哔哩哔哩次元游戏人生

目　录

目　录 I
C、C#、C++ 18
一、　数据类型 18
二、　运算符优先级和结合性 18
三、　构造函数 19
四、　虚函数 19
1. 虚函数 19
2. 纯虚函数 19
3. 接口(Interface)和抽象类的区别 19
五、　C#中四种访问修饰符 20
1. 属性修饰符 20
2. 存取修饰符 20
3. 类修饰符 20
4. 成员修饰符 20
六、　ref 和 out 关键字 20
1. ref 20
2. out 21
3. ref 和 out 的相同点 21
4. ref 和 out 的不同点 21
七、　分部类和方法 21
1. 分部类 21
2. 分部方法 22
八、　面向对象的三大特点 22
1. 继承 22
2. 封装 22
3. 多态性 22
九、　面向对象程序设计方法的主要优点 22
十、　C#索引 23
十一、　值类型和引用类型 23
Lua 语言 24
一、　Lua 基本语法 24
1. 单行注释 24
2. 多行注释 24
3. 标示符 24
4. 关键字 24
5. 全局变量 24
二、　数据类型 25
1. 空-nil 26
2. 布尔-boolean 26
3. 数字-number 27
4. 字符串-string 27
5. 表-table 28

哔哩哔哩次元游戏人生

6. 函数-function 29
7. 线程-thread 30
8. 自定义类型-userdata 30
三、 Lua 的变量、赋值、索引 30
1. 变量 30
2. 赋值语句 31
3. 索引 31
四、 循环 32
1. 循环类型 32
2. 循环控制语句 35
3. 无限循环 36
五、 流程控制 36
1. if 语句 37
2. if…else…语句 37
3. if...elseif...else 语句 38
4. if 嵌套语句 38
六、 函数 39
1. 函数定义 39
2. 多返回值函数 40
3. 可变参数 41
4. 固定参数加上可变参数 42
5. 命名参数 43
七、 再论函数 44
1. 闭包 44
2. 非全局变量 46
3. 正确的尾调用 48
八、 运算符 50
1. 算术运算符 50
2. 关系运算符 50
3. 逻辑运算符 51
4. 其他运算符 51
5. 运算符优先级 51
九、 Lua 字符串 51
1. 转义字符 52
2. 字符串操作 52
3. 字符串截取 54
4. 字符串格式化 54
5. 字符与整数相互转换 56
6. 匹配模式 56
十、 Lua 数组 57
1. 一维数组 57
2. 多维数组 57
十一、 Lua 迭代器 59
1. 泛型 for 迭代器 59

哔哩哔哩次元游戏人生

2. 无状态的迭代器.....60
3. 多状态的迭代器.....60
十二、 表-table.....61
1. 表的构造.....61
2. Table 操作.....62
十三、 数据文件与持久化.....64
1. 序列化.....66
十四、 环境.....70
1. 使用动态名字访问全局变量.....71
2. 声明全局变量.....72
3. 非全局的环境.....73
十五、 模块与包.....75
1. 模块.....75
2. require 函数.....75
3. C 包.....77
十六、 Package.....77
1. 基本方法.....77
2. 私有成员-Privacy.....79
3. 包与文件.....80
4. 使用全局表.....80
十七、 元表-Metatable.....82
1. setmetatable(table,metatable).....82
2. getmetatable(table).....82
3. __index 元方法.....82
4. __newindex 元方法.....83
5. 为表添加操作符.....84
6. __call 元方法.....85
7. __tostring 元方法.....86
十八、 协同程序-coroutine.....87
1. 什么是协同(coroutine).....87
2. 线程和协同程序区别.....87
3. 基本语法.....87
十九、 文件 I/O.....89
1. 模式值.....90
2. 简单模式.....90
3. 完全模式.....91
二十、 错误处理.....92
1. 语法错误.....92
2. 运行错误.....93
3. 错误处理.....93
二十一、 调试-Debug.....94
二十二、 垃圾回收.....97
1. 垃圾回收器函数.....97
二十三、 面向对象程序设计.....98

哔哩哔哩次元游戏人生

1. 类.....99
2. 继承.....100
3. 多重继承.....102
4. 私有性-privacy.....104
5. Single-Method 的对象实现方法.....105
二十四、 优化.....106
1. 尽可能使用局部变量.....106
二十五、 工具.....106
1. IntelliJ IDEA.....106
Unity.....109
一、 PipelineAsset.....109
1. Quality.....109
2. Lighting.....110
3. Shadows.....110
二、 Lighting 窗口.....110
1. Environment.....110
2. Other Settings.....110
三、 Navigation 窗口.....111
3. Agents（代理类型）.....111
四、 Nav Mesh Agent.....111
五、 协程与委托.....111
1. Coroutine 协同程序.....111
2. Delegate 委托.....111
六、 Event 事件.....112
七、 TextMesh Pro.....114
八、 富文本.....114
1. 标记格式.....114
2. 嵌套元素.....115
3. 标签参数.....115
九、 2D PSD 导入器.....115
1. 减少像素重叠.....116
2. 组合碰撞体.....117
3. 形状精灵烘焙.....118
4. 像素美化与锐化.....119
十、 粒子系统 Particle System.....120
1. 模拟阶段.....120
2. 渲染阶段.....121
十一、 脚本生命周期.....121
十二、 Animation.....123
1. 关于模型中的动画的位移变换问题.....123
十三、 自动布局组件.....123
1. Content Size Fitter.....123
十四、 后处理.....124
1. 为场景添加后处理效果的步骤.....124

哔哩哔哩次元游戏人生

2. 后处理效果应用顺序和效果组合 127
3. 辉光-Bloom 128
4. 色调映射-Tonemapping 128
5. 白平衡-White Balance 129
6. 胶片颗粒-Film Grain 129
7. 景深-Depth of Field 129
8. 帕尼尼投影-Panini Projection 130
9. 镜头畸变-Lens Distortion 130
10. 运动模糊-Motion Blur 131
11. 色差-Chromatic Aberration 131
12. 晕映-Vignette 131
13. 颜色分级-Lift Gamma Gain 132
14. 通道混合-Channel Mixer 132
15. 颜色曲线- Color Curves 133
16. 颜色调节-Color Adiustment 133
17. 分离调色-Split Toning 133
18. 阴影、中间调、高光-Shadow、Midtones、Highlights 134
十五、 Unity 反射 134
十六、 四元数 135
十七、 其他知识 135
1. 图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别？ 135
2. 使用 Unity3d 实现 2d 游戏，有几种方式？ 135
3. 向量的点乘、叉乘以及归一化的意义？ 135
十八、 .meta 文件 135
1. .meta 文件的创建 135
2. .meta 文件的作用 135
3. 常见的错误操作 135
Unity Script 136
一、 GameObject 136
1. 属性（Properties） 136
2. 构造函数（Constructors） 136
3. 公共方法 136
4. 静态方法 137
二、 Transform 137
1. 公共方法 137
三、 Vector3 138
1. 描述 138
2. Static Properity-静态属性 138
3. Static Methods-静态方法 138
4. 向量归一化的区别 139
四、 KeyCode 编码 140
五、 数学函数库 142
1. 绝对值-Mathf.Abs 142
2. 反余弦-Mathf.Acos 142

哔哩哔哩次元游戏人生

3. 近似-Mathf.Approximately....142
4. 反正弦-Mathf.Asin....142
5. 反正切-Mathf.Atan2....142
6. 反正切-Mathf.Atan....142
7. 最小整数- Mathf.CeilToInt....142
8. 上限值-Mathf.Ceil....142
9. 范围限制 0-1-Mathf.Clamp01....142
10. 限制- Mathf.Clamp....142
11. 最近的二次方- Mathf.ClosestPowerOfTwo....143
12. 余弦- Mathf.Cos....143
13. 度转弧度- Mathf.Deg2Rad....143
14. 弧度转度- Mathf.Rad2Deg....143
15. 增量角- Mathf.DeltaAngle....143
16. 小正数- Mathf.Epsilon....143
17. 指数- Mathf.Exp....144
18. 最小整数- Mathf.FloorToInt....144
19. 向下取整- Mathf.Floor....144
20. 正无穷- Mathf.Infinity....144
21. 反差值- Mathf.InverseLerp....144
22. 是否为 2 的幂- Mathf.IsPowerOfTwo....144
23. 插值角度- Mathf.LerpAngle....144
24. 插值- Mathf.Lerp....144
25. 基数为 10 的对数- Mathf.Log10....144
26. 对数- Mathf.Log....144
27. 最大值- Mathf.Max....145
28. 最小值- Mathf.Min....145
29. 移动角- Mathf.MoveTowardsAngle....145
30. 移向- Mathf.MoveTowards....145
31. 负无穷- Mathf.NegativeInfinity....145
32. 下一个 2 的幂- Mathf.NextPowerOfTwo....145
33. 乒乓- Mathf.PingPong....145
34. 圆周率- Mathf.PI....145
35. 次方- Mathf.Pow....145
36. 重复- Mathf.Repeat....145
37. 四舍五入到整数- Mathf.RoundToInt....146
38. 四舍五入- Mathf.Round....146
39. 有向- Mathf.Sign....146
40. 正弦- Mathf.Sin....146
41. 平方根- Mathf.Sqrt....146
42. 正切- Mathf.Tan....146
43. 平滑阻尼角度- Mathf.SmoothDampAngle....146
44. 平滑阻尼- Mathf.SmoothDamp....146
45. 平滑插值- Mathf.SmoothStep....146
六、 UGUI 事件接口....146

哔哩哔哩次元游戏人生

1. IPointEnterHandler 146
2. IPointExitHandler 147
3. IPointDownHandler 147
4. IPointUpHandler 147
5. IPointClickHandler 147
6. IDragHandler 147
7. IInitializePotentialDragHandler 147
8. IBeginDragHandler 147
9. IEndDragHanlder 147
10. IDropHandler 147
Unity 案例 148
一、 2D 角色移动 148
二、 3D 角色移动 148
三、 3D 相机控制 148
四、 输入管理-InputManager 148
五、 场景异步加载与过渡 149
六、 自定义 URP 后处理 149
七、 通用对象池 149
1. 概述 149
2. 理解 149
3. 优点 150
4. 缺点 150
5. 简单对象池 150
6. 对象池管理 155
八、 Unity 脚部 IK 159
1. 构建测试场景 159
2. 创建 Animator 并设置所需参数 159
3. 创建脚部 IK 161
4. 设置动画 IK 权重 163
5. 可能出现的问题 165
技巧 166
一、 修改 Unity 缓存位置 166
1. 修改 Unity GICache 166
2. 修改 Unity 的 Cache 缓存位置 166
3. 修改 Unity Asset Store 的默认下载路径 169
二、 优化 169
1. 全平台优化方案 169
2. 脚本性能优化 171
3. 物理性能优化 171
4. Shader 优化方案 171
三、 快捷键 175
1. Unity 175
2. Visual Studio 175
四、 脚本栏的扩展 175

哔哩哔哩次元游戏人生

1. 序列化[Serializable] 175
2. 强制序列化[SerializeField] 175
3. 属性标题[Header] 176
4. 属性提示[Tooltip] 176
5. 空行属性[Space] 176
6. 范围值属性[Range()] 176
7. 隐藏属性[HideInInSpector] 176
8. 输入域[TextArea] 176
9. [RequireComponent(typeof(xxx))] 176
10. [CreateAssetMenu(fileName = "xx", menuName = "xx", order = x)] 176
11. 永远执行-[ExecuteAlways] 176
12. 禁止多组件-[DisallowMultipleComponent] 176
13. 添加组件菜单-[AddComponentMenu] 176
14. 在编辑模式下执行-[ExecuteInEditMode] 176
五、 VisualStudio 176
六、 修改 unity 模板 177
七、 资源网站 177
八、 切片图集(Slice Atlas) 177
1. 下载 177
2. 切片类型 179
九、 未运行游戏的状态监视 ShaderGraph 180
十、 组件 Preset 180
十一、 调整运行窗口颜色 181
十二、 修改鼠标样式和启动图像 182
1. 设置鼠标样式 182
2. 设置启动图像 183
十三、 图片跟随文本自适应 185
Unity Editor 187
一、 Editor 文件夹 187
二、 Editor Default Resources 187
三、 Gizmos 文件夹 187
Unity Editor 实践 188
Unity 工具 190
一、 Terrain Tool 190
3. 快捷键 190
二、 Cinemachine 190
三、 Recorder 190
4. 下载 190
5. 打开 Recorder 界面 191
6. Recording Mode(记录模式) 191
7. Frame Rate 帧率属性 191
8. Capture-捕捉设置 193
四、 Post Processing 194
1. 开始后处理 194

哔哩哔哩次元游戏人生

2. 泛光-Bloom.....195
设计模式.....196
一、 单例模式.....196
1. Unity 泛型单例.....196
二、 观察者模式.....197
数据结构.....199
一、 基本概率和术语.....199
1. 数据结构三要素.....199
数据结构习题.....200
一、 数组.....200
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。.....200
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。201
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）。.201
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。.....203
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。.....204
二、 计算机组成原理.....204
1. 写出代码判断一个整数是不是 2 的 N 次方。.....204
2. 写出下面的程序的打印结果，并解释其目的。.....205
3. 写出下面的程序的打印结果，并解释其目的。.....206
三、 递归.....206
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)。.....206
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。.....206
算法- Algorithm.....208
一、 洗牌算法.....208
1. Fisher–Yates shuffle.....208
2. Knuth-Durstenfeld Shuffle.....208
二、 排序算法.....215
1. 冒泡排序-Bubble Sort.....215
计算机组成原理.....216
一、 常用进位计数制.....216
1. 二进制.....216
2. 八进制.....216
3. 十进制.....216

陈恒捷电子游戏人生

4. 十六进制.....216
5. 推广.....216
二、 各种进制数之间的转换.....216
1. 二进制数转换为十六进制数.....216
2. 十六进制数转换为二进制数.....217
3. 二进制数转换为十进制数.....217
4. 十进制数转换为二进制数.....217
三、 数的机器码表示.....217
1. 机器数与真值.....217
四、 位运算符.....219
1. 与 - &.....219
2. 或 - |.....220
3. 异或 - ^.....220
4. 取反 - ~.....221
5. 左移 - <<.....221
6. 右移 - >>.....221
物理.....222
一、 牛顿第二定律.....222
二、 抛体运动.....222
线性代数.....224
一、 向量.....224
二、 行列式.....224
1. 二阶行列式.....224
2. 三阶行列式.....224
3. n 阶行列式.....224
4. 性质.....224
5. 特殊的行列式.....225
三、 全排列和对换.....225
1. 全排列（排列）.....225
2. 逆序.....225
3. 对换.....225
四、 点和向量.....226
1. 具体来讲.....226
2. 自由向量.....226
五、 向量在轴上的投影.....226
六、 数量积、向量积.....227
1. 数量积（点乘）.....227
2. 向量积（叉乘）.....227
七、 向量的内积（点乘）.....228
1. 内积性质.....228
八、 向量的范数（长度）.....228
1. 性质.....228
概率论与数理统计.....230
初等数学.....231

哔哩哔哩次元游戏人生

高等数学.....232
一、 导数.....232
数据库.....233
Computer Graphics.....234
一、 贝塞尔曲线路径.....234
1. 贝塞尔曲线.....234
2. 一次 Bezier 曲线.....234
3. 二次 Bezier 曲线.....234
4. 三次 Bezier 曲线.....235
5. 高阶 Bezier 曲线.....235
二、 程序化网格生成.....236
1. 渲染物体.....236
2. UV.....236
3. 创建网格.....237
4. 生成法线.....238
5. 生成 UV 坐标.....238
Shader Graph.....239
一、 菲涅尔效果-Fresnel Effect Node.....239
1. 端口.....239
2. 生成的代码示例.....239
二、 乘节点-Multiply Node.....239
1. 端口.....239
2. 生成的代码示例.....239
三、 加法节点- Add Node.....240
1. 端口.....240
2. 生成的代码示例.....240
四、 除法-Divide Node.....240
1. 端口.....240
2. 生成的代码示例.....240
五、 步节点-Step Node.....240
1. 端口.....241
2. 生成的代码示例.....241
六、 平滑步节点-Smoothstep Node.....241
1. 端口.....241
2. 生成的代码示例.....241
七、 极坐标-Polar Coordinates Node.....242
1. 端口.....242
2. 生成的代码示例.....242
八、 重新映射-Remap Node.....242
1. 端口.....243
2. 生成的代码示例.....243
九、 限制-Clamp Node.....243
1. 端口.....243
2. 生成的代码示例.....243

哔哩哔哩次元游戏人生

十、 时间-Time Node 243
1. 端口....243
2. 生成的代码示例....244
十一、 瓦片与偏移-Tiling And Offset Node....244
1. 端口....244
2. 生成的代码示例....245
十二、 取反-Neagte Node....245
1. 端口....245
2. 生成的代码示例....245
十三、 简单噪声-Simple Noise Node 245
1. 端口....245
2. 控件....245
3. 生成的代码示例....245
十四、 旋转-Rotate Node 247
1. 端口....247
2. 控件....247
3. 生成的代码示例....247
十五、 幂节点-Power Node....248
1. 端口....248
2. 生成的代码示例....248
十六、 饱和节点-Saturate Node....248
1. 端口....248
2. 生成的代码示例....248
十七、 线性插值-Lerp Node 248
1. 端口....249
2. 生成的代码示例....249
十八、 反插值- Inverse Lerp Node....249
1. 端口....250
2. 生成的代码示例....250
十九、 翻书-Flipbook Node 250
1. 端口....251
2. 控件....251
3. 生成的代码示例....252
二十、 向下取整- Floor Node 252
1. 端口....252
2. 生成的代码示例....252
二十一、 取余- Modulo Node 253
1. 端口 0....253
2. 生成的代码示例....253
二十二、 场景深度-Scene Depth Node....253
1. 端口....253
2. 深度采样模式....253
3. 生成的代码示例....253
二十三、 One Minus Node 253

哔哩哔哩次元游戏人生

1. 端口.....253
2. 生成的代码示例.....254
二十四、 位置-Position Node.....254
1. 端口.....254
2. 控件.....254
二十五、 绝对值-Absolute Node.....254
1. 端口.....254
2. 生成的代码示例.....254
二十六、 变换-Transform Node.....254
1. 端口.....254
2. 控件.....255
3. World and Absolute World.....255
4. 生成的代码示例.....255
二十七、 标准化-Normalize Node.....257
1. 端口.....257
2. 生成的代码示例.....257
二十八、 长度-Length Node.....257
1. 端口.....257
2. 生成的代码示例.....257
二十九、 距离-Distance Node.....257
1. 端口.....257
2. 生成的代码示例.....258
三十、 点乘-Dot Product Node.....258
1. 端口.....258
2. 生成的代码示例.....258
三十一、 小数-Fraction Node.....258
1. 端口.....258
2. 生成的代码示例.....258
三十二、 随机范围-Random Range Node.....258
1. 端口.....258
2. 生成的代码示例.....258
三十三、 比较-Comparison Node.....259
1. 端口.....259
2. 控件.....259
3. 生成的代码示例.....259
三十四、 快捷键.....260
ShaderGraph 实际案例.....261
一、 2D 简易描边效果-减法版.....261
1. 创建 Shader Graph.....261
2. 创建描边轮廓.....262
3. 修改描边轮廓颜色.....262
4. 得到最终成像.....263
二、 2D 描边效果-加法版.....263
1. 创建 Shader Graph.....263

哔哩哔哩次元游戏人生

2. 放大原图像遮罩.....264
3. 获得描边.....265
4. 修改描边颜色.....266
5. 得到最终成像.....266
6. 可能出现的问题.....267
三、 2D UI 技能冷却缓动.....267
1. 创建 Shader Graph.....267
2. 创建切分效果.....268
3. 重新映射.....269
4. 扩展-旋转 UV.....270
5. 扩展-极坐标.....270
6. 在代码中控制.....271
四、 2D 图像 RGB 分离.....273
1. 创建 Shader Graph.....273
2. 添加可外部访问的参数.....274
3. 添加节点.....274
五、 2D 全息效果.....276
1. 创建 Shader Graph.....276
2. 添加可外部访问的参数.....277
3. 创建带全息效果的图像.....278
4. 让全息纹理流动起来.....278
5. 扩展-调节全息贴图显示范围.....278
六、 2D 溶解效果.....279
1. 创建 Shader Graph.....279
2. 添加可外部访问的参数.....280
3. 创建溶解效果.....281
4. 添加溶解边缘颜色.....281
七、 2D 流光效果简易版.....282
1. 创建 Shader Graph.....282
2. 添加可外部访问的参数.....283
3. 显示含有流光效果的图像.....283
4. 让流光纹理流动起来.....284
5. 反向运动.....285
6. 旋转流光.....285
7. 可能出现的问题.....285
八、 2D 流光效果进阶版.....286
1. 创建 Shader Graph.....286
九、 转场过渡效果-简易版.....287
1. 创建 Shader Graph.....287
2. 添加可外部访问的参数.....288
3. 实现硬性过渡.....289
4. 将硬性过渡更改为柔性过渡.....289
十、 2D 激光效果.....290
十一、 2D 帧动画.....290

哔哩哔哩次元游戏人生

1. 创建 Shader Graph.....290
2. 添加主纹理.....291
3. 分割主纹理.....292
4. 让图集动起来.....292
5. 设置图集的最大可显示帧.....293
6. 调整帧动画速度.....295
7. 添加额外参数.....295
8. 可能存在的问题.....295
十二、 后处理速度线.....296
1. 创建 Shader Graph.....296
2. 制作速度线.....297
3. 让速度线流动起来.....298
4. 调整速度线的 Scale 和 Tiling.....298
5. 制作蒙版.....299
6. 最终输出.....299
7. 将节点转换为参数.....300
8. 后处理运用.....300
十三、 2D 像素效果.....301
1. 创建 Shader Graph.....301
2. 原理解释.....302
3. 修改图像对应的 UV 坐标.....302
4. 使用 Slider 控制像素化效果.....303
5. 设置是否应用像素化.....305
6. 自定义像素化行列（拓展）.....305
十四、 2D 植被飘动.....306
1. 创建 Shader Graph.....306
2. 原理.....307
3. 将主纹理输出到片元着色器.....307
4. 让顶点位置发生偏移.....307
5. 使用噪声调整偏移强度.....309
6. 对顶点应用噪声偏移.....309
7. 修改噪声视觉效果.....310
8. 创建影响蒙版.....310
十五、 2D 简单故障效果.....311
1. 创建 Shader Graph.....311
2. 创建色差效果.....312
3. 创建故障纹理.....314
4. 调节故障纹理密度.....316
5. 设置故障偏移方向.....316
6. 闪烁效果.....317
7. 完成最终效果.....317
十六、 2DSDF 圆.....323
1. 圆心在原点.....324
2. 圆心不在原点上.....324

哔哩哔哩次元游戏人生

十七、 2DSDF 线段.....325
1. 位于区域一.....326
2. 位于区域二和三.....327
3. 计算端点向量和到目标点的向量.....328
4. 计算 r.....328
5. 计算aq向量.....329
6. 计算点距离线段的长度.....329
十八、 矩形-Rectangle.....330
1. 矩形中心在原点.....330
2. 矩形中心不在原点.....332
3. 创建所需参数.....332
4. 计算向量ap1.....332
5. 计算目标点位于区域一、二、三时的模.....332
6. 计算区域四点的最短距离.....333
7. 排除目标点不位于区域四的情况.....333
8. 计算最终结果.....334
9. 缺点.....335
十九、 3D 菲尼尔边缘光.....335
1. 什么是菲尼尔效应.....335
2. 创建 LitShaderGraph.....340
二十、 3D 法线膨胀描边效果.....344
二十一、 3D 积雪效果.....344
二十二、 3D 广告牌效果(Billboard).....344
Shader Lab.....345
一、 Shader 概念.....345
1. 什么是 Shader.....345
2. Shader 和 Material（材质）的关系与区别.....345
3. Unity 中的 Shader.....345
二、 简单概念.....346
1. 顶点-Vertices.....346
2. 法线-Normals.....346
3. 切线-Tangents.....347
4. UV 坐标-UV coordinates.....347
三、 纹理.....348
1. 金属纹理-Metallic Texture.....349
2. 法线贴图（凹凸贴图）-Normal Map(Bump Map).....349
3. 高度图-Height Map.....357
4. 遮蔽贴图- Occlusion Map.....358
四、 渲染流水线.....359
1. 什么是渲染流水线.....359
2. 应用阶段(Application Stage).....359
3. 几何阶段(Geometry Stage).....360
4. 光栅化阶段(Rasterizer Stage).....360
5. 像素处理阶段(Pixel processing stage).....360

哔哩哔哩次元游戏人生

五、 CPU 和 GPU 之间的通信 361
1. 把数据加载到显存中 361
2. 设置渲染状态 361
3. 调用 Draw Call 361
六、 GPU 流水线 362
1. 顶点着色器 363
2. 裁剪 364
3. 屏幕映射 364
4. 三角形设置 365
5. 三角形遍历 366
6. 片元着色器 366
7. 逐片元操作 367
七、 什么是 Draw Call 372
1. CPU 和 GPU 是如何实现并行工作的？ 372
2. 为什么 Draw Call 多了会影响帧率？ 373
3. 如何减少 Draw Call？ 373
八、 语法基础 374
1. ShaderLab 的组织结构 374
2. 详细解释 388
九、 顶点和片元着色器基础 403
1. CGPROGRAM/ENDCG 403
2. 数据类型 404
3. 编译指令 406
4. 语义 410
5. 结构体 413
6. Unity 内置文件和变量 414
十、 其他 420
1. 着色器和材质 420
2. 调试着色器 420
3. 添加 URP 兼容 422
十一、 Unity 中的基础光照 424
十二、 有向距离函数-SDF-Sign Distance Function 424
1. 圆-Circle 424
2. 线段-LineSegment 426
3. 矩形-Rectangle 429
4. 圆角矩形- Rounded Rectangle 430
5. SDF-菱形(rhombus) 434
十三、 伪随机噪声-Pseudo Random Noise 435
1. Hashing 435
Visual Effect Graph 438
一、 篝火 438
PhotoShop 439
一、 去除水印 439

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、 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。

四、 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。

1. 虚函数

2. 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。

3. 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。

五、 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。

六、 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

哈理工游戏人生

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。

七、 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()
    {
```

哔哩哔哩次元游戏人生

```
        }
    }
```

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。

八、 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 。

1） 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。

九、 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。

唯唯修理次元游戏人生

可改善程序的可维护性。
能够更好地支持大型程序设计。
增强了计算机处理信息的范围。

十、 C#索引

十一、 运算符重载

十二、 值类型和引用类型

十三、 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。

哔哩哔哩欢乐游戏人生

```
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。

悟空的里发元游戏人生

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
```


哔哩哔哩次元游戏人生

```
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。

哔哩哔哩次元游戏人生

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

博理学理学无游戏人生

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

降里降降次元游戏人生

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")

"*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
```


哔哩哔哩次元游戏人生

```
    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 ?
```

咕哩咔哩次元游戏人生

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})
```


哔哩哔哩次元游戏人生

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、 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
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

哔哩哔哩次元游戏人生

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

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

哔哩哔哩次元游戏人生

Unity

一、 PipelineAsset

1. Quality

4) 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）

5） 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

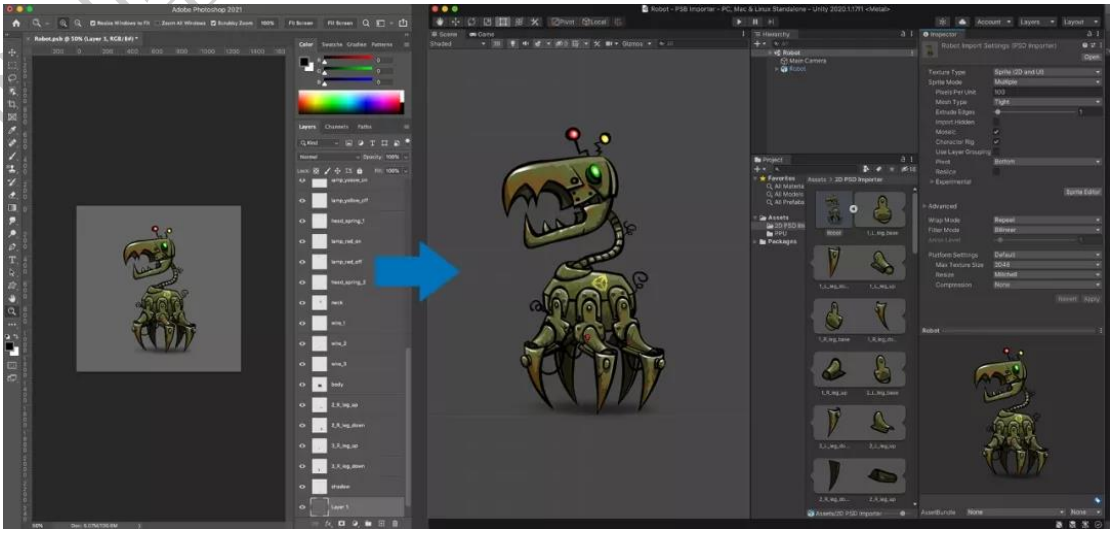


图7. 图 8.1 PSD 导入示例

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

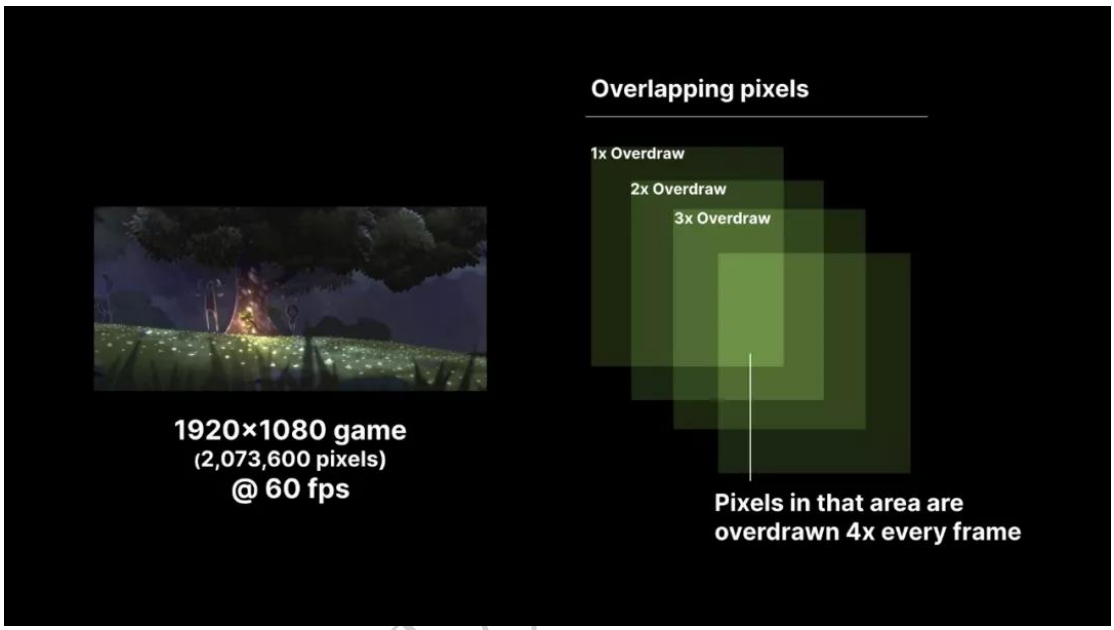


图8. 重叠像素优化示例

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。

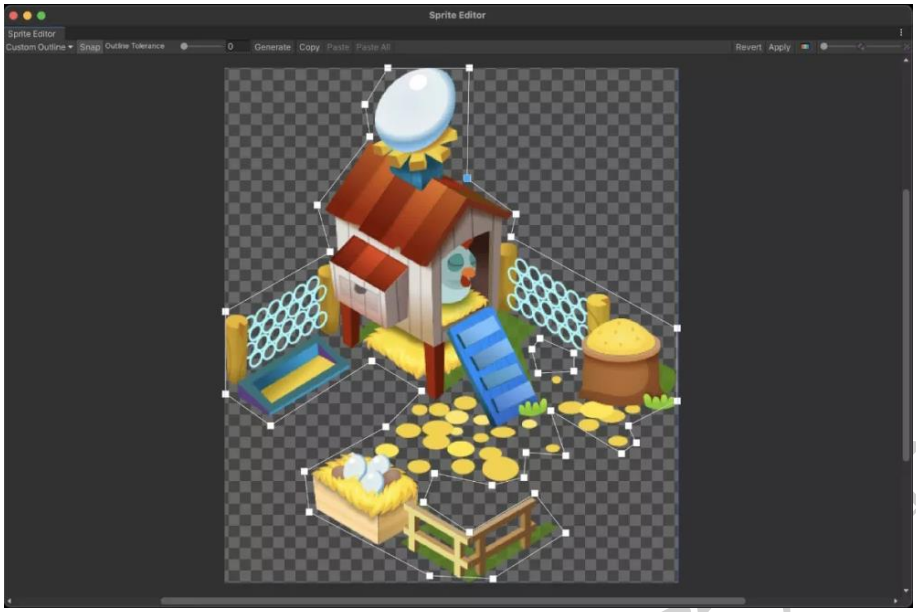


图9. 图 8.3 合并重叠的静态精灵.

2. 组合碰撞体

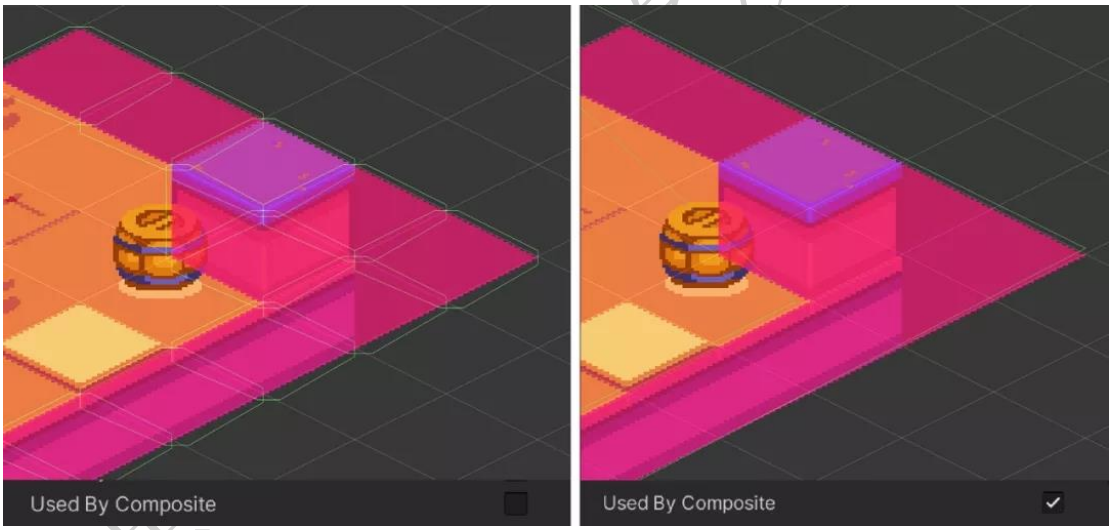


图10. 组合碰撞体的关闭和开启

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

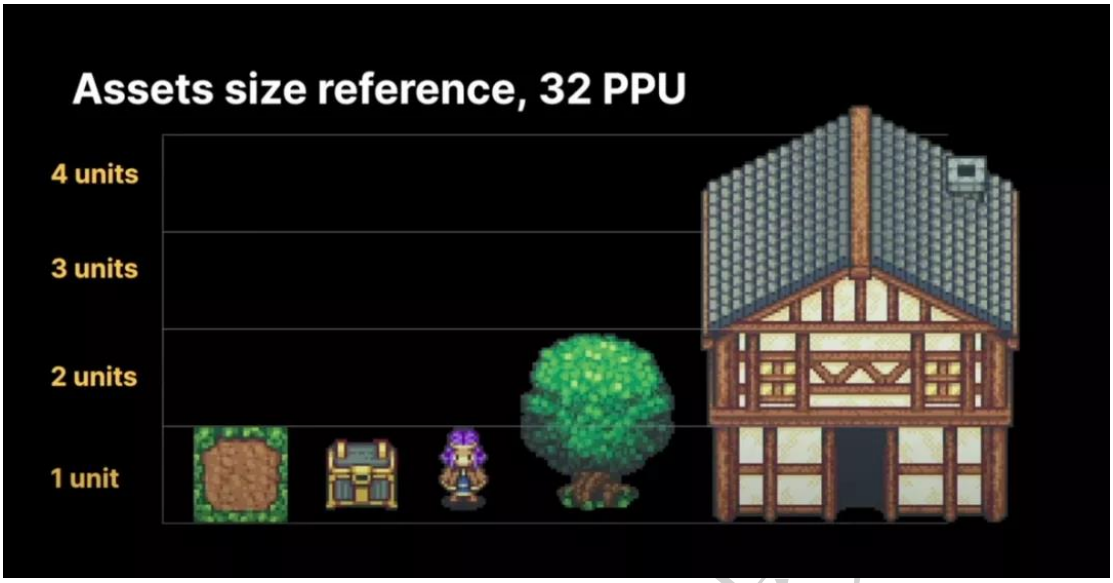


图12. 资源尺寸参考——32 像素每单位（PPU）

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：

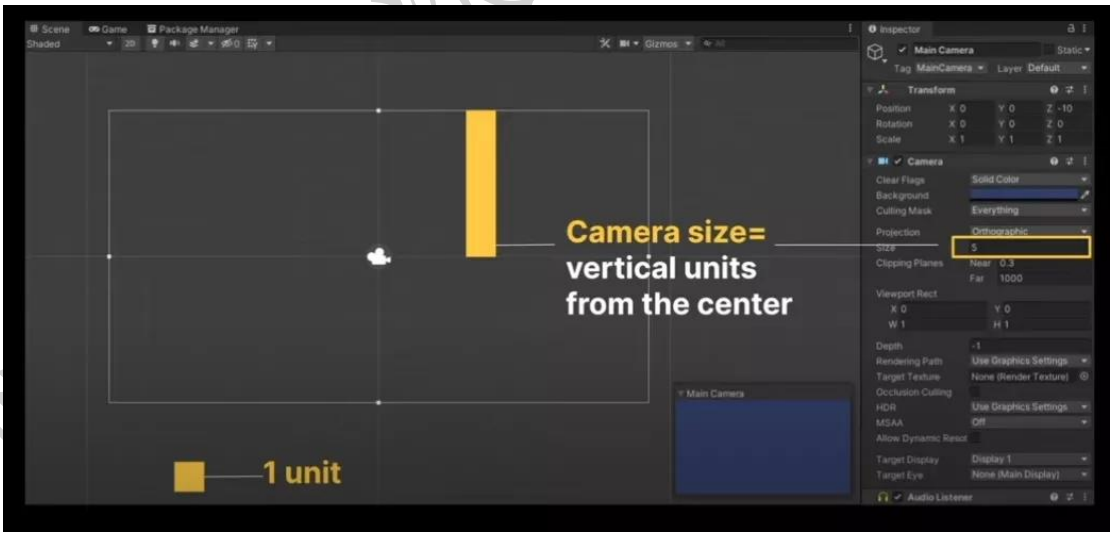


图13. 图 8.7 图像尺寸描述示例

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。

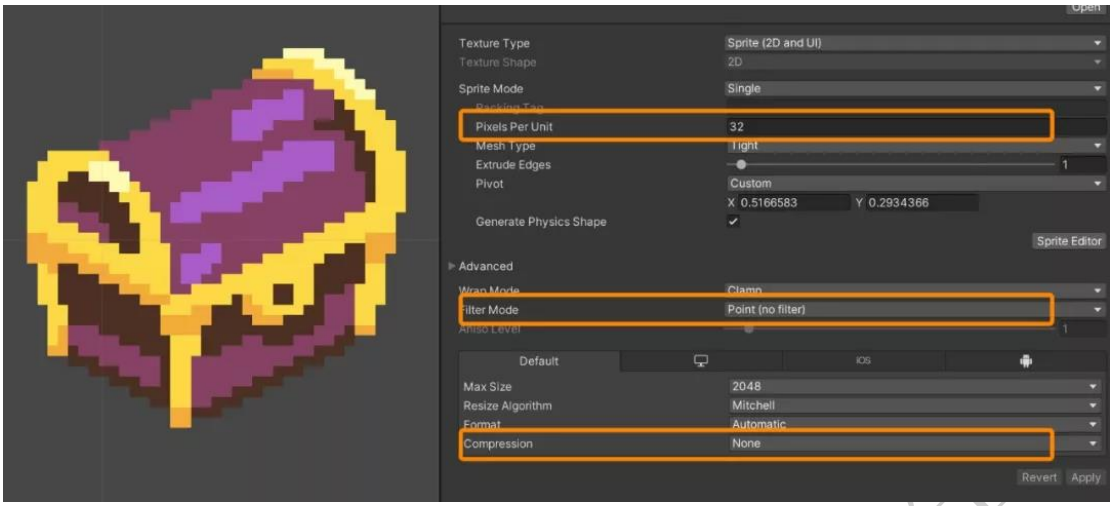


图14.　　图 8.8 精灵图设置

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。

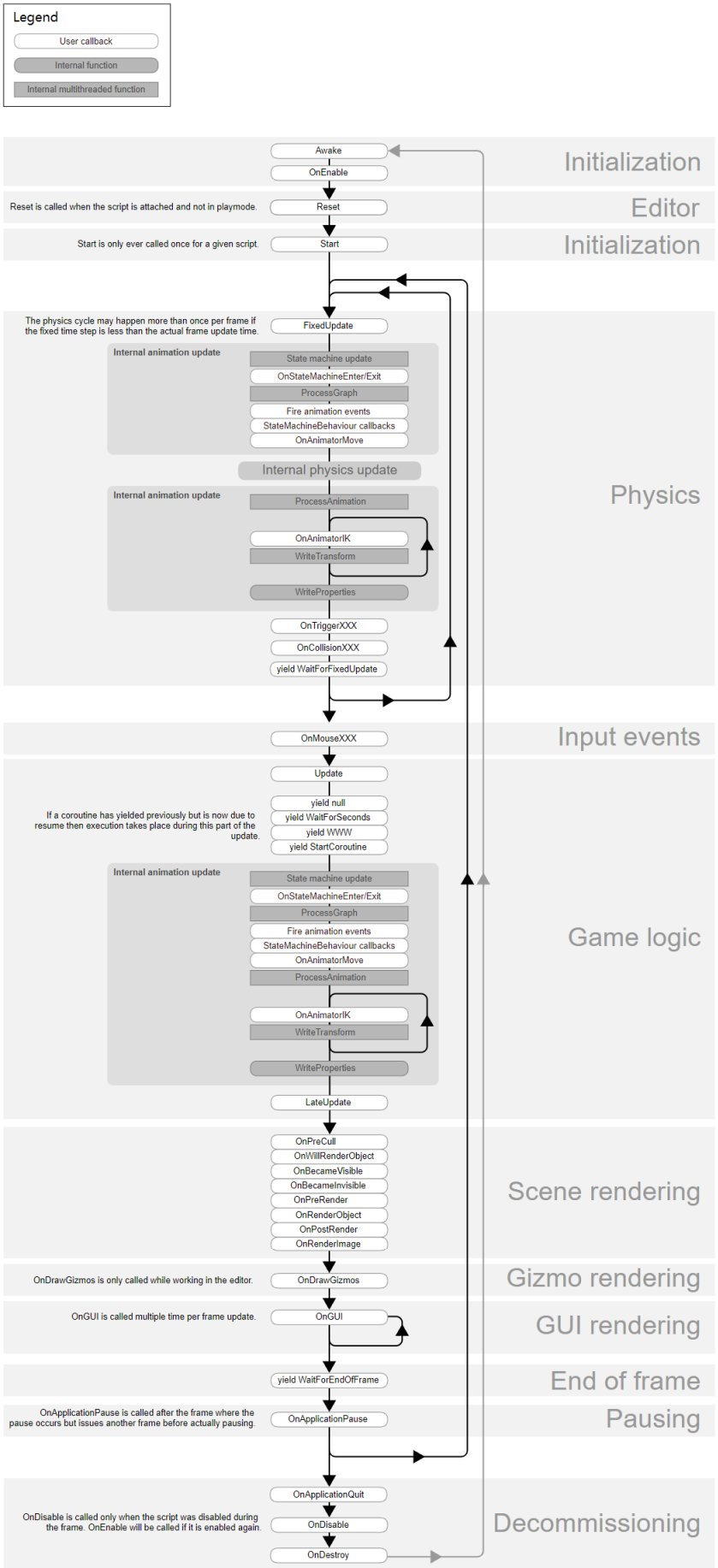
2. 渲染阶段

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。

哔哩哔哩沙无游戏人生

[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

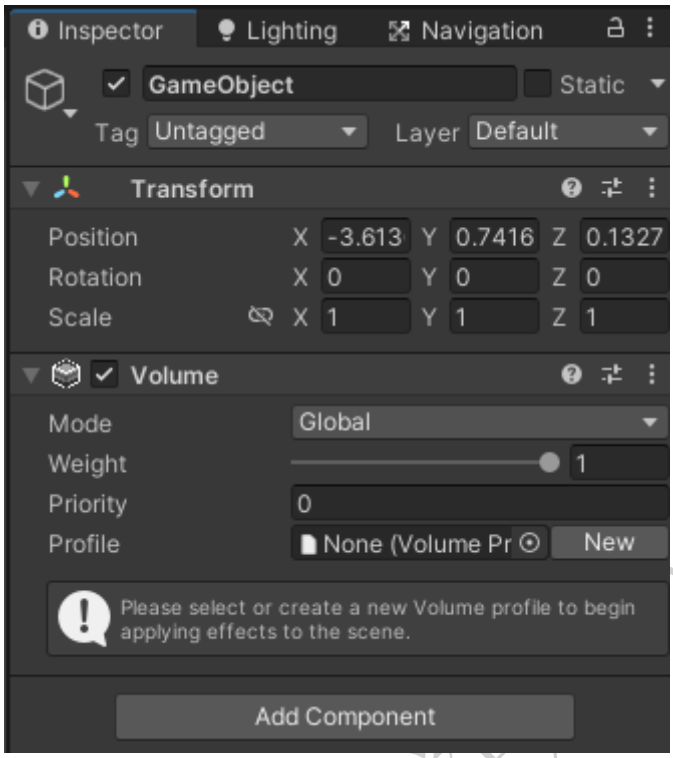


图15.　　图 13.1 为对象添加 Volume 组件

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

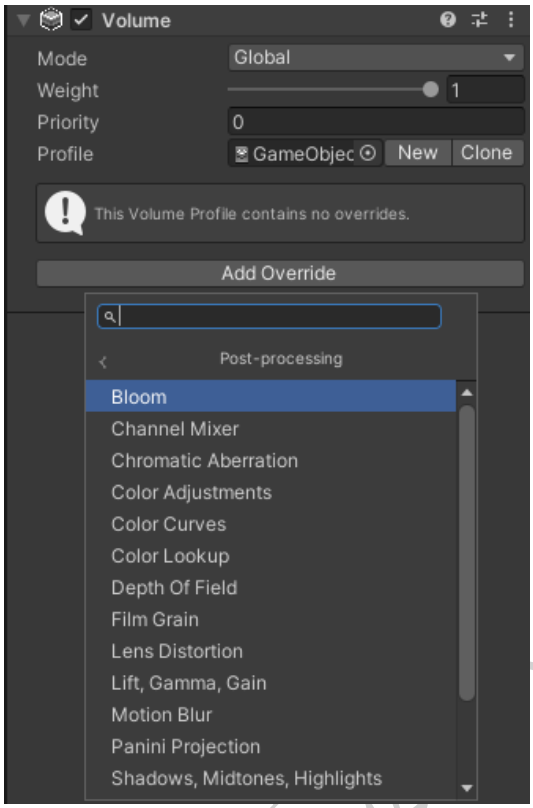


图16.　　图 13.2 使用 Override 添加后处理效果

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

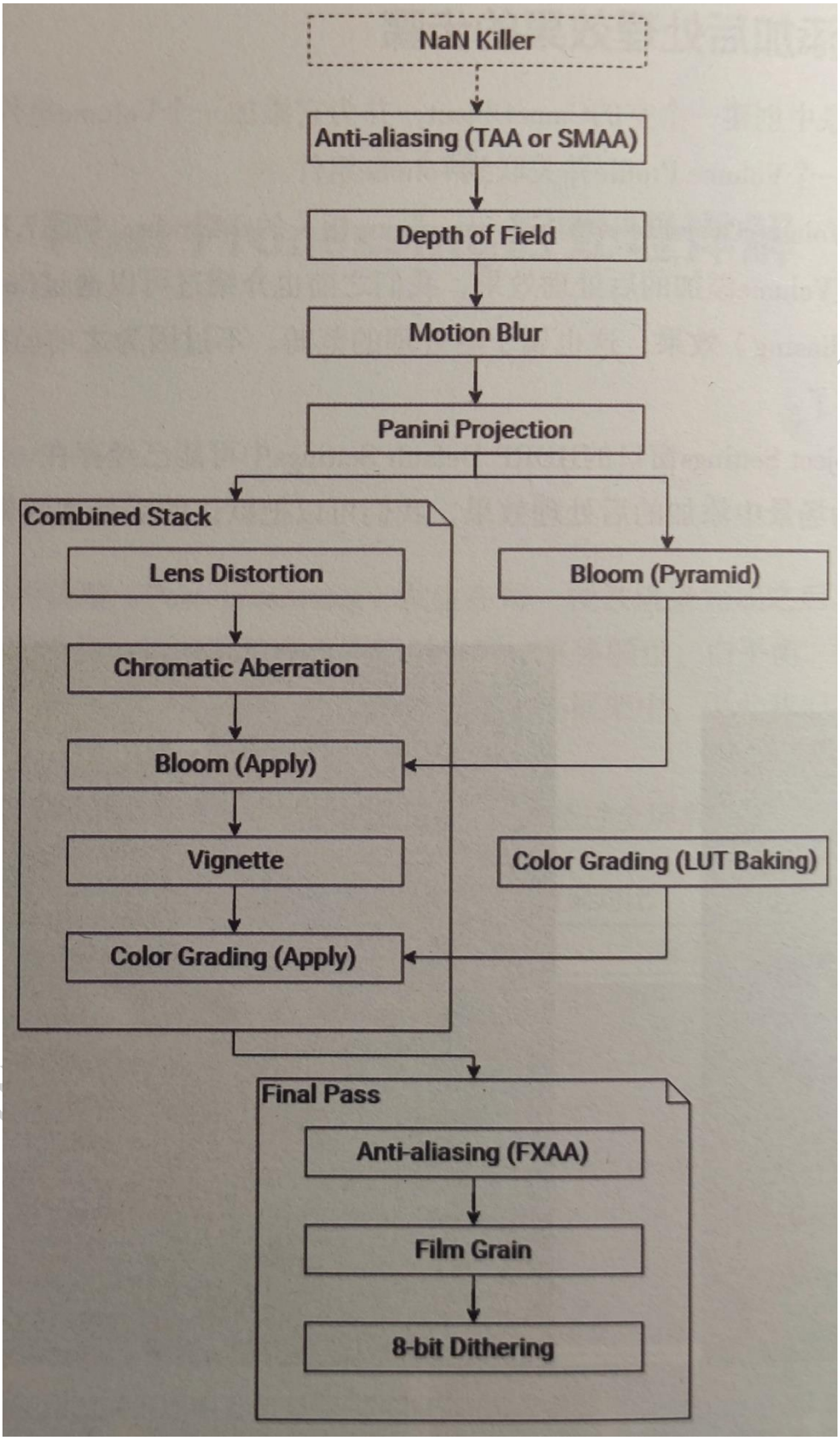


图17.　　图 13.3 HDRP 渲染管线应用后处理效果的顺序

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。

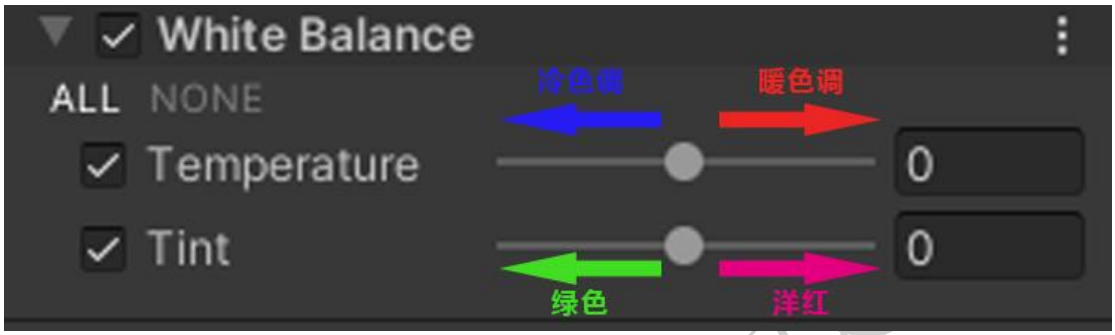


图18.　　白平衡

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。

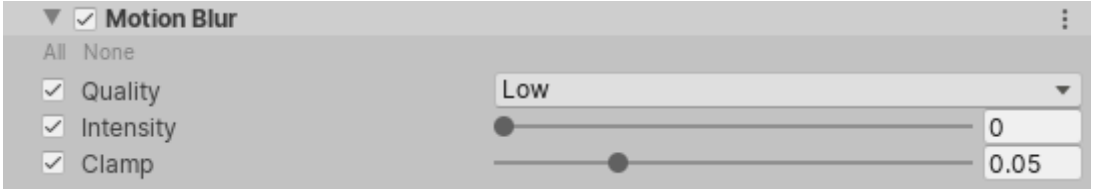


图19. Motion Blur

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。

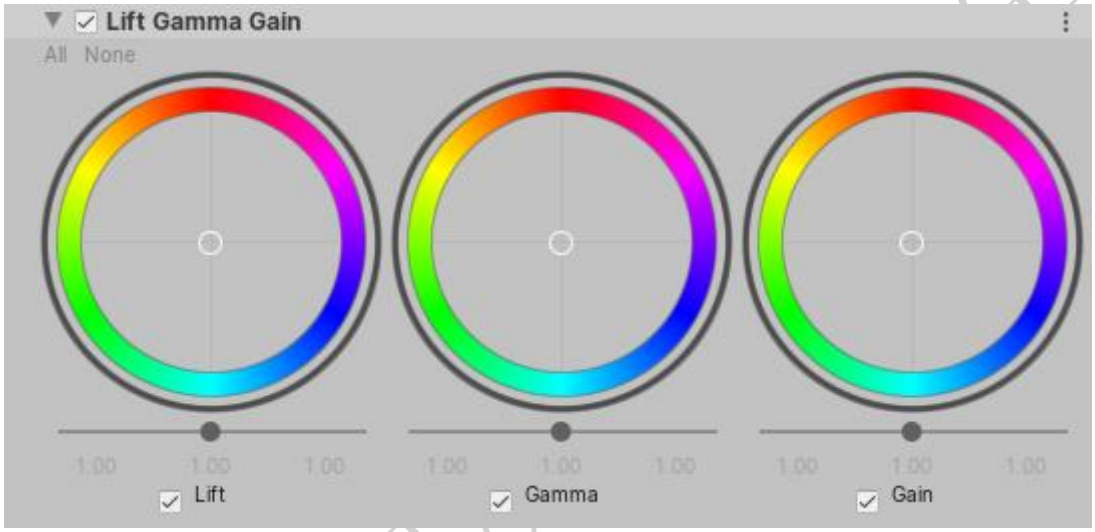


图20. LiftGammaGain

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 作用模型贴图使用。

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

哔哩哔哩尔元游戏人生

			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. 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
    {
```

陈理陈理次元游戏人生

```
            return false;
        }
    }
```

17. 指数- Mathf.Exp

public static float Exp(float power);

反回 e 的 Power 次方的值。

18. 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. IPointEnterHandler

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

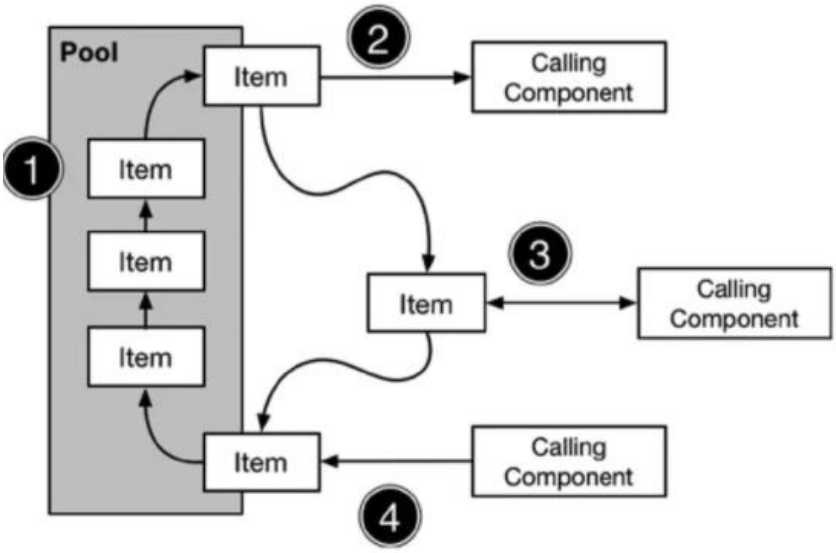


图22.　　对象池

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。

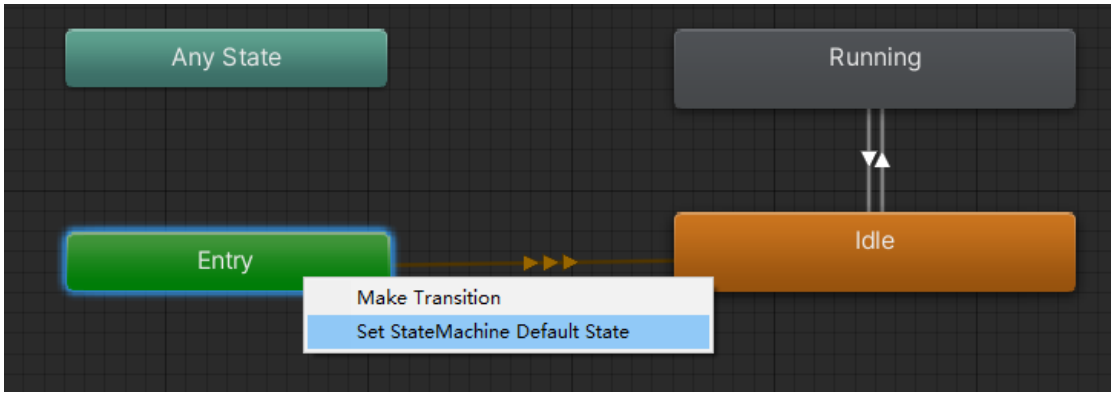


图24.　　设置状态机的默认状态

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。

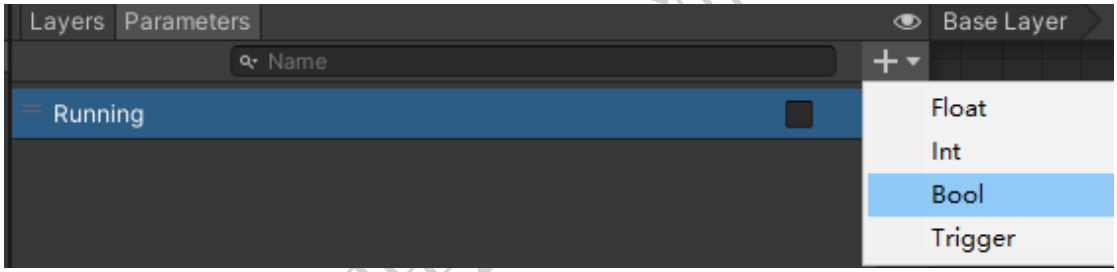


图25.　　创建动画参数

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。

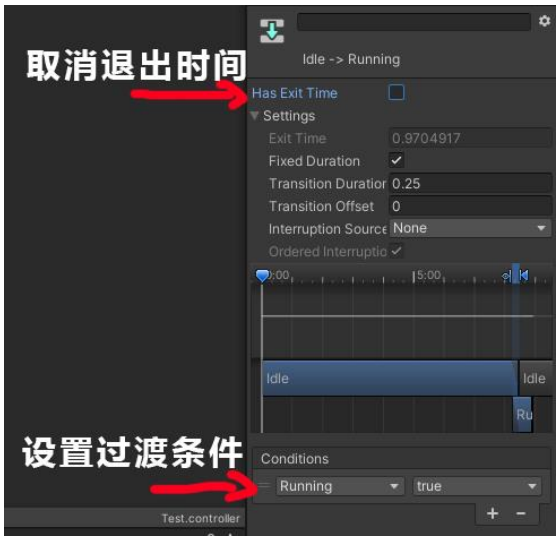


图26.　　设置过渡条件

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);
```

