【GDC2025】【双影奇境】使用Capabilities编写所有玩法代码

【GDC2025】【双影奇境】使用Capabilities编写所有玩法代码

小木子
IT宅男/篮球爱好者/一级奶爸
关注他

航海家 SuperSodaSea 等 721 人赞同 ›

前言

在之前的《双影奇境》代码解析01-Tutorial中我们提到了游戏内几乎所有Gameplay都是围绕Capabilities开发的，Capabilities是什么，为什么采用Capabilities模式来开发Gameplay而不是虚幻原生的Actor-Component模式，Capabilities有什么优势和缺点，应该如何按照Capabilities模式编写Gameplay代码，相信阅读完本篇你能从中找到答案；

本篇几乎不用任何前置知识，内容简单易懂，这在大家都在搞AI的GDC大会上显得特别另类和难得，而且这次演讲对独立开发者极具参考价值，因为它讲的就是一种能高效产出的Gameplay编程模式；

正文

我叫Ylva，是Hazelight工作室的高级玩法程序员，本次演讲我将向你们介绍Capabilities代码架构，这样你们在解决问题时就能将它加入你们的工具箱中。如果这个模式适合你们的项目，你将能够创造出丰富的游戏玩法。

这是本次演讲的概览：

Overview

我们将从广义概念上了解什么是capability pattern，为什么要使用它（包括收益和代价），如何实现Capabilities（包含细节和伪代码），以及它们在什么地方使用（《双影奇境》中的实际玩法示例）。

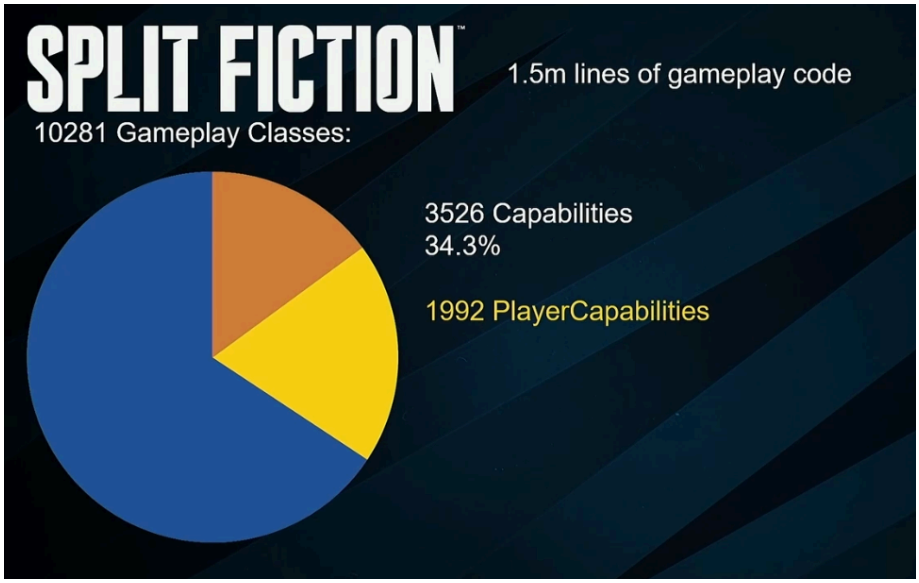
本次演讲结束后，我们将能够：

1. 了解Capabilities的益处和代价
2. 区分Capabilities与其他结构（如ECS⁺）的不同
3. 了解至少五个Capabilities适用的游戏玩法领域
4. 编写Capabilities的伪代码

TalkObjectives

《双影奇境》是一款双人合作冒险游戏，而Capabilities使我们能够创造如此多的玩法，就像Hazelight之前的作品一样，《双影奇境》拥有丰富的多样性，为了在不剧透太多的情况下证明这一点，我将播放一个简短的预告片。

在《双影奇境》中，我们有150万行Gameplay代码，我们有10000个玩法类其中大约三分之一是Capabilities，而在所有Capabilities中超过一半是所谓的“Player Capabilities”即玩家特有的行为，这些类仅在玩家角色上有意义。



AllCapabilities

我想强调的是，Capabilities作为一个概念并不局限于Hazelight所使用的这种方式。我们使用虚幻引擎+和Angelscript，我们的C++代码中没有特定玩法相关的东西，蓝图中也极少。我们所有的Capabilities都驻留在Angelscript中，但本次演讲的重点不是Angelscript。当然，你可以在你的工程/脚本语言中使用Capabilities。

What-Concept(是什么)

让我们看看Capabilities的广义概念。我们使用虚幻引擎，而虚幻引擎采用的是游戏对象-组件（GameObject-Component）结构，许多现代游戏引擎使用流行的实体-组件-系统（Entity-Component-System, ECS）结构，Entity代表EntityID，Component仅包含数据和工具函数，而System则负责所有行为，Capabilities可以被视为ECS中System的一个远房表亲，因为两者都负责行为，但Capabilities是用于GameObject-Component结构中的。顺带一提，如果你不熟悉ECS，2017年GDC有一个很棒的演讲，Tim Ford主讲的《守望先锋：游戏玩法架构与网络代码》（Overwatch Gameplay Architecture and Netcode）。

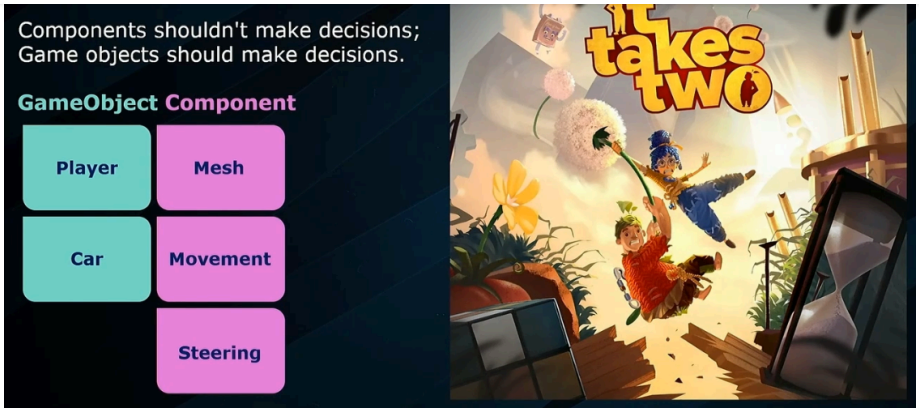
GameObjectComponentCapability

尽管Capabilities和System都负责行为，但它们在粒度和理念上存在根本差异。

这里有一段《双影奇境》的视频，玩家正在用弓射箭，在右侧你可以看到Capabilities。在我们的调试工具中它们激活时会变成绿色，我们将行为解耦为：瞄准、动画、蓄力、绘制弹道UI和射击。因此与ECS系统和其他行为解决方案相比，Capabilities的粒度非常细。值得注意的是，即使是像射击这样类似Event的行为，我们也使用Capabilities。

在讨论细节之前让我们谈谈我们是如何做出这个决定的。

故事始于Hazelight首款游戏《逃出生天》✦（A Way Out）的总结会议。会上决定无论下一款游戏是什么其结构都需要改变。在《逃出生天》中不同的Components和Blueprints试图修改相同的状态和代码导致了许多问题，这容易产生Bug且混乱，尤其是在联网时，因此对于《双人成行》我们希望将行为从Components中移出。于是诞生了一个理念：关于Compnents与GameObjects决策的理念。Compnents不应做决策，GameObjects才应该做决策。



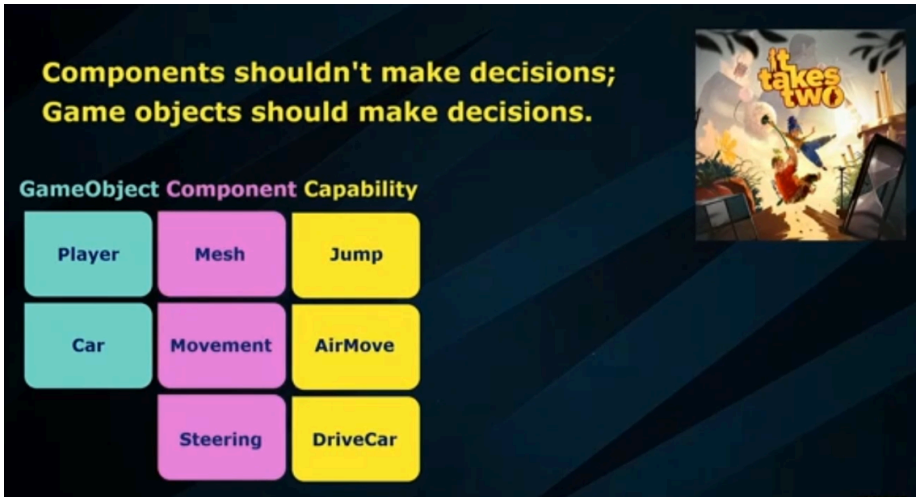
Background

事实上，这非常重要，请允许我重复一遍：

Components shouldn't make decisions, game objects should make decisions

如果我们不允许Components包含行为，所有这些行为最终都会跑到GameObjects里，接下来的问题是：我们如何避免游戏对象变成臃肿类？当一个类包含的行为远远超出其职责范围时，就会产生臃肿类。例如在游戏开发实践中，你可能也做过类似的事：把所有逻辑都塞进玩家类导致有成千上万行代码，一切都非常混乱。我们的解决方案是将GameObjects的行为扩展到Capabilities中。

然而如果GameObjects的行为很简单，那就完全没问题，把所有逻辑都保留在游戏对象类中(KISS原则)。



GameObject-Component-Capability

我们的Components类似于ECS中的Components，也就是说它们包含数据和工具函数，它们包含你可以读写的公共数据，但是为了避免意大利面条式代码，**我们的Capabilities是完全无法被外部访问的，它们甚至不能互相访问**，那么如果Capabilities是解耦的并且无法从任何地方访问，射击Capabilities如何知道弓已经蓄力足够从而允许射击呢？

