第八周

蓝图进阶

本周内容

课程内容

- Gameplay Framework 介绍
- 蓝图间通讯
- 蓝图 Timeline
- 蓝图结构体、蓝图宏
- 蓝图与 Prefab
- 蓝图调试

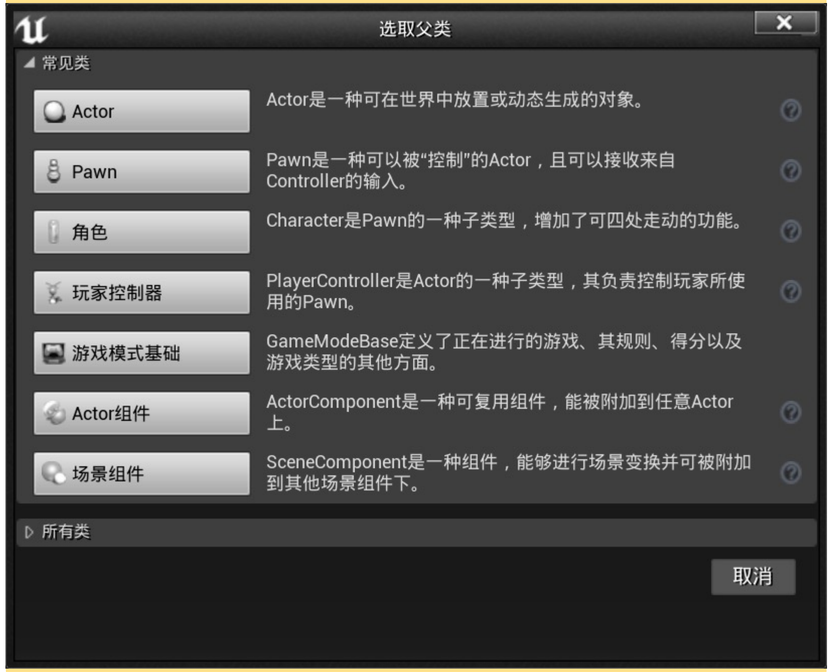
学习成果

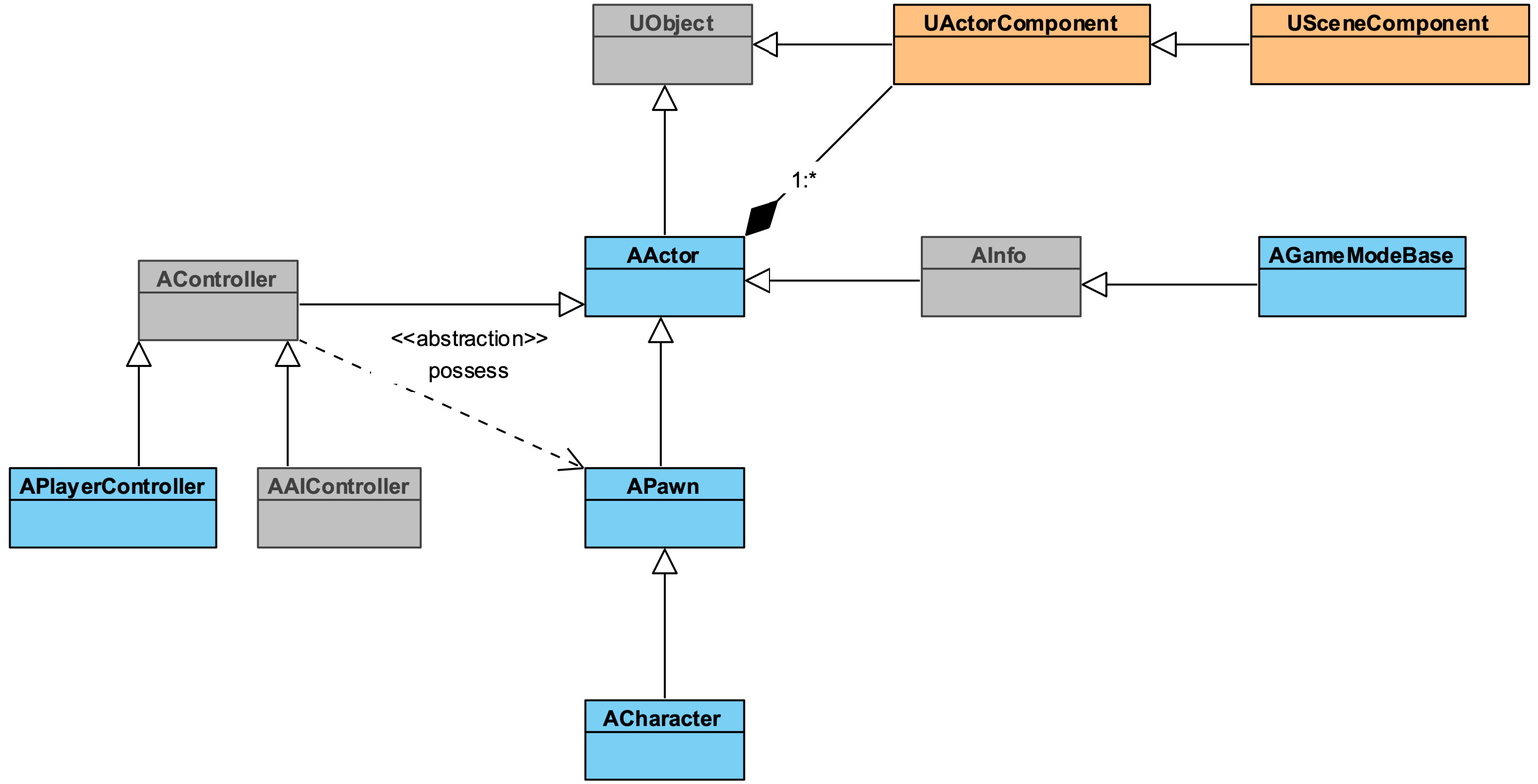
- 对蓝图开发有一个全面的理解
- 能够开始进行完整项目的蓝图开发

GAMEPLAY 框架

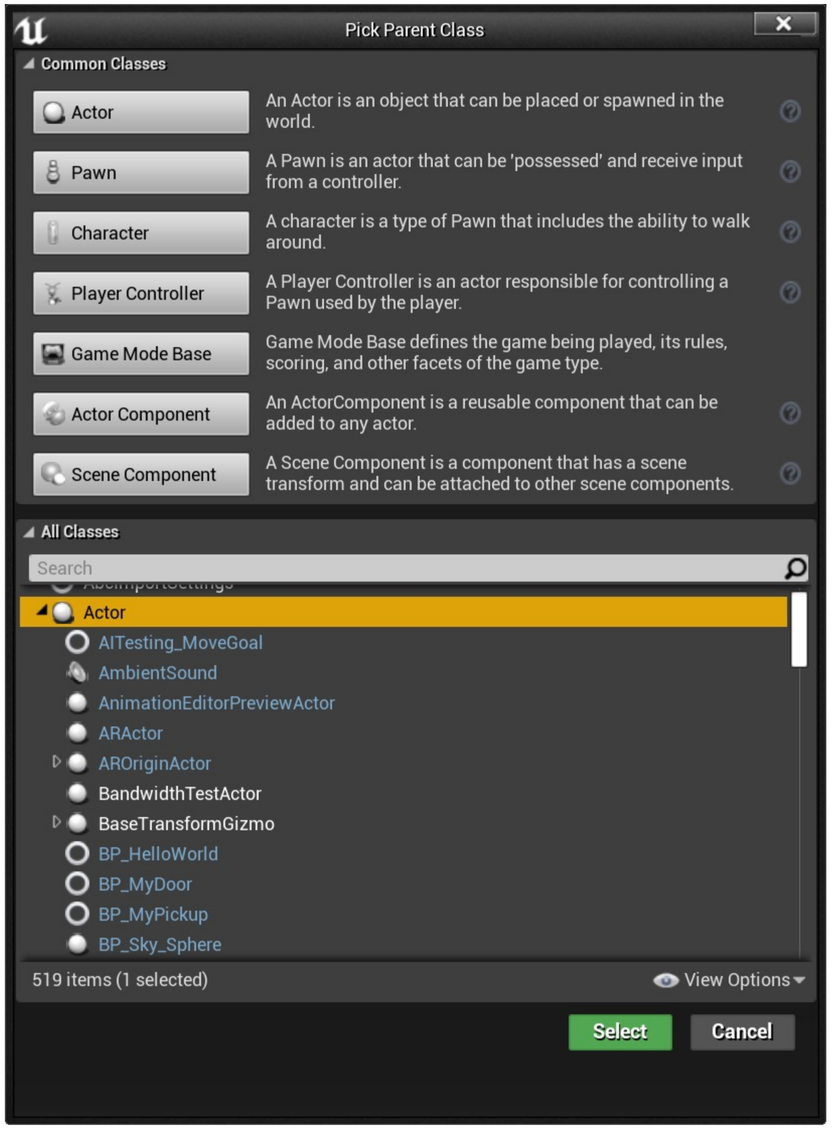
Gameplay Framework

回顾上周：选取父类


选取父类
常见类
Actor
Actor是一种可在世界中放置或动态生成的对象。
Pawn
Pawn是一种可以被“控制”的Actor，且可以接收来自Controller的输入。
角色
Character是Pawn的一种子类型，增加了可四处走动的功能。
玩家控制器
PlayerController是Actor的一种子类型，其负责控制玩家所使用的Pawn。
游戏模式基础
GameModeBase定义了正在进行的游戏、其规则、得分以及游戏类型的其他方面。
Actor组件
ActorComponent是一种可复用组件，能被附加到任意Actor上。
场景组件
SceneComponent是一种组件，能够进行场景变换并可被附加到其他场景组件下。
所有类
取消

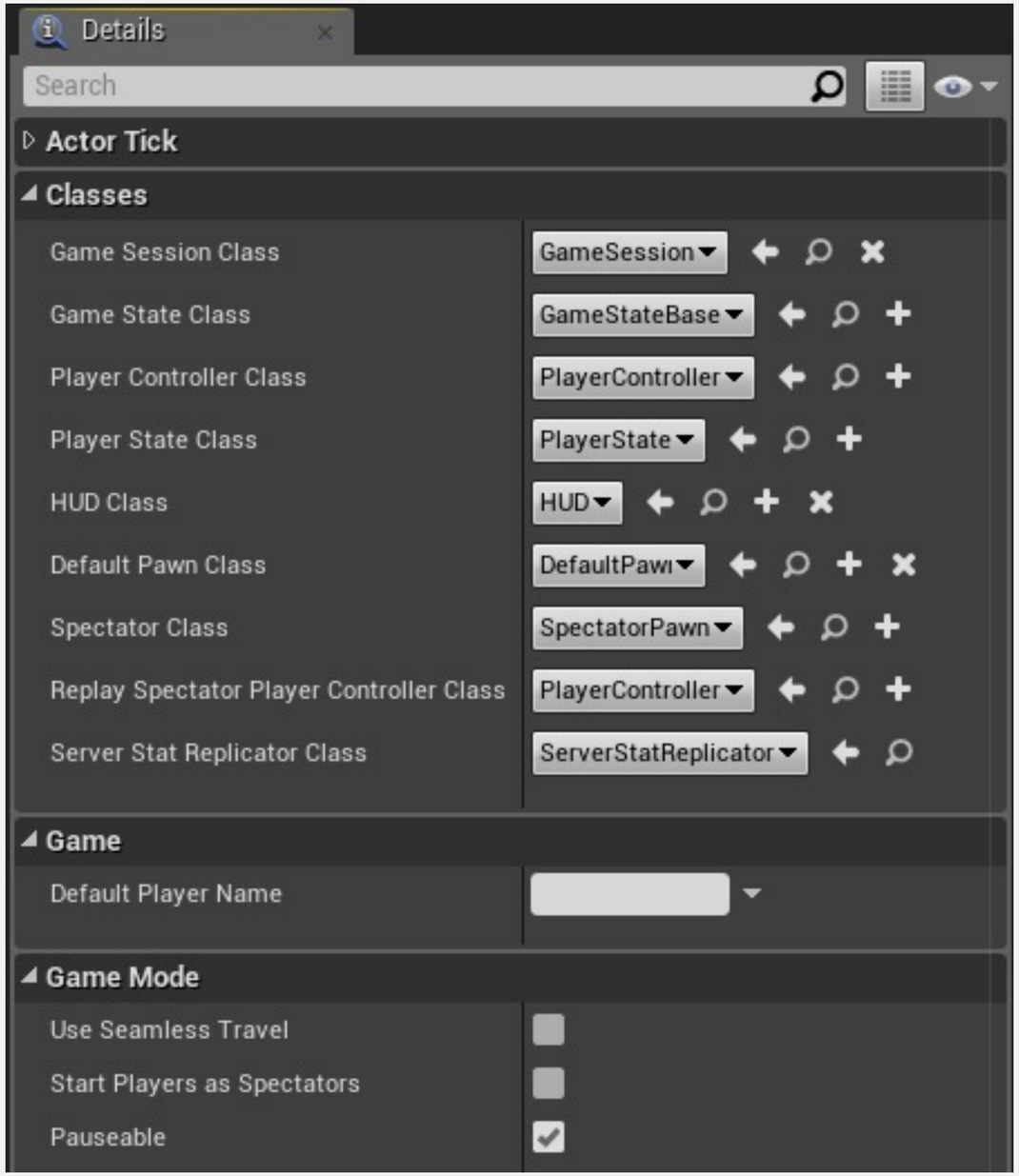

UObject
UActorComponent
USceneComponent
1:*
AActor
AInfo
AGameModeBase
AController
<<abstraction>>
possess
APlayerController
AAIController
APawn
ACharacter

OOD 设计典范


Pick Parent Class
Common Classes
Actor
An Actor is an object that can be placed or spawned in the world.
Pawn
A Pawn is an actor that can be 'possessed' and receive input from a controller.
Character
A character is a type of Pawn that includes the ability to walk around.
Player Controller
A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base
Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component
An ActorComponent is a reusable component that can be added to any actor.
Scene Component
A Scene Component is a component that has a scene transform and can be attached to other scene components.
All Classes
Search
Actor
AITesting_MoveGoal
AmbientSound
AnimationEditorPreviewActor
ARActor
AROriginActor
BandwidthTestActor
BaseTransformGizmo
BP_HelloWorld
BP_MyDoor
BP_MyPickup
BP_Sky_Sphere
519 items (1 selected)
View Options
Select
Cancel

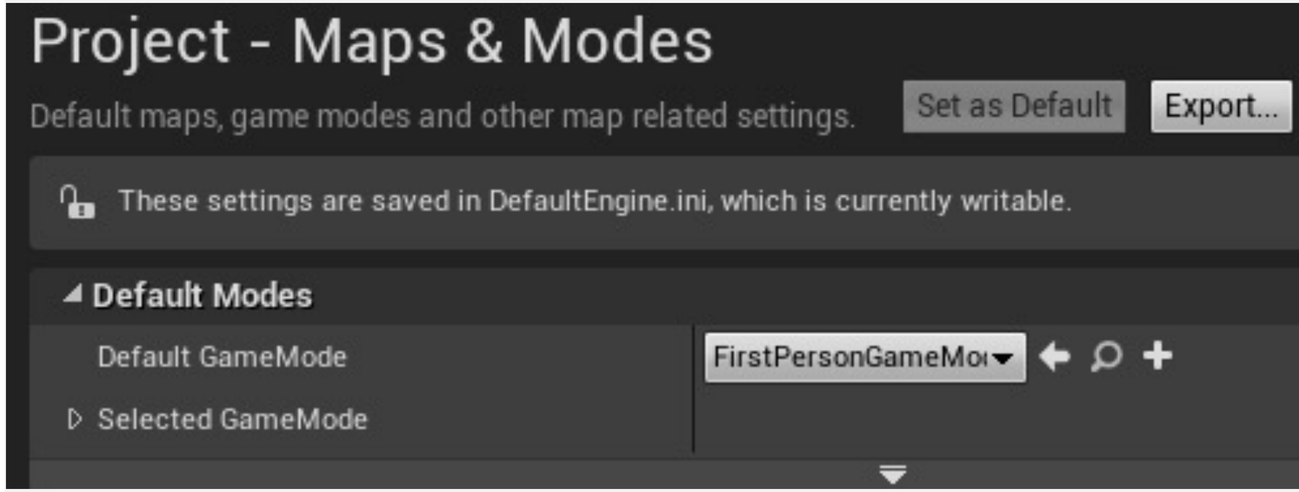
游戏模式（Game Mode）

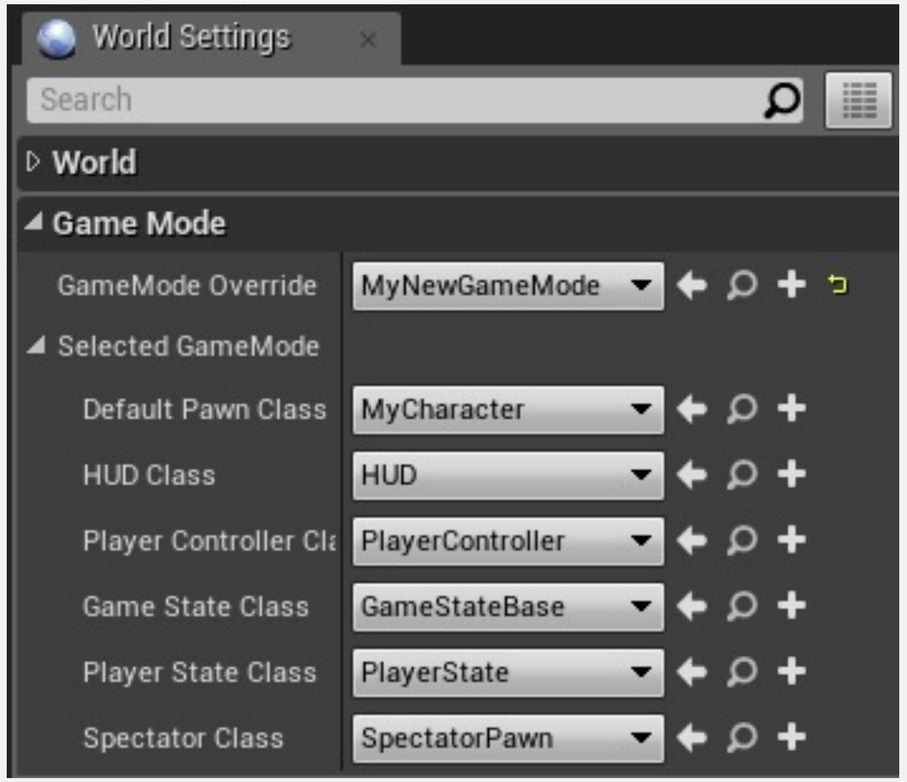
- 游戏模式类用于定义游戏的规则。
- 游戏模式还指定将用于创建Pawn、玩家控制器、游戏状态、HUD和其他类的默认类，如右图所示。
- 每个关卡都可以有不同的游戏模式。如果不为关卡指定游戏模式，则将使用已经为项目设置好的游戏模式。
- 在多人游戏中，游戏模式仅存在于服务器上，不会复制到客户端。



指定游戏模式

- 要为项目指定默认游戏模式，前往关卡编辑器中的“编辑”（Edit）>“项目设置...”（Project Settings...），然后在“项目”（Project）类别中，选择“地图和模式”（Maps & Modes）选项。在“默认游戏模式”（Default GameMode）属性下拉菜单中选择“游戏模式”（Game Mode），如右上图所示。
- 要指定关卡的游戏模式，单击关卡编辑器中的“设置”（Settings）按钮，并选择“世界场景设置”（World Settings）选项。在“游戏模式覆盖”（GameMode Override）属性下拉菜单中选择“游戏模式”（Game Mode），如右下图所示。
- 关卡的游戏模式将覆盖项目的默认游戏模式。





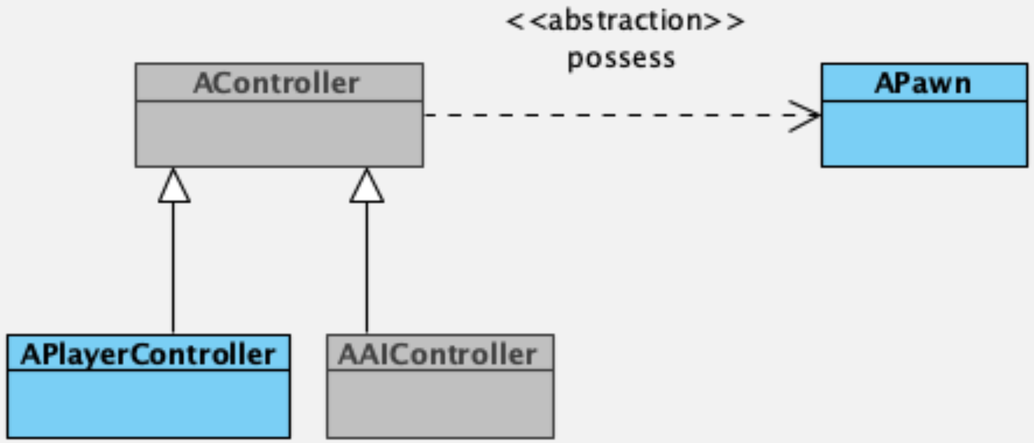
游戏状态（Game state）

游戏状态类用于记录表示游戏当前状态的变量，在多人游戏中与所有客户端共享。

基本思路是通过游戏模式在服务器上定义规则，而游戏状态则管理游戏中更改的信息，并需要发送给客户端。

要使用基于游戏状态类的新蓝图，必须将自定义游戏状态类分配给游戏模式的游戏状态类参数。

游戏状态类是从游戏状态基类延伸而来，会增加一些多人功能。

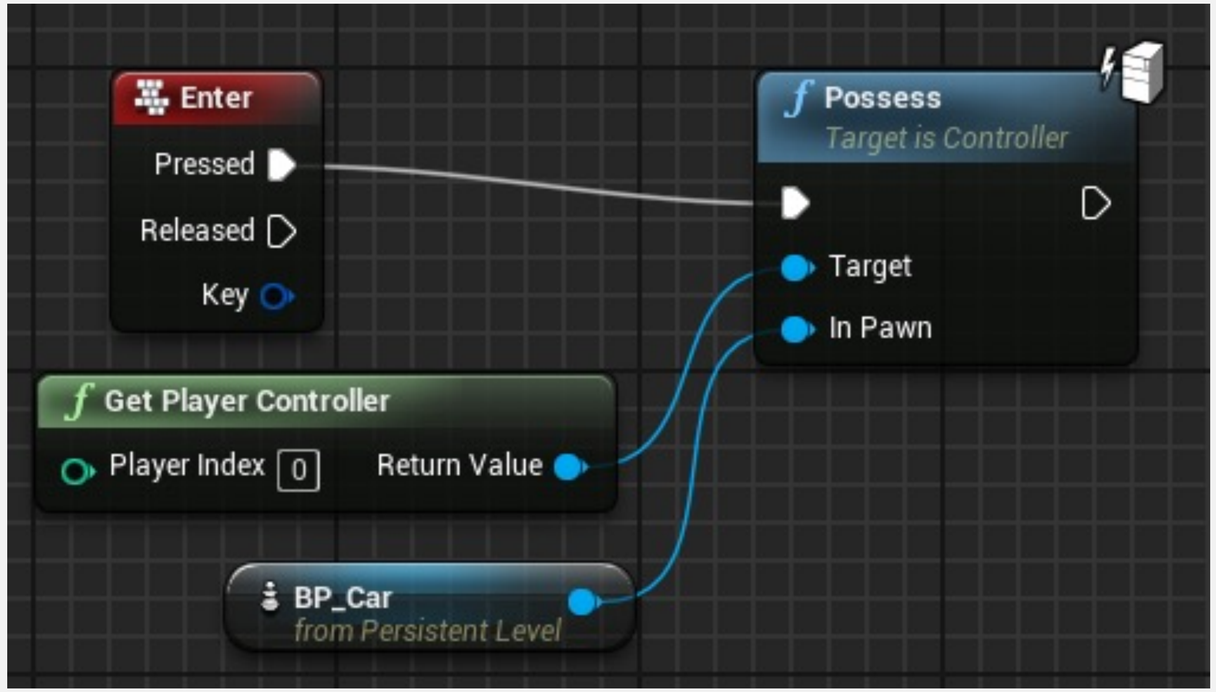


玩家控制器（Player Controller）

控制器类拥有两个主要子类。玩家控制器类由人类玩家使用，AI控制器类使用人工智能来控制Pawn。

Pawn和角色类如果由玩家控制器支配，则仅接收输入事件。

由玩家控制器支配的Pawn类可以在游戏中更改。右图来自于关卡蓝图，显示了支配（Possess）函数的用法。在该示例中，当按下Enter键时，将由玩家控制器支配关卡中的BP_Car Pawn Actor。



玩家状态（Player State）

玩家状态类用于记录特定玩家的信息，这些信息在多人游戏中需要与其他客户端共享。

玩家控制器仅存在于客户端上，而玩家状态会从服务器复制到所有客户端。

要使用基于玩家状态类的新蓝图，必须在游戏模式的玩家状态类参数中设置。

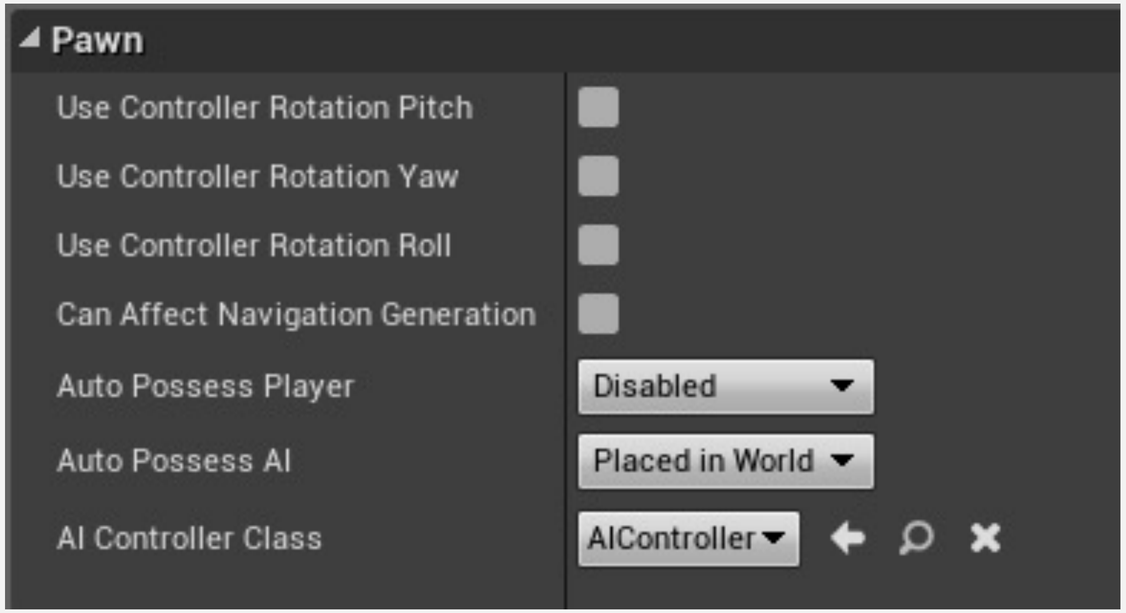
PAWN

Pawn是可以由控制器（玩家或AI）控制（支配）的Actor。

Pawn类表示身体，控制器类表示头脑。

右图显示了从Pawn类继承的一些参数。Pawn类可以使用支配它的控制器的旋转值。

其他属性表示控制器如何支配Pawn。

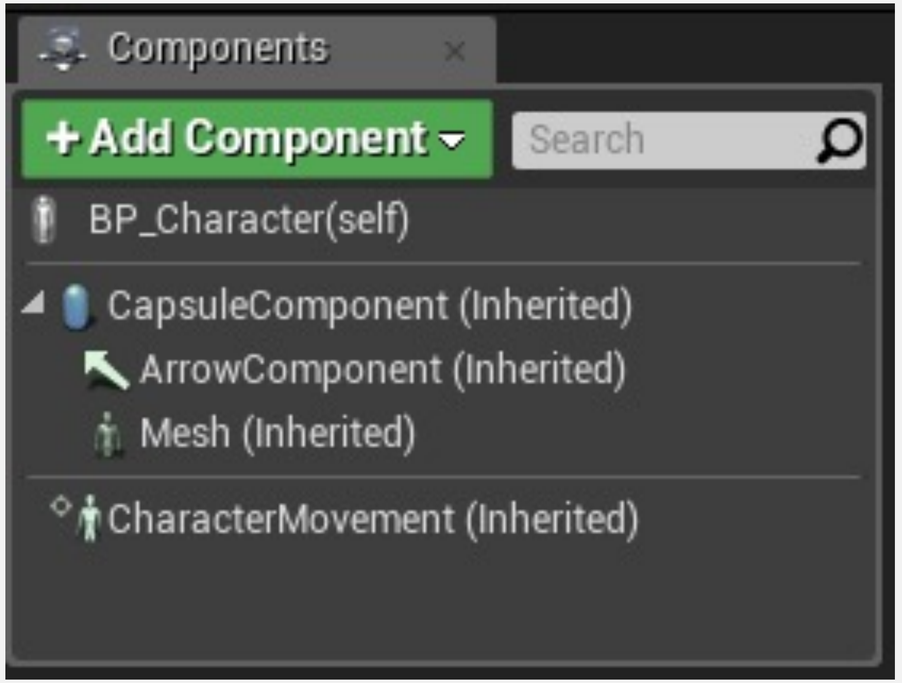


角色（Character）

角色类是Pawn类的子类，它用来表示可以行走、奔跑、跳跃、游泳和飞翔的两足角色。该类已经有一组用来帮助达到这一目的的组件。

右图显示了角色类中存在的组件。

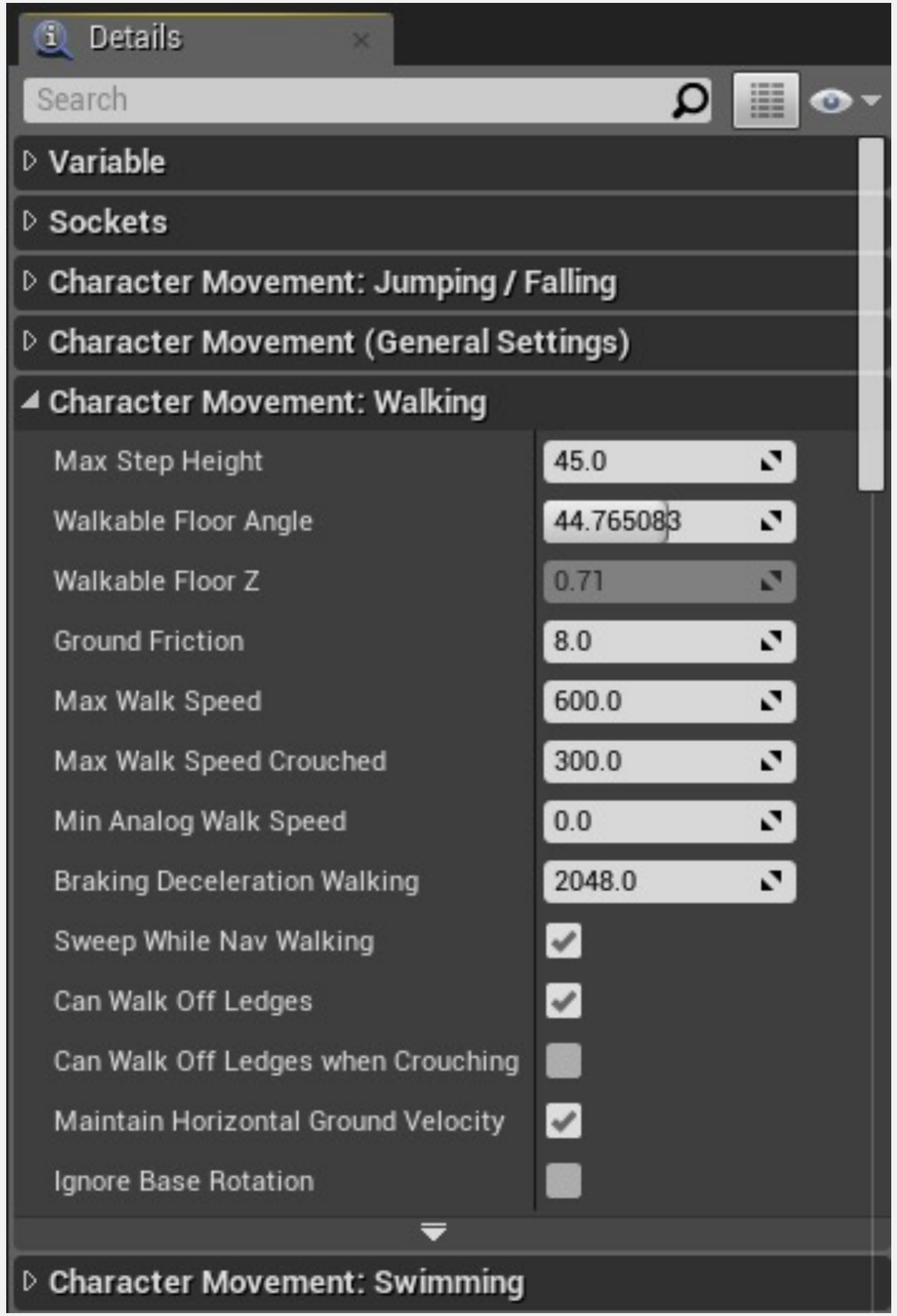
- CapsuleComponent用于碰撞测试。
- ArrowComponent表示角色的当前方向。
- Mesh组件是用于视觉呈现角色的骨架网格体。Mesh组件的动画由动画蓝图来控制。
- CharacterMovement组件用于定义各种类型的角色运动，如行走、奔跑、跳跃、游泳和飞翔。