哔哩哔哩欢乐游戏人生

```
        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);
```

```
                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。

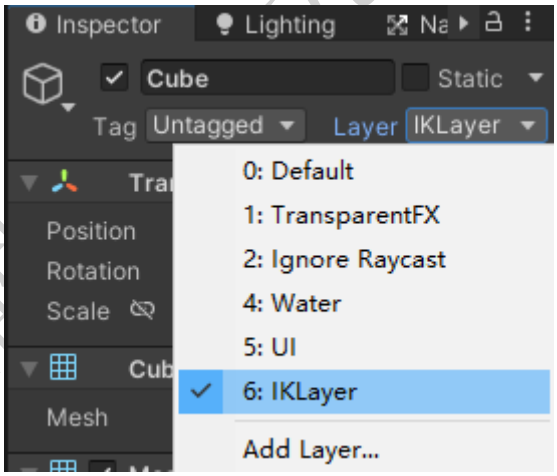


图27. 设置对象的 Layer

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。

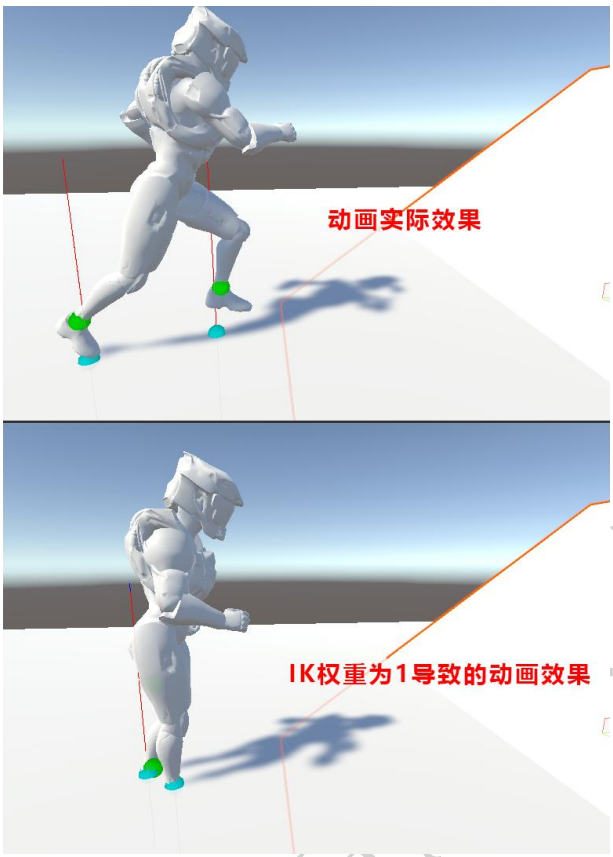


图28.　　权重对于动画的影响

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。

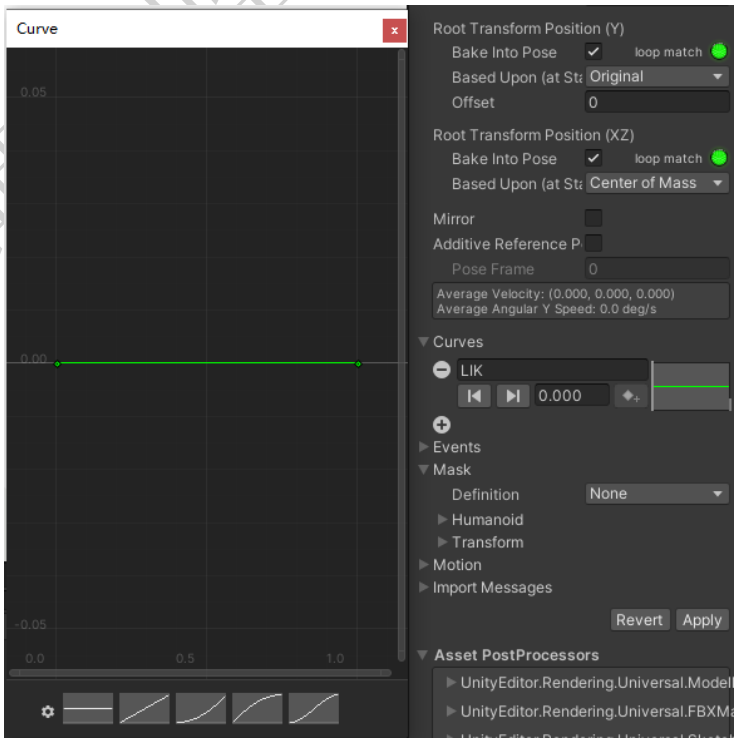


图29.　　设置动画曲线

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。

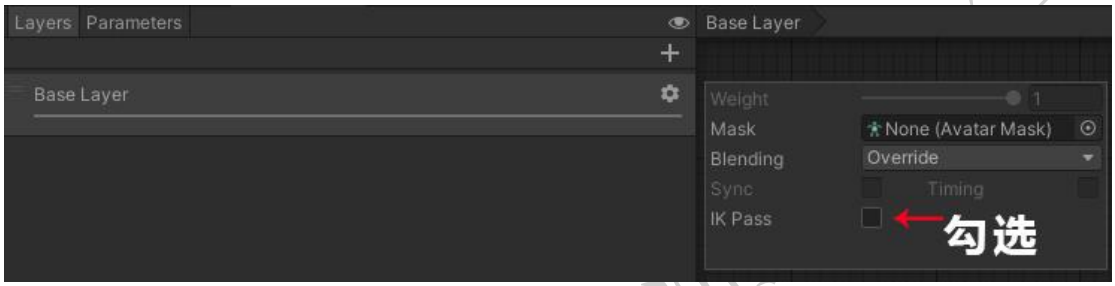


图30.　　检查 IK Pass 是否勾选

哔哩哔哩次元

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。

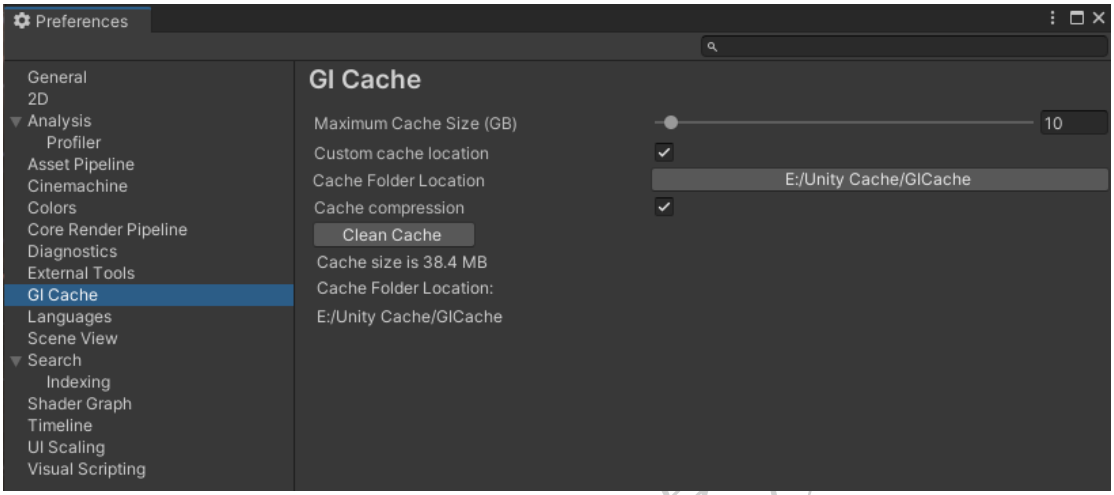


图31. 图 1.1 示意图

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32. 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33. 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34. 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35. 图 1.5 找到我的电脑的属性选项

② 打开高级系统设置

图36. 图 1.6 找到高级系统设置

③ 打开环境变量

图37.　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38.　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

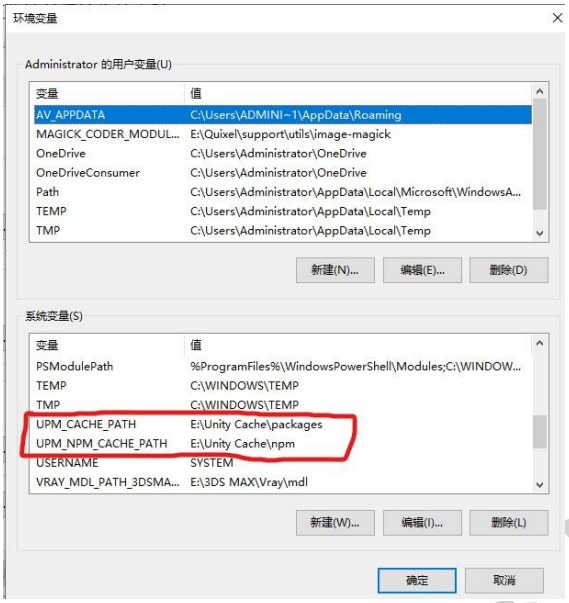


图39.　　图 1.8 新建环境变量

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40.　　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

1）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。

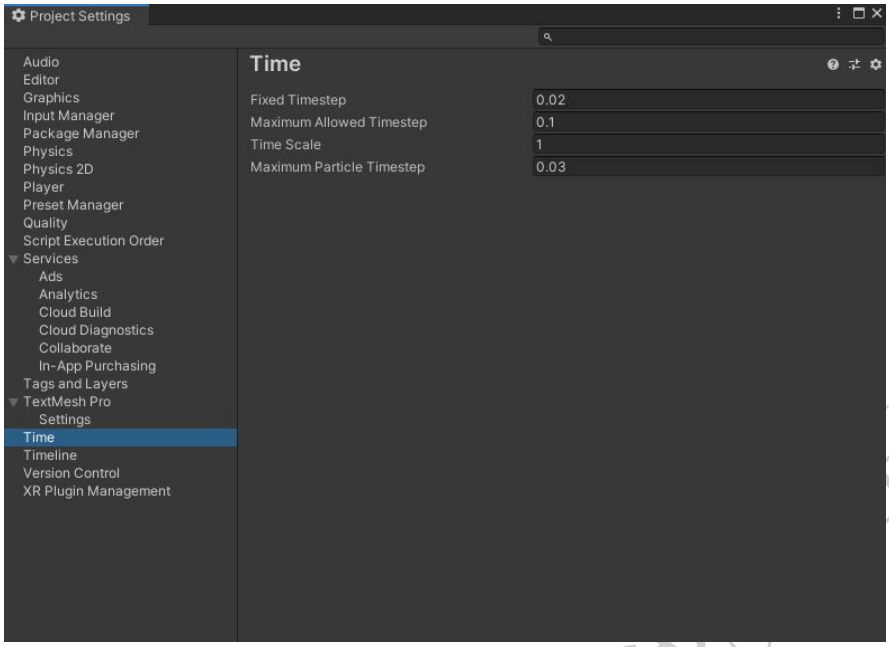


图41.　　图 3.1 TimeManager 属性面板

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。

2） 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。

3） 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。

4） 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。

5） 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

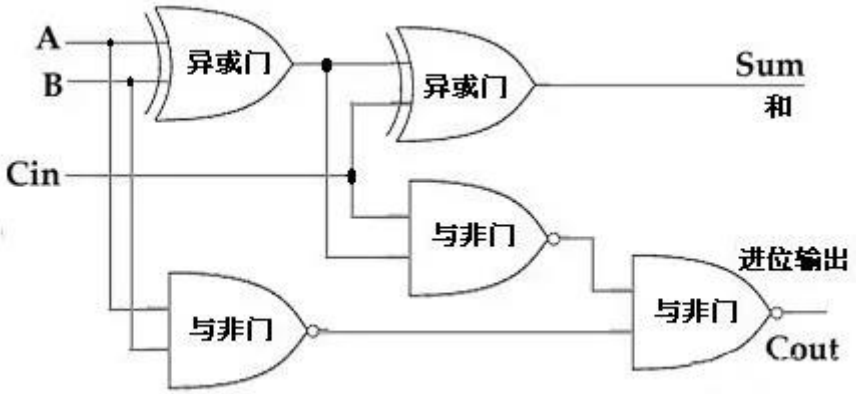


图42. 图 6.2 全加器逻辑电路

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

哔哩哔哩你不游戏人生

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。

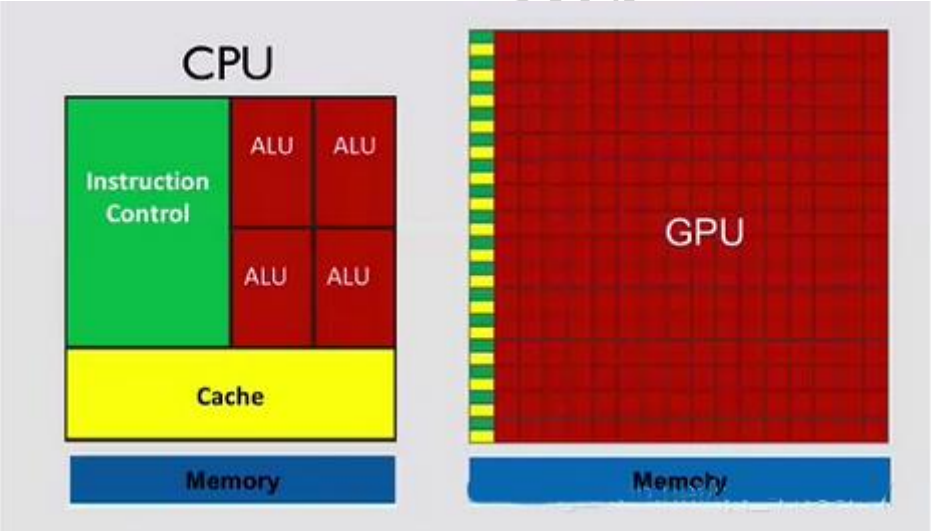


图43.　　图 6.3 架构图

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

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。

哔哩哔哩次元游戏人生

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，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）……

哔哩哔哩@无游戏人生

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、 修改 unity 模板

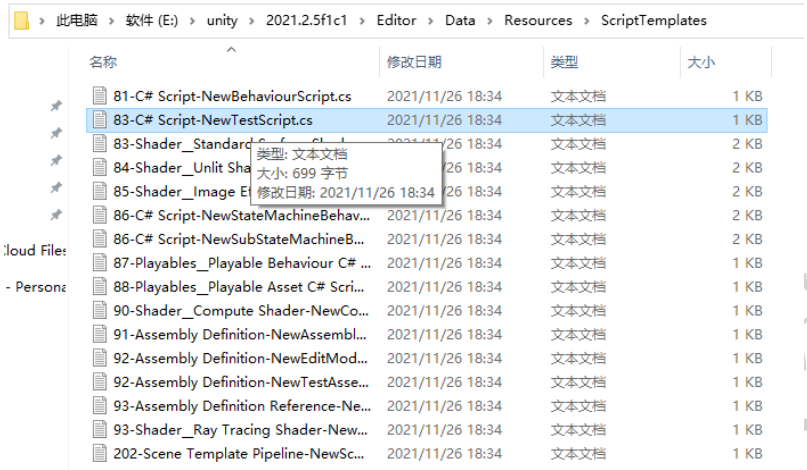


图44. 图 7.1 unity 脚本模板

找到 Unity 版本对应的安装路径，修改对应的脚本模板：

Unity 版本安装路径\Editor\Data\Resources\ScriptTemplates\下的模板直接修改其格式、编码，这样之后在使用时，就可以省略其修改；如果需要修改其内容，则可以在脚本模板中使用一些通配符，然后在脚本生成中进行替换处理。

七、 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/

八、 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。

1. 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。

哔哩哔哩次元旅游主

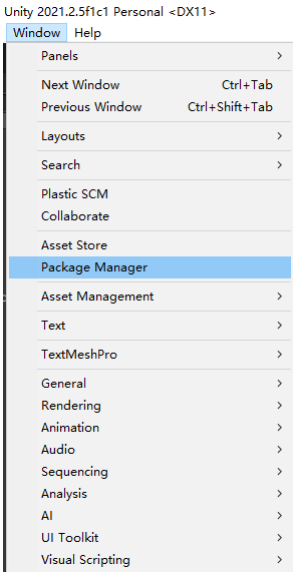


图45.　　图 9.1 打开 PackageManager

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。

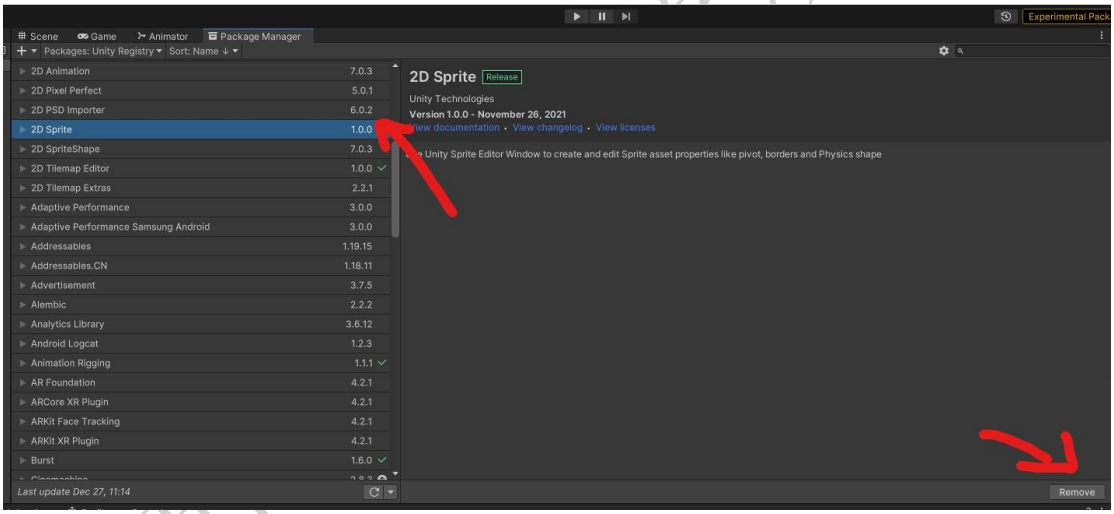


图46.　　图 9.2 将 2D Sprite 导入

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。

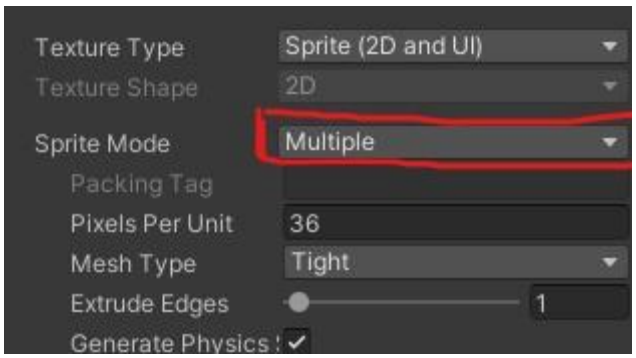


图47.　　图 9.3 设置精灵图为多图模式

2. 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。

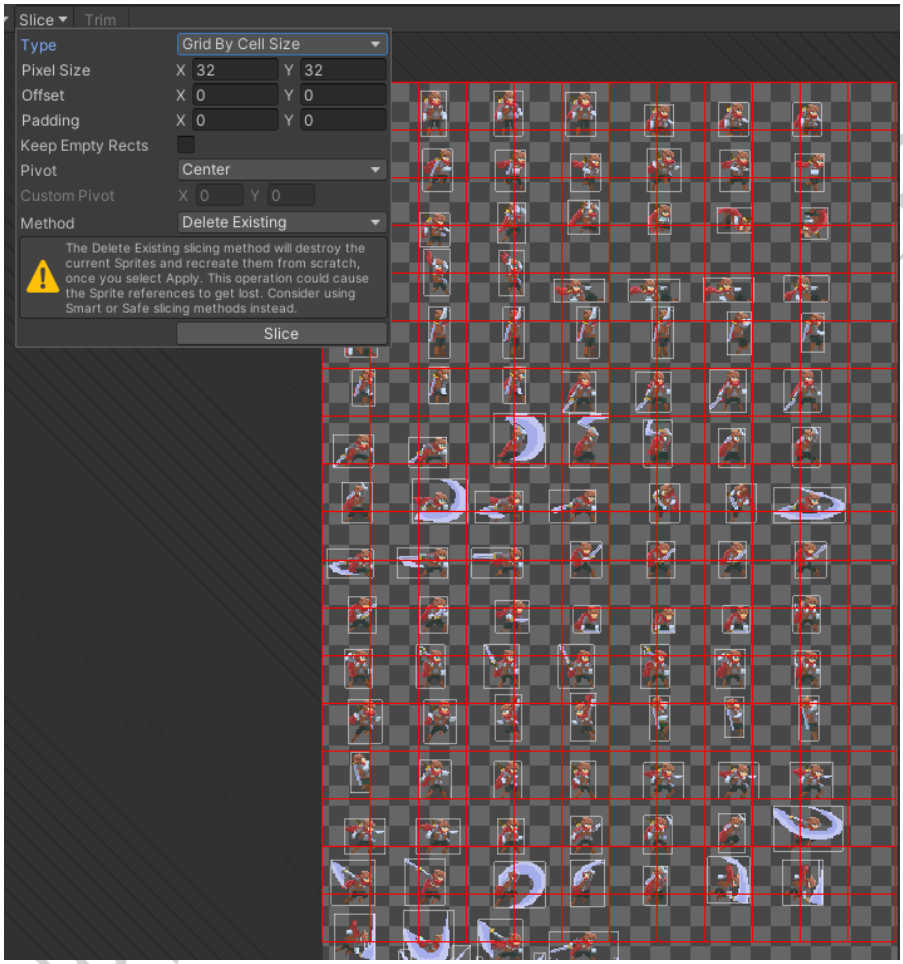


图48.　　图 9.4 按单元格大小切片

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。

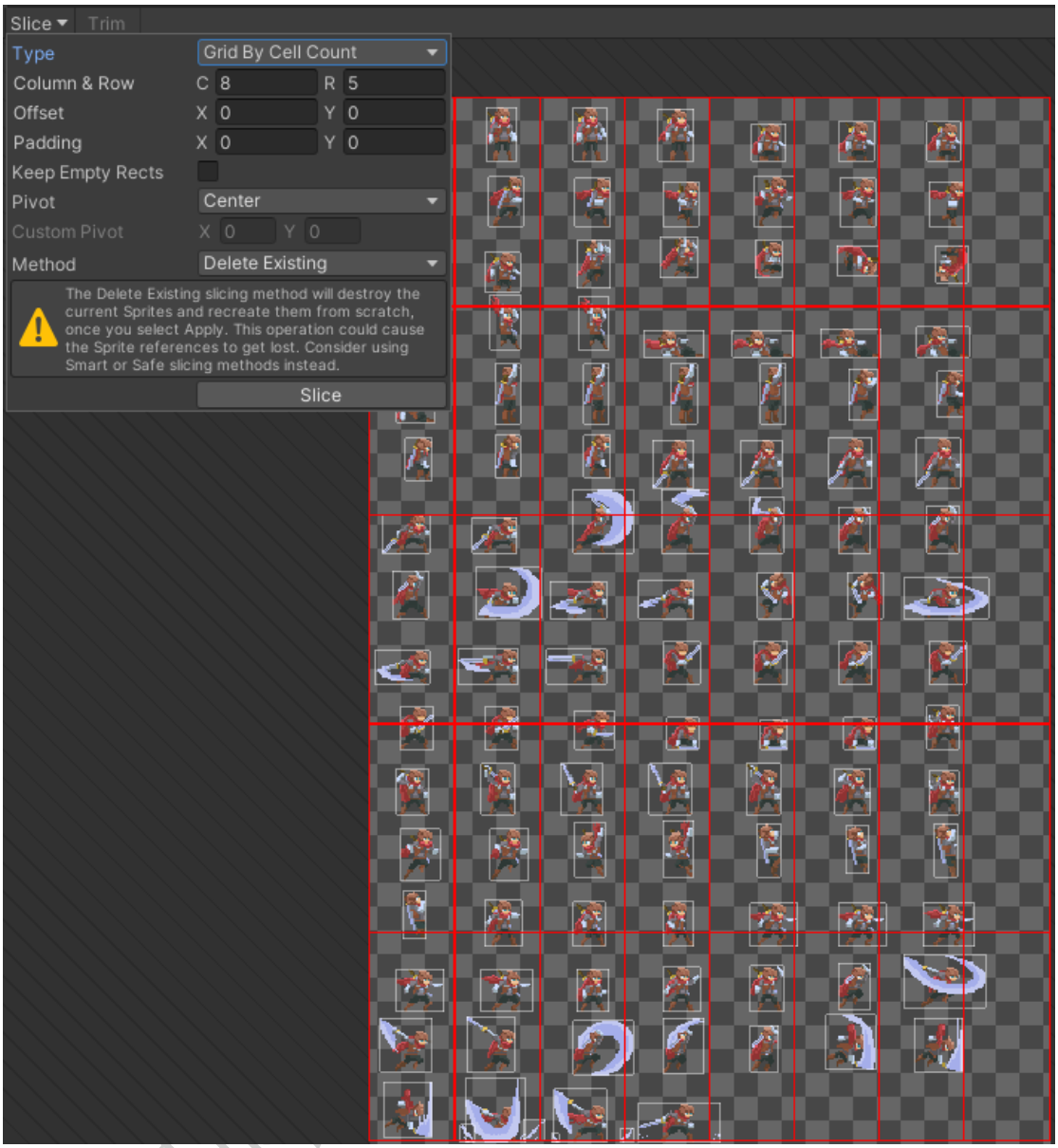


图49.　　图 9.5 按单元格数量切片

九、 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。

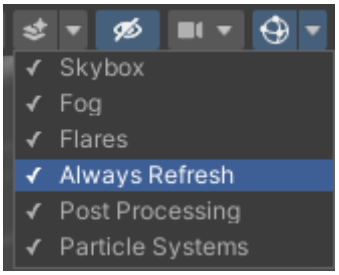


图50.　　保持实时刷新

十、 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。

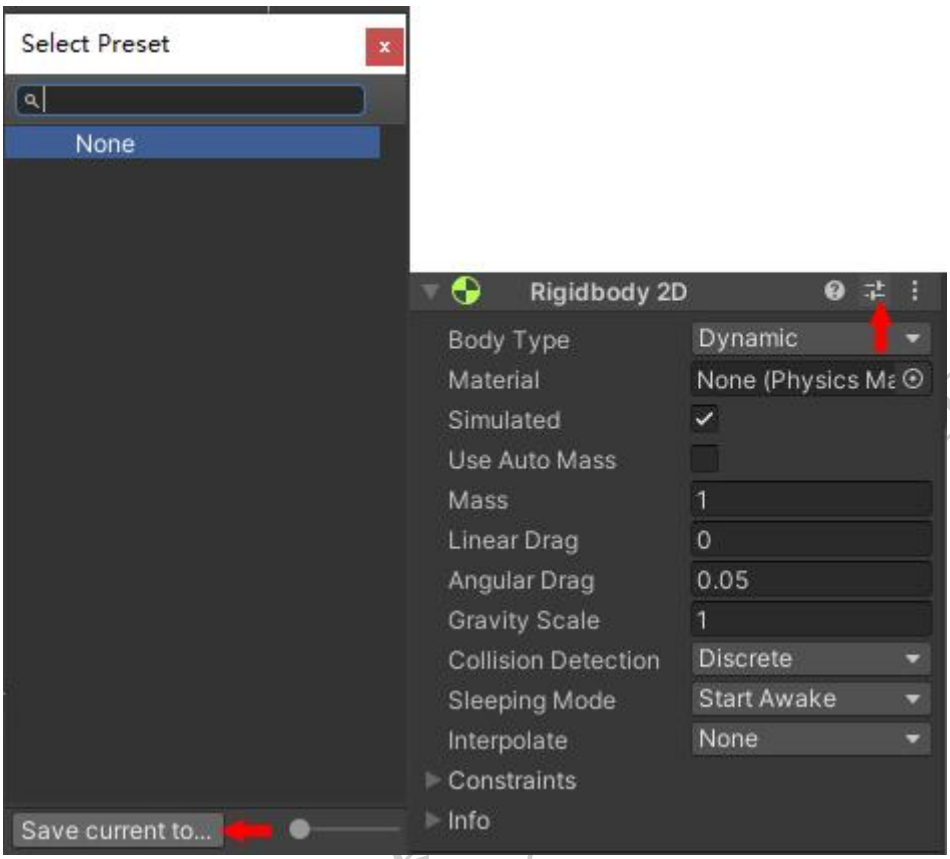


图51.　　组件预设

十一、　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。

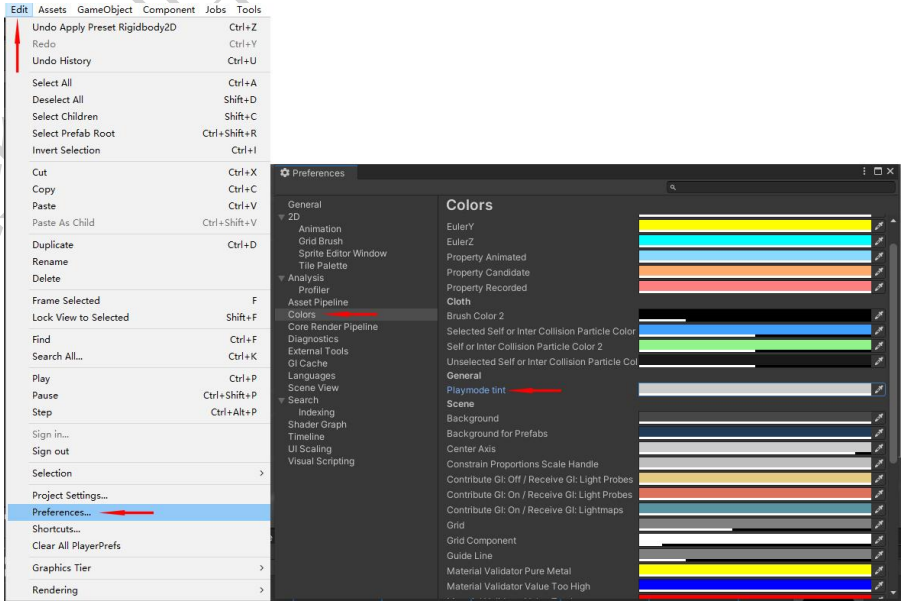


图52.　　修改 Playmode 颜色

十二、 修改鼠标样式和启动图像

在 Unity 中找到 Edit->Project Settings->Player

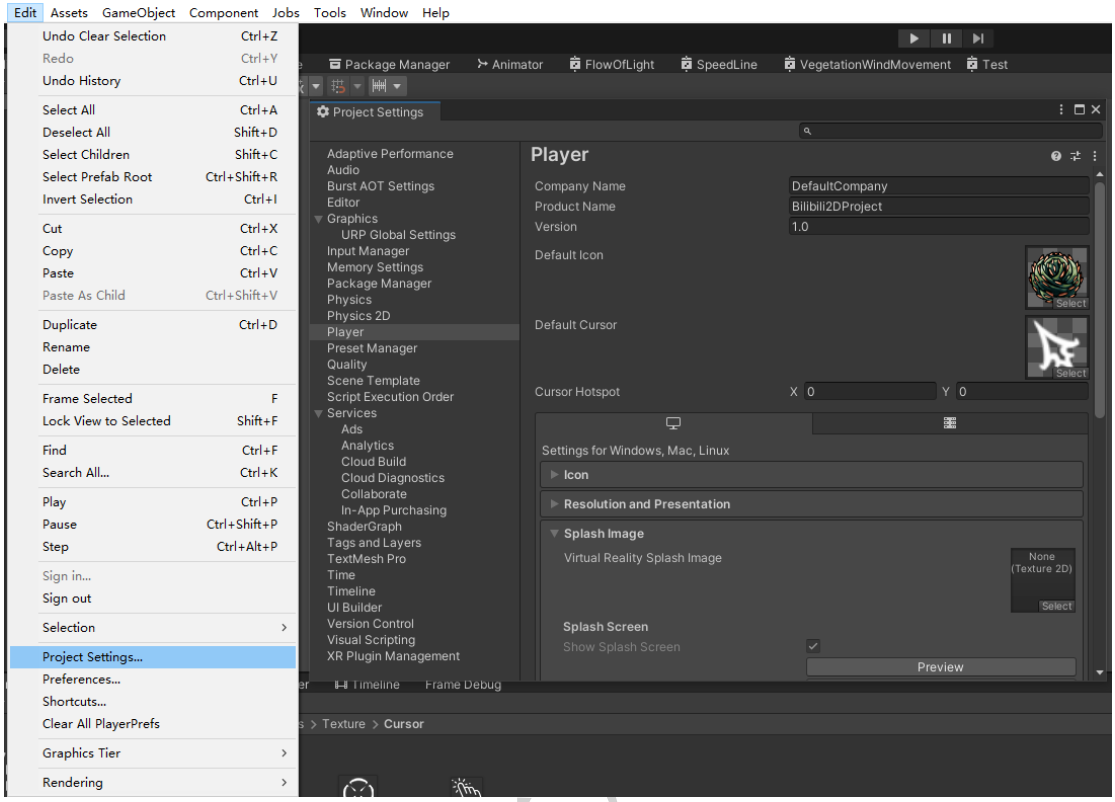


图53. Payer 设置界面

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。

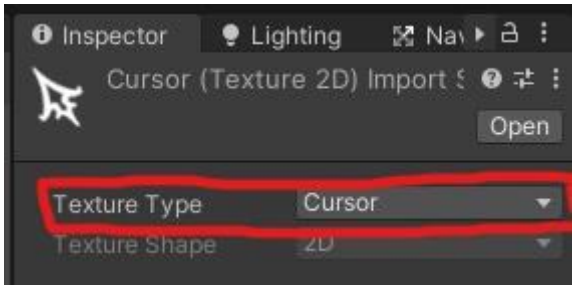


图54. 鼠标样式的纹理类型

2. 设置启动图像

在 Player 设置中找到 Splash Image 设置。

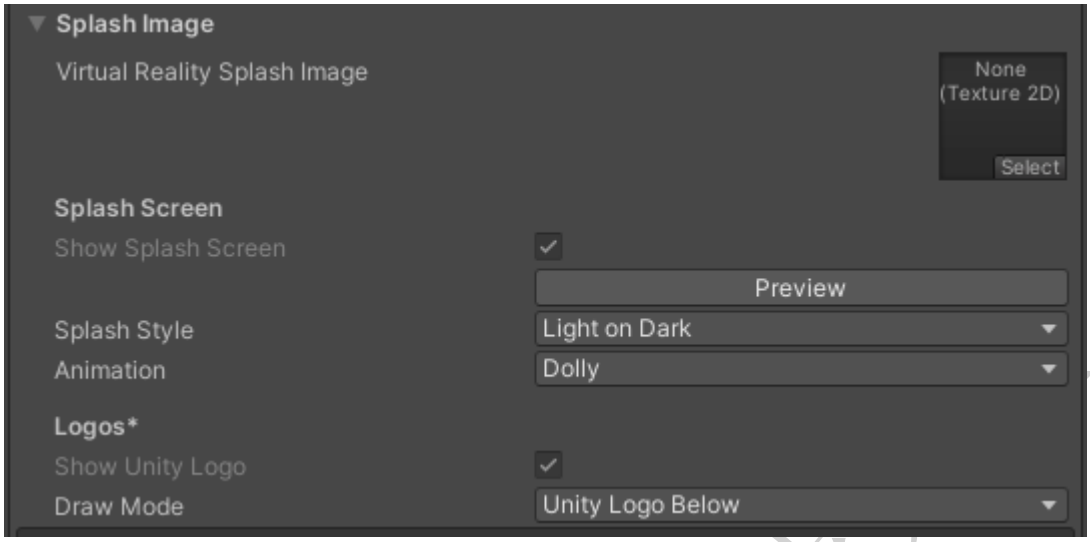


图55. Splash Image

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。

十三、 图片跟随文本自适应

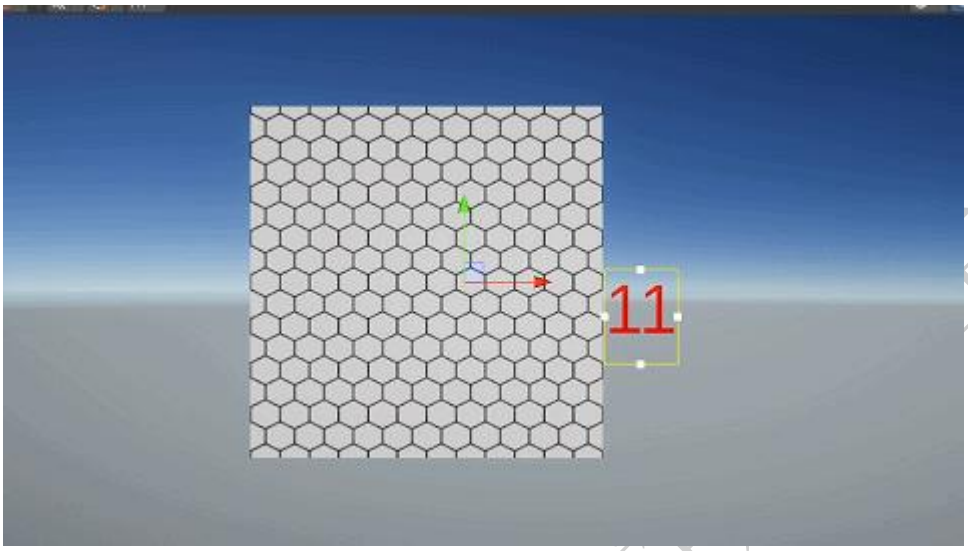


图56. 图像跟随文本移动（这是一个 gif 图，可另存为查看）

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。

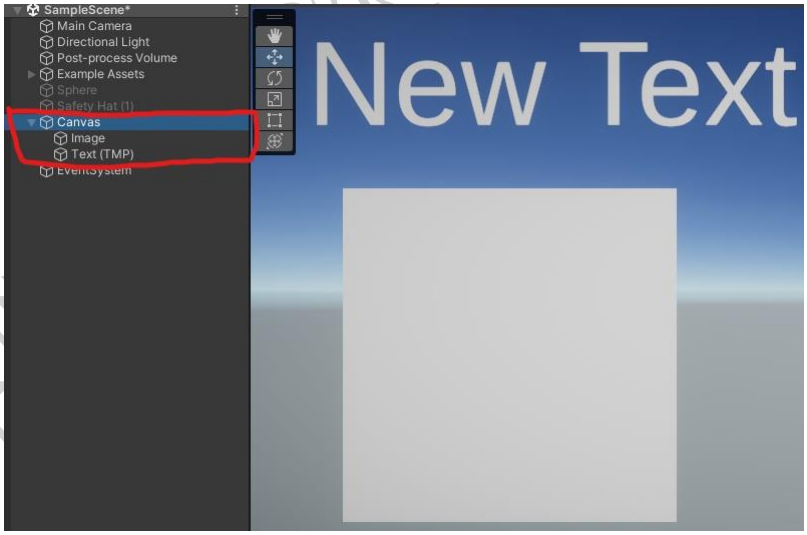


图57. 创建图像和 Text 文本（我这里使用的 Text Mesh Pro）

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）

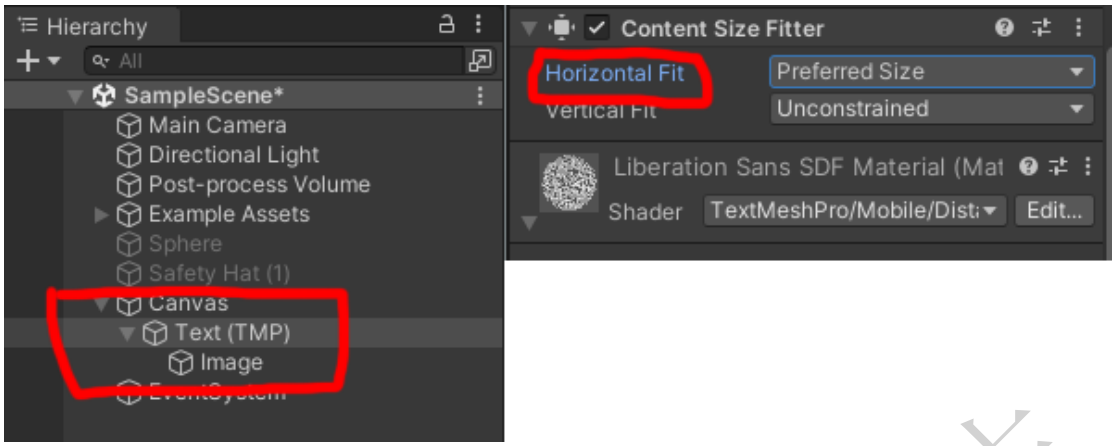


图58.　　让 Image 成为 Text 的子物体

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。

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

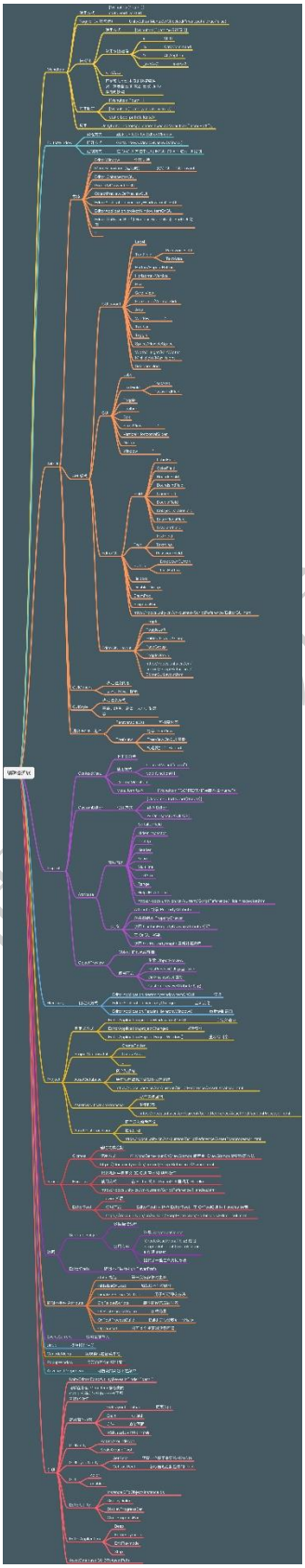


图59.　　图 1. 1 知识地图

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

3. 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

4. 下载

在 Unity 中找到 Window 下的 Package Manager，将 package Manager 界面显示出来。

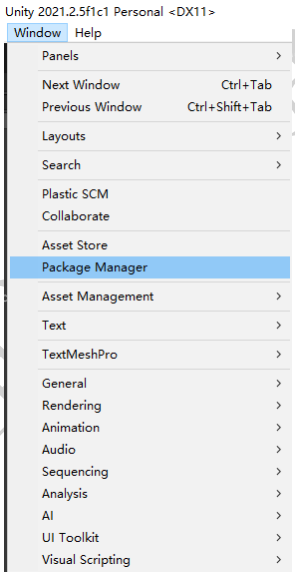


图60. 图 3.1 打开 PackageManager

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

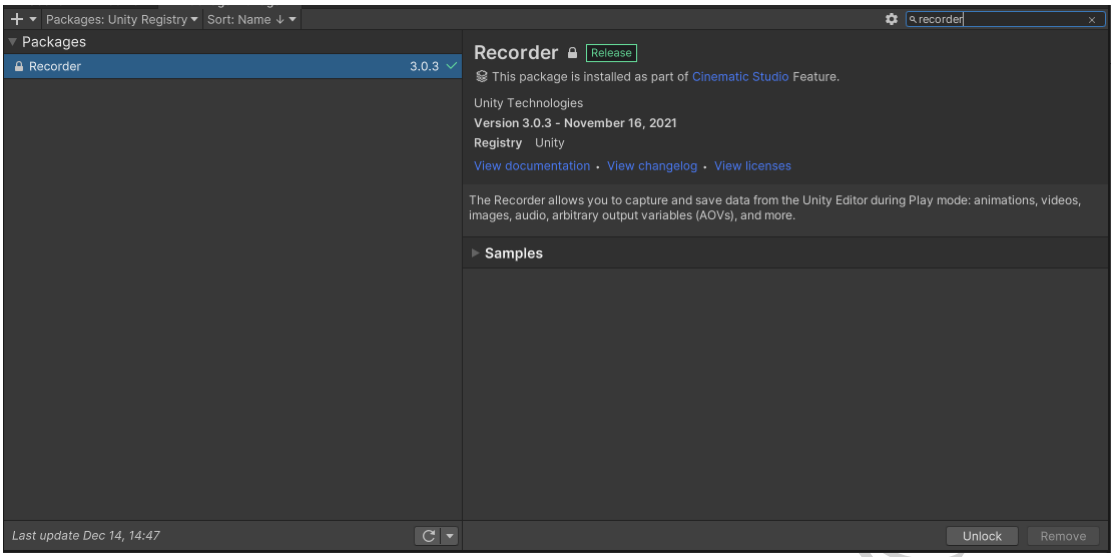


图61.　　图 3.2 下载 Recorder

5. 打开 Recorder 界面

从 Unity 的主菜单（Window > General > Recorder > Recorder window）打开 Recorder 窗口。

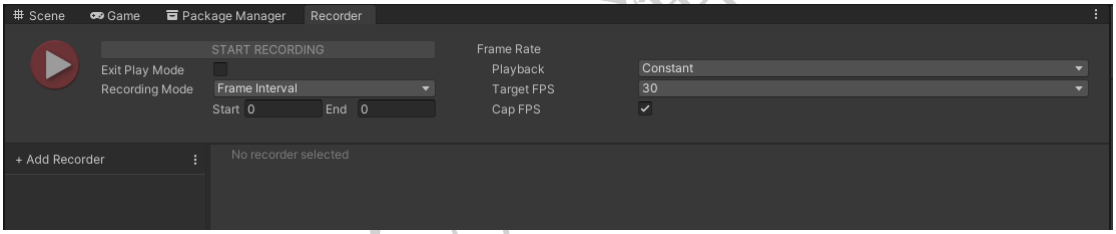


图62.　　图 3.3 Recorder 界面

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。

6. 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。

7. 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。

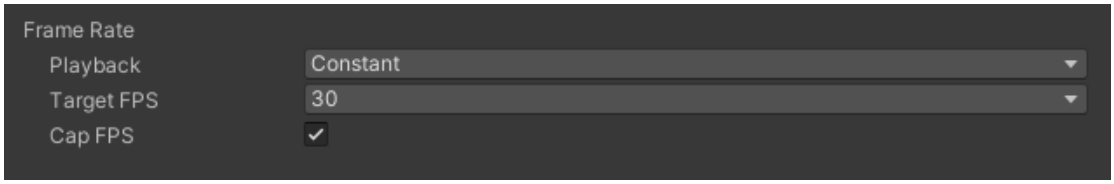


图63.　　图 3.4 Frame Rate

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64.　　表 3.1 占位符功能表

8. Capture-捕捉设置

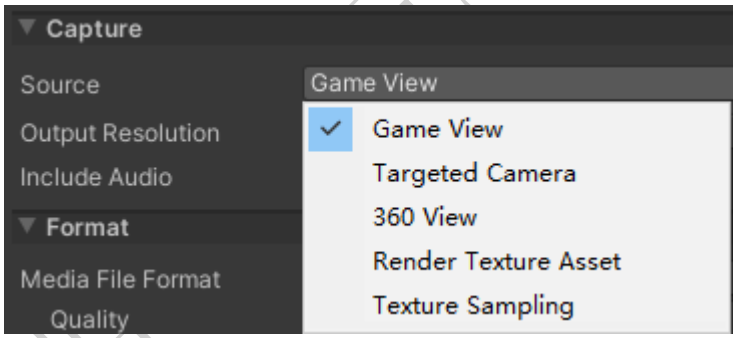


图65.　　图 3.5 捕捉设置

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。

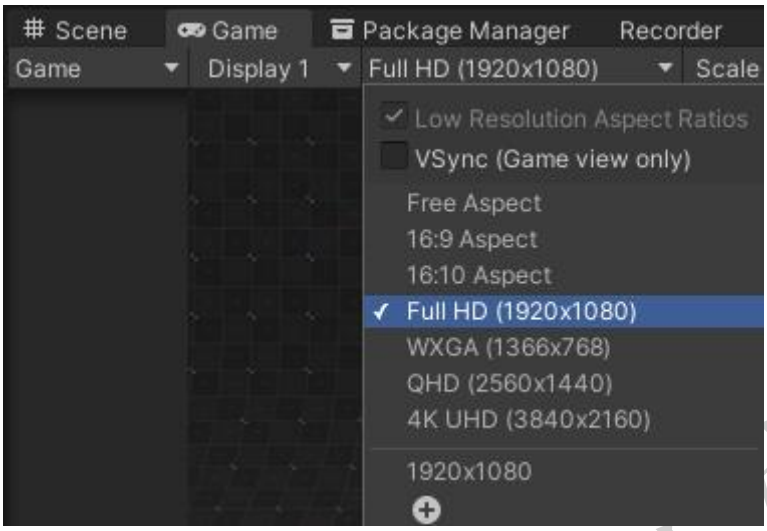


图66.　　图 3.6 游戏视图的分辨率

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。

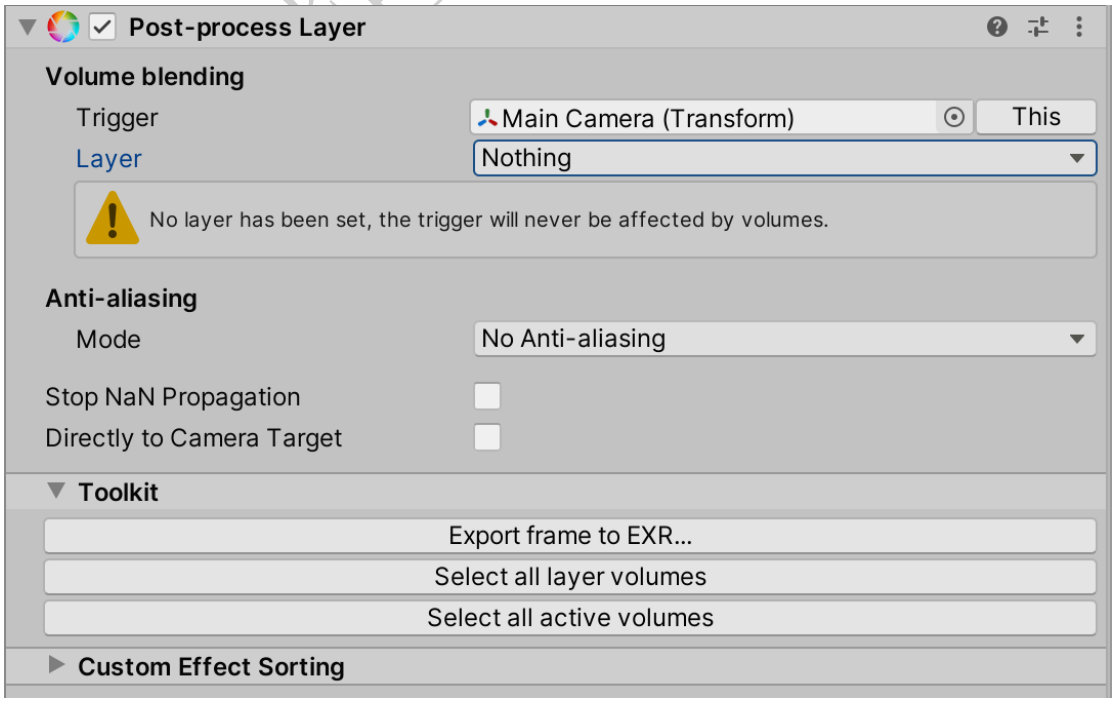


图67.　　图 12.1 Post Process Layer

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()
    {
```

哔哩哔哩 步无游戏人生

```
            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.

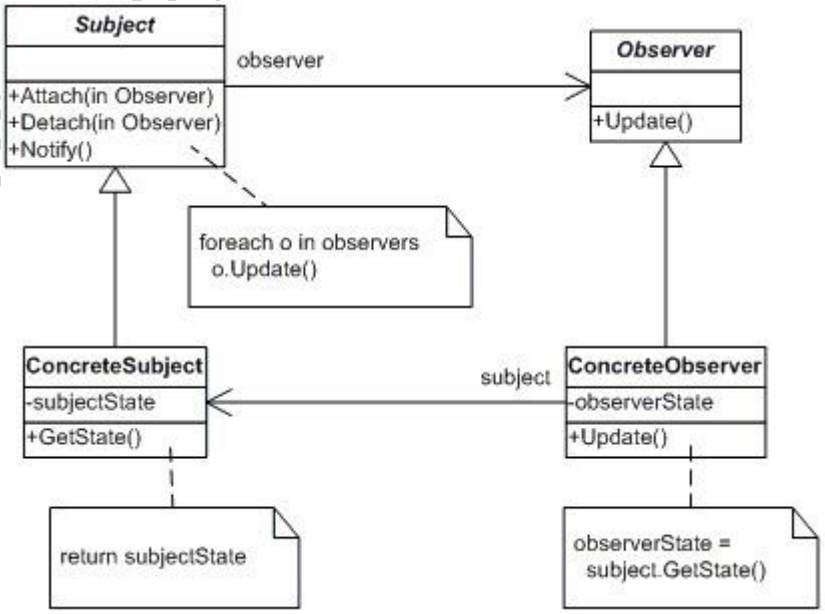


图68. 图 2.1 观察者模式结构图

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）。

哔哩哔哩次元游戏人生

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;

        //结构体没有无参构造函数
        public NumberStruct(int value)
        {
            direction = ElementDirection.Left;
            number = value;
        }

        //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;
        }
    }
```

哔哩哔哩次元游戏人生

```
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));
        }
    }

    public List<int> JohnsonTrotterAlgorithm(List<NumberStruct> list)
    {
        //只要还存在可移动的元素就将继续
        while (FindMaxMobileIndex(list) != -1)
        {
            AddToArrangement(list,ref this.arrangementList);
        }
        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];
            }
            else
            {
                return false;
```

哔哩哔哩次元游戏人生

```
                }
            }
            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];
                }
                else
                {
                    return false;
                }
            }
        }

        /// <summary>
        ///  查找当前最大的可移动元素的索引下标
        /// </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<int> temapry = new List<int>();
            for (int index = 0; index < list.Count; index++)
            {
                if (CanMobile(list, index))
                {
                    temapry.Add(list[index].number);//将所有的可移动元素添加进临时列表
                }
            }
            if (temapry.Count == 0)//没有任何可移动元素
            {
                return -1;
            }
            else
            {
                int maxIndex = 0;//默认第一个是最大的
                for (int i = 0; i < temapry.Count - 1; i++)
                {
                    if (temapry[i] < temapry[i + 1])
```

哔哩哔哩次元游戏人生

```
                {
                    maxIndex = i + 1;
                }
                else
                {
                    maxIndex = i;
                }
            }
            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)
    {
        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，不是交互里面的值
    /// </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 , int currentIndex)
    {
        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)
                    {
                        NumberStruct tempary = list[currentIndex];
                        list[currentIndex] = list[currentIndex + 1];
```

哔哩哔哩次元游戏人生

```
                    list[currentIndex] = tempary;
                }
                else
                {

                }
                break;
            }
        case ElementDirection.Left:
            {
                if (currentIndex > 0)//方向向左且不是第一个元素
                {
                    NumberStruct tempary = list[currentIndex];
                    list[currentIndex] = list[currentIndex - 1];
                    list[currentIndex] = tempary;
                }
                else
                {

                }
                break;
            }
        default:
            break;
        }
    }

    private void ChangeDirection(List<NumberStruct> list, int currentIndex)
    {
        if (list[currentIndex].direction == ElementDirection.Right)
        {
            list[0].direction = ElementDirection.Left;
        }
        else
        {
            list[currentIndex].direction = ElementDirection.Right;
        }
    }
  }
}
/*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）
```

哔哩哔哩次元游戏人生

调转所有大于 k 的元素的方向
将新排列添加到列表中*/

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	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	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

哔哩哔哩次元游戏人生

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

g. 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

h. 定义式

$$[X]_{原}=\begin{cases} X, & 0 \le x < 2^{n-1} \\ 2^{n-1} - X, & 2^{n-1} < x \le 0 \end{cases}$$

i. 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$[X]_{原}$ $\xleftrightarrow[\text{数值位按位取反}]{\text{符号位不变}}$ $[x]_{反}$ $\underset{\text{符号位不变，末位减 1}}{\overset{\text{符号位不变，末位加 1}}{\rightleftharpoons}}$ $[X]_{补}$

$[X]_{反}$ $\xleftrightarrow[\text{数值位按位取反}]{\text{符号位的 1 变为“–”}}$ 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。

梦理降哩次元游戏人生

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 将 a 的二进制位左移 2 位，右补 0，

1 << 5，相当于 1 × (2^5) = 32

1 << 0，相当于 1 × (2^0) = 1

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。

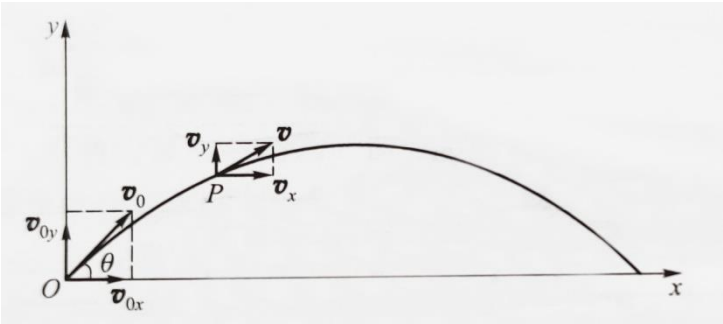


图69. 抛体运动

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、 向量

二、 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。

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）

哔哩哔哩次元游戏人生

$$D=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{i1}+a'_{i1} & a_{i2}+a'_{i2} & \cdots & a_{in}+a'_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

$$=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}+\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a'_{i1} & a'_{i2} & \cdots & a'_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

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！

$$\text{上三角行列式}=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{vmatrix}=a_{11}a_{12}\cdots a_{nn}$$

$$\text{下三角行列式}=\begin{vmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}=a_{11}a_{12}\cdots a_{nn}$$

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！

$$\text{副对角线上三角行列式}=\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & 0 & \cdots & 0 \end{vmatrix}=(-1)^{\frac{1}{2}n(n-1)}a_{11}a_{12}\cdots a_{nn}$$

$$\text{副对角线下三角行列式}=\begin{vmatrix} 0 & 0 & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{in} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}=(-1)^{\frac{1}{2}n(n-1)}a_{11}a_{12}\cdots a_{nn}$$

三、 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。

哔哩哔哩次元游戏人生

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。

四、 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。

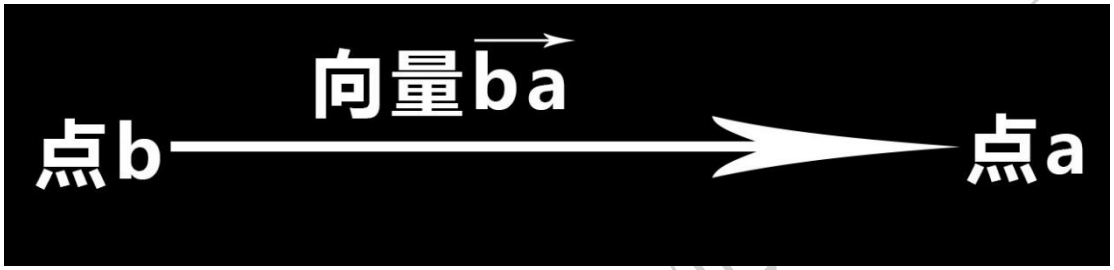


图70. 二维向量

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。

五、 向量在轴上的投影

六、 数量积、向量积

1. 数量积（点乘）

向量$\vec{a}$和$\vec{b}$的数量积，计作$\vec{a}\cdot\vec{b}$，即$\vec{a}\cdot\vec{b}=|\vec{a}||\vec{b}|\cos\theta$

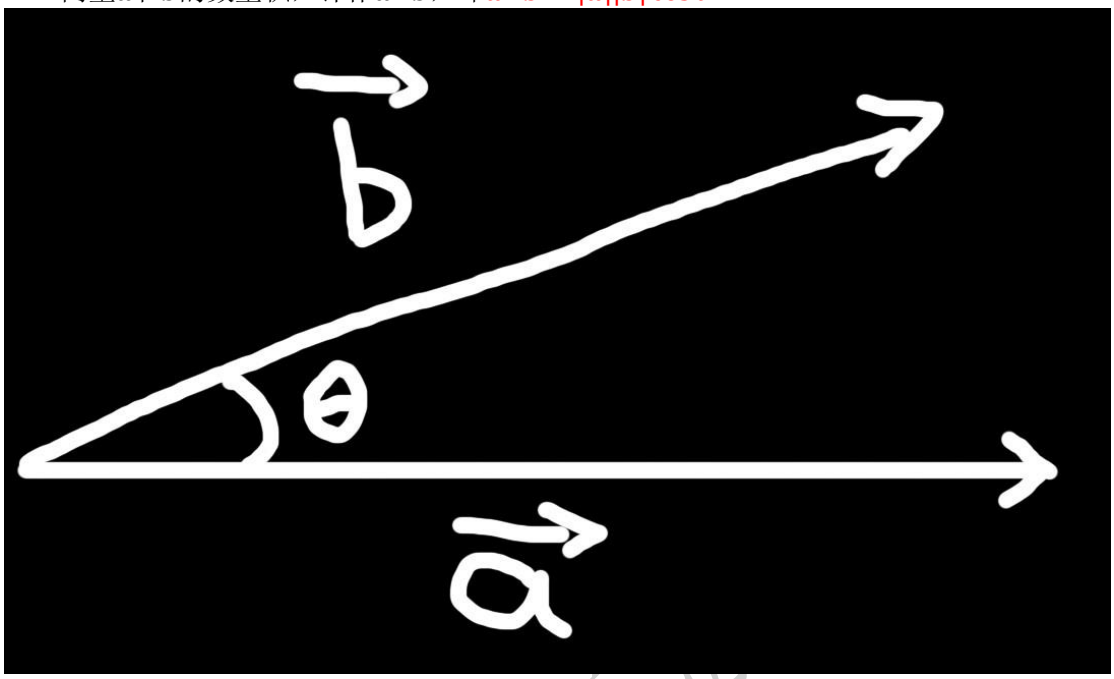


图71. 向量的数量积

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

$(\lambda\vec{a})\cdot\vec{b}=\lambda(\vec{a}\cdot\vec{b})$（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$\vec{b}$）

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。

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）

七、 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，$[\vec{X},\ \vec{X}]>0$。

$[\vec{X},\ \vec{Y}]^2\leq[\vec{X},\ \vec{X}][\vec{Y},\ \vec{Y}]\Leftrightarrow[\vec{X},\ \vec{Y}]^2\leq||\vec{X}||\cdot||\vec{Y}||$

八、 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：$\vec{a}=(1,3,5)^T$则$\|\vec{a}\|=\sqrt{1^2+3^2+5^2}=\sqrt{35}$

1. 性质

1） 非负性

当$\vec{X}\neq 0$时，$\|\vec{X}\|>0$；当$\vec{X}=0$时，$\|\vec{X}\|=0$。

哔哩哔哩次元游戏人生

2） 齐次性

$\|\lambda\vec{\boldsymbol{X}}\| = \sqrt{[\lambda\vec{\boldsymbol{X}},\lambda\vec{\boldsymbol{X}}]} = \sqrt{\lambda^2[\vec{\boldsymbol{X}},\vec{\boldsymbol{X}}]} = |\lambda|\sqrt{[\vec{\boldsymbol{X}},\vec{\boldsymbol{X}}]} = |\lambda|\|\vec{\boldsymbol{X}}\|$（其中$\lambda$为常数）

若$\|\vec{\boldsymbol{X}}\| = 1$时，则称$\vec{\boldsymbol{X}}$为单位向量。若$\alpha \neq 0$，取$\vec{\boldsymbol{X}} = \frac{\alpha}{\|\alpha\|}$，则$\vec{\boldsymbol{X}}$是一个单位向量。由向量$\vec{\alpha}$得到$\vec{\boldsymbol{X}}$的过程称为把向量$\vec{\alpha}$单位化。

哔哩哔哩次元游戏人生

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

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：

$$B(t) = P_0 + (P_1 - P_0)t = (1 - t)P_0 + tP_1, t\epsilon[0,1] \quad （1）$$

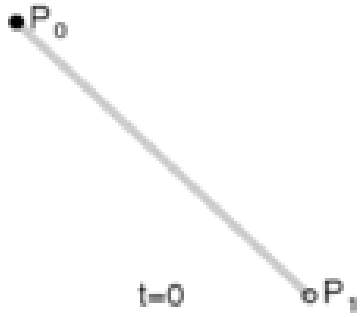


图 3.1 一次贝塞尔曲线 t 在[0,1]之间

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：

$$B(t) = P_0(1 - t)^2 + 2P_1t(1 - t) + p_2t^2, t \in [0,1] \quad （2）$$

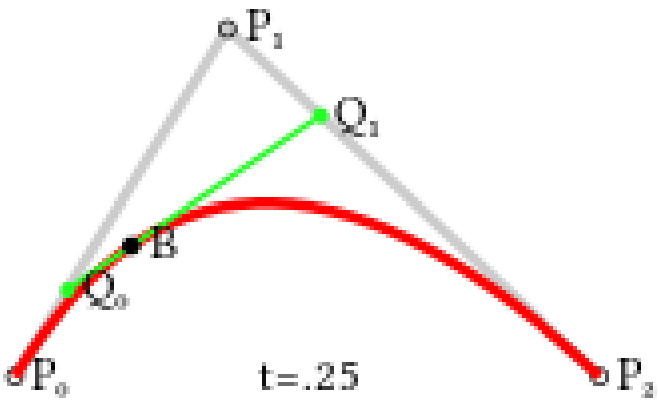


图 3.2 二次贝塞尔曲线的结构 t 在[0,1]之间

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：

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + p_3t^3, t \in [0,1] \quad (3)$$

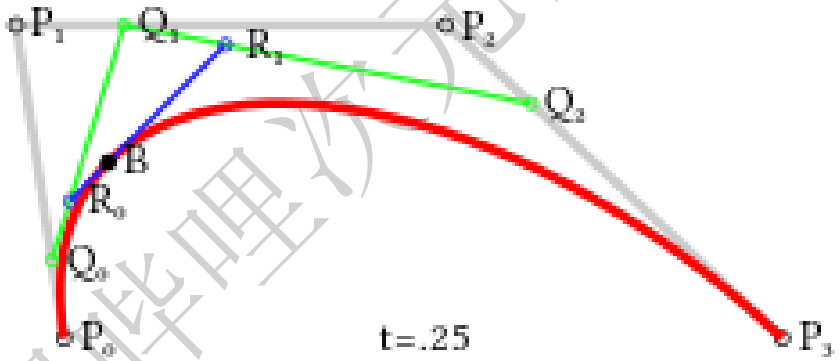


图 3.3 三次贝塞尔曲线的结构 t 在[0,1]之间

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。

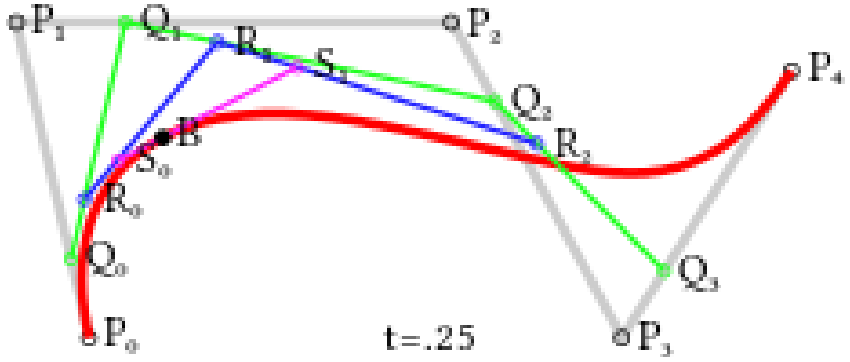


图 3.4 四次贝塞尔曲线的结构 t 在[0,1]之间

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。

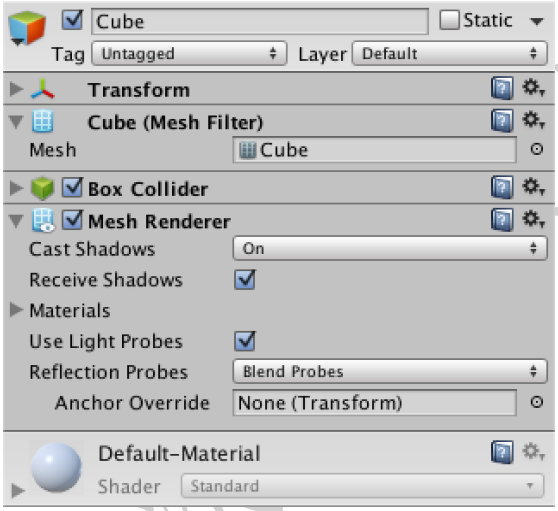


图 3.5 Unity 默认立方体

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。

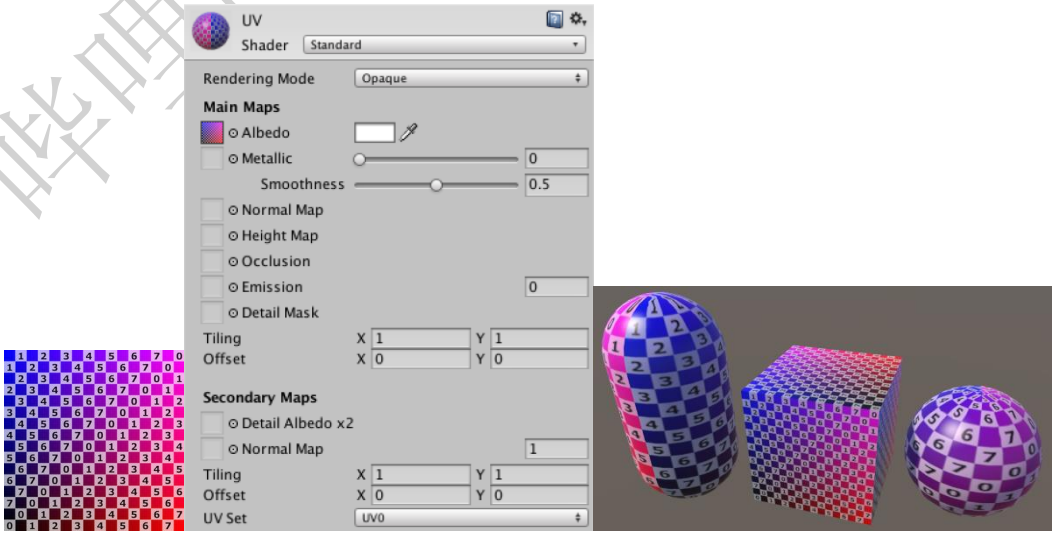


图 3.6 应用于 Unity 的 UV 测试坐标

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。

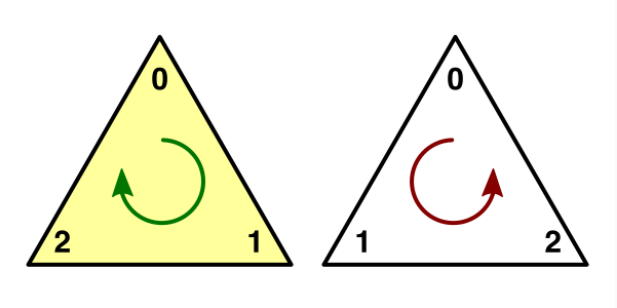


图 3.7 三角形顶点两种遍历情况

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。

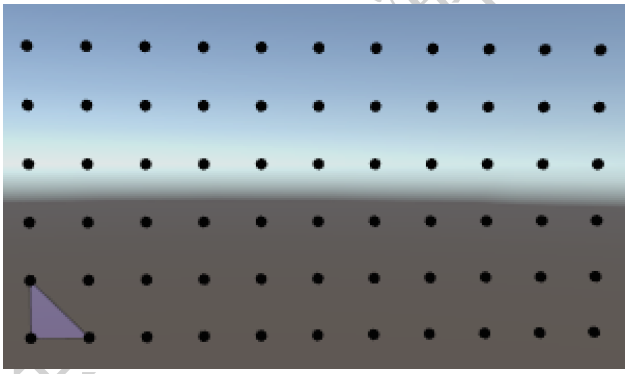
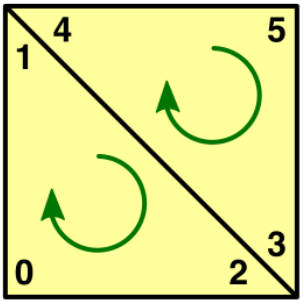


