



VEngine 用户指南

0.1版

引言:	5
主要特性:	5
基本架构:	5
1.编辑器	5
2.播放器	5
项目窗口:	6
项目组成:	7
1.资源目录	7
2.Xcode工程相关的	7
编辑器:	8
1.基本布局	8
2.基本操作	10
场景节点系统:	11
1.场景的组织架构	11
2.什么是Transform?	11
3.什么是SceneNode?	12
4.什么是Agent?	12
资源系统:	13
1.为什么需要资源系统?	13
2.支持的资源格式	13
3.资源的动态更新	13
4.导入COLLADA文件	13
材质系统:	14
1.为什么需要材质脚本，而不是直接编写Shader?	14
2.如何工作?	14
3.材质实例脚本的格式	14
4.什么是参数源?	15

5.内置 (builtin) 参数源表	16
6.材质原型脚本内容的语法	16
7.为什么使用Shader节点?	17
8.常用材质节点表	17
9.材质实例的格式	19
10.材质实例脚本内容的语法	19
动画系统:	21
1.如何工作?	21
2.如何使用动画?	21
Agent系统:	22
1.Agent系统如何工作?	22
2.单个Agent如何工作?	22
3.编写最简单的Agent的代码	23
4.使用Agent	25
5.SceneNodeAgent	25
6.ActorAgent	25
7.GUIAgent	26
8.最简单的状态操作	26
GUI系统:	27
1.GUI层	27
2.GUI控件的行为	27
碰撞系统:	28
1.碰撞器的分类	28
2.全局碰撞器如何工作?	28
3.局部碰撞器如何工作?	28
许可证:	29
1.许可证如何工作?	29
2.如何创建许可证?	29

引言：

VEngine是一个3D游戏引擎，它的底层使用C++编写，使用Swift编写游戏逻辑，它拥有高效而且轻量级的渲染架构和碰撞检测系统以及简单易用的用户接口，任何掌握了Swift的人都能轻松上手，用最少的代码量实现游戏逻辑。VEngine目前只支持macOS和iOS，但是不久的将来它支持Android、Windows、PS4、XB1等不同平台，实现使用Swift语言的“一次编写，到处运行”。

主要特性：

- 1.完全并发（concurrent）的渲染架构，可以最大限度的发挥硬件的性能。
- 2.支持使用Swift编写游戏逻辑模块。
- 3.独创的材质脚本系统，通过材质脚本语言来编写跨平台材质，避免编写不跨平台的Shader代码。
- 4.独创的代理人（Agent）系统来组织和管理逻辑模块，让游戏逻辑的组织更加清晰。
- 5.强大的分层式动画管理器，可实现多个动画的加权融合，达到动画平滑过渡的效果。
- 6.自带高效的轻量级碰撞系统，可以减轻物理系统的负担，也便于性能的优化。
- 7.动态光照系统，支持平行光（direction light）、聚光灯（spot light）、点光源（point light）这三种灯光，可实现基于ESM的动态软阴影。
- 8.支持COLLADA文件格式，可直接导入COLLADA内的模型、材质、纹理到资源系统内。
- 9.强大的资源系统。
- 10.完善的编辑器支持。
- 11.强大的场景系统，可以在场景系统上衍生出更多的应用方式。
- 12.基于OpenAL的音效系统。
- 13.图形API的抽象层，为了支持多种图形API提供保证，目前支持OpenGL和Metal，将来会支持OpenGL ES、Vulkan、DirectX等。
- 14.多平台支持，核心代码使用C++编写使得整个引擎拥有天生的跨平台优势，目前支持macOS和iOS平台，将来会支持Android、Windows、PS4、XB1等平台。

基本架构：

1.编辑器

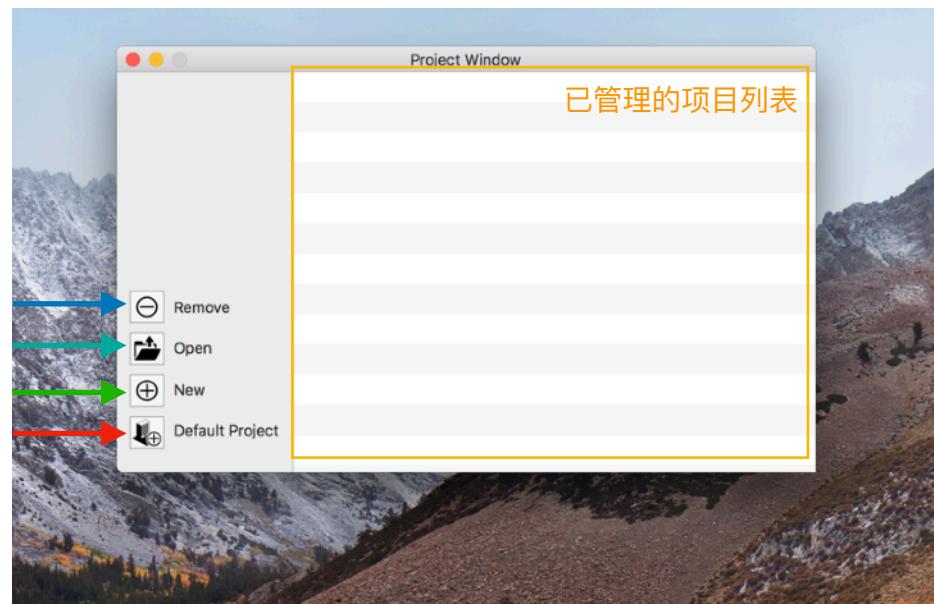
编辑器主要用于整个游戏项目和场景的编辑，通过编辑器用户可以创建新场景，再进入场景内添加、摆放资源，编辑GUI控件，给场景内的节点添加游戏逻辑等。

2.播放器

播放器用于游戏的运行，通常一个平台对应一个播放器。

项目窗口：

移除一个项目
打开一个已存在的项目
创建空项目
创建默认项目



如上图所示，VEngine刚开始运行时会弹出项目窗口，在项目窗口右边是已管理的项目列表，双击任何一个项目可打开编辑器窗口。在左边有创建默认项目和创建空项目按钮，可以创建项目并打开编辑器窗口。另外可以通过打开一个已有的项目按钮打开已有项目，并通过移除项目按钮移除项目。

