СОЗДАНИЕ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ С ИСПОЛЬЗОВАНИЕМ **UE4**

UNREAL® ENGINE

VR

для разработчиков

Митч **МАККЕФРИ**


UNREAL
ENGINE

Mitch **McCAFFREY**

UNREAL® ENGINE VR Cookbook

Митч **МАККЕФРИ**

UNREAL® ENGINE VR для разработчиков

БОМБОРА™
Москва 2019

УДК 004.9
ББК 32.973.26-018
М15

MITCH MCCAFFREY
UNREAL ENGINE VR COOKBOOK: DEVELOPING VIRTUAL REALITY WITH UE4

Перевод книги подготовлен в ходе проведения исследования в рамках Программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ) и с использованием средств субсидии в рамках государственной поддержки ведущих университетов Российской Федерации «5—100»

Макеффри, Митч.

М15 Unreal Engine VR для разработчиков / Митч Макеффри ; [пер. с англ. Н.И. Веселко, О.В. Максименковой, А.А. Незнанова]. — Москва : Эксмо, 2019. — 256 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-101419-3

Первое руководство по созданию виртуальной реальности с использованием движка Unreal Engine 4 на русском языке!

VR — новый, удивительный рубеж рубеж для разработчиков игр и специалистов по визуализации. А Unreal® Engine 4 — идеальная платформа для этого. «Unreal® Engine VR. Руководство по разработке» — это исчерпывающее руководство по созданию потрясающих приложений на любых VR-устройствах, совместимых с Unreal Engine 4.

УДК 004.9
ББК 32.973.26-018

ISBN 978-5-04-101419-3

Оглавление

ВВЕДЕНИЕ

Становление виртуальной реальности (*VR*) происходит на наших глазах, именно сейчас зарождается огромный спрос на новый и захватывающий опыт в *VR*. В качестве основы растущего многомиллиардного рынка *VR* ставит восхитительно сложные задачи перед разработчиками компьютерных игр, причем одновременно с погружением многих ранее не затронутых отраслей в мир компьютерной графики в реальном времени.

Создание аркадной игры в классическом стиле, визуализация изысканной виллы на травянистых холмах Тосканы — независимо от типа игры *VR* обеспечивает беспрецедентный уровень погружения в любую приглянувшуюся вам сферу. Однако это погружение вызывает множество проблем. При разработке под *VR*, хотите того или нет, вы пишете свои правила.

В этой книге рассмотрены общие парадигмы человеко-машинного взаимодействия, появившихся за последние несколько лет, и лучшие практики в этой области. Все сообщество *VR* — от крупнейших игроков до разработчиков-одиночек — с потрясающей скоростью вносит свой вклад в свод знаний *VR*. Эта книга не только продемонстрирует, как реализовать эти парадигмы средствами *Unreal Engine*, но и поможет выбрать решения, подходящие именно вашему проекту.

В основе книги рецептов лежит практический подход к изучению особенностей *VR*-разработки. Каждый «рецепт» представляет собой вариант сборки достаточно общей системы, подходящей для разнообразных *VR*-игр. Независимо от того, разрабатываете вы игру-стрелялку от первого лица или симулятор для отдыха, содержание каждого примера достаточно абстрактно для использования в игре любого жанра, но дополнительно предлагает конкретные решения, хорошо работающие для определенных типов игр.

Кому будет полезна эта книга?

Эта книга предназначена для людей, которые уже знакомы с навигацией в *Unreal Engine* 4 (*UE*4) и *Blueprints*. Если у вас мало опыта, ознакомьтесь с документацией по *Unreal Engine* перед чтением этой книги в интернете по адресу: *http://docs.unrealengine.com*.

Тем не менее я объясняю большинство вещей, когда дело касается непосредственно кодирования, а большая часть математики раскрывается в приложениях и покрывается основным содержанием книги, поэтому серьезных навыков разработки программного кода от читателя не требуется.

Как организована книга

Книга состоит из трех частей.

1. Часть 1 «Начало работы»: в главах 1–3 содержится введение в некоторые термины, используемые в этой книге и в отрасли *VR*. В этой части также содержатся инструкции по созданию простых проектов для различных *VR*-шлемов.
2. Часть 2 «Рецепты»: главы 4–10 содержат основные «рецепты» книги. Эта часть охватывает все: от взаимодействия с контроллером до схем движения в *VR*.
3. Часть 3 «Приложения»: это вспомогательная информация о редакторе *VR* и ресурсах, которые помогут вам в *VR*-разработке.

Условные обозначения, используемые в этой книге

В этой книге используются следующие соглашения при оформлении текста:

- моноширинным шрифтом набраны блоки программного кода;
- *курсивным начертанием* выделены ключевые слова или фразы.

> **Заметка**
>
> Означает подсказку, предложение или общее примечание.

ДОПОЛНЕНИЕ

Содержит вспомогательную информацию к основному тексту, такую как объяснение используемых математических принципов или работ, связанных с основным содержанием.

> **Предупреждение**
>
> Обозначает предупреждение или предостережение.

Почему *Blueprints*?

Когда вы программируете в *UE*4, есть два способа реализации логики игры: язык визуального программирования *Blueprints* и ставший традиционным язык программирования *C++*.

По сравнению с *Blueprints*, *C++* может быть сложнее, поскольку изучение синтаксиса может занять некоторое время; однако он предлагает различные варианты в использовании скрытых возможностей ядра. Слабое владение *C++* не создаст проблем при работе с этой книгой, потому что большая часть материала представлена на уровне, чтобы возможности *Blueprints* позволяли его освоить.

Использование *Blueprints* также облегчает способы миграции вашей работы из одного проекта в другой, что позволит вам взять любую работу, проделанную по этой книге, и легко применить ее в ваших проектах.

Дополнительные материалы

По адресу *https://eksmo.ru/files/ue4_vr_projects.zip* можно скачать архив с файлами с исходным кодом для каждой части. Это позволит быстро проверить работоспособность рецептов, приведенных в книге.

Об авторе

Митч Маккефри — независимый разработчик игр и создатель многих общественных ресурсов для разработчиков виртуальной реальности на *Unreal Engine*. Он обучает лучшим практикам разработки игр на своем *YouTube*-канале *Micth's VR Lab*, где демонстрируются примеры, предложенные членами сообщества *VR*-разработчиков. Адрес его сайта *https://mitchellmccaffrey.com*.

ЧАСТЬ I

НАЧАЛО РАБОТЫ

ГЛАВА 1

ТЕРМИНОЛОГИЯ И ЛУЧШИЕ ПРАКТИКИ

Мир разработки виртуальной реальности (*VR*) может показаться сложным из-за разнообразия доступного на рынке оборудования и программного обеспечения. Кроме того, поскольку *VR* — новая среда, многие принципы, принятые разработчиками игр, неприменимы и могут не работать в мире *VR*-игр.

Если вы не уверены, что понимаете разницу между *Oculus VR*, *OSRV* и *OpenVR*, или просто ищете общие рекомендации о том, как начать работать с *VR*, эта глава для вас.

1.1. Терминология

Экосистема *VR* непрерывно усложняется и развивается благодаря обилию технологий, программного и аппаратного обеспечения. Чтобы убедиться, что мы верно понимаем друг друга, рассмотрим ключевые части этой экосистемы, которые должен знать каждый *VR*-разработчик, использующий *Unreal Engine* 4 (*UE*4). Если вы знакомы с текущим состоянием отрасли виртуальной реальности и используемыми технологиями, можете пропустить эту главу.

1.1.1. Устройства

Для работы с виртуальной реальностью вам потребуются *шлем виртуальной реальности* [*Head Mounted Display — HMD*] и контроллер. *Unreal Engine* на уровне коробочного решения поддерживает большинство из представленных на рынке решений, что избавляет разработчиков от необходимости однозначного выбора конкретного *VR*-шлема на первых этапах проекта. Кроме того, в *UE*4 для *VR* реализован высокий уровень абстракции, что позволяет легко справиться со сменой устройства, для которого вы разрабатываете проект (или если вы ориентируетесь сразу на несколько устройств).

Таблица 1.1 содержит список и описание *VR*-шлемов, штатно поддерживаемых *UE*4.

Таблица 1.1. Совместимые *VR*-шлемы (*HMD*)

VR-шлем	Описание
Samsung Gear VR	*Gear VR* — аппаратура, совместимая со смартфонами *Samsung*, разработанная в партнерстве *Oculus* и *Samsung* с целью обобщить возможности *Oculus* в области программного обеспечения для мобильной виртуальной реальности. При работе с *Gear VR* вы можете использовать все функции и преимущества, которые дает среда разработки *Oculus Mobile software development kit* (*Oculus Mobile SDK*). *Gear VR* в настоящее время поддерживает только отслеживание поворотов головы виртуальной модели
HTC Vive	*HTC Vive* разработан в результате партнерства, сходного с *Gear VR*, но в котором *HTC* выступили поставщиком аппаратного обеспечения, а *Valve* — программного *SDK* в виде *SteamVR/OpenVR*. *Vive* обеспечивает отслеживание вращений, а также взаимно однозначное отслеживание положения шлема и включенных контроллеров движения. С прицелом на разработку под *Vive*, *Valve* предоставляет *OpenVR SDK* (для получения дополнительной информации о *OpenVR* см. таблицу 1.3), который предоставляет доступ к различным датчикам и данным настройки, необходимым для *VR*
Oculus Rift	Аппаратура *Oculus Rift* прошла через много итераций, представленных как выпуски для разработчиков, не предназначенные для широкой публики. В 2016 году *Oculus* выпустила первую массовую версию (*CV*1), предназначенную для продажи пользователям. *CV*1 — это полностью интегрированная система, состоящая из оборудования *Oculus* и программного обеспечения (см. *SDK Oculus* в таблице 1.3). *Rift* поддерживает отслеживание вращения, подобно *HTC Vive*. В комплект также входит контроллер *Xbox One* и пульт дистанционного управления *Oculus Remote*, являющийся устройством ввода по умолчанию; *VR*-контроллеры *Oculus Touch* (см. таблицу 1.2) продаются отдельно

VR-шлем	Описание
Google Cardboard	*Google Cardboard* — это небольшая и недорогая (350–1500 рублей) картонная коробочка, в которой есть одна кнопка и линзы. Коробка позволяет закрепить смартфон и просматривать стереометрический контент. «Картон» использует встроенные датчики смартфона для обнаружения вращения головы пользователя (разумеется, на неоткалиброванном телефоне измерения будут неточными). *Google Cardboard* использует собственные *SDK Android* для обеспечения совместимости с устройствами (что может приводить к существенным задержкам рендеринга по сравнению с другими решениями). Тем не менее «Картон» на сегодня является самой дешевой *VR*-гарнитурой
Google VR/Daydream VR	*Google VR* или *Daydream VR* — еще одна инициатива компании *Google*. По сравнению с *Google Cardboard* эта программно-аппаратная система контролируется жестче. Для работы с ней ваш смартфон и гарнитура должны быть классифицированы как *Daydream Ready*, что позволяет *Google* использовать более продвинутые функции программного обеспечения. *Daydream* также поддерживает контроллер *Daydream* (см. таблицу 1.2), предоставляющий сенсорную панель и обеспечивающий отслеживание вращения. Таргетинг *Daydream* реализуется через *Google VR SDK* (таблица 1.3)
PlayStation VR	*PlayStation VR* является периферийным устройством для *VR*-игр на игровых приставках (консолях) *Sony PlayStation* 4 и *PlayStation* 4 *Pro*. Отслеживание положения осуществляется при помощи внешней камеры *PlayStation*, отслеживание вращения реализовано гиростабилизаторами (*IMUs*, *inertial measurement units*) примерно так же, как у других шлемов, представленных в этой таблице. Игроки могут использовать контроллеры *PlayStation Move* (таблица 1.2). Также можно использовать контроллер *DualShock* 4, в том числе для отслеживания положения
OSRV	Виртуальная реальность с открытым исходным кодом (*Open Source Virtual Reality, OSRV*) — это инициатива, поддерживаемая несколькими корпоративными партнерами. В настоящее время *OSRV* предлагает *Hacker Development Kit (HDK)* в виде *HDK* 1.3 и *HDK* 2, разработанных компанией *Razer*. Обе гарнитуры обеспечивают отслеживание положения и вращения

В *UE*4 контроллеры перемещения поддерживаются с помощью единого компонента *Motion Controller Component*, который упрощает взаимодействие, в том числе с несколькими контроллерами. Список контроллеров перемещения, поддерживаемых *UE*4, приведен в таблице 1.2.

Таблица 1.2. Совместимые контроллеры

Контроллеры перемещения	Описание
Oculus Touch	Контроллеры *Oculus Touch* позволяют отслеживать как вращение, так и положение при помощи системы *Constellation*. Каждый контроллер имеет пару кнопок (*A* и *B* на правом, *X* и *Y* на левом контроллере) и аналоговый джойстик под большой палец, кнопку под указательный палец («спусковой крючок») и кнопку для остальных пальцев. Поддержка этих контроллеров осуществляется через *Oculus SDK*
Vive	Контроллеры *Vive* позволяют отслеживать вращение и перемещение при помощи системы *Lighthouse*. Каждый контроллер содержит круговой трекпад, который работает как аналоговый джойстик или кнопка, а также кнопку под указательный палец («спусковой крючок»), кнопку меню и кнопку «Захват». Работа с контроллерами *Vive* обеспечивается при помощи *OpenVR SDK*

Контроллеры перемещения	Описание
PlayStation Move	Контроллер *Move* позволяет отслеживать вращение и перемещение при помощи камеры *PlayStation* и светящегося шара на самом контроллере. Помимо обычных для контроллеров *PlayStation* кнопок *Cross* (крест), *Circle* (окружность), *Triangle* (треугольник), *Square* (квадрат), *Start* (запуск) и *Select* (выбор), контроллер *Move* имеет кнопку *Move* (перемещение), а также «спусковой крючок» (или кнопку *T*)
Daydream	Контроллер *Daydream* поддерживает только отслеживание вращения. У контроллера также имеется сенсорная панель, которая может выступать в качестве кнопки, и кнопка *App* под ней. Работа с контроллером *Daydream* осуществляется при помощи *Google VR SDK*

1.1.1.1. Программное обеспечение

Существует множество программных библиотек, *SDK** и *API***, помогающих взаимодействовать с оборудованием виртуальной реальности. Идеология *UE*4 построена на повышении уровня абстракции до единых аппаратно-независимых интерфейсов и компонентов, что облегчает совместимость. При необходимости в ваших руках остается возможность прямого взаимодействия с различными *SDK*. Вам как разработчику будет полезно ознакомиться различиями в концепциях проектирования при помощи этих *SDK*, поскольку при разработке игры или реализации другого проекта могут требоваться преимущества конкретных функций выбранного *SDK*. Для работы с пакетами *SDK*, вам не нужно скачивать какие-либо отдельные файлы, поскольку *UE*4 включает их при скачивании модулей.

