哔哩哔哩次元游戏人生

目　录

目　录 I
C、C#、C++ 14
一、　数据类型 14
二、　运算符优先级和结合性 14
三、　构造函数 15
四、　虚函数 15
1.　虚函数 15
2.　纯虚函数 15
3.　接口(Interface)和抽象类的区别 15
五、　C#中四种访问修饰符 16
1.　属性修饰符 16
2.　存取修饰符 16
3.　类修饰符 16
4.　成员修饰符 16
六、　ref 和 out 关键字 16
1.　ref 16
2.　out 17
3.　ref 和 out 的相同点 17
4.　ref 和 out 的不同点 17
七、　分部类和方法 17
1.　分部类 17
2.　分部方法 18
八、　面向对象的三大特点 18
1.　继承 18
2.　封装 18
3.　多态性 18
Unity 19
一、　PipelineAsset 19
1.　Quality 19
2.　Lighting 20
3.　Shadows 20
二、　Lighting 窗口 20
1.　Environment 20
2.　Other Settings 20
三、　Navigation 窗口 21
3.　Agents（代理类型） 21
四、　Nav Mesh Agent 21
五、　协程与委托 21
1.　Coroutine 协同程序 21
2.　Delegate 委托 21
六、　Event 事件 22
七、　TextMesh Pro 24

哔哩哔哩次元游戏人生

八、 富文本.....24
1. 标记格式.....24
2. 嵌套元素.....25
3. 标签参数.....25
九、 2D PSD 导入器.....25
1. 减少像素重叠.....26
2. 组合碰撞体.....27
3. 形状精灵烘焙.....28
4. 像素美化与锐化.....29
十、 粒子系统 Particle System.....30
1. 模拟阶段.....30
2. 渲染阶段.....31
十一、 脚本生命周期.....31
十二、 Animation.....33
1. 关于模型中的动画的位移变换问题.....33
十三、 后处理.....33
1. 为场景添加后处理效果的步骤.....34
2. 后处理效果应用顺序和效果组合.....36
3. 辉光-Bloom.....37
4. 色调映射-Tonemapping.....37
5. 白平衡-White Balance.....38
6. 胶片颗粒-Film Grain.....38
7. 景深-Depth of Field.....38
8. 帕尼尼投影-Panini Projection.....39
9. 镜头畸变-Lens Distortion.....39
10. 运动模糊-Motion Blur.....40
11. 色差-Chromatic Aberration.....40
12. 晕映-Vignette.....40
13. 颜色分级-Lift Gamma Gain.....41
14. 通道混合-Channel Mixer.....41
15. 颜色曲线- Color Curves.....42
16. 颜色调节-Color Adiustment.....42
17. 分离调色-Split Toning.....42
18. 阴影、中间调、高光-Shadow、Midtones、Highlights.....43
十四、 其他知识.....43
1. 图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别？.....43
2. 使用 Unity3d 实现 2d 游戏，有几种方式？.....43
3. 向量的点乘、叉乘以及归一化的意义？.....43
Unity Script.....44
一、 GameObject.....44
1. 属性（Properties）.....44
2. 构造函数（Constructors）.....44
3. 公共方法.....44
4. 静态方法.....45

哔哩哔哩次元游戏人生

二、 Transform 45
1. 公共方法 45
三、 Vector3 46
1. 描述 46
2. Static Properity-静态属性 46
3. Static Methods-静态方法 46
4. 向量归一化的区别 47
四、 KeyCode 编码 48
五、 数学函数库 50
1. 绝对值-Mathf.Abs 50
2. 反余弦-Mathf.Acos 50
3. 近似-Mathf.Approximately 50
4. 反正弦-Mathf.Asin 50
5. 反正切-Mathf.Atan2 50
6. 反正切-Mathf.Atan 50
7. 最小整数- Mathf.CeilToInt 50
8. 上限值-Mathf.Ceil 50
9. 范围限制 0-1-Mathf.Clamp01 50
10. 限制- Mathf.Clamp 50
11. 最近的二次方- Mathf.ClosestPowerOfTwo 51
12. 余弦- Mathf.Cos 51
13. 度转弧度- Mathf.Deg2Rad 51
14. 弧度转度- Mathf.Rad2Deg 51
15. 增量角- Mathf.DeltaAngle 51
16. 小正数- Mathf.Epsilon 51
17. 指数- Mathf.Exp 52
18. 最小整数- Mathf.FloorToInt 52
19. 向下取整- Mathf.Floor 52
20. 正无穷- Mathf.Infinity 52
21. 反差值- Mathf.InverseLerp 52
22. 是否为 2 的幂- Mathf.IsPowerOfTwo 52
23. 插值角度- Mathf.LerpAngle 52
24. 插值- Mathf.Lerp 52
25. 基数为 10 的对数- Mathf.Log10 52
26. 对数- Mathf.Log 52
27. 最大值- Mathf.Max 53
28. 最小值- Mathf.Min 53
29. 移动角- Mathf.MoveTowardsAngle 53
30. 移向- Mathf.MoveTowards 53
31. 负无穷- Mathf.NegativeInfinity 53
32. 下一个 2 的幂- Mathf.NextPowerOfTwo 53
33. 乒乓- Mathf.PingPong 53
34. 圆周率- Mathf.PI 53
35. 次方- Mathf.Pow 53

哔哩哔哩沃尼游戏人生

36. 重复- Mathf.Repeat.....53
37. 四舍五入到整数- Mathf.RoundToInt.....54
38. 四舍五入- Mathf.Round.....54
39. 有向- Mathf.Sign.....54
40. 正弦- Mathf.Sin.....54
41. 平方根- Mathf.Sqrt.....54
42. 正切- Mathf.Tan.....54
43. 平滑阻尼角度- Mathf.SmoothDampAngle.....54
44. 平滑阻尼- Mathf.SmoothDamp.....54
45. 平滑插值- Mathf.SmoothStep.....54
六、 UGUI 事件接口.....54
1. IPointEnterHandler.....54
2. IPointExitHandler.....55
3. IPointDownHandler.....55
4. IPointUpHandler.....55
5. IPointClickHandler.....55
6. IDragHandler.....55
7. IInitializePotentialDragHandler.....55
8. IBeginDragHandler.....55
9. IEndDragHanlder.....55
10. IDropHandler.....55
Unity 案例.....56
一、 2D 角色移动.....56
二、 3D 角色移动.....56
三、 3D 相机控制.....56
四、 输入管理-InputManager.....56
五、 场景异步加载与过渡.....57
六、 自定义 URP 后处理.....57
七、 通用对象池.....57
1. 概述.....57
2. 理解.....57
3. 优点.....58
4. 缺点.....58
5. 简单对象池.....58
6. 对象池管理.....63
技巧.....68
一、 修改 Unity 缓存位置.....68
1. 修改 Unity GICache.....68
2. 修改 Unity 的 Cache 缓存位置.....68
3. 修改 Unity Asset Store 的默认下载路径.....71
二、 优化.....71
1. 全平台优化方案.....71
2. 脚本性能优化.....73
3. 物理性能优化.....73

哔哩哔哩次元游戏人生

4. Shader 优化方案 73
三、 快捷键 77
1. Unity 77
2. Visual Studio 77
四、 脚本栏的扩展 77
1. 序列化[Serializable] 77
2. 强制序列化[SerializeField] 77
3. 属性标题[Header] 78
4. 属性提示[Tooltip] 78
5. 空行属性[Space] 78
6. 范围值属性[Range()] 78
7. 隐藏属性[HideInInSpector] 78
8. 输入域[TextArea] 78
9. [RequireComponent(typeof(xxx))] 78
10. [CreateAssetMenu(fileName = "xx", menuName = "xx", order = x)] 78
11. 永远执行-[ExecuteAlways] 78
12. 禁止多组件-[DisallowMultipleComponent] 78
13. 添加组件菜单-[AddComponentMenu] 78
14. 在编辑模式下执行-[ExecuteInEditMode] 78
五、 VisualStudio 78
六、 修改 unity 模板 79
七、 资源网站 79
八、 切片图集(Slice Atlas) 79
1. 下载 79
2. 切片类型 81
九、 未运行游戏的状态监视 ShaderGraph 82
十、 组件 Preset 82
十一、 调整运行窗口颜色 83
十二、 修改鼠标样式和启动图像 84
1. 设置鼠标样式 84
2. 设置启动图像 85
Unity Editor 88
一、 Editor 文件夹 88
二、 Editor Default Resources 88
三、 Gizmos 文件夹 88
Unity Editor 实践 89
Unity 工具 91
一、 Terrain Tool 91
3. 快捷键 91
二、 Cinemachine 91
三、 Recorder 91
4. 下载 91
5. 打开 Recorder 界面 92
6. Recording Mode(记录模式) 92

哔哩哔哩次元游戏人生

7. Frame Rate 帧率属性 92
8. Capture-捕捉设置 94
四、 Post Processing 95
1. 开始后处理 95
2. 泛光-Bloom 96
设计模式 97
一、 单例模式 97
1. Unity 泛型单例 97
二、 观察者模式 98
数据结构 100
一、 基本概率和术语 100
1. 数据结构三要素 100
数据结构习题 101
一、 数组 101
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。 101
算法- Algorithm 105
一、 洗牌算法 105
1. Fisher–Yates shuffle 105
2. Knuth-Durstenfeld Shuffle 105
物理 111
一、 牛顿第二定律 111
二、 抛体运动 111
线性代数 113
一、 向量 113
二、 行列式 113
1. 二阶行列式 113
2. 三阶行列式 113
3. n 阶行列式 113
4. 性质 113
5. 特殊的行列式 114
三、 全排列和对换 114
1. 全排列（排列） 114
2. 逆序 114
3. 对换 114
四、 点和向量 115
1. 具体来讲 115
2. 自由向量 115
五、 向量在轴上的投影 115
六、 数量积、向量积 116
1. 数量积（点积） 116
2. 向量积（叉乘） 116
七、 向量的内积（点积） 117
1. 内积性质 117

哔哩哔哩次元游戏人生

八、 向量的范数（长度）.....117
1. 性质.....117
概率论与数理统计.....119
初等数学.....120
高等数学.....121
一、 导数.....121
数据库.....122
Computer Graphics.....123
一、 贝塞尔曲线路径.....123
1. 贝塞尔曲线.....123
2. 一次 Bezier 曲线.....123
3. 二次 Bezier 曲线.....123
4. 三次 Bezier 曲线.....124
5. 高阶 Bezier 曲线.....124
二、 程序化网格生成.....125
1. 渲染物体.....125
2. UV.....125
3. 创建网格.....126
4. 生成法线.....127
5. 生成 UV 坐标.....127
Shader Graph.....128
一、 菲涅尔效果-Fresnel Effect Node.....128
1. 端口.....128
2. 生成的代码示例.....128
二、 乘节点-Multiply Node.....128
1. 端口.....128
2. 生成的代码示例.....128
三、 加法节点- Add Node.....129
1. 端口.....129
2. 生成的代码示例.....129
四、 除法-Divide Node.....129
1. 端口.....129
2. 生成的代码示例.....129
五、 步节点-Step Node.....129
1. 端口.....130
2. 生成的代码示例.....130
六、 平滑步节点-Smoothstep Node.....130
1. 端口.....130
2. 生成的代码示例.....130
七、 极坐标-Polar Coordinates Node.....131
1. 端口.....131
2. 生成的代码示例.....131
八、 重新映射-Remap Node.....131
1. 端口.....132

哔哩哔哩次元游戏人生

2. 生成的代码示例.....132
九、 限制-Clamp Node.....132
1. 端口.....132
2. 生成的代码示例.....132
十、 时间-Time Node.....132
1. 端口.....132
2. 生成的代码示例.....133
十一、 瓦片与偏移-Tiling And Offset Node.....133
1. 端口.....133
2. 生成的代码示例.....134
十二、 取反-Neagte Node.....134
1. 端口.....134
2. 生成的代码示例.....134
十三、 简单噪声-Simple Noise Node.....134
1. 端口.....134
2. 控件.....134
3. 生成的代码示例.....134
十四、 旋转-Rotate Node.....136
1. 端口.....136
2. 控件.....136
3. 生成的代码示例.....136
十五、 幂节点-Power Node.....137
1. 端口.....137
2. 生成的代码示例.....137
十六、 饱和节点-Saturate Node.....137
1. 端口.....137
2. 生成的代码示例.....137
十七、 线性插值-Lerp Node.....137
1. 端口.....138
2. 生成的代码示例.....138
十八、 反插值- Inverse Lerp Node.....138
1. 端口.....139
2. 生成的代码示例.....139
十九、 翻书-Flipbook Node.....139
1. 端口.....140
2. 控件.....140
3. 生成的代码示例.....141
二十、 向下取整- Floor Node.....141
1. 端口.....141
2. 生成的代码示例.....141
二十一、 取余- Modulo Node.....142
1. 端口 0.....142
2. 生成的代码示例.....142
二十二、 场景深度-Scene Depth Node.....142

哔哩哔哩次元游戏人生

1. 端口.....142
2. 深度采样模式.....142
3. 生成的代码示例.....142
二十三、 One Minus Node.....142
1. 端口.....142
2. 生成的代码示例.....143
二十四、 位置-Position Node.....143
1. 端口.....143
2. 控件.....143
二十五、 绝对值-Absolute Node.....143
1. 端口.....143
2. 生成的代码示例.....143
二十六、 变换-Transform Node.....143
1. 端口.....143
2. 控件.....144
3. World and Absolute World.....144
4. 生成的代码示例.....144
二十七、 标准化-Normalize Node.....146
1. 端口.....146
2. 生成的代码示例.....146
二十八、 长度-Length Node.....146
1. 端口.....146
2. 生成的代码示例.....146
二十九、 距离-Distance Node.....146
1. 端口.....146
2. 生成的代码示例.....147
三十、 点乘-Dot Product Node.....147
1. 端口.....147
2. 生成的代码示例.....147
三十一、 小数-Fraction Node.....147
1. 端口.....147
2. 生成的代码示例.....147
三十二、 随机范围-Random Range Node.....147
1. 端口.....147
2. 生成的代码示例.....147
三十三、 比较-Comparison Node.....148
1. 端口.....148
2. 控件.....148
3. 生成的代码示例.....148
三十四、 快捷键.....149
ShaderGraph 实际案例.....150
一、 2D 简易描边效果-减法版.....150
1. 创建 Shader Graph.....150
2. 创建描边轮廓.....151

哔哩哔哩次元游戏人生

3. 修改描边轮廓颜色.....151
4. 得到最终成像.....152
二、 2D 描边效果-加法版.....152
1. 创建 Shader Graph.....152
2. 放大原图像遮罩.....153
3. 获得描边.....154
4. 修改描边颜色.....155
5. 得到最终成像.....155
6. 可能出现的问题.....156
三、 2D UI 技能冷却缓动.....156
1. 创建 Shader Graph.....156
2. 创建切分效果.....157
3. 重新映射.....158
4. 扩展-旋转 UV.....159
5. 扩展-极坐标.....159
6. 在代码中控制.....160
四、 2D 图像 RGB 分离.....162
1. 创建 Shader Graph.....162
2. 添加可外部访问的参数.....163
3. 添加节点.....163
五、 2D 全息效果.....165
1. 创建 Shader Graph.....165
2. 添加可外部访问的参数.....166
3. 创建带全息效果的图像.....167
4. 让全息纹理流动起来.....167
5. 扩展-调节全息贴图显示范围.....167
六、 2D 溶解效果.....168
1. 创建 Shader Graph.....168
2. 添加可外部访问的参数.....169
3. 创建溶解效果.....170
4. 添加溶解边缘颜色.....170
七、 2D 流光效果简易版.....171
1. 创建 Shader Graph.....171
2. 添加可外部访问的参数.....172
3. 显示含有流光效果的图像.....172
4. 让流光纹理流动起来.....173
5. 反向运动.....174
6. 旋转流光.....174
7. 可能出现的问题.....174
八、 2D 流光效果进阶版.....175
1. 创建 Shader Graph.....175
九、 转场过渡效果-简易版.....176
1. 创建 Shader Graph.....176
2. 添加可外部访问的参数.....177

哔哩哔哩次元游戏人生

3. 实现硬性过渡.....178
4. 将硬性过渡更改为柔性过渡.....178
十、 2D 激光效果.....179
十一、 2D 帧动画.....179
1. 创建 Shader Graph.....179
2. 添加主纹理.....180
3. 分割主纹理.....181
4. 让图集动起来.....181
5. 设置图集的最大可显示帧.....182
6. 调整帧动画速度.....184
7. 添加额外参数.....184
8. 可能存在的问题.....184
十二、 后处理速度线.....185
1. 创建 Shader Graph.....185
2. 制作速度线.....186
3. 让速度线流动起来.....187
4. 调整速度线的 Scale 和 Tiling.....187
5. 制作蒙版.....188
6. 最终输出.....188
7. 将节点转换为参数.....189
8. 后处理运用.....189
十三、 2D 像素效果.....190
1. 创建 Shader Graph.....190
2. 原理解释.....191
3. 修改图像对应的 UV 坐标.....191
4. 使用 Slider 控制像素化效果.....192
5. 设置是否应用像素化.....194
6. 自定义像素化行列（拓展）.....194
十四、 2D 植被飘动.....195
1. 创建 Shader Graph.....195
2. 原理.....196
3. 将主纹理输出到片元着色器.....196
4. 让顶点位置发生偏移.....196
5. 使用噪声调整偏移强度.....198
6. 对顶点应用噪声偏移.....198
7. 修改噪声视觉效果.....199
8. 创建影响蒙版.....199
十五、 2D 简单故障效果.....200
1. 创建 Shader Graph.....200
2. 创建色差效果.....201
3. 创建故障纹理.....203
4. 调节故障纹理密度.....205
5. 设置故障偏移方向.....205
6. 闪烁效果.....206

哔哩哔哩次元游戏人生

7. 完成最终效果.....206
十六、 2DSDF 圆.....207
1. 圆心在原点.....207
2. 圆心不在原点上.....208
十七、 2DSDF 线段.....209
1. 位于区域一.....210
2. 位于区域二和三.....211
3. 计算端点向量和到目标点的向量.....212
4. 计算 r.....212
5. 计算aq向量.....213
6. 计算点距离线段的长度.....213
十八、 矩形-Rectangle.....214
1. 矩形中心在原点.....214
2. 矩形中心不在原点.....216
3. 创建所需参数.....216
4. 计算向量ap1.....216
5. 计算目标点位于区域一、二、三时的模.....216
6. 计算区域四点的最短距离.....217
7. 排除目标点不位于区域四的情况.....217
8. 计算最终结果.....218
9. 缺点.....219
十九、 3D 描边效果.....219
二十、 3D 积雪效果.....219
二十一、 3D 广告牌效果.....219
Shader Lab.....220
一、 Shader 概念.....220
1. 什么是 Shader.....220
2. Shader 和 Material（材质）的关系与区别.....220
3. Unity 中的 Shader.....220
二、 简单概念.....221
1. 顶点-Vertices.....221
2. 法线-Normals.....221
3. 切线-Tangents.....222
4. UV 坐标-UV coordinates.....222
三、 渲染流水线.....223
1. 什么是渲染流水线.....223
2. 应用阶段(Application Stage).....224
3. 几何阶段(Geometry Stage).....224
4. 光栅化阶段(Rasterizer Stage).....224
5. 像素处理阶段(Pixel processing stage).....225
四、 CPU 和 GPU 之间的通信.....225
1. 把数据加载到显存中.....225
2. 设置渲染状态.....225
3. 调用 Draw Call.....226

哔哩哔哩次元游戏人生

五、 GPU 流水线.....226
1. 顶点着色器.....227
2. 裁剪.....228
3. 屏幕映射.....229
4. 三角形设置.....230
5. 三角形遍历.....230
6. 片元着色器.....231
7. 逐片元操作.....232
六、 什么是 Draw Call.....237
1. CPU 和 GPU 是如何实现并行工作的？.....237
2. 为什么 Draw Call 多了会影响帧率？.....238
3. 如何减少 Draw Call？.....238
七、 语法基础.....239
1. ShaderLab 的组织结构.....239
2. 详细解释.....253
八、 顶点和片元着色器基础.....268
1. CGPROGRAM/ENDCG.....268
2. 数据类型.....269
3. 编译指令.....271
九、 有向距离函数-SDF-Sign Distance Function.....278
1. 圆-Circle.....278
2. 线段-LineSegment.....280
3. 矩形-Rectangle.....283
4. 圆角矩形- Rounded Rectangle.....284
Visual Effect Graph.....285
一、 篝火.....285

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

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。

Unity

一、 PipelineAsset

1. Quality

2) HDR

HDR 是 High Dynamic Range 的缩写，即高动态范围。动态范围指的是最高和最低亮度值的比值。在真实世界中，一个场景中最亮和最暗区域的范围可以非常大，例如，太阳发出的光可能要比场景中某个影子上的点的亮度高出数万倍，这些范围远远超过图像或显示器能够显示的范围。

通常在显示设备使用的颜色缓冲中每个通道的精度为 8 位，意味着我们只能使用这 256 种（即 2^8）不同的颜色来表示真实世界中的所有亮度，所以在虚拟世界中显示颜色会损失一定的精度。早期的拍摄设备利用人眼的特点，使用了伽马曲线来对捕捉到的图像进行编码，尽可能充分地利用这些有限的存储空间。

HDR 的出现给我们带来了新的希望，HDR 使用远远高于 8 位的精度（如 32 位）来记录亮度信息，使得我们可以表示超过 0-1 内的亮度值，从而可以更加精确地反映真实的光照环境。虽然在最后我们依旧需要把信息转换到显示设备使用的 LDR 内，但是中间的计算却可以让我们得到更加真实可信的效果。

NVIDIA 曾总结过使用 HDR 进行渲染的动机：让亮的物体可以真的非常亮，暗的物体可以真的非常暗，同时又可以看到两者之间的细节。

使用 HDR 来存储的图像被称为高动态范围图像（HDRI），例如使用一张 HDRI 图像来作为场景的天空盒（Skybox），可以更加真实地反映物体周围的环境，从而得到更加真实的反射效果。

HDR 对与光照的叠加也有非常重要的作用。如果我们的场景中有很多光源获胜光源强度很大，那么一个物体经过多次光照渲染叠加后最终得到的光照亮度很可能会超过 1.如果没有使用 HDR，超过 1 的部分会被截取掉，使得场景丢失了很多亮部区域细节。一旦开启了 HDR，我们就可以保留这些超过范围的光照结果，尽管最后我们仍然，需要把它们转换到 LDR 进行显示，但我们可以使用色调映射（tonemapping）技术来控制这个转换的过程，从而允许我们最大限度地保留需要的亮度细节。

HDR 的使用可以允许我们在屏幕后处理中拥有更多控制权。例如，Bloom 效果需要检测屏幕中亮度大于某个阈值的像素，把它们提取出来后进行模糊，再叠加到原图像中。如果我们不使用 HDR，就只能使用小于 1 的阈值来提取需要的像素，但很多时候我们实际上是需要提取那些非常亮的区域，比如车窗上的强烈的反光。如果不使用 HDR，这些值实际上可能和街道上一些颜色偏白的区域一样，使应保持阴暗的地方也出现泛光的效果。而使用 HDR，这些问题都可以得到解决，我们只需使用超过 1 的阈值来提取那些非常亮的区域即可。

使用 HDR 可以让我们不会丢失高亮度区域的颜色值，提供了更真实的光照效果，并为一些屏幕后处理提供更多的控制能力。但是 HDR 也有缺点，由于使用了浮点缓冲来存储高精度图像，不仅需要更大的显存空间，渲染速度也会变慢，除此之外，一些硬件并不支持 HDR。而且一旦使用 HDR，我们就无法使用硬件的抗锯齿功能。

在 Unity 中如果我们同时打开了硬件的抗锯齿（在 Edit -> Project Settings -> Qualitty ->

Anti Aliasing 中打开）和摄像机的 HDR，Unity 会向我们发出警告，提醒我们因为开启了抗锯齿而无法再使用 HDR 缓冲。(我们可以基于屏幕后处理的抗锯齿操作来弥补这一点)

3） 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。

降维次元游戏人生

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

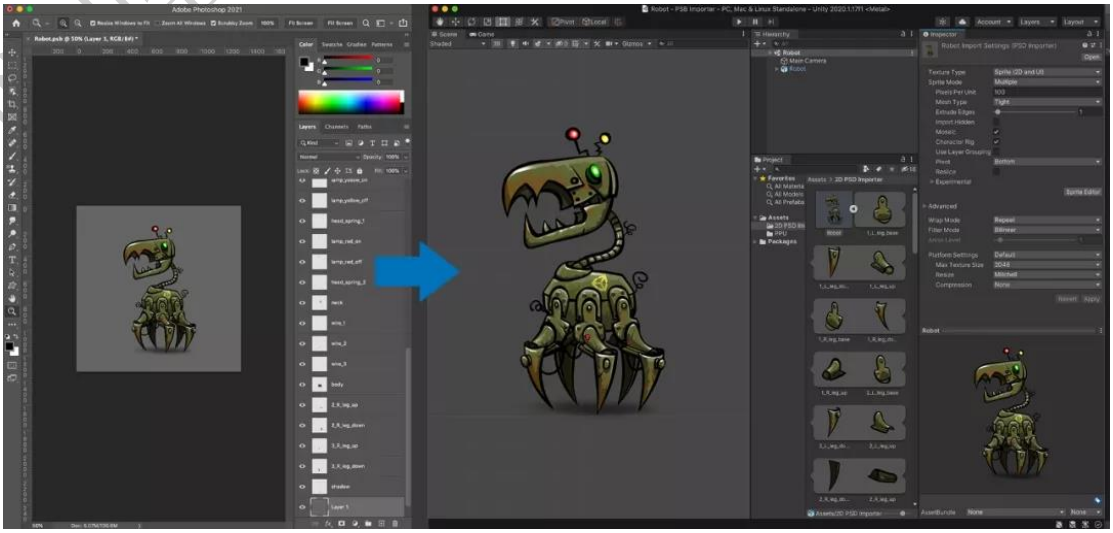


图7. 图 8.1 PSD 导入示例

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

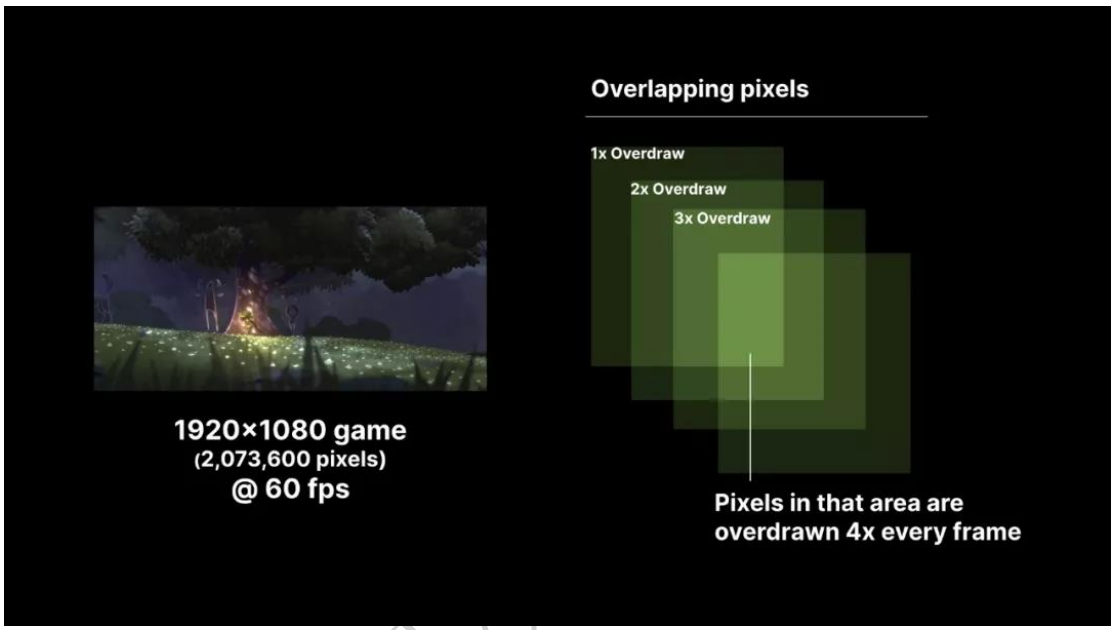


图8. 重叠像素优化示例

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。

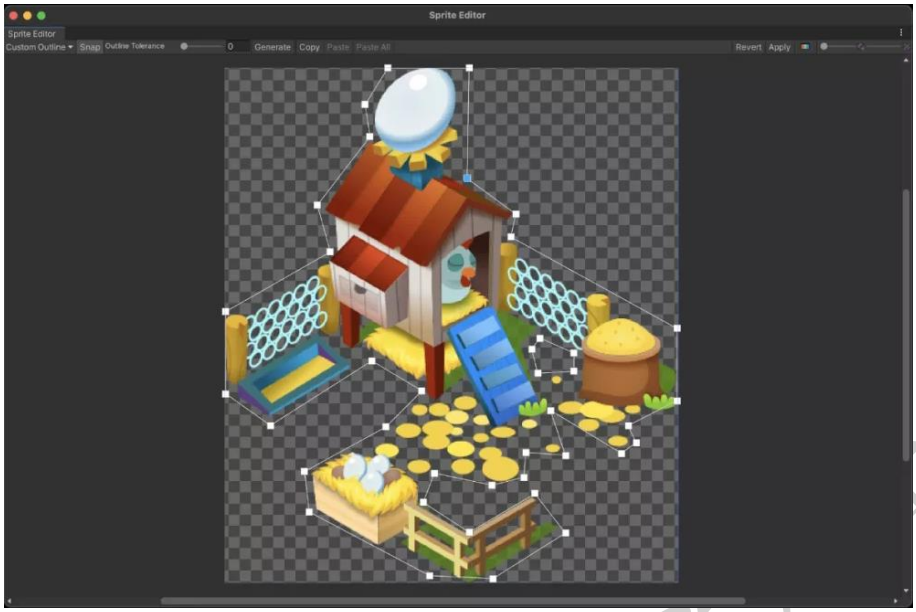


图9. 图 8.3 合并重叠的静态精灵.

2. 组合碰撞体

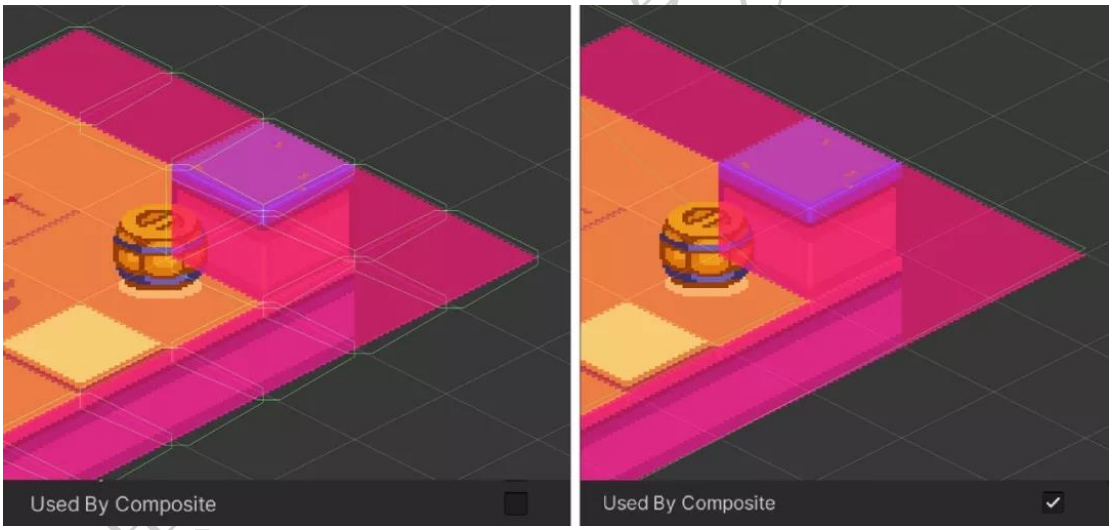


图10. 组合碰撞体的关闭和开启

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

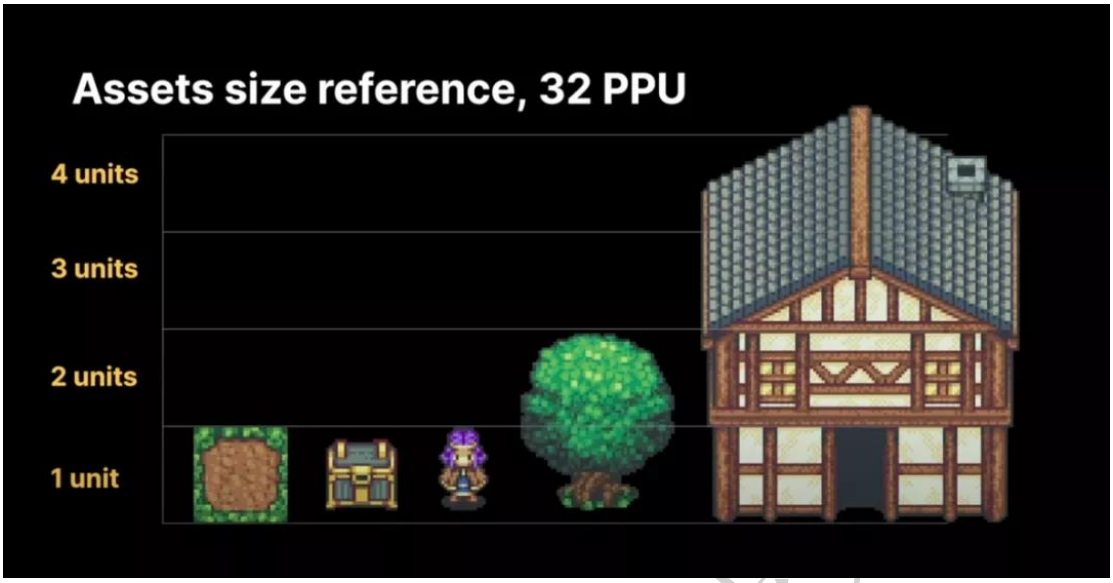


图12. 资源尺寸参考——32 像素每单位（PPU）

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：

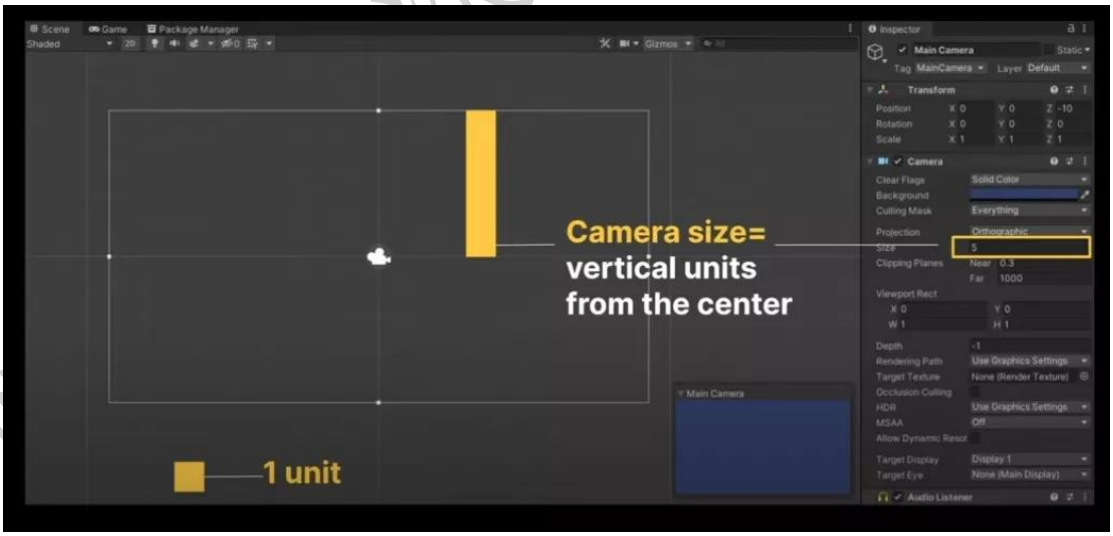


图13. 图 8.7 图像尺寸描述示例

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。

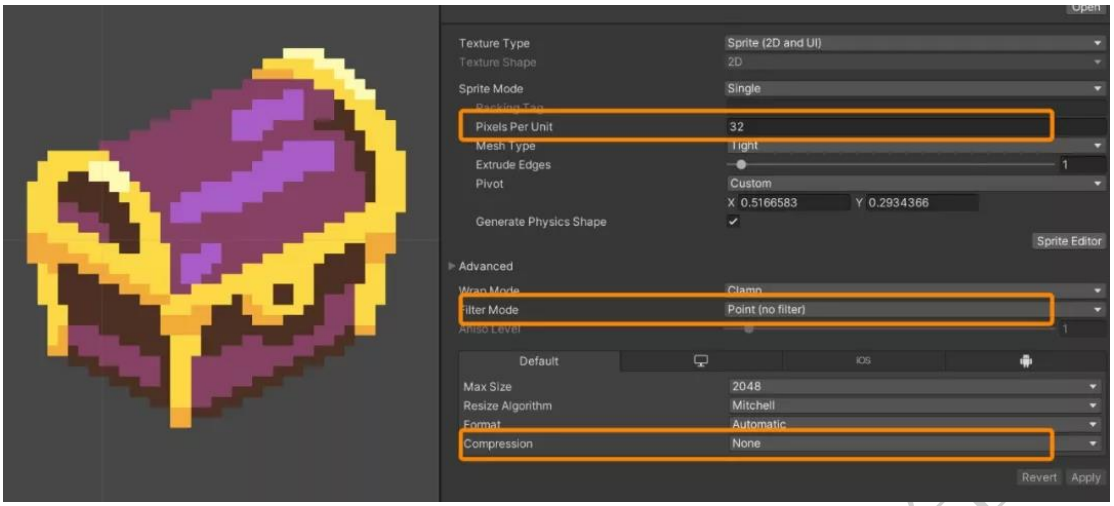


图14.　　图 8.8 精灵图设置

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。

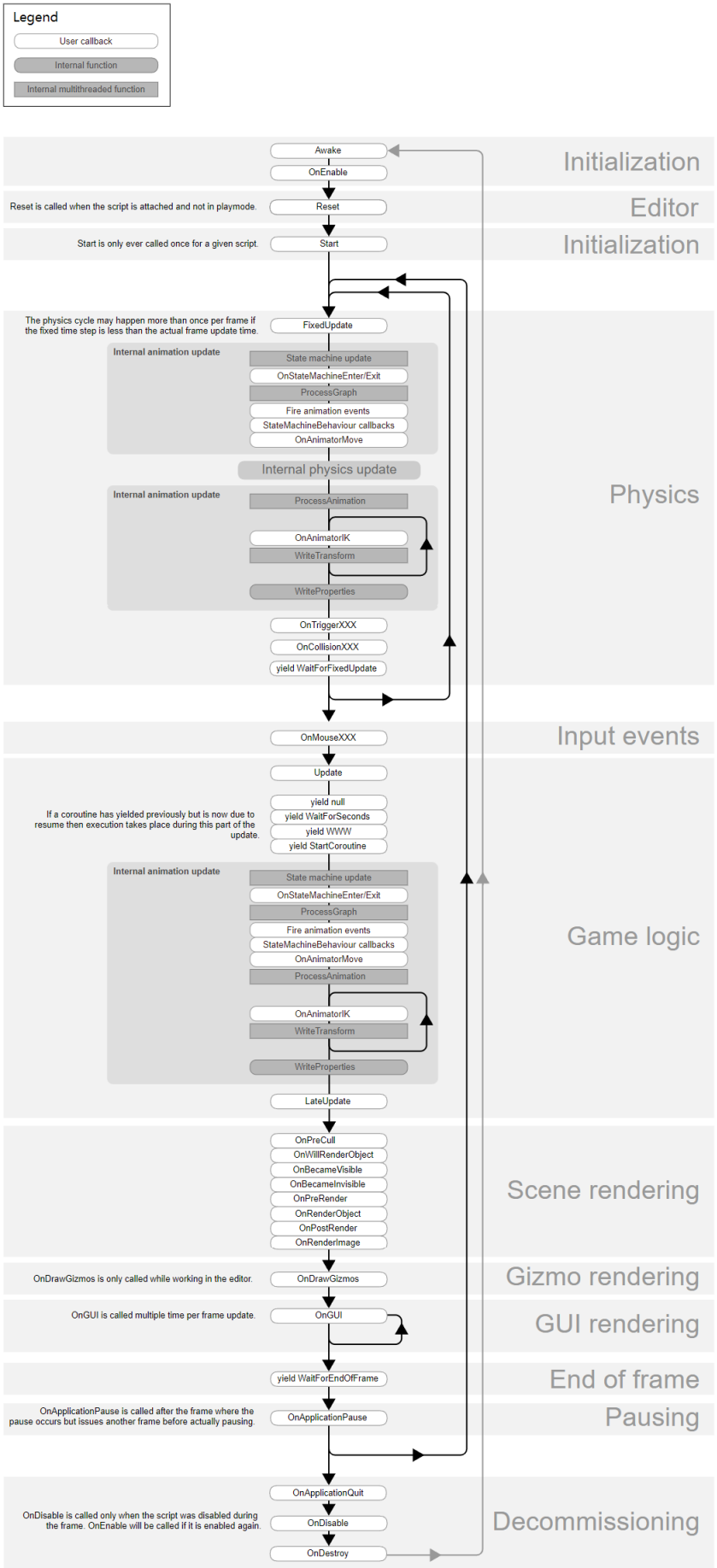
2. 渲染阶段

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、 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。

哔哩哔哩 无游戏人生

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。

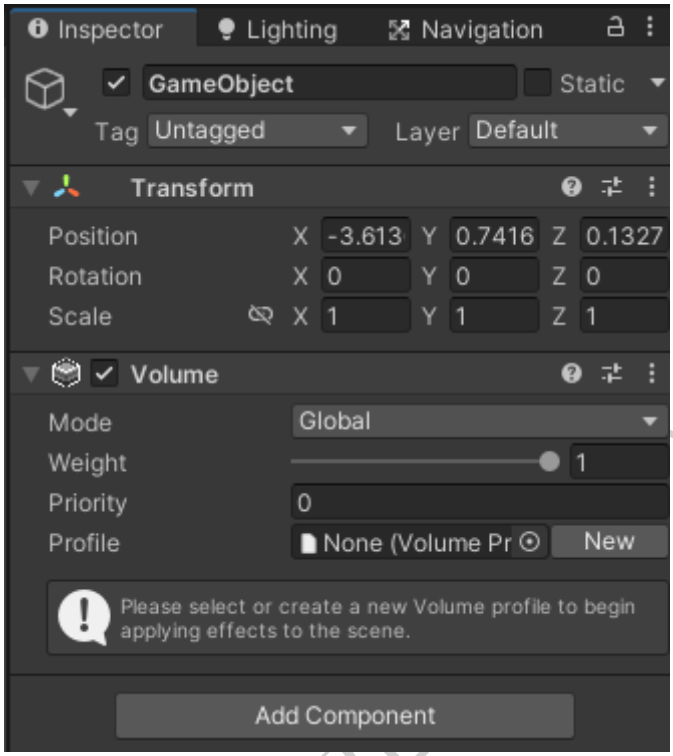


图15. 图 13.1 为对象添加 Volume 组件

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。

哔哩哔哩狄龙生

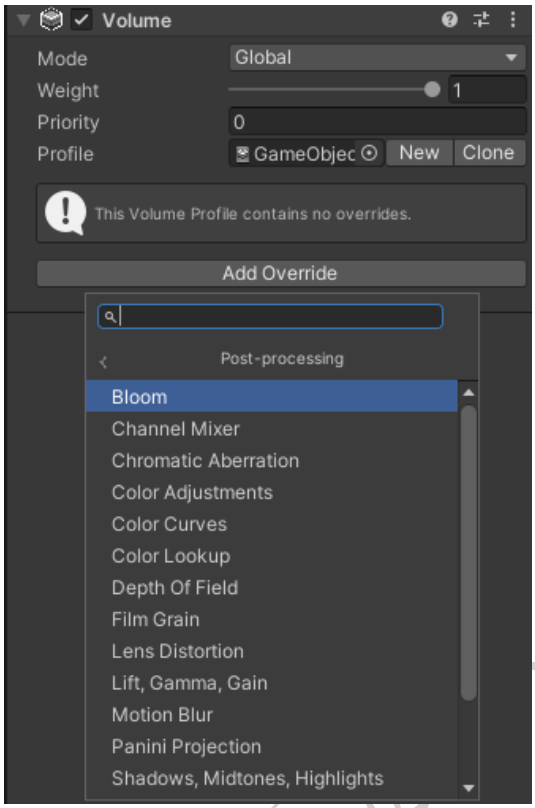


图16.　　图 13.2 使用 Override 添加后处理效果

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 次元人生

2. 后处理效果应用顺序和效果组合

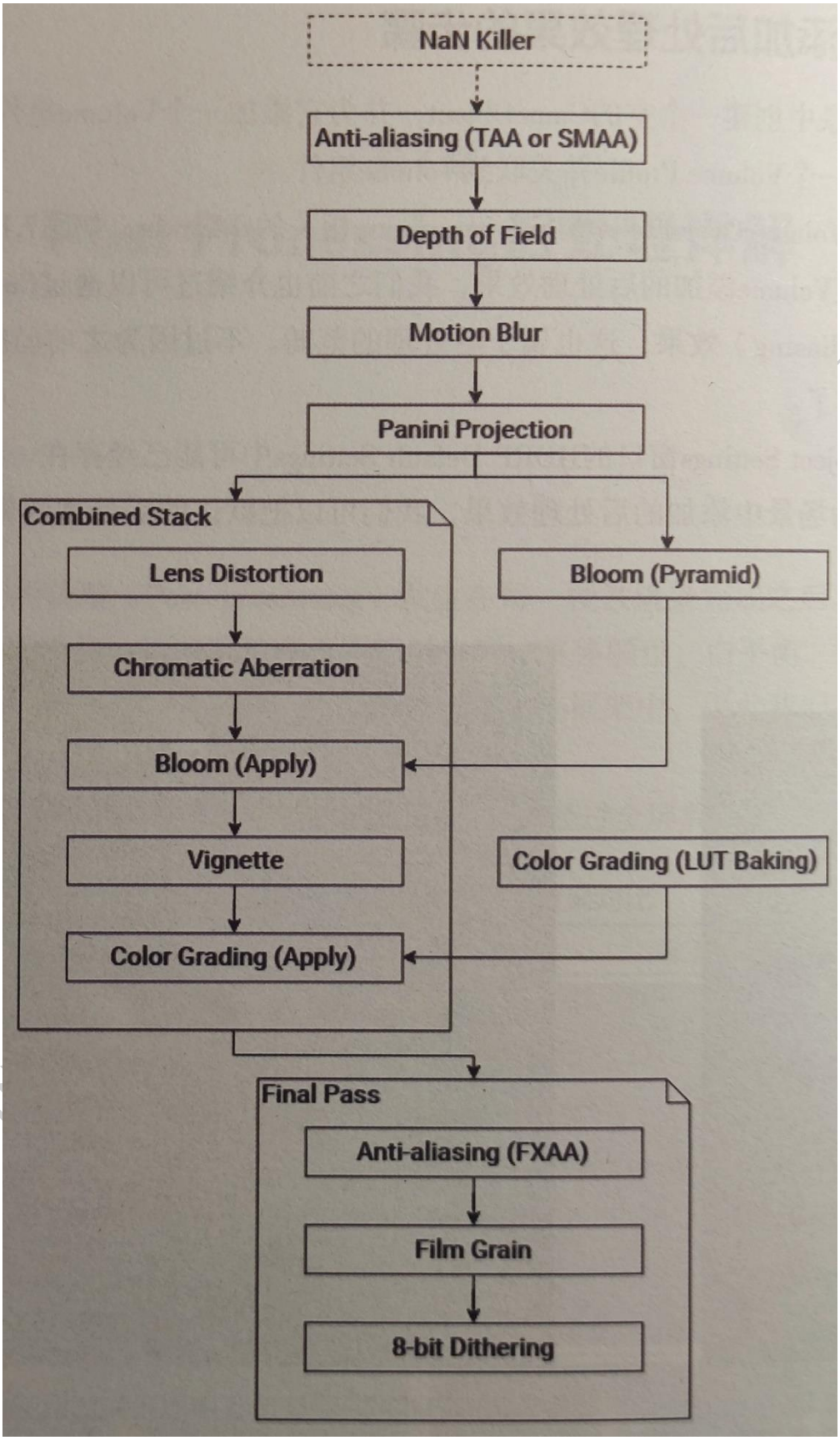


图17.　　图 13.3 HDRP 渲染管线应用后处理效果的顺序

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。

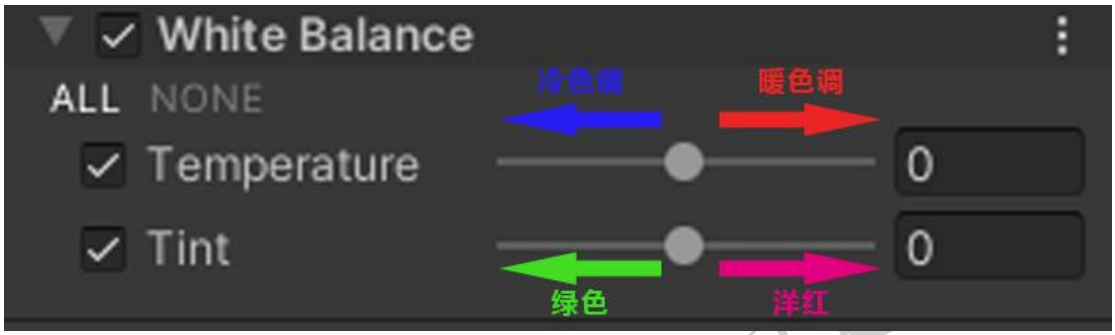


图18. 白平衡

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。

哔哩哔哩 次元游戏人生

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。

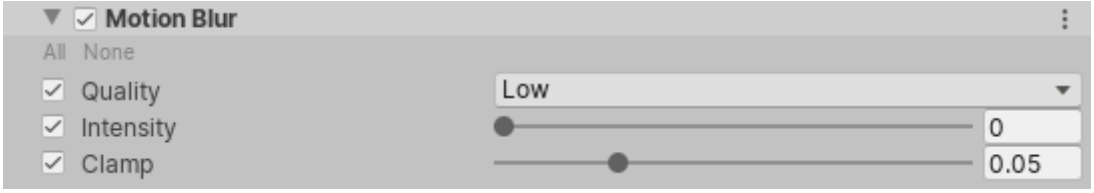


图19. Motion Blur

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

哔哩哔哩次元游龙生

	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。

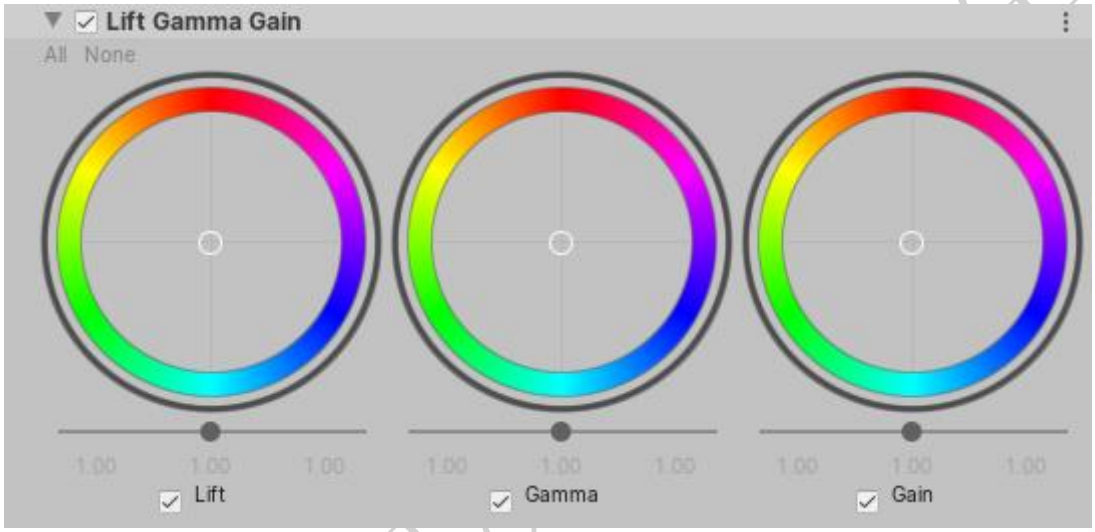


图20. LiftGammaGain

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。

十四、 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。

唯理学理次元游戏人生

Unity Script

一、 GameObject

1. 属性（Properties）

1） activeInHierarchy

定义场景中的游戏对象是否处于可见状态。

2） isStatic

可以获取和设置游戏对象的静态编辑标志。

3） layer

游戏对象所在的层级。

4） Scene

游戏对象所属的场景。

5） sceneCullingMask

用于确定在哪个场景中渲染游戏对象。

6） Tag

这个游戏对象的标签。

7） transform

游戏对象的 transform component。

2. 构造函数（Constructors）

1） GameObject

创建一个名为 name 的新的游戏对象。

3. 公共方法

1） AddComponent

将一个新的组件类添加到游戏对象。

2） BroadcastMessage

在此游戏对象或其任何子项中的每个 MonoBehaviour 上调用名为 methodName 的方法。

3） CompareTag

比较游戏对象的 Tag

4） GetComponent

获取游戏对象上的组件，如果有要获取的组件就返回该类型的组件，如果没有，就返回 null。

5） GetComponentInChildren

使用深度优先搜索返回该游戏对象或其任何子项中该类型的组件。（没有返回空）

6） GetComponentInParent

获取在该游戏对象或其任何父项中指定类型的组件。（有只返回一个，没有返回空）

7） GetComponents

返回该游戏对象中指定类型的所有组件。

哔哩哔哩次元游戏人生

8） GetComponentsInChildren

返回该游戏对象或其任何子项中指定类型的所有组件。

9） GetComponentsInParent

返回该游戏对象或其任何父项中指定类型的所有组件。

10）SendMessage

在此游戏对象中的每个 MonoBehaviour 上调用名为 methodName 的方法。

11）SendMessageUpwards

在此游戏对象中的每个 MonoBehaviour 和行为的每个父节点上调用名为 methodName 的方法。

12）SetActive

根据给定的布尔值决定激活还是停用该游戏对象。

13）TryGetComponent

尝试获取指定类型的组件。

4. 静态方法

1） Find

按名称在场景中查找游戏对象并返回它。

2） FindGameObjectsWithTag

根据 Tag 返回一组激活状态的游戏对象。如果未找到游戏对象，则返回空数组。

3） FindWithTag

根据 Tag 返回一个激活状态的游戏对象。如果未找到游戏对象，则返回 null。

二、 Transform

1. 公共方法

1） LookAt

① 声明

public void LookAt(Transform target);

public void LookAt(Transform target, Vector3 worldUp = Vector3.up);

② Parameters-参数

target-要指向的对象。

worldUp-指定向上方向的向量。

③ 描述

旋转变换，使前向矢量指向 target 的当前位置。然后它旋转变换以将其向上方向向量指向向量暗示的方向 worldUp。如果省略该 worldUp 参数，该函数将使用世界 y 轴。worldUp 如果向前方向垂直于，则旋转的向上矢量将仅与矢量匹配 worldUp。

④ 声明

public void LookAt (Vector3 worldPosition);

public void LookAt (Vector3 worldPosition , Vector3 worldUp = Vector3.up);

⑤ Parameters-参数

worldPosition-看向某点

worldUp-指定向上方向的向量。

④ 描述

旋转变换，使前向向量指向 worldPosition。然后它旋转变换以将其向上方向向量指向

哔哩哔哩次元游戏人生

向量暗示的方向 worldUp。如果省略该 worldUp 参数，该函数将使用世界 y 轴。worldUp 如果向前方向垂直于，则旋转的向上矢量将仅与矢量匹配 worldUp。

三、 Vector3

1. 描述

3D 向量和点的表示。

这种结构在整个 Unity 中用于传递 3D 位置和方向。它还包含用于执行常见向量运算的函数。

除了下面列出的函数，其他类也可用于操作向量和点。例如 Quaternion 和 Matrix4x4 类对于旋转或变换向量和点很有用。

2. Static Properity-静态属性

1） back

Vector3(0, 0, -1) 的简写。

2） down

Vector3(0, -1, 0) 的简写。

3） forward

Vector3(0, 0, 1) 的简写。

4） left

Vector3(-1, 0, 0) 的简写。

5） negativeInfinity

Vector3(float.NegativeInfinity, float.NegativeInfinity, float.NegativeInfinity) 的简写。

6） one

Vector3(1, 1, 1) 的简写。

7） positiveInfinity

Vector3(float.PositiveInfinity, float.PositiveInfinity, float.PositiveInfinity) 的简写。