角色运动（Character Movement）

CharacterMovement组件是用C++编写的，用于处理运动以及多人游戏中的复制和预测。

各种类型的运动都有很多属性可以在组件上调整。



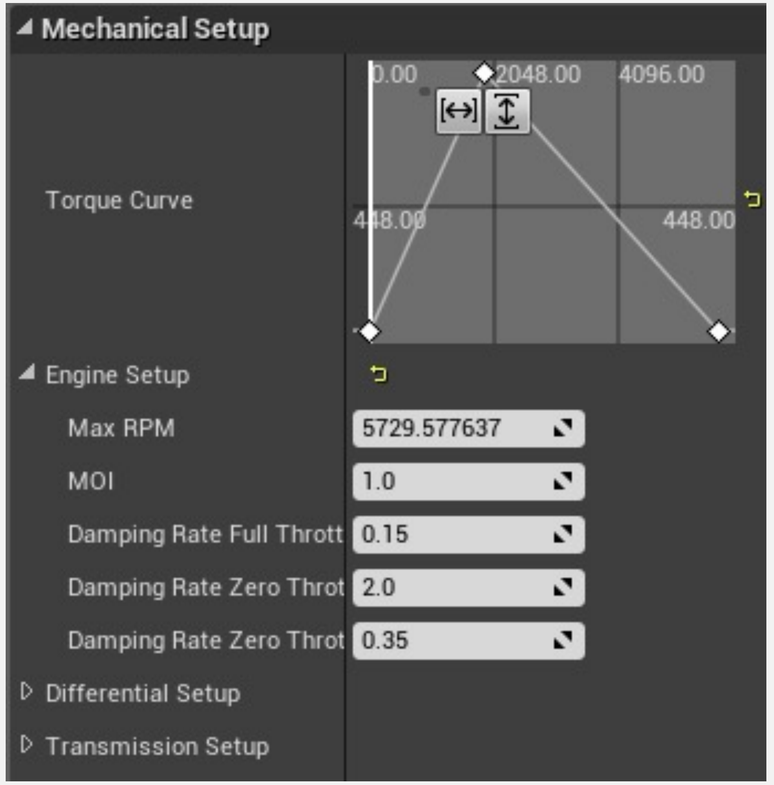
载具（Vehicle）

轮式载具（Wheeled Vehicle）类是Pawn类的另一个子类示例。

它包含用C++编写的VehicleMovement组件，并包含用于模拟载具行为的代码。它有许多参数，可用来定义载具的运动和物理。它还可以处理多人游戏中的复制和预测。

右上图来自于载具模板。

右下图显示了VehicleMovement组件的一些属性。



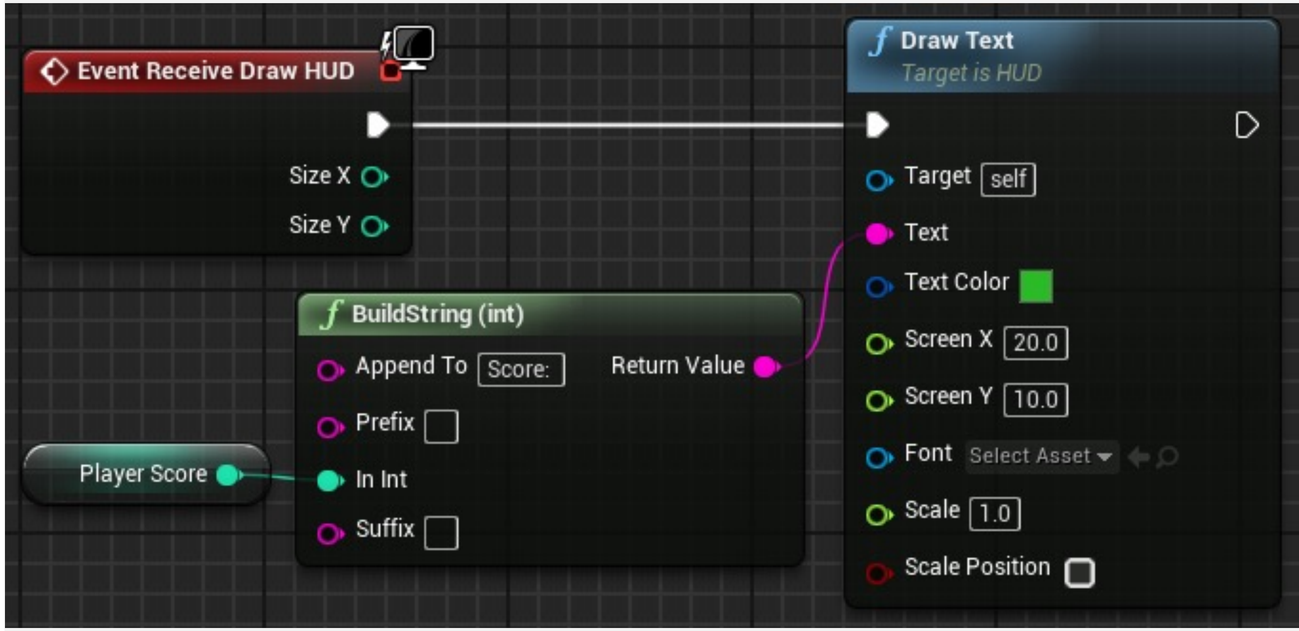
HUD

- HUD 即 heads-up display，这是一类屏幕上的信息显示，用于快速访问重要信息。
- HUD在游戏中用于向玩家显示各种信息，如分数、时间、能量等。

在虚幻引擎中，HUD类是包含画布的基类，画布是可以绘制文本和纹理等Primitives的对象。

HUD类包含名为“接收绘制HUD”（Receive Draw HUD）的事件，该事件用于在每一帧绘制Primitives。

HUD类仅存在于每个客户端上，不进行网络同步。

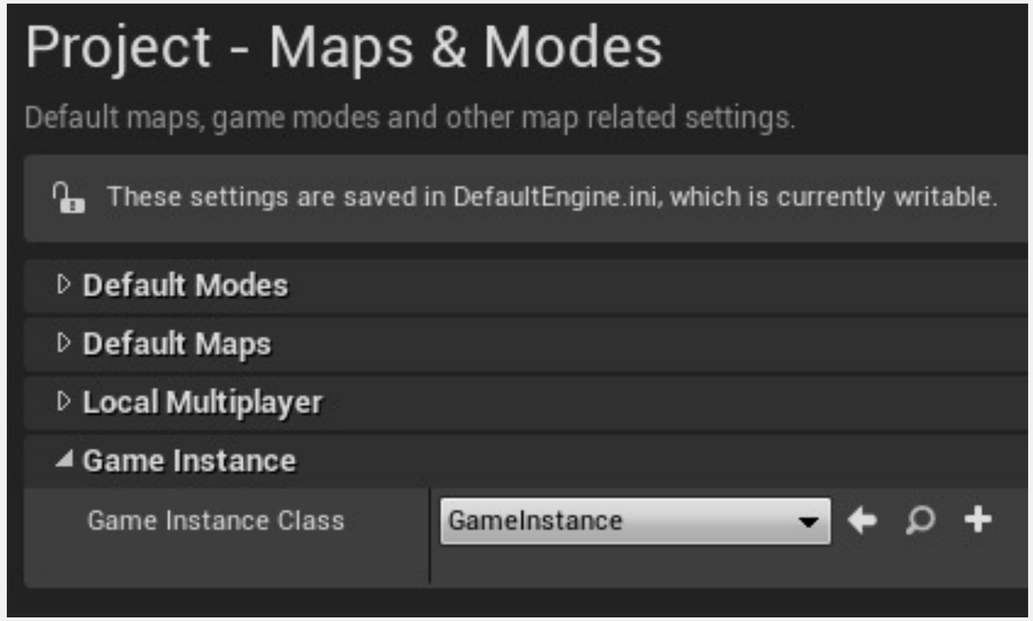


游戏实例（Game Instance）

- 游戏实例类的实例会在游戏开始时创建，并仅在游戏关闭时移除。
- 关卡中的所有Actor和其他对象会完全销毁，并在每次关卡加载时重新产生。
- 游戏实例类和它包含的数据在各个关卡之间保持不变。

游戏实例类仅存在于每个客户端上，不进行复制。

要分配游戏中使用的游戏实例类，前往“编辑”（Edit）>“项目设置”（Project Settings）>“地图和模式”（Maps & Modes）修改项目设置。



ACTOR 组件

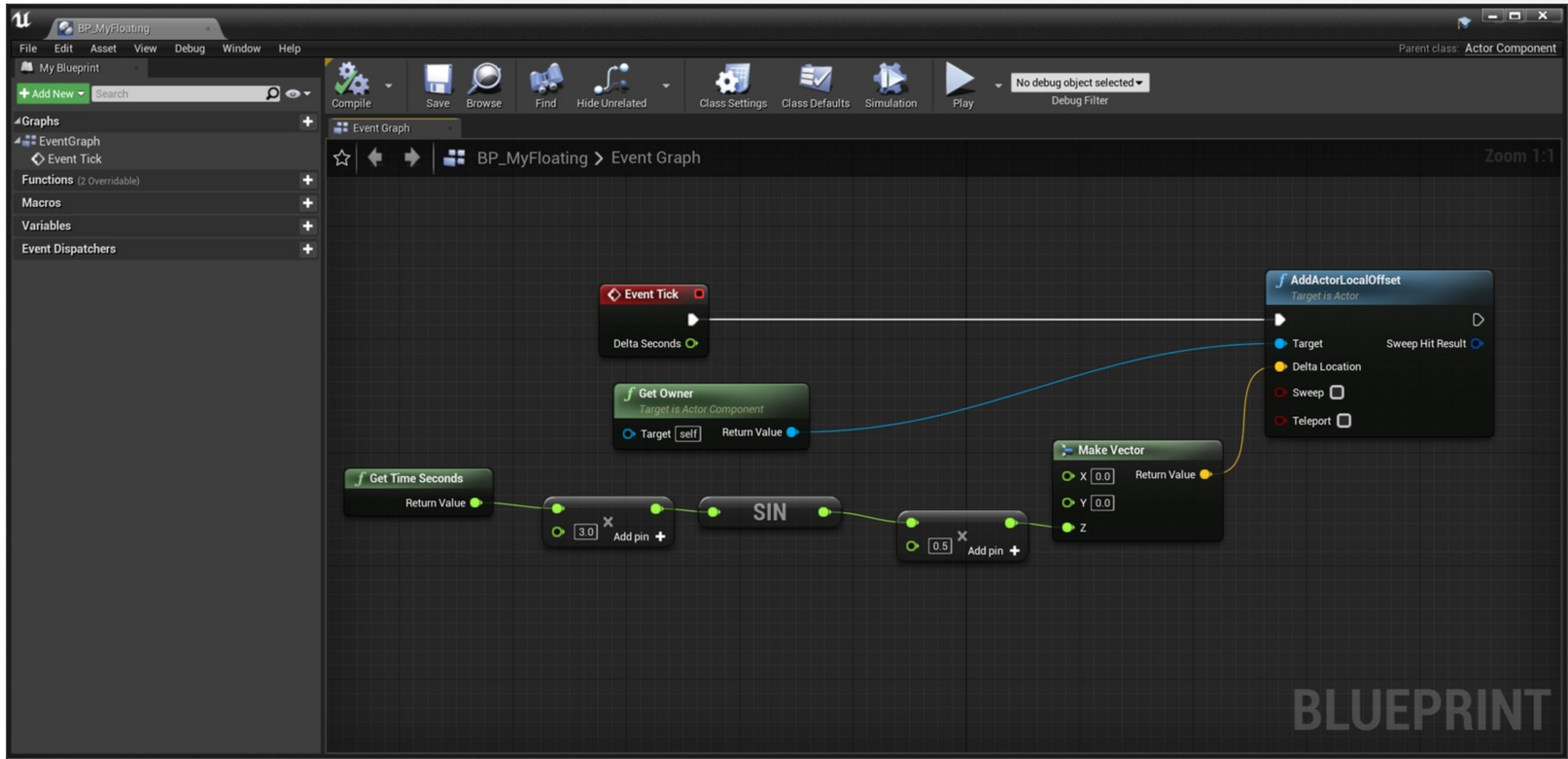
Actor Component

ACTOR 组件

- Static Mesh
- Skeletal Mesh
- 各种灯光
- 碰撞检测
- ……

蓝图 ACTOR 组件

- 选择 Actor Component 为父类
- 也可以选择其他组件为父类
- 如果对性能有一定要求，建议使用 C++ 开发
- DEMO
  - 参考 Rotating Movement 组件
  - 开发一个 Floating Movement 组件
- 如果是C++，可从 Movement Component 派生



蓝图通信

Blueprint Communication

什么是蓝图通信？

回顾上周

- 蓝图作为一种“脚本语言”
- 集中在单个蓝图内部去实践

本节内容

- 可能需要多个蓝图共同完成一个功能，如何实现？
- 例如：按下灯的开关时，灯可以打开

蓝图通信机制用来

- 向特定的对象发消息、查询状态
- 在特定事件时，对关心这个事件的所有对象进行广播

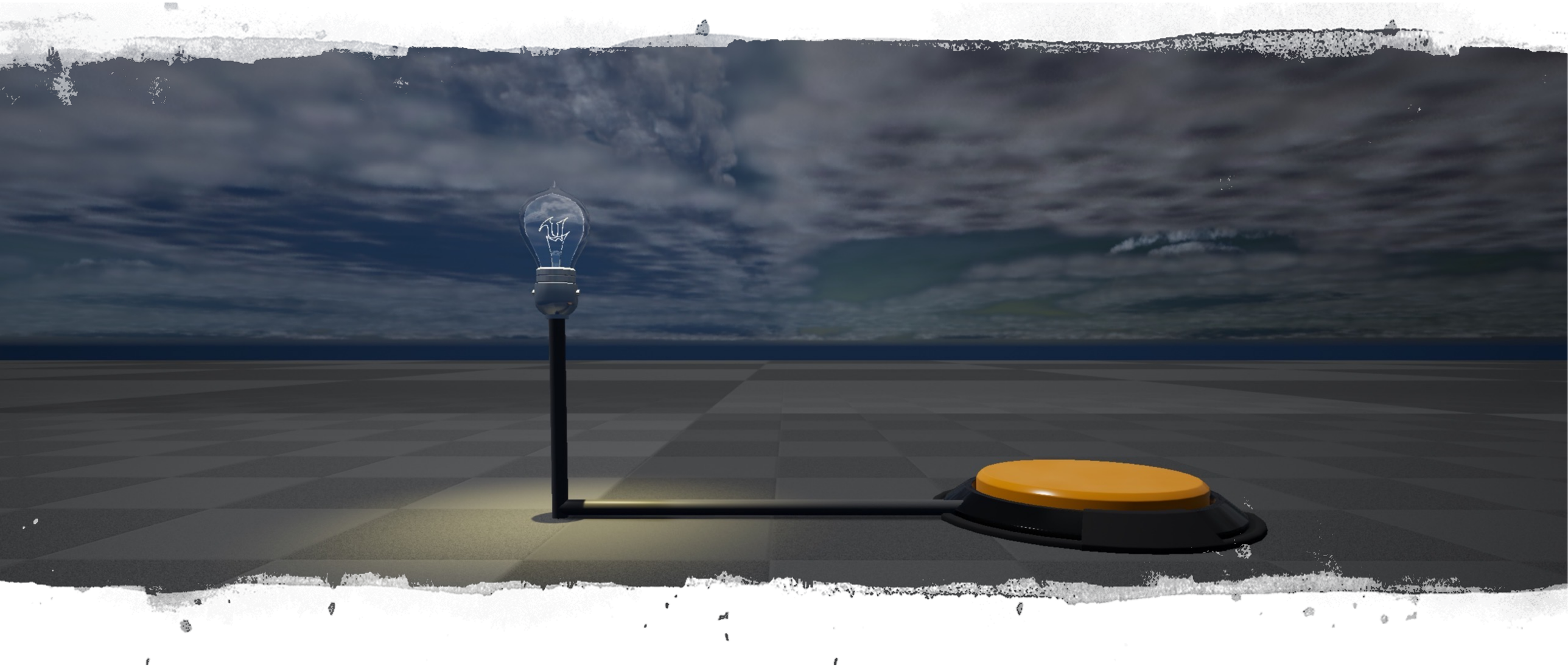
蓝图通信的类型

三种基本类型

- 直接通信（Direct Communication）
- 蓝图接口调用（Blueprint Interface message calls）
- 事件调度器（Event Dispatchers）

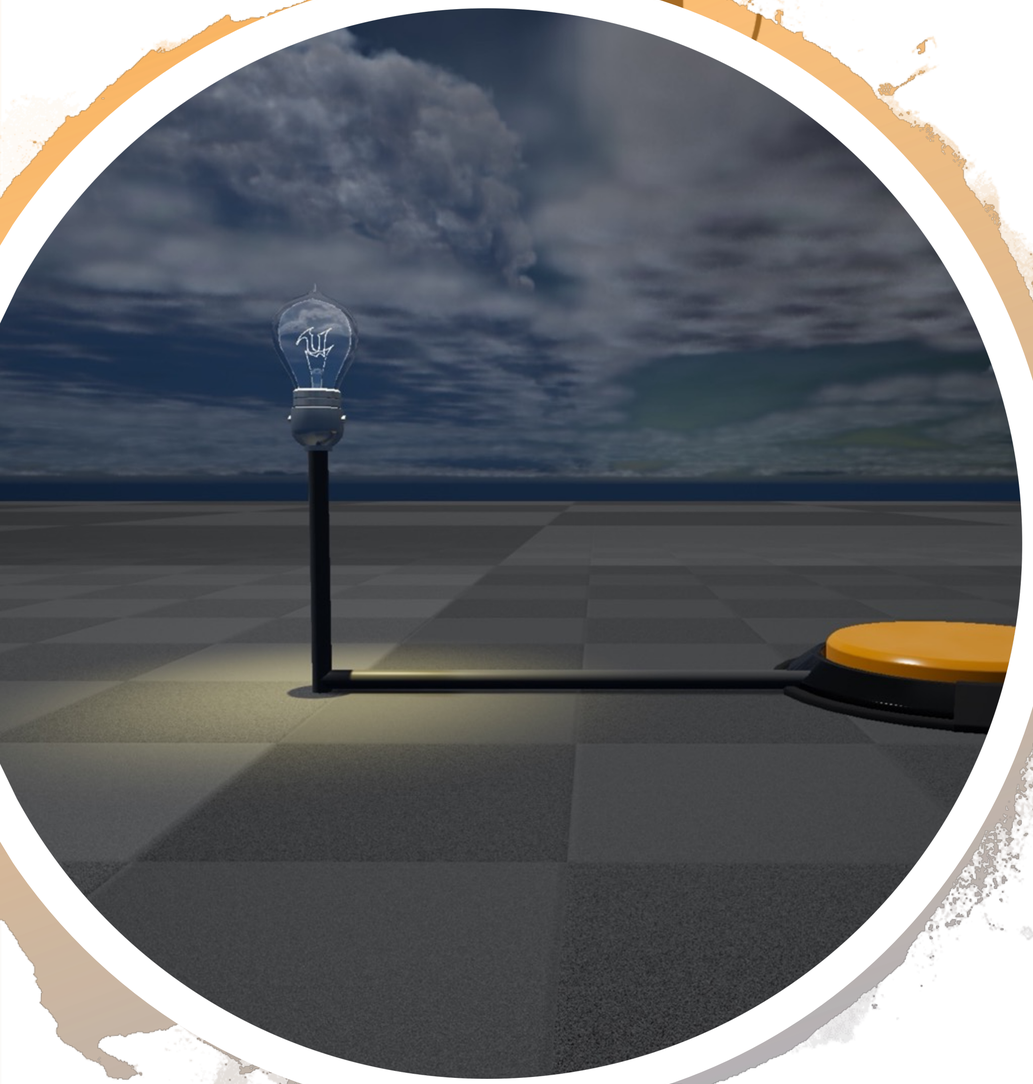
蓝图通信的参与者

- 消息的发送者（Sender）
- 消息的接受者（Receiver）



DEMO

开关+灯



DEMO

- **按下开关时，打开/关闭灯：**
  - 两个蓝图：灯、开关

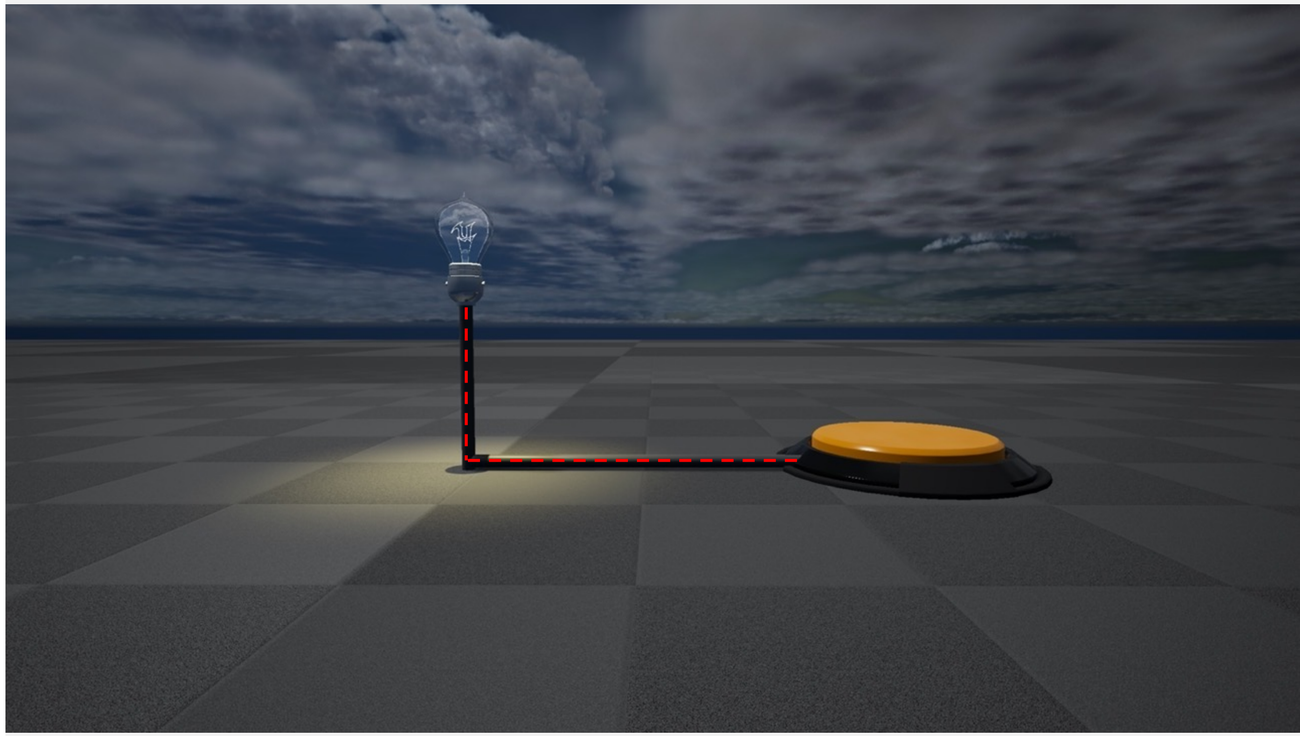
- **知识点：**
  - 直接蓝图通信
  - 对象引用
  - 自定义事件

直接蓝图通信

Direct Blueprint Communication

直接蓝图通信

- 直接蓝图通信是一种简单的蓝图通信方法
- 发送者需要知道接受者是谁

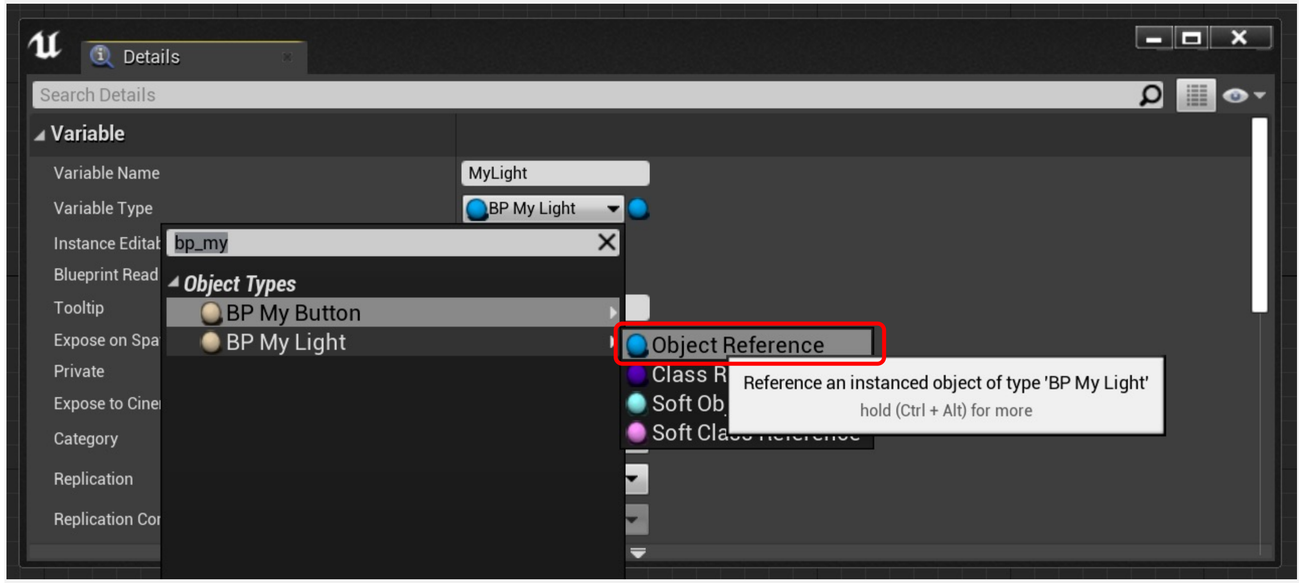


直接蓝图通信的目标

如何建立两个对象直接的链接?