项目组成：

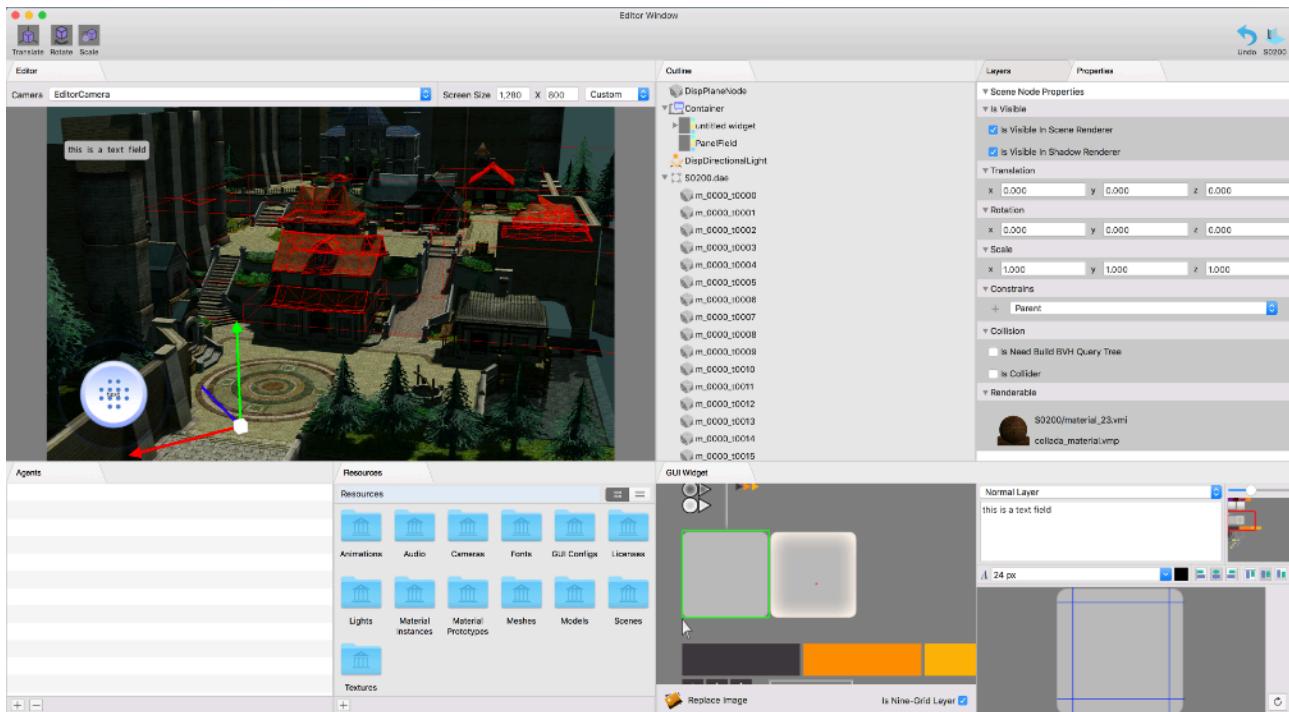
1. 资源目录

子路径名	描述
Audio	音效资源
Characters	包含动画的3D模型资源和动画资源
Fonts	字体资源
Licences	许可证资源，一个Player如果没有生成对应的许可证的话，运行时会显示“unregistered”标签
Materials	材质资源，包含材质实例和材质原型两部分
Scenes	场景资源
Models	不包含动画的静态模型资源
Textures	纹理资源
Scripts	Swift脚本资源，所有的游戏脚本必须放在这个目录里

2.Xcode工程相关的

子路径名	描述
Frameworks	macOS平台和iOS平台需要用到的框架
iOSLogic.xcodeproj	iOS平台的播放器使用的逻辑
iOSPlayer.xcodeproj	iOS平台的播放器的主体Xcode工程
macOSLogic.xcodeproj	macOS平台的播放器的逻辑工程
macOSPlayer.xcodeproj	macOS平台的播放器的主体工程
macOSPlayer_Debug.xcodeproj	macOS平台的播放器的主体工程的除错版，编译后不会拷贝资源。

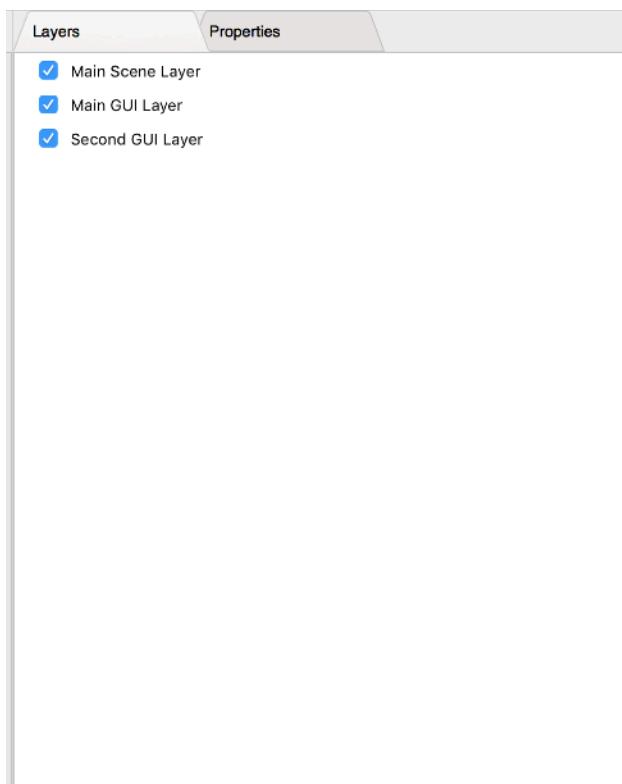
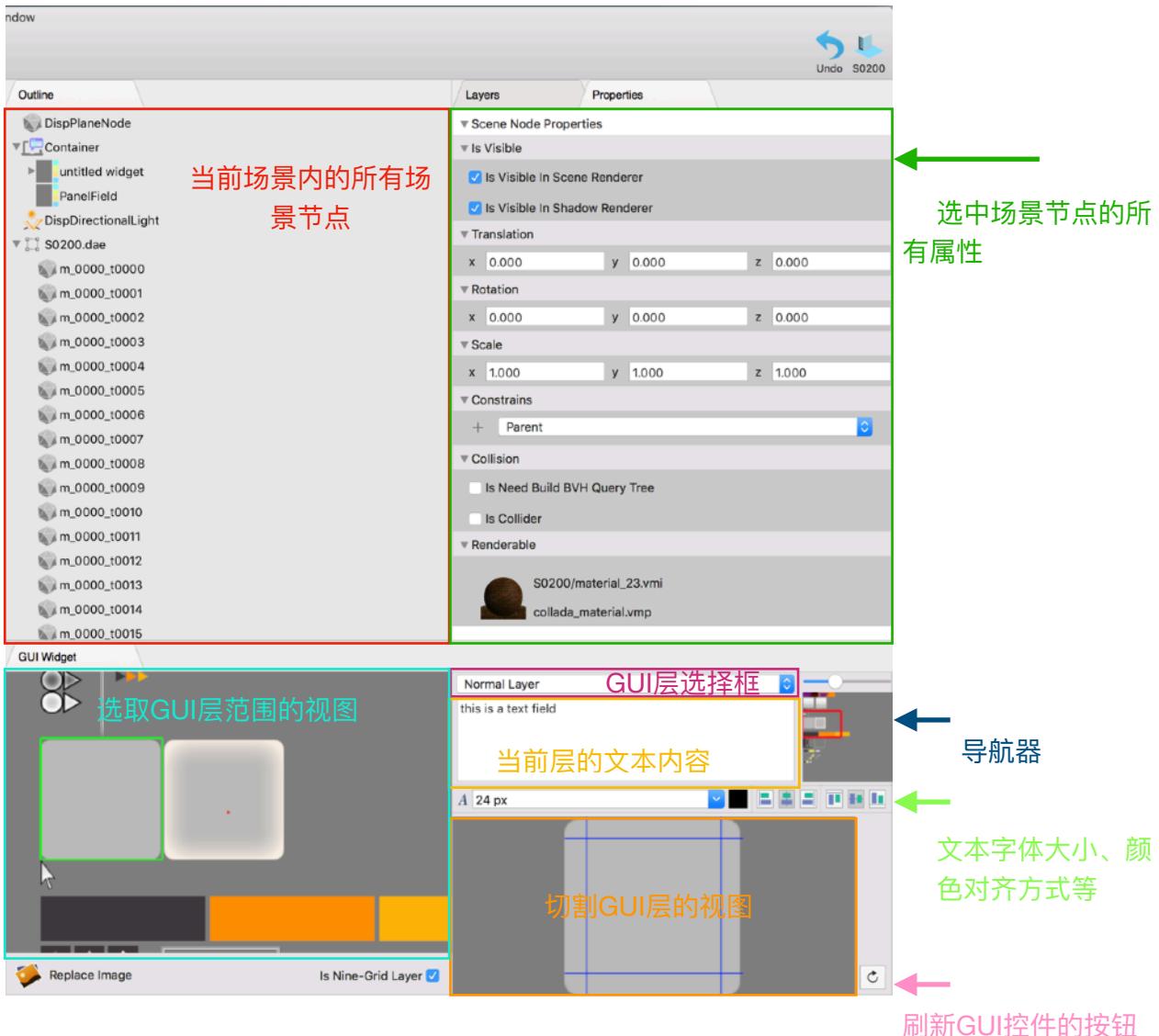
编辑器：