答案是：**当Capabilities需要彼此通信时，它们通过Components来实现。**

在这个例子中，弓蓄力Capability将一个蓄力值写入BowComponent，而弓射击Capability则读取那个蓄力值来检查：我们是否蓄力足够以允许射击，还有其他Capabilities也对读取这个蓄力值感兴趣，比如动画和弹道UI，它们也检查蓄力值来决定行为。有些Capabilities对不止一个Component感兴趣，瞄准、蓄力和射击Capabilities都对InputComponent感兴趣，因此Components和Capabilities的数量之间没有严格的对应关系。

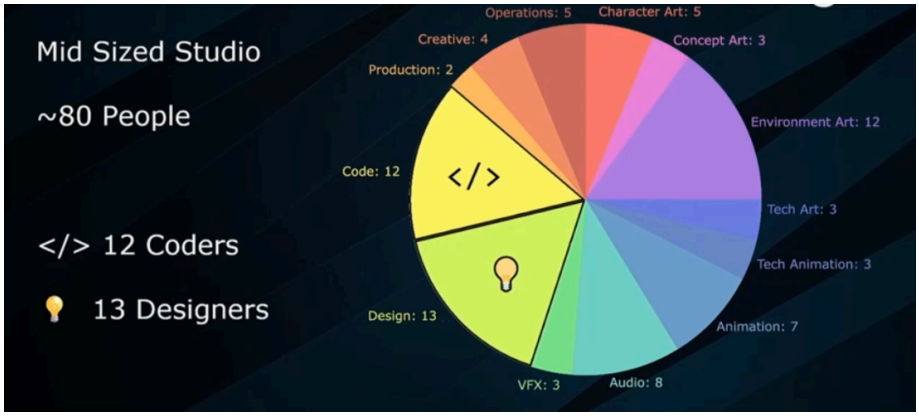
CommunicationViaComponents

这里有一个更复杂的例子，我们有12个与玩家爬梯子相关的Capabilities，再次强调你可以看到每个类都非常细粒度，可以看到通用的爬梯Capability在整个攀爬过程中都是激活的，它不关心我们是从地面、顶部还是空中进入攀爬的，在我们的开发菜单中，很容易精确追踪是哪个Capability触发了攀爬。

Why-Benefits&Costs(为什么选择这种方案-优势和代价)

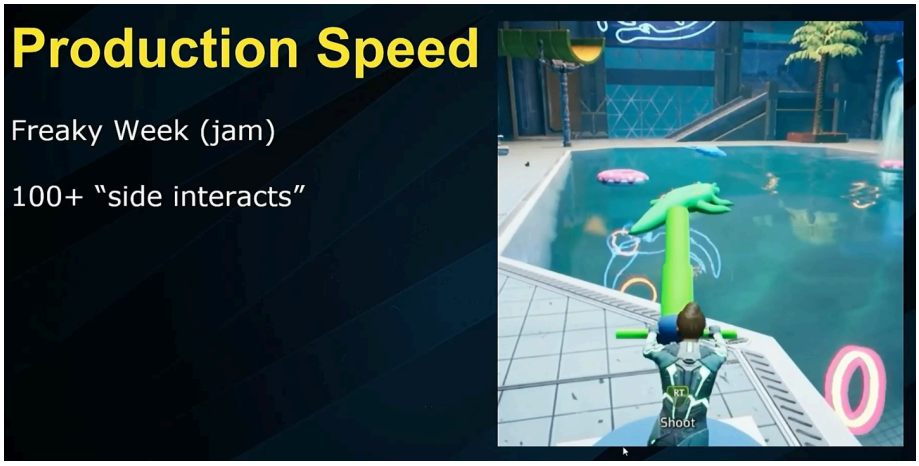
为什么这是个好主意，它的优势是什么，GameObjects-Components结构非常面向对象，而这种面向对象结构通常对设计师和许多玩法程序员来说，比ECS更直观，这种直观性带来了**生产效率**。

提供一些背景：Hazelight有80名员工，我们有13名设计师和12名程序员，其中10名是玩法程序员。脚本编写即使对有才华的设计师来说也可能令人生畏，但由于Capabilities如此简单和细粒度，它实际上成为了Hazelight设计师变得非常技术化的绝佳入门途径，拥有能够自主设计功能的自主设计师，也解放了我们玩法程序员去编写更多代码，我认为这真的很棒。我们的设计师甚至能够修复他们自己遇到的大部分Network Bugs。我不会指望设计师去写ECS系统，但对于Capabilities来说，它们真的很容易管理。事实上，我的一些设计师曾问我：“嘿，你能把这个Capabilities模式添加到我的业余项目里吗？”所以我们的设计师真的很喜欢Capabilities。



ProductionSpeed

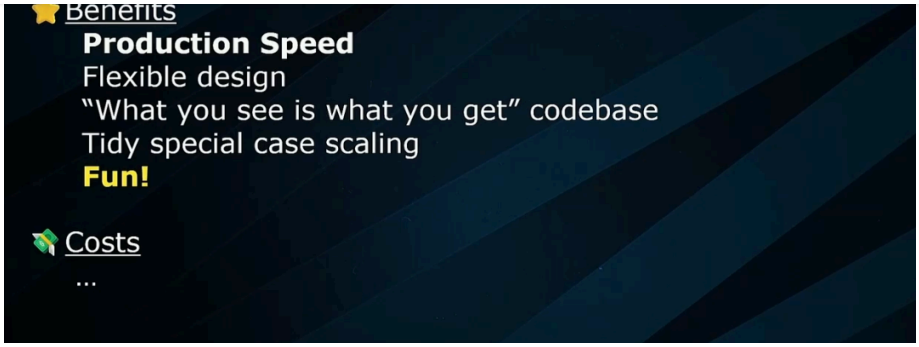
为了强调Hazelight如何从中受益，我们有时会举办所谓的“疯狂周”，这就像一个GameJam，我们可以自由发挥，创造新关卡或游戏机制。大约一年前，我们举办了一个专注于制作“Side-Interacts+”的疯狂周，Side-Interacts是玩家在环境中可以做的有趣小事情，让游戏世界感觉更有生机活力。这其实不到一周更像是三天，我们制作了超过一百个环境交互，直接放入现有关卡，没有太多稳定性问题，其中大部分都被保留到了最终游戏中。因此这种直观的结构使我们具有自主性，它激发了我们的创造力并且设计非常灵活，因为改变起来很容易。这种生产方式创造了多样化的游戏玩法。这也许有点个人化但我只是觉得作为玩法程序员，Capabilities是目前创造玩法最有趣的方式，在创造新玩法时几乎没有什么阻力，从代码规范的角度来看，优势在于行为本身非常细粒度和内聚，并且很容易添加任意数量的特殊情况处理。这些类本身也非常符合“所见即所得”的原则，所以从代码角度也绝对有好处。



SideInteracts

现在让我们停下来回顾一下Capabilities的优势：**生产效率高，对设计变更灵活，代码所见即所得，整洁地处理特殊情况扩展，并且真的很有趣，易于发挥创造力**

让我明确一点，Capabilities模式也有代价，但这些代价在看伪代码时会更加直观。所以，我们将先深入细节，然后再讨论代价。

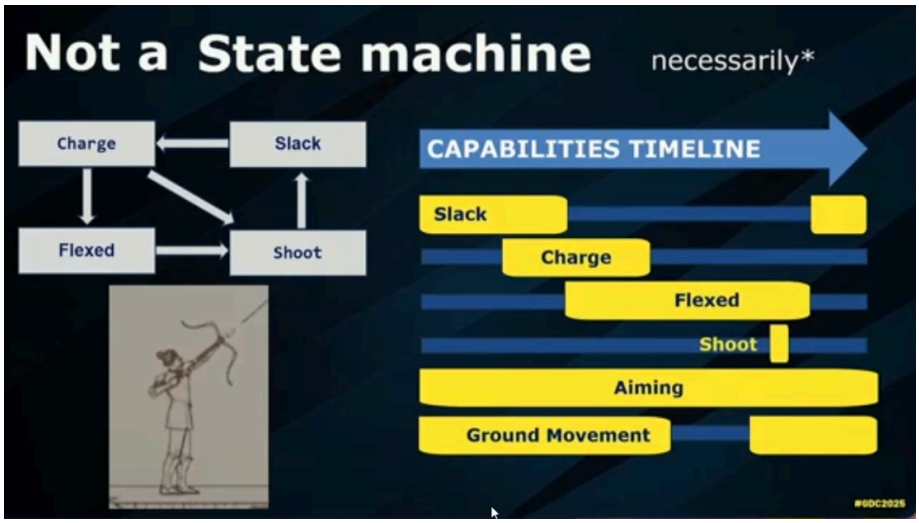


Benefits

How-Details&Pseudocode(如何实现 细节＆伪代码)

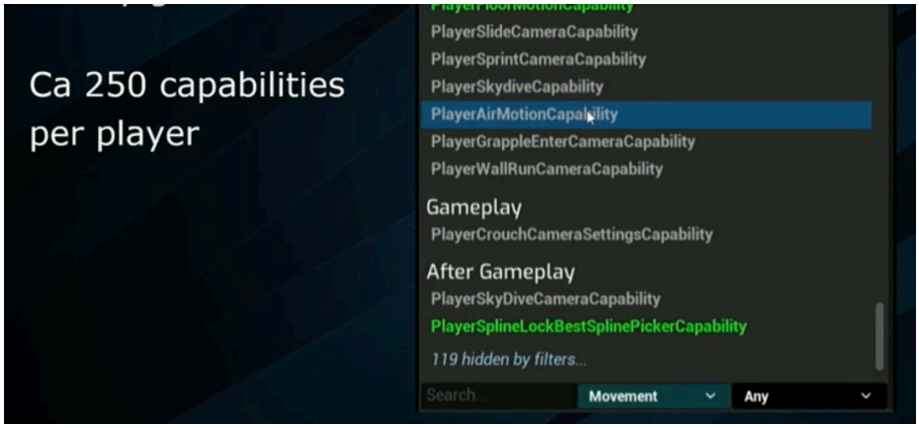
让我们深入细节，Hazelight的团队试图给出一个定义：什么是Capability？Capability的本质是什么？我们得出了这句话：**Capability是一个单一的功能单元，它包含自身所有的状态和决策逻辑(a capability is a single unit of functionality that contains all of its own state and decision making)**