Список и описание совместимых *SDK* приведены в таблице 1.3.

Таблица 1.3. Совместимые *SDK*

SDK	Описание
Oculus PC SDK	*Oculus PC SDK* обеспечивает *UE*4 всей необходимой информацией для рендера подходящего представления. Эта информация включает в себя положение и ориентацию головы пользователя, а также его межзрачковое расстояние (*interpupillary distance*, *IPD*). *UE*4 может сгенерировать кадр и загрузить его в *Oculus compositor*, где будут применены искажения, необходимые для настройки линз *VR*-шлема. К счастью, движок в большинстве случаев справляется с этим и не требует от вас вмешательства. *Oculus SDK* также предоставляет слои, которые дают вам возможность рисовать на экране объекты, имеющие разное разрешение. Это полезно для пользовательского интерфейса (*UI*), где может потребоваться, например, отобразить текст с разрешением выше, чем у фона. Такие функции, как *ATW* (*Asynchronous Timewarp*; см. таблицу 1.4) автоматически обрабатываются во время выполнения и не требуют от вас дополнительных действий

* *Software Development Kit* — комплект для разработки программного обеспечения. — *Прим. ред.*

** *Application Programming Interface* — интерфейс прикладного программирования. — *Прим. ред.*

SDK	Описание
Oculus Mobile SDK	*Oculus Mobile SDK* предоставляет доступ ко многим возможностям, соответствующим *PC SDK*, включая информацию о положении головы и другую пользовательскую информацию, необходимую для рендеринга. Как и среда исполнения *PC*, мобильная среда исполнения использует *ATW*, но она дополнительно поддерживает рендеринг переднего (или кадрового) буфера, также известный как *scanline racing* (см. таблицу 1.4)
Oculus Audio SDK	*Oculus Audio SDK* предоставляет возможности реалистично распределить звук в пространстве и создать 3*D*-звук для *VR*. Этот *SDK* позволяет применять *HRTF** (*head-related transfer function*) к аудиоисточникам для имитации направленного звука (посредством моделирования геометрии человеческого уха). Он также может имитировать расстояние от звука до уха, гася громкость. *HRTF Oculus Audio SDK* реализован в *UE*4, позволяя вам настраивать кривые, ограничивающие распространения звука. Однако в настоящее время реализовать их можно только в проектах на базе *DirectX* (например, *PC*)
OpenVR SDK	*OpenVR SDK* во многом схож с *Oculus PC SDK* и предоставляет доступ к информации о *VR*-шлеме (включая положение и ориентацию в пространстве). *OpenVR SDK* также предоставляет интерфейсы для доступа к системе *Chaperone* (описывающей виртуальные границы, обозначающие игровое пространство для пользователя) и *Overlay* для генерации 2*D*-контента в компоновщике аналогичном слоям *Oculus*. *OpenVR* также поддерживает перепроецирование (аналогично *ATW*) и предсказание
OSRV SDK	*OSVR SDK* дает вам доступ к набору интерфейсов для создания *VR*-сюжетов. Построенный по принципу «поддержка всего», данный *SDK* включает наиболее значимые интерфейсы для получения такой информации *VR*-шлема, как положение и ориентация в пространстве и интерфейсы для отслеживания глаза, всенаправленных беговых дорожек (*omnidirectional treadmills*), жестов и так далее. *Unreal Engine* интегрируется с *OSRV SDK*, начиная с версии 4.12
Google VR SDK	*Google VS SDK* дает вам доступ к данным отслеживания положения головы *VR*-шлема и набор функций для оптимизации вашего *VR*-проекта. Это позволяет включить устойчивый режим производительности, что важно для запусков на смартфоне, потому что смартфон может перегреться от высокой нагрузки и резко снизить производительность, что может плохо повлиять на восприятие пользователем *VR*. *Google VS SDK* также предоставляет доступ к функциям *scanline racing*, которые оптимизируют задержку в линии рендеринга (см. таблицу 1.4)

Помимо *SDK* и библиотек, которые *UE*4 использует для взаимодействия с различной *VR*-аппаратурой, в ядре также реализованы некоторые программные функции, позволяющие значительно улучшить впечатления пользователя от *VR*. Однако существуют и другие программные функции (например, *ATW*), реализованные в различных средах исполнения, как следствие, включенные в ядро по умолчанию и неподконтрольные разработчикам. Таблица 1.4 содержит примеры функциональности обоих видов, доступных в *UE*4.

* *Head-related transfer function* — передаточная функция, описывающая положение источника звука относительно слушателя. — *Прим. ред.*

Заметка

Многие из этих функций подробно рассматриваются далее в главе 10, «Оптимизация *VR*».

Таблица 1.4. Примеры поддерживаемой функциональности

Особенности	Описание
Instanced stereo rendering	*Instanced Stereo rendering* генерирует картинку для левого и правого глаза игрока одновременно, а не последовательно. Это может привести к увеличению времени рендеринга и на *CPU*, и на *GPU*. Эта функция изначально поддерживается *UE*4, но по умолчанию отключена, так как некоторые функции рендеринга с ней не работают. Иногда ее использование может привести к снижению производительности графического процессора, поэтому важно протестировать ваш проект
Hidden and visible mesh optimization	Оптимизация сетки скрытых областей в *UE*4 позволяет визуализатору удалять пиксели, которые не будут видны из-за искажения линз *VR*-шлема, что в итоге экономит время *GPU*. По умолчанию оптимизации включены и не требуют от вас дополнительных действий для использования
Timewarp/reprojection	Обе функции *timewarp* (используется со средой исполнения *Oculus*) и *reprojection* (используется со средой исполнения *OpenVR*) позволяют рендерингу компенсировать потерянные кадры (если приложение не отрисует вовремя следующее обновление) вместо того, чтобы показать тот же кадр дважды. Обе среды выполнения перепроецируют предыдущий кадр в последний поворот головы пользователя, обеспечивая сглаживание (тонкости различий между этими двумя методами рассматриваются в главе 10). Как уже упоминалось, *timewarp* и *reprojection* обрабатываются соответствующими средами исполнения, и для их включения не требуется дополнительных действий. Обратите внимание, что разработчику не следует полагаться на эти функции для исправления низкой частоты кадров. Вместо этого следует использовать их в качестве запасного варианта для решения проблем с производительностью, которые находятся вне зоны вашего контроля
Front buffer rendering/ scaling racing	При выводе на дисплей в обычных 3*D*-приложениях, чтобы избежать «рваного» изображения (состоящего из разных кадров), изображения могут быть отрендерены во вторичном буфере [*back buffer*], затем выгружается в первичный/передний буфер [*front buffer*] для отправки на экран. Но это вызывает как минимум лишний кадр задержки при работе *VR*-приложений (детальное объяснения см. в главе 10). Чтобы избежать этого, большинство *SDK* (в основном мобильных) рендерят непосредственно в первичный буфер (буквально гонка пиксель за пикселем, построчное считывание из кадрового буфера на дисплей, отсюда выражение «гонка сканирования»). Данный эффект достигается из-за предсказуемости отдельных методов перепроецирования и прогнозирования, реализованных при визуализации кадра *VR*. *Oculus Mobile SDK* поддерживает такое поведение по умолчанию, но для *Google VR* ее требуется включить вручную

Особенности	Описание
Deferred/forward rendering	Существует два основных метода рендеринга 3*D*-сцены: прямой и отложенный. Изначально большинство игровых движков были основаны на прямом (*forward*) подходе из-за своей простоты и эффективности для малой сцены. Тем не менее многие движки перешли или теперь включают отложенный (*deferred*) подход, потому что его возможности шире для динамического освещения и некоторых дополнительных эффектов экранного пространства, таких как *SSRs* (*Screen scare reflections*). Однако, поскольку производительность чрезвычайно важна в *VR*, прямой рендеринг может быть более привлекательным вариантом. Для смарфонов прямой рендеринг является единственным вариантом, то есть это все, что вы имеете в арсенале для *Gear VR* и *Google VR*. Что касается настольных приложений, *UE*4 по умолчанию использует отложенный рендеринг. Но специально для этих приложений компания *Epic* сейчас работает над вариантом прямого рендеринга, который обещает существенный прирост производительности

1.1.1.2. Unreal Engine

Создание игр и других виртуальных сюжетов требует совместного использования разнообразных инструментов. Замечательно, что *UE*4 содержит большое число специализированных модулей, предоставляющих разработчику отличный набор инструментов. Таблица 1.5 описывает модули, используемые в этой книге.

Таблица 1.5. Модули *Unreal Engine*, задействованные в этой книге

Системы	Описание
Unreal Motion Graphics (*UMG*)	Система процедурно-генерируемого интерфейса с пользователем на основе данных в *UE*4. Для использования виджетов *UMG* можно использовать визуальный конструктор *UMG*, который позволяет легко создавать интерфейс с пользователем без написания какого-либо кода. *UMG* также поддерживает анимацию по ключевым кадрам и динамическое масштабирование интерфейса
Blueprints	Визуальный скриптовый язык *UE*4. Он позволяет провести быстрое прототипирование и не увязнуть в синтаксисе языка (в этой книге вы почти всегда будете использовать *Blueprints*)
Sequencer	Продвинутый кинематографический инструмент в составе *UE*4. Он позволяет создавать анимационные последовательности на основе ключевых кадров, а также вырезать и перемещать их в стиле традиционных инструментов нелинейного видеомонтажа
Persona	Редактор анимации *UE*4. Он имеет огромное количество функций, от возможности легко смешивать анимации до моделирования движения на основе инверсной кинематики

1.2. Лучшие практики

Область *VR* развивается очень бурно. Прежде чем мы определимся с правильными методами работы, придется поставить много экспериментов, проанализировать результаты и усвоить важные уроки. Часто ваши радужные надежды на то, что определенные функции или механизмы будут работать в *VR*, проваливаются при реальной работе. В то же время решения, о которых вы никогда не подумали бы, и поэтому реализуете и опробуете их в последнюю очередь, работают просто великолепно.

Следует заметить, что благодаря исследованиям и тестам, как *Oculus*, так и *Epic* выявили некоторые моменты, которые большинство найдет неудобными при работе с *VR*. На момент написания книги эти статьи можно были найти по ссылкам:

- *https://developer.oculus.com/documentation/*
- *https://docs.unrealengine.com/en-us/Platforms/VR/ContentSetup*

Руководство по лучшим практикам *UE*4 — отличное место старта для любого разработчика *Unreal Engine VR*, потому что многие из затронутых тем относятся к *UE*4 и оптимизации вашей работы с *VR*. Многие из них, такие как отключение *HZB* (*hierarchical Z-Buffer*) и экземплярный стерео-рендеринг (*instanced stereo rendering*), подробнее рассматриваются в главе 10. Если вы ищете общие подсказки по использованию *VR*, то обратитесь к руководству с лучшими практиками для *Oculus*.

Приведем некоторые общие соображения.

1. По возможности избегайте применения эффектов к экранному пространству. В лучшем случае они будут выглядеть плохо, а в худшем — могут стать причиной исчезновения стереоэффекта из-за несоответствия изображений, сгенерированных для разных глаз.
2. Стремитесь к средней частоте кадров немного большей, чем частота развертки целевого дисплея. Это гарантирует, что никакие «загвоздки» при расчете мира или рендеринге не вызовут серьезного дискомфорта (не полагайтесь на технологию перепроецирования).
3. Если требуется увеличить частоту кадров, рекомендуется уменьшить разрешение визуализации. Это можно сделать с помощью консольной команды `r.screenPercentage` *UE*4 (глава 10).
4. Движения головы пользователя должны быть под контролем камеры постоянно. Избегайте использования кинематографических углов камер и старайтесь применять асинхронную загрузку в меню или во время загрузки уровней. Избегайте тряски камеры и других эффектов, которые перемещают камеру без контроля пользователя.
5. Не перекрывайте поле зрения игрока; это вызывает дискомфорт при вращении.
6. Избегайте резких ускорений; они создают визуально-вестибулярное несоответствие (несоответствие между тем, что видят глаза, и тем, что «говорит» вестибулярный аппарат) и вызывают тошноту (подробнее об этом в главе 9 «Перемещение в *VR*»).

Повторюсь, оба документа содержат отличную информацию для любого *VR*-разработчика. Я советую вам прочитать их целиком.

1.3. Заключение

Теперь вы знакомы с некоторыми терминами, которые будут использоваться в этой книге, включая различную *VR*-аппаратуру, поддерживаемую *UE*4, и некоторые библиотеки, необходимые для создания *VR*-сюжетов (даже если вы не будете взаимодействовать непосредственно с ними). Вы получили представление о методах, применяемых для оптимизации виртуальной реальности (чтобы узнать больше, ознакомьтесь с главой 10). И, наконец, мы рассмотрели некоторые из лучших практик в сфере виртуальной реальности.

ГЛАВА 2

НАСТРОЙКА ШЛЕМА ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ

UE4 серьезно облегчает создание *VR*-сюжетов, но, как и во многих других случаях, очень полезно понимать, как настроить свой проект, чтобы избежать распространенных ошибок.

В этой главе вы узнаете, как настроить свой проект для работы с желаемым *VR*-шлемом, а также основы работы с *VR*.

2.1. *Gear VR*

Так как *Gear VR* работает на устройствах под управлением операционной системы *Android*, большинство шагов в данном разделе касаются настройки вашего проекта под *Android*-устройство. Кроме того, вам необходимо настроить несколько параметров, чтобы выложить ваш продукт в магазин *Oculus*.