1. 基本布局

如上图所示，这是打开一个场景后编辑器显示的内容，下面分别介绍。





场景渲染层的使能视图

- 主场景层就是场景内除了GUI以外的物体
- 主GUI层是优先级较高的GUI层
- 第二GUI层是优先级较低的GUI层。

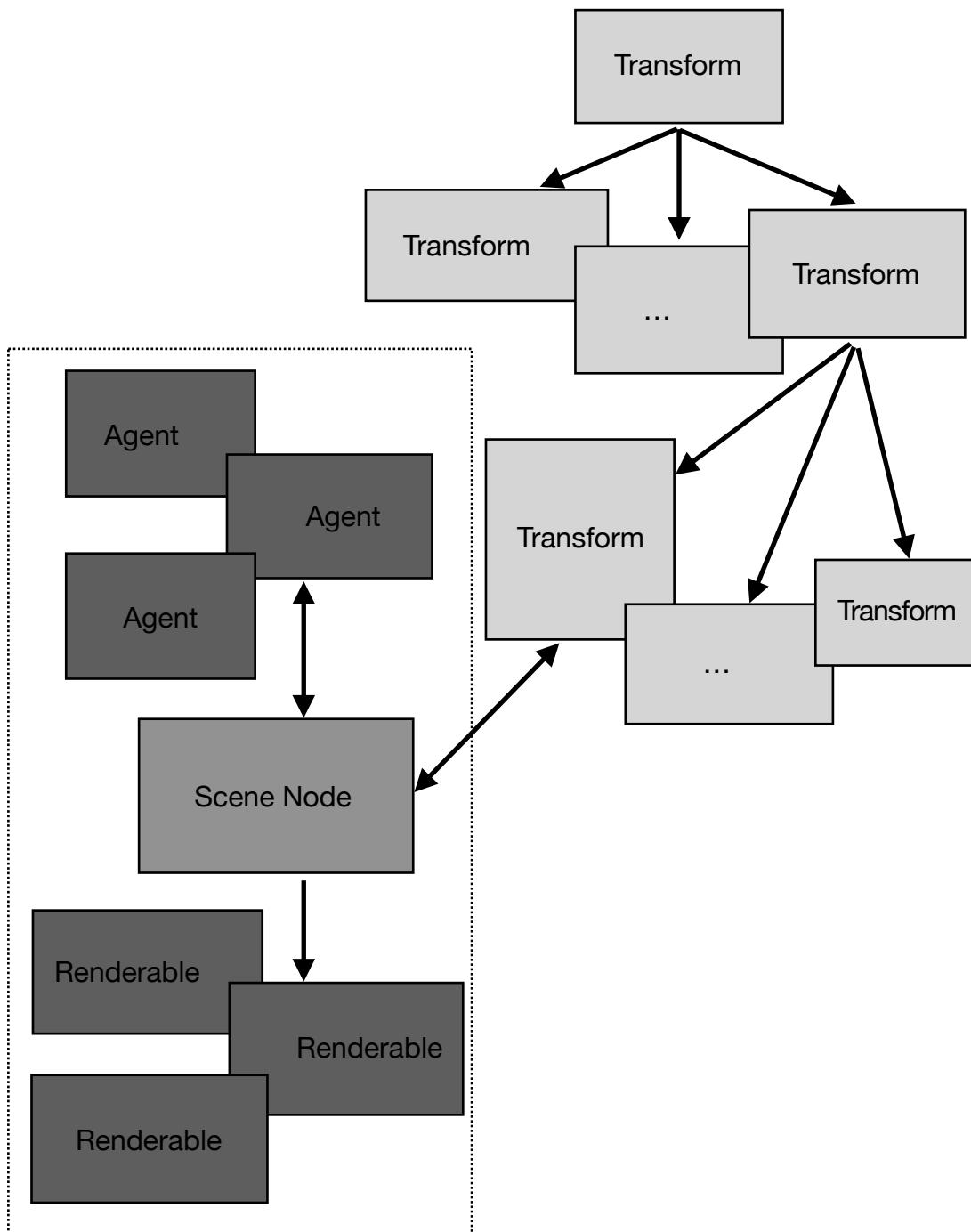
2. 基本操作

- a. 旋转摄像机: 按住Command 按下左键拖动鼠标
- b. 平移摄像机: 按住Option 按下左键拖动鼠标
- c. 推拉摄像机: 按住Command 按下右键拖动鼠标
- d. 摄像机聚焦到某物体: 选中一个3D物体, 按快捷键f
- e. 切换3D控制器为位移控制器: 选中一个3D物体, 按快捷键w
- f. 切换3D控制器为旋转控制器: 选中一个3D物体, 按快捷键e
- g. 切换3D控制器为缩放控制器: 选中一个3D物体, 按快捷键r



场景节点系统：

1. 场景的组织架构



如上图所示，场景内所有的物体都是用Transform组织起来的树形结构，而每一个Transform下面挂载一个SceneNode，一个SceneNode里又包含了一组Agent和一组Renderable。这就是目前场景组织的大致架构。

2. 什么是Transform？

Transform给出一个物体在所处空间内的位移、旋转和缩放。变换矩阵主要有两种。

a. 3D空间内的Transform。

b. 2D的GUI空间内的Transform。

Transform存在层级关系，父Transform发生改变时，父Transform下的所有子Transform会递归改变。

3.什么是SceneNode?

SceneNode是包含了Agent和Renderable的容器，SceneNode和Transform互相持有，SceneNode下所有的Renderable的坐标数据全都来自于SceneNode所持有的Transform，SceneManager每帧都会去更新场景内所有的Transform和SceneNode。

4.什么是Agent?

Agent是一个使用Swift编写的游戏逻辑控制对象，具体见Agent系统章节。

资源系统：

1.为什么需要资源系统？

原因有二，

a.需要实现资源的异步加载和已加载资源的缓存。

b.需要做到存储介质无关的资源读取。

第一条很好理解，第二条什么是介质无关的资源读取？游戏打包后资源可能存在于不同的地方，有可能在硬盘里、有可能在闪存在、有可能网络上有可能在压缩包内，资源存在的形式多种多样，但是使用方法是一样的，这就是介质无关的资源读取。

2.支持的资源格式

扩展名	描述
wav	音效资源格式
dae	COLLADA资源格式
ttf、ttc	字体资源格式
lic	许可证资源格式
vni、vmp	材质实例和材质原型
plist	场景文件和项目文件
png、dds	纹理资源格式

3.资源的动态更新

在打开场景编辑器时，对资源系统的任何修改都会在编辑器内检测到，尤其是在对资源的重命名时。对资源的任何重命名都会导致整个场景的自适应调整。

4.导入COLLADA文件

一个COLLADA内使用的纹理路径是绝对路径，当创建默认工程时会自动创建一个名为COLLADATextures的子目录，默认工程内的COLLADA文件所引用的纹理全在COLLADATextures子目录内，通过文件菜单的“导入COLLADA”子菜单项导入这些COLLADA文件到场景内时，如果该文件引用了纹理，则会自动导入对应的纹理到资源系统内。

