VEngine
User Guide

Ver0.1

Introduction：

VEngine is a 3D game engine, The underlying code is built by C++ and game logic is written using Swift, It has an efficient and lightweight rendering architecture and collision detection system as well as an easy-to-use programming interface, Anyone who has mastered Swift can easily get started and implement game logic with the least amount of code. VEngine currently only supports macOS and iOS, but it will support Android, Windows, PS4, XB1 and other platforms in the near future, Implementing "write once, run anywhere" using Swift language.

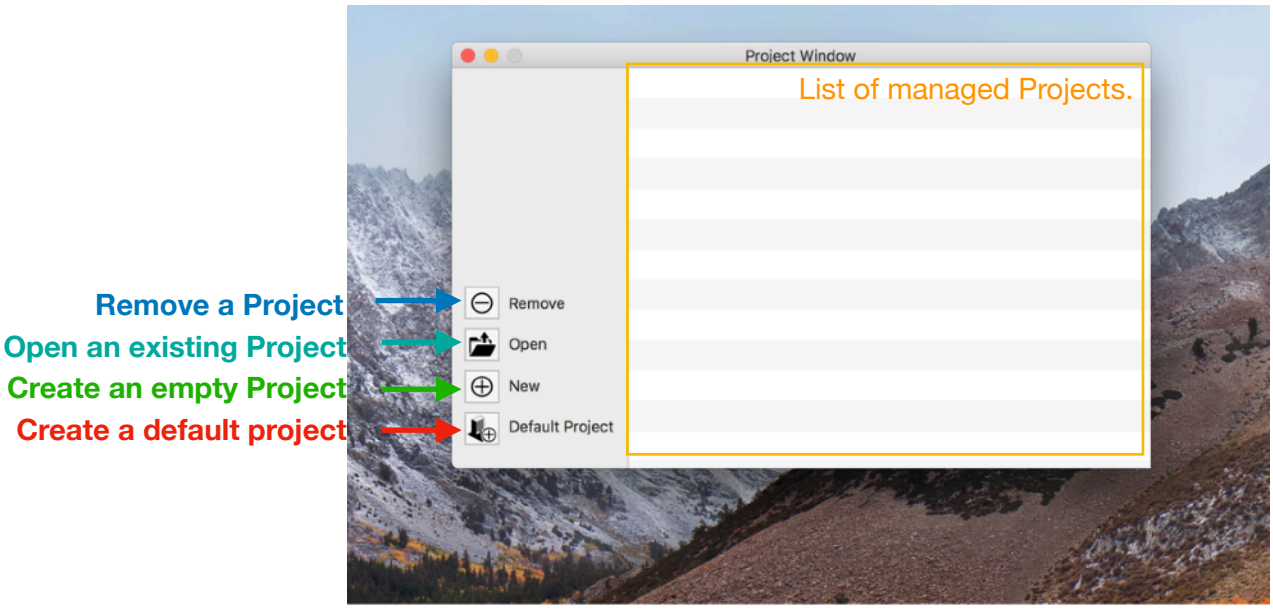
Main Features：

1.Fully concurrent architecture that maximizes hardware performance.
2.Support for writing game logic modules using Swift.
3.The creative Material Scripting System writes cross-platform material through the material scripting language, avoiding writing platform related shader code.
4.The creative Agent System organizes and manages logic modules to make the organization of game logic clearer.
5.Powerful layered Animation Manager that enables weighted blending of multiple animations to achieve smooth transitions.
6.Built-in an efficient lightweight Collision System that eases the burden on physical systems and facilitates performance optimization.
7.The dynamic Lighting System supports three types of lights: parallel light, spotlight, and point light, enabling dynamic soft shadows based on ESM.
8.Support COLLADA file format, can directly import COLLADA model, material, texture into the resource system.
9.Powerful Resource System.
10.Easy-to-use Editor support.
11.Powerful Scene System.
12.Sound System based on OpenAL.
13.The abstraction layer of the graphics API, in order to support multiple graphics APIs, currently supports OpenGL and Metal, and will support OpenGL ES, Vulkan, and DirectX in the future.
14.Multi-platform support, core code written in C++ makes the entire game engine have a natural cross-platform advantage, currently supports macOS and iOS platforms, and will support Android, Windows, PS4, XB1 and other platforms in the future.

Basic architecture:

1.Editor
The Editor is mainly used for editing the entire game project and scene. Through the editor, the user can create a new scene, then add and place resources in the scene, edit GUI widgets, and add game logic to the scene nodes in the scene.
2.Player
The Player is used to run the game, usually one platform corresponds to one player.

Project Window：



As shown in the above figure, when VEngine starts running, the Project Window will pop up. On the right side of the Project Window is a list of managed projects. Double-click any project to open the Editor Window. On the left there is a Default Project and New button to create a default project or empty project and open the editor window. You can also open an existing project by Open button and remove the project by Remove button.

Project Composition:

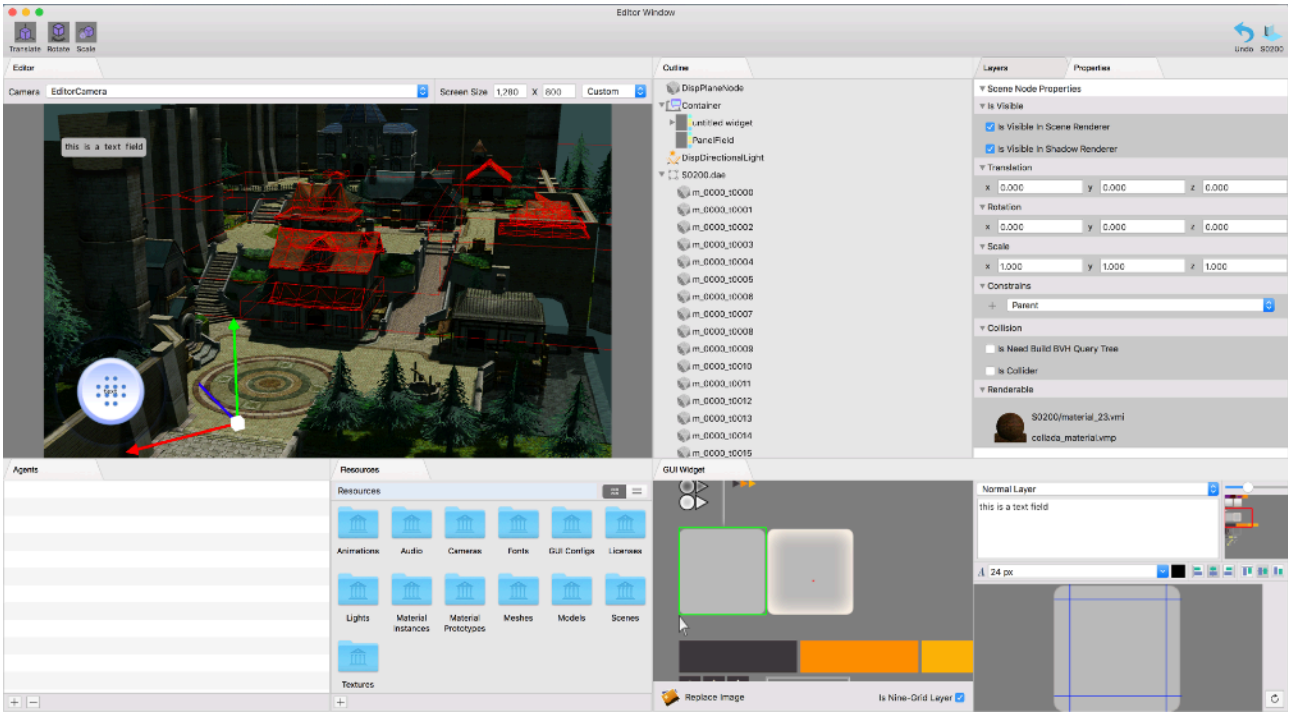
1. Resource Directory

Sub path name	Description
Audio	Sound resources
Characters	3D model resources and animation resources including animation
Fonts	Font resources
Licences	License resource, a Player will display "unregistered" tag if no corresponding license is generated
Materials	Material resources, including material instances and material prototypes
Scenes	Scene resources
Models	Static model resources without animation
Textures	Texture resources
Scripts	Swift script resources, all game scripts must be placed in this directory