Документация компании *Epic* содержит описание настройки, позволяющей *UE* обнаружить ваше устройство и запустить на нем приложение. Это описание доступно по адресу:

https://docs.unrealengine.com/en-us/Platforms/Gear VR/Prerequisites

Убедитесь, что вы выполнили следующие действия.

1. Включили режим разработчика на вашем устройстве.
2. Включили *USB*-отладку на вашем устройстве.
3. Установили пакет разработчика (*SDK*) *Android*.
4. Сгенерировали и скачали файл подписи *Oculus Signature File* (*OSIG*), который позволяет вашему приложению работать на устройстве за пределами магазина *Oculus*.
5. Скопировали *OSIG* в папку *Engine/Build/Android/Java/assets* в раздел вашего *UE*4 активной версии.

2.1.1. Настройка проекта *Gear VR*

После настройки вашего *Android*-устройства и *UE*4, вам необходимо настроить сам проект. Создайте новый пустой *Blueprint*-проект: укажите в качестве целевого устройства *Mobile/Tablet*, режим масштабирования установите «*Scalable 3D or 2D*» и не включайте стартовый контент (рис. 2.1). Эти настройки гарантируют, что некоторые дополнительные графические функции *UE*4 по умолчанию отключены (точные характеристики можно посмотреть в главе 10 «*VR*-оптимизация»).

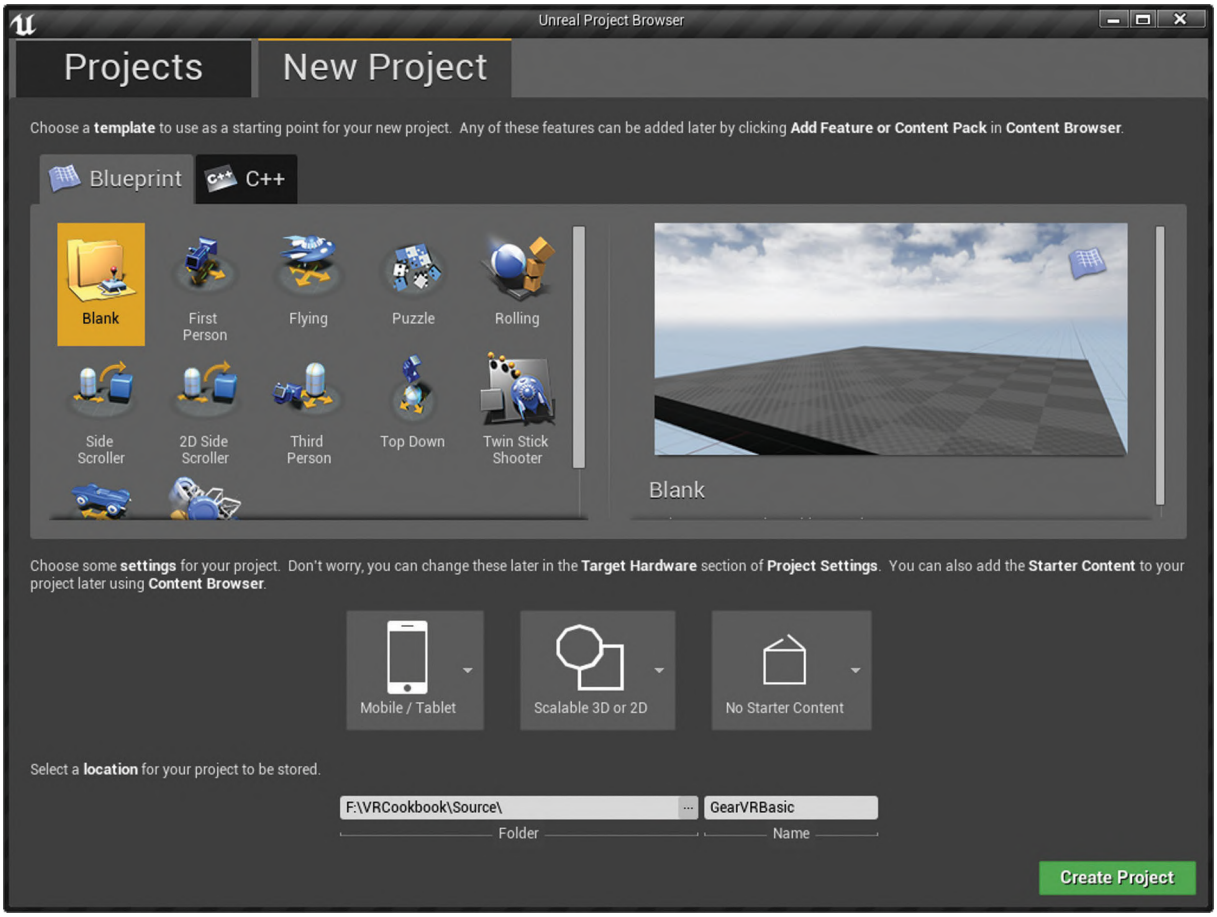

Рис. 2.1. Создание проекта для *Gear VR*

После создания проекта необходимо настроить структуру разделов и создать ресурсы, необходимые для создания простой игры для *Gear VR*.

1. Создайте две папки в корневой папке проекта. Назовите их *Blueprints* и *Materials*.
2. Создайте папку *HUD** (*heads-up display*) и поместите ее в папку *Blueprints*.
3. Создайте новый *Game Mode Base* (режим игры): нажмите правой кнопкой мыши в папке *Blueprints*, под разделом *Create Base Asset* выберите *Blueprint Class*, затем в появившемся окне выберите *Game Mode Base*.
4. Дайте ему имя *GearVRGM*.
5. Создайте новый *HUD Blueprint*, нажав правой кнопкой мыши в папке *HUD* и выбрав *Blueprint Class*. В появившемся окне разверните список *All Classes* и найдите *HUD*. Выберите только те из них, которые называются *HUD* и нажмите на кнопку *Select* (рис. 2.2).

* *Head-up display* (*HUD*) — индикатор на лобовом стекле (ИЛС) . — *Прим. ред.*

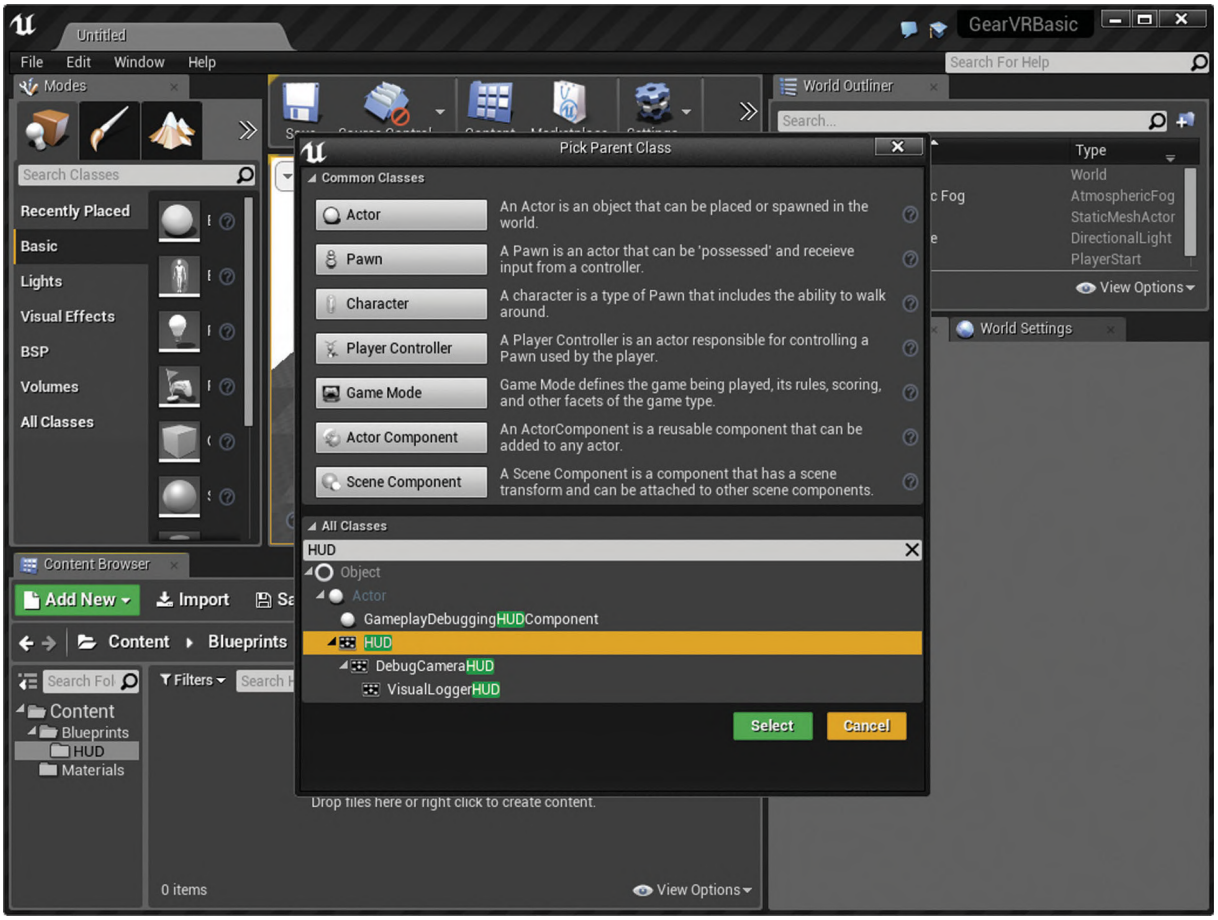

Рис. 2.2. Создание *Gear VR HUD*

6. Назовите вновь созданный *HUD GlobalMenu*. Он будет отражать вашу загрузочную панель и определять, когда открывать глобальное меню *Gear VR*.
7. Нажмите правой кнопкой мыши в папке *Materials* и создайте *Material* под названием `UICircle`. Его вы будете анимировать, чтобы создать меню прогресса загрузки (заметим, что рекомендация для *Gear VR*-разработчиков не требует, чтобы индикатор был кругом. Но это выглядит красиво и является хорошим упражнением для демонстрации возможностей *UE4 Material Editor*).

Ваш проект должен соответствовать рис. 2.3.

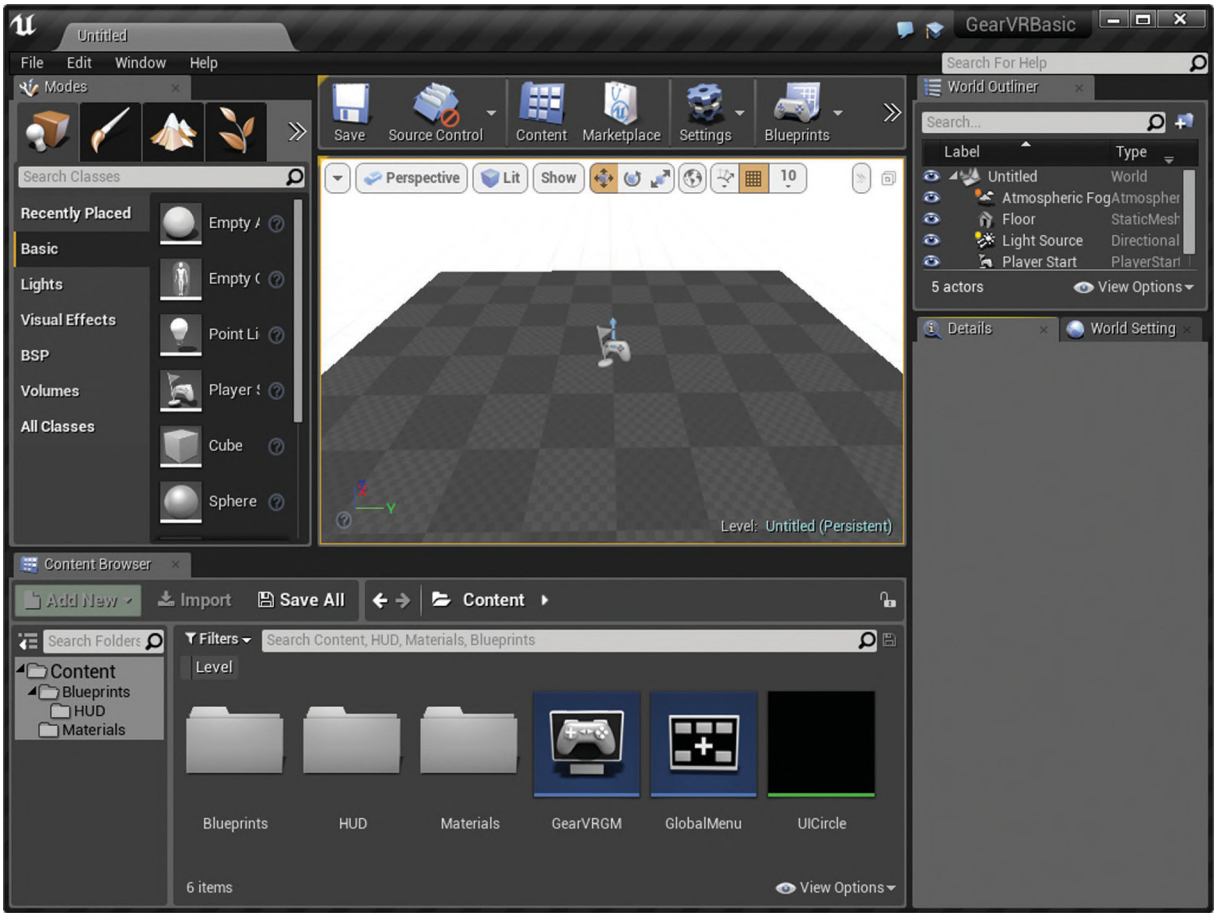

Рис. 2.3. Структура проекта *Gear VR*

Далее установим настройки проекта, чтобы приложение работало в *Gear VR*.