材质系统：

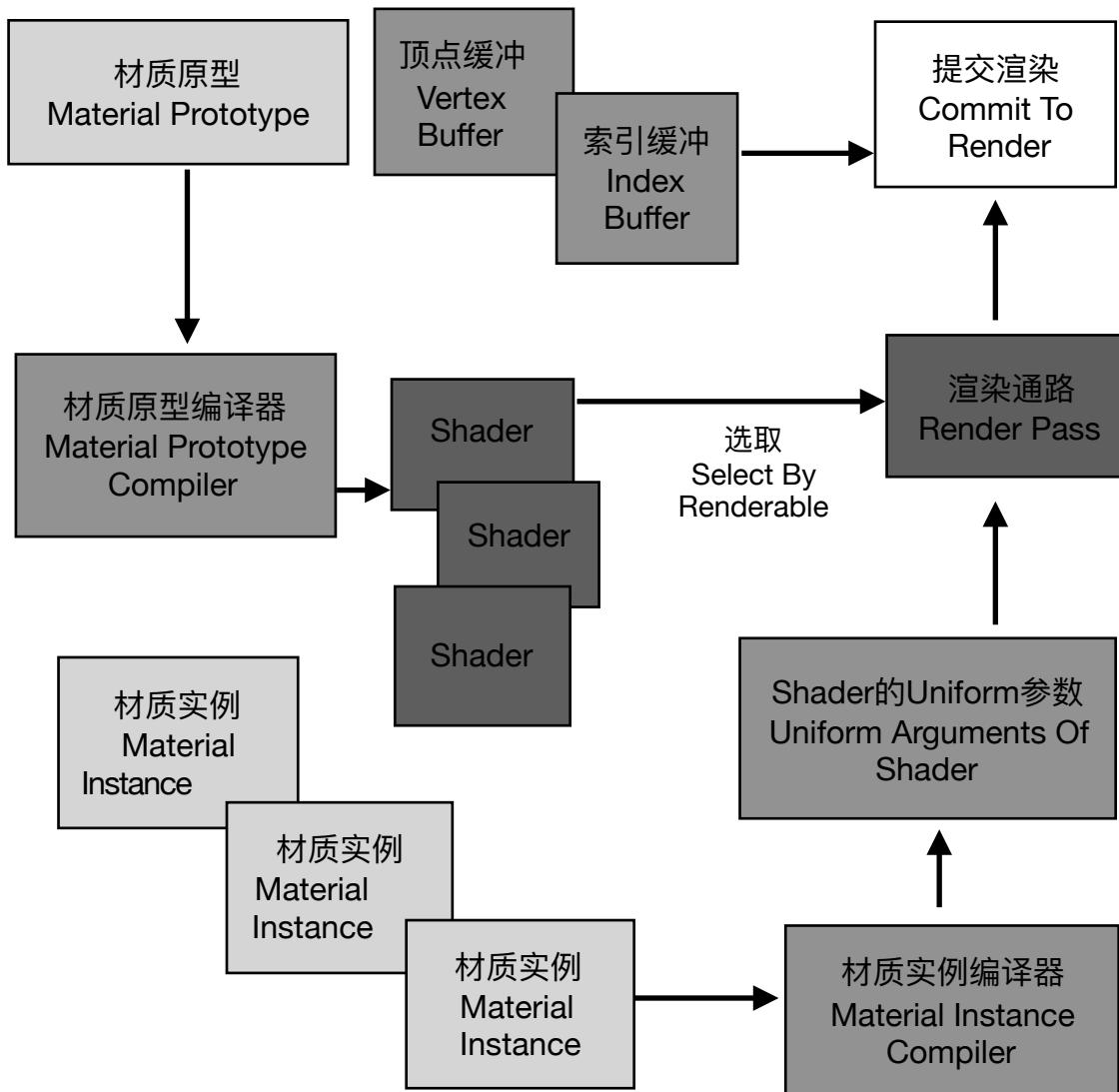
1.为什么需要材质脚本，而不是直接编写Shader?

原因有二，

a.Shader是图形API相关的，各种图形API都有自己的一套Shader，导致跨API移植需要重写Shader。

b.Shader是和渲染器相关的，不同的渲染器（前向渲染器、延迟着色器、前向+渲染器等等）使用的Shader不一样，导致Shader重复编写。

2.如何工作？



如上图所示，材质分为材质原型和材质实例两部分，材质原型通过材质原型编译器的编译变成一组Shader，然后根据不同的Renderable选择出适当的Shader组合成Render Pass。另一方面，材质实例经过材质实例编辑器的编译成为Shader使用的参数，参数设置到Render Pass里，合并了Vertex Buffer和Index Buffer最终提交给渲染器渲染。

3.材质实例脚本的格式

```
extern { 用户定义的导出的参数源 }
```

```
pass (使用的参数名 : 使用的参数源 )
```

[float4 输出颜色变量名 : 渲染目标]

{ 材质原型脚本的内容 }

```
1 extern
2 {
3     float4 Emission;
4     float4 Ambient;
5     float Shininess;
6     float Transparency;
7     float2 TexCoordOffset;
8 }
9 pass (Position      : VaryWorldPosition,
10        TexCoord       : VaryTexCoord0,
11        Normal         : VaryNormal,
12        Binormal       : VaryBinormal,
13        Tangent        : VaryTangent,
14        AmbientLighting : AmbientLightColor,
15        EmissionColor   : Emission,
16        AmbientColor    : Ambient,
17        ShininessValue  : Shininess,
18        TransparencyValue : Transparency,
19        TexCoordOffsetValue : TexCoordOffset,
20        DiffuseMap      : ColorMap0,
21        SpecularMap    : ColorMap1,
22        DetailNormalMap : NormalMap0)
23 [float4 outColor   : RenderTarget0]
24 }

25 float2 tmpTexCoord;
26 float4 DiffuseColor;
27 float4 SpecularColor;
28 float3 tmpLightingColor0;
29 float3 tmpLightingColor1;
30 float3 tmpLightingColor2;
31 float3 scaledDiffuseLighting;
32 float3 scaledSpecularLighting;
33 float3 scaledAmbientLighting;
34 float3 finalAmbientLighting;
35 float3 finalLightingColor;
36 float4 color0;
37
38 Add(TexCoord, TexCoordOffsetValue)[tmpTexCoord];
39
40 /// sample normal texture, obtain surface normal for each pixel.
41 NormalTextureSample(DetailNormalMap, tmpTexCoord)[SurfaceNormal];
42 SurfaceSpecularPower = ShininessValue;
43
44 TextureSample(DiffuseMap, tmpTexCoord)[DiffuseColor];
45 TextureSample(SpecularMap, tmpTexCoord)[SpecularColor];
46
47 /// calculate diffuse, specular and ambient lighting.
48 Mul(DiffuseLighting, DiffuseColor.rgb)[scaledDiffuseLighting];
49 Mul(SpecularLighting, SpecularColor.rgb)[scaledSpecularLighting];
50 Mul(AmbientLighting, AmbientColor.rgb)[scaledAmbientLighting];
51 Mul(scaledAmbientLighting, DiffuseColor.rgb)[finalAmbientLighting];
52
53 /// mix diffuse, specular, ambient and emission lighting.
54 Add(scaledDiffuseLighting, scaledSpecularLighting)[tmpLightingColor0];
55 Add(finalAmbientLighting, EmissionColor.rgb)[tmpLightingColor1];
56 Add(tmpLightingColor0, tmpLightingColor1)[tmpLightingColor2];
57 /// clamp lighting to 0 to 1
58 Clamp(tmpLightingColor2, 0.0, 1.0)[finalLightingColor];
59
60 /// finally, do the alpha test.
61 color0 = {finalLightingColor, DiffuseColor.a};
62 AlphaTest(color0)[outColor];
63 }
```

用户定义的导出的参数源

使用的参数名和使用的参数源，使用“:”号隔开

输出的颜色变量名和渲染目标，使用“:”号隔开

材质原型脚本的内容

4. 什么是参数源？

参数源是一个id，用来标示出参数从何而来，参数源包含两个部分，

a. 内置的参数源。

b. 用户定义的参数源。

内置的参数源来自于渲染器，用户的参数源来自于材质实例。

5. 内置 (builtin) 参数源表

参数源名称	类型	描述
VaryWorldPosition	float4	逐像素的世界坐标
VaryProjPosition	float4	经过投影矩阵后的逐像素坐标
VaryTexCoord0	float2	第一套纹理坐标
VaryTexCoord1	float2	第二套纹理坐标
VaryTexCoord2	float2	第三套纹理坐标
VaryTexCoord3	float2	第四套纹理坐标
VaryNormal	float3	逐像素的法线值
VaryTangent	float3	逐像素的切线值
VaryBinormal	float3	逐像素的副法线值
VaryColor	float4	逐像素的色彩值
CameraPosition	float3	摄像机世界坐标
CameraDirection	float3	摄像机世界方向
ColorMap0	texture2D	第一张色彩纹理，如果使用了alpha test的话，该纹理会用做渲染阴影图的alpha test之用
ColorMap1	texture2D	第二张色彩纹理
ColorMap2	texture2D	第三张色彩纹理
ColorMap3	texture2D	第四张色彩纹理
NormalMap0	texture2D	法线纹理
EnvironmentMap0	textureCube	第一张立方体纹理
EnvironmentMap1	textureCube	第二张立方体纹理