8） right

Vector3(1, 0, 0) 的简写。

9） up

Vector3(0, 1, 0) 的简写。

10）zero-零向量

Vector3(0, 0, 0) 的简写。

3. Static Methods-静态方法

1） Angle

计算两个 3 维向量之间的角度

2） ClampMagnitude-限制模

返回一个向量的副本，其大小被限制为 maxLength。

3） Cross

两个向量的叉乘。

4） Distance

返回 a 向量和 b 向量之间的距离。

咪哩咔哩次元游戏人生

5） Dot

两个向量的点乘。

6） Lerp

在两点之间线性插值。

7） LerpUnclamped

在两个向量之间线性插值。

8） Max

返回由两个向量的最大分量组成的向量。

9） Min

返回由两个向量的最小分量组成的向量。

10）MoveTowards

计算当前位置和指定的位置之间的位置，移动不超过 maxDistanceDelta 指定的距离。

11）Normalize

使这个向量的大小为 1。

12）OrthoNormalize

使向量标准化并相互正交。（即两两向量内积为 0）

13）Project

将一个向量投影到另一个向量上。

14）ProjectOnPlane

将向量投影到由垂直于该平面的法线定义的平面上。

15）Reflect

从法线定义的平面反射向量。

16）RotateTowards

向目标旋转。

17）Scale

将两个向量相乘。

18）SignedAngle

计算向量 from 和 to 之间相对于轴的有符号角。

19）Slerp

在两个向量之间进行球面插值。

20）SlerpUnclamped

在两个向量之间进行球面插值。

21）SmoothDamp

随着时间的推移，逐渐将向量改变为期望的目标。

4. 向量归一化的区别

1） 不同点

Vector3.normalized：不改变当前向量，返回一个新的规范化的向量。

Vector3.Normalized：改变当前向量，当前向量长度为一。

哔哩哔哩次元游戏人生

2） 相同点

实现规范化，让一个向量保持相同的方向，但它的长度为 1，如果这个向量太小而不能被规范化，一个零向量会被返回。

四、 KeyCode 编码

值	描述	值	描述
BackSpace	退格键	Delete	正向删除键
Tab	制表键	Clear	清除键
Return	回车键	Pause	PC 上的暂停键
Escape	退出键	Space	空格键
Keypad0	数字小键盘 0	Keypad1	数字小键盘 1
Keypad2	数字小键盘 2	Keypad3	数字小键盘 3
Keypad4	数字小键盘 4	Keypad5	数字小键盘 5
Keypad6	数字小键盘 6	Keypad7	数字小键盘 7
Keypad8	数字小键盘 8	Keypad9	数字小键盘 9
KeypadPeriod	数字小键盘句号	KeypadDivide	数字小键盘除号
KeypadMultiply	数字小键盘乘号	KeypadMinus	数字小键盘减号
KeypadPlus	数字小键盘加号	KeypadEnter	数字小键盘回车键
KeypadEquals	数字小键盘等号	UpArrow	上箭头键
DownArrow	下箭头键	RightArrow	右箭头键
LeftArrow	左箭头键	Insert	插入键
Home	起始键	End	结束键
PageUp	上一页	PageDown	下一页
F1	F1 功能键	F2	F2 功能键
F3	F3 功能键	F4	F4 功能键
F5	F5 功能键	F6	F6 功能键
F7	F7 功能键	F8	F8 功能键
F9	F9 功能键	F10	F10 功能键
F11	F11 功能键	F12	F12 功能键
Alpha0	键盘顶部的数字键 0	Alpha1	键盘顶部的数字键 1
Alpha2	键盘顶部的数字键 2	Alpha3	键盘顶部的数字键 3
Alpha4	键盘顶部的数字键 4	Alpha5	键盘顶部的数字键 5
Alpha6	键盘顶部的数字键 6	Alpha7	键盘顶部的数字键 7
Alpha8	键盘顶部的数字键 8	Alpha9	键盘顶部的数字键 9
Exclaim	感叹号键	DoubleQuote	双引号键
Hash	哈希键	Dollar	美元符号键
Percent	百分号键	Ampersand	&键
Quote	单引号键	LeftParen	左括号键
RightParen	右括号键	Asterisk	星号键
Plus	加号键	Comma	逗号键
Minus	减号键	Period	句号键
Slash	斜杠键	Colon	冒号键
Semicolon	分号键	Less	小于符号键

哔哩哔哩欢乐新天地

Equals	等号键	Greater	大于符号键
Question	问号键	At	@符号键
LeftBracket	左中括号键	Backslash	反斜杠键
RightBracket	右中括号键	Caret	托字符键
Underscore	下划线键	BackQuote	反引号键
A	字母 A 键	B	字母 B 键
C	字母 C 键	D	字母 D 键
E	字母 E 键	F	字母 F 键
G	字母 G 键	H	字母 H 键
I	字母 I 键	J	字母 J 键
K	字母 K 键	L	字母 L 键
M	字母 M 键	N	字母 N 键
O	字母 O 键	P	字母 P 键
Q	字母 Q 键	R	字母 R 键
S	字母 S 键	T	字母 T 键
U	字母 U 键	V	字母 V 键
W	字母 W 键	X	字母 X 键
Y	字母 Y 键	Z	字母 Z 键
LeftCurlyBracket	左大括号键	Pipe	|
RightCurlyBracket	由大括号键	Tilde	波浪号
Numlock	小键盘锁定键	CapsLock	大写锁定键
ScrollLock	滚动锁定键	RightShift	右换挡键
LeftShift	左换挡键	RightControl	右控制键
LeftControl	左控制键	RightAlt	右 Alt 键
LeftAlt	左 Alt 键	LeftMeta	如果在输入管理器设置中启用了物理键，则映射到左 Windows 键或左命令键，否则仅映射到左命令键。
LeftCommand	左命令键	LeftApple	左 Apple 键
LeftWindows	左 Windows 键	RightMeta	如果在输入管理器设置中启用了物理键，则映射到右侧 Windows 键或右侧 Command 键，否则仅映射到右侧 Command 键。
RightCommand	右命令键	RightApple	右 Apple 键
RightWindows	右 Windows 键	AltGr	右侧更改键
Help	帮助键	Print	打印键
SysReq	系统请求键	Break	中断键
Menu	菜单键	Mouse0	第 1 个鼠标键（鼠

哔哩哔哩欢乐游戏人生

			标左键）
Mouse1	第 2 个鼠标键（鼠标右键）	Mouse2	第 3 个鼠标键（鼠标中键）
Mouse3	第 4 个鼠标键	Mouse4	第 5 个鼠标键
Mouse5	第 6 个鼠标键	Mouse6	第 7 个鼠标键

五、 数学函数库

1. 绝对值-Mathf.Abs

① public static float Abs(float f);

返回浮点型的绝对值

② public static int Abs(int value);

返回整型的绝对值

2. 反余弦-Mathf.Acos

public static float Acos(float f);

根据余弦值，返回弧度。

3. 近似-Mathf.Approximately

public static bool Approximately(float a, float b);

比较两个浮点数值，看它们是否非常接近，由于浮点数值不精确，不建议使用等于来比较它们。

4. 反正弦-Mathf.Asin

public static float Asin(float f);

根据正弦值，返回弧度。

5. 反正切-Mathf.Atan2

public static float Atan2(float y, float x);

根据 Y 除以 X，返回对应的弧度。返回值表示相对直角三角形，对角的角，其中 X 是临边边长，Y 是对边边长。返回值是 X 轴和一个二维向量开始于 0，结束在(x,y)处之间的角。

6. 反正切-Mathf.Atan

public static float Atan(float f);

根据正切值，返回以弧度为单位的角度，返回值介于负二分之 π 和正二分之 π 之间。

7. 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. IPointerEnterHandler

当鼠标光标移入该对象时触发。

降维次元游戏人生

2. IPointExitHandler

当鼠标光标移出该对象时触发。

3. IPointDownHandler

鼠标点击 A 对象，按下鼠标时 A 对象响应此事件。

4. IPointUpHandler

鼠标点击 A 对象，抬起鼠标时 A 对象响应此事件。(无论鼠标在何处抬起，都会在 A 对象中响应此事件)响应此事件的前提，是 A 对象必须响应过 IPointDownHandler。

5. IPointClickHandler

鼠标点击 A 对象，抬起鼠标时 A 对象响应此事件（鼠标按下和抬起时鼠标要处于同一对象上）。

6. IDragHandler

当鼠标在 A 对象按下并拖拽时，A 对象每帧响应一次此事件。（如果不实现此接口，则后面的四个接口方法都不会触发）

7. IInitializePotentialDragHandler

当鼠标在 A 对象按下还没开始拖拽时，A 对象响应此事件。此接口事件与 IPointDownHandler 接口类似，二者的执行顺序：先执行 IPointDownHandler，然后 执行此接口事件（需先实现 IDragHandler 接口）

8. IBeginDragHandler

当鼠标在 A 对象按下并开始拖拽时，A 对象响应此事件，此事件在 IInitializePotentialDragHandler 之后响应，在 OnDrag 之前响应。

9. IEndDragHanlder

当鼠标抬起时，A 对象响应此事件。

10. IDropHandler

A、B 对象必须均实现 IDropHandler 接口，且 A 至少实现 IDragHandler 接口。当鼠标从 A 对象上开始拖拽，在 B 对象上抬起时，B 对象响应此事件。此时 name 获取的是 B 对象的 name 属性，eventData.PointerDrag 表示发起拖拽的对象

哔哩哔哩次元游戏人生

Unity 案例

一、 2D 角色移动

二、 3D 角色移动

三、 3D 相机控制

四、 输入管理-InputManager

构建输入管理，主要使用事件来传递玩家输入的信息。

```
public class InputManager : SingleComponent<InputManager>
{
    #region 水平输入事件完整声明格式
    /*
    * 事件的拥有者(Event Source) -----> InputManager 类
    * 事件(Event) -----> OnSelectSkill
    * 事件的响应者(Event Subscriber) -----> EquipmentSlotParent
    * 事件处理器(Event Handler) -----> SetSelectSkill
    * 事件的订阅关系(+= Operator) -----> +=
    */
    public delegate void InputHorizontalEventHandler(float _horizontalValue);//为了事件而声明的委托类型最好加上 EventHandler 后缀,这个委托可以指向一个无返回值的方法，
    private InputHorizontalEventHandler inputHorizontalEventHandler;//完整声明格式中最重要的就是这个委托类型的字段，用来存储事件处理器
    public event InputHorizontalEventHandler OnInputHorizontal
    {
        add
        {
            inputHorizontalEventHandler += value;//我们虽然不知道未来传进来的事件处理器是什么，我们可以用上下文关键字 Value 来表示
        }
        remove
        {
            inputHorizontalEventHandler -= value;//移除事件处理器
        }
    }
    #endregion
    //action 委托必定没有返回值,func 委托必定具有一个返回值
    public event Action<float> OnInputVertical;//垂直输入事件的简略声明

    protected override void Awake()
    {
        base.Awake();
    }
```

哔哩哔哩次元游戏人生

```
    private void Update()
    {
        #region 水平方向输入
        if (inputHorizontalEventHandler != null)
        {
            inputHorizontalEventHandler(Input.GetAxisRaw("Horizontal"));//只能由事件拥有者在其内部调用内部逻辑调用
        }
        #endregion
        #region 垂直方向输入
        if (OnInputVertical != null)
        {
            OnInputVertical(Input.GetAxisRaw("Vertical"));
        }
        #endregion
    }
}
```

五、 场景异步加载与过渡

六、 自定义 URP 后处理

七、 通用对象池

1. 概述

对象池模式(The Object Pool Pattern)是单例模式的一个变种，它提供了获取一系列相同对象实例的入口。当你需要对象来代表一组可替代资源的时候就变的很有用，每个对象每次可以被一个组件使用。

2. 理解

对象池模式管理一个可代替对象的集合。组件从池中借出对象，用它来完成一些任务并当任务完成时归还该对象。被归还的对象接着满足请求，不管是同一个组件还是其他组件的请求。对象池模式可以管理那些代表的现实资源或者通过重用来分摊昂贵初始化代价的对象。

哔哩哔哩次元游戏人生

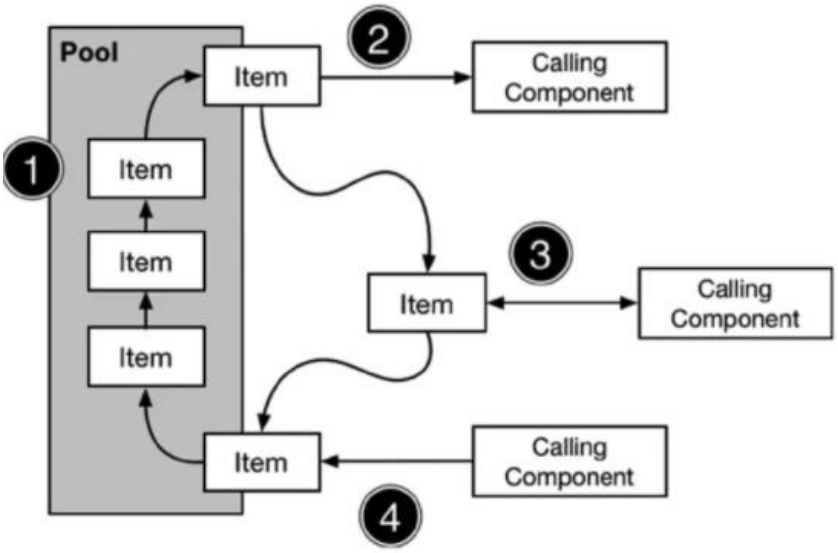


图22.　　对象池

第一步操作是构建对象池。

第二步操作是借出对象。

第三步操作是组件用借出的对象来完成一些任务。这时候并不需要对象池再做什么，但是这也意味着该对象将被租借一段时间并且不能在被其他组件借出。

第四步操作就是归还，组件归还借出的对象这样可以继续满足其他的租借请求。

3. 优点

复用池中对象，没有分配内存和创建堆中对象的开销， 没有释放内存和销毁堆中对象的开销， 进而减少垃圾收集器的负担， 避免内存抖动；不必重复初始化对象状态， 对于比较耗时的 constructor 和 finalize 来说非常合适。

4. 缺点

① 并发环境中， 多个线程可能同时需要获取池中对象， 进而需要在堆数据结构上进行同步或者因为锁竞争而产生阻塞， 这种开销要比创建销毁对象的开销高数百倍。

② 由于池中对象的数量有限， 势必成为一个可伸缩性瓶颈。

③ 很难正确的设定对象池的大小， 如果太小则起不到作用， 如果过大， 则占用内存资源高， 可以起一个线程定期扫描分析， 将池压缩到一个合适的尺寸以节约内存，但为了获得不错的分析结果， 在扫描期间可能需要暂停复用以避免干扰(造成效率低下)， 或者使用非常复杂的算法策略(增加维护难度)。

④ 设计和使用对象池容易出错， 设计上需要注意状态同步， 这是个难点， 使用上可能存在忘记归还(就像 c 语言编程忘记 free 一样)， 重复归还(可能需要做个循环判断一下是否池中存在此对象， 这也是个开销)， 归还后仍旧使用对象(可能造成多个线程并发使用一个对象的情况)等问题。

5. 简单对象池

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

namespace UniversalPool
{
```

```
[DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
[AddComponentMenu("GameObject/PoolObject")]
public class PoolObject : MonoBehaviour
{
    public string poolName;//对象池名称
    public bool isPooled; //定义对象是在池中等待还是在使用中
}

public enum PoolInflationType//对象池膨胀类型
{
    INCREASE,// 当对象池膨胀时，将一个对象添加到池中。
    DOUBLE// 当对象池膨胀时，将池的大小增加一倍
}

class Pool
{
    private Stack<PoolObject> availableObjStack = new Stack<PoolObject>();//Stack
栈是一个先进后出（LIFO）表
    private Queue<PoolObject> availableObjQueue = new
Queue<PoolObject>();//Queue 队列是先进先出(FIFO)

    //未使用对象的根节点
    private GameObject rootObj;
    private PoolInflationType inflationType;
    private string poolName;
    private int objectsInUse = 0;

    //构造函数
    public Pool(string poolName, GameObject poolObjectPrefab, GameObject
rootPoolObj, int initialCount, PoolInflationType type,bool useStack = true)
    {
        if (poolObjectPrefab == null)
        {
#if UNITY_EDITOR
            Debug.LogError("[ObjPoolManager] 对象池预制体为空 !");
#endif
            return;
        }
        this.poolName = poolName;
        this.inflationType = type;
        this.rootObj = new GameObject(poolName + "Pool");//为对象池命名
        this.rootObj.transform.SetParent(rootPoolObj.transform,
false);//worldPositionStays 如果为 true，则修改父相对位置、缩放和旋转，以使对象保持与
以前相同的世界空间位置、旋转和缩放。
```

哔哩哔哩次元游戏人生

```
        GameObject go = GameObject.Instantiate(poolObjectPrefab);
        PoolObject po = go.GetComponent<PoolObject>();
        if (po == null)
        {
            po = go.AddComponent<PoolObject>();//为对象添加 PoolObject 组件
        }
        po.poolName = poolName;//设置对象所属对象池名称
        AddObjectToPool(po, useStack);//确保对象池中至少存在一个对象

        //填充对象池
        FillPool(Mathf.Max(initialCount, 1), useStack);
    }

    //添加到栈对象池,o(1)
    private void AddObjectToPool(PoolObject po,bool useStack = true)
    {
        po.gameObject.SetActive(false);
        po.gameObject.name = poolName;
        if (useStack)
        {
            availableObjStack.Push(po);//入栈
        }
        else
        {
            availableObjQueue.Enqueue(po);//入队列,将对象添加到。
        }
        po.isPooled = true;//对象是在对象池中
        //添加到根节点
        po.gameObject.transform.SetParent(rootObj.transform, false);
    }

    /// <summary>
    ///  填充对象池到指定数量
    /// </summary>
    /// <param name="initialCount">填充的数量</param>
    private void FillPool(int initialCount, bool useStack = true)
    {
        for (int index = 0; index < initialCount; index++)
        {
            PoolObject po = null;
            if (useStack)
            {
                po = GameObject.Instantiate(availableObjStack.Peek());//Peek()返
```

哔哩哔哩次元游戏人生

```
回位于栈顶的对象但不将其移除
                }
                else
                {
                    po = GameObject.Instantiate(availableObjQueue.Peek());//Peek()返
回位于队首的对象但不将其移除
                }
                AddObjectToPool(po, useStack);
            }
        }

        //栈中的下一个可用对象，O(1)
        public GameObject NextAvailableObject(bool autoActive,bool useStack = true)
        {
            PoolObject po = null;
            if (useStack ? availableObjStack.Count > 1 : availableObjQueue.Count > 1)//
判断当前是否有足够可使用对象
            {
                //有就出栈，Pop()删除并返回 Stack 顶部的对象，Dequeue 删除并
返回队首的对象
                po = useStack ? availableObjStack.Pop() :
availableObjQueue.Dequeue();
            }
            else
            {
                int increaseSize = 0;
                switch (inflationType)
                {
                    case PoolInflationType.INCREASE:
                        increaseSize = 1;
                        break;
                    case PoolInflationType.DOUBLE:
                        {
                            increaseSize = useStack ? availableObjStack.Count :
availableObjQueue.Count + Mathf.Max(objectsInUse, 0);
                            break;
                        }
                }
    #if UNITY_EDITOR
                Debug.Log(string.Format("Growing pool {0}: {1} populated",
poolName, increaseSize));
    #endif
                if (increaseSize > 0)
                {
```

哔哩哔哩次元游戏人生

```
                    FillPool(increaseSize, useStack);//填充对象池
                    if (useStack)
                    {
                        po = availableObjStack.Pop();//有就出栈，Pop()删除并返回
Stack 顶部的对象
                    }
                    else
                    {
                        po = availableObjQueue.Dequeue();//有就出队列，Dequeue
删除并返回队首的对象
                    }
                }
            }

            GameObject result = null;
            if (po != null)
            {
                objectsInUse++;//已使用对象数增加
                po.isPooled = false;
                result = po.gameObject;
                if (autoActive)
                {
                    result.SetActive(true);
                }
            }

            return result;
        }

        //将对象返回到对象池中，O(1)
        public void ReturnObjectToPool(PoolObject po,bool useStack = true)
        {
            if (poolName.Equals(po.poolName))//Equals(String)确定此实例是否与另一
个指定的 String 对象具有相同的值
            {
                objectsInUse--;//已使用对象减少
                /* 我们本可以使用 Stack.Contains(Object)来检查该对象是否在对象
池中。
                 * 但是会使此方法的时间复杂度变为 O（n）*/
                if (po.isPooled)
                {
    #if UNITY_EDITOR
                    Debug.LogWarning(po.gameObject.name + " 已经在对象池里
了。 您为什么要再次返回？ 请检查用法。");
```

哔哩哔哩次元游戏人生

```
#endif
                }
                else
                {
                    AddObjectToPool(po, useStack);
                }
            }
            else
            {
                Debug.LogError(string.Format("试图将对象添加到不正确的对象池中
{0} {1}", po.poolName, poolName));
            }
        }
    }
}
```

6. 对象池管理

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

namespace UniversalPool
{
    [DisallowMultipleComponent]//不能在一个对象上重复添加该脚本。
    [AddComponentMenu("GameObject/PoolResourceManager")]
    public class PoolResourceManager : MonoBehaviour
    {
        //使用数据字典为对象池添加关键字
        private Dictionary<string, Pool> poolDict = new Dictionary<string, Pool>();

        private static PoolResourceManager instance = null;

        public static PoolResourceManager Instance
        {
            get
            {
                if (instance == null)//不存在实例就创建一个
                {
                    GameObject go = new GameObject("ResourceManager",
typeof(PoolResourceManager));
                    go.transform.localPosition = new Vector3(0, 0, 0);
                    instance = go.GetComponent<PoolResourceManager>();

                    if (Application.isPlaying)
                    {
                        DontDestroyOnLoad(instance.gameObject);
```

哔哩哔哩次元游戏人生

```
                }
                else
                {
                    Debug.LogWarning("[ResourceManager] 您最好在编辑器模式下忽略 ResourceManager");
                }
            }

            return instance;
        }
    }
    #region 初始化对象池
    public void InitPool(string poolName, int size, PoolInflationType type = PoolInflationType.DOUBLE,bool useStack = true)
    {
        if (poolDict.ContainsKey(poolName))//数据字典中是否已有关键字
        {
            return;
        }
        else
        {
            GameObject Prefab = Resources.Load<GameObject>(poolName);//在 Resources 文件夹中获取
            if (Prefab == null)
            {
                Debug.LogError("[ResourceManager] Invalide prefab name for pooling :" + poolName);
                return;
            }
            poolDict[poolName] = new Pool(poolName, Prefab, gameObject, size, type, useStack);
        }
    }

    /// <summary>
    /// 初始化对象池
    /// </summary>
    /// <param name="cell">需要初始化的对象</param>
    /// <param name="size">对象数量</param>
    /// <param name="type">膨胀类型</param>
    public void InitPool(GameObject cell, int size, PoolInflationType type = PoolInflationType.DOUBLE, bool useStack = true)
    {
        if (poolDict.ContainsKey(cell.name))
```

哔哩哔哩次元游戏人生

```
                {
                    return;
                }
                else
                {
                    poolDict[cell.name] = new Pool(cell.name, cell, gameObject, size, type,
useStack);
                }
            }
            #endregion

            /// <summary>
            ///  获取池中的可用对象,如果池中没有可用的对象则为 null。
            /// </summary>
            /// <param name="poolName">对象池名称</param>
            /// <param name="autoActive">是否自动激活</param>
            /// <param name="autoCreate">自动创建对象数量</param>
            public GameObject GetObjectFromPool(string poolName, bool autoActive = true,
int autoCreate = 0, bool useStack = true)
            {
                GameObject result = null;

                if (!poolDict.ContainsKey(poolName) && autoCreate > 0)
                {
                    InitPool(poolName, autoCreate, PoolInflationType.INCREASE);
                }

                if (poolDict.ContainsKey(poolName))
                {
                    Pool pool = poolDict[poolName];
                    result = pool.NextAvailableObject(autoActive, useStack);
    #if UNITY_EDITOR
                    if (result == null)//对象池中找不到可用对象
                    {
                        Debug.LogWarning("[ResourceManager]:No object available in " +
poolName);
                    }
    #endif
                }
    #if UNITY_EDITOR
                else
                {
                    Debug.LogError("[ResourceManager]:Invalid pool name specified: " +
poolName);
```

哔哩哔哩次元游戏人生

```
            }
#endif
            return result;
        }

        /// <summary>
        ///  将 GameObject 返回对象池
        /// </summary>
        /// <param name="go"></param>
        public void ReturnObjectToPool(GameObject go, bool useStack = true)
        {
            PoolObject po = go.GetComponent<PoolObject>();
            if (po == null)
            {
#if UNITY_EDITOR
                Debug.LogWarning("指定的对象不是对象池实例: " + go.name);
#endif
            }
            else
            {
                Pool pool = null;
                //out 关键字是只出不进
                if (poolDict.TryGetValue(po.poolName, out pool))//TryGetValue 获取与
指定键关联的值
                {
                    pool.ReturnObjectToPool(po, useStack);
                }
#if UNITY_EDITOR
                else
                {
                    Debug.LogWarning("没有名称可用的对象池: " + po.poolName);
                }
#endif
            }
        }

        /// <summary>
        ///  将对象返回对象池
        /// </summary>
        /// <param name="t"></param>
        public void ReturnTransformToPool(Transform t, bool useStack = true)
        {
            if (t == null)
            {
```

哔哩哔哩次元游戏人生

```
    #if UNITY_EDITOR
                        Debug.LogError("[ResourceManager] try to return a null transform to
pool!");
    #endif
                        return;
                    }
                    ReturnObjectToPool(t.gameObject, useStack);
                }


                //判断该对象是否有对象池
                public bool HasPool(GameObject cell)
                {
                    return poolDict.ContainsKey(cell.name);
                }
            }
    }
```

哔哩哔哩次元游戏人生

技巧

一、 修改 Unity 缓存位置

1. 修改 Unity GICache

打开 Unity 的编辑器，找到 Editor/Preferences/GICache，修改缓存位置。

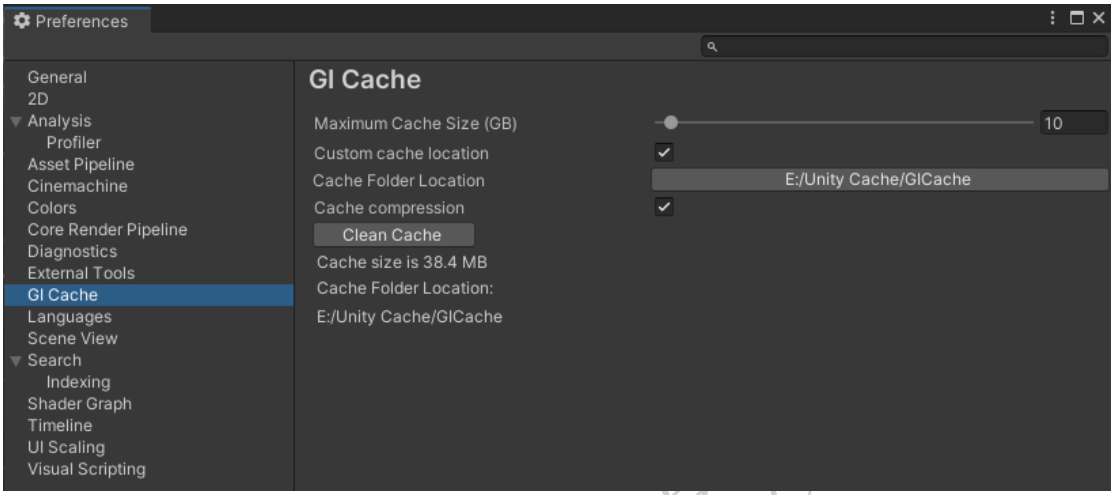


图23. 图 1.1 示意图

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24. 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25. 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26. 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27. 图 1.5 找到我的电脑的属性选项

② 打开高级系统设置

图28. 图 1.6 找到高级系统设置

③ 打开环境变量

图29.　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30.　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

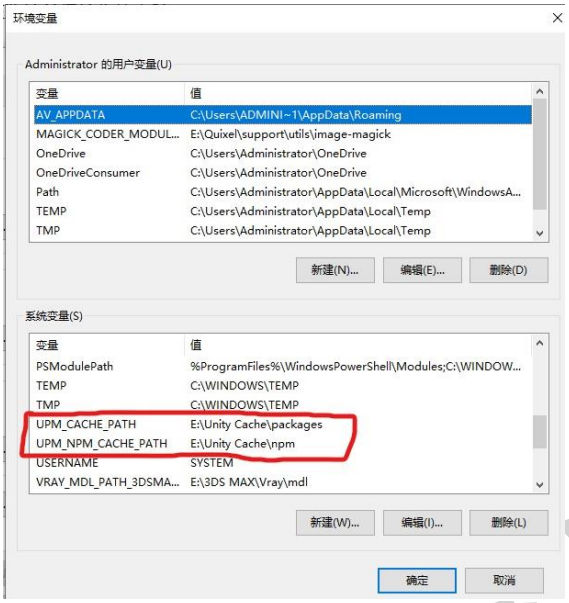


图31.　　图 1.8 新建环境变量

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32.　　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。