1. Откройте *Project Settings (Edit ⇒ Project settings* в меню) и перейдите в секцию *Android* в разделе *Platforms*. Здесь вы настраиваете ваш *Android*-проект.
2. Нажмите на кнопку *Configure Now*, размещенную на ленте в верхней части окна.
3. В секции *APKPackaging* установите *Minimum SDK* и *Target SDK Version* в значение 19.
4. В этой же секции проверьте, что включено *Package game data inside .apk?* Это гарантирует, что **.obb** файл (файл, который содержит большие двоичные данные, такие как текстура) упакован внутри файла **.apk** (*Android Application Package*). Это необходимо, потому что *Oculus Store* принимает только единственный **.apk** файл.
5. В секции *Advanced APKPackiging* проверьте, что включено *Configure the AndroidManifest for deployment to Gear VR*. Это нужно, чтобы сообщить устройству, что необходимо открыть ваше приложение как *Gear VR*-приложение. Эти настройки показаны на рис. 2.4.

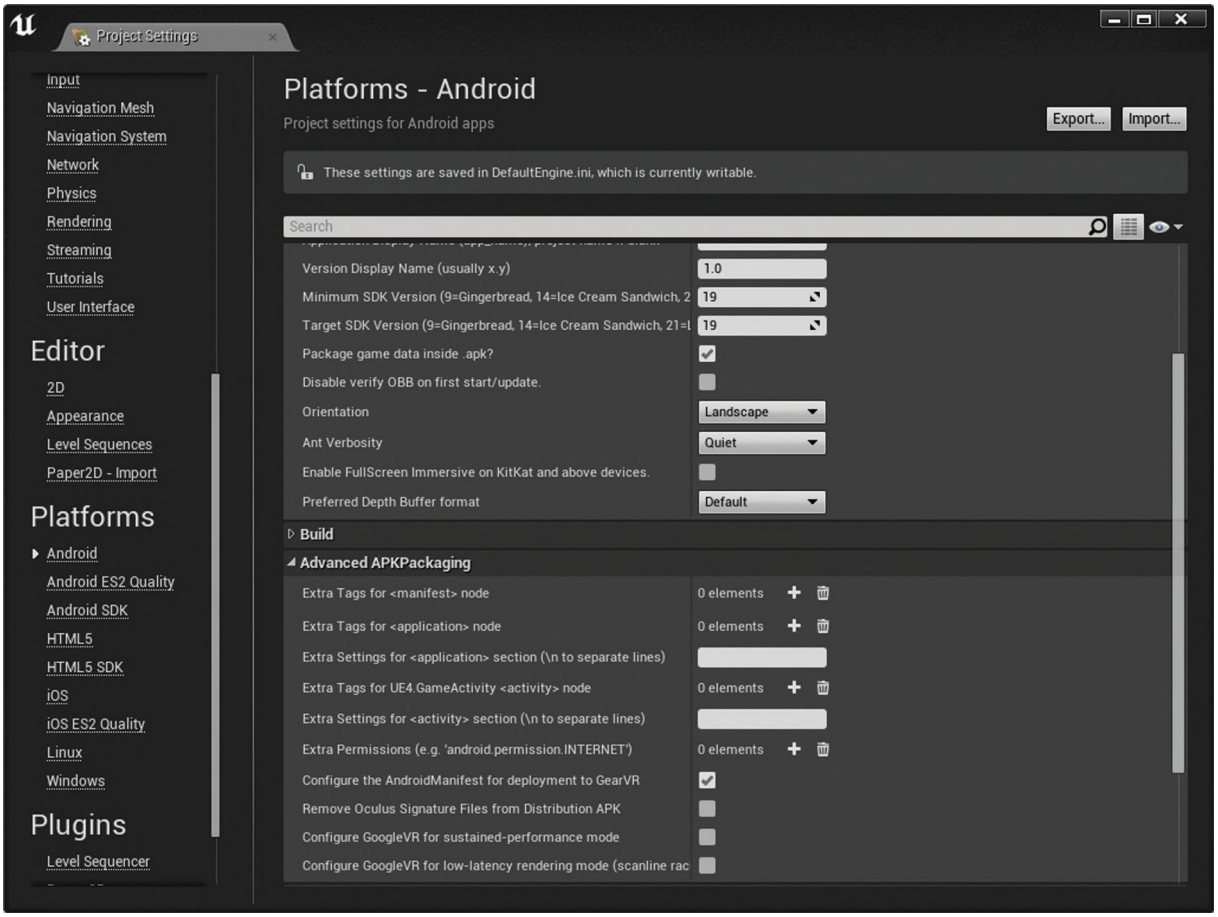

Рис. 2.4. *Gear VR*: настройка параметров *Android*

6. Удалите стандартный джойстик в *UE*4. Для этого перейдите в окно *Input Settings* в секции *Engine* в разделе настроек проекта *Project Settings*.
7. Раскройте список для *Default Touch Interface* и выберите функцию *Clear* (рис. 2.5).

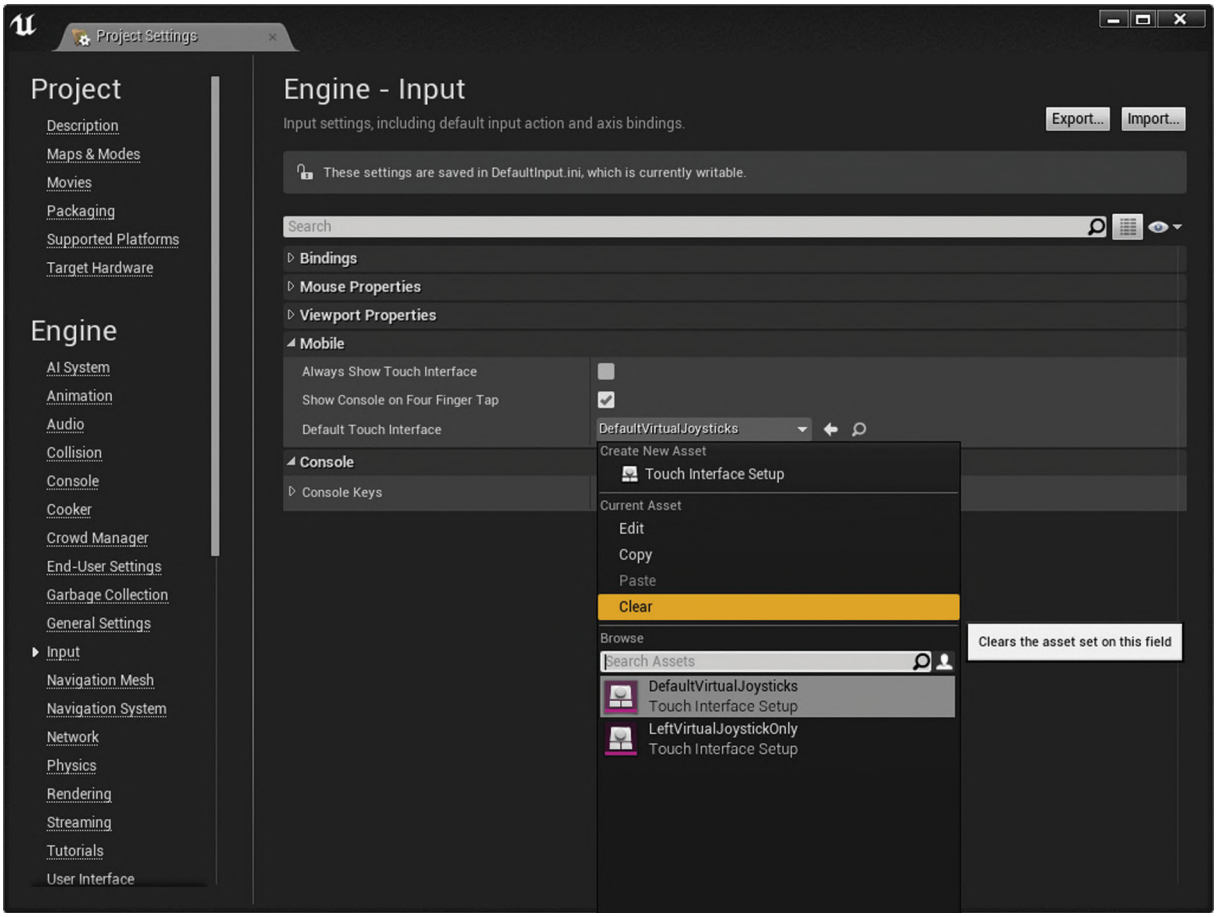

Рис. 2.5. *Gear VR*: удаление стандартного джойстика

8. Чтобы указать использование настраиваемого *Game Mode Base*, который вы создали, перейдите в *Project Settings*, затем в секцию *Maps & Models* в разделе *Project*.
9. В раскрывающемся списке *Default GameMode* выберите *GearVRGM*, который вы создали ранее (рис. 2.6).

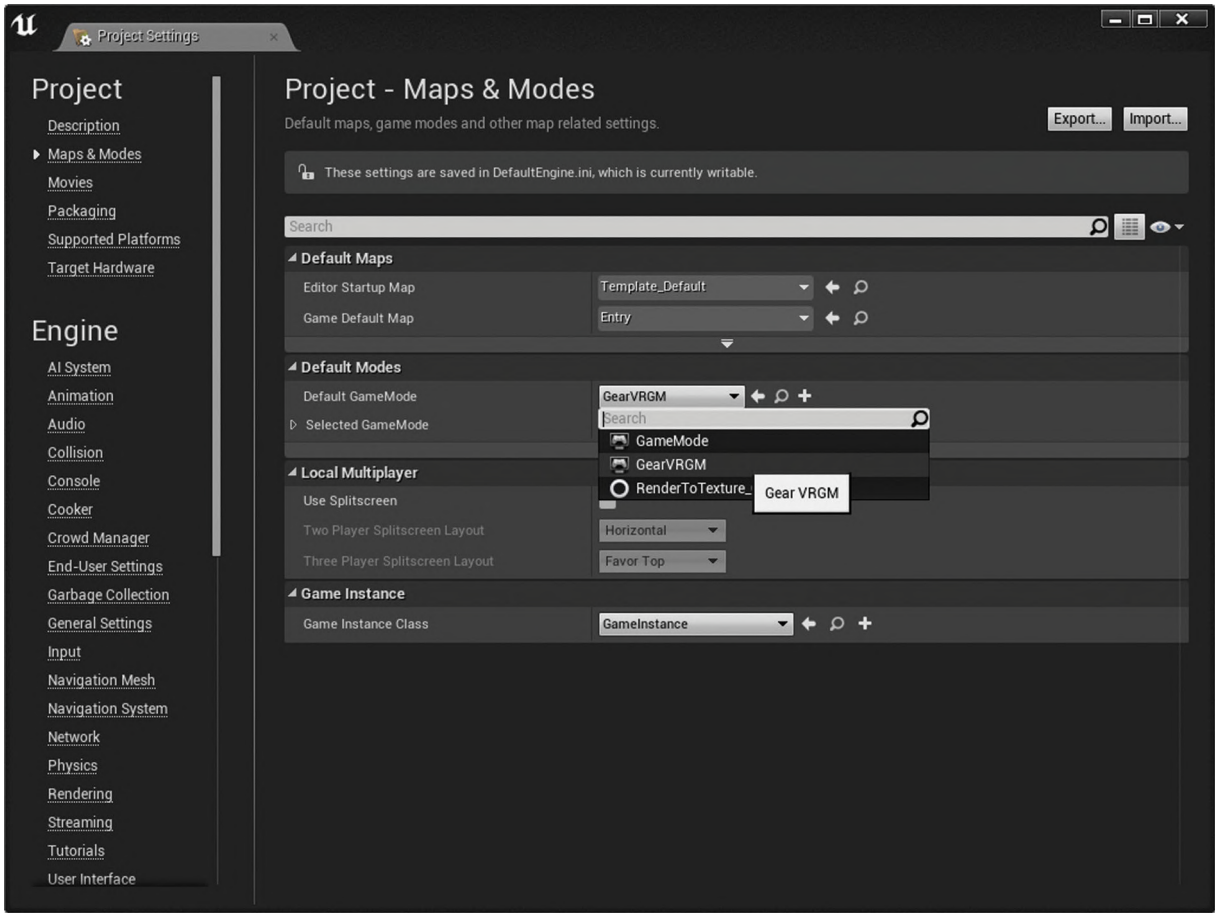

Рис. 2.6. *Gear VR*: активация нового *Game Mode* по умолчанию

2.1.2. Настройка глобального меню *Gear VR*

Чтобы приложение приняли в *Oculus Store*, вам необходимо настроить открытие *Gear VR*-меню по длительному нажатию на кнопку *Home*. По умолчанию короткое нажатие кнопки *Home* предлагает пользователю выйти из приложения, а долгое открывает глобальное меню. Однако если вы хотите сделать переход к глобальному меню или создать вашу собственную функцию по короткому нажатию, необходимо переопределить эти функции и создать собственное событие, которое открывает глобальное меню по длительному нажатию клавиши *Home*.

> **Заметка**
>
> В нашем примере используется *HUD Blueprint*, что удобно для такого типа интерфейсов. Однако в первоначальной версии *Unreal Engine* 4.13, разработчики убрали совместимость *HUD Blueprint* и *VR*. Если вы используете эту версию *Unreal Engine*, вы должны заменить W*idget Component* в классе *Pawn* на *HUD Blueprint* (см. главу 6, «Графика движения и 2*D*-интерфейсы в *Unreal*»).

1. Откройте ваш *GearVRGM Game Mode Base Blueprint* и преобразуйте его *HUD Class* вашим глобальным меню (рис. 2.7.). Это укажет движку использовать настроенный вами *HUD*.