6. 材质原型脚本内容的语法

语法	变量声明
格式	类型 变量名; 类型 变量名 = 变量值;
例子	float2 tmpTexCoord; float2 tmpTexCoord = {0.0, 0.0};
描述	声明变量并赋值（可选）

语法	变量赋值
格式	变量名 = 变量值;
例子	tmpTexCoord = {0.0, 0.0};

描述	给变量赋值
语法	添加节点
格式	节点名(输入参数)[输出参数];
例子	Add(TexCoord, TexCoordOffsetValue)[tmpTexCoord]; Sub(1.0, TransparencyValue)[invertedTransparencyValue];
描述	添加一个Shader节点

7.为什么使用Shader节点?

Shader节点这个概念的存在是为了方便配合编辑器使用的，理想的情况下不需要编写任何一行代码，通过编辑器的编辑就可以生成材质。为了实现这个目标，于是便将一些常用的Shader逻辑封装到Shader节点里，供编辑器使用。(现在的VEngine并未实现材质编辑器，会在将来的某个时候添加上)

8.常用材质节点表

Shader节点名	输入参数	输出参数	描述
TextureSample	(texture2D 纹理, float2 纹理坐标)	[float3 rgb纹素值] [float4 rgba纹素值]	通过一个2D的纹理坐标采样 2D纹理输出一个纹素值
CubeTextureSample	(textureCube 纹理, float3 纹理坐标)	[float3 rgb纹素值] [float4 rgba纹素值]	通过一个3D的纹理坐标采样 一个立方体纹理输出一个纹素值
NormalTextureSample	(texture2D 法线纹理, float2 纹理坐标)	[float3 xyz法线值]	通过一个2D的纹理坐标采样 2D法线纹理输出一个法线值
Add	(float 数值a, float 数值b) (float2 数值a, float2 数值b) (float3 数值a, float3 数值b) (float4 数值a, float4 数值b)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对a和b两个值相加后输出
Sub	(float 数值a, float 数值b) (float2 数值a, float2 数值b) (float3 数值a, float3 数值b) (float4 数值a, float4 数值b)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对a和b两个值相减后输出

Shader节点名	输入参数	输出参数	描述
Mul	(float 数值a, float 数值b) (float2 数值a, float2 数值b) (float3 数值a, float3 数值b) (float4 数值a, float4 数值b)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对a和b两个值相乘后输出
Scale	(float 输入值, float 缩放值) (float2 输入值, float 缩放值) (float3 输入值, float 缩放值) (float4 输入值, float 缩放值)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对输入值乘以缩放值后输出
Clamp	(float 输入值, float 最小值, float 最大值) (float2 输入值, float 最小值, float 最大值) (float3 输入值, float 最小值, float 最大值) (float4 输入值, float 最小值, float 最大值)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	将输入值限制在给定的最大值和最小值之间
Pow	(float 输入值, float 幂) (float2 输入值, float 幂) (float3 输入值, float 幂) (float4 输入值, float 幂)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对输入值进行幂运算后输出
Dot	(float2 数值a, float2 数值b) (float3 数值a, float3 数值b) (float4 数值a, float4 数值b)	[float2 输出值] [float3 输出值] [float4 输出值]	对a和b两个矢量值点乘后输出
Cross	(float2 数值a, float2 数值b) (float3 数值a, float3 数值b) (float4 数值a, float4 数值b)	[float2 输出值] [float3 输出值] [float4 输出值]	对a和b两个矢量值叉乘后输出
Normalize	(float2 输入值) (float3 输入值) (float4 输入值)	[float2 输出值] [float3 输出值] [float4 输出值]	对输入矢量归一化后输出

Shader节点名	输入参数	输出参数	描述
Mix	(float 数值a, float 数值b, float 混合权重) (float2 数值a, float2 数值 b, float 混合权重) (float3 数值a, float3 数值 b, float 混合权重) (float4 数值a, float4 数值 b, float 混合权重)	[float 输出值] [float2 输出值] [float3 输出值] [float4 输出值]	对输入值a和输入值b通过混 合权重混合后输出
GaussianBlurSample	(texture2D 纹理, float2 纹 理坐标, float2 纹理解析 度, float 模糊半径)	[float4 rgba纹素]	对一张纹理进行给定半径的 高斯模糊采样, 输出一个纹 素 (模糊半径为1~5, 性能 较差)
HoriGaussianBlurSample2	(texture2D 纹理, float2 纹 理坐标, float2 纹理解析 度)	[float4 rgba纹素]	对一张纹理进行半径2像素 的横向高斯模糊采样, 输 出一个纹素 (性能较好)
VertGaussianBlurSample2	(texture2D 纹理, float2 纹 理坐标, float2 纹理解析 度)	[float4 rgba纹素]	对一张纹理进行半径2像素 的纵向高斯模糊采样, 输 出一个纹素 (性能较好)
AlphaTest	(float4 输入色彩)	[float4 输出色彩]	通过输入色彩的alpha值进 行alpha测试, 通常用于材质脚 本的末尾

9. 材质实例的格式

```
instance (" 材质原型名 ", 附加属性(可选))
{ 材质实例脚本的内容 }
```

材质原型名

```

1 instance ("phong_EcAcDt0ScNt0_AB.vmp", Transparent:2) 附加属性 (可选)
2 {
3     float4 EmissionColor = {0.000000, 0.000000, 0.000000, 1.000000};
4     float4 AmbientColor = {0.200000, 0.200000, 0.200000, 1.000000};
5     texture2D DiffuseMap (Filter:Bilinear, WrapU:Repeat, WrapV:Repeat) = "S1000/10water26.dds";
6     float4 SpecularColor = {0.200000, 0.200000, 0.200000, 1.000000};
7     texture2D DetailNormalMap = "default_normal_map.png";
8     float ShininessValue = 20.000000;
9     float TransparencyValue = 0.500000;
10    float2 TexCoordOffsetValue = {0.000000, 0.000000};
```