图 3.8 第一个三角形

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

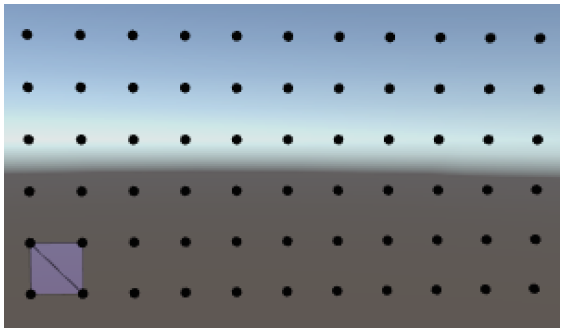


图72. 图 3.9 由两个三角形组成的四边形　　图 3.10 第一个四边形

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。

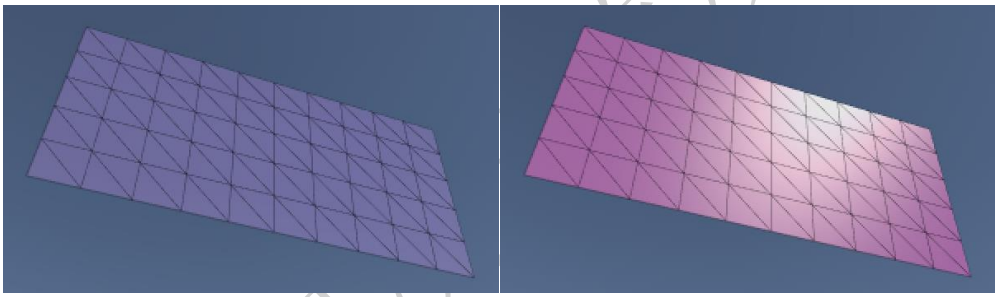


图73.　图 3.11 没有法线与有法线

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。

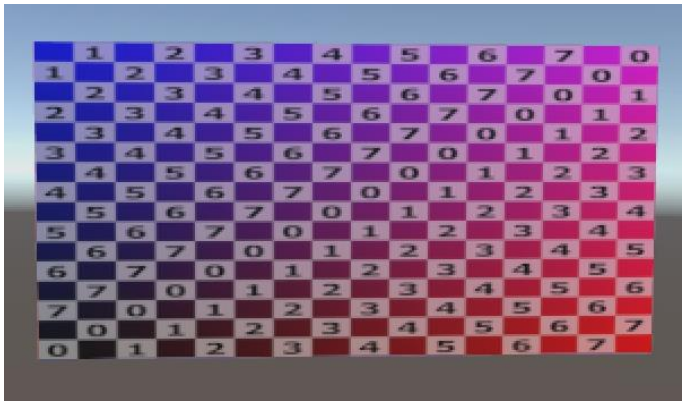


图74.　图 3.12 有 UV 坐标的网格

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、 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。

哔哩哔哩次元游戏人生

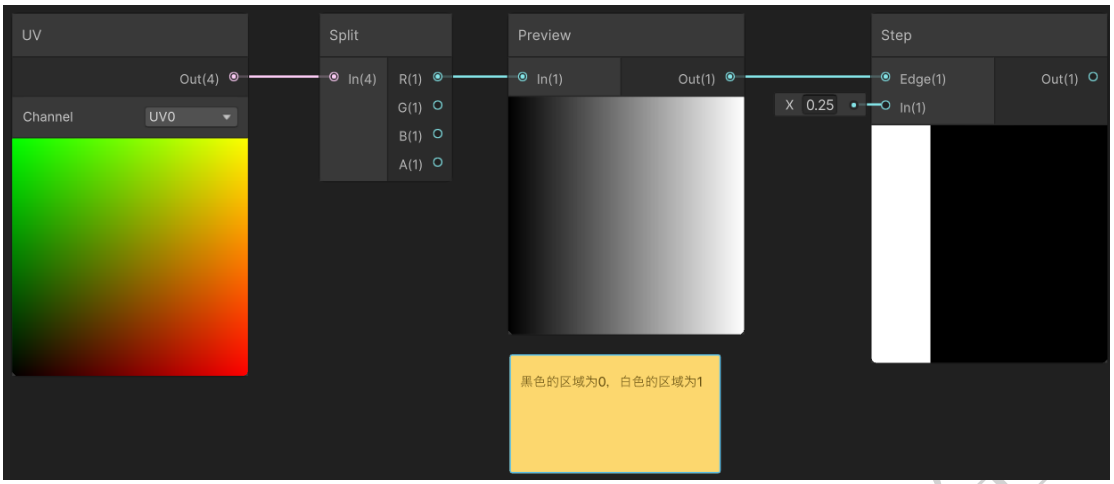


图75.　　图 4.1 Step 节点可视化

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);
}
```

六、 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);
    }
```

七、 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);
    }
```

八、 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。

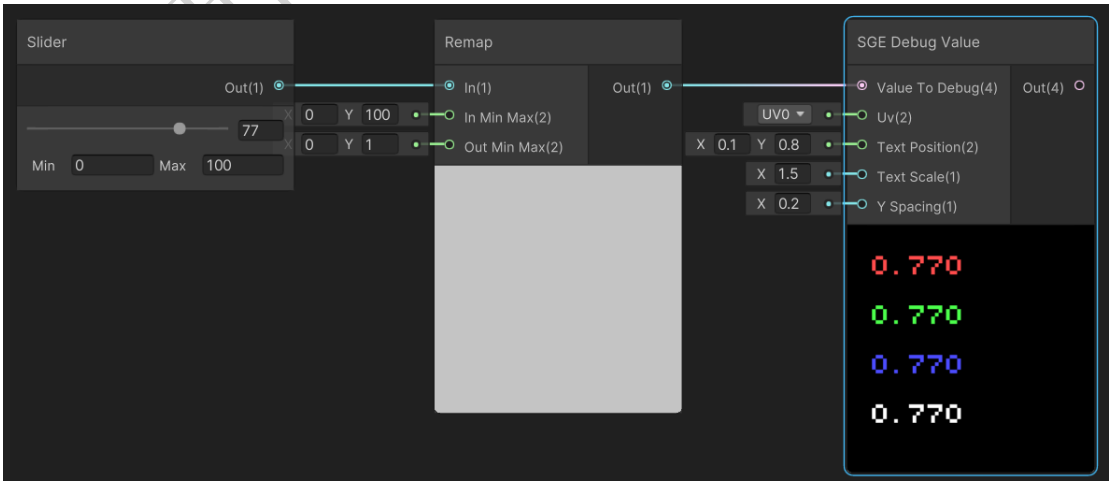


图76. 图 6.1 Remap Node 示例

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);
}
```

九、 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77. 图 6.2 创建节点时输入 Limit

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、 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

[1] n. 夹具，夹钳；v. （用夹具）夹紧，夹住；

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;
```

十一、 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。

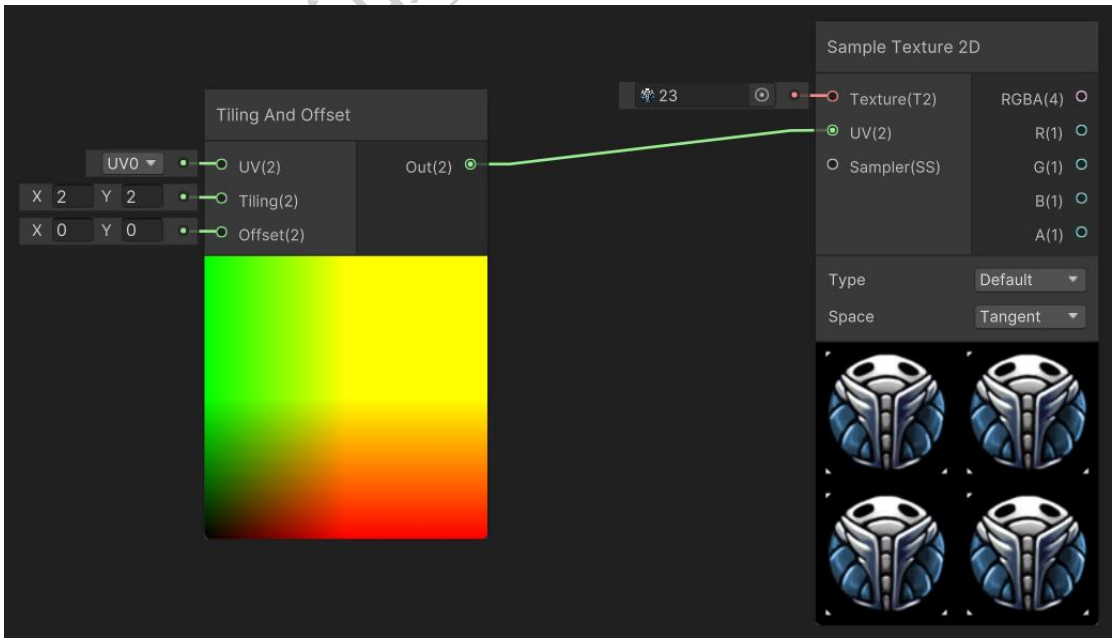


图78. 图 9.1 将 Tiling 的分量都设置为 2

注意，要让图像重复，该图像在 Unity 中的设置必须为 Repeat[2]。

图79.　　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;
}
```

十二、 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;
}
```

十三、 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)
{
```

[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；

哔哩哔哩次元游戏人生

```
    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;
}

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));
```

哔哩哔哩次元游戏人生

```
        t += unity_valueNoise(float2(UV.x*Scale/freq, UV.y*Scale/freq))*amp;

        Out = t;
    }
```

十四、 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;
    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;
```

哔哩哔哩次元游戏人生

```
        rMatrix = rMatrix * 2 - 1;
        UV.xy = mul(UV.xy, rMatrix);
        UV += Center;
        Out = UV;
    }
```

十五、 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);
}
```

十六、 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);
}
```

十七、 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。

哔哩哔哩次元游戏人生

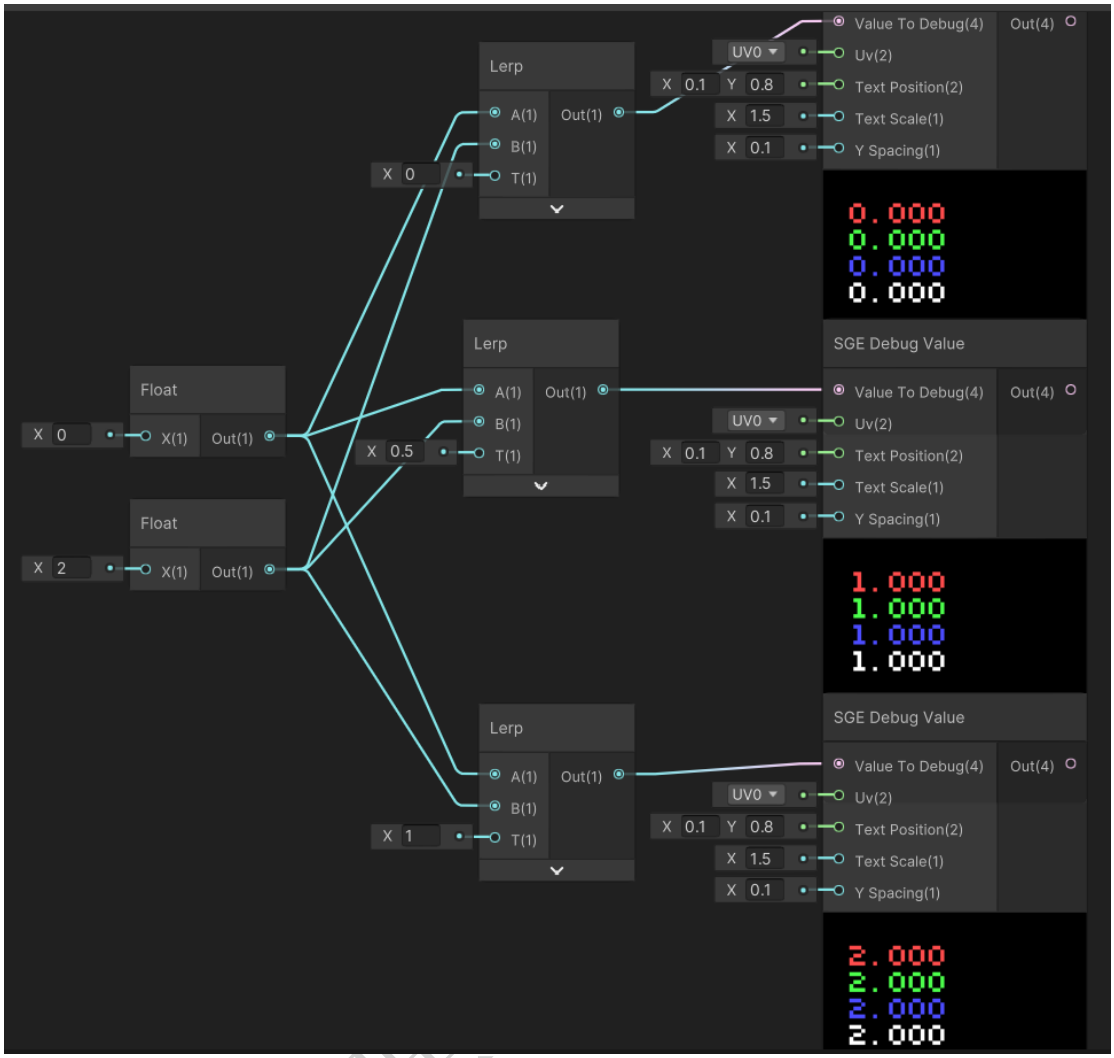


图80. 线性插值图示

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);
}
```

十八、 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。

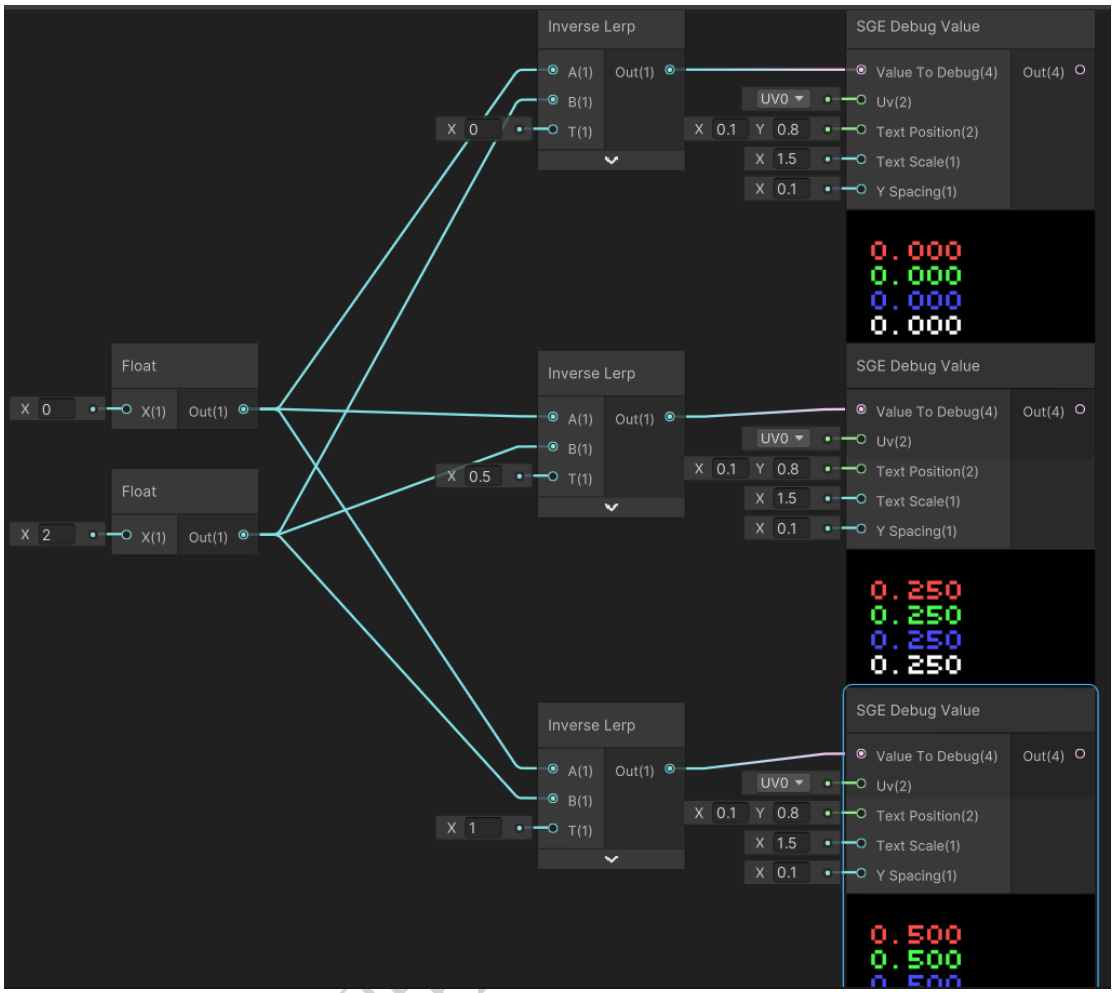


图81. 反插值图示

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);
}
```

十九、 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。

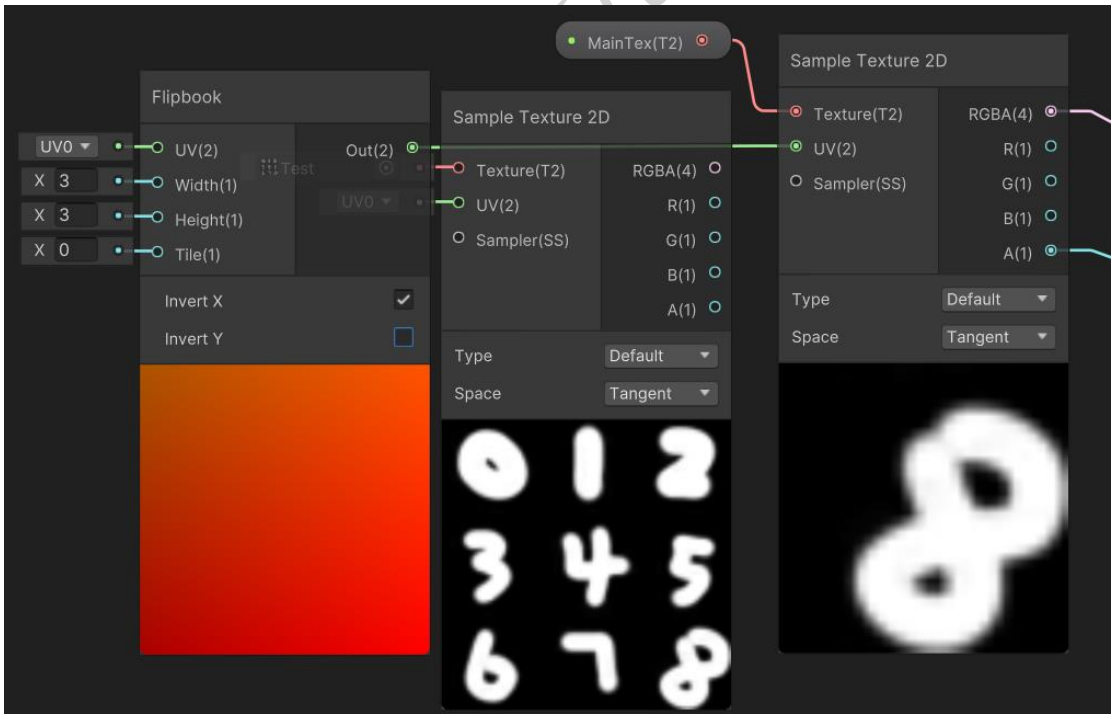


图82. 启用 X 关闭 Y

2） 启用 X 和 Y

图片最上层为第一行，最右边为第一列。

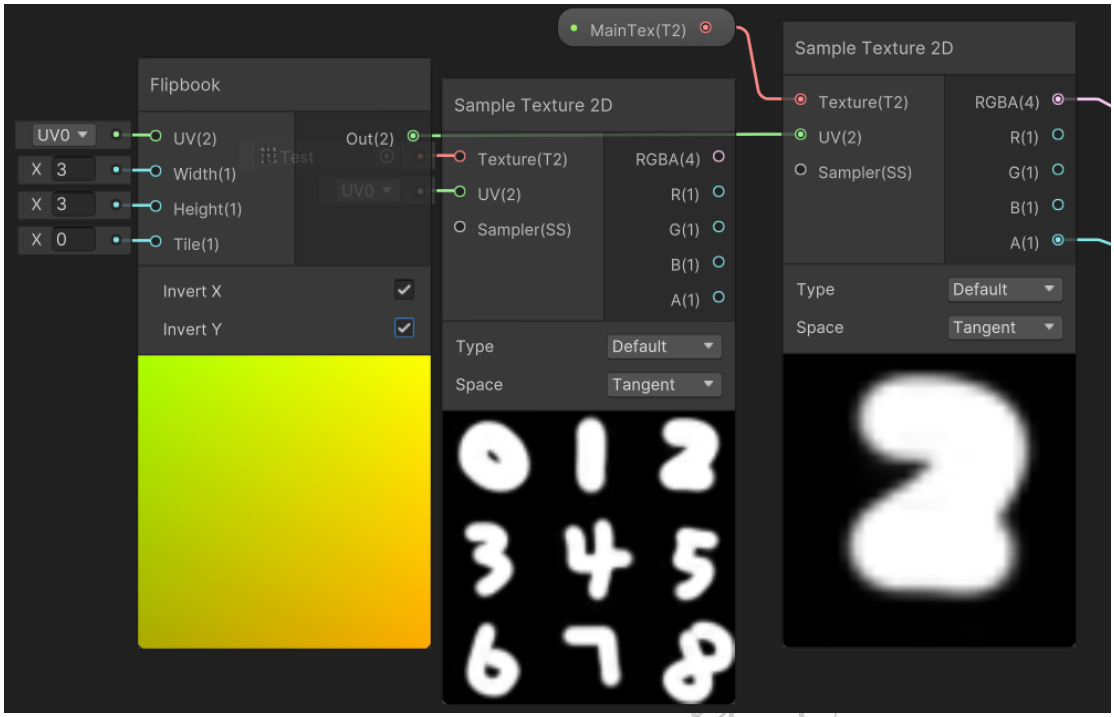


图83.　　启用 X 和 Y

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、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);
}
```

二十一、 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);
}
```

二十二、 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
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);
}
```

二十三、 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

哔哩哔哩次元游戏人生

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;
}
```

二十四、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。

二十五、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。

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);
}
```

二十六、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

哔哩哔哩次元游戏人生

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);

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));

哔哩哔哩次元游戏人生

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);

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);

哔哩哔哩次元游戏人生

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;

二十七、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);
}
```

二十八、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);
}
```

二十九、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

哔哩哔哩次元游戏人生

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、 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);
}
```

三十一、 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);
}
```

三十二、 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。

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);
```

哔哩哔哩次元游戏人生

```
    Out = lerp(Min, Max, randomno);
}
```

三十三、 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;
}
```

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;
}
```

哔哩哔哩次元游戏人生

6） GreaterOrEqual

```
void Unity_Comparison_GreaterOrEqual_float(float A, float B, out float Out)
{
    Out = A >= B ? 1 : 0;
}
```

三十四、快捷键

Ctrl+C 复制

Ctrl+V 粘贴

Ctrl+G 将选中内容编组

Delete 删除

Space 创建节点

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。

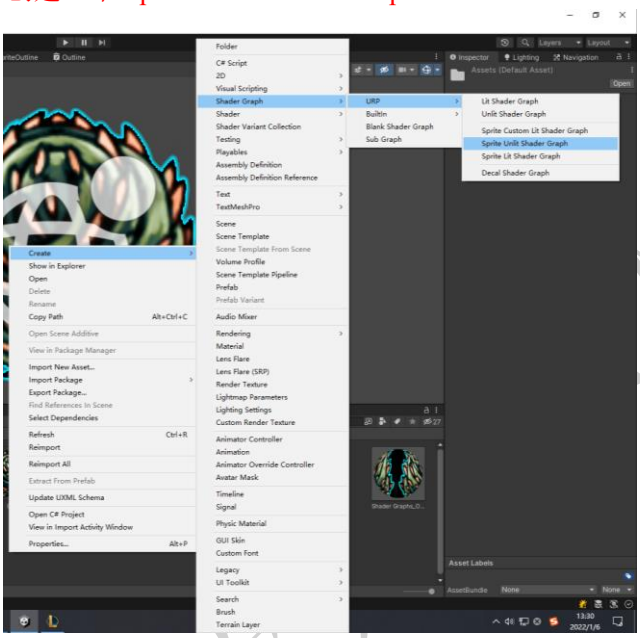


图84. 图 1.1 创建 Sprite Unlit Shader Graph

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85. 图 1.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

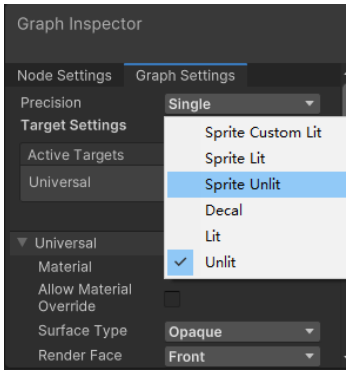


图86. 图 1.3 修改 Material

[1] adj. 普遍的；通用的 n. 普遍原则；普遍看法

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）。

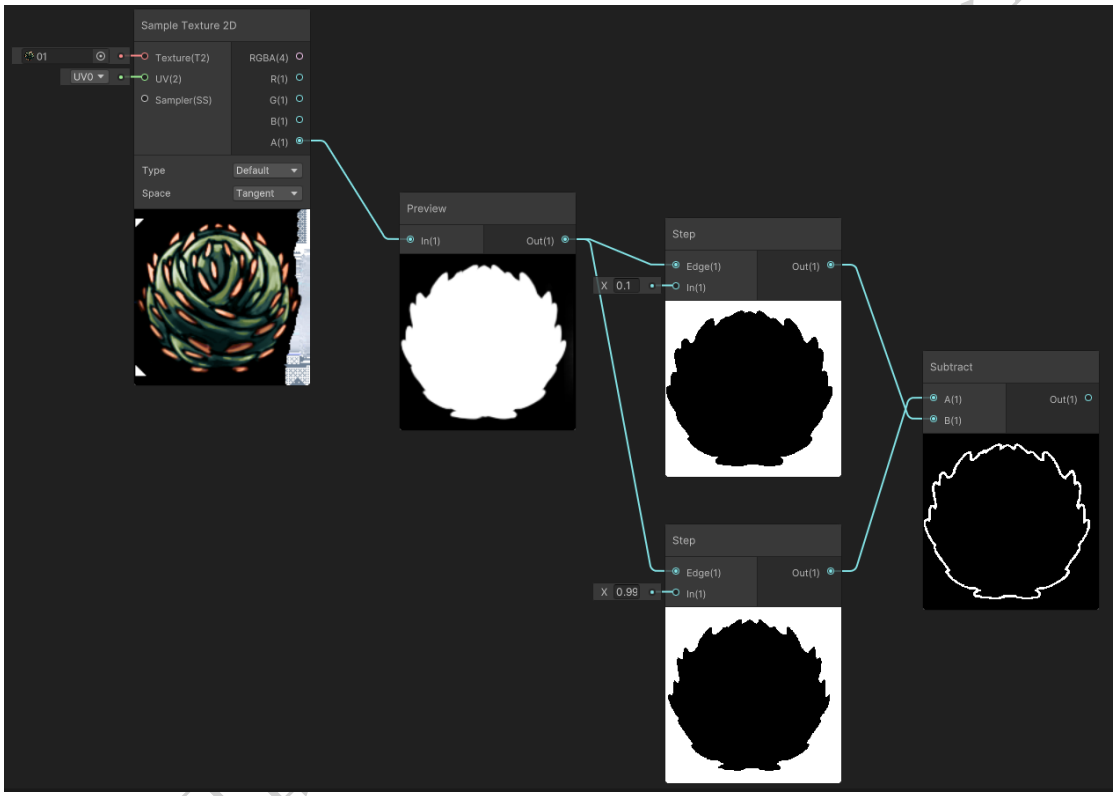


图87.　　图 1. 4 描边轮廓

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。

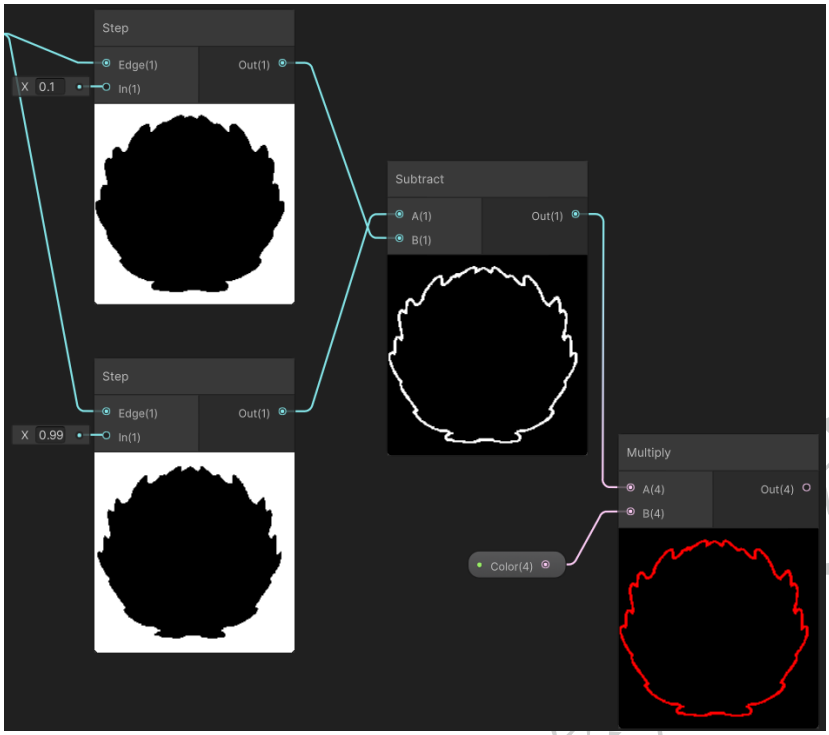
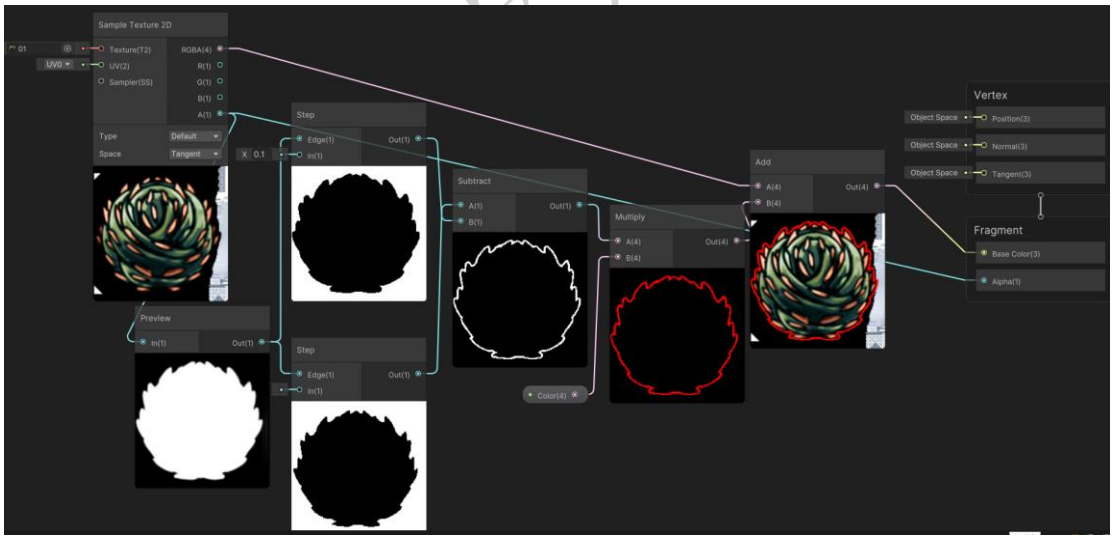


图88.　　图 1.5 修改描边轮廓

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。

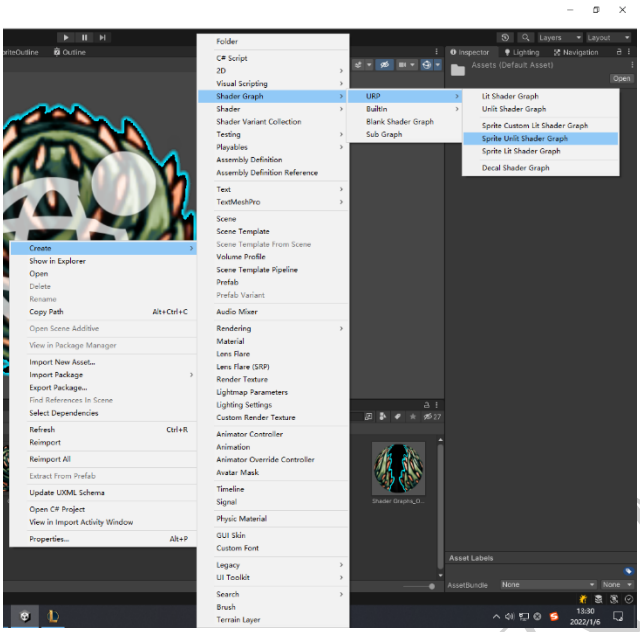


图89.　　图 2.1 创建 Sprite Unlit Shader Graph

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90.　　图 2.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

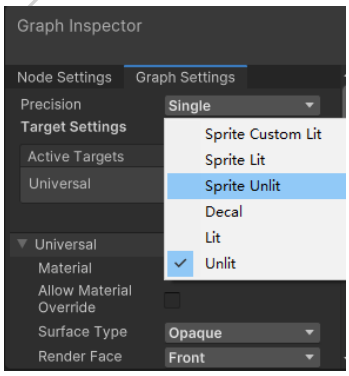


图91.　　图 2.3 修改 Material

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。

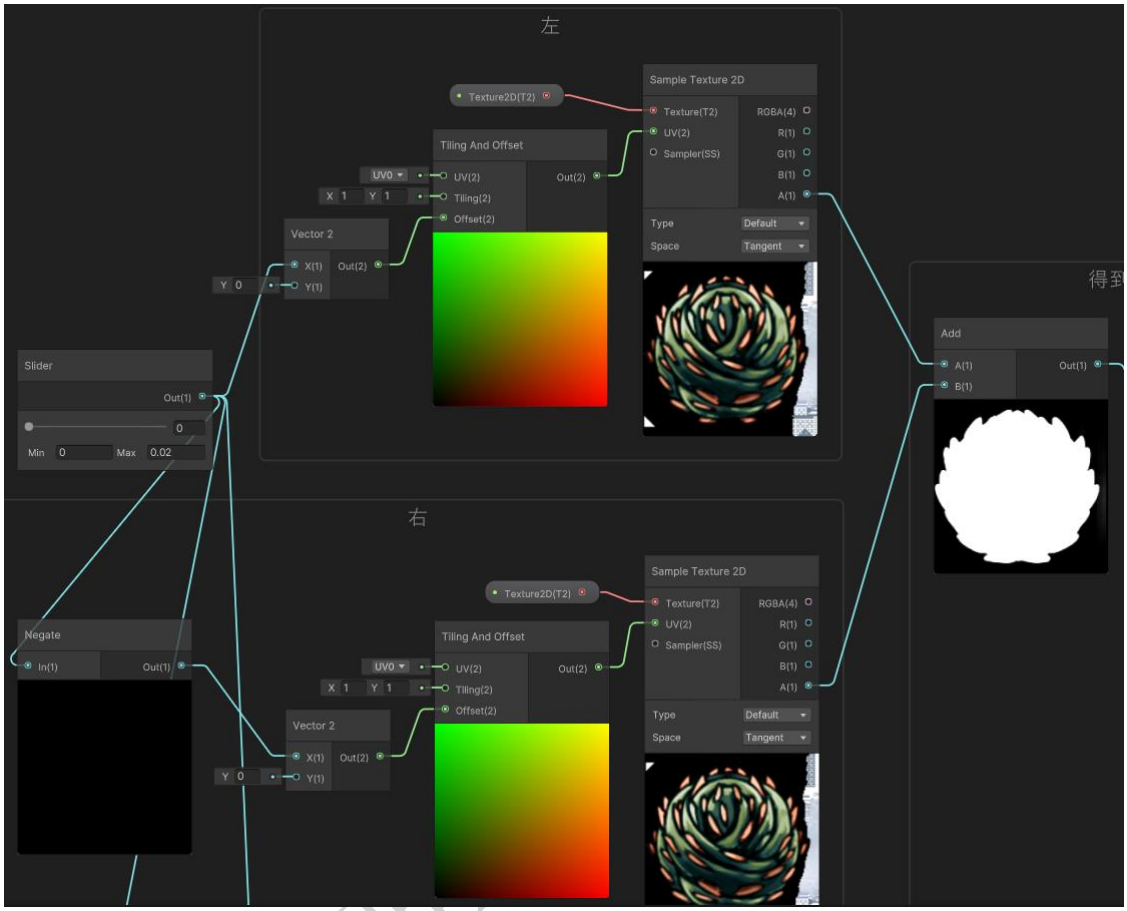


图92.　　图 2.4 得到左右偏移图像

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。

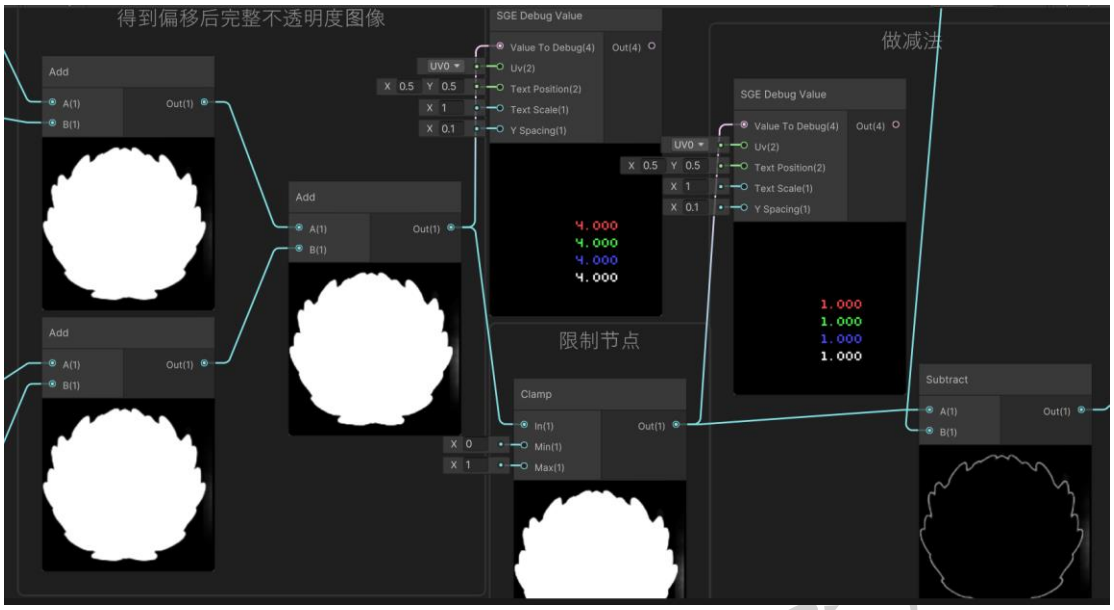


图93.　　图 2.5 通过 Clamp 节点得到正确描边

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。

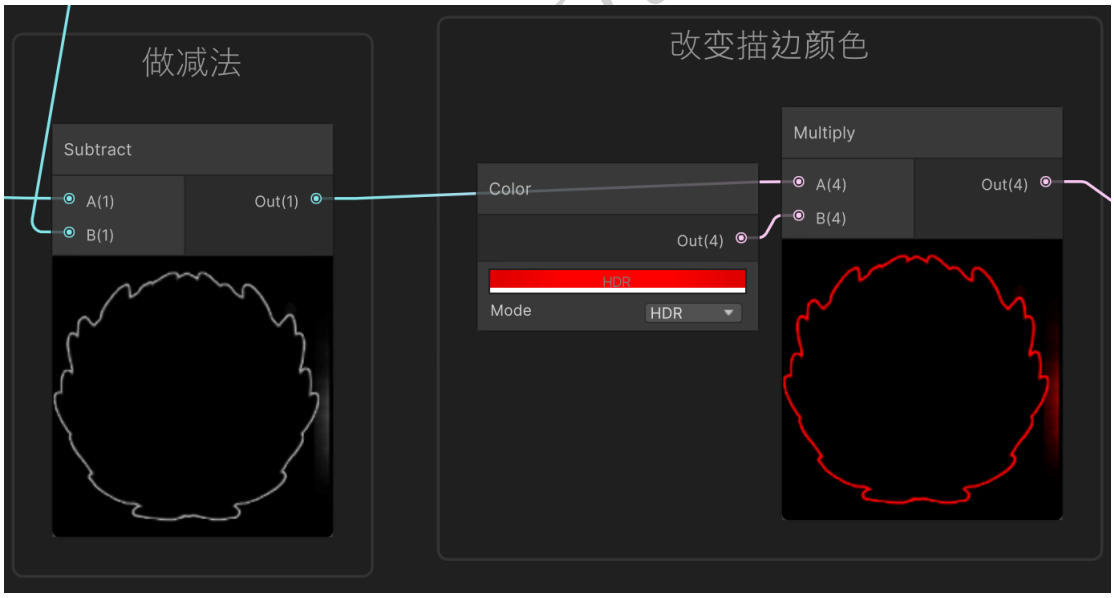


图94.　　图 2.6 修改描边区域的颜色

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。

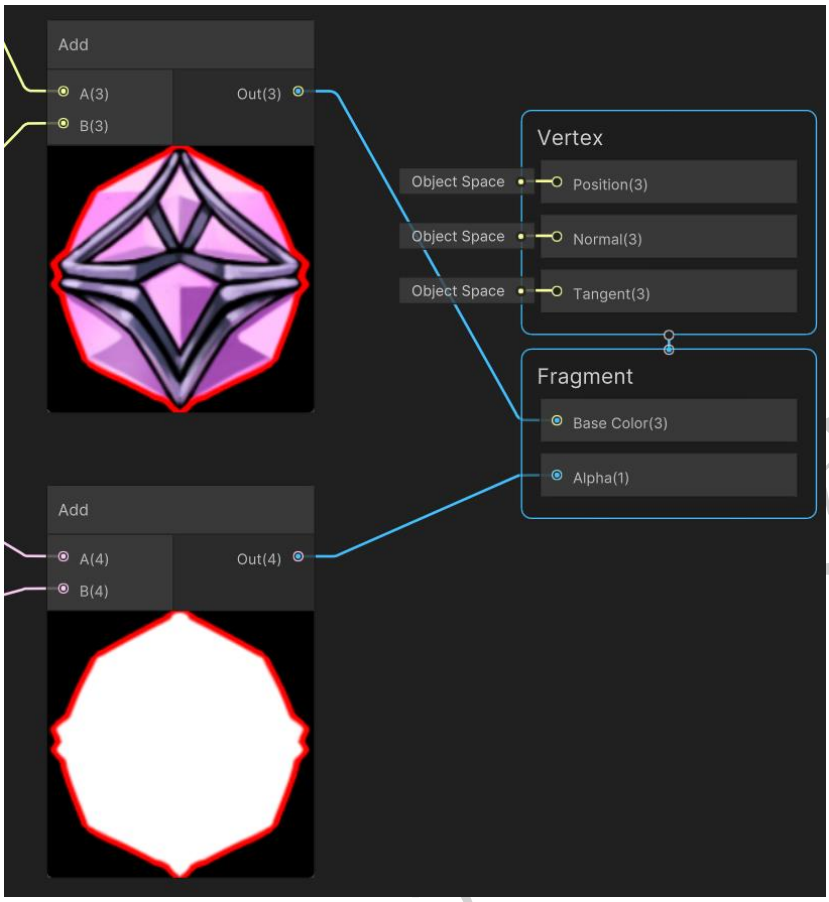


图95.　　图 2.7 输出图像

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。

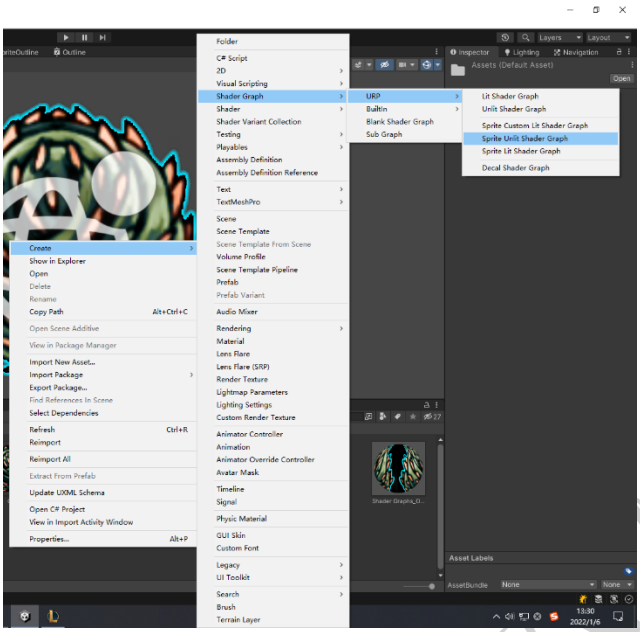


图96.　　图 3.1 创建 Sprite Unlit Shader Graph

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97.　　图 3.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

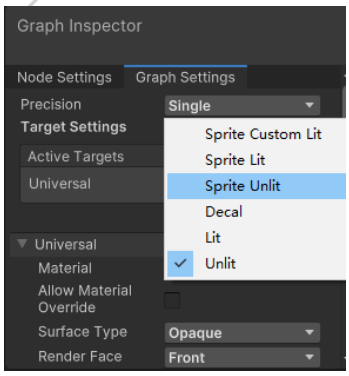


图98.　　图 3.3 修改 Material

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

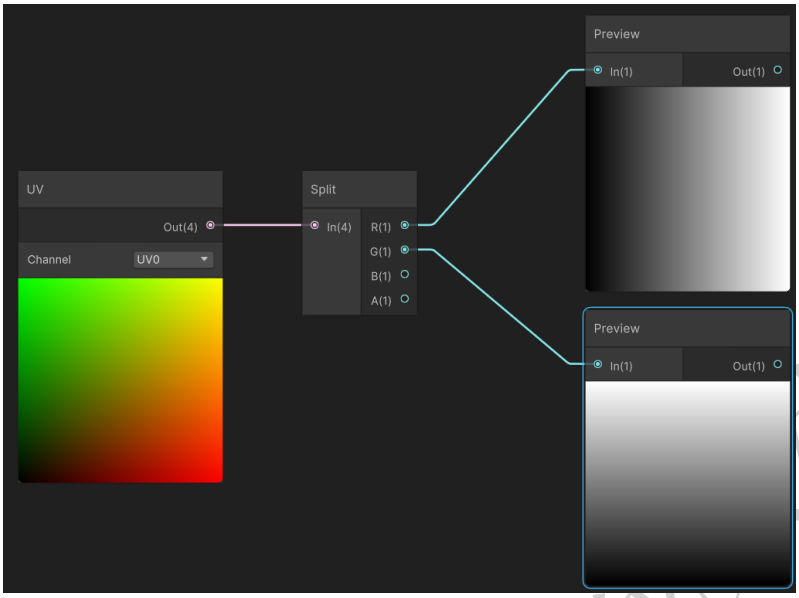


图99.　　图 3.4 预览 UV 的 R 通道和 G 通道

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。

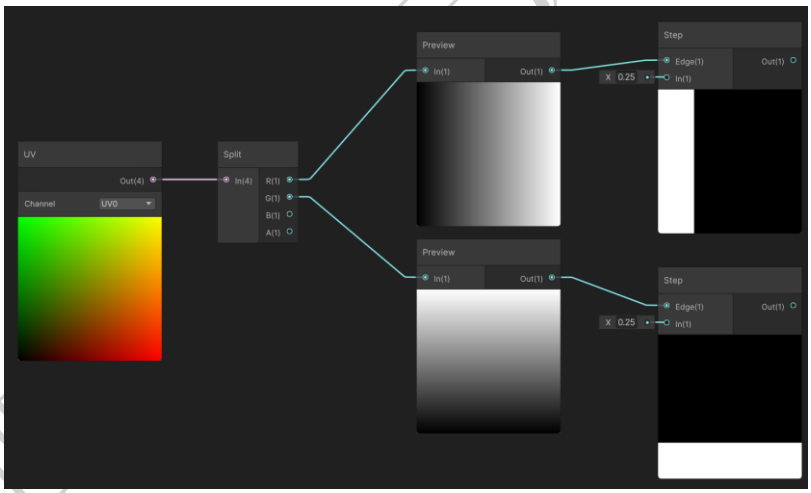


图100.　　图 3.5 用 Step 节点对 R 通道和 G 通道进行切分

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；

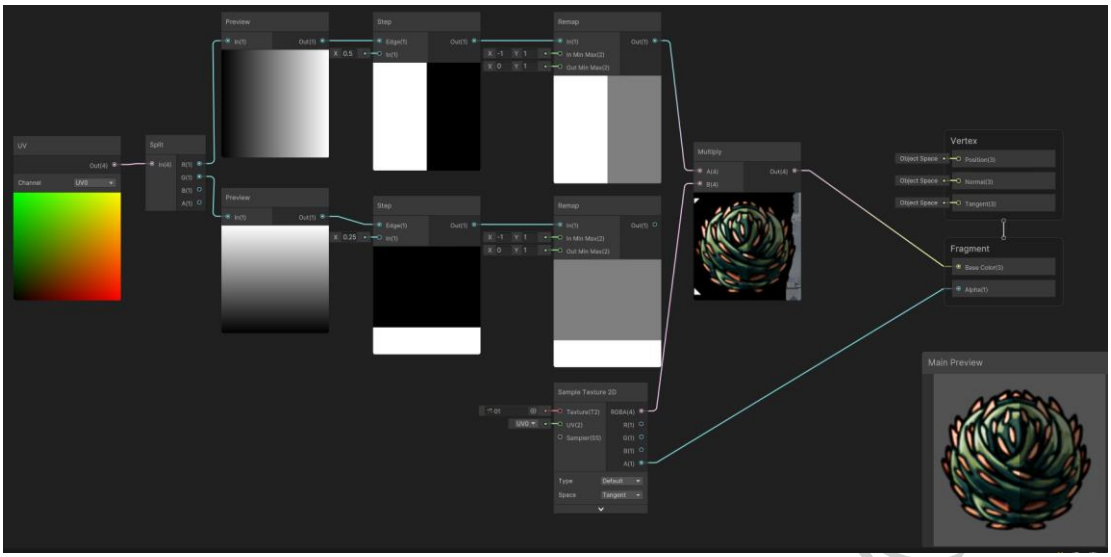


图101.　　图 3.6 重新映射后的最终图像

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102.　　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。

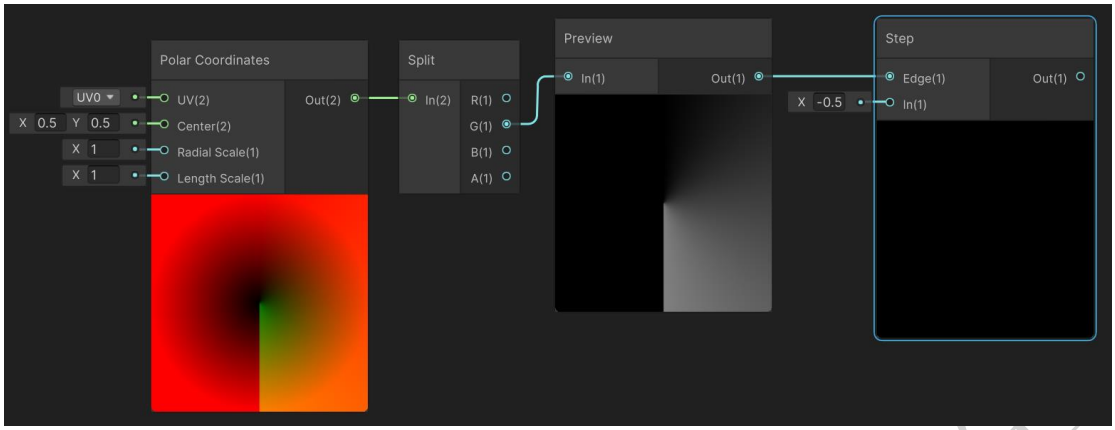


图103.　　图 3.8 使用极坐标实现冷却过渡

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。

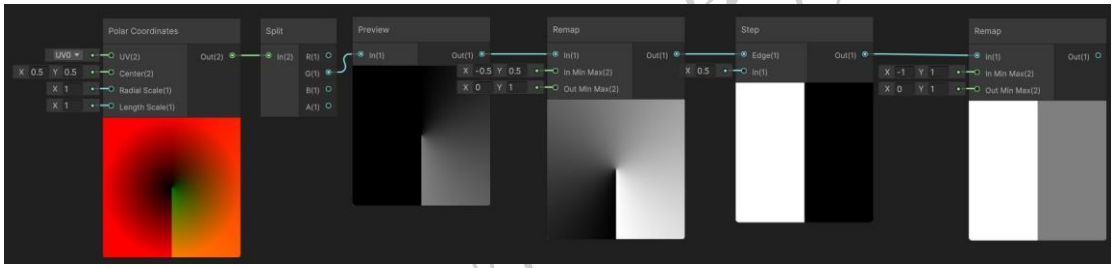


图104.　　图 3.9 使用 Remap 节点重新映射 G 通道值

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

```
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105.　　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。

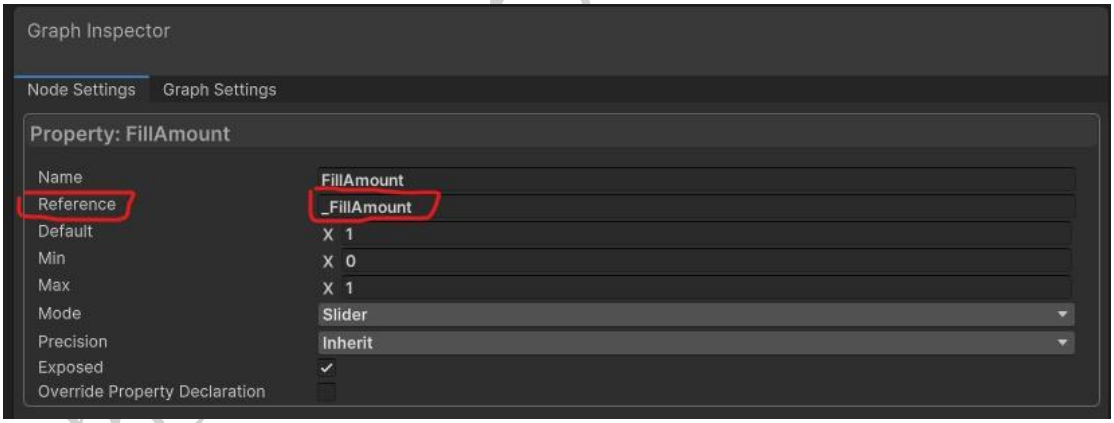


图106.　　图 3.11 Graph Inspector 界面的 Reference

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107.　　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。

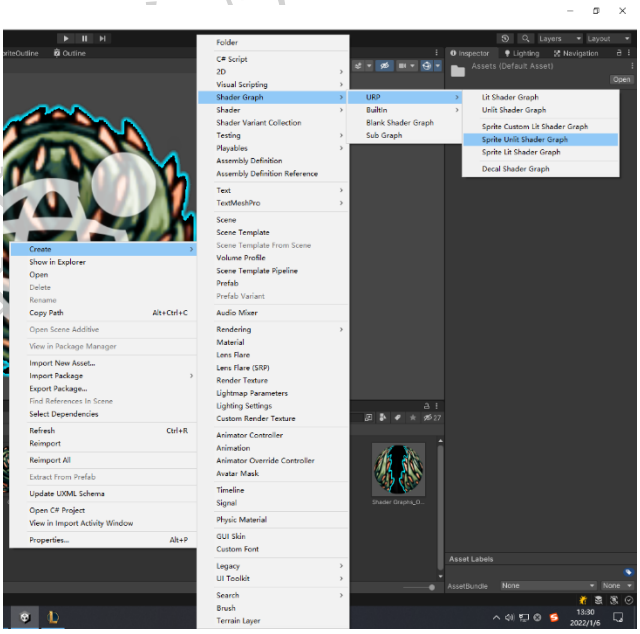


图108.　　图 4.1 创建 Sprite Unlit Shader Graph

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109.　图 4.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

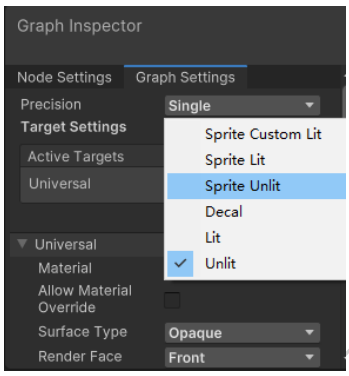


图110.　图 4.3 修改 Material

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。

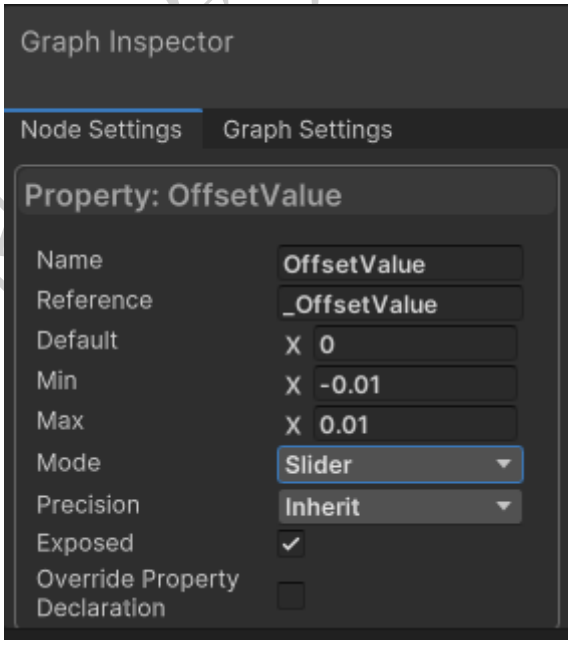


图111.　在 Graph Inspector 中修改参数模式

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

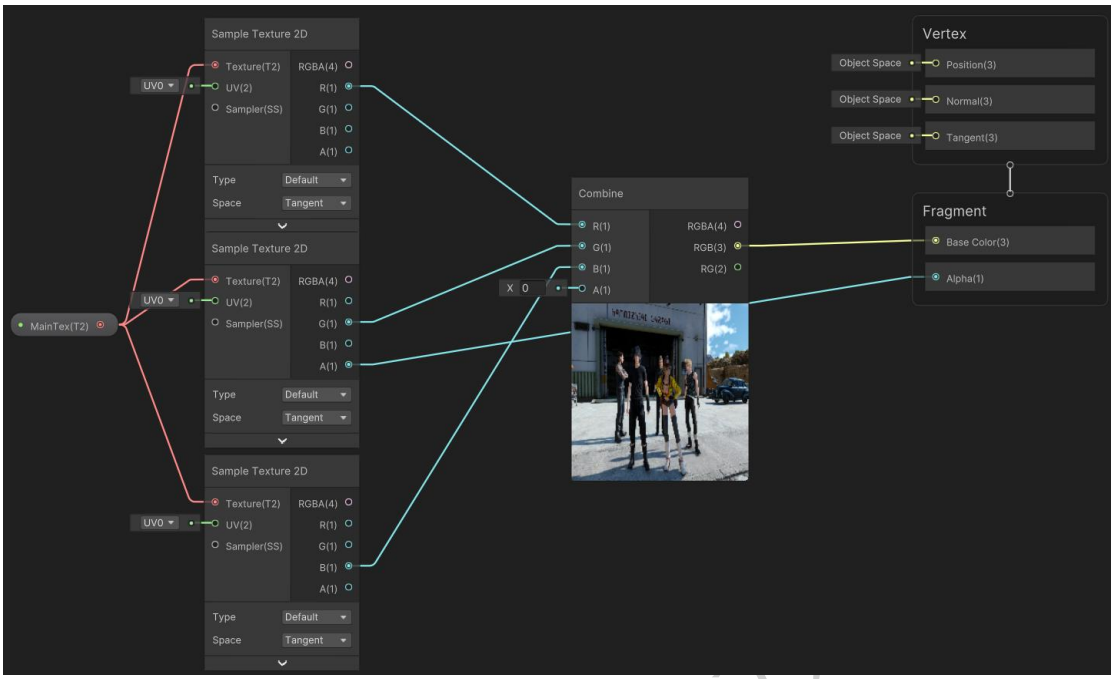


图112.　创建 RGB 分离的采样节点

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

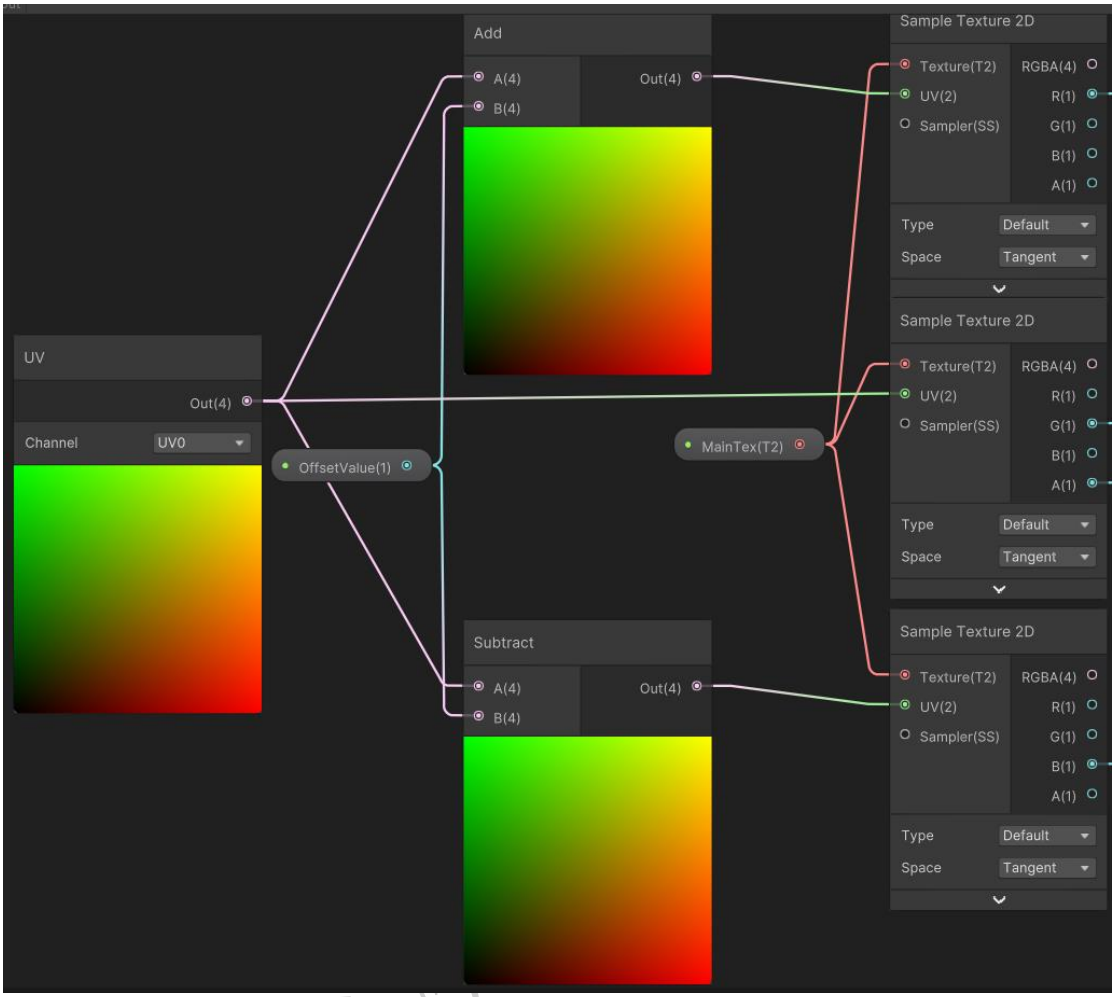


图113.　图 4.6 RGB 分离 UV 偏移

图114.　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。

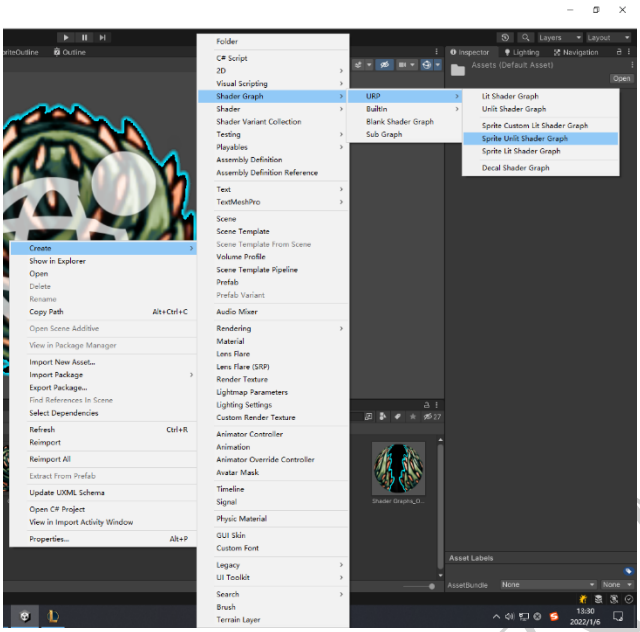


图115.　图 5.1 创建 Sprite Unlit Shader Graph

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116.　图 5.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