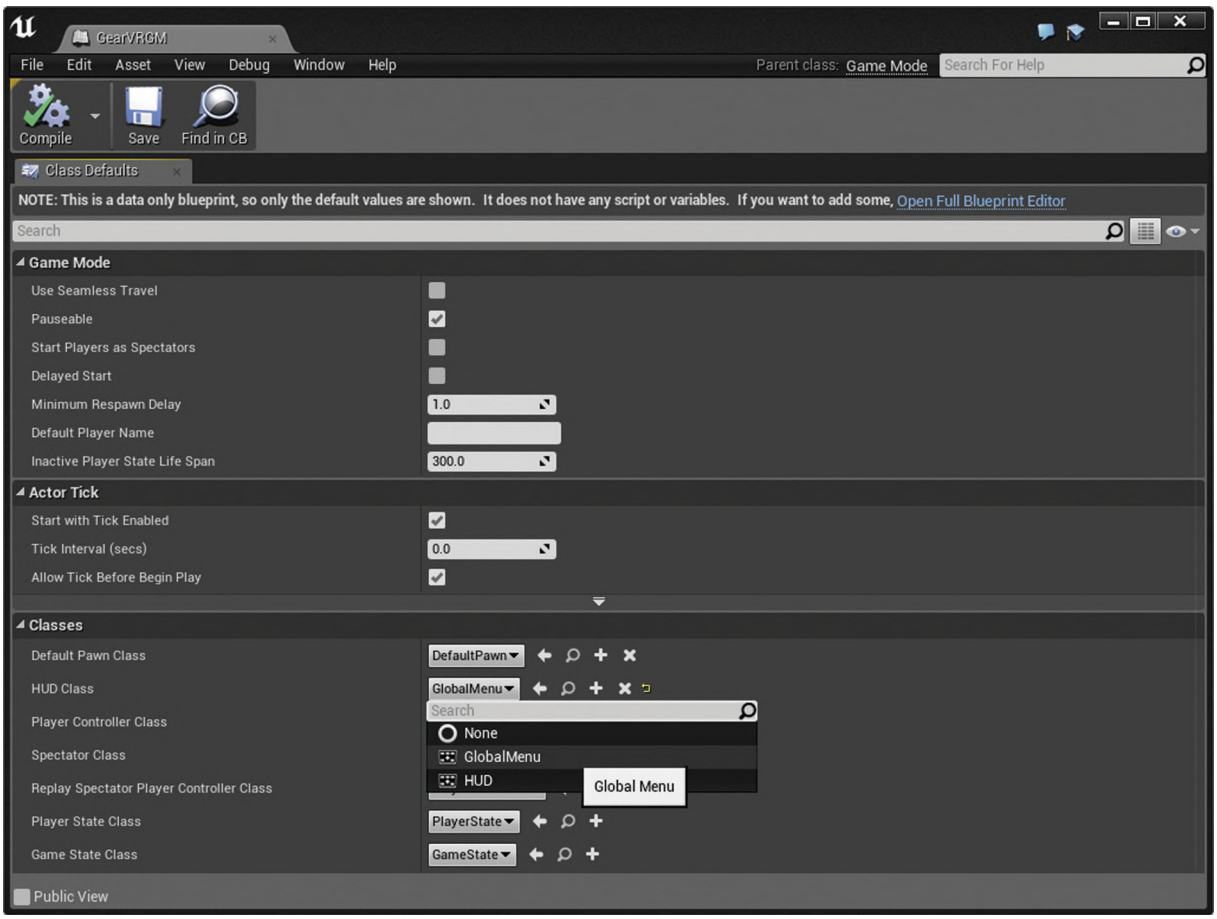

Рис. 2.7. *Gear VR*: настройка *Game Mode* для использования созданного *HUD*

2. Откройте *GlobalMenu Blueprint* и создайте новую функцию `OpenMenu`. В ней создайте узел *ExecuteConsoleCommand* и подключите его к входу выполнения функции.
3. В поле ввода команды узла наберите `OVRGLOBALMENU`; это вызовет открытие меню *Gear VR* при каждом вызове этой функции (рис. 2.8).

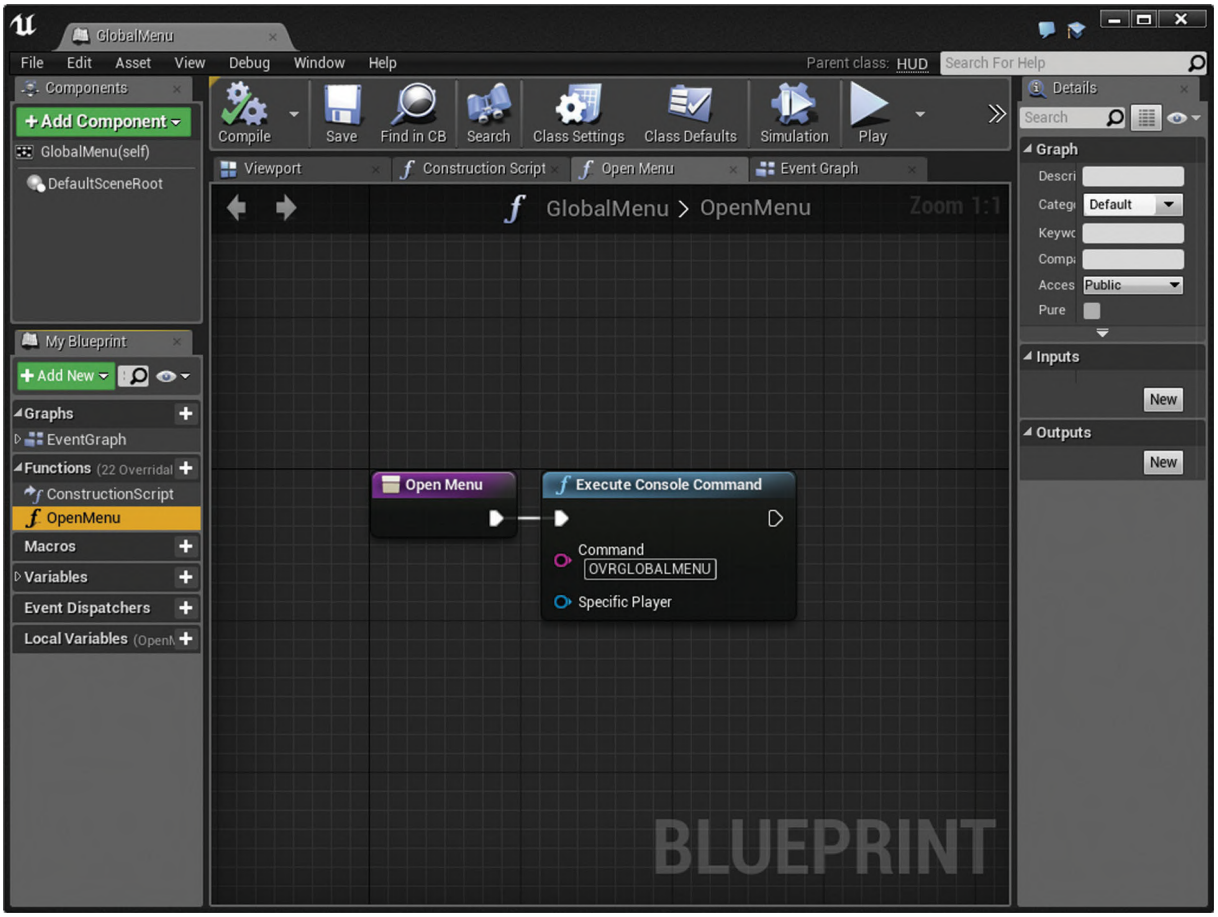

Рис. 2.8. *Gear VR*: создание функции *OpenMenu* для *HUD*

4. Создайте три новые переменные.

a. Задайте тип первой `Timer Handle` и назовите ее `BackButtonTimer`.

b. Задайте тип второй `Material Instance Dynamic` и назовите ее `CircleMat`.

c. Задайте третьей переменной тип `Float` и назовите ее `CircleRadius`. Значение по умолчанию поставьте `30`. Не забудьте, что если у вас не отображается поле для значения по умолчанию, то скомпилируйте ваш *Blueprint* (*Toolbar* ⇒ *Compile*).

5. Создайте новое событие `AndroidBack`. Для этого нажмите правой кнопкой мыши в *Event Graph* и найдите `AndroidBack`. (Не забудьте проверить, что у вас выключен *context-sensitive*-поиск, иначе вам будут отображаться не все существующие узлы).

6. Создайте новый узел *SetTimerByFunctionName* и соедините его с контактом `Pressed`.

7. В этом узле в качестве имени вызываемой функции укажите `OpenMenu`. Это заставляет таймер вызывать функцию, настроенную ранее.

8. Установите в `Time` значение `0.75`; это требует *Oculus* в руководстве разработчиков.

9. Создайте новый сеттер* для переменной `BackButtonTimer` (с вкладки `Variables` мышкой перенесите его в *Event Graph* и выберите сеттер или создайте его от выхода выполнения `SetTimerByFunctionName`) и соедините его со входом выполнения и `Return Value`.
10. Создайте новый геттер для `BackButtonTimer`, создайте узел *Clear and Invalidate Timer By Handle* и подключите его к контакту `Released` события `AndroidBack` (рис. 2.9.). Это создаст новый таймер, который начнет вызываться каждые `0.75` секунды, когда вы нажмете кнопку `AndroidBack`; таймер выключится, когда вы отпустите кнопку.

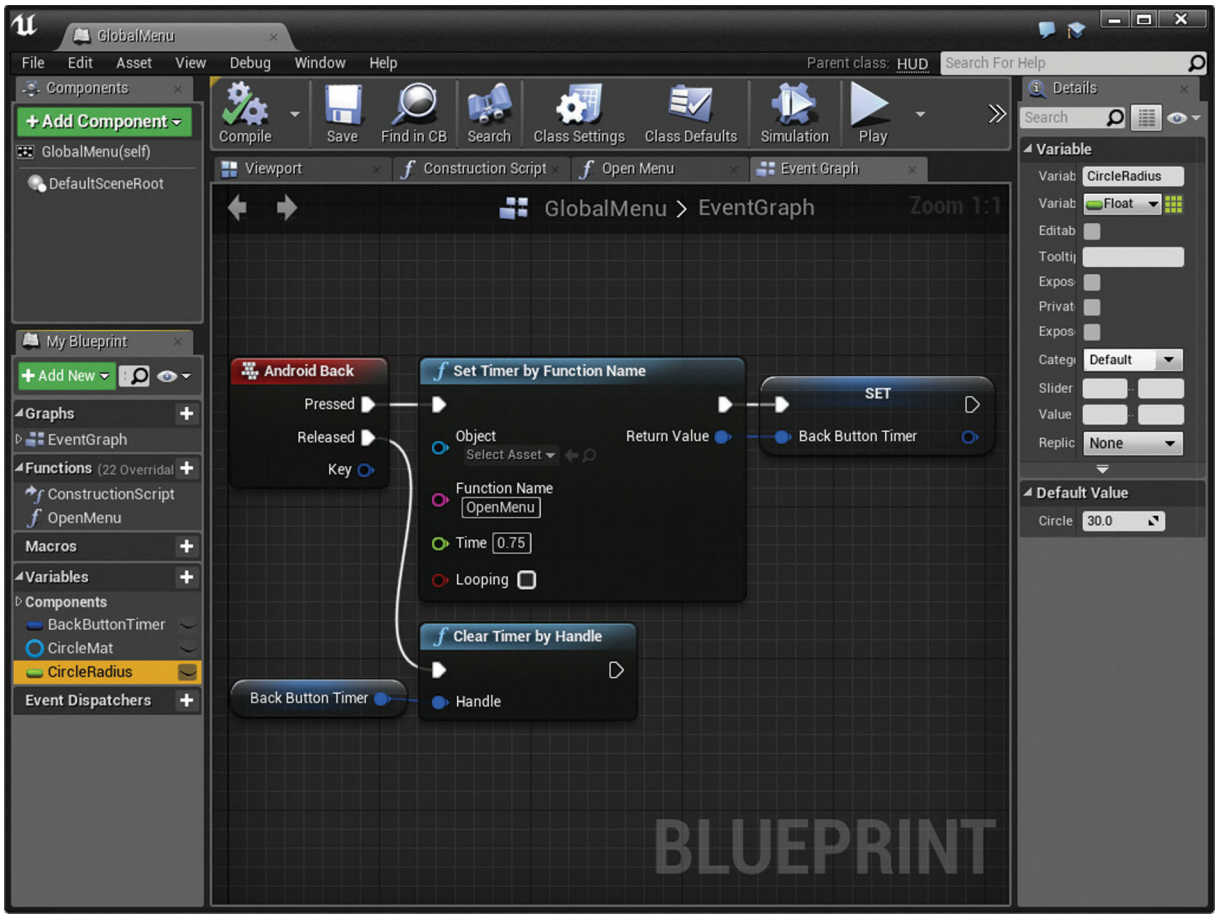

Рис. 2.9. *Gear VR*: создание таймера для открытия меню после долгого нажатия кнопки *Back*

11. Создайте вызов функции `GetOwningPlayerController` нажатием правой кнопкой мыши в `Event Graph`.
12. От `Return Value` этого узла вызовите событие `EnableInput`. Убедитесь, что `GetOwningPlayerController` соединен со входом `Player Controller`.
13. Соедините контакт вызова события `BeginPlay` с `EnableEvent`. Это гарантирует, что событие `AndroidBack` будет в состоянии захватить входное событие.

* Здесь и далее сеттером и геттером названы методы доступа для установки и получения значения полей соответственно, традиционно называемые в англоязычных источниках «сеттер» и «геттер». — *Прим. пер.*

14. После `EnableEvent` добавьте вызов нового события `ExecuteConsoleCommand`, задайте `gearvr.handlebackbutton` значение `0` в качестве параметра команды. Это гарантирует, что вы сможете настроить свою функциональность клавиши *Back*.
15. Соедините вход `Specific Player` с `GetOwningPlayerController`.
16. После `ExecuteConsoleCommand` добавьте вызов функции `CreateDynamicMaterialInstance` и в `Source Material` выберите материал `UICircle`, который вы создали раннее. Это создает динамический материал, который вы сможете анимировать для загрузочного круга.
17. От контакта `ReturnValue` узла *CreateDynamicMaterialInstance* создайте сеттер переменной `CircleMat` (рис. 2.10.).