引用 Actor

- Object引用（Object Reference）变量类型



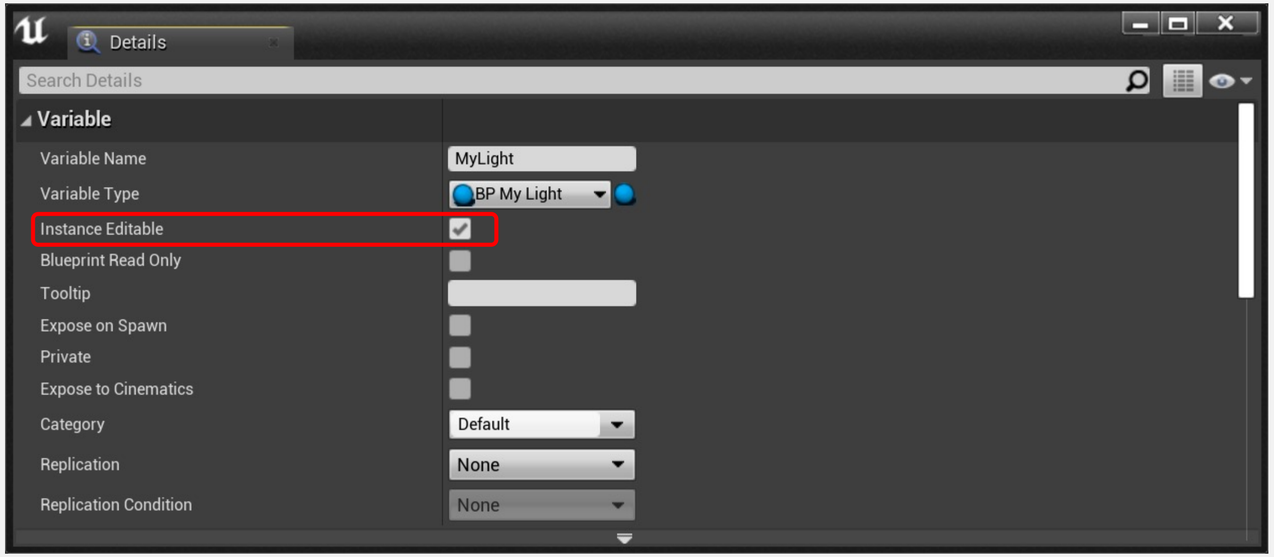
直接蓝图通信的目标

使用变量建立引用

- 在发送者蓝图中创建一个变量，用来指定接受者
- 指定这个变量为“实例可编辑”（Instance Editable）则可在关卡编辑时指定

举个例子：

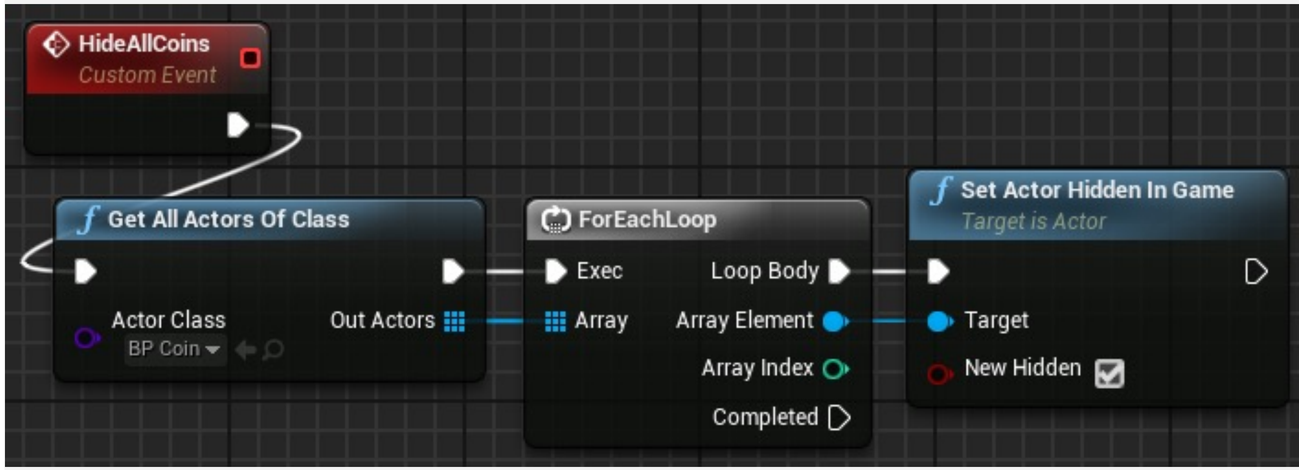
- 在我们这个DEMO中，BP_MyButton 使用一个“MyLight”变量来指定其控制的灯



直接蓝图通信的目标

获取指定 class 的所有 Actor

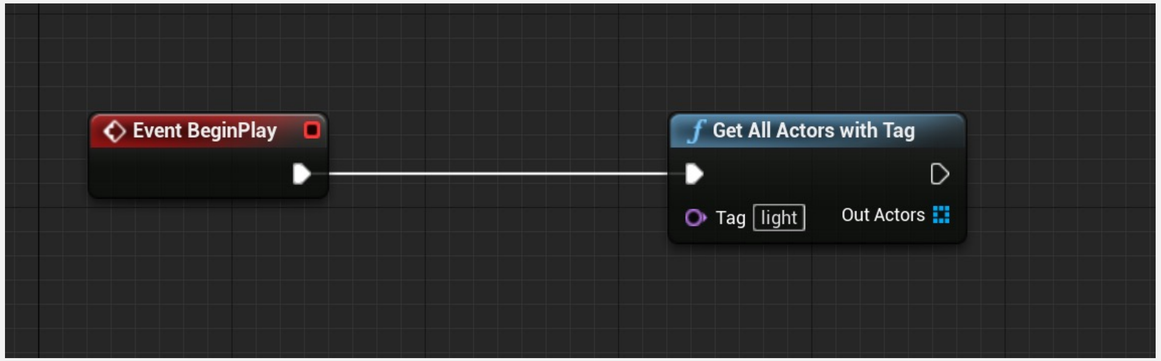
- 获取指定 class 的所有Actor（Get All Actors Of Class）函数获取当前关卡中属于指定类的所有Actor的引用
- Actor类（Actor Class）参数指定将在搜索中使用的类
- 输出Actor（Out Actor）参数是一个数组，其中包含对关卡中指定类的Actor实例的引用。
- 这个运算可能十分占用资源，建议在 BeginPlay 的时候找到想要的 Actor 存储起来

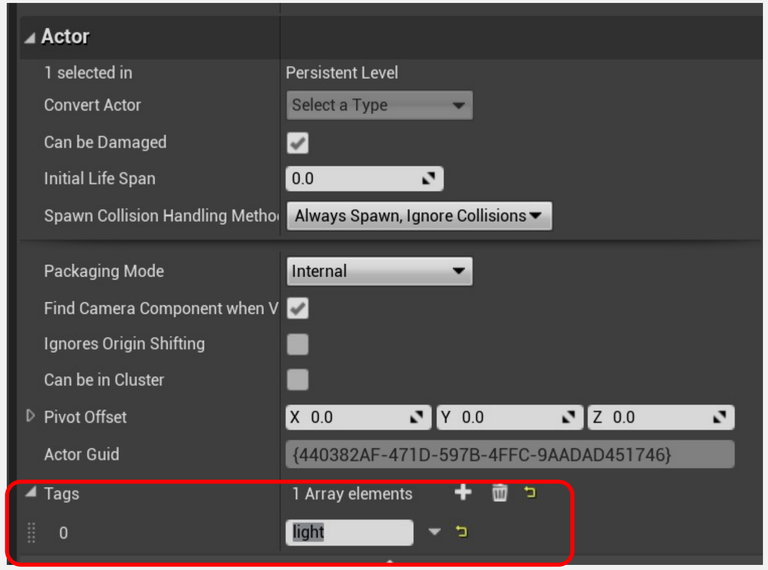


直接蓝图通信的目标

获取指定 Tag 的所有 Actor

- 类似的节点：Get All Actors with Tag
- 需要在关卡编辑中指定 Actor 的 Tag





类型转换

Casting

蓝图中的类型转换

“类型转换为”（Cast To）节点将引用变量类型转换为新指定的类型。在某些情况下，必须执行该操作才能访问类或蓝图的变量和函数。

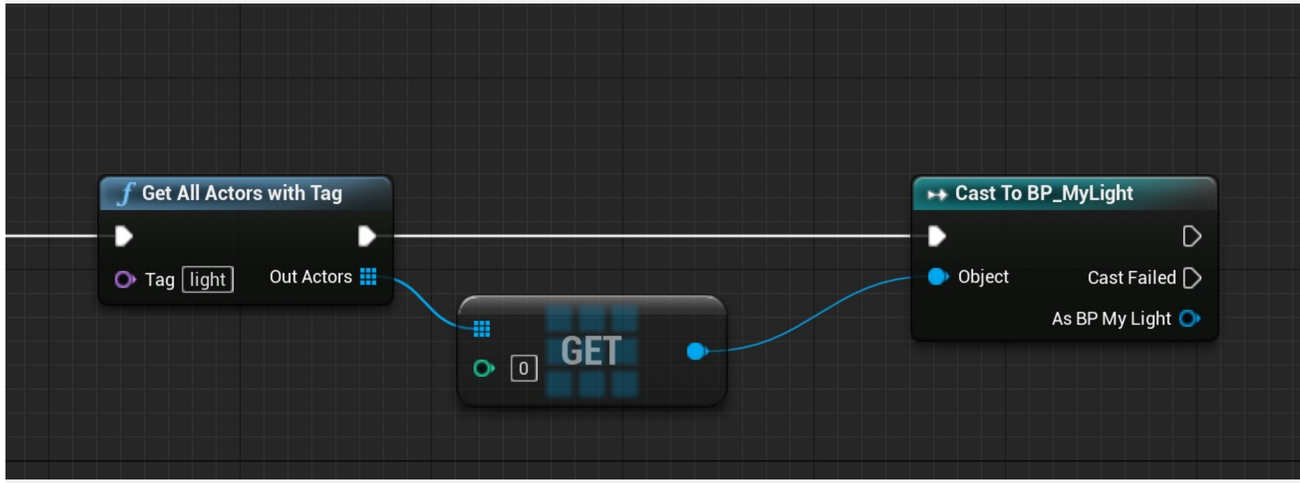
输入

•对象（Object）：接收对象引用。

输出

•类型转换失败（Cast Failed）：引用对象不是类型转换中使用的类型时，使用这个执行引脚。

•作为[新类型]（As [new type]）：使用类型转换中指定的新类型输出引用。



为什么要进行类型转换？

要从面向对象的设计讲起

- 类派生体系

Gameplay Framework 中的常用类

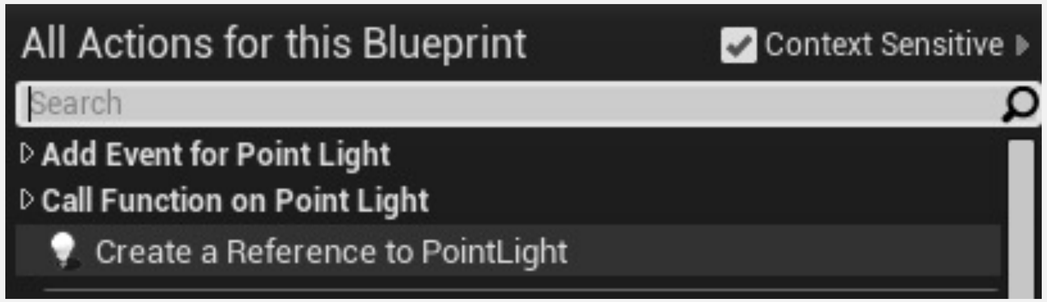
- Actor
- Pawn
- Character

在关卡蓝图中使用直接通信

向关卡蓝图添加对关卡Actor的引用很简单，这样关卡蓝图就可以直接与关卡Actor通信。

要添加引用，请完成以下步骤：

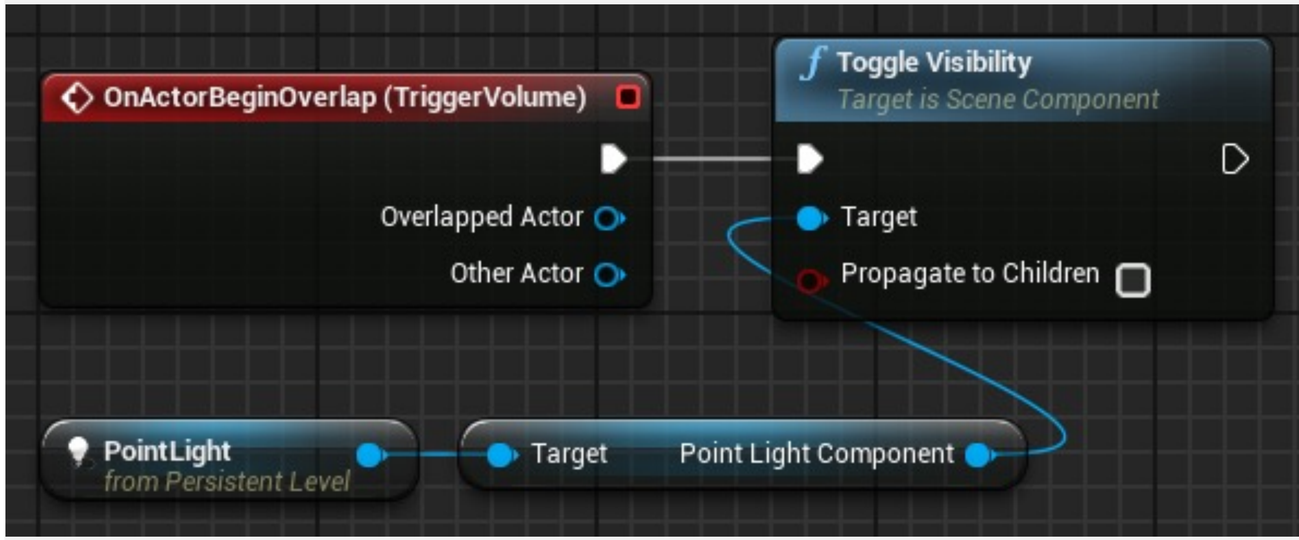
- 首先在关卡编辑器中选择Actor。
- 打开关卡蓝图。
- 在事件图表中单击右键，并选择选项“创建对[Actor名称]的引用”（Create a Reference to [Actor name]）。



在关卡蓝图中使用直接通信

举个例子

- 在右图中，已经在关卡蓝图中放置了一个触发体积的重叠事件。
- 当发生重叠时，将在点光源Actor的点光源组件上调用切换可视性（Toggle Visibility）函数。



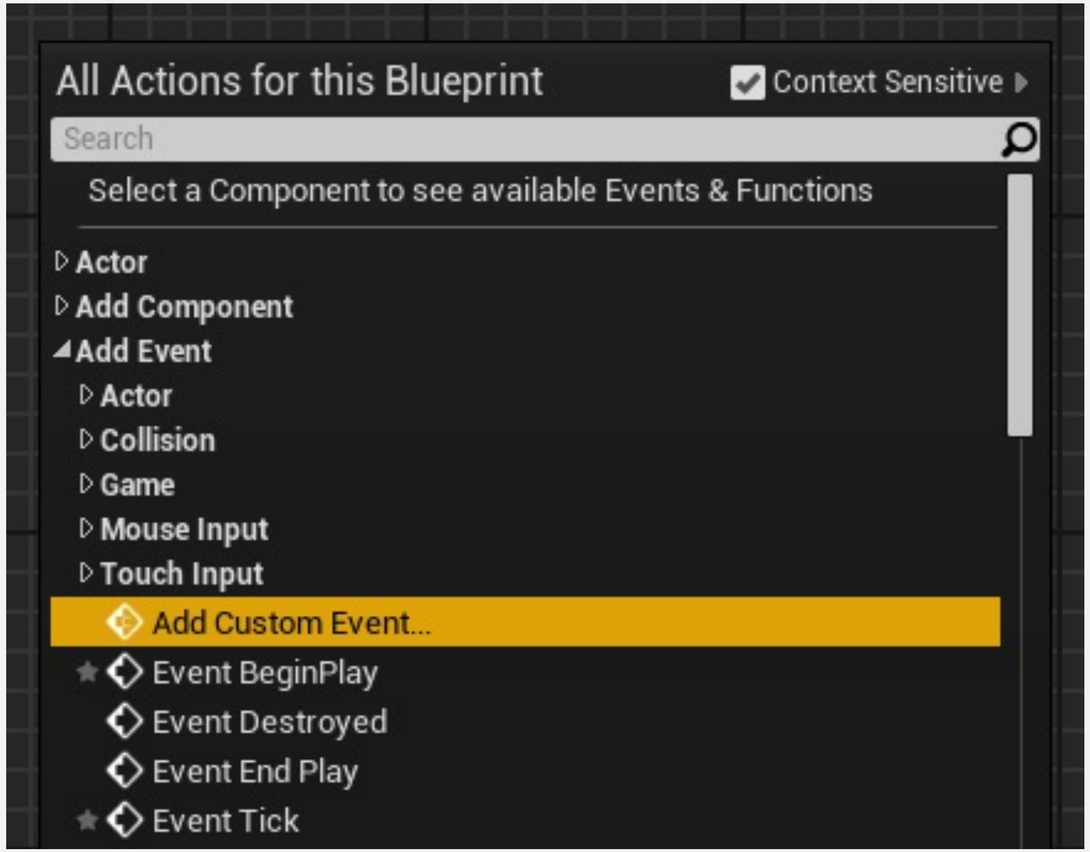
自定义事件

Custom Events

自定义事件

虚幻引擎提供了多个预定义事件，但可以创建新事件以在蓝图中使用。这些事件可以从在其中定义这些事件的蓝图中调用，也可以从其他蓝图中调用。

要创建自定义事件，在事件图表中单击右键，展开“添加事件”（Add Event）类别，然后选择“添加自定义事件...”（Add Custom Event...）。

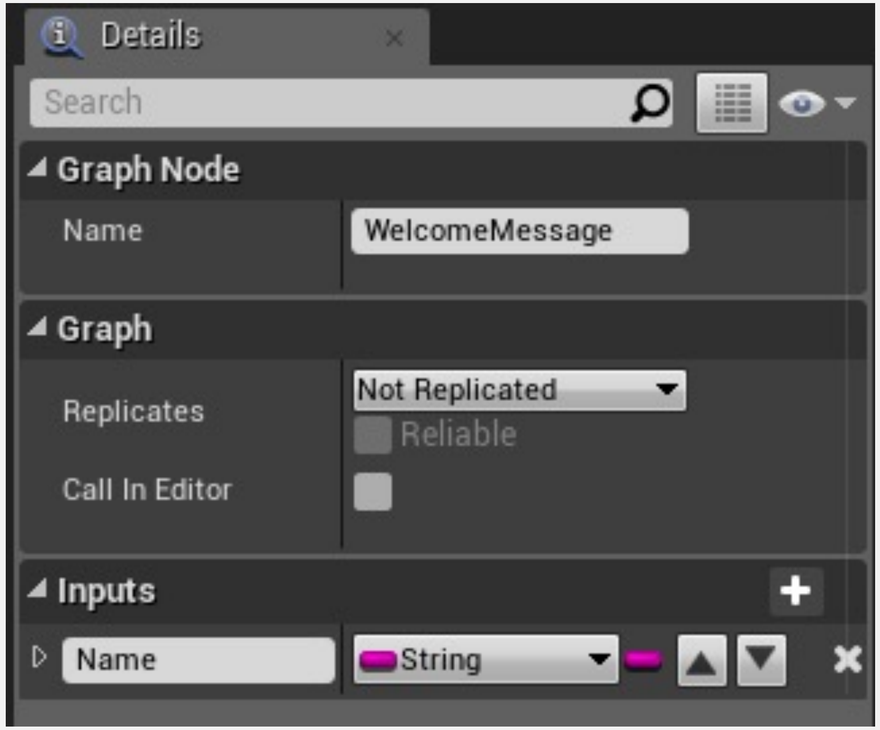


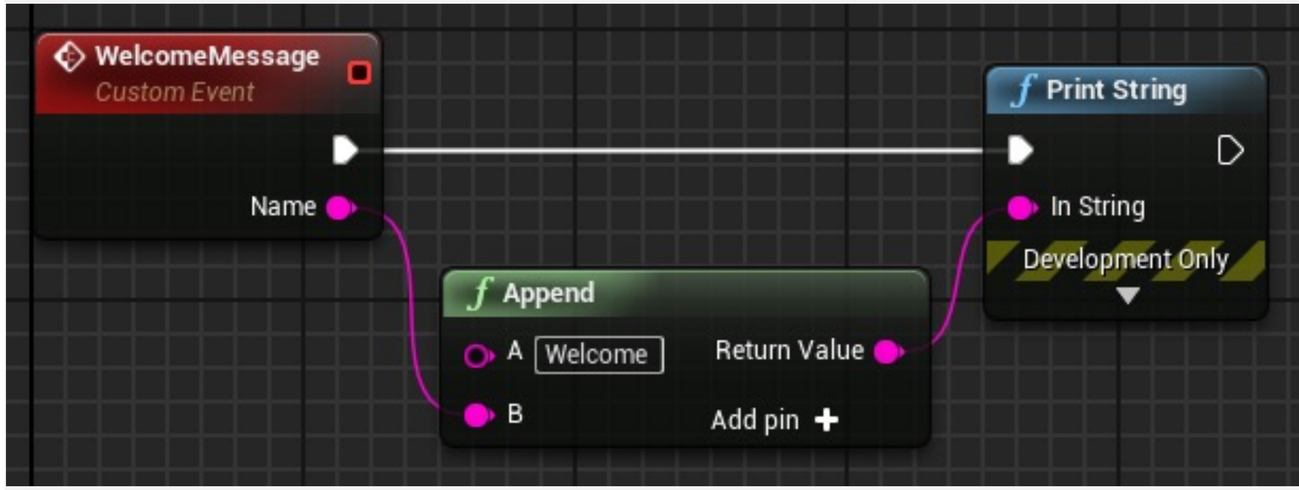
自定义事件：输入参数

通过选择自定义事件（Custom Event）节点，您可以管理事件名称和输入参数。事件没有输出参数。

右图所示自定义事件名为“欢迎消息”（WelcomeMessage），其输入参数为“名称”（Name）。

这个事件将使用作为参数传递的名称，创建并在屏幕上输出自定义消息。

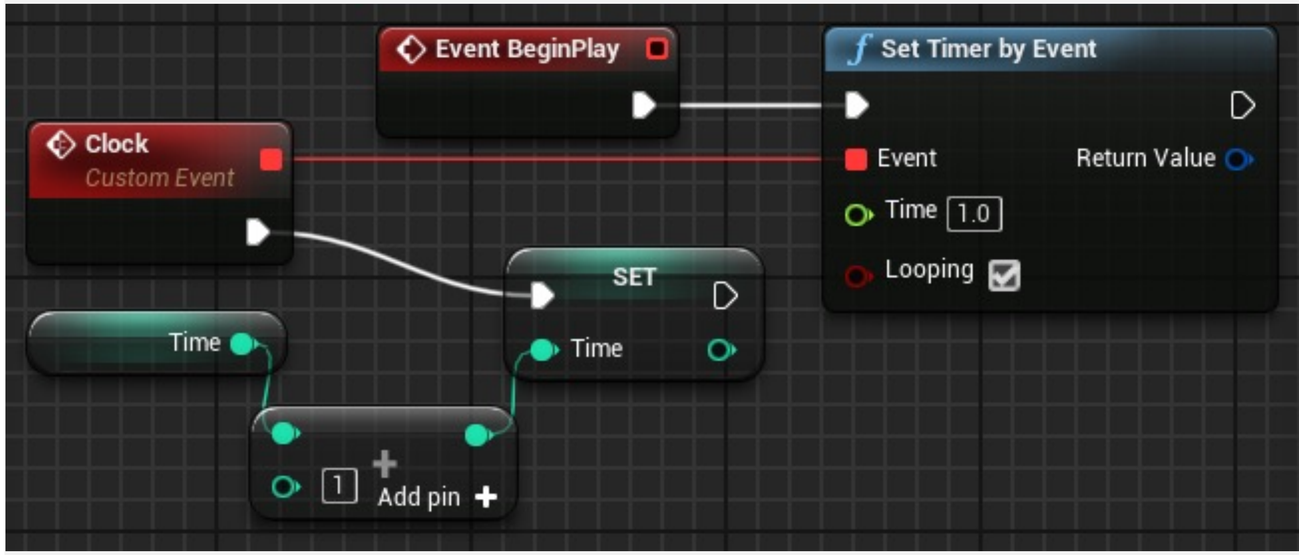




自定义事件：委托

右上角有一个红色小正方形的事件称为委托。这只是对事件的引用。某些操作接收事件作为参数，并使用委托来实现事件。

在右图中，名为“时钟”（Clock）的自定义事件委托连线到“按事件设置定时器”（Set Timer by Event）节点的“事件”（Event）输入引脚，这样每一秒都会调用“时钟”（Clock）事件。



DEMO

对关卡中的多种对象进行交互

DEMO

- 通过按键，对玩家面前的对象进行操作
  - 对象可能是多种蓝图对象
  - 开关电灯
  - 开关电视机

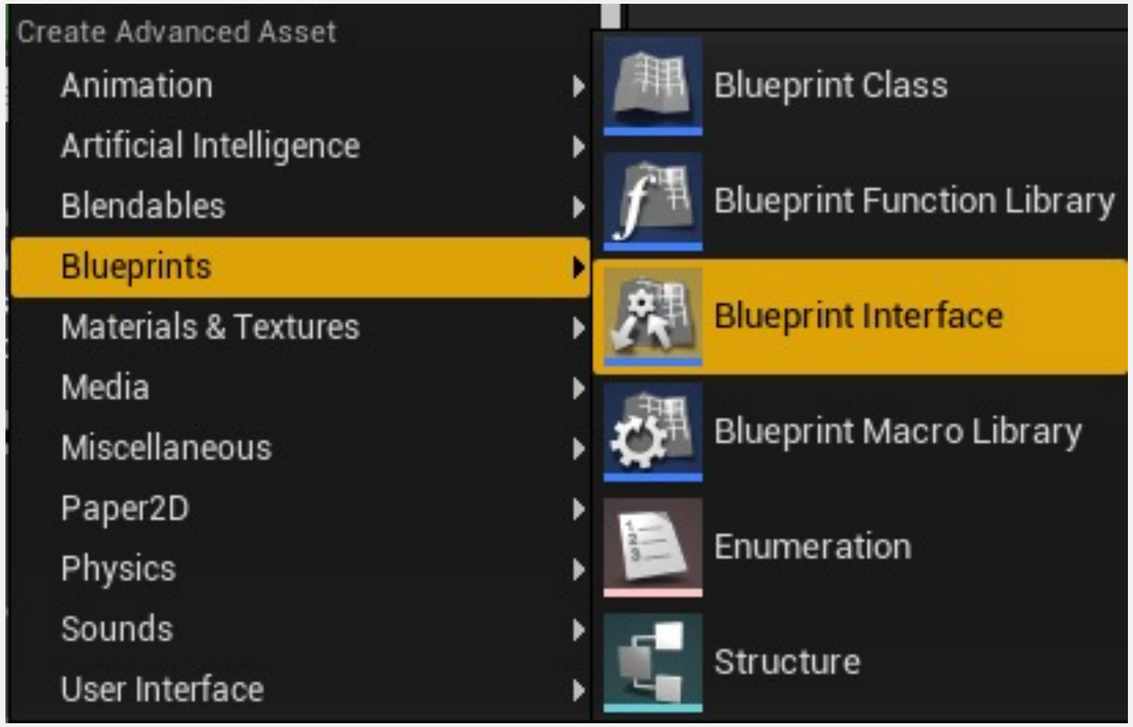
- **知识点：**
  - 蓝图接口
  - 用户输入处理

蓝图接口

蓝图接口（BPI）仅包含函数定义，不包含实现。

如果蓝图类实现BPI，则使用提供的定义，然后实现自己用于该函数的逻辑。

要创建新蓝图接口，单击内容浏览器中的绿色“新增”（Add New）按钮，然后在“蓝图”（Blueprints）子菜单中选择“蓝图接口”（Blueprint Interface）。

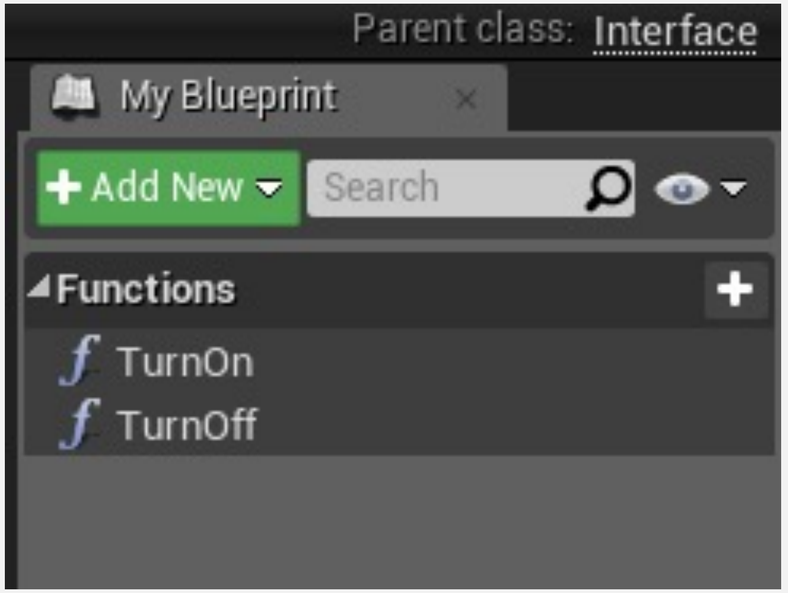


蓝图接口：
创建函数

在该示例中，创建了名为“BP_Interface_TurnOnOff”的蓝图接口。

编辑BPI时，可以指定函数，并创建输入和输出参数，但不能在接口中实现逻辑。

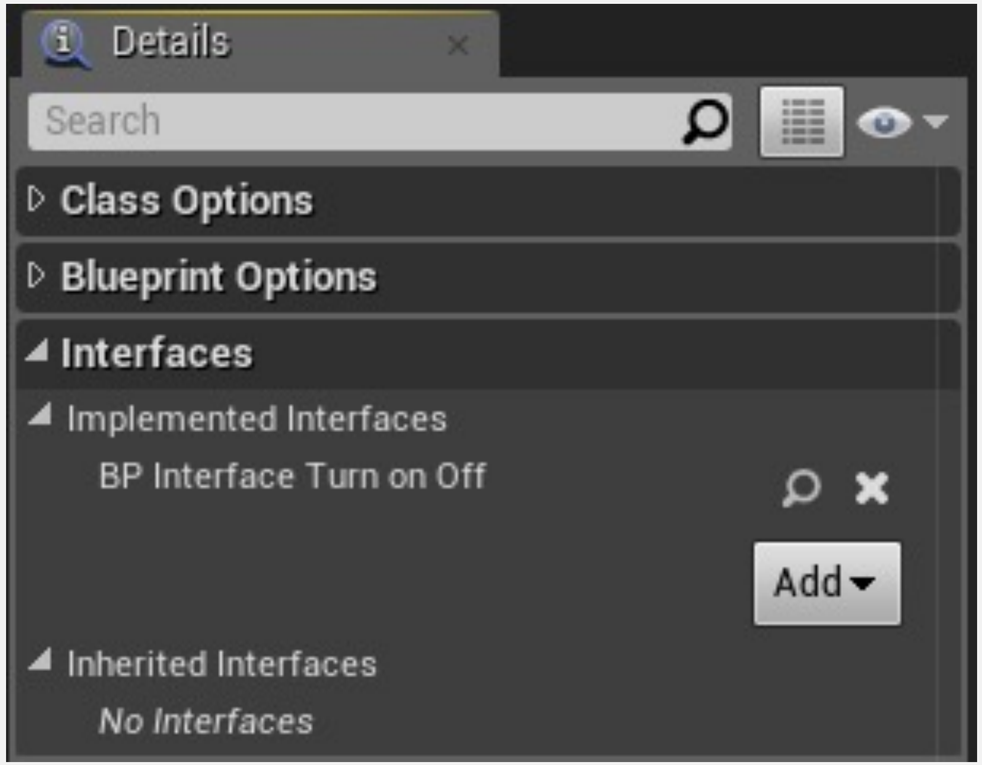
该BPI有两个函数：TurnOn和TurnOff，如右图所示。



蓝图接口：
添加到蓝图

要将蓝图接口添加到蓝图类，单击蓝图编辑器工具栏上的“类设置”（Class Settings）按钮。在“细节”（Details）面板的“接口”（Interfaces）部分中，单击“添加”（Add）按钮，并选择蓝图接口类。

在右图中，添加了BP接口打开关闭（BP Interface Turn On Off）蓝图接口。

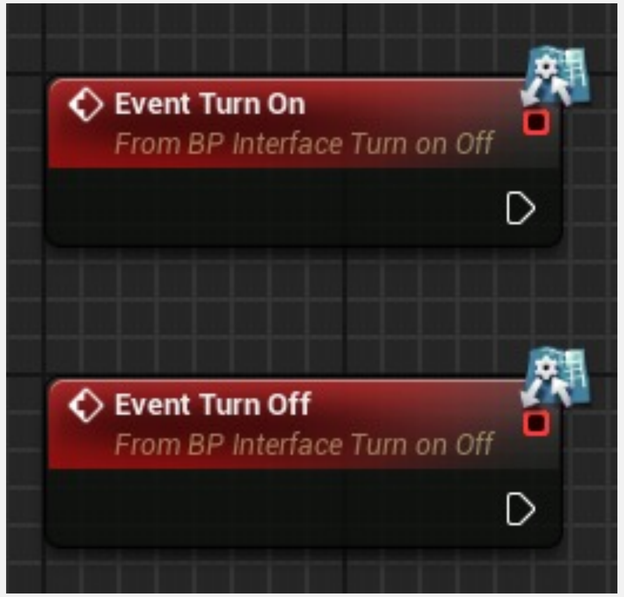


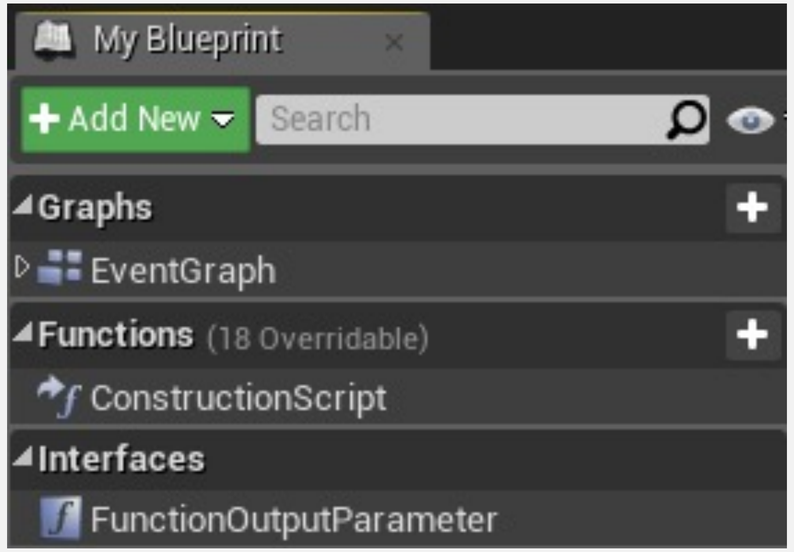
蓝图接口：实现函数

没有输出参数的蓝图接口函数在实现BPI的蓝图中显示为事件。

有输出参数的蓝图接口函数出现在“我的蓝图”（My Blueprint）面板的“接口”（Interfaces）部分中。

右下图显示了名为“函数输出参数”（FunctionOutputParameter）的函数。双击打开函数进行编辑，以实现该函数。

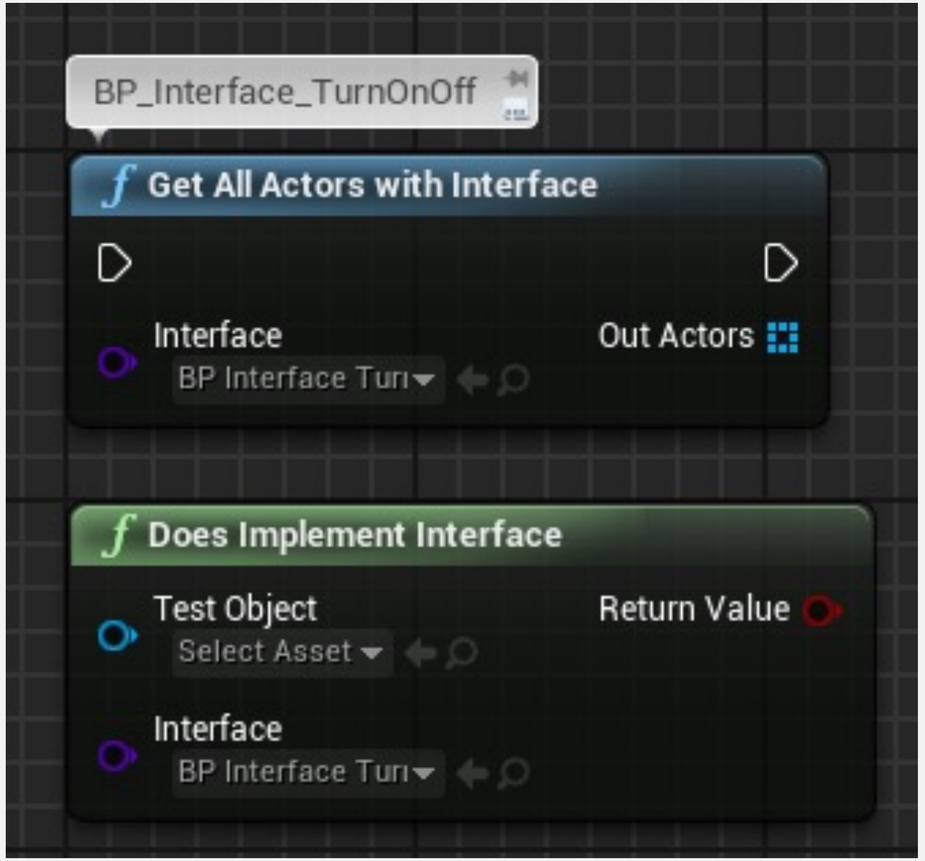




蓝图接口：工具函数

蓝图接口有一些相关的工具函数。下面列出了两个示例：

- 获取所有带有指定接口的 Actor（Get All Actors with Interface）：在当前关卡中，查找所有实现指定BPI的Actor。
- 是否实现接口（Does Implement Interface）：测试某个具体对象是否实现BPI。



玩家输入

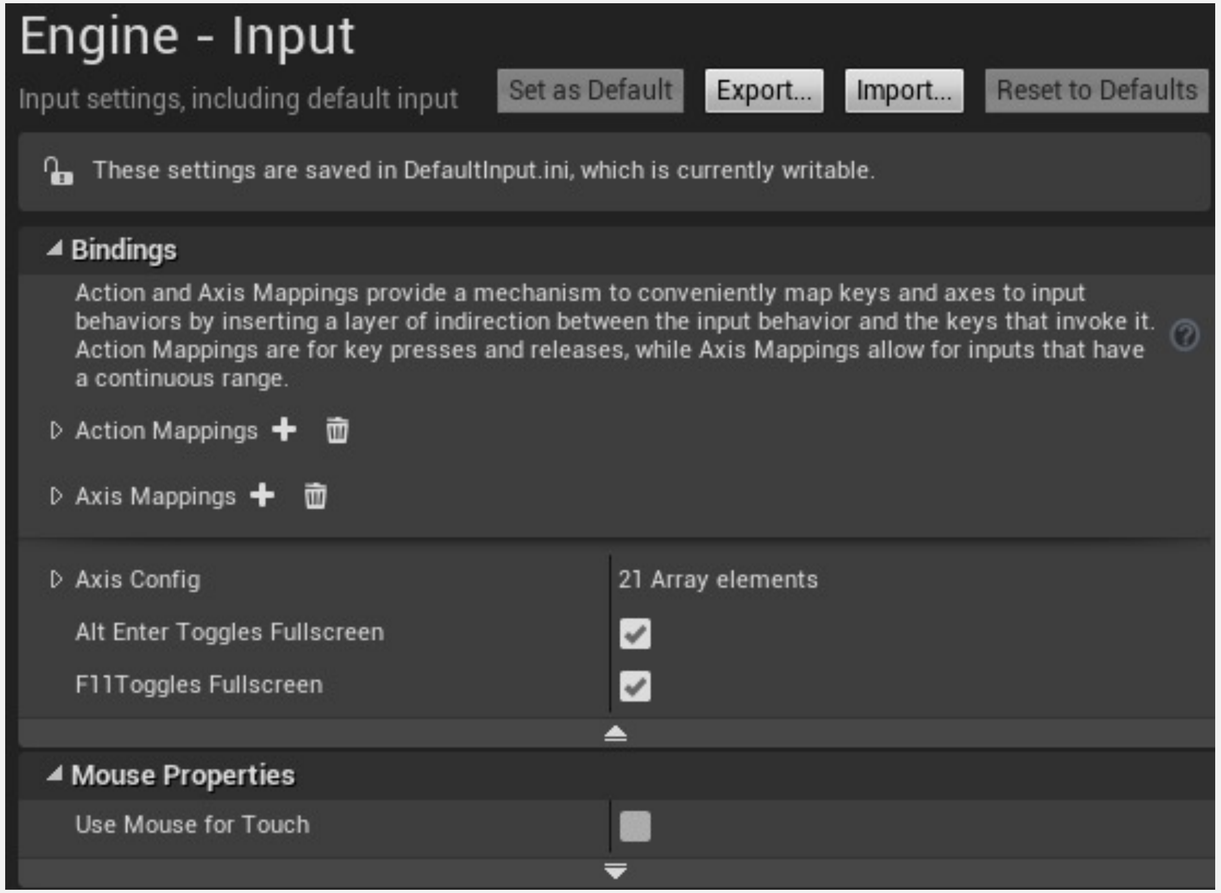
Player Input

输入映射

可以创建新输入事件来表示在游戏中有意义的操作。

例如，相较于为鼠标左键创建输入事件来触发枪支操作，更好的方法是创建操作事件“开枪”（Fire），然后映射能够触发这个事件的所有按键和按钮。

要访问输入映射，在关卡编辑器菜单中，前往“编辑”（Edit）>“项目设置...”（Project Settings...），然后在“引擎”（Engine）类别中，选择“输入”（Input）选项。

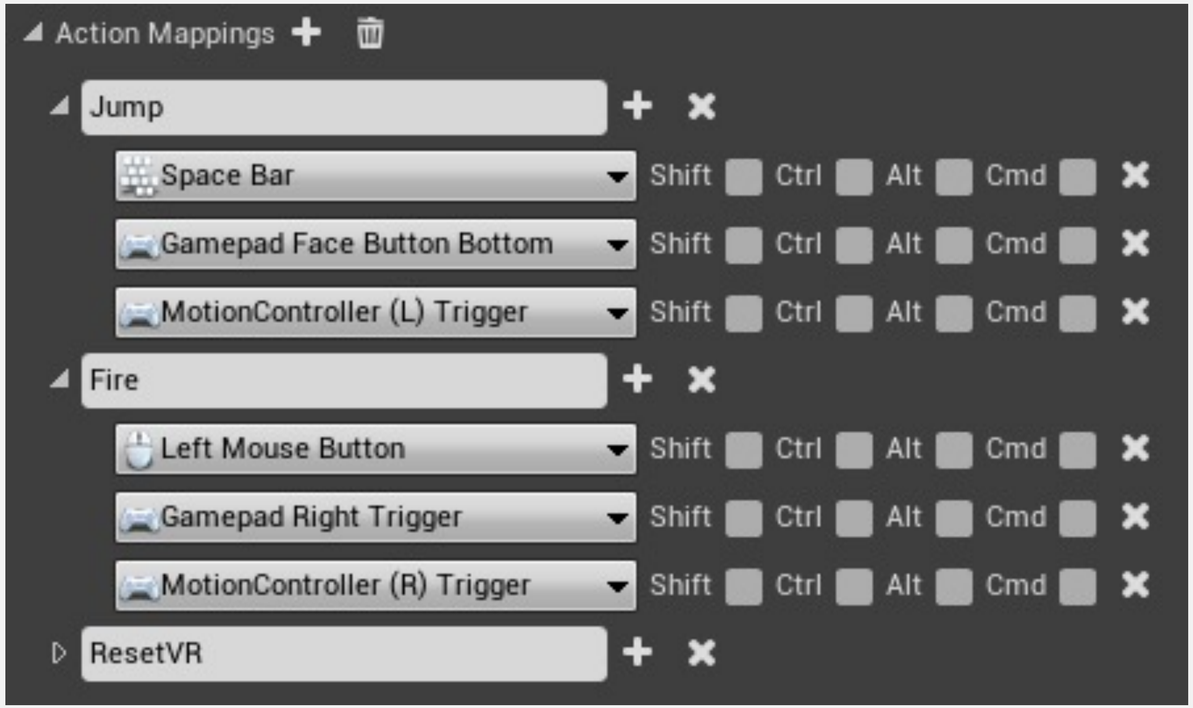


操作映射

操作映射用于按键和按钮的按下和松开操作。

右图所示示例是来自第一人称模板的操作映射。

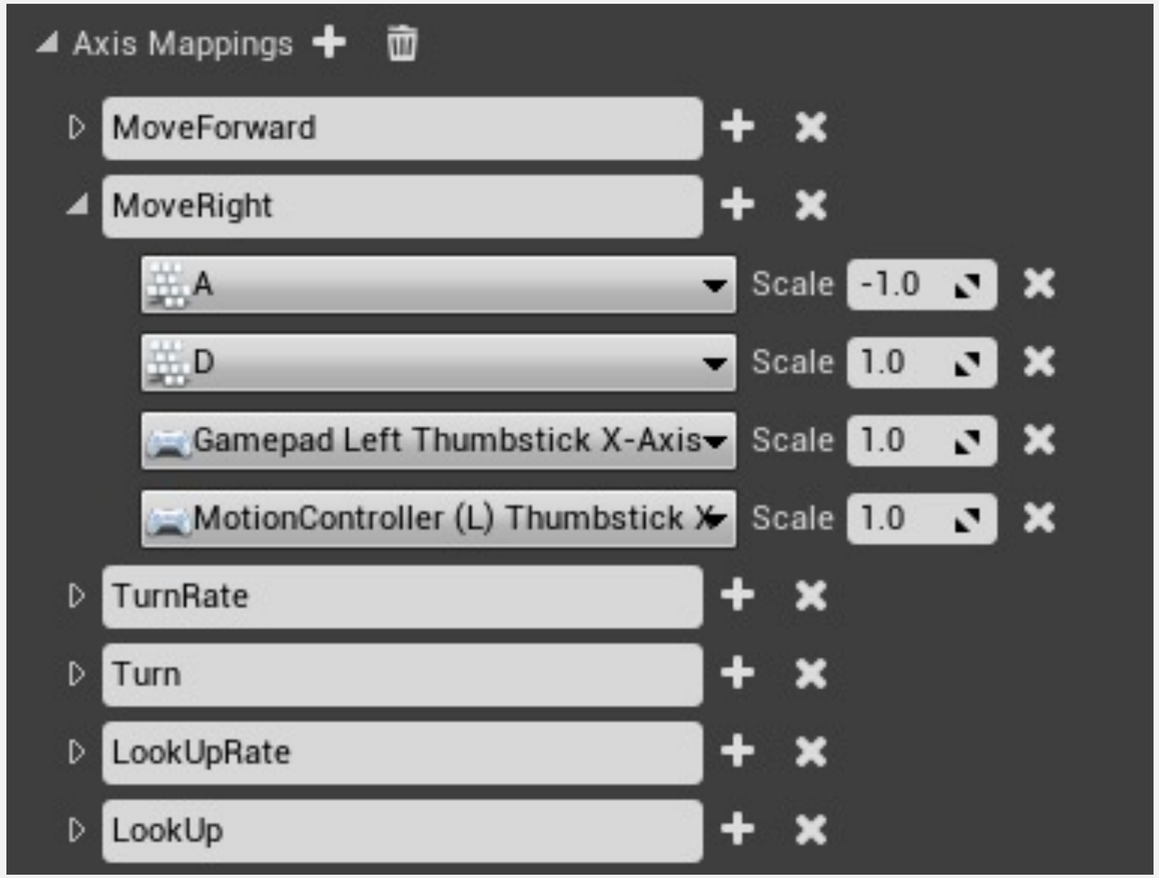
在该示例中，创建了一个名为“跳跃”（Jump）的操作，可以通过空格、游戏手柄底下的按钮或运动控制器的左扳机触发。



轴映射

轴映射允许拥有连续范围的输入，例如鼠标移动或游戏手柄的类比摇杆。

按键和按钮也可以在轴映射中使用。在右侧示例中，右移（MoveRight）操作映射到“D”键时，“缩放”（Scale）属性设置为“1.0”；映射到“A”键时，该属性值设置为“–1.0”，表示反向。

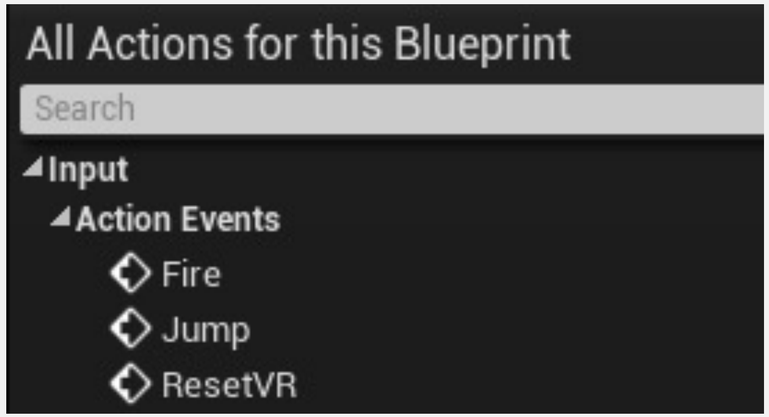


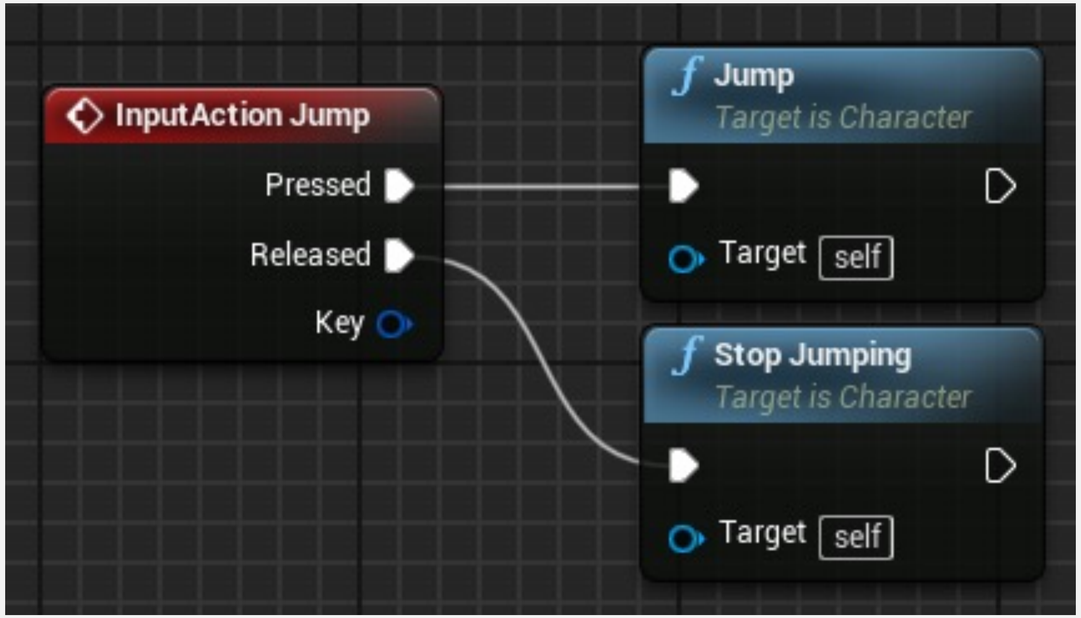
输入操作事件

所有操作映射均可在蓝图编辑器中使用，它们位于快捷菜单中的“输入”（Input）>“操作事件”（Action Events）下面。

当与输入操作（InputAction）事件关联的按键或按钮被按下或松开时，会生成这个事件。

右下图显示了输入操作（InputAction）事件的示例。



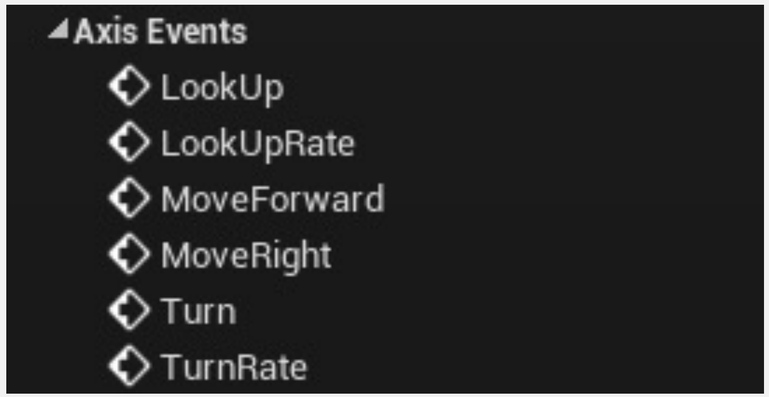


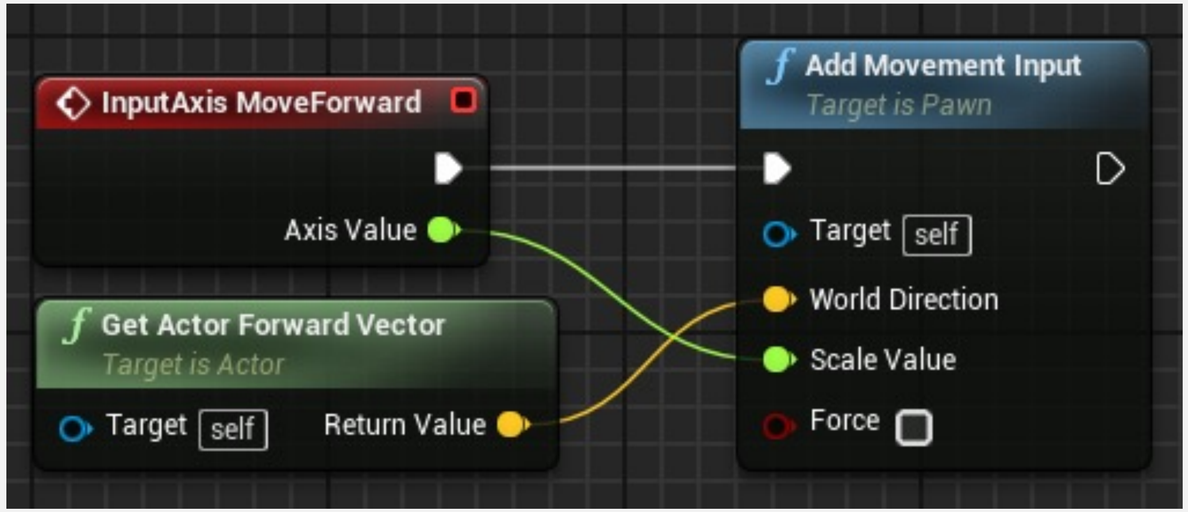
输入轴事件

所有轴映射均可在蓝图编辑器中使用，它们位于快捷菜单中的“输入”（Input）>“轴事件”（Axis Events）下面。

输入轴（InputAxis）事件持续报告轴的当前值。

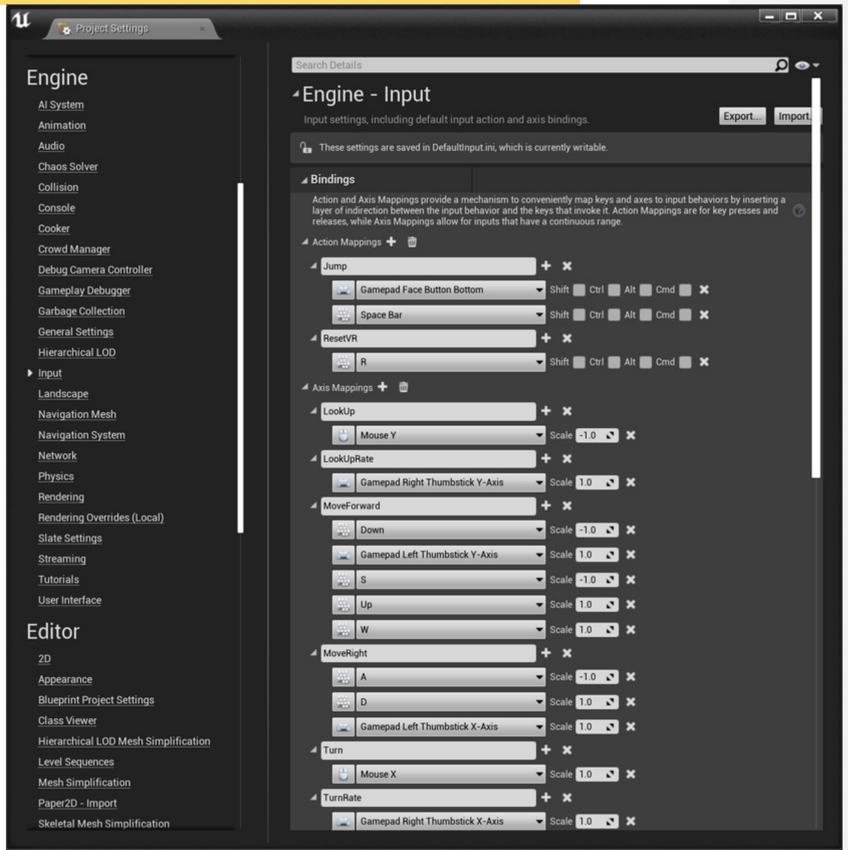
右下图显示了输入轴（InputAxis）事件的示例。

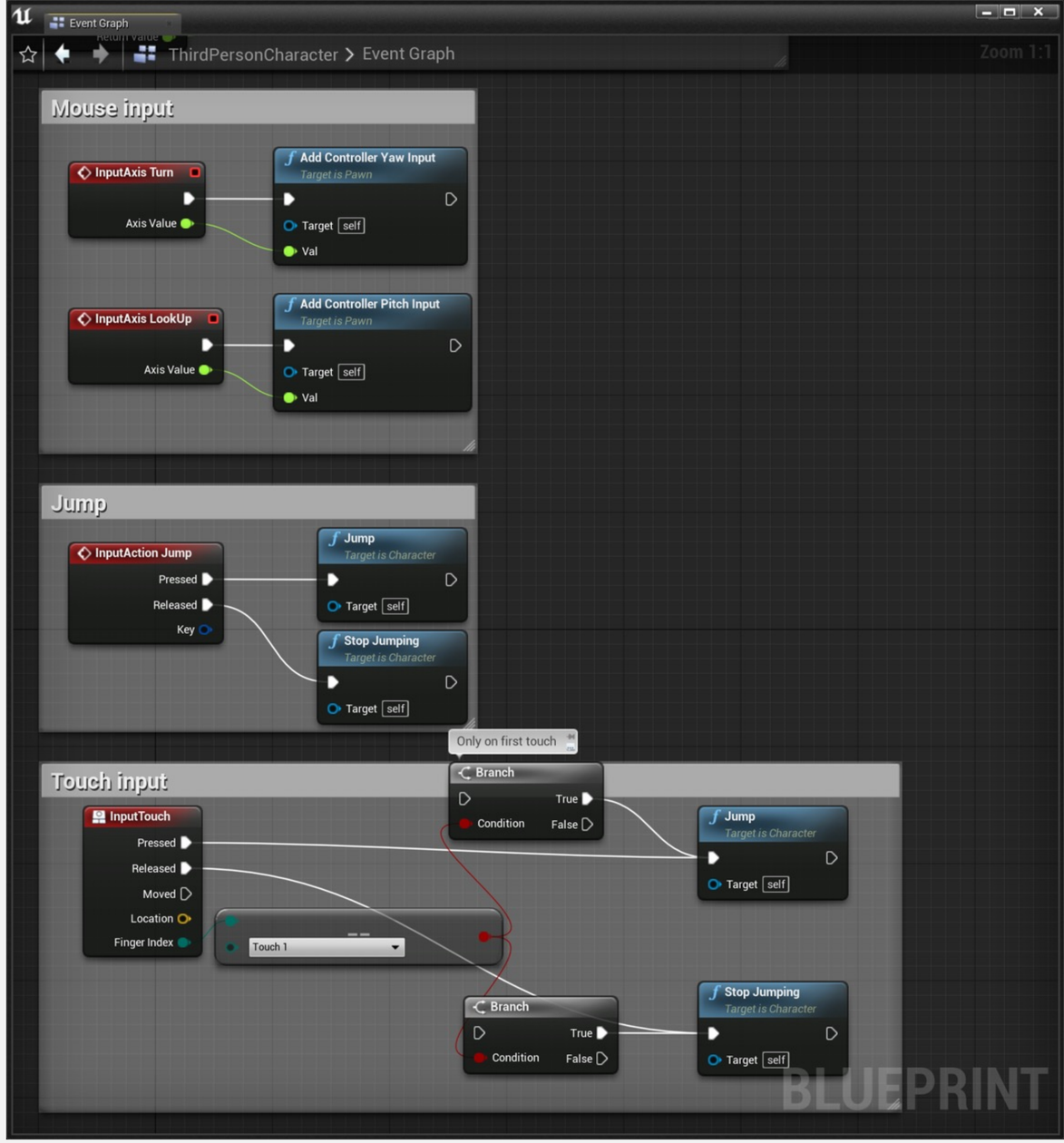




案例分析：
ThirdPersonCharacter

- Project Settings
- Event Graph





TRACES

Trace 类型

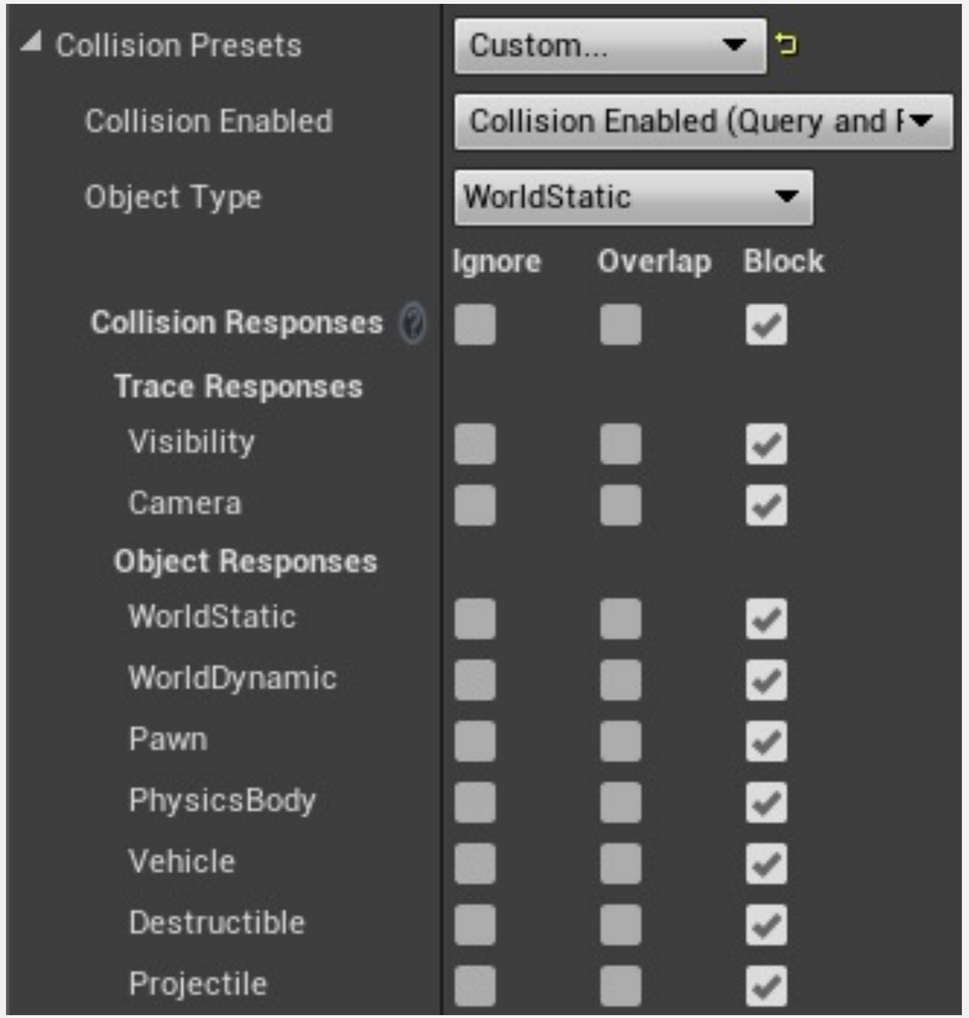
追踪用于测试定义的线条上是否有碰撞，并可以返回命中的第一个对象或多个对象。

追踪可以通过通道或Object类型实现。

通道可以是“可视性”（Visibility）或“摄像机”（Camera）。Object类型可以是“WorldStatic”、“WorldDynamic”、“Pawn”、“PhysicsBody”、“Vehicle”、“Destructible”或“Projectile”。

右图显示了静态网格体Actor的碰撞响应。

Object类型通过“Object类型”（Object Type）属性下拉列表定义，而Visibility和Camera追踪响应在“碰撞响应”（Collision Responses）表的“追踪响应”（Trace Responses）部分中定义。



命中结果结构体

当追踪函数发生碰撞时，返回一个或多个命中结果结构体。“分解命中结果”（Break Hit Result）节点可以用来访问命中结果的元素，如右图所示。

命中结果的部分元素如下所示：

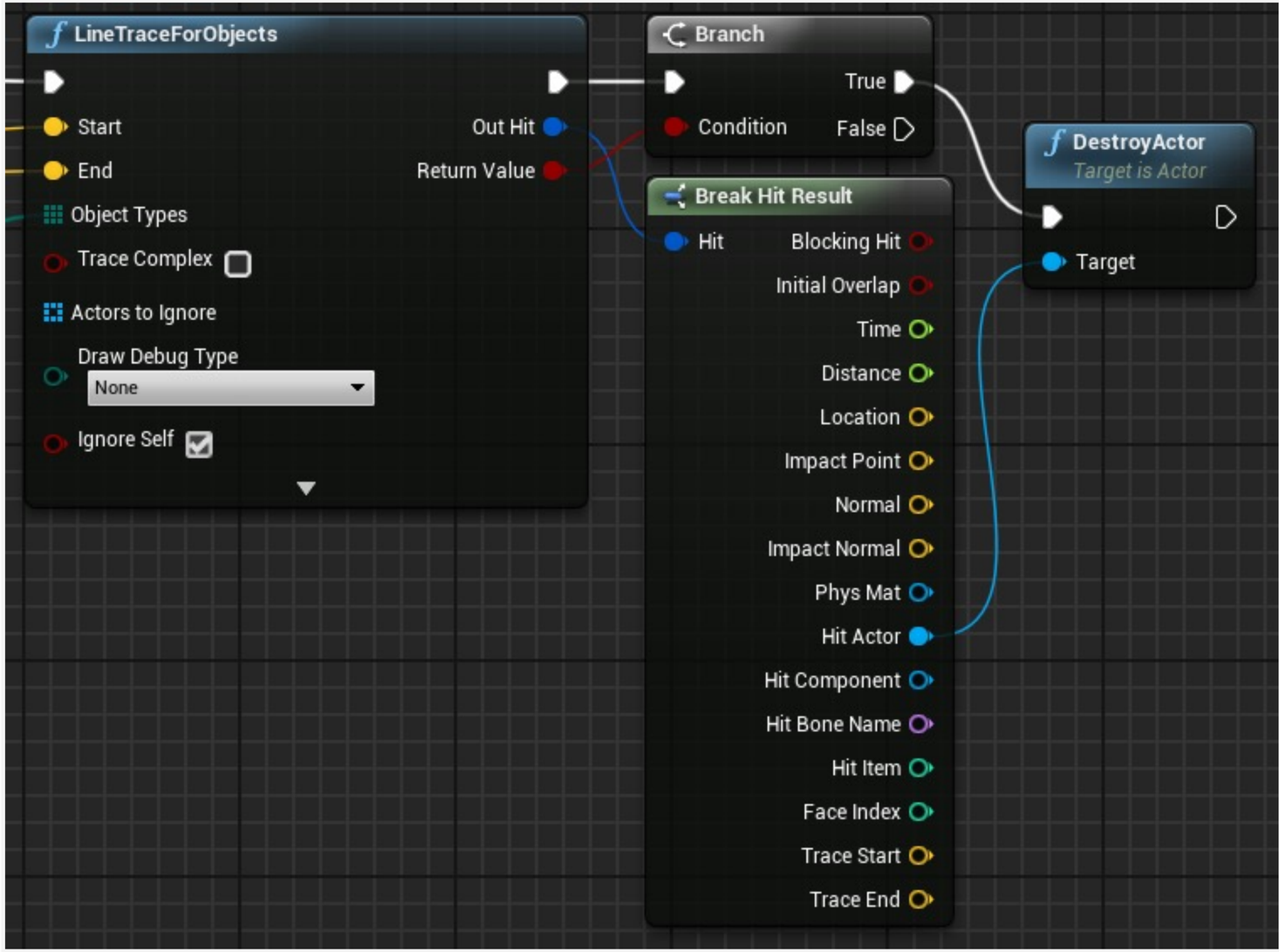
- 阻止命中（Blocking Hit）:布尔值，指示是否有阻止命中。
- 位置（Location）：命中位置。
- 法线（Normal）：场景空间中命中的法线矢量。
- 命中Actor（Hit Actor）：追踪对命中Actor的引用。

命中结果结构体：示例

右侧示例使用了Break Hit Result节点。如果发生命中，则对象的线条追踪（LineTraceForObjects）函数的“输出命中”（Out Hit）输出参数返回命中结果结构体。

销毁Actor（DestroyActor）函数使用命中Actor引用作为目标，从游戏中移除命中的Actor。

下一张幻灯片将介绍LineTraceForObjects函数。



Line Trace For Objects

LineTraceForObjects函数测试定义线条上是否有碰撞，并返回与函数调用中指定的某个Object类型相匹配的第一个命中Actor的命中结果结构体。

输入

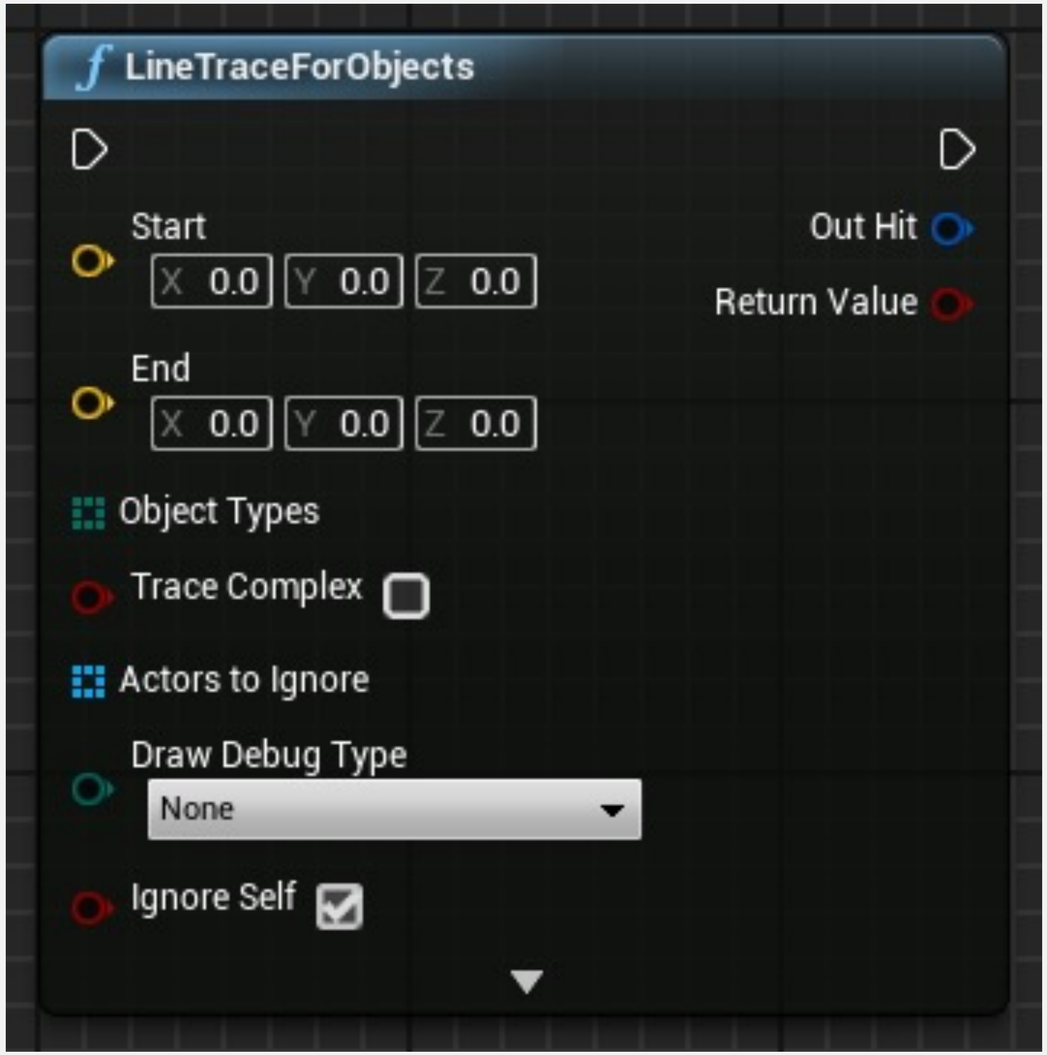
•起点（Start）和终点（End）：位置矢量，定义用于碰撞测试的线条的起点和终点。

•对象类型（Object Types）：包含碰撞测试中将要使用的对象类型的数组。

•复杂追踪（Trace Complex）：指示是否使用复杂碰撞的布尔值。

•要忽略的Actor（Actors to Ignore）：应在碰撞测试中忽略的关卡Actor的数组。

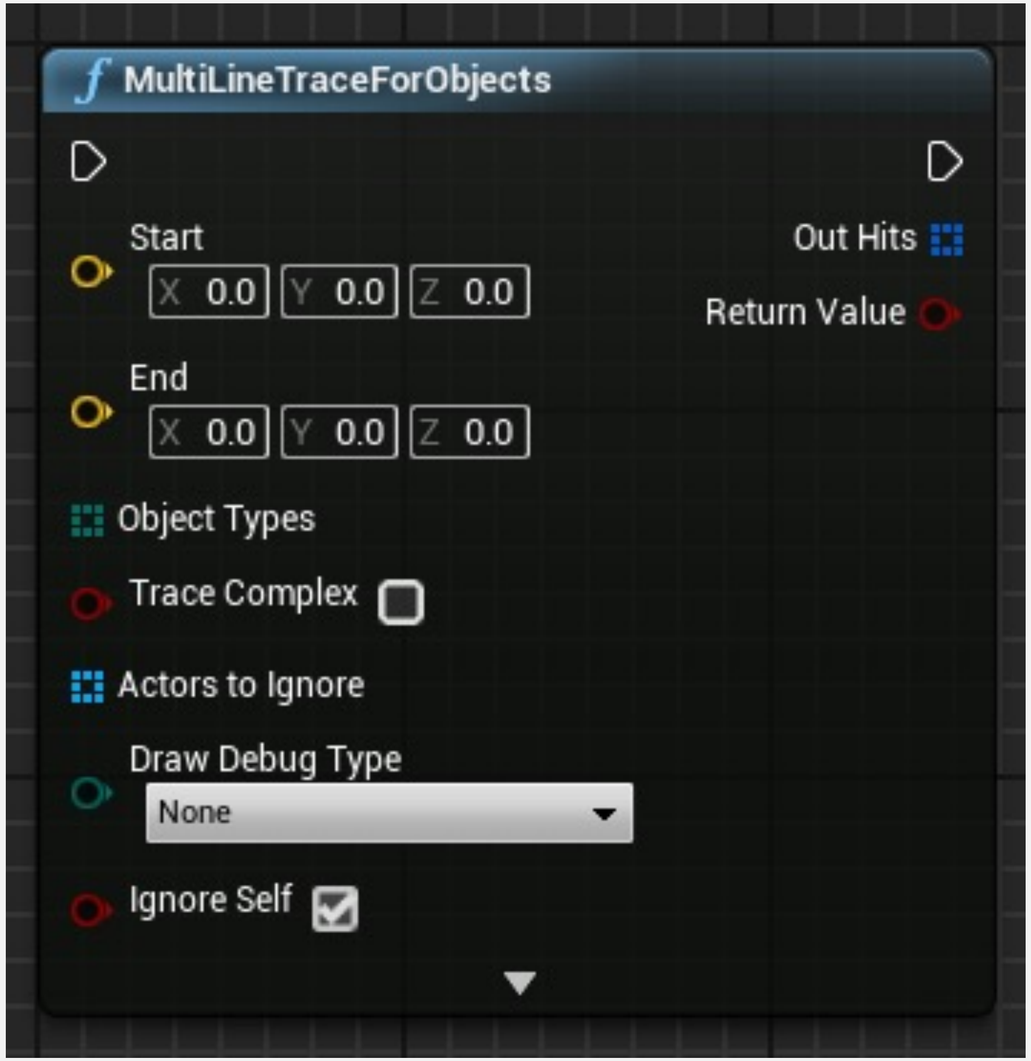
•绘制调试类型（Draw Debug Type）：允许绘制表示追踪的3D线条。



Multi Line Trace For Objects

MultiLineTraceForObjects函数的输入参数与LineTraceForObjects函数相同。

这两个函数的区别在于，MultiLineTraceForObjects函数返回命中结果结构体数组，而不是一个结构体，因此执行成本更高。



Line Trace By Channel

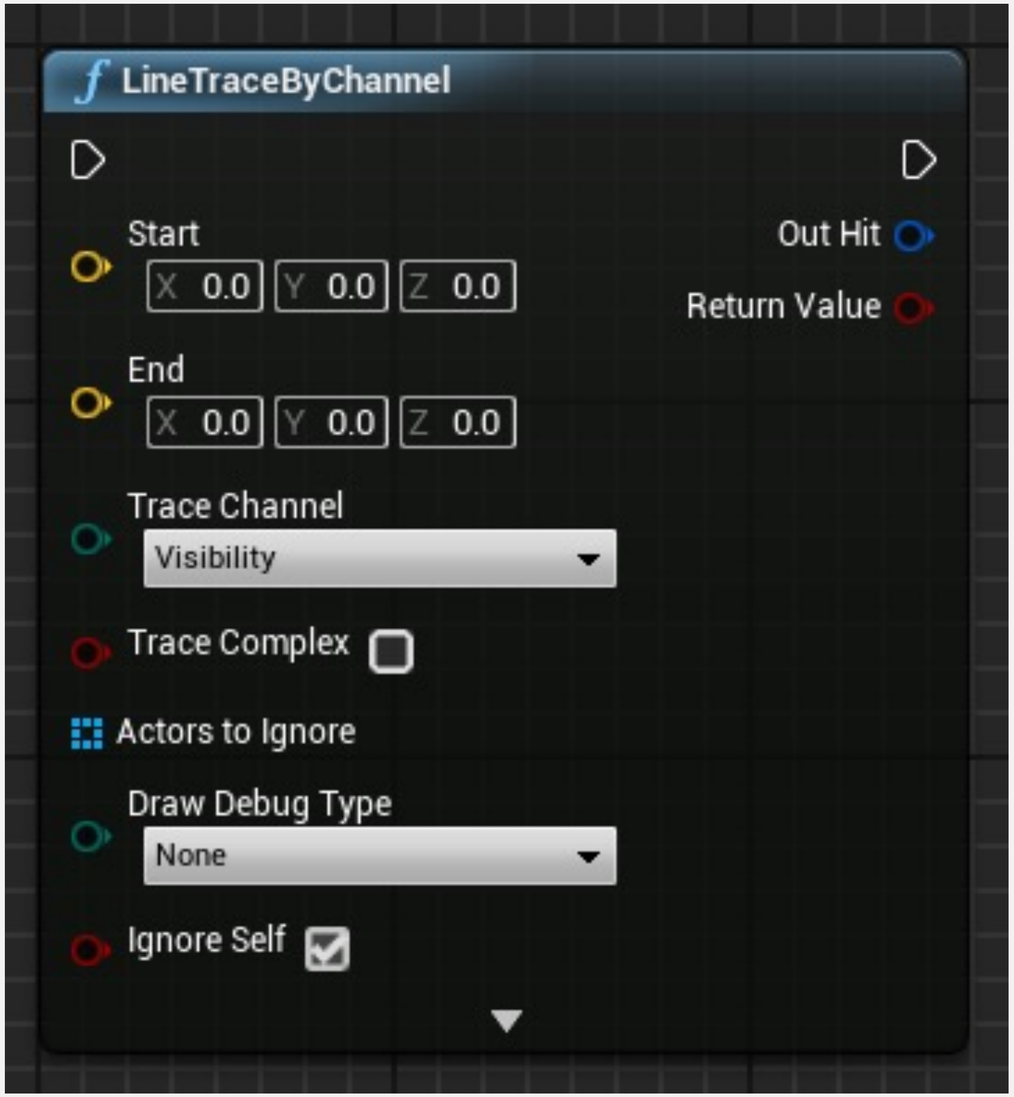
LineTraceByChannel函数使用“Visibility”或“Camera”线条通道，测试定义线条上是否有碰撞，并返回包含碰撞测试中第一个命中Actor的数据的命中结果结构体。

输入

•起点（Start）和终点（End）：位置矢量，定义用于碰撞测试的线条的起点和终点。

•追踪通道（Trace Channel）：用于碰撞测试的通道。通道可以是“Visibility”或“Camera”。

•复杂追踪（Trace Complex）、要忽略的Actor（Actors to Ignore）或绘制调试类型（Draw Debug Type）：LineTraceForObjects函数中使用的相同参数（见幻灯片7）。



Multi Line Trace By Channel

MultiLineTraceByChannel函数的输入参数与LineTraceByChannel函数相同。

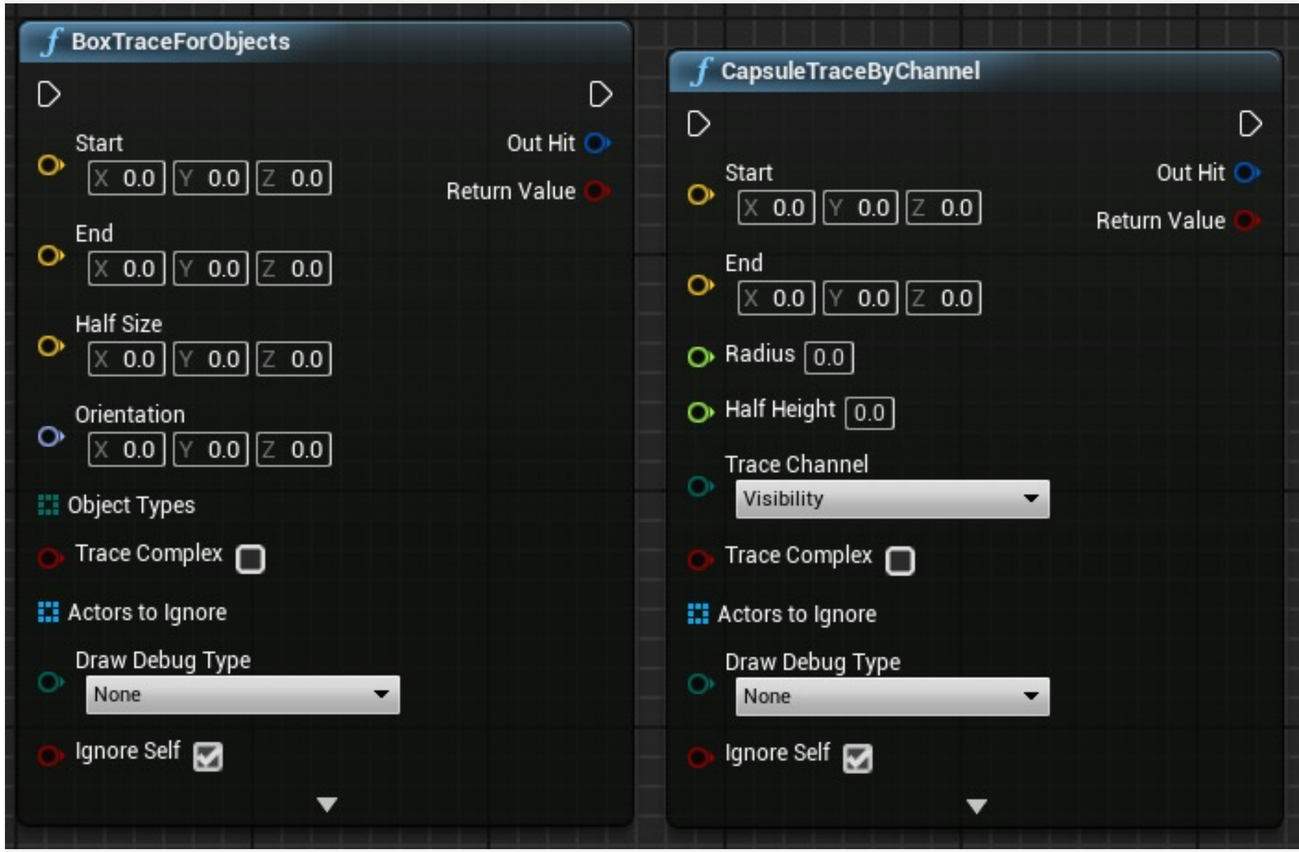
这两个函数的区别在于，MultiLineTraceByChannel函数返回命中结果结构体数组，而不是一个结构体，因此执行成本更高。

Shape Traces

追踪也可以使用形状实现。Box、胶囊体和球体形状有相应的追踪函数，但这些函数执行成本高于线条追踪。

对于所有这些形状，都有可以按通道和按Object类型进行追踪的函数。还有一些用于单一命中或多个命中的函数。

右图显示了BoxTraceForObjects和CapsuleTraceByChannel函数。



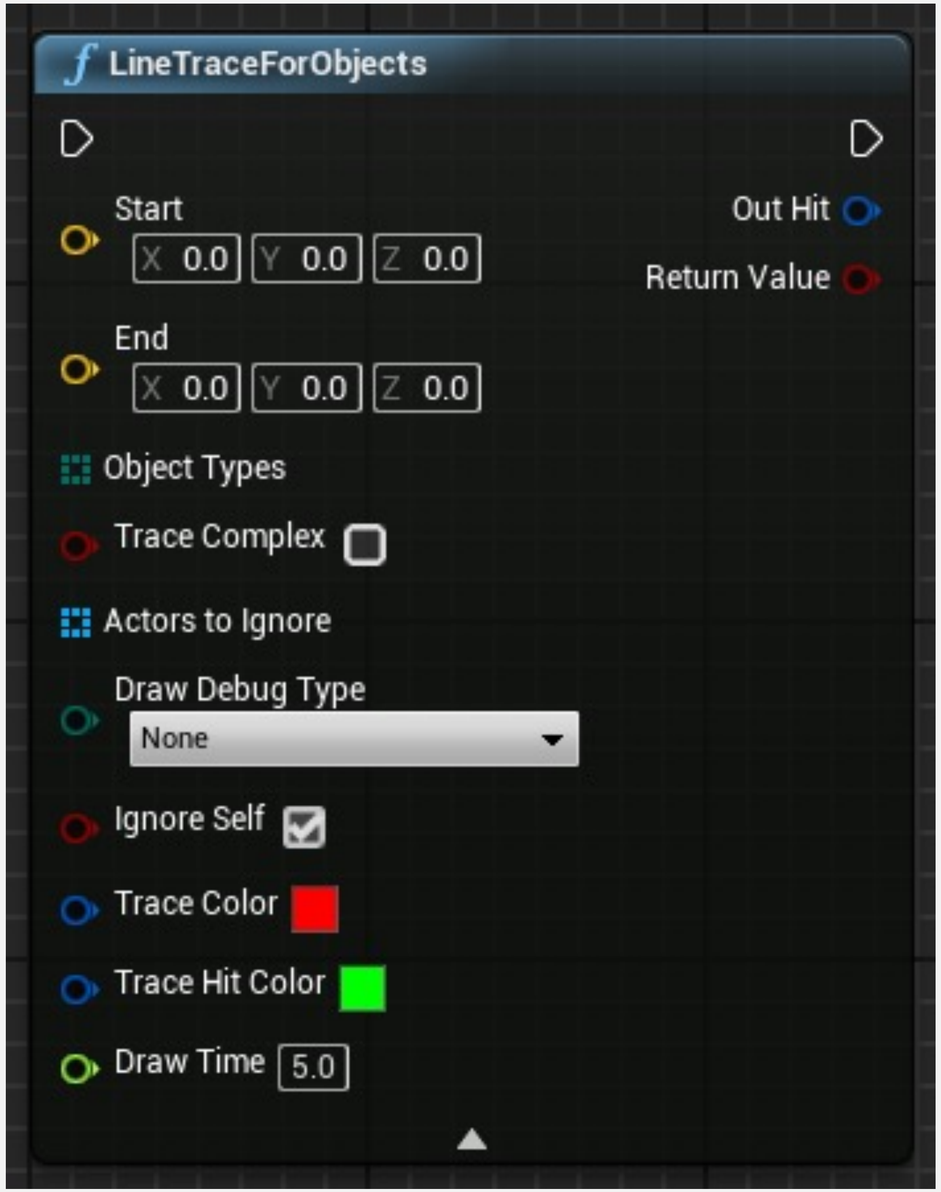
调试

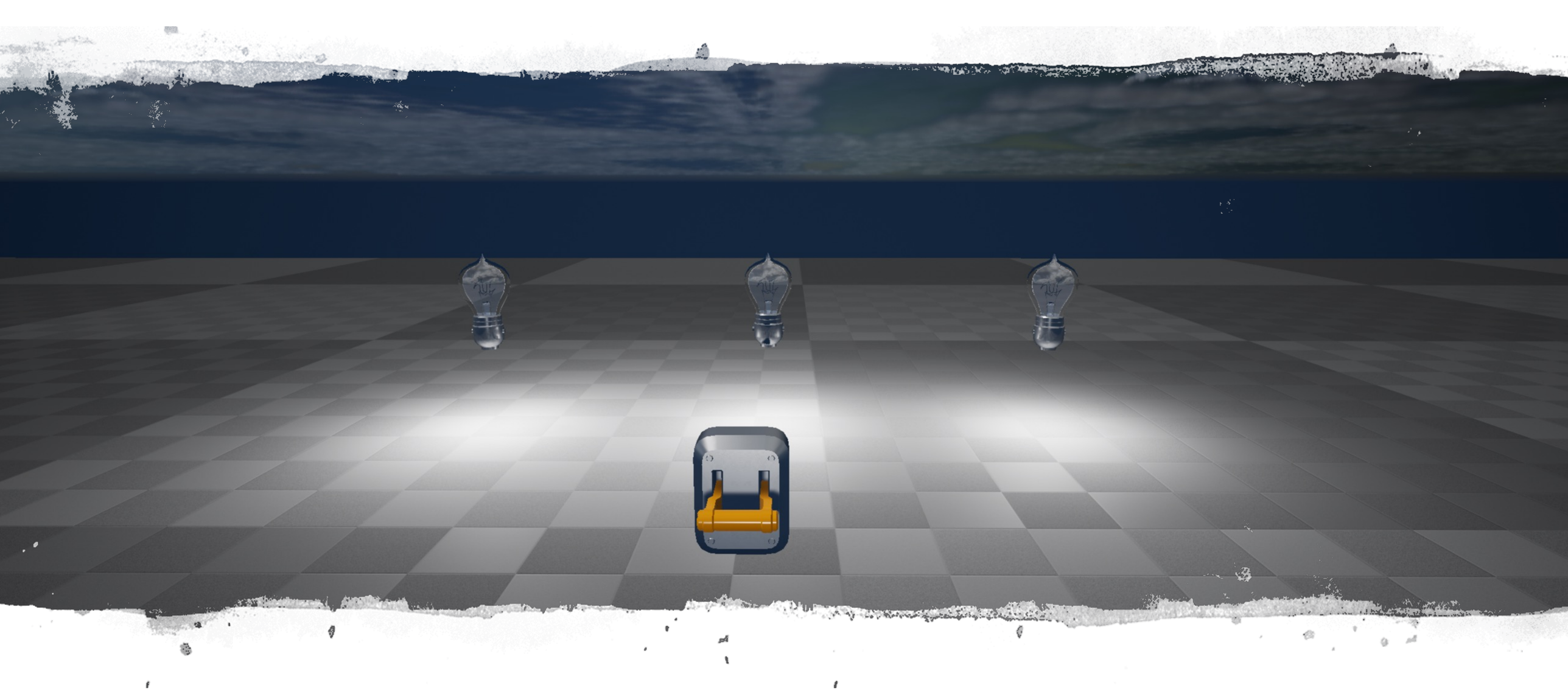
Trace 节点有一个选项，用于绘制测试追踪时有用的线条。

绘制调试类型（Draw Debug Type）参数可以设置为以下某个值：

- 无（None）：不绘制线条。
- 持续一帧（For One Frame）：线条仅出现一帧。
- 持续时间（For Duration）：线条在“绘制时间”（Draw Time）参数中指定的时间保持出现。
- 持久（Persistent）：线条不消失。

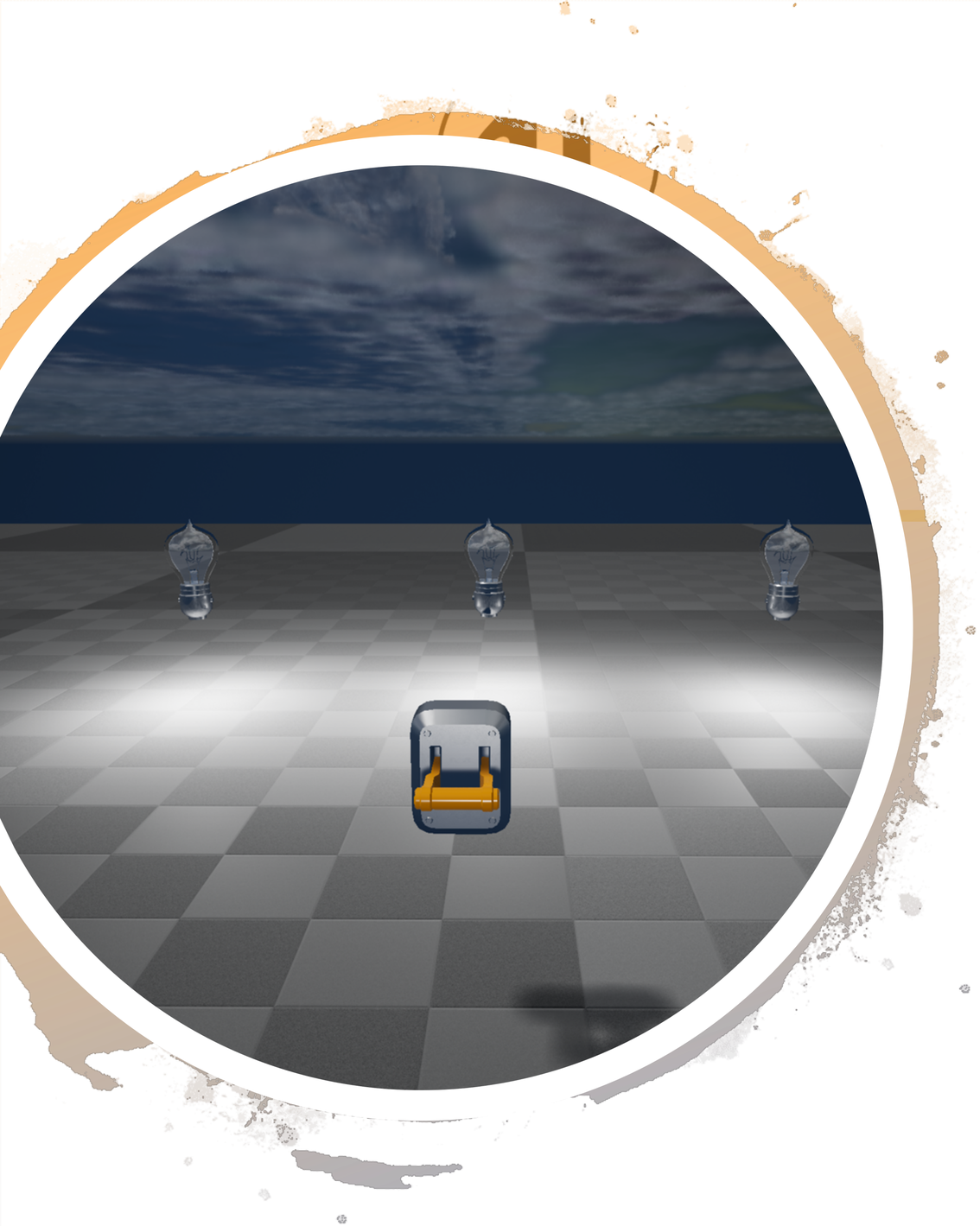
要显示追踪颜色（Trace Color）、追踪命中颜色（Trace Hit Color）和绘制时间（Draw Time）参数，单击函数底部的小箭头。





DEMO

电源总开关



DEMO

关闭电源总开关时，所有电器全部关闭

- 电源总开关不知道都那哪些电器
- 所有的电器都应该关心电源关闭事件

- **知识点**
  - 蓝图事件调度器

事件调度器

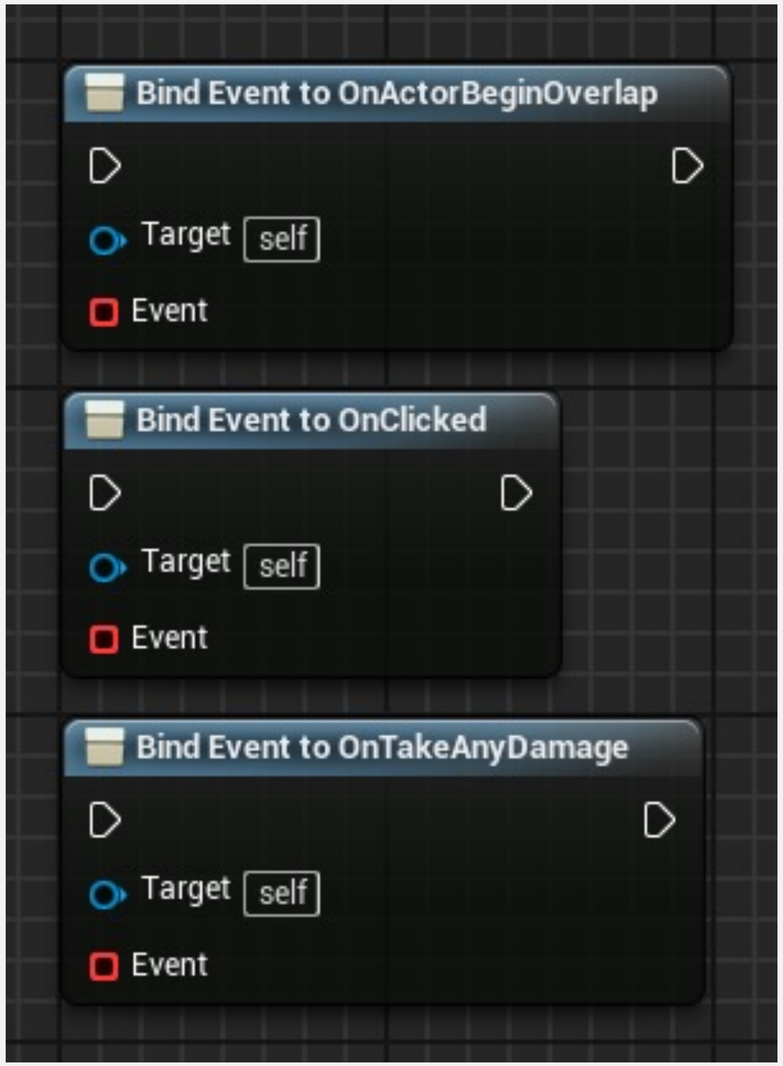
- 事件调度器允许蓝图类之间进行一种类型的通信
- 事件调度器在“我的蓝图”（My Blueprint）面板中创建。右图显示了用名称“ButtonPressed”创建的事件调度器。

绑定事件（BIND EVENT）节点

绑定事件（Bind Event）节点将一个事件绑定到另一个事件或事件调度器，后者可能位于另一个蓝图中。当调用一个事件时，也会调用与之绑定的所有其他事件。绑定可以使用事件委托（事件节点上的红色引脚）来完成。

输入

- 目标（Target）：这个对象包含用于接收绑定的事件。
- 事件（Event）：这是对稍后将会调用的事件的引用，该事件将绑定到另一个事件。



DEMO

使用 Timeline 制作动画

DEMO

实现功能

- 门的开关动画：控制 Transform
- 灯泡的开关动画：控制材质参数

知识点

- Timeline
- Dynamic Material Instance

时间轴

Timeline

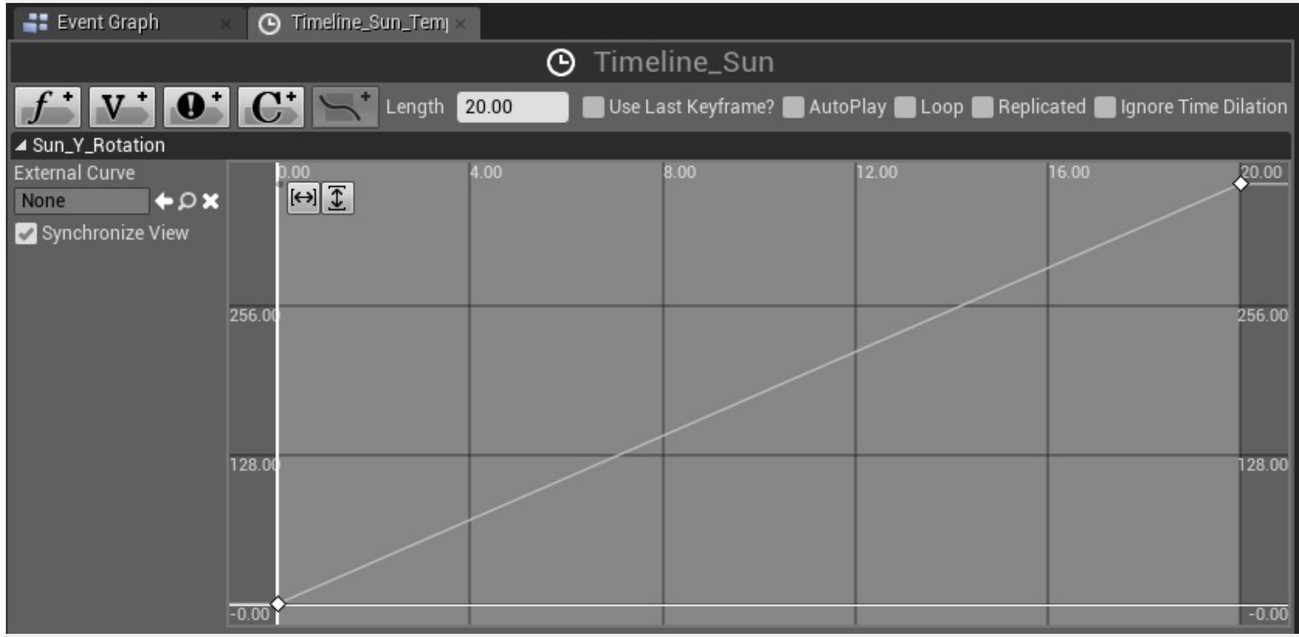
时间轴

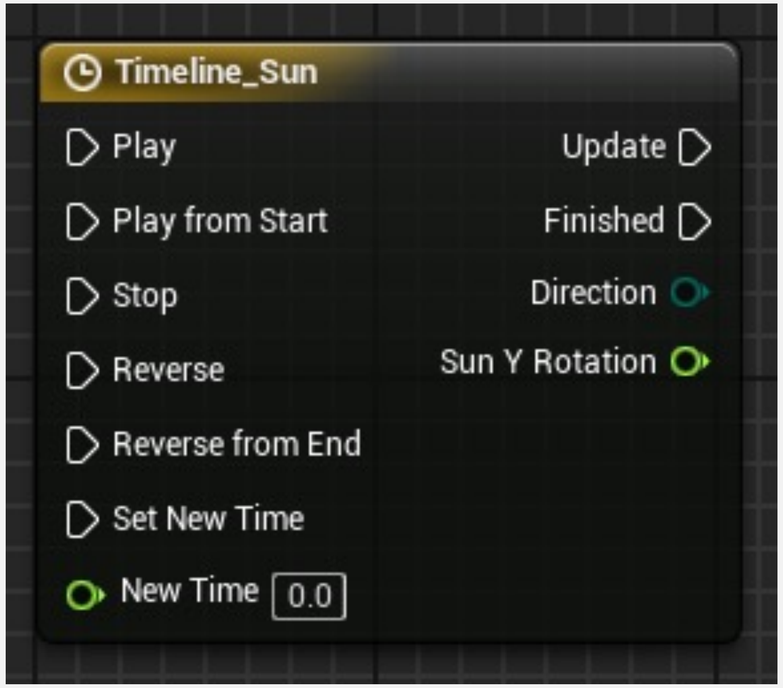
时间轴允许在蓝图内创建简单的基于时间的动画。将时间轴添加到事件图表后，可以通过双击在蓝图编辑器中进行编辑。

添加到时间轴的变量显示为输出参数，因此可以访问它们的值。右上图显示了一个时间轴编辑器，其中有一个名为“Sun_Y_Rotation”的轨迹。

时间轴运行期间，将持续调用“更新”（Update）引脚。

时间轴可以向前或向后播放。

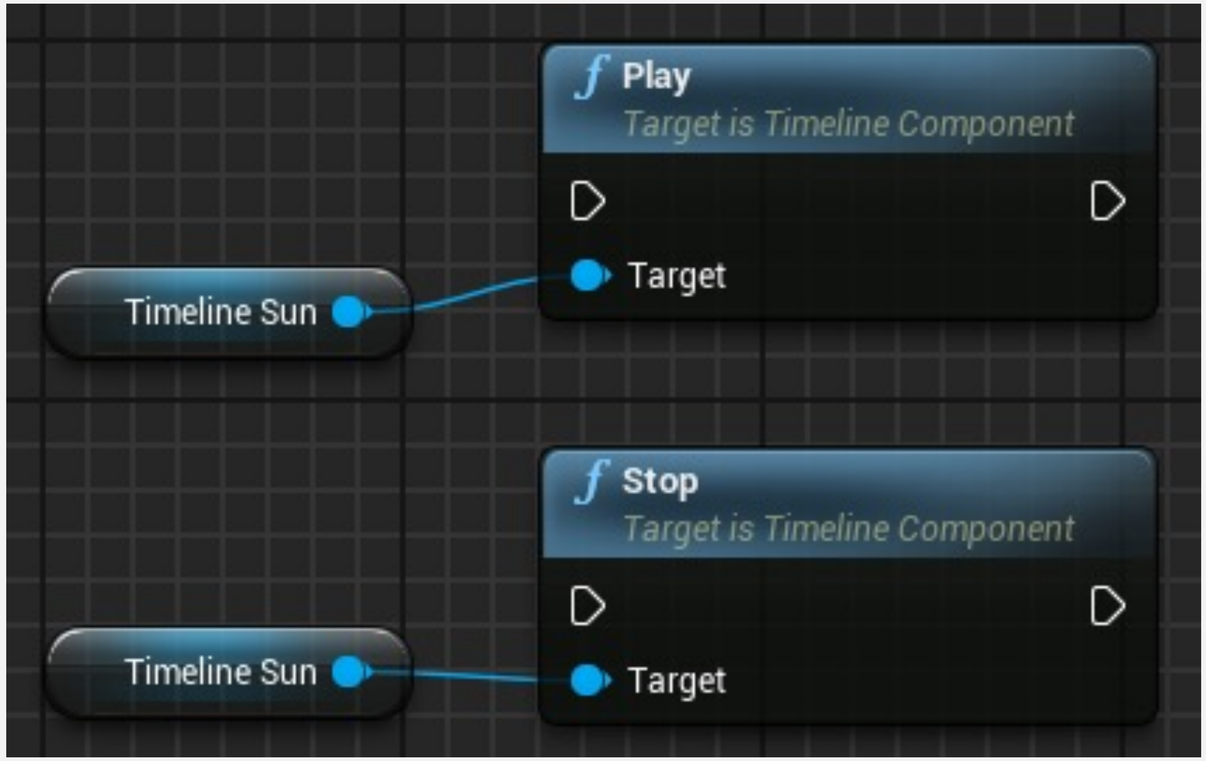




时间轴与变量

创建后，可以作为变量访问时间轴，因此可以从图表中的任意位置调用时间轴函数。

时间轴的“获取”（Get）节点可从“我的蓝图”（My Blueprint）面板的“变量”（Variables）>“组件”（Components）获取。



LERP节点

“Lerp”是“线性插值”的简称。该函数节点根据Alpha参数的值，生成两个指定值的范围内的值。

输入

•A：接收浮点值，它表示可以返回的最低值。

•B：接收浮点值，它表示可以返回的最大值。

•Alpha：接收“0-1”之间的浮点值。如果值为“0”，则返回最低值；如果值为“1”，则返回最大值。

输出

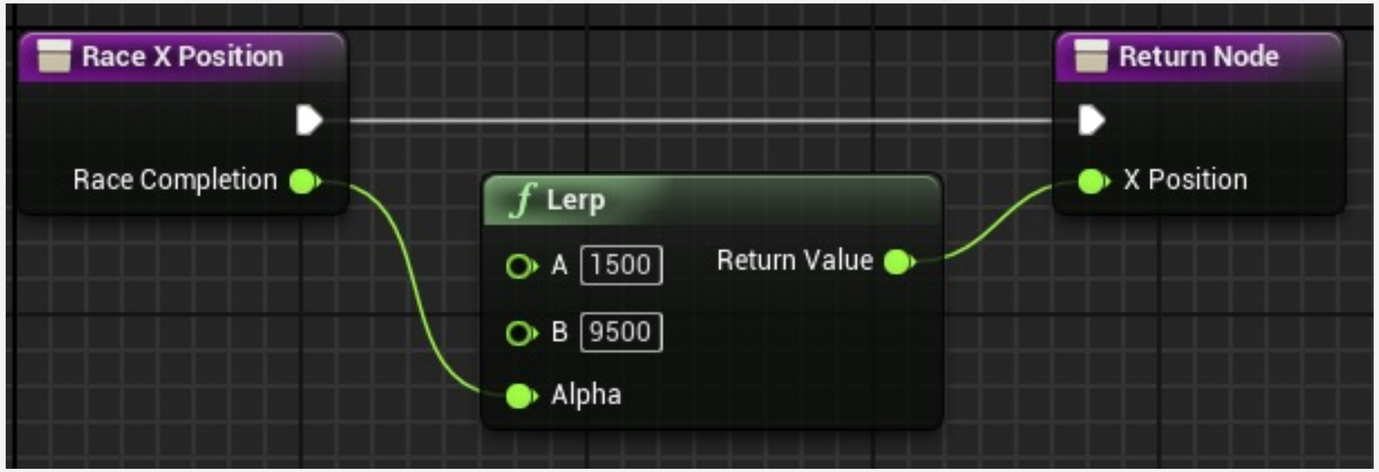
•返回值（Return Value）：输出介于A和B参数值之间的浮点值，该值取决于Alpha参数的值。

LERP节点：示例

在右侧示例中，沿着X轴有一条轨迹。这条轨迹起始位置是X = 1500，终止位置是X = 9500。

Lerp节点使用的A参数值设为“1500”，B参数设为“9500”。Alpha参数接收“0-1”之间的值，表示轨迹的完成度，并返回相应的值来表示X位置。

如果Alpha参数值设为“0.5”，则返回值为“5500”，表示轨迹中间。



插值到（INTERP TO）

Interp To函数用于平滑地更改值，直到达到指定目标值。一些示例包括用于浮点值的FInterp To函数、用于矢量的VInterp To和用于旋转体的RInterp To函数。

输入

•当前（Current）：当前值。

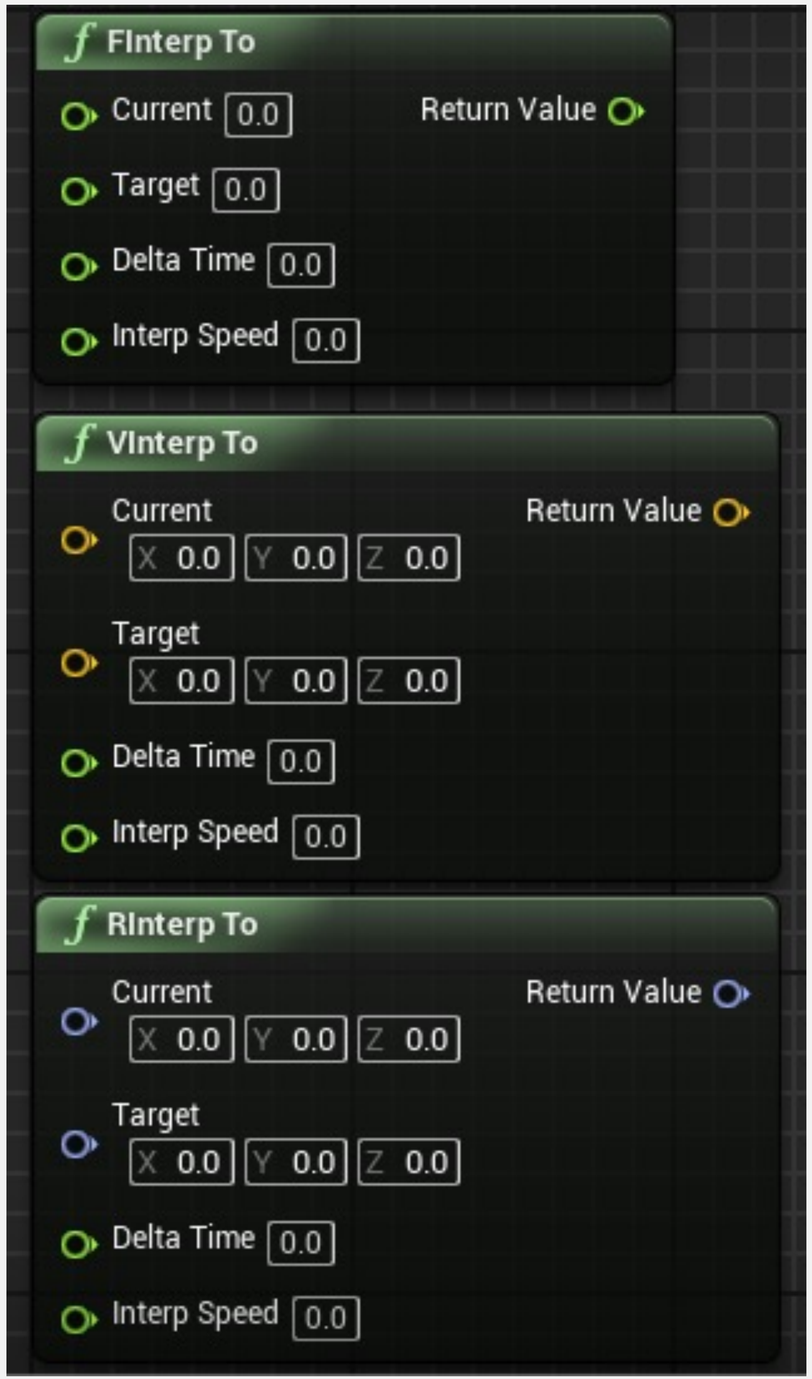
•目标（Target）：要达到的目标值。

•时间差量（Delta Time）：自上一次执行以来经过的时间间隔。

•插值速度（Interp Speed）：插值速度。

输出

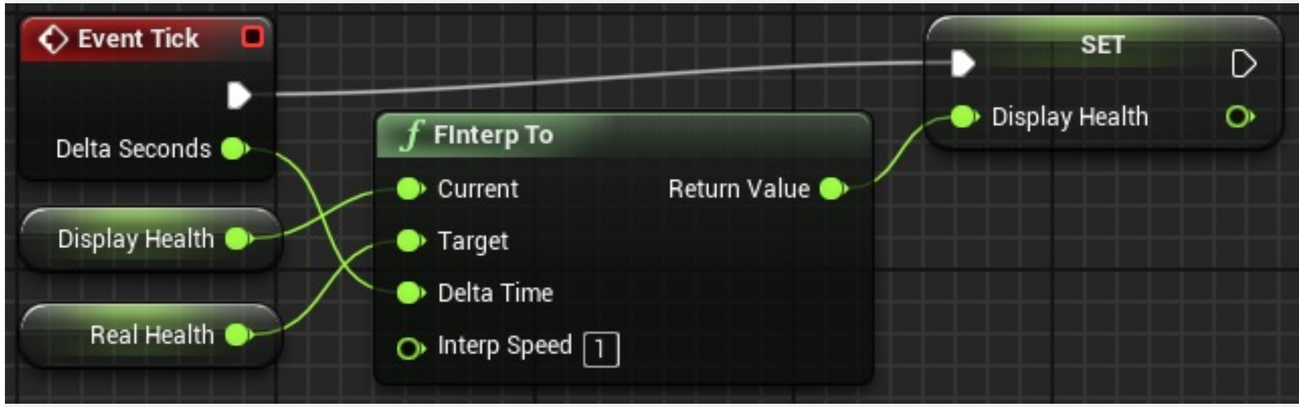
•返回值（Return Value）：更接近于目标值的新值。



插值到（INTERP TO）：示例

右侧示例包含两个变量。“实际生命值”（Real Health）变量存储玩家的当前生命值。“显示生命值”（Display Health）变量用于在画面上显示生命条。

当玩家遭受伤害时，Real Health值立即改变，但Display Health值将使用FInterp To函数进行修改，因此生命值条将平滑地减少，直到Display Health值等于Real Health值为止。



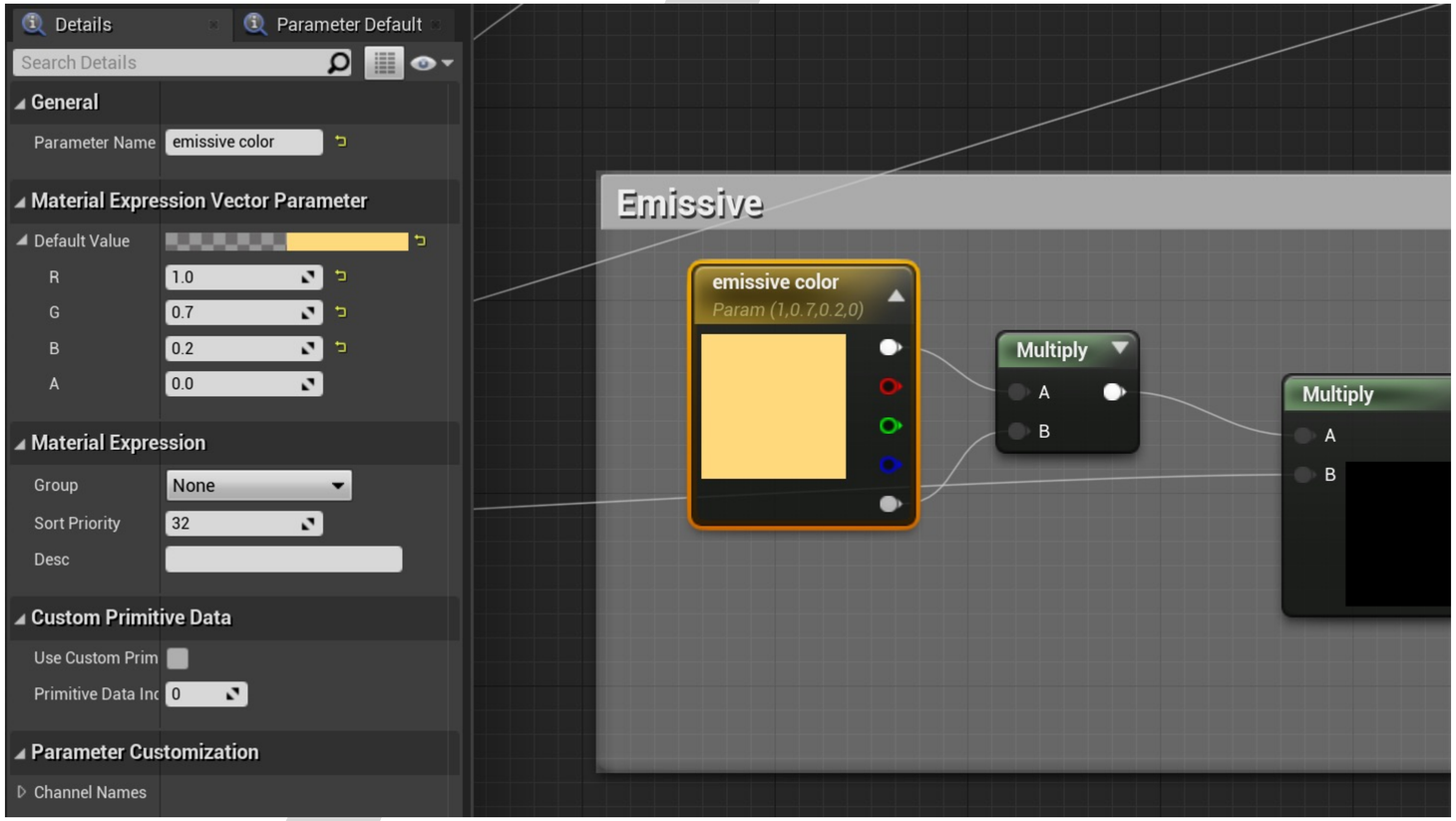
动态材质实例

Dynamic Material Instance

回顾上周的门

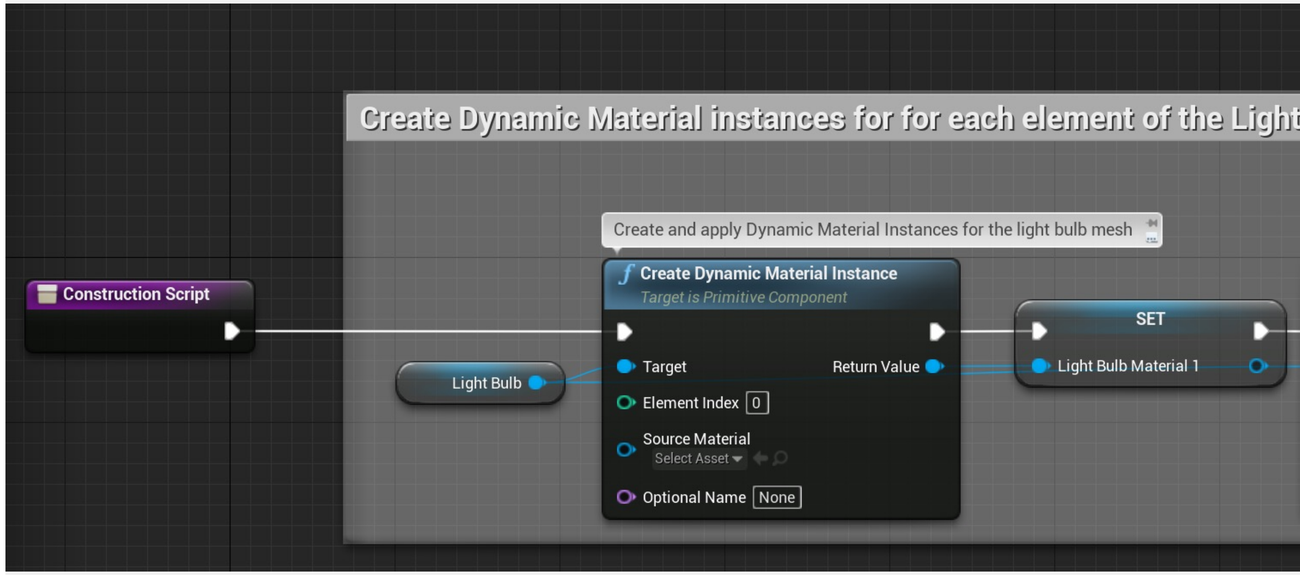
- 通过指定另外一个材质实现“红色”的门
- 这种方式只能实现瞬间的“切换”
- 有没有一种方法可以实现平滑过渡？

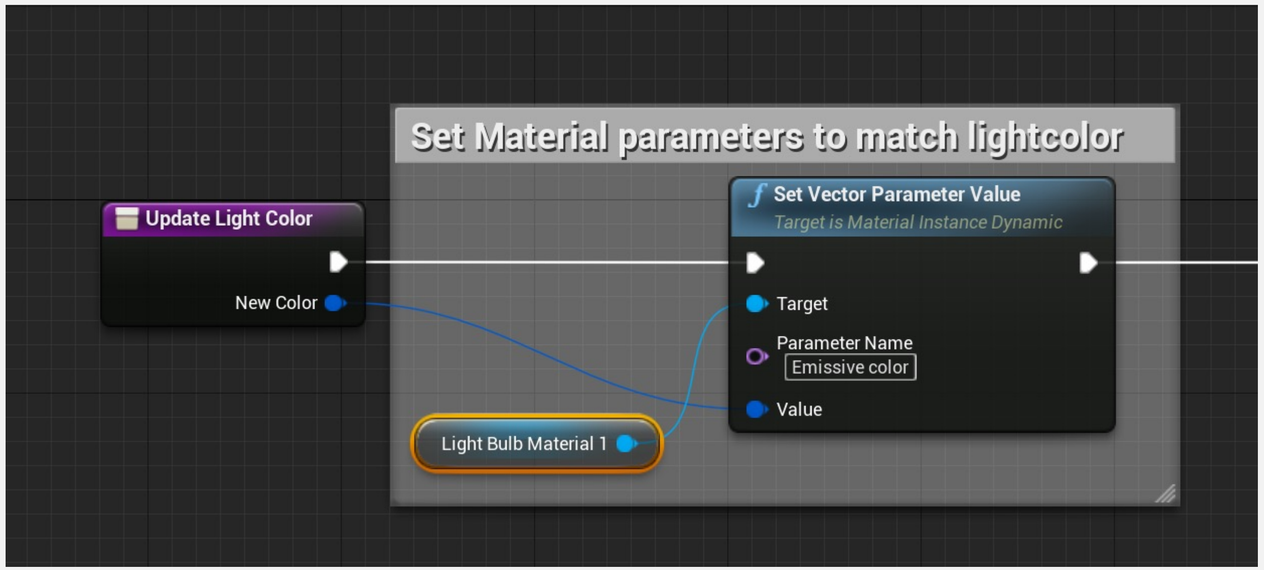
材质参数


Details
Parameter Default
Search Details
General
Parameter Name
emissive color
Material Expression Vector Parameter
Default Value
R
1.0
G
0.7
B
0.2
A
0.0
Material Expression
Group
None
Sort Priority
32
Desc
Custom Primitive Data
Use Custom Prim
Primitive Data Inc
0
Parameter Customization
Channel Names
Emissive
emissive color
Param (1,0.7,0.2,0)
Multiply
A
B
Multiply
A
B

使用动态材质实例

- 创建动态材质实例
- 更新材质参数





结构体

Structures

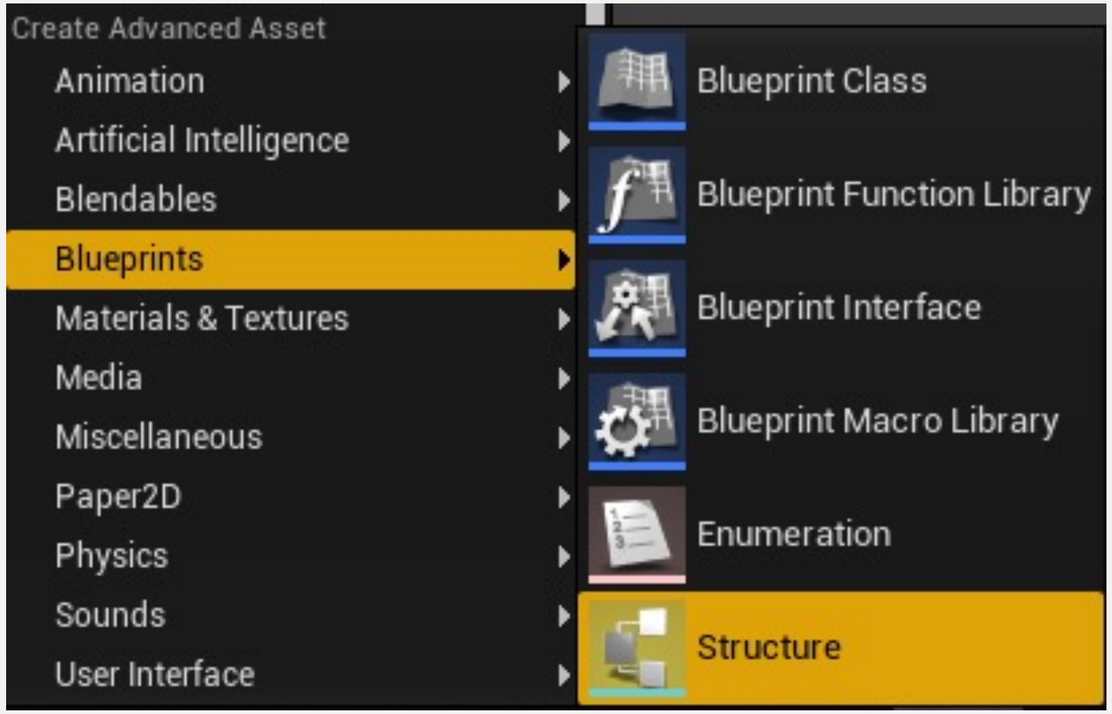
结构体（Struct）

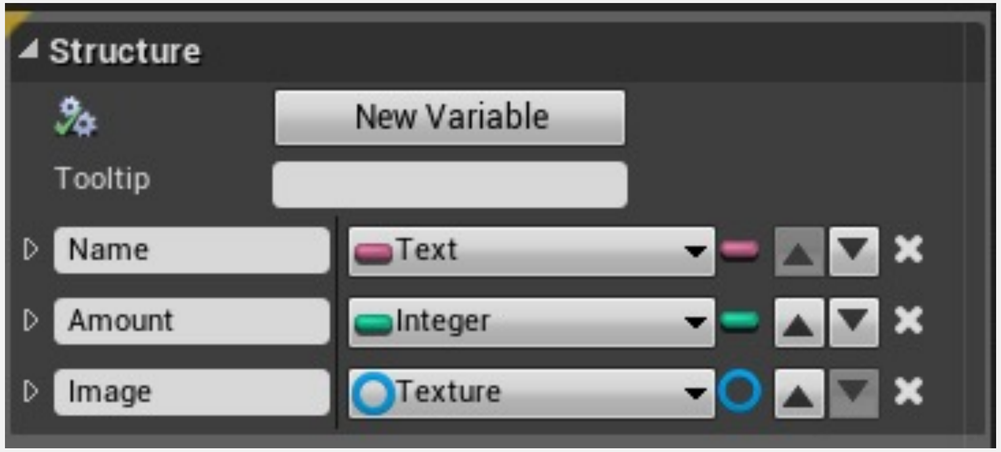
- 结构体（struct）可以用来将多个相关变量汇总在一个位置。
- 结构体的成员变量可以有不同的类型。
- 结构体还可以包含其他结构体和数组。

要创建新结构体，单击内容浏览器中的绿色“新增”（Add New）按钮，然后在“蓝图”（Blueprints）子菜单中选择“结构体”（Structure）。将它重命名为“项目结构体”（Item Struct）。

双击“项目结构体”（Item Struct）进行编辑。

在“我的蓝图”（My Blueprint）面板中，单击“变量”（Variables）类别中的“+”按钮以向该结构体添加变量。



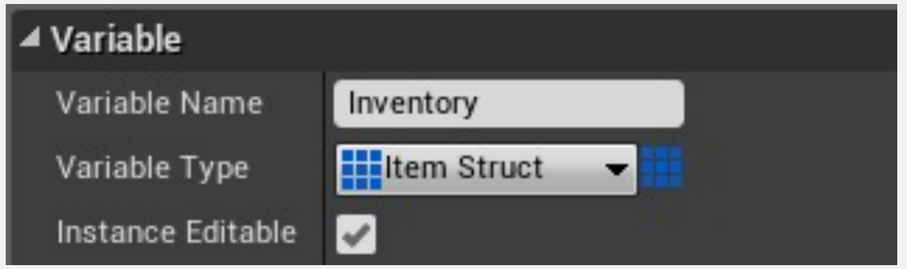


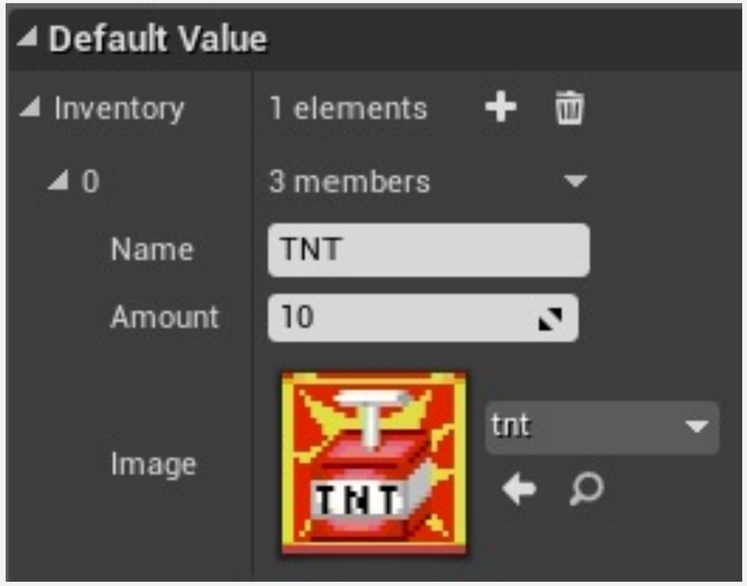
结构体：示例

作为示例，在蓝图中创建一个表示玩家物品栏的新变量。将其命名为“物品栏”（Inventory），并使用上一张幻灯片中创建的变量类型“项目结构体”（Item Struct）。单击“变量类型”（Variable Type）下拉菜单旁边的图标，并选择“数组”（Array）。

编译蓝图。

在“物品栏”（Inventory）变量的“细节”（Details）面板的“默认值”（Default Value）部分中，可以向数组添加新元素。每个元素都将包含Item Struct结构体中定义的变量。





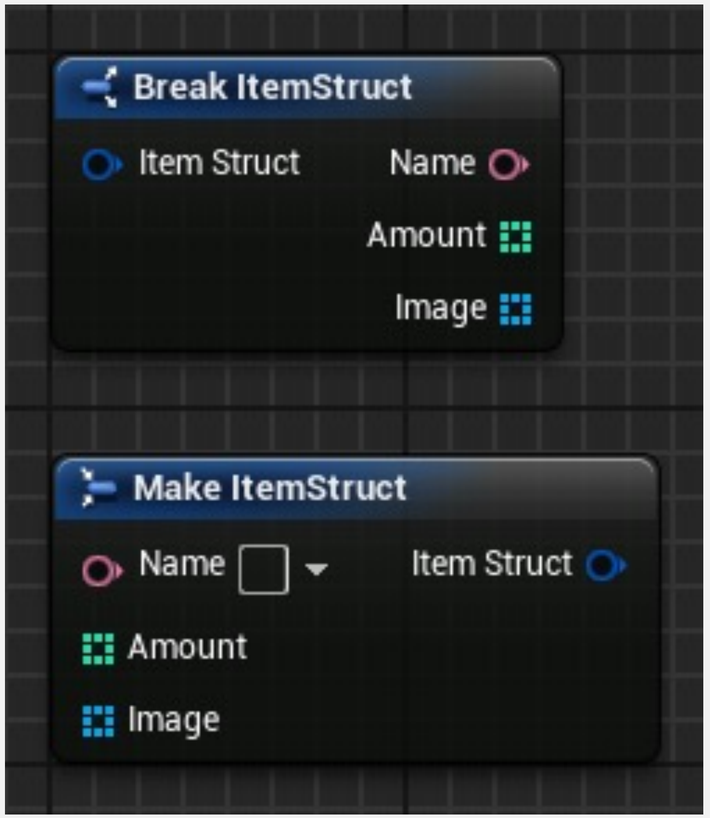
结构体：Break & Make

当某个结构体被引用时，分解（Break）和创造（Make）节点将在蓝图中变为可用。

分解节点以结构体为输入，并将它的元素分隔开来。

创造节点以独立的元素为输入，并创建新的结构体。

右图显示了Item Struct结构体的分解和创造节点。



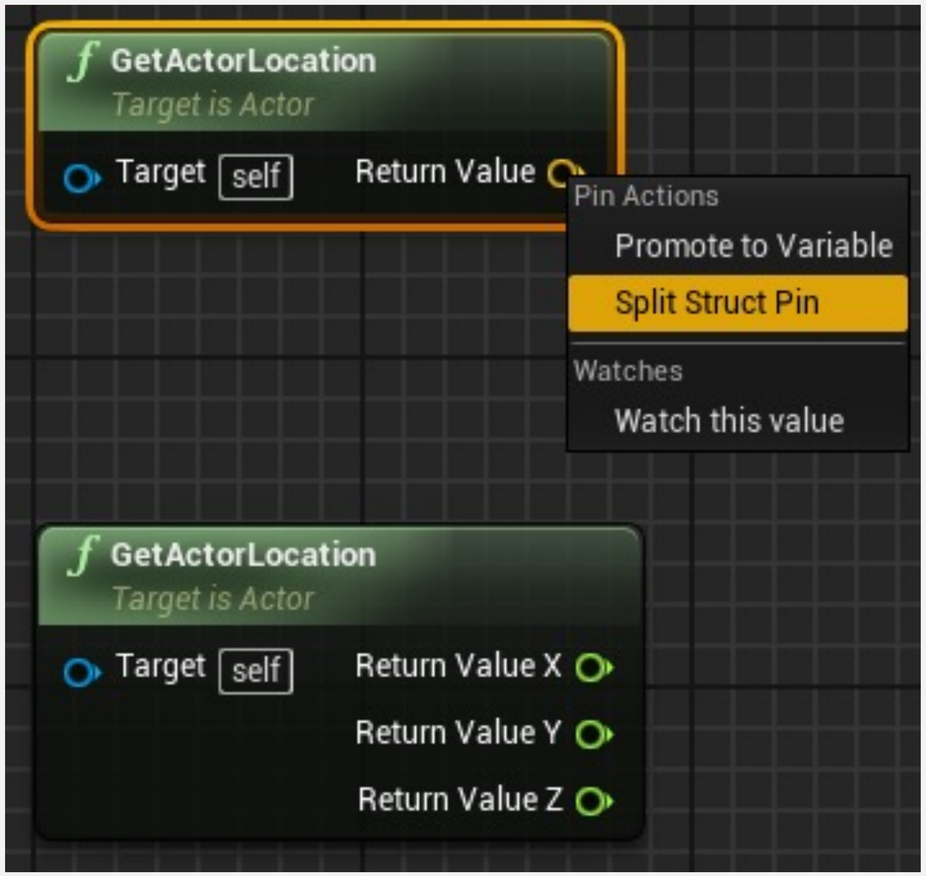
结构体：拆分结构体引脚

当一个结构体是某个函数的输入或输出参数时，可以将它的引脚拆分，为结构体的每个元素创建输出引脚。

为此，右键单击结构体引脚，并选择“拆分结构体引脚”（Split Struct Pin）。

例如，蓝图中的矢量是一个结构体，它包含三个浮点变量，名称分别为“X”、“Y”和“Z”。

右图显示具有结构体引脚且每个结构体元素也有相应引脚的GetActorLocation函数。

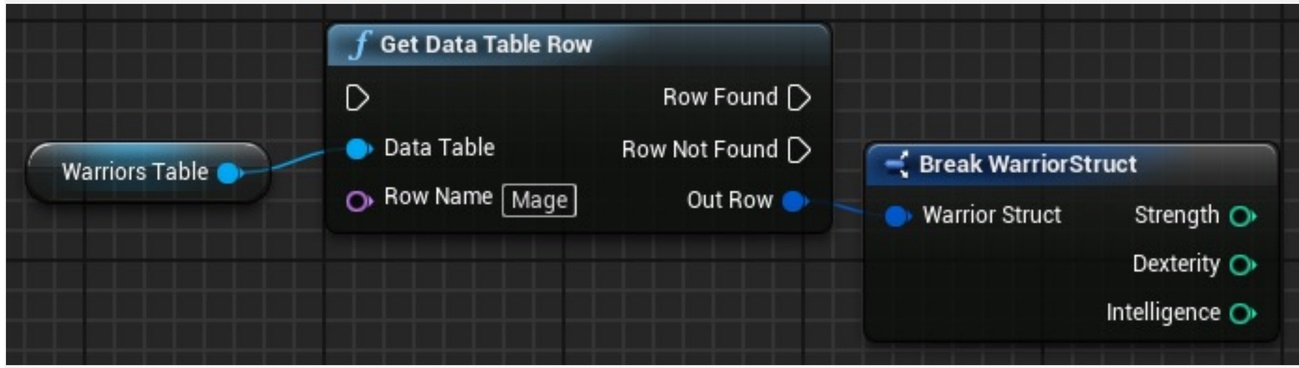


数据表

可以创建一个变量以在蓝图中表示数据表。

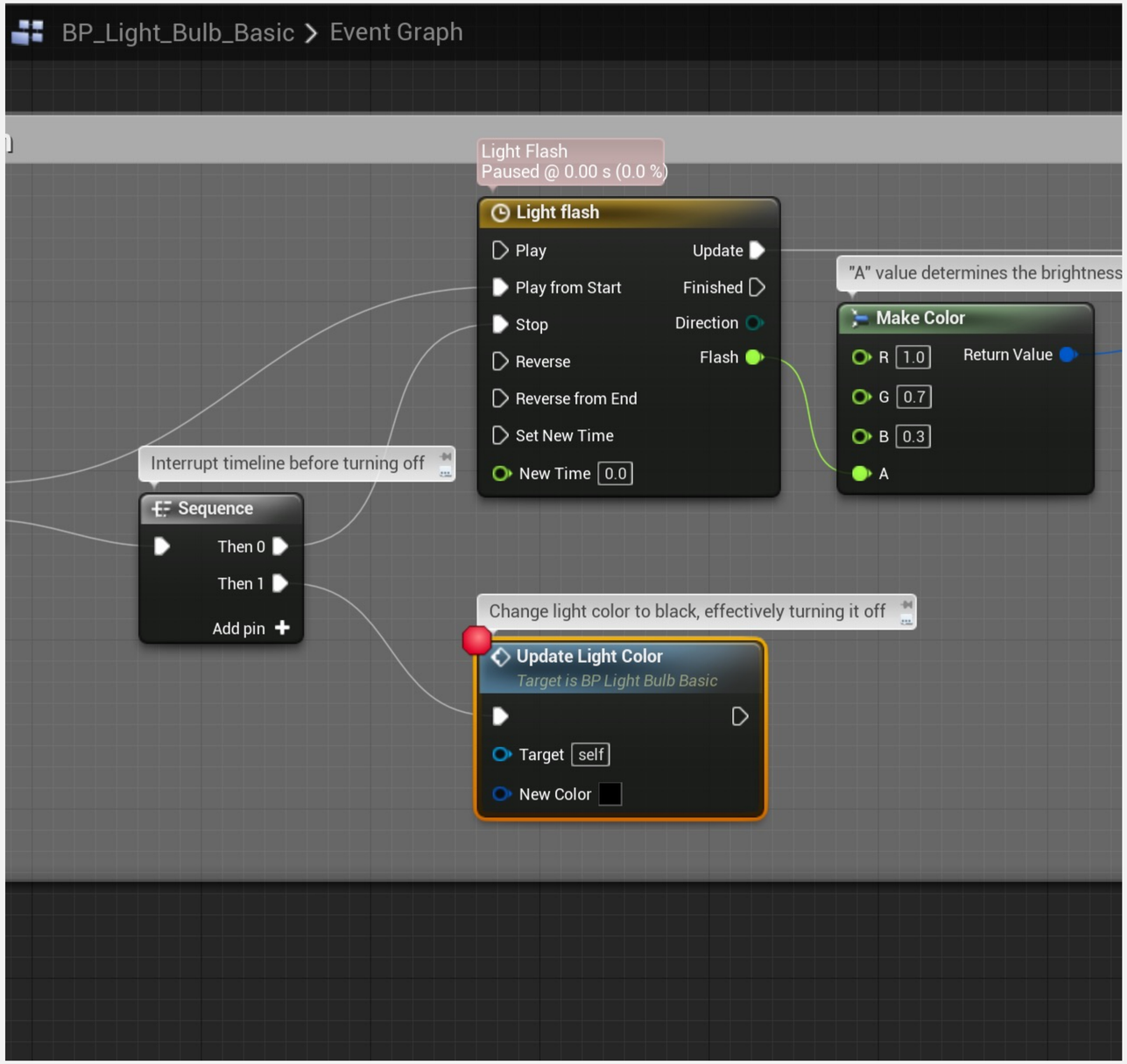
“获取数据表行”（Get Data Table Row）函数使用“行名称”（Row Name）参数，将表中的一行作为结构体返回。

分解结构体（Break Struct）节点可以用于访问结构体的属性。

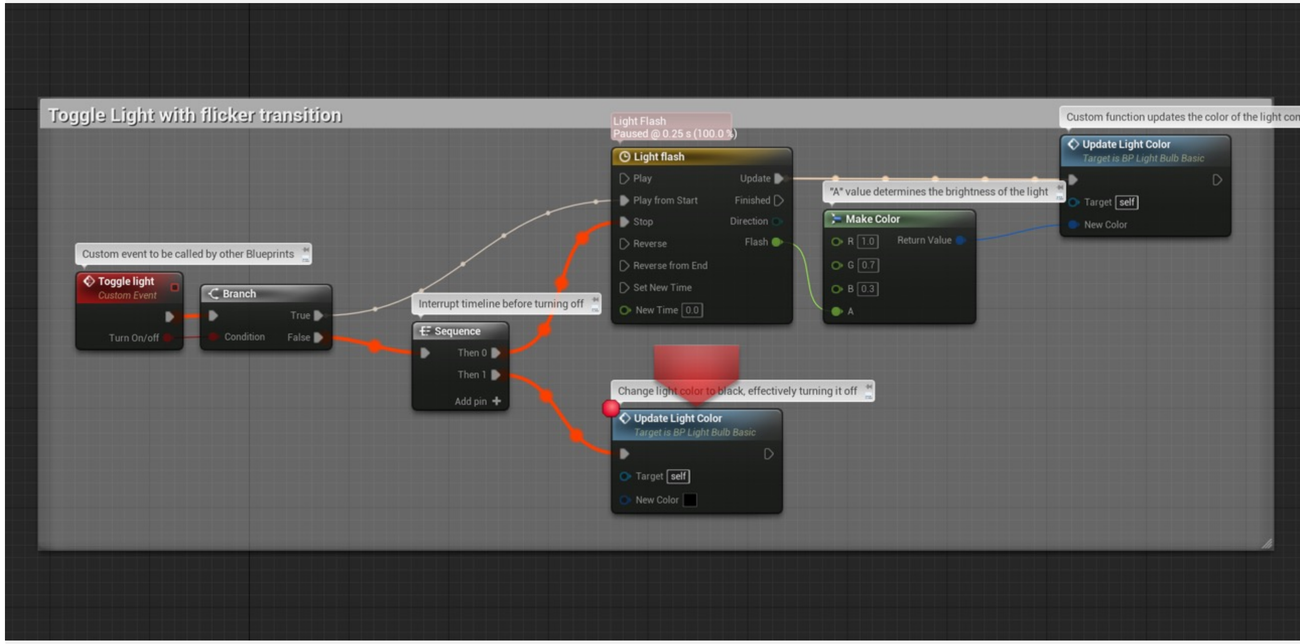


蓝图调试

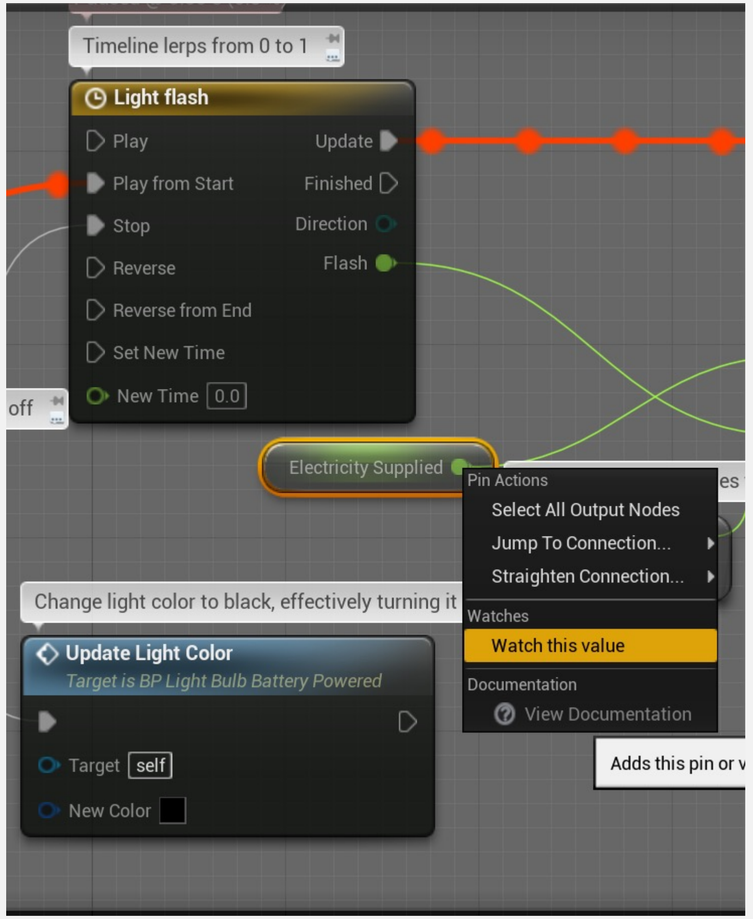
蓝图调试：断点


BP_Light_Bulb_Basic > Event Graph
Light Flash
Paused @ 0.00 s (0.0 %)
Light flash
Play
Play from Start
Stop
Reverse
Reverse from End
Set New Time
New Time 0.0
Update
Finished
Direction
Flash
"A" value determines the brightness
Make Color
R 1.0
G 0.7
B 0.3
A
Return Value
Interrupt timeline before turning off
Sequence
Then 0
Then 1
Add pin
Change light color to black, effectively turning it off
Update Light Color
Target is BP Light Bulb Basic
Target self
New Color

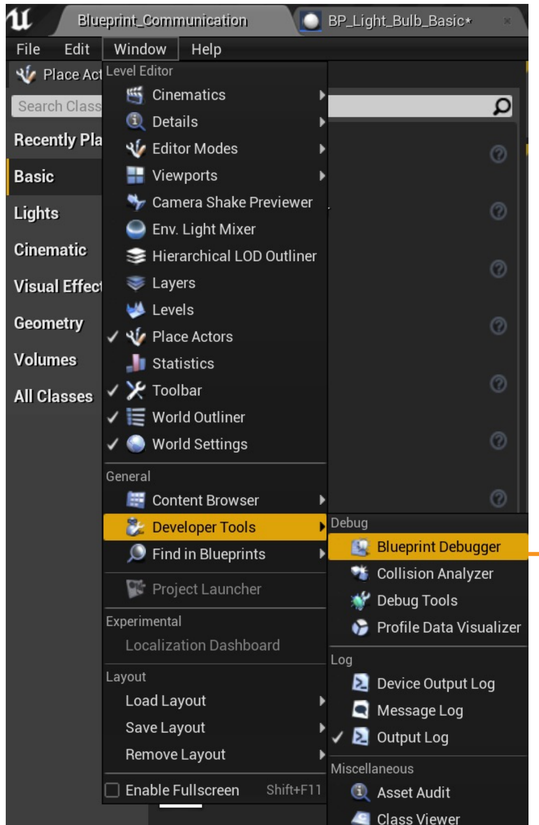
蓝图调试：逻辑可视化

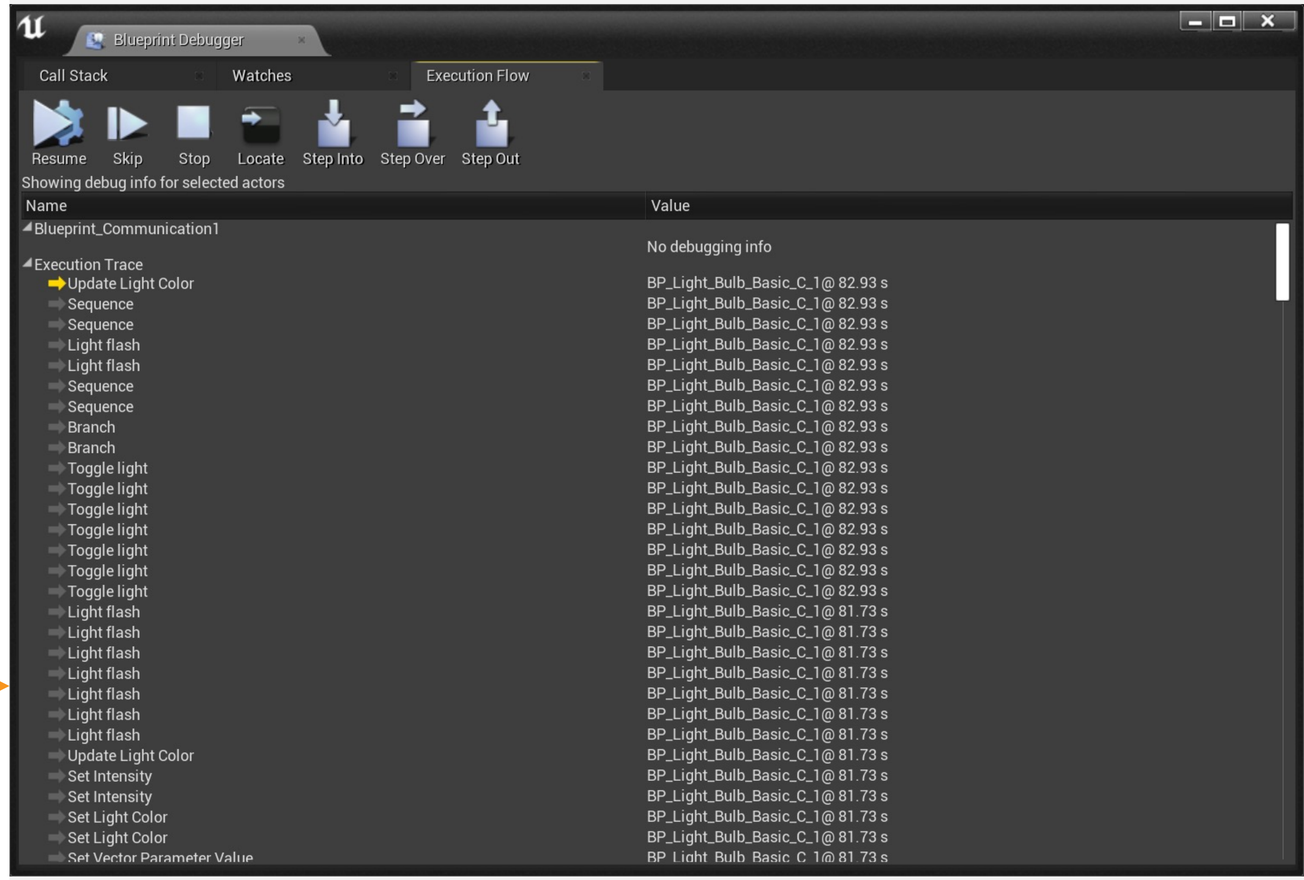

Toggle Light with flicker transition
Custom event to be called by other Blueprints
Toggle light
Custom Event
Turn On/off
Branch
Condition
True
False
Interrupt timeline before turning off
Sequence
Then 0
Then 1
Add pin
Light Flash
Paused @ 0.25 s (100.0 %)
Light flash
Play
Play from Start
Stop
Reverse
Reverse from End
Set New Time
New Time 0.0
Update
Finished
Direction
Flash
"A" value determines the brightness of the light
Make Color
R 1.0
G 0.7
B 0.3
A
Return Value
Custom function updates the color of the light con
Update Light Color
Target is BP Light Bulb Basic
Target self
New Color
Change light color to black, effectively turning it off
Update Light Color
Target is BP Light Bulb Basic
Target self
New Color

蓝图调试：观察变量的值


Timeline lerps from 0 to 1
Light flash
Play
Play from Start
Stop
Reverse
Reverse from End
Set New Time
New Time
0.0
Update
Finished
Direction
Flash
off
Electricity Supplied
Pin Actions
Select All Output Nodes
Jump To Connection...
Straighten Connection...
Watches
Watch this value
Documentation
View Documentation
Change light color to black, effectively turning it
Update Light Color
Target is BP Light Bulb Battery Powered
Target
self
New Color
Adds this pin or v

蓝图调试器窗口


Blueprint_Communication
BP_Light_Bulb_Basic*
File
Edit
Window
Help
Level Editor
Cinematics
Details
Editor Modes
Viewports
Camera Shake Previewer
Env. Light Mixer
Hierarchical LOD Outliner
Layers
Levels
Place Actors
Statistics
Toolbar
World Outliner
World Settings
General
Content Browser
Developer Tools
Find in Blueprints
Project Launcher
Experimental
Localization Dashboard
Layout
Load Layout
Save Layout
Remove Layout
Enable Fullscreen
Shift+F11
Debug
Blueprint Debugger
Collision Analyzer
Debug Tools
Profile Data Visualizer
Log
Device Output Log
Message Log
Output Log
Miscellaneous
Asset Audit
Class Viewer
Recently Pla
Basic
Lights
Cinematic
Visual Effect
Geometry
Volumes
All Classes


Blueprint Debugger
Call Stack
Watches
Execution Flow
Resume
Skip
Stop
Locate
Step Into
Step Over
Step Out
Showing debug info for selected actors
Name
Value
Blueprint_Communication1
No debugging info
Execution Trace
Update Light Color BP_Light_Bulb_Basic_C_1@ 82.93 s
Sequence BP_Light_Bulb_Basic_C_1@ 82.93 s
Sequence BP_Light_Bulb_Basic_C_1@ 82.93 s
Light flash BP_Light_Bulb_Basic_C_1@ 82.93 s
Light flash BP_Light_Bulb_Basic_C_1@ 82.93 s
Sequence BP_Light_Bulb_Basic_C_1@ 82.93 s
Sequence BP_Light_Bulb_Basic_C_1@ 82.93 s
Branch BP_Light_Bulb_Basic_C_1@ 82.93 s
Branch BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Toggle light BP_Light_Bulb_Basic_C_1@ 82.93 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Light flash BP_Light_Bulb_Basic_C_1@ 81.73 s
Update Light Color BP_Light_Bulb_Basic_C_1@ 81.73 s
Set Intensity BP_Light_Bulb_Basic_C_1@ 81.73 s
Set Intensity BP_Light_Bulb_Basic_C_1@ 81.73 s
Set Light Color BP_Light_Bulb_Basic_C_1@ 81.73 s
Set Light Color BP_Light_Bulb_Basic_C_1@ 81.73 s
Set Vector Parameter Value BP_Light_Bulb_Basic_C_1@ 81.73 s

蓝图与Prefab

蓝图与 Prefab

Unity Prefab 工作流

- 创建一系列带有组件的 GameObject
- 基于它们生成 Prefab
- 场景中放置 Prefab 的实例
- 或者在运行时将它们实例化
- Prefab 可以嵌套

Unreal 蓝图类

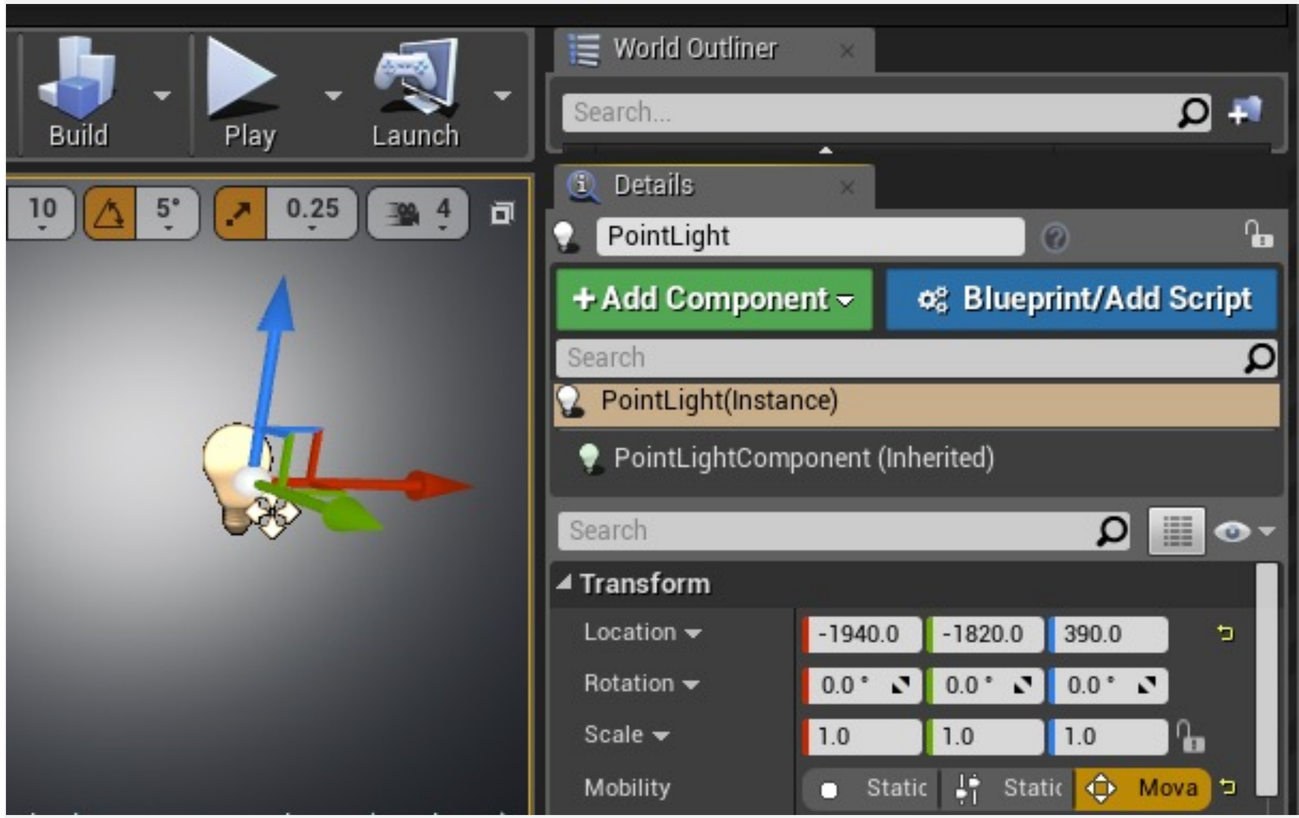
- 创建一个带有组件的 Actor
- 蓝图 / 添加脚本（Blueprint / Add Script）
- 将它们拖拽到任意场景关卡中，创建其实例
- 或使用 Spawn Actor from Class 在运行时进行实例化
- 新的蓝图类继承某个现有的蓝图类，在它的基础上增强
- 可以使用 Child Actor

创建蓝图类：关卡中的Actor

您可以根据关卡中放置的Actor创建蓝图类。

为此，选择关卡中的Actor，然后单击关卡编辑器“细节”（Details）面板中的蓝色按钮“蓝图/添加脚本”（Blueprint/Add Script）。

这个Actor的类将用作新蓝图的父类。



对象属性值

- 默认值
- 实例可编辑值

添加子ACTOR组件

添加子Actor组件（Add Child Actor Component）函数将一个Actor添加为另一个Actor的组件。因此，组件Actor遵循父Actor的变换。当父Actor被销毁时，Actor组件也会被销毁。新Actor的类必须在Add Child Actor Component函数的“细节”（Details）面板中指定。

输入

- 目标（Target）：对将拥有新组件的Actor的引用。
- 手动连接（Manual Attachment）：布尔值。如果值为“false”，则将自动连接新Actor。
- 相对变换（Relative Transform）：新组件将使用的变换。

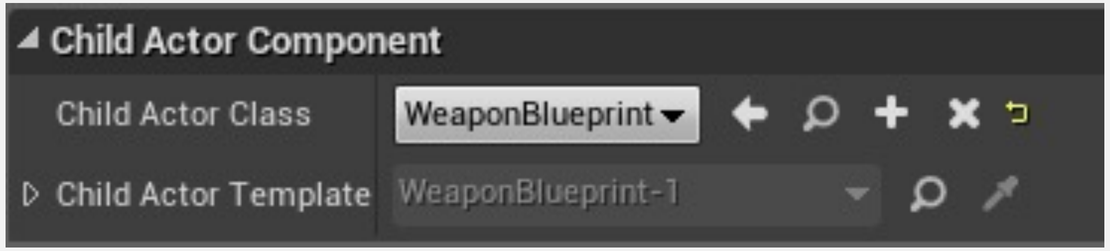
输出

- 返回值（Return Value）：对所创建Actor的引用。

添加子ACTOR组件：示例

在右侧示例中，有一个蓝图“WeaponBlueprint”，表示武器。在另一个表示游戏角色的蓝图中，有一个将在游戏期间调用的事件，以向该角色添加“WeaponBlueprint”类型的武器。

下图来自于Add Child Actor Component函数的“细节”（Details）面板。当选中该函数时会显示这个面板。新Actor将会使用的类必须在“子Actor类”（Child Actor Class）属性字段中指定。

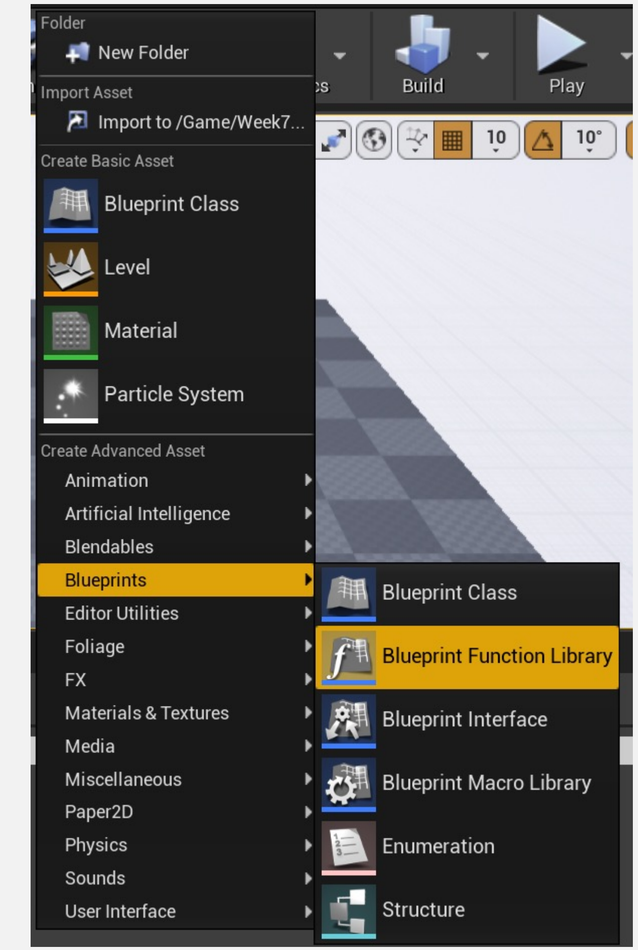


蓝图函数库

Blueprint Function Library

蓝图函数库

- 实现公用函数

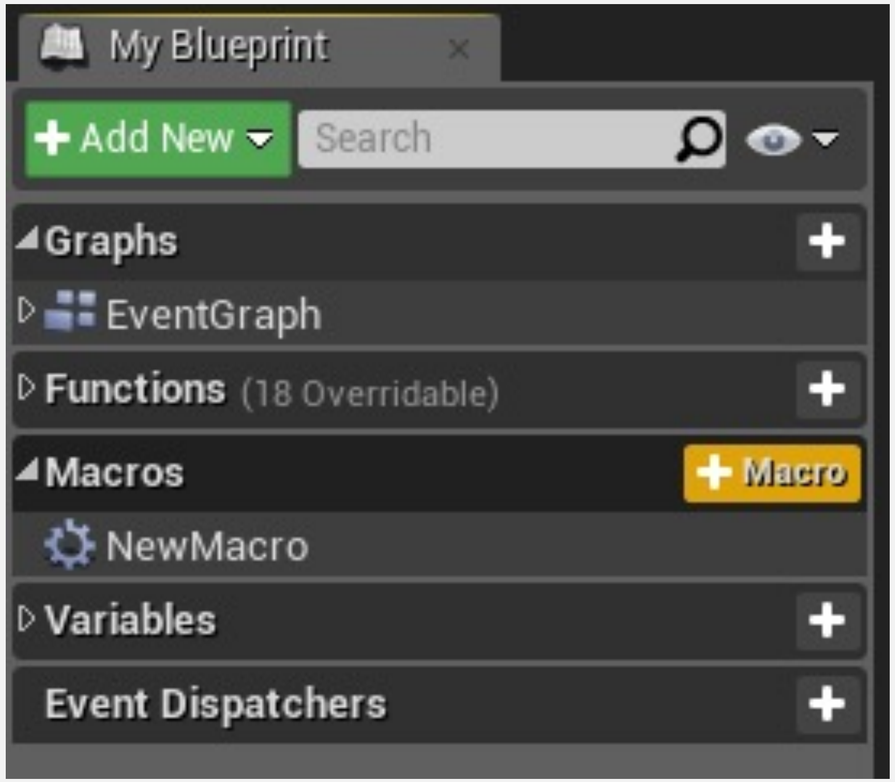


蓝图宏

Blueprint Macros

宏

- 另一种将操作汇集在一个常用位置的方法是使用宏。
- 宏类似于节点的折叠图表。
- 在编译时，宏操作将延伸到使用这个宏的所有位置。
- 宏可以有输入和输出参数以及若干输入和输出引脚。



宏
自定义事件
函数

宏、自定义事件和函数提供了不同的脚本组织方法。它们各有自己的优点和局限性。它们共同的特点是都有输入参数。

- **宏**有输出参数，并可以有多个执行路径。它们不能从另一个蓝图调用。
- **事件**没有输出参数。它们可以从其他蓝图调用，并可以有“委托”引用。它们支持时间轴。
- **函数**可以从其他蓝图调用，并且有输出参数。函数不支持潜在操作，如Delay操作。

Latent Functions

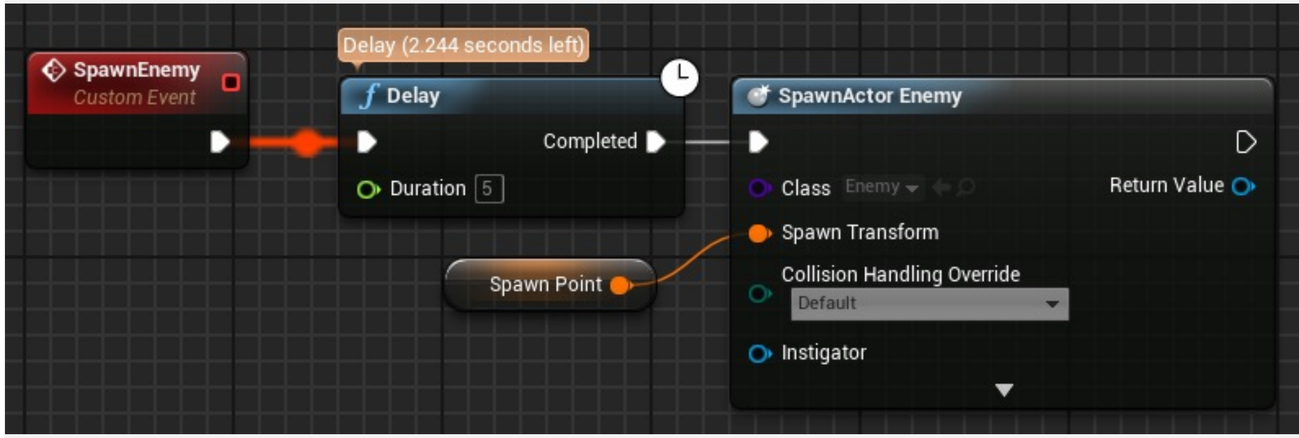
Latent Functions：延迟（DELAY）

延迟（Delay）函数是一个潜在函数，仅在“持续时间”（Duration）参数中指定的时间耗尽后，执行与“完成”（Completed）引脚相连的操作。

潜在函数不遵循蓝图的普通执行流。它们并行运行，并且可能会经历几次tick事件才会完成。

右侧示例显示了名为“产生敌人”（SpawnEnemy）的自定义事件，它使用了一个Delay函数来确保至少经过五秒后才会产生新敌人Actor。即使在五秒内再次调用了SpawnEnemy事件，Delay函数也不会创建新敌人。

在编辑器中运行以查看Delay函数中剩余的时间（见示例）。



Latent Functions：定时器（TIMER）

定时器（Timer）负责在经过指定时间后执行指定函数（或自定义事件）。

有两个函数可以定义Timer：

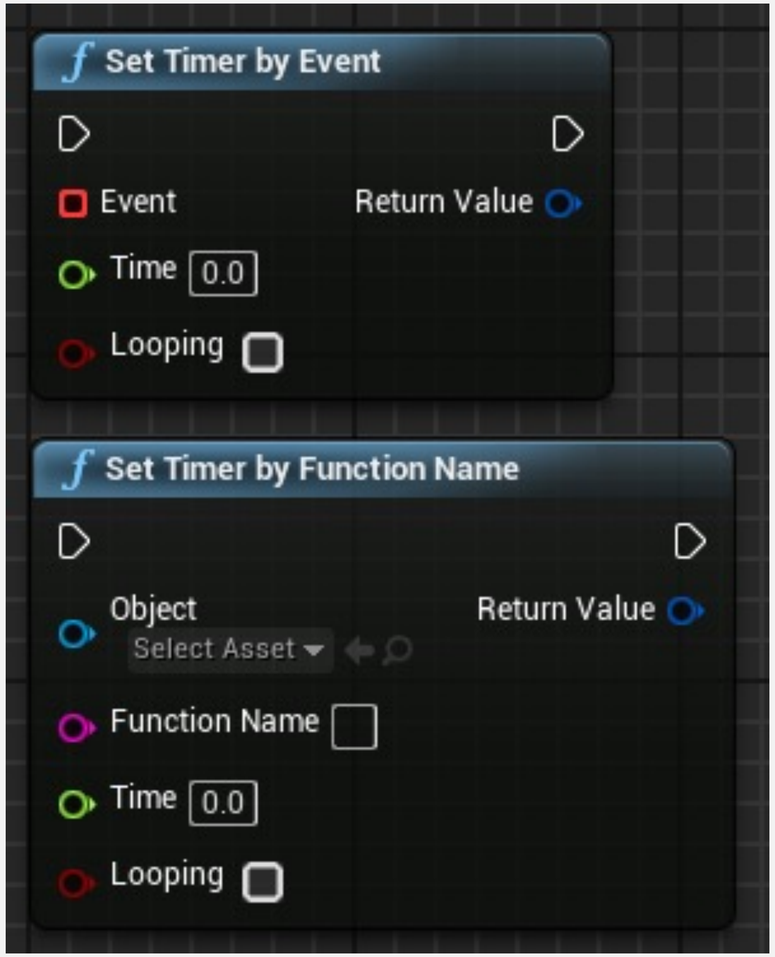
•按事件设置定时器（Set Timer by Event）：以自定义事件引用作为输入参数。

•按函数名称设置定时器（Set Timer by Function Name）：以函数名称和包含该函数的Object作为输入参数。

两个函数都有以下参数：

•时间（Time）：表示Timer的时长，以秒为单位。

•循环（Looping）：指示Timer是继续执行还是仅执行一次。



Latent Functions：可再触发延迟（RETRIGGERABLE DELAY）

可再触发延迟（Retriggerable Delay）函数是一个潜在函数，仅在“持续时间”（Duration）参数中指定的时间耗尽后，执行与“完成”（Completed）引脚相连的操作。

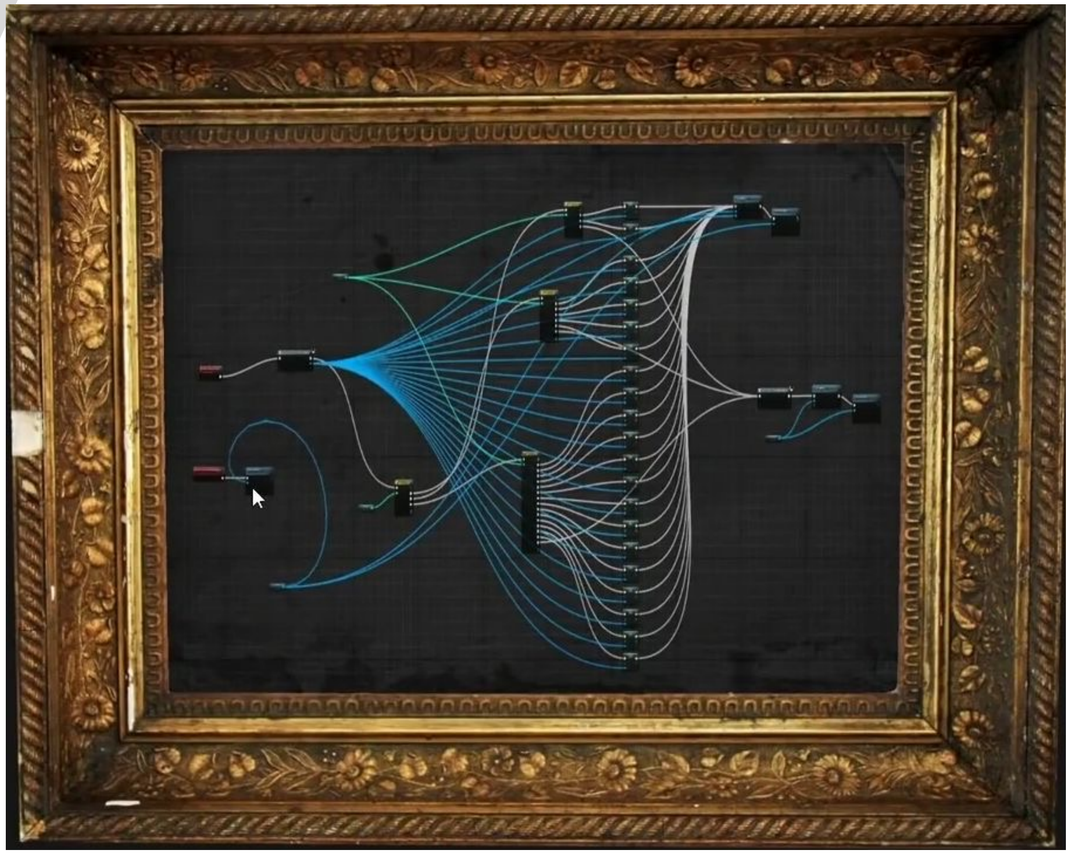
可再触发延迟（Retriggerable Delay）和延迟（Delay）函数的区别是如果再次调用了Retriggerable Delay函数，则Duration参数的倒计时值将清零。

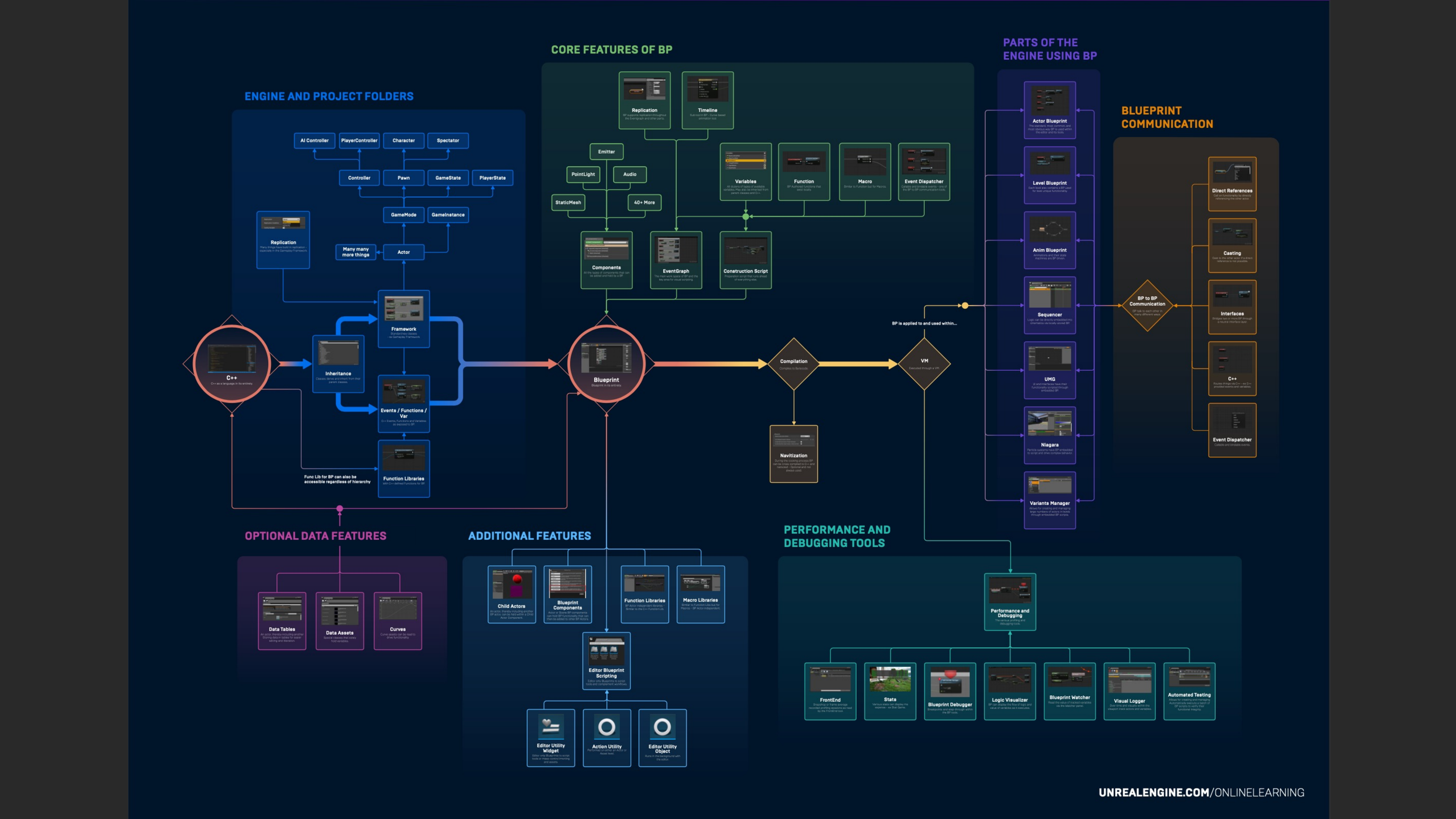
总结一下

Summary

本周内容总结

- 初步了解 Gameplay Framework 的常用类
- 使用蓝图通信机制实现多个对象的组织与协作
- 使用 Timeline 实现对象的动画
- 蓝图结构体等其他蓝图编程元素
- 调试蓝图

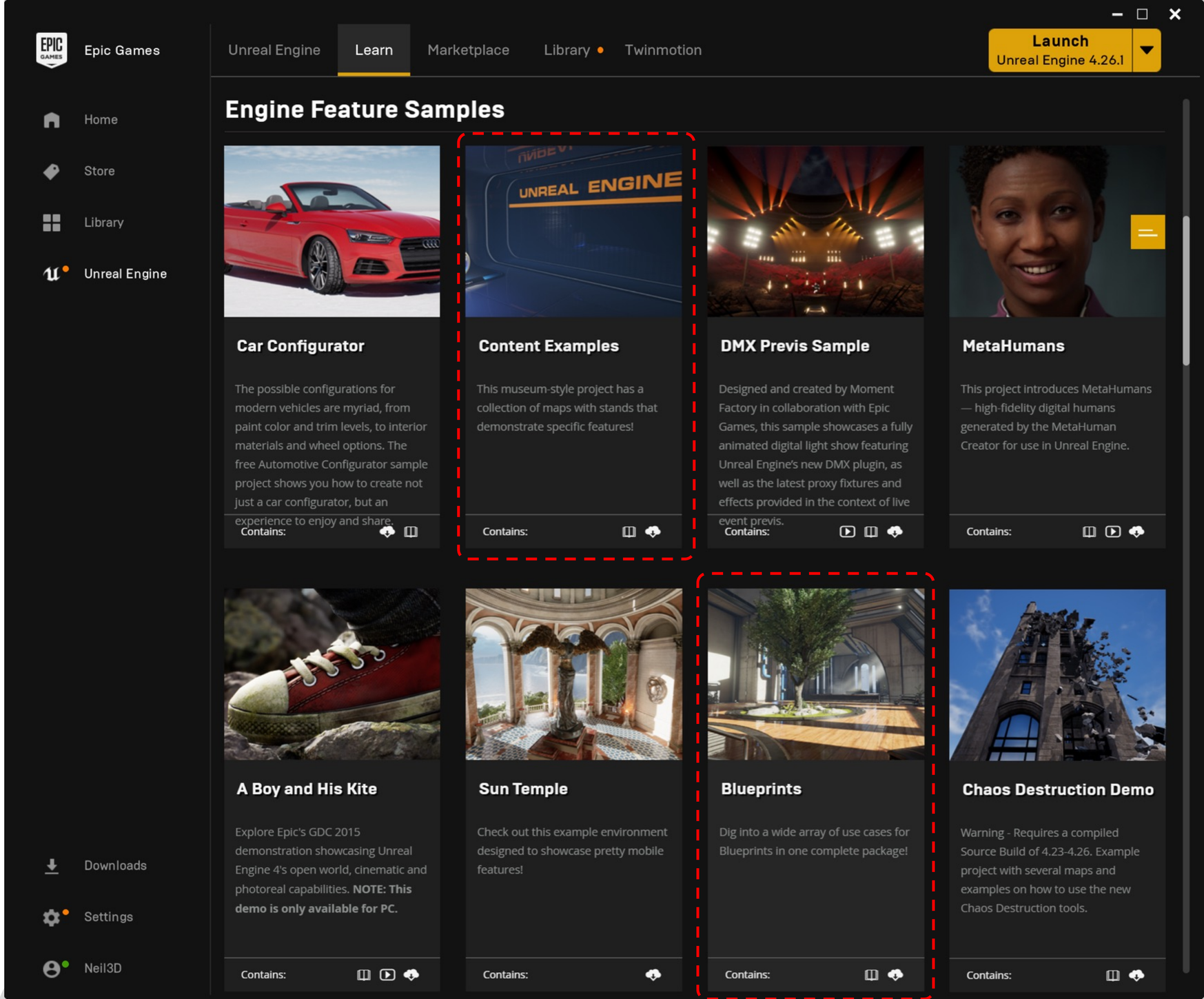



ENGINE AND PROJECT FOLDERS
AI Controller
PlayerController
Character
Spectator
Controller
Pawn
GameState
PlayerState
GameMode
GameInstance
Actor
Many many more things
Replication
Many things have built in replication - especially in the Gameplay Framework.
C++
C++ as a language in its entirety
Inheritance
Classes, structs, and more from their parent classes.
Framework
Boilerplate classes in Gameplay Framework
Events / Functions / Var
C++ Events, Functions and Variables as exposed to BP
Function Libraries
With C++ defined Functions for BP
Func Lib for BP can also be accessible regardless of hierarchy
CORE FEATURES OF BP
Replication
BP supports replication throughout the Eventgraph and other parts.
Timeline
Sub-node in BP - curve based animation tool
Emitter
PointLight
Audio
StaticMesh
40+ More
Variables
All types of hidden or available variables they are by default from parent classes and C++
Function
BP-Authored functions that exist locally
Macro
Similar to Function but for Macros.
Event Dispatcher
Caters and Process events - one of the BP to BP communication tools.
Components
All the types of components that can be added and/or to a BP
EventGraph
The main event space of BP and the key area for visual scripting
Construction Script
Preparation script that runs ahead of everything else.
Blueprint
Blueprint in its entirety
Compilation
Compiles to Bytecode
Navitization
During the cooking process BP can be mixed compiled to C++ and hardcoded - Optional and not always used.
BP is applied to and used within...
VM
Executed through a VM.
PARTS OF THE ENGINE USING BP
Actor Blueprint
The standard, most common, and most obvious way BP is used within the editor and its tools.
Level Blueprint
Each level also contains a BP used for level unique functionality
Anim Blueprint
Animations and their state machines are BP driven.
Sequencer
Logic can be directly embedded into cinematics via locally scoped BP.
UMG
UI and inter faces have their functionality scripted through embedded BP.
Niagara
Particle systems have BP embedded in script and drive complex behavior
Variants Manager
Allows for creating and managing large numbers of actors in levels through embedded BP scripts.
BLUEPRINT COMMUNICATION
BP to BP Communication
BP talk to each other in many different ways.
Direct References
Cast or hard-code to by directly referencing the other actor.
Casting
Case to the other actor if a direct reference is not possible.
Interfaces
Bridges two or more BP through a neutral interface layer.
C++
Routes things via C++ - ex C++ provided events and variables.
Event Dispatcher
Collects and broadcasts events.
OPTIONAL DATA FEATURES
Data Tables
An actor, thereby including another Blueprint, data in tables for easier editing and iteration.
Data Assets
Special classes that is also hold variables.
Curves
Curve assets can be read to drive functionality.
ADDITIONAL FEATURES
Child Actors
An actor, thereby including another BP actor, can be held within a Child Actor Component.
Blueprint Components
Actor or Scene BP components can host BP functionality that can then be added to other BP Actors.
Function Libraries
BP Actor independent libraries - Similar to the C++ FunctionLib.
Macro Libraries
Similar to Function Libs but for Macros - BP Actor independent.
Editor Blueprint Scripting
Editor only Blueprints to script tools and component workflows.
Editor Utility Widget
Editor only Blueprints to script tools or mass-control importing and assets.
Action Utility
Performing actions on the Asset or Asset level.
Editor Utility Object
Runs in the background with the editor.
PERFORMANCE AND DEBUGGING TOOLS
Performance and Debugging
The various profiling and debugging tools
FrontEnd
Snapshots of frame, prestige recordings, and more can be read by the FrontEnd tool.
Stats
Various stats can display info anywhere - ex Stat Game.
Blueprint Debugger
Breakpoints and step through within the BP tools.
Logic Visualizer
BP can display the flow of logic and value of variables as it executes.
Blueprint Watcher
Read the value of a single node via the watcher panel.
Visual Logger
Over time and visually within the viewport mark actors and variables.
Automated Testing
Allows for creating and managing. Automatically executes a batch of BP scripts to verify their functional integrity.

学习资料推荐

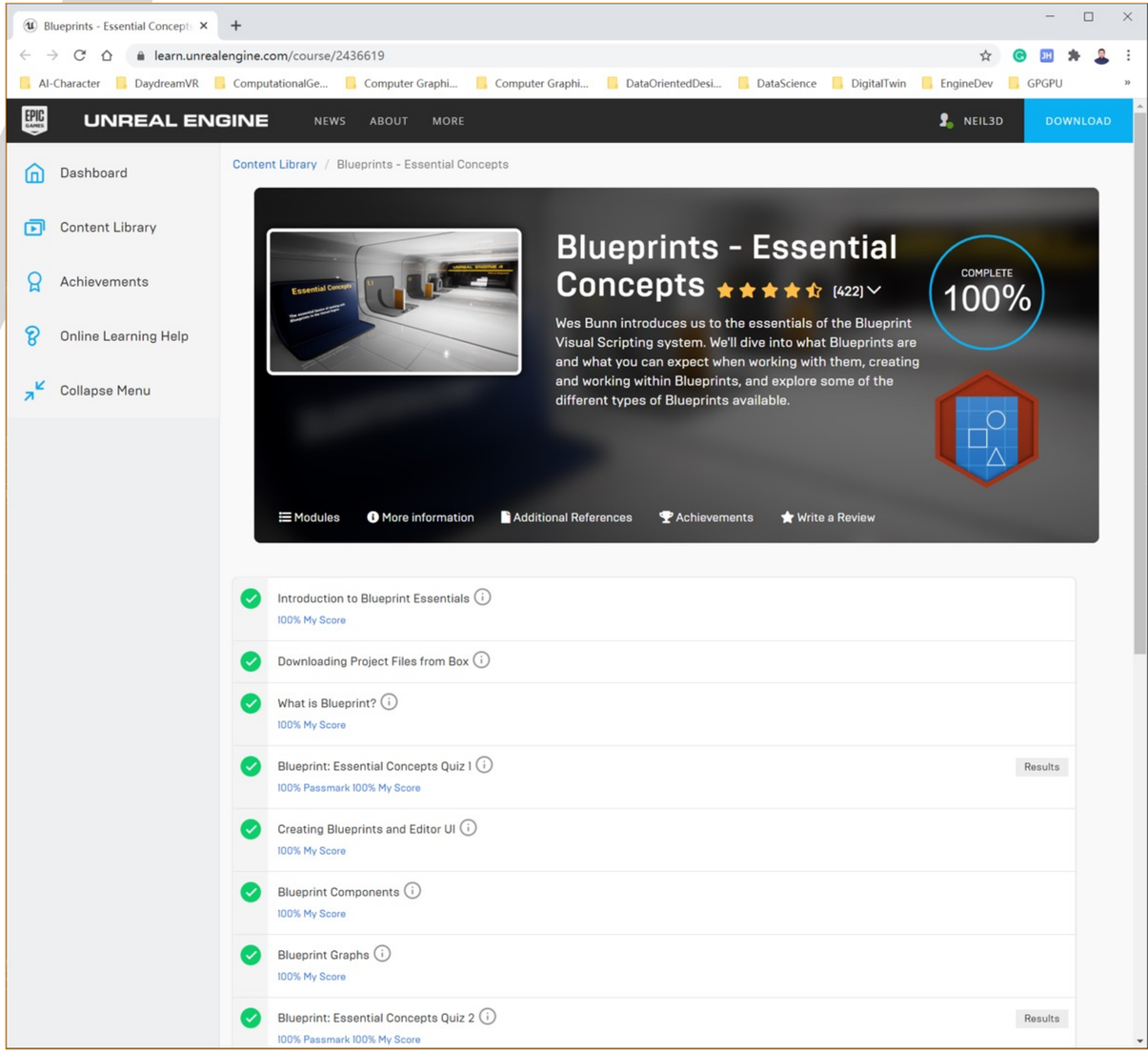
引擎示例工程

- Content Examples
- Blueprints



学习资料推荐

- Epic 官方视频教程: Blueprint Kickstart
  https://learn.unrealengine.com/course/3537777
- Epic 官方视频教程: Blueprints - Essential Concepts
  https://learn.unrealengine.com/course/2436619
- Epic 官方文档：Blueprint Visual Scripting
  https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html



学习资料推荐

知乎专栏：

https://www.zhihu.com/column/blueprints-in-depth

使用 **C++** 扩展蓝图

- 通过 CustomThunk 实现参数类型通配符
- 通过派生 UK2Node 进行深度定制

谢 谢 大 家 !

Questions?