2. Xcode project-related

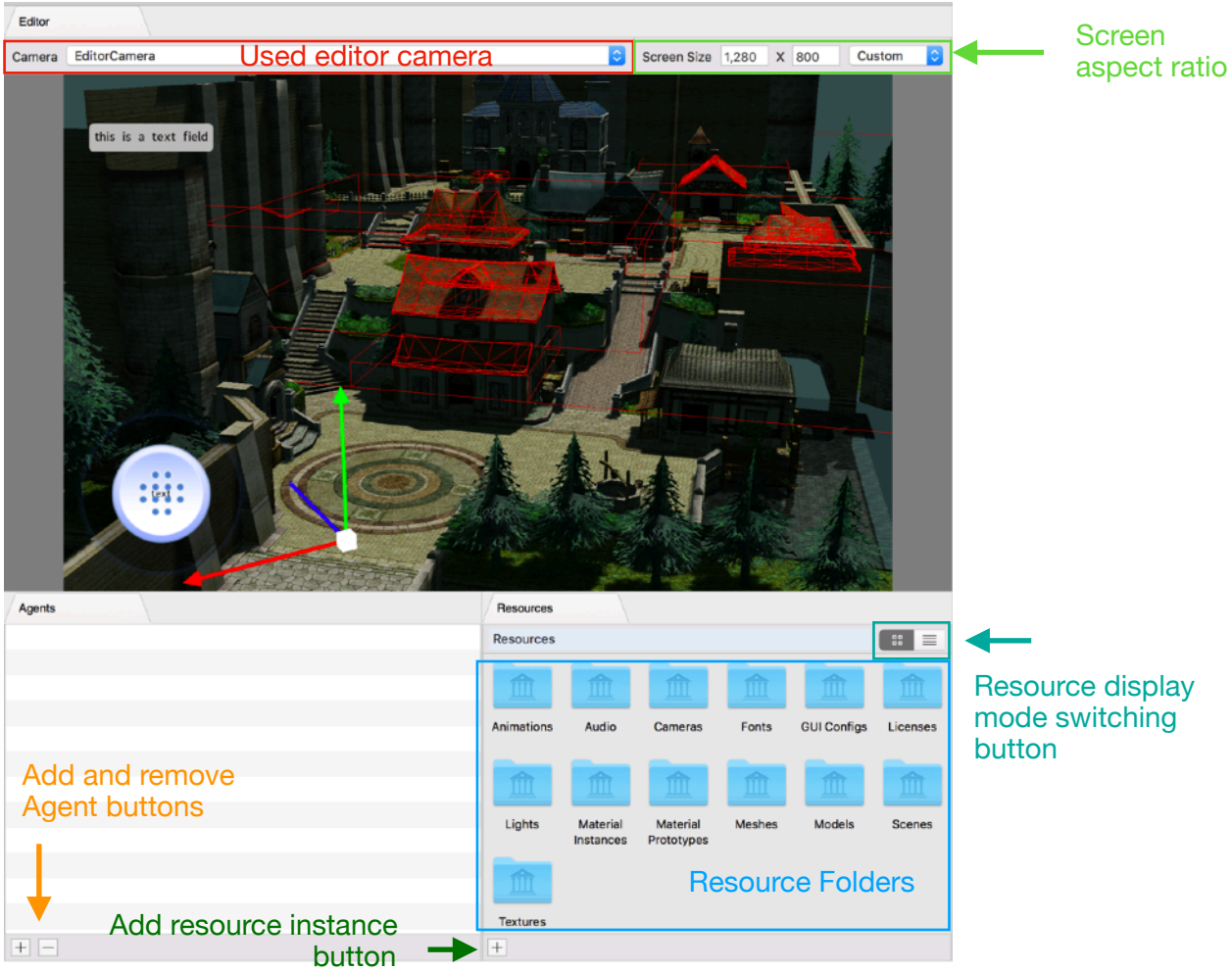
Sub path name	Description
Frameworks	The framework needed for macOS platform and iOS platform
iOSLogic.xcodeproj	Logic for Player on the iOS Platform
iOSPlayer.xcodeproj	Main body Xcode project of player of iOS platform
macOSLogic.xcodeproj	Logic for Player on the macOS Platform
macOSPlayer.xcodeproj	Main body Xcode project of player of macOS platform
macOSPlayer_Debug.xcodeproj	The debug version of the main body Xcode project of the macOS platform player does not copy resources after compilation.

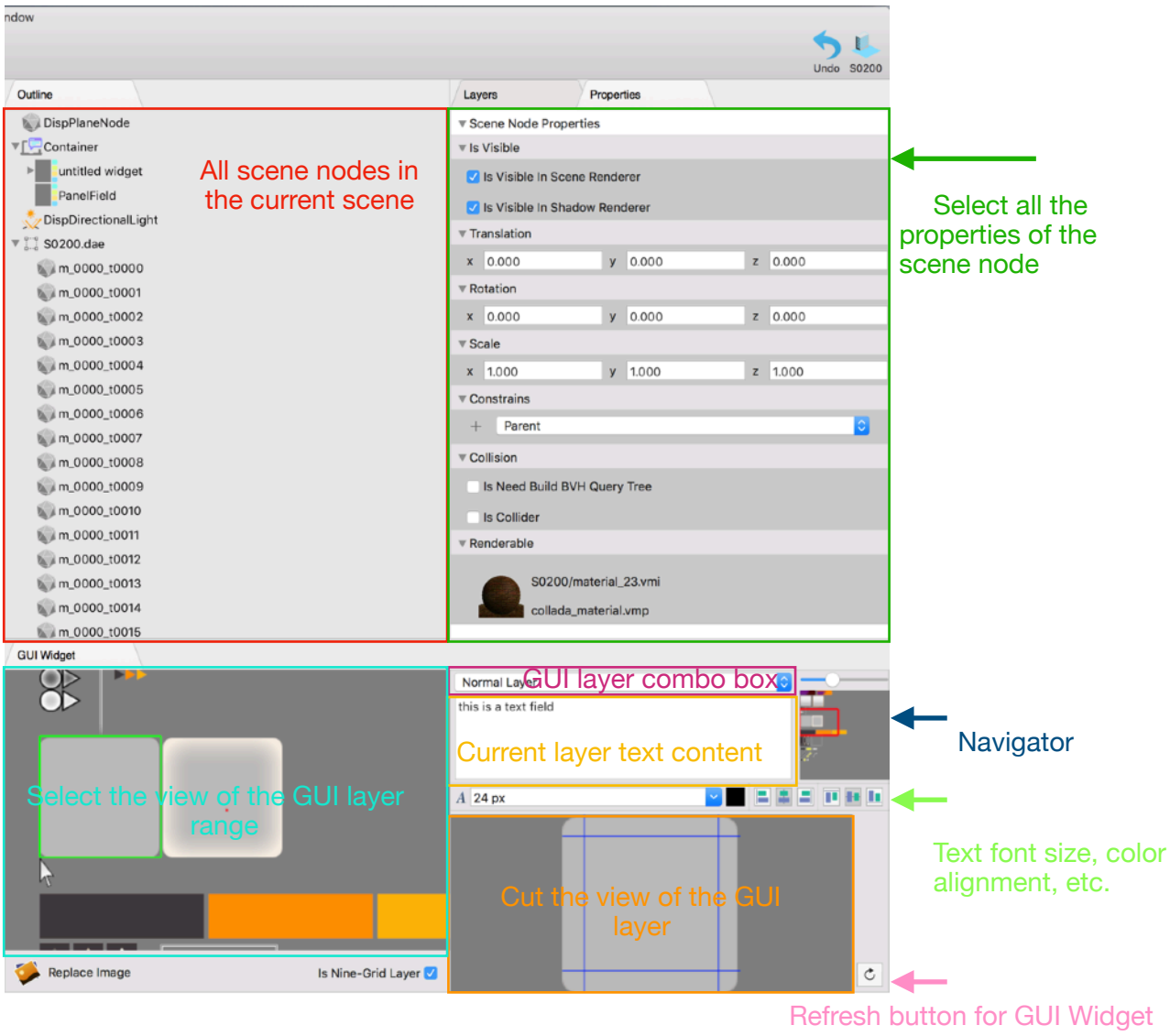
Editor:

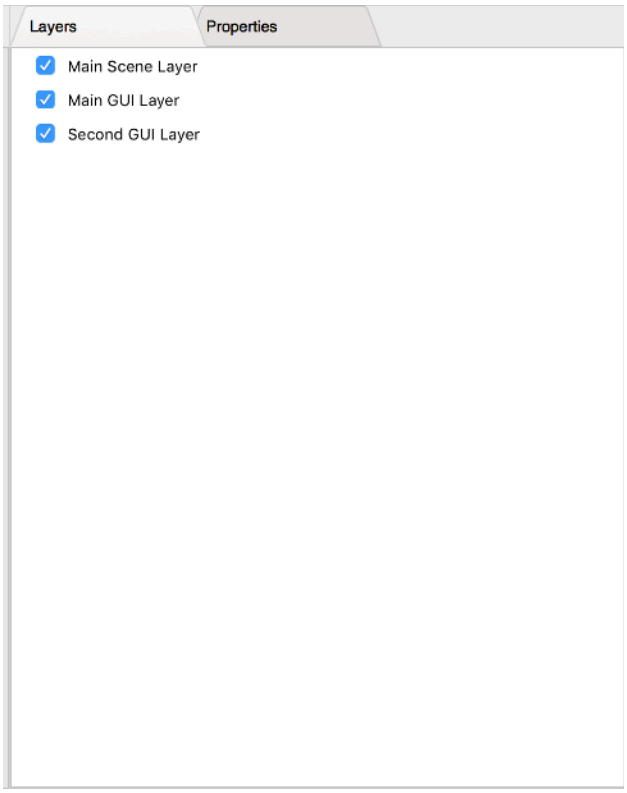


1. Basic layout

As shown in the above figure, this is the content displayed by the Editor after opening a scene, as described below.