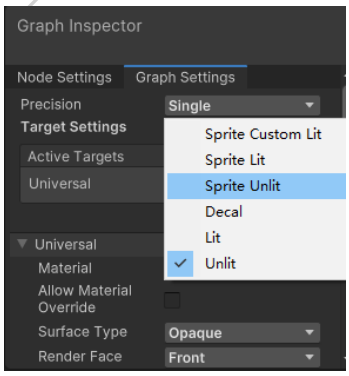


图117.　图 5.3 修改 Material

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。

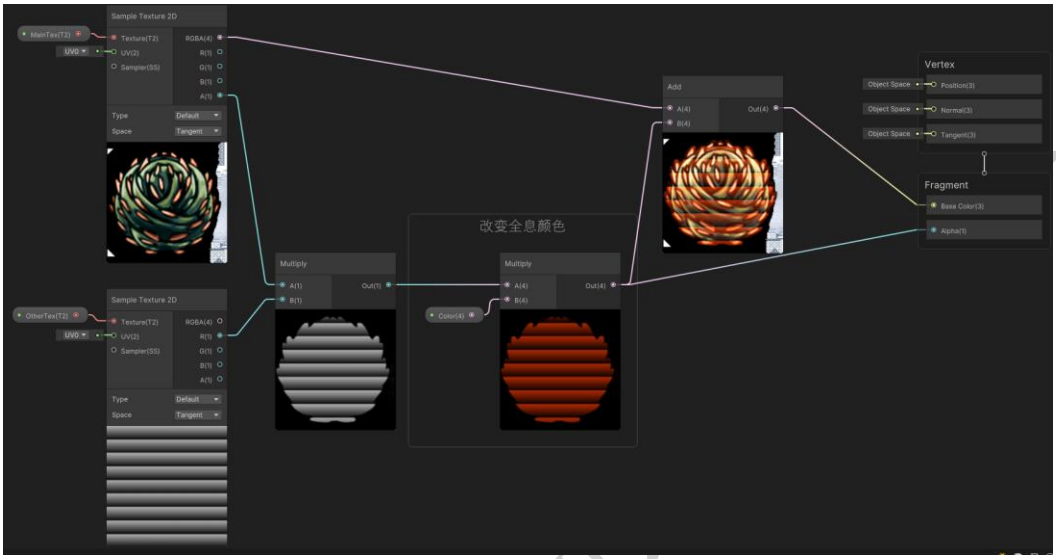


图118.　　图 5.4 创建带全息效果的图像

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。

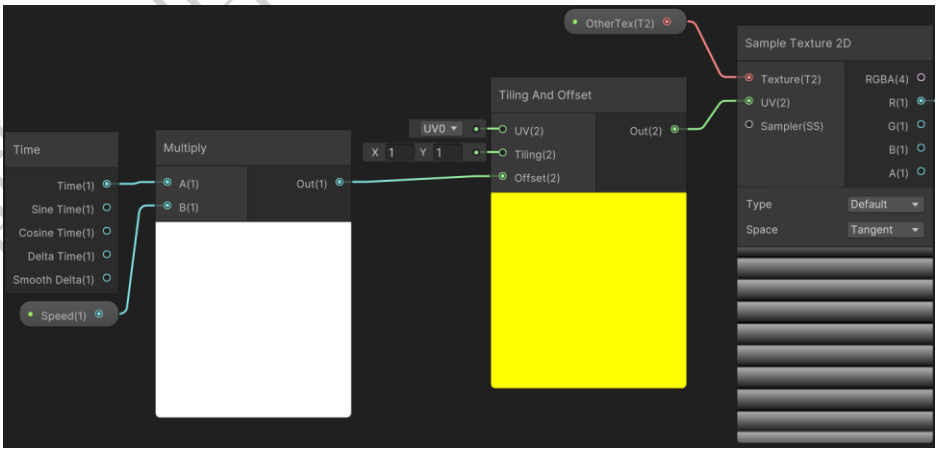


图119.　　图 5.5 让全息贴图流动

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。

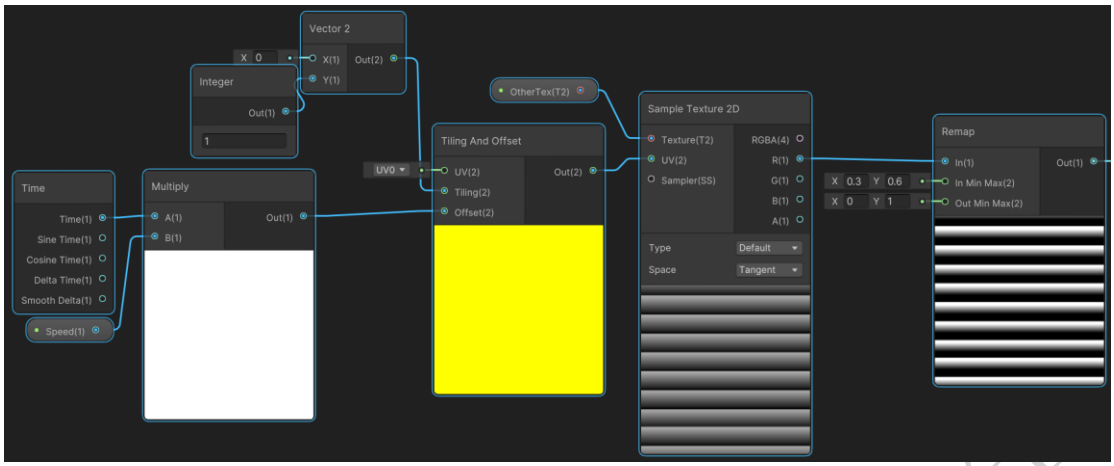


图120.　　图 5.6 调节全息范围、强度和密度

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121.　　图 5.6 使用 Clamp 节点对值进行限制

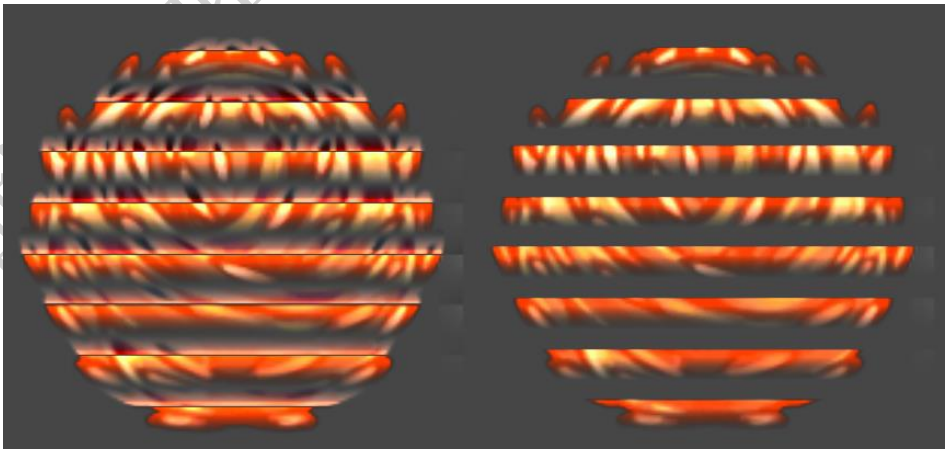


图122.　　图 5.7 调节前（左）和调节后（右）对比

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。

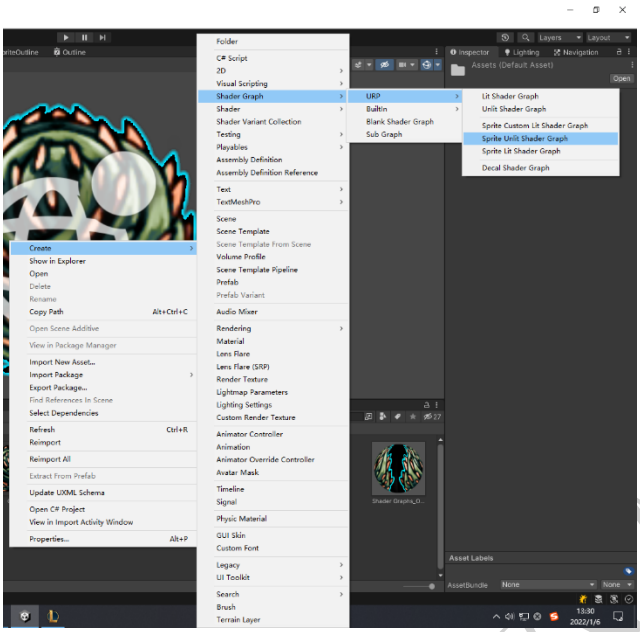


图123.　图 6.1 创建 Sprite Unlit Shader Graph

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124.　图 6.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

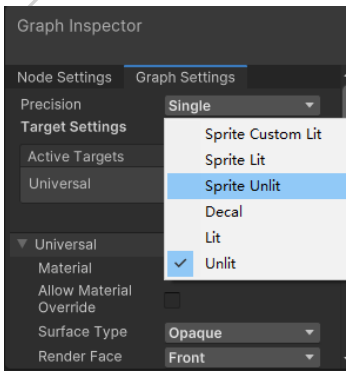


图125.　图 6.3 修改 Material

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

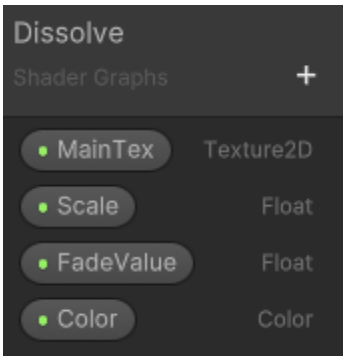


图126.　　图 6.4 可外部访问的参数

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。

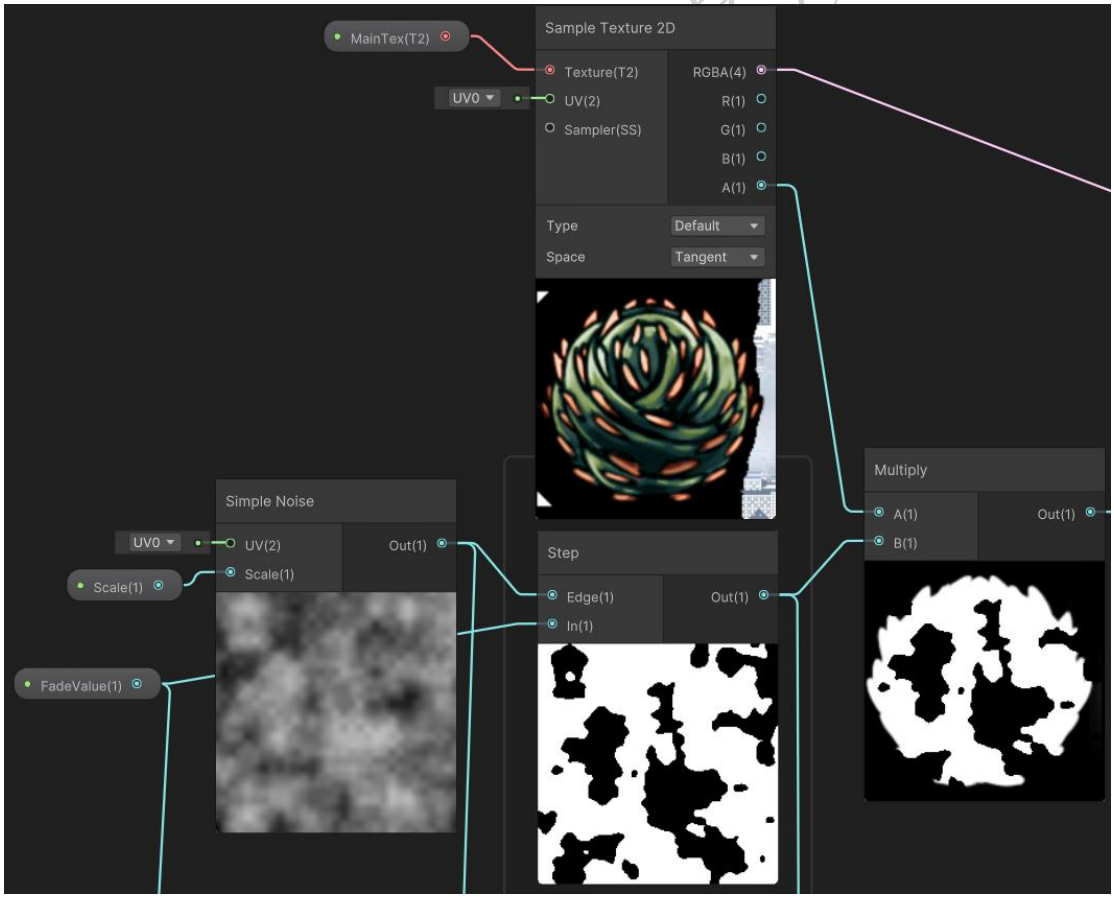


图127.　　图 6.5 当前溶解度对应的溶解效果

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。

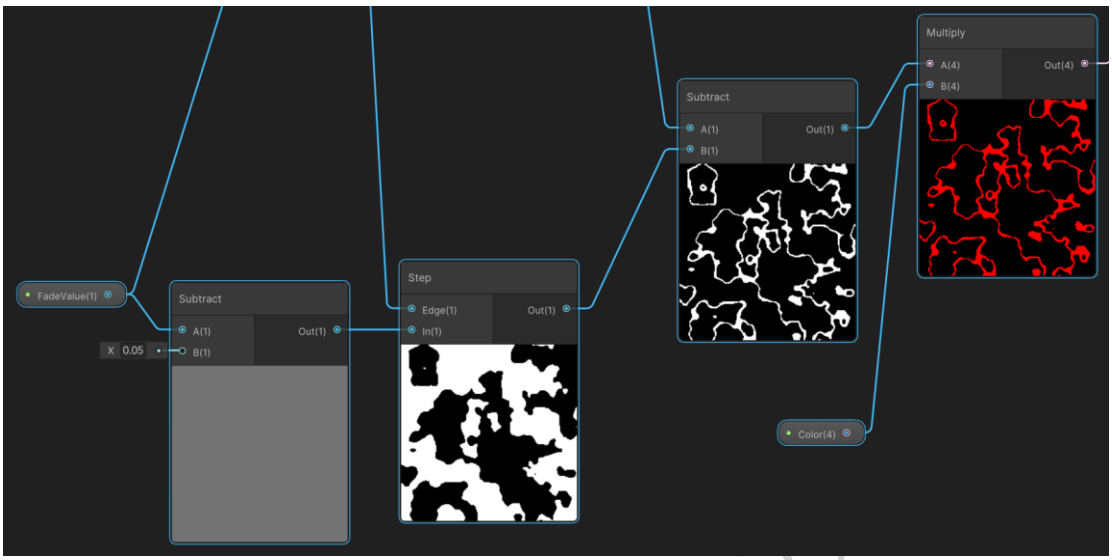


图128.　　图 6.6 溶解效果内边缘描边

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。

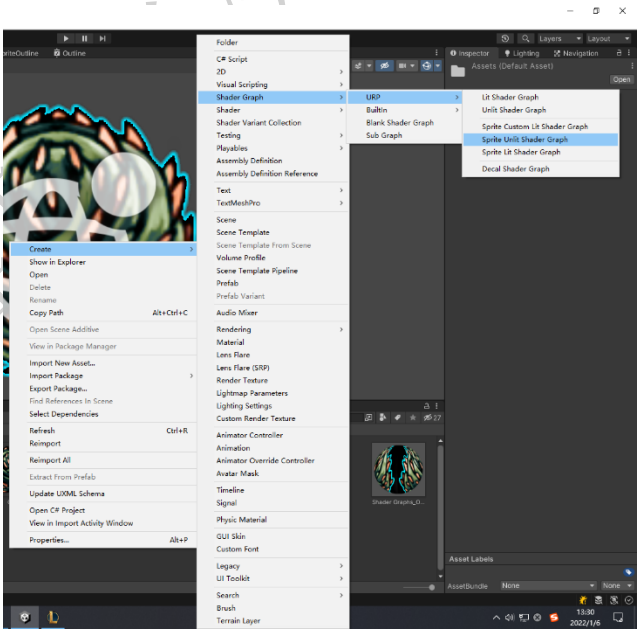


图129.　　图 7.1 创建 Sprite Unlit Shader Graph

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130.　　图 7.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

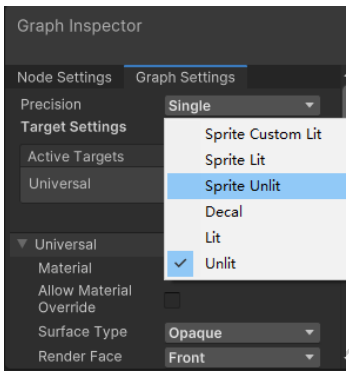


图131.　　图 7.3 修改 Material

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。

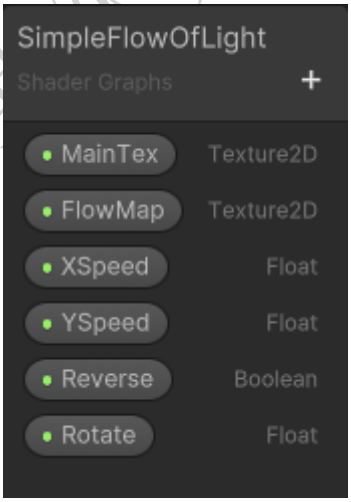


图132.　　图 7.4 2D 流光简易版参数

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

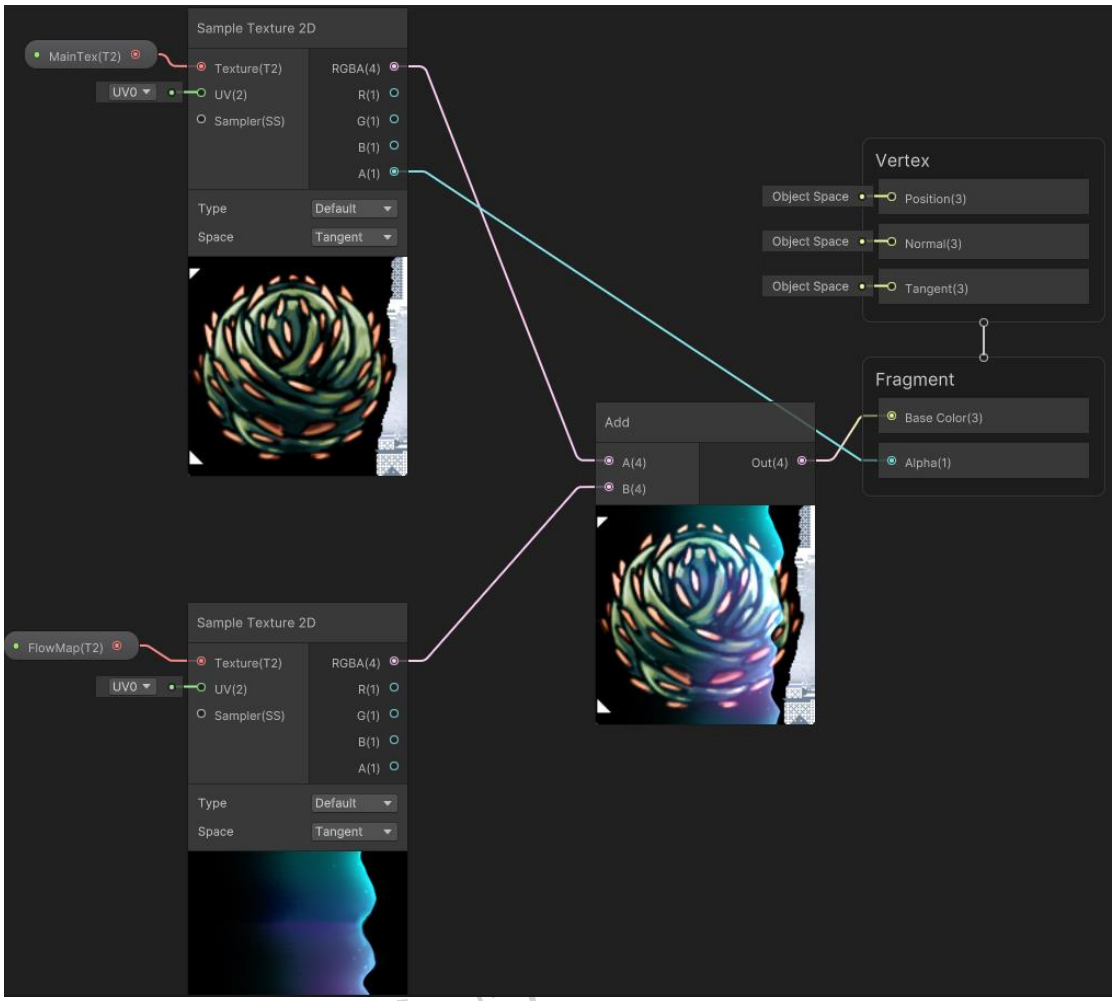


图133.　　图 7.5 含有流光效果的图像

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134.　　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。

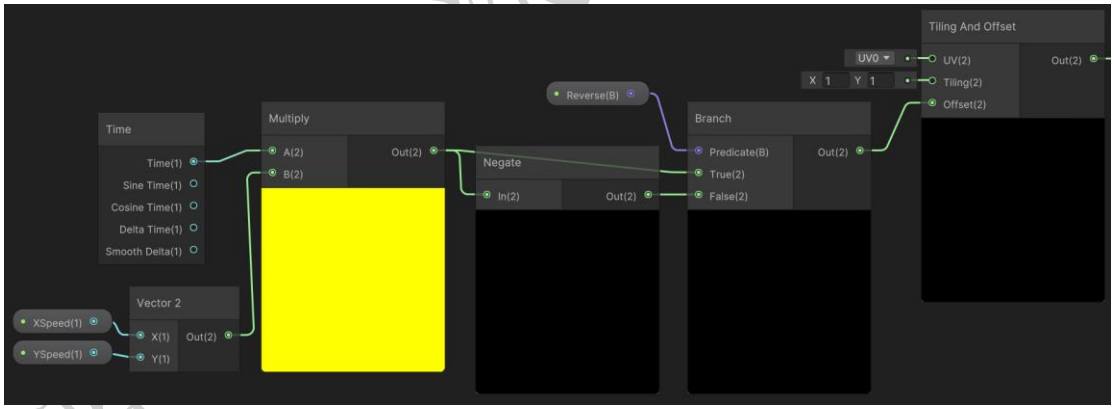


图135.　　图 7.7 流光反向运动

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）。

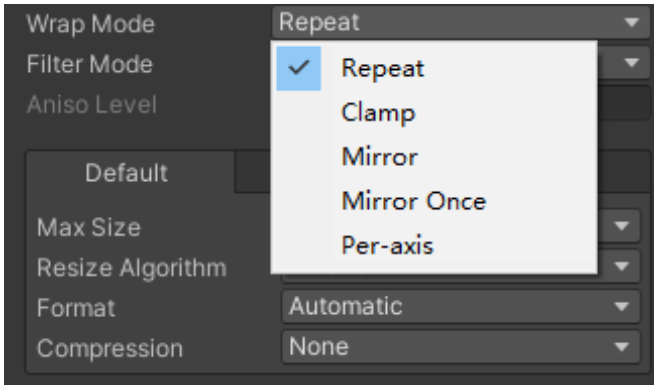


图136.　图检查纹理设置

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。

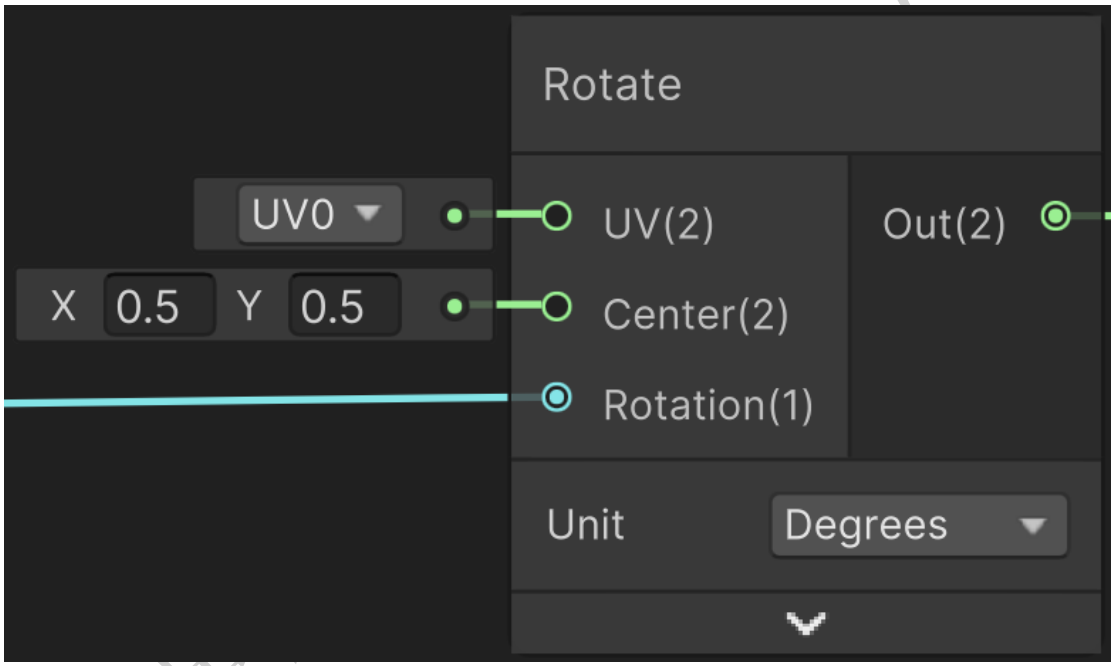


图137.　图 Rotate 节点控件

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。

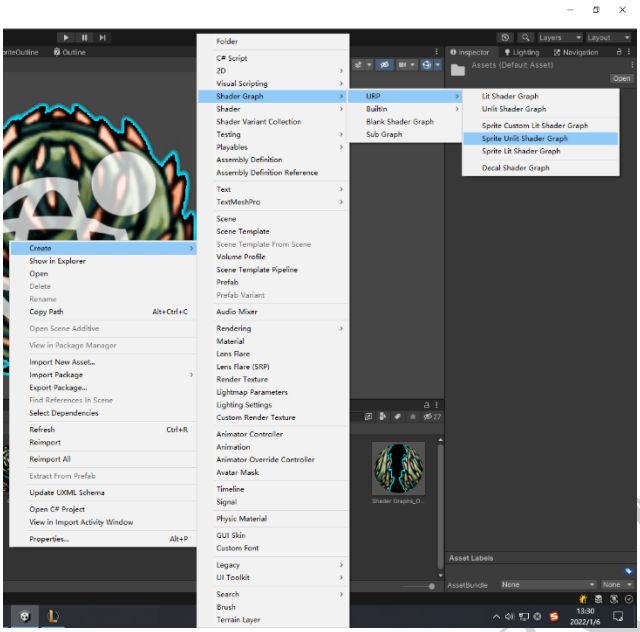


图138.　　图 7.1 创建 Sprite Unlit Shader Graph

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139.　　图 7.2 Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

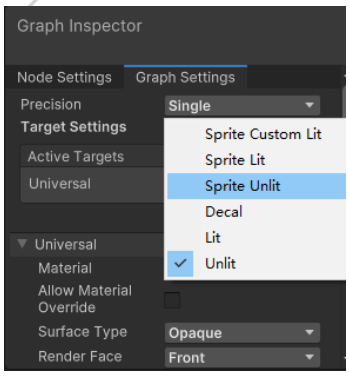


图140.　　图 7.3 修改 Material

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

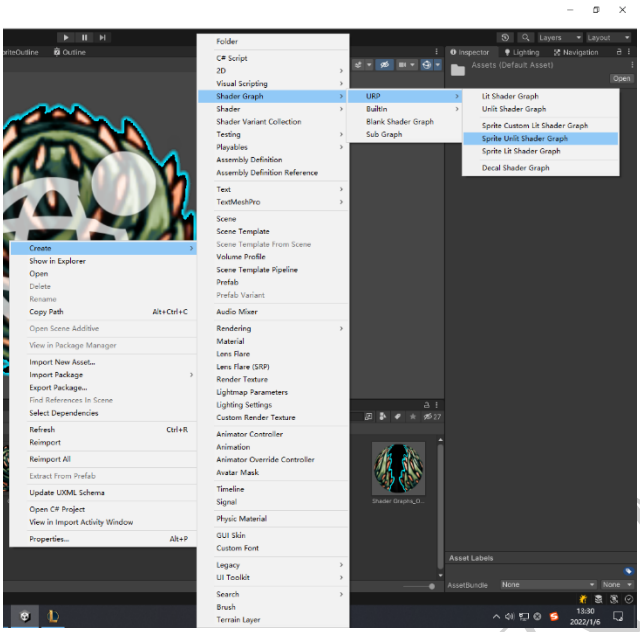


图1. 创建 Sprite Unlit Shader Graph

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141. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

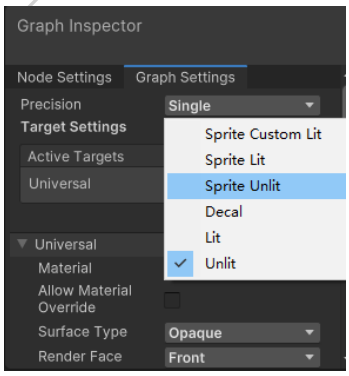


图142. 修改 Material

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。

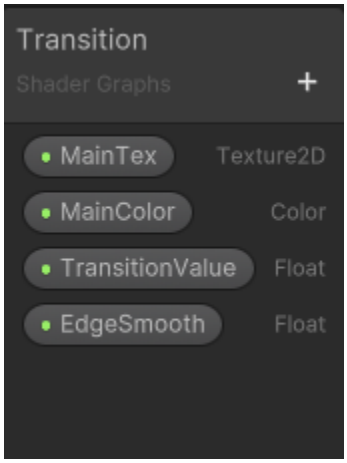


图143.　过渡效果参数

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。

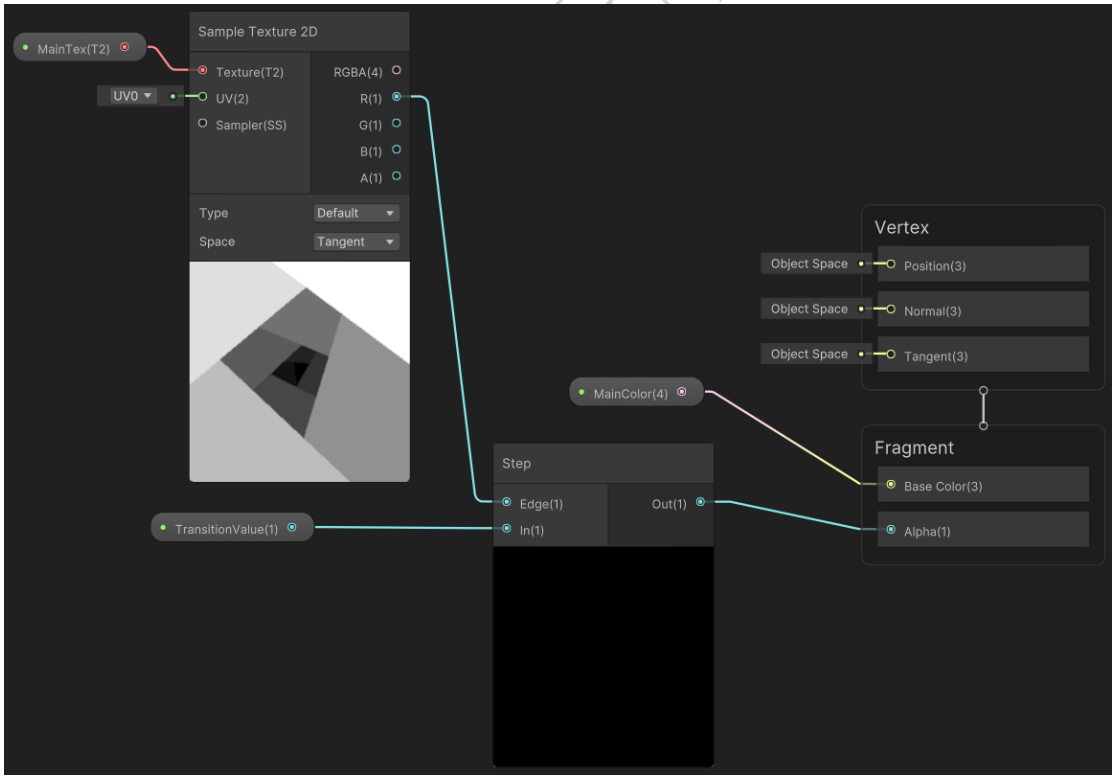


图144.　硬性过渡

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。

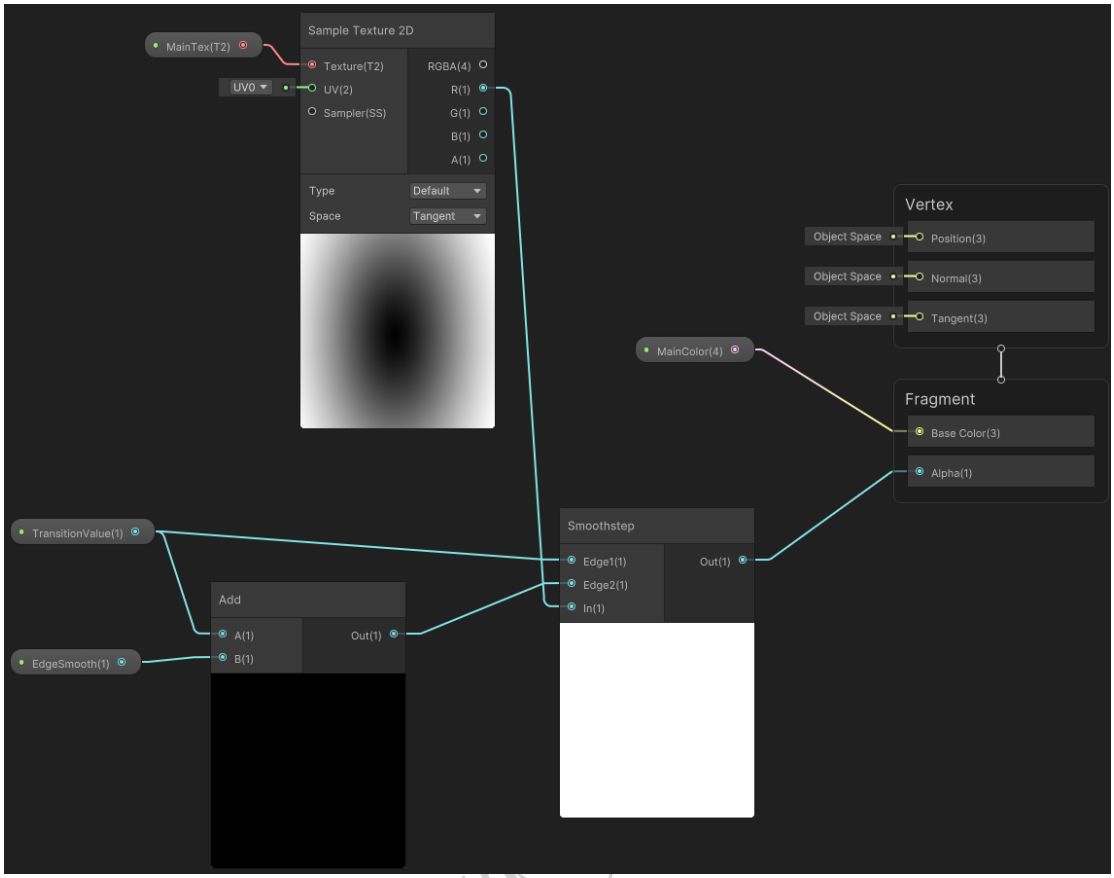


图145. 柔性过渡

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

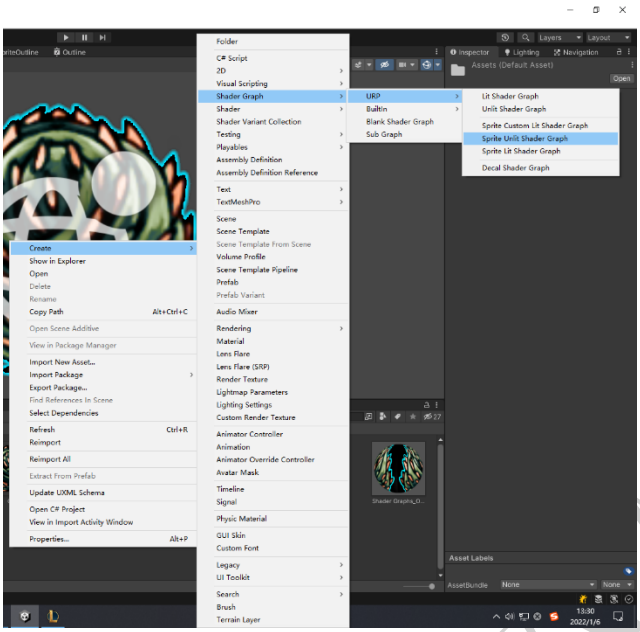


图2. 创建 Sprite Unlit Shader Graph

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146. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

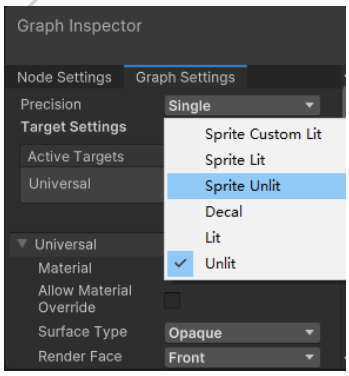


图147. 修改 Material

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

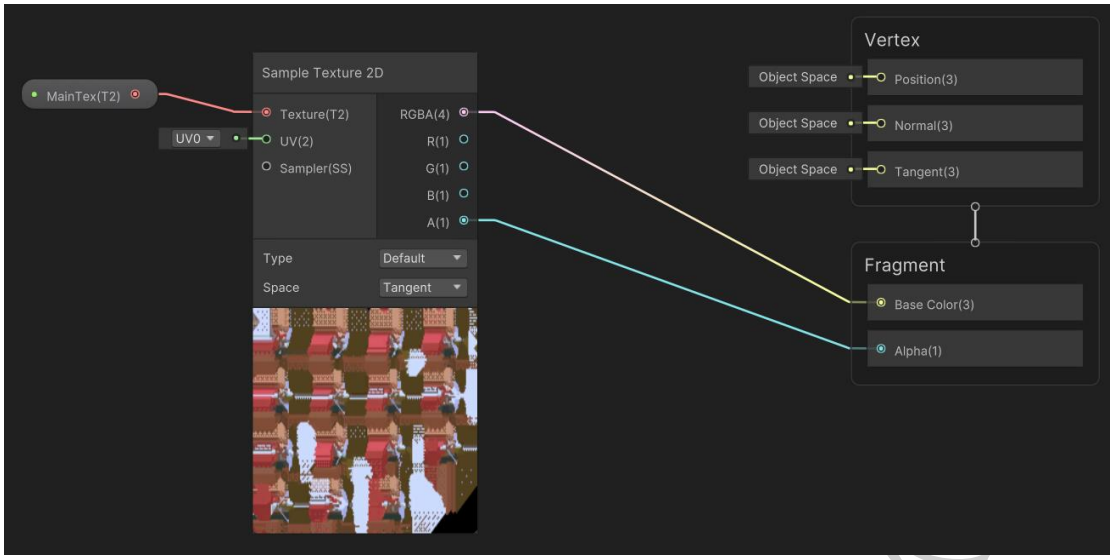


图148.　2D 帧动画主纹理

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。

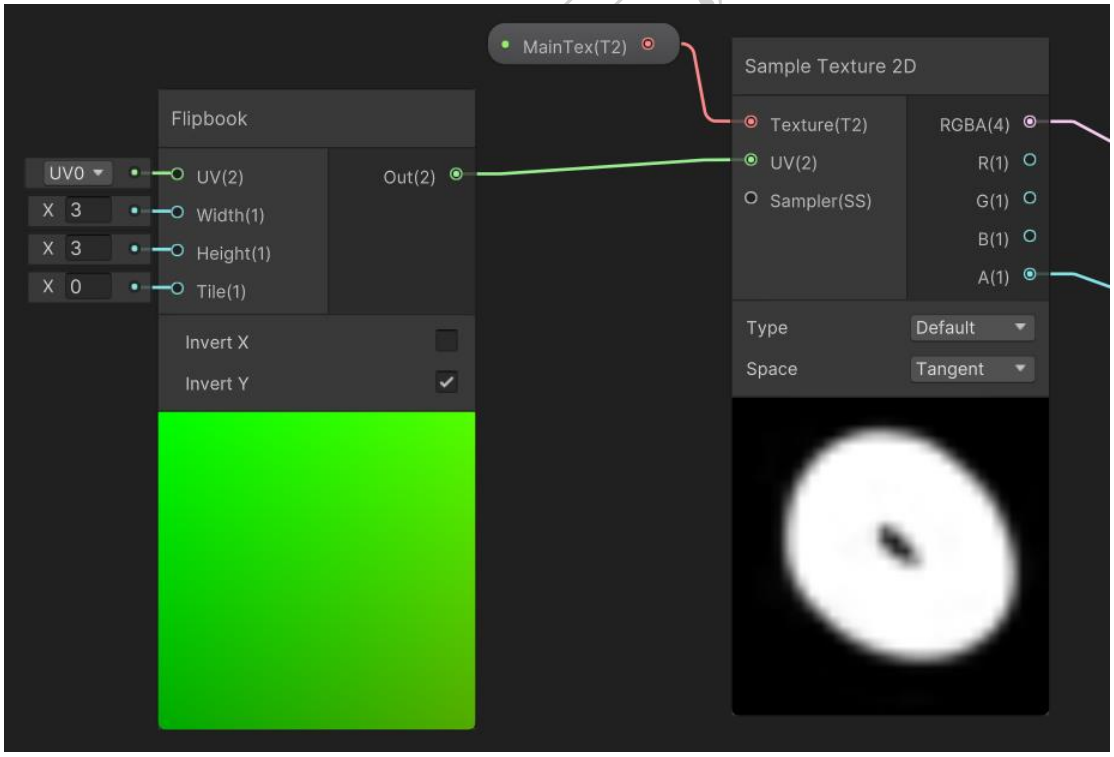


图149.　使用 Flipbook 节点

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。

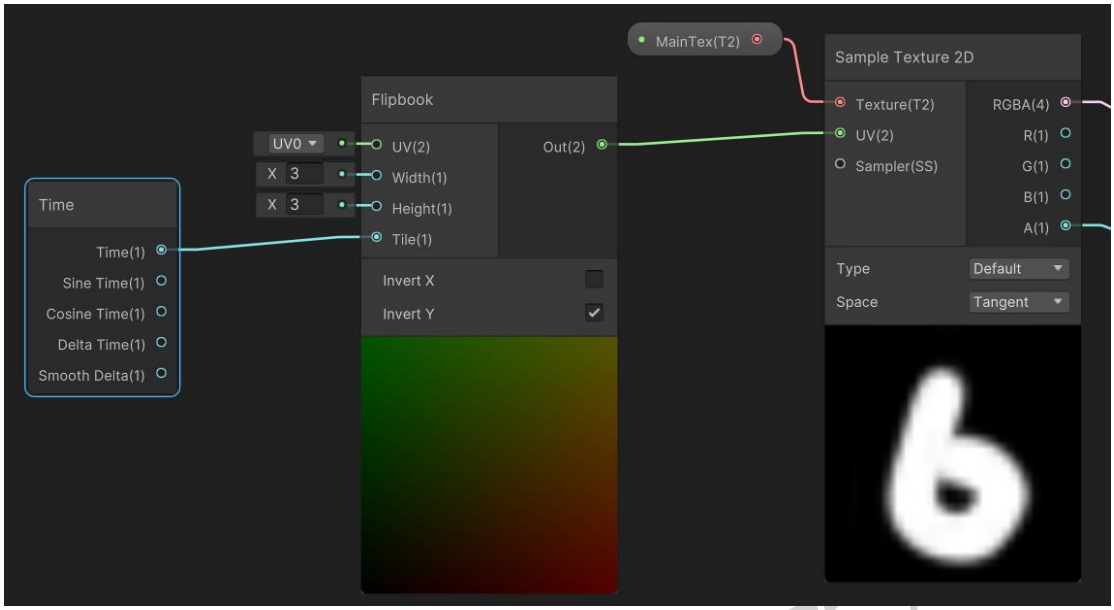


图150.　使用 Time 节点让瓦片依次显示

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。

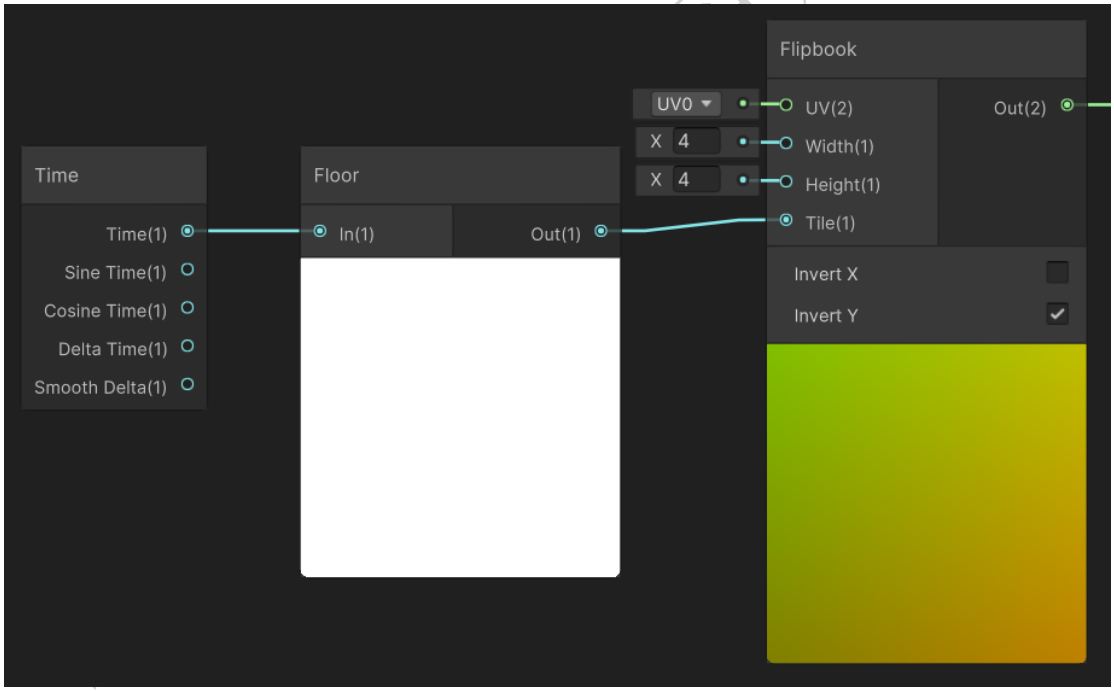


图151.　对 Time 节点输出取整

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。

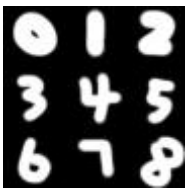


图152.　测试图

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。

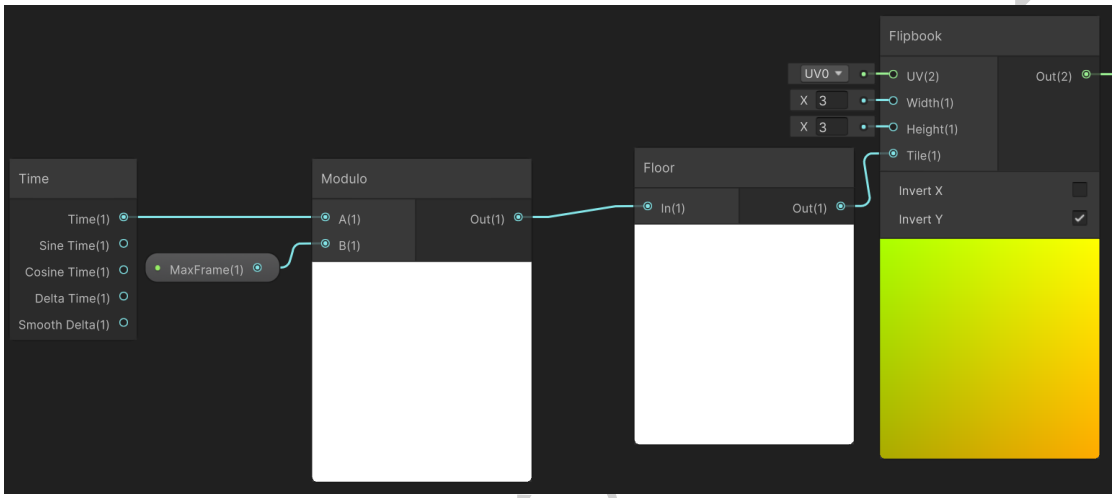


图153.　对 Time 节点输出取余

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。

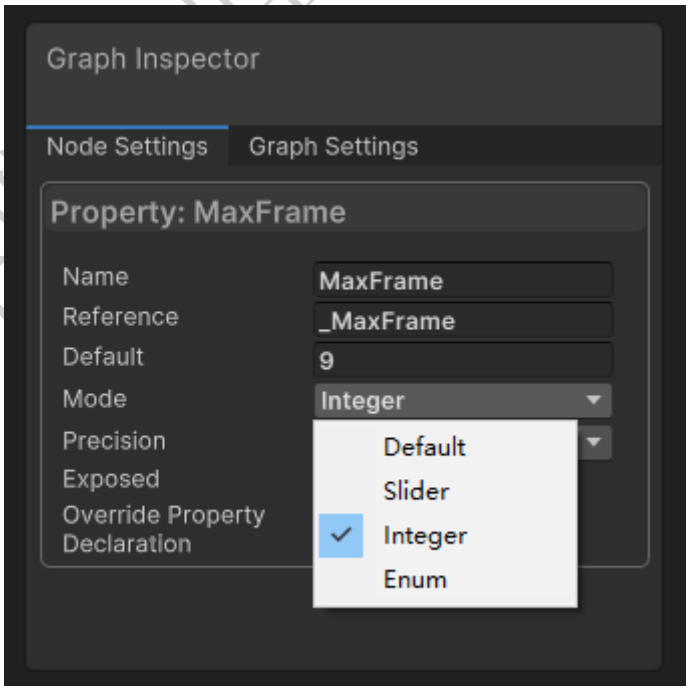


图154.　修改参数类型为整型

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。

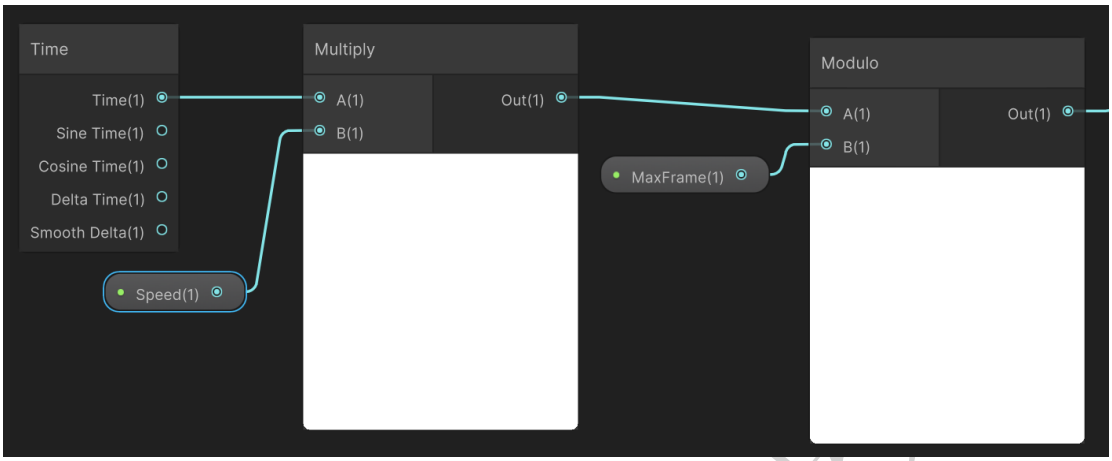


图155. 调节帧动画速度

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。

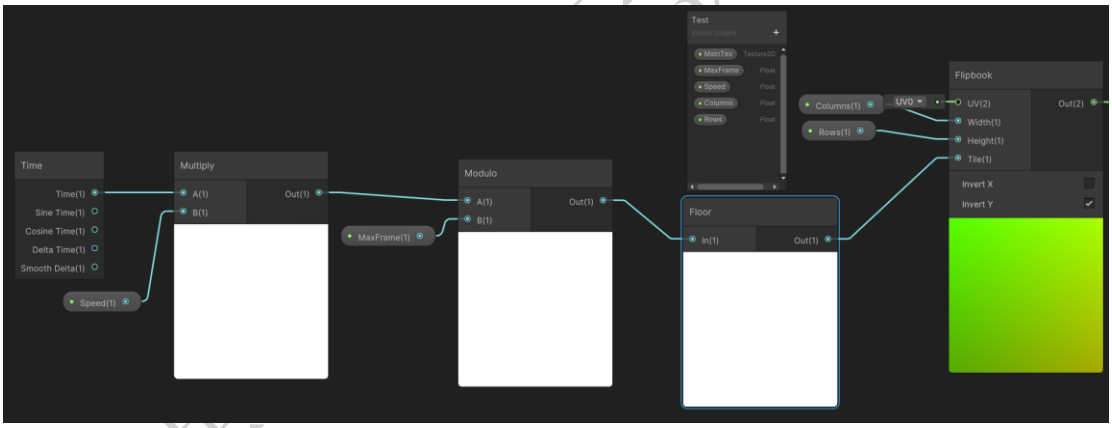


图156. 添加行和列参数

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。

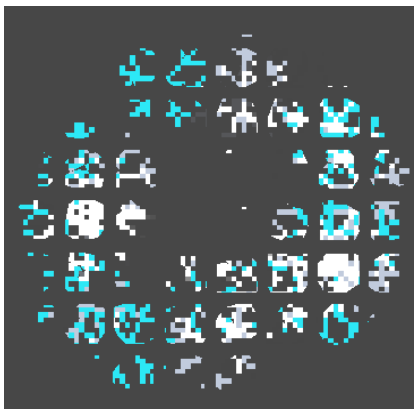


图157. 显示图形出错

要让图片正确显示，我们需要修改该纹理的 Mesh Type 为 Full Rect。

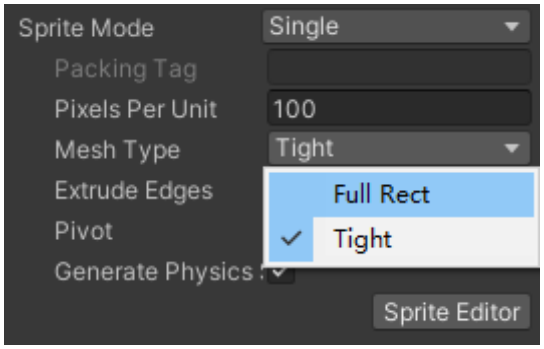


图158.　修改 Mesh Type

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。

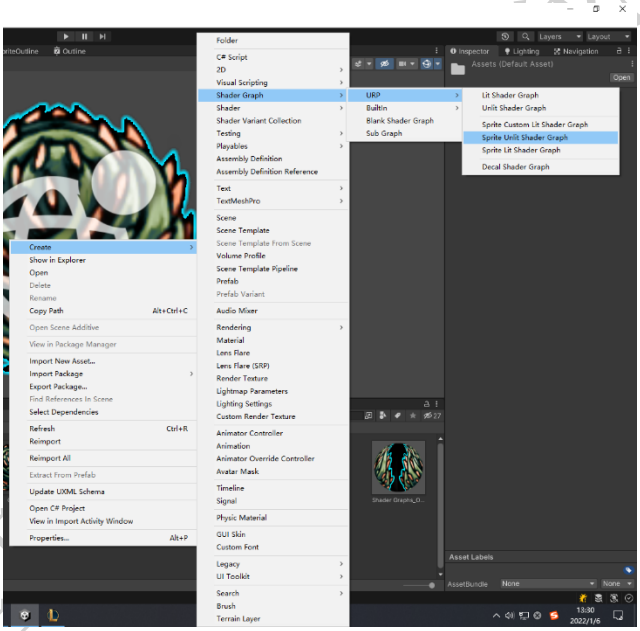


图3. 创建 Sprite Unlit Shader Graph

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159.　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

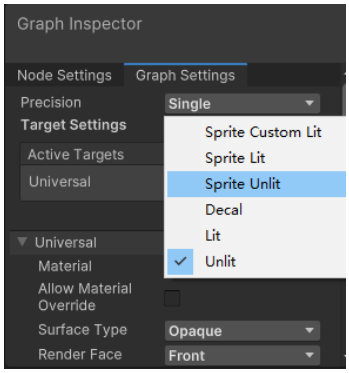


图160.　　修改 Material

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。

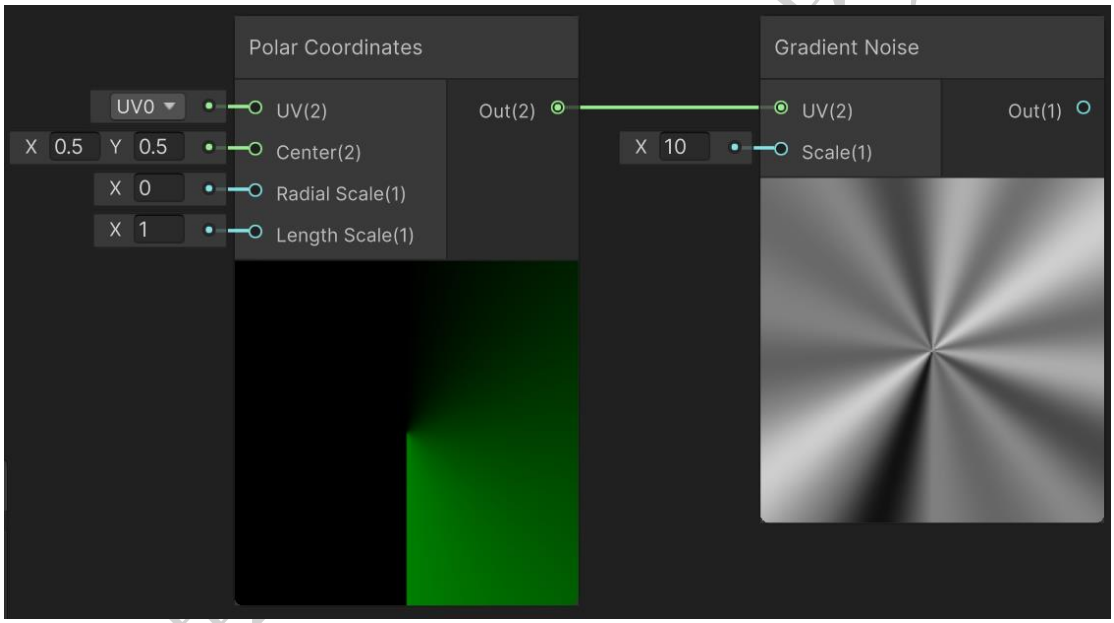


图161.　　速度线雏形

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。

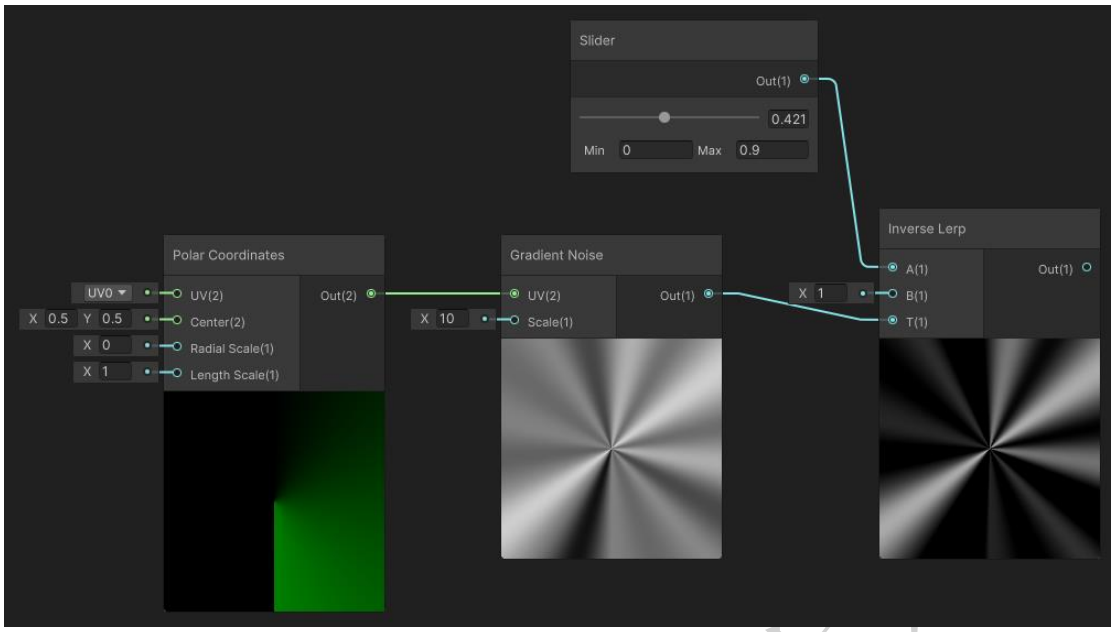


图162.　　对速度线图像进行过滤

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。

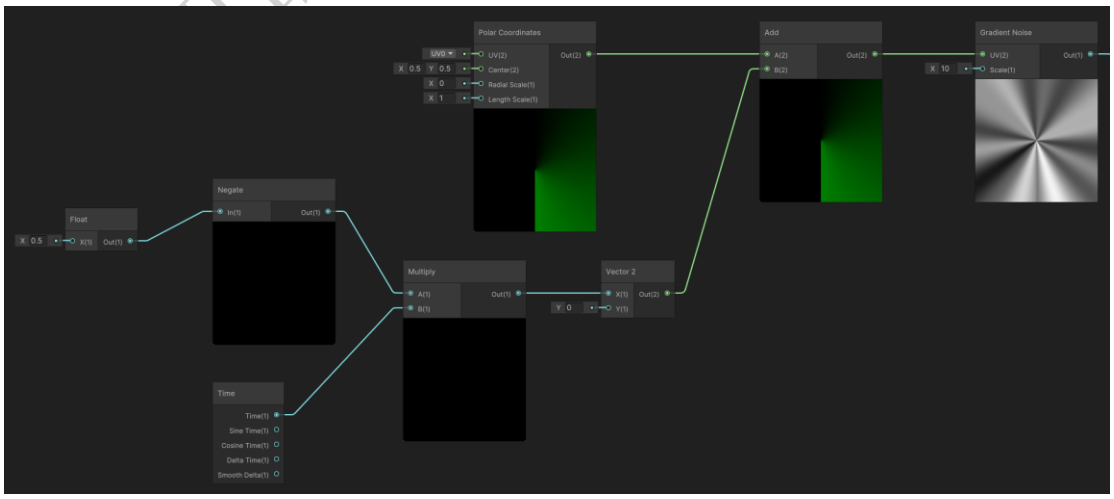


图163.　　速度线运动

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。

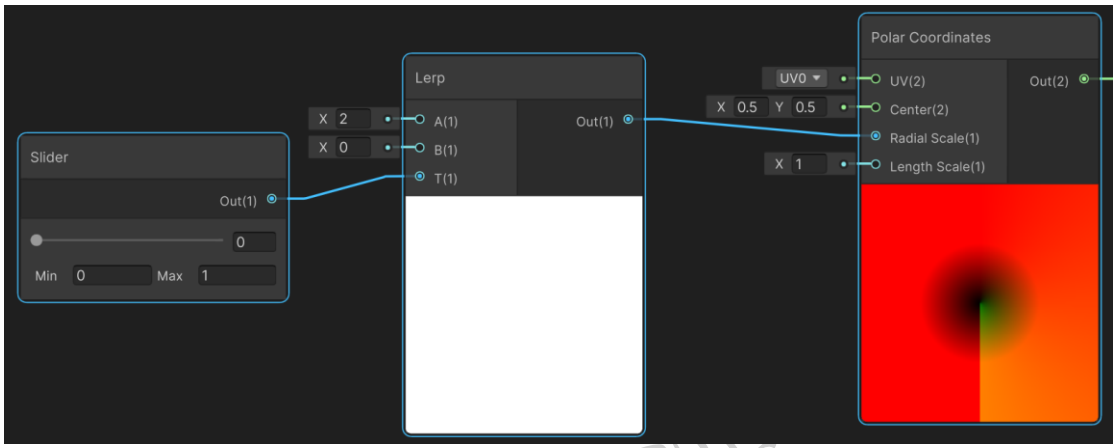


图164.　Mask 显示规模

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。

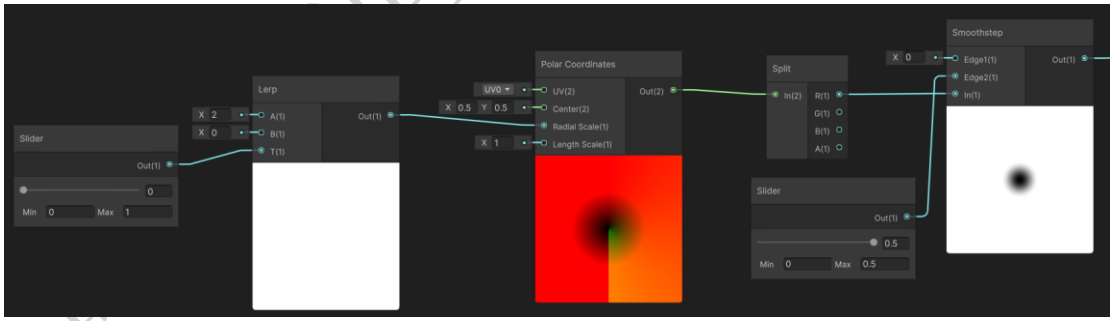


图165.　速度线蒙版

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。

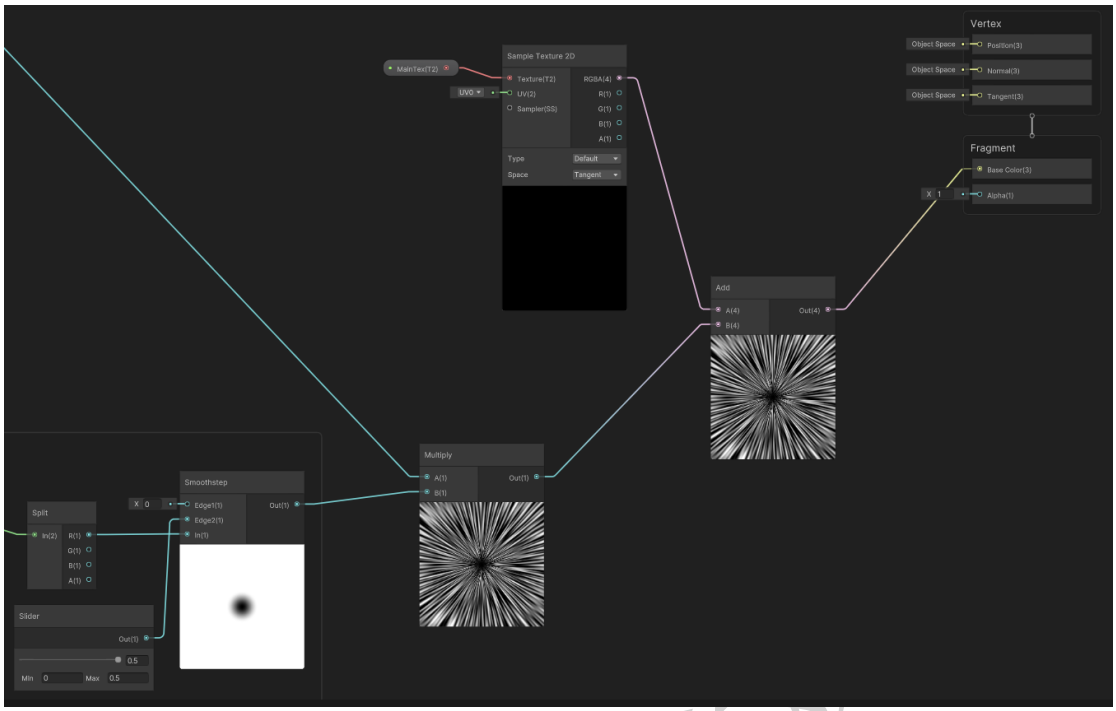


图166.　　速度线最终输出

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。

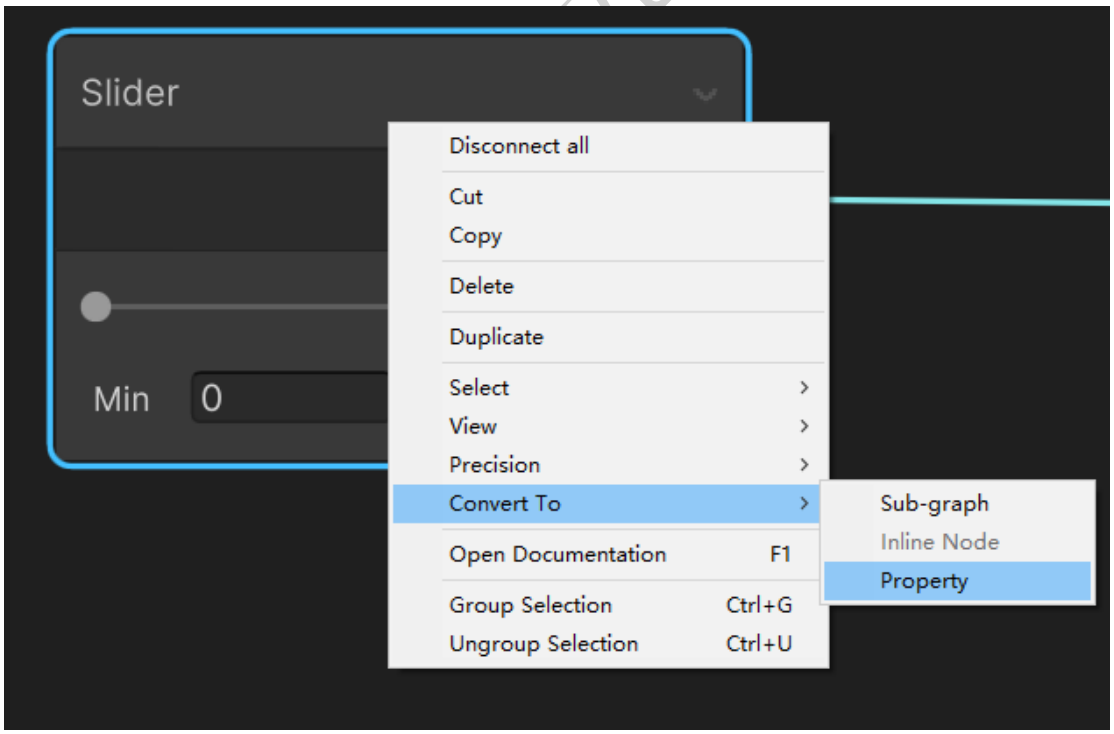


图167.　　转换为参数

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。

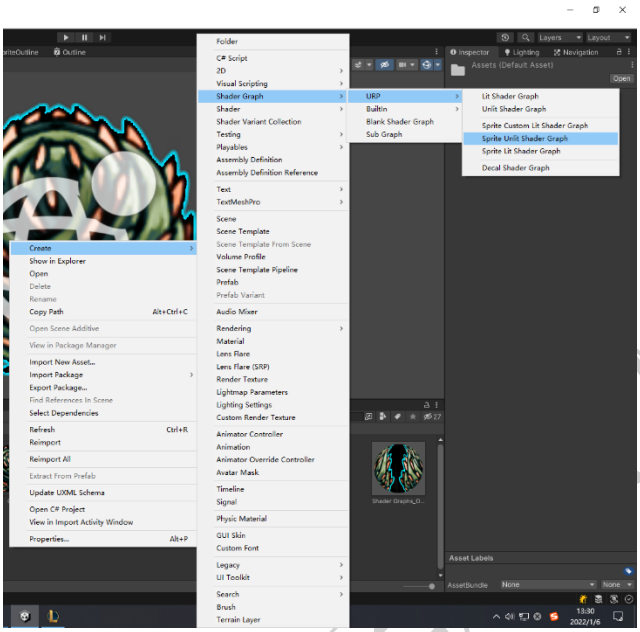


图4. 创建 Sprite Unlit Shader Graph

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168. Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