作为一个广义概念，我通常将其描述为一种状态。但当我对其他程序员这么说时，他们会告诉我：如果Capabilities是状态，那这不就是一个状态机吗？让我来解惑一下。状态机设计模式像流程图一样运作，检查特定状态之间的Transitions。你可以让Capabilities像状态机一样工作，但你并不必须这样做，因为Capabilities可以**并行激活（Active in Parallel）**，它们没有链接关系，并且其中一些状态实际重叠（Overlap）是有意义的，事实上，GameObject（这里指玩家）上甚至有很多的Capabilities在运行，比如瞄准和地面移动。



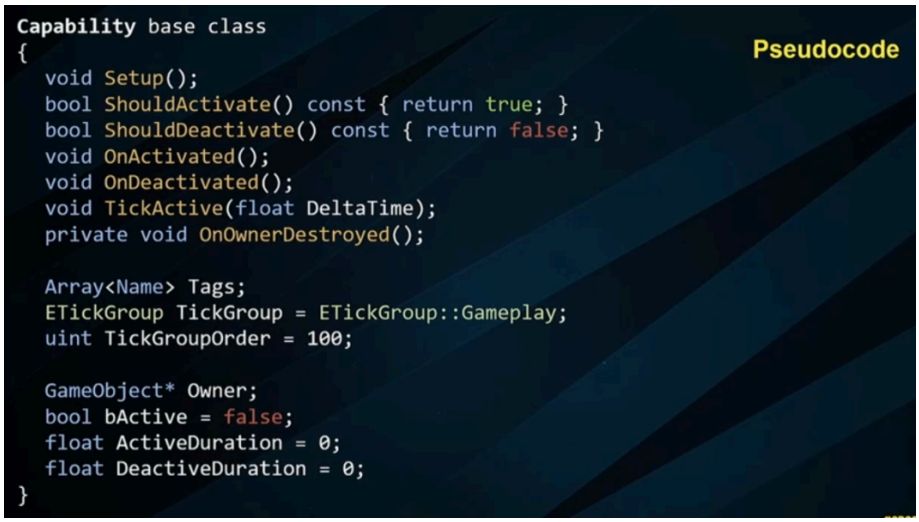
NotStateMachine

所以在游戏中的任何时刻，我们每个玩家身上大约有250个Capabilities。我快速滚动一下我们的开发菜单，这里我们列出了玩家身上的所有Capabilities，玩家身上大约一半的Capabilities与移动相关。其中一些也是当前关卡特有的Capabilities，比如弓，Capabilities按帧更新顺序优先级（Tick Order Priority）列出，我们设定帧更新顺序是为了避免更新顺序问题（Update Order Issues）(注：视频里看到TickOrder的顺序依次为：SeparatedTickOrder, Input, BeforeMovement, InfluenceMovement, ActionMovement, Movement, LastMovement, BeforeGameplay, Gameplay, AfterGameplay, AfterPhysics, Audio, PostWork, LastDemotable)，例如我们的AirMovement有一个非常晚的帧更新顺序，它可以被视为最后的保底以防没有更早的Capabilities执行了移动。



PlayerCapabilities

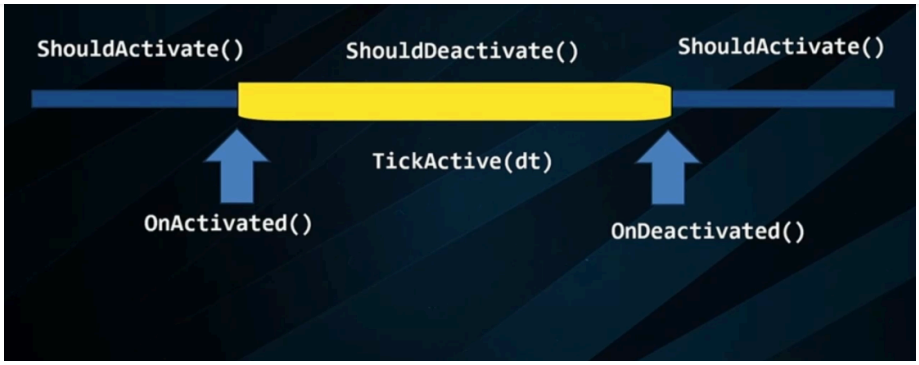
这是Capability基类，它有六个你可以重写（Override）的函数： `Setup`， `ShouldActivate`， `ShouldDeactivate`， `OnActivated`， `OnDeactivated`， `TickActive`，你还可以看到我之前提到的帧更新组（Tick Group）和顺序（Order），在底部有一些实例定义的成员变量，中间还有标签（Tags），但现在先不用担心Tags，稍后会讨论它们。



CapabilityBaseClass

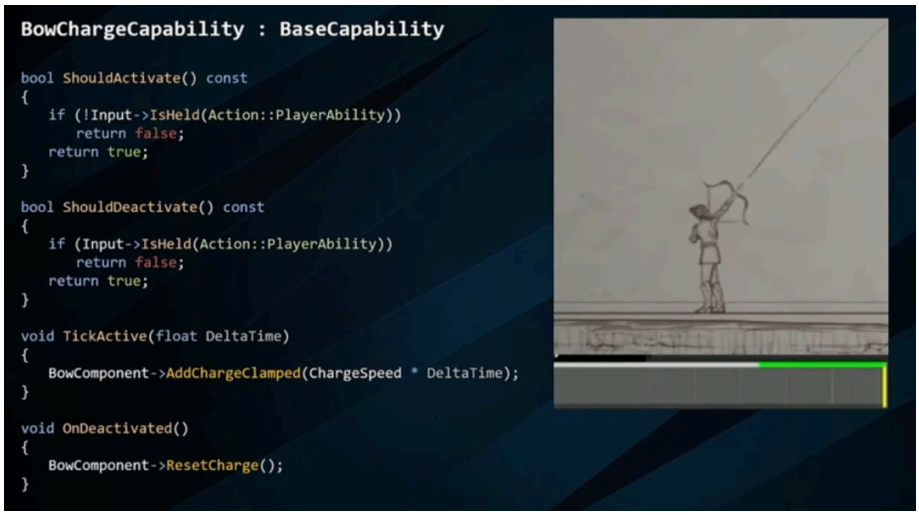
Capabilities函数的调用流程如下：

1. 当GameObject Spawned时，我们在该GameObject上的Capabilities上运行 `Setup`，通常用于初始化本地成员变量，比如从GameObject上获取Component的引用/指针
2. 当某个Capability处于非激活状态（Inactive）时，我们**每帧检查** `ShouldActivate`
3. 在某个时刻， `ShouldActivate` 返回 `true`，这个Capability变为激活状态（Active），我们运行 `OnActivated`
4. 当这个Capability处于激活状态时，我们改为**每帧检查** `ShouldDeactivate`，并且**每帧运行** `TickActive`
5. 当 `ShouldDeactivate` 返回 `true` 时，我们运行 `OnDeactivated`，然后回到**每帧检查** `ShouldActivate`



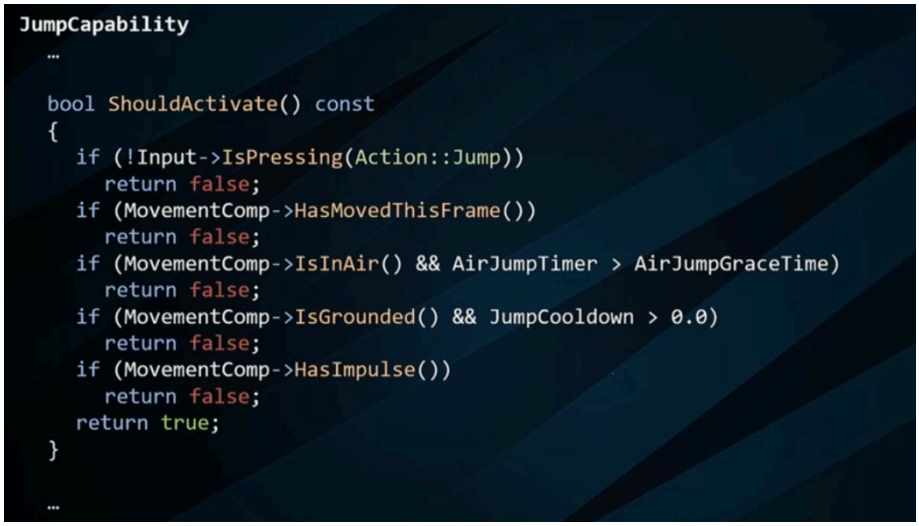
CapabilitiesTimeline

这是一个弓蓄力Capability的示例。它是上面基类的一个子类，在右侧你可以看到时间线，当Capability激活时它变成绿色，`ShouldActivate` 检查玩家输入；在 `TickActive` 中我们增加弓组件上的蓄力值；当我们释放输入时，Capability失活（Deactivates）并重置蓄力值。



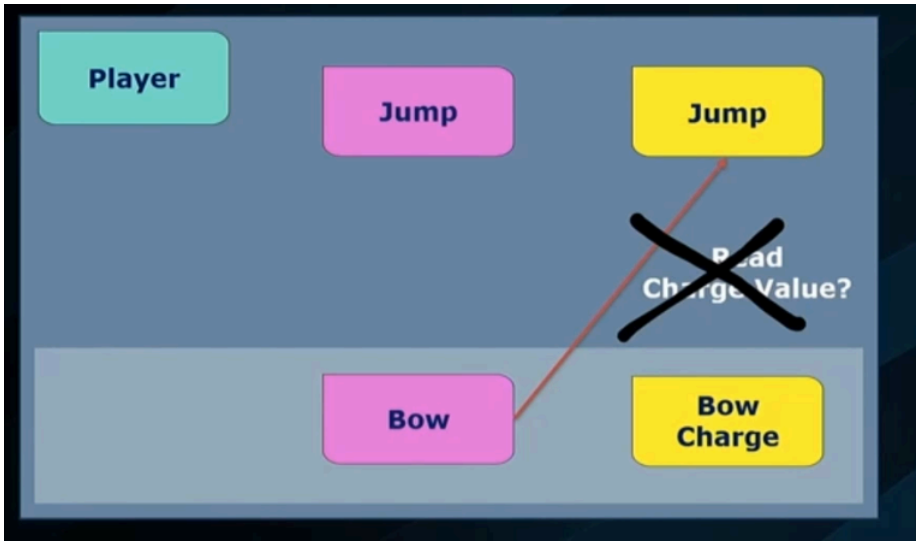
BowChargeCapability

在这些函数里只有单行代码就提前返回可能感觉有点傻。在这种代码标准下当你有大量检查时就会更有意义。这是《双影奇境》中JumpCapability，我们在这里设置了许多条件并且每个返回都在单独一行，所以如果我们的Capabilities没有激活（例如我们按了跳跃键但没反应），我们可以轻松地在所有这些 `return` 语句上设置断点看看哪里失败了（“为什么我的Capabilities不激活？”）



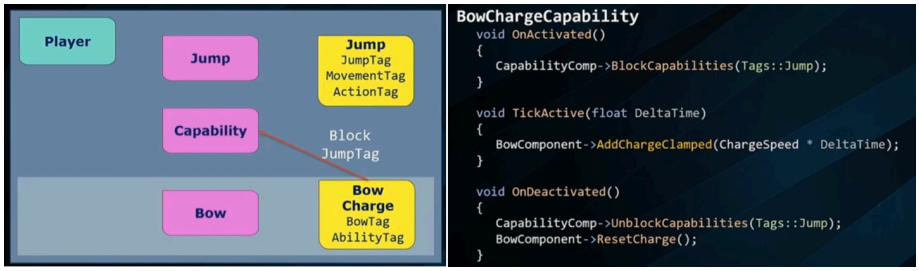
JumpCapability

功能，在这种情况下我们需要将游戏中所有禁止跳跃的特殊情况都添加到实际的JumpCapability的 `ShouldActivate` 检查中，我们不希望因为这些特殊情况把 `ShouldActivate` 函数搞得一团糟，那样会很快变得不可读、混乱并产生奇怪的依赖，所以我们不想那样做。



StupidMethod

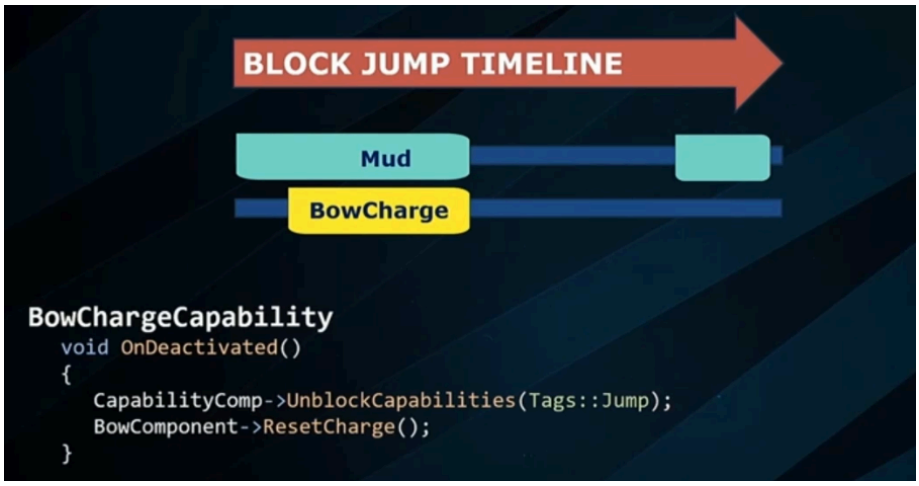
我们的解决方案是在Capability上添加标签（Tags），你可以将其视为枚举（Enum）、字符串（String）或其他ID，一个Capability可以拥有**任意数量**的Tags。紧接着我们介绍下CapabilityComponent，这个组件负责一些记录工作，我们现在可以做的是：弓蓄力Capability告诉CapabilityComponent：“请阻止（Block）所有带有JumpTag的Capabilities”，因此除了弓蓄力Capability已经在做的（激活时增加蓄力，失活时重置蓄力），我们还想在激活时Block the JumpTag，在失活时Unblock the JumpTag。



BlockCapabilities

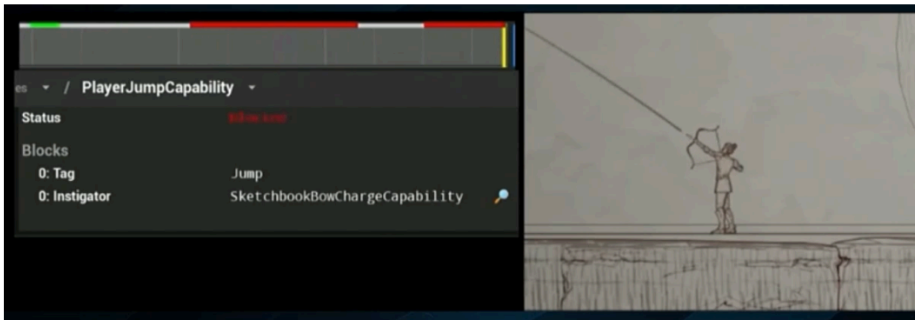
视频里可以看到JumpCapability在时间线上的情况，我们可以看到它激活时变绿，当我们开始弓蓄力时，它变红因为它被Block了，所以现在玩家在弓
本身是完全解耦的。

为Capability关活的无论玩家是否站在泥潭里它都会Unblocks the JumpTag。这是一个Bug，我们的愿望是让跳跃阻塞（Jump Blocks）彼此独立。我们的解决方案是不仅跟踪哪个Tag被阻塞，还要跟踪是谁（Who）发起了（Instigated）每个阻塞。



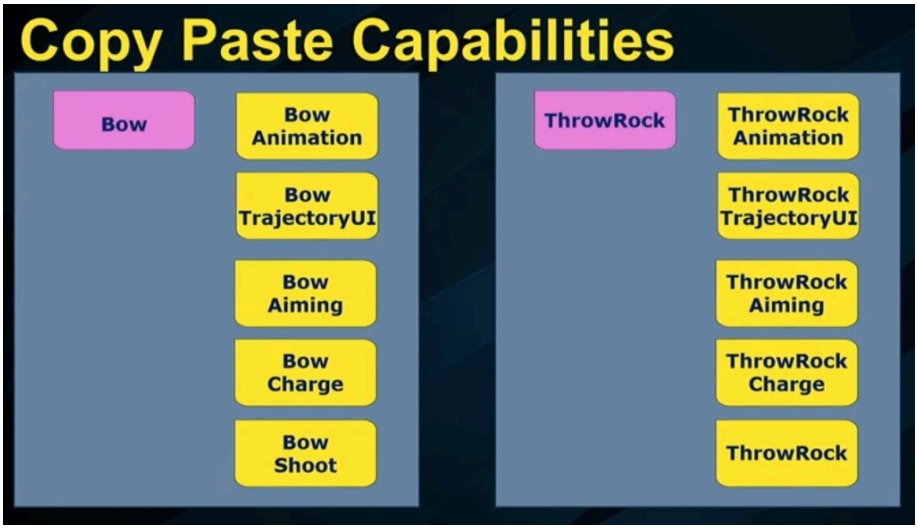
BlockJumpTimeline

因此在我们的弓蓄力Capability中，阻塞（Block）和解除阻塞（Unblock）函数也接收一个参数，指明是谁（Instigator）请求这个阻塞，所以当我们解除阻塞（Unblock）时，我们只移除我们这个发起者（Instigator）的阻塞，来自其他发起者（如泥潭）的阻塞不受影响。顺便提一下，我们有很多不同的工具可以将各种东西转换为发起者（Instigators），GameObject可以是发起者，Capability可以是发起者，字符串和枚举也可以是发起者，这只是我们追踪请求来源的一种方式，在我们的开发菜单中，我们显示每个阻塞背后的发起者，在这里我们可以看到跳跃被蓄力Capability阻塞了。因此发起者（Instigators）是Capabilities相互影响同时保持解耦的一个非常好的方式。



Instigators

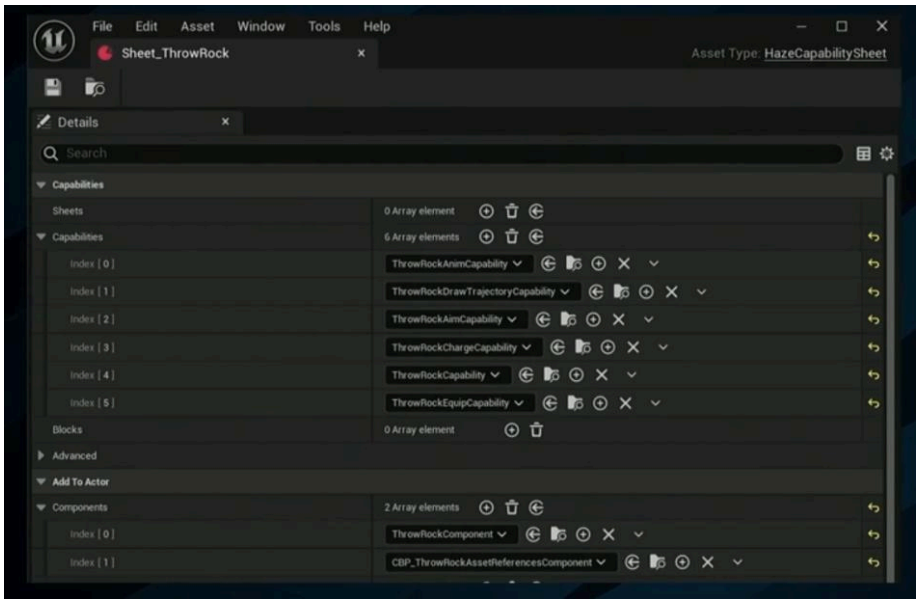
在Hazelight，我们还有一个秘密，关于我们如何能够创造数百个游戏玩法，假设我们想给游戏添加一个新功能：让玩家拥有投掷石块的能力，射击弓箭和投掷石块有点类似，不是吗？所以我们只需复制并重命名那些类，我们的秘密是：我们**大量复制粘贴代码**。事实上我们更倾向于复制粘贴而不是拥有一个通用的投射物系统或通用的投射物Capability，因为我们的玩法行为差异很大，让它们独立实际上会更清晰。这就是你最终拥有超过10000个玩法类的方式，这对于我们这类每个情况都是特殊情况的游戏来说效果很好，如果我们想改变投掷石块的工作方式，我们可以确信弓箭相关的东西不会出问题，此时它们完全不相关了。



CopyPasteCapabilities

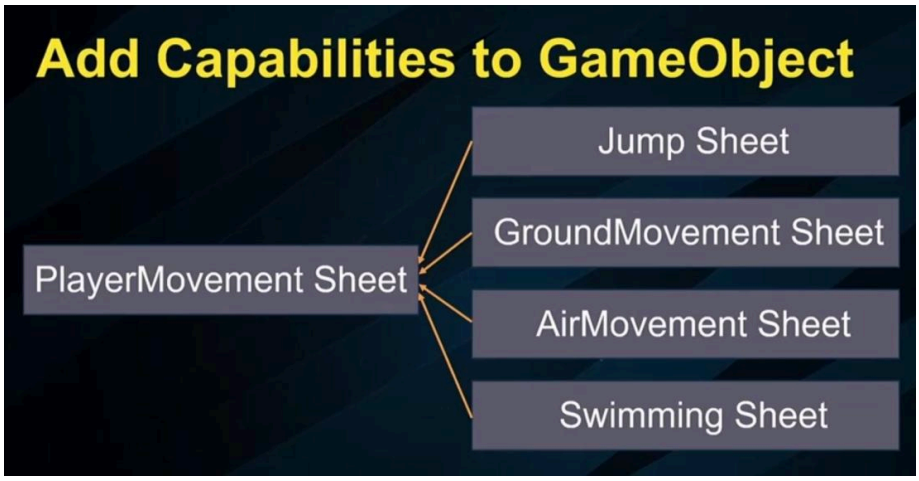
我们有了新能力——投掷石块，我们想把它添加到玩家身上能相关的Capability classes和Component classes收集到

Component classes的列表，当GameObject生成的，这些类稍后会在其上创建。组件（Components）可以是蓝图组件子类（Blueprint Component Subclasses），我们主要在需要引用资源（Assets）时使用它们，比如这里可能是石块网格体（RockMesh）或弹道UI的资源。Capabilities对我们来说只能是代码（Code Only），我们不允许创建蓝图Capabilities子类。



Sheet_ThrowRock

表单也可以包含其他表单，例如，玩家移动表单包含了所有与玩家移动相关的其他表单，这就像一种收集和管理Features的好方式。

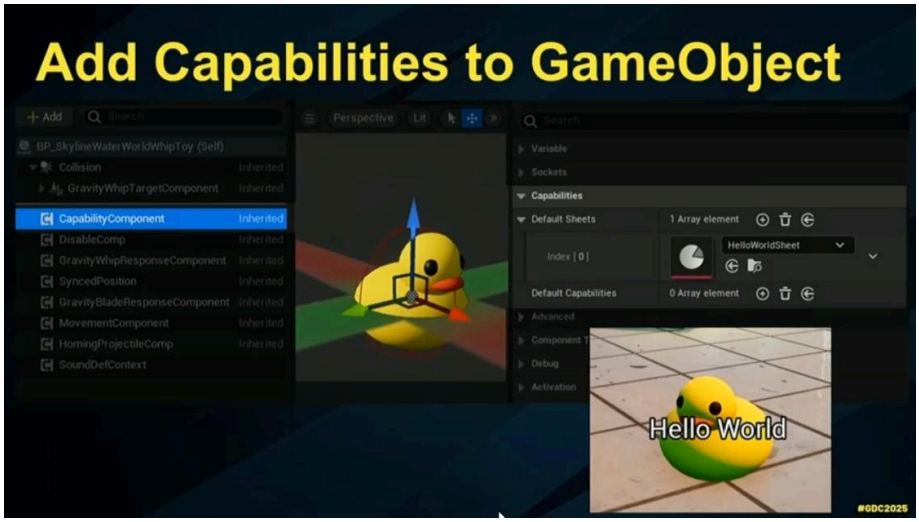


PlayerMovementSheet

这是表单包含内容的伪代码：Capability classes和Component classes，以及其他表单。

SheetAsset

中，你可以看到我们将表单添加到该组件上，在这个例子中我们添加了“Hello World”表单，它让鸭子在运行时打印“Hello World”。

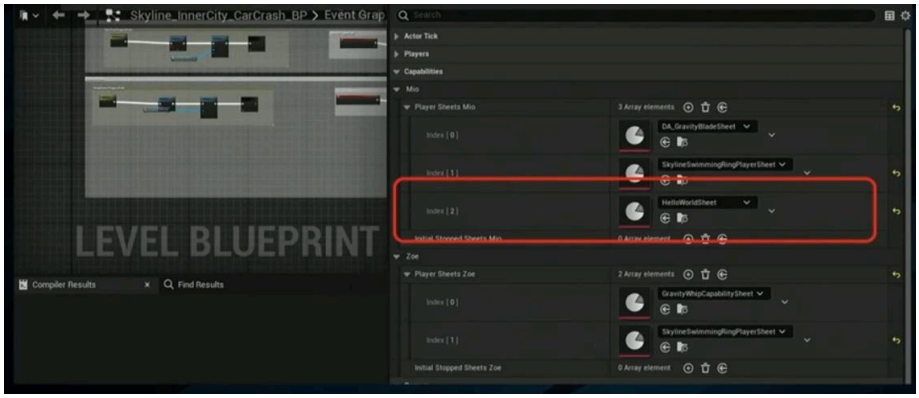


DuckExample

这是CapabilityComponent在底层的样子，在运行时它跟踪记录标签阻塞器（TagBlockers），就像我们之前讨论的跳跃标签阻塞（Jump Tag Block）；以及当GameObject生成时将被添加到其上的表单（DefaultSheets）；我们也支持添加单个Capability，如果你不想添加整个表单的话。值得注意的是，CapabilityComponent只是一个挂载点（Attachment Point），这里不存储Capability实例，因为我们不允许访问Capability实例，通常我们也不在运行时动态添加或删除Capabilities(注：这里有个例外就是在《双影奇境》代码解析01-Tutorial中提到的，ATutorialVolume可以直接添加Capability)，像一些交互（Interactions）它们会向玩家添加或删除表单，然后在关卡过渡（Level Transitions）时，玩家获得或失去表单，但除此之外我们可以确信，在整个关卡中GameObject上的Capabilities将保持不变。

CapabilityComponent

这很不错，因为它可以轻松地向GameObject添加特定的Capabilities，例如是否只有一个玩家或两个玩家都应该能够投掷石块，事实上，对于玩家我们在关卡上有定制的设置（注：这里指的是在关卡蓝图上配置）。我们添加表单后它们将在底层添加到玩家身上。这是我们为我们的需求定制的工具也是我们《双影奇境》的解决方案，但如果未来项目的需求发生变化，我们会改变我们的解决方案(注：潜台词就是直接配置在关卡蓝图里面太不灵活了)。



LevelBlueprint

还有一个底层的CapabilitySystem, 你可以将其视为一个TickManager，也许是一个单例（Singleton），它**包含所有Capabilities实例**，并按帧更新顺序优先级（Tick Order Priority）排序。我们的一些帧更新组（Tick Groups）包括：输入（Input）、移动前（BeforeMovement）、移动（Movement）、移动后（AfterMovement）、玩法前（BeforeGameplay）、玩法（Gameplay）、玩法后（AfterGameplay）、物理前（BeforePhysics）、物理（Physics）、物理后（AfterPhysics）等等。CapabilitySystem遍历所有Capabilities，按帧更新顺序优先级更新它们，检查它们是否应该激活或失活。如果需要就会切换激活状态，如果激活了它们就执行Capability的帧更新。

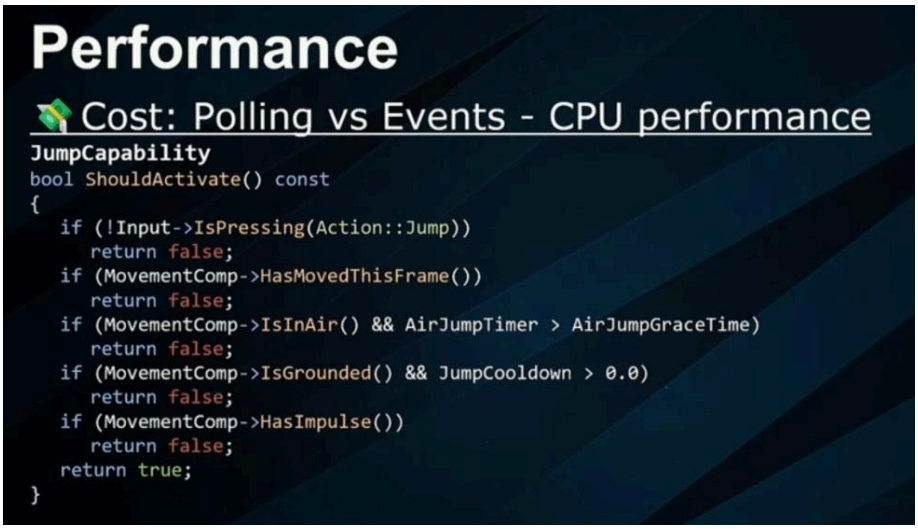
```
CapabilitySystem
{
  private TMap<ETickGroup, TArray<Capability>> AllCapabilities;

  UpdateCapabilities()
  {
    for each capability
        check should activate or deactivate (including blocks)
            Toggle activeness, Tick if active
  }
}
```

CapabilitySystem

Capability基类实现细节：在 `Setup` 时将自己注册到CapabilitySystem中，当游戏对象所有者（Owner）被销毁时，如果Capabilities是激活的我们首先需要运行 `OnDeactivated` ，然后才将Capability从CapabilitySystem中移除。

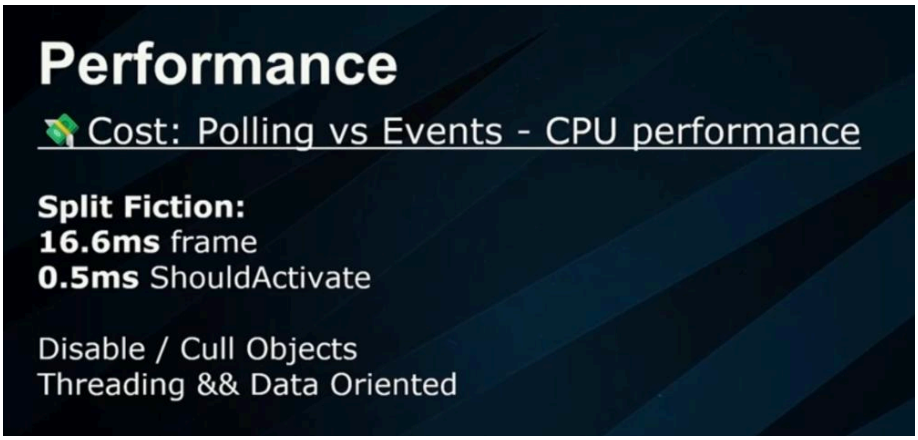
```
Capability base class
  …
    void Setup() virtual
    {
        CapabilitySystem::Register(this);
    }

    void OnOwnerDestroyed()
    {
        if (IsActive())
            OnDeactivated();
        CapabilitySystem::Unregister(this);
    }
  …
```

不是来自Capabilities本身，而是因为我们使用轮询（Polling）而不是事件（Events），最简单的描述是：在基于事件的结构中，当你按下跳跃按钮时，那是一个事件，你有一个回调函数（Callback）注册到跳跃函数（Jump Function），然而我们每帧轮询输入以检查：你现在是否按着它？所以在下图我们的JumpCapability中，我们每帧检查跳跃动作输入（Jump Action Input）。很难量化在事件驱动结构中，有多少逻辑在 `ShouldActivate` 中不需要运行，因为这些检查中也有一些环境条件，不仅仅是类似事件的东西（如输入动作）。



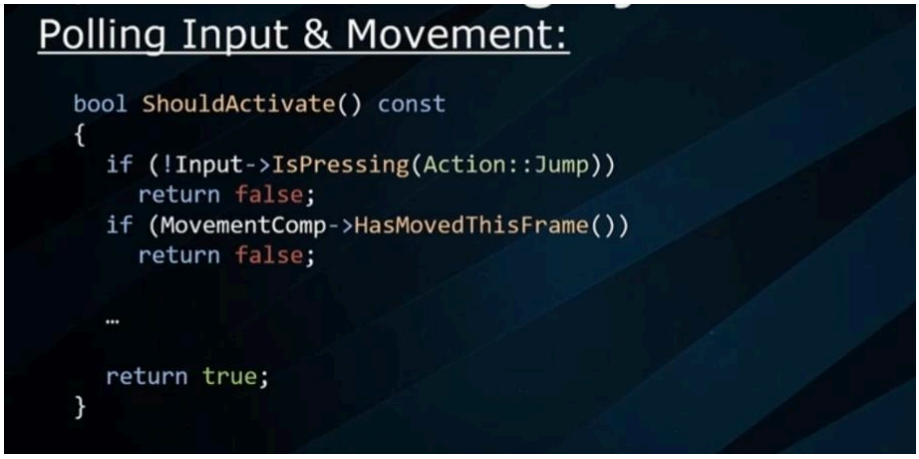
JumpCapabilityShouldActivate

我们实际测量数据为：《双影奇境》中一个典型的16.6毫秒帧内(即每秒60帧)我们花费0.5毫秒在所有Capabilities上运行 `ShouldActivate` ，如果对象离得很远我们也会禁用它们或减少调用。当然这**取决于你制作的游戏类型**，对我们来说这不是问题。如果你尝试使用Capabilities并在这里遇到瓶颈，你可以尝试进行线程化和面向数据编程（Data-Oriented），比如也许你可以结合使用其他结构，并非你所有的游戏玩法都需要在Capabilities中。



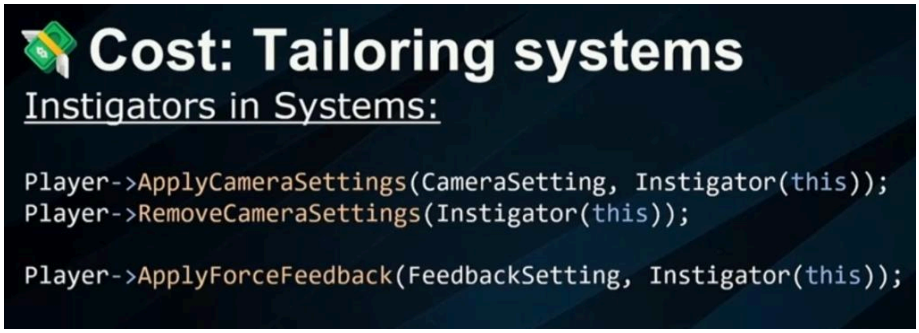
CPUPerformance

然而Capabilities还有其他更大的代价，那就是需要手动去调整其他系统以与轮询良好协作，例如输入和移动。因此这可能是一个或大或小的代价，取决于你的引擎，即使我们使用虚幻引擎，我们也有输入和移动的自定义解决方案，只是为了便于Capabilities轮询东西，比如“我们是否在这一帧准确按下了输入”，或者“是否已经有移动执行了移动，我们不应该再移动”等。



TailoringSystems

还有一个代价是将发起者（Instigators）集成到你的系统中。我们大量使用发起者，例如覆盖设置（Overriding Settings）、推入/弹出摄像机（Pushing/Popping Cameras）、应用力反馈（Applying Force Feedback）以及追踪伤害（Track Damage）等等。发起者特别适合与Capabilities一起使用，因为通常当你在爬梯子时，例如你可能想要不同的摄像机设置，你希望摄像机拉近（Zoom In）然后当你离开梯子时，你希望移除那个特定的摄像机设置，恢复到之前的状态。所以强烈推荐使用发起者。



TailoringSystemsInstigators

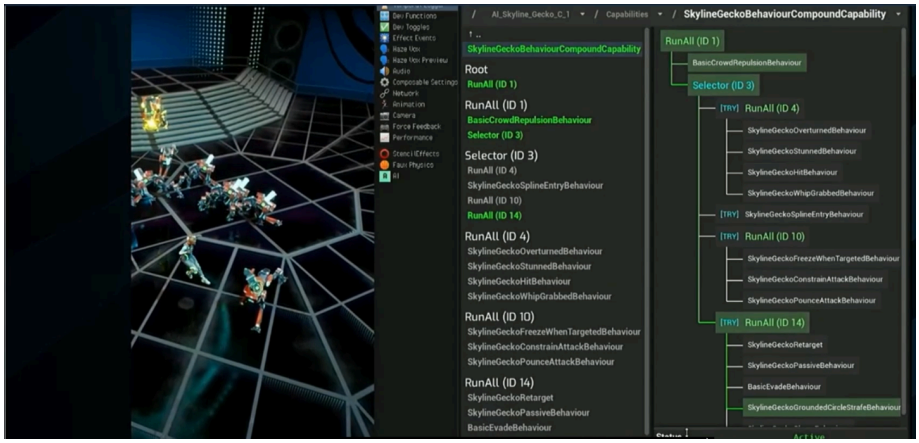
还有配套工具的研发。在我们的开发菜单中，有一个我们称为TemporalLogger的东西，它有点像一种可拖动的Timeline工具。使用这样的工具调试Capabilities非常快捷和容易。你可以精确地看到Capability何时激活或不激活，是什么阻塞了它，这对于时间相关和网络差异引起的Bug来说是无价之宝。顺便说一句，你可以将几乎任何数据记录到时间记录器，根据数据类型，我们会以不同方式显示。我们有工具组件来记录位置（Location）和动画（Animation），所以当你在时间线上拖动时，它会移动和播放动画。如果你想尝试Capabilities，我强烈建议你编写一个这样的工具，编写和维护一个时间记录器至关重要(注：类似与RewindDebugger的工具)。

TemporalLogger

When-Gameplay Examples(游戏中何时使用了它)

在我们游戏中至少5个领域使用了Capabilities，我们已经看了一些能力和移动的例子，我们也将Capabilities的变体用于AI行为树（AI Behavior Trees）、Boss攻击模式（Boss Attack Patterns）、谜题（Puzzles）等等。关于这些的具体细节我可以讲很多，但由于时间有限，我将简要介绍它们，以证明Capabilities的通用性，或许在你尝试使用Capabilities时它可以作为灵感的来源。

首先是AI行为树，下图所示一个AI机器狗我们称之为壁虎，因为它能在墙上爬行。它有很多普通的Capabilities。它还有我们称为Compound Capability的东西，Compound Capability可以包含Child Capabilities。这是我们创建AI行为树的方式。

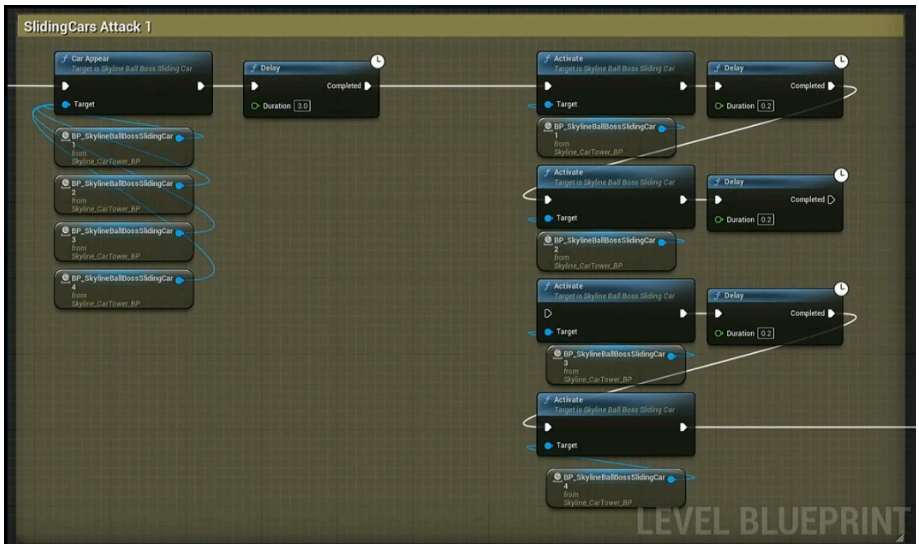


AIBehaviorTree

对于Boss攻击模式（Boss Attack Patterns），下图是一个球型Boss, 它是一个向玩家扔汽车的停车场管理员。

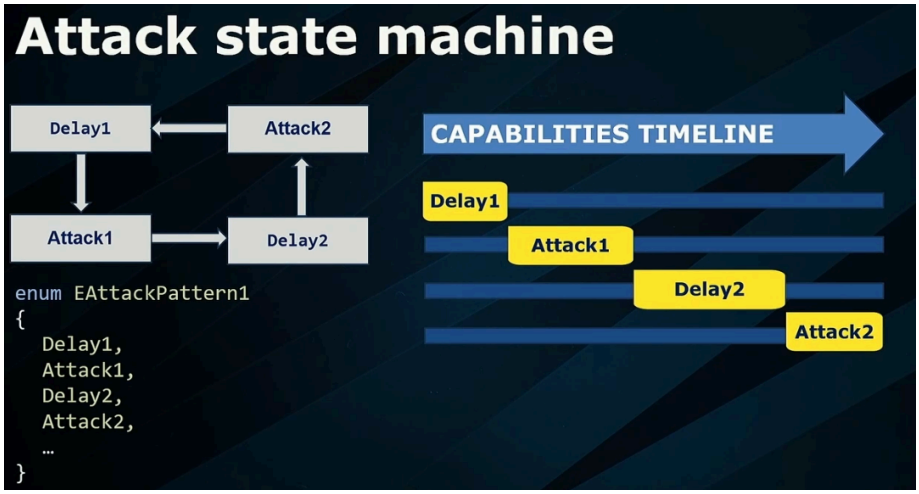
BallBoss

Boss攻击模式的Capabilities有一段历史，有时设计师做原型时会用蓝图做，然后程序员再将其移到代码中。我们想要Capabilities可以像蓝图一样结构化，易于概览易于添加延迟易于修改。



BallBossPrototypes

传统上我们处理Boss攻击模式的方法是创建一个枚举来表示攻击模式，并让它像状态机一样改变。所以当枚举推进时，Capability发生变化。这里有一个Capabilities固有的问题：我们只允许一个GameObject上存在一个Capability的实例。例如，你不能在玩家身上有两个PlayerJumpCapability。这没有意义。这意味着我们最终得到了大量相似的、令人痛苦的Capabilities。在Hazelight，我们虽然喜欢复制粘贴但这里显然有点过头了。



BossAttackStateMachine

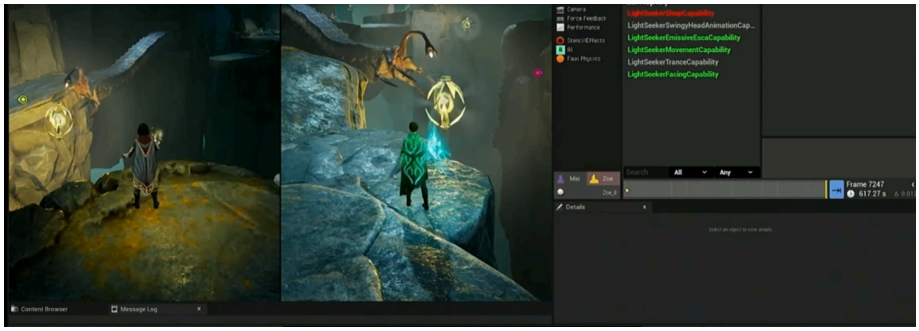
我们制作了我们称为ActionQueue的东西，它是一个包含空转（Idles）、事件（Events）、持续时间（Durations）的队列，最值得注意的是Capabilities。**ActionCapability**: 只有当它位于队列最前端时才会检查 `ShouldActivate` ，并且它会一直停留在那里直到失活（Deactivating）。

ActionQueue

现在我和设计师可以非常轻松地调整攻击的持续时间和顺序，我们可以为Boss添加阶段（Phases），复用攻击模式，并通过非常简单的乘数值让它们在战斗进程中变得更激烈。将行为放在代码中而不是蓝图中的额外好处是：在版本控制中进行跟踪和差异比较非常容易。我们在《双影奇境》中大约200个地方使用了ActionQueue，有趣的是我们大约15个月前才发明它。《双影奇境》的大部分内容实际上是在没有ActionQueue的情况下构建的，但我预测我们将来会大量使用它。顺便说一句，我强烈建议你们观看Elliot Mailer在GDC 2022上的演讲《C++ Coroutines are Now》。协程（Coroutines）在底层的工作原理上有根本的不同。你可以说队列（Queues）和协程（Coroutines）都暂停执行，但我们是一个结构体队列，而协程是一个关键字。我发现有趣的是，它们乍一看很相似，我认为这是因为它们都试图以多种方式解决如何编写攻击模式的愿望。所以我建议你看那个演讲，以获得更多关于如何编写游戏玩法代码的参考。

Coroutines

谜题（Puzzles）：下图这个趋光蠕虫可以帮助玩家跨越平台间隙。当其中一个玩家在附近发光时它会被引诱出来，当光熄灭时它会退回到巢穴中。我们的大多数谜题实际上没有任何Capabilities，也就是说所有逻辑都非常简单，我们可以直接将逻辑保留在实际的GameObject中。它不需要被扩展到Capabilities中。但在这个例子中，它相当不错，因为光蠕虫有很多不同的状态，所有这些Capabilities和状态都与实际的玩家Capabilities解耦，即使它们受到玩家Capabilities的影响。

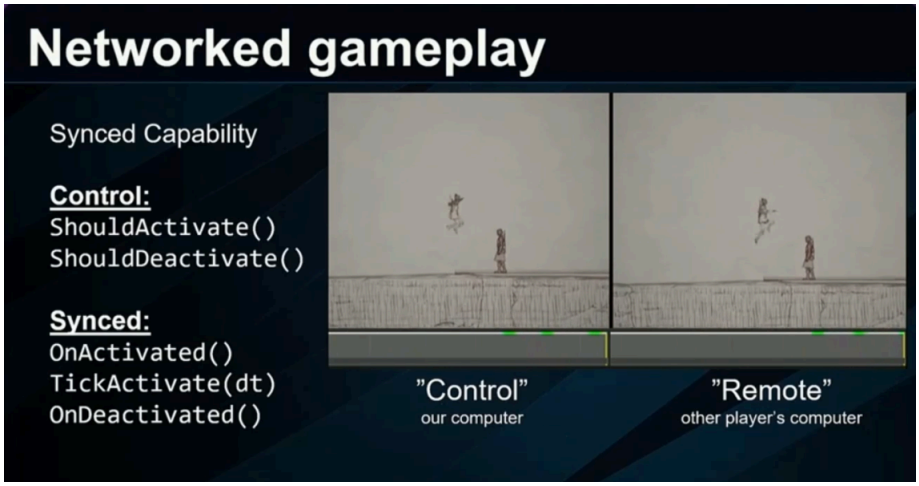


LightSeeker

事实上，任何时候你有东西使用状态，即使是像下面这个接待员这样的OptionalContent——它即不是敌人也不是谜题——它只是为玩家提供一些有趣的互动。我们实际上为它那种像素化面部表情使用了ActionQueue，所以我们将ActionQueue用于不仅仅是攻击模式，任何按顺序工作的东西，我们都可以使用ActionQueue。

SkylineInnerReceptionistBot

我猜你们不会都做分屏合作游戏，但对我们来说，Capabilities带来了很多网络便利。我们可以制作Synced Capabilities。例如玩家跳跃是一个Synced Capability，你的输入只存在于你的电脑上。所以当尝试跳跃的 `ShouldActivate` 运行时它只在控制端运行，跳跃在 `OnActivated` 、 `TickActive` 和 `OnDeactivated` 时同步到两台电脑，以确保在网络上看起来一切正常，因此如果讨论的Capabilities是Player Capability比如跳跃，控制端显然是该玩家。环境中的东西比如敌人身上的Capabilities，通常由主机（Host）控制。所以即使我们只有两个玩家，其中一人实际上是主机。

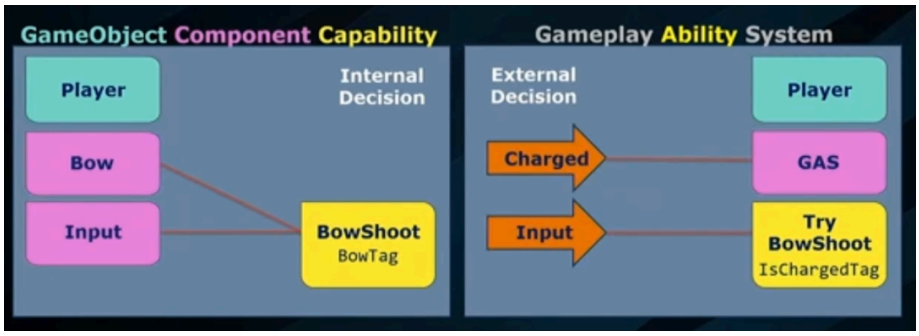


SyncedCapability

对于ActionQueue如攻击模式，我们从主机同步the Timin
电脑上本地执行队列。我们也有Local Capabilities，其 Sh

测（Prediction）和保底方案，因为此时我们并不真正知道另一个玩家是跳了还是真的在掉线或者他们在做什么，直到我们收到来自SyncedCapabilities的信息。而且有时Capabilities本身也对同步不感兴趣，所以能够拥有本地Capabilities也很不错。

我也收到一个问题：Capabilities和虚幻的GAS有什么区别？我本人没有用过GAS。这只是我试图拼凑和理解的信息。我相信两者都试图实现细粒度和内聚的行为，但从理念上看Capabilities和GAS的结构是相反的，在Capabilities中ShootCapability本身决定射击何时发生，决策逻辑包含在Capabilities中，激活（Activation）取决于**轮询（Polled）**的数据；在GAS中，**外部类和事件回调**决定射击能力何时触发，所以在GAS中，你可以说行为是被推送的。GAS也是数据驱动的，所以一个能力（Ability）是一个资源（Asset）。在Detail中你指定标签条件，这些条件启用或禁用该能力。所以它们的能力标签（Ability Tags），既阻塞也允许行为，因此当射击的输入事件发生时，会有一个尝试射击的回调，该回调是否被允许取决于标签的状态。这归结为一个事件权衡的问题，使用事件可以获得很多好处：你获得解耦的类，你获得性能，因为你进行Lazy Evaluation。但你确实会得到一些依赖，并且可能难以Overview和调试，以跳跃能力为例：你尝试按跳跃键，没反应，在Capabilities中设置断点并查看哪里失败非常容易，但在基于事件的结构中要弄清楚问题出在哪里可能有点麻烦。这就是Capabilities和GAS之间的主要区别。



VSGAS

Summary(总结)

Capabilities的收益和代价是：

- **收益：** 生产效率高，对设计变更灵活，代码所见即所得，整洁地处理特殊情况扩展，工作方式真的很有趣
- **代价：** 轮询与事件相比的性能代价，也许更重要的是调整其他系统以适配Capabilities系统，以及编写工具以便于调试Capabilities

与其他模式相比，该模式的独特之处在于：

1. 它们不同于状态机，因为Capabilities是**并行**（Parallel）运行的，不是顺序的
2. 它们类似于ECS中的系统，但被用于**Game Object-Component**架构中
3. 它们在某些方面类似于虚幻的GAS，但Capabilities通过**轮询**数据来设计行为，而不是推送
4. 其核心理念是：**GameObjects做决策**，而不是Components或其他东西

五个Capabilities有用的游戏玩法领域是：

1. 移动（Movement）
2. 能力（Abilities）
3. AI状态（AI States）/ 行为树（Behavior Trees）
4. Boss攻击模式（Boss Attack Patterns）
5. 谜题（Puzzles）/ 环境互动（Environments）

以及更多。我认为Sky is the Limit。我相信你可以将它们用于很多方面。

结论

即使Capabilities对你的项目没用，也许Hazelight的一些理念是有用的，那就是：**首要的是，为你的需求定制解决方案**，工具和迭代速度在制作游戏时至关重要，**不要害怕放弃行业教条**，如果它对你的工作室和项目有益，例如我们大量复制粘贴代码，我认为很多工作室会建议不要这样做。

我希望你们喜欢这次演讲，并希望你们将来能尝试Capabilities并拓展其潜力，谢谢大家。

QA

> **观众：**非常感谢你的演讲。我最近刚和朋友通关了《双影奇境》。我非常好奇…（剧透预警）在最终关卡，有一个场景，玩家可以通过一条小线在不同的世界之间切换，这些不同世界的碰撞可以无缝过渡。我有点想知道，这个游戏玩法是如何实现的？
> **Ylva:** 我不完全确定，但我问过我们的技术总监，我相信答案是：我们实际上把所有的世界同时（Simultaneously）放在关卡的不同位置（Different Places），然后只是把它们渲染（Render）在彼此之上（On Top of Each Other）
> **观众：**性能真好。
> **Ylva:** 是的，我们有一个很棒的技术总监，他太厉害了。

（全结束）

我的总结

1. 很多方式是反教条的，比如复制拷贝，大量使用Tick，但它是对于《双影奇境》来说是合适的，它是偏线性的双人闯关游戏，Tick数量可控，目标平台是PC, PS5，XBox等平台，不用考虑手机那羸弱的计算能力，相反对于设计师来说那就太爽了，没有什么事情能比写Tick逻辑更让人舒服的事情了，看似反教条其实对应了那句“没有最好只有最合适”；
2. **配套工具(调试工具，配表工具，自动检查工具等等)研发的重要性**，有的时候程序的代码跑通基本流程不算工作完成，没有配套工具会很影响最终产品的发布时间和最终质量，程序一定要把配套工具和核心Runtime代码看待的一样重要；
3. 理想很丰满现实很骨感，当你的设计师没有任何脚本经验并期待他能独立完成玩法设计工作时还是需要花大量时间去培训学习，比如熟悉脚本语言，一些复杂耗时的计算不可以放到Tick中去等等；
4. **个人感觉**大部分多人联网游戏(手机平台)不太适合使用Capabilities模式开发，很难做优化；
5. Instigators模式可以加到自己项目中去，因为很多时候都会下意识使用计数的方式；

Next

下一篇《双影奇境》代码解析02-Capabilities 将会剖析一些代码细节，比如上面提到的光蠕虫，爬梯子，弓箭，如何利用HazeCapabilitySheet串联流程等等；

发布于 2025-07-13 08:58 · 北京

游戏开发 虚幻引擎 虚幻 4（游戏引擎）

理性发言，友善互动

35 条评论 默认 最新

Acmarkdry
他这个capabilities的调试工具很有意思，因为3d游戏开发有一个很大的问题，当一个表现出现的时候，比如你的场景里面突然出现一个球，你如果不了解本身机制的话，是不知道这个球是从哪里来的，如果可以可视化的看到在这个瞬间有哪些capabilities被激活，就可以辅助调试
2025-07-13 · 广东 回复 10

borderwing
是 这个调试工具反而感觉是更有价值的一环 debug逻辑来源感觉很清晰
2025-07-21 · 广东 回复 4

和风
看下来其实整体的底层逻辑是，数据驱动，以及永远保证逻辑是单向的。其实是很不错的想法，至少保证了复杂度是有明确的上限和调试是有明确的stack的。实际上游戏开发都喜欢用老套的MVCS，大多数时候低水平的开发者缺乏架构意识或者说软件设计水平不高，反而导致了MVCS完全没有带来解耦，反而是难以调试追踪的callback、长度惊人的调用链、不可计数的异步和耦合到爆炸的System、Controller。MVCS实际上是一种很理想的契约，他承诺了各个模块各司其职、他推崇单一职责、推崇开闭原则，但是现实并不是这样。而且MVCS最大的弊端在于，实际上中型以上的游戏来说，使用MVCS几乎直接和不可以单元测试划了等号，这个架构有太多的依赖。本文介绍的这种数据驱动+行为容器的范式和我个人比较喜欢推崇的MVVM，我觉得比MVCS先进非常多。可惜我国的游戏开发从业者水平堪忧
01-30 · 上海 回复 2

Danta1ion
这不就是GameplayTag和GAS
2025-07-13 · 上海 回复 4

Acmarkdry
其实我觉得更像是ecs的进化版本，ecs一个很重要的概念就是通过分层将数据和函数给拆分开来，他们将ecs里面的system给抽出来从一个全局的管理类变成做某个单一功能的组件，从而让开发逻辑更清晰。
2025-07-13 · 广东 回复 8

展开其他 4 条回复 ›

uniHk
框架设计的确是没有最好的，只有最适合的
2025-07-18 · 广东 回复 5

游戏风子
能做的事和gas类似，但换了个思路。最大的优势是好的规范和高自由度，能大幅提升开发效率降低维护成本，变相的也能提高玩法开发的上限（期待后续作品更高的品质）。劣势主要在网络开发上（需要改造做些取舍），性能问题看游戏类型，只要不是开放世界类的，问题都不大，开放世界的优化还是要看项目情况，理论上可以换ecs版的实现，相应的也会增加理解和维护成本（没见过类似的实现）。另外过于灵活并不适合常规团队开发，尤其是国内这种开发环境🤭🤭🤭
2025-07-14 · 上海 回复 4

罗蘑菇
很有意思的想法，很多设计理念其实跟GAS 也相似，参考了ECS的 system设计思想，Capability的职能尽可能单一，但是并没有进行原子化，如对于投掷石块和射箭的例子，他们通过复制代码断开这种相似能力的耦合，并没有对基础能力做更细粒度的原子化拆分，Capability粒度尽可能的细却又不那么细，个
计因项目而异，不适合做过于复杂的技能系统的）。这

鲨鱼辣椒星星酱
其实可以优化一下轮询，又多少个Capabilities就生成多少个Capabilities的数组，浪费点内存，但是值得，Tags 可以用计数法，只有在editor模式下才能看到发起者，runtime,就是一个计数，不过这一套很又启发
2025-07-14 · 浙江 回复 1

知乎用户928 ‣ **Mirro**
智能指针不也是计数吗。只不过打包后没有来源可能不利于bug反馈。毕竟不是所有的问题都是有条件在editor调式的
01-23 · 四川 回复 喜欢

Mirro
总感觉计数法会出现问题不安全
2025-07-22 · 江苏 回复 喜欢

KKK
请问有视频链接吗求求了
2025-07-16 · 福建 回复 喜欢

zqj
为啥做性能测试，as的基于mir的jit不如lua5.4快；
03-08 · 四川 回复 喜欢

徐徐穹
确实很清晰
01-21 · 河南 回复 喜欢

点击查看全部评论 〉

理性发言，友善互动

推荐阅读

虚幻5渲染编程（动画篇）【第二卷：Expanding...
Yivan... 发表于虚幻5渲染...

Cygames2024技术大会-碧蓝幻想 Relink分享
指南7

虚幻引擎2D游戏开发教程系列
本教程系列涵盖了使用虚幻引擎创建 2D 游戏的所有方面。 这个系列有文字和视频两种形式。 虚幻引擎文字教程系列 开始使用虚幻引擎本教程是虚幻引擎的入门指南。它会带您学习虚幻编辑器，向...
黑客不黑 发表于独立游戏开...

【Shaders for GameDev】Part2：血条 Demo、距离场...
苏格拉没有... 发表于学习笔记