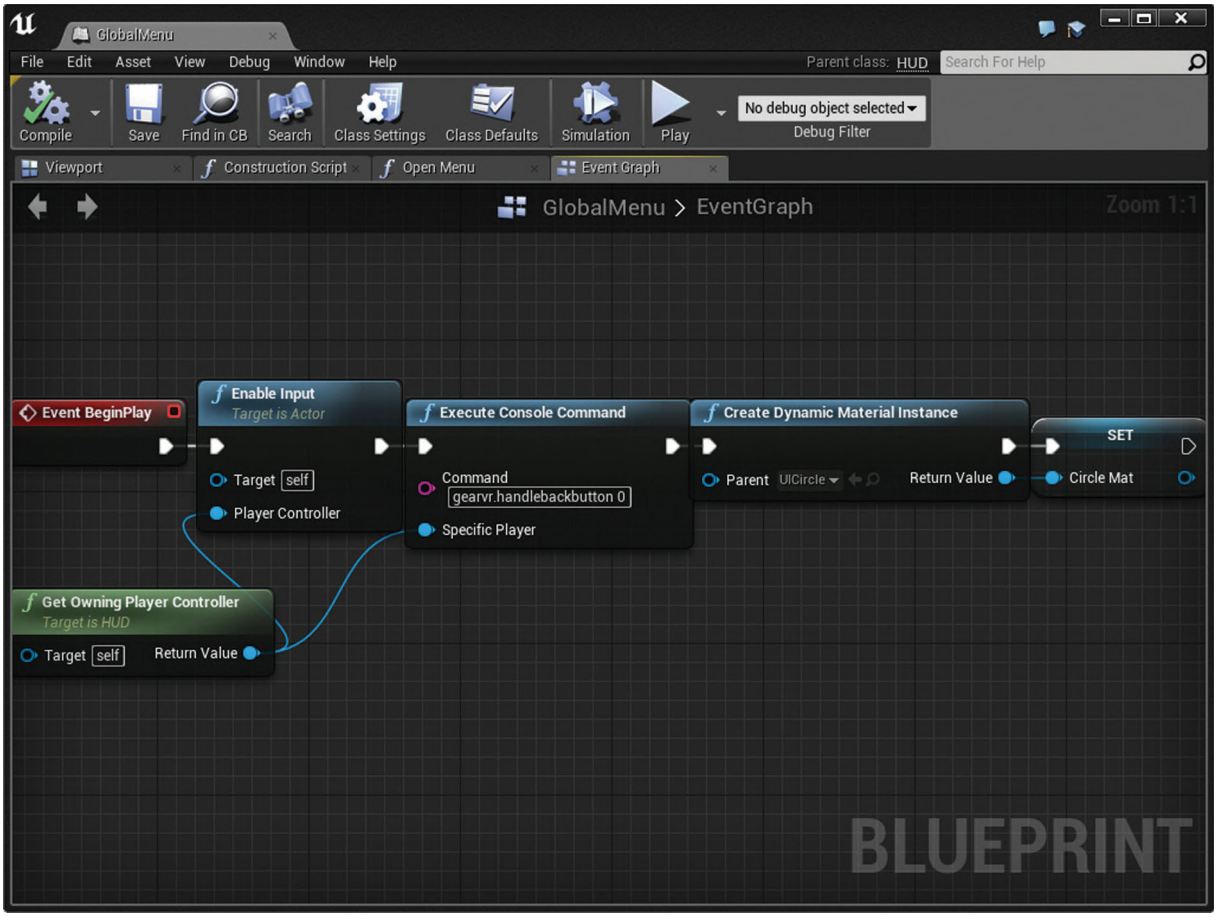

Рис. 2.10. *Gear VR*: включение ввода и настройка динамичного материала для *HUD*

Следующая задача, с которой нам предстоит справиться, — это анимация и прорисовка материала на экране при нажатии кнопки *Back*.

1. Нажмите правой кнопкой мыши в *Event Graph* и создайте узел *EventReceiveDrawHUD*.
2. Создайте узел *Branch* и соедините его с входом выполнения предыдущего узла.
3. Создайте новый геттер переменной `BackButtonTimer`, вызовите функцию `IsTimerActiveByHandle` и соедините `Return Value` с *Condition* узла *Branch*. Это позволит пользовательскому интерфейсу перезапускать отрисовку только при нажатии кнопки *Back*.
4. Создайте геттер переменной `CircleMat` и вызовите событие *SetScalarParameterValue*.
5. В `Parameter Name` введите значение `PercentComplete`. Это соответствует скалярному параметру материала `UICircle`, который мы скоро создадим.
6. Соедините выход `True` узла *Branch* с `SetScalarParameterValue`.
7. Создайте еще один геттер переменной `BackButtonTimer` и вызовите функцию `GetTimerElapsedTimeByHandle`.
8. Перетащите `Return Value` этой функции и создайте новый узел `Float/Float`. Введите `0.75` во втором контакте. Это нормирует затраченное время между 0 и 1.
9. Соедините выход этого узла с входом `Value` узла *SetScalarParametrValue*.
10. Создайте новый геттер для переменной `CircleMat` и вызовите функцию `DrawMaterial`, соединенную с вызовом узла *SetScalarParameterValue*.
11. Для входных данных `Size X` и `Size Y` вызовите функцию `Int * Float`, установив `0.5` во втором контакте. Это делается для определения центра экрана.
12. Выходы обоих узлов `Multiply` соедините с узлами `Float — Float`, установив во втором контакте геттер для переменной `CircleRadius`. Это исправляет размеры элементов пользовательского интерфейса и гарантирует, что это фактический центр экрана.
13. Соедините вычисленные `Size X` и `Size Y` со `Screen X` и `Screen Y`, соответственно, узла `DrawMaterial`.
14. Создайте новый геттер для переменной `CircleRadius` и вызовите функцию `Float * Float`, установив во втором контакте значение `2`, затем передайте его в `ScreenW` и `ScreenH` узла *DrawMaterial*.
15. Установите `MaterialUWidth` и `MaterialUHeight` равными `1`.

Настроенное вами событие должно выглядеть так, как показано на рис. 2.11.

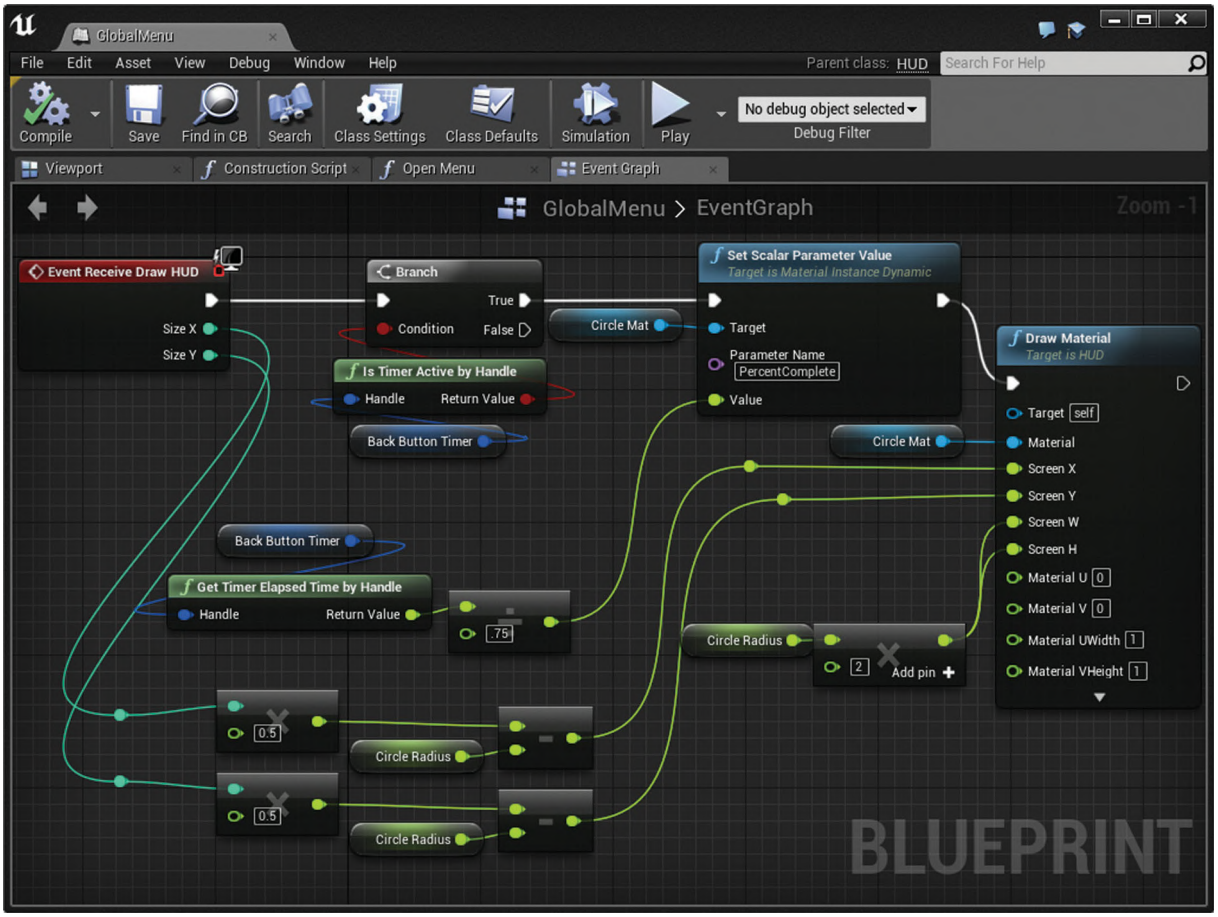

Рис. 2.11. *Gear VR*: анимация и прорисовка материала *HUD*

2.1.3. Глобальное меню изменения материалов *Gear VR*

В идеале, когда пользователь долго нажимает на кнопку *Back*, должен происходить некий отклик, чтобы пользователь знал, что происходит.

Для этого мы создадим индикатор загрузки, который похож на тот, что был у *Oculus*.

1. Откройте `UICurcle` созданный ранее материал из папки *Materials*.
2. На панели `Details` установите `User Interface` в `Material Domain`; это ограничивает входы и выходы узлов, но у вас все равно будет все необходимое.
3. В той же панели установите `Translucent` в `Blend Mode`, так как у вас будет два кольца: кольцо прогресса и фоновое прозрачное кольцо.
4. Создайте новый узел *TextureCoordinate* (нажмите правую кнопку мыши и найдите его по названию или зажмите `U` и нажмите левую кнопку).
5. Создайте новый узел *VectorToRadialValue*; это поможет вам создать круг.
6. Если вы соедините выход `TexCoord` с входом `Vector` узла `VectorToRadislValue` и соедините `Liner Distance` с `Final Color` вашего материала, вы заметите, что это создаст круговой

градиент, выходящий из верхнего левого угла. Чтобы перевести это в центр вашего материала, необходимо отмасштабировать и переместить вход, который вы даете узлу *VectorToRadialValue*.

a. Добавьте узел *ConstantBiasScale* и на панели *Details* укажите смещение (`Bias`) `-0.5` и масштаб (`Scale`) `2.0`.

b. Чтобы увидеть эффект от этого, установите `ConstantBiasScale` между `TexCoord` и `VectorToRadialValue` и соедините с ними.

c. Вы должны увидеть круговой градиент из центра материала.

7. Отключите контакт `Liner Distance` от `Final Color` и соедините `Vector Converted to Angle` с `Final Color`. То, что у вас выйдет, должно быть похоже на показанное на рис. 2.12.

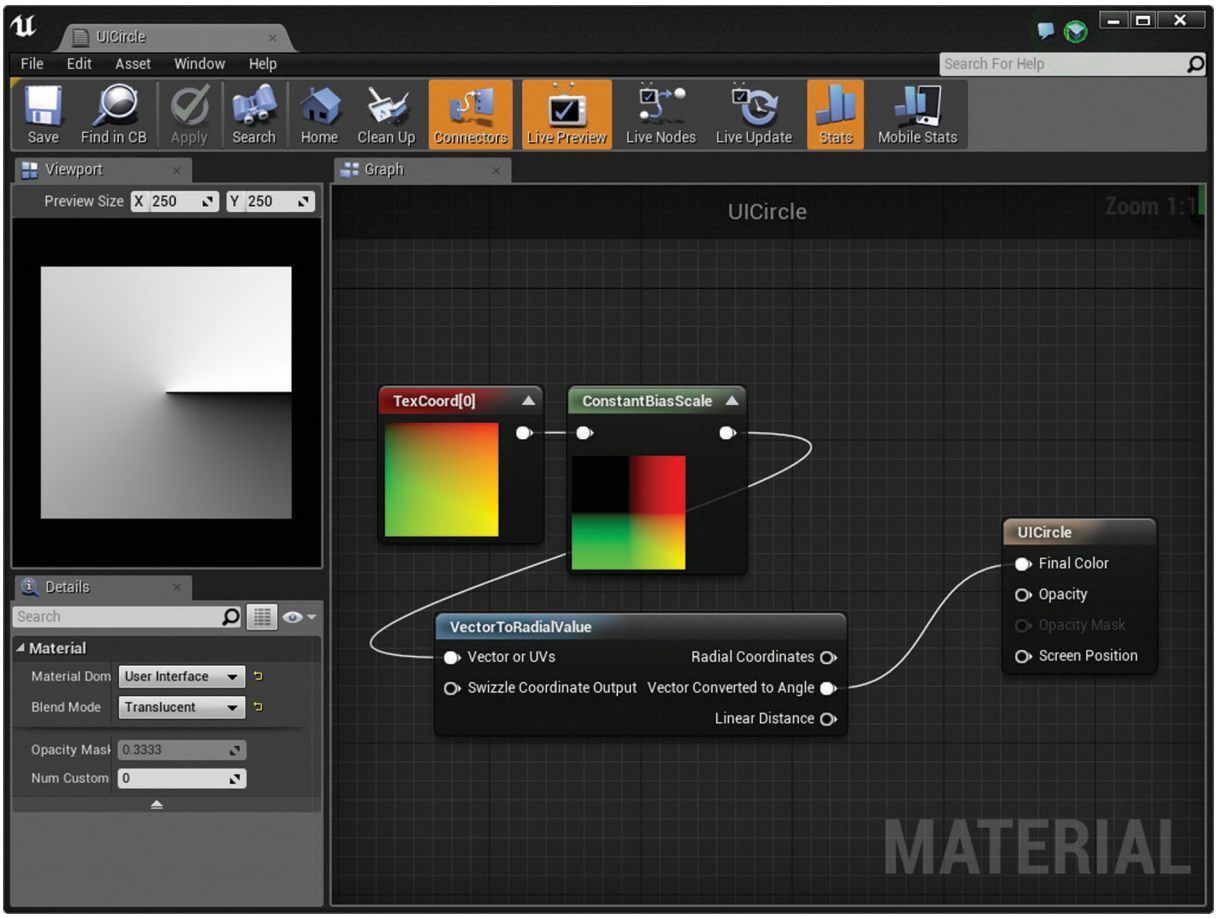

Рис. 2.12. *Gear VR*: материал *HUD* — простой градиент

Вы должны увидеть, как новый градиент поможет вам создать круговой знак.