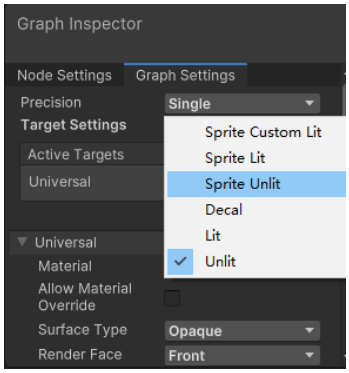


图169. 修改 Material

[13] adj. 普遍的；通用的 n. 普遍原则；普遍看法

2. 原理解释

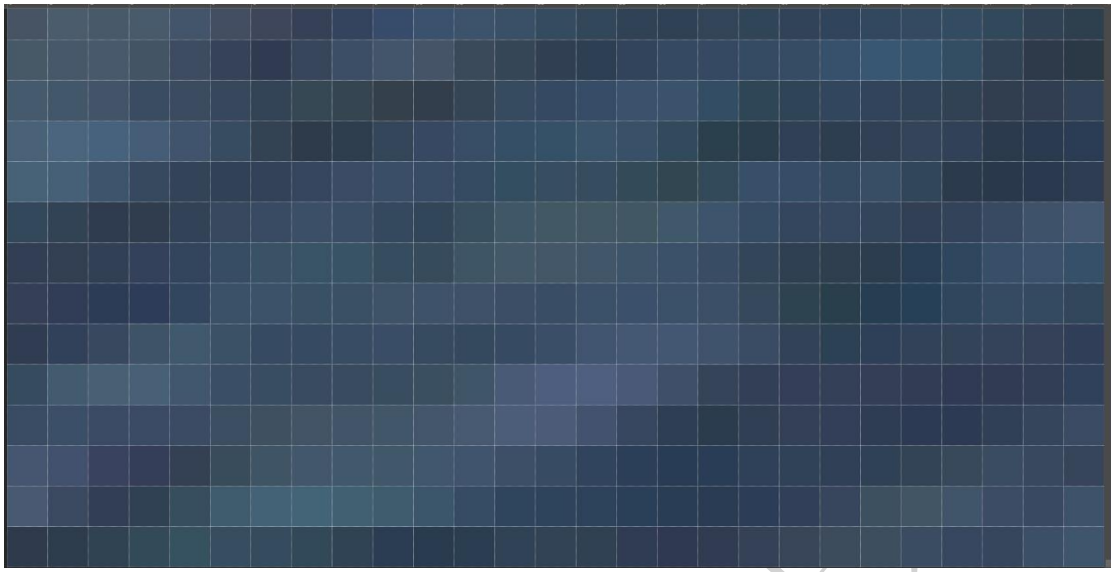


图170. 像素点

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。

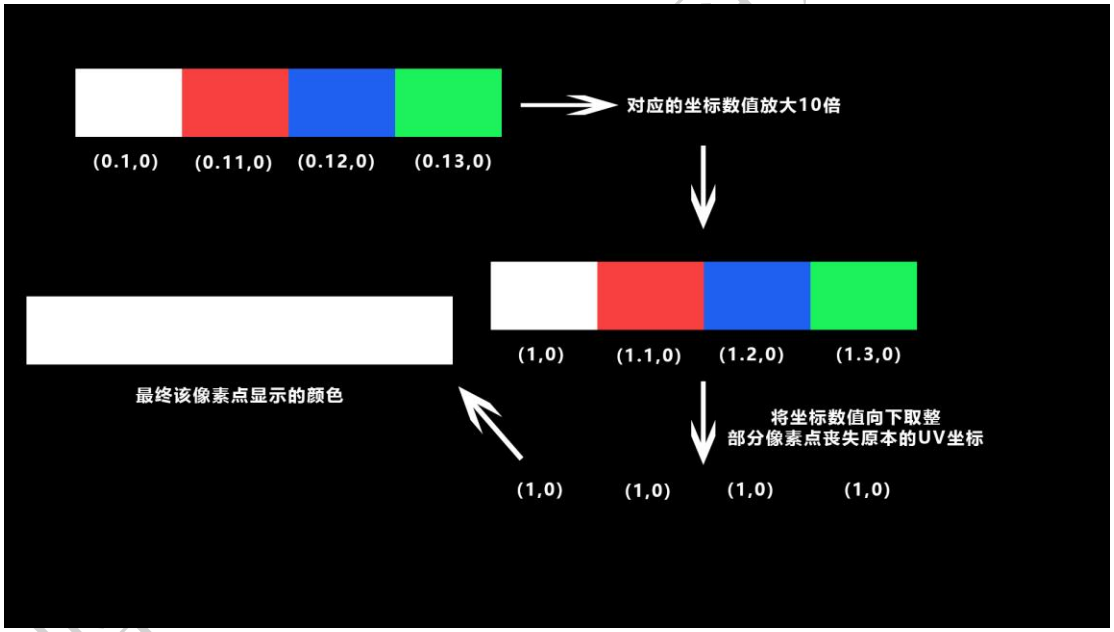


图171. 像素化原理

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。

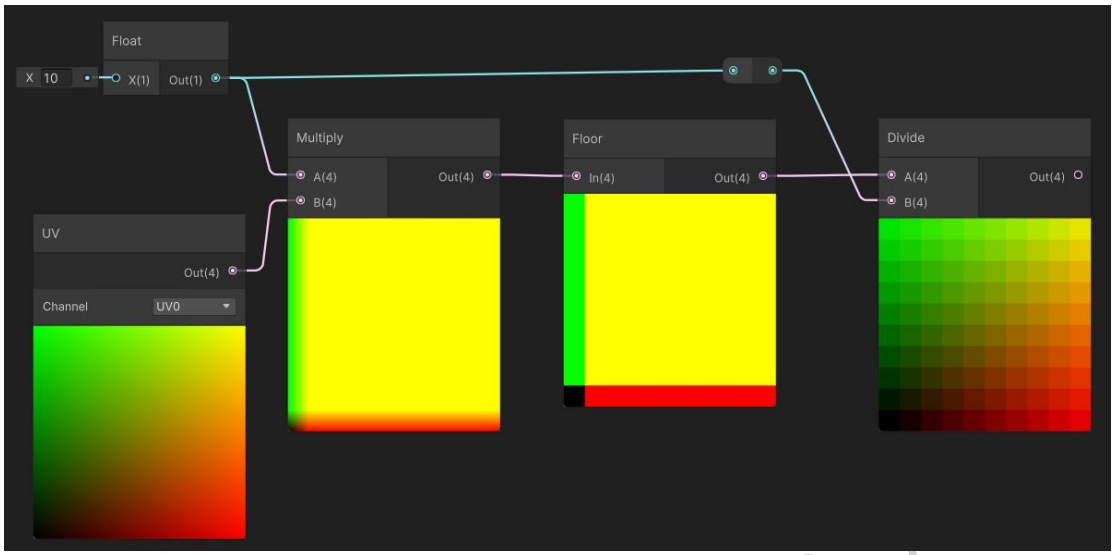


图172.　　修改 UV 坐标

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。

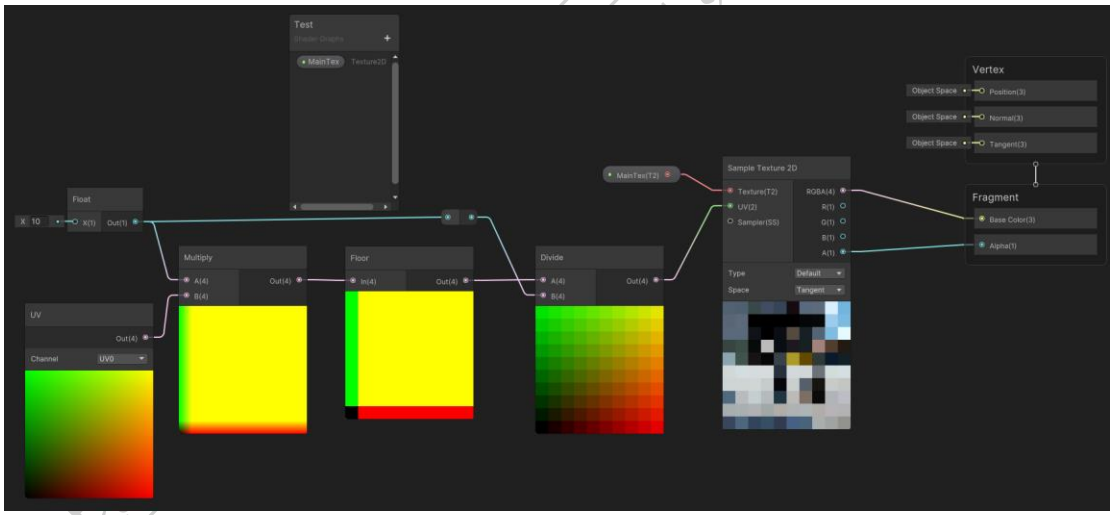


图173.　　输出

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。

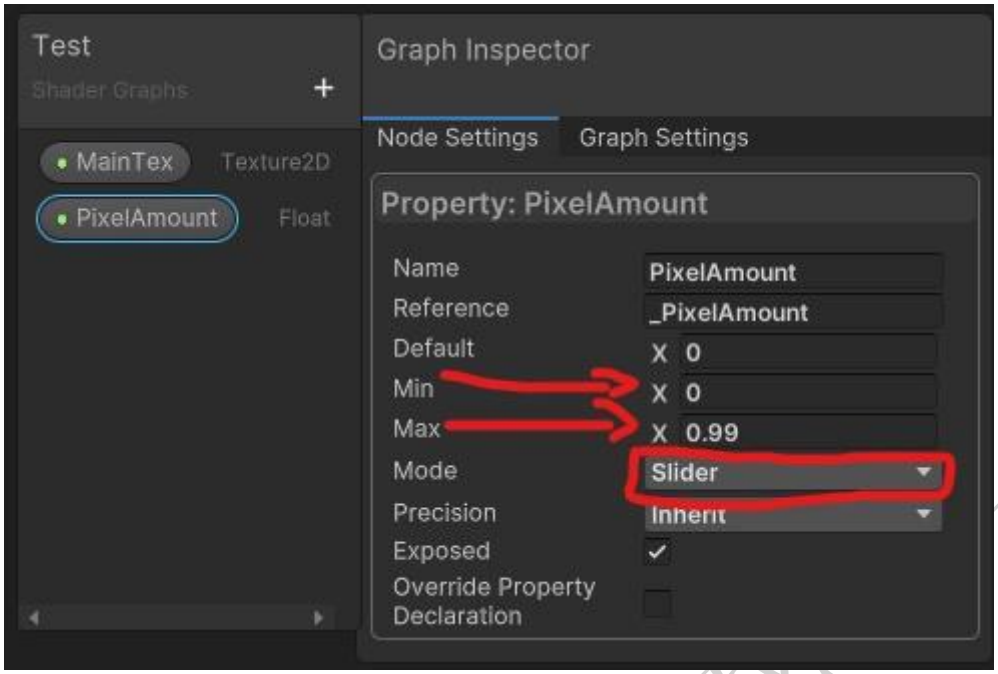


图174.　　修改参数模式为 Slider

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。

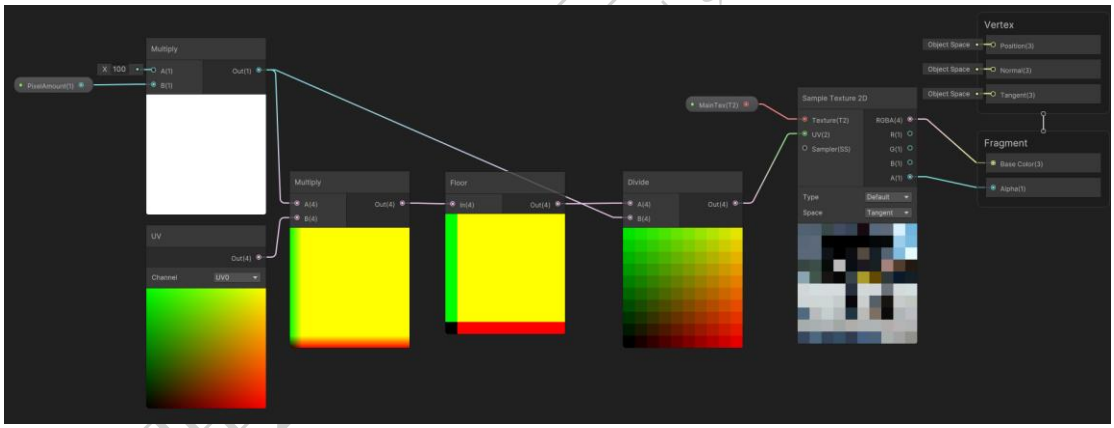


图175.　　使用参数控制

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）

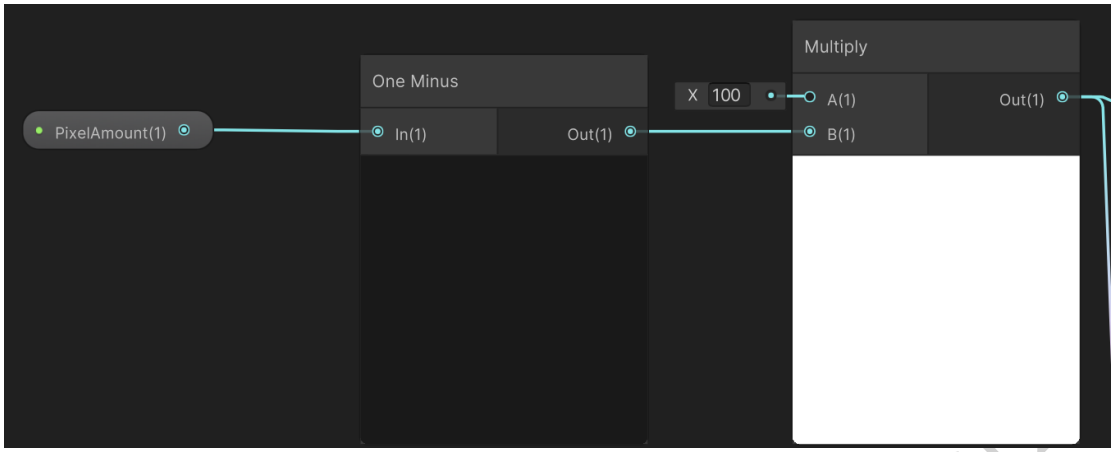


图176.　　使用 OneMinus 修改参数

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。

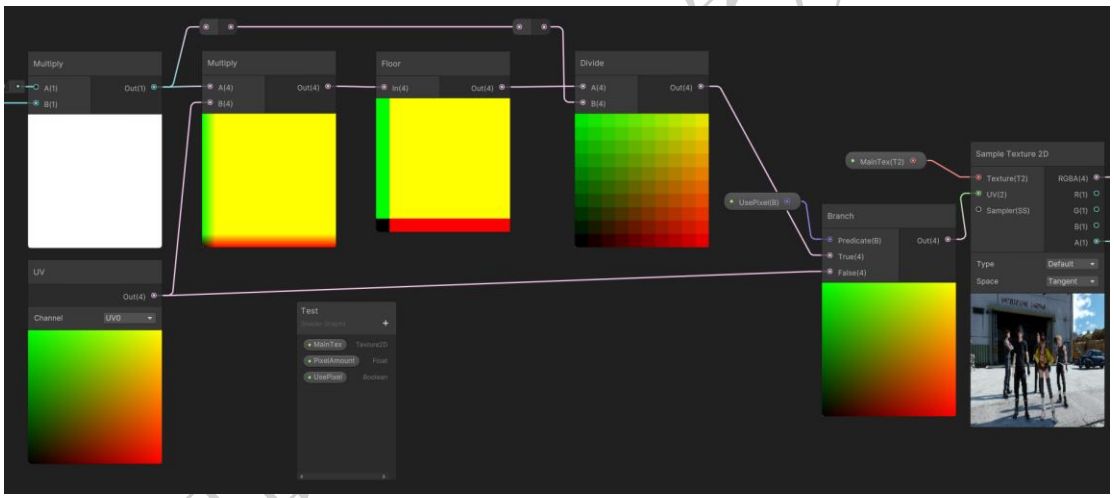


图177.　　使用 Branch 确定是否使用像素化效果

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。

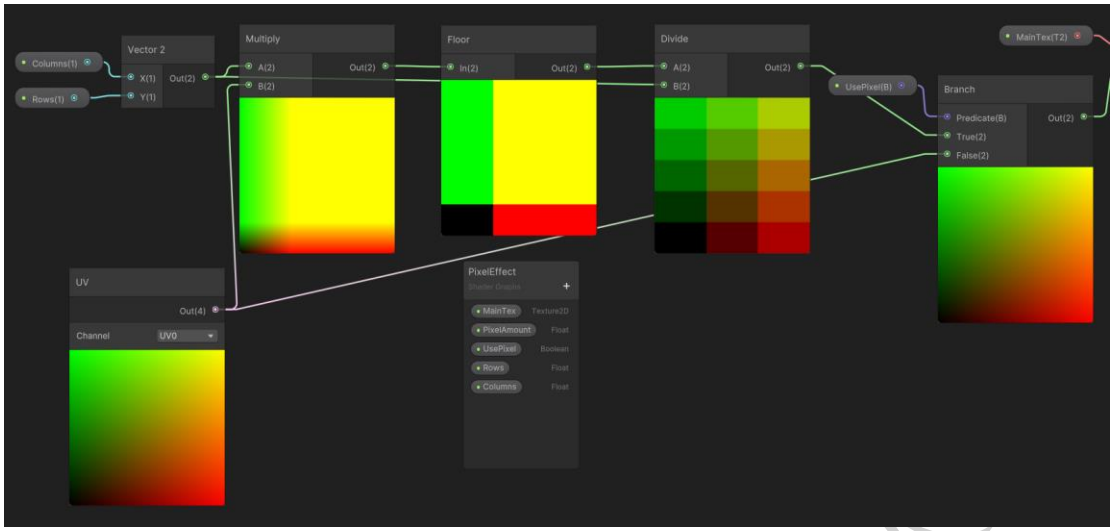


图178.　自定义像素化行数和列数

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。

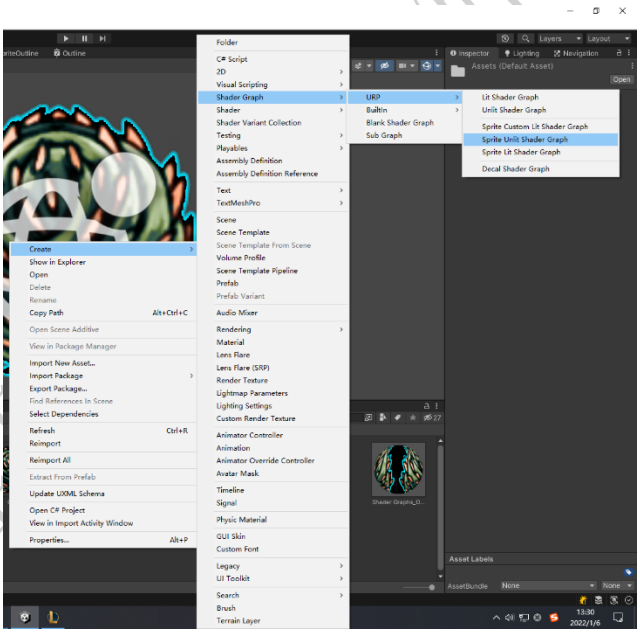


图5. 创建 Sprite Unlit Shader Graph

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179.　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。

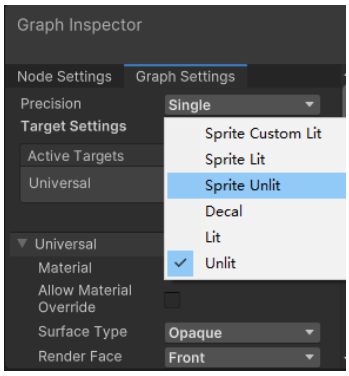


图180.　　修改 Material

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。

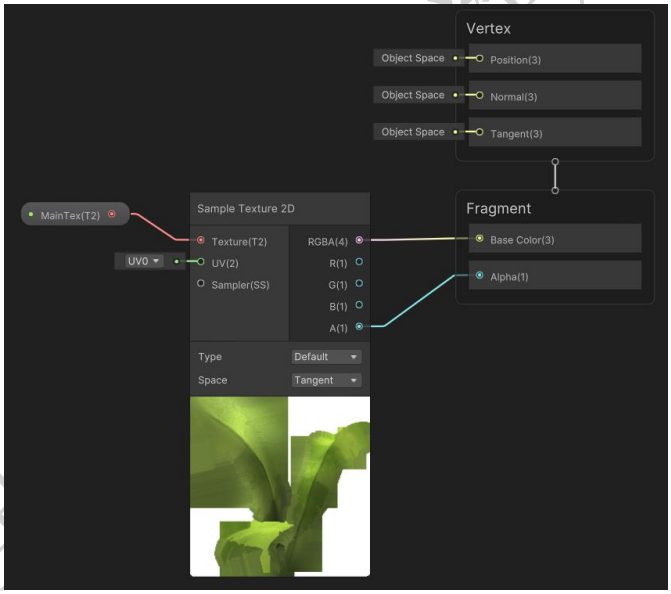


图181.　　主纹理输出

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。

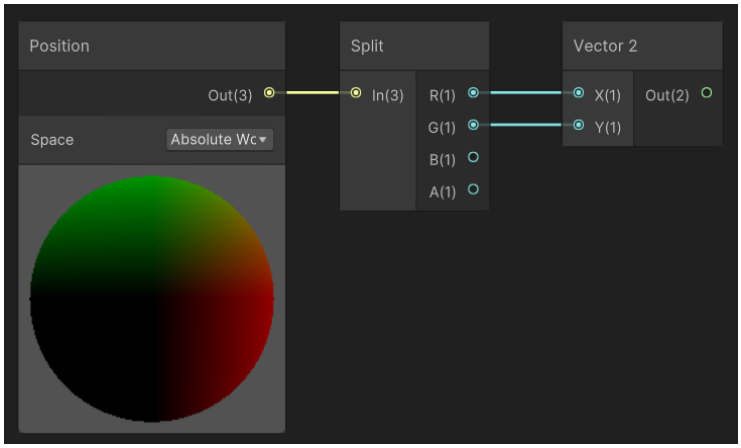


图182.　剥离 X 分量和 Y 纹理

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。

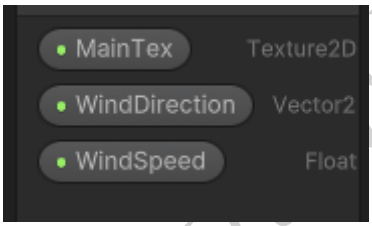


图183.　添加风速和风向参数

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。

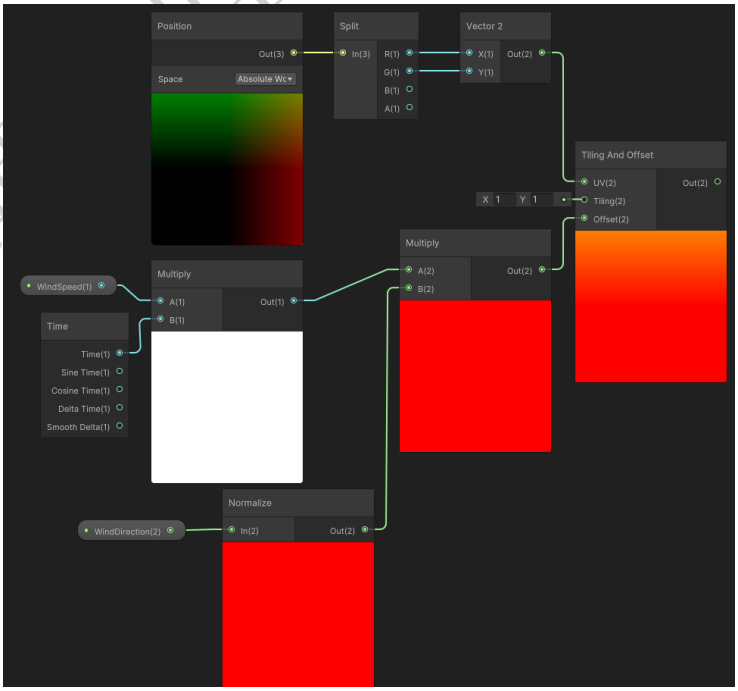


图184.　进行偏移

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。

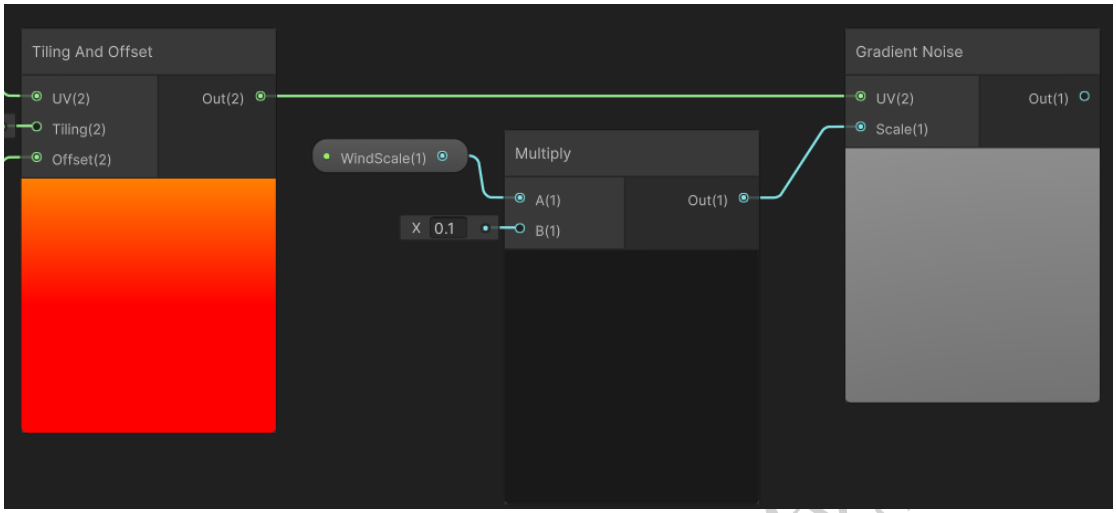


图185. 使用噪声调整偏移强度

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。

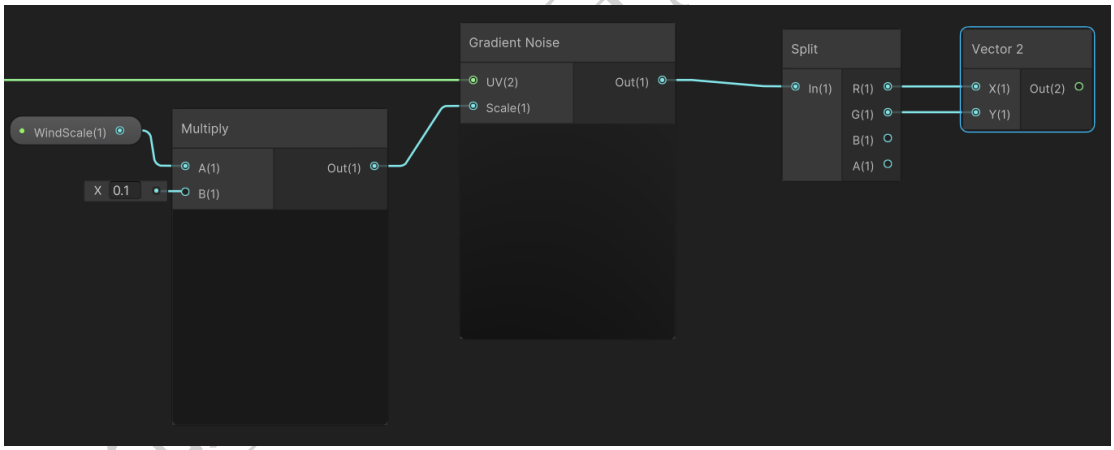


图186. 确保输出是二维向量

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。

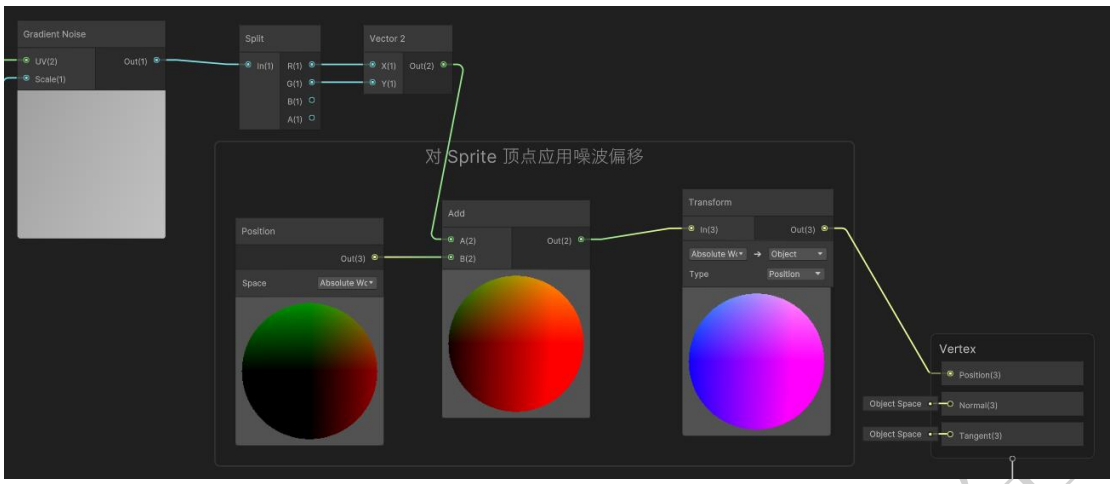


图187.　　对顶点应用噪声偏移

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。

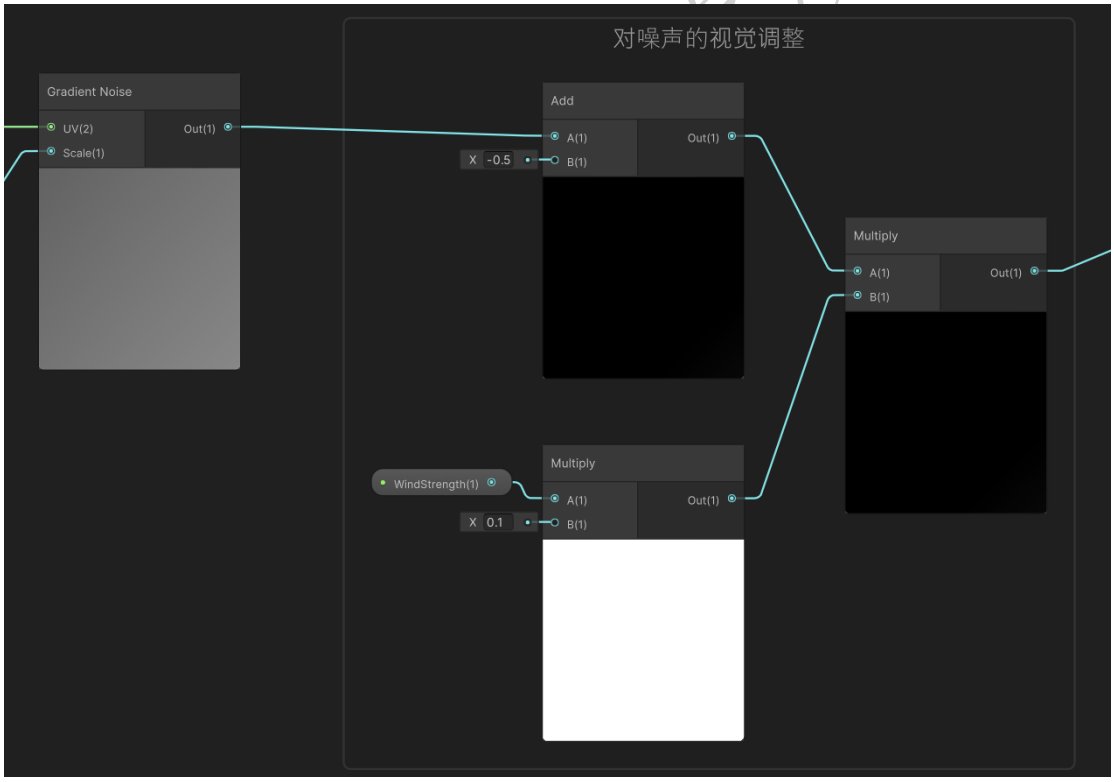


图188.　　调整噪声强度

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。

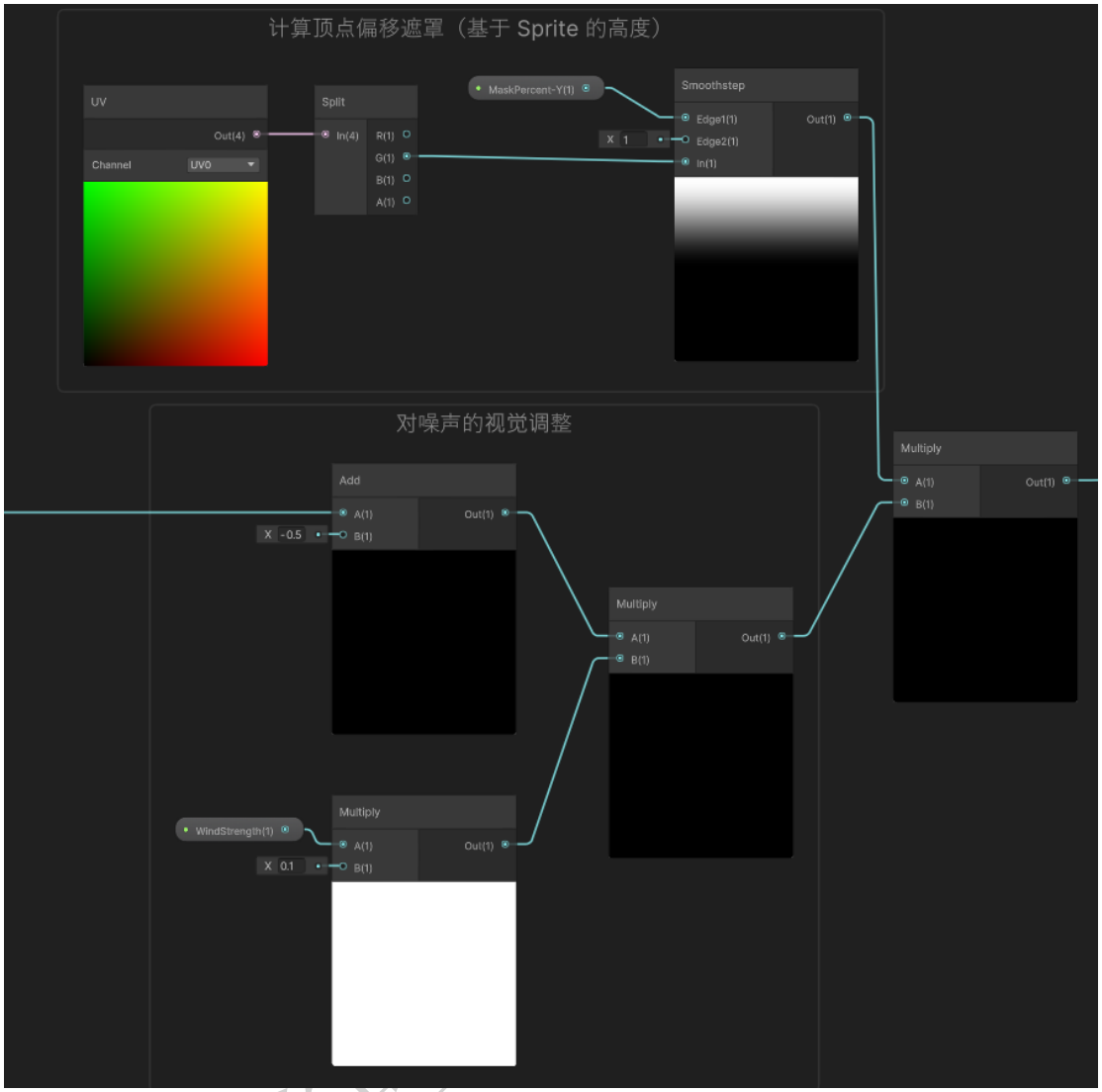


图189.　　顶点偏移蒙版

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。

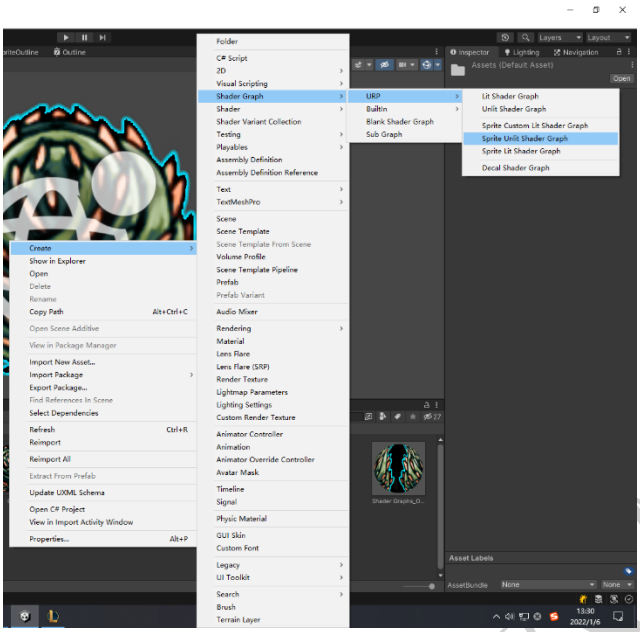


图190.　创建 Sprite Unlit Shader Graph

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191.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

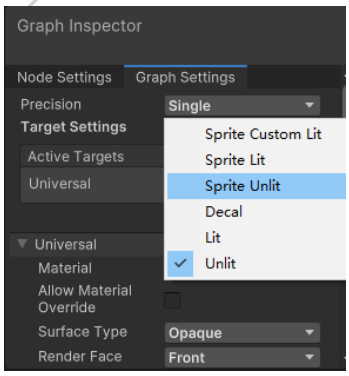


图192.　修改 Material

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。

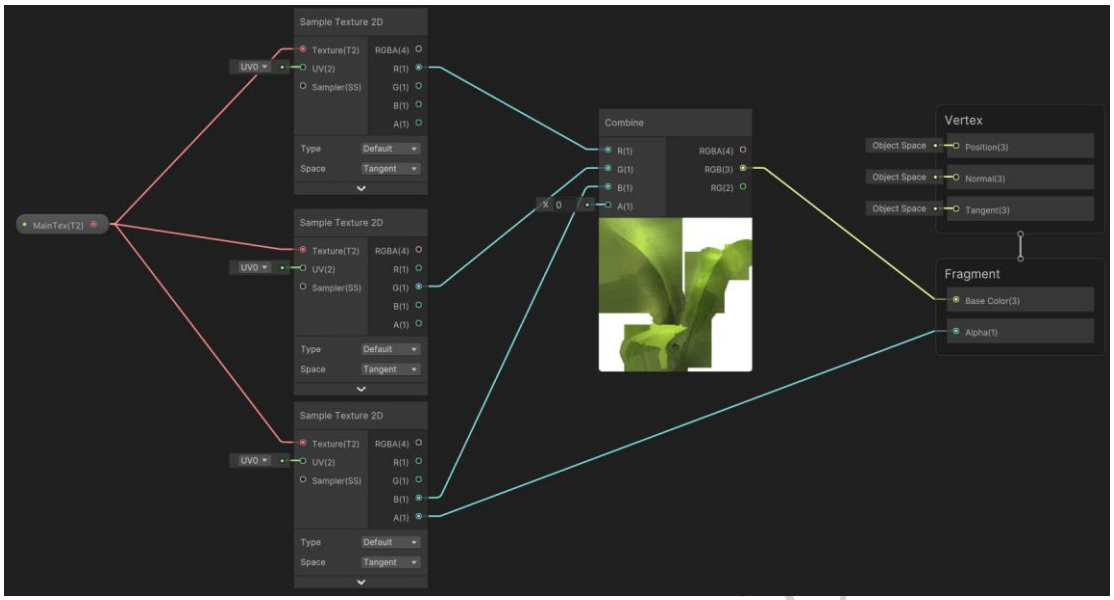


图193.　将主纹理进行输出

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。

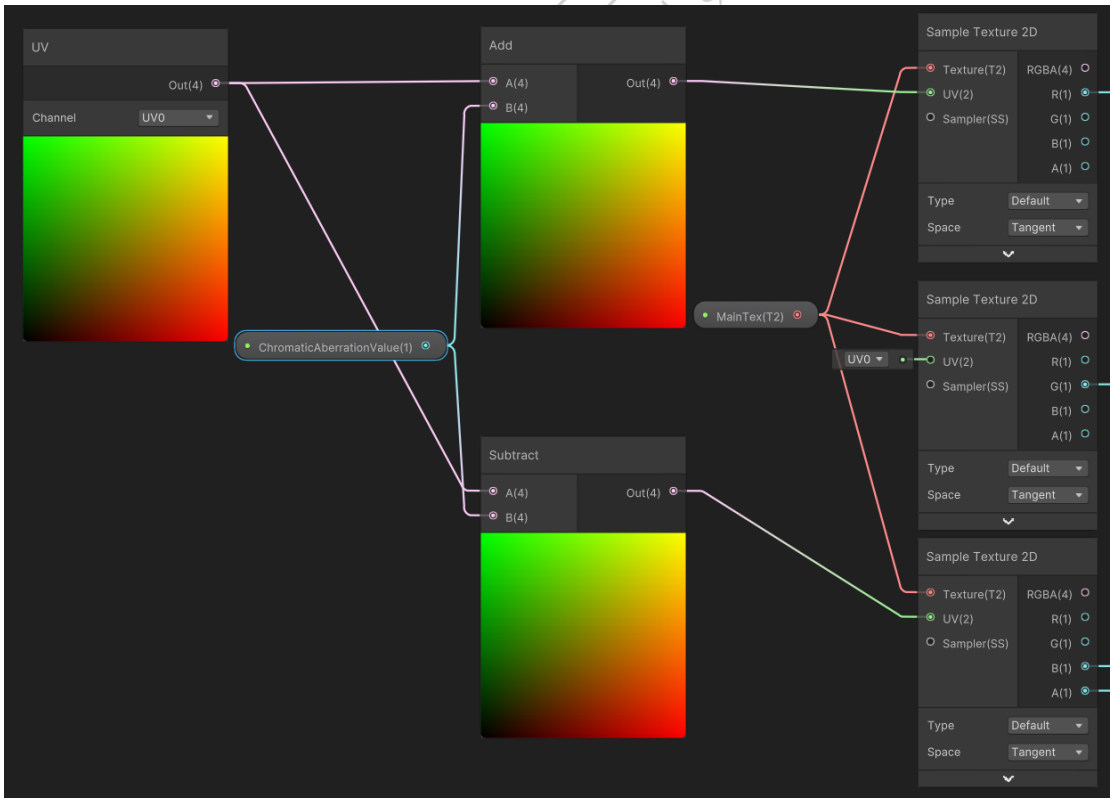


图194.　对 UV 进行加减，实现色差效果

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。

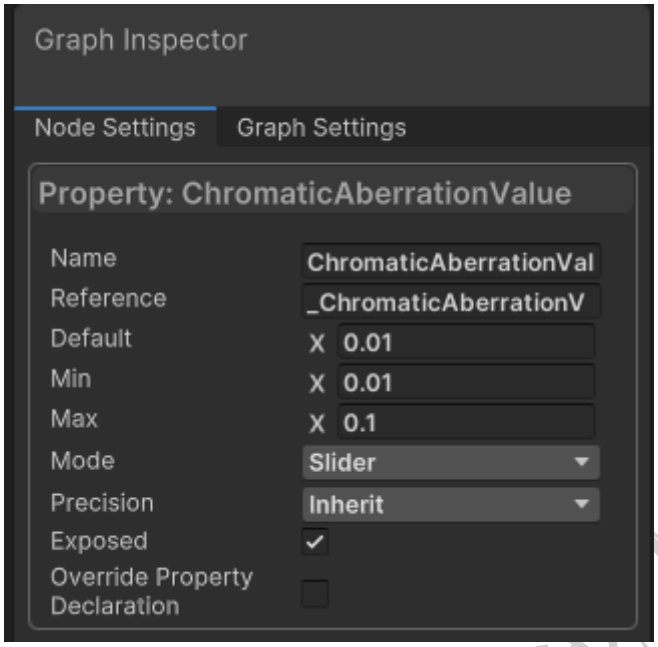


图195.　设置色差参数的模式为 Slider

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。

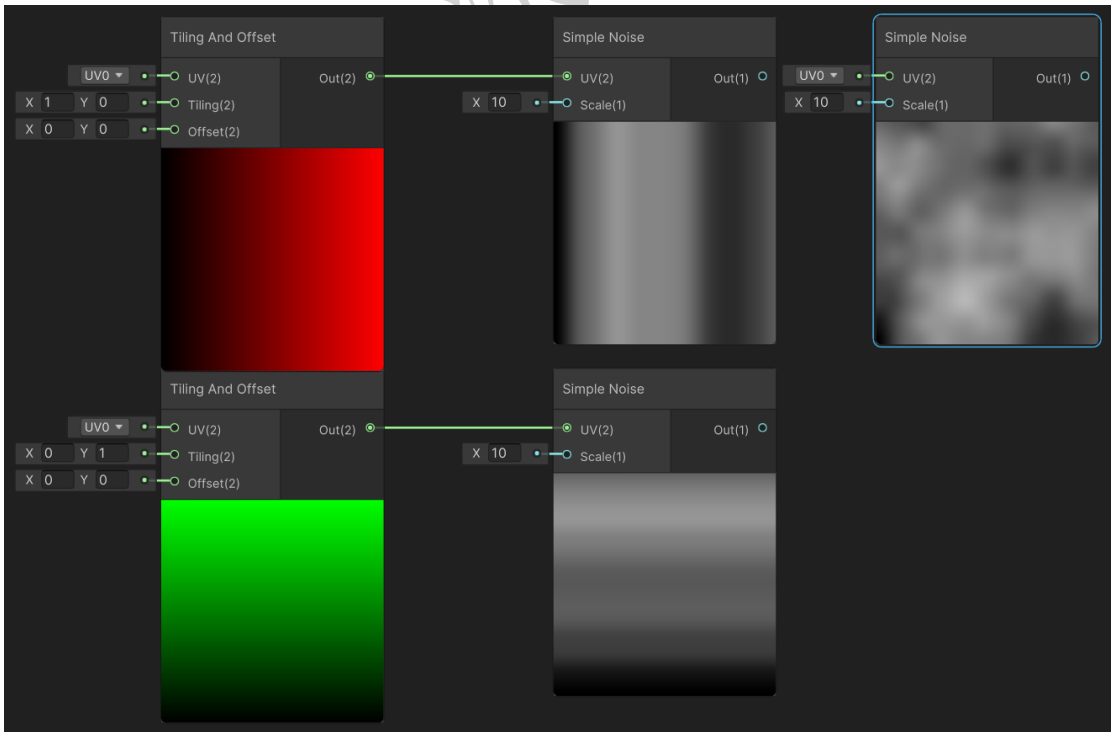


图196.　效果对照

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。

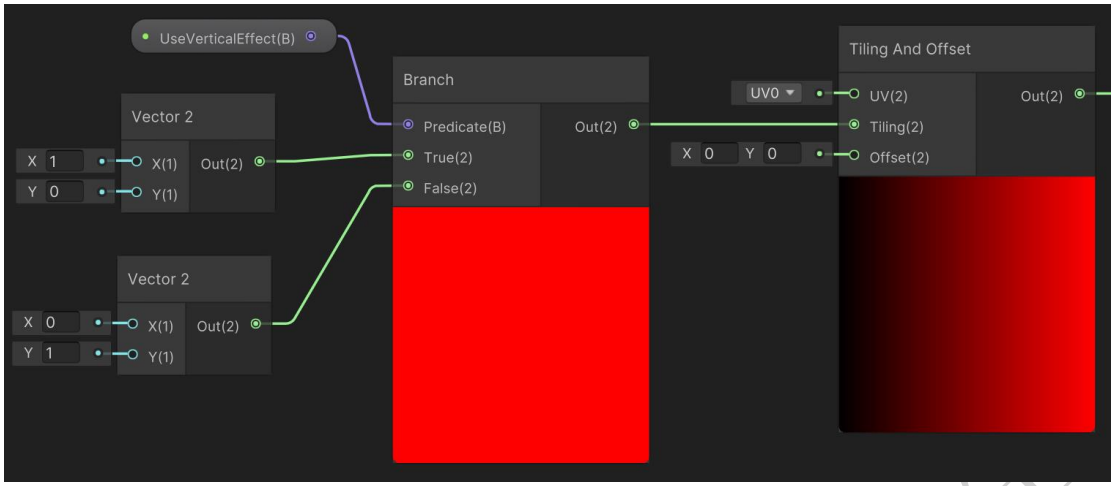


图197.　使用 Boolean 参数控制故障纹理

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。

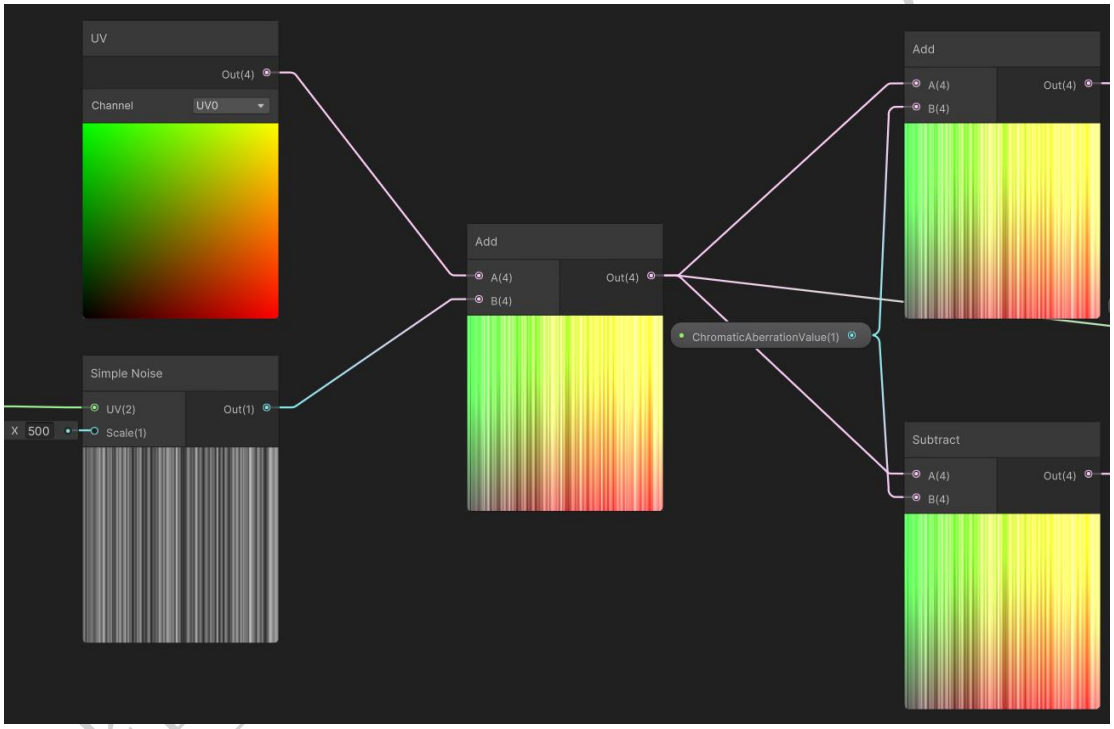


图198.　调整 UV，并将其用于色差

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。

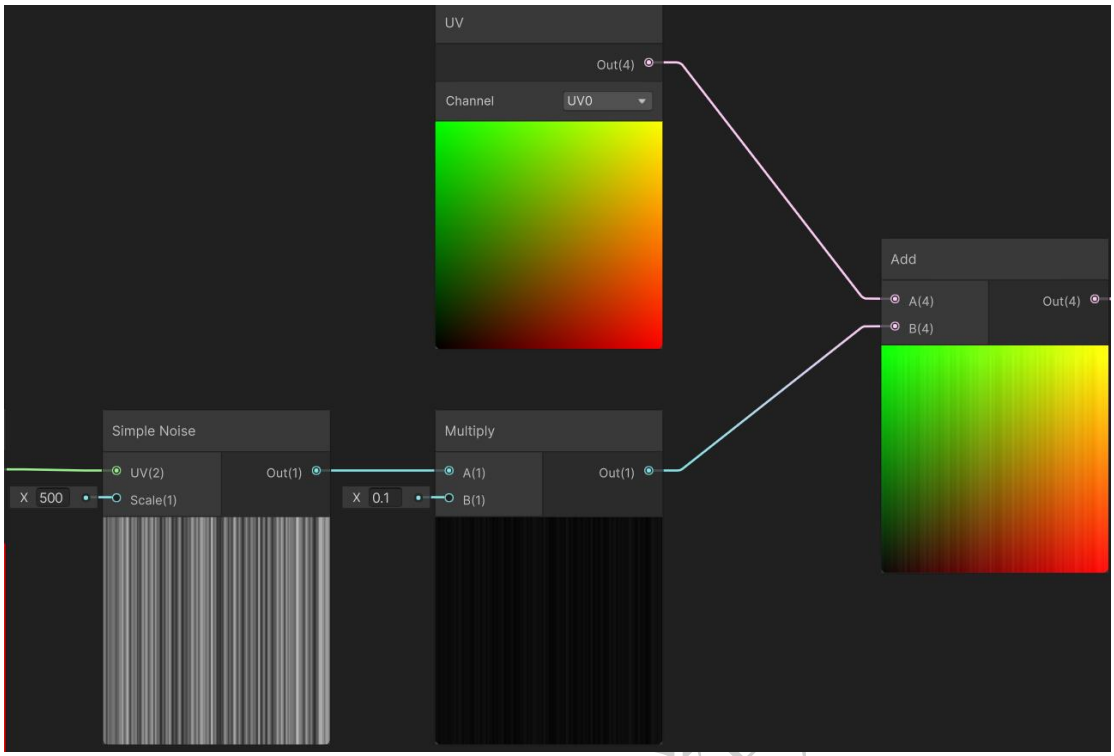


图199.　　减弱故障纹理的强度

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。

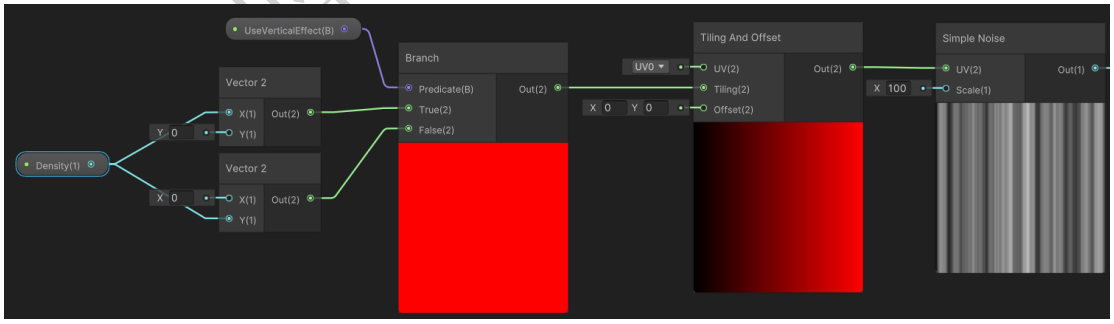


图200.　　调节故障纹理的密度

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。

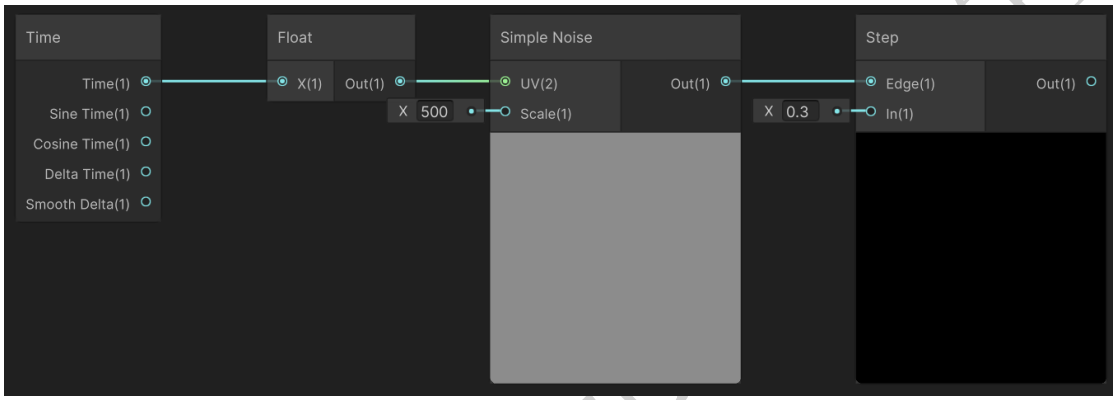


图201. 调整闪烁

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。

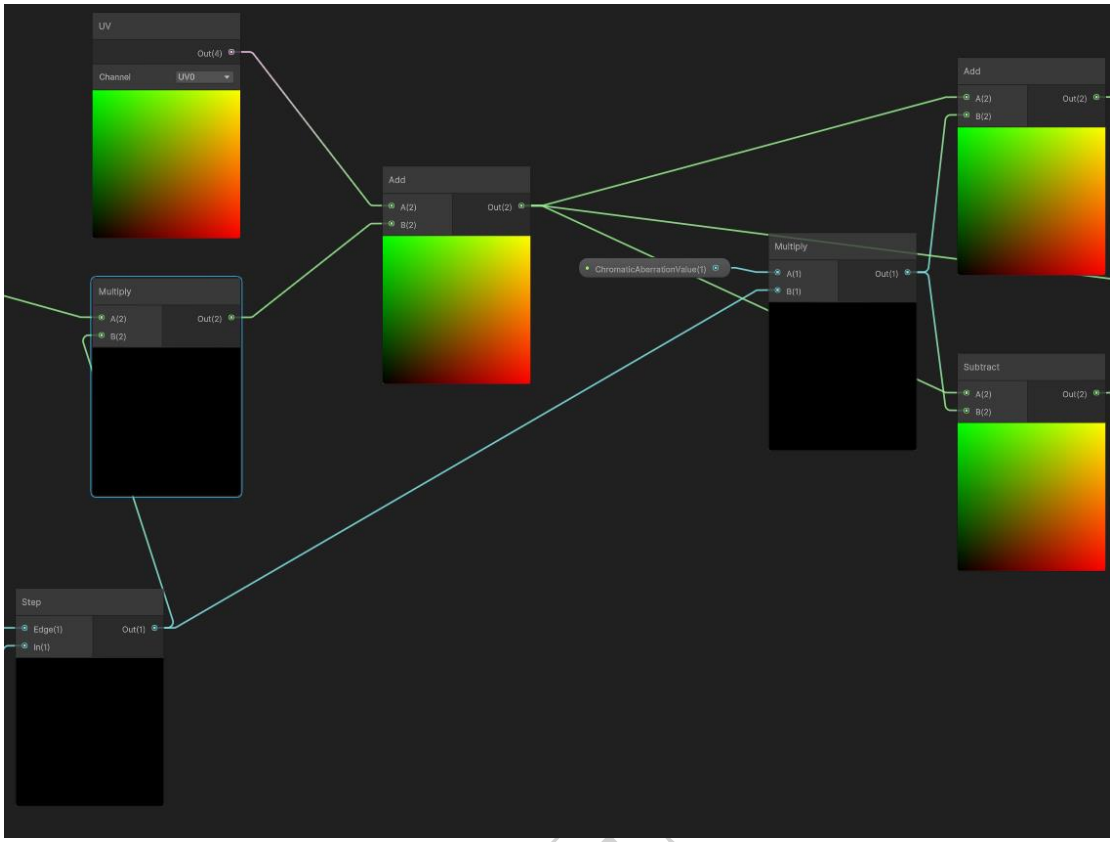


图202.　闪烁效果控制色差和故障的发生

十六、　赛博朋克故障效果

十七、　UV 中心缩放

1. 思路讲解

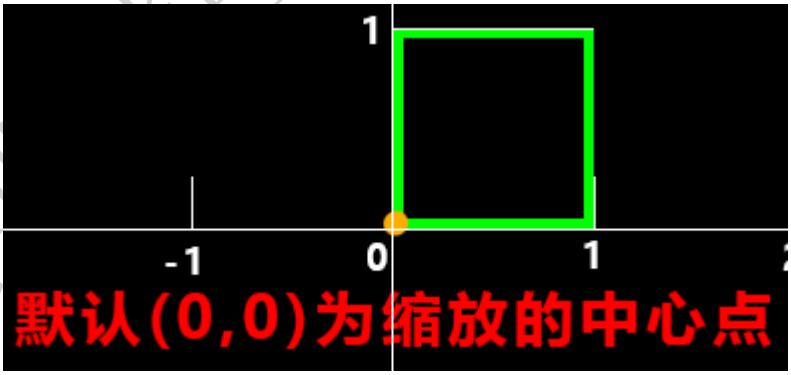


图203.　默认 UV 缩放点

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。

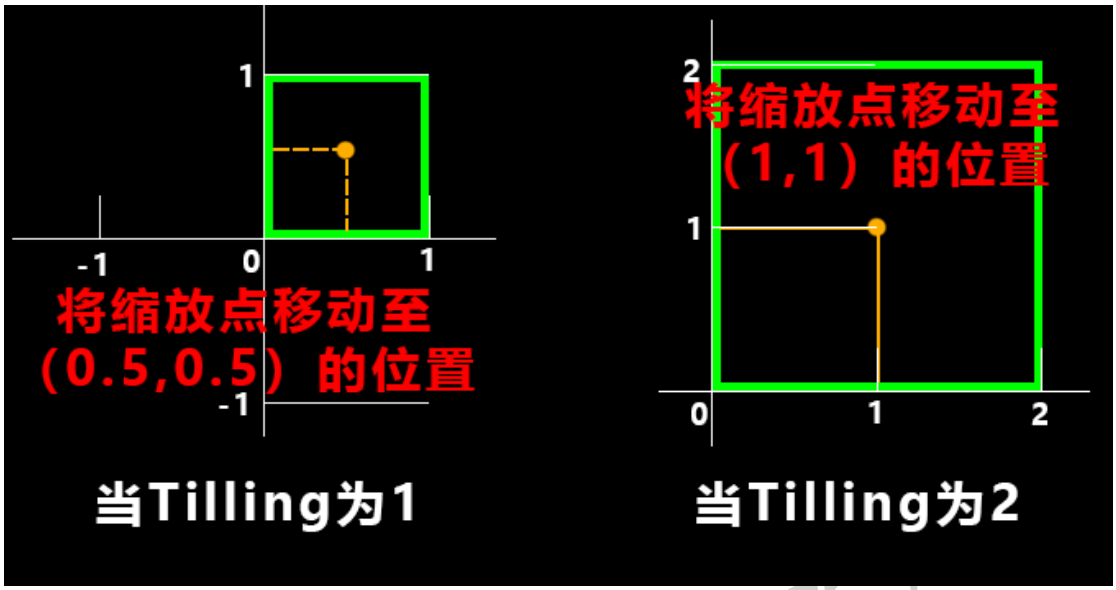


图204.　当 Tilling 为 1 和 2 时的缩放点坐标

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。

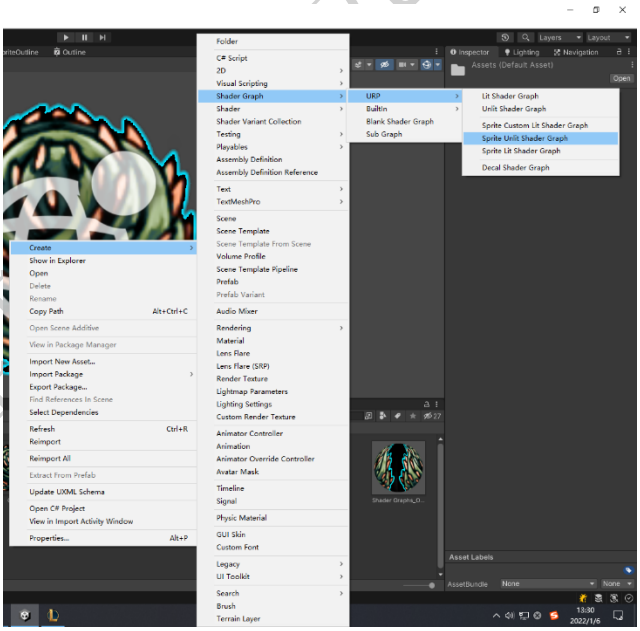


图205.　创建 Sprite Unlit Shader Graph

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206.　Graph Inspector 面板按钮

找到 Universal 里的 Material，将其修改为 Sprite Unlit。

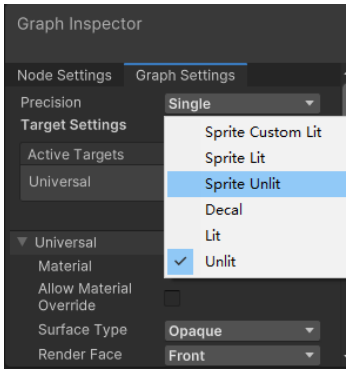


图207.　修改 Material

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。

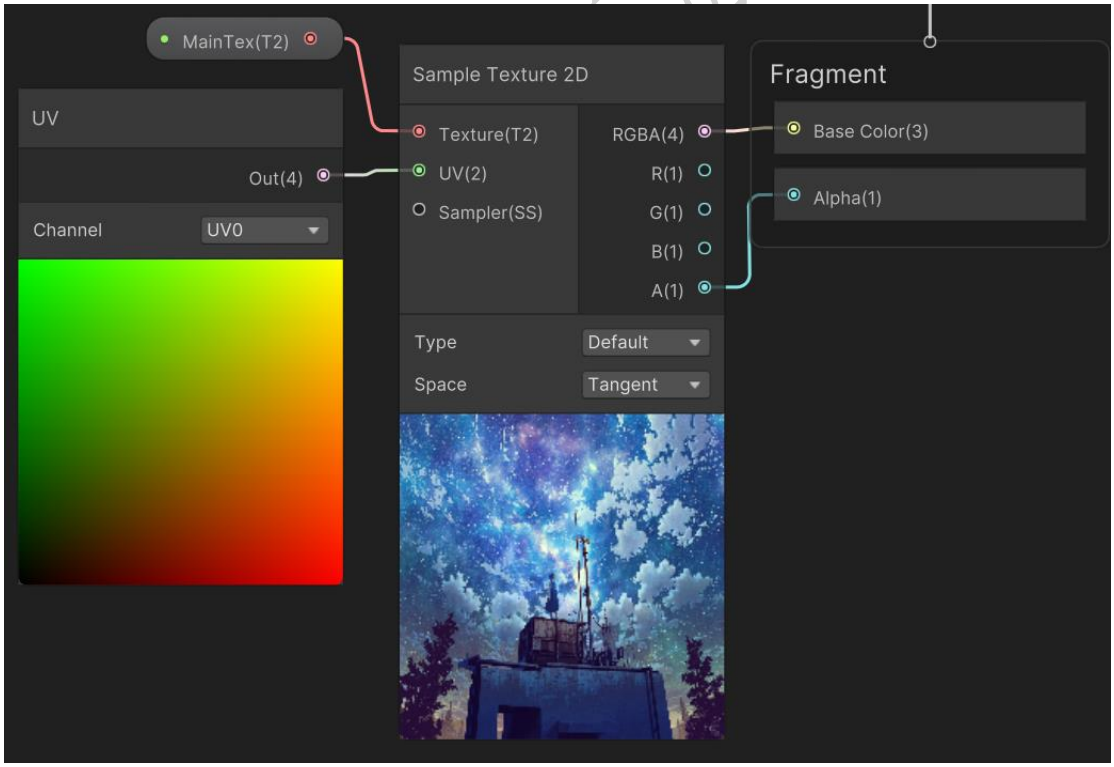


图208.　输入标准 UV 所得的图像

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。

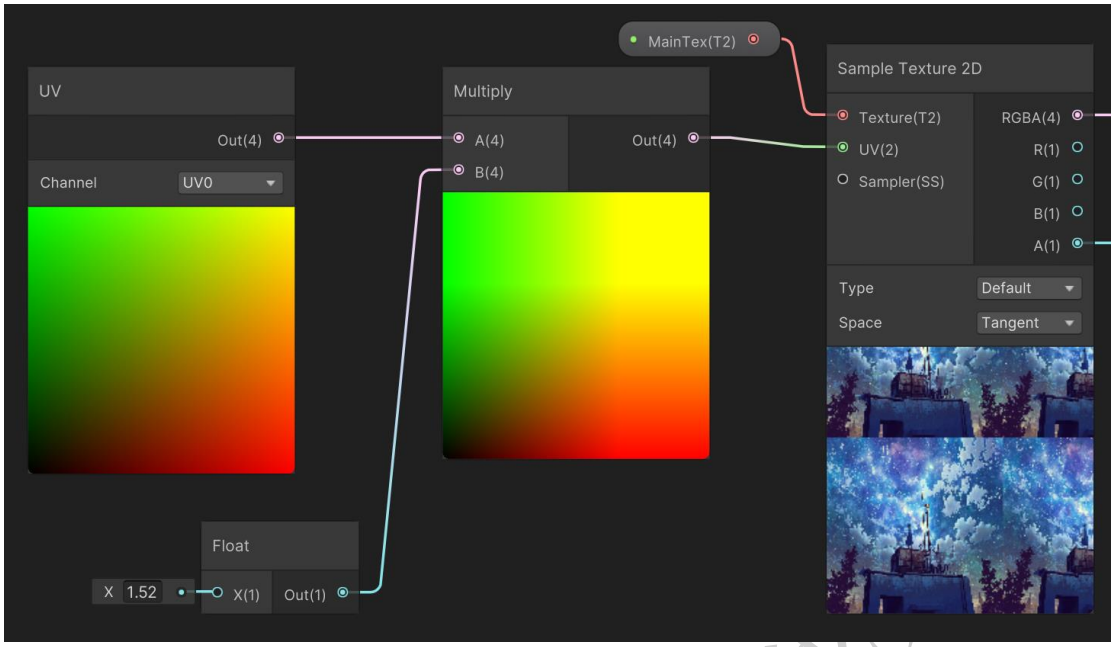


图209.　　查看 UV 的默认缩放中心点

首先将 Tiling 暴露到参数面板，方便在外部对其进行修改。

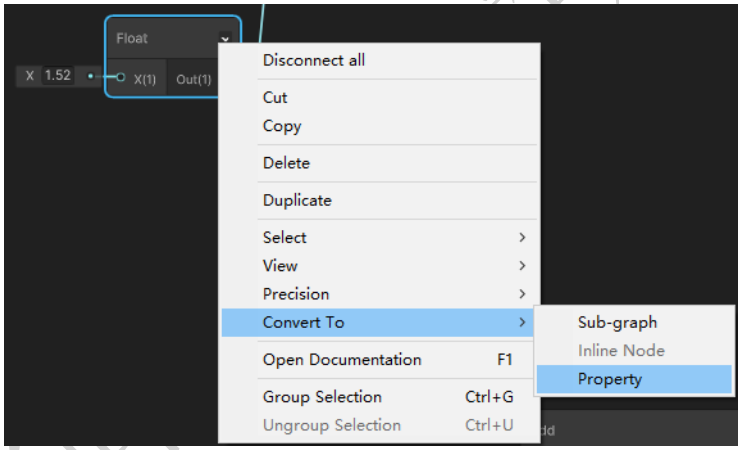


图210.　　将节点转换为参数暴露到参数面板

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。

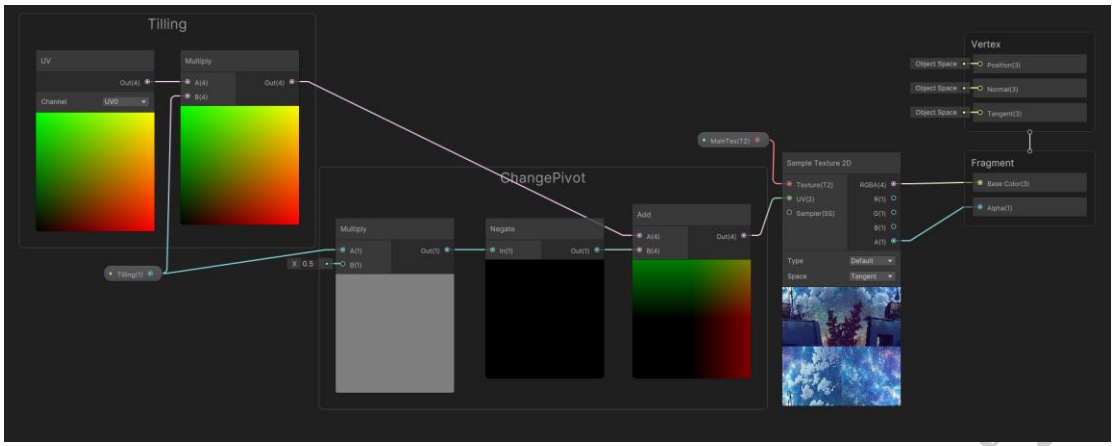


图211.　　虚假的中心缩放最终连接图

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

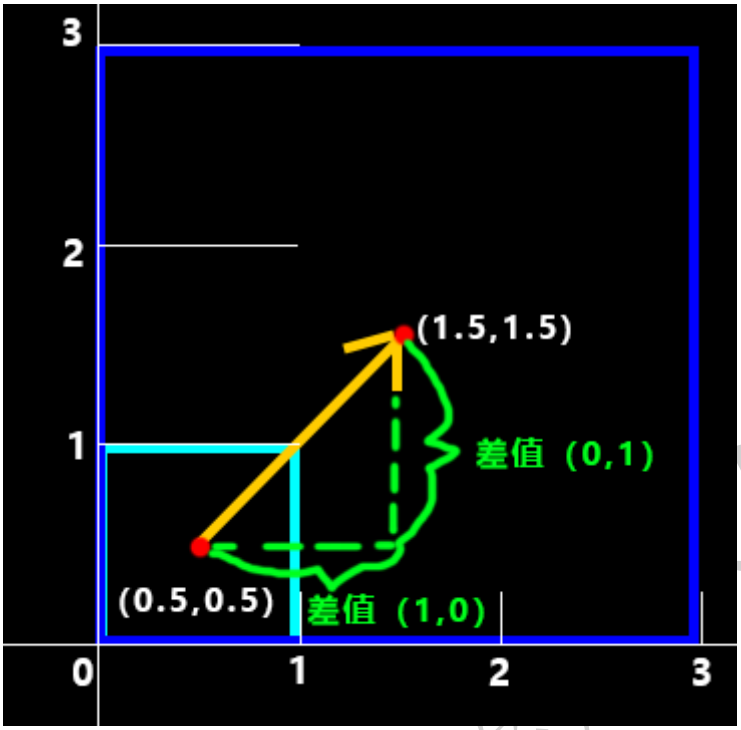


图212.　原始中心坐标与新中心坐标的差值

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。

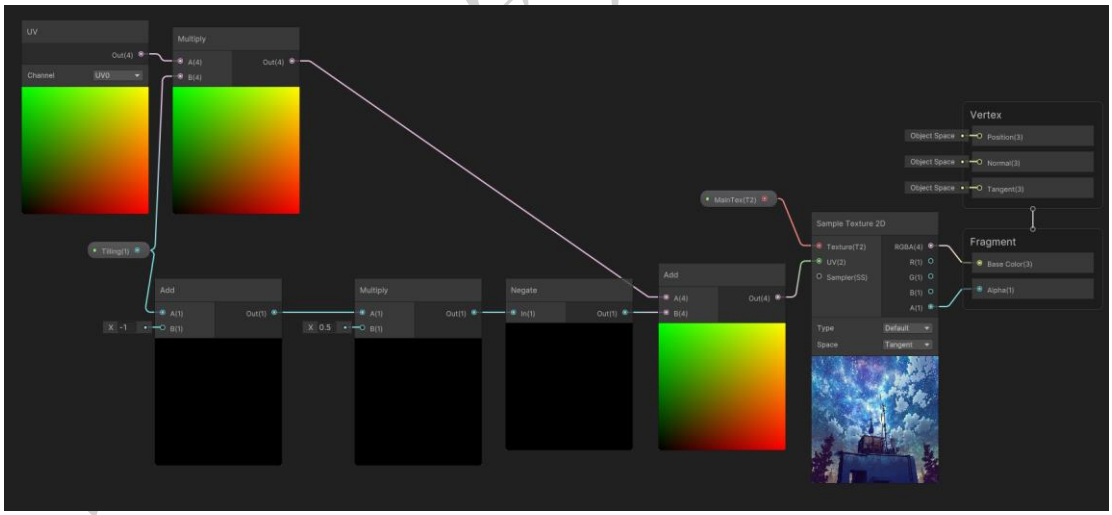


图213.　真实的中心缩放

十八、　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。

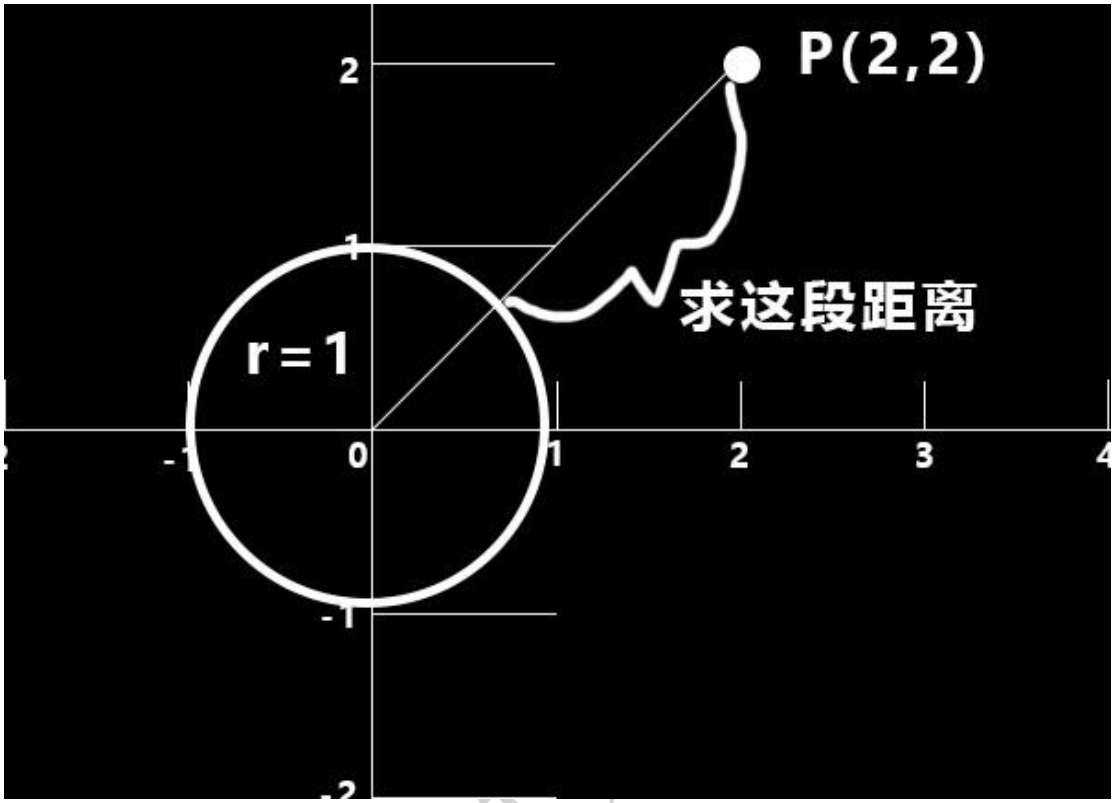


图214. 圆心在坐标轴原点

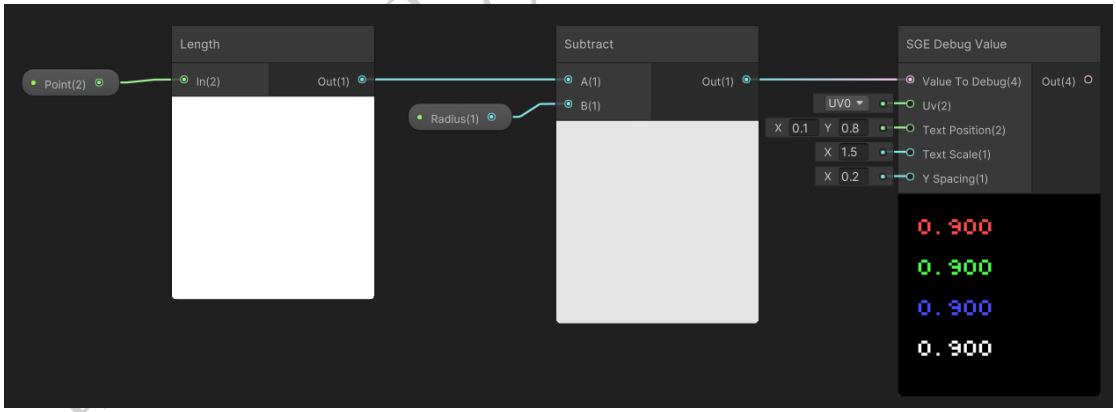


图215. 圆心在原点时的 ShaderGraph

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。）

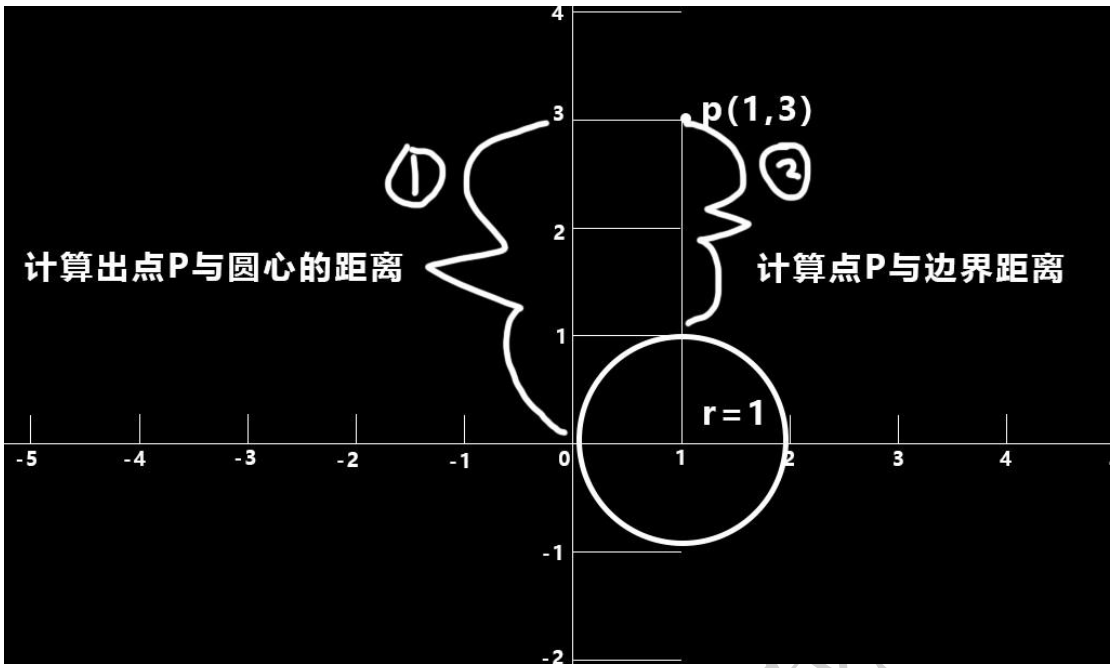


图216.　　圆心不在坐标轴原点

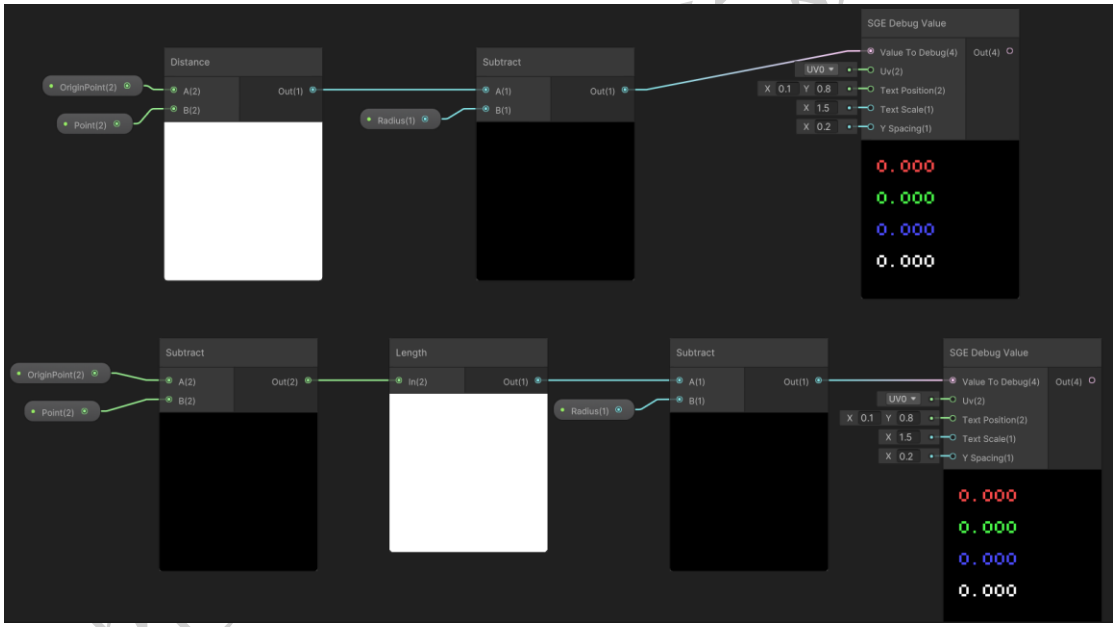


图217.　　圆心不在原点时的两种连接方法

十九、　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。

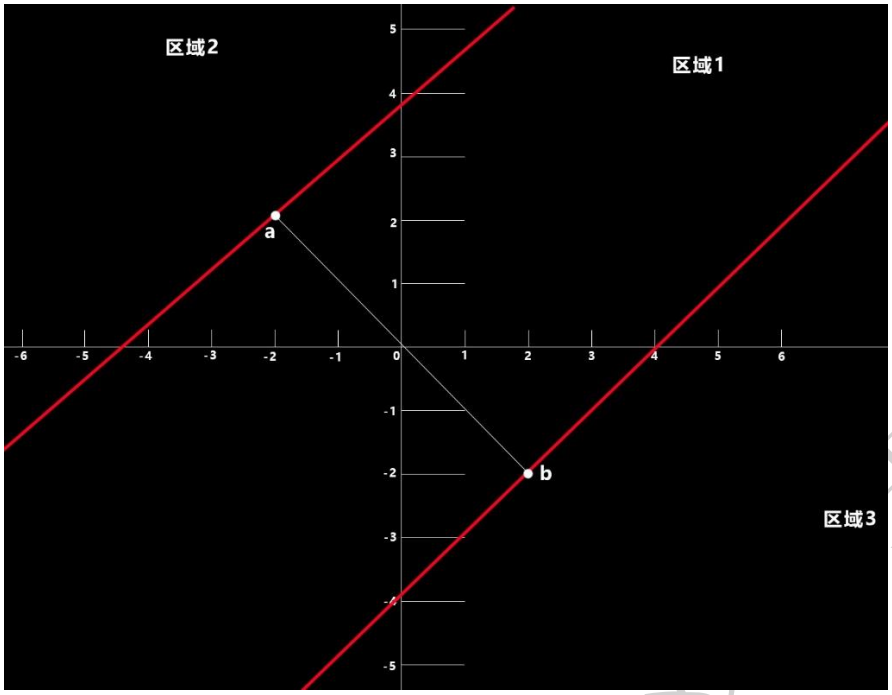


图218.　　SDF 线段区域划分

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

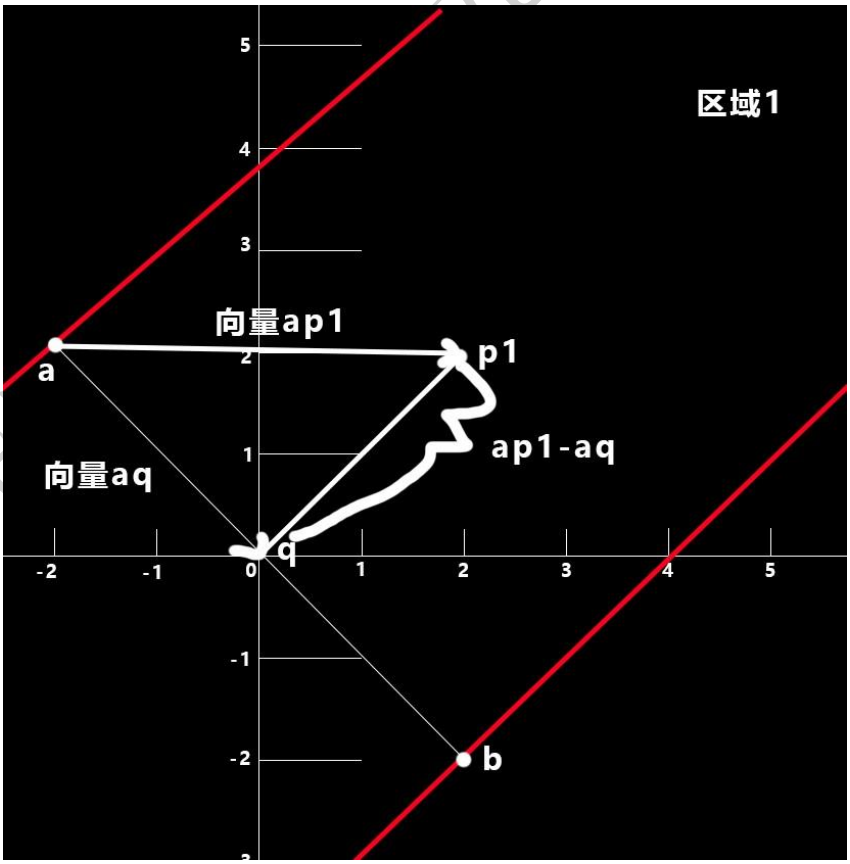


图219.　　点位于区域一内

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，$\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[17]$\overrightarrow{ab}$。（r[18]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

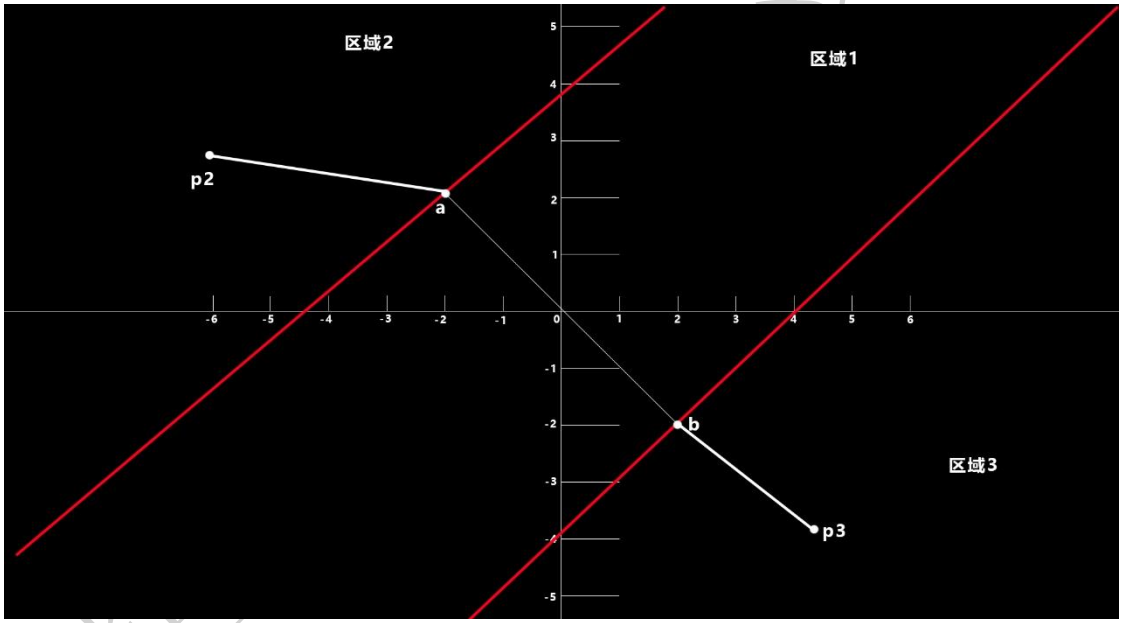


图220. 点位于区域二和三

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=$\|\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[19]$\overrightarrow{ab}\|$

区域二：d=$\|\overrightarrow{ap_2} = \overrightarrow{ap_2} - 0 * \overrightarrow{ab}\|$

区域三：d=$\|\overrightarrow{bp_3} = \overrightarrow{ap_3} - 1 * \overrightarrow{ab}\|$

[17] 这里的 r 是单词 Ratio 的缩写
[18] 这里的 r 是单词 Ratio 的缩写
[19] 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）

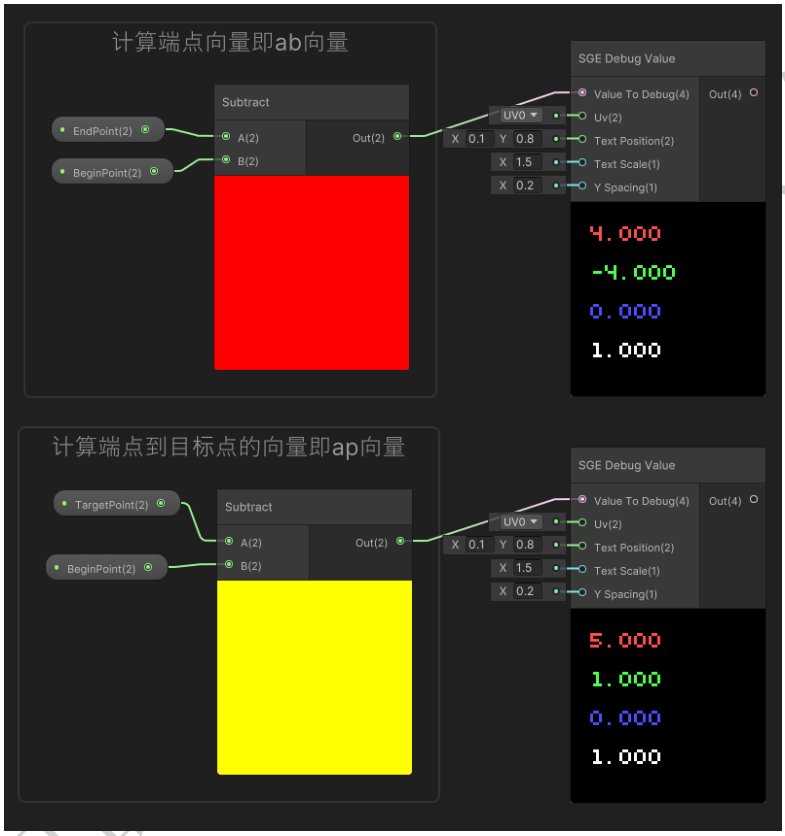


图221.　　计算得到两个向量

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。

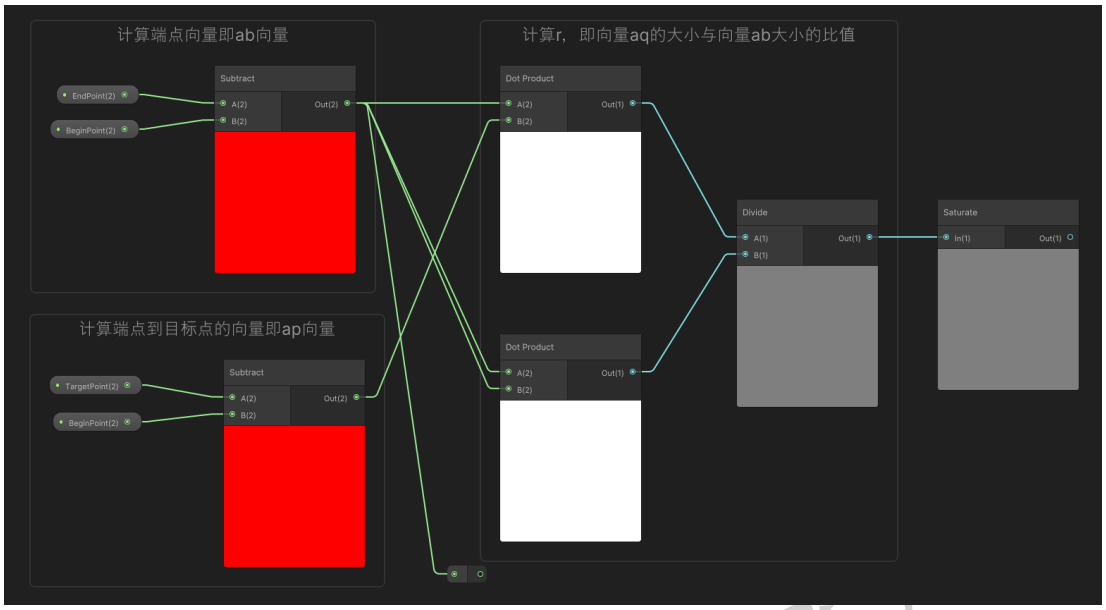


图222.　　计算 r

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。

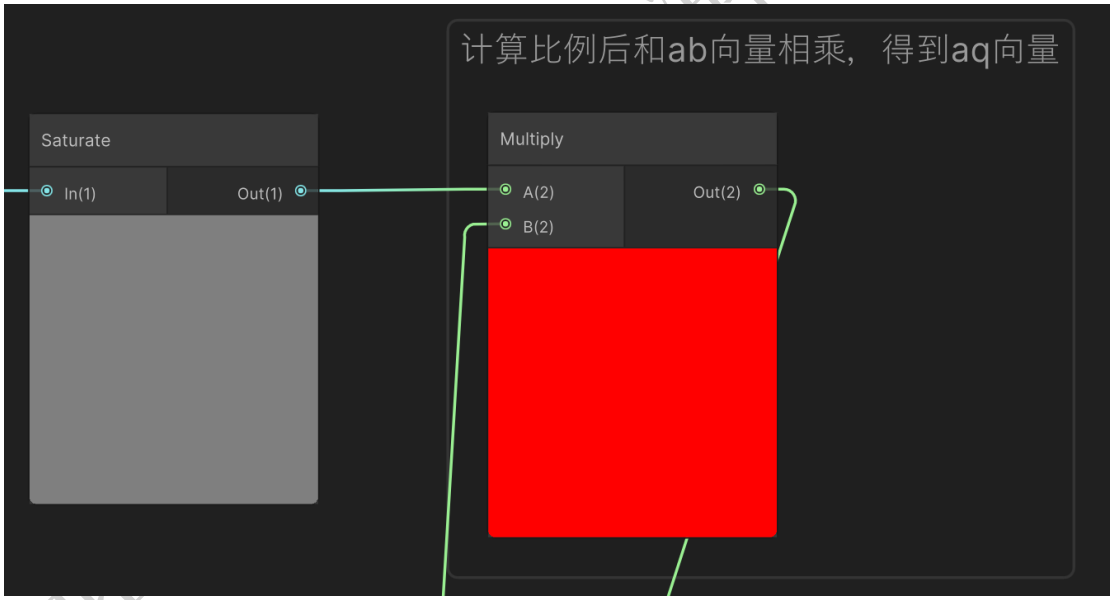


图223.　　得到 aq 向量

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）。

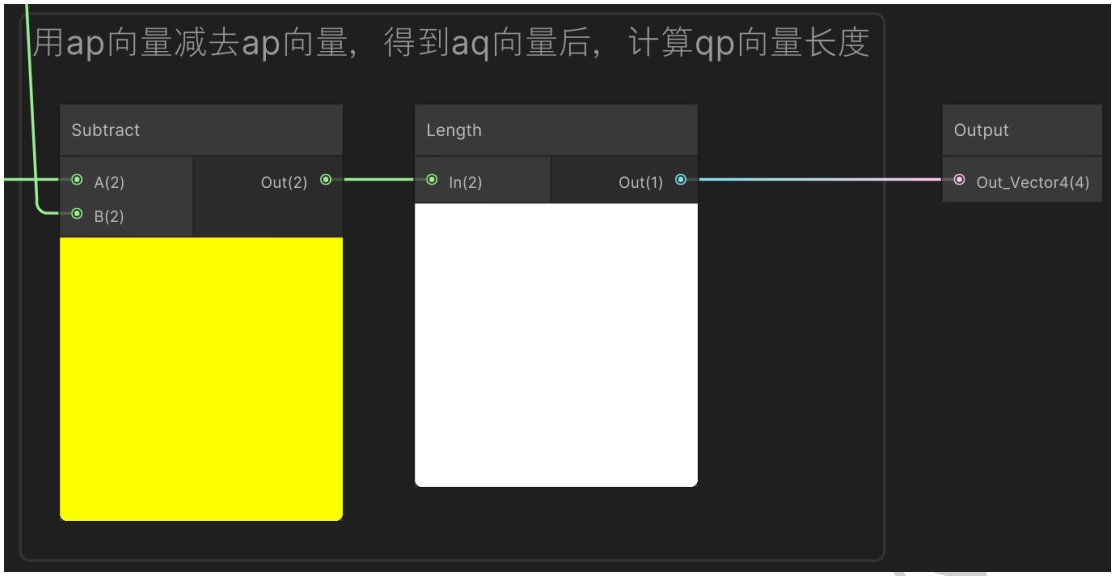


图224.　　计算 qp 向量长度

二十、　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。

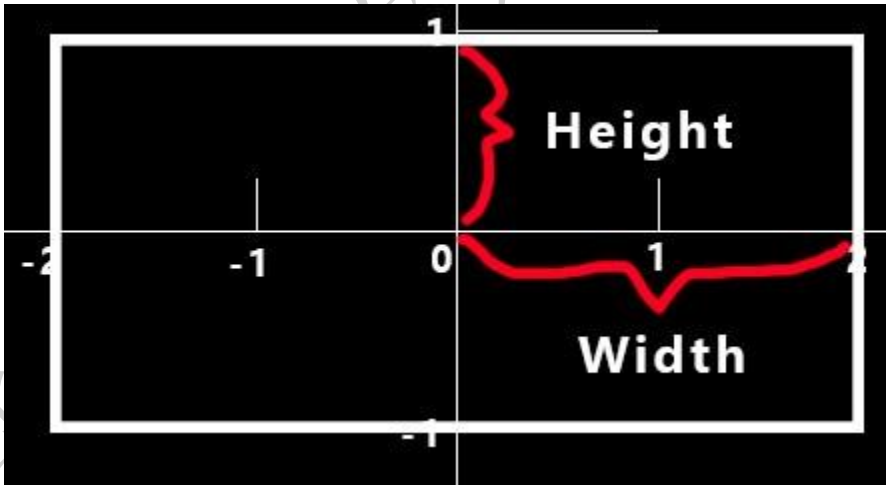


图225.　　矩形绘制原理

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。

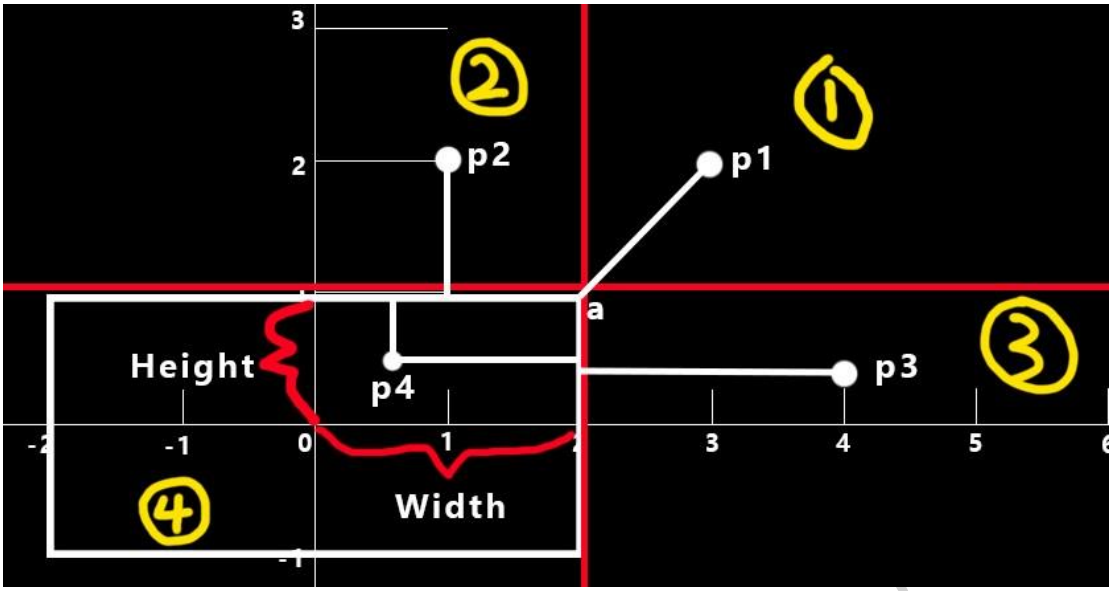


图226.　　目标点到矩形的情况划分

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。

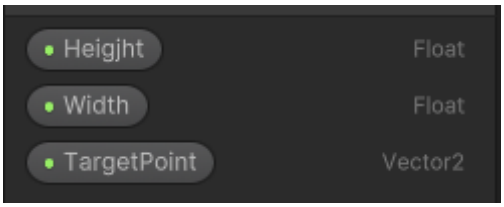


图227.　SDF 矩形参数设置

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。

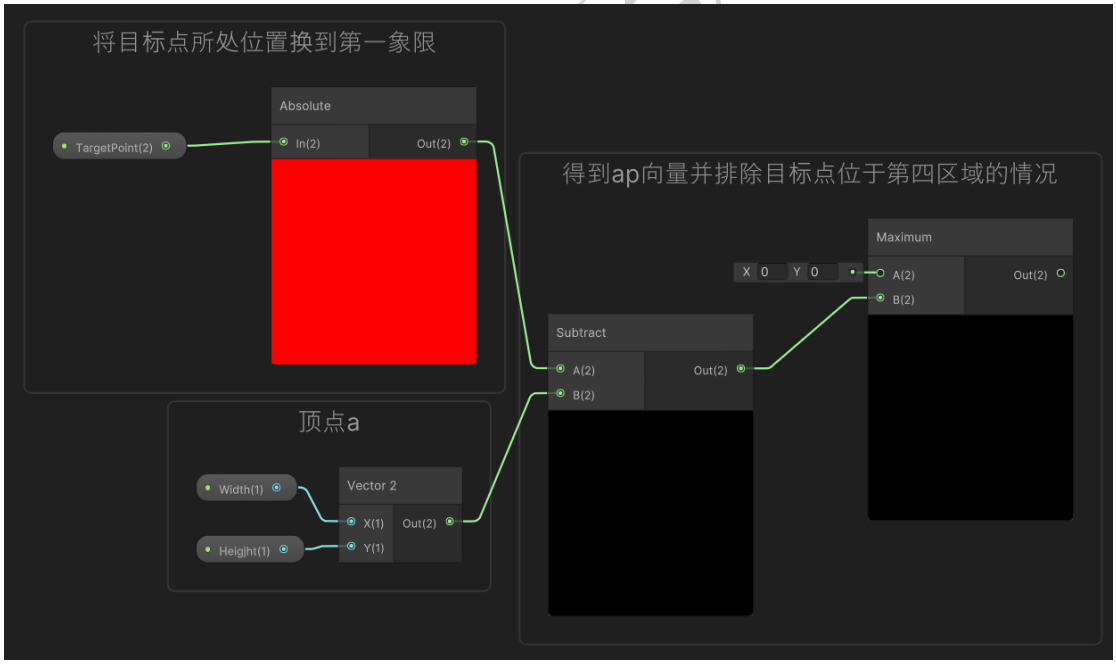


图228.　计算向量 ap1

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。

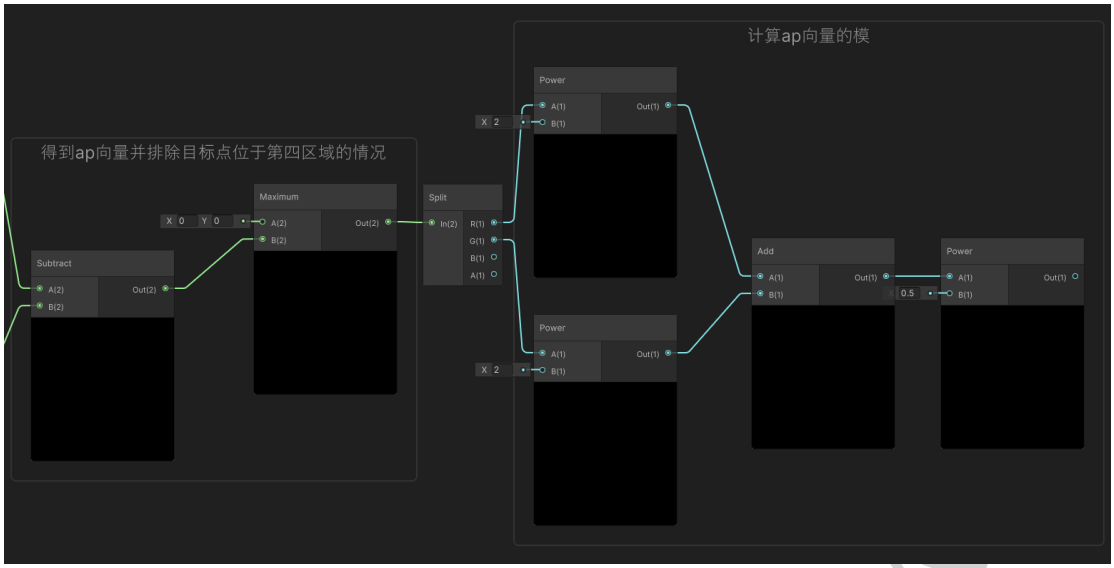


图229.　　计算目标点位于区域一、二、三时的模

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）

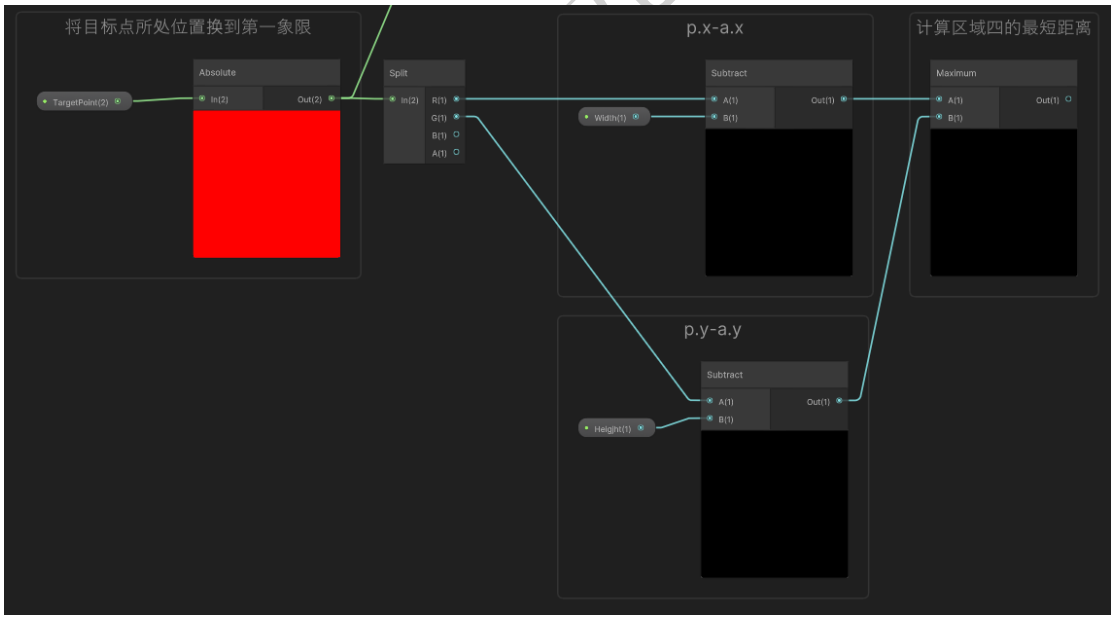


图230.　　计算区域四中点的最短距离

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。

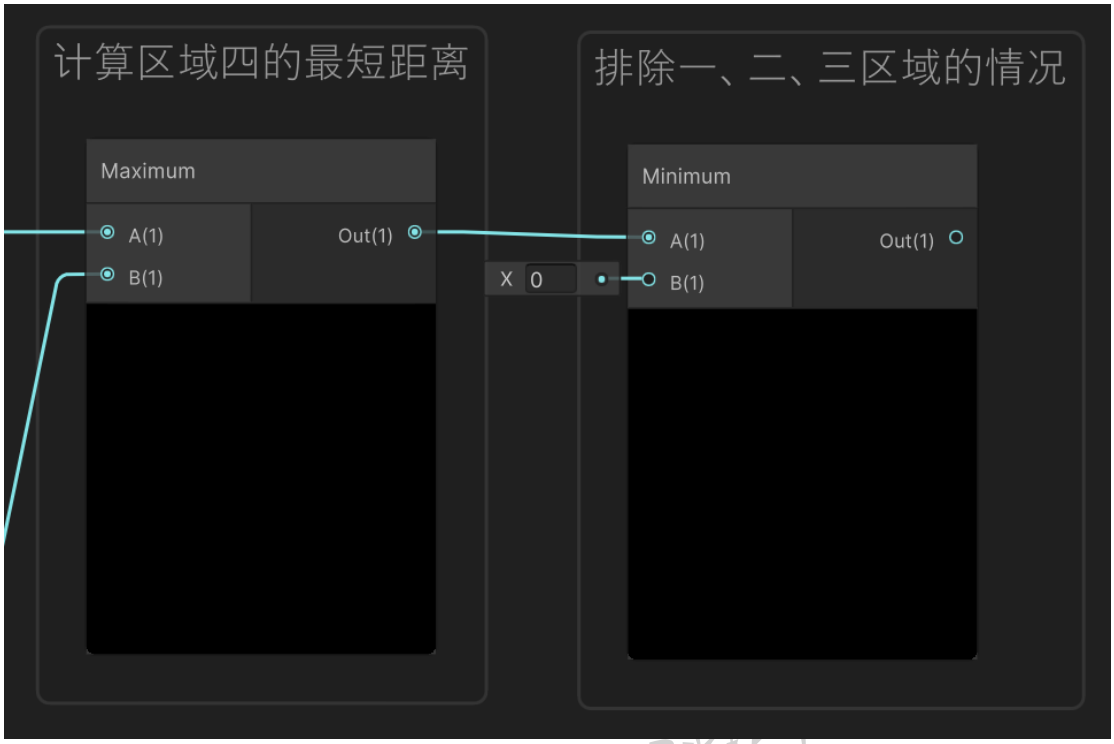


图231.　　排除目标点不位于区域四的情况

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。

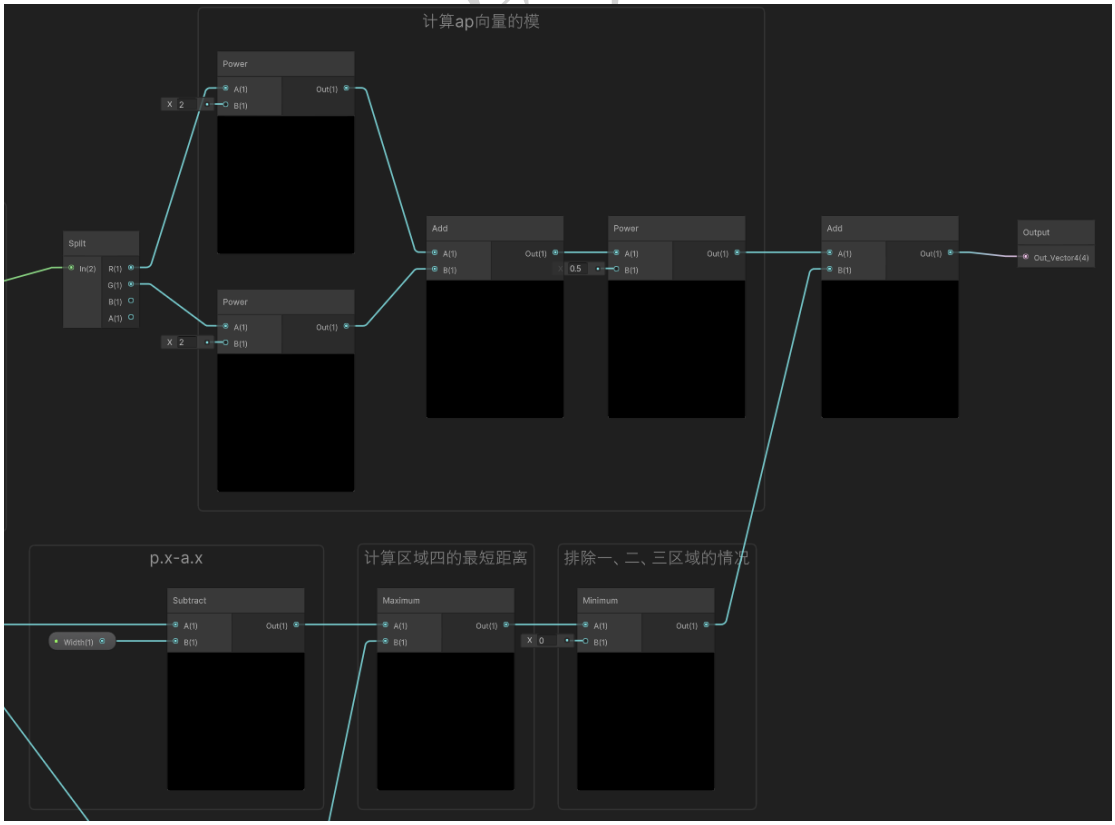


图232.　　SDF 矩形最终结果

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。

二十一、 3D 菲尼尔边缘光

1. 什么是菲尼尔效应

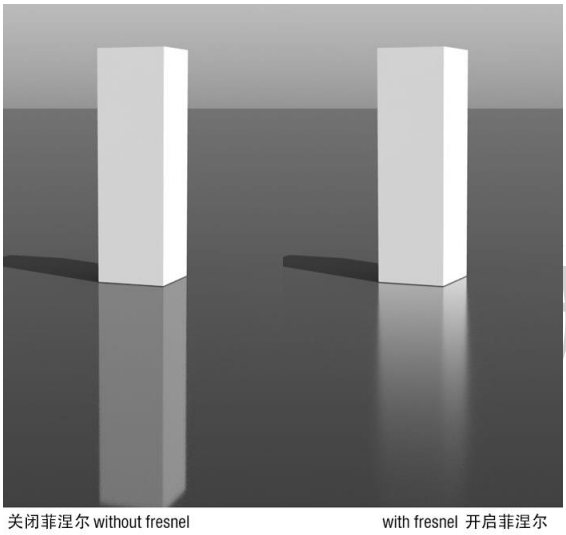


图233. 菲尼尔效果的示例

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234. 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

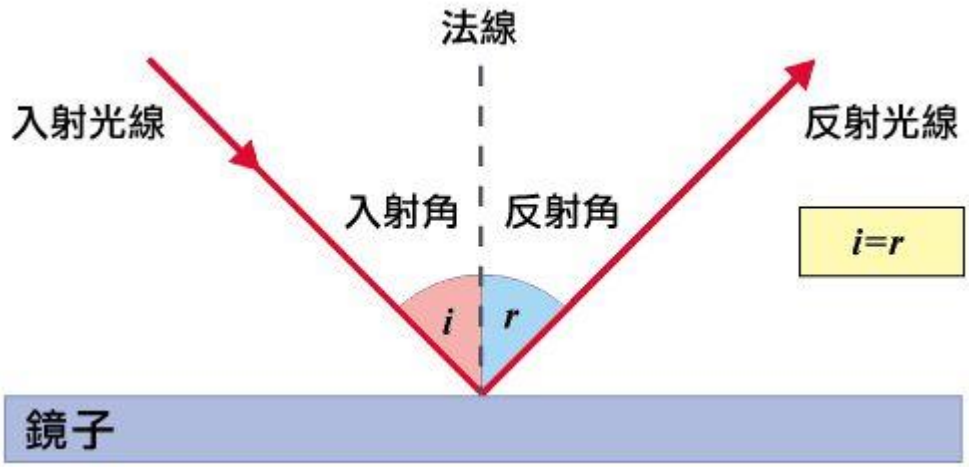


图235. 入射角示例

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。

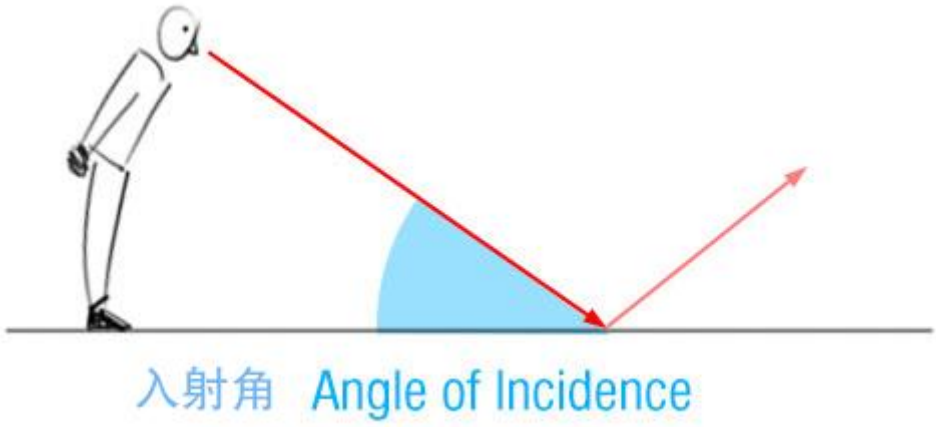


图236. 水平面的入射角示例

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。

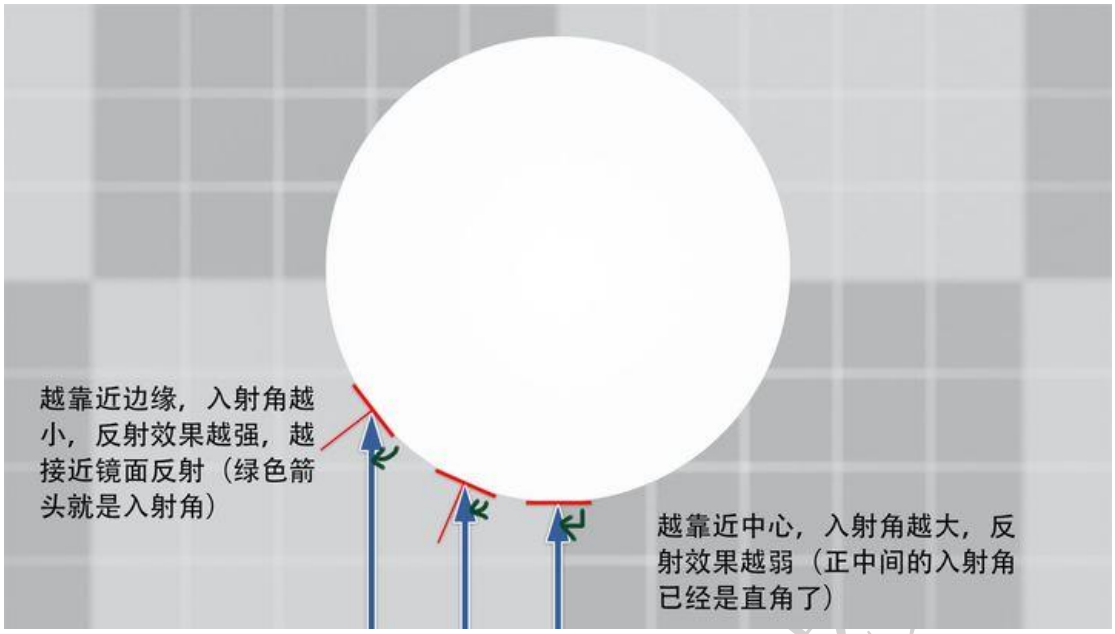


图237.　　球体的入射角示例

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。

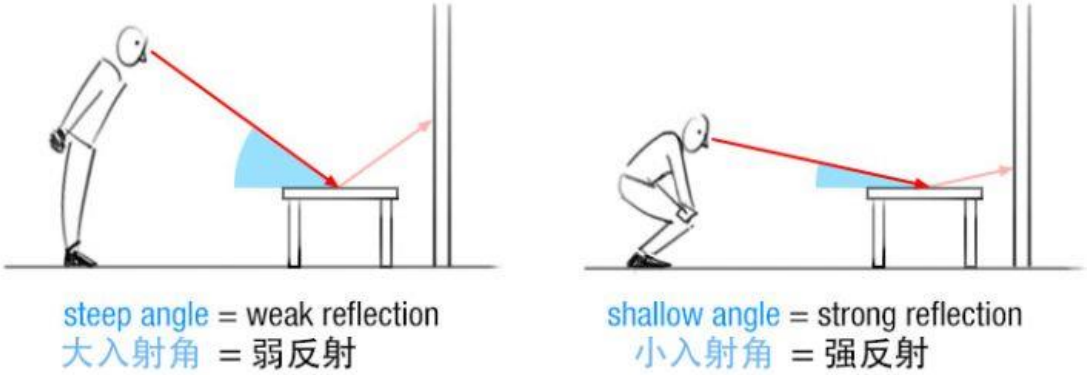


图238.　　入射角特性示例

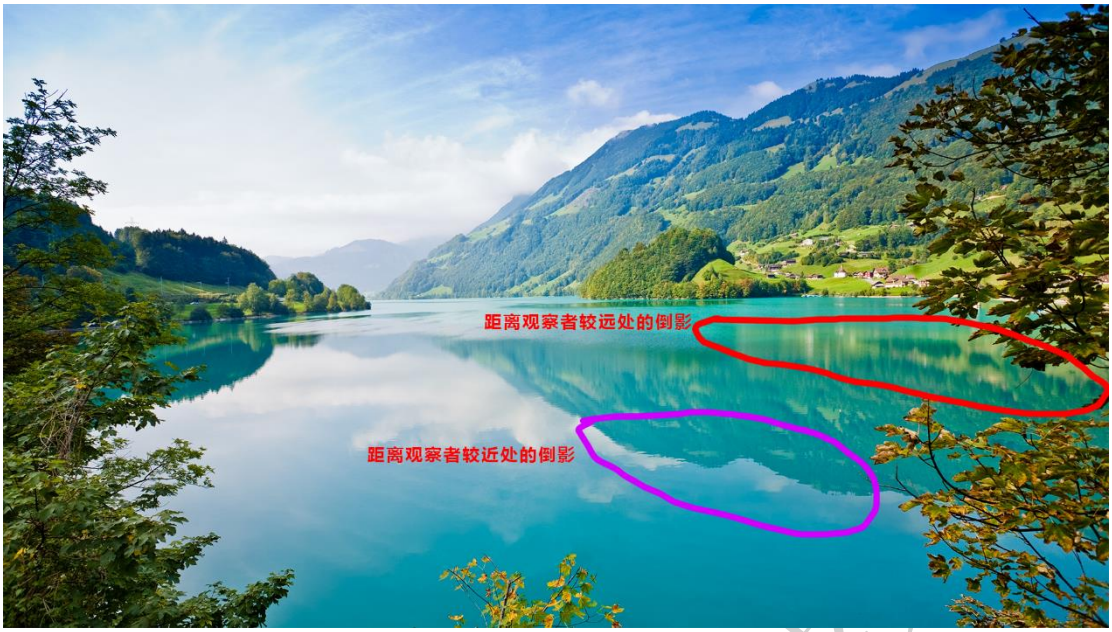


图239.　倒影的清晰程度与观察者和倒影的距离有关

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。

哔哩哔哩次元游戏

图240.　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。

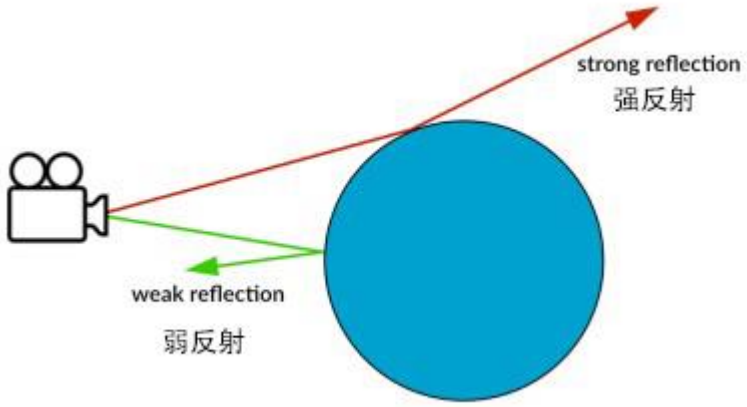


图241.　曲面的反射示例

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242.　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。

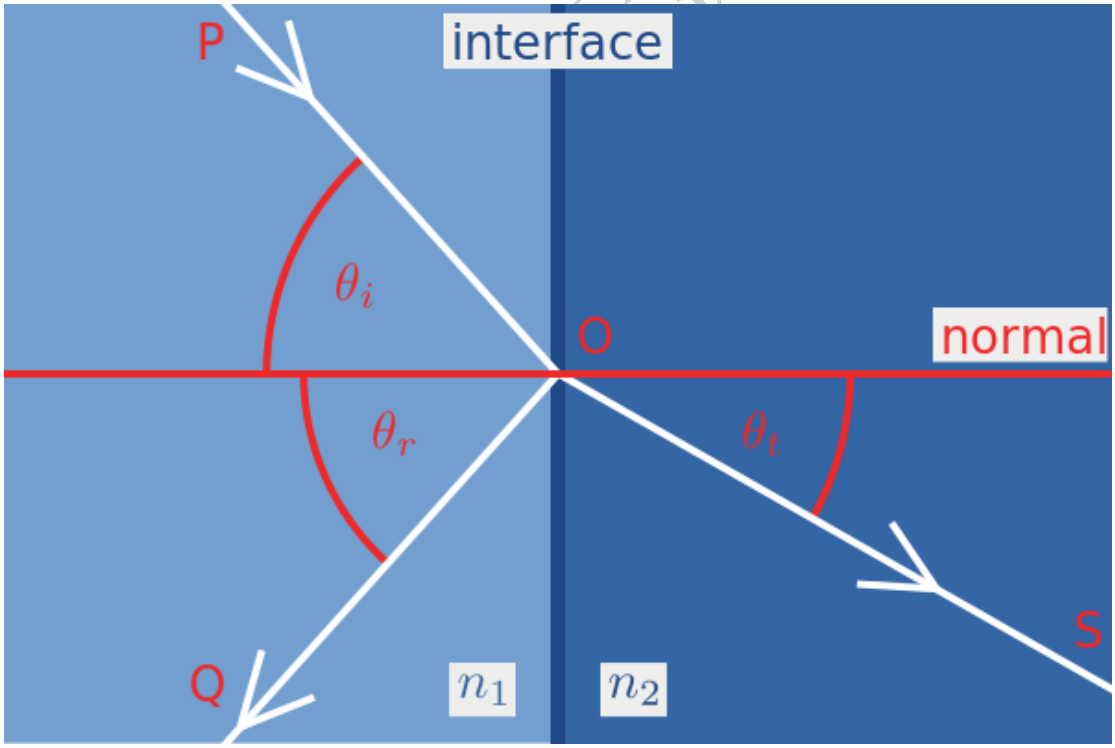


图243.　入射光线、折射光线与反射光线

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。

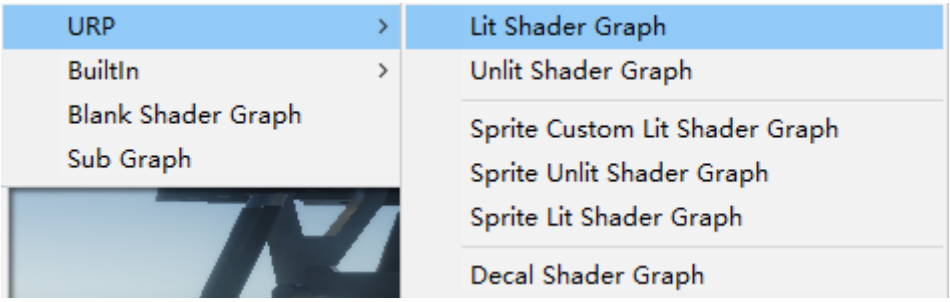


图244.　创建 LitShaderGraph

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。

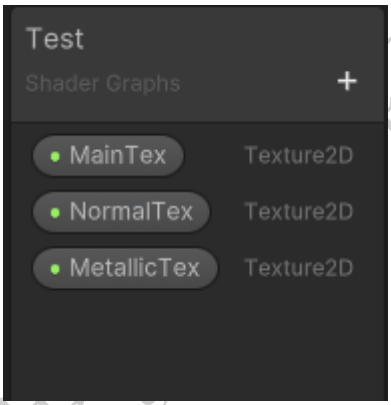


图245.　菲涅尔边缘光的基础纹理参数

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

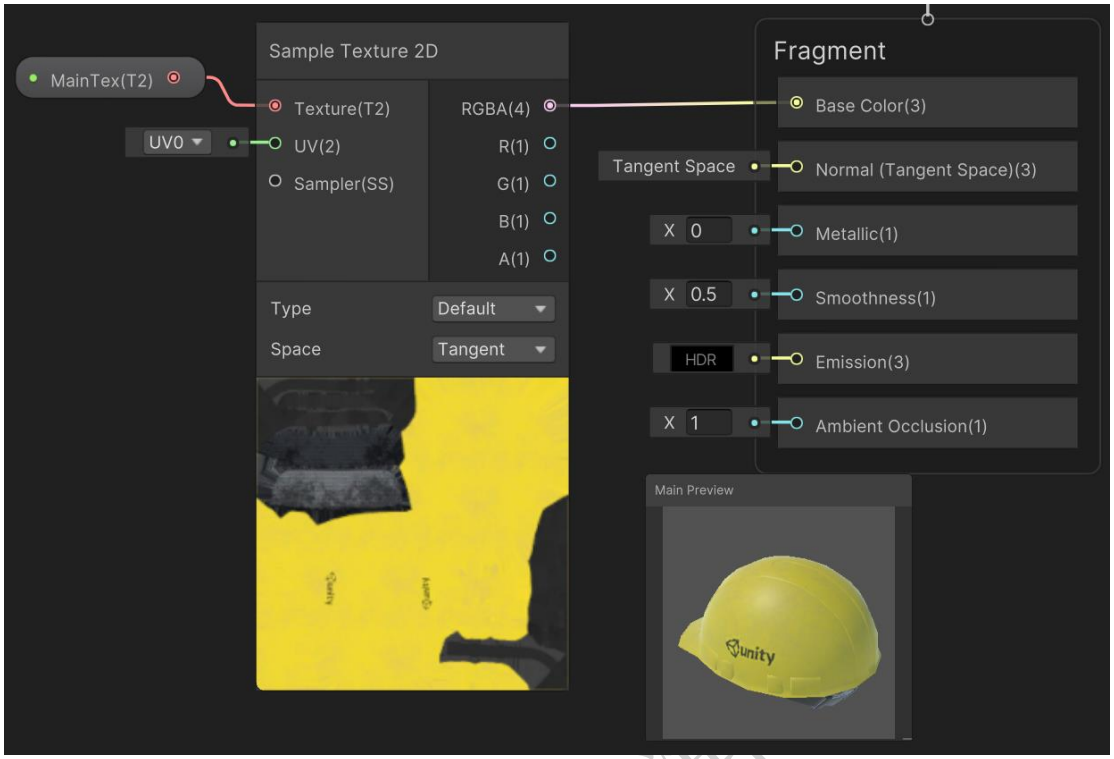


图246.　菲涅尔效果的漫反射贴图

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）。

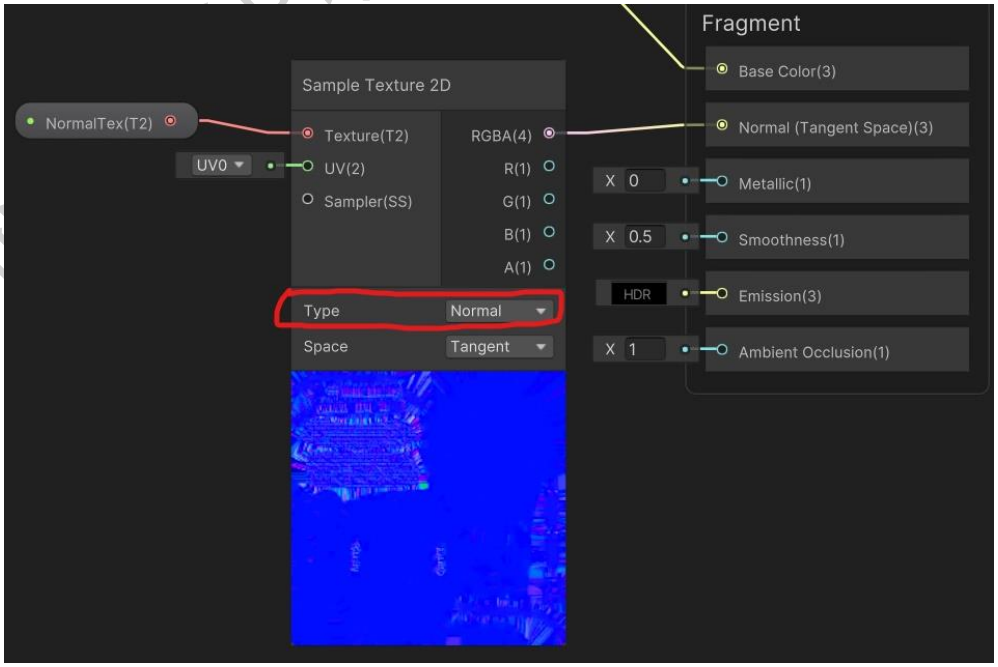


图247.　菲尼尔效果应用法线贴图

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。

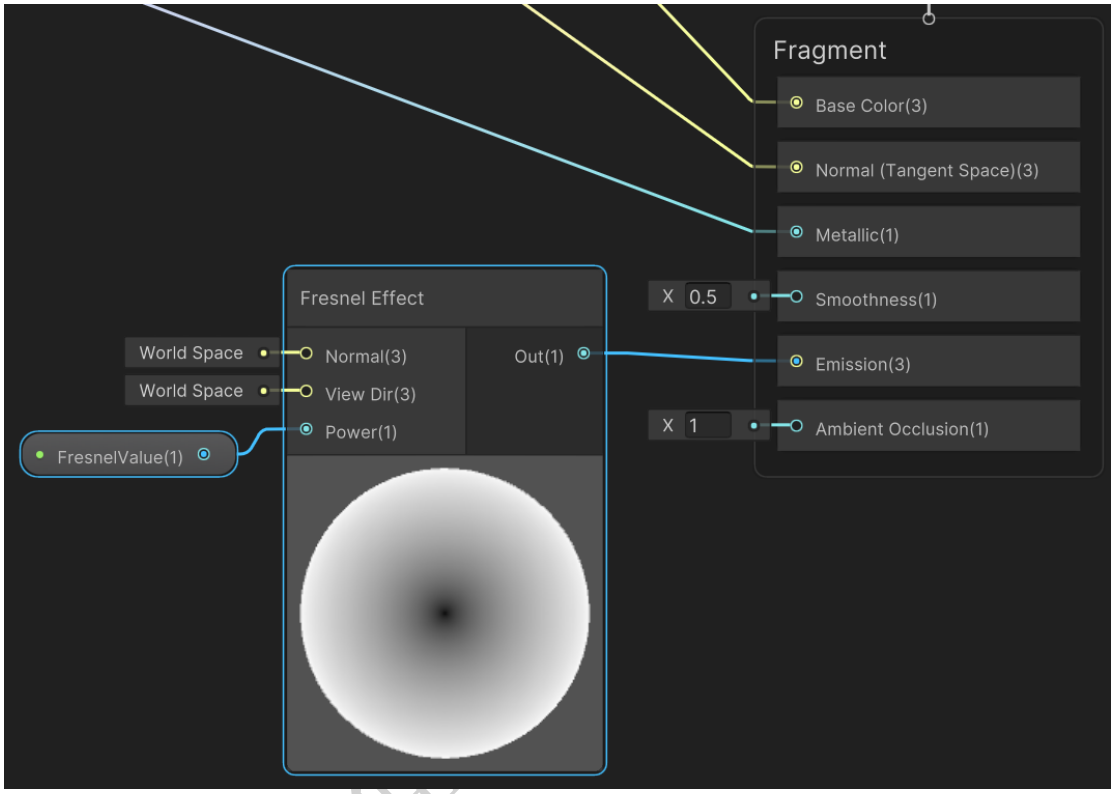


图248. 直接使用菲涅尔节点

也可以使用法线方向和视口方向，去自己完成菲涅尔节点。

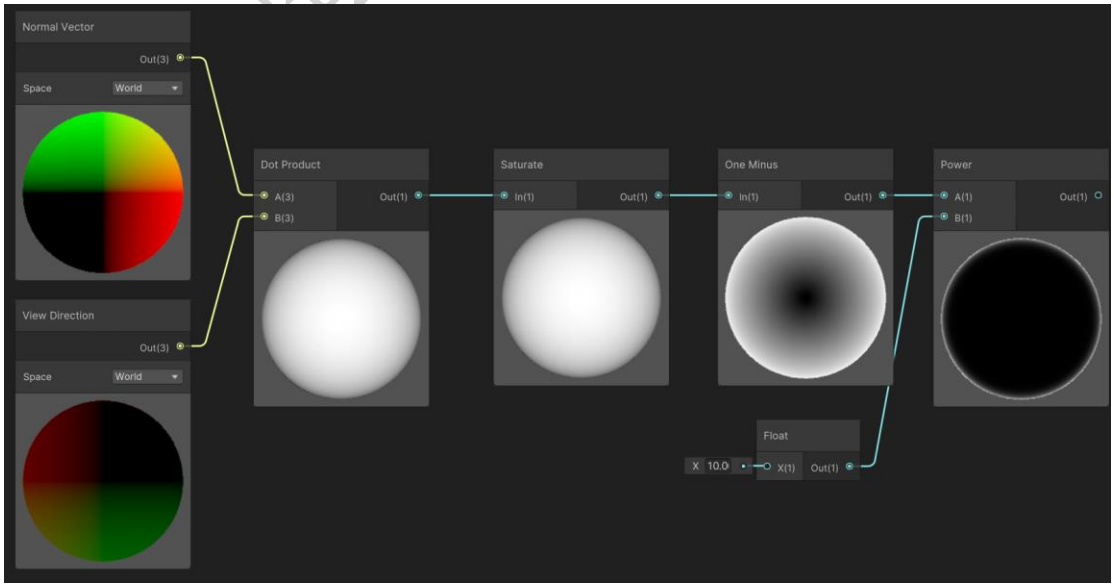


图249. 自建菲涅尔节点

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[20]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。

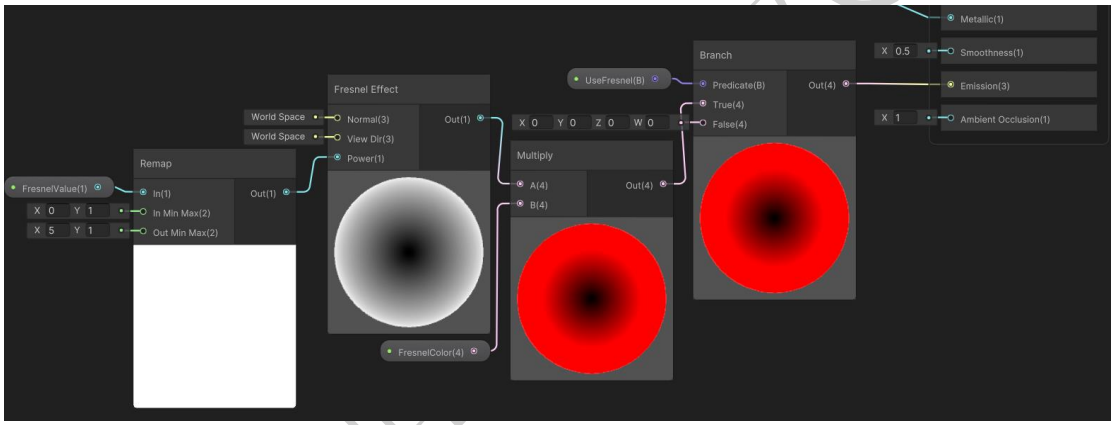


图250. 将 RresnelValue 进行重映射

二十二、3D 法线膨胀描边效果

二十三、后处理描边

https://zhuanlan.zhihu.com/p/50450545

二十四、3D 积雪效果

二十五、3D 广告牌效果(Billboard)

二十六、光学迷彩

[20] 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

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。

唯唯咔咔游戏人生

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。

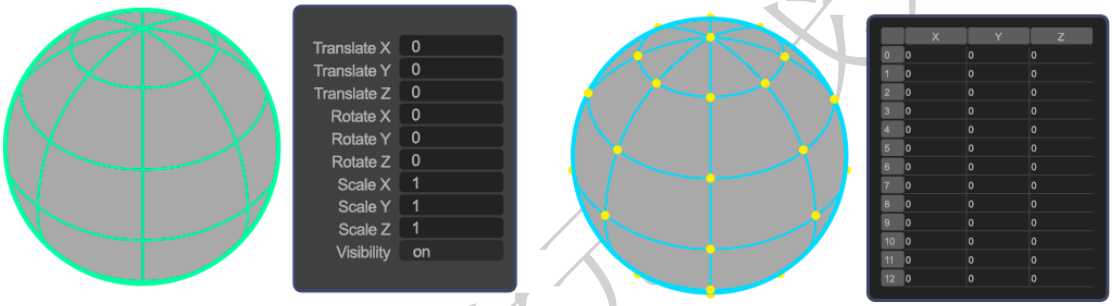


图251. 左边是来自 Maya 的 Transform 节点，右边是 Shape 节点

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

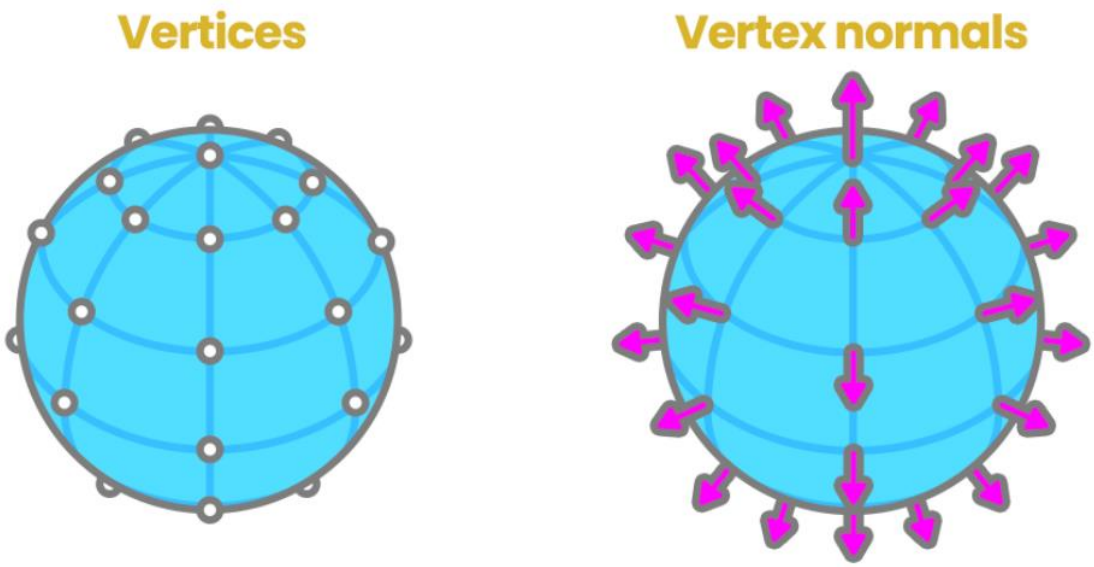


图252.　每个顶点法线的图形表示

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。

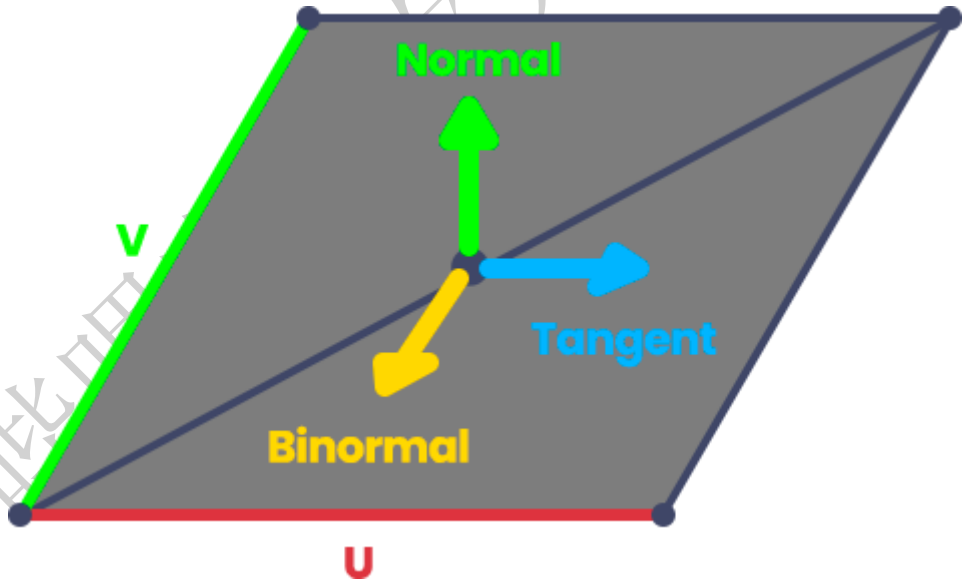


图253.　默认情况下，我们无法在着色器中访问 Binormal，我们必须根据法线和切线计算它。

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。

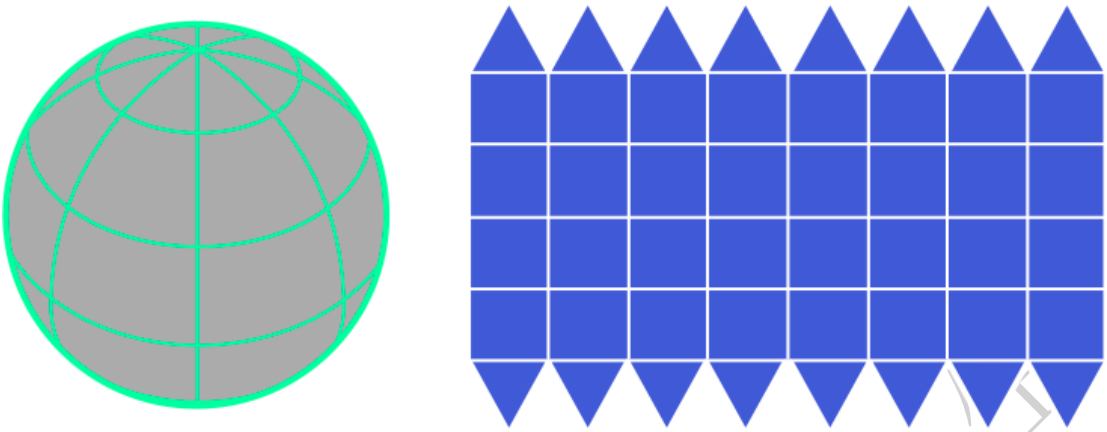


图254.　顶点可以在 UV 贴图中以不同的方式排列

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。

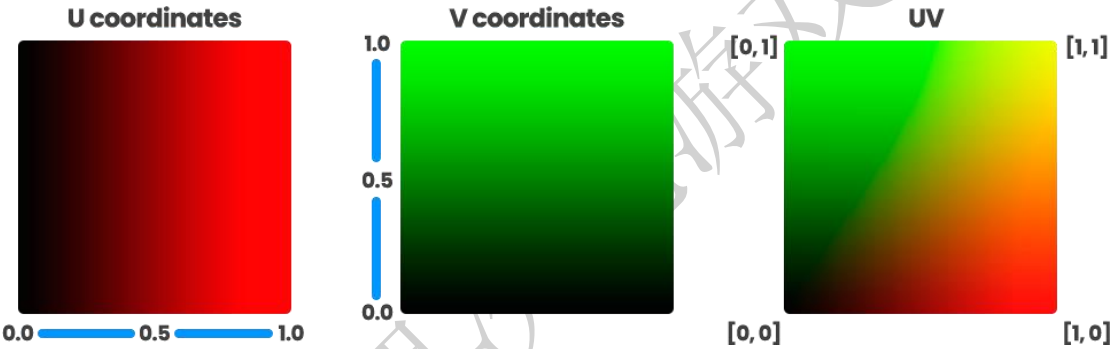


图255.　笛卡尔平面中 UV 坐标的图形参考

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。

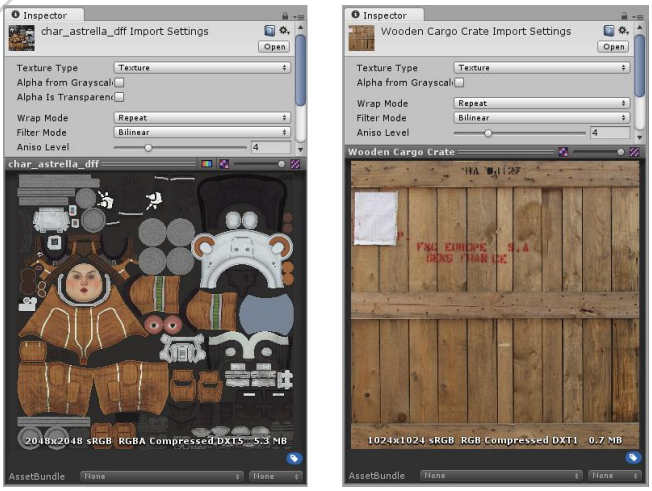


图256.　典型的 Albedo 纹理贴图的两个示例。

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。

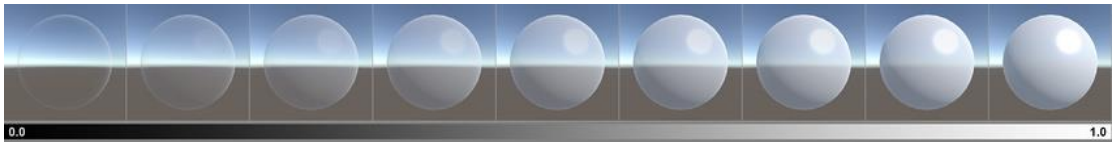


图257.　透明度值范围从 0 到 1，使用适用于逼真透明对象的透明模式

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。

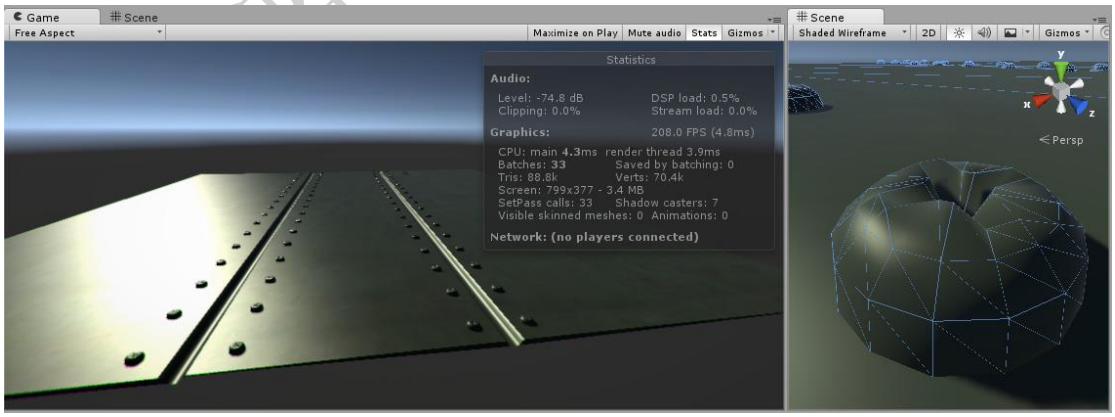


图258.　一张飞机金属片，其细节建模是真实的几何模型

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。

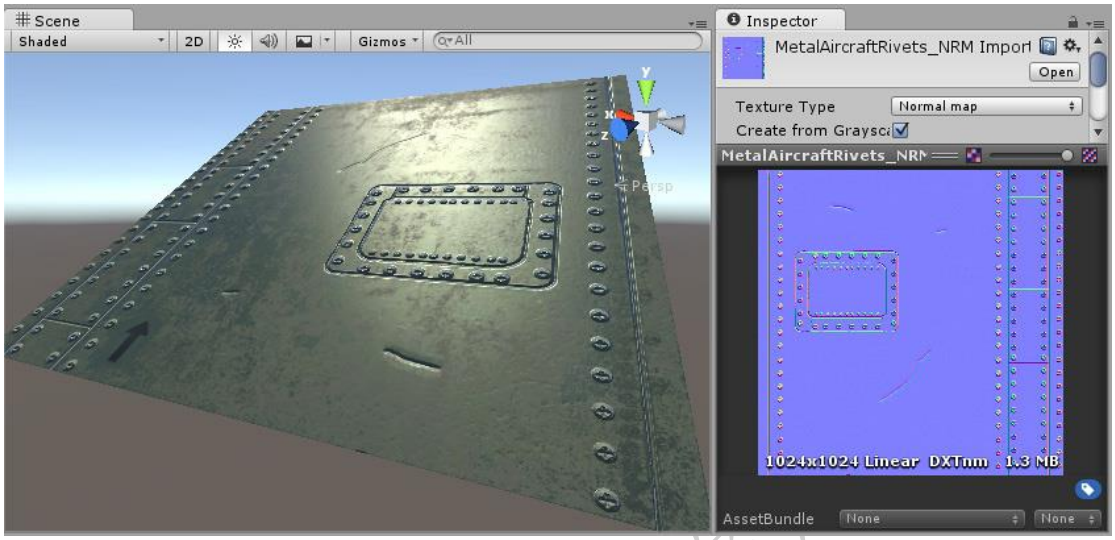


图259. 螺钉、凹槽和划痕在法线贴图中定义，它会修改光线在这个低多边形平面表面的反射方式，给人以真实 3D 细节的感觉

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。

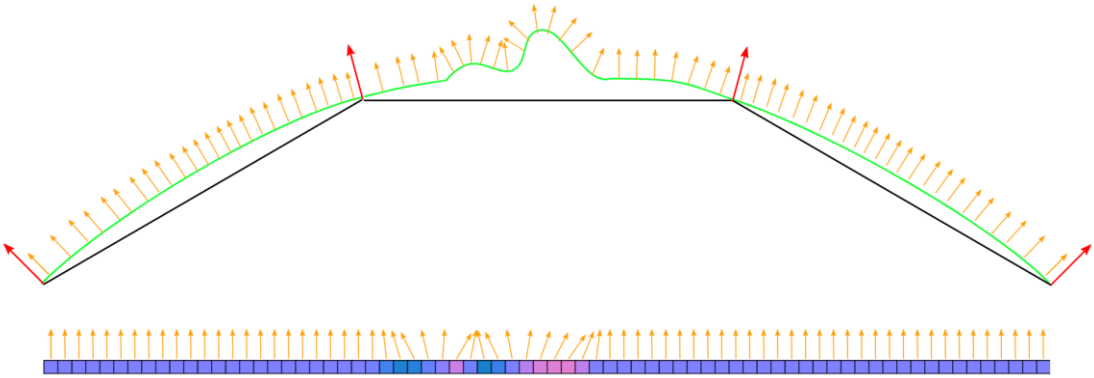


图260. 跨三个多边形的法线映射，被视为一张 2D 图

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。

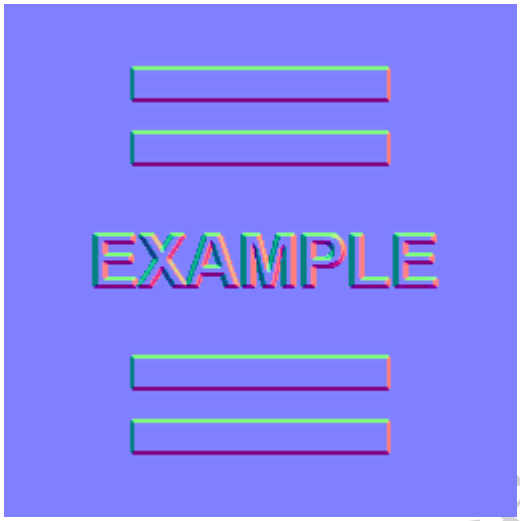


图261.　一个示例法线贴图纹理

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。

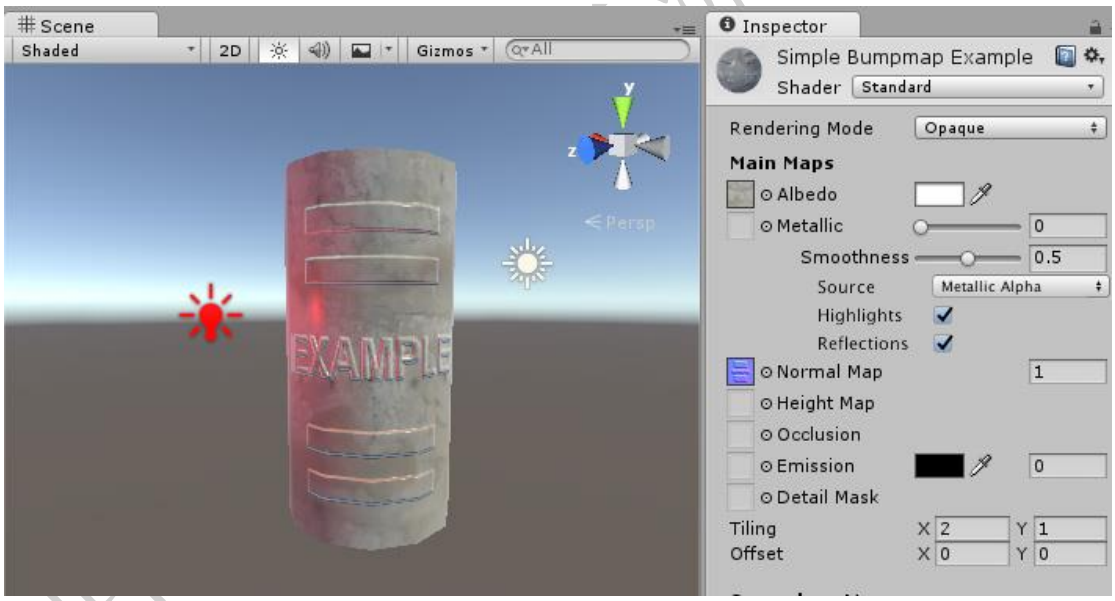


图262.　应用法线贴图的示例

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263.　石墙纹理及其对应的法线贴图纹理

图264.　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；

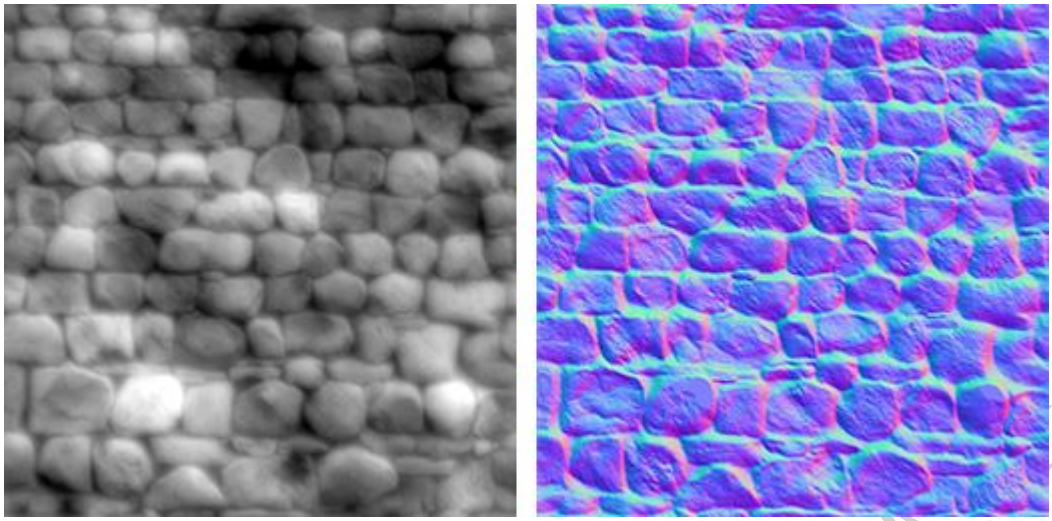


图265.　左侧是用于凹凸贴图石墙的高度图。右侧是用于凹凸贴图石墙的法线贴图

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。

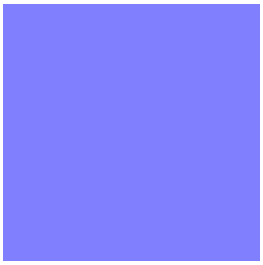


图266.　仅使用 #8080FF 的法线贴图，它转换为 0, 0, 1 或“向上”的法线向量

(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267.　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268.　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269.　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270.　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”。

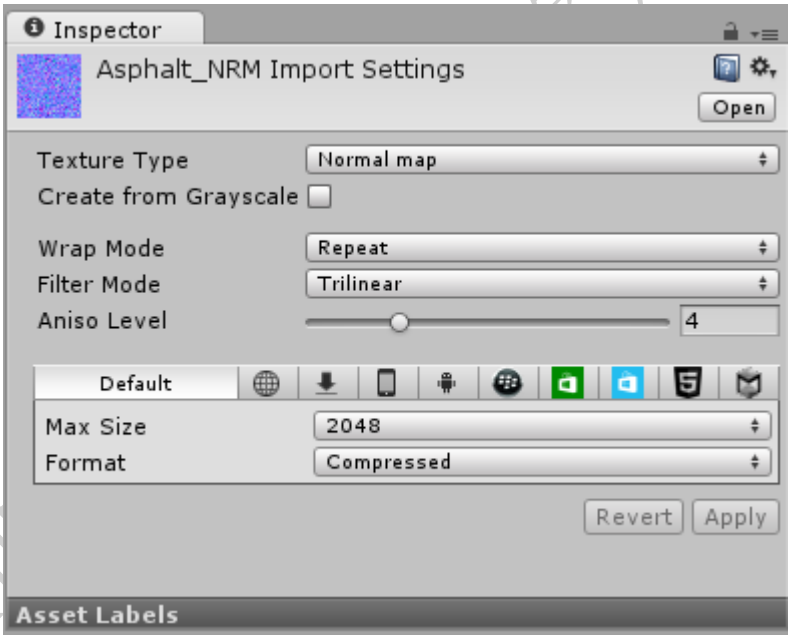


图271.　将导入的 Texture，设置为 Normal Map

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272.　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。

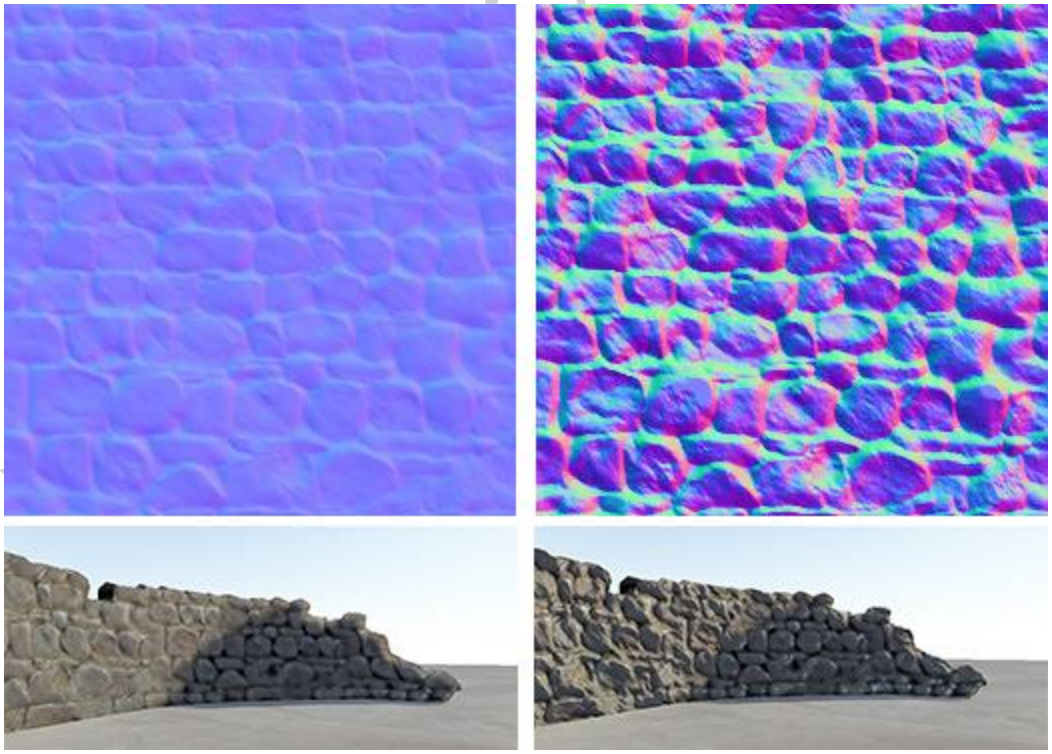


图273.　将 Height Map 作为 Normal Map 导入时的低（左）和高（右）凹凸设置，及对模型产生的影响

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。

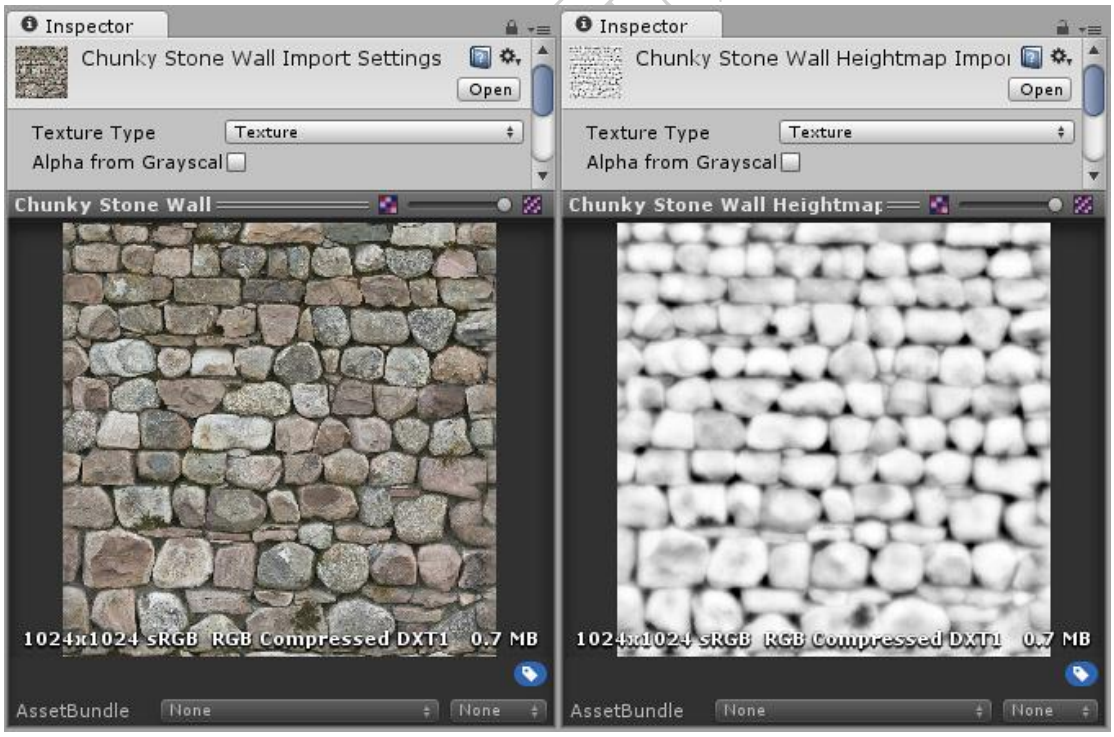


图274. Albedo Map 和要匹配的 Height Map。

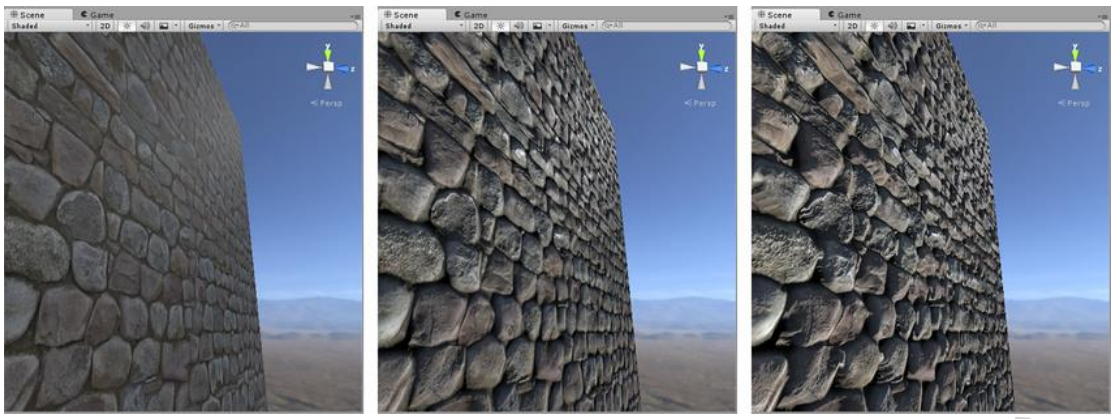


图275.　Hight Map、Normal Map 和 Albedo Map 的应用

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276.　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

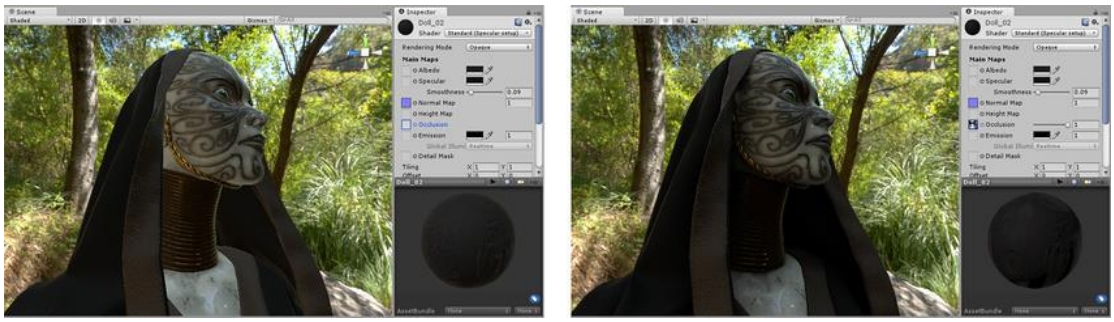


图277.　应用 Occlusion Map 之前和之后对比

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。

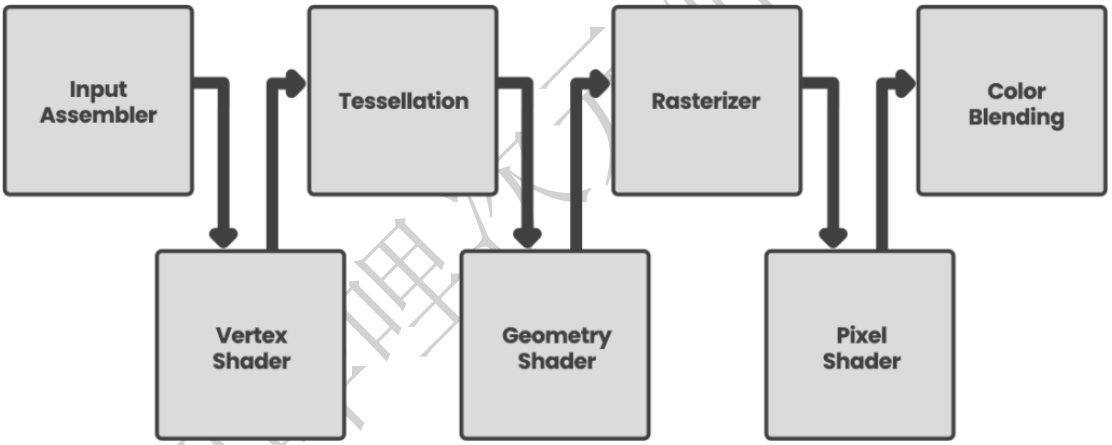


图278.　逻辑渲染管道

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。

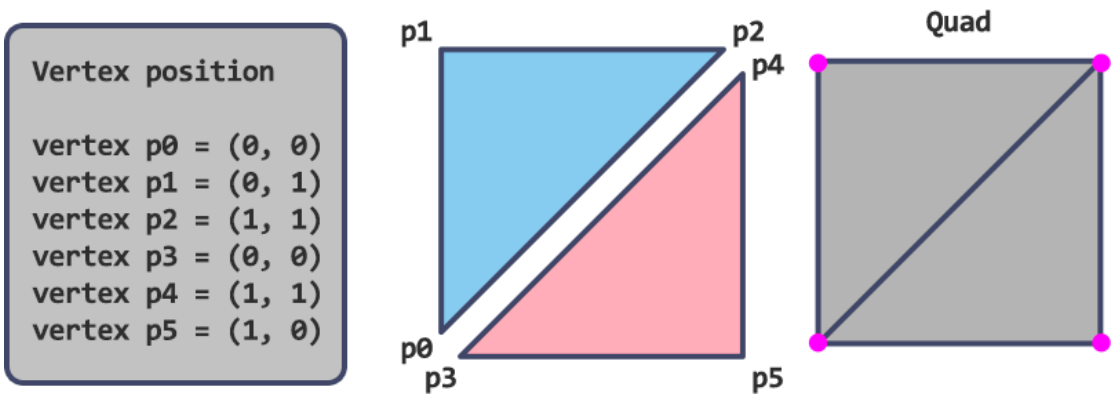


图279.　渲染图元

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。

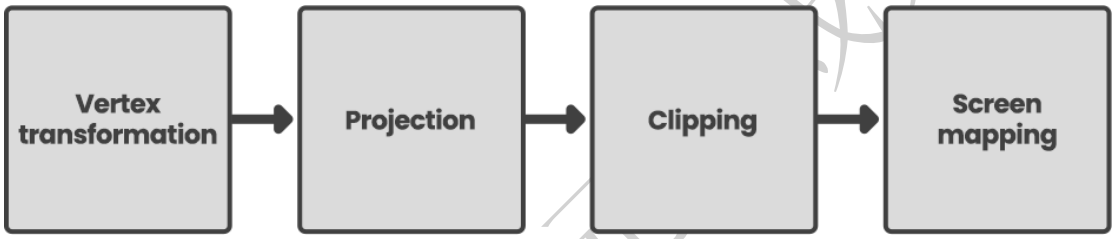


图280.　几何阶段子过程

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。

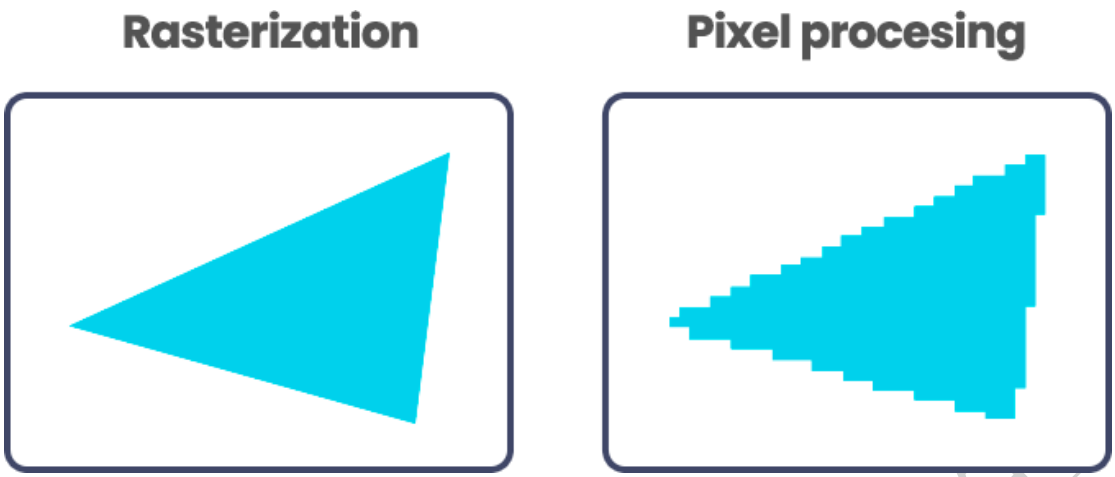


图281.　几何图形覆盖的区域在屏幕上转换为像素

五、 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。

哔哩哔哩沈沉游戏人生

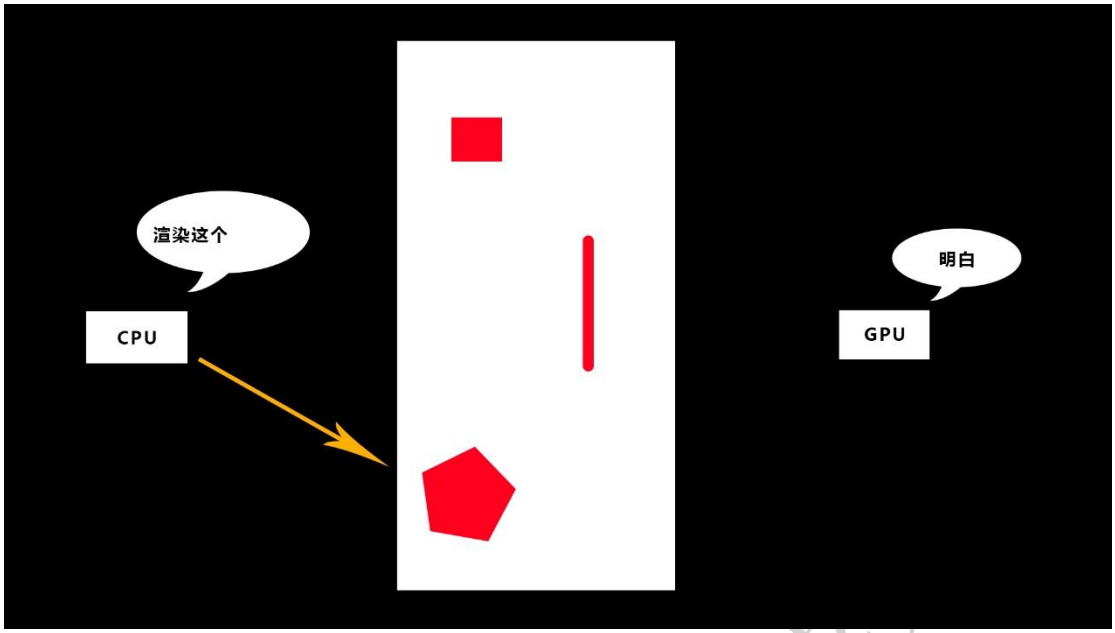


图282.　Cpu 通过调用 Draw Call 告知 GPU 开始进行一个渲染过程。

六、 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。

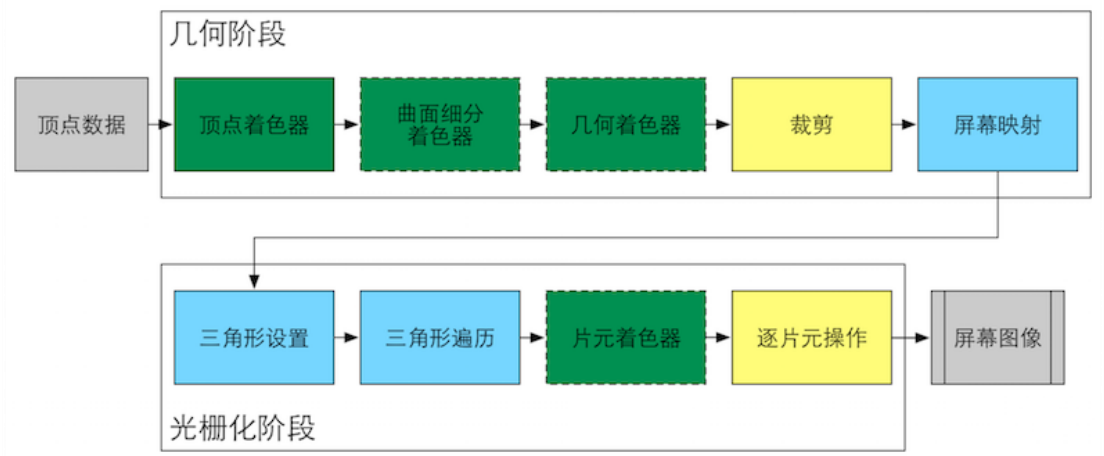


图283.　GPU 的渲染流水线实现

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。

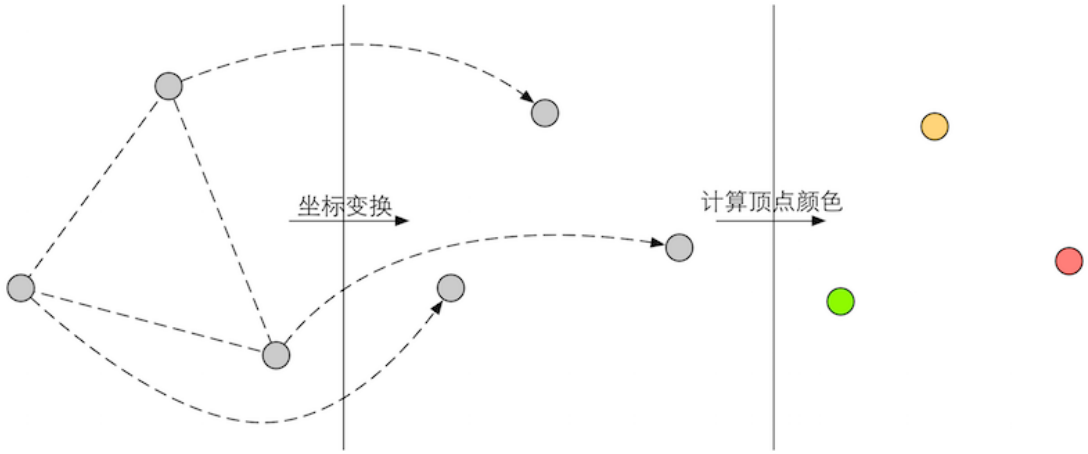


图284.　在顶点着色器中对顶点位置进行坐标变换并计算顶点颜色的过程

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285.　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。

2. 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。

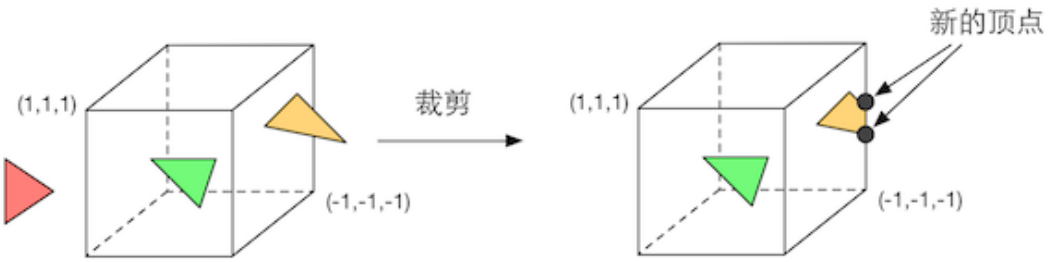


图286.　将图元裁剪到单位立方体内

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。

3. 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。

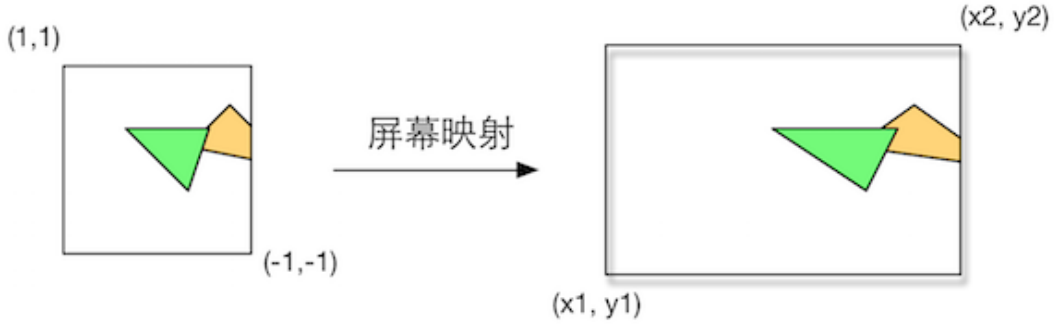


图287.　屏幕映射

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。

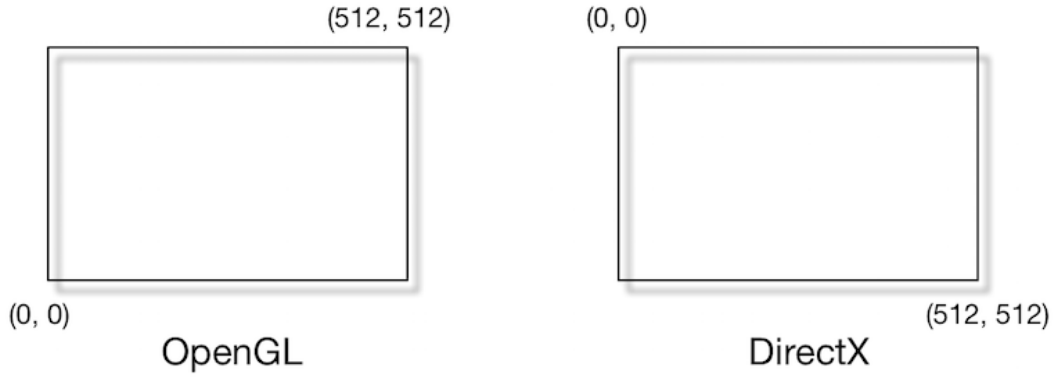


图288.　OpenGL 和 DirectX 屏幕坐标系的差异

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。

4. 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项点，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。

5. 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。

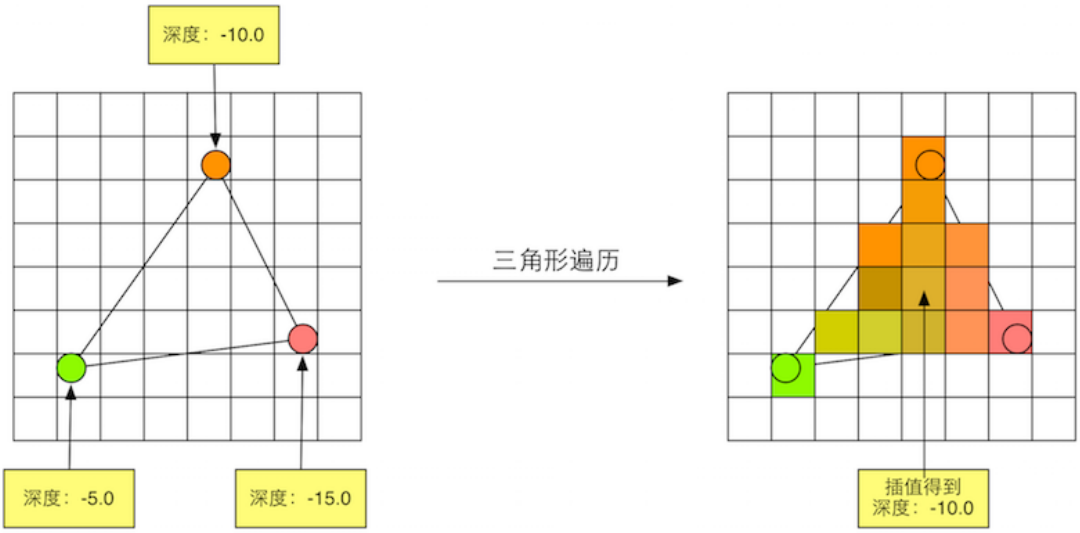


图289. 三角形遍历的过程

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。

6. 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。

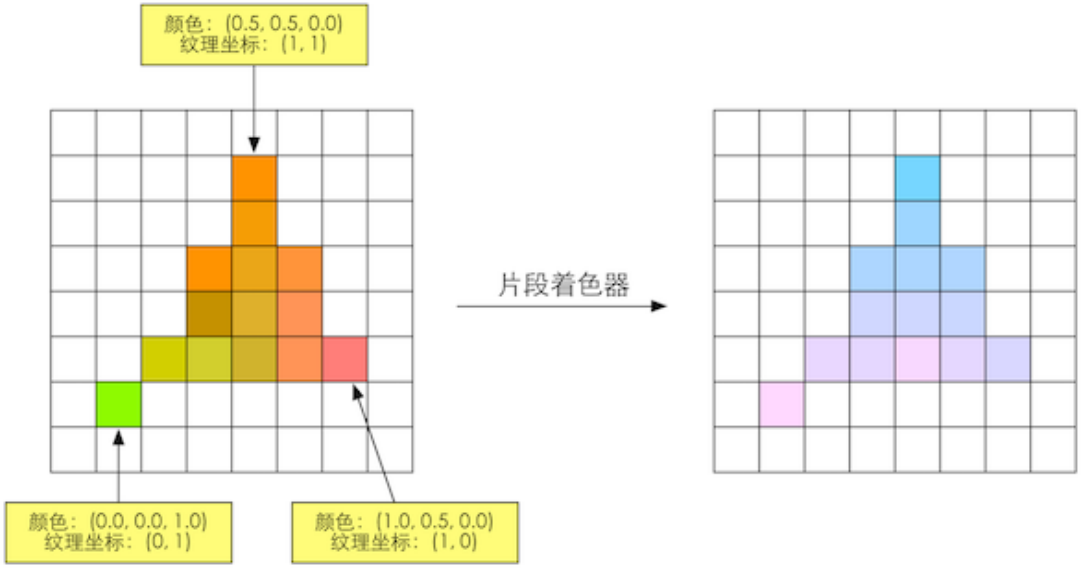


图290.　根据三角形遍历差值后的片元信息，片元着色器计算该片元的输出颜色

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）。

7. 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291.　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。

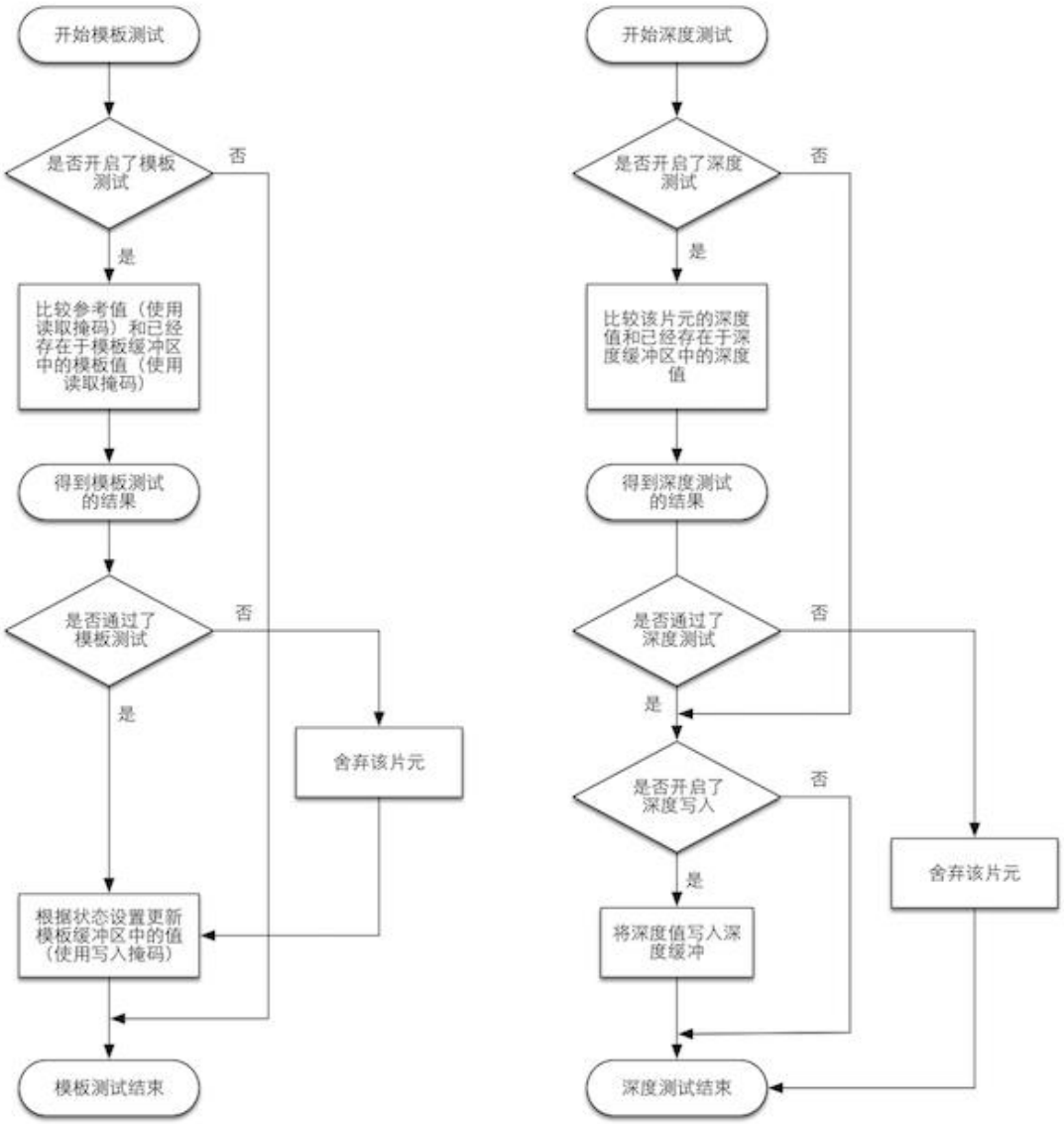


图292.　　模板测试和深度测试的简化流程图

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

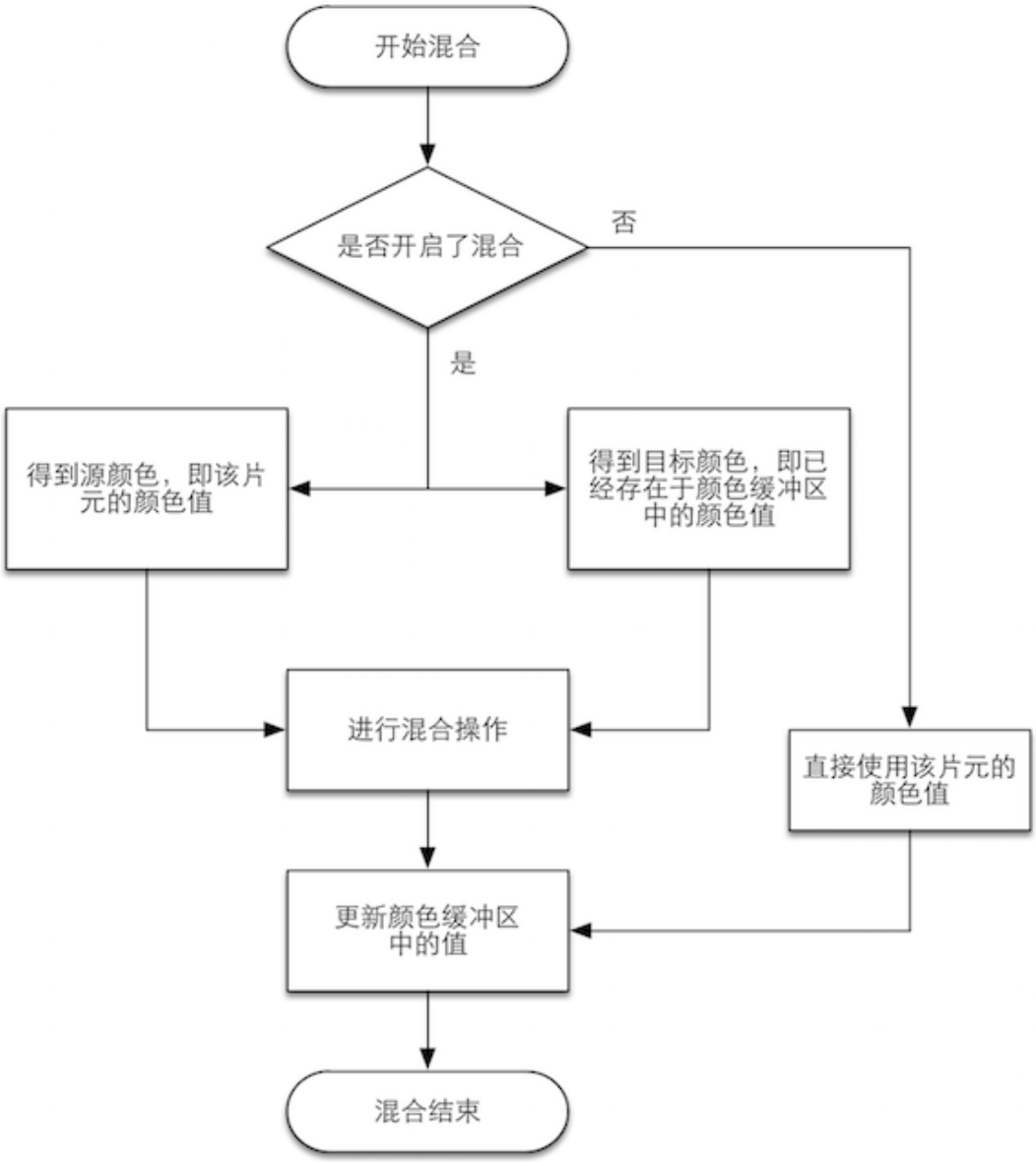


图293.　混合操作的简化流程图

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！

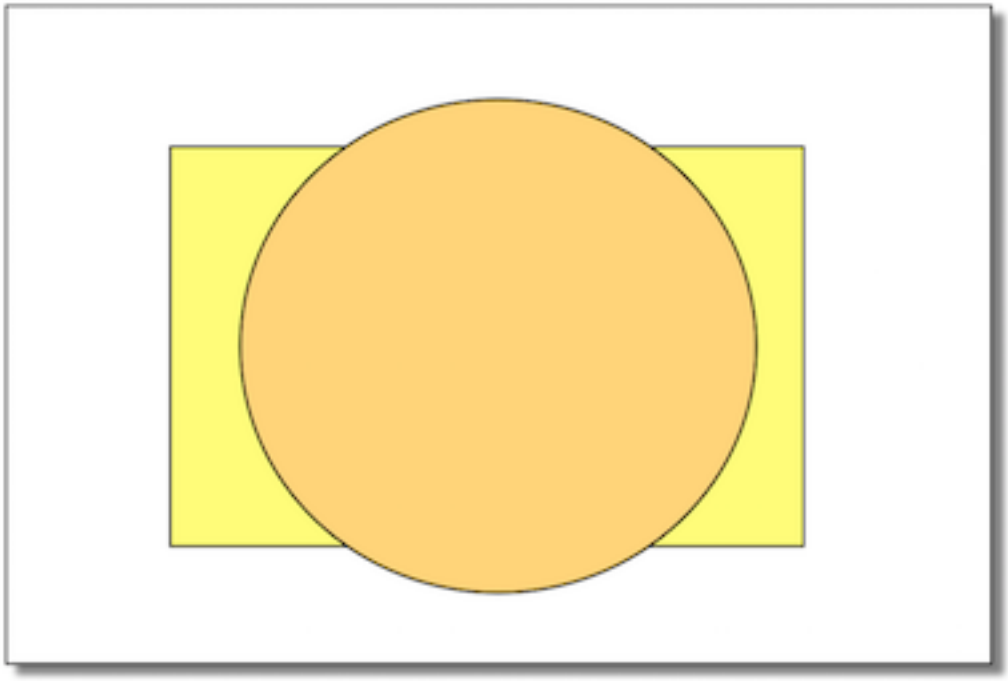


图294.　示例

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。

七、 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）。

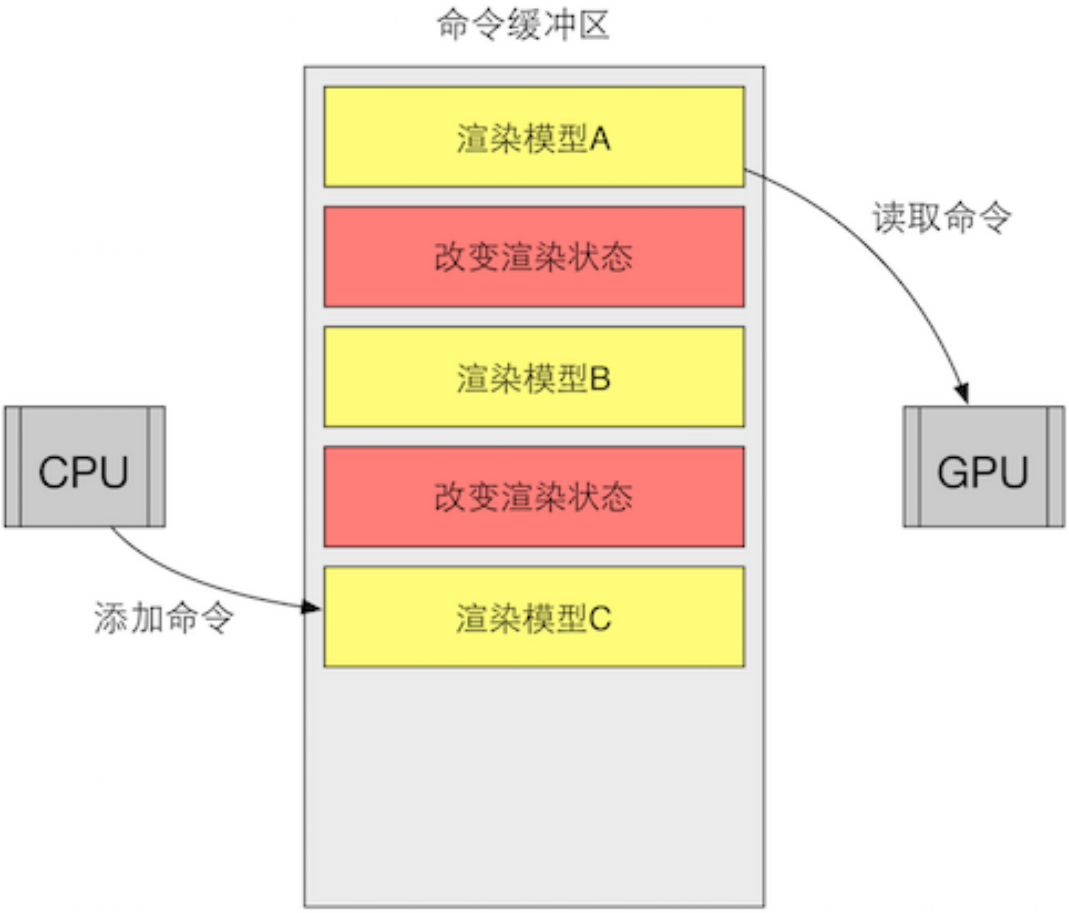


图295. 命令缓冲区

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。

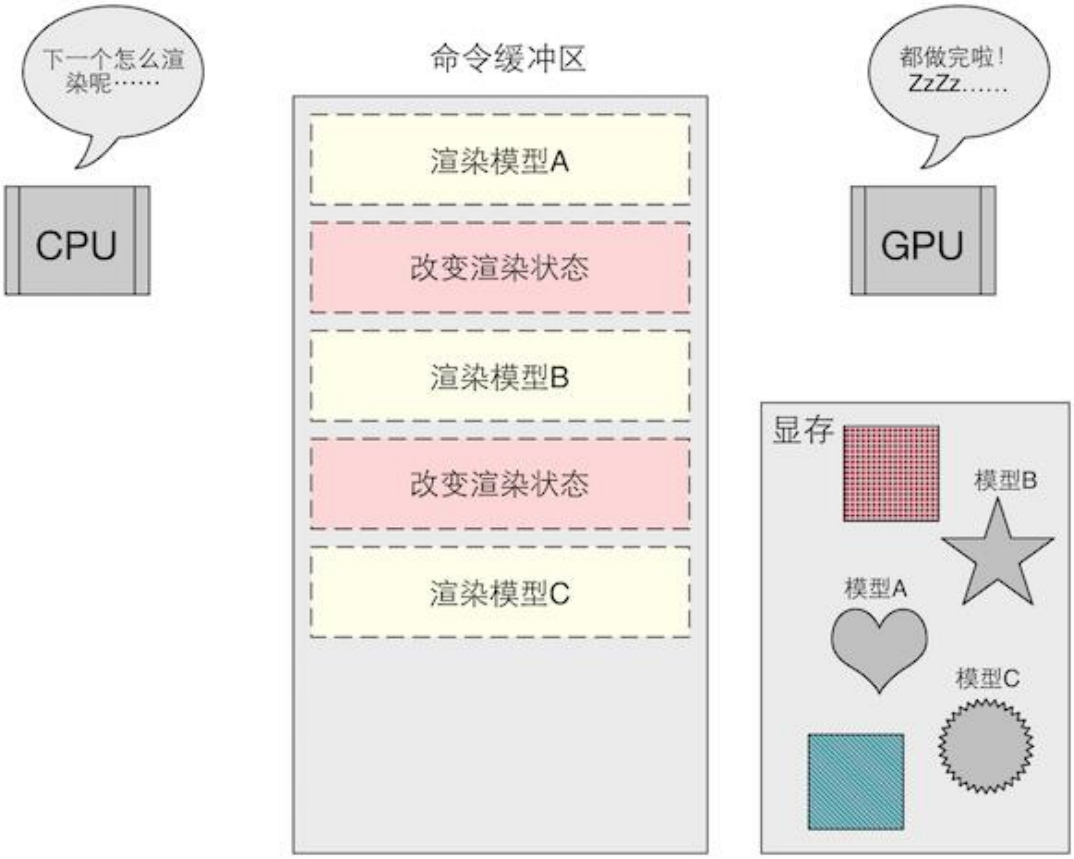


图296. 命令缓冲区中的虚线方框表示 GPU 已经完成的命令

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。

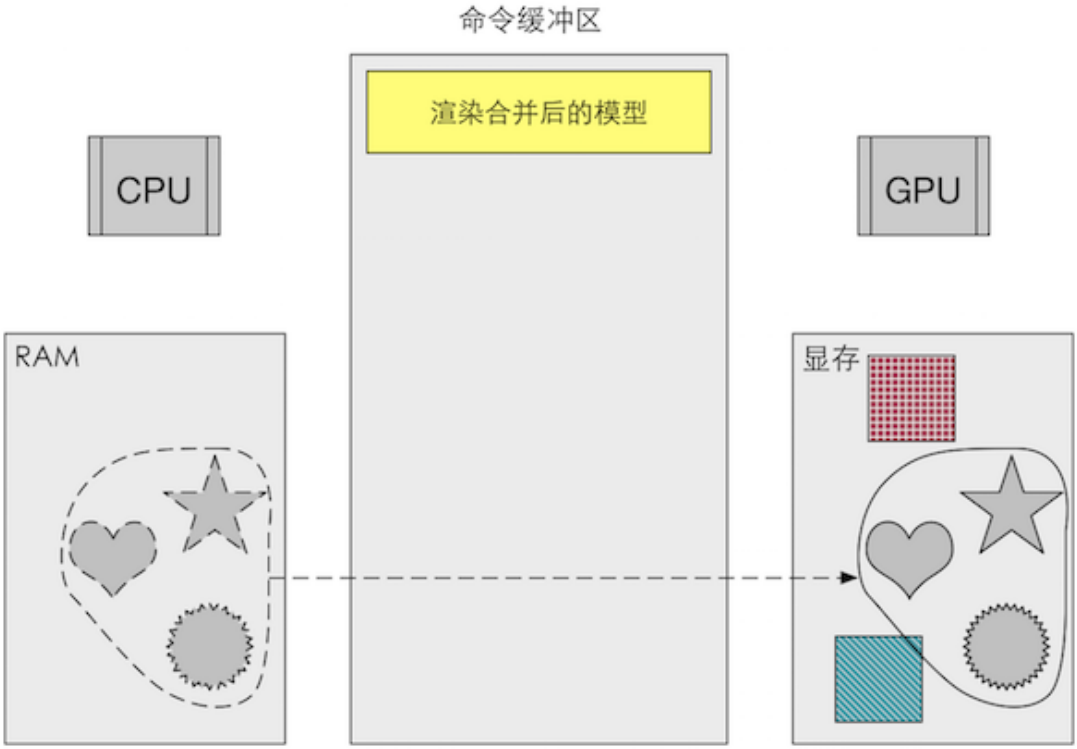


图297.　利用批处理

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。

八、　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 ...
}

// 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)
2D	"defaulttexture"{}	_2D ("2D ",2D)=""{}
Cube	"defaulttexture"{}	_ Cube ("Cube ", Cube)=

哔哩哔哩次元游戏人生

		"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
        {
            CGPROGRAM
            ...
```

哔哩哔哩次元游戏人生

```
                // 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。

a. 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：

```
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 KeywordEnum Drawer。

哔哩哔哩次元游戏人生

b. 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);
```

哔哩哔哩次元游戏人生

```
                #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。

c. 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。

d. 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。

哔哩哔哩次元游戏人生

[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。

e. 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
// 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。

哔哩哔哩次元游戏人生

“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]
    pass
    {
    }
}
```

SubShader 中定义了一系列 Pass 以及可选的状态([RenderSetup])和标签([Tags])设置。

唯理科技次元游戏人生

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。

“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	Tags{"Queue"="Transparent"}

	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	
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	Tags{"PreviewType"="Plane"}

哔哩哔哩次元游戏人生

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。

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。

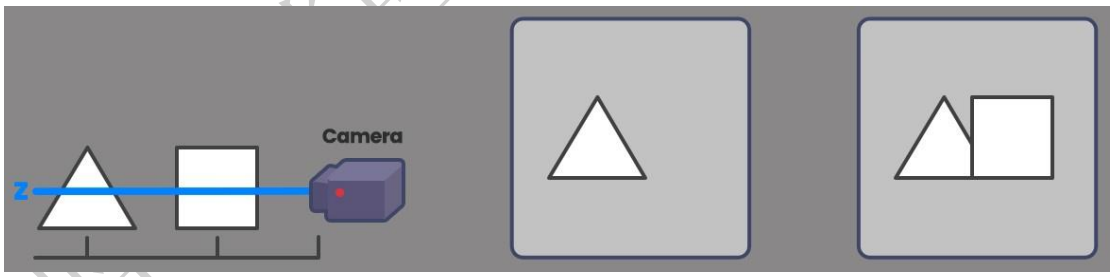


图298. GUP 绘制演示

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" }

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

哔哩哔哩次元游戏人生

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。

哔哩哔哩次元游戏人生

```
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。
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

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

哔哩哔哩次元游戏人生

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
```

哔哩哔哩次元游戏人生

```
        }
    }
```

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。

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。

哔哩哔哩次元游戏人生

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。

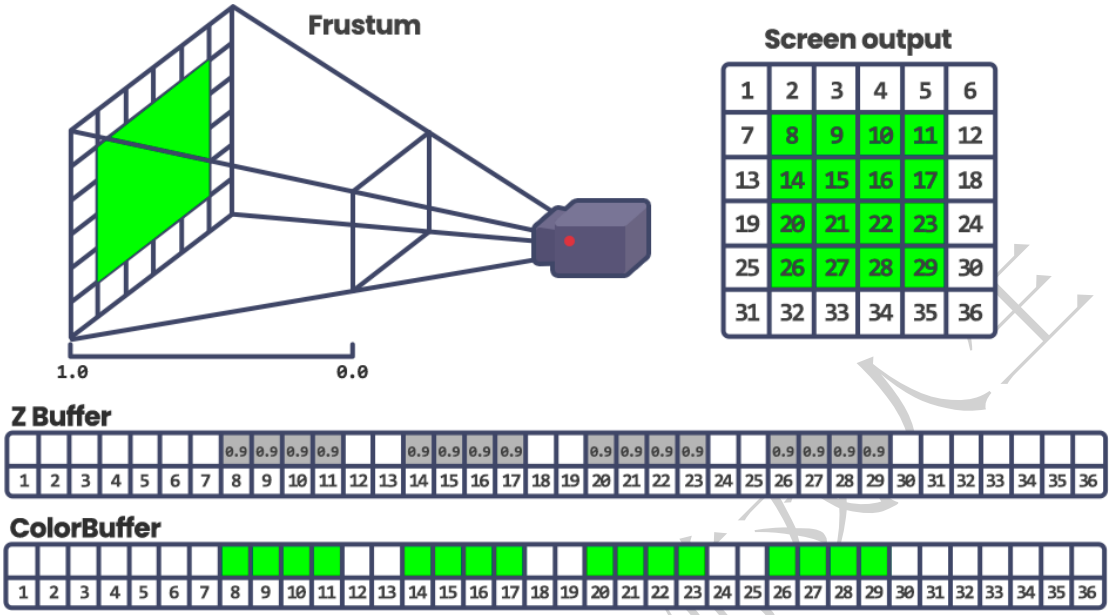


图299.　Z-Buffer 存储场景中物体的深度，Color Buffer 存储 RGBA 颜色信息

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。

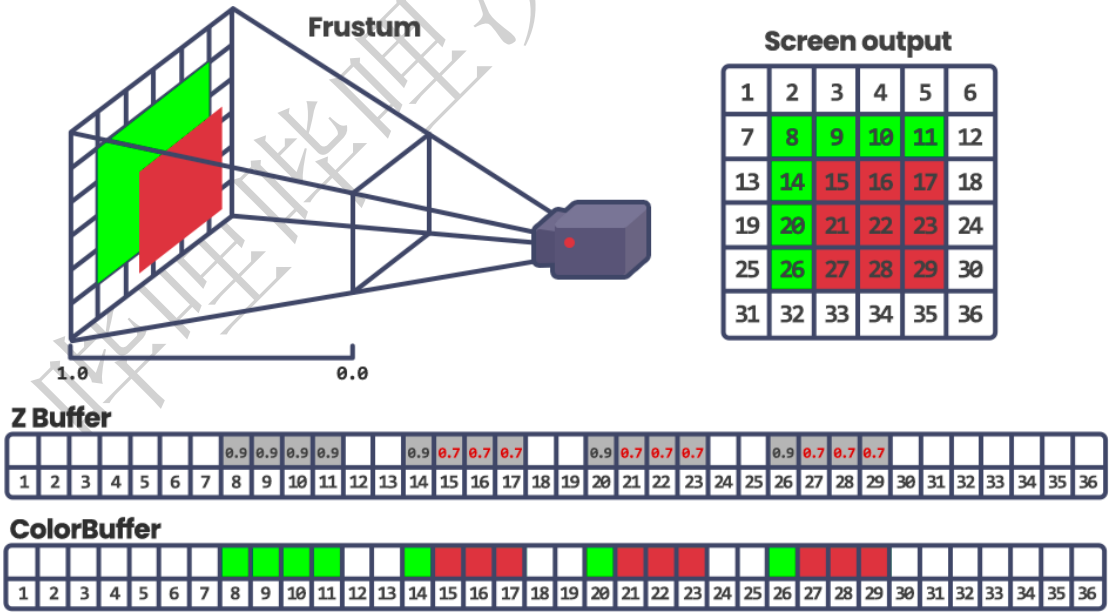


图300.　距离更近的图像

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
```

哔哩哔哩次元游戏人生

```
{
    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。

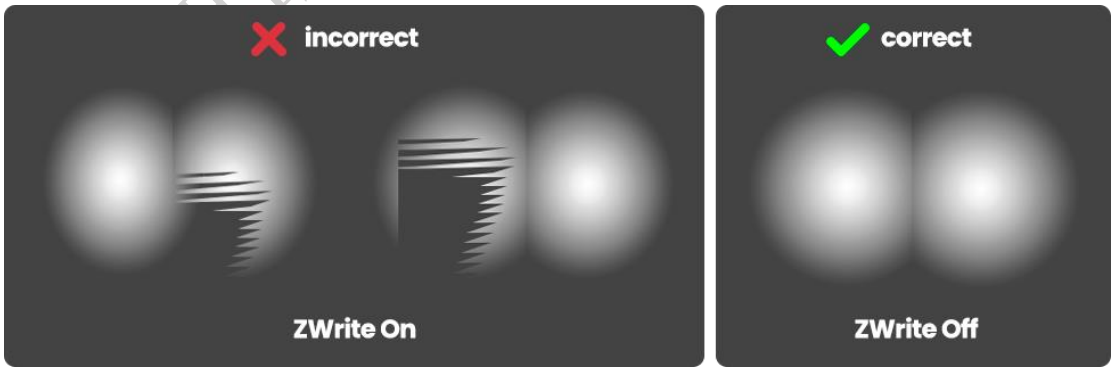


图301. z-fighting

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。

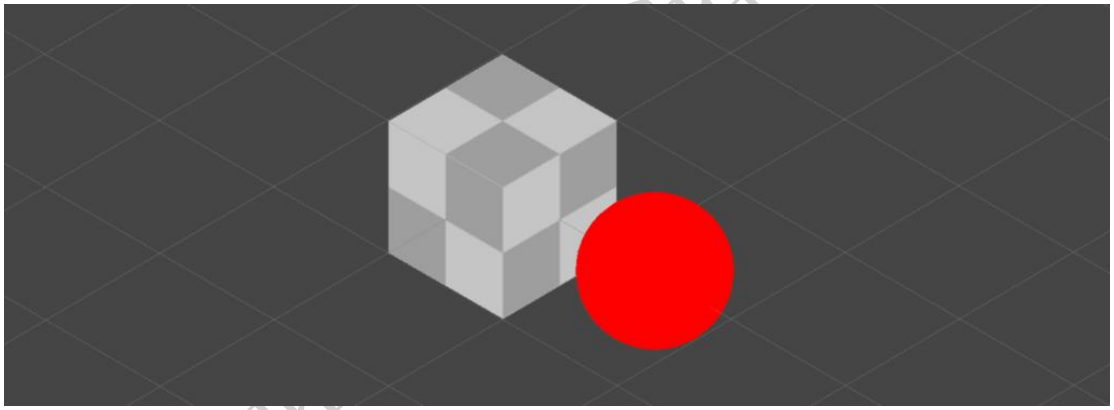


图302. Sphere Is Forward Of Square

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。

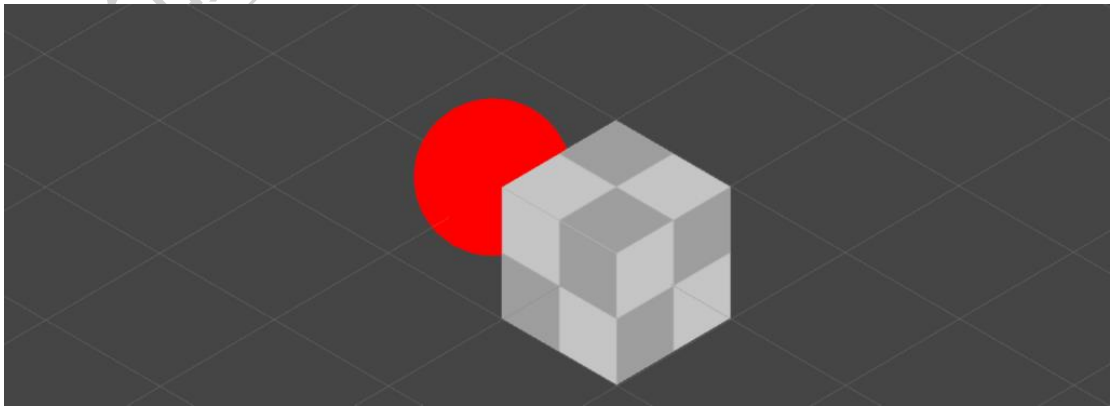


图303. Square Is Forward Of Sphere

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。

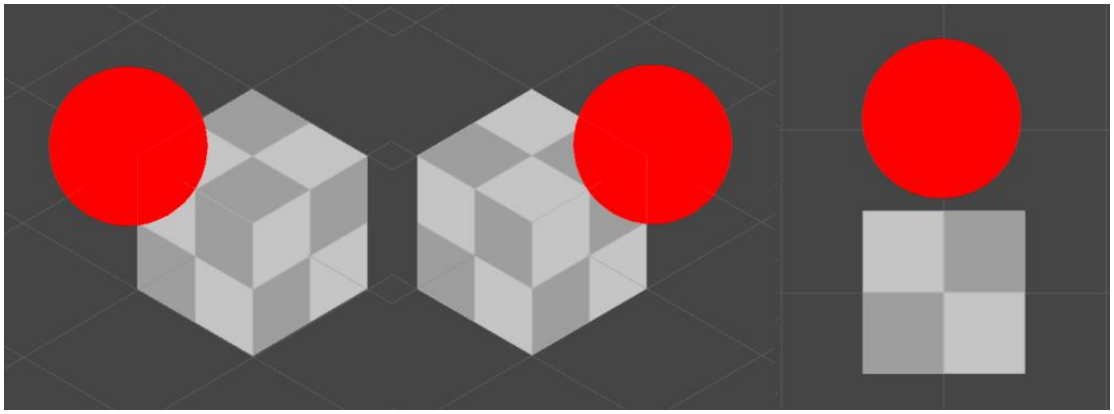


图304.　ZTest Always

```
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。

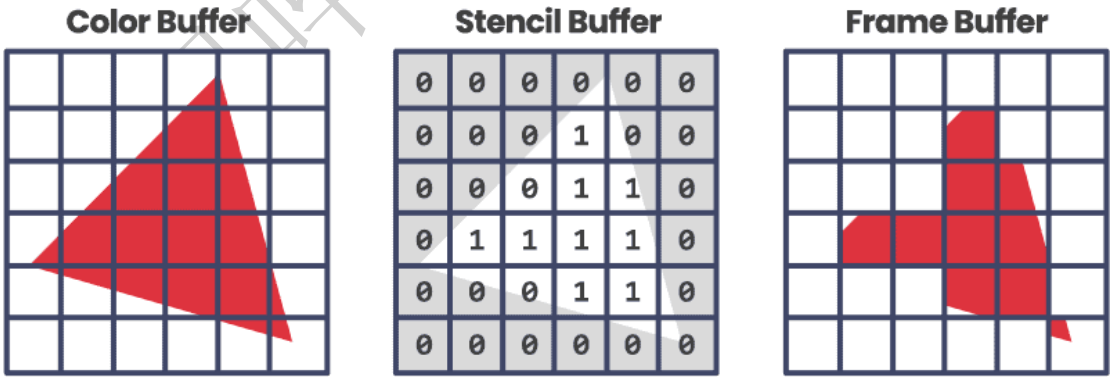


图305.　模板缓冲

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306. 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)。

哔哩哔哩次元游戏人生

图307.　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
    }
}
```

哔哩哔哩次元游戏人生

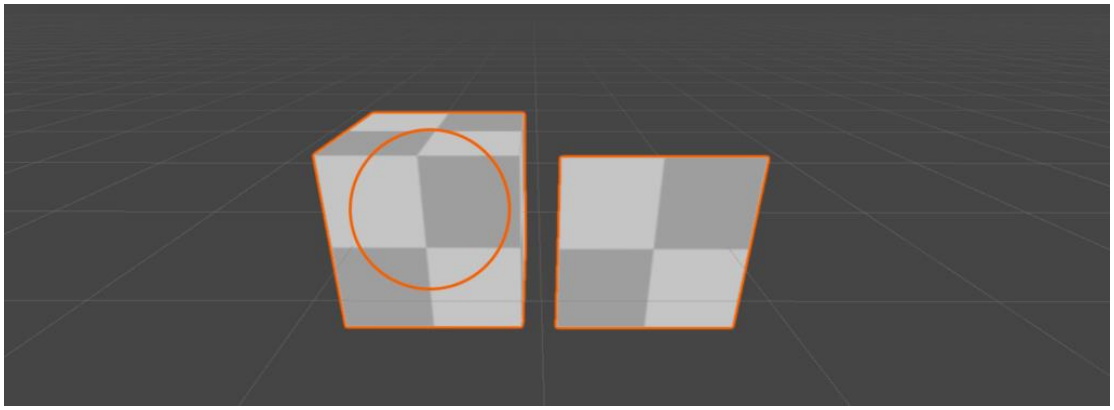


图308.　模板示例

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
    }
}
```

哔哩哔哩次元游戏人生

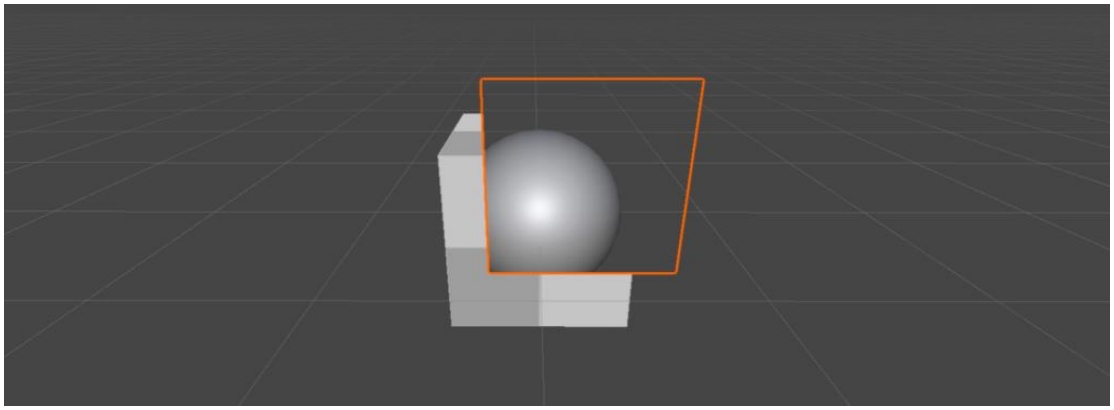


图309.　遮罩

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。

九、　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
            }
        }
    }
```

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

哔哩哔哩张无游戏人生

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

哔哩哔哩次元游戏人生

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。

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

哔哩哔哩次元游戏人生

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
}
```

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，

哔哩哔哩次元游戏人生

	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
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。

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。

哔哩哔哩次元游戏人生

```
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
}
```

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;
```

哔哩哔哩次元游戏人生

```
            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);
            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

哔哩哔哩次元游戏人生

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)。

哔哩哔哩次元游戏人生

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入参数，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。

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。

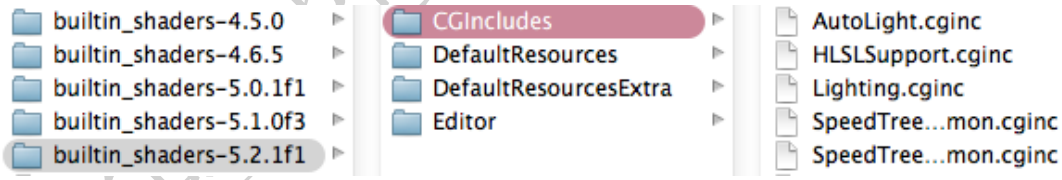


图310. Unity 的内置着色器

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
UnityGloballlumination.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
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;
        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，

哔哩哔哩次元游戏人生

	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))

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

哔哩哔哩次元游戏人生

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

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，括号内是需要传入的两个变量，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
UnitylmageBasedLighting,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、 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311.　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。

哔哩哔哩次元游戏人生

```
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

// #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 stange 和 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。

十一、 Unity 中的基础光照

十二、 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。

哔哩哔哩次元游戏人生

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。

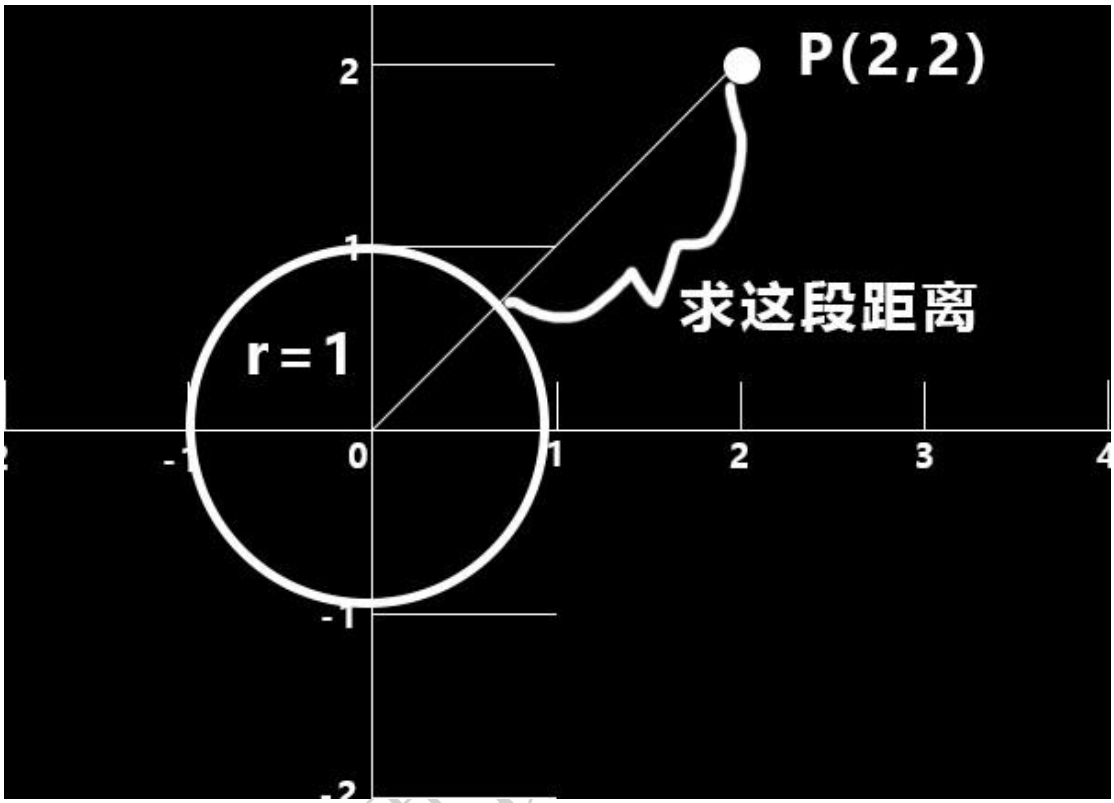


图312.　圆心在坐标轴原点

```
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。）

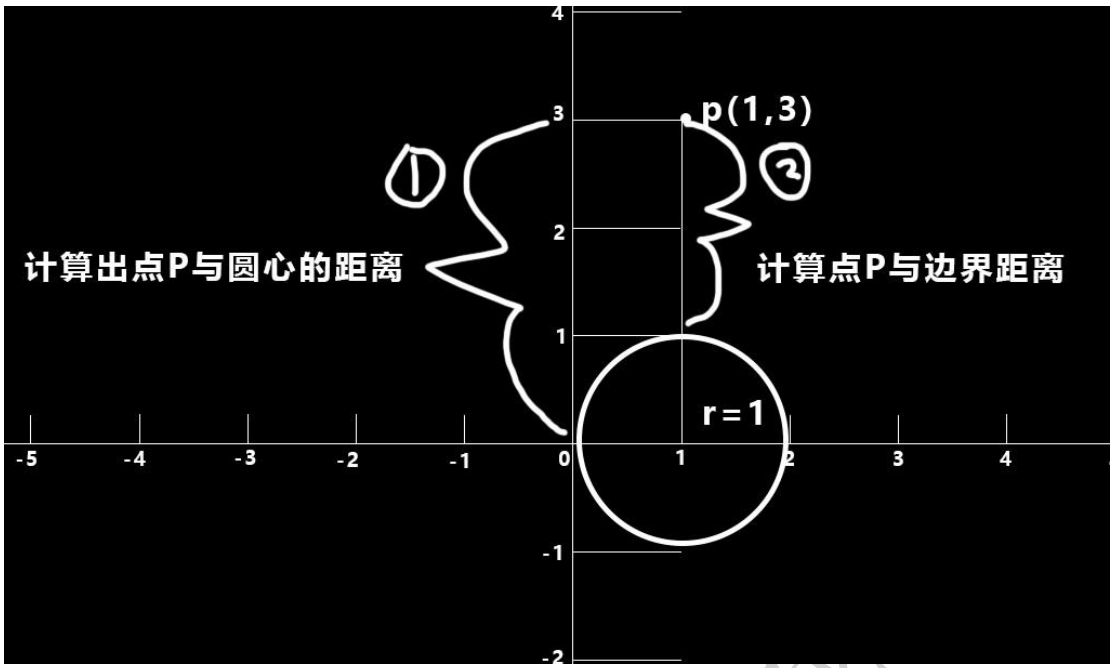


图313.　　圆心不在坐标轴原点

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。

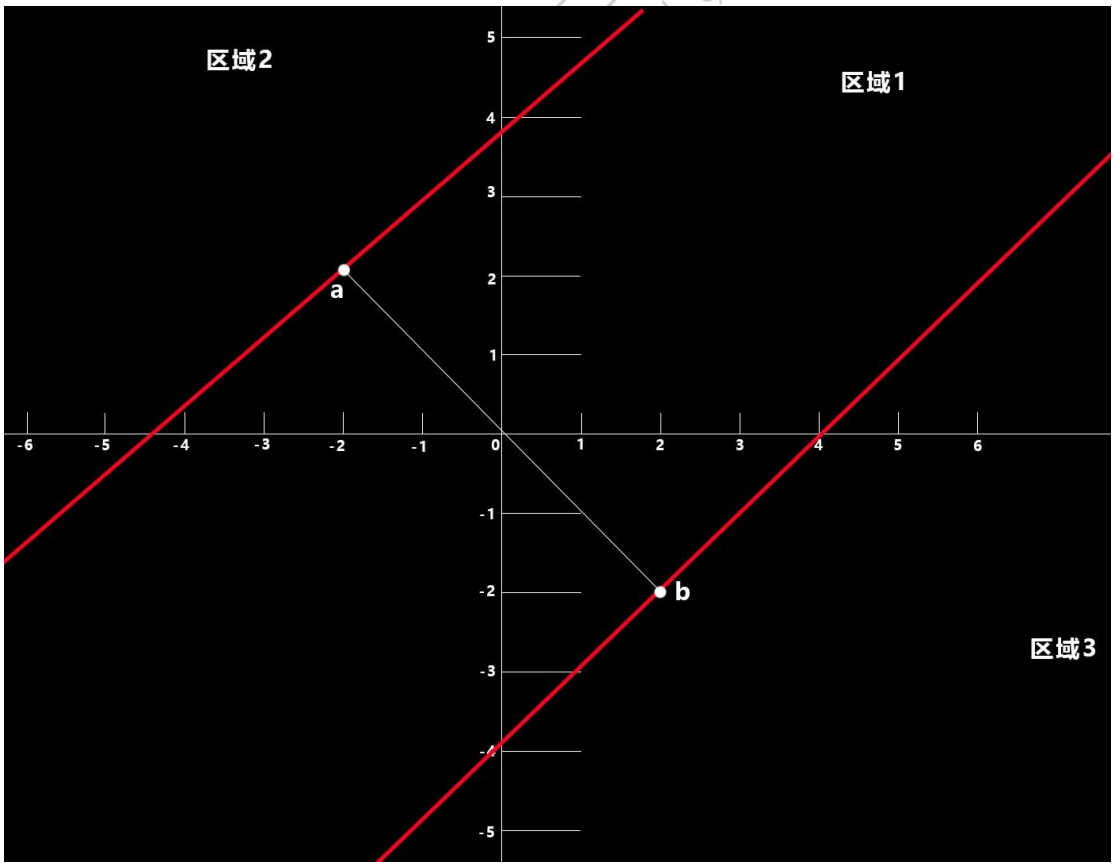


图314.　　SDF 线段区域划分

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

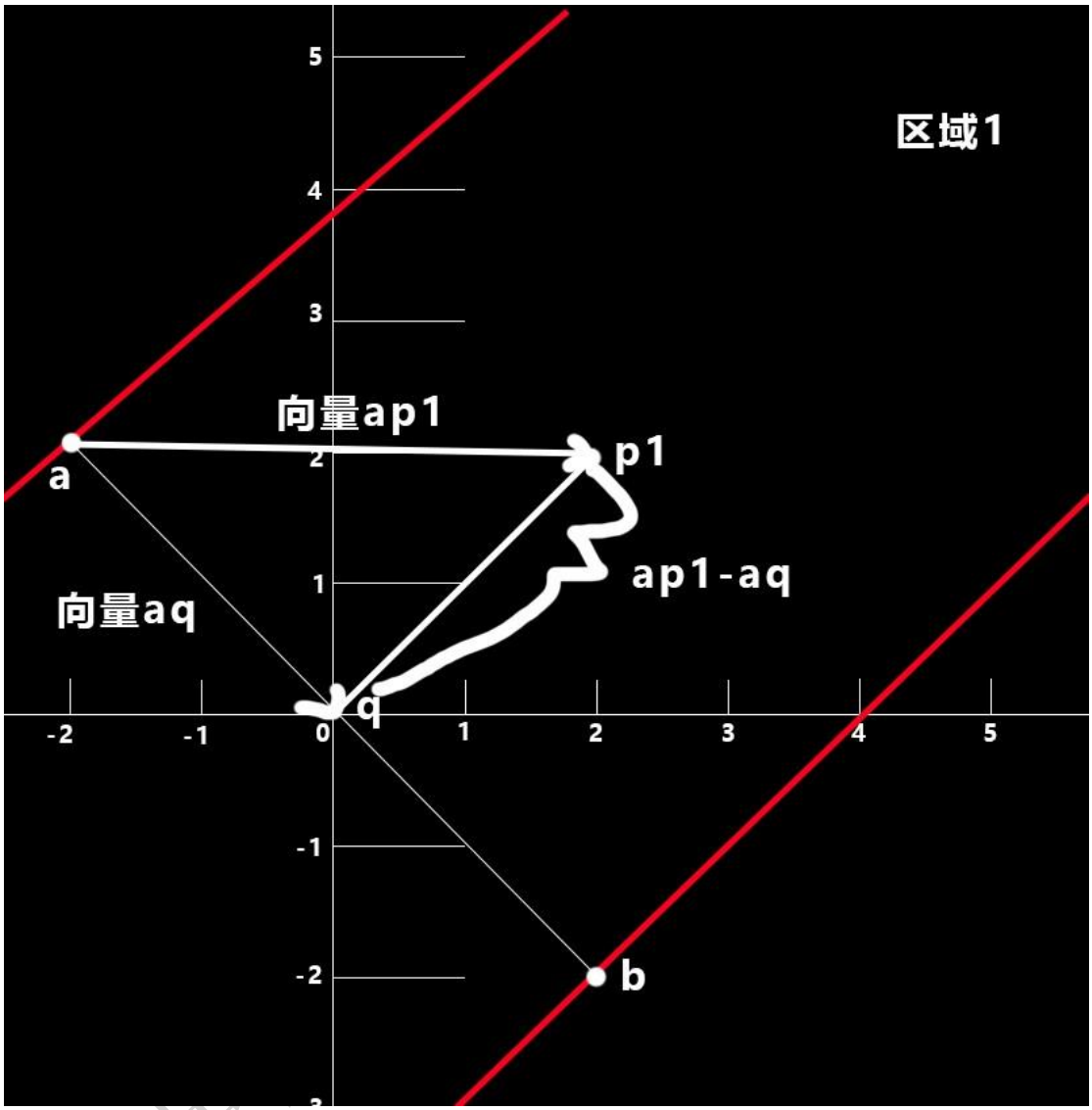


图315. 点位于区域一内

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，$\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r^{5}\overrightarrow{ab}$。（r[6]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。

[5] 这里的 r 是单词 Ratio 的缩写
[6] 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

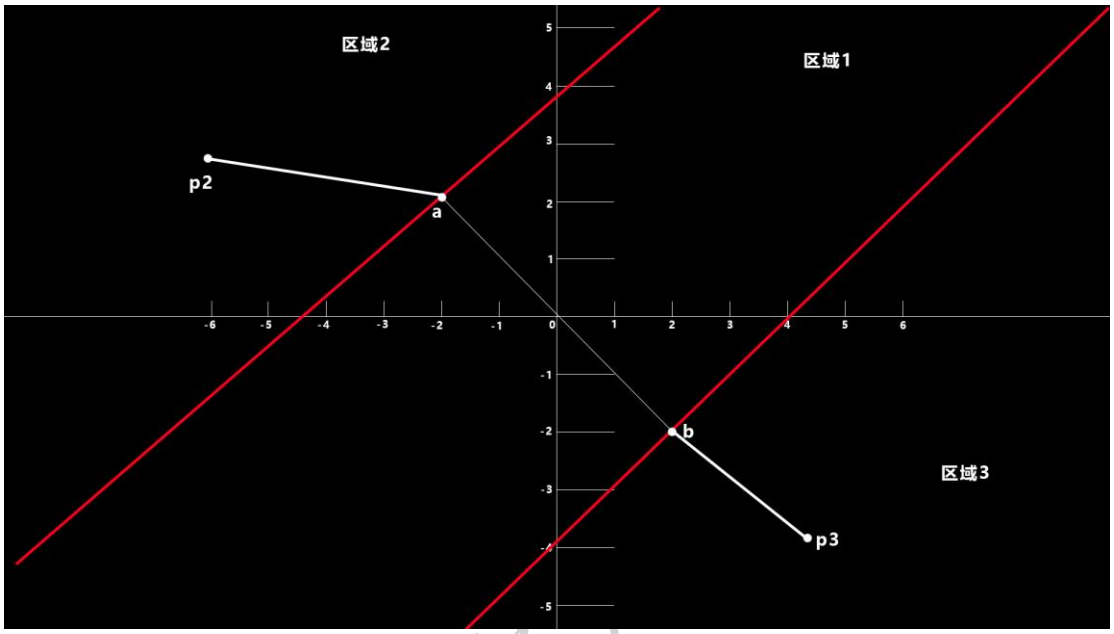


图316. 点位于区域二和三

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=$\|\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r$[7]$\overrightarrow{ab}\|$

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);
    return length(ap-h*ab);
```

[7] 这里的 r 是单词 Ratio 的缩写

}

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。

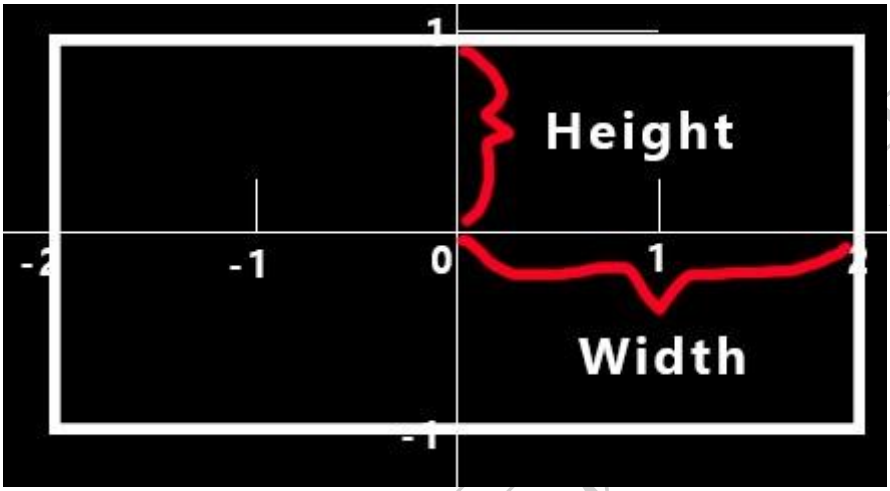


图317.　　矩形绘制原理

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。

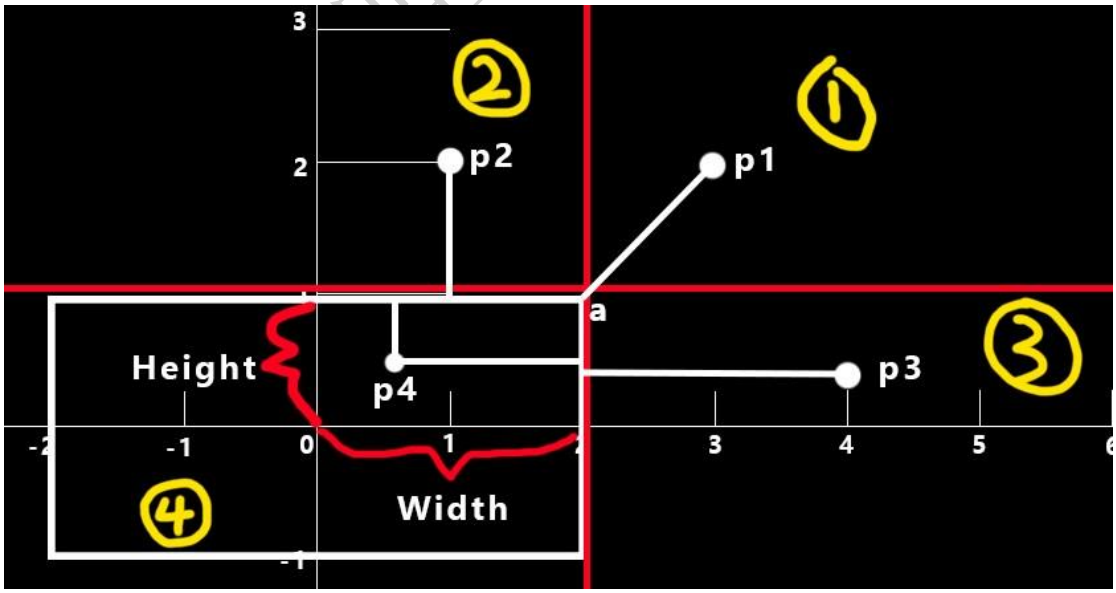


图318.　　目标点到矩形的情况划分

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。

哔哩哔哩伪元游戏人生

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。

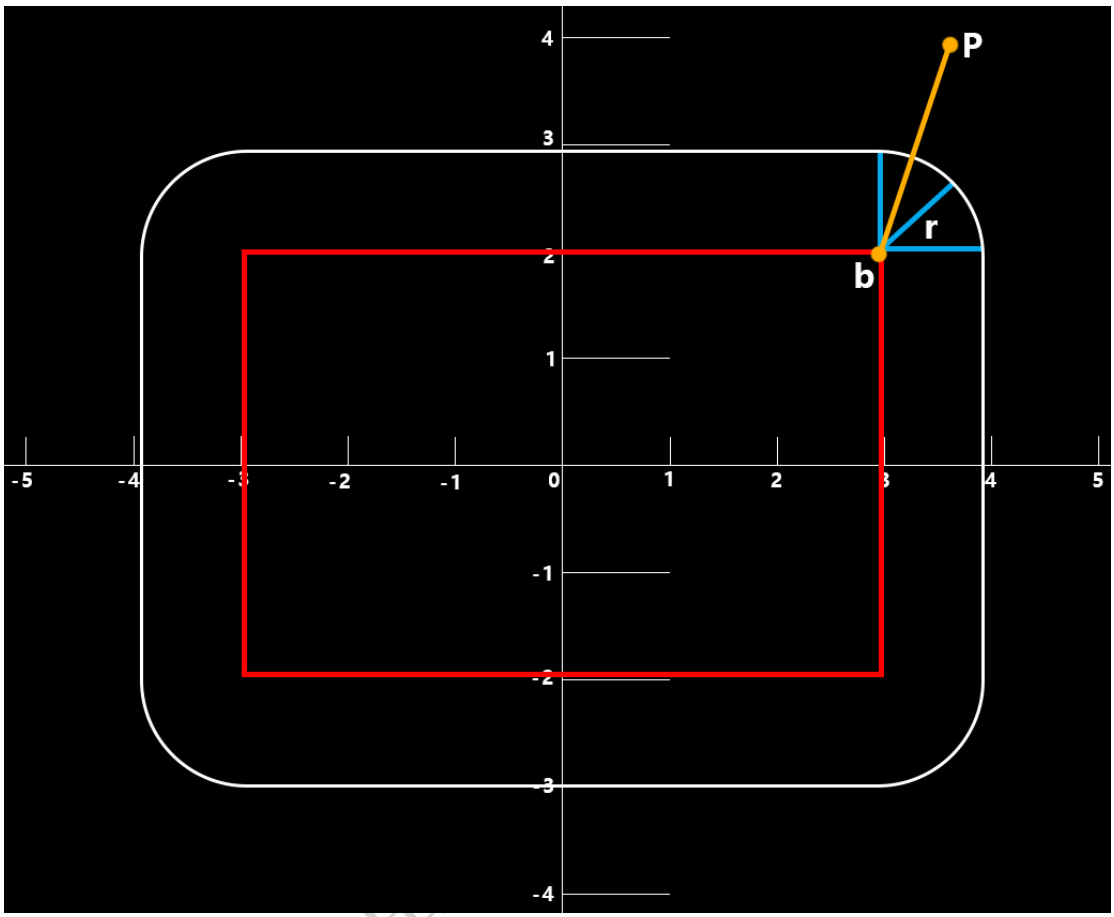


图319. 目标点 P 到圆角矩形

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。

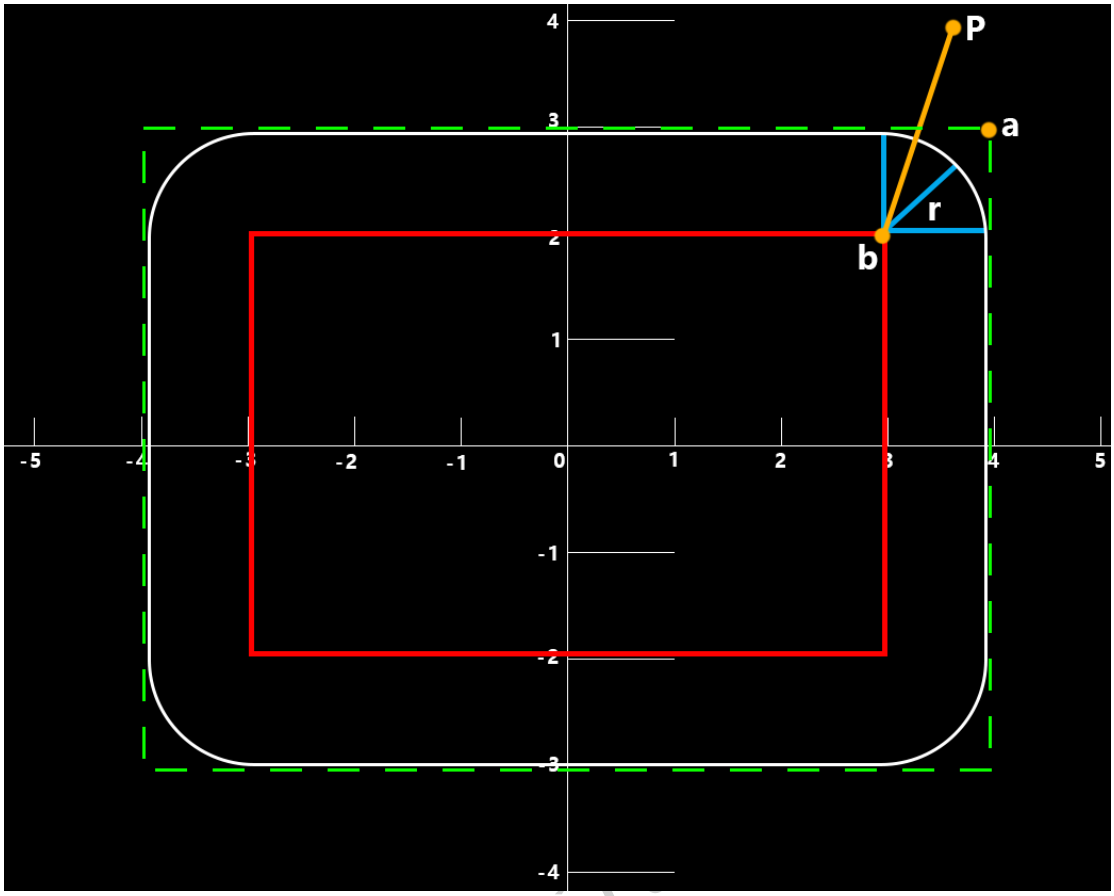


图320.　根据圆角半径去推断出初始矩形

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。

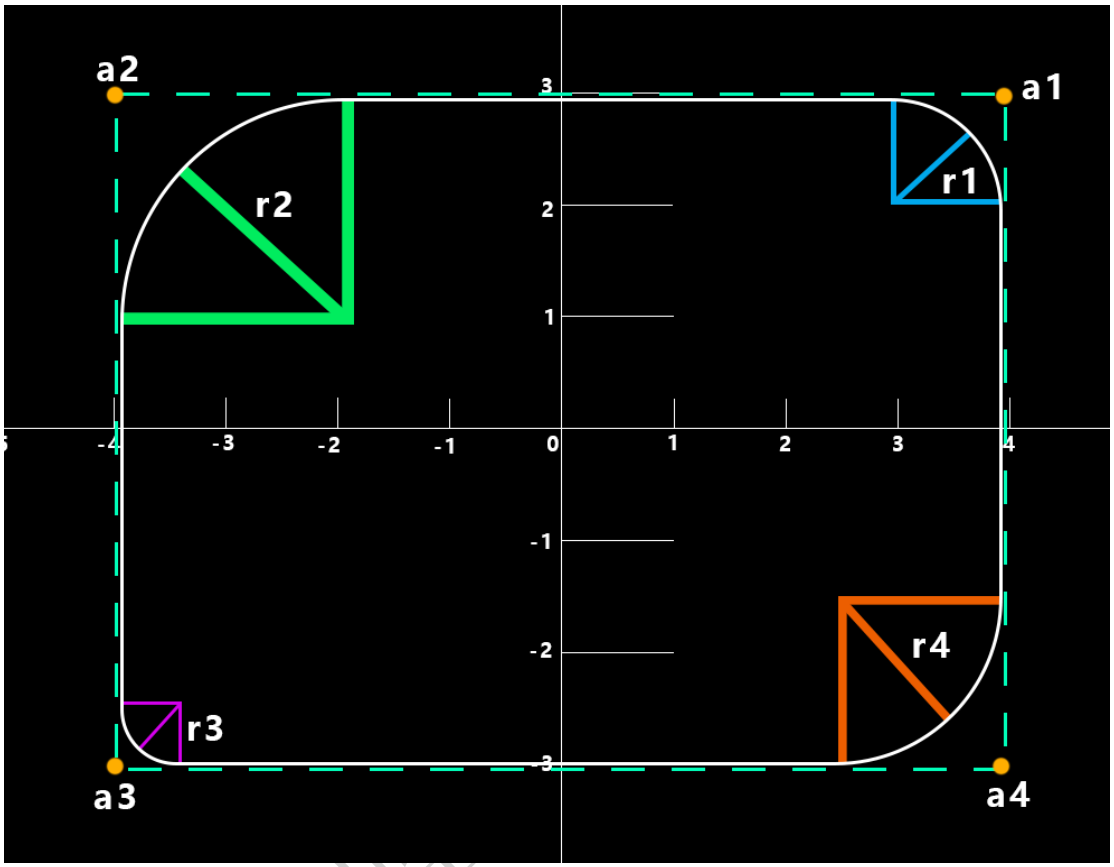


图321. 圆角半径不同

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;
}
```

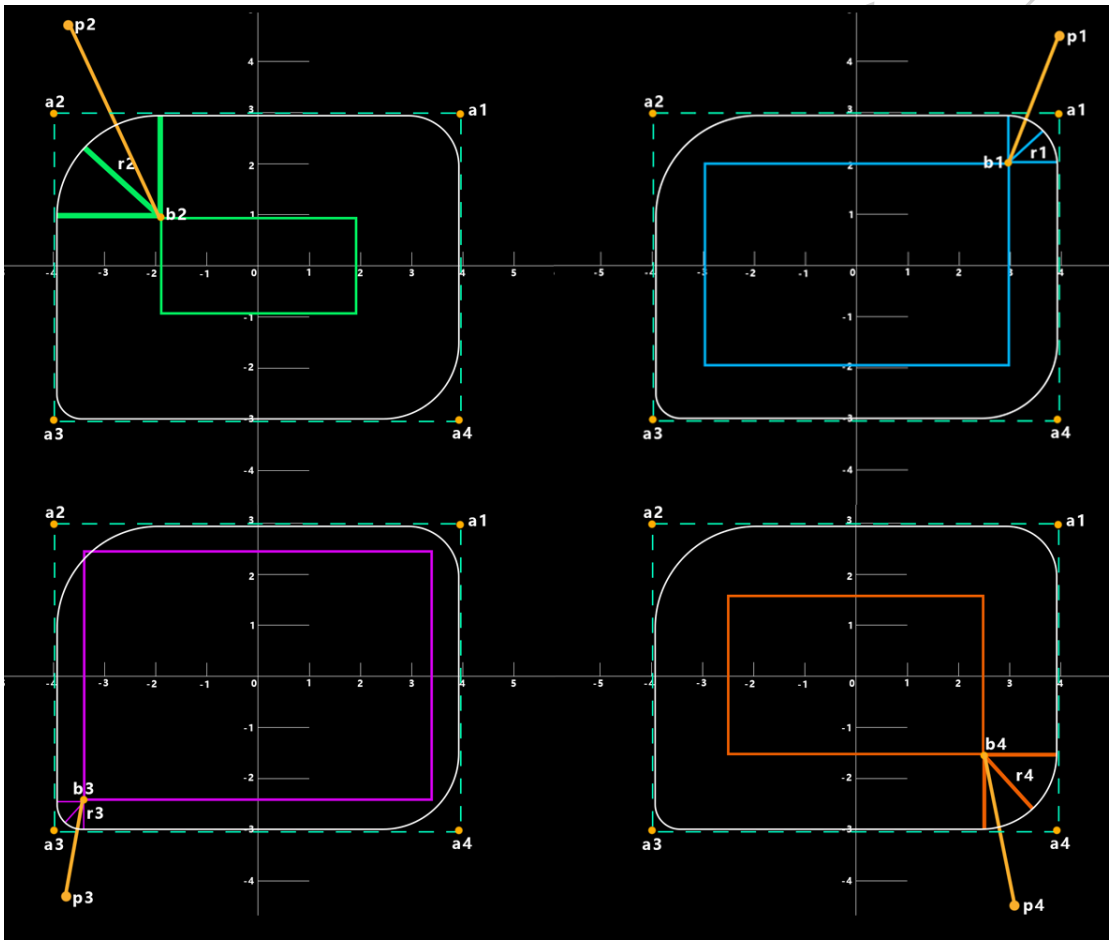


图322.　　圆角半径不同下的情况

5. SDF-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，只不过菱形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。

3） 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）。

$a(0, b.y)$ $d(b.x, 0)$ $c\left(\frac{b.x}{2}, \frac{b.y}{2}\right)$ $p(p.x, p.y)$ （这里的点 p 是取绝对值后的）

$$h = \frac{\overrightarrow{cp} \cdot \overrightarrow{ca}}{\overrightarrow{ca} \cdot \overrightarrow{ca}} = \frac{(p.x - \frac{1}{2}b.x, p.y - \frac{1}{2}b.y) \cdot (-\frac{1}{2}b.x, \frac{1}{2}b.y)}{}$$

4） 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;
}
```

十三、 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[8])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[9])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。

[8] adv. 内在地，固有地；本质的；
[9] 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

哔哩哔哩次元游戏人生

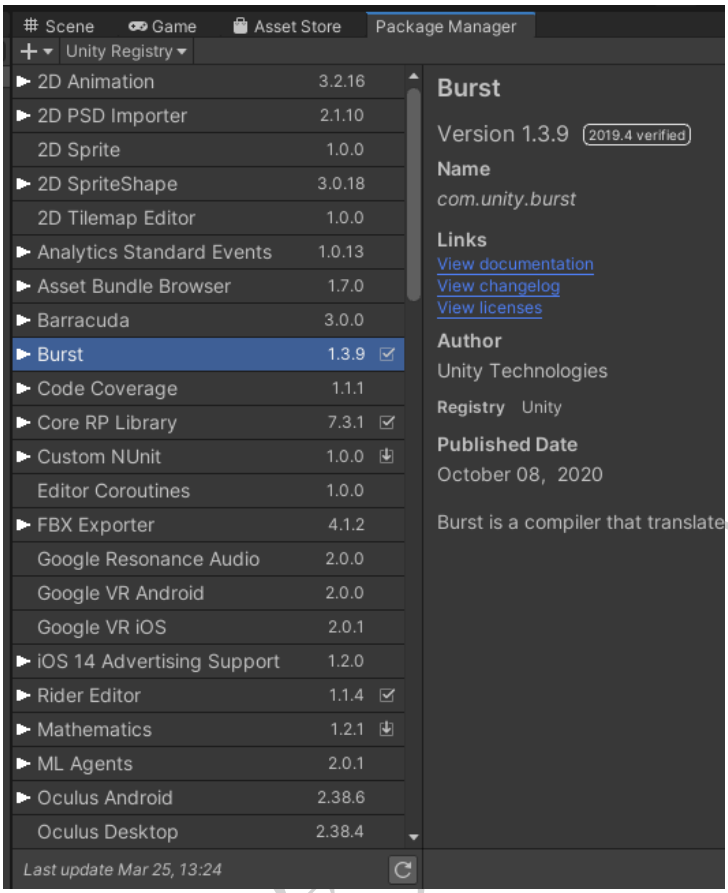


图1. Burst Package

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[10])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 {
```

[10] 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/

哔哩哔哩次元游戏人生

Visual Effect Graph

一、 篝火

哔哩哔哩次元游戏人生

PhotoShop

一、 去除水印

哔哩哔哩次元游戏人生