Scene render layer enable view
a. The main scene layer is the object in the scene other than the GUI
b. The main GUI layer is the higher priority GUI layer
c. The second GUI layer is a lower priority GUI layer

2. Basic operation

a. Rotate the camera: Hold the Command key and press the left mouse button to drag the mouse

b. Pan camera: Hold down the Option key Press the left mouse button

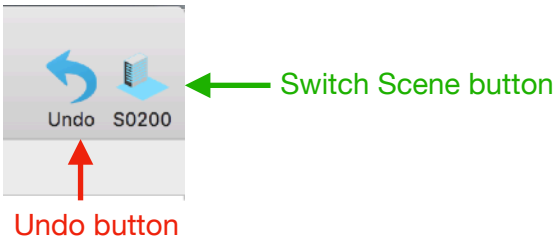
c. Tracking shot: Hold down Command key and press the right mouse button

d. The camera focuses on an object: Select a 3D object and press the shortcut key f

e. Switch 3D Manipulator to Translation Manipulator: Select a 3D object and press shortcut key w

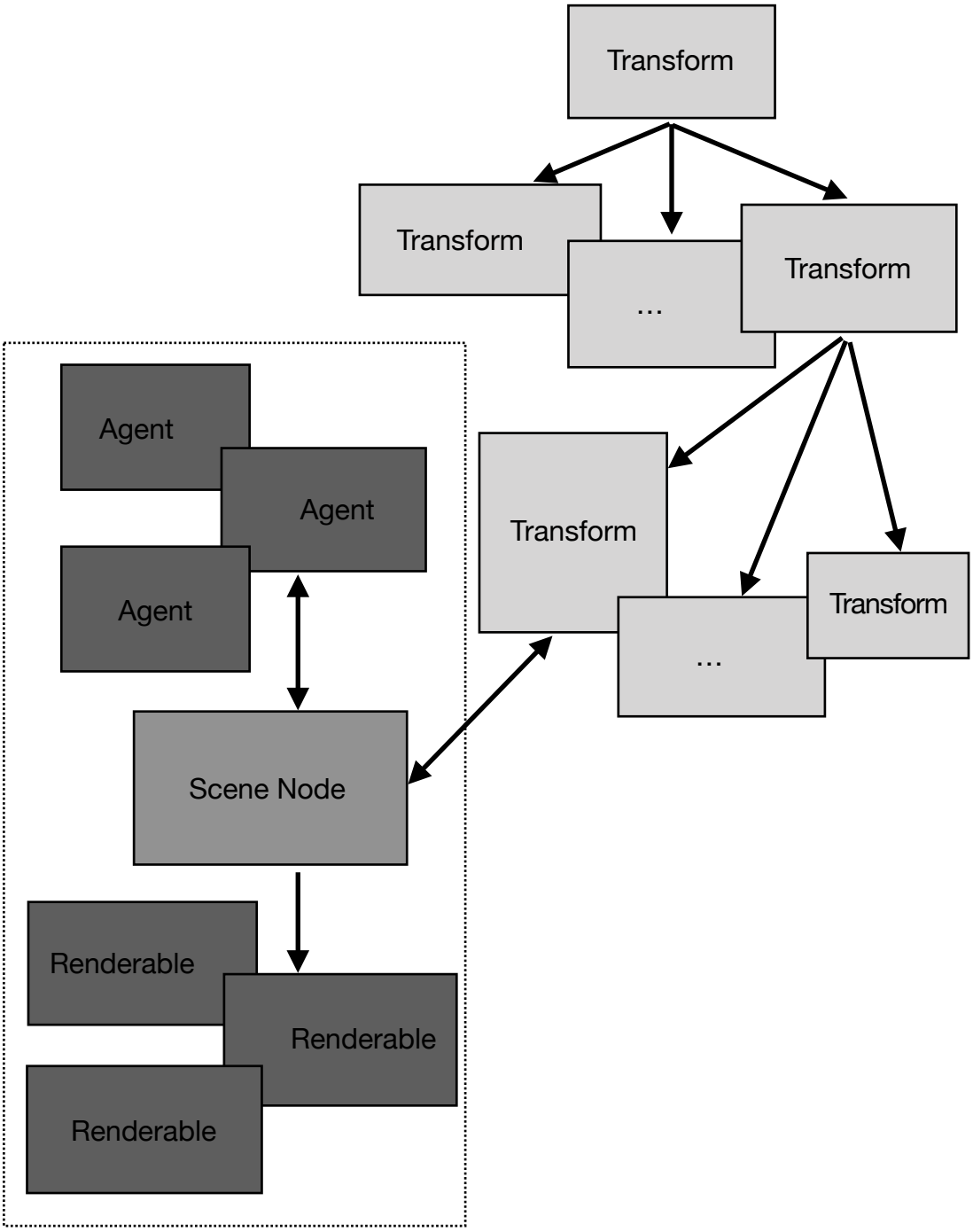
f. Switch 3D Manipulator to Rotation Manipulator: Select a 3D object and press shortcut key e

g. Switch 3D Manipulator to Scale Manipulator: Select a 3D object and press shortcut key r



Scene Node System:

1.The hierarchical structure of the scene



As shown in the above figure, all the objects in the scene are tree hierarchically organized by Transform. Each Transform below mounts a SceneNode, and a SceneNode contains a group of Agents and a set of Renderables. This is the general structure of the current scene organization.

2. What is Transform?

Transform gives the translation, rotation, and scaling of an object in its space. There are two main types of Transform.

a. Transform in 3D space.

b. Transform in the 2D GUI space.

Transform has a hierarchical relationship. When the parent Transform changes, all child Transforms under the parent Transform will change recursively.

3. What is a SceneNode?

SceneNode is a container containing Agents and Renderables. SceneNode and Transform hold each other. The coordinate data of all Renderables under SceneNode comes from the Transform that SceneNode holds. SceneManager will update all the Transform and SceneNode in the scene every frame.

4. What is an Agent?

Agent is a game logic control object written in Swift, see the Agent System chapter.

Resource System:

1. Why do you need a Resource System?

There are two reasons,

a.Need to implement asynchronous loading of resources and caching of loaded resources.

b.Need to implement media-independent resource loading.

The first one is well understood, and the second is what media-independent resource reads? After the game is packaged, the resources may exist in different places. It may be in the hard disk, possibly in the flash memory, and possibly in the compressed package on the network. The resources exist in various forms, but the use method is the same. It is a media-independent resource load.

2. Supported resource formats

扩展名	描述
wav	Audio resource format
dae	COLLADA resource format
ttf、ttc	Font resource format
lic	License resource format
vmi、vmp	Material Instance and Material Prototype resource formats
plist	Scene and project resource formats
png、dds	Texture resource formats

3. Dynamic update of resources

When you open the Editor, any changes to the resource system are detected in the Editor, especially when you rename the resource. Any renaming of resources will lead to adaptive adjustment of the entire scene.

4. Import the COLLADA file

The texture path used in a COLLADA is an absolute path. When the default project is created, a subdirectory named COLLADATextures is automatically created. The textures referenced by the COLLADA files in the default project are all in the COLLADATextures subdirectory.When these COLLADA files are imported into the scene through the Import COLLADA submenu item of the File menu, if the file references a texture, the corresponding texture is automatically imported into the resource system.

Material System:

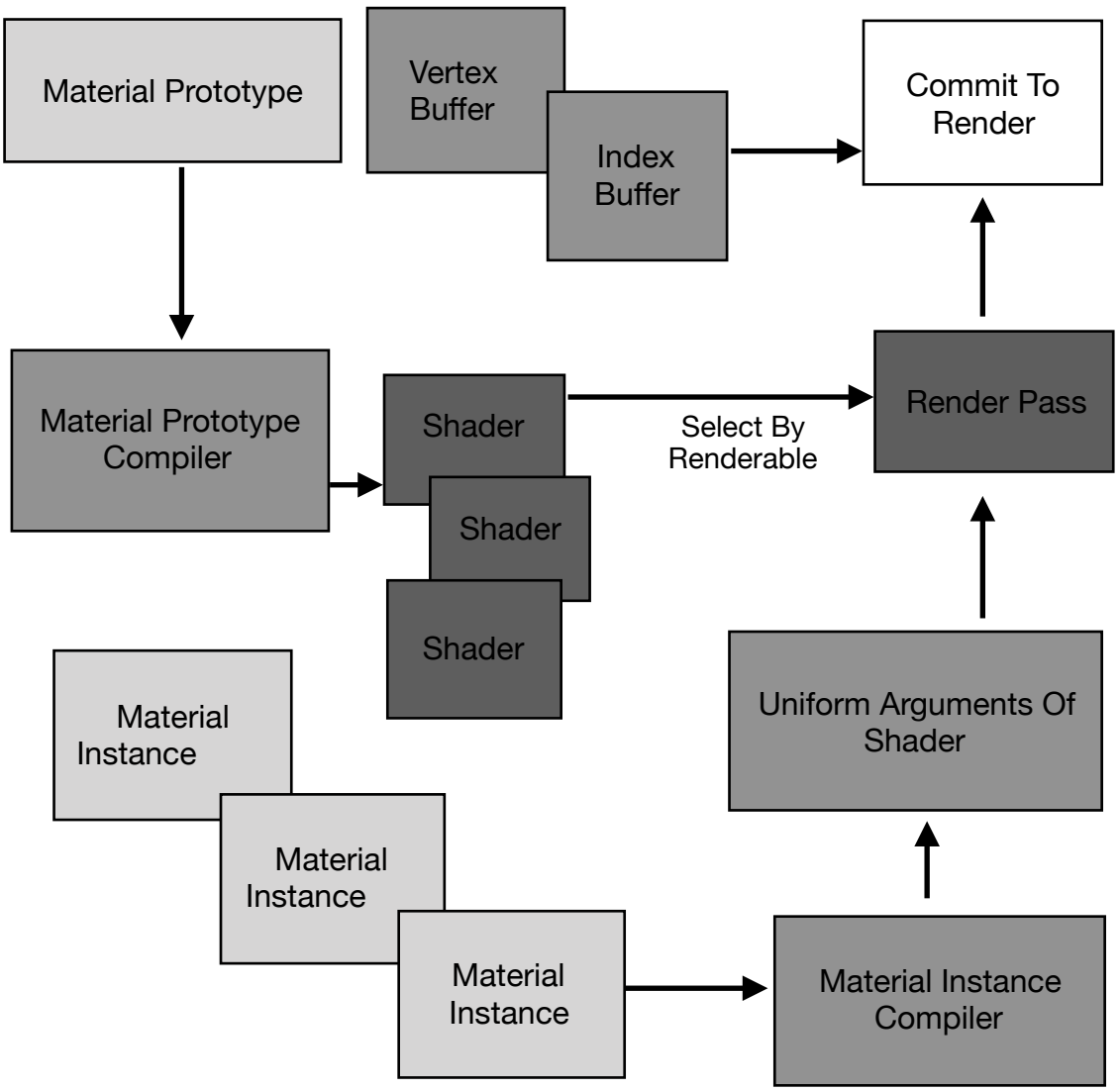
1. Why do I need a material script instead of writing a Shader?

There are two reasons.

a. Shader is related to the graphics API. Each graphics API has its own set of Shader, which leads to the need to repeatedly write shaders across APIs.

b. Shader is related to the renderer. Different renderers (forward renderers, deferred shaders, forward plus renderers, etc.) use different shaders, which causes Shader to repeatedly write.

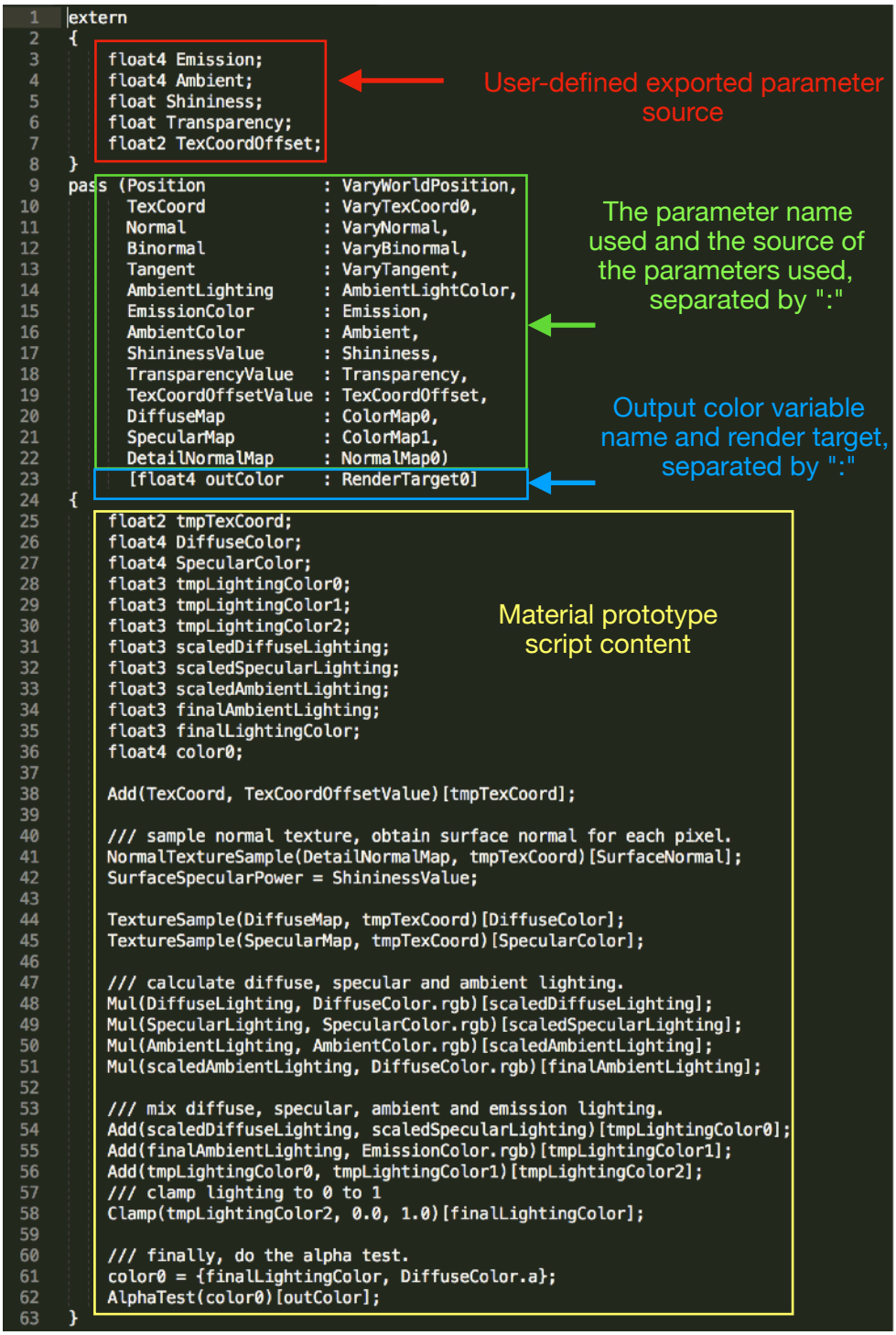
2. How does it work?



As shown in the above figure, the material is divided into two parts: the material prototype and the material instance. The material prototype is compiled into a set of Shader by the material prototype compiler, and then the appropriate Shader combination is selected as the Render Pass according to the different Renderable.On the other hand, the material instance is compiled by the material instance compiler to be the arguments used by the Shader, the arguments are set to the Render Pass, and the Vertex Buffer and Index Buffer are merged and finally submitted to the renderer for rendering.

3.Material Instance Script Format

extern { user-defined exported Parameter Source }
pass (the parameter name used : the source of the parameter used)
[float4 output color variable name : render target]
{ material prototype script content }




```
extern
{
    float4 Emission;
    float4 Ambient;
    float Shininess;
    float Transparency;
    float2 TexCoordOffset;
}
pass (Position            : VaryWorldPosition,
      TexCoord            : VaryTexCoord0,
      Normal              : VaryNormal,
      Binormal            : VaryBinormal,
      Tangent             : VaryTangent,
      AmbientLighting     : AmbientLightColor,
      EmissionColor       : Emission,
      AmbientColor        : Ambient,
      ShininessValue      : Shininess,
      TransparencyValue   : Transparency,
      TexCoordOffsetValue : TexCoordOffset,
      DiffuseMap          : ColorMap0,
      SpecularMap         : ColorMap1,
      DetailNormalMap     : NormalMap0)
      [float4 outColor    : RenderTarget0]
{
    float2 tmpTexCoord;
    float4 DiffuseColor;
    float4 SpecularColor;
    float3 tmpLightingColor0;
    float3 tmpLightingColor1;
    float3 tmpLightingColor2;
    float3 scaledDiffuseLighting;
    float3 scaledSpecularLighting;
    float3 scaledAmbientLighting;
    float3 finalAmbientLighting;
    float3 finalLightingColor;
    float4 color0;

    Add(TexCoord, TexCoordOffsetValue)[tmpTexCoord];

    /// sample normal texture, obtain surface normal for each pixel.
    NormalTextureSample(DetailNormalMap, tmpTexCoord)[SurfaceNormal];
    SurfaceSpecularPower = ShininessValue;

    TextureSample(DiffuseMap, tmpTexCoord)[DiffuseColor];
    TextureSample(SpecularMap, tmpTexCoord)[SpecularColor];

    /// calculate diffuse, specular and ambient lighting.
    Mul(DiffuseLighting, DiffuseColor.rgb)[scaledDiffuseLighting];
    Mul(SpecularLighting, SpecularColor.rgb)[scaledSpecularLighting];
    Mul(AmbientLighting, AmbientColor.rgb)[scaledAmbientLighting];
    Mul(scaledAmbientLighting, DiffuseColor.rgb)[finalAmbientLighting];

    /// mix diffuse, specular, ambient and emission lighting.
    Add(scaledDiffuseLighting, scaledSpecularLighting)[tmpLightingColor0];
    Add(finalAmbientLighting, EmissionColor.rgb)[tmpLightingColor1];
    Add(tmpLightingColor0, tmpLightingColor1)[tmpLightingColor2];
    /// clamp lighting to 0 to 1
    Clamp(tmpLightingColor2, 0.0, 1.0)[finalLightingColor];

    /// finally, do the alpha test.
    color0 = {finalLightingColor, DiffuseColor.a};
    AlphaTest(color0)[outColor];
}
```

4. What is the parameter source?

The parameter source is an id, used to indicate where the parameters come from. There are two types of parameter sources.

a. Built-in parameter source.

b. User-defined parameter source.