Одна проблема связана с текущим градиентом. Она состоит в том, что он начинается справа, мы же хотим, чтобы он начинался сверху и при анимации двигался, подобно часовой стрелке.

8. Добавьте узел *Swizzle* и соедините его между `ConstantBiasScale` и `VectorToRadialValue`. Убедитесь, что соединены `XY` и `YX`.

9. Теперь, когда ваш материал развернут, вы заметите, что он перевернут вверх ногами. Чтобы исправить это, создайте узел `OneMinus` и поставьте его между `TexCoord` и `ConstantBiasScale`. Это транспонирует `TexCoord` и развернет градиент в правильном направлении.
10. Чтобы контролировать процент выполнения, создайте новый узел *Scalar Parameter* и назовите его `PercentComplete` на панели *Details*.
11. От этого узла создайте узел *Add*, соедините его с `Vector Converted To Angle` узла *VectorToRadialValue* вторым входом.
12. Создайте новый узел *Floor*. Это позволит вам менять то, что ниже 1, до черного, а все, что выше, до белого. Мы воспользуемся этим позже.
13. Теперь соедините выход *Floor* с `Final Color`; результат должен быть похож на рис. 2.13. (Поменяйте значения `PercentComplete` и посмотрите, что изменится.)

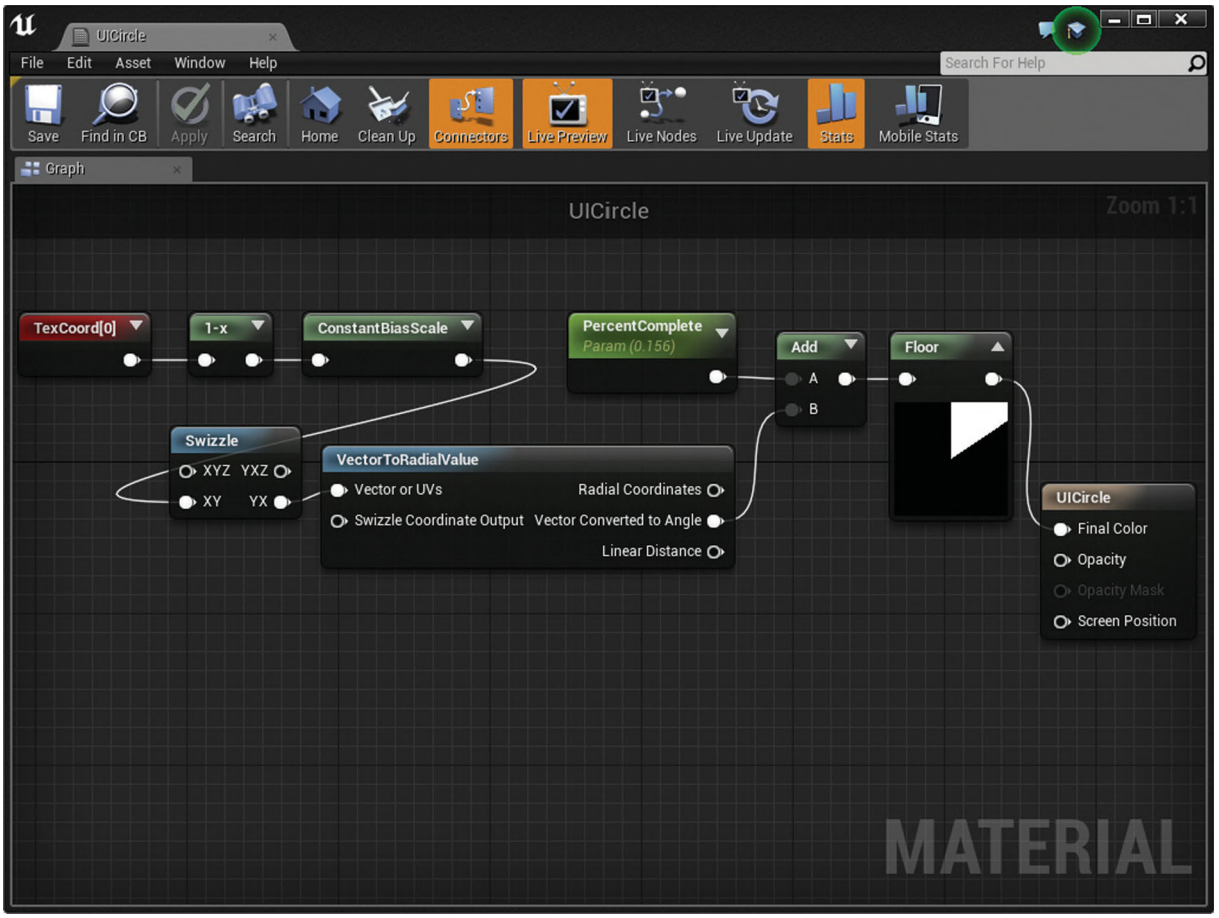

Рис. 2.13. *Gear VR*: материал *HUD* — анимация вращения

Таким образом мы создали первый шаблон. Теперь создадим круг. Для этого мы сделаем два круга: внешний круг и внутренний, который вы будете вырезать из внешнего, чтобы сделать кольцо.

1. Создайте узел *OneMinus* и соедините его с `Linear Distance` для инвертирования градиента.
2. Создайте внутренний круг первым, добавив узел *Substract* и соединив его с выходом *OneMinus*.

a. Создайте еще один узел типа `Scalar Parameter` и, назовите его `Width` и задайте значение `0.4`. Он будет использоваться для контроля ширины кольца.

b. Соедините этот параметр со вторым входом узла *Subtract*.

3. Создайте новый *Ceil* узел, который будет округлять до `1` любое значение выше `0`.
4. Добавьте новый узел *Clamp*, чтобы обеспечить ограничение *Ceil* от `0` до `1`.
5. Чтобы создать внешний круг, от узла *OneMinus* добавьте еще один узел *Ceil* для преобразования градиента в черно-белый.
6. Все, что осталось сделать, это вычесть из значения узла *Ceil* внешнего круга значение *Clamp* внутреннего круга.
7. Чтобы закончить шаблон кольца, добавьте узел *Multiply* и соедините его с выходами *Floor* и *Substract*. В результате при изменении `PercentComplete` вы получите анимацию кольца.
8. Соедините выход *Multiply* с `Opacity` материала и отключите все, что связано с узлом *Final Color*. У вас должно получиться что-то похожее на рис. 2.14.

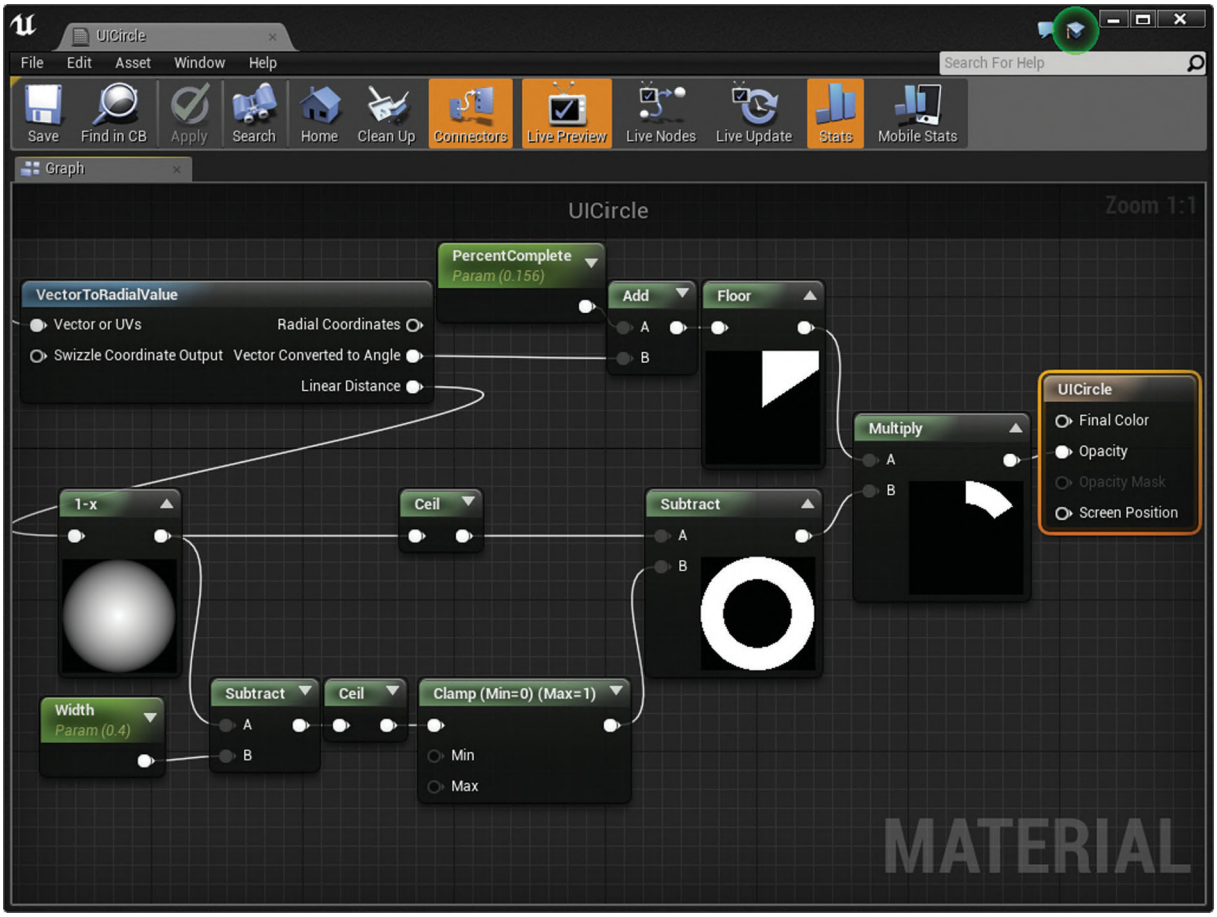

Рис. 2.14. *Gear VR*: материал *HUD* — анимация шаблона и кольцо

9. Чтобы контролировать цвет кольца, добавьте новый узел *Vector Parameter* в граф и назовите его `Color`.
10. Установите нужный вам цвет по умолчанию (у нас будет установлен приятный голубой цвет `R = 0.266, G = 0.485, B = 0.896`) и соедините его с *Final Color*.
11. Вы можете остановиться здесь, потому что материал уже настроен; однако было бы хорошо сделать, чтобы фон кольца подсвечивался.

a. Создайте новый узел типа `Scalar Parameter` и назовите его `MinTransparency`. Установите его значение `0.1`.

b. Соедините его с новым узлом *Add*, соединив второй вход с *Floor*.

12. Создайте новый узел *Clamp*. В заключение, соедините выход этого узла с первым входом *Multiply*, который вы использовали для комбинации двух шаблонов. У вас должен получиться материал, похожий на рис. 2.15.

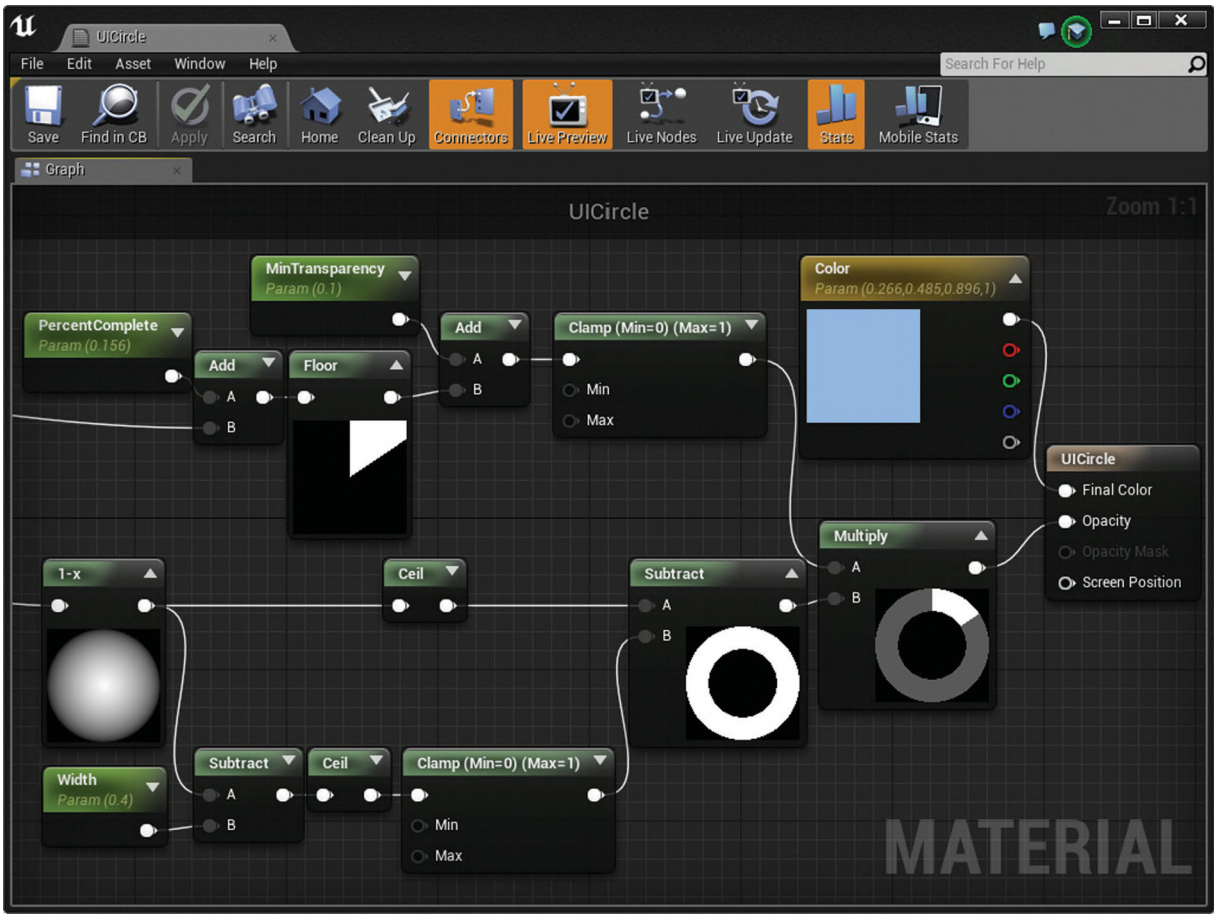

Рис. 2.15. *Gear VR*: материал *HUD* — анимация шаблона с прозрачностью

Теперь вы можете соединить ваше *Android*-устройство с компьютером, выбрать его во вкладке *Launch* и, дождавшись запуска, зажать кнопку *Back* на вашем контроллере и увидеть анимированный круг, после чего вы окажетесь в глобальном меню (рис. 2.16.).