材质实例脚本的内容

10. 材质实例脚本内容的语法

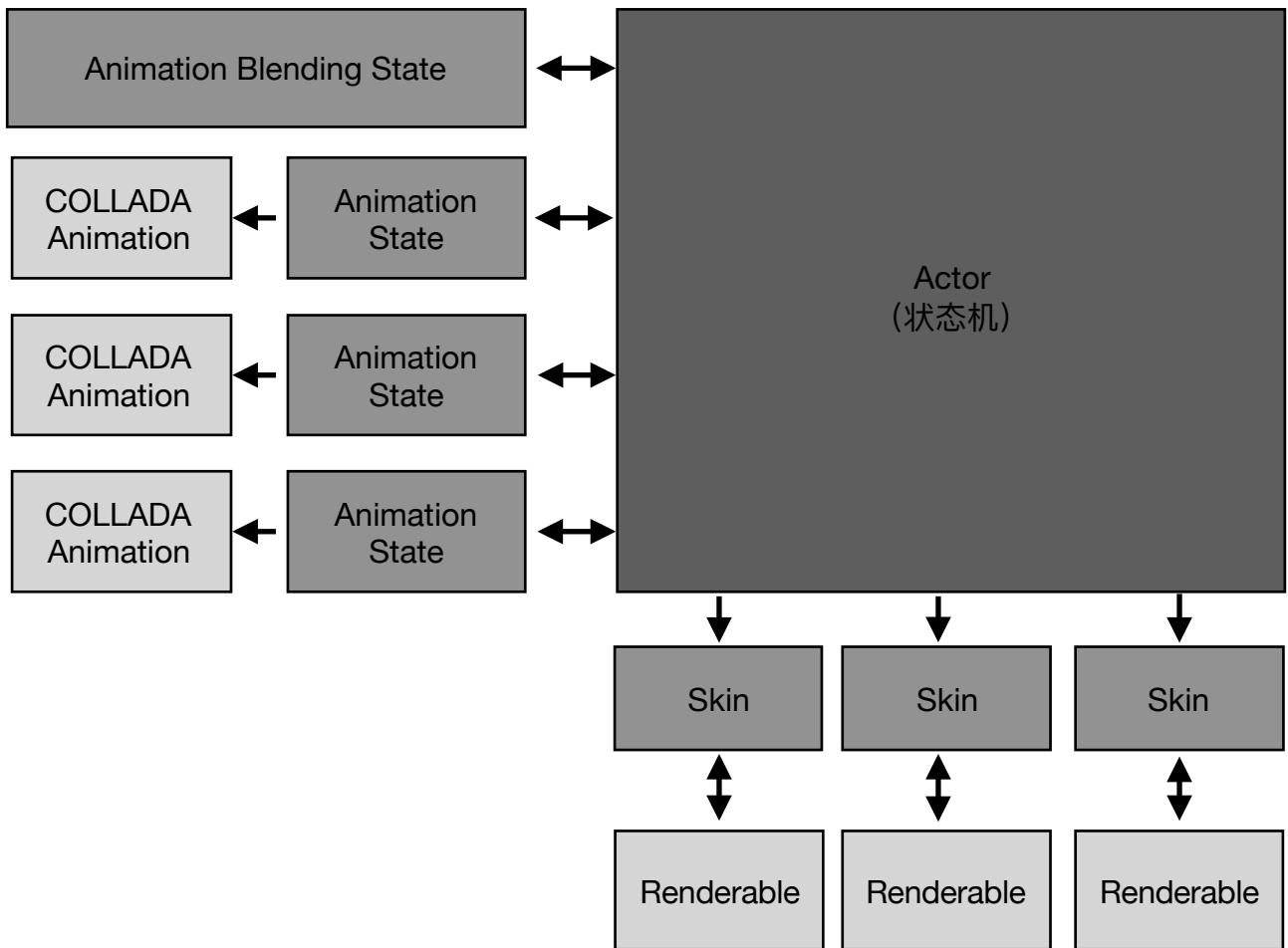
语法	参数赋值
格式	参数类型 参数名 = 参数数值;

例子	float4 EmissionColor = {0.0, 0.0, 0.0, 1.0}; float TransparencyValue = 0.5;
描述	给一个基本类型的参数赋值

语法	纹理赋值
格式	纹理类型 纹理名 = “纹理名”; 纹理类型 纹理名 (纹理属性) = “纹理名”;
例子	texture2D DiffuseMap (Filter:Bilinear, WrapU:Repeat, WrapV:Repeat) = “S1000/10water26.dds”; texture2D DetailNormalMap = “default_normal_map.png”
描述	给一个纹理类型的参数赋值

动画系统：

1. 如何工作？



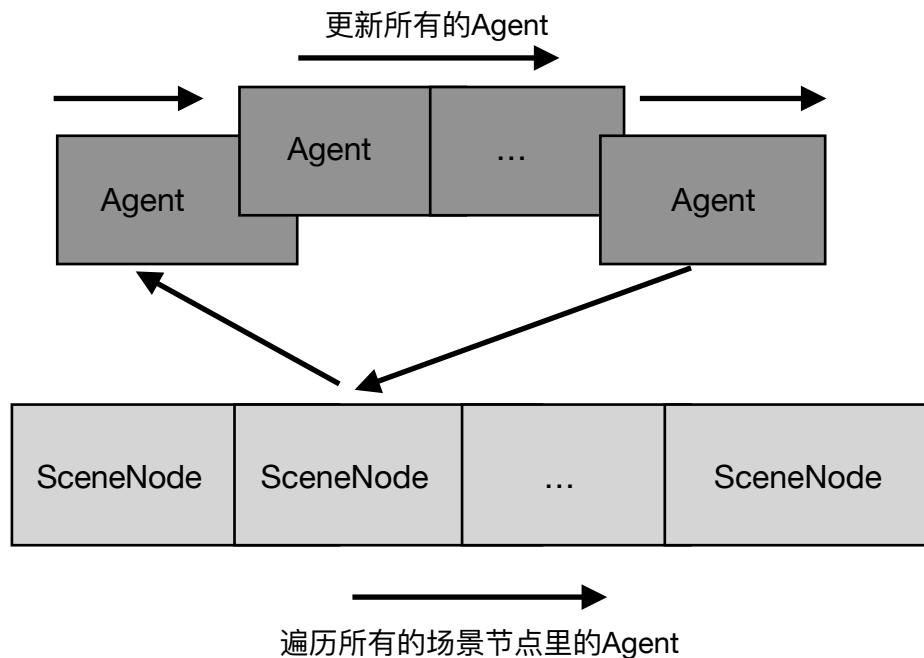
如上图所示，动画系统的核心是Actor，Actor本质是一个状态机。一个Actor内包含若干个Animation State，每个Animation State对应一个COLLADA动画。同时有一个Animation Blending State用来在多个Animation间实现多个动画的混合。Actor总是运行在单个动画状态或者混合动画状态。Actor计算出整个动画中每一帧骨架的所有骨骼矩阵，将这些矩阵传递给Actor挂载下的Skin。最终交给Renderable以供渲染。

2. 如何使用动画？

- a. 在导出COLLADA资源时将有蒙皮的3D模型和动画分离，动画作为单独的文件分别导出。
- b. 将有蒙皮的3D模型和动画资源全部放入到Characters目录内（可以建立子目录，方便管理）。
- c. 打开编辑器，会自动扫描所有的COLLADA资源，并且在游戏逻辑项目里自动生成动画的引用文件
- d. 用Xcode打开游戏工程，编写对应的ActorAgent（详情请见代理人章节）。
- e. 编译游戏工程，会自动导出所有的Agent信息，此时打开编辑器，可在代理人视图内给场景节点添加动画了。

Agent系统：

1. Agent系统如何工作？



如图所示，有一个单独线程会和渲染线程同步的速率遍历整个场景内所有的场景节点，并更新每个场景节点下的Agent。

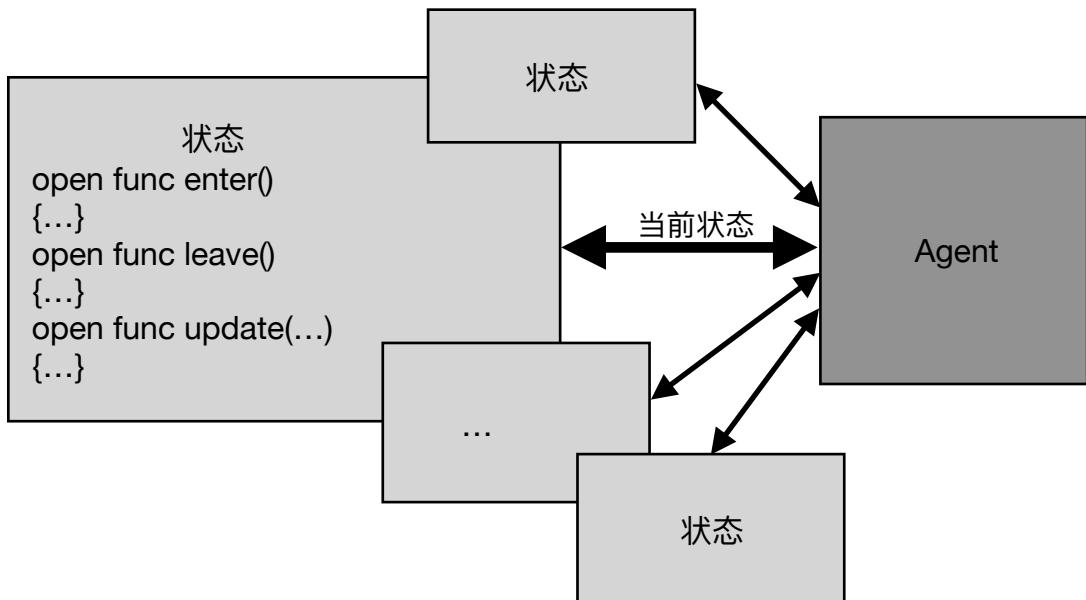
Agent分为三种，a.SceneNodeAgent。b.ActorAgent。c.GUIAgent。

SceneNodeAgent可以控制一个3D场景节点的位移、旋转、缩放的动画和场景节点下的ActorAgent的动画切换，以及可渲染物的材质属性等。

ActorAgent主要用来管理一个Actor的动画播放。

GUIAgent主要用来处理GUI的逻辑。

2. 单个Agent如何工作？

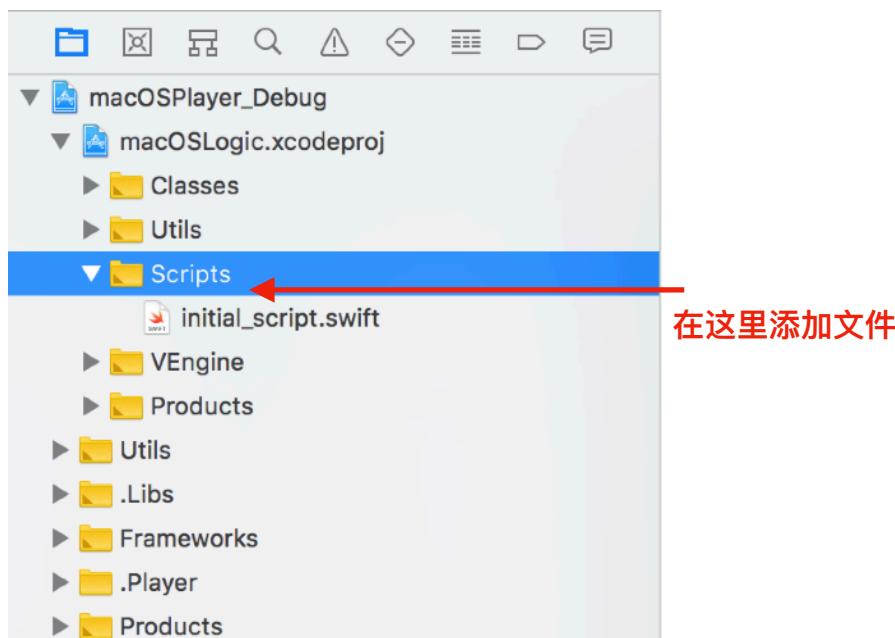


如上图所示，Agent是一个状态机，每个Agent内包含若干个状态，每个状态至少包含enter、leave、update三个方法，同时有且只有一个当前状态。

- a.当一个Agent更新时会调用它自己的当前状态的update方法。
- b.当一个状态转换到另一个状态时，会调用当前状态的leave和目标状态的enter方法，转换完成后，目标状态变成新的当前状态。

3.编写最简单的Agent的代码

- a.首先打开游戏工程下的macOSPlayer_Debug.xcodeproj
- b.在如下位置添加swift文件，添加的文件确保要放在项目下的Scripts子目录内。



c.在新添加的swift文件内编写如下内容。

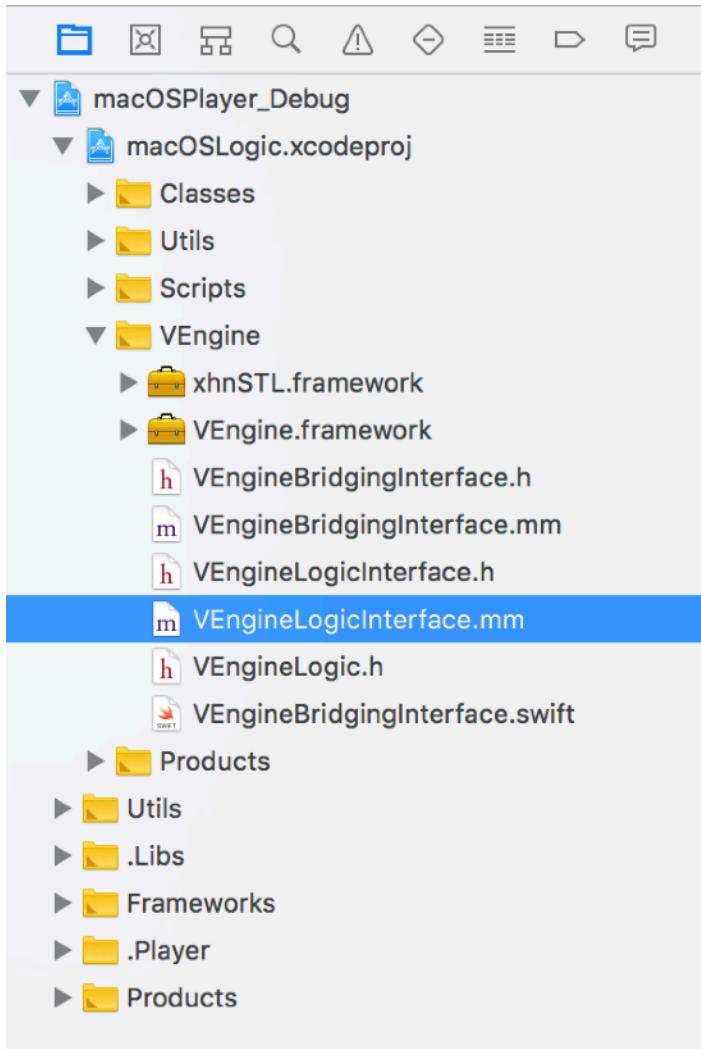


The screenshot shows the Xcode interface with the following file path in the top bar: macOSPlayer_Debug > macOSLogic.xcodeproj > Scripts > new_script.swift > No Selection. The code editor displays the following Swift code:

```
1 import Foundation
2
3 open class MySceneAgent : SceneNodeAgent
4 {
5 }
6
7 open class MyActorAgent : ActorAgent
8 {
9 }
10
11 open class MyGUIAgent : GUIAgent
12 {
13 }
```

d.编译整个工程。

e.找到VEngineLogicInterface.mm



看到文件里有如下内容，表示已经成功添加了三种空的Agent。

```
s_createSceneNodeAgentProcDic[@"MySceneAgent"] = [[CreateSceneNodeAgentProc alloc] initWithProc:^(void* sceneNode)
s_createActorAgentProcDic[@"MyActorAgent"] = [[CreateActorAgentProc alloc] initWithProc:^(void* renderSys, void* actor)
s_createGUIAgentProcDic[@"MyGUIAgent"] = [[CreateGUIAgentProc alloc] initWithProc:^(void* renderSys, void* widget)
```

4. 使用Agent

a. 选中需要添加Agent的场景节点。

b. 按照如下流程操作。



5. SceneNodeAgent

a. 默认的SceneNodeAgent的基类已经包含了EmptyState、RepeatState、MoveToState、RotateToState、ScaleToState和TransitToState六个状态。

b. 如果SceneNodeAgent类以及派生类内部包含访问权限是open，同时继承自SceneNodeState和SceneNodeStateInterface的子类，则会将该子类作为状态添加到该Agent对象内。

c. 如果SceneNodeAgent派生类包含一个以上的状态时，则会将这些状态里的第一个状态作为该派生类初始化后的初始状态，如果该派生类没有包含状态，则会用EmptyState作为该场景代理人的默认状态。