The built-in parameter source comes from the renderer, and the user's parameter source comes from the material instance.

5. Built-in parameter source table

参数源名称	类型	描述
VaryWorldPosition	float4	World coordinates for each pixel
VaryProjPosition	float4	Each pixel coordinates after the projection matrix
VaryTexCoord0	float2	The first set of texture coordinates
VaryTexCoord1	float2	The second set of texture coordinates
VaryTexCoord2	float2	The third set of texture coordinates
VaryTexCoord3	float2	The fourth set of texture coordinates
VaryNormal	float3	The normal value of each pixel
VaryTangent	float3	The tangent value of each pixel
VaryBinormal	float3	The binormal value of each pixel
VaryColor	float4	The color value of each pixel
CameraPosition	float3	Camera world coordinates
CameraDirection	float3	Camera world direction
ColorMap0	texture2D	The first color texture, if you use the alpha test, the texture will be used to render the alpha test of the shadow map
ColorMap1	texture2D	The second color texture
ColorMap2	texture2D	The third color texture
ColorMap3	texture2D	The fourth color texture
NormalMap0	texture2D	Normal texture
EnvironmentMap0	textureCube	The first cube texture
EnvironmentMap1	textureCube	The second cube texture

6. The syntax of the material prototype script content

Syntax	Variable declaration
Format	data type variable name; data type variable name = variable value;
Example	float2 tmpTexCoord; float2 tmpTexCoord = {0.0, 0.0};
Description	Declaring variables and assigning them (optional)

Syntax	Variable assignment
Format	variable name = variable value;
Example	tmpTexCoord = {0.0, 0.0};

Description	Assign a value to a variable

Syntax	Add Shader node
Format	Shader Node name(Input parameters)[Output parameters];
Example	Add(TexCoord, TexCoordOffsetValue)[tmpTexCoord]; Sub(1.0, TransparencyValue)[invertedTransparencyValue];
Description	Add a Shader node

7. Why use Shader nodes?

The concept of the Shader Node exists for easy use with editors. Ideally, you don't need to write a single line of code. You can use the Editor to edit the material.In order to achieve this goal, some commonly used Shader logic is encapsulated into the Shader Node for use by the Editor. (Now VEngine doesn't implement the material editor. It will be added sometime in the future)

8.Common Shader Node table

Shader Node name	Input parameters	Output parameters	Description
TextureSample	(texture2D texture, float2 texture coordinate)	[float3 rgb texel value] [float4 rgba texel value]	Sampling 2D textures through a 2D texture coordinate to output a texel value
CubeTextureSample	(textureCube texture, float3 texture coordinate)	[float3 rgb texel value] [float4 rgba texel value]	Sampling a cubic texture through a 3D texture coordinate to output a texel value
NormalTextureSample	(texture2D normal texture, float2 texture coordinate)	[float3 xyz normal value]	Sampling 2D normal textures through a 2D texture coordinate to output a normal value
Add	(float value a, float value b) (float2 value a, float2 value b) (float3 value a, float3 value b) (float4 value a, float4 value b)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Add the two values a and b to output
Sub	(float value a, float value b) (float2 value a, float2 value b) (float3 value a, float3 value b) (float4 value a, float4 value b)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Subtraction the two values a and b to output

Shader Node name	Input parameters	Output parameters	Description
Mul	(float value a, float value b) (float2 value a, float2 value b) (float3 value a, float3 value b) (float4 value a, float4 value b)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Multiply the values of a and b to output
Scale	(float input value, float scaling value) (float2 input value, float scaling value) (float3 input value, float scaling value) (float4 input value, float scaling value)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Multiply the input value by the scaling value to output
Clamp	(float input value, float min, float max) (float2 input value, float min, float max) (float3 input value, float min, float max) (float4 input value, float min, float max)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Limit the input value between the given maximum and minimum values
Pow	(float input value, float power) (float2 input value, float power) (float3 input value, float power) (float4 input value, float power)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Performs a power operation on the input value
Dot	(float2 value a, float2 value b) (float3 value a, float3 value b) (float4 value a, float4 value b)	[float2 output value] [float3 output value] [float4 output value]	Dot product of two vector values a and b to output
Cross	(float2 value a, float2 value b) (float3 value a, float3 value b) (float4 value a, float4 value b)	[float2 output value] [float3 output value] [float4 output value]	Cross product of two vector values a and b to output
Normalize	(float2 input value) (float3 input value) (float4 input value)	[float2 output value] [float3 output value] [float4 output value]	Normalized input vector value

Shader Node name	Input parameters	Output parameters	Description
Mix	(float value a, float value b, float interpolant) (float2 value a, float2 value b, float interpolant) (float3 value a, float3 value b, float interpolant) (float4 value a, float4 value b, float interpolant)	[float output value] [float2 output value] [float3 output value] [float4 output value]	Interpolation of the input value a and the input value b to output
GaussianBlurSample	(texture2D texture, float2 texture coordinate, float2 texture resolution, float blur radius)	[float4 rgba texel value]	Gaussian blur sampling of a given radius for a texture, outputting a texel (blur radius 1 to 5, poor performance)
HoriGaussianBlurSample2	(texture2D texture, float2 texture coordinate, float2 texture resolution)	[float4 rgba texel value]	Horizontal Gaussian Blur sampling with a radius of 2 pixels for a texture, outputting a texel (good performance)
VertGaussianBlurSample2	(texture2D texture, float2 texture coordinate, float2 texture resolution)	[float4 rgba texel value]	Vertical Gaussian Blur sampling with a radius of 2 pixels for a texture, outputting a texel (good performance
AlphaTest	(float4 input color)	[float4 output color]	Alpha test by the alpha value of the input color, usually used at the end of the material script

9.Material Instance Format

instance (“ Material prototype name ”, Additional attributes (optional))
{ Material Instance Script Contents }

Material prototype name

```
instance ( "phong_EcAcDt0ScNt0_AB.vmp", Transparent:2 ) Additional attributes (optional)
{
    float4 EmissionColor = {0.000000, 0.000000, 0.000000, 1.000000};
    float4 AmbientColor = {0.200000, 0.200000, 0.200000, 1.000000};
    texture2D DiffuseMap (Filter:Bilinear, WrapU:Repeat, WrapV:Repeat) = "S1000/10water26.dds";
    float4 SpecularColor = {0.200000, 0.200000, 0.200000, 1.000000};
    texture2D DetailNormalMap = "default_normal_map.png";
    float ShininessValue = 20.000000;
    float TransparencyValue = 0.500000;
    float2 TexCoordOffsetValue = {0.000000, 0.000000};
}
```