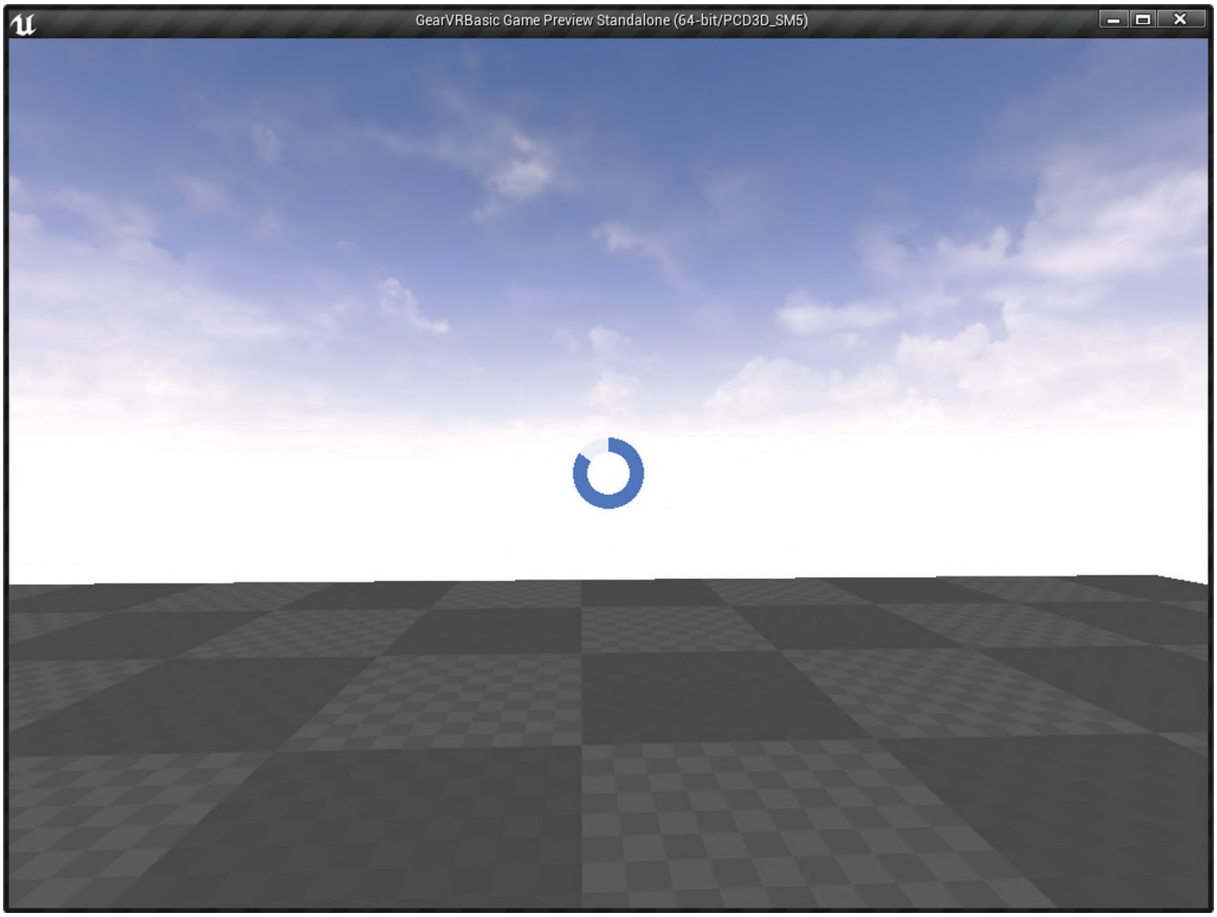

Рис. 2.16. *Gear VR: материал HUD в игре*

2.2. *Rift* и *Vive*

В отличие от *Gear VR*, с *Rift* и *Vive* легко настраивать проект и начинать работу. Основная причина — это то, что вы можете разворачивать приложение в системе, установленной на вашем рабочем месте, что сокращает время запуска.

Аналогично *Gear VR*, разработчики предоставляют подробную документацию по настройке *Rift*, которую вы можете найти по адресу: *https://docs.unrealengine.com/en-US/Platforms/Oculus/QuickStart*

Настройку для *Vive* можете найти по адресу:

https://docs.unrealengine.com/en-US/Platforms/SteamVR/QuickStart

Чтобы разрабатывать под *Rift*, вам необходимо:

1) установить *Oculus Rift Software*;
2) нажать на кнопку *VR Preview* в окне редактора.

Чтобы разрабатывать под *Vive*:

1) установить *SteamVR* и запустить *Room Setup*;
2) нажать на кнопку *VR Preview* в окне редактора.

2.2.1. Настройка проекта *Rift* и *Vive*

Для создания простого проекта вам необходимы только *Player Pawn* и *Game Mode*.

Чтобы сделать это, создайте пустой *Blueprint*-проект, установив целевое устройство *Mobile/Tablet* и масштабируемость *Scale* 3*D or* 2*D*, и не подключайте стартовый контент (см. рис. 2.1). Эти настройки гарантируют, что некоторые дополнительные графические функции *UE*4 по умолчанию отключены (точные характеристики можно посмотреть в главе 10 «Оптимизация *VR*»).

Хорошая идея настроить базовую структуру вашего проекта с самого начала. Как только это будет сделано, добавим базовые *Blueprints*, которые нужны для *Rift* и *Vive*.

1. Создайте новый раздел в корневом разделе проекта. Назовите его *Blueprints*.
2. Создайте два новых *Blueprints*: один — это *Game Mode*, который мы назовем `VRGameMode`, а второй — *Pawn* под названием *VRPawn*. Ваш проект должен быть похож на рис. 2.17.

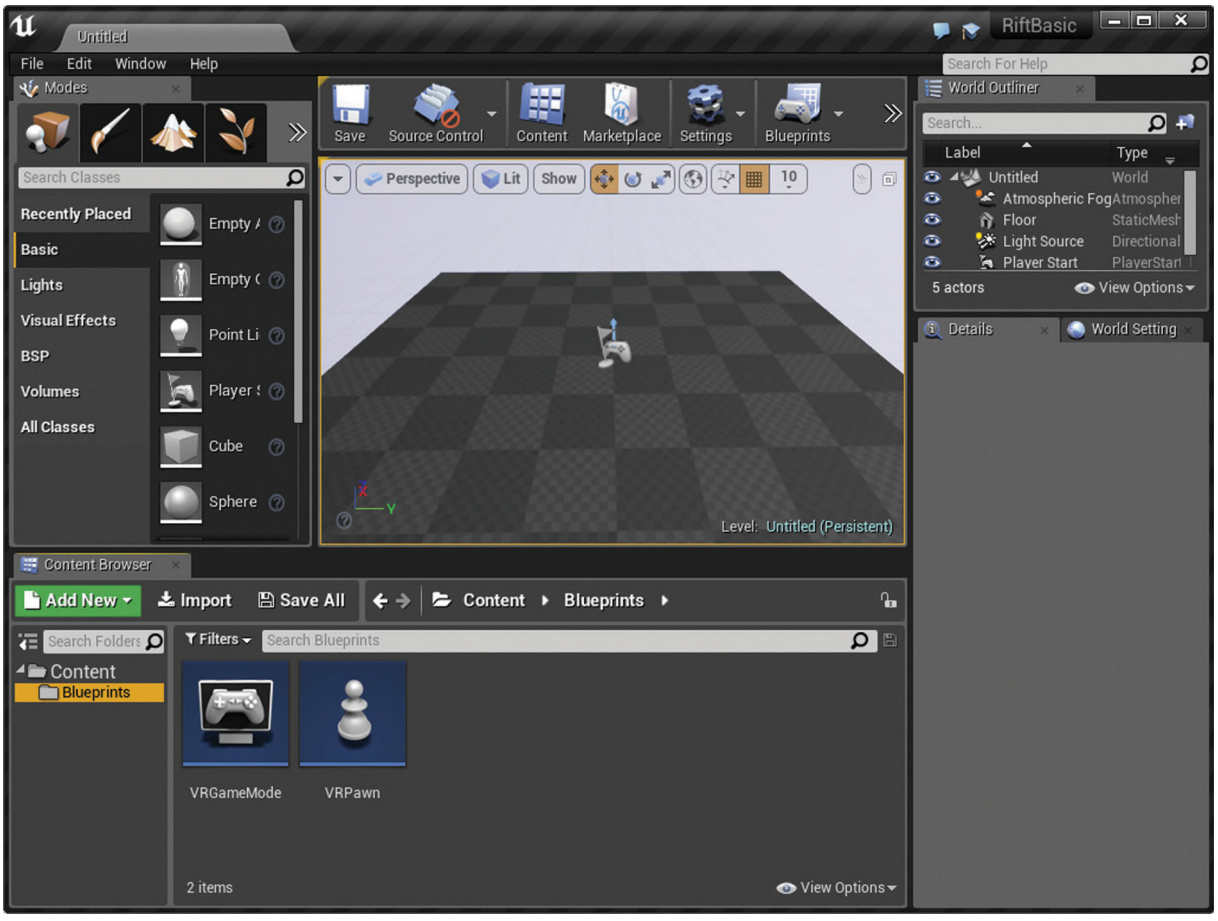

Рис. 2.17. Базовая настройка настольного *VR*-проекта

3. В *Project Settings* выберите секцию *Maps & Modes* и измените *Default GameMode* на *VRGameMode*, который вы создали (рис. 2.18).

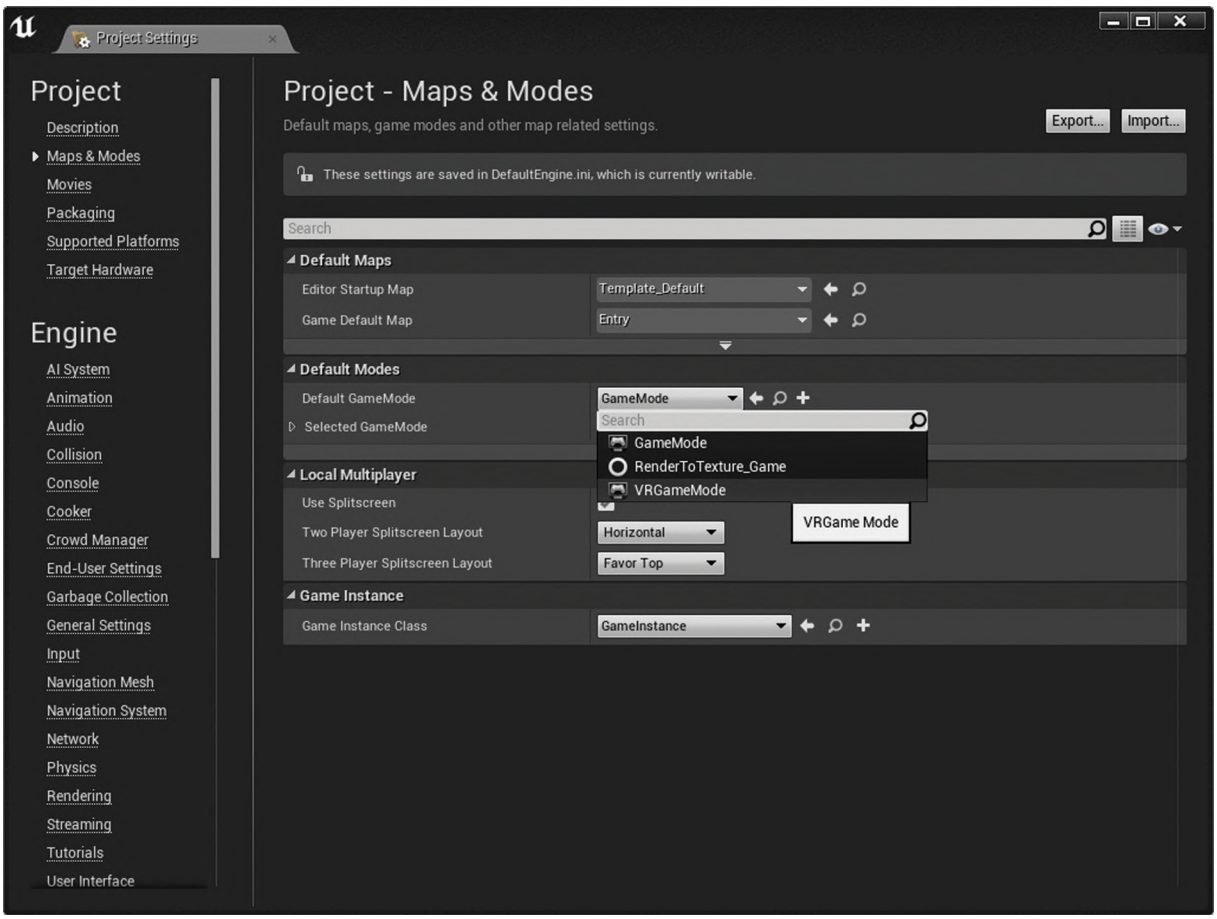

Рис. 2.18. Настольный *VR*: