生

如果索引是字符串，可用一个单字符将两个字符串索引连接起来构成一个单一的索引下标，例如一个矩阵 m，索引下标为 s 和 t，假定 s 和 t 都不包含冒号，代码为 m[s..':'..t]，

如果 s 或者 t 包含冒号将导致混淆，比如("a:", "b") 和("a", ":b")，当对这种情况有疑问的时候可以使用控制字符来连接两个索引字符串，比如'\0'。

实际应用中常常使用稀疏矩阵，稀疏矩阵指矩阵的大部分元素都为空或者 0 的矩阵。

例如，我们通过图的邻接矩阵来存储图，也就是说：当 m,n 两个节点有连接时，矩阵的 m,n 值为对应的 x，否则为 nil。如果一个图有 10000 个节点，平均每个节点大约有 5 条边，为了存储这个图需要一个行列分别为 10000 的矩阵，总计 10000*10000 个元素，实际上大约只有 50000 个元素非空（每行有五列非空，与每个节点有五条边对应）。很多数据结构的书上讨论采用何种方式才能节省空间，但是在 Lua 中你不需要这些技术，因为用 table 实现的数据本身天生的就具有稀疏的特性。如果用我们上面说的第一种多维数组来表示，需要 10000 个 table，每个 table 大约需要五个元素（table）；如果用第二种表示方法来表示，只需要一张大约 50000 个元素的表，不管用那种方式，你只需要存储那些非 nil 的元素。

十一、 Lua 迭代器

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址。

在 Lua 中迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

1. 泛型 for 迭代器

泛型 for 在自己内部保存迭代函数，实际上它保存三个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：

```
for key, value in pairs(t)
do
    print(key, value)
end
```

上面代码中，key, value 为变量列表；pairs(t)为表达式列表。

PS：

```
array = {"Google", "Runoob"}   --这是一个直接初始 table
for key,value in ipairs(array)
do
    print(key, value)
end
```

以上示例中我们使用了 Lua 默认提供的迭代函数 ipairs。

下面我们看看泛型 for 的执行过程：

① 首先

初始化，计算 in 后面表达式的值，表达式应该返回泛型 for 需要的三个值：迭代函数、状态常量、控制变量；与多值赋值一样，如果表达式返回的结果个数不足三个会自动用 nil 补足，多出部分会被忽略。

② 第二

将状态常量和控制变量作为参数调用迭代函数（注意：对于 for 结构来说，状态常量没有用处，仅仅在初始化时获取他的值并传递给迭代函数）。

③ 第三

将迭代函数返回的值赋给变量列表。

哔哩哔哩次元游戏人生

④ 第四

如果返回的第一个值为 nil 循环结束，否则执行循环体。

⑤ 第五

回到第二步再次调用迭代函数

2. 无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器，因此在循环中我们可以利用无状态迭代器避免创建闭包花费额外的代价。

每一次迭代，迭代函数都是用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器只利用这两个值可以获取下一个元素。

这种无状态迭代器的典型的简单的例子是 ipairs，它遍历数组的每一个元素，元素的索引需要是数值。

PS：

```
function square(iteratorMaxCount,currentNumber)--iteratorMaxCount 表示迭代的总次数
    if (currentNumber<iteratorMaxCount)
    then
        currentNumber = currentNumber+1;
        return currentNumber, currentNumber*currentNumber
    end
end

for index,number in square,3,0--总的迭代次数为 3，从 0 开始
do
    print(index,number);
end
```

上述代码我们使用了一个简单的函数来实现迭代器，实现了数字 n 的平方。

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），ipairs 和迭代函数都很简单。

PS：

```
function iter (a, i)
    i = i + 1
    local value = a[i]
    if value then
        return i, value
    end
end

function ipairs (a)
    return iter, a, 0
end
```

当 Lua 调用 ipairs(a)开始循环时，他获取三个值：迭代函数 iter、状态常量 a、控制变量初始值 0；然后 Lua 调用 iter(a,0)返回 1, a[1]（除非 a[1]=nil）；第二次迭代调用 iter(a,1)返回 2, a[2]……直到第一个 nil 元素。

3. 多状态的迭代器

很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简

哔哩哔哩次元游戏人生

单的方法是使用闭包，还有一种方法就是将所有的状态信息封装到 table 内，将 table 作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在 table 内，所以迭代函数通常不需要第二个参数。

```
array = {"Google", "Runoob"}

function elementIterator (collection)
    local index = 0    --这是一个局部变量
    local count = #collection    --使用#来计算字符串的长度
    -- 闭包函数
    return function ()
        index = index + 1
        if (index <= count)
        then
            --   返回迭代器的当前元素
            return collection[index]
        end
    end
end

for element in elementIterator(array)
do
    print(element)
end
```

上述代码，我们将所有的信息都存放在了 table 内，所以迭代函数不需要第二个参数。

十二、 表-table

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数组、字典等。

Lua table 使用关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil。

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

Lua 也是通过 table 来解决模块（module）、包（package）和对象（Object）的。 例如 string.format 表示使用"format"来索引 table string。

1. 表的构造

构造器是创建和初始化表的表达式。表是 Lua 特有的功能强大的东西。最简单的构造函数是{}，用来创建一个空表。

PS：

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1]= "Lua"

-- 移除引用
mytable = nil
-- lua 垃圾回收会释放内存
```

对于 table，数据类型 nil 还有一个"删除"作用，给 table 表里的变量赋一个 nil 值，等同于把它们删掉。

当我们为 table a 设置元素，然后将 a 赋值给 b，则 a 与 b 都指向同一个内存（即引用的地址）。如果将 a 设置为 nil ，则 b 依然能访问 table 的元素。如果没有指定的变量指向 a，Lua 的垃圾回收机制会清理相对应的内存。

PS：

```
mytable = {}   -- 一个简单的 table
print("mytable 的类型是 ",type(mytable))   --使用 type 函数测试给定变量或者值的类型

mytable[1]= "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

alternatetable = mytable   --指向同一个内存（即引用的地址）

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])

alternatetable["wow"] = "修改后"

print("mytable 索引为 wow 的元素是 ", mytable["wow"])

alternatetable = nil   -- 给 table 表里的变量赋一个 nil 值，等同于把它们删掉
print("alternatetable 是 ", alternatetable)

-- mytable 仍然可以访问
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

mytable = nil
print("mytable 是 ", mytable)
```

2. Table 操作

以下列出了 Table 操作常用的方法：

序号	方法&用途
1	table.concat (table [, sep [, start [, end]]]): concat 是 concatenate(连锁，连接)的缩写。 table.concat()函数列出参数中指定 table 的数组部分从 start 位置到 end 位置的所有元素，元素间以指定的分隔符(sep)隔开。
2	table.insert (table, [pos,] value): 在 table 的数组部分指定位置(pos)插入值为 value 的一个元素，其他元素向后移动。 pos 参数可选，默认为数组部分末尾。
3	table.maxn (table)

哔哩哔哩次元游戏人生

	指定 table 中所有正数 key 值中最大的 key 值。如果不存在 key 值为正数的元素，则返回 0。(Lua5.2 之后该方法已经不存在了,本文使用了自定义函数实现)
4	table.remove (table [, pos]) 返回 table 数组部分位于 pos 位置的元素。其后的元素会被前移。pos 参数可选，默认为 table 长度，即从最后一个元素删起。
5	table.sort (table [, comp]) 对给定的 table 进行升序排序。

1） 连接

我们可以使用 concat() ，将一个列表中的元素连接成一个字符串，然后将其输出。

PS：

```
fruits = {"banana","orange","apple"}
-- 返回将 table 中的元素连接后的字符串
print("连接后的字符串 ",table.concat(fruits))
-- 指定 table 中的各元素以什么字符进行连接
print("连接后的字符串 ",table.concat(fruits,"：  "))
-- 指定从 table 中的第几个元素开始，连接到第几个元素结束
print("连接后的字符串 ",table.concat(fruits,", ", 2,3))
```

2） 插入和移除

PS：

```
fruits = {"banana","orange","apple"}

-- 未指定插入位置，默认在 table 的末尾进行插入
table.insert(fruits,"mango")
print("索引为 4 的元素为 ",fruits[4])

-- 指定了插入位置，在索引为 2 的键处插入，其他元素后移
table.insert(fruits,2,"grapes")
print("索引为 2 的元素为 ",fruits[2])

print("最后一个元素为 ",fruits[5])
table.remove(fruits)--未指定位置，将从最后一个元素开始移除
print("移除后最后一个元素为 ",fruits[5])
```

3） 排序

PS：

```
fruits = {"banana","orange","apple","grapes"}
print("排序前")
for key,value in ipairs(fruits)
do
    print(key,value)
end
```

哔哩哔哩次元游戏人生

```
table.sort(fruits)--将 table 表按照顺序进行排序
print("排序后")
for key,value in ipairs(fruits)
do
    print(key,value)
end
```

4） 最大值

table.maxn 在 Lua5.2 之后该方法已经不存在了，我们定义了 table_maxn 方法来实现。

PS：

```
function table_maxn(t)
    local mn=nil;
    for key, value in pairs(t)
    do
        if(mn==nil)
        then
            mn=value
        end
        if mn < value
        then
            mn = value
        end
    end
    return mn
end
tbl = {[1] = 2, [2] = 6, [3] = 34, [26] =5}--关联型数组
print("tbl 最大值： ", table_maxn(tbl))
print("tbl 长度 ", #tbl)
```

注意：

当我们获取 table 的长度的时候无论是使用 # 还是 table.getn 其都会在索引中断的地方停止计数，而导致无法正确取得 table 的长度。

可以使用以下方法来代替：

PS：

```
function table_leng(t)
    local leng=0
    for key, value in pairs(t)
    do
        leng=leng+1
    end
    return leng;
end
```

十三、 数据文件与持久化

当我们处理数据文件的，一般来说，写文件比读取文件内容来的容易。因为我们可以很好的控制文件的写操作，而从文件读取数据常常碰到不可预知的情况。一个健壮的程序

哔哩哔哩次元游戏人生

不仅应该可以读取存有正确格式的数据还应该能够处理坏文件（译者注：对数据内容和格式进行校验，对异常情况能够做出恰当处理）。正因为如此，实现一个健壮的读取数据文件的程序是很困难的。

文件格式可以通过使用 Lua 中的 table 构造器来描述。我们只需要在写数据的稍微做一些做一点额外的工作，读取数据将变得容易很多。方法是：将我们的数据文件内容作为 Lua 代码写到 Lua 程序中去。通过使用 table 构造器，这些存放在 Lua 代码中的数据可以像其他普通的文件一样看起来引人注目。

为了更清楚地描述问题，下面我们看看例子。如果我们的数据是预先确定的格式，比如 CSV（逗号分割值），我们几乎没得选择。但是，如果我们打算创建一个文件为了将来使用，除了 CSV，我们可以使用 Lua 构造器来我们表述我们数据，这种情况下，我们将每一个数据记录描述为一个 Lua 构造器。将下面的代码

Donald E. Knuth,Literate Programming,CSLI,1992

Jon Bentley,More Programming Pearls,Addison-Wesley,1990

写成

PS：

```
Entry{"Donald E. Knuth",
"Literate Programming",
"CSLI",
1992}
Entry{"Jon Bentley",
"More Programming Pearls",
"Addison-Wesley",
1990}
```

记住 Entry{...}与 Entry({...})等价，他是一个以表作为唯一参数的函数调用。所以，前面那段数据在 Lua 程序中表示如上。如果要读取这个段数据，我们只需要运行我们的 Lua 代码。

PS：

```
local count = 0
function Entry (b) count = count + 1 end
dofile("data")
print("number of entries: " .. count)
```

下面这段程序收集一个作者名列表中的名字是否在数据文件中出现，如果在文件中出现则打印出来。（作者名字是 Entry 的第一个域；所以，如果 b 是一个 entry 的值，b[1]则代表作者名）

PS：

```
local authors = {}      -- a set to collect authors
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

注意，在这些程序段中使用事件驱动的方法：Entry 函数作为回调函数，dofile 处理数据文件中的每一记录都回调用它。当数据文件的大小不是太大的情况下，我们可以使用 name-value 对来描述数据。

PS：

```
Entry{
```

哔哩哔哩次元游戏人生

```
author = "Donald E. Knuth",
title = "Literate Programming",
publisher = "CSLI",
year = 1992
}

Entry{
author = "Jon Bentley",
title = "More Programming Pearls",
publisher = "Addison-Wesley",
year = 1990
}
```

（如果这种格式让你想起 BibTeX，这并不奇怪。Lua 中构造器正是根据来自 BibTeX 的灵感实现的）这种格式我们称之为自描述数据格式，因为每一个数据段都根据他的意思简短的描述为一种数据格式。相对 CSV 和其他紧缩格式，自描述数据格式更容易阅读和理解，当需要修改的时候可以容易的手工编辑，而且不需要改动数据文件。例如，如果我们想增加一个域，只需要对读取程序稍作修改即可，当指定的域不存在时，也可以赋予默认值。使用 name-value 描述的情况下，上面收集作者名的代码可以改写为。

PS：

```
local authors = {}      -- a set to collect authors
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

现在，记录域的顺序无关紧要了，甚至某些记录即使不存在 author 这个域，我们也只需要稍微改动一下代码即可。

```
function Entry (b)
if b.author then authors[b.author] = true end
end
```

Lua 不仅运行速度快，编译速度也快。例如，上面这段搜集作者名的代码处理一个 2MB 的数据文件时间不会超过 1 秒。另外，这不是偶然的，数据描述是 Lua 的主要应用之一，从 Lua 发明以来，我们花了很多心血使他能够更快的编译和运行大的 chunks。

1. 序列化

我们经常需要序列化一些数据，为了将数据转换为字节流或者字符流，这样我们就可以保存到文件或者通过网络发送出去。我们可以在 Lua 代码中描述序列化的数据，在这种方式下，我们运行读取程序即可从代码中构造出保存的值。

通常，我们使用这样的方式 varname = <exp>来保存一个全局变量的值。varname 部分比较容易理解，下面我们来看看如何写一个产生值的代码。

对于一个数值，PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    else ...
end
```

对于字符串值，PS：

```
if type(o) == "string" then
   io.write("""", o, """)
```

然而，如果字符串包含特殊字符（比如引号或者换行符），产生的代码将不是有效的 Lua 程序，这时候你可能用下面方法解决特殊字符的问题。

PS：

```
if type(o) == "string" then
   io.write("[[", o, "]]")
```

千万不要这样做！双引号是针对手写的字符串的而不是针对自动产生的字符串。如果有人恶意的引导你的程序去使用"]]..os.execute('rm *')..[["这样的方式去保存某些东西（比如它可能提供字符串作为地址）你最终的 chunk 将是这个样子。

```
varname = [[ ]]..os.execute('rm *')..[[ ]]
```

如果你 load 这个数据，运行结果可想而知的。为了以安全的方式引用任意的字符串，string 标准库提供了格式化函数专门提供"%q"选项。它可以使用双引号表示字符串并且可以正确的处理包含引号和换行等特殊字符的字符串。

这样一来，我们的序列化函数可以写为。

PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    else ...
end
```

1） 保存不带循环的 table

我们下一个艰巨的任务是保存表。根据表的结构不同，采取的方法也有很多。没有一种单一的算法对所有情况都能很好地解决问题。简单的表不仅需要简单的算法而且输出文件也需要看起来美观。

PS：

```
function serialize (o)
    if type(o) == "number" then
        io.write(o)
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    elseif type(o) == "table" then
        io.write("{\n")
        for k,v in pairs(o) do
            io.write(" ", k, " = ")
            serialize(v)
            io.write(",\n")
        end
    io.write("}\n")
    else
        error("cannot serialize a " .. type(o))
    end
```

哔哩哔哩次元游戏人生

```
end
```

尽管代码很简单，但很好地解决了问题。只要表结构是一个树型结构（也就是说，没有共享的子表并且没有循环），上面代码甚至可以处理嵌套表（表中表）。对于所进不整齐的表我们可以少作改进使结果更美观，这可以作为一个练习尝试一下。（提示：增加一个参数表示缩进的字符串，来进行序列化）。前面的函数假定表中出现的所有关键字都是合法的标示符，如果表中有不符合 Lua 语法的数字关键字或者字符串关键字，上面的代码将碰到麻烦。

一个简单的解决这个难题的方法是将 io.write(" ", k, " = ")改为：

PS：

```
io.write(" [")
serialize(k)
io.write("] = ")
```

这样一来，我们改善了我们的函数的健壮性，比较一下两次的结果。

PS：

```
-- result of serialize{a=12, b='Lua', key='another "one"'}
--  第一个版本
{
a = 12,
b = "Lua",
key = "another \"one\"",
}
--  第二个版本
{
["a"] = 12,
["b"] = "Lua",
["key"] = "another \"one\"",
}
```

我们可以通过测试每一种情况，看是否需要方括号，另外，我们将这个问题留作一个练习给大家。

2） 保存带有循环的 table

针对普通拓扑概念上的带有循环表和共享子表的 table，我们需要另外一种不同的方法来处理。构造器不能很好地解决这种情况，我们不使用。为了表示循环我们需要将表名记录下来，下面我们的函数有两个参数，table 和对应的名字。另外，我们还必须记录已经保存过的 table 以防止由于循环而被重复保存。我们使用一个额外的 table 来记录保存过的表的轨迹，这个表的下表索引为 table，而值为对应的表名。

我们做一个限制，要保存的 table 只有一个字符串或者数字关键字。下面的这个函数序列化基本类型并返回结果。

PS：

```
function basicSerialize (o)
    if type(o) == "number" then
        return tostring(o)
    else -- assume it is a string
        return string.format("%q", o)
    end
```

```
end
```

关键内容在接下来的这个函数，saved 这个参数是上面提到的记录已经保存的表的踪迹的 table。

PS：

```
function save (name, value, saved)
    saved = saved or {} -- initial value
    io.write(name, " = ")
    if type(value) == "number" or type(value) == "string" then
        io.write(basicSerialize(value), "\n")
    elseif type(value) == "table" then
        if saved[value] then -- value already saved?
            -- use its previous name
            io.write(saved[value], "\n")
        else
            saved[value] = name -- save name for next time
            io.write("{}\n") -- create a new table
            for k,v in pairs(value) do -- save its fields
                local fieldname = string.format("%s[%s]", name,
                                                basicSerialize(k))
                save(fieldname, v, saved)
            end
        end
    else
        error("cannot save a " .. type(value))
    end
end
```

如果我们将要保存的 table 为。

PS：

```
a = {x=1, y=2; {3,4,5}}
a[2] = a     -- cycle
a.z = a[1]     -- shared sub-table
```

调用 save('a', a)之后结果为。

PS：

```
a = {}
a[1] = {}
a[1][1] = 3
a[1][2] = 4
a[1][3] = 5
a[2] = a
a["y"] = 2
a["x"] = 1
a["z"] = a[1]
```

（实际的顺序可能有所变化，它依赖于 table 遍历的顺序，不过，这个算法保证了一个新的定义中需要的前面的节点都已经被定义过）

哔哩哔哩次元游戏人生

如果我们想保存带有共享部分的表，我们可以使用同样 table 的 saved 参数调用 save 函数，例如我们创建下面两个表。

PS：

```
a = {{"one", "two"}, 3}
b = {k = a[1]}
```

接着保存它们。

PS：

```
save('a', a)
save('b', b)
```

结果将分别包含相同部分：

PS：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = {}
b["k"][1] = "one"
b["k"][2] = "two"
```

然而如果我们使用同一个 saved 表来调用 save 函数。

PS：

```
local t = {}
save('a', a, t)
save('b', b, t)
```

结果将共享相同部分。

PS：

```
a = {}
a[1] = {}
a[1][1] = "one"
a[1][2] = "two"
a[2] = 3
b = {}
b["k"] = a[1]
```

上面这种方法是 Lua 中常用的方法，当然也有其他一些方法可以解决问题。比如，我们可以不使用全局变量名来保存，即使用封包，用 chunk 构造一个 local 值然后返回之；通过构造一张表，每张表名与其对应的函数对应起来等。Lua 给予你权力，由你决定如何实现。

十四、 环境

Lua 用一个名为 environment 普通的表来保存所有的全局变量。（更精确的说，Lua 在一系列的 environment 中保存他的“global”变量，但是我们有时候可以忽略这种多样性）这种结果的优点之一是他简化了 Lua 的内部实现，因为对于所有的全局变量没有必要非要有不同的数据结构。另一个(主要的)优点是我们可以像其他表一样操作这个保存全局变量的表。为了简化操作，Lua 将环境本身存储在一个全局变量_G 中，（_G._G 等于_G）。例

哔哩哔哩次元游戏人生

如，下面代码打印在当前环境中所有的全局变量的名字。

```
for n in pairs(_G) do print(n) end
```

这一章我们将讨论一些如何操纵环境的有用的技术。

1. 使用动态名字访问全局变量

通常，赋值操作对于访问和修改全局变量已经足够。然而，我们经常需要一些元编程（meta-programming）的方式，比如当我们需要操纵一个名字被存储在另一个变量中的全局变量，或者需要在运行时才能知道的全局变量。为了获取这种全局变量的值，有的程序员可能写出下面类似的代码。

PS：

```
loadstring("value = " .. varname)()
```

or

```
value = loadstring("return " .. varname)()
```

如果 varname 是 x，上面连接操作的结果为："return x"（第一种形式为 "value = x"），当运行时才会产生最终的结果。然而这段代码涉及到一个新的 chunk 的创建和编译以及其他很多额外的问题。你可以换种方式更高效更简洁的完成同样的功能，代码如下。

PS：

```
value = _G[varname]
```

因为环境是一个普通的表，所以你可以使用你需要获取的变量（变量名）索引表即可。

也可以用相似的方式对一个全局变量赋值，_G[varname] = value。小心：一些程序员对这些函数很兴奋，并且可能写出这样的代码：_G["a"] = _G["var1"]，这只是 a = var1 的复杂的写法而已。

对前面的问题概括一下，表域可以是型如"io.read" or "a.b.c.d"的动态名字。我们用循环解决这个问题，从_G 开始，一个域一个域的遍历。

PS：

```
function getfield (f)
    local v = _G    -- start with the table of globals
    for w in string.gfind(f, "[%w_]+") do
    v = v[w]
    end
    return v
end
```

我们使用 string 库的 gfind 函数来迭代 f 中的所有单词（单词指一个或多个子母下线的序列）。相对应的，设置一个域的函数稍微复杂些。

PS：

```
a.b.c.d.e = v
```

实际等价于。

```
local temp = a.b.c.d
temp.e = v
```

也就是说，我们必须记住最后一个名字，必须独立的处理最后一个域。新的 setfield 函数，当其中的域不存在的时候还需要创建中间表。

PS：

```
function setfield (f, v)
    local t = _G -- start with the table of globals
```

```
        for w, d in string.gfind(f, "([%w_]+)(.?)") do
            if d == "." then -- not last field?
                t[w] = t[w] or {} -- create table if absent
                t = t[w] -- get the table
            else -- last field
                t[w] = v -- do the assignment
            end
        end
    end
```

这个新的模式匹配以变量 w 加上一个可选的点（保存在变量 d 中）的域。如果一个域名后面不允许跟上点，表明它是最后一个名字。

PS：

```
setfield("t.x.y", 10)
```

创建一个全局变量表 t，另一个表 t.x，并且对 t.x.y 赋值为 10。

```
print(t.x.y)     --> 10
print(getfield("t.x.y"))     --> 10
```

2. 声明全局变量

全局变量不需要声明，虽然这对一些小程序来说很方便，但程序很大时，一个简单的拼写错误可能引起 bug 并且很难发现。然而，如果我们喜欢，我们可以改变这种行为。

因为 Lua 所有的全局变量都保存在一个普通的表中，我们可以使用 metatables 来改变访问全局变量的行为。

第一个方法代码如下所示。

PS：

```
setmetatable(_G, {
    __newindex = function (_, n)
    error("attempt to write to undeclared variable "..n, 2)
    end,

    __index = function (_, n)
        error("attempt to read undeclared variable "..n, 2)
    end,
})
```

这样一来，任何企图访问一个不存在的全局变量的操作都会引起错误。

```
> a = 1
stdin:1: attempt to write to undeclared variable a
```

但是我们如何声明一个新的变量呢？使用 rawset，可以绕过 metamethod。

```
function declare (name, initval)
    rawset(_G, name, initval or false)
end
```

or 带有 false 是为了保证新的全局变量不会为 nil。

注意：你应该在安装访问控制以前（before installing the access control）定义这个函数，否则将得到错误信息，毕竟你是在企图创建一个新的全局声明。只要刚才那个函数在正确的地方，你就可以控制你的全局变量了。

PS：

哔哩哔哩次元游戏人生

```
a = 1
stdin:1: attempt to write to undeclared variable a
declare "a"
a = 1        -- OK
```

但是现在，为了测试一个变量是否存在，我们不能简单的比较他是否为 nil。如果他是 nil 访问将抛出错误。所以，我们使用 rawget 绕过 metamethod。

```
if rawget(_G, var) == nil then
    -- 'var' is undeclared
    ...
End
```

改变控制允许全局变量可以为 nil 也不难，所以我们需要的是创建一个辅助表用来保存所有已经声明的变量的名字。不管什么时候 metamethod 被调用的时候，他会检查这张辅助表看变量是否已经存在。

PS：

```
local declaredNames = {}
function declare (name, initval)
    rawset(_G, name, initval)
    declaredNames[name] = true
end
setmetatable(_G, {
    __newindex = function (t, n, v)
    if not declaredNames[n] then
        error("attempt to write to undeclared var. "..n, 2)
    else
        rawset(t, n, v) -- do the actual set
    end
end,
    __index = function (_, n)
    if not declaredNames[n] then
        error("attempt to read undeclared var. "..n, 2)
    else
        return nil
    end
end,
})
```

两种实现方式，代价都很小可以忽略不计的。第一种解决方法，metamethods 在平常操作中不会被调用。第二种解决方法，他们可能被调用，不过当且仅当访问一个值为 nil 的变量时。

3. 非全局的环境

全局环境的一个问题是，任何修改都会影响你的程序的所有部分。例如，当你安装一个 metatable 去控制全局访问时，你的整个程序都必须遵循同一个指导方针。如果你想使用标准库，标准库中可能使用到没有声明的全局变量，你将碰到坏的结果。

Lua 5.0 允许每个函数可以有自己的环境来改善这个问题，听起来这很奇怪；毕竟，全局变量表的目的就是为了全局性使用。然而在 Section 15.4 我们将看到这个机制带来很多有

哔哩哔哩次元游戏人生

趣的结构，全局的值依然是随处可以获取的。

可以使用 setfenv 函数来改变一个函数的环境。Setfenv 接受函数和新的环境作为参数。除了使用函数本身，还可以指定一个数字表示栈顶的活动函数。数字 1 代表当前函数，数字 2 代表调用当前函数的函数（这对写一个辅助函数来改变他们调用者的环境是很方便的）依此类推。

应用 setfenv 失败的例子。

PS：

```
a = 1 -- create a global variable
-- change current environment to a new empty table
setfenv(1, {})
print(a)
```

导致：stdin:5: attempt to call global `print' (a nil value)

（你必须在单独的 chunk 内运行这段代码，如果你在交互模式逐行运行他，每一行都是一个不同的函数，调用 setfenv 只会影响他自己的那一行。）一旦你改变了你的环境，所有全局访问都使用这个新的表，如果她为空，你就丢失所有你的全局变量，甚至_G，所以，你应该首先使用一些有用的值封装（populate）她，比如老的环境。

PS：

```
a = 1 -- create a global variable
-- change current environment
setfenv(1, {_G = _G})
_G.print(a) --> nil
_G.print(_G.a)     --> 1
```

现在，当你访问"global" _G，他的值为旧的环境，其中你可以使用 print 函数，也可以使用继承封装(populate)你的新的环境。

PS：

```
a = 1
local newgt = {}      -- create new environment
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)     -- set it
print(a)     -->1
```

在这段代码新的环境从旧的环境中继承了 print 和 a；然而，任何赋值操作都对新表进行，不用担心误操作修改了全局变量表。另外，你仍然可以通过_G 修改全局变量。

PS：

```
-- continuing previous code
a = 10
print(a)     --> 10
print(_G.a)     --> 1
_G.a = 20
print(_G.a)     --> 20
```

当你创建一个新的函数时，他从创建他的函数中继承了环境变量。所以，如果一个 chunk 改变了他自己的环境，这个 chunk 所有在改变之后定义的函数都共享相同的环境，都会受到影响。

哔哩哔哩次元游戏人生

十五、 模块与包

1. 模块

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。以下为创建自定义模块 module.lua，文件代码格式如下：

PS：

```
-- 文件名为 module.lua
-- 定义一个名为 module 的模块
module = {}

-- 定义一个常量
module.constant = "这是一个常量"

-- 定义一个函数
function module.func1()
    io.write("这是一个公有函数！\n")
end

local function func2()   --定义一个私有函数
    print("这是一个私有函数！")
end

function module.func3()
    func2()
end

return module
```

由上面的代码可知，模块(module)的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。

上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的公有函数来调用。

2. require 函数

Lua 提供了一个名为 require 的函数用来加载模块。要加载一个模块，只需要简单地调用就可以了。

PS：

require("<模块名>")或者 require "<模块名>"

执行 require 后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

1） 直接调用模块

PS：

哔哩哔哩次元游戏人生

```
-- module  模块为上面提到的 module.lua
require("module")
print(module.constant)
module.func3()
```

执行结果为：

这是一个常量

这是一个私有函数！

2） 通过别名调用

上面展示了如何直接调用模块，我们也可以给加载的模块定义一个别名变量，方便后续的调用。

PS：

```
-- module  模块为上文提到到  module.lua
--  别名变量  m
local m = require("module")   --定义一个局部变量去存储模块
print(m.constant)
m.func3()
```

其最终输出结果，和上面直接调用一致。

3） 加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开.profile 文件（没有则创建，打开.bashrc 文件也可以），例如把 "~/lua/" 路径加入 LUA_PATH 环境变量里：

```
#LUA_PATH
export LUA_PATH="~/lua/?.lua;;"
```

文件路径以 ";" 号分隔，最后的 2 个 ";;" 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效。

```
source ~/.profile
```

这时假设 package.path 的值是：

/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/usr/local/lib/lua/5.1/?.lua;/usr/local/lib/lua/5.1/?/init.lua

那么调用 require("module") 时就会尝试打开以下文件目录去搜索目标。

/Users/dengjoe/lua/module.lua;

./module.lua

/usr/local/share/lua/5.1/module.lua

/usr/local/share/lua/5.1/module/init.lua

/usr/local/lib/lua/5.1/module.lua

/usr/local/lib/lua/5.1/module/init.lua

如果找过目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。

哔哩哔哩沃克游戏人生

搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA_CPATH 来初始。

搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

3. C 包

Lua 和 C 是很容易结合的，使用 C 为 Lua 写包。

与 Lua 中写包不同，C 包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制。

Lua 在一个叫 loadlib 的函数内提供了所有的动态连接的功能。这个函数有两个参数，库的绝对路径和初始化函数。所以典型的调用的例子如下：

PS：

```
local path = "/usr/local/lua/lib/libluasocket.so"
local f = loadlib(path, "luaopen_socket")
```

loadlib 函数加载指定的库并且连接到 Lua，然而它并不打开库（也就是说没有调用初始化函数），反之他返回初始化函数作为 Lua 的一个函数，这样我们就可以直接在 Lua 中调用他。

如果加载动态库或者查找初始化函数时出错，loadlib 将返回 nil 和错误信息。我们可以修改前面一段代码，使其检测错误然后调用初始化函数：

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- 或者 path = "C:\\windows\\luasocket.dll"，这是 Window 平台下
local f = assert(loadlib(path, "luaopen_socket"))
f()  -- 真正打开库
```

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的 stub 文件，安装二进制库的时候可以随便放在某个目录，只需要修改 stub 文件对应二进制库的实际路径即可。

将 stub 文件所在的目录加入到 LUA_PATH，这样设定后就可以使用 require 函数加载 C 库了。

十六、 Package

很多语言专门提供了某种机制组织全局变量的命名，比如 Modula 的 modules，Java 和 Perl 的 packages，C++的 namespaces。每一种机制对在 package 中声明的元素的可见性以及其他一些细节的使用都有不同的规则。但是他们都提供了一种避免不同库中命名冲突的问题的机制。每一个程序库创建自己的命名空间，在这个命名空间中定义的名字和其他命名空间中定义的名字互不干涉。

Lua 并没有提供明确的机制来实现 packages。然而，我们通过语言提供的基本的机制很容易实现他。主要的思想是：像标准库一样，使用表来描述 package。

使用表实现 packages 的明显的好处是：我们可以像其他表一样使用 packages，并且可以使用语言提供的所有的功能，带来很多便利。大多数语言中，packages 不是第一类值 (first-class values)（也就是说，他们不能存储在变量里，不能作为函数参数。。。）因此，这些语言需要特殊的方法和技巧才能实现类似的功能。

Lua 中，虽然我们一直都用表来实现 packages，但也有其他不同的方法可以实现 package。

1. 基本方法

包的简单方法，是在包内的每一个对象前都加包名作为前缀。例如，假定我们正在写

一个操作复数的库，我们使用表来表示复数，表有两个域 r（实数部分）和 i（虚数部分），我们在另一张表中声明我们所有的操作来实现一个包。

PS：

```
complex = {}
function complex.new (r, i) return {r=r, i=i} end
-- defines a constant `i'
complex.i = complex.new(0, 1)
function complex.add (c1, c2)
    return complex.new(c1.r + c2.r, c1.i + c2.i)
end
function complex.sub (c1, c2)
    return complex.new(c1.r - c2.r, c1.i - c2.i)
end
function complex.mul (c1, c2)
    return complex.new(c1.r*c2.r - c1.i*c2.i,
        c1.r*c2.i + c1.i*c2.r)
end
function complex.inv (c)
    local n = c.r^2 + c.i^2
    return complex.new(c.r/n, -c.i/n)
end
return complex
```

这个库定义了一个全局名：complex，其他的定义都是放在这个表内。有了上面的定义，我们就可以使用符合规范的任何复数操作了。

PS：

```
c = complex.add(complex.i, complex.new(10, 20))
```

这种使用表来实现的包和真正的包的功能并不完全相同。首先，我们对每一个函数定义都必须显示的在前面加上包的名称。第二，同一包内的函数相互调用必须在被调用函数前指定包名。我们可以使用固定的局部变量名，来改善这个问题，然后，将这个局部变量赋值给最终的包。依据这个原则，我们重写上面的代码。

PS：

```
local P = {}
complex = P     -- package name
P.i = {r=0, i=1}
function P.new (r, i) return {r=r, i=i} end
function P.add (c1, c2)
    return P.new(c1.r + c2.r, c1.i + c2.i)
end
```

当在同一个包内的一个函数调用另一个函数的时候（或者调用自身），他仍然需要加上前缀名。至少，它不再依赖于固定的包名。另外，只有一个地方需要包名。可能你注意到包中最后一个语句 return complex。

这个 return 语句并非必需的，因为 package 已经赋值给全局变量 complex 了。但是，我们认为 package 打开的时候返回本身是一个很好的习惯。额外的返回语句并不会花费什么代价，并且提供了另一种操作 package 的可选方式。

哔哩哔哩次元游戏人生

2. 私有成员-Privacy

有时候，一个 package 会公开他的所有内容，也就是说，任何 package 的客户端都可以访问他。然而，一个 package 也拥有自己的私有部分（也就是只有 package 本身才能访问）也是很有用的。在 Lua 中一个传统的方法是将私有部分定义为局部变量来实现。例如，我们修改上面的例子增加私有函数来检查一个值是否为有效的复数。

PS：

```
local P = {}
complex = P
local function checkComplex (c)
    if not ((type(c) == "table") and
    tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
end
function P.add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return P.new(c1.r + c2.r, c1.i + c2.i)
end
...
return P
```

1） 优点

package 中所有的名字都在一个独立的命名空间中。Package 中的每一个实体（entity）都清楚地标记为公有还是私有。另外，我们实现了一个真正的私有（privacy），私有实体在 package 外部是不可访问的。

2） 缺点

访问同一个 package 内的其他公有的实体写法冗余，必须加上前缀 P.。还有一个大的问题是，当我们修改函数的状态(公有变成私有或者私有变成公有)我们必须修改函数的调用方式。

有一个有趣的方法可以立刻解决上述缺点，我们可以将 package 内的所有函数都声明为局部的，最后将他们放在最终的表中。按照这种方法，上面的 complex package 将可以修改成下面的代码。

PS：

```
local function checkComplex (c)
    if not ((type(c) == "table")
    and tonumber(c.r) and tonumber(c.i)) then
        error("bad complex number", 3)
    end
end
local function new (r, i) return {r=r, i=i} end
local function add (c1, c2)
    checkComplex(c1);
    checkComplex(c2);
    return new(c1.r + c2.r, c1.i + c2.i)
```


哔哩哔哩次元游戏人生

```
end
...
complex = {
    new = new,
    add = add,
    sub = sub,
    mul = mul,
    div = div,
}
```

现在我们调用函数的时候就不再需要加上前缀了，公有的和私有的函数调用方法相同。在 package 的结尾处，有一个简单的列表列出所有公有的函数。可能大多数人觉得这个列表放在 package 的开始处更自然，但我们不能这样做，因为我们必须首先定义局部函数。

3. 包与文件

我们经常写一个 package 然后将所有的代码放到一个单独的文件中。然后我们只需要执行这个文件即加载 package。例如，如果我们将上面我们的复数的 package 代码放到一个文件 complex.lua 中，命令“require complex”将打开这个 package。记住 require 命令不会将相同的 package 加载多次。

需要注意的问题是，搞清楚保存 package 的文件名和 package 名的关系。当然，将他们联系起来是一个好的想法，因为 require 命令使用文件而不是 packages。一种解决方法是在 package 的后面加上后缀（比如.lua）来命名文件。Lua 并不需要固定的扩展名，而是由你的路径设置决定。例如，如果你的路径包含："/usr/local/lualibs/?.lua"，那么复数 package 可能保存在一个 complex.lua 文件中。

有些人喜欢先命名文件后命名 package。也就是说，如果你重命名文件，package 也会被重命名。这个解决方法提供了很大的灵活性。例如，如果你有两个有相同名称 package，你不需要修改任何一个，只需要重命名一下文件。在 Lua 中我们使用_REQUIREDNAME 变量来重命名。记住，当 require 加载一个文件的时候，它定义了一个变量来表示虚拟的文件名。

PS：

```
local P = {} -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
End
```

代码中的 if 测试使得我们可以不需要 require 就可以使用 package。如果_REQUIREDNAME 没有定义，我们用固定的名字表示 package（例子中 complex）。另外，package 使用虚拟文件名注册他自己。如果使用者将库放到文件 cpx.lua 中并且运行 require cpx，那么 package 将本身加载到表 cpx 中。如果其他的使用者将库改名为 cpx_v1.lua 并且运行 require cpx_v1，那么 package 将自动将本身加载到表 cpx_v1 当中。

4. 使用全局表

上面这些创建 package 的方法的缺点是，他们要求程序员注意很多东西。比如，在声明的时候也很容易忘掉 local 关键字。全局变量表的 Metamethods 提供了一些有趣的技术，也可以用来实现 package。这些技术中共同之处在于，package 使用独占的环境。这很容易

实现，如果我们改变了 package 主 chunk 的环境，那么由 package 创建的所有函数都共享这个新的环境。

最简单的技术实现，一旦 package 有一个独占的环境，不仅她的所有函数共享环境，并且它的所有全局变量也共享这个环境。所以，我们可以将所有的公有函数声明为全局变量，然后他们会自动作为独立的表（表指 package 的名字）存在，所有 package 必须要做的是将这个表注册为 package 的名字。下面这段代码阐述了复数库使用这种技术的结果。

PS：

```
local P = {}
complex = P
setfenv(1, P)
```

现在，当我们声明函数 add，她会自动变成 complex.add。

```
function add (c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end
```

另外，我们可以在这个 package 中不需要前缀调用其他的函数。例如，add 函数调用 new 函数，环境会自动转换为 complex.new。这种方法提供了对 package 很好的支持，程序员几乎不需要做什么额外的工作，调用同一个 package 内的函数不需要前缀，调用公有和私有函数也没什么区别。如果程序员忘记了 local 关键字，也不会污染全局命名空间，只不过使得私有函数变成公有函数而已。另外，我们可以将这种技术和前一节我们使用的 package 名的方法组合起来。

PS：

```
local P = {}     -- package
if _REQUIREDNAME == nil then
    complex = P
else
    _G[_REQUIREDNAME] = P
end
setfenv(1, P)
```

向上面这样就不能访问其他的 packages 了，一旦我们将一个空表 P 作为我们的环境，我们就不能访问以前的所有全局变量了。下面有好几种方法可以解决这个问题，但都各有利弊。

最简单的解决方法是使用继承，像前面我们看到的一样。

PS：

```
local P = {}     -- package
setmetatable(P, {__index = _G})
setfenv(1, P)
```

（你必须在调用 setfenv 之前调用 setmetatable，你能说出原因么？）使用这种结构，package 就可以直接访问所有的全局标示符，但必须为每一个访问付出一小点代价。理论上来讲，这种解决方法带来一个有趣的结果，你的 package 现在包含了所有的全局变量。例如，使用你的 package 也可以调用标准库的 sin 函数：complex.math.sin(x)。

另外一种快速的访问其他 packages 的方法是声明一个局部变量来保存老的环境。

PS：

```
local P = {}
pack = P
```

哔哩哔哩次元游戏人生

```
local _G = _G
setfenv(1, P)
```

现在，你必须对外部的访问加上前缀_G.，但是访问速度更快，因为这不涉及到 metamethod。与继承不同的是这种方法，使得你可以访问老的环境；这种方法的好与坏是有争议的，但是有时候你可能需要这种灵活性。

一个更加正规的方法是，只把你需要的函数或者 packages 声明为 local。

PS：

```
local P = {}
pack = P
local sqrt = math.sqrt
local io = io
-- no more external access after this point
setfenv(1, P)
```

这一技术要求稍多，但他使你的 package 的独立性比较好，他的速度也比前面那几种方法快。

十七、 元表-Metatable

在 Lua table 中我们可以访问对应的 key 来得到 value 值，但是却无法对两个 table 进行操作(比如相加)。

因此 Lua 提供了元表(Metatable)，允许我们改变 table 的行为，每个行为关联了对应的元方法。

例如，使用元表我们可以定义 Lua 如何计算两个 table 的相加操作 a+b。

当 Lua 试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个叫 __add 的字段，若找到，则调用对应的值。 __add 等即时字段，其对应的值（往往是一个函数或是 table）就是"元方法"。

有两个很重要的函数来处理元表：

1. setmetatable(table,metatable)

对指定 table 设置元表(metatable)，如果元表(metatable)中存在 __metatable 键值，setmetatable 会失败。

2. getmetatable(table)

返回对象的元表(metatable)。

PS：

```
mytable = {}                          -- 普通表
mymetatable = {}                      -- 元表
setmetatable(mytable,mymetatable)     -- 把 mymetatable 设为 mytable 的元表
```

以上代码也可以直接写成一行：

```
mytable = setmetatable({},{})
```

下面的代码为返回对象元表：

```
getmetatable(mytable)                 -- 这会返回 mymetatable
```

3. __index 元方法

这是元表(metatable)最常用的键。

当你通过键来访问 table 的时候，如果这个键没有值，那么 Lua 就会寻找该 table 的 metatable（假定有 metatable）中的__index 键。如果__index 包含一个表格，Lua 会在表格中查找相应的键。

我们可以在使用 lua 命令进入交互模式查看：

PS：

```
other = { foo = 3 }
t = setmetatable({}, { __index = other })
t.foo
```

如果__index 包含一个函数的话，Lua 就会调用那个函数，table 和键会作为参数传递给函数。

__index 元方法查看表中元素是否存在，如果不存在，返回结果为 nil；如果存在则由__index 返回结果。

PS：

```
mytable = setmetatable({key1 = "value1"},
        {
    __index = function(mytable, key)
        If( key == "key2")
        then
            return "metatablevalue"
        else
            return nil
        end
    end
        })
print(mytable.key1,mytable.key2)
```

上述代码，将 mytable 表赋值为 {key1 = "value1"}。

mytable 设置了元表，元方法为 __index。

在 mytable 表中查找 key1，如果找到，返回该元素，找不到则继续。

在 mytable 表中查找 key2，如果找到，返回 metatablevalue，找不到则继续。

判断元表有没有__index 方法，如果__index 方法是一个函数，则调用该函数。

元方法中查看是否传入 "key2" 键的参数（mytable.key2 已设置），如果传入 "key2" 参数返回 "metatablevalue"，否则返回 mytable 对应的键值。

我们可以将以上代码写得更为简洁。

PS：

```
mytable = setmetatable({key1 = "value1"}, { __index = { key2 = "metatablevalue" } })
print(mytable.key1,mytable.key2)
```

1） 总结

Lua 查找一个表元素时的规则，其实就是如下 3 个步骤:

① 在表中查找，如果找到，返回该元素，找不到则继续。

② 判断该表是否有元表，如果没有元表，返回 nil，有元表则继续。

③ 判断元表有没有__index 方法，如果__index 方法为 nil，则返回 nil；如果__index 方法是一个表，则重复 1、2、3；如果__index 方法是一个函数，则返回该函数的返回值。

4. __newindex 元方法

__newindex 元方法用来对表进行更新，__index 则用来对表进行访问 。

当表中的一个索引赋值，解释器就会查找__newindex 元方法：如果存在则调用这个函

哔哩哔哩次元游戏人生

数而不进行赋值操作。

下面的示例演示了 __newindex 元方法的应用：

PS：

```
mymetatable = {}
--使用 setmetatable 设置元表
mytable = setmetatable({key1 = "value1"}, { __newindex = mymetatable })

print(mytable.key1)
--在对新索引键（newkey）赋值时，会调用元方法，而不进行赋值。
mytable.newkey = "新值 2"
print(mytable.newkey,mymetatable.newkey)
--而如果对已存在的索引键（key1），则会进行赋值，而不调用元方法 __newindex。
mytable.key1 = "新值 1"
print(mytable.key1,mymetatable.key1)
```

上述示例中为表设置了元方法 __newindex，在对新索引键（newkey）赋值时（mytable.newkey = "新值 2"），会调用元方法，而不进行赋值。而如果对已存在的索引键（key1），则会进行赋值，而不调用元方法 __newindex。

下面的示例使用了 rawset 函数来更新表：

PS：

```
mytable = setmetatable({key1 = "value1"},
            {
        __newindex = function(mytable, key, value)   --这是元方法__newindex
            rawset(mytable, key, "\""..value.."\"")
        end
})

mytable.key1 = "new value"
mytable.key2 = 4

print(mytable.key1,mytable.key2)
```

5. 为表添加操作符

下面的代码演示了两表的相加操作。

PS：

```
-- 计算表中最大值，table.maxn 在 Lua5.2 以上版本中已无法使用
-- 自定义计算表中最大键值函数 table_maxn，即计算表的元素个数
function table_maxn(table)
    local maxnumber = 0
    for key, value in pairs(table)
    do
        if (maxnumber < key)
        then
            maxnumber = key
        end
    end
```

哔哩哔哩次元游戏人生

```
        return maxnumber
    end

    -- 两表相加操作
    mytable = setmetatable({ 1, 2, 3 },
                {
        __add = function(mytable, newtable)
                for i = 1, table_maxn(newtable)
                do
                        --在 table 的数组部分指定位置(pos)插入值为 newtable[i]的一个元素，其他元素向后移动。
                        table.insert(mytable, table_maxn(mytable)+1,newtable[i])
                end
                return mytable
        end
    })

    secondtable = {4,5,6}

    mytable = mytable + secondtable--对表执行相加操作
    for key,value in ipairs(mytable)
    do
        print(key,value)
    end
```

模式	描述
__add	对应的运算符 '+'
__sub	对应的运算符 '-'
__mul	对应的运算符 '*'
__div	对应的运算符 '/'
__mod	对应的运算符 '%'
__unm	对应的运算符 '-'
__concat	对应的运算符 '..'
__eq	对应的运算符 '=='
__lt	对应的运算符 '<'
__le	对应的运算符 '<='

6. __call 元方法

__call 元方法在 Lua 调用一个值时调用。

下面的示例演示了如何计算表中元素的和。

PS：

```
-- 计算表中最大值，table.maxn 在 Lua5.2 以上版本中已无法使用
-- 自定义计算表中最大键值函数 table_maxn，即计算表的元素个数
function table_maxn(t)
    local maxNumber = 0
```


哔哩哔哩次元游戏人生

```
    for key, value in pairs(t)
    do
        if maxNumber < key
            then
            maxNumber = key
        end
    end
    return maxNumber
end
--setmetatable 就是对指定的 table 设置元表
mytable = setmetatable({10},
        {
    __call = function(mytable, newtable)--定义元方法__call，会在调用一个值时调用。
        sum = 0
        for i = 1, table_maxn(mytable)    --table_maxn 计算表中的元素个数
        do
            sum = sum + mytable[i]
        end
        for i = 1, table_maxn(newtable)    --table_maxn 计算表中的元素个数
        do
            sum = sum + newtable[i]
        end
        return sum
    end
})
newtable = {10,20,30}
print(mytable(newtable))
```

7. __tostring 元方法

__tostring 元方法用于修改表的输出行为。

下面的示例自定义了表的输出内容。

PS：

```
--setmetatable 就是对指定的 table 设置元表
mytable = setmetatable ({ 10, 20, 30 },
        {
    __tostring = function(mytable)
        sum = 0
        for key, value in pairs(mytable)
        do
            sum = sum + value
        end
        return "表所有元素的和为 " .. sum
    end
})
print(mytable)
```

哔哩哔哩次元游戏人生

十八、 协同程序-coroutine

1. 什么是协同(coroutine)

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

协同是非常强大的功能，但是用起来也很复杂。

2. 线程和协同程序区别

线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

3. 基本语法

方法	描述
coroutine.create()	创建 coroutine，返回 coroutine，参数是一个函数，当和 resume 配合使用的时候就唤醒函数调用。
coroutine.resume()	重启 coroutine，和 create 配合使用。操作成功返回 true，否则返回 false。
coroutine.yield()	挂起 coroutine，将 coroutine 设置为挂起状态，这个和 resume 配合使用能有很多有用的效果。
coroutine.status()	查看 coroutine 的状态 注：coroutine 的状态有三种：dead，suspended，running，具体什么时候有这样的状态请参考下面的程序。
coroutine.wrap()	创建 coroutine，返回一个函数，一旦你调用这个函数，就进入 coroutine，和 create 功能重复。
coroutine.running()	返回正在跑的 coroutine，一个 coroutine 就是一个线程，当使用 running 的时候，就是返回一个 corouting 的线程号。

下面的示例演示了上表中各个方法的用法。

PS：

```
--创建一个 coroutine
co = coroutine.create(
        function(i)
            print(i);
        end
)

coroutine.resume(co, 1)     -- 输出结果为 1
print(coroutine.status(co))   -- 输出结果为 dead
```

哔哩哔哩次元游走人生

```
print("----------")
--创建 coroutine 和 creat 功能一样
co = coroutine.wrap(
        function(i)
            print(i);
        end
)

co(1)--使用协同程序

print("----------")

co2 = coroutine.create(
        function()
            for i=1,10
            do
                print(i)
                if i == 3
                then
                    print(coroutine.status(co2))    --running
                    print(coroutine.running()) --thread:XXXXXX
                end
                coroutine.yield()--挂起协程
            end
        end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))     -- suspended
print(coroutine.running())--因为协程已被挂起，因此线程号为 nil

print("----------")
```

从 coroutine.running 就可以看出来,coroutine 在底层实现就是一个线程。

当 create 一个 coroutine 的时候就是在新线程中注册了一个事件。

当使用 resume 触发事件的时候，create 的 coroutine 函数就被执行了，当遇到 yield 的时候就代表挂起当前线程，等候再次 resume 触发事件。

接下来我们分析一个更详细的实例。

PS：

```
function foo (a)
    print("foo  函数输出", a)
    return coroutine.yield(2 * a) --  返回   2*a  的值
```

哔哩哔哩次元游戏人生

```
end
--创建一个协程
co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)
    --因为在 foo 函数中挂起了协程，因此就不会继续执行后面的内容，除非重新启动协程
    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)   -- a，b 的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                          -- b 的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

首先我们调用 resume，将协同程序唤醒，resume 操作成功返回 true，否则返回 false；接着协同程序开始运行，运行到 yield 语句时，yield 会挂起协同程序，第一次 resume 返回；（注意：此处 yield 返回，参数是 resume 的参数）

第二次 resume，再次唤醒协同程序；（注意：此处 resume 的参数中，除了第一个参数，剩下的参数将作为 yield 的参数），yield 返回。

如果使用的协同程序继续运行完成后继续调用 resume 方法则输出：cannot resume dead coroutine。

resume 和 yield 的配合强大之处在于，resume 处于主程中，它将外部状态（数据）传入到协同程序内部；而 yield 则将内部的状态（数据）返回到主程中。

十九、文件 I/O

Lua I/O 库用于读取和处理文件。分为简单模式（和 C 一样）、完全模式。

简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。

完全模式（complete model）使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法

简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。

PS：

```
file = io.open (filename [, mode])     --  打开文件的操作语句
```

1. 模式值

模式	描述
r	以只读方式打开文件，该文件必须存在。
w	打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF 符保留）
r+	以可读写方式打开文件，该文件必须存在。
w+	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
a+	与 a 类似，但此文件可读可写
b	二进制模式，如果文件是二进制文件，可以加上 b
+	加号表示对文件既可以读也可以写

2. 简单模式

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

以下为 file.lua 文件代码，操作的文件为 test.lua(如果没有你需要创建该文件)，代码如下。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
-- 设置默认输入文件为 test.lua
io.input(file)
-- 输出文件第一行
print(io.read())
-- 关闭打开的文件
io.close(file)
-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")
-- 设置默认输出文件为 test.lua
io.output(file)
-- 在文件最后一行添加 Lua 注释
io.write("--  test.lua 文件末尾注释")
-- 关闭打开的文件
io.close(file)
```

模式	描述
"*n"	读取一个数字并返回它。例：file.read("*n")

哔哩哔哩次元游学生

"*a"	从当前位置读取整个文件。例：file.read("*a")
"*l"（默认）	读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("*l")
number	返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5)

1） 其他 IO 方法有：

a. io.tmpfile()

返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除。

b. io.type(file)

检测 obj 是否一个可用的文件句柄。

c. io.flush()

向文件写入缓冲中的所有数据。

d. io.lines(optional file name)

返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回 nil,但不关闭文件。

3. 完全模式

通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name 来代替 io.function_name 方法。以下实例演示了如何同时处理同一个文件。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
-- 输出文件第一行
print(file:read())
-- 关闭打开的文件
file:close()
-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")
-- 在文件最后一行添加 Lua 注释
file:write("--test")
-- 关闭打开的文件
file:close()
```

上述代码的 read 的参数与简单模式一致。

哔哩哔哩次元游戏人生

1） file:seek(optional whence, optional offset)

设置和获取当前文件位置,成功则返回最终的文件位置(按字节),失败则返回 nil 加错误信息。参数 whence 值可以是：

"set"：从文件头开始。

"cur"：从当前位置开始[默认]。

"end"：从文件尾开始。

offset：默认为 0。

不带参数 file:seek()就会返回当前位置，file:seek("set")则是定位到文件头，file:seek("end")则定位到文件尾并返回文件大小。

2） file:flush()

向文件写入缓冲中的所有数据

3） io.lines(optional file name)

打开指定的文件 filename 为读模式并返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回 nil,并自动关闭文件。

若不带参数时 io.lines() <=> io.input():lines(); 读取默认输入设备的内容，但结束时不关闭文件。

```
for line in io.lines("main.lua")
    do
    print(line)
    end
```

以下实例使用了 seek 方法，定位到文件倒数第 25 个位置并使用 read 方法的*a 参数，即从当期位置(倒数第 25 个位置)读取整个文件。

PS：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")
file:seek("end",-25)
print(file:read("*a"))
-- 关闭打开的文件
file:close()
```

二十、 错误处理

程序运行中错误处理是必要的，在我们进行文件操作，数据转移及 web service 调用过程中都会出现不可预期的错误。如果不注重错误信息的处理，就会造成信息泄露，程序无法运行等情况。

1. 语法错误

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。

PS：

```
a  ==  2
```

以上代码执行结果为：

lua: test.lua:2: syntax error near '=='。

正如你所看到的，以上出现了语法错误，一个"="号跟两个"="号是有区别的。一个"=" 是赋值表达式，而两个 "=" 是比较运算。

PS：

```
for a= 1,10
```

```
    print(a)
end
```

上述代码会报如下错误：

lua: test2.lua:2: 'do' expected near 'print'

语法错误比程序运行错误更简单，运行错误无法定位具体错误，而语法错误我们可以很快的解决，上面的示例我们只要在 for 语句下添加 do 即可。

2. 运行错误

运行错误是程序可以正常执行，但是会输出报错信息。

下面的示例由于参数输入错误，程序在执行时会报错。

PS：

```
function add(a,b)
    return a+b
end

add(10)
```

上述代码在编译时是可以成功的，但是在运行的时候会产生下面的错误。

PS：

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
    test2.lua:2: in function 'add'
    test2.lua:5: in main chunk
    [C]: ?
```

lua 里调用函数时，即使实参列表和形参列表不一致也能成功调用，多余的参数会被舍弃，缺少的参数会被补为 nil。

以上报错信息是由于参数 b 被补为 nil 后，nil 参与了+运算。

假如 add 函数内不是 "return a+b" 而是 "print(a,b)" 的话，结果会变成 "10 nil" 不会报错。

3. 错误处理

我们可以使用两个函数，即 assert 和 error 来处理错误。

1） assert

PS：

```
local function add(a,b)
    assert(type(a) == "number", "a 不是一个数字")
    assert(type(b) == "number", "b 不是一个数字")
    return a+b
end
add(10)
```

当我们执行上述代码时，会出现下面的错误。

```
lua: test.lua:3: b 不是一个数字
stack traceback:
    [C]: in function 'assert'
    test.lua:3: in local 'add'
    test.lua:6: in main chunk
    [C]: in ?
```


哔哩哔哩次元游戏人生

实例中 assert 首先检查第一个参数，若没问题，assert 不做任何事情；否则，assert 以第二个参数作为错误信息抛出。

2） error

语法格式如下。

PS：error (message [, level])

error 的功能是终止正在执行的函数，并返回 message 的内容作为错误信息(error 函数永远都不会返回)

通常情况下，error 会附加一些错误位置的信息到 message 头部。

Level 参数指示获得错误的位置:

Level=1：这是默认的，为调用 error 位置(文件+行号)

Level=2：指出哪个调用 error 的函数的函数

Level=0：不添加错误位置信息

3） pcall 和 xpcall、debug

Lua 中处理错误，可以使用函数 pcall（protected call）来包装需要执行的代码。

pcall 接收一个函数和要传递给后者的参数，并执行，执行结果：有错误、无错误；返回值 true 或者为 false， errorinfo。

语法格式如下。

PS：

```
If   pcall(function_name, ….)
then
-- 没有错误
else
-- 一些错误
end
```

下面是 pcall 函数的一个实际案例。

PS：

```
print(pcall(function(i) print(i) end, 33))
```

输出结果为 33 true。

pcall 以一种"保护模式"来调用第一个参数，因此 pcall 可以捕获函数执行中的任何错误。

通常在错误发生时，希望落得更多的调试信息，而不只是发生错误的位置。但 pcall 返回时，它已经销毁了调用栈的部分内容。

Lua 提供了 xpcall 函数，xpcall 接收第二个参数—— 一个错误处理函数，当错误发生时，Lua 会在调用栈展开（unwind）前调用错误处理函数，于是就可以在这个函数中使用 debug 库来获取关于错误的额外信息了。

debug 库提供了两个通用的错误处理函数：

① debug.debug

提供一个 Lua 提示符，让用户来检查错误的原因。

② debug.traceback

根据调用栈来构建一个扩展的错误消息。

二十一、 调试-Debug

Lua 提供了 debug 库用于提供创建我们自定义调试器的功能。Lua 本身并未有内置的调试器，但很多开发者共享了他们的 Lua 调试器代码。

哔哩哔哩次元游戏人生

序号	方法	用途
1	debug():	进入一个用户交互模式，运行用户输入的每个字符串。使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。输入一行仅包含 cont 的字符串将结束这个函数， 这样调用者就可以继续向下运行。
2	getfenv(object):	返回对象的环境变量。
3	gethook(optional thread):	返回三个表示线程钩子设置的值：当前钩子函数，当前钩子掩码，当前钩子计数
4	getinfo ([thread,] f [, what]):	返回关于一个函数信息的表。你可以直接提供该函数，也可以用一个数字 f 表示该函数。数字 f 表示运行在指定线程的调用栈对应层次上的函数：0 层表示当前函数（getinfo 自身）；1 层表示调用 getinfo 的函数（除非是尾调用，这种情况不计入栈）；等等。如果 f 是一个比活动函数数量还大的数字，getinfo 返回 nil。
5	debug.getlocal ([thread,] f, local):	此函数返回在栈的 f 层处函数的索引为 local 的局部变量 的名字和值。这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。
6	getmetatable(value):	把给定索引指向的值的元表压入堆栈。如果索引无效，或是这个值没有元表，函数将返回 0 并且不会向栈上压任何东西。
7	getregistry():	返回注册表表，这是一个预定义出来的表，可以用来保存任何 C 代码想保存的 Lua 值。
8	getupvalue (f, up)	此函数返回函数 f 的第 up

哔哩哔哩次元游戏人生

		个上值的名字和值。如果该函数没有那个上值，返回 nil。 以'('（开括号）打头的变量名表示没有名字的变量（去除了调试信息的代码块）。
9	sethook ([thread,] hook, mask [, count]):	将一个函数作为钩子函数设入。字符串 mask 以及数字 count 决定了钩子将在何时调用。掩码是由下列字符组合成的字符串，每个字符有其含义： 'c': 每当 Lua 调用一个函数时，调用钩子； 'r': 每当 Lua 从一个函数内返回时，调用钩子； 'l': 每当 Lua 进入新的一行时，调用钩子。
10	setlocal ([thread,] level, local, value):	这个函数将 value 赋给栈上第 level 层函数的第 local 个局部变量。如果没有那个变量，函数返回 nil。如果 level 越界，抛出一个错误。
11	setmetatable (value, table):	将 value 的元表设为 table（可以是 nil）。返回 value。
12	setupvalue (f, up, value):	这个函数将 value 设为函数 f 的第 up 个上值。如果函数没有那个上值，返回 nil 否则，返回该上值的名字。
13	traceback ([thread,] [message [, level]]):	如果 message 有，且不是字符串或 nil， 函数不做任何处理直接返回 message。否则，它返回调用栈的栈回溯信息。字符串可选项 message 被添加在栈回溯信息的开头。数字可选项 level 指明从栈的哪一层开始回溯（默认为 1，即调用 traceback 的那里）。

哔哩哔哩次元游戏人生

二十二、 垃圾回收

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来，也不用考虑在对象不再被使用后怎样释放它们所占用的内存。

Lua 运行了一个垃圾收集器来收集所有死对象（即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。Lua 中所有用到的内存，如字符串、表、用户数据、函数、线程、内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。它使用这两个数字来控制垃圾收集循环：垃圾收集器间歇率和垃圾收集器步进倍率。这两个数字都使用百分数为单位（例如：值 100 在内部表示 1）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。增大这个值会减少收集器的积极性。当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。设置这个值为 200 就会让收集器等到总内存使用量达到之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。不要把这个值设得小于 100，那样的话收集器就工作的太慢了以至于永远都干不完一个循环。默认值是 200，这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字（比你的程序可能用到的字节数还大 10% ）， 收集器的行为就像一个 stop-the-world 收集器。接着你若把间歇率设为 200，收集器的行为就和过去的 Lua 版本一样了：每次 Lua 使用的内存翻倍时，就做一次完整的收集。

1. 垃圾回收器函数

Lua 提供了以下函数 collectgarbage ([opt [, arg]])用来控制自动内存管理:

1） `collectgarbage("collect")`

做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：

2） `collectgarbage("count")`

以 K 字节数为单位返回 Lua 使用的总内存数。这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。

3） `collectgarbage("restart")`

重启垃圾收集器的自动运行。

4） `collectgarbage("setpause")`

将 arg 设为收集器的间歇率。返回间歇率的前一个值。

collectgarbage("setpause", 200)，表示内存增大 2 倍（200/100）时自动释放一次内存（200 是默认值）。

5） `collectgarbage("setstepmul")`

返回步进倍率的前一个值。

collectgarbage("setstepmul", 200)，收集器单步收集的速度相对于内存分配速度的倍率，设置 200 时倍率等于 2 倍（200/100）。（200 是默认值）

6） `collectgarbage("step")`

单步运行垃圾收集器。步长"大小"由 arg 控制。传入 0 时，收集器步进（不可分割的）一步。传入非 0 值，收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。如果收集器结束一个循环将返回 true。

7） collectgarbage("stop")

停止垃圾收集器的运行。在调用重启前，收集器只会因显式的调用运行。

下面的代码演示了一个简单的垃圾回收示例。

PS：

mytable = {"apple", "orange", "banana"}—这是一个直接初始表

print(collectgarbage("count"))—输出结果为 20.9560546875

mytable = nil --删除表

print(collectgarbage("count"))—输出结果为 20.9853515625

print(collectgarbage("collect"))—输出结果为 0

print(collectgarbage("count"))—输出结果为 19.4111328125

二十三、 面向对象程序设计

Lua 中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表(Table)也有：

PS：

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
End
```

上面代码定义并创建了一个新的函数，并且保存在 Account 对象的 withdraw 域内，下面我们可以这样调用：

PS：

```
Account.withdraw(100.00)
```

这种函数就是我们所谓的方法，然而，在一个函数内部使用全局变量名 Account 是一个不好的习惯。首先，这个函数只能在这个特殊的对象（这里指 Account）中使用；第二，即使对这个特殊的对象而言，这个函数也只有在对象被存储在特殊的变量（指 Account）中才可以使用。如果我们改变了这个对象的名字，函数 withdraw 将不能工作：

PS：

```
a = Account;
Account = nil
a.withdraw(100.00) -- ERROR!
```

这种行为违背了前面的对象应该有独立的生命周期的原则。

一个灵活的方法是：定义方法的时候带上一个额外的参数，来表示方法作用的对象。这个参数经常为 self 或者 this。

PS：

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

现在，当我们调用这个方法的时候不需要指定他操作的对象了。

PS：

```
a1 = Account; Account = nil
...
a1.withdraw(a1, 100.00) -- 这是可行的
```

哔哩哔哩次元游戏人生

使用 self 参数定义函数后，我们可以将这个函数用于多个对象上。

PS：

```
a2 = {balance=0, withdraw = Account.withdraw}
...
a2.withdraw(a2, 260.00)
```

self 参数的使用是很多面向对象语言的要点。大多数语言将这种机制隐藏起来，这样程序员不必声明这个参数（虽然仍然可以在方法内使用这个参数）。Lua 也提供了通过使用冒号操作符来隐藏这个参数的声明。我们可以重写上面的代码：

PS：

```
function Account:withdraw (v)
  self.balance = self.balance - v
end
a:withdraw(100.00)
```

冒号的效果相当于在函数定义和函数调用的时候，增加一个额外的隐藏参数。这种方式只是提供了一种方便的语法，实际上并没有什么新的内容。我们可以使用 dot 语法定义函数而用冒号语法调用函数，反之亦然，只要我们正确的处理好额外的参数。

PS：

```
Account = {
  balance=0,
  withdraw = function (self, v)
  self.balance = self.balance - v
end
}
function    Account:deposit (v)
  self.balance = self.balance + v
end
Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

现在我们的对象拥有一个标示符，一个状态和操作这个状态的方法。但他们依然缺少一个 class 系统，继承和隐藏。先解决第一个问题：我们如何才能创建拥有相似行为的多个对象呢？明确地说，我们怎样才能创建多个 accounts？（针对上面的对象 Account 而言）

1. 类

一些面向对象的语言中提供了类的概念，作为创建对象的模板。在这些语言里，对象是类的实例。Lua 不存在类的概念，每个对象定义他自己的行为并拥有自己的形状（shape）。然而，依据基于原型（prototype）的语言比如 Self 和 NewtonScript，在 Lua 中仿效类的概念并不难。在这些语言中，对象没有类。相反，每个对象都有一个 prototype（原型），当调用不属于对象的某些操作时，会最先会到 prototype 中查找这些操作。在这类语言中实现类（class）的机制，我们创建一个对象，作为其它对象的原型即可（原型对象为类，其它对象为类的 instance）。类与 prototype 的工作机制相同，都是定义了特定对象的行为。

在 Lua 中，使用继承的思想，很容易实现 prototypes。更明确的来说，如果我们有两个对象 a 和 b，我们想让 b 作为 a 的 prototype 只需要：

PS：

```
setmetatable(a, {__index = b})
```

这样，对象 a 调用任何不存在的成员都会到对象 b 中查找。术语上，可以将 b 看作类，a 看作对象。回到前面银行账号的例子上。为了使得新创建的对象拥有和 Account 相似的行为，我们使用__index metamethod，使新的对象继承 Account。注意一个小的优化：我们不需要创建一个额外的表作为 account 对象的 metatable；我们可以用 Account 表本身作为 metatable。

PS：

```
function Account:new (o)
    o = o or {} -- create object if user does not provide one
    setmetatable(o, self)
    self.__index = self
    return o
end
```

（当我们调用 Account:new 时，self 等于 Account；因此我们可以直接使用 Account 取代 self。然而，使用 self 在我们下一节介绍类继承时更合适）。有了这段代码之后，当我们创建一个新的账号并且调用一个方法的时候，有什么发生呢？

PS：

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

当我们创建这个新的账号 a 的时候，a 将 Account 作为他的 metatable（调用 Account:new 时，self 即 Account）。当我们调用 a:deposit(100.00)，我们实际上调用的是 a.deposit(a,100.00)（冒号仅仅是语法上的便利）。然而，Lua 在表 a 中找不到 deposit，因此他回到 metatable 的__index 对应的表中查找，情况大致如下：

PS：

```
getmetatable(a).__index.deposit(a, 100.00)
```

a 的 metatable（元表）是 Account，Account.__index 也是 Account（因为 new 函数中 self.__index = self）。所以我们可以重写上面的代码为。

PS：

```
Account.deposit(a, 100.00)
```

也就是说，Lua 传递 a 作为 self 参数，然后调用原始的 deposit 函数。所以，新的账号对象从 Account 继承了 deposit 方法。使用同样的机制，可以从 Account 继承所有的域。继承机制不仅对方法有效，对表中所有的域都有效。所以，一个类不仅提供方法，也提供了他的实例的成员的默认值。

注意：

在我们第一个 Account 定义中，我们提供了成员 balance 默认值为 0，所以，如果我们创建一个新的账号而没有提供 balance 的初始值，他将继承默认值。

PS：

```
b = Account:new()
print(b.balance)        --> 0
```

当我们调用 b 的 deposit 方法时，实际等价于 b.balance = b.balance + v

（因为 self 就是 b）。表达式 b.balance 等于 0 并且初始的存款（b.balance）被赋予 b.balance。下一次我们访问这个值的时候，不会在涉及到 index metamethod，因为 b 已经存在他自己的 balance 域。

2. 继承

通常的面向对象语言中，继承是使得类可以访问其他类的方法，这在 Lua 中也很容易

实现，下面我们就假定有一个基类 Account。

PS：

```
Account = {balance = 0} –这是一个直接初始表
function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
return o
end
function Account:deposit (v)
  self.balance = self.balance + v
end
function Account:withdraw (v)
      if v > self.balance
            then
            error"insufficient funds"
      end
      self.balance = self.balance - v
end
```

我们打算从基类派生出一个子类 SpecialAccount，这个子类允许客户取款超过它的存款余额限制，我们从一个空类开始，从基类继承所有操作。

PS：

```
SpecialAccount = Account:new()
s = SpecialAccount:new{limit=1000.00}
```

SpecialAccount 从 Account 继承了 new 方法，当 new 执行的时候，self 参数指向 SpecialAccount。所以，s 的 metatable 是 SpecialAccount，__index 也是 SpecialAccount。这样，s 继承了 SpecialAccount，后者继承了 Account。

PS：

```
s:deposit(100.00)
```

当我们执行上述代码，Lua 在 s 中找不到 deposit 域，他会到 SpecialAccount 中查找，在 SpecialAccount 中找不到，会到 Account 中查找。使得 SpecialAccount 的特殊之处在于，它可以重定义从父类中继承来的方法。

PS：

```
function SpecialAccount:withdraw (v)
      if v - self.balance >= self:getLimit()
            then
            error"insufficient funds"
      end
      self.balance = self.balance - v
end
function SpecialAccount:getLimit ()
      return self.limit or 0
end
```

现在，当我们调用方法 s:withdraw(200.00)，Lua 不会到 Account 中查找，因为它第一

哔哩哔哩次元游戏人生

次就在 SpecialAccount 中发现了新的 withdraw 方法，由于 s.limit 等于 1000.00（记住，我们创建 s 的时候初始化了这个值）程序执行了取款操作，s 的 balance 变成了负值。

在 Lua 中面向对象有趣的一个方面是你不需要创建一个新类去指定一个新的行为。如果仅仅一个对象需要特殊的行为，你可以直接在对象中实现，例如，如果账号 s 表示一些特殊的客户：取款限制是他的存款的 10%，你只需要修改这个单独的账号：

PS：

```
function s:getLimit ()
    return self.balance * 0.10
end
```

这样声明之后，调用 s:withdraw(200.00)将运行 SpecialAccount 的 withdraw 方法，但是当方法调用 self:getLimit 时，我们新定义的 getLimit 被触发。

3. 多重继承

由于 Lua 中的对象不是原生(primitive)的，所以在 Lua 中有很多方法可以实现面向对象的程序设计。我们前面所见到的使用 index metamethod 的方法可能是简洁、性能、灵活各方面综合最好的。然而，针对一些特殊情况也有更适合的实现方式，下面我们在 Lua 中实现多重继承。

实现的关键在于：将函数用作__index。记住，当一个表的 metatable 存在一个__index 函数时，如果 Lua 调用一个原始表中不存在的函数，Lua 将调用这个__index 指定的函数。这样可以用__index 实现在多个父类中查找子类不存在的域。

多重继承意味着一个类拥有多个父类，所以，我们不能用创建一个类的方法去创建子类。取而代之的是，我们定义一个特殊的函数 createClass 来完成这个功能，将被创建的新类的父类作为这个函数的参数。这个函数创建一个表来表示新类，并且将它的 metatable 设定为一个可以实现多继承的__index metamethod。尽管是多重继承，每一个实例依然属于一个在其中能找得到它需要的方法的单独的类。所以，这种类和父类之间的关系与传统的类与实例的关系是有区别的。特别是，一个类不能同时是其实例的 metatable 又是自己的 metatable。在下面的实现中，我们将一个类作为他的实例的 metatable，创建另一个表作为类的 metatable。

PS：

```
-- look up for `k' in list of tables 'plist'
local function search (k, plist)
    for i=1, table.getn(plist)
    do
    local v = plist[i][k] -- try 'i'-th superclass
    if v
        then
        return v
        end
    end
end
function createClass (...)
local c = {} -- new class
-- class will search for each method in the list of its
-- parents (`arg' is the list of parents)
 setmetatable(c, {__index = function (t, k)
```

```
return search(k, arg)
end})
-- prepare `c' to be the metatable of its instances
c.__index = c
-- define a new constructor for this new class
function c:new (o)
  o = o or {}
  setmetatable(o, c)
return o
end
-- return new class
return c
end
```

让我们用一个小例子阐明一下 createClass 的使用，假定我们前面的类 Account 和另一个类 Named，Named 只有两个方法 setname and getname。

PS：

```
Named = {} --这是一个空表
function Named:getname ()
    return self.name
end
function Named:setname (n)
    self.name = n
end
```

为了创建一个继承于这两个类的新类，我们将调用 createClass。

PS：

```
NamedAccount = createClass(Account, Named)
```

为了创建和使用实例，我们将像往常一样设置。

PS：

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())      --> Paul
```

现在我们看看上面最后一句发生了什么，Lua 在 account 中找不到 getname，因此他查找 account 的 metatable 的__index，即 NamedAccount。但是，NamedAccount 也没有 getname，因此 Lua 查找 NamedAccount 的 metatable 的__index，因为这个域包含一个函数，Lua 调用这个函数并首先到 Account 中查找 getname，没有找到，然后到 Named 查找，找到并返回最终的结果。当然，由于搜索的复杂性，多重继承的效率比起单继承要低。一个简单的改善性能的方法是将继承方法拷贝到子类，使用这种技术，index 方法如下。

PS：

```
setmetatable(c, {__index = function (t, k)
    local v = search(k, arg)
    t[k] = v -- save for next access
    return v
end})
```

应用这个技巧，访问继承的方法和访问局部方法一样快（特别是第一次访问）。缺点是

系统运行之后，很难改变方法的定义，因为这种改变不能影响继承链的下端。

4. 私有性-privacy

很多人认为私有性是面向对象语言的应有的一部分。每个对象的状态应该是这个对象自己的事情。在一些面向对象的语言中，比如 C++和 Java 你可以控制对象成员变量或者成员方法是否私有。其他一些语言比如 Smalltalk 中，所有的成员变量都是私有，所有的成员方法都是公有的。第一个面向对象语言 Simula 不提供任何保护成员机制。

如前面我们所看到的，Lua 中的主要对象设计不提供私有性访问机制。部分原因是因为，我们使用通用数据结构 tables 来表示对象的结果。但是，这也反映了 Lua 的设计思想，Lua 没有打算被用来进行大型的程序设计，相反，Lua 目标定于小型到中型的程序设计，通常是作为大型系统的一部分。典型的，被一个或者很少几个程序员开发，甚至被非程序员使用。所以，Lua 避免太冗余和太多的人为限制，如果你不想访问一个对象内的一些东西就不要访问。

然而，Lua 的另一个目标是灵活性，提供程序员元机制（meta-mechanisms），通过它你可以实现很多不同的机制。虽然 Lua 中基本的面向对象设计并不提供私有性访问的机制，我们可以用不同的方式来实现他。虽然这种实现并不常用，但知道他也是有益的，不仅是因为它展示了 Lua 的一些有趣的角落，也因为它可能是某些问题的很好地解决方案。

设计的基本思想是，每个对象用两个表来表示：一个描述状态；另一个描述操作（或者叫接口）。对象本身通过第二个表来访问，也就是说，通过接口来访问对象。为了避免未授权的访问，表示状态的表中不涉及到操作；表示操作的表也不涉及到状态，取而代之的是，状态被保存在方法的闭包内。例如，用这种设计表述我们的银行账号，我们使用下面的函数工厂创建新的对象。

PS：

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}
    local withdraw = function (v)
        self.balance = self.balance - v
    end
    local deposit = function (v)
        self.balance = self.balance + v
    end
    local getBalance = function () return self.balance end
    return {
        withdraw = withdraw,
        deposit [1]= deposit,
        getBalance = getBalance
    }
end
```

首先，函数创建一个表用来描述对象的内部状态，并保存在局部变量 self 内。然后，函数为对象的每一个方法创建闭包（也就是说，嵌套的函数实例）。最后，函数创建并返回外部对象，外部对象中将局部方法名指向最终要实现的方法。这儿的关键点在于，这些方法没有使用额外的参数 self，代替的是直接访问 self。因为没有这个额外的参数，我们不能使用冒号语法来访问这些对象，函数只能像其他函数一样调用。

[1] v. 放下，放置；储蓄；存放，寄存；沉积，沉淀；n. 沉积物，沉积层；订金；押金；存款；

PS：

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())   --输出结果为 60
```

这种设计，使得任何存储在 self 表中的部分都是私有的，newAccount 返回之后，没有什么方法可以直接访问对象，我们只能通过 newAccount 中定义的函数来访问他。虽然我们的例子中仅仅将一个变量放到私有表中，但是我们可以将对象的任何的部分放到私有表中。

我们也可以定义私有方法，他们看起来像是公有的(Public)，但我们并不将其放到接口中。例如，我们的账号可以给某些用户取款享有额外的 10%的存款上限，但是我们不想用户直接访问这种计算的详细信息，我们实现如下。

PS：

```
function newAccount (initialBalance)
    local self = {
        balance = initialBalance,
        LIM = 10000.00,
    }
    local extra = function ()
        if self.balance > self.LIM then
            return self.balance*0.10
        else
            return 0
        end
    end
    local getBalance = function ()
        return self.balance + self.extra()
    end
```

像上面这样，用户就没有办法直接访问 extra 函数了。

5. Single-Method 的对象实现方法

前面的程序设计的方法有一种特殊情况：对象只有一个单一的方法。这种情况下，我们不需要创建一个接口表，取而代之的是，我们将这个单一的方法作为对象返回。这听起来有些不可思议，其实，一个保存状态的迭代子函数就是一个 single-method 对象。

PS：

```
function newObject (value)
    return function (action, v)
        if action == "get" then return value
        elseif action == "set" then value = v
        else error("invalid action")
        end
    end
end
```

single-method 对象使用起来很简单。

PS：

```
d = newObject(0)
```

哔哩哔哩次元游戏人生

```
print(d("get"))     --输出结果为 0
d("set", 10)
print(d("get"))     --输出结果为 10
```

这种非传统的对象实现是非常有效的，语法 d("set",10)虽然很罕见，但也只不过比传统的 d:set(10)长两个字符而已。每一个对象是用一个单独的闭包，代价比起表来小的多。这种方式没有继承但有私有性，访问对象状态的唯一方式是通过它的内部方法。

Tcl/Tk 的窗口部件（widgets）使用了相似的方法，在 Tk 中一个窗口部件的名字表示一个在窗口部件上执行各种可能操作的函数（a widget command）。

二十四、 优化

1. 尽可能使用局部变量

应该尽可能的使用局部变量，这可以避免命名冲突；访问局部变量的速度比全局变量更快。

二十五、 技巧

对于 lua 中的 Table，我们可以直接使用连接字符去循环获取对应的字段。

```
for i=0 , j do
table["GameObject"..i]
end
```

二十六、 工具

1. IntelliJ IDEA

1） 快捷键

快捷键	介绍
Ctrl+ F	在当前文件进行文本查找
Ctrl + R	在当前文件进行文本替换
Ctrl + Z	撤销
Ctrl + Y	删除光标所在行 或 删除选中的行
Ctrl + X	剪切光标所在行 或 剪切选择内容
Ctrl + /	注释光标所在行代码，会根据当前不同文件类型使用不同的注释符号
Ctrl + C	复制光标所在行 或 复制选择内容
Ctrl + D	复制光标所在行 或 复制选择内容，并把复制内容插入光标位置下面
Ctrl + W	递进式选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展选中范围
Ctrl + O	选择可重写的方法
Ctrl + I	选择可继承的方法
Ctrl + +	展开代码
Ctrl + -	折叠代码
Ctrl + [	移动光标到当前所在代码的花括号开始位置
Ctrl +]	移动光标到当前所在代码的花括号结束位置

哔哩哔哩次元游戏人生

Ctrl + Enter	智能分隔行
Ctrl + End	跳到文件尾
Ctrl + Home	跳到文件头
Ctrl + Alt + L	格式化代码，可以对当前文件和整个包目录使用
Ctrl + Alt + O	优化导入的类，可以对当前文件和整个包目录使用
Ctrl + Alt + 左方向键	退回到上一个操作的地方
Ctrl + Alt + 右方向键	前进到上一个操作的地方
Ctrl + Shift + F	根据输入内容查找整个项目 或 指定目录内文件
Ctrl + Shift + R	根据输入内容替换对应内容，范围为整个项目 或 指定目录内文件
Ctrl + Shift + J	自动将下一行合并到当前行末尾
Ctrl + Shift + Z	取消撤销
Ctrl + Shift + W	递进式取消选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展取消选中范围
Ctrl + Shift + N	通过文件名定位 / 打开文件 / 目录，打开目录需要在输入的内容后面多加一个正斜杠
Ctrl + Shift + U	对选中的代码进行大 / 小写轮流转换
Ctrl + Shift + Enter	自动结束代码，行末自动添加分号
F2	跳转到下一个高亮错误 或 警告位置
F3	在查找模式下，定位到下一个匹配处

2） 插件

Chinese：中文语言包，使用后界面文字将变为中文。

EmmyLua：用于 Lua 开发。

Save Action：自动格式化，安装后在 Save Action 的 General 和 Formatting Actions 中勾选。

Rainbow Brackets：使用 ctrl+鼠标右键单击，使最近的括号内容获得高亮效果；alt+鼠标右键单击，使最近的括号内容外获得暗淡效果。

CodeGlance：代码迷你缩放图。

Translation：内置翻译。

CamelCase：命名风格转换插件，可以在 kebab-case，SNAKE_CASE，PascalCase，camelCase，snake_case 和空格风格之间切换。苹果快捷键为⇧+⌥+ U，windows 为 Shift + Alt +U。

3） 常规设置

a. IDEA 设置智能提示忽略大小写

哔哩哔哩次元游戏人生

b. IDEA 设置鼠标滚轮操作

c. IDEA 以新窗口的形式打开多个项目

d. IDEA 忽略某个文件或者文件夹

e. IDEA 统一编辑文件编码

f. 去掉 IntelliJ IDEA 的拼写检查

2. EmmyLua

4) 注解功能

a. @class

EmmyLua 利用 @class 注解来模拟面向对象中的类，可以继承，可以定义字段/属性。

完整格式：--@class MY_TYPE[:PARENT_TYPE] [@comment]

b. @type

利用 @type 注解来标记目标变量的类型，以增强代码提示以及其它功能。

完整格式：---@type MY_TYPE[|OTHER_TYPE] [@comment]

应用目标

① Local 变量

PS：

```
---@type Car @instance of car
local car1 = {}

---@type Car|Ship @transport tools, car or ship. Since lua is dynamic-typed, a variable may be of different types
---use | to list all possible types
```

哔哩哔哩次元游戏人生

local transport = {}

② Global 变量

PS：

---@type Car @global variable type

global_car = {}

③ Property 属性

PS：

local obj = {}

---@type Car @property type

obj.car = getCar()

c. @alias

可以使用 @alias 将一些复杂不容易输入的类型注册为一个新的别名。

完整格式：---@alias NEW_NAME TYPE

PS：

---@alias Handler fun(type: string, data: any):void

---@param handler Handler

function addHandler(handler)

end

d. @param

利用 @param 注解来标记函数定义参数的类型，以增强代码提示以及其它功能。

完整格式：---@param param_name MY_TYPE[|other_type] [@comment]

应用目标

① 函数参数

PS：

---@param car Car

local function setCar(car)

...

end

PS：

---@param car Car

setCallback(function(car)

...

end)

② For 循环参数

PS：

---@param car Car

for k, car in ipairs(list) do

end

哔哩哔哩次元游戏人生

e. @return

利用 @return 注解来标记函数的返回值类型。

完整格式：---@return MY_TYPE[|OTHER_TYPE] [@comment]

应用目标

① 函数

PS：

```
---@return Car|Ship
local function create()
    ...
end

---Here car_or_ship doesn't need @type annotation, EmmyLua has already inferred the type via "create" function
local car_or_ship = create()
```

PS：

```
---@return Car
function factory:create()
    ...
end
```

f. @field

利用 @field 注解来标记某个类的额外的属性（即使这个属性没有出现在代码里）。

完整格式：---@field [public|protected|private] field_name FIELD_TYPE[|OTHER_TYPE] [@comment]

应用目标

① 在 @class 注解之后

PS：

```
---@class Car
---@field public name string @add name field to class Car, you'll see it in code completion
local cls = class()
```

g. @generic

利用 @generic 注解来模拟高级语言中的 泛型

完整格式：--@generic T1 [: PARENT_TYPE] [, T2 [: PARENT_TYPE]]

应用目标

① 函数

PS：

```
---@generic T : Transport, K
---@param param1 T
```

哔哩哔哩次元游戏人生

```
---@param param2 K
---@return T
local function test(param1, param2)
    -- todo
end

---@type Car
local car = ...

local value = test(car)
```

h. @vararg

使用 @vararg 注解一个函数的不定参数部分的类型。
完整格式：---@vararg TYPE
PS：

```
---@vararg string
---@return string
local function format(...)
    local tbl = { ... } -- inferred as string[]
end
```

i. @language

可以利用 @language 的方式来标注一段文本为某种代码格式，从而可以显示高亮。
完整格式：---@language LANGUAGE_ID
PS：

```
---@language JSON
local jsonText = [[{
    "name":"Emmy"
}]]
```

哔哩哔哩次元游戏人生

Unity

一、 PipelineAsset

1. Quality

1） HDR

HDR 是 High Dynamic Range 的缩写，即高动态范围。动态范围指的是最高和最低亮度值的比值。在真实世界中，一个场景中最亮和最暗区域的范围可以非常大，例如，太阳发出的光可能要比场景中某个影子上的点的亮度高出数万倍，这些范围远远超过图像或显示器能够显示的范围。

通常在显示设备使用的颜色缓冲中每个通道的精度为 8 位，意味着我们只能使用这 256 种（即 2^8）不同的颜色来表示真实世界中的所有亮度，所以在虚拟世界中显示颜色会损失一定的精度。早期的拍摄设备利用人眼的特点，使用了伽马曲线来对捕捉到的图像进行编码，尽可能充分地利用这些有限的存储空间。

HDR 的出现给我们带来了新的希望，HDR 使用远远高于 8 位的精度（如 32 位）来记录亮度信息，使得我们可以表示超过 0-1 内的亮度值，从而可以更加精确地反映真实的光照环境。虽然在最后我们依旧需要把信息转换到显示设备使用的 LDR 内，但是中间的计算却可以让我们得到更加真实可信的效果。

NVIDIA 曾总结过使用 HDR 进行渲染的动机：让亮的物体可以真的非常亮，暗的物体可以真的非常暗，同时又可以看到两者之间的细节。

使用 HDR 来存储的图像被称为高动态范围图像（HDRI），例如使用一张 HDRI 图像来作为场景的天空盒（Skybox），可以更加真实地反映物体周围的环境，从而得到更加真实的反射效果。

HDR 对与光照的叠加也有非常重要的作用。如果我们的场景中有很多光源获胜光源强度很大，那么一个物体经过多次光照渲染叠加后最终得到的光照亮度很可能会超过 1.如果没有使用 HDR，超过 1 的部分会被截取掉，使得场景丢失了很多亮部区域细节。一旦开启了 HDR，我们就可以保留这些超过范围的光照结果，尽管最后我们仍然，需要把它们转换到 LDR 进行显示，但我们可以使用色调映射（tonemapping）技术来控制这个转换的过程，从而允许我们最大限度地保留需要的亮度细节。

HDR 的使用可以允许我们在屏幕后处理中拥有更多控制权。例如，Bloom 效果需要检测屏幕中亮度大于某个阈值的像素，把它们提取出来后进行模糊，再叠加到原图像中。如果我们不使用 HDR，就只能使用小于 1 的阈值来提取需要的像素，但很多时候我们实际上是需要提取那些非常亮的区域，比如车窗上的强烈的反光。如果不使用 HDR，这些值实际上可能和街道上一些颜色偏白的区域一样，使应保持阴暗的地方也出现泛光的效果。而使用 HDR，这些问题都可以得到解决，我们只需使用超过 1 的阈值来提取那些非常亮的区域即可。

使用 HDR 可以让我们不会丢失高亮度区域的颜色值，提供了更真实的光照效果，并为一些屏幕后处理提供更多的控制能力。但是 HDR 也有缺点，由于使用了浮点缓冲来存储高精度图像，不仅需要更大的显存空间，渲染速度也会变慢，除此之外，一些硬件并不支持 HDR。而且一旦使用 HDR，我们就无法使用硬件的抗锯齿功能。

在 Unity 中如果我们同时打开了硬件的抗锯齿（在 Edit -> Project Settings -> Qualitty -> Anti Aliasing 中打开）和摄像机的 HDR，Unity 会向我们发出警告，提醒我们因为开启了抗锯齿而无法再使用 HDR 缓冲。（我们可以基于屏幕后处理的抗锯齿操作来弥补这一点）

2） Anti Aliasing(抗锯齿)

更高等级的抗锯齿，所消耗的性能就会越多

2. Lighting

1） Shadow Resolution（阴影解析度）

阴影解析度设置得更高，所消耗的性能就会越多。

3. Shadows

1） MaxDistance（阴影可见的最大距离）

如果设置为 40，则在距离为 40 内阴影就会显现出来。

2） Cascade Count（渲染层级数）

设置为 2 时阴影会被分成两段进行渲染，在特定距离内渲染的阴影质量会有所不同。

3） Soft Shadows（软阴影）

开启软阴影将会使得阴影显示得更加的柔和。

二、 Lighting 窗口

1. Environment

1） Environment Lightning

① Source

设置环境光的光源，可选择的有 Skybox（由天空盒决定）、Color（纯色）

2. Other Settings

1） Fog 开启雾效

① Color

设置雾效颜色。

② Mode

Linear-雾密度随距离线性增加。

Exponential-雾密度随距离呈指数增长。

Exponential Squared-雾密度随距离增加更快（指数和平方）。

③ Density

控制雾的密度。随着设定密度值的增加，雾会看起来更浓。

2） Halo Texture

设置要用于围绕灯光绘制光晕的纹理。

3） Halo Strength

从 0 到 1 之间的值定义光晕周围的可见性。

4） Flare Fade Speed

定义时间（以秒为单位） 镜头光晕

最初出现后从视野中消失。默认设置为 3。

5） Flare Strength

从 0 到 1 之间的值定义来自灯光的镜头光晕的可见性。

哔哩哔哩次元游戏人生

6） Spot Cookie

三、 Navigation 窗口

3. Agents（代理类型）

1） Name

代理类型预设名称。

2） Raius

代理的半径，用于计算 ,碰撞在障碍物和其他代理之间。

3） Height

4） Step Height

5） Max Slope

斜坡的最大角度，大于该角度的地区将无法到达。

四、 Nav Mesh Agent

要在脚本中使用需要使用命名空间，using UnityEngine.AI。

五、 协程与委托

1. Coroutine 协同程序

因为 Unity3D 其实是单线程运行的，为了使 Unity3D 中的游戏脚本具有类似于多线程的功能，Unity3D 的研发团队便为其引入了 Coroutine（也称为协同程序或协程）功能，协同程序功能的实现主要是使用了 C#语言中的迭代器。协同程序是在 Unity3D 的 Update 函数之后执行的，协同程序的主要作用是延缓 Update 函数的执行。协同程序并不是进程或者线程，它的执行过程更加类似于子进程，即是一种不带返回值的函数调用。

当一个函数方法被调用是，正常情况下的执行逻辑是这个被调用的函数方法一定会在返回之前全部执行完毕，即这个函数方法的内部逻辑必须是在一帧的过程中执行完毕，所以调用一个函数方法是无法实现逻辑在一段时间内可局部暂停从而逐步执行的效果。例如当玩家或者敌人因受到攻击生命值减少时，可能还会在接下来的较短时间里受到多次攻击，这个时候需要保存前面状态的逻辑然后附加新的状态，这样才不至于有过多重复的函数调用，更不至于每切换一次状态都得重新加载一次代码。

update 函数返回后将运行正常协程更新。协程是一个可暂停执行 (yield) 直到给定的 YieldInstruction 达到完成状态的函数。

1）协程的不同用法

yield return null;一帧之后继续。

yield return new WaitForSeconds;等待指定的时间后恢复，实际暂停时间等于给定时间除以 Time.timeScale。

yield return StartCoroutine();组合协程，这个协程的恢复条件是，指定的协程运行已完全终止。

2. Delegate 委托

委托是一种类（Class），是一种引用类型的数据类型。委托可以存储一个或者多个方法的引用（只要类型兼容）。当开发者实例化委托后，即创建委托的实例，这个委托类型的实例可以和任何其他方法相关联起来，即可以存储一个或者多个方法的引用。开发者可以直接通过委托的实例间接调用（Invoke 或 Call）来调用方法。

哼哼哈哈次元游戏人生

委托是一种类，它可以指向一个自定义参数的参数列表和自定义返回值类型的方法。如果在项目中，有别的方法，而这个方法的签名和返回值，与我们声明的委托类型的签名和返回值类型完全相同、匹配或者说类型兼容。那么，委托类型的实例便可以代替这些方法，间接的调用它们各自的这些方法。

1） 自定义委托的声明

访问修饰符+delegate 关键字+[目标方法]+委托类型名字+圆括号内的目标方法

目标方法就是我们委托可以指向的[这个方法]就叫做目标方法，或者说我们创建的委托可以存储的这个方法的引用，存储的这个方法就叫做目标方法。

示例： public delegate void MyDelegate();

这个委托就可以指向一个没有返回值，参数列表为空的方法。

2） 单播委托

示例： MyDelegate = new MyDelegate(ChangeColor);

示例： MyDelegate = ChangeColor;

我们把一个委托封装了一个方法的形式，叫做单播委托。

3） 多播委托

一个委托内封装了多个方法的形式，叫做多播委托。

示例：

MyDelegate += new MyDelegate(Teleport);

MyDelegate += ChangeColor;

4） Action 委托和 Func 委托

Action 委托和 Func 委托都是系统中内置的一个委托类型，并且是一个泛型委托，它们都可以指向最多具有 16 个参数个数的方法。

Func 委托必定会指向一个具有返回值的目标方法，并且 Func 委托的参数类型排列必须在返回值类型之前。

Action 委托必定指向一个没有返回值，即返回值为空的目标方法。

委托可以作为参数传递到方法中。

由于委托是一种方法级别的紧耦合，使用委托必须慎之又慎。委托使用不当会导致代码的可读性下降、内存泄漏和程序性能下降。如果在项目中同时将多线程、异步调用和委托回调捆绑在一起，程序代码的后期维护和阅读将会变得非常困难。

六、 Event 事件

因为 Unit3D 构建消息系统是围绕着 SendMessage 和 BroadCastMessage 而构建的，但是消息系统太过于依赖反射(reflection)机制来查找消息对应的被调用函数，当条件反射机制被频繁的使用会极大的影响系统的性能并且增加代码的重构与维护成本。因此目前 C#使用事件来实现消息系统，而事件机制是以委托作为基础的。

1） 事件的功能

当某件有趣的事情发生时，一个类或对象中的事件就会通知别的类、别的对象（向别的类、对象提供通知），别的类和别的对象，在接收到这个通知之后就会纷纷作出它们各自的响应。即，事件是一种具有通知功能的成员，事件的核心功能就是通知。

2） 事件命名的一些约定

.Net 约定，在一般情况下，如果这个方法它是触发事件的方法或者事件本身，在命名时需要在前面加上 On 来作为前缀。

如果一个类，它是用来传递事件消息（EventArgs）的，这个类的名字就应该叫做[xx

代理聆理次元游戏人生

事件+EventArgs]。EventArgs 就是 EventArgument 的简写，并且我们会习惯性的把这个类，派生自、继承自 EventArgs 这个类，而 EventArgs 类是微软为开发者准备好的一个基类。

为了事件而声明的委托类型，.Net 又规定了，我们最好在委托类型的后面加上 EventHandler 来作为后缀。

3） 事件不是特殊形式的委托类型字段

委托和事件的关系类似于属性和字段的关系，事件只是委托类型字段的包装器和限制器，事件的存在限制了外界对这个委托类型字段的访问，使得外界只能通过“+=”或“-=”操作符来添加与移除事件处理器。

4） 事件相关名词

事件在发生的时候，必定会产生一些和事件本身相关的数据。我们把与事件本身相关的数据、也就是我们所说的消息叫做事件参数：EventArgs。

响应事件或处理事件是一种通过通知和事件参数而采取行动的行为。在程序世界当中，如果一个事件发生了，别的类要作出响应，那么别的类（事件响应者）必定会去调用它的一个方法（事件处理器），而我们把这个方法就叫做事件处理器：EventHandler。

5） 事件模型

事件模型在构建和运作时具有五个步骤：

a. 这个类要有一个事件成员（事件成员会通知其他类、对象作出响应）。

b. 一群别的类关心着、订阅着这个事件。

c. 这个事件发生了。

d. 订阅了这个事件的“人们”就会被依次的通知到，通知的顺序也就是当初订阅这个事件的顺序。

e. 知晓信息的“人”，便会根据所得到的事件信息（事件参数）对这个事件进行响应。

事件模型的五个组成部分：

① 事件拥有者

事件的源头，事件的拥有者一定是一个类（或者说是对象），谁拥有这个事件，谁就是事件的拥有者。

② 事件

事件是成员类型，其核心功能是通知，通知其他类、对象作出响应。事件不会主动发生，一定是由事件拥有者的内部逻辑触发。

③ 事件响应者

当事件发生后，有哪些对象、类被通知到了，被通知到的对象、类就是事件的响应者。

④ 事件处理器

事件响应者的其中一个方法成员，一个事件响应者可以有多个事件处理器。

⑤ 事件订阅

事件发生后，通知到的一定是订阅（通过“+=”操作符订阅）了事件的对象或类。事件响应者要订阅事件，其事件处理器的返回值类型和参数列表一定要和事件的委托类型一致。

6） 事件是基于委托

① 类型兼容

事件需要使用委托类型来做一个约束，这个约束既规定了这个事件能发送什么样的消息给事件的响应者，也规定了事件的响应者能收到什么样的消息。这决定了事件响应者的

哔哩哔哩次元游戏人生

事件处理器必须和这个约束相匹配，才能订阅这个事件。

② 存储方法的引用

当事件的响应者，向事件的拥有者提供了一个与之匹配的事件处理器之后，我们需要一个地方把这个事件处理器及之后的其他事件处理器保存、存储、记录下来。能够记录、存储、保存方法的这个任务，只有委托类型实例能够办到。

七、 TextMesh Pro

Unity3D 的最终文本解决方案就是 TextMesh Pro。TextMesh Pro 的功能非常强大并且易于在项目中使用，因为 TextMesh Pro 由高级文本渲染技术和一套自定义的文本着色器构建而成；TextMesh Pro 不仅可以大幅度的改进渲染出的文本视觉质量，而且还同时为使用者创建新的文本样式和纹理提供了空前绝后的灵活度。

TextMesh Pro 相较与用像素描述的 Unity UI Text 能够更好地控制文本文字的整体格式和布局，TextMesh Pro 提供的功能不仅让使用者感到眼花缭乱更会感到耳目一新。在保证和传统 Unity UI Text 的功能的基础上，新增了对超过 30 种富文本标签的支持、多种艺术字体和 Sprite（精灵图）支持等功能。

虽然 TextMesh Pro 创建的字体是通过公式描述的几何体，但是因为却和 Unity 的文本组件一样，TextMesh Pro 每个字符都是使用的两个三角形面片构成的矩形，因此使用 TextMesh Pro 带来的视觉质量改进和灵活性与使用传统的 Unity UI Text 相比不会带来任何性能成本的损失。

如果需要使用代码访问 TextMesh Pro 对象，我们需要使用一个名叫 TMPro 的命名空间。

using TMPro;

八、 富文本

要想使 Unity3D UI 元素的文本文字和它的文本网格能够包含更多种类的字体样式和大小，就需要使用富文本（Rich Text），而 Unity3D 的 Rich Text Format（富文本格式）支持在 UI 系统和旧版的 GUI 系统都有效。Unity3D 的 Text，GUIStyle，GUIText 和 TextMesh 类都可以设置富文本，只要使用富文本需要对应的标签。在使用 Debug.log 功能时也可以使用富文本的标签，富文本可以提高错误代码的视觉效果，让开发者更容易查找错误位置。

1. 标记格式

标签系统受 HTML 启发，但是并不旨在于标准 HTML 严格兼容，其基本思想是可以将一段文本包含在一对匹配的富文本标签内。

```
We are <b>not</b> amused.
```

图1. 图 7.1 标签示例

如图 3.15 所示，标签只是尖括号（“<>”）字符和尖括号字符里的一段文本。标签内的名称表示其名称，图中的标签是字母 b，即英语单词 bolder 的缩写。结束标签与开始标签非常相似，只是的名称前面会带有斜杠（“/”）字符，因此</b>就是 b 标签的结束标签。每个开始标签必须有一个对应的结束标签，如果未关闭开始标签，则标签对应的富文本效果将无法生效，将显示为常规文本。

这些标签不会被直接展示给用户，而是被解释为特定的文本样式。

We are **not** amused.

图2. 图 7.2 实际显示效果

2. 嵌套元素

被标记的文本部分(包括它周围的标记)被称为元素（Element）。通过将一个元素“嵌套”到另一个元素中，可以对一个文本部分应用多种样式。当嵌套有多个元素时，结束标记的顺序与开始标记的顺序相反。

```
We are <b><i>definitely not</i></b> amused
```

图3. 图 7.3 嵌套元素示例

We are ***definitely not*** amused

图4. 图 7.4 嵌套元素实际显示效果

3. 标签参数

一些标签对文本具有简单的“全有或全无”效果，但其他标签可能允许一些其他变化。例如，应用颜色标签时，颜色标签需要知道应用何种颜色。像这样的信息是通过使用参数添加到标签中的：

```
We are <color=green>green</color> with envy
```

图5. 图 7.5 标签参数示例

We are green with envy

图6. 图 7.6 标签参数实际应用效果

结束标签不包含参数值，且参数标签不能包含空格。在默认情况下，富文本在 Editor GUI 系统中是被禁用了的，要启用 Editor GUI 系统中被禁用了的富文本，就可以使用自定义的 GUIStyle 来显式启用。该 richText 属性应设置为 true，并将样式传递给相关的 GUI 函数：

```
GUIStyle style = new GUIStyle ();
style.richText = true;
GUILayout.Label("<size=30>Some <color=yellow>RICH</color> text</size>",style);
```

九、 2D PSD 导入器

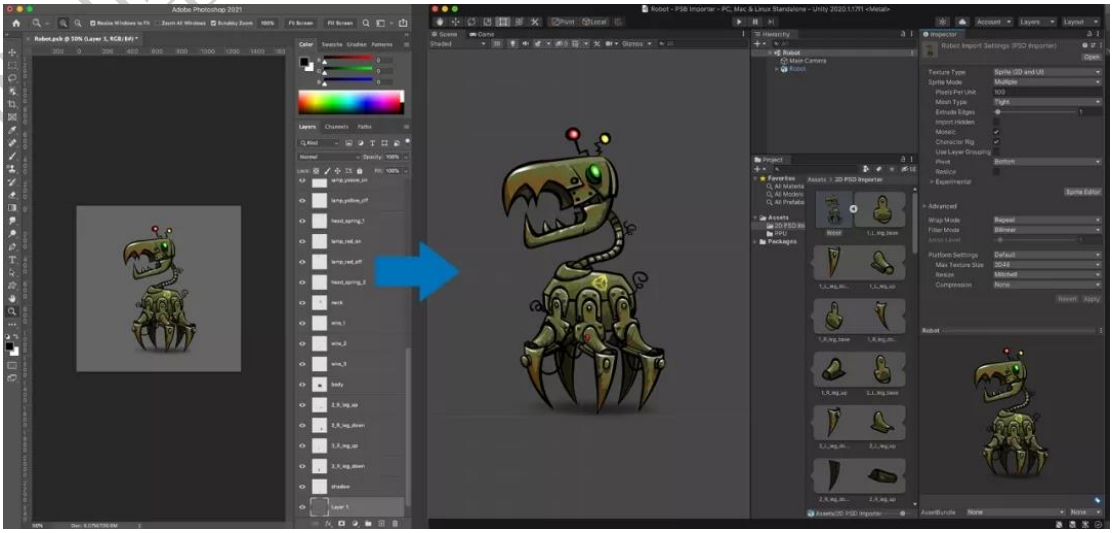


图7. 图 8.1 PSD 导入示例

要想将带图层的 Adobe Photoshop PSB 源文件导入到 Unity 中，就可以使用 Unity3D 中的 2D PSD 导入器。Unity3D 中的 2D PSD 导入器可让开发者使用导入器拥有的马赛克（Mosaic）等功能，根据导入到 Unity 的 Adobe Photoshop PSB 源文件的图层和绘画出的角色骨架去自动化的生成一系列的精灵表单（Sprite Sheet）。在精灵表单（Sprite Sheet）完全生成后，Unity3D 就会将角色的精灵图（Sprite）进行重新组合然后再保存至源文件中。Unity3D 中的 2D PSD 导入器的作用在于可以帮助引擎更好地兼容 2D 动画系统（Animation）与多图层角色的设计流程，帮助开发者更加便捷高效的完成动画的制作，极大程度的减少了 2D 项目的工作周期和工作难度。

1. 减少像素重叠

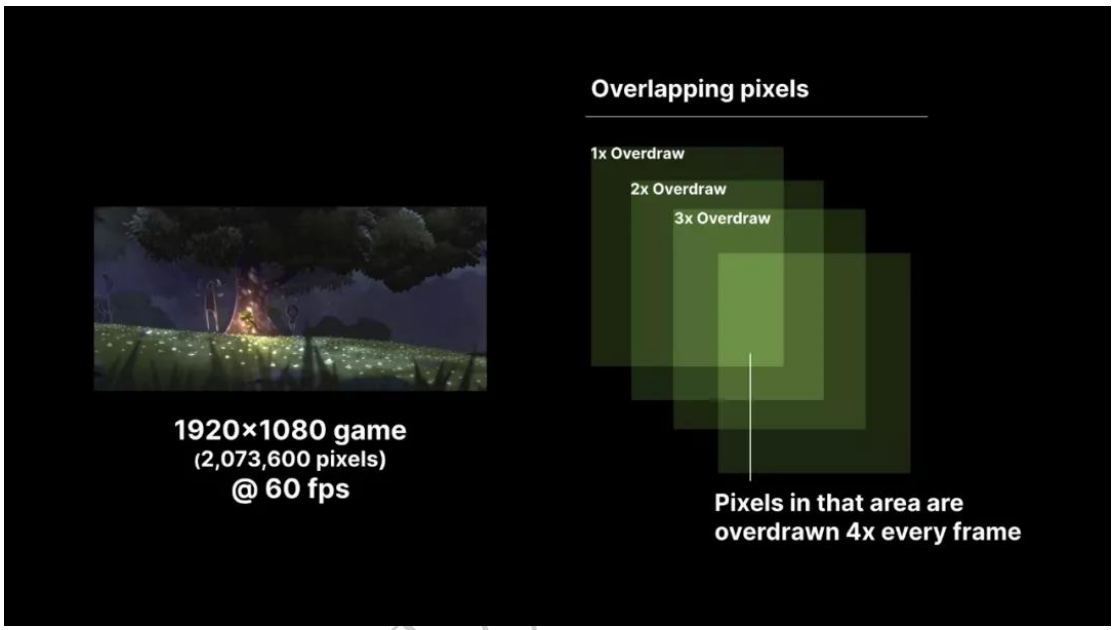


图8. 重叠像素优化示例

重叠像素优化功能旨在减少最终呈现到屏幕上的色彩所需要的绘制次数。当需要半不透明度的像素出现在屏幕上时，这个像素的色彩最终必须考虑到像素的重叠。当多个半透明的像素交汇于同一位置时，系统就可能多次地绘制这一重叠像素，一旦多个重叠像素都被过多的绘制，那么 GPU 的性能就会降低，在低端设备或高帧率模式下最终显示到屏幕上的画面就会显得不尽如人意。通常情况下，系统会将游戏场景中的所有精灵图（Sprite）都视为透明像素，在这种情况下就需要考虑一些减少过度绘制的方法来提升 GPU 的性能。

在精灵遮罩（Sprite Mask）所划定的范围内，Unity3D 是允许像素被多次绘制的。当精灵遮罩被设置为矩形时，整个精灵遮罩的矩形区域都将允许像素被过渡绘制。但是，如果在 Unity3D 中设置了紧缩（Tight）的网格类型（Mesh Type），那么程序生成的精灵遮罩就会跟随精灵图的大致轮廓，向 PNG 格式的图片一样忽略掉空白的部分，只显示出具有实际像素的部分，从而来减少过度绘制像素的区域。

在 Unity3D 中还有一种减少像素被过渡绘制的方法，就是将重叠的静态精灵图进行合并，使这些不参与视差、静止不动的精灵图在项目设计完毕以后合并为一个整体。

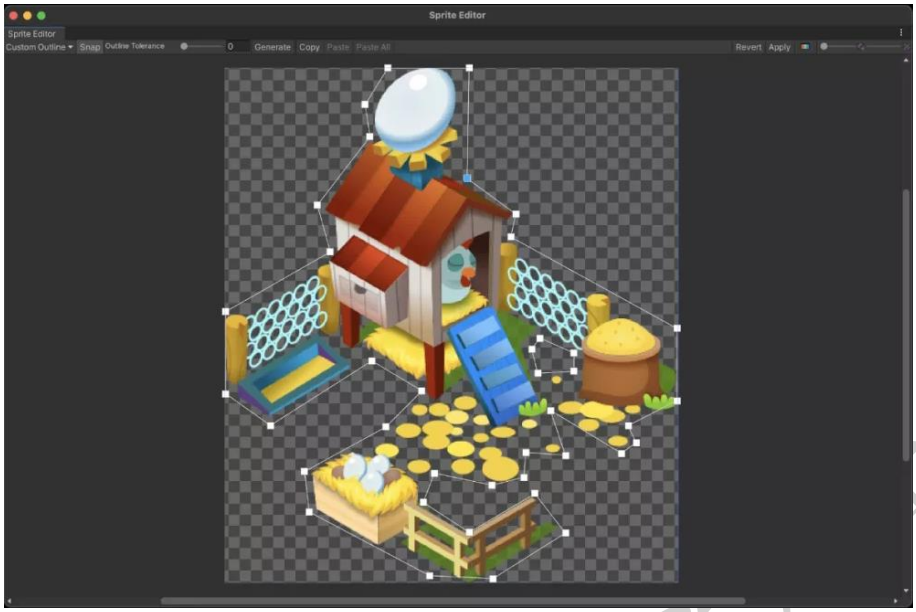


图9. 图 8.3 合并重叠的静态精灵.

2. 组合碰撞体

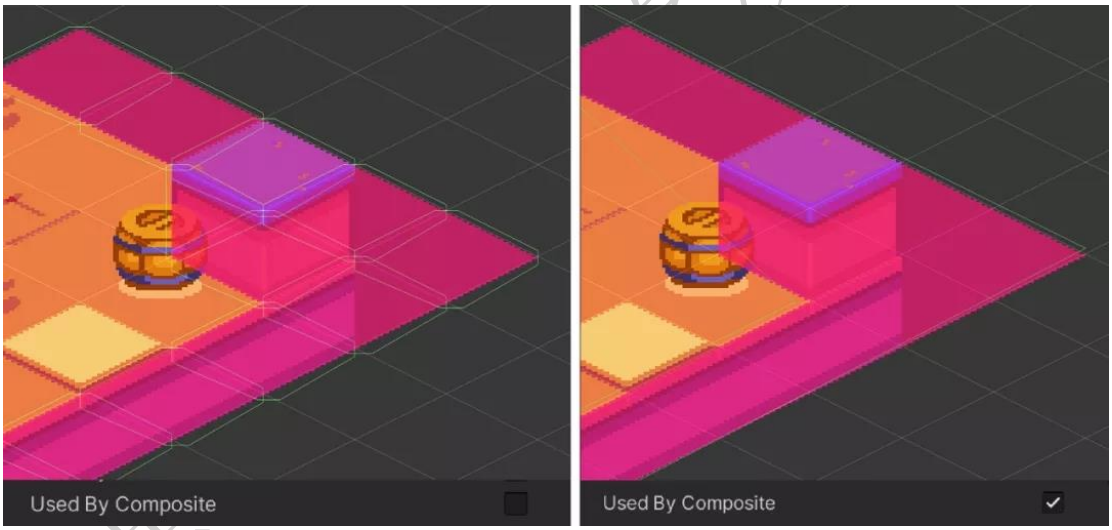


图10. 组合碰撞体的关闭和开启

当游戏场景中同时存在有多个附带静态 2D 碰撞体（被设置为静态的 2D 碰撞体）的精灵图时，开发者可以使用 Composite 2D Collider 组件把多个碰撞体组合成一整个对象，减少计算机的实际运算负担，进一步简化场景中的碰撞运算。因为多个独立的碰撞体被合并为了一个整体，因此这个合并后的对象表面的碰撞将会变得更加的流畅。如图 3.24 所示，把紫色(purple)的图块视为 2D 的碰撞体，图中的 2D 平面是通过 Unity3D 中的瓦片地图(Tilemaps)构建的，而 Composite Collider（合成碰撞体）是可以与 Tilemaps 一起使用，只需要开发者在 Inspector 视图中的添加组件选项中选择 Composite Collider 并勾选 Composite 的 toggle 选项后便可以启用 Composite Collider 功能，将多个独立的碰撞体合并为一个整体。

在 Unity3D 中的 Composite 2D Collider 组件是可以同时运用于父级的 GameObjects。要将其运用于父级游戏对象，在 Inspector 视图中需要勾选 2D 碰撞体的 Used by Composite 选项（表面所使用的碰撞体是一个复合碰撞体）来启用子碰撞体。

要优化 2D 碰撞体还有一种方法，就是在 Unity 中使用 2D 的多边形碰撞体(Polygon

Colider 2D)，手动的去绘制一个能够覆盖整个游戏对象(GameObject)的简单多边形(Simple Polygon)。

3. 形状精灵烘焙

Sprite Shape 即形状精灵，是 Unity 中的 2D 工具，它能够帮助开发者更加灵活地设计 2D 世界的形状和关卡。Sprite Shape API 默认情况下允许程序在处于运行状态时修改 Sprite Shape 样条线的节点（在运行状态时修改 Sprite Shape 样条线的节点，会对实际的游戏性能产生一定的影响，影响的大小取决于修改的数量）。为了提高图像性能，在 Unity3D 项目中可以将无需在程序运行时动态修改的精灵图，预先烘焙或者缓存 Sprite Shape 样条线的几何形状。选择形状精灵控制器（Sprite Shape Controller），点击启用样条线编辑（Edit Spline）、缓存几何体（Cache Geometry）即可完成网格的烘焙。当开发者想要在项目运行状态下修改 Sprite Shape 的样条线，可以在使用 Unity 2020.2.3f1c1 及以上版本时，在 Unity3D 中的 Window-> Package Manager（包管理器）中安装 1.3 及以上版本的 Burst 软件包来极大提升图像性能。

图11.　　使用形状精灵快速制作场景美术

由于 Sprite Shape Profiles 绘制出的属于矢量图，因此具备有市面上常见矢量绘图软件如 Photoshop、Illustrator 相似的作画方法，在 Unity 中开发者可以将其用于绘制场景中的游戏性和装饰性元素。Sprite Shape 的功能不仅能够兼容多种多样的艺术风格，还能够在极少操作步骤的情况下快速填充大片区域，在实际的开发过程中可以减少美术人员对纹理贴图的绘制，加快开发速度。

4. 像素美化与锐化

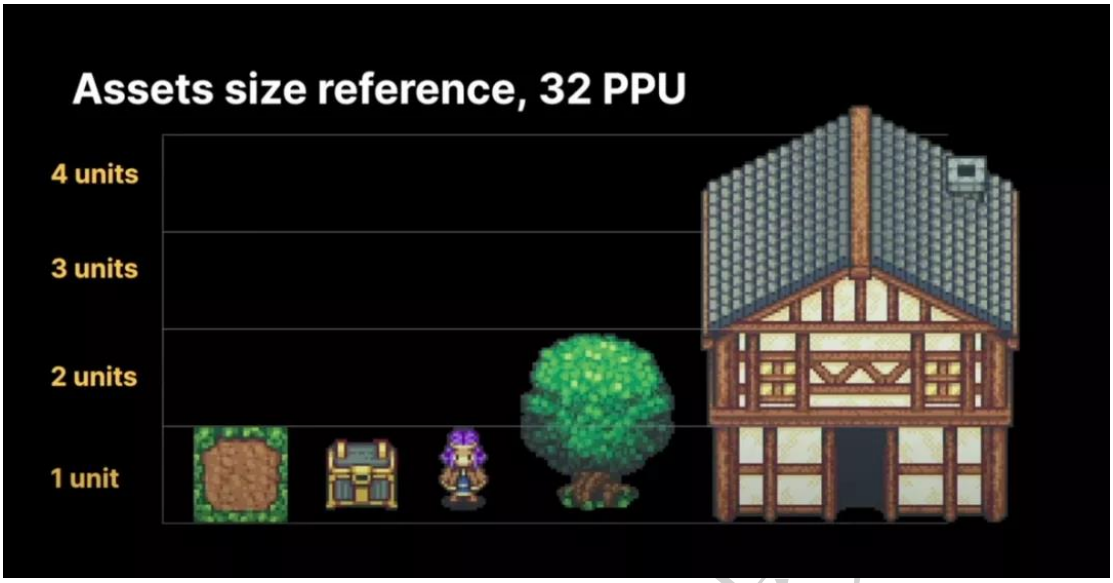


图12. 资源尺寸参考——32 像素每单位（PPU）

当图像显示在屏幕上时，是按照像素进行显示的，而 2D 游戏的开发关键点之一就在于游戏的画面资源是需要具有一致的像素密度，即多少个像素点对应一个单位。如图 3.26 所示如果设置项目中的像素密度为 32 PPU，那么在项目中每一个面积单位就包含有 32 个像素点。

在 Unity3D 中以镜头的中心为坐标原点、以垂直方向为单位在默认的摄像机 Camera 上描述图像的尺寸。就如图 3.27 所示，在 Camera 的 Inspector 视图中设置摄像机的大小(Size)，在 Camera 镜头的垂直方向上可以显示出 10 个图块：

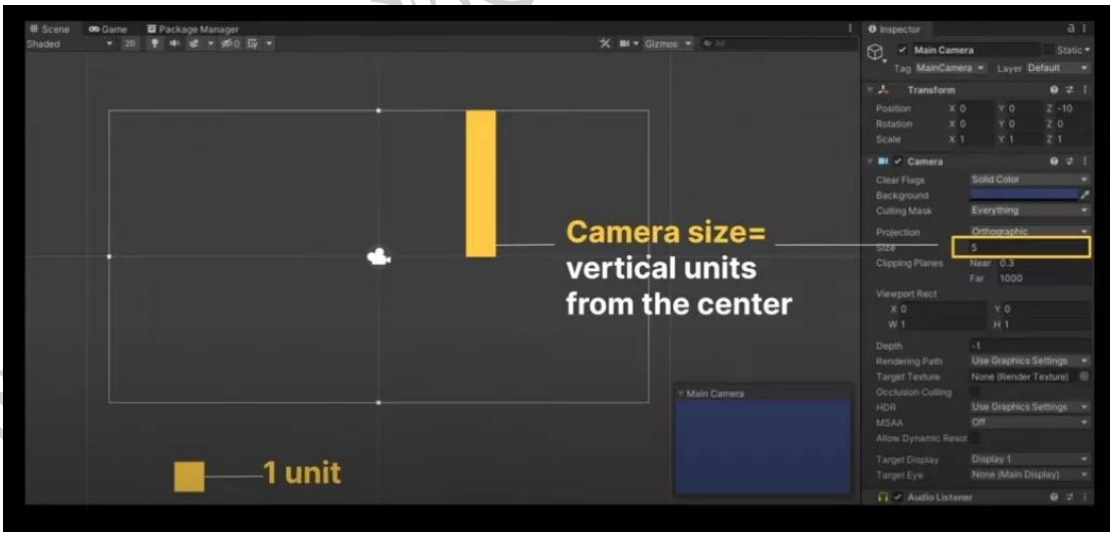


图13. 图 8.7 图像尺寸描述示例

如果场景视图(Scene)显示出的精灵图(Sprite)与参考图像(reference picture)相比显得过于渺小或者过于庞大，那么就很有可能是因为导入的图像资源的分辨率设置不统一。

当导入的图像资源在未做任何修改的情况下，与其他的图像资源显得过于违和，可能是因为在默认情况下 Unity3D 的精灵图的 PPU(Pixels Per Unit)是设置为 100 的，只需要将设置进行一些简单的调整，比如将其改为 32 PPU，那么导入的精灵图就可能会正常显示。

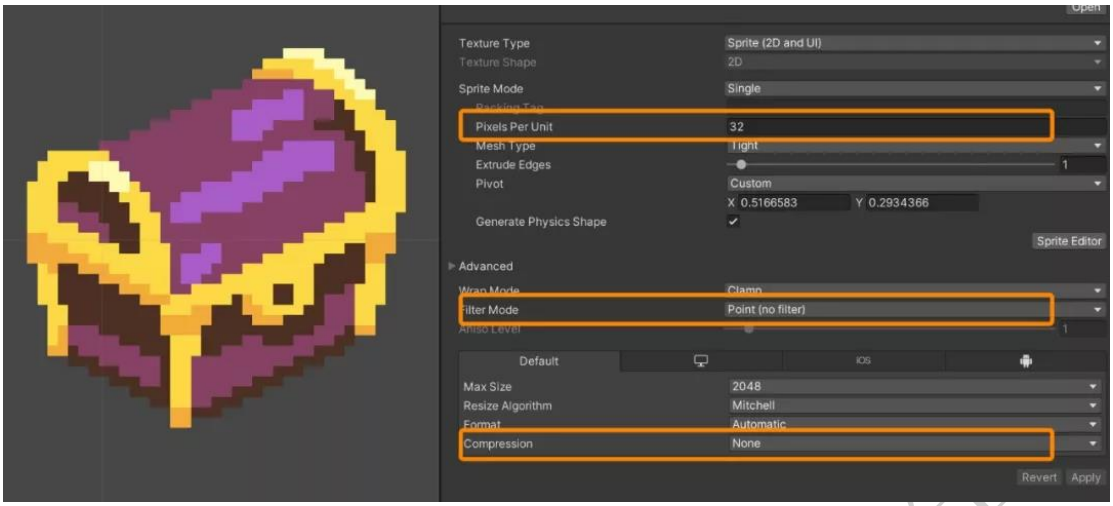


图14.　　图 8.8 精灵图设置

Unity3D 引擎为了能够兼容更多的美术风格，在引擎中内置了能够自定义的过滤模式(Filter Mode)。如果开发者想要在游戏中使用复古风的像素图像，那么就可以将过滤模式变更为点(Point)，以让图像效果在划定的屏幕空间内变得更加的锐利。如果不对过滤模式加以修改，那么引擎默认会开启双线性过滤(Bilinear Filtering)来平整突出的边角。

十、 粒子系统 Particle System

在影视作品与游戏作品中会经常使用到粒子系统，通常火焰、烟尘、雨滴或者发光轨迹之类的抽象视觉效果无一例外都是使用的粒子系统。粒子系统所解决的现象是用其它传统的渲染技术难以实现的真实感的游戏图形，即粒子系统表示三维计算机图形学中模拟一些特定的现象的技术。

粒子系统是最常用的过程动画技术之一，它被认为是模拟不规则模糊景物最成功的一种图形生成算法，一般用于自然景物的动态模拟。对于任何物体，不论是液态、气态、固态，都可以认为是有简单的粒子构成。这符合物理学的定律：物体都是由最基本的微粒构成的。其基本思想是：采用大量的、具有一定生命和属性的微小粒子图元作为基本的元素来表示不规则的、模糊的物体。

通常情况下，粒子系统在三维空间中的位置与运动是由发射器控制的。发射器主要是由一组粒子行为参数以及在三维空间中的位置所表示。粒子行为参数可以包括粒子生成速度（即单位时间粒子生成的数目）、粒子初始速度向量（例如什么时候向什么方向运动）、粒子寿命（经过多长时间粒子湮灭）、粒子颜色、在粒子生命周期中的变化以及其它参数等等。使用大概值而不是绝对值的模糊参数占据全部或者绝大部分是很正常的，一些参数定义了中心值以及允许的变化。

粒子系统并不是一个简简单单的静态系统而已，在粒子系统中，随着时间的推移，系统中已有粒子不断运动，其属性也随之不断地发生改变，并且还有新的粒子不断地加入，旧的粒子消失。在这个过程中，每个粒子都被赋予了生命周期，都将经历产生、活动和死亡的全过程。

典型的粒子系统更新循环可以划分为两个不同的阶段：参数模拟阶段以及渲染阶段。

1. 模拟阶段

在参数模拟阶段，系统根据开发者设定的生成速度以及更新间隔计算应产生的新粒子的数目，新的粒子在三维空间中的生成位置由粒子系统中发射器所处的位置，以及开发者所设定的生成区域共同确定。而新诞生的粒子的属性参数，例如速度、颜色、生命周期等是由开发者设置的发射器的参数决定的。

然后检查粒子系统中的每个粒子的生命周期，一旦粒子超出了它的生命周期就将这些

粒子从模拟过程中移除，否则就根据物理模拟去继续更改粒子的位置与特性。物理模拟的复杂程度是不一的，它可以像计算位移一样简单，也可以像计算核裂变反应一样复杂漫长，具体的复杂程度是由玩家在粒子系统的设置来定。

粒子系统的规则不是固定的，不同的粒子系统有着特定的规则去作用于其中的粒子，即影响粒子生命周期的差值过程。例如，许多的粒子系统会对粒子的阿尔法值(Alpha)即不透明度进行插值运算，使粒子的阿尔法值在生命周期内淡出或者淡入。

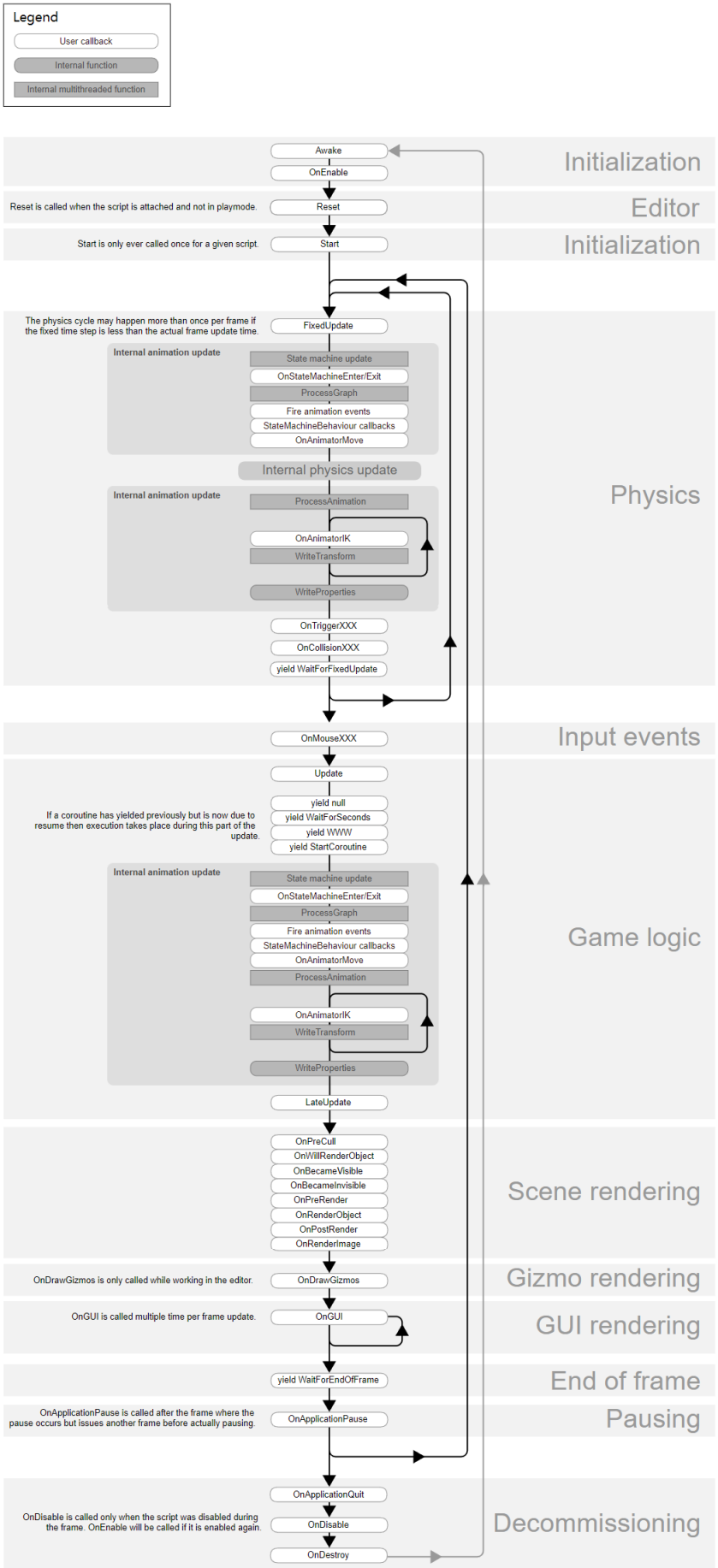
2. 渲染阶段

在更新完成之后，通常每个粒子用经过纹理映射的四边形 sprite 进行渲染，也就是说四边形总是面向观察者。但是，这个过程不是必须的，在一些低分辨率或者处理能力有限的场合粒子可能仅仅渲染成一个像素，在离线渲染中甚至渲染成一个元球，从粒子元球计算出的等值面可以得到相当好的液体表面。另外，也可以用三维网格渲染粒子。

十一、 脚本生命周期

Awake->OnEnable->Start->Update->FixedUpdate->LateUpdate->OnGUI->Reset->OnDisable->OnDestroy

哔哩哔哩次元游戏人生


Legend
User callback
Internal function
Internal multithreaded function
Awake
OnEnable
Initialization
Reset is called when the script is attached and not in playmode.
Reset
Editor
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
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
Internal physics update
Internal animation update
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
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
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
LateUpdate
Game logic
OnPreCull
OnWillRenderObject
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
OnApplicationQuit
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
OnDestroy
Decommissioning

十二、 Animation

1. 关于模型中的动画的位移变换问题

角色动画的 APPLY ROOT MOTION 和 BAKE INTO POSE 问题。

导入到 unity3d 的一些带动画的模型，有的动画会有位移变换，有的则没有。而是否将模型在动画中的位移变换，应用到模型在场景中的位移，有两个地方需要设置：Apply Root Motion 和 Bake into Pose，因此 unity 中将动画的变换分成两种：Root Transform 和 Body Transform。

如果设置 Root Transform，将会影响模型的实际位置，前提是需勾上 Apply Root Motion，如果不勾上变换将不应用。模型的位置不会改变。

如果设置 Body Transform，在场景中模型位置会发生变换， 但实际的位置将不会发生变化。Apply Root Motion 起两个作用：

① 决定是否应用 Root Transform，如果应用，在播放动画的同时，模型的位置会同时发生变化，如果不勾选，将不应用 Root Transform。所有的变化将不起作用。

② 在动画结束后，将 Body Transform 中的变换在动画结束后应用到模型。

下面将分为四种情况(×代表不勾选，√代表勾选)：

Apply Root Motion	Bake into Pose
×	√
√	√
√	×
×	×

第一种

不勾选“apply Root motion”，勾选“bake into pose”，属于 Body Transform，在场景中，如果模型动画带位移，会自动播放位移，但根结点不会位移，所以动画结束后，变化不会应用到模型，模型仍在起始位置。

第二种

属于 Body Transform，动画过程中模型位置不会变化，动画结束后，变换会应用到模型，模型处于动画结束时的位置。

第三种

属于 Root Transform，变化应用到模型，动画过程中模型跟着动画不断地变换，结束后，模型处于动画结束时的位置。

第四种

属于 Root Transform，但变换不被应用，所以模型一直在本地不动。

1) 总结

① 勾选上 Apply Root Motion，模型的实际位置会随动画中模型的位移而发生变化；不勾选就不发生变换。

② 勾选上 Bake into Pose，播放有位移的动画时，模型实际位置不会改变，直到动画结束，其位置才改变；不勾选，模型随动画中位移而位移。

十三、 自动布局组件

1. Content Size Fitter

Property		功能
Horizontal Fit	Unconstrained	不要根据布局元素变动宽度。
	Min Size	根据布局元素的最小宽度

		变动宽度。
	Preferred[1] Size	根据布局元素的首选宽度变动宽度。
Vertical Fit	Unconstrained	不要根据布局元素变动高度。
	Min Size	根据布局元素的最小高度变动高度。
	Preferred Size	根据布局元素的首选高度变动高度。

Content Size Fitter 用作布局控制器，用于控制其自身布局元素的大小。大小由游戏对象上的布局元素组件提供的最小或首选大小确定，此类布局元素可以是图像或文本组件、布局组或布局元素组件。

值得注意的是，当 Rect Transform 调整大小时（无论是通过 Content Size Fitter 还是其他方式），调整大小都是围绕 Pivot[2]进行的。这意味着可以使用 Pivot 来控制调整大小的方向。

这是一个例子，当 Pivot 位于中心时，Content Size Fitter 将基于 Rect Transform 在所有方向上均匀地(equally[3])向外扩张；当 Pivot 位于左上角时，Content Size Fitter 将基于 Rect Transform 向右下角方向扩张。

十四、 后处理

实时渲染中的后处理(Post-processing)发生在每一帧被渲染完成之后。我们可以通过后处理为画面添加各种效果，比如调整画面明暗对比度、整体颜色、白平衡、抗锯齿、景深、运动模糊等。HDRP 和 URP 中有关后处理的技术也被集成在 Volume 框架中，因此我们不必像导入默认渲染管线一样导入额外的后处理包。

1. 为场景添加后处理效果的步骤

① 步骤 1

在场景中创建一个空的 GameObject，并为它添加一个 Volume 组件。

[1] adj. 更合意的；优先（考虑）的；被偏爱的 v. 偏爱；宁可（prefer 的过去式和过去分词）
[2] n. 支点，枢轴；核心；转动，旋转；v. （使）在枢轴上旋转；（以脚为支点）转身；
[3] adv. （程度）相当地，同样地；平均地，相等地；（用于补充观点）同样（重要）地

哔哩哔哩沙无游戏人生

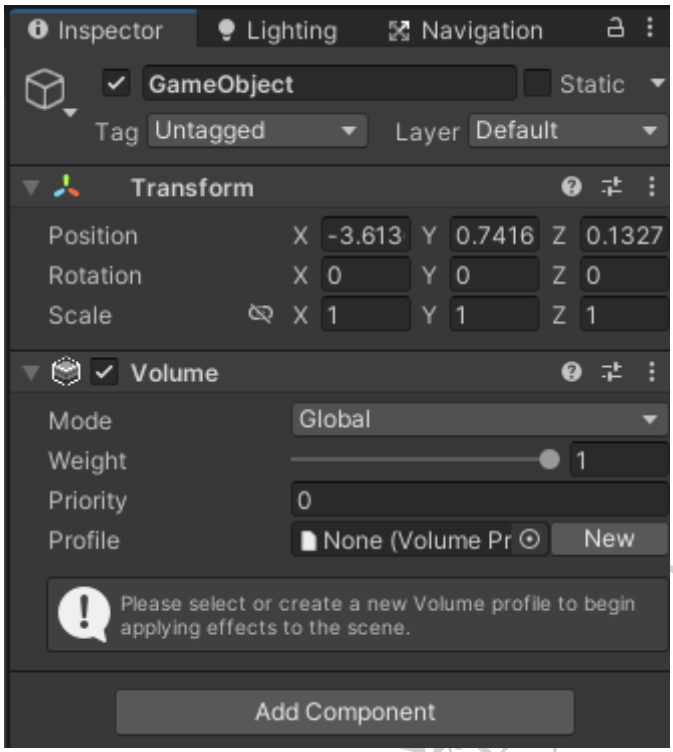


图15.　　图 13.1 为对象添加 Volume 组件

② 步骤 2

如果是新项目，那么 Volume 组件是没有 Volume Profile 的。我们需要创建一个 Volume Profile 并关联到 volumes 组件。

如果已有 Volume Profile，可以自行选择使用哪一个，或者点击 New 新建一个。

③ 步骤 3

为 Volume 组件赋予了 Volume Profile，下面就会出现一个 Volume Override，我们点击 Volume Override 就可以添加与 Post-processing 相关的 Override。

哔哩哔哩次元游民生

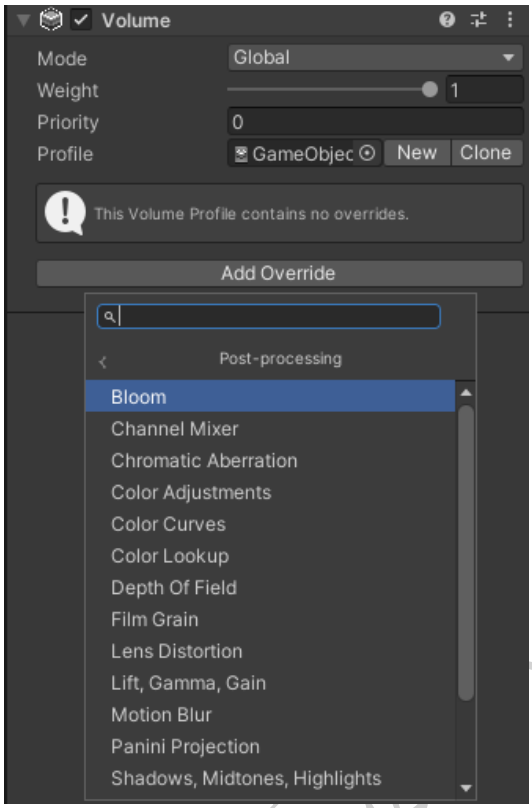


图16.　　图 13.2 使用 Override 添加后处理效果

④　步骤 4（HDRP 项目）

在 Project Settings 窗口的 HDRP Default Settings 中可能已经存在一些默认的后处理效果。为了不影响为场景中添加的后处理效果，我们可以把默认的后处理效果禁用。

哔哩哔哩优质人生

2. 后处理效果应用顺序和效果组合

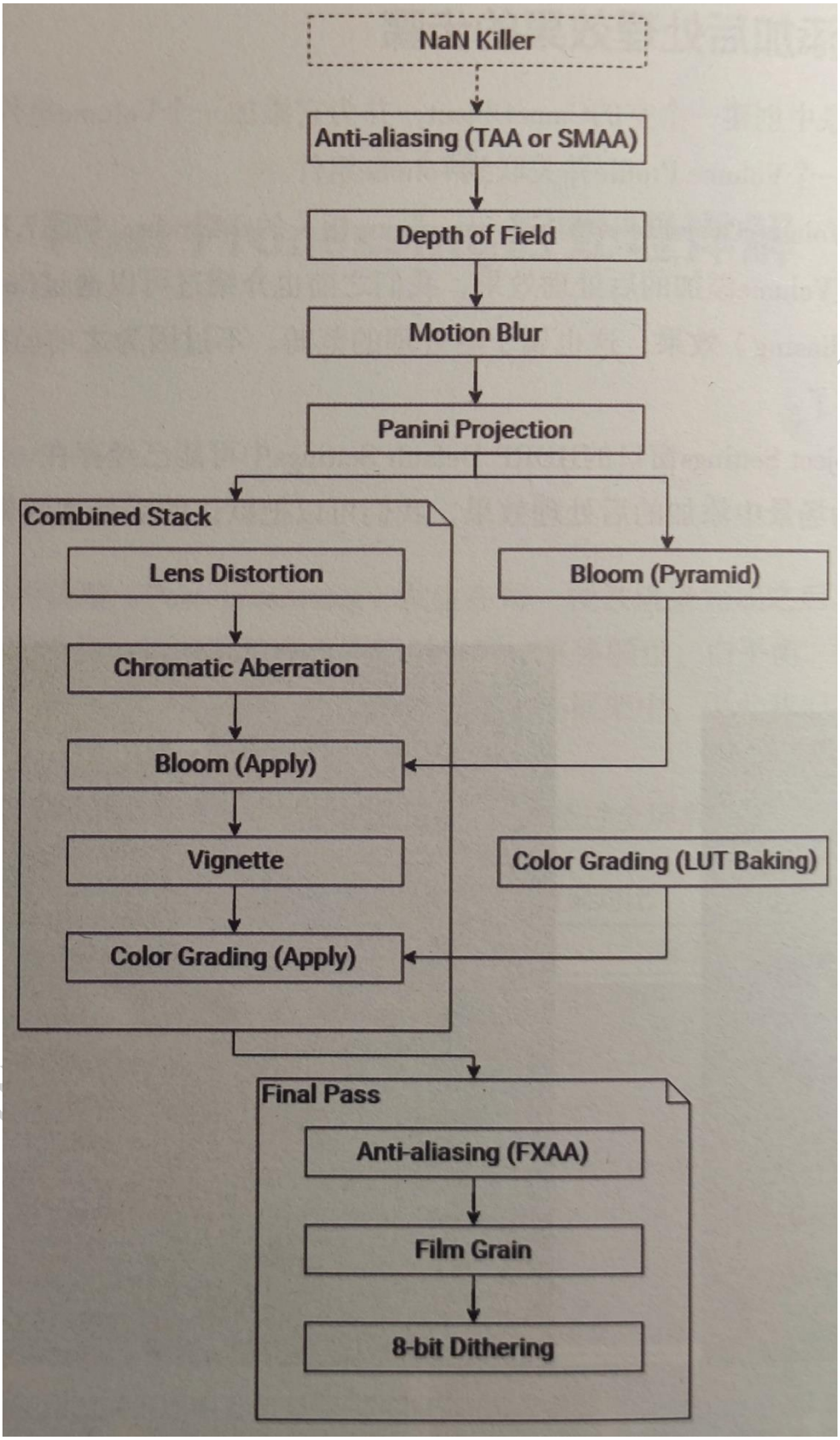


图17.　　图 13.3 HDRP 渲染管线应用后处理效果的顺序

HDRP 渲染管线会按照上图所示的顺序来应用后处理效果。HDRP 也会将多个后处理效果集中到同一个 Computer Shader 中减少渲染绘制批次。

3. 辉光-Bloom

Bloom 效果使图像中的明亮区域发光。Bloom 效果还可以使用 Lens Dirt 效果用全屏的污迹或灰尘层来衍射 Bloom 效果，本质上就是往整个画面上叠加一张纹理。

1） Bloom

临界值-Threshold	用于过滤亮度小于此数值的像素。也就是说，亮度小于此数值的像素不会参与辉光的计算。
强度-Intensity	用于调整辉光的强度
散射-Scatter	数值越大，明亮部分的辉光效果范围越大，默认值为 0.7。
着色-Tint	针对辉光效果进行着色
钳-Clamp	设置 Unity 用于计算 Bloom 的最大强度。如果场景中的像素比这更强烈，URP 会以其当前强度渲染它们，但会使用此强度值进行 Bloom 计算。默认值为 65472。
高质量过滤-High Quality Filtering	启用此选项以使用高质量采样。这会减少闪烁并提高整体平滑度，但会占用更多资源并且会影响性能。
跳过迭代-Skip Iterations	处理序列中的最后一次迭代可能对渲染帧的外观影响不大。在此设置中，您定义要跳过的最终迭代次数。增加此值可减少处理负载并提高性能，尤其是在移动设备上。默认值为 1。

2） Lens Dirt

贴图-Texture	指定纹理以将污垢（如污迹或灰尘）效果应用到镜头。
强度-Intensity	设置镜头污垢效果的强度。

3） 解决性能问题

① 禁用高质量过滤。Bloom 然后使用双线性过滤而不是双三次过滤。这会降低 Bloom 效果的整体平滑度，但会大大提高性能，尤其是在低端硬件和平台上。在某些极端情况下，您可能会在场景中看到块状图形伪影。

② 使用较低分辨率的镜头污垢纹理来减少内存压力并加快跨体积的混合。

4. 色调映射-Tonemapping

将画面中的 HDR（高动态）值重映射到 LDR（低动态）值，重映射以后的画面适合在普通显示器上显示。

1） Mode

None	不应用色调映射
Neutral	对色彩和饱和度影响最小。如果我们会在当前画面上应用更多的颜色分级（Color

哔哩哔哩次元游戏人生

	Grading），可以应用此模式。
ACES	比 Neutral 模式产生的对比度高，会影响色彩和饱和度。应用此模式后画面效果更接近于电影画面。

5. 白平衡-White Balance

用于解决画面中偏色问题（偏色问题会导致画面看上去不真实）。另外，也可以用白平衡将画面调整为偏暖或者偏冷，偏绿或者偏洋红。

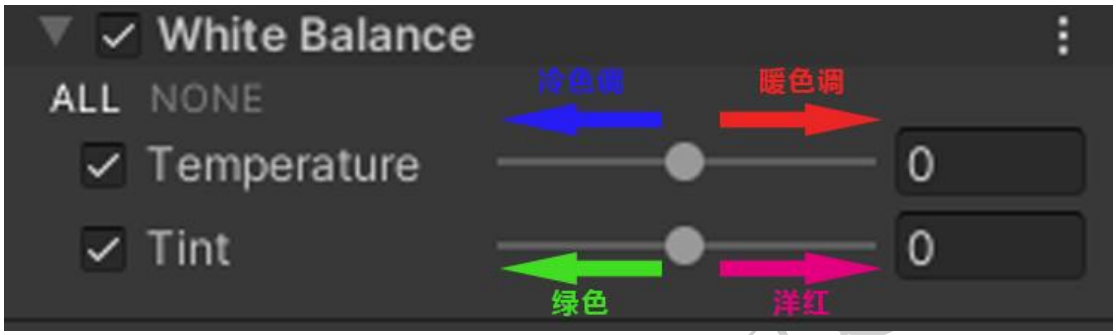


图18.　　白平衡

6. 胶片颗粒-Film Grain

模拟使用物理胶片拍摄时画面中出现的噪点。这些噪点通常由物理胶片上的小粒子所导致。

Type	可以选择预置的 Thin、Medium、Largel 噪点，也可以自己指定（Custom）一张纹理作为噪点的来源。
Intensity	用于控制噪点的强度。
Response	用于控制生成噪点的曲线。数值越大，在高亮区域的噪点越少。

7. 景深-Depth of Field

现实世界的相机镜头只能在固定距离内对某个物体聚焦，更近或者更远的物体会呈现虚化的效果。景深就是用来模拟真实相机镜头的特性的。

通用渲染管线 (URP) 有两种景深模式：

Gaussian：此模式类似相机的效果，但并未完全模仿它们。它具有有限的模糊半径，仅进行远场模糊。这种模式速度最快，是低端平台的最佳模式。

Bokeh：一种速度较慢但质量更高的模式，可以模仿真实相机的效果。它可以进行近场和远场模糊，并在高亮度区域（也称为热点）上生成散景。

1） 高斯景深-Gaussian Depth of Field

Property	Description
Start	设置远景开始模糊时与相机的距离。
End	设置远场模糊达到其最大模糊半径时与相机的距离。
最大半径-Max Radius	设置远景模糊可以达到的最大半径。默认值为 1。 注意：大于 1 的值可能会导致场景中出现

	欠采样的伪影。如果您的模糊效果不平滑或看起来有静态噪点，请尝试将该值减小到 1 或更低。
高质量采样-High Quality Sampling	使用更高质量的采样来减少闪烁并提高整体模糊平滑度。这可能会导致一些性能成本。

2） 散景景深-Bokeh Depth of Field

Property	Description
Focus Distance	设置从相机到焦点的距离。
Focal Length	设置相机传感器和相机镜头之间的距离（以毫米为单位）。值越大，景深越浅。
光圈-Aperture	设置光圈比率（称为 f-stop 或 f-number）。该值越小，景深越浅。
Blade Count	使用滑块设置相机用来形成光圈的光圈叶片数量。使用的光圈叶片越多，散景就越圆。
Blade Curvature	使用滑块设置相机用来形成光圈的光圈叶片曲率。 该值越小，光圈叶片越明显。值为 1 使散景完美地呈现圆形。
Blade Rotation	使用滑块以度数设置光圈叶片的旋转。

8. 帕尼尼投影-Panini Projection

帕尼尼投影会对画面进行变形操作，让画面看起来好像包裹在一个圆柱体上（中间凸起）。

Property	Description
距离-Distance	用于控制强度。数值越大、中间越凸起。
裁剪以适应屏幕-Crop to Fit	裁剪画面以适应屏幕尺寸。

9. 镜头畸变-Lens Distortion

镜头畸变用于模拟真实世界中镜头的畸变效果。

Property	Description
强度-Intensity	用于控制镜头畸变的强度。数值越大，镜头效果越扭曲。
X 轴乘数-X Mutiplier	控制 X 轴向的畸变强度。如果数值为 0，则 X 轴向没有畸变。
Y 轴乘数-Y Mutiplier	控制 Y 轴向的畸变强度。如果数值为 0，则 Y 轴向没有畸变。
Center	设置画面畸变发生的中心点。
缩放-Scale	使用滑块设置全局屏幕缩放的值。这会缩放渲染以隐藏屏幕的边框。 当畸变过度时，画面周围一圈的像素点会变形得太厉害导致无法使用，这时可以增大该数值，把画面放大，以隐藏周围一圈变形过度的像素。

10. 运动模糊-Motion Blur

用于模拟真实世界中，因为相机或物体的移动速度大于曝光速度而导致的模糊效果。通用渲染管线 (URP) 只会模糊相机运动。

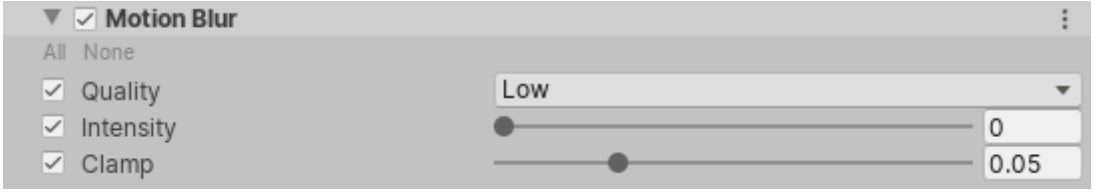


图19. Motion Blur

Property	Description
质量-Quality	设置效果的质量。较低的预设提供更好的性能，但视觉质量较低。
强度-Intensity	将运动模糊滤镜的强度设置为 0 到 1 之间的值。值越高，模糊效果越强，但可能导致性能降低，具体取决于 Clamp 参数。
Clamp	设置相机旋转产生的速度可以具有的最大值。这限制了高速模糊，以避免过多的性能成本。取值范围为 0 到 0.2。默认值为 0.05。

1） 解决性能问题

要降低运动模糊对性能的影响，可以降低质量。较低的质量设置可提供较高的性能，但可能会出现更多的视觉伪影。减小 Clamp 以减小用于 Unity 的最大速度。值越低，性能越高。

11. 色差-Chromatic Aberration

用于模拟真实镜头会产生的色差现象。色差现象产生的本质是因为镜头无法把所有颜色放到同一个点上，导致的颜色失真。

Property	Description
强度-Intensity	设置色差效果的强度。值介于 0 和 1 之间。值越高，效果越强烈。默认值为 0，即禁用效果。

12. 晕映-Vignette

可以将画面周围变暗或者减少饱和度。在真实相机上也会发生类似的现象，这通常是因为叠加的滤镜或者第二套镜头导致的。我们可以通过这样的方式将观众的视觉引导到画面中央区域。

Property	Description
Color	在这里挑选晕映的颜色。
中心点-Center	默认的中心点为正中心，修改值可以调节中心点的位置。
Intensity	设置晕影效果的强度。
平滑度-Smoothness	值越高，晕映边框越平滑。
是否为圆形-Rounded	如果勾选该选项，则整个变暗区域的形状就会是正圆形。禁用时，晕影与当前纵横

	比的形状相匹配。

13. 颜色分级-Lift Gamma Gain

此效果可执行三种颜色分级。Lift Gamma Gain 轨迹球遵循 ASC CDL 标准。当调整轨迹球上点的位置时，它会将图像的色调移向给定色调范围内的该颜色。使用不同的轨迹球来影响图像内的不同范围。调整轨迹球下方的滑块可以控制该范围的颜色亮度。

Property	Description
Lift	控制画面中暗色调的颜色和亮度，可以使用该选项设置更夸张的阴影效果。
Gamma	控制画面中中度色调的颜色和亮度。
Gain	控制画面中高亮区域的颜色和亮度，该设置会让高亮的地方更亮。

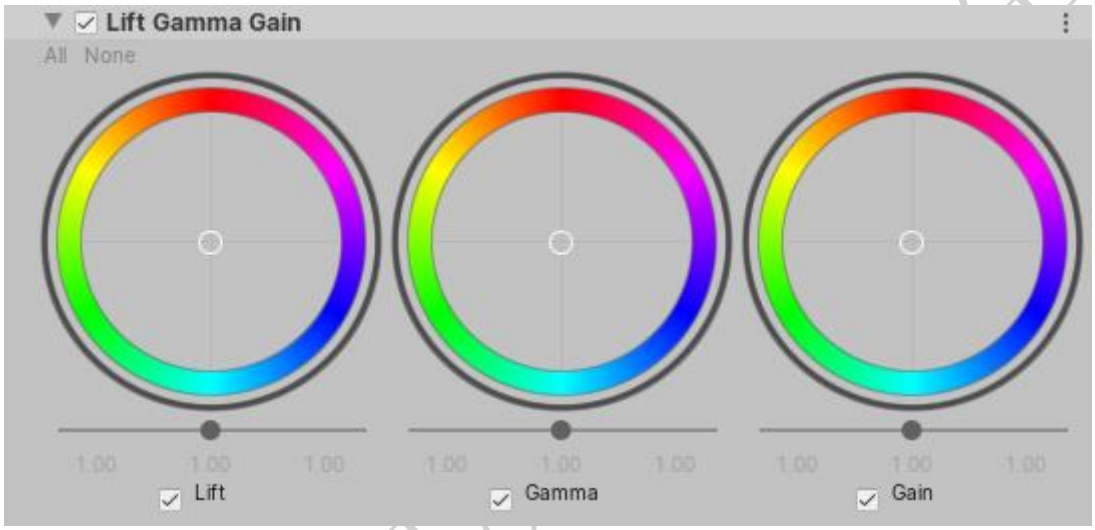


图20. LiftGammaGain

14. 通道混合-Channel Mixer

通过对红色通道、绿色通道和蓝色通道中的 RGB 数值进行调整，对整个画面的 RGB 值进行调整。在默认情况下红色通道的 Red 值为 100，其他为 0；绿色通道的 Green 值为 100，其他为 0；蓝色通道的 Blue 值为 100，其他为 0。

如果您增加绿色通道对红色通道整体混合的影响，则最终图像的所有绿色区域（包括中性/单色）都会染成更偏红的色调。

图21. Channel Mixer

Property	Description
Red	设置红色通道对所选输出通道的影响。
Green	设置绿色通道对所选输出通道的影响。

Blue	设置蓝色通道对选定输出通道的影响。

15. 颜色曲线- Color Curves

使用曲线控制画面中指定区域的色彩、饱和度或者亮度。使用颜色曲线可以实现色彩替换或者调整饱和度。

Property	Description
Master	该曲线影响整个图像的亮度。图表的 x 轴表示输入亮度，y 轴表示输出亮度。
Red	该曲线影响整个图像的红色通道强度。图表的 x 轴表示输入强度，y 轴表示红色通道的输出强度。
Green	该曲线影响整个图像的绿色通道强度。图表的 x 轴表示输入强度，y 轴表示绿色通道的输出强度。
Blue	该曲线影响整个图像的蓝色通道强度。图表的 x 轴表示输入强度，y 轴表示蓝色通道的输出强度。
色相与色相-Hue Vs Hue	该曲线根据输出色调（y 轴）移动输入色调（x 轴）。您可以使用它来微调特定范围的色调或执行颜色替换。
色相与饱和度-Hue Vs Sat	该曲线根据输入色调（x 轴）调整饱和度（y 轴）。可以使用它来淡化特别明亮的区域或创建艺术效果。
饱和度与饱和度-Sat Vs Sat	该曲线根据输入饱和度（x 轴）调整饱和度（y 轴）。您可以使用它来微调饱和度调整颜色调整.
流明和饱和度-Lum Vs Sat	该曲线根据输入亮度（x 轴）调整饱和度（y 轴）。您可以使用它来降低黑暗区域的饱和度，以提供有趣的视觉对比。

16. 颜色调节-Color Adiustment

使用此效果来调整最终渲染图像的整体色调、亮度和对比度。

Property	Description
后曝光-Post Exposure	以 EV（不是 EV 100）调整场景的整体曝光。URP 在 HDR 效果之后和色调映射之前应用此功能。
对比度-Contrast	调大其数值时，会扩展色调的范围；反之会缩小色调的范围。
颜色滤镜-Color Filter	为整个画面叠加所选颜色。
色彩偏移-Hue Shift	调整画面中所有的颜色。
饱和度-Saturation	调整画面中所有颜色的饱和度。

17. 分离调色-Split Toning

根据画面中不同区域的亮度，调整相关区域的整体颜色。可以为画面中的阴影和高光

区域分别指定不同的颜色。

Property	Description
阴影-Shadows	为阴影区域设置颜色。
高光-Highlights	为高光区域设置颜色。
平衡-Balance	大于 0 的数值会让整体画面的颜色偏向高光区的颜色；小于 0 的数值会让整体画面的颜色偏向阴影区的颜色。

18. 阴影、中间调、高光-Shadow、Midtones、Highlights

可用于精确控制画面中的阴影、中间调和高光区域的颜色和亮度（比使用 Lit、Gamma 和 Gain 参数要准确很多）。

1） Shadow Limits

Property	Description
Start	设置渲染的阴影和中间色调之间过渡的起点。
End	设置渲染的阴影和中间色调之间的过渡结束点。

2） Highlight Limits

Property	Description
Start	设置中间调和高光之间的过渡起点。
End	设置中间调和高光之间的过渡结束点。

十五、 Unity 反射

哔哩哔哩次元游戏人生

十六、 四元数

十七、 其他知识

1. 图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别？

Sprite 作为 UI 精灵使用，Texture 作为模型贴图使用。

2. 使用 Unity3d 实现 2d 游戏，有几种方式？

① 使用本身的 GUI、UGUI。

② 把摄像机的 Projection(投影)值调为 Orthographic(正交投影)，不考虑 z 轴。

③ 使用 2d 插件，如：2DToolKit、NGUI。

3. 向量的点乘、叉乘以及归一化的意义？

① 点乘描述了两个向量的相似程度，结果越大两向量越相似，还可表示投影。

② 叉乘得到的向量垂直于原来的两个向量。

③ 标准化向量：用在只关系方向，不关心大小的时候。

十八、 .meta 文件

1. .meta 文件的创建

当新的资源导入 unity 时，会自动创建一个同级同名的.meta 文件。

2. .meta 文件的作用

.meta 文件中记录的信息主要包含：

1) GUID(Globally Unique Identifier)，唯一 ID

Unity 是基于 GUID 去管理资源的，如果 GUID 丢失或者发生改变，则很可能造成相关引用丢失。使用 GUID 的好处就是，即使你移动、重命名或者修改资源的内容，这个资源仍然可以通过 GUID 来被引用（只要 GUID 不变，资源就能够被引用）。

2) Importer 信息

根据资源类型的不同，Import 数据也不同。比如文件夹是"DefaultImporter"，Asset 是"NativeFormatImporter"，字体文件是"TrueTypeFontImporter"，纹理是"TextureImporter"等。

3. 常见的错误操作

① 在版本管理的时候(svn/git)提交新资源但忘记提交它的.meta 文件。

② 直接在文件夹管理器中（不是在 Unity 中），只移动资源，忽略了同名.meta。

③ 直接在文件夹管理器中（不是在 Unity 中），重命名资源文件，但没有同步重命名.meta 文件。

④ 手动删除.meta 文件。

以上情况,Unity 会自动为这个资源生成新的.meta 文件，此时 GUID 也会发生改变，导致资源引用失效。

1) 特别强调一种错误提交操作

（这个操作其实属于场景错误的一种）

某些工种不需要打开 Unity 工作，但是却又需要往 Unity 目录中提交资源，新资源没有经过 Unity 生成.meta 文件后同步提交，就会导致其他人员在更新文件下来后，都自动生成了一个不同的 GUID.meta 文件，而且.meta 文件都处于待提交状态！如果一个人把这个新资源用于开发制作，并且提交了.meta 文件，那么其他人就会有.meta 冲突。在解决冲突的时候如果使用了自己的.meta 文件，还可能导致引用丢失！

哔哩哔哩次元游戏人生

Unity Script

一、 GameObject

1. 属性（Properties）

1） activeInHierarchy

定义场景中的游戏对象是否处于可见状态。

2） isStatic

可以获取和设置游戏对象的静态编辑标志。

3） layer

游戏对象所在的层级。

4） Scene

游戏对象所属的场景。

5） sceneCullingMask

用于确定在哪个场景中渲染游戏对象。

6） Tag

这个游戏对象的标签。

7） transform

游戏对象的 transform component。

2. 构造函数（Constructors）

1） GameObject

创建一个名为 name 的新的游戏对象。

3. 公共方法

1） AddComponent

将一个新的组件类添加到游戏对象。

2） BroadcastMessage

在此游戏对象或其任何子项中的每个 MonoBehaviour 上调用名为 methodName 的方法。

3） CompareTag

比较游戏对象的 Tag

4） GetComponent

获取游戏对象上的组件，如果有要获取的组件就返回该类型的组件，如果没有，就返回 null。

5） GetComponentInChildren

使用深度优先搜索返回该游戏对象或其任何子项中该类型的组件。（没有返回空）

6） GetComponentInParent

获取在该游戏对象或其任何父项中指定类型的组件。（有只返回一个，没有返回空）

7） GetComponents

返回该游戏对象中指定类型的所有组件。

哔哩哔哩次元游戏人生

8） GetComponentsInChildren

返回该游戏对象或其任何子项中指定类型的所有组件。

9） GetComponentsInParent

返回该游戏对象或其任何父项中指定类型的所有组件。

10）SendMessage

在此游戏对象中的每个 MonoBehaviour 上调用名为 methodName 的方法。

11）SendMessageUpwards

在此游戏对象中的每个 MonoBehaviour 和行为的每个父节点上调用名为 methodName 的方法。

12）SetActive

根据给定的布尔值决定激活还是停用该游戏对象。

13）TryGetComponent

尝试获取指定类型的组件。

4. 静态方法

1） Find

按名称在场景中查找游戏对象并返回它。

2） FindGameObjectsWithTag

根据 Tag 返回一组激活状态的游戏对象。如果未找到游戏对象，则返回空数组。

3） FindWithTag

根据 Tag 返回一个激活状态的游戏对象。如果未找到游戏对象，则返回 null。

二、 Transform

1. 公共方法

1） LookAt

① 声明

public void LookAt(Transform target);

public void LookAt(Transform target, Vector3 worldUp = Vector3.up);

② Parameters-参数

target-要指向的对象。

worldUp-指定向上方向的向量。

③ 描述

旋转变换，使前向矢量指向 target 的当前位置。然后它旋转变换以将其向上方向向量指向向量暗示的方向 worldUp。如果省略该 worldUp 参数，该函数将使用世界 y 轴。worldUp 如果向前方向垂直于，则旋转的向上矢量将仅与矢量匹配 worldUp。

④ 声明

public void LookAt (Vector3 worldPosition);

public void LookAt (Vector3 worldPosition , Vector3 worldUp = Vector3.up);

⑤ Parameters-参数

worldPosition-看向某点

worldUp-指定向上方向的向量。

④ 描述

旋转变换，使前向向量指向 worldPosition。然后它旋转变换以将其向上方向向量指向

哔哩哔哩次元游戏人生

向量暗示的方向 worldUp。如果省略该 worldUp 参数，该函数将使用世界 y 轴。worldUp 如果向前方向垂直于，则旋转的向上矢量将仅与矢量匹配 worldUp。

三、 Vector3

1. 描述

3D 向量和点的表示。

这种结构在整个 Unity 中用于传递 3D 位置和方向。它还包含用于执行常见向量运算的函数。

除了下面列出的函数，其他类也可用于操作向量和点。例如 Quaternion 和 Matrix4x4 类对于旋转或变换向量和点很有用。

2. Static Properity-静态属性

1） back

Vector3(0, 0, -1) 的简写。

2） down

Vector3(0, -1, 0) 的简写。

3） forward

Vector3(0, 0, 1) 的简写。

4） left

Vector3(-1, 0, 0) 的简写。

5） negativeInfinity

Vector3(float.NegativeInfinity, float.NegativeInfinity, float.NegativeInfinity) 的简写。

6） one

Vector3(1, 1, 1) 的简写。

7） positiveInfinity

Vector3(float.PositiveInfinity, float.PositiveInfinity, float.PositiveInfinity) 的简写。

8） right

Vector3(1, 0, 0) 的简写。

9） up

Vector3(0, 1, 0) 的简写。

10）zero-零向量

Vector3(0, 0, 0) 的简写。

3. Static Methods-静态方法

1） Angle

计算两个 3 维向量之间的角度

2） ClampMagnitude-限制模

返回一个向量的副本，其大小被限制为 maxLength。

3） Cross

两个向量的叉乘。

4） Distance

返回 a 向量和 b 向量之间的距离。

咪哩咔哩次元游戏人生

5） Dot

两个向量的点乘。

6） Lerp

在两点之间线性插值。

7） LerpUnclamped

在两个向量之间线性插值。

8） Max

返回由两个向量的最大分量组成的向量。

9） Min

返回由两个向量的最小分量组成的向量。

10）MoveTowards

计算当前位置和指定的位置之间的位置，移动不超过 maxDistanceDelta 指定的距离。

11）Normalize

使这个向量的大小为 1。

12）OrthoNormalize

使向量标准化并相互正交。（即两两向量内积为 0）

13）Project

将一个向量投影到另一个向量上。

14）ProjectOnPlane

将向量投影到由垂直于该平面的法线定义的平面上。

15）Reflect

从法线定义的平面反射向量。

16）RotateTowards

向目标旋转。

17）Scale

将两个向量相乘。

18）SignedAngle

计算向量 from 和 to 之间相对于轴的有符号角。

19）Slerp

在两个向量之间进行球面插值。

20）SlerpUnclamped

在两个向量之间进行球面插值。

21）SmoothDamp

随着时间的推移，逐渐将向量改变为期望的目标。

4. 向量归一化的区别

1） 不同点

Vector3.normalized：不改变当前向量，返回一个新的规范化的向量。

Vector3.Normalized：改变当前向量，当前向量长度为一。

哔哩哔哩次元游戏人生

2） 相同点

实现规范化，让一个向量保持相同的方向，但它的长度为 1，如果这个向量太小而不能被规范化，一个零向量会被返回。

四、 KeyCode 编码

值	描述	值	描述
BackSpace	退格键	Delete	正向删除键
Tab	制表键	Clear	清除键
Return	回车键	Pause	PC 上的暂停键
Escape	退出键	Space	空格键
Keypad0	数字小键盘 0	Keypad1	数字小键盘 1
Keypad2	数字小键盘 2	Keypad3	数字小键盘 3
Keypad4	数字小键盘 4	Keypad5	数字小键盘 5
Keypad6	数字小键盘 6	Keypad7	数字小键盘 7
Keypad8	数字小键盘 8	Keypad9	数字小键盘 9
KeypadPeriod	数字小键盘句号	KeypadDivide	数字小键盘除号
KeypadMultiply	数字小键盘乘号	KeypadMinus	数字小键盘减号
KeypadPlus	数字小键盘加号	KeypadEnter	数字小键盘回车键
KeypadEquals	数字小键盘等号	UpArrow	上箭头键
DownArrow	下箭头键	RightArrow	右箭头键
LeftArrow	左箭头键	Insert	插入键
Home	起始键	End	结束键
PageUp	上一页	PageDown	下一页
F1	F1 功能键	F2	F2 功能键
F3	F3 功能键	F4	F4 功能键
F5	F5 功能键	F6	F6 功能键
F7	F7 功能键	F8	F8 功能键
F9	F9 功能键	F10	F10 功能键
F11	F11 功能键	F12	F12 功能键
Alpha0	键盘顶部的数字键 0	Alpha1	键盘顶部的数字键 1
Alpha2	键盘顶部的数字键 2	Alpha3	键盘顶部的数字键 3
Alpha4	键盘顶部的数字键 4	Alpha5	键盘顶部的数字键 5
Alpha6	键盘顶部的数字键 6	Alpha7	键盘顶部的数字键 7
Alpha8	键盘顶部的数字键 8	Alpha9	键盘顶部的数字键 9
Exclaim	感叹号键	DoubleQuote	双引号键
Hash	哈希键	Dollar	美元符号键
Percent	百分号键	Ampersand	&键
Quote	单引号键	LeftParen	左括号键
RightParen	右括号键	Asterisk	星号键
Plus	加号键	Comma	逗号键
Minus	减号键	Period	句号键
Slash	斜杠键	Colon	冒号键
Semicolon	分号键	Less	小于符号键

哔哩哔哩欢乐新天地

Equals	等号键	Greater	大于符号键
Question	问号键	At	@符号键
LeftBracket	左中括号键	Backslash	反斜杠键
RightBracket	右中括号键	Caret	托字符键
Underscore	下划线键	BackQuote	反引号键
A	字母 A 键	B	字母 B 键
C	字母 C 键	D	字母 D 键
E	字母 E 键	F	字母 F 键
G	字母 G 键	H	字母 H 键
I	字母 I 键	J	字母 J 键
K	字母 K 键	L	字母 L 键
M	字母 M 键	N	字母 N 键
O	字母 O 键	P	字母 P 键
Q	字母 Q 键	R	字母 R 键
S	字母 S 键	T	字母 T 键
U	字母 U 键	V	字母 V 键
W	字母 W 键	X	字母 X 键
Y	字母 Y 键	Z	字母 Z 键
LeftCurlyBracket	左大括号键	Pipe	|
RightCurlyBracket	由大括号键	Tilde	波浪号
Numlock	小键盘锁定键	CapsLock	大写锁定键
ScrollLock	滚动锁定键	RightShift	右换挡键
LeftShift	左换挡键	RightControl	右控制键
LeftControl	左控制键	RightAlt	右 Alt 键
LeftAlt	左 Alt 键	LeftMeta	如果在输入管理器设置中启用了物理键，则映射到左 Windows 键或左命令键，否则仅映射到左命令键。
LeftCommand	左命令键	LeftApple	左 Apple 键
LeftWindows	左 Windows 键	RightMeta	如果在输入管理器设置中启用了物理键，则映射到右侧 Windows 键或右侧 Command 键，否则仅映射到右侧 Command 键。
RightCommand	右命令键	RightApple	右 Apple 键
RightWindows	右 Windows 键	AltGr	右侧更改键
Help	帮助键	Print	打印键
SysReq	系统请求键	Break	中断键
Menu	菜单键	Mouse0	第 1 个鼠标键（鼠

哔哩哔哩欢乐游戏人生

			标左键）
Mouse1	第 2 个鼠标键（鼠标右键）	Mouse2	第 3 个鼠标键（鼠标中键）
Mouse3	第 4 个鼠标键	Mouse4	第 5 个鼠标键
Mouse5	第 6 个鼠标键	Mouse6	第 7 个鼠标键

五、 数学函数库

1. 绝对值-Mathf.Abs

① public static float Abs(float f);

返回浮点型的绝对值

② public static int Abs(int value);

返回整型的绝对值

2. 反余弦-Mathf.Acos

public static float Acos(float f);

根据余弦值，返回弧度。

3. 近似-Mathf.Approximately

public static bool Approximately(float a, float b);

比较两个浮点数值，看它们是否非常接近，由于浮点数值不精确，不建议使用等于来比较它们。

4. 反正弦-Mathf.Asin

public static float Asin(float f);

根据正弦值，返回弧度。

5. 反正切-Mathf.Atan2

public static float Atan2(float y, float x);

根据 Y 除以 X，返回对应的弧度。返回值表示相对直角三角形，对角的角，其中 X 是临边边长，Y 是对边边长。返回值是 X 轴和一个二维向量开始于 0，结束在(x,y)处之间的角。

6. 反正切-Mathf.Atan

public static float Atan(float f);

根据正切值，返回以弧度为单位的角度，返回值介于负二分之 π 和正二分之 π 之间。

7. 向上最小整数- Mathf.CeilToInt

public static int CeilToInt(float f);

返回大于或等于 f 的最小整数，返回值是整型。

8. 上限值-Mathf.Ceil

public static float Ceil(float f);

返回大于或等于 f 的最小整数，返回值是浮点型。

9. 范围限制 0-1-Mathf.Clamp01

public static float Clamp01(float value);

限制值的范围在 0 到 1 之间。如果值小于 0 则返回 0，值大于 1 则返回 1。

10. 限制- Mathf.Clamp

① public static float Clamp(float value, float min, float max);

哔哩哔哩次元游戏人生

如果给定的浮点值小于最小值，则返回最小值。如果给定值大于最大值，则返回最大值。使用 Clamp 将值限制在由最小值和最大值定义的范围内。

注意：如果最小值大于最大值，则该方法返回最小值。

② public static int Clamp (int value , int min , int max);

将给定值限制在由给定最小整数和最大整数值定义的范围之间。如果它在最小值和最大值之间，则返回给定值。

如果给定值小于最小值，则返回最小值。如果给定值大于最大值，则返回最大值。min 和 max 参数包括在内。例如，Clamp(10, 0, 5) 将返回最大参数 5 而不是 4。

11. 最近的二次方- Mathf.ClosestPowerOfTwo

public static int ClosestPowerOfTwo(int value);

返回距离 Value 值最近的 2 的 n 次幂，Mathf.ClosestPowerOfTwo(7)最近值为 8，Mathf.ClosestPowerOfTwo(19)最近值为 16。

12. 余弦- Mathf.Cos

public static float Cos(float f);

f 为输入的角度，以弧度为单位。返回由 f 指定的余弦值，值介于负 1 到 1 之间。

13. 度转弧度- Mathf.Deg2Rad

public static float Deg2Rad;

将角度转换为弧度值，其值等于$\frac{2\pi}{360}$。

14. 弧度转度- Mathf.Rad2Deg

public static float Rad2Deg;

弧度值转换为角度，其值等于$\frac{360}{2\pi}$。

15. 增量角- Mathf.DeltaAngle

public static float DeltaAngle(float current, float target);

计算以度为单位的两个给定角度之间的最短差。

16. 小正数- Mathf.Epsilon

public static float Epsilon;

一个很小的浮点值（只读）。浮点数可以具有的不为零的最小值。

① 规则：

anyValue + Epsilon = anyValue

anyValue - Epsilon = anyValue

0 + Epsilon = Epsilon

0 - Epsilon = -Epsilon

② Example:

```
bool isEqual(float a, float b)
{
    if (a >= b - Mathf.Epsilon && a <= b + Mathf.Epsilon)
    {
        return true;
    }
    else
    {
```

陈理想次元游戏人生

```
            return false;
        }
    }
```

17. 指数- Mathf.Exp

public static float Exp(float power);

反回 e 的 Power 次方的值。

18. 向下最小整数- Mathf.FloorToInt

public static int FloorToInt(float f);

返回小于或等于 f 的最大整数。

Mathf.FloorToInt (10.7f)，返回值为 10。

Mathf.FloorToInt(-10.7f)，返回值为-11。

19. 向下取整- Mathf.Floor

public static float Floor(float f);

返回小于或等于 f 的最大整数。

20. 正无穷- Mathf.Infinity

public static float Infinity;

表示正无穷，也就是无穷大，∞。

21. 反差值- Mathf.InverseLerp

public static float InverseLerp(float a, float b, float value);

计算 value 位于 a 和 b 之间的位置，其值为均匀分布的分布函数，即$\frac{x-a}{b-a}$。

22. 是否为 2 的幂- Mathf.IsPowerOfTwo

public static bool IsPowerOfTwo(int value);

如果该值是 2 的幂，返回 true。

23. 插值角度- Mathf.LerpAngle

public static float LerpAngle(float a, float b, float t);

和 lerp 类似。

24. 插值- Mathf.Lerp

public static float Lerp(float a, float b, float t);

基于浮点数 t，返回 a 到 b 之间的差值，t 限制在 0-1 之间。

当 t 等于 0 时，返回 a 的值；当 t 等于 1 时，返回 b 的值；当 t 等于 0.5 时，返回 a 和 b 的中点。

25. 基数为 10 的对数- Mathf.Log10

public static float Log10(float f);

返回指定数字的以 10 为底的对数。

26. 对数- Mathf.Log

① public static float Log(float f, float p);

返回指定基数中指定数字的对数。

② public static float Log(float f);

返回指定数字的自然对数（以 e 为底）。

哔哩哔哩次元游戏人生

27. 最大值- Mathf.Max

public static float Max(float a, float b);

public static float Max(params float[] values);

返回两个或多个值中的最大值。

28. 最小值- Mathf.Min

public static float Min(float a, float b);

public static float Min(params float[] values);

返回两个或多个值中的最小值。

29. 移动角- Mathf.MoveTowardsAngle

public static float MoveTowardsAngle(float current, float target, float maxDelta);

和 MoveTowards 类似。

变量 current 和 target，假定以度为单位。maxDelta 出于优化原因，不支持负值。要 current 远离目标角度，请改为在该角度上增加 180。

30. 移向- Mathf.MoveTowards

public static float MoveTowards(float current, float target, float maxDelta);

将值 current 移向 target。

这与 Mathf.Lerp 基本相同，但该函数将确保移动速度永远不会超过 maxDelta。maxDelta 为负值，将会让 current 远离 target。

31. 负无穷- Mathf.NegativeInfinity

public static float NegativeInfinity;

负无穷，-∞。

32. 下一个 2 的幂- Mathf.NextPowerOfTwo

public static int NextPowerOfTwo(int value);

返回大于或等于该值的，2 的幂。

Mathf.NextPowerOfTwo(7)，返回 8。

Mathf.NextPowerOfTwo(139)，返回 256。

Mathf.NextPowerOfTwo(256)，返回 256。

33. 乒乓- Mathf.PingPong

public static float PingPong(float t, float length);

PingPong 返回一个值，该值将在值 0 和 length 之间来回震荡。

PingPong 要求值 t 是自增值，例如 Time.time 和 Time.unscaledTime。

34. 圆周率- Mathf.PI

public static float PI;

3.1415926

35. 次方- Mathf.Pow

public static float Pow(float f, float p);

返回 f 的 p 次方。

36. 重复- Mathf.Repeat

public static float Repeat(float t, float length);

循环值 t，使其永远不会大于 length 且永远不会小于 0。这类似于模运算符，但它适用于浮点数。例如当 t 为 3.0 且 length 为 2.5，结果将是 0.5。当 t 为 5 且 length 为 2.5，

哔哩哔哩次元游戏人生

结果将为 0.0。但是请注意，该行为并未像模运算符那样为负数定义。

37. 四舍五入到整数- Mathf.RoundToInt

public static int RoundToInt(float f);

将 f 四舍五入到最接近的整数。

如果数字以 .5 结尾，它介于两个整数之间，其中一个是偶数，另一个是奇数，那么返回偶数。

38. 四舍五入- Mathf.Round

public static float Round(float f);

将 f 四舍五入到最接近的整数。

如果数字以 .5 结尾，它介于两个整数之间，其中一个是偶数，另一个是奇数，那么返回偶数。

39. 有向- Mathf.Sign

public static float Sign(float f);

当 f 为正或 0 时，返回 1；为负时返回-1.

40. 正弦- Mathf.Sin

public static float Sin(float f);

返回 f 所对应的正弦值，返回值介于负 1 到正 1 之间。

41. 平方根- Mathf.Sqrt

public static float Sqrt(float f);

返回 f 的平方根。

42. 正切- Mathf.Tan

public static float Tan(float f);

返回 f 的正切值，f 以弧度为单位。

43. 平滑阻尼角度- Mathf.SmoothDampAngle

public static float SmoothDampAngle(float current, float target, ref float currentVelocity, float smoothTime, float maxSpeed = Mathf.Infinity, float deltaTime = Time.deltaTime);

随着时间的推移逐渐改变当前角度到期望的角度，这个函数可以用来平滑任何一种值、颜色、位置、变量。

44. 平滑阻尼- Mathf.SmoothDamp

public static float SmoothDamp(float current, float target, ref float currentVelocity, float smoothTime, float maxSpeed = Mathf.Infinity, float deltaTime = Time.deltaTime);

随着时间的推移，逐渐改变一个值到目标值。

45. 平滑插值- Mathf.SmoothStep

public static float SmoothStep(float from, float to, float t);

此函数在 Lerp 之间进行插值，min 和 max 以与 Lerp 类似的方式进行插值。但是，插值会从开始逐渐加速，并逐渐减慢到结束。这对于创建淡入淡出和其他过渡非常有用。

六、 UGUI 事件接口

命名空间：Using UnityEnging.EventSystems;

1. IPointerEnterHandler

当鼠标光标移入该对象时触发。

降维次元游戏人生

2. IPointExitHandler

当鼠标光标移出该对象时触发。

3. IPointDownHandler

鼠标点击 A 对象，按下鼠标时 A 对象响应此事件。

4. IPointUpHandler

鼠标点击 A 对象，抬起鼠标时 A 对象响应此事件。(无论鼠标在何处抬起，都会在 A 对象中响应此事件)响应此事件的前提，是 A 对象必须响应过 IPointDownHandler。

5. IPointClickHandler

鼠标点击 A 对象，抬起鼠标时 A 对象响应此事件（鼠标按下和抬起时鼠标要处于同一对象上）。

6. IDragHandler

当鼠标在 A 对象按下并拖拽时，A 对象每帧响应一次此事件。（如果不实现此接口，则后面的四个接口方法都不会触发）

7. IInitializePotentialDragHandler

当鼠标在 A 对象按下还没开始拖拽时，A 对象响应此事件。此接口事件与 IPointDownHandler 接口类似，二者的执行顺序：先执行 IPointDownHandler，然后 执行此接口事件（需先实现 IDragHandler 接口）

8. IBeginDragHandler

当鼠标在 A 对象按下并开始拖拽时，A 对象响应此事件，此事件在 IInitializePotentialDragHandler 之后响应，在 OnDrag 之前响应。

9. IEndDragHanlder

当鼠标抬起时，A 对象响应此事件。

10. IDropHandler

A、B 对象必须均实现 IDropHandler 接口，且 A 至少实现 IDragHandler 接口。当鼠标从 A 对象上开始拖拽，在 B 对象上抬起时，B 对象响应此事件。此时 name 获取的是 B 对象的 name 属性，eventData.PointerDrag 表示发起拖拽的对象

哔哩哔哩次元游戏人生

Unity 案例

一、 2D 角色移动

二、 3D 角色移动

三、 3D 相机控制

四、 输入管理-InputManager

构建输入管理，主要使用事件来传递玩家输入的信息。

```
public class InputManager : SingleComponent<InputManager>
{
    #region 水平输入事件完整声明格式
    /*
    * 事件的拥有者(Event Source) -----> InputManager 类
    * 事件(Event) -----> OnSelectSkill
    * 事件的响应者(Event Subscriber) -----> EquipmentSlotParent
    * 事件处理器(Event Handler) -----> SetSelectSkill
    * 事件的订阅关系(+= Operator) -----> +=
    */
    public delegate void InputHorizontalEventHandler(float _horizontalValue);//为了事件而声明的委托类型最好加上 EventHandler 后缀,这个委托可以指向一个无返回值的方法，
    private InputHorizontalEventHandler inputHorizontalEventHandler;//完整声明格式中最重要的就是这个委托类型的字段，用来存储事件处理器
    public event InputHorizontalEventHandler OnInputHorizontal
    {
        add
        {
            inputHorizontalEventHandler += value;//我们虽然不知道未来传进来的事件处理器是什么，我们可以用上下文关键字 Value 来表示
        }
        remove
        {
            inputHorizontalEventHandler -= value;//移除事件处理器
        }
    }
    #endregion
    //action 委托必定没有返回值,func 委托必定具有一个返回值
    public event Action<float> OnInputVertical;//垂直输入事件的简略声明

    protected override void Awake()
    {
        base.Awake();
```

哔哩哔哩次元游戏人生

```
    }

    private void Update()
    {
        #region 水平方向输入
        if (inputHorizontalEventHandler != null)
        {
            inputHorizontalEventHandler(Input.GetAxisRaw("Horizontal"));//只能由事件拥有者在其内部调用内部逻辑调用
        }
        #endregion
        #region 垂直方向输入
        if (OnInputVertical != null)
        {
            OnInputVertical(Input.GetAxisRaw("Vertical"));
        }
        #endregion
    }
}
```

五、 场景异步加载与过渡

六、 自定义 URP 后处理

七、 通用对象池

1. 概述

对象池模式(The Object Pool Pattern)是单例模式的一个变种，它提供了获取一系列相同对象实例的入口。当你需要对象来代表一组可替代资源的时候就变的很有用，每个对象每次可以被一个组件使用。

2. 理解

对象池模式管理一个可代替对象的集合。组件从池中借出对象，用它来完成一些任务并当任务完成时归还该对象。被归还的对象接着满足请求，不管是同一个组件还是其他组件的请求。对象池模式可以管理那些代表的现实资源或者通过重用来分摊昂贵初始化代价的对象。

哔哩哔哩次元游戏人生

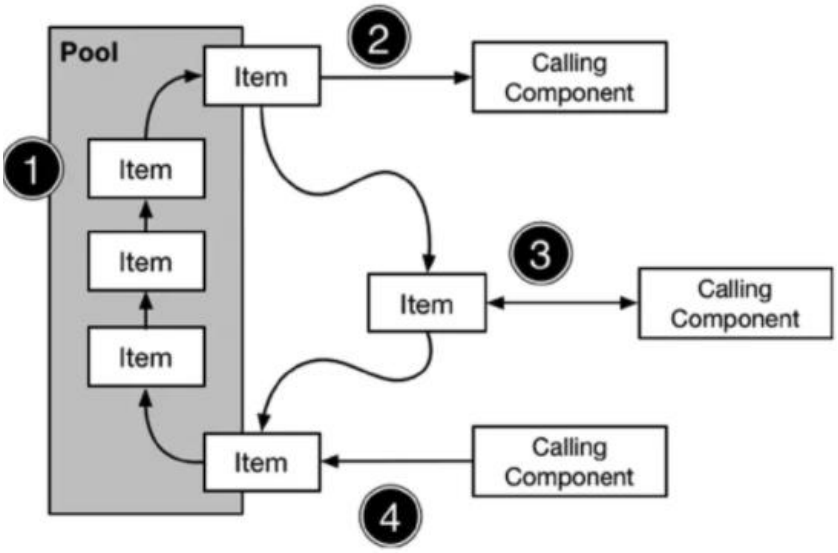


图22.　　对象池

第一步操作是构建对象池。

第二步操作是借出对象。

第三步操作是组件用借出的对象来完成一些任务。这时候并不需要对象池再做什么，但是这也意味着该对象将被租借一段时间并且不能在被其他组件借出。

第四步操作就是归还，组件归还借出的对象这样可以继续满足其他的租借请求。

3. 优点

复用池中对象，没有分配内存和创建堆中对象的开销， 没有释放内存和销毁堆中对象的开销， 进而减少垃圾收集器的负担， 避免内存抖动；不必重复初始化对象状态， 对于比较耗时的 constructor 和 finalize 来说非常合适。

4. 缺点

① 并发环境中， 多个线程可能同时需要获取池中对象， 进而需要在堆数据结构上进行同步或者因为锁竞争而产生阻塞， 这种开销要比创建销毁对象的开销高数百倍。

② 由于池中对象的数量有限， 势必成为一个可伸缩性瓶颈。

③ 很难正确的设定对象池的大小， 如果太小则起不到作用， 如果过大， 则占用内存资源高， 可以起一个线程定期扫描分析， 将池压缩到一个合适的尺寸以节约内存，但为了获得不错的分析结果， 在扫描期间可能需要暂停复用以避免干扰(造成效率低下)， 或者使用非常复杂的算法策略(增加维护难度)。

④ 设计和使用对象池容易出错， 设计上需要注意状态同步， 这是个难点， 使用上可能存在忘记归还(就像 c 语言编程忘记 free 一样)， 重复归还(可能需要做个循环判断一下是否池中存在此对象， 这也是个开销)， 归还后仍旧使用对象(可能造成多个线程并发使用一个对象的情况)等问题。

5. 简单对象池

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

namespace UniversalPool
{
```

```
[DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
[AddComponentMenu("GameObject/PoolObject")]
public class PoolObject : MonoBehaviour
{
    public string poolName;//对象池名称
    public bool isPooled; //定义对象是在池中等待还是在使用中
}

public enum PoolInflationType//对象池膨胀类型
{
    INCREASE,// 当对象池膨胀时，将一个对象添加到池中。
    DOUBLE// 当对象池膨胀时，将池的大小增加一倍
}

class Pool
{
    private Stack<PoolObject> availableObjStack = new Stack<PoolObject>();//Stack
栈是一个先进后出（LIFO）表
    private Queue<PoolObject> availableObjQueue = new
Queue<PoolObject>();//Queue 队列是先进先出(FIFO)

    //未使用对象的根节点
    private GameObject rootObj;
    private PoolInflationType inflationType;
    private string poolName;
    private int objectsInUse = 0;

    //构造函数
    public Pool(string poolName, GameObject poolObjectPrefab, GameObject
rootPoolObj, int initialCount, PoolInflationType type,bool useStack = true)
    {
        if (poolObjectPrefab == null)
        {
#if UNITY_EDITOR
            Debug.LogError("[ObjPoolManager] 对象池预制体为空 !");
#endif
            return;
        }
        this.poolName = poolName;
        this.inflationType = type;
        this.rootObj = new GameObject(poolName + "Pool");//为对象池命名
        this.rootObj.transform.SetParent(rootPoolObj.transform,
false);//worldPositionStays 如果为 true，则修改父相对位置、缩放和旋转，以使对象保持与
以前相同的世界空间位置、旋转和缩放。
```

哔哩哔哩次元游戏人生

```
        GameObject go = GameObject.Instantiate(poolObjectPrefab);
        PoolObject po = go.GetComponent<PoolObject>();
        if (po == null)
        {
            po = go.AddComponent<PoolObject>();//为对象添加 PoolObject 组件
        }
        po.poolName = poolName;//设置对象所属对象池名称
        AddObjectToPool(po, useStack);//确保对象池中至少存在一个对象

        //填充对象池
        FillPool(Mathf.Max(initialCount, 1), useStack);
    }

    //添加到栈对象池,o(1)
    private void AddObjectToPool(PoolObject po,bool useStack = true)
    {
        po.gameObject.SetActive(false);
        po.gameObject.name = poolName;
        if (useStack)
        {
            availableObjStack.Push(po);//入栈
        }
        else
        {
            availableObjQueue.Enqueue(po);//入队列,将对象添加到。
        }
        po.isPooled = true;//对象是在对象池中
        //添加到根节点
        po.gameObject.transform.SetParent(rootObj.transform, false);
    }

    /// <summary>
    ///  填充对象池到指定数量
    /// </summary>
    /// <param name="initialCount">填充的数量</param>
    private void FillPool(int initialCount, bool useStack = true)
    {
        for (int index = 0; index < initialCount; index++)
        {
            PoolObject po = null;
            if (useStack)
            {
                po = GameObject.Instantiate(availableObjStack.Peek());//Peek()返
```

哔哩哔哩次元游戏人生

```
回位于栈顶的对象但不将其移除
                }
                else
                {
                    po = GameObject.Instantiate(availableObjQueue.Peek());//Peek()返
回位于队首的对象但不将其移除
                }
                AddObjectToPool(po, useStack);
            }
        }

        //栈中的下一个可用对象，O(1)
        public GameObject NextAvailableObject(bool autoActive,bool useStack = true)
        {
            PoolObject po = null;
            if (useStack ? availableObjStack.Count > 1 : availableObjQueue.Count > 1)//
判断当前是否有足够可使用对象
            {
                //有就出栈，Pop()删除并返回 Stack 顶部的对象，Dequeue 删除并
返回队首的对象
                po = useStack ? availableObjStack.Pop() :
availableObjQueue.Dequeue();
            }
            else
            {
                int increaseSize = 0;
                switch (inflationType)
                {
                    case PoolInflationType.INCREASE:
                        increaseSize = 1;
                        break;
                    case PoolInflationType.DOUBLE:
                        {
                            increaseSize = useStack ? availableObjStack.Count :
availableObjQueue.Count + Mathf.Max(objectsInUse, 0);
                            break;
                        }
                }
    #if UNITY_EDITOR
                Debug.Log(string.Format("Growing pool {0}: {1} populated",
poolName, increaseSize));
    #endif
                if (increaseSize > 0)
                {
```

哔哩哔哩次元游戏人生

```
                    FillPool(increaseSize, useStack);//填充对象池
                    if (useStack)
                    {
                        po = availableObjStack.Pop();//有就出栈，Pop()删除并返回
Stack 顶部的对象
                    }
                    else
                    {
                        po = availableObjQueue.Dequeue();//有就出队列，Dequeue
删除并返回队首的对象
                    }
                }
            }

            GameObject result = null;
            if (po != null)
            {
                objectsInUse++;//已使用对象数增加
                po.isPooled = false;
                result = po.gameObject;
                if (autoActive)
                {
                    result.SetActive(true);
                }
            }

            return result;
        }

        //将对象返回到对象池中，O(1)
        public void ReturnObjectToPool(PoolObject po,bool useStack = true)
        {
            if (poolName.Equals(po.poolName))//Equals(String)确定此实例是否与另一
个指定的 String 对象具有相同的值
            {
                objectsInUse--;//已使用对象减少
                /* 我们本可以使用 Stack.Contains(Object)来检查该对象是否在对象
池中。
                 * 但是会使此方法的时间复杂度变为 O（n）*/
                if (po.isPooled)
                {
    #if UNITY_EDITOR
                    Debug.LogWarning(po.gameObject.name + " 已经在对象池里
了。 您为什么要再次返回？ 请检查用法。");
```

哔哩哔哩次元游戏人生

```
#endif
                }
                else
                {
                    AddObjectToPool(po, useStack);
                }
            }
            else
            {
                Debug.LogError(string.Format("试图将对象添加到不正确的对象池中
{0} {1}", po.poolName, poolName));
            }
        }
    }
}
```

6. 对象池管理

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

namespace UniversalPool
{
    [DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
    [AddComponentMenu("GameObject/PoolResourceManager")]
    public class PoolResourceManager : MonoBehaviour
    {
        //使用数据字典为对象池添加关键字
        private Dictionary<string, Pool> poolDict = new Dictionary<string, Pool>();

        private static PoolResourceManager instance = null;

        public static PoolResourceManager Instance
        {
            get
            {
                if (instance == null)//不存在实例就创建一个
                {
                    GameObject go = new GameObject("ResourceManager",
typeof(PoolResourceManager));
                    go.transform.localPosition = new Vector3(0, 0, 0);
                    instance = go.GetComponent<PoolResourceManager>();

                    if (Application.isPlaying)
                    {
                        DontDestroyOnLoad(instance.gameObject);
```

哔哩哔哩次元游戏人生

```
                }
                else
                {
                    Debug.LogWarning("[ResourceManager] 您最好在编辑器模式下忽略 ResourceManager");
                }
            }

            return instance;
        }
    }
    #region 初始化对象池
    public void InitPool(string poolName, int size, PoolInflationType type = PoolInflationType.DOUBLE,bool useStack = true)
    {
        if (poolDict.ContainsKey(poolName))//数据字典中是否已有关键字
        {
            return;
        }
        else
        {
            GameObject Prefab = Resources.Load<GameObject>(poolName);//在 Resources 文件夹中获取
            if (Prefab == null)
            {
                Debug.LogError("[ResourceManager] Invalide prefab name for pooling :" + poolName);
                return;
            }
            poolDict[poolName] = new Pool(poolName, Prefab, gameObject, size, type, useStack);
        }
    }

    /// <summary>
    /// 初始化对象池
    /// </summary>
    /// <param name="cell">需要初始化的对象</param>
    /// <param name="size">对象数量</param>
    /// <param name="type">膨胀类型</param>
    public void InitPool(GameObject cell, int size, PoolInflationType type = PoolInflationType.DOUBLE, bool useStack = true)
    {
        if (poolDict.ContainsKey(cell.name))
```

哔哩哔哩次元游戏人生

```
                {
                    return;
                }
                else
                {
                    poolDict[cell.name] = new Pool(cell.name, cell, gameObject, size, type,
useStack);
                }
            }
            #endregion

            /// <summary>
            ///  获取池中的可用对象,如果池中没有可用的对象则为 null。
            /// </summary>
            /// <param name="poolName">对象池名称</param>
            /// <param name="autoActive">是否自动激活</param>
            /// <param name="autoCreate">自动创建对象数量</param>
            public GameObject GetObjectFromPool(string poolName, bool autoActive = true,
int autoCreate = 0, bool useStack = true)
            {
                GameObject result = null;

                if (!poolDict.ContainsKey(poolName) && autoCreate > 0)
                {
                    InitPool(poolName, autoCreate, PoolInflationType.INCREASE);
                }

                if (poolDict.ContainsKey(poolName))
                {
                    Pool pool = poolDict[poolName];
                    result = pool.NextAvailableObject(autoActive, useStack);
    #if UNITY_EDITOR
                    if (result == null)//对象池中找不到可用对象
                    {
                        Debug.LogWarning("[ResourceManager]:No object available in " +
poolName);
                    }
    #endif
                }
    #if UNITY_EDITOR
                else
                {
                    Debug.LogError("[ResourceManager]:Invalid pool name specified: " +
poolName);
```

哔哩哔哩次元游戏人生

```
            }
#endif
            return result;
        }

        /// <summary>
        ///  将 GameObject 返回对象池
        /// </summary>
        /// <param name="go"></param>
        public void ReturnObjectToPool(GameObject go, bool useStack = true)
        {
            PoolObject po = go.GetComponent<PoolObject>();
            if (po == null)
            {
#if UNITY_EDITOR
                Debug.LogWarning("指定的对象不是对象池实例: " + go.name);
#endif
            }
            else
            {
                Pool pool = null;
                //out 关键字是只出不进
                if (poolDict.TryGetValue(po.poolName, out pool))//TryGetValue 获取与
指定键关联的值
                {
                    pool.ReturnObjectToPool(po, useStack);
                }
#if UNITY_EDITOR
                else
                {
                    Debug.LogWarning("没有名称可用的对象池: " + po.poolName);
                }
#endif
            }
        }

        /// <summary>
        ///  将对象返回对象池
        /// </summary>
        /// <param name="t"></param>
        public void ReturnTransformToPool(Transform t, bool useStack = true)
        {
            if (t == null)
            {
```

哔哩哔哩次元游戏人生

```
    #if UNITY_EDITOR
                    Debug.LogError("[ResourceManager] try to return a null transform to
pool!");
    #endif
                    return;
                }
                ReturnObjectToPool(t.gameObject, useStack);
            }


            //判断该对象是否有对象池
            public bool HasPool(GameObject cell)
            {
                return poolDict.ContainsKey(cell.name);
            }
        }
    }
```

八、 Unity 脚部 IK

1. 构建测试场景

首先将所需的模型导入到 Unity 中，然后构建一些用于脚部 IK 的测试物体。

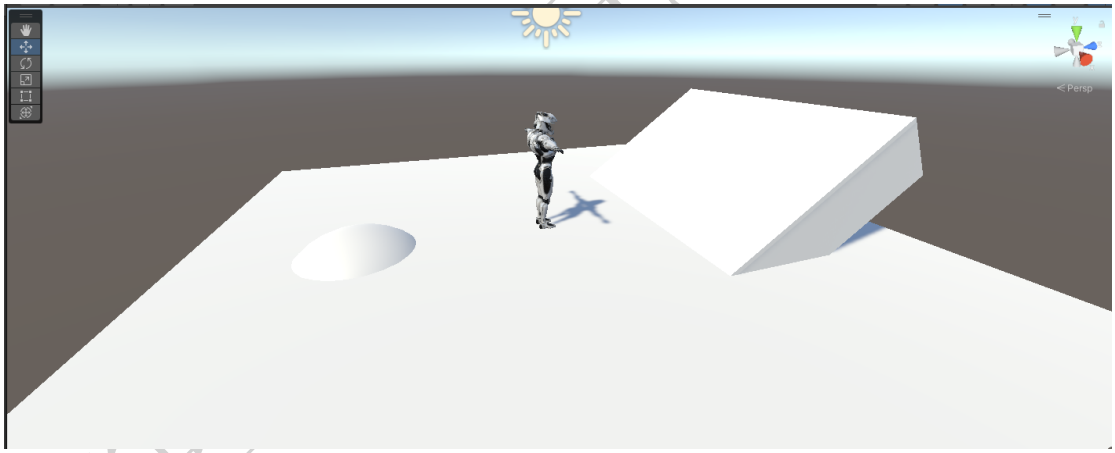


图23. 构建测试场景

模型需要有 Rigidbody 组件和 Collider 组件，否则，在查看后续的效果时，会非常不方便。

2. 创建 Animator 并设置所需参数

构建完成后就可以创建一个 Animator，我们会用 Animator 去控制动画的播放。将所需的动画拖拽到 Animator 中，我们拖拽进去的第一个动画片段，会被 Animator 设置为默认开始动画。

要修改默认的开始动画，只需要将鼠标移动到 Entry 上，然后点击鼠标右键，找到 Set StateMachine Default State 就可以修改默认的开始动画了。

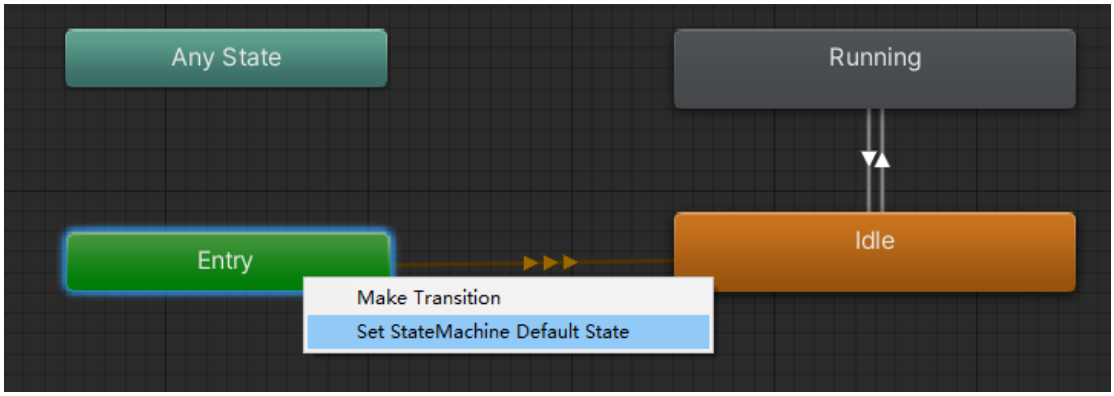


图24.　　设置状态机的默认状态

要在两个动画片段之间切换，我们则需要在两个动画之间建立一条有向的过渡线。

我们在角色身上添加一个 Animator 组件，并将我们创建的 Animator Controller 赋予它。

过渡线是可以设置条件的，默认情况下是没有任何条件的，一旦设置条件，只有当条件满足时才能从一个动画切换到另一个动画。

为了让动画依据设置的参数去切换，我们需要在 Animator 界面中找到 Paramaters，在里面选择所需的参数类型。

因为这里只是一个简单的测试，就只创建一个 Bool 类型的参数即可。

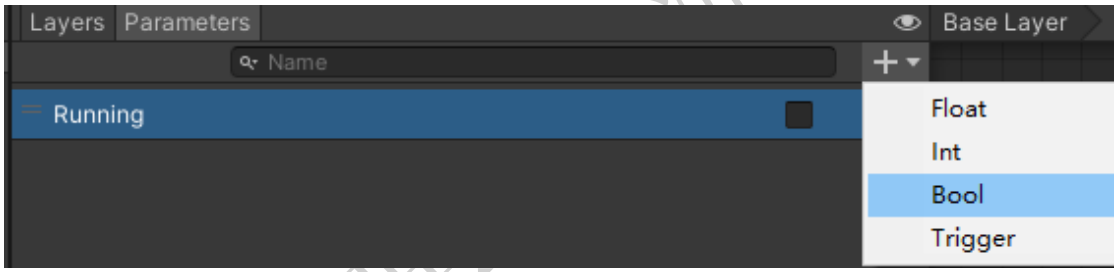


图25.　　创建动画参数

有了动画参数以后，我们就可以找到过渡线，为其添加动画切换的条件。因为过渡线在默认情况下，是有退出时间的，只有当前动画播放到对应位置时，才能进行动画切换。有了切换条件后，我们就可以关闭它，让其在任何时候都可以进行动画切换，而不需要等待片段播放到特定时间。

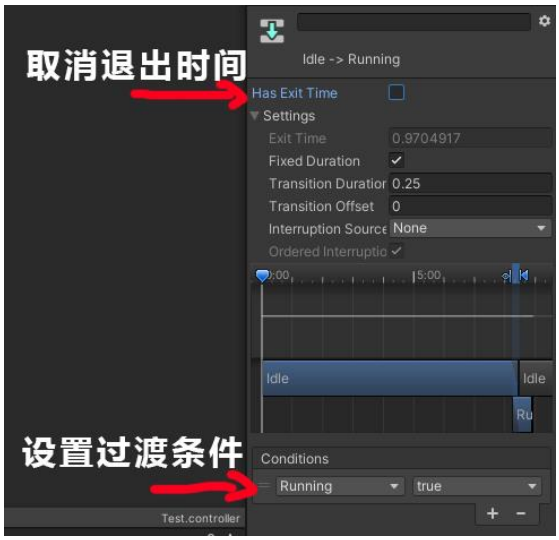


图26.　　设置过渡条件

3. 创建脚部 IK

1） 设置参数

现在已经可以切换动画了，我们就可以开始创建脚步 IK 的脚本了。

我们在脚本中创建一系列的参数，这些参数主要是获取 IK 的位置、实时脚步位置以及脚所对应的旋转值。要实现 IK，我们需要对角色所处位置进行一个检测，这需要使用到射线检测，为此需要一系列与射线检测有关的参数。

```
private Animator theAnimator;
private Vector3 leftFootIk, rightFootIk;//射线检测需要的 IK 位置
private Vector3 leftFootPosition, rightFootPosition;//IK 位置赋值
private Quaternion leftFootRotation, rightFootRotation;//IK 旋转赋值

[SerializeField] private LayerMask iKLayer;//射线可以检测到的层
[SerializeField] [Range(0, 0.2f)] private float rayHitOffset;//射线检测位置与 IK 位置的偏移
[SerializeField] private float rayCastDistance;//射线检测距离

private RaycastHit hitInfo;
[SerializeField] private bool enableIK = true;//是否启用 IK
[SerializeField] private float iKSphereRadius = 0.05f;
[SerializeField] private float positionSphereRadius = 0.05f;
```

2） 获取组件以及为参数赋值

在Awake()函数中获取组件，并为部分参数赋初值。

```
theAnimator = this.gameObject.GetComponent<Animator>();
leftFootIk = theAnimator.GetIKPosition(AvatarIKGoal.LeftFoot);//获取 IK 位置
rightFootIk = theAnimator.GetIKPosition(AvatarIKGoal.RightFoot);
```

3） 设置脚部位置和旋转

要设置脚部的 IK 位置和旋转值，我们需要将方法写在 OnAnimatorIK 函数中。OnAnimatorIK 函数，会在 Animator 组件更新其内部 IK 系统之前调用。权重就代表 IK 对于动画的影响程度，值为 1 时 IK 将完全掌控动画的相关部分，值为 0 时就是原动画展示出的效果。

```
private void OnAnimatorIK(int layerIndex)
    {
        leftFootIk = theAnimator.GetIKPosition(AvatarIKGoal.LeftFoot);
        rightFootIk = theAnimator.GetIKPosition(AvatarIKGoal.RightFoot);

        if (!enableIK)
        {
            return;
        }

        #region 设置 IK 权重
        theAnimator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, 1);
```

```
        theAnimator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, 1);

        theAnimator.SetIKPositionWeight(AvatarIKGoal.RightFoot, 1);
        theAnimator.SetIKRotationWeight(AvatarIKGoal.RightFoot, 1);
        #endregion
        #region 设置 IK 位置和旋转值
        theAnimator.SetIKPosition(AvatarIKGoal.LeftFoot, leftFootPosition);
        theAnimator.SetIKRotation(AvatarIKGoal.LeftFoot, leftFootRotation);

        theAnimator.SetIKPosition(AvatarIKGoal.RightFoot, rightFootPosition);
        theAnimator.SetIKRotation(AvatarIKGoal.RightFoot, rightFootRotation);
        #endregion
    }
```

在这里我们先将所有的权重设置为 1，后面会再对其进行更改。

4） 获得位置和旋转值

在上面的 OnAnimatorIK 函数中，我们已经设置了动画的 IK 位置和旋转值。目前，我们的这些参数值都还是默认值，我们需要在游戏运行过程中去获取。

我们知道，角色脚的位置和角度，与其所站的平台有关。在平地上，角色的脚就是和地面平行的；在坡面上，脚就应该和坡面平行。要获得这些数据，我们就需要使用射线检测去实现了，因为射线检测与 Unity 的物理系统相关，因此这部分内容需要写在 FixedUpdate 函数中。

为了方便查看，使用 Debug.DrawLine 去绘制线段，这也可以写在 OnDrawGizmos 函数中，只不过需要进行一些微小的改动。

```
Debug.DrawLine(leftFootIk + (Vector3.up * 0.5f), leftFootIk + Vector3.down * rayCastDistance, Color.blue, Time.deltaTime);
Debug.DrawLine(rightFootIk + (Vector3.up * 0.5f), rightFootIk + Vector3.down * rayCastDistance, Color.blue, Time.deltaTime);
```

当射线检测到指定层物体时，我们就返回碰撞点的各种信息，脚步的旋转值就与返回的法线信息有关，即下面的 hit.normal。

```
        if (Physics.Raycast(leftFootIk + (Vector3.up * 0.5f), Vector3.down, out RaycastHit hit, rayCastDistance + 1, iKLayer))
        {
            Debug.DrawRay(hit.point, hit.normal, Color.red, Time.deltaTime);

            leftFootPosition = hit.point + Vector3.up * rayHitOffset;// 如果让脚的位置等于碰撞点，那么可能会出现穿模的情况。
            leftFootRotation = Quaternion.FromToRotation(Vector3.up, hit.normal) * transform.rotation;
        }

        if (Physics.Raycast(rightFootIk + (Vector3.up * 0.5f), Vector3.down, out RaycastHit hit_01, rayCastDistance + 1, iKLayer))
        {
            Debug.DrawRay(hit_01.point, hit_01.normal, Color.red, Time.deltaTime);
```

哔哩哔哩次元游戏人生

```
                rightFootPosition = hit_01.point + Vector3.up * rayHitOffset;
                rightFootRotation = Quaternion.FromToRotation(Vector3.up, hit_01.normal) *
transform.rotation;
            }
```

上面的代码中，我们将脚的位置进行了一定的抬升，这是因为，如果让脚的位置等于碰撞点，那么可能会出现穿模的情况。

5） 绘制 IK 位置和脚的位置

为了方便大家理解，我们在 OnDrawGizmos 函数中将 IK 位置和脚的位置绘制出来，IK 位置用绿色表示，脚的位置用青色表示。

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(leftFootIk, iKSphereRadius);
        Gizmos.DrawSphere(rightFootIk, iKSphereRadius);
        Gizmos.color = Color.cyan;
        Gizmos.DrawSphere(leftFootPosition, positionSphereRadius);
        Gizmos.DrawSphere(rightFootPosition, positionSphereRadius);
    }
```

当我们完成上述内容后，我们还需要回到 Unity 中，为场景中的测试对象设置一个 Layer。

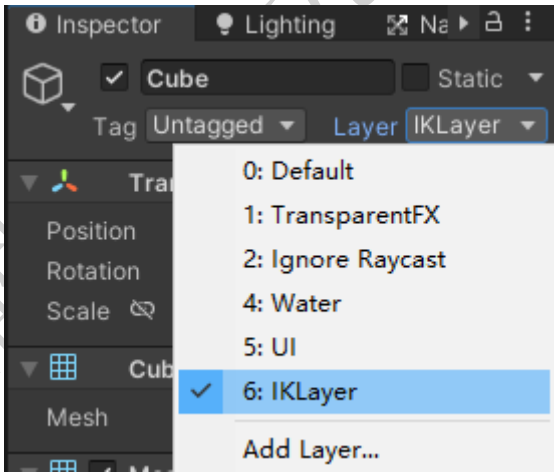


图27. 设置对象的 Layer

当射线触碰到对象时，就会判断是否和设定的检测层一致，如果一致就返回碰撞点的各种信息。

4. 设置动画 IK 权重

当我们完成上面的部分，就可以将脚本挂载到角色身上，以测试其实际效果。因为现在的权重是 1，当我们执行待机动画时，几乎没有问题，毕竟此时角色的脚是一直处于地面的。

一旦我们将动画切换为跑动，在原本的动画中，角色的脚是需要抬起来的。因为我们设置的 IK 权重为 1，导致脚的动画将被 IK 完全影响。

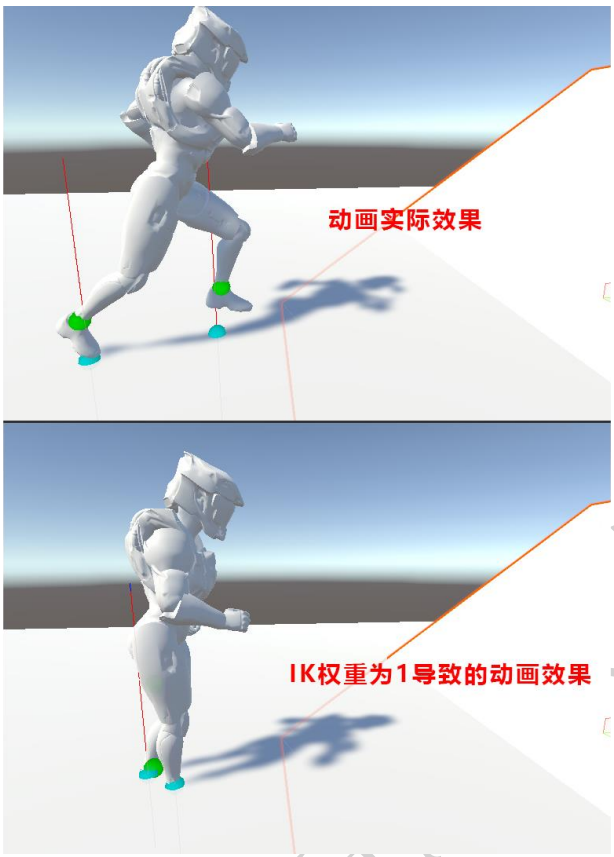


图28.　　权重对于动画的影响

为了能够解决这种不良的影响，我们可以创建一条动画曲线，为其添加一系列关键帧，用这些关键帧去控制 IK 的值。

在 Animator 中我们可以创建一个，和动画曲线名称一致的参数，它会在动画播放时自动获取对应曲线的值。然后，在代码中，我们可以获取这个参数的值，来实时修改权重。

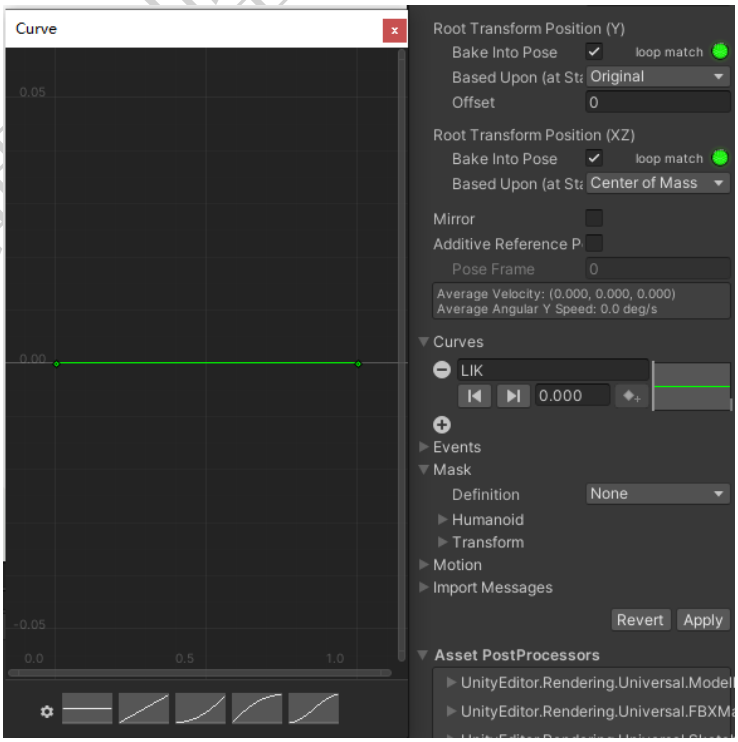


图29.　　设置动画曲线

当需要脚完全和地面贴合时，其 IK 的权重就应该为 1，当脚需要离开地面时，其 IK 的权重就应该为 0。

```
theAnimator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, theAnimator.GetFloat("LIK"));
theAnimator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, theAnimator.GetFloat("LIK"));

theAnimator.SetIKPositionWeight(AvatarIKGoal.RightFoot, theAnimator.GetFloat("RIK"));
theAnimator.SetIKRotationWeight(AvatarIKGoal.RightFoot, theAnimator.GetFloat("RIK"));
```

修改完成后再次运行，现在就不会再出现 IK 对动画影响过大的情况了。

5. 可能出现的问题

没有 IK 效果出现，请检查是否勾选了对应 Layer 的 IK Pass 选项。只有勾选了 IK Pass 的 laye 才会调用 OnAnimatorIK 方法，每个勾选了 IK Pass 的 layer 调用一次。

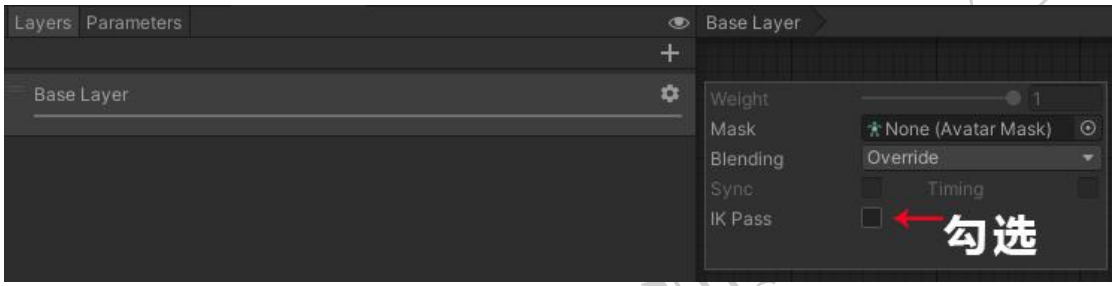


图30.　　检查 IK Pass 是否勾选

九、 射线检测应用

1. 添加 Layer 层

在使用射线检测之前，我们需要先为被检测的游戏对象设置一个 Layer。Unity 本身为我们预先设置了一系列的 Layer，但实际开发过程中我们是一定会添加许多自己的 Layer 的。

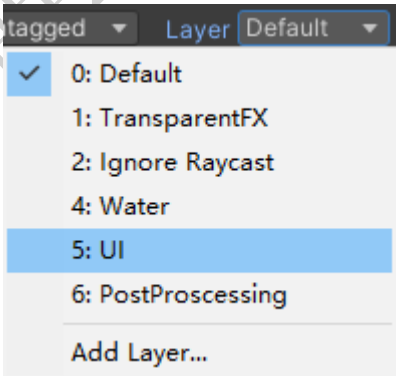


图31.　　Unity 默认的 Layer

首先我们需要明白的一点是，Layer 最多只能有 32 层，且开始的下标是 0，结束的下标是 31。这是为什么呢？

答案其实很简单，Unity 中是用 int32 来表示 32 个 Layer 层。int32 表示的二进制数一共只有 32 位。因此，我们所设定的 Layer 层，最多也就只能是 32 层。

我们可以为其新增三个 Layer 层用作后面的测试。我这里将它们命名为，Enemy、Friend、Stranger。

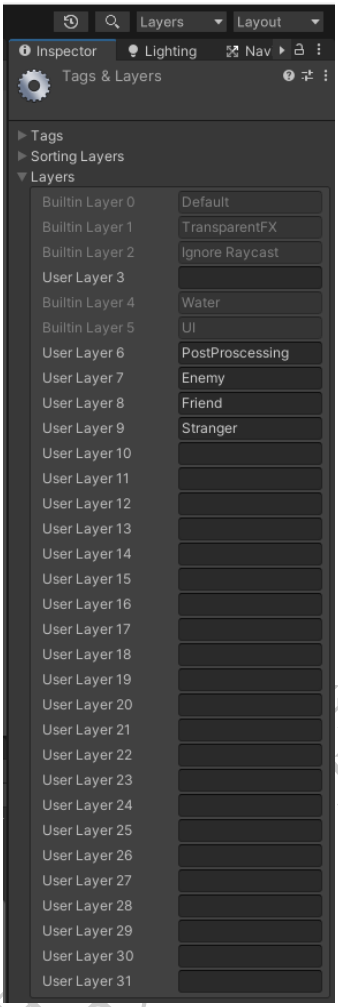


图32. 新增 Layer 层

2. Layer 和 LayerMask

现在我们已经新增了一些 Layer，我们就可以创建一个 C#脚本，并将其挂载到对象上去进行测试。

首先，我们直接在脚本中打印一下脚本对象的 Layer，查看一下它在代码中实际是何值。

PS：

```
private void Start()
    {
        Debug.Log(this.gameObject.layer);
    }
```

哔哩哔哩 游戏人生

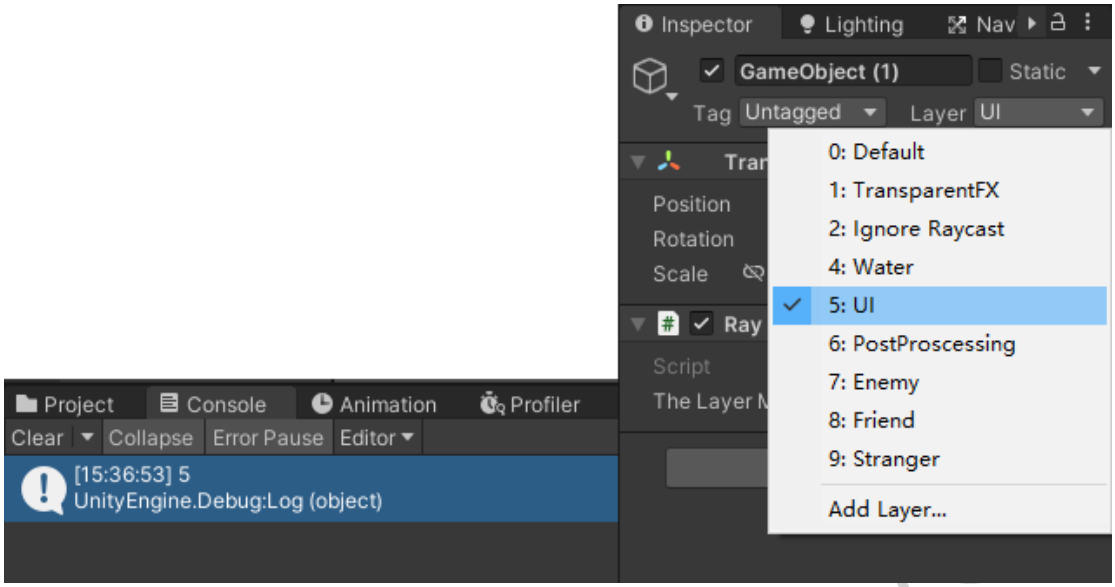


图33.　　打印出的当前脚本对象的 Layer

我们可以看到，打印的结果就是我们所设定的层数。再在代码中公开一个 LayerMask 变量，将这个 LayerMask 也打印出来，看看它的值和 Layer 有什么区别。

PS：

```
public LayerMask theLayerMask;
private void Start()
{
    Debug.Log("this is layer value "+this.gameObject.layer);
    Debug.Log("this is layerMask value " + theLayerMask.value);
}
```

我们先保持 LayerMask 层和对象的 Layer 层一致，也就是 LayerMask 只勾选一个层，然后我们来查看一下它们的值有什么不同。

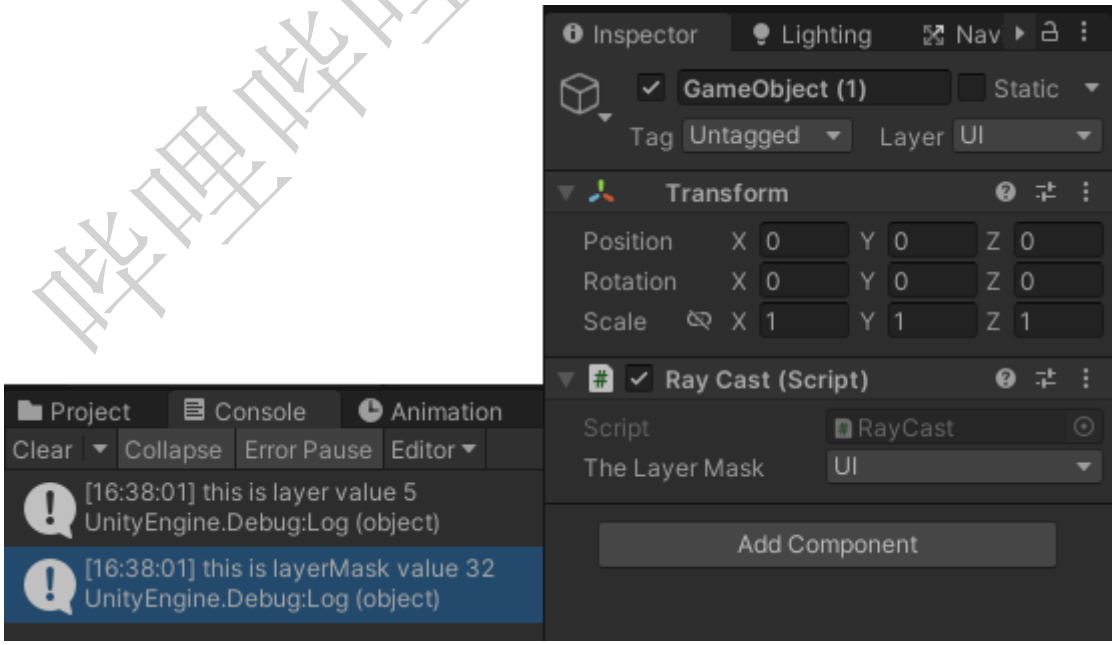


图34.　　保持 LayerMask 层和 Layer 层一致

从上图我们可以得知，我们的 Layer 输出的值是 5，而 LayerMask 输出的值则是 32。

哔哩哔哩次元游戏

如果，我们将 LayerMask 勾选多个层级，那么它的输出值又会有什么变化呢？

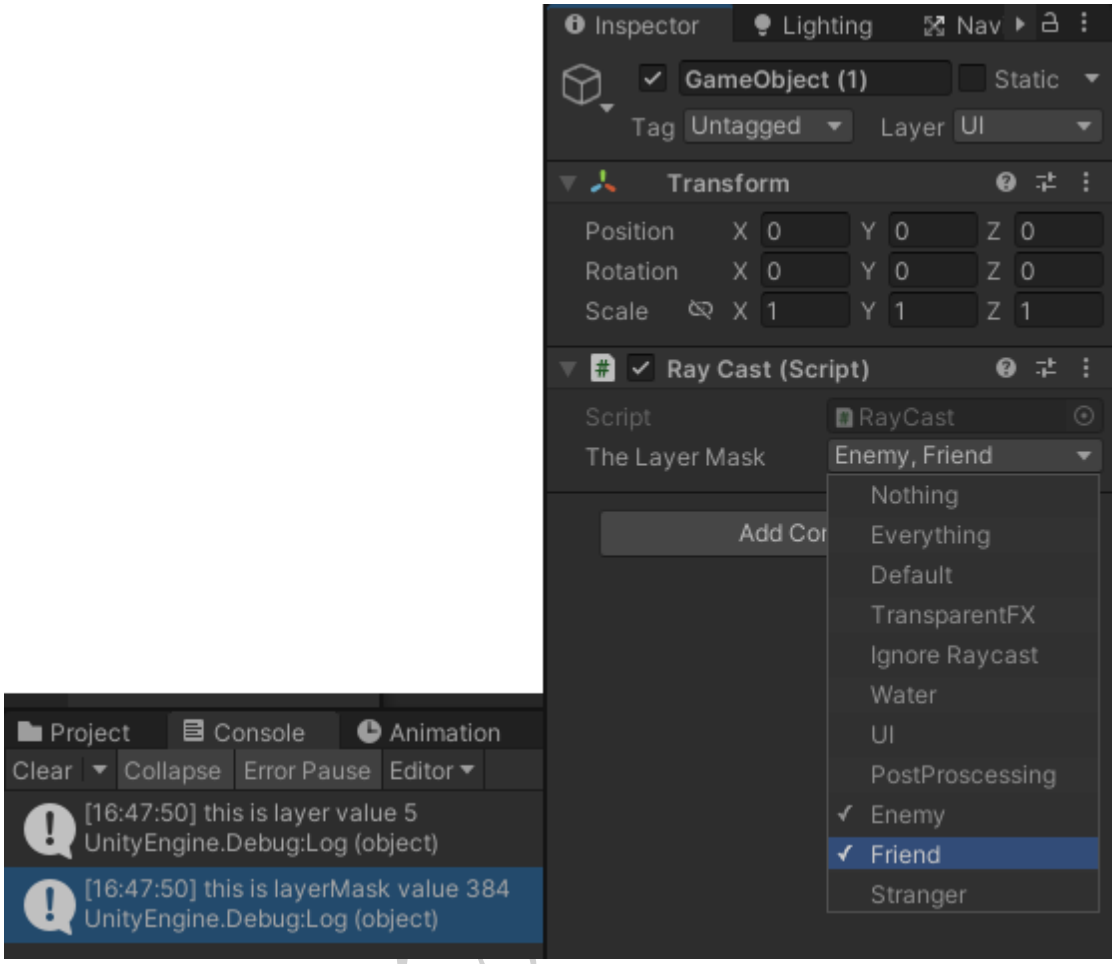


图35. LayerMask 勾选多个层级

如上图所示，我们勾选了第七层 Enemy 和第八层 Friend，我们的输出结果就是 384。那么这个值到底是怎么得到的呢？

前面已经说过了，Unity 中的 Layer 最多只能有 32 层，也即 32 位二进制。而我们在 LayerMask 中所勾选的层级，就表示其对应的二进制位为 1，而未勾选的部分则对应 0。之前我们只勾选了第五层 UI，它的 LayerMask 其实对应的就是 0010 0000 这个二进制数。我们可以对数值 1 左移 5 位来表示这个二进制数，即 1 << 5 = 0010 0000 = 32。

那么，LayerMask 勾选第七层和第八层时，所对应的二进制数就是 0001 1000 0000。我们可以将左移运算符和按位或运算符结合，来表示这个二进制数，即 1 << 7 | 1 << 8 = 0001 1000 0000 = 384。

PS：

```
public LayerMask theLayerMask;
    private void Start()
    {
        Debug.Log("this is layer value "+this.gameObject.layer);
        Debug.Log("this is layerMask value " + theLayerMask.value);
        Debug.Log("this is 5 Layer value " + (1 << 5));
        Debug.Log("this is 7 and 8 layerMask value " + (1 << 7 | 1 << 8));
    }
```

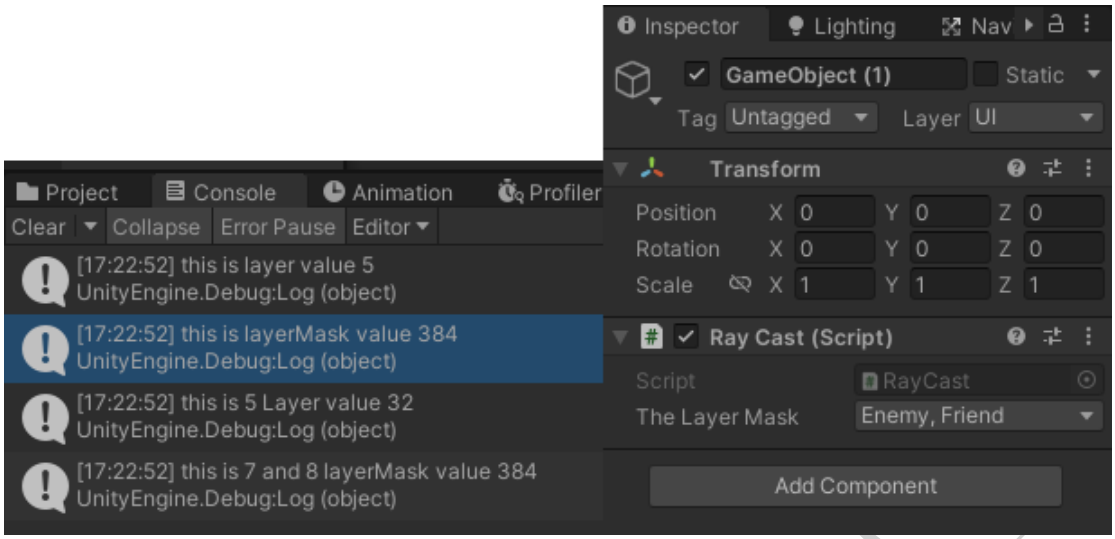


图36. 打印出计算机计算的二进制结果

3. 使用单体射线检测

现在我们清楚了 Layer 和 LayerMask 的值，以及如何在计算机中表示它们，下面就可以使用射线检测 RayCast 了。

我们首先声明一个 LayeMask 类型的私有变量，然后使用 SerializeField 字段去强制序列化，使该私有变量在 Inspector 面板可见。要对 3D 对象使用射线检测则是使用 Physics 中的 RayCast；对 2D 对象使用射线检测则是使用 Physics2D 中的 RayCast。

最简单的射线检测只有两个参数，一个起始点坐标和射线的方向。这里需要特别说明一下，transform.forward、transform.right 以及 transform.up。

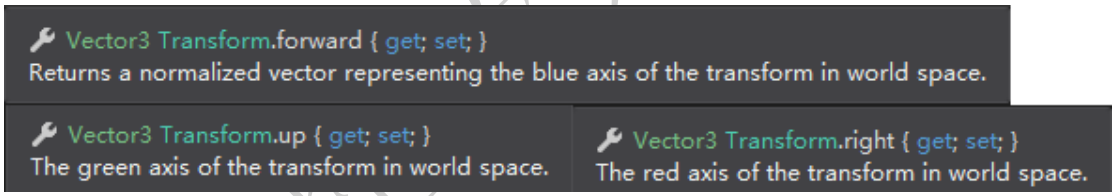


图37. transform.up、forward、right

从上图可以知道，transform.forward 对应的是 Unity 中的蓝色轴，也就是 Unity 中的 Z 轴；transform. right 对应的是 Unity 中的红色轴，也就是 Unity 中的 X 轴；transform.up 对应的是 Unity 中的绿色轴，也就是 Unity 中的 Y 轴。因为我这里的 2D 项目不会使用到 Z 轴，因此射线的方向就只能是 right 或者 up。

PS：

```
[SerializeField] //强制序列化，使 Private 在 Inspector 面板可见
    private LayerMask checkLayer;
    private void Update()
    {
        //3D 的射线检测
        Physics.Raycast(this.transform.position, this.transform. right);
        //2D 的射线检测
        Physics2D.Raycast(this.transform.position, this.transform. right);
    }
```

但，我们的射线检测的参数并不是只有起始坐标和方向这两个参数。我们还可以设置射线的最大距离，射线可以检测的 Layer 层以及射线反馈的信息等。为了方便调整射线检

测，我们需要将这些变量声明出来。

声明 Float 类型的变量去控制射线可以检测的最大距离，声明 RaycastHit2D 类型的参数去接收 2D 射线检测返回的信息，声明 RaycastHit 类型的参数去接收 3D 射线检测返回的信息。在 3D 射线检测中作为参数传入时，需在 RaycastHit 变量前加上 Out 关键字，2D 的射线检测返回结果就是 RaycastHit2D 类型的，直接赋值即可。

PS：

```
[SerializeField] //强制序列化，使 Private 在 Inspe 面板可见
    private LayerMask checkLayer;
    [SerializeField]
    private float maxDistance = 10;
    private RaycastHit hitInfo;
    private RaycastHit2D hitInfo2D;
    private void Update()
    {
        //3D 的射线检测
        Ray theRay = new Ray(this.transform.position, this.transform.right);
        Physics.Raycast(theRay, out hitInfo, maxDistance, checkLayer);
        //2D 的射线检测
        hitInfo2D = Physics2D.Raycast(this.transform.position, this.transform.right,
maxDistance, checkLayer);
    }
```

4. 旋转游戏对象

前面我们已经在 Update 函数中执行了射线检测，接下来就是编写一个方法去旋转我们的脚本对象。因为这里只需旋转一个轴即可，就直接使用欧拉角；如果需要避免万向节死锁，则使用四元数来旋转脚本对象。

为了便于控制旋转的速度，我们需要一个 Float 类型的参数去控制，这里将额外使用一个 Range 字段，去控制该变量的数值范围。

PS：

```
[SerializeField, Range(1, 10)] private float rotateSpeed = 5;
private void Update()
    {
        Vector3 temporary = new Vector3(0, 0, this.transform.eulerAngles.z -
Time.deltaTime * 10 * rotateSpeed);
        this.transform.eulerAngles = temporary;
        //2D 的射线检测
        hitInfo2D = Physics2D.Raycast(this.transform.position, this.transform.right,
maxDistance, checkLayer);
    }
```

5. 可视化射线

为了便于观察射线检测，可以使用 OnDrawGizmos 或 OnDrawGizmosSelected 函数来辅助。OnDrawGizmos 是始终绘制提示信息，而 OnDrawGizmosSelected 是只在脚本对象被选中时才开始绘制。

哔哩哔哩次元游戏人生

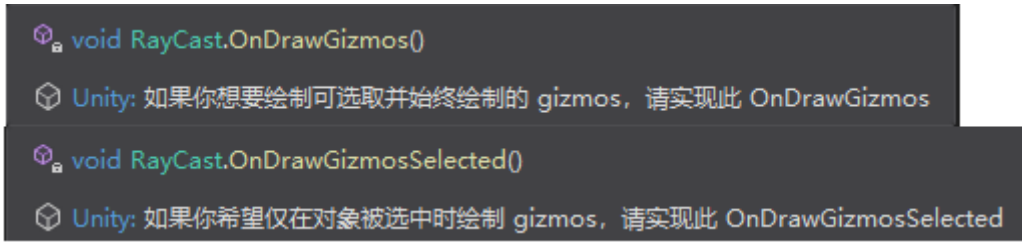


图38.　　Gizmos 方法区别

为了便于区分，我们可以为绘制出的辅助射线设置一个我们想要的颜色，然后使用 Gizmos.DrawLine 去绘制一条射线。Gizmos.DrawLine 的第一个参数是起始位置，第二个参数则是射线的结束位置。

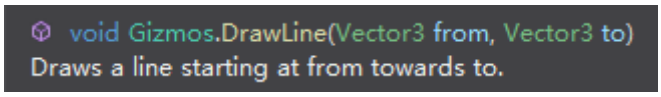


图39.　　Gizms.DrawLine 方法

PS：

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(this.transform.position, this.transform.right);
    }
```

我们回到 Unity 中，打开我们的 Gizmos。（根据版本的不同，Gizoms 的图标以及开启位置会有所不同，请自行操作）

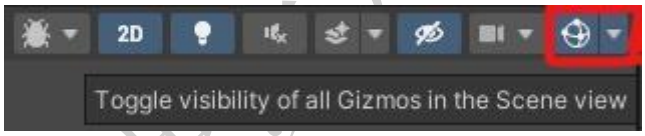


图40.　　打开 Gizmos

一旦打开了 Gizmos，我们就可以在 Unity 中看到一系列的辅助信息了。我们可以看到绘制出的蓝色射线似乎不正确，射线的终点位置并没有跟随着我们进行移动。这是什么原因？

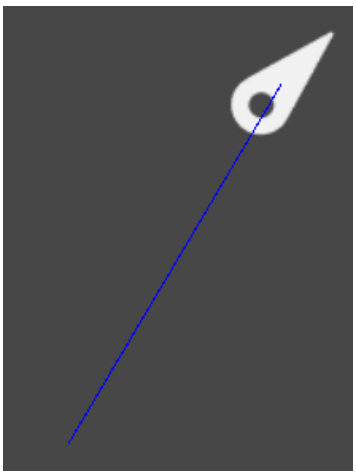


图41.　　绘制的射线位置似乎不太准确

其实，我们只需要仔细查看一下 transform.right 的解释就清楚了。transform.right、forward、up 对应的是世界空间下的，并不是局部空间的，因此我们的目标点将始终指向世

哔哩哔哩：无游戏人生

界空间中的那一点。

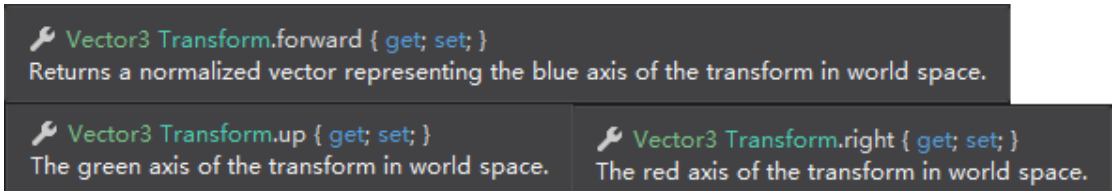


图42. transform.up、forward、right

要让辅助射线跟随对象移动，我们可以将对象的当前位置和 transform.right 相加。

PS：

```
private void OnDrawGizmos()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(this.transform.position, this.transform.position +
this.transform.right);
    }
```

在前面我们已经说了，游戏对象的 Layer 输出的它的层数，而 LayerMask 输出的是一个二进制数，且 LayerMask 是可以包含多个 Layer 层。那么，我们射线检测到的对象，就可以使用左移和按位与去判断 Layer 层，从而对不同 Layer 对象执行不同的行为。

PS：

```
private void OnDrawGizmos()
    {
        Color temporary = new Color(0,0,0);
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Friend")) != 0)
        {
            temporary = Color.green;
        }
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Enemy")) != 0)
        {
            temporary = Color.red;
        }
        if (hitInfo2D.collider != null && (1 << hitInfo2D.collider.gameObject.layer &
LayerMask.GetMask("Stranger")) != 0)
        {
            temporary = Color.blue;
        }
        Gizmos.color = temporary;
        Gizmos.DrawLine(this.transform.position, this.transform.position +
this.transform.right * (hitInfo2D.collider == null ? maxDistance : hitInfo2D.distance));
    }
```

6. 添加用于检测的对象

现在我们已经有了射线检测，也完善了不同 Layer 层所对应的行为，接下来我们就可以在 Unity 中添加用于检测的对象了。因为射线检测与 Unity 的物理系统相关，因此我们

哔哩哔哩次元游戏人生

必须为被检测的对象添加上 Collider 组件，否则射线检测无法检测到对象。

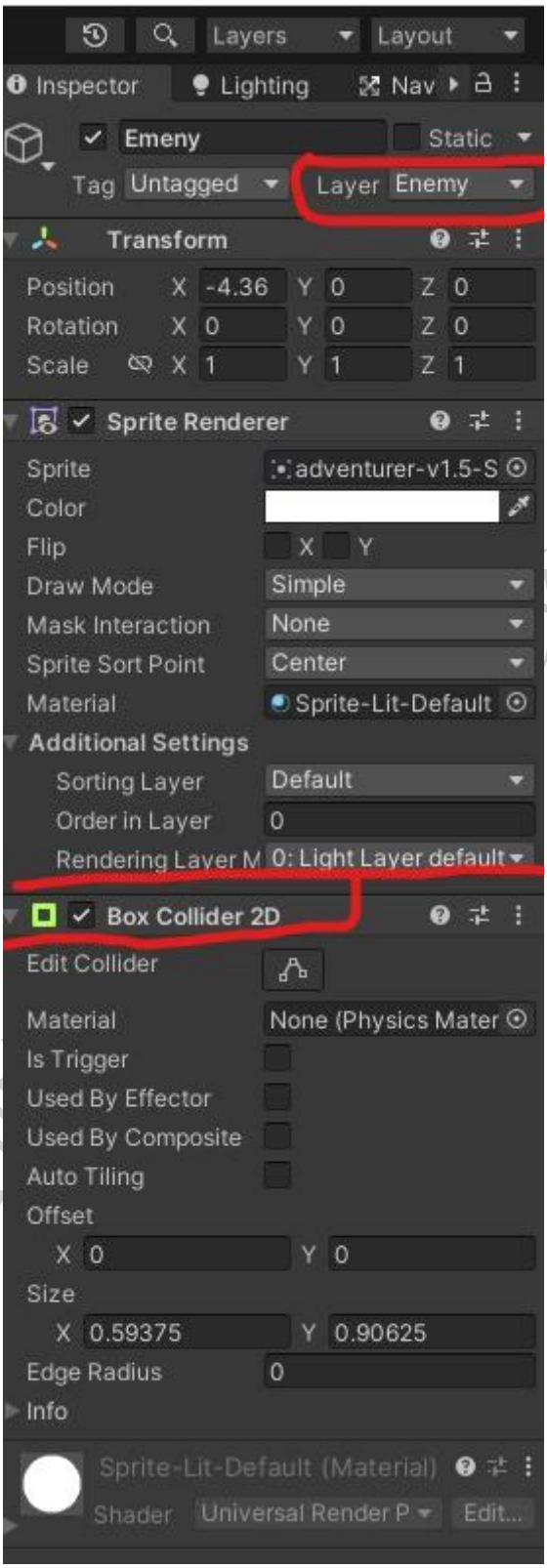


图43.　　射线检测的对象，需要包含 Collider 组件

7. 使用多对象的射线检测

前面讲了单对象的射线检测，它只会返回检测到的第一个对象的信息，如果在射线的检测方向上还有其他可被检测的对象，这些对象将无法被检测到。为此，我们就可以使用

多对象的射线检测，即 Physics2D.RaycastAll。

RaycastAll 和 RacyCast 功能上没大的区别，只是 RaycastAll 返回的是，射线方向上所有可被检测对象的信息；而 RacyCast 返回的是，射线方向上第一个被检测对象的信息，也就是说只要检测到一个对象，就不再继续向后检测了。

十、 射线检测的性能消耗

名称	速度
Raycast	1ms
RaycastNonAlloc	2 ms
RaycastAllTest	2 ms
Linecast	1 ms
BoxCastTest	17 ms
BoxCastNonAllocTest	18 ms
BoxCastAllTest	20 ms
CheckBoxTest	1 ms
OverlapBoxTest	62 ms
OverlapBoxNonAllocTest	24 ms
CapsuleCastTest	162 ms
CapsuleCastNonAllocTest	227 ms
CapsuleCastAllTest	1 ms
CheckCapsuleTest	84 ms
OverlapCapsuleTest	30 ms
OverlapCapsuleNonAllocTest	96 ms
SphereCastTest	171 ms
SphereCastNonAllocTest	226 ms
SphereCastAllTest	22o6 ms
CheckSphereTest	1 ms
OverlapSphereTest	33 ms
OverlapSphereNonAllocTest	16 ms

Ray 和 Line 的开销非常小，撇开这个可以发现，性能消耗的顺序是这样的。

同方法下不同模型 GC 开销：Box < Sphere < Capsule

同模型下不同方法 GC 开销：CheckXXX < OverlapXXX < XXXCast

哔哩哔哩次元游戏人生

技巧

一、 修改 Unity 缓存位置

1. 修改 Unity GICache

打开 Unity 的编辑器，找到 Editor/Preferences/GICache，修改缓存位置。

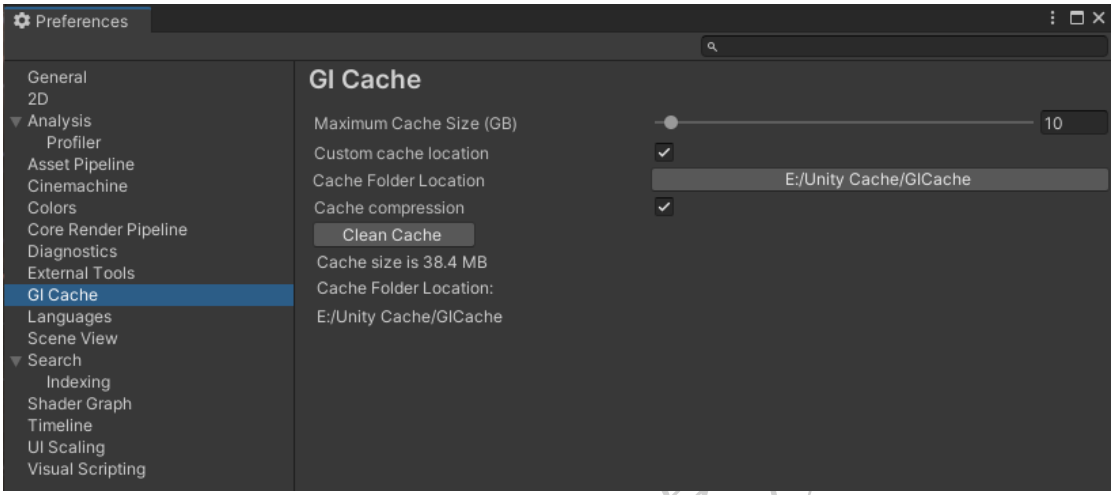


图44. 图 1.1 示意图

找到 Cache Folder Location 后根据自己的需要，设置 GICache 的缓存位置即可。GICache 的默认缓存位置是 C 盘的 C:\Users\你的用户名\AppData\Local\Unity，AppData 文件是隐藏文件，需要手动设置显示隐藏文件。

图45. 图 1.2 显示隐藏文件

2. 修改 Unity 的 Cache 缓存位置

1） 修改 Unity Cache

Unity 手册-全局缓存：https://docs.unity3d.com/cn/2021.2/Manual/upm-cache.html

Unity Package Manager 在下载包内容和元数据时，会将它们存储在全局缓存中。这样可以更有效地重复使用和共享包，甚至在脱机时也可以安装和更新存储的包。

在默认情况下，Unity 会将全局缓存存储在根目录中，根目录取决于操作系统（在 Windows 上还取决于用户帐户类型）：

操作系统	默认根目录
Windows（用户帐户）	%LOCALAPPDATA%\Unity\cache
Windows（系统用户帐户）	%ALLUSERSPROFILE%\Unity\cache
macOS	$HOME/Library/Unity/cache
Linux	$HOME/.config/unity3d/cache

图46. 图 1.3 各操作系统默认根目录

Package Manager 使用两个不同的共享缓存，各自有不同的用途。它们存储在上述文件夹位置下的子目录中：

子文件夹	描述
npm	存储使用 npm 协议从注册表获得的数据。这包括包元数据和包 tarball。
packages	此缓存包含从注册表获取的包 tarball 的未压缩内容。

图47. 图 1.4 子文件夹描述

随着项目增加，我们的缓存会越来越大，导致我们的默认缓存位置空间不足。为了避免这种情况的出现，就需要修改 Unity 的缓存位置。

① 找到我的电脑，右键找到属性选项

图48. 图 1.5 找到我的电脑的属性选项

② 打开高级系统设置

图49. 图 1.6 找到高级系统设置

③ 打开环境变量

图50.　图 1.7 环境变量

④　自定义共享缓存位置

可以使用两个环境变量来覆盖任一全局缓存文件夹：

- **UPM_NPM_CACHE_PATH 覆盖 npm 文件夹路径。例如 (macOS)：**
  `UPM_NPM_CACHE_PATH=/dev/ssd/shared/Unity/cache/npm`
- **UPM_CACHE_PATH 覆盖 packages 文件夹路径。例如 (macOS)：**
  `UPM_CACHE_PATH=/dev/ssd/shared/Unity/cache/packages`

图51.　图 1.8 覆盖全局缓存文件夹

⑤　在系统变量中新建 2 个环境变量

变量名 1：UPM_CACHE_PATH

变量值 1：新的缓存文件夹地址

变量名 2：UPM_NPM_CACHE_PATH

变量值 2：新的缓存文件夹地址

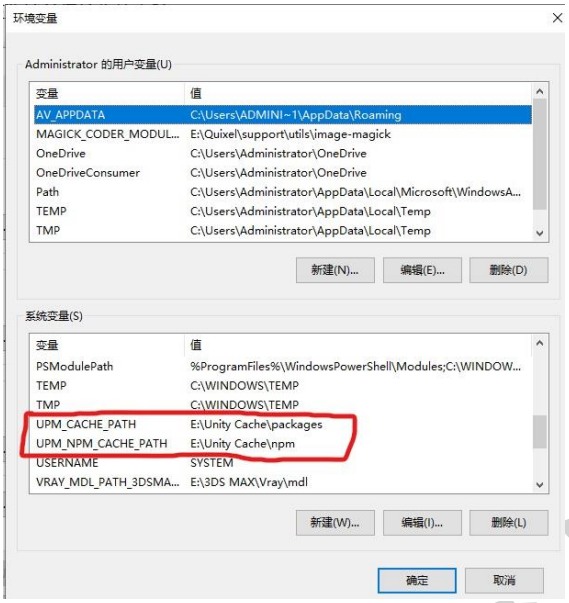


图52.　　图 1.8 新建环境变量

⑥　重启电脑。

⑦　打开 UnityHub 和 Unity 项目，查看新的缓存目录里是否缓存了东西，如果有就可以将默认缓存文件夹删除。如果没有，就重复上述操作。

3. 修改 Unity Asset Store 的默认下载路径

①　在 Unity 商城下载的文件，默认存储路径是 C:\Users\用户名\AppData\Roaming\Unity\Asset Store-5.x

AppData 文件是隐藏文件，需要手动设置以显示隐藏文件。

图53.　　图 2.1 显示隐藏文件

②　找到 Asset Store-5.x 文件夹后，将其整个剪贴到你想要存放的目录。

③　然后进 cmd，输入命令：

mklink /j "C:\Users\Administrator\AppData\Roaming\Unity\Asset Store-5.x" "F:\Unity Asset Store\Asset Store-5.x"

创建目录软链接。

二、　优化

1. 全平台优化方案

2) FixedUpdate 函数优化

在脚本中使用 Unity 的 FixedUpdate 函数时，一定要牢记尽可能不要写太多无需在物理帧重复调用的代码在方法内，因为默认情况下 Unity 的虚拟机在执行 FixedUpdate 函数时是以每秒执行 50-100 次该方法来处理场景中的每个脚本与每个对象的。FixedUpdate 函数的执行频率在 Unity 中是可以被修改的，只需要在菜单中依次选择 Edit→Project Setting→Time 命令，就可以在 Inspector 视图中显示 TimeManager 属性面板并修改默认的执行频率。

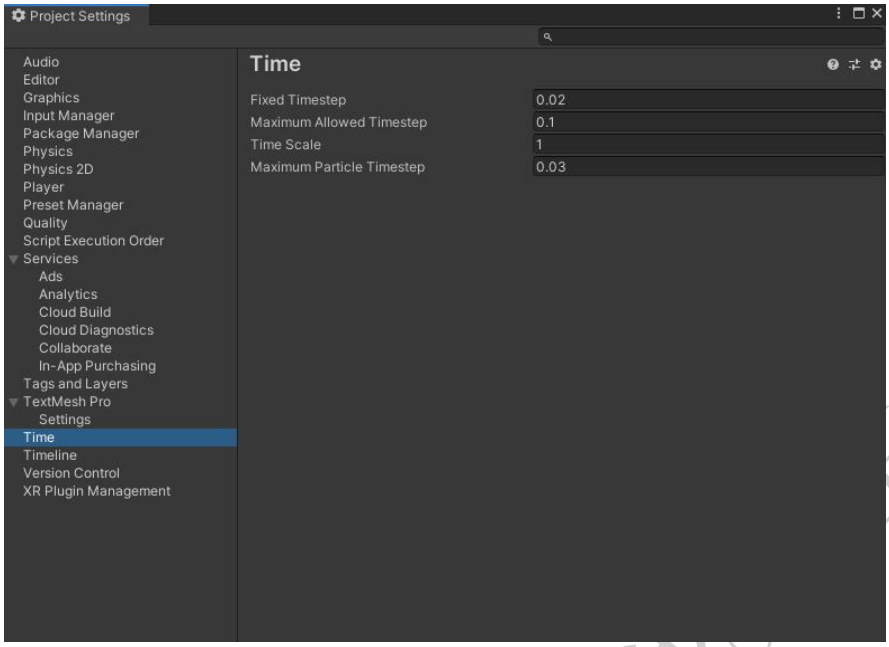


图54.　　图 3.1 TimeManager 属性面板

FixedUpdate 函数与 Update 函数的区别在于，Update 函数会在每次渲染新的一帧时执行（即每帧调用），它会受到当前渲染物体的影响，毕竟渲染的帧率并不是一成不变的，所以渲染的时间间隔也会变化，也就是说设备的性能会影响到 Update 函数的更新频率，而 FixedUpdate 函数则不会受到帧率的影响，在 TimeManager 面板可以看出它是以固定的时间间隔被调用的。使用在用法的处理上，FixedUpdate 更多适用于处理物理引擎。Update 因为受渲染物体的影响，使用更多地把 Update 用于脚本逻辑的控制上。

3） 新建脚本时删除不需要函数

当开发者在 Unity3D 中创建新的脚本时，Unity 默认会生成空的 Update 函数，但是并不是每一个脚本都需要使用到它。我们可以在引擎安装的对应位置找到默认脚本代码，引擎更目录->Editor->Data->Rescources->ScriptTemplates 下的 81-C# Script-NewBehaviourScript.cs。找到默认的脚本代码后使用记事本将其打开，删除对应的 Update 函数并将其保存后退出，这样在 Unity 创建新的脚本时就不会在默认添加 Update 函数了。

如果并未修改新建脚本中自带的默认方法，当该新建脚本并不需要用到 Update 函数时，应该将 Update 函数从脚本中删除。如果要在场景中查找对象，应该在游戏初始化时进行查找，即尽量在 Start 或 Awake 函数中执行查找，而不是在初始化完成后再在 Update 函数内执行 Find、FindObjectOfType、FindGameObjectsWithTag 这些寻找物体的函数，这一行为会极大的消耗设备的性能。

4） 优化数学计算

因为在程序世界中除法的运算比乘法要复杂，因此能够使用乘法运算时优先考虑乘法运算而不是除法运算。另外，如果可以在代码中使用整型（int）就不要优先考虑浮点型（float），尽量少用复杂的数学函数比如三角函数等，使用复杂的数学函数会增加性能的消耗。

5） 减少临时变量使用

当脚本中需要使用到变量时，应当优先考虑全局变量而不是局部变量，特别是在 Update 等实时调用的函数中。

6） 减少不常用方法使用次数

当一个程序不必要每帧都执行时，就可以考虑将其剥离出 Update 等实时调用的函数，然后在 Coroutines（协程）函数中执行函数体；当函数需要被定时重复的调用时，可以使用 InvokeRepeating 函数实现。

```
Void Start()
{
//启动 1.5 秒后每隔 1 秒执行一次 DoSomeThing 函数
InvokeRepeating("DoSomeThing",1.5f,1.0f);
}
```

2. 脚本性能优化

1） 减少 Update 内 GetComponent 的调用

当开发者在脚本中使用 GetComponent 方法或者 unity 内置的其他组件访问器（如 Find 等），应该在脚本生命周期最开始的时候就定义它，即在 Enable、Start、Awake 函数中使用。因为这些函数会带来明显的性能开销，如果在程序运行的过程中去使用会带来卡顿等现象，影响用户的体验。

2） 尽量避免动态内存的分配

为数组、表单等分配内存应尽可能的在脚本生命周期最开始的时候，在程序的运行过程中应该避免再分配新的对象，特别是在 Update 这样的实时调用函数中。在实时调用函数中分配新的内存，不仅仅会增加内存的开销，与此同时也会增加内存回收的性能开销。

3） 减少 GUILayout 布局的使用

老子曾经说过，有得必有失，有失必有得，事多无兼得者。GUILayout 虽然能让开发者极为便捷的调整间距，但是不可避免的会带来一些性能的损失。对于不需要过多变化的可以使用 GUI 手动处理布局，以减少性能上的开销，如果想要完全静止 GUILayout 布局的使用可以在脚本中设置 useGUILayout 为 false。

3. 物理性能优化

1） 减少网格碰撞体使用

在开发过程中能够使用球体(Sphere)、盒体(Box)等碰撞体来尽可能地接近网格的形状，就因少的使用网格碰撞体，因为网格碰撞体的数据过于复杂会造成较大的性能消耗。

另外，父对象的子碰撞体，在刚体中将作为复合碰撞体(Composite Collider)成为一个整体，在场景中被视为一个集体进行控制。

2） 精度需求不高应避免大量使用自带 Mesh 组件

Unity3D 引擎内置的多边形的模型精度比较高，当对于物体的精细度要求不高时，因避免大量使用 Unity 自带的 Sphere 等 Mesh 组件，可以导入一系列简单的 Low Poly 风格的 3D 模型代替。

3） 使用 Profiler

Unity 场景中的物理计算的总量，是由场景中非休眠刚体和碰撞体的数量以及碰撞体的复杂度决定的。在程序运行时可以使用 Unity 的性能分析工具—Profiler，来查看所制作的游戏在各方面所耗费的时间，分析当前游戏在各种情况下 GPU、CPU 性能，为后续的开发和优化提供数据参考。

4. Shader 优化方案

对数据、文字、声音、图形、图形和视频等信息进行加工处理就是计算机的基本功

哔哩哔哩次元游戏人生

能，在这之中的数据可以分为两个大类：一类是数值数据，这类数据是表示数量大小的数据，如+314，-3.14 等，有“量”的概念；而另外一类是非数值数据，这类数据并不会用来表示数值的大小，通常情况下也不会对这类数据进行算术运算，比如字符、字符串、图形符号等。在现实生活中，人们通常使用的是十进制数进行数据的统计与计算，但是计算机内的直接存储和运算却难以直接使用十进制。因此，计算机便将十进制数作为人机交互的中介，对于数据的存储和运算就采用易于物理实现、运算规则简单、可靠性高逻辑判断方便的二进制数。

因为计算机中的数据都是以二进制形式进行存储和运算的，而文字、声音、图形、图形和视频等信息并不是以二进制进行表示的，因此要将它们进行数字化（即将非二进制数转换为二进制数）。

1） 数的加减乘除

① 加法运算

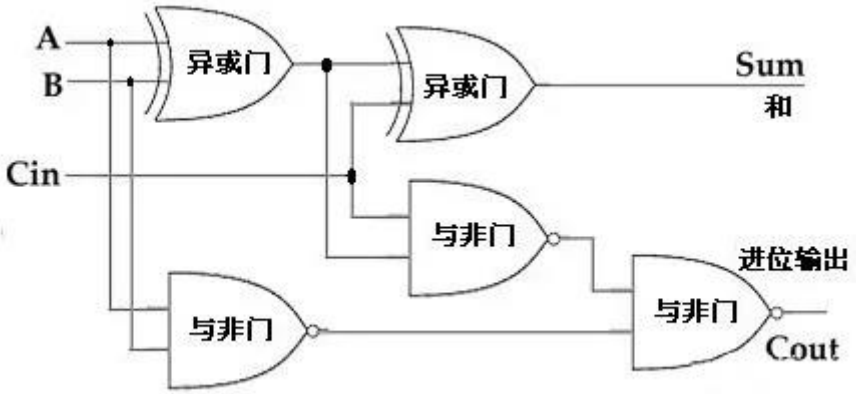


图55.　　图 6.2 全加器逻辑电路

计算机在实际运算过程中，加法二进制的最终数据会在计算机的全加器逻辑电路进行，也就是物理层。其中，异或门输出的是 Y=A^B（即两个值不相同结果为 1，相同则为 0），与非门的输出是先进行与运算(&，即两个数值相同时返回的结果为 1，不同时返回的结果为 0)再进行取反的非运算(~)，即 Y=!(A&B)。通过推导可以得到公式：

$$Sum = A\text{^}B\text{^}Cin \quad Count = (A\text{^}B\&Cin)|(A\&B) \quad (15)$$

```
public int GetSum(int a,int b)
{
        if (b == 0) //判断参数 b 的值
        {
            return a;//返回数值 a
        }
        int sum, carry;
        sum = a ^ b;
        carry = (a & b) << 1;
        return GetSum(sum, carry);
}//递归运算
```

“按位与”运算符（&）的规则是，当两个数值进行与运算时，先将两个数值转换为位数相同的二进制数，再依次对每一位数进行比较，只有当对应的两个二进位均为 1 时，返回的结果才会为 1；如果两个二进位有一个不为 1，返回的结果就是 0。

01010100&00111011 = 00010000

“左移”运算符（<<）的运算规则是，在“<<”左边的运算数的各二进位全部被左移

若干位，在“<<”右边的数值指定各二进位被移动的位数，高位丢弃，低位补 0。

如果 x=15，也就是 00001111，左移 2 位得 00111100 即 x<<2=60

由前面可以知道，加法运算其实就是不断的异或(^)加上移位操作的结果。

② 减法运算

当在进行减法运算时，首先想到的是加上一个负数。但是在计算机中，用于区分正负的符号位是不能直接参与运算的，在运算时必须分别计算符号位和数值位。当两个数值进行加法运算时，如果这两个数值是同号的（即都是正数或负数），则数值相加即可；如果这两个数值是异号的（即一正一负），则要进行减法运算。在进行减法运算时需要比较两者的绝对值大大小，用较大的数值去减去较小的数值，在得出运算结果时还要选择恰当的符号。

上述这些问题可以通过使用补码解决，补码可以把减法运算转换为加法运算。正数的补码最高位为符号 0，数值位和原码保持一致；而负数的补码最高位是符号 1，数值位是原码逐位取反得到反码，然后再在反码的最低位加一。数值 0 的补码只有一种形式，就是 N 位 0。

a = 3 = 0000 0011,b = 2 = 0000 0010

c = a-b=a+(-b)

c = 0000 0011 + 1111 1110

c = 1 0000 0001

因为是 8 位计算，最高位进位得到的 1 加入其中就是 9 位，超过了 8 位限制因此进位得到的 1 被舍弃，最终得到的结果就不是 1 0000 0001 而是 0000 0001。

在计算机中，由于机器码的位数是有限的，当最高位因为计算而进位超出了给定的取值范围，就被称为溢出。当两个正数进行加法运算，所得到的最终结果大于了机器码所能表示的最大正数，这一情况就被称为正溢。当两个负数进行加法运算时，所得到的最终结果小于了机器码所能表示的最小负数，这一情况就被称为负溢。在开发过程中出现溢出会导致计算的最终结果出现错误，因此正常情况下溢出是不被允许的。

③ 乘法运算

在计算机中乘法有两种计算方式，一种就是循环加法，一种是乘法器，乘法器分为模拟乘法器和硬件乘法器。

2*3=0000 0010 * 0000 0011

3 的第 0 位是 1,2 左移 0 位，结果为 0000 0010 = 2

3 的第 1 位是 1,2 左移 1 位，结果为 0000 0100 = 4

3 的其他位都是 0，因此不再移位

两次左移的结果相加，即 0000 0010 + 0000 0100 = 2+4 = 6

可以看出乘法器是根据乘数的 bit 位的 0 与 1，来将被乘数进行左移多少位，然后累加得到最终的结果。

④ 除法运算

最简单的除法运算可以通过循环减法来实现，比如 5 除以 2,5 一直减去 2，直到减完的结果小于 2，得到的就是除法的结果，余数就是剩下的 1；除法也具有专门做除法的逻辑电路。

50 / 20 = 0011 0010 / 0001 0100

取 50 的最高位 001|1 0010，001 < 0001 0100

左移一位 0011| 0010, 0011 < 0001 0100

左移一位 0011 0|010，0 0110 < 0001 0100

左移一位 0011 00|10, 00 1100 < 0001 0100

哔哩哔哩游戏人生

左移一位 0011 001|0, 001 1001 > 0001 0100，商 1，余 101|0

将大于 20 的数位记录下来就是 0000 0010，结果为 2，余数 1010 = 10

所以结果就是 50 / 20 = 2 余 10

可以看出除法器是根据从被除数高位开始取数，每次移动一位取到数后和除数比较，如果大于除数就记录当前被除数的数位，然后将记录的结果变为二进制数就是最后的结果。

虽然现在在进行乘法运算和除法运算时，获得了除法器和乘法器的帮助使得计算的效率比过去的累加和累减高得多，但是这些操作依然是非常消耗性能的操作，尤其是在进行除法运算的时候，还必须要对数值进行大于小于的判断。因此在片元作色器的编写过程中应该尽可能的避免高性能消耗的除法操作，转而使用其他低性能消耗的运算进行组合替代。

2） 逻辑判断

CPU 的主要功能是解释计算机指令以及处理计算机软件中的数据，而逻辑判断是所有程序中必不可少的操作，一般情况下的逻辑判断都是在 CPU 上运行的，因此 CPU 是一台计算机的运算核心和控制核心。而 GPU 则和 CPU 不同，GPU(Graphics Processing Unit)作为计算机中的图像处理器，它比 CPU 更加的专门化，GPU 处理器专门用来进行图像运算，在执行高复杂度的数学计算和几何计算时，运行效率远远高于 CPU 的运行效率。换句话说，GPU 图像处理器就是专为执行复杂的数学和几何计算而诞生的。

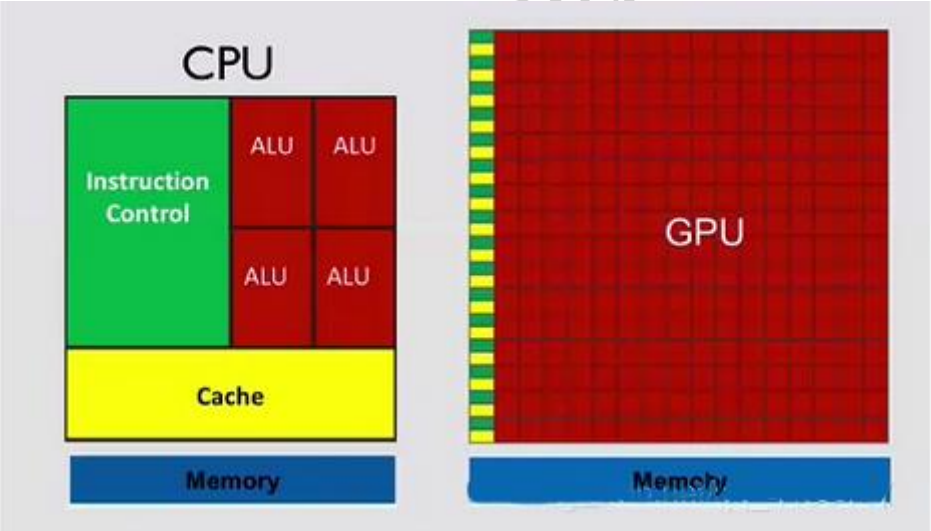


图56.　　图 6.3 架构图

CPU 和 GPU 拥有各自擅长的领域，GPU 擅长处理那些具有极高的可预测性和大量相似的运算以及高延迟、高吞吐的架构运算。而 CPU 所擅长处理的就是需要快速完成响应的实时信息，因此 CPU 便需要针对延迟进行特别的优化，所以晶体管数量和能耗都被用在了分支预测、乱序执行、低延迟缓存等控制部分。

在 Unity Shader 中编写的三目运算最后也会被编译为 if-else 语句，毕竟三目运算实际上仅仅只是简化了的 if-else 语句。虽然 Shader 中的每个片元依然会执行逻辑判断的每个分支，但是寄存器的写入只在片元采取行动的分支上执行。因此在 Shader 可以用 Shader 变体来进行不同状态下的渲染分支，用一些函数去代替 if-else 的判断，不推荐在 Shader 中使用逻辑判断，因为会带来不必要的性能消耗。

例如可以用 setp()函数来做大小与的判断：

a = step(b, c);// 等价于 if (b <= c) a = 1 else a = 0

当需要为角色添加一个描边光效时，可以根据 NdotV 的结果来设定描边光效越靠近边

缘越强，然后使用一个变量去控制描边光效在角色表面的范围。

在 Shader 代码编写之初，使用的是三目运算进行判断：

fixed3 c = (1 - NdotV) < _StatusColorEdge ? fixed3(0,0,0) : blendColor;

经过一次优化，将三目运算变更为 clamp 函数和 ceil 函数的组合，clamp 函数会保证 NdotV 和边界控制的结果永远都是在 0-1 的范围，保证值的正确性；而 ceil 函数会返回大于或者等于指定表达式的最小整数，即将结果向下取整变为 0 或者 1，然后再对颜色进行插值运算：

fixed temp = ceil(clamp(1 - NdotV - _StatusColorEdge, 0, 1));

fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);

最后用 step 函数代替两个函数，同时减法也换成了加法：

fixed temp = step(NdotV + _StatusColorEdge, 1);

fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);

当开发者在 Shader 脚本中使用 if 进行判断的时候，可以将其拆分为两种情况：

if (a > b)

{x}else

{x}

判断条件中的 b 为静态条件，即 const float 或者 define。当 b 为静态条件时，编译器会对这类情况做优化，使其直接进行批量计算，因为同一批像素从 Shader 中获得的都是相同的数值。

判断条件中的 b 为动态数值，也就是说具体的值在运行前无法得知的，只有在程序运行的时候才会知道具体的值。就会导致 GPU 做分批处理，无法最大化利用同时计算，因为无法确定的值就会导致同一批像素在一个区间内浮动，那么同一时间有的返回 true 有的返回 false。

三、 快捷键

1. Unity

① Ctrl + Shift + F 移动物体与视窗对齐(Align With View)

② Ctrl + Alt + F 将物体移动到 Scene 视图中央

③ V 顶点捕捉

④ Ctrl+D 复制

⑤ Ctrl + Shift 自动捕捉平面

⑥ 按住 Alt 键在 hierarchy 窗口可快速完全展开和完全收起

⑦ Shift+Space 最大化窗口

⑧ Shift+F 动态聚焦所选物体（复按可以调整聚焦距离）

2. Visual Studio

① Ctrl + Shift + 空格键 查看代码重载（需要光标在括号中间）

② Ctrl + R 重命名

③ Ctrl + F 查找

④ Ctrl + K + C 快速注释选中，未选中就注释该行

⑤ Ctrl + K + U 快速取消选中的注释，未选中就取消该行的注释

⑥ Ctrl+键盘右键 跳过该单词

⑦ Shift+任意方向键 范围选取

⑧ Ctrl + Alt +方向键 多行编辑

⑨

哔哩哔哩次元游戏人生

四、 脚本栏的扩展

1. 序列化[Serializable]

表示类或结构可以序列化。

2. 强制序列化[SerializeField]

使 Private 变量在 Inspector 窗口中可见。

3. 属性标题[Header]

为后续区域代码拟定一个标题，用于区分和概述该区域代码含义。

4. 属性提示[Tooltip]

实现在 inspector 窗口，鼠标位于该变量名字上时，提示该变量的描述信息。

5. 空行属性[Space]

在 Inspector 窗口创建空行，以隔开上下可视参数。

6. 范围值属性[Range()]

使得变量的值仅在该范围内修改，并且可以在 Inspector 窗口呈现滑动条修改变量的效果。

7. 隐藏属性[HideInInSpector]

使 Public 变量在 Inspector 窗口中不可见。

8. 输入域[TextArea]

对于字数较长的字符串，扩展其在 Inspector 中的编辑区大小。

9. [RequireComponent(typeof(xxx))]

可要求我们的脚本对象必须含有一个该组件，若脚本对象没有该组件，则会为该脚本对象自动添加 xxx 组件。

10. [CreateAssetMenu(fileName = "xx", menuName = "xx", order = x)]

将 ScriptableObject 派生类型标记为自动在 Assets/Create 子菜单中列出，以便可以轻松创建该类型的实例并将其作为“.asset”文件存储在项目中。

11. 永远执行-[ExecuteAlways]

使脚本的实例始终执行，无论是作为播放模式的一部分还是在编辑时。

默认情况下，MonoBehaviours 仅在播放模式下执行，并且仅在它们位于包含用户场景的主舞台的游戏对象上时执行。它们不会在编辑模式下执行，也不会在预制模式下编辑的对象上执行，即使在播放模式下也是如此。通过添加此属性，MonoBehaviour 的任何实例都将始终执行其回调函数。

12. 禁止多组件-[DisallowMultipleComponent]

保证该脚本在脚本对象上的唯一性，用于 MonoBehaviour 或其子类，如果物体上已经有了这个组件，再次添加就会弹窗提示.并且添加的操作无效。

13. 添加组件菜单-[AddComponentMenu]

AddComponentMenu 属性允许您将脚本放置在“组件”菜单中的任何位置，而不仅仅是“组件->脚本”菜单。

14. 在编辑模式下执行-[ExecuteInEditMode]

使脚本的所有实例在编辑模式下执行。

默认情况下，MonoBehaviours 仅在播放模式下执行。通过添加此属性，

唯理哲理次元游戏人生

MonoBehaviour 的任何实例都将在编辑器处于编辑模式时执行其回调函数。

五、 VisualStudio

//TODO: (未实现）……

//UNDONE:（没有做完）……

//HACK:（修改）……

等到再次打开 VS 的时候，找到 ：视图>任务列表,即可显示所有带有 TODO 注释的代码位置。

六、 资源网站

https://itch.io/

3d 模型和动画：https://www.mixamo.com/#/

https://opengameart.org/

https://www.spriters-resource.com/

Shader 学习：https://learn.jettelly.com/course/unity-shader-bible/usb-chapter-1/forward-rendering/

模型资源下载：https://sketchfab.com/feed

特效纹理制作：http://mebiusbox.github.io/contents/EffectTextureMaker/

MatCap 获取：

① http://pixologic.com/zbrush/downloadcenter/library/#prettyPhoto

② https://www.google.com.hk/search?q=MatCap&newwindow=1&safe=strict&hl=zh-CN&biw=1575&bih=833&tbm=isch&tbo=u&source=univ&sa=X&ved=0ahUKEwju8JDTpZnSAhUGn5QKHawODTIQsAQIIg

③ https://www.pinterest.com/evayali/matcap/

哔哩哔哩次元游戏人生

七、 切片图集(Slice Atlas)

要在 unity 中对图集进行切片，需要下载 Unity 2D Sprite 相关组件。

15. 下载

如果你的项目中没有 Package Manager 界面，可以在 Unity 中找到 Window 下的 Package Manager，将 package Manager 界面显示出来。

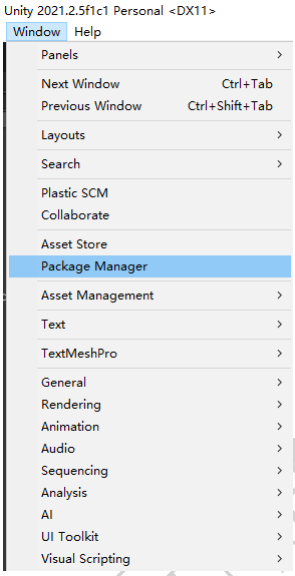


图57. 图 9.1 打开 PackageManager

打开了 package Manager 界面后，在 Unity Package 界面搜索 2D Sprite（在 Unity Registry 大类中）,找到并点击 Install 将其导入到 Unity 中。

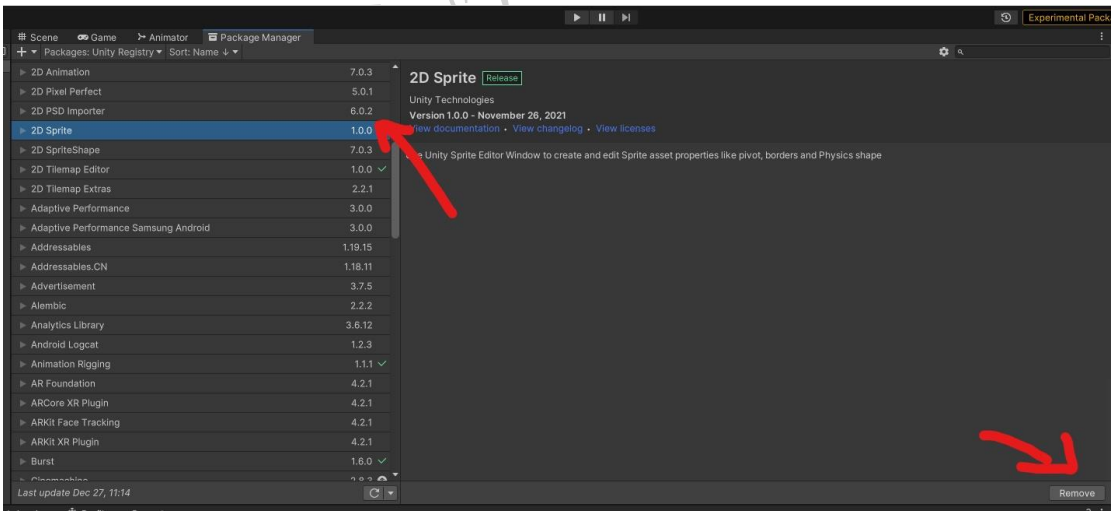


图58. 图 9.2 将 2D Sprite 导入

要将一张图集进行切片，只需要将图集的 SpriteMode 设置为 Multiple，然后进行切片。

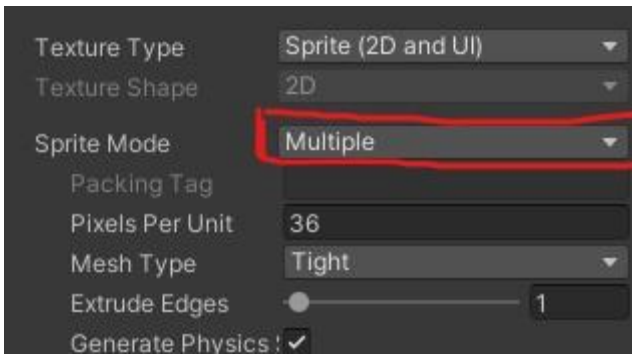


图59.　　图 9.3 设置精灵图为多图模式

16. 切片类型

1） Automatic-自动

让 Unity 通过基于每个潜在精灵的周围透明度生成边界来自动将纹理切片为多个单独的精灵。

2） Grid by Cell Size-按单元格大小

Pixel Size 的 X 值和 Y 值分别确定切分的像素瓦片的高度和宽度。

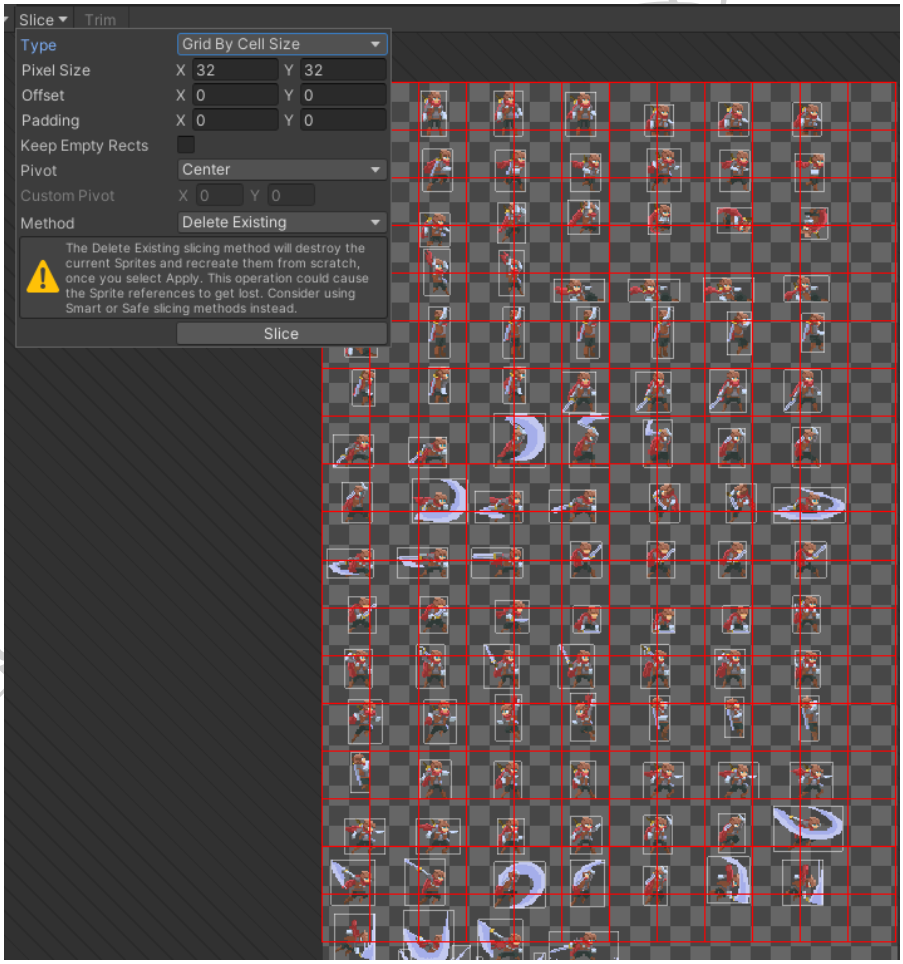


图60.　　图 9.4 按单元格大小切片

3） Grid by Cell Count-按单元格数量

C 即 Column（列）,R 即 Row（行）；它们确定切分的列数和行数。

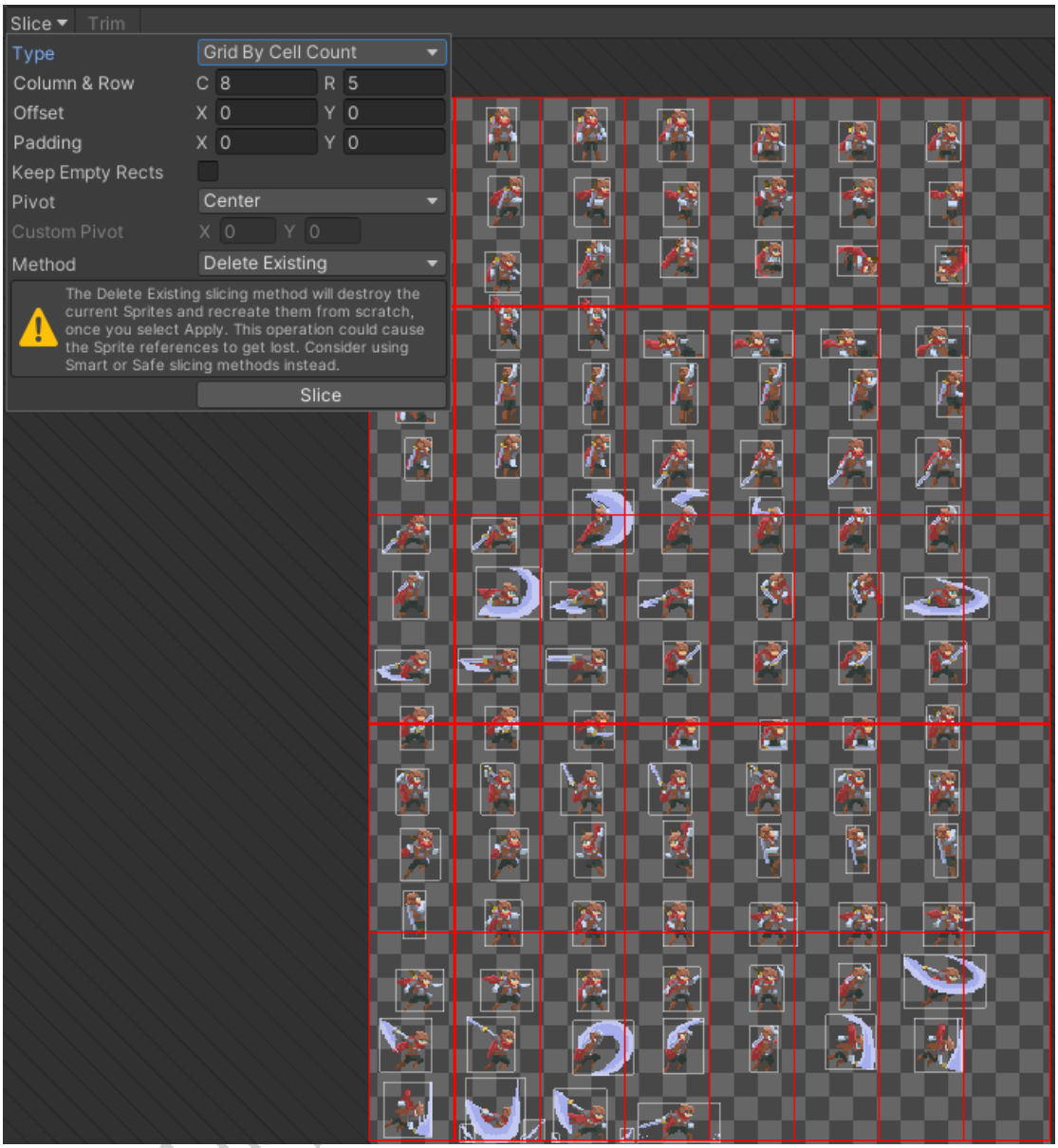


图61.　　图 9.5 按单元格数量切片

八、 未运行游戏的状态监视 ShaderGraph

在未运行游戏的情况下，想要观察 ShaderGraph 的效果，一种方法是在 Scene 窗口持续按住鼠标右键。

另一种方法是开启 Unity Scene 窗口的 Always Refesh，得到实时刷新的效果。

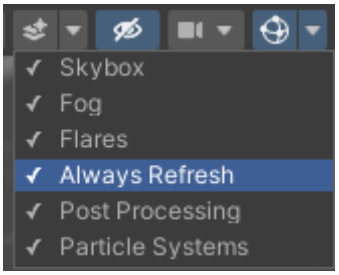


图62.　　保持实时刷新

九、 组件 Preset

Preset 允许用户存储该类型组件的预设，减少用户重复调整参数所耗费的时间。

点击组件右上角的工具图标，然后点击 Save Current 即可将该组件的数据保存为一个预设。

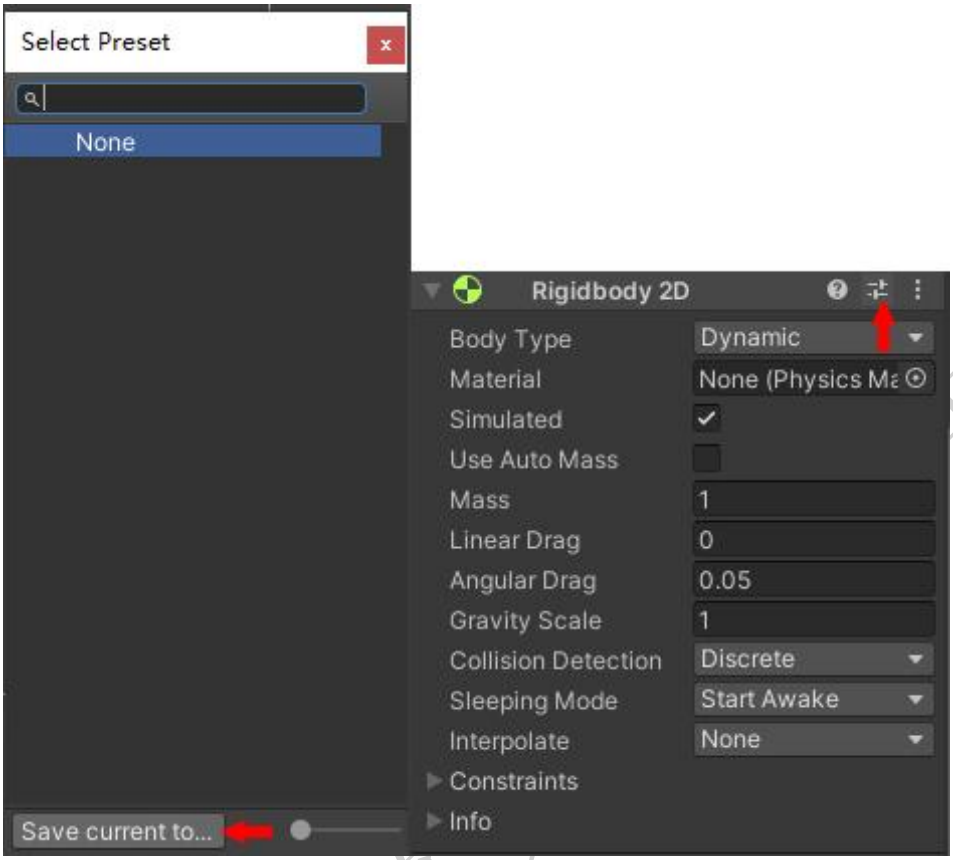


图63. 组件预设

十、 调整运行窗口颜色

我们可以更改 Unity 处于运行状态下的颜色，这可以帮助我们更容易的知道当前是否处于运行状态。

我们在 Edit->Preferences->Colors，找到 Playmode Tint，修改其颜色即可。

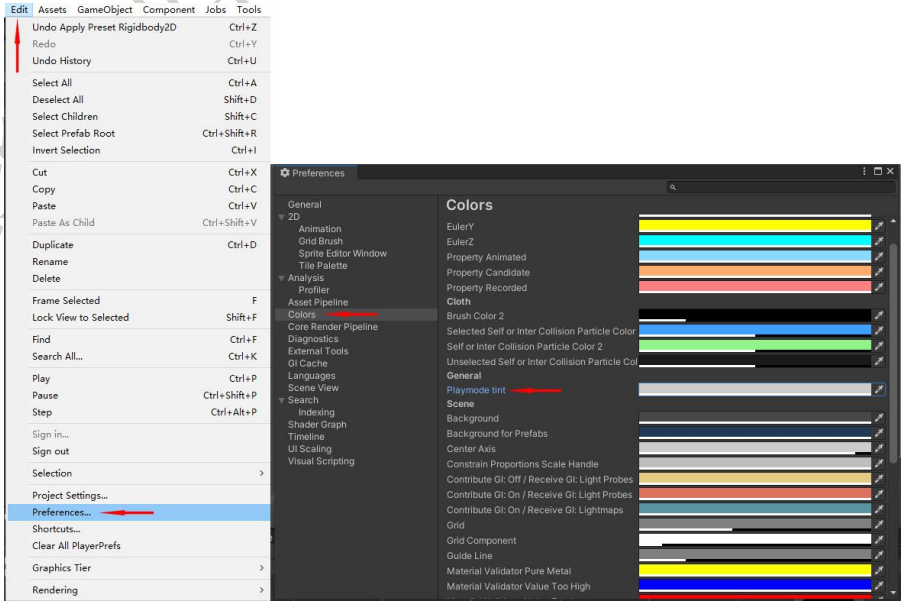


图64. 修改 Playmode 颜色

十一、 修改鼠标样式和启动图像

在 Unity 中找到 Edit->Project Settings->Player

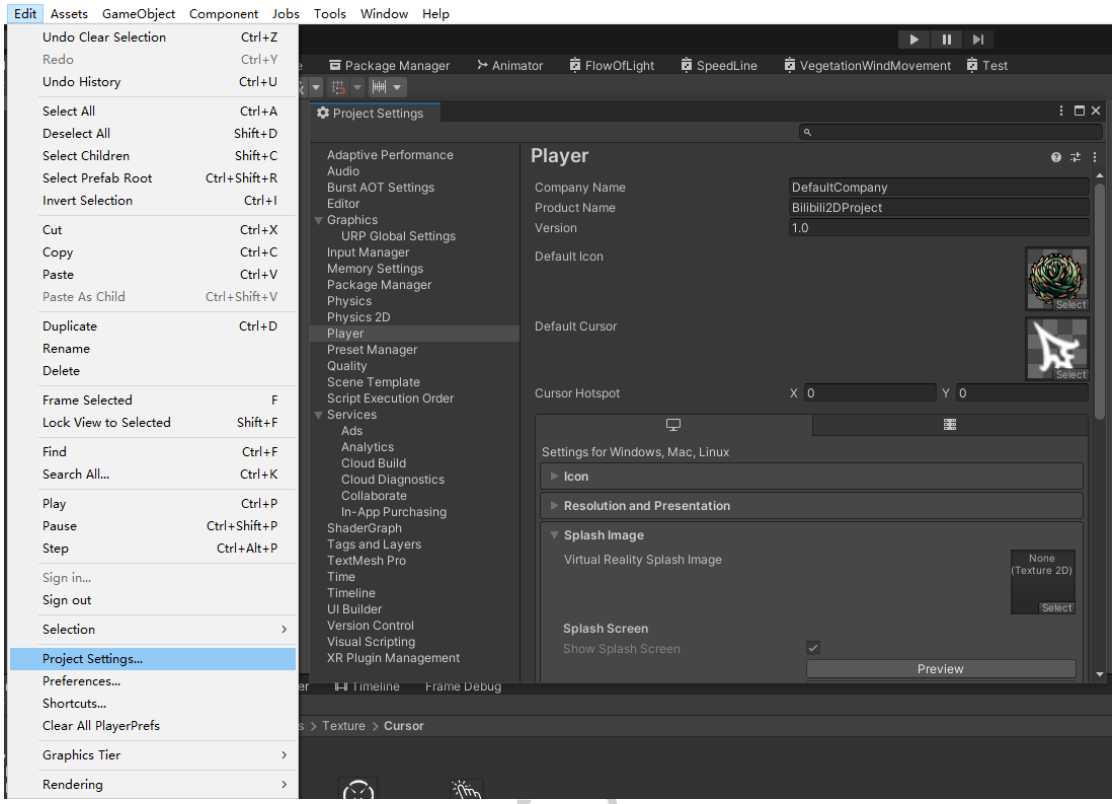


图65. Payer 设置界面

1. 设置鼠标样式

属性	功能描述
公司名称-Company Name	输入您的公司名称。
产品名称-Product Name	输入应用程序运行时出现在菜单栏上的名称。Unity 也使用它来定位首选项文件。
版本-Version	输入您的应用程序的版本号。
默认图标样式-Default Icon	选择要用作每个平台上应用程序默认图标的 Texture 2D 文件。
默认鼠标样式-Default Cursor	设置默认鼠标样式
Cursor Hotspot	设置鼠标作用点的偏移量,默认左上角为原点。

注意，要将纹理设置为鼠标样式，该纹理在 Unity 中的类型应为 Cursor。

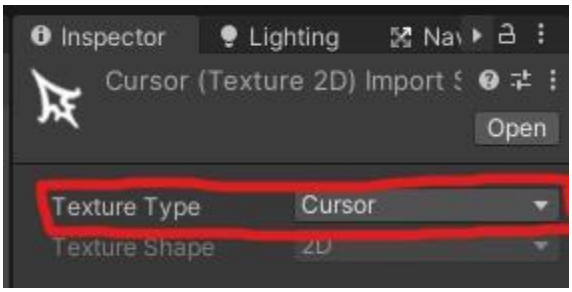


图66. 鼠标样式的纹理类型

2. 设置启动图像

在 Player 设置中找到 Splash Image 设置。

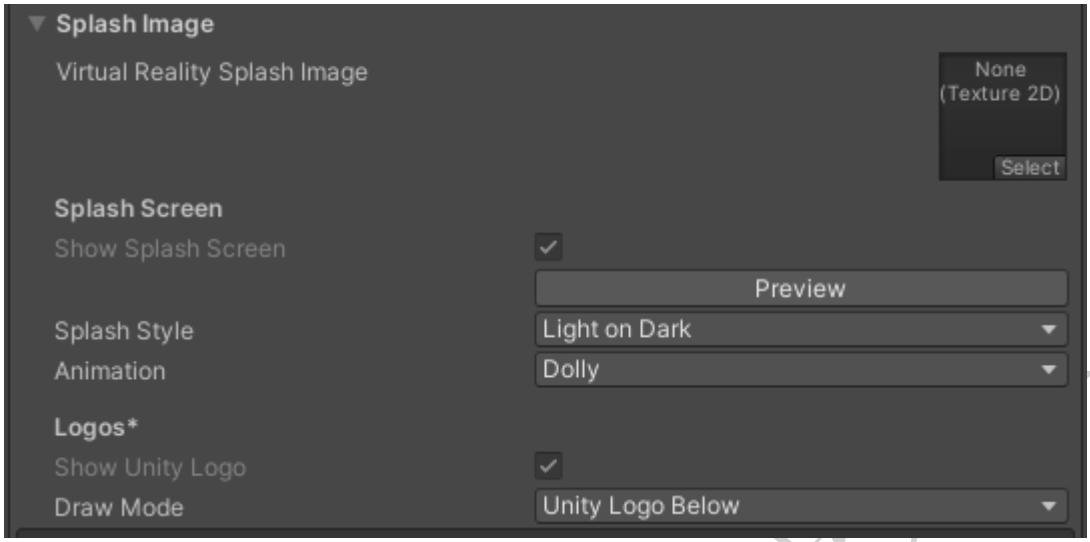


图67. Splash Image

设置名称		描述
Virtual Reality Splash Image		设置要用于使用虚拟现实的应用程序的图像。图片必须是 2D Texture .
显示启动画面-Show Splash Screen		默认启用。禁用此设置以在应用程序启动时不显示程序启动画面。如果是 Unity 个人版，则无法禁用此设置。
预览-Preview		选择预览按钮以在游戏视图中查看启动画面的预览。
动画类型-Splash Style		Unity logo 的颜色样式设置。
	Light on Dark	会使用浅色版本的 Unity Logo。这适合在深色背景上显示，这是默认设置。
	Dark on Light	选择 Dark on LightUnity ，会使用深色版本的 Unity Logo，最适合在浅色背景上显示。
Animation		设置启动画面如何在屏幕上出现和消失。
	Static	不应用任何动画。
	Dolly	Logo 和背景会有缩放效果。这是默认设置。
自定义-Custom		选择此设置可自定义背景和 Logo 缩放量。

哔哩哔哩次元游戏

	徽标缩放-Logo Zoom	当 Logo 到达 Logo 动画总持续时间的末尾时，Logo 的目标缩放（从 0 到 1）。
	背景缩放-Background Zoom	当背景到达初始屏幕动画的总持续时间结束时的目标缩放（从 0 到 1）。
Show Unity Logo		默认启用。禁用此选项不会在初始屏幕上显示 Unity 徽标。如果有是 Unity 个人版，则无法禁用此设置。
绘图模式-Draw Mode		选择 Logo 出现在初始屏幕上的顺序模式
	Unity Logo Below	Unity 的 Logo 将显示在其他 Logo 之下。
	All Sequential	依次显示 Unity Logo 以及 Logo 列表中的所有 Logo。
	Logo Duration	设置每个 Logo 出现在屏幕上的时间长度。值在 2 到 10 秒之间。
叠加不透明度-Overlay Opacity		调整覆盖不透明度设置的值以使 Logo 突出。这会影响 Logo 的背景颜色和图像颜色，具体取决于您设置 Splash 样式（Light on Dark 或 Dark on Light）的方式。 将不透明度设置为较低的值可以以减少此效果。也可以通过将其设置为 0 来禁用该效果。例如，如果 Splash Style 为 Light on Dark，具有白色背景，如果将 Overlay Opacity 设置为 1，则背景变为灰色，如果设置为 0，则背景变为白色。 如果是 Unity 个人版，此设置的最小值为 0.5。
模糊背景图像-Blur Background Image		启用此设置以模糊您设置的背景图像。如果禁用此设置，它将显示没有模糊效果的背景图像。

哔哩哔哩次元游戏人生

背景图像-Background Image	设置对 Sprite 图像以用作背景。Unity 会调整背景图像以填充屏幕。

十二、 图片跟随文本自适应

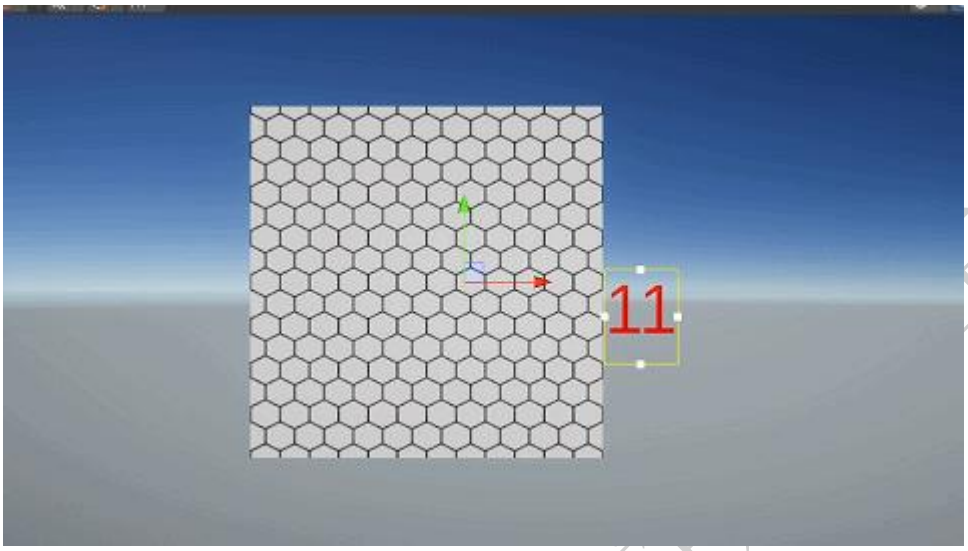


图68. 图像跟随文本移动（这是一个 gif 图，可另存为查看）

在 Unity 中我们可能经常会遇到一种需求，就是让图像跟随着文本进行移动。

要实现这种效果，我们可以使用代码去实现，但这会麻烦许多。我们就可以考虑使用 Unity 中的内置组件，也就是 Content Size Fitter 组件。

首先我们需要创建一个 Image 和一个 Text，如下图所示。

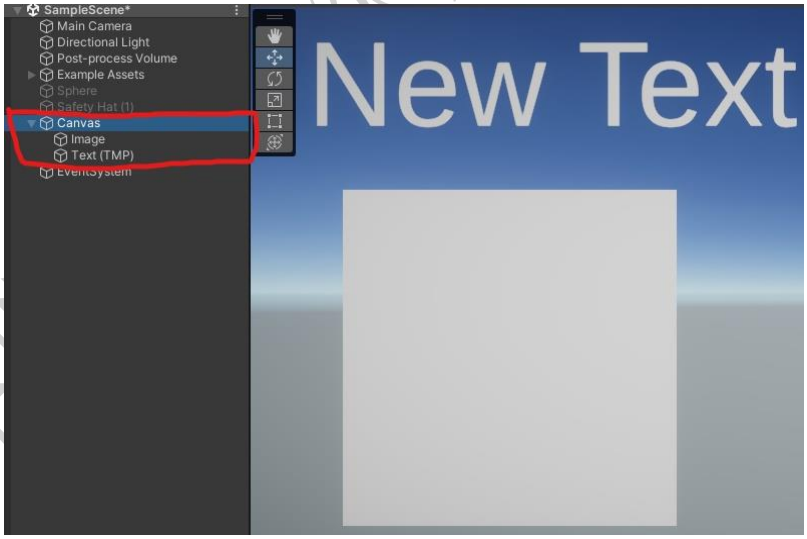


图69. 创建图像和 Text 文本（我这里使用的 Text Mesh Pro）

下面就可以让 Image 成为 Text 的子物体，接着为 Text 添加一个 Content Size Fitter 组件，并设置 Horizontal Fit 为 Preferred[1] Size。

[1] adj. 更合意的；优先（考虑）的；被偏爱的 v. 偏爱；宁可（prefer 的过去式和过去分词）

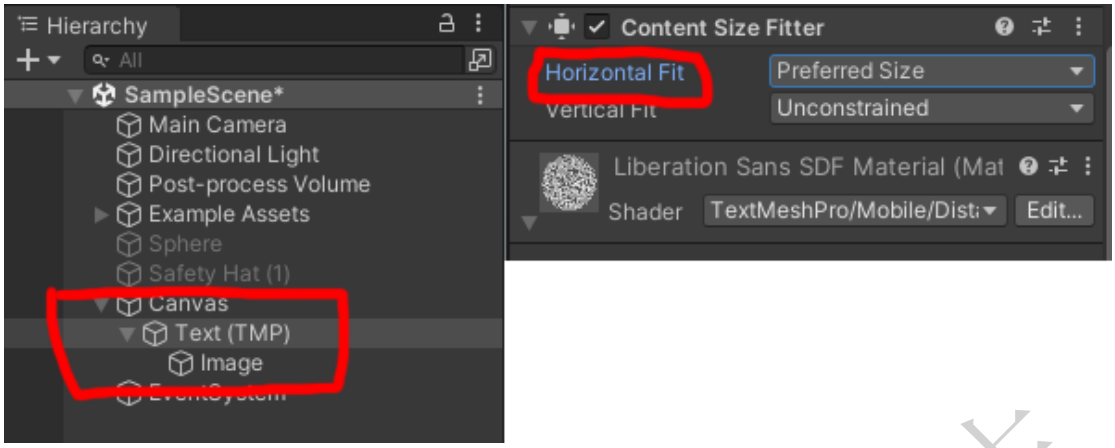


图70.　　让 Image 成为 Text 的子物体

最后我们需要设置 Image 的 Pivot 位置。如果将 pivot 设置在左边，它就会以 Text 的左边为基准进行移动；将 pivot 设置在右边，它就会以 Text 的右边为基准进行移动。

十三、　修改 Unity 的脚本模板

在 Unity 中新建脚本时，我们新建的脚本里往往就已经有了些基础的内容，这就是通过脚本模板去实现的。但是，在实际使用的过程中，我们往往会删除 Unity 默认构建的这些内容。

下面我们就需要去修改脚本模板，去自定义新建脚本里的默认内容。

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

Unity 脚本|0 个引用
public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    Unity 消息|0 个引用
    void Start()
    {

    }

    // Update is called once per frame
    Unity 消息|0 个引用
    void Update()
    {

    }
}
```

图71.　　新建 C#脚本时的默认内容

我们找到存放 Unity 引擎的根目录，引擎版本->Editor->Data->Resources->ScriptTemplates[2]。在 ScriptTemplates 里找到 NewBehaviourScript，这个就是我们新建 C#脚本时的内容模板，我们只需要修改它里面的内容即可。

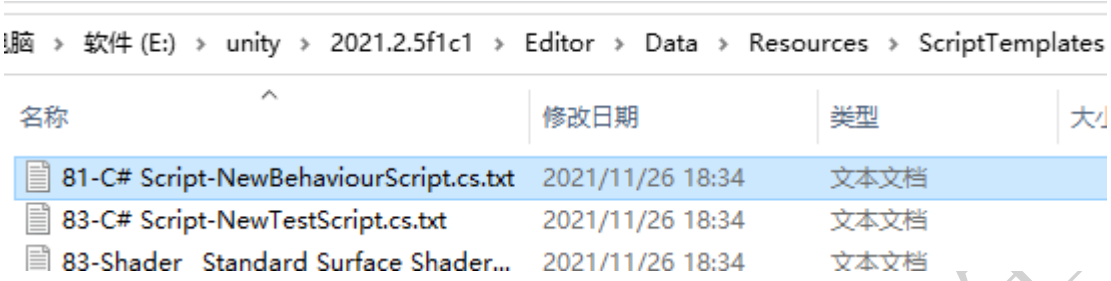


图72. Unity C#模板文件

我们可以看到，这个目录文件下还有其他的模板，就拿 Shader 的模板举例吧。在 Build In 渲染管线下，Unity 新建的默认 Shader 脚本，其包含的基础内容符合我们的要求，我们不需要进行大量的更改。

但是在 URP 和 HDRP 渲染管线下，我们就需要修改 Shader 脚本里的内容了，因为 HDRP 和 URP 是使用的 HLSL 语言，不是 CG 语言。我们所需要引入的相关内容包也不相同，且我们需要告知 Shader 使用的是哪一个渲染管线，并删除那些不适用于 URP 和 HDRP 的内容。

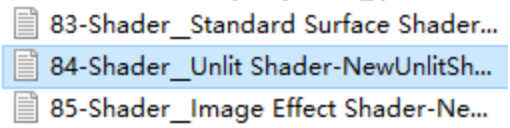


图73. 修改 Unlit Shader 的模板内容以适应 URP 和 HDRP

首先先将 CG 语言修改为 HLSL 语言，再在标签里添加"RenderPipeline"并使其等于"UniversalRenderPipeline"。

为了使我们的着色器能够完美编译并使其过程更加高效，我们必须包含依赖项“Core.hlsl”，Core.hlsl 本身取代了“UnityCg.cginc”。

Core.hlsl 必须编写其位置的完整路径，这个路径可以自己去查找，就在当前项目的 ShaderLibrary 文件夹下。

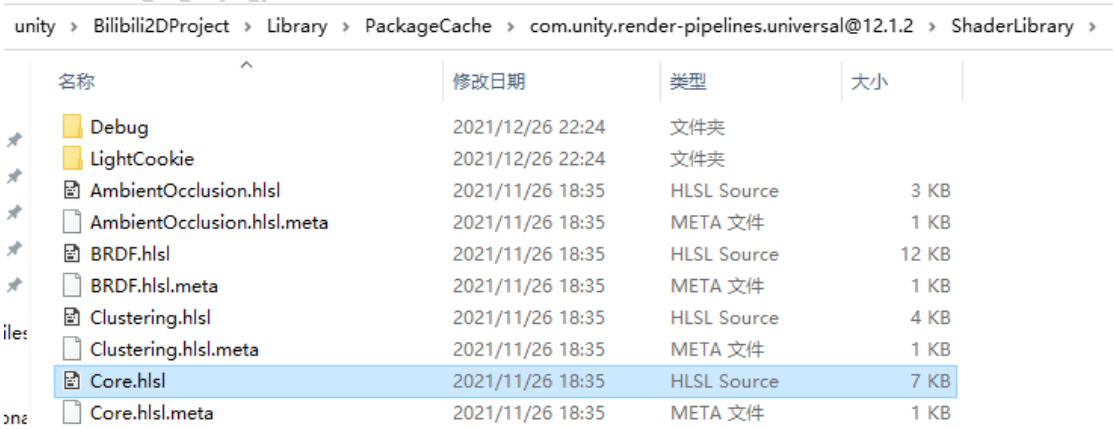


图74. Core.hlsl 路径

一旦我们替换了这些依赖项，GPU 将无法编译雾坐标，我们需要删除这部分内容。同

[2] n. 模板；范本（template 的复数）；属性单元

时也无法编译 UnityObjectToClipPos 函数，我们将不得不使用包含在“Core.hlsl”中的 TransformObjectToHClip 函数去代替。

如果此时我们保存并返回 Unity 创建一个新的 Shader，我们可以看到在控制台中生成了“无法识别的标识符”类型的错误，这是什么原因呢？请记住，默认情况下，片元着色器阶段的 col 向量都是 fixed4 类型。

Universal RP 不能编译“fixed”类型的变量或向量，我们将不得不使用“half 或 float”。

我们可以手动替换该类型的变量和向量。或者包含“HLSLSupport.cginc”依赖项，为着色器编译添加辅助宏和跨平台定义。

一旦包含了这种依赖关系，我们就可以使用 fixed 类型的变量和向量，因为现在我们的程序将根据其精度识别这种类型的数据，并自动将它们替换为 half 或 float。

哔哩哔哩次元游戏人生

Unity Editor

需要使用编辑器类代码时，需要在脚本前面加上 using UnityEditor 命名空间。

一、 Editor 文件夹

Editor 文件夹可以放在工程项目的任何文件夹下，一个工程项目中可以存在多个"Editor"文件夹。编辑器扩展相关的脚本都要放在该文件夹内，该文件夹中的脚本只会对 Unity 编辑器起作用。

在项目打包的时候，Editor 文件夹中的内容不会被打包到项目中。如果与 Unity 编辑器相关脚本不放在该文件夹中，打包项目时可能会出错。如果非要有些编辑器相关脚本不放在该文件夹中，需要在该类的前后加上 UNITY_EDITOR 的宏定义。

二、 Editor Default Resources

该文件夹需要放在 Assets 根目录下，用来存储编辑器所需要的图片等资源，书写的时候需要注意中间有空格隔开。在进行项目打包时，此文件夹也不会被打包，访问方法为：EditorGUIUtility[1].Load()。

也可以在 Editor 文件夹内创建一个 Resources 文件夹，将相关资源放在该文件夹内，通过 Resources.Load()获取文件夹中的资源，也是可以的。

三、 Gizmos 文件夹

Gizmos 文件夹也需要放在 Assets 根目录下，可以用来存放 Gizmos.DrawIcon()的图片资源。

哔哩哔哩次元游戏人生

[1] Utility adj.有用的，多功能的 n.应用程序；实用

Unity Editor 实践

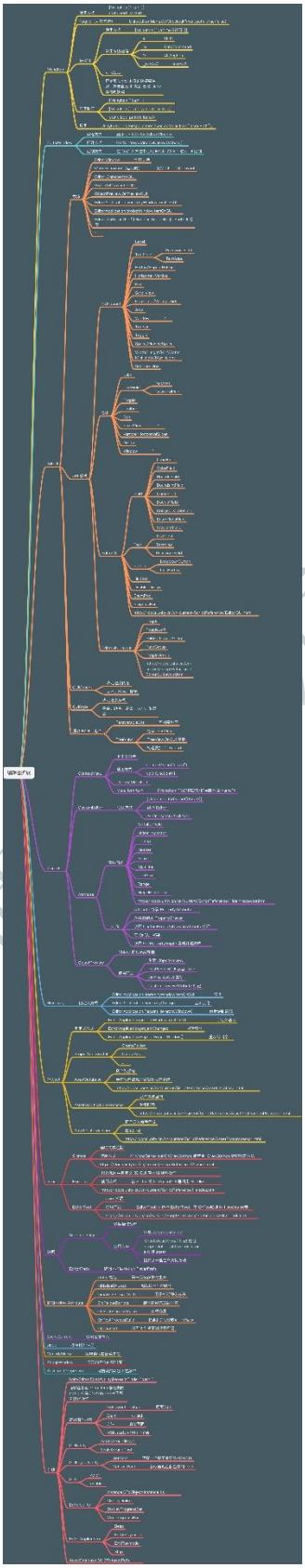


图75.　　图 1. 1 知识地图

哔哩哔哩 游戏人生

我

哔哩哔哩次元游戏人生

Unity 工具

一、 Terrain Tool

1. 快捷键

① A＋ 鼠标拖动 调整笔刷强度
② S＋ 鼠标拖动 调整笔刷大小
③ D＋ 鼠标拖动 旋转笔刷
④ Shift 使笔刷变得平滑
⑤ Ctrl 反转笔刷功能，地形的升高变为降低

二、 Cinemachine

三、 Recorder

手册地址：https://docs.unity3d.com/Packages/com.unity.recorder@3.0/manual/index.html

1. 下载

在 Unity 中找到 Window 下的 Package Manager，将 package Manager 界面显示出来。

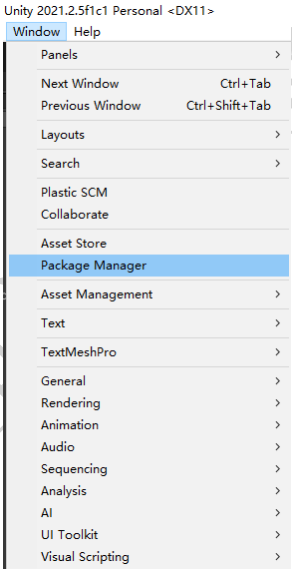


图76. 图 3.1 打开 PackageManager

打开了 package Manager 界面后，在 Unity Package 界面搜索 Recorder,找到并点击 Install 将其导入到 Unity 中。

哔哩哔哩 游戏人生

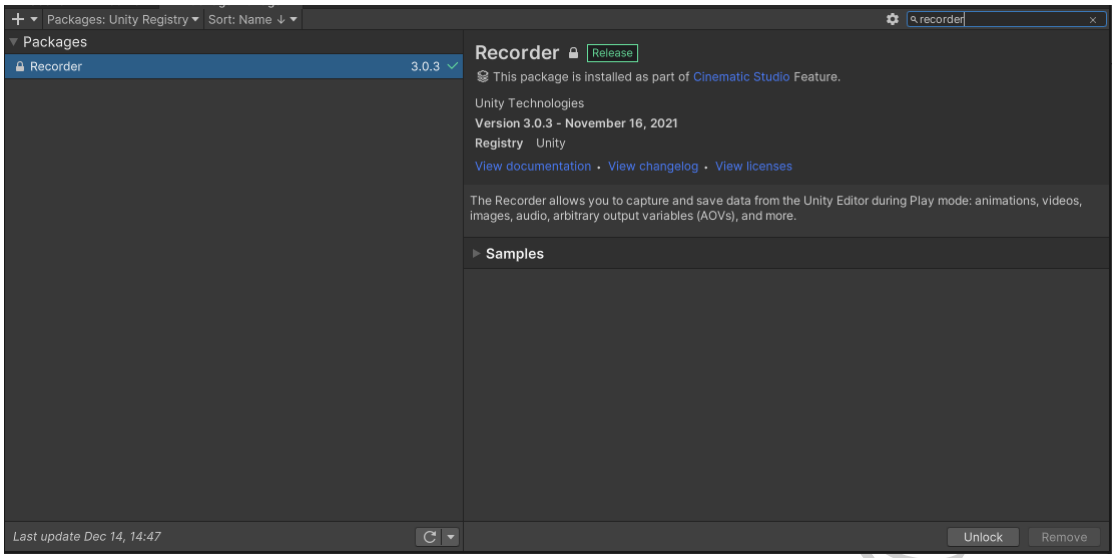


图77.　　图 3.2 下载 Recorder

2. 打开 Recorder 界面

从 Unity 的主菜单（Window > General > Recorder > Recorder window）打开 Recorder 窗口。

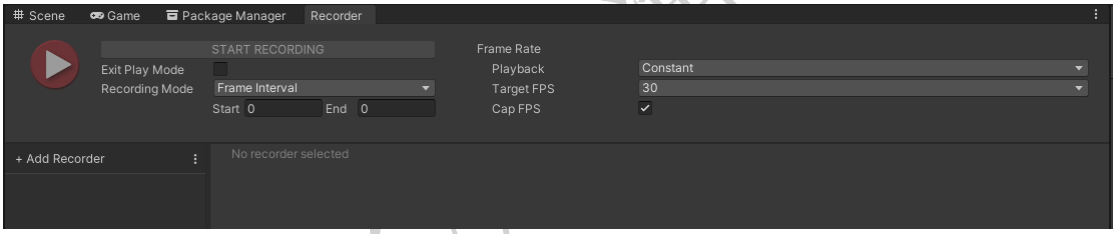


图78.　　图 3.3 Recorder 界面

可以使用 Recorder 输出以下格式的数据。

Animation Clip Recorder：生成 .anim 格式的动画剪辑。

Movie Recorder：生成 .mp4 或 .webm 格式的视频。

Image Sequence Recorder：生成 .jpeg、.png 或 .exr (OpenEXR) 格式的图像文件序列。

Audio Recorder：生成 .wav 格式的音频剪辑。

3. Recording Mode(记录模式)

① Manual

当您分别手动单击开始录制或停止录制时开始或停止录制。

② Single Frame

记录单帧。使用目标框架属性指定此框架。

③ Frame Interval

在播放模式下记录一组连续的帧。使用 Start 和 End 属性指定开始和停止的时间。

④ Time Interval

在播放模式下记录特定的持续时间（以秒为单位）。使用 Start 和 End 属性指定开始和停止的时间。

4. Frame Rate 帧率属性

使用帧速率属性指定如何在录制期间限制帧速率。帧速率会影响 Recorder 输出的文

件的大小和数量。

图79.　　图 3.4 Frame Rate

1） Playback

指定如何在录制过程中控制帧率。

① Constant（恒定）

将记录器限制为特定的帧速率。使用 Target 属性指定此速率。

② Variable（变化）

使用应用程序的帧速率。使用 Max FPS 属性指定录制期间应用程序的速率上限。

注：该记录视频时不支持变帧率。

2） Target FPS

设置捕获记录的帧速率。当您将 Playback 设置为 Constant 时，会出现此属性。

无论您以更高还是更低的帧速率运行应用程序，Recorder 都会以该速率进行捕获。例如，如果您将目标 FPS 设置为 30 fps 的自定义值，但您以 60 fps 运行应用程序，则记录器以 30 fps 进行捕获。

3） Max FPS

限制播放模式下的更新速率。当您将 Playback 设置为 Variable 时，此属性可用。为防止您的应用程序超过此帧速率，记录器会在播放期间插入延迟。使用此属性可减小输出的文件大小。

4） 文件输出

① 可用占位符

可以在文件名和路径字段中插入占位符，以在文件名和路径字符串中包含自动生成的文本，记录器用于保存输出文件。Recorder 用适当的上下文值替换这些占位符以构建实际字符串。

可以在同一字符串中组合多个占位符。例如，您生成录音的日期和时间。

在+通配符菜单可以帮助您快速添加它们的文件名字符串，还可以手动将字符串中的任何输入。要在路径字符串中使用占位符，必须手动输入它们。

注意：确保在占位符名称中包含 <> 括号。另请注意，占位符名称区分大小写。

占位符	Description-描述	输出示例
<Recorder>	• 在记录器窗口中：记录器列表中的记录器名称。 • 在时间轴轨道中：记录器剪辑的名称。	My Recorder
<Time>	记录生成的时间。使用 00h00m 格式。	16h52m
<Take>	编号值。使用 000 格式。	002
<Date>	记录的生成日期。使用 yyyy-MM-dd 格式。	2020-11-03
<Project>	当前 Unity 项目的名称。	My Project

哔哩哔哩次元游戏人生

<Product>	产品名称 Unity 的播放器常规设置中的字段。	My Product
<Scene>	当前 Unity 场景的名称。	My Scene
<Resolution>	以像素为单位的输出图像尺寸（宽 x 高）。使用 WxH 格式。	1920x1080
<Frame>	当前帧数。使用 0000 格式。 这对于将每一帧作为单独文件输出的图像序列记录器很有用。	0154
<Extension>	输出格式的文件扩展名（不带句号）。	png
<GameObject>	正在录制的游戏对象的名称。	My GameObject
<GameObjectScene>	包含正在录制的游戏对象的场景的名称。	My Scene

图80. 表 3.1 占位符功能表

5. Capture-捕捉设置

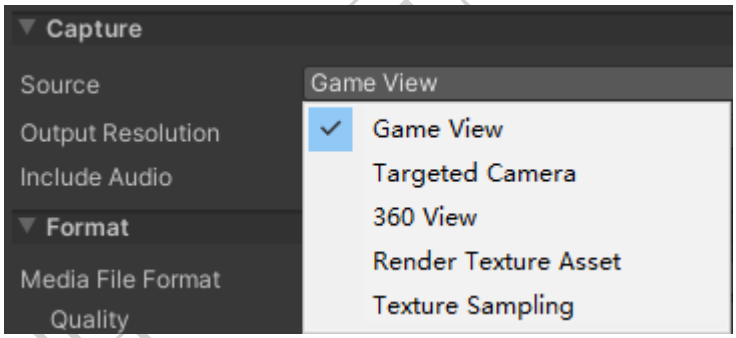


图81. 图 3.5 捕捉设置

1） Source-捕捉源

① Game View

记录在游戏视图中渲染的帧。

② Targeted Camera

记录由特定相机捕获的帧，即使游戏视图不使用该相机。

③ 360 View

录制 360 度视频。

④ Render Texture Asset

记录渲染纹理中渲染的帧。

⑤ Texture Sampling

在捕获过程中对源摄像机进行超采样以在记录中生成抗锯齿图像。

2） Output Resolution-输出分辨率

① Match Window Size

匹配当前选择的游戏视图的分辨率和纵横比。

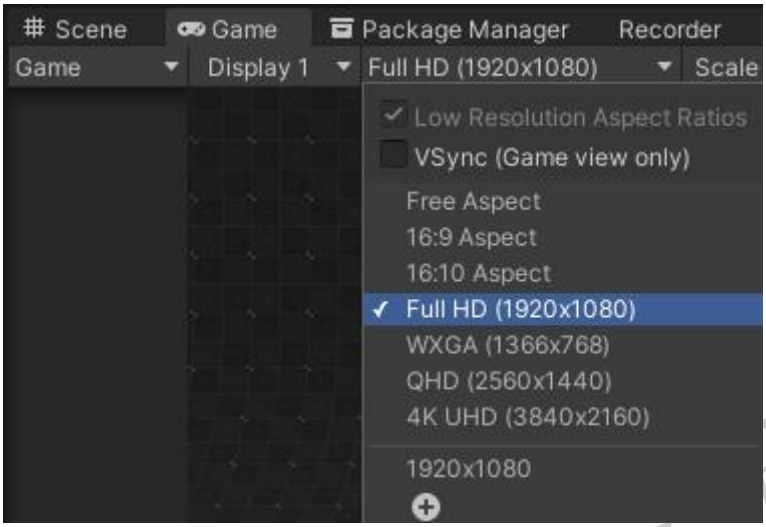


图82.　　图 3.6 游戏视图的分辨率

② Custom-自定义

使用用户在 W 和 H 字段中提供的自定义宽度和高度值。

四、 Post Processing

后处理是全屏图像处理效果的总称，它发生在相机绘制场景之后但在场景渲染到屏幕上之前。后处理可以极大地改善产品的视觉效果，而只需很少的设置时间。

您可以使用后处理效果来模拟物理相机和胶片属性。

1. 开始后处理

1） 后处理层-Post-process Layer

要在场景中启用后期处理，请将 Rendering > Post Process Layer 组件添加到 Main Camera GameObject。此组件允许您为此后处理层配置抗锯齿，选择即将应用后处理的层，并选择哪些游戏对象能够触发此后处理层。

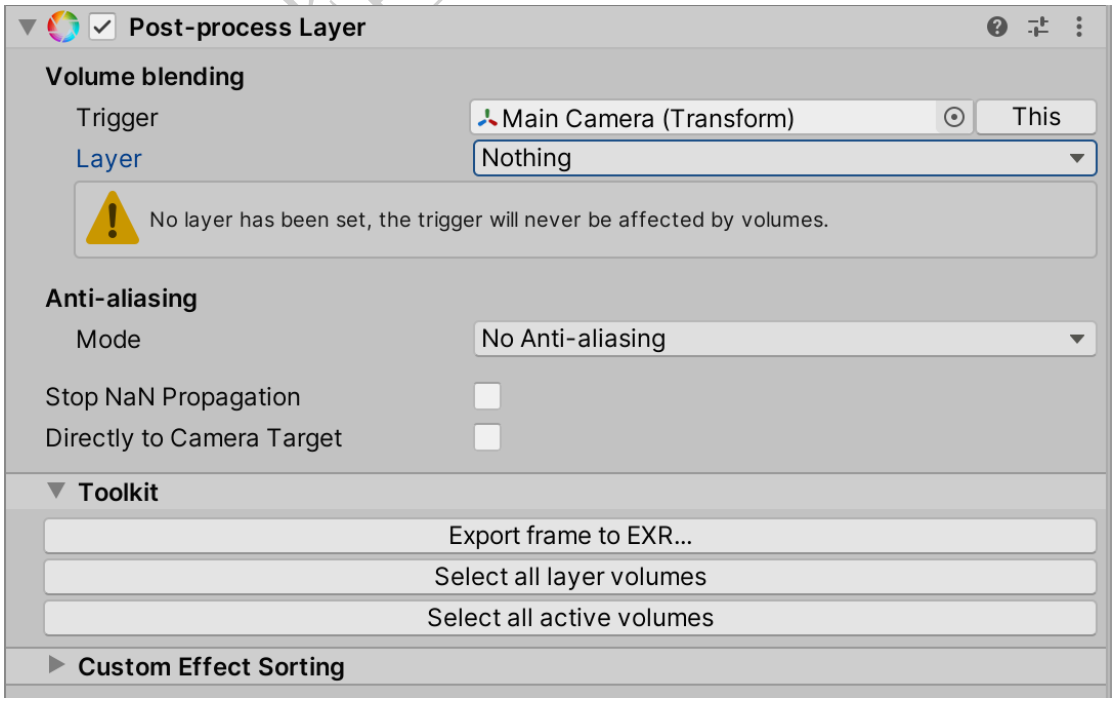


图83.　　图 12.1 Post Process Layer

Volume blending

2. 泛光-Bloom

1） 作用

Bloom 效果使图像中的明亮区域发光。

Bloom 效果还具有 Dirtiness 功能，您可以使用它来应用全屏的污迹或灰尘层来衍射 Bloom 效果，本质上就是往整个画面上叠加一张纹理。

2） Bloom settings

强度-Intensity	设置泛光的强度
临界值-Threshold	用于过滤亮度小于此数值的像素。也就是说，亮度小于此数值的像素不会参与泛光的计算。
渐变阈值-Soft Knee	为低于或高于阈值之间的过渡设置渐变阈值（0 = 硬阈值，1 = 软阈值）
Clamp	设置钳位像素的值以控制 Bloom 量。该值以伽马空间表示。
扩散-Diffusion	以与屏幕分辨率无关的方式设置遮蔽效果的程度。
镜头畸变比例-Anamorphic Ratio	设置畸变比例以垂直（范围 [-1,0]）或水平（范围 [0,1]）缩放 Bloom 。这模拟了变形镜头的效果。
Color	选择泛光的色调颜色。
快速模式-Fast Mode	启用此复选框可通过降低 Bloom 效果质量来提高性能。

3） Dirtiness settings:

纹理-Texture	为泛光叠加一张纹理，比如包含光晕或者灰尘的纹理
Intensity	用于调整纹理叠加的强度

4） 细节

对于正确曝光的 HDR 场景，Threshold（临界值）应将 设置为 1，以便只有值大于 1 的像素会泄漏到周围的对象中。在 LDR 中工作时降低此值，否则 Bloom 效果将不可见。

5） 表现

降低 Diffusion 参数将使效果更快。Anamorphic Ratio 离 0 越远，速度越慢。如果您正在为移动或低端平台开发想获得显着的性能提升，请启用 Fast Mode。

较低分辨率的镜头 Dirtiness Texture 会导致更快的查找和跨体积混合。

哔哩哔哩次元游戏人生

设计模式

一、 单例模式

提供一个访问该类的全局访问点，并保证该类仅有一个实例。单例模式不仅可以保证实例的唯一性，还可以严格的控制外部对实例的访问。

1. Unity 泛型单例

1） 泛型参数的约束

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用 where 上下文关键字指定的。下表列出了六种类型的约束：

where T: struct

类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型。有关更多信息，请参见使用可以为 null 的类型（C# 编程指南）。

where T : class

类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。

where T：new()

类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，new() 约束必须最后指定。

where T：<基类名>

类型参数必须是指定的基类或派生自指定的基类。

where T：<接口名称>

类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。

where T：U

为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。

```
public class SingleComponent<T> : MonoBehaviour where T : SingleComponent<T>
{
//设置为私有字段的变量，在派生类中是无法访问的
    private static T instance;
    public static T Instance//[单例模式]通过关键字 Static 和其他语句之生成对象的一个实例，确保该脚本在场景中的唯一性
    {
        get
        {
            return instance;
        }
    }

    //关键字 Virtual，使此函数变为一个虚函数，让其可以在一个或多个派生类中被重新定义。
    protected virtual void Awake()
    {
```

```
            if (instance != null)//场景中的实例不唯一时
            {
                Destroy(instance.gameObject);
            }
            else
            {
                instance = this as T;
            }
        }
    //设置为 protected 类型的变量，派生类可以访问，非派生类无法直接访问
        protected virtual void OnDestroy()
        {
            Destroy(instance.gameObject);
        }

        public static bool IsInitialized//单例是否已初始化
        {
            get
            {
                return instance != null;//返回的是一个 bool 值，即该实例是否为空
            }
        }
    }
```

二、 观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为.

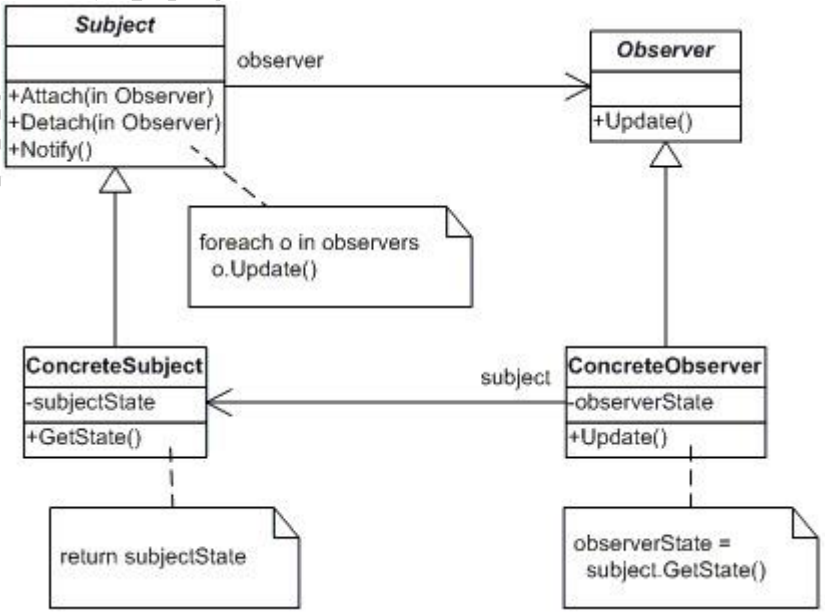


图84. 图 2.1 观察者模式结构图

在 Unity 中，观察者模式的实现是基于 C#的事件委托机制，要彻底的解除通知者和观

察者之间的耦合就可以采用事件委托（Delegate）。委托就是一种引用方法的类型，拥有参数和返回值，作为一种方法的抽象，是特殊的“函数类”，一旦我们为委托分配了方法，委托将会执行与该方法相同的行为，其实例代表了一个或多个具体的函数。C#中关于回调函数的实现机制便是委托，事件（Event）是基于委托的，事件为委托提供了一种用于实现类之间消息发布与接收的结构。在 C#中订阅与取消事件可以使用“+=”与“-=”，在 Unity 中可以通过添加 Handler 来监听事件消息，进而实现消息响应，可以方便的创建事件与编写响应，通过创建 Handle 来监听事件进而完成游戏逻辑。事件只能被其他的模块监听注册，但是不能由其他的模块触发。要触发事件，只能由模块内的公开方法在其内部进行事件触发，这样可以使得程序更加安全。但是在委托的使用过程中会出现过于依赖反射机制（reflection）来查找消息对应的被调用函数，这会影响部分性能.

哔哩哔哩次元游戏人生

数据结构

一、 基本概率和术语

数据(Data)：

对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素(Data Element)：

是数据的基本单位。有时，一个数据元素可由若干个数据项(Data Item)组成。例如：一本书的书目信息为一个数据元素，而书目信息中的每一项（如书名、作者名等）为一个数据项。数据项是数据的不可分割的最小单位。

数据对象(Data Object)：是性质相同的数据元素的集合，是数据的一个子集。

数据结构(Data Structure)是相互之间存在一种或多种特定关系的数据元素的集合。

1. 数据结构三要素

逻辑结构、存储结构（物理结构）、数据的运算。

其中，逻辑结构又分为线性结构（线性表、栈、队列）和非线性结构（树、图、集合）。

哔哩哔哩次元游戏人生

数据结构习题

一、 数组

1. 给你一个整数数组 nums 。如果任一值在数组中出现至少两次，返回 true；如果数组中每个元素互不相同，返回 false。

1） 解法一

使用冒泡进行暴力破解，这种方法的时间复杂度为 $O(n^2)$。

```
public bool ContainsDuplicate(int[] nums)
    {
        for (int i = 0; i < nums.Length-1; i++)
        {
            for (int j = i + 1; j < nums.Length; j++)
            {
                if (nums[i] == nums[j])//当有数值相同时就返回 true
                {
                    return true;
                }
            }
        }
        return false;
    }
```

2） 解法二

将数组排序后再进行比较，时间复杂度 O(n)。

```
public bool ContainsDuplicate(int[] nums)
    {
        Array.Sort(nums);//对数组排序
        for (int i = 0; i < nums.Length - 1; i++)
        {
            if (nums[i] == nums[i + 1])//和后一个进行比较
            {
                return true;
            }
        }
        return false;
    }
```

3） 解法三

哈希表是根据关键字而直接进行访问的数据结构，它建立了关键字和存储地址之间的一种直接映射关系，利用哈希表的特点我们可以去判断数组中是否有重复元素。

```
public bool ContainsDuplicate(int[] nums)
    {
        HashSet<int> intSet = new HashSet<int>(nums);//设置数组中每一个数值的哈希值
```

哔哩哔哩次元游戏人生

```
        if (intSet.Count != nums.Length)//哈希表中的元素数，是否等于数组的长度
            return true;//因为一个数值只对应一个哈希值，因此有重复元素时，哈希表中数据会小于数组长度
        else
            return false;
    }
```

2. 数组里有{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}，请随机打乱顺序，生成一个新的数组。

主要是运用洗牌算法，将数组打乱。

1） Fisher－Yates shuffle

```
private void FisherList<T>(ref T[] dataArray)
    {
        List <T> dataList = new List<T>(dataArray);
        List<T> cache = new List<T>();//构建
        while (dataList.Count > 0)
        {
            int randomIndex = UnityEngine.Random.Range(0, dataList.Count);//随机获得一个数组下标
            cache.Add(dataArray[randomIndex]);//将数组添加到 List 中
            dataList.RemoveAt(randomIndex);//将随机下标对应的元素移除出 List
        }
        dataArray = cache.ToArray();//将 List 转换为数组
    }
```

2） Knuth-Durstenfeld Shuffle

```
private void KnuthList<T>(ref  T[] dataArray)
    {
        for (int i = 0; i < dataArray.Length - 1; i++)
        {
            int randomIndex = UnityEngine.Random.Range(i, dataArray.Length);
            T temp = dataArray[randomIndex];
            dataArray[randomIndex] = dataArray[i];
            dataArray[i] = temp;
        }
    }
```

3. 给你一个整数数组 nums，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。（子数组是数组中的一个连续部分）。

1） 解法一——使用贪心算法

当我们从头到尾遍历这个数组的时候，对于数组里的一个整数，它有几种选择呢？它只有两种选择，一个是和之前的数值合并，另一个是以自己作为起始点，新开一个子数组。

如果之前子数组的总体和大于 0 的话，我们认为其对后续结果是有贡献的。这种情况

哔哩哔哩次元游戏人生

下我们将值加入之前的子数组。

如果之前子数组的总体和为 0 或者小于 0 的话，我们认为其对后续结果是没有贡献，甚至是有害的（小于 0 时）。这种情况下我们选择以这个数字开始，另起一个子数组。

时间复杂度 O(n)

空间复杂度 O(1)

```
private int MaxSubArray(int[] nums)
    {
        int sum = 0;//用于计算子数组中的和
        var result = nums[0];//将数组第一个元素作为默认返回结果
        for (int i = 0; i < nums.Length; i++)
        {
            if (sum + nums[i] > nums[i]) //意味着在这个数字之前总和> 0 我们可以使用它，否则我们只使用当前数字
            {
                sum += nums[i];
            }
            else
            {
                sum = nums[i]; //否则我们只使用当前数值，重新开始统计。
            }
            /*可以将前面的部分换成
             * sum = Math.Max(nums[i], sum + nums[i]);
             */

            result = sum > result ? sum : result; //保存当前数字或总和值
        }

        return result;
    }
```

2） 动态规划

若前面元素和大于 0，则将其和当前元素相加。若前面元素和小于 0，就让元素和等于当前元素，重新进行统计。

```
public int MaxSubArray(int[] nums)
    {
        int sum = nums[0];//当前连续子数组的和
        int max = nums[0];//当前和最大的连续子数组
        for (int i = 1; i < nums.Length; i++)
        {
            if (nums[i] > sum + nums[i])//当前连续子数组的和小于当前数，表示之前的连续子数组和小于 0
            {
                sum = nums[i];
            }
            else //当前连续子数组的和大于当前数
```

哔哩哔哩次元游戏人生

```
            {
                sum = sum + nums[i];
            }
            if (sum > max)
            {
                max = sum;
            }
        }
        return max;
    }
```

4. 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。

1） 解法一——暴力法

暴力遍历数组中所有可能，记录下符合的下标的值。其时间复杂度为 O(n^2)，空间复杂度为 O(1)。

```
public int[] TwoSum(int[] nums, int target)
    {
        int[] n = new int[2];
        for (int i = 0; i < nums.Length; i++)
        {
            for (int j = i + 1; j < nums.Length; j++)
            {
                if (target == nums[i] + nums[j])
                {
                    //所得值和目标值相等，就返回下标
                    n[0] = i;
                    n[1] = j;
                }
            }
        }
        return n;
    }
```

2） 解法二——哈希

哈希表可以用空间来换取时间，将查找时间从 O(n)降低为 O(1)。 哈希表正是为此目的而构建的，它支持以近似恒定的时间进行快速查找。这里用“近似”来描述，是因为一旦出现冲突，查找用时可能会退化到 O(n)。 但只要仔细的挑选哈希函数，在哈希表中进行查找的用时应当被摊销为 O(1)。

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速

哔哩哔哩优元游戏人生

度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

给定表 M，存在函数 f(key)，对任意给定的关键字值 key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表 M 为哈希(Hash）表，函数 f(key)为哈希(Hash) 函数。

```
public int[] TwoSum(int[] nums, int target)
    {
        Dictionary<int, int> arrayDictionary = new Dictionary<int, int>();
        for (int i = 0; i < nums.Length; i++)
        {
            if (arrayDictionary.ContainsKey(target - nums[i]))//检查数据字典中是否有该关键字
            {
                return new int[] { arrayDictionary[target - nums[i]], i };
            }
            else
            {
                arrayDictionary[nums[i]] = i;
                //.net4.0 以后就可以直接赋值了，而且当出现相同的键值时会直接替换，反而如果在这里使用 add 方法当如果有重复值时，如果不判断的时候就会报错。
            }
        }
        return new int[] { 0, 0 };
    }
```

5. 给你两个按非递减顺序排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。请你合并 nums2 到 nums1 中，使合并后的数组同样按非递减顺序排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。nums2 的长度为 n。

二、计算机组成原理

1. 写出代码判断一个整数是不是 2 的 N 次方。

1） 解法一——循环除 2

我们知道，2 的幂是一定可以被 2 整除的。我们只需要让其和每一个 2 的幂值比较，直到相等为止，也就是采用穷举法。

```
private void NumberIsTwoPower(int number)
    {
        int i = 0;
        while (i <= number)
        {
            i *= 2;
```

```
                if (i == number)
                {
                    Debug.Log("是 2 的幂次");
                    return;
                }
            }
            Debug.Log("不是 2 的幂次");
            return;
        }
```

2） 解法二——按位与运算判断

其实一个数 n，如果是 2 的幂次方数，则 n 的二进制的补码中一定只有一个 1，那 n&(n-1)就会让 n 的 2 进制补码中的 1 消失，所以 n&(n-1) == 0。

例 1：n 是 2 的幂次方

n　　= 8 => 0000 1000

n-1 = 7 => 0000 0111

例 2：n 不是 2 的幂次方

n　　= 7 => 0000 0111

n-1 = 6 => 0000 0110

根据按位与的运算法则，相同为 1，不同为 0。则，上面两个例子的结果一定为 0。

```
private void NumberIsTwoPower(int number)
    {
        if ((number > 0) && (number & (number - 1)) == 0)
        {
            Debug.Log("是 2 的幂次");
        }
        else
        {
            Debug.Log("不是 2 的幂次");
        }
    }
```

2. 写出下面的程序的打印结果，并解释其目的。

```
int Foo()
{
    int n = 742;
    int count = 0;
    while (n != 0)
    {
        count++;
        n &= (n - 1);
    }
    return count;
}
```

哔哩哔哩次元游戏人生

令：n=11 0101 1000，则

n-1=11 0101 0111。

有按位与运算得 n&(n-1)=11 0101 0000。

程序输出的结果为 6，返回 n 的二进制表达式中，数值位 ‘1’ 的个数。

3. 写出下面的程序的打印结果，并解释其目的。

```
int Bar()
{
    return ((195 & 0xF0) >> 4) | ((195 & 0xF) << 4);
}
```

输出的结果为 60。

1） 原理

三、 递归

1. 斐波那契数（通常用 F(n)表示）形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：F(0) = 0，F(1) = 1，F(n) = F(n - 1)+F(n - 2)，其中 n > 1。现给定 n 请计算 F(n)。

```
private int Fibonacci(int n)
    {
        if (n <= 0)
        {
            return 0;
        }
        if (n <= 2)
        {
            return 1;
        }
        return checked(Fibonacci(n - 2) + Fibonacci(n - 1));
    }
```

2. 已知 28657，46368 是斐波那契数列中的两个相邻数，输出这两个数之前的斐波那契数列。

```
private void FindForwardFibonacci(int max_value, int min_value,ref List<int> fibonacciList)
    {
        int tempMinValue;//创建一个变量用于存储计算前的最小值
        while (max_value != 1)
        {
            tempMinValue = min_value;
            min_value = max_value - tempMinValue;
            max_value = tempMinValue;
            fibonacciList.Add(min_value);
        }
```

哔哩哔哩次元游戏人生

```
}
```

哔哩哔哩次元游戏人生

算法- Algorithm

一、 洗牌算法

1. Fisher－Yates shuffle

Fisher-Yates shuffle 是一种用于生成有限序列的随机排列的算法——简单来说，该算法对序列进行了混洗。该算法有效地将所有元素放入一个表中；通过从表中随机抽取一个元素来不断确定下一个元素，直到没有元素剩余。该算法产生一个无偏的排列：每个排列都是同样可能的。

Fisher-Yates shuffle 是以首先描述它的 Ronald Fisher 和 Frank Yates 的名字命名的，在 Donald Knuth 之后也被称为 Knuth shuffle。Fisher-Yates shuffle 的一种变体，称为 Sattolo 算法，可用于生成长度为 n 的随机循环排列，而不是随机排列。

将集合进行遍历，随机后的每一个元素都会存放在缓存的一个新的集合中。每当一个随机筛选的元素添加到新的集合后，原集合就会将该元素移除。待原集合的元素被移除完毕，再将缓存集合中的元素全部移动到原聚合。该算法的时间复杂度为 O(n) 空间复杂度为 O(n)

```
private void FisherList<T>(List<T> dataList)
    {
        List<T> cache = new List<T>();
        while (dataList.Count > 0)
        {
            int randomIndex = UnityEngine.Random.Range(0,
dataList.Count);
            cache.Add(dataList[randomIndex]);
            dataList.RemoveAt(randomIndex);
        }
        for (int i = 0; i < cache.Count; i++)
        {
            dataList.Add(cache[i]);
        }
    }
```

2. Knuth-Durstenfeld Shuffle

在 Fisher–Yates shuffle 基础上，不采用缓存集合的思路，在集合当中直接进行互换，时间复杂度为 O(n)、空间复杂度为 O(1)。

```
private void KnuthList<T>(List<T> dataList)
{
        for (int i = 0; i < dataList.Count - 1; i++)
        {
            int randomIndex = UnityEngine.Random.Range(i,
```


哔哩哔哩次元游戏人生

```
dataList.Count);
                T temp = dataList[randomIndex];
                dataList[randomIndex] = dataList[i];
                dataList[i] = temp;
            }
    }
```

二、 Johnson－Trotter 全排列算法

从 n 个不同元素中取出 m（m≤n）个元素，按照一定的顺序排成一列，叫做从 n 个元素中取出 m 个元素的一个排列(permutation)。特别地，当 m=n 时，这个排列被称作全排列(all permutation)。Johnson-Trotter 是一种基于最小变换的全排列生成算法。

我们给每一个排列中的每个元素 k 赋予一个方向。我们在所讨论的每个元素上画一个箭头来指出它的方向，像这样$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$。

如果元素 k 的箭头指向一个相邻的较小的元素（元素比箭头方向的相邻元素大），我们说它在这个以箭头标记的排列中是可移动（mobile）的。例如对于排列$\overleftarrow{1}\overleftarrow{2}\overleftarrow{3}$来说，3 相对于 2 是可移动的，而 3 对于 1 则不可移动（因为这时 1 不是 3 箭头方向的相邻元素）。通过使用移动元素这个概念，我们可以给出所谓的 Johnson-Trotter 算法的描述，它也是用来生成排列的。

1. 算法描述

//输入：一个正整数 n
//输出：{1, ... , n}的所有排列的列表
将第一个排列初始化为 1 2 .. n （箭头全向左）
while 存在一个移动元素 do
  求最大的移动元素 k
  把 k 和它箭头指向的相邻元素互换（包括元素值和元素的方向）
  调转所有大于 k 的元素的方向
  将新排列添加到列表中

2. 案例

对于$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$（这是第一个排列），该算法的每一步都会输出一种排列。

① 找出最大的可移动元素，对此进行移动，并记录一种排列。输出$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$。

② 找出最大的可移动元素，对此进行移动，并记录一种排列。输出$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$。

③ 现在 3 已经走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素。输出$\overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$。输出完毕后，最大的可移动元素 2 箭头所指方向的元素是 3，其将变为不可移动。我们需要翻转所有比现可移动元素（也就是 2）大的元素的方向（也就是 3）。翻转后序列为$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$。

④ 找出最大的可移动元素（这里是 3），对此进行移动（每一步都只移动最大的可移动元素），并记录一种排列。输出$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$。

⑤ 找出最大的可移动元素（这里是 3），对此进行移动，并记录一种排列。输出$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$。此时 3 再次走到尽头，但是已经没有下一个最大可移动元素了，所有的排列都被输出。

再来看看$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}$的全排列（其本身是第一个排列）。

① 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出

哔哩哔哩次元游戏人生

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}$。

② 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}$。

③ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$。

④ 现在 4 已经走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素，（下一个最大可移动元素是 3）。对此进行移动，并记录一种排列，输出 $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$。

输出完毕后，我们需要翻转所有比现可移动元素（也就是 3）大的元素的方向（也就是 4）。翻转后序列为 $\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$。

⑤ 现在 4 已经重新成为了最大的可移动元素，对此进行移动，并记录一种排列。输出 $\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}$。

⑥ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}$。

⑦ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}$。

⑧ 现在 4 再次走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素，（下一个最大可移动元素是 3）。对此进行移动，并记录一种排列，输出 $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overrightarrow{4}$。

输出完毕后，我们需要翻转所有比现可移动元素（也就是 3）大的元素的方向（也就是 4）。翻转后序列为 $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}$。

⑨ 现在 4 又再次重新成为了最大的可移动元素，对此进行移动，并记录一种排列。输出 $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}$。

⑩ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}$。

⑪ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$。

⑫ 现在 4 又走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素，（下一个最大可移动元素是 2）。对此进行移动，并记录一种排列，输出 $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$。

输出完毕后，我们需要翻转所有比现可移动元素（也就是 2）大的元素的方向（也就是 4 和 3）。翻转后序列为 $\overrightarrow{4}\ \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$。

⑬ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$。

⑭ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$。

⑮ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$。

⑯ 现在 4 又又走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素，（下一个最大可移动元素是 3）。对此进行移动，并记录一种排列，输出 $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overrightarrow{4}$。

输出完毕后，我们需要翻转所有比现可移动元素（也就是 3）大的元素的方向（也就是 4）。翻转后序列为 $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}$。

⑰ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出

咸鱼能理次元游戏人生

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}$。

⑱ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{2}\ \overleftarrow{4}\ \overrightarrow{3}\ \overleftarrow{1}$。

⑲ 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{4}\ \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$。

⑳ 现在 4 又又又走到了尽头，箭头所指方向已经没有元素，则开始寻找下一个最大可移动元素，（下一个最大可移动元素是 3）。对此进行移动，并记录一种排列，输出$\overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$。输出完毕后，我们需要翻转所有比现可移动元素（也就是 3）大的元素的方向（也就是 4）。翻转后序列为$\overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$。

21 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}\ \overrightarrow{3}$。

22 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}\ \overrightarrow{3}$。

23 找出最大的可移动元素（这里是 4），对此进行移动，并记录一种排列。输出 $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}$。

24 现在已经无法找到任何的可移动元素了，因为所有的元素箭头所指方向，要么没有元素，要么其值比自身大。所有，我们的算法就到此为止。

3. 优势

记要全排列的元素总数为 n，全排列一共 n!种排列，在每次枚举下一个排列时，我们都在上一个排列中寻找最大的可移动项，然后对其进行移动，因此时间复杂度是 $O(n^2!)$，空间复杂度为 O(n)。

4. 代码

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

namespace Algorithm
{
    [Serializable]
    public enum ElementDirection
    {
        Right,
        Left
    }

    [Serializable]
    public class NumberStruct
    {
        public ElementDirection direction;
        public int number;
        public int index;

        //结构体没有无参构造函数
```

哔哩哔哩次元游戏人生

```
        public NumberStruct(int value)
        {
            direction = ElementDirection.Left;
            number = value;
            index = value - 1;
        }

        //重载 > 和 < 运算符
        public static bool operator >(NumberStruct a, NumberStruct b)
        {
            return a.number > b.number;
        }

        public static bool operator <(NumberStruct a, NumberStruct b)
        {
            return a.number < b.number;
        }
    }

    public class JohnsonTrotter : MonoBehaviour
    {
        public int totalNumber = 0;
        [SerializeField] private List<NumberStruct> numberList = new
List<NumberStruct>();
        [SerializeField] private List<int> arrangementList = new List<int>();

        private void Start()
        {
            for (int i = 0; i < totalNumber; i++)
            {
                numberList.Add(new NumberStruct(i + 1));
            }
            arrangementList = JohnsonTrotterAlgorithm(numberList);
        }

        public List<int> JohnsonTrotterAlgorithm(List<NumberStruct> list)
        {
            AddToArrangement(list, ref this.arrangementList);
            int temparyIndex = FindMaxMobileIndex(list);
            Debug.Log(temparyIndex);
            //只要还存在可移动的元素就将继续
            while (temparyIndex != -1)
            {
                SwapElement(ref list, ref temparyIndex,false);
```

哔哩哔哩次元游戏人生

```
        }
        return arrangementList;
    }

    /// <summary>
    ///  检测当前元素是否可移动
    /// </summary>
    /// <param name="list">元素列表</param>
    /// <param name="index">元素索引</param>
    /// <returns></returns>
    private bool CanMobile(List<NumberStruct> list, int index)
    {
        if (list[index].direction == ElementDirection.Right)
        {
            //方向向右时，索引一旦等于总元素个数-1，则表示当前是最后一个元素，则其右边一定没有元素
            if (index < list.Count - 1)
            {
                return list[index] > list[index + 1];//当左侧的元素大于右侧时，该元素为　可移动元素
            }
            else
            {
                return false;
            }
        }
        else
        {
            //方向向左时，索引一旦等于 0，则表示当前是第一个元素，则其左边一定没有元素
            if (index > 0)
            {
                return list[index] > list[index - 1];//当左侧的元素小于右侧时，该元素为　可移动元素
            }
            else
            {
                return false;
            }
        }
    }

    /// <summary>
    ///  查找当前最大的可移动元素的索引下标
```

哔哩哔哩次元游戏人生

```
        /// </summary>
        /// <param name="list">元素列表</param>
        /// <returns>有可移动元素，返回最大的可移动元素的索引下标；没有就返回-1</returns>
        private int FindMaxMobileIndex(List<NumberStruct> list)
        {
            List<NumberStruct> tempary = new List<NumberStruct>();
            for (int index = 0; index < list.Count; index++)
            {
                if (CanMobile(list, index))
                {
                    tempary.Add(list[index]);//将所有的可移动元素添加进临时列表
                }
            }
            if (tempary.Count == 0)//没有任何可移动元素
            {
                return -1;
            }
            else
            {
                int maxIndex = -1;//默认第一个是最大的，这个返回的是原始列表的当前下标
                for (int i = 0; i < tempary.Count - 1; i++)
                {
                    if (tempary[i] < tempary[i + 1])
                    {
                        maxIndex = tempary[i + 1].index;
                    }
                    else
                    {
                        maxIndex = tempary[i].index;
                    }
                }
                return maxIndex;//返回最大的可移动元素的索引下标
            }
        }

        /// <summary>
        ///  将当前排列添加进排列表中
        /// </summary>
        /// <param name="list"></param>
        private void AddToArrangement(List<NumberStruct> list, ref List<int> arrangementList)
        {
```

哔哩哔哩次元游戏人生

```
            string tempary = "";
            for (int i = 0; i < list.Count; i++)
            {
                tempary += list[i].number.ToString();//将 list 中的数值转化为字符串
            }
            arrangementList.Add(int.Parse(tempary));//int.Parse 将数字的字符串表示形式转换为它的等效 32 位有符号整数。
        }

        /// <summary>
        ///  交换 NumberStruct List 中的元素，不是交换里面的值
        /// </summary>
        /// <param name="list">NumberStruct 类型的表</param>
        /// <param name="currentIndex">当前最大可移动元素的索引下标</param>
        private void SwapElement(ref List<NumberStruct> list, ref int currentIndex,bool changDirection)
        {
            switch (list[currentIndex].direction)
            {
                case ElementDirection.Right:
                    {
                        //方向向右且不是最后一个元素
                        if (currentIndex < list.Count - 1)
                        {
                            //方向向右时，临时存储的就是右边的那个元素的值
                            NumberStruct tempary = list[currentIndex];
                            list[currentIndex] = list[currentIndex + 1];
                            list[currentIndex].index = tempary.index;//现在currentIndex 里的值已被改变，我们需要将原本的下标赋予它
                            tempary.index = list[currentIndex + 1].index;//现在将临时的 index 进行修改，令其等于后一个元素的 index
                            list[currentIndex + 1] = tempary;
                            AddToArrangement(list, ref this.arrangementList);
                        }
                        else//当元素走到尽头
                        {
                            //int temparyNumber = -1;
                            if (FindMaxMobileIndex(list) != -1) //我们依旧可以找到一个可移动的元素
                            {
                                //移动一次那个当前最大的可移动元素
                                currentIndex = FindMaxMobileIndex(list);
                                SwapElement(ref list, ref currentIndex, true);
                            }
```

哔哩哔哩次元游戏人生

```
                    else
                    {
                        currentIndex = -1;
                        return;
                    }
                }
                break;
            }
        case ElementDirection.Left:
            {
                Debug.Log("wwwwwwwwwwwwwwwwww");
                if (currentIndex > 0)//方向向左且不是第一个元素
                {
                    //方向向左时，临时存储的就是左边的那个元素的值
                    NumberStruct tempary = list[currentIndex - 1];
                    list[currentIndex - 1] = list[currentIndex];
                    list[currentIndex - 1].index = tempary.index;//现在
currentIndex 里的值已被改变，我们需要将原本的下标赋予它
                    tempary.index = list[currentIndex].index;//现在将临时的
index 进行修改，令其等于前一个元素的 index
                    list[currentIndex] = tempary;
                    AddToArrangement(list, ref this.arrangementList);

                }
                else//当元素走到尽头
                {
                    //int temparyNumber = -1;
                    if (FindMaxMobileIndex(list) != -1) //我们依旧可以找到
一个可移动的元素
                    {
                        //移动一次那个当前最大的可移动元素
                        currentIndex = FindMaxMobileIndex(list);
                        SwapElement(ref list,ref currentIndex, true);
                    }
                    else
                    {
                        currentIndex = -1;
                        return;
                    }
                }
                break;
            }
        }
```

哔哩哔哩次元游戏人生

```
                if (changDirection)
                {
                    ChangeAllGreaterElementDirection(ref list, currentIndex);
                    currentIndex = FindMaxMobileIndex(list);//交换完成后查找下一个最
大的可移动元素索引下标
                }
                else
                {
                    for (int i = 0; i < list.Count; i++)
                    {

Debug.Log("==============================================");
                        Debug.Log(list[i].direction);
                        Debug.Log(list[i].number);
                        Debug.Log(list[i].index);

Debug.Log("==============================================");
                    }
                    currentIndex = FindMaxMobileIndex(list);//交换完成后查找下一个最
大的可移动元素索引下标
                    Debug.Log(currentIndex);
                }
            }

            /// <summary>
            ///
            /// </summary>
            /// <param name="list">NumberStruct 类型的表</param>
            /// <param name="currentIndex">当前的索引下标</param>
            private void ChangeAllGreaterElementDirection(ref List<NumberStruct> list, int
currentIndex)
            {
                for (int i = 0; i < list.Count; i++)
                {
                    if (i == currentIndex || list[i] < list[currentIndex])
                    {
                        continue;//使用 continue 结束本次循环
                    }
                    if (list[i].direction == ElementDirection.Right)
                    {
                        list[i].direction = ElementDirection.Left;
                    }
                    else
                    {
```

哔哩哔哩次元游戏人生

```
                    list[i].direction = ElementDirection.Right;
                }
            }
        }
    }
}
```

三、 排序算法

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序				$O(1)$	不稳定
快速排序	$O(n\log_{2^n})$	$O(n\log_{2^n})$	$O(n^2)$	$O(\log_{2^n})$	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(1)$	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(n)$	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

$O(1)<O(\log_{2^n})<O(n)<O(n\log_{2^n})<O(n^2)<O(n^3)<O(2^n)$

1. 冒泡排序-Bubble Sort

哔哩哔哩次元游龙人生

计算机组成原理

一、 常用进位计数制

1. 二进制

计算机中信息的存储、处理和传送采用的都是二进制，不论是指令还是数据，或是多媒体信息（声音、图形、图像等），都必须采用二进制编码形式才能存入计算机中。

二进制是一种最简单的进位计数制，它只有两个不同的数码：0 和 1，即基数为 2，逢 2 进 1。任意位数的权是 2^i。

任何一个二进制数都可表示为：$(N)_2 = \sum_{i=n}^{-m} K_i * 2^i$

数字：0、1

后缀：B

例：1010B

2. 八进制

数字：0、1、2、3、4、5、6、7

后缀：O 或 Q

例：137.67Q

3. 十进制

在进位计数制中，每个数位所用到的数码符号的个数叫做基数。十进制是人们最熟悉的一种进位计数制，每个数位允许选用 0~9 共 10 个不同数码中的某一个，因此十进制的基数为 10.每个数位计满 10 就向高位进位，即“逢 10 进 1”。

在一个数中，数码在不同的数位上所表示的数值是不同的。每个数码所表示的数值就等于该数码本身乘以一个与它所在数位有关的常数，这个常数叫做权。

例如：十进制数 6543.21，数码 6 所在数位的权威 1000，这一位所代表的数值即为$6 * 10^3 = 6000$，5 所在数位的权为 100，这一位所代表的数值即为$5 * 10^2 = 500$。。。。。。

数字：0、1、2、3、4、5、6、7、8、9

后缀：D

例：1356D

4. 十六进制

十六进制数的基数为 16，逢 16 进 1。大多数计算机都是采用十六进制来描述计算机中的指令和数据的。

任何一个十六进制数可表示为： $(N)_{16} = \sum_{i=n}^{-m} K_i * 16^i$

数字：0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F

后缀：H

例：19EH

5. 推广

一个基数为 R 的 R 机制数可表示为

$$(N)_R = \sum_{i=n}^{-m} K_i * R^i$$

二、 各种进制数之间的转换

1. 二进制数转换为十六进制数

将一个二进制数转换成十六进制数的方法，是将二进制数的整数部分和小数部分分别

哔哩哔哩次元游戏人生

进行转换，即以小数点为界，整数部分从小数点开始往左数，每 4 位分成一组，当最左边的数不足 4 位时，可根据需要，在数的最左边添加若干个 0 以补足 4 位；对于小数部分，从小数点开始往右数，每 4 位分成一组，当最右边的数不足 4 位时，可根据需要，在数的最右边添加若干个 0 以补足 4 位，最终使二进制数的总的位数是 4 的倍数，然后用相应的十六进制数取而代之。

例如：111011.1010011011B=0011 1011.1010 0110 1100B=3B.A6CH。

2. 十六进制数转换为二进制数

要将十六进制数转换成二进制数，只要将 1 位十六进制数写成 4 位二进制数，然后将整数部分最左边的 0 和小数部分最右边的 0 去掉即可。

例如：3B.328H = 0011 1011.0011 0010 1000B = 3B.A6CH。

3. 二进制数转换为十进制数

要将一个二进制数转换成十进制数，只要把二进制数的各位数码与它们的权相乘，再把乘积相加，就得到对应的十进制数，这种方法称为按权展开相加法。

例如：100011.1011B=$1*2^5+1*2^1+1*2^0+1*2^{-1}+1*2^{-3}+1*2^{-4}$=35.6875D。

4. 十进制数转换为二进制数

要将一个十进制数转换成二进制数，通常采用的方法是基数乘除法。这种转换方法是对十进制数的整数部分和小数部分分别进行处理，整数部分用除基取余法，小数部分用乘基取整法，最后将它们拼接起来即可。

三、 数的机器码表示

1. 机器数与真值

二进制数有正负之分，如 N_1 = +0.101101，N_2 = -0.101101，则 N_1 是个正数，N_2 是个负数。机器不能直接把符号+、-表示出来，为了能在计算机中表示正负数，必须引入符号位，即把正负符号也用 1 位二进制数码来表示。把符号位和数值位一起编码来表示相应的数的表示方法包括原码、补码、反码、移码等。

1） 机器数

用二进制数 0 或 1 来表示数的符号，0 表示正号，1 表示负号。小数点隐含在某一固定位置上，不占存储空间。机器数的位数受机器字长的限制，超过机器字长的数值位要舍去。

机器数可分为，无符号数和带符号数。无符号数，其机器字长的所有二进制位均表示数值。带符号数，其第一个二进制位为符号位，其余为数值部分。

2） 真值

真值，即用$\pm|A|$表示的实际数值。

例：8 位机器数为 11011011

若为无符号整数，其真值为 219。

若为带符号整数，且采用源码表示，则最高位为符号位，11011011 表示二进制数，-1011011，其真值为-91。

3） 原码

符号位为 0 表示正数，为 1 表示负数，数值部分用二进制数的绝对值表示的方法称为原码表示法，通常用$[X]_{原}$表示 X 的原码。

例如，要表示+59 和-59 的原码。假设机器数的位数 8 位（机器的字长为 8 位），最高位是符号位，其余 7 位是数值位，那么，+59 和-59 的原码分别表示为

$[+59]_{原}$=0011 1011　　$[-59]_{原}$=1011 1011

j. 注意

0 的原码有两个值，有“正零”和“负零”之分，机器遇到这两种情况都当做 0 处理。

$[+0]_{原}$=0000 0000　　$[-0]_{原}$=1000 0000

k. 定义式

$$[X]_{原}=\begin{cases} X, & 0 \le x < 2^{n-1} \\ 2^{n-1} - X, & 2^{n-1} < x \le 0 \end{cases}$$

l. 真值、机器数、原码互转

例：当 X=+0.1101 时，$[X]_{原}$=0.1101，在机器中表示为0↓1101

当 X=-0.1101 时，$[X]_{原}$=1.1101

当 X=+1110 时，$[X]_{原}$=01110，在机器中表示为 01110↓

当 X=-1110 时，$[X]_{原}$=11110

① 当 X 为“+”时，X_0=0；当 X 为“-”时，X_0=1。

$[X]_{原}=$ $\xleftrightarrow[\text{数值位不变}]{\text{0 变为+，1 变为−}}$ 真值 X。

$[X]_{原}=$ $\xleftrightarrow[\text{数值位不变}]{\text{符号取反位}}$ $[-x]_{原}$。

4） 补码

补码的定义：把某数 X 加上模数 k，称为以 K 为模的 X 的补码。

$[X]_{补}=K+X$

因此，正数的补码的最高位为符号 0，数值部分为该数本身；负数的补码的最高位为符号 1，数值部分为用模减去该数的绝对值。

求一个二进制数的补码的简便方法是：正数的补码与其原码相同；负数的补码是符号位不变，数值位逐位取反（求其反码），然后在最低位加 1。

注意：0 的补码只有一种形式，就是 n 位 0。

a. 真值、补码、原码互转

当 X≥0，则$[X]_{补}=[X]_{原}$=X，且置符号位为 0。

当 X<0，则 X 与$[X]_{补}$互转，置符号位为 1，将 X 的数值位按位取反，末位加 1，即得$[X]_{补}$。若$[X]_{补}$与$[X]_{原}$互转，符号位不变，数值位按位取反，末尾加 1。

哔哩哔哩次元游戏人生

b. $[X]_{补}$与$-[X]_{补}$的关系

计$[X]_{补}$称为机器正数，$-[X]_{补}$称为机器负数。求$-[X]_{补}$，也称为对$[X]_{补}$的求补或变补。

$[X]_{补}\leftrightarrow-[X]_{补}$：找低位第一个 1，该 1 和其后的 0 不变，其他位取反。

补码的移位关系：

右移：符号位不变，数值位右移，空位补与符号位相同的代码。$[X]_{补}$右移一位得到$[\frac{X}{2}]_{补}$。

左移：符号位不变，数值位左移，空位补 0，$[X]_{补}$左移一位得到$[2X]_{补}$。

5） 反码

引入反码的目的是便于求负数的补码。正数的反码与原码相同，负数的反码是符号位不变，数值位逐位取反。

反码实质上是补码的一个特例，区别在于反码的模比补码的模小一个最低位上的 1。

例如：$[+59]_{反}=[+59]_{原}$=0011 1011，而$[-59]_{原}$=1011 1011，因此，[-59]反=1100 0100。

注意：0 的反码也有两个，$[+0]_{反}$=0000 0000，$[-0]_{反}$=1111 1111。

当 X≥0，则$[X]_{反}$=X，且置符号位为 0。

当 X<0，置符号位为 1，将 X 的数值位按位取反，即得$[X]_{反}$。

a. 反码与原码、补码及真值的关系

当 X≥0，则$[X]_{补}=[X]_{原}=[X]_{反}$=X，且置符号位为 0。

当$[X]_{原}$ $\underset{\text{数值位按位取反}}{\overset{\text{符号位不变}}{\leftrightarrow}}$ $[x]_{反}$ $\underset{\text{符号位不变，末位减 1}}{\overset{\text{符号位不变，末位加 1}}{\rightleftharpoons}}$ $[X]_{补}$

$[X]_{反}$ $\underset{\text{数值位按位取反}}{\overset{\text{符号位的 1 变为“–”}}{\leftrightarrow}}$ X

四、 位运算符

1. 与 – &

参加运算的两个数，按二进制位进行“与”运算。

运算规则：只有两个数的二进制同时为 1，结果才为 1，否则为 0。（负数按补码形式参加按位与运算）

即 0 & 0= 0 ，0 & 1= 0，1 & 0= 0， 1 & 1= 1。

例：3 &5 即 00000011 & 00000101 = 00000001 ，所以 3 & 5 的值为 1。

1） “与运算”的特殊用途

a. 清零

如果想将一个单元清零，即使其全部二进制位为 0，只要与一个各位都为零的数值相与，结果为零。

梦想哩哩哩次元游戏人生

b. 取一个数中指定位

比如取数 X=1010 1110 的低 4 位，只需要另找一个数 Y，令 Y 的低 4 位为 1，其余位为 0，即 Y=0000 1111，然后将 X 与 Y 进行按位与运算（X&Y=0000 1110）即可得到 X 的指定位。

例：设 X=10101110，

取 X 的低 4 位，用 X & 0000 1111 = 0000 1110 即可得到；

还可用来取 X 的 2、4、6 位。

c. 判断奇偶

只要根据最末位是 0 还是 1 来决定，为 0 就是偶数，为 1 就是奇数。因此可以用 if ((a & 1) == 0)代替 if (a % 2 == 0)来判断 a 是不是偶数。

2. 或 – |

参加运算的两个数，按二进制位进行“或”运算。

运算规则：参加运算的两个数只要两个数中的一个为 1，结果就为 1。

即 0 | 0= 0 ,1 | 0= 1，0 | 1= 1, 1 | 1= 1。

例：2 | 4 即 00000010 | 00000100 = 00000110，所以 2 | 4 的值为 6。

负数按补码形式参加按位或运算。

1) “或运算”的特殊用途

常用来将一个数据的某些位设置为 1。

比如将数 X=1010 1110 的低 4 位设置为 1，只需要另找一个数 Y，令 Y 的低 4 位为 1，其余位为 0，即 Y=0000 1111，然后将 X 与 Y 进行按位或运算（X|Y=1010 1111）即可得到。

3. 异或 – ^

参加运算的两个数，按二进制位进行“异或”运算。

运算规则：参加运算的两个数，如果两个相应位为“异”（值不同），则该位结果为 1，否则为 0。

即 0 ^ 0=0 ， 0 ^ 1= 1 ， 1 ^ 0= 1 ， 1 ^ 1= 0 。

例： 2 ^ 4 即 00000010 ^ 00000100 =00000110 ，所以 2 ^ 4 的值为 6 。

1) “异或运算”的特殊作用

a. 翻转指定位

比如将数 X=1010 1110 的低 4 位进行翻转，只需要另找一个数 Y，令 Y 的低 4 位为 1，其余位为 0，即 Y=0000 1111，然后将 X 与 Y 进行异或运算（X^Y=1010 0001）即可得到。

例：X=10101110，使 X 低 4 位翻转，用 X ^ 0000 1111 = 1010 0001 即可得到。

哔哩哔哩张元游戏人生

b. 与 0 相异或，保留原值

X ^ 0000 0000 = 1010 1110。

c. 交换两个数

```
void Swap(int &a, int &b)
{
    if (a != b)
    {
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

4. 取反 – ~

参加运算的一个数，按二进制位进行“取反”运算。

运算规则：~1=0； ~0=1；

1） 取反运算符的特殊作用

a. 使一个数的最低位为零

使 a 的最低位为 0，可以表示为：a & ~1。~1 的值为 1111 1111 1111 1110，再按"与"运算，最低位一定为 0。因为" ~"运算符的优先级比算术运算符、关系运算符、逻辑运算符和其他运算符都高。

5. 左移 – <<

左移，相当于乘法。将一个数的各二进制位全部左移若干位（最左边的二进制位丢弃，右边补 0）。

例：a = a << 2 ，a=5。则是将 a 的二进制位左移 2 位，右补 0。

a = 0000 0000 0000 0000 0000 0000 0000 0101

则 a << 2 = 0000 0000 0000 0000 0000 0000 0001 0100

1 << 5，相当于 1 × (2^5) = 32

1 << 0，相当于 1 × (2^0) = 1

因此，对于正数，若左移时舍弃的高位不包含 1，则每左移一位，相当于该数乘以 2。

6. 右移 – >>

右移，相当于除法。将一个数的各二进制位全部右移若干位，正数左补 0，负数左补 1，右边丢弃。

16 >> 3，相当于 16 / (2^3) = 2

哔哩哔哩次元游戏人生

物理

一、 牛顿第二定律

牛顿第二定律表明，处于经典力学范围内的现实物体所获得的加速度大小(Acceleration Size)和这个物体所受到的合外力的大小成正比，而与物体的质量成反比，即下列公式：

$$F = ma = m\frac{d_v}{d_t}$$

在国际单位制中，力 F（Force）的单位为 N（牛），牛顿定律的重要意义表现在两个方面。

① 定量地量度了物体平动惯性的大小

众所周知，一个物体的惯性还表现在该物体受到外力时，是否容易改变速度这一事实上。在相同力的作用下，凡是容易被改变速度的物体，我们就说它的惯性小；凡是不容易被改变速度的物体，我们就说它的惯性大。若是一个相同的外力作用在两个质量分别为 m1 和 m2 的物体上，我们用 a1 和 a 来 2 分别表示它们因为力的作用所产生的加速度的数值，则得到下列公式：

$$\frac{m_1}{m_2} = \frac{a_2}{a_1}$$

即在相同外力的作用下，物体的质量和加速度成反比，质量大的物体所产生的加速度（Acceleration）小。即是说，一个物体的质量越大，要想改变它的速度就越困难，因此可以说，物体的质量是物体惯性大小的量度。

② 概括了力的叠加原理

如果几个力同时作用在一个物体上，则这一物体的加速度等于各个力单独作用时所产生的加速度的矢量和。这一结论称为力的叠加原理。因此，牛顿第二定律中的 F 应代表物体所受的各个力的矢量和，简称合外力。

在直角坐标系中，牛顿第二定律的分量形式：

$$F_x = ma_x = m\frac{d^2{}_x}{d_{t^2}} \quad F_y = ma_y = m\frac{d^2{}_y}{d_{t^2}}$$

在圆周运动和平面曲线运动时，牛顿第二定律的分量形式：

$$F_n = ma_n = m\frac{v^2}{\rho} \quad F_t = ma_t = m\frac{d_v}{d_t}$$

在上式中 Fn 和 Ft 分别表示合外力的法向向量和切向分量，即法向力和切向力；ρ 为质点所在曲线某点的曲率半径。

二、 抛体运动

抛体运动的加速度是重力加速度 g，大小不变、方向始终竖直向下。

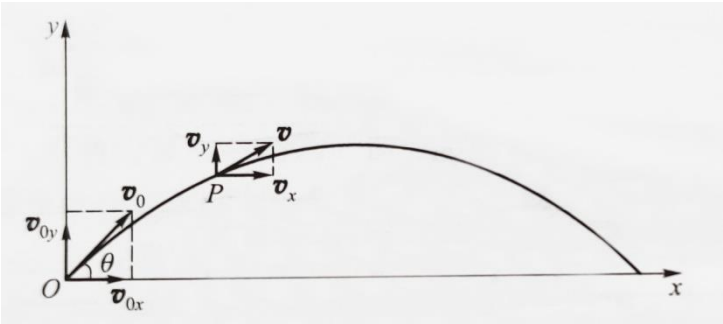


图85. 抛体运动

如图所示，一个物体被抛出的初始位置即为坐标原点，取与 X 轴(Horizontal Axis)垂直向上的方向为 Y 轴(Vertical Axis)的正方向，相对应的取与 X 轴水平向右为正方向建立一个空间直角坐标系，设物体以速度 v_0 沿着与 X 轴夹角 θ 方向抛出，由于加速度等于重力加速度，所以：

$$a_x = 0 \qquad a_y = -g$$

上式表明，抛体运动可以看做两个运动的叠加，即水平方向上的匀速直线运动与竖直方向上的匀变速直线运动的合成。由于在抛出的瞬间，物体位于坐标原点 O，所以初始条件为：

$$t = 0，x_0 = 0，y_0 = 0；v_{0_x} = v_0 \cos\theta，v_{0_y} = v_0 \sin\theta$$

又因为加速度为速度对时间的的一阶导数

$$a_x = \frac{d_{v_x}}{d_t} = 0 \qquad a_y = \frac{d_{v_y}}{d_t} = -g$$

得到垂直方向和水平方向的速度公式

$$v_x = v_0 \cos\theta \qquad v_y = v_0 \sin\theta - gt$$

再因为速度为位移对时间的一阶导数

$$v_x = \frac{d_x}{d_t} = v_0 \cos\theta^{1} \qquad v_y = \frac{d_y}{d_t} = v_0 \sin\theta - gt$$

得到斜抛运动公式

$$x = (v_0 \cos\theta)t \qquad y = (v_0 \sin\theta)t - \frac{1}{2}gt^2$$

将斜抛运动方程中的时间 t 消去，可得轨道方程

$$y = x\tan\theta - \frac{1}{2}gt^2$$

在斜抛运动中，当$v_0 = 0$时，物体上升到最高点，得到物体到达最高点所需的时间为

$$t = \frac{v_0 \sin\theta}{g}$$

而物体从抛出到落地所需的总时间 $t_{总}$等于上升到最高点所需时间 t 的两倍

$$t_{总} = 2t = \frac{2v_0 \sin\theta}{g}$$

斜抛运动的物体由抛出到落地所通过的水平距离，即射程 R 为

$$R = \frac{v_0^2 \sin 2\theta}{g}$$

当抛出方向与 X 轴的夹角 θ 为 0 时，斜抛运动将变为平抛运动；当抛出方向与 X 轴的夹角 θ 为 45 度（即$\frac{\pi}{4}$）时，$R_{max} = \frac{v_0^2}{g}$，此时射程最大；当抛出方向与 X 轴的夹角 θ 为 90 度（即$\frac{\pi}{2}$）时，则运动为竖直上抛运动；当抛出方向与 X 轴的夹角 θ 为-90 度（即$-\frac{\pi}{2}$）时，则运动为竖直下抛运动。

抛体运动在 Unity 中用途广泛，例如玩家的子弹轨迹就可使用斜抛运动方程，使用单独的脚本实现物理特性，不使用 Unity 自带的物理系统，在子弹数量多的情况下可减少性能的开销。

[1] 得到水平方向的速度

哔哩哔哩次元游戏人生

线性代数

一、 点和向量

点(Point)是 n 维空间中的一个位置，在数学中，坐标属于标量，只表示大小，不表示方向。

矢量(Vector,也被称为向量)，通常来说，矢量是指 n 维空间中一种包含了模(magnitude)和方向(direction)的有向线段，它同时具备长度和方向信息，顶点的法线向量、切线向量等都是矢量。

在数学上，常用一条有向线段来表示向量。有向线段的长度表示向量的大小，有向线段的方向表示向量的方向。

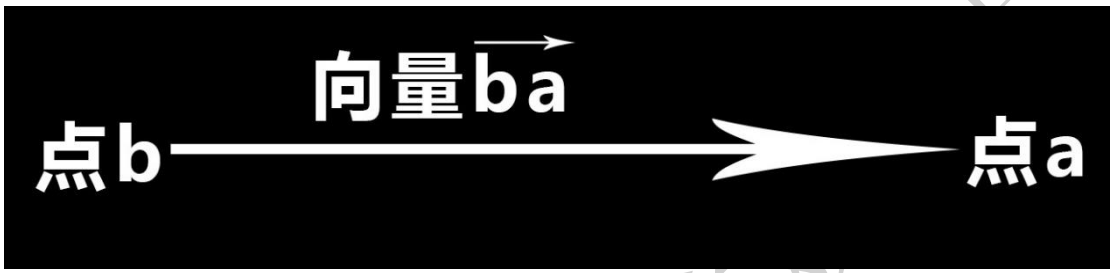


图86. 二维向量

上图表示以点 b 为起点，点 a 为终点的向量$\overrightarrow{\boldsymbol{ba}}$。区分点和矢量之间的不同非常重要，尽管它们在数学表达式上是一样的，都是一串数字而已。如果一定要给它们之间建立一个联系的话，我们可以认为，任何一个点都可以表示成一个从原点出发的矢量。

1. 具体来讲

矢量的模指的是这个矢量的长度。一个矢量的长度可以是任意的非负数。

矢量的方向则描述了这个矢量在空间中的指向。

矢量的表示方法和点类似。我们可以使用 v＝(x,y)来表示二维矢量，用 v＝(x,y,z)来表示三维矢量，用 v＝(x,y,z,w)来表示四维矢量。

为了方便阐述，我们对不同类型的变量在书写和印刷上使用不同的样式。

对于标量，我们使用小写字母来表示，如 a，b，x，y，z，，a 等。

对于矢量，我们使用小写的粗体字母来表示，如 **a**，**b**，**u**，**v** 等。

对于矩阵，我们使用大写的粗体字母来表示，如 **A**，**B**，**S**，**M**，**R** 等。

2. 自由向量

在实际问题中，有些向量与其起点有关(例如质点运动的速度与该质点的位置有关，一个力与该力的作用点的位置有关) ,有些向量与其起点无关。由于一切向量的共性是它们都有大小和方向，因此在数学上我们只研究与起点无关的向量,并称这种向量为自由向量，即只考虑向量的大小和方向而不论它的起点在什么地方。当遇到与起点有关的向量时,可在一般原则下作特别处理。

对于自由向量，如果两个向量$\vec{\boldsymbol{a}}$和$\vec{\boldsymbol{b}}$的大小相等,且方向相同，我们就说向量$\vec{\boldsymbol{a}}$和$\vec{\boldsymbol{b}}$是相等的，记作$\vec{\boldsymbol{a}}=\vec{\boldsymbol{b}}$.这就是说，经过平行移动后能完全重合的向量是相等的。

二、 向量在轴上的投影

三、 数量积、向量积

1. 数量积（点乘）

向量$\vec{\boldsymbol{a}}$和$\vec{\boldsymbol{b}}$的数量积，计作$\vec{\boldsymbol{a}}\cdot\vec{\boldsymbol{b}}$，即$\vec{\boldsymbol{a}}\cdot\vec{\boldsymbol{b}}=|\vec{\boldsymbol{a}}||\vec{\boldsymbol{b}}|\cos\theta$

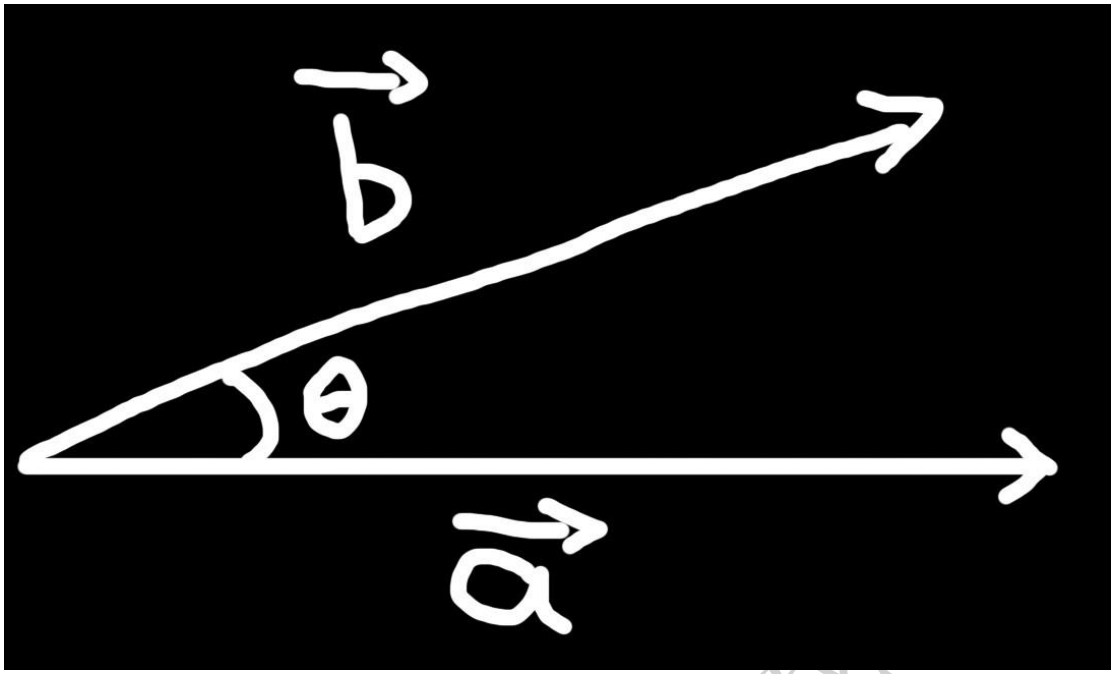


图87.　　向量的数量积

由于$|\vec{b}|\cos\theta = |\vec{b}|\cos(\vec{a},\vec{b})$，当$\vec{a} \neq 0$时是向量$\vec{b}$在向量$\vec{a}$的方向上的投影，用 $\mathrm{Prj_a b}$ 来表示这个投影，便有$\vec{a}\cdot\vec{b} = |\vec{a}|\mathrm{Prj}_a\mathrm{b}$。

当$\vec{b} \neq 0$，$\vec{a}\cdot\vec{b} = |\vec{b}|\mathrm{Prj}_b\mathrm{a}$。

这就是说，两向量的数量积等于其中一个向量的模和另一个向量在这向量的方向上的投影的乘积。

1）代数表示

设$\vec{a} = (X_1, Y_1, Z_1)$，$\vec{b} = (X_2, Y_2, Z_2)$。

则$\vec{a}\cdot\vec{b} = X_1X_2 + Y_1Y_2 + Z_1Z_2$

2）推论

$\vec{a}\cdot\vec{a} = |\vec{a}|^2$（因为向量是同一向量，夹角 θ 为 0）=>$\vec{a}\cdot\vec{a} = |\vec{a}||\vec{a}|\cos 0$。

有两个非零向量$\vec{a}$和$\vec{b}$如果$\vec{a}\cdot\vec{b} = 0$，那么$\vec{a} \perp \vec{b}$；反之，如果$\vec{a} \perp \vec{b}$，那么$\vec{a}\cdot\vec{b} = 0$。（因为零向量与任何向量都垂直，因此前置条件非零向量可不要。）

3）运算规律

① 交换律

$$\vec{a}\cdot\vec{b} = \vec{b}\cdot\vec{a}$$

② 分配律

$$(\vec{a}+\vec{b})\cdot\vec{c} = \vec{a}\cdot\vec{c} + \vec{b}\cdot\vec{c}$$

③ 结合律

$$(\lambda\vec{a})\cdot\vec{b} = \lambda(\vec{a}\cdot\vec{b})\text{（其中λ为常数）}$$

因为$\vec{a}\cdot\vec{b} = |\vec{a}||\vec{b}|\cos\theta$所以当$\vec{a}$和$\vec{b}$都不是零向量时，$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}$

2. 向量积（叉乘）

$\vec{c}$的模$|\vec{c}| = |\vec{a}||\vec{b}|\sin\theta$，其中 θ 为$\vec{a}$和$\vec{b}$之间的夹角。（$\vec{c}$的指向按右手准则，从$\vec{a}$转向$\vec{b}$确定）

向量$\vec{c}$叫做向量$\vec{a}$与$\vec{b}$的向量积，计作$\vec{a}\times\vec{b}$=>$\vec{c} = \vec{a}\times\vec{b}$（向量$\vec{c}$垂直于$\vec{a}$与$\vec{b}$）

哔哩哔哩次元游戏

1） 代数表示

$$\vec{a}\times\vec{b}=\begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

2） 推论

$\vec{a}\times\vec{a}=0$（因为夹角为 0）

对于两个非零向量$\vec{a}$和$\vec{b}$，如果$\vec{a}\times\vec{b}=0$，那么$\vec{a}\parallel\vec{b}$，反之，如果$\vec{a}\parallel\vec{b}$，那么$\vec{a}\times\vec{b}=0$。（由于零向量与任何向量都平行，前置条件可忽略）

3） 运算规律

$\vec{a}\times\vec{b}=-\vec{b}\times\vec{a}$，交换律对向量积不成立。所以我们可以使用叉乘的正负值来判断 a，b 的相对位置，即 b 是处于 a 的顺时针还是逆时针方向。

分配律：$(\vec{a}+\vec{b})\times\vec{c}=\vec{a}\times\vec{c}+\vec{b}\times\vec{c}$

结合律：$(\lambda\vec{a})\times\vec{b}=\vec{a}\times(\lambda\vec{b})=\lambda(\vec{a}\times\vec{b})$（其中λ为常数）

4） 几何意义

由公式$|\vec{c}|=|\vec{a}||\vec{b}|\sin\theta$可以知道，如果以向量$\vec{a}$和$\vec{b}$为边构成一个平行四边形，那么这两个向量外积的模，与这个平行四边形的面积相等。

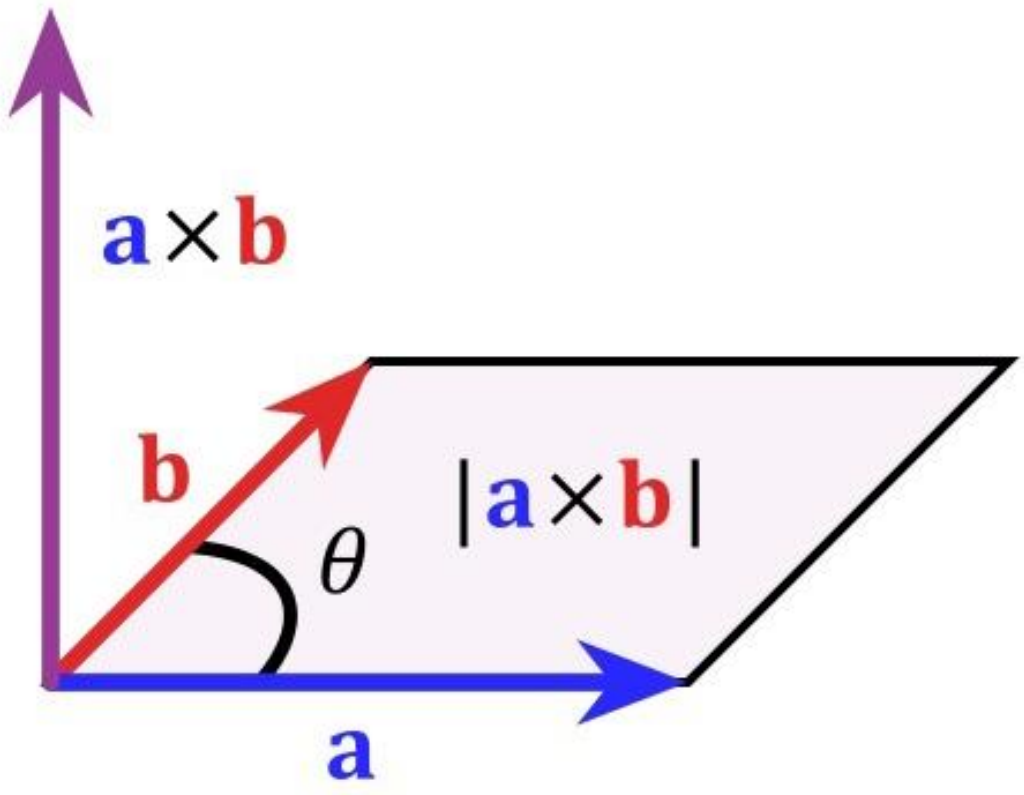


图88. 叉乘的几何意义

四、 向量的内积（点乘）

设有 n 维向量

$\vec{X}=\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$，$\vec{Y}=\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$，令$[\vec{X},\ \vec{Y}]=x_1y_1+x_2y_2+\cdots+x_ny_n$

$[\vec{X}, \vec{Y}]$ 称为向量$\vec{X}$和$\vec{Y}$的内积。

内积是两个向量之间的一种运算，其结果是一个实数。

1. 内积性质

其中$\vec{X}$、$\vec{Y}$、$\vec{Z}$为 n 维向量，λ为实数。

1） 对称性

$[\vec{X}, \vec{Y}]= [\vec{Y}, \vec{X}]$

2） 线性性

$[\lambda\vec{X}, \vec{Y}]= \lambda[\vec{X}, \vec{Y}]= [\vec{X}, \lambda\vec{Y}]$

3） 分配律

$[\vec{X}+\vec{Y}, \vec{Z}]= [\vec{X}, \vec{Z}]+ [\vec{Y}, \vec{Z}]$

4） 正定性

当$\vec{X} = 0$时，$[\vec{X}, \vec{X}]=0$；当$\vec{X} \neq 0$时，$[\vec{X}, \vec{X}]>0$。

$[\vec{X}, \vec{Y}]^2\leq [\vec{X}, \vec{X}][\vec{Y}, \vec{Y}]\Leftrightarrow[\vec{X}, \vec{Y}]^2 \leq ||\vec{X}|| \cdot ||\vec{Y}||$

五、 向量的范数（长度）

令$\|\vec{X}\| = \sqrt{[\vec{X},\vec{X}]} = \sqrt{{x_1}^2 + {x_2}^2 + \cdots + {x_n}^2}$

$\|\vec{X}\|$称为 n 维向量$\vec{X}$的长度（或范数）。

例子：$\vec{\alpha} = (1,3,5)^T$则$\|\vec{\alpha}\| = \sqrt{1^2 + 3^2 + 5^2} = \sqrt{35}$

1. 性质

1） 非负性

当$\vec{X} \neq 0$时，$\|\vec{X}\| > 0$；当$\vec{X} = 0$时，$\|\vec{X}\| = 0$。

2） 齐次性

$\|\lambda\vec{X}\| = \sqrt{[\lambda\vec{X},\lambda\vec{X}]} = \sqrt{\lambda^2[\vec{X},\vec{X}]} = |\lambda|\sqrt{[\vec{X},\vec{X}]} = |\lambda|\|\vec{X}\|$（其中$\lambda$为常数）

若$\|\vec{X}\| = 1$时，则称$\vec{X}$为单位向量。若$\alpha \neq 0$，取$\vec{X} = \frac{\alpha}{\|\alpha\|}$，则$\vec{X}$是一个单位向量。由向量$\vec{\alpha}$得到$\vec{X}$的过程称为把向量$\vec{\alpha}$单位化。

六、 行列式

行列式一定是一个正方形矩阵得到的，即行列式一定是 n 行 n 列。

1. 二阶行列式

1） 几何意义

平行四边形的面积。

2） 计算

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - cd$$

即主对角线相乘减去副对角线相乘。

2. 三阶行列式

1） 几何意义

平行六面体的体积。

哔哩哔哩次元游戏人生

2） 计算

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

3. n 阶行列式

n 阶行列式完全展开后有 n!项。

1） 计算

不同行不同列元素乘积的代数和。

$$(-1)^{t} a_{1p_1}a_{2p_2}a_{3p_3}\cdots a_{np_n}$$

该项前面带何值是看，列这个排列是偶排列还是奇排列。当 $p_1p_2p_3\ldots p_n$ 为偶排列时，该项前面带正号；当 $p_1p_2p_3\ldots p_n$ 为奇排列时，该项前面带负号。

4. 性质

① 行列式与它的转置行列式相等。

② 对换行列式的两行（列），行列式不变。

③ 如果行列式有两行（列）完全相同，则此行列式等于 0。

④ 行列式中如果有两行（列）元素成比例，则此行列式等于 0。

⑤ 行列式某一行（列）中所有元素都乘同一数，等于用数 k 乘此行列式。

⑥ 行列式中某一行（列）的所有元素的公因子可以提到行列式记号外面。

⑦ 把行列式的某一行（列）的各元素乘同一数然后加到另一行（列）对应的元素上去，行列式不变。（某行（列）的 k 倍加到另一行（列），行列式值不变）

⑧ 某行（列）所有元素都是两个数的和，则可把行列式写成两个行列式之和。（n 阶行列式如果每一行（列）都是两个数的和，则应写成 2^n 个 n 阶行列式的和）

$$D = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{i1} + a'_{i1} & a_{i2} + a'_{i2} & \cdots & a_{in} + a'_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

$$= \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} + \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a'_{i1} & a'_{i2} & \cdots & a'_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

5. 特殊的行列式

1） 主对角线上三角与下三角行列式

上三角行列式是对角线下方的元素全为零，下三角行列式是对角线上方的元素全为零！

$$\text{上三角行列式} = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{vmatrix} = a_{11}a_{12}\cdots a_{nn}$$

$$\text{下三角行列式} = \begin{vmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} = a_{11}a_{12}\cdots a_{nn}$$

哔哩哔哩欢乐游戏人生

2） 副对角线上三角与下三角行列式

上三角行列式是对角线下方的元素全为零，下三角行列式是对角线上方的元素全为零！

$$副对角线上三角行列式=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & 0 & \cdots & 0 \end{vmatrix}=(-1)^{\frac{1}{2}n(n-1)}a_{11}a_{12}\cdots a_{nn}$$

$$副对角线下三角行列式=\begin{vmatrix} 0 & 0 & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}=(-1)^{\frac{1}{2}n(n-1)}a_{11}a_{12}\cdots a_{nn}$$

七、 全排列和对换

1. 全排列（排列）

把 n 个不同的元素排成一列，叫做这 n 个元素的全排列。

2. 逆序

一个排列中，如果一个大的数排在一个小的数的前面，就称这两个数构成一个逆序。一个排列中的所有逆序的总数叫做这个排列的逆序数。逆序数为奇数的排列叫做奇排列，逆序数为偶数的排列叫做偶排列。

3. 对换

在排列中，将任意两个元素对调，其余的元素不动，这种作出新排列的叫做对换，将相邻两个元素对换，叫做相邻对换。

1） 定理

① 一个排列中的任意两个元素对换，排列奇偶性改变。

② 奇排列对换成标准排列的对换次数为奇数，偶排列对换成标准排列的对换次数为偶数。

③ 任意一个 n 阶排列可经一系列对换变成自然排列。

八、 矩阵

1. 矩阵的定义

由m × n 个数a_{ij}（$i=1,2,\cdots,m;j=1,2,\cdots,n$）排成的 m 行 n 列的数表

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix}$$ 称为 m 行 n 列矩阵，简称m × n矩阵。

1） 方阵

行数和列数都等于 n 的矩阵称为 n 阶矩阵或 n 阶方阵。阶矩阵 A 也计作A_n。

2） 行向量

只有一行的矩阵$A=[a_1 \quad a_2 \quad \cdots \quad a_n]$，称为行矩阵，又称行向量。

3） 列向量

只有一列的矩阵$B=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_3 \end{bmatrix}$，称为列矩阵，又称列向量。

哔哩哔哩次元游戏人生

4） 同型矩阵

两个矩阵的行数相等、列数也相等时，就称它们是同型矩阵。若两个矩阵是同型矩阵，且对应元素相等，就称矩阵 A 与矩阵 B 相等，计作 A=B。

5） 零矩阵

元素都是零的矩阵称为零矩阵，计作 O。（注意，不同型的零矩阵是不同的）

6） 对角矩阵

除主对角线的元素以外，其余元素都是 0 的矩阵。

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{nn} \end{bmatrix}$$

7） 单位矩阵

主对角线的元素全是 1，其余元素都是 0 的矩阵。

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

8） 上三角和下三角矩阵

上三角矩阵，其主对角线下方的元素全是 0。

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \vdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{mn} \end{bmatrix}，当 i > j 时，a_{ij} = 0。$$

下三角矩阵，其主对角线上方的元素全是 0。

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{12} & a_{22} & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{12} & a_{12} & a_{12} & a_{mn} \end{bmatrix}，当 i < j 时，a_{ij} = 0。$$

9） 对称矩阵与反对称矩阵

b. 对称矩阵

设 A 为 n 阶方阵，如果满足$A^T = \mathrm{A}$，即$a_{ij} = a_{ji}(i, j = 1,2,\cdots,n)$。那么 A 称为对称矩阵，简称对称阵，对称矩阵的特点是，它的元素以对角线为对称轴对应相等。

PS：

$\begin{bmatrix} 1 & 2 & 0 \\ 2 & 3 & -1 \\ 0 & -1 & 5 \end{bmatrix}$是一个对称矩阵。

$\begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix}$是一个对称矩阵。

c. 反对称矩阵

设 A 为 n 阶方阵，如果满足$A^T = -\mathrm{A}$，即$a_{ij} = -a_{ji}(i, j = 1,2,\cdots,n)$。那么 A 称为反对

哔哩哔哩次元游戏人生

称矩阵，反对称矩阵的特点是，它的元素以对角线为对称轴，元素数值对应相反。

PS：

$$\begin{bmatrix} 0 & -1 & 3 \\ 1 & 0 & 2 \\ -3 & -2 & 0 \end{bmatrix}\text{是一个反对称矩阵。}$$

2. 矩阵的计算

1） 矩阵加法

设有两个$m \times n$矩阵$A = (a_{ij})$和 $B = (b_{ij})$，那么矩阵 A 与 B 的和记作 A+B，规定为

$$A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}。$$

注意，只有当两个矩阵是同型矩阵时，这两个矩阵才能进行加法运算。

a. 运算规律

① $A + B = B + A$

② $(A + B) + C = A + (B + C)$

③ $A + (-A) = O$

④ $B = A + (-B)$

b. 例题

① 若$A = \begin{bmatrix} 1 & 3 & -2 \\ 0 & -4 & 5 \end{bmatrix}$，$B = \begin{bmatrix} 8 & 3 & 5 \\ 1 & 2 & -6 \end{bmatrix}$，求 A+B。

解：

$$A + B = \begin{bmatrix} 1 + 8 & 3 + 3 & (-2) + 5 \\ 0 + 1 & (-4) + 2 & 5 + (-6) \end{bmatrix} = \begin{bmatrix} 9 & 6 & 3 \\ 1 & -2 & -1 \end{bmatrix}$$

注意，只有当两个矩阵是同型矩阵时，这两个矩阵才能进行加法运算。

2） 数与矩阵相乘

数λ与矩阵 A 的乘积计作λA 或 Aλ，规定为$\lambda A = A\lambda = \begin{bmatrix} \lambda a_{11} & \lambda a_{12} & \cdots & \lambda a_{1n} \\ \lambda a_{21} & \lambda a_{22} & \cdots & \lambda a_{2n} \\ \vdots & \vdots & & \vdots \\ \lambda a_{m1} & \lambda a_{m2} & \cdots & \lambda a_{mn} \end{bmatrix}$。

（即，数乘以矩阵中的每一个元素）

a. 运算规律

① $(\lambda\mu)A = \lambda(\mu A)$

② $(\lambda + \mu)A = \lambda A + \mu A$

③ $\lambda(A + B) = \lambda A + \lambda B$

哔哩哔哩次元游戏人生

b. 例题

① 若$A=\begin{bmatrix}1 & 3 & -2\\0 & -4 & 5\end{bmatrix}$，$B=\begin{bmatrix}8 & 3 & 5\\1 & 2 & -6\end{bmatrix}$，求 3A。

解：

$$3A=\begin{bmatrix}3*1 & 3*3 & 3*(-2)\\3*0 & 3*(-4) & 3*5\end{bmatrix}=\begin{bmatrix}3 & 9 & -6\\0 & -12 & 15\end{bmatrix}$$

3） 矩阵与矩阵相乘

前面的一个矩阵决定行数，后一个矩阵决定列数。注意，只有当第一个矩阵（左矩阵）的列数，等于第二个矩阵（右矩阵）的行数时，两个矩阵才能相乘。

a. 运算规律

矩阵之间的乘法不满足交换律，特殊情况下除外。

① $(AB)C=A(BC)$ （结合律）

② $\lambda(AB)=(\lambda A)B=A(\lambda B)$

③ $A(B+C)=AB+AC$，$(B+C)A=BA+CA$

④ $AE=EA=A$，单位矩阵在矩阵乘法中的作用类似于数 1。

b. 例题

4） 矩阵的转置

把矩阵 A 的行换成同序数的列，所得到的一个新矩阵，叫做 A 的转置矩阵，计作 A^T。

矩阵$A=\begin{bmatrix}1 & 2 & 0\\3 & -1 & 1\end{bmatrix}$的转置矩阵为$A^T=\begin{bmatrix}1 & 3\\3 & -1\\0 & 1\end{bmatrix}$。

c. 运算规律

矩阵的转置也是一种运算，满足下述的运算规律。

① $(A^T)^T=A$

② $(A+B)^T=A^T+B^T$

③ $(\lambda A)^T=\lambda A^T$

④ $(AB)^T=B^TA^T$

⑤ $(A^T)^{-1}=(A^{-1})^T$

哔哩哔哩次元游戏人生

d. **例题**

5） 方阵的行列式

6） 重点例题

e. **设$A = E - \alpha\alpha^T$，$B = E + 2\alpha\alpha^T$，其中$\alpha = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$，则 AB 为。（矩阵乘法和转置运算）**

3. 矩阵的逆

4. 矩阵的秩

5. 缩放矩阵

1） 二维缩放

缩放变换非常简单，它的目的是增大或者缩小物体的尺寸。假设一个物体以坐标轴原点为缩放中心点，它顶点的最大值是(x,y)。如果我们要求它在 X 方向上缩放 S_x 倍，在 Y 方向上缩放 S_y 倍，进行坐标变换后的坐标为$x' = S_x * x$ 而$y' = S_y * y$。我们也是可以使用矩阵来表示上述的两个公式。

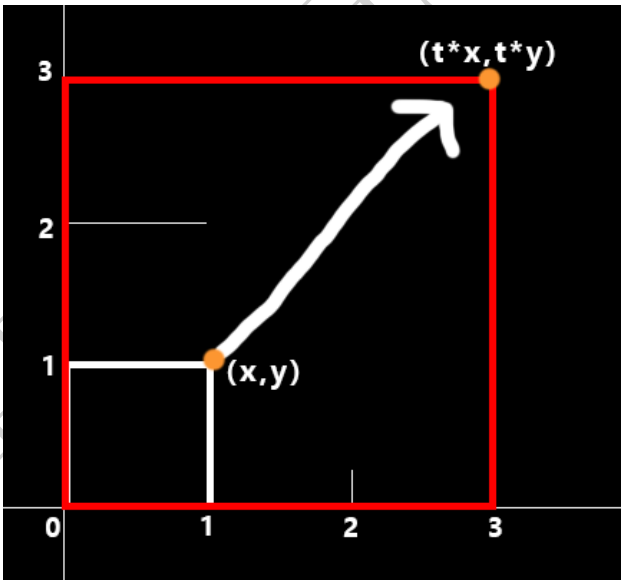


图89. 二维缩放矩阵示意图

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} S_x * x \\ S_y * y \end{bmatrix}$$

2） 三维缩放

由前面的二维缩放矩阵，我们可以很轻易的得知三维缩放矩阵，我们只需要将矩阵进行升维即可。对于一个三维坐标（x, y, z），其矩阵的表达可以写成，$Vector_2$=Matrix * $Vector_1$。

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} S_x * x \\ S_y * y \\ S_z * z \end{bmatrix}$$

6. 平移矩阵

1） 二维平移矩阵

如果我们想要平移空间中的物体，假设在 X 方向上的位移是 tx，Y 方向上的位移是 ty，那么我们就可以得知其 X 坐标和 Y 坐标的变换公式。

$x' = x + t_x$ 而$y' = y + t_y$。

a. 几何意义

2） 三维平移矩阵

7. 旋转矩阵

1） 二维旋转矩阵

假设我们要旋转二维平面中的某一点 Q(x,y)，它与原点的距离为ρ，与 X 轴形成的夹角为α，根据三角函数我们可以将 x 写作$x = \rho * \cos\alpha$，$y = \rho * \sin\alpha$。（极坐标系转化为直角坐标系）假设我们沿着原点逆时针将 Q 旋转θ度，那么旋转后的坐标可以被类似的写作$x' = \rho * \cos(\alpha + \theta)$，$y' = \rho * \sin(\alpha + \theta)$。

由三角函数的和差化积公式，我们可以将上述式子进行展开。

$x' = \rho * \cos\alpha\cos\theta - \rho * \sin\alpha\sin\theta$，$y' = \rho * \sin\alpha\cos\theta + \rho * \cos\alpha\sin\theta$。

最后，再将里面的极坐标系公式转化成熟悉的直角坐标系，就可以将旋转后的坐标，表示成旋转前的坐标与三角函数的乘积之和。

$x' = x\cos\theta - y\sin\theta$，$y' = y\cos\theta + x\sin\theta$。

我们将上面的公式进一步进行简化，简化成矩阵的形式。

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$（这里使用矩阵乘法）

2） 三维旋转矩阵

8. 投影矩阵

9. 二维错切变换

10. 齐次坐标

任何高维矩阵，可以完整的表述出一个低纬度的矩阵。

哔哩哔哩伪元游戏人生

概率论与数理统计

哔哩哔哩次元游戏人生

初等数学

哔哩哔哩次元游戏人生

高等数学

一、 导数

哔哩哔哩次元游戏人生

数据库

哔哩哔哩次元游戏人生

色彩构成

一、 三原色

原色是指不能透过其他颜色的混合调配而得出的“基本色”。以不同比例将原色混合，可以产生出其他的新颜色。以数学的向量空间来解释色彩系统，则原色在空间内可作为一组基底向量，并且能组合出一个“色彩空间”。

1. 光学三原色-RGB

一般来说叠加型的三原色是红色、绿色、蓝色（又称三基色，用于电视机、投影仪等显示设备）。使用这三种原色可以产生其他颜色，例如红色与绿色混合可以产生黄色或橙色，绿色与蓝色混合可以产生青色（Cyan），蓝色与红色混合可以产生紫色或品红色（Magenta）。当这三种原色以等比例叠加在一起时，会变成灰色；若将此三原色的强度均调至最大并且等量重叠时，则会呈现白色。

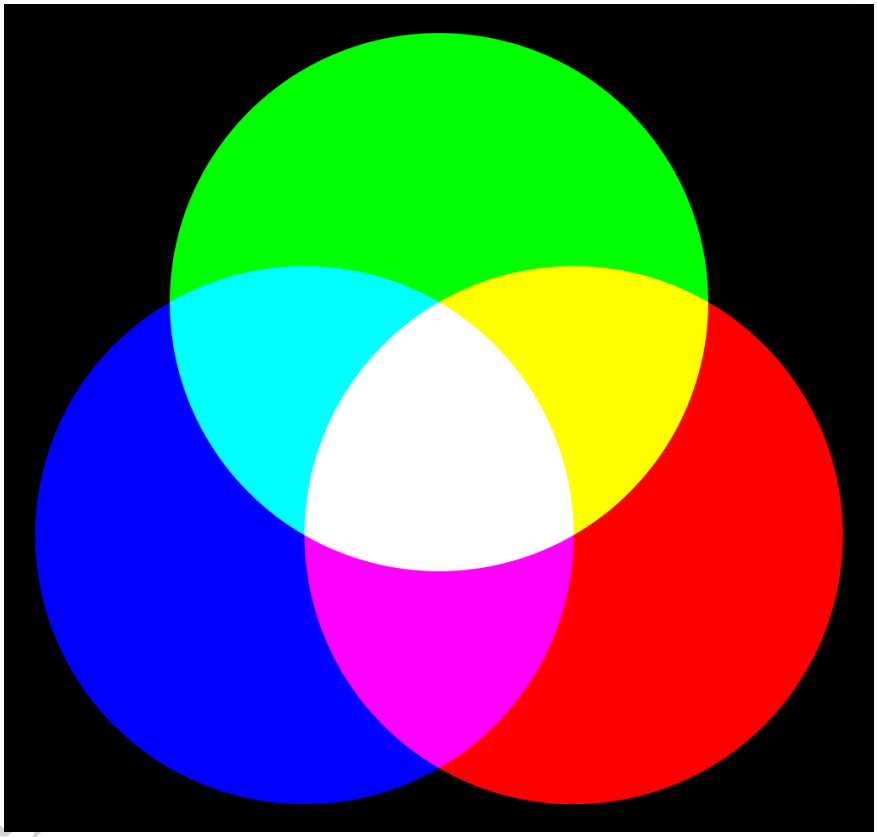


图90. RGB 色彩空间

这套原色系统常被称为“RGB 色彩空间”，亦即由红（R）绿（G）蓝（B）所组合出的色彩系统。

2. 颜料三原色-CMKY

而消减型的三原色是品红色、黄色、青色（用于书本、杂志等的印刷）。减型原色又称为减色原色，一般来说以反射光源或颜料着色时所使用的色彩是属于“消减型”的原色系统。

在传统的颜料着色技术上，通常红、黄、蓝会被视为原色颜料，这种系统较受艺术家的欢迎。当这三种原色混合时可以产生其他颜色，例如黄色与青色混合可以产生绿色，黄色与品红色混合可以产生红色，品红色与青色混合可以产生蓝色。当这三种原色以等比例叠加在一起时，会变成灰色；若将此三原色的饱和度均调至最大并且等量混合时，理论上

会呈现黑色，但实际上由于颜料的原因呈现的是浊褐色。

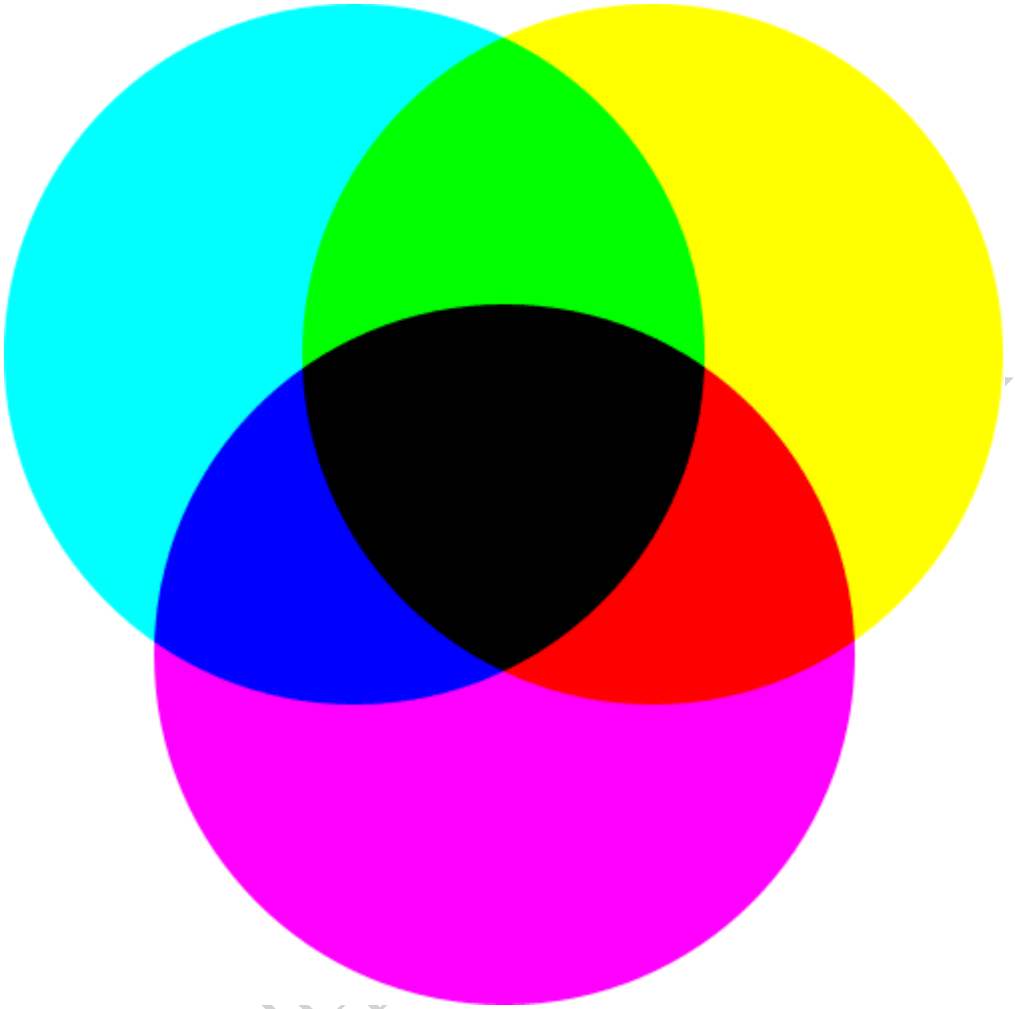


图91.　　CMYK 色彩空间

正因如此，在印刷技术上，人们采用了第四种“原色”——黑色，以弥补三原色之不足。这套原色系统常被称为“CMYK 色彩空间”，亦即由青（C）品红（M）黄（Y）以及黑（K）所组合出的色彩系统。

在消减型系统中，在某颜色中加入白色并不会改变其色相，仅仅减少该色的饱和度。

3. 心理原色

心理原色为蓝色、黄色、红色、绿色、橙色和紫色这六种心理原色加上黑白两色，形塑了色彩感知，理论上还能让人产生不同的心理作用。大脑根据三组色彩和无彩色讯息，处理视网膜上的视觉资讯：红色与绿色、蓝色与橙色、黄色与紫色、白色与黑色。

二、　色相-Hue[1]

色相是指色光由于光频率和波长的不同，而形成地特定色彩的性质，也有人把它叫做色阶，色纯，彩度，色别，色质，色调等。按照太阳光谱的次序把色相排列在一个圆环上，并使其首尾衔接，就称为色相环，再按照相等地色彩差别分为若干主要色相，这就是红，橙，黄，绿，青，紫，等主要色相。

[1] n. 色相、色调

图92.　　色相环

三、　明度

在色度学和色彩理论中，明度，也称为色值或色调，是颜色明度的表示。它是颜色外观模型中的颜色外观参数之一。

明度是指物体反射出来地光波数量地多少，即光波的强度，它决定了颜色的深浅程度。某一色相的颜色，由于反射同一频率光波地数量不同而产生明度差别，例如粉红反射光波较多，其亮度接近浅灰的程度，比大红反射的光波量较少，其亮度接近深灰的明度，他们的色相相同，明度却不同。

还有一个因素影响色彩亮度，人类的正常视觉对不同色光的敏感程度是不一致地，人们对黄、草绿、天蓝色的敏感程度高，所以感觉这些颜色较亮，对红、深蓝视觉敏感度低，所以觉得这些颜色比较暗，人们通常用从白到灰到黑的颜色划成若干明度不同地接替，作为比较其他各种颜色亮度地标准明度色阶。

因此，明度的通俗理解就是，这个颜色加了多少白颜料，加了多少黑颜料。加的白颜料越多，其明度越高；加的黑颜料越多，其明度越暗。

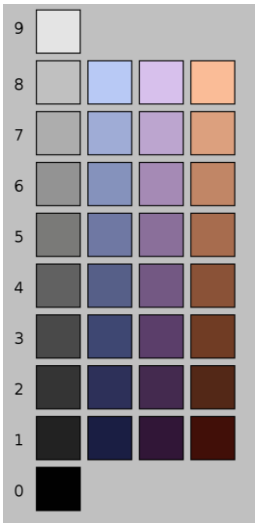


图93.　　在相同的色相中，从上到下展现不同的色彩明度。

哔哩哔哩次元游戏人生

四、 纯度-饱和度

1. 理论

纯度是指物体反射光波频率的纯净程度，单一或混杂的频率决定所产生颜色的鲜明程度。这是一个外来词汇，由于翻译的不同，也有把它翻译为：饱和度、彩度、色纯、色度、色阶，这些词的含义是一样的。当然，有些译法容易混淆，值得商榷。单一频率的色光纯度最高，随着其他频率色光的混杂或增加，纯度也随之减低。物体色越接近光谱中红、橙、黄、绿、青、蓝、紫系列中的某一色相，纯度越高；相反的。颜色纯度越低时，越接近黑、白、灰这些无彩色系列的颜色。

图94. 较高饱和度与较低饱和度

左边的图片拥有较高的饱和度，因此整体的色彩比较鲜艳，右边的图片拥有相对较低的饱和度，因此整体的颜色更加偏向于灰色。一张照片的饱和度越高，画面的“攻击性”就越强，就越能引起人的注意，但是过高的饱和度有时候会让人产生反感的情绪。

图95.　　较高饱和度的照片

一张照片的饱和度越低，画面就越“平和”，就越能给人安静、舒适的视觉感受，但是过低的饱和度有时候会让画面产生不通透感。

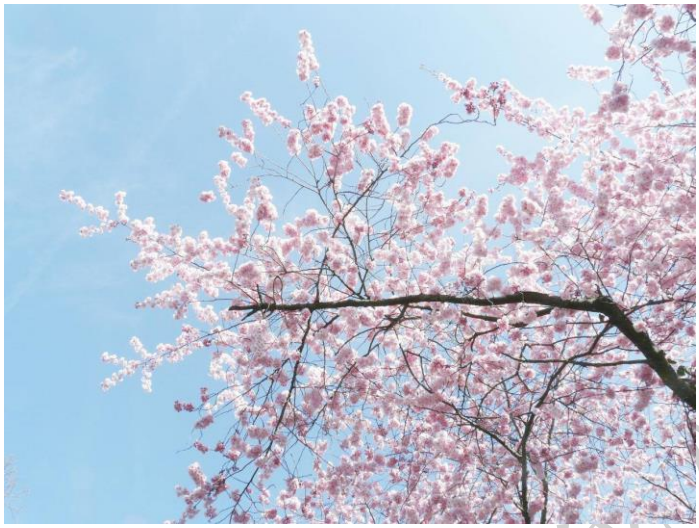


图96.　　较低饱和度的照片

2. 饱和度的应用

1） 增强画面表现力

如果有时候我们感觉画面有些沉闷，我们可以尝试提高画面的饱和度，让画面的色彩更加鲜艳，从而让画面拥有更强的色彩表现力。

例如这张照片的色彩饱和度就较低，因此画面整体看起来比较沉闷，我们提高饱和度就可以让画面更加鲜艳，也更能凸显出色彩的层次。

图97.　　左为原图，右为提高饱和度后的图像

2） 削弱色彩攻击性

画面饱和度过高会让色彩的“攻击性”增强，但是过高的饱和度容易让画面对人产生过度的刺激，有时候过高的饱和度还会让画面显得沉重，或者画面显得比较脏。这时候我们就可以通过降低画面的饱和度削弱色彩的攻击性。

3） 黑白照片

当我们把画面的饱和度将至最低时，画面就会变成黑白照片。

图98.　　将饱和度降至最低

4）局部色彩

我们可以去分别调整不同颜色或不同区域的饱和度，让画面形成高饱和和低饱和并存的局部色彩效果，可以更好的凸显画面主体、增强画面的视觉表现力。

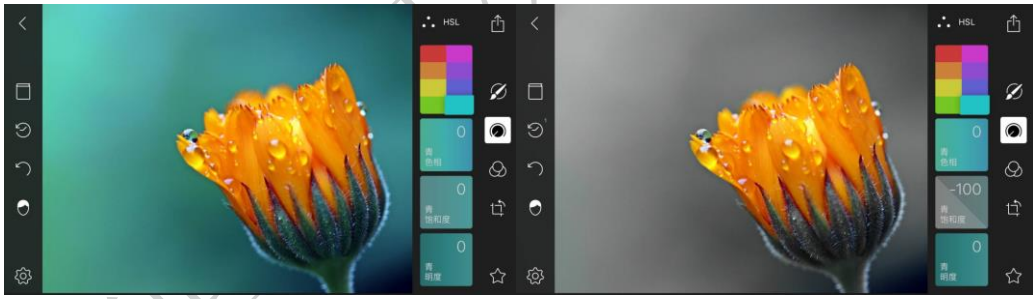


图99.　　左为未应用局部饱和度的原始图像，右为应用局部饱和度的图像

五、色调

色相是最纯净的颜色形式，而色调则是混合色。

色相是最纯净也最明亮的色彩。

Computer Graphics

一、 贝塞尔曲线路径

1. 贝塞尔曲线

在 1962 年，贝塞尔(Bezier)曲线由法国雷诺汽车公司的工程师 Pierre Bezier 对伯恩斯坦多项式进行扩展用于汽车工业辅助设计而产生，之后进行推广而得到广泛应用。贝塞尔曲线的优点是能够直观地表示给定条件与曲线形状的关系，只需要定义曲线的起始坐标点、终止坐标点以及两个相互分离的中间点（控制点）即可完成。用贝塞尔曲线来绘制复杂类型的曲线可以先分段定义多段曲线，然后再将这些曲线段连接起来形成所需要的曲线或曲面形。由于贝塞尔曲线的简单性特点，使得很多图形工作者选择使用它来绘制形状较为复杂的图形或模型。

2. 一次 Bezier 曲线

一次贝塞尔曲线函数中的 t 会经过由点 P0 至点 P1 组成的 B(t)所描述的曲线。当$t = 0.25$时，B(t)即是一条由点 P0 至 P1 路径的四分之一处。就像是由数值 0 至数值 1 的连续数值 t，B(t)描述了一条由点 P0 至点 P1 的直线。高阶的贝塞尔曲线全部基于这个线性插值。

一次贝塞尔曲线的数学表达如下：

$$B(t) = P_0 + (P_1 - P_0)t = (1 - t)P_0 + tP_1, t\epsilon[0,1] \quad （1）$$

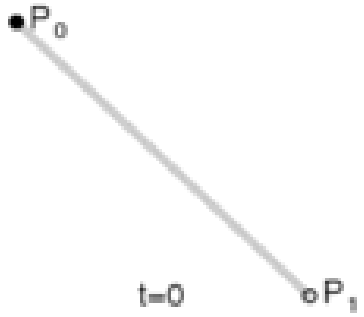


图 3.1 一次贝塞尔曲线 t 在[0,1]之间

3. 二次 Bezier 曲线

二次贝塞尔曲线具有三个控制点。二次的贝塞尔曲线其实是点对点的两个一次贝塞尔曲线的线性插值。由三个点 P_0、P_1 和 P_2 构建的一条二次贝塞尔曲线，其实是两条线性的贝塞尔曲线构成，即是线性贝塞尔曲线 P_0 和 P_1 和线性贝塞尔曲线 P_1 和 P_2。

二次贝塞尔曲线的路径由给定点 P_0、P_1、P_2 控制，这条线由下式给出：

$$B(t) = P_0(1 - t)^2 + 2P_1t(1 - t) + p_2t^2, t \in [0,1] \quad （2）$$

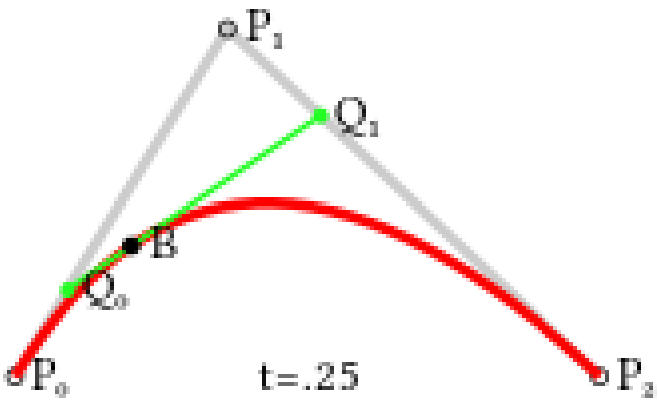


图 3.2 二次贝塞尔曲线的结构 t 在[0,1]之间

4. 三次 Bezier 曲线

三次贝塞尔曲线具有四个控制点。三次的贝塞尔曲线是由点对点的两条二次贝塞尔曲线的线性插值得到。由四个点 P_0、P_1、P_2 和 P_3 组成的三次贝塞尔曲线，其实是由两条二次贝塞尔曲线构成，即是二次贝塞尔曲线 P_0、P_1 和 P_2 和二次贝塞尔曲线 P_1、P_2 和 P_3 得到的线性插值。

三次贝塞尔曲线的路径由给定点 P_0、P_1、P_2、P_3 控制，这条线由下式给出：

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + p_3t^3, t \in [0,1] \quad (3)$$

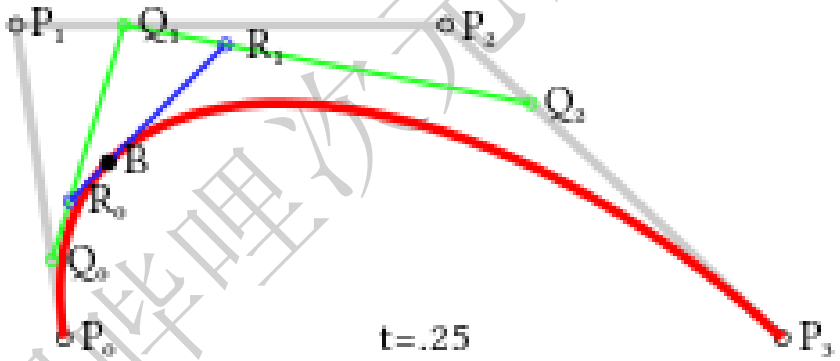


图 3.3 三次贝塞尔曲线的结构 t 在[0,1]之间

5. 高阶 Bezier 曲线

更高阶的贝塞尔曲线总能够通过两个低阶的贝塞尔曲线插值的堆叠获得，通俗的说就是通过对两条低阶的贝塞尔曲线进行线性差值，就可以求得一条高一阶的贝塞尔曲线。

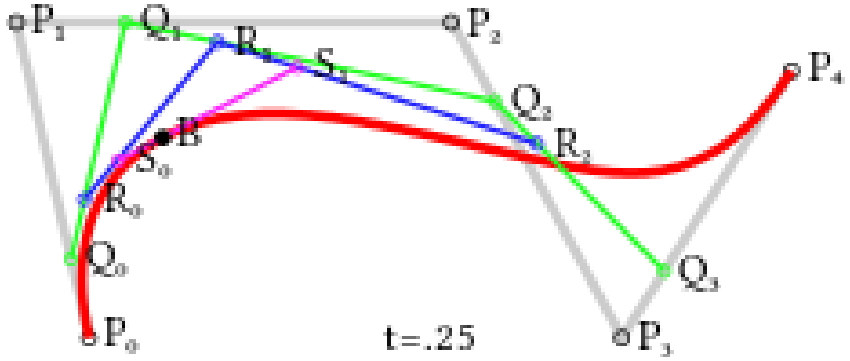


图 3.4 四次贝塞尔曲线的结构 t 在[0,1]之间

二、 程序化网格生成

1. 渲染物体

如果要想在 Unity3D 中渲染出某些内容，可以是渲染从另一个程序导出的 3D 模型，也可以是一个通过程序生成的网格，还可以是 Sprite、UI 或者粒子系统，就应该考虑使用 Unity3D 中的 Mesh。

如果要让 Object 显示 3D 模型，则该 Object 需要包含两个组件。第一个组件是 Mesh Filter（网格过滤器），该组件包含对要显示的网格的引用；第二个组件是 Mesh Renderer（网格渲染器），使用网格渲染器可以配置网格的渲染方式，应该使用何种 Material（材质），是否应该接收阴影等。

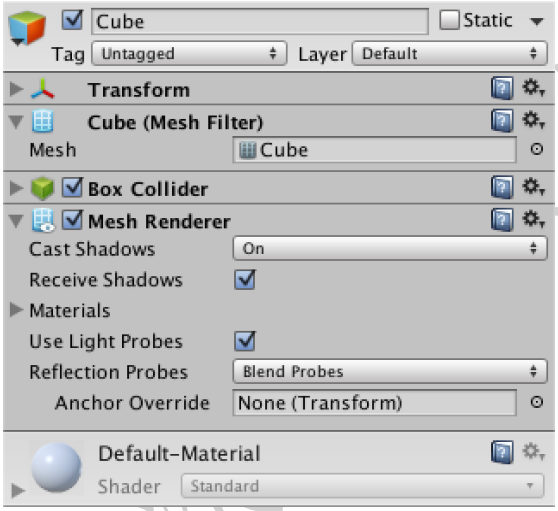


图 3.5 Unity 默认立方体

2. UV

UV 只是 3D 应用程序用于将纹理映射到模型的 2D 坐标，U 坐标表示 2D 纹理的水平轴，V 坐标表示垂直轴。UV 坐标通常位于(0,0)和(1,1)之间，覆盖了整个纹理。根据纹理设置，超出该范围的坐标将被限制或者平铺。

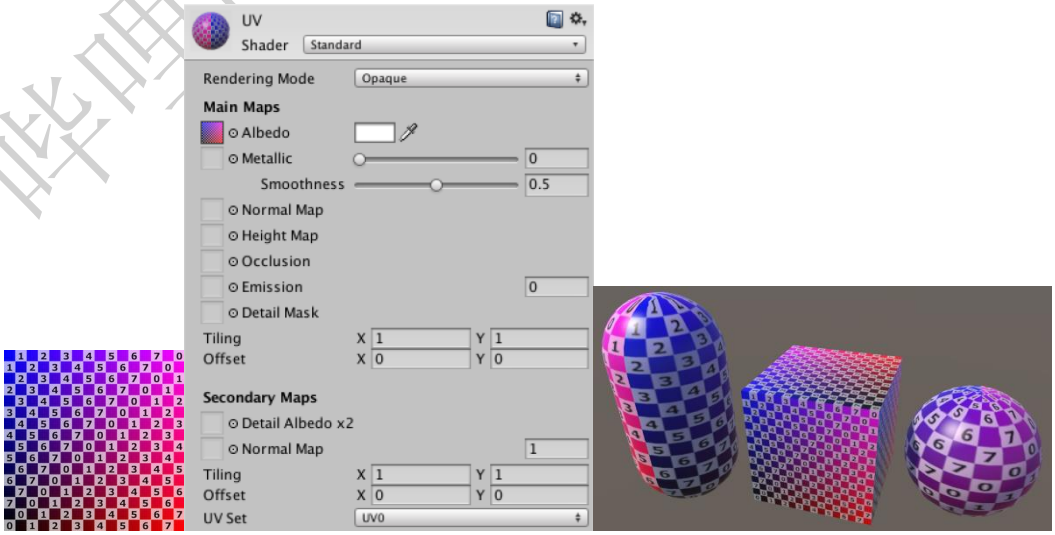


图 3.6 应用于 Unity 的 UV 测试坐标

3. 创建网格

每一个 Mesh 网格都是由若干三角形面片组成的，三角面是通过顶点索引数组定义的。因为每个三角形都具有三个点，因此三个连续的顶点索引就能够描绘出一个三角形。

这个三角形面片从哪一侧可见，是由构成这个三角形面片的顶点索引的方向决定的。在默认情况下，如果顶点索引按照顺时针方向排列，那么该三角形面片就是朝向前方且可见的，而逆时针方向的三角形面片就会被舍弃，因此程序就不需要花费时间去渲染对象的内部。

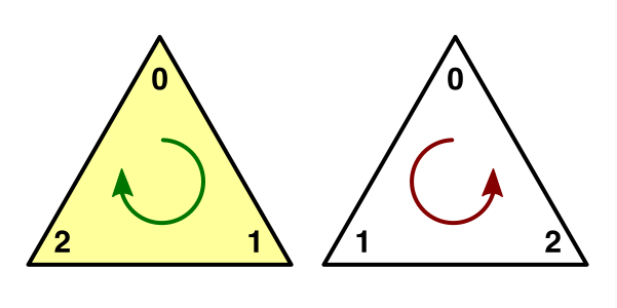


图 3.7 三角形顶点两种遍历情况

因此，要使三角形在 Z 轴向下看时能够显现出来，我们必须更改其顶点的遍历顺序。当我们有了一个三角形，便覆盖了网格第一个图块的一半。

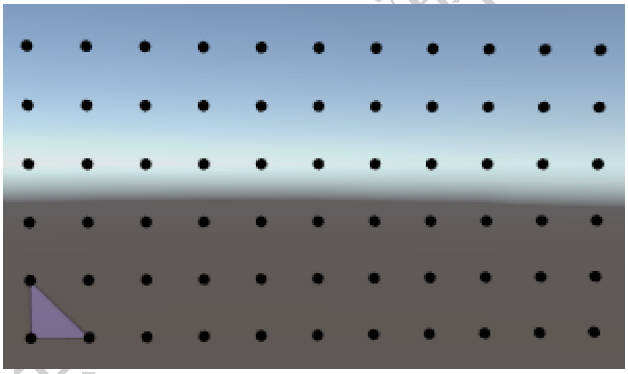
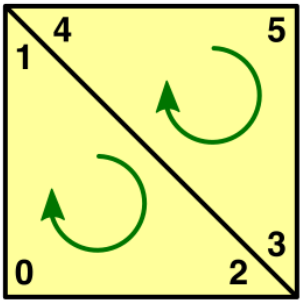


图 3.8 第一个三角形

若想要覆盖整个图块，我们需要两个三角形来组合成矩形。由于这些三角形共享两个顶点，在遍历时只需遍历 4 次即可。然后进行循环生成，我们就可以得到一个完整的矩形网格。



哔哩哔哩 游戏人生

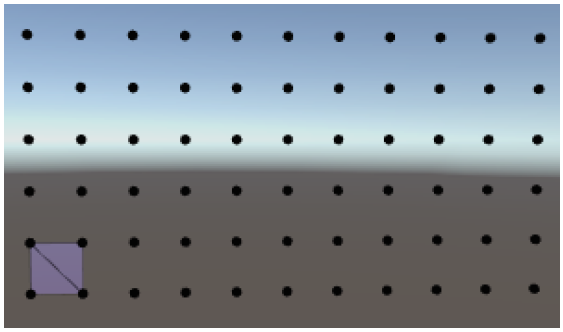


图100.　　图 3.9 由两个三角形组成的四边形　　图 3.10 第一个四边形

4. 生成法线

法线是一个垂直于表面的向量，通常情况下使用的都是单位长度的法线，它们指向表面的外部而不是表面的内部。要想确定一束光线照射到表面的角度（如果有的话）可以使用法线。默认的法线方向是（0，0，1），具体如何使用取决于着色器。由于三角形始终是平坦的，因此不需要提供有关法线的单独信息。但是，提供法线信息可以达到一种欺骗的效果。实际上，顶点没有法线，三角形面片也没有，要想假装拥有的是一个平滑的曲面而不是一堆扁平三角形，可以通过将自定义的法线附加到顶点并在两个三角形之间进行插值。只要不去特别注意网格的鲜明轮廓，这种幻觉就会令人信服。

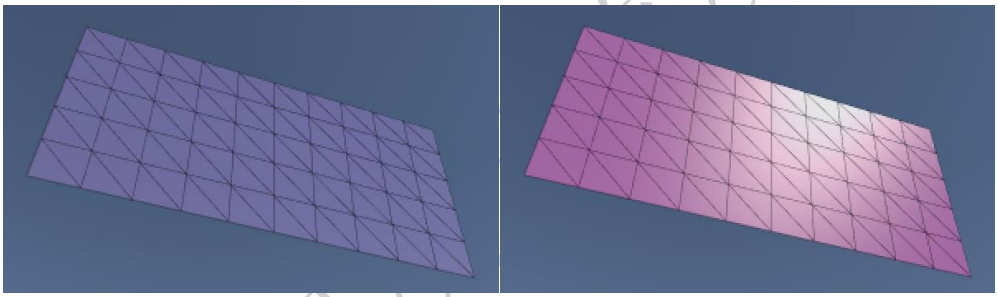


图101.　图 3.11 没有法线与有法线

5. 生成 UV 坐标

我们生成网格后，如果不提供 UV 坐标，则它们将全为 0。要使纹理适合我们的整个网络，只需将顶点的位置除以网格尺寸即可。在进行除法运算得到 UV 坐标时可能会出现 UV 坐标不正确的问题，这是因为使用了整数类型（integer）除以整数类型，这就导致最终所得结果是另一个整数。因为 UV 坐标介于 0 和 1 之间，为了确保坐标正确，我们必须使用浮点数（float）来进行计算。

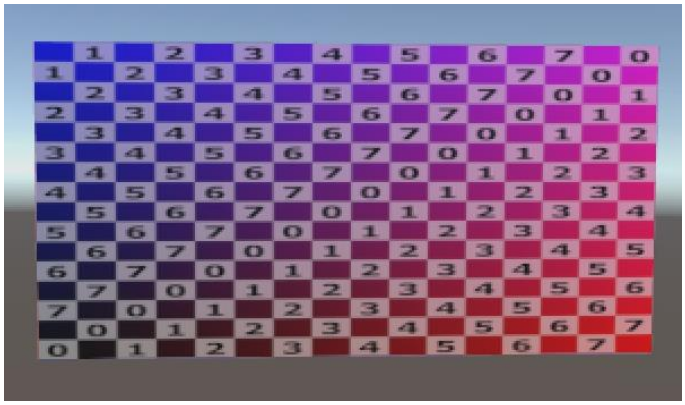


图102.　　图 3.12 有 UV 坐标的网格

Shader Graph

一、 菲涅尔效果-Fresnel Effect Node

菲涅尔效果是根据视角在表面上产生不同反射率的效果，当您接近掠射角时，会反射更多光。菲涅尔效果节点通过计算表面法线和视图方向之间的角度来近似得到，这个角度越大，返回值就越大。这种效果通常用于实现边缘照明，在许多艺术效果中都很常见。

1. 端口

名称	方向	类型	描述
Normal	Input	Vector 3	法线方向。默认绑定到世界空间法线
View Dir	Input	Vector 3	视口方向。默认绑定到世界空间视图方向
Power	Input	Vector 3	幂计算的指数
Out	Output	Float	输出结果

2. 生成的代码示例

以下示例代码表示此节点的一种可能结果。

```
void Unity_FresnelEffect_float(float3 Normal, float3 ViewDir, float Power, out float Out)
{
    Out = pow((1.0 - saturate(dot(normalize(Normal), normalize(ViewDir)))), Power);
}
```

二、 乘节点-Multiply Node

返回输入值 A 乘以输入值 B 的结果。如果两个输入都是向量类型，则输出类型将是与这些输入的评估类型具有相同维度的向量类型。如果两个输入都是矩阵类型，则输出类型将是与这些输入的评估类型具有相同维度的矩阵类型。如果一个输入是向量类型，另一个是矩阵类型，那么输出类型将是一个与输入向量具有相同维度的向量。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic	第一个输入值
B	Input	Dynamic	第二个输入值
Out	Output	Dynamic	幂计算的指数

2. 生成的代码示例

以下示例代码表示该节点的不同可能结果。

1） 向量 * 向量

```
void Unity_Multiply_float4_float4(float4 A, float4 B, out float4 Out)
{
    Out = A * B;
}
```

2） 向量 * 矩阵

```
void Unity_Multiply_float4_float4x4(float4 A, float4x4 B, out float4 Out)
```

哔哩哔哩次元游戏人生

```
{
    Out = mul(A, B);
}
```

3） 矩阵 * 矩阵

```
void Unity_Multiply_float4x4_float4x4(float4x4 A, float4x4 B, out float4x4 Out)
{
    Out = mul(A, B);
}
```

三、 加法节点- Add Node

返回两个输入值 A 和 B 的总和。在 PS 中描述的话，A 和 B 就相当于两个图层，使用加法节点就是将两个图层进行了一个组合。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic	输入的第一个值
B	Input	Dynamic	输入的第二个值
Out	Output	Dynamic	输出值

2. 生成的代码示例

```
void Unity_Add_float4(float4 A, float4 B, out float4 Out)
{
    Out = A + B;
}
```

四、 除法-Divide Node

1. 端口

名称	方向	类型	描述
A	Input	Dynamic	输入的第一个值
B	Input	Dynamic	输入的第二个值
Out	Output	Dynamic	输出值

2. 生成的代码示例

```
void Unity_Divide_float4(float4 A, float4 B, out float4 Out)
{
    Out = A / B;
}
```

五、 正切- Tangent Node

1. 端口

名称	方向	类型	描述
In	Input	Dynamic	输入值
Out	Output	Dynamic	输出值

2. 生成的代码示例

```
void Unity_Tangent_float4(float4 In, out float4 Out)
{
    Out = tan(In);
```

哔哩哔哩次元游戏人生

}

六、 步节点-Step Node

如果输入 In(B)的值大于或等于输入 Edge(A)的值，则返回 1（白色），否则返回 0（黑色）。也就是说 Step 节点输出的内容非黑即白，不存在中间的过渡，我们可以用 Step 节点进行判断，即对图像进行切割。

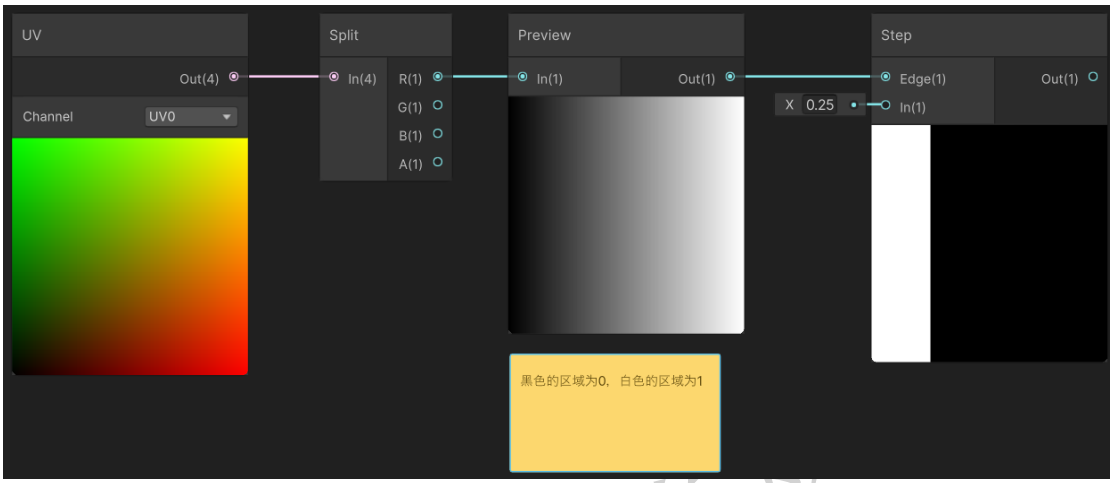


图103. 图 4.1 Step 节点可视化

由上图可以看到，输入的值 B 为 0.25，值 A 为一张从左(0)到右(1)的渐变纹理。因为输入的 A 值左四分之一位置的值都小于等于 B 值，因此在输出的时候都被输出为 1（白色）；右四分之三位置的值都大于 B 值，因此在输出的时候都被输出为 0（黑色）。

1. 端口

名称	方向	类型	描述
Edge	Input	Dynamic	Step 值
In	Input	Dynamic	输入值
Out	Output	Dynamic	输出值

2. 生成的代码示例

```
void Unity_Step_float4(float4 Edge, float4 In, out float4 Out)
{
    Out = step(Edge, In);
}
```

七、 平滑步节点-Smoothstep Node

如果输入 In 的值分别介于输入 Edge1 和 Edge2 的值之间，则返回 0 和 1 之间的平滑 Hermite 插值结果。如果输入 In 的值小于输入 Step1 的值，则返回 0；如果大于输入 Step2 的值，则返回 1 。

该节点类似于 Lerp 节点，但有两个显着差异。首先，用户使用此节点指定范围，返回值介于 0 和 1 之间。这可以看作与 Lerp 节点相反。其次，该节点使用平滑 Hermite 插值而不是线性插值。这意味着插值将从开始逐渐加速并逐渐减慢。这对于创建看起来自然的动画、淡入淡出和其他过渡非常有用。

1. 端口

名称	方向	类型	描述
Edge1	Input	Dynamic Vector	最小 Step 值
Edge2	Input	Dynamic Vector	最大 Step 值

In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Smoothstep_float4(float4 Edge1, float4 Edge2, float4 In, out float4 Out)
{
    Out = smoothstep(Step1, Step2, In);
}
```

八、 极坐标-Polar Coordinates Node

将输入 UV 的值转换为极坐标。在数学中，极坐标系是一个二维坐标系，这个坐标系中的每个点，由该点距离参考点的距离和距离参考方向的角度确定。

由此产生的效果是，UV 输入的 x 通道将转换为与输入中心值指定的点的距离值，而 UV 的 y 通道将转换为围绕该点的旋转角度值。

这些值可以分别通过输入 Radial Scale（径向比例）和 Length Scale（长度比例）的值进行缩放。

1. 端口

名称	方向	类型	描述
UV	Input	Vector 2	输入的 UV 值
Center	Input	Vector 2	中心参考点
Radial Scale	Input	Float	距离值的比例
Length Scale	Input	Float	角度值的刻度
Out	Output	Vector 2	输出值

2. 生成的代码示例

```
void Unity_PolarCoordinates_float(float2 UV, float2 Center, float RadialScale, float LengthScale, out float2 Out)
{
    float2 delta = UV - Center;
    float radius = length(delta) * 2 * RadialScale;
    float angle = atan2(delta.x, delta.y) * 1.0/6.28 * LengthScale;
    Out = float2(radius, angle);
}
```

九、重新映射-Remap Node

根据输入 In Min Max 的 x 和 y 分量之间的输入 In 值的线性插值，返回输入 Out Min Max 的 x 和 y 分量之间的值。

哔哩哔哩次元游戏人生

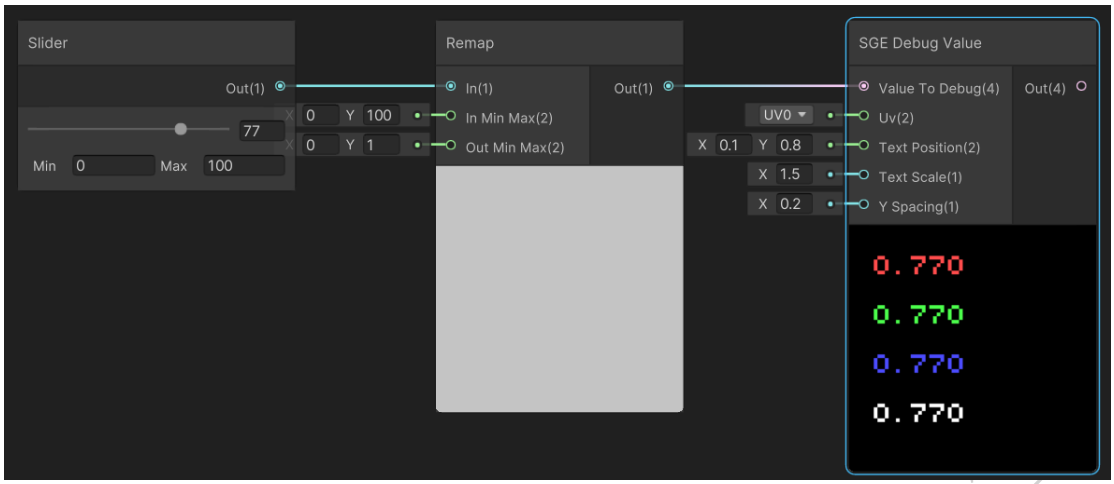


图104.　　图 6.1 Remap Node 示例

当我们为 Remap 节点输入一个值，其最终的输出值为（输入值-输入最小值）*（输出最大值-输出最小值）/（输入最大值-输入最小值）+输出最小值。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
In Min Max	Input	Vector 2	输入插值的最小值和最大值
Out Min Max	Input	Vector 2	输出插值的最小值和最大值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Remap_float4(float4 In, float2 InMinMax, float2 OutMinMax, out float4 Out)
{
    Out = OutMinMax.x + (In - InMinMax.x) * (OutMinMax.y - OutMinMax.x) /
(InMinMax.y - InMinMax.x);
}
```

十、 限制-Clamp Node

Clamp[1]节点的作用就是限制输入值的范围，将输入值限制在设定的最小值和最大值之间。如果在 Shader Graph，创建节点时输入 Limit，出现的节点也是 Clamp 节点，即 Clamp 节点就是起限制的作用。

图105.　　图 6.2 创建节点时输入 Limit

[1] n. 夹具，夹钳；v. （用夹具）夹紧，夹住；

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Min	Input	Dynamic Vector	限制范围最小值
Max	Input	Dynamic Vector	限制范围最大值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Clamp_float4(float4 In, float4 Min, float4 Max, out float4 Out)
{
    Out = clamp(In, Min, Max);
}
```

十一、 时间-Time Node

提供访问各种时间的着色器中的参数。

1. 端口

名称	方向	类型	描述
Time	Output	Float	时间值
Sine Time	Output	Float	正弦时间值
Cosine Time	Output	Float	余弦时间值
Delta Time	Output	Float	当前帧时间值
Smooth Delta	Output	Float	当前帧平滑时间值

Time 端口就如字面意思，代表时间的流逝，Time 端口输出的值会不断增加。

Sine Time 和 Cosine Time 端口，因为分别代表的是正弦时间值和余弦时间值，而正弦和余弦都是一个周期函数，且值都在负 1 到正 1 之间，因此使用这两个端口会具有周期性。

2. 生成的代码示例

```
float Time_Time = _Time.y;
float Time_SineTime = _SinTime.w;
float Time_CosineTime = _CosTime.w;
float Time_DeltaTime = unity_DeltaTime.x;
float Time_SmoothDelta = unity_DeltaTime.z;
```

十二、 瓦片与偏移-Tiling And Offset Node

1. 端口

名称	方向	类型	描述
UV	Input	Vector 2	输入的 UV 值
Tiling	Input	Vector 2	每个通道的瓦片数
Offset	Input	Vector 2	每个通道的偏移量
Out	Output	Vector 2	输出的 UV 值

Tiling 端口是一个二维向量，X 分量代表横向重复多少图，Y 分量代表纵向重复多少图。如下图所示，将分量都设置为了 2，图像的重复数就变成了 4，即 X 分量乘 Y 分量的值。

哔哩哔哩：元游戏人生

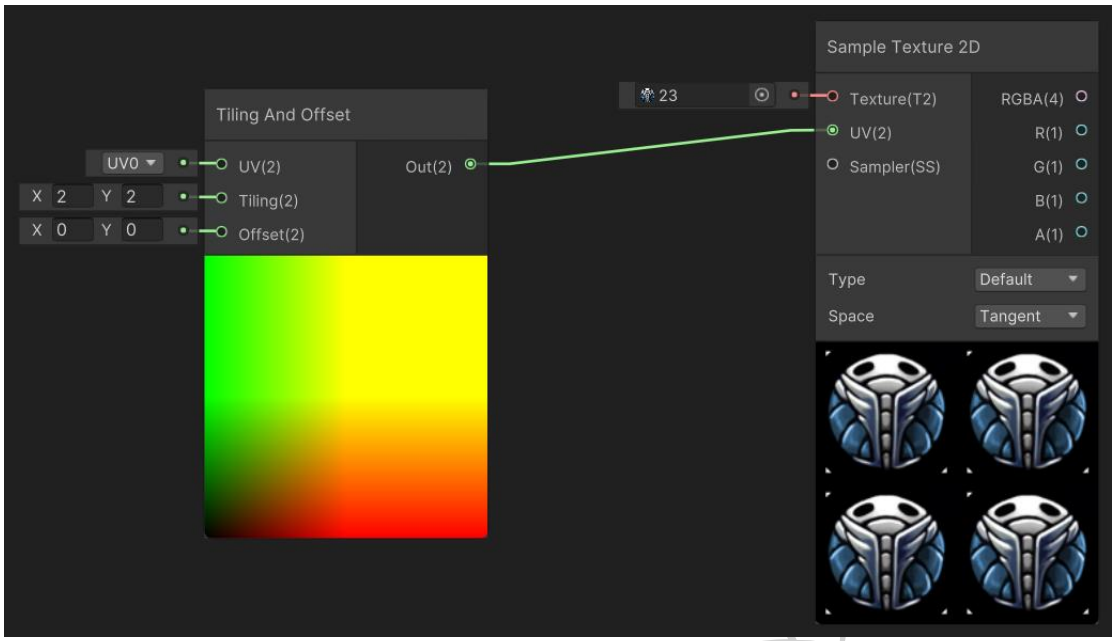


图106.　图 9.1 将 Tiling 的分量都设置为 2

注意，要让图像重复，该图像在 Unity 中的设置必须为 Repeat[2]。

图107.　设置图像的 Wrap Mode

Offset 端口也是一个二维向量，X 分量代表横向移动多少，Y 分量代表纵向移动多少。

2. 生成的代码示例

```
void Unity_TilingAndOffset_float(float2 UV, float2 Tiling, float2 Offset, out float2 Out)
{
    Out = UV * Tiling + Offset;
}
```

十三、取反-Neagte Node

返回输入值的反向符号值。正值会变成负值，负值变为正值。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Negate_float4(float4 In, out float4 Out)
{
    Out = -1 * In;
}
```

[2] v. 重复，重说；复述，跟读（尤指为学习）；
n.（事件的）重演，重现；重播节目；（同类货物的）再次运送；

十四、 简单噪声-Simple Noise Node

1. 端口

名称	方向	类型	描述
UV	Input	Vector 2	输入 UV 值
Scale	Input	Float	噪声比例
Out	Output	Float	输出值

2. 控件

名称	类型	选项	描述
Hash Type	Dropdown（下拉菜单）	Deterministic, LegacySine	选择用于生成噪声的随机数的散列函数

3. 生成的代码示例

```
inline float unity_noise_randomValue (float2 uv)
{
    return frac(sin(dot(uv, float2(12.9898, 78.233)))*43758.5453);
}

inline float unity_noise_interpolate (float a, float b, float t)
{
    return (1.0-t)*a + (t*b);
}

inline float unity_valueNoise (float2 uv)
{
    float2 i = floor(uv);
    float2 f = frac(uv);
    f = f * f * (3.0 - 2.0 * f);

    uv = abs(frac(uv) - 0.5);
    float2 c0 = i + float2(0.0, 0.0);
    float2 c1 = i + float2(1.0, 0.0);
    float2 c2 = i + float2(0.0, 1.0);
    float2 c3 = i + float2(1.0, 1.0);
    float r0 = unity_noise_randomValue(c0);
    float r1 = unity_noise_randomValue(c1);
    float r2 = unity_noise_randomValue(c2);
    float r3 = unity_noise_randomValue(c3);

    float bottomOfGrid = unity_noise_interpolate(r0, r1, f.x);
    float topOfGrid = unity_noise_interpolate(r2, r3, f.x);
    float t = unity_noise_interpolate(bottomOfGrid, topOfGrid, f.y);
    return t;
```

哔哩哔哩次元游戏人生

```
}

void Unity_SimpleNoise_float(float2 UV, float Scale, out float Out)
{
    float t = 0.0;

    float freq = pow(2.0, float(0));
    float amp = pow(0.5, float(3-0));
    t += unity_valueNoise(float2(UV.x*Scale/freq, UV.y*Scale/freq))*amp;

    freq = pow(2.0, float(1));
    amp = pow(0.5, float(3-1));
    t += unity_valueNoise(float2(UV.x*Scale/freq, UV.y*Scale/freq))*amp;

    freq = pow(2.0, float(2));
    amp = pow(0.5, float(3-2));
    t += unity_valueNoise(float2(UV.x*Scale/freq, UV.y*Scale/freq))*amp;

    Out = t;
}
```

十五、 旋转-Rotate Node

1. 端口

名称	方向	类型	描述
UV	Input	Vector 2	输入 UV 值
Center	Input	Vector 2	中心点到 旋转 大约
Rotation	Input	Float	要应用的旋转量
Out	Output	Vector 2	输出 UV 值

2. 控件

名称	类型	选项	描述
Unit	Dropdown（下拉菜单）	Radians（弧度），Degrees（度数）	切换旋转值输入单位

3. 生成的代码示例

1） 弧度

```
void Unity_Rotate_Radians_float(float2 UV, float2 Center, float Rotation, out float2 Out)
{
    UV -= Center;
    float s = sin(Rotation);
    float c = cos(Rotation);
    float2x2 rMatrix = float2x2(c, -s, s, c);
    rMatrix *= 0.5;
    rMatrix += 0.5;
```

哔哩哔哩次元游戏人生

```
        rMatrix = rMatrix * 2 - 1;
        UV.xy = mul(UV.xy, rMatrix);
        UV += Center;
        Out = UV;
    }
```

2） 度数

```
void Unity_Rotate_Degrees_float(float2 UV, float2 Center, float Rotation, out float2 Out)
{
        Rotation = Rotation * (3.1415926f/180.0f);
        UV -= Center;
        float s = sin(Rotation);
        float c = cos(Rotation);
        float2x2 rMatrix = float2x2(c, -s, s, c);
        rMatrix *= 0.5;
        rMatrix += 0.5;
        rMatrix = rMatrix * 2 - 1;
        UV.xy = mul(UV.xy, rMatrix);
        UV += Center;
        Out = UV;
}
```

十六、 幂节点-Power Node

返回值为 A 的 B 次幂，例如输入的 A 值为 2，B 值为 3，则输出的值就为 8。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic Vector	底数
B	Input	Dynamic Vector	指数
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Power_float4(float4 A, float4 B, out float4 Out)
{
        Out = pow(A, B);
}
```

十七、 饱和节点-Saturate Node

返回介于 0 和 1 之间的 In 的值。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Saturate_float4(float4 In, out float4 Out)
{
        Out = saturate(In);
```

哔哩哔哩次元游戏人生

}

十八、 线性插值-Lerp Node

根据 T 值，返回在 A 和 B 之间线性插值的结果。

例如，当输入 T 的值为 0 时，返回值等于输入 A 的值，当它为 1 时，返回值等于输入 B 的值，当它为 0.5 时，返回值是 A 和 B 的中点。

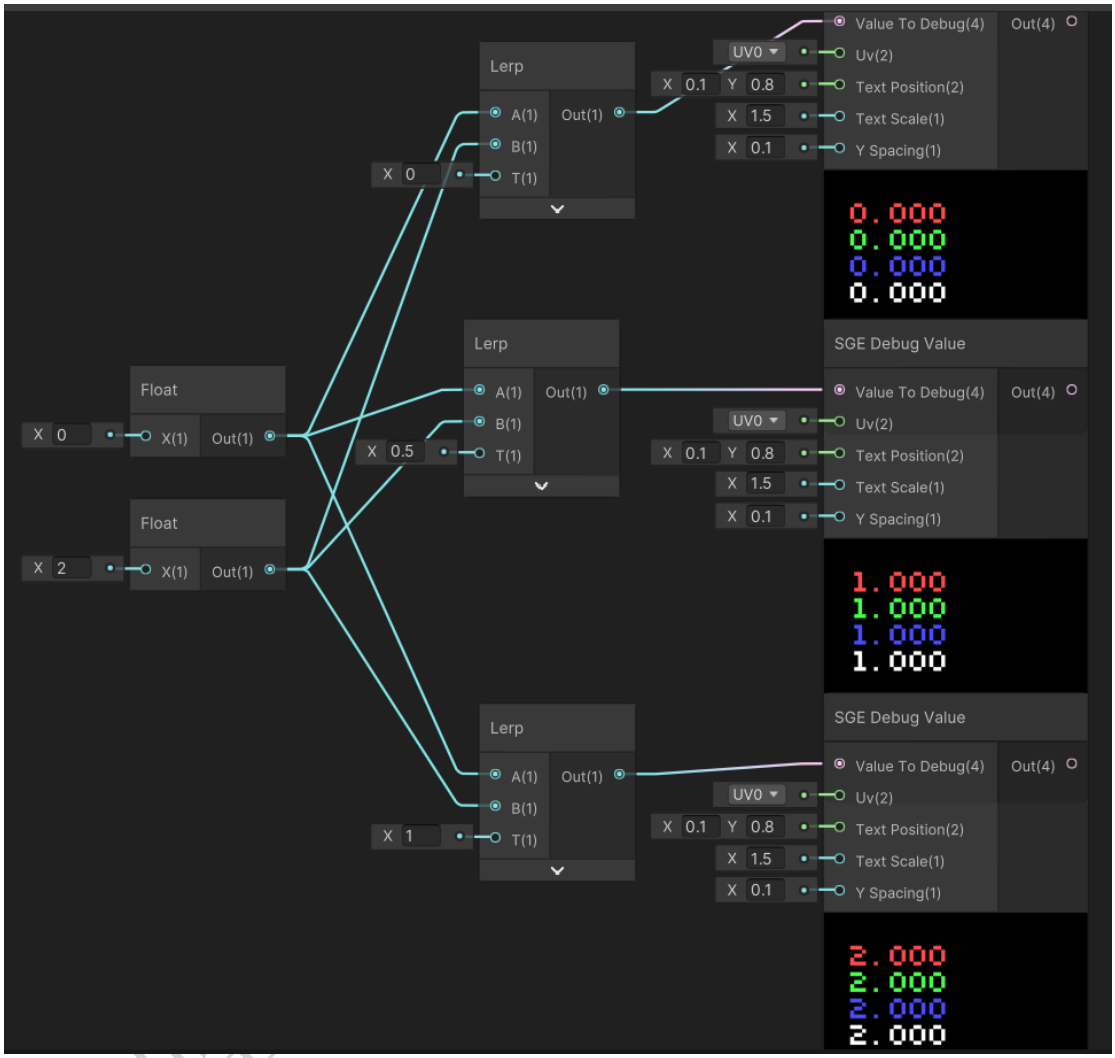


图108. 线性插值图示

1. 端口

名称	方向	类型	描述
A	Input	Dynamic Vector	第一个输入值
B	Input	Dynamic Vector	第二个输入值
T	Input	Dynamic Vector	时间输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Lerp_float4(float4 A, float4 B, float4 T, out float4 Out)
{
    Out = lerp(A, B, T);
}
```

十九、 反插值- Inverse Lerp Node

返回在 A 到 B 的范围内由输入 T 指定的插值线性参数。逆 Lerp 是 Lerp 节点的逆运算。

它可用于根据 Lerp 的输出值，来确定 Lerp 的 T 值是什么。 与 lerp 正好相反，Inverse Lerp 是计算出 T 值在 A 到 B 之间所占比例。

例如，T 值为 1 的 Lerp 在 0 和 2 之间的值为 0.5。 因此，在 0 和 2 之间且 T 值为 0.5 的 Inverse Lerp 的值为 1。

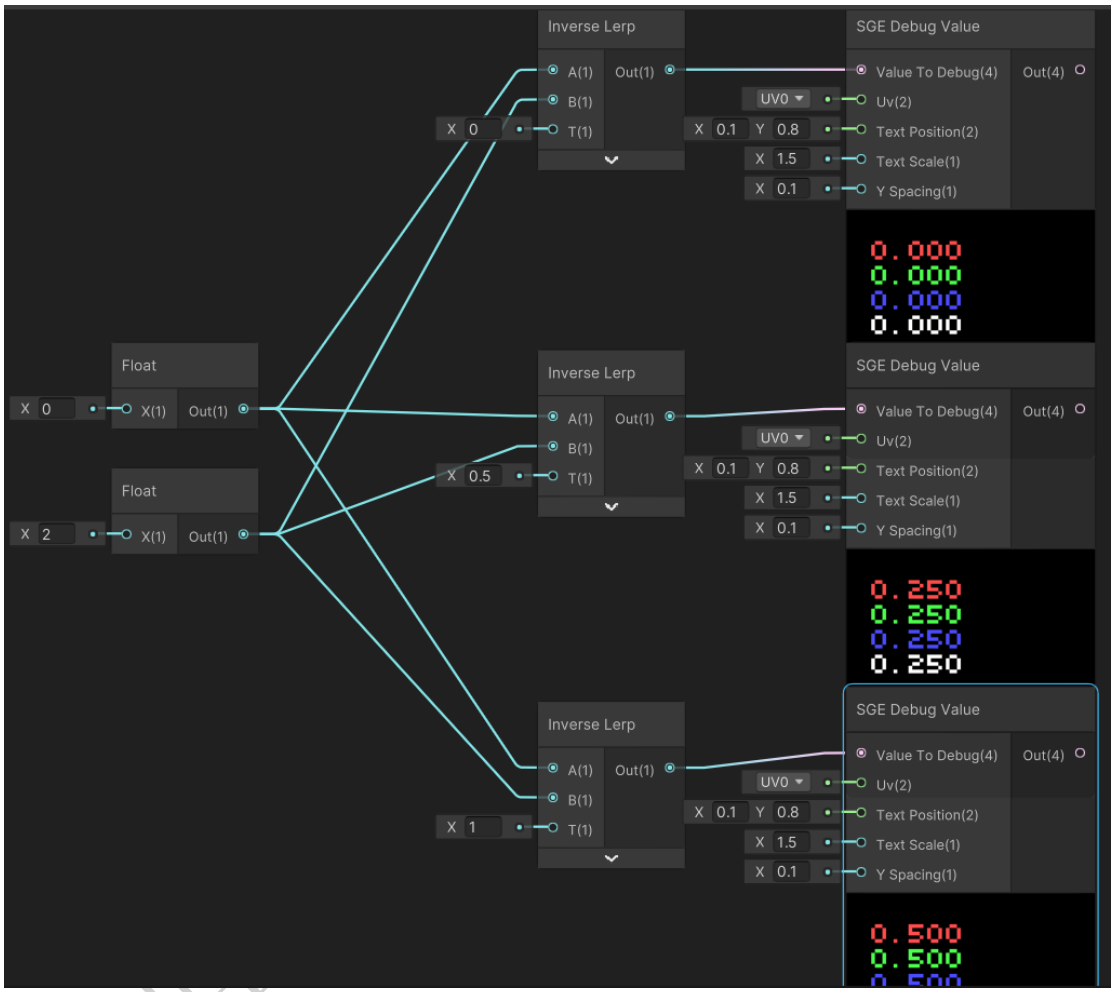


图109. 反插值图示

如图所示，反插值输出的值就是均匀分布的分布函数所对应的值。

当输入的 A 值为 0，B 值为 1，T 为 1 时，T 所占区间长度的比例为二分之一，当 T 为 0.5 时，所占区间长度的比例为四分之一。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic Vector	第一个输入值
B	Input	Dynamic Vector	第二个输入值
T	Input	Dynamic Vector	时间输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

void Unity_InverseLerp_float4(float4 A, float4 B, float4 T, out float4 Out)

```
{
    Out = (T - A)/(B - A);
}
```

二十、 翻书-Flipbook Node

创建一个 翻动提供给输入 UV 的 UV 的书或纹理表动画。工作表上的平铺数量由输入 Width 和 Height 的值定义。当前 Tile 的索引由输入 Tile 的值定义。

该节点可用于创建纹理动画功能，通常用于粒子效果和精灵，方法是向输入 Tile 提供 Time 并输出到 Texture Sampler 的 UV 输入端口。

从 UV 空间的左下角开始，UV 数据通常在 0 到 1 的范围内。这可以通过 UV 预览左下角的黑色值看出。作为翻动书籍通常从左上角开始，默认情况下启用反转 Y 参数。

1. 端口

名称	方向	类型	描述
UV	Input	Vector 2	输入 UV 值
Width	Input	Float	水平瓦片数量
Height	Input	Float	垂直瓦片数量
Tile	Input	Float	当前瓦片索引
Out	Output	Vector 2	输出值

2. 控件

名称	类型	选项	描述
Invert X	Toggle	True, False	如果启用，则图块从右到左迭代
Invert Y	Toggle	True, False	如果启用，则图块从上到下迭代

1） 启用 X 关闭 Y

图片最下层为第一行，最右边为第一列。

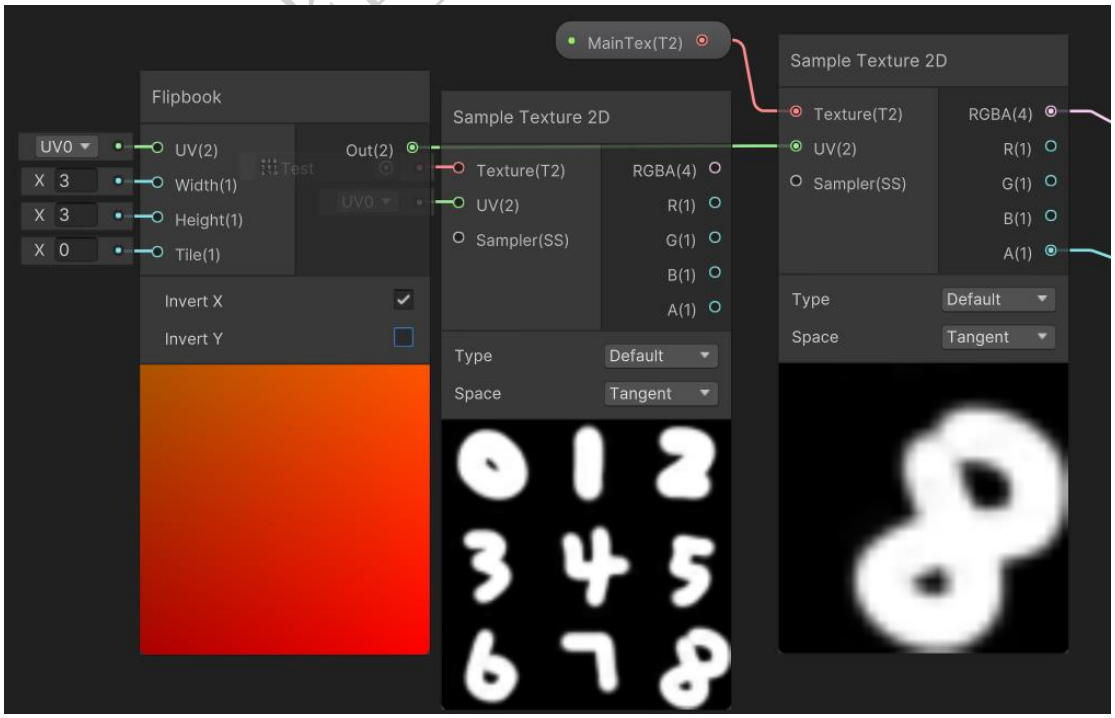


图110.　启用 X 关闭 Y

2） 启用 X 和 Y

图片最上层为第一行，最右边为第一列。

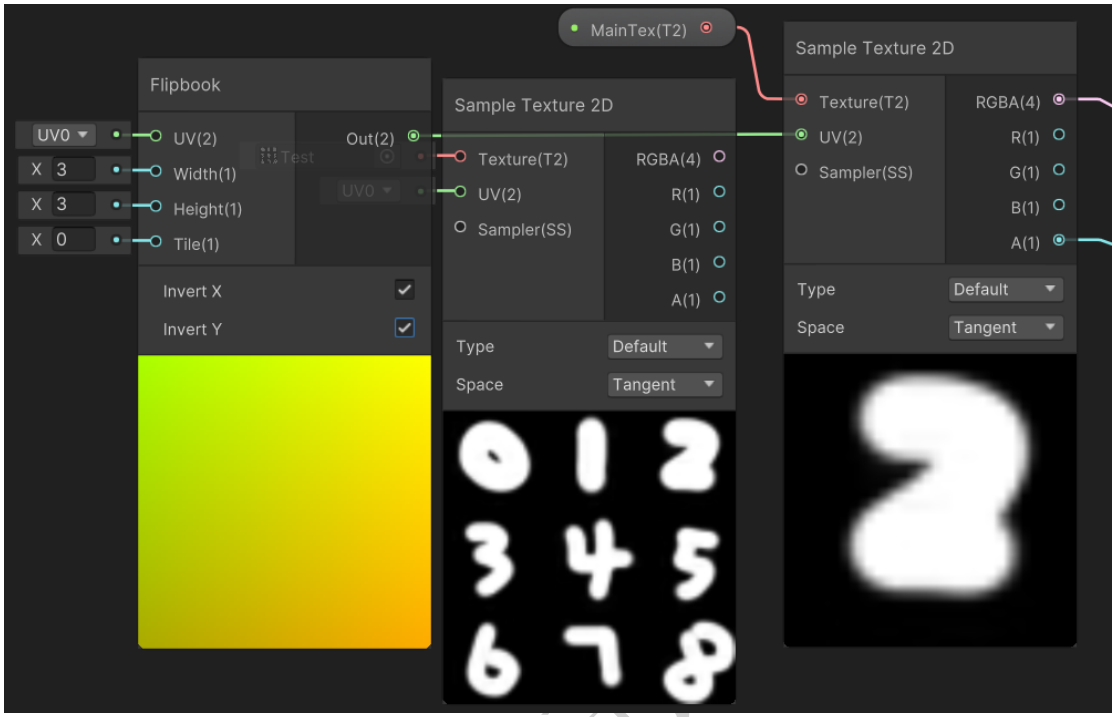


图111.　启用 X 和 Y

3. 生成的代码示例

```
float2 _Flipbook_Invert = float2(FlipX, FlipY);
void Unity_Flipbook_float(float2 UV, float Width, float Height, float Tile, float2 Invert, out float2 Out)
{
    Tile = fmod(Tile, Width * Height);
    float2 tileCount = float2(1.0, 1.0) / float2(Width, Height);
    float tileY = abs(Invert.y * Height - (floor(Tile * tileCount.x) + Invert.y * 1));
    float tileX = abs(Invert.x * Width - ((Tile - Width * floor(Tile * tileCount.x)) + Invert.x * 1));
    Out = (UV + float2(tileX, tileY)) * tileCount;
}
```

二十一、 向下取整- Floor Node

返回小于或等于输入值 In 的最大整数值或整数。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Floor_float4(float4 In, out float4 Out)
{
```

```
    Out = floor(In);
}
```

二十二、向上取整- Ceiling Node

返回大于或等于输入值的最小整数值或整数。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Ceiling_float4(float4 In, out float4 Out)
{
    Out = ceil(In);
}
```

二十三、取余- Modulo Node

返回输入 A 除以输入 B 的余数。

1. 端口 0

名称	方向	类型	描述
A	Input	Dynamic Vector	第一个输入值
B	Input	Dynamic Vector	第二个输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Modulo_float4(float4 A, float4 B, out float4 Out)
{
    Out = fmod(A, B);
}
```

二十四、场景深度-Scene Depth Node

使用输入 UV 提供对当前相机深度缓冲区的访问，该输入 UV 应该是标准化的屏幕坐标。

深度缓冲区访问需要在活动渲染管线上启用深度缓冲区。此过程因渲染管道而异。建议您阅读活动渲染管道的文档以获取有关启用深度缓冲区的信息。如果深度缓冲区不可用，此节点将返回中灰色。

该节点执行的 HLSL 代码是按 Render Pipeline 定义的，不同的 Render Pipelines 可能会产生不同的结果。 希望自定义渲染管道支持此节点，需要明确定义它的行为。 如果未定义，此节点将返回 1（白色）。

此节点只能在片元着色器阶段使用。

1. 端口

名称	方向	类型	描述
UV	Input	Vector 4	标准化屏幕坐标
Out	Output	Float	输出值

2. 深度采样模式

名称	描述

哔哩哔哩次元游戏人生

Linear01	0 到 1 之间的线性深度值
原生-Raw	原始深度值
Eye	深度转换为眼睛空间单位

3. 生成的代码示例

```
void Unity_SceneDepth_Raw_float(float4 UV, out float Out)
{
    Out = SHADERGRAPH_SAMPLE_SCENE_DEPTH(UV);
}
```

二十五、One Minus Node

返回 1 减去输入值的结果。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_OneMinus_float4(float4 In, out float4 Out)
{
    Out = 1 - In;
}
```

二十六、位置-Position Node

提供对网格顶点或片段位置的访问，取决于节点所属的图形部分的有效着色器阶段。使用控件的下拉菜单选择输出值的坐标空间。

1. 端口

名称	方向	类型	描述
Out	Output	Dynamic Vector	输出网格顶点/片段的位置。

2. 控件

名称	类型	选项	描述
Space	下拉菜单	Object, View, World, Tangent, Absolute World	选择对应的坐标空间进行位置输出。

1） World and Absolute World

位置节点为我们提供了，World 和 Absolute World 空间这两个下拉选项。绝对世界(Absolute World)选项总是返回所有可编写脚本的渲染管线场景中对象的绝对世界位置。World 选项返回选定的 Scriptable Render Pipeline 的默认世界空间。

高清渲染管线(High Definition Render Pipeline)使用相管相机(Camera Relative)作为其默认世界空间。

通用渲染管线(Universal Render Pipeline)使用绝对世界作为其默认世界空间。

二十七、绝对值-Absolute Node

返回输入值的绝对值，如果输入值为负则将其变为正数，输入值为正则保持不变。

哔哩哔哩次元游戏人生

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Absolute_float4(float4 In, out float4 Out)
{
    Out = abs(In);
}
```

二十八、变换-Transform Node

将输入值（In）从一个坐标空间转移到另一个坐标空间。

1. 端口

名称	方向	类型	描述
In	Input	Vector 3	输入值
Out	Output	Vector 3	输出值

2. 控件

名称	类型	选项	描述
From	下拉菜单	Object, View, World, Tangent, Absolute World	选择要转换的空间
To	下拉菜单	Object, View, World, Tangent, Absolute World	选择要转换的空间

3. World and Absolute World

世界空间选项使用 Scriptable Render Pipeline 默认世界空间来转换位置坐标。绝对世界空间选项使用绝对世界空间来转换所有 Scriptable Render Pipelines 中的位置坐标。

如果使用变换节点转换不用于位置值的坐标空间，Unity 建议您使用 World space 选项。对不代表位置的值使用绝对世界可能会导致意外行为。

4. 生成的代码示例

1） World -> World

```
float3 _Transform_Out = In;
```

2） World -> Object

```
float3 _Transform_Out = TransformWorldToObject(In);
```

3） World -> Tangent

```
float3x3 tangentTransform_World = float3x3(IN.WorldSpaceTangent,
IN.WorldSpaceBiTangent, IN.WorldSpaceNormal);
float3 _Transform_Out = TransformWorldToTangent(In, tangentTransform_World);
```

4） World -> View

```
float3 _Transform_Out = TransformWorldToView(In)
```

5） World -> Absolute World

```
float3 _Transform_Out = GetAbsolutePositionWS(In);
```


哔哩哔哩次元游戏人生

6） Object -> World

float3 _Transform_Out = TransformObjectToWorld(In);

7） Object -> Object

float3 _Transform_Out = In;

8） Object -> Tangent

float3x3 tangentTransform_World = float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal);

float3 _Transform_Out = TransformWorldToTangent(TransformObjectToWorld(In), tangentTransform_World);

9） Object -> View

float3 _Transform_Out = TransformWorldToView(TransformObjectToWorld(In));

10）Object -> Absolute World

float3 _Transform_Out = GetAbsolutePositionWS(TransformObjectToWorld(In));

11）Tangent -> World

float3x3 transposeTangent = transpose(float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal));

float3 _Transform_Out = mul(In, transposeTangent).xyz;

12）Tangent -> Object

float3x3 transposeTangent = transpose(float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal));

float3 _Transform_Out = TransformWorldToObject(mul(In, transposeTangent).xyz);

13）Tangent -> Tangent

float3 _Transform_Out = In;

14）Tangent -> View

float3x3 transposeTangent = transpose(float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal));

float3 _Transform_Out = TransformWorldToView(mul(In, transposeTangent).xyz);

15）Tangent -> Absolute World

float3x3 transposeTangent = transpose(float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal));

float3 _Transform_Out = GetAbsolutePositionWS(mul(In, transposeTangent)).xyz;

16）View -> World

float3 _Transform_Out = mul(UNITY_MATRIX_I_V, float4(In, 1)).xyz;

17）View -> Object

float3 _Transform_Out = TransformWorldToObject(mul(UNITY_MATRIX_I_V, float4(In, 1)).xyz);

18）View -> Tangent

float3x3 tangentTransform_World = float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal);

float3 _Transform_Out = TransformWorldToTangent(mul(UNITY_MATRIX_I_V, float4(In, 1)).xyz, tangentTransform_World);

哔哩哔哩次元游戏人生

19) View -> View

float3 _Transform_Out = In;

20) View -> Absolute World

float3 _Transform_Out = GetAbsolutePositionWS(mul(UNITY_MATRIX_I_V, float4(In, 1))).xyz;

21) Absolute World -> World

float3 _Transform_Out = GetCameraRelativePositionWS(In);

22) Absolute World -> Object

float3 _Transform_Out = TransformWorldToObject(In);

23) Absolute World -> Tangent

float3x3 tangentTransform_World = float3x3(IN.WorldSpaceTangent, IN.WorldSpaceBiTangent, IN.WorldSpaceNormal);

float3 _Transform_Out = TransformWorldToTangent(In, tangentTransform_World);

24) Absolute World -> View

float3 _Transform_Out = TransformWorldToView(In)

25) Absolute World -> Absolute World

float3 _Transform_Out = In;

二十九、标准化-Normalize Node

在只关心方向，而不关心大小的情况下，我们需要将输入值进行标准化。输出向量将与输入值具有相同的方向，但长度为 1。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Normalize_float4(float4 In, out float4 Out)
{
    Out = normalize(In);
}
```

三十、长度-Length Node

返回长度输入向量的长度，也称为向量的大小。一个向量的长度用勾股定理计算。

一个二维向量的长度（向量的模）计算可以为：$\sqrt{x^2+y^2}$

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Float	输出值

2. 生成的代码示例

```
void Unity_Length_float4(float4 In, out float Out)
{
    Out = length(In);
```

哔哩哔哩次元游戏人生

}

三十一、 距离-Distance Node

返回输入 A 和 B 的值之间的欧几里德距离。这对于计算空间中两点之间的距离非常有用，并且通常用于计算有向距离函数(Signed Distance Function)。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic Vector	第一个输入值
B	Input	Dynamic Vector	第二个输入值
Out	Output	Float	输出值

2. 生成的代码示例

```
void Unity_Distance_float4(float4 A, float4 B, out float Out)
{
    Out = distance(A, B);
}
```

三十二、 点乘-Dot Product Node

返回输入值的数量积。对于归一化的输入向量，如果指向完全相同的方向，则返回 1，如果它们指向完全相反的方向，则返回 -1，如果向量垂直，则返回 0。

1. 端口

名称	方向	类型	描述
A	Input	Dynamic Vector	第一个输入值
B	Input	Dynamic Vector	第二个输入值
Out	Output	Float	输出值

2. 生成的代码示例

```
void Unity_DotProduct_float4(float4 A, float4 B, out float Out)
{
    Out = dot(A, B);
}
```

三十三、 小数-Fraction Node

返回输入值的小数部分；大于等于 0 小于 1。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Fraction_float4(float4 In, out float4 Out)
{
    Out = frac(In);
}
```

三十四、 随机范围-Random Range Node

返回一个基于 Seed 伪随机数，这个数介于输入的最小值(Min)和最大值(Max)之间。

哔哩哔哩次元游戏人生

1. 端口

名称	方向	类型	描述
Seed	Input	Vector 2	随机数种子
Min	Input	Float	最小值
Max	Input	Float	最大值
Out	Output	Float	输出值

2. 生成的代码示例

```
void Unity_RandomRange_float(float2 Seed, float Min, float Max, out float Out)
{
    float randomno =   frac(sin(dot(Seed, float2(12.9898, 78.233)))*43758.5453);
    Out = lerp(Min, Max, randomno);
}
```

三十五、 比较-Comparison Node

基于下拉菜单，去比较 A 和 B 的值，这通常用作分支节点的输入。

1. 端口

名称	方向	类型	描述
A	Input	Float	第一个输入值
B	Input	Float	第二个输入值
Out	Output	Boolean	输出值

2. 控件

名称	类型	选项	描述
	下拉菜单	Equal, NotEqual, Less, LessOrEqual, Greater, GreaterOrEqual	比较

3. 生成的代码示例

1） Equal

```
void Unity_Comparison_Equal_float(float A, float B, out float Out)
{
    Out = A == B ? 1 : 0;
}
```

2） NotEqual

```
void Unity_Comparison_NotEqual_float(float A, float B, out float Out)
{
    Out = A != B ? 1 : 0;
}
```

3） Less

```
void Unity_Comparison_Less_float(float A, float B, out float Out)
{
    Out = A < B ? 1 : 0;
```

哔哩哔哩次元游戏人生

```
}
```

4） LessOrEqual

```
void Unity_Comparison_LessOrEqual_float(float A, float B, out float Out)
{
    Out = A <= B ? 1 : 0;
}
```

5） Greater

```
void Unity_Comparison_Greater_float(float A, float B, out float Out)
{
    Out = A > B ? 1 : 0;
}
```

6） GreaterOrEqual

```
void Unity_Comparison_GreaterOrEqual_float(float A, float B, out float Out)
{
    Out = A >= B ? 1 : 0;
}
```

三十六、 指数-Exponential Node

输入值 In 为指数的幂。指数的底数 base 可以从节点上的 Base 下拉菜单中选择。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 控件

名称	类型	选项	描述
Base	下拉菜单	BaseE, Base2	Base E：返回 e 的输入 In 的幂 Base 2：返回 2 的输入 In 的幂

3. 生成的代码示例

1） BaseE

```
void Unity_Exponential_float4(float4 In, out float4 Out)
{
    Out = exp(In);
}
```

2） Base2

```
void Unity_Exponential2_float4(float4 In, out float4 Out)
{
    Out = exp2(In);
}
```

哔哩哔哩次元游戏人生

三十七、 符号-Sign Node

对于每个组件，如果输入 In 的值小于零，则返回 -1，如果等于零，则返回 0，如果大于零，则返回 1。

1. 端口

名称	方向	类型	描述
In	Input	Dynamic Vector	输入值
Out	Output	Dynamic Vector	输出值

2. 生成的代码示例

```
void Unity_Sign_float4(float4 In, out float4 Out)
{
    Out = sign(In);
}
```

三十八、 快捷键

Ctrl+C 复制

Ctrl+V 粘贴

Ctrl+D 直接创建副本

Ctrl+G 将选中内容编组

Delete 删除

Space 创建节点

我

哔哩哔哩次元游戏人生

ShaderGraph 实际案例

一、 2D 简易描边效果-减法版

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

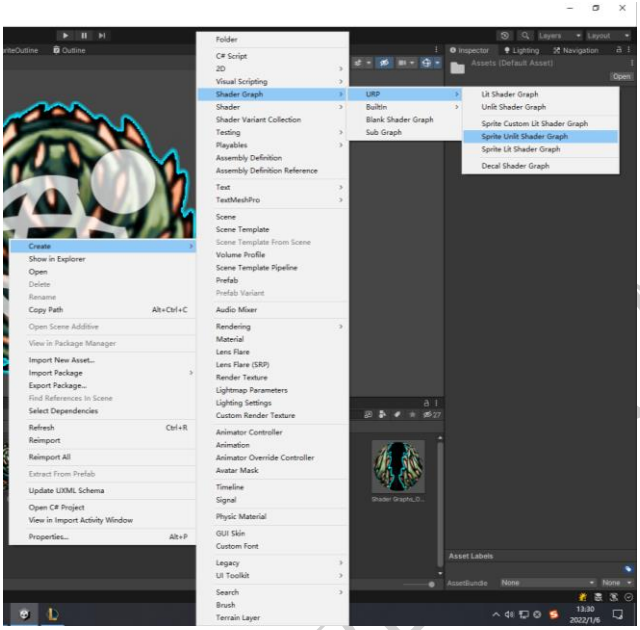


图112. 图 1.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[1],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图113. 图 1.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

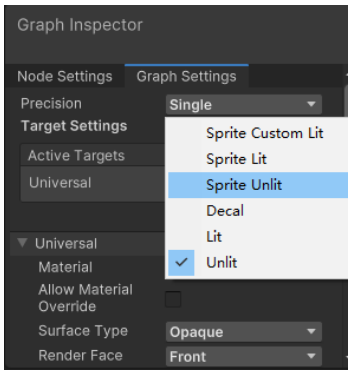


图114. 图 1.3 修改 Material

2. 创建描边轮廓

使用空格键添加 Sample Texture 2D 节点和 Step 节点，并为 Sample Texture 2D 选择一

[1] adj. 普遍的；通用的 n. 普遍原则；普遍看法

张参考的纹理贴图。接着我们使用 preview 节点去预览 Alpha 通道，白色区域是我们要显示的区域，黑色区域是不被显示的区域。

如果你的纹理贴图中含有杂项，就需要在 Step 节点中输入一个较小的值，去剔除这部分杂项的内容。其原理就是利用 Step 节点的运算，将杂项的部分变成黑色，使其不会被显示出来（操作会使图像数据有所损失）。然后按下 Ctrl 加 C 键，复制一个 Step 节点，输入一个较大的值，得到一个更小的遮罩（不显示的内容更少了，显示的内容更多了）。因为小的遮罩显示的内容更多，减去大的遮罩后会有剩余（黑色的区域为 0，白色的区域为 1，1 减去 0 自然剩下 1）。

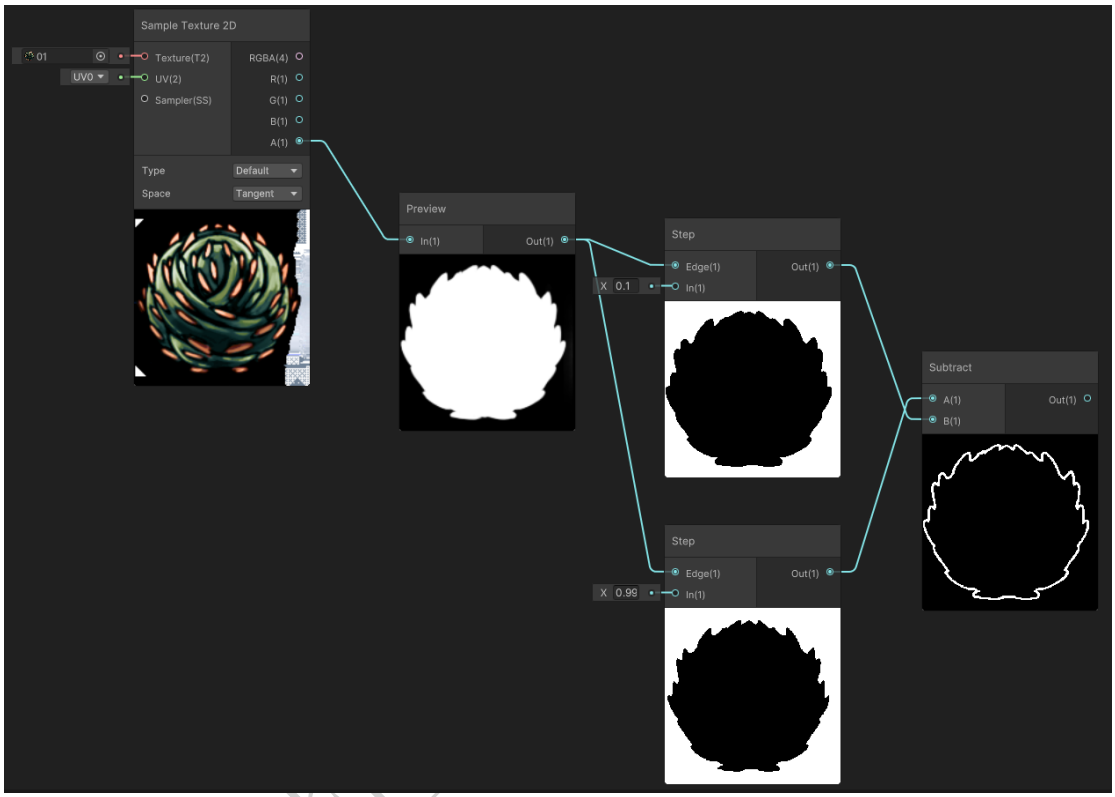


图115.　　图 1.4 描边轮廓

3. 修改描边轮廓颜色

创建一个 Color Node（颜色节点），如果要使用高动态范围，就将这个颜色节点的 Mode（模式）修改为 HDR。

有了颜色节点后，我们就可以使用 Multiply Node（乘法节点）将轮廓和颜色相乘，得到一个颜色被修改后的描边轮廓。

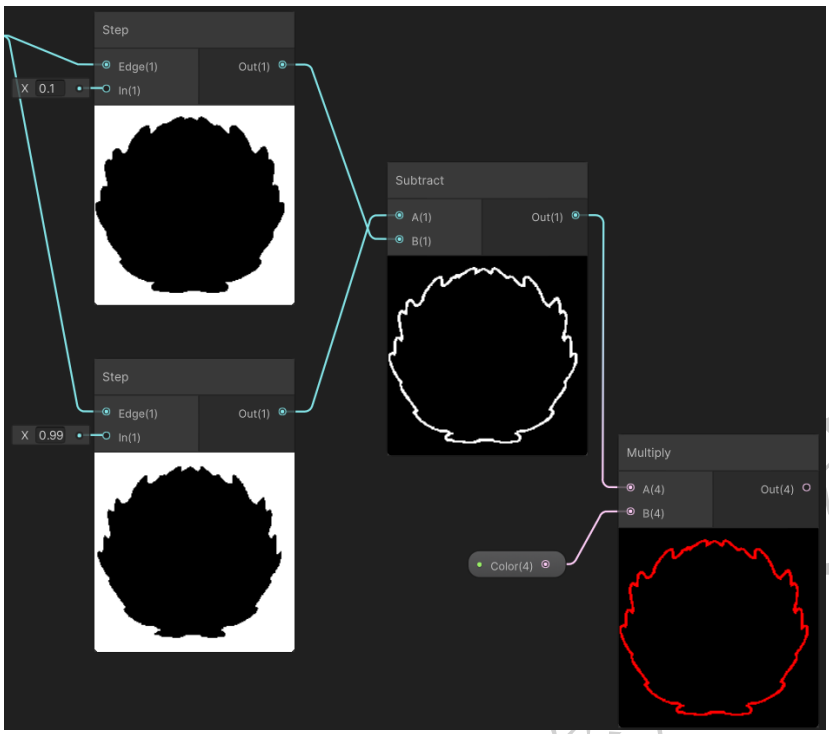
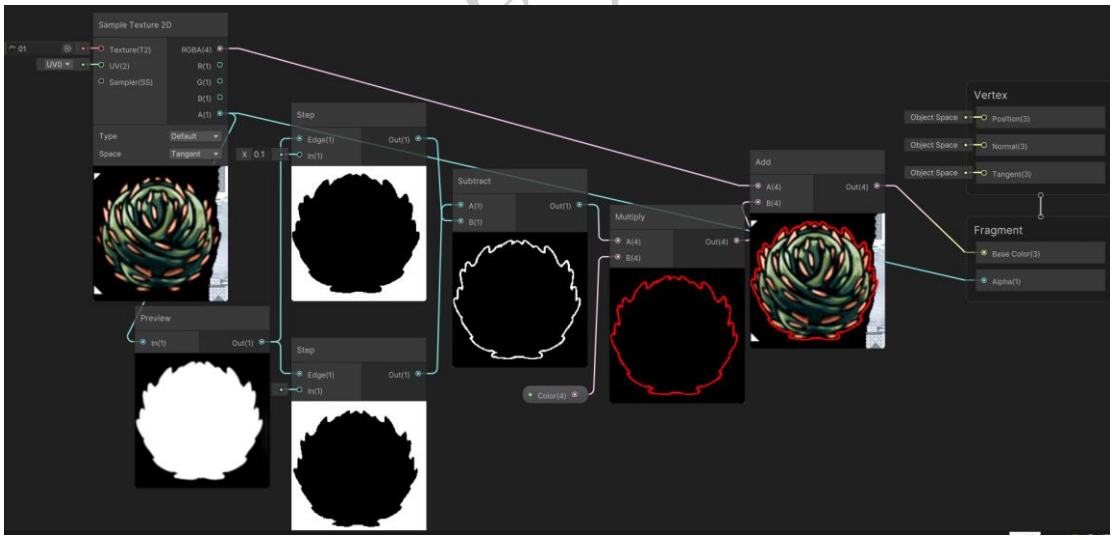


图116.　　图 1.5 修改描边轮廓

4. 得到最终成像

有了可以修改颜色的描边轮廓以后，我们就可以使用 Add Node（加法节点），将原图像和描边轮廓进行一个组合输出。



二、 2D 描边效果-加法版

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

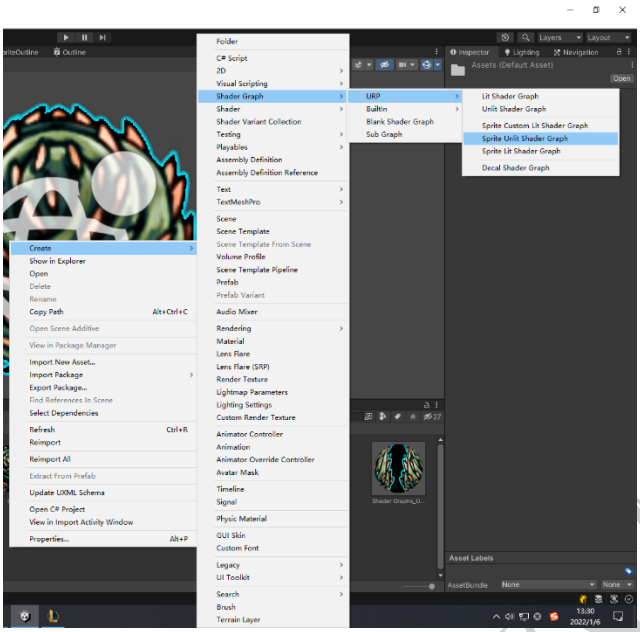


图117.　图 2.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[2],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图118.　图 2.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

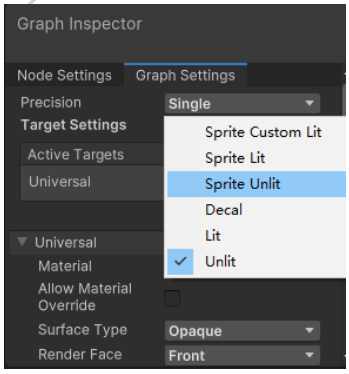


图119.　图 2.3 修改 Material

2. 放大原图像遮罩

在 2D 简易描边效果，我们通过 Step 节点取得了图像的描边，但是这个 2D 描边存在一些问题，对于像素图集这个描边方法就会失效。2D 简易描边效果主要是在原图形上做减法，即用 Step 节点剔除部分区域，将剩余部分用于描边。而用于显示的描边区域恰好就是柔性过渡的部分，而在像素图集中没有足够的柔性像素，我们一旦剔除就是剔除全部像素。

[2] adj. 普遍的；通用的 n. 普遍原则；普遍看法

因此用减法做描边对像素图集行不通，这时候我们就需要考虑做加法。也就是让原图形变得更大，扩展一部分出去，用扩展后的图像减去原本的图像来得到描边。为了扩大图像，我们需要使用 Tiling And Offset 节点将原图形向四个方向进行偏移，再将其合并成一张。

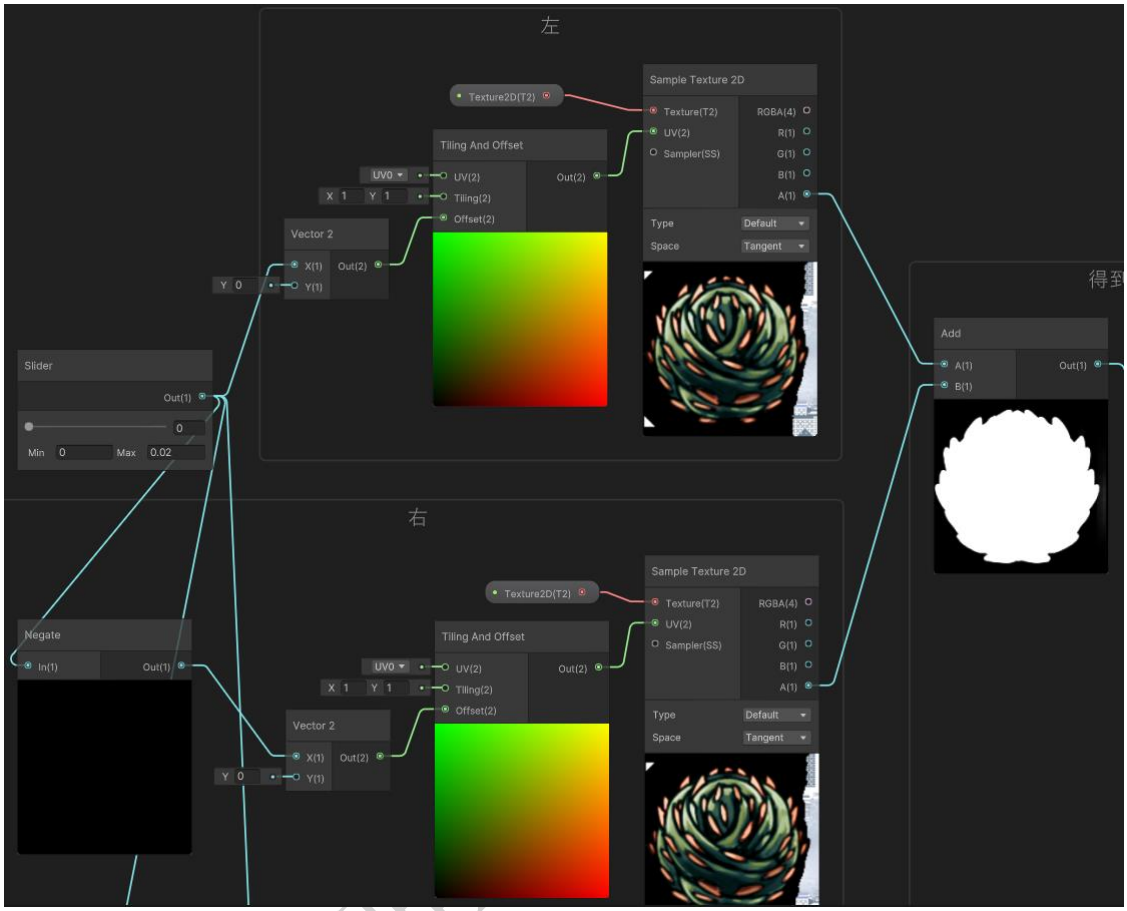


图120.　图 2.4 得到左右偏移图像

上图已经得到了左右偏移的图像，上下偏移的图像也是类似，只不过修改的二维向量变为 Y。使用 Negate 节点，我们就我们使用一个参数去控制偏移量，而不需要再额外设置一个参数作为反方向的偏移量。

3. 获得描边

如果我们将完整的扩展图像和原图形相减会发现，描边并没有出现，出现的依旧是一张完整的遮罩图像。正常情况下遮罩的值是在 0 到 1 之间，因为我们将四层偏移图像进行了叠加，得到的遮罩的值便变成了 0 到 4，因此我们需要使用 Clamp 节点去修正，保证大于 1 的值都变成 1。

当我们使用 Clamp 节点修正后，再将其与原图形的遮罩相减，就会得到正确的描边了。

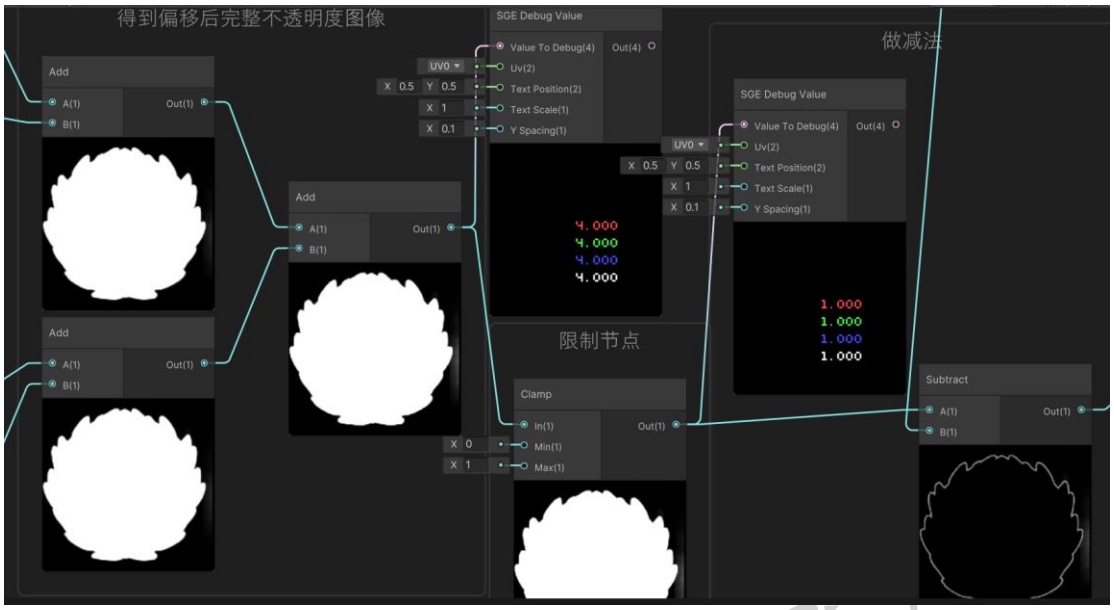


图121.　　图 2.5 通过 Clamp 节点得到正确描边

4. 修改描边颜色

创建一个 Color Node（颜色节点），如果要使用高动态范围，就将这个颜色节点的 Mode（模式）修改为 HDR。

有了颜色节点后，我们就可以使用 Multiply Node（乘法节点）将轮廓和颜色相乘，得到一个颜色被修改后的描边轮廓。

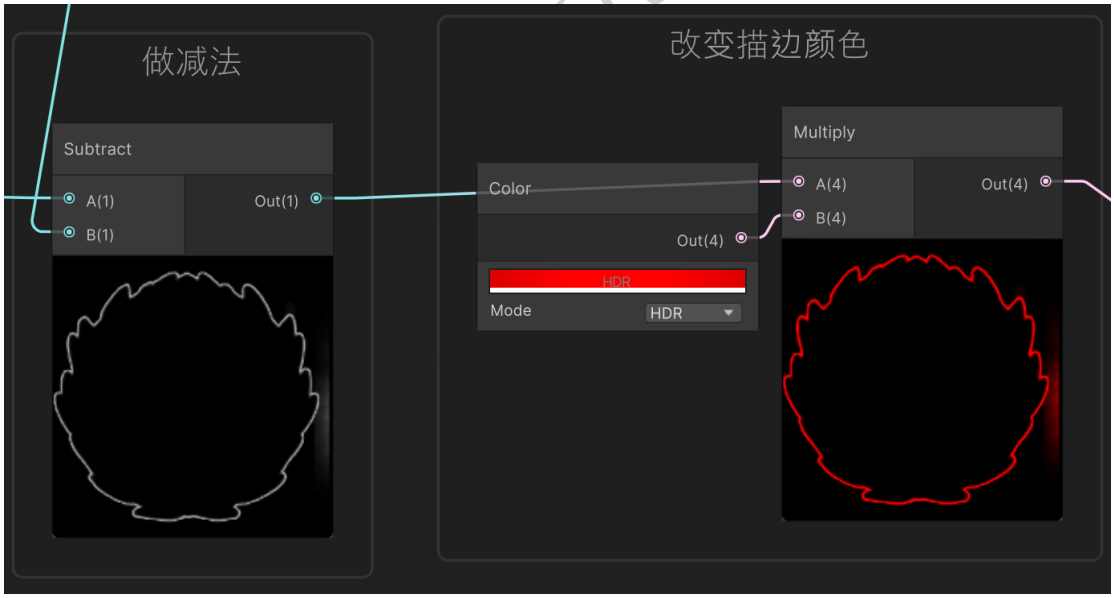


图122.　　图 2.6 修改描边区域的颜色

5. 得到最终成像

我们使用 Add 节点，将修改颜色后的描边与原图形相加将其输出到 Base Color；因为显示描边的是扩展出去的部分，如果使用原图像的 Alpha 通道进行输出，会发现描边的部分将不会显示，因此我们需要再将遮罩层相加，得到扩展后的正确遮罩。

现在我们使用像素图集也能够正确显示描边了。

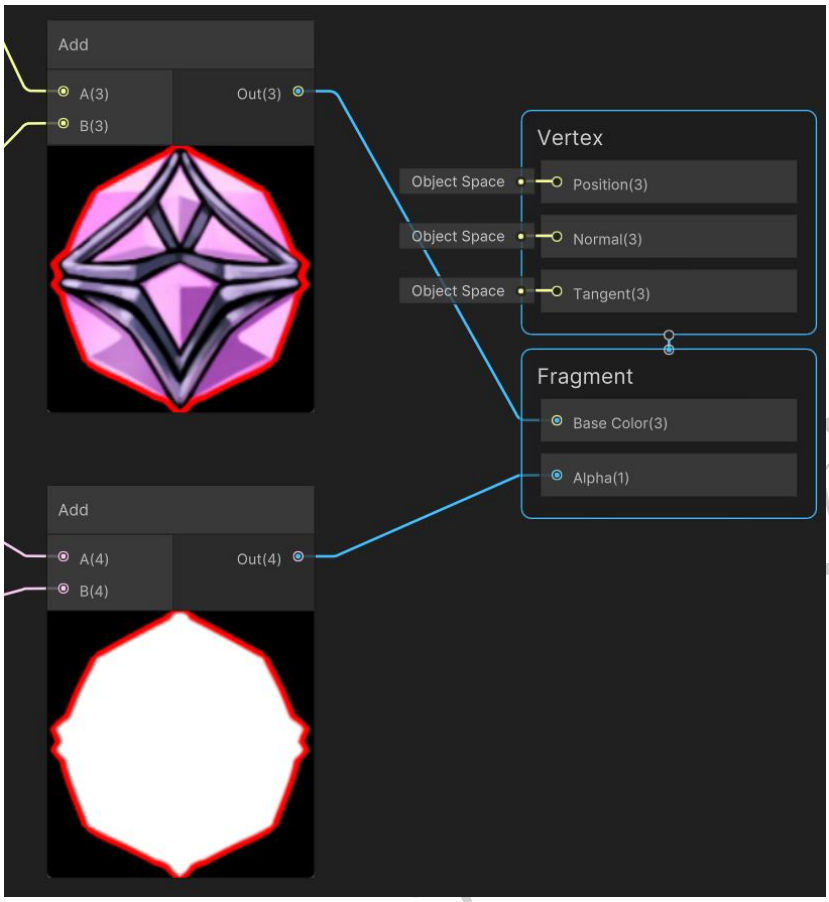


图123.　图 2.7 输出图像

6. 可能出现的问题

因为我们使用的 Clamp 节点是将超出的部分进行限制，而部分像素进行四次叠加后的值并未超出范围，因此这些像素将不会被限制，导致这些像素的值将比原图像遮罩的像素值大。因为这个原因，未进行偏移的情况下，这部分图像中描边效果依旧会出现，而像素图集因为没有足够的过渡像素，叠加的值必然会被限制，因此未偏移时不会出现描边。

三、 2D UI 技能冷却缓动

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

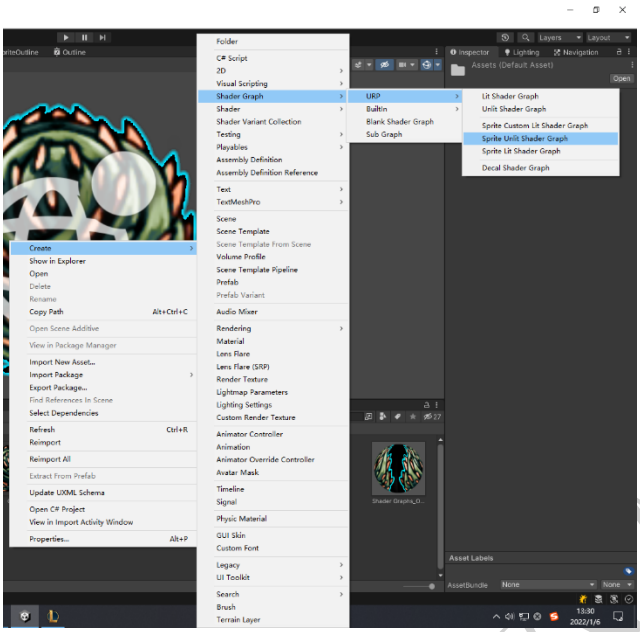


图124.　　图 3.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[3],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图125.　　图 3.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

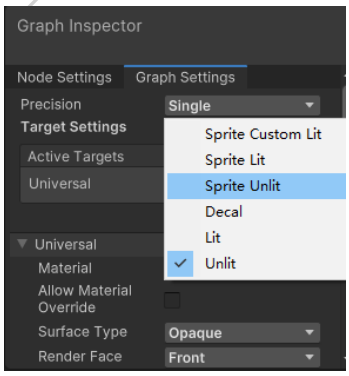


图126.　　图 3.3 修改 Material

2. 创建切分效果

首先创建一个 UV 节点和一个 Split（分离）节点，用 Split 节点将 UV 节点的 RG 通道分离出来，紧接着用 Preivew 节点去分别预览 R 通道和 G 通道，我们就会发现 R 通道是横向的渐变，而 G 通道则是纵向的渐变；因此可以得知 R 通道代表的 X，而 G 通道则代表的 Y。

[3] adj. 普遍的；通用的 n. 普遍原则；普遍看法

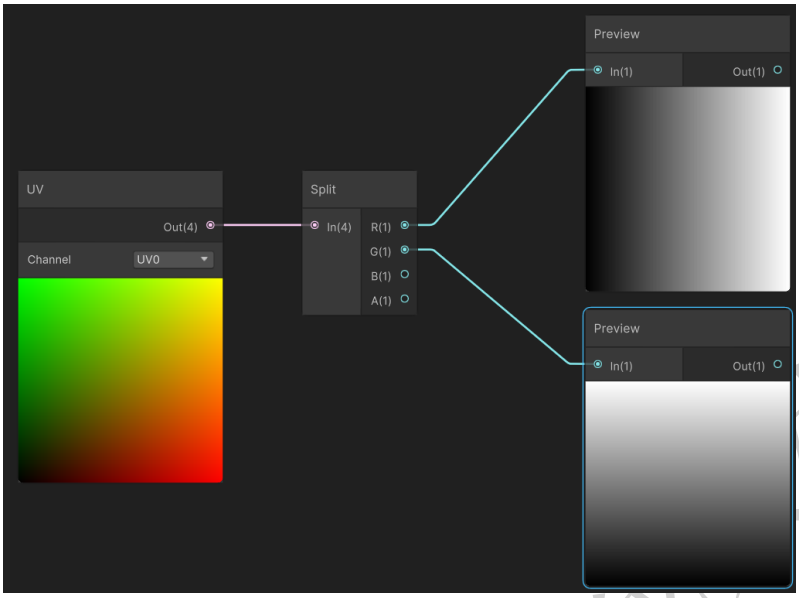


图127.　　图 3.4 预览 UV 的 R 通道和 G 通道

用 Step[4]节点对 R 通道和 G 通道进行切分，因为 R 通道和 G 通道都是从 0（黑）到 1（白）的渐变，因此当我们给 Step 节点输入 0.25 时，可以看到大于 0.25 的区域都是显示为黑色，而小于等于 0.25 的区域都显示为白色。

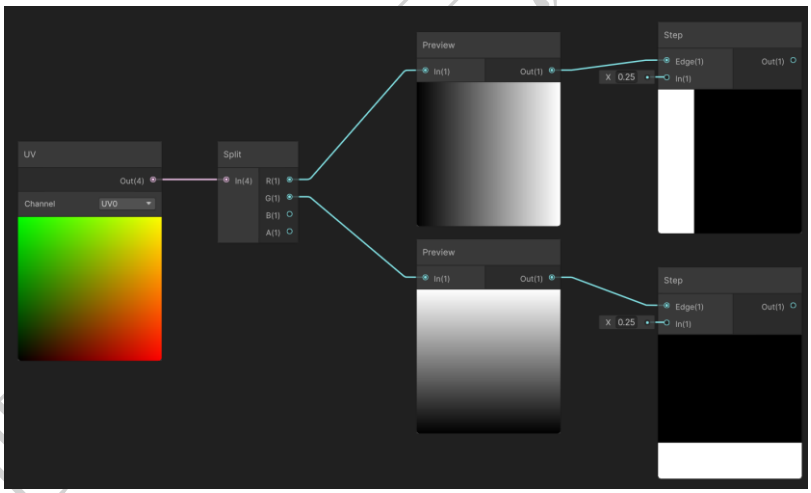


图128.　　图 3.5 用 Step 节点对 R 通道和 G 通道进行切分

3. 重新映射

因为黑色的区域是不被显示的区域，而这里我们是需要其能够显示出来的。因此我们需要创建一个 Remap Node,将值为 0 的区域进行一个调整，使该区域的值变成大于 0。完成映射后只需要再创建一个 Sample Texture 2D 节点和 Multiply 节点，将映射后的纹理和原图像相乘就可以得到最终成像。

[4] n. 迈步，脚步；梯级，台阶；步骤，措施；阶段，进程；
v. 跨步走；踏，踩；做，采取行动；（短距离）移动，行走；

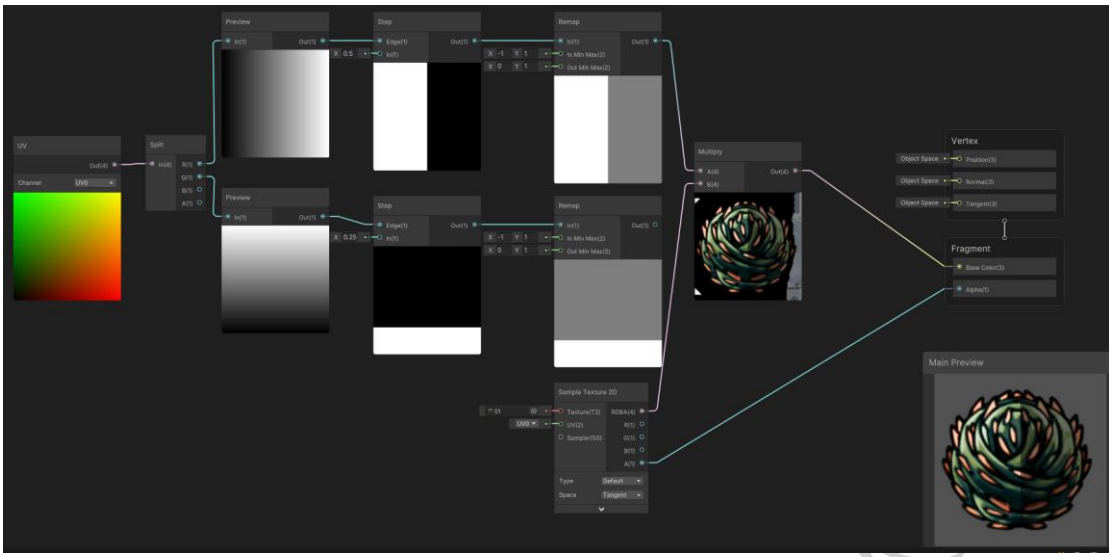


图129.　　图 3.6 重新映射后的最终图像

4. 扩展-旋转 UV

为了能够自定义渐变的方向，我们可以将 UV 旋转一下，只需要将 UV 改为 Rotate 即可。为了能够更好的调整旋转值，我们最好是使用 Remap 节点和 Slider 节点对旋转角度做一个映射，如果直接使用 Slider 节点控制旋转也是可以的，只不过当我们需要知道旋转了多少角度的时候就不太方便了，而使用 Remap 节点我们可以更轻易的明白旋转了多少角度。

图130.　　图 3.7 将冷却缓动的 UV 进行旋转

5. 扩展-极坐标

如果我们不想使用横向过渡和纵向过渡，我们可以使用 Polar Coordinates（极坐标）节

点，将二维空间直角坐标系转换为极坐标系，实现径向过渡。

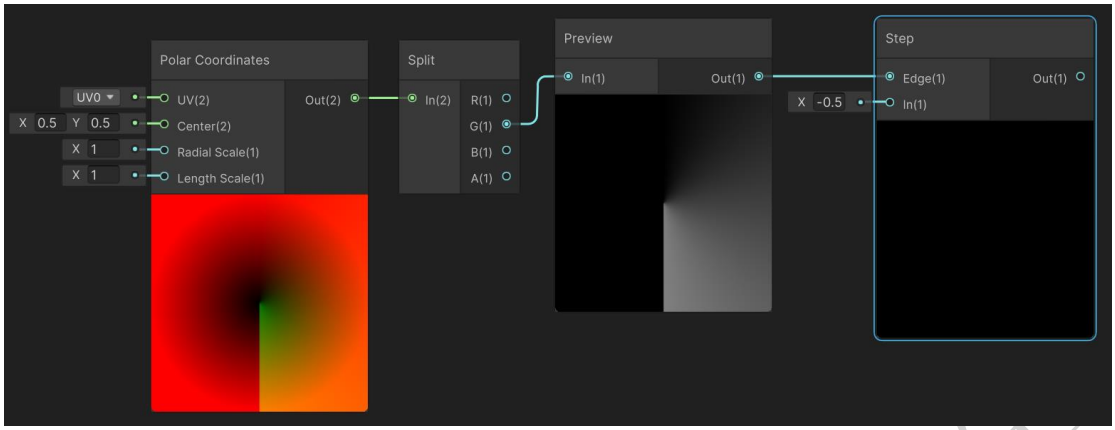


图131.　　图 3.8 使用极坐标实现冷却过渡

由上图可以知道，从该极坐标分离出的 G 通道的最小值是-0.5，这不方便我们在外部对其进行修改。为了方便外部的修改，我们需要将分离出的 G 通道进行 Remap；这里需要使用两次 Remap，第一次是将该极坐标分离出的 G 通道的范围调整至 0 到 1 之间，第二次 Remap 是修改 Step 后的黑色区域，使之能够显示出来。

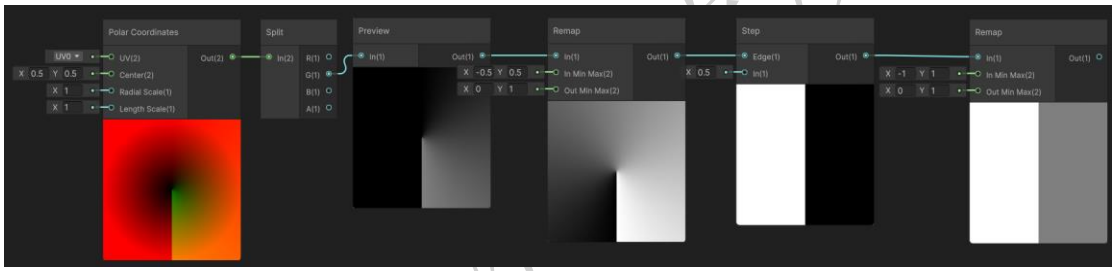


图132.　　图 3.9 使用 Remap 节点重新映射 G 通道值

6. 在代码中控制

1） 方法一

直接使用字符串去控制 Shader 代码中的参数，这个字符串即参数的名称。在访问 Shader 的参数前，一定要获得该 Shdaer 的实例，即 Material（材质）。

哔哩哔哩动画技术

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

Unity 脚本(1 个资产引用)|0 个引用
public class NewBehaviourScript : MonoBehaviour
{
    public Material FillMaterial;
    public float FillAmount;
    private int Add;
    Unity 消息|0 个引用
    void Start()
    {
        FillMaterial = this.gameObject.GetComponent<SpriteRenderer>().material;
        方法二
    }
    Unity 消息|0 个引用
    void Update()
    {
        #region 方法一
        if (Input.GetKeyDown(KeyCode.Z))
        {
            FillAmount = FillMaterial.GetFloat("_FillAmount") - 0.1f;
            FillMaterial.SetFloat("_FillAmount", FillAmount);
        }
        #endregion
        方法二
    }
}
```

图133.　图 3.10 访问 Shader 参数方法一

请注意，这个字符串不是面板上显示的名称，而是在 Graph Inspector 界面显示的 reference。

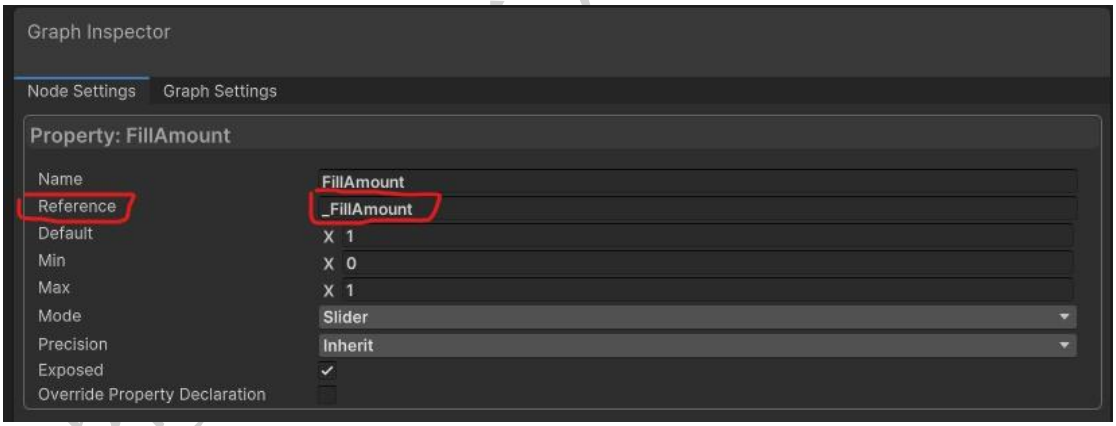


图134.　图 3.11 Graph Inspector 界面的 Reference

2）方法二

将字符串转换为唯一的整型标识符（使用 Shader.PropertyToID 获取着色器属性名称的唯一标识符），在脚本中用一个整型变量去存储它。

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

Unity 脚本(1 个资产引用)|0 个引用
public class NewBehaviourScript : MonoBehaviour
{
    public Material FillMaterial;
    public float FillAmount;
    private int Add;
    Unity 消息|0 个引用
    void Start()
    {
        FillMaterial = this.gameObject.GetComponent<SpriteRenderer>().material;
        # region 方法二
        Add = Shader.PropertyToID("_FillAmount");//使用Shader.PropertyToID获取着色器属性名称的唯一标识符。
        #endregion
    }
    Unity 消息|0 个引用
    void Update()
    {
        方法一
        # region 方法二
        if (Input.GetKeyDown(KeyCode.X))
        {
            FillAmount = FillMaterial.GetFloat(Add) + 0.1f;
            FillMaterial.SetFloat(Add, FillAmount);
        }
        #endregion
    }
}
```

图135.　　图 3.12 访问 Shader 参数方法二

四、 2D 图像 RGB 分离

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

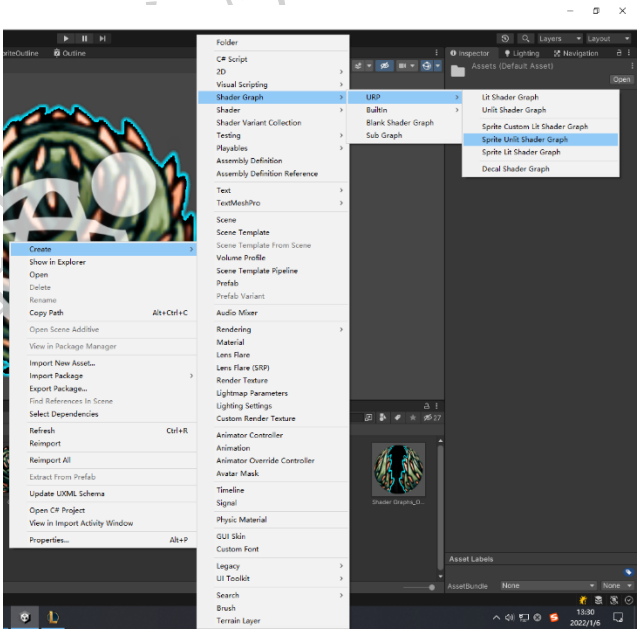


图136.　　图 4.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[5],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图137.　图 4.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

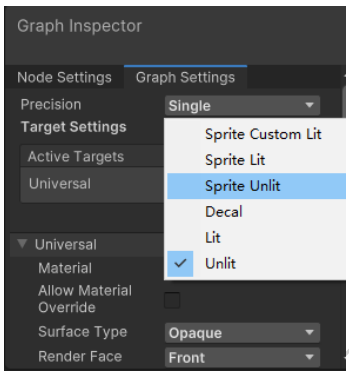


图138.　图 4.3 修改 Material

2. 添加可外部访问的参数

创建一个 Texture 2D 类型和 Float 类型的参数，并将 Texture 2D 类型的参数命名为 MainTex。为了方便我们调节 Float 类型的参数，我们修改它的模式为 Slider 即滑动条模式，并给予一个适当的最小值和最大值，这里给予最小值为-0.01，最大值为 0.01。

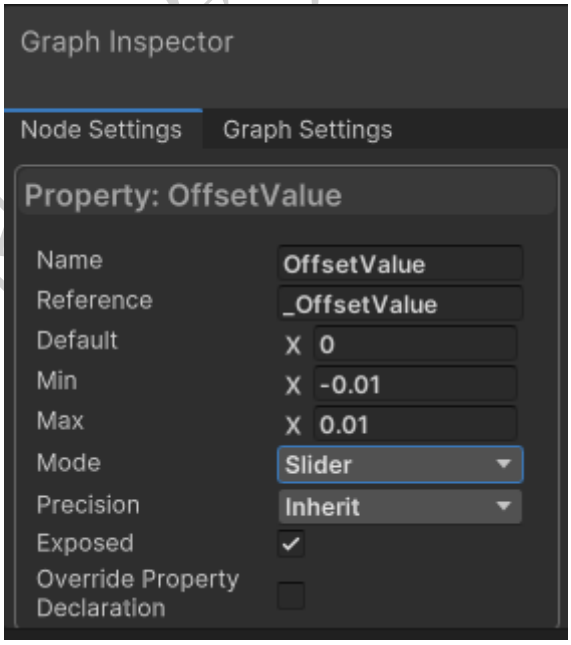


图139.　在 Graph Inspector 中修改参数模式

3. 添加节点

首先创建 3 个 Sample Texture 2D 节点，并将之前创建的主纹理参数赋予这三个节点。然后创建一个 Vector3 节点或 Combine 节点，分别将 3 个 Sample Texture 2D 的 R、G、B 通道进行结合，完成后将其输出。

[5] adj. 普遍的；通用的 n. 普遍原则；普遍看法

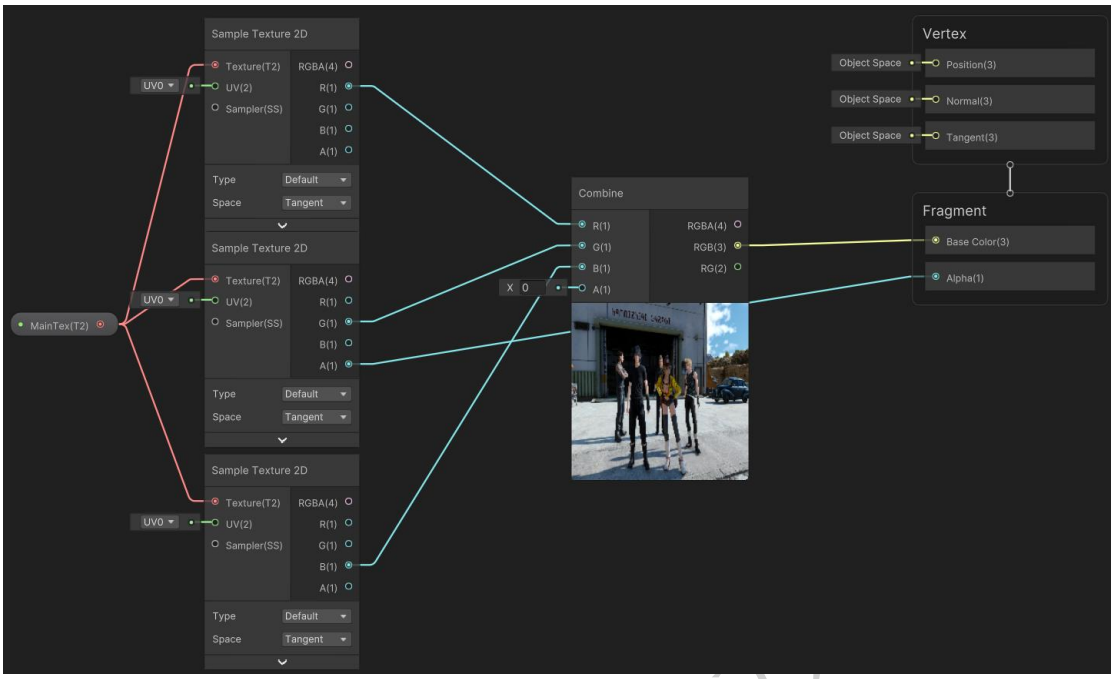


图140.　创建 RGB 分离的采样节点

接下来创建 UV 节点、加法节点和减法节点，加法节点和减法节点共用之前创建的 Float 参数，这个 Float 参数将决定 UV 偏移的量。加法节点和减法节点之所以共用一个参数，是为了保证偏移量的一致性，当然也可以使用独立的参数对其进行控制。

哔哩哔哩次元游戏

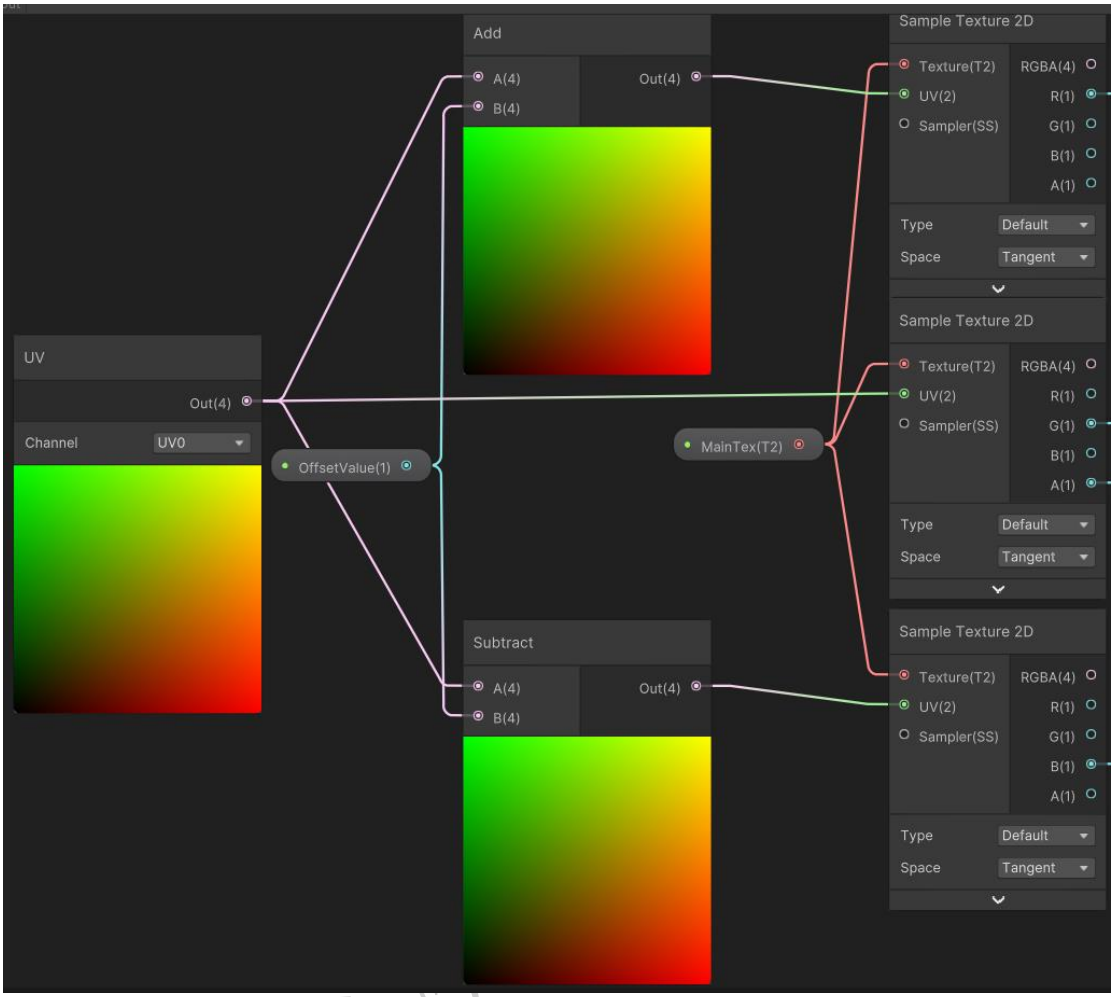


图141.　图 4.6 RGB 分离 UV 偏移

图142.　图 4.7 RGB 分离的最终效果图

五、 2D 全息效果

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

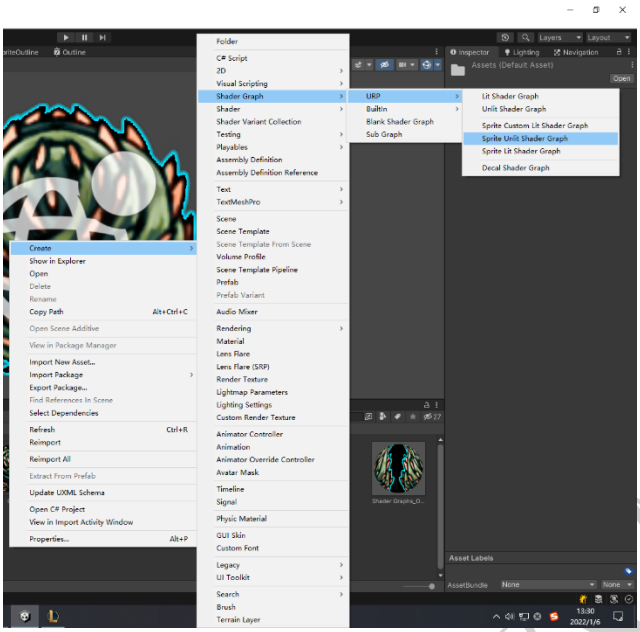


图143.　图 5.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[6],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图144.　图 5.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

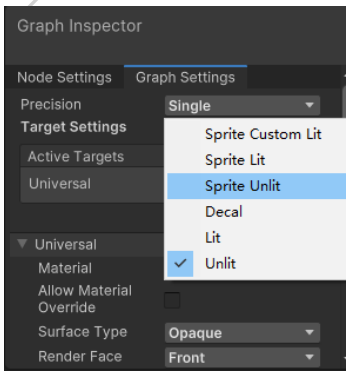


图145.　图 5.3 修改 Material

2. 添加可外部访问的参数

创建两个 Texture 2D 类型、一个 Float 类型和一个 Color 类型的参数，将其中一个 Texture 2D 类型的参数命名为 MainTex，是要显示的主图像，另一个 Texture 2D 类型的参数是全息贴图。Float 类型的参数用于控制全息效果的流动速度，Color 类型的参数调整全息的颜色。

[6] adj. 普遍的；通用的 n. 普遍原则；普遍看法

3. 创建带全息效果的图像

首先创建两个 Sample Texture 2D 节点，将之前创建的 Texture 2D 类型的参数分别赋予这两个节点。创建一个 Multiply 节点，将原图像的 Alpha 通道和全息图像的 R 通道相乘，得到带有全息效果的原图像遮罩。紧接着再创建一个 Multiply 节点，将带有全息效果的原图像遮罩和 Color 类型的参数相乘，得到可以调整颜色的遮罩，将这个遮罩连接到 Alpha 输出。最后创建一个 Add 节点，将原图像和调整颜色后的原图像遮罩相加，得到最终成像。

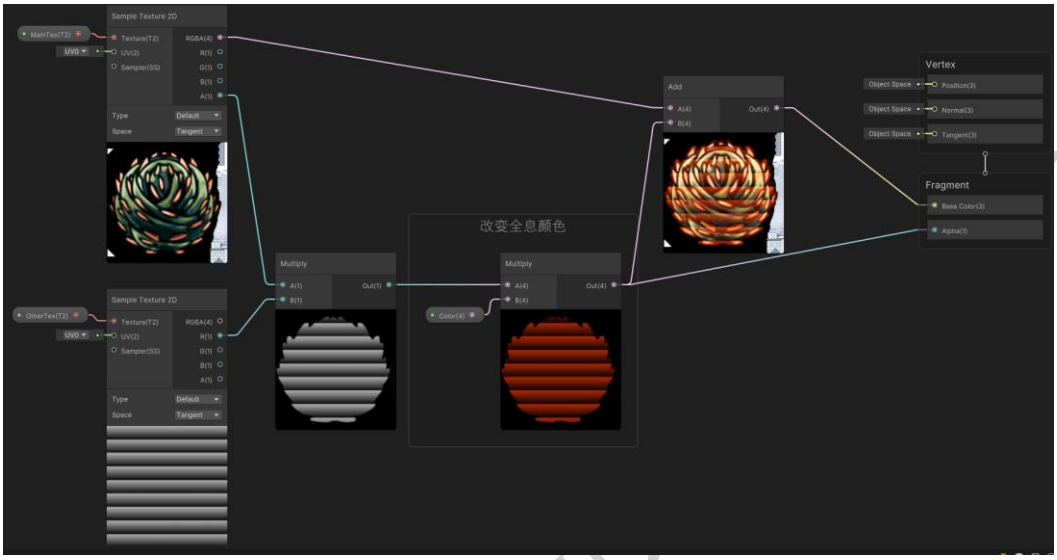


图146.　　图 5.4 创建带全息效果的图像

越接近白色的地放的不透明度越大，越接近黑色的地方显示的不透明度越小。

4. 让全息纹理流动起来

让贴图流动起来主要是使用 Time 节点。我们将之前设置的用于控制流速的参数和 Time 节点的 Time 端口相乘，然后再创建一个 Tiling And Offset 节点，将相乘后所得的结果给予 Tiling And Offset 节点的 Offset 端口，最后将其连向采样全息贴图的 UV 端口。现在我们就可以看到全息贴图流动了起来。

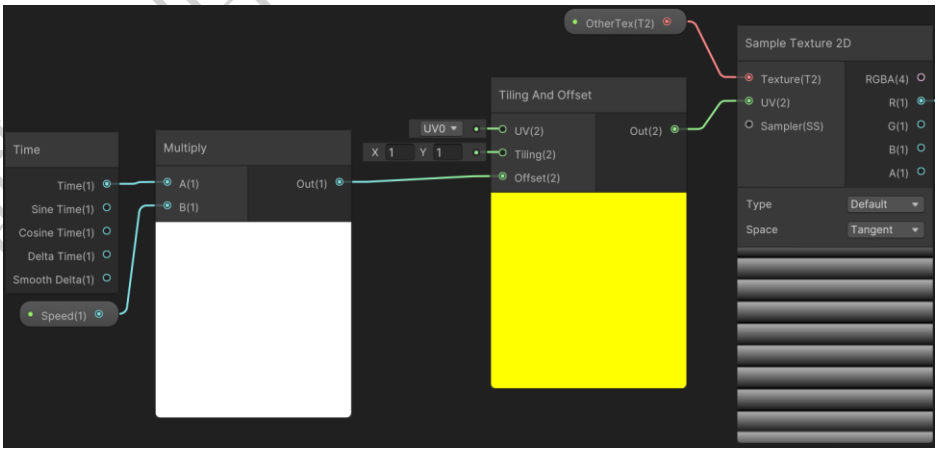


图147.　　图 5.5 让全息贴图流动

5. 扩展-调节全息贴图显示范围

虽然全息效果已经实现了，但如果我们想调整全息的显示强度、范围和密度怎么办呢？我们使用 Remap 节点，将全息贴图重新映射一下，调节全息的显示强度和范围，然后

再使用一个二维数组去调节全息贴图的密度。

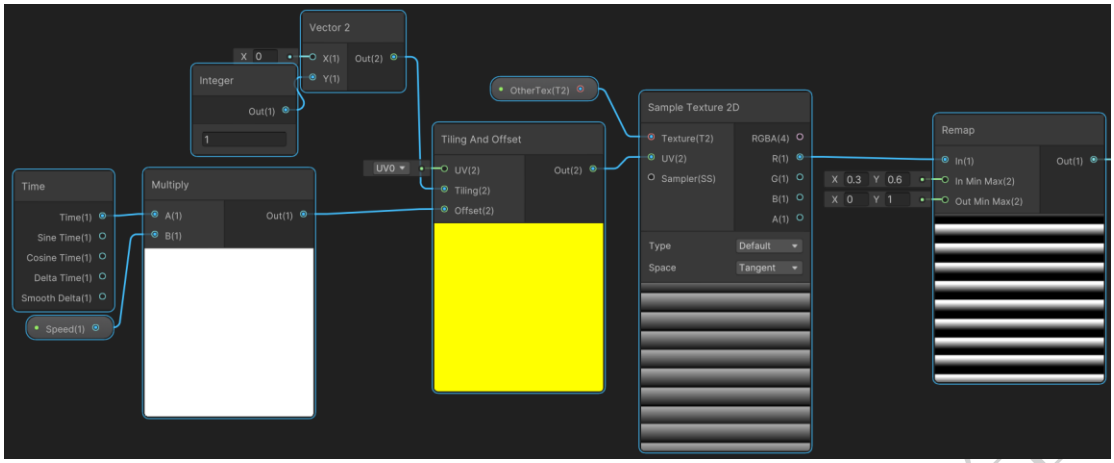


图148.　　图 5.6 调节全息范围、强度和密度

我们像上图那样使用 Remap 节点，可能会导致重新映射的值为负数（详细请见 Remap 节点的运算），这会导致最终的成像有问题。因此，我们需要限制重新映射后所得值的范围，这需要使用 Clamp 节点。利用 Clamp 节点将值的范围限制在 0 到 1 之间，所有大于 1 的值都将变为 1，小于 0 的值都将变为 0。

图149.　　图 5.6 使用 Clamp 节点对值进行限制

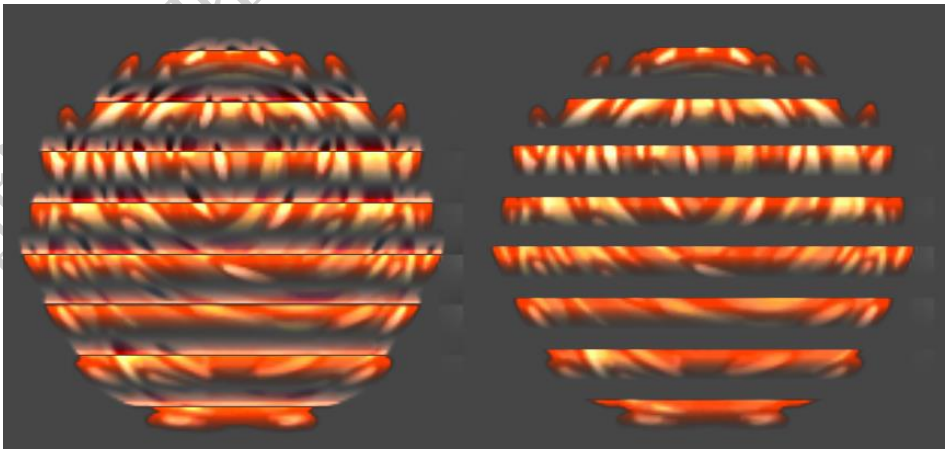


图150.　　图 5.7 调节前（左）和调节后（右）对比

六、　2D 溶解效果

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

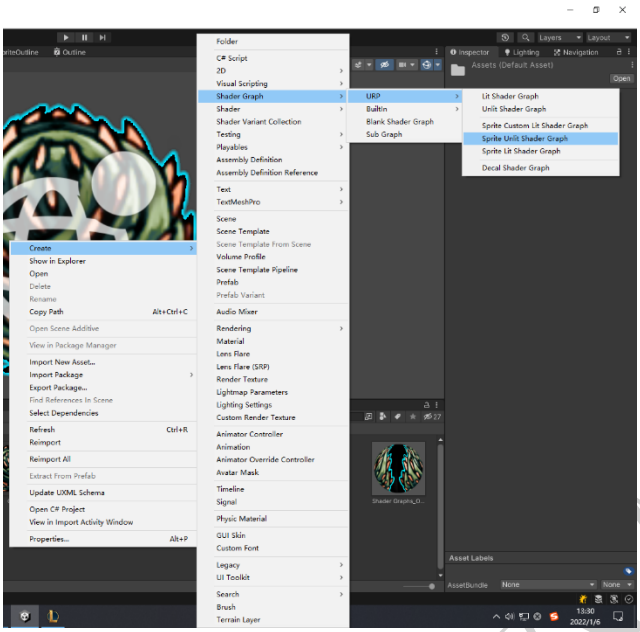


图151.　图 6.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[7],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图152.　图 6.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

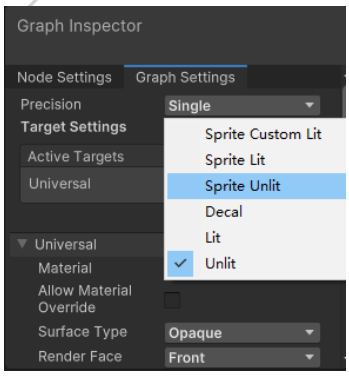


图153.　图 6.3 修改 Material

2. 添加可外部访问的参数

因为是溶解效果，我们需要在外部控制其溶解度，因此需要一个 Float 类型的变量，用其控制溶解度。一个 Texture2D 类型的参数显示主纹理，一个 Clolor 类型的参数用于控制溶解边缘的颜色。

[7] adj. 普遍的；通用的 n. 普遍原则；普遍看法

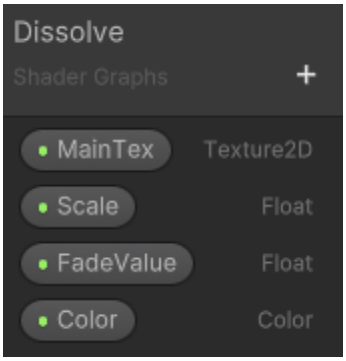


图154.　　图 6.4 可外部访问的参数

如果需要在外部控制溶解的密度，可再添加一个 Float 类型的参数。

3. 创建溶解效果

溶解效果我们使用 Simple Noise 节点，这个节点会生成一些噪声图像，我们使用这些噪声图像配合 Step 节点实现溶解效果。当然也可以将 Simple Noise 节点换成一张采样贴图，不过采样贴图的效率不如程序生成的高。

我们利用 Step 节点去剔除 Simple Noise 节点的部分区域，再将其和主纹理的 Alpha 通道相乘，得到当前溶解度所对应的溶解遮罩图像。

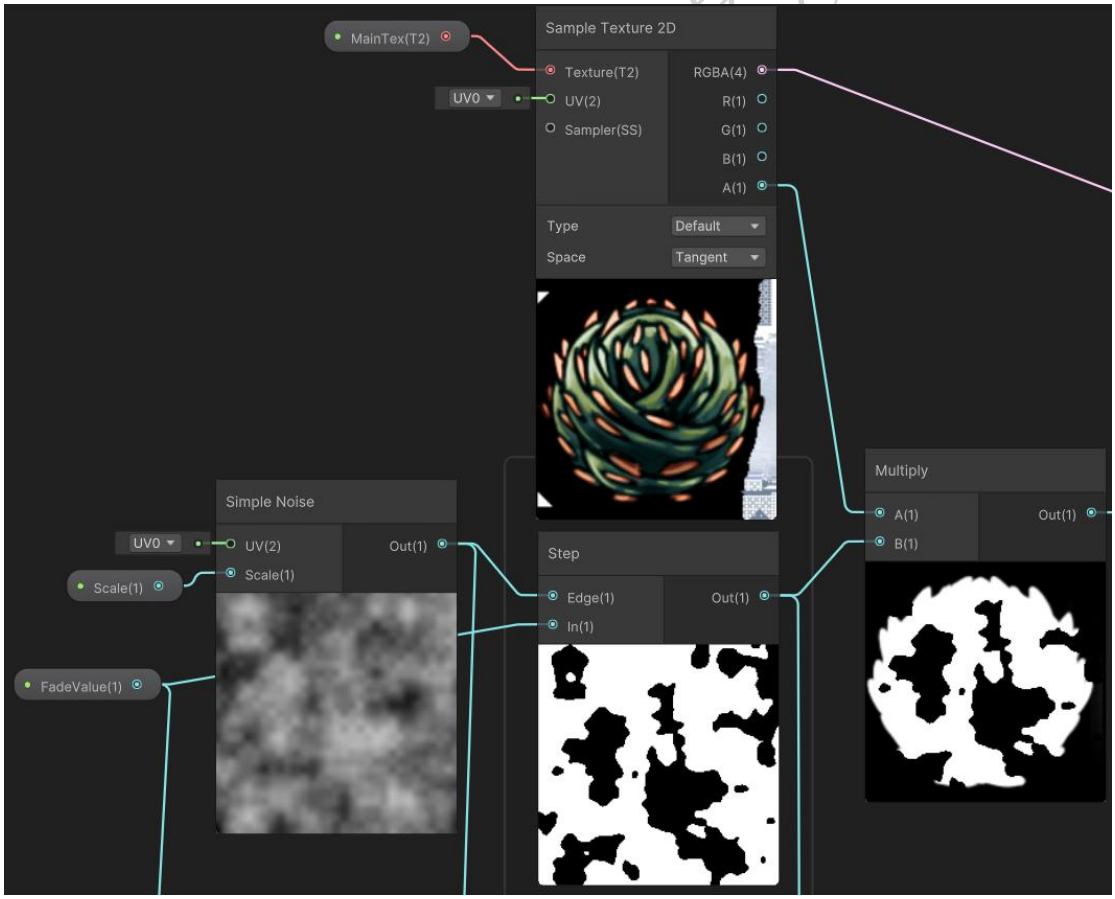


图155.　　图 6.5 当前溶解度对应的溶解效果

现在我们只需要将溶解遮罩传递给 Alpha 输出，将原图像传递给 BaseColor 就可以得到带有溶解效果的图像了。

4. 添加溶解边缘颜色

虽然现在拥有了溶解效果，但是其视觉表现并不好，我们就考虑为溶解的内边缘添加

描边颜色，丰富其视觉表现。

这里使用减法描边去实现，我们将当前溶解度稍作减少，然后再用 Step 节点剔除区域，将两个溶解遮罩相减，就可以得到内边缘的描边了。

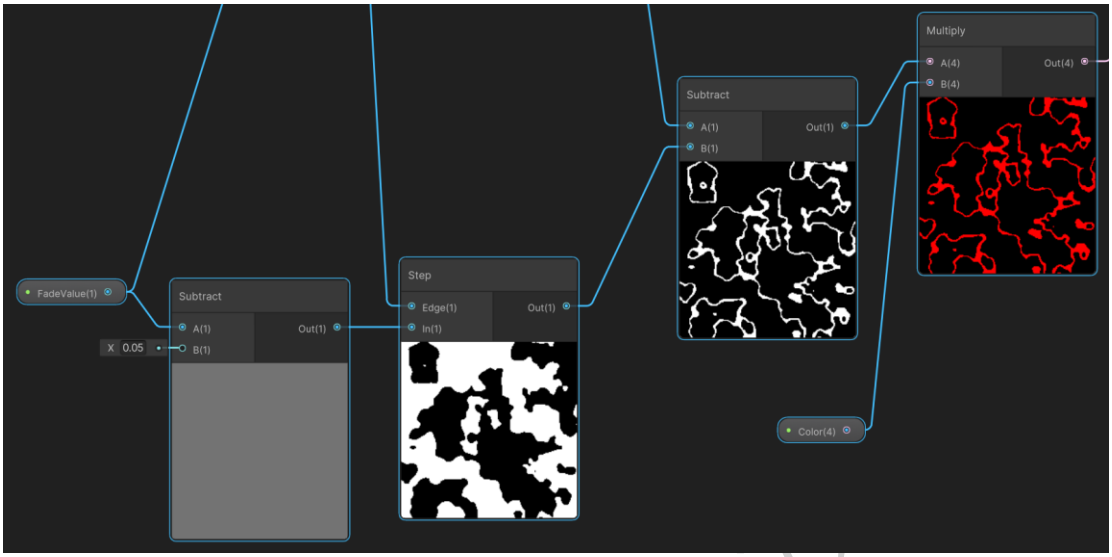


图156.　　图 6.6 溶解效果内边缘描边

然后我们就将颜色参数和描边遮罩相乘，就可以得到修改颜色后的图像遮罩了。要将这部分描边显示出来，需要将其和原图形相加，并输出到 BaseColor。

七、 2D 流光效果简易版

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

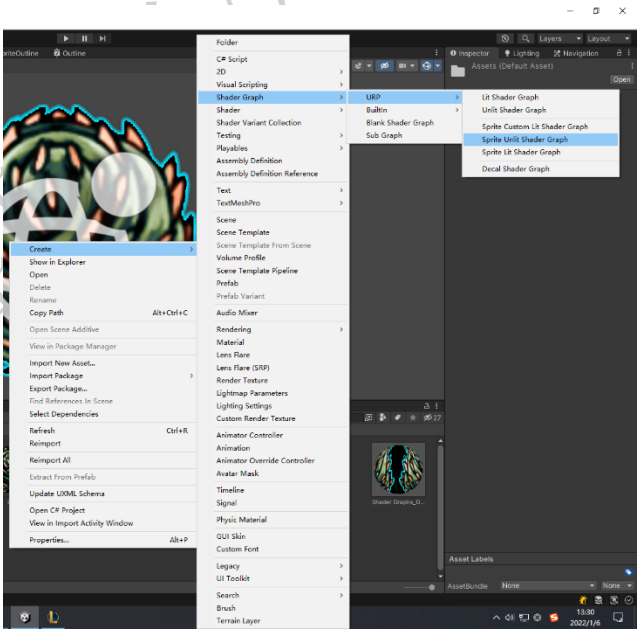


图157.　　图 7.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[8],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图158.　　图 7.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

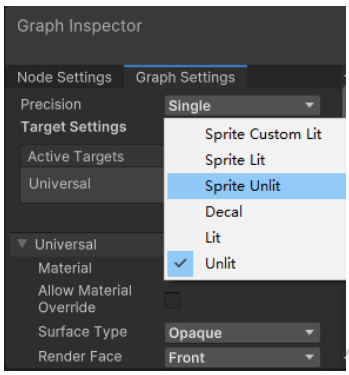


图159.　　图 7.3 修改 Material

2. 添加可外部访问的参数

添加两个 Texture2D 类型的参数，一个用于显示主纹理，另一个用于显示流光纹理。因为需要控制流光的流动速度，而流动的方向有横向和纵向，因此可以使用一个 Vector2 类型的参数，也可以使用两个 Float 类型的参数。如果要旋转流光，或者使流光一键反向移动，则需要额外添加两个参数，一个 Float 类型的参数和一个 Boolean 类型的参数。

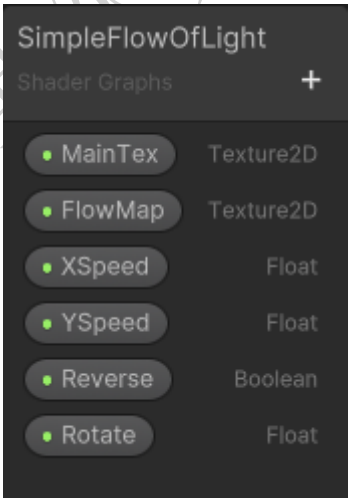


图160.　　图 7.4 2D 流光简易版参数

3. 显示含有流光效果的图像

首先创建两个 Sample Texture 2D 节点，将主纹理贴图和流光纹理贴图分别给予它们，然后使用加法节点将这两张纹理进行组合，并将其和原图形的遮罩分别输出到 BaseColor 端口和 Alpha 端口。

[8] adj. 普遍的；通用的 n. 普遍原则；普遍看法

哔哩哔哩不游戏人生

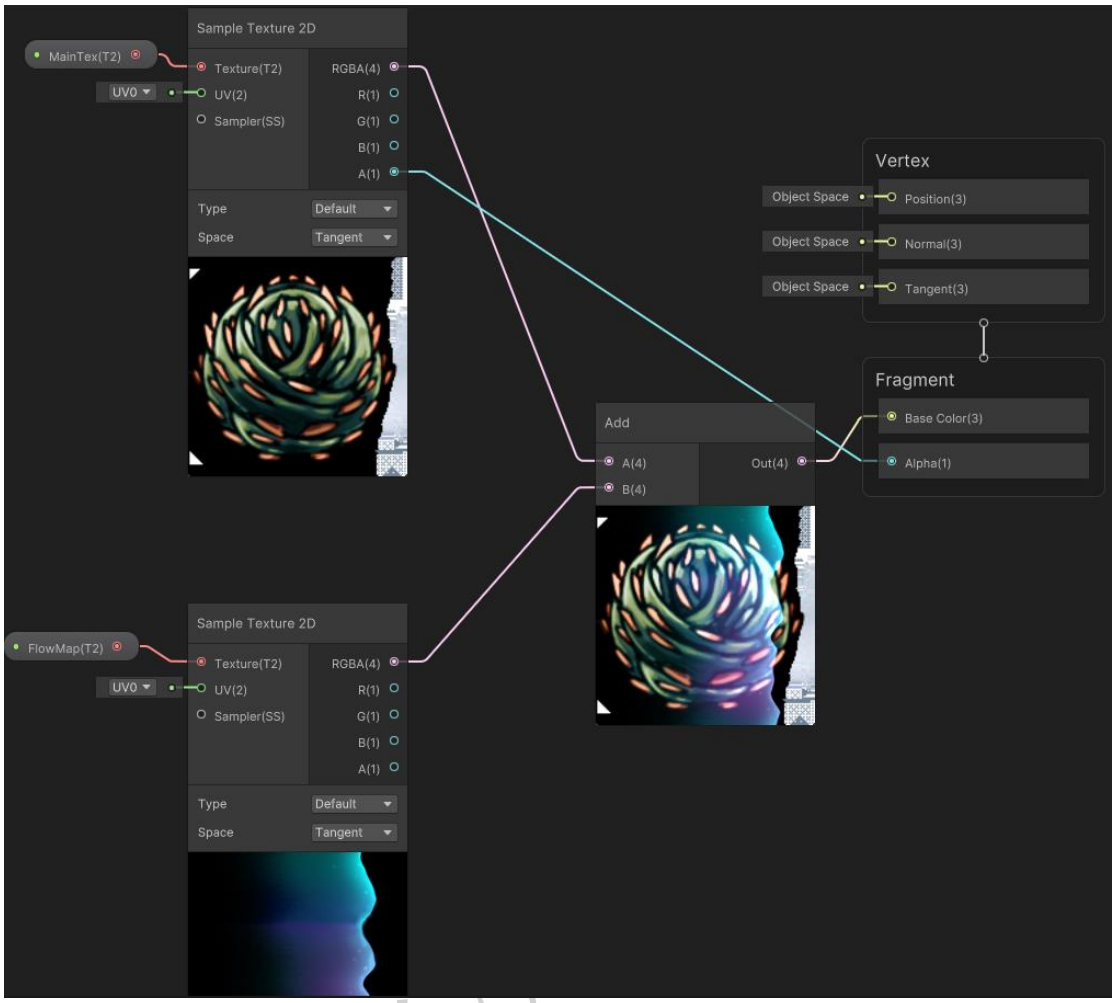


图161.　图 7.5 含有流光效果的图像

4. 让流光纹理流动起来

现在流光纹理已经和原图形一同显示出来了，接下来我们就需要使用 Time 节点和 Tiling And Offset 节点让流光纹理流动起来。

我们创建 Time 节点和 Tiling And Offset 节点，将 Tiling And Offset 节点的输出给予流光纹理的 UV 端口。因为 Tiling And Offset 节点的 Offset 端口是一个二维向量，因此我们创建一个 Vector2 节点，并将之前创建的控制流动速度的参数分别给予其 X 端口和 Y 端口。紧接着创建一个乘法节点，将二维速度向量和 Time 节点的 Time 端口相乘，并输出到 Tiling And Offset 节点的 Offset 端口。

现在就可以看到流光纹理流动起来了。

图162.　　图 7.6 让流光纹理流动起来

5. 反向运动

现在我们可以看到，当速度大于 0 时流光的流动方向是向左下角的，我们希望数值大于 0 时 X 方向是向右运动，Y 方向是向上运动的。因此我们在乘法结果输入到 Tiling And Offset 节点之前，使用 Negate 节点对其进行取反操作。

现在，速度值大于 0 时 X 方向就是向右运动，而 Y 方向则是向上运动的了。但是，如果我们想要其反方向运动呢，难不成还要重新去调整数值吗？这岂不是太麻烦了，因此我们使用 Branch 节点做一个判断。

当需要反向运动时就不取反，不需要反向运动的时候就取反，并使用一个 Boolean 参数去控制。

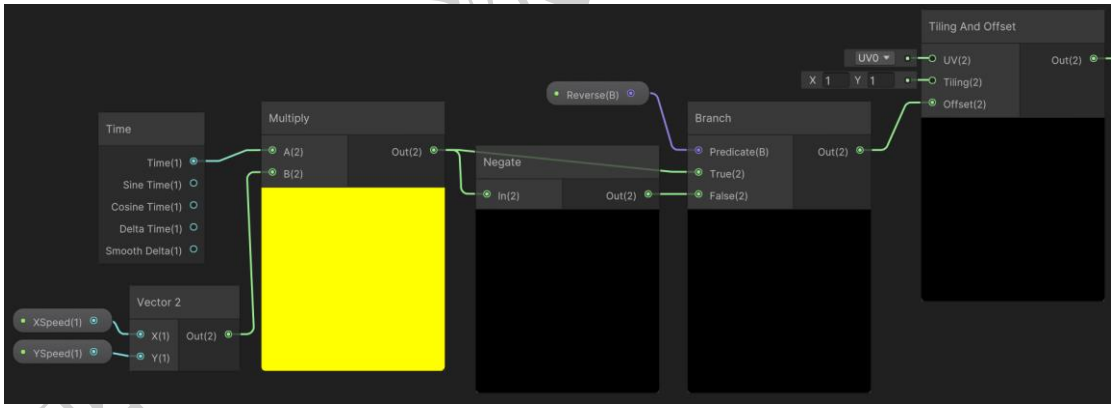


图163.　　图 7.7 流光反向运动

6. 旋转流光

流光效果现在已经可以一键反向运动了，但如果我们有改变流光纹理方向的需求呢？难不成我们需要去美术软件中修改流光纹理的方向吗？这岂不是很浪费时间。

因此我们使用 Rotate 节点去旋转流光纹理的 UV，使流光纹理的方向能够在 Unity 中修改。

创建一个 Rotate 节点，将之前创建的参数给予它的 Rotation 端口，并将其输出到 Tiling And Offset 节点的 UV 端口。

7. 可能出现的问题

如果按照上述操作进行，却没有出现流光效果，请检查一下流光纹理的设置，看其

WarpMode 是否为 Repeat（重复）。

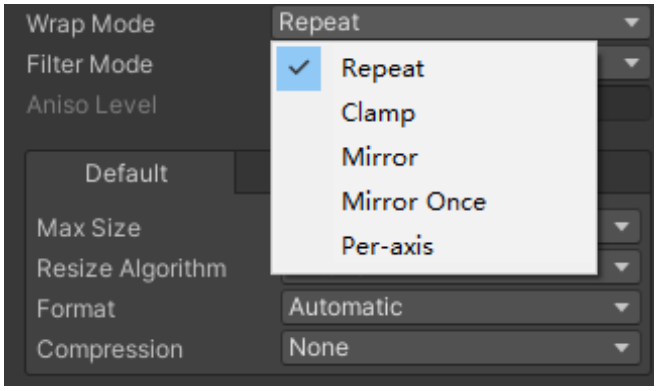


图164.　图检查纹理设置

如果旋转值设定的很小，但是却旋转了很多度，则需要检查 Rotate 节点的下拉菜单，查看其模式是否为 Degree-角度。

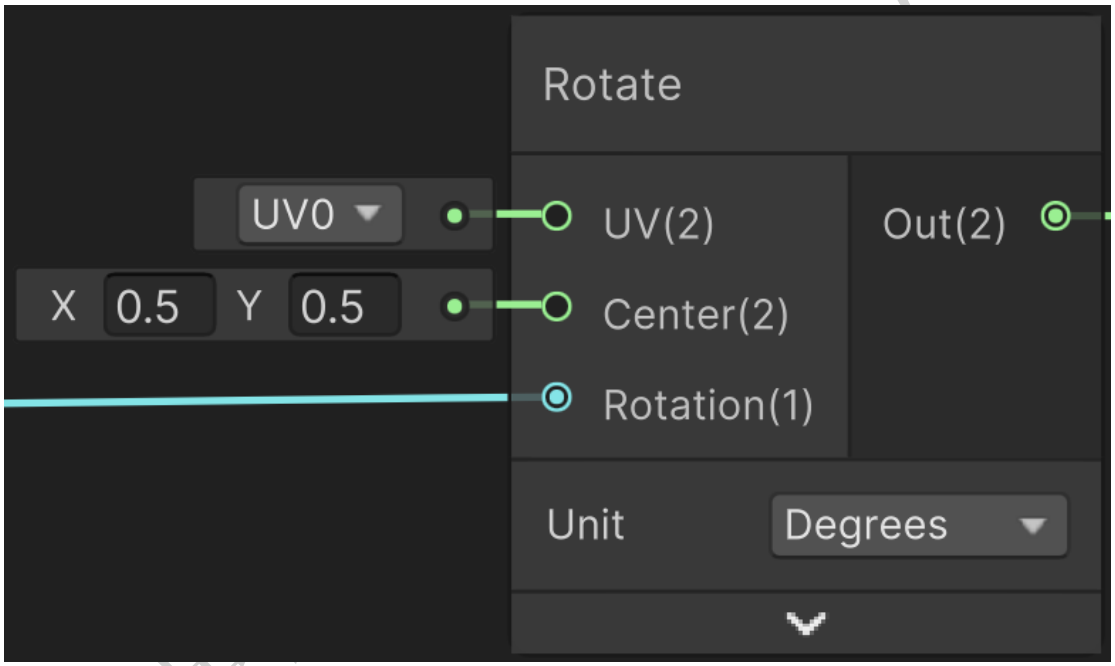


图165.　图 Rotate 节点控件

八、 2D 流光效果进阶版

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

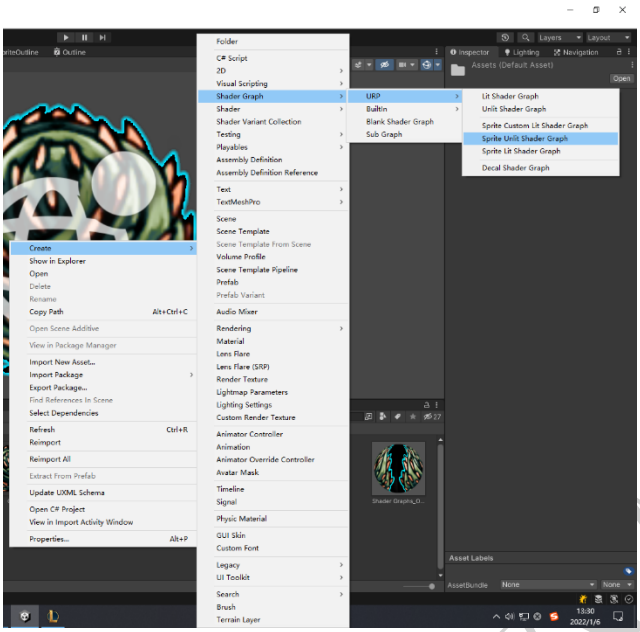


图166.　　图 7.1 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[9],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图167.　　图 7.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

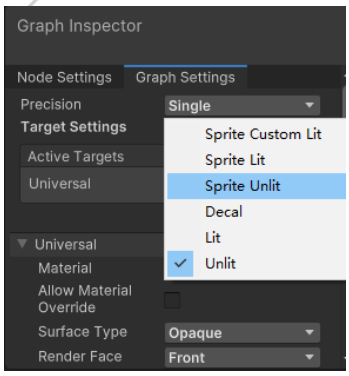


图168.　　图 7.3 修改 Material

九、　转场过渡效果-简易版

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

[9] adj. 普遍的；通用的 n. 普遍原则；普遍看法

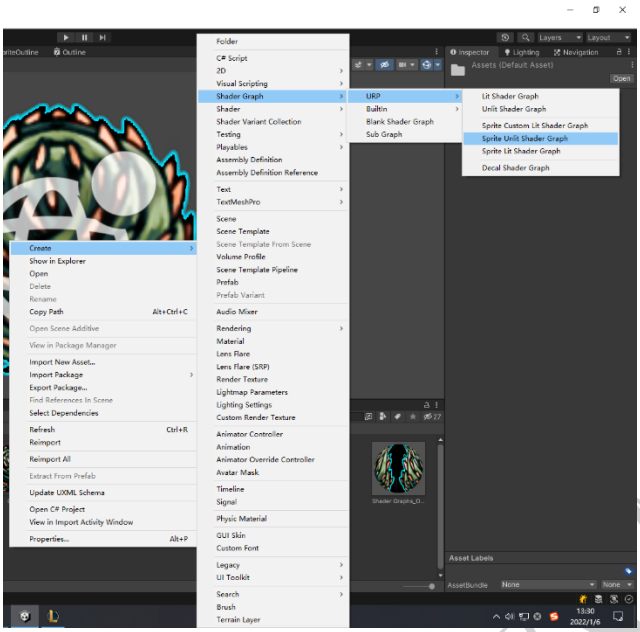


图1. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[10],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图169. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

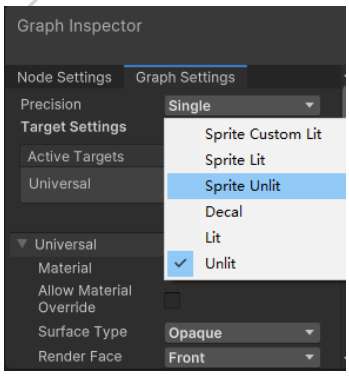


图170. 修改 Material

2. 添加可外部访问的参数

首先我们需要一个 Texture2D 类型的参数用于显示主纹理，一个 Color 类型的参数用于控制颜色。因为我们需要在外部通过脚本去控制过渡效果，因此需要一个 Float 类型的参数去控制。

在这个 Shader 中，我们将使用 Smoothstep 节点，这个节点和 Step 有些许不同，它可以实现柔性过渡，而 Step 节点是硬性过渡。所以，我们还需要一个 Float 类型的参数，去

[10] adj. 普遍的；通用的 n. 普遍原则；普遍看法

控制柔性过渡的区域。

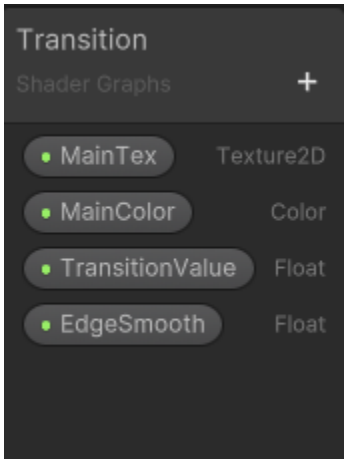


图171.　过渡效果参数

3. 实现硬性过渡

首先我们创建一个 Sample Texture 2D 节点，将主纹理参数赋予 Sample Texture 2D 节点，将主颜色参数输出到 BaseColor 端口。

接下来我们创建 Step 节点，将主纹理的 R 通道给予 Step 节点的 Edge 端口，将控制过渡效果的参数给予 Step 节点的 In 端口。

紧接着，将 Step 节点输出到 Alpha 端口。

现在就可以点击左上角的 SaveAsset，保存现在的 ShaderGraph。

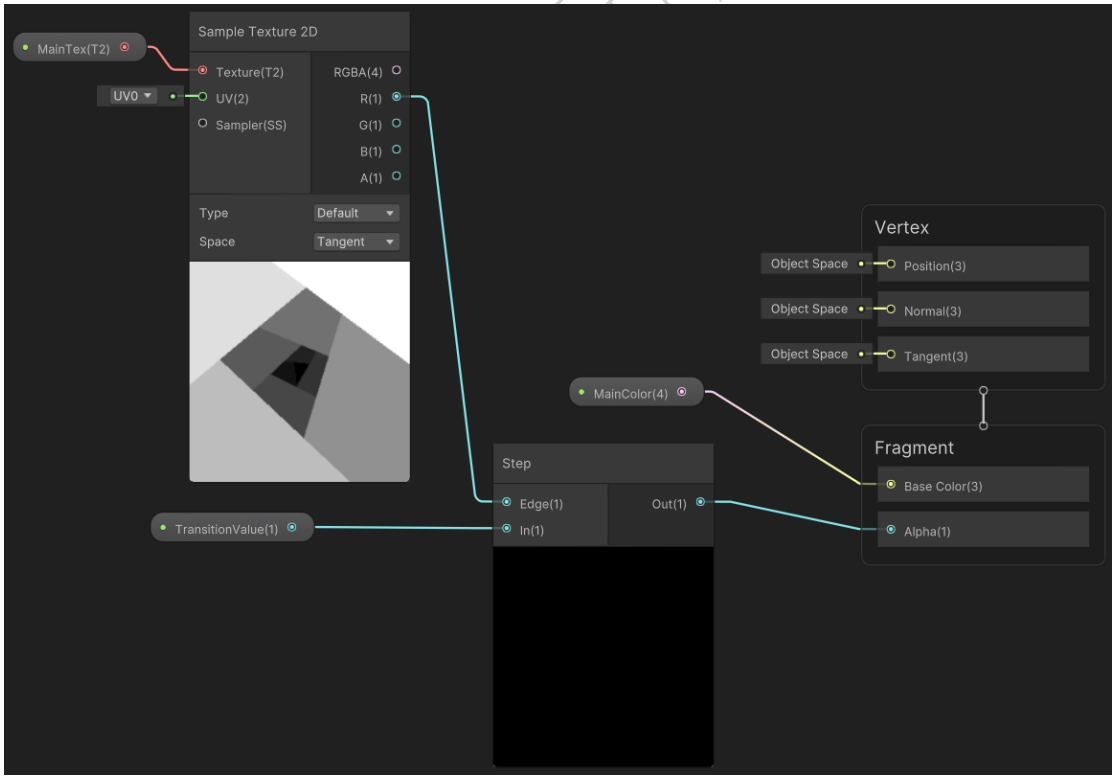


图172.　硬性过渡

4. 将硬性过渡更改为柔性过渡

虽然现在使用 step 实现了过渡效果，但是现在的过渡太过生硬，边缘不够柔和。因此我们需要使用 Smoothstep 节点去让过渡的边缘变得更加的柔和。

我们创建 Smoothstep 节点，将控制过渡效果的参数给予 Edge1 端口。使用加法节点略

微增加控制过渡效果的参数值，然后将其输入到 Edge2 端口。

现在就可以看到边缘变得柔和了。

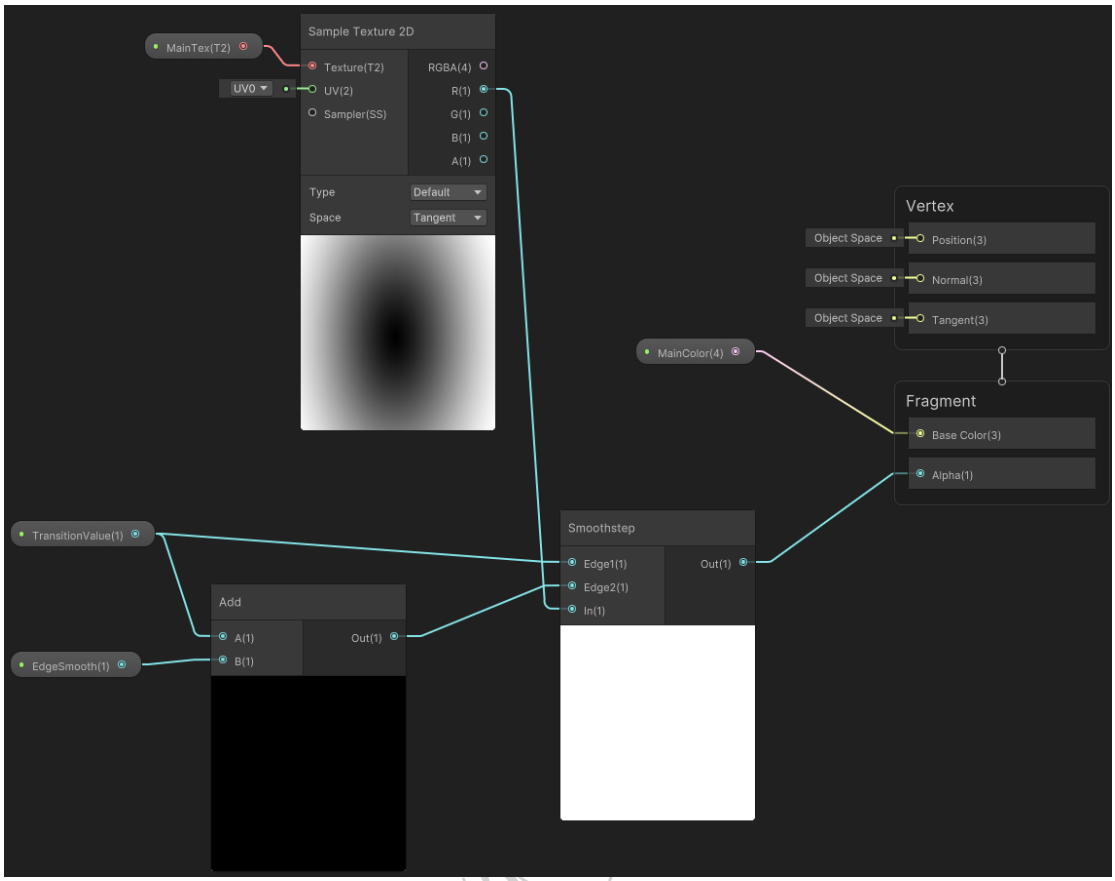


图173. 柔性过渡

十、 2D 激光效果

十一、 2D 帧动画

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

哔哩哔哩

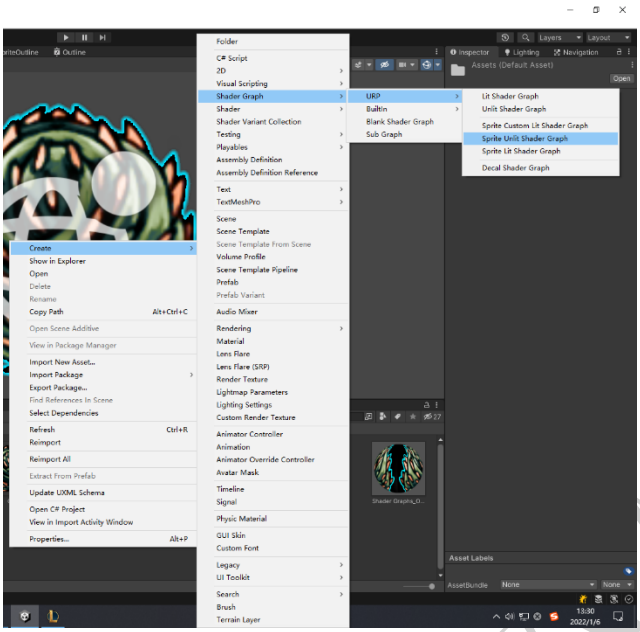


图2. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[11],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图174. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

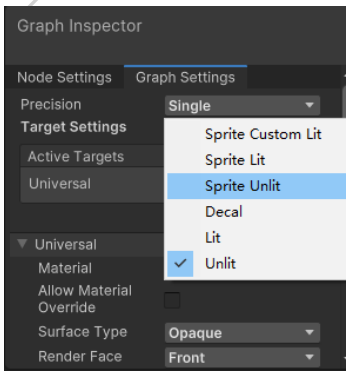


图175. 修改 Material

2. 添加主纹理

添加一个 Texture2D 类型的参数，并将其命名为 MainTex。然后创建一个 Sample Texture 2D 节点，将主纹理输出到 BaseColor 端口和 Alpha 端口。

[11] adj. 普遍的；通用的 n. 普遍原则；普遍看法

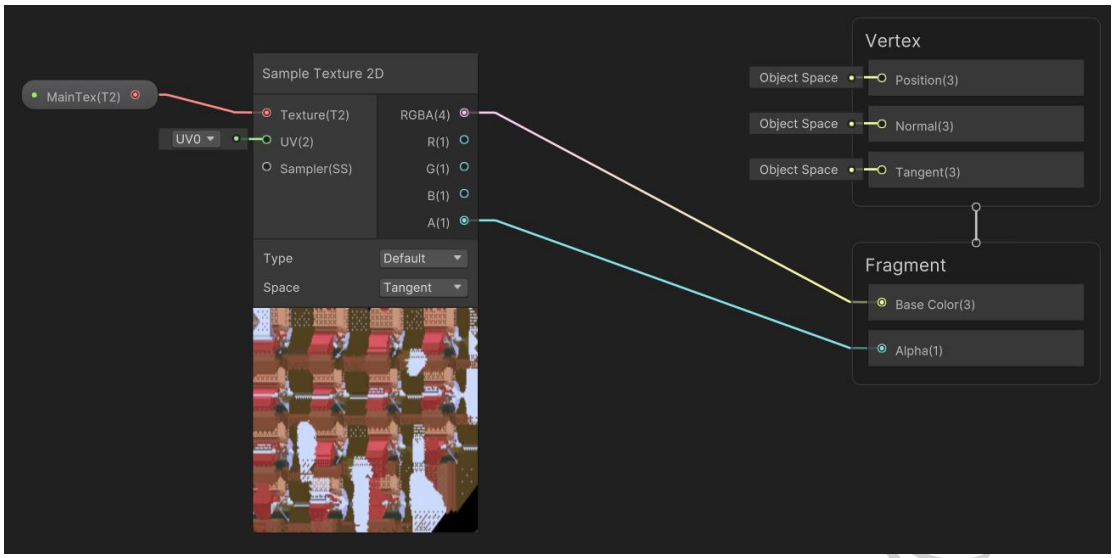


图176.　2D 帧动画主纹理

3. 分割主纹理

创建一个 Flipbook 节点，来对图集进行切分。Flipbook 节点的 Width 端口和 Height 端口，分别代表切分出的行数和列数，切分是根据图集的宽和高来进行计算。Flipbook 节点切分的瓦片，相当于存储在一个数组里，我们使用下标去访问这些瓦片，即 0 代表的是切分出的第一张瓦片。

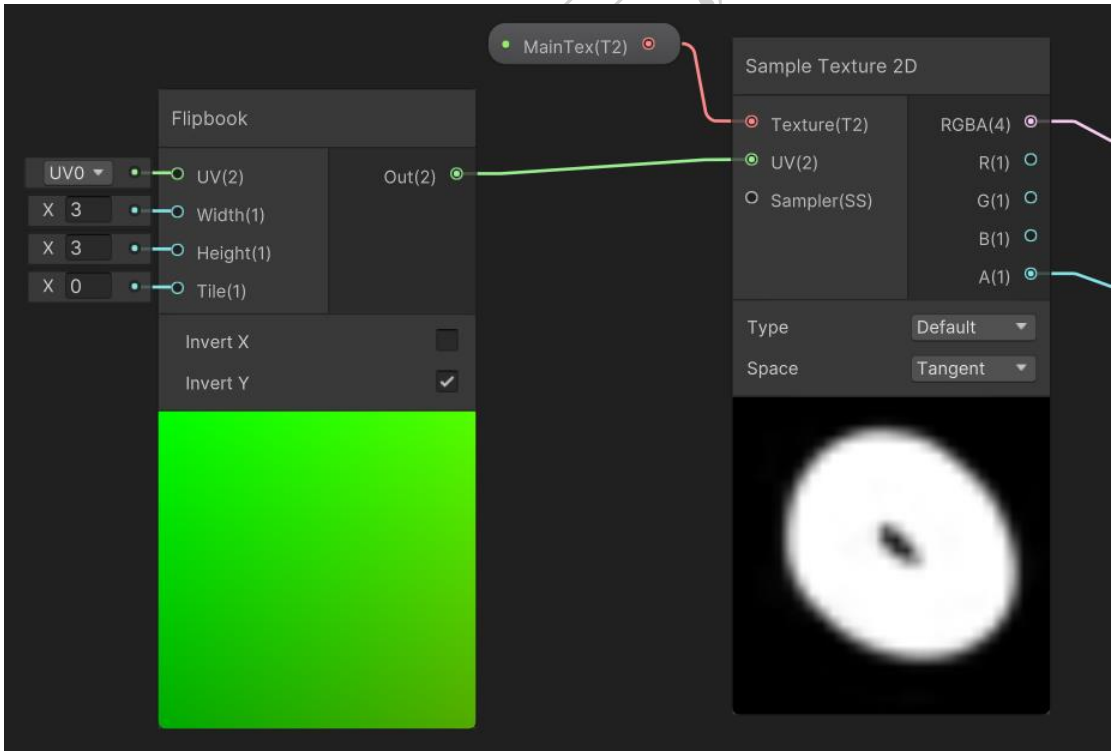


图177.　使用 Flipbook 节点

4. 让图集动起来

知道了切分的瓦片是如何访问的，我们就可以尝试使用 Time 节点，来让瓦片图像依次显示。

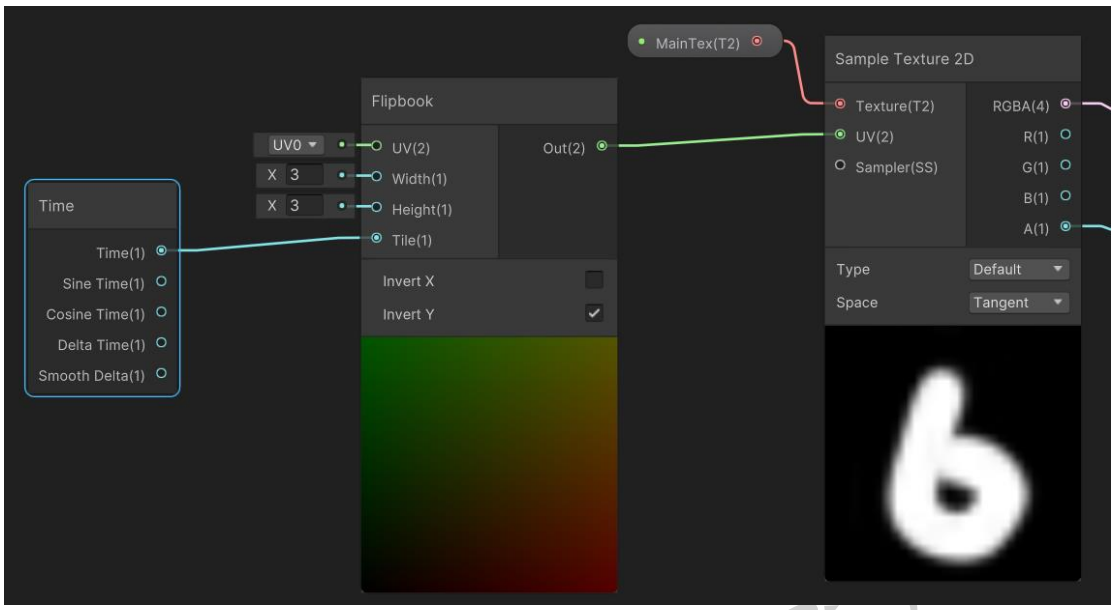


图178.　　使用 Time 节点让瓦片依次显示

因为 Time 节点的输出不是一个整数，而瓦片所对应的下标是整数，因此我们需要将 Time 节点的输出取整。我们创建一个 Floor 节点，来对 Time 节点的输出进行向下取整。

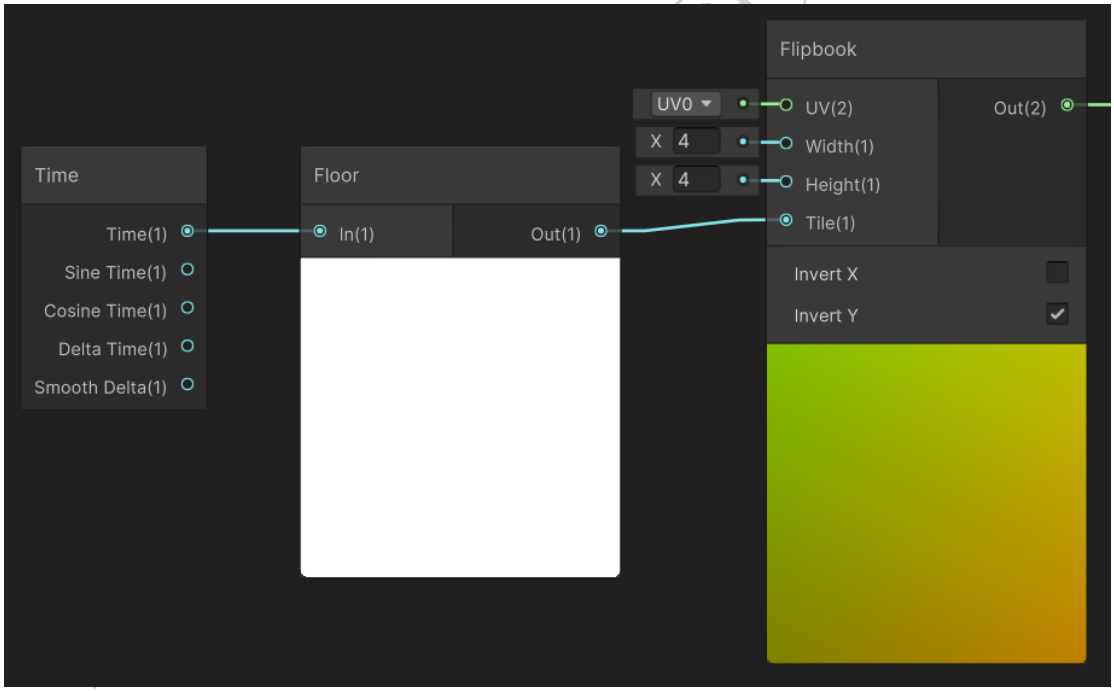


图179.　　对 Time 节点输出取整

5. 设置图集的最大可显示帧

如果我们进行切分后的图集没有空白区域，那么现在就可以正常进行帧动画了。但是，如果切分后的图集有空白部分，那么在进行帧动画的时候会将其也显示出来，这会破坏帧动画的循环效果。

恰好，这部分空白区域往往是在最后的部分。因此，我们标记出可显示的最大瓦片数，然后利用这一个值对 Time 节点的输出进行取余。

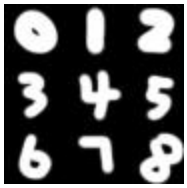


图180.　测试图

就如上图，其最大瓦片数为 9。如果我们让 Time 节点的输出对 9 取余，那么循环的数字就是 0 到 8；如果我们让 Time 节点的输出对 8 取余，那么循环的数字就是 0 到 7。

我们创建一个 Modulo 节点，对 Time 节点的输出进行求余，并使用一个整型参数去设置瓦片的最大数量。

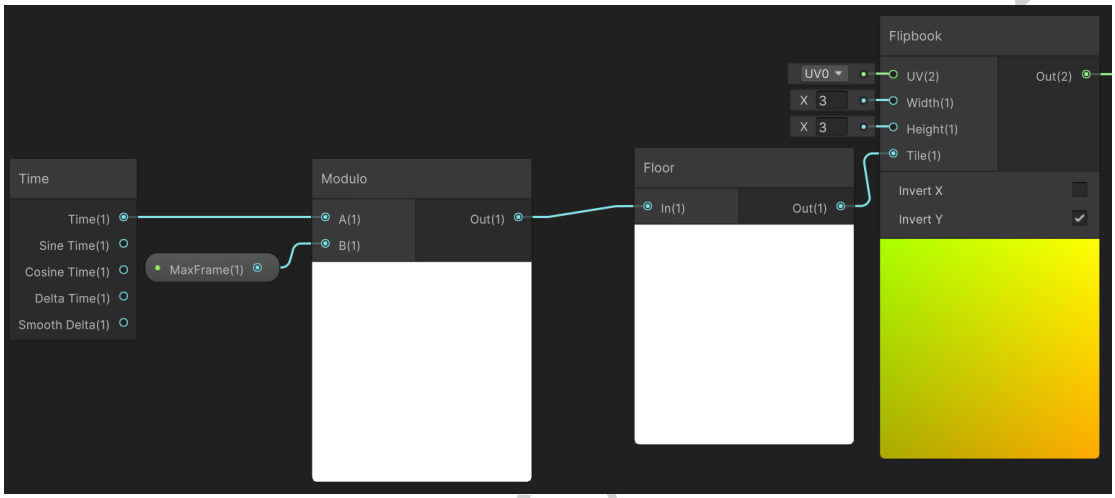


图181.　对 Time 节点输出取余

注意，取余一定要在取整之前。

在 2021.2.5 版本的 URP 中，创建的默认是 Float 型，需要在 Graph Inspector 中将其修改为整型。

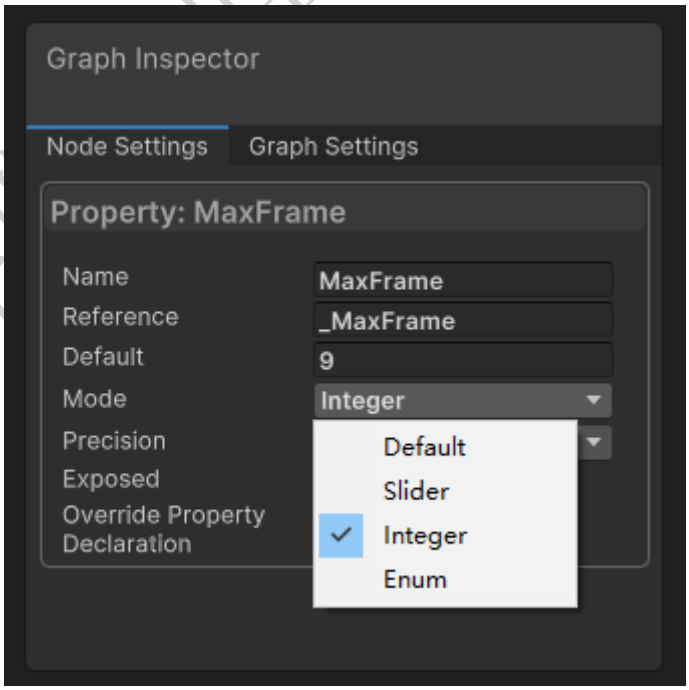


图182.　修改参数类型为整型

6. 调整帧动画速度

我们的帧动画不可能一直是这个速度，为此我们需要一个参数去调节动画速度。依然创建一个整型的参数，帧动画往往是 24、30、60 这些整数，设置为浮点型也是可行的。

创建一个乘法节点，将 Time 节点和速度参数相乘。

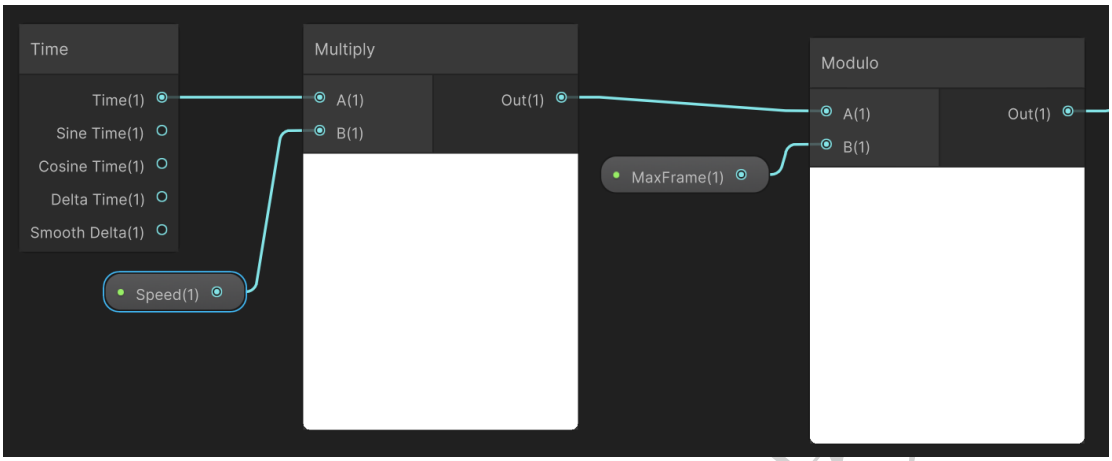


图183. 调节帧动画速度

7. 添加额外参数

我们需要在外部设置图集的行数和列数，为此需要两个整型参数去控制。当然，设置为一个二维向量也是可行的。

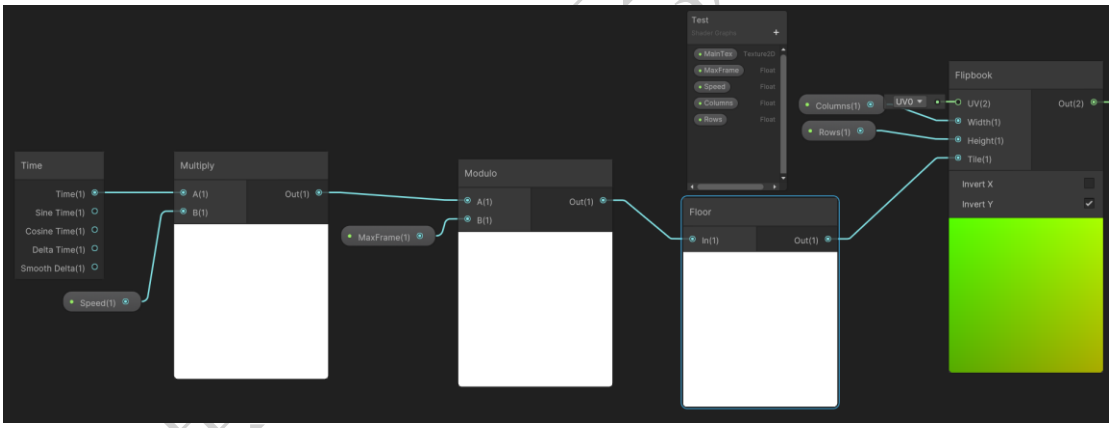


图184. 添加行和列参数

8. 可能存在的问题

当使用 Sprite 组件去显示时，显示出的可能并不是一张完整的图像。

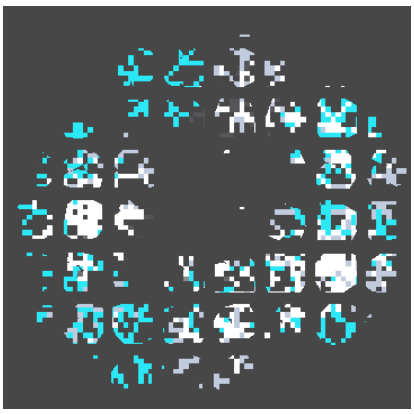


图185. 显示图形出错

要让图片正确显示，我们需要修改该纹理的 Mesh Type 为 Full Rect。

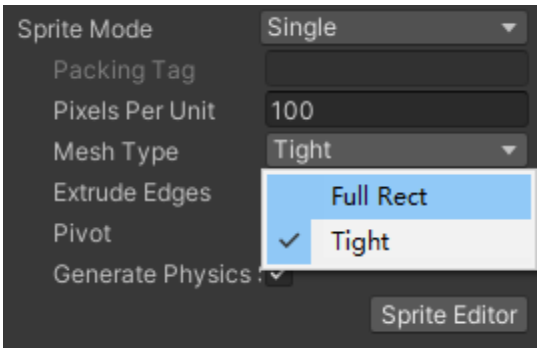


图186.　修改 Mesh Type

十二、　后处理速度线

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

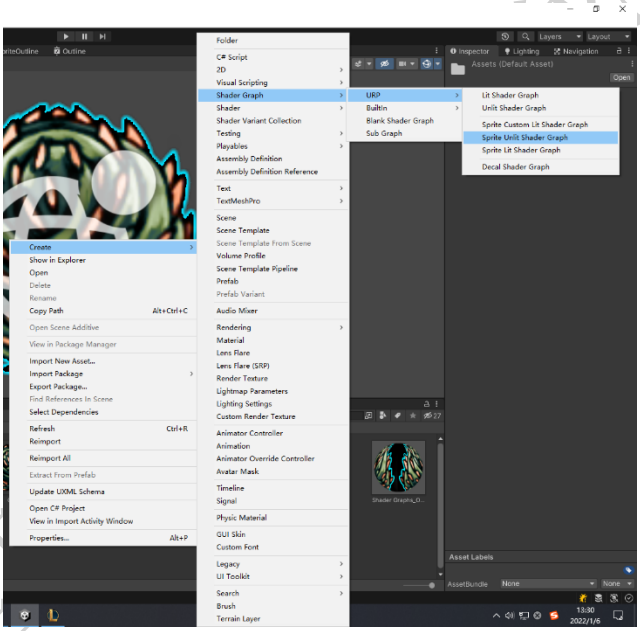


图3. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[12],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图187.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

[12] adj. 普遍的；通用的 n. 普遍原则；普遍看法

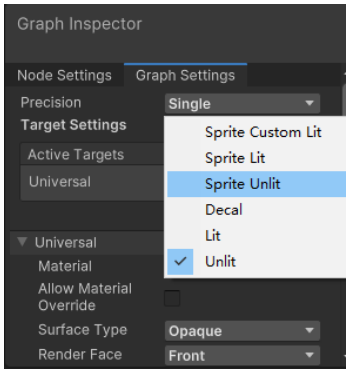


图188.　　修改 Material

2. 制作速度线

速度线我们使用 GradientNoise 配合极坐标实现。首先创建一个 GradientNoise 和 Polar Coordinates 节点，将极坐标节点的输出给予 GradientNoise 的 UV 端口，我们就会看到 GradientNoise 的图像变了。我们将极坐标节点的 Radial Scale 调小，就会发现已经具有速度线的雏形了。

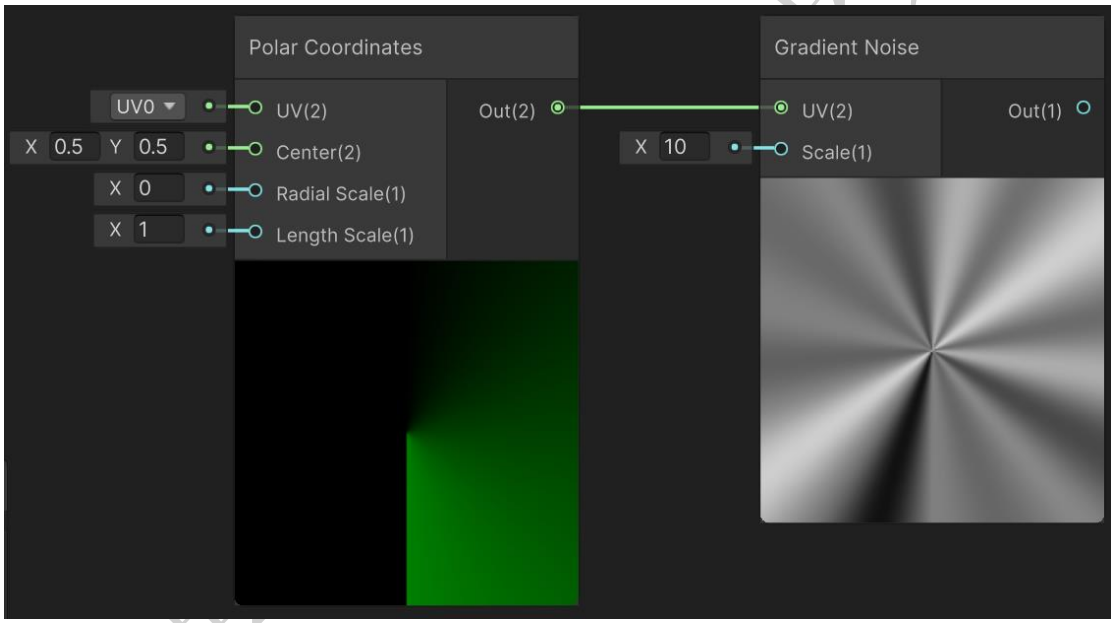


图189.　　速度线雏形

接下来我们再创建一个 Inverse Lerp 节点和 Slider 节点，Inverse Lerp 是计算出 T 值在 A 到 B 值之间所占比例。其运算就是均匀分布的分布函数，即 T 减去 A 的值，除以 B 减去 A，利用运算特性，我们可以对显示的速度线进行一个过滤。

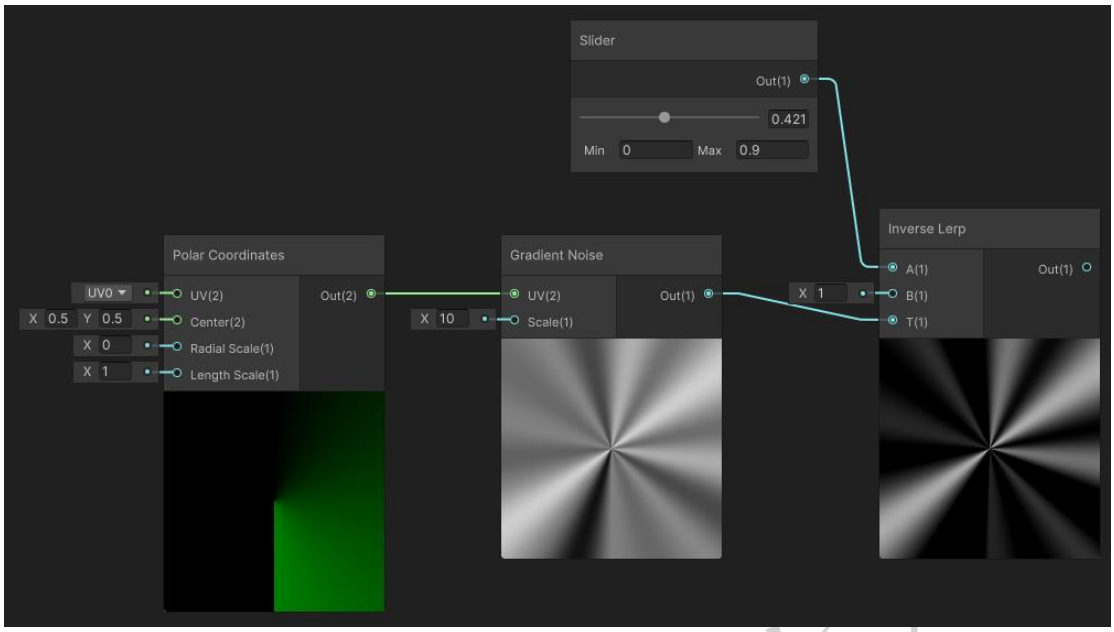


图190.　　对速度线图像进行过滤

因为数学运算的原因，输出的值可能存在负的情况，这是不对的。因此我们需要对输出结果进行一个范围限制，因此我们添加一个 Saturate 节点，将值规定在 0 和 1 之间。

3. 让速度线流动起来

现在已经有了速度线，我们就可以考虑让它流动起来了，让速度线流动起来需要使用 Time 节点。但是在使用 Time 节点之前，我们需要弄清楚速度线的 UV 流动方向。我们创建一个加法节点和一个二维向量，将极坐标的输出与这个二维向量相加。

我们知道 Polar Coordinates 节点的输出是一个二维向量，X 分量是半径，Y 分量是角度。当我们增加 X 分量，会看到我们的速度线移动是向中心聚拢的，因此我们可以明确最终输出到这个分量的值应该是一个负值。

我们创建一个 Time 节点、乘法节点和一个 Float 节点，因为我们知道输入 X 分量的值是负数，因此在 Float 和 Time 节点相乘前需要使用 Negate 节点对 Float 值进行取反。当然，直接将 Float 值设定为负值也可以，只是这里的 Float 值代表是运动的速度，在我们的认知中速度一般是大于等于 0 的，正负号通常代表方向，所以显示为正值比较好。

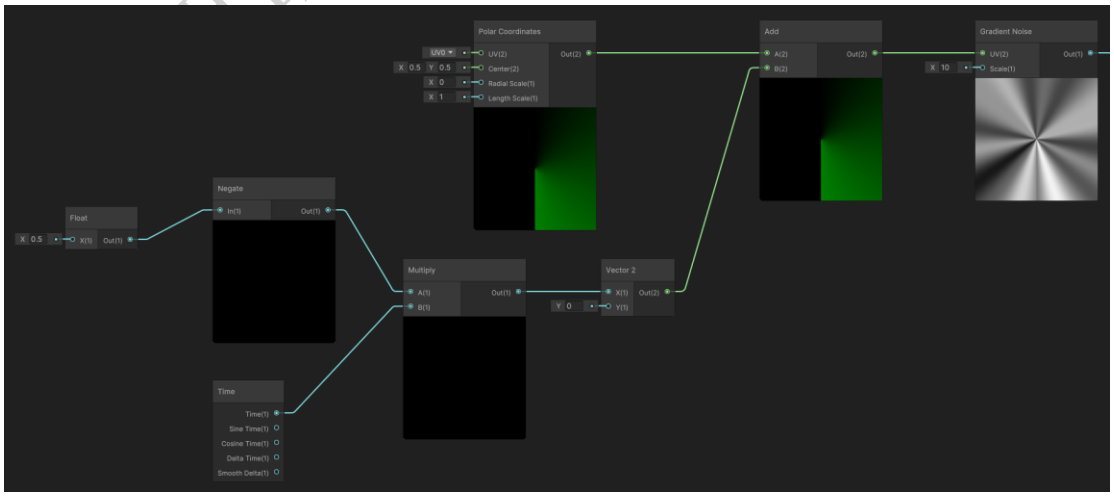


图191.　　速度线运动

4. 调整速度线的 Scale 和 Tiling

现在速度线已经可以进行运动了，但是我们的速度线不可能一直是这个样子，因此我

们创建一个 Float 节点和一个 Slider 节点，用它们去调整速度线的 Scale 和 Tiling。

Polar Coordinates 节点的 Radial Scale（径向比例）和 Length Scale（长度比例），分别控制的是速度线的 Scale 和 Tiling。而 Scale 的值最好不要大于 1，否则速度线的图像不好看且看起来很假，因此我们用 Slider 节点控制；而 Tiling 则用 Float 节点控制。

5. 制作蒙版

下面我们就开始制作蒙版了，蒙版将决定速度线的显示区域。首先创建一个 Lerp 节点和一个 Slider 节点，Lerp 节点会根据 T 值，返回在 A 和 B 之间线性插值的结果。我们将 Lerp 节点的 A 值设定为 2，B 值设定为 0。

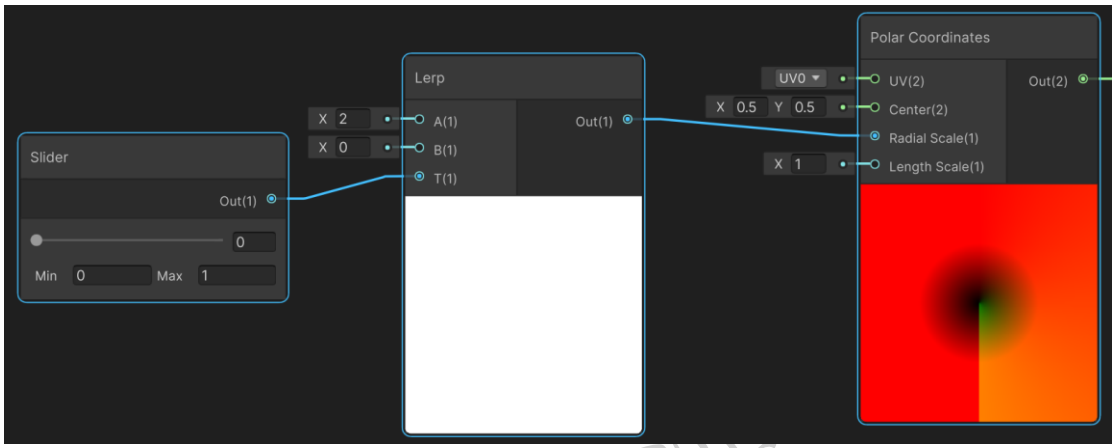


图192.　Mask 显示规模

接着创建一个 Polar Coordinates 节点，将 Lerp 的结果输如到极坐标的 Radial Scale 端口。调整 Slider 节点的值，就可以看到极坐标的区域发生了变化。

接下来我们使用 Split 节点将极坐标的 R 通道分离出来，并使用 SmoothStep 节点去调整边缘值，使边缘变得平滑。如果 In 的值小于 Edge1 的值，返回 0，In 的值大于 Edge2 的值，则返回 1 。In 值位于两者之间，则在 0 到 1 之间进行差值。

我们创建一个 Slider 节点，去控制 Edge2 的值，以便于调整蒙版的最终区域。

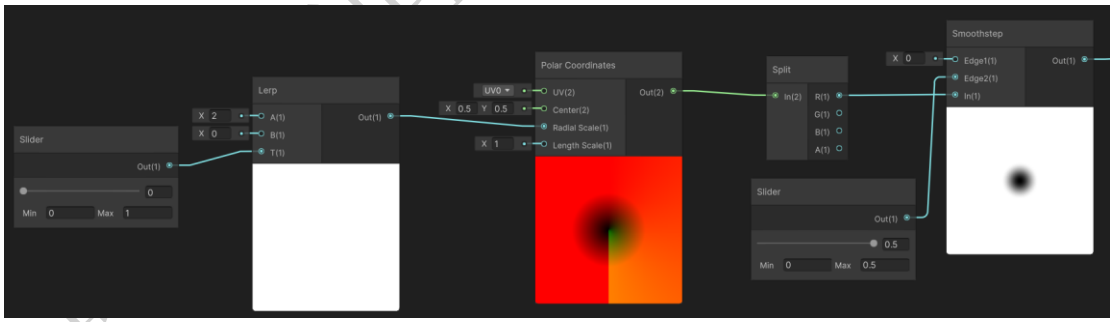


图193.　速度线蒙版

6. 最终输出

将蒙版和速度线相乘，然后创建一个 Texture2D 参数和一个 Sample Texture2D 节点，并将 Texture2D 参数命名为 MainTex。

将蒙版后的速度线和主纹理图像相加，输出到 Base Color 端口。

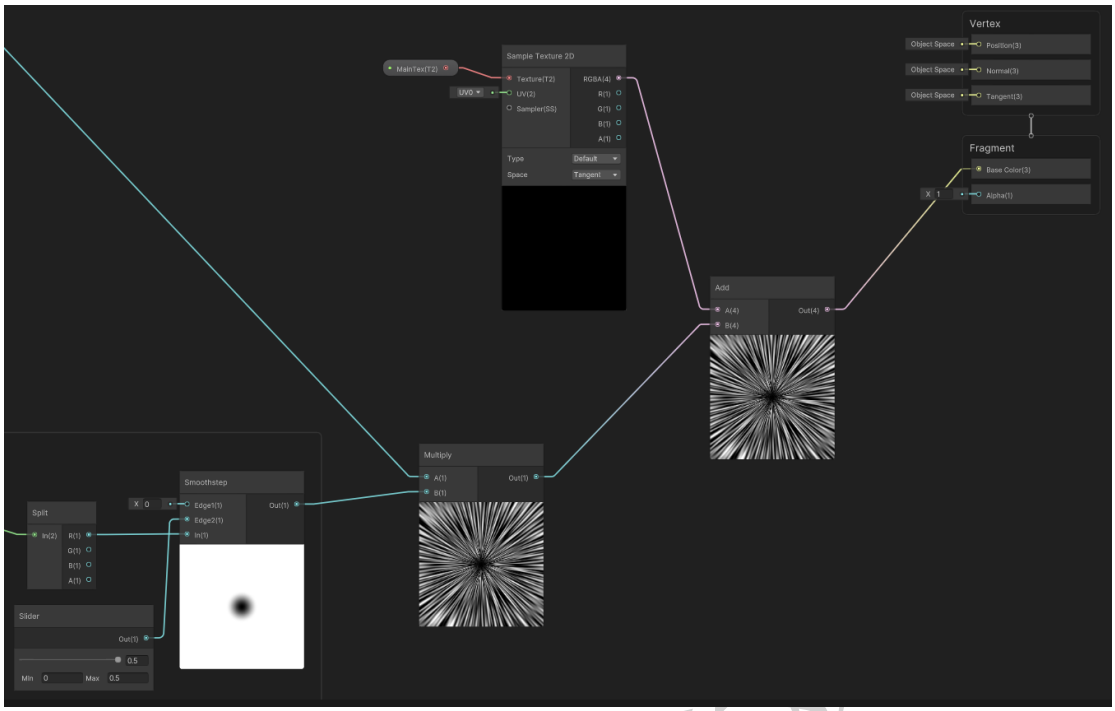


图194.　　速度线最终输出

7. 将节点转换为参数

在 ShaderGraph 完成后，我们就需要将可外部修改的部分转换为参数暴露出去。要暴露参数，只需要在该节点上单击鼠标右键，找到 Convert To->Property。

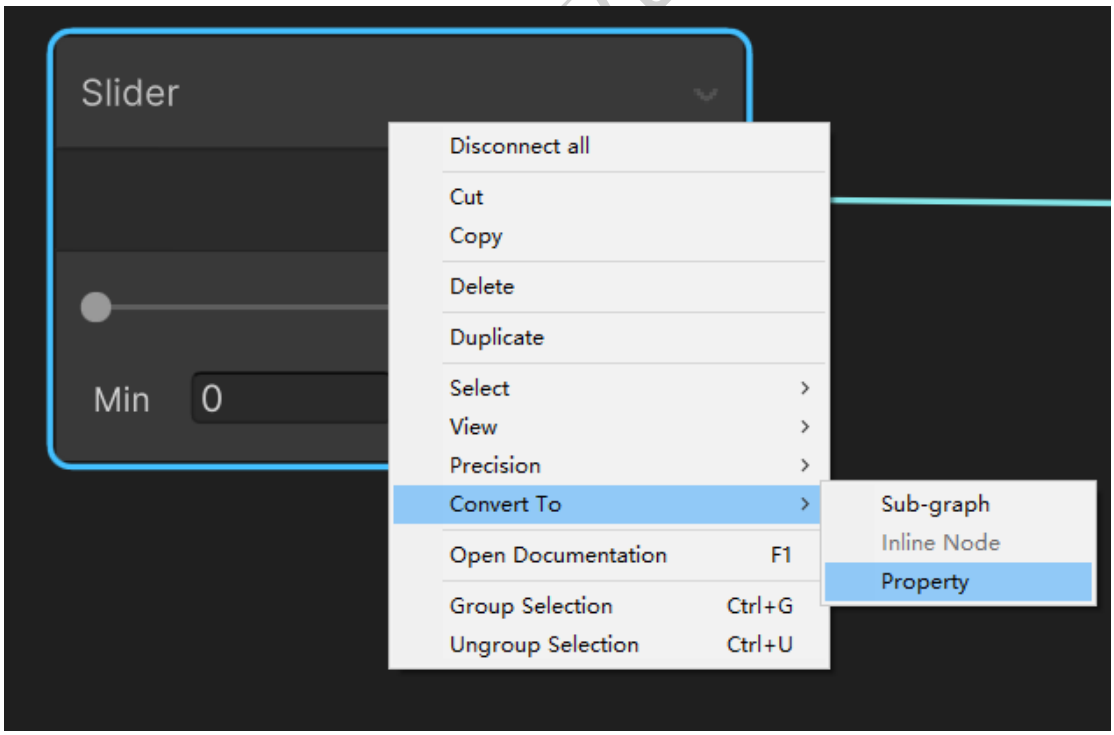


图195.　　转换为参数

8. 后处理运用

十三、 2D 像素效果

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

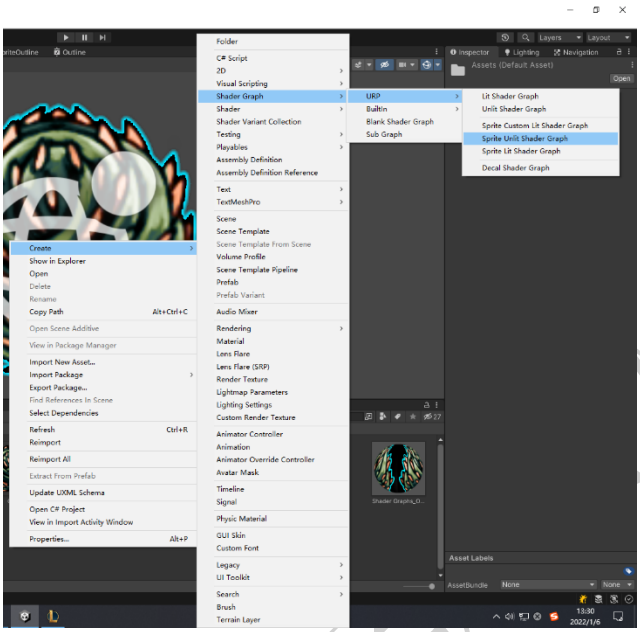


图4. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[13],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图196. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

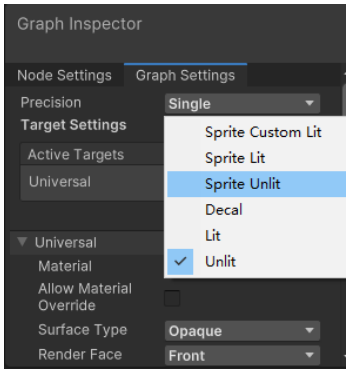


图197. 修改 Material

[13] adj. 普遍的；通用的 n. 普遍原则；普遍看法

2. 原理解释

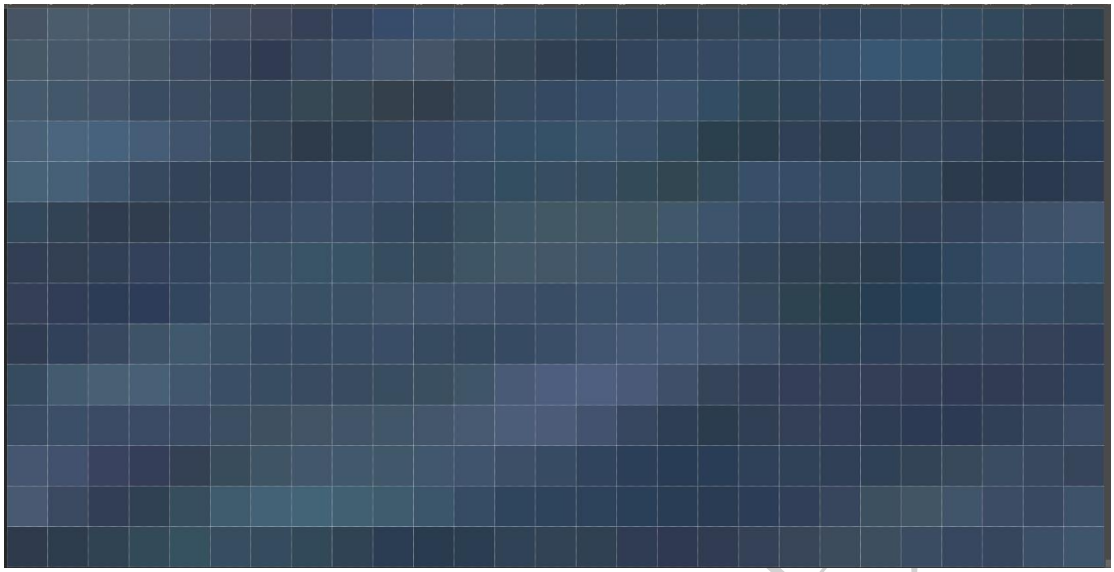


图198. 像素点

如上图所示，我们的图像是由一个个像素点组成的，而这些像素点在 UV 中都具有一个独立的坐标。

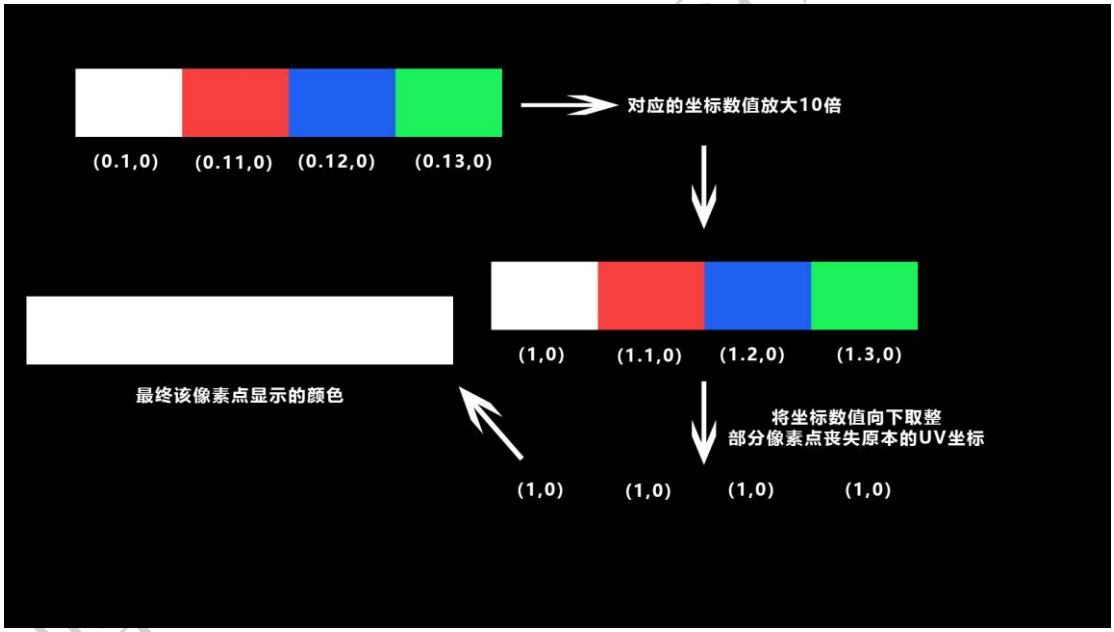


图199. 像素化原理

我们知道，每一个像素点都对应着一个 UV 坐标，因为 UV 坐标是在 0 到 1 这个范围内的。那么必定有像素点所对应的 UV 坐标是有小数的，我们将所有像素点的 UV 坐标放大，然后再对放大后的 UV 坐标进行向下取整。那么部分像素点将失去原本所对应的 UV 坐标，而被另一个像素点的 UV 坐标所代替。因为 UV 坐标范围是在 0 到 1 之间，我们必须再重新将 UV 的值变回到这个范围，因此必须除以放大的倍数。

在渲染的时候，部分像素点所显示的颜色，将不再是原本的颜色，而是变为新的 UV 坐标所对应的像素点的颜色。

3. 修改图像对应的 UV 坐标

首先我们需要创建一个 UV 节点，我们依据前面所述的原理，利用乘法节点将 UV 的数值放大，然后对放大后的 UV 进行向下取整，最后再除以相同的放大倍数，将 UV 坐标

重新标准化。

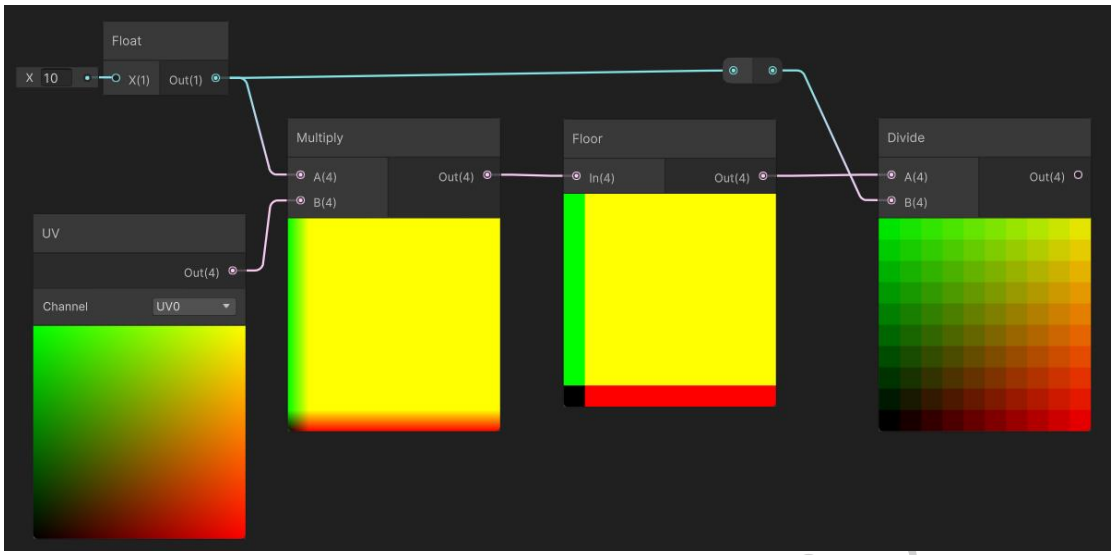


图200.　　修改 UV 坐标

接着我们添加一个 Texture2D 类型的参数，以及一个 Sample Texture 2D 节点，然后将 Texture2D 类型的参数命名为 MainTex。将重新标准化后的 UV 输入到 Sample Texture 2D 节点的 UV 端口，再将 Sample Texture 2D 节点输出到 Base Color 端口和 Alpha 端口。

现在我们就可以预览到像素化的效果了。

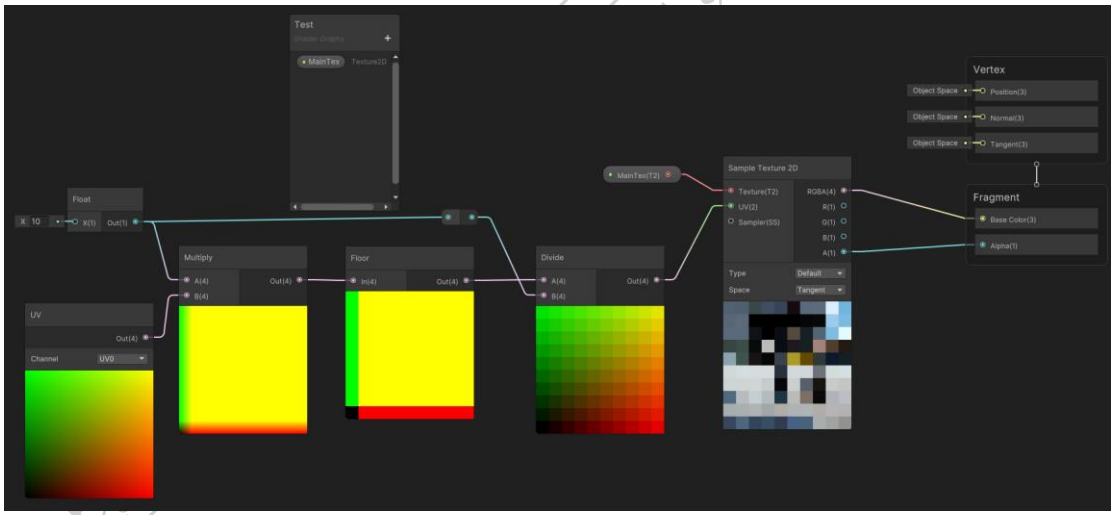


图201.　　输出

4. 使用 Slider 控制像素化效果

为了便于控制像素化的效果，我们创建一个 Float 类型的参数，将它的模式修改为 Slider，这样我们在外部就可以通过滑动条去控制。

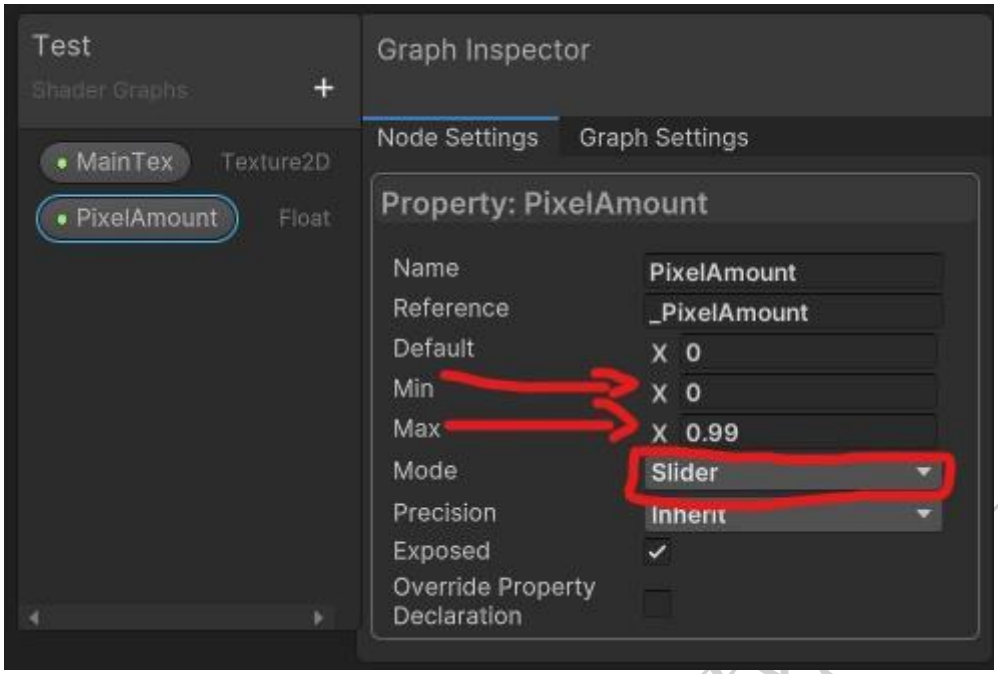


图202.　　修改参数模式为 Slider

因为我们现在的这个参数是介于 0 到 1 之间的，但是我们是使用这个参数来放大 UV 的，因此我们需要在它放大 UV 之前先乘上一个较大的值。然后将结果和 UV 相乘，并用这个结果将 UV 重新标准化。

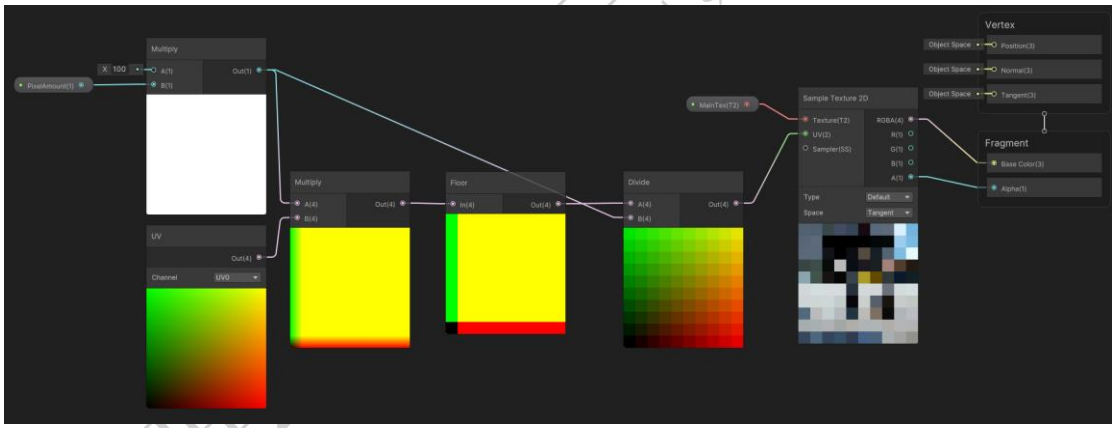


图203.　　使用参数控制

现在我们修改参数就可以看到像素化效果强度会发生改变，值越接近 0 强度越大，越接近 1 强度越小。如果我们希望值为 0 时的像素化效果弱一点，值为 1 时像素化效果强一点，那么我们就可以用 1 减去当前参数值。

在 ShaderGraph 中可以使用 One Minus 节点实现，也可以用减法节点实现。（使用减法节点需要注意顺序）

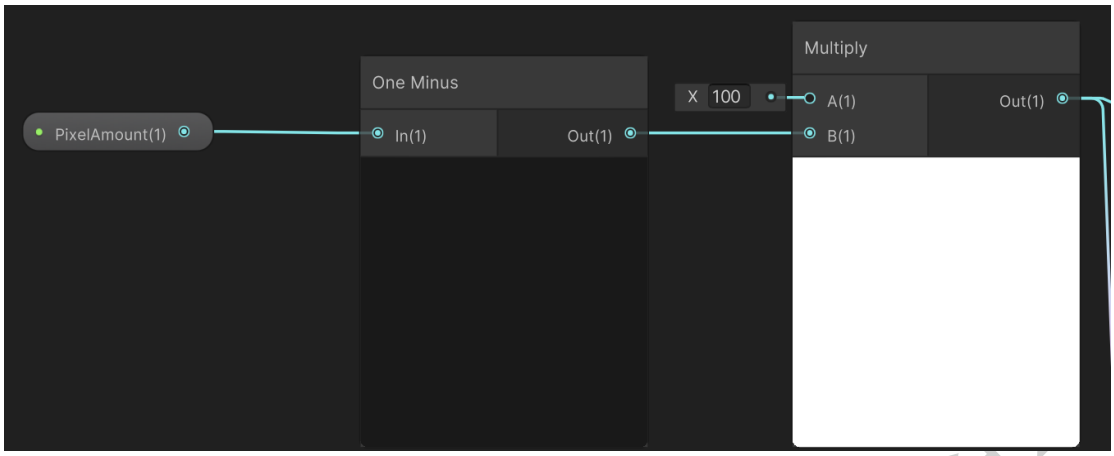


图204.　　使用 OneMinus 修改参数

5. 设置是否应用像素化

创建一个 Boolean 类型的参数，用这个参数去控制是否应用像素化效果。

接着创建一个 Branch 节点，将 Boolean 参数赋予 Branch 节点，然后将未修改的 UV 连接到 Branch 节点的 false 端口，将修改后的 UV 连接到 true 端口。当 Boolean 参数为 false 时，就不会应用像素化效果，只有为 true 时才会应用像素化效果。

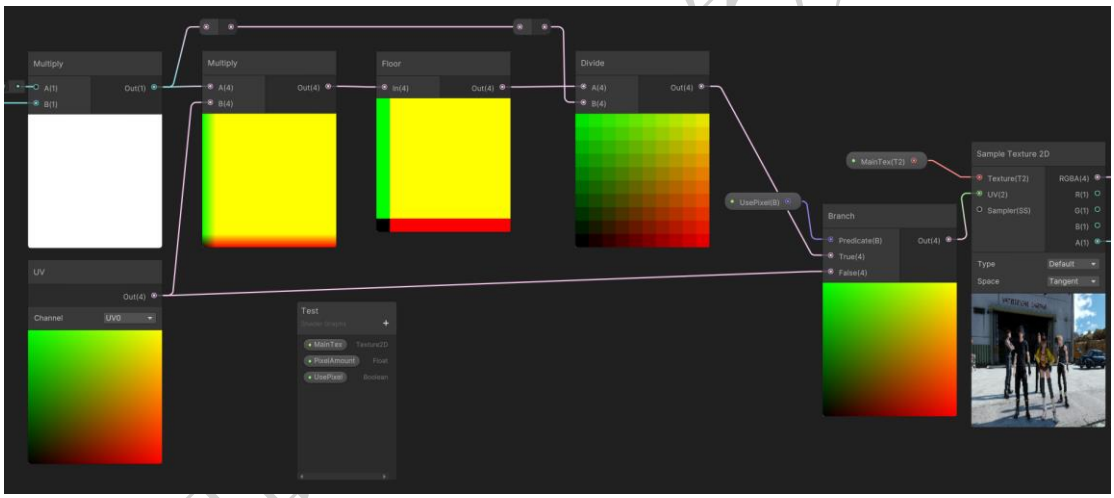


图205.　　使用 Branch 确定是否使用像素化效果

6. 自定义像素化行列（拓展）

现在我们像素化所显示的行数和列数都是相同的，如果我们要自定义像素化显示的行数和列数，那么我们就可以使用两个参数去分别控制二维向量的 X 分量和 Y 分量。其中，X 分量代表列数，Y 分量代表行数。

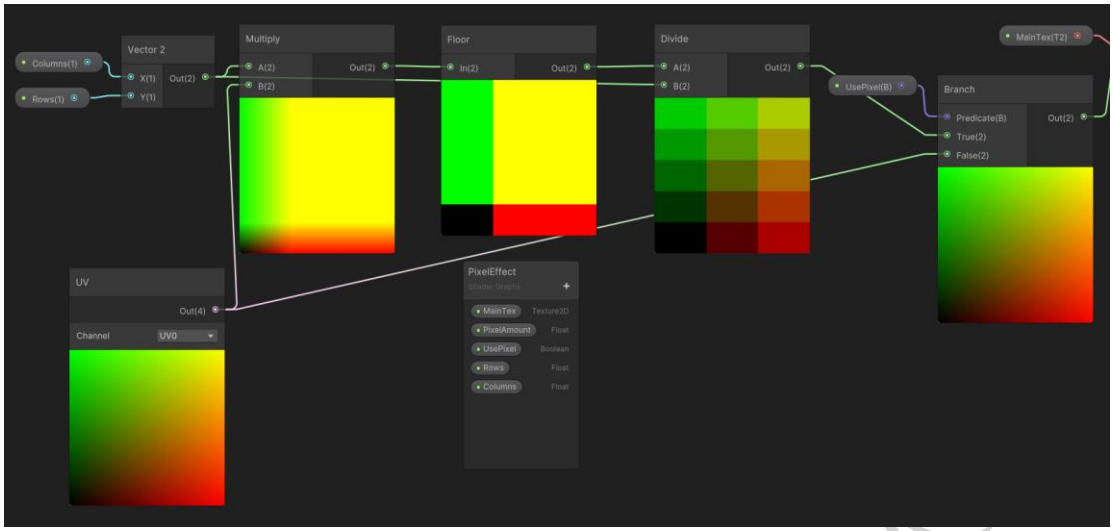


图206.　自定义像素化行数和列数

十四、　2D 植被飘动

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

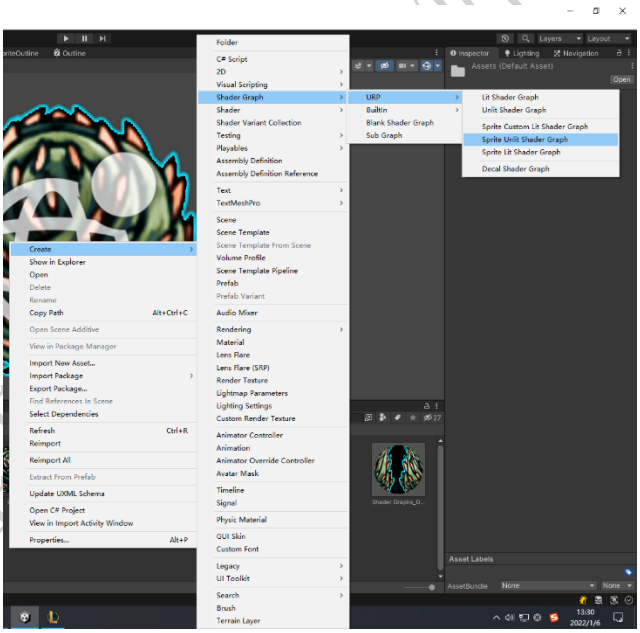


图5. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[14],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图207.　Graph Inspector 面板按钮

[14] adj. 普遍的；通用的 n. 普遍原则；普遍看法

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

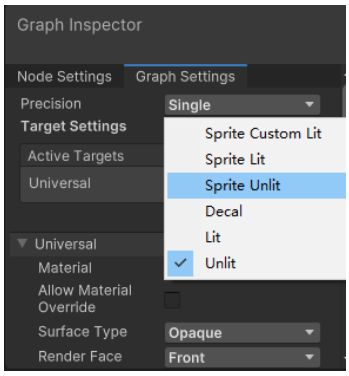


图208.　　修改 Material

2. 原理

通过顶点着色器，对顶点进行一个坐标变换，通过这一操作可以实现顶点动画。

3. 将主纹理输出到片元着色器

创建一个 Texture2D 类型的参数，并将其命名为 MainTex。然后创建一个 Sample Texture 2D 节点，将主纹理给予它，接着将其输出到片元着色器的 Base Color 端口和 Alpha 端口。

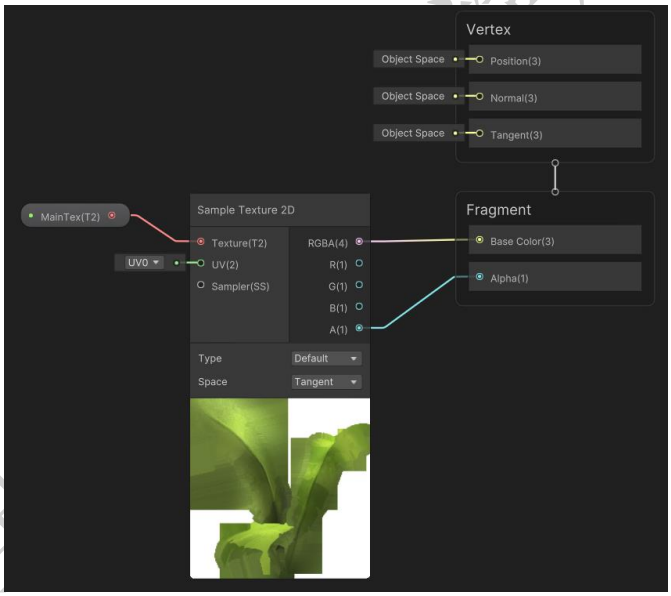


图209.　　主纹理输出

4. 让顶点位置发生偏移

首先创建一个 Position 节点，并使用绝对世界空间。因为们这里是 2D 项目，因此需要使用 Split 节点将三维向量的 X 分量和 Y 分量剥离出来，组合成一个二维向量。

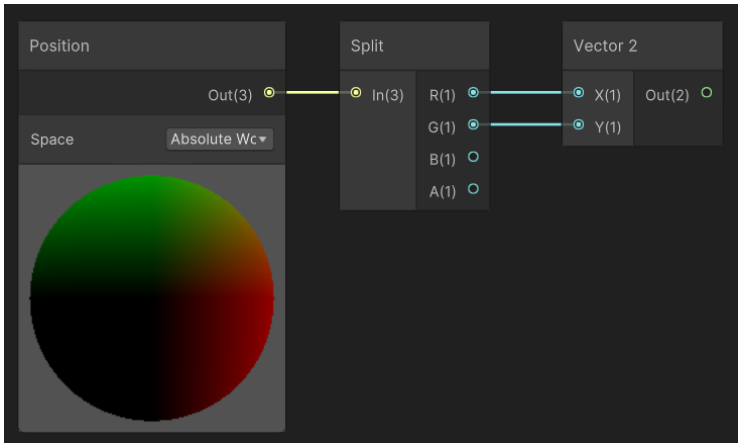


图210.　剥离 X 分量和 Y 纹理

要进行偏移，我们首先需要想到的就是 Tiling And Offset 节点，我们将分离出的二维向量输入到 Tiling And Offset 节点的 UV 端口。然后我们创建一个 Float 类型和一个 Vector2 类型的参数，其中 Float 类型的参数控制风速，二维向量控制风向。当然，也可以使用两个 Float 类型的参数，去组合一个自定义的二维向量。

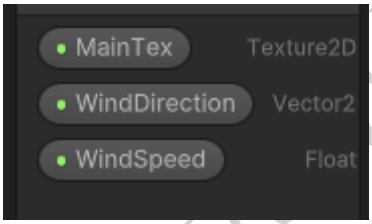


图211.　添加风速和风向参数

我们创建一个 Time 节点和两个乘法节点，先用 Time 节点的 Time 端口和风速相乘，然后再将所得值和风向相乘。因为风向只关注方向不关心大小，因此在和风速相乘前，我们需要使用 Normalize 节点，将这个二维向量进行标准化。

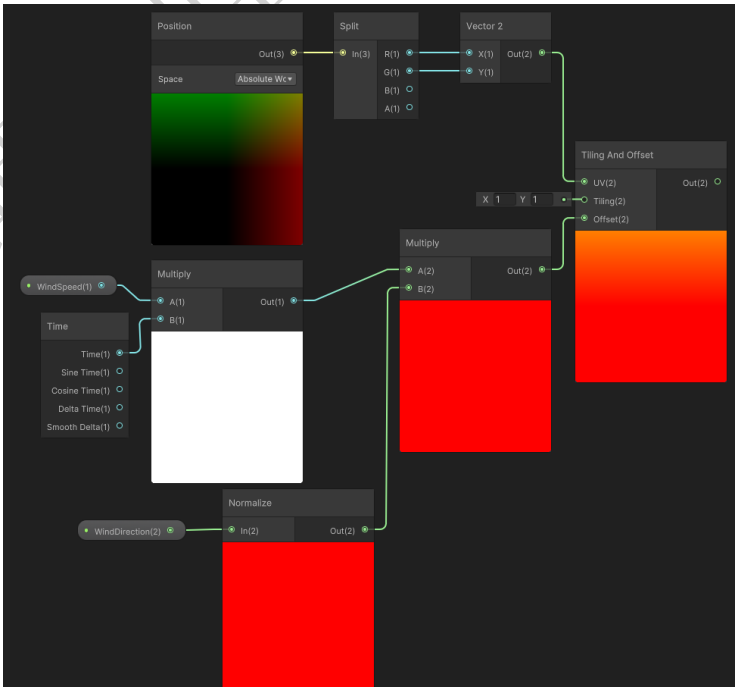


图212.　进行偏移

5. 使用噪声调整偏移强度

创建一个 Gradiet Noise 节点，将偏移结果输入到这个节点。因为我们需要在外部调整噪声的规模，所以需要创建一个 Float 类型的参数去调整它。

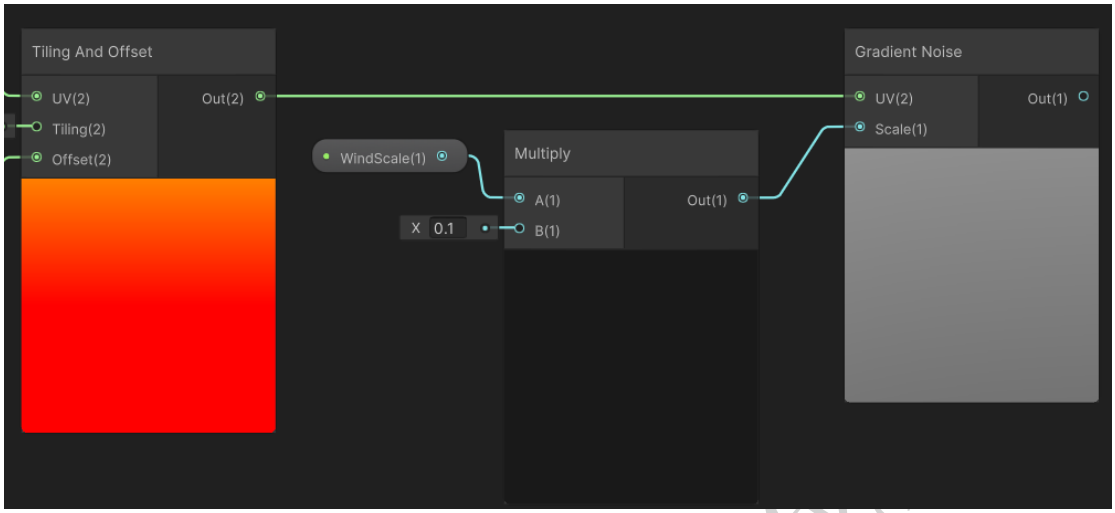


图213. 使用噪声调整偏移强度

6. 对顶点应用噪声偏移

还是和之前的一样，因为这是 2D 图像，我们只希望影响 X 和 Y。因此需要确保输出的二维向量，而不是三维向量。

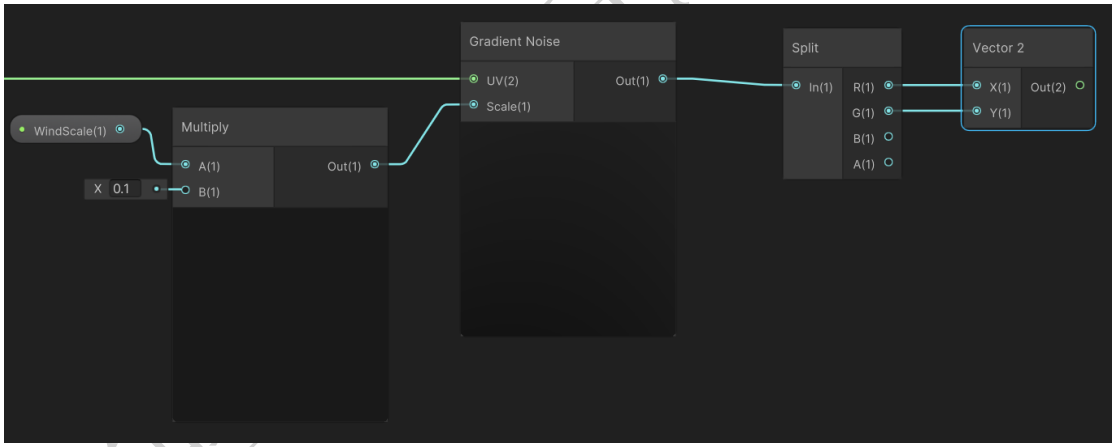


图214. 确保输出是二维向量

接下来创建一个 Position 节点，依旧使用绝对世界空间。用加法节点将顶点的噪声偏移和 Position 节点相加，然后使用 Transform 节点，将绝对世界空间转换为对象空间，输出到顶点着色器的 Position 端口。

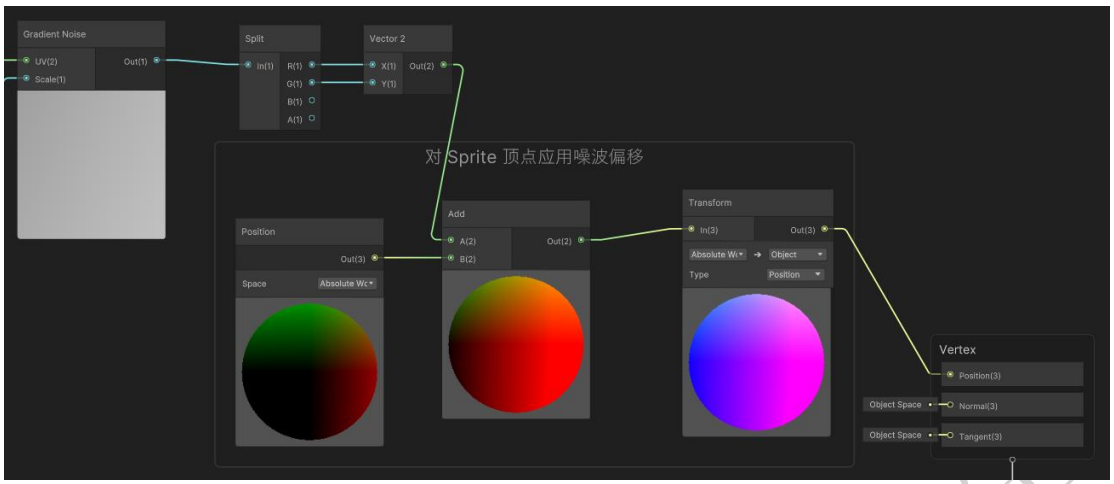


图215.　　对顶点应用噪声偏移

现在就可以创建一个 ShaderGraph 实例，在 Scene 窗口查看其偏移效果了。

7. 修改噪声视觉效果

现在的噪声效果似乎过于强烈了，我们将其稍微减小一点，然后添加一个 Float 类型的参数去调节减小后的噪声强度。

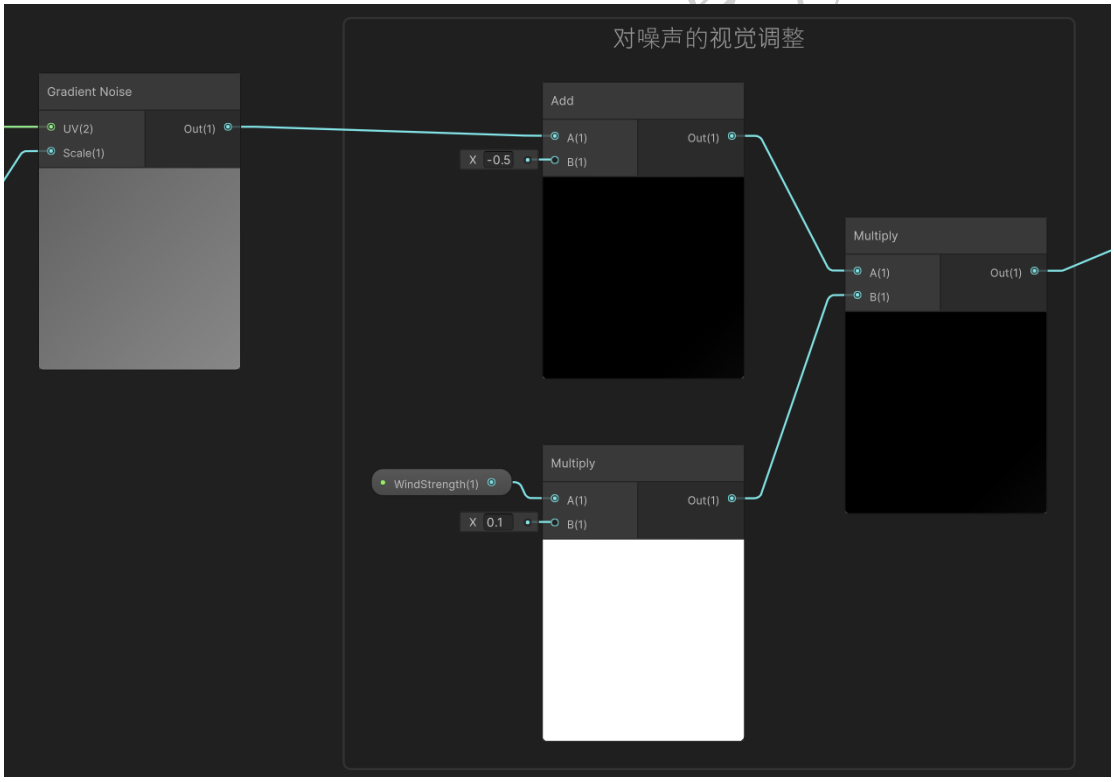


图216.　　调整噪声强度

8. 创建影响蒙版

为了能够设定风对于对象的影响区域，我们需要添加一个蒙版，在顶点噪声偏移进行应用之前，使其和蒙版相乘。

首先创建一个 UV 节点和一个 Float 类型的参数，这个参数将决定蒙版的区域。使用 Split 节点将 UV 节点进行拆分，我们知道拆分后的 R 通道代表 X，G 通道代表 Y。这里使用 G 通道作为蒙版的基础，用 SmoothStep 节点调整蒙版的区域。

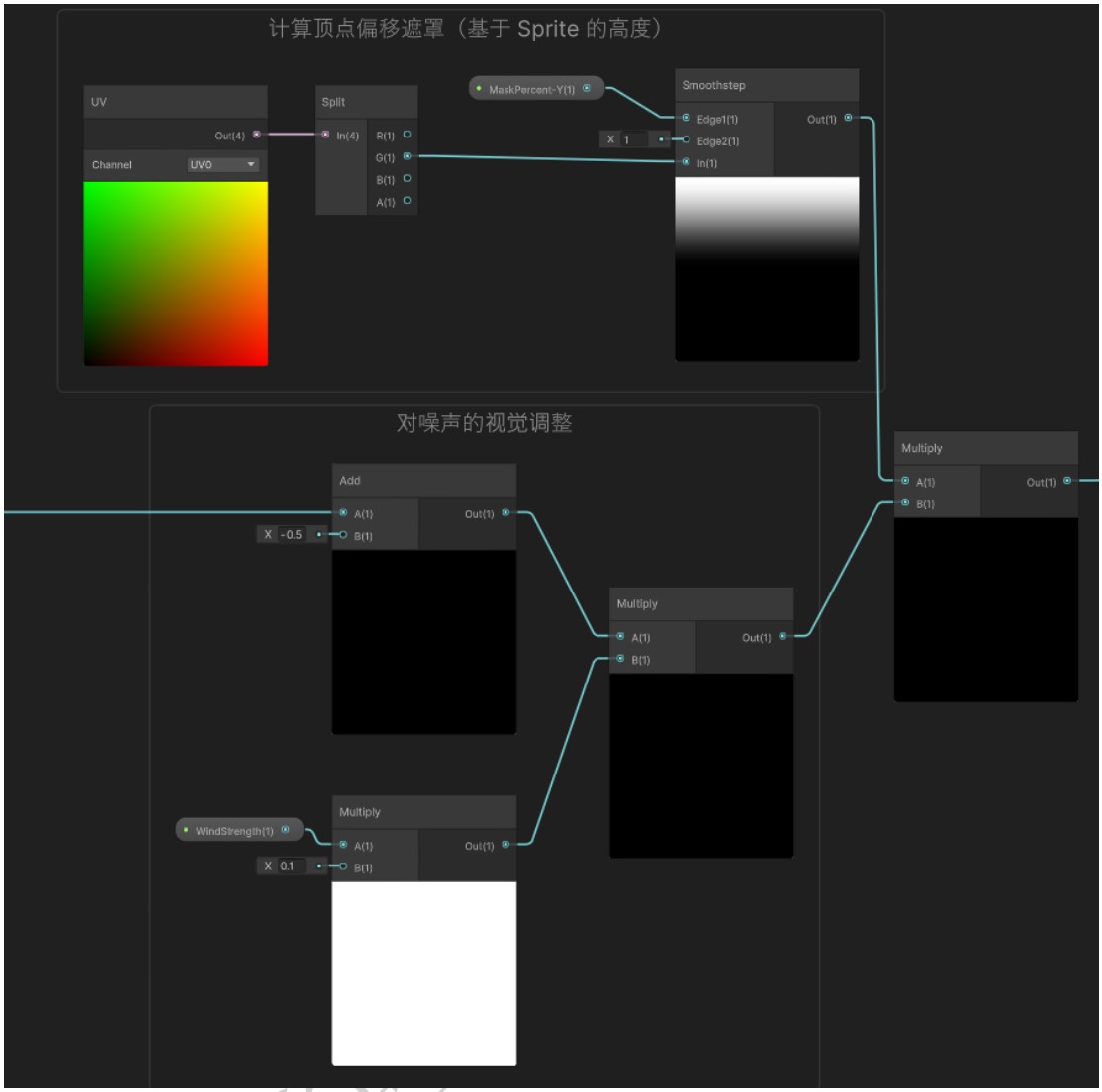


图217.　　顶点偏移蒙版

十五、　2D 简单故障效果

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

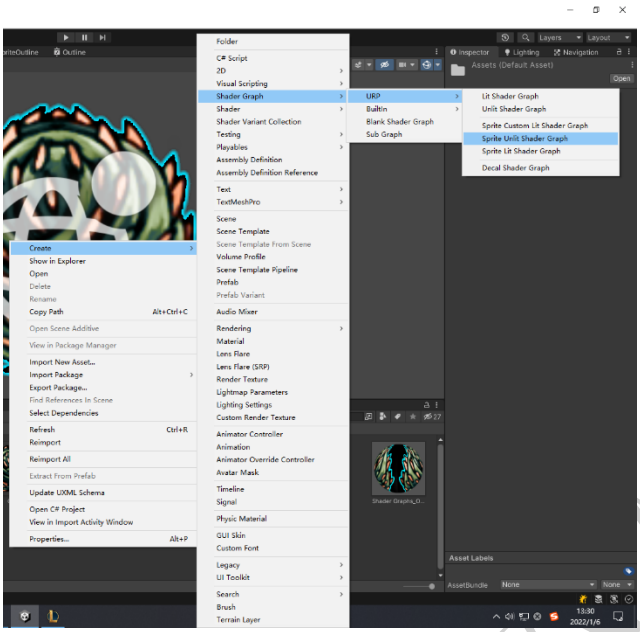


图218.　创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[15],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图219.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

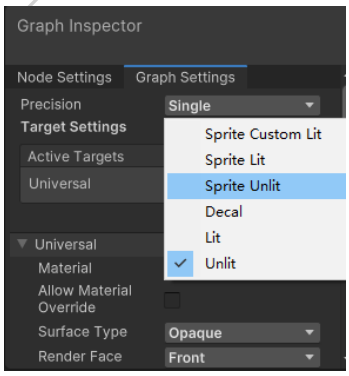


图220.　修改 Material

2. 创建色差效果

在大多数游戏中，故障效果往往伴随着色差效果。我们先创建一个 Float 类型和一个 Texture2D 类型的参数，并将 Texture2D 类型的参数命名为 MainTex。Float 类型的参数用于控制色差效果强度，Texture2D 类型的参数用于显示主纹理。

首先创建三个 Sample Texture2D 节点，将主纹理赋予它们。对于这三个节点，每一个我们都只取一个通道，然后使用 Combine 节点将它们进行组合，并输出到 BaseColor 端

[15] adj. 普遍的；通用的 n. 普遍原则；普遍看法

口。任取一个 Sample Texture2D 节点，将它的 Alpha 通道输出到片元着色器的 Alpha 端口。

图221.　将主纹理进行输出

我们创建 UV 节点、加法节点和减法节点，用之前设置好的 Float 参数，去对 UV 做加法和减法，并将其结果分别给予不同的 Sample Texture2D 节点。

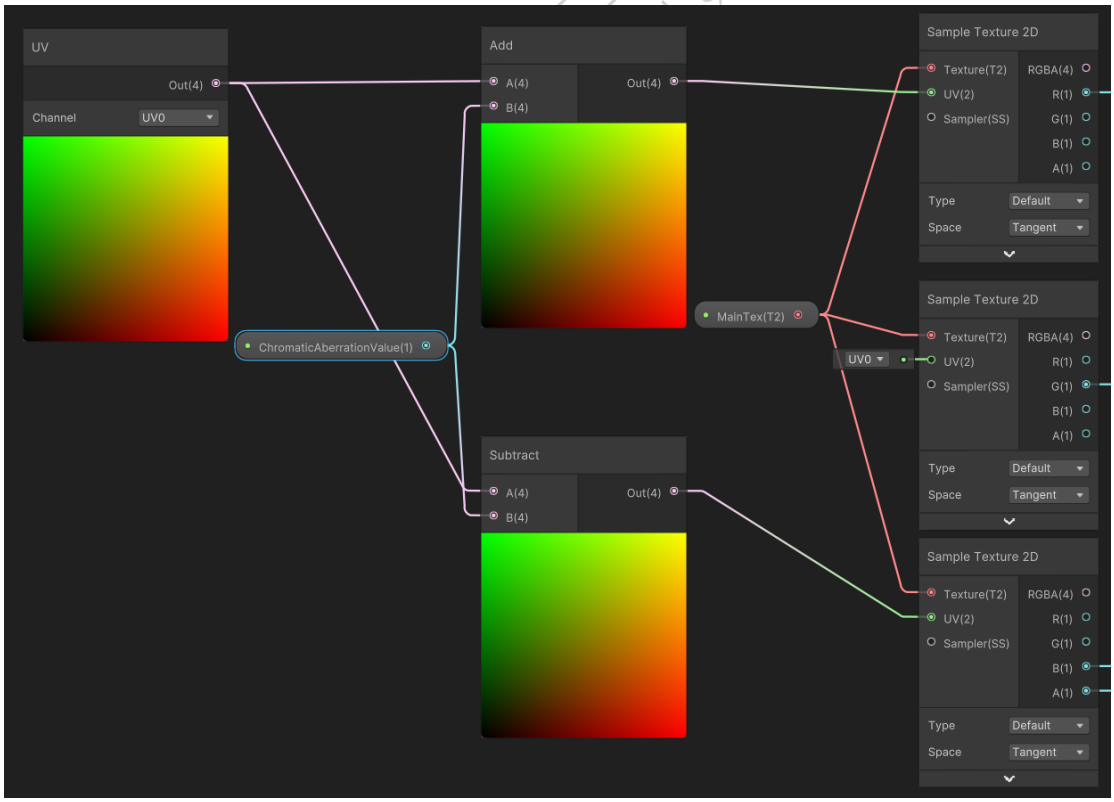


图222.　对 UV 进行加减，实现色差效果

现在我们调节这个 Float 参数，就可以看到色差效果了。为了便于在外部调节，我们还是将这个 Float 参数设置为 Slider，并将其最小值设置为 0.01，最大值设置为 0.1。

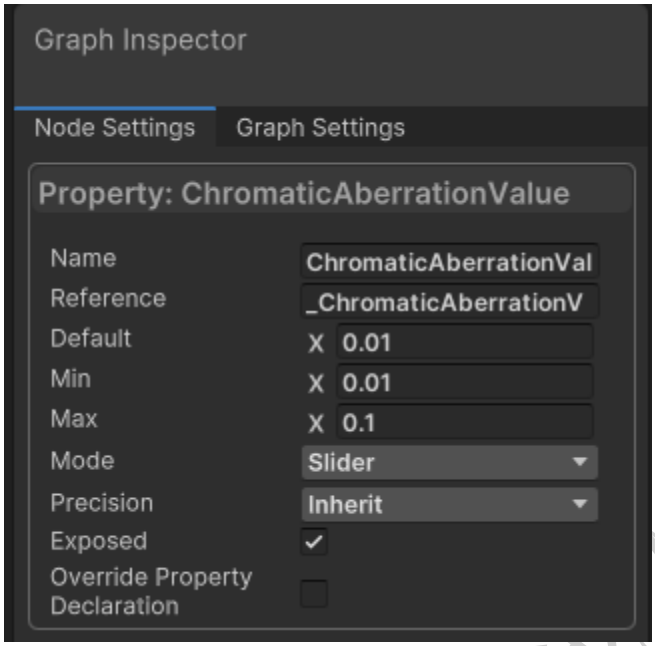


图223.　设置色差参数的模式为 Slider

3. 创建故障纹理

我们创建 Tiling And Offset Node 和一个 Simple Noise Node。我们将 Tiling And Offset Node 连接到 Simple Noise Node 的 UV 端口，然后去调整 Tiling 端口的值，以改变输出的 UV。

可以看到，如果我们将 Tiling 端口的值设置为(1,0)，则 UV 向 Y 方向进行拉伸；如果我们将 Tiling 端口的值设置为(0,1)，则 UV 向 X 方向进行拉伸。

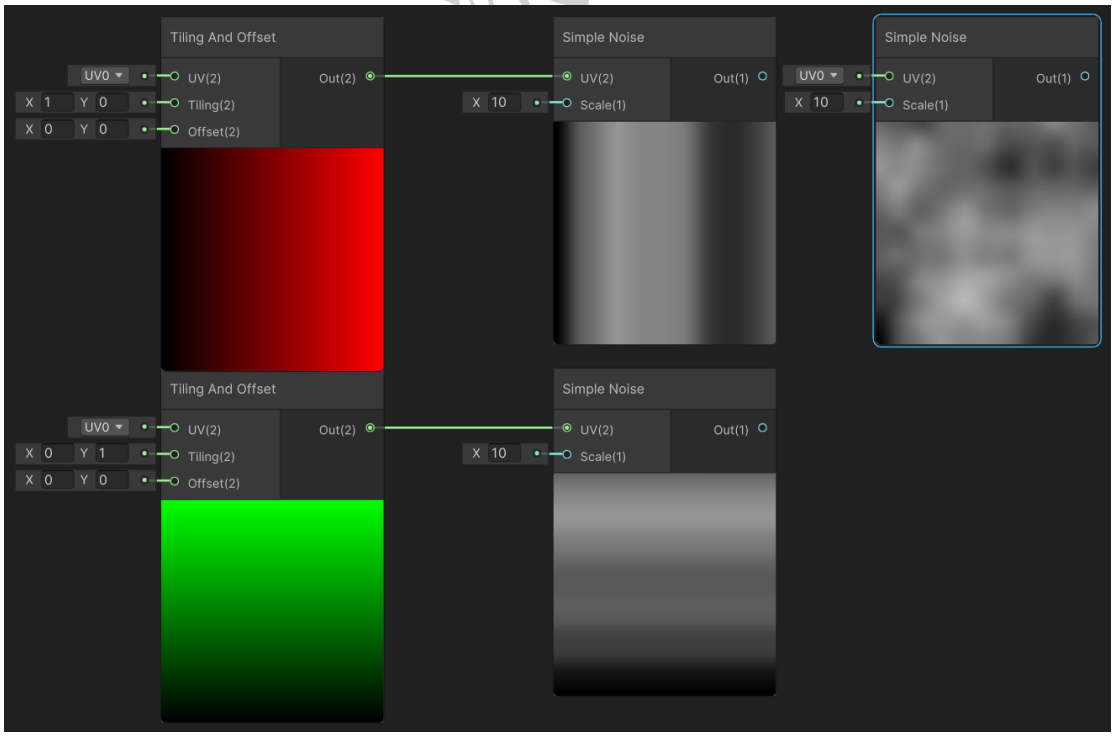


图224.　效果对照

我们创建一个 Boolean 参数、一个 Branch 节点和两个二维向量，使用 Boolean 参数去控制使用何种故障纹理。

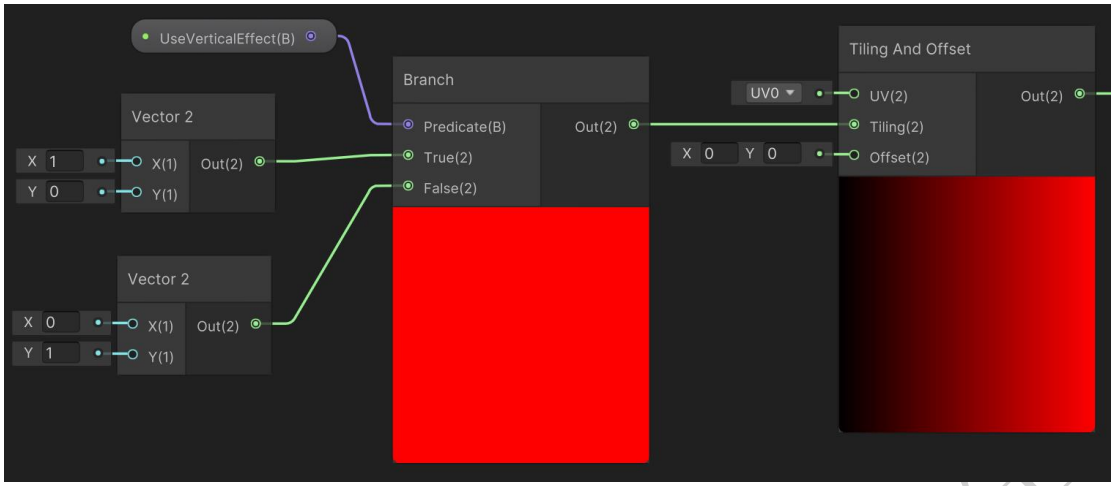


图225.　使用 Boolean 参数控制故障纹理

我们将所得的故障纹理和 UV 相加，用叠加后的 UV 去作为色差效果的基础。

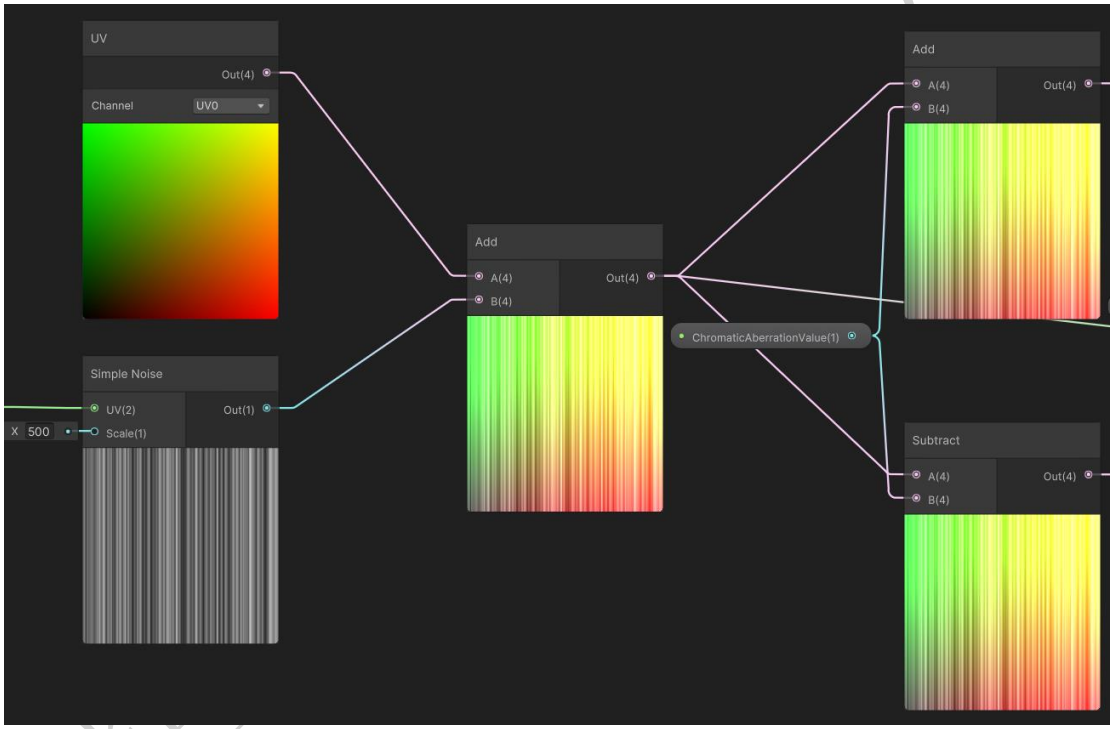


图226.　调整 UV，并将其用于色差

但是，我们发现，现在的故障效果太强了。因此，我们需要使用乘法节点，将故障的效果强度减弱。

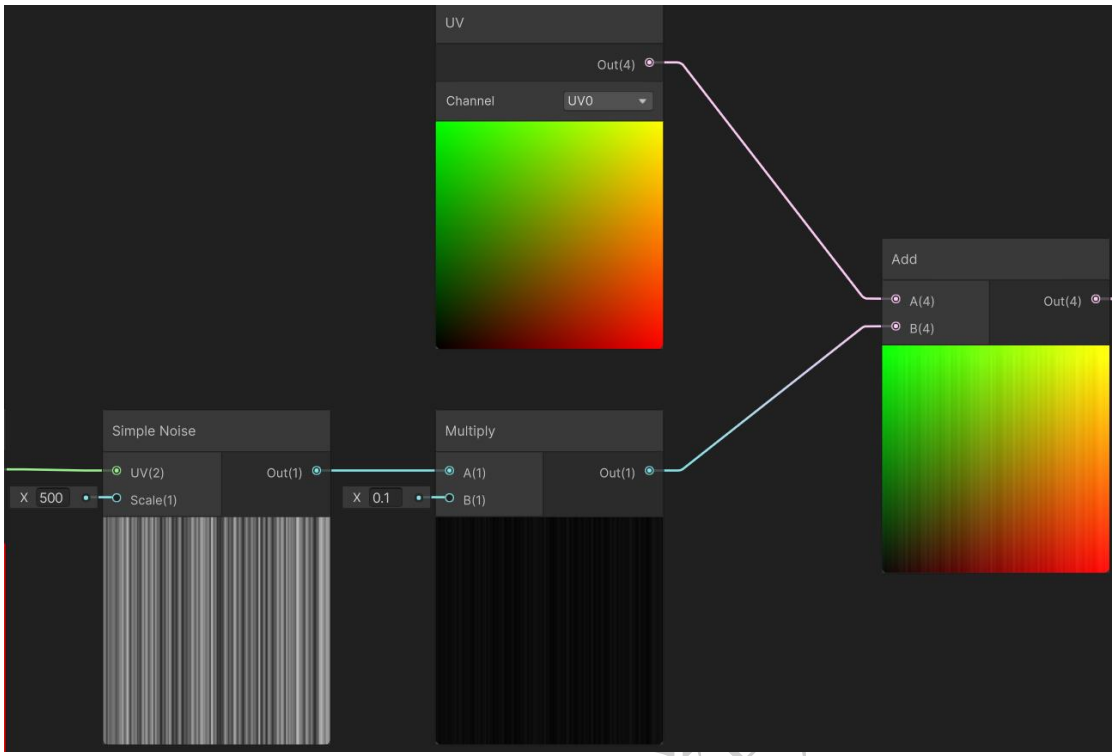


图227.　　减弱故障纹理的强度

4. 调节故障纹理密度

现在我们已经可以预览到故障效果了。但是，在实际运用中，故障纹理的密度并不是一成不变的。因此，我们可以考虑用一个参数，在外部去修改故障纹理的密度。

我们添加一个 Float 类型的参数，并将它的模式设置为整型(Integer)。因为我们是使用 Simple Noise 节点去生成的故障纹理，我们就可以通过调节其 Scale 端口的数值，以设置故障纹理的密度。

同时，我们前面也使用了 Tiling And Offset Node 去拉伸 UV。我们知道，Tiling And Offset Node 的 Tiling 值，代表图像的重复度。我们增加这个端口的数值，也可以控制故障纹理的密度。

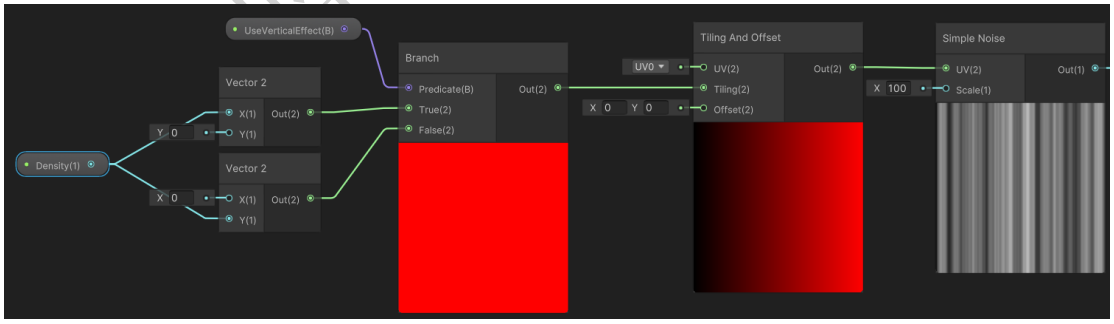


图228.　　调节故障纹理的密度

5. 设置故障偏移方向

我们可以解除故障纹理和 UV 的叠加，来查看一下图像的偏移方向。我们可以发现，现在的偏移方向是向左下。我们希望可以在外部设置偏移的方向，为此，我们可以使用一个枚举参数去控制。

创建两个二维向量，使其分别对应一个枚举参数的端口，然后将故障纹理分别给予这两个二维向量的 X 分量和 Y 分量。当只使用 X 分量时，图像就是向左偏移；当只使用 Y 分量时图像就是向下偏移。现在，当我们选择使用对应的枚举变量时，就可以应用与之对

应的偏移方向了。

6. 闪烁效果

现在的故障效果是直接应用的，但是在实际游戏中，故障效果并不是一直存在的，而是不间断的闪烁。为了实现闪烁效果，我们就需要使用到 Time 节点了。我们创建一个 Time 节点和一个 Simple Noise 节点，我们将 Time 节点的 time 端口给予 Simple Noise 节点的 UV 端口，就会看到有闪烁的效果出现了。如果我们查看它的数值，就会发现其值在 0 到 1 之间不断变化，而我们需要的值只有 0 和 1，不需要之间的过渡值。

因此，我们将考虑使用 Step 节点。我们知道输入值大于或等于 Edge，则返回 1，否则返回 0。将 Simple Noise 节点的输出给予 Step 节点的 Edge 端口，我们就会看到现在的闪烁效果就是只有黑白了。

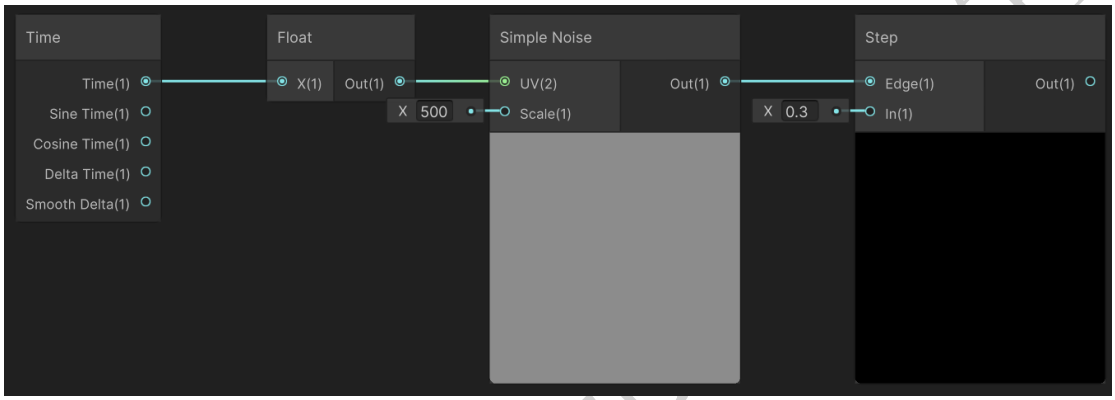


图229. 调整闪烁

因为只有当 Step 节点的输入值大于或等于 Edge 时，才返回 1，即白色。但是，将输入值固定，那么所产生的闪烁效果也就没有那么随机了。

因此我们希望，Step 节点的输入值也能够随着时间发生变化，而 Random Range 节点恰好可以制造一个随机数。

我们创建一个 Random Range 节点，并设置好最小值和最大值，用 Time 节点的 time 端口作为随机种子，然后将输出结果给予 Step 的 In 端口。

7. 完成最终效果

现在我们已经有了闪烁效果、色差效果和故障效果，我们就可以将它们进行一个组合了。我们希望故障效果和色差效果是同时发生，在未发生故障时没有色差效果出现，也就是让闪烁效果同时影响故障和色差效果。我们将色差效果的强度参数和闪烁效果相乘，然后在故障纹理和原始 UV 相加之前，另其和闪烁效果相乘。

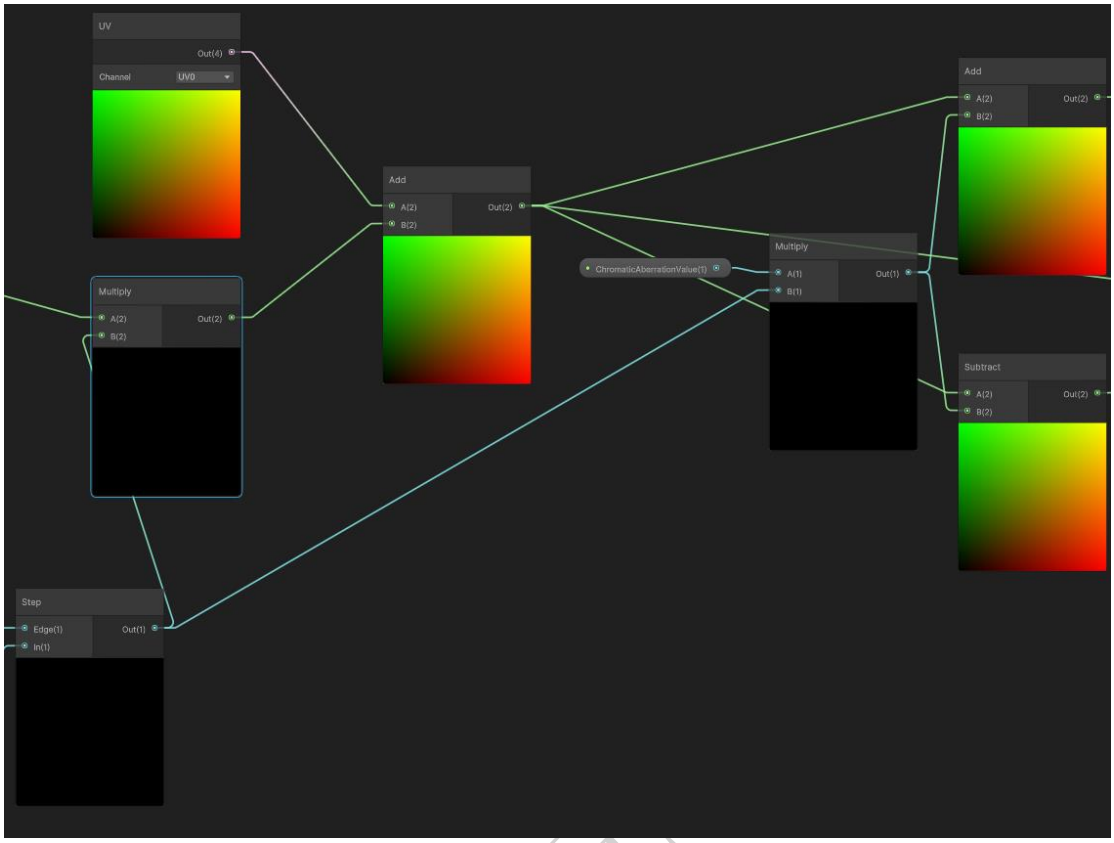


图230.　闪烁效果控制色差和故障的发生

十六、　UV 中心缩放

1. 思路讲解

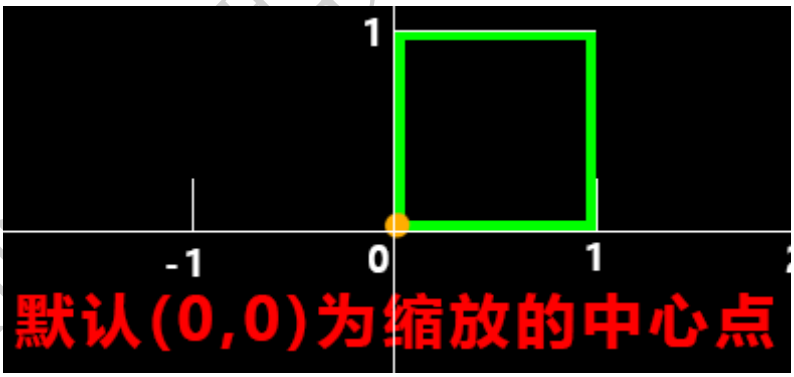


图231.　默认 UV 缩放点

通常情况下我们缩放 UV，其缩放的中心点是 UV 的左下角（即(0,0)点）,但在很多情况下，我们是需要从中心点进行缩放的。

为了实现这一效果，我们需要移动缩放的中心点，在后面我就直接简称为缩放点。在 Tilling 为 1 时，我们的缩放点位置就应该是(0.5,0.5)；当 Tilling 为 2 时，我们的缩放点位置就应该是(1,1)。由数学归纳法可以得知，要实现中心缩放，我们需要将缩放点的 X 和 Y 都移动 Tilling 值的一半。

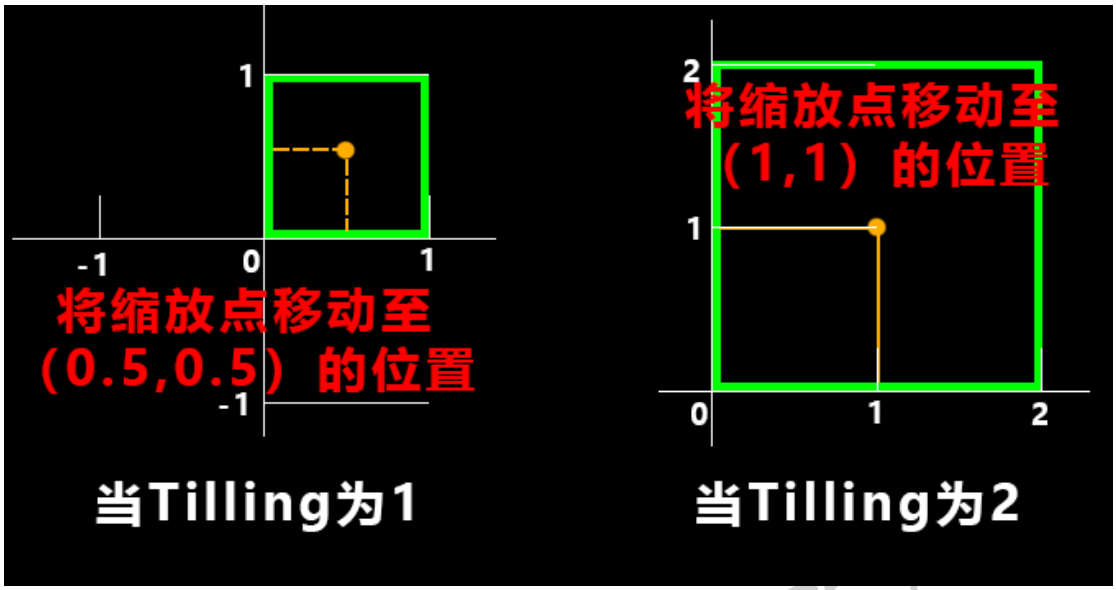


图232.　当 Tilling 为 1 和 2 时的缩放点坐标

在这里，我们一定要先进行缩放（也就是先进行乘法），然后才进行移动（缩放完成后再进行加法）。这是因为，一旦我们先进行加法运算，再进行乘法运算，相同值的情况下所得结果一定与前者不同。

2. 创建 ShaderGraph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

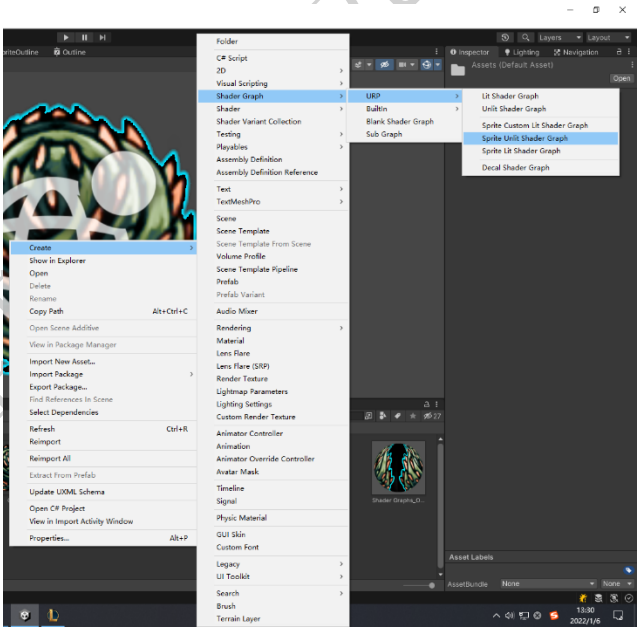


图233.　创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[16],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

[16] adj. 普遍的；通用的 n. 普遍原则；普遍看法

图234.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

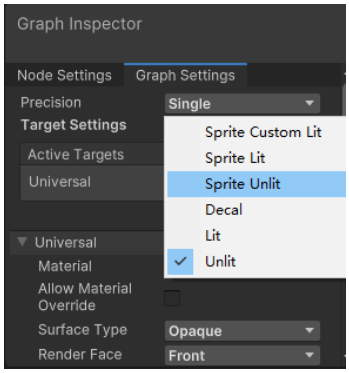


图235.　修改 Material

3.　虚假的中心缩放

我们在 Shader Graph 的参数面板中，添加一个 Texture2D 类型的参数，并将其命名为 MainTex。接着我们创建一个 UV 节点和 Sample Texture 2D 节点，将纹理和 UV 赋予 Sample Texture 2D 节点，并将其输出到片元着色器的 BaseColor 端口和 Alpha 端口。

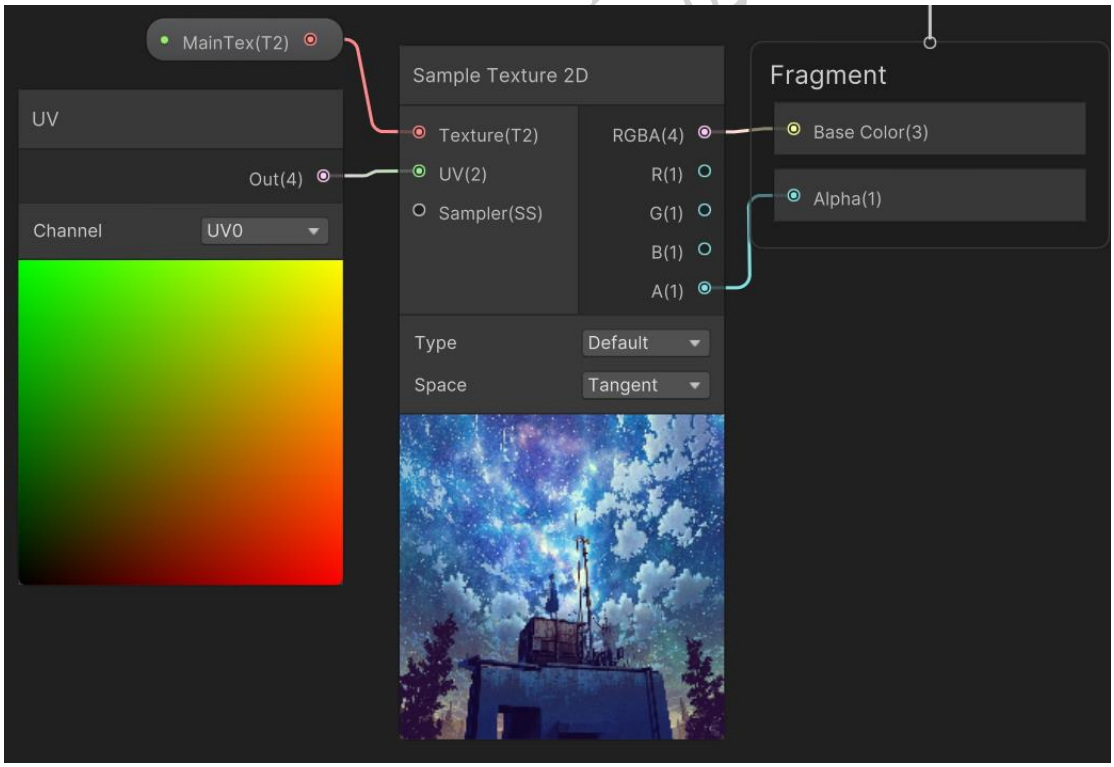


图236.　输入标准 UV 所得的图像

如果我们此时使用乘法节点将 UV 进行放大，就会看到，图像是以左下角为中心点进行缩放的，这是 UV 的默认缩放位置。后面所需要进行的操作，就是去让缩放点位置发生变化。

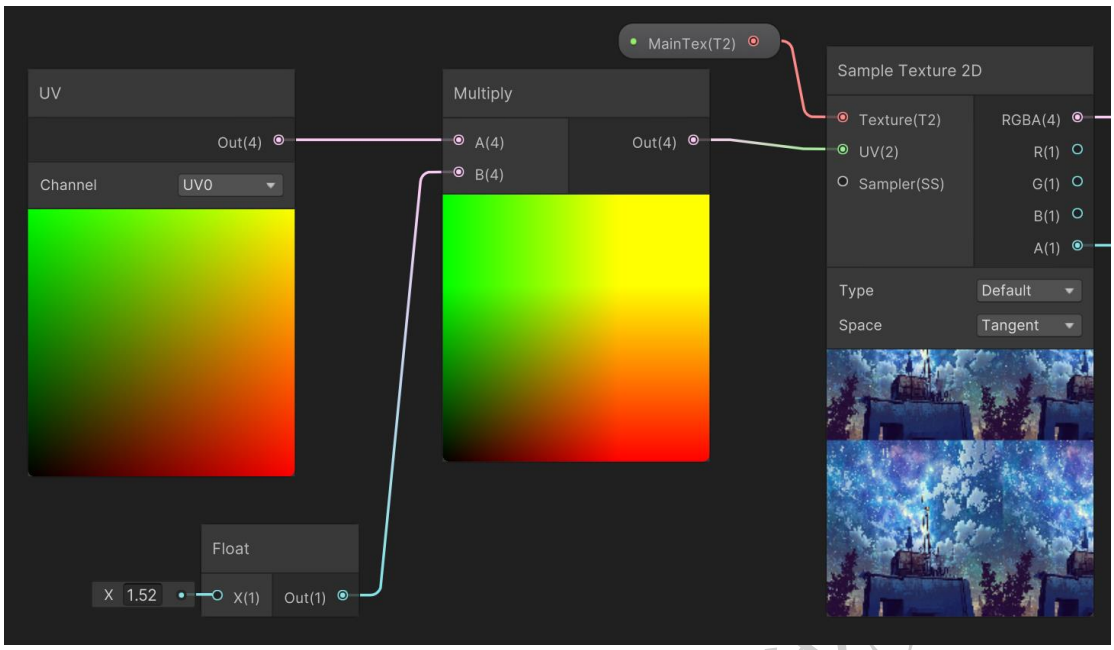


图237.　　查看 UV 的默认缩放中心点

首先将 Tiling 暴露到参数面板，方便在外部对其进行修改。

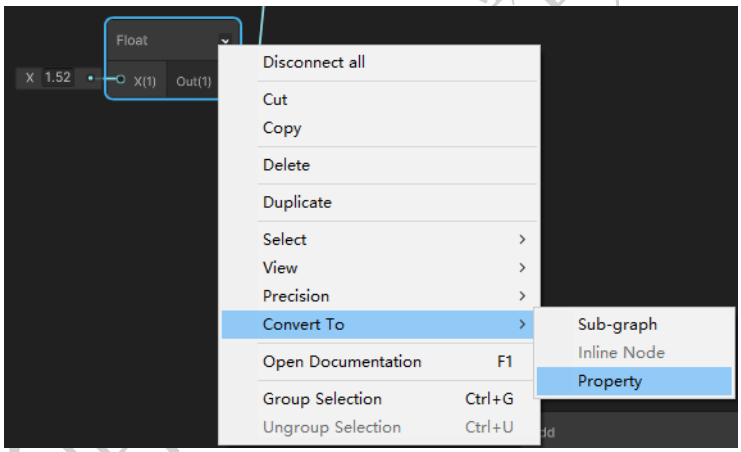


图238.　　将节点转换为参数暴露到参数面板

接着由前面的数学归纳可知，UV 的中心位置坐标与 Tiling 的值紧密相关，我们需要在 UV 缩放完成后，使 UV 整体的 X 和 Y 均移动 Tiling 值的一半。

我们知道，UV 的 X 加上正数就是向左偏移，Y 加上正数就是向下偏移，因此我们需要加上的是一个负数。也就是说在得到 Tilling 值的一半后，需要使用 Negate 节点对其取反。

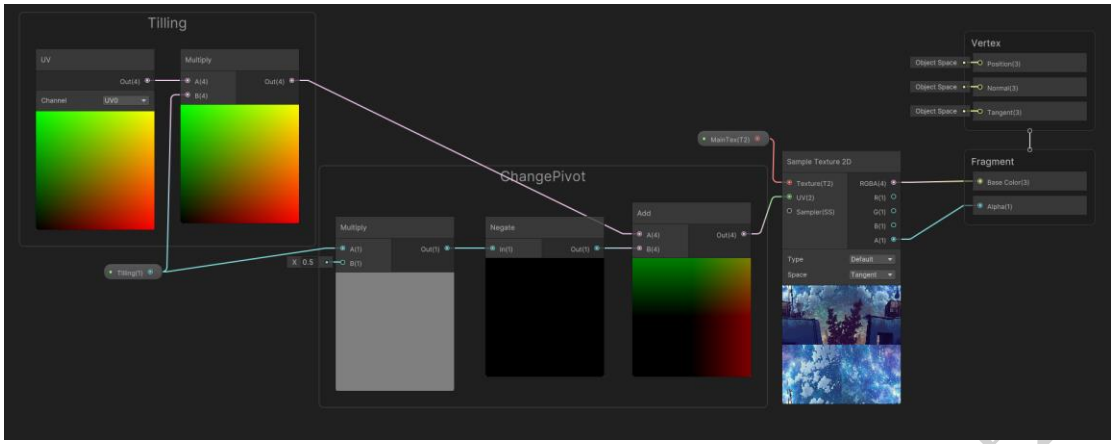


图239.　　虚假的中心缩放最终连接图

4. 真实的中心缩放

从上面的最终输出结果可以看出，虽然我们现在是可以从中心进行缩放了，但是我们的整体图像也发生了移动。当未发生任何缩放时，也即 Tiling 为 1 时，原图形被分为了四块，这是我们不希望看到的。

我们希望得到的结果是，原始图像始终位于中心的位置，缩放时新的图像从原始图像周围出现。为了实现真实的中心缩放，我们需要了解怎么让原始图像的位置保持在中心。

缩放倍数	原始中心点	缩放后的中心点	差值
放大 1 倍	(0.5,0.5)	(0.5,0.5)	(0,0)
放大 2 倍	(0.5,0.5)	(1,1)	(0.5,0.5)
放大 3 倍	(0.5,0.5)	(1.5,1.5)	(1,1)
放大 4 倍	(0.5,0.5)	(2,2)	(1.5,1.5)
放大 5 倍	(0.5,0.5)	(2.5,2.5)	(2,2)
放大 6 倍	(0.5,0.5)	(3,3)	(2.5,2.5)

我们观察上面的表格，会发现原始的中心点坐标与缩放倍数无关，与缩放倍率有关的是缩放后的那个中心点坐标，其始终是缩放倍数的一半。由数学归纳法可以推导出，原始中心点与缩放后中心点的公式，即原始坐标+（缩放后-原始）。

上面的坐标差值，我们通过图表可以得知，其与缩放倍数有关，其差值就是（当前缩放倍数-1）/2。在 ShaderGraph 中，我们可以只使用一个 Float 参数作为差值，即一维向量，当其连接到二维向量时，shaderGraph 会自动帮我们转换为二维向量。

哔哩哔哩次元游戏人生

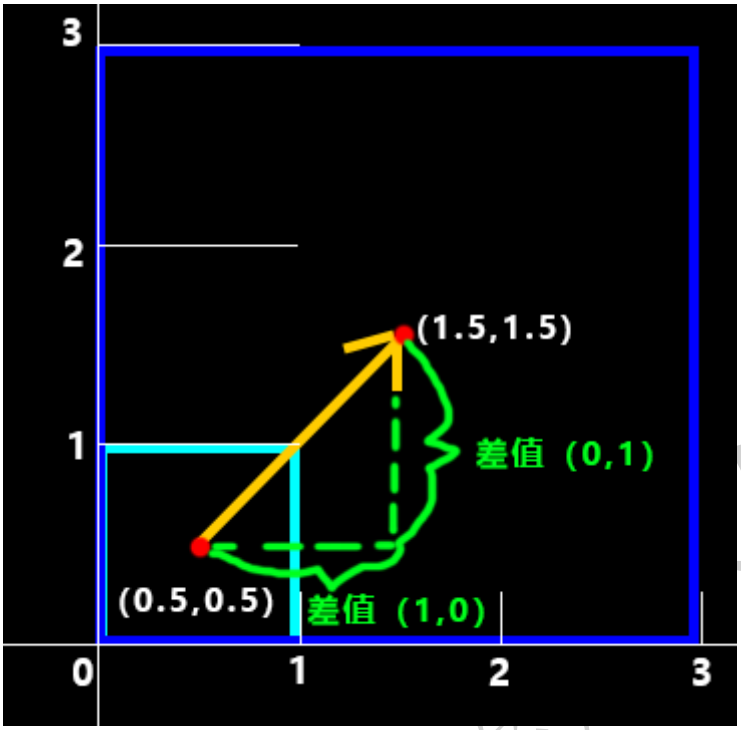


图240.　原始中心坐标与新中心坐标的差值

其原理其实也是在整体缩放完成后，再对整体的 UV 进行移动。只不过，我们的 X 与 Y 移动的距离是新旧中心点的差值，而不是左下角的默认缩放点，与新中心点坐标之间的差值。

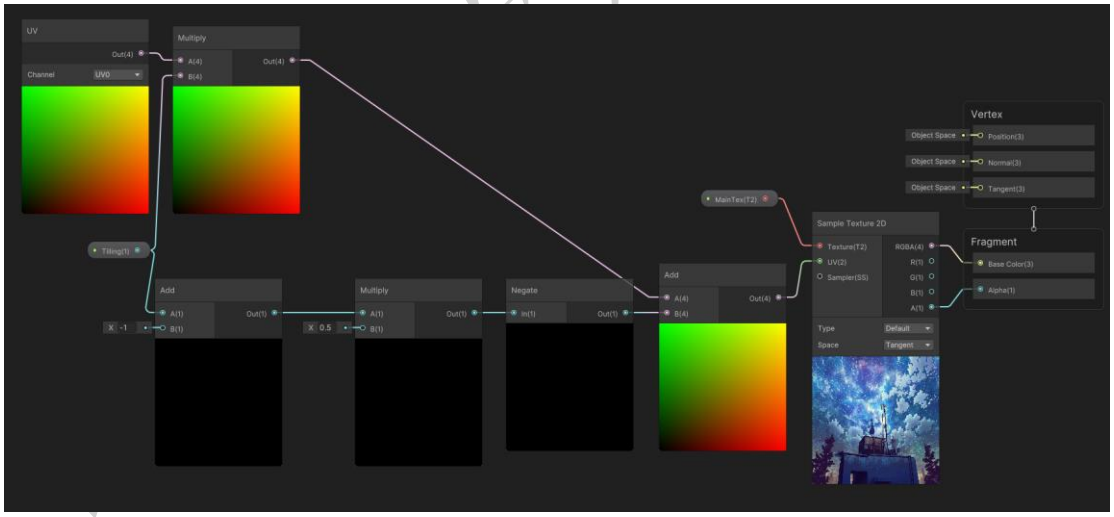


图241.　真实的中心缩放

十七、　2D 月食

1. 思路

在现实生活中我们经常会看到月食的现象，众所周知，月亮的亮度是来自于太阳的。在太阳光线到达月亮之前，如果被地球挡住了，那这部分光线就不会到达月球，以至于出现月食。

我们知道月食是怎样产生的后，就明白可以用一张遮罩去模拟这种效果。我们可以使用一张遮罩图，然后缩放或者偏移这张遮罩图，就可以实现月食。但，这不是我们这里想使用的，我们需要的是通过数学方法生成的一张简单圆形遮罩图。

使用数学方法生成的优势在于，可以更加便捷高效的调整遮罩图像（可以多个数学方法联合使用），且不会因为放大缩小而丢失精度，也不会出现一些奇奇怪怪的问题。且，我们使用采样贴图是位图，而使用数学方法生成的，则是一张矢量图形。

2. 创建 ShaderGraph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

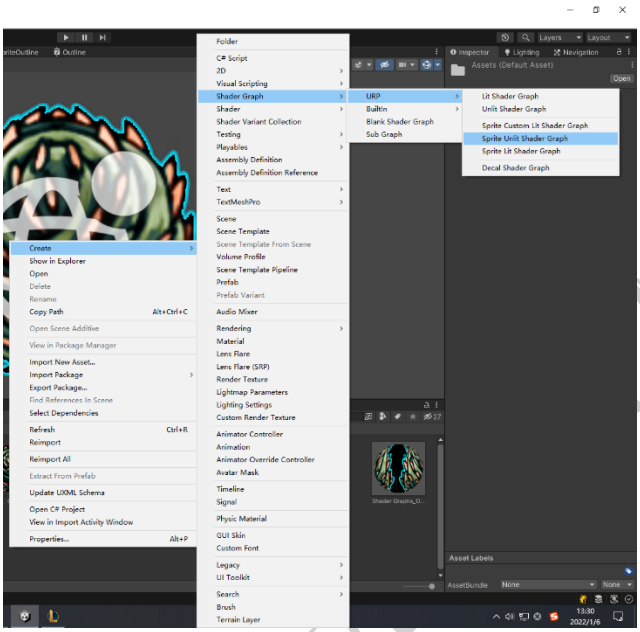


图242.　创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[17],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图243.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

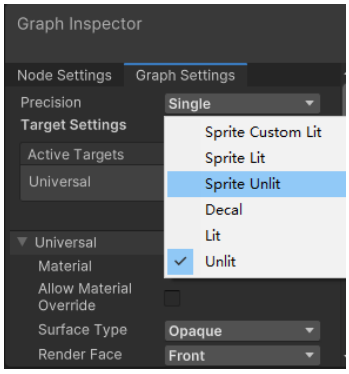


图244.　修改 Material

[17] adj. 普遍的；通用的 n. 普遍原则；普遍看法

3. 使用圆的 SDF 去生成有向距离场纹理

首先，我们需要在参数面板中添加一个 Texture2D 和 Float 类型的参数，Texture2D 类型的参数将作为主纹理，我们将其命名为 MainTex。Float 类型的参数将控制圆形遮罩的圆点位置，以及用于后续对遮罩区域的调整。

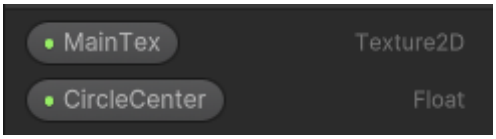


图245.　月食的参数面板主纹理和圆心

接着我们需要创建一个 Vector2 节点、加法、减法节点和 UV 节点。我们使用加法节点，让圆心位置和二维向量相加，我们之所以使用一个 Float 类型去控制圆心位置，是因为我们只希望生成的遮罩沿主对角线移动，且 ShaderGraph 在计算时会自动帮我们将其转化为二维向量。

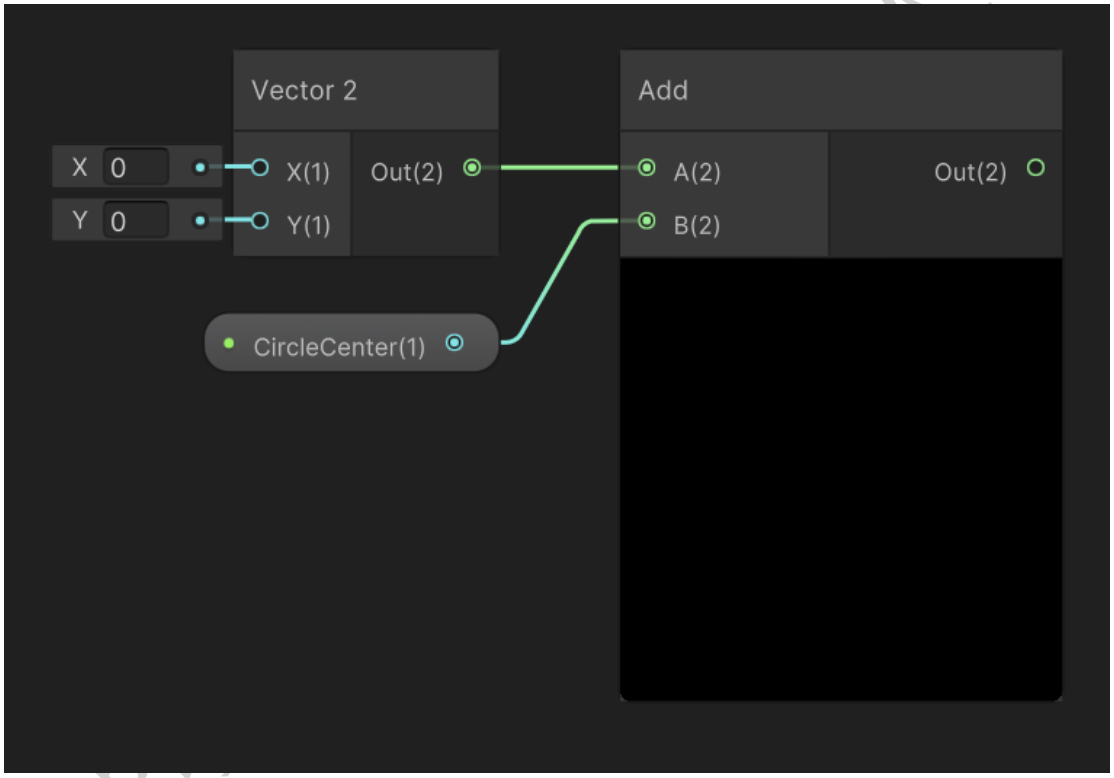


图246.　控制圆点的位置

接着我们需要计算 UV 上面每一个坐标到达圆心的距离，这会使用到向量的减法，在进行减法运算时需要注意被减数是谁。这里的被减数是 UV 节点，而不是我们的圆心坐标。

计算完成每一个 UV 坐标到圆心的向量后，我们需要使用 Length 节点去计算每一个向量的模。当计算完后，我们就可以看到 ShaderGraph 中有了一个距离场，注意，这并不是有向距离场，因为此时我们还没有减去圆的半径。

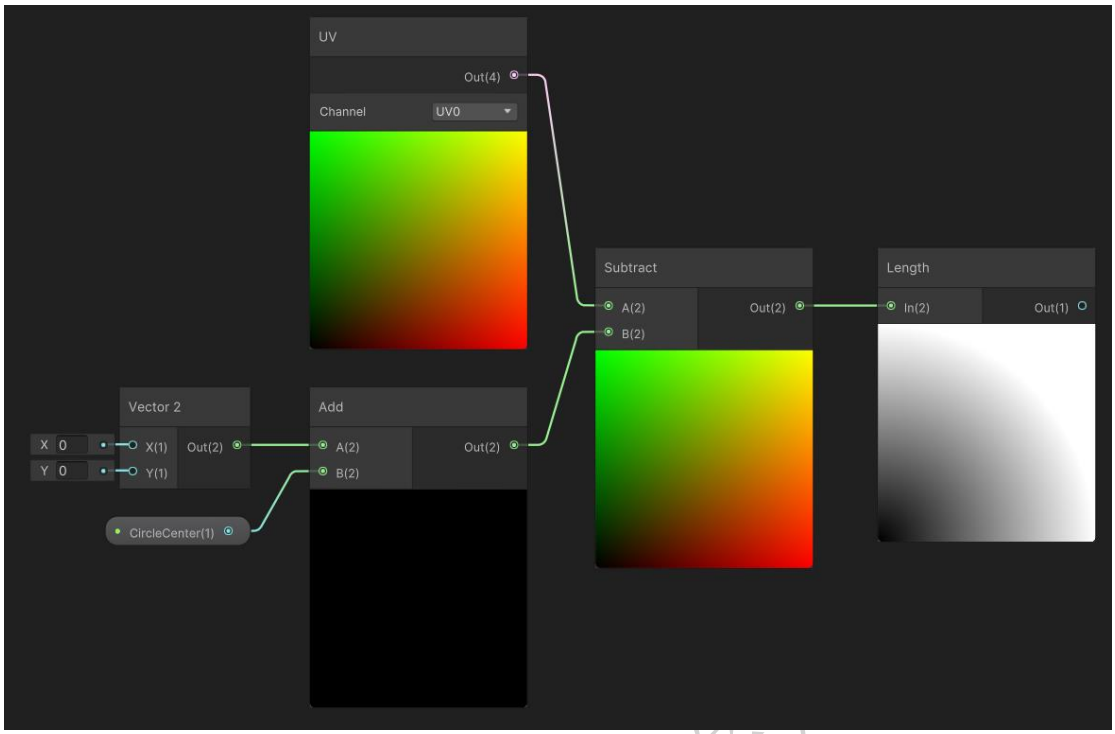


图247.　计算每一个 UV 坐标到圆心的距离，并生成距离场

现在我们已经有了距离场，我们只需要再减去一个半径就是有向距离场了。但是，这个半径究竟应该是多少呢？这里考虑到后续效果的实现，将使用一个 Float 类型的参数和乘法节点去控制半径，使其不会大于圆心坐标的数值。

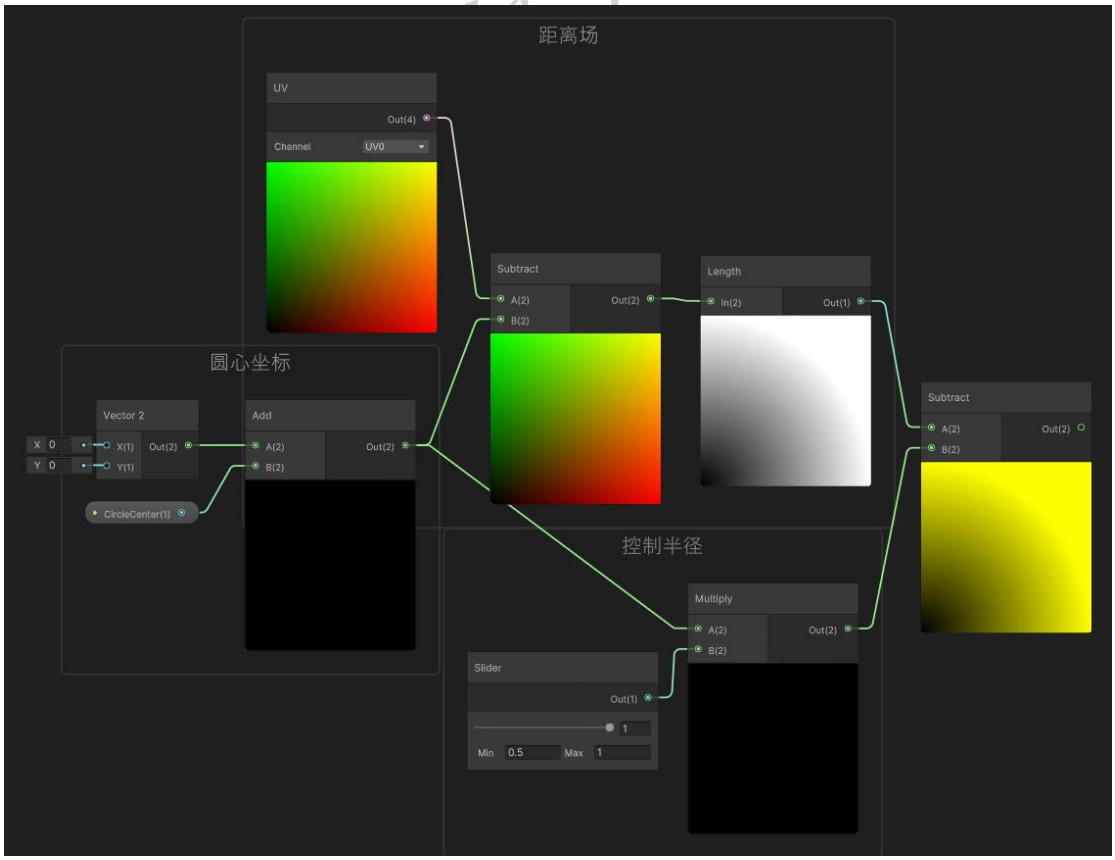


图248.　控制圆的半径并生成有向距离场

4. 生成蒙版

通过 ShaderGraph 我们可以知道，生成的有向距离场是一个柔性的渐变，而我们的蒙版一个是只有纯黑和纯白这两种情况。这时我们就可以考虑使用 Step 节点了，一旦 Step 节点的输入值 In 大于或等于输入值 Edge，则返回 1（白色），否则返回 0（黑色）。利用这一特性，我们便可以根据有向距离场去生成一张蒙版了。

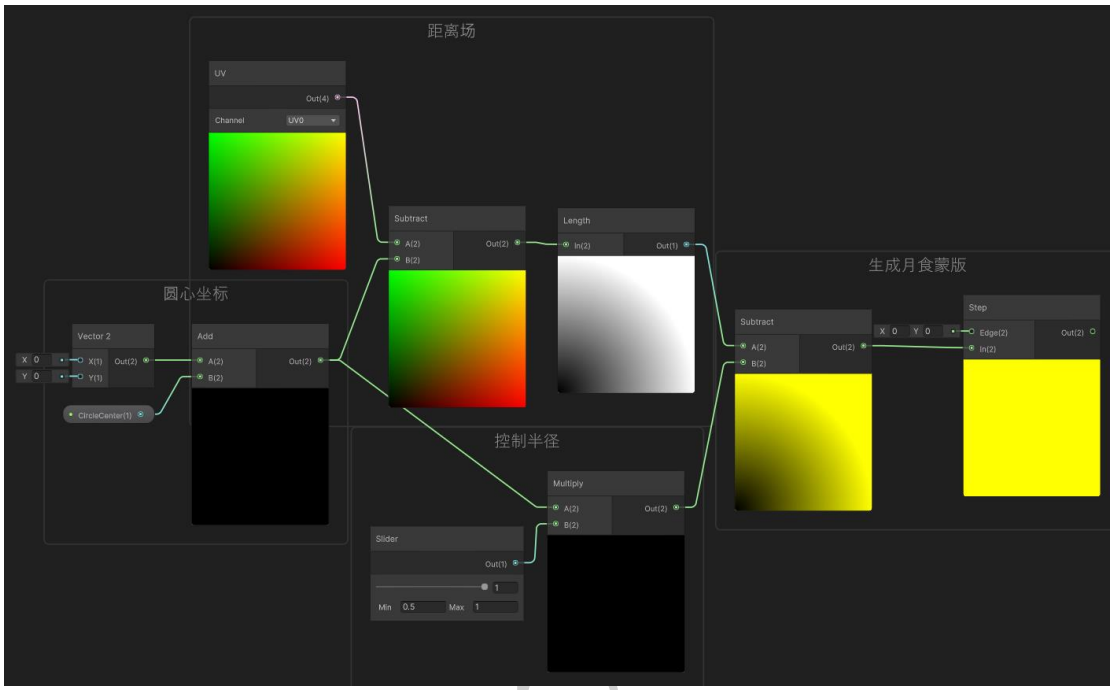


图249.　　使用 Step 节点生成月食蒙版

5. 实现月食效果

现在我们已经拥有了月食的蒙版了，就可以创建一个 Sample Texture 2D 节点和乘法节点，并将主纹理赋予 Sample Texture2D 节点。然后，我们将原图形的 Alpha 通道和月食蒙版相乘，得到一个被月食效果影响的新蒙版。完成操作后，将其输出到对应的片元着色器的端口。

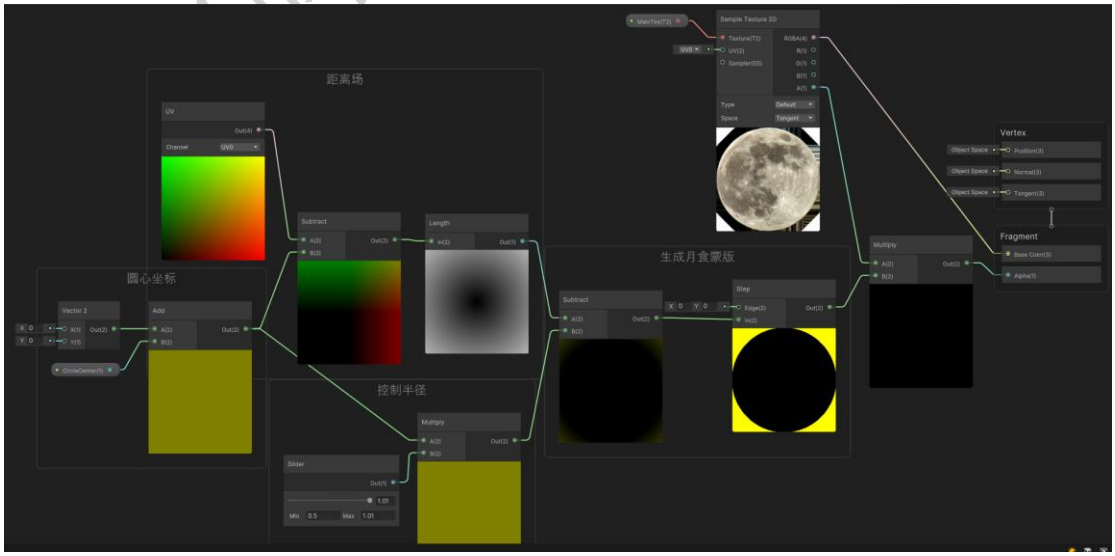


图250.　　最终的月食效果

6. 可能出现的问题

在这个效果中，我使用的是一张带有圆形 Alpha 通道的图像，如果使用的图像本身不具备圆形的 Alpha 通道，我们则需要再使用圆的 SDF 去生成一张蒙版。

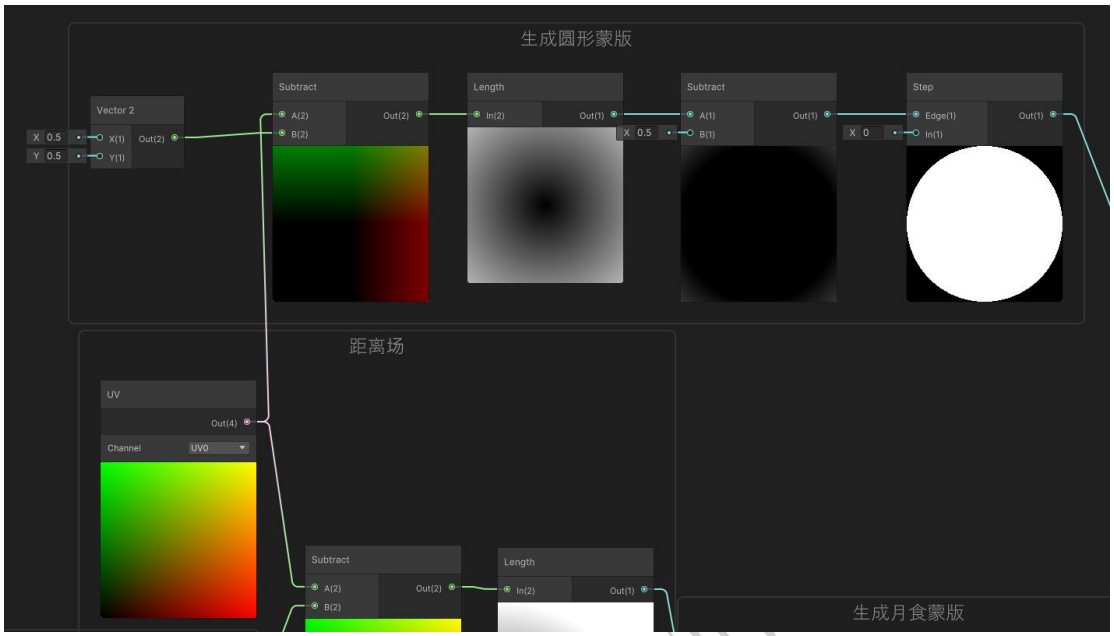


图251. 生成一张圆形蒙版，用于实现月食效果

十八、 2D 元球-Metaball

十九、 2DSDF 圆

符号距离函数（sign distance function），简称 SDF，又可以称为有向距离函数，在空间中的一个有限区域上确定一个点到区域边界的距离并同时对距离的符号进行定义：点在区域边界外部为正，内部为负，位于边界上时为 0。

简单来说，SDF 就是某一点到某一个区域边界的最短距离。

求某一点到圆的边的距离是最为简单的，这里分两种情况。大致来说都是一样的，就是用点与圆心的距离减去半径来进行判别。

1. 圆心在原点

圆心在原点时，我们只需要知道两个参数，任意点 P 的坐标和圆的半径 r，就可以求出点 P 距离圆边缘的距离。

哔哩哔哩欢元源

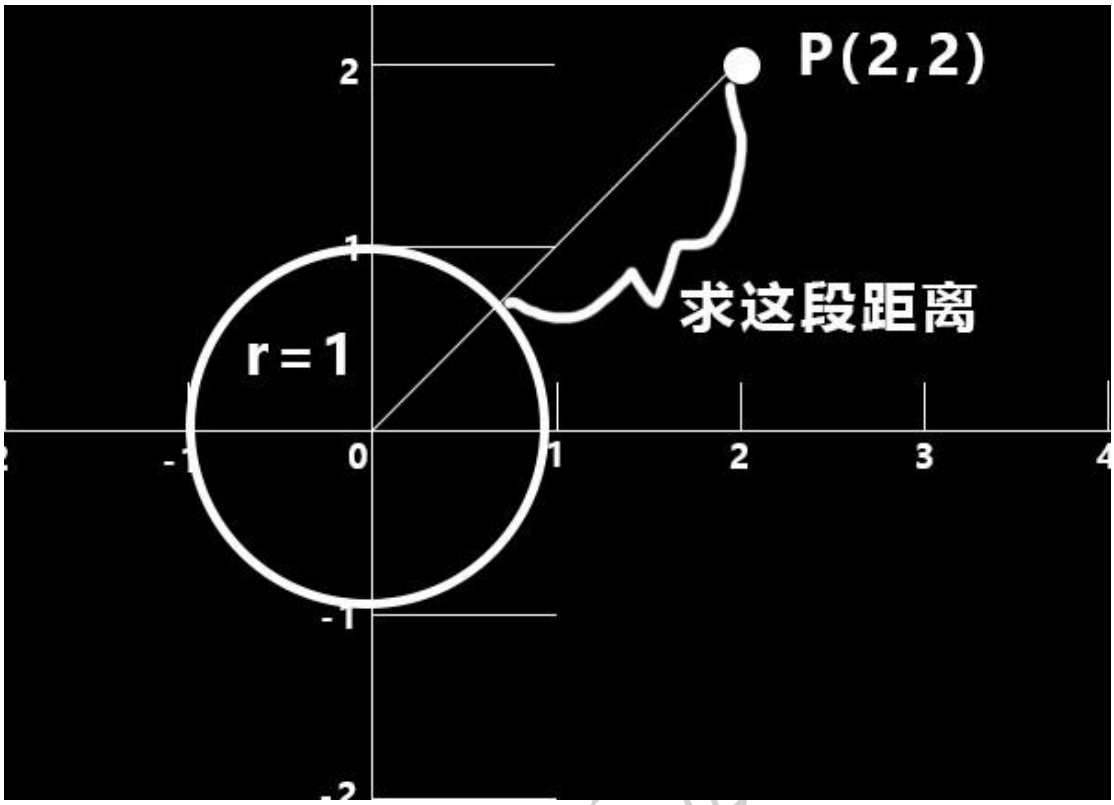


图252.　　圆心在坐标轴原点

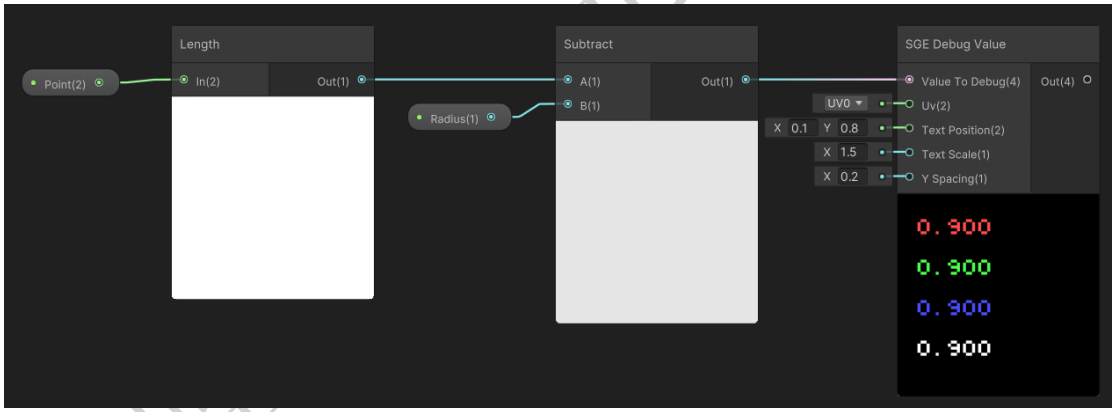


图253.　　圆心在原点时的 ShaderGraph

2. 圆心不在原点上

圆心不在原点上就需要三个参数，圆心坐标、点 P 坐标、圆半径。首先计算圆心与点 P 的距离，然后再用半径去减。（因为求的是圆心与点的距离，这与向量的方向无关，因此不需要关心被减数是谁。）

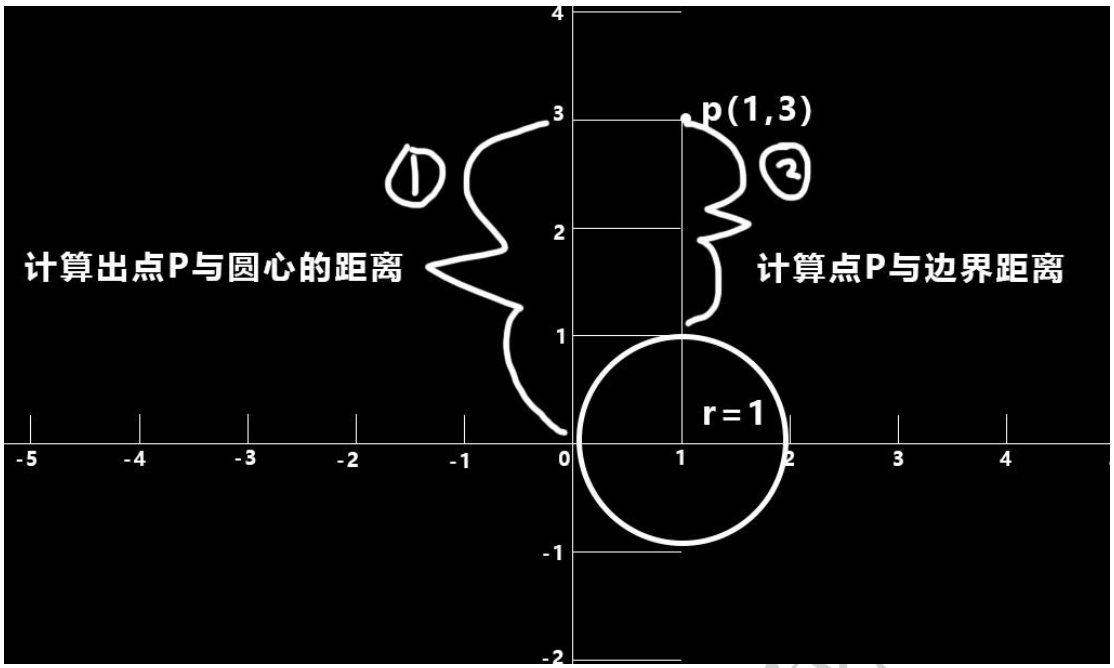


图254.　　圆心不在坐标轴原点

图255.　　圆心不在原点时的两种连接方法

二十、　2DSDF 线段

线段的推论很重要，后续很多类型都以此为基础进行。

线段的 SDF 输入参数有三个：任意点 p 的坐标，线段两端点 a，b 坐标。

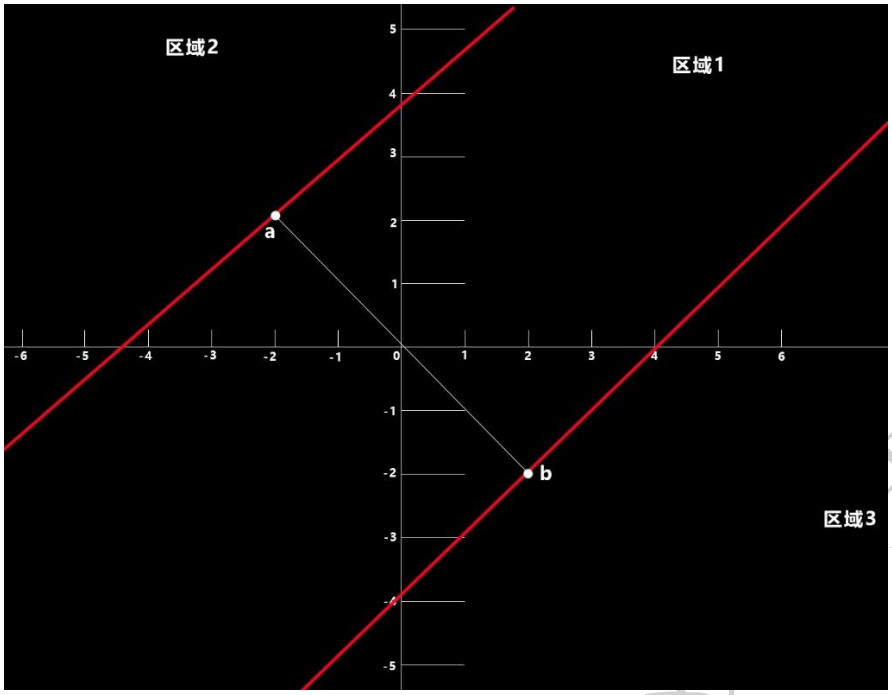


图256.　　SDF 线段区域划分

如上图所示，一条线段可以被两条直线分为三个部分（两条红线都垂直于这条线段）。计算任意点 P 与线段的最短距离，会根据点 P 不同而有所变化。

1. 位于区域一

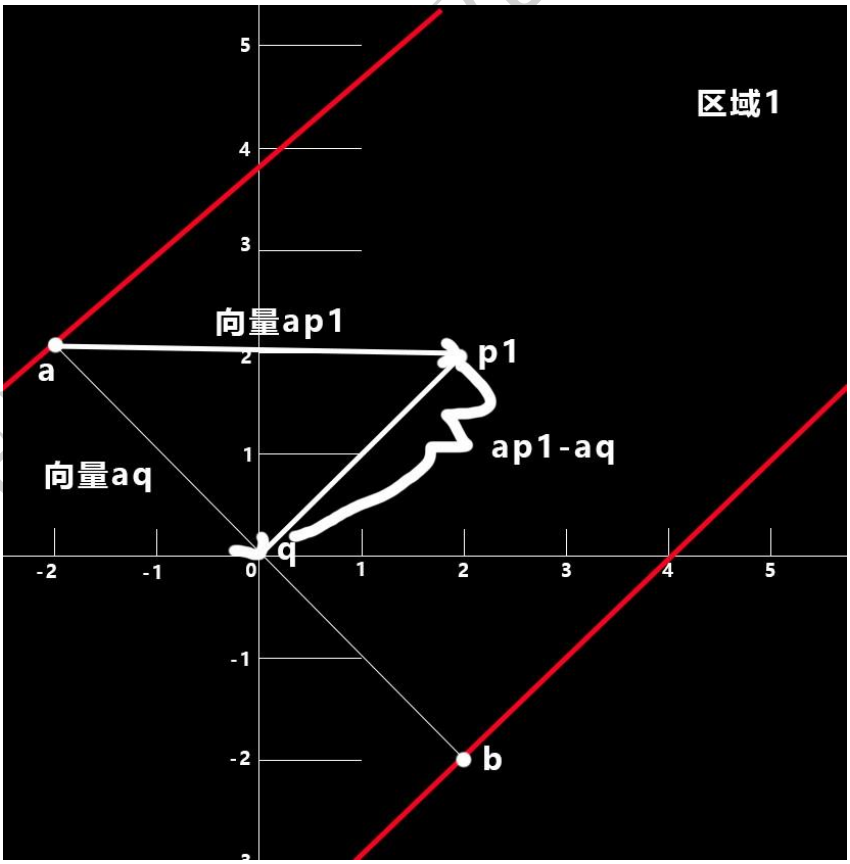


图257.　　点位于区域一内

当点位于区域一以内，点到线段的最短距离，就是过点做一条垂直于线段的直线。如

上图，点 p_1 到线段的最短距离就是向量$\overrightarrow{qp_1}$的大小，即 d=$\|\overrightarrow{qp_1}\|$。

由向量的运算法则可以知道，$\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[18]$\overrightarrow{ab}$。（r[19]表示向量$\overrightarrow{aq}$的大小与向量$\overrightarrow{ab}$大小的比值）

如果我们知道点 q 的坐标，可以由三角函数$\cos\theta = \frac{|\overrightarrow{aq}|}{|\overrightarrow{ap_1}|}$（余弦=临边/斜边），计算出这两个向量的夹角，进而得到向量$\overrightarrow{qp_1}$的大小。

但是现在我们并不知道点 q 的坐标，因此我们无法计算出这个夹角，我们需要通过其他方法去进行计算以获得向量$\overrightarrow{qp_1}$的大小。

我们根据向量数量积的公式$\overrightarrow{ab} \cdot \overrightarrow{ap_1} = |\overrightarrow{ab}||\overrightarrow{ap_1}|\cos\theta$以及三角函数$\cos\theta = \frac{|\overrightarrow{aq}|}{|\overrightarrow{ap_1}|}$。可以将数量积公式中的余弦剔除，得到$\overrightarrow{ab} \cdot \overrightarrow{ap_1} = |\overrightarrow{ab}||\overrightarrow{aq}|$。接着我们在等式两边同除$|\overrightarrow{ab}|$的平方，可以得到最终的等式$\frac{\overrightarrow{ab}\cdot\overrightarrow{ap_1}}{|\overrightarrow{ab}||\overrightarrow{ab}|} = \frac{\overrightarrow{ab}\cdot\overrightarrow{ap_1}}{\overrightarrow{ab}\cdot\overrightarrow{ab}} = \frac{|\overrightarrow{aq}|}{|\overrightarrow{ab}|}$=r。

因为$\overrightarrow{aq}$向量与$\overrightarrow{ab}$向量是一个共线向量，那么$\overrightarrow{aq}$向量就可以用$\overrightarrow{ab}$向量表示出来。

2. 位于区域二和三

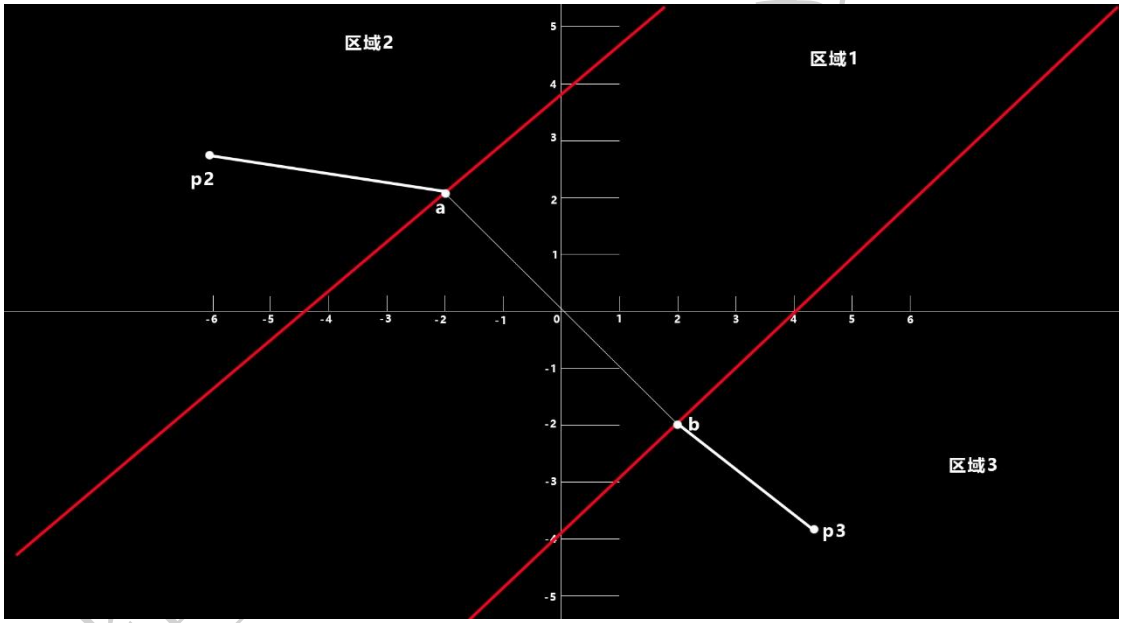


图258. 点位于区域二和三

当点位于区域二和三时，其最短距离就是到线段端点的距离。

区域二：d=$\|\overrightarrow{ap_2}\|$

区域三：d=$\|\overrightarrow{bp_3}\|$

此时，我们可以发现，区域二三的公式和区域一的其实非常类似，只不过区域二三的公式中并没有出现 r 这个变量。前面已经提到了，r 代表的是比值，即向量$\overrightarrow{aq}$的大小与向量$\overrightarrow{ab}$大小的比值。在区域二，我们可以认为比值结果为 0,；在区域三，我们可以认为比值结果为 1；这样就可以让所有区域都使用一个表达式。

区域一：d=$\|\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[20]$\overrightarrow{ab}\|$

区域二：d=$\|\overrightarrow{ap_2} = \overrightarrow{ap_2} - 0 * \overrightarrow{ab}\|$

区域三：d=$\|\overrightarrow{bp_3} = \overrightarrow{ap_3} - 1 * \overrightarrow{ab}\|$

[18] 这里的 r 是单词 Ratio 的缩写
[19] 这里的 r 是单词 Ratio 的缩写
[20] 这里的 r 是单词 Ratio 的缩写

但是，比值是通过公式去计算$\frac{\vec{ab}\cdot\vec{ap_1}}{|\vec{ab}||\vec{ab}|}=\frac{|\vec{aq}|}{|\vec{ab}|}$=r 的，其结果并不被限制在 0 到 1。因此我们必须在结果计算出来后，对结果进行限制。

3. 计算端点向量和到目标点的向量

首先创建三个 Vector2 类型的参数，这三个参数就是端点和目标点。接着我们用其中一个端点去减目标点，便可以得到一条从端点出发，指向目标点的向量。（这里需要注意谁是被减数，因为向量指向被减的一方）

得到端点向量也是如此，用两个端点值相减。（这里依旧要注意被减数是谁）

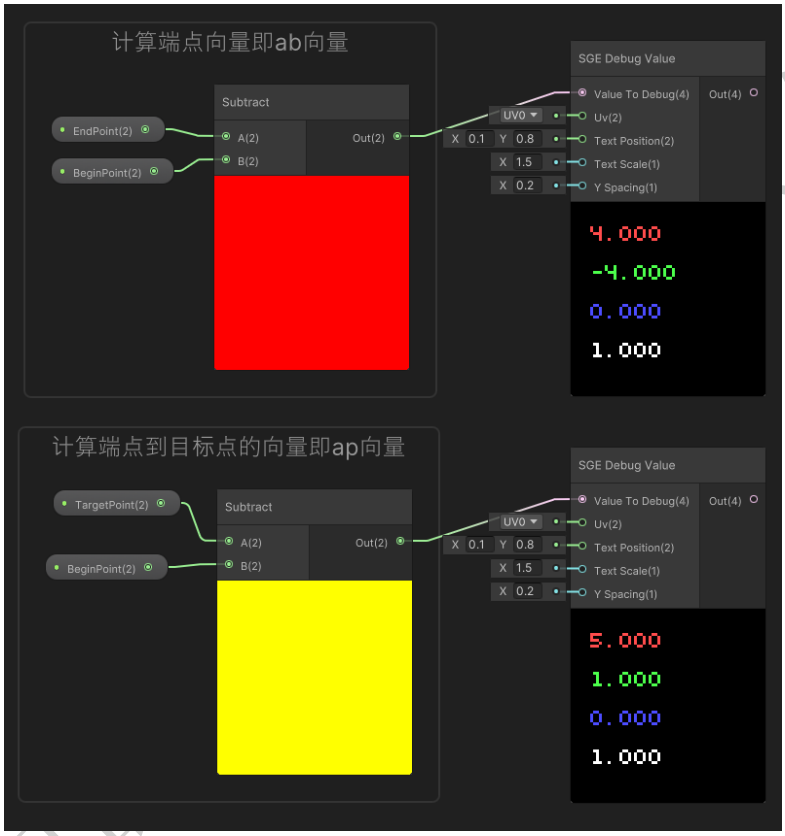


图259.　　计算得到两个向量

4. 计算 r

在我们得到了$\vec{ab}$向量和$\vec{ap}$向量后，我们就需要通过公式$\frac{\vec{ab}\cdot\vec{ap}}{|\vec{ab}||\vec{ab}|}=\frac{|\vec{aq}|}{|\vec{ab}|}$=r 去计算 r 的值了。我们需要创建两个 Dot Product Node，去计算$\vec{ab}$向量与$\vec{ap}$向量的点乘以及$\vec{ab}$向量与自身的内积。

在计算完 r 后我们必须对它值的范围进行一个限制，保证 r 在 0 到 1 之间，要限制范围可以使用 Clamp 节点或者 Saturate 节点。

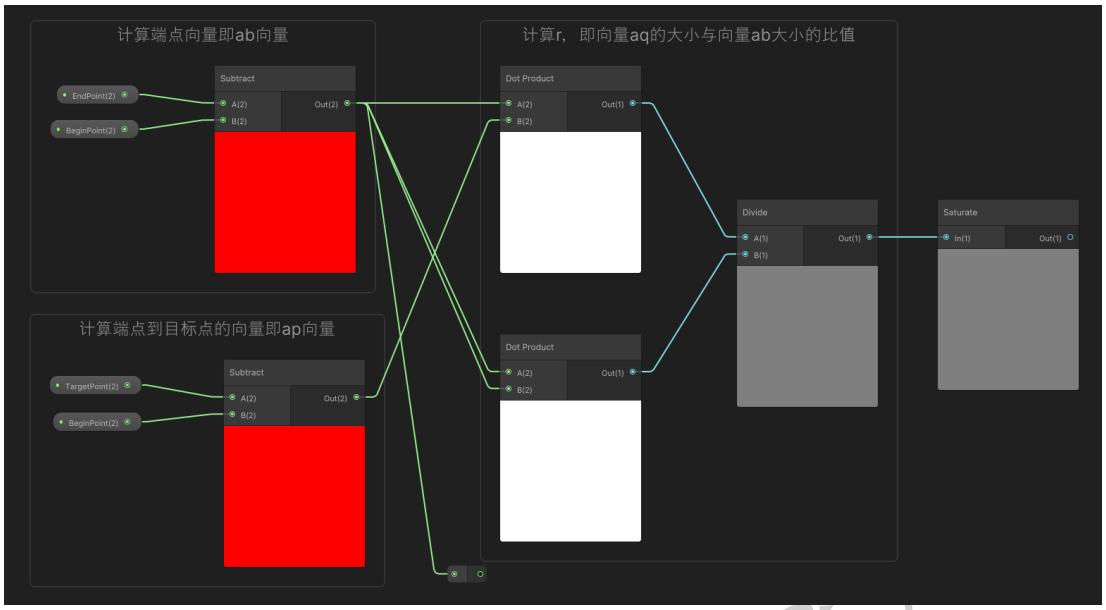


图260.　　计算 r

5. 计算$\overrightarrow{aq}$向量

因为$\overrightarrow{aq}$向量与$\overrightarrow{ab}$向量是一个共线向量，那么$\overrightarrow{aq}$向量就可以用$\overrightarrow{ab}$向量表示出来。

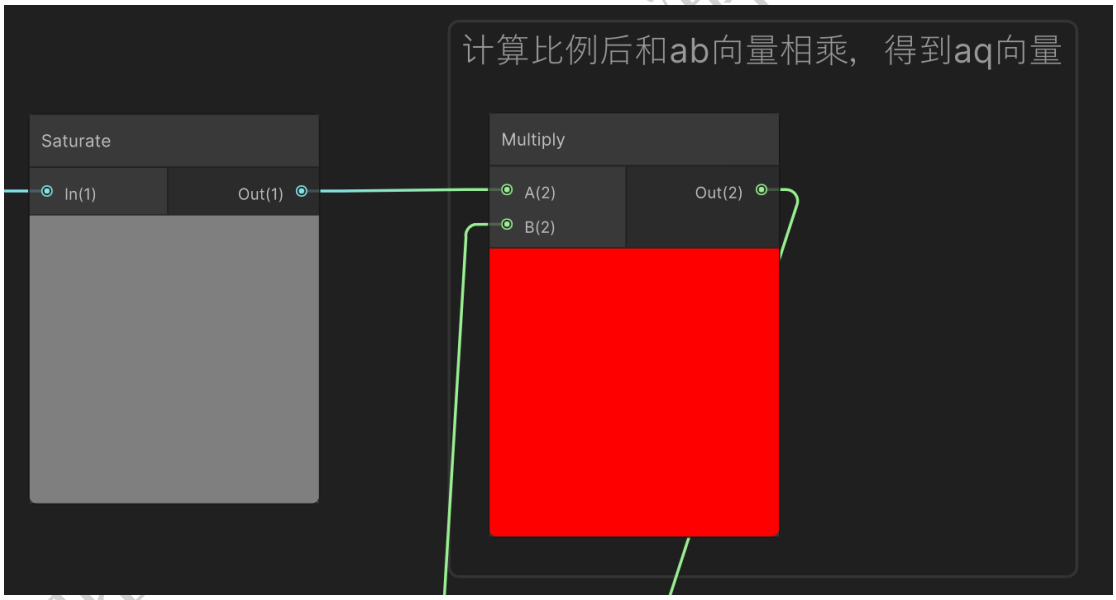


图261.　　得到 aq 向量

6. 计算点距离线段的长度

当我们知道了$\overrightarrow{aq}$向量后，就可以根据向量的运算法则，去得到$\overrightarrow{qp}$向量，然后计算向量的大小（也即点与线段的最短距离）。

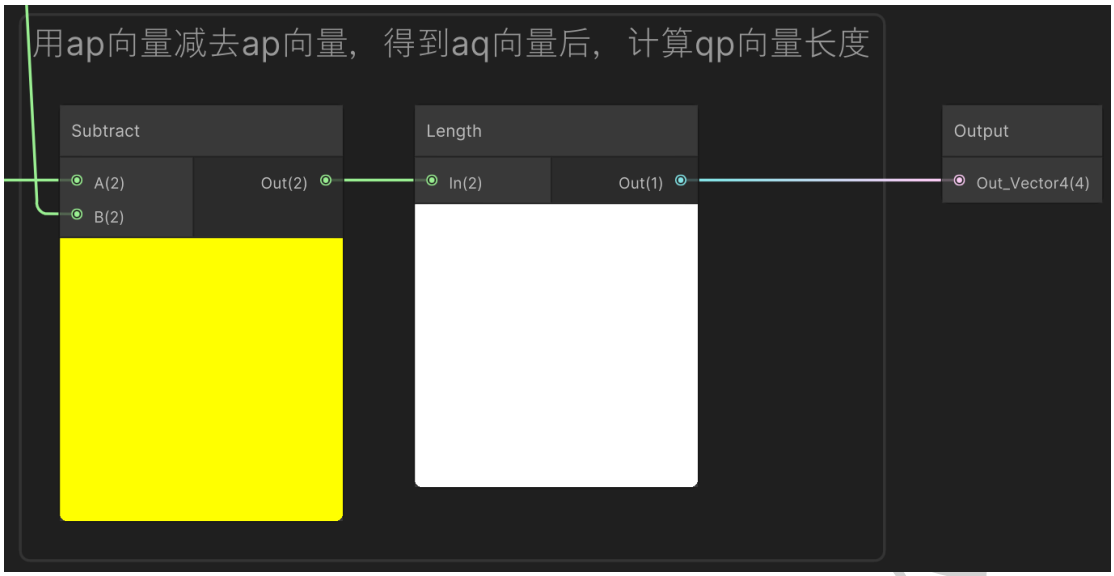


图262.　　计算 qp 向量长度

二十一、 2DSDF 矩形

对于矩形，我们可以很清楚的认识到，它的图像是关于中心点对称的。（中心点可能是原点也可能不是原点）

那么，要绘制一个矩形，所需的参数就只有宽和高两个。而点到矩形的距离公式与中心点有关，当矩形的中心点位于原点，就只需要两个参数，即宽高和目标点（如果宽和高使用 Float 类型，则是三个参数）；当矩形的中心点不是原点，就需要多加上一个参数，这个参数就是中心点的坐标。

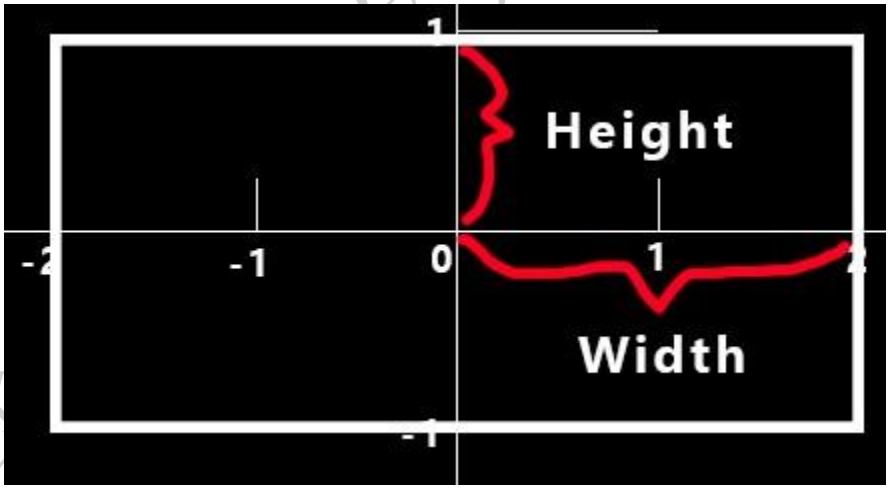


图263.　　矩形绘制原理

1. 矩形中心在原点

当矩形中心点在原点时，矩形的四个顶点就会分别处于四个象限中。因为矩形是关于中心点对称的，因此二、三、四象限的情况都可以在第一象限中处理。

要将目标点转换到第一象限，只需要对其取绝对值即可。

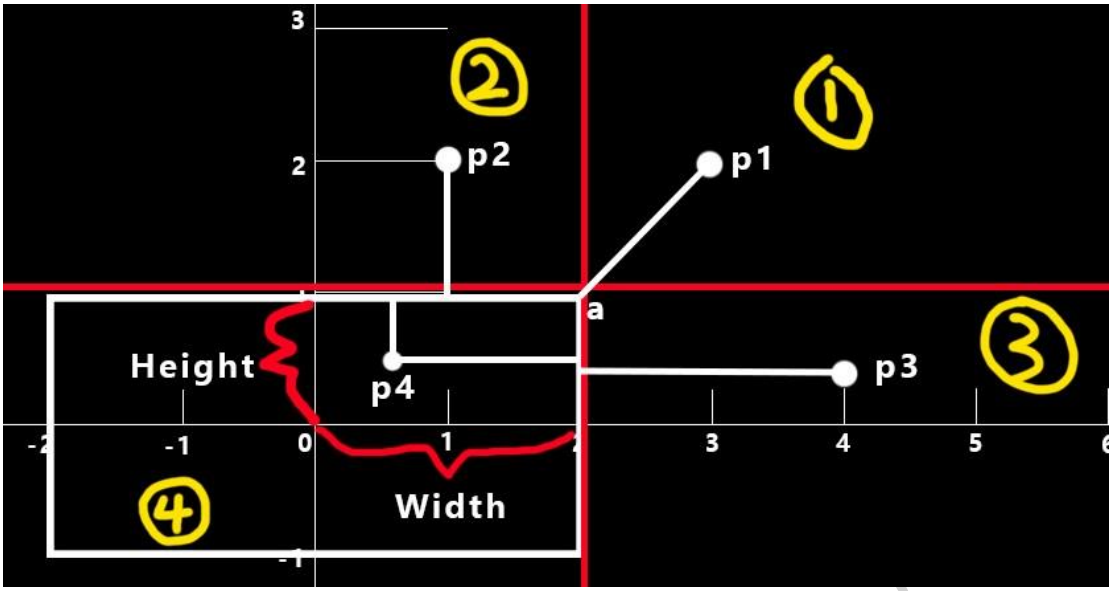


图264.　　目标点到矩形的情况划分

如上图所示，因为我们是在第一象限中处理点到矩形的距离。因此，便可以以第一象限的矩形顶点为原点，构建一个坐标轴，将目标点的情况划分为四种。

当目标点所处位置为第一种情况时，到矩形的最短距离就是向量$\overrightarrow{ap_1}$的大小。

$d=\|\overrightarrow{ap_1}\| = \sqrt{(p_1.x - a.x)^2 + (p_1.y - a.y)^2}$

当目标点所处位置为第二种情况时，到矩形的最短距离就是，目标点与顶点 Y 坐标的差。

$d=p_2.y - a.y$

当目标点所处位置为第三种情况时，到矩形的最短距离就是，目标点与顶点 X 坐标的差。

$d=p_3.x - a.x$

当目标点所处位置为第四种情况时，点位于矩形的内部，其到矩形的最短距离就是，目标点与顶点 X 坐标和 Y 坐标差值中的最大值。

$d=\max(p_4.x - a.x, p_4.y - a.y)$

因为，在第二种情况，有$p_2.x - a.x < 0$；第三种情况下，有$p_3.y - a.y < 0$。因此，我们使用 max 函数去统一前三种情况的公式，即

$d=\sqrt{(\max(p.x - a.x, 0))^2 + (\max(p.y - a.y, 0))^2}$

由于 max 函数可以直接对向量使用，所以可以将其进一步简化，得到

$d=\|\max(p - a, 0)\|$（由向量运算可知，p-a 就是向量ap）

现在我们再来单独考虑第四种情况，如果我们将第四个区域的点带入上面的公式，得到的值为 0，而前三个区域所得值一定大于 0。（因为在前三个区域，$p.x - a.x$和$p.y - a.y$至少有一个不为 0）

区域四中的点，其坐标数值一定小于顶点，所以使用$\max(p.x - a.x, p.y - a.y)$获得最短距离。又因为，我们要求公式只能适用于区域四的点，所以我们需要将其他区域排除，也就是获得区域四点的最短距离后，让其和 0 比较，取它们中最小的那个。（区域四的点和顶点相减必定小于 0）

整合上面信息，可以得到区域四的公式 $d= \min(\max(p.x - a.x, p.y - a.y), 0)$。（虽然该公式得到的区域四值为负，但我们会在程序中计算它的大小，所以并没有问题）

最后，我们将所有区域的公式进行一个整合，得到最终的公式。

$d=\|\max(p - a, 0)\| + \min(\max(p.x - a.x, p.y - a.y), 0)$

2. 矩形中心不在原点

当矩形的中心点不在原点，我们就需要额外添加一个参数，即矩形的中心点。因为矩形中心点不在原点的情况过于多样，因此我们很难得到一个统一的公式。但是，因为我们计算的是距离，所以我们可以将矩形和目标点同时进行平移，保证矩形和目标点的相对位置不发生变化，然后使用中心点在原点的公式进行计算。

3. 创建所需参数

这里只对中心点在原点的情况，进行设置。我们创建三个参数，两个 Float 类型的参数和一个 Vector2 类型的参数。这两个 Float 类型的参数，对应的就是矩形的宽和高；Vector2 类型的参数，对应的就是目标点的坐标。

图265. SDF 矩形参数设置

4. 计算向量$\overrightarrow{ap_1}$

首先需要使用 Absolute 节点，将目标点转换到第一象限，然后和端点相减得到$\overrightarrow{ap_1}$向量。再让$\overrightarrow{ap_1}$向量和 0 进行比较，如果目标点是第二种情况，则$\overrightarrow{ap_1}$向量的 X 分量将为 0；如果目标点是第三种情况，则$\overrightarrow{ap_1}$向量的 Y 分量将为 0。

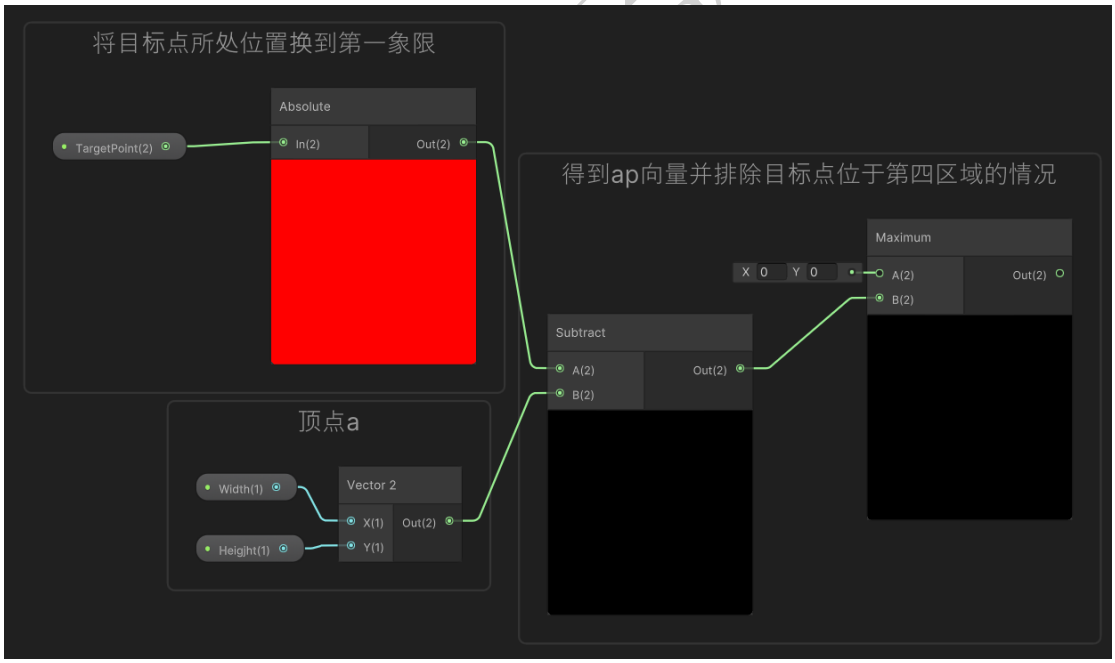


图266. 计算向量 ap1

5. 计算目标点位于区域一、二、三时的模

我们使用 Power 节点和 Split 节点，去分别计算 X 分量和 Y 分量的平方，再对其相加后所得的结果开方。

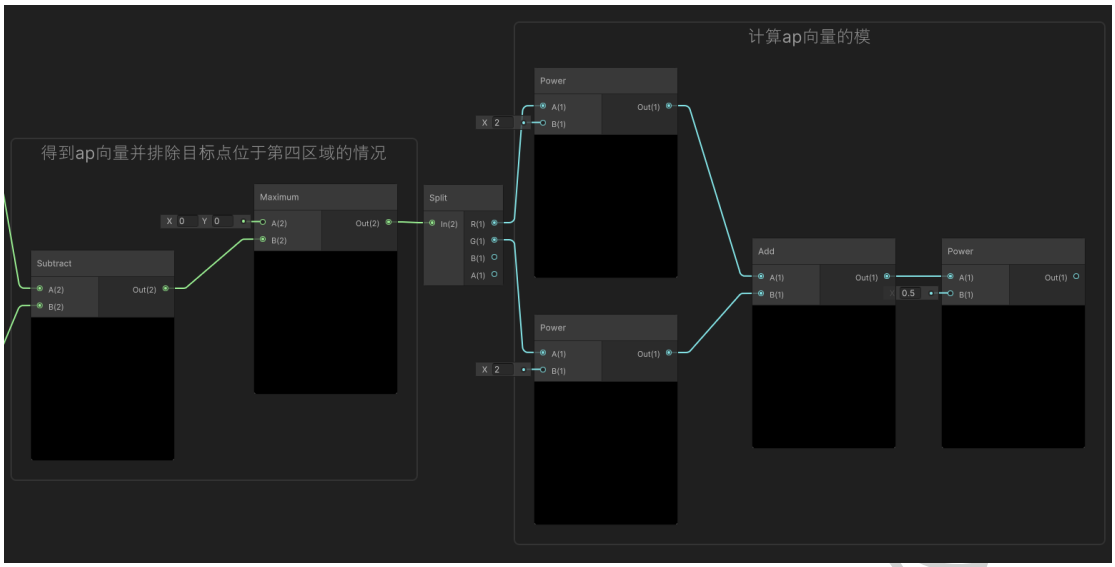


图267.　　计算目标点位于区域一、二、三时的模

现在我们得到了公式的前半部分，下面我们就将计算后半部分。

6. 计算区域四点的最短距离

为了计算区域四种，点到矩形的最短距离，我们首先需要使用 Split 节点，将目标点的 X 分量和 Y 分量分离出来。接着用矩形的顶点，去和 X 分量、Y 分量做减法。将所得结果进行比较，其中数值最大的一个就是目标点距离矩形的最短距离。（因为区域四中的点，其坐标一定是小于矩形顶点的坐标，所以计算的所得值为负）

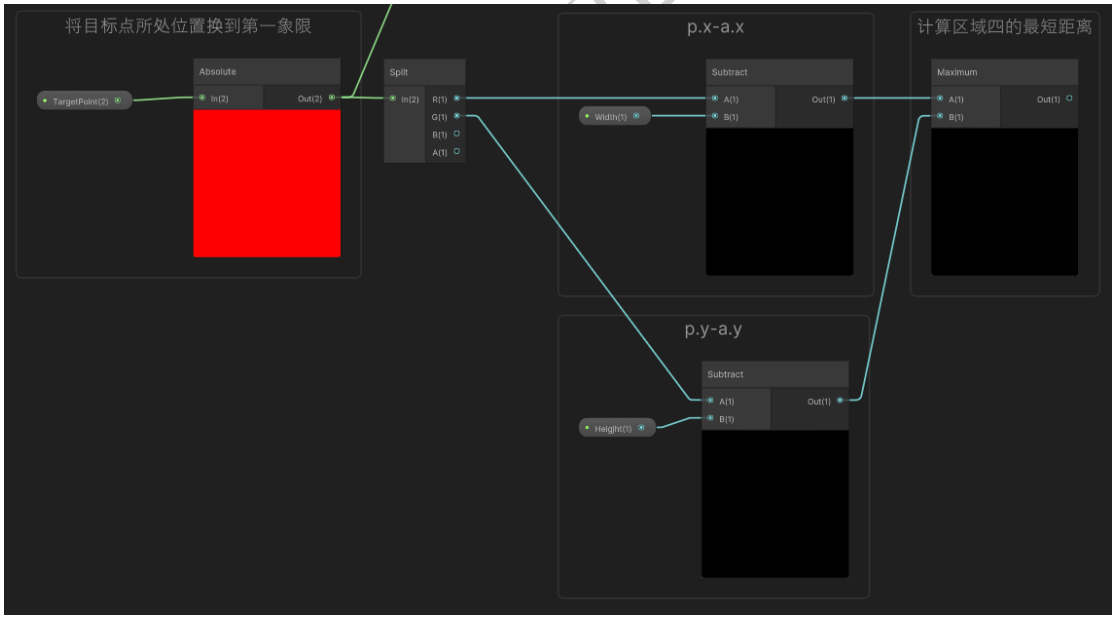


图268.　　计算区域四中点的最短距离

7. 排除目标点不位于区域四的情况

当目标点不位于区域四时，上面的 ShaderGraph 输出值将大于 0，因此我们需要使用 Minimum 节点将其数值归 0。

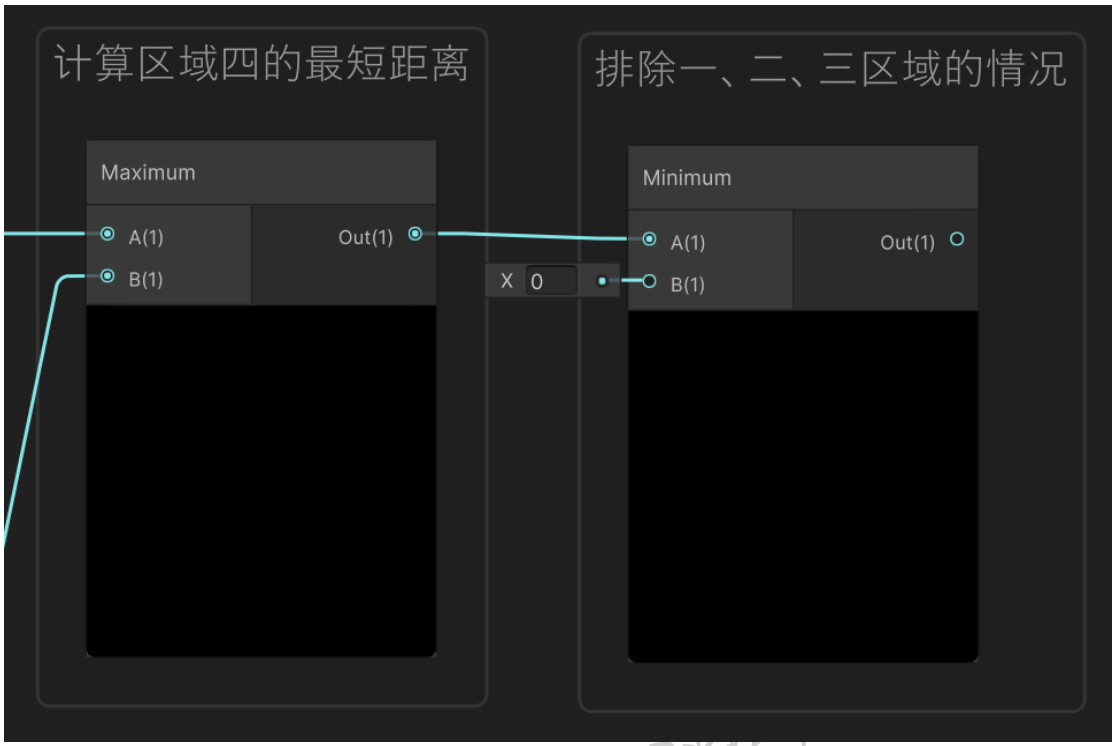


图269.　排除目标点不位于区域四的情况

8. 计算最终结果

使用加法节点，将前半部分和后半部分相加。当目标点位于区域时，我们所得的最终结果一定是小于 0 的。

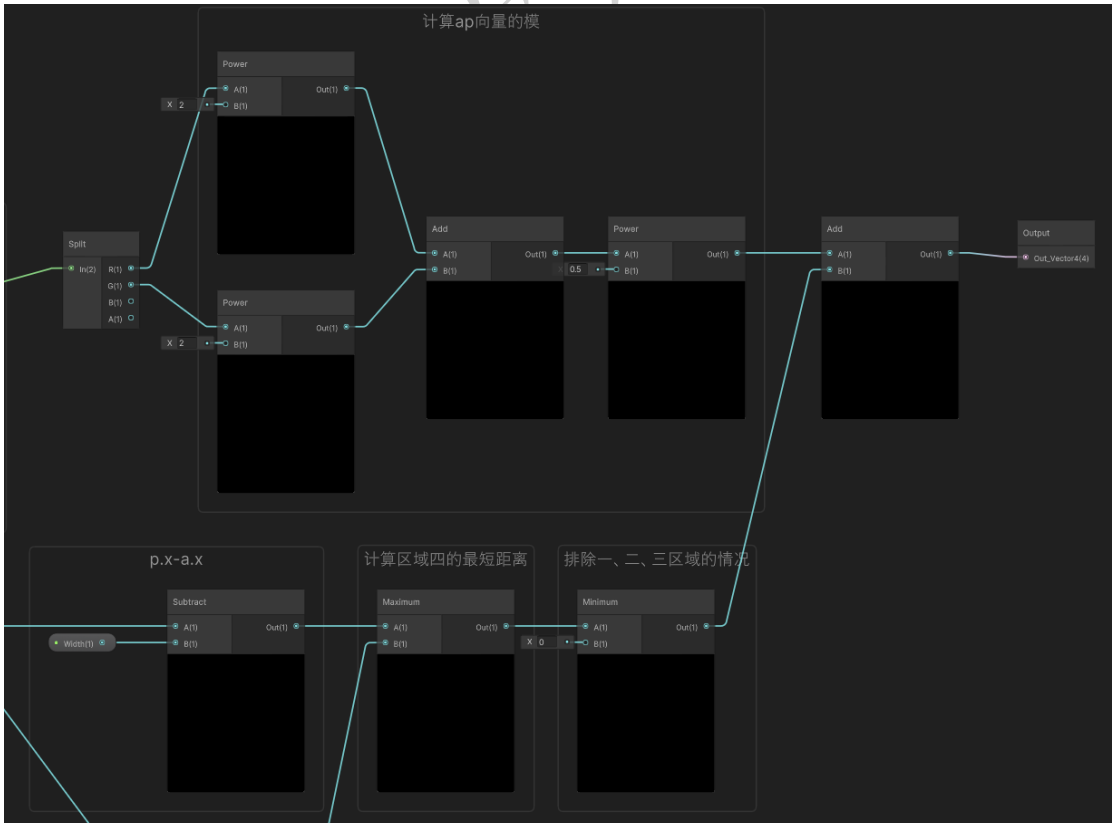


图270.　SDF 矩形最终结果

9. 缺点

从前面连接的 ShaderGraph 可以看到，仅仅只是简单的数学公式，我们却运用了大量的节点，这比起 ShaderLab 显然更加耗时。因此，ShaderGraph 并不适合进行大量的数学运算，一旦所需的数学公式一多，其节点的连接就会看起来非常混乱了，也不够直观了。（本来 ShaderGraph 就是为了能够更加直观的感受流程的变化）

要进行大量的数学运算时，还是使用 ShaderLab 去进行编码，这会使之更加的易读。

二十二、2D 简易放大镜

1. 核心思想

利用圆的 SDF 去生成一张蒙版，然后用这张蒙版去影响放大后的图像和原始图像，最后只需将两张图像进行组合输出即可。

2. 创建 ShaderGraph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

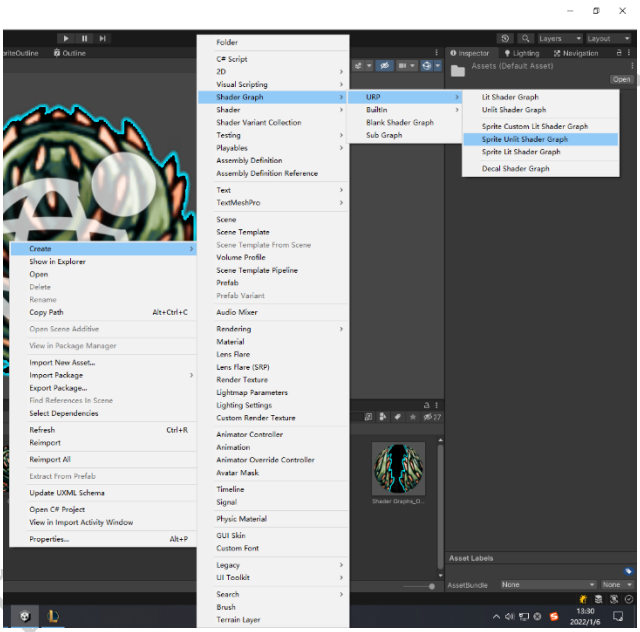


图1. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[21],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图2. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

[21] adj. 普遍的；通用的 n. 普遍原则；普遍看法

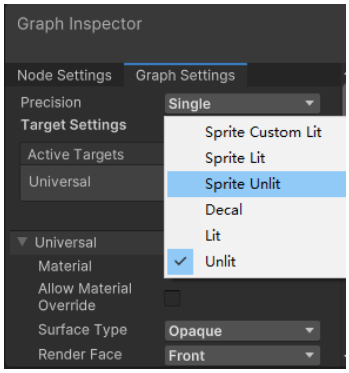


图3. 修改 Material

图4.

3. 添加参数

我们先将放大镜效果所需的参数添加到参数面板。名叫 MainTex 的 Texture2D 主纹理是必不可少的，然后是三个 Float 类型的参数和一个 Vector2 类型的参数。二维向量就是圆心的坐标，三个 Float 参数分别是圆半径、缩放率及蒙版的边缘平滑度。

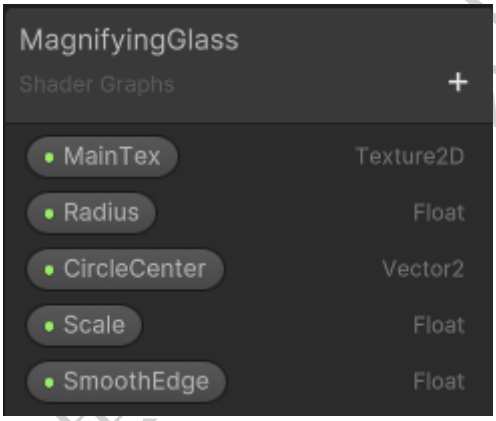


图5. 放大镜效果的参数面板

4. 放大 UV

要实现放大镜的效果，我们首先需要做的就是将 UV 整体进行缩放。创建一个 UV 节点和减法节点，在缩放前先减去圆心坐标是为了便于后面使用圆的 SDF。

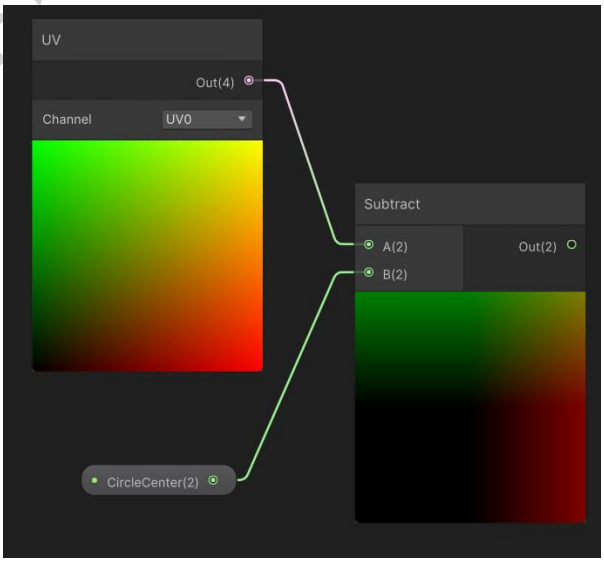


图6. 偏移 UV 坐标，同时为圆的 SDF 做准备

接着我们便可以使用乘法节点，去缩放偏移后的 UV。完成缩放后，再使用加法去抵消之前的偏移量。接着创建一个 Sample Texture2D 节点，将完成缩放和偏移修正后的 UV 以及主纹理给予它，我们就可以看到缩放后的效果了。

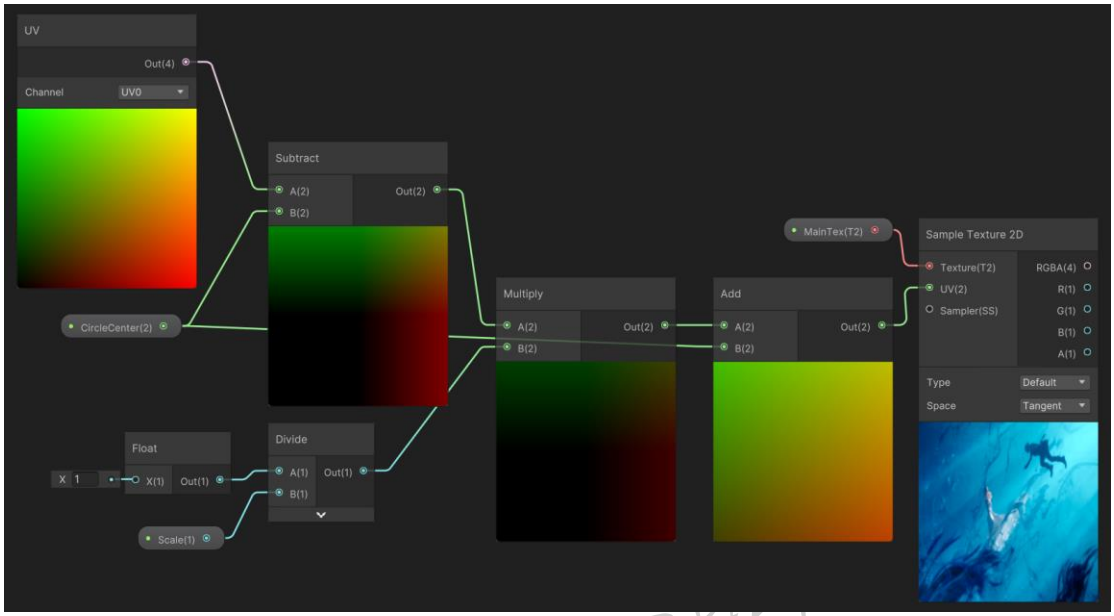


图7. 得到放大镜效果的缩放效果图像

5. 利用圆的 SDF 创建蒙版

当我们拥有了缩放后的图像后，就可以利用圆的 SDF 去生成蒙版了。在缩放之前我们就已经有了 UV 坐标到圆心的向量，因此我们只需再使用 length 节点去获得距离场，然后再减去圆的半径就可以得到有向距离场了。

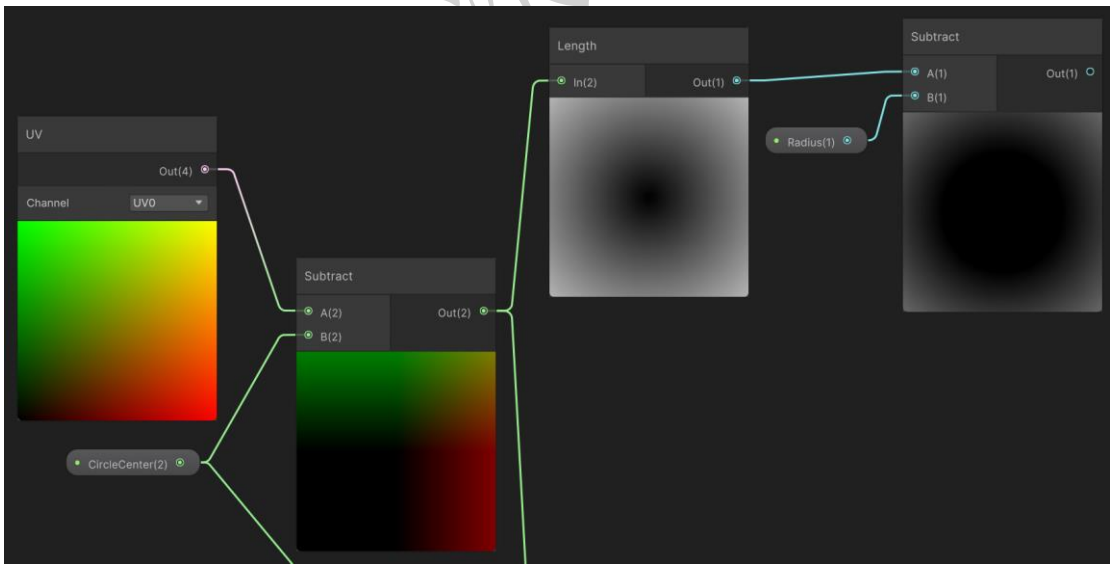


图8. 利用之前得到的向量，计算出圆的有向距离场

有了有向距离场以后，我们就可以使用 SmoothStep 节点和之前创建的 SmoothEdge 参数，去生成一个带有边缘平滑过渡的蒙版了。

这里再说明一下 SmoothStep 节点，SmoothStep 节点的输入值如果介于 Edge1 和 Edge2 的值之间，则返回 0 和 1 之间的平滑 Hermite 插值结果。如果输入值小于 Edge1 的值，则返回 0；如果大于 Edge2 的值，则返回 1 。

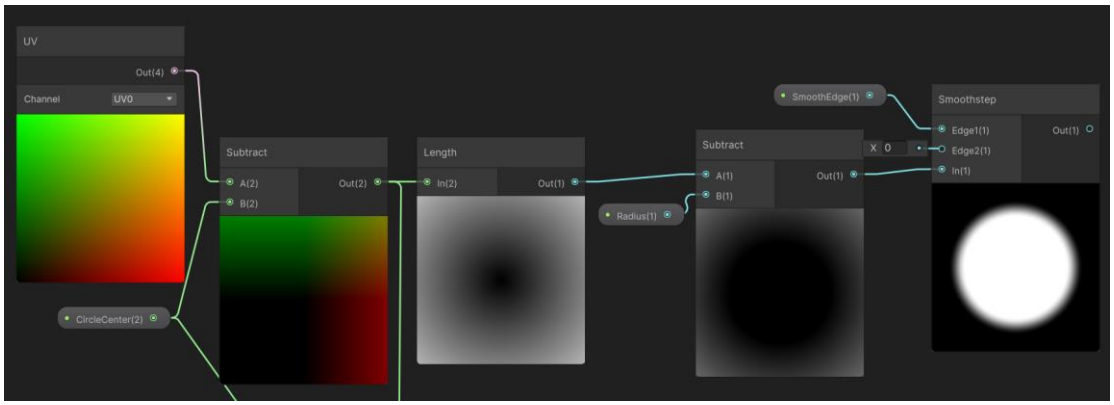


图9. 利用圆的距离场和 SmoothStep 函数，生成带边缘过渡的蒙版

现在有了一张蒙版了，这张蒙版将被用于缩放后的图像。我们需要使用 One Minus 节点，去得到另一个作用于原始图像的蒙版。

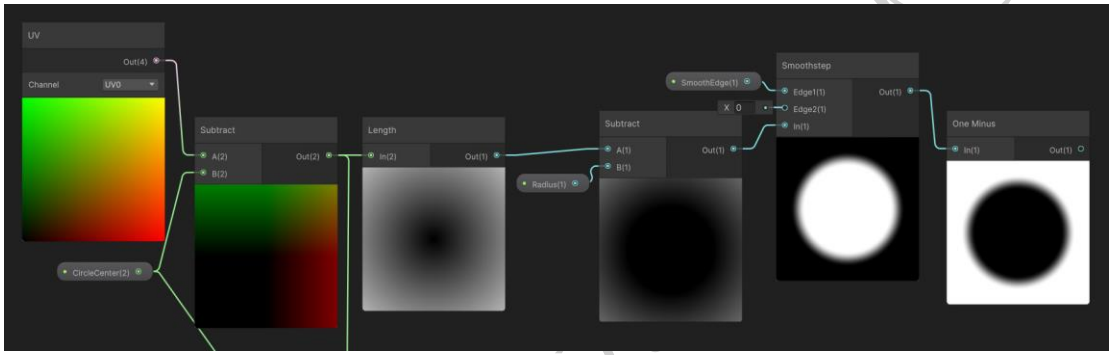


图10. 得到用于原始图像的蒙版

6. 得到最终效果

现在我们已经拥有了两张蒙版了，下面就是让它们去影响图像了，在最后将被影响的图像进行组合，并将其输出到片元着色器的 Base Color 端口和 Alpha 端口。

哔哩哔哩欢乐生

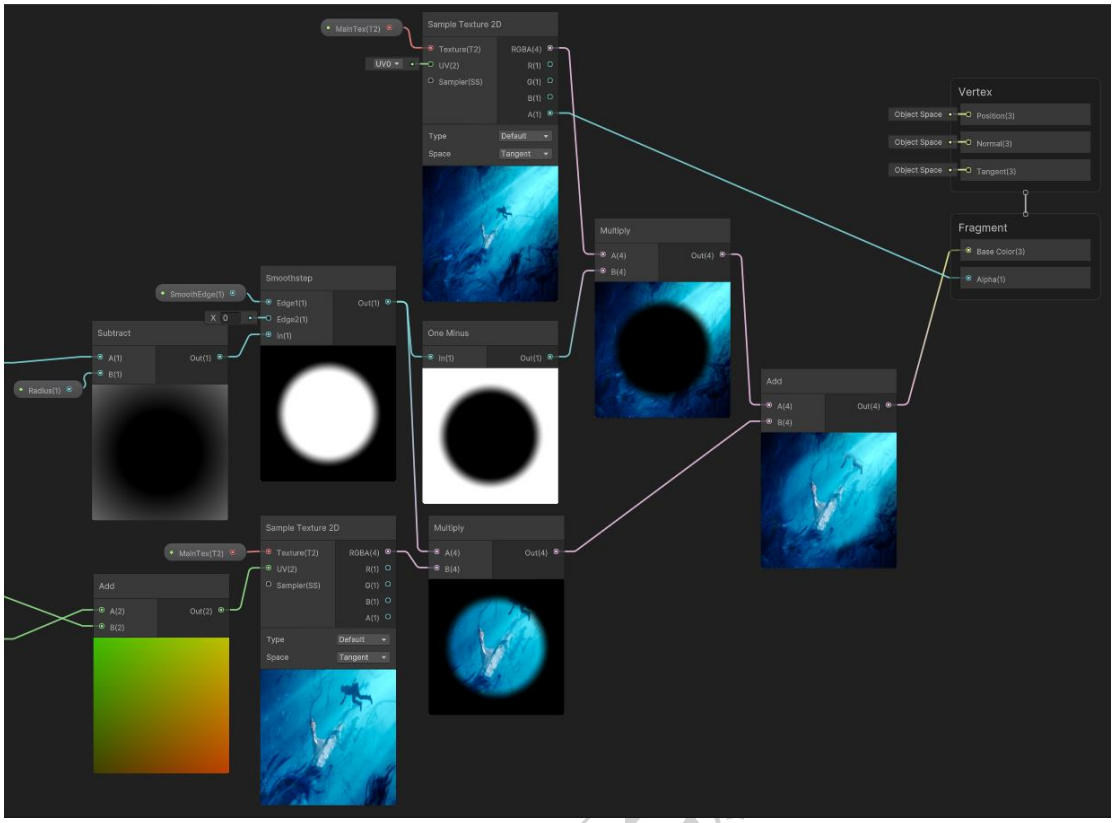


图11.　　组合得到放大镜效果的最终效果

7. 利用 C#脚本控制圆心坐标

创建一个用于测试的 C#脚本，写下下面的内容。

```
public Material testone;
private float textureWidth,textureHeight;
private float screenWidth, screenHeight;

private void Awake()
{
    //获取材质球
    testone = this.gameObject.GetComponent<Image>().material;
    //主纹理宽高
    textureWidth = testone.GetTexture("_MainTex").width;
    textureHeight = testone.GetTexture("_MainTex").height;
    //屏幕宽高
    screenHeight = Screen.height;
    screenWidth = Screen.width;
}

private void Update()
{
    testone.SetVector("_CircleCenter", new Vector4(Input.mousePosition.x /
screenWidth, Input.mousePosition.y / screenHeight, 0, 0));
}
```

我们使用鼠标的坐标除以屏幕的宽高，去近似一个得到 UV 坐标。这里只是用于测试，无需过于精确。

8. 其他方法

如果按照前面的方法，我们就需要对同一张纹理采样两次，这会造成额外的性能消耗。实际上，我们可以只使用一次采样就可以完成放大镜效果，那就是直接对 UV 进行处理。

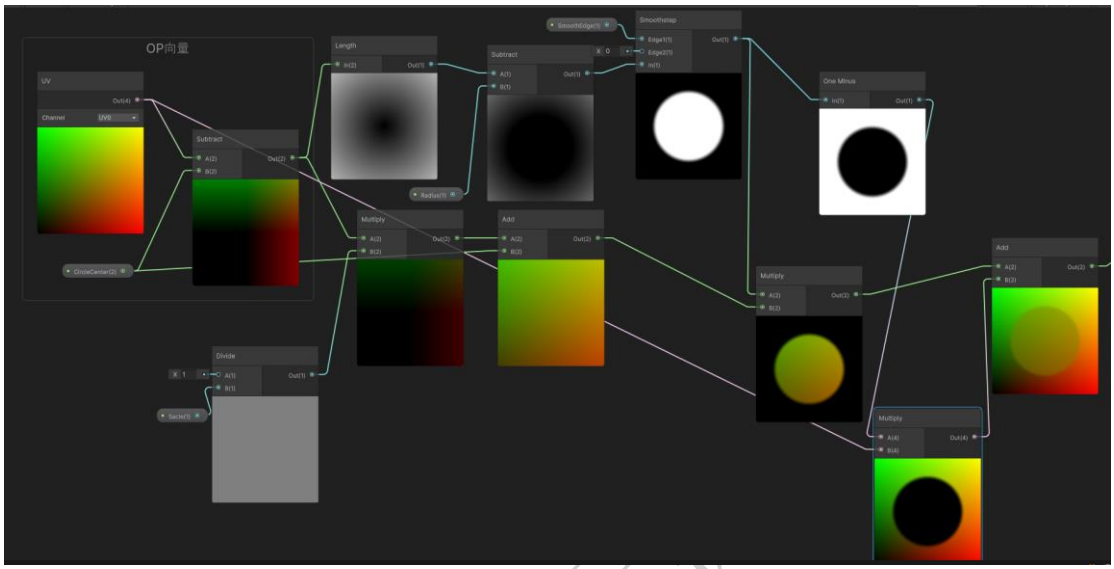


图12. 直接对 UV 使用蒙版

直接对 UV 使用蒙版所得到的效果，和对图像使用蒙版是不一样的。前者是在过程中进行了截断，后者是在已得结果上做修改。

二十三、 3D 菲尼尔边缘光

1. 什么是菲尼尔效应

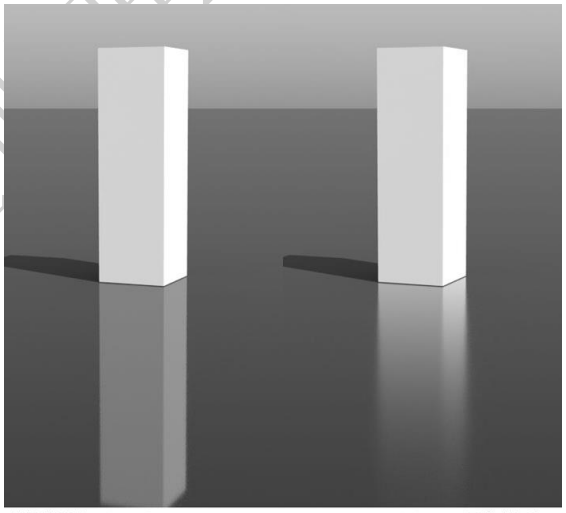


图13. 菲尼尔效果的示例

简单来说，就是某种材质在不同距离上呈现出的不同反射效果。

同一个物体，观察的角度不同，其表面的反射率是不一样的。菲涅耳效应模拟的就是物体材质反射率随角度改变的效果。

图14.　　肥皂泡的菲涅尔效果

在肥皂泡的边缘处，即视线与表面夹角处比较小时，反射效果更强烈；在肥皂泡中心附近，即视线与表面夹角近乎垂直时，看起来更透明一些。[这些反射只出现在球体的边缘（即当它的表面处于掠射角时），并且随着 Material 的平滑度增加，它们变得更加明显和清晰。]

1） 什么是入射角

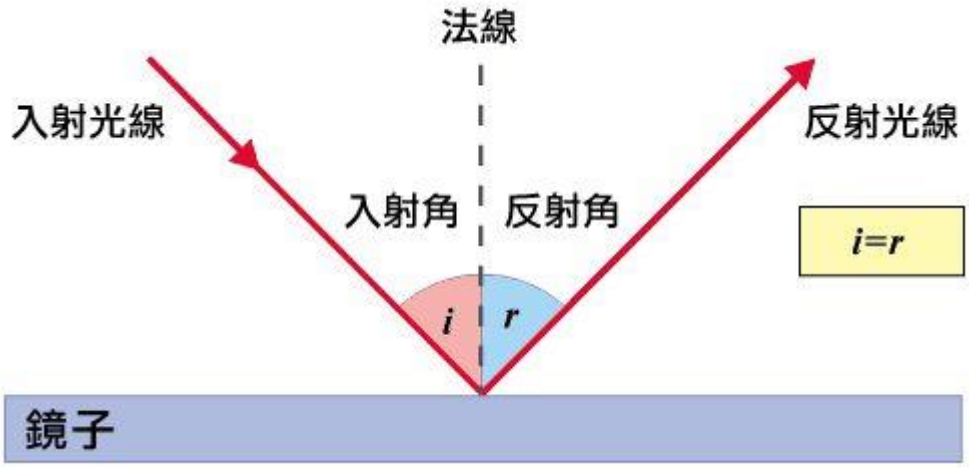


图15.　　入射角示例

如果从观察者的眼睛里发出一道光，射向某一个物体，这道光与物体交点处的切线所形成的夹角就是入射角。

如果射向的物体是一个水平面，那么入射角就是视线与水平线之间的锐角。

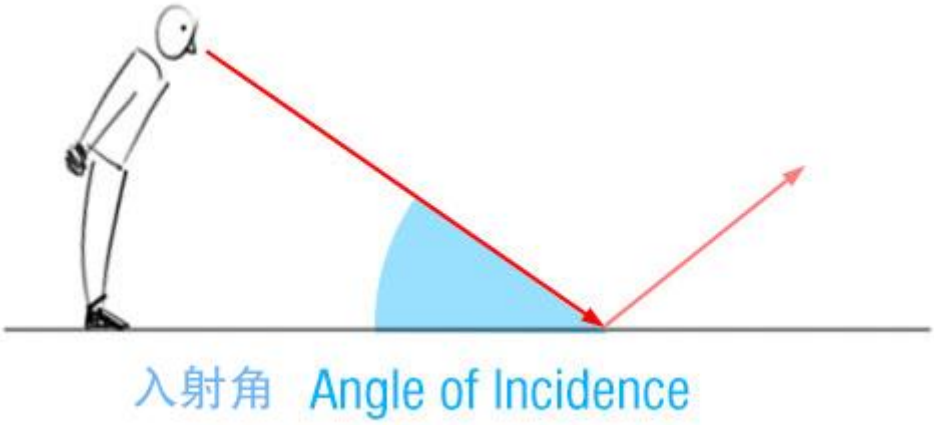


图16.　　水平面的入射角示例

如果射向的物体是球体，那么入射角就是视线与球体在交点处的切线之间的锐角。

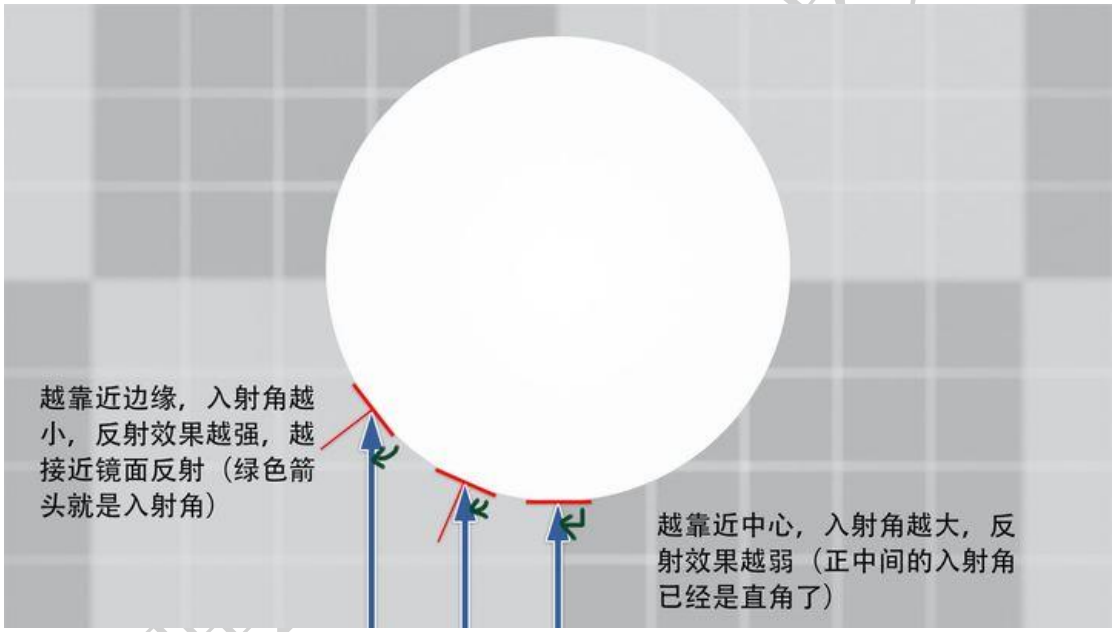


图17.　　球体的入射角示例

2）入射角特性

入射角有两个非常重要的特性。

① 随着观察距离的变化而变化，距离越近，入射角越大；距离越远，入射角越小。

② 入射角大小影响反射效果，入射角越大，反射效果越弱；入射角越小，反射效果越强。

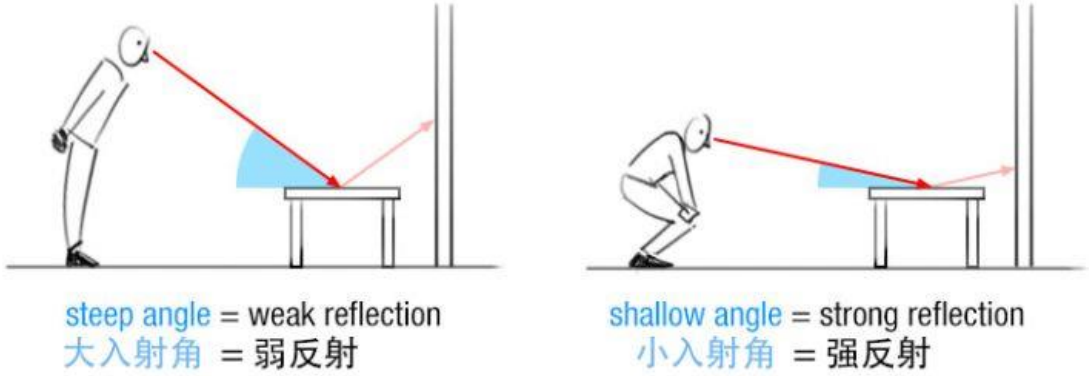


图18. 入射角特性示例

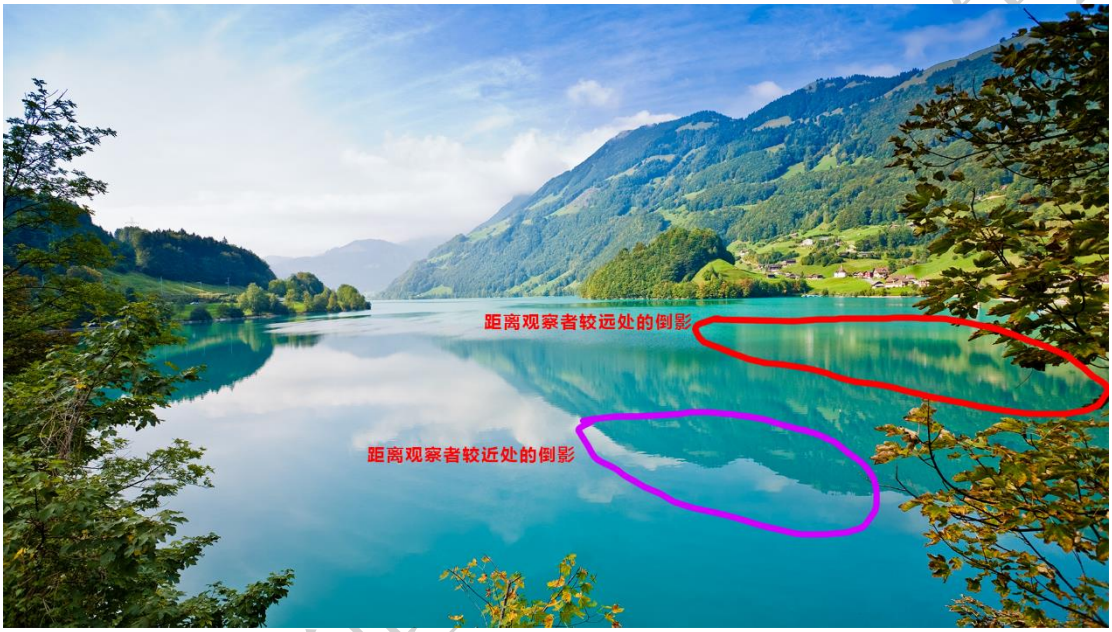


图19. 倒影的清晰程度与观察者和倒影的距离有关

上图有一座山在水面形成了倒影，观察者站在湖边观察山的倒影，会看到观察对象的距离不同，其倒影的清晰程度也会发生变化。山尖的倒影离人更近一些，看起来就更模糊一些，山脚的倒影离人更远一些，其倒影就清晰一些。

这是因为，在观察山尖的倒影时，入射角更大，所以反射效果更弱；在观察山脚的倒影时，入射角更小，所以反射效果更强。

简单的讲，就是视线垂直于表面时，反射较弱，而当视线非垂直表面时，夹角越小，反射越明显。如果你看向一个圆球，那圆球中心的反射较弱，靠近边缘较强。不过这种过度关系被折射率影响。

图20.　水面的菲涅尔案例

反射效果越弱，水体看起来越透明，像空气；反射效果越强，水体看起来越光滑，像镜子。那么在视觉上呈现给人的感受是，上图山尖的反射效果弱，因此山尖的倒影看起来不清晰，甚至看不到倒影；山脚的反射效果强因此山脚的倒影像镜子一般清晰。

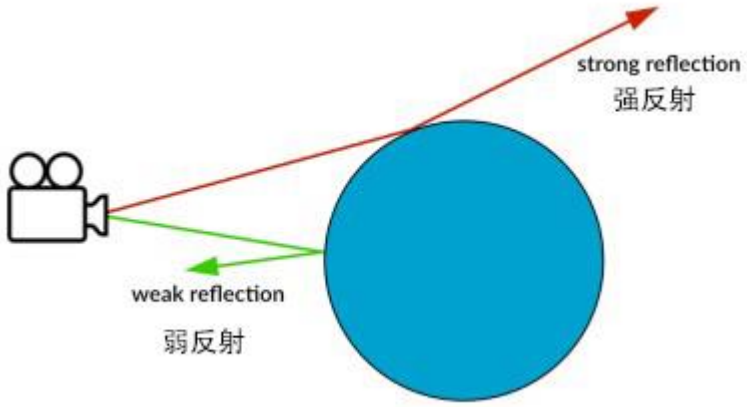


图21.　曲面的反射示例

在曲面反射中，离观察者越远的曲面，由于入射角更小，越容易呈现出清晰的镜面反射。反而是正对观察者的曲面，由于入射角接近直角，反射效果会很弱，几乎看不到反射。

我们可以做一个简单的理解，菲涅尔效果就是在反射效果中，离你近的反射得更模糊，离你远的反射得更清晰。

图22.　　使用菲涅尔效果和不使用菲涅尔效果的对比

菲涅尔是自然环境中很常见的光学现象，在游戏和动画制作中，为了表现写实的光影效果，很多材质都会允许开启菲涅尔（例如水面、玻璃、瓷釉、矿物等等）。在二维绘画中，菲涅尔效果也被用于强化画面的写实性。如果追求写实风格的画面渲染，菲涅尔效果是一种很实用的技术手段。

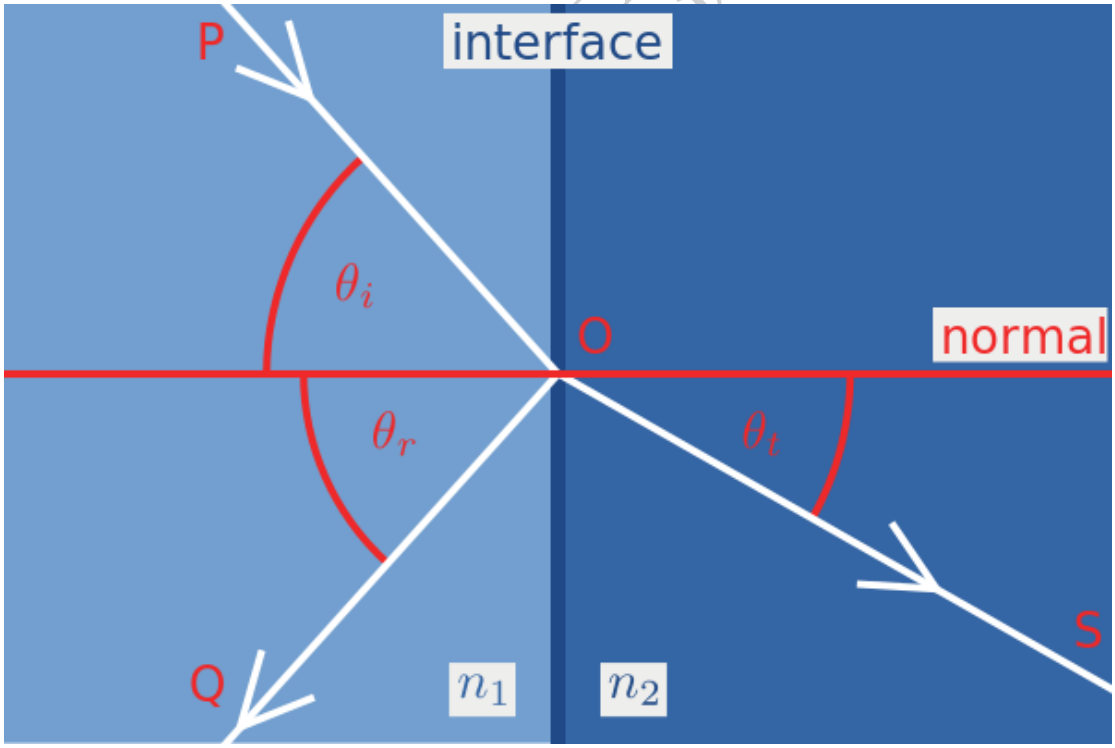


图23.　　入射光线、折射光线与反射光线

2. 创建 LitShaderGraph

处于 3D 场景中的真实对象，通常是会受到场景中光源的影响的。如果是 NPR 渲染，而不是真实感渲染，那么其光照效果就需要我们自己去自定义了。

要使我们创建出的 Shader 实例，能够被场景的光源所影响。我们就需要创建 Lit Shader Graph，而不是 Unlit Shader Graph，Unlit Shader Graph 是表示不受光照影响。

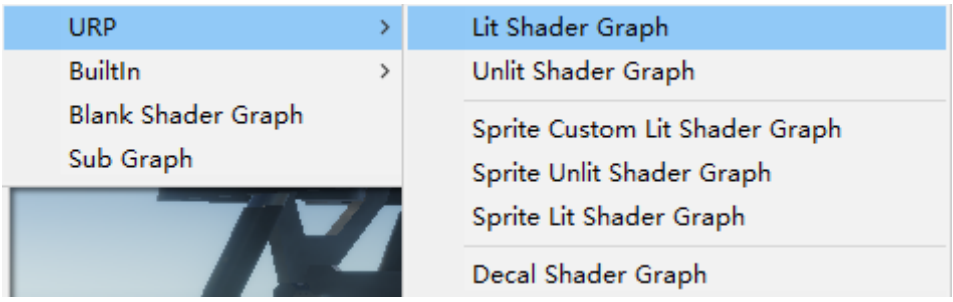


图24.　　创建 LitShaderGraph

3. 创建所需的基本纹理参数

因为我们是真实感的渲染，这会使用到一些基本真实感渲染的纹理，其中就包括法线贴图、金属度贴图和漫反射贴图。

我们首先在 ShaderGraph 的参数面板里，创建三个 Texture2D 类型的参数（具体需要多少，可以根据自己的情况去增加或减少），并根据所需的纹理类型去设置它的名称，漫反射贴图的名称就命名为 MainTex。

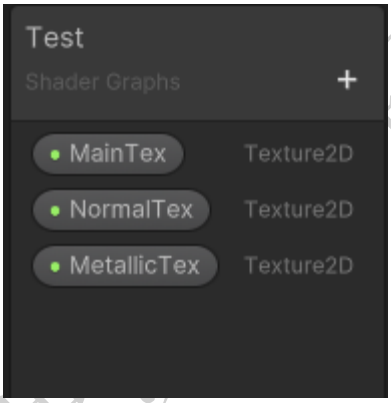


图25.　　菲涅尔边缘光的基础纹理参数

4. 输出基础纹理

1） Albedo Map

Albedo Map 是一张不包含任何光照信息的图像，它代表物体表面的颜色，我们会将其输出到片元着色器的 Base Color 端口。

哔哩哔哩 游戏人生

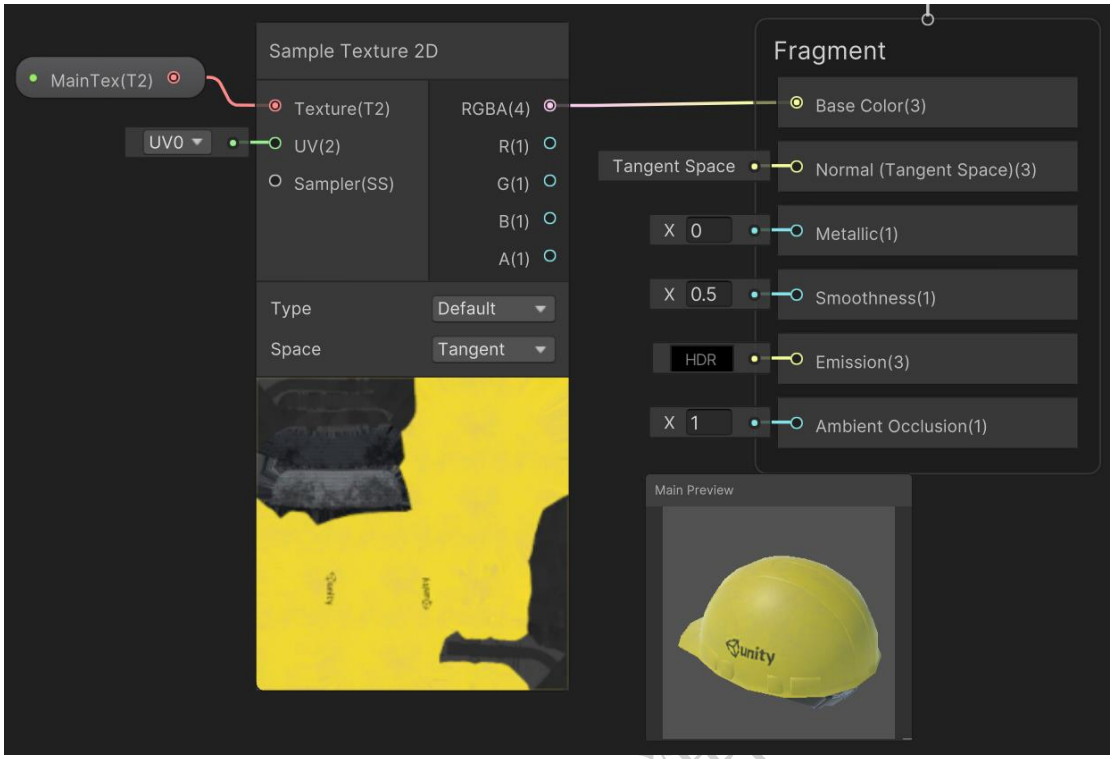


图26.　　菲涅尔效果的漫反射贴图

将其输出到片元着色器后，我们就可以在 ShaderGraph 的预览窗口看到实际的应用效果了。

2） Normal Map

法线贴图是凹凸贴图的一种，是一种特殊的纹理，可以让用户为模型添加表面细节。换句话说，就是去修改光源作用在物体上的效果，使其看起来更真实。在 ShaderGraph 中使用 Normal Map 时，需要将 Sample Texture2D 的类型切换为法线的类型，只有这样该 Normal Map 才能够被正确应用（Normal Map 对应的是片元着色器的 Normal 端口）。

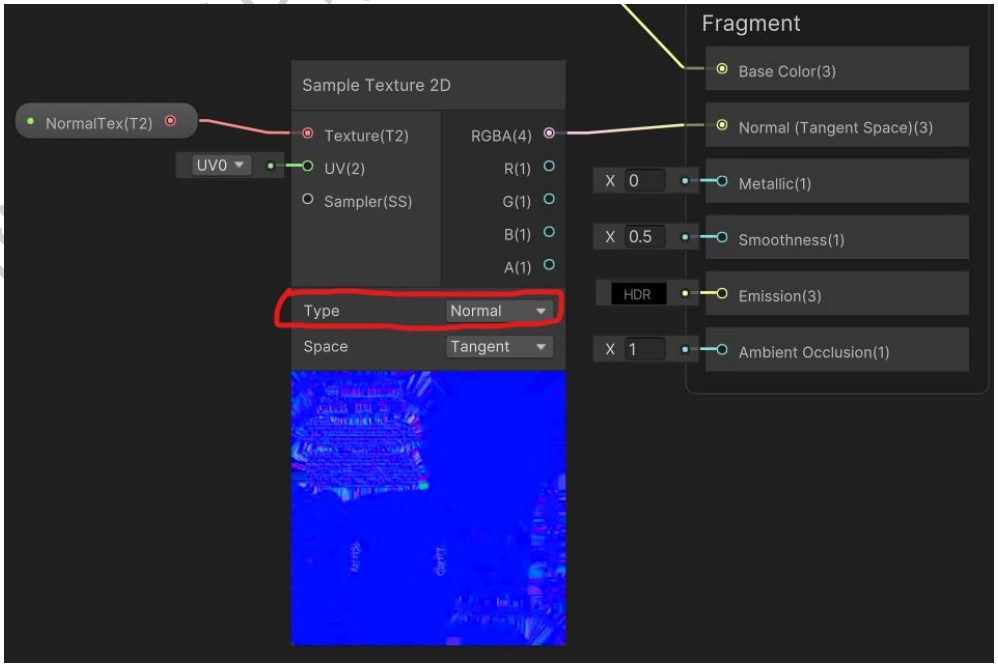


图27.　　菲尼尔效果应用法线贴图

3） Metallic Map

Metallic Map 决定了物体表面的“金属感”，当物体表面具有更多金属性时，它会反射更多的环境光，并且其漫反射纹理将变得不那么明显。在 ShaderGraph 中，我们依旧需要将其输出到片元着色器中对应的端口，也即 Metallic 端口。

5. 使用菲涅尔节点

ShaderGraph 为我们内置了菲涅尔节点，我们可以使用空格键创建该节点。

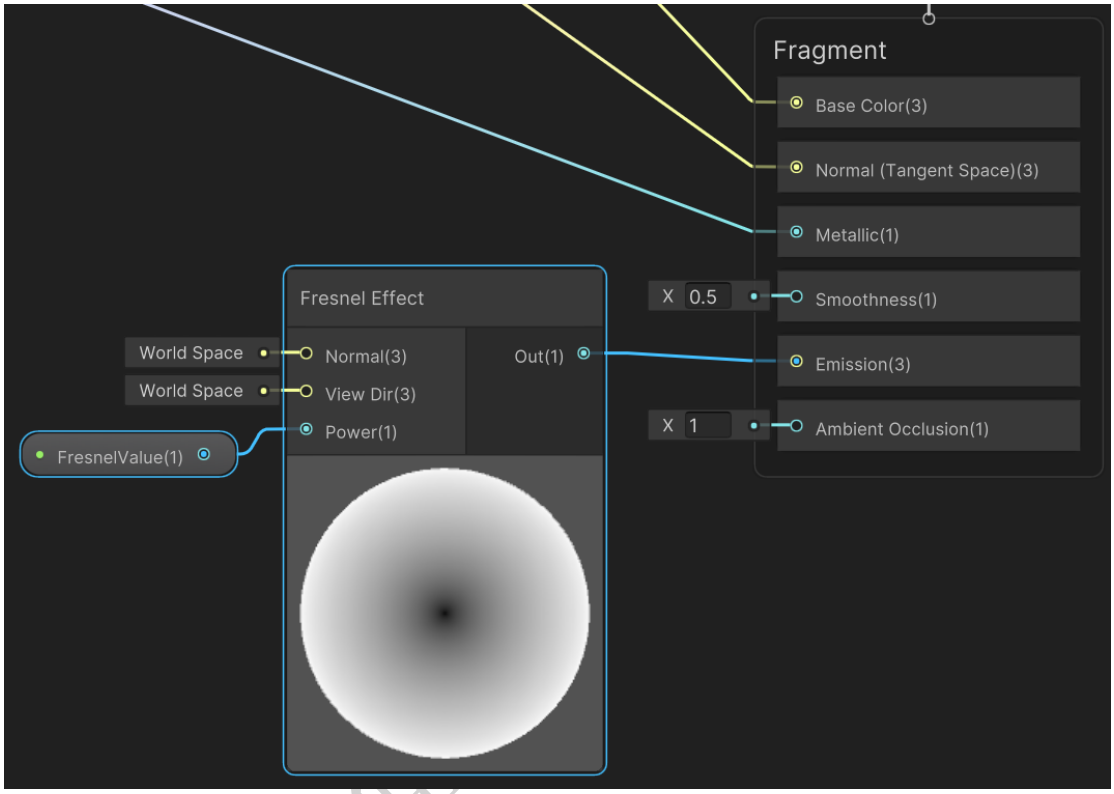


图28. 直接使用菲涅尔节点

也可以使用法线方向和视口方向，去自己完成菲涅尔节点。

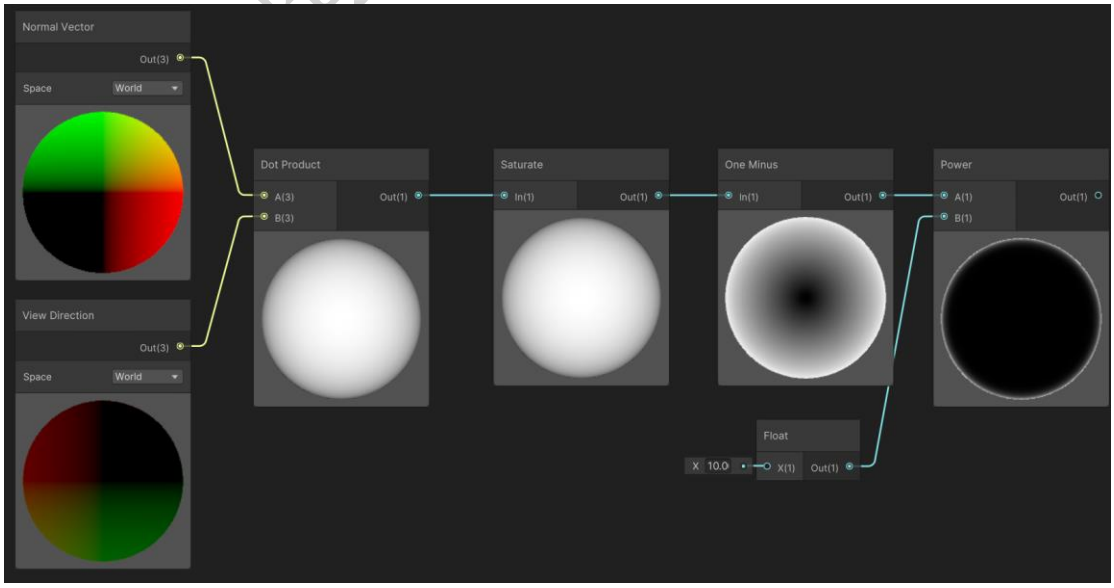


图29. 自建菲涅尔节点

通过调节菲涅尔节点的 Power 值，可以调整其作用范围。我们可以在参数面板中创建一个 Float 类型的参数，用这个参数去控制菲涅尔节点的 Power 端口值。

现在我们就可以将其输出到片元着色器的自发光端口，即 Emission[22]端口。

6. 修改自发光颜色

我们已经有了菲涅尔边缘光，但是我们在后期一定会遇到修改其颜色的需求。为此，我们可以在参数面板创建一个 HDR 的 Color 参数，用这个参数去调整最终呈现的自发光颜色。这里使用 HDR，可以方便 Bloom 后处理视觉效果的呈现。

7. 重映射 FresnelValue

现在已经完成了所有的效果，但 FresnelValue 的值依旧不是很理想，且我们希望能够自己决定菲涅尔效果是否开启。我们设想，当开启时的 FresnelValue 值为 0，则是默认的影响区域；当开启时的 FresnelValue 值为 1，则将菲涅尔效果的影响区域降到最小。

为此我们需要先使用 Remap 节点，将 FresnelValue 的值进行一个重新映射，以保证其符合我们的需求。我们需要根据 Remap 节点的输出公式，即（输入值-输入最小值）*（输出最大值-输出最小值）/（输入最大值-输入最小值）+输出最小值，去推导我们所需的参数数值。

然后我们再使用一个 Branch 节点，和一个 Boolean 类型的参数，去控制是否应用菲涅尔效果。

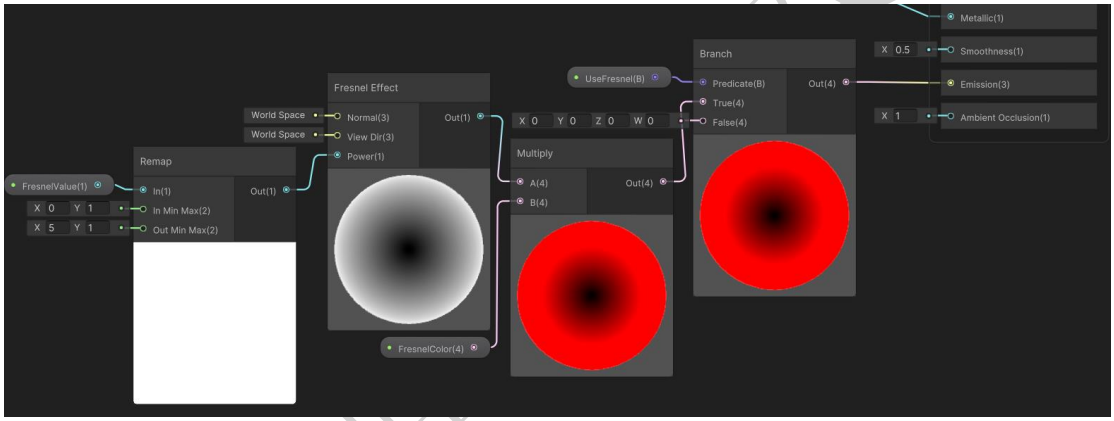


图30. 将 RresnelValue 进行重映射

二十四、 3D 全息投影

1. 创建 Lit ShaderGraph

处于 3D 场景中的真实对象，通常是会受到场景中光源的影响的。如果是 NPR 渲染，而不是真实感渲染，那么其光照效果就需要我们自己去自定义了。

要使我们创建出的 Shader 实例，能够被场景的光源所影响。我们就需要创建 Lit Shader Graph，而不是 Unlit Shader Graph，Unlit Shader Graph 是表示不受光照影响。

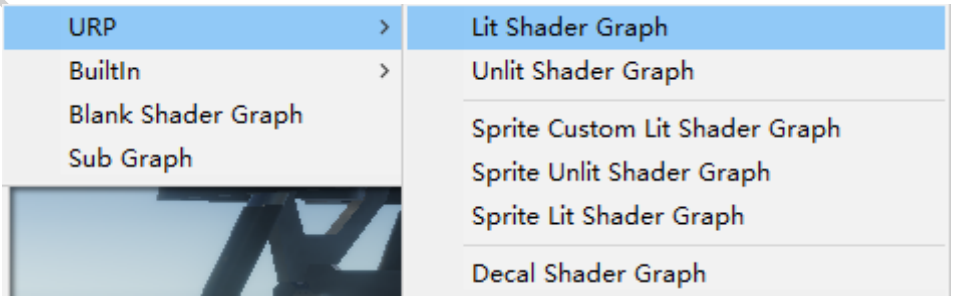


图31. 创建 LitShaderGraph

因为这个效果会涉及到透明度，因此需要将 Shader Graph 的 Surface Type，从不透明

[22] n. 排放物，散发物；（尤指光、热、气等的）散发，排放；射精，遗精；自发光

Opaque 修改为 Transparent 透明。

2. 添加 PBR 纹理

首先在 ShaderGraph 的参数面板中添加 5 个 Texture2D 参数，将其分别命名为 MainTex、Normal、Metallic、Emission 和 AmbientOcclusion。

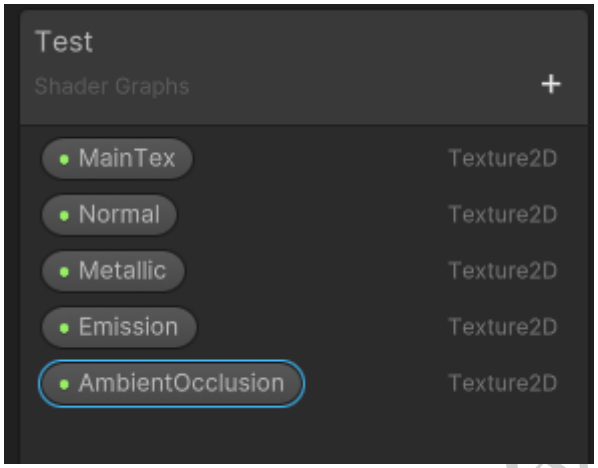


图32. 3D 全息效果的纹理参数

它们分别对应主纹理、法线、金属度纹理、自发光纹理和环境光遮蔽纹理，要使用这些纹理还需要创建 Sample Texture2D 节点，并将其结果输出到片元着色器对应的端口。法线纹理的 Sample Texture2D 节点，其 Type 需要修改为 Normal，否则法线效果将无法正确应用。

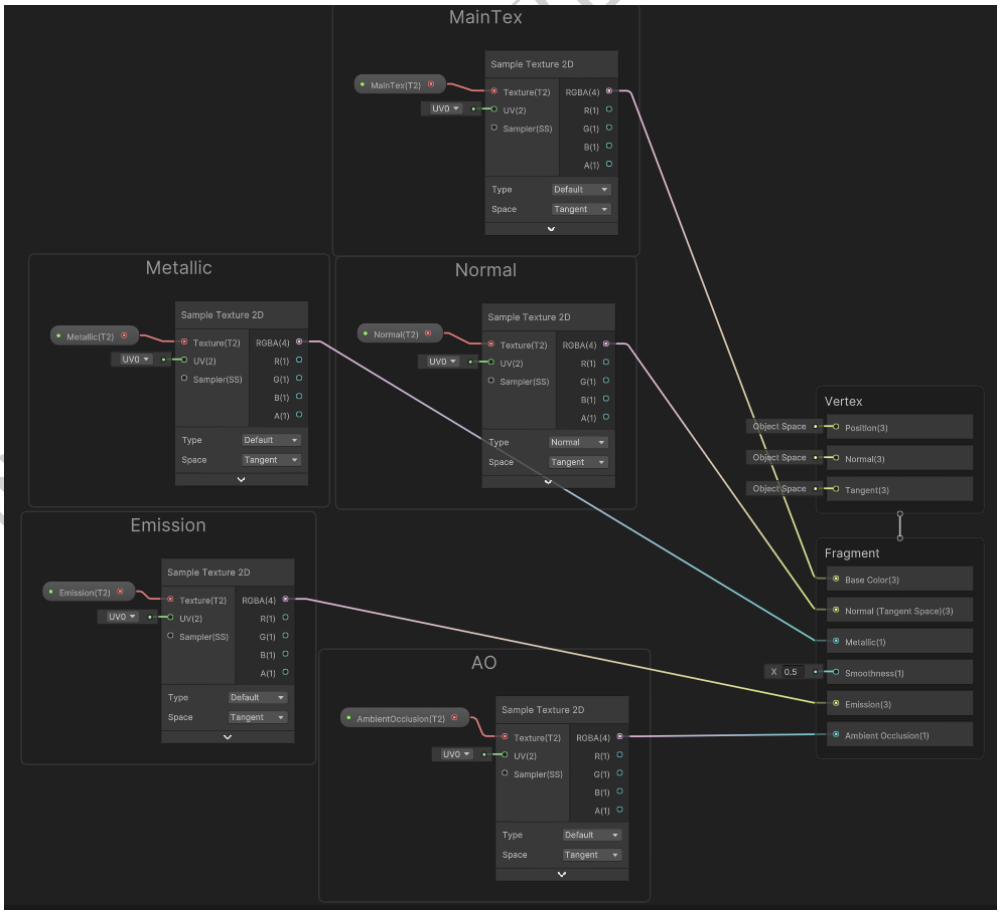


图33. 3D 全息效果纹理输出到对应的端口

3. 修改自发光颜色

当我们连接完成以后，就可以回到 Unity 中创建一个 ShaderGraph 的实例，并将其赋予游戏对象查看效果。为了使画面效果更好看，这里使用后处理 Bloom 效果，去增强自发光效果的表现。但是，我们现在的自发光贴图只有一种颜色，我们可以在 ShaderGraph 中再为其添加一个 HDR 的 Color 参数，在其输出前使用乘法节点修改自发光纹理的颜色。

4. 使用正切函数生成全息纹理

1） 添加参数

我们知道正切函数是一个奇函数，奇函数的图象关于原点中心对称的，且正切函数的周期是 π，在 $\frac{\pi}{2}+k\pi$ 处的值是无穷，也可以说是不存在。利用正切函数的特性，我们就可以用其来生成全息效果的纹理。

首先我们需要在 ShaderGraph 的参数面板中，新增三个 Float 类型的参数，它们分别用于控制全息纹理的密度、移动的速度和流动的方向。

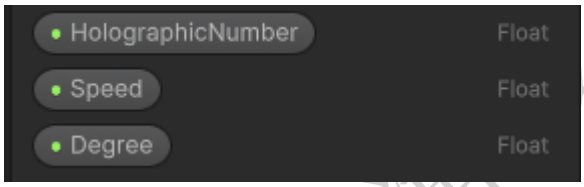


图34. 3D 全息纹理的参数

在参数面板中，我们也可以根据需要去设置 Float 参数的模式。这里我将流动的方向设置为了 Slider，这样在外面就可以使用滑动条去控制它的值了。

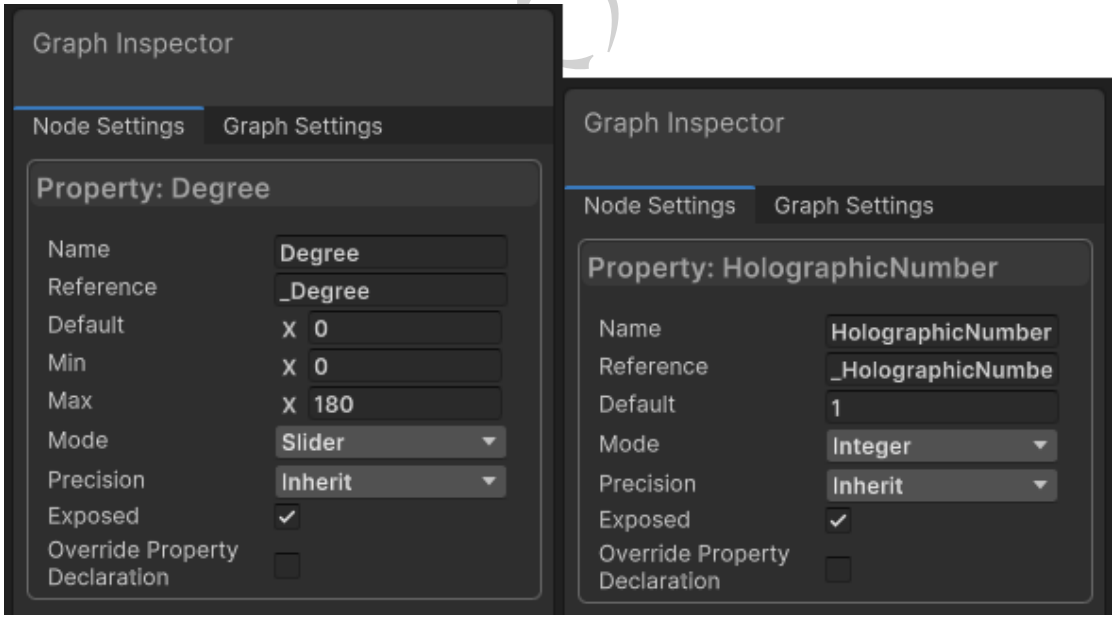


图35. 根据需要设置 3D 全息效果参数的模式

2） 生成全息纹理

创建一个 Position 节点，将它的空间设置为绝对世界空间(Absolute World)，接着使用 Split 节点分离出 R 通道。当然，这里也可以使用屏幕空间。

再创建两个乘法节点和一个正切函数节点(Tangent)，利用之前创建的参数乘以正切函数的周期，然后再乘上我们所分离出的 R 通道值，就可以得到具有完整正切函数周期的 R 通道数值。将这个值输入到正切函数节点(Tangent)，就可以看见当前密度的全息贴图预览

图像了。

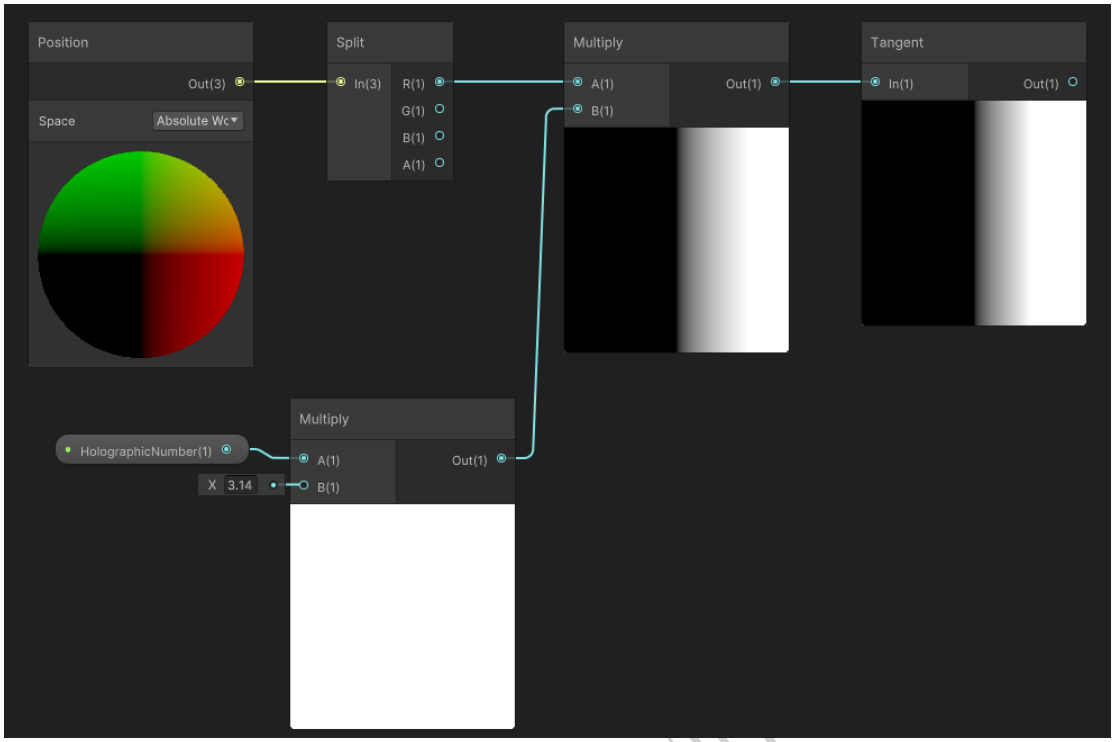


图36.　　利用正切函数得到 3D 全息效果的纹理

因为正切函数在$\frac{\pi}{2}+k\pi$处的值是无穷，因此我们需要使用 Saturate 节点去限制这部分超出范围的值，否则最终效果就可能出现问题。限制完成后，使用加法节点将自发光和全息纹理相加。

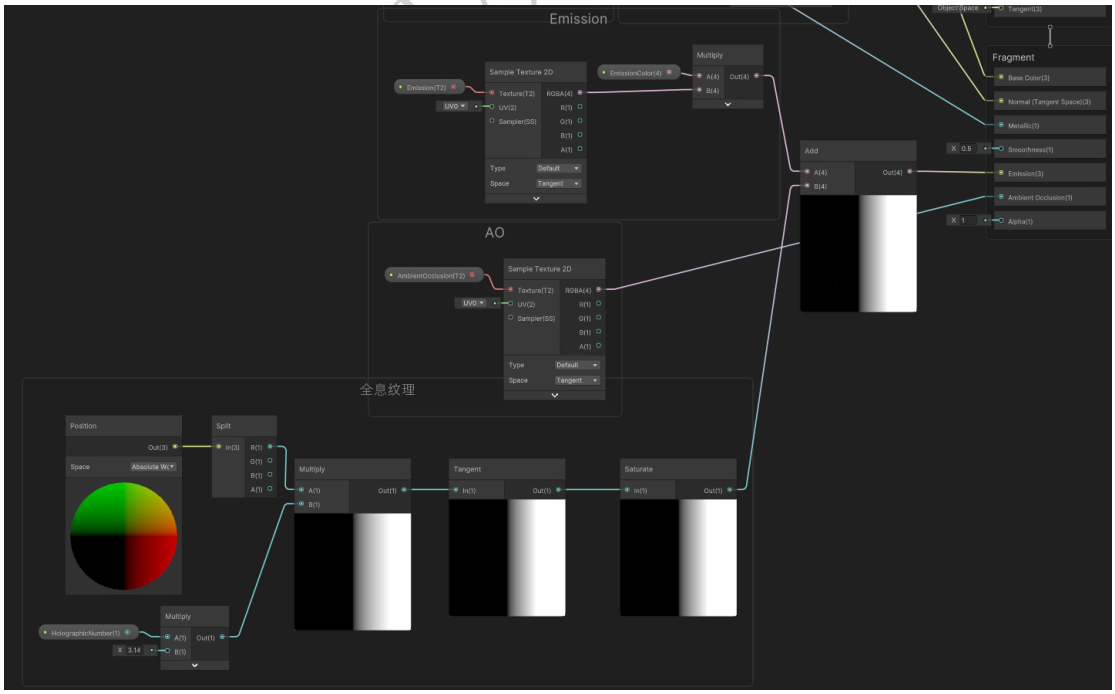


图37.　　将生成的全息纹理和自发光贴图相加

3） 移动全息纹理

有了全息纹理以后，就可以使用 Time 节点让全息纹理流动起来了。为了限制流动的

速度不至于太快，我们不能直接使用之前设置的速度参数，需要额外需要使用一个乘法节点去减低其数值。

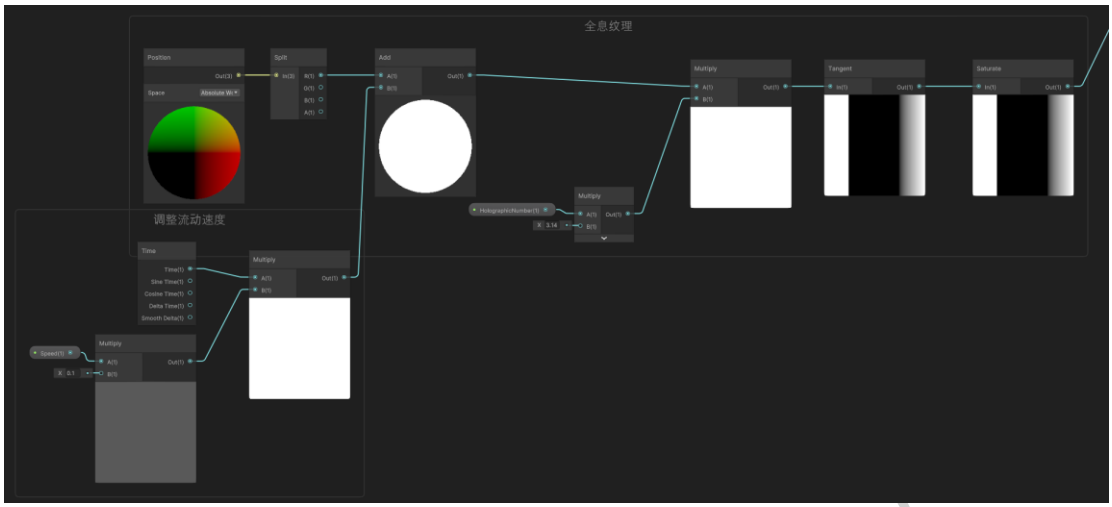


图38.　　调整全息纹理的流动速度

4） 调节全息纹理的流动方向

现在全息纹理已经可以进行流动了，下面我们就可以使用 Rotate 节点去旋转 UV 坐标。Rotate 节点有两种模式，以弧度为单位进行旋转和以角度为单位进行旋转，这里我使用角度。

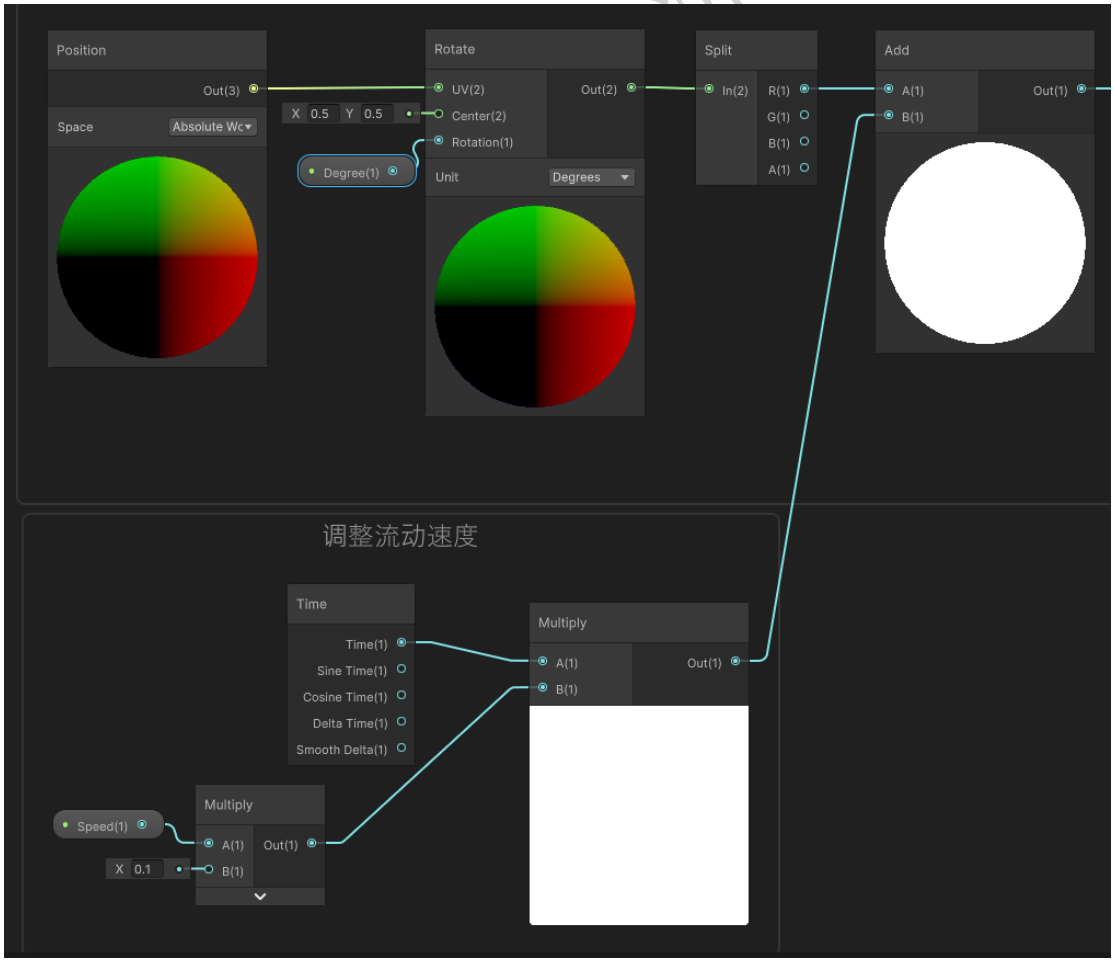


图39.　　使用 Rotate 节点旋转全息纹理

注意，Rotate 节点需要在 Split 节点之前应用，否则结果会出现问题。

5） 以对象的中心点为世界坐标的原点构建新的坐标系

我们现在回到 Unity 中，就已经可以看到全息纹理的效果被应用了，但我们拖动对象就会出现问题。当我们拖动游戏对象时，我们的全息纹理会根据对象的移动而进行改变，这是我们所不想看到的。

造成这一现象的原因就是我们使用的坐标空间，可以将绝对世界空间转换成以对象中心点为原点的一个相对世界空间。为此，我们需要将先模型的中心点转换到绝对世界空间，这需要使用到 Transform 节点。转换完成以后，我们可以将原本的绝对世界空间位置坐标移动到新的原点位置。

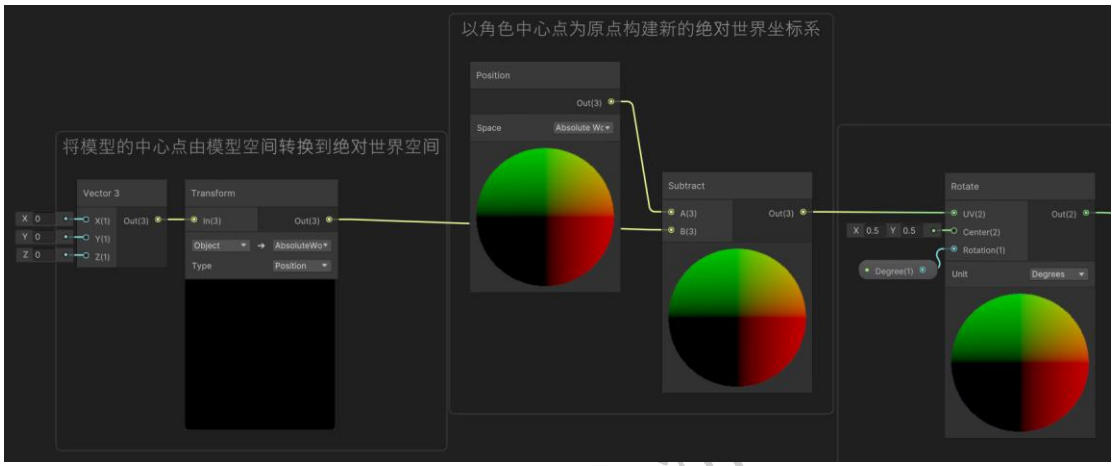


图40. 将模型的中心点由模型空间转换到绝对世界空间，以角色中心点为原点构建新的绝对世界坐标系

完成后再回到 Unity 中拖动游戏对象，就不会再出现之前全息纹理发生变化的问题了。

5. 添加菲涅尔效果

有了全息效果后，我们可以考虑再为其添加一个边缘的自发光效果，也就是使用菲涅尔效果来增强视觉效果。菲涅尔效果我们可以直接使用 ShderGraph 的菲涅尔节点得到，也可以自己去将其实现出来，这里我选择自己将其实现。

1） 添加菲涅尔所需参数

添加四个 Float 类型的参数和两个 HDR Color 类型的参数。Color 参数将分别控制菲涅尔效果内部和外部的颜色，Float 类型的参数分别对应边缘的范围、强度以及内部的强度和不透明度。

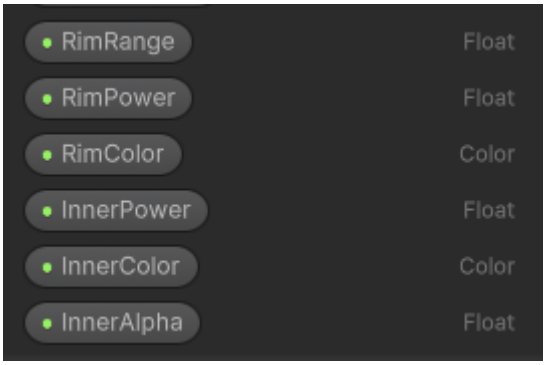


图41. 3D 全息效果的菲涅尔效果参数

2） 生成菲尼尔效果

因为视线与法线的夹角越大反射越强，因此使用法线方向和视口方向的点积，去得到

法线和视线之间的关系。如果法线和视线都是标准化后的向量，那么所得到的点积结果，其范围就在-1 到 1 之间。

我们使用 Saturate 节点过滤掉负值的部分，即视线与法线所成夹角大于 90 度的部分。因为视线与法线的夹角越大反射越强，使用 One Minus 节点去得到正确的菲涅尔效果。

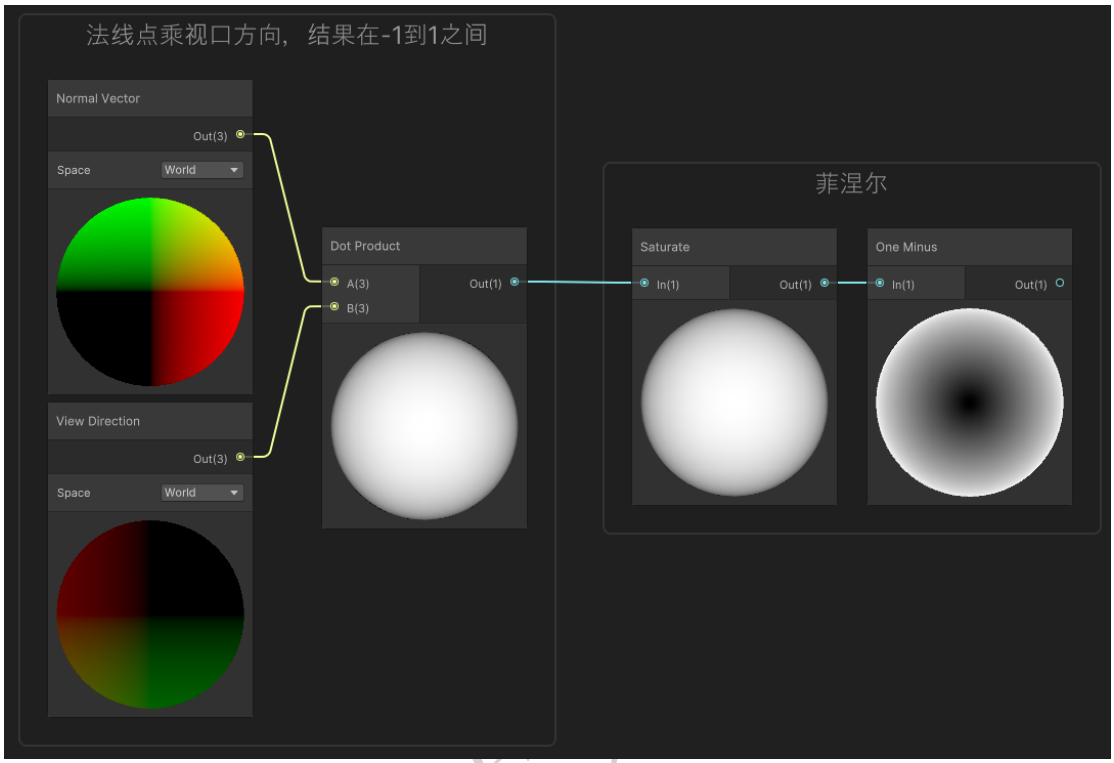


图42.　　3D 全息效果的菲涅尔效果

3） 使用菲涅尔效果作为自发光

有了菲涅尔效果后，我们就可以考虑为它的边缘和内部添加颜色了。创建一个 Lerp 节点，将内部的颜色参数作为 A 端口的输入，边缘的颜色参数作为 B 端口的输入，生成的菲涅尔效果作为 T 端口的输入。得到正确的菲涅尔效果的边缘和内部颜色后，再和生成的全息纹理相乘，最后和原自发光贴图进行组合输出。

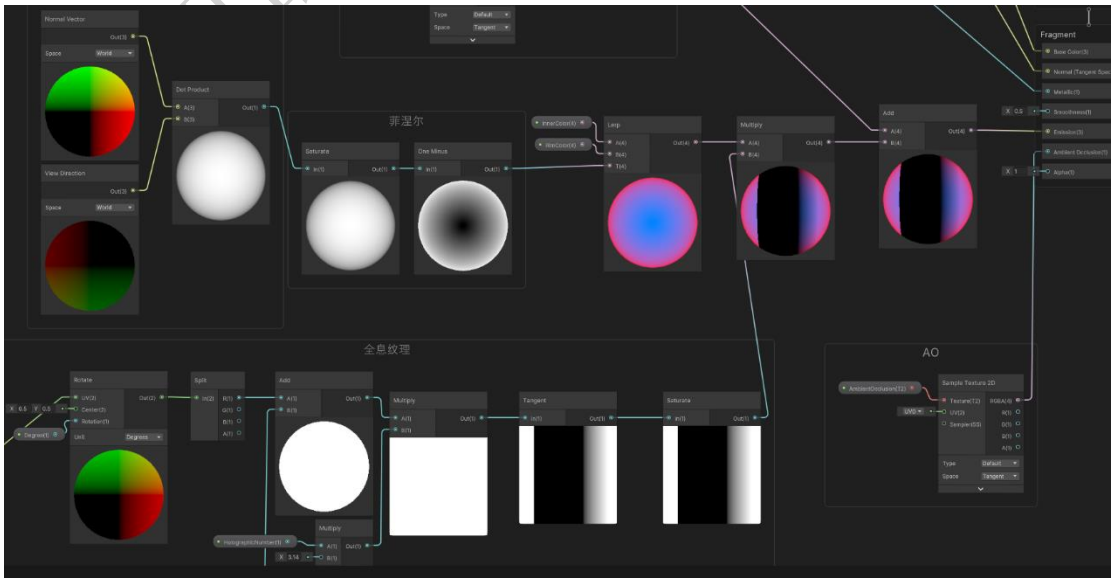


图43.　　为菲涅尔效果的边缘和内部应用各自的颜色

4） 调节菲涅尔效果边缘

很好，现在我们已经为菲涅尔效果的边缘和内部都应用了属于它自己的颜色，下面我们就可以考虑调节菲涅尔效果的边缘范围和强度了。

我们使用 Power 节点配合之前创建的边缘范围参数，去控制菲涅尔效果的边缘范围。边缘范围参数我们将其 Mode 修改为 Slider，并将其最大值设置为 1，最小值设置为 0。

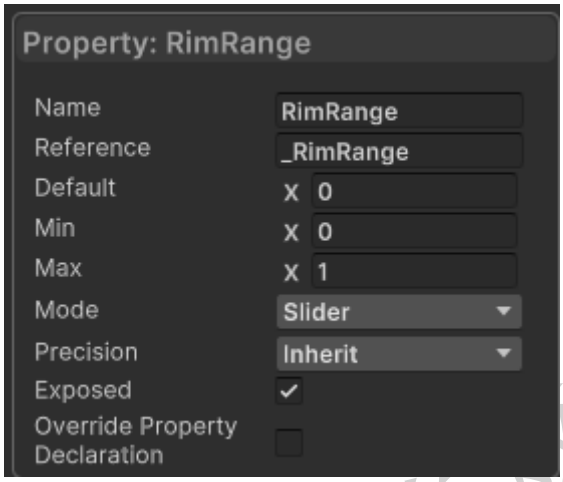


图44.　　3D 全息效果的 RimRange 参数 Mode

但是，这样设置的话，幂函数的指数部分值就很小了，不利于调节范围。因此，我们就可以考虑使用 Remap 节点，去重映射值的范围。我们需要将 Remap 节点输入的最小值设置为 0，输出的最小值设置为幂函数的最大指数值，输出的最大值设置为 1。

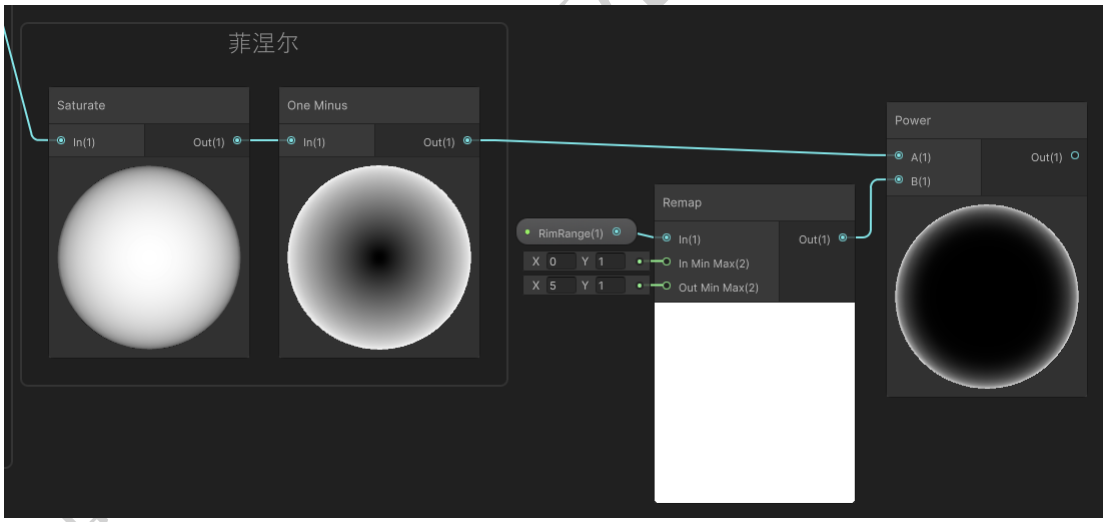


图45.　　使用幂节点调节菲涅尔效果的边缘范围

调节了范围以后，我们就可以使用乘法节点去调节整体的强度了，用于调节整体强度的参数依旧设置为 Slider，其值可以根据自己需要去设置。

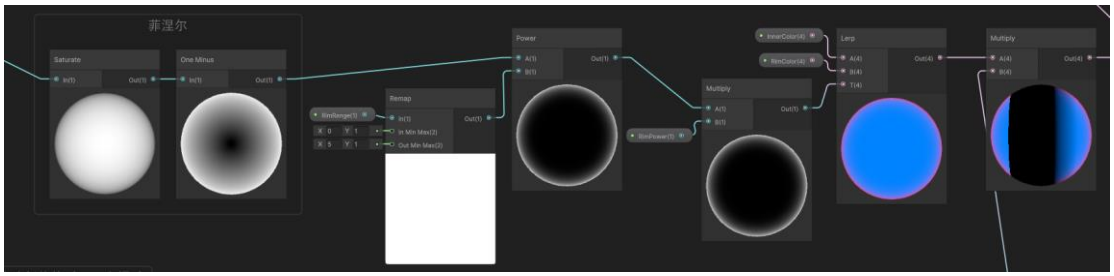


图46.　　调节 3D 全息效果的菲涅尔边缘强度

5） 调节菲涅尔效果内部

前面调节了边缘的范围和整体强度，下面我们就可以再修改一下菲涅尔效果内部的强度。修改内部的强度，我们只需要使用加法节点为整个菲涅尔效果加上一个整体强度，然后再使用 Saturate 节点，保证整体的强度不会超出 1 即可。

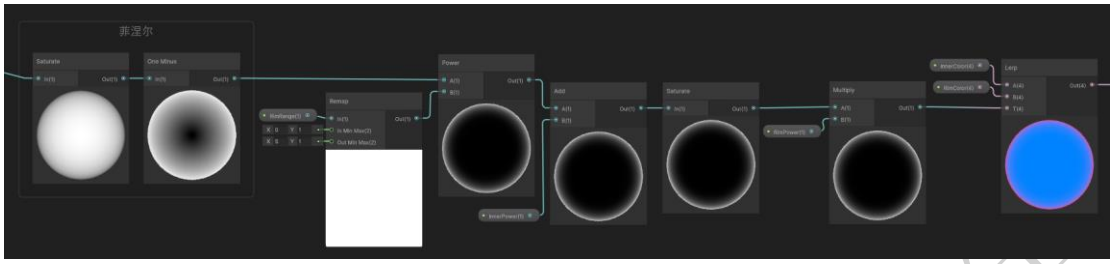


图47.　　调节 3D 全息效果菲涅尔内部强度

这里调节内部强度的参数使用 Slider Mode，这里建议不要将它的最大值设置的过大，否则不利于后续的效果调节。

6） 利用调节后的菲涅尔效果调节全息纹理范围

现在回到 Unity 中查看效果，我们就会发现全息纹理对效果的影响太大了，我们希望它能依据菲涅尔效果去修改自身的强度。因此，我们需要将全息纹理和调节后的菲涅尔效果做乘法，得到与当前菲涅尔效果相关的全息纹理。

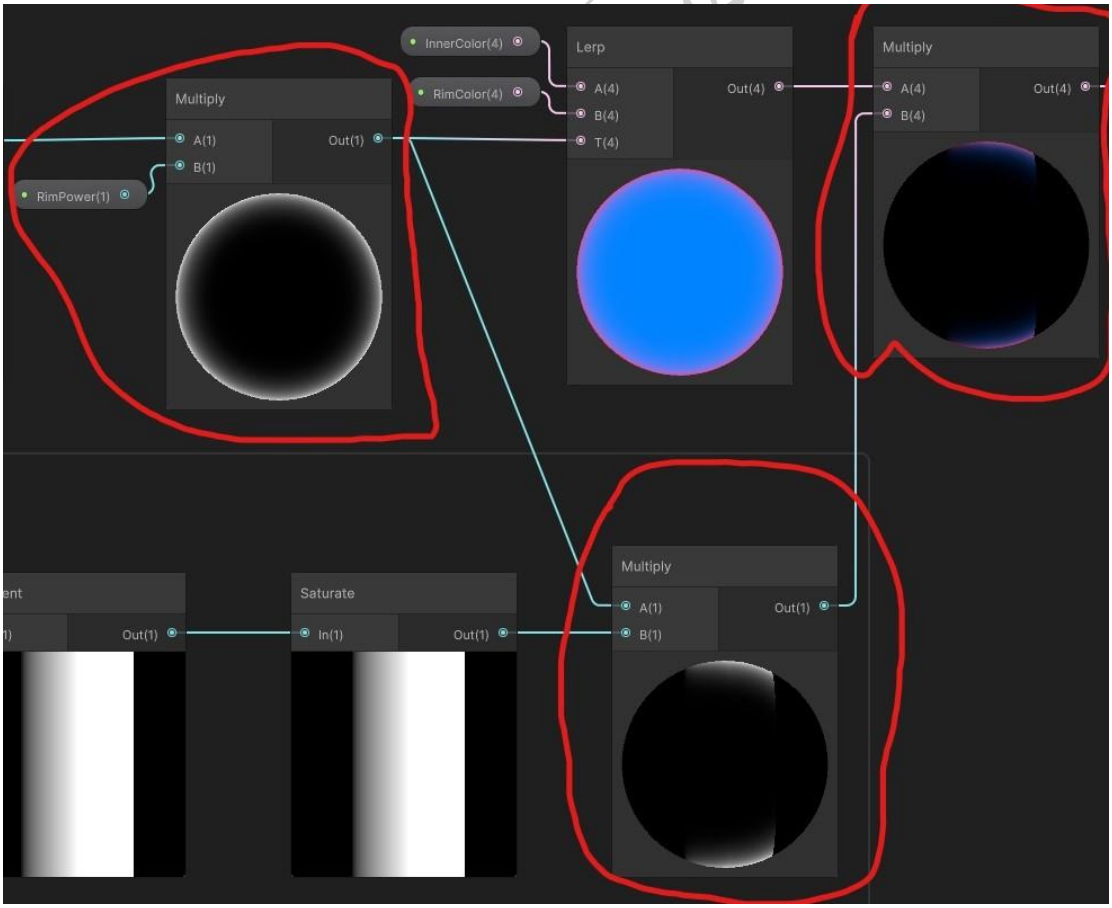


图48.　　对全息效果纹理应用菲涅尔效果

6. 将菲涅尔效果用于 Alpha

到目前为止我们还没有连接 Alpha 通道，我们可以直接将菲涅尔效果输出到我们的 Alpha 通道上。但是，在传递之前我依然对它进行一些调整，方便视觉效果的调整。

我们知道，Alpha 端口让越接近白色的区域不透明度接近 1，越接近黑色的区域不透明度越接近 0。这可能不方便我们查看模型的基础纹理，因此我们可以考虑在输出前，直接增加黑色区域的不透明度，使本不应该显示出来的区域那个呈现在我们面前。

我们使用加法节点和 Saturate 节点配合 Float 参数，去调整不透明度低于 1 的区域，使其能够尽可能显示出来。使用 Saturate 节点，可以保证我们输入到 Alpha 端口的最大值不会超出 1。

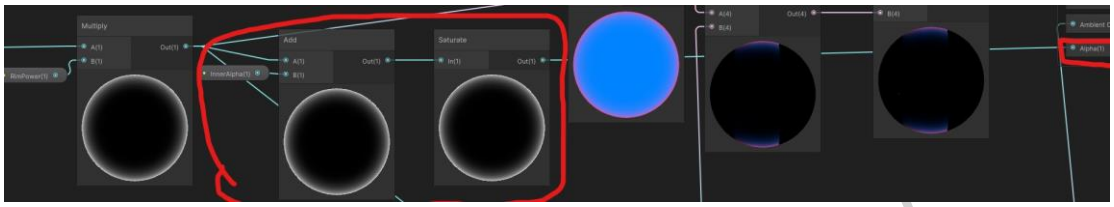


图49. 将调整后的菲涅尔效果输出到 Alpha 端口前做一些调整

这里使用的 Float 参数依旧将其 Mode 设置为了 Slider，并将其最大值和最小值分别设置为了 1 和 0。

二十五、 3D 法线膨胀描边效果

二十六、 后处理描边

https://zhuanlan.zhihu.com/p/50450545

二十七、 3D 积雪效果

二十八、 3D 广告牌效果

二十九、 (Billboard)

三十、 光学迷彩

三十一、 置灰效果

在游戏中，我们经常会看到 UI 的置灰效果，我们可以让美术在 PS 中生成一张置灰效果的图像，但是这就会在游戏实际运行中占用一定的存储空间。如果有大量的 UI 需要置灰，那么其所占的存储空间就会非常多，这不是一个好的解决办法。

哔哩哔哩次元游戏

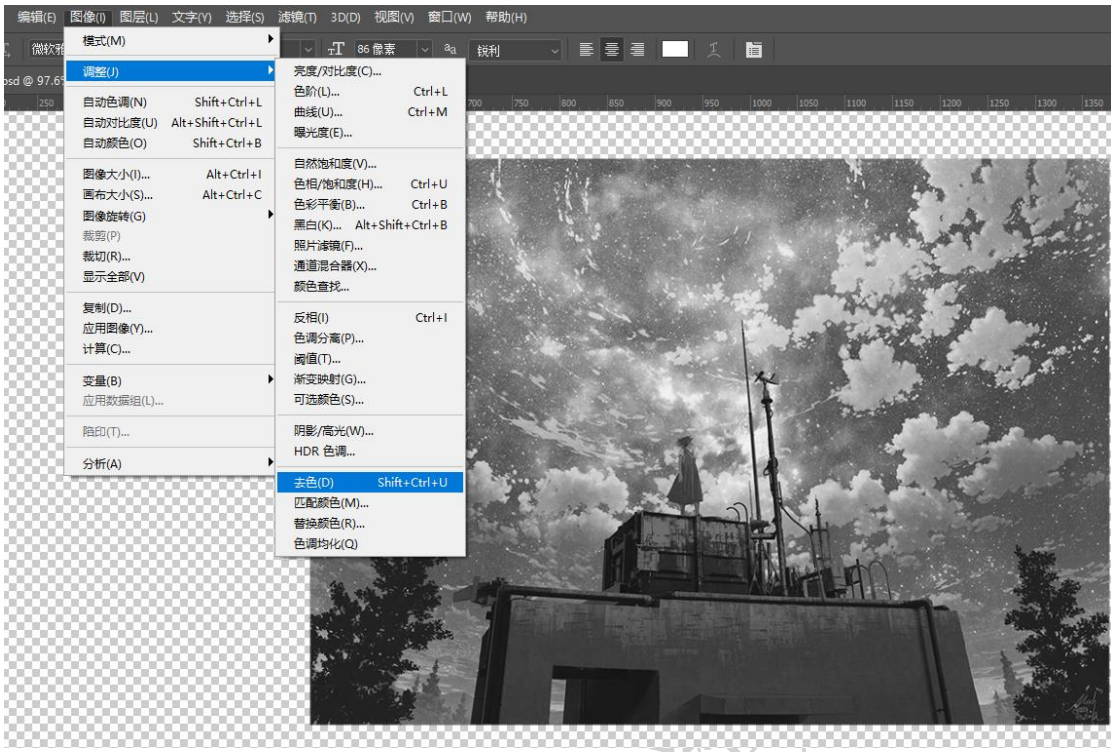


图50. PhotoShop 中对图像进行去色

我们可以创建一个 Shader，去做全屏的后处理以达到全屏置灰的效果，也可以对单一图像使用，来实现单一纹理的置灰。

1. 创建 Shader Graph

点击鼠标右键，创建一个 Sprite Unlit Shader Graph。

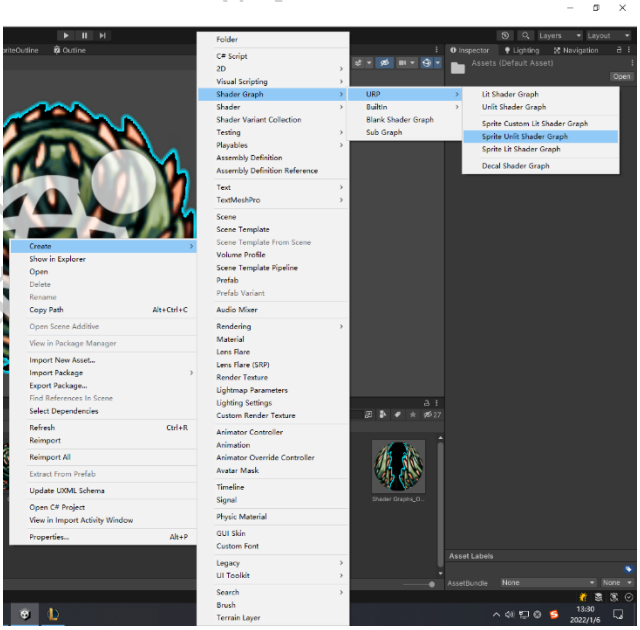


图51. 创建 Sprite Unlit Shader Graph

如果创建的不是 Sprite Unlit Shader Graph 而是 Unlit Shader Graph，我们则需要在 Shader Graph 的设置界面对其进行一个调整。

找到 Graph Inspector 界面的 Universal[23],如果在 Shader Graph 中没有看到 Graph Inspector 界面，我们就需要检查在右上角是否开启了 Graph Inspector。

图52.　　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

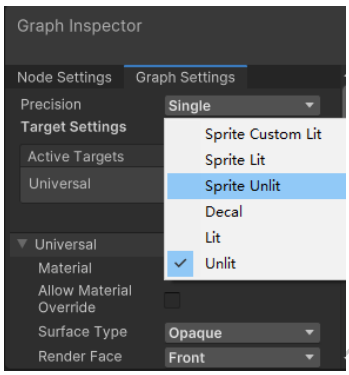


图53.　　修改 Material

2. 方法一——取 RGB 的平均值

置灰效果最直接的方法就是求 R G B 通道的平均值。这是因为，光学三原色(RGB)红绿蓝进行相加为白色，它属于无色系（黑白灰）中的一种。而灰色的数值处于黑色和白色之间，因此三通道相加后的数值取平均值，就可以得到一个灰色。使用平均法得到的灰色，会略暗于 PS 的去色效果，这主要是因为 RGB 三通道颜色的亮度权重不同导致的。

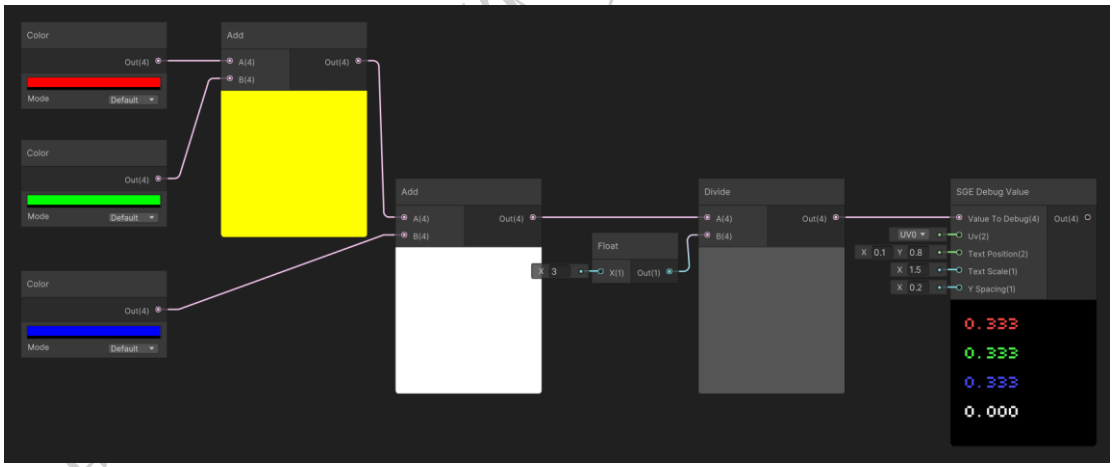


图54.　　对三通道做加法后取平均值

3. 方法二——RGB 通道值乘对应的权重

人眼对 RGB 三原色的敏感程度是不同的，对于绿色的敏感度最高，对红色的敏感度次之，对蓝色的敏感度最低。因此，可以直接对 RGB 设置不同的权重，来达到灰度显示的效果，比较常用的 RGB 权重值为 R：0.299, G：0.587,B：0.114。

置灰公式$\text{GrayValue} = R * 0.299 + G * 0.587 + B * 0.114$。

[23] adj. 普遍的；通用的 n. 普遍原则；普遍看法

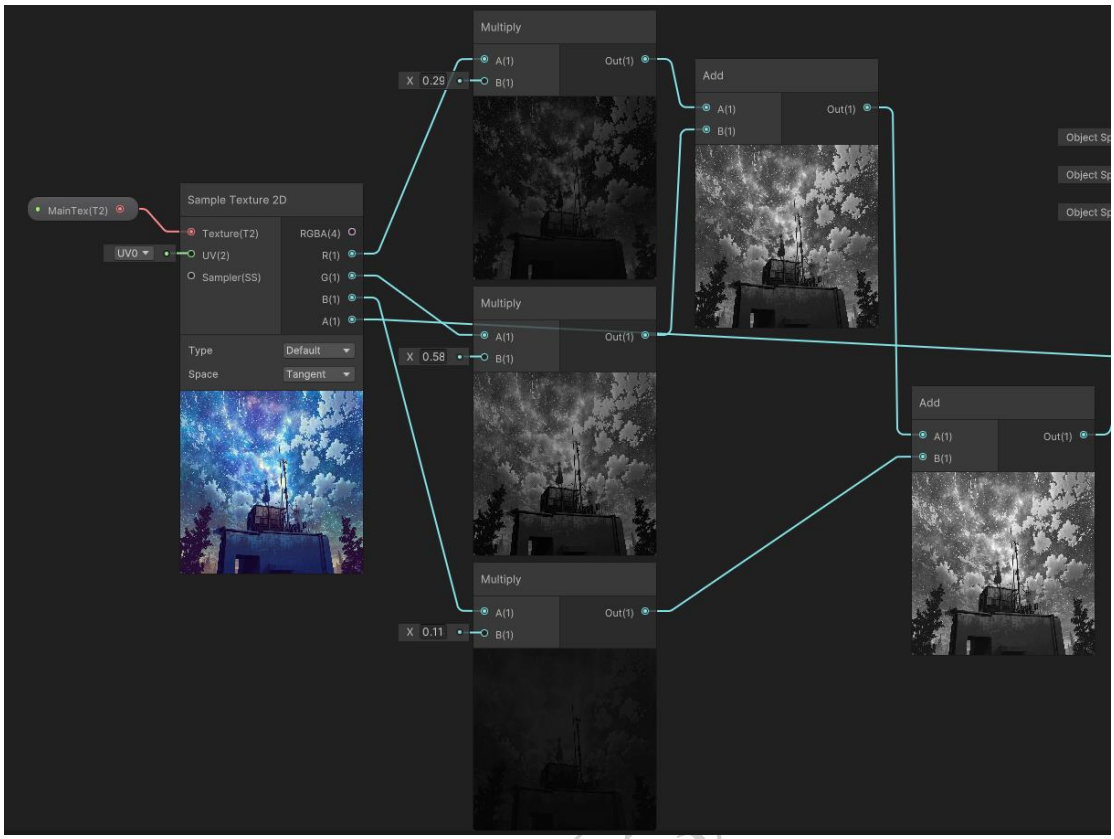


图55.　　对 RGB 三通道应用不同权重得到灰色

注意，这里使用加法得到的结果是正确的。这是 Unity 默认帮我们处理后的结果，因为分离后 RGB 三通道相当于一个一维向量，我们进行加法时都是加在同一个分量上的。但是，片元着色器的 BaseColor 端口以及 ShaderGraph 节点的预览图像，其实是一个四维向量。

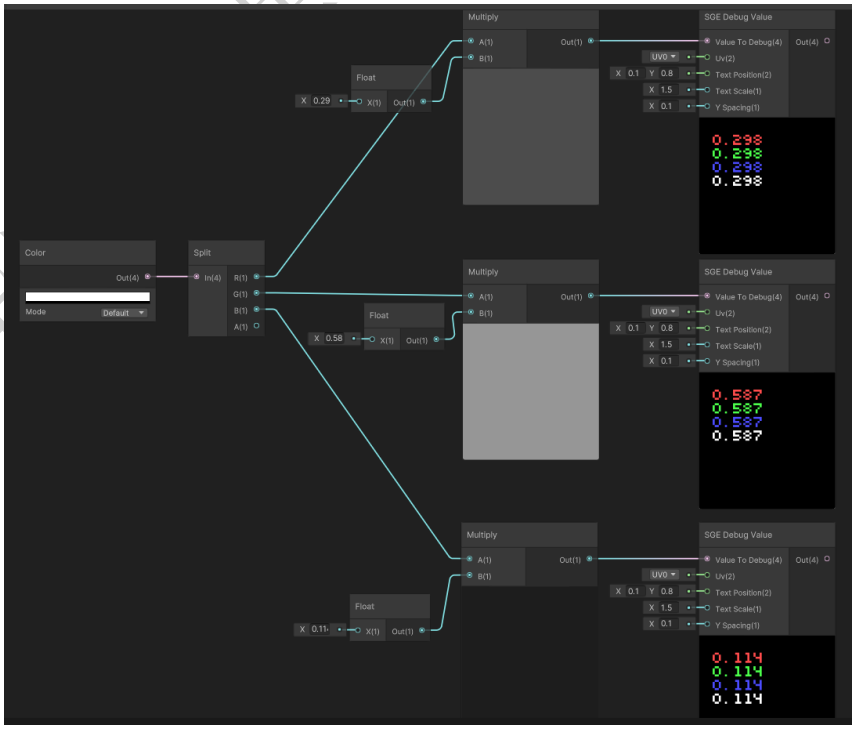


图56.　　Unity 在背后为我们做的处理

Unity 在背后用同样的数值，帮我们补齐了其他的分量，因此我们看见的预览图像和输出是正确的。

如果使用 Combine[24]节点去组合分离出的通道，那么我们是不会得到正确的结果的。

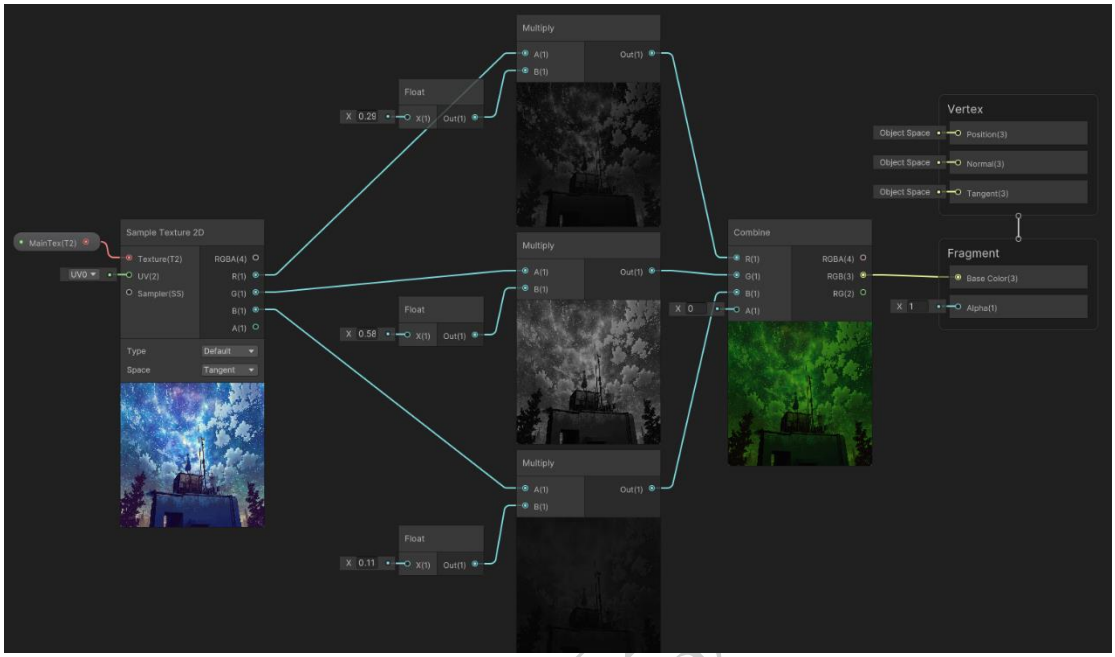


图1. 对计算权重后 RGB 通道使用 Combine 节点

这是因为 RGB 三通道的数值传入 Combine 节点时，它只会取对应的分量，然后再去组合得到一个四维向量。因为取值时是取的一维，所以 Unity 不会帮我们修正。如果要使用 Combine[25]节点得到正确的结果，我们必须将 RGB 三通道的值相加，然后再进行传入。

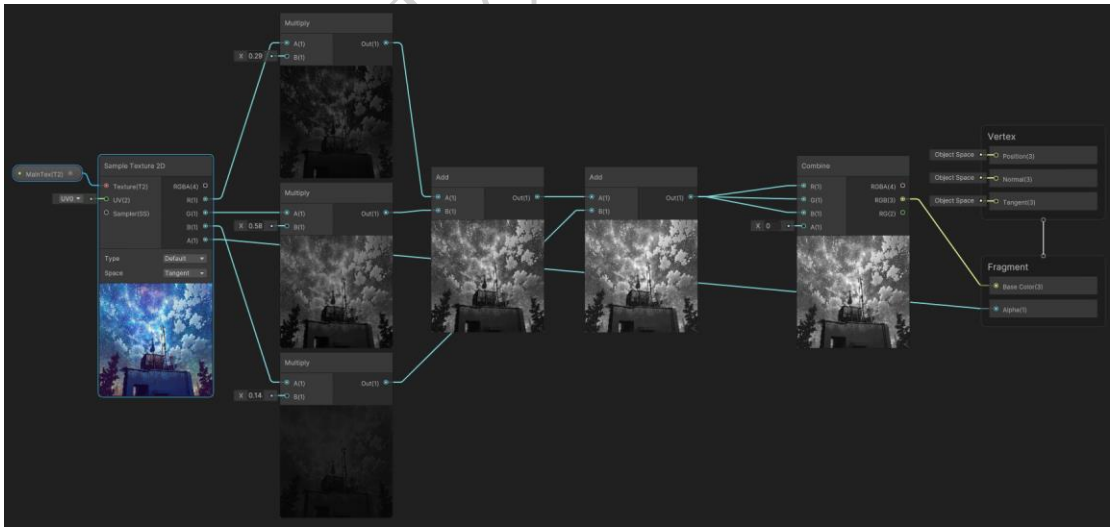


图2. Combine 节点三通道传入同一数值，得到正确的置灰效果

4. 方法三——使用数量积计算

方法三是在方法二的基础上进行改进，即利用数量积公式，隐藏乘法节点的运算和加法节点的运算，使 ShaderGraph 看起来更简洁。

这是点乘的代数表示，设$\vec{\boldsymbol{a}} = (X_1, Y_1, Z_1)$，$\vec{\boldsymbol{b}} = (X_2, Y_2, Z_2)$，则$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = X_1X_2 + Y_1Y_2 + Z_1Z_2$。

[24] v. （使）结合，组合；（使）化合；合并，协力； n.联盟，联合体；联合收割机

[25] v. （使）结合，组合；（使）化合；合并，协力； n.联盟，联合体；联合收割机

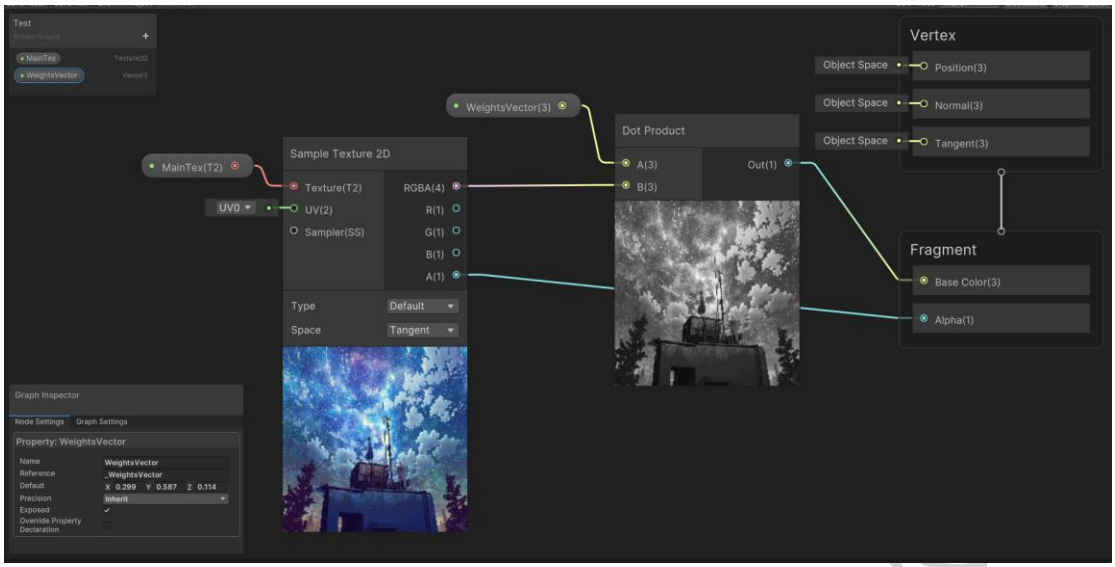


图3. 使用点乘去隐式运算，简化视图

哔哩哔哩次元游戏人

Shader Lab

一、 Shader 概念

1. 什么是 Shader

① GPU 流水线上一些可高度编程的阶段，而由着色器编译出来的最终代码是会在 GPU 上运行的（对于固定渲染管线的渲染来说，着色器有事等同于一些特定的渲染设置）；

② 有一些特定类型的着色器，如顶点着色器、片元着色器等；

③ 依靠着色器我们可以控制流水线中的渲染细节，例如用顶点着色器来进行顶点变换以及传递数据，用片元着色器来进行逐像素的渲染。

Shader 中文名称叫“着色器”,字面意思就是“给物体上色的机器”。它也是通过编写代码实现的，因此本质上也是程序。只不过 Shade 程序跟普通程序不一样，它并不是运将在 CPU 上，而是运行在 GPU 上,Shader 的目的是为了告诉 GPU 如何计算和输出图像。

Shader 所处的阶段只是渲染流水线中的一部分，它主要由顶点着色器(Vertex Shader)和片段着色器(Fragment Shader)组成。

编写 Shader 的语言主要有两种:

(1)基于 Direct3D 的 HLSL,全称 High Level Shading Language，由微软开发,在自家的 Windows 平台上兼容性非常好,因此成为游戏开发的首选。

(2)基于 OpenGL 的 GLSL,全称 OpenGL Shading Language,具有良好的移植性,可以在不同平台使用。

NIVIDIA 希望显卡的程序开发独立于 DX 和 GL 的图形软件库,于是联合微软共同开发了 CG 语言(C for Graphics)。因为它是在 HLSL 的基础上进行开发的，所以其语法跟 HILSL 非常相似。并且，使用 CG 语言编写的 Shader 拥有跨平台性。因此 CG 语言是编写 UnityShader 的首选语言。

2. Shader 和 Material（材质）的关系与区别

Shader 实际上就是一段程序,它负责把输人的顶点数据按照代码里指定的方式进行处理,并对输入的颜色或者贴图等进行计算,然后输出数据。图像绘制单元获取到输出的数据便可将图像绘制出来,最终呈现在屏幕上。

Shader 程序代码，再加上开放的参数设置以及关联的贴图等资源，为实现某种效果而打包存储在一起，最终得到的就是材质(Material)。Material 是 Shader 的实例化资源，一个 Shader 可以实例化为多个材质,并且调节为不同的材质效果。最后把材质指定给某个模型就可以渲染出对应的效果了。

3. Unity 中的 Shader

Unity 一共提供了 6 种 Unity Shader 模板供我们选择——Standard Surface Shader，Unlit Shader，Image Effect Shader，Custom Render Texture，Compute Shader 以及 Ray Tracing Shader。

其中，Standard Surface Shader 会产生一个包含了标准光照模型的表面着色器模板，Unlit Shader 则会产生一个不包含光照（但包含雾效）的基本的顶点/片元着色器，Image Effect Shader 则为我们实现各种屏幕后处理效果提供了一个基本模板。最后，Compute Shader 会产生一种特殊的 Shader 文件，这类 Shader 旨在利用 GPU 的并行性来进行一些与常规渲染流水线无关的计算。

唯唯咩咩游戏人生

二、 简单概念

1. 顶点-Vertices

对象的顶点对应于定义二维或三维空间中表面区域的点集。在 Maya 和 Blender 中，顶点表示为网格和对象的交点。

这些要点有两个主要特点：

① 它们是变换组件的子级。

② 它们具有根据对象总体积的中心定义的位置。

这是什么意思？假设在 Maya 3 中，有两个默认节点与一个对象关联。这些被称为变换和形状。与 Unity 中一样，变换节点定义对象围绕对象轴心的位置、旋转和缩放。相反，形状节点（变换节点的子节点）包含几何属性，即对象的顶点与其体积相关的位置。这意味着我们可以移动、旋转或缩放对象的一组顶点，但同时，我们可以更改特定点或顶点的位置。

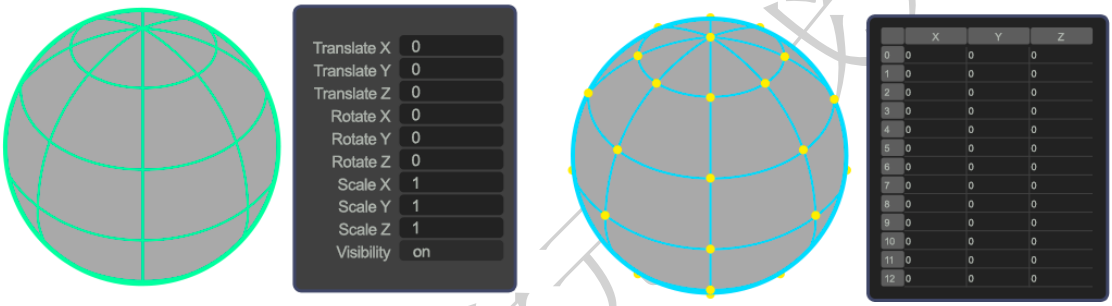


图4. 左边是来自 Maya 的 Transform 节点，右边是 Shape 节点

2. 法线-Normals

曲线的法线是垂直于曲线上一点的切线的直线，曲面上某一点的法线指的是经过这一点并且与该点切平面垂直的那条直线（即向量）。

假设我们有一张白纸，我们请朋友在这张纸的正面画画。如果两边相等，我们如何确定哪个是空白纸的正面？这就是法线存在的原因。法线对应于多边形表面上的垂直向量，用于确定面或顶点的方向。

在 Maya 中，我们可以通过选择属性 vertex normals 来可视化对象的法线。它使我们能够看到顶点在空间中的位置，并确定对象不同面之间的硬度级别。

哔哩哔哩次元成人生

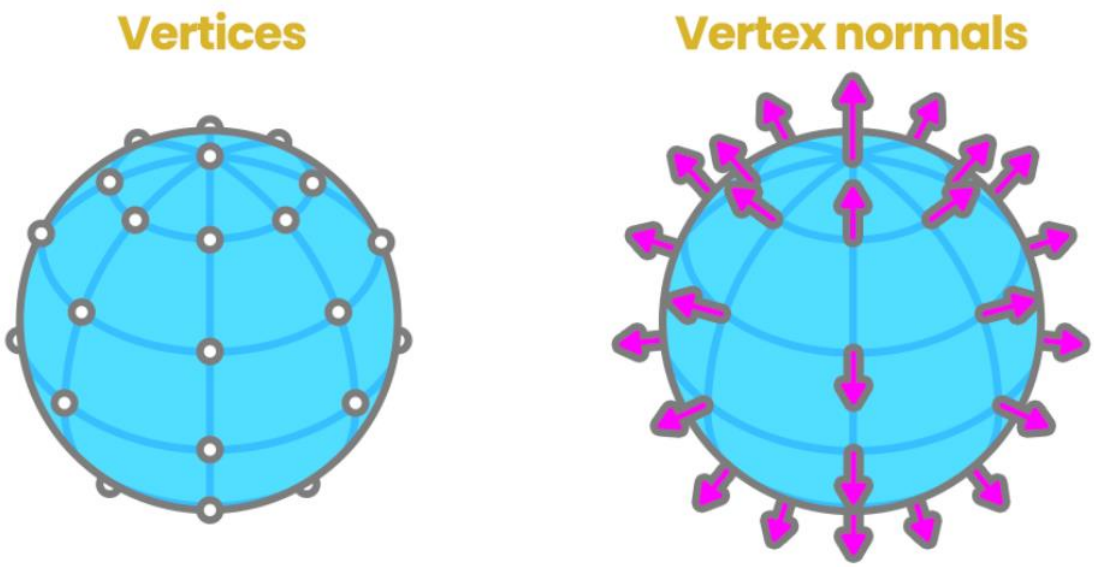


图5. 每个顶点法线的图形表示

3. 切线-Tangents

几何上，切线指的是一条刚好触碰到曲线上某一点的直线，更准确地说，当切线经过曲线上的某点(即切点)时，切线的方向与曲线上该点的方向是相同的，此时，"切线在切点附近的部分"最接近"曲线在切点附近的部分"(无限逼近思想)。tangent 在拉丁语中就是"to touch"的意思。类似的概念也可以推广到平面相切等概念中。

在 Unity 中，切线是沿水平纹理方向跟随网格表面的长度单位的向量，即切线遵循每个几何体面上 UV 的 U 坐标。

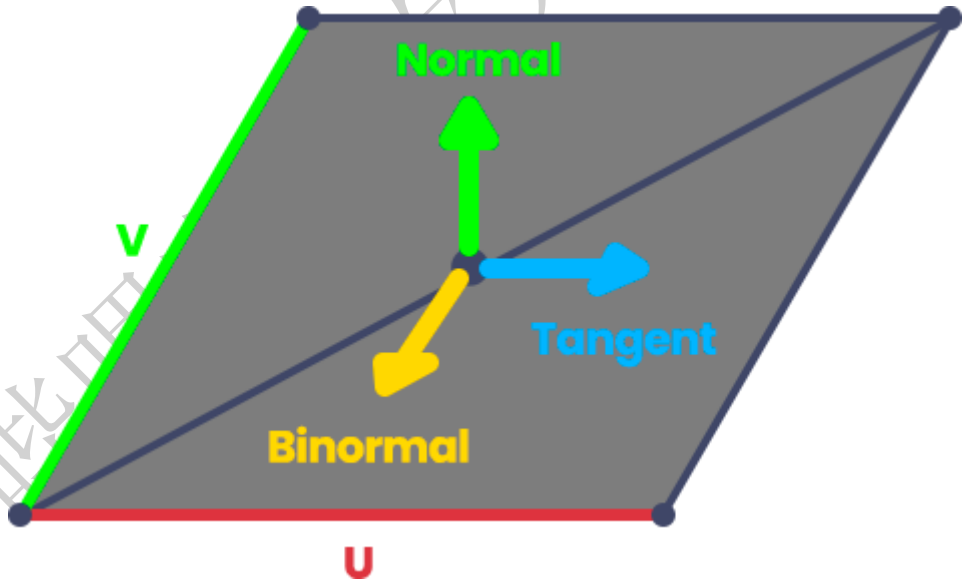


图6. 默认情况下，我们无法在着色器中访问 Binormal，我们必须根据法线和切线计算它。

4. UV 坐标-UV coordinates

UV coordinates 允许我们将二维纹理定位在三维对象的表面上。这些坐标充当参考点，控制纹理图中的哪些纹素对应于网格中的每个顶点。

在 UV 坐标上定位顶点的过程称为“UV 映射(UV mapping)”。这是一个创建、编辑和组织对象网格的平面化二维表示的 UV 的过程。在我们的着色器中，我们可以访问这个属性，或者在我们的 3d 模型上定位纹理，或者在其中保存信息。

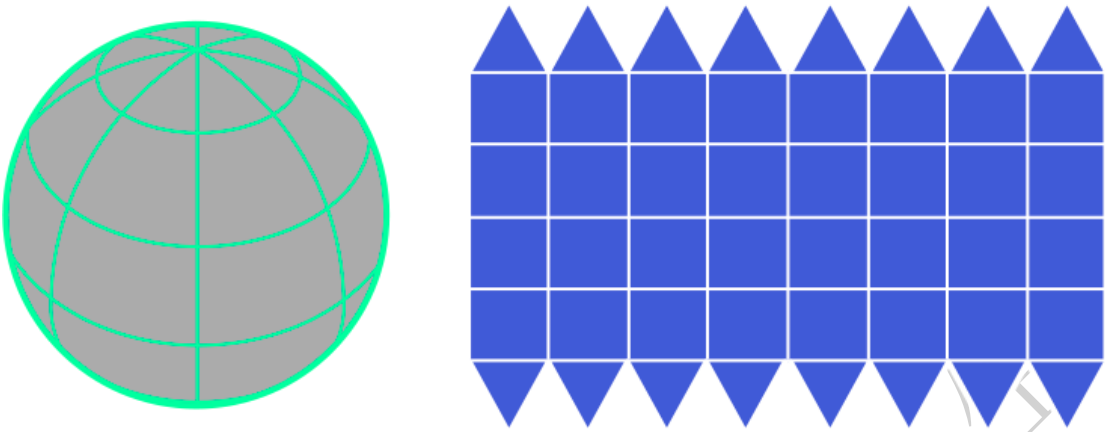


图7. 顶点可以在 UV 贴图中以不同的方式排列

UV 坐标的值在 0.0 和 1.0 之间，其中“零”[0.0, 0.0] 表示起点，“一”[1.0, 1.0] 表示终点。

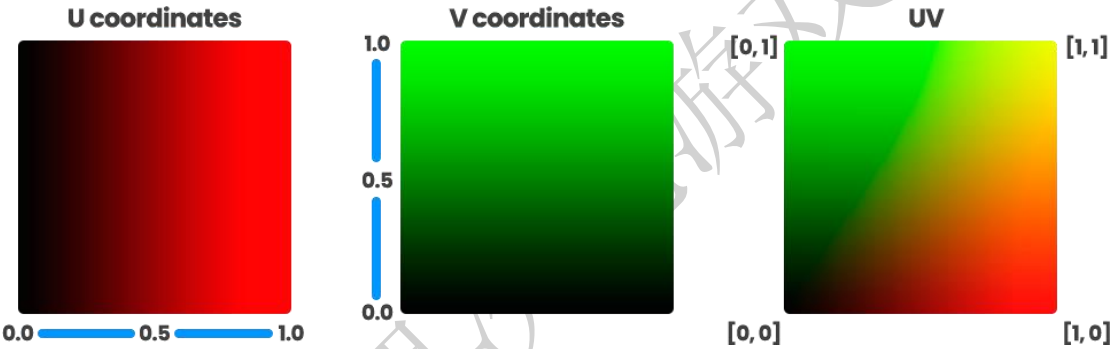


图8. 笛卡尔平面中 UV 坐标的图形参考

三、 纹理

1. 反照率颜色和透明度-Albedo

为 Albedo 指定单一颜色有时很有用，但为 Albedo 属性指定纹理贴图更为常见。这应该代表物体表面的颜色。需要注意的是，Albedo 纹理不应包含任何光照，因为光照将根据对象所带有的其他纹理和光源进行计算。

图9. 典型的 Albedo 纹理贴图的两个示例。

Albedo 颜色的 alpha 值控制材质的透明度，这仅在材质的 Rendering Mode 设置为 Transparency 而不是 Opaque 时才有效。如上所述，选择正确的透明度模式很重要，因为它决定了我们是否仍会看到满值的反射和镜面高光，或者它们是否也会根据透明度值淡出。

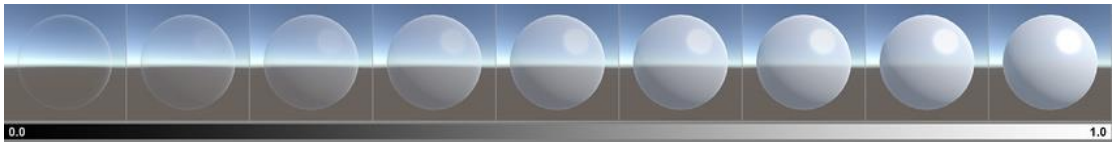


图10.　　透明度值范围从 0 到 1，使用适用于逼真透明对象的透明模式

使用分配给 Albedo 属性的纹理时，我们可以通过确保 albedo 纹理图像具 alpha 通道来控制材质的透明度。Alpha 通道值映射到透明度级别，白色是完全不透明的，黑色是完全透明的。这将使 Material 可以具有不同透明度的区域的效果。

2. 金属纹理-Metallic Texture

材质的金属参数(metallic parameter)决定了其表面的“金属感”。当物体表面具有更多金属性时，它会反射更多的环境光，并且其漫反射纹理变得不那么明显。如果一个物体表面是全金属的，其表面颜色将完全由环境反射驱动。当物体表面的金属性较低时，其漫反射纹理会更加清晰，并且任何表面反射都可见且在表面颜色之上，而不是使其变得模糊不清。

我们一旦为 Metallic 分配纹理后，该材质的金属度就将由 Texture 的红色通道(Red Channel)控制，材质的平滑度由 Texture 的 Alpha 通道控制。（这意味着该 Texture 的绿色和蓝色通道将被忽略）。这张 Texture 可以将区域定义为粗糙或光滑，金属或非金属，这在处理覆盖不同要求的模型区域纹理贴图时非常有用。

3. 法线贴图（凹凸贴图）-Normal Map(Bump[1] Map)

法线贴图是凹凸贴图的一种，它们是一种特殊的纹理，可以让用户为模型添加表面细节，例如凹凸、凹槽和划痕，这些细节可以捕捉光线，就好像它们是由真实的几何图形表示出来的一样。

要显示一个表面，该表面具有凹槽和螺钉或铆钉，就像是飞机的机身。有一种方法可以实现，就是将这些细节建模为几何图形，如下图所示。

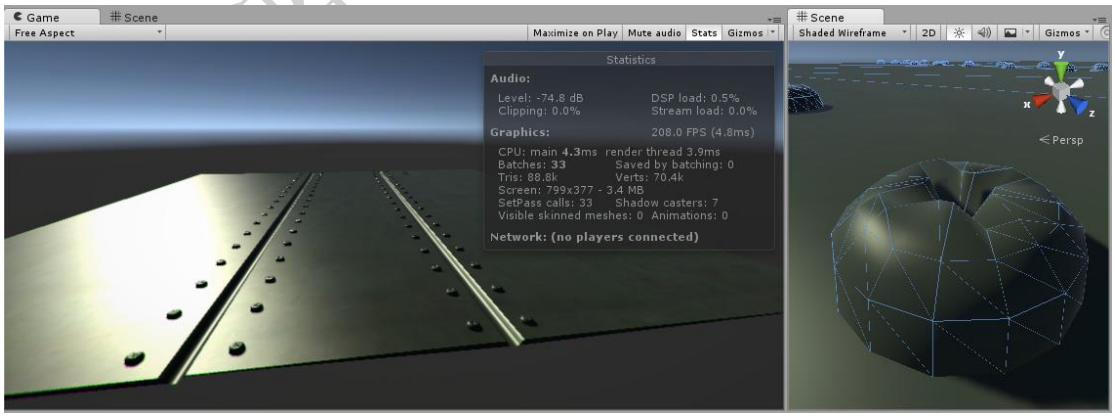


图11.　　一张飞机金属片，其细节建模是真实的几何模型

将如此微小的细节，建模为“真实”几何图形通常不是一个好的办法。 在上图的右侧，可以看到构成单个螺丝头细节所需的多边形。 在具有大量精细表面细节的大型模型上，这将需要绘制大量多边形。 为了避免这种情况，我们应该使用法线贴图去表示精细的表面细节，以及用于模型较大形状的较低分辨率的多边形表面。

[1] v. 碰上，撞上；（无意地）碰，撞；颠簸行进；拖拽；　n. 肿块；隆起，凸块；碰撞，撞击；

如果我们用 Bump[2] map 来表示这些细节，表面几何会变得更简单，并且细节只被表示为一个纹理，它可以调节光线从表面反射的方式。 这是现代图形硬件可以非常快速地完成的事情。 金属表面现在可以是一个低多边形平面，螺丝、铆钉、凹槽和划痕会捕捉到光线，并且由于这张 Texture 而显得有深度。

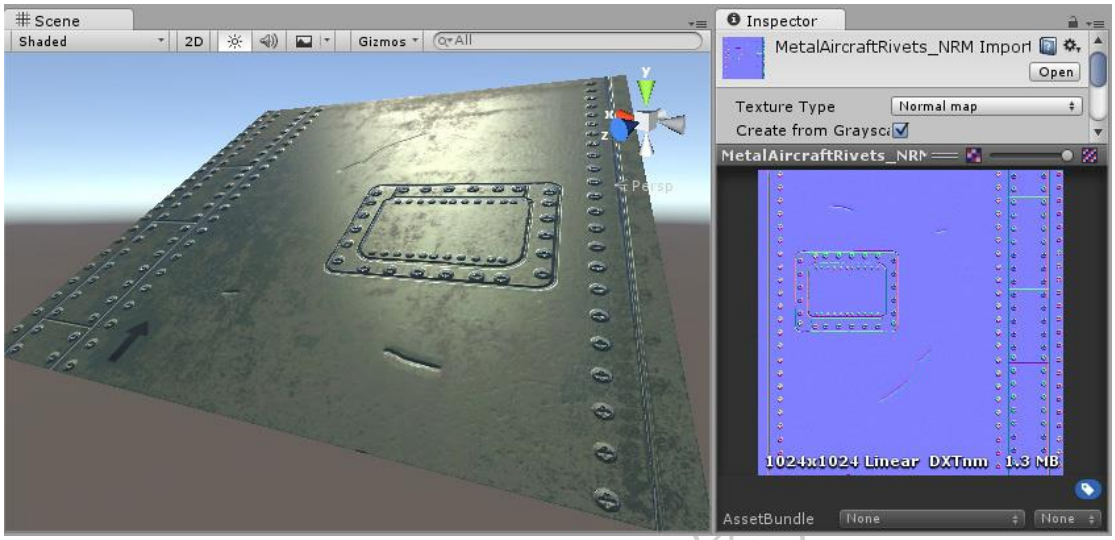


图12. 螺钉、凹槽和划痕在法线贴图中定义，它会修改光线在这个低多边形平面表面的反射方式，给人以真实 3D 细节的感觉

在现代游戏开发艺术流水线中，设计师将使用他们的 3D 建模应用程序来生成基于非常高分辨率源模型的法线贴图。然后将这些 Noraml Map 映射到模型的低分辨率游戏版本，以便使用法线贴图来渲染原始高分辨率细节。

1） 什么是 Noraml Map

Normal Map 通过使用 Texture 来存储有关如何修改模型中的表面法线的信息，使表面法线的这种修改更进一步。法线贴图是映射到模型表面的图像纹理，类似于常规颜色纹理，但是每个像素在 Normal Map 的纹理中（称为 texel）表示表面法线方向与平面（或平滑插值）多边形的“真实”表面法线方向的偏差。

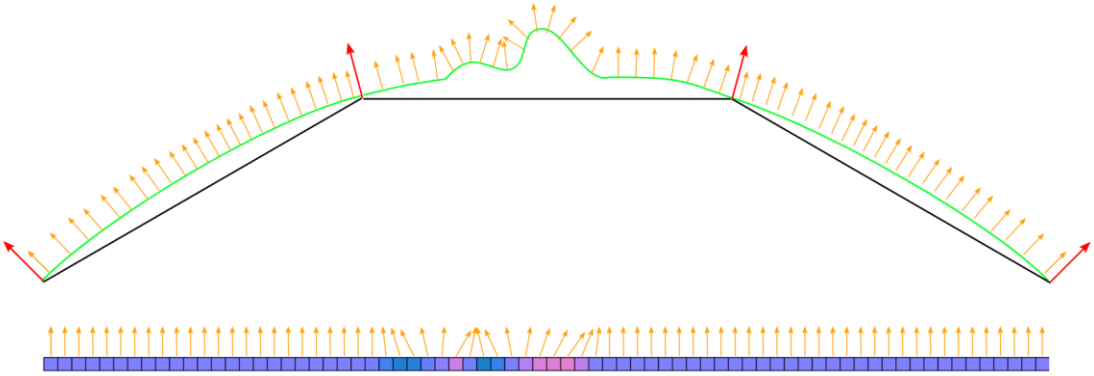


图13. 跨三个多边形的法线映射，被视为一张 2D 图

它还是 3D 模型表面上三个多边形的 2D 表示，每个橙色箭头对应于法线贴图纹理中的一个像素，图像的下面部分是 Normal Map Texture 的单像素切片。在中心，可以看到法线已被修改，在多边形表面上出现了几个凸起。由于光照在表面上出现的方式，使得这些凹凸只会更明显，因为这些修改后的法线将用于光照的计算。

[2] v. 碰上，撞上；（无意地）碰，撞；颠簸行进；拖拽； n. 肿块；隆起，凸块；碰撞，撞击；

原始法线贴图文件中可见的颜色通常具有蓝色色调，并且不包含任何实际的明暗阴影，这是因为颜色本身并不打算按原样显示。相反，每个纹理像素的 RGB 值表示方向向量的 X、Y 和 Z 值，并作为对多边形表面的基本插值平滑法线的修改而应用。

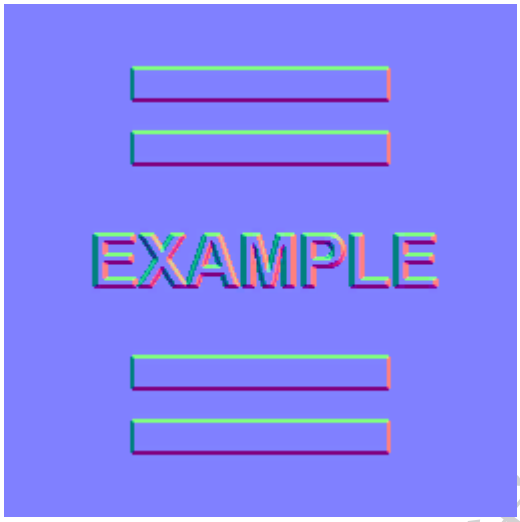


图14.　　一个示例法线贴图纹理

上图是一个简单的 Normal Map，它包含一些凸起区域信息。当我们将 Material 与颜色贴图（Albedo Map）组合并将上图应用于一个圆柱体网格的表面时，结果如下下图所示。

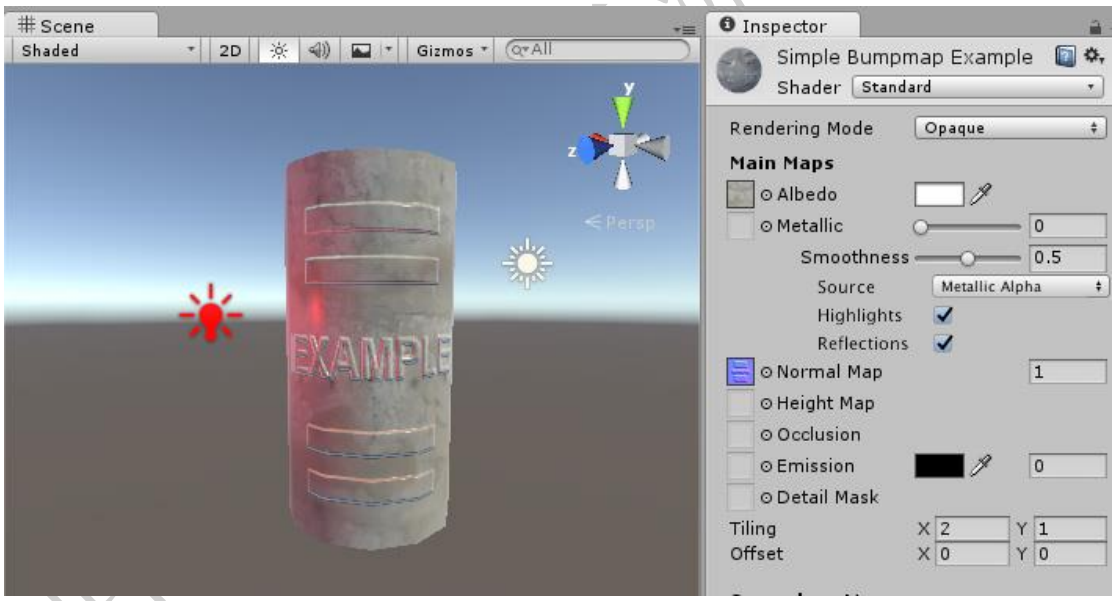


图15.　　应用法线贴图的示例

同样，这不会影响该网格的实际多边形的性质，只会影响其表面照明的计算方式。表面上这种明显的凸起字母和形状并没有真正存在，从掠过的角度观察表面将揭示平坦表面的真实性质，但是从大多数观察角度来看，圆柱体现在似乎具有从表面凸起的浮雕细节。

2） 如何获取或制作 Normal Map

通常，法线贴图是由 3D 或纹理设计师结合他们制作的模型或纹理制作的，它们通常反映 Albedo map 的布局和内容。有时它们是手工制作的，有时它们是从 3D 应用程序中渲染出来的。

从 3D 应用程序中渲染，基本概念是 3D 设计师将制作模型的两个版本，一个包含所有细节为多边形的高分辨率模型，以及一个较低分辨率的模型。高分辨率模型太详细而无法

在游戏中以最佳方式运行（网格中的三角形面片太多），但它在 3D 建模应用程序中用于生成法线贴图。模型的低分辨率版本可以省略现在存储在法线贴图中的非常精细的几何细节，以便可以使用法线贴图来渲染它。一个典型的用例是显示角色服装上的折痕、纽扣、带扣和接缝的凹凸细节。

有一些软件包可以分析常规照片纹理中的光照，并从中提取 Normal Map。其工作原理是假设原始纹理是从一个恒定方向照亮的，并且分析明暗区域并假设它们与有角度的表面相对应。但是，当实际使用 Bump Map 时，您需要确保您的 Albedo 纹理没有来自图像中任何特定方向的光照。理想情况下，Albedo Map 应该代表完全没有光照的表面颜色，因为光照信息将由 Unity 根据光照方向、表面角度和凹凸贴图信息计算得出。

下面是两个例子，一个是简单的重复石墙纹理及其对应的法线贴图，一个是角色的纹理图集及其对应的法线贴图。

图16.　　石墙纹理及其对应的法线贴图纹理

图17.　　角色纹理图集，及其对应的法线贴图纹理图集

3） Bump Map、Normal Map 和 Height Map 有什么区别

Normal Map 和 Height Map 都是 Bump[3] Map 的类型，它们都包含用于表示简单多边形网格表面上明显细节的数据，但它们各自以不同的方式存储这些数据。

[3] v. 碰上，撞上；（无意地）碰，撞；颠簸行进；拖拽；　n. 肿块；隆起，凸块；碰撞，撞击；

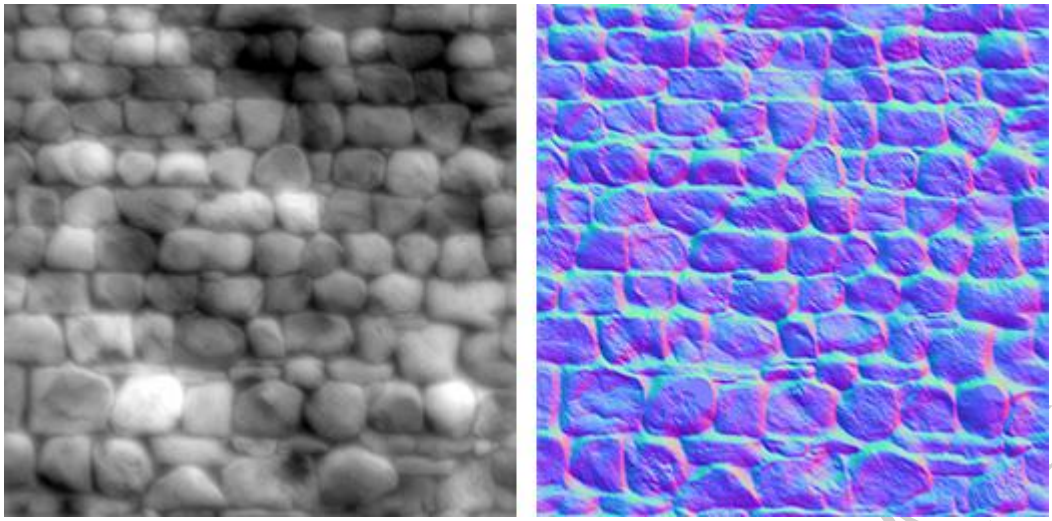


图18.　　左侧是用于凹凸贴图石墙的高度图。右侧是用于凹凸贴图石墙的法线贴图

在上面的左侧，您可以看到用于石墙的 Height Map。Height Map 是一个简单的黑白纹理，其中每个像素代表表面上的点应该出现的凸起量。像素颜色越白，区域看起来就越高。

Normal Map 是一个 RGB 纹理，其中每个像素代表表面应该朝向的方向，相对于其未修改的表面法线的差异。因为向量存储在 RGB 值中的方式，这些 Texture 往往是蓝紫色调。

现代实时 3D 图形硬件依赖于 Normal Map，因为它们包含修改光线在表面反射时的外观所需的向量。Unity 也可以接受 Height Map 用于 Bump[4] Map，但必须在导入时将其转换为法线贴图才能使用它们。

4） 为什么 Normal Map 是蓝紫色的

理解这一点对于使用法线贴图并不重要！可以跳过这一段。

RGB 颜色值用于存储向量的 X、Y、Z 方向，其中 Z 为“向上”（与 Unity 通常将 Y 用作“向上”的约定相反）。此外，纹理中的值被视为减半，增加了 0.5。这允许存储所有方向的向量。

因此，要将 RGB 颜色转换为矢量方向，必须先乘以 2，然后再减去 1。例如，RGB 值 (0.5, 0.5, 1) 或十六进制的#8080FF 会导致向量为 (0,0, 1) ，出于法线映射的目的，它是表示“向上”的，这表示模型表面没有变化。

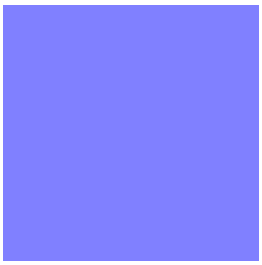


图19.　　仅使用 #8080FF 的法线贴图，它转换为 0, 0, 1 或“向上”的法线向量

(0.43, 0.91, 0.80) 的值给出了 (－0.14, 0.82, 0.6) 的向量，这是对表面相当陡峭的修改。在一些石头边缘顶部的石墙法线贴图的亮青色区域中可以看到这样的颜色。结果是这些边缘以与石头更平坦的表面截然不同的角度捕捉光线。

[4] v. 碰上，撞上；（无意地）碰，撞；颠簸行进；拖拽；　n. 肿块；隆起，凸块；碰撞，撞击；

图20.　　石墙纹理及其对应的法线贴图纹理

这些石头的法线贴图中明亮的青色区域，表示对每块石头顶部边缘的多边形表面法线的修改强度之大，使它们在 Unity 中，可以以正确的角度去捕捉光线。

下面展示了 Bump Map 的一些应用案例。

图21.　　没有凹凸贴图效果的石墙

上图是没有使用任何 Bump Map 的石墙，其岩石的边缘和刻面无法捕捉场景中的定向光源。

图22.　　应用了凹凸贴图的石墙

上图是应用了 Bump Map 的石墙，其面向光源一边石头的边缘，反射定向光源与石头表面和背面的边缘有着巨大的不同。

图23.　　应用了凹凸贴图的石墙，在不同的照明场景中

相同凹凸贴图的石墙，在不同的照明场景中。由上图可以看到，石墙的每个像素都根据光线照射到基础模型（多边形）的角度来点亮，并通过法线贴图中的向量进行了调整。因此，面向光源的像素是明亮的，而背向光源的像素是较暗的，或者处于阴影之中。

5） 如何导入和使用 Normal Map 和 Height Map

像往常一样，可以通过将 Texture 文件放在 assets 文件夹中来导入法线贴图。但是，需要告诉 Unity，这个 Texture 是一个 Noraml Map。可以在 Import Inspector 中将“纹理类型”设置更改为“Normal Map”。

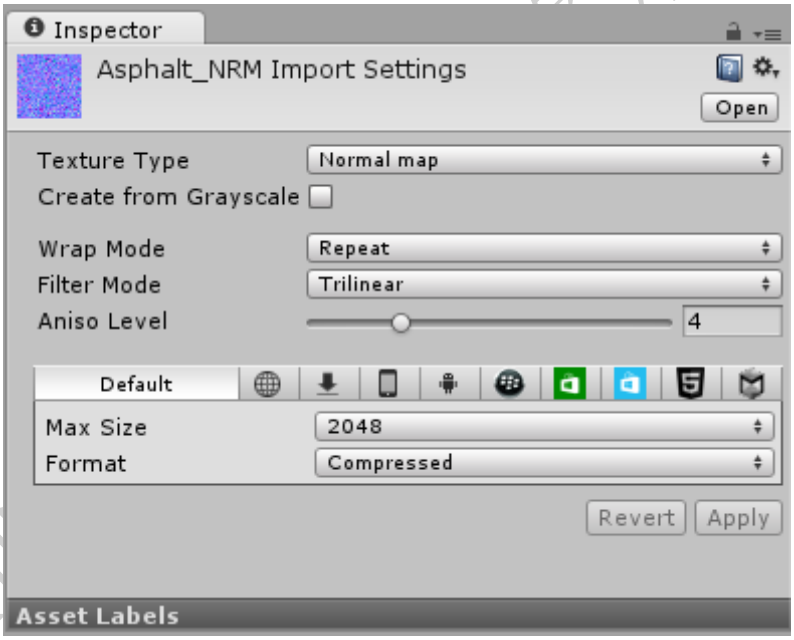


图24.　　将导入的 Texture，设置为 Normal Map

导入黑白的 Height Map 作为 Normal Map，过程也几乎相同，只是您需要选中“从灰度创建”选项。

图25.　　导入 Height Map 作为 Normal Map

选择“Create From Greyscale”后，Inspector 中将出现一个凹凸滑块。当从高度图中的高度转换时，您可以使用它来控制法线贴图中的角度有多陡峭。低凹凸值意味着即使高度图中的强烈对比也会转化为柔和的角度和凹凸。高值会产生夸张的凹凸和对凹凸的非常高对比度的照明响应。

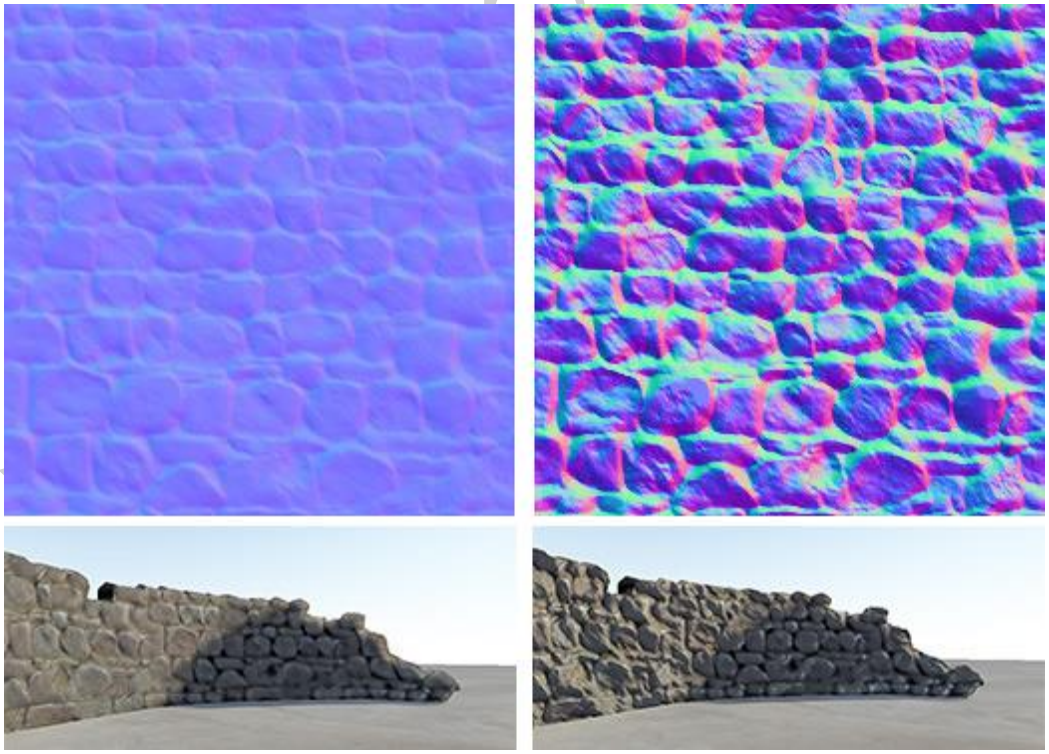


图26.　　将 Height Map 作为 Normal Map 导入时的低（左）和高（右）凹凸设置，及对模型产生的影响

6） 次要 Normal Map

在 Material inspector 的下方有第二个法线贴图插槽，这适用于标准着色器(Standard Shader)。允许我们使用额外的 Normal Map 来创建额外的细节，我们可以使用与常规

Noraml Map 相同的方式来添加法线贴图，但这里的 Texture 应该使用不同的比例或平铺频率，以便两个 Noraml Map 一起产生高水平的细节。

对于不同规模的物体，例如常规 Noraml Map 可以定义墙壁或车辆上镶板的细节，并为镶板边缘设置小树林。辅助 Noraml Map 可以为表面上的划痕和磨损提供非常精细的凹凸细节，可以以基础 Noraml Map 的 5 到 10 倍的比例平铺。这些细节可能非常精细，只有在仔细检查时才能看到。要在基础 Noraml Map 上获得如此多的细节，需要基础 Noraml Map 非常大，但是通过将两个不同比例的两者结合起来，可以使用两个相对较小的 Noraml Map 来实现较高的整体细节水平。

4. 高度图-Height Map

Height Map（也称为视差贴图- parallax map）是与法线贴图类似的概念，但是这种技术更复杂，因此性能也更昂贵。Height Map 通常与 Normal Map 一起使用，并且通常用于为纹理贴图负责渲染的大凹凸和突起的表面，提供一些额外的细节定义。

Normal Map 修改纹理表面的光照，而视差高度贴图则更进一步，实际上它移动了可见表面纹理的区域，以实现一种表面级别的遮挡效果。这意味着明显的凹凸将使其近侧（面向相机）扩大和夸大，而其远侧（背向相机）将缩小并被遮挡在视野之外。

这种效果虽然可以产生非常令人信服的 3D 几何图形表示，但其仍仅限于对象网格的平面多边形的表面。

这意味着虽然表面凸起看起来会突出并相互遮挡，但模型的“轮廓”将永远不会被修改，因为最终效果会被绘制到模型的表面上，并且不会修改实际的几何形状。Height Map 应该是灰度图像，白色区域代表纹理的高区域，黑色代表低区域。

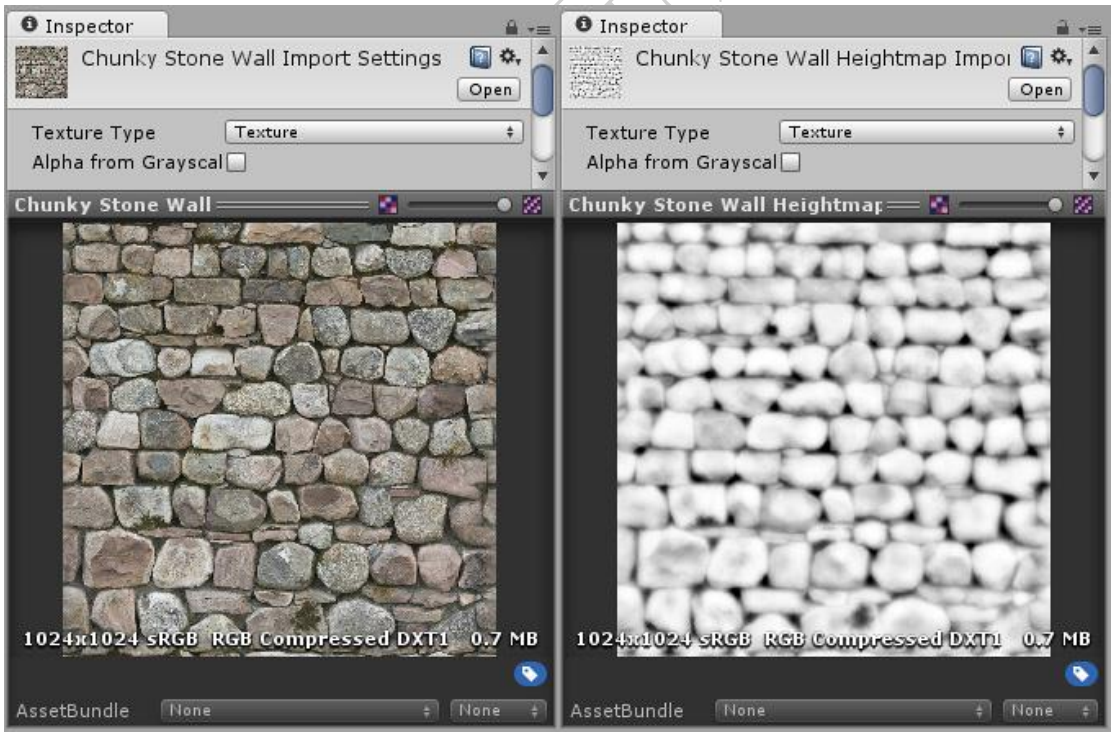


图27. Albedo Map 和要匹配的 Height Map。

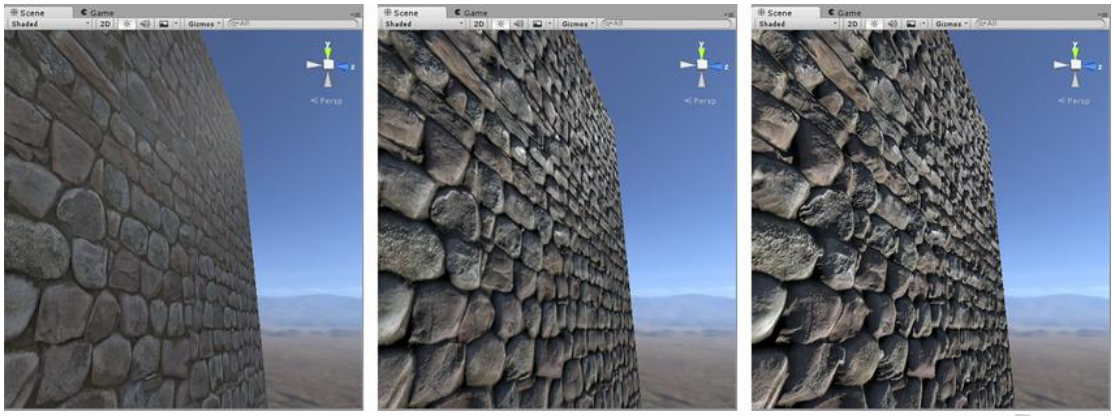


图28.　　Hight Map、Normal Map 和 Albedo Map 的应用

上图中从左到右，左边的是指定了 Albedo Map 但没有 Normal Map 或 Height Map 的石墙材质。中间的是，额外分配了一张 Normal Map，光照在其表面上产生的效果，与左边的图像有所变化，但该图像的岩石不会相互遮挡。右边则是分配三张纹理的最终效果，岩石似乎从地表突出，更近的岩石似乎挡住了它们后面的岩石。

5. 遮蔽贴图- Occlusion Map

Occlusion Map 用于提供有关模型的哪些区域，应接收高或低间接照明的信息。间接照明来自环境照明和反射，因此模型的陡峭凹面部分（如裂缝或褶皱）实际上不会接收到太多间接光。

遮挡纹理贴图通常由 3D 应用程序，使用建模器或第三方软件直接从 3D 模型计算得到。

Occlusion Map 是灰度图像，白色表示应该接收完全间接照明的区域，黑色表示没有间接照明。对于简单的表面，就像 Height Map 一样。

在其他时候，生成正确的 Occlusion Map 稍微复杂一些。 例如，如果场景中的角色戴着头罩，头罩的内侧边缘应设置为非常低的间接照明，或者根本不设置。

图29.　　此遮挡贴图识别角色袖子上暴露或隐藏在环境光照下的区域

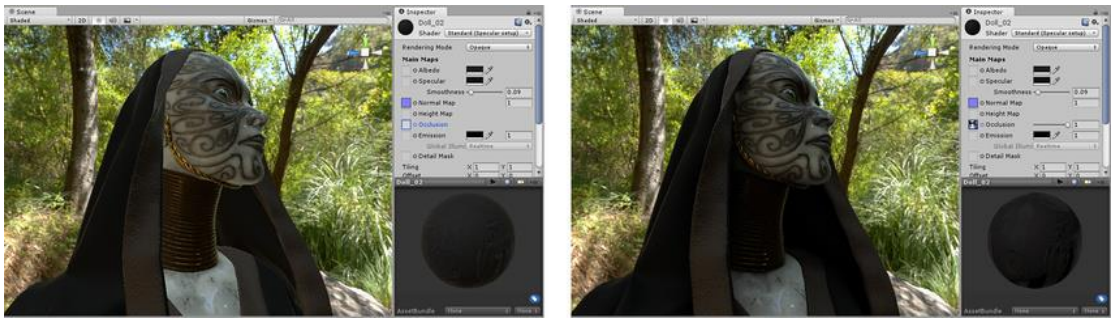


图30.　　应用 Occlusion Map 之前和之后对比

如上图所示，部分区域被遮挡了，尤其是颈部周围的织物褶皱，左侧的光线太亮。分配环境光遮蔽贴图(Ambient Occlusion Map)后，这些区域不再被来自周围树木繁茂的环境的绿色环境光照亮。

四、　渲染流水线

1. 什么是渲染流水线

模型从 3D 到渲染成 2D 图像,中间需要经过一系列的处理过程,这个过程就像是一个流水线。换句话说，计算机需要从一系列的顶点数据、纹理等信息出发，把这些信息最终转换成一张人眼可以看到的图像。而这个工作通常是由 CPU 和 GPU 共同完成。Unity 将此架构分为四个阶段：应用阶段、几何阶段、光栅化阶段和像素处理阶段。

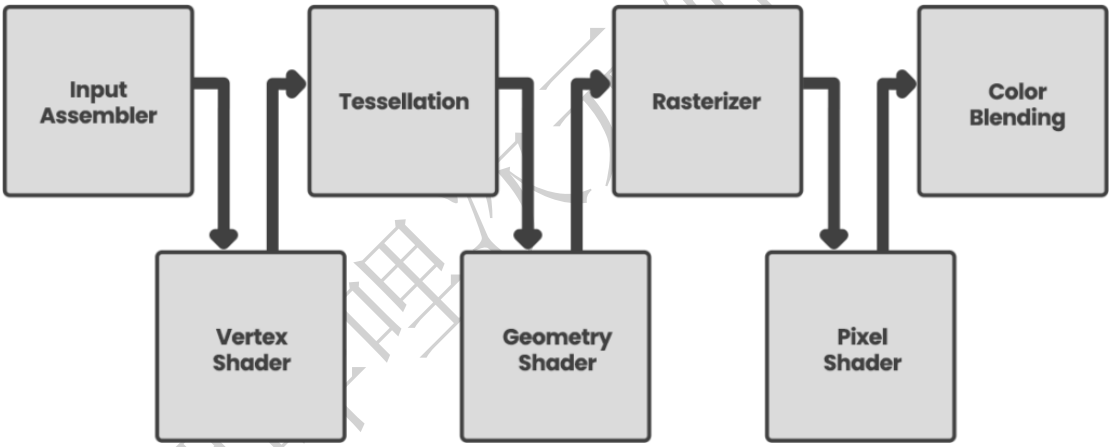


图31.　　逻辑渲染管道

2. 应用阶段(Application Stage)

应用阶段从 CPU 开始，负责场景中发生的各种操作，我们这些开发者具有这个阶段的绝对控制权。例如，碰撞检测、纹理动画、键盘输入、鼠标输入等等。它的功能是读取内存中存储的数据，以便以后生成图元（例如三角形、直线、顶点）。在应用阶段结束时，所有这些信息都被发送到几何处理阶段，通过矩阵乘法生成顶点的变换。

在这一阶段中，开发者有 3 个主要任务：首先，我们需要准备好场景数据，例如摄像机的位置、视锥体、场景中包含了哪些模型、使用了哪些光源等等；其次，为了提高渲染性能，我们往往需要做一个剔除 (culling)工作，以把那些不可见的物体剔除出去，这样就不需要再移交给几何阶段进行处理；最后，我们需要设置好每个模型的渲染状态。这些渲染状态包括但不限于它使用的材质(漫反射颜色、高光反射颜色)、使用的纹理、使用的 Shader 等。这一阶段最重要的输出是渲染所需的几何信息，即渲染图元(rendering primitives)。通俗来讲，渲染图元可以是点、线、三角面等。这些渲染图元将会被传递给下一个阶段——几何阶段。

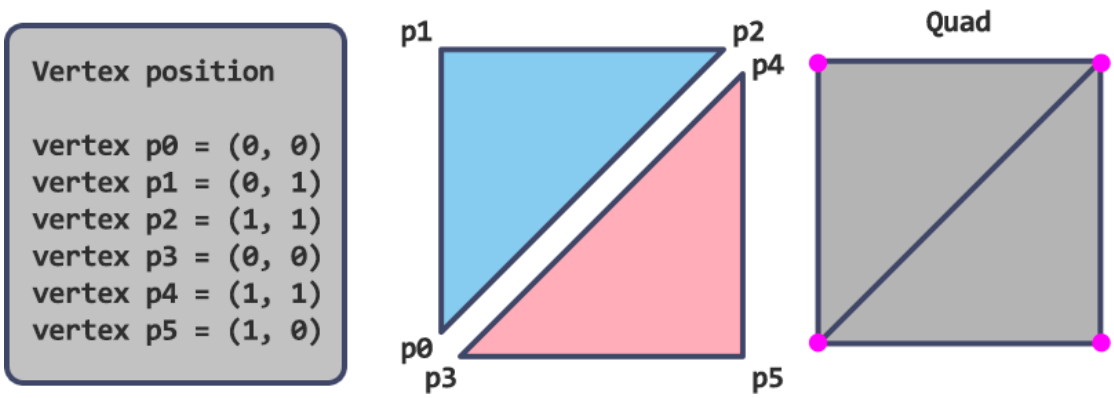


图32.　　渲染图元

3. 几何阶段(Geometry Stage)

几何阶段用于处理所有和我们要绘制的几何相关的事情。例如，决定需要绘制的图元是什么，怎样绘制它们，在哪里绘制它们。这一阶段通常在 GPU 上进行，该阶段分为四个子过程：顶点着色、投影、裁剪和屏幕映射。

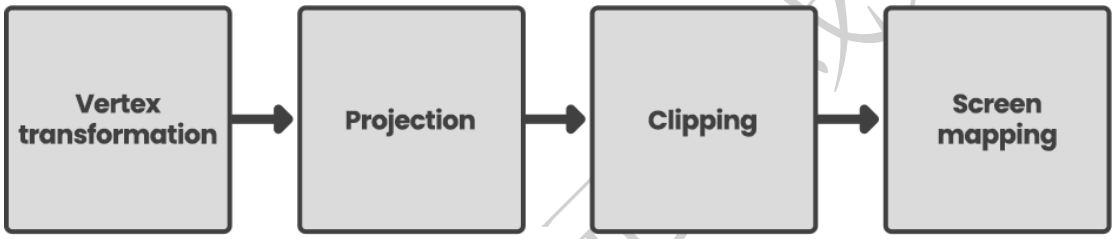


图33.　　几何阶段子过程

几何阶段负责和每个渲染图元打交道，进行逐顶点、逐多边形的操作。几何阶段的一个重要任务就是把顶点坐标变换到屏幕空间中，再交给光栅器进行处理。通过对输入的渲染图元进行多步处理后，这阶段将会输出屏幕空间的二维顶点坐标、每个顶点对应的深度值、着色等相关信息，并传递给下一个阶段。

4. 光栅化阶段(Rasterizer Stage)

至此，我们的物体有了屏幕坐标（2D 坐标），现在我们要寻找投影区域的像素。找到屏幕上对象所占据的所有像素的过程称为光栅化。这个过程可以看作是我们场景中的对象与屏幕上的像素之间同步的一个步骤，在这一阶段将会使用上个阶段传递的数据来产生屏幕上的像素，并渲染出最终的图像。这一阶段也是在 GPU 上运行。光栅化的任务主要是决定每个渲染图元中的哪些像素应该被绘制在屏幕上。它需要对上一个阶段得到的逐顶点数据(例如纹理坐标、顶点颜色等)进行插值，然后再进行逐像素处理。

和上一个阶段类似，光栅化阶段也可以分成更小的流水线阶段。

对于每个对象，光栅化器执行两个过程：三角形设置(Triangle setup)和三角形遍历(Triangle traversal)。

Triangle setup 负责生成将发送到 Triangle traversal 的数据，它包括屏幕上对象边缘的方程。Triangle traversal 列出了多边形对象的区域所覆盖的像素。这样，就产生了一组称为“片元”的像素。但是，这个词多次用于指代单个像素。

5. 像素处理阶段(Pixel processing stage)

当所有像素都准备好投射到屏幕上时，最后一个阶段开始使用先前过程中的插值。此时，片元着色器阶段，也称为像素着色器阶段开始，负责每个像素的可见性。基本上它所做的是计算像素的最终颜色，然后将其发送到颜色缓冲区。

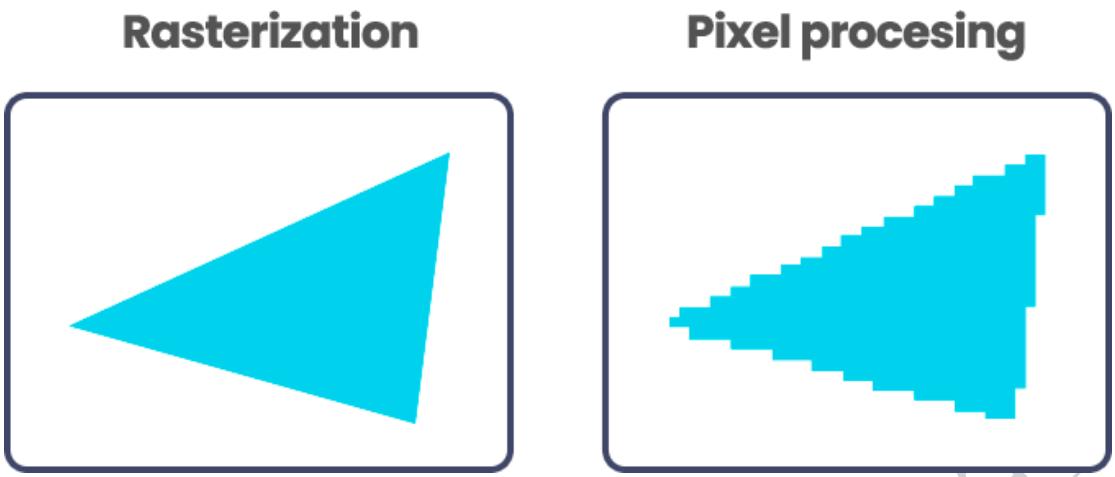


图34.　　几何图形覆盖的区域在屏幕上转换为像素

五、　渲染管线的类型

渲染管线(render pipelines)的类型，正如我们已经知道的，Unity 有三种类型的渲染管线。在默认情况下，我们可以找到与属于该软件的最旧引擎相对应的内置渲染管线(Built-in RP)，在另一方面，Universal RP 和 High Definition RP 属于一种渲染管线，称为可编写脚本的 RP(Scriptable RP)，它更新(up-to-date)得更及时，并针对更好的图形表现进行了预优化(pre-optimized)。

不管(Regardless)渲染管线是什么，如果我们想在屏幕上生成图像，我们必须通过“管线”。一个渲染管线可以有不同的处理路径，被称为渲染路径。一个渲染路径对应一系列光照(lighting)和着色(shading)相关的操作对象，这使我们能够以图形方式处理照明场景（例如，具有定向光和球体）。

这些路径的示例是前向渲染(forward rendering)、延迟渲染(deferred shading)、传统延迟(legacy deferred)和遗留下来的顶点照明(vertex lit)，它们中的每一个都具有不同的功能(capabilities)和性能特征(characteristics)。

在 Unity 中，默认渲染路径对应前向渲染(forward rendering)；这是初始路径。这是因为它有更好的显卡兼容性和光照计算限制，使其可以更加优化过程。

请注意，在 Universal RP 中，我们只能使用 forward 作为渲染路径，而 High Definition RP 允许使用前向或延迟渲染来渲染照明材质。

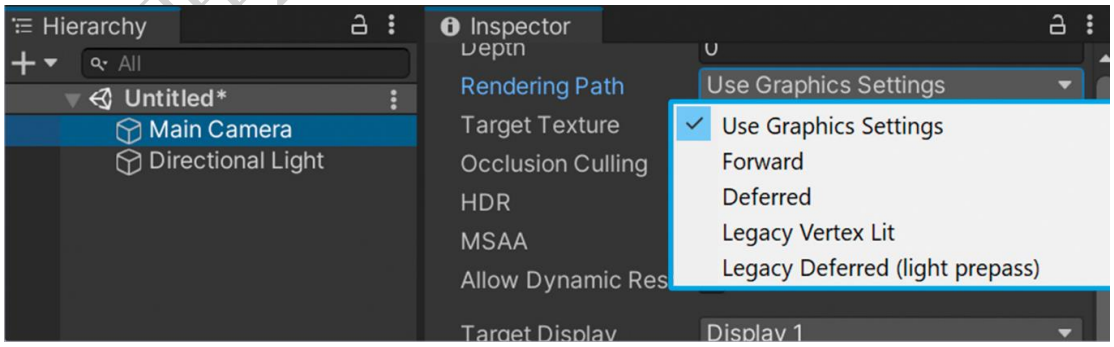


图35.　　内置渲染管线中选择渲染路径

要在内置渲染管道中选择渲染路径，我们必须转到我们的 Hierarchy 窗口，选择主相机，在属性“渲染路径”中，我们可以根据我们项目的需要改变配置。

为了理解这个概念，我们将假设我们有一个“对象”和一个“直接光(Diret light)”在同一个场景中。光与物体之间的相互作用基于两点，光源性质和材质性质(Material

characteristics)。这两个元素之间的交互称为光照模型，基本光照模型对应于三个不同属性的总和，它们是环境颜色(ambient color)、漫反射(diffuse reflection)和镜面反射(specular reflection)。光照计算在着色器(shader)中进行，可以逐顶点(per-vertex)进行或逐片元(per fragment)。 当按顶点计算光照时，称为逐顶点光照并且在顶点着色器阶段执行，同样，当它按片元计算时被称为 per-fragment 或 per-pixel shader，在片元着色器阶段执行。

1. 前向渲染-Forward rendering

前向渲染是默认渲染路径，支持材质的所有典型特征（例如法线贴图、像素光照(pixel lighting)、阴影等）。 这个渲染路径有两个不同的代码，我们可以在着色器中使用的通道，第一个是基本通道(base pass)和第二个附加通道(additional pass)。

在 base pass 中，我们可以定义 ForwardBase 光照模式，在 additional pass 中，我们可以定义 ForwardAdd 光照模式。两者都是具有光照计算的着色器的特征函数。Base pass 可以按逐像素处理定向光，如果场景中有多个定向光源，则将使用最亮的光源。此外，base pass 可以处理光照探针、全局照明(global illumination)和环境照明（skylight）。

顾名思义，additional pass 可以处理每个像素的“附加光”，或者影响物体的阴影，这是什么意思？如果我们在场景中有两盏灯，我们的对象将只受其中一个的影响，但是，如果我们为此配置定义了一个额外的通道，那么它将受到两者的影响。

我们必须考虑的一点是，每个被照亮的通道都会生成一个单独的 draw call。draw call 是每次我们想在计算机屏幕上绘制元素时，在 GPU 中进行的图形调用。这些调用是需要大量计算的，因此需要将它们保持在最低限度，如果我们正在为移动设备开发项目，则更是如此。

为了理解这个概念，我们将假设我们场景中有四个球体和一个定向光。 每个 Sphere 本质上都会产生对 GPU 的调用，这意味着他们每一个都会默认生成一个独立的 draw call。同样，定向光会影响场景中的所有球体，因此，它将为每个 Sphere 生成一个额外的 draw call，这主要是因为着色器中已经包含了第二个 pass 来计算阴影投影，因此，四个球体加上单向光总共会产生 8 个 draw call。

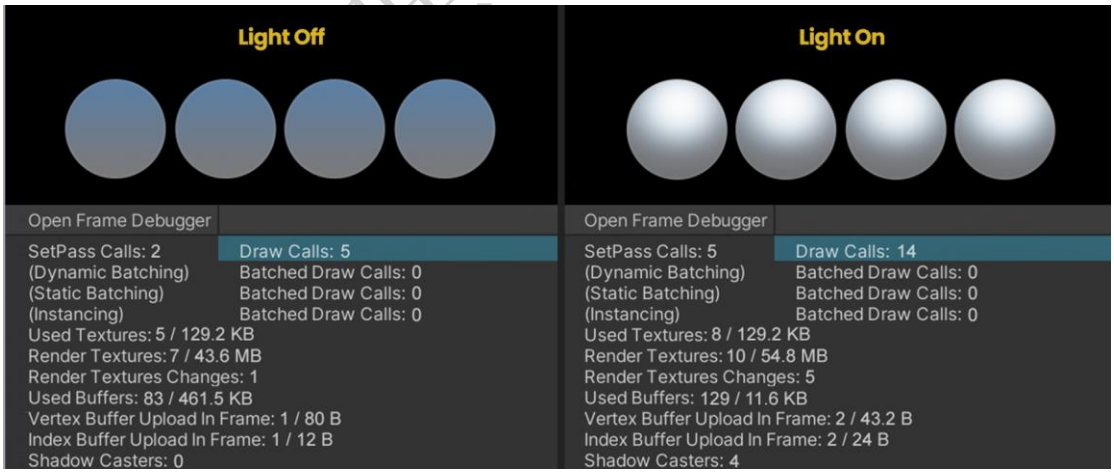


图36. 开启光照和关闭光照时的 draw call

在上图中，我们可以看到有光源时 draw call 的增加，其增加的计算包括环境颜色和作用于对象的光源。确定了 base pass 后，如果我们在着色器中添加另一个 pass，那么我们将继续为每个对象添加一个新的 draw call，因此，图形负载将显着地增加。

2. 延迟渲染- Deferred shading

此渲染路径确保只有一个 lighting pass 用于计算场景中的每个光源，并且仅在受它们影响的像素中，所有这些都是通过几何体和光照的分离来实现的。这是一个优点

(advantage)，因为我们可以生成大量影响不同对象的光，从而提高最终渲染的保真度，但名义上增加了 GPU 上的每像素的计算。虽然 Deferred Shading 在多光源计算方面优于 Forward，但它带来了一些硬件兼容性限制和问题。

3. 应该使用什么渲染

这是当今非常普遍且反复出现的问题。过去，只有 Build-in RP，所以很容易开始一个项目，无论是 2D 项目还是 3D 项目。但是现在，我们必须根据游戏的需求来选择。因此，我们可能会想，我们怎么知道我们的项目需要什么？要回答这个问题，我们必须考虑以下因素。

如果我们要开发 PC 游戏，我们可以使用三种 Unity 渲染管线中的任何一种，因为通常，PC 比移动设备甚至控制台具有更大的计算能力。那么，如果我们的游戏是用于 PC 平台的，我们需要判断是否以高清晰度或中等清晰度的方式游玩它？如果要以高清进行游玩，我们则需要使用 HDRP 或者 Build-in。

如果我们希望我们的游戏具有中等清晰度的图形，我们可以使用 Universal RP，或者像前面一样使用 Bulid-in RP 也可以。现在我们有一个疑问，为什么内置渲染管线在这两种情况下都适用？

Build-in 与其他的不同，这个渲染管线更加灵活，它更具技术性并且没有预优化。High Definition RP 已针对生成高端图形进行了预优化，Universal RP 已针对中档图形进行了预优化。

选择渲染管线时的另一个重要因素是 Shader。一般来说，在 High Definition RP 和 Universal RP 中，Shader 都是在 Shader Graph 中创建的，Shader Graph 是一个包，它包含一个允许我们通过节点创建着色器的接口，它有积极和消极的一面。一方面，我们可以通过节点直观地创建 Shader，而无需在 HLSL 中编写代码，但是另一方面，如果我们将 Unity 版本更新到较新版本（例如，从 2019 年更新到 2020 年），着色器很可能会停止编译，因为 Shader Graph 具有独立于 Unity 版本的版本和更新。

在 Unity 中创建 Shader 的最佳方法是通过 HLSL 语言，因为这样我们可以确保我们的程序在不同的渲染管线中编译，并且无论 Unity 进行任何升级都可以继续工作。

六、 CPU 和 GPU 之间的通信

渲染流水线的起点是 CPU,即应用阶段。应用阶段大致可分为下面 3 个阶段：

① 把数据加载到显存中

② 设置渲染状态

③ 调用 DrawCall

1. 把数据加载到显存中

所有渲染所需的数据都需要从硬盘(Hard Disk Drive，HDD)中加载到系统内存(Random Access Memory,，RAM)中。然后，网格和纹理等数据又被加载到显卡上的存储空间——显存(Video Random Access Memory,，VRAM)中。这是因为，显卡对于显存的访问速度更快，而且大多数显卡对于 RAM 没有直接的访问权利。

当把数据加载到显存中后，RAM 中的数据就可以移除了。但对于一些数据来说，CPU 仍然需要访问它们(例如，我们希望 CPU 可以访问网格数据来进行碰撞检测)，那么我们可能就不希望这些数据被移除，因为从硬盘加载到 RAM 的过程是十分耗时的。

在这之后，开发者还需要通过 CPU 来设置渲染状态，从而“指导”GPU 如何进行渲染工作。

2. 设置渲染状态

什么是渲染状态呢?一个通俗的解释就是，这些状态定义了场景中的网格是怎样被渲染

的。例如，使用哪个顶点着色器(Vertex Shader)/片元着色器(Fragment Shader)、光源属性、材质等。如果我们没有更改渲染状态，那么所有的网格都将使用同一种渲染状态。

3. 调用 Draw Call

Draw Call 实际上就是一个命令，它的发起方是 CPU，接收方是 GPU。这个命令仅仅会指向一个需要被渲染的图元(primitives)列表，而不会再包含任何材质信息——这是因为我们已经在上一个阶段中完成了！

当给定了一个 Draw Call 时，GPU 就会根据渲染状态（例如材质、纹理、着色器等）和所有输入的顶点数据来进行计算，最终输出成屏幕上那些漂亮的像素。而这个计算过程就是 GPU 流水线。

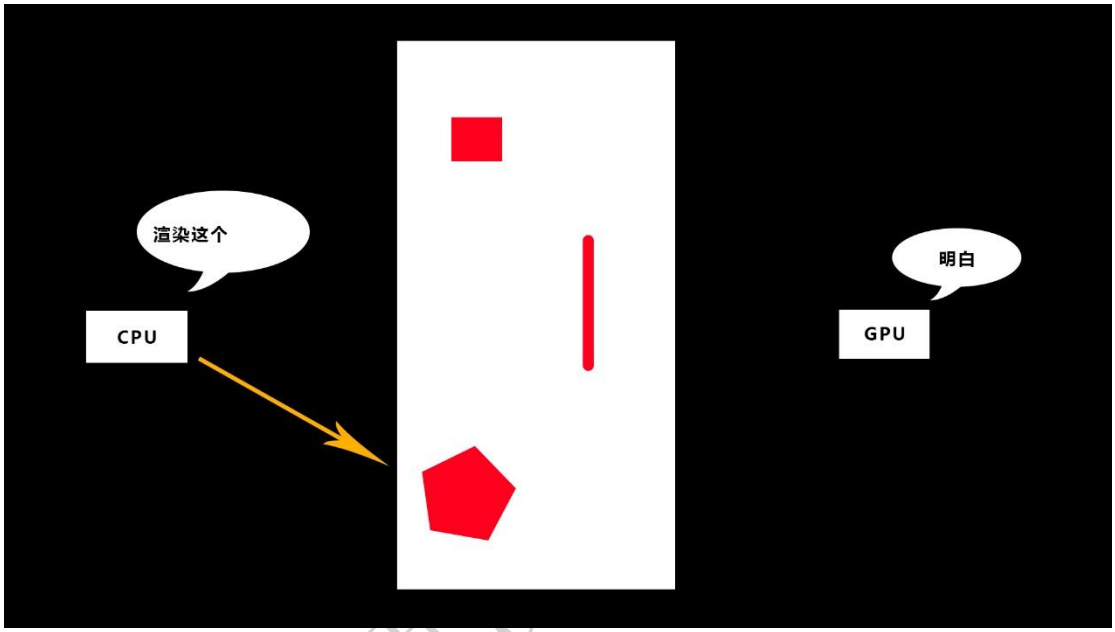


图37. Cpu 通过调用 Draw Call 告知 GPU 开始进行一个渲染过程。

七、 GPU 流水线

图形渲染流水线的完整流程需要 CPU、内存、GPU 等硬件的共同参与才能完成,其中 GPU 在整个谊染流水线中起到非常重要的作用，并且 Shader 也是在 GPU 上运行的。

几何阶段和光栅化阶段，开发者无法拥有绝对的控制权，其实现的载体是 GPU。GPU 通过实现流水线化，大大加快了渲染速度。虽然我们无法完全控制这两个阶段的实现细节，但 GPU 向开发者开放了很多控制权。

几何阶段和光栅化阶段可以分成若干更小的流水线阶段，这些流水线阶段由 GPU 来实现，每个阶段 GPU 提供了不同的可配置性或可编程性。

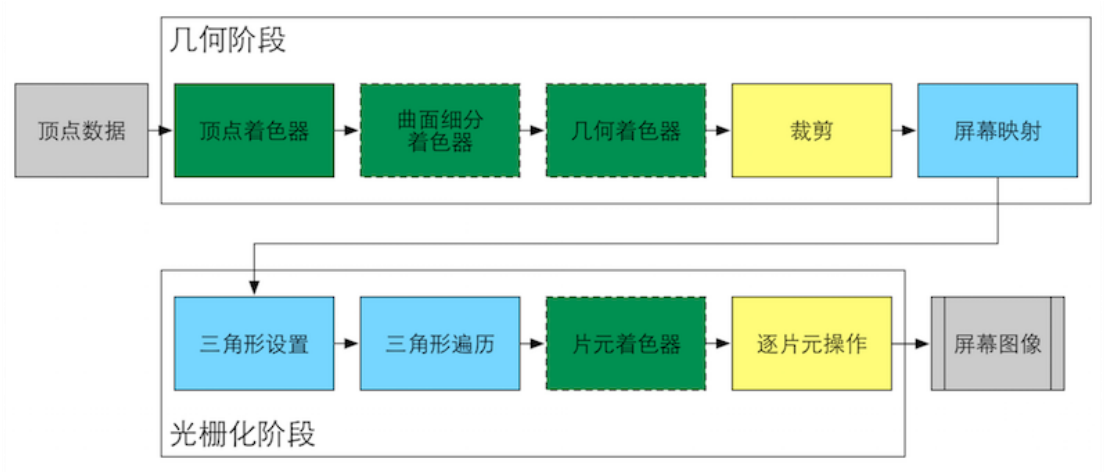


图38.　　GPU 的渲染流水线实现

上图的颜色表示了不同阶段的可配置性或可编程性；绿色表示该流水线阶段是完全可编程控制的，黄色表示该流水线阶段可以配置但不是可编程的，蓝色表示该流水线阶段是由 GPU 固定实现的，开发者没有任何控制权。实线表示该 Shader 必须由开发者编程实现，虚线表示该 Shader 是可选的。

从图中也可以看出，GPU 的渲染流水线接收顶点数据作为输入。这些顶点数据是由应用阶段加载到显存中，再由 Draw Call 指定的。这些数据随后被传递给顶点着色器。

顶点着色器（Vertex Shader）是完全可编程的，它通常用于实现顶点的空间变换、顶点着色等功能。曲面细分着色器（Tessellation Shader）是一个可选的着色器，它用于细分图元。几何着色器（Geometry Shader）同样是一个可选的着色器，它可以被用于执行逐图元（Per-Primitive）的着色操作，或者被用于产生更多新的图元。下一个流水线阶段是裁剪(Clipping)，这一阶段的目的是将那些不在摄像机视野内的顶点裁剪掉，并剔除某些三角图元的面片。这个阶段是可配置的。

例如，我们可以使用自定义的裁剪平面来配置裁剪区域，也可以通过指令控制裁剪三角图元的正面还是背面。几何概念阶段的最后一个流水线阶段是屏幕映射（Screen Mapping.）。这一阶段是不可配置和编程的，它负责把每个图元的坐标转换到屏幕坐标系中。光栅化概念阶段中的三角形设置（Triangle Setup）和三角形遍历（Triangle Traversal）阶段也都是固定函数（Fixed-Function）的阶段。

接下来的片元着色器（Fragment Shader），则是完全可编程的，它用于实现逐片元（Per-Fragment）的着色操作。最后，逐片元操作（Per-Fragment Operations）阶段负责执行很多重要的操作，例如修改颜色、深度缓冲、进行混合等，它不是可编程的，但具有很高的可配置性。

1. 顶点数据

顶点数据用来为后面的顶点着色器等阶段提供处理的数据，是渲染管线数据的主要来源。送入到渲染管线的数据包括顶点坐标、纹理坐标、顶点法线和顶点颜色等顶点属性。为了让 OpenGL 明白顶点数据构成的是什么图元，我们需要在绘制指令中传递相对应的图元信息。常见的图元包括：点(GL_POINTS)、线(GL_LINES)、线条(GL_LINE_STRIP)、三角面(GL_TRIANGLES)。

2. 顶点着色器

顶点着色器（Vertex Shader）是流水线的第一个阶段，它的输入来自于 CPU。顶点着色器的处理单位是顶点，也就是说，输入进来的每个顶点都会调用一次顶点着色器。顶点着色器本身不可以创建或者销毁任何顶点，而且无法得到顶点与顶点之间的关系。例如，

我们无法得知两个顶点是否属于同一个三角网格。但正是因为这样的相互独立性，GPU 可以利用本身的特性并行化处理每一个顶点，这意味着这一阶段的处理速度会很快。

顶点着色器需要完成的工作主要有：坐标变换和逐顶点光照。当然，除了这两个主要任务外，顶点着色器还可以输出后续阶段所需的数据。

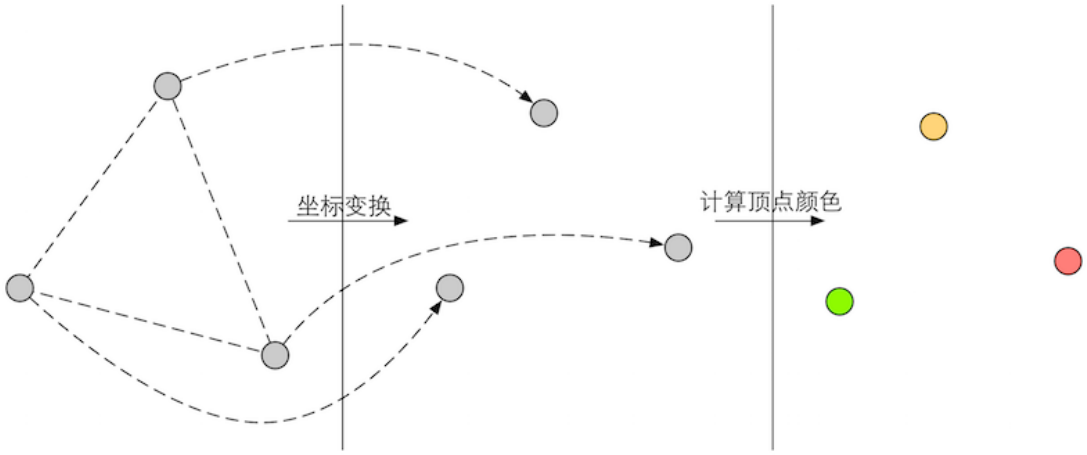


图39.　　在顶点着色器中对顶点位置进行坐标变换并计算顶点颜色的过程

坐标变换，就是对顶点的坐标（即位置）进行某种变换。顶点着色器可以在这一步中改变顶点的位置，这在顶点动画中是非常有用的。例如，我们可以通过改变顶点位置来模拟水面、布料等。但需要注意的是，无论我们在顶点着色器中怎样改变顶点的位置，一个最基本的顶点着色器必须完成的一个工作是，把顶点坐标从模型空间转换到齐次裁剪空间。想想看，我们在顶点着色器中是不是会看到类似下面的代码：

o.pos = mul（UNITY_MVP，v.position）；

类似上面这句代码的功能，就是把顶点坐标转换到齐次裁剪坐标系下，接着通常再由硬件做透视除法后，最终得到归一化的设备坐标（Normalized Device Coordinates，NDC）。

图40.　　将模型顶点的位置变换到齐次裁剪坐标空间下

需要注意的是，图 2.8 给出的坐标范围是 OpenGL 同时也是 Unity 使用的 NDC，它的 z 分量范围在［-1，1］之间，而在 DirectX 中，NDC 的 z 分量范围是［0，1］。顶点着色器可以有不同的输出方式。最常见的输出路径是经光栅化后交给片元着色器进行处理。而在现代的 Shader Model 中，它还可以把数据发送给曲面细分着色器或几何着色器。

3. 曲面细分

曲面细分是利用镶嵌化处理技术对三角面进行细分，以此来增加物体表面的三角面的数量，是渲染管线一个可选的阶段。它由外壳着色器(Hull Shader)、镶嵌器(Tessellator)和域着色器(Domain Shader)构成，其中外壳着色器和域着色器是可编程的，而镶嵌器是由硬件

管理的。我们可以借助曲面细分的技术实现细节层次(Level-of-Detail)的机制，使得离摄像机越近的物体具有更加丰富的细节，而远离摄像机的物体具有较少的细节。

4. 几何着色器

几何着色器也是渲染管线一个可选的阶段。我们知道，顶点着色器的输入是单个顶点(以及属性)， 输出的是经过变换后的顶点。与顶点着色器不同，几何着色器的输入是完整的图元(比如，点)，输出可以是一个或多个其他的图元(比如，三角面)，或者不输出任何的图元。几何着色器的拿手好戏就是将输入的点或线扩展成多边形。下图展示了几何着色器如何将点扩展成多边形。

5. 裁剪

由于我们的场景可能会很大，而摄像机的视野范围很有可能不会覆盖所有的场景物体，一个很自然的想法就是，那些不在摄像机视野范围的物体不需要被处理。而裁剪（Clipping）就是为了完成这个目的而被提出来的。

一个图元和摄像机视野的关系有 3 种：完全在视野内、部分在视野内、完全在视野外。完全在视野内的图元就继续传递给下一个流水线阶段，完全在视野外的图元不会继续向下传递，因为它们不需要被渲染。而那些部分在视野内的图元需要进行一个处理，这就是裁剪。例如，一条线段的一个顶点在视野内，而另一个顶点不在视野内，那么在视野外部的顶点应该使用一个新的顶点来代替，这个新的顶点位于这条线段和视野边界的交点处。

由于我们已知在 NDC 下的顶点位置，即顶点位置在一个立方体内，因此裁剪就变得很简单：只需要将图元裁剪到单位立方体内。

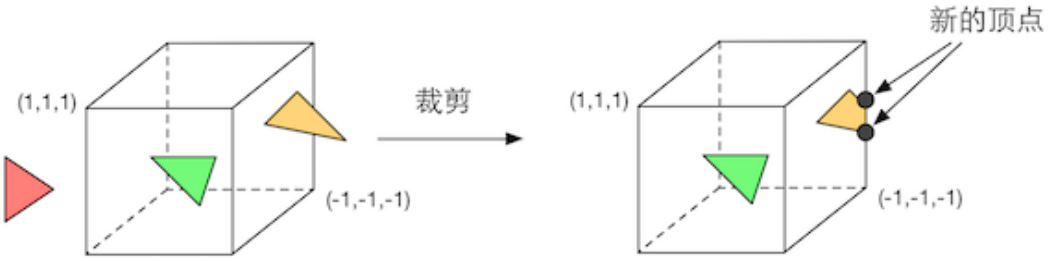


图41. 将图元裁剪到单位立方体内

如上图所示，只有在单位立方体的图元才需要被继续处理。完全在单位立方体外部的图元（红色三角形）被舍弃，完全在单位立方体内部的图元（绿色三角形）将被保留。和单位立方体相交的图元（黄色三角形）会被裁剪，新的顶点会被生成，原来在外部的顶点会被舍弃。

和顶点着色器不同，这一步是不可编程的，即我们无法通过编程来控制裁剪的过程，而是硬件上的固定操作，但我们可以自定义一个裁剪操作来对这一步进行配置。

6. 屏幕映射

这一步输入的坐标仍然是三维坐标系下的坐标（范围在单位立方体内）。屏幕映射（Screen Mapping）的任务是把每个图元的 x 和 y 坐标转换到屏幕坐标系 (Screen Coorcdinates)下，屏幕坐标系是一个二维坐标系，它和我们用于显示画面的分辨率有很大关系。

假设，我们需要把场景渲染到一个窗口上，窗口的范围是从最小的窗口坐标(x_1,y_1)到最大的窗口坐标(x_2,y_2)，其中 $x_1<x_2$ 且 $y_1<y_2$，由于我们输入的坐标范围在负 1 到 1，因此可以想象到，这个过程实际是一个缩放的过程。屏幕映射不会对输入的 Z 坐标做任何处理，

安际上，屏幕坐标系和 Z 坐标一起构成了一个坐标系，叫做窗口坐标系(Window Coordinates)。这些值会一起被传连到光栅化阶段。

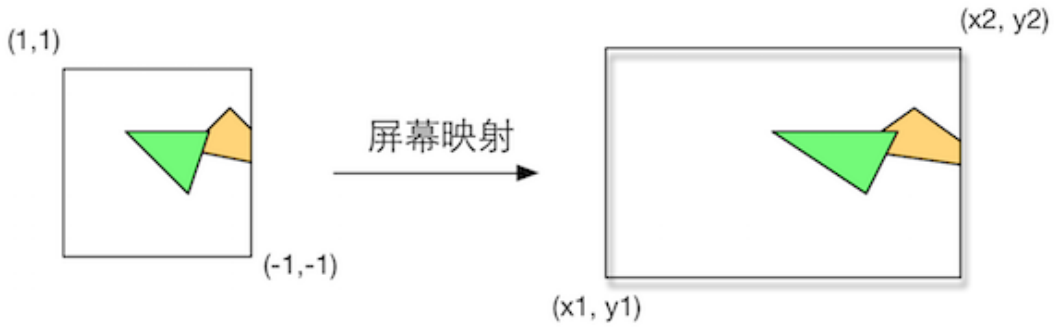


图42.　　屏幕映射

屏幕映射得到的屏幕坐标决定了这个顶点对应屏幕上哪个像素以及距离这个像素有多远。屏幕坐标系在 OpenGL 和 DirectX 之间的差异问题，OpenGL 把屏幕的左下角当成最小的窗口坐标值，而 DirectX 则定义了屏幕的左上角为最小的窗口坐标值。

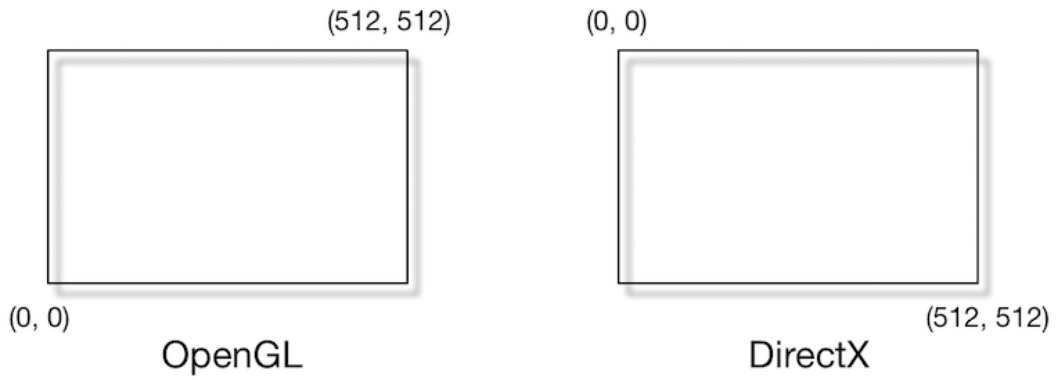


图43.　　OpenGL 和 DirectX 屏幕坐标系的差异

产生这种差异的原因是，微软的窗口都使用了这样的坐标系统，因为这和我们的阅读方式是一致的：从左到右、从上到下，并且很多图像文件也是按照这样的格式进行存储的。

不管原因如何，差异就这么造成了，留给我们开发者的就是，要时刻小心这样的差异，不同的软件也会有一些微小的差异，如果你发现得到的图像是倒转的，那么很有可能就是这个原因造成。

7. 三角形设置

由这一步开始就进入了光栅化阶段。从上一个阶段输出的信息是屏幕坐标系下的顶点位置以及和它们相关的额外信息，如深度值（z 坐标）、法线方向、视角方向等。光栅化阶段有两个最重要的目标：计算每个图元赋盖了哪些像素，以及为这些像素计算它们的颜色。

光栅化的第一个流水线阶段是三角形设置(Triangle Setup)。这个阶段会计算光栅化一个三角形所需的信息。上一个阶段给出的都是三角网格的顶点，即我们得到的是三角网格每条边的两个端点，但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。为了能够计算边界像素的坐标信息，我们就需要得到三角形边界的表示方式。这样一个计算三角网格表示数据的过程就叫做三角形设置。它的输出是为了给下一个

阶段做准备。

8. 三角形遍历

三角形遍历（Triangle Traversal）阶段将会检查每个像素是否被一个三角网格所覆盖。如果被覆盖的话，就会生成一个片元（fragment）。而这样一个找到哪些像素被三角网格覆盖的过程就是三角形遍历，这个阶段也被称为扫描变换（Scan Conversion）。

三角形遍历阶段会根据上一个阶段的计算结果来判断一个三角网格覆盖了哪些像素，并使用三角网格 3 个顶点的顶点信息对整个覆盖区域的像素进行插值。

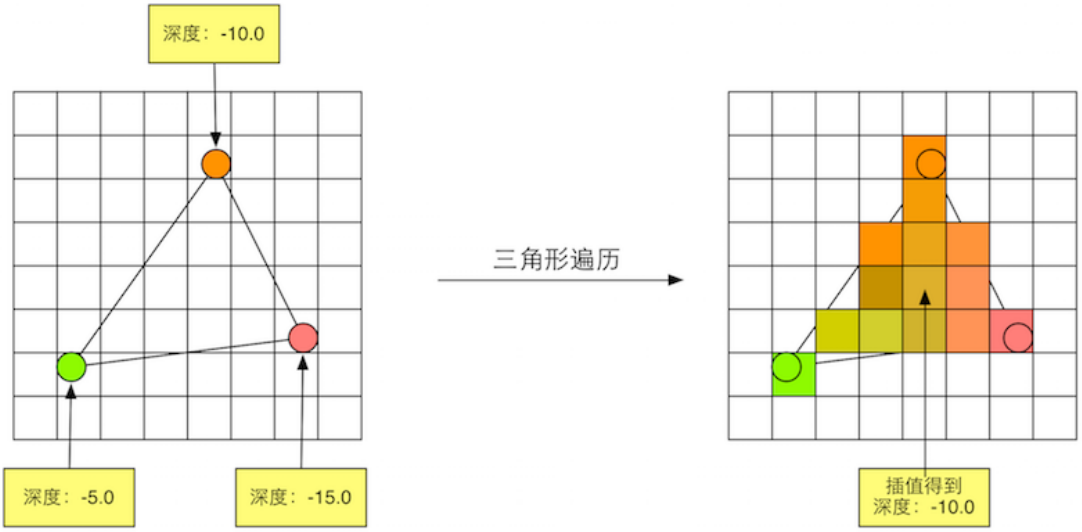


图44. 三角形遍历的过程

这一步的输出就是得到一个片元序列。需要注意的是，一个片元并不是真正意义上的像素，而是包含了很多状态的集合，这些状态用于计算每个像素的最终颜色。这些状态包括了（但不限于）它的屏幕坐标、深度信息，以及其他从几何阶段输出的顶点信息，例如法线、纹理坐标等。

9. 片元着色器

片元着色器（Fragment Shader）是另一个非常重要的可编程着色器阶段。在 DirectX 中，片元着色器被称为像素着色器（Pixel Shader），但片元着色器是一个更合适的名字，因为此时的片元并不是一个真正意义上的像素。

前面的光栅化阶段实际上并不会影响屏幕上每个像素的颜色值，而是会产生一系列的数据信息，用来表述一个三角网格是怎样覆盖每个像素的。而每个片元就负责存储这样一系列数据。真正会对像素产生影响的阶段是下一个流水线阶段——逐片元操作（Per Fragment Operations）。

片元着色器的输入是上一个阶段对顶点信息插值得到的结果，更具体来说，是根据那些从顶点着色器中输出的数据插值得到的。而它的输出是一个或者多个颜色值。

腾讯课堂

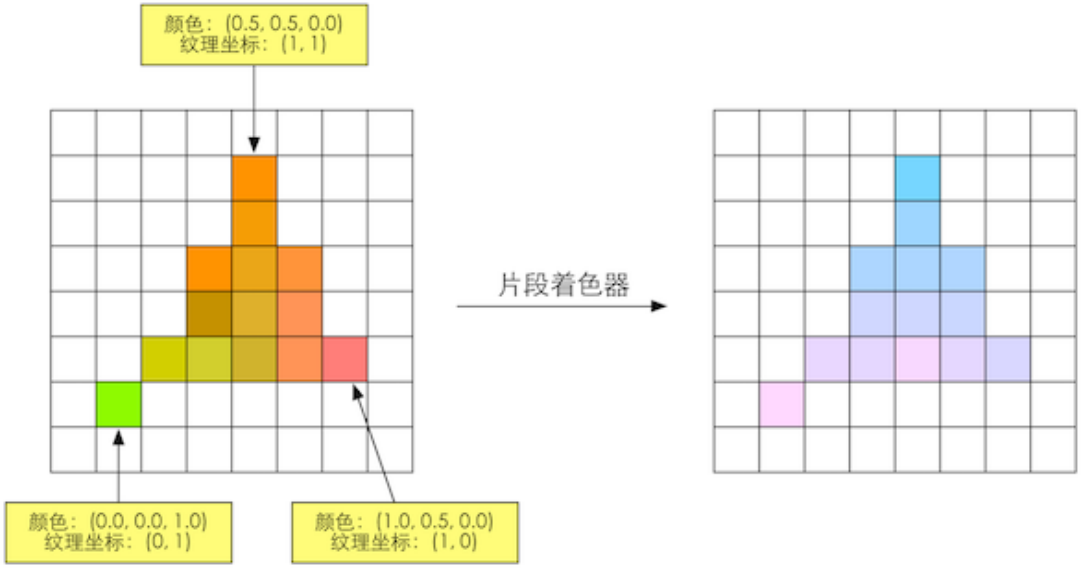


图45.　　根据三角形遍历差值后的片元信息，片元着色器计算该片元的输出颜色

这一阶段可以完成很多重要的渲染技术，其中最重要的技术之一就是纹理采样。为了在片元着色器中进行纹理采样，我们通常会在顶点着色器阶段输出每个顶点对应的纹理坐标，然后经过光栅化阶段对三角网格的 3 个顶点对应的纹理坐标进行插值后，就可以得到其覆盖的片元的纹理坐标了。

虽然片元着色器可以完成很多重要效果，但它的局限在于，它仅可以影响单个片元。也就是说，当执行片元着色器时，它不可以将自己的任何结果直接发送给它的邻居们。有一个情况例外，就是片元着色器可以访问到导数信息（gradient，或者说是 derivative）。

10. 逐片元操作

终于到了渲染流水线的最后一步。逐片元操作（Per-Fragment Operations）是 OpenGL 中的说法，在 DirectX 中，这一阶段被称为输出合并阶段（Output-Merger）。Merger 这个词可能更容易让读者明白这一步骤的目的：合并。而 OpenGL 中的名字可以让读者明白这个阶段的操作单位，即是对每一个片元进行一些操作。那么问题来了，要合并哪些数据？又要进行哪些操作呢？

这一阶段有几个主要任务。

（1）决定每个片元的可见性。这涉及了很多测试工作，例如深度测试、模板测试等。

（2）如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说是混合。

需要指明的是，逐片元操作阶段是高度可配置性的，即我们可以设置每一步的操作细节。

这个阶段首先需要解决每个片元的可见性问题。这需要进行一系列测试。这就好比考试，一个片元只有通过了所有的考试，才能最终获得和 GPU 谈判的资格，这个资格指的是它可以和颜色缓冲区进行合并。如果它没有通过其中的某一个测试，那么对不起，之前为了产生这个片元所做的所有工作都是白费的，因为这个片元会被舍弃掉。Poor fragment！

图46.　　逐片元阶段所做的操作

测试的过程实际上是个比较复杂的过程，而且不同的图形接口（例如 OpenGL 和 DirectX）的实现细节也不尽相同。这里给出两个最基本的测试——深度测试和模板测试的实现过程。能否理解这些测试过程将关乎读者是否可以理解后面提到的渲染队列，尤其是处理透明效果时出现的问题。

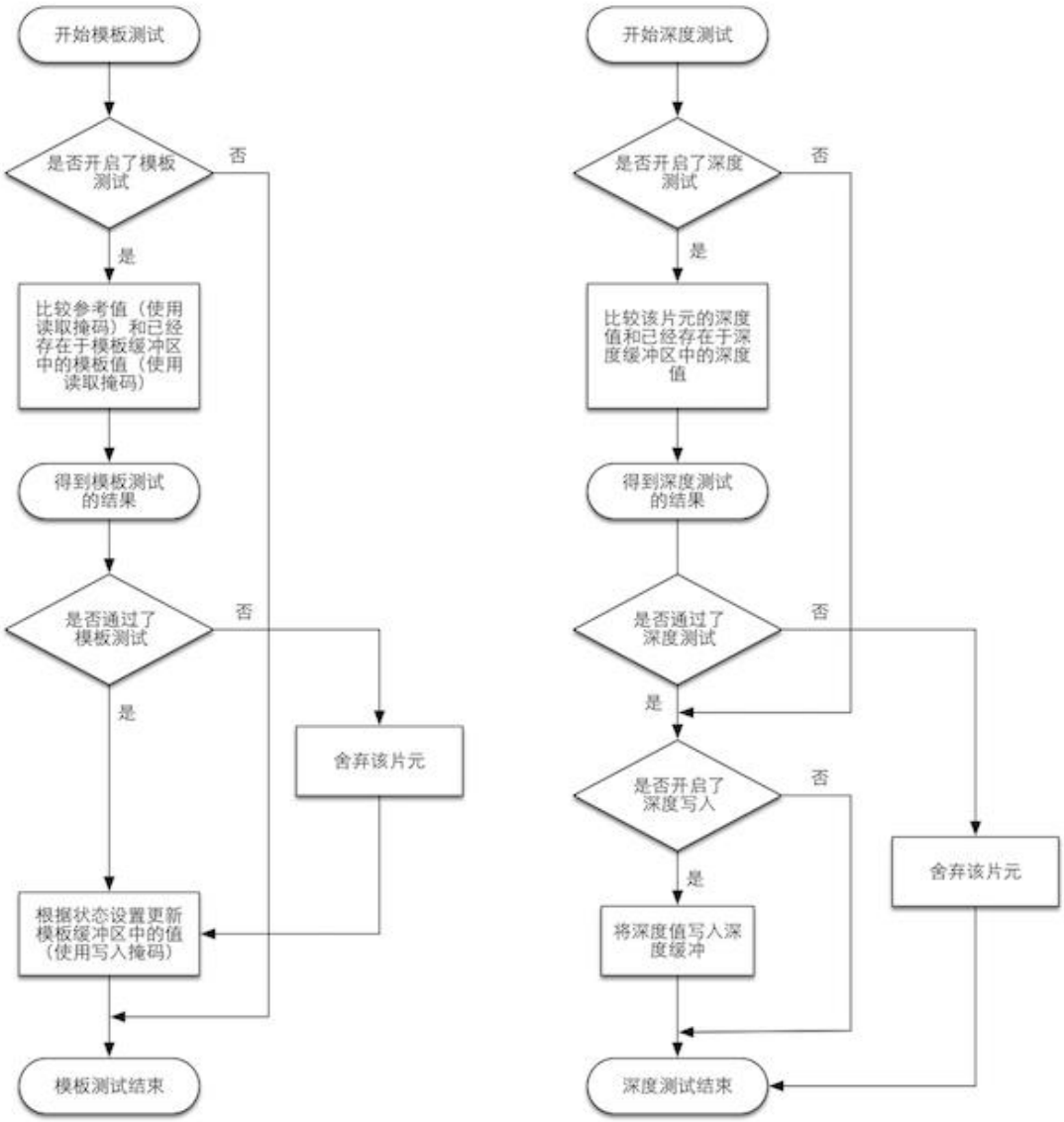


图47.　　模板测试和深度测试的简化流程图

1）模板测试-Stencil Test

与之相关的是模板缓冲（Stencil Buffer）。实际上，模板缓冲和我们经常听到的颜色缓冲、深度缓冲几乎是一类东西。如果开启了模板测试，GPU 会首先读取（使用读取掩码）模板缓冲区中该片元位置的模板值，然后将该值和读取（使用读取掩码）到的参考值（reference value）进行比较，这个比较函数可以是由开发者指定的，例如小于时舍弃该片元，或者大于等于时舍弃该片元。

如果这个片元没有通过这个测试，该片元就会被舍弃。不管一个片元有没有通过模板测试，我们都可以根据模板测试和下面的深度测试结果来修改模板缓冲区，这个修改操作也是由开发者指定的。开发者可以设置不同结果下的修改操作，例如，在失败时模板缓冲区保持不变，通过时将模板缓冲区中对应位置的值加 1 等。

模板测试通常用于限制渲染的区域。另外，模板测试还有一些更高级的用法，如渲染

阴影、轮廓渲染等。

2） 深度测试- Depth Test

如果一个片元幸运地通过了模板测试，那么它会进行下一个测试一深度测试（Depth Test）。深度测试同样是可以高度配置的，如果开启了深度测试，GPU 会把该片元的深度值和已经存在于深度缓冲区中的深度值进行比较。这个比较函数也是可由开发者设置的，例如小于时舍弃该片元，或者大于等于时舍弃该片元。

通常这个比较函数是小于等于的关系，即如果这个片元的深度值大于等于当前深度缓冲区中的值，那么就会舍弃它。这是因为，我们总想只显示出离摄像机最近的物体，而那些被其他物体遮挡的就不需要出现在屏幕上。

如果这个片元没有通过这个测试，该片元就会被舍弃。和模板测试有些不同的是，如果一个片元没有通过深度测试，它就没有权利更改深度缓冲区中的值。而如果它通过了测试，开发者还可以指定是否要用这个片元的深度值覆盖掉原有的深度值，这是通过开启/关闭深度写入来做到的。

在后面的学习中会发现，透明效果和深度测试以及深度写入的关系非常密切。

3） 混合操作-Blend

如果一个幸运的片元通过了上面的所有测试，它就可以自豪地来到合并功能的面前。

为什么需要合并？我们要知道，这里所讨论的渲染过程是一个物体接着一个物体画到屏幕上的。而每个像素的颜色信息被存储在一个名为颜色缓冲的地方。因此，当我们执行这次渲染时，颜色缓冲中往往已经有了上次渲染之后的颜色结果，那么，我们是使用这次渲染得到的颜色完全覆盖掉之前的结果，还是进行其他处理？这就是合并需要解决的问题。

对于不透明物体，开发者可以关闭混合（Blend）操作。这样片元着色器计算得到的颜色值就会直接覆盖掉颜色缓冲区中的像素值。但对于半透明物体，我们就需要使用混合操作来让这个物体看起来是透明的。

哔哩哔哩次元游戏人生

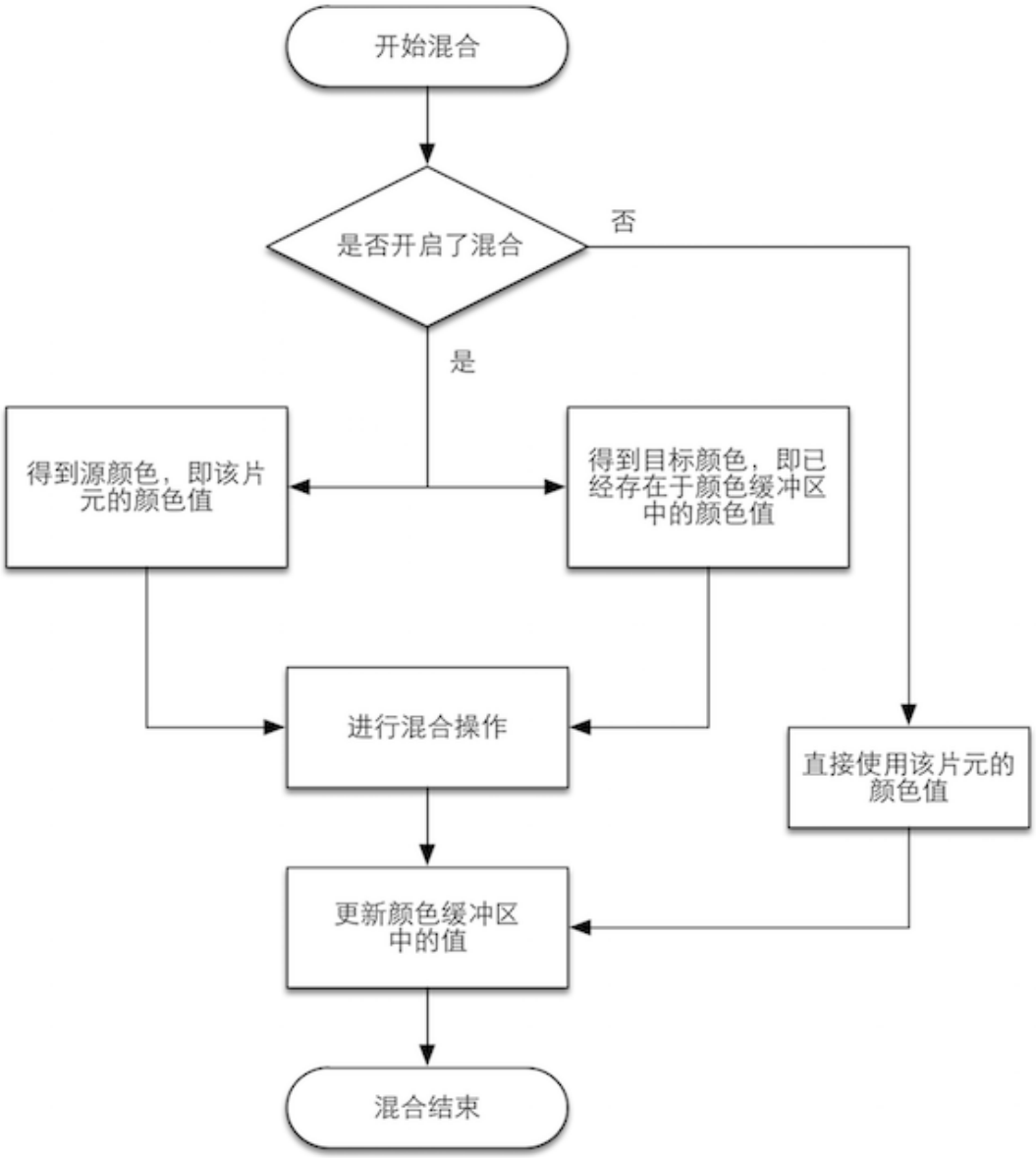


图48.　　混合操作的简化流程图

从流程图中我们可以发现，混合操作也是可以高度配置的：开发者可以选择开启/关闭混合功能。如果没有开启混合功能，就会直接使用片元的颜色覆盖掉颜色缓冲区中的颜色，而这也是很多初学者发现无法得到透明效果的原因（没有开启混合功能）。

如果开启了混合，GPU 会取出源颜色和目标颜色，将两种颜色进行混合。源颜色指的是片元着色器得到的颜色值，而目标颜色则是已经存在于颜色缓冲区中的颜色值。之后，就会使用一个混合函数来进行混合操作。这个混合函数通常和透明通道息息相关，例如根据透明通道的值进行相加、相减、相乘等。

混合很像 Photoshop 中对图层的操作：每一层图层可以选择混合模式，混合模式决定了该图层和下层图层的混合结果，而我们看到的图片就是混合后的图片。

上面给出的测试顺序并不是唯一的，而且虽然从逻辑上来说这些测试是在片元着色器之后进行的，但对于大多数 GPU 来说，它们会尽可能在执行片元着色器之前就进行这些测试。这是可以理解的，想象一下，当 GPU 在片元着色器阶段花了很大力气终于计算出片元的颜色后，却发现这个片元根本没有通过这些检验，也就是说这个片元还是被舍弃了，那

之前花费的计算成本全都浪费了！

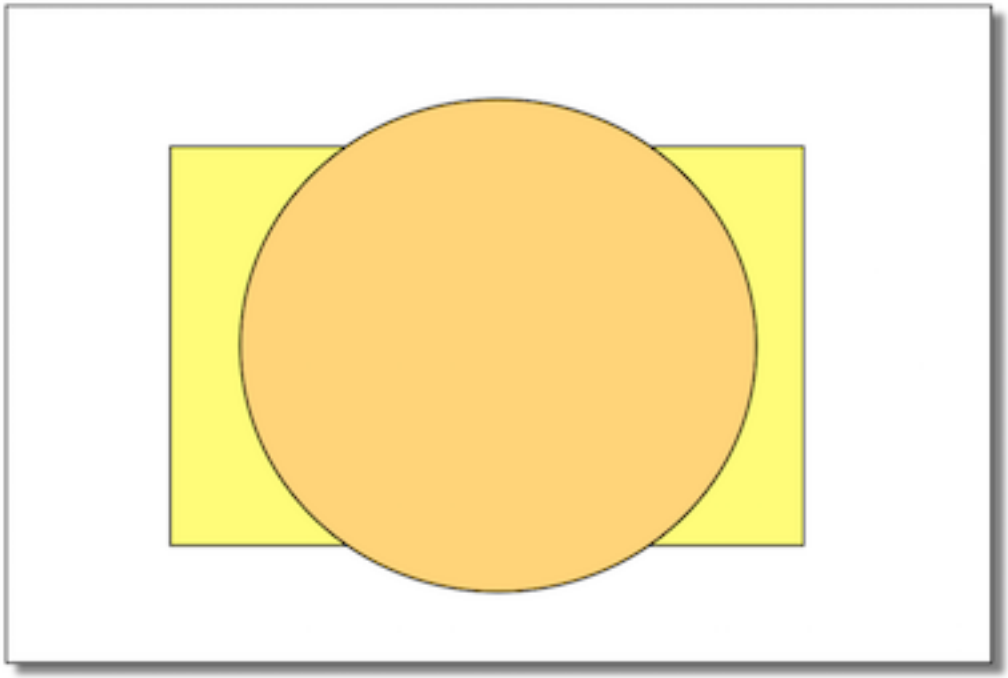


图49.　　示例

上图中包含了两个对象：球和长方体，绘制顺序是先绘制球（在屏幕上显示为圆），再绘制长方体（在屏幕上显示为长方形）。如果深度测试在片元着色器之后执行，那么在渲染长方体时，虽然它的大部分区域都被遮挡在球的后面，即它所覆盖的绝大部分片元根本无法通过深度测试，但是我们仍然需要对这些片元执行片元着色器，造成了很大的性能浪费。

作为一个想充分提高性能的 GPU，它会希望尽可能早地知道哪些片元是会被舍弃的，对于这些片元就不需要再使用片元着色器来计算它们的颜色。在 Unity 给出的渲染流水线中，我们也可以发现它给出的深度测试是在片元着色器之前。这种将深度测试提前执行的技术通常也被称为 Early-Z 技术。

但是，如果将这些测试提前的话，其检验结果可能会与片元着色器中的一些操作冲突。例如，如果我们在片元着色器进行了透明度测试，而这个片元没有通过透明度测试，我们会在着色器中调用 APl（例如 clip 函数）来手动将其舍弃掉。这就导致 GPU 无法提前执行各种测试。

因此，现代的 GPU 会判断片元着色器中的操作是否和提前测试发生冲突，如果有冲突，就会禁用提前测试。但是，这样也会造成性能上的下降，因为有更多片元需要被处理了。这也是透明度测试会导致性能下降的原因。

当模型的图元经过了上面层层计算和测试后，就会显示到我们的屏幕上。我们的屏幕显示的就是颜色缓冲区中的颜色值。但是，为了避免我们看到那些正在进行光栅化的图元，GPU 会使用双重缓冲（Double Buffering）的策略。

这意味着，对场景的渲染是在幕后发生的，即在后置缓冲（Back Buffer）中。一旦场景已经被渲染到了后置缓冲中，GPU 就会交换后置缓冲区和前置缓冲（Front Buffer）中的内容，而前置缓冲区是之前显示在屏幕上的图像。由此，保证了我们看到的图像总是连续的。

八、 什么是 Draw Call

在前面的章节中，我们已经了解了 Draw Call 的含义。Draw Call 本身的含义很简单，就是 CPU 调用图像编程接口，如 OpenGL 中的 glDrawElements 命令或者 DirectX 中的 DrawIndexedPrimitive 命令，以命令 GPU 进行渲染的操作。

一个常见的误区是，Draw Call 中造成性能问题的元凶是 GPU，认为 GPU 上的状态切换是耗时的，其实不是的，真正“拖后腿”其实的是 CPU。

在深入理解 Draw Call 之前，我们先来看一下 CPU 和 GPU 之间的流水线化是怎么实现的，即它们是如何相互独立一起工作的。

1. CPU 和 GPU 是如何实现并行工作的？

如果没有流水线化，那么 CPU 需要等到 GPU 完成上一个渲染任务才能再次发送渲染命令。但这种方法显然会造成效率低下。因此，我们需要让 CPU 和 GPU 可以并行工作。而解决方法就是使用一个命令缓冲区（Command Buffer）。

命令缓冲区包含了一个命令队列，由 CPU 向其中添加命令，而由 GPU 从中读取命令，添加和读取的过程是互相独立的。命令缓冲区使得 CPU 和 GPU 可以相互独立工作。当 CPU 需要渲染一些对象时，它可以向命令缓冲区中添加命令，而当 GPU 完成了上一次的渲染任务后，它就可以从命令队列中再取出一个命令并执行它。

命令缓冲区中的命令有很多种类，而 Draw Call 是其中一种，其他命令还有改变渲染状态等（例如改变使用的着色器，使用不同的纹理等）。

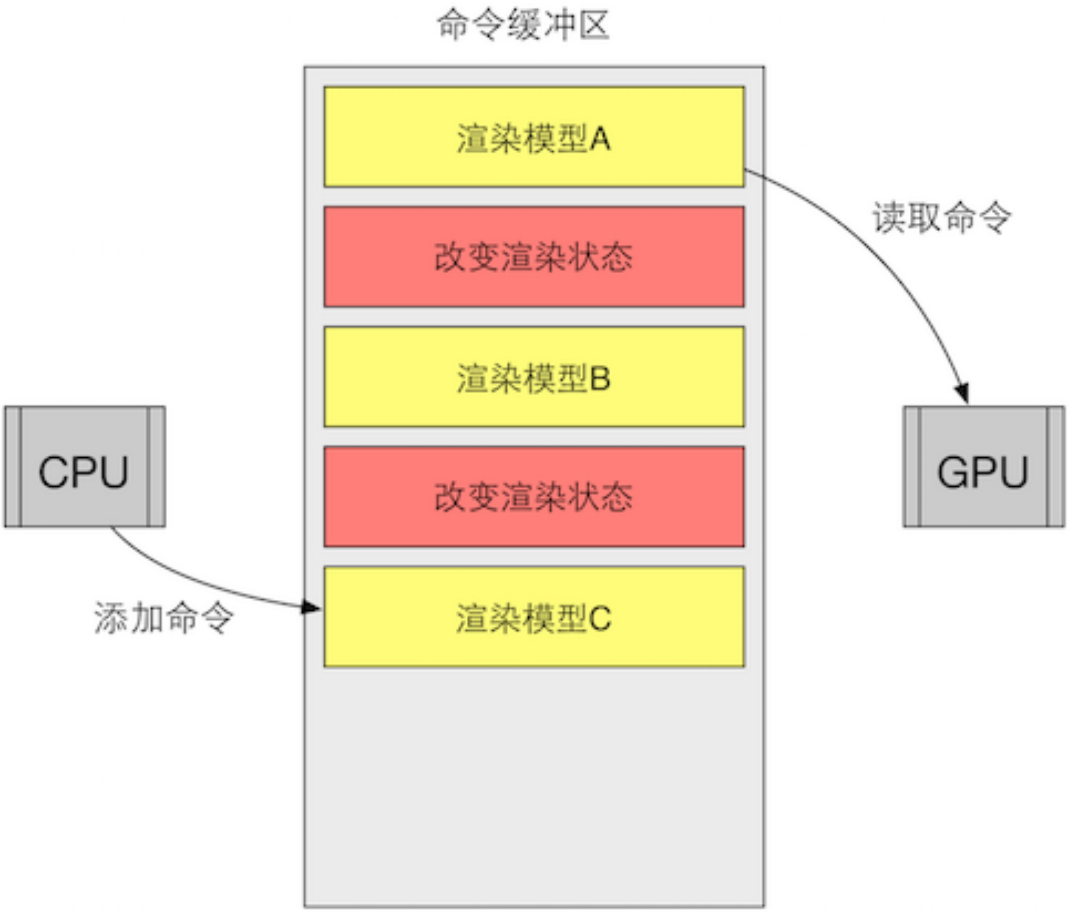


图50. 命令缓冲区

CPU 通过图像编程接口向命令缓冲区中添加命令，而 GPU 从中读取命令并执行。黄

色方框内的命令就是 Draw Call，而红色方框内的命令用于改变渲染状态。我们使用红色方框来表示改变渲染状态的命令，是因为这些命令往往更加耗时。

2. 为什么 Draw Call 多了会影响帧率？

我们先来做一个实验：创建 10000 个小文件，每个文件的大小为 1KB，然后把它们从一个文件夹复制到另一个文件夹。你会发现，尽管这些文件的空间总和不超过 10MB，但要花费很长时间。现在，我们再来创建一个单独的文件，它的大小是 10MB，然后也把它从一个文件夹复制到另一个文件夹。而这次复制的时间却少很多！这是为什么呢？明明它们所包含的内容大小是一样的。原因在于，每一个复制动作需要很多额外的操作，例如分配内存、创建各种元数据等。如你所见，这些操作将造成很多额外的性能开销，如果我们复制了很多小文件，那么这个开销将会很大。

渲染的过程虽然和上面的实验有很大不同，但从感性角度上是很类似的。在每次调用 DrawCall 之前，CPU 需要向 GPU 发送很多内容，包括数据、状态和命令等。在这一阶段，CPU 需要完成很多工作，例如检查渲染状态等。而一旦 CPU 完成了这些准备工作，GPU 就可以开始本次的渲染。GPU 的渲染能力是很强的，渲染 200 个还是 2000 个三角网格通常没有什么区别，因此渲染速度往往快于 CPU 提交命令的速度。如果 Draw Call 的数量太多，CPU 就会把大量时间花费在提交 Draw Call.上，造成 CPU 的过载。

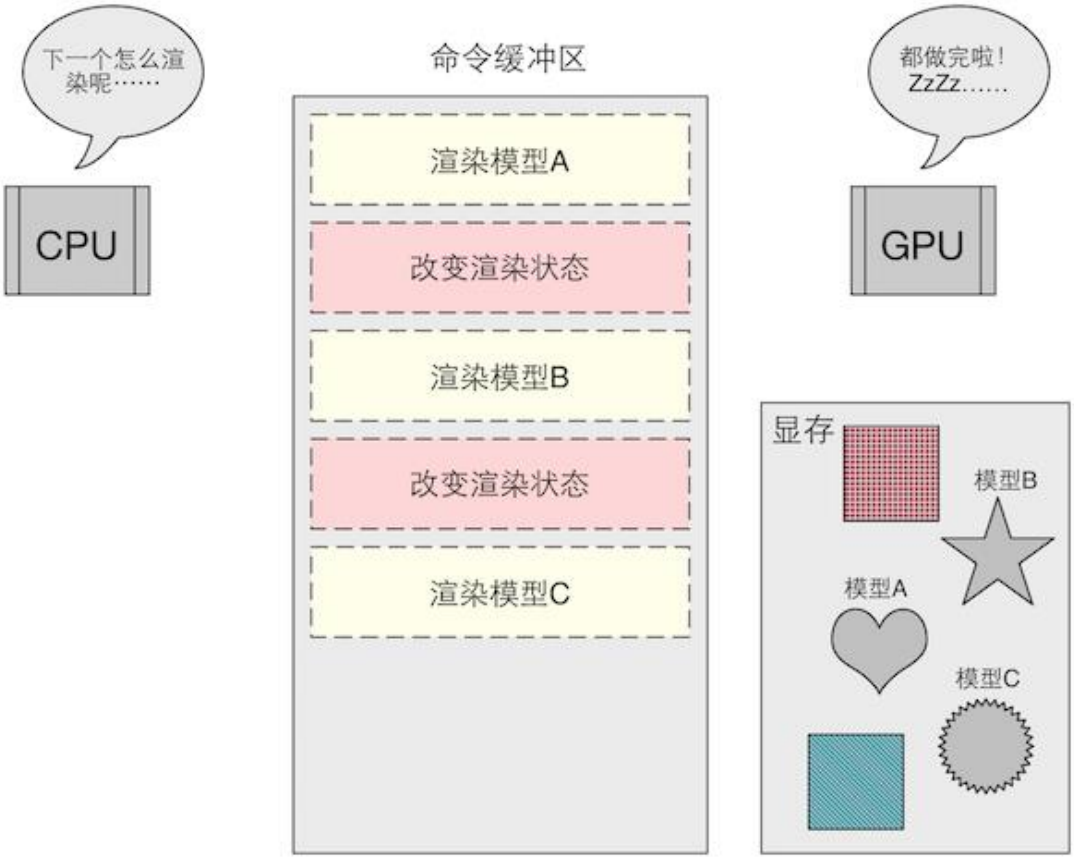


图51. 命令缓冲区中的虚线方框表示 GPU 已经完成的命令

如上图所示，此时，命令缓冲区中没有可以执行的命令了，GPU 处于空闲状态，而 CPU 还没有准备好下一个渲染命令。

3. 如何减少 Draw Call？

尽管减少 Draw Call 的方法有很多，但我们这里仅讨论使用批处理（Batching）的方法。

我们讲过，提交大量很小的 Draw Call 会造成 CPU 的性能瓶颈，即 CPU 把时间都花费在准备 Draw Call 的工作上了。那么，一个很显然的优化想法就是把很多小的 DrawCall 合并成一个大的 Draw Call，这就是批处理的思想。

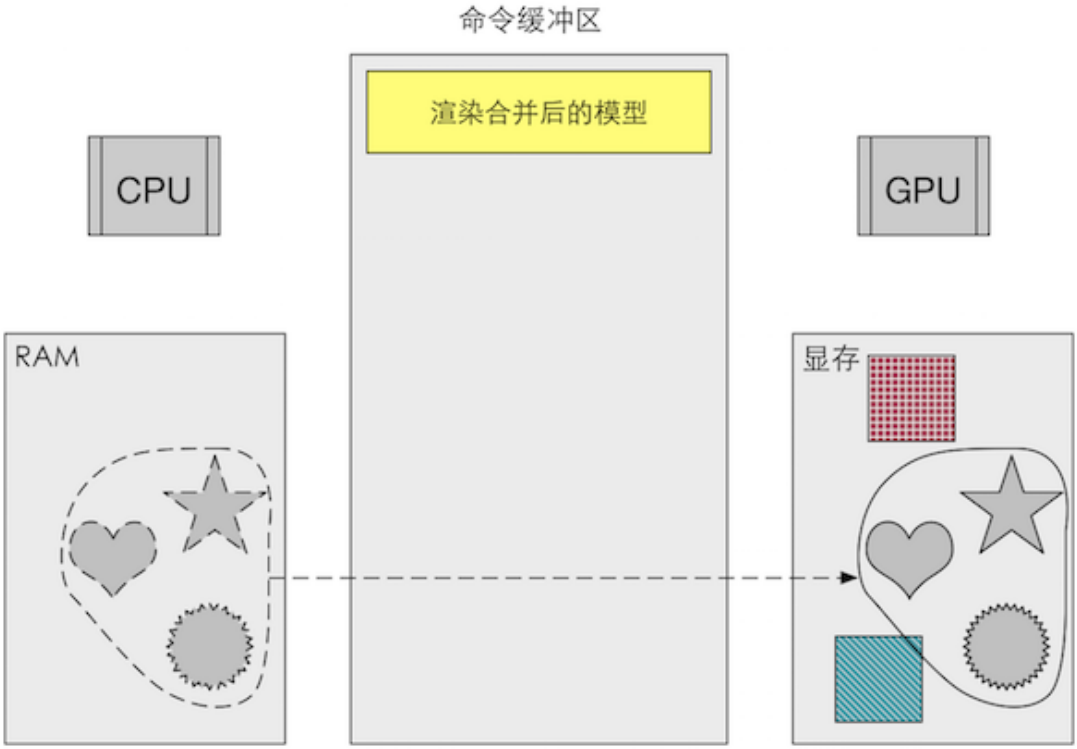


图52.　利用批处理

CPU 在 RAM 把多个网格合并成一个更大的网格，再发送给 GPU，然后在一个 Draw Call 中渲染它们。但要注意的是，使用批处理合并的网格将会使用同一种渲染状态。也就是说，如果网格之间需要使用不同的渲染状态，那么就无法使用批处理技术。

需要注意的是，由于我们需要在 CPU 的内存中合并网格，而合并的过程是需要消耗时间的。因此，批处理技术更加适合于那些静态的物体，例如不会移动的大地、石头等，对于这些静态物体我们只需要合并一次即可。当然，我们也可以对动态物体进行批处理。但是，由于这些物体是不断运动的，因此每一帧都需要重新进行合并然后再发送给 GPU，这对空间和时间都会造成一定的影响。

在游戏开发过程中，为了减少 Draw Call 的开销，有两点需要注意。

1) 避免使用大量很小的网格。当不可避免地需要很小的网格结构时，考虑是否可以合并它们。
2) 避免使用过多的材质。尽量在不同的网格之间共用同一个材质。

九、　矩阵和坐标系

十、　语法基础

在 Unity 中，所有的 Shader 程序都是使用名为“ShaderLab”的声明性语言进行编写的，在 Shader 文件中， ShaderLab 通过嵌套花括号的方式声明着色器的各部分内容。例如:哪些属性需要在材质面板上显示;使用哪种混合模式进行颜色混合;如果运行失败，回退到哪个着色器等，ShaderLab 语法不区分大小写。而真正意义上的 Shader 代码则是在 CGPROGRAM 代码块中编写的。

1. ShaderLab 的组织结构

无论选择何种语言编写 Shader,无论编写的 Shader 是何种类型、UnityShader 总是通过 ShaderLab 语言进行包装并组织结构。通常情况下，Shader 的大致结构如下所示:

```
Shader"Name"
{
    Propertien
    {
        //开放到材质面板的属性
    }
    SubShader
    {
    //顶点-片段着色器
    //或者表面着色器
    //或者固定函数着色器

        Pass
        {
            CGPROGRAM
            // CG 或者 HLSL 程序块
            ENDCG
        }
    }
    SubShader
    {
    //更加精简的版本，为了在旧的图形设备上运行
    }
    Fallback"ExampleOtherShader"
```

着色器结构在 Cg 和 HLSL 中是相同的，唯一改变的是 Cg 和 HLSL 中的程序块。两者都在当前版本的 Unity 中编译以实现兼容性。

着色器以 Inspector 中的路径(InspectorPath) 和名称(shaderName) 开始，然后是属性（例如纹理、矢量、颜色等），然后是 SubShader，最后是可选的 Fallback。

“inspectorPath”是指我们将选择着色器以将其应用于材质的位置。此选择是通过 Unity Inspector 进行的。

我们必须记住，我们不能将着色器直接应用于多边形对象，而是必须通过先前创建的材质来完成。

我们必须考虑的一个结构因素是 GPU 将以线性方式从上到下读取程序，因此，如果我们创建一个函数并将其放置在将要使用它的代码块下方，GPU 将无法读取它，在着色器处理中产生错误，因此 Fallback 将分配不同的着色器，以便图形硬件可以继续其处理。

```
// 1 . declare our function
float4 ourFunction()
{
    // your operation here ...
}
```

哔哩哔哩次元游戏人生

```
// 2. use our function
fixed4 frag (v2f i) : SV_Target
{
    // function being used here
    float4 f = ourFunction();
    return f;
}
```

如上面的代码，函数“ ourFunction ”已经写在代码块所在位置之前。GPU 将首先读取函数“ ourFunction ”，然后继续进入名为“ frag ”的片段阶段。（这可以正确运行）

而如果我们交换它们的顺序，那么将会产生错误。因为名为“ OurFunction ”的函数已经写在正在使用它的代码块下面。

```
// 2. use our function
fixed4 frag (v2f i) : SV_Target
{
    // function being used here
    float4 f = ourFunction();
    return f;
}

// 1 . declare our function
float4 ourFunction()
{
    // your operation here ...
}
```

1） 属性

“ PropertyName ”是指属性的名称（例如_MainTex），“ display name ”对应于将在 Unity 检查器中显示的属性的名称（例如 Texture），“ type ”表示属性类型（例如 Color，Vector、2D 等），最后，顾名思义，“ defaultValue ”是分配给属性的默认值（例如，如果属性是“Color”，我们可以将其设置为白色）

```
Properties
{
    _MainTex (“Texture”, 2D) = “white” {}
}
```

需要考虑的一个因素是，当我们声明一个属性时，它在属性域内保持“开放”状态，因此我们必须避免在代码行末尾使用分号 (;)，否则 GPU 将无法阅读程序。

属性类型	默认值的定义语法	例子
Int	number	_Int("Int",int)=2
Float	number	_Float("Float",Float)=1.5
Range(min,max)	number	_Range("Range ", Range（0.0,5.0）)=3.0
Color	(number, number, number, number)	_ Color ("Color ", Color)=(1,1,1,1)
Vector	(number, number, number, number)	_ Vector ("Vector ", Vector)=(2,3,6,1)

哔哩哔哩次元游戏人生

2D	"defaulttexture"{}	_2D ("2D ",2D)=""{}
Cube	"defaulttexture"{}	_ Cube ("Cube ", Cube)= "white" {}
3D	"defaulttexture"{}	_3D ("3D ",3D)= "black" {}

对于 Int、Float、Range 这些数字类型的属性，其默认值就是一个单独的数字；对于 Color 和 Vector 这类属性，默认值是用圆括号包围的一个四维向量；对于 2D、Cube、3D 这三种纹理类型，默认值的定义稍微复杂，它们的默认值是通过一个字符串后跟一个花括号来指定的，其中，字符串要么是空的，要么是内置的纹理名称，如“white”“black”“gray”或者“bump”。

为了在 Shader 中可以访问到这些属性，我们需要在 Cg 代码片中定义和这些属性类型相匹配的变量。需要说明的是，即使我们不在 Properties 语义块中声明这些属性，也可以直接在 Cg 代码片中定义变量。此时，我们可以通过脚本向 Shader 中传递这些属性。因此，Properties 语义块的作用仅仅是为了让这些属性可以出现在材质面板中。

b. 数字和滑块属性

```
// name (“display name”, Range(min, max)) = defaultValue
// name (“display name”, Float) = defaultValue
// name (“display name”, Int) = defaultValue

Shader “InspectorPath / shaderName”
{
    Properties
    {
        _Specular (“Specular”, Range(0.0, 1.1)) = 0.3
        _Factor (“Color Factor”, Float) = 0.3
        _Cid (“Color id”, Int) = 2
    }
}
```

上面的例子，我们声明了三个属性，一个名为 _Specular 的“浮动范围”类型，另一个名为_Factor 的“浮动比例”类型，最后一个名为 _Cid 的“整数”类型。

c. 颜色和向量属性

```
// name ("display name", Color) = (R, G, B, A
// name ("display name", Vector) = (0, 0, 0, 1)

Shader "InspectorPath / shaderName"
{
    Properties
    {
        _Color ("Tint", Color) = (1, 1, 1, 1)
        _VPos ("Vertex Position", Vector) = (0, 0, 0, 1)
```

哔哩哔哩次元游戏人生

```
        }
    }
```

上面的例子，我们声明了两个属性，一个名为 _Color 的“颜色”和一个名为_VPos 的“矢量” 。

d. 纹理属性

这些属性允许我们在着色器中实现纹理。

如果我们想在我们的对象（例如一个 3D 角色）上放置一个纹理，那么我们必须为其纹理创建一个 2D 属性，然后将其传递给一个名为“tex2D”的函数，该函数将要求我们提供两个参数：纹理以及我们对象的 UV 坐标。

我们将在视频游戏中经常使用的属性是“Cube”，它本身就是指“Cubemap”。这种类型的纹理对于生成反射贴图非常有用，例如，我们角色的盔甲或一般金属元素的反射。

```
// name ("display name", 2D) = "defaultColorTexture"
// name ("display name", Cube) = "defaultColorTexture"
// name ("display name", 3D) = "defaultColorTexture"

Shader "InspectorPath / shaderName"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white"{}
        _Reflection ("Reflection", Cube) = "black" {}
        _3DTexture ("3D Texture", 3D) = "white" {}
    }
}
```

在声明一个属性时，考虑到它将用 ShaderLab 声明性语言编写，而我们的程序将用 Cg 或 HLSL 语言编写，这一点非常重要。由于它们是两种不同的语言，我们必须创建“连接变量”。

这些变量是使用“uniform”一词全局声明的，但是，可以跳过此步骤，因为程序将它们识别为全局变量。所以，要给“.shader”添加一个属性，我们必须先在 ShaderLab 中声明该属性，然后在 Cg 或 HLSL 中使用同名的全局变量，最后才能使用它。

```
Shader "InspectorPath / shaderName"
{
    Properties
    {
    // declare properties
    _MainTex ("Texture", 2D) = "white" {}
    _Color ("Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Pass
        {
```

哔哩哔哩次元游戏人生

```
                CGPROGRAM
                ...
                // add connection variables
                sampler2D _Maintex;
                float4 _Color;
                ...
                half4 frag (v2f i) : SV_Target
                {
                    // use
                    half4 col = tex2D(_MainTex, i.uv);
                    return col * _Color;
                }
                ENDCG
            }
        }
    }
```

在上面的示例中，我们声明了两个属性：_MainTex 和_Color。然后我们在 CGPROGRAM 中创建了两个连接变量，它们对应于“ sampler2D _MainTex ”和“ float4 _Color ”。属性和连接变量必须具有相同的名称，以便程序可以识别它们。

2） Material Property Drawer

我们可以在 ShaderLab 中找到的另一种属性是“Drawer”。它们允许我们在 Unity Inspector 中生成自定义属性，从而促进着色器中条件的编程。

默认情况下，我们的着色器中不包含这种类型的属性，相反，我们必须根据需要声明它们。迄今为止，共有七个不同的 Drawer：

Toggle、Enum、KeywordEnum、PowerSlider、IntRange、Space、Header。

它们中的每一个都有一个特定的功能，并且是独立声明的。

由于这些属性的存在，我们就可以在我们的程序中生成多个状态，以此来创建动态效果而无需在执行时更改材质。我们通常将这些 Drawer 与两种类型的着色器变体一起使用，它们指的是#pragma multi_compile 和#pragma shader_feature。

e. 切换-Toggle

在 ShaderLab 中，我们不能使用布尔类型的属性，相反，我们有实现相同功能的 Toggle。这个 Drawer 将允许使用我们着色器中的条件从一种状态切换到另一种状态。要运行它，我们必须首先在括号之间添加单词 Toggle，然后声明我们的属性，考虑到它必须是 Float 类型。

它的默认值必须是一个整数，要么是零，要么是一，为什么？因为 0 代表“关闭”，而 1 代表“开启”。

[Toggle] _PropertieName (“Display Name”, Float) = 0

如我们所见：我们在括号中添加 Toggle，然后声明属性，然后是显示名称，然后是 Float 数据类型，最后我们将属性初始化为“Off”，因为我们在其默认值中添加了一个零。

使用这个 Drawer 时我们必须考虑的一点是，如果我们想在我们的代码中实现它，我们将不得不使用 #pragma shader_feature。这属于着色器变体，其功能是根据其所处的状态（启用或禁用）生成不同的条件。为了理解它的实现，我们将执行以下操作：

哔哩哔哩次元游戏人生

```
Shader "InspectorPath / shaderName"
    {
    Properties
    {
        _Color ("Color", Color) = (1, 1, 1, 1)
        // declare Toggle drawer
        [Toggle] _Enable ("Enable ? ", Float) = 0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // declare pragma
            #pragma shader_feature _ENABLE_ON
            ...
            float4 _Color;
            ...
            half4 frag (v2f i) : SV_Target
            {
                half4 col = tex2D(_MainTex, i.uv);

                // generate condition
                #if _ENABLE_ON
                    return col;
                #else
                    return col * _Color;
                #endif
            }
            ENDCG
        }
    }
}
```

在此示例中，我们声明了一个名为“ _Enable ”的切换类型属性。然后我们将它添加到 CGPROGRAM 中的 shader_feature 中，然而，与我们程序中的属性不同，Toggle 被声明为“ _ENABLE_ON ”，这是为什么呢？在 shader_feature 添加的变量是“常量”，因此它们用大写字母书写，这意味着，例如，如果我们的属性被称为 _Change，那么在着色器变量中它应该被添加为“_CHANGE”。_ON 一词对应于 Toggle 的默认状态，因此，如果 _Enable 属性处于活动状态，我们在片段着色器阶段返回默认纹理颜色，否则我们将 _Color 属性与自身相乘。

值得一提的是，shader_feature 不能为一个应用程序编译多个变体，这是什么意思？Unity 不会包含我们在最终构建中未使用的变体，这意味着我们将无法在执行时从一种状态移动到另一种状态。为此，我们将不得不使用具有变体着色器“ multi_compile ”的

哔哩哔哩次元游戏人生

KeywordEnum Drawer。

f. 关键字枚举-KeywordEnum

这个 Drawer 在材质检查器中生成一个弹出式菜单。与 Toggle 不同，此 Drawer 允许您为着色器配置多达九种不同的状态。要执行它，我们必须在括号中添加单词“KeywordEnum”，然后列出我们将要使用的状态集。

```
[KeywordEnum(State_Off, State_01, etc...)]
    _PropertieName (“Display name”, Float) = 0
```

在上面的示例中，我们在括号中添加了 KeywordEnum Drawer，然后我们列出了它的状态，其中第一个对应于默认状态（StateOff）。我们继续声明属性，在材质检查器中显示名称，它的 Float 数据类型，最后，我们使用它的默认值进行初始化。

要在我们的代码中声明这个 Drawer，我们可以使用着色器变体 shader_feature 和 multi_compile。具体选择哪一个，取决于我们希望在最终构建中包含的变体数量。

正如我们已经知道的，shader_feature 只会从材质检查器中导出选定的变体，而 multi_compile 会导出在着色器中找到的所有变体，无论它们是否被使用。鉴于此功能，multi_compile 非常适合导出或编译将在执行时更改的多个状态。

```
Shader "InspectorPath / shaderName"
{
    Properties
    {
    // declare Toggle drawer
    [KeywordEnum(Off, Red, Blue)]
        _Options ("Color Options", Float) = 0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // declare pragma and conditions
            #pragma multi_compile _OPTIONS_OFF _OPTIONS_RED
_OPTIONS_BLUE
            ...
            half4 frag (v2f i) : SV_Target
            {
                half4 col = tex2D(_MainTex, i.uv);

                // generate conditions
                #if _OPTIONS_OFF
                    return col;
                #elif _OPTIONS_RED
                    return col * float4(1, 0, 0, 1);
```

```
                #elif _OPTIONS_BLUE
                    return col * float4(0, 0, 1, 1);
                #endif
            }
            ENDCG
        }
    }
}
```

在上面的示例中，我们声明了一个名为“ _Options ”的 KeywordEnum 类型属性，并为其配置了三种状态（Off、Red 和 Blue）。稍后我们将它们添加到 CGPROGRAM 中的 multi_compile 并将它们声明为常量。

```
#pragma multi_compile _OPTIONS_OFF _OPTIONS_RED  _OPTIONS_BLUE
```

最后，使用条件，我们为着色器定义了三种状态，它们对应于主纹理的颜色变化。

g. 枚举-Enum

此 Drawer 与 KeywordEnum 非常相似，不同之处在于它可以将“值/id”定义为参数，并将此属性传递给着色器中的命令，以从检查器动态更改其功能。

它的语法如下：

```
[Enum(valor, id_00, valor, id_01, etc ...)]
    _PropertyName (“Display Name”, Float) = 0
```

枚举不使用着色器变体，而是由命令或函数声明。为了理解它们的实现，我们将执行以下操作：

```
Shader "InspectorPath / shaderName"
{
    Properties
    {
    // declare drawer
    [Enum(Off, 0, Front, 1, Back, 2)]
        _Face ("Face Culling", Float) = 0
    }
    SubShader
    {
        // pass the property to the command
        Cull [_Face]
        Pass {} ...
    }
}
```

如上面所示，我们声明了一个名为“_Face”的属性类型“Enum”，并将值作为参数传递：Off,0,Front,1,Back , 2。然后我们将属性添加到 SubShader 的命令“Cull”；通过这种方式，我们可以从材质检查器中更改我们想要渲染的对象面。

哔哩哔哩次元游戏人生

h. PowerSlider and IntRange

这些 Drawer 在处理数值范围和精度时非常有用。一方面，我们有 PowerSlider，它允许我们生成带有曲线控制的非线性滑块。

[PowerSlider(3.0)] _PropertyName ("Display name", Range (0.01, 1)) = 0.08

另一方面，我们有 IntRange，顾名思义，它添加了整数值的数值范围。

[IntRange] _PropertyName ("Display name", Range (0, 255)) = 100

请注意，如果我们想在着色器中使用这些属性，则必须在 CGPROGRAM 中以与常规属性相同的方式声明它们。要了解如何使用它，我们将执行以下操作：

```
Shader "InspectorPath / shaderName"
{
    Properties
    {
        // declare drawer
        [PowerSlider(3.0)]
            _Brightness ("Brightness", Range (0.01, 1)) = 0.08
        [IntRange]
            _Samples ("Samples", Range (0, 255)) = 100
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // generate connection variables
            float _Brightness;
            int _Samples;
            ...
            ENDCG
        }
    }
}
```

在上面的示例中，我们声明了一个名为 _Brightness 的 PowerSlider 和一个名为 _Samples 的 IntRange。最后，使用相同的名称，我们在 CGPROGRAM 中生成连接变量。

i. Space and Header

最后，这些 Drawer 对组织任务有很大帮助。

“Space”允许我们在一个属性和另一个属性之间添加空格。如果我们希望我们的属性在材料检查器中单独显示，我们可以在它们之间添加空格。

_PropertyName01 ("Display name", Float) = 0

哔哩哔哩次元游戏人生

```
// we add the space
[Space(10)]
_PropertyName02 ("Display name", Float ) = 0
```

在这种情况下，我们在属性_PropertyName01 和属性_PropertyName02 之间添加了十个空间点。

“Header”在 Unity Inspector 中添加了一个标题。这在我们的属性中生成类别时非常有用。

```
// we add the header
[Header(Category name)]
_PropertyName01 ("Display name", Float ) = 0
_PropertyName02 ("Display name", Float ) = 0
```

在上面的示例中，我们在启动属性之前添加了一个小标题，它将在 Inspector 中可见。为了理解这两个属性，我们将执行以下操作：

```
Shader “InspectorPath / shaderName”
{
    Properties
    {
        [Header(Specular properties)]
        _Specularity ("Speculatrity", Range (0.01, 1)) = 0.08
        _Brightness ("Brightness", Range (0.01, 1)) = 0.08
        _SpecularColor ("Specular Color", Color) = (1, 1, 1 , 1)
        [Space(20)]
        [Header(Texture properties)]
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader {} ...
}
```

3） 子着色器-SubShader

着色器的第二个组件是 SubShader。每个着色器至少由一个 SubShader 组成，以完美加载程序。当有多个 SubShader 时，Unity 会分别处理，并根据硬件特性，从列表中的第一个到最后一个开始。为了理解这一点，我们假设着色器将在支持金属图形 API (iOS) 的硬件上运行。为此，Unity 将运行第一个支持金属图的 SubShader 并运行它。当不支持 SubShader 时，Unity 将尝试使用与默认着色器对应的 Fallback 组件，因此硬件可以继续其任务而不会出现图形错误。

Unity 提供这种语义的原因在于，不同的显卡具有不同的能力。例如，一些旧的显卡仅能支持一定数目的操作指令，而一些更高级的显卡可以支持更多的指令数，那么我们希望在旧的显卡上使用计算复杂度较低的着色器，而在高级的显卡上使用计算复杂度较高的着色器，以便提供更出色的画面。

```
SubShader
{
    //可选的
    [Tags]
    //可选的
    [RenderSetup]
```

哔哩哔哩次元游戏人生

```
        pass
        {
        }
    }
```

SubShader 中定义了一系列 Pass 以及可选的状态([RenderSetup])和标签([Tags])设置。每个 Pass 定义了一次完整的渲染流程，但如果 Pass 的数目过多，往往会造成渲染性能的下降。因此，我们应尽量使用最小数目的 Pass。状态和标签同样可以在 Pass 声明。不同的是，SubShader 中的一些标签设置是特定的。也就是说，这些标签设置和 Pass 中使用的标签是不一样的。而对于状态设置来说，其使用的语法是相同的。但是，如果我们在 SubShader 进行了这些设置，那么将会用于所有的 Pass。

a. 状态设置

ShaderLab 提供了一系列渲染状态的设置指令，这些指令可以设置显卡的各种状态，例如是否开启混合/深度测试等。

状态名称	设置指令	解释
Cull	Cull Back|Front|Off	设置剔除模式；剔除背面/正面/关闭剔除
ZTest	ZTest Less Greater|LEqual|GEqual|Equal|NotEqual|Always	设置深度测试时使用的函数
ZWrite	ZWrite On|Off	开启/关闭深度写入
Blend	Blend SrcFactor DstFactor	开启并设置混合模式

当在 SubShader 块中设置上述渲染状态时，将会应用到所有的 Pass。如果我们不想这样（例如在双面渲染中，我们希望在第一个 Pass 中剔除正面来对背面进行渲染，在第二个 Pass 中剔除背面来对正面进行渲染），可以在 Pass 语义块中单独进行上面的设置。

b. SubShader 的标签

标签是显示我们的着色器处理方式和时间的标签。 与 GameObject 标签一样，这些标签可用于识别着色器的渲染方式或一组着色器的图形行为。

SubShader 的标签(Tags)是一个键值对(Key/Value Pair)，它的键和值都是字符串类型。这些键值对是 SubShader 和渲染队列之间的沟通桥梁。它们用来告诉 Unity 的渲染引擎：我希望怎样以及何时渲染这个对象。

```
Tags
{
    "TagName1" = "TagValue1"
    "TagName2" = "TagValue2"
}
```

标签可以写在两个不同的字段中，要么在 SubShader 内，要么在 Pass 内。这一切都取决于我们想要获得的结果。如果我们在 SubShader 中写一个标签，它将影响着色器中包含的所有通道，但如果我们将它写在 Pass 中，它只会影响选定的通道。

哔哩哔哩次元游戏人生

“Queue”是我们更频繁使用的标签，因为它允许我们定义对象表面的外观。默认情况下，所有表面都定义为“opaque”，即它们没有透明度。

标签类型	说明	例子
Queue	控制渲染顺序，指定该物体属于哪一个渲染队列，通过这种方式可以保证所有的透明物体可以在所有不透明物体后面被渲染，我们也可以自定义使用的渲染队列来控制物体的渲染顺序	Tags{"Queue"="Transparent"}
RenderType	对着色器进行分类，例如这是一个不透明的着色器，或是一个透明的着色器等。这可以被用于着色器替换（Shader Replacement）功能	Tags{"RenderType "="Opaque"}
DisableBatching	一些 SubShader 在使用 Unity 的批处理功能时会出现问题，例如使用了模型空间下的坐标进行顶点动画。这时可以通过该标签来直接指明是否对该 SubShader 使用批处理	Tags{"DisableBatching"="true"}
ForceNoShadowCasting	控制使用该 SubShader 的物体是否会投射阴影	Tags{"ForceNoShadowCasting"="true"}
IgnoreProjector	如果该标签值为“True”，那么使用该 SubShader 的物体将不会受 Projector 的影响。通常用于半透明物体。	Tags{"IgnoreProjector"="true"}
CanUseSpriteAtlas	当该 SubShader 是用于精灵（sprites）时，将该标签设为“False”。	Tags{"CanUseSpriteAtlas"="False"}
PreviewType	指明材质面板将如何预览该材质。默认情况下，材质将显示为	Tags{"PreviewType"="Plane"}

哔哩哔哩沐元游戏人生

	一个球形，我们可以通过把该标签的值设为“Plane′”“SkyBox”来改变预览类型	

c. Pass 语义块

Pass 语义块包含的语义如下：

```
Pass
{
    [Name]
    [Tags]
    [RenderSetup]
    //Other code
}
```

首先，我们可以在 Pass 中定义该 Pass 的名称，例如：

Name "MyPassName"

通过这个名称，我们可以使用 ShaderLab 的 UsePass 命令来直接使用其他 Unity Shader 中的 Pass。

例如：UsePass "MyShader/MYPASSNAME"

这样可以提高代码的复用性。需要注意的是，由于 Unity 内部会把所有 Pass 的名称转换成大写字母的表示，因此，在使用 UsePass 命令时必须使用大写形式的名字。

其次，我们可以对 Pass 设置渲染状态。SubShader 的状态设置同样适用于 Pass。除了上面提到的状态设置外，在 Pass 中我们还可以使用固定管线的着色器命令。

Pass 同样可以设置标签，但它的标签不同于 SubShader 的标签。这些标签也是用于告诉渲染引擎我们希望怎样来渲染该物体。

标签类型	说明	例子
LightMode	定义该 Pass 在 Unity 的渲染流水线中的角色	Tags{"LightMode"="ForwardBase"}
RequireOptions	用于指定当满足某些条件时才渲染该 Pass，它的值是一个由空格分隔的字符串。目前，Unity 支持的选项有:SoftVegetation。在后面的版本中，可能会增加更多的选项	Tags{"RequireOptions"="SoftVegetation"}

除了上面普通的 Pass 定义之外，Unity Shader 还支持一些特殊的 Pass，以便进行代码复用或实现更复杂的效果。

UsePass：如我们之前提到的一样，可以使用该命令来复用其他 Unity Shader 中的 Pass。

GrabPass：该 Pass 负责抓取屏幕并将结果存储在一张纹理中，以用于后续的 Pass 处理。

哔哩哔哩次元游戏人生

d. Fallback

紧跟在各个 SubShader 语义后面的，可以是一个 Fallback 指令。它用于告诉 Unity，“如果上面所有的 SubShader 在这块显卡上都不能运行，那么就使用这个最低级的 Shader 吧！”

它的语义如下：

FallBack "name"

//或者

FallBack Off

如上所述，我们可以通过一个字符串来告诉 Unity 这个“最低级的 Unity Shader”是谁。我们也可以任性地关闭 Fallback 功能，但一旦你这么做，你的意思大概就是：“如果一块显卡跑不了上面所有的 SubShader，那就不要管它了！”

事实上，Fallback 还会影响阴影的投射。在渲染阴影纹理时，Unity 会在每个 Unity Shader 中寻找一个阴影投射的 Pass。通常情况下，我们不需要自己专门实现一个 Pass，这是因为 Fallback 使用的内置 Shader 中包含了这样一个通用的 Pass。因此，为每个 Unity Shader 正确设置 Fallback 是非常重要的。

2. 详细解释

1） 队列标签-Queue Tags

默认情况下，此标记不会以图形方式显示为一行代码。这是因为它在 GPU 中默认编译，因为它与每种材质的对象处理顺序直接相关。

Tags { "Queue" = "Geometry" }

这个标签与相机和 GPU 有着密切的关系。

每次我们在场景中定位一个对象时，我们都会将它的信息传递给 GPU（例如顶点的位置、法线、颜色等）。在 Game View 案例中，不同之处在于我们发送到 GPU 的信息对应于相机截锥体内的对象。一旦信息进入 GPU，我们就会将此数据发送到 VRAM，并要求它在屏幕上绘制对象。

绘制对象的过程称为“draw call”。着色器的通道越多，渲染中的绘制调用就越多。一个 pass 相当于一个 draw call，因此，如果我们有一个里面有两个 pass 的着色器，那么这个材质只会在 GPU 上生成两个 draw call 。

现在，GPU 是如何在屏幕上绘制这些元素的呢？简而言之，GPU 将首先绘制距离相机最远的对象，最后绘制最接近它的元素。此计算将根据对象与相机之间的距离，按照其“Z”轴进行。

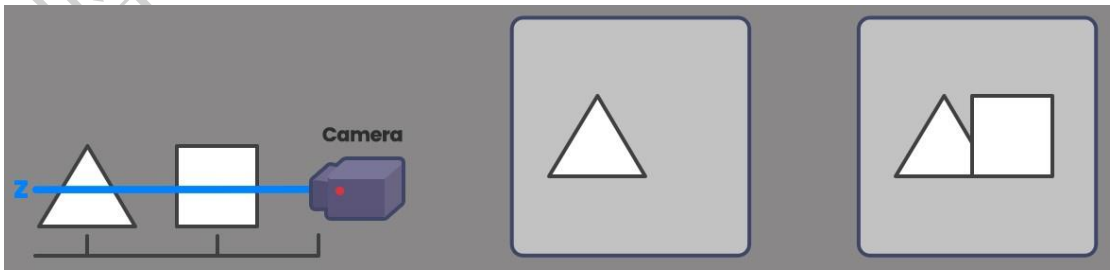


图53. GUP 绘制演示

从图中我们可以看到，由于距离相机最远，所以首先在屏幕上绘制三角形，最后绘制正方形，并一起生成 2 个 draw call。

Unity 有一个称为“渲染队列(Render Queue)”的处理队列，它允许我们修改 GPU 上

对象的处理顺序。有两种方法可以修改渲染队列：

第一种是通过检查器中的材质属性，第二种是使用标签(Tag)“Queue”。

如果我们修改 shader 中的 Queue 值，材质中 Render Queue 的默认值也会被修改。

这个属性有从 0 到 5000 的顺序值，其中“0”对应最远的元素，“5000”对应离相机最近的元素。有些命令值有预定义的组，它们是：

Background（背景）、Geometry（几何学）、AlphaTest（阿尔法测试）、Transparent（透明）、Overlay（叠加）。

```
//Goes from 0 to 1499, default value 1000.
Tags {"Queue" = "Background"}
//Goes from 1500 to 2399, default value 2000.
Tags {"Queue" = "Geometry"}
//Goes from 2400 to 2699, default value 2450.
Tags {"Queue" = "AlphaTest"}
//Goes from 2700 to 3599, default value 3000.
Tags {"Queue" = "Transparent"}
//Goes from 3600 to 5000, default value 4000.
Tags {"Queue" = "Overlay"}
```

Background 主要用于离相机很远的元素（例如天空盒）。

Geometry 是队列中的默认值，用于场景中的不透明对象（例如图元和一般对象）。

AlphaTest 用于半透明对象，这些对象必须在不透明对象前面，但在透明对象后面（例如玻璃、草或植被）。

Transparent 用于必须在其他元素前面的透明元素。

最后，Overlay 对应于场景中最重要的元素（例如 UI 图像）。

```
Shader "InspectorPath / shaderName"
{
    Properties {} …
    SubShader
    {
        Tags { "Queue" = "Geometry" }
    }
}
```

High Definition RP 使用 Render Queue 的方式与 Built-in RP 不同，因为材质不会直接在 Inspector 界面中显示此属性，而是引入了两种控制方法，它们是：

Material order（材质顺序）和 And render order（并渲染顺序）。

HDRP 一起使用这两种排序方法来控制对象处理。

2） 渲染类型标签-Render Type Tags

Unity 的官方文档中说，可“使用 RenderType 标签覆盖着色器的行为”。

上面的说法是什么意思？基本上，使用这个标签，我们可以在 SubShader 中从一种状态更改为另一种状态，从而在与给定类型匹配的任何材质上添加效果。

为了实现它的功能，我们至少需要两个着色器：

第一个替代品（我们要在运行时添加的颜色或效果）。

另一个要替换的（分配给材质的着色器）。

它的语法：Tags { "RenderType"="type" }

哔哩哔哩次元游戏人生

与队列标签一样，RenderType 具有不同的可配置值，这些值取决于我们要执行的任务。

Opaque-不透明（这是默认的）、Transparent-透明、TransparentCutout-透明剪裁、Background-背景、Overlay-覆盖、TreeOpaque-不透明树、TreeTransparentCutout-树透明切口、TreeBillboard-树广告牌、Grass-草、GrassBillboard-草广告牌。

默认情况下，每次创建新着色器时都会设置“不透明”类型。同样，Unity 中的大多数内置着色器都分配有此值，因为它们没有透明度配置。但是，我们可以自由更改此类别；一切都取决于我们应用于匹配的效果。

为了彻底理解这个概念，我们将执行以下操作。在我们的项目中，

① 我们将在场景中创建一些 3D 对象。

② 创建一个名为 USBReplacementController 的 C#脚本。

③ 然后我们将创建一个名为 USB_replacement_shader 的着色器。

④ 最后，我们将创建一个名为 USB_replaced_mat 的材质。

我们将使用 Camera.SetReplacementShader 在材质 USB_replaced_mat 上动态分配着色器。材质着色器必须具有与替换着色器相同的标记渲染类型才能执行该功能。

为了举例说明，我们将 Mobile/Unlit 着色器分配给 USB_replaced_mat。这个内置着色器有一个类型为“RenderType”的标记等于“不透明”。因此，着色器 USB_replacement_shader 必须与要执行的操作匹配相同的 RenderType。

USBReplacementController 脚本必须作为组件直接分配给相机。该控制器将负责用替换着色器替换着色器，只要它们在 RenderType 中具有相同的配置。

```
[ExecuteInEditMode]
public class USBReplacementController : MonoBehaviour
{
    public Shader m_replacementShader;

    private void OnEnable()
    {
        if (m_replacementShader != null)
        {
            // // 相机将替换场景中的所有着色器
            //其 "RenderType"需保持一致
            GetComponent<Camera>().SetReplacementShader(m_replacementShader,
"RenderType");
        }
    }

    private void OnDisable()
    {
        // 重置默认着色器
        GetComponent<Camera>().ResetReplacementShader();
    }
}
```

值得一提的是，我们在类上定义了 [ExecuteInEditMode] 函数。此属性将允许我们在

编辑模式下预览更改。

我们将使用 USB_replacement_shader 作为替换着色器。

正如我们已经知道的，每次我们创建一个新的着色器时，它都会将其 RenderType 配置为“不透明”。因此，USB_replacement_shader 可能会替换我们之前分配给材质的 Unlit 着色器。

为了清楚地预览更改，我们将进入 USB_replacement_shader 的片段着色器阶段并添加红色，然后将其乘以输出颜色。

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

接着必须确保在 USBReplacementController 脚本中的 Shader 类型替换变量中包含 USB_replacement_shader。

3） 子着色器混合-SubShader Blending

混合是将两个像素混合为一个的过程。它的命令与内置 RP 和 Scriptable RP 兼容。

混合发生在称为“合并(merging)”的阶段，该阶段将像素的最终颜色（已在片段着色器阶段处理的那些像素）与其深度相结合。这个阶段，发生在渲染管道的末端；在片段着色器阶段之后，执行模板缓冲区、z 缓冲区和颜色混合。

默认情况下，这个属性没有写在我们的着色器中，因为它是一个可选函数，主要在我们处理透明对象时使用，例如，当我们必须在另一个像素前面绘制一个具有低不透明度级别的像素时。

它的默认值是“Blend Off”，但我们可以激活它来生成不同类型的 Blending，就像出现在 Photoshop 中的混合模式那样。

Blend [SourceFactor] [DestinationFactor]

“混合(Blend)”是一个函数，它需要两个称为“factors”的值才能运行，并且基于一个方程，它将是我们在屏幕上获得的最终颜色。根据 Unity 中的官方文档，定义 Blending 值的公式如下：

B = SrcFactor * SrcValue [OP] DstFactor * DstValue.

要理解这个操作，我们必须考虑以下几点：片段着色器阶段首先发生，然后，作为一个可选过程；合并阶段。

“SrcValue”（源值），已经在片元着色器阶段进行了处理，将对应像素的 RGB 颜色输出。

“DstValue”（目标值）对应于已写入“目标缓冲区”的 RGB 颜色，更好地称为“渲染目标”（SV_Target）。当混合选项在我们的着色器中未激活时，SrcValue 会覆盖 DstValue。但是，如果我们激活此操作，两种颜色会混合以获得一种新颜色，它会覆盖之前的 DstValue。

“SrcFactor”（源因子）和“DstFactor”（目标因子）是三个维度的向量，根据它们的配置而变化。它们的主要作用是修改 SrcValue 和 DstValue 的值来达到有趣的效果。

我们可以在 Unity 文档中找到的一些 Factor 是：

① Off：禁用混合选项。

降哩降哩没有游戏人生

② One, (1, 1, 1)。
③ Zero, (0, 0, 0)。
④ SrcColor：等于 SrcValue 的 RGB 值。
⑤ SrcAlpha：等于 SrcValue 的 Alpha 值。
⑥ OneMinusSrcColor：1 减去 SrcValue 的 RGB 值（1-R，1-G，1-B）。
⑦ OneMinusSrcAlpha：1 减去 SrcValue 的 Alpha (1 – A, 1 – A, 1- A)。
⑧ DstColor：等于 DstValue 的 RGB 值。
⑨ DstAlpha：等于 DstValue 的 Alpha 值。
⑩ OneMinusDstColor：1 减去 DstValue 的 RGB 值（1-R，1-G，1-B）。
⑪ OneMinusDstAlpha：1 减去 DstValue 的 Alpha (1 – A, 1 – A, 1- A)。

值得一提的是，Alpha 通道的 Blending 与我们处理一个像素的 RGB 颜色的方式是一样的，但是因为不经常使用，它是在一个独立的过程中完成的。同样，通过不执行此过程，渲染目标上的写入得到优化。

假设我们有一个值为 [0.5 R , 0.45 G , 0.35 B] 的 RGB 颜色像素。此颜色已由片元着色器阶段处理，因此它对应于 DstValue。现在，我们将这个值乘以等于 [1, 1, 1]的“ SrcFactor One ”。每个数字乘以“1”得到相同的值，因此，SrcFactor 和 DstValue 之间的结果与其初始值相同。

B = [0.5R, 0.45G, 0.35B] [OP] DstFactor * DstValue.

“OP”是指我们将要执行的操作。默认情况下，它设置为“Add”。

B = [0.5R, 0.45G, 0.35B] + DstFactor * DstValue.

一旦我们获得了第一个操作的值，它就会被 DstValue 覆盖，因此设置为相同的颜色 [0.5R, 0.45G, 0.35B]。所以，我们将这个颜色乘以“ DstFactor DstColor ”，它等于我们在 DstFactor 中的当前值。

DstFactor [0.5R, 0.45G, 0.35B] * DstValue [0.5R, 0.45G, 0.35B] = [0.25R, 0.20G, 0.12B].

最后，像素的输出颜色是。

B = [0.5R, 0.45G, 0.35B] + [0.25R, 0.20G, 0.12B].

B = [0.75R, 0.65G, 0.47B]

如果我们想在我们的着色器中激活混合，我们必须使用 Blend 命令，然后是 SrcFactor，然后是 DstFactor。它的语法如下：

```
Shader "InspectorPath / shaderName"
{
    Properties {} …
    SubShader
    {
        Tags { "Queue" = "Transparent" "RenderType"="Transparent"}
        Blend SrcAlpha OneMinusSrcAlpha
    }
}
```

如果我们想在着色器中使用混合，则需要添加和修改“渲染队列(Render Queue)”。我们已经知道，“Queue”标签的默认值为“ Geometry ”，这意味着我们的对象会显得不透明。如果我们希望我们的对象看起来是透明的，那么我们必须首先将“Queue”更改为“Transparent”，然后添加某种混合类型。

最常见的混合类型如下：

① Blend SrcAlpha OneMinusSrcAlpha-通用透明混合

哔哩哔哩次元游戏人生

② Blend One One-加法混合颜色
③ Blend OneMinusDstColor One-温和的加法混合颜色
④ Blend DstColor Zero-乘法混合颜色
⑤ Blend DstColor SrcColor-乘法混合*2
⑥ Blend SrcColor One-叠加混合
⑦ Blend OneMinusSrcColor One-柔光混合
⑧ Blend Zero OneMinusSrcColor-负色混合

配置我们的混合模式的另一种方法是通过依赖“UnityEngine.Rendering.BlendMode”。这行代码允许我们从检查器更改材质中对象的混合。要首先设置它，我们必须将“Toggle Enum”添加到我们的属性中，然后声明 SrcFactor 和 DstFactor。

```
// [Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend (“Source Factor”, Float) = 1
// [Enum(UnityEngine.Rendering.BlendMode)] _DstBlend (“Destination Factor”, Float) = 1

Shader "InspectorPath / shaderName"
{
    Properties
    {
        [Enum(UnityEngine.Rendering.BlendMode)]
            _SrcBlend ("SrcFactor", Float) = 1
        [Enum(UnityEngine.Rendering.BlendMode)]
            _DstBlend ("DstFactor", Float) = 1
    }
    SubShader
    {
        Tags { "Queue" = "Transparent" "RenderType"="Transparent"}
        Blend [_SrcBlend] [_DstBlend]
    }
}
```

混合选项可以写在不同的字段中：在 SubShader 字段或 Pass 字段中，位置将取决于 pass 的数量和我们需要的最终结果。

4） 子着色器透明度蒙版-SubShader AlphaToMask

有一些混合类型很容易控制，例如“SrcAlpha OneMinusSrcAlpha Blend”，它在包含 Alpha 通道的情况下添加了透明效果，但在其他情况下，混合无法为我们的着色器生成透明度。在这种情况下，使用了“AlphaToMask”属性，它在 Alpha 通道上应用蒙版，并且是一种与内置 RP 和 Scriptable RP 兼容的技术。

与混合不同，蒙版只能将值“一或零”分配给 Alpha 通道。而 Blending 可以产生不同程度的透明度；从“0.0”到“1.0”的级别，AlphaToMask 只能生成整数。这转化为更苛刻的透明度类型，在特定情况下有效，例如，它对一般植被和创建空间门户效果非常有用。

AlphaToMask On。

AlphaToMask Off。（默认值）

要激活此命令，我们可以在 SubShader 字段和 pass 字段中声明它。它只有两个值：“On 和 Off”，声明方式如下：

```
Shader "InspectorPath / shaderName"
```

```
{
    Properties {} …
    SubShader
    {
        Tags { "Queue" = "Transparent" "RenderType"="Transparent"}
        Blend OneMinusDstColor One
        AlphaToMask On
    }
}
```

需要注意的是，与 Blending 不同的是，在这种情况下，不需要添加 Transparency 标签或其他命令。我们只需添加 AlphaToMask 并自动第四个颜色通道“A”获取我们程序中蒙版的质量。

5） 子着色器颜色蒙版-SubShader ColorMask

此命令允许我们的 GPU 将自身限制为写入选定的颜色通道，并且与内置 RP 和 Scriptable RP 兼容。

当我们创建着色器时，默认情况下 GPU 会写入与颜色 (RGBA) 对应的所有通道，但是，在某些情况下，我们可能只想显示一些颜色通道（例如红色通道或效果的“R”）。

ColorMask R-我们的对象看起来是红色的。

ColorMask G-我们的对象看起来是绿色的。

ColorMask B-我们的对象看起来是蓝色的。

ColorMask A-我们的对象会受到透明度的影响。

ColorMask RG-我们可以使用两个通道混合。

正如我们已经知道的，缩写词 RGBA 代表红色、绿色、蓝色和 Alpha，因此，如果我们将蒙版配置为“G”值，我们只会将绿色通道显示为颜色输出。这个 ShaderLab 命令使用起来非常简单，我们可以在 SubShader 和 Pass 中使用它。

```
Shader "InspectorPath / shaderName"
{
    Properties {} ...
    SubShader
    {
        Tags { "Queue" = "Geometry"}
        ColorMask RGB
    }
}
```

6） 子着色器剔除和深度测试-SubShader Culling and Depth Testing

要理解这两个概念，我们首先要了解 Z-Buffer（也称为 Depth Buffer，即深度缓冲）和深度测试是如何工作的。

在开始之前，我们必须考虑像素具有深度值。

这些值存储在深度缓冲区中，该缓冲区确定一个对象在屏幕上是在另一个之前还是之后。

另一方面，深度测试是确定深度缓冲区中的像素是否会更新的条件。

正如我们已经知道的，一个像素有一个以 RGB 颜色测量并存储在 Color Buffer 中的分配值。Z-Buffer 添加了一个额外的值，该值根据与相机的距离测量像素的深度，但仅对于位于其截锥体内的那些表面，这允许两个像素的颜色相同，但深度不同。

哔哩哔哩张无游戏人生

物体离相机越近，Z-Buffer 值越低，buffer 值较低的像素会覆盖值较高的像素。

为了理解这个概念，让我们假设我们的场景中有一个相机和一些图元，它们都位于“Z”空间轴上。现在，为什么在 z 轴上？Z-Buffer 中的“Z”来自于 Z 值测量相机与空间“Z”轴上的物体之间的距离，而“X 和 Y”值测量的是空间的水平和垂直位移屏幕。

“Buffer”一词指的是一个“内存空间”，数据将在其中临时存储，因此，Z-Buffer 是指我们场景中的物体与相机之间的深度值，分配给每个像素。

例如，我们要绘制一个总共有 36 个像素的屏幕。

每次我们在场景中放置一个对象时，该对象都会在屏幕上占据一定的像素区域。所以，让我们假设我们想要在场景中放置一个绿色方块。鉴于其性质，它将占据像素 8 到 29，因此，该区域内的所有像素都被激活并涂成绿色，同样，此信息将被发送到 Z-Buffer 和 Color Buffer。

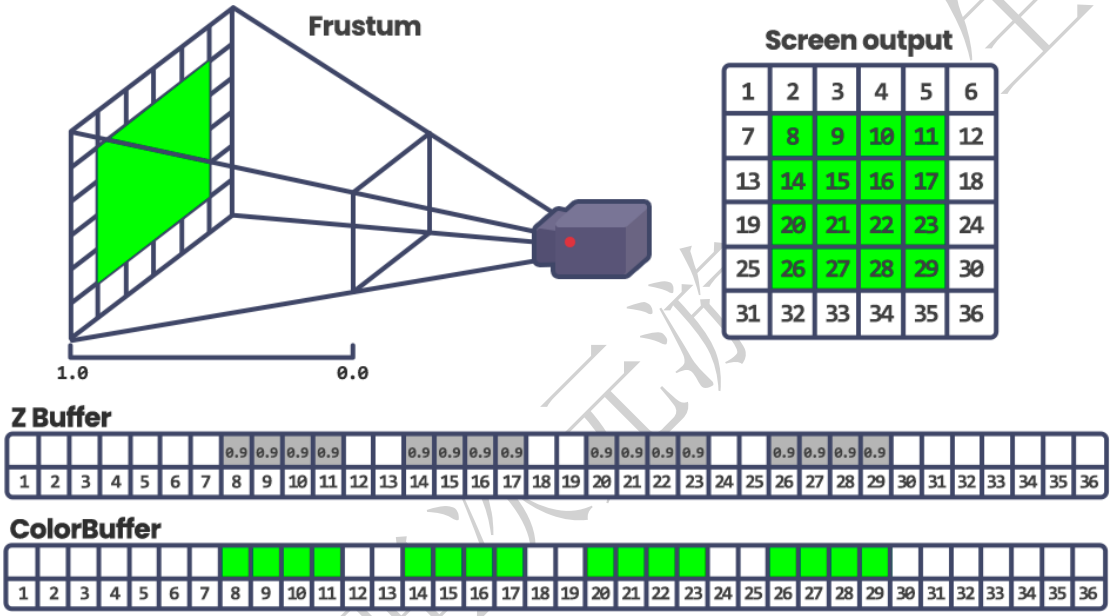


图54. Z-Buffer 存储场景中物体的深度，Color Buffer 存储 RGBA 颜色信息

我们在场景中放置了一个新的正方形，这次是红色的并且更靠近相机。为了与上一个区别，我们将这个正方形变小，从 15 像素到 29 像素。我们可以看到，这个区域已经被初始正方形的信息占据了，那么这里发生了什么？由于红色方块距离相机的距离较短，这会覆盖 Z-Buffer 和 Color Buffer 的值，激活该区域的像素，替换之前的颜色。

哔哩哔哩沃元游生

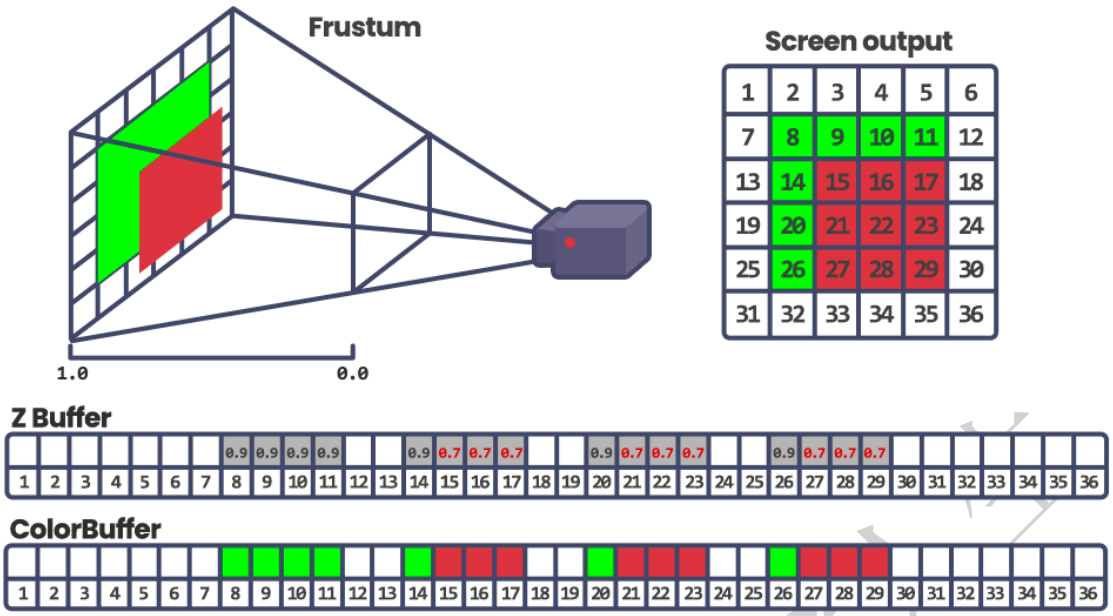


图55.　　距离更近的图像

在向更靠近相机的场景添加新元素的情况下，将以相同的方式重复此过程。总之，Z-Buffer 和 Color Buffer 的值会被距离相机最近的物体覆盖。

产生有吸引力的视觉效果的一种方法是修改 Z-Buffer 值。为此，我们将讨论 Unity 中包含的三个选项：Cull、ZWrite 和 ZTest。

像标签一样，剔除和深度测试选项可以写在不同的字段中：在 SubShader 字段或 Pass 字段中。位置将取决于我们想要达到的结果以及我们想要使用的 Pass 次数。

为了理解这个概念，假设我们要创建一个着色器来表示钻石的表面。为此，我们需要两个通道：第一个用于钻石的背景颜色，第二个用于其表面的光泽。在这个假设的情况下，由于我们需要两个实现不同功能的通道，因此有必要在每个通道中独立配置 Culling 选项。

7） 剔除-Cull

此属性在内置 RP 和 Scriptable RP 中兼容，控制在像素深度处理中将删除多边形的哪个面。这是什么意思？回想一下，多边形对象有内面和外面。默认情况下，外面是可见的（Cull Back）；但是，我们可以按照下面显示的方案激活内面。

① Cull Off：对象的两个面都被渲染

② Cull Back：默认情况下，渲染对象的背面。

③ Cull Front：渲染对象的正面。

该命令具有三个值，分别是：Back、Front 和 Off。默认情况下，“Back”处于活动状态，但是，出于优化目的，通常与剔除相关的代码行在着色器中不可见。如果我们想修改剔除选项，我们必须在我们想要使用的模式后面添加单词“Cull”。

```
Shader"InspectorPath / shaderName"
{
    Properties {} ...
    SubShader
    {
        // Cull Off
        // Cull Front
        Cull Back
```

```
        }
    }
```

我们还可以通过从 Enum Drawer 中声明并作为函数参数传递的依赖项“UnityEngine.Rendering.CullMode”在 Unity Inspector 中动态配置 Culling 选项。

```
Shader "InspectorPath/shaderName"
{
    Properties
    {
        [Enum(UnityEngine.Rendering.CullMode)]
        _Cull ("Cull", Float) = 0
    }
    SubShader
    {
        Cull [_Cull]
    }
}
```

另一个有用的选项是通过语义 SV_IsFrontFace，它允许我们在两个网格面上投影不同的颜色和纹理。为此，我们只需声明一个布尔变量并在片元着色器阶段将此类语义分配为参数。

```
fixed4 frag (v2f i, bool face : SV_IsFrontFace) : SV_Target
{
    fixed4 colFront = tex2D(_FrontTexture, i.uv);
    fixed4 colBack = tex2D(_BackTexture, i.uv);

    return face ? colFront : colBack;
}
```

值得一提的是，此选项仅在“Cull”命令之前已在着色器中设置为“Off”时有效。

8） 深度写入-ZWrite

此命令控制将对象的表面像素写入 Z-Buffer，也就是说，它允许我们忽略或尊重相机和对象之间的深度距离。ZWrite 有两个值，分别是：On 和 Off，其中“On”对应其默认值。我们通常在使用透明胶片时使用此命令，例如，当我们激活混合选项时。

① ZWrite Off，用于透明度。

② ZWrite On，默认值。

为什么在使用透明胶片时要禁用 Z-Buffer？主要是因为半透明的像素叠加（Z-fighting）。当我们处理半透明物体时，GPU 不知道哪个物体在另一个物体前面是很常见的，当我们在场景中移动相机时，会在像素之间产生重叠效果。要解决这个问题，我们必须简单地通过将 ZWrite 命令设置为“Off”来停用 Z-Buffer ，如下例所示：

```
Shader "InspectorPath / shaderName"
{
    Properties {} …
    SubShader
    {
        Tags { "Queue" = "Transparent" "RenderType"="Transparent"}
        Blend SrcAlpha OneMinusSrcAlpha
```

```
            ZWrite Off
        }
    }
```

Z-fighting（深度冲突）发生在我们有两个或多个与相机距离相同的物体时，导致 Z-Buffer 中的值相同。

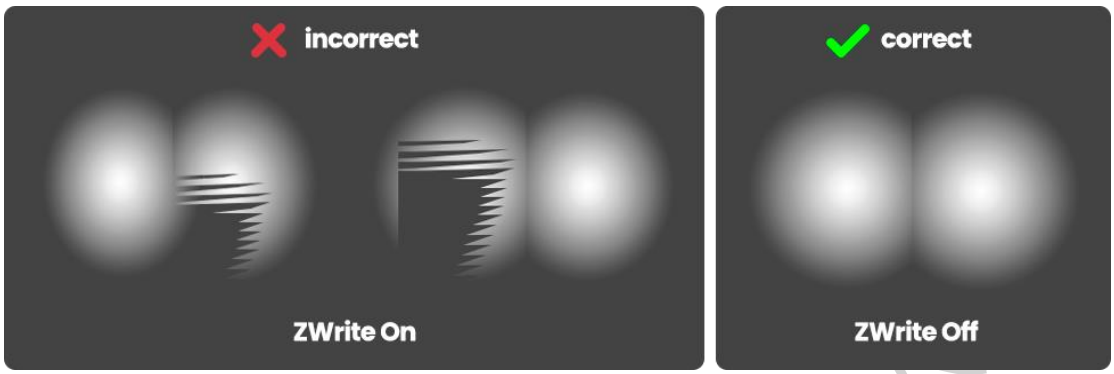


图56.　　z-fighting

当试图在渲染管道的末端渲染一个像素时，就会出现这种效果。由于 Z-Buffer 无法确定哪个元素在另一个元素之后，它会产生闪烁的线条，这些线条会根据相机的位置改变形状。

要纠正这个问题，我们只需要使用“ZWrite off”命令禁用 Z-Buffer。

9） 深度测试-ZTest

ZTest 控制应如何执行深度测试，它通常用于多通道着色器中以生成颜色和深度的差异。该属性有七个不同的值，它们是：

① ZTest Less: (<) 绘制前面的对象。它会忽略相同距离或位于着色器对象后面的对象。

② ZTest Greater: (>) 在后面绘制对象。它不会绘制位于相同距离或位于着色器对象前面的对象。

③ ZTest LEqual：（≤）默认值。绘制位于前面或相同距离的对象。它不会在着色器对象后面绘制对象。

④ ZTest GEqual：（≥）在后面或相同距离处绘制对象。不在着色器对象前面绘制对象。

⑤ ZTest Equal: (==) 绘制相同距离的对象。不在着色器对象的前面或后面绘制对象。

⑥ ZTest NotEqual: (!=) 绘制距离不同的对象。不绘制与着色器对象距离相同的对象。

⑦ ZTest Always：无论对象相对于相机的距离如何，都绘制所有像素。

为了理解这个命令，我们将做以下练习：假设我们的场景中有两个对象；一个立方体和一个球体。球体相对于相机位于立方体的前面，像素深度与预期一致。

哔哩哔哩次元游戏人

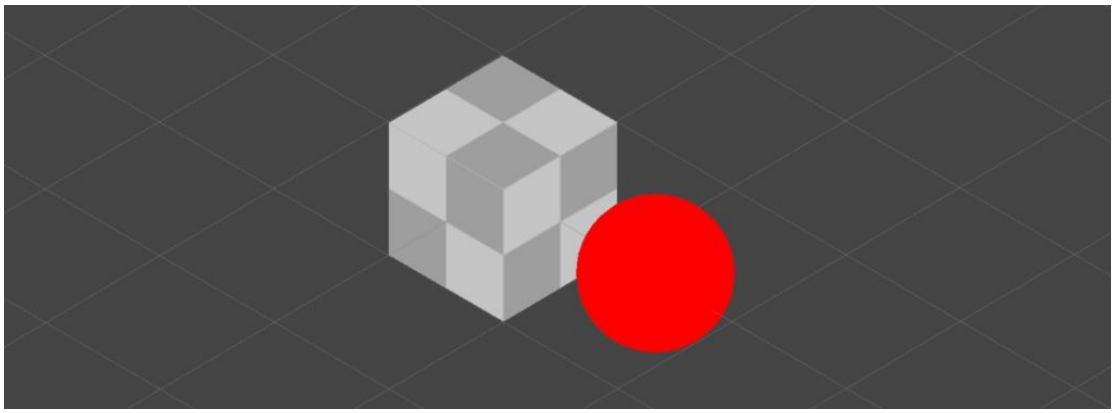


图57.　Sphere Is Forward Of Square

如果我们再次将球体定位在立方体后面，深度值将如预期的那样。因为 Z-Buffer 是存储屏幕上每个像素的深度值。深度值是根据物体与相机的接近程度来计算的。

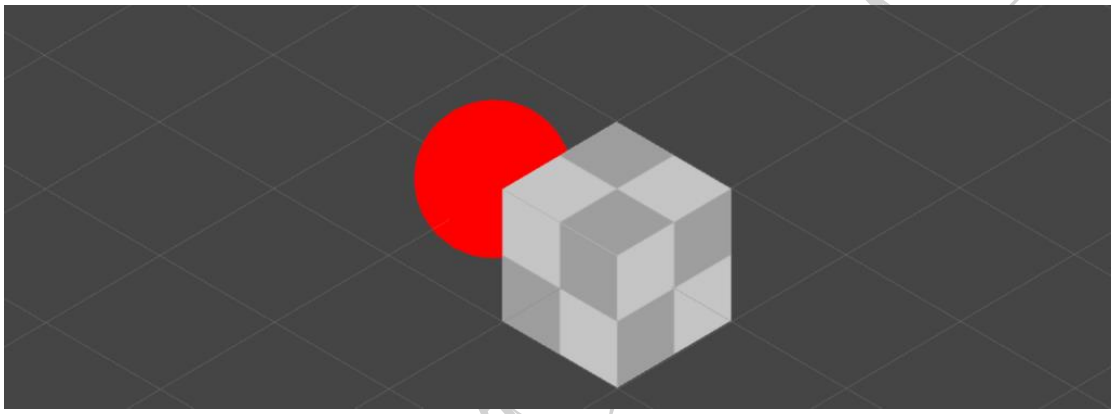


图58.　Square Is Forward Of Sphere

现在，如果我们始终激活 ZTest 会发生什么？在这种情况下，不会进行深度测试，因此，所有像素都会在屏幕上以相同的深度出现。

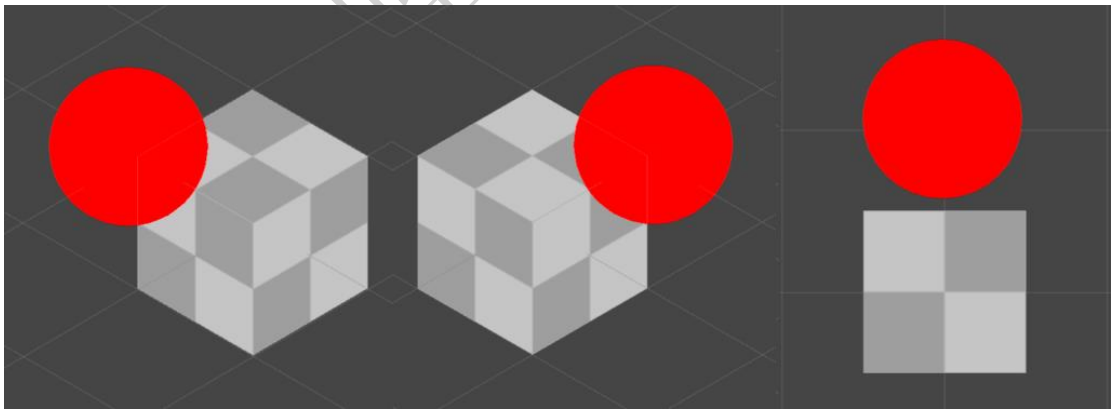


图59.　ZTest Always

```
Shader "InspectorPath / shaderName"
{
    Properties {} ...
    SubShader
    {
        Tags { "Queue" = "Transparent" "RenderType"="Transparent"}
        ZTest LEqual
```

```
    }
}
```

10）模板-Stencil

“模板缓冲区为帧缓冲区中的每个像素存储一个八位（0 到 255）的整数值。在为给定像素运行片段着色器阶段之前，GPU 可以将模板缓冲区中的当前值与确定的参考值进行比较，这个过程称为模板测试。如果模板测试通过，GPU 将执行深度测试。如果模板测试失败，GPU 会跳过该像素的其余处理。这意味着您可以使用模板缓冲区作为掩码来告诉 GPU 哪些像素要绘制，哪些像素要丢弃”。

为了理解它，我们必须考虑到模板缓冲区本身是一个必须创建的“纹理”，为此，它为帧缓冲区中的每个像素存储一个从 0 到 255 的整数值。

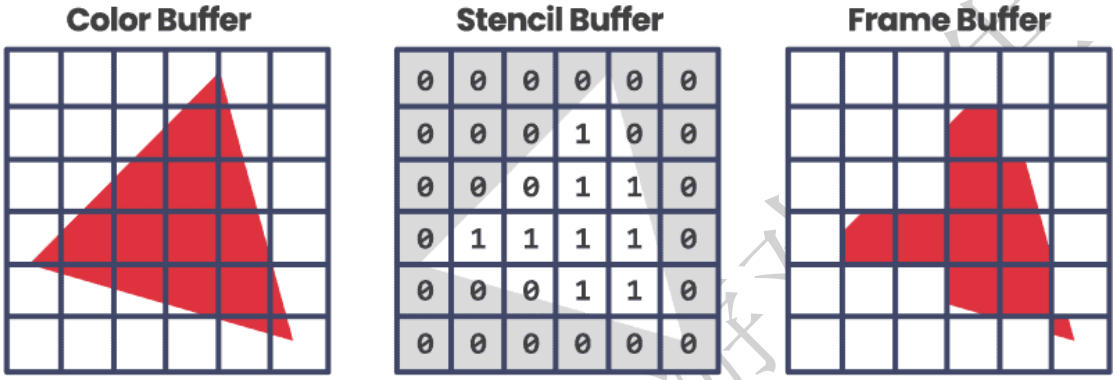


图60.　　模板缓冲

正如我们已经知道的，当我们在场景中定位对象时，它们的信息被发送到顶点着色器阶段（例如顶点位置）。在这个阶段，我们对象的属性从 object-space 转换为 world-space，然后是 view-space，最后是 clip-space。此过程仅发生在相机截锥体内的那些对象上。

当这些信息被正确处理后，它会被发送到光栅化阶段，它允许我们以像素为单位投影场景中对象的坐标，但是，在到达这一点之前，它会经过一个称为剔除(Culling)和深度测试(Depth testing)的处理的前一个阶段。在这个阶段，我们可以在着色器中操作各种过程，其中包括 Cull、ZWrite、ZTest 和 Stencil。

基本上，模板缓冲区所做的是激活模板测试，它允许丢弃“片元”（像素），这样它们就不会在片段着色器阶段进行处理，从而在我们的着色器中生成遮罩效果。Stencil Test 执行的函数具有以下语法：

```
if ( StencilRef & StencilReadMask [Comp] StencilBufferValue & StencilReadMask)
{
    Accept Pixel.
}
else
{
    Discard Pixel.
}
```

“StencilRef”是我们要传递给 Stencil Buffer 作为参考的值，这是什么意思？请记住，模板缓冲区是覆盖对象像素区域的“纹理”。StencilRef 用作映射 Stencil Buffer 中找到的所有像素的 id。

例如，我们将 StencilRef 的值设为 2。

图61.　StencilRef

在上面的示例中，覆盖胶囊区域的所有像素都被标记为 StencilRef 的值，因此现在 Stencil Buffer 等于 2。

随后，将为所有具有默认值为 255 的参考值的像素创建一个掩码 (StencilReadMask)。

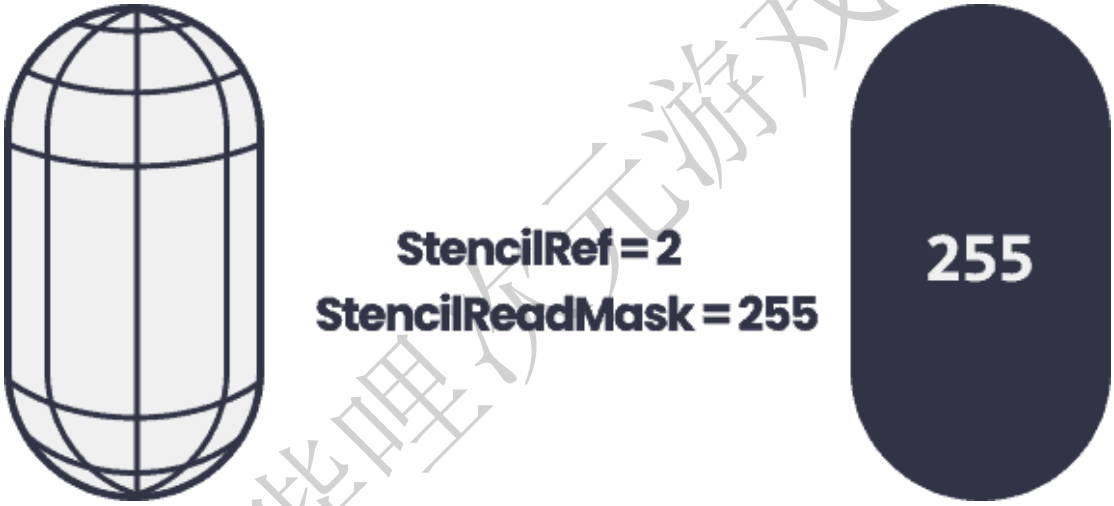


图62.　StencilReadMask

因此，上述操作如下。

```
if ( 2 & 255 [Comp] 2 & 255)
{
    Accept Pixel.
}
else
{
    Discard Pixel.
}
```

“Comp”指的是一个比较函数，它给出一个真假值。如果该值为真，则程序将像素写入帧缓冲区，否则，该像素将被丢弃。

在比较函数中，我们可以找到以下预定义的运算符：

操作	功能
Comp Never-从不比较	该操作将始终传递 False。
Comp Less	<

Comp Equal	==
Comp LEqual	≤
Comp Greater	>
Comp NotEqual	!=
Comp GEqual	≥
Comp Always-总是比较	该操作将始终传递 True。

我们至少需要两个着色器来使用模板缓冲区：一个用于遮罩，一个用于被遮罩的对象。

假设我们的场景中有三个对象：一个立方体、一个球体和一个正方形。球体在立方体内部，我们想用正方形作为“遮罩”来隐藏立方体，以便可以看到里面的球体。为了创建这个示例蒙版，我们将创建一个名为 USB_stencil_ref 的着色器，其语法如下：

```
SubShader
{
    Tags { "Queue" = "Geometry -1" }
    ZWrite Off
    ColorMask 0

    Stencil
    {
        Ref 1
        Comp Always
        Pass Replace
    }
}
```

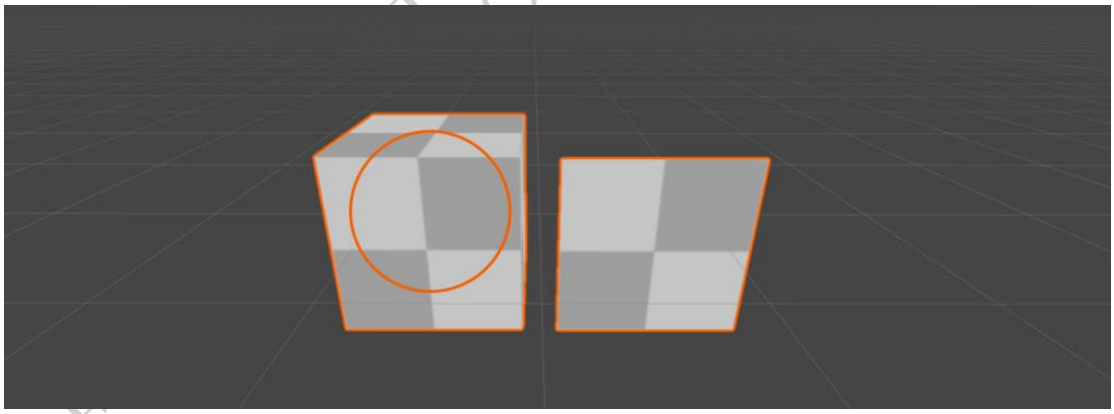


图63.　　模板示例

让我们分析以上内容。我们做的第一件事是配置“队列(Queue)”等于“几何减一”。几何默认为 2000，因此，几何减 1 等于 1999，这将处理我们的正方形（蒙版），我们将首先在 Z 缓冲区中应用此着色器。

但是，我们知道，Unity 默认情况下会根据对象在场景中与相机相关的位置来处理对象，因此如果我们想禁用此功能，我们必须将属性“ ZWrite 设置为 Off ”。

然后我们将“ ColorMask 设置为零”，以便在帧缓冲区中丢弃掩码像素并显示为透明。

在这一点上，我们的正方形仍然不能作为掩码工作，因此我们接下来必须做的是添加命令“Stencil”，以便它可以正常工作。

我们的参考值“Ref 2”（StencilRef）在 GPU 上使用“比较操作”中定义的操作与 Stencil Buffer 的当前内容进行比较。

“Comp Always（始终比较）”确保在模板缓冲区中设置“2”，同时考虑到屏幕上正方形所覆盖的区域。最后“Pass Replace”指定 Stencil Buffer 的当前值被 StencilRef 值“替换”。

现在我们要分析我们要屏蔽的对象。我们将创建另一个名为 USB_stencil_value 的着色器来说明这一点。它的语法如下：

```
SubShader
{
    Tags { "Queue" = "Geometry" }
    Stencil
    {
        Ref 1
        Comp NotEqual
        Pass Keep
    }
}
```

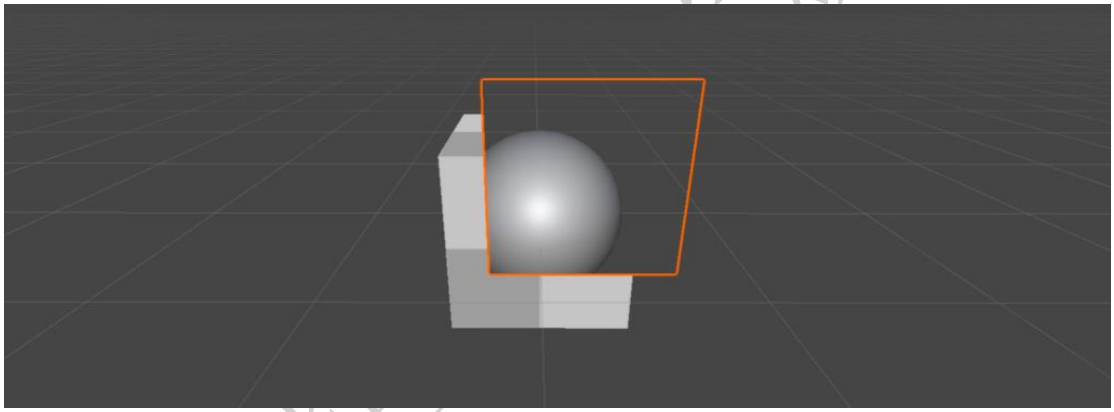


图64.　　遮罩

与我们之前的着色器不同，我们将保持 Z-Buffer 处于活动状态，以便根据其相对于相机的位置在 GPU 上进行渲染。然后我们添加 Stencil 命令，以便我们的对象可以被屏蔽。我们再次添加“Ref 2”以将此着色器与遮罩的 USB_stencil_ref 链接。

然后，在比较操作中，我们分配“Comp NotEqual”。这表明暴露在正方形周围的立方体区域将被渲染，因为模板测试通过（没有比较或 true）。

另一方面，对于正方形区域，相等（Equal）Stencil Test 不会通过，像素会被丢弃。“Pass Keep”表示 Cube 保持 Stencil Buffer 的当前内容。

如果我们想使用多个掩码，我们可以给一个不同于“2”的 Ref 属性编号。

十一、　限定语义

[NoScaleOffset]- 材质面板不显示 UV 偏移

[HideInInspector]- 在显示面板隐藏属性

[Normal]- 表明贴图为法线贴图

[HDR]-表示贴图是 HDR 贴图

[Gamma]

十二、 顶点和片元着色器基础

1. CGPROGRAM/ENDCG

我们上面回顾的所有部分都是用 ShaderLab 声明性语言编写的，我们在图形编程语言中的真正的挑战是从 CGPROGRAM 或 HLSLPROGRAM 声明开始。

默认情况下，我们可以看到 Unity 在我们的程序中包含了 CG 这个词。关于这一点，官方文档（版本 2021.2）说：

“Unity 最初使用的是 Cg 语言，因此有一些词的名称和.cginc 扩展名，但软件不再使用这种语言，但是仍然包含这些词，并且出于兼容性原因，代码仍在编译中”。

如上所述，由于 HLSL 目前是官方的图形编程语言（实际上 Shader Graph 是基于这种语言），因此在未来的版本中，我们的着色器可能会出现 HLSLPROGRAM 声明而不是 CGPROGRAM。

任何方式我们都可以简单地通过替换单词 CGPROGRAM 为 HLSLPROGRAM 和 ENDCG 为 ENDHLSL 来更新我们的着色器，然后我们的程序将在内置 RP 和通用 RP 和高清 RP 中编译。

如果我们想将程序更新为 HLSL，我们将不得不更改一些设置并向我们的着色器添加一些路线，这将使这更难理解。

```
Pass      // Cg version
{
          CGPROGRAM
          ENDCG
}

Pass      // HLSL version
{
          HLSLPROGRAM
          ENDHLSL
}
```

我们的着色器编译所需的所有函数都将写入 CGPROGRAM 和 ENDCG 字段中。这个片段之外的那些函数将被 Unity 当作 ShaderLab 属性，如果它们不对应这种语言，那么我们的程序将无法编译。

```
Shader   "InspectorPath / shaderName "
{
    Properties {} ...
    SubShader
    {
        Tags { "RenderType "= "Opaque "}
        Pass
        {
            CGPROGRAM
            // all our functions written here
            ENDCG
        }
    }
}
```

哔哩哔哩次元游戏人生

2. 数据类型

Cg 和 HLSL 之间的数据类型存在细微差别。当我们在当前版本的 Unity 中创建默认着色器时，我们可以找到精度不同的浮点数，其中包括：

float、half、fixed。

用 Cg 编写的着色器可以在这三种精度下完美编译，但是 HLSL 无法编译“fixed”数据类型，因此，如果我们使用这种语言，我们将不得不用 half 或 float 去替换这种类型的所有变量和向量。

“float”（Cg 和 HLSL）是一种高精度数据类型，它是 32 位的，通常用于计算世界空间中的位置、纹理坐标 (UV) 或涉及复杂函数（如三角函数或幂函数）的标量计算。

“half”（Cg 和 HLSL）是半精度，为 16 位，其精度范围是-60000~+60000，通常用于计算低幅度矢量、方向、对象空间位置和高动态范围颜色。

“fixed”（Cg）精度低，只有 11 位，其精度范围是-2.0~+2.0，一般用于简单运算（如基本颜色存储）的计算。

上面的精度范围并不是绝对正确的，尤其是在不同平台和 GPU 上，它们实际的精度可能和上面给出的范围不一致。

大多数现代的桌面 GPU 会把所有计算都按最高的浮点精度进行计算，也就是说，float、half、fixed 在这些平台上实际是等价的。这意味着，我们在 PC 上很难看出因为 half 和 fixed 精度而带来的不同。

但在移动平台的 GPU 上，它们的确会有不同的精度范围，而且不同精度的浮点值的运算速度也会有所差异。因此，我们应该确保在真正的移动平台上验证我们的 Shader。

fixed 精度实际上只在一些较旧的移动平台上有用，在大多数现代的 GPU 上，它们内部把 fixed 和 half 当成同等精度来对待。

向量在使用中经常出现的一个问题是，如果我们只对所有变量使用浮动数据类型（Float）会发生什么？在实践中这是可能的，我们必须考虑到浮点数是一种高精度数据类型，这意味着它有更多的小数，因此，GPU 将花费更长的时间来计算它，增加时间并产生更多消耗。

在所需数据类型中使用向量和（或）变量至关重要，因此我们可以优化我们的程序，减少 GPU 上的图形负载。

我们可以在这两种语言中找到的其他广泛使用的数据类型是“Int、sampler2D 和 samplerCube”。

“Sampler”是指纹理的采样状态。在这种类型的数据中，我们可以存储纹理及其 UV 坐标。

通常，当我们想在着色器中使用纹理时，我们必须使用“Texture2D”类型来存储纹理并创建“SamplerState”进行采样。数据类型“Sampler”允许我们将纹理和采样状态存储在单个变量中。要详细了解采样器的功能，我们将执行以下操作：

```
// declare our _MainTex texture as a global variable
Texture2D _MainTex;

// declare the _MainTex sample as a global variable
SamplerState sampler_MainTex;

// go to the fragment shader stage
half4 frag(v2f i) : SV_Target
{
```

```
        // inside the col vector sample the texture in UV coordinates.
        half4 col = _MainTex.Sample(sampler_MainTex, i.uv);

        // return the colour of the texture.
        return col;
    }
```

上述代码我们必须使用“Texture2D”类型来存储纹理并创建“SamplerState”进行采样，但其实只需使用“Sampler”即可优化上述过程。

```
    // declare our sampler for _MainTex
    sampler2D _MainTex;

    // go to the fragment shader stage
    half4 frag(v2f i) : SV_Target
    {
        // sample the texture in UV coordinates using the function
        // tex2D().
        half4 col = tex2D(_MainTex, i.uv)

        // return the colour of the texture.
        return col;
    }
```

两个示例都返回与我们在属性中生成的纹理对应的相同值，并将从 Unity Inspector 分配。

我们可以在程序中使用标量值、向量(vectors)和矩阵(matrices)。

标量值是返回实数的值，可以是整数或小数（例如 1、0.4、4 等），要在我们的着色器中声明它们，我们必须首先添加“数据类型”，然后添加“标量值名称”，最后初始化它的默认值。

它的语法如下：

```
    float name = n;     // e.g. float a = 0;
    half name = n;      // e.g. float b = 1.456;
    fixed name = n;     // e.g. float c = 2.5;
```

向量(vectors)是返回一个多维的值（例如 XYZW）并且要在我们的着色器中声明它们，我们必须首先添加“数据类型”，然后是“维数”，然后是“向量名称”，最后初始化它的默认值.

它的语法如下：

```
    float2 name = n;     // e.g. float2 uv = float2(0.5, 0.5);
    half3 name = n;      // e.g. half3 position = float3(0, 0, 0);
    fixed4 name = n;     // e.g. fixed4 color = float4(1, 1, 1, 1);
```

矩阵是具有存储在行和列中的值的矩阵，它们具有多个维度，主要用于剪切、旋转和更改顶点位置。

要在我们的着色器中声明一个矩阵，我们必须首先添加“数据类型”，然后是“维数乘以自身”，然后是“矩阵名称”，最后是“初始化其默认值”，同时考虑到坐标 Xx、Yy & Zz 将等于 1。

哔哩哔哩次元游戏人生

它的语法如下：

```
float3x3 nombre = float3x3      // 三行三列矩阵
(
    1, 0, 0,
    0, 1, 0,
    0, 0, 1
);

half2x2 nombre = half2x2            // 两行两列矩阵
(
    1, 0,
    0, 1
);

fixed4x4 nombre = fixed4x4      // 四行四列矩阵
(
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 0
);
```

3. 编译指令

CG 程序片段通过指令嵌入在 PASs 中，夹在指令 CGPROGRAM 和 ENDCG 之间，通常看起来是这样的：

#pragma vertex vert 允许将称为 vert 的顶点着色器阶段编译到 GPU 作为顶点着色器。这是必不可少的，因为如果没有这行代码，GPU 将无法将“vert”函数识别为顶点着色器阶段，因此，我们将无法使用来自对象的信息，也无法将信息传递给片段着色器阶段投影到屏幕上。

#pragma fragment frag 指令与 pragma vertex 提供相同的功能，不同之处在于它允许称为“frag”的片元着色器阶段在代码中作为片段着色器进行编译。

与以前的指令不同，#pragma multi_compile_fog 具有双重功能。首先 multi_compile 指的是着色器变体，它允许我们在着色器中生成具有不同功能的变体。其次，单词“_fog ”在 Unity 的 Lighting 窗口中启用了雾功能，这意味着，如果我们转到该选项卡，Environment / Other Setting，我们可以激活或停用着色器的雾选项。

```
Pass
{
    //…….设置渲染状态…
    CGPROGRAM
    //编译指令
    #pragma vertex vert
    #pragma fragment frag
    //CG 代码
    ENDCG
}
```

哔哩哔哩次元游戏人生

在程序片段之前，通常需要先使用# pragma 声明编译指令，它们将告诉 Unity，哪个函数包含了顶点着色器的代码，哪个函数包含了片元着色器的代码。

编译指令	作用
#pragma vertex name	定义顶点着色器的名称，通常会使用 vert
#pragma fragment name	定义片元着色器的名称，通常会使用 frag
#pragma target name	定义 Shader 要编译的目标级别，默认为 2.5

其中 name 就是我们指定的函数名，这两个函数的名字不一定是 vert 和 frag，它们可以是任意自定义的合法函数名。但我们一般使用 vert 和 frag 来定义这两个函数，因为它们很直观。

1） 编译目标等级

当编写完 Shader 程序之后，其中的 CG 代码可以被编译到不同的 Shader Models（简称 SM）中，为了能够使用更高级的 GPU 功能，需要对应使用更高等级的编译目标。但是有一点需要注意的是，高等级的编译目标可能会导致 Shader 无法在旧的 GPU 上运行，因此编译目标等级并不是越高越好。

声明编译目标的级别可以使用＃pragma target name 指令，或者也可以使用＃pragma require feature 指令直接声明某个特定的功能，例如：

#pragma target 3.5 //目标等级为 3.5

#pragma require geometry tessellation//需要几何细分功能

指令	描述
#pragma target 2.0	默认的 Shader Target 等级。相当于 Direct3D 9 上的 Shader Model 2.0，不支持对顶点纹理的采样，不支持显式的 LOD 纹理采样等。
#pragma target 3.0	相当于 Direct3D 9 上的 Shader Model 3.0，支持对顶点纹理的采样等。
#pragma target 4.0	相当于 Direct3D 10 上的 Shader Model 4.0，支持几何着色器等。
#pragma target 5.0	相当于 Direct3D 11 上的 Shader Model 5.0。

需要注意的是，由于 Unity 版本的不同，Unity 支持的 Shader Target 种类也不同。虽然更高等级的 Shader Target 可以让我们使用更多的临时寄存器和运算指令，但一个更好的方法是尽可能减少 Shader 中的运算，或者通过预计算的方式来提供更多的数据。

2） 渲染平台

Unity 具有跨平台的特性，它支持很多的渲染 API，例如 Direct3D 11、OpenGL。默认情况下，Unity 会为所有支持的平台编译一份 Shader 程序，不过可以通过编译器指令 #pragma only_renderers PlatformName 或者#pragma exclude_renderers PlatformName 指定编译某些平台或不编译某些平台。

名称	渲染平台
d3d11	Direct3D 11/12
glcore	OpenGL 3.x/4.x
gles	OpenGl ES 2.0
gles3	OpenGl ES 3.x

哔哩哔哩次元游戏人生

metal	iOS/Mac Metal
vulkan	Vulkan
d3d11_9x	Direct3D 11 9.x 所支持的级别，通常在 WSA 平台使用
xboxone	Xbox One
ps4	PlayStation4
n3ds	任天堂 3DS
wiiu	任天堂 WiiU

在使用的时候，代码可能如下所示：

```
#pragma only_renderers d3d11 //目标只编译 Direct3D 11/12 平台
#pragma exclude_renderers glcore //不编译 OpenGL 3.s/4.x
```

3） 着色器函数

在开始讲解之前，先看一个最简单的 Shader，代码如下：

```
Shader "Custom/Simplest Shader"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma Vertex vert
            #pragma fragment frag

            void vert(int float4 vertex : POSITION,out float4 position : SV_POSITION)
            {
                position = UnityObjectToClipPos(vertex);
            }

            void frag(in float4 vertex : SV_POSITION,out fixed4 color : SV_Target)
            {
                Color = fixed4(1,0,0,1);
            }
            ENDCG
        }
    }
}
```

这是一个典型的顶点-片元着色器，名称为 Simple Shader。在这个 Shader 中只包含一个 Subshader，而 Subshader 中又只包含一个 Pass，最核心的 CG 程序嵌套在 CGPROGRAM 和 ENDCG 之间。

前两行编译指令分别定义了顶点着色器的名称为 vert，片元着色器的名称为 frag。通常情况下，为了代码的可读性和易用性考虑，一般都会使用 vert 和 frag 来命名。

在 Shader 中，顶点-片元着色器主要是通过顶点函数和片元函数来实现的。学过数学的读者对函数应该不是很陌生。通俗来讲，函数就是把输入的参数经过一些计算之后再进行输出，例如：$f(x)=2x$、$y=x^2$ 等都是函数。

哔哩哔哩次元游戏人生

a. 无返回值的函数

函数分为有返回值和无返回值。无返回值的顾名思义就是函数不会返回任何变量，而是通过 out 关键字将变量输出，上述 Simple Shader 中的顶点函数和片元函数使用的就是无返回值的函数。

其语法结构如下所示：

```
Voide name(in  参数, out  参数)
{
    //函数体
}
```

关键词解释：

Void：函数以 void 开拓，表示返回值为空。

name：定义函数的名称，后续可以通过这个名称调用函数。

In：输入参数，语法为：in+数据类型+名称，一个函数可以有多个输入，关键词 in 可以省略。

Out：输出参数，语法为：out+数据类型+名称，一个函数可以有多个输出。

下面我们再来看一遍顶点函数的代码：

```
void vert(int float4 vertex : POSITION,out float4 position : SV_POSITION)
        {
            position = UnityObjectToClipPos(vertex);
        }
```

顶点着色器输入一个 float4 类型的数据，名称为 vertex，经过 Unity 内置空间变换函数 UnityObjectToClipPos 把模型空间坐标转换到裁切空间坐标，然后输出为 float4 类型的 position。

```
void frag(in float4 vertex : SV_POSITION,out fixed4 color : SV_Target)
{
            Color = fixed4(1,0,0,1);
}
```

顶点函数输出的顶点坐标输入到片元着色器之后，最终输出 fixed4 类型的数据，名称为 color。在函数内 color 变量被赋值为(1,0,0,1)，也就是红色，最终 Shader 效果显示为红色。

b. 有返回值的函数

既然有无返回值的函数，那么反过来肯定会存在有返回值的函数。有返回值的函数不再使用 out 关键字输出参数，而是会在最后通过 return 关键字返回一个变量，语法结构如下：

```
Type name(in  参数)
{
    //函数体
    return 返回值
}
```

哔哩哔哩次元游戏人生

4. 语义

```
Shader "Unlit/NewUnlitShader"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Color("Color Tint",Color) = (1.0,1.0,1.0,1.0)
    }
    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 a2f
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float2 texcoord : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                fixed3 color : COLOR0;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;

            v2f vert (a2f v)
            {
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP,v.vertex);
                o.color = v.normal * 0.5 + fixed(0.5,0.5,0.5);
```

哔哩哔哩次元游戏人生

```
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed3 c = i.color;
                c *= _Color.rgb;
                return fixed4(c,1.0);
            }
            ENDCG
        }
    }
}
```

在顶点着色器和片元着色器的输入输出变量后还有一个冒号以及一个全部大写的名称，例如 SV POSIT1ION、POSITION、COLOR0。这些大写的名字是什么意思呢？它们有什么用呢？

实际上，这些是 Cg/HLSL 提供的语义（semantics）。如果读者从前接触过 Cg/HLSL 编程的话，可能对这些语义很熟悉。读者可以在微软官方网站的关于 DirectX 的文档（msdn.microsoft/en-us/library/windows/desktop/bb509647(v＝vs.85).aspx＃VS）中找到关于语义的详细说明页面。

根据文档我们可以知道，语义实际上就是一个赋给 Shader 输入和输出的字符串，这个字符串表达了这个参数的含义。通俗地讲，这些语义可以让 Shader 知道从哪里读取数据，并把数据输出到哪里，它们在 Cg/HLSL 的 Shader 流水线中是不可或缺的。需要注意的是，Unity 并没有支持所有的语义。

语义可以执行大量繁琐的操作，使用户能够避免直接与 GPU 底层进行交流。

通常情况下，这些输入输出变量并不需要有特别的意义，也就是说，我们可以自行决定这些变量的用途。例如在上面的代码中，顶点着色器的输出结构体中我们用 COLORO 语义去描述 color 变量。color 变量本身存储了什么，Shader 流水线并不关心。

而 Unity 为了方便对模型数据的传输，对一些语义进行了特别的含义规定。例如，在顶点着色器的输入结构体 a2v 用 TEXCOORD0 来描述 texcoord，Unity 会识别 TEXCOORD0 语义，以把模型的第一组纹理坐标填充到 texcoord 中。需要注意的是，即便语义的名称一样，如果出现的位置不同，含义也不同。例如，TEXCOORDO 既可以用于描述顶点着色器的输入结构体 a2v，也可用于描述输出结构体 v2f。但在输入结构体 a2v 中，TEXCOORD0 有特别的含义，即把模型的第一组纹理坐标存储在该变量中，而在输出结构体 v2f 中，TEXCOORD0 修饰的变量含义就可以由我们来决定。

在 DirectX 10 以后，有了一种新的语义类型，就是系统数值语义（system-value semantics）。这类语义是以 SV 开头的，SV 代表的含义就是系统数值（system-value）。这些语义在渲染流水线中有特殊的含义。例如在上面的代码中，我们使用 SV_POSITION 语义去修饰顶点着色器的输出变量 pos，那么就表示 pos 包含了可用于光栅化的变换后的顶点坐标（即齐次裁剪空间中的坐标）。用这些语义描述的变量是不可以随便赋值的，因为流水线需要使用它们来完成特定的目的，例如渲染引擎会把用 SV_POSITION 修饰的变量经过光栅化后显示在屏幕上。读者有时可能会看到同一个变量在不同的 Shader 里面使用了不同的语义修饰。例如，一些 Shader 会使用 POSITION 而非 SV_POSITION 来修饰顶点着色器的输出。SV_POSITION 是 DirectX 10 中引入的系统数值语义，在绝大多数平台上，它和

POSITION 语义是等价的，但在某些平台（例如索尼 PS4）上必须使用 SV_POSITION 来修饰顶点着色器的输出，否则无法让 Shader 正常工作。同样的例子还有 COLOR 和 SV_Target。因此，为了让我们的 Shader 有更好的跨平台性，对于这些有特殊含义的变量我们最好使用 SV 开头的语义进行修饰。

1） 顶点着色器的输入语义

顶点着色器的输入语义，就是从应用阶段传递模型数据给顶点着色器时 Unity 所使用的常用语义。这些语义虽然没有 SV 开头，但 Unity 内部赋予了它们特殊的含义。

语义	描述
POSITION	顶点的坐标信息，通常为 float3 或 float4 类型
NORMAL	顶点的法线信息，通常为 float3 类型
TEXCOORDn，如 TEXCOORD0、TEXCOORD1	该顶点的纹理坐标，TEXCOORD0 表示第一组纹理坐标，以此类推。通常为 float2、float3 或 float4 类型。
TANGENT	顶点的切向量，通常为 float4
COLOR	顶点的颜色信息，通常为 float4 类型

其中 TEXCOORDn 中 n 的数目是和 Shader Model 有关的，例如一般在 Shader Model2（即 Unity 默认编译到的 Shader Model.版本）和 Shader Model3 中，n 等于 8，而在 Shader Model4 和 Shader Model:5 中，n 等于 16。通常情况下，一个模型的纹理坐标组数一般不超过 2，即我们往往只使用 TEXCOORDO 和 TEXCOORDI。在 Unity 内置的数据结构体 appdata_full 中，它最多使用了 6 个坐标纹理组。

当顶点信息包含的元素少于顶点着色器输入所需要的元素时，缺少的那部分会被 0 填充，而 w 分离会被 1 填充。例如：顶点的 UV 坐标通常是二维向量，只包含 x 和 y 元素。如果输入的语义 TEXCOORD0 被声明为 float4 类型，那么顶点着色器最终获取到的数据将变成(x,y,0,1)。

2） 顶点着色器输出语义——片元着色器器的输入语义

在整个渲染流水线中，顶点着色器最重要的一项任务就是需要输出顶点在裁切空间中的坐标，这样 GPU 就可以知道顶点在屏幕上的栅格化位置以及深度值。在顶点函数中，这个输出参数需要使用 float4 类型的 SV_POSITION 语义进行填充。

顶点着色器产生的输出值将会在三角形遍历阶段经过差值计算，最终作为像素值输入到片元着色器。换句话说，顶点着色器的输出即为片元着色器的输入。

语义	描述
SV_POSITION	裁剪空间中的顶点坐标，结构体中必须包含一个用该语义修饰的变量。等同于 DirectX9 中的 POSITION，但最好使用 SV_POSITION
COLOR0	通常用于输出第一组顶点颜色，但不是必需的
COLOR1	通常用于输出第二组顶点颜色，但不是必需的
TEXCOORD0~TEXCOORD7	通常用于输出纹理坐标，但不是必需的

上面的语义中，除了 SV_POSITION 是有特别含义外，其他语义对变量的含义没有明确要求，也就是说，我们可以存储任意值到这些语义描述变量中。通常，如果我们需要把

一些自定义的数据从顶点着色器传递给片元着色器，一般选用 TEXCOORDO 等。

片元着色器会自动获取顶点着色器输出的裁切空间顶点坐标，所以片元着色器函数输入的 SV_POSITION 可以省略。这也解释了为什么有些 Shader 的片元函数中只有输出参数，但是没有输入参数。

3） 片元着色器输出语义

语义	描述
SV_Target	输出值将会存储到渲染目标(render target)中，等同于 DirectX 9 中的 Color 语义，但最好使用 SV_Target。

片元着色器通常只会输出一个 fixed4 类型的颜色信息，输出的值会存储到渲染目标(render target)中，输出的参数使用 SV_Target 语义进行填充。

5. 结构体

在实际编写过程中，着色器通常需要输入输出多个参数，为了使代码编写更加方便，并且看起来更加美观，我们引入结构体类型。

1） 结构体语法

结构体允许存储多个不同类型的变量，并将多个变量包装成为一个整体进行输入或者输出。

```
Struct Type
{
    //变量_1;
    //变量_2;
    //变量_3;
    //变量_n;
}
```

Struct：定义结构体的关键字。

Type：给当前结构体定义一种类型，着色器函数定义输入和输出数据类型时会用到。结构体内包含的变量仍然需要定义数据类型和名称，然后填充对应的语义。最后通过［结构体名称］.［变量名称］的语法访问，例如：v.vertex，表示访问名称为 v 的结构体内的 vertex 变量。

2） 返回结构体的函数

函数可以写成有返回和无返回的形式，这是否意味着函数也可以返回结构体呢？答案是肯定的，下面就把上述无返回值的 Shader 改写成返回结构体的 Shader。

6. Unity 内置文件和变量

1） 内置的包含文件

在实际编写 Shader 的过程中，会频繁使用同一组数据作为顶点着色器的输人参数，例如：模型空间的顶点坐标和第一套 UV 坐标。为了提高代码的重复使用率以及 Shader 的编写速度，Unity 提供了一系列的包含文件，其中有预先定义的变量、各种辅助函数和空间变换矩阵等。

包含文件(include file)是以 cginc（cg include 的缩写）作为扩展名的文本文件。如果使用的是 Windows 系统，Unity 所有的包含文件存放在安装目录\Editor\Data\CGIncludes\路径下；如果使用的是 Mac 系统，包含文件则存放在 Applications\Unit\/Unity.app\Contents\CGIncludes\路径下。

哔哩哔哩次元游戏人生

在编写 Shader 时，只需要使用编译指令提前把对应的文件包含进 Shader，就可以直接使用了。与其他编译指令一样，包含文件的声明也要写在 CG 代码块内，它的语法结构如下所示：

```
CGPROGRAM
#include "UnityCG.cginc"
ENDCG
```

通过#include 关键词声明包含指令，引号内的名称即为要包含进 Shader 的文件，只要输入了正确的文件名称，Shader 在编译的时候就会自动在这个路径下查找对应的文件。

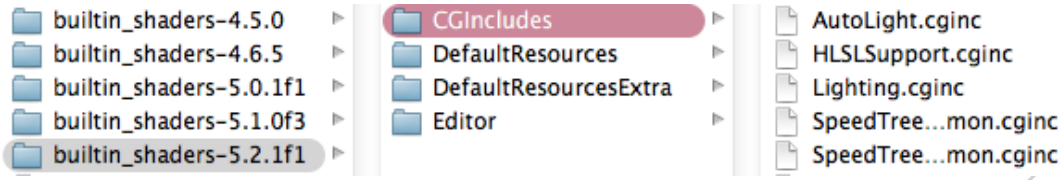


图65.　Unity 的内置着色器

从上图中可以看出，从官网下载的文件中包含了多个文件夹。其中，CGIncludes 文件夹中包含了所有的内置包含文件；DefaultResources 文件夹中包含了一些内置组件或功能所需要的 Unity Shader，例如一些 GUI 元素使用的 Shader；DefaultResourcesExtra 则包含了所有 Unity 中内置的 Unity Shader；Editor 文件夹目前只包含了一个脚本文件，它用于定义 StandardShader 所用的材质面板。这些文件都是非常好的参考资料，在我们想要学习内置着色器的实现或是寻找内置函数的实现时，都可以在这里找到内部实现。

我们也可以从 Unity 的应用程序中直接找到 CGIncludes 文件夹。在 Mac 上，它们的位置是：/Applications/Unity/Unity.app/Contents/CGIncludes；在 Windows 上，它们的位置是：Unity 的安装路径/Data/CGIncludes。

UnityCG,cginc	包含了最常使用的帮助函数、宏和结构体等。
AutoLight.cginc	包含灯光和阴影功能，Surface Shader 内部实现所使用的文件
HLSLSupport.cginc	包含有辅助宏和跨平台编译 Shader 的定义，编译的时候会自动包含进 Shader
Lighting.cginc	包含 Lambert 和 BlinnPhong 光照函数，编写表面着色器的时候会自动包含进来
Tessellation.cginc	包含曲面细分函数
UnityGloballlllumination.cginc	包含计算全局光照的函数
UnityImageBasedLighting.cginc	包含计算 IBL（基于图像的光照）的函数
UnityShader Variables.cginc	包含常用的全局变量，如 UNITY_MATRIX_MVP，编译的时候会自动包含进 Shader
UnityLightingCommon.cginc	包含计算全局光照所需要的结构体
UnityPBSLighting.cginc	包含基于物理着色所需要的光照计算函数，用于表面着色器
UnityShaderUtilities.cginc	包含了顶点从模型空间变换到裁切空间的变换函数
UnityShadowLibrary.cginc	包含了计算阴影时所用到的宏和函数

哔哩哔哩次元游侠人

UnityStandardBRDF.cginc	包含了标准着色器中用到的计算 BRDF（双向反射分布函数）所需的函数

可以看出，有一些文件是即便我们没有使用＃include 指令，它们也是会被自动包含进来的，例如 UnityShaderVariables.cginc。我们可以直接使用 UNITY_MATRIXMVP 变量来进行顶点变换。

2） UnityCG.cginc

a. 顶点输入结构体

Unity 在包含文件中声明了大量的结构体，在编写 Shader 的时候可以直接使用。

```
struct appdata_base
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_tan
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_full
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    float4 texcoord2 : TEXCOORD2;
    float4 texcoord3 : TEXCOORD3;
    fixed4 color : COLOR;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct appdata_img
{
    float4 vertex : POSITION;
```

哔哩哔哩次元游戏人生

```
        half2 texcoord : TEXCOORD0;
        UNITY_VERTEX_INPUT_INSTANCE_ID
    };

    struct v2f_img
    {
        float4 pos : SV_POSITION;
        half2 uv : TEXCOORD0;
        UNITY_VERTEX_INPUT_INSTANCE_ID
        UNITY_VERTEX_OUTPUT_STEREO
    };
```

结构体	包含信息
appdata_base	顶点坐标、顶点法线和第一套 UV 坐标
appdata_tan	顶点坐标、切向量、顶点法线和第一套 UV 坐标
appdata_full	顶点坐标、切向量、顶点法线、第一至四套 UV 坐标和顶点对应颜色。
appdata_img	顶点坐标、第一套 UV 坐标
v2f_img	裁切空间顶点坐标、纹理坐标

b. 顶点变换函数

函数	说明
float4 UnityObjectToClipPos (float3 pos)	将顶点从模型空间变换到齐次裁切空间，等同于 mul(UNITYMATRIX MVP，float4(pos，1.0))
float3 UnityObjectToViewPos (float3 pos)	将顶点从模型空间转换到摄像机空间，等同于 mul(UNITYMATRIX_MV，floate4(pos，1.0).xyz。当输入为 float4 类型，Unity 会自动重载为 float.3 类型
float3 UnityWorldToViewPos (float3 pos)	将顶点从世界空间变换到摄像机空间，等同于 mul(UNITY.MATRIX V，float4(pos，1.0)).xyz
float4 UnityWorldToClipPos (float3 pos)	将顶点从世界空间变换到齐次裁切空间，等同于 mul(UNITYMATRIX_VP，float4(pos，1.0))
float4 Unity View ToClipPos (float3 pos)	将顶点从摄像机空间变换到齐次裁切空间，等同于 mul(UNITYMATRIX P，float4(pos，1.0))

哔哩哔哩次元游戏人生

c. 向量变换函数

函数	说明
float3 UnityObjectToWorldDir(float3 dir)	将向量从模型空间转换到世界空间，已经标准化处理
float3 Unity WorldToObjectDir(float3 dir）	将向量从世界空间变换到模型空间，已经标准化处理
float3 UnityObjectTo WorldNormal(float3 norm）	将法线从模型空间变换到世界空间，已经标准化处理

d. 灯光辅助函数

UnityCG.cginc 中还定义了一些计算顶点指向灯光的方向向量。

函数	说明
float3 WorldSpaceLightDir (in float4 localPos)	输人模型空间顶点坐标，返回世界空间中从顶点指向灯光的向量，没有被标准化（遗留的函数，不建议使用）
float3 UnityWorldSpaceLightDir (in float3 worldPos)	输入世界空间顶点坐标，返回世界空间从顶点指向灯光的向量，没有被标准化
float3 ObjSpaceLightDir(in float4 v）	输入模型空间顶点坐标，返回模型空间中从顶点指向灯光的向量，没有被标准化。
float3 Shade4 PointLights（…）	输入一系列所需变量，返回 4 个点光源的光照信息，在前向渲染中使用这个函数计算逐顶点的光照

需要注意，上面的函数仅适用于前向渲染路径(ForwardBase 或 ForwardAdd Pass 类型)

e. 视角向量函数

UnityCG.cginc 中还定义了一些计算顶点指向摄像机的方向向量，也被称为视角向量。

函数	说明
float3 WorldSpace ViewDir (float4 v)	输入模型空间顶点，返回世界空间中从顶点指向摄像机的向量，没有被标准化（遗留的函数，不建议使用）
float3 Unity WorldSpace ViewDir(float3 v)	输入世界空间中的顶点，返回世界空间中从顶点指向摄像机的向量，没有被标准化
float3 ObiSpace ViewDir (float4 v)	输入模型空间顶点，返回模型空间中从顶点指向摄像机的向量，没有被标准化

哔哩哔哩次元游戏人生

f. 其他辅助函数和宏

UnityCG.cginc 中还定义了一些其他类型的函数和宏。

函数	说明
TRANSFORM_TEX(tex,name)	宏定义，输入 UV 坐标和纹理名称，得到贴图的纹理坐标
fixed3 UnpackNormal(fixed4 packednormal)	将法线向量从[0,1]映射到[-1,1]
half Luminance(half3 rgb)	将颜色数据转变为灰度数据
float4 ComputeScreenPos(float4 pos)	输入裁切空间顶点坐标，得到屏幕空间纹理坐标，用于屏幕空间纹理映射
float4 ComputeGrabScreenPos(float4 pos)	输入裁切空间顶点坐标，得到采样 GrabPass 的纹理坐标

宏在使用之前需要先定义，通过一个标识符代替一个字符串。在使用的时候只需要输入识别符即可，编译的时候 Unity 会自动将识别字符替换成字符串，因此可以将宏简单理解为字符串的替代。

宏定义的语法结构为：

#define name string

#define：表示宏定义的指令。

name：宏名称，后续可以直接输入名称进行使用。

string：编译的时候要把宏名称替换成的内容，可以是数字、表达式、函数等。

例如，上面表格中的 TRANSFORM_TEX(tex，name)宏，它在包含文件中的定义如下：

```
//Transforms 2D UV by scale/bias property
#define TRANSEORM_TEX(tex，name)(tex.xy name #_ST.xy name #_ST.zw)
```

TRANSFORM_TEX(tex，name)为宏名称，括号内是需要传人的两个变量，分别为模型的 UV 坐标和纹理名称。在宏定义的内容中，纹理名称后又补上了 ST，就是用来表示纹理平铺和偏移的变量。模型 V 先乘以平铺值，再加上偏移值，整个计算纹理坐标的算法与 Texture Property Shader 中所使用的算法一样，后续可以直接使用宏进行纹理坐标的计算。

3） UnityShaderVariables.cginc

在 UnityShader Variables.cginc 文件中，Unity 提供了一些内置的全局变量，例如，变换矩阵、灯光参数、时间变量等，这些变量在 Shader 中可以直接使用，不需要再进行包含声明。

a. 空间变换矩阵

在 UnityShaderVariables.cginc 包含文件中，Unity 提供了很多空间变换矩阵，使用者可以直接使用 CG 函数 mul(Matrix,Vertex)将顶点在不同空间之间进行变换。

矩阵	说明
UNITY_MATRIX_MVP	模型-观察-投影矩阵，用于将顶点/向量从模型空间变换到裁剪空间
UNITY_MATRIX_MV	模型-观察矩阵，用于将顶点/向量从模型空间变换到摄像机空间
UNITY_MATRIX_V	观察矩阵，用于将顶点/向量从世界空间变

	换到摄像机空间
UNITY_MATRIX_P	投影矩阵，用于将顶点/向量从摄像机空间变换到裁剪空间
UNITY_MATRIX_VP	观察投影矩阵，用于将顶点/向量从世界空间变换到裁剪空间
UNITY_MATRIX_T_MV	UNITY._MATRIX_MV 的转置矩阵
UNITY_MATRIX_IT_MV	UNITY_MATRIX_MV 的逆转置矩阵
unity_ObjectToWorld	模型矩阵，用于将顶点/向量从模型空间变换到世界空间
unity_WorldToObject	-Object2World 的逆矩阵，用于将顶点/向量从世界空间变换到模型空间

b. 时间变量

在实现动态效果的时候经常会使用时间因子作为动态变量，下表中汇中了所有可以直接使用的时间变量，表中所有的变量都是 float4 类型。

变量	说明
_Time	关卡从开始到现在所运行的时间，4 个分量分别为 t/20，t，t*2，t*3
_SinTime	将运行时间（t/8，t/4，t/2，t）输入到正弦函数
_CosTime	将运行时间（t/8，t/4，t/2，t）输入到余弦函数
Unity_DeltaTime	每一帧的递增时间，4 个分量分别为 dt，1/dt，smoothDt，1/smoothDt

4） 其他包含文件

除了最常用的 UnityCG.cginc 文件之外，Unity 还提供很多用于其他功能的包含文件。

包含文件	描述
AutoLight.cginc	包含灯光和阴影功能，Surface Shader 内部实现所使用的文件
HLSLSupport.cginc	包含有辅助宏和跨平台编译 Shader 的定义，编译的时候会自动包含进 Shader
Lighting.cginc	包含 Lambert 和 BlinnPhong 光照函数，编写表面着色器的时候会自动包含进来
Tessellation.cginc	包含曲面细分函数
UnityGlobalIllumination.cginc	包含计算全局光照的函数
UnityImageBasedLighting,cginc	包含计算 IBL（基于图像的光照）的函数
UnityShaderVariables.cginc	包含常用的全局变量，编译的时候会自动包含进 Shader
UnityLightingCommon.cginc	包含计算全局光照所需要的结构体
UnityPBSLighting.cginc	包含基于物理着色所需要的光照计算函数，用于表面着色器
UnityShaderUtilities.cginc	包含了顶点从模型空间变换到裁切空间的

哔哩哔哩次元游戏人生

	变换函数
UnityShadowLibrary.cginc	包含了计算阴影时所用到的宏和函数
UnityStandardBRDF.cginc	包含了标准着色器中用到的计算 BRDF（双向反射分布函数）所需的函数

十三、 其他

1. 着色器和材质

“材质是对如何渲染表面的定义，包括对纹理、平铺、偏移、颜色等的引用。材质的选项取决于正在使用的着色器”。

我们将材质视为“着色器的容器”，这意味着我们拥有执行表面计算（着色器）的程序和能够读取这些计算的容器（材质）。

单独的材料不能执行任何操作。如果它没有着色器，它就不会知道它应该如何渲染，同样，如果它不是通过材质，着色器就无法应用于对象，因此，材质和着色器之间的类比是“图形化的”数学计算预览。

2. 调试着色器

当我们用 C# 编写脚本时，在 Unity 中我们可以使用 Debug.Log（对象消息）函数来调试我们的程序。此功能允许我们在控制台上打印我们的代码操作，但是，此功能在 Cg 或 HLSL 中不可用。

那么我们如何调试着色器呢？为此，我们必须考虑几个因素，包括：“颜色”。在着色器中，我们经常看到三种重要的颜色，它们是白色(1, 1, 1, 1)，黑色(0, 0, 0, 1) 和洋红色(1, 0, 1, 1)。白色代表默认值，黑色代表初始化值。洋红色代表图形错误，实际上，将洋红色的资源导入 Unity 到我们的场景中是很常见的。

图66. 调试 Shder

为了解决这个概念，让我们回顾一下 ShaderLab 中的属性声明，为此，我们将重新创建一个颜色属性。

_Color (“Tint”, Color) = (1, 1, 1, 1)

在上面的代码中，属性已经默认声明为“白色”，我们可以在四维向量中证实这一点；在操作结束时 (1, 1, 1, 1)。此属性本身将在 Unity Inspector 中生成一个“颜色选择器”，其颜色和/或初始化值将为白色。

接着我们分析另一个属性。

_MainTex (“Texture”, 2D) = “white”{}

同样，这个属性本身会在 Unity Inspector 中生成一个“纹理选择器”，但是如果我们

不使用任何纹理，它的默认颜色会是什么？它的颜色为白色，我们可以在声明中确认这一点。

我们在代码中的内部向量和变量的声明中经常看到黑色，其实很常见的是将向量初始化为“0”然后添加一些操作，例如四维向量“ float4 col = 0 ”已初始化为“0”，这意味着其所有通道（RGBA）的默认值为零。

注意，白色代表一个像素的最大照明值，黑色代表其最小照明值。记住这一点很重要，因为随着我们的深入，我们将有一些操作超过最大值 (x > 1) 和最小值 (x < 0)。在这些情况下，会出现颜色饱和度，为此，我们将使用一些函数，例如“ clamp ”，这将允许我们将值限定在某个特定范围内。

为什么我们的物体有时会呈现洋红色？我们已经知道，这种颜色代表“图形错误”。基本上，当 GPU 无法执行在着色器中找到的操作时，它会返回默认的洋红色。在 Unity 中，出现这种颜色的原因主要是两个因素：

① 渲染管道尚未配置。

② 着色器在其代码中存在错误。

当我们将资产导入软件时，我们需要知道的第一件事是，我们在哪个渲染管线中工作？请注意，在 Unity 中存在三种类型的渲染管道，每种类型都有不同的配置。

假设我们在 Universal RP 中导入包含具有标准表面着色器的材质的资产，那么我们的资产将显示为洋红色，因为这种类型的着色器仅在内置 RP(Built-in RP)中受支持。为了解决这个图形错误，我们必须选择材质，转到检查器并将着色器类型更改为“ Shader Graphs ”中的类型。

一旦我们确定了我们正在使用的渲染管道的类型，我们必须继续进行着色器评估。

着色器中的错误在图形上很明显，因为对象获得了洋红色。如果我们注意控制台，我们可以找到它的定义。要解决它们，我们必须考虑一些因素：在 HLSL 和 ShaderLab 中完成指令或功能。大多数错误是由拼写错误或语法产生的。下面我们将回顾一个相当常见的：

```
// in console
Shader error in “Unlit/ simpleColor”: syntax error: unexpected token '}' at line 60 (on d3d11)
```

我们如何阅读上述声明？在这种情况下，在“Unlit”菜单中找到的名为“USB_simple_color”的着色器在代码行 60 中有错误，无法在 Direct3D 11 中编译。

现在，这是一个着色器问题吗？为此，我们将分析产生错误的代码行。

```
58      fixed4 frag (v2f i) : SV_TARGET
59      {
60          fixed4 col = tex2D(_MainTex, i.uv);
61          return col
62      }
```

根据控制台，错误是在代码行号 60 中生成的，正如我们所见，那里没有问题。我们必须问自己，发生了什么？错误是因为我们忘记关闭代码行号 61 中的操作。如果我们查看返回值，我们会发现 col 向量缺少分号 (;) 所以 GPU 认为操作继续，所以它不能编译它。

以下是我们经常看到的另一个错误：

```
Shader error in 'Unlit/simpleColor': cannot implicitly convert from 'half2' to 'half4' at line 60 (on d3d11)

half4 frag (v2f i) : SV_TARGET
```

```
{
    half4 col = tex2D(_MainTex, i.uv);
    half2 uv = 0;
    col = uv;
    return col;
}
```

在这种情况下，由于我们试图将四维向量 (col) 转换为二维向量 (uv)，因此产生了错误。col 向量是四维的，有 RGBA 或 XYZW 通道，而 uv 向量是二维的，只有两个通道（RG、RB、GB 等）的组合，因此不能从二维向量转换为四维向量。

3. 添加 URP 兼容

到目前为止，我们已经实现的大部分变量、函数和向量都适用于 Cg 和 HLSL，但是，在某些情况下，我们必须添加 URP 支持，以便我们的着色器可以编译。

在 Shader Graph 的情况下，如果我们想使用 Universal RP 通过代码创建着色器，我们必须添加一些依赖项，以便 GPU 可以读取此渲染管道，这种依赖关系可以在不同的路径中找到。

Packages / Core RP Library / ShaderLibrary

Packages / Universal RP / ShaderLibrary

我们在项目中安装 Shader Graph 时会自动包含“Core RP Library”，同样，当我们选择此渲染管道作为我们的渲染引擎时，会包含“Universal RP”包。

这两个包都有扩展名为“.hlsl”的文件，我们的程序需要这些文件在 HLSL 中编译着色器。为了理解这个概念， 我们将首先添加标签“RenderPipeline”并使其等于“UniversalRenderPipeline”，这样 GPU 就会知道这个着色器在这个渲染引擎中有兼容性。

```
Tags
{
    "RenderType”=“Transparent"
    "Queue”=“Transparent"
    "RenderPipeline"="UniversalRenderPipeline"
}
```

Universal RP 使用 HLSL 语言，因此 CGPROGRAM/ENDCG 块必须分别替换为 HLSLPROGRAM 和 ENDHLSL。

```
Pass
{
    HLSLPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    ...
    ENDHLSL
}
```

为了使我们的着色器能够完美编译并使其过程更加高效，我们必须包含依赖项“Core.hlsl”，其中包含函数、结构和其他内容，它允许程序在 Universal RP 中正常运行。Core.hlsl 本身取代了“UnityCg.cginc”，这意味着我们必须消除着色器的这种依赖性。

```
HLSLPROGRAM
#pragma vertex vert
#pragma fragment frag
```

哔哩哔哩次元游戏人生

```
// #pragma multi_compile_fog
// #include "UnityCg.cginc"

#include "Package/com.unity.render.pipelines.universal/ShaderLibrary/Core.hlsl"
...
ENDHLSL
```

与 Unity 安装中包含的 UnityCg.cginc 不同，一旦我们在项目中安装 Universal RP，就会将 Core.hlsl 作为一个包添加，因此，我们必须在项目中编写其位置的完整路径。一旦我们替换了这些依赖项，就会发生两件事。

① GPU 将无法编译雾坐标（UNITY_FOG_COORDS、UNITY_TRANSFER_FOR 和 UNITY_APPLY_FOR），因为它们包含在 UnityCg.cginc 中。

② 出于同样的原因，也无法编译 UnityObjectToClipPos 函数。

相反，我们将不得不使用包含在依赖项“SpaceTransforms.hlsl”中的函数 TransformObjectToHClip(v.vertex)，而该依赖项又包含在“Core.hlsl”中。

函数 TransformObjectToHClip 执行与 UnityObjectToClipPos 相同的操作，即将顶点的位置从对象空间转换到裁剪空间，但前者在执行过程中效率更高。

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

如果此时我们保存着色器并返回 Unity，我们可以看到在控制台中生成了“无法识别的标识符”类型的错误，这是什么原因呢？请记住，默认情况下，片元着色器阶段的 col 向量都是 fixed4 类型。所以我们可以在函数中检查它。

```
fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    // UNITY_APPLY_FOG(i.fogcoord, col);
    return col * _Color;
}

// Console
Shader error in "USB/USB_simple_color_URP": unrecognized identifier "fixed4" at line 62 (on d3d11)
```

正如我们已经知道的，Universal RP 不能编译“固定(fixed)”类型的变量或向量，我们将不得不使用“half 或 float”。因此，在这一点上，我们可以做两件事。

① 手动替换半固定类型的变量和向量。

② 或者包含“HLSLSupport.cginc”依赖项，为着色器编译添加辅助宏和跨平台定

哔哩哔哩次元游戏人生

义。

```
HLSLPROGRAM
#pragma vertex vert
#pragma fragment frag

// #pragma multi_compile_fog
// #include “UnityCg.cginc”

#include "HLSLSupport.cginc"
#include “Package/com.unity.render.pipelines.universal/ShaderLibrary/Core.hlsl”
...
ENDHLSL
```

一旦包含了这种依赖关系，我们就可以使用固定类型的变量和向量，因为现在我们的程序将根据其精度识别这种类型的数据，并自动将它们替换为 half 或 float。

1） UnlitShader-基础模板

PS：

```
Shader "Unlit/#NAME#"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _BaseColor("Base Color",Color)=(1,1,1,1)
    }
    SubShader
    {
        Tags
        {
            "RenderType"="Transparent"
            "Queue"="Transparent"
            "UniversalMaterialType" = "Unlit"
            "RenderPipeline"="UniversalRenderPipeline"//告诉 GPU，我们使用的是通用渲染管线
        }
        LOD 100

        //自动将代码包含在 Subshader 内部的每个 Pass 中。
        HLSLINCLUDE

            //为着色器编译添加辅助宏和跨平台定义
            #include "HLSLSupport.cginc"
            //用 Core.hlsl 取代了 UnityCG.cginc
            #include "E:/unity/Bilibili2DProject/Library/PackageCache/com.unity.render-pipelines.universal@12.1.2/ShaderLibrary/Core.hlsl"
```

哔哩哔哩次元游戏人生

```
            /*使用 UnityPerMaterial CBUFFER 来确保着色器兼容 SRP 批处理，
CBUFFER 需要包括所有公开的属性。
            但它不能包括其他未公开的变量，纹理也不需要被包括。*/
            CBUFFER_START(UnityPerMaterial)
        float4 _BaseColor;
            CBUFFER_END

            TEXTURE2D(_MainTex);
            SAMPLER(sampler_MainTex);

            struct VertexInput
            {
                float4 vertex : POSITION;//顶点的坐标信息
                float2 uv : TEXCOORD0;
            };

            struct VertexOutput
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;//裁剪空间中的顶点坐标
            };
        ENDHLSL

        Pass
        {
            Tags
            {
        //在启用 2D 渲染器时使用，而不是前向渲染器。
            "LightMode" = "Universal2D"
            }
            Blend SrcAlpha OneMinusSrcAlpha    //通用透明混合
            Cull Off  //渲染对象的两个面
            ZTest LEqual  //深度测试，绘制位于前面或相同距离的对象。
            ZWrite Off    //关闭深度写入

            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            VertexOutput vert (VertexInput v)
            {
                VertexOutput o;

                //使用函数 TransformObjectToHClip 替换 UnityObjectToClipPos，就是
```

哔哩哔哩次元游戏人生

将顶点的位置从对象空间转换到裁剪空间，前者在执行过程中效率更高

```
                o.vertex = TransformObjectToHClip(v.vertex);
                o.uv = v.uv;
                return o;
            }

            float4 frag (VertexOutput i) : SV_Target
            {
                float4 baseTex= SAMPLE_TEXTURE2D(_MainTex,sampler_MainTex,
i.uv);
                return baseTex * _BaseColor;
            }
            ENDHLSL
        }
    }
}
```

2） 属性-Property

在 Shaderlab 示例中，我们有一个 HLSLINCLUDE，它会自动将代码包含在 Subshader 内部的每个 Pass 中。

我们可以使用 UnityPerMaterial CBUFFER 来确保着色器兼容 SRP 批处理。这个 CBUFFER 需要包括所有公开的属性(与 Shaderlab 属性块中的相同)。但它不能包括其他未公开的变量，纹理也不需要被包括。

PS：

```
HLSLINCLUDE
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"

    CBUFFER_START(UnityPerMaterial)
    float4 _BaseMap_ST;
    float4 _BaseMap_TexelSize;
    float4 _BaseColor;
    //float4 _ExampleDir;
    //float _ExampleFloat;
    CBUFFER_END
ENDHLSL
```

需要注意的是_BaseMap_ST 与_BaseMap_TexelSize 是两个东西，前者是纹理的缩放与偏移，而后者代表纹理的大小。

用[MainTexture]声明一个属性时，Unity 会将其作为材质的主要纹理。出于兼容性原因，_MainTex 属性名称是保留名称。Unity 使用具有名称的属性_MainTex 作为主要纹理，即使它没有该[MainTexture]属性。

3） 结构体-Struct

在定义顶点或片元着色器功能之前，我们需要定义一些用于将数据传入和传出的结构。在 built-in 函数中，它们通常被命名为“appdata”和“v2f”（顶点到片段的缩写），而 URP 着色器则倾向于使用“Attributes”和“Varyings”。

PS：

哔哩哔哩次元游戏人生

```
struct Attributes {
    float4 positionOS     : POSITION;
    float2 uv              : TEXCOORD0;
    float4 color           : COLOR;
};
```

该属性结构将输入到顶点着色器。它允许我们使用大写字母中被称为语义的部分从网格中获取每个顶点的数据。其中包括：顶点位置（POSITION），顶点颜色（COLOR）和 UV（又称为纹理坐标）。网格具有 8 个不同的 UV 通道，可以通过 TEXCOORD0 到 TEXCOORD7 进行访问。

我们还可以通过 NORMAL 访问顶点法线，并通过 TANGENT 访问切线。在这些结构之后，通常还会看到已定义了纹理和采样器（虽然纹理位于着色器属性中，但尚未在 hlsl 中定义。其他属性包括在 CBUFFER 中）。

在 URP 中，我们使用以下内容：

PS：

```
TEXTURE2D(_BaseMap);
SAMPLER(sampler_BaseMap);
```

4） 顶点着色器-VertexShader

我们的顶点着色器做的主要事情是将网格从对象空间转换为裁剪空间，以实现在目标屏幕位置正确渲染片元/像素。

在内置着色器(built-in)中，可以使用 UnityObjectToClipPos 函数执行此操作，但是 URP 已将其重命名为 TransformObjectToHClip（可以在函数库 SpaceTransforms.hlsl 中找到）。也就是说，还有另一种方法来处理 URP 中的转换，如下所示。

PS：

```
Varyings vert(Attributes IN) {
    Varyings OUT;

    VertexPositionInputs positionInputs = GetVertexPositionInputs(IN.positionOS.xyz);
    OUT.positionCS = positionInputs.positionCS;
    // Or this :
    //OUT.positionCS = TransformObjectToHClip(IN.positionOS.xyz);

    OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
    OUT.color = IN.color;
    return OUT;
}
```

我们从 Attributes 中输入对象空间的位置，并获得一个 VertexPositionInputs 结构。

① positionWS，在世界空间中的位置

② positionVS，视图空间中的位置

③ positionCS，裁剪空间中的位置

④ positionNDC，标准化设备坐标中的位置

顶点着色器还负责将数据传递到片元。对于顶点颜色(Vertex Color)，这只是一个简单的 OUT.color = IN.color。

如果我们希望能够对纹理进行采样，则还需要传递模型的 UV（纹理坐标）。虽然我们可以做 OUT.uv = IN.uv;（假设两者均为 float2），通常会使用 TRANSFORM_TEX 宏，该宏

哔哩哔哩次元游戏人生

采用 uv 和 texture 属性名称，并应用材质检查器的偏移和平铺进行矫正（存储在“ _BaseMap” +“ _ ST”中，S 用于比例尺和 T））。此宏位于内置和 URP 中（在 core / ShaderLibrary / Macros.hlsl 内部，应自动包含在 Core.hlsl 中）。

实际上，这只是 IN.uv.xy * _BaseMap_ST.xy + _BaseMap_ST.z 的简写，因此您也可以这样写（将_BaseMap 换成预期的纹理属性。（texture）_STfloat4 变量还必须添加到 UnityPerMaterial CBUFFER。

PS：

VertexNormalInputs normalInputs = GetVertexNormalInputs(IN.normalOS, IN.tangentOS);

GetVertexNormalInputs 可用于将对象空间的法线和切线转换为世界空间。

① normalWS，在世界空间中的法线向量。

② tangentWS，在世界空间中的切线向量。

③ bitangentWS，在世界空间中的副切线向量。

还有一个仅将法线作为输入的版本，将 tangentWS 保留为（1,0,0），bitangentWS 保留为（0,1,0），或者您也可以改用 TransformObjectToWorldNormal(IN.normalOS)。

5） 片元着色器-FragmentShader

PS：

```
half4 frag(Varyings IN) : SV_Target
{
    half4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv);

    return baseMap * _BaseColor * IN.color;
}
```

这将生成一个 Shader，该 Shader 基于_BaseMap 纹理输出一个 Half4 颜色，该 Shader 还由_BaseColor 和顶点颜色（IN.color）进行着色。

SV_Target 部分是与 half4 输出一起使用的语义，它告诉着色器它是颜色输出。

还有一个 SV_Depth 输出，它是一个浮点数，用于覆盖每个像素的 Z 缓冲区值。（可以将它们放入一个结构中以同时输出 SV_Target 和 SV_Depth）。在大多数情况下，不需要覆盖它，对于许多 GPU，它都会关闭某些基于深度缓冲区的优化，因此除非您知道自己在做什么和需要做什么，否则不要覆盖它。

我们的片元着色器使用 URP ShaderLibrary 提供的 SAMPLE_TEXTURE2D 宏对_BaseMap 纹理进行采样，该宏将纹理，采样器和 UV 作为输入。

我们可能还想做的是，如果像素的 alpha 值低于某个阈值，则将其丢弃，以使整个网格都不可见。

例如，对于四边形上的草/叶纹理。既可以在不透明着色器中也可以在透明着色器中完成此操作，通常将其称为 Alpha Clip。如果您熟悉 ShaderGraph，可以使用主节点上的“Alpha Clip Threshold”输入来处理它。

解决此问题的常用方法是提供_Cutoff 属性以控制阈值，然后执行以下操作。（此属性必须添加到我们的 Shaderlab 属性以及 UnityPerMaterial CBUFFER 中以实现 SRP Batcher 兼容性）。

PS：

```
if (_BaseMap.a < _Cutoff){
    discard;//丢弃
}
// OR
```

哔哩哔哩次元游戏人生

clip(_BaseMap.a - _Cutoff);
// inside the fragment function, before returning

环境光在 URP 下用_GlossyEnvironmentColor 获取，但得到的效果可能与 Builit-in 下的结果相差较大，这时候可以考虑用球谐函数获取

PS：

```
//URP 使用的环境光
half3 ambient = _GlossyEnvironmentColor
//使用球谐函数获取
half3 ambient = SampleSH(worldNormal);

//--Builit-in
half3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
```

4. 其他 URP 内容

1） General

General 设置可以控制管线渲染帧的核心部分。

a. Depth Texture

允许 URP 生成_CameraDepthTexture。URP 会使用这个深度贴图，它作为场景中所有摄像机的深度贴图。但是我们也可以在某个摄像机中重写这个深度贴图。

b. Opaque Texture

启用此选项可为场景中的所有摄像机都创建一个_CameraOpaqueTexture，默认设置开启。此设置的功能很像内置渲染管线中的 GrabPass。Opaque Texture 在 URP 渲染任何透明网格之前立即提供场景的快照。您可以在透明着色器中使用它来创建毛玻璃、水折射或热浪等效果。

c. Opaque Downsampling

将不透明纹理上的采样模式设置为以下选项之一：

None：使用与摄像机相同的分辨率生成不透明通道的副本。

2x Bilinear：使用双线性滤波生成二分之一分辨率图像。

4x Box：使用盒状滤波生成四分之一分辨率图像。这会产生柔和模糊的副本。

4x Bilinear：使用双线性滤波生成四分之一分辨率图像。

d. Terrain Holes

如果禁用此选项，URP 会在您针对 Unity Player 进行构建时移除所有地形孔洞着色器变体，从而减少构建时间。

2） Shader Include 头文件

Core	Packages/com.unity.render-

哔哩哔哩次元游戏人生

	pipelines.universal/ShaderLibrary/Core.hlsl
Light	Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl
Shadows	Packages/com.unity.render-pipelines.universal/ShaderLibrary/Shadows.hlsl
Surface Shaders	

3） 光照

Universal RP 不支持表面着色器，但是 ShaderLibrary 确实提供了帮助我们处理大量光照计算的功能，其包含在 Lighting.hlsl 中。

在 Lighting.hlsl 中，有一个 GetMainLight 函数，如果您熟悉着色器图中的自定义照明，您可能已经知道。为了使用此功能，我们首先在 HLSLPROGRAM 的顶部引用 Lighting.hlsl 文件，我还将添加一些 multi_compile 指令，这些指令提供了接收阴影所需的关键字。

PS：

```
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
#pragma multi_compile _ _SHADOWS_SOFT

#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

接下来，我们将需要顶点法线来处理阴影或者光照，因此我们将它们添加到 Attributes(appdata)和 Varyings(v2f)结构中，并更新顶点着色器。

PS：

```
struct Attributes {
    ...
    float4 normalOS      : NORMAL;
};

struct Varyings {
    ...
    float3 normalWS       : NORMAL;
    float3 positionWS     : TEXCOORD2;
};
...
Varyings vert(Attributes IN) {
    Varyings OUT;
    VertexPositionInputs positionInputs = GetVertexPositionInputs(IN.positionOS.xyz);
    ...
    OUT.positionWS = positionInputs.positionWS;

    VertexNormalInputs normalInputs = GetVertexNormalInputs(IN.normalOS.xyz);
    OUT.normalWS = normalInputs.normalWS;

    return OUT;
```

哔哩哔哩次元游戏人生

```
}
```

在片元着色器中，我们现在可以采用世界空间法线，并使用世界空间位置来计算阴影坐标。

PS：

```
half4 frag(Varyings IN) : SV_Target {
    half4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv);
    half4 color = baseMap * _BaseColor * IN.color;

    float4 shadowCoord = TransformWorldToShadowCoord(IN.positionWS.xyz);
    Light light = GetMainLight(shadowCoord);

    half3 diffuse = LightingLambert(light.color, light.direction, IN.normalWS);

    return half4(color.rgb * diffuse * light.shadowAttenuation, color.a);
}
```

虽然我们的着色器将从其他着色器接收阴影，但是请注意，它没有 ShadowCaster 传递，因此不会将阴影投射到自身或其他对象上。

如果我们需要阴影，但对象上没有漫反射阴影，则也可以删除漫反射阴影计算，而只需使用 light.shadowAttenuation。如果要进一步扩展以包括环境/烘焙 GI 和其他光源，请以 Lighting.hlsl 中的 UniversalFragmentBlinnPhong 方法为例，或者让它为您处理照明。它使用 InputData 结构，PBR 示例也将使用该结构。

a. 光照模式标签-LightMode Tag

PS：

```
Tags { "LightMode" = "UniversalForward" }
```

模式	描述
UniversalForward	用于前向渲染
ShadowCaster	用于投射阴影
DepthOnly	似乎在为场景视图渲染深度纹理时使用，而不是在运行中使用吗？不过，某些渲染器功能可能会使用它。
Meta	仅在光照贴图烘焙期间使用
Universal2D	在启用 2D 渲染器时使用，而不是前向渲染器。
UniversalGBuffer	与延迟渲染有关

可以在子着色器中定义多个 Pass，但是每一个 Pass 都应该用一个特定的 LightMode 标记。URP 使用了单通道前向渲染器，所以只有第一个 Pass“UniversalForward”通道(GPU 支持的)将用于渲染对象——不能同时渲染多个对象。虽然我们可以让其他传递没有标记，但要注意它们将中断 SRP 批处理程序的批处理。相反，我们建议使用单独的着色器或者材质，无论是在单独的 MeshRenderers 上，还是使用 Forward Renderer 上的 Render Objects 特性，用一个 overrideMaterial 在一个特定的图层上重新渲染对象。

哔哩哔哩次元游戏人生

b. PBR 光照

基于物理的渲染（PBR）是 Unity 的“Standard”着色器使用的着色/光照模型，以及 UPR 的“ Lit”着色器和 ShaderGraph 中的 PBR 主节点。

如前面所述，内置管线中的阴影/光照通常由 Surface Shaders 处理，其中“Standard”选用是 PBR 模型。它们使用了一个曲面函数，该函数输出了反照率，法线，发射，平滑度，遮挡，Alpha 和 Metallic（如果使用“ StandardSpecular”工作流程，则为 Specular）。Unity 将采用这些并在幕后生成一个顶点和片段着色器，为您处理某些计算，例如 PBR 阴影/照明和阴影。

Universal RP 不支持表面着色器，但是 ShaderLibrary 确实提供了帮助我们处理大量光照计算的功能。这些包含在 Lighting.hlsl 中。

PS：

```
half4 UniversalFragmentPBR(InputData inputData, half3 albedo, half metallic, half3 specular, half smoothness, half occlusion,    half3 emission, half alpha)

// 在 v10.xx 中添加了带有 SurfaceData 结构的版本
// 对于之前的版本，需要改用以上版本。
//（但是您仍然可以使用 SurfaceData 结构来组织/保存数据）
half4 UniversalFragmentPBR(InputData inputData, SurfaceData surfaceData)

// 还有:
half4 UniversalFragmentBlinnPhong(InputData inputData, half3 diffuse, half4 specularGloss, half smoothness, half3 emission, half alpha)
//复制 Unity v4 之前的“旧”表面着色器，
//并由 URP 的“ SimpleLit”着色器使用
//使用 Lambert（漫反射）和 BlinnPhong（镜面反射）照明模型
```

首先，我们应该添加 PBR 光照模型使用的一些属性。我省去了金属/高光贴图和遮挡贴图，主要是因为它们没有很好的功能来为您处理采样（除非您从 LitInput.hlsl 中复制它们，这是 URP 提供的 Lit shader 的一部分），而不是实际的 ShaderLibrary），并且此部分已经相当长且足够复杂。实际上我几乎无法解释，因为它主要是知道在哪里使用哪个函数。您以后总是可以使用 LitInput 作为示例来添加它们。

PS：

```
Properties {
    _BaseMap ("Base Texture", 2D) = "white" {}
    _BaseColor ("Example Colour", Color) = (0, 0.66, 0.73, 1)
    _Smoothness ("Smoothness", Float) = 0.5

    [Toggle(_ALPHATEST_ON)] _EnableAlphaTest("Enable Alpha Cutoff", Float) = 0.0
    _Cutoff ("Alpha Cutoff", Float) = 0.5

    [Toggle(_NORMALMAP)] _EnableBumpMap("Enable Normal/Bump Map", Float) = 0.0
    _BumpMap ("Normal/Bump Texture", 2D) = "bump" {}
    _BumpScale ("Bump Scale", Float) = 1
```


哔哩哔哩次元游戏人生

```
        [Toggle(_EMISSION)] _EnableEmission("Enable Emission", Float) = 0.0
        _EmissionMap ("Emission Texture", 2D) = "white" {}
        _EmissionColor ("Emission Colour", Color) = (0, 0, 0, 0)
        }
    ...
    // And need to adjust the CBUFFER to include these too
    CBUFFER_START(UnityPerMaterial)
        float4 _BaseMap_ST; // Texture tiling & offset inspector values
        float4 _BaseColor;
        float _BumpScale;
        float4 _EmissionColor;
        float _Smoothness;
        float _Cutoff;
    CBUFFER_END
```

我们还需要对 Unlit 着色器代码进行大量更改，包括添加一些 multi_compile 和 shader_features 以及对 Attributes 和 Varyings 结构进行调整，因为我们需要来自网格的法线和切线数据并将其发送到片元中以便使用它们用于照明计算。

“属性”块中的这些 TOGGLE 特性使我们能够从材质检查器启用/禁用 shader_feature 关键字。（或者，我们可以为着色器编写自定义编辑器/检查器 GUI 或使用调试检查器）。

如果要支持烘焙的光照贴图，我们还需要在 TEXCOORD1 通道中传递光照贴图的 UV。我还使用了来自 ShaderLibrary 的 SurfaceInput.hlsl 来帮助完成某些事情，它可以帮助 SurfaceData 结构保存 PBR 所需的数据以及一些用于采样的反射率，法线和发射贴图的函数（请注意，该结构似乎已经移动了到 URP v10 中的 SurfaceData.hlsl，但 SurfaceInput.hlsl 会自动包含它）

PS：

```
    // Material Keywords
    #pragma shader_feature _NORMALMAP
    #pragma shader_feature _ALPHATEST_ON
    #pragma shader_feature _ALPHAPREMULTIPLY_ON
    #pragma shader_feature _EMISSION
    //#pragma shader_feature _METALLICSPECGLOSSMAP
    //#pragma shader_feature _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    //#pragma shader_feature _OCCLUSIONMAP

    //#pragma shader_feature _SPECULARHIGHLIGHTS_OFF
    //#pragma shader_feature _ENVIRONMENTREFLECTIONS_OFF
    //#pragma shader_feature _SPECULAR_SETUP
    #pragma shader_feature _RECEIVE_SHADOWS_OFF

    // URP Keywords
    #pragma multi_compile _ _MAIN_LIGHT_SHADOWS
    #pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
    #pragma multi_compile _ _ADDITIONAL_LIGHTS_VERTEX _ADDITIONAL_LIGHTS
```

哔哩哔哩次元游戏人生

```
#pragma multi_compile _ _ADDITIONAL_LIGHT_SHADOWS
#pragma multi_compile _ _SHADOWS_SOFT
#pragma multi_compile _ _MIXED_LIGHTING_SUBTRACTIVE

// Unity defined keywords
#pragma multi_compile _ DIRLIGHTMAP_COMBINED
#pragma multi_compile _ LIGHTMAP_ON
#pragma multi_compile_fog

// Some added includes, required to use the Lighting functions
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
// And this one for the SurfaceData struct and albedo/normal/emission sampling functions.
// Note : It also defines the _BaseMap, _BumpMap and _EmissionMap textures for us, so we should use these as Shaderlab Properties too.
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/SurfaceInput.hlsl"

struct Attributes {
    float4 positionOS     : POSITION;
    float3 normalOS       : NORMAL;
    float4 tangentOS      : TANGENT;
    float4 color          : COLOR;
    float2 uv             : TEXCOORD0;
    float2 lightmapUV     : TEXCOORD1;
};

struct Varyings {
    float4 positionCS                    : SV_POSITION;
    float4 color                         : COLOR;
    float2 uv                            : TEXCOORD0;
    DECLARE_LIGHTMAP_OR_SH(lightmapUV, vertexSH, 1);
    // Note this macro is using TEXCOORD1
#ifdef REQUIRES_WORLD_SPACE_POS_INTERPOLATOR
    float3 positionWS                    : TEXCOORD2;
#endif
    float3 normalWS                      : TEXCOORD3;
#ifdef _NORMALMAP
    float4 tangentWS                     : TEXCOORD4;
#endif
    float3 viewDirWS                     : TEXCOORD5;
    half4 fogFactorAndVertexLight        : TEXCOORD6;
    // x: fogFactor, yzw: vertex light
#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    float4 shadowCoord                   : TEXCOORD7;
#endif
```

哔哩哔哩次元游戏人生

```
};

//TEXTURE2D(_BaseMap);
//SAMPLER(sampler_BaseMap);
// Removed, since SurfaceInput.hlsl now defines the _BaseMap for us
```

我们的“变量”现在还包含正在使用的光照贴图 UV，法线和切线，但是我们还添加了“视图方向”，这对于照明计算，雾，顶点照明支持和接收阴影的阴影坐标是必不可少的。

PS：

```
#if SHADER_LIBRARY_VERSION_MAJOR < 9
    // This function was added in URP v9.x.x versions
    // If we want to support URP versions before, we need to handle it instead.
    // Computes the world space view direction (pointing towards the viewer).
    float3 GetWorldSpaceViewDir(float3 positionWS) {
        if (unity_OrthoParams.w == 0) {
            // Perspective
            return _WorldSpaceCameraPos - positionWS;
        } else {
            // Orthographic
            float4x4 viewMat = GetWorldToViewMatrix();
            return viewMat[2].xyz;
        }
    }
#endif

Varyings vert(Attributes IN) {
    Varyings OUT;

    // Vertex Position
    VertexPositionInputs positionInputs = GetVertexPositionInputs(IN.positionOS.xyz);
    OUT.positionCS = positionInputs.positionCS;
#ifdef REQUIRES_WORLD_SPACE_POS_INTERPOLATOR
    OUT.positionWS = positionInputs.positionWS;
#endif
    // UVs & Vertex Colour
    OUT.uv = TRANSFORM_TEX(IN.uv, _BaseMap);
    OUT.color = IN.color;

    // View Direction
    OUT.viewDirWS = GetWorldSpaceViewDir(positionInputs.positionWS);

    // Normals & Tangents
    VertexNormalInputs normalInputs = GetVertexNormalInputs(IN.normalOS,
IN.tangentOS);
```

哔哩哔哩次元游戏人生

```
        OUT.normalWS =    normalInputs.normalWS;
    #ifdef _NORMALMAP
        real sign = IN.tangentOS.w * GetOddNegativeScale();
        OUT.tangentWS = half4(normalInputs.tangentWS.xyz, sign);
    #endif

        // Vertex Lighting & Fog
        half3 vertexLight = VertexLighting(positionInputs.positionWS,
normalInputs.normalWS);
        half fogFactor = ComputeFogFactor(positionInputs.positionCS.z);
        OUT.fogFactorAndVertexLight = half4(fogFactor, vertexLight);

        // Baked Lighting & SH (used for Ambient if there is no baked)
        OUTPUT_LIGHTMAP_UV(IN.lightmapUV, unity_LightmapST, OUT.lightmapUV);
        OUTPUT_SH(OUT.normalWS.xyz, OUT.vertexSH);

        // Shadow Coord
    #ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
        OUT.shadowCoord = GetShadowCoord(positionInputs);
    #endif
        return OUT;
    }
```

现在，我们还可以更新该片元着色器以实际使用 UniversalFragmentPBR 函数。由于它需要 InputData 结构输入，因此我们需要创建和设置它。代替在片元着色器中执行此操作，我们将创建另一个函数来帮助组织事物。

类似地，要处理所有反照率、金属、镜面、平滑度、遮挡、自发光(Emission)和 Alpha 输入，我们将使用 SurfaceData 结构（由我们之前包含的 SurfaceInput.hlsl 提供），并创建另一个函数来处理它。

PS：

```
    InputData InitializeInputData(Varyings IN, half3 normalTS){
        InputData inputData = (InputData)0;

    #if defined(REQUIRES_WORLD_SPACE_POS_INTERPOLATOR)
        inputData.positionWS = IN.positionWS;
    #endif

        half3 viewDirWS = SafeNormalize(IN.viewDirWS);
    #ifdef _NORMALMAP
        float sgn = IN.tangentWS.w; // should be either +1 or -1
        float3 bitangent = sgn * cross(IN.normalWS.xyz, IN.tangentWS.xyz);
        inputData.normalWS = TransformTangentToWorld(normalTS,
half3x3(IN.tangentWS.xyz, bitangent.xyz, IN.normalWS.xyz));
    #else
        inputData.normalWS = IN.normalWS;
```

哔哩哔哩次元游戏人生

```
    #endif

        inputData.normalWS = NormalizeNormalPerPixel(inputData.normalWS);
        inputData.viewDirectionWS = viewDirWS;

    #if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
        inputData.shadowCoord = IN.shadowCoord;
    #elif defined(MAIN_LIGHT_CALCULATE_SHADOWS)
        inputData.shadowCoord = TransformWorldToShadowCoord(inputData.positionWS);
    #else
        inputData.shadowCoord = float4(0, 0, 0, 0);
    #endif

        inputData.fogCoord = IN.fogFactorAndVertexLight.x;
        inputData.vertexLighting = IN.fogFactorAndVertexLight.yzw;
        inputData.bakedGI = SAMPLE_GI(IN.lightmapUV, IN.vertexSH,
inputData.normalWS);
        return inputData;
    }

    SurfaceData InitializeSurfaceData(Varyings IN){
        SurfaceData surfaceData = (SurfaceData)0;
        // Note, we can just use SurfaceData surfaceData; here and not set it.
        // However we then need to ensure all values in the struct are set before returning.
        // By casting 0 to SurfaceData, we automatically set all the contents to 0.

        half4 albedoAlpha = SampleAlbedoAlpha(IN.uv, TEXTURE2D_ARGS(_BaseMap,
sampler_BaseMap));
        surfaceData.alpha = Alpha(albedoAlpha.a, _BaseColor, _Cutoff);
        surfaceData.albedo = albedoAlpha.rgb * _BaseColor.rgb * IN.color.rgb;

        // Not supporting the metallic/specular map or occlusion map
        // for an example of that see : https://github.com/Unity-
Technologies/Graphics/blob/master/com.unity.render-pipelines.universal/Shaders/LitInput.hlsl

        surfaceData.smoothness = _Smoothness;
        surfaceData.normalTS = SampleNormal(IN.uv, TEXTURE2D_ARGS(_BumpMap,
sampler_BumpMap), _BumpScale);
        surfaceData.emission = SampleEmission(IN.uv, _EmissionColor.rgb,
TEXTURE2D_ARGS(_EmissionMap, sampler_EmissionMap));
        surfaceData.occlusion = 1;
        return surfaceData;
    }
```

哔哩哔哩次元游戏人生

```
half4 frag(Varyings IN) : SV_Target {
    SurfaceData surfaceData = InitializeSurfaceData(IN);
    InputData inputData = InitializeInputData(IN, surfaceData.normalTS);

    // In URP v10+ versions we could use this :
    // half4 color = UniversalFragmentPBR(inputData, surfaceData);

    // But for other versions, we need to use this instead.
    // We could also avoid using the SurfaceData struct completely, but it helps to organise things.
    half4 color = UniversalFragmentPBR(inputData, surfaceData.albedo, surfaceData.metallic,
        surfaceData.specular, surfaceData.smoothness, surfaceData.occlusion,
        surfaceData.emission, surfaceData.alpha);

    color.rgb = MixFog(color.rgb, inputData.fogCoord);

    // color.a = OutputAlpha(color.a);
    // Not sure if this is important really. It's implemented as :
    // saturate(outputAlpha + _DrawObjectPassData.a);
    // Where _DrawObjectPassData.a is 1 for opaque objects and 0 for alpha blended.
    // But it was added in URP v8, and versions before just didn't have it.
    // And I'm writing thing for v7.3.1 currently
    // We could still saturate the alpha to ensure it doesn't go outside the 0-1 range though :
    color.a = saturate(color.a);

    return color;
}
```

当前，虽然我们的着色器可以接收阴影，但它不包含 ShadowCaster 传递，因此不会投射任何阴影。这将在下一部分中处理。

c. ShadowCaster

如果我们希望着色器投射阴影，则需要通过标签“ LightMode” =“ ShadowCaster”的传递。可以在“Unlit”和“Lit”着色器上进行此操作，但要注意，尽管它们会投射阴影，但如果您不在 UniversalForwardPass 中处理阴影，它们将不会接收阴影。

除了使用 UsePass 时（在 Shaderlab 部分中已讨论过）。尽管我们可以使用其他着色器中的阴影投射器，例如 UsePass“Universal Render Pipeline/Lit/ShadowCaster”，但由于该着色器中使用的 CBUFFER 可能不同，因此 SRP Batcher 兼容性可能会丢失。

相反，您应该自己定义这些 Pass，有一个取巧的解决方法，我们可以执行以下操作。

PS：

```
Pass {
    Name "ShadowCaster"
    Tags { "LightMode"="ShadowCaster" }
```


哔哩哔哩次元游戏人生

```
        ZWrite On
        ZTest LEqual

        HLSLPROGRAM
        // Required to compile gles 2.0 with standard srp library
        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x gles
        //#pragma target 4.5

        // Material Keywords
        #pragma shader_feature _ALPHATEST_ON
        #pragma shader_feature _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A

        // GPU Instancing
        #pragma multi_compile_instancing
        #pragma multi_compile _ DOTS_INSTANCING_ON

        #pragma vertex ShadowPassVertex
        #pragma fragment ShadowPassFragment

        #include "Packages/com.unity.render-
pipelines.core/ShaderLibrary/CommonMaterial.hlsl"
        #include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/SurfaceInput.hlsl"
        #include "Packages/com.unity.render-
pipelines.universal/Shaders/ShadowCasterPass.hlsl"

        ENDHLSL
    }
```

我们使用了 ShadowCasterPass.hlsl 中的函数，意味着定义该 Pass 较为容易，但是它需要使用_BaseMap，_BaseColor 和_Cutoff 属性，我们也需要将它们添加到 UnityPerMaterial CBUFFER 中。阴影投射器中的 fragment 函数仅在需要阴影的位置返回 0，并丢弃不应有阴影的像素（请注意，仅在启用了_ALPHATEST_ON 关键字的情况下才会发生裁剪）

如果我们的常规着色器通道也进行顶点位移，则也需要将其添加到 ShadowCaster 通道中，以便正确投射位移的阴影。为了解决这个问题，我们要么将 ShadowCasterPass 的内容复制到我们的过程中，要么只是定义一个新的顶点函数并交换#pragma 顶点 ShadowPassVertex。

PS：

```
#pragma vertex vert

...

// function copied from ShadowCasterPass and edited slightly.
```

哔哩哔哩次元游戏人生

```
Varyings vert(Attributes input) {
    Varyings output;
    UNITY_SETUP_INSTANCE_ID(input);

    // Example Displacement
    input.positionOS += float4(0, _SinTime.y, 0, 0);

    output.uv = TRANSFORM_TEX(input.texcoord, _BaseMap);
    output.positionCS = GetShadowPositionHClip(input);
    return output;
}
```

d. DepthOnly

着色器还应包含标记为“ LightMode” =“ DepthOnly”的过程。此过程与 ShadowCaster 非常相似，但没有阴影偏差偏移。我不完全确定 URP 中使用 DepthOnly 传递的用途。场景视图似乎在渲染深度纹理时使用了它（由 ShaderGraph 中的“Scene Depth”节点使用），而“游戏视图”深度纹理在没有此传递的情况下似乎可以正常工作。但是，可能还有其他一些东西，例如自定义渲染功能（用于前向渲染器）依赖于 DepthOnly 传递。

我们可以以类似的方式处理 DepthOnly 传递，但有一些细微差异：

PS：

```
Pass {
    Name "DepthOnly"
    Tags { "LightMode"="DepthOnly" }

    ZWrite On
    ColorMask 0

    HLSLPROGRAM
    // Required to compile gles 2.0 with standard srp library
    #pragma prefer_hlslcc gles
    #pragma exclude_renderers d3d11_9x gles
    //#pragma target 4.5

    // Material Keywords
    #pragma shader_feature _ALPHATEST_ON
    #pragma shader_feature _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A

    // GPU Instancing
    #pragma multi_compile_instancing
    #pragma multi_compile _ DOTS_INSTANCING_ON

    #pragma vertex DepthOnlyVertex
    #pragma fragment DepthOnlyFragment
```

哔哩哔哩次元游戏人生

```
        #include "Packages/com.unity.render-pipelines.core/ShaderLibrary/CommonMaterial.hlsl"
        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/SurfaceInput.hlsl"
        #include "Packages/com.unity.render-pipelines.universal/Shaders/DepthOnlyPass.hlsl"

        // Again, using this means we also need _BaseMap, _BaseColor and _Cutoff shader properties
        // Also including them in cbuffer, except _BaseMap as it's a texture.

        ENDHLSL
    }
```

这次使用 Unity 的 URP 着色器提供的 DepthOnlyPass 。同样，如果需要顶点位移，我们应该将 DepthOnlyVertex 函数复制到我们的代码中，将其重命名为 vert，然后像上面的 ShadowCaster 示例中一样添加位移代码。

4） GetVertexNormalInputs

用于将对象空间的法线和切线转换为世界空间。

PS：

VertexNormalInputs normalInputs = GetVertexNormalInputs(IN.normalOS, IN.tangentOS);

内容	描述
normalWS	世界空间中的法线向量
tangentWS	世界空间中的切线向量
bitangentWS	世界空间中的副切线向量

还有一个仅将法线作为输入的版本，将 tangentWS 保留为（1,0,0），bitangentWS 保留为（0,1,0），或者也可以改用 TransformObjectToWorldNormal(IN.normalOS)。

5） 着色器变体- Variants[5]

在着色器中，我们可以指定更多的#pragma 指令，其中一些指令包括 multi_compile 和 shader_feature。URP 支持变体，你可能需要某些额外功能，可在#pragma 添加以下关键字

a. multi_compile

PS：

#pragma multi_compile _A _B _C (...etc)

在此示例中，我们将生成着色器的三个变体，其中_A，_B 和_C 是关键字。

PS：

```
#ifdef _A
// 如果 A 启用，编译此代码
#endif

#ifndef _B
// 当 B 被禁用时编译此代码，也就是只在 A 和 C 中。
```

[5] n. 变种，变形；（单词拼写或读音的）变异，变体　adj. 不同的，变体的，变种的

哔哩哔哩次元游戏人生

// 注意#ifndef 中额外的“n”表示“如果没有定义”
#else
// 如果 B 启用，编译此代码
#endif

#if defined(_A) || defined(_C)
// 用 A 或 c (aka 与上面的相同，假设没有其他关键字)编译此代码
// 如果需要多个条件，则必须使用长形式的"#if defined()"
// 其中|| = or， && = and
// 注意，因为关键字是在一个 multi_compile 语句中定义的
// 实际上不可能同时启用两者，所以&&在这里没有意义。
#endif

// 还有#elif，用于 else if 语句。

b. SHADER_FEATURE

PS：
#pragma shader_feature _A _B
这与 multi_compile 完全相同，但是未使用的变体将不包括在最终版本中。因此，在运行时启用/禁用这些关键字是不好的，因为它所需的着色器可能未包含在构建中！如果需要在运行时处理关键字，请改用 multi_compile。
这些指令还有“顶点”和“片元”版本，可用于仅针对顶点或片元程序编译着色器变体，从而减少了变体的总数。
PS：
#pragma multi_compile_vertex _ _A
#pragma multi_compile_fragment _ _B
// also shader_feature_vertex and shader_feature_fragment
在此示例中，_A 关键字仅用于顶点程序，_B 仅用于片元。不能同时启用_A 和_B 的变体。Unity 告诉我们，这会产生 2 个着色器变体，尽管当您查看实际的编译代码时，它更像是一个禁用两个着色器的着色器变体和两个“half”的变体。

c. 着色器变体的增长

每增加一个 multi_compile 和 shader_feature，它就会为启用/禁用关键字的每种可能组合生成越来越多的着色器变体。
PS：
#pragma multi_compile _A _B _C
#pragma multi_compile _D _E
#pragma shader_feature _F _G
在这里，第一行将生成 3 个着色器变体。但是第二行需要为已启用_D 或_E 的那些变体生成 2 个着色器变体。
因此，A&D，A&E，B&D，B&E，C&D 和 C&E。现在有 6 个变体。
第三行，是这 6 个中的每一个的另外 2 个变体，因此我们现在总共有 12 个着色器变

哔哩哔哩次元游戏人生

体。由于该行是 shader_feature，因此某些变体可能不会包含在构建中。

每个添加了 2 个关键字的 multi_compile 都会使产生的变体数量加倍，因此包含 10 个变体的着色器将产生 1024 个着色器变体！它需要编译最终构建中需要包含的每个着色器变体，因此将增加构建时间以及构建大小。

d. 如何查看着色器的变体个数

如果要查看一个着色器产生多少个着色器变体，请单击该着色器，然后在检查器中有一个“Compile and Show Code”按钮，旁边是一个小的下拉箭头，其中列出了所包含的变体数。如果单击“skip unused shader_features”，则可以切换以查看变体的总数。

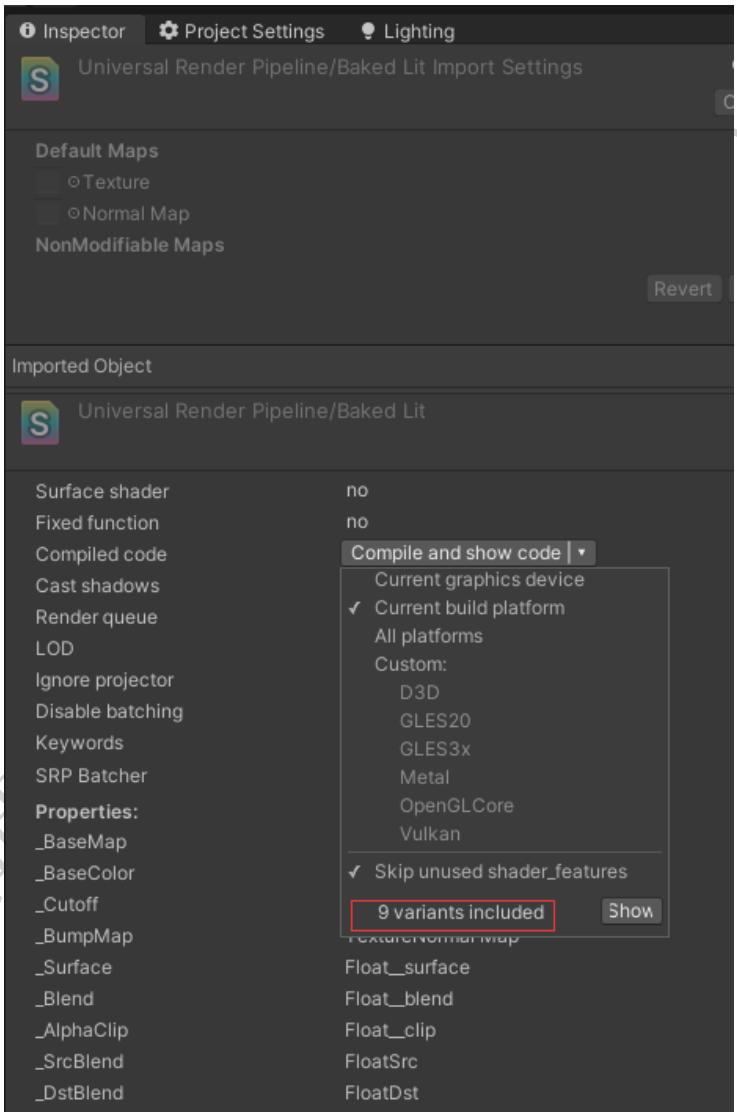


图67. 查看 Shader 变体数量

e. 关键字

每个项目最多可有 256 个关键字，因此最好遵循其他着色器的命名约定。还会需要注

意到的是，对于许多 multi_compile[6]和 shader_features 而言，第一个关键字通常仅保留为“_”。实际上，这实际上不会产生关键字，因此会为 256 个最大值的其他关键字留出更多空间。

PS：

```
#pragma multi_compile _ _KEYWORD

#pragma shader_feature _KEYWORD
// 仅是 shader_features 的简写
#pragma shader_feature _ _KEYWORD

// 如果您需要知道该关键字是否已禁用
// 然后我们可以这样做：
#ifndef _KEYWORD
// 或#if! defined（_KEYWORD）
// 或#ifdef _KEYWORD #else
// code
#endif
```

我们还可以通过使用 multi_compile 和 shader_feature 的本地版本来避免耗尽最大的关键字数。这些生成的关键字对于该着色器来说是本地的，但是每个着色器最多也只能有 64 个本地关键字。

PS：

```
#pragma multi_compile_local _ _KEYWORD
#pragma shader_feature_local _KEYWORD

// 还有 local_fragment/vertex！
#pragma multi_compile_local_fragment _ _KEYWORD
#pragma shader_feature_local_vertex _KEYWORD
```

6） 内置 Shader 辅助函数

a. 阴影贴图

必须 include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Shadows.hlsl"。

Built-in	URP
UNITY_DECLARE_SHADOWMAP（tex）	TEXTURE2D_SHADOW_PARAM（textureName，samplerName）
UNITY_SAMPLE_SHADOW（tex，uv）	SAMPLE_TEXTURE2D_SHADOW（textureName，samplerName，coord3）
UNITY_SAMPLE_SHADOW_PROJ（tex，uv）	SAMPLE_TEXTURE2D_SHADOW（textureName，samplerName，coord4.xyz/coord4.w）

[6] v. 汇编，编纂；编译（电脑程式）；收集，搜集（信息，资料）；累积（得分）

哔哩哔哩次元游戏人生

b. 贴图/采样器 声明宏

Unity 有很多纹理/采样器 用来兼容不同平台 ，这些仍然在 URP 中，但现在有了不同的名称和新增的内容。需要注意 SCREENSPACE_TEXTURE 变成了 TEXTURE2D_X。如果你想要在 VR 中（Single Pass Instanced 或 Multi-view 模式）制作屏幕效果，你必须使用 TEXTURE2D_X 定义纹理。这个宏会为你处理正确的纹理声明（是否为数组）。对这个纹理采样的时候必须使用 SAMPLE_TEXTURE2D_X，并且对 uv 使用 UnityStereoTransformScreenSpaceTex。

Built-in	URP
UNITY_DECLARE_TEX2D(name)	TEXTURE2D(textureName);SAMPLER(samplerName);
UNITY_DECLARE_TEX2D_NOSAMPLER(name)	TEXTURE2D(textureName);
UNITY_DECLARE_TEX2DARRAY(name)	TEXTURE2D_ARRAY(textureName);SAMPLER(samplerName);
UNITY_SAMPLE_TEX2D(name,uv)	SAMPLER_ TEXTURE2D(textureName, samplerName,coord2);
UNITY_SAMPLE_TEX2D_SAMPLER(name,samplername,uv)	SAMPLER_ TEXTURE2D(textureName, samplerName,coord2);
UNITY_SAMPLE_TEX2DARRAY(name,uv)	SAMPLER_ TEXTURE2D_ARRAY(textureName, samplerName,coord2,index);
UNITY_SAMPLE_TEX2DARRAY_LOD(name,uv,lod)	SAMPLER_ TEXTURE2D_ARRAY_LOD(textureName, samplerName,coord2,index,lod);

c. 顶点变换函数

需要包含 #include"Packages/com.unity.renderpipelines.core/ShaderLibrary/SpaceTransforms.hlsl"

Built-in	URP	描述	路径
float4 UnityObjectToClipPos(float3 pos)	float4 TransformObjectToHClip(float3 positionOS)		
float4 UnityObjectToViewPos (float3 pos)	TransformWorldToView (TransformObjectToWorld (positionOS))		

d. 法线转换函数

Built-in	URP	描述	路径
float4	float4		

哔哩哔哩次元游戏人生

UnityObjectToWorldNormal（float3 pos）	TransformObjectToWorldNormal（float3 normalOS）		

e. 通用辅助函数

Built-in	URP	描述	路径
float3 WorldSpaceViewDir(float4 v)	float 3 GetWorldSpaceViewDir(float3 positionWs)		include"Packages/ com.unity.render-pipelines.universal/ShaderLibrary/ ShaderVariablesFunctions.hlsl"
float3 ObjSpaceViewDIr(float4 v)	Gone.Do TransformWorldToObject(GetCameraPositionWS())		
float2 ParallaxOffset(half h,half height,half3 viewDir)			
float3 Luminance(fixed3 c)	real Luminance(real3 linearRgb)		include"Packages/ com.unity.render-pipelines.core/ShaderLibrary /Color.hlsl"
float3 DecodeLightMap(fixed4 color)	real3 DecodeLightmap(real4 encodedIlluminance,real4 decodeInstructions)		include"Packages/ com.unity.render-pipelines.core/ShaderLibrary / EntityLighting.hlsl"

f. 前向渲染(Forward)辅助函数

Built-in	URP	描述	路径
float3 WorldSpaceLightDir(float4 v)	_MainLightPosition.xyz-TransformObjectToWorld (objectSpacePosition)		Include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Input.hlsl"
float3 ObjSpaceLightDIr(float4 v)	TransformWorldToObject(_MainLightPosition.xyz)-objectSpacePosition		Include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Input.hlsl"
float2	可以使用 half3		For VertexLighting(...) include

哔哩哔哩次元游戏人生

ShaderPointLights(…)	VertexLighting(float3 positionWS, half3 normalWS)		"Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

g. 屏幕空间辅助函数

Built-in	URP	描述	路径
float4 ComputeScreenPos(float4 clipPos)	float4 ComputeScreenPos(float4 positionCS)		Include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/ShaderVariablesFunctions.hlsl"
float4 ComputeGrabScreenPos(float4 clipPos)	移除了		

h. 顶点光照辅助函数

Built-in	URP	描述	路径
float3 ShadeVertexLights(float4 vertex,float3 normal)	移除了，可以尝试使用 UNITY_LIGHTMODEL_AMBIENT.xyz + VertexLighting(...)		For VertexLighting(...) include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

i. 阴影-Shadow

"Packages/com.unity.render-pipelines.universal/ShaderLibrary/Shadows.hlsl".

Built-in	URP	描述	路径
UNITY_SHADOW_COORDS(x)			
TRANSFER_SHADOW(α)	α.shadowCoord = TransformWorldToShadowCoord(worldSpacePosition)	当启用级联时(CSM)在片元着色器应用防止	

哔哩哔哩次元游戏人生

		伪影	
SHADOWS_SCREEN	移除了，不支持。		

j. 雾-Fog

"Packages/com.unity.render-pipelines.universal/ShaderLibrary/ShaderVariablesFunctions.hlsl".

Built-in	URP	描述	路径
UNITY_FOG_COORDS(x)	移除了。DIY，例如 float fogCoord : TEXCOORD0;		
UNITY_TRANSFER_FOG(o*，outpos)	o.fogCoord = ComputeFogFactor(clipSpacePosition.z);		
UNITY_APPLY_FOG(coord，col)	color = MixFog(color，i.fogCoord);		

k. 深度-Depth

要使用相机深度纹理，需要 include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/DeclareDepthTexture.hlsl"，然后会自动声明 _CameraDepthTexture，也会包含辅助函数 SampleSceneDepth(...)和 LoadSceneDepth(...)。

Built-in	URP	描述	路径
LinearEyeDepth(sceneZ)	LinearEyeDepth(sceneZ，_ZBufferParams)		Include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"
Linear01Depth(sceneZ)	Linear01Depth(sceneZ，_ZBufferParams)		Include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"

十四、 内置数学函数

在 Cg 和 HLSL 中，我们都可以找到有助于我们编程效果的内在函数。 这些函数对应于一般的数学运算，我们根据我们希望获得的结果在特定情况下使用它们。

Abs	Exp	Length
Ceil	Exp2	Lerp
Clamp	Floor	Min
Cos	Step	Max
Sin	Smoothstep	Pow
Tan	Frac	

哔哩哔哩沃沃游戏人生

1. 绝对值-Abs

这个函数引用一个数字的绝对值，作为参数，我们可以传递一个标量值和一个向量。

PS：

```
float abs(float n)
{
    return max(-n, n);
}
float2 abs (float2 n);
float3 abs (float3 n);
float4 abs (float4 n);
```

绝对值将始终返回正数，其数学符号系统由两个框住数字的边栏组成。

$$|-3| = 3 \ ， |-4| = 4 \ ， |5| = 5$$

我们的程序可以将绝对值函数用于多种效果，包括重新创建万花筒或生成三平面映射(triplanar mapping)。事实上，对于第一种情况，我们可以通过计算 UV 坐标中的绝对值来实现这样的效果。 同时，在三平面映射中，我们可以确定网格法线的绝对值，从而在正轴和负轴上生成投影。

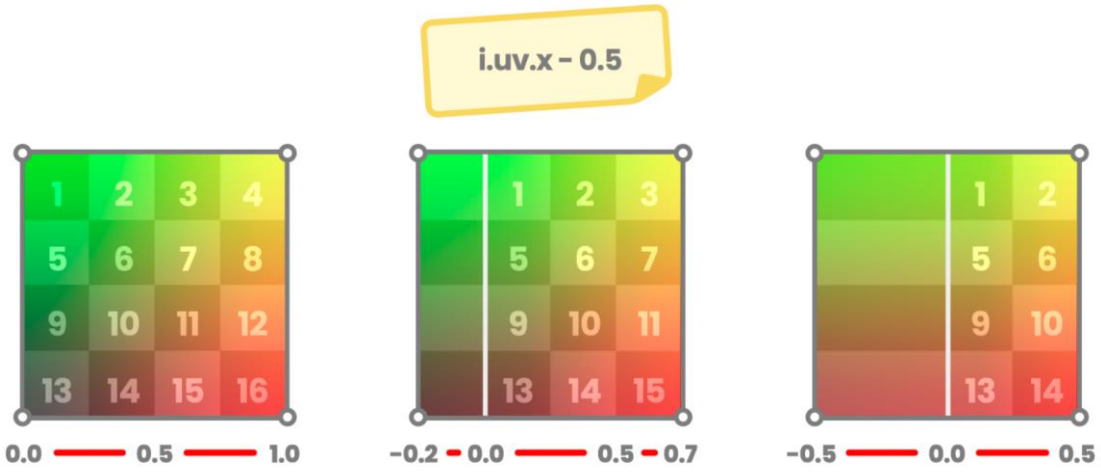


图68. UV 坐标从 0.0f 开始，到 1.0f 结束

如果我们注意上图 UV 坐标(UV coordinates)中的起点，我们会注意到 U 坐标以 Quad 为中心减去 0.5f，其最小值变为负数 -0.5f。在上图示例中，纹素被拉伸了，因为该图像的 wrap mode 已被设置为了 Clamp 模式。 在这种情况下，我们可以应用绝对值来产生“镜像”效果。

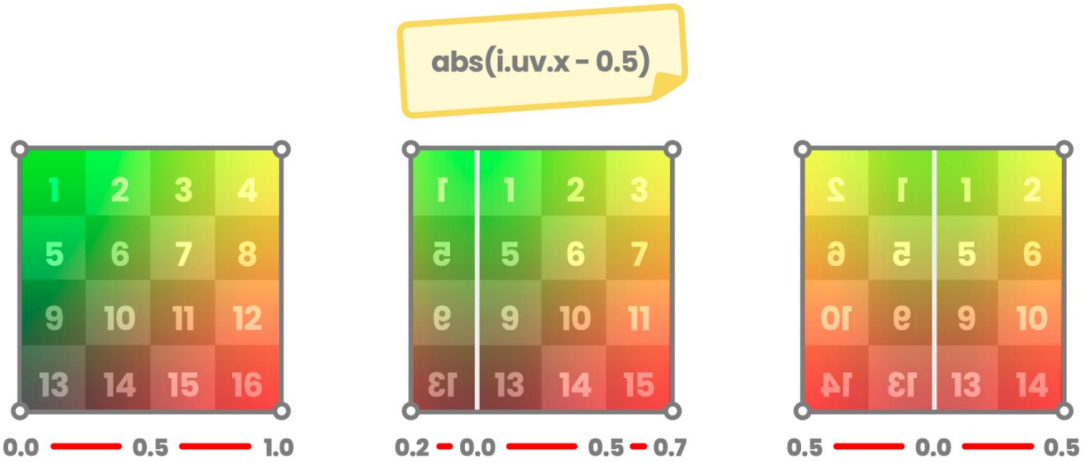


图69.　　对被减后的 UV 坐标应用绝对值

1） 万花筒效果

接下来，我们将开发万花筒(kaleidoscope)的效果来充分理解这个概念。因为我们的代码的原因，该效果只适用于 Tiling 为 1 的情况。我们将首先创建一个新的着色器类型“Unlit Shader”，我们将其称为“USB_function_ABS”。然后，我们在着色器属性中声明一个新属性，稍后我们将使用它来旋转 UV 坐标。

PS：

```
Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        // _Rotation 用于调整旋转角度
        _Rotation ("Rotation", Range(0, 360)) = 0
    }
```

由于完整的旋转有 360 度，因此_Rotation 取值范围在 0 到 360 之间。我们继续使用属性的全局变量或连接变量。

PS：

```
Pass
{
    CGPROGRAM
    …
    //这里是连接变量，属性和连接变量必须具有相同的名称
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Rotation;
    …
    ENDCG
}
```

我们可以计算 U 和 V 坐标的绝对值来生成效果，并在片元着色器阶段使用这些新值作为输出。

PS：

```
fixed4 frag (v2f i) : SV_Target
            {
```

```
            // 计算 U 坐标被减后的绝对值
            float u = abs(i.uv.x - 0.5);
            // 计算 V 坐标被减后的绝对值
            float v = abs(i.uv.y - 0.5);

            fixed col = tex2D(_MainTex, float2(u, v));
            UNITY_APPLY_FOG(i.fogCoord, col);

            return col;
        }
```

根据我们分配为_MainTex 的纹理 Wrap Mode 的不同，可能会发生两种情况。如果纹理的 Wrap Mode 设置为“Repeat”，UV 坐标的负值区域会被相同纹素重复填充。如果纹理的 Wrap Mode 设置对应于“Clamp”，则图像将被拉伸。

无论纹理的 Wrap Mode 设置是何种，其镜像效果都会很明显，因为为每个 UV 坐标都使用了 Abs 函数。如果我们想旋转 UV 坐标，我们可以使用 Unity_Rotate_Degrees_float 函数，包含在 Shader Graph 包中。

PS：

```
void Unity_Rotate_Degrees_float(float2 UV,float2 Center,float Rotation,out float2 Out)
        {
            Rotation = Rotation * (UNITY_PI/180.0f);//这里将角度换成了弧度
            UV -= Center;
            float s = sin(Rotation);
            float c = cos(Rotation);
            float2x2 rMatrix = float2x2(c, -s, s, c);
            rMatrix *= 0.5;
            rMatrix += 0.5;
            rMatrix = rMatrix * 2 - 1;
            UV.xy = mul(UV.yx, rMatrix);
            UV += Center;
            Out = UV;
        }
```

由于函数是“void”类型，我们必须在片元着色器阶段的字段中初始化一些变量，然后将它们作为参数传递。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            // 计算所有 U 坐标被减后的绝对值
            float u = abs(i.uv.x - 0.5);
            // 计算所有 V 坐标被减后的绝对值
            float v = abs(i.uv.y - 0.5);
            // 将连接变量的值赋予局部变量
            float rotation = _Rotation;
            // 我们以枢轴为中心进行旋转
            float center = 0.5;
```

哔哩哔哩次元游戏人生

```
            // 让我们为纹理生成新的 UV 坐标
            float2 uv = 0;
            Unity_Rotate_Degrees_float(float2(u,v), center, rotation, uv);
            fixed4 col = tex2D(_MainTex, uv);
            UNITY_APPLY_FOG(i.fogCoord, col);
            return col;
        }
```

Unity_Rotate_Degrees_float 函数中的第一个参数，对应于我们要旋转的 UV 坐标，第二个参数旋转的中心点或枢轴，然后是旋转度数，最后是我们将用于纹理的新 UV 的输出。

2） 为 Shader 添加 URP 管线的兼容

我们创建的 Shader，默认是用于 Build in 渲染管线的。在 URP 和 HDRP 中，我们需要告知我们所使用的渲染管线，以及所需的对应库文件。如果不这样做，我们的 Shader 将无法在 URP 和 HDRP 种运行。

2. 上限值-Ceil

根据 NVIDIA 的官方文档，Ceil 返回不小于一个标量(scalar)或每个向量分量的最小整数。这是什么意思？函数 ceil(n)将返回一个整数，即不带小数最接近其的值。例如，如果一个数字等于 0.5f，ceil 就将返回 1。

0.0f 和 1.0f 之间的所有数字都将返回 1，因为后者是不小于（即>=）其参数的最小整数值。

PS：

```
float ceil(float n)
{
    return -floor(-n);
}
```

此功能对于在游戏中生成“缩放或放大镜”效果非常有帮助。 要实现这一效果，我们只需计算 U 和 V 坐标的 ceil(n) 值，将最终值乘以 0.5，然后在 UV 的默认值和 ceil(n) 产生的值之间生成线性插值。

为了深入理解这个概念，我们将创建一个新的着色器，其类型为“Unlit Shader”，我们将其命名为 USB_function_CEIL，我们将首先在片元着色器阶段声明纹理_MainTex 的新 UV 坐标。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            // 对每一个 U 坐标取上限值
            float u = ceil(i.uv.x);
            // 对每一个 V 坐标取上限值
            float v = ceil(i.uv.y);
            // 我们为纹理分配新的 UV 坐标
            fixed4 col = tex2D(_MainTex, float2(u, v));
            UNITY_APPLY_FOG(i.fogCoord, col);
            return col;
        }
```

我们上面执行的操作会返回位置[1, 1]，这是什么原因呢？ 因为 U 和 V 坐标都从 0.0f

哔哩哔哩次元游戏人生

开始并在 1.0f 结束（最接近他们的上整数是 1），所以 ceil(n) 将只返回纹理中的最后一个纹素。 因此，它将生成从纹理的右上角到左下端的缩放效果。

1） 缩放效果

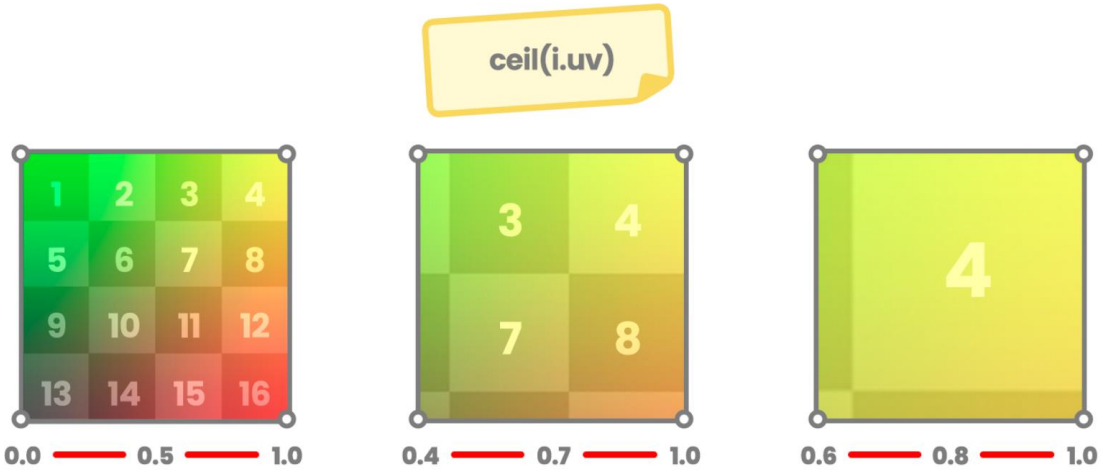


图70. 对默认 UV 坐标应用 Ceil 函数

Ceil 将返回 UV 纹理的最后一个纹素，其坐标为(1,1)。对于这种效果，我们需要一个属性，便于我们在外部去增加或减少被影响的区域。 为了实现这一效果，我们将在属性(property)中声明一个 Float 类型的变量，我们将称之为 _Zoom。

PS：

```
Properties
    {
        _MainTex ("主纹理", 2D) = "white" {}
        _Zoom ("Zoom", Range(0, 1)) = 0
    }
```

在该范围内，“0”表示缩放比例为零，“1”表示缩放比例为百分之一百。 然后我们在程序中声明连接变量。

PS：

```
Pass
{
    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Zoom;
    …
}
```

由于我们的缩放是从纹理的中心开始，我们可以通过将其乘以 0.5 来修改它的位置。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            // 对每一个 U 坐标取上限值，然后乘 0.5，取中点。
            float u = ceil(i.uv.x) * 0.5;
            // 对每一个 V 坐标取上限值
            float v = ceil(i.uv.y) * 0.5;
```

哔哩哔哩次元游戏

```
//为纹理分配新的 UV 坐标，然后乘 0.5，取中点。
fixed4 col = tex2D(_MainTex, float2(u, v));
UNITY_APPLY_FOG(i.fogCoord, col);
return col;
}
```

此时，纹理已经可以其中心放大了。 但是，我们仍然无法欣赏缩放效果，因为我们所返回 UV 坐标的依然是一个点。因此，我们将看到单一颜色填充满整个区域。 我们可以做的是在 UV 的默认值和 ceil(n)函数产生的值之间生成线性插值。

PS：

```
fixed4 frag (v2f i) : SV_Target
{
    // 对每一个 U 坐标取上限值，然后乘 0.5，取中点。
    float u = ceil(i.uv.x) * 0.5;
    // 对每一个 V 坐标取上限值
    float v = ceil(i.uv.y) * 0.5;
    float uLerp = lerp(u, i.uv.x, _Zoom);//根据_Zoom 的值，在原始值和放大值之间差值
    float vLerp = lerp(v, i.uv.y, _Zoom);
    //为纹理分配新的 UV 坐标，然后乘 0.5，取中点。
    fixed4 col = tex2D(_MainTex, float2(u, v));
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

在上面的示例中，我们为不同的坐标创建了两个名为 uLerp 和 vLerp 的变量。 我们通过属于 Cg/HLSL 语言的 lerp(x, y, t) 函数进行线性插值，lerp 函数的 T 值，我们使用了属性 _Zoom，其范围在 0.0f 和 1.0f 之间。当我们从 Unity Inspector 修改属性_Zoom 的值，我们就可以看到其是如何影响放大区域的。

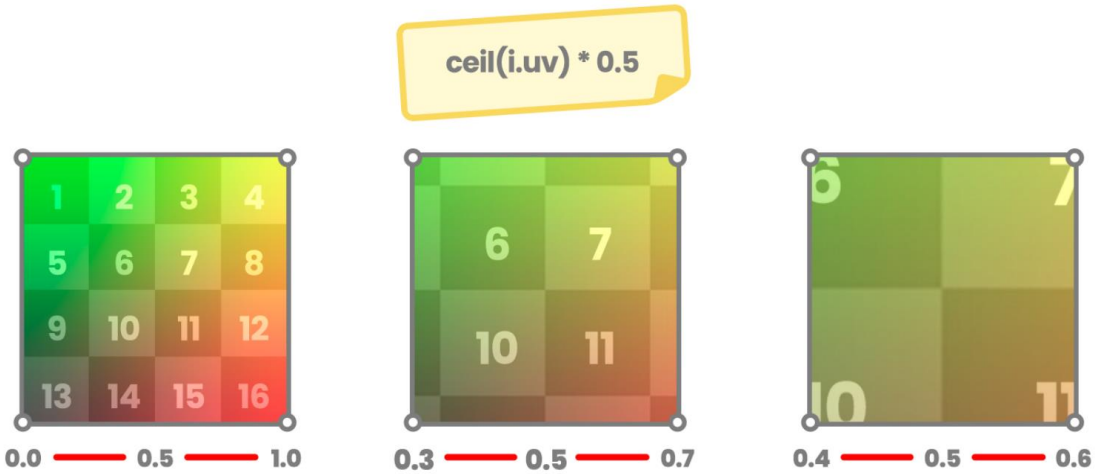


图71. 放大中心的区域

3. 限制-Clamp

当我们将结果限制在一个特点范围内时，Clamp 函数很方便。默认情况下，它允许我们设置范围的最小值和最大值，所有超过最大值的值都会变成最大值，所有超过最小值的

值都会变成最小值。在我们的开发过程中，我们经常会遇到一些导致数字小于“零”或大于“一”的操作，比如在计算网格法线与光照方向的点积时，我们可以得到一个范围介于 -1.0f 和 1.0f 之间。由于负值会在最终效果中产生颜色伪影，使用 Clamp 函数，我们可以限制和重新定义数值，使其保持在所设定的范围内。

PS：

```
float clamp (float a, float x, float b)
{
    return max(a, min(x, b));
}
```

在上面的函数中，参数“a”是指最小返回值，而参数“b”是指范围内的最大返回值，至于参数“x”对应的就是被限制范围的值。 这是什么意思？ 让我们假设“a”和“b”有一个设定范围，x 有一个可变数字。 当“x”等于 1.0f 时，如果参数“a”等于 0.1f，参数“b”等于 0.9f，则“x”的最大返回值为 0.9f； 为什么？ 因为“b”定义了值的最高限值。在相反的情况下，当“x”等于 0.0f 时，返回值为 0.1f，因为“a”定义了“x”的最小返回值。

1） 调整图像明暗度

为了深入理解这个概念，我们将创建一个新的着色器，其类型为“Unlit Shader”，并将其命名为 USB_function_CLAMP，我们将声明三个 Float 类型的属性。

PS：

```
Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Tvalue ("T", Range(0, 1)) = 0
        _Minvalue ("Min", Range(0, 1)) = 0
        _Maxvalue ("Max", Range(0, 1)) = 0
    }
```

然后需要声明全局或连接变量，以便在程序中使用这些属性。

PS：

```
Pass
{
    …
    //这里是连接变量，属性和连接变量必须具有相同的名称
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Tvalue;
    float _Minvalue;
    float _Maxvalue;
    …
}
```

我们将使用刚刚创建的 Shader 来增加或减少主纹理的伽马颜色。 因此，我们可以做两件事，一件是在我们的程序中自己构建一个简单的函数，将一个值限制在一个范围内。二是使用 Cg/HLSL 语言中包含的函数“clamp”。

在这里，我们将自己构建一个 Clamp 函数，来对值进行一个限制。为此，我们将自己定位在顶点着色器和片元着色器阶段之间，并编写一个我们称为“ourClamp”的新方法。

哔哩哔哩次元游戏人生

PS：

```
v2f vert(appdata v) { … }
float ourClamp(float a, float x, float b)
{
 return max(a, min(x, b));
}
fixed4 frag(v2f i) : SV_Target { … }
```

正如我们在函数中看到的，ourClamp 将 float t 限制在 float a 和 float b 之间。 然后，在片元着色器阶段，我们创建一个名为“darkness”的 Float 变量，并将 ourClamp 函数的返回值赋予它。为了使用 ourClamp 函数，我们需要按照参数(arguments[7])的顺序，去正确的填写我们的连接变量。

PS：

```
float ourClamp(float a, float x, float b) { … }
fixed4 frag (v2f i) : SV_Target
        {
            float darkness = ourClamp(_Minvalue, _Tvalue, _Maxvalue);
            /采样纹理
            fixed4 col = tex2D(_MainTex, i.uv);
            //应用雾效
            UNITY_APPLY_FOG(i.fogCoord, col);
            return col;
        }
```

darkness 是 float 类型，这意味着它只有一维。 因此，将 col 向量乘以 darkness 将影响向量 col (RGBA)的四个通道（即矩阵和数值相乘，等于数乘以矩阵中的每一个数）。

PS：

```
float ourClamp(float a, float x, float b) { … }
fixed4 frag (v2f i) : SV_Target
        {
            float darkness = ourClamp(_Minvalue, _Tvalue, _Maxvalue);
            /数乘矩阵，等于数乘以矩阵当中的每一个数
            fixed4 col = tex2D(_MainTex, i.uv) * darkness;
            //应用雾效
            UNITY_APPLY_FOG(i.fogCoord, col);
            return col;
        }
```

我们可以通过在 Cg/HLSL 语言中包含的 “clamp”函数来简化上述操作。

PS：

```
fixed4 frag(v2f i) : SV_Target
{
    // Cg and HLSL include the clamp function.
    float darkness = clamp(_Minvalue, _Tvalue, _Maxvalue);
    fixed4 col = tex2D(_MainTex, i.uv) * darkness;
    UNITY_APPLY_FOG(i.fogCoord, col);
```

[7] n. 参数

哔哩哔哩东无游戏人生

```
        return col;
    }
```

4. 正弦和余弦函数-Sin and Cos

这些三角函数指的是角度的正弦和余弦，余弦是邻边和斜边的比值，正弦是对边和斜边的比值。我们可以在标量值和向量上同时使用“cos 和 sin”。

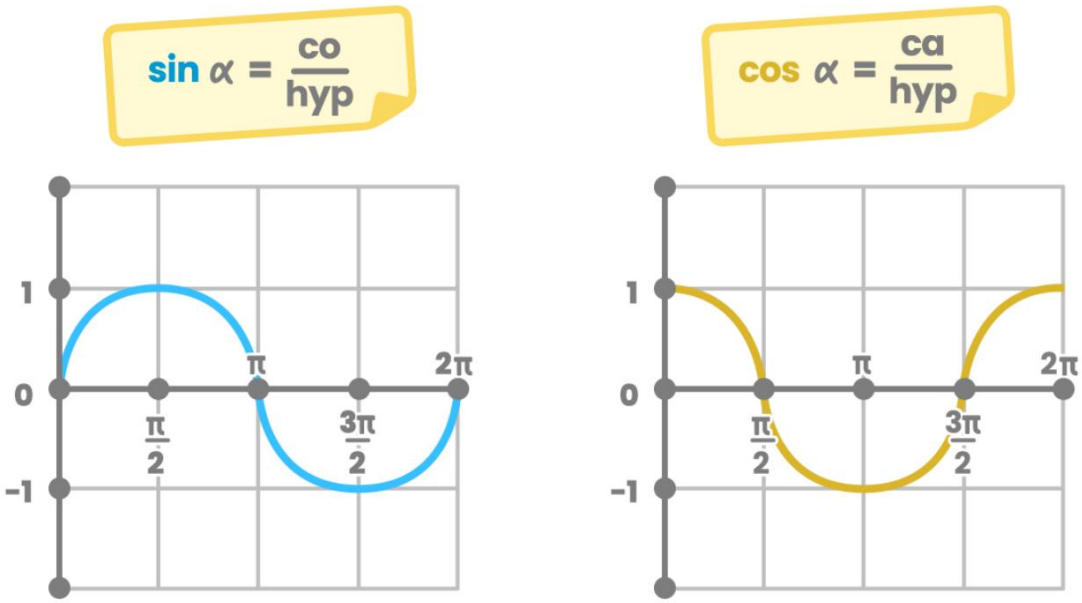


图72. 笛卡尔平面上函数的图形表示。

上图的左侧是正弦函数对应的图像，右侧则是余弦函数对应的图像。Cg/HLSL 语言中也包含的有这些函数，在计算机图形学中它们非常有用。有了它们，我们可以生成多个几何图形甚至矩阵变换，一个实际示例是对象中顶点的旋转。

我们已经知道，一个顶点具有三个空间坐标（XYZ），它们被视为一个向量（矢量）。鉴于其性质，我们可以转换这些值并从矩阵生成旋转。

假设我们想在二维空间中旋转一个顶点，我们在其“Y”轴上应用函数“sin”将获得从上到下的波动，在其“X”轴上使用“cos”函数将再现圆周运动。旋转主要是 sin 和 cos 的时间差造成。

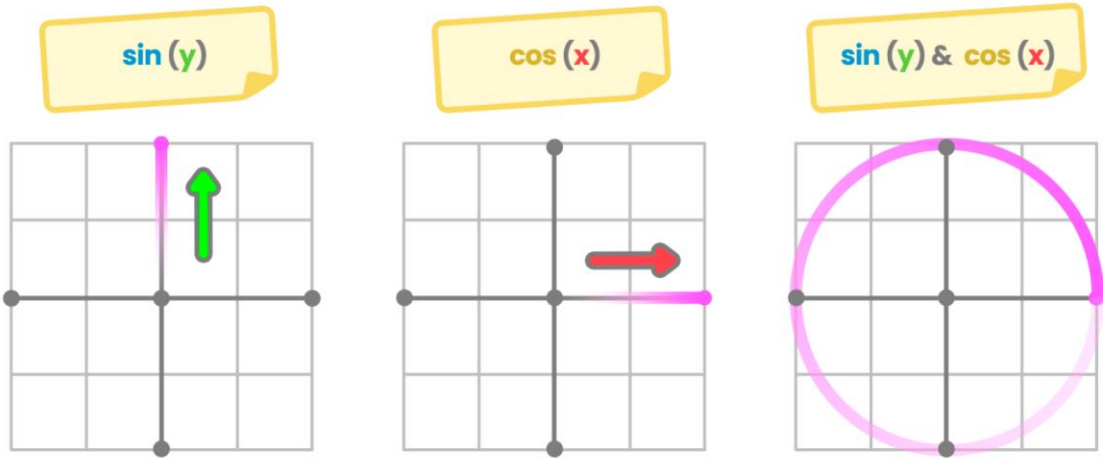


图73. 使用三角函数修改顶点位置

为了理解这个概念，我们将创建一个新的着色器，其类型“Unlit Shader”，我们将其

命名为 USB_function_SINCOS。我们将使用这个着色器在立方体的顶点上生成一个小的旋转动画。我们将首先声明一个 Float 属性，稍后我们将在函数中使用它来确定旋转的速度。

PS：

```
Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Speed ("Rotation Speed", Range(0, 3)) = 1
    }
```

为了实现旋转效果，我们需要借助矩阵来旋转坐标，这一次我们将在对象上使用三阶矩阵 (float3x3)。

PS：

```
Pass
{
    …
    sampler2D _MainTex;
            float4 _MainTex_ST;
            float _Speed;

            // 旋转顶点的方法
            float3 rotation(float3 vertex)
            {
                // 构建一个三阶矩阵
                float3x3 m = float3x3
                (
                    1, 0, 0,
                    0, 1, 0,
                    0, 0, 1
                );
                //将矩阵乘以顶点输入
                return mul(m, vertex);
            }
    …
}
```

“旋转”函数返回一个三维向量，它不执行任何特定的动作，因为三阶矩阵“m”是一个单位矩阵。但是，我们可以稍后在顶点着色器阶段使用它来变换对象的顶点。我们将首先选择一个旋转轴；以这个旋转轴为基准旋转顶点。

哔哩哔哩次元游戏人生

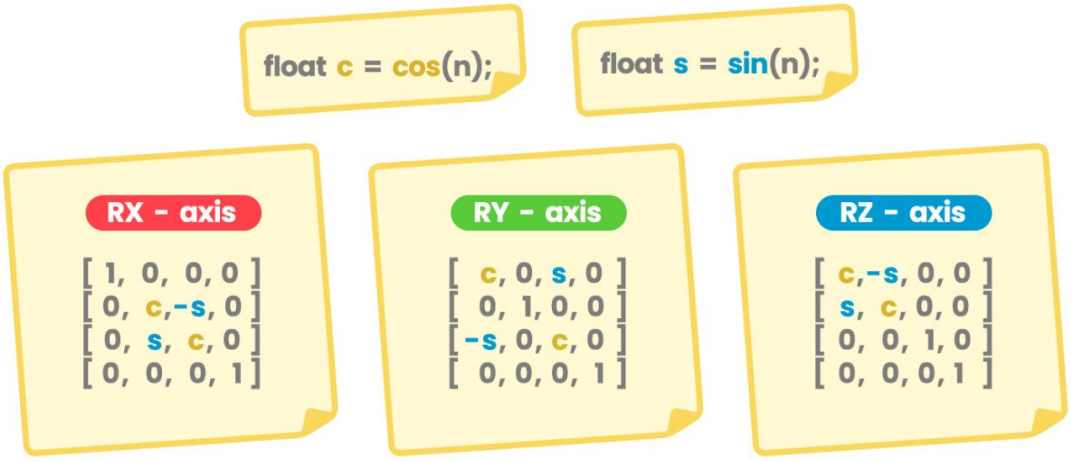


图74. 四维矩阵中的旋转轴

在上图中，每个矩阵代表一个轴的旋转变换。 例如，在这个案例中，我们将使用“Y”轴（RY 轴）进行练习。因此，我们将不得不在旋转方法中添加两个 float 类型的变量，并使用 sin 和 cos 三角函数。

PS：

```
float3 rotation(float3 vertex)
        {
            // 让我们添加旋转变量，为了让其能随时间变化，我们需要乘以
_Time
            float c = cos(_Time.y * _Speed);
            float s = sin(_Time.y * _Speed);
            // 构建一个三阶矩阵
            float3x3 m = float3x3
            (
                c, 0, s,
                0, 1, 0,
                -s, 0, c
            );
            //将矩阵乘以顶点输入
            return mul(m, vertex);
        }
```

_Time 会影响场景中立方体的顶点，因此它们将开始根据旋转轴移动。我们的旋转功能可以完美运行；现在，我们必须在顶点着色器阶段实现它。为此，我们必须考虑 UnityObjectToClipPos 方法（如果是 URP 和 HDRP 需要使用 TransformObjectToHClip），因为，它允许将对象的顶点从对象空间转换到裁剪空间。 因此，我们必须在顶点坐标转换为屏幕位置之前实现顶点的旋转。

PS：

```
v2f vert (appdata v)
        {
            v2f o;
            float3 rotVertex = rotation(v.vertex);
           //使用函数 TransformObjectToHClip 替换 UnityObjectToClipPos，就是
```

将顶点的位置从对象空间转换到裁剪空间，前者在执行过程中效率更高

```
        o.vertex = TransformObjectToHClip(v.vertex);

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

在练习中，已声明了一个名为“rotVertex”的新 3 维向量，其中存储了网格顶点的输入及其在对象空间中的旋转。然后将该向量用作 UnityObjectToClipPos 方法中的参数。

5. 正切-Tan

这个三角函数是指一个角的正切，即对边与邻边的比值，Tan $= \frac{对边}{邻边} = \frac{sin}{cos}$。

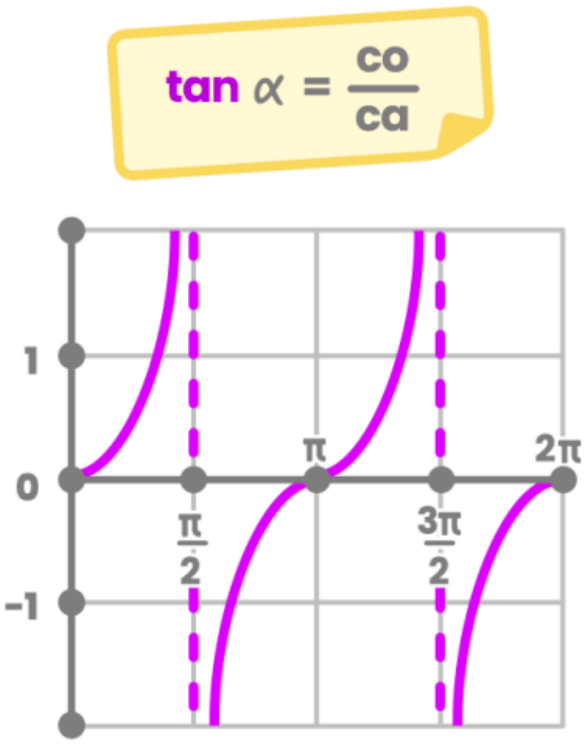


图75. 笛卡尔平面上正切函数的图形表示

通过上面的图像，我们可以知道正切函数的值在$\frac{\pi}{2} + k\pi$处是不存在的。与 sin 和 cos 一样，tan 在计算几何图形和重复图案时非常有用。一个实际的示例是生成网格状程序蒙版，我们可以使用它来生成对象上的全息投影效果。为此，我们可以简单地计算片元着色器阶段中某个 UV 坐标处的切线的绝对值。

1） 全息图像纹理

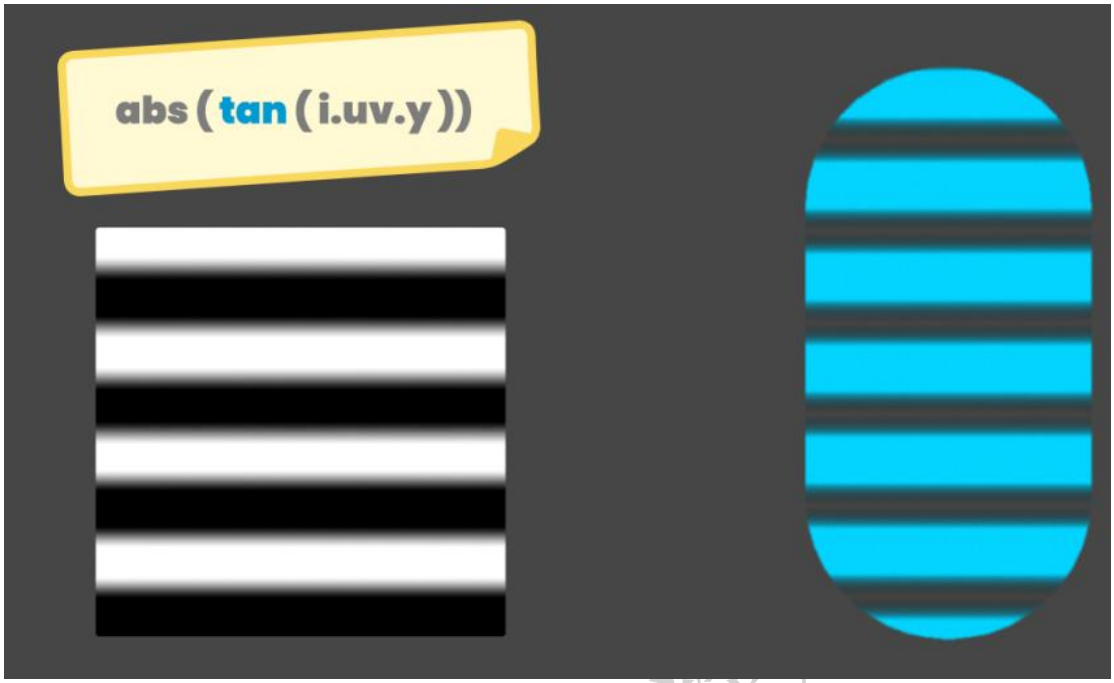


图76. 使用正切函数实现的全息效果

我们将创建一个类型为“Unlit Shader”的新着色器来说明，将这个着色器命名为USB_function_TAN。我们将首先声明一个 Color 属性和一个 Int Range 来增加或减少我们想要的全息贴图的行数。

PS：

```
Properties
    {
        _MainTex ("主纹理", 2D) = "white" {}
        _Color ("Color", Color) = (1, 1, 1, 1)
        [IntRange]_Rows ("行数", Range(2, 10)) = 10
    }
```

如上图所示，这样的视觉效果是具有透明度的；因此我们必须在 SubShader 中添加混合选项(Blending Options)，以便将黑色识别为 Alpha 通道。首先将“Queue”和”RenderType”更改为“Transparent”，然后添加某种混合类型。这里我们使用通用透明混合 Blend SrcAlpha OneMinusSrcAlpha。

PS：

```
SubShader
{
        Tags
        {
            "RenderType"="Opaque"
            "RenderPipeline"="UniversalRenderPipeline"//告诉 GPU，我们使用的是通用渲染管线
        }
        Blend SrcAlpha OneMinusSrcAlpha
        Pass { … }
```

```
}
```

接着确保声明的属性和连接变量名称一致，然后进入片元着色器阶段添加全息投影的功能。

PS：

```
//连接变量，必须与前面的属性同名
        sampler2D _MainTex;
        float4 _MainTex_ST;
        float4 _Color;
        float _Rows;
fixed4 frag (v2f i) : SV_Target
        {
            float4 tanCol = abs(tan(i.uv.y * _Rows));
            tanCol *= _Color;//调整全息贴图的颜色
            // sample the texture
            fixed4 col = tex2D(_MainTex, i.uv) * tanCol;
            return col;
        }
```

在上面的代码中，我们声明了一个名为 tanCol 的四维向量。它存储的是 V 坐标和 _Rows 属性之间乘积的正切值计算完成后，取绝对值的结果。同时我们也知道，正切的结果不像正弦和余弦在正负一之间，为此我们需要将它的结果进行一个限制，防止其出现过饱和的情况。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            float4 tanCol = clamp(0,abs(tan(i.uv.y * 3.14 * _Rows)),1);

            tanCol *= _Color;//调整全息贴图的颜色
            // sample the texture
            fixed4 col = tex2D(_MainTex, i.uv) * tanCol;
            return col;
        }
```

6. 指数- Exp, Exp2 and Pow

这些函数的特点是在其运算中使用指数，例如函数“exp”返回以 e 为底的标量和向量值的指数，“e”(2.7182828182846f)。

PS：

exp (2) = 7.3890560986f　相当于　$2.71828182846f^2$

```
float exp (float n)
{
    float e = 2.71828182846;
    float en = pow (e, n);
    return en;
}
```

哔哩哔哩次元游戏人生

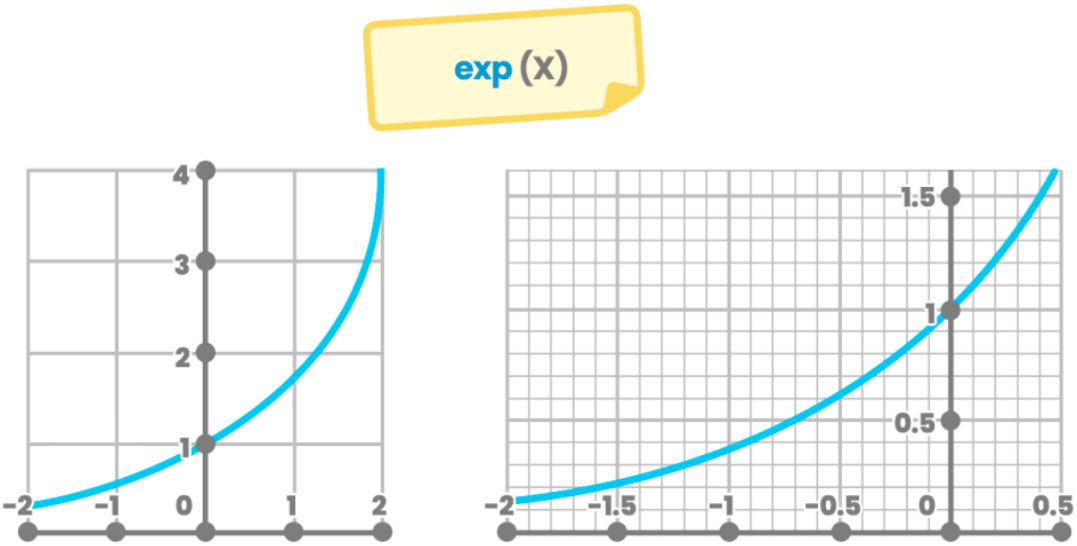


图77.　　以自然底数 e 为底数的指数函数图像

此外，“exp2”返回以 2 为底的指数。

exp2 (3) = 8　exp2 (4) = 16　exp2 (5) = 32　相当于　$2^3 = 8$、$2^4 = 16$、$2^5 = 32$。

另一方面，“pow”有两个参数，底数 (x) 及其指数 (n)。

pow (3, 2) = 9　　pow (2, 2) = 4　　pow (4, 2) = 16　　相当于　$3^2 = 9$、$2^2 = 4$、4^2

这些功能的有用性将取决于正在执行的操作，它们通常用于计算噪声、增加输出颜色中的伽马值和重复模式。

7. 向下取整-Floor

该函数返回一个不大于其参数的整数值，即不带小数位的标量或向量数值，向下取整。例如 1.97f 的 Floor 就返回 1，就相当于舍弃了小数部分。

floor (1.56) = 1　　floor (0.34) = 0　　floor (2.99) = 2　相当于 1.56f - 0.56f = 1

PS：

```
float floor (float n)
{
    float fn;
    fn = n - frac(n);
    return fn;
}
```

哔哩哔哩次元游戏

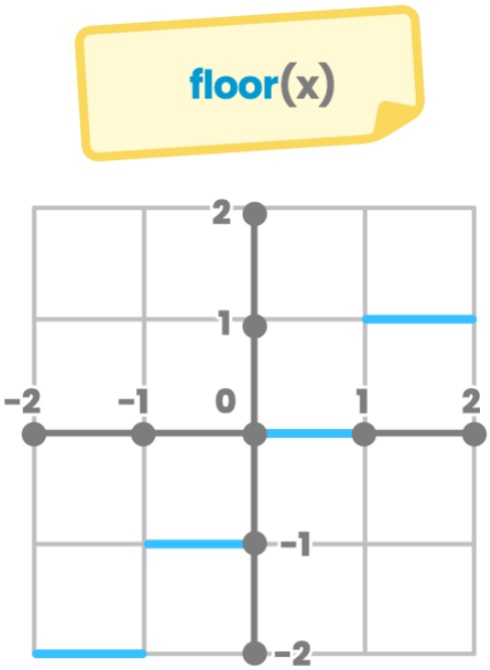


图78.　　笛卡尔平面上向下取整的图形表示

1） 纯色块

与 ceil 函数（向上取整）功能相反，在使用纯色块创建视觉效果时，Floor 函数非常有用，例如卡通 Shder 或一般的重复性图案。我们用一个例子了解实现卡通着色器的原理，为此我们将创建一个新的着色器，其类型为“Unlit Shader”，我们将其称为 USB_functions_FLOOR。

我们将首先在 Shader 中添加两个属性，第一个属性来生成多个分割，第二个属性用于增加输出颜色中的 gamma。

PS：

```
Shader "USB/USB_function_FLOOR"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        [IntRange]_Sections ("Sections", Range (2, 10)) = 5
        _Gamma ("Gamma", Range (0, 1)) = 0
    }
    SubShader { … }
}
```

由于必须统一聚合各个部分，因此为 _Sections 变量定义了[IntRange]，使其以整数进行变动。和之前的流程一样，我们必须在 Pass 中创建连接变量，这样我们才能实现 ShaderLab 和我们程序的直接通信。

PS：

```
Pass
{
    HLSLPROGRAM
    …
```

哔哩哔哩次元游戏人生

```
        //连接变量，必须与前面的属性同名
        sampler2D _MainTex;
        float4 _MainTex_ST;
        float _Sections;
        float _Gamma; …
        ENDHLSL
    }
```

接下来，我们将进入片元着色器阶段并声明一个新变量，该变量将根据 UV 的 V 坐标生成纯色块。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            float fv = floor(i.uv.y);
            // sample the texture
            fixed4 col = tex2D(_MainTex, i.uv);
            return col;
        }
```

在上述的代码中，变量“fv”已经被声明和初始化，它只有一个维度。它的值等于 UV 坐标中每一个 V 坐标的下限值；因此它等于“零”。这是因为 UV 坐标从 0.0f 开始，到 1.0f 结束，且 Floor 函数返回的是下限值。

我们可以通过指定一个四维向量作为输出来证实该操作，其中前三个分量等于“fv”的值。

PS：

```
fixed4 frag (v2f i) : SV_Target
        {
            float fv = floor(i.uv.y);
            // sample the texture
            //fixed4 col = tex2D(_MainTex, i.uv);
            //return col;
            return float4 (fv.xxx,1);
        }
```

图79.　　对原始 UV 取下限值，得到的是左下角的那个像素点

要添加纯色块，我们可以简单地将 UV 乘以一定的值，然后除以十进制值。至于 gamma，我们可以直接将属性添加到输出颜色中。

```
fixed4 frag (v2f i) : SV_Target
        {
            //先将 UV 放大后对其取整，然后再将 UV 缩放回去
            float2 fv = floor(i.uv * _Sections) * (_Sections * 0.01);
            return float4 (fv.xy,0,1) + _Gamma;
        }
```

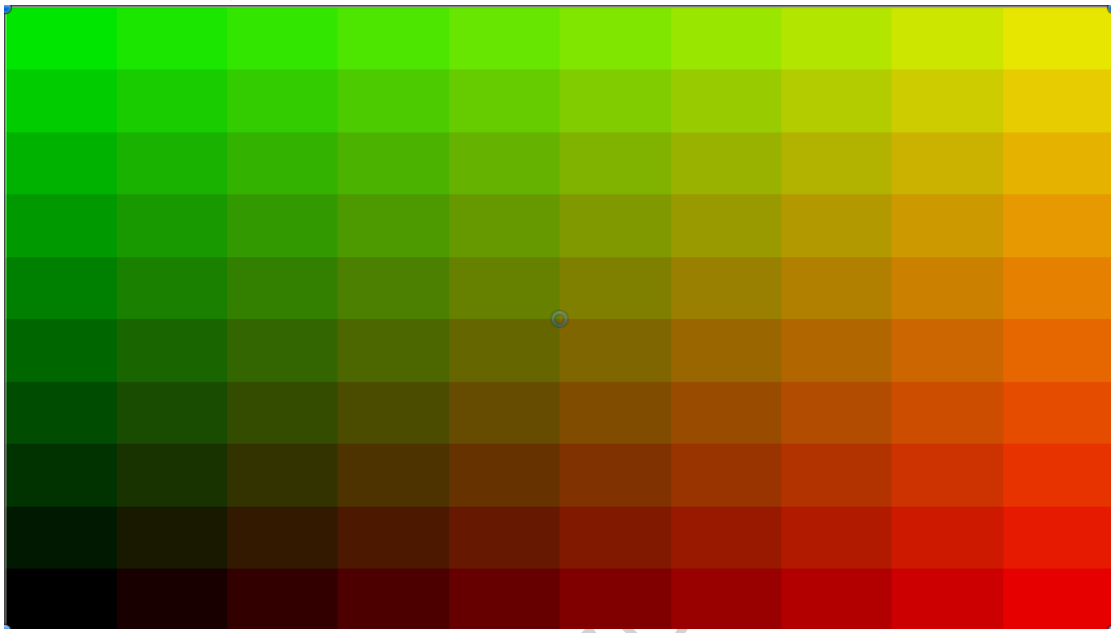


图80.　　利用 Floor 函数使 UV 失去部分精度后，所得的图像

实现原理与卡通 Shader 相同，不同之处在于，卡通 Shader 在计算中使用全局光照而不是 UV 坐标。

8. 分段和平滑分段-Step and Smoothstep

step 和 smoothstep 是非常相似的函数，实际上它们都有一个称为“edge”的参数，负责区分两个值之间的返回值。我们将从 step 开始，然后再讲解 smoothstep，因为它的结构比 Step 更精细。

NVIDIA 的官方文档解释是：参数值大于或等于 Edge，step 就返回 1；否则，它返回 0。

PS：

```
float step (float x, float edge)
{
    return edge >= x;
}
```

以上面的语句为例，我们可以在片元着色器阶段执行一个简单的操作来了解它是如何工作的。

PS：

```
float4 frag (v2f i) : SV_Target
        {
            // 确定边缘的分界值
            float edge = 0.5;
            // 用于存储 Step 函数的返回结果
            float3 sstep = 0;
```

```
        sstep = step (i.uv.y, edge);
        return float4(sstep, 1);
    }
```

我们首先在上述操作中声明一个名为“sstep”的三维向量，在下图中可以看到该 Shader 的图形表示。我们使用 UV 的 V 坐标，去和“Edge”值 0.5 进行比较并返回。UV 的 V 坐标，是从下到上的一张渐变，其值从下到上是 0 到 1。

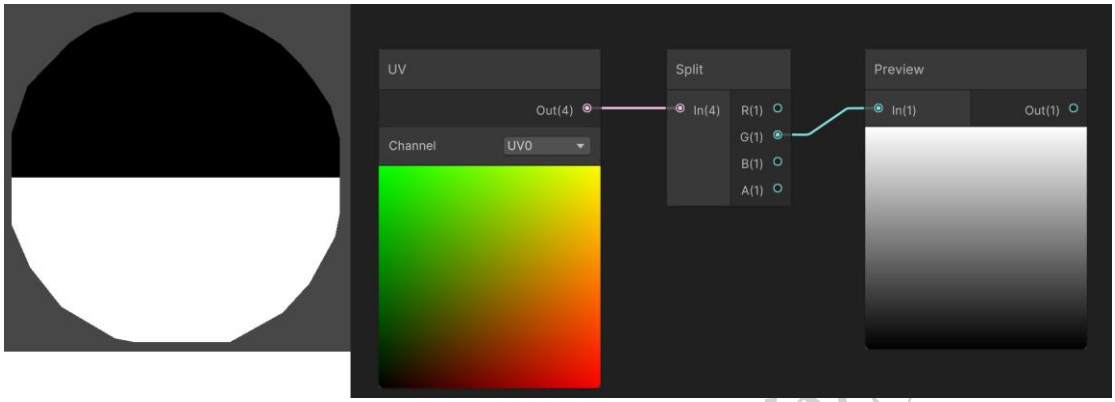


图81.　当前状态 Step 函数返回的图像，以及在 ShaderGraph 中 UV 的 V 通道图像

我们可以看到我们的输出和英伟达官方解释有出入，所有小于或等于 Edge 值的将返回“一”（白色），相反情况下返回（黑色）即“零”。

smoothstep 函数的行为与前一个没有太大区别；它唯一的区别在于在返回值之间生成线性插值。

PS：

```
float smoothstep (float a, float b, float edge)
{
    float t = saturate((edge - a) / (b - a));
    return t * t * (3.0 - (2.0 * t));
}
```

回到片元着色器阶段，我们可以添加一个新变量来确定返回值之间的插值量。

PS：

```
float4 frag (v2f i) : SV_Target
    {
        // 确定边缘的分界值
        float edge = 0.5;
        //插值量
        float smooth = 0.1;
        // 用于存储 Step 函数的返回结果
        float3 sstep = 0;
        sstep = smoothstep((i.uv.y - smooth), (i.uv.y + smooth), edge);
        return float4(sstep, 1);
    }
```

在前面的代码中，我们在 0.4f 和 0.6f 之间创建了一条从 1 到 0 的插值。

哔哩哔哩次元游戏

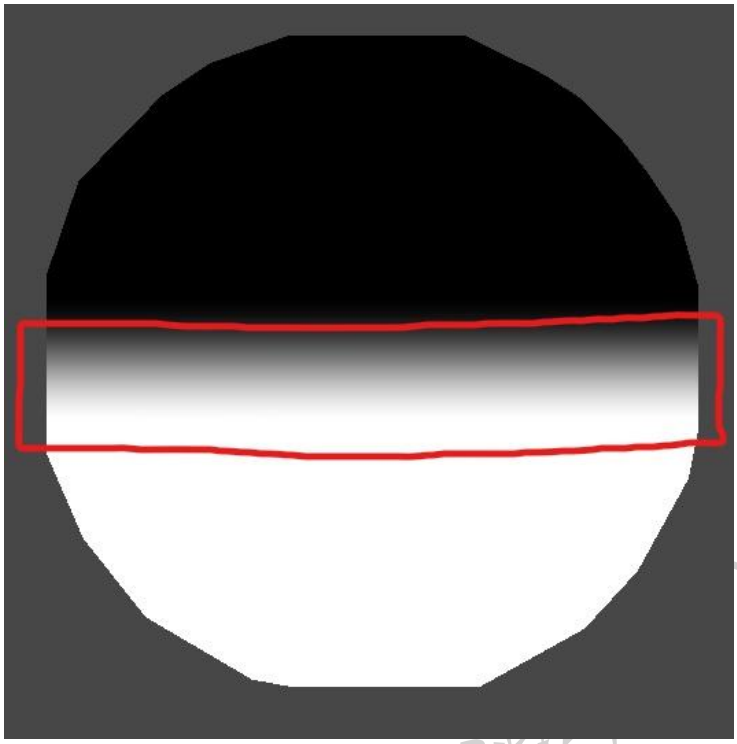


图82.　　SmoothStep 的效果呈现

9. 长度- Length

正如标题所提到的，长度是指表示两点之间距离的大小。这个函数在创建几何形状时很方便，例如我们可以生成圆形或带有圆边的多边形。

PS：

```
float length (float n)
{
    //这里计算的内积，然后计算模
    return sqrt(dot(n,n));
}
```

1） 圆形

像往常一样，我们将创建一个新的着色器，其类型为“Unlit Shader”，我们将其命名为 USB_function_Length。这次我们将在我们的程序中使用一些函数来表示一个圆圈，我们添加将用于放大、居中和平滑形状的属性，它们可以是 Float Range 或其他数字属性。

PS：

```
Shader "USB/USB_function_LENGTH"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Radius ("Radius", Range(0.0, 0.5)) = 0.3
        _Center ("Center", Range(0, 1)) = 0.5
        _Smooth ("Smooth", Range(0.0, 0.5)) = 0.01
    }
    SubShader
    {
```

```
            …
        Pass
        {
            …
            float _Smooth;
            float _Radius;
            float _Center;
            …
        }
    }
}
```

要创建一个圆，我们只需计算 UV 坐标与圆心的距离并减去半径，也就是使用圆的 SDF。

PS：

```
float circle (float2 p, float radius)
        {
            //创建一个圆形
            float c = length(p) - radius;
            return c;
        }
```

然而正如我们在下图中看到的那样，前面的操作返回了一个从点 0,0f 开始到 1,0f 结束的模糊圆，我们需要想办法将其置于中心位置。

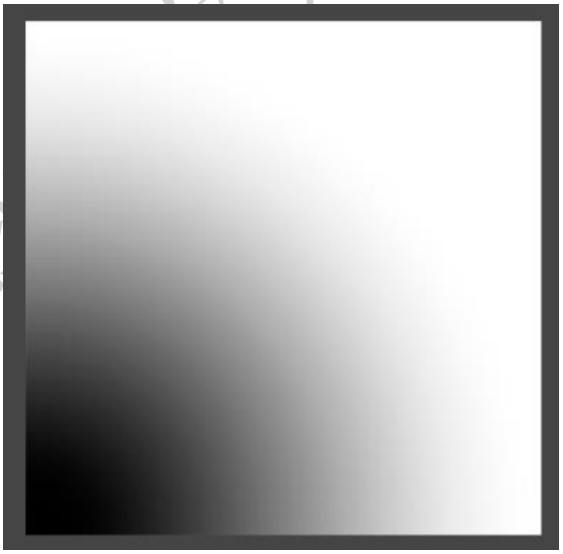


图83.　圆心为于(0, 0)点的有向距离场

我们可以在函数中添加一个新参数，该参数对应的就是圆心位置。

PS：

```
float circle (float2 p,float center, float radius)
        {
            //创建一个圆形
            float c = length(p - center) - radius;
            return c;
```

```
}
```

但是，现在的圆圈仍然会模糊。如果我们想控制模糊量，我们可以使用 smoothstep 函数，正如我们现在已经知道的，它会在两个值之间生成线性插值（即一个柔性的过渡）。

PS：

```
float circle (float2 p,float center, float radius, float smooth)
        {
            //创建一个圆形
            float c = length(p - center) - radius;
             return smoothstep(c - smooth, c + smooth, radius);
        }
```

在之前我们添加了一个名为“smooth”的新参数，这将允许我们控制模糊量。接下来我们可以在片元着色器阶段将其进行应用。

PS：

```
float circle (float2 p, float center, float radius, float smooth)
{ … }
fixed4 frag (v2f i) : SV_Target
        {
            float c = circle (i.uv, _Center, _Radius, _Smooth);
            return float4(c.xxx ,1);
        }
```

图84.　Radius 0,3f, Center 0,5f and Smooth 0,023f

10. 小数- Frac

该函数返回一个数值的小数部分，即它的十进制值，例如 1.534f 返回 0.534f。

frac (3,27) = 0,27f　frac (0,47) = 0,47f　frac (1,0) = 0,0f　　相当于　3.27f – 3 = 0.27f

PS：

```
float frac (float n)
{
    return n - floor(n);
}
```

我们可以在噪声计算、随机重复模式等多种操作中使用到 Frac 函数。

1） 重复图形

为了理解这个概念，我们将创建一个新的着色器，其类型为“Unlit Shader”，将其命名为 USB_function_FRAC。首先添加一个名为”Size”的 Float Range 属性，稍后将使用“size”来增加或减小圆的大小。

PS：

```
Shader "USB/USB_function_FRAC"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Size ("Size", Range(0.0, 0.5)) = 0.3
    }
    SubShader
    {
        …
        Pass
        {
            …
            float _Size;
            …
        }
    }
}
```

我们将执行的第一个操作是将 UV 坐标放大，以在片元着色器阶段获得重复图案。

PS：

```
float4 frag (v2f i) : SV_Target
            {
                i.uv *= 3;//将 UV 放大 3 倍
                float4 col = tex2D(_MainTex, i.uv);
                return col;
            }
```

图85. 纹理的 Wrap Mode 对应于 Clamp

如果在 Unity 中的纹理，其 Wrap Mode 设置为“clamp”，我们将可以注意到图像中的边缘纹素与自身一起被拉伸了，就像上图所示的一样。左边的图像是默认的 UV 坐标，而右边的图像将 UV 坐标放大三倍的结果。

哩哩哩次元游戏人生

在这种情况下，我们可以使用“frac”函数返回 UV 坐标的小数值，以生成自定义的重复模式。

PS：

```
float4 frag (v2f i) : SV_Target
        {
            i.uv *= 3;//将 UV 放大 3 倍
            float2 fuv = frac(i.uv);//只取 UV 的小数部分，舍弃整数部分，所有的UV 区域就都变成了 0 到 1
            float circle = length(fuv - 0.5);//生成一个圆的距离场
            float wCircle = floor(_Size / circle);//翻转颜色并返回一个整数值
            return float4(wCircle.xxx, 1);
        }
```

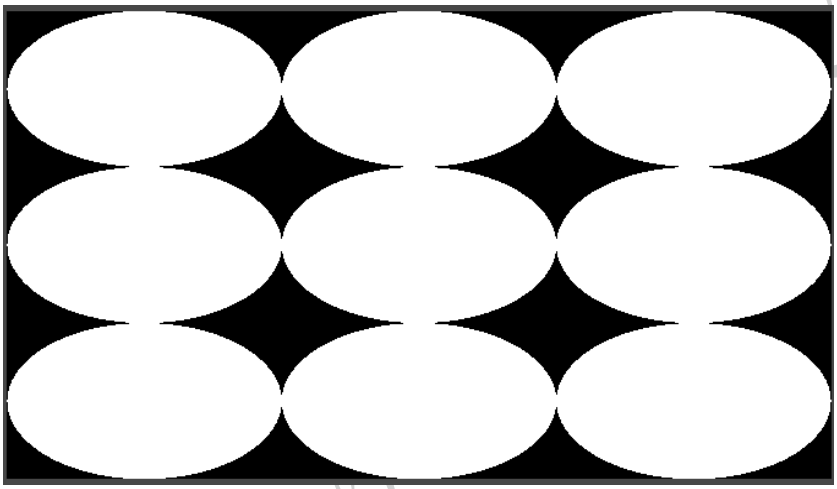


图86.　利用 Frac 函数，实现的重复图案效果

11. 线性插值-Lerp

该函数常用于颜色转换，它允许在两个值之间进行线性插值，例如我们可以在我们的一个角色上使用 lerp 通过交叉淡化从一个皮肤切换到另一个皮肤。

PS：

```
float lerp (float a, float b, float n)
{
    return a + n * (b - a);
}
```

1） 渐变

我们将创建一个类型为“Unlit Shader”的 Shader 来举例说明该函数，我们将其命名为 USB_function_LERP。我们将首先声明两个 Texture，稍后将使用它们作为“皮肤”，然后加上一个 Float Range 来执行交叉淡入淡出。

PS：

```
Shader "USB/USB_function_LERP"
{
    Properties
    {
        _Skin01 ("Skin 01", 2D) = "white" {}
        _Skin02 ("Skin 02", 2D) = "white" {}
```

```
            _Lerp ("Lerp", Range(0, 1)) = 0.5
        }
        SubShader
        {
            …
            Pass
            {
            …
            sampler2D _Skin01;
            float4 _Skin01_ST;
            sampler2D _Skin02;
            float4 _Skin02_ST;
            float _Lerp;
            …
            }
        }
    }
```

由于我们将使用两个纹理，因此在每种情况下都需要使用 UV 坐标。它们必须在顶点输入和输出中声明，主要原因有两个。一是因为，通过 TRANSFORM_TEX 函数，纹理会受到“平铺和偏移”的影响。另一个则是因为，我们稍后会在片元着色器阶段使用它们。

PS：

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv_s01 : TEXCOORD0;//表示第一组纹理坐标
    float2 uv_s02 : TEXCOORD1;//表示第二组纹理坐标
};

struct v2f
{
    float4 vertex : SV_POSITION;
    float2 uv_s01 : TEXCOORD0;
    float2 uv_s02 : TEXCOORD1;
};

v2f vert (appdata v)
{
    v2f o;

    //使用函数 TransformObjectToHClip 替换 UnityObjectToClipPos，就是将顶点的位置从对象空间转换到裁剪空间，前者在执行过程中效率更高
    o.vertex = TransformObjectToHClip(v.vertex);

    o.uv_s01 = TRANSFORM_TEX(v.uv_s01, _Skin01);
```

哔哩哔哩欢元游戏人生

```
        o.uv_s02 = TRANSFORM_TEX(v.uv_s02, _Skin02);
        return o;
    }
```

在片元着色器阶段，我们可以声明两个四维向量，在每种情况下使用 tex2D 函数，然后使用 lerp 函数在它们之间进行线性插值。

PS：

```
    float4 frag (v2f i) : SV_Target
    {
        float4 skin01 = tex2D(_Skin01, i.uv_s01);
        float4 skin02 = tex2D(_Skin02, i.uv_s02);

        //在每种皮肤纹理之间进行线性插值
        fixed4 render = lerp(skin01, skin02, _Lerp);
        return render;
    }
```

如果我们注意_Lerp 属性，我们会注意到它的值被限制在 0.0f 和 1.0f 之间。默认情况下，它的值为 0.5f；因此，如果我们为 _Skin[n] 的每个属性分配两个不同的纹理，其显示的结果就是两张纹理都是 0.5f 的透明度。

图87.　　两个纹理之间的线性插值

12. 最大和最小值-Min and Max

一方面，“min”是指两个向量(Vector[8])或标量(Scalar[9])之间的最小值，而“max”则相反。我们将会在很多不同的操作中频繁使用到这些函数，例如我们可以使用 max 来计算对象上的扩散，返回“零”与网格法线与光的方向之间的点积之间的最大值。

PS：

```
float min (float a, float b)
{
    float n = (a < b ? a : b);
    return n;
}
```

[8] n. 矢量，向量；（传染疾病的）介体　　v. 引导（飞机）

[9] adj. 标量的；数量的；梯状的，分等级的　　n. [数] 标量；[数] 数量

```
float max (float a, float b)
{
    float n = (a > b ? a : b);
    return n;
}
```

在漫反射中，我们将会使用到这两个方法。

13. 时间和动画- Timing and animation

在 Unity 中，我们将经常使用三个内置着色器变量来为我们的效果设置动画属性。它们就是_Time、_SinTime 和 _CosTime。

这些变量是四维向量，每个分量代表一个速度级别，例如，“_Time.y”等于自场景或级别加载以来经过的时间（以秒为单位），类似于 Time 的函数 .timeSinceLevelLoad。相反，“_Time.x”对应于相同的值，除以 20。

PS：

_Time.x = t / 20; _Time.y = t; _Time.z = t * 2; _Time.w = t * 3;

_CosTime 和_SinTime 变量具有相同的行为，因为它们对应于_Time 的简化函数，例如，_CosTime.w 等于操作“cos(_Time.y)”；两者都具有相同的变化区间，只是周期有所不同。

_SinTime.x = t / 8; _SinTime.y = t / 4; _SinTime.z = t / 3; _SinTime.w = t;

_CosTime.x = t / 8; _CosTime.y = t / 4; _CosTime.z = t / 3; _CosTime.w = t;

1） UV 偏移动画

我们可以使用_Time 函数来生成图标的偏移动画来加深概念。为此，我们必须在片元着色器阶段为 UV 的坐标添加_Time。

PS：

```
float4 frag (v2f i) : SV_Target
{
    //让 UV 的 V 值随着时间增加
    i.uv.x += _Time.y;
    float4 col = tex2D(_MainTex, i.uv);
    return col;
}
```

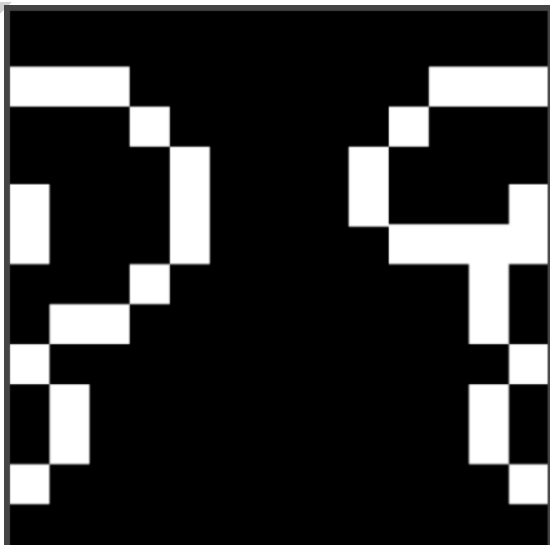


图88. 四边形上的 U 偏移

或者我们也可以使用_SinTime 和_CosTime 生成旋转动画，同时影响 U 和 V 的运动。

PS：

```
float4 frag (v2f i) : SV_Target
        {
            //让 UV 的 U 值随着时间增加
            i.uv.x += _SinTime.w;
            // 让 UV 的 V 值随着时间增加
            i.uv.y += _CosTime.w;
            float4 col = tex2D(_MainTex, i.uv);
            return col;
        }
```

图89. Quad 上 U 和 V 方向的偏移旋转，Tiling 等于 4

十五、 配置(Configure[10])输入和输出

在这里我们将详细介绍，如何为对象配置法线并将其坐标从对象空间转换到世界空间所需的步骤。正如我们已经知道的，如果想在对象上使用法线，那么我们必须将语义NORMAL[n]存储在一个三维向量中。

第一步是在顶点输入中包含它。

PS：

```
struct appdata
{
    …
    float3 normal : NORMAL;
};
```

请记住，向量的第四维对应于它的 W 分量，在正常情况下，它的默认值为“零”，因为它是空间中的方向。一旦我们将法线声明为输入，我们必须判断是否需要将这些值传递给片元着色器阶段？如果答案是肯定的，那么我们将不得不再次声明法线，但这次是作为输出。这个类比同样适用于我们在程序中使用的所有输入和输出。

PS：

[10] v. 安装，装配；（计算机）配置，设定

```
struct v2f
{
    …
    float3 normal : TEXCOORD1;// 表示第二组纹理坐标
};
```

为什么我们在顶点输入和顶点输出中都声明了法线？这是因为我们必须在顶点着色器阶段连接这两个属性，然后将它们传递给片元着色器阶段。 需要注意的是，根据 HLSL 官方文档，片元着色器阶段没有 NORMAL 语义，因此，我们必须使用至少可以存储三个空间坐标的语义。 这就是我们在上面的示例中使用 TEXCOORD1 的原因。这种语义有四个维度 (XYZW)，非常适合使用法线。

在顶点输入和顶点输出中都声明了一个属性后，我们可以进入顶点着色器阶段来连接它们。

PS：

```
v2f vert (appdata v)
{
    v2f o;
    …
    // 我们将输出与输入连接起来
    o.normal = v.normal;
    …
    return o;
}
```

我们刚刚做了什么？我们将 struct v2f 的法线输出与 struct appdata 的法线输入连接起来。请记住，v2f 在片元着色器阶段用作参数，这意味着我们可以在此阶段使用法线作为属性。

为了说明这个概念，我们将创建一个示例函数，我们将其称为 unity_light 它负责在片元着色器阶段计算(Calculate[11])表面上的光照。该函数本身不会执行任何实际操作；但是，它将帮助我们了解一些我们应该知道的因素。

PS：

```
void unity_light (in float3 normals, out float3 Out)
{
    Out = [Op] (normals);
}
```

根据函数的声明，我们可以看到 unity_light 是一个无返回值的函数；有两个参数，即法线和输出值。请注意，所有光照计算操作都需要将法线作为其变量之一。为什么？ 因为没有这个变量，程序将不知道光应该如何与物体表面相互作用。我们将在片元着色器阶段应用 unity_light 函数。

PS：

```
fixed4 frag (v2f i) : SV_Target
{
    // 将法线存储在向量中
    half3 normals = i.normal;
    // 将我们的灯光初始化为黑色
```

[11] v. 计算，核算；预测，推测

哔哩哔哩次元游戏人生

```
        half3 light = 0;
        // 初始化我们的函数并传递向量
        unity_light(normals, light);
        return float4(light.rgb, 1);
    }
```

由于 unity_light 对应两个参数，因此创建了两个 3 维向量，它们将作为函数中的参数。第一个对应于法线，第二个是输出值，对应光照输出。如果我们仔细观察，我们会注意到在前一阶段（即 i.normal）声明的法线输出是在对象空间(Object-space)中，我们怎么知道的呢？因为到目前为止还没有生成任何类型的矩阵变换。

光照计算的操作必须在世界空间(World-space)中，为什么？ 因为入射(Incidence[12])值是在世界范围内找到的；同样，在场景中，游戏对象根据网格(Grid[13])中心进行定位，因此，我们必须在片元着色器阶段转换法线的空间坐标。

PS：

```
    // 创建一个新函数
    half3 normalWorld (half3 normal)
    {
        return normalize(mul(unity_ObjectToWorld, float4(normal, 0))).xyz;
    }

    half4 frag (v2f i) : SV_Target
    {
        // 将世界空间法线存储在向量中
        float3 normals = normalWorld(i.normal);
        float3 light = 0;
        unity_light(normals, light);
        return float4(light.rgb, 1);
    }
```

在上面的示例中，normalWorld 函数将空间坐标转换后返回给法线。它使用了 unity_ObjectToWorld 矩阵，它允许我们从对象空间到世界空间。

哔哩哔哩次元游戏人生

[12] n. 发生率；入射

[13] n. 网格，方格；栅栏；输电网，电力网；赛车起跑线；美式橄榄球球场，橄榄球赛； v. 使……构成网格状，布置成网格结构

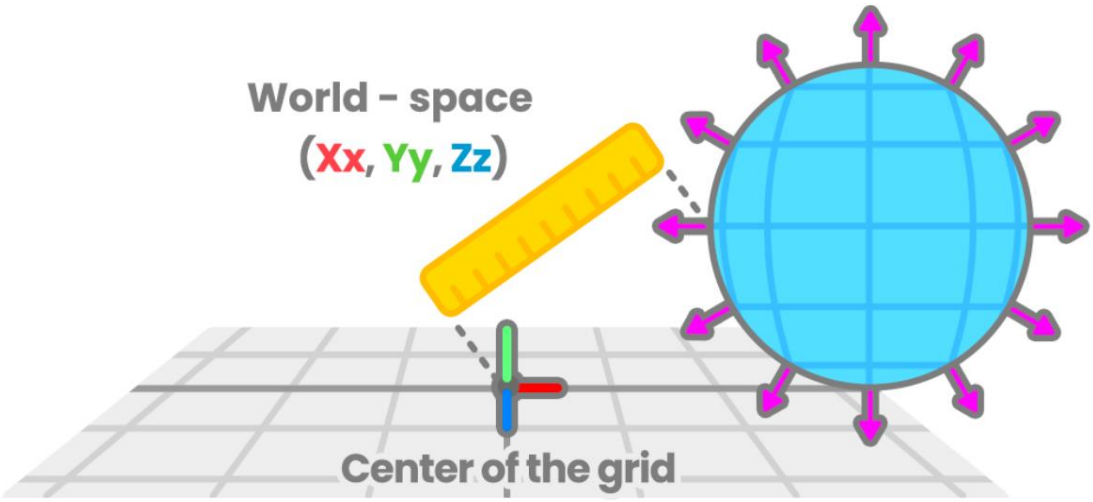


图90.　　游戏对象根据网格中心进行定位

十六、　Unity 中的基础光照

渲染总是围绕着一个基础问题：我们如何决定一个像素的颜色？从宏观上来说，渲染包含了两大部分——决定一个像素的可见性，决定这个像素上的光照计算。

1. 我们是如何看见这个多彩世界的

我们可能常常会问类似于这样的问题："这个物体是什么颜色的？"当我们在描述"这个物体是红色的"时，实际上是因为这个物体会反射更多的红光波长，而吸收了其他波长。而如果一个物体在我们看来是黑色的，实际上是因为它吸收了绝大部分的波长。

1） 光源

光不是从石头里蹦出来的，而是由光源发射出来的。在实时渲染中，我们通常把光源当成个没有体积的点，用 l 来表示它的方向。那么，我们如何测量一个光源发射出了多少光呢？也就是说，我们如何量化光呢？在光学里，我们使用辐照度（irradiance）来量化光。对于平行光来说它的辐照度可通过计算在垂直于 l 的单位面积上单位时间内穿过的能量来得到。在计算光照模型时，我们需要知道一个物体表面的辐照度，而物体表面往往是和 l 不垂直的，那么如何计算这样的表面的辐照度呢？我们可以使用光源方向 l 和表面法线 n 之间的夹角的余弦值来得到。需要注意的是，这里默认方向矢量的模都为 1。

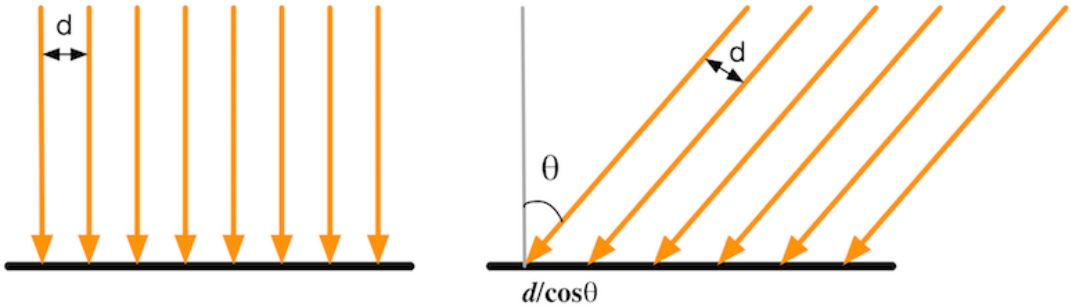


图91.　　使用余弦值计算光照的辐照度

在左图中，光是垂直照射到物体表面，因此光线之间的垂直距离保持不变；而在右图中，光是斜着照射到物体表面，在物体表面光线之间的距离是$d/\cos\theta$，因此单位面积上接收到的光线数目少于左图。

因为辐照度是和照射到物体表面时光线之间的距离$d/\cos\theta$成反比的，因此辐照度就和

$\cos\theta$成正比。$\cos\theta$可以使用光源方向 l 和表面法线 n 的点积来得到。这就是使用点积来计算辐照度的由来。

2） 吸收和散射

光线由光源发射出来后，就会与一些物体相交。通常，相交的结果有两个，一个是散射（scattering），另一个是吸收（absorption）。

散射只改变光线的方向，但不改变光线的密度和颜色。而吸收只改变光线的密度和颜色，但不改变光线的方向。光线在物体表面经过散射后，有两种方向。一种将会散射到物体内部，这种现象被称为折射(refraction)或透射(transmission)；另一种将会散射到外部，这种现象被称为反射(reflection)。

对于不透明物体，折射进入物体内部的光线还会继续与内部的颗粒进行相交，其中一些光线最后会重新发射出物体表面，而另一些则被物体吸收。那些从物体表面重新发射出的光线将具有和入射光线不同的方向分布和颜色。

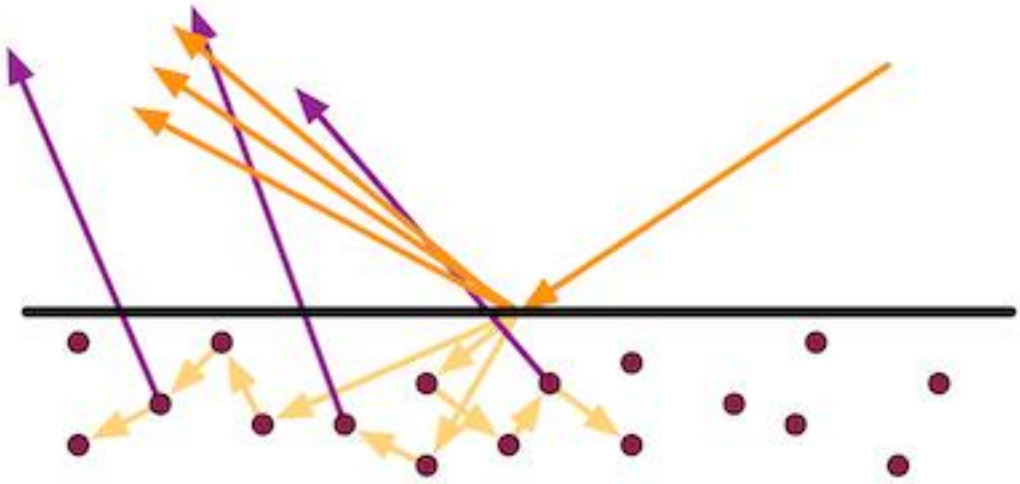


图92. 散射时，光线会发生折射和反射现象

为了区分这两种不同的散射方向，我们在光照模型中使用了不同的部分来计算它们。高光反射（specular）部分表示物体表面是如何反射光线的，而漫反射（diffuse）部分则表示有多少光线会被折射、吸收和散射出表面。根据入射光线的数量和方向，我们可以计算出射光线的数量和方向，我们通常使用出射度（exitance）来描述它。辐照度和出射度之间是满足线性关系的，而它们之间的比值就是材质的漫反射和高光反射属性。

在后面的内容中，我们仅假设漫反射部分是没有方向性的，也就是说，光线在所有方向上是平均分布的。同时，我们也只考虑某一个特定方向上的高光反射。

3） 着色

着色(shading)指的是，根据材质属性（如漫反射属性等）、光源信息（如光源方向、辐照度等），使用一个等式去计算沿某个观察方向的出射度的过程。我们也把这个等式称为光照模型(Lighting Model)。不同的光照模型有不同的目的。例如，一些用于描述粗糙的物体表面，一些用于描述金属表面等。

4） BRDF 光照模型

我们已经了解了光线在和物体表面相交时会发生哪些现象。当已知光源位置和方向、视角方向时，我们就需要知道一个表面是如何和光照进行交互的。例如，当光线从某个方向照射到一个表面时，有多少光线被反射？反射的方向有哪些？

而 BRDF（Bidirectional Reflectance DistributionFunction）就是用来回答这些问题的。

当给定模型表面上的一个点时，BRDF 包含了对该点外观的完整的描述。在图形学中，BRDF 大多使用一个数学公式来表示，并且提供了一些参数来调整材质属性。通俗来讲，当给定入射光线的方向和辐照度后，BRDF 可以给出在某个出射方向上的光照能量分布。BRDF 都是对真实场景进行理想化和简化后的模型，也就是说，它们并不能真实地反映物体和光线之间的交互，这些光照模型被称为是经验模型。尽管如此，这些经验模型仍然在实时渲染领域被应用了多年。我们可以从邓恩的著作《3D 数学基础：图形与游戏开发》（英文名：《3D Math Primer For Graphics And Game Development））中提到的一句名言来体会这其中的原因。

计算机图形学的第一定律：如果它看起来是对的，那么它就是对的。

然而，有时我们希望可以更加真实地模拟光和物体的交互，这就出现了基于物理的 BRDF 模型。

2. 标准光照模型

虽然光照模型有很多种类，但在早期的游戏引擎中往往只使用一个光照模型，这个模型被称为标准光照模型。实际上，在 BRDF 理论被提出之前，标准光照模型就已经被广泛使用了。

在 1973 年，著名学者裴祥风（Bui Tuong Phong）提出了标准光照模型背后的基本理念。标准光照模型只关心直接光照（direct light），也就是那些直接从光源发射出来照射到物体表面后，经过物体表面的一次反射直接进入摄像机的光线。

它的基本方法是，把进入到摄像机内的光线分为 4 个部分，每个部分使用一种方法来计算它的贡献度。这 4 个部分是。

① 自发光（emissive）部分，本书使用 $C_{emissive}$ 来表示。这个部分用于描述当给定一个方向时，一个表面本身会向该方向发射多少辐射量。需要注意的是，如果没有使用全局光照（global illumination）技术，这些自发光的表面并不会真的照亮周围的物体，而是它本身看起来更亮了而已。

② 高光反射（specular）部分，本书使用 $C_{specular}$ 来表示。这个部分用于描述当光线从光源照射到模型表面时，该表面会在完全镜面反射方向散射多少辐射量。

③ 漫反射（diffuse）部分，本书使用 $C_{diffuse}$ 来表示。这个部分用于描述，当光线从光源照射到模型表面时，该表面会向每个方向散射多少辐射量。

④ 环境光（ambient）部分，本书使用 $C_{ambient}$ 来表示。它用于描述其他所有的间接光照。

1） 环境光

虽然标准光照模型的重点在于描述直接光照，但在真实的世界中，物体也可以被间接光照(indirect light)所照亮。间接光照指的是，光线通常会在多个物体之间反射，最后进入摄像机，也就是说，在光线进入摄像机之前，经过了不止一次的物体反射。例如，在红地毯上放置一个浅灰色的沙发，那么沙发底部也会有红色，这些红色是由红地毯反射了一部分光线，再反弹到沙发在标准光照模型中。

在标准光照模型中，我们使用了一种被称为环境光的部分来近似模拟间接光照。环境光的计算非常简单，它通常是一个全局变量，即场景中的所有物体都使用这个环境光。下面的等式给出了计算环境光的部分。

$$C_{ambient} = g_{ambient}$$

2） 自发光

光线也可以直接由光源发射进入摄像机，而不需要经过任何物体的反射。标准光照模型使用自发光来计算这个部分的贡献度。它的计算也很简单，就是直接使用了该材质的自

发光颜色。

$$C_{\text{emissive}} = m_{\text{emissive}}$$

通常在实时渲染中，自发光的表面往往并不会照亮周围的表面，也就是说，这个物体并不会被当成一个光源。但是，在现在的 Unity 全局光照系统的帮助下，则可以模拟这类自发光物体对周围物体的影响。

3） 漫反射

漫反射光照是用于对那些被物体表面随机散射到各个方向的辐射度进行建模的。

在漫反射中，视角的位置是不重要的，因为反射是完全随机的，因此可以认为在任何反射方向上的分布都是一样的。但是，入射光线的角度很重要。

漫反射光照符合兰伯特定律(Lambert's law)，反射光线的强度与表面法线和光源方向之间夹角的余弦值成正比。因此，漫反射部分的计算如下。

$$C_{\text{diffuse}} = (C_{\text{light}} \cdot m_{\text{diffuse}}) \max(0, \hat{n} \cdot \hat{l})$$

其中，$\hat{n}$是表面法线，$\hat{l}$是指向光源的单位矢量，m_{diffuse}是材质的漫反射颜色，C_{light}是光源颜色。需要注意的是，我们需要防止法线和光源方向点乘的结果为负值，为此，我们使用取最大值的函数(max()函数)来将其截取到 0，这可以防止物体被从后面来的光源照亮。

4） 高光反射

这里的高光反射是一种经验模型，也就是说，它并不完全符合真实世界中的高光反射现象。它可用于计算那些沿着完全镜面反射方向被反射的光线，这可以让物体看起来是有光泽的，例如金属材质。

计算高光反射需要知道的信息比较多，如表面法线、视角方向、光源方向、反射方向等。我们假设这些矢量都是单位矢量。

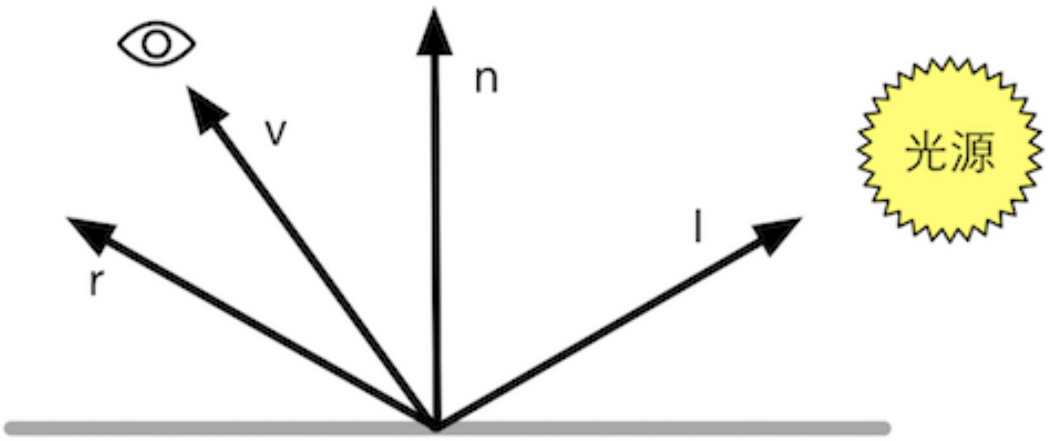


图93.　　使用 Phong 模型计算高光反射

在这四个矢量中，我们实际上只需要知道其中 3 个矢量即可，而第四个矢量——反射方向可以通过其他信息计算得到。

$$\hat{r} = 2(\hat{n} \cdot \hat{l})\hat{n} - \hat{l}$$

这样，我们就可以利用 Phong 模型来计算高光反射的部分。

$$C_{\text{specular}} = (C_{\text{light}} \cdot m_{\text{specular}}) \max(0, \hat{v} \cdot \hat{r})^{m_{gloss}}$$

其中，m_{gloss}是材质的光泽度（gloss），也被称为反光度（shininess）。它用于控制高光区域的“亮点”有多宽，m_{gloss}越大，亮点就越小。m_{specular}是材质的高光反射颜色，它用于控制该材质对于高光反射的强度和颜色。C_{light}则是光源的颜色和强度。同样，这里也需

要防止$\hat{v}\cdot\hat{r}$的结果为负数（使用 max 函数，防止点乘结果为负）。

和上述的 Phong 模型相比，Blinn 提出了一个简单的修改方法来得到类似的效果。它的基本思想是，避免计算反射方向$\hat{r}$。为此，Blinn 模型引入了一个新的矢量$\hat{h}$，它是通过对$\hat{v}$和$\hat{l}$的取平均后再归一化得到的。

$$即\hat{h}=\frac{\hat{v}+\hat{l}}{|\hat{v}+\hat{l}|}$$

然后，使用$\hat{n}$和$\hat{h}$之间的夹角进行计算，而非$\hat{v}$和$\hat{r}$之间的夹角。

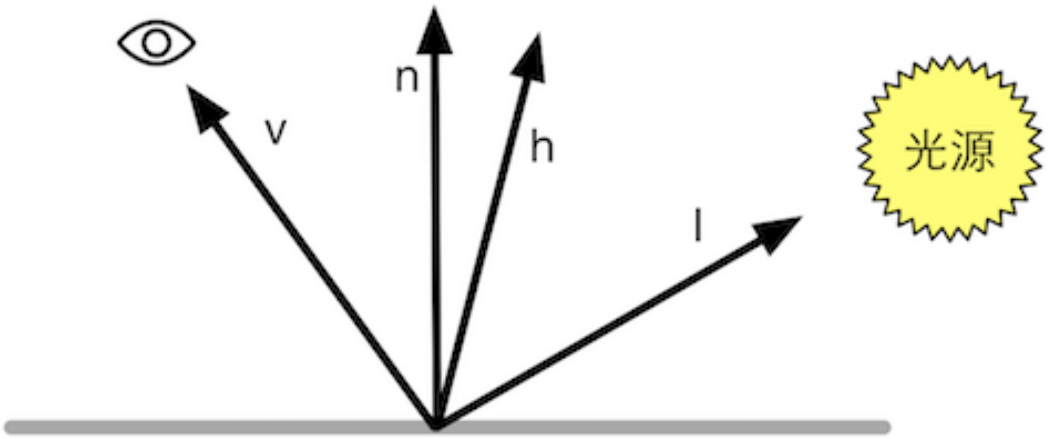


图94. Blinn 模型

总结一下，Blinn 模型的公式如下。

$$C_{specular}=(C_{light}\cdot m_{specular})\max(\max(0,\hat{n}\cdot\hat{h})^{m_{gloss}}$$

在硬件实现时，如果摄像机和光源距离模型足够远的话，Blinn 模型会快于 Phong 模型，这是因为，此时可以认为$\hat{v}$和$\hat{l}$都是定值，因此将$\hat{h}$是一个常量。

但是，当$\hat{v}$或者$\hat{l}$不是定值时，Phong 模型可能反而更快一些。需要注意的是，这两种光照模型都是经验模型，也就是说，我们不应该认为 Blinn 模型是对“正确的”Phong 模型的近似。实际上，在一些情况下，Blinn 模型更符合实验结果。

5） 逐像素还是逐顶点

上面，我们给出了基本光照模型使用的数学公式，那么我们在哪里计算这些光照模型呢？通常来讲，我们有两种选择，一种是在片元着色器中计算，也被称为逐像素光照（per-pixel lighting）；另一种在顶点着色器中计算，也被称为逐顶点光照（per-vertex lighting）。

在逐像素光照中，我们会以每个像素为基础，得到它的法线（可以是对顶点法线插值得到的，也可以是从法线纹理中采样得到的），然后进行光照模型的计算。这种在面片之间对顶点法线进行插值的技术被称为 Phong 着色（Phong shading），也被称为 Phong 插值或法线插值着色技术。这不同于我们之前讲到的 Phong 光照模型。

与之相对的是逐顶点光照，也被称为高洛德着色（Gouraud shading）。在逐顶点光照中，我们在每个顶点上计算光照，然后会在渲宣染图元内部进行线性插值，最后输出成像素颜色。由于顶点数目往往远小于像素数目，因此逐顶点光照的计算量往往要小于逐像素光照。但是，由于逐顶点光照依赖于线性插值来得到像素光照，因此，当光照模型中有非线性的计算（例如计算高光反射时）时，逐顶点光照就会出问题。

而且，由于逐顶点光照会在渲染图元内部对顶点颜色进行插值，这会导致渲染图元内

部的颜色总是暗于顶点处的最高颜色值，这在某些情况下会产生明显的棱角现象。

6） 总结

虽然标准光照模型仅仅是一个经验模型，也就是说，它并不完全符合真实世界中的光照现象。但由于它的易用性、计算速度和得到的效果都比较好，因此仍然被广泛使用。而也是由于它的广泛使用性，这种标准光照模型有很多不同的叫法。例如，一些资料中称它为 Phong 光照模型，因为裴祥风（Bui Tuong Phong）首先提出了使用漫反射和高光反射的和来对反射光照进行建模的基本思想，并且提出了基于经验的计算高光反射的方法（用于计算漫反射光照的兰伯特模型在那时已经被提出了）。

而后，由于 Blinn 的方法简化了计算而且在某些情况下计算更快，我们把这种模型称为 Blinn-Phong 光照模型。

3. 实现漫反射光照模型

4. 实现高光反射光照模型

十七、 基础纹理

纹理最初的目的就是使用一张图片来控制模型的外观。使用纹理映射(texture mapping)技术，我们可以把一张图“黏”在模型表面，逐纹素(texel)地控制模型的颜色（纹素是为了和像素区分开来）。

在美术人员建模的时候，通常会在建模软件中利用纹理展开技术，把纹理映射坐标(texture-mapping coordinates)存储在每个顶点上。纹理映射坐标定义了该顶点在纹理中对应的 2D 坐标。通常，这些坐标使用一个二维变量(u,v)来表示，其中 u 是横向坐标，而 v 是纵向坐标。因此，纹理映射坐标也被称为 UV 坐标。

尽管纹理的大小可以是多种多样的，例如可以是 256×256 或者 1024×1024，但顶点 UV 坐标的范围通常都被归一化到［0，1］范围内。需要注意的是，纹理采样时使用的纹理坐标不一定是在［0，1］范围内。实际上，这种不在［0，1］范围内的纹理坐标有时会非常有用。与之关系紧密的是纹理的平铺模式(Tiled)，它将决定宣染引擎在遇到不在［0，1］范围内的纹理坐标时如何进行纹理采样。

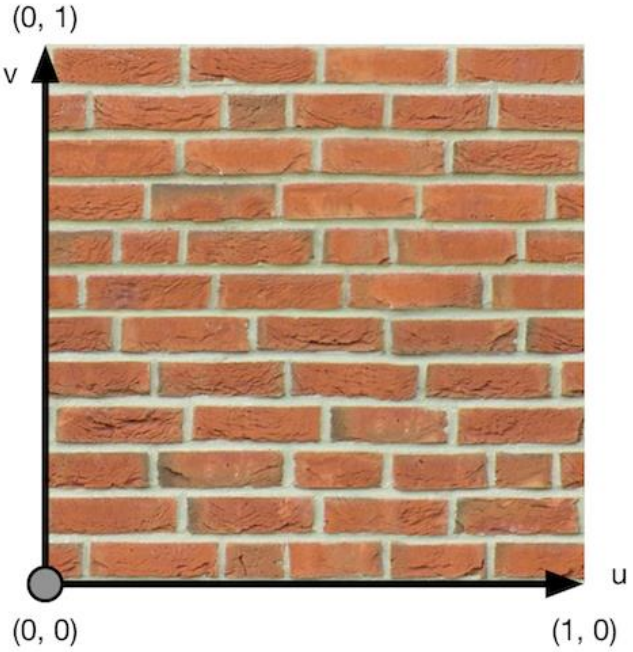


图95. 纹理坐标示例

我们曾不止一次地提到过 OpenGL 和 DirectX 在二维纹理空间中的坐标系差异问题。重要的事情要说很多次，我们再来回顾一下。在 OpenGL 里，纹理空间的原点位于左下角，而在 DirectX 中，原点位于左上角。幸运的是，Unity 在绝大多数情况下为我们处理好了这个差异问题，也就是说，即便游戏的目标平台可能既有 OpenGL 风格的，也有 DirectX 风格的，但我们在 Unity 中使用的通常只有一种坐标系。Unity 使用的纹理空间是符合 OpenGL 的传统的，也就是说，原点位于纹理左下角。

1. 单张纹理
2. 凹凸映射
3. 渐变纹理
4. 遮罩纹理

十八、 透明效果

透明是游戏中经常要使用的一种效果，在实时渲染中要实现透明效果，通常会在渲染模型时控制它的透明通道（Alpha channel）。当开启透明混合后，当一个物体被渲染到屏幕上时，每个片元除了颜色值和深度值之外，它还有另一个属性——透明度。当透明度为 1 时，表示该像素是完全不透明的，而当其为 0 时，则表示该像素完全不会显示。

在 Unity 中，我们通常使用两种方法来实现透明效果。第一种是使用透明度测试（Alpha Test）这种方法其实无法得到真正的半透明效果；另一种是透明度混合（Alpha Blending）。

当场景中包含很多模型时，我们并没有考虑是先渲染 A，再渲染 B，最后再渲染 C，还是按照其他的顺序来宣染。事实上，对于不透明（opaque）物体，不考虑它们的渲染顺序也能得到正确的排序效果，这是由于强大的深度缓冲（depth buffer，也被称为 z-buffer）的存在。在实时渲染中，深度缓冲是用于解决可见性（visibility）问题的，它可以决定哪个物体的哪些部分会被渲染在前面，而哪些部分会被其他物体遮挡。

它的基本思想是，根据深度缓存(z-buffer)中的值来判断该片元距离摄像机的距离，当渲染一个片元时，需要把它的深度值和已经存在于深度缓冲中的值进行比较（如果开启了深度测试），如果它的值距离摄像机更远，那么说明这个片元不应该被渲染到屏幕上（有物体挡住了它）；否则，这个片元应该覆盖掉此时颜色缓冲(color buffer)中的像素值，并把它的深度值更新到深度缓冲(z-buffer)中（如果开启了深度写入）。

使用深度缓冲，可以让我们不用关心不透明物体的渲染顺序，例如 A 挡住 B，即便我们先渲染 A 再渲染 B 也不用担心 B 会遮盖掉 A，因为在进行深度测试时会判断出 B 距离摄像机更远，也就不会写入到颜色缓冲中。但如果想要实现透明效果，事情就不那么简单了，这是因为，当使用透明度混合时，我们关闭了深度写入（ZWrite）。

简单来说，透明度测试和透明度混合的基本原理如下。

1. 透明度测试

它采用一种“霸道极端”的机制，只要一个片元的透明度不满足条件（通常是小于某个阈值），那么它对应的片元就会被舍弃。被舍弃的片元将不会再进行任何处理，也不会对颜色缓冲产生任何影响；否则，就会按照普通的不透明物体的处理方式来处理它，即进行深度测试、深度写入等。也就是说，透明度测试是不需要关闭深度写入的，它和其他不透明物体最大的不同就是它会根据透明度来舍弃一些片元。虽然简单，但是它产生的效果也很极端，要么完全透明，即看不到，要么完全不透明，就像不透明物体那样。

腾讯课堂沐元游戏人生

2. 透明度混合

这种方法可以得到真正的半透明效果。它会使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色值进行混合，得到新的颜色。

但是，透明度混合需要关闭深度写入（我们下面会讲为什么需要关闭），这使得我们要非常小心物体的渲染顺序。需要注意的是，透明度混合只关闭了深度写入，但没有关闭深度测试。这意味着，当使用透明度混合渲染一个片元时，还是会比较它的深度值与当前深度缓冲中的深度值，如果它的深度值距离摄像机更远，那么就不会再进行混合操作。这一点决定了，当一个不透明物体出现在一个透明物体的前面，而我们先渲染了不透明物体，它仍然可以正常地遮挡住透明物体。也就是说，对于透明度混合来说，深度缓冲是只读的。

3. 渲染顺序的重要性

前面说到，对于透明度混合技术，需要关闭深度写入，此时我们就需要小心处理透明物体的染顺序。那么，我们为什么要关闭深度写入呢？

如果不关闭深度写入，一个半透明表面背后的表面本来是可以透过它被我们看到的，但由于深度测试时判断结果是该半透明表面距离摄像机更近，导致后面的表面将会被剔除，我们也就无法透过半透明表面看到后面的物体了。但是，我们由此就破坏了深度缓冲的工作机制，而这是一个非常非常非常（重要的事情要讲 3 遍）糟糕的事情，尽管我们不得不这样做。

关闭深度写入导致渲染顺序将变得非常重要。我们来考虑最简单的情况，假设场景里有两个物体 A 和 B，如下图所示，其中 A 是半透明物体，而 B 是不透明物体。

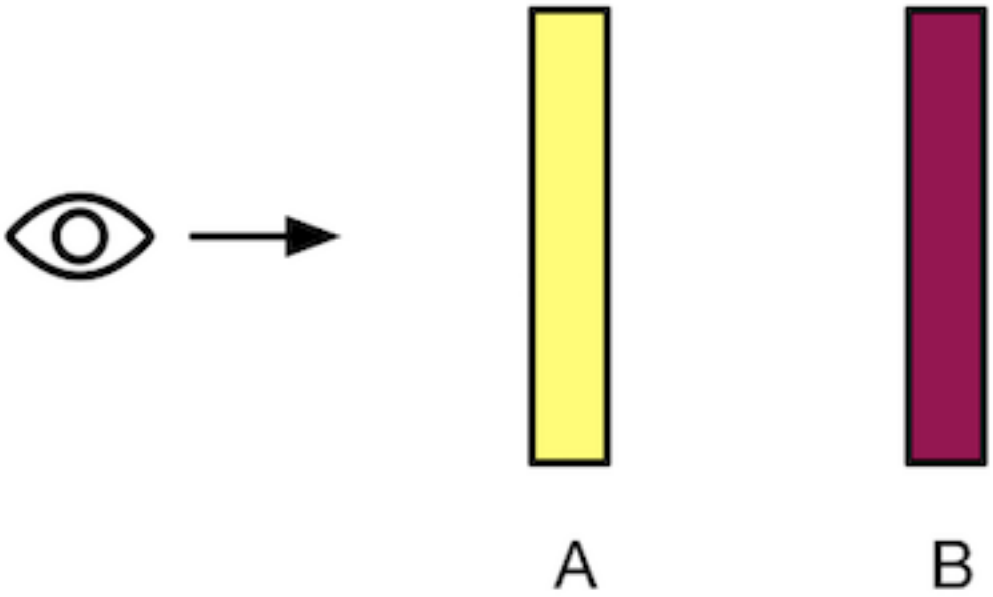


图96. A 半透明，B 不透明

我们来考虑不同的渲染顺序会有什么结果。

第一种情况，我们先渲染 B，再渲染 A。那么由于不透明物体开启了深度测试和深度写入，而此时深度缓冲中没有任何有效数据，因此 B 首先会写入颜色缓冲和深度缓冲。随后我们渲染 A，透明物体仍然会进行深度测试，因此我们发现和 B 相比 A 距离摄像机更近，因此，我们会使用 A 的透明度来和颜色缓冲中的 B 的颜色进行混合，得到正确的半透明效果。

第二种情况，我们先渲染 A，再渲染 B。渲染 A 时，深度缓冲区中没有任何有效数据，因此 A 直接写入颜色缓冲，但由于对半透明物体关闭了深度写入，因此 A 不会修改深度缓冲。等到渲染 B 时，B 会进行深度测试，它发现，“咦，深度缓存中还没有人来过，那我就放心地写入颜色缓冲了！”，结果就是 B 会直接覆盖 A 的颜色。从视觉上来看，B 就出现在了 A 的前面，而这是错误的。

从这个例子可以看出，当关闭了深度写入后，渲染顺序是多么重要。由此我们知道，我们应该在不透明物体渲染完之后再渲染半透明物体。那么，如果都是半透明物体，渲染顺序还重要吗？答案是肯定的。

还是假设场景里有两个物体 A 和 B，如下图所示，其中 A 和 B 都是半透明物体。

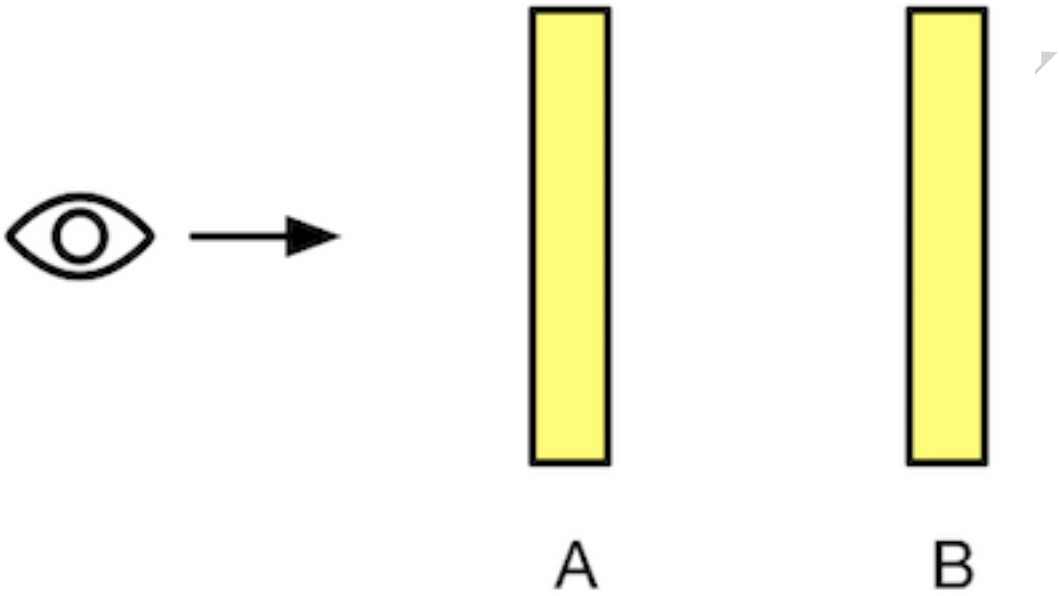


图97. 两个都是半透明物体

我们还是考虑不同的渲染顺序有什么不同结果。

第一种情况，我们先渲染 B，再渲染 A。那么 B 会正常写入颜色缓冲，然后 A 会和颜色缓冲中的 B 颜色进行混合，得到正确的半透明效果。

第二种情况，我们先渲染 A，再渲染 B。那么 A 会先写入颜色缓冲，随后 B 会和颜色缓冲中的 A 进行混合，这样混合结果会完全反过来，看起来就好像 B 在 A 的前面，得到的就是错误的半透明结构。

从这个例子可以看出，半透明物体之间也是要符合一定的渲染顺序的。

基于这两点，渲染引擎一般都会先对物体进行排序，再渲染。常用的方法是。

① 先渲染所有不透明物体，并开启它们的深度测试和深度写入。

② 把半透明物体按它们距离摄像机的远近进行排序，然后按照从后往前的顺序渲染这些半透明物体，并开启它们的深度测试，但关闭深度写入。

那么，问题都解决了吗？不幸的是，仍然没有。在一些情况下，半透明物体还是会出现“穿帮镜头”。如果我们仔细想想的话，上面给出的第 2 步中渲染顺序仍然是含糊不清的一“按它们距离摄像机的远近进行排序”，那么它们距离摄像机的远近是如何决定的呢？读者可能会马上脱口而出，“就是距离摄像的深度值嘛！”但是，深度缓冲中的值其实是像素级别的，即每个像素有一个深度值，但是现在我们对单个物体级别进行排序，这意味着排序结果是，要么物体 A 全部在 B 前面渲染，要么 A 全部在 B 后面渲染。但如果存在循环

重叠的情况，那么使用这种方法就永远无法得到正确的结果。

图98.　　3 个物体循环重叠的情况

在上图中，由于 3 个物体互相重叠，我们不可能得到一个正确的排序顺序。这种时候，我们可以选择把物体拆分成两个部分，然后再进行正确的排序。但即便我们通过分割的方法解决了循环覆盖的问题，还是会有其他的情况来”捣乱”。

哔哩哔哩次元弹幕主

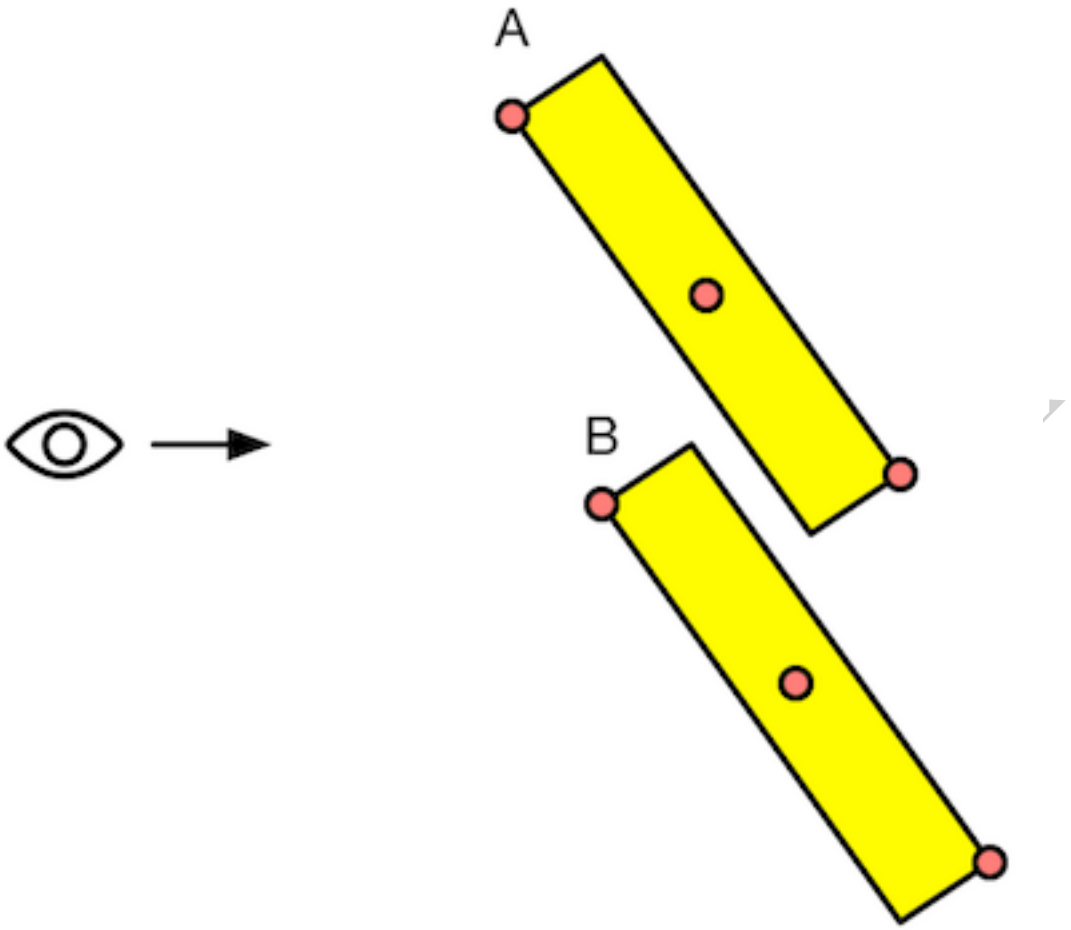


图99. 使用哪个深度对物体进行排序

上图红色点分别标明了网格上距离摄像机最近的点、最远的点以及网格中点，这里的问题是如何排序？我们知道，一个物体的网格结构往往占据了空间中的某一块区域，也就是说，这个网格上每一个点的深度值可能都是不一样的，我们选择哪个深度值来作为整个物体的深度值和其他物体进行排序呢？是网格中点吗？还是最远的点？还是最近的点？

不幸的是，对于上图中的情况，选择哪个深度值都会得到错误的结果，我们的排序结果总是 A 在 B 的前面，但实际上 A 有一部分被 B 遮挡了。这也意味着，一旦选定了一种判断方式后，在某些情况下半透明物体之间一定会出现错误的遮挡问题。这种问题的解决方法通常也是分割网格。

尽管结论是，总是会有一些情况打乱我们的阵脚，但由于上述方法足够有效并且容易实现，因此大多数游戏引擎都使用了这样的方法。为了减少错误排序的情况，我们可以尽可能让模型是凸面体，并且考虑将复杂的模型拆分成可以独立排序的多个子模型等。其实就算排序错误结果有时也不会非常糟糕，如果我们不想分割网格，可以试着让透明通道更加柔和，使穿插看起来并不是那么明显。我们也可以使用开启了深度写入的半透明效果来近似模拟物体的半透明。

4. Unity Shader 的渲染顺序

Unity 为了解决渲染顺序的问题，提供了渲染队列(render queue)这一解决方案。我们可以使用 SubShader 的 Queue 标签来决定我们的模型将归于哪个渲染队列。Unity 在内部使用一系列整数索引来表示每个渲染队列，且索引号越小表示越早被渲染。在 Unity 5 中，

Unity 提前定义了 5 个渲染队列，当然在每个队列中间我们可以使用其他队列。

5. 使用透明度测试

6. 使用透明度混合

7. 开启深度写入的半透明效果

十九、 有向距离函数-SDF-Sign Distance Function

1. 圆-Circle

符号距离函数（sign distance function），简称 SDF，又可以称为有向距离函数，在空间中的一个有限区域上确定一个点到区域边界的距离并同时对距离的符号进行定义：点在区域边界外部为正，内部为负，位于边界上时为 0。

简单来说，SDF 就是某一点到某一个区域边界的最短距离。

求某一点到圆的边的距离是最为简单的，这里分两种情况。大致来说都是一样的，就是用点与圆心的距离减去半径来进行判别。

1） 圆心在原点

我们只需要知道两个参数，任意点 P 的坐标和圆的半径 r，就可以求出点 P 距离圆边缘的距离。

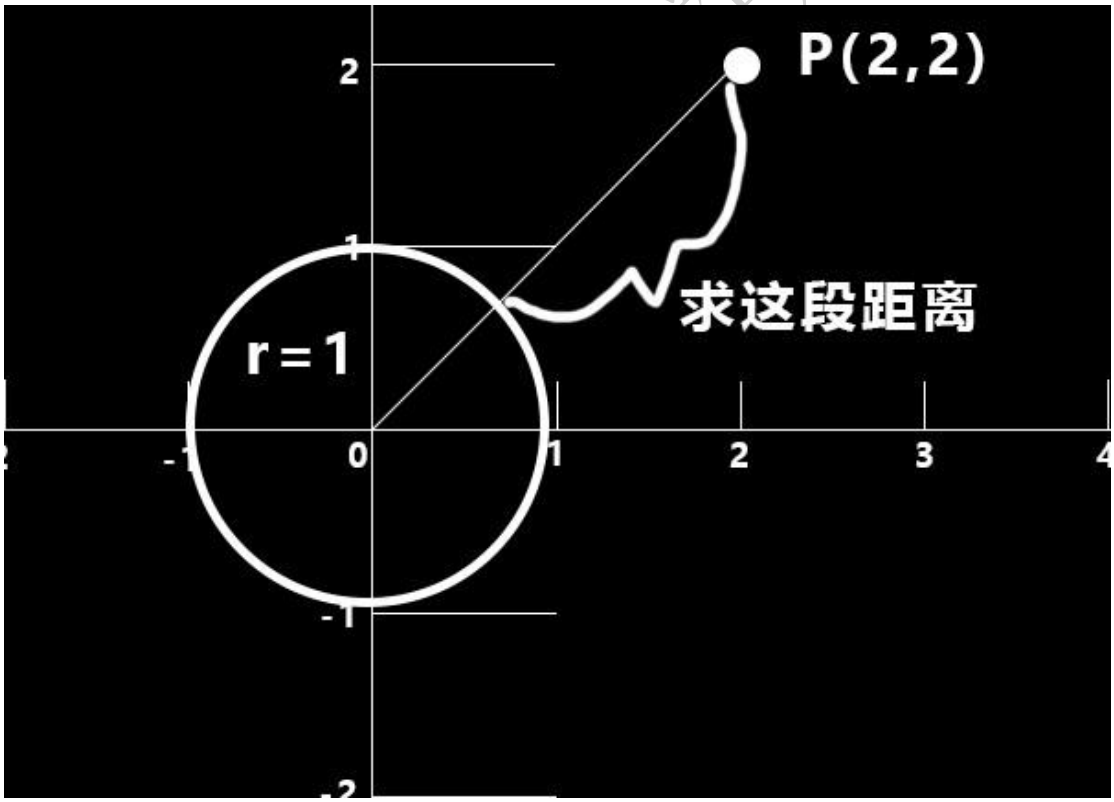


图100. 圆心在坐标轴原点

```
float myCircleSDF( in vec2 p, in float r )
{
    return length(p)-r;
}
```

2） 圆心不在原点上

圆心不在原点上就需要三个参数，圆心坐标、点 P 坐标、圆半径。首先计算圆心与点 P 的距离，然后再用半径去减。（因为求的是圆心与点的距离，这与向量的方向无关，因此不需要关心被减数是谁。）

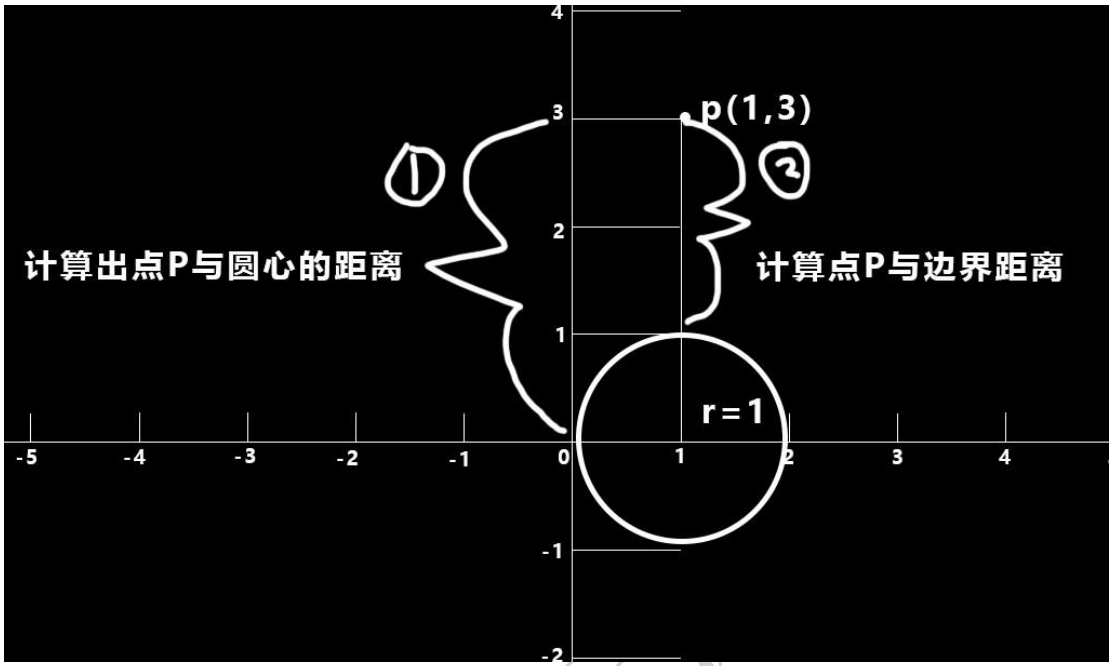


图101.　　圆心不在坐标轴原点

2. 线段-LineSegment

线段的推论很重要，后续很多类型都以此为基础进行。

线段的 SDF 输入参数有三个：任意点 p 的坐标，线段两端点 a，b 坐标。

哔哩哔哩

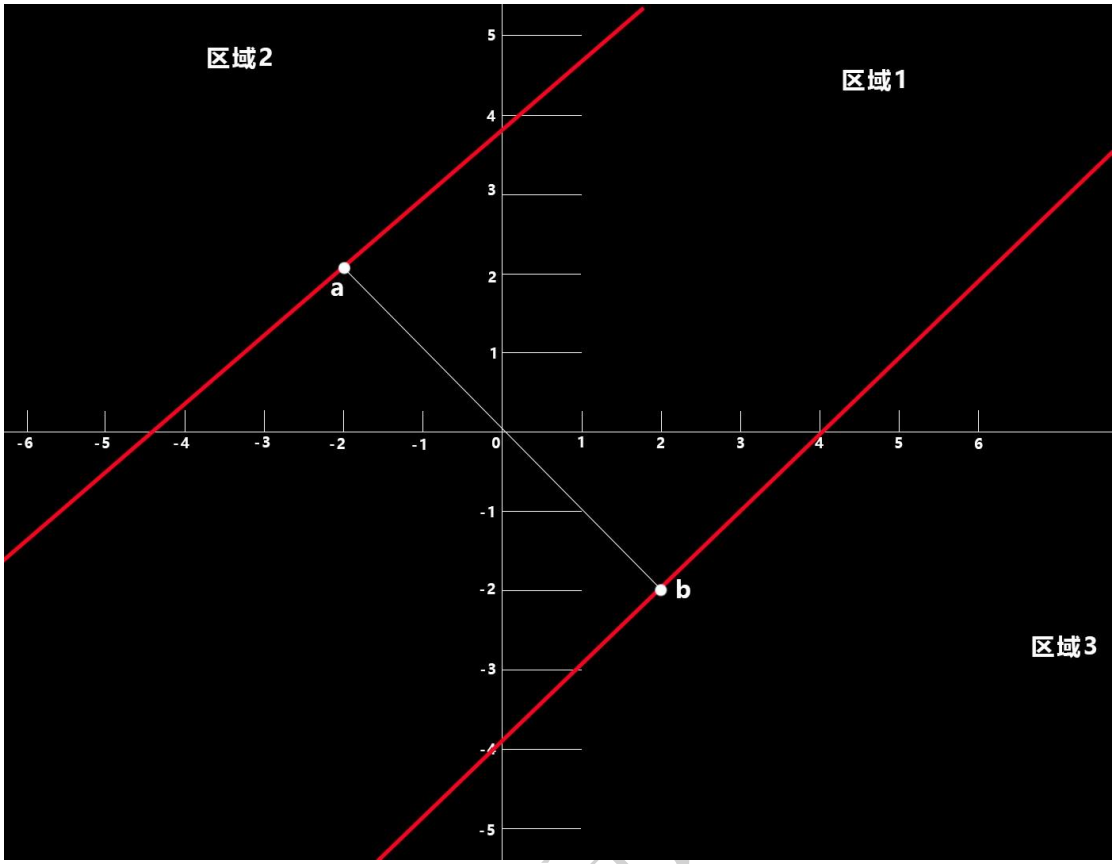


图102.　SDF 线段区域划分

如上图所示，一条线段可以被两条直线分为三个部分（两条红线都垂直于这条线段）。计算任意点 P 与线段的最短距离，会根据点 P 不同而有所变化。

哔哩哔哩

1） 位于区域一

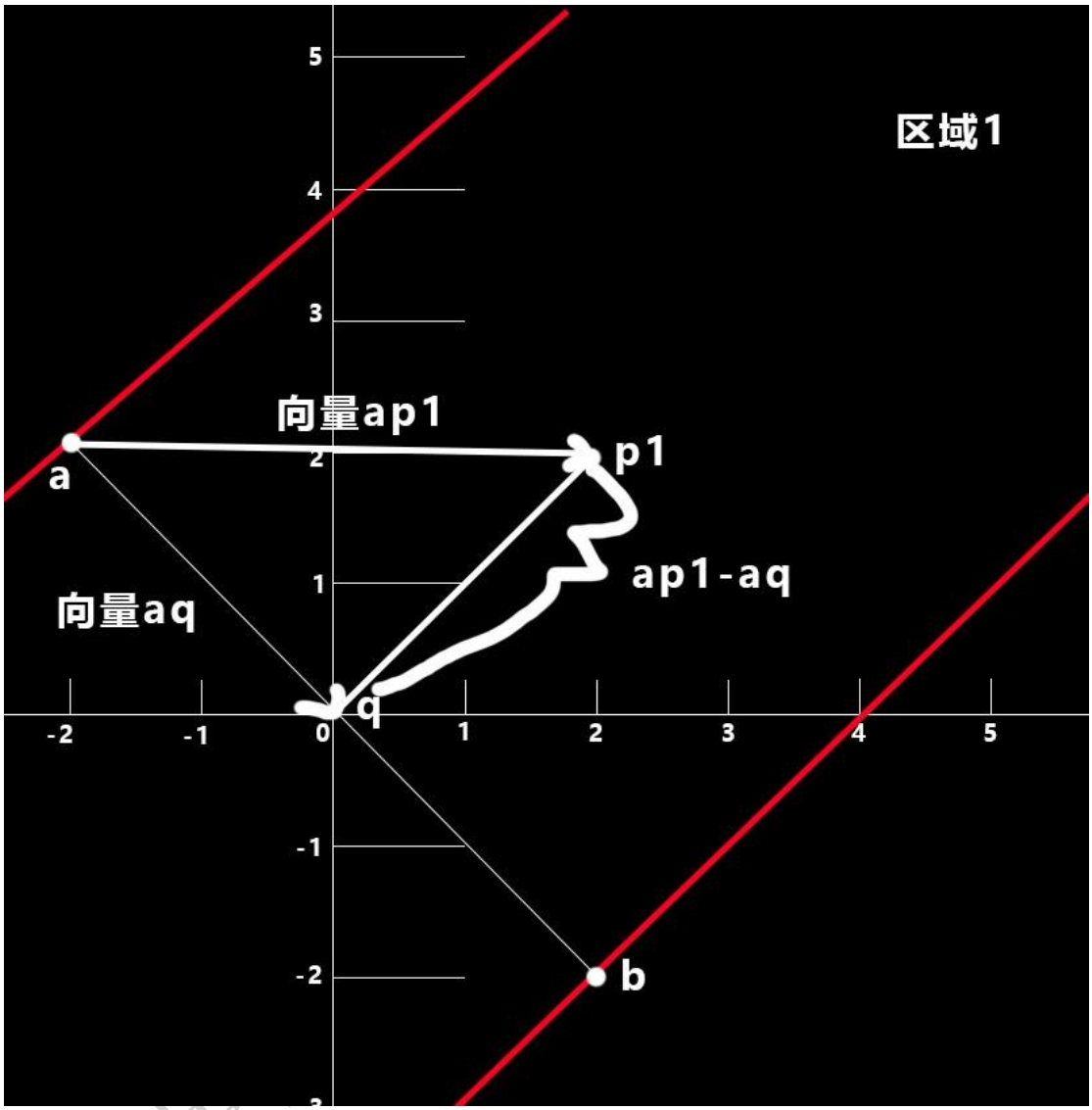


图103.　　点位于区域一内

当点位于区域一内，点到线段的最短距离，就是过点做一条垂直于线段的直线。如上图，点 p_1 到线段的最短距离就是向量$\overrightarrow{qp_1}$的大小，即 $d=\|\overrightarrow{qp_1}\|$。

由向量的运算法则可以知道，$\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r$[14]$\overrightarrow{ab}$。（r[15]表示向量$\overrightarrow{aq}$的大小与向量$\overrightarrow{ab}$大小的比值）

但是，在实际运用中我们并不知道点 q 的坐标，因此需通过现有的已知条件去进行计算。

如果我们知道点 q 的坐标，可以由三角函数$\cos\theta=\frac{|\overrightarrow{aq}|}{|\overrightarrow{ap_1}|}$（余弦=临边/斜边），计算出这两个向量的夹角，进而得到向量$\overrightarrow{qp_1}$的大小。

但是现在我们并不知道点 q 的坐标，因此我们无法计算出这个夹角，我们需要通过其他方法去进行计算。

[14] 这里的 r 是单词 Ratio 的缩写
[15] 这里的 r 是单词 Ratio 的缩写

我们根据向量数量积的公式$\overrightarrow{ab}\cdot\overrightarrow{ap_1}=|\overrightarrow{ab}||\overrightarrow{ap_1}|\cos\theta$以及三角函数$\cos\theta=\frac{|\overrightarrow{aq}|}{|\overrightarrow{ap_1}|}$。可以将数量积公式中的余弦剔除，得到$\overrightarrow{ab}\cdot\overrightarrow{ap_1}=|\overrightarrow{ab}||\overrightarrow{aq}|$。接着我们在等式两边同除$|\overrightarrow{ab}|$的平方，可以得到最终的等式$\frac{\overrightarrow{ab}\cdot\overrightarrow{ap_1}}{|\overrightarrow{ab}||\overrightarrow{ab}|}=\frac{|\overrightarrow{aq}|}{|\overrightarrow{ab}|}$=r。

2） 位于区域二和三

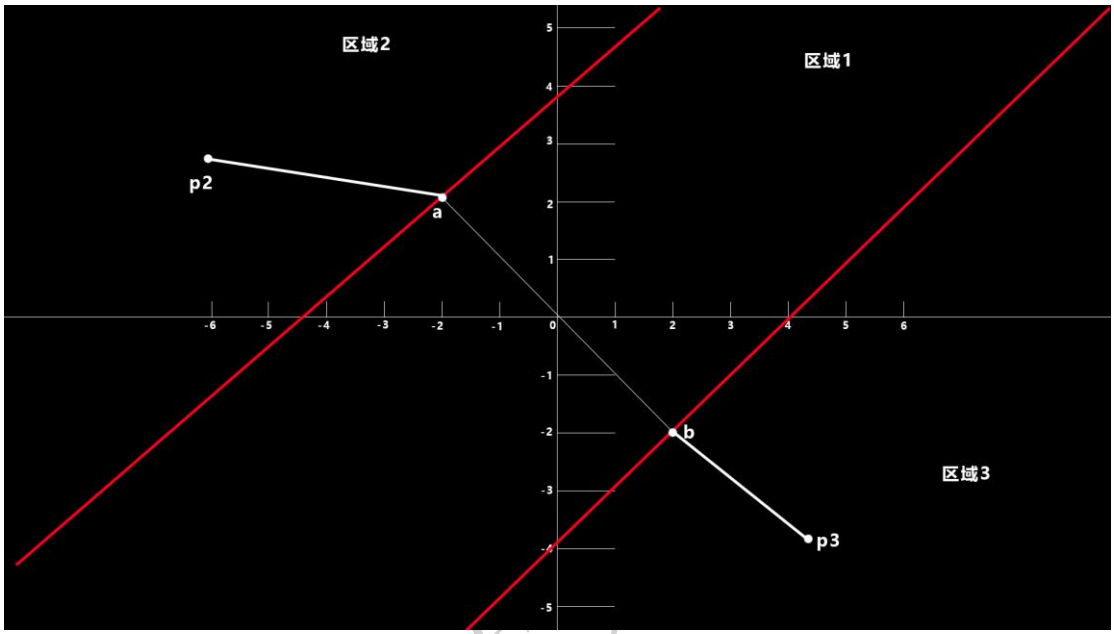


图104. 点位于区域二和三

当点位于区域二和三时，其最短距离就是到线段端点的距离。

区域二：d=$\|\overrightarrow{ap_2}\|$

区域三：d=$\|\overrightarrow{bp_3}\|$

此时，我们可以发现，区域二三的公式和区域一的其实非常类似，只不过区域二三的公式中并没有出现 r 这个变量。前面已经提到了，r 代表的是比值，即向量$\overrightarrow{aq}$的大小与向量$\overrightarrow{ab}$大小的比值。在区域二，我们可以认为比值结果为 0,；在区域三，我们可以认为比值结果为 1；这样就可以让所有区域都使用一个表达式。

区域一：d=$\|\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r^{16}\overrightarrow{ab}\|$

区域二：d=$\|\overrightarrow{ap_2}=\overrightarrow{ap_2}-0*\overrightarrow{ab}\|$

区域三：d=$\|\overrightarrow{bp_3}=\overrightarrow{ap_3}-1*\overrightarrow{ab}\|$

但是，比值是通过公式去计算$\frac{\overrightarrow{ab}\cdot\overrightarrow{ap_1}}{|\overrightarrow{ab}||\overrightarrow{ab}|}=\frac{|\overrightarrow{aq}|}{|\overrightarrow{ab}|}$=r 的，其结果并不被限制在 0 到 1。因此我们必须在结果计算出来后，对结果进行限制。

3） 代码

```
float SDFLineSegment (in vec2 p, in vec2 a, in vec2 b)
{
    vec2 ap = p-a;
    vec2 ab = b-a;
    float h = clamp(dot(ap,ab)/dot(ab,ab), 0.0, 1.0);
```

[16] 这里的 r 是单词 Ratio 的缩写

```
        return length(ap-h*ab);
    }
```

3. 矩形-Rectangle

对于矩形，我们可以很清楚的认识到，它的图像是关于中心点对称的。（中心点可能是原点也可能不是原点）

那么，要绘制一个矩形，所需的参数就只有宽和高两个。而点到矩形的距离公式与中心点有关，当矩形的中心点位于原点，就只需要两个参数，即宽高和目标点（如果宽和高使用 Float 类型，则是三个参数）；当矩形的中心点不是原点，就需要多加上一个参数，这个参数就是中心点的坐标。

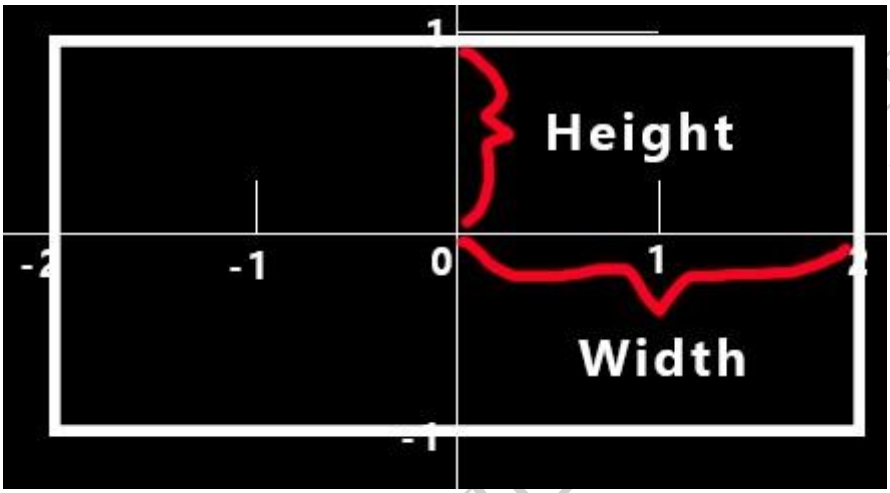


图105.　　矩形绘制原理

1） 矩形中心在原点

当矩形中心点在原点时，矩形的四个顶点就会分别处于四个象限中。因为矩形是关于中心点对称的，因此二、三、四象限的情况都可以在第一象限中处理。

要将目标点转换到第一象限，只需要对其取绝对值即可。

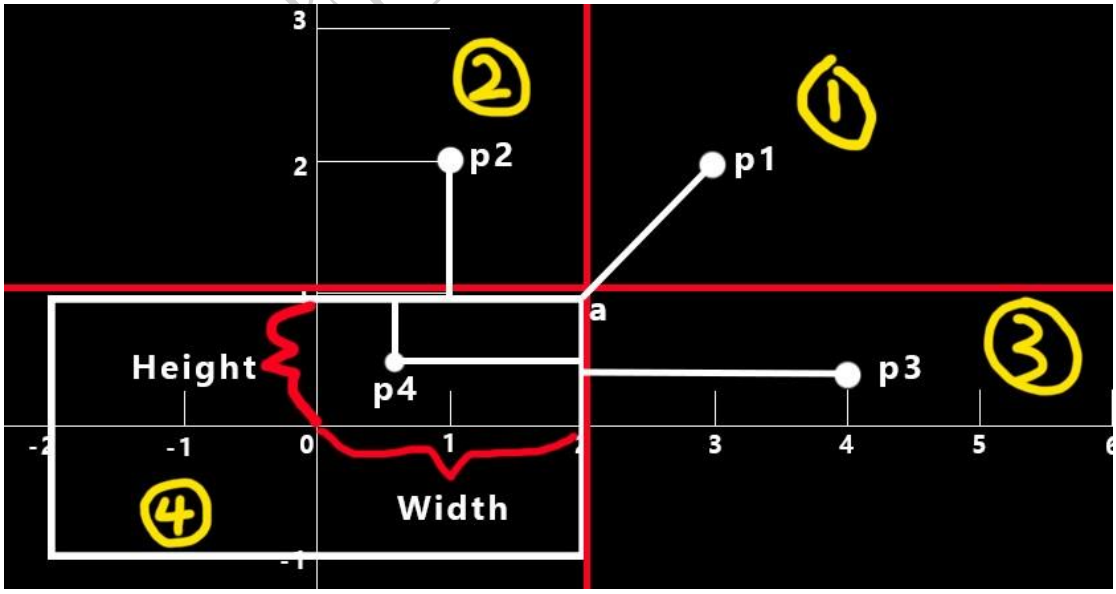


图106.　　目标点到矩形的情况划分

如上图所示，因为我们是在第一象限中处理点到矩形的距离。因此，便可以以第一象限的矩形顶点为原点，构建一个坐标轴，将目标点的情况划分为四种。

当目标点所处位置为第一种情况时，到矩形的最短距离就是向量$\overrightarrow{ap_1}$的大小。

$d=\|\overrightarrow{ap_1}\| = \sqrt{(p_1.x - a.x)^2 + (p_1.y - a.y)^2}$

当目标点所处位置为第二种情况时，到矩形的最短距离就是，目标点与顶点 Y 坐标的差。

$d=p_2.y - a.y$

当目标点所处位置为第三种情况时，到矩形的最短距离就是，目标点与顶点 X 坐标的差。

$d=p_3.x - a.x$

当目标点所处位置为第四种情况时，点位于矩形的内部，其到矩形的最短距离就是，目标点与顶点 X 坐标和 Y 坐标差值中的最大值。

$d=\max(p_4.x - a.x, p_4.y - a.y)$

因为，在第二种情况，有$p_2.x - a.x < 0$；第三种情况下，有$p_3.y - a.y < 0$。因此，我们使用 max 函数去统一前三种情况的公式，即

$d=\sqrt{(\max(p.x - a.x, 0))^2 + (\max(p.y - a.y, 0))^2}$

由于 max 函数可以直接对向量使用，所以可以将其进一步简化，得到

$d=\|\max(p - a, 0)\|$（由向量运算可知，p-a 就是向量ap）

现在我们再来单独考虑第四种情况，如果我们将第四个区域的点带入上面的公式，得到的值为 0，而前三个区域所得值一定大于 0。（因为在前三个区域，$p.x - a.x$和$p.y - a.y$至少有一个不为 0）

区域四中的点，其坐标数值一定小于顶点，所以使用$\max(p.x - a.x, p.y - a.y)$获得最短距离。又因为，我们要求公式只能适用于区域四的点，所以我们需要将其他区域排除，也就是获得区域四点的最短距离后，让其和 0 比较，取它们中最小的那个。（区域四的点和顶点相减必定小于 0）

整合上面信息，可以得到区域四的公式 $d= \min(\max(p.x - a.x, p.y - a.y), 0)$。（虽然该公式得到的区域四值为负，但我们会在程序中计算它的大小，所以并没有问题）

最后，我们将所有区域的公式进行一个整合，得到最终的公式。

$d=\|\max(p - a, 0)\| + \min(\max(p.x - a.x, p.y - a.y), 0)$

a. 代码

```
float SDFRectangle(in vec2 p, in vec2 a)
{
    vec2 q = abs(p)-a;
    return length(max(q, 0.0)) + min(max(q.x, q.y), 0.0);
}
```

2） 矩形中心不在原点

当矩形的中心点不在原点，我们就需要额外添加一个参数，即矩形的中心点。因为矩形中心点不在原点的情况过于多样，因此我们很难得到一个统一的公式。但是，因为我们计算的是距离，所以我们可以将矩形和目标点同时进行平移，保证矩形和目标点的相对位置不发生变化，然后使用中心点在原点的公式进行计算。

4. 圆角矩形- Rounded Rectangle

通过前面的内容，我们已经知道如何计算目标点到矩形和圆的最短距离了，下面我们就可以结合前面的内容去推导出目标点到圆角矩形的最短距离了。在这里，只讨论圆角矩

哔哩哔哩欢乐游戏人生

形中心点位于原点的情况，中心点不位于原点，就参照前面的方法将其移至原点。

1） 圆角半径相同

当圆角半径相同的时候，我们就无需关心圆角所处的象限，因此我们所有的计算都将在第一象限中进行，一旦有目标点不在第一象限，就对其取绝对值将其转化到第一象限。

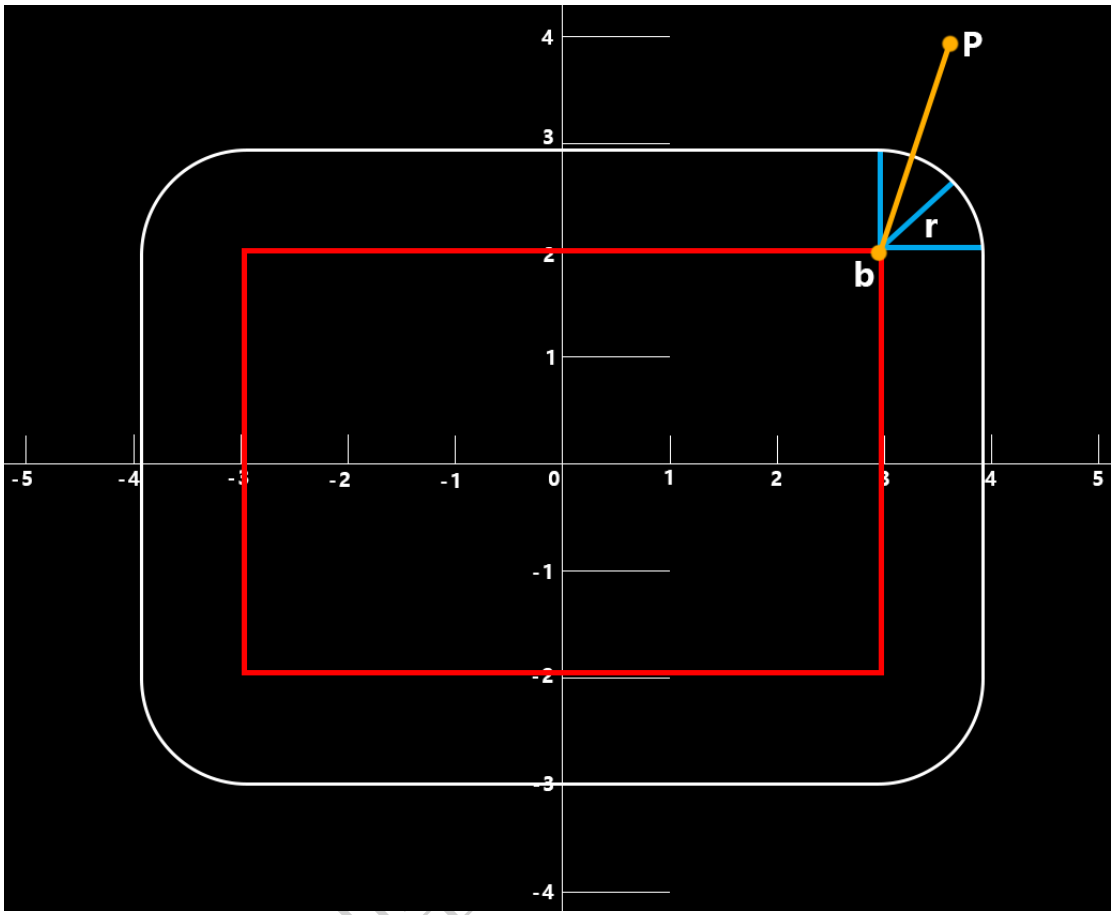


图107.　　目标点 P 到圆角矩形

通过上面的图像可以知道，目标点 P 到圆角矩形的最短距离，就是目标点到矩形的最短距离（矩形就是上面的红色部分）减去圆角的半径 R（半径就是蓝色部分）。

$$d = \|\overrightarrow{pb}\| - r$$

a. 得到初始矩形端点

要进行上面的计算，我们需要知道点 b 的坐标。点 b 的坐标如何得到呢？其实很简单，圆角矩形就是在矩形的基础上减去部分区域得到的，我们只需要将其重新恢复成一个矩形的模样（下面的绿色虚线部分，我这里为了能更清晰的展示，将初始矩形稍微调大了一点），再由圆角的半径就可以得到点 b 的坐标。

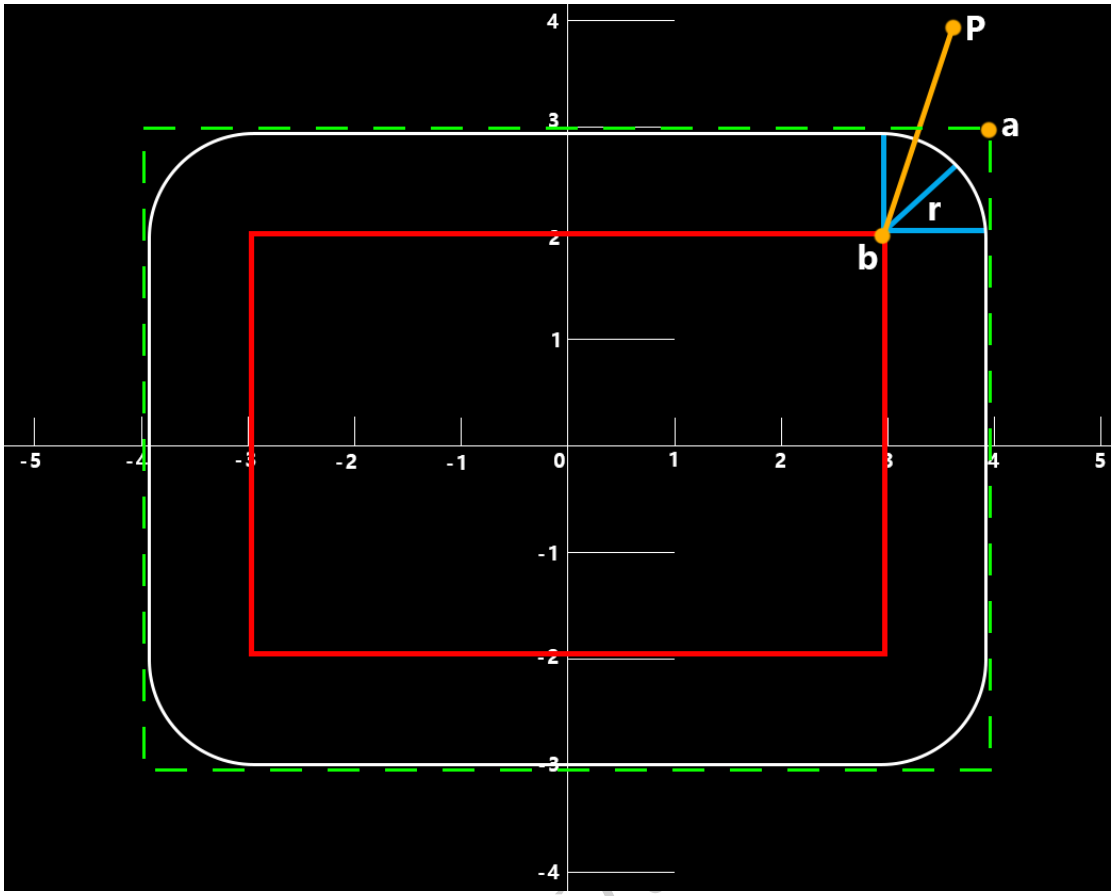


图108.　　根据圆角半径去推断出初始矩形

b.　计算点 b 的坐标

得到初始的矩形后，我们就可以通过它的端点坐标和圆角半径，进而得出点 b 的坐标。

$$b = (a._x - r, a._y - r)$$

c.　计算目标点到圆角矩形的最短距离

有了点 b 的坐标后，我们就可以根据之前得出的 SDF 矩形公式，去计算出目标点到矩形的最短距离 d。得到 d 后，再用圆角半径去减，即可得到目标点到圆角矩形的最短距离。

d.　Shadertoy 代码

```
float SDFRoundedRectangle ( in vec2 p, in vec2 a, in float r )
{
    //计算得到 q 向量，即bp向量。
    vec2 q = abs(p)-a+r;//  相当于 abs(p)-(a-r)
    return min(max(q.x,q.y),0.0) + length(max(q,0.0)) - r;
```

}

2） 圆角半径不同

上面讲解了如何计算圆角半径一致时的 SDF，但在实际情况下，圆角矩形的 4 个圆角半径可能都是不同的。所以，我们在计算 SDF 时，需要根据目标点所处的象限，去选择对应的圆角。

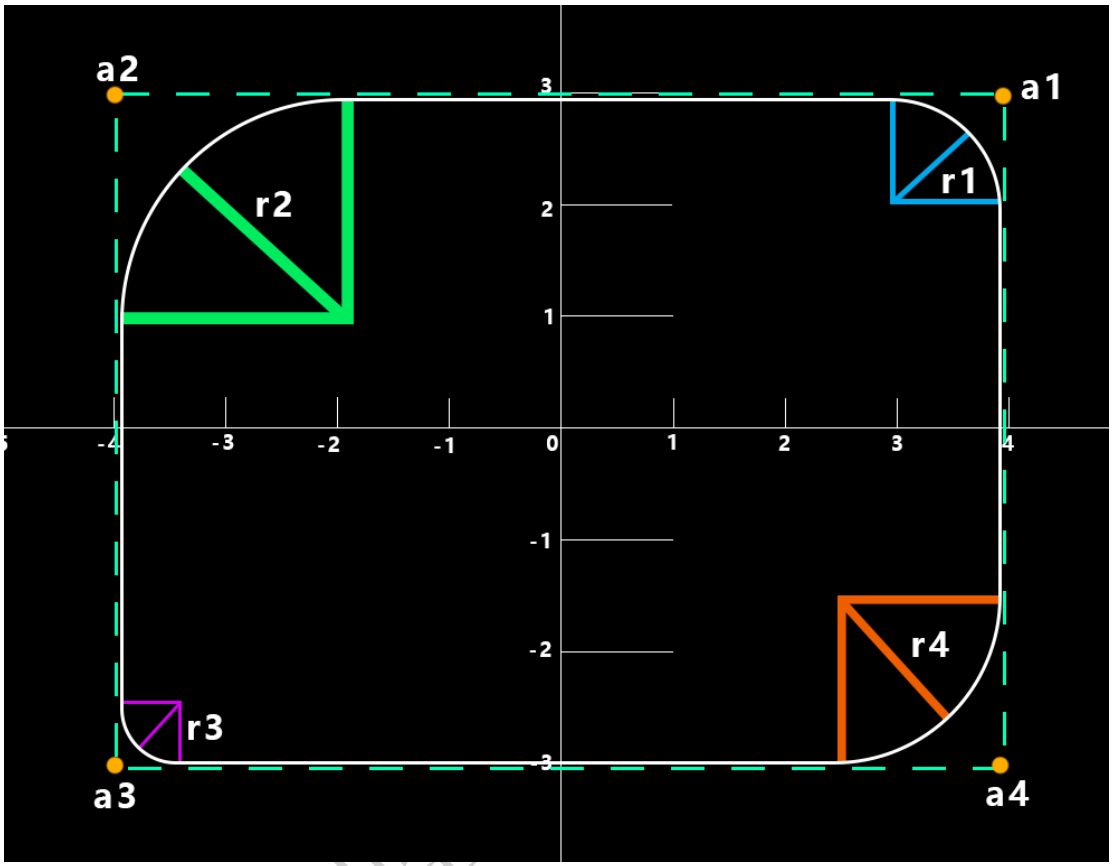


图109.　　圆角半径不同

a. 使用四维向量存储圆角半径

我们知道圆角矩形是具有四个圆角的，在设置参数时我们可以使用四个 float 型的参数，但是这样会使函数的参数列表过多。这时，我们就可以考虑使用一个四维向量去存储，这个四维向量的分量，分别对应一、二、三、四象限的圆角半径。

b. 判断目标点对应的圆角

在上面，我们已经使用四维向量将四个圆角半径存储了下来，接下来就是先根据目标点的 X 分量，去判断目标点是处于一、四象限还是二、三象限，排除部分不符合的圆角。

```
r.xy = (p.x > 0.0) ? r.xw : r.yz;
```

当我们使用完目标点的 X 分量后，再继续使用它的 Y 分量，去得到与目标点相对应的那一个圆角。

```
r.x = (p.y > 0.0) ? r.x : r.y;
```

c. Shadertoy 代码

```
//一、二、三、四象限圆角半径大小分别对应 r.x，r.y，r.z，r.w
float SDFRoundedRectangle_Different (in vec2 p, in vec2 a, in vec4 r)
{
    //判断 p 在哪个象限
    r.xy = (p.x>0.0)?r.xw:r.yz;
    r.x = (p.y>0.0)?r.x:r.y;
    //求最短距离
    vec2 q = abs(p)-a + r.x;
    return length(max(q, 0.0)) + min(max(q.x, q.y), 0.0) - r.x;
}
```

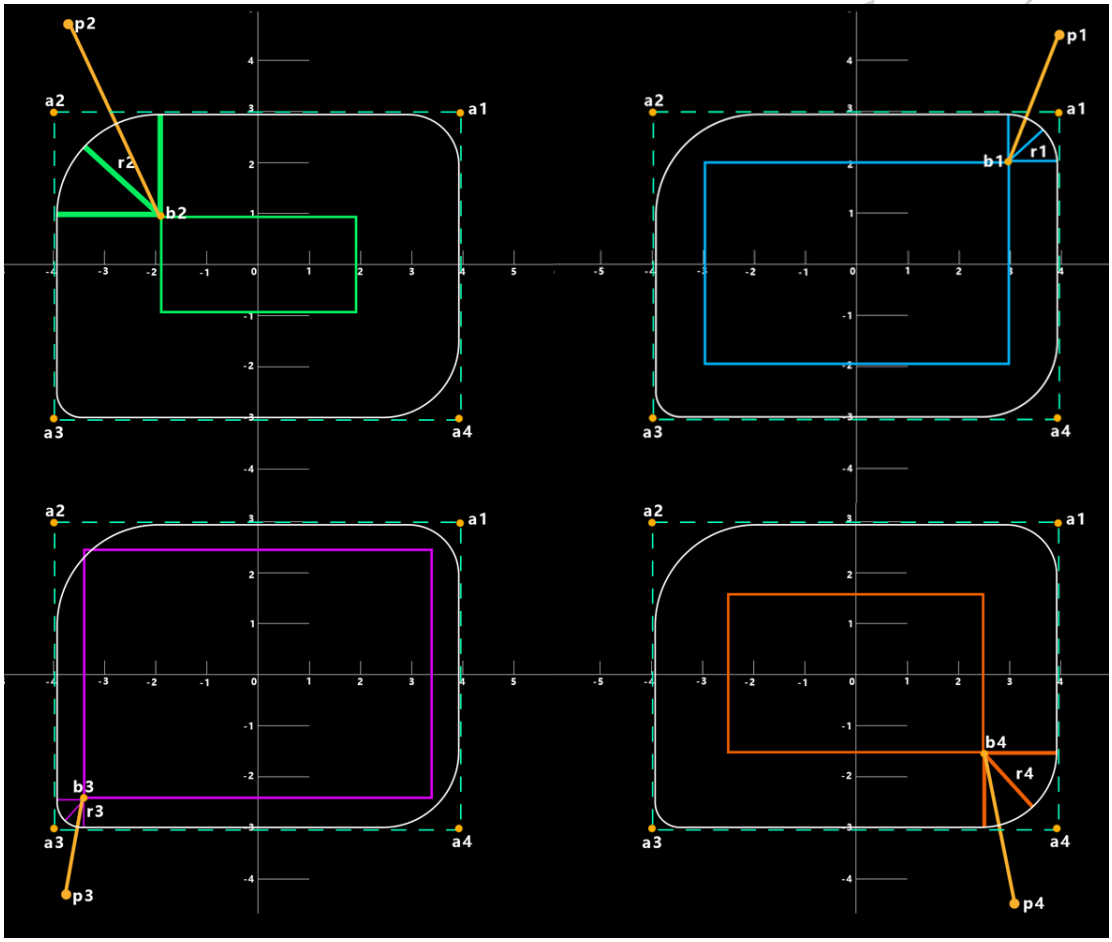


图110.　　圆角半径不同下的情况

5. 菱形-Rhombus

假设菱形的中心点在原点，菱形的 SDF 的输入参数有两个，一个是目标点 P 的坐标，另一个是二维向量 b。

菱形的推导过程和线段的思想是一致的，只不过这里并没有选择线段的两个端点，而是选择了线段的中点和一个端点（下图的点 a 和点 c），这也是我们在代码中使用 clamp 函数时，设置其范围为-1.0 到 1.0 的原因。

1） 推导过程

首先我们需要知道几个重要点的坐标，因为菱形是一个中心对称图像，因此所有的目标点都可以将其转换到第一象限（也就是对目标点 P 取绝对值）。下面的运算将涉及到向量的点乘，也即数量积。

$a(0,b.y)$ $d(b.x,0)$ $c\left(\frac{b.x}{2},\frac{b.y}{2}\right)$ $p(p.x,p.y)$ （这里的点 p 是取绝对值后的，二维向量 b 代表菱形的宽和高）

$$h=\frac{\overrightarrow{cp}\cdot\overrightarrow{ca}}{\overrightarrow{ca}\cdot\overrightarrow{ca}}=\frac{\left(p.x-\frac{1}{2}b.x,p.y-\frac{1}{2}b.y\right)\cdot\left(-\frac{1}{2}b.x,\frac{1}{2}b.y\right)}{\left(-\frac{1}{2}b.x\ ,\ \frac{1}{2}b.y\right)\cdot\left(-\frac{1}{2}b.x\ ,\ \frac{1}{2}b.y\right)}$$

$$=\frac{\left(-\frac{1}{2}p.x*b.x+\frac{1}{4}b.x*b.x\right)+\left(-\frac{1}{2}p.y*b.y-\frac{1}{4}b.y*b.y\right.}{\frac{1}{4}b.x*b.x+\frac{1}{4}b.y*b.y}$$

$$=\frac{-2*(p.x*b.x-p.y*b.y)+(b.x*b.x-b.y*b.y)}{b.x*b.x+b.y*b.y}\left(\text{将分母的}\frac{1}{4}\text{移至了分子}\right)$$

$$=(-2*ndot(p,b)+ndot(b,b))/dot(b,b)$$

这里的 ndot 其实代表的就是上面的$(p.x*b.x-p.y*b.y)$这个方法，这里只是为了看起来方便封装了一个方法。由于点 c 是线段 ad 中点，所以 h 的取值范围是-1≤h≤1。

通过前面的图像我们可以知道，$\overrightarrow{qp}=\overrightarrow{op}-\overrightarrow{oq}$（这里是向量的减法运算）。

所以只要求出$\overrightarrow{oq}$，就可以得到最终结果。（q 点是直线 ad 上的点，也就是过点 P 做一条垂直于线段 ad 的直线，其与线段 ad 的交点就是点 q）

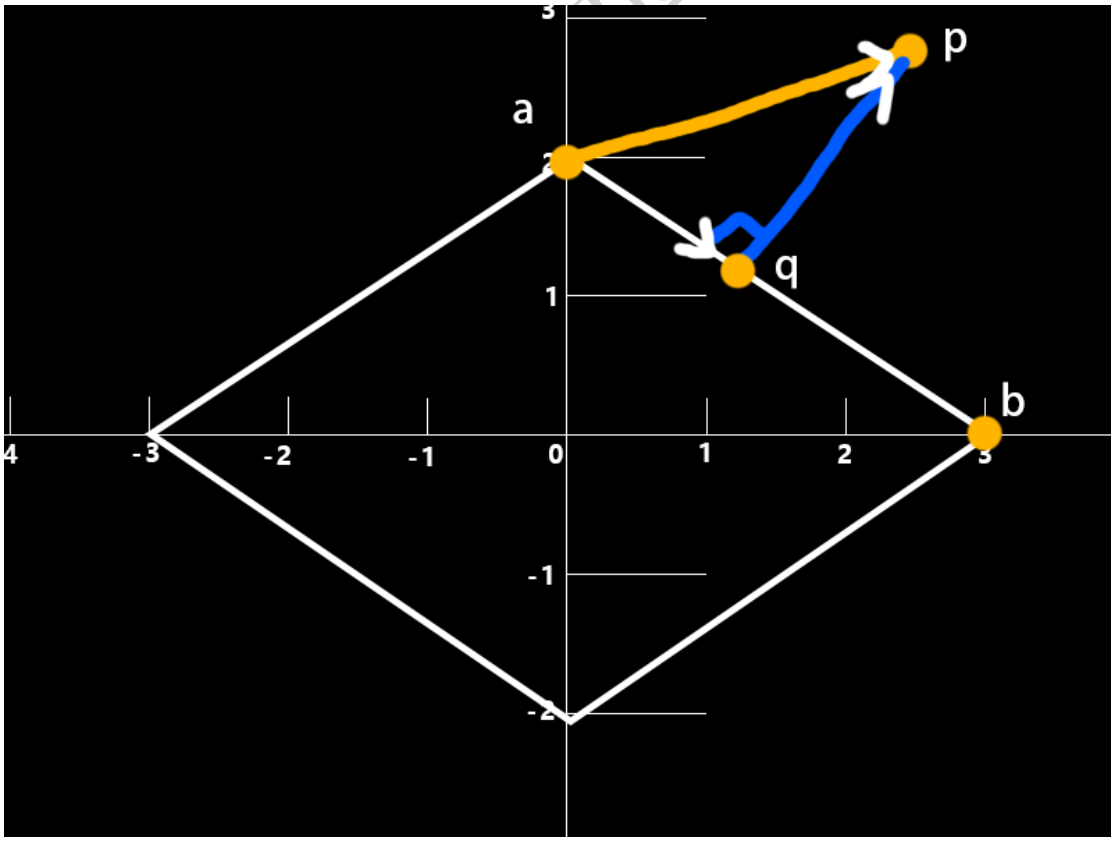


图111.　　点到一条线段的最短距离 qp 向量

我们由前面的线段 SDF 可以知道，在区域二以内目标点到线段的最短距离，就是过目

标点做一条垂直于线段的直线，这是它的最短距离 d=$\|\overrightarrow{qp} = \overrightarrow{op} - \overrightarrow{oq} = \overrightarrow{op} - r^{17}\overrightarrow{ac}\|$。但是我们现在实际并不是一条线段，而是被中点 C 分成了两条线，因此我们上面的 r 部分取值将不再是 0 到 1，而将变为-1 到 1。

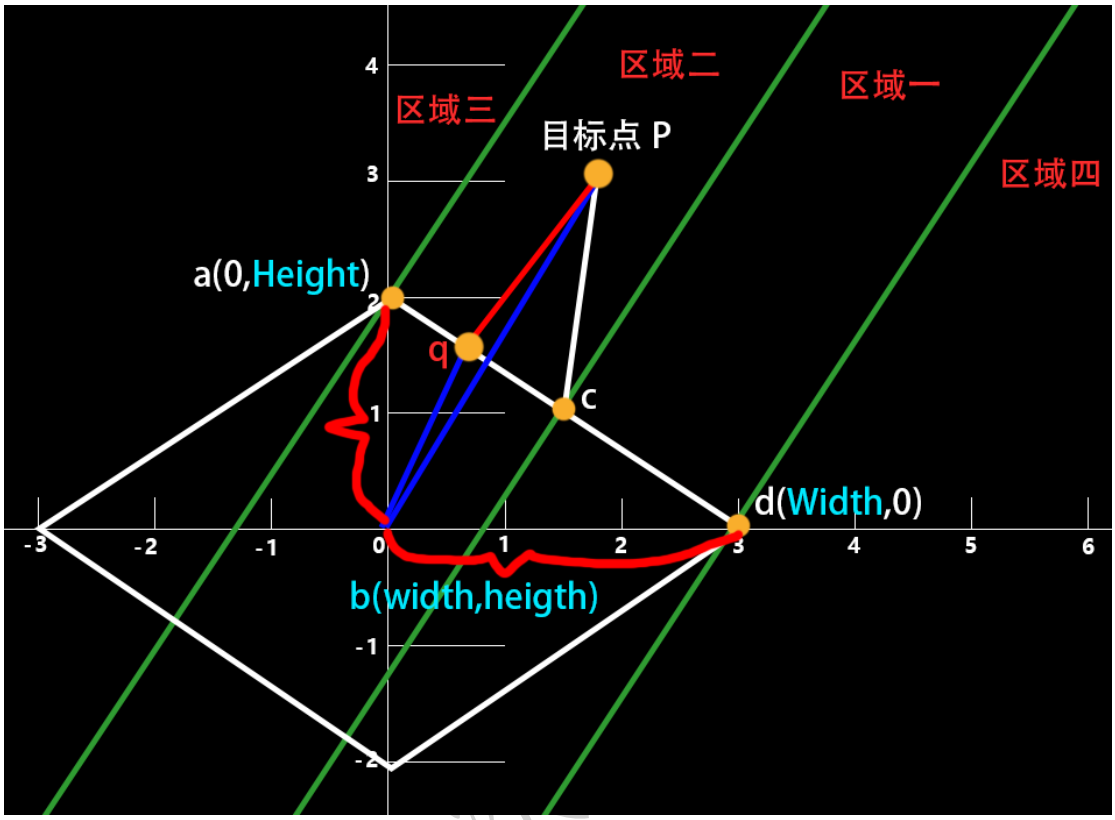


图112.　菱形区域划分

我们通过上面的图像可以知道，中点 C 将曾经的线段 ad 分割为了两部分，也就是线段 ac 和 cd。我们可以用一个公式来表示点 q 位于 ad 上的所有情况，

$$\begin{cases} X = 0.5b.x - 0.5b.x * h \\ Y = 0.5b.y + 0.5b.y * h \end{cases}$$。

当 h=-1 时代表的就是点 d(b.x,0)，当 h=1 时代表的就是点 a(0,b.y) ，也就是当 h 从-1 变化到 1，q 点就从 d 点滑动到 a 点。（如果将上式的+号和-减互换，则是从 a 滑向 d）

所以$\overrightarrow{oq} = 0.5 * b * vec2(1.0 - h, 1.0 + h)$（这里使用了矩阵乘法）

d=$\|\overrightarrow{qp}\| = length(\overrightarrow{op} - 0.5 * b * vec2(1.0 - h, 1.0 + h))$（这里是向量的减法运算）

当我们得到了向量$\overrightarrow{qp}$的长度后，就需要判断距离的符号了，这里使用向量的叉乘。

$$\overrightarrow{cp} \times \overrightarrow{ca} = \frac{1}{2}(p.x * b.y + p.y * b.x - b.x * b.y)$$

该向量积的计算过程如下：

PS：

[17] 这里的 r 是单词 Ratio 的缩写

$$\begin{vmatrix} i & j & k \\ p.x-\frac{1}{2}width & p.y-\frac{1}{2}height & 0 \\ -\frac{1}{2}width & \frac{1}{2}height & 0 \end{vmatrix}$$

$$= 0 + 0 + \left(\frac{1}{2}height * p.x - \frac{1}{4} * height * width\right)\mathrm{k}$$

$$-\left(-\frac{1}{2}width * p.y + \frac{1}{4} * height * width\right)\mathrm{k} - 0 - 0$$

$$= \frac{1}{2}height * p.x + \frac{1}{2}width * p.y - \frac{1}{2}width * \frac{1}{2}height$$

$$= c.y * p.x + c.x * p.y - c.x * c.y = \frac{1}{2}(p.x * b.y + p.y * b.x - b.x * b.y)$$

当上述叉乘结果大于 0，则 p 点在向量 ca 右侧（即在菱形外部），距离符号为正；上述叉乘结果小于 0，则 p 点在向量 ca 左侧（即在菱形内部），距离符号为负。由于判断正负与具体值无关，所以代码里可以省略$\frac{1}{2}$这个无用的计算。

2） Shdertoy 代码

```
float ndot(vec2 a, vec2 b )
{
    return a.x * b.x - a.y * b.y;
}

float sdRhombus( in vec2 p, in vec2 b )
{
    p = abs(p);

    float h = clamp( (-2.0*ndot(p,b) + ndot(b,b) )/dot(b,b), -1.0, 1.0 );
    float d = length( p - 0.5*b*vec2(1.0-h,1.0+h) );
    d *= sign( p.x*b.y + p.y*b.x - b.x*b.y );

    return d;
}
```

6. 梯形

7. 月食

首先我们需要分析月食效果是如何产生的，在太阳光线到达月亮之前，如果被地球挡住了，那这部分光线就不会到达月球，以至于出现月食。因此，我们可以用一张白色的大圆形蒙版和一张黑色的小圆形蒙版，去模拟这种效果。但如果我们这样做，就需要两张蒙版的图像，为此我们可以考虑直接生成一张完整的月食蒙版。

月食的 SDF 函数需要 4 个参数，目标点的坐标(targetPoint)、圆 A 和 B 的半径，以及两个圆心之间的距离(centerDistance)。当我们获取到目标点的坐标后，需要将它的 Y 坐标取绝对值，这是因为这里的月食效果是关于 X 轴对称的。

哔哩哔哩次元游戏人生

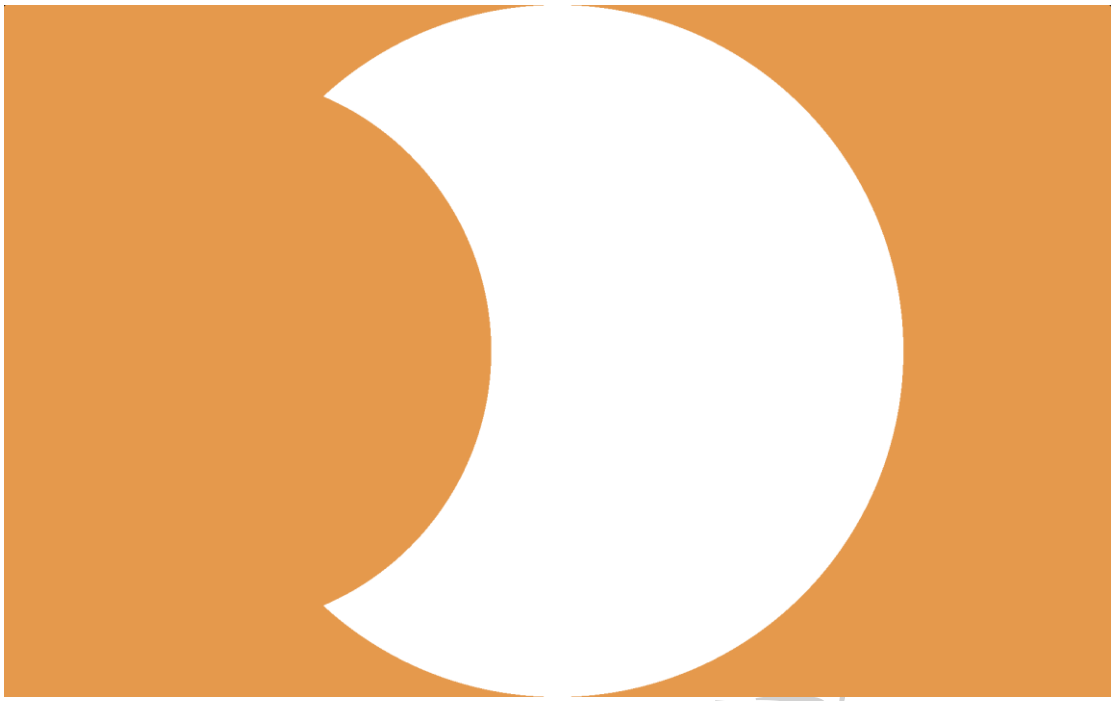


图113.　月食效果，关于大圆圆心 Y 轴对称

我们知道，当两个圆心之间的间距为 0 时，它们所呈现的图像就是一个圆环。我们可以将两个圆的面积进行相减，进而得到圆环的面积，$\pi r_a^2 - \pi r_b^2 = 余下部分的面积$。

3） Shdertoy 代码

```
float sdMoon(vec2 p, float d, float ra, float rb )
{
    p.y = abs(p.y);
    float a = (ra*ra - rb*rb + d*d)/(2.0*d);
    float b = sqrt(max(ra*ra-a*a,0.0));

    这部分用于精确距离场
    if( d*(p.x*b-p.y*a) > d*d*max(b-p.y,0.0) )
    {
        return length(p-vec2(a,b));
    }

    return Sign(max( (length(p)-ra),-(length(p-vec2(d,0))-rb)));
}
```

4） Unity Unlit Shader

```
Shader "Unlit/SDF-Moon"
{
    Properties
    {
        [MainTexture] _BaseMap("Texture", 2D) = "white" {}
        _BaseColor("Base Color",Color)=(1,1,1,1)
        _CircleARadius("圆 A 的半径",Range(0.5,1.0)) = 0.5
        _CircleBRadius("圆 B 的半径",Range(0.1,0.5)) = 0.3
```

```
        _CenterDistacne("圆心间距",Range(0,0.6)) = 0.6
    }
    SubShader
    {
        Tags
        {
            "RenderType"="Transparent"
            "Queue"="Transparent"
            "RenderPipeline"="UniversalRenderPipeline"//告诉 GPU，我们使用的是通用渲染管线
        }
        LOD 100

        //自动将代码包含在 Subshader 内部的每个 Pass 中。
        HLSLINCLUDE

            //为着色器编译添加辅助宏和跨平台定义
            #include "HLSLSupport.cginc"
            //用 Core.hlsl 取代了 UnityCG.cginc
            #include "E:/unity/Bilibili2DProject/Library/PackageCache/com.unity.render-pipelines.universal@12.1.2/ShaderLibrary/Core.hlsl"

            /*使用 UnityPerMaterial CBUFFER 来确保着色器兼容 SRP 批处理，CBUFFER 需要包括所有公开的属性。
            但它不能包括其他未公开的变量，纹理也不需要被包括。*/
            CBUFFER_START(UnityPerMaterial)
                float4 _BaseMap_ST;
                float4 _BaseMap_TexelSize;
                float4 _BaseColor;
                float _CircleARadius;
                float _CircleBRadius;
                float _CenterDistacne;
            CBUFFER_END

            TEXTURE2D(_BaseMap);
            SAMPLER(sampler_BaseMap);

            struct VertexInput
            {
                float4 vertex : POSITION;//顶点的坐标信息
                float2 uv : TEXCOORD0;
            };

            struct VertexOutput
```

哔哩哔哩次元游戏人生

```
        {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;//裁剪空间中的顶点坐标
        };
    ENDHLSL

    Pass
    {
        HLSLPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        //d 为两圆圆心相距的距离
        float sdfMoon(float2 p, float dis, float ra, float rb )
        {
            p.y = abs(p.y);

            float a = (ra * ra - rb * rb + dis * dis)/(2.0 * dis);
            float b = sqrt(max(ra * ra - a * a,0.0));//开方
            if( dis * (p.x * b - p.y * a) > dis * dis * max(b - p.y,0.0) )
            {
                return length(p - float2(a,b));
            }
            return max((length(p) - ra),-(length(p - float2(dis,0)) - rb));
        }

        VertexOutput vert (VertexInput v)
        {
            VertexOutput o;

            //使用函数 TransformObjectToHClip 替换 UnityObjectToClipPos，就是将顶点的位置从对象空间转换到裁剪空间，前者在执行过程中效率更高
            o.vertex = TransformObjectToHClip(v.vertex);

            o.uv = TRANSFORM_TEX(v.uv, _BaseMap);
            return o;
        }

        float4 frag (VertexOutput i) : SV_Target
        {
            float2 targetPoint = i.uv - 0.5;
            targetPoint *= 1;
            _CenterDistacne *= _CosTime.w;//_CosTime.x = t / 8; _CosTime.y = t / 4; _CosTime.z = t / 3; _CosTime.w = t;
```

哔哩哔哩次元游戏人生

```
                float d =
sdfMoon(targetPoint,_CenterDistacne,_CircleARadius,_CircleBRadius);

                float3 col = 1 - sign(d);
                float4 baseTex= SAMPLE_TEXTURE2D(_BaseMap ,
sampler_BaseMap, i.uv);

                return baseTex * _BaseColor *    float4 (col.xxx,1);
            }
            ENDHLSL
        }
    }
}
```

二十、 伪随机噪声-Pseudo Random Noise

1. Hashing

1) 可视化-Visualization

随机性可以使事物不可预测，更多变和更自然。感知到的现象不管是真正的随机，还是由于缺乏信息或者观察者对其的理解，而以某种形式出现，这并不重要。因此，我们可以做一些完全确定的东西，而不是随机的，只要这不明显。这种决定很好，因为软件本质上(inherently[18])是确定性的。设计不佳的多线程代码可能会导致争用现象，从而导致不可预知的结果，但这不是随机性的可靠来源。真正可靠的随机性只能从外部获得，例如对大气噪声进行采样的硬件，而这些硬件通常不可用。

通常情况下，是不需要真正的随机。因为，由它生成的任何内容都是无法重现的一次性事件，每次结果都会有所不同。理想情况下，我们有一个过程，对于任何特定的(specific[19])输入，都会产生唯一且固定的明显随机输出。这就是哈希函数的用途。

哔哩哔哩次元游戏人生

[18] adv. 内在地，固有地；本质的；
[19] adj. 明确的，具体的；特定的；特有的，独特的；有特效的； n. 细节，详情；特效药

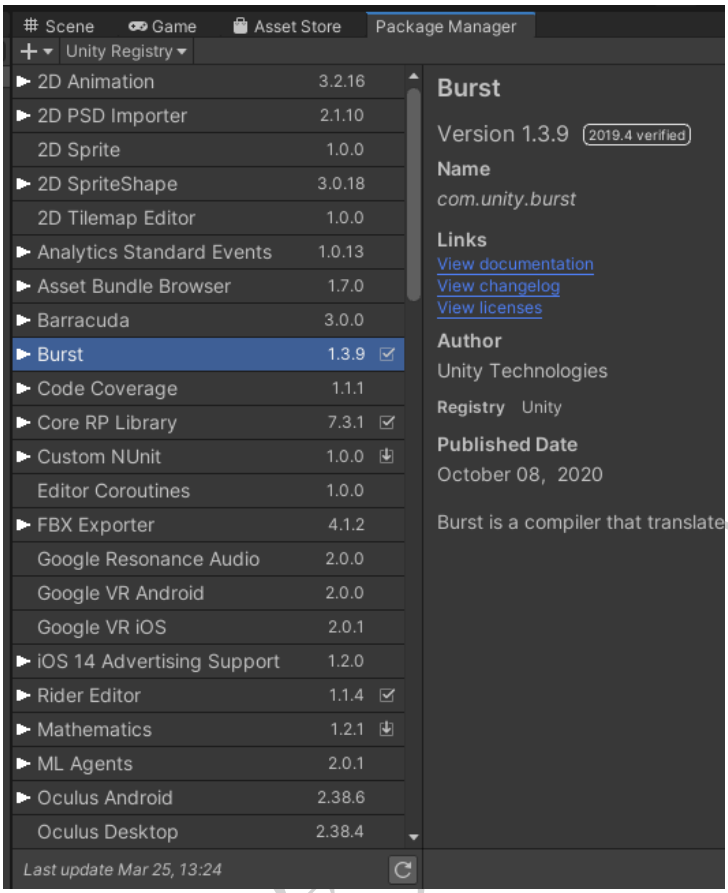


图1. Burst Package

下面我们将创建一个由小立方体组成的 2D 网格，并使用它来可视化哈希函数。为了实现它需要导入 Burst 包（Burst 包请在 PackageManager 里下载），还将使用 Unity 的 URP(Universal Render Pipeline)渲染管线，因此请使用通用渲染管线并为其创建资源。

2） 哈希作业

我们将为网格中的所有多维数据集创建哈希值。创建一个 HashVisualization 脚本，使用哈希去填充 NativeArray，哈希值本质上(essentially[20])是没有固有含义的位的集合。我们将为它们赋予一个 uint 类型，它最接近于通用数据的 32 位（4 个字节）数据包。最初，我们将直接使用 Job 的执行索引作为哈希值。

PS：

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine;

using static Unity.Mathematics.math;

public class HashVisualization : MonoBehaviour {
```

[20] adv. 本质上，根本上；大体上，基本上

```
        [BurstCompile(FloatPrecision.Standard, FloatMode.Fast, CompileSynchronously =
true)]
        struct HashJob : IJobFor {

            [WriteOnly]
            public NativeArray<uint> hashes;

            public void Execute(int i) {
                hashes[i] = (uint)i;
            }
        }
    }
```

a. 为什么不使用 int 作为 hash type

int 类型表示一个有符号整数，它有一个特殊的符号位。uint 类型是无符号的，因此没有特殊的符号位，uint 值的所有位都以相同的方式处理。

3) 初始化和呈现-Initialization and Rendering

4) 着色-Shader

5) 图案-Patterns

https://catlikecoding.com/unity/tutorials/pseudorandom-noise/hashing/

二十一、Unity 中的渲染优化技术

程序优化的第一条准则：不要优化。程序优化的第二条准则（仅针对专家！）：不要优化。——Michael A.Jackson

在进行程序优化的时候，人们经常会引用英国的计算机科学家 Michael A.Jackson 在 1988 年的优化准则。Jackson 是想借此强调，对问题认识不清以及过度优化往往会让事情变得更加复杂，产生更多的程序错误。

然而，如果我们在游戏开发过程中从来都没有考虑优化，那么结果往往是惨不忍睹的。一个正确的做法是，从一开始就把优化当成是游戏设计中的一部分。正在阅读本书的读者，有可能是移动游戏的开发者。和 PC 相比，移动设备上的 GPU 有着完全不同的架构设计，它能使用的带宽、功能和其他资源都非常有限。这要求我们需要时刻把优化谨记在心，才可以避免等到项目完成时才发现游戏根本无法在移动设备上流畅运行的结果。

下面我们将会阐述一些 Unity 中常见的优化技术。这些优化技术都是和渲染相关的，例如，使用批处理、LOD（Level of Detail）技术等。

游戏优化不仅是程序员的工作，更需要美工人员在游戏的美术上进行一定的权衡，例如，避免使用全屏的屏幕特效，避免使用计算复杂的 shader，减少透明混合造成的 overdraw 等。也就是说，这是由程序员和美工人员等各个部分人员共同参与的工作。

1. 移动平台的特点

和 PC 平台相比，移动平台上的 GPU 架构有很大的不同。由于处理资源等条件的限制，移动设备上的 GPU 架构专注于尽可能使用更小的带宽和功能，也由此带来了许多和 PC 平台完全不同的现象。

例如，为了尽可能移除那些隐藏的表面，减少 overdraw（即一个像素被绘制多次），

降龙降龙元游戏人生

PowerVR 芯片（通常用于 iOS 设备和某些 Android 设备）使用了基于瓦片的延迟渲染（Tiled-based DeferredRendering，TBDR）架构，把所有的渲染图像装入一个个瓦片（tile）中，再由硬件找到可见的片元，而只有这些可见片元才会执行片元着色器。另一些基于瓦片的 GPU 架构，如 Adreno（高通的芯片）和 Mali（ARM 的芯片）则会使用 Early-Z 或相似的技术进行一个低精度的的深度检测，来剔除那些不需要渲染的片元。还有一些 GPU，如 Tegra（英伟达的芯片），则使用了传统的架构设计，因此在这些设备上，overdraw 更可能造成性能的瓶颈。

由于这些芯片架构造成的不同，一些游戏往往需要针对不同的芯片发布不同的版本，以便对每个芯片进行更有针对性的优化。尤其是在 Android 平台上，不同设备使用的硬件，如图形芯片、屏幕分辨率等，大相径庭，这对图形优化提出了更高的挑战。相比与 Android 平台，iOS 平台的硬件条件则相对统一。

2. 影响性能的因素

首先，在学习如何优化之前，我们得了解影响游戏性能的因素有哪些，才能对症下药。对于一个游戏来说，它主要需要使用两种计算资源，即 CPU 和 GPU。它们会互相合作，来让我们的游戏可以在预期的帧率和分辨率下工作。其中，CPU 主要负责保证帧率，GPU 主要负责分辨率相关的一些处理。

据此，我们可以把造成游戏性能瓶颈的主要原因分成以下几个方面。

1） CPU

① 过多的 Draw Call

② 复杂的脚本或者物理模拟

2） GPU

a. 顶点处理

① 过多的顶点

② 过多的逐顶点计算

b. 片元处理

① 过多的片元（即可能是由于分辨率造成的，也可能是由于 overdraw 造成的）

② 过多的逐片元计算

c. 带宽

① 使用了尺寸很大且未压缩的纹理

② 分辨率过高的帧缓存

对于 CPU 来说，限制它的主要是每一帧中 draw call 的数目。简单来说，就是 CPU 在每次通知 GPU 进行渲染之前，都需要提前准备好顶点数据（如位置、法线、颜色、纹理坐标等），然后调用一系列 API 把它们放到 GPU 可以访问到的指定位置，最后，调用一个绘制命令，来告诉 GPU，“嘿，我把东西都准备好了，你赶紧出来干活（渲染）吧！”。而调用绘制命令的时候，就会产生一个 draw call。过多的 draw call 会造成 CPU 的性能瓶颈，这是因为每次调用 draw call 时，CPU 往往都需要改变很多渲染状态的设置，而这些操作是

哔哩哔哩次元游戏人生

非常耗时的。如果一帧中需要的 draw call 数目过多的话，就会导致 CPU 把大部分时间都花费在提交 draw call 的工作上面了。当然，其他原因也可能造成 CPU 瓶颈，例如物理、布料模拟、蒙皮、粒子模拟等，这些都是计算量很大的操作。

而对于 GPU 来说，它负责整个渲染流水线。它从处理 CPU 传递过来的模型数据开始，进行顶点着色器、片元着色器等一系列工作，最后输出屏幕上的每个像素。因此，GPU 的性能瓶颈和需要处理的顶点数目、屏幕分辨率、显存等因素有关。而相关的优化策略可以从减少处理的数据规模（包括顶点数目和片元数目）、减少运算复杂度等方面入手。

1） CPU 优化

① 使用批处理技术减少 Draw Call 数目

2） GPU 优化

a. 减少需要处理的片元数目

① 优化几何体
② 使用模型的 LOD(Level of Detail)技术
③ 使用遮挡剔除(Occlusion Culling)技术

b. 减少计算复杂度

① 使用 Shader 的 LOD(Level of Detail)技术
② 代码方面的优化

3） 节省内存带宽

① 减少纹理大小
② 利用分辨率缩放

3. 减少 DrawCall 数目

我们平时最常看到的优化技术就是批处理(batching)了。批处理的实现原理就是为了减少每一帧需要的 Draw Call 数目。为了把一个对象渲染到屏幕上，CPU 需要检查哪些光源影响了该物体，绑定 Shader 并设置它的参数，再把渲染命令发送给 GPU。当场景中包含了大量对象时，这些操作就会非常耗时。

一个极端的例子是，如果我们需要渲染一千个三角形，把它们按一千个单独的网格进行渲染所花费的时间要远远大于渲染一个包含了一千个三角形的网格。在这两种情况下，GPU 的性能消耗其实并没有多大的区别，但 CPU 的 Draw Call 数目就会成为性能瓶颈。因此，批处理的思想很简单，就是在每次调用 Draw Call 时尽可能多地处理多个物体。

那么，什么样的物体可以一起处理呢？答案就是使用同一个材质的物体。这是因为，对于使用同一个材质的物体，它们之间的不同仅仅在于顶点数据的差别。我们可以把这些顶点数据合并在一起，再一起发送给 GPU，就可以完成一次批处理。

Unity 中支持两种批处理方式：一种是动态批处理，另一种是静态批处理。对于动态批处理来说，优点是一切处理都是 Unity 自动完成的，不需要我们自己做任何操作，而且物体是可以移动的，但缺点是，限制很多，可能一不小心就会破坏了这种机制，导致 Unity 无法动态批处理一些使用了相同材质的物体。而对于静态批处理来说，它的优点是自由度很高，限制很少；但缺点是可能会占用更多的内存，而且经过静态批处理后的所有物体都不可以再移动了（即便在脚本中尝试改变物体的位置也是无效的）。

唯理逻辑次元游戏人生

1） 动态批处理

如果场景中有一些模型共享了同一个材质并满足一些条件，Unity 就可以自动把它们进行批处理，从而只需要花费一个 draw call 就可以渲染所有的模型。动态批处理的基本原理是，每一帧把可以进行批处理的模型网格进行合并，再把合并后模型数据传递给 GPU，然后使用同一个材质对其渲染。除了实现方便，动态批处理的另一个好处是，经过批处理的物体仍然可以移动，这是由于在处理每帧时 Unity 都会重新合并一次网格。

虽然 Unity 的动态批处理不需要我们进行任何额外工作，但只有满足条件的模型和材质才可以被动态批处理。需要注意的是，随着 Unity 版本的变化，这些条件也有一些改变。

能够进行动态批处理的网格的顶点属性规模要小于 900。例如，如果 shader 中需要使用顶点位置、法线和纹理坐标这 3 个顶点属性，那么要想让模型能够被动态批处理，它的顶点数目不能超过 300。需要注意的是，这个数字在未来有可能会发生变化，因此不要依赖这个数据。

一般来说，所有对象都需要使用同一个缩放尺度（可以是（1，1，1）、（1，2，3）、（1.5，1.4，1.3）等，但必须都一样）。一个例外情况是，如果所有的物体都使用了不同的非统一缩放，那么它们也是可以被动态批处理的。但在 Unity5 中，这种对模型缩放的限制已经不存在了。

使用光照纹理（（lightmap）的物体需要小心处理。这些物体需要额外的渲染参数，例如，在光照纹理上的索引、偏移量和缩放信息等。因此，为了让这些物体可以被动态批处理，我们需要保证它们指向光照纹理中的同一个位置。

多 Pass 的 shader 会中断批处理。在前向渲染中，我们有时需要使用额外的 Pass 来为模型添加更多的光照效果，但这样一来模型就不会被动态批处理了。

一个场景中包含了 3 个立方体，它们使用同一个材质，同时还包含了一个使用其他材质的球体。场景中还包含了一个平行光，但我们关闭了它的阴影效果，以避免阴影计算对批处理数目的影响。这样一个场景的渲染统计数据如下图所示。

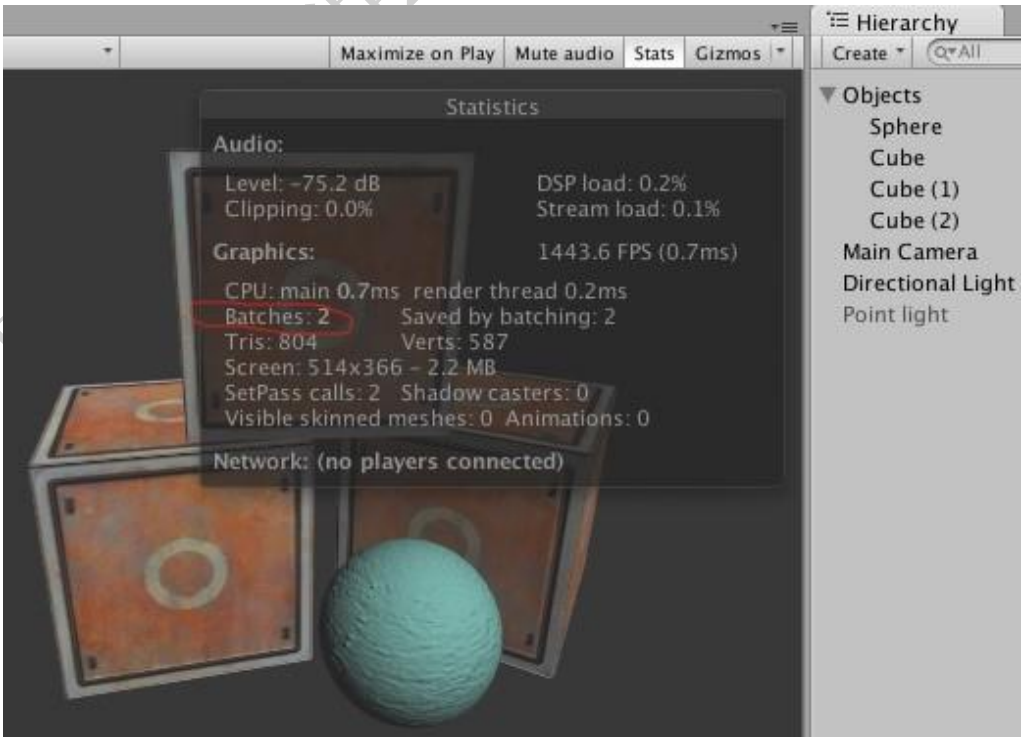


图2. 动态批处理

从上图中可以看出，要渲染这样一个包含了 4 个物体的场景共需要两个批处理。其中，一个批处理用于绘制经过动态批处理合并后的 3 个立方体网格，另一个批处理用于绘制球体。我们可以从 Save by batching 看出批处理帮我们节省了两个 draw call。

现在，我们再向场景中添加一个点光源，并调整它的位置使它可以照亮场景中的 4 个物体。由于场景中的物体都使用了多个 Pass 的 shader，因此，点光源会对它们产生光照影响。

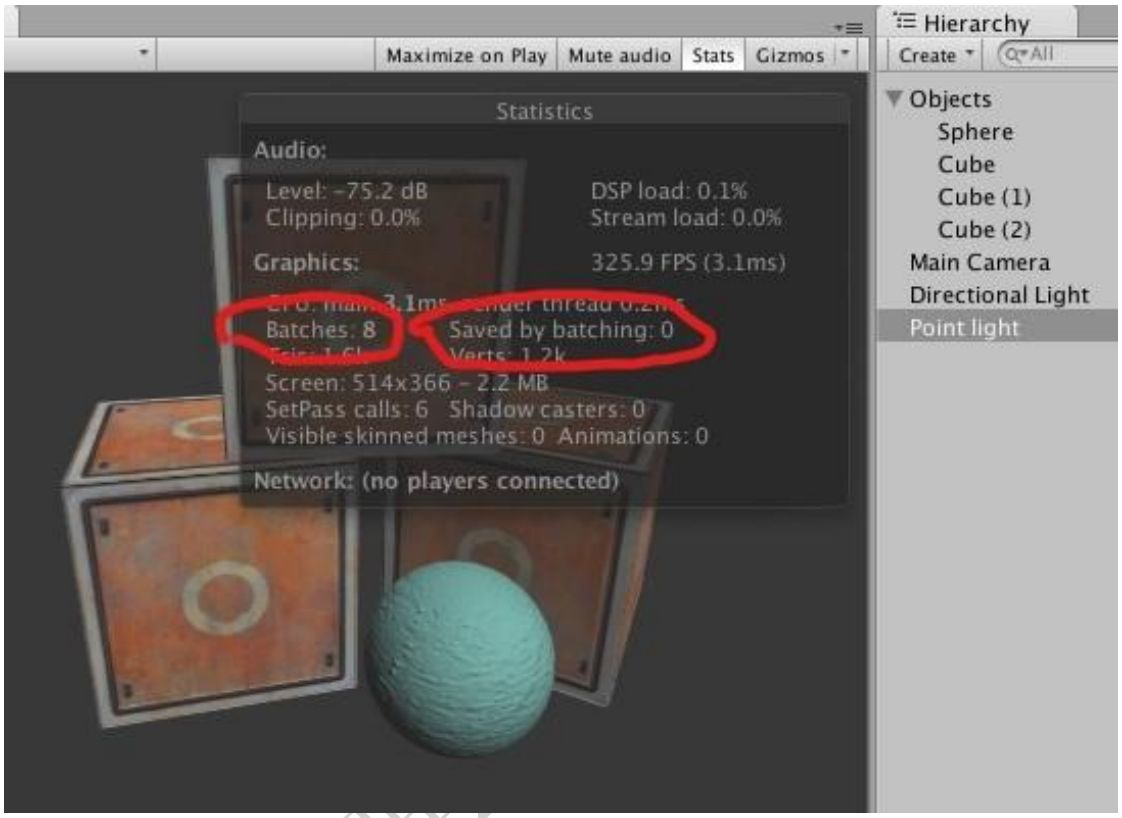


图3. 多光源对动态批处理的影响结果

从上图中可以看出，渲染一帧所需的批处理数目增大到了 8，而 Save by batching 的数目也变成了 0。这是因为，使用了多个 Pss 的 shader 在需要应用多个光照的情况下，破坏了动态批处理的机制，导致 Unity 不能对这些物体进行动态批处理。而由于平行光和点光源需要对 4 个物体分别产生影响，因此，需要 2×4 个批处理操作。需要注意的是，只有物体在点光源的影响范围内，Unity 才会调用额外的 Pss 来处理它。因此，如果场景中点光源距离物体很远，那么它们仍然会被动态批处理的。

动态批处理的限制条件比较多，例如很多时候，我们的模型数据往往会超过 900 的顶点属性限制。这种时候依赖动态批处理来减少 draw call 显然已经不能够满足我们的需求了。这时，我们可以使用 Unity 的静态批处理技术。

2） 静态批处理

Unity 提供了另一种批处理方式，即静态批处理。相比于动态批处理来说，静态批处理适用于任何大小的几何模型。它的实现原理是，只在运行开始阶段，把需要进行静态批处理的模型合并到一个新的网格结构中，这意味着这些模型不可以在运行时刻被移动。但由于它只需要进行一次合并操作，因此，比动态批处理更加高效。

静态批处理的另一个缺点在于，它往往需要占用更多的内存来存储合并后的几何结构。这是因为，如果在静态批处理前一些物体共享了相同的网格，那么在内存中每一个物体都会对应一个该网格的复制品，即一个网格会变成多个网格再发送给 GPU。

如果这类使用同一网格的对象很多，那么这就会成为一个性能瓶颈了。例如，如果在一个使用了 1000 个相同树模型的森林中使用静态批处理，那么，就会多使用 1000 倍的内存，这会造成严重的内存影响。这种时候，解决方法要么忍受这种牺牲内存换取性能的方法，要么不要使用静态批处理，而使用动态批处理技术（但要小心控制模型的顶点属性数目），或者自己编写批处理的方法。

一个场景中包含了 3 个茶壶(Teapot[21])模型，它们使用同一个材质，同时还包含了一个使用不同材质的立方体。场景中还包含了一个平行光，但我们关闭了它的阴影效果，以避免阴影计算对批处理数目的影响。在运行前，这样一个场景的渲染统计数据如下图所示。

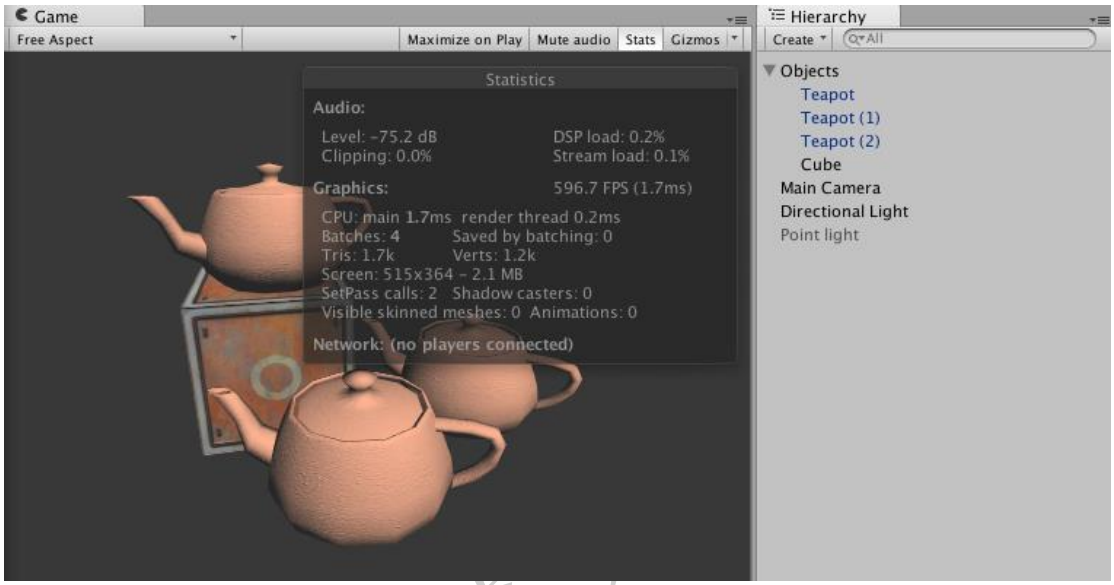


图4. 静态批处理前的渲染统计数据

从图中可以看出，尽管 3 个 Teapot 模型使用了相同的材质，但它们仍然没有被动态批处理。这是因为，Teapot 模型包含的顶点数目是 393，而它们使用的 shader 中需要使用 4 个顶点属性（顶点位置、法线方向、切线方向和纹理坐标），超过了动态批处理中限定的 900 限制。此时，要想减少 draw call 就需要使用静态批处理。

静态批处理的实现非常简单，只需要把 Inspector 面板上把 Static 复选框勾选上即可（实际上我们只需要勾选 Batching Static 即可）。

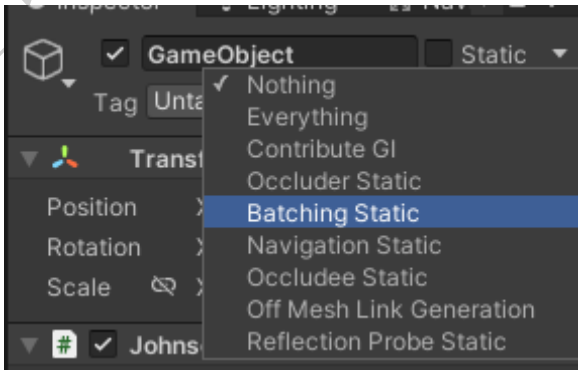


图5. 把对象标为 Static

这时，我们再观察渲染统计窗口中的批处理数目，还是没有变化。但是不要急，运行程序后，变化就出现了。

[21] n. 茶壶

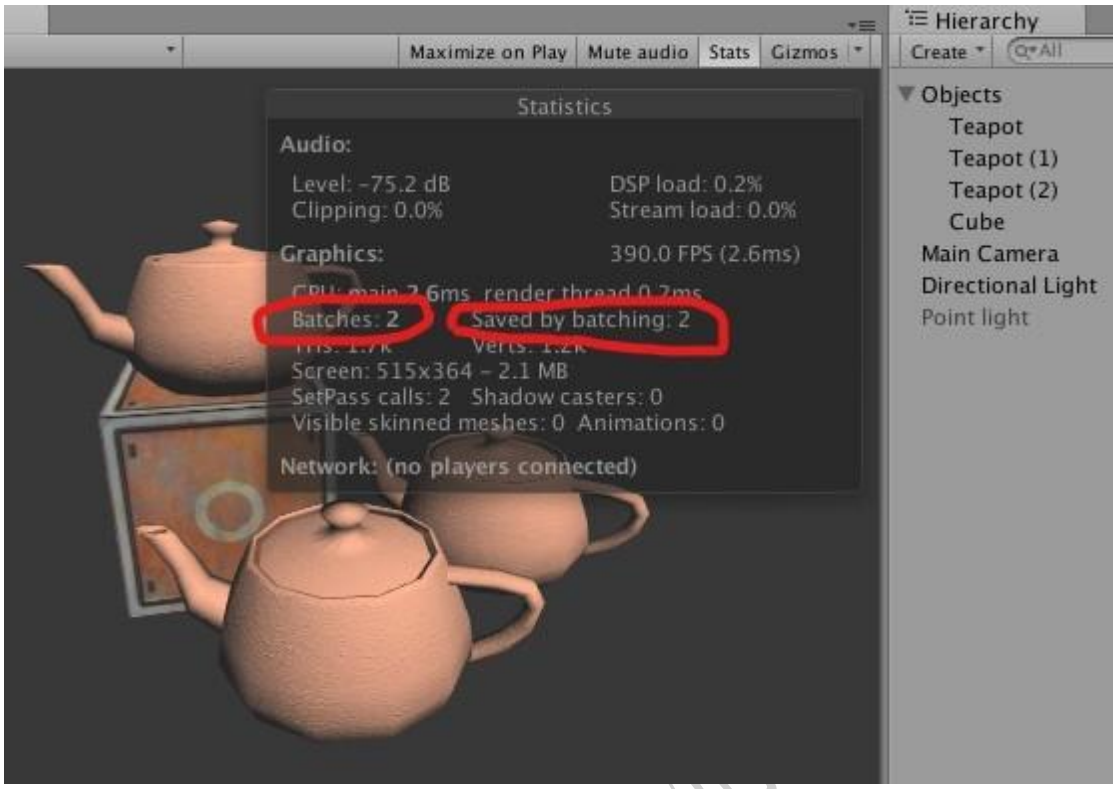


图6. 静态批处理

3） 共享材质

从之前的内容可以看出，无论是动态批处理还是静态批处理，都要求模型之间需要共享同一个材质。但不同的模型之间总会需要有不同的渲染属性，例如，使用不同的纹理、颜色等。这时，我们需要一些策略来尽可能地合并材质。

如果两个材质之间只有使用的纹理不同，我们可以把这些纹理合并到一张更大的纹理中，这张更大的纹理被称为是一张图集（atlas）。一旦使用了同一张纹理，我们就可以使用同一个材质，再使用不同的采样坐标对纹理采样即可。

但有时，除了纹理不同外，不同的物体在材质上还有一些微小的参数变化，例如，颜色不同、某些浮点属性不同。但是，不管是动态批处理还是静态批处理，它们的前提都是要使用同一个材质。是同一个，而不是使用了同一种 Shader 的材质，也就是说它们指向的材质必须是同一个实体。这意味着，只要我们调整了参数，就会影响到所有使用这个材质的对象。那么想要微小的调整怎么办呢？一种常用的方法就是使用网格的顶点数据（最常见的就是顶点颜色数据）来存储这些参数。

前面说过，经过批处理后的物体会被处理成更大的 VBO(Vertex Buffer Object)发送给 GPU，VBO 中的数据可以作为输入传递给顶点着色器，因此，我们可以巧妙地对 VBO 中的数据进行控制，从而达到不同效果的目的。一个例子是，森林场景中所有的树使用了同一种材质，我们希望它们可以通过批处理来减少 draw call，但不同树的颜色可能不同。这时，我们可以利用网格的顶点的颜色数据来调整。

需要注意的是，如果我们需要在脚本中访问共享材质，应该使用 Renderer.sharedMaterial 来保证修改的是和其他物体共享的材质，但这意味着修改会应用到所有使用该材质的物体上。另一个类似的 API 是 Renderer.material，如果使用 Renderer.material 来修改材质，Unity 会创建一个该材质的复制品，从而破坏批处理在该物体上的应用，这可能并不是我们希望看到的。

4） 批处理的注意事项

在选择使用动态批处理还是静态批处理时，我们有一些小小的建议。

① 尽可能选择静态批处理，但得时刻小心对内存的消耗，并且记住经过静态批处理的物体不可以再被移动。

② 如果无法进行静态批处理，而要使用动态批处理的话，那么请小心上面提到的各种条件限制。例如，尽可能让这样的物体少并且尽可能让这些物体包含少量的顶点属性和顶点数目。

③ 对于游戏中的小道具，例如可以捡拾的金币等，可以使用动态批处理。

④ 对于包含动画的这类物体，我们无法全部使用静态批处理，但其中如果有不动的部分，可以把这部分标识成“Static”。

除了上述提示外，在使用批处理时还有一些需要注意的地方。由于批处理需要把多个模型变换到世界空间下再合并它们，因此，如果 shader 中存在一些基于模型空间下的坐标的运算，那么往往会得到错误的结果。一个解决方法是，在 shader 中使用 DisableBatching 标签来强制使用该 Shader 的材质不会被批处理。另一个注意事项是，使用半透明材质的物体通常需要使用严格的从后往前的绘制顺序来保证透明混合的正确性。对于这些物体，Unity 会首先保证它们的绘制顺序，再尝试对它们进行批处理。这意味着，当绘制顺序无法满足时，批处理无法在这些物体上被成功应用。

4. 减少需要处理的顶点数目

尽管 draw call 是一个重要的性能指标，但顶点数目同样有可能成为 GPU 的性能瓶颈。

1） 优化几何体

3D 游戏制作通常都是由模型制作开始的。而在建模时，有一条规则我们需要记住，即尽可能减少模型中三角面片的数目，一些对于模型没有影响、或是肉眼非常难察觉到区别的顶点都要尽可能去掉。为了尽可能减少模型中的顶点数目，美工人员往往需要优化网格结构。在很多三维建模软件中，都有相应的优化选项，可以自动优化网格结构。

在 Unity 的渲染统计窗口中，我们可以查看到渲染当前帧需要的三角面片数目和顶点数目。需要注意的是，Unity 中显示的数目往往要多于建模软件里显示的顶点数，通常 Unity 中显示的数目要大很多。谁才是对的呢？其实，这是因为在不同的角度上计算的，都有各自的道理，但我们真正应该关心的是 Unity 里显示的数目。

我们在这里简单解释一下造成这种不同的原因。三维软件更多地是站在我们人类的角度理解顶点的，即组成几何体的每一个点就是一个单独的点。而 Unity 是站在 GPU 的角度上去计算顶点数的。在 GPU 看来，有时需要把一个顶点拆分成两个或更多的顶点。这种将顶点一分为多的原因主要有两个：一个是为了分离纹理坐标（uv splits），另一个是为了产生平滑的边界（smoothingsplits）。

它们的本质，其实都是因为对于 GPU 来说，顶点的每一个属性和顶点之间必须是一对一的关系。而分离纹理坐标，是因为建模时一个顶点的纹理坐标有多个。例如，对于一个立方体，它的 6 个面之间虽然使用了一些相同的顶点，但在不同面上，同一个顶点的纹理坐标可能并不相同。对于 GPU 来说，这是不可理解的，因此，它必须把这个顶点拆分成多个具有不同纹理坐标的顶点。而平滑边界也是类似的，不同的是，此时一个顶点可能会对应多个法线信息或切线信息。这通常是因为我们要决定一个边是一条硬边（hard edge）还是一条平滑边（smooth edge）。

对于 GPU 来说，它本质上只关心有多少个顶点。因此，尽可能减少顶点的数目其实才是我们真正需要关心的事情。因此，最后一条几何体优化建议就是：移除不必要的硬边以

腾讯游戏学堂

及纹理衔接，避免边界平滑和纹理分离。

2） 模型的 LOD 技术

另一个减少顶点数目的方法是使用 LOD（Level of Detail）技术。这种技术的原理是，当一个物体离摄像机很远时，模型上的很多细节是无法被察觉到的。因此，LOD 允许当对象逐渐远离摄像机时，减少模型上的面片数量，从而提高性能。

在 Unity 中，我们可以使用 LOD Group 组件来为一个物体构建一个 LOD。我们需要为同一个对象准备多个包含不同细节程度的模型，然后把它们赋给 LOD Group 组件中的不同等级，Unity 就会自动判断当前位置上需要使用哪个等级的模型。

3） 遮挡剔除技术

我们最后要介绍的顶点优化策略就是遮挡剔除（Occlusion culling）技术。遮挡剔除可以用来消除那些在其他物件后面看不到的物件，这意味着资源不会浪费在计算那些看不到的顶点上，进而提升性能。

我们需要把遮挡剔除和摄像机的视锥体剔除（Frustum Culling）区分开来。视锥体剔除只会剔除掉那些不在摄像机的视野范围内的对象，但不会判断视野中是否有物体被其他物体挡住。而遮挡剔除会使用一个虚拟的摄像机来遍历场景，从而构建一个潜在可见的对象集合的层级结构。在运行时刻，每个摄像机将会使用这个数据来识别哪些物体是可见的，而哪些被其他物体挡住不可见。使用遮挡剔除技术，不仅可以减少处理的顶点数目，还可以减少 overdraw，提高游戏性能。

模型的 LOD 技术和遮挡剔除技术可以同时减少 CPU 和 GPU 的负荷。CPU 可以提交更少的 draw call，而 GPU 需要处理的顶点和片元数目也减少了。

5. 减少需要处理的片元数目

另一个造成 GPU 瓶颈的是需要处理过多的片元。这部分优化的重点在于减少 overdraw。简单来说，overdraw 指的就是同一个像素被绘制了多次。

Unity 还提供了查看 overdraw 的视图，我们可以在 Scene 视图左上方的下拉菜单中选中 Overdraw 即可。实际上，这里的视图只是提供了查看物体相互遮挡的层数，并不是真正的最终屏幕绘制的 overdraw。也就是说，可以理解为它显示的是，如果没有使用任何深度测试和其他优化策略时的 overdraw。这种视图是通过把所有对象都渲染成一个透明的轮廓，通过查看透明颜色的累计程度，来判断物体之间的遮挡。当然，我们可以使用一些措施来防止这种最坏情况的出现。

1） 控制绘制顺序

为了最大限度地避免 overdraw，一个重要的优化策略就是控制绘制顺序。由于深度测试的存在，如果我们可以保证物体都是从前往后绘制的，那么就可以很大程度上减少 overdraw。这是因为，在后面绘制的物体由于无法通过深度测试，因此，就不会再进行后面的渲染处理。

在 Unity 中，那些渲染队列数目小于 2500（如“Background”“Geometry[22]”和“AlphaTest”）的对象都被认为是不透明（opaque[23]）的物体，这些物体总体上是从前往后绘制的，而使用其他的队列（如“Transparent”“Overlay”等）的物体，则是从后往前绘制的。这意味着，我们可以尽可能地把物体的队列设置为不透明物体的渲染队列，而尽量避免使用半透明队列。

而且，我们还可以充分利用 Unity 的渲染队列来控制绘制顺序。例如，在第一人称射

[22] n. 几何，几何学；几何形状，几何构造；几何体系

[23] adj. （玻璃、液体等）不透明的；难懂的，隐晦的；迟钝的 n. 不透明体；（用于底片）不透明颜料 v. 使不透明

击游戏中，对于游戏中的主要人物角色来说，他们使用的 shader 往往比较复杂，但是，由于他们通常会挡住屏幕的很大一部分区域，因此我们可以先绘制它们（使用更小的渲染队列）。而对于一些敌方角色，它们通常会出现在各种掩体后面，因此，我们可以在所有常规的不透明物体后面渲染它们（使用更大的渲染队列）。而对于天空盒子(Sky Box)来说，它几乎覆盖了所有的像素，而且我们知道它永远会出现在所有物体的后面，因此，它的队列可以设置为“Geometry ＋ 1”。这样，就可以保证不会因为它而造成 overdraw。

这些排序的思想往往可以节省掉很多渲染时间。

2） 时刻警惕透明物体

对于半透明对象来说，由于它们没有开启深度写入，因此，如果要得到正确的渲染效果，就必须从后往前渲染。这意味着，半透明物体几乎一定会造成 overdraw。如果我们不注意这一点，在一些机器上可能会造成严重的性能下降。例如，对于 GUI 对象来说，它们大多被设置成了半透明，如果屏幕中 GUI 占据的比例太多，而主摄像机又没有进行调整而是投影整个屏幕，那么 GUI 就会造成大量 overdraw。

因此，如果场景中包含了大面积的半透明对象，或者有很多层相互覆盖的半透明对象（即便它们每个的面积可能都不大），或者是透明的粒子效果，在移动设备上也会造成大量的 overdraw，这是应该尽量避免的。

对于上述 GUI 的这种情况，我们可以尽量减少窗口中 GUI 所占的面积。如果实在无能为力，我们可以把 GUI 的绘制和三维场景的绘制交给不同的摄像机，而其中负责三维场景的摄像机的视角范围尽量不要和 GUI 的相互重叠。当然，这样会对游戏的美观度产生一定影响，因此，我们可以在代码中对机器的性能进行判断，例如，首先关闭一些耗费性能的功能，如果发现这个机器表现非常良好，再尝试开启一些特效功能。

在移动平台上，透明度测试也会影响游戏性能。虽然透明度测试没有关闭深度写入，但由于它的实现使用了 discard[24]或 clip 操作，而这些操作会导致一些硬件的优化策略失效。例如，我们之前讲过 PowerVR 使用的基于瓦片的延迟渲染技术，为了减少 overdraw 它会在调用片元着色器前就判断哪些瓦片被真正渲染的。但是，由于透明度测试在片元着色器中使用了 discard 函数改变了片元是否会被渲染的结果，因此，GPU 就无法使用上述的优化策略了。也就是说，只有在执行了所有的片元着色器后，GPU 才知道哪些片元会被真正渲染到屏幕上，这样，原先那些可以减少 overdraw 的优化就都无效了。这种时候，使用透明度混合的性能往往比使用透明度测试更好。

3） 减少实时光照和阴影

实时光照对于移动平台是一种非常昂贵的操作。如果场景中包含了过多的点光源，并且使用了多个 Pass 的 Shader，那么很有可能会造成性能下降。例如，一个场景里如果包含了 3 个逐像素的点光源，而且使用了逐像素的 Shader，那么很有可能将 draw call 数目（CPU 的瓶颈）提高 3 倍，同时也会增加 overdraw（GPU 的瓶颈）。这是因为，对于逐像素的光源来说，被这些光源照亮的物体需要被再渲染一次。更糟糕的是，无论是静态批处理还是动态批处理，对于这种额外的处理逐像素光源的 Pass 都无法进行批处理，也就是说，它们会中断批处理。

当然，游戏场景还是需要光照才能得到出色的画面效果。我们看到很多成功的移动平台的游戏，它们的画面效果看起来好像包含了很多光源，但其实这都是骗人的。这些游戏往往使用了烘焙技术，把光照提前烘焙到一张光照纹理（lightmap）中，然后在运行时刻只需要根据纹理采样得到光照结果即可。另一个模拟光源的方法是使用 God Ray。场景中很多小型光源的效果都是靠这种方法模拟的。它们一般并不是真的光源，很多情况是通过

[24] v. 扔掉，弃置；打出（无用的牌），垫（牌） n. 被抛弃物；（纸牌游戏中）垫出的牌

透明纹理模拟得到的。在移动平台上，一个物体使用的逐像素光源数目应该小于 1（不包括平行光）。如果一定要使用更多的实时光，可以选择用逐顶点光照来代替。

6. 节省带宽

大量使用未经压缩的纹理以及使用过大的分辨率都会造成由于带宽而引发的性能瓶颈。

1） 减少纹理大小

之前提到过，使用纹理图集可以帮助我们减少 draw call 的数目，而这些纹理的大小同样是一个需要考虑的问题。需要注意的是，所有纹理的长宽比最好是正方形，而且长宽值最好是 2 的整数幂。这是因为有很多优化策略只有在这种时候才可以发挥最大效用。在 Unity5 中，即便我们导入的纹理长宽值并不是 2 的整数幂，Unity 也会自动把长宽转换到离它最近的 2 的整数幂值。但我们仍然应该在制作美术资源时就把这条规则谨记在心，防止由于放缩而造成不好的影响。

除此之外，我们还应该尽可能使用多级渐远纹理技术（mipmapping）和纹理压缩。在 Unity 中，我们可以通过纹理导入面板来查看纹理的各个导入属性。通过把纹理类型设置为 Advanced，就可以自定义许多选项，例如，是否生成多级渐远纹理（mipmaps），当勾选了 Generate Mip Maps 选项后，Unity 就会为同一张纹理创建出很多不同大小的小纹理，构成一个纹理金字塔。而在游戏运行中就可以根据距离物体的远近，来动态选择使用哪一个纹理。这是因为，在距离物体很远的时候，就算我们使用了非常精细的纹理，但肉眼也是分辨不出来的。这种时候，我们完全可以使用更小、更模糊的纹理来代替，这可以让 GPU 使用分辨率更小的纹理，大量节省访问的像素数目。在某些设备上，关闭多级渐远纹理往往会造成严重的性能问题。因此，除非我们确定该纹理不会发生缩放，例如 GUI 和 2D 游戏中使用的纹理等，都应该为纹理生成相应的多级渐远纹理。

纹理压缩同样可以节省带宽。但对于像 Android 这样的平台，有很多不同架构的 GPU，纹理压缩就变得有点复杂，因为不同的 GPU 架构有它自己的纹理压缩格式，例如，PowerVRAM 的 PVRTC 格式、Tegra 的 DXT 格式、Adreno 的 ATC 格式。所幸的是，Unity 可以根据不同的设备选择不同的压缩格式，而我们只需要把纹理压缩格式设置为自动压缩即可。但是，GUI 类型的纹理同样是个例外，一些时候由于对画质的要求，我们不希望对这些纹理进行压缩。

2） 利用分辨率缩放

过高的屏幕分辨率也是造成性能下降的原因之一，尤其是对于很多低端手机，除了分辨率高其他硬件条件并不尽如人意，而这恰恰是游戏性能的两个瓶颈：过大的屏幕分辨率和糟糕的 GPU。因此，我们可能需要对于特定机器进行分辨率的放缩。当然，这样可能会造成游戏效果的下降，但性能和画面之间永远是个需要权衡的话题

在 Unity 中设置屏幕分辨率可以直接调用 Screen.SetResolution。实际使用中可能会遇到一些情况，雨松 MOM0 有一篇文章（xuanyusong 网站的/archives/3205）详细讲解了如何使用这种技术，读者可参考。

7. 减少计算复杂度

计算复杂度同样会影响游戏的渲染性能。

1） Shader 的 LOD 技术

Shader 的 LOD 技术可以控制使用的 Shader 等级。它的原理是，只有 Shader 的 LOD 值小于某个设定的值，这个 Shader 才会被使用，而使用了那些超过设定值的 Shader 的物体将不会被渲染。

我们通常会在 SubShader 中使用类似下面的语句来指明该 shader 的 LOD 值：

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

我们也可以在 Unity Shader 的导入面板上看到该 Shader 使用的 LOD 值。

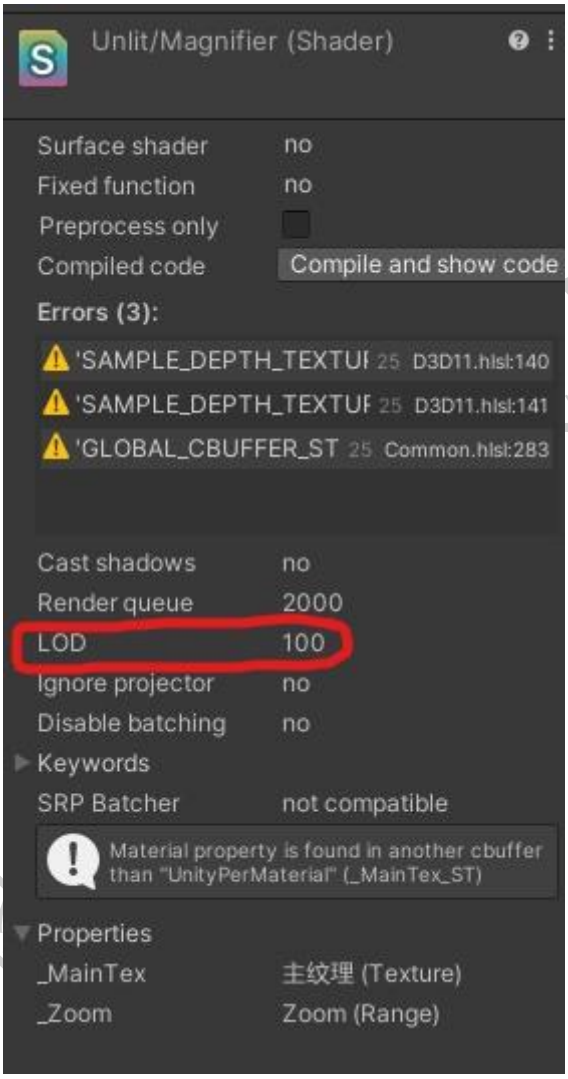


图7. 在 Inspector 界面查看 Shader 的 Lod 值

在默认情况下，允许的 LOD 等级是无限大的。这意味着，任何被当前显卡支持的 Shader 都可以被使用。但是，在某些情况下我们可能需要去掉一些使用了复杂计算的 Shader 渲染。这时，我们可以使用 Shader.maximumLOD 或 Shader.globalMaximumLOD 来设置允许的最大 LOD 值。

Unity 内置的 Shader 使用了不同的 LOD 值，例如，Diffuse 的 LOD 为 2OO，而 Bumped Specular 的 LOD 为 400。

2） 代码方面的优化

在实现游戏效果时，我们可以选择在哪里进行某些特定的运算。通常来讲，游戏需要计算的对象、顶点和像素的数目排序是对象数＜顶点数＜像素数。因此，我们应该尽可能地把计算放在每个对象或逐顶点上。例如实现高斯模糊和边缘检测时，我们把采样坐标的计算放在了顶点着色器中，这样的做法远好于把它们放在片元着色器中。

而在具体的代码编写上，不同的硬件甚至需要不同的处理。因此，一些普遍的规则在某些硬件上可能并不成立。更不幸的是，通常 Shader 代码的优化并不那么直观，尤其是一些平台上缺少相关的分析器，例如 iOS 平台。

首先第一点是，尽可能使用低精度的浮点值进行运算。最高精度的 float/highp 适用于存储诸如顶点坐标等变量，但它的计算速度是最慢的，我们应该尽量避免在片元着色器中使用这种精度进行计算。而 half/mediump 适用于一些标量、纹理坐标等变量，它的计算速度大约是 float 的两倍。而 fixed/lowp 适用于绝大多数颜色变量和归一化后的方向向量，在进行一些对精度要求不高的计算时，我们应该尽量使用这种精度的变量。它的计算速度大约是 float 的 4 倍，但要避免对这些低精度变量进行频繁的 swizzle 操作（如 color.xwxw）。还需要注意的是，我们应当尽量避免在不同精度之间的转换，这有可能会造成一定的性能下降。

对于绝大多数 GPU 来说，在使用插值寄存器把数据从顶点着色器传递给下一个阶段时，我们应该使用尽可能少的差值变量。例如，如果需要对两个纹理坐标进行插值，我们通常会把它们打包在同一个 float4 类型的变量中，两个纹理坐标分别对应了 xy 分量和 zw 分量。

然而，对于 PowerVR 平台来说，这种插值变量是非常廉价的，直接把不同的纹理坐标存储在不同的插值变量中，有时反而性能更好。尤其是，如果在 PowerVR 上使用类似 tex2D(_MainTex, uv.zw)，这样的语句来进行纹理采样，GPU 就无法进行一些纹理的预读取，因为它会认为这些纹理采样是需要依赖其他数据的。因此，如果我们特别关心游戏在 PowerVR 上的性能，就不应该把两个纹理坐标打包在同一个四维变量中。

尽可能不要使用全屏的屏幕后处理效果。如果美术风格实在是需要使用类似 Bloom、热扰动的屏幕特效，我们应该尽量使用 fixed/lowp 进行低精度运算（纹理坐标除外，可以使用 half/mediump）。那些高精度的运算可以使用查找表(LUT)或者转移到顶点着色器中进行处理。除此之外，尽量把多个特效合并到一个 Shader 中。例如，我们可以把颜色校正和添加噪声等屏幕特效在 Bloom 特效的最后一个 Pass 中进行合成。

a. 数的加减乘除

① 加法运算

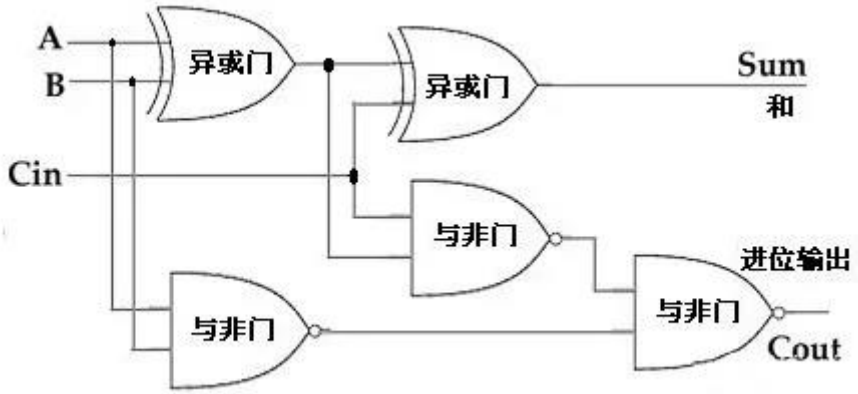


图8. 图 6.2 全加器逻辑电路

计算机在实际运算过程中，加法二进制的最终数据会在计算机的全加器逻辑电路进行，也就是物理层。其中，异或门输出的是 Y=A^B（即两个值不相同结果为 1，相同则为 0），与非门的输出是先进行与运算(&，即两个数值相同时返回的结果为 1，不同时返回的结果为 0)再进行取反的非运算(~)，即 Y=!(A&B)。通过推导可以得到公式：

$$Sum = A\text{^}B\text{^}Cin \quad Count = (A\text{^}B\&Cin)|(A\&B) \quad (15)$$

```
public int GetSum(int a,int b)
{
        if (b == 0) //判断参数 b 的值
        {
            return a;//返回数值 a
        }
        int sum, carry;
        sum = a ^ b;
        carry = (a & b) << 1;
        return GetSum(sum, carry);
}//递归运算
```

“按位与”运算符（&）的规则是，当两个数值进行与运算时，先将两个数值转换为位数相同的二进制数，再依次对每一位数进行比较，只有当对应的两个二进位均为 1 时，返回的结果才会为 1；如果两个二进位有一个不为 1，返回的结果就是 0。

01010100&00111011 = 00010000

“左移”运算符（<<）的运算规则是，在“<<”左边的运算数的各二进位全部被左移若干位，在“<<”右边的数值指定各二进位被移动的位数，高位丢弃，低位补 0。

如果 x=15，也就是 00001111，左移 2 位得 00111100 即 x<<2=60

由前面可以知道，加法运算其实就是不断的异或(^)加上移位操作的结果。

② 减法运算

当在进行减法运算时，首先想到的是加上一个负数。但是在计算机中，用于区分正负的符号位是不能直接参与运算的，在运算时必须分别计算符号位和数值位。当两个数值进行加法运算时，如果这两个数值是同号的（即都是正数或负数），则数值相加即可；如果这两个数值是异号的（即一正一负），则要进行减法运算。在进行减法运算时需要比较两者的绝对值大大小，用较大的数值去减去较小的数值，在得出运算结果时还要选择恰当的符号。

上述这些问题可以通过使用补码解决，补码可以把减法运算转换为加法运算。正数的补码最高位为符号 0，数值位和原码保持一致；而负数的补码最高位是符号 1，数值位是原码逐位取反得到反码，然后再在反码的最低位加一。数值 0 的补码只有一种形式，就是 N 位 0。

a = 3 = 0000 0011,b = 2 = 0000 0010

c = a-b=a+(-b)

c = 0000 0011 + 1111 1110

c = 1 0000 0001

因为是 8 位计算，最高位进位得到的 1 加入其中就是 9 位，超过了 8 位限制因此进位得到的 1 被舍弃，最终得到的结果就不是 1 0000 0001 而是 0000 0001。

在计算机中，由于机器码的位数是有限的，当最高位因为计算而进位超出了给定的取值范围，就被称为溢出。当两个正数进行加法运算，所得到的最终结果大于了机器码所能表示的最大正数，这一情况就被称为正溢。当两个负数进行加法运算时，所得到的最终结果小于了机器码所能表示的最小负数，这一情况就被称为负溢。在开发过程中出现溢出会导致计算的最终结果出现错误，因此正常情况下溢出是不被允许的。

③ 乘法运算

在计算机中乘法有两种计算方式，一种就是循环加法，一种是乘法器，乘法器分为模

哔哩哔哩 不游戏人生

拟乘法器和硬件乘法器。

2*3=0000 0010 * 0000 0011

3 的第 0 位是 1,2 左移 0 位，结果为 0000 0010 = 2

3 的第 1 位是 1,2 左移 1 位，结果为 0000 0100 = 4

3 的其他位都是 0，因此不再移位

两次左移的结果相加，即 0000 0010 + 0000 0100 = 2+4 = 6

可以看出乘法器是根据乘数的 bit 位的 0 与 1，来将被乘数进行左移多少位，然后累加得到最终的结果。

④ 除法运算

最简单的除法运算可以通过循环减法来实现，比如 5 除以 2,5 一直减去 2，直到减完的结果小于 2，得到的就是除法的结果，余数就是剩下的 1；除法也具有专门做除法的逻辑电路。

50 / 20 = 0011 0010 / 0001 0100

取 50 的最高位 001|1 0010，001 < 0001 0100

左移一位 0011| 0010, 0011 < 0001 0100

左移一位 0011 0|010，0 0110 < 0001 0100

左移一位 0011 00|10, 00 1100 < 0001 0100

左移一位 0011 001|0, 001 1001 > 0001 0100，商 1，余 101|0

将大于 20 的数位记录下来就是 0000 0010，结果为 2，余数 1010 = 10

所以结果就是 50 / 20 = 2 余 10

可以看出除法器是根据从被除数高位开始取数，每次移动一位取到数后和除数比较，如果大于除数就记录当前被除数的数位，然后将记录的结果变为二进制数就是最后的结果。

虽然现在在进行乘法运算和除法运算时，获得了除法器和乘法器的帮助使得计算的效率比过去的累加和累减高得多，但是这些操作依然是非常消耗性能的操作，尤其是在进行除法运算的时候，还必须要对数值进行大于小于的判断。因此在片元作色器的编写过程中应该尽可能的避免高性能消耗的除法操作，转而使用其他低性能消耗的运算进行组合替代。

b. 逻辑判断

CPU 的主要功能是解释计算机指令以及处理计算机软件中的数据，而逻辑判断是所有程序中必不可少的操作，一般情况下的逻辑判断都是在 CPU 上运行的，因此 CPU 是一台计算机的运算核心和控制核心。而 GPU 则和 CPU 不同，GPU(Graphics Processing Unit)作为计算机中的图像处理器，它比 CPU 更加的专门化，GPU 处理器专门用来进行图像运算，在执行高复杂度的数学计算和几何计算时，运行效率远远高于 CPU 的运行效率。换句话说，GPU 图像处理器就是专为执行复杂的数学和几何计算而诞生的。

哔哩哔哩欢乐游戏人生

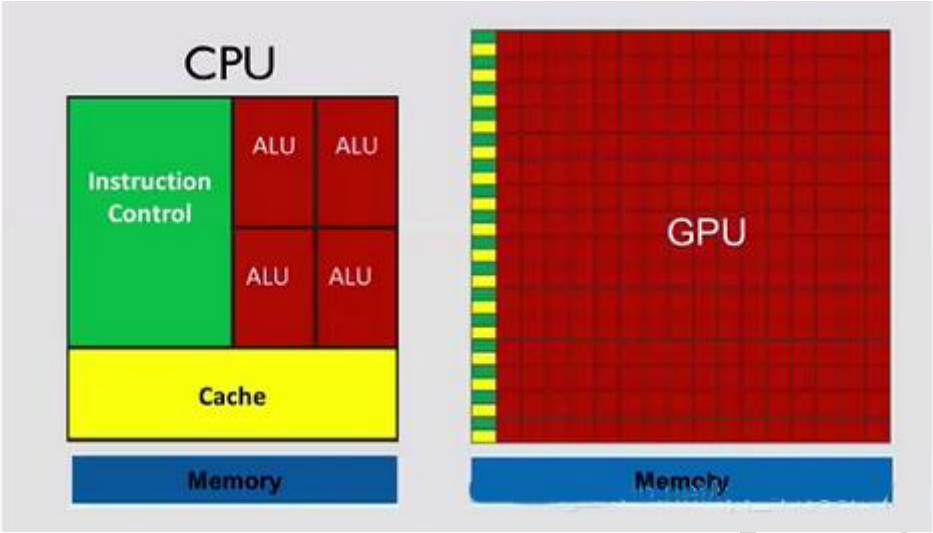


图9. 图 6.3 架构图

CPU 和 GPU 拥有各自擅长的领域，GPU 擅长处理那些具有极高的可预测性和大量相似的运算以及高延迟、高吞吐的架构运算。而 CPU 所擅长处理的就是需要快速完成响应的实时信息，因此 CPU 便需要针对延迟进行特别的优化，所以晶体管数量和能耗都被用在了分支预测、乱序执行、低延迟缓存等控制部分。

在 Unity Shader 中编写的三目运算最后也会被编译为 if-else 语句，毕竟三目运算实际上仅仅只是简化了的 if-else 语句。虽然 Shader 中的每个片元依然会执行逻辑判断的每个分支，但是寄存器的写入只在片元采取行动的分支上执行。因此在 Shader 可以用 Shader 变体来进行不同状态下的渲染分支，用一些函数去代替 if-else 的判断，不推荐在 Shader 中使用逻辑判断，因为会带来不必要的性能消耗。

例如可以用 setp()函数来做大小与的判断：

```
a = step(b, c);// 等价于  if (b <= c) a = 1 else a = 0
```

当需要为角色添加一个描边光效时，可以根据 NdotV 的结果来设定描边光效越靠近边缘越强，然后使用一个变量去控制描边光效在角色表面的范围。

在 Shader 代码编写之初，使用的是三目运算进行判断：

```
fixed3 c = (1 - NdotV)    < _StatusColorEdge ? fixed3(0,0,0) : blendColor;
```

经过一次优化，将三目运算变更为 clamp 函数和 ceil 函数的组合，clamp 函数会保证 NdotV 和边界控制的结果永远都是在 0-1 的范围，保证值的正确性；而 ceil 函数会返回大于或者等于指定表达式的最小整数，即将结果向下取整变为 0 或者 1，然后再对颜色进行插值运算：

```
fixed temp = ceil( clamp(1 - NdotV - _StatusColorEdge, 0, 1));
fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);
```

最后用 step 函数代替两个函数，同时减法也换成了加法：

```
fixed temp = step(NdotV + _StatusColorEdge, 1);
fixed3 c = lerp(fixed3(0,0,0), blendColor, temp);
```

当开发者在 Shader 脚本中使用 if 进行判断的时候，可以将其拆分为两种情况：

```
if (a > b)
{x}else
{x}
```

判断条件中的 b 为静态条件，即 const float 或者 define。当 b 为静态条件时，编译器

会对这类情况做优化，使其直接进行批量计算，因为同一批像素从 Shader 中获得的都是相同的数值。

判断条件中的 b 为动态数值，也就是说具体的值在运行前无法得知的，只有在程序运行的时候才会知道具体的值。就会导致 GPU 做分批处理，无法最大化利用同时计算，因为无法确定的值就会导致同一批像素在一个区间内浮动，那么同一时间有的返回 true 有的返回 false。

二十二、 额外拓展

1. 什么是伽马校正

要想渲染出更符合真实光照环境的场景，就需要使用线性空间。而 Unity 默认的空间是伽马空间，在伽马空间下进行渲染会导致很多非线性空间下的计算，从而引入一些误差。而要把伽马空间转换到线性空间，就需要进行伽马校正(Gamma Correction)。

伽马校正一词来源伽马曲线。通常，伽马曲线的表达式是$L_{out} = L_{in}^{\gamma}$。

其中指数部分的发音就是伽马。最开始的时候，人们使用伽马曲线来对拍摄的图像进行伽马编码(gamma encoding)。事情的起因可以从在真实环境中拍摄一张图片说起。摄像机的原理可以简化为，把进入到镜头内的光线亮度编码成图像（例如一张 JEPG）中的像素。如果采集到的亮度是 0，像素就是 0；亮度是 1，像素就是 1；亮度是 0.5 像素就是 0.5。如果我们只用 8 位空间来存储像素的每个通道的话，这意味着 0~1 区间可以对应 256 种不同的亮度值。但是，后来人们发现，人眼有一个有趣的特性，就是对光的灵敏度在不同亮度上是不一样的。在正常的光照条件下，人眼对较暗区域的变化更加敏感，而对较亮区域的变化比较不敏感。

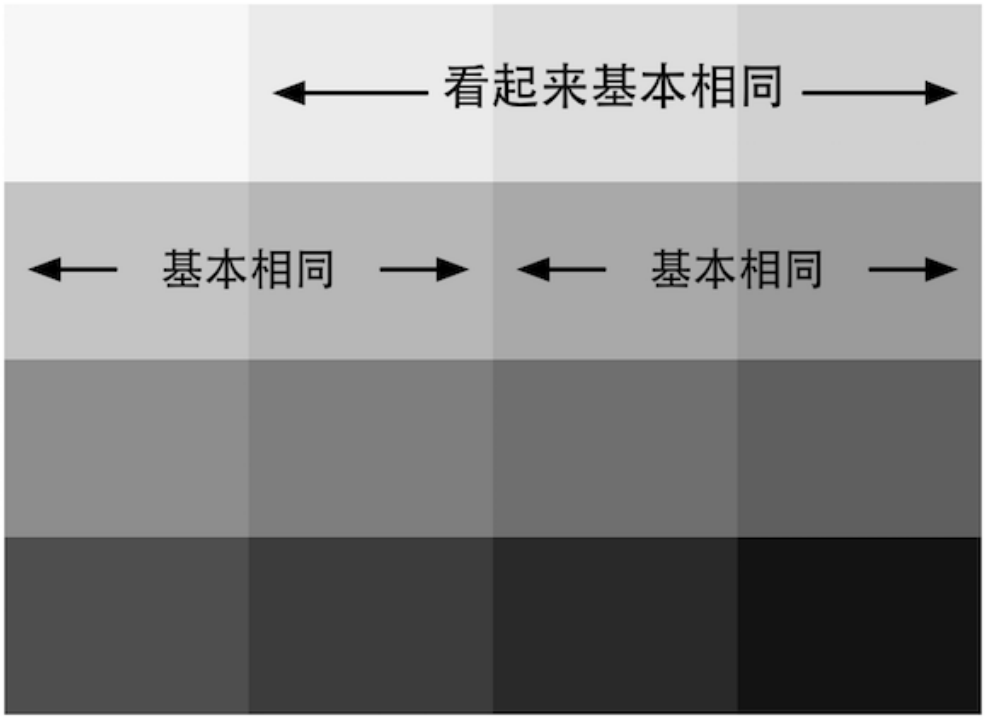


图10. 感知示例图

上图说明了一件事情，亮度上的线性变化对人眼感知来说是非均匀的。Youtube 上有一个名为 Color is Broken 的非常有趣的视频，在这个视频中，作者用了一个非常生动的例子来说明这个现象。当一个屋子的光照由一盏灯增加到两盏灯的时候，人眼对这种亮度变化的感知性要远远大于从 101 盏灯增加到 102 盏灯的变化，尽管从物理上来说这两种变化

基本是相同的。

那么，这和之前讲的拍照有什么关系呢？如果使用 8 位空间来存储每个通道的话，我们仍然把 0.5 亮度编码成值为 0.5 的像素，那么暗部和亮部区域我们都使用了 128 种颜色来表示，但实际上，对亮部区域使用这么多颜色是种存储浪费。一种更好的方法是，我们应该把把更多的空间来存储更多的暗部区域，这样存储空间就可以被充分利用起来了。摄影设备如果使用了 8 位空间来存储照片的话，会使用大约为 0.45 的编码伽马来对输入的亮度进行编码，得到一张编码后的图像。因此，图像中 0.5 像素值对应的亮度其实并不是 0.5，而大约为 0.22。这是因为，$0.5 \approx 0.22^{0.45}$。

对拍摄图像使用的伽马编码使得我们可以充分利用图像的存储空间。但当把图片放到显示器里显示时，我们应该对图像再进行一次解码操作，使得屏幕输出的亮度和捕捉到的亮度是符合线性的。这时，人们发现了一个奇妙的巧合一 CRT 显示器本身几乎已经自动做了这个解码操作！这又从何说起呢？

在早期，CRT（Cathode Ray Tube，阴极射线管）几乎是唯一的显示设备。这类设备的显示机制是，使用一个电压轰击它屏幕上的一种图层，这个图层就可以发亮，我们就可以看到图像了。但 CRT 显示器有一个特性，它的输入电压和显示出来的亮度关系不是线性的，也就是说，如果我们把输入电压调高两倍，屏幕亮度并没有提高两倍。我们把显示器的这个伽马曲线称为显示伽马（diplay gamma）。非常巧合的是，CRT 的显示伽马值大约就是编码伽马的倒数。CRT 显示器的这种特性，正好补偿了图像捕捉设备的伽马曲线，人们想，“天呐，太棒了，我们不需要做任何调整就可以让拍摄的图像在电脑上看起来和原来的一样了！”虽然现在 CRT 设备很少见了，并且后来出现的显示设备有着不同的伽马响应曲线，但是，人们仍在硬件上做了调整来提供兼容性。

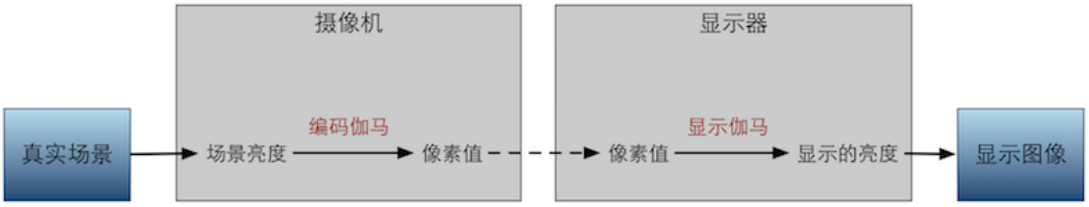


图11.　　编码伽马和显示伽马

随后，微软联合爱普生、惠普提供了 sRGB 颜色空间标准，推荐显示器的显示伽马值为 2.2，并配合 0.45 的编码伽马就可以保证最后伽马曲线之间可以相互抵消（因为 2.2×0.45≈1）。绝大多数的摄像机、PC 和打印机都使用了上述的 sRGB 标准。

读到现在，读者可能还是有所疑问，这和渲染有什么关系？答案是关系很大。事实上，由于游戏界长期以来都忽视了伽马校正的问题，造成了我们渲染出来的游戏总是暗沉沉的，总是和真实世界不像。由于编码伽马和显示伽马的存在，我们一不小心就可能在非线性空间下进行计算，或是使得输出的图像是非线性的。

对于输出来说，如果我们直接输出渲染结果而不进行任何处理，在经过显示器的显示伽马处理后，会导致图像整体偏暗，出现失真的状况。

我们放置一个球体，并把场景中的环境光照设为全黑，再把平行光的方向设置为从上方直接射到球体表面，球体使用的材质为内置的漫反射材质。

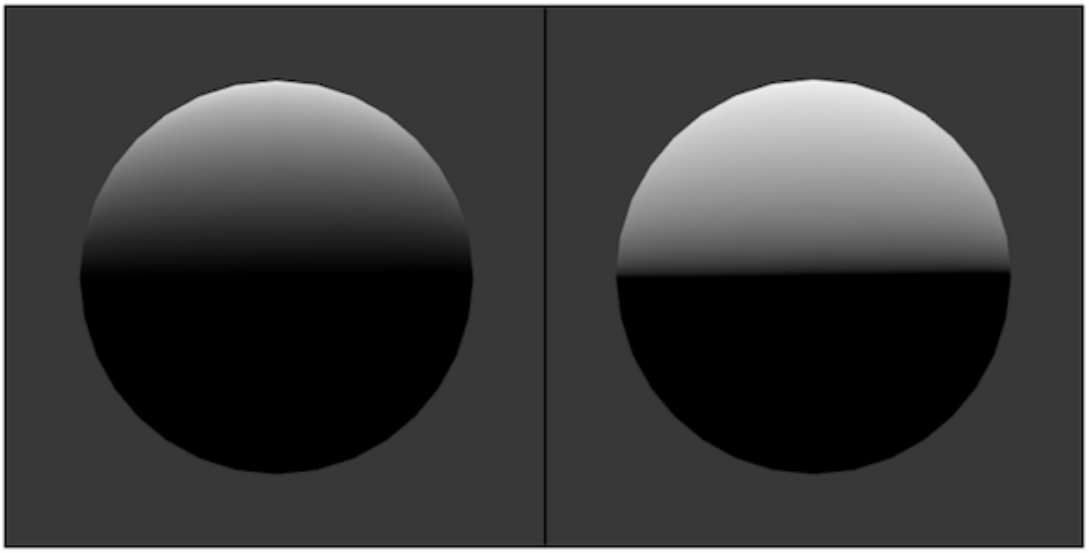


图12.　　左边为伽马空间下的渲染结果，右边为线性空间下的渲染结果

从上图可以看出，伽马空间下的渲染结果整体偏暗，一些读者甚至认为这看起来更加正确。然而，实际此时屏幕输出的亮度和球面的光照结果并不是线性的。

假设球面上有一点 A，它的法线和光线方向成 60°，还有一点 B，它的法线和光线方向成 90°。那么，在 Shader 中计算漫反射光照时，我们会得出 A 的输出是（0.5，0.5，0.5），B 的输出是（1.0，1.0，1.0）。

在上图的左图中，我们没有进行伽马校正，因此，由于显示器存在显示伽马就引入了非线性关系，也就是说 A 点的亮度其实并不是 B 亮度的一半，而约为它的 1/4。在上图的右图中，我们使用了线性空间，Unity 会在把像素写入颜色缓冲前进行一次伽马校正，来抵消屏幕的显示伽马的作用，此时得到屏幕亮度才是真正跟像素值成正比的。

伽马的存在还会对混合(blend[25])造成影响。在场景中，我们放置了 3 个互相重叠的圆，它们使用的材质均为简单的透明混合材质，并使用了一个边界模糊的圆作为输入纹理，场景在伽马空间和线性空间下的效果会有所不同。

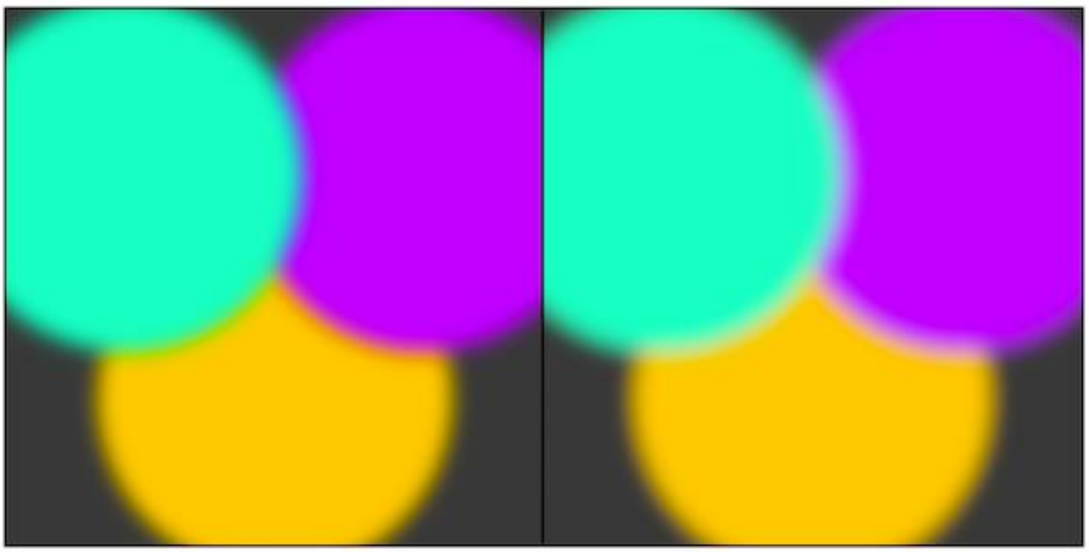


图13.　　左图为伽马空间下的混合结果；右图为线性空间下的混合结果

在上图左图所示的伽马空间下，我们可以看到在绿色和红色的混合边界处出现了不正

[25] v. （使）混合，调和；（使）协调，融合；调制，配制　n. 混合物；交融，融合；混合词

常的蓝色渐变。而正确的混合结果应该是右边图所示的从绿色到红色的渐变。除此之外，我们也可以看到左边图中交叉的边界似乎都变暗了。这是因为在混合后进行输出时，显示器的显示伽马导致接缝处颜色变暗。

实际上，渲染中非线性输入最有可能的来源就是纹理。为了充分利用存储空间，大多数图像文件都进行了提前的校正，即已经使用了一个编码伽马对像素值编码。但这意味着它们是非线性的，如果我们在 Shader 中直接使用纹理采样值就会造成在非线性空间的计算，使得结果和真实世界的结果不一致。我们在使用多级渐远纹理（mipmaps）时也需要注意。如果纹理存储在非线性空间中，那么在计算多级渐远纹理时就会在非线性空间里计算。由于多级渐远纹理的计算是种线性计算——即采样的过程，需要对某个方形区域内的像素取平均值，这样就会得到错误的结果。正确的做法是，我们要把非线性的纹理转换到线性空间后再计算多级渐远纹理。

如上所说，伽马的存在使得我们很容易得到非线性空间下的渲染结果。在游戏渲染中，我们应该保证所有的输入都被转换到了线性空间下，并在线性空间下进行各种光照计算，最后在输出前通过一个编码伽马进行伽马校正后再输出到颜色缓冲中。Untiy 的颜色空间设置就可以满足我们的需求。当我们选择伽马空间时，实际上就是“放任模式”，不会对 Shader 的输入进行任何处理，即使输入可能是非线性的；也不会对输出像素进行任何处理，这意味着输出的像素会经过显示器的显示伽马转换后得到非预期的亮度，通常表现为整个场景会比较昏暗。当选择线性空间时，Unity 会把输入纹理设置为 sRGB 模式，在这种模式下，硬件在对纹理进行采样时会自动将其转换到线性空间中；并且，GPU 会在 Shader 写入颜色缓冲前自动进行伽马校正或是保持线性在后面进行伽马校正，这取决于当前的渲染配置。如果我们开启了 HDR 的话，渲染就会使用一个浮点精度的缓冲。这些缓冲有足够的精度不需要我们进行任何伽马校正，此时所有的混合和屏幕后处理都是在线性空间下进行的。当渲染完成要写入显示设备的后备缓冲区（backbuffer）时，再进行一次最后的伽马校正。如果我们没有使用 HDR，那么 Unity 就会把缓冲设置成 sRGB 格式，这种格式的缓冲就像一个普通的纹理一样，在写入缓冲前需要进行伽马校正，在读取缓冲时需要再进行一次解码操作。如果此时开启了混合（像我们之前的那样），在每次混合时，硬件会首先把之前颜色缓冲中存储的颜色值转换回线性空间中，然后再与当前的颜色进行混合，完成后再进行伽马校正，最后把校正后的混合结果写入颜色缓冲中。这里需要注意，透明通道是不会参与伽马校正的。

然而，Unity 的线性空间并不是所有平台都支持的，例如，移动平台就无法使用线性空间。此时，我们就需要自己在 Shader 中进行伽马校正。对非线性输入纹理的校正代码通常如下：

```
float 3 diffuseCol = pow(tex2D(diffTex,texCoord),2.2);
```

在最后输出前，对输出像素值的校正代码通常如下面这样：

```
fragColor.rgb = pow(fragColor.rgb,1.0/2.2);
return fragColor;
```

但是，手动对输出像素进行伽马校正会在使用混合时出现问题。这是因为，校正会导致写入颜色缓冲内的颜色是非线性的，这样混合就发生在非线性空间中。一种解决方法是，在中间计算时不要对输出颜色值进行伽马校正，但在最后需要进行一个屏幕后处理操作来对最后的输出进行伽马校正，也就是说我们需要保证伽马校正发生在渲染的最后一步，但这可能会造成一定的性能损耗。

你会说，伽马这么麻烦，什么时候可以舍弃它呢？如果有一天我们对图像的存储空间能够大大提升，通用的格式不再是 8 位时，例如是 32 位时，伽马也许就会消失。因为，我们有足够多的颜色空间可以利用，不需要为了充分利用存储空间进行伽马编码的工作了。

这就是我们的 HDR 了。

2. HDR

HDR 是 High Dynamic Range 的缩写，即高动态范围。动态范围指的是最高和最低亮度值的比值。在真实世界中，一个场景中最亮和最暗区域的范围可以非常大，例如，太阳发出的光可能要比场景中某个影子上的点的亮度高出数万倍，这些范围远远超过图像或显示器能够显示的范围。

通常在显示设备使用的颜色缓冲中每个通道的精度为 8 位，意味着我们只能使用这 256 种（即 2^8）不同的颜色来表示真实世界中的所有亮度，所以在虚拟世界中显示颜色会损失一定的精度。早期的拍摄设备利用人眼的特点，使用了伽马曲线来对捕捉到的图像进行编码，尽可能充分地利用这些有限的存储空间。

HDR 的出现给我们带来了新的希望，HDR 使用远远高于 8 位的精度（如 32 位）来记录亮度信息，使得我们可以表示超过 0-1 内的亮度值，从而可以更加精确地反映真实的光照环境。虽然在最后我们依旧需要把信息转换到显示设备使用的 LDR 内，但是中间的计算却可以让我们得到更加真实可信的效果。

NVIDIA 曾总结过使用 HDR 进行渲染的动机：让亮的物体可以真的非常亮，暗的物体可以真的非常暗，同时又可以看到两者之间的细节。

使用 HDR 来存储的图像被称为高动态范围图像（HDRI），例如使用一张 HDRI 图像来作为场景的天空盒（Skybox），可以更加真实地反映物体周围的环境，从而得到更加真实的反射效果。

HDR 对与光照的叠加也有非常重要的作用。如果我们的场景中有很多光源获胜光源强度很大，那么一个物体经过多次光照渲染叠加后最终得到的光照亮度很可能会超过 1.如果没有使用 HDR，超过 1 的部分会被截取掉，使得场景丢失了很多亮部区域细节。一旦开启了 HDR，我们就可以保留这些超过范围的光照结果，尽管最后我们仍然，需要把它们转换到 LDR 进行显示，但我们可以使用色调映射（tonemapping）技术来控制这个转换的过程，从而允许我们最大限度地保留需要的亮度细节。

HDR 的使用可以允许我们在屏幕后处理中拥有更多控制权。例如，Bloom 效果需要检测屏幕中亮度大于某个阈值的像素，把它们提取出来后进行模糊，再叠加到原图像中。如果我们不使用 HDR，就只能使用小于 1 的阈值来提取需要的像素，但很多时候我们实际上是需要提取那些非常亮的区域，比如车窗上的强烈的反光。如果不使用 HDR，这些值实际上可能和街道上一些颜色偏白的区域一样，使应保持阴暗的地方也出现泛光的效果。而使用 HDR，这些问题都可以得到解决，我们只需使用超过 1 的阈值来提取那些非常亮的区域即可。

使用 HDR 可以让我们不会丢失高亮度区域的颜色值，提供了更真实的光照效果，并为一些屏幕后处理提供更多的控制能力。但是 HDR 也有缺点，由于使用了浮点缓冲来存储高精度图像，不仅需要更大的显存空间，渲染速度也会变慢，除此之外，一些硬件并不支持 HDR。而且一旦使用 HDR，我们就无法使用硬件的抗锯齿功能。

在 Unity 中如果我们同时打开了硬件的抗锯齿（在 Edit -> Project Settings -> Qualitty -> Anti Aliasing 中打开）和摄像机的 HDR，Unity 会向我们发出警告，提醒我们因为开启了抗锯齿而无法再使用 HDR 缓冲。(我们可以基于屏幕后处理的抗锯齿操作来弥补这一点)

3. MatCap

1） 概述

Material Capture(材质捕获)，通常被简称为 MatCap，在 Zbrush、Sculptris、Mudbox 等 3D 软件中有比较多的使用。

MatCap Shader 的基本思路是，使用某特定材质球的贴图，作为当前材质的视图空间环境贴图（view-space environment map），来实现具有均匀表面着色的反射材质物体的显示。考虑到物体的所有法线的投影的范围 x 和 y 值均在-1 到 1 之间,构成了一个圆形，所以 MatCap 贴图中存储光照信息的区域是一个圆形。基于 MatCap 思想的 Shader，可以无需提供任何光照，只需提供一张或多张合适的 MatCap 贴图作为光照结果的“指导”即可。

图14.　　MatCap 示例

不像一般的 Shader 需要提供光照，需要在 Shader 代码中进行漫长的演算，基于 MatCap 思想的 Shader，相当于 MatCap 贴图就已经把光照结果的标准答案告知 Shader，我们只用在试卷下写出答案，进行一些再加工即可。

需要注意，MatCap Shader 有一定的局限性。因为从某种意义上来说，基于 MatCap 的 Shader，就是某种固定光照条件下，从某个特定方向，特定角度的光照表现结果。

正是因为是选择的固定的 MatCap 贴图，得到相对固定的整体光照表现，若单单仅使用 MatCap，就仅适用于摄像机不调整角度的情形，并不适合摄像机会频繁旋转，调节角度的情形。但我们可以在某些 Shader 中，用 MatCap 配合与光照交互的其他属性，如将 MatCap 结合一个作为光照反射的颜色指导的 Reflection Cube Map，就有了与光照之间的交互表现。这样，就可以适当弥补 MatCap 太过单一整体光照表现的短板。

2） MatCap 贴图的获取

如果需要使用基于 MatCap 的 Shader，合适的 MatCap 贴图是必不可少的。显而易见，MatCap 贴图的获取，一般来说有两种方式。

① 自己制作，即对着 3D 软件中的材质球截图。

② 从网络上获取。在网络上使用“matcap“等关键字搜索后获得。

获取 MatCap 贴图的网址

④ http://pixologic.com/zbrush/downloadcenter/library/#prettyPhoto

⑤ https://www.google.com.hk/search?q=MatCap&newwindow=1&safe=strict&hl=zh-CN&biw=1575&bih=833&tbm=isch&tbo=u&source=univ&sa=X&ved=0ahUKEwju8JDTpZnSAhUGn5QKHawODTIQsAQIIg

⑥ https://www.pinterest.com/evayali/matcap/

3） 基于 MatCap 思想的车漆 Shader 实现

此车漆 Shader，除了用到 MatCap，主要还需要提供一个 Reflection Cube Map 作为反射的颜色指导，就可以适当弥补 MatCap 太过单一的整体光照表现的短板，实现非常真实且高效的车漆 Shader 效果。

此 Shader 创建一个 Material 实例后，Material 在 Inspector 中的属性表现如下。

哔哩哔哩不游戏

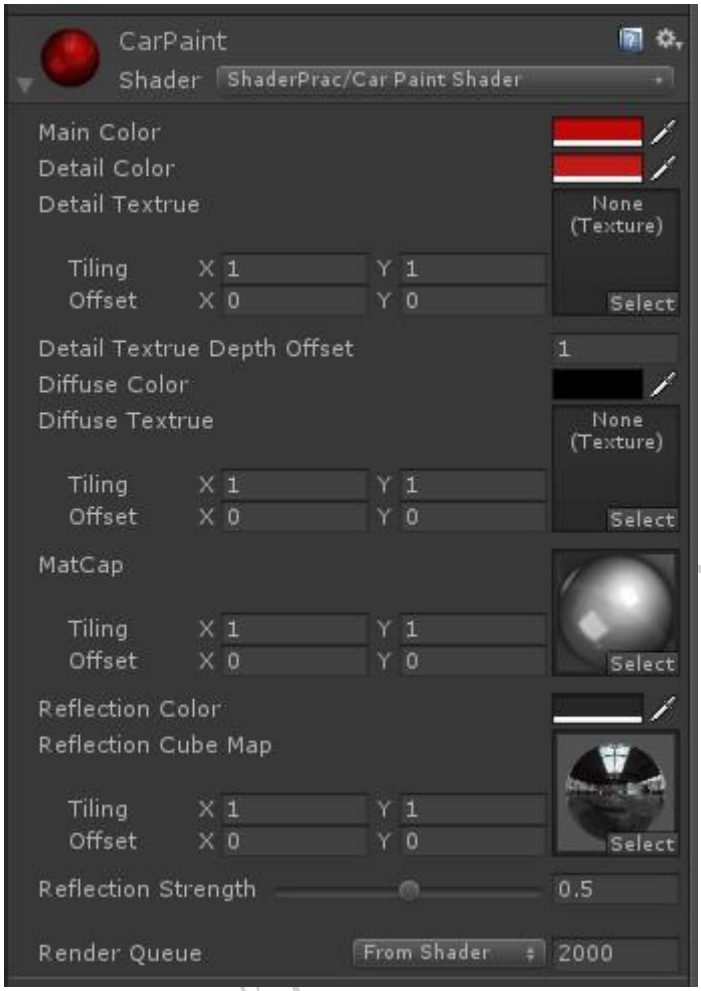


图15. Mateialc 的检视面板

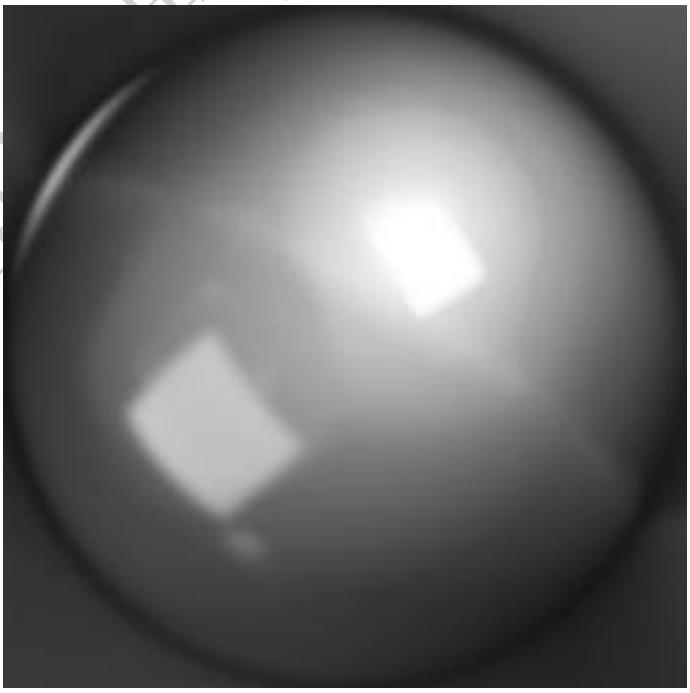


图16. MatCap 贴图

a. 代码

PS：

```
Shader "ShaderPrac/Car Paint Shader"
{
    Properties
    {
        //主颜色
        _MainColor("Main Color", Color) = (1.0, 1.0, 1.0, 1.0)
        //细节颜色
        _DetailColor("Detail Color", Color) = (1.0, 1.0, 1.0, 1.0)
        //细节纹理
        _DetailTex("Detail Textrue", 2D) = "white" {}
        //细节纹理深度偏移
        _DetailTexDepthOffset("Detail Textrue Depth Offset", Float) = 1.0
        //漫反射颜色
        _DiffuseColor("Diffuse Color", Color) = (0.0, 0.0, 0.0, 0.0)
        //漫反射纹理
        _DiffuseTex("Diffuse Textrue", 2D) = "white" {}
        //Material Capture 纹理
        _MatCap("MatCap", 2D) = "white" {}
        //反射颜色
        _ReflectionColor("Reflection Color", Color) = (0.2, 0.2, 0.2, 1.0)
        //反射立方体贴图
        _ReflectionMap("Reflection Cube Map", Cube) = "" {}
        //反射强度
        _ReflectionStrength("Reflection Strength", Range(0.0, 1.0)) = 0.5
    }

    SubShader
    {
        Tags
        {
            "Queue" = "Geometry"
            "RenderType" = "Opaque"
        }

        Pass
        {
            Blend Off
            Cull Back
            ZWrite On

            CGPROGRAM
```

哔哩哔哩次元游戏人生

```
#include "UnityCG.cginc"
#pragma fragment frag
#pragma vertex vert

float4 _MainColor;
float4 _DetailColor;
sampler2D _DetailTex;
float4 _DetailTex_ST;
float _DetailTexDepthOffset;
float4 _DiffuseColor;
sampler2D _DiffuseTex;
float4 _DiffuseTex_ST;
sampler2D _MatCap;
float4 _ReflectionColor;
samplerCUBE _ReflectionMap;
float _ReflectionStrength;

//顶点输入结构
struct VertexInput
{
    float3 normal : NORMAL;
    float4 position : POSITION;
    float2 UVCoordsChannel1: TEXCOORD0;
};

//顶点输出(片元输入)结构
struct VertexToFragment
{
    float3 detailUVCoordsAndDepth : TEXCOORD0;
    float4 diffuseUVAndMatCapCoords : TEXCOORD1;
    float4 position : SV_POSITION;
    float3 worldSpaceReflectionVector : TEXCOORD2;
};

//-----------------------------------------------------------
// 顶点着色器
//-----------------------------------------------------------
VertexToFragment vert(VertexInput input)
{
    VertexToFragment output;

    //漫反射 UV 坐标准备：存储于 TEXCOORD1 的前两个坐标 xy。
    output.diffuseUVAndMatCapCoords.xy =
TRANSFORM_TEX(input.UVCoordsChannel1, _DiffuseTex);
```

哔哩哔哩次元游戏人生

```
//MatCap 坐标准备：将法线从模型空间转换到观察空间，存储于TEXCOORD1 的后两个纹理坐标 zw
output.diffuseUVAndMatCapCoords.z = dot(normalize(UNITY_MATRIX_IT_MV[0].xyz), normalize(input.normal));
output.diffuseUVAndMatCapCoords.w = dot(normalize(UNITY_MATRIX_IT_MV[1].xyz), normalize(input.normal));
//归一化的法线值区间[-1,1]转换到适用于纹理的区间[0,1]
output.diffuseUVAndMatCapCoords.zw = output.diffuseUVAndMatCapCoords.zw * 0.5 + 0.5;

//坐标变换
output.position = mul(UNITY_MATRIX_MVP, input.position);

//细节纹理准备准备 UV,存储于 TEXCOORD0 的前两个坐标 xy
output.detailUVCoordsAndDepth.xy = TRANSFORM_TEX(input.UVCoordsChannel1, _DetailTex);

//深度信息准备,存储于 TEXCOORD0 的第三个坐标 z
output.detailUVCoordsAndDepth.z = output.position.z;

//世界空间位置
float3 worldSpacePosition = mul(unity_ObjectToWorld, input.position).xyz;

//世界空间法线
float3 worldSpaceNormal = normalize(mul((float3x3)unity_ObjectToWorld, input.normal));

//世界空间反射向量
output.worldSpaceReflectionVector = reflect(worldSpacePosition - _WorldSpaceCameraPos.xyz, worldSpaceNormal);

return output;
}

//-----------------------------------------------------------
// 片元着色器
//-----------------------------------------------------------
float4 frag(VertexToFragment input) : COLOR
{
//镜面反射颜色
float3 reflectionColor = texCUBE(_ReflectionMap, input.worldSpaceReflectionVector).rgb * _ReflectionColor.rgb;
```

哔哩哔哩次元游戏人生

```
                //漫反射颜色
                float4 diffuseColor = tex2D(_DiffuseTex,
input.diffuseUVAndMatCapCoords.xy) * _DiffuseColor;

                //主颜色
                float3 mainColor = lerp(lerp(_MainColor.rgb, diffuseColor.rgb,
diffuseColor.a), reflectionColor, _ReflectionStrength);

                //细节纹理
                float3 detailMask = tex2D(_DetailTex,
input.detailUVCoordsAndDepth.xy).rgb;

                //细节颜色
                float3 detailColor = lerp(_DetailColor.rgb, mainColor, detailMask);

                //细节颜色和主颜色进行插值，成为新的主颜色
                mainColor = lerp(detailColor, mainColor,
saturate(input.detailUVCoordsAndDepth.z * _DetailTexDepthOffset));

                //从提供的 MatCap 纹理中，提取出对应光照信息
                float3 matCapColor = tex2D(_MatCap,
input.diffuseUVAndMatCapCoords.zw).rgb;

                //最终颜色
                float4 finalColor=float4(mainColor * matCapColor * 2.0, _MainColor.a);

                return finalColor;
            }

            ENDCG
        }
    }

    Fallback "VertexLit"
}
```

Shader 注释已经比较详细，下面对代码中也许会不太理解，需要注意的地方进行说明。

要使用 MatCap 贴图，主要是将法线从模型空间转换到视图空间，并切换到适合提取纹理 UV 的区域[0,1]。

Unity 内置的矩阵 UNITY_MATRIX_IT_MV，是 UNITY_MATRIX_MV 的逆转置矩阵，其作用正是将法线从模型空间转换到观察空间(Object To View)。于是顶点着色器 vert 中的这两句代码就很容易理解了。

PS：

哔哩哔哩次元游戏人生

//MatCap 坐标准备：将法线从模型空间转换到观察空间，存储于 TEXCOORD1 的后两个纹理坐标 zw

output.diffuseUVAndMatCapCoords.z =dot(normalize(UNITY_MATRIX_IT_MV[0].xyz), normalize(input.normal));

output.diffuseUVAndMatCapCoords.w= dot(normalize(UNITY_MATRIX_IT_MV[1].xyz), normalize(input.normal));

而得到的视图空间的法线，区域是[-1，1],要转换到提取纹理 UV 的区域[0,1]，就需要乘以 0.5 并加上 0.5，那么顶点着色器 vert 中接下来的的这句代码也就可以理解了。

PS：

//归一化的法线值区间[-1,1]转换到适用于纹理的区间[0,1]

output.diffuseUVAndMatCapCoords.zw= output.diffuseUVAndMatCapCoords.zw * 0.5 + 0.5;

稍后，在片元着色器 frag 中，将使用顶点着色器 vert 中准备好的法线转换成的 UV 值，提取出 MatCap 的光照细节即可。

PS：

//从提供的 MatCap 纹理中，提取出对应光照信息

float3 matCapColor = tex2D(_MatCap, input.diffuseUVAndMatCapCoords.zw).rgb;

此 Car Paint Shader 其他实现，主要就是一些基本的 Shader 知识点的配合，如顶点坐标转换，反射，漫反射，细节纹理的计算，都是比较基础的内容，这里就不再赘述，直接看 Shader 源码即可。

从 Github 下载：【Github】Shader 源码与相关 MatCap 资源下载

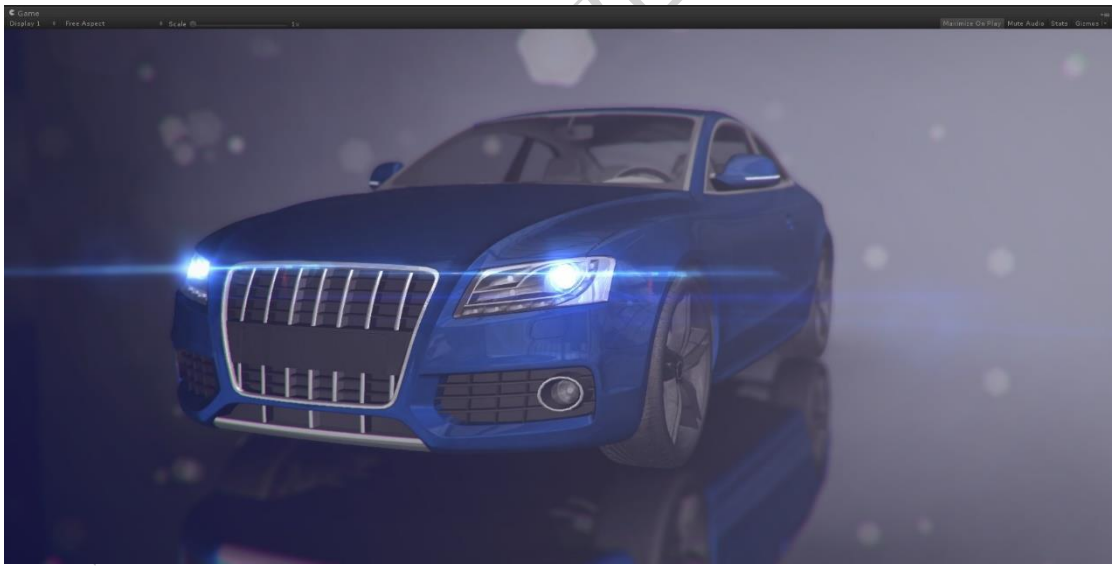


图17.　　MatCap 车漆

Visual Effect Graph

一、 篝火

哔哩哔哩次元游戏人生

PhotoShop

一、 去除水印

二、 快捷键

Ctrl + T 自由变换
Ctrl + Shift + N 新建图层
Ctrl + G 将选中图层进行编组
Ctrl + E 合并图层
Ctrl + Shift + E 合并可见的图层
Ctrl + D 取消选择
Ctrl + Shift + I 反选
W 魔棒工具
B 画笔工具
E 橡皮擦工具
T 文字
X 切换前景色和背景色

哔哩哔哩次元游戏人生