Material Instance Script Contents

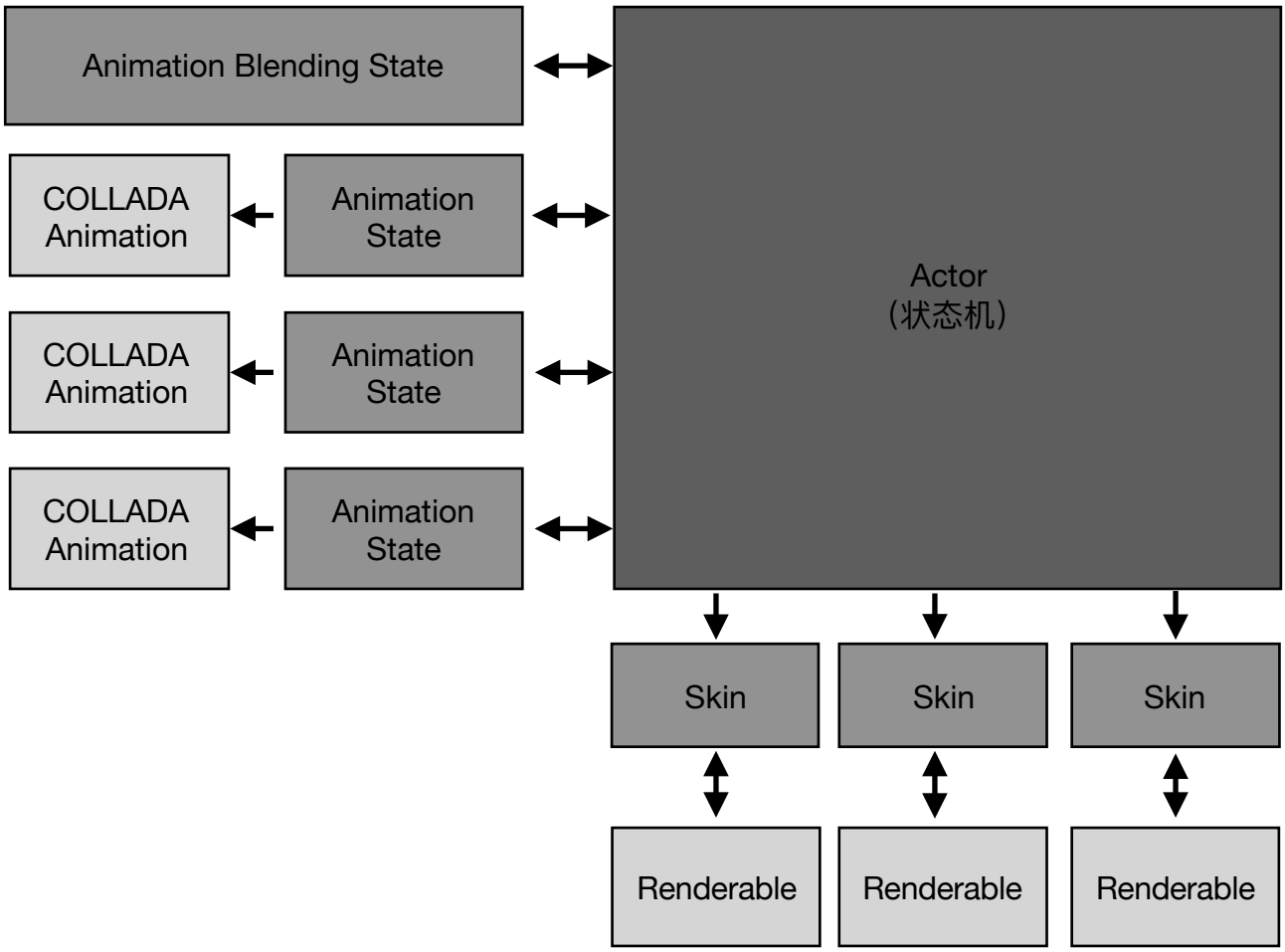
10.The syntax of the material instance script content

Syntax	Argument assignment
Format	data type argument name = argument value;
Example	float4 EmissionColor = {0.0, 0.0, 0.0, 1.0}; float TransparencyValue = 0.5;
Description	Assign a basic type of argument

Syntax	Texture assignment
Format	texture type texture argument name = “texture resource name”; texture type texture argument name (texture attributes) = “texture resource name”;
Example	texture2D DiffuseMap (Filter:Bilinear, WrapU:Repeat, WrapV:Repeat) = “S1000/10water26.dds”; texture2D DetailNormalMap = “default_normal_map.png”
Description	Assign a texture type of argument

Animation System:

1.How does it work?



As shown in the above figure, the core of the animation system is Actor, and Actor is a state machine.An Actor contains several Animation States. Each Animation State corresponds to a COLLADA animation. There is an Animation Blending State used to implement multiple animation blending between multiple Animations.Actor is always running in a single animation state or mixed animation state.The Actor calculates all the skeletal matrices for each frame of the entire animation and passes these matrices to the Skins attached to the Actor. Finally submitted to the Renderable for rendering.

2.How to use animations?

a. Separate 3D models and animations when exporting COLLADA resources. The animations are exported as separate files.

b. Put all the skinned 3D models and animation resources into the Characters directory (you can create subdirectories for easy management).

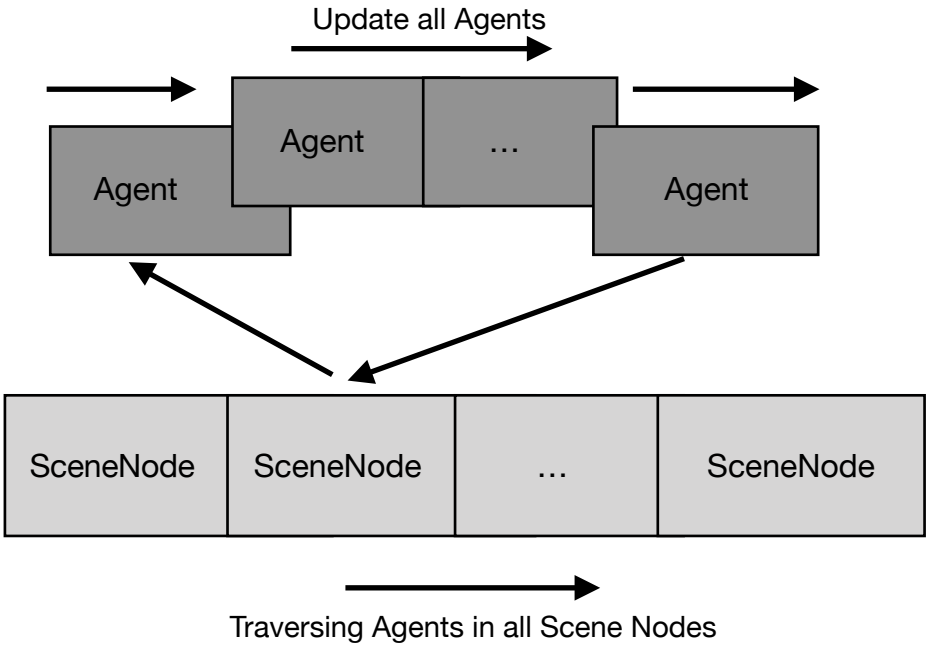
c. Open the Editor, it will automatically scan all COLLADA resources, and automatically generate animation reference files in the game logic project.

d. Open the game project with Xcode and write the corresponding ActorAgent (for details, see the Agent System section).

e. Compiling the game project will automatically export all Agent information. At this time, open the Editor and add the ActorAgent to the scene node in the agent view.

Agent System:

1.How does the Agent System work?



As shown in the figure, a single thread will traverse all Scene Nodes within the scene at the same frequency as the rendering thread, and update the Agent under each Scene Node.

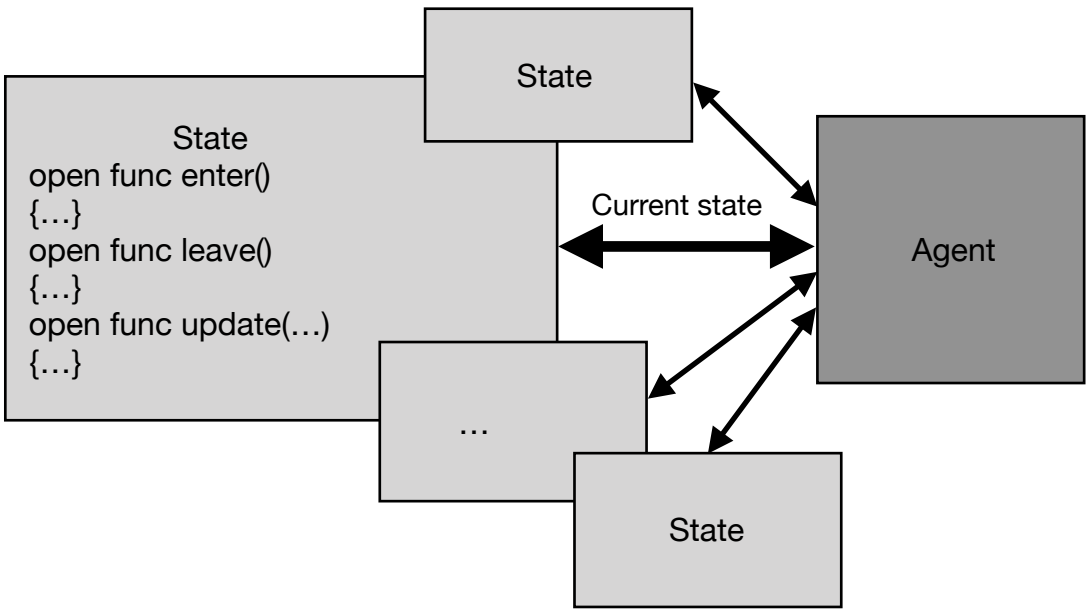
There are three types of Agents, a.SceneNodeAgent. b.ActorAgent. c.GUIAgent.

SceneNodeAgent can control the translation, rotation, scaling animation of a 3D Scene Node, and the animation switching of ActorAgent under the Scene Node, as well as the material arguments of the renderable.

ActorAgent is mainly used to manage the animation of an Actor.

GUIAgent is mainly used to handle the logic of the GUI.

2.How does a single Agent work?



As shown in the above figure, Agent is a state machine. Each Agent contains several states. Each state contains at least three methods: enter, leave, and update. And there is only one current state.