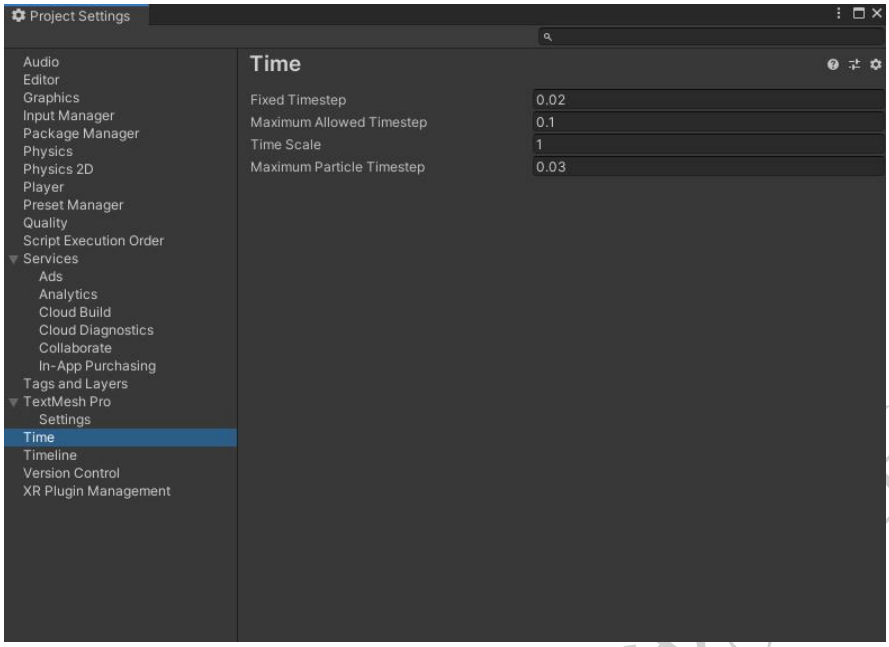


图33.　　图 3.1 TimeManager 属性面板

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

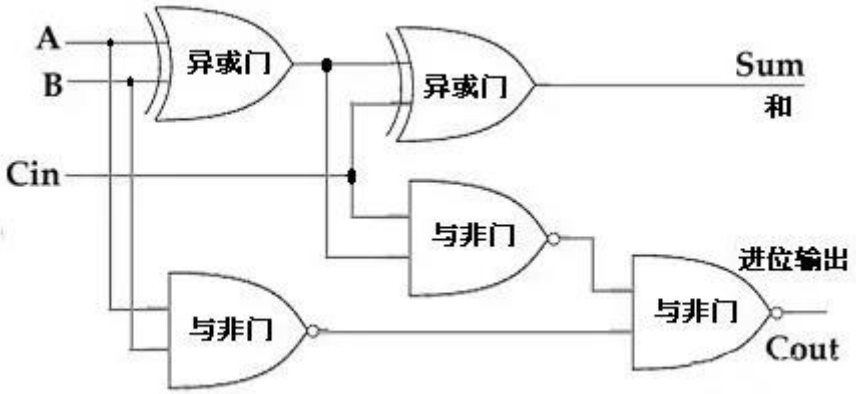


图34. 图 6.2 全加器逻辑电路

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

哔哩哔哩你干游戏人生

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。

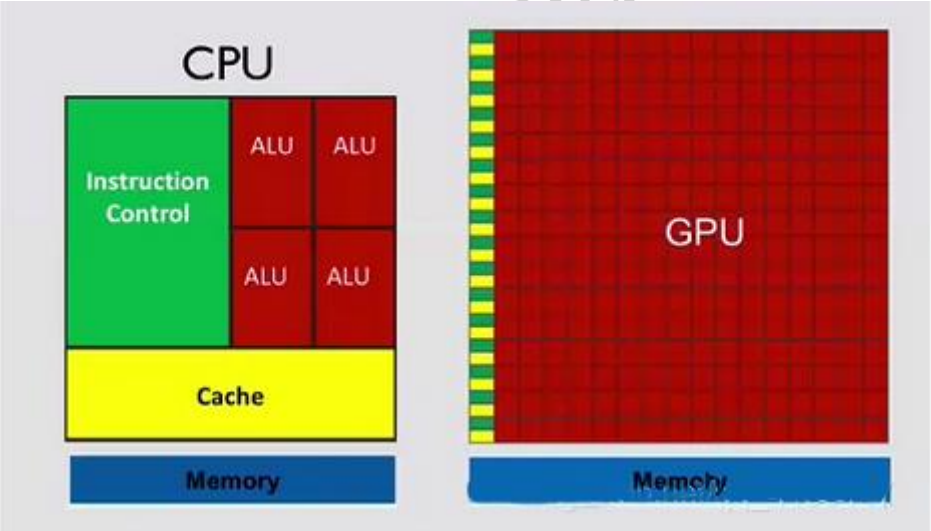


图35.　　图 6.3 架构图

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

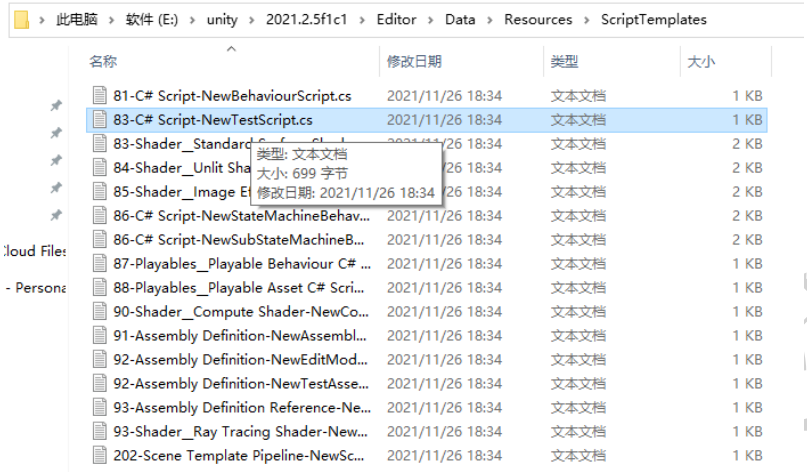


图36. 图 7.1 unity 脚本模板

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。

哔哩哔哩次元旗舰店

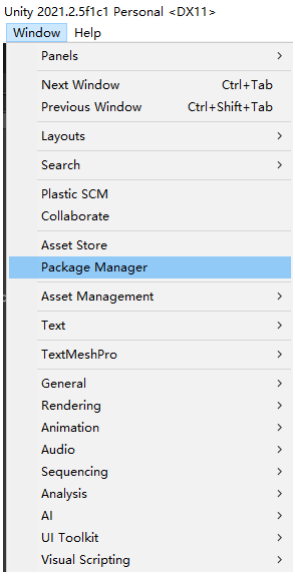


图37.　　图 9.1 打开 PackageManager

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。

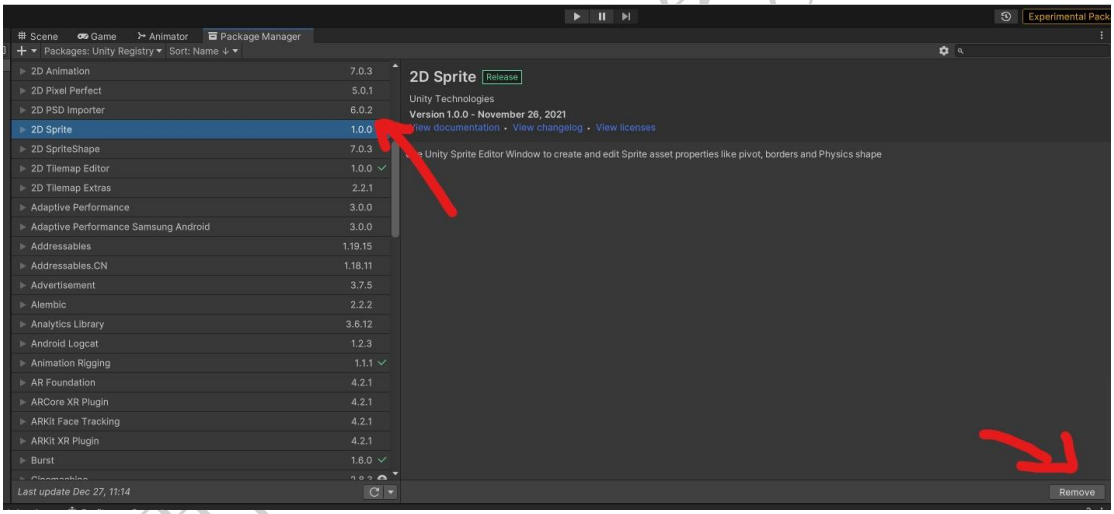


图38.　　图 9.2 将 2D Sprite 导入

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。

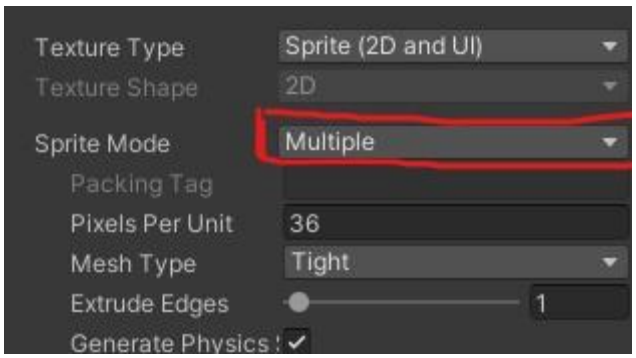


图39.　　图 9.3 设置精灵图为多图模式

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。

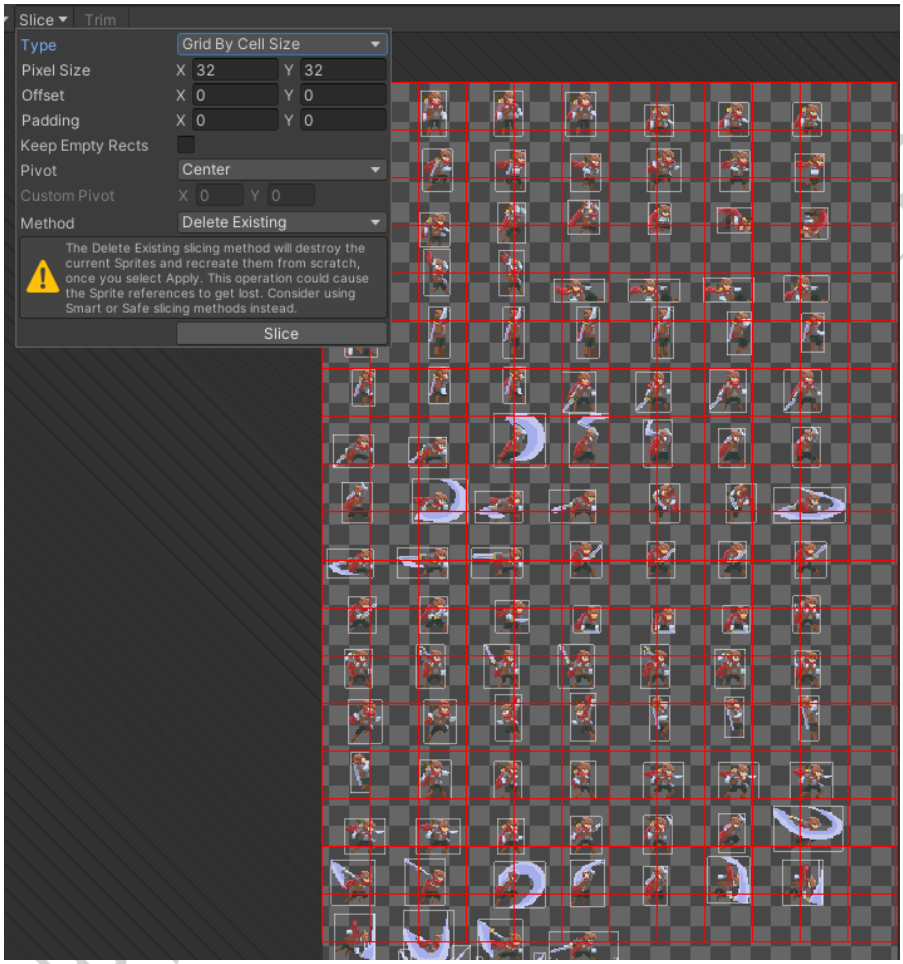


图40.　　图 9.4 按单元格大小切片

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。

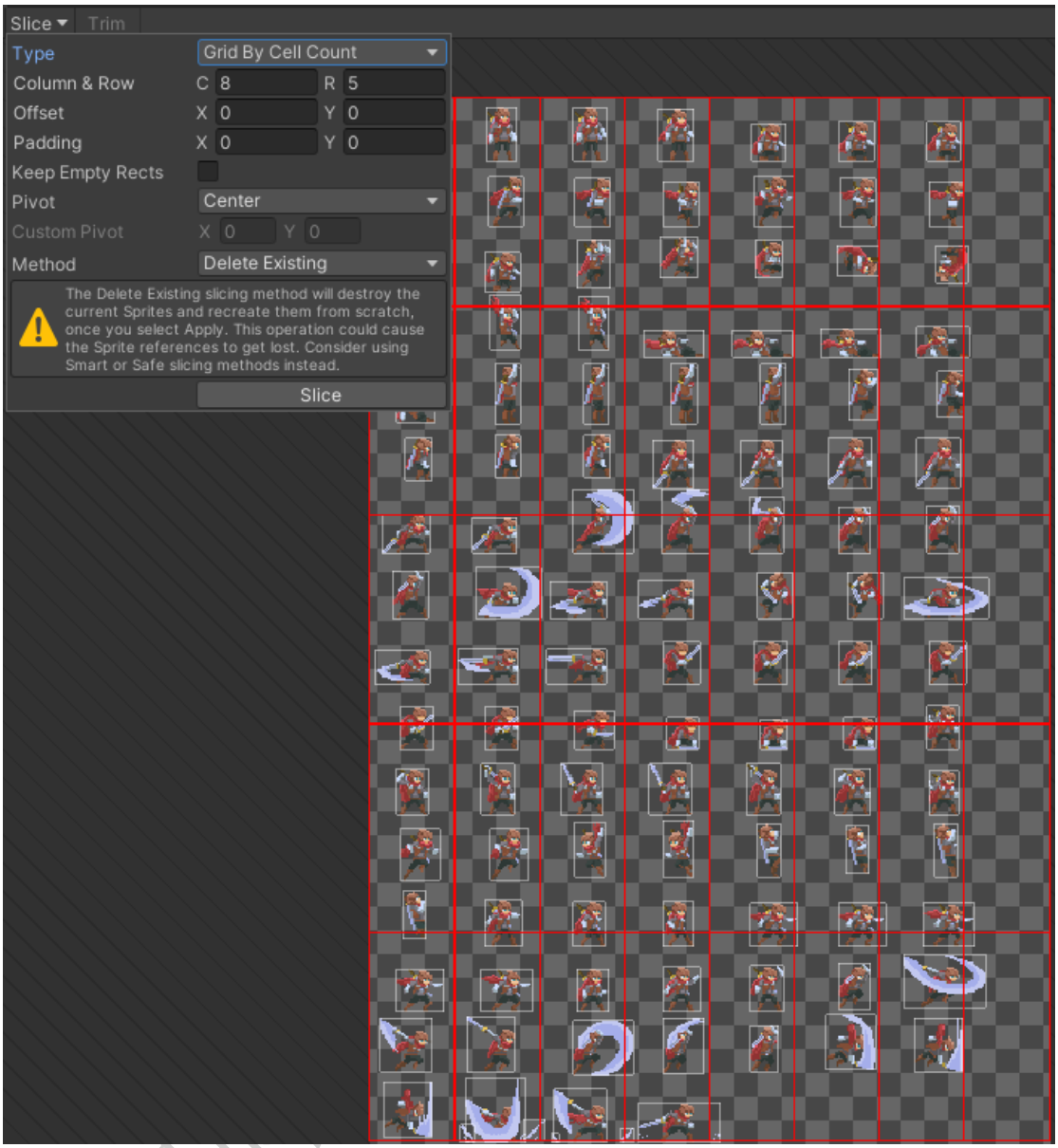


图41.　　图 9.5 按单元格数量切片

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。

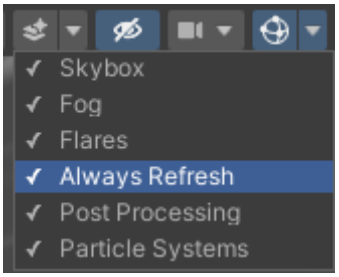


图42.　　保持实时刷新

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。

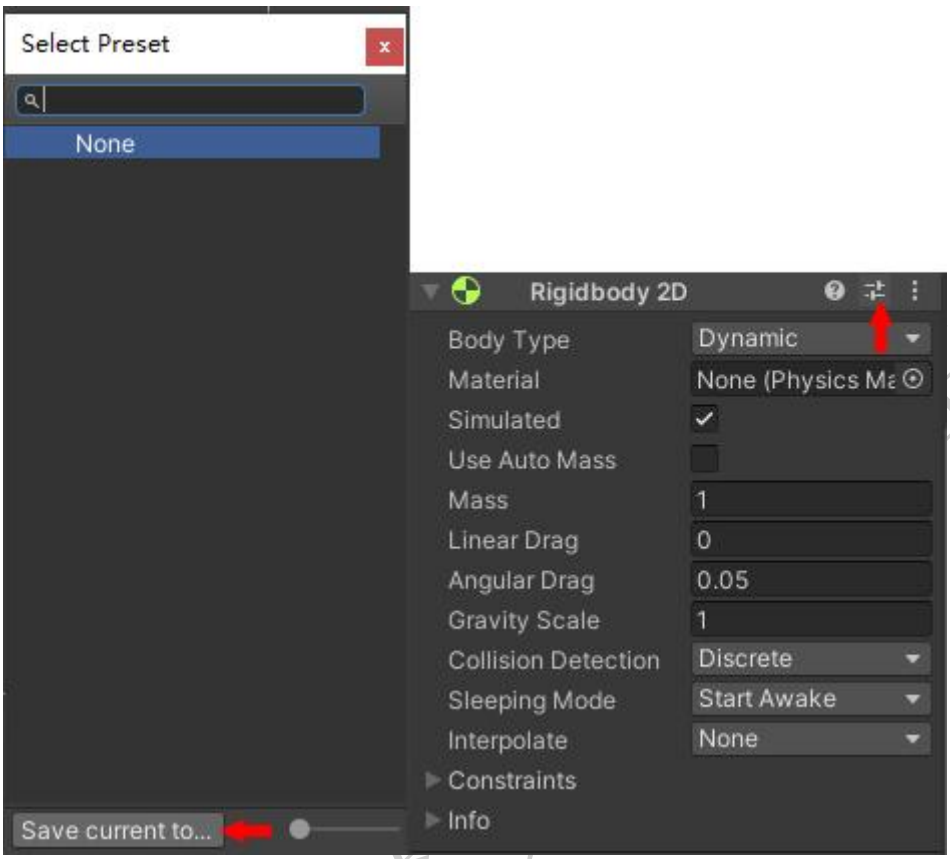


图43.　　组件预设

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。

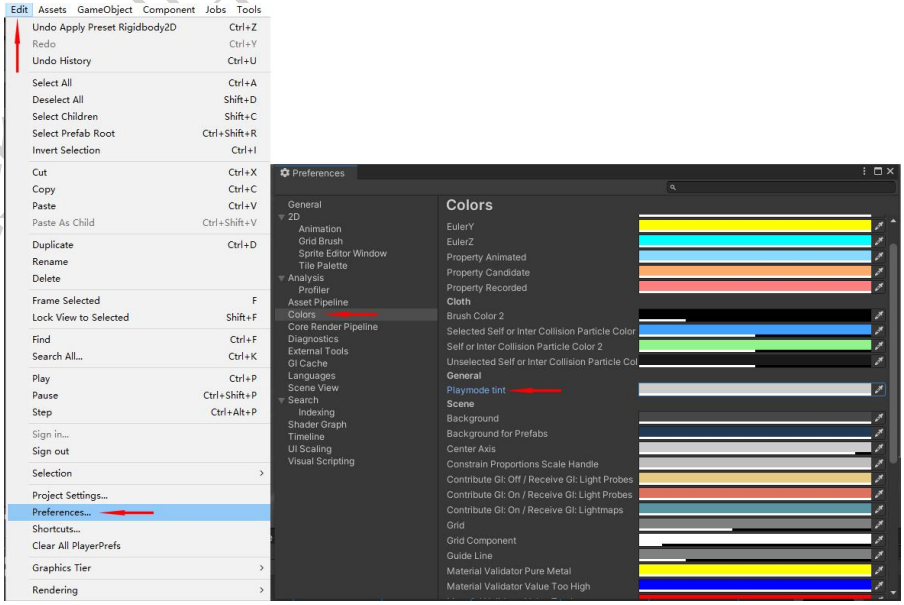


图44.　　修改 Playmode 颜色

十二、 修改鼠标样式和启动图像

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

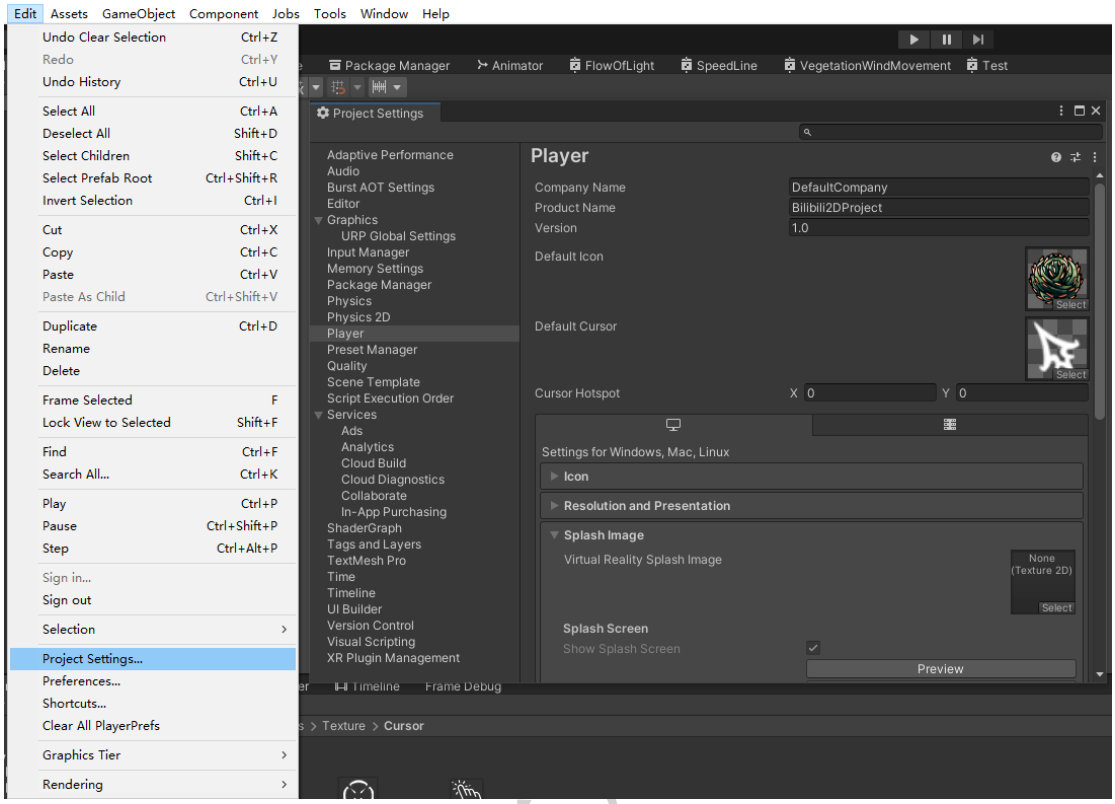


图45. Payer 设置界面

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。

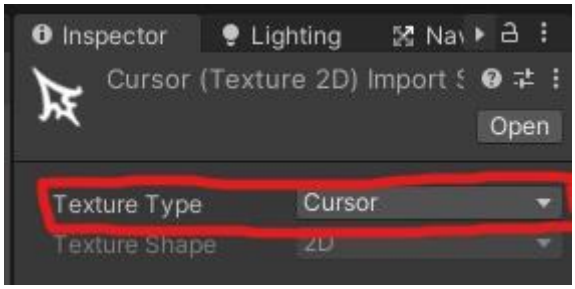


图46. 鼠标样式的纹理类型

2. 设置启动图像

在 Player 设置中找到 Splash Image 设置。

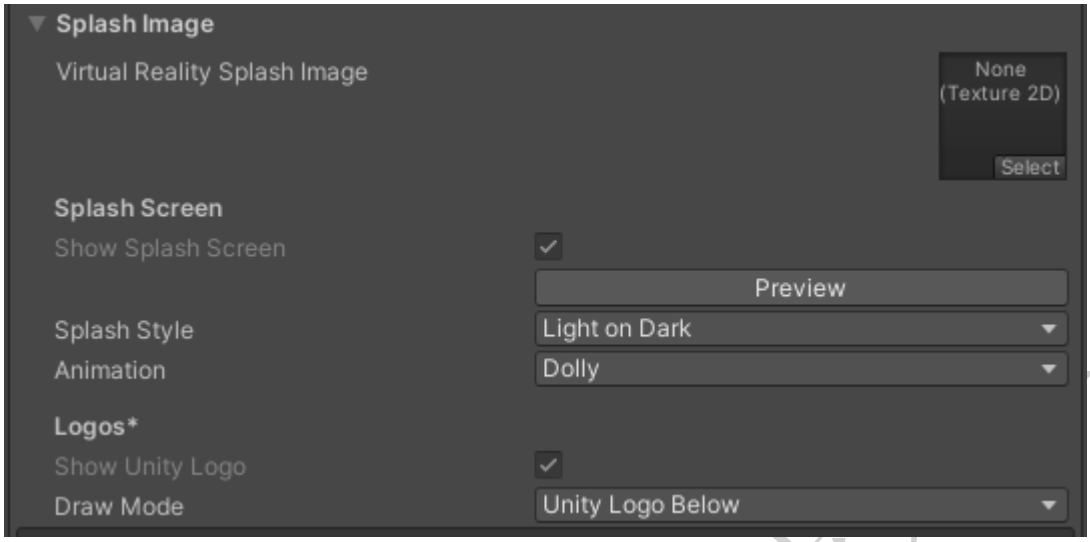


图47. Splash Image

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。

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

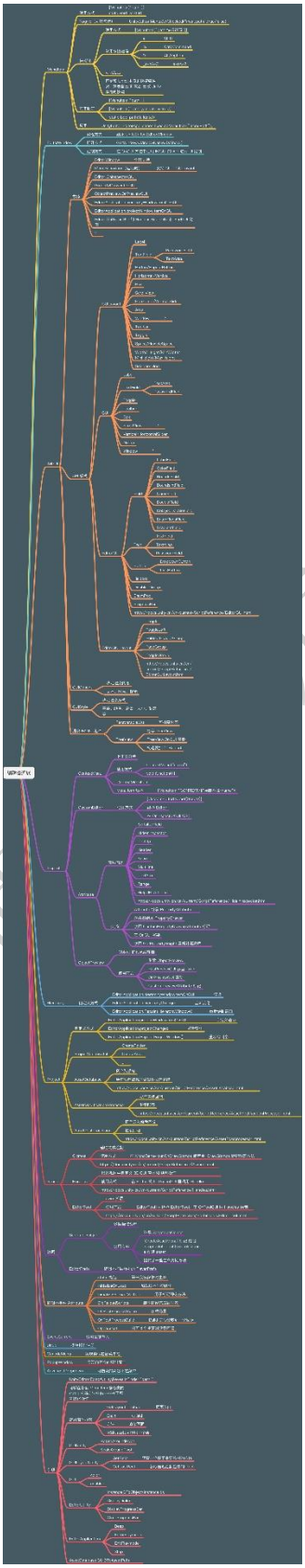


图48.　　图 1. 1 知识地图

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。

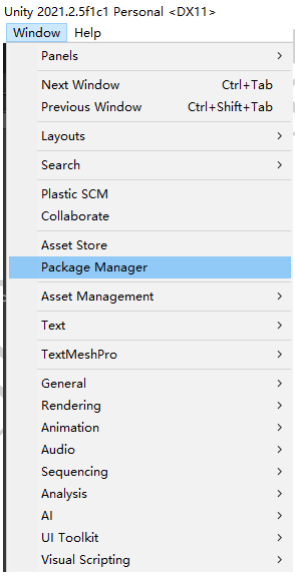


图49. 图 3.1 打开 PackageManager

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

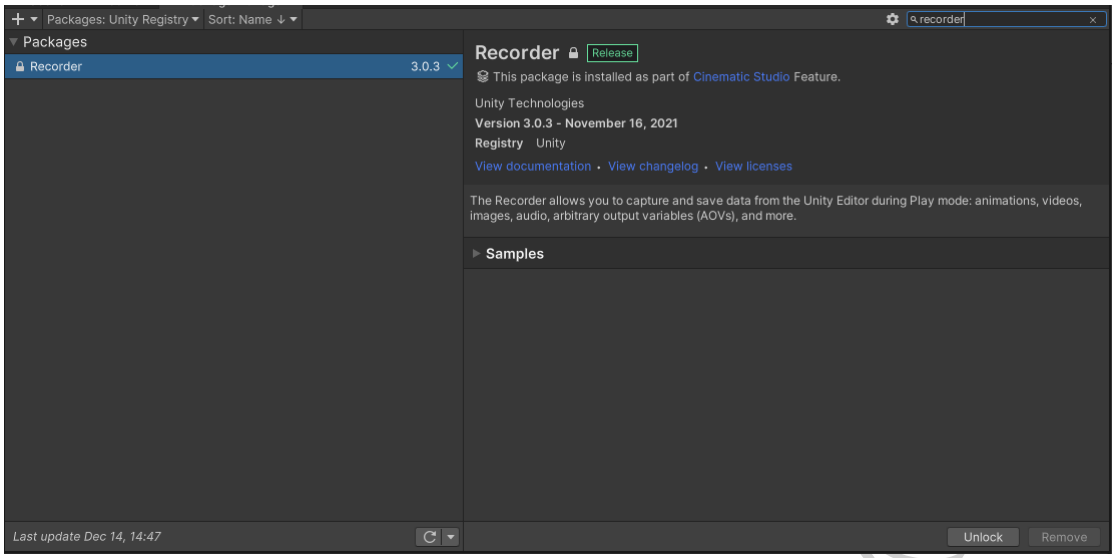


图50.　　图 3.2 下载 Recorder

5. 打开 Recorder 界面

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

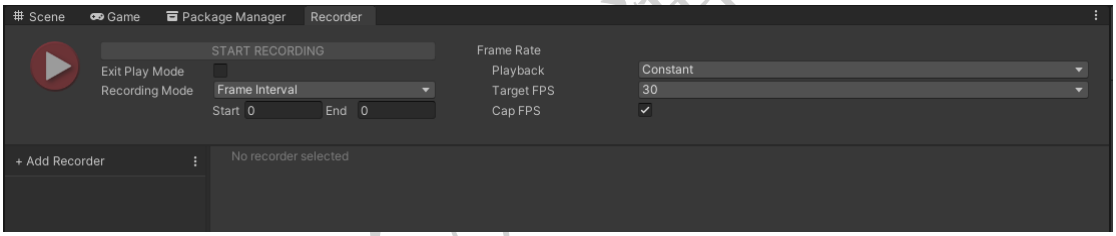


图51.　　图 3.3 Recorder 界面

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。

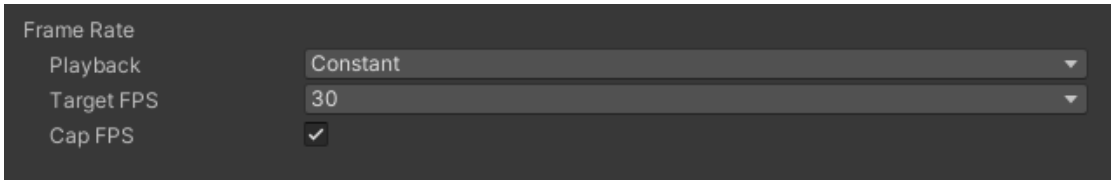


图52.　　图 3.4 Frame Rate

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53. 表 3.1 占位符功能表

8. Capture-捕捉设置

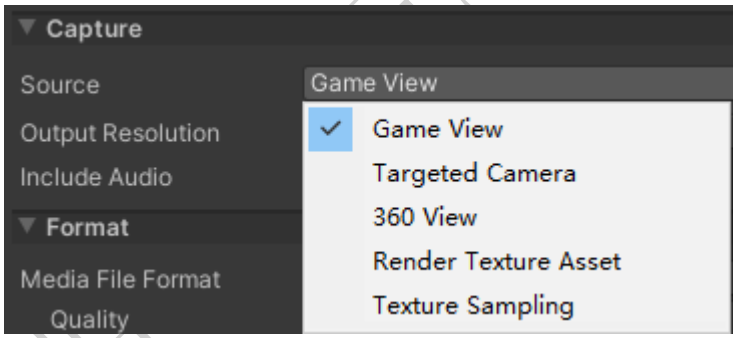


图54. 图 3.5 捕捉设置

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。

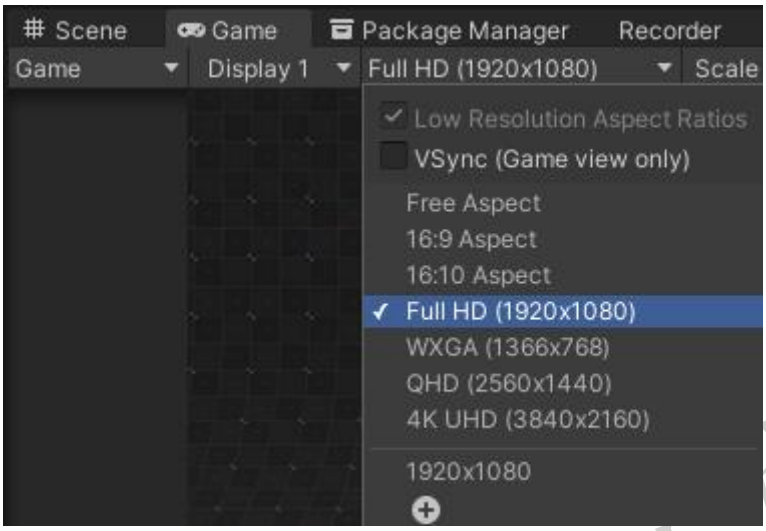


图55.　　图 3.6 游戏视图的分辨率

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。

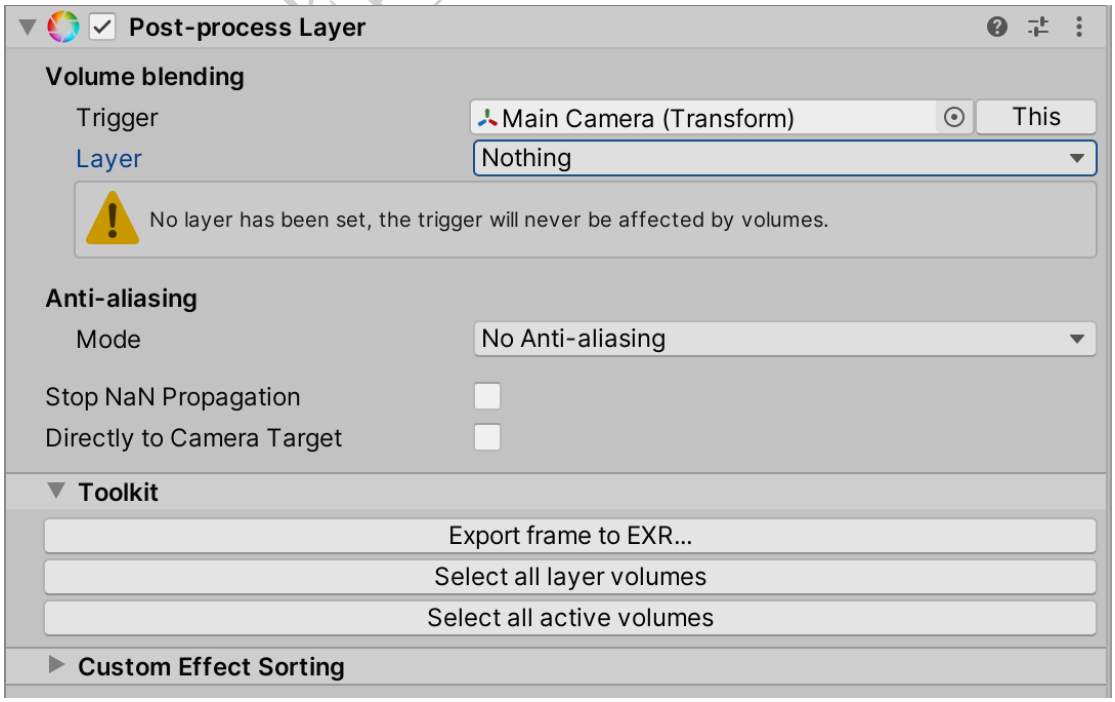


图56.　　图 12.1 Post Process Layer

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.

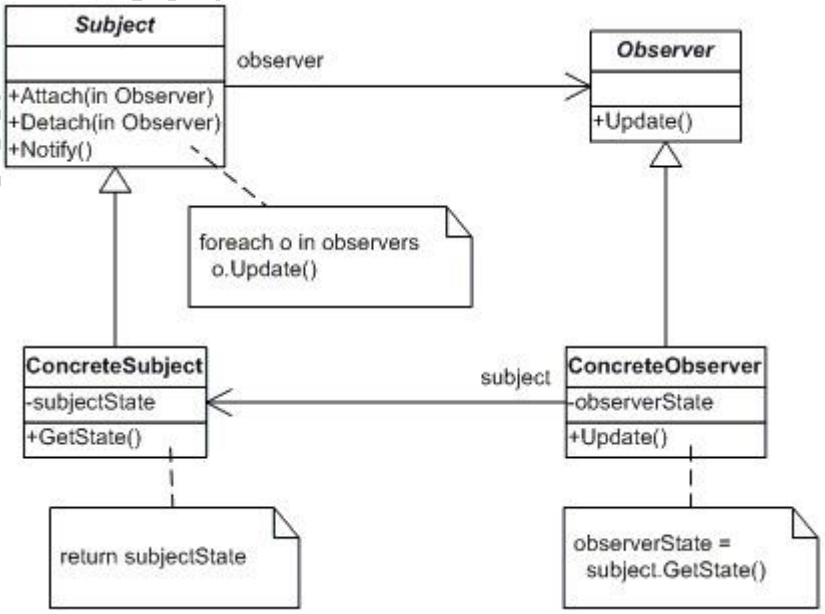


图57. 图 2.1 观察者模式结构图

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 class Solution
{
    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 class Solution
{
    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;
    }
}
```

哔哩哔哩次元游戏人生

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 class Solution
{
    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
        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;
        }
    }
```

哔哩哔哩次元游戏人生

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。

哔哩哔哩次元游戏人生

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");
        }
    }
```

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、 排序算法

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序				$O(1)$	不稳定
快速排序	$O(n\log_{2^n})$	$O(n\log_{2^n})$	$O(n^2)$	$O(\log_{2^n})$	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(1)$	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(n)$	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

$O(1)<O(\log_{2^n})<O(n)<O(n\log_{2^n})<O(n^2)<O(n^3)<O(2^n)$

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

a. 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

b. 定义式

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

c. 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 $X\geqslant0$，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 $X\geqslant0$，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

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。

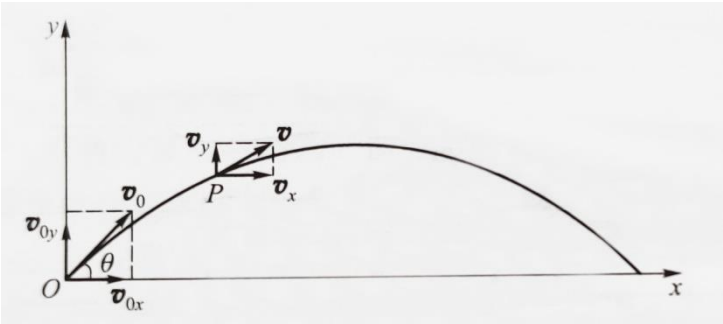


图58. 抛体运动

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。

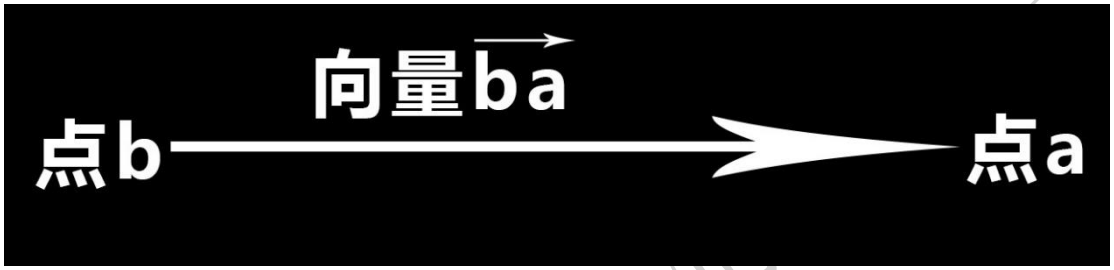


图59. 二维向量

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（点积）

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

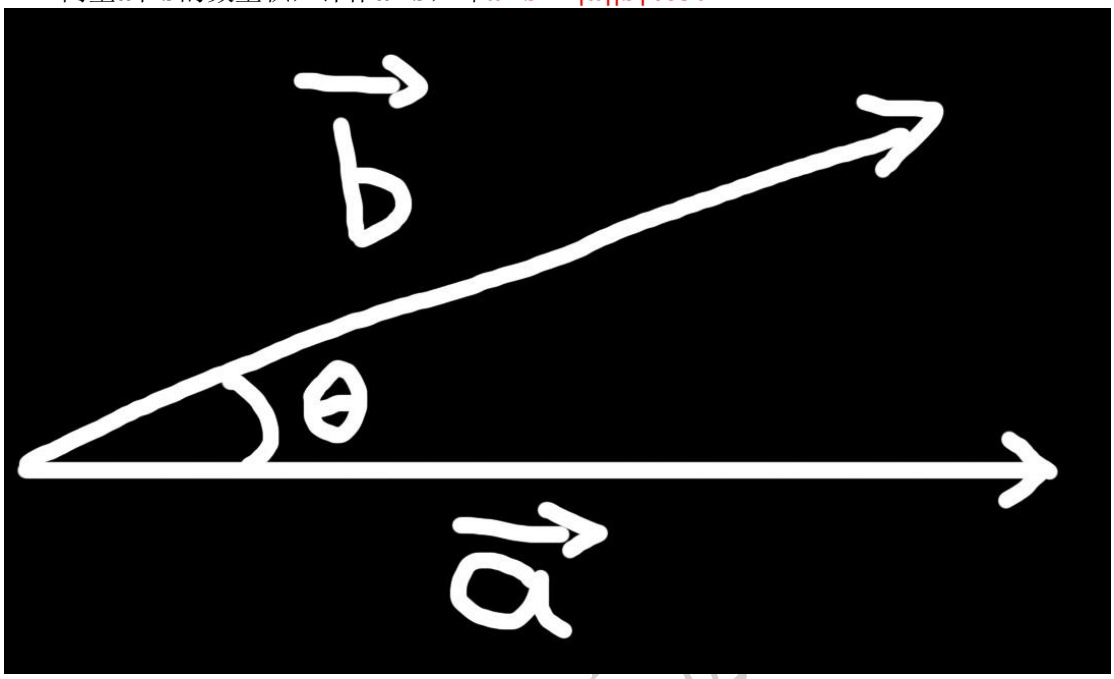


图60. 向量的数量积

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（点积）

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$\vec{\boldsymbol{X}} = \frac{\alpha}{\|\vec{\alpha}\|}$，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：

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

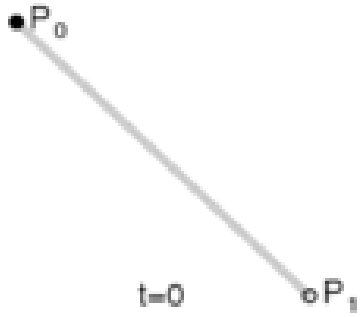


图 3.1 一次贝塞尔曲线 t 在[0,1]之间

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：

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

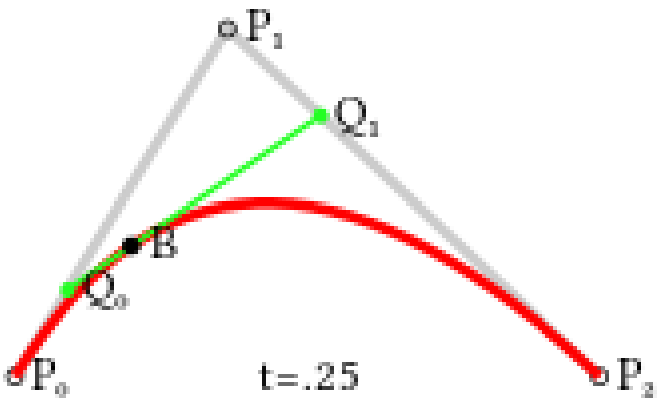


图 3.2 二次贝塞尔曲线的结构 t 在[0,1]之间

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：

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

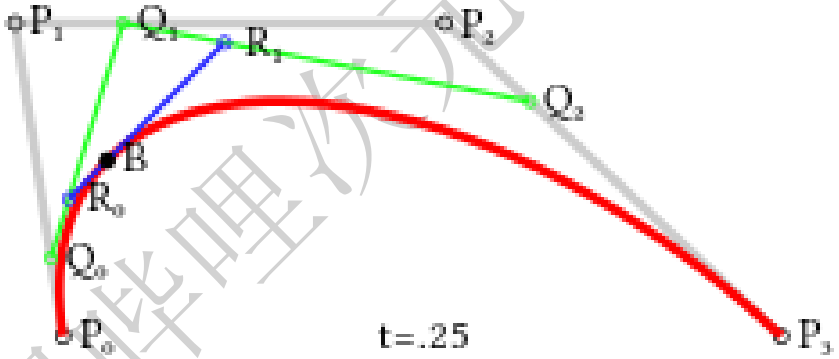


图 3.3 三次贝塞尔曲线的结构 t 在[0,1]之间

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。

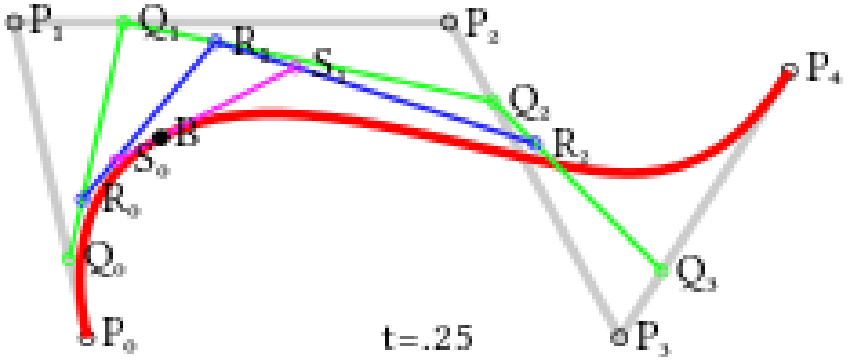


图 3.4 四次贝塞尔曲线的结构 t 在[0,1]之间

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。

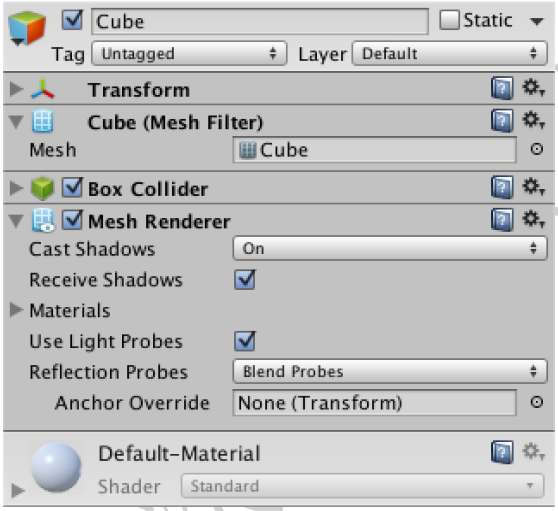


图 3.5 Unity 默认立方体

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。

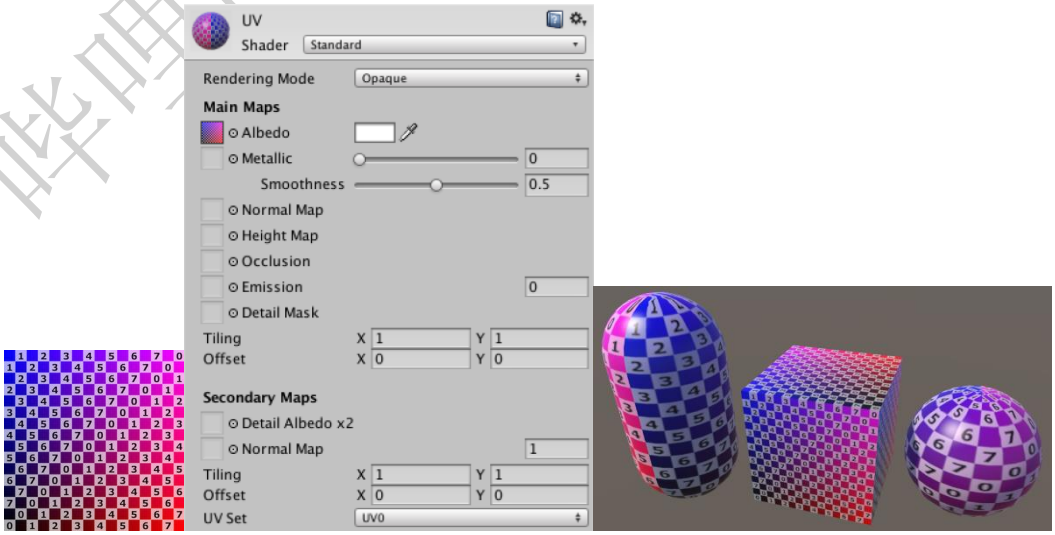


图 3.6 应用于 Unity 的 UV 测试坐标

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。

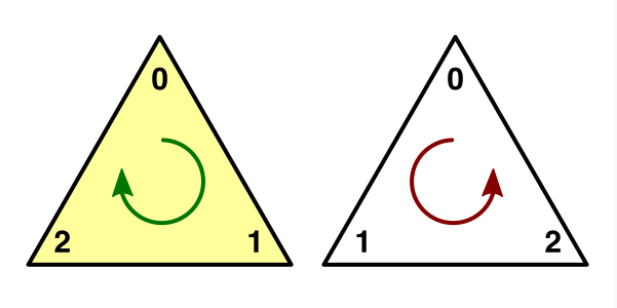


图 3.7 三角形顶点两种遍历情况

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。

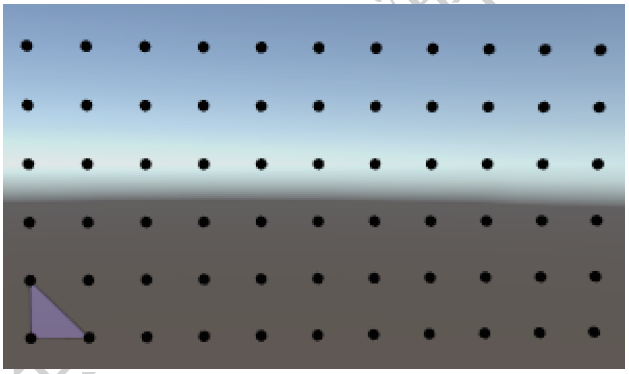
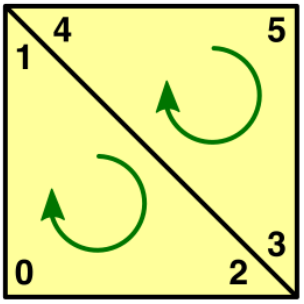


图 3.8 第一个三角形

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

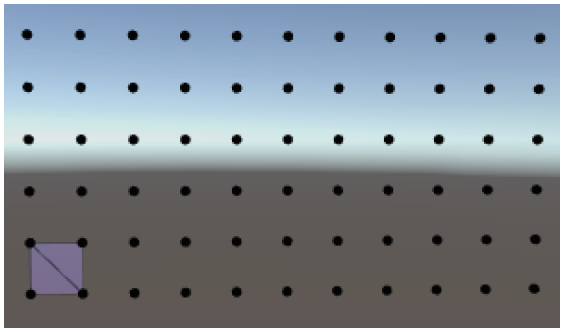


图61. 图 3.9 由两个三角形组成的四边形　　图 3.10 第一个四边形

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。

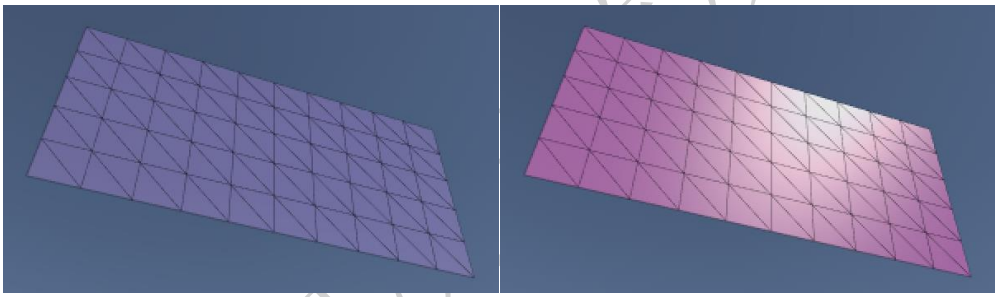


图62.　图 3.11 没有法线与有法线

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。

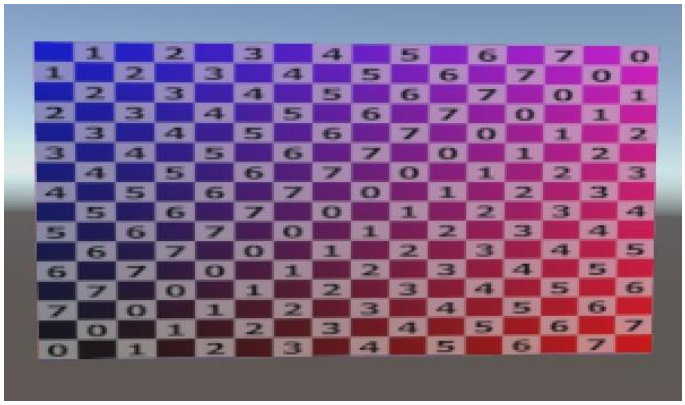


图63.　图 3.12 有 UV 坐标的网格

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。这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这一点。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，在许多艺术风格中都很常见。

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;
}
```