6. ActorAgent

a. 每一个ActorAgent的状态对应一个COLLADA动画文件，当ActorAgent实例化时会自动加载它所有状态所对应的COLLADA动画文件到Actor对象里。

b. 和SceneNodeAgent一样，存在默认状态，也就是默认动作，当ActorAgent启动时会自动循环执行默认动作。

7.GUIAgent

a.一个GUIAgent只有五种状态，普通状态、悬停状态、拖拽状态、选中状态和按下状态。普通状态就是一个GUI控件的初始状态，悬停状态是鼠标悬停在GUI控件上的状态，（对于没有鼠标的移动平台，则会在状态切换到拖拽状态前很短的时间内进入该状态。），拖拽状态是拖动一个GUI控件时，该控件处于的状态，选择状态是GUI控件被选择后的状态，按下状态是GUI控件被作为按钮按下后的状态。

b.GUIAgent需要配合GUI控件的行为来使用，有关GUI控件行为，请见GUI控件章节。

8.最简单的状态操作

```
424     open class DraggingState : GUIState, GUIStateInterface
425     {
426         open func update(_ elapsedTime : Double) -> Void {}
427         open func enter(_ prevState : GUIState?) -> Void {
428             if let owner = mOwner as? JoystickAgent {
429                 owner.setJoystickStatus()
430             }
431             let agentSys = VAgentSystem.get()
432             guard let agent = agentSys!.findSceneNodeAgent() as MageMainAgent? else { return }
433             if let currentState = agent.getCurrentState() {
434                 currentState => MageMainAgent.RunState.self
435             }
436         }
437         open func leave() -> Void {
438             let agentSys = VAgentSystem.get()
439             guard let agent = agentSys!.findSceneNodeAgent() as MageMainAgent? else { return }
440             if let currentState = agent.getCurrentState() {
441                 currentState => MageMainAgent.IdleState.self
442             }
443         }
444     }
```

如上图所示，通过类型是MageMainAgent的SceneNodeAgent的派生类的getCurrentState方法获取到当前的状态，然后通过操作符“=>”转换到“MageMainAgent.RunState”状态。

```
181     open func enter(_ prevState : SceneNodeState?) -> Void {
182         if let agent = self[MageActorAgent.self] {
183             agent => MageActorAgent.Mage_run.self
184         }
185         cameraSceneNode = nil
186         let cameras = VSceneNode.findSceneNodes(byDisplayName: "DispCamera")
187         for camera in cameras {
188             if let tran = camera.transform {
189                 if let cam = tran as? VCamera {
190                     cameraSceneNode = cam.owner!
191                 }
192             }
193         }
194     }
```

而ActorAgent的状态转换则如上图所示，在一个SceneNodeAgent内部可通过如下方法：

self[ActorAgent派生类名.self]

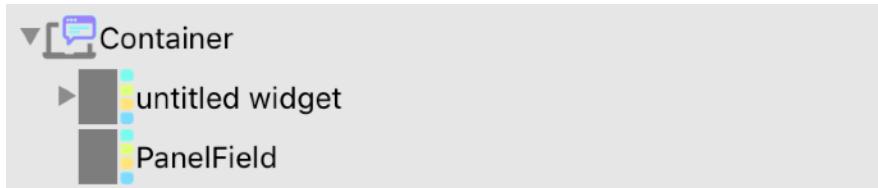
来取得当前SceneNode下的ActorAgent，然后状态转换的方法和SceneNodeAgent类似。

GUIAgent不能够手动控制状态转换，GUIAgent的状态转换完全取决于GUI的行为，相关内容请参阅GUI控件章节。

GUI系统:

1.GUI层

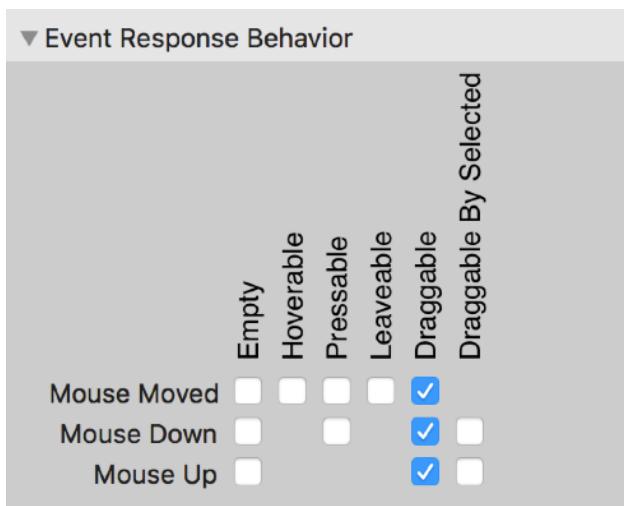
GUI分为主GUI层和第二GUI层两个部分，主GUI层的更新同步于主渲染线程，而第二GUI层的更新是异步的。



如上图所示，再任何场景内都包含一个名为“Container”的场景节点，任何在该节点下的GUI控件都会被加入到第二GUI层。

除此之外的3D节点下面都能挂载GUI节点，任何挂载在3D节点下的GUI节点都会加入到主GUI层。

2.GUI控件的行为



控件的事件分为三种鼠标移动、鼠标按下、鼠标弹起。

行为分为五种行为空行为、可悬停、可按压、可离开、可拖拽和被选择后可拖拽。

如上图所示，通过组合控件在三种事件上的行为，来实现控件的整体事件响应行为。

碰撞系统：

1. 碰撞器的分类

碰撞器分为全局碰撞器和局部碰撞器两类。

2. 全局碰撞器如何工作？

a. 遍历所有被标记为碰撞器的场景节点下所有的Renderable。

b. 将这些Renderable的AABB添加到全局的碰撞BVH树内。

c. 通过全局BVH树找出彼此间发生碰撞的场景节点。

d. 将碰撞信息传递给发生碰撞的场景节点。

3. 局部碰撞器如何工作？

a. 加载场景时发现一个场景节点需要创建BVH问询树，则会构建该场景节点的精确到三角面的BVH树。

b. 在脚本调用阶段可通过射线、球体、胶囊中的一种形状和建立了BVH问询树的场景节点做碰撞检测。

c. 检测结果可用于对游戏中角色的控制。

许可证:

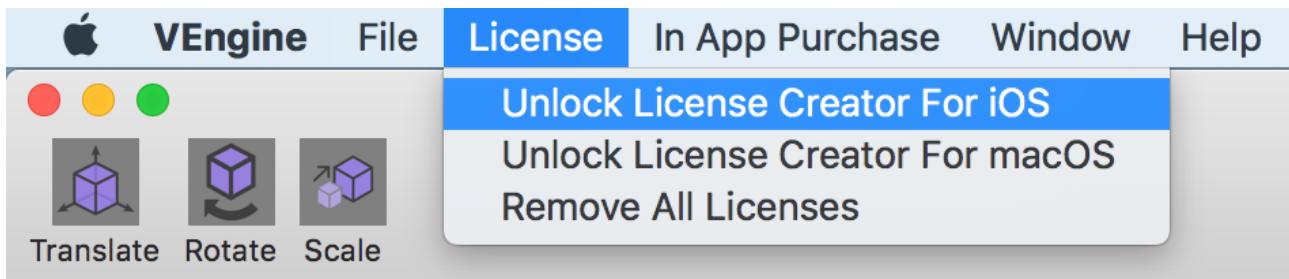
1. 许可证如何工作?

a. 许可证在同一个平台下是和bundle identifier绑定的，同一个平台的一个许可证对应一个bundle identifier。

b. 播放器运行时通过自身的bundle identifier去查询是否存在合适的许可证，如果没有，则会在屏幕左上角显示“unregistered”的标签。

2. 如何创建许可证?

如果要创建许可证，则必须解锁创建许可证的功能。



在完成了IAP的支付后便可以使用创建许可证的功能了。