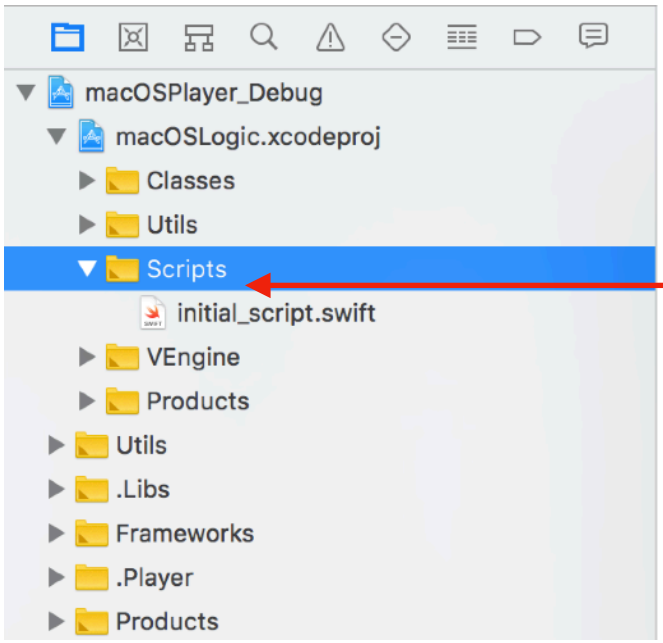
a. When an agent updates it will call its own current state of the "update" method.

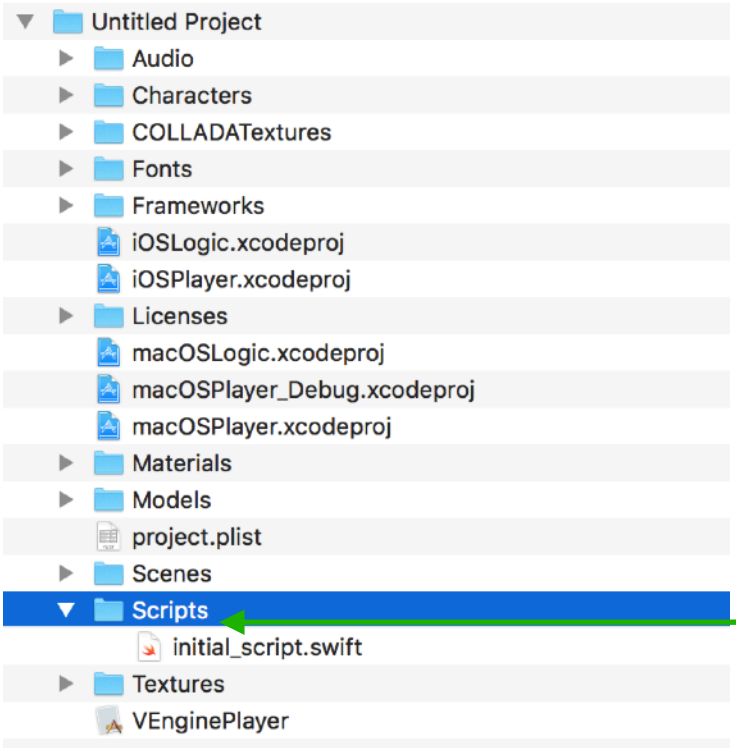
b. When a state transitions to another state, the "leave" of the current state and the "enter" method of the target state are called. After the conversion is completed, the target state becomes the new current state.

3. Write the simplest Agent code

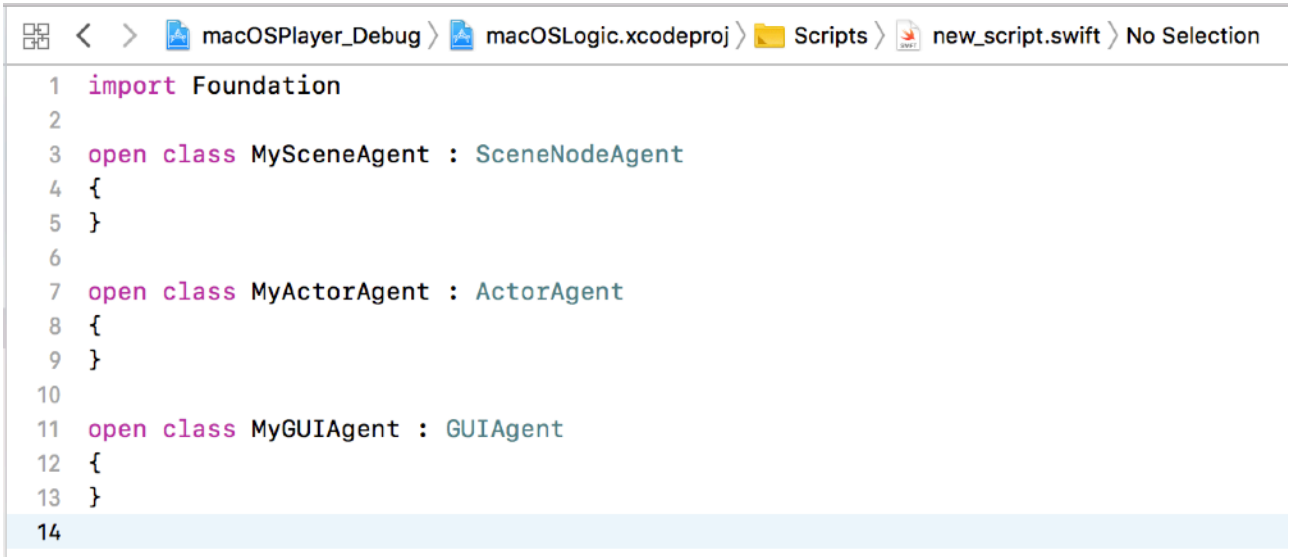
a. First open the macOSPlayer_Debug.xcodeproj in the game project.

b. Add the Swift file in the following location. The added file is guaranteed to be placed in the Scripts subdirectory under the project.

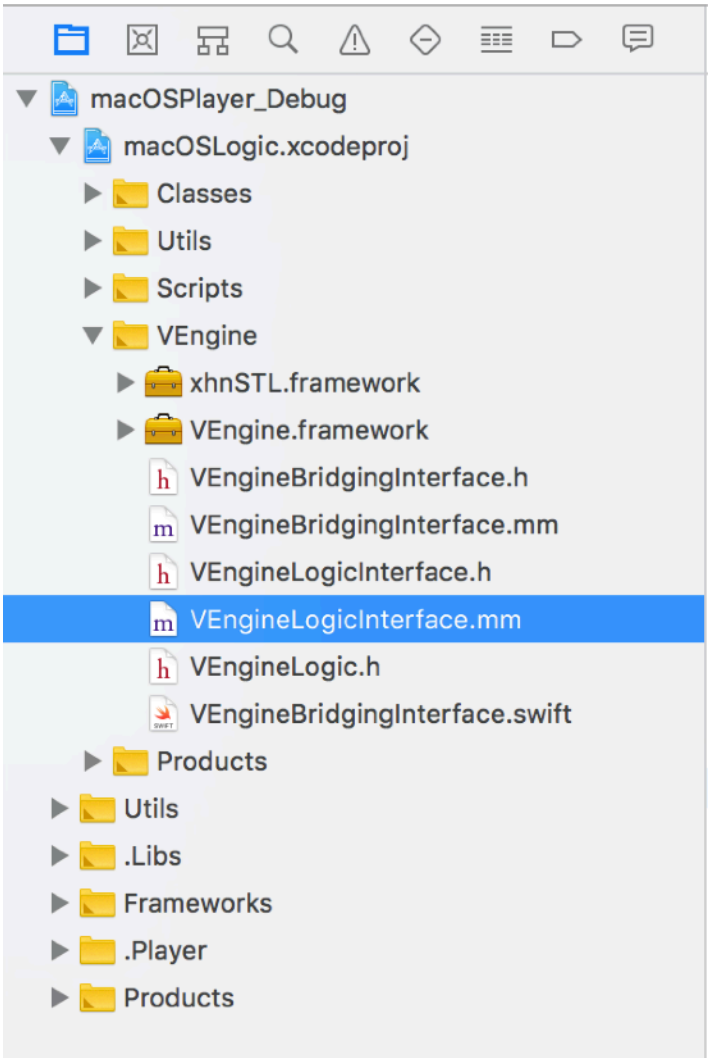




c. Write the following in the newly added Swift file.