哔哩哔哩次元游戏人生

2） 向量 * 矩阵

```
void Unity_Multiply_float4_float4x4(float4 A, float4x4 B, out float4 Out)
{
    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

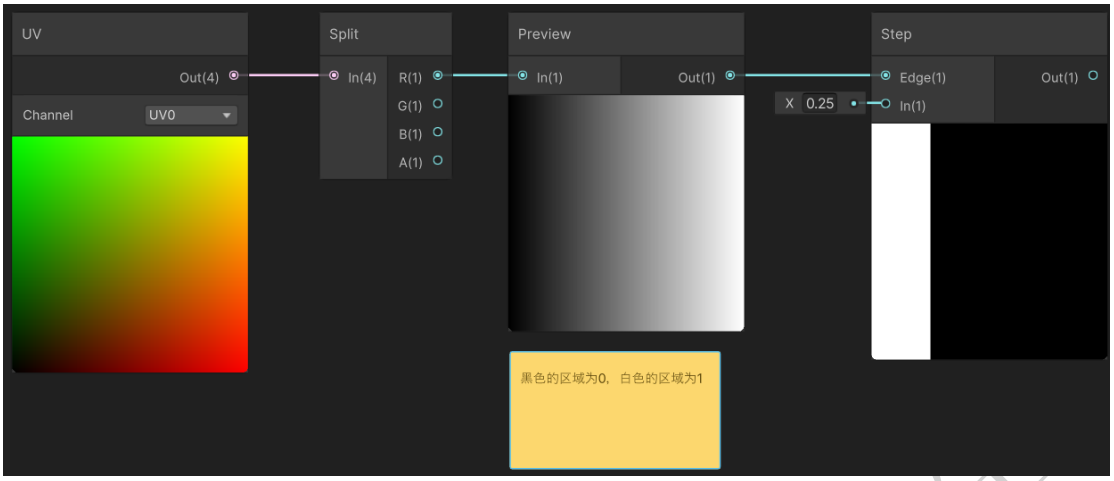


图64.　　图 4.1 Step 节点可视化

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。

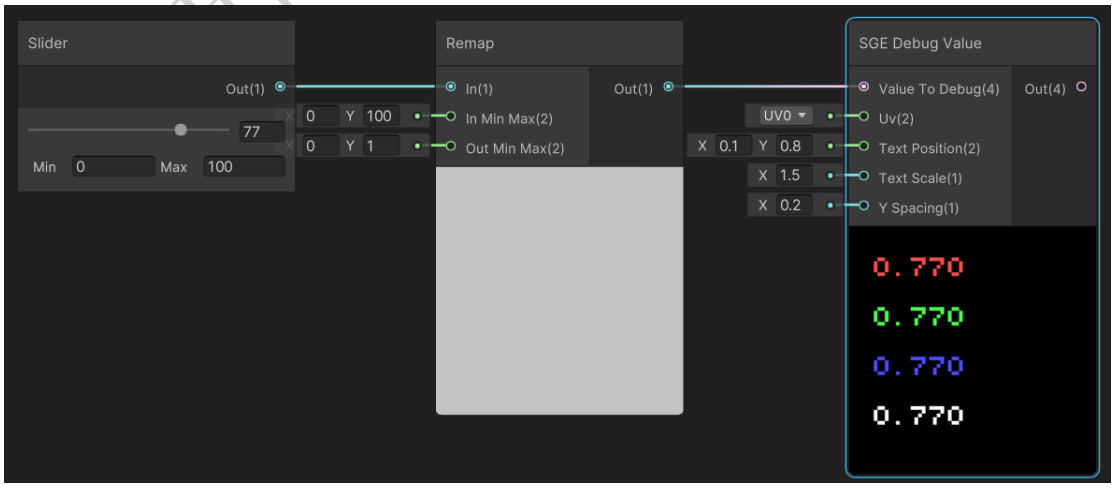


图65. 图 6.1 Remap Node 示例

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66. 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。

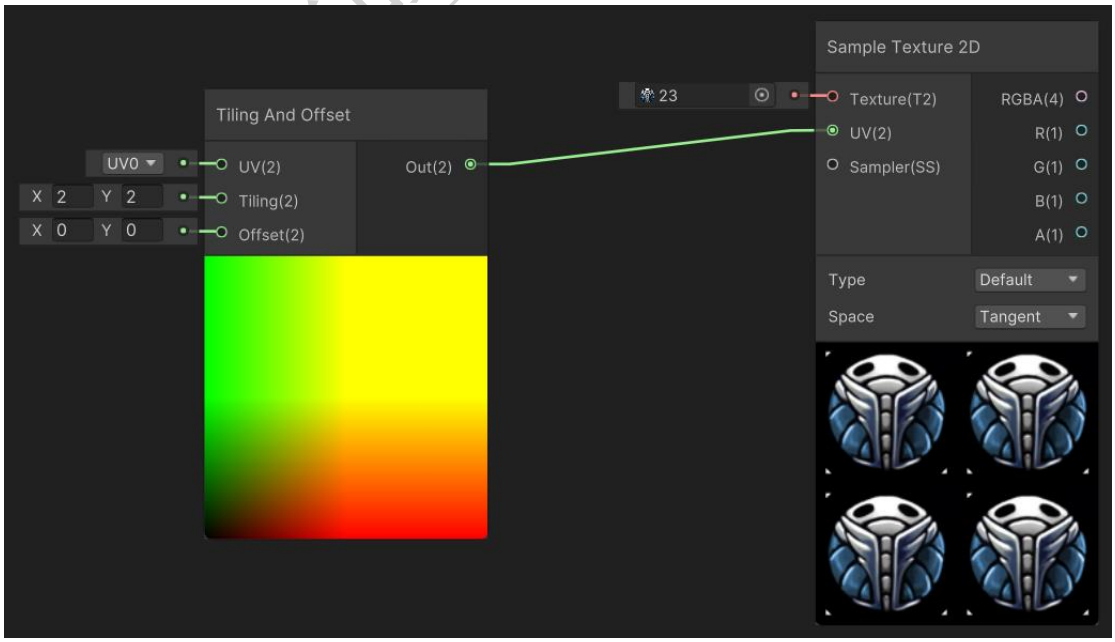


图67. 图 9.1 将 Tiling 的分量都设置为 2

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

图68.　　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

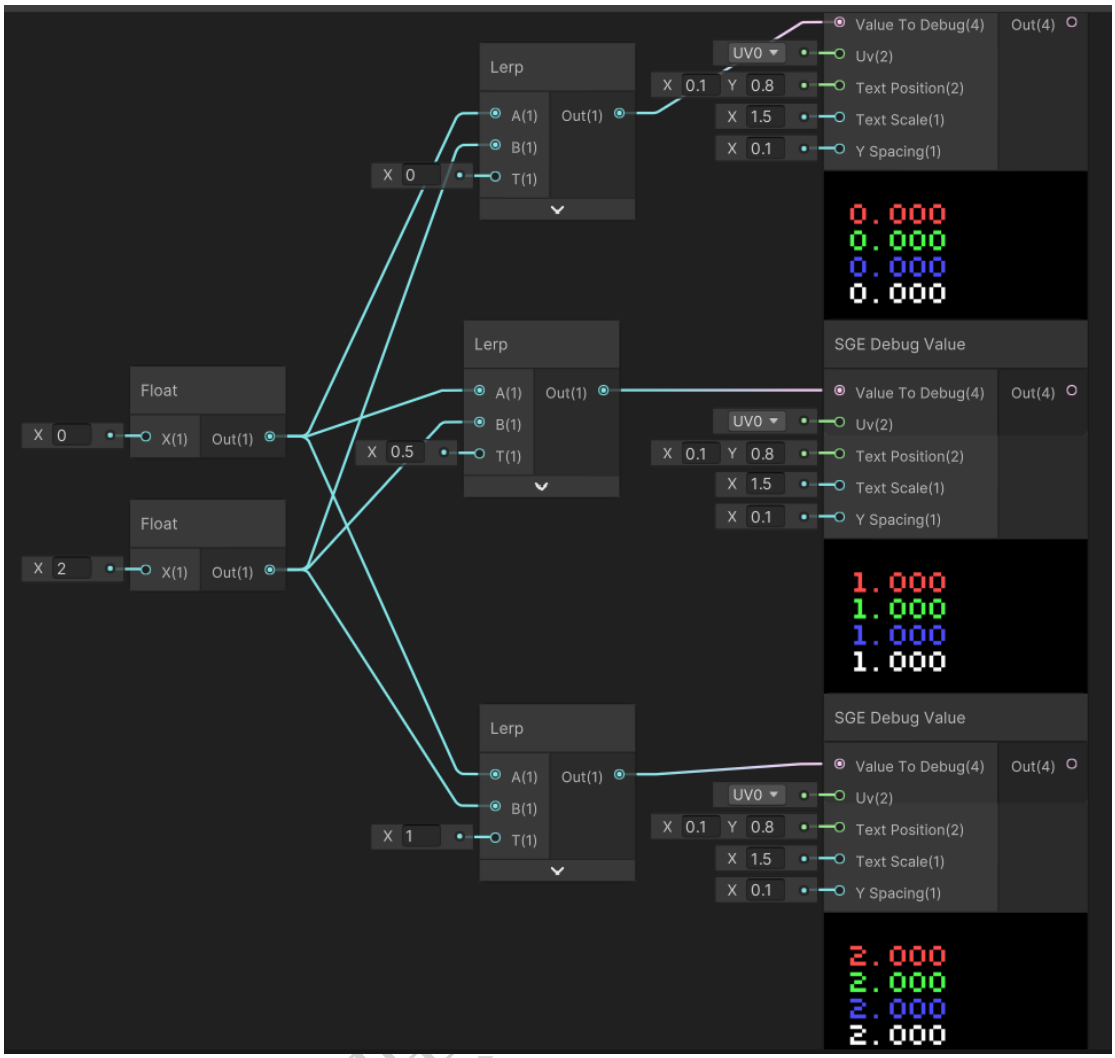


图69. 线性插值图示

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。

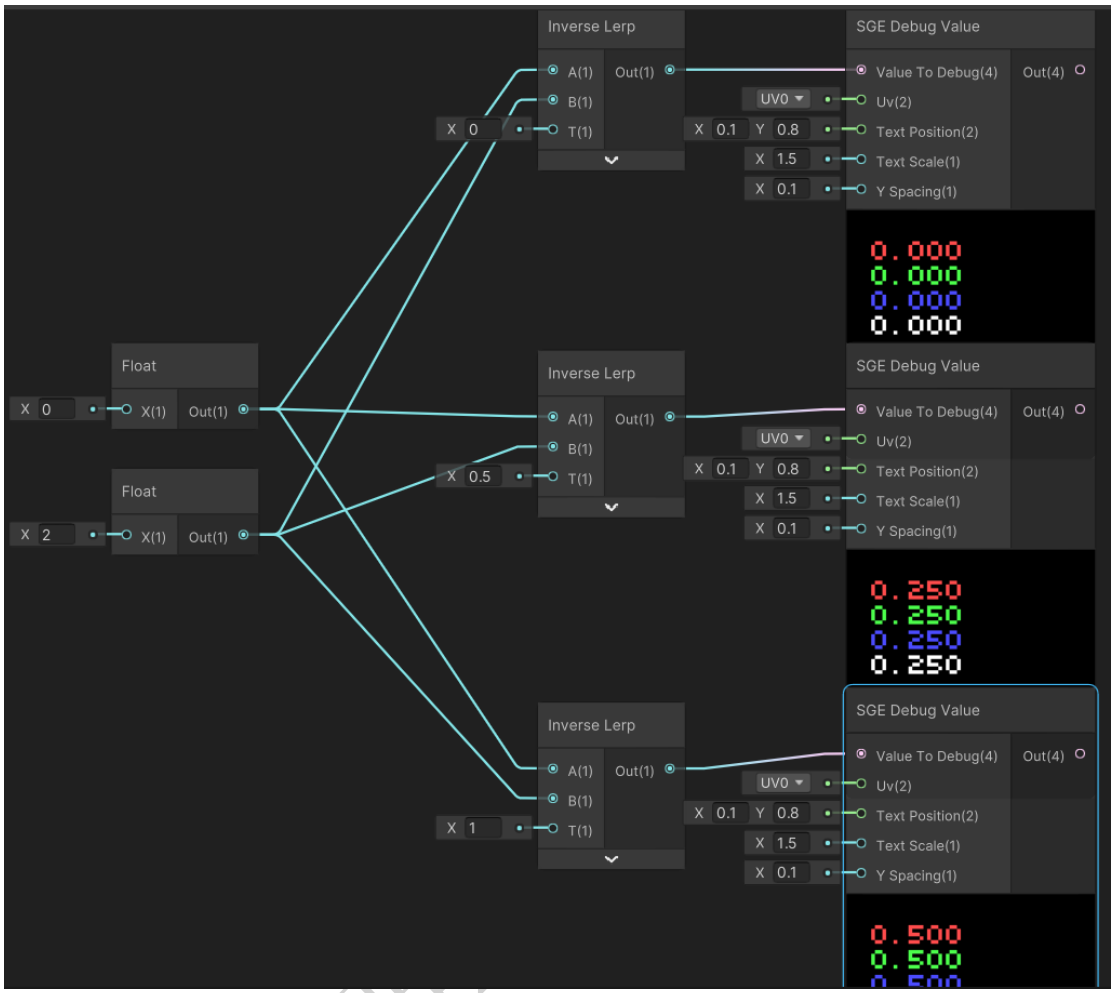


图70. 反插值图示

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。

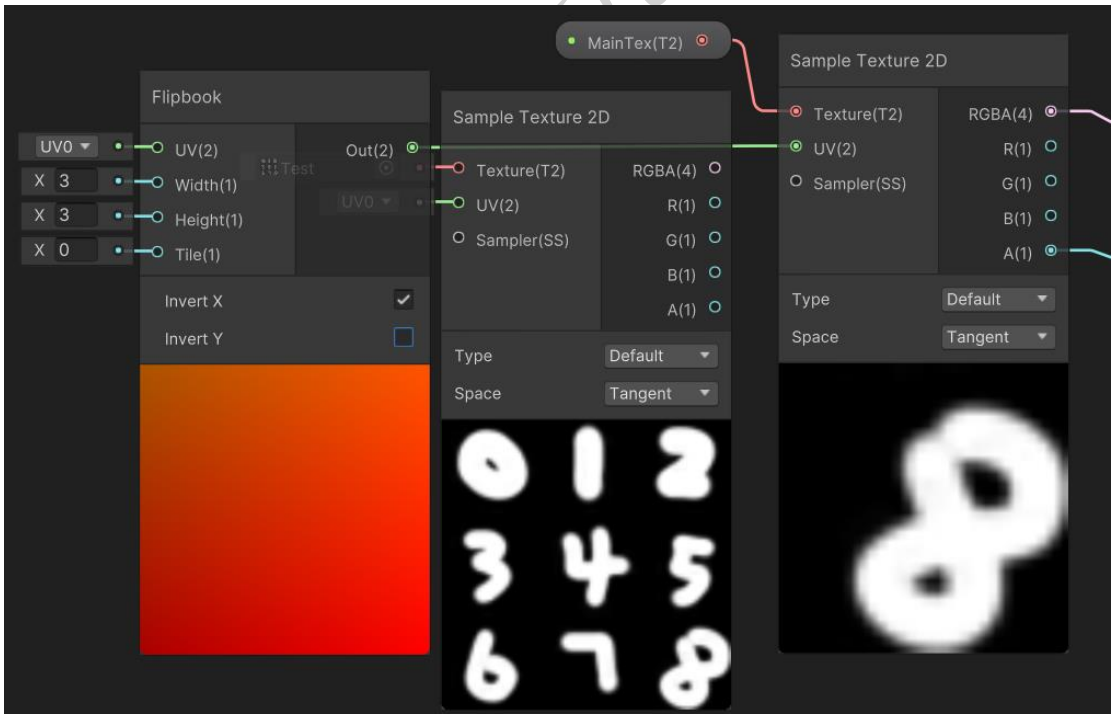


图71. 启用 X 关闭 Y

2） 启用 X 和 Y

图片最上层为第一行，最右边为第一列。

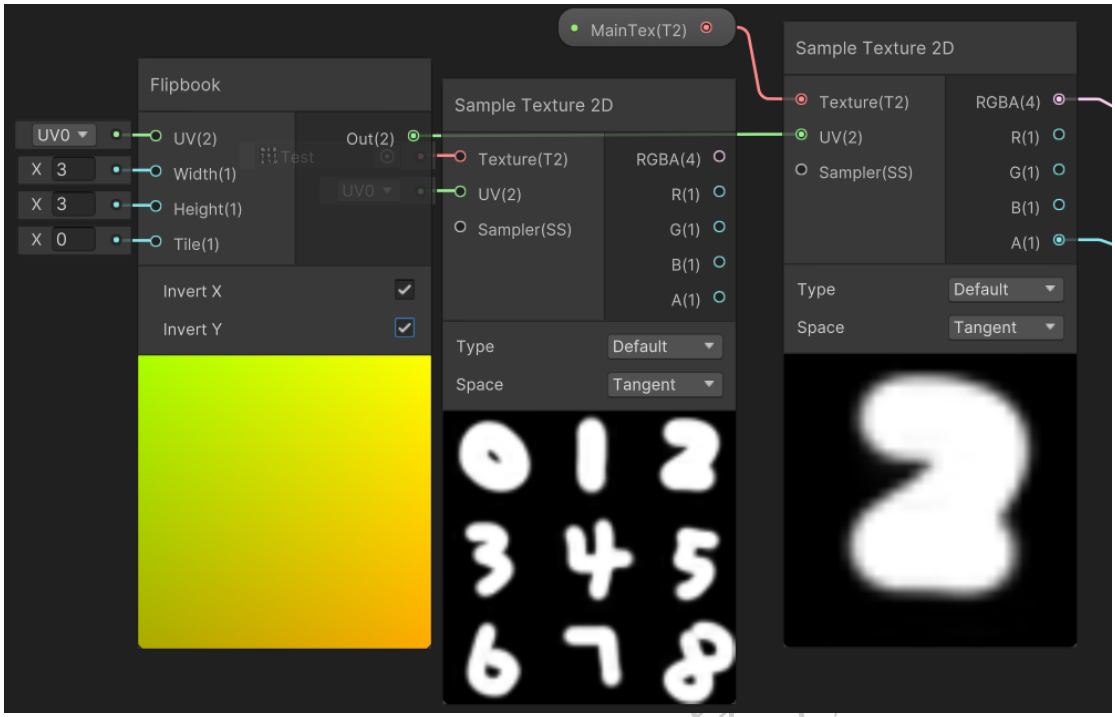


图72.　　启用 X 和 Y

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。

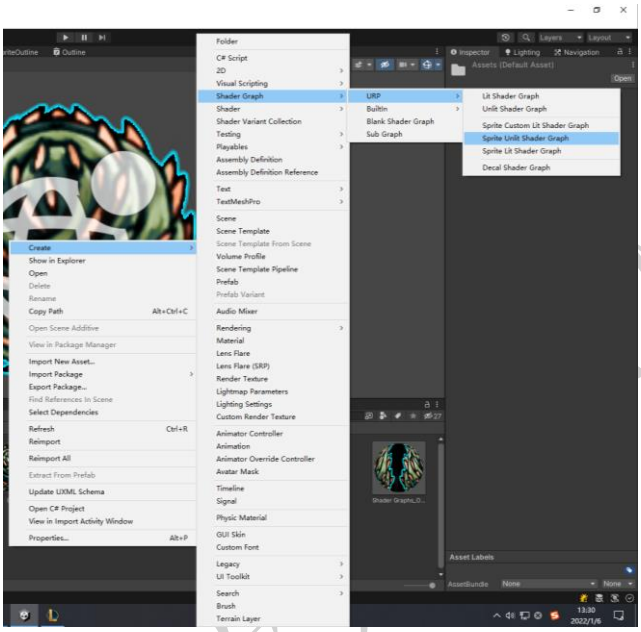


图73. 图 1.1 创建 Sprite Unlit Shader Graph

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74. 图 1.2 Graph Inspector 面板按钮

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

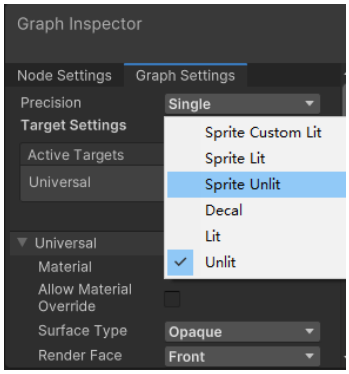


图75. 图 1.3 修改 Material

[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）。

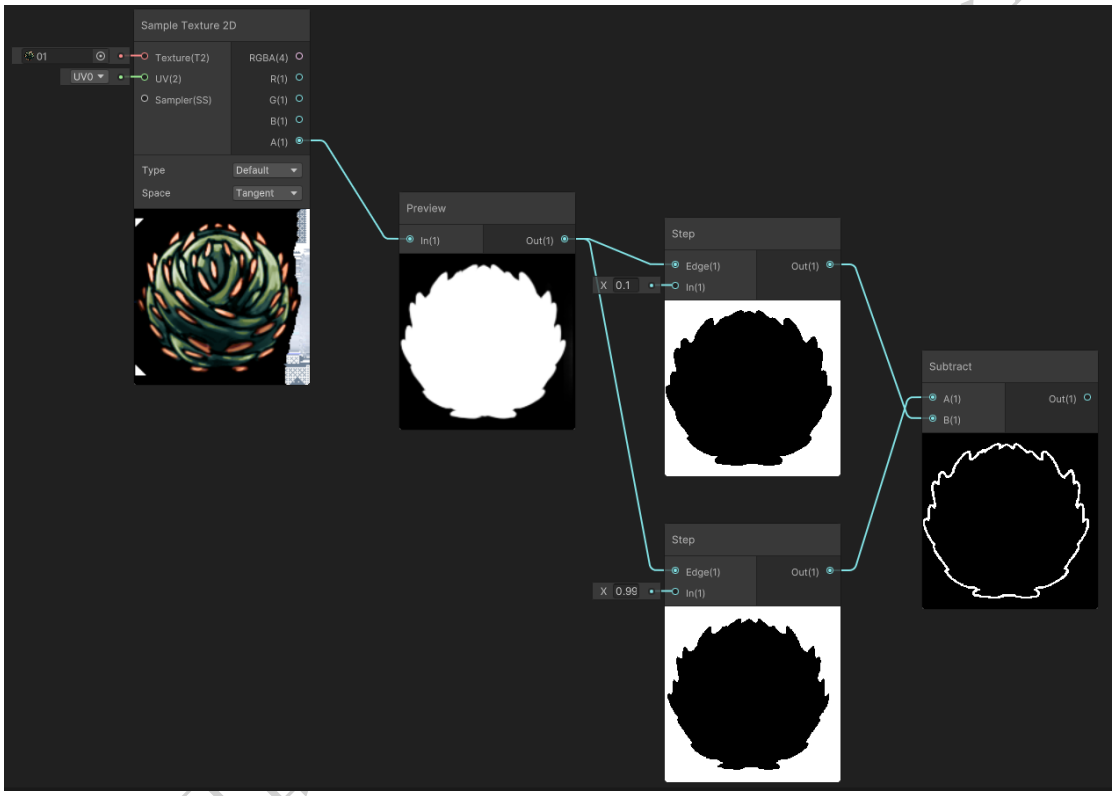


图76.　　图 1.4 描边轮廓

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。

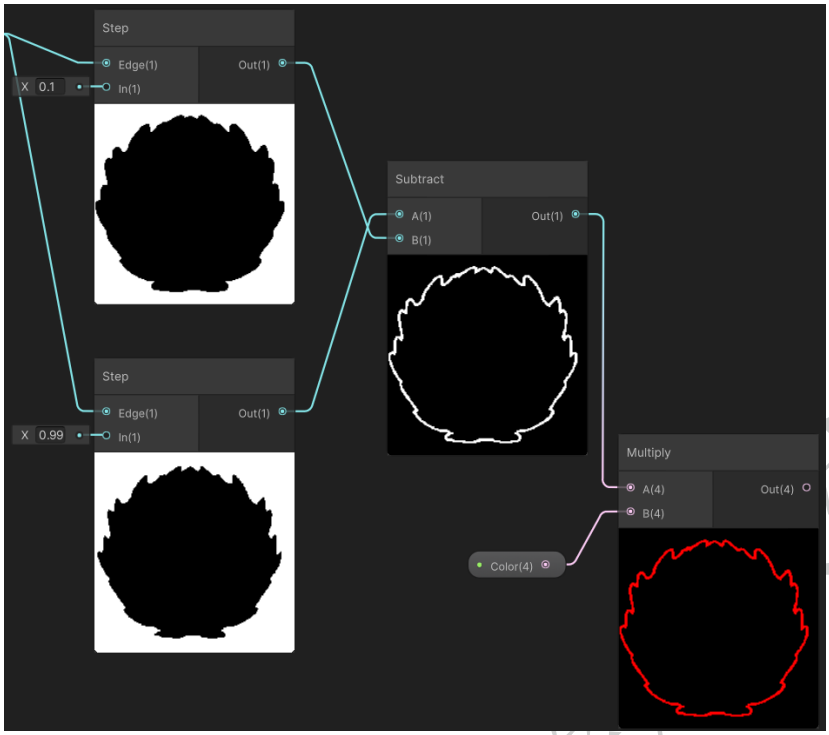
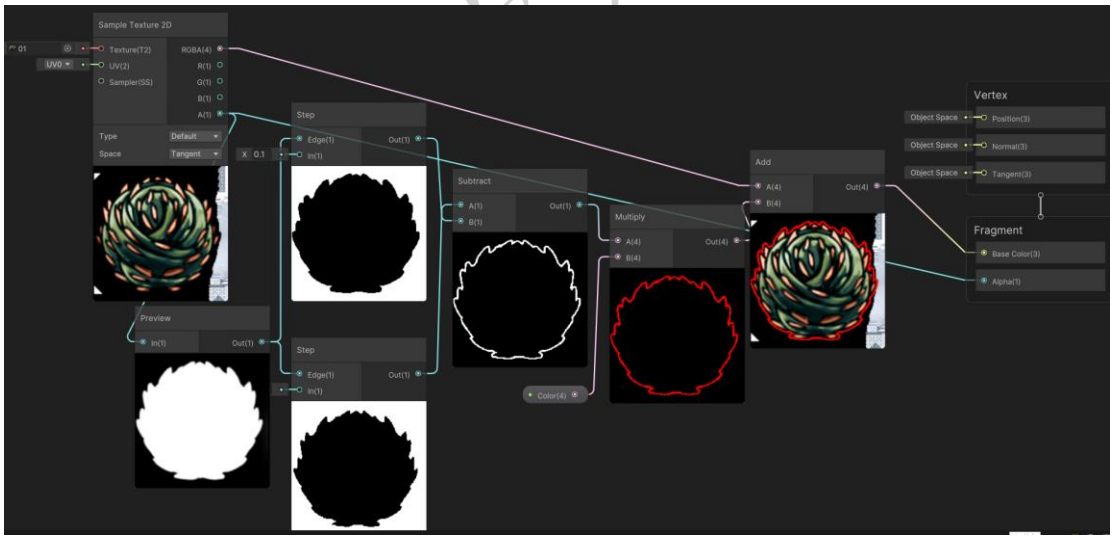


图77.　　图 1.5 修改描边轮廓

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。

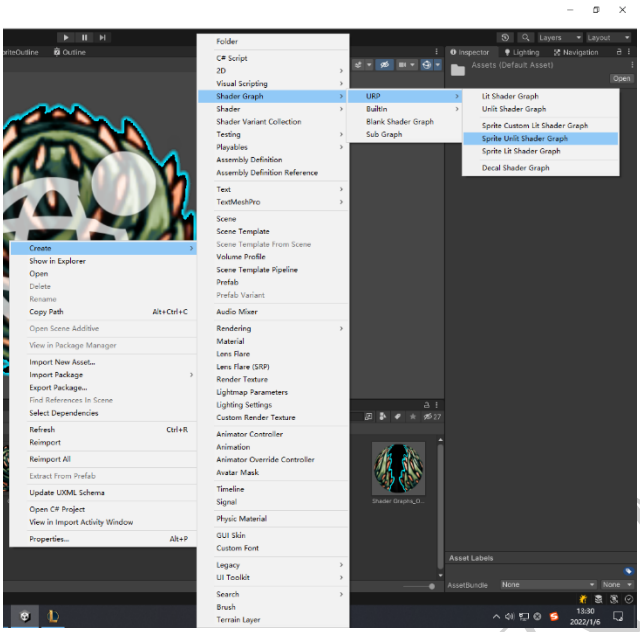


图78.　　图 2.1 创建 Sprite Unlit Shader Graph

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79.　　图 2.2 Graph Inspector 面板按钮

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

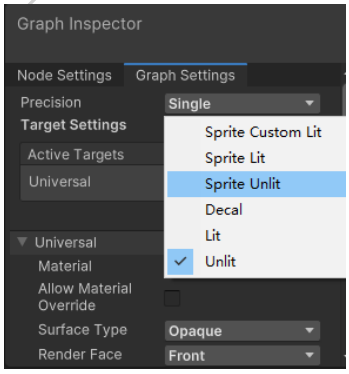


图80.　　图 2.3 修改 Material

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。

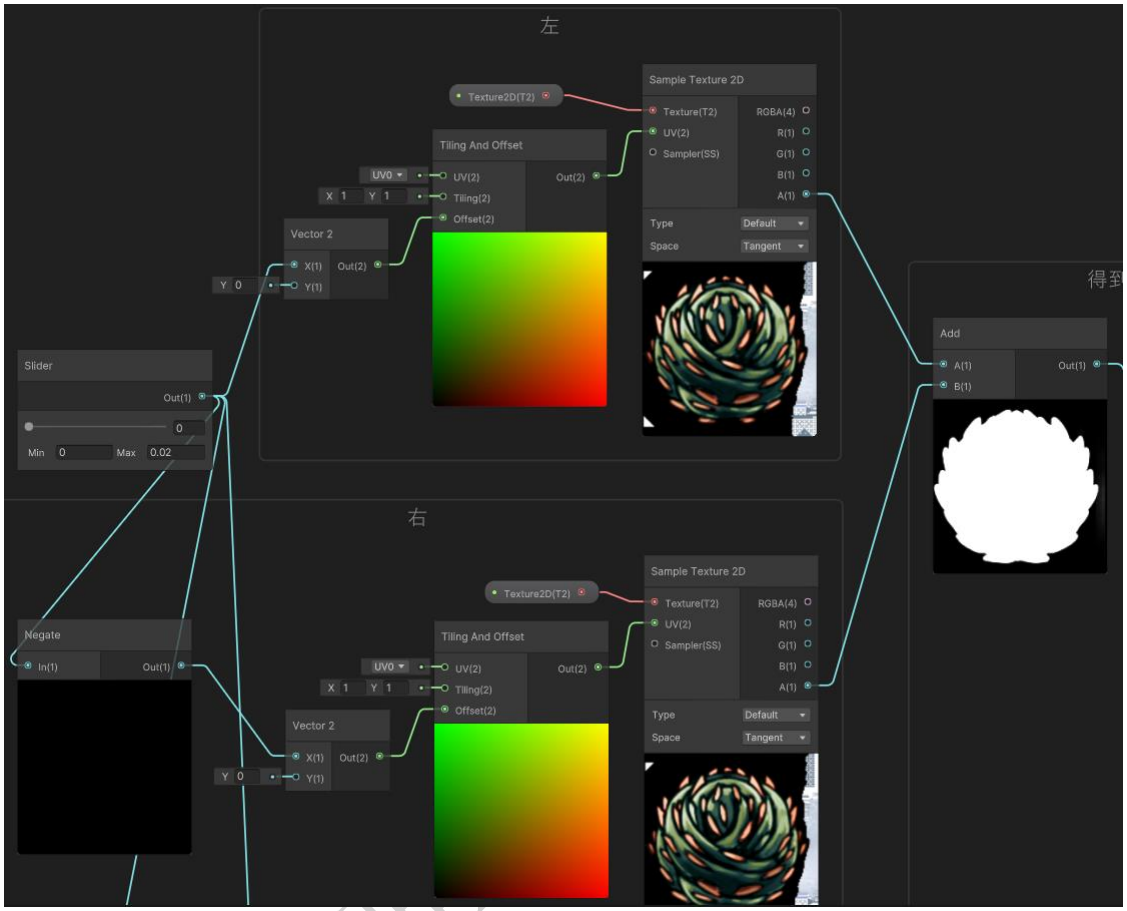


图81.　　图 2.4 得到左右偏移图像

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。

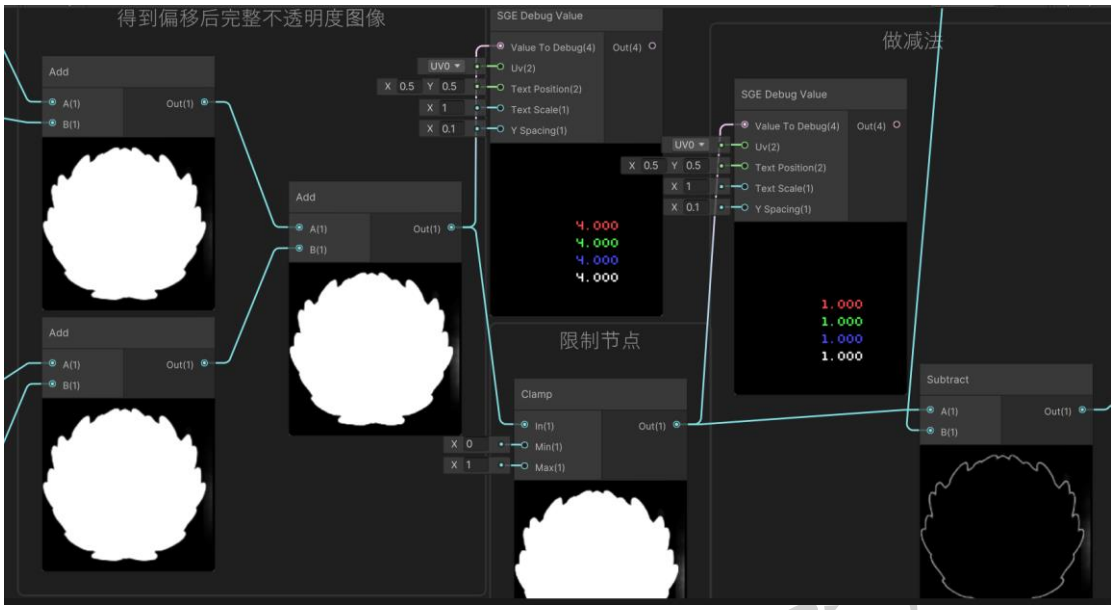


图82.　　图 2.5 通过 Clamp 节点得到正确描边

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。

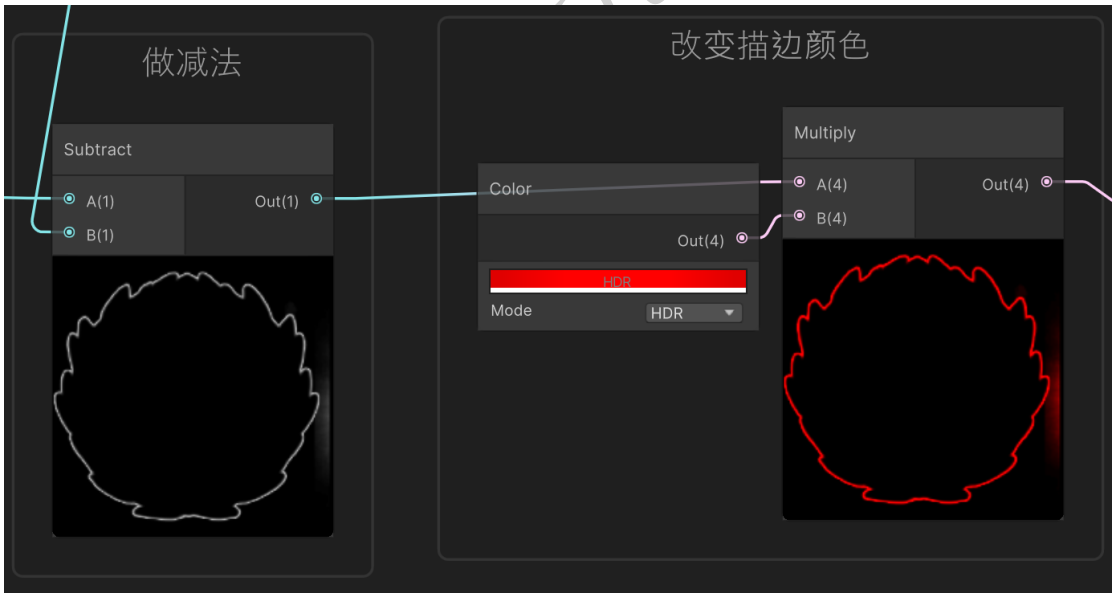


图83.　　图 2.6 修改描边区域的颜色

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。

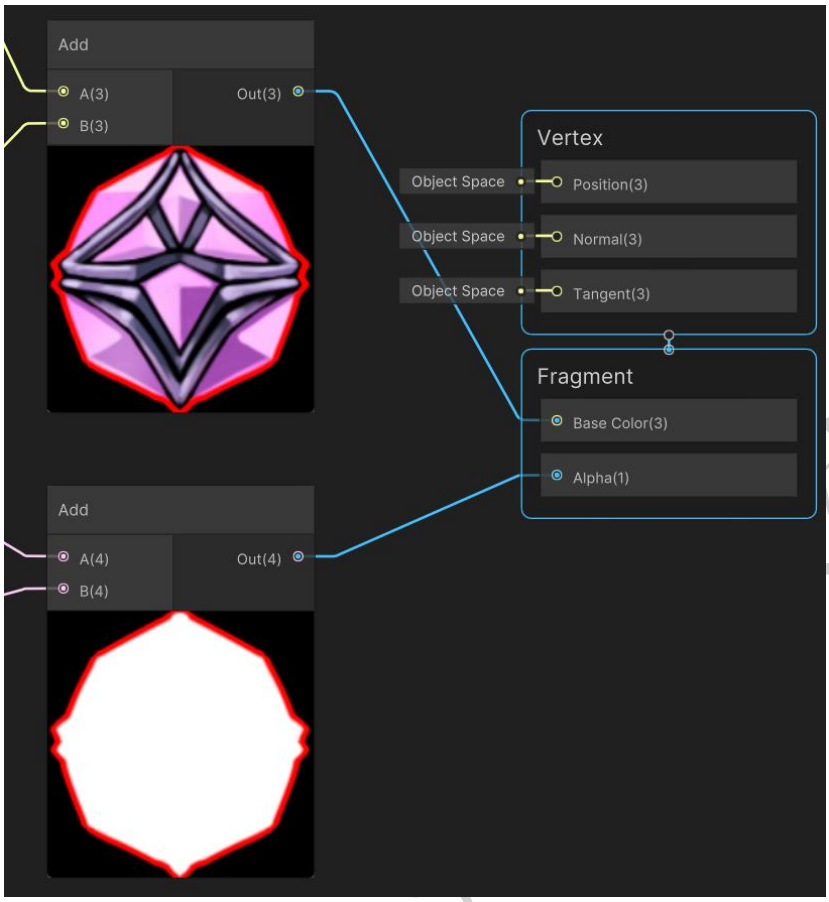


图84.　　图 2.7 输出图像

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。

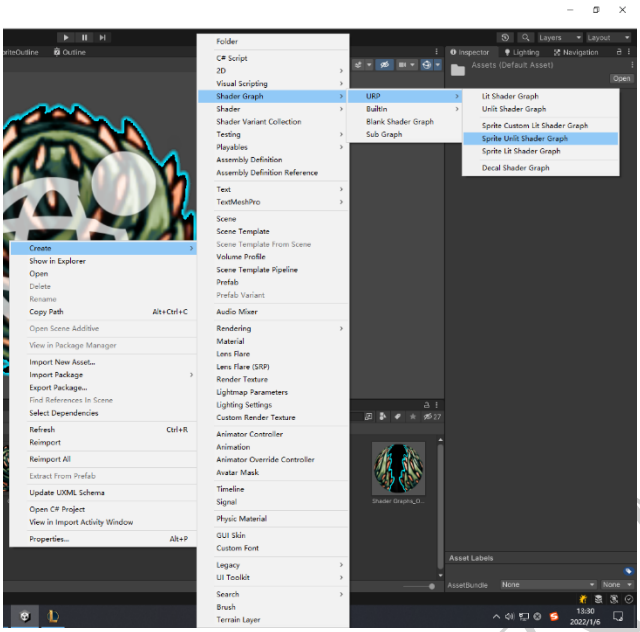


图85.　　图 3.1 创建 Sprite Unlit Shader Graph

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86.　　图 3.2 Graph Inspector 面板按钮

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

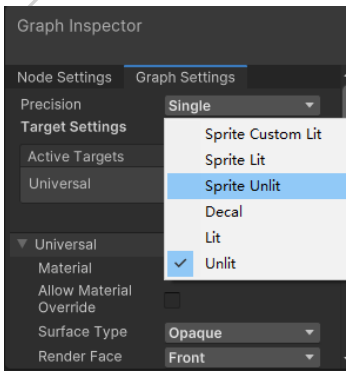


图87.　　图 3.3 修改 Material

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

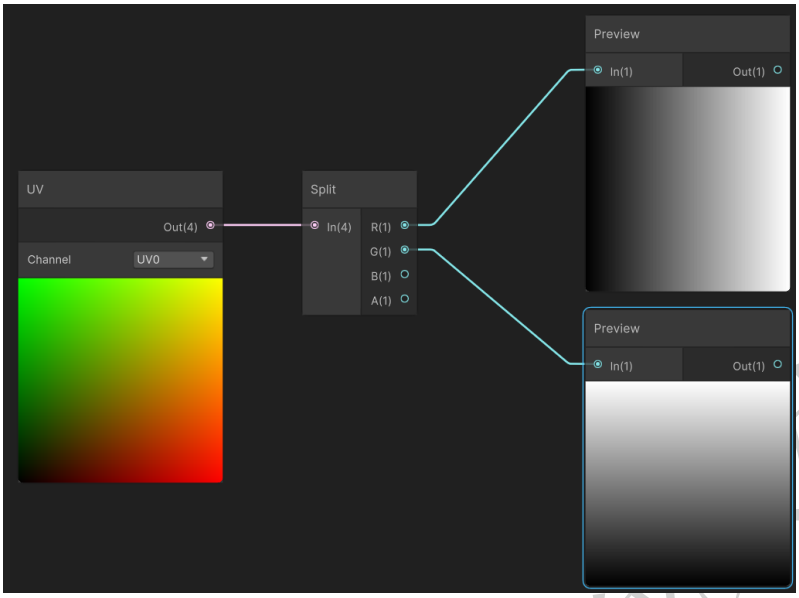


图88.　　图 3.4 预览 UV 的 R 通道和 G 通道

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。

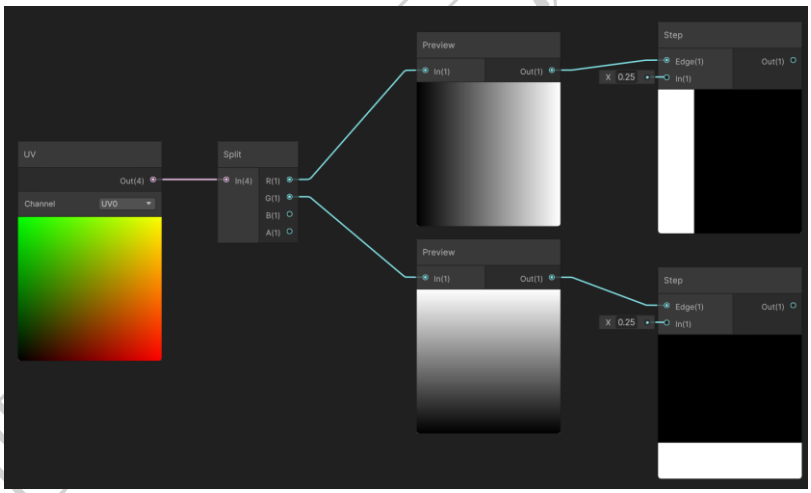


图89.　　图 3.5 用 Step 节点对 R 通道和 G 通道进行切分

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；

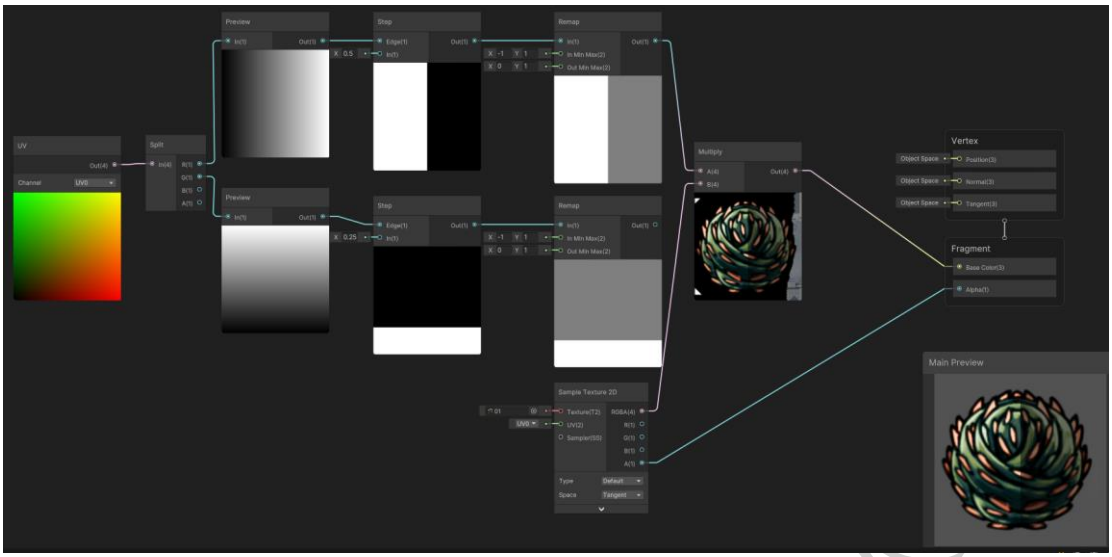


图90.　　图 3.6 重新映射后的最终图像

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91.　　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。

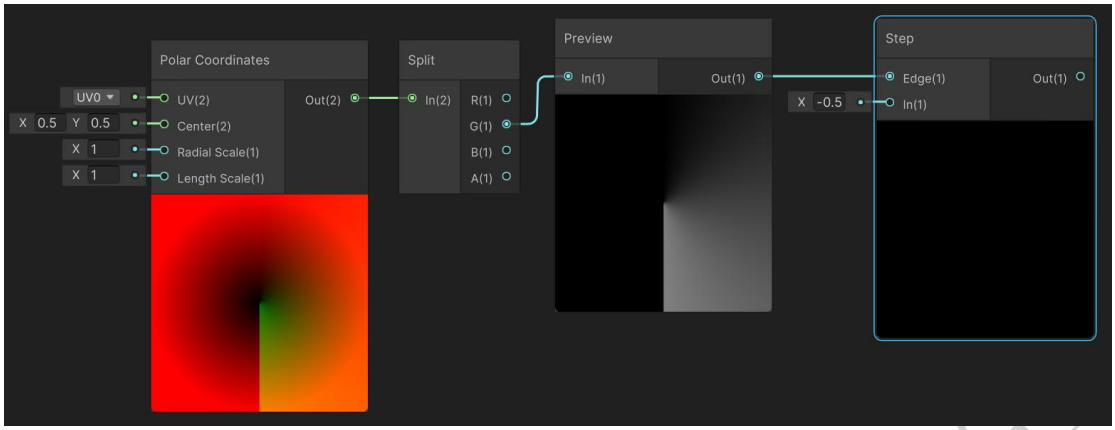


图92.　　图 3.8 使用极坐标实现冷却过渡

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。

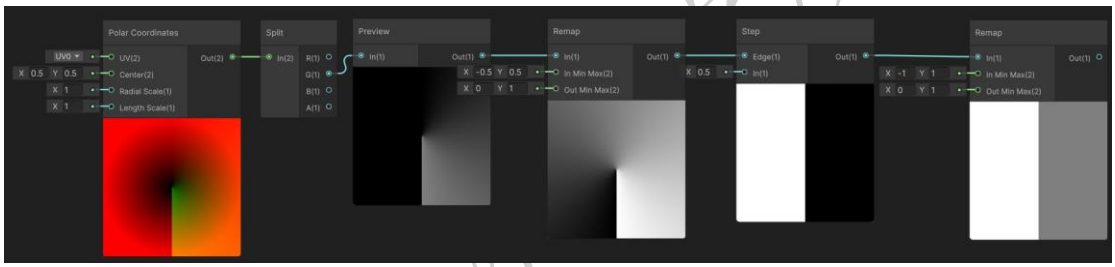


图93.　　图 3.9 使用 Remap 节点重新映射 G 通道值

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94.　　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。

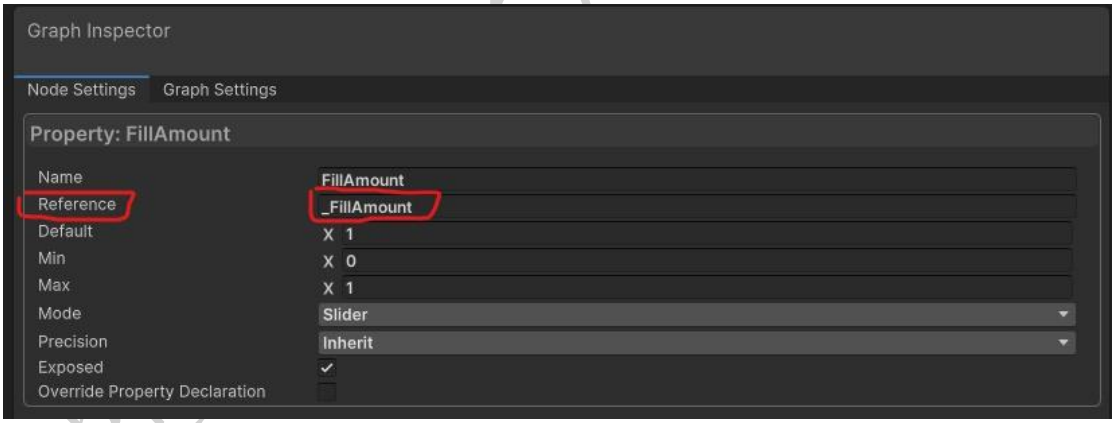


图95.　　图 3.11 Graph Inspector 界面的 Reference

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96.　　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。

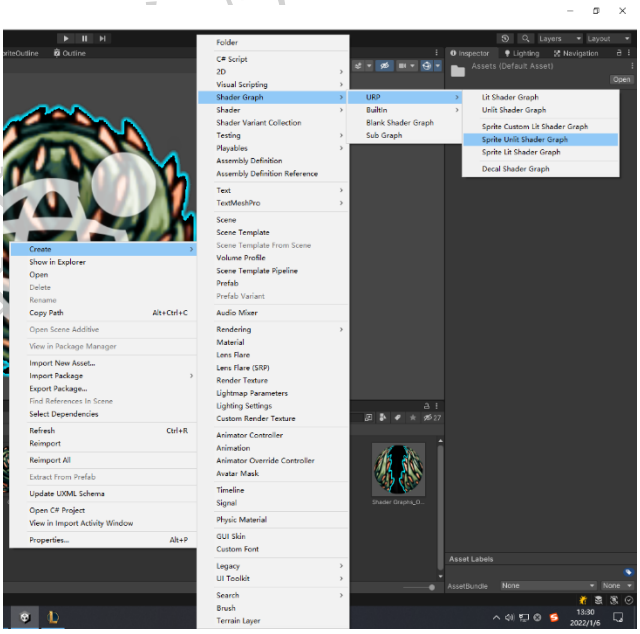


图97.　　图 4.1 创建 Sprite Unlit Shader Graph

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98.　　图 4.2 Graph Inspector 面板按钮

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

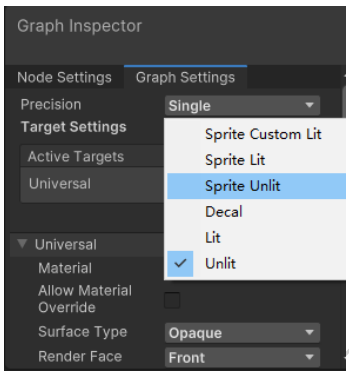


图99.　　图 4.3 修改 Material

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。

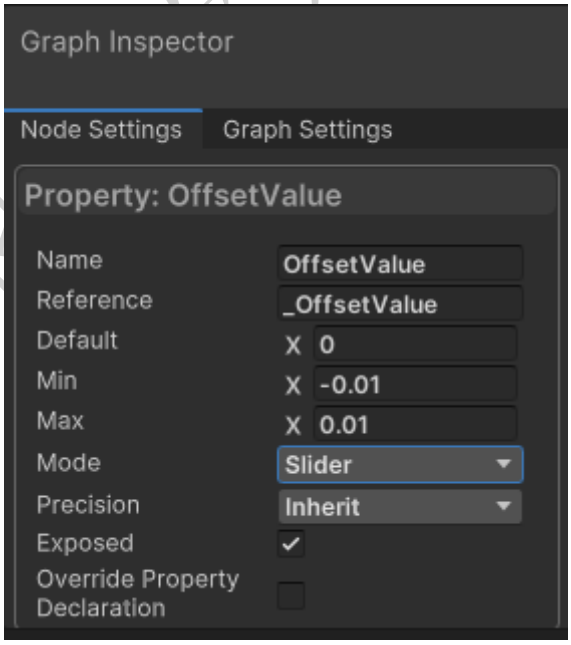


图100.　　在 Graph Inspector 中修改参数模式

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

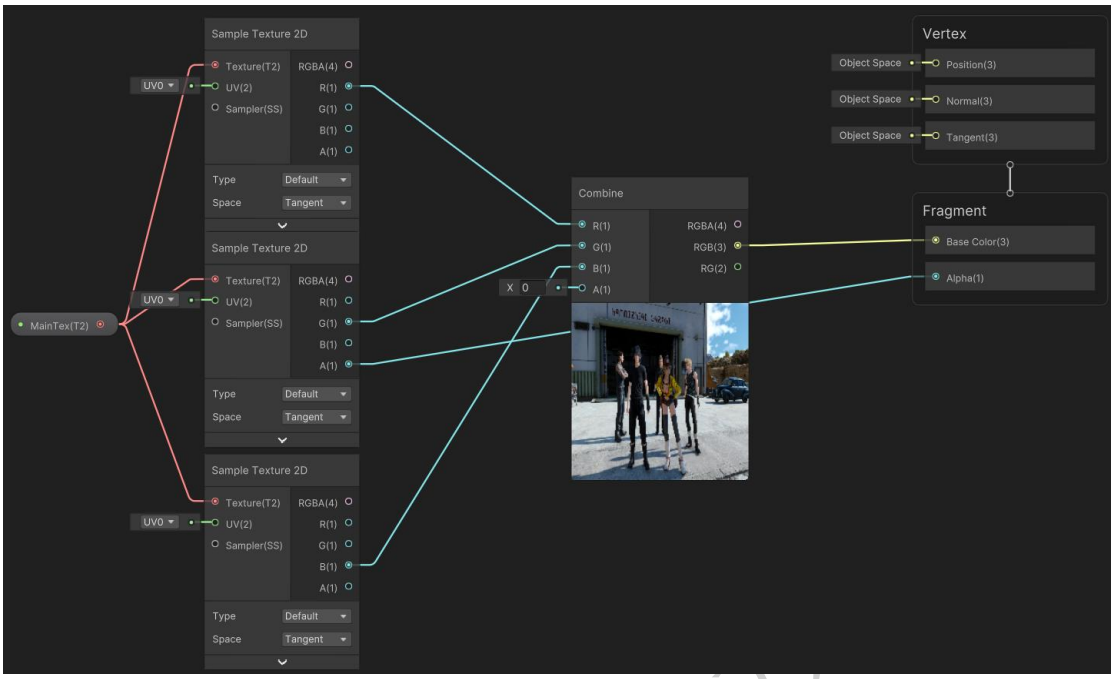


图101.　创建 RGB 分离的采样节点

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

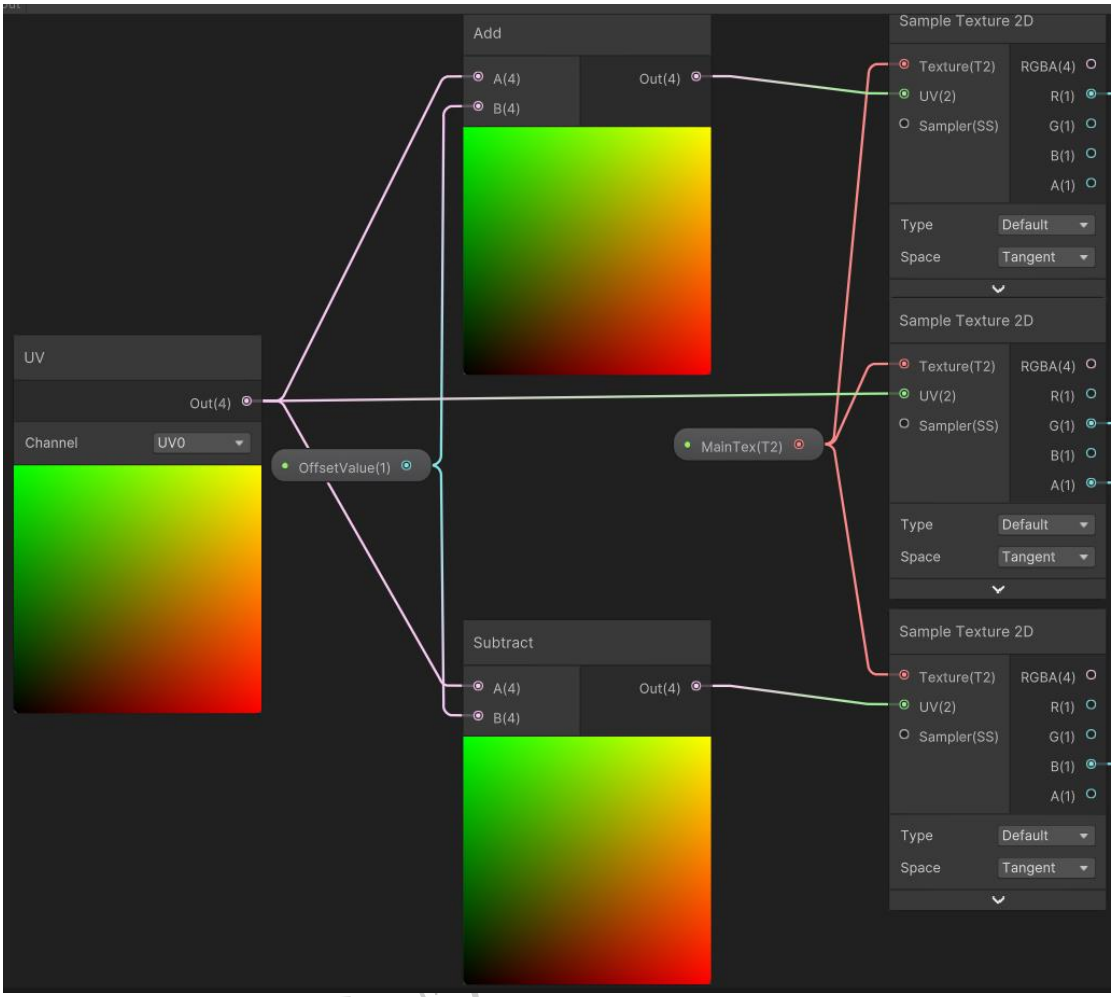


图102.　图 4.6 RGB 分离 UV 偏移

图103.　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。

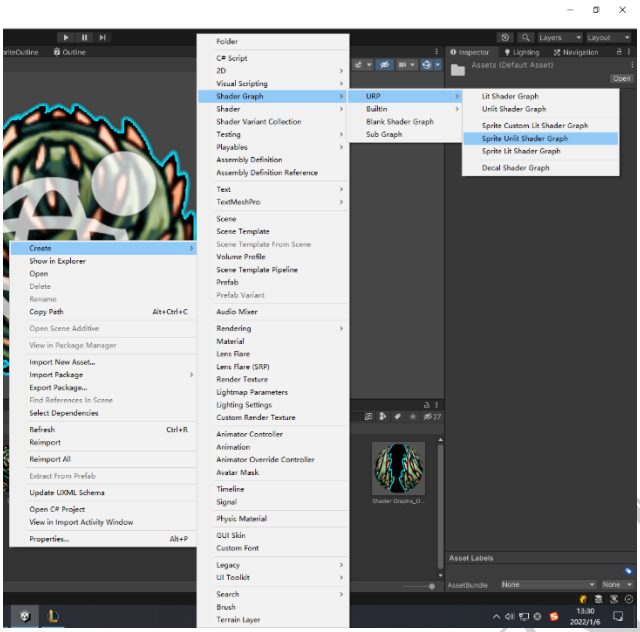


图104.　图 5.1 创建 Sprite Unlit Shader Graph

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105.　图 5.2 Graph Inspector 面板按钮

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

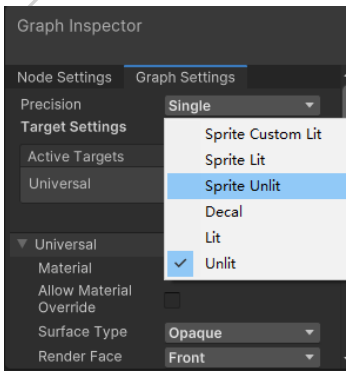


图106.　图 5.3 修改 Material

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。

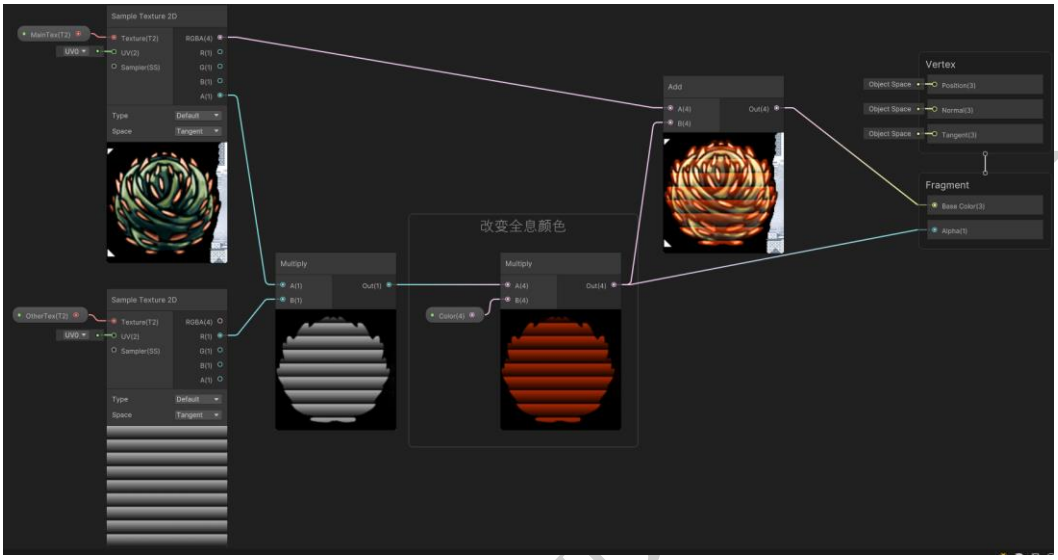


图107.　　图 5.4 创建带全息效果的图像

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。

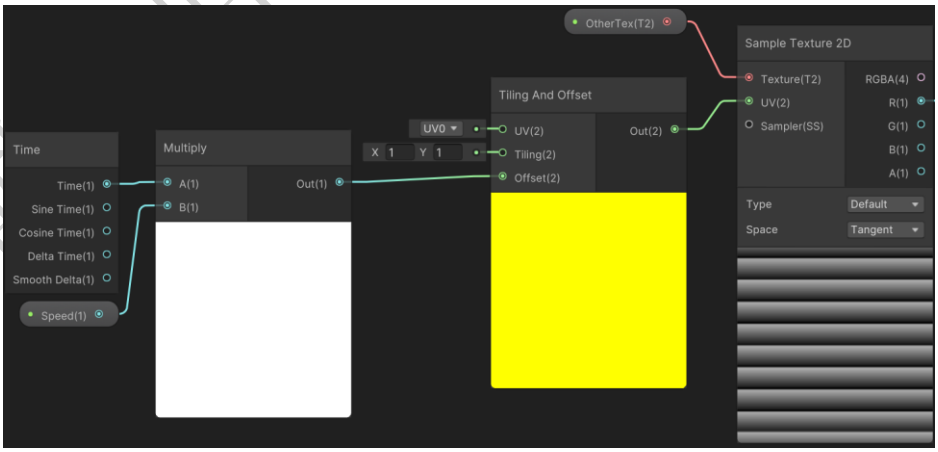


图108.　　图 5.5 让全息贴图流动

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。

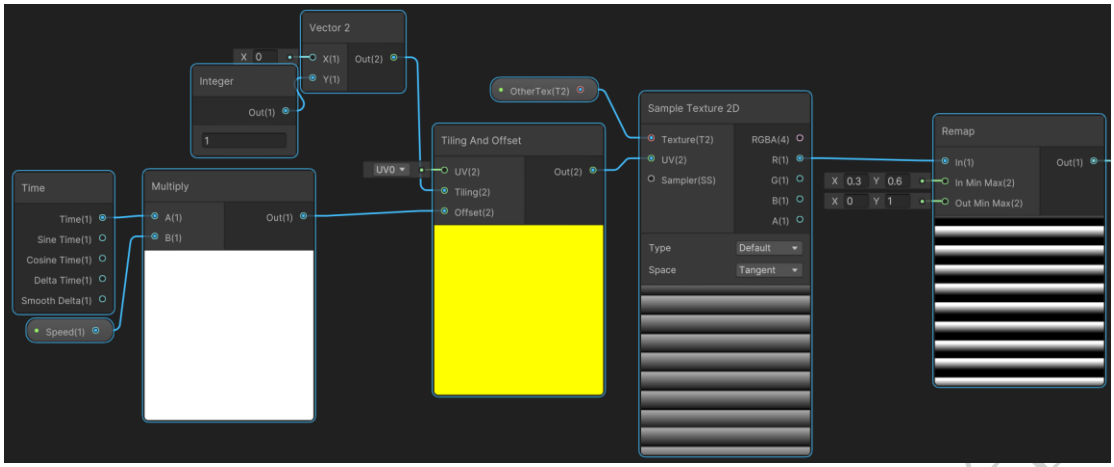


图109.　　图 5.6 调节全息范围、强度和密度

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110.　　图 5.6 使用 Clamp 节点对值进行限制

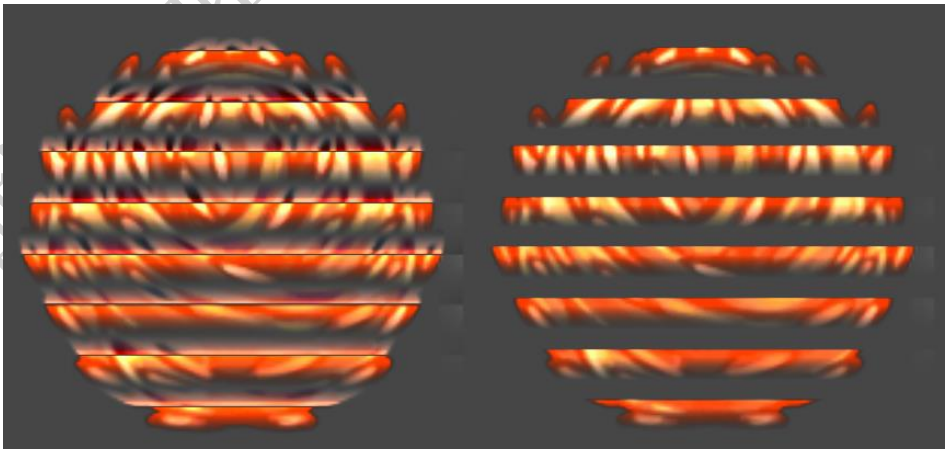


图111.　　图 5.7 调节前（左）和调节后（右）对比

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。

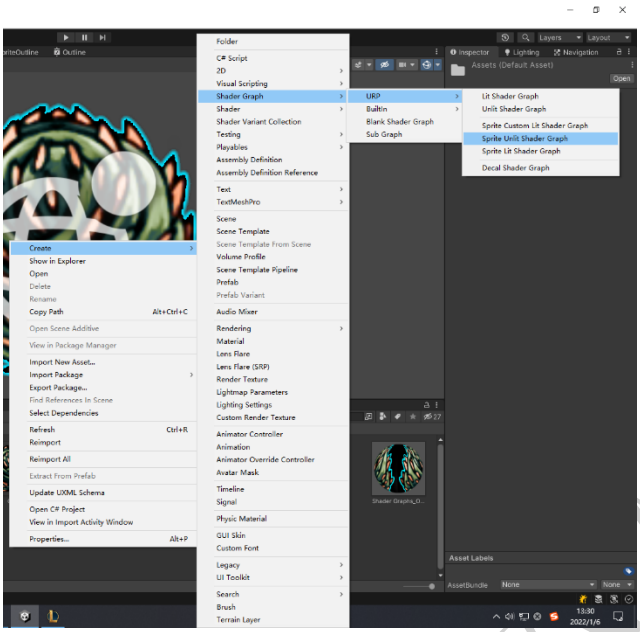


图112.　　图 6.1 创建 Sprite Unlit Shader Graph

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113.　　图 6.2 Graph Inspector 面板按钮

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

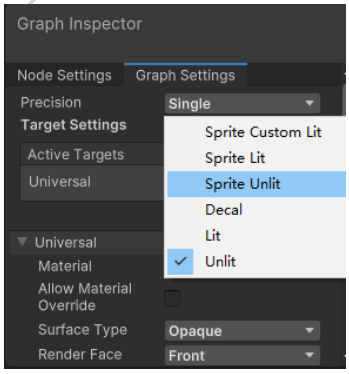


图114.　　图 6.3 修改 Material

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

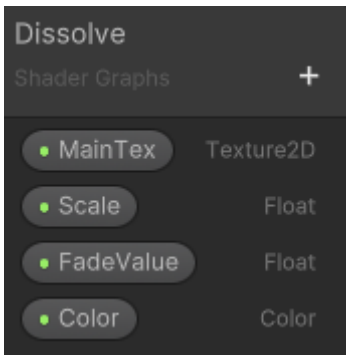


图115.　　图 6.4 可外部访问的参数

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。

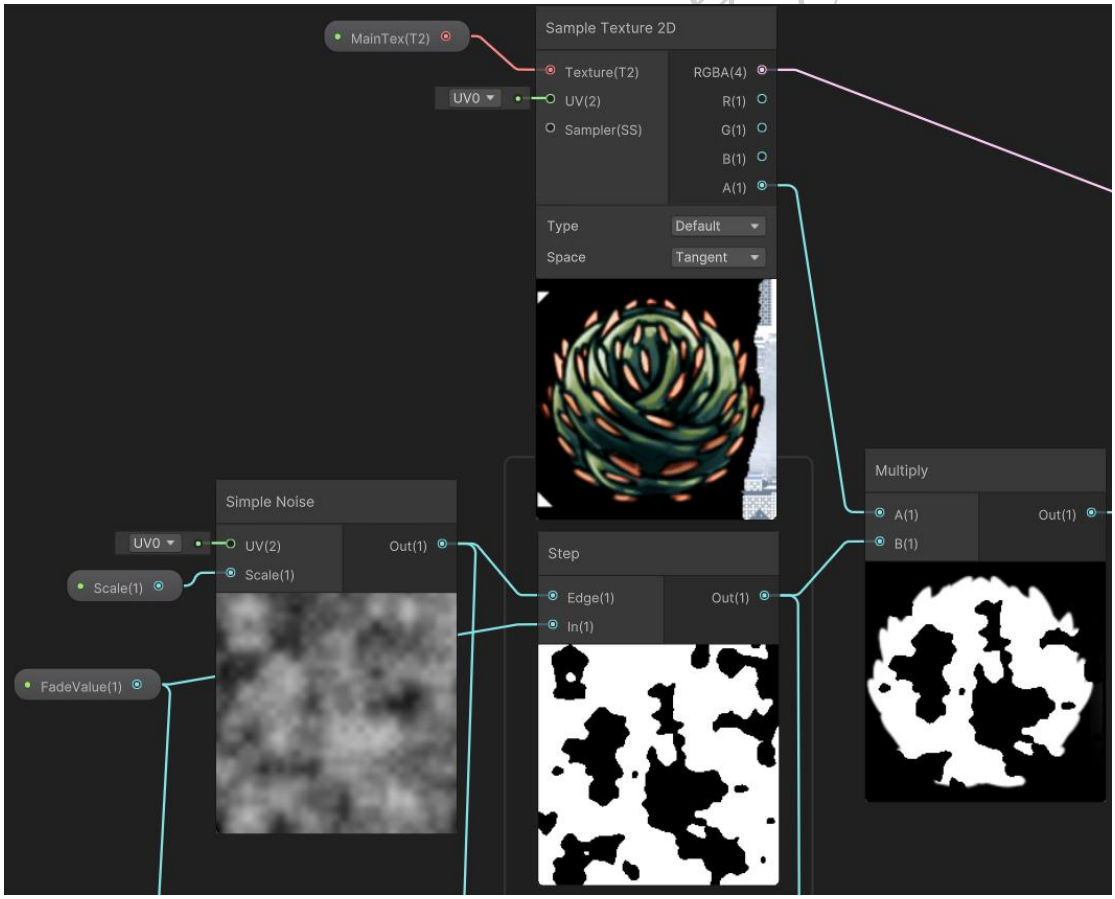


图116.　　图 6.5 当前溶解度对应的溶解效果

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。

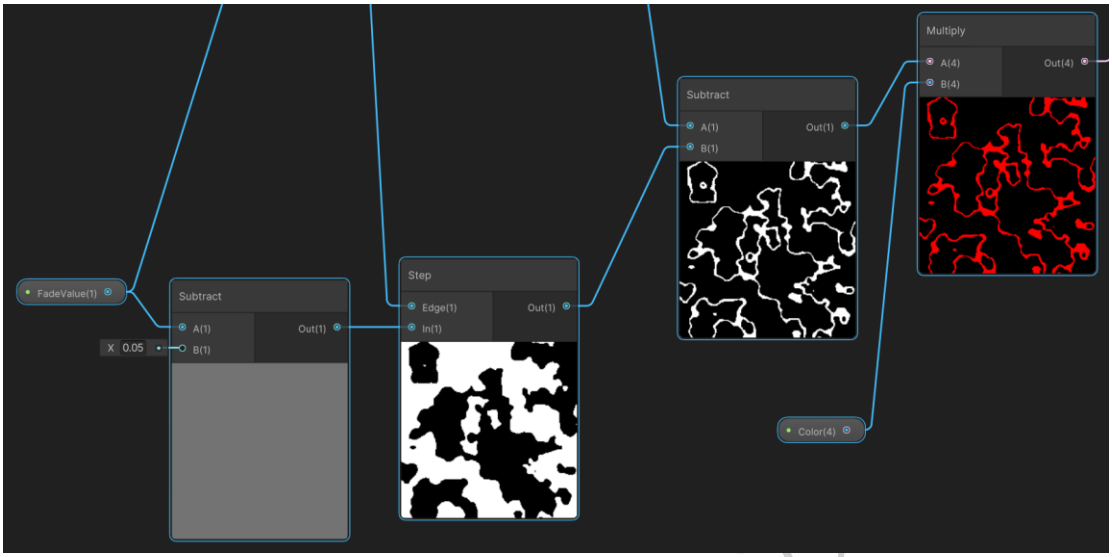


图117.　　图 6.6 溶解效果内边缘描边

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。

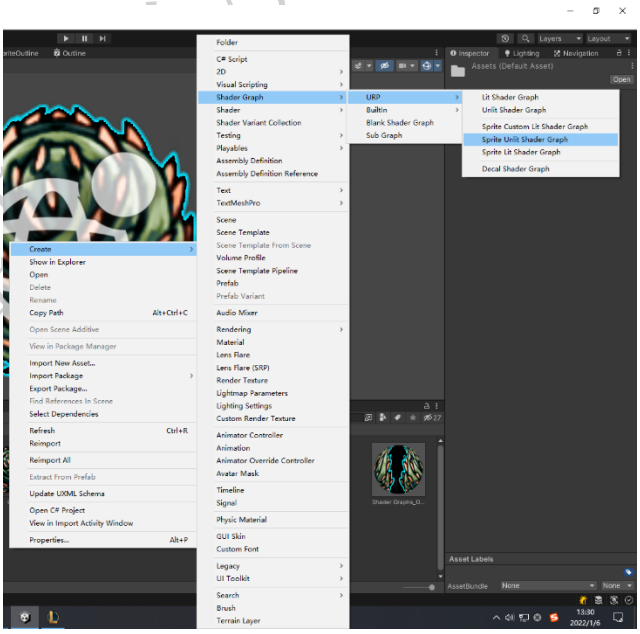


图118.　　图 7.1 创建 Sprite Unlit Shader Graph

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119.　图 7.2 Graph Inspector 面板按钮

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

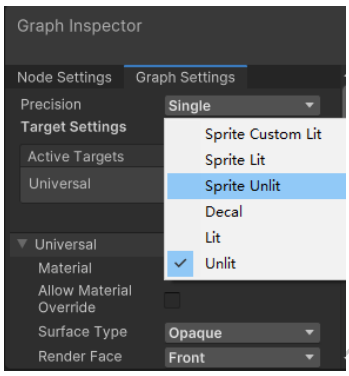


图120.　图 7.3 修改 Material

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。

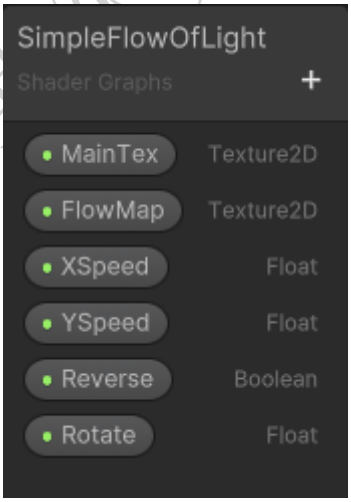


图121.　图 7.4 2D 流光简易版参数

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

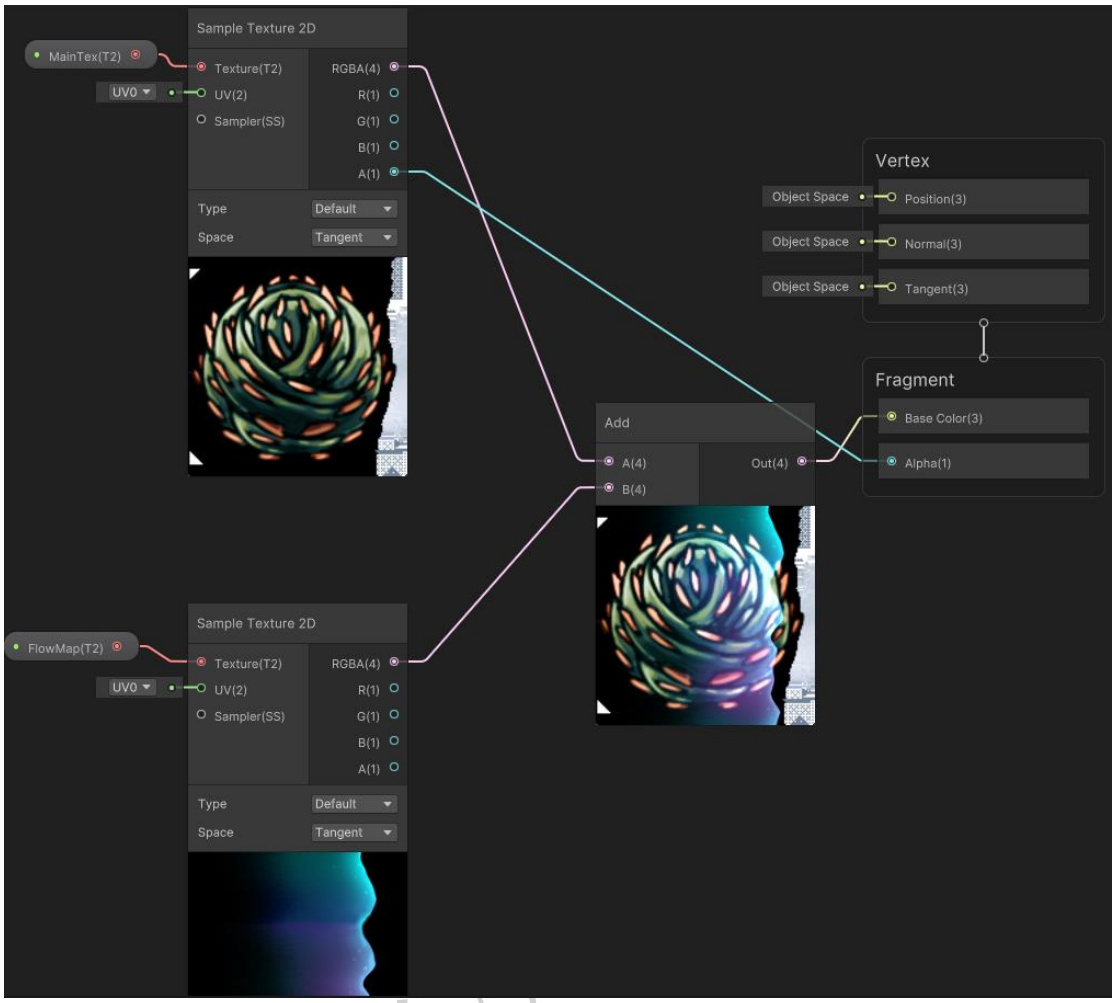


图122.　　图 7.5 含有流光效果的图像

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123.　　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。

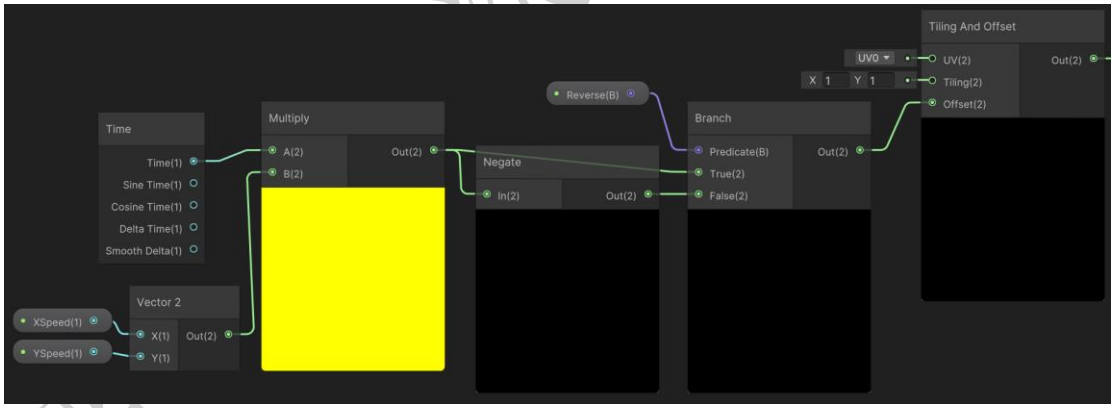


图124.　　图 7.7 流光反向运动

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）。

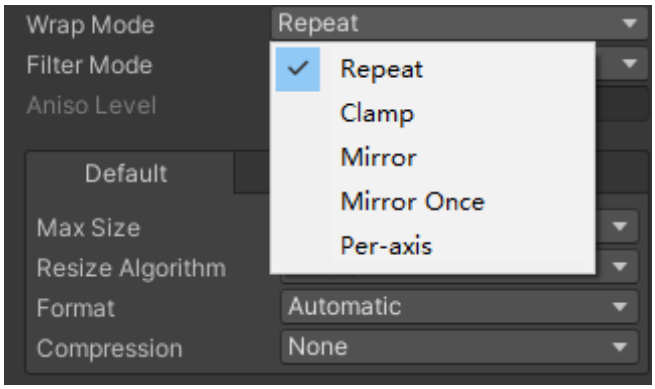


图125.　图检查纹理设置

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。

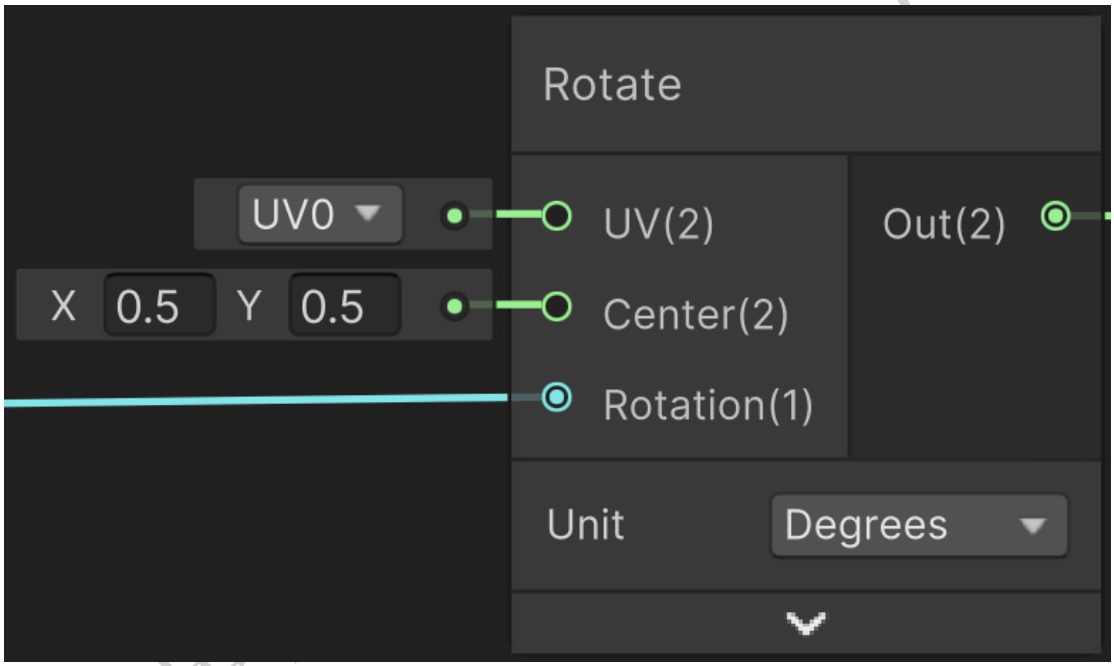


图126.　图 Rotate 节点控件

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。

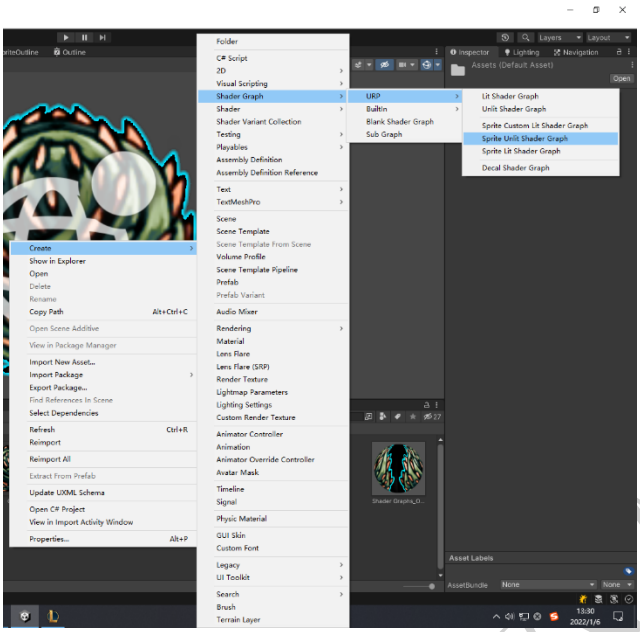


图127.　图 7.1 创建 Sprite Unlit Shader Graph

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128.　图 7.2 Graph Inspector 面板按钮

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

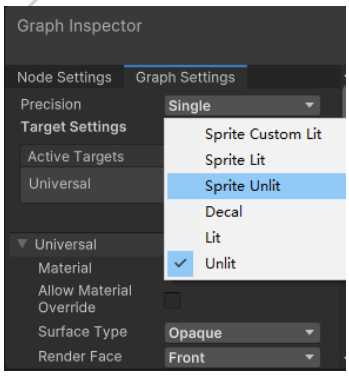


图129.　图 7.3 修改 Material

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

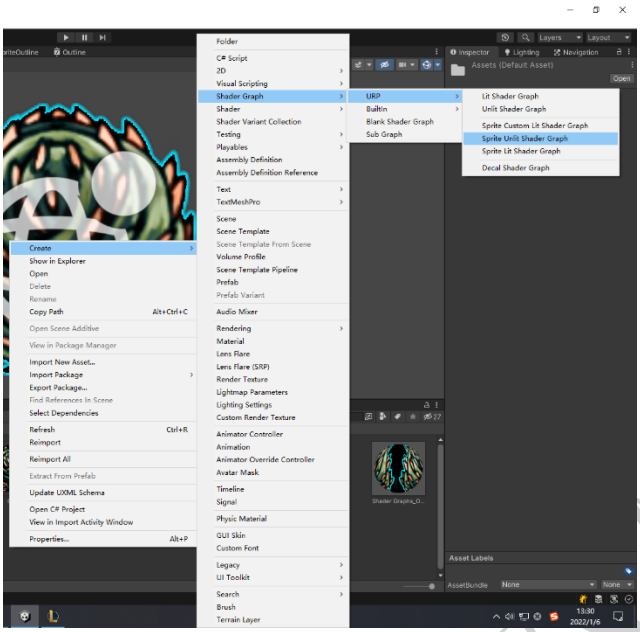


图1. 创建 Sprite Unlit Shader Graph

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130. Graph Inspector 面板按钮

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

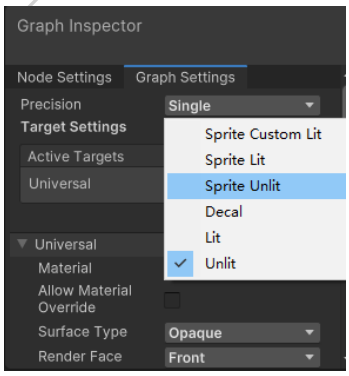


图131. 修改 Material

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。

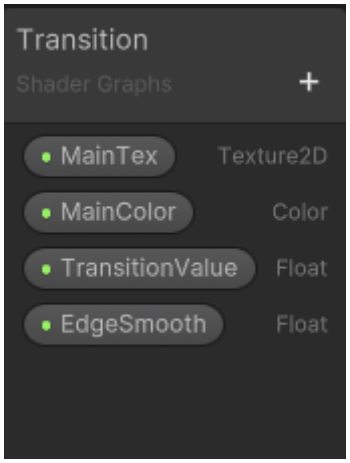


图132.　过渡效果参数

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。

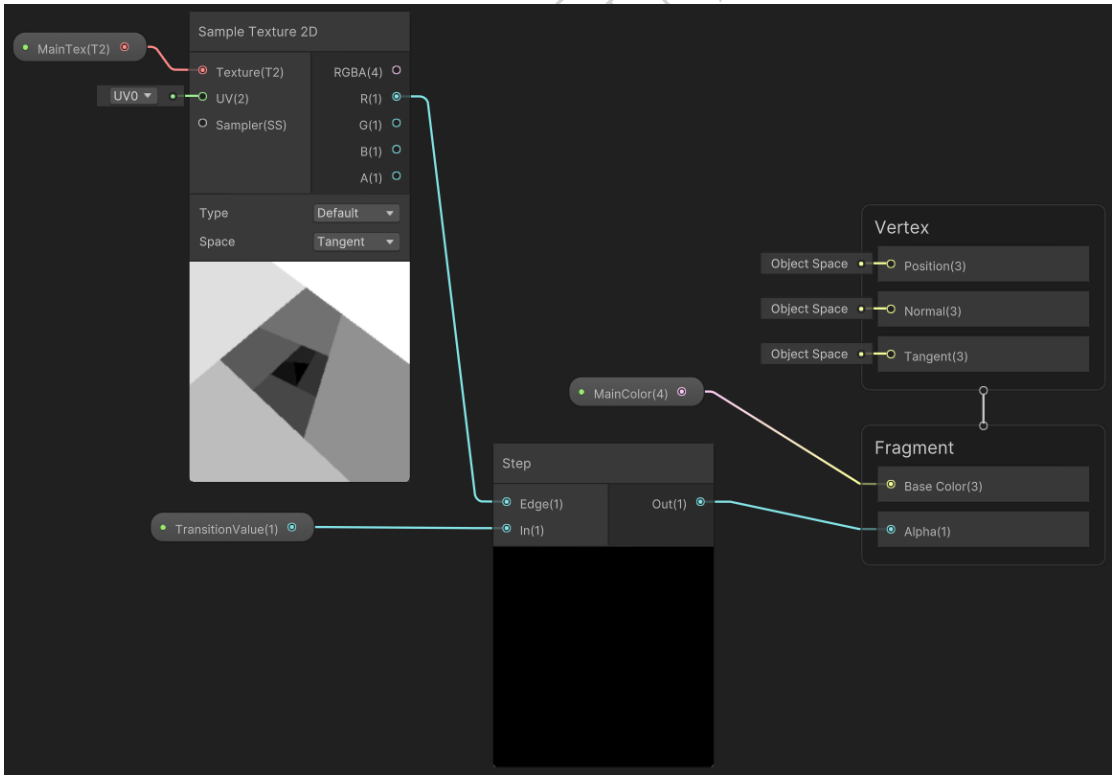


图133.　硬性过渡

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。

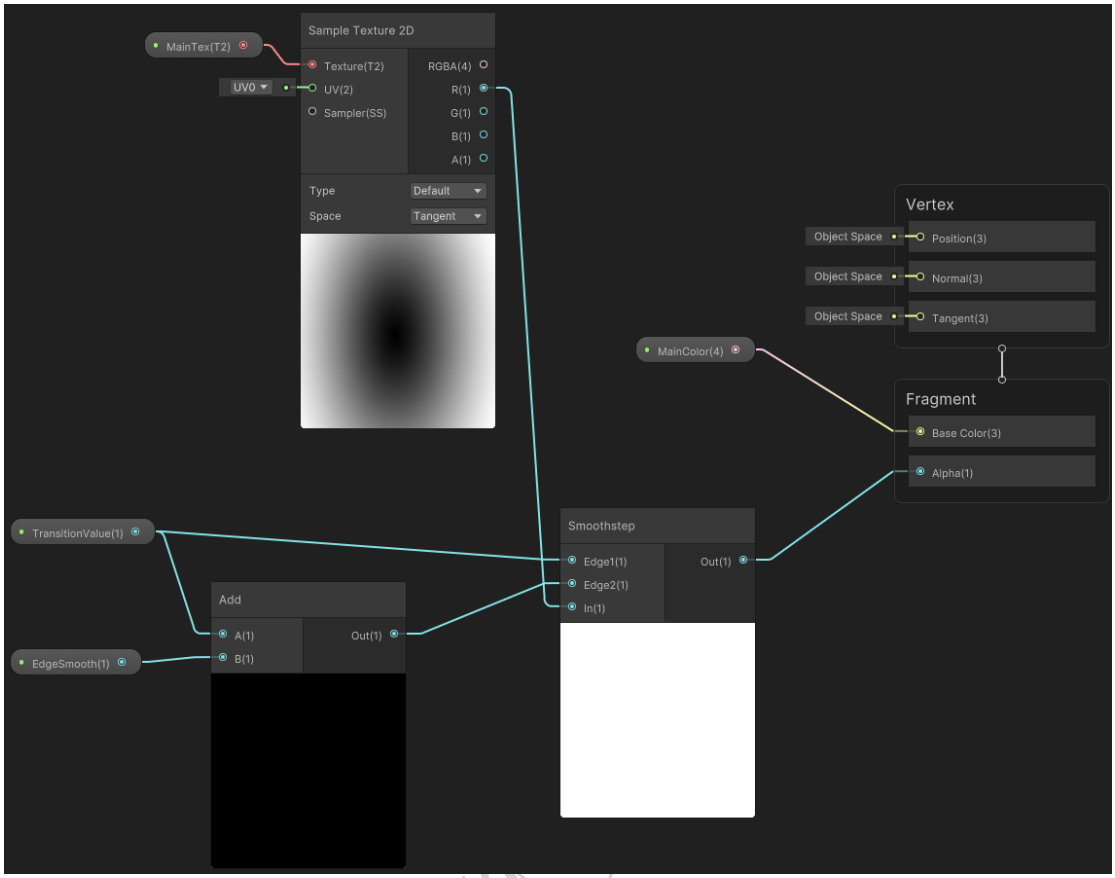


图134. 柔性过渡

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

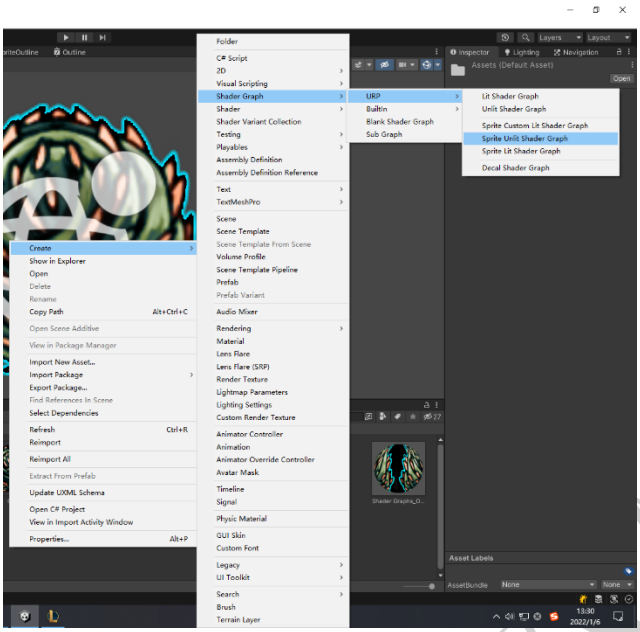


图2. 创建 Sprite Unlit Shader Graph

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135. Graph Inspector 面板按钮

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

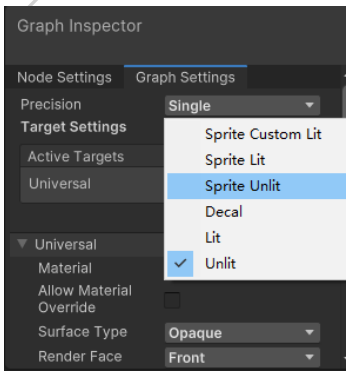


图136. 修改 Material

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

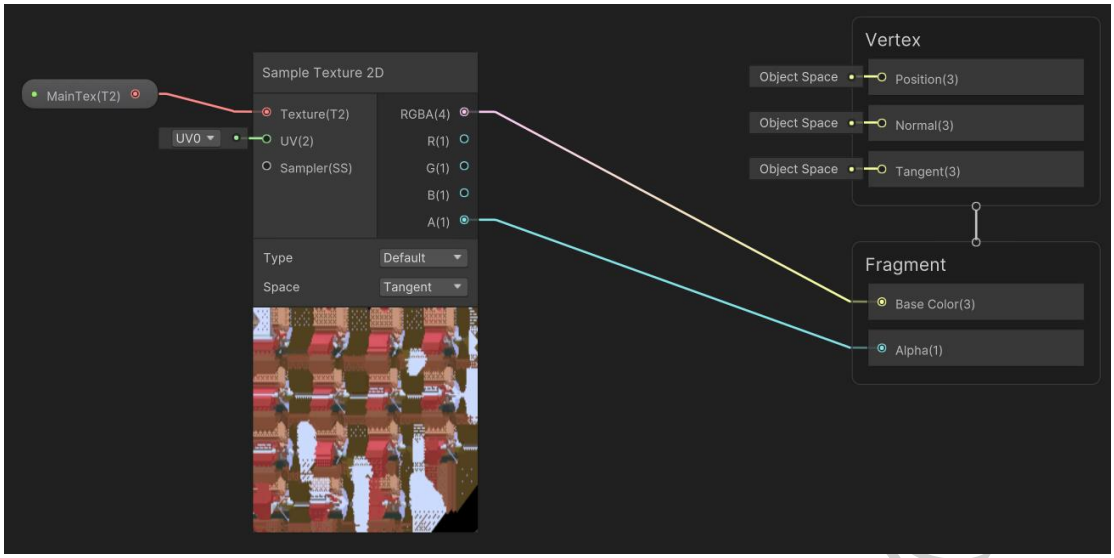


图137.　2D 帧动画主纹理

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。

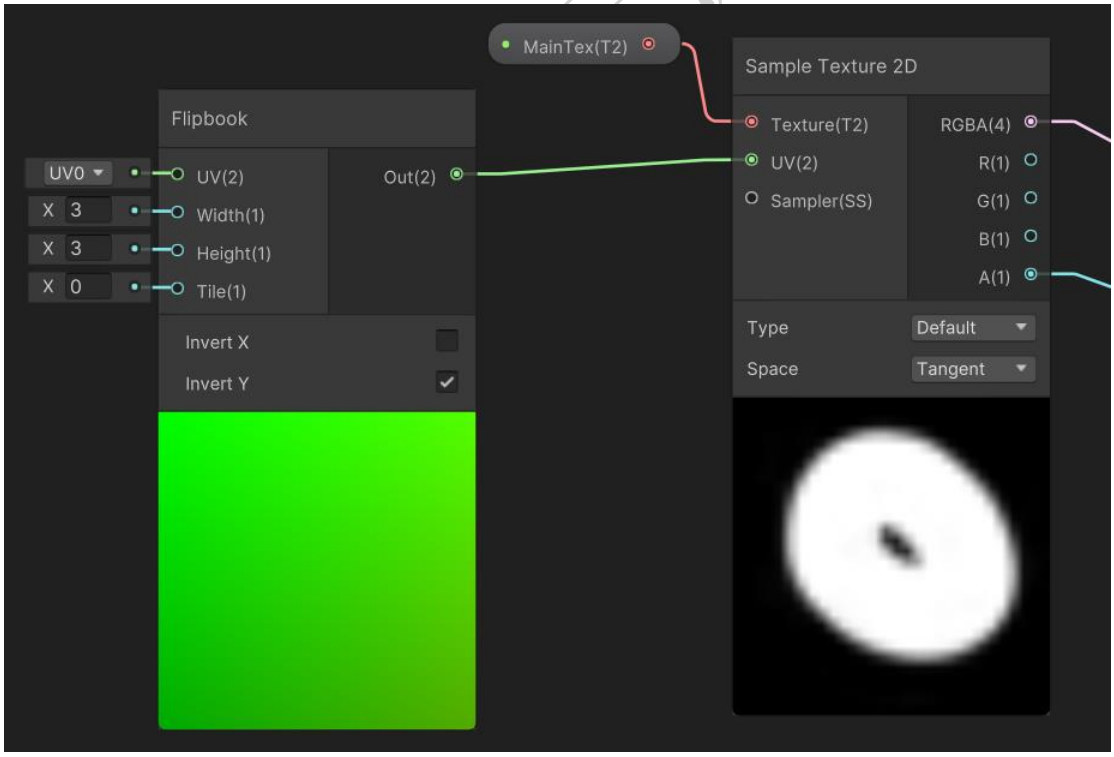


图138.　使用 Flipbook 节点

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。

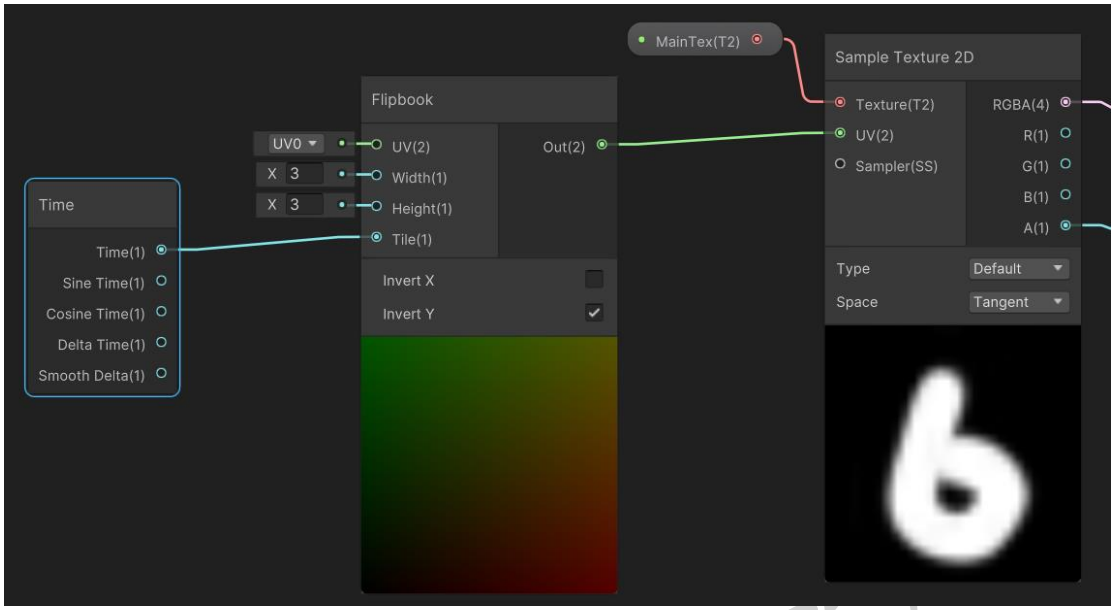


图139.　使用 Time 节点让瓦片依次显示

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。

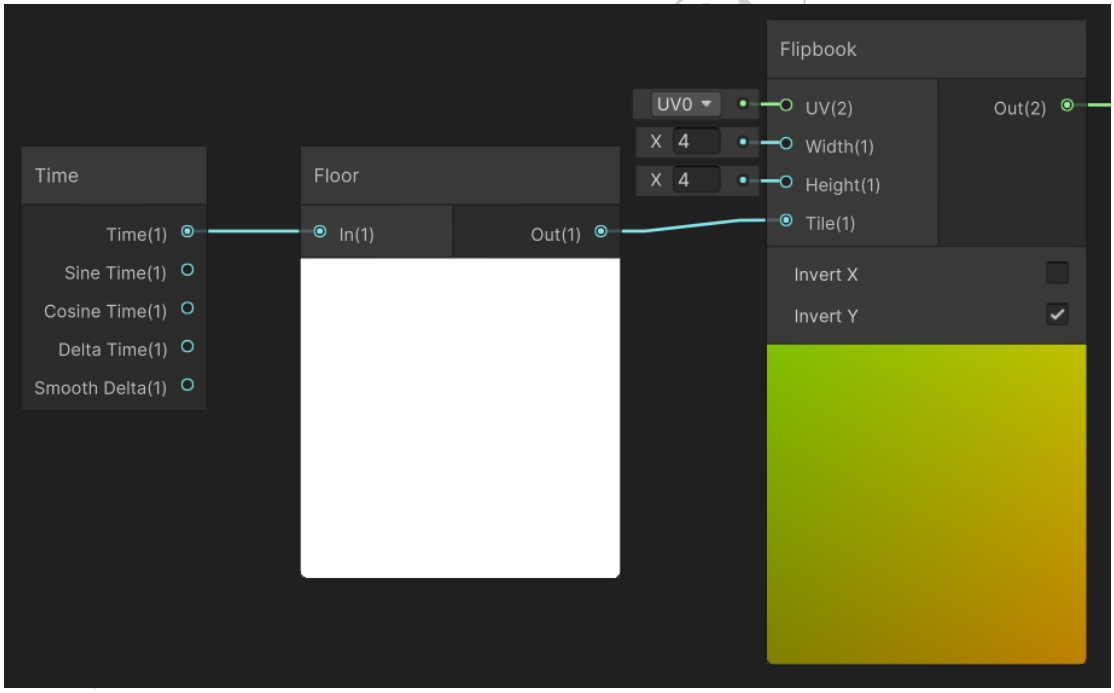


图140.　对 Time 节点输出取整

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。

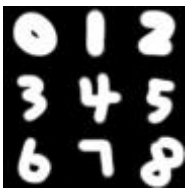


图141.　测试图

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。

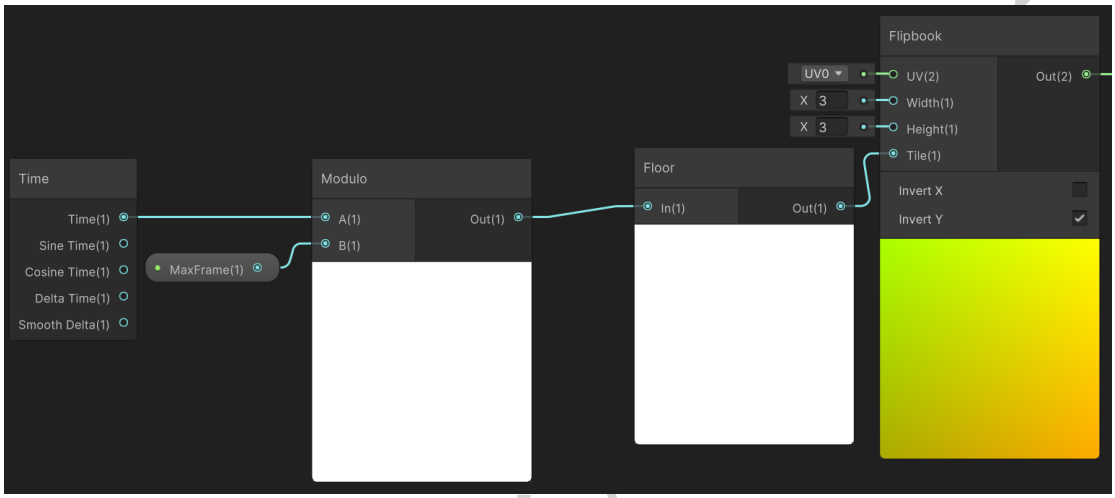


图142.　对 Time 节点输出取余

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。

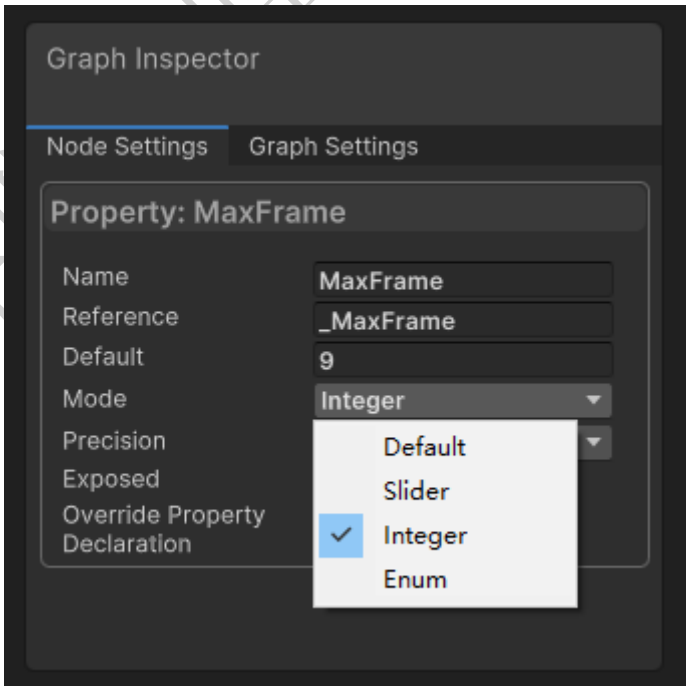


图143.　修改参数类型为整型

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。

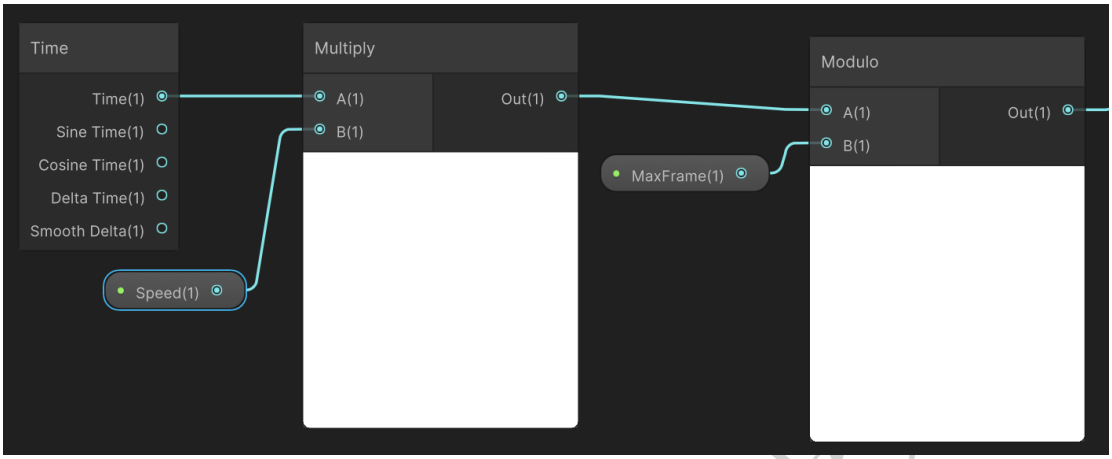


图144. 调节帧动画速度

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。

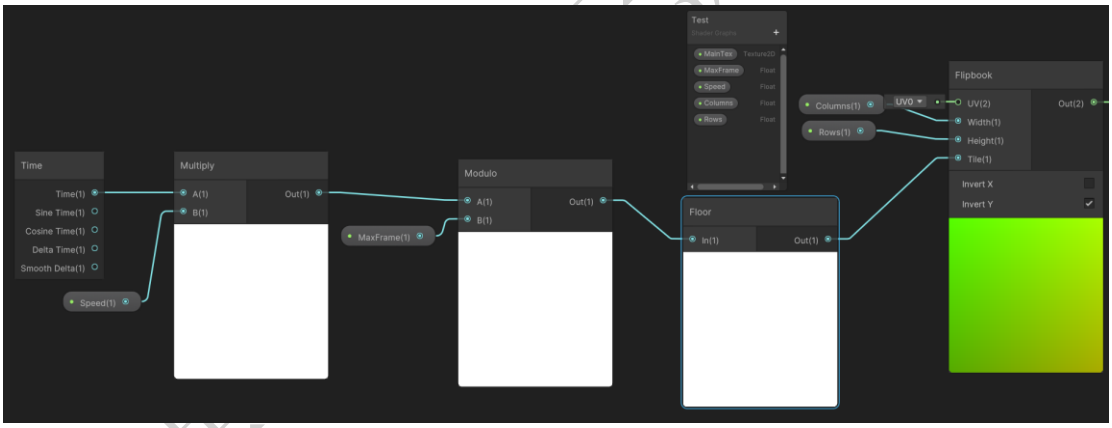


图145. 添加行和列参数

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。

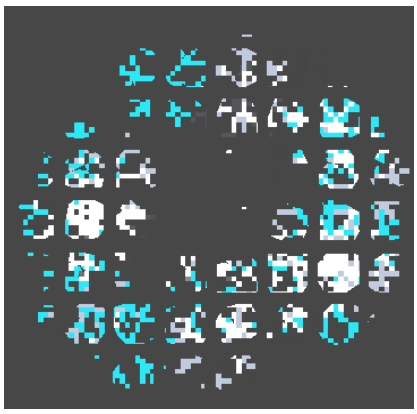


图146. 显示图形出错

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

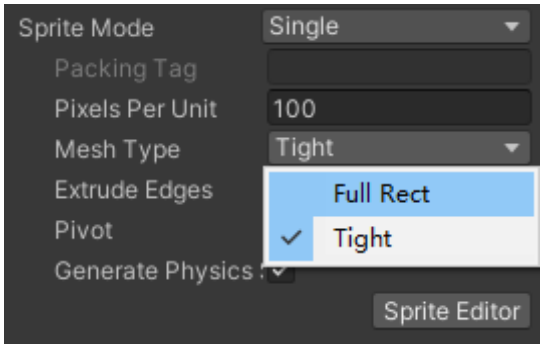


图147.　修改 Mesh Type

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。

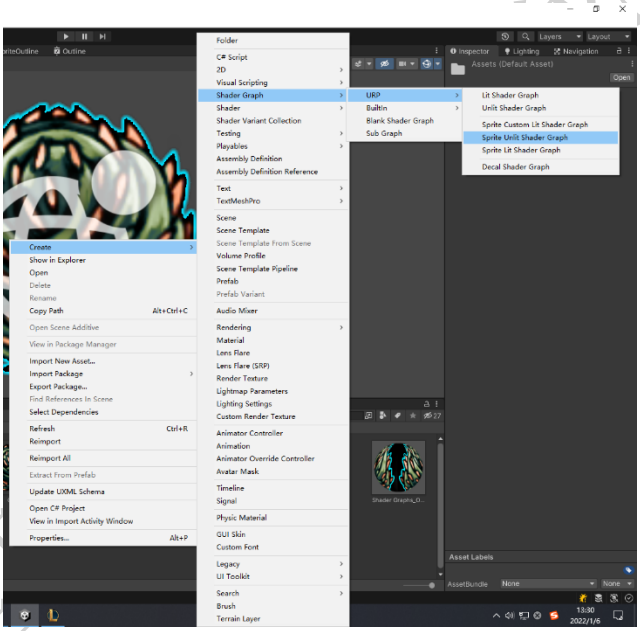


图3. 创建 Sprite Unlit Shader Graph

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148.　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

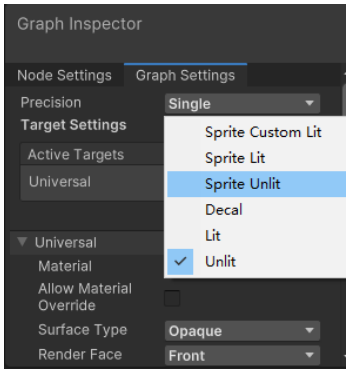


图149.　　修改 Material

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。

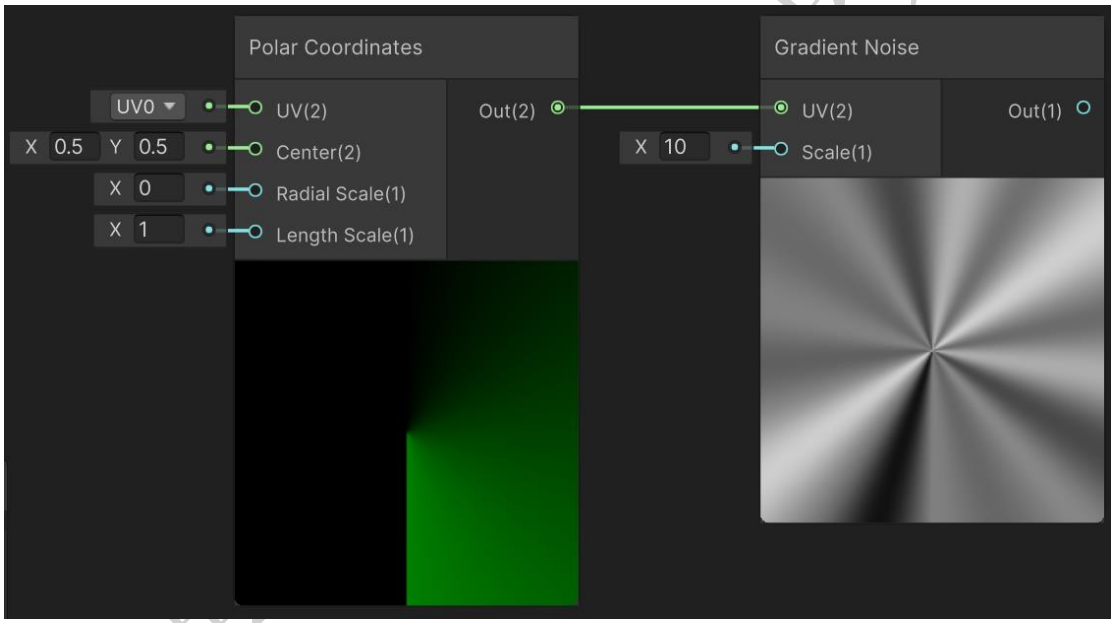


图150.　　速度线雏形

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。

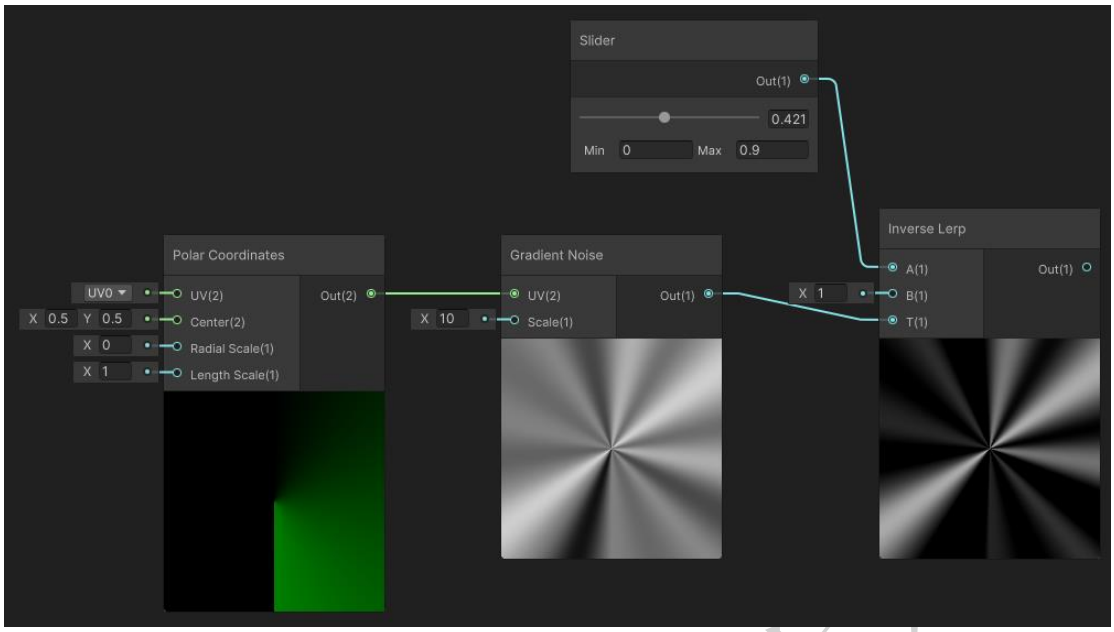


图151.　　对速度线图像进行过滤

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。

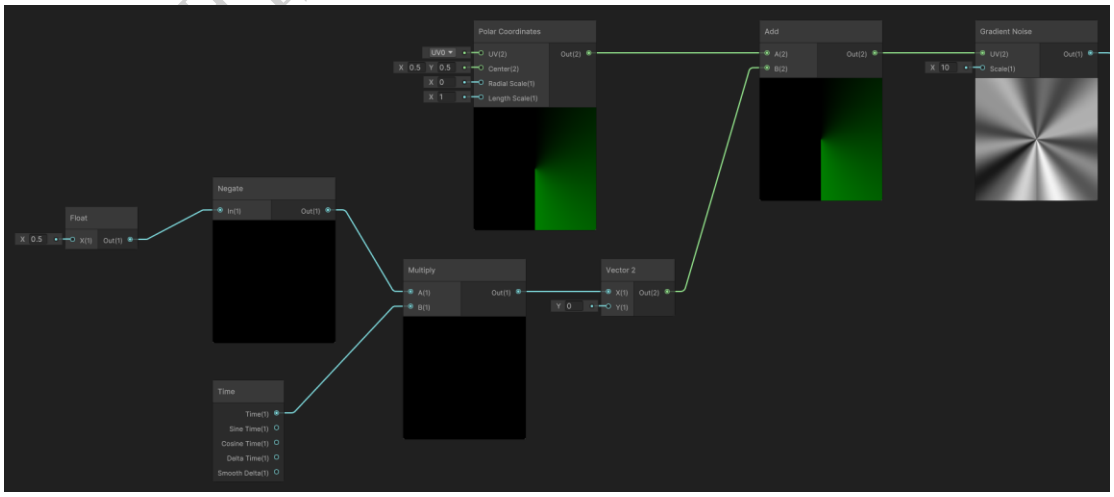


图152.　　速度线运动

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。

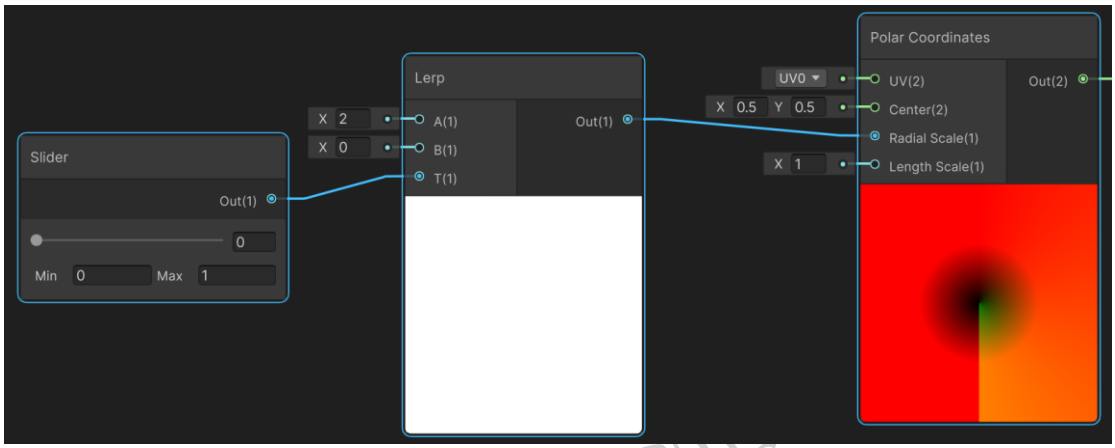


图153.　Mask 显示规模

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。

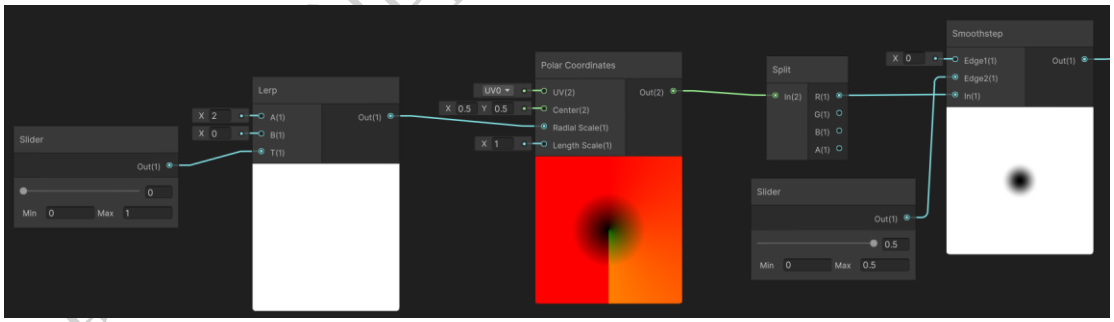


图154.　速度线蒙版

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。

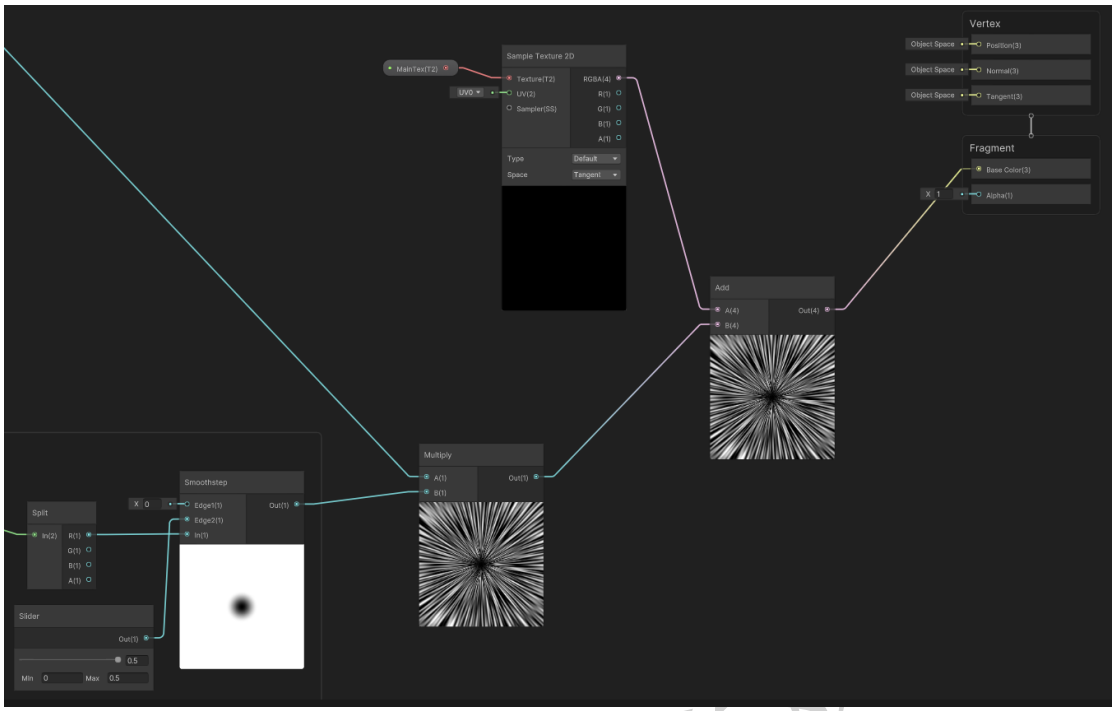


图155.　　速度线最终输出

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。

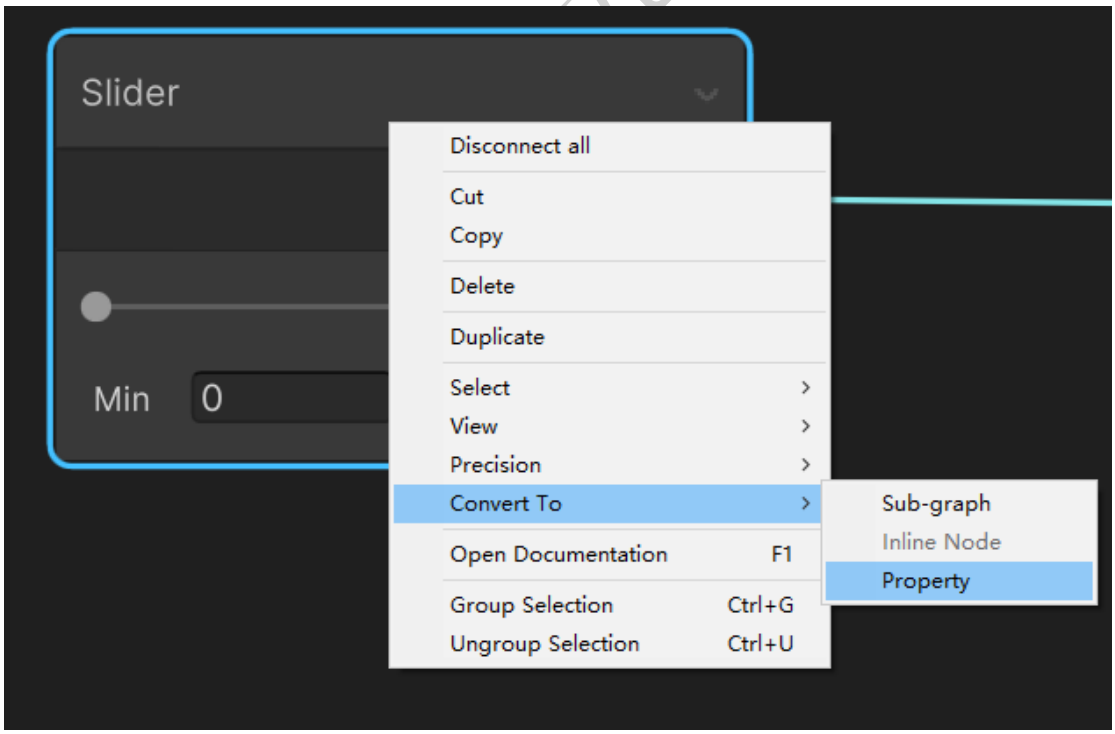


图156.　　转换为参数

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。

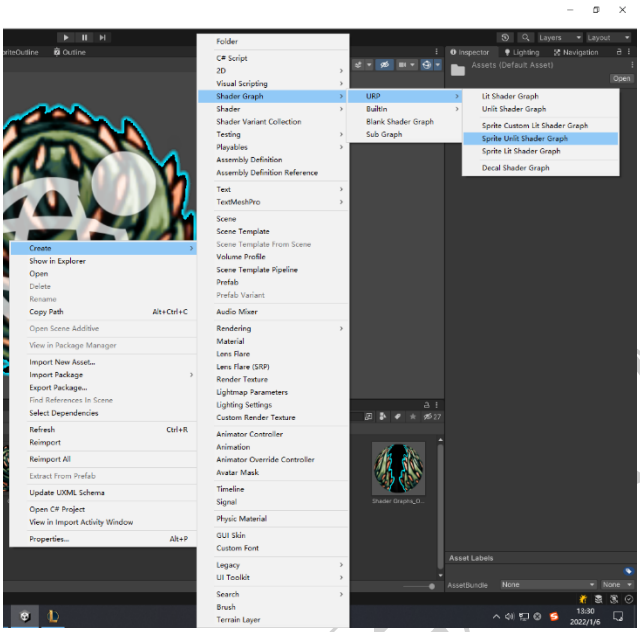


图4. 创建 Sprite Unlit Shader Graph

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157. Graph Inspector 面板按钮

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

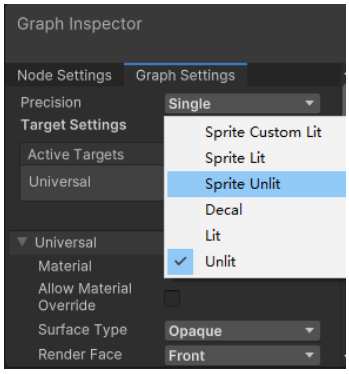


图158. 修改 Material

[13] adj. 普遍的；通用的 n. 普遍原则；普遍看法

2. 原理解释

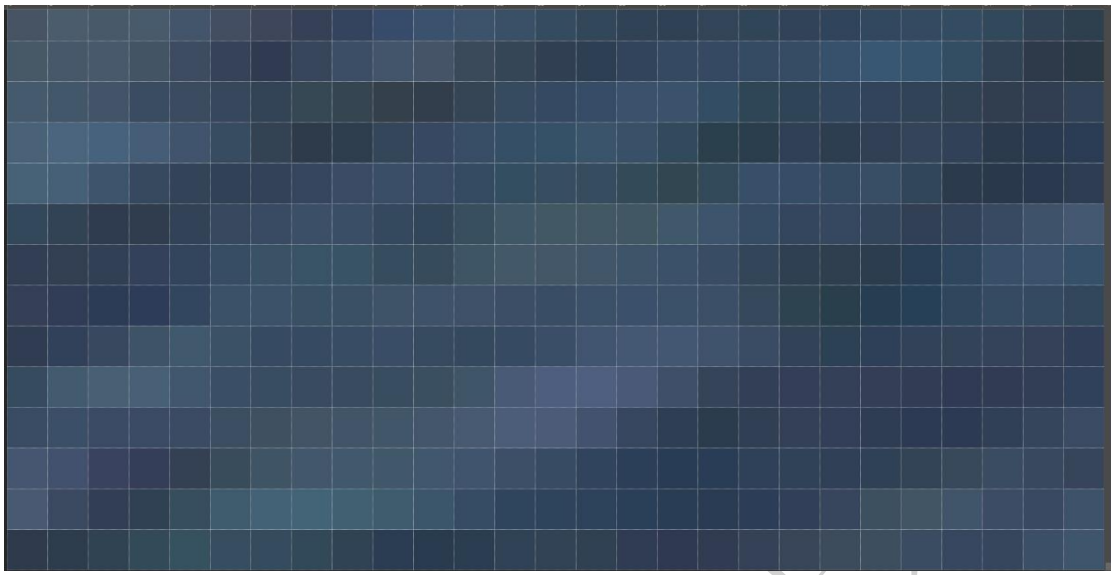


图159. 像素点

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。

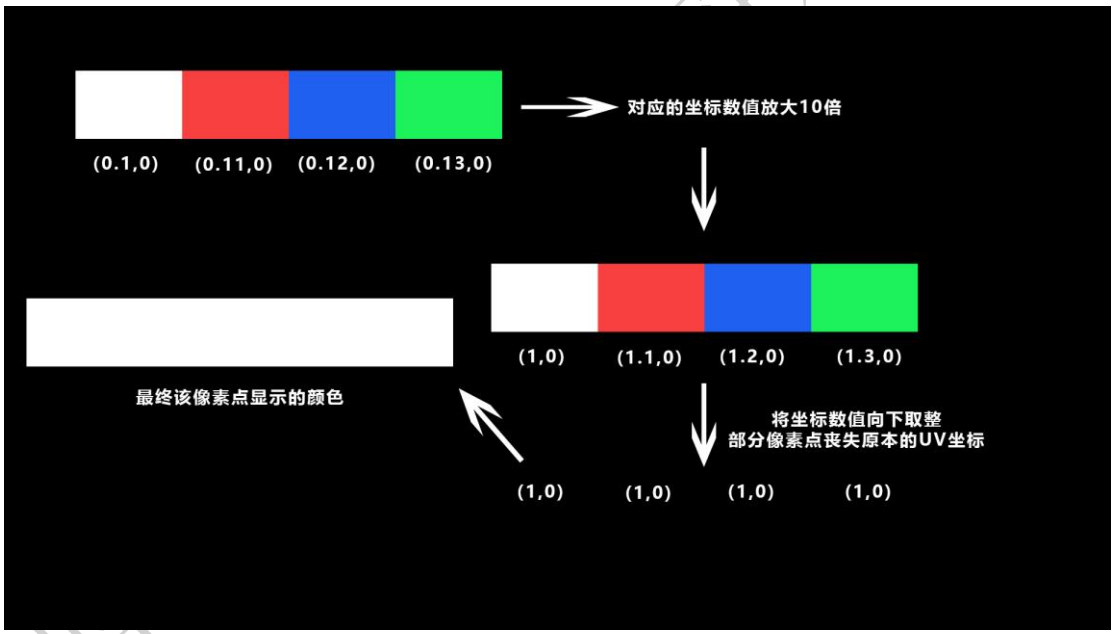


图160. 像素化原理

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。

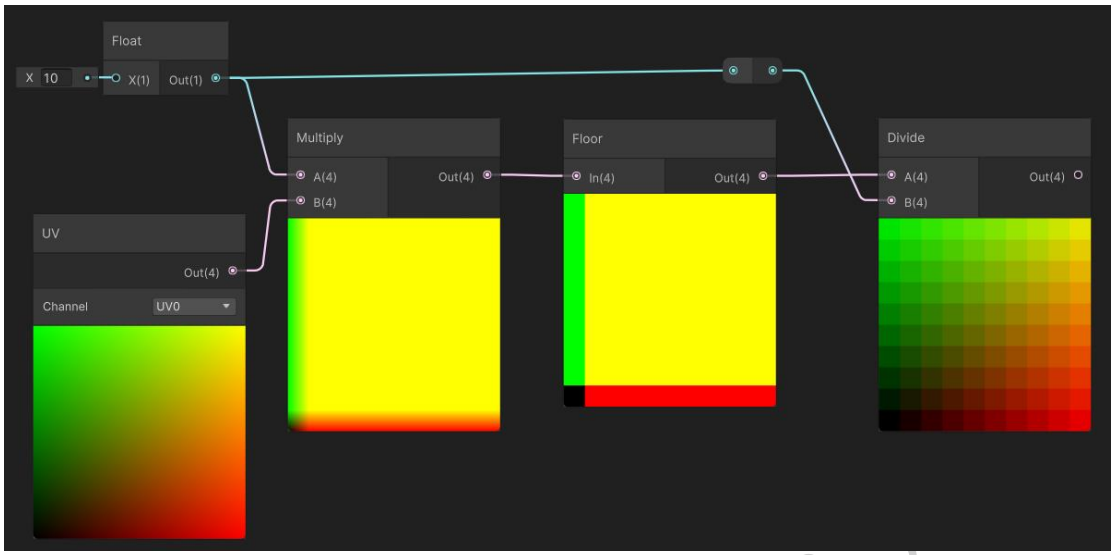


图161.　　修改 UV 坐标

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。

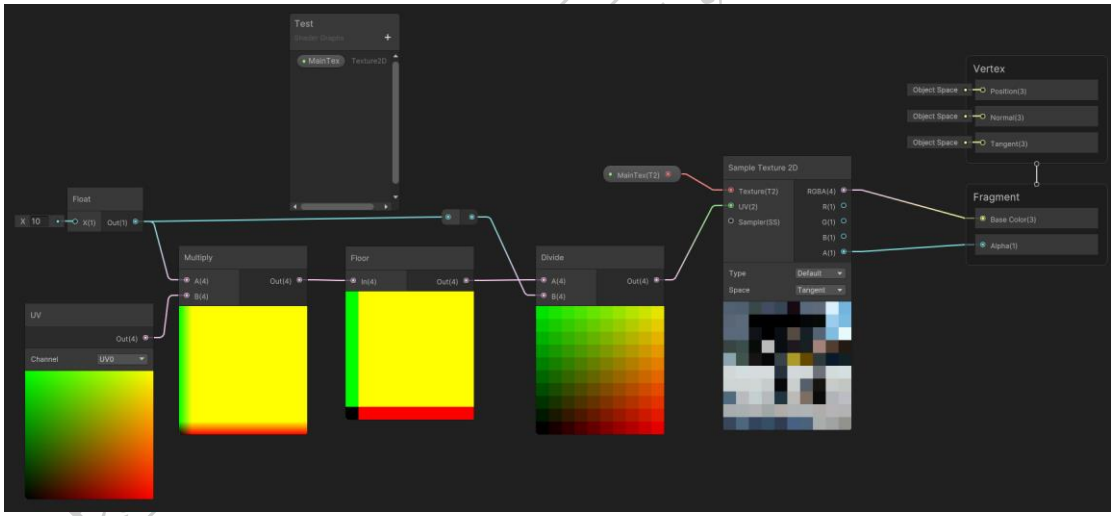


图162.　　输出

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。

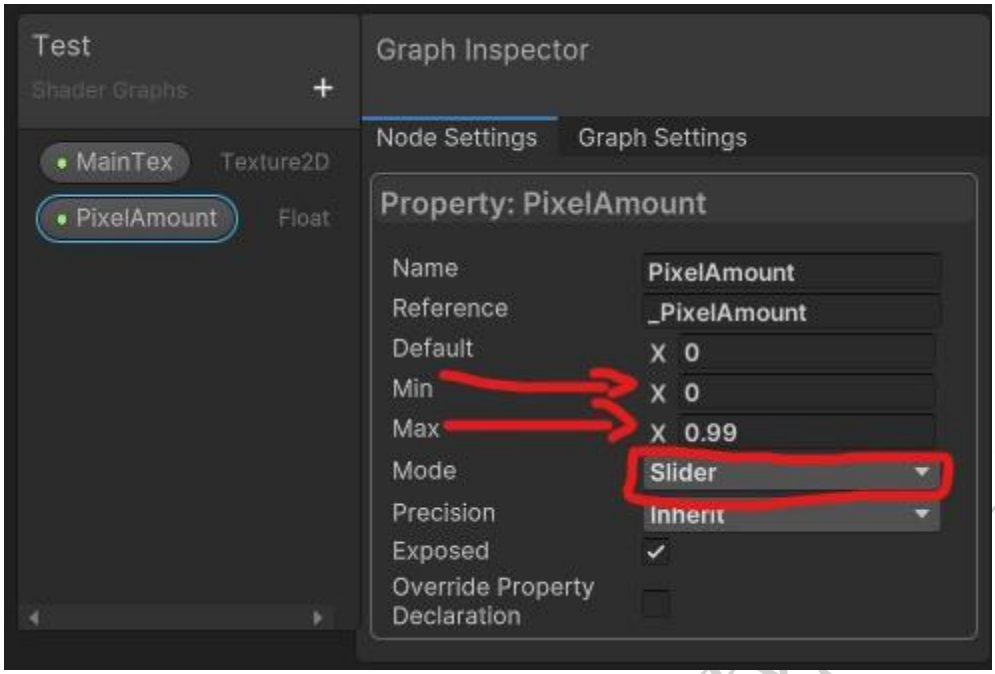


图163.　　修改参数模式为 Slider

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。

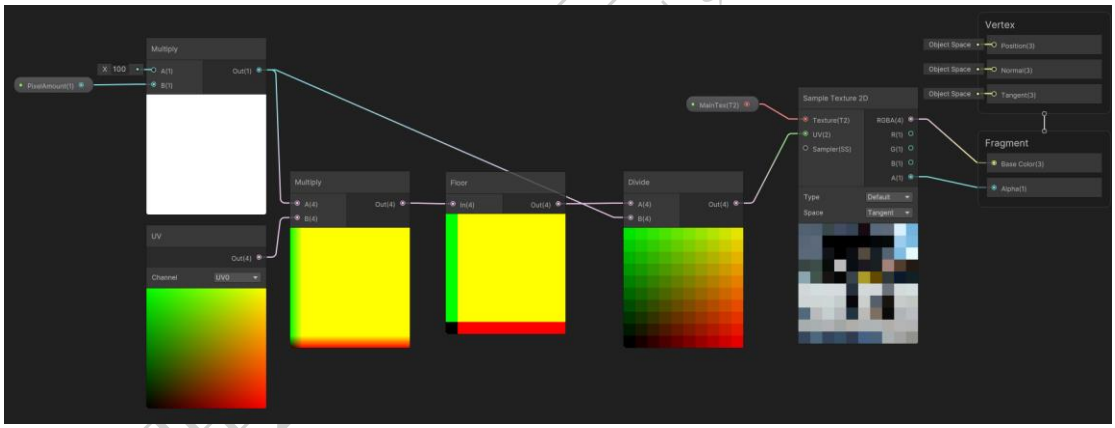


图164.　　使用参数控制

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）

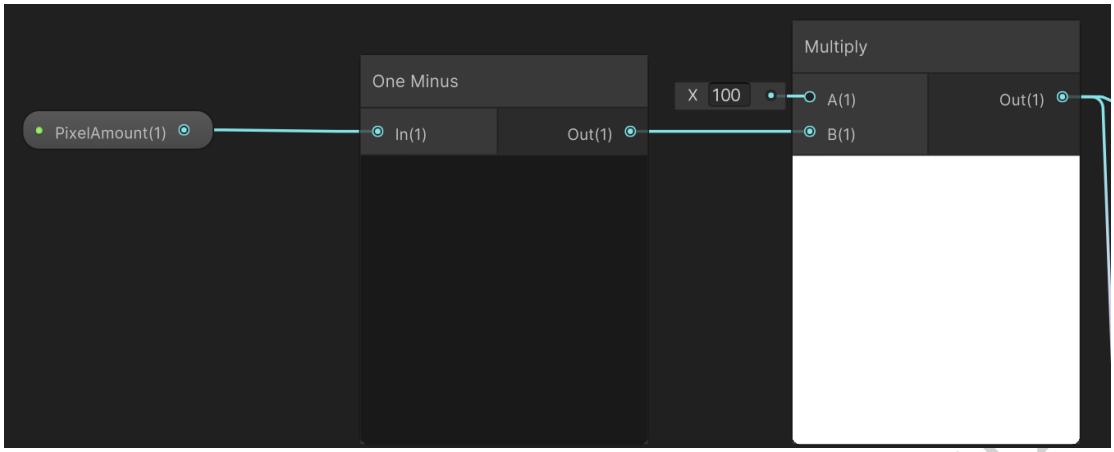


图165.　使用 OneMinus 修改参数

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。

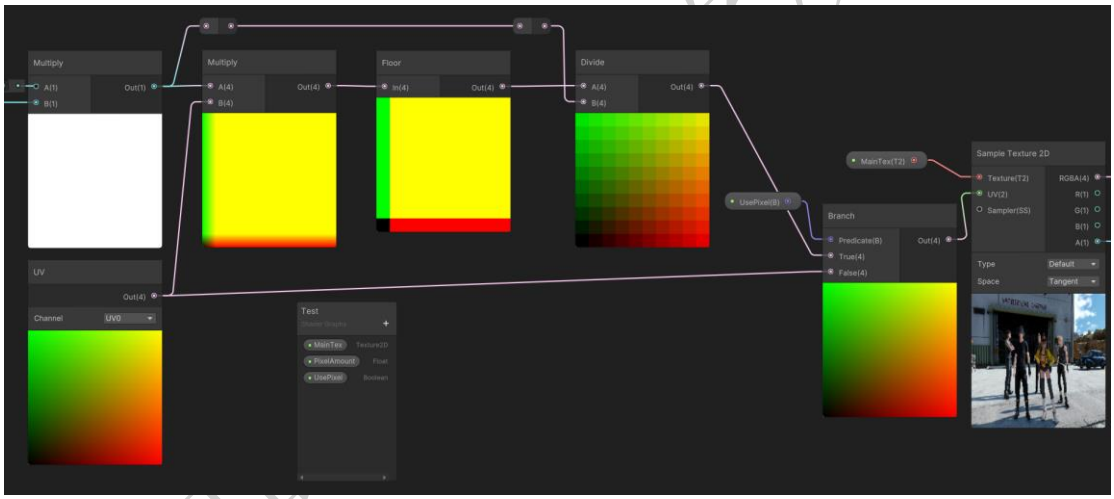


图166.　使用 Branch 确定是否使用像素化效果

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。

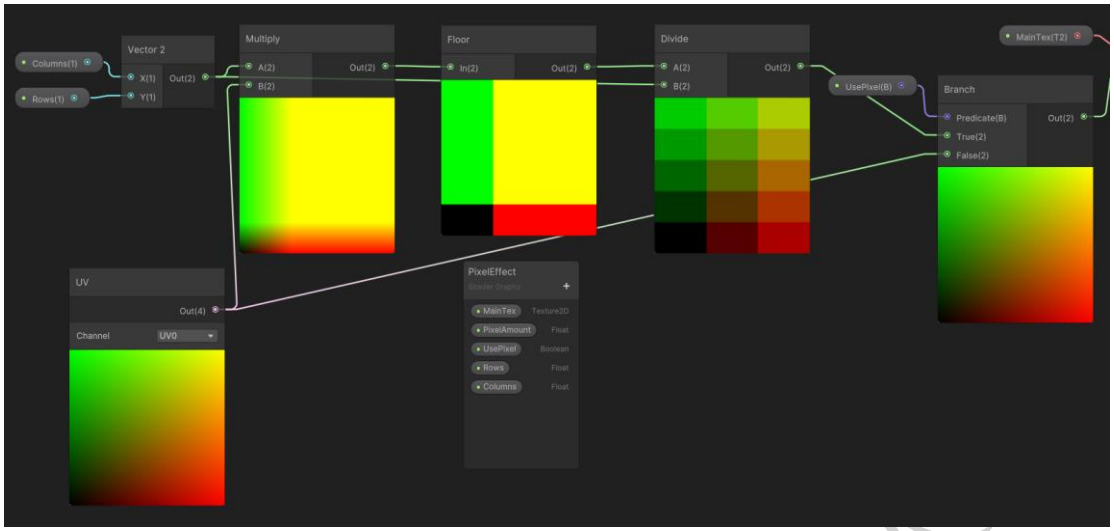


图167.　自定义像素化行数和列数

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。

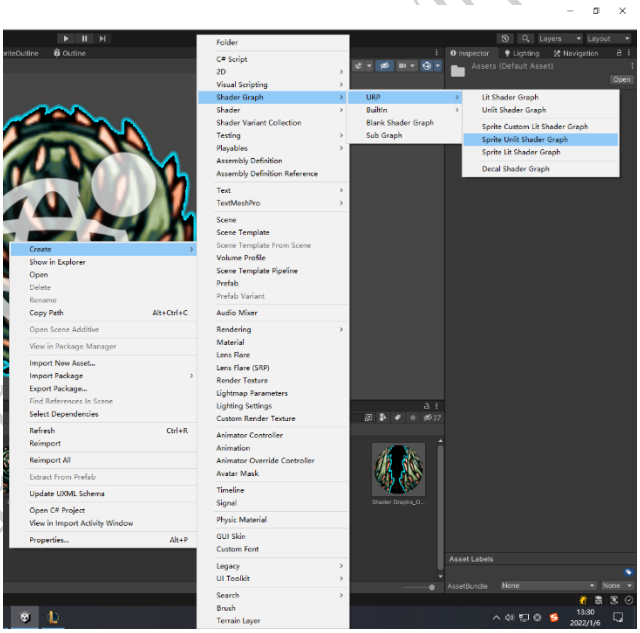


图5. 创建 Sprite Unlit Shader Graph

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168.　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。

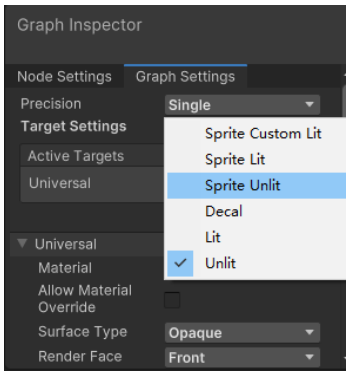


图169.　　修改 Material

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。

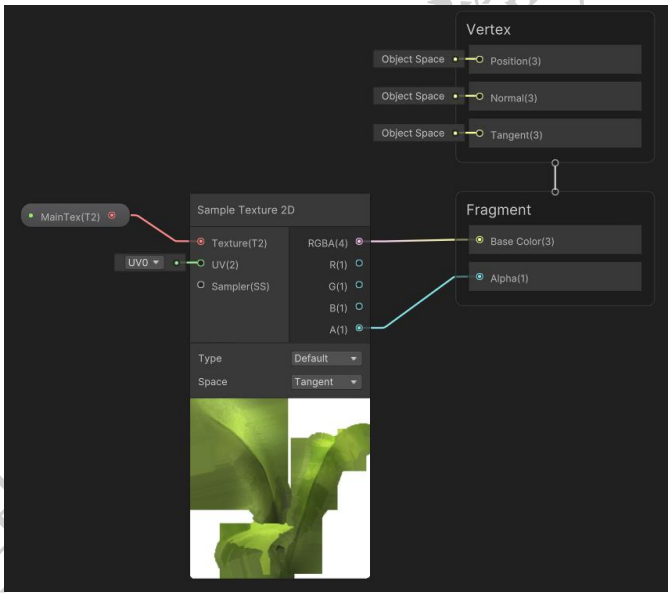


图170.　　主纹理输出

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。

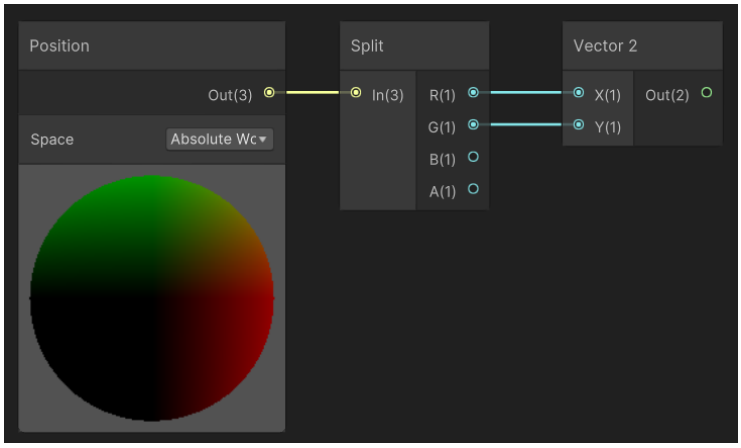


图171.　剥离 X 分量和 Y 纹理

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。

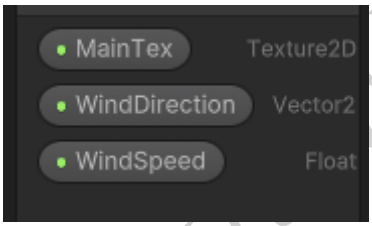


图172.　添加风速和风向参数

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。

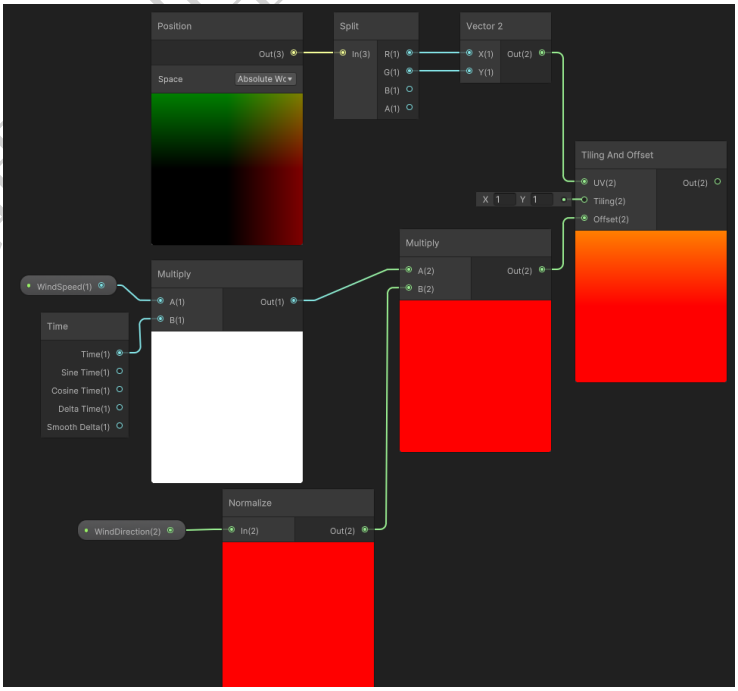


图173.　进行偏移

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。

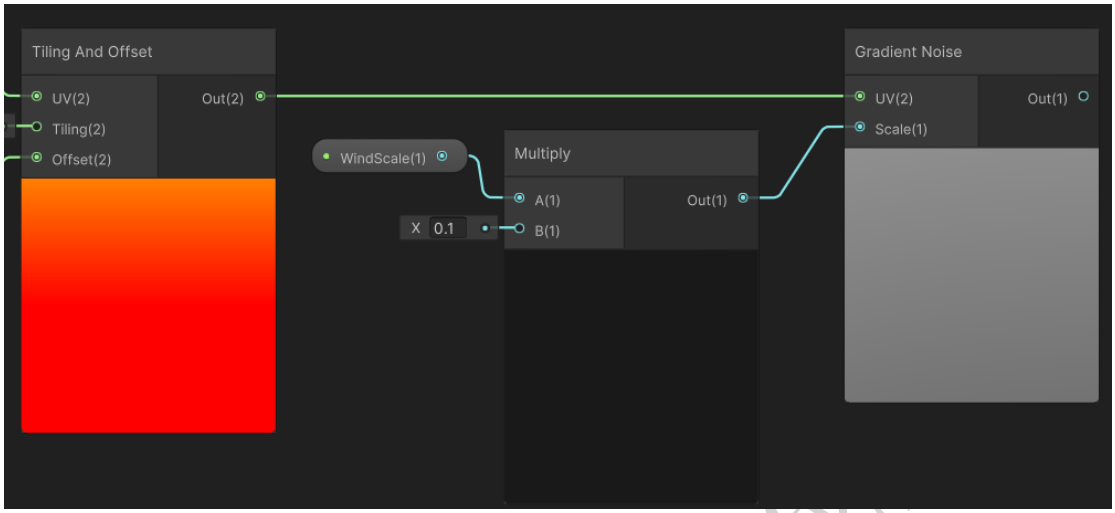


图174.　　使用噪声调整偏移强度

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。

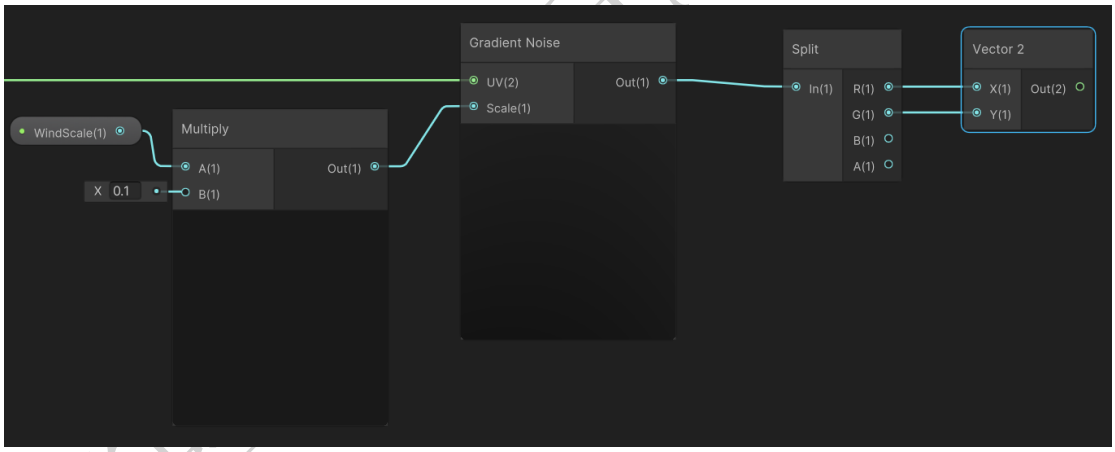


图175.　　确保输出是二维向量

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。

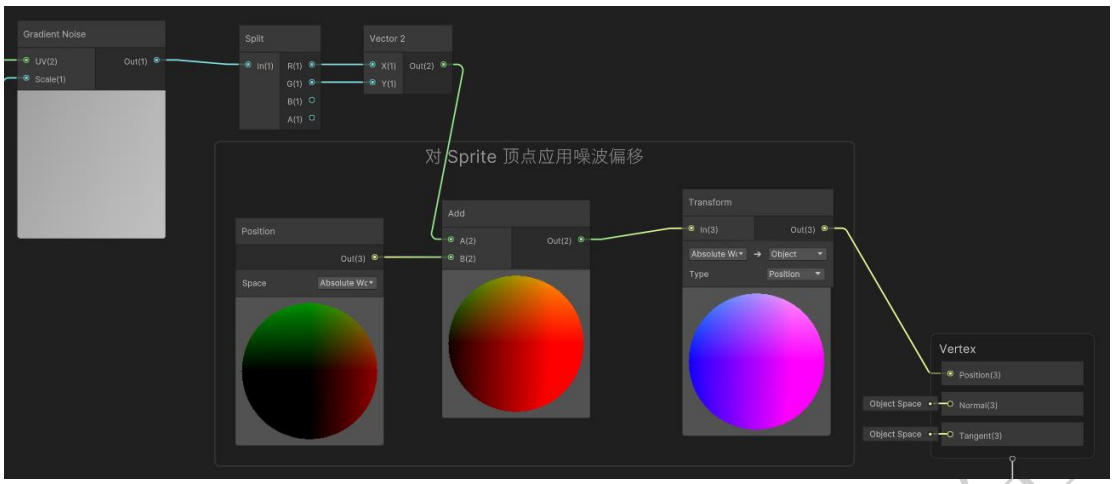


图176.　　对顶点应用噪声偏移

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。

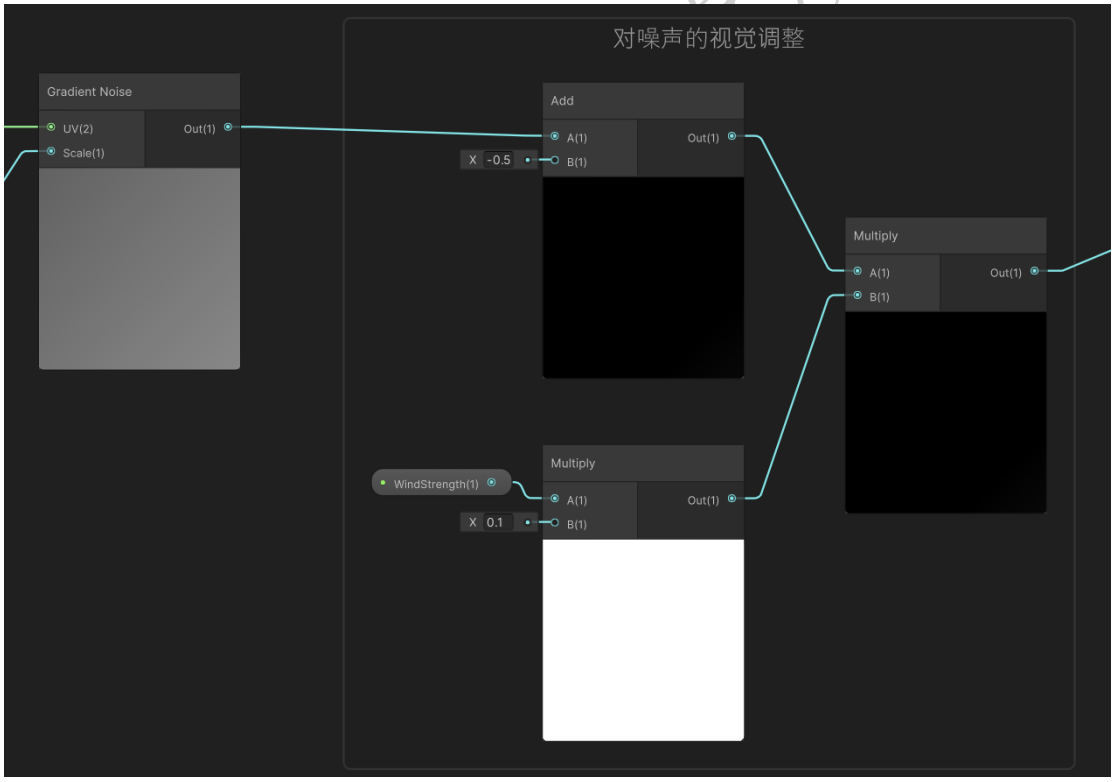


图177.　　调整噪声强度

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。

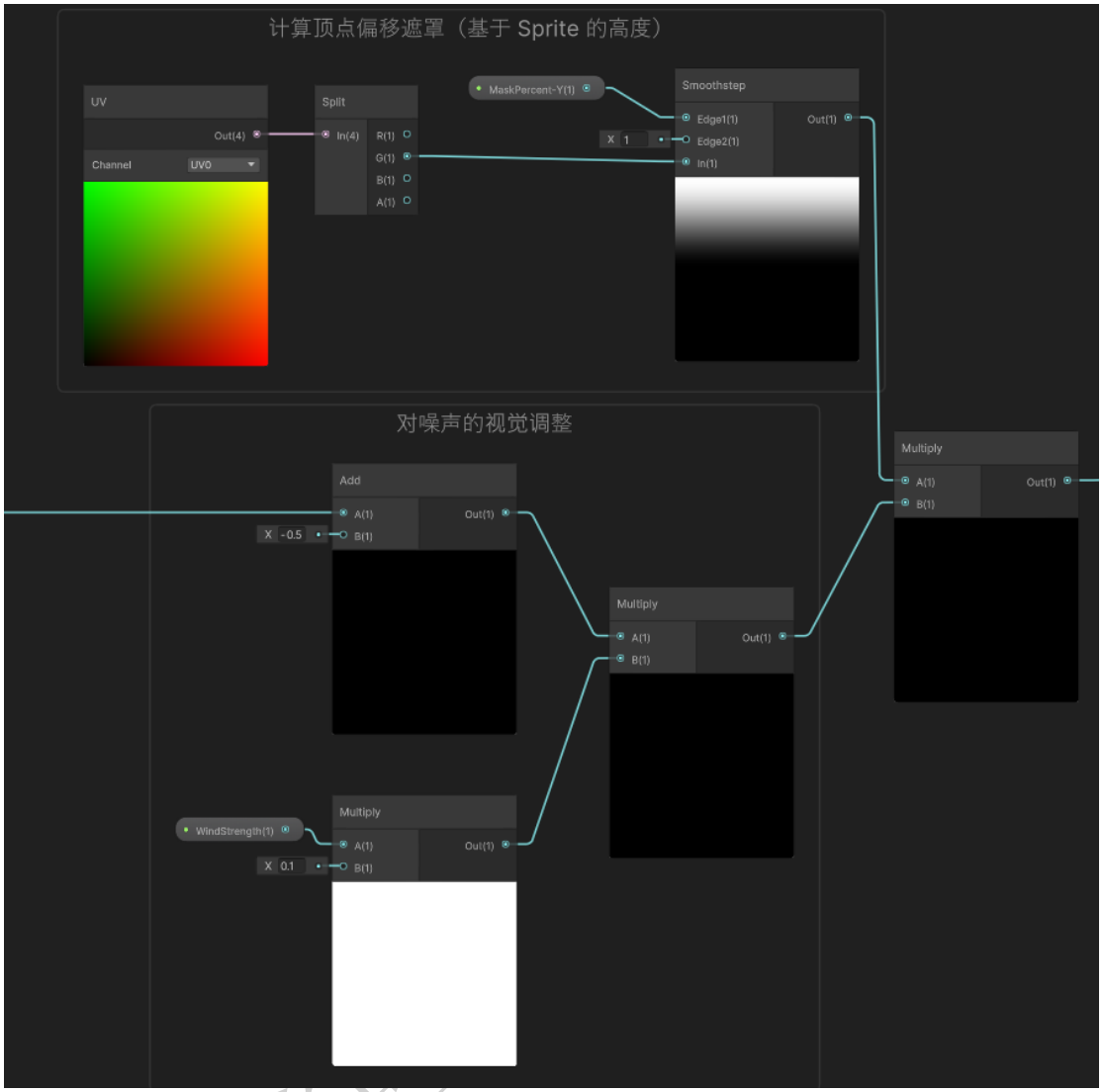


图178.　　顶点偏移蒙版

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。

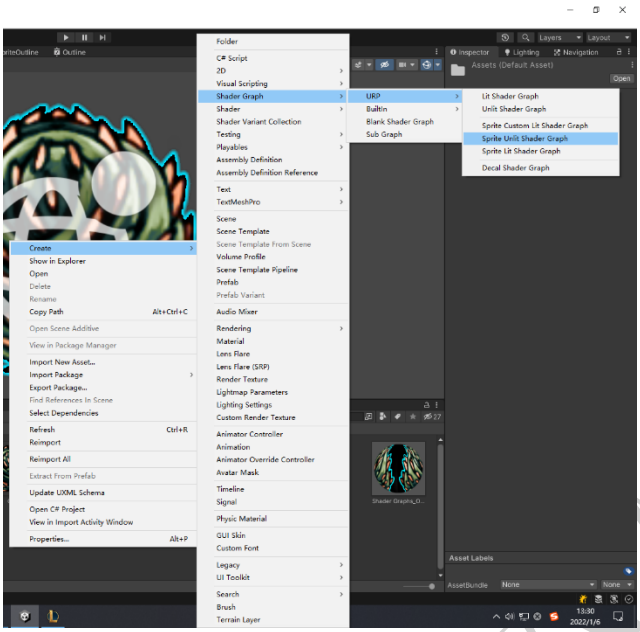


图6. 创建 Sprite Unlit Shader Graph

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179. Graph Inspector 面板按钮

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

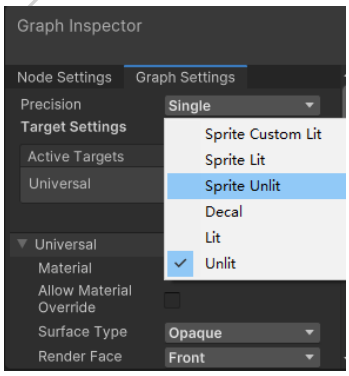


图180. 修改 Material

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181.　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。

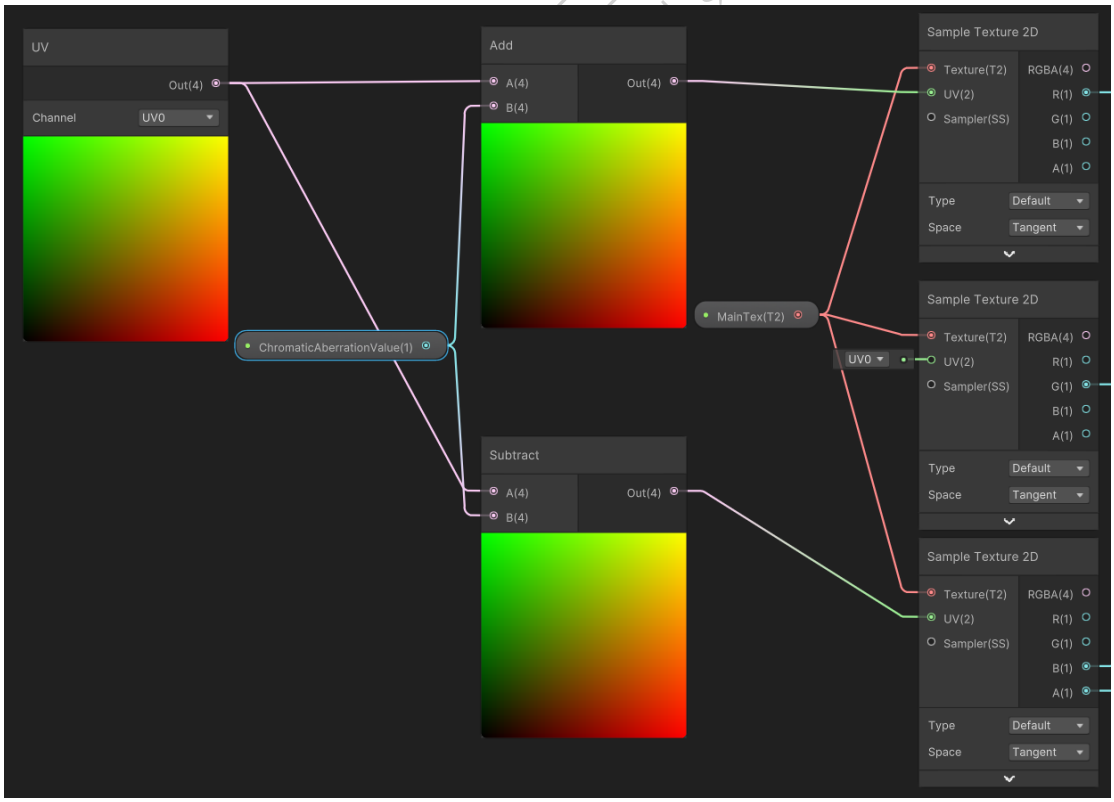


图182.　对 UV 进行加减，实现色差效果

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。

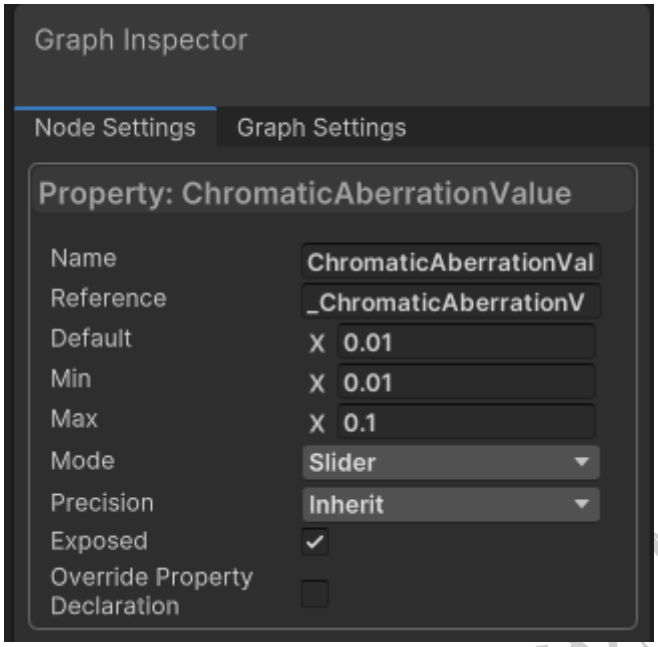


图183.　设置色差参数的模式为 Slider

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。

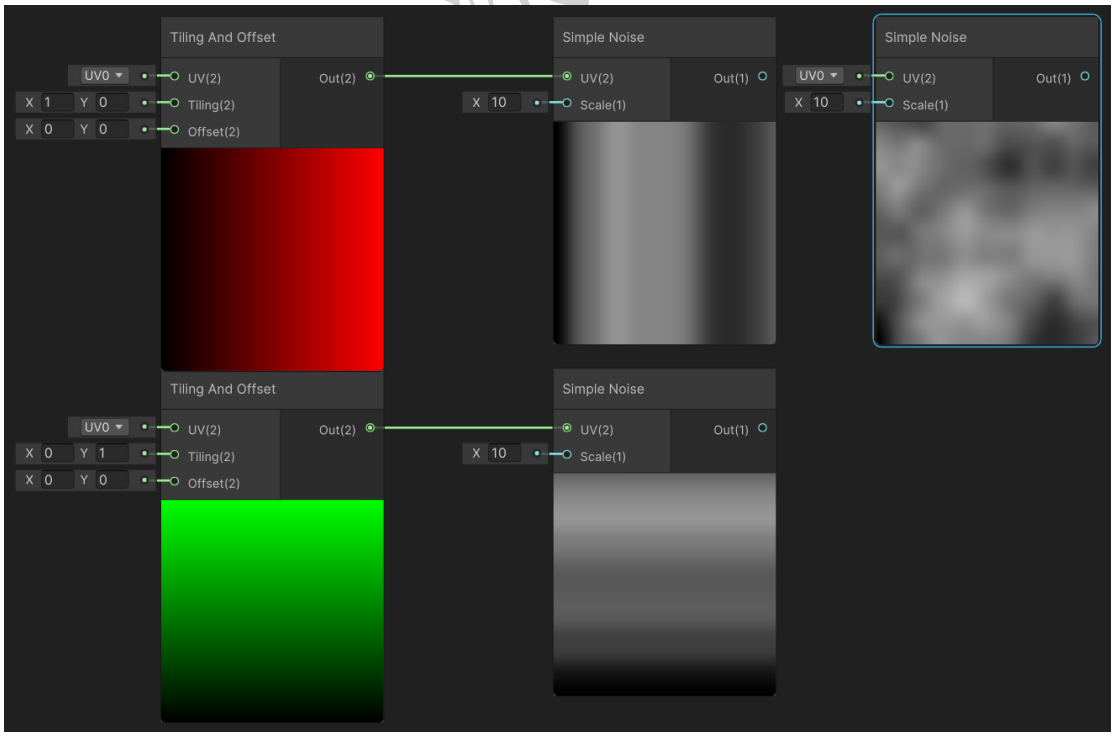


图184.　效果对照

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。

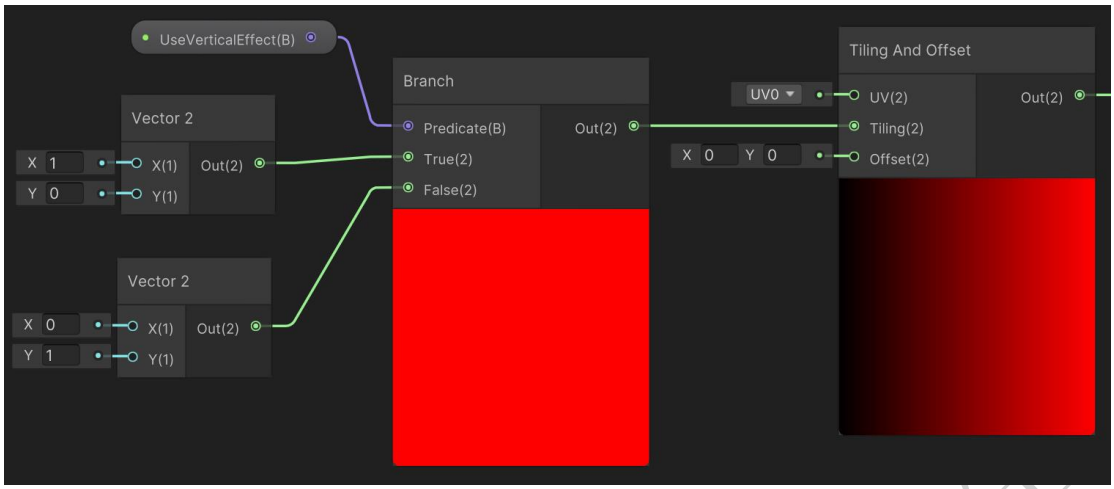


图185.　使用 Boolean 参数控制故障纹理

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。

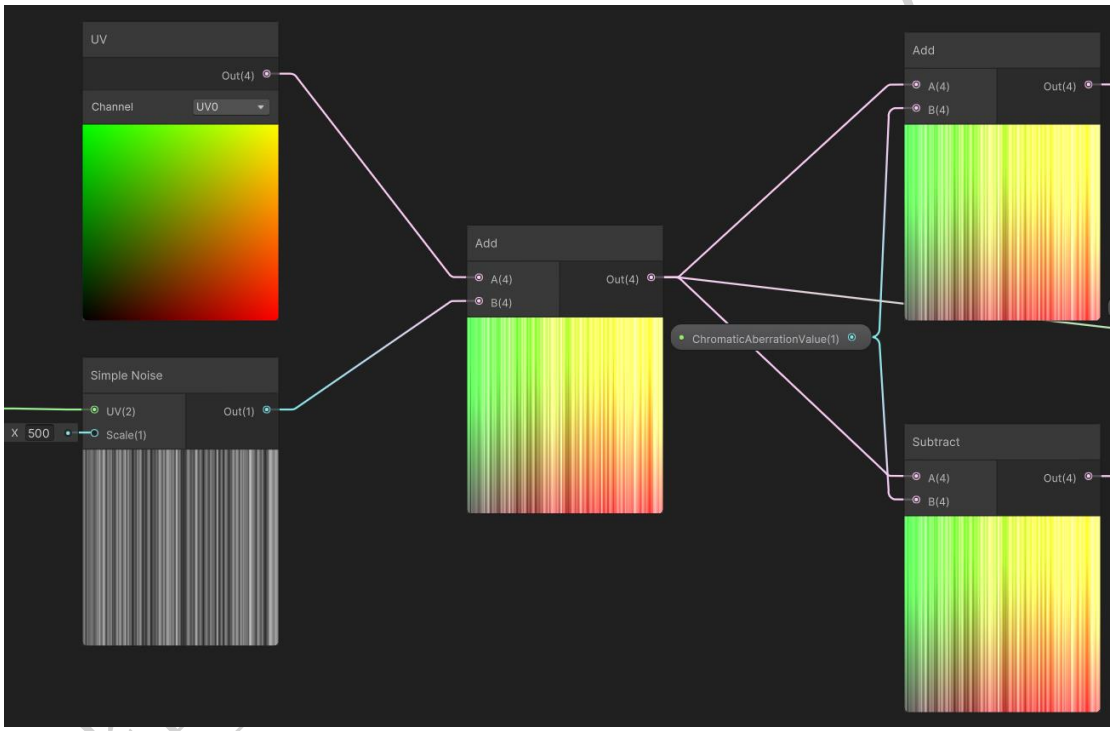


图186.　调整 UV，并将其用于色差

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。

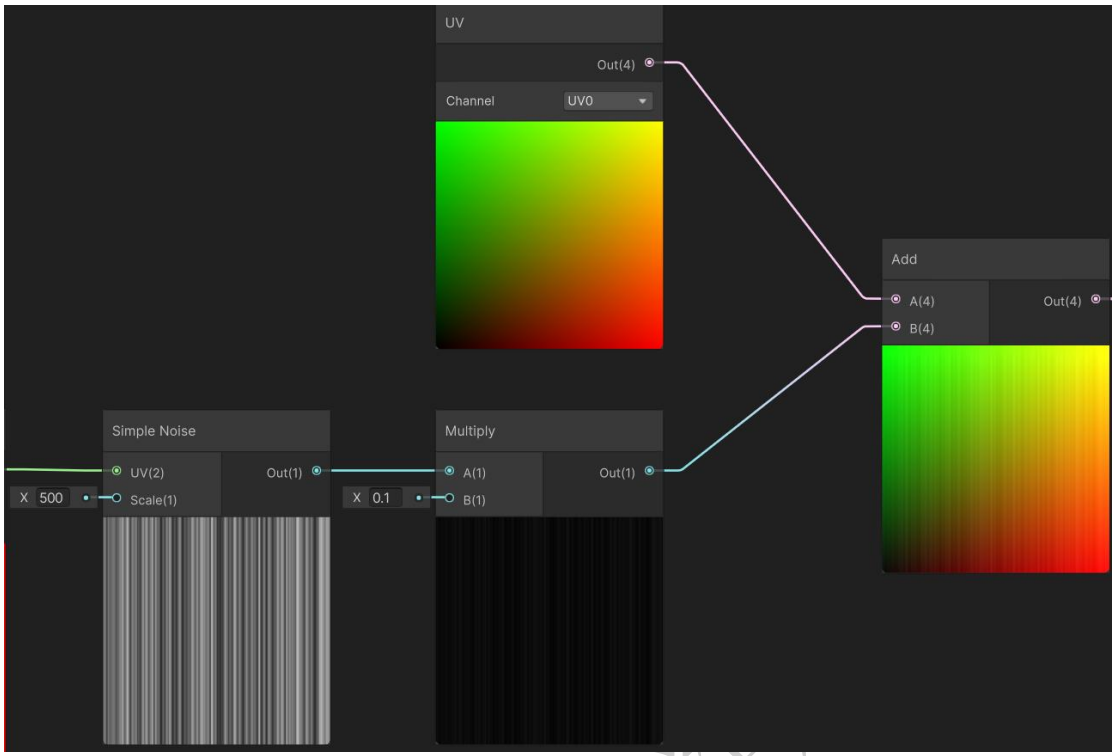


图187.　　减弱故障纹理的强度

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。

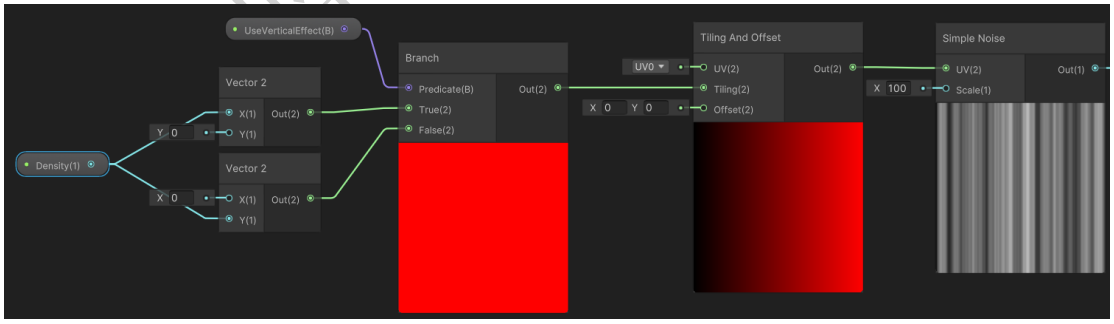


图188.　　调节故障纹理的密度

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。

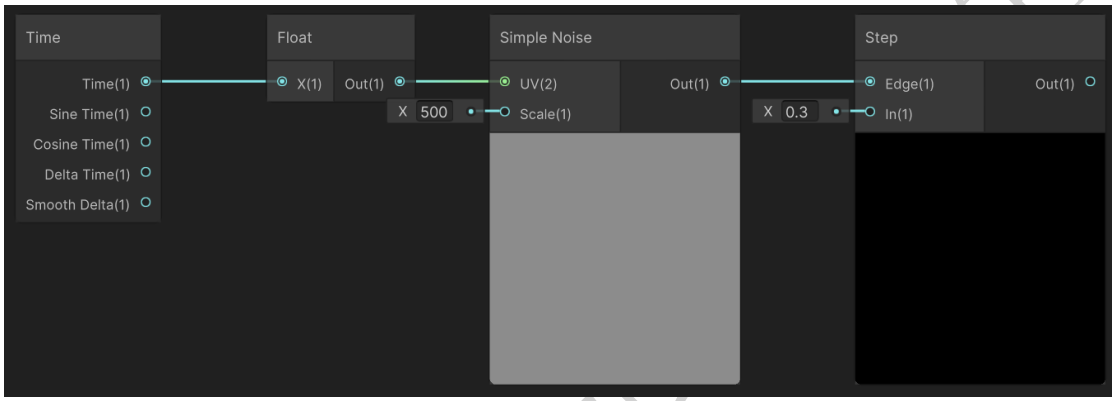


图189. 调整闪烁

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。

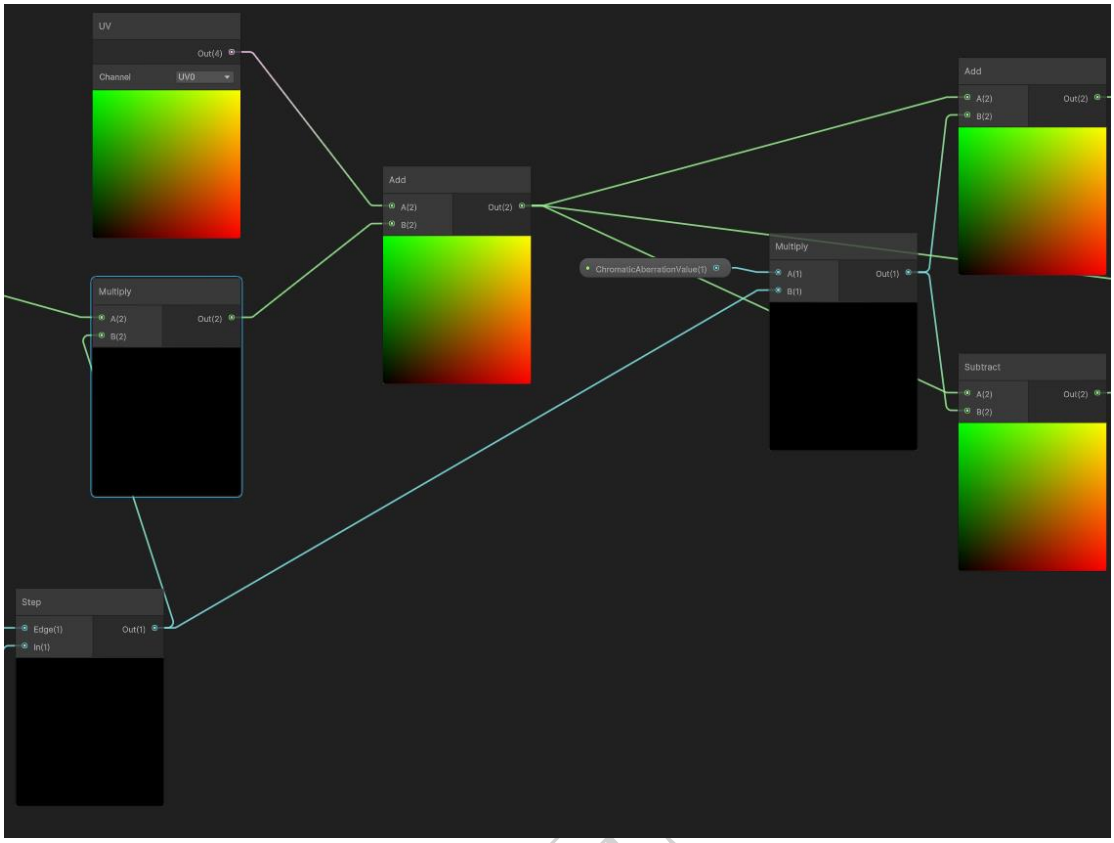


图190.　闪烁效果控制色差和故障的发生

十六、　2DSDF 圆

符号距离函数（sign distance function），简称 SDF，又可以称为有向距离函数，在空间中的一个有限区域上确定一个点到区域边界的距离并同时对距离的符号进行定义：点在区域边界外部为正，内部为负，位于边界上时为 0。

简单来说，SDF 就是某一点到某一个区域边界的最短距离。

求某一点到圆的边的距离是最为简单的，这里分两种情况。大致来说都是一样的，就是用点与圆心的距离减去半径来进行判别。

1. 圆心在原点

圆心在原点时，我们只需要知道两个参数，任意点 P 的坐标和圆的半径 r，就可以求出点 P 距离圆边缘的距离。

哔哩哔哩欢乐

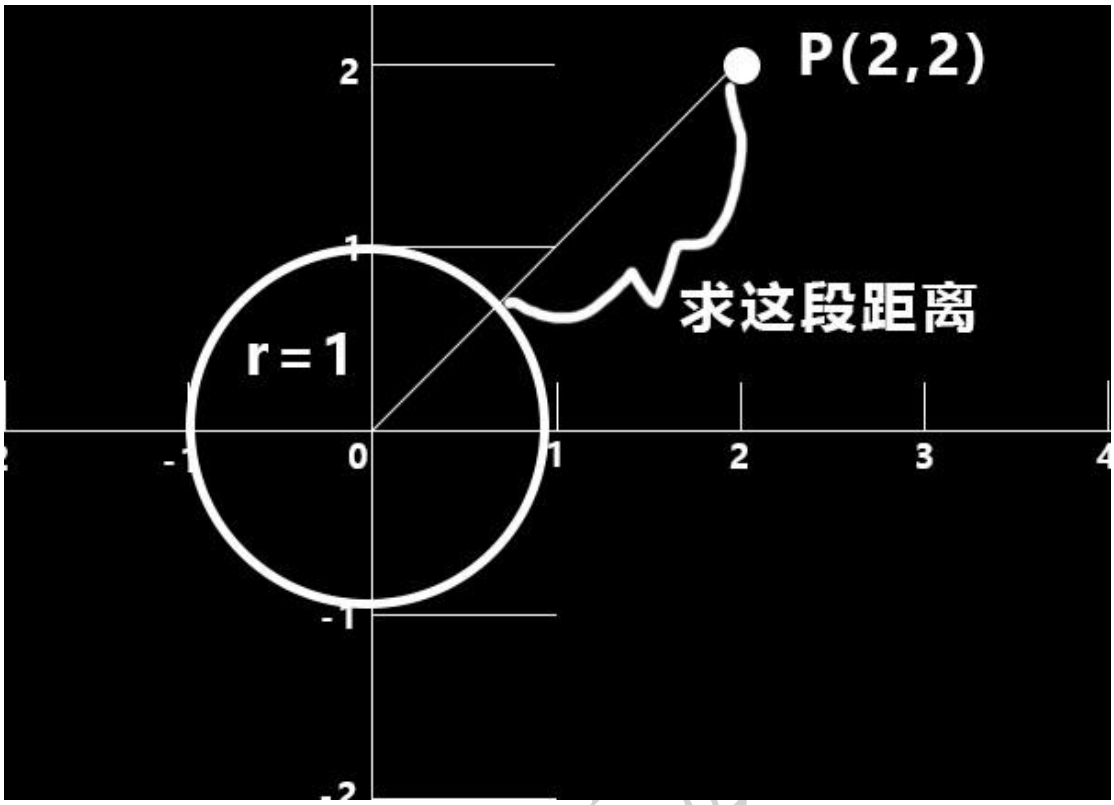


图191.　　圆心在坐标轴原点

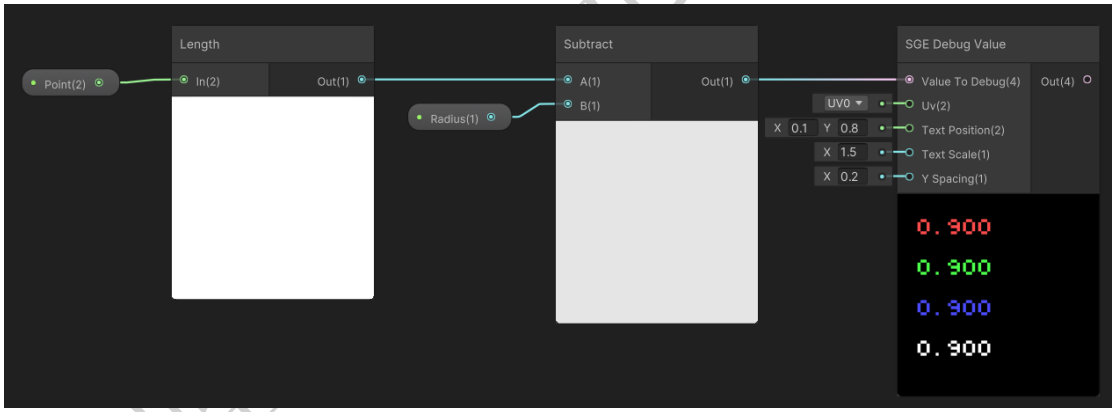


图192.　　圆心在原点时的 ShaderGraph

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。）

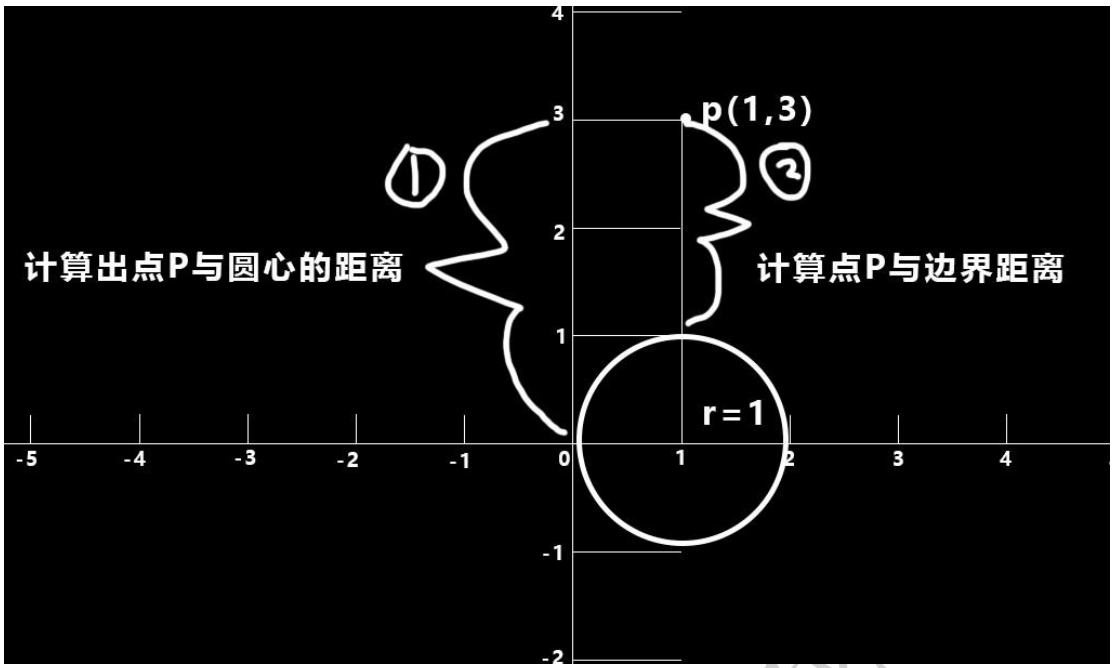


图193.　　圆心不在坐标轴原点

图194.　　圆心不在原点时的两种连接方法

十七、　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。

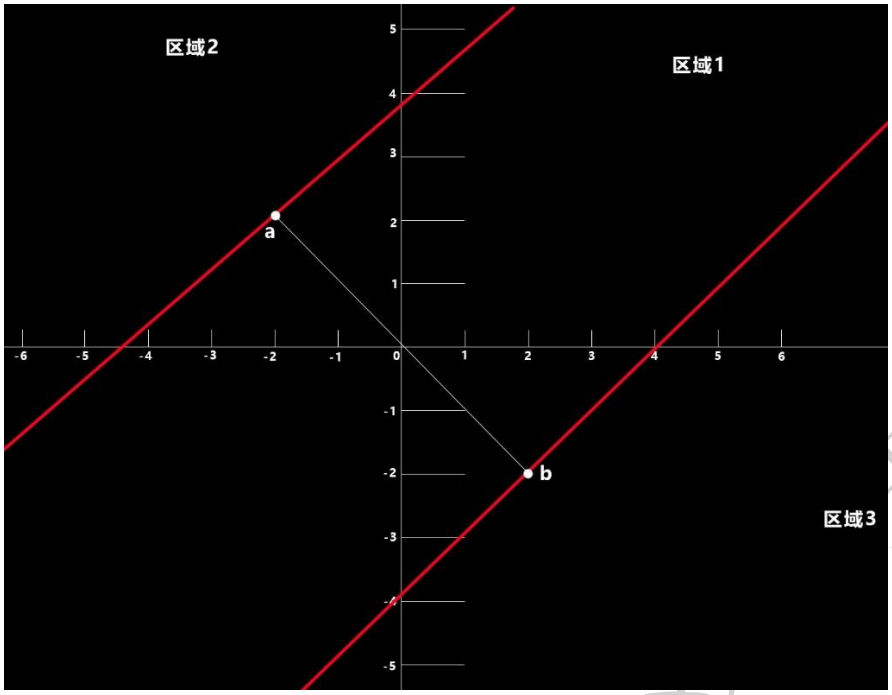


图195.　　SDF 线段区域划分

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

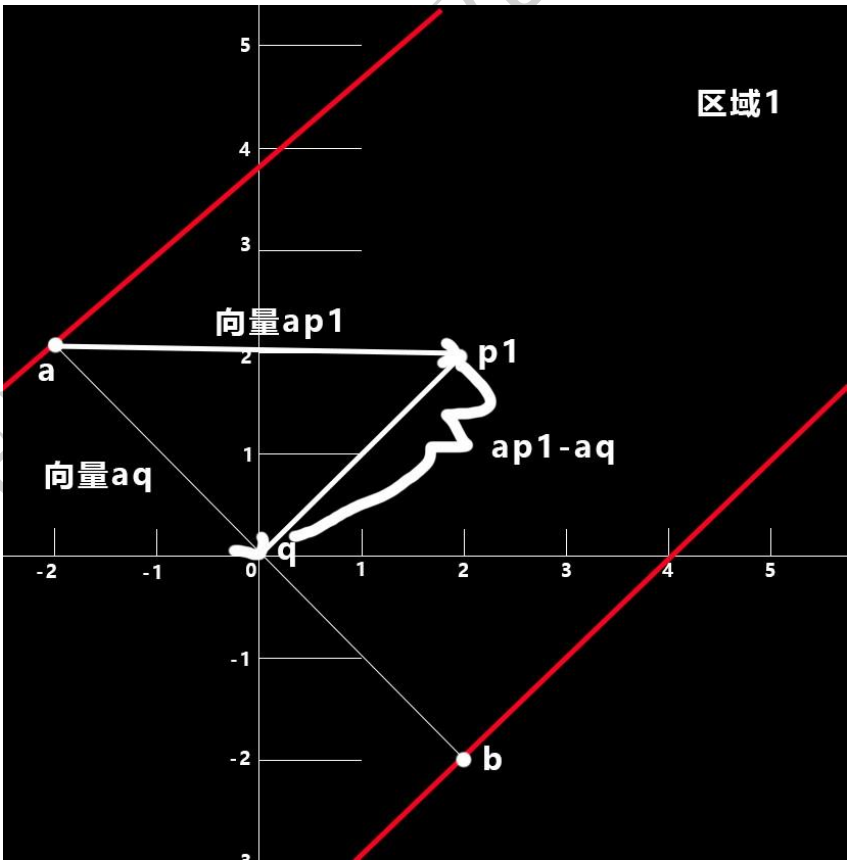


图196.　　点位于区域一内

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，$\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[16]$\overrightarrow{ab}$。（r[17]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

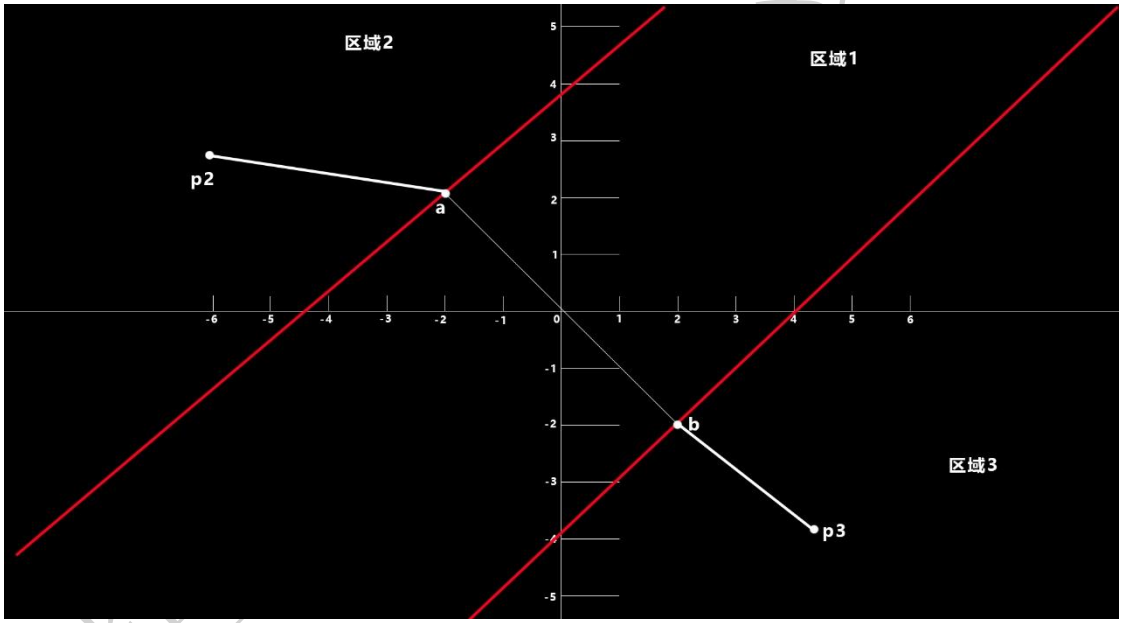


图197.　　点位于区域二和三

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=$\|\overrightarrow{qp_1} = \overrightarrow{ap_1} - \overrightarrow{aq} = \overrightarrow{ap_1} - r$[18]$\overrightarrow{ab}\|$

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

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

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

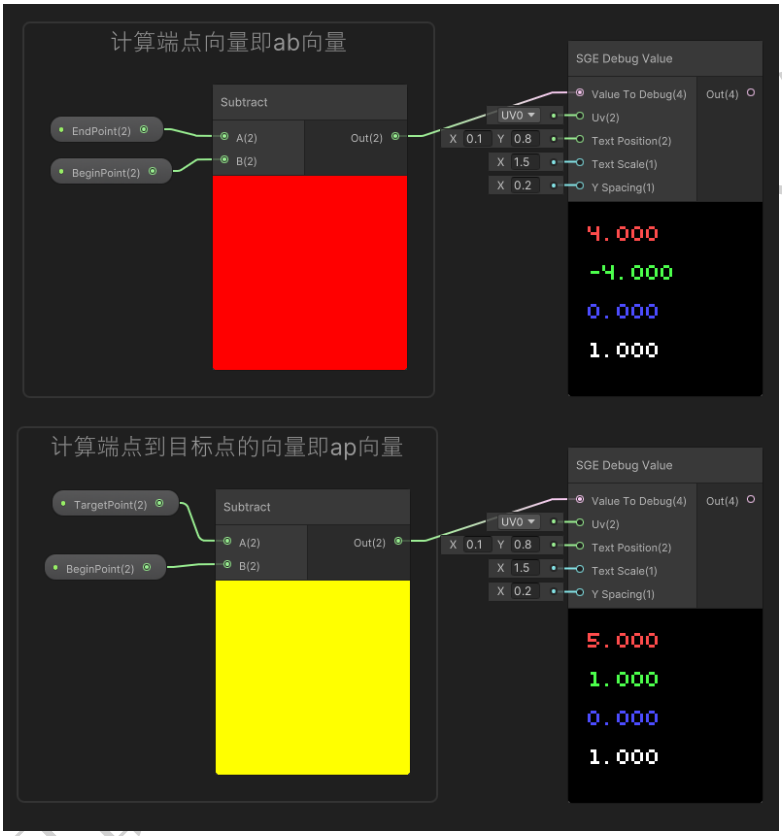


图198.　　计算得到两个向量

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。

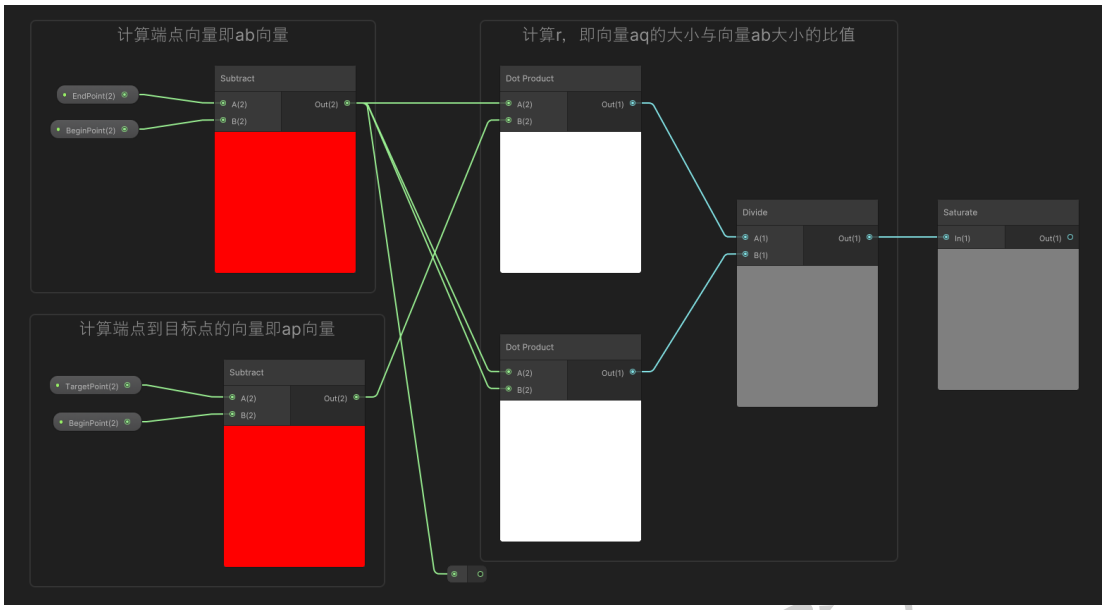


图199.　　计算 r

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。

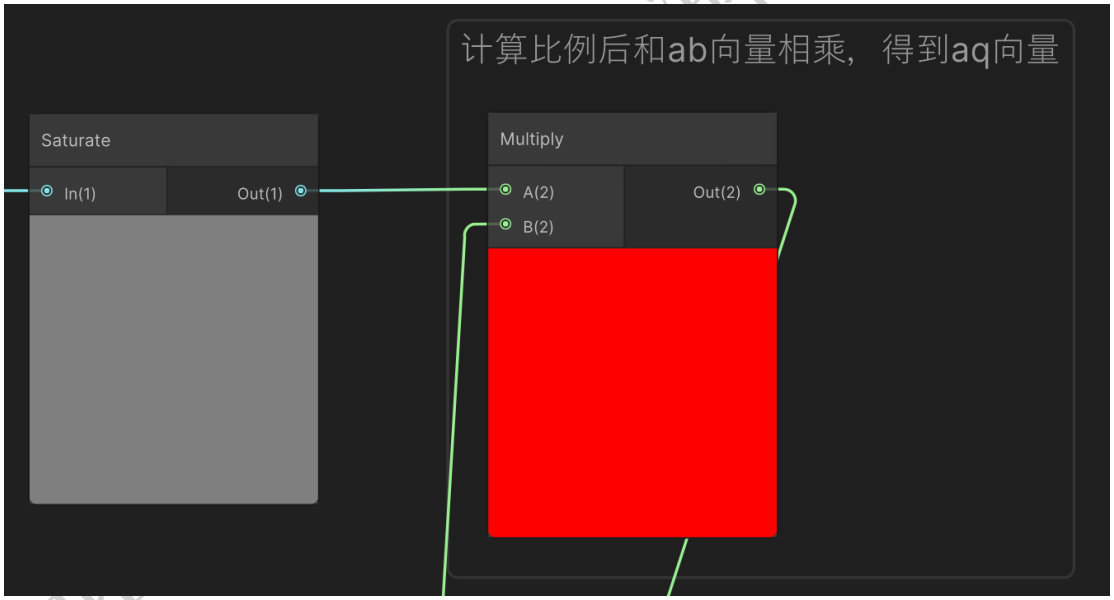


图200.　　得到 aq 向量

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）。

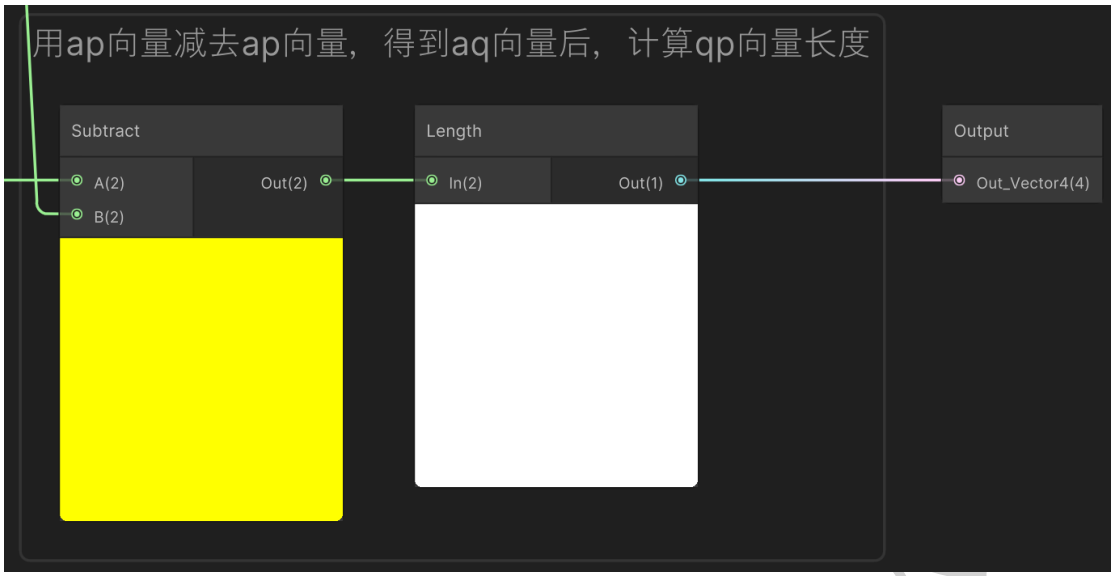


图201.　计算 qp 向量长度

十八、　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。

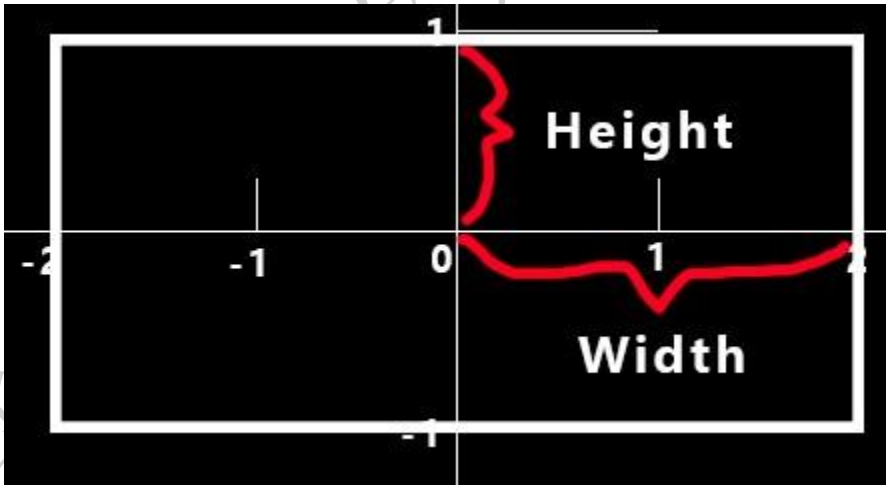


图202.　矩形绘制原理

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。

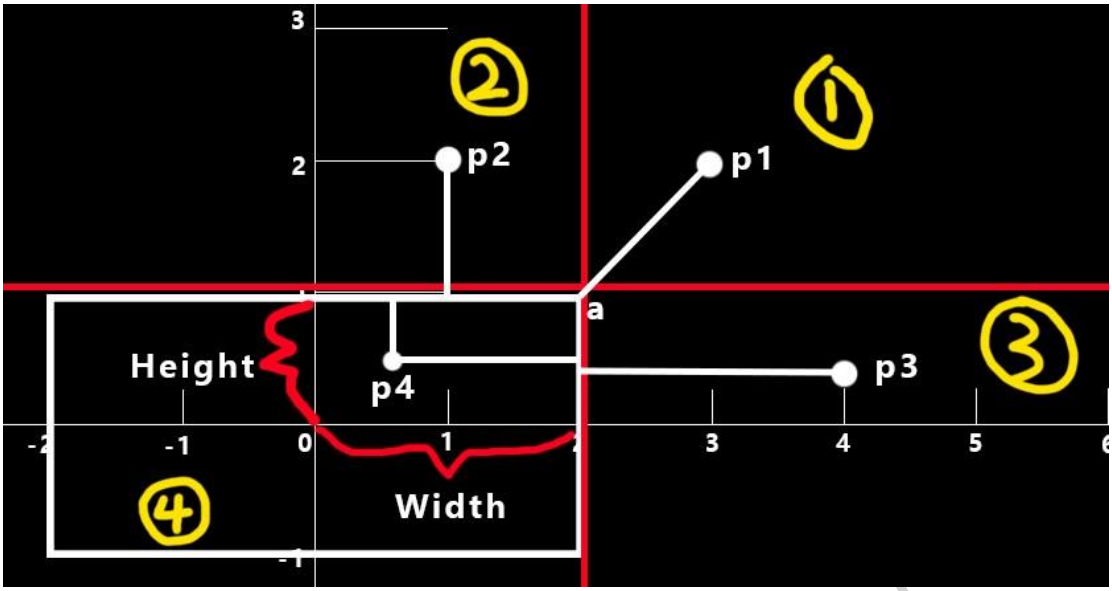


图203.　　目标点到矩形的情况划分

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。

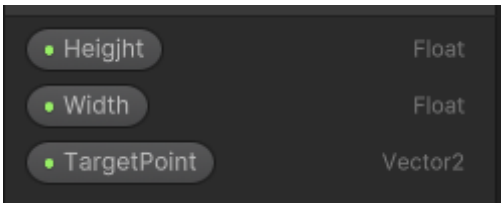


图204.　SDF 矩形参数设置

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。

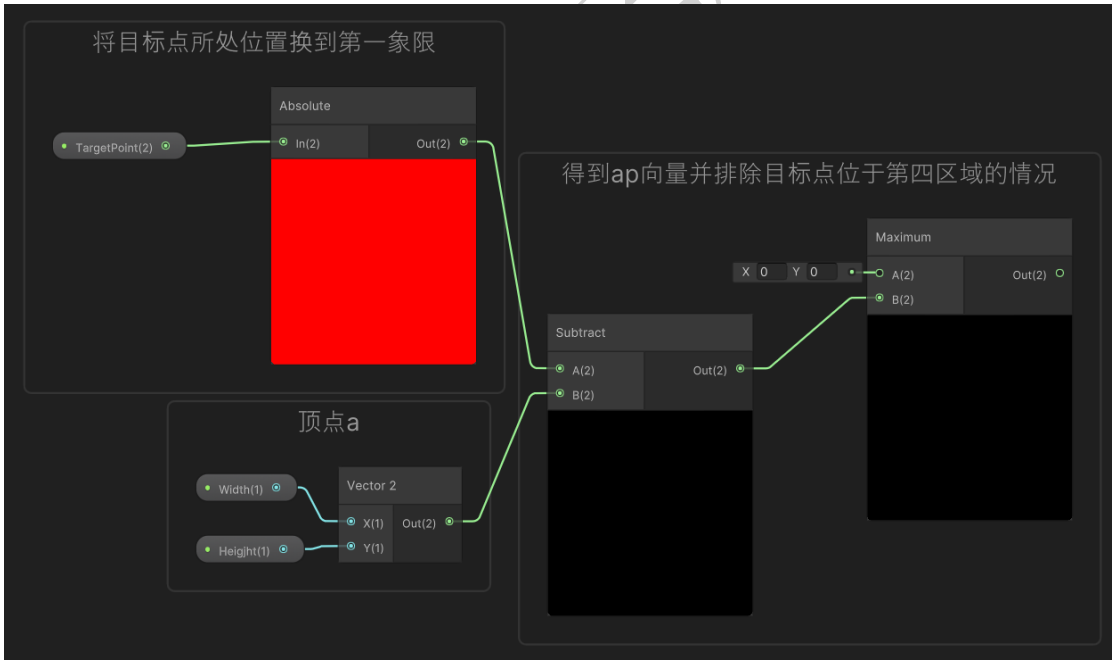


图205.　计算向量 ap1

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。

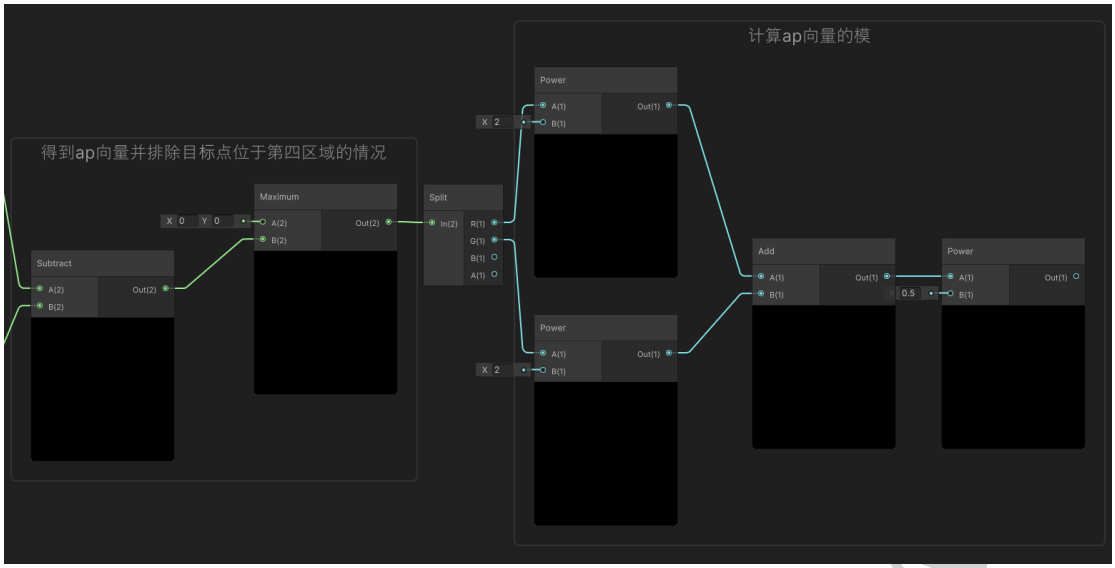


图206.　　计算目标点位于区域一、二、三时的模

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）

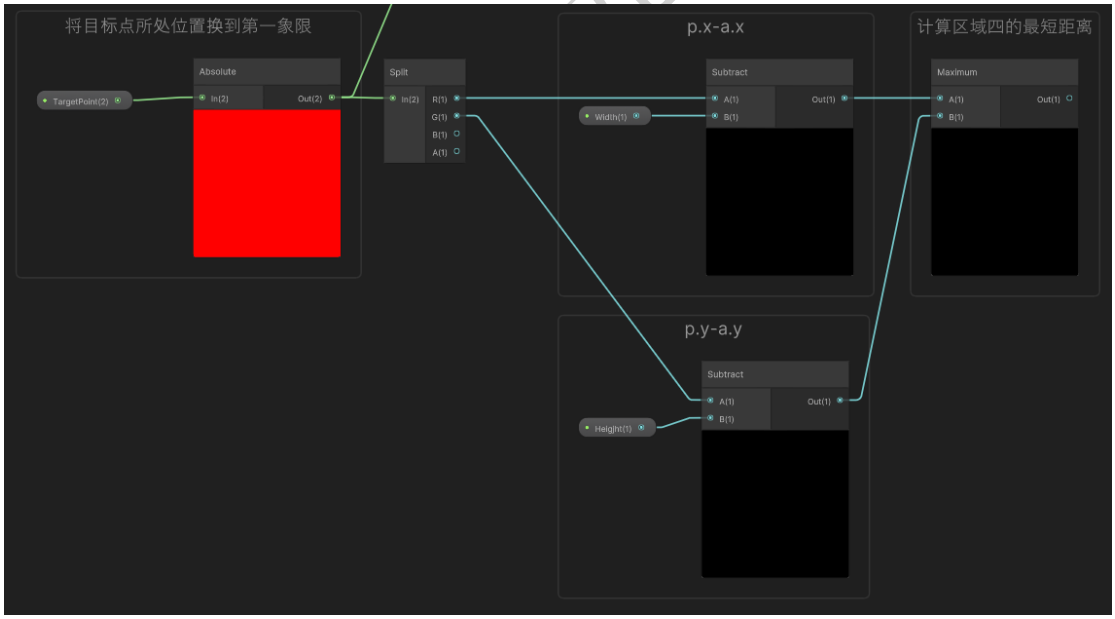


图207.　　计算区域四中点的最短距离

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。

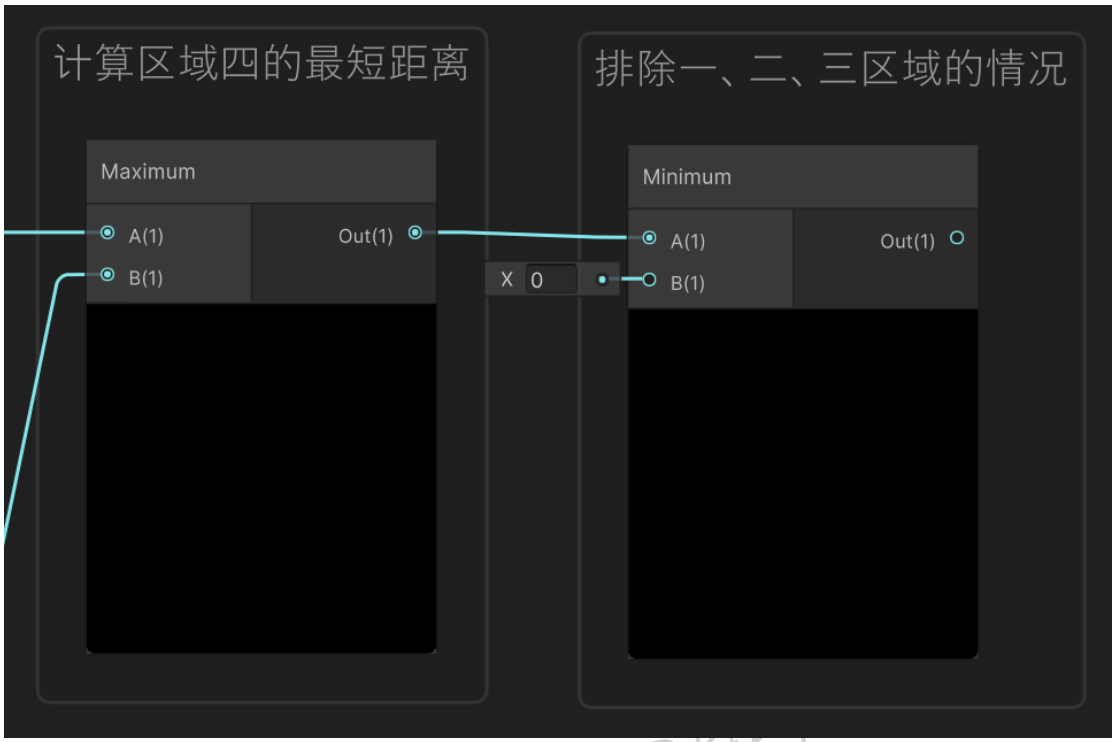


图208.　　排除目标点不位于区域四的情况

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。

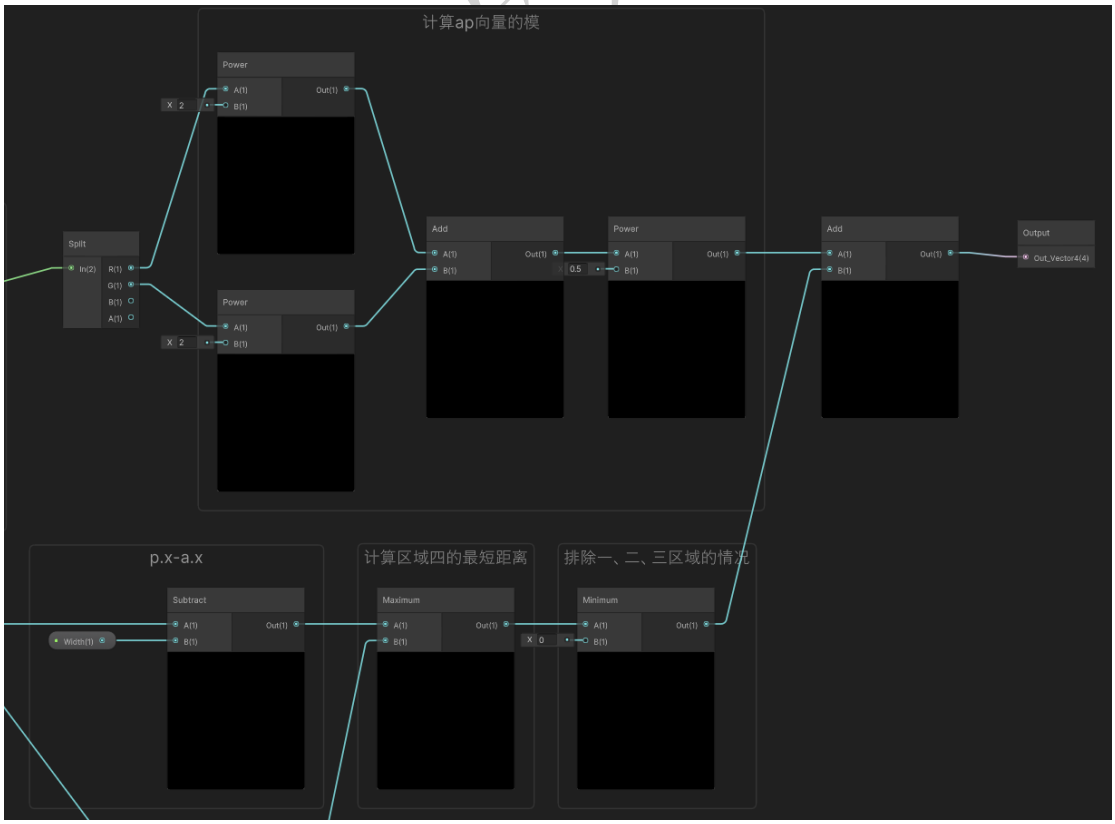


图209.　　SDF 矩形最终结果

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。

十九、 3D 描边效果

二十、 3D 积雪效果

二十一、 3D 广告牌效果

哔哩哔哩次元游戏人生

Shader Lab

一、 Shader 概念

1. 什么是 Shader

① GPU 流水线上一些可高度编程的阶段，而由着色器编译出来的最终代码是会在 GPU 上运行的（对于固定渲染管线的渲染来说，着色器有事等同于一些特定的渲染设置）；

② 有一些特定类型的着色器，如顶点着色器、片元着色器等；

③ 依靠着色器我们可以控制流水线中的渲染细节，例如用顶点着色器来进行顶点变换以及传递数据，用片元着色器来进行逐像素的渲染。

Shader 中文名称叫“着色器”,字面意思就是“给物体上色的机器”。它也是通过编写代码实现的，因此本质上也是程序。只不过 Shade 程序跟普通程序不一样，它并不是运将在 CPU 上，而是运行在 GPU 上,Shader 的目的是为了告诉 GPU 如何计算和输出图像。

Shader 所处的阶段只是渲染流水线中的一部分，它主要由顶点着色器(Vertex Shader)和片段着色器(Fragment Shader)组成。

编写 Shader 的语言主要有两种:

(1)基于 Direct3D 的 HLSL,全称 High Level Shading Language，由微软开发,在自家的 Windows 平台上兼容性非常好,因此成为游戏开发的首选。

(2)基于 OpenGL 的 GLSL,全称 OpenGL Shading Language,具有良好的移植性,可以在不同平台使用。

NIVIDIA 希望显卡的程序开发独立于 DX 和 GL 的图形软件库,于是联合微软共同开发了 CG 语言(C for Graphics)。因为它是在 HLSL 的基础上进行开发的，所以其语法跟 HILSL 非常相似。并且，使用 CG 语言编写的 Shader 拥有跨平台性。因此 CG 语言是编写 UnityShader 的首选语言。

2. Shader 和 Material（材质）的关系与区别

Shader 实际上就是一段程序,它负责把输人的顶点数据按照代码里指定的方式进行处理,并对输入的颜色或者贴图等进行计算,然后输出数据。图像绘制单元获取到输出的数据便可将图像绘制出来,最终呈现在屏幕上。

Shader 程序代码，再加上开放的参数设置以及关联的贴图等资源，为实现某种效果而打包存储在一起，最终得到的就是材质(Material)。Material 是 Shader 的实例化资源，一个 Shader 可以实例化为多个材质,并且调节为不同的材质效果。最后把材质指定给某个模型就可以渲染出对应的效果了。

3. Unity 中的 Shader

Unity 一共提供了 6 种 Unity Shader 模板供我们选择——Standard Surface Shader，Unlit Shader，Image Effect Shader，Custom Render Texture，Compute Shader 以及 Ray Tracing Shader。

其中，Standard Surface Shader 会产生一个包含了标准光照模型的表面着色器模板，Unlit Shader 则会产生一个不包含光照（但包含雾效）的基本的顶点/片元着色器，Image Effect Shader 则为我们实现各种屏幕后处理效果提供了一个基本模板。最后，Compute Shader 会产生一种特殊的 Shader 文件，这类 Shader 旨在利用 GPU 的并行性来进行一些与常规渲染流水线无关的计算。

唯唯咩咩游戏人生

二、 简单概念

1. 顶点-Vertices

对象的顶点对应于定义二维或三维空间中表面区域的点集。在 Maya 和 Blender 中，顶点表示为网格和对象的交点。

这些要点有两个主要特点：

① 它们是变换组件的子级。

② 它们具有根据对象总体积的中心定义的位置。

这是什么意思？假设在 Maya 3 中，有两个默认节点与一个对象关联。这些被称为变换和形状。与 Unity 中一样，变换节点定义对象围绕对象轴心的位置、旋转和缩放。相反，形状节点（变换节点的子节点）包含几何属性，即对象的顶点与其体积相关的位置。这意味着我们可以移动、旋转或缩放对象的一组顶点，但同时，我们可以更改特定点或顶点的位置。

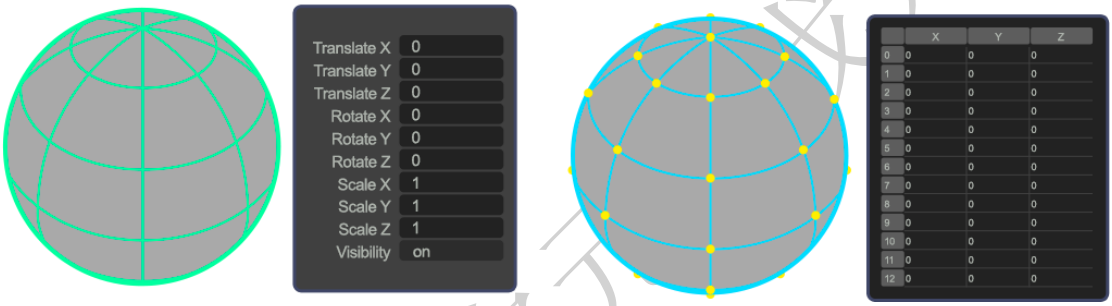


图210. 左边是来自 Maya 的 Transform 节点，右边是 Shape 节点

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

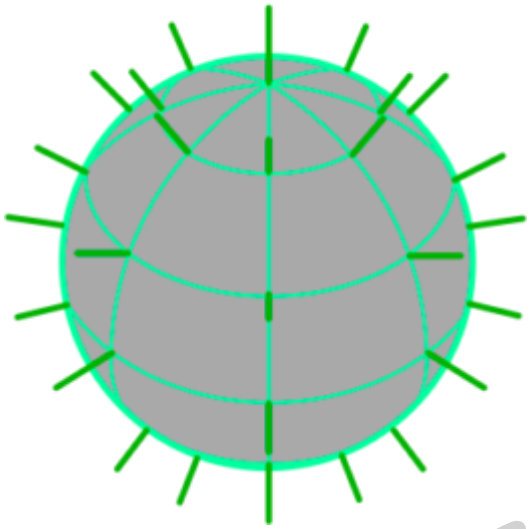


图211.　每个顶点法线的图形表示

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。

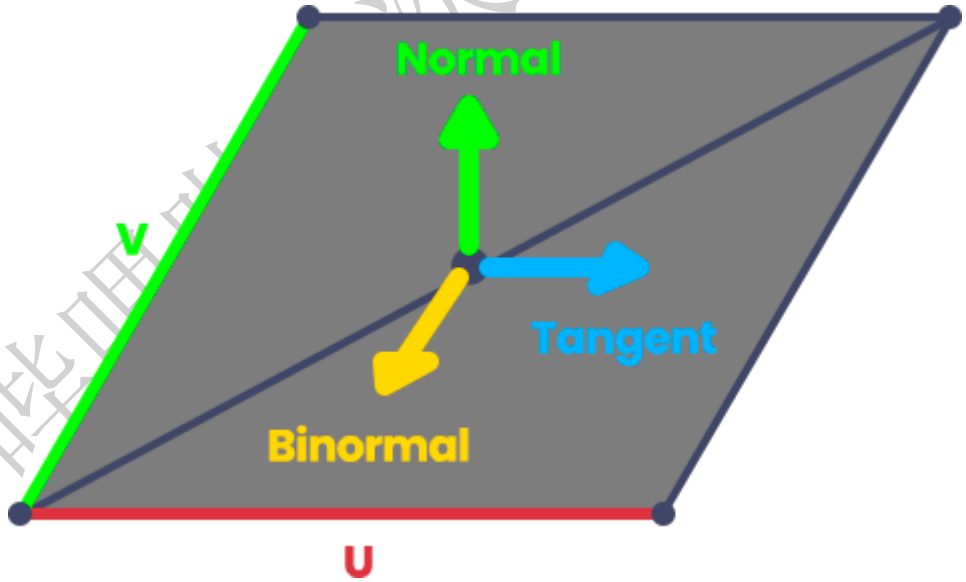


图212.　默认情况下，我们无法在着色器中访问 Binormal，我们必须根据法线和切线计算它。

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。

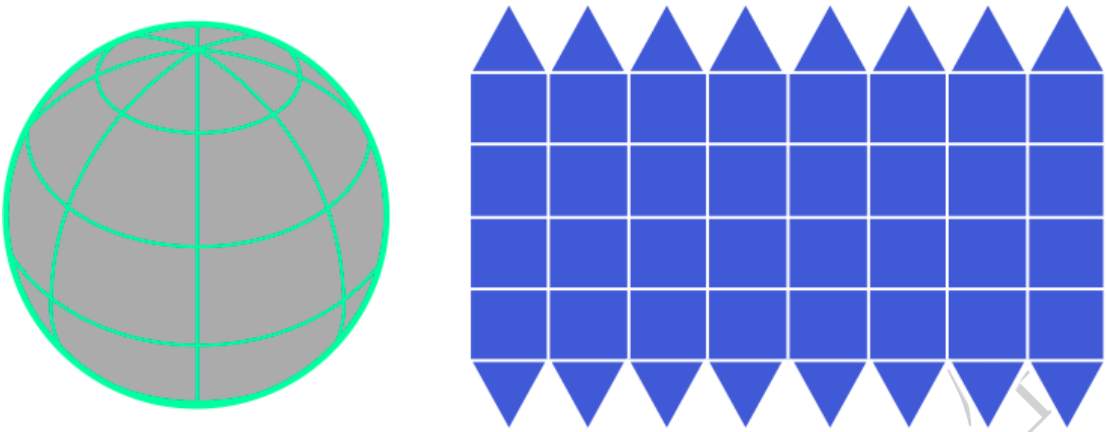


图213.　顶点可以在 UV 贴图中以不同的方式排列

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。

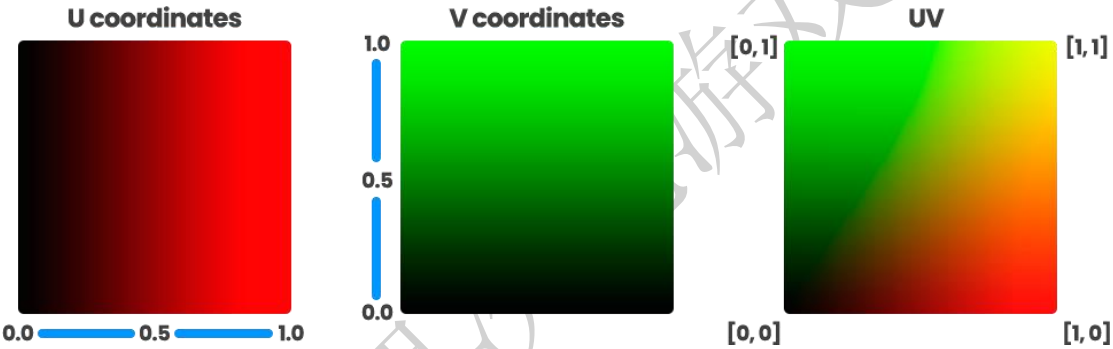


图214.　笛卡尔平面中 UV 坐标的图形参考

三、　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。

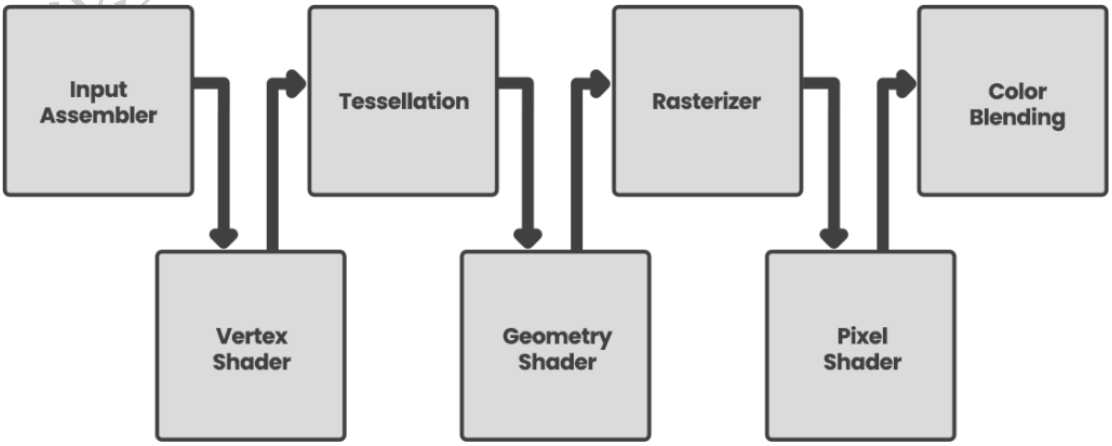


图215.　逻辑渲染管道

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。

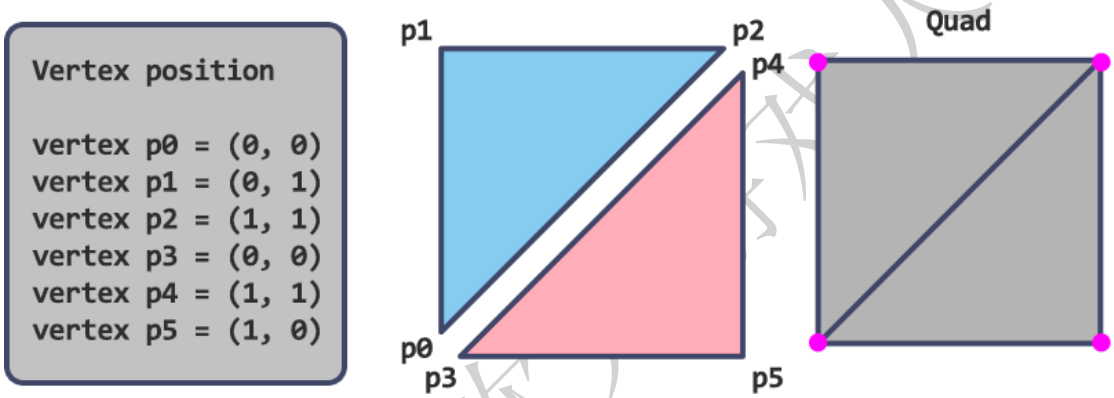


图216. 渲染图元

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。

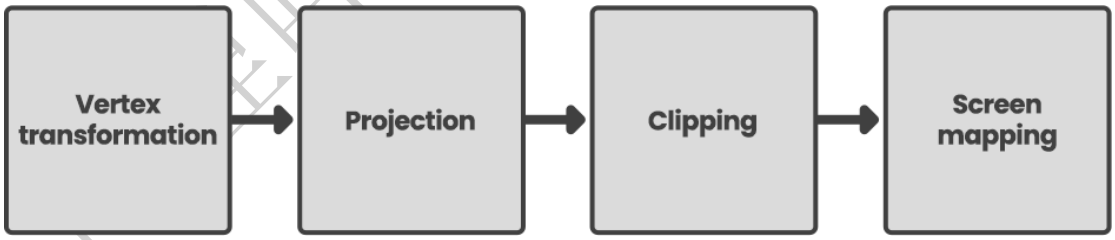


图217. 几何阶段子过程

几何阶段负责和每个谊染图元打交道，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。

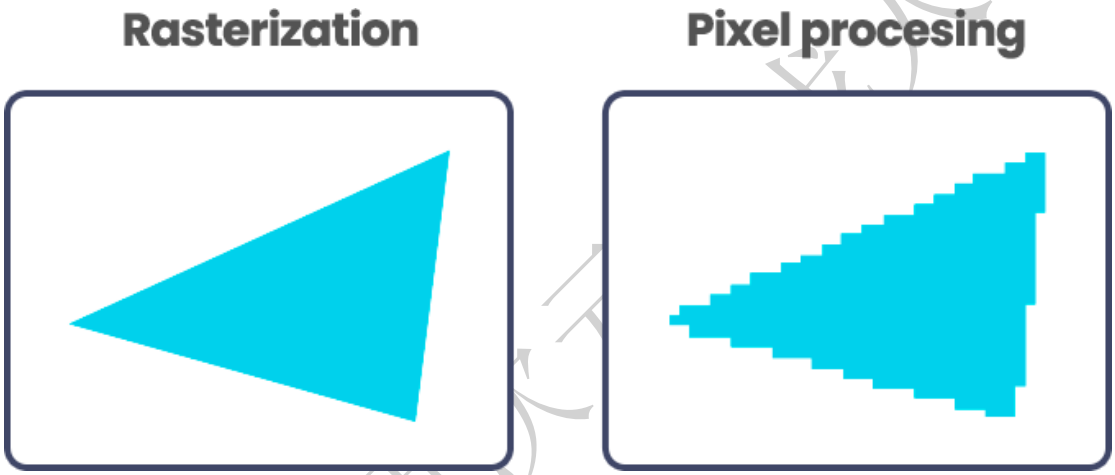


图218. 几何图形覆盖的区域在屏幕上转换为像素

四、 CPU 和 GPU 之间的通信

渲染流水线的起点是 CPU,即应用阶段。应用阶段大致可分为下面 3 个阶段：

① 把数据加载到显存中

② 设置渲染状态

③ 调用 DrawCall

1. 把数据加载到显存中

所有渲染所需的数据都需要从硬盘(Hard Disk Drive，HDD)中加载到系统内存(Random Access Memory,，RAM)中。然后，网格和纹理等数据又被加载到显卡上的存储空间——显存(Video Random Access Memory,，VRAM)中。这是因为，显卡对于显存的访问速度更快，而且大多数显卡对于 RAM 没有直接的访问权利。

当把数据加载到显存中后，RAM 中的数据就可以移除了。但对于一些数据来说，CPU 仍然需要访问它们(例如，我们希望 CPU 可以访问网格数据来进行碰撞检测)，那么我们可能就不希望这些数据被移除，因为从硬盘加载到 RAM 的过程是十分耗时的。

在这之后，开发者还需要通过 CPU 来设置渲染状态，从而“指导”GPU 如何进行渲染工作。

2. 设置渲染状态

什么是渲染状态呢?一个通俗的解释就是，这些状态定义了场景中的网格是怎样被渲染

的。例如，使用哪个顶点着色器(Vertex Shader)/片元着色器(Fragment Shader)、光源属性、材质等。如果我们没有更改渲染状态，那么所有的网格都将使用同一种渲染状态。

3. 调用 Draw Call

Draw Call 实际上就是一个命令，它的发起方是 CPU，接收方是 GPU。这个命令仅仅会指向一个需要被渲染的图元(primitives)列表，而不会再包含任何材质信息——这是因为我们已经在上一个阶段中完成了！

当给定了一个 Draw Call 时，GPU 就会根据渲染状态（例如材质、纹理、着色器等）和所有输入的顶点数据来进行计算，最终输出成屏幕上那些漂亮的像素。而这个计算过程就是 GPU 流水线。

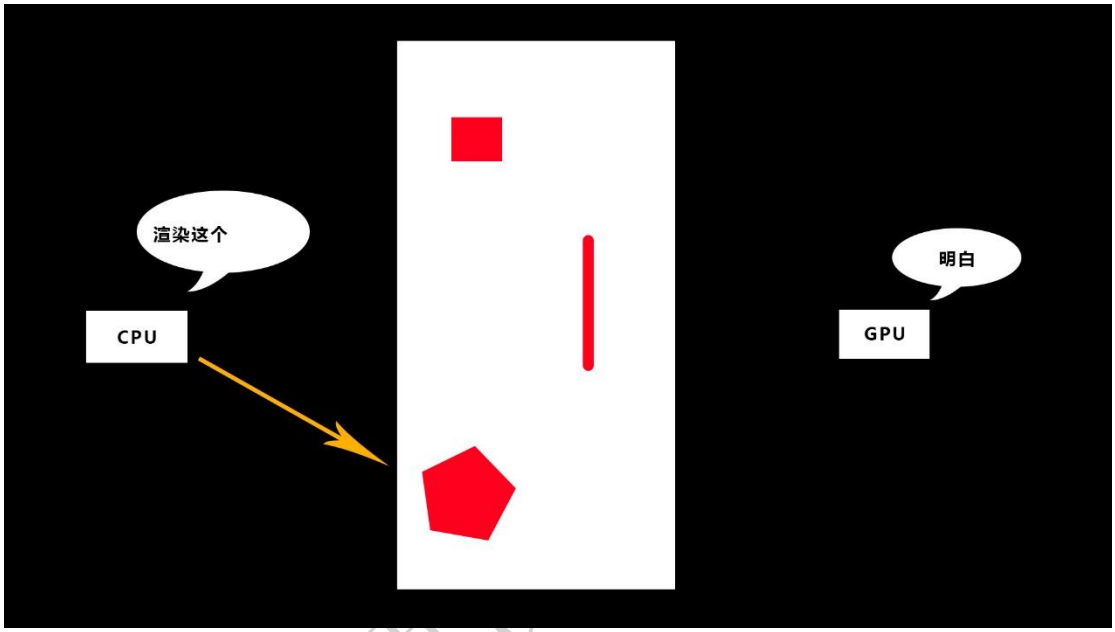


图219. Cpu 通过调用 Draw Call 告知 GPU 开始进行一个渲染过程。

五、 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。

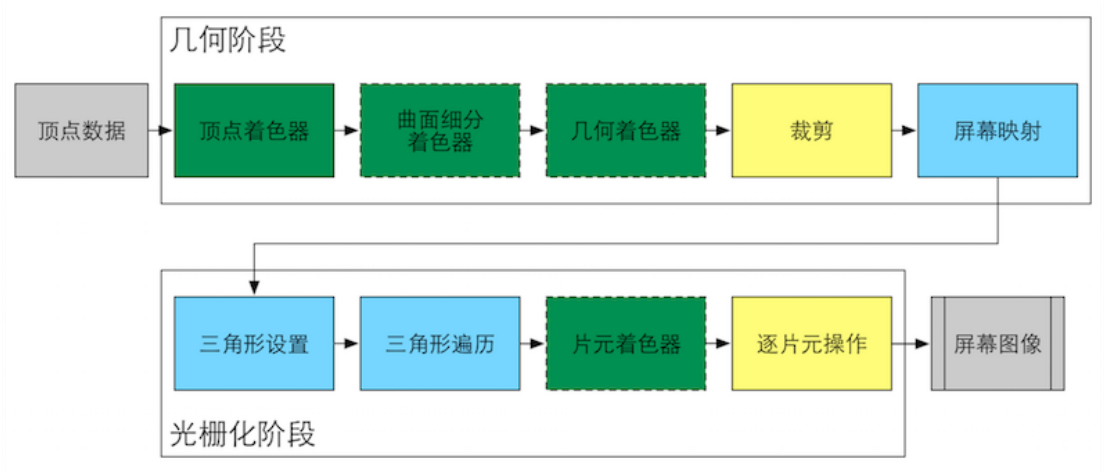


图220.　GPU 的渲染流水线实现

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。

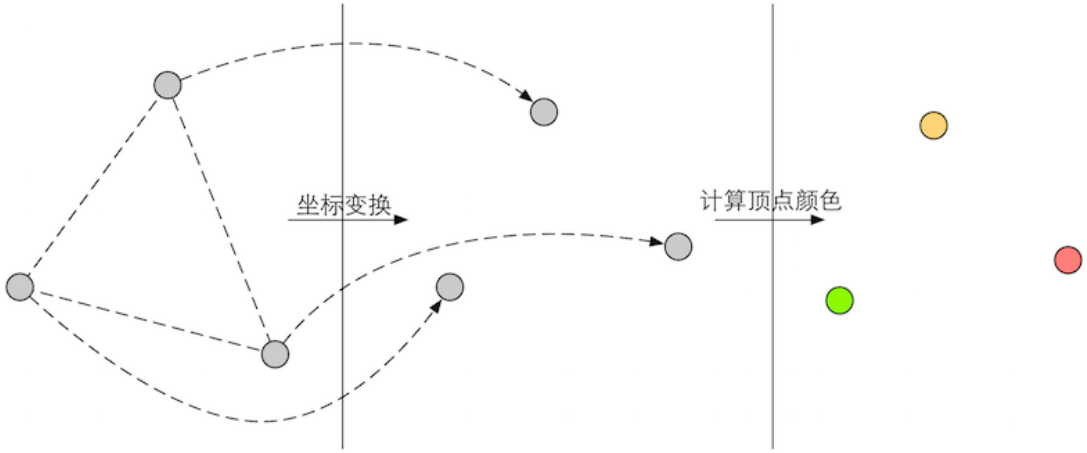


图221.　在顶点着色器中对顶点位置进行坐标变换并计算顶点颜色的过程

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222.　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。

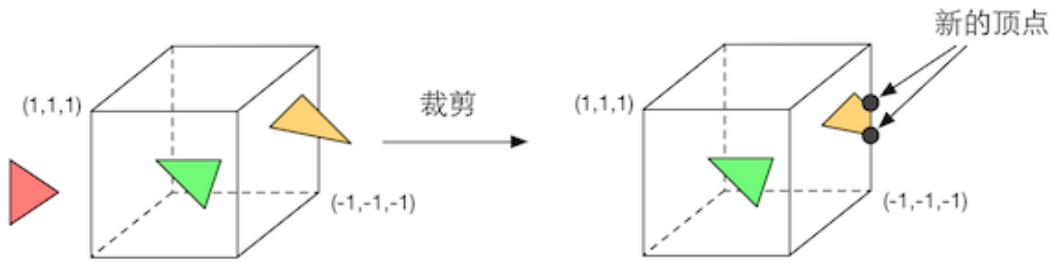


图223.　将图元裁剪到单位立方体内

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。

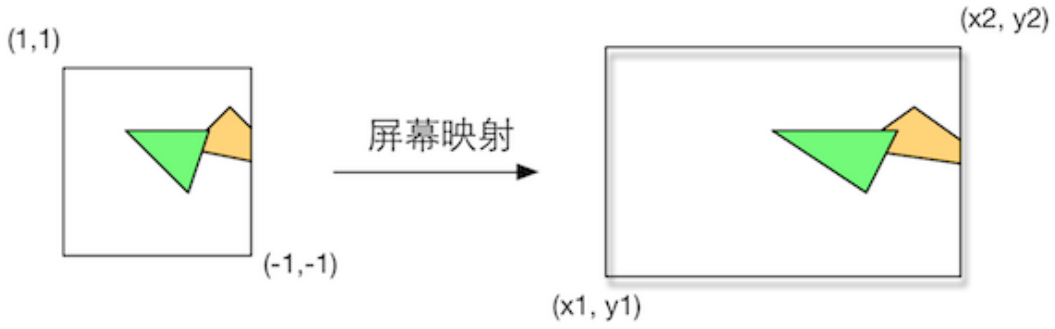


图224.　屏幕映射

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。

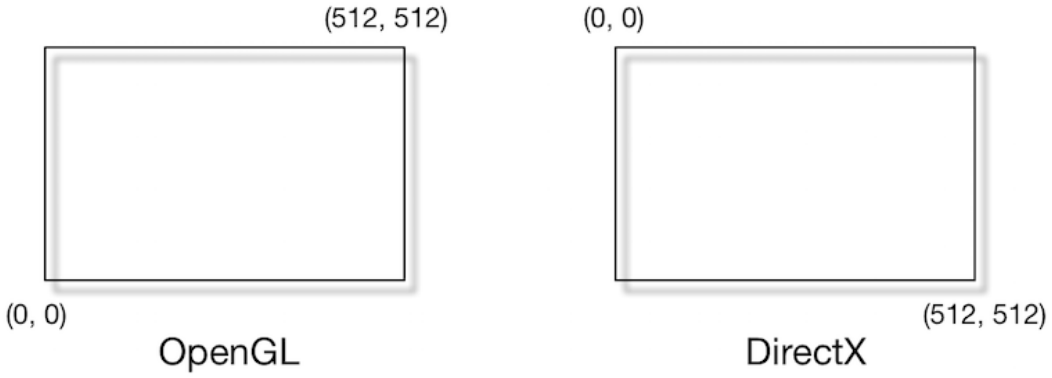


图225. OpenGL 和 DirectX 屏幕坐标系的差异

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顶点，即我们得到的是三角网格每条边的两个端点，但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。为了能够计算边界像素的坐标信息，我们就需要得到三角形边界的表示方式。这样一个计算三角网格表示数据的过程就叫做三角形设置。它的输出是为了给下一个阶段做准备。

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。

哔哩哔哩次元游戏人生

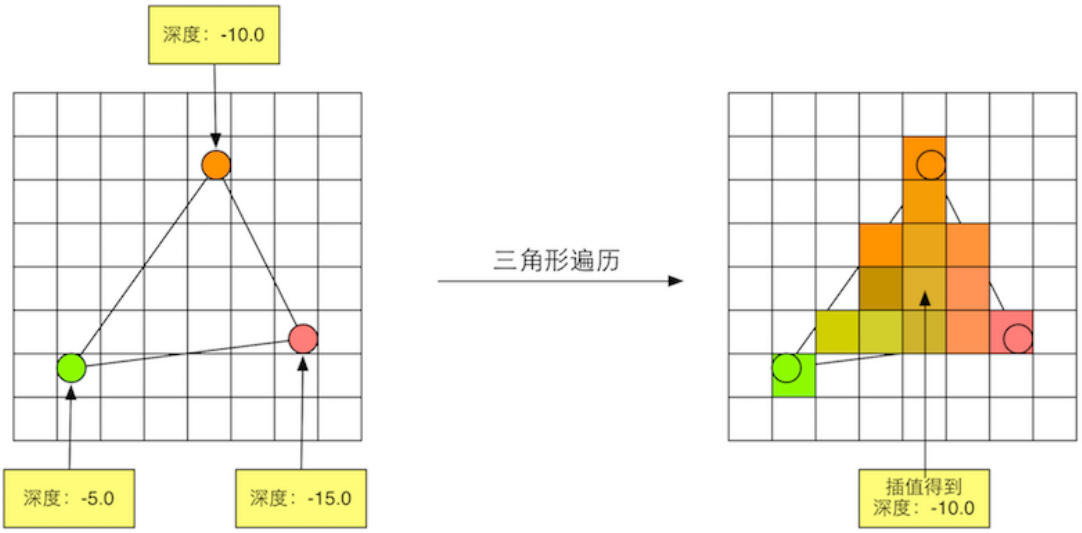


图226.　三角形遍历的过程

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。

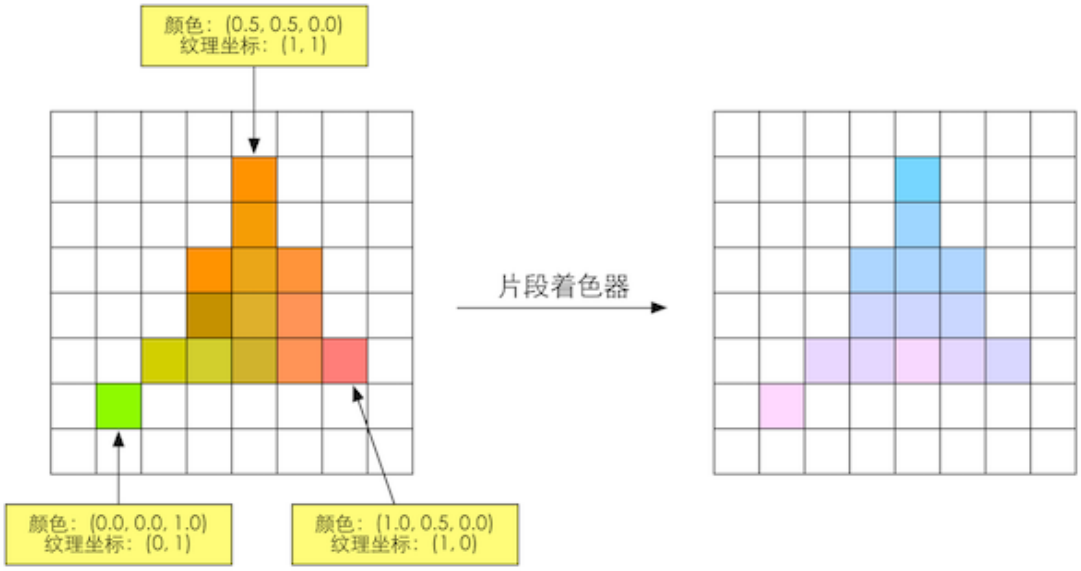


图227.　根据三角形遍历差值后的片元信息，片元着色器计算该片元的输出颜色

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228.　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。

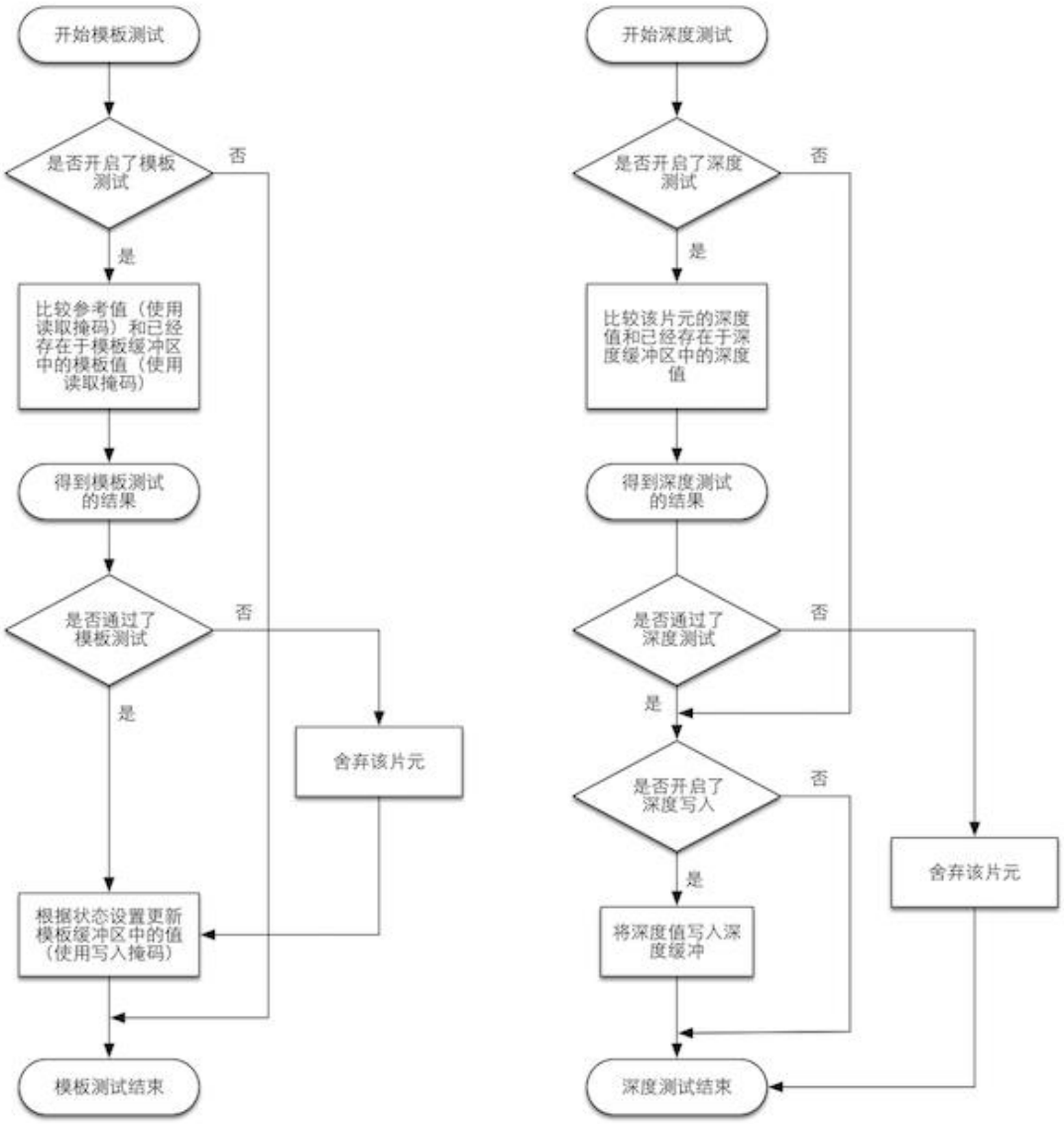


图229.　　模板测试和深度测试的简化流程图

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

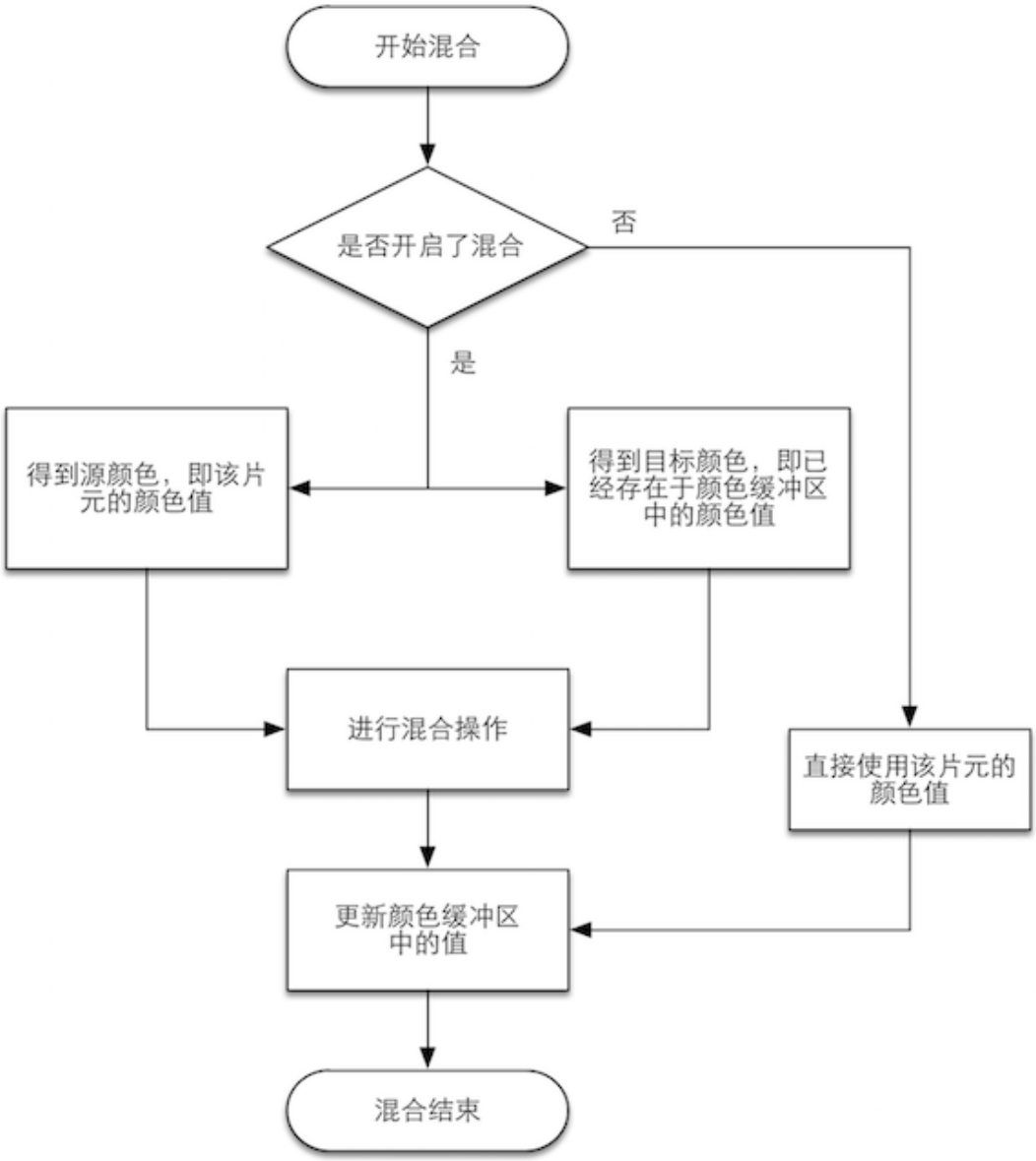


图230.　混合操作的简化流程图

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！

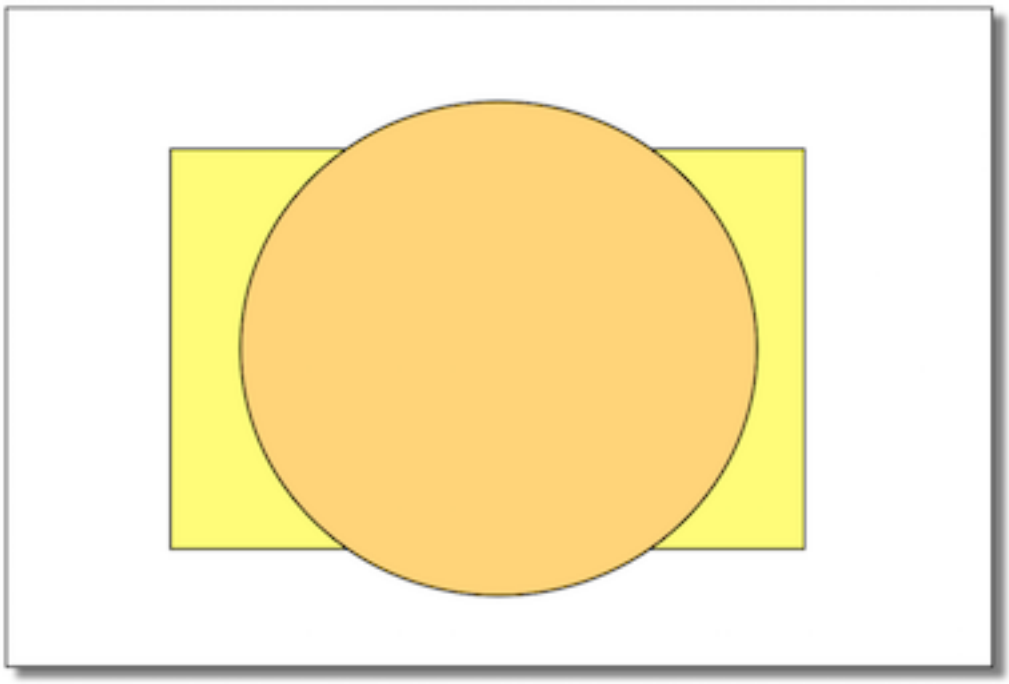


图231.　示例

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。

六、 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）。

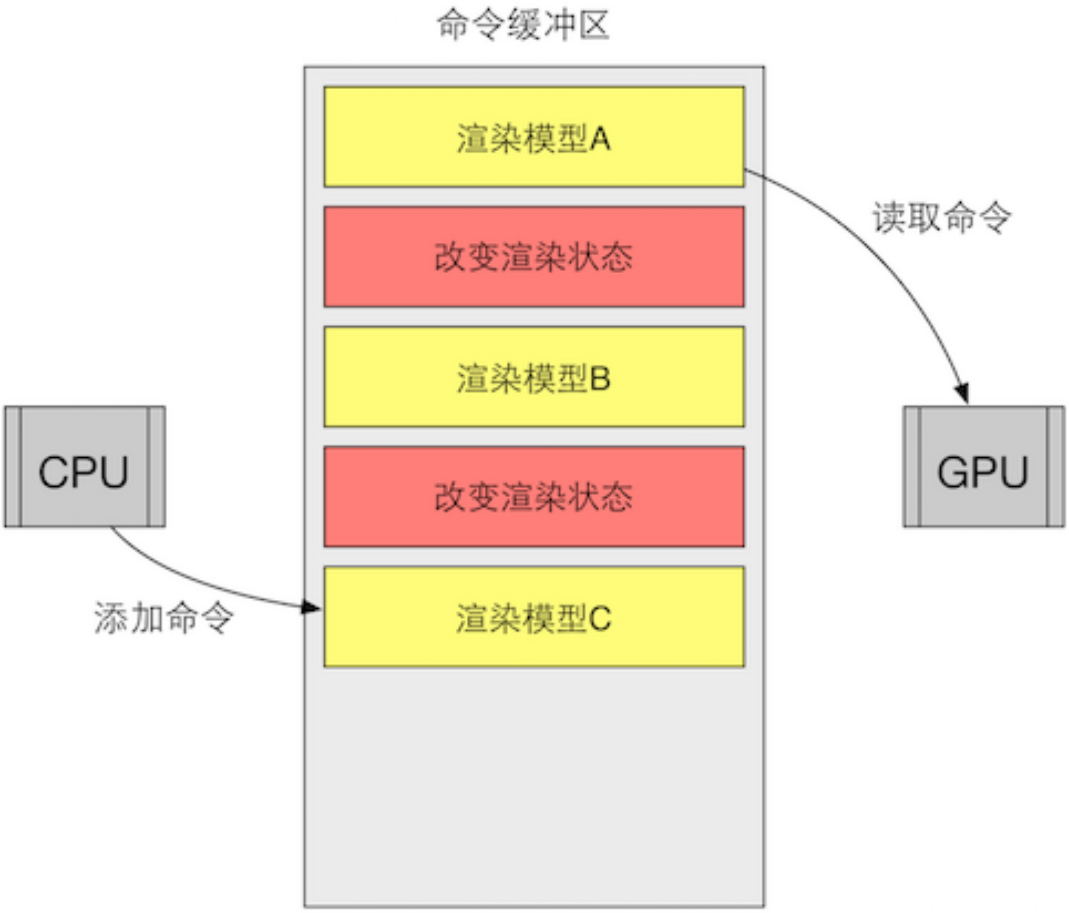


图232. 命令缓冲区

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。

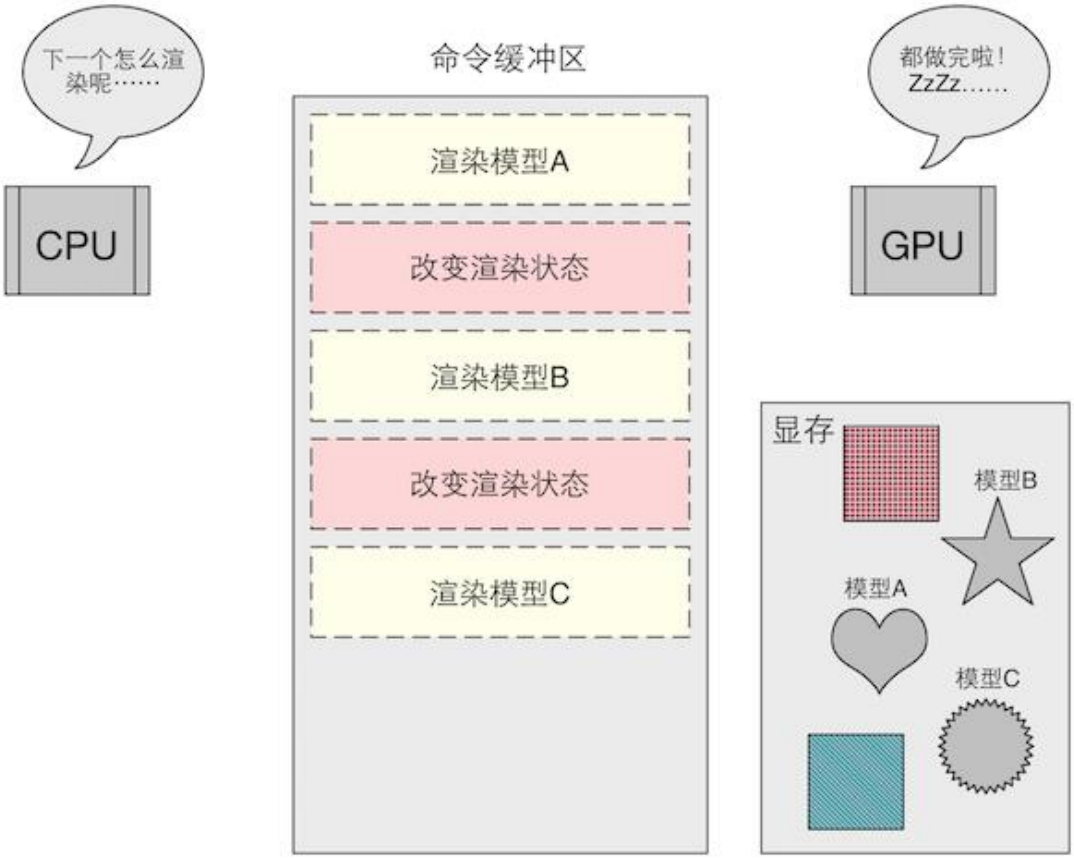


图233.　命令缓冲区中的虚线方框表示 GPU 已经完成的命令

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。

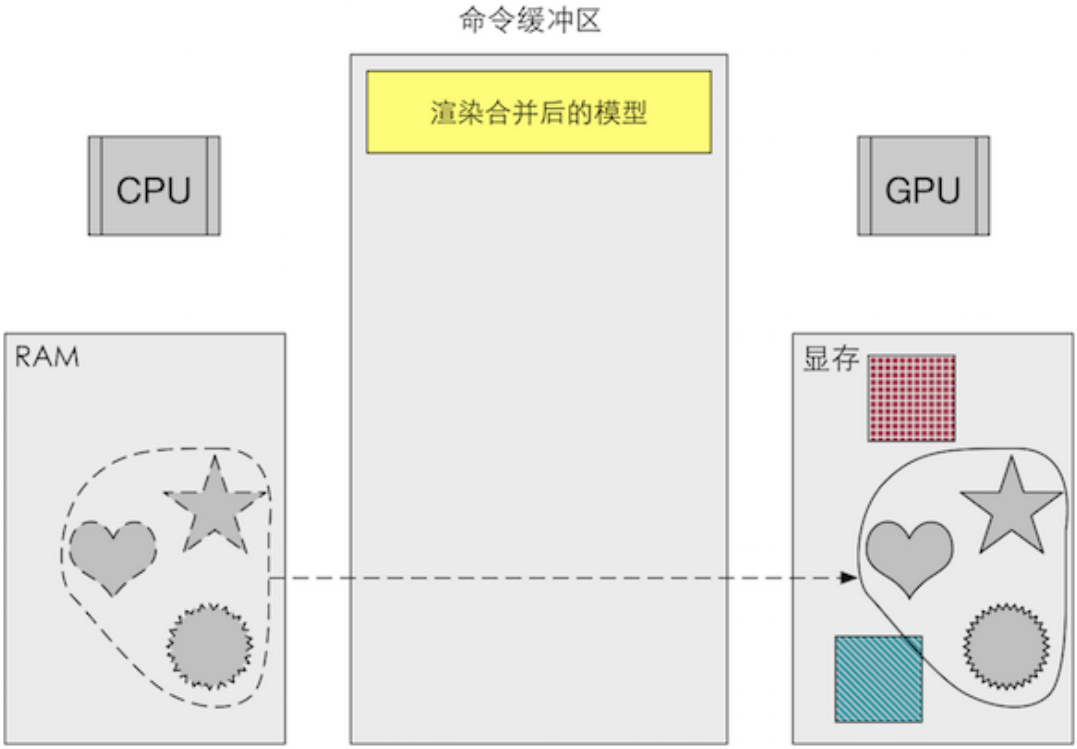


图234.　利用批处理

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。

七、　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。

哔哩哔哩次元游戏人生

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。

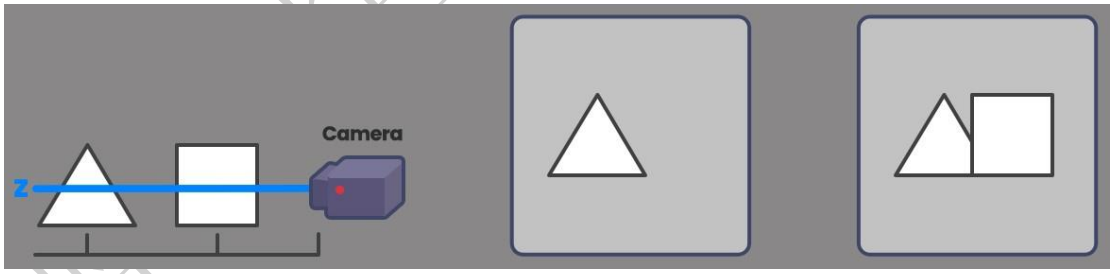


图235.　GUP 绘制演示

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。

哔哩哔哩虚元游戏人生

```
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
```

```
        }
    }
```

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。

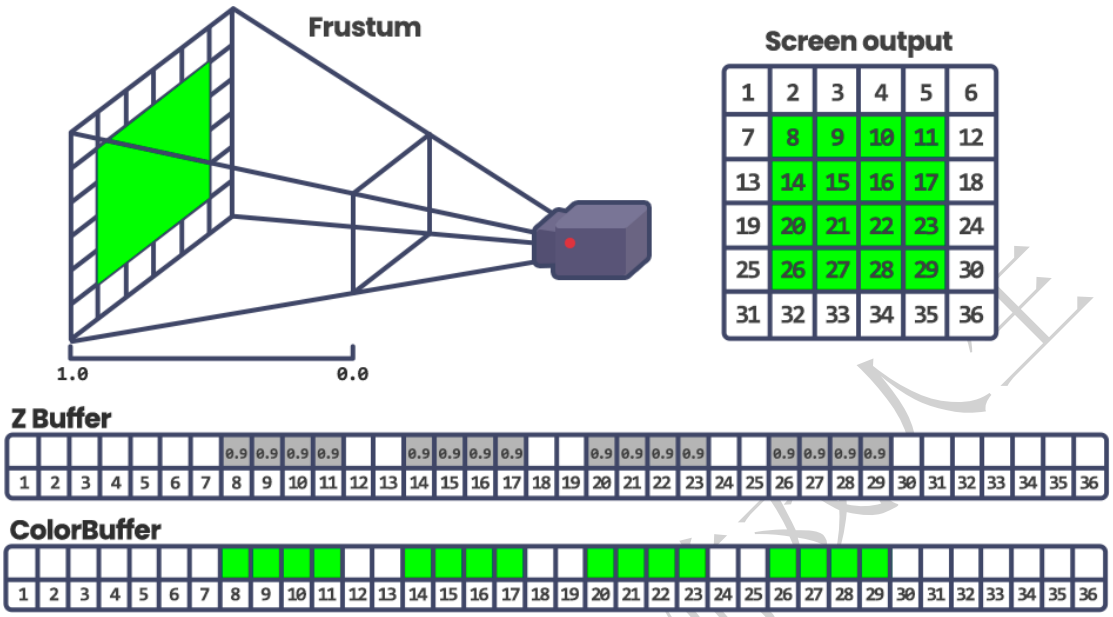


图236.　Z-Buffer 存储场景中物体的深度，Color Buffer 存储 RGBA 颜色信息

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。

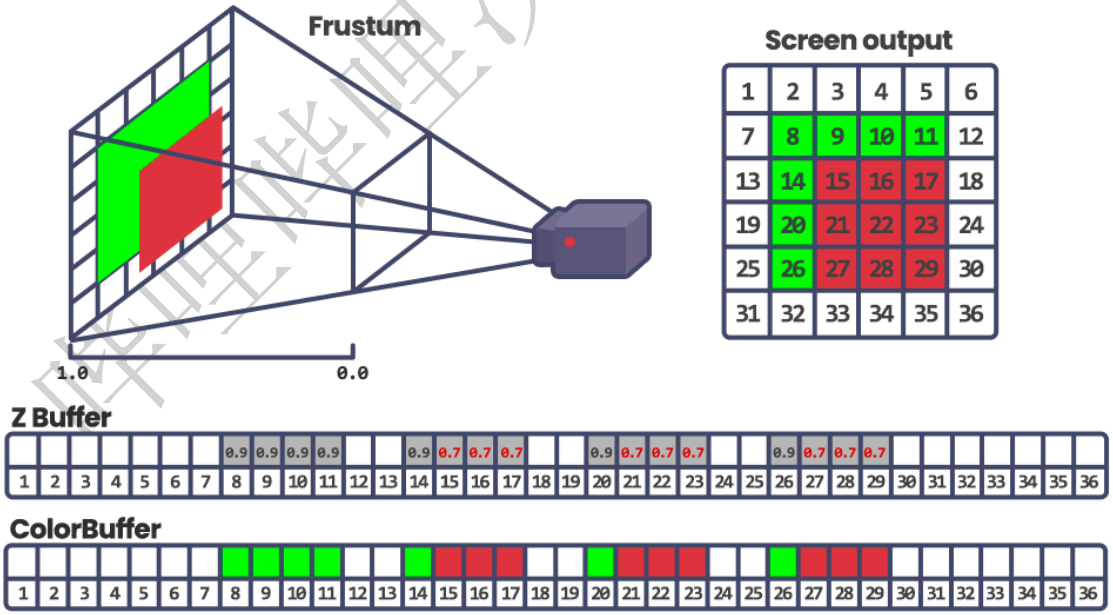


图237.　距离更近的图像

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。

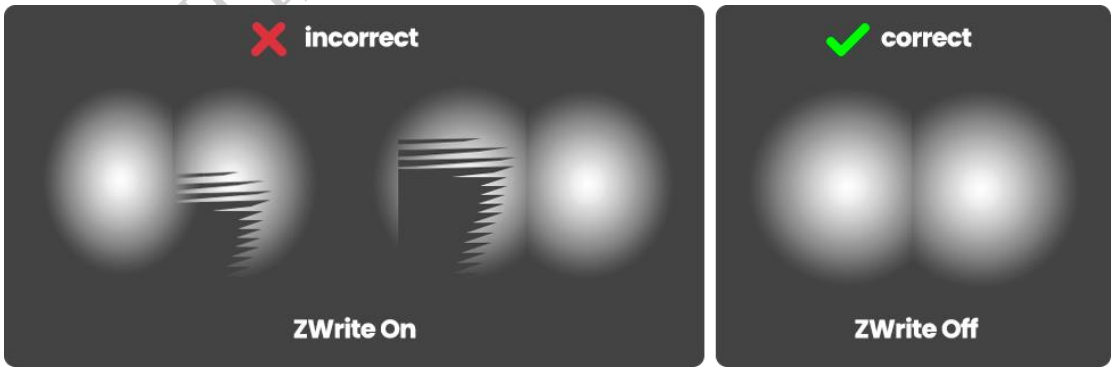


图238. z-fighting

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。

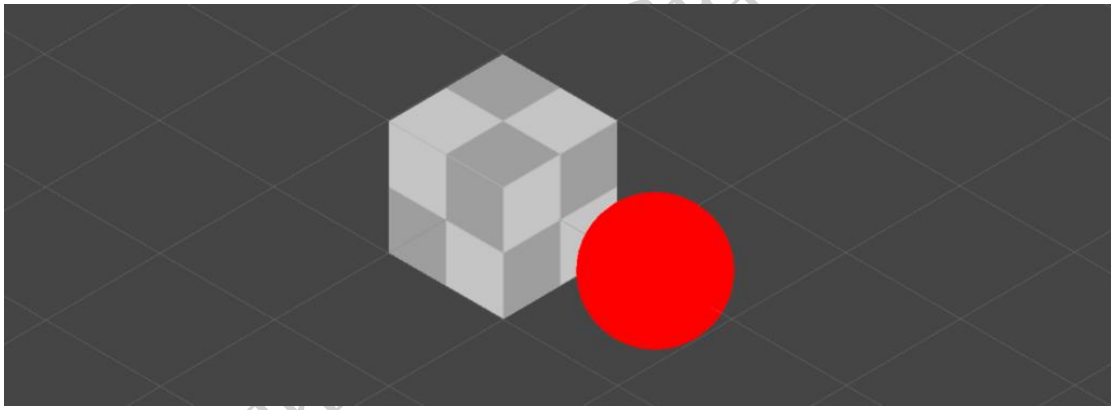


图239. Sphere Is Forward Of Square

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。

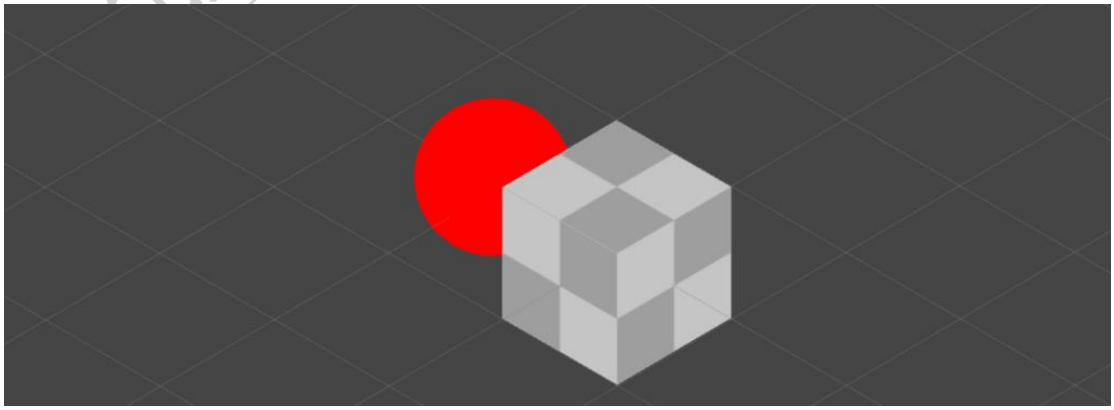


图240. Square Is Forward Of Sphere

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。

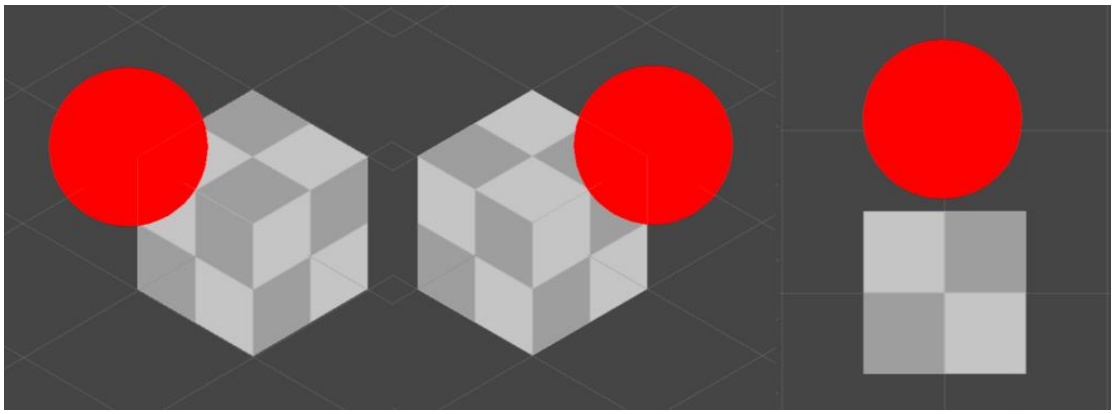


图241.　ZTest Always

```
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。

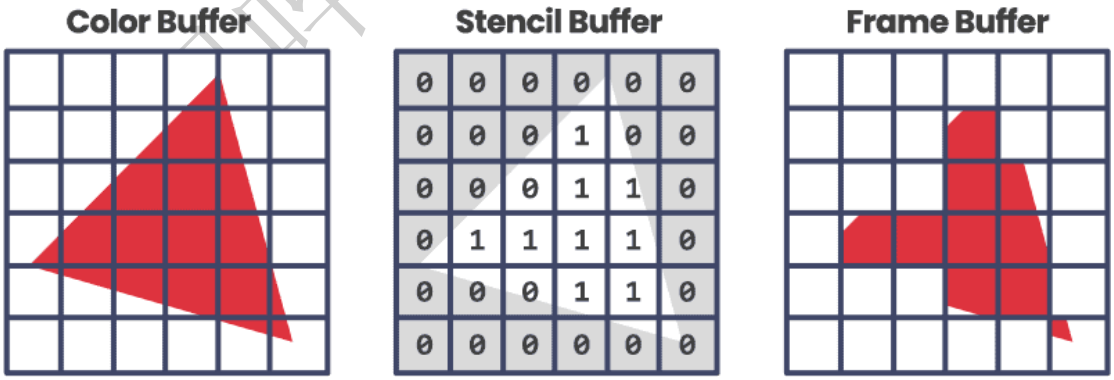


图242.　模板缓冲

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243. 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244.　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

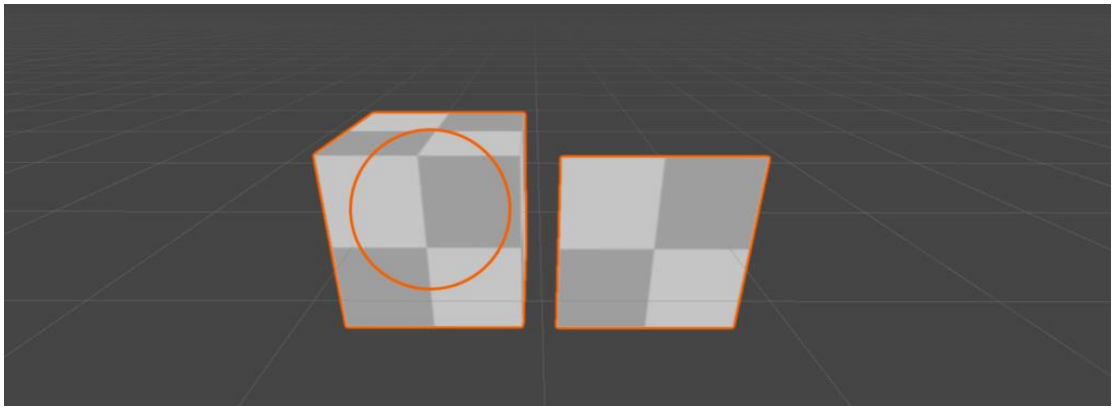


图245.　模板示例

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

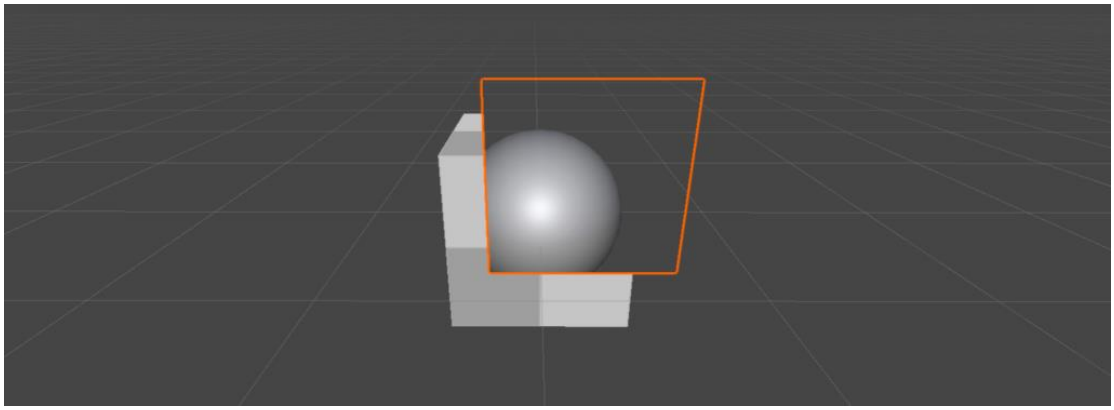


图246.　遮罩

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。

八、　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)
            {
```

哔哩哔哩次元游戏人生

```
                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

6. Unity 内置文件和变量

九、 有向距离函数-SDF-Sign Distance Function

1. 圆-Circle

符号距离函数（sign distance function），简称 SDF，又可以称为有向距离函数，在空间中的一个有限区域上确定一个点到区域边界的距离并同时对距离的符号进行定义：点在区域边界外部为正，内部为负，位于边界上时为 0。

简单来说，SDF 就是某一点到某一个区域边界的最短距离。

求某一点到圆的边的距离是最为简单的，这里分两种情况。大致来说都是一样的，就是用点与圆心的距离减去半径来进行判别。

哔哩哔哩次元游戏人生

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。

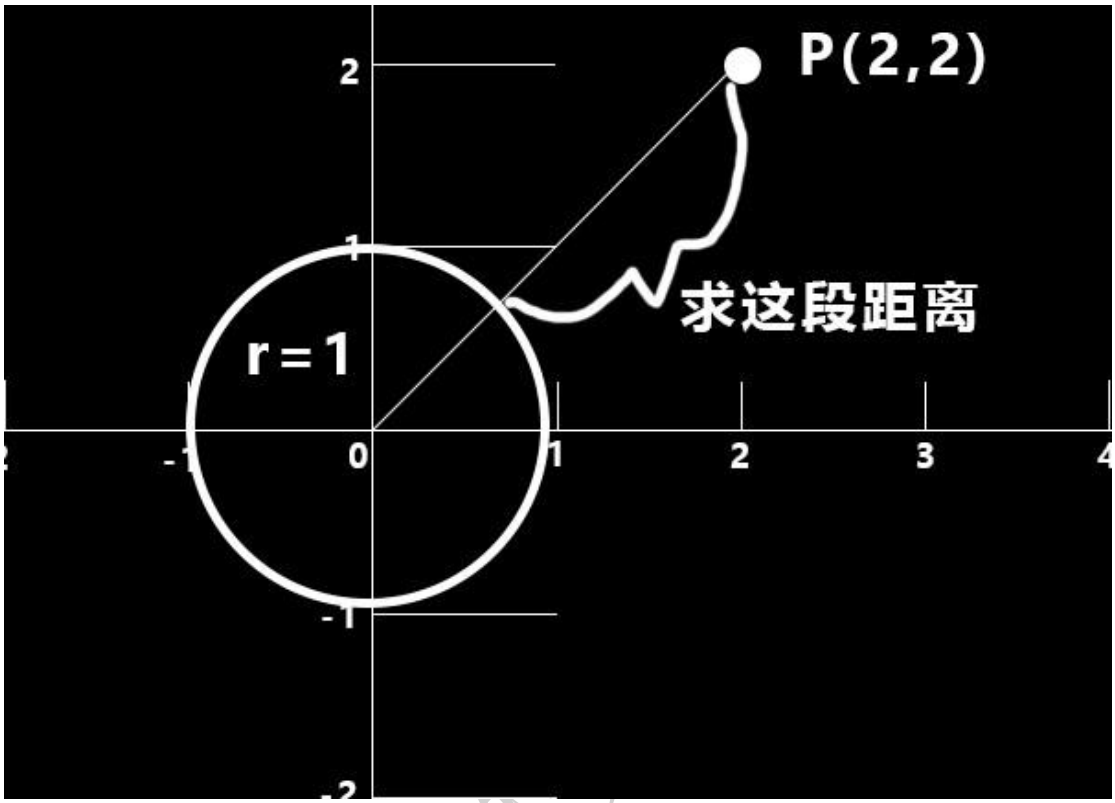


图247.　圆心在坐标轴原点

```
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。）

哔哩哔哩

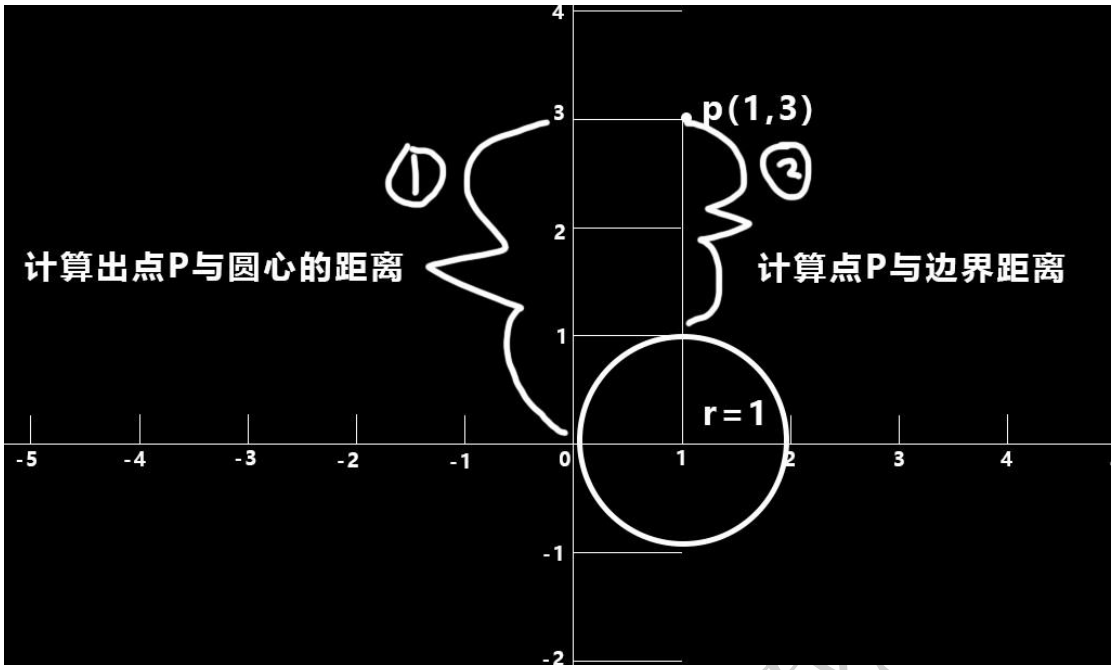


图248.　　圆心不在坐标轴原点

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。

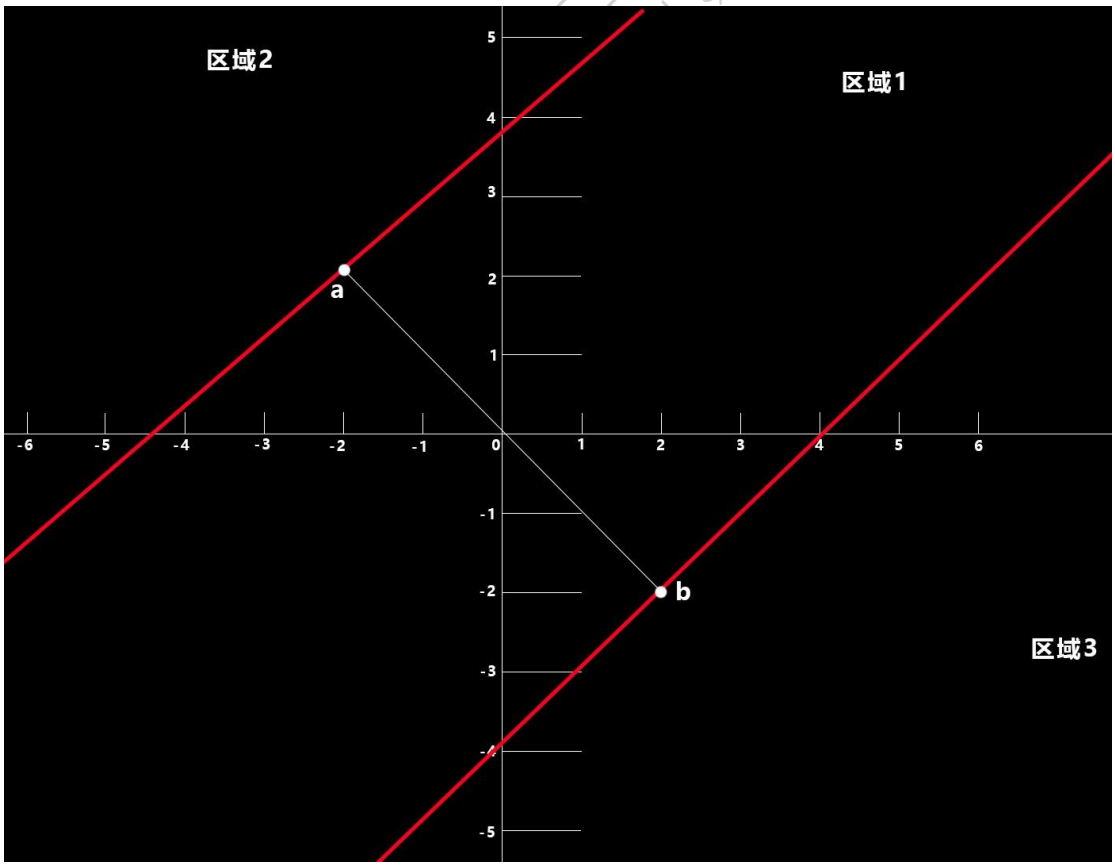


图249.　　SDF 线段区域划分

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

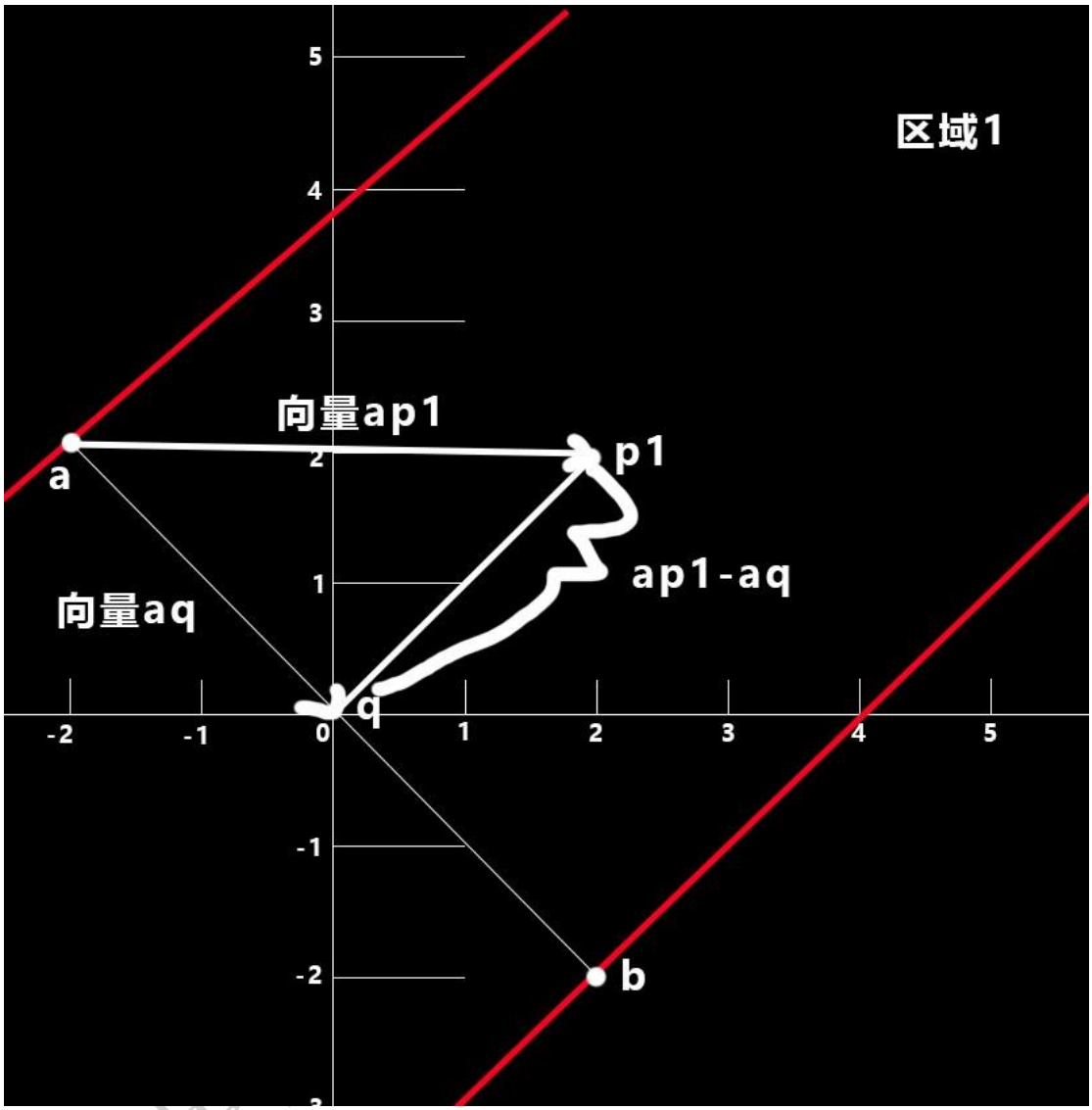


图250.　　点位于区域一内

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，$\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r^{1}\overrightarrow{ab}$。（$r^{2}$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。

[1] 这里的 r 是单词 Ratio 的缩写
[2] 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

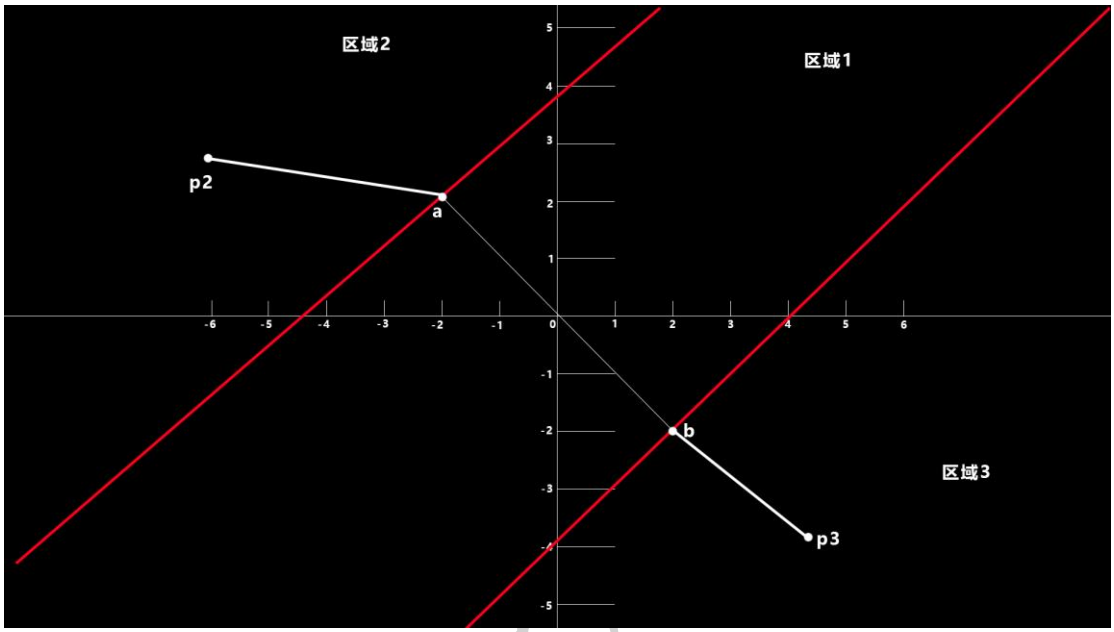


图251. 点位于区域二和三

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=$\|\overrightarrow{qp_1}=\overrightarrow{ap_1}-\overrightarrow{aq}=\overrightarrow{ap_1}-r$[3]$\overrightarrow{ab}\|$

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);
```

[3] 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。

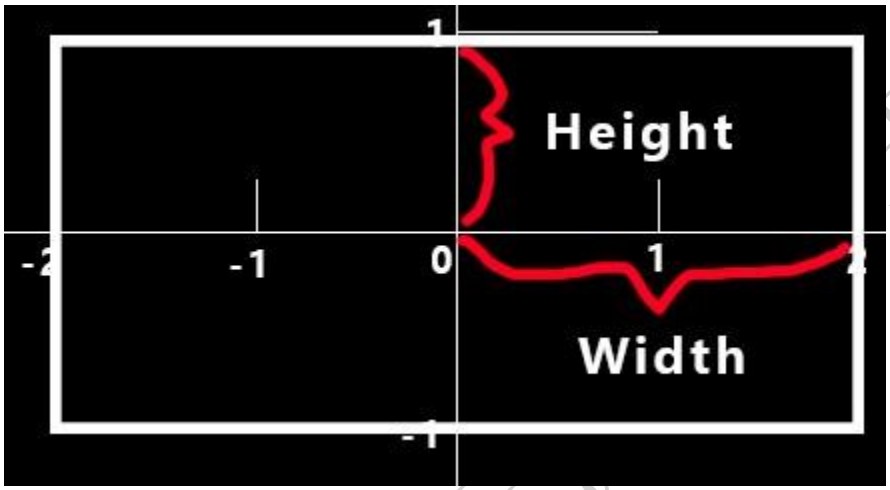


图252.　　矩形绘制原理

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。

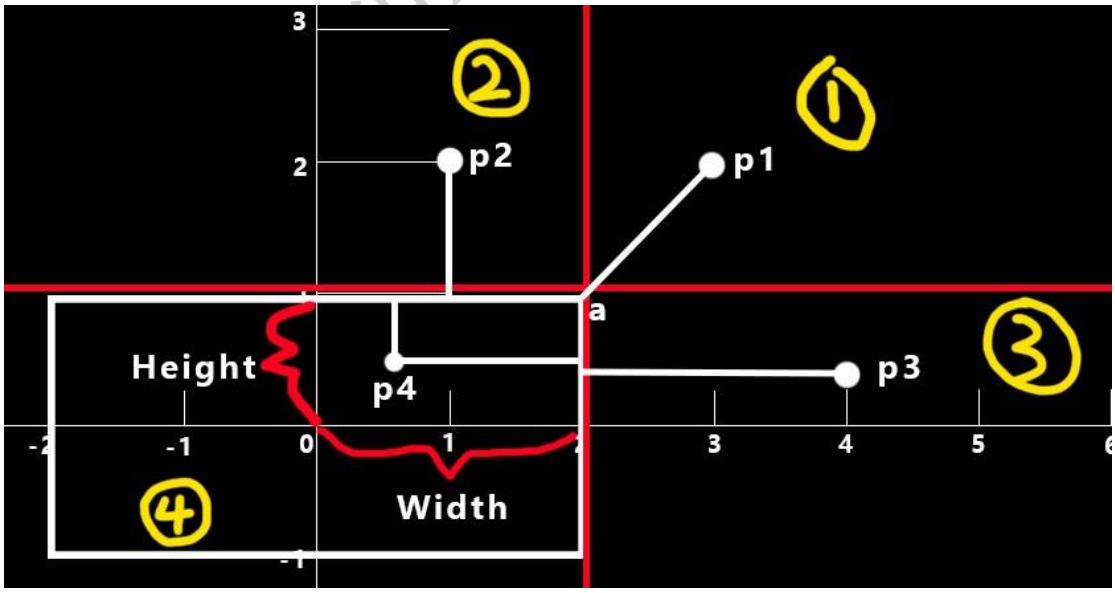


图253.　　目标点到矩形的情况划分

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

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