d. Compile the entire project.
e. Find VEngineLogicInterface.mm



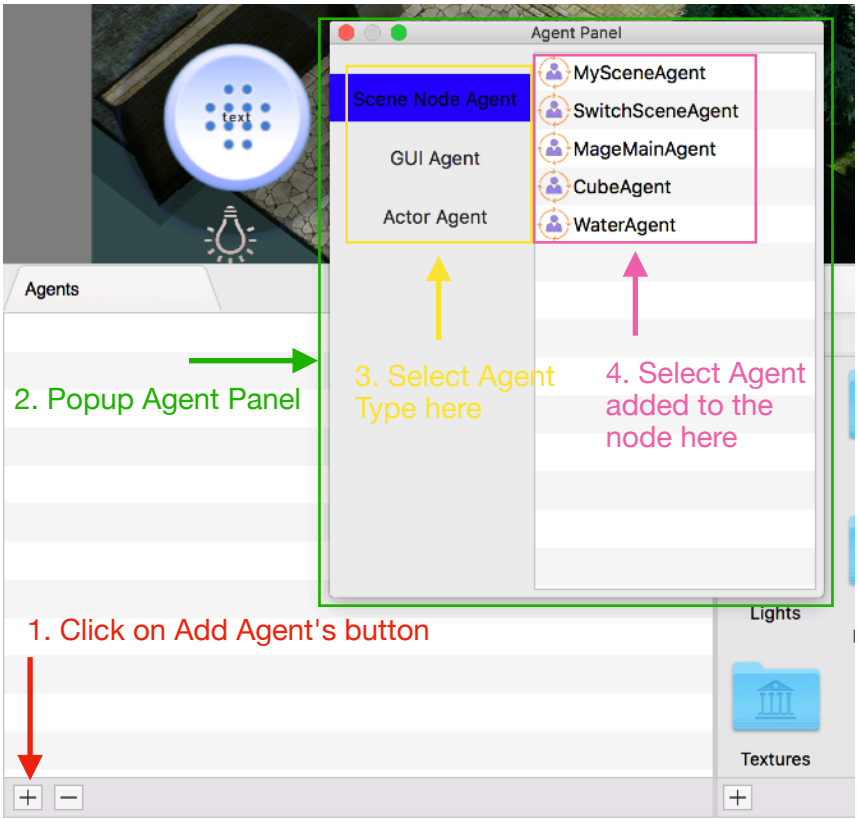
See the following contents in the file, indicating that three empty Agents have been successfully added.

```
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. Use Agent

a. Select the Scene Node that needs to add the Agent.

b. Follow the flow below.



5.SceneNodeAgent

a. The default SceneNodeAgent base class already contains six states EmptyState, RepeatState, MoveToState, RotateToState, ScaleToState, and TransitToState.

b. If the SceneNodeAgent class and the derived class have internal access permissions that are open and inherit from SceneNodeState and SceneNodeStateInterface, the subclass will be added as a State to the Agent object.

c. If the derived class of SceneNodeAgent contains more than one state, the first state in these states will be used as the initial state after the derived class is initialized. If the derived class does not contain a state, EmptyState is used as the The default state of the scene agent.

6.ActorAgent

a. The state of each ActorAgent corresponds to a COLLADA animation file. When the ActorAgent is instantiated, the COLLADA animation file corresponding to all its states is automatically loaded into the Actor object.

b. Like SceneNodeAgent, there is a default state, which is the default action. When the ActorAgent starts, it will automatically loop the default action.

7.GUIAgent

a. A GUIAgent has only five states, normal state, hovering state, dragging state, selected state, and pressed state. The "normal state" is the initial state of a GUI widget. The "hovering state" is the state of the mouse hovering over the GUI widget. (For mobile platforms that do not have a mouse, this state is entered shortly before the state is switched to the "dragging state".), "dragging state" is the state of the widget when dragging a GUI widget, "selected state" is the state after the GUI Widget is selected, and "pressed state" is the state after the GUI Widget is pressed as a button.

b. The GUIAgent needs to be used in conjunction with the GUI Widget's behavior. For GUI Widget behavior, see the GUI System chapter.

8. The simplest state operation

```
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     }
```

As shown in the above figure, the current state is obtained by the "GetCurrentState" method of the derived class of the SceneNodeAgent of type MageMainAgent, and then converted to the "MageMainAgent.RunState" state by the operator "=>".

```
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         }
```

The state transition of the ActorAgent is shown in the above figure. Inside a SceneNodeAgent, the following method can be used:

self[ActorAgent derived class name.self]

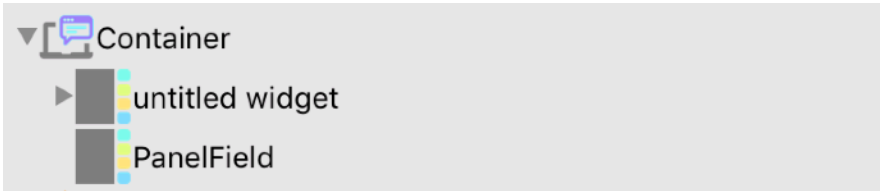
To obtain the ActorAgent under the current SceneNode, the state transition method is similar to the SceneNodeAgent.

The GUIAgent cannot manually control the state transition. The state transition of the GUIAgent completely depends on the behavior of the GUI. For related contents, refer to the GUI System chapter.

GUI System:

1.GUI layers

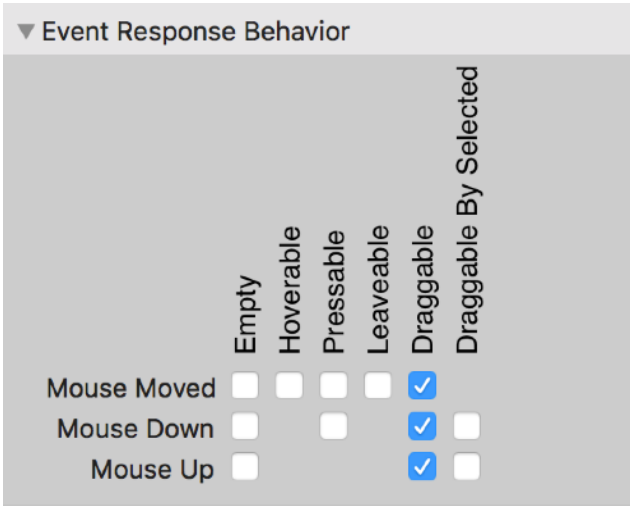
The GUI is divided into two layers: the main GUI layer and the second GUI layer. The update of the main GUI layer is synchronized with the main rendering thread, and the update of the second GUI layer is asynchronous.



As shown in the above figure, any scene contains a Scene Node named “Container”. Any GUI widget under this node will be added to the second GUI layer.

The GUI nodes can be mounted on the other 3D nodes. Any GUI nodes mounted on the 3D nodes will be added to the main GUI layer.

2.GUI Widget behavior



GUI Widget events are divided into three "Mouse Moved", "Mouse Down", and "Mouse Up".

There are five types of behaviors: "Empty", "Hoveable", "Pressable", "Leaveable", "Draggable", and "Draggable By Selected".

As shown in the above figure, the Widget's overall event response behavior is achieved by combining the Widget's behavior on three events.

Collision System:

1. Classification of collision detectors

Collision detectors are classified into global collision detector and local collision detector.

2.How does the global collision detector work?

a. Iterates over all Renderables under Scene Nodes marked as Collider.

b. Add these Renderables AABBs to the global Collision BVH tree.

c. Through the global BVH tree to find Scene Nodes that collide with each other.

d. Pass collision information to the Scene Nodes where the collision occurred.

3. How does the local collision detector work?

a.When a scene node is found during the loading of the scene, a BVH Query tree needs to be created, and then a BVH tree with the exact triangle of the scene node is constructed.

b.In the script invocation phase, collision detection can be performed by using a shape in the ray, sphere, or capsule, and a scene node that establishes a BVH Query tree.

c. The detection results can be used to control the characters in the game.

License System:

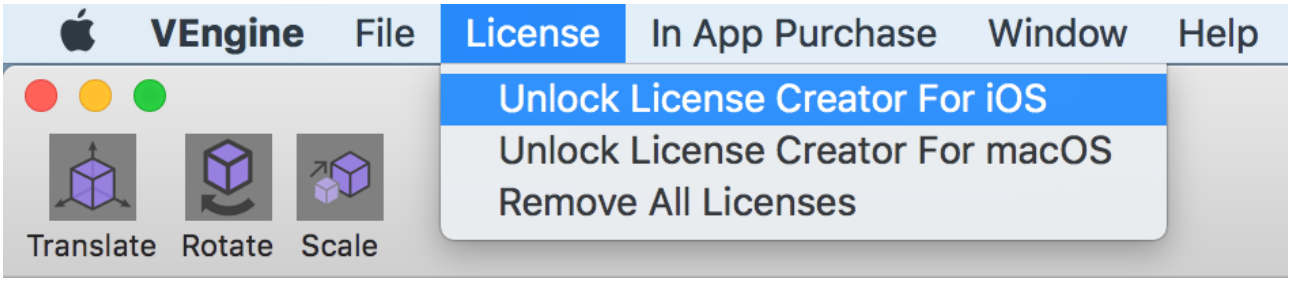
1. How does the license system work?

a. The license is bundled with the Bundle Identifier on the same platform. A license for the same platform corresponds to a Bundle Identifier.

b. The Player runs its own Bundle Identifier to check if there is a suitable license. If not, it will display the “unregistered” label in the upper left corner of the screen.

2. How to create a license?

If you want to create a license, you must unlock the capabilities of the license creator.



After the IAP payment is completed, the function of creating a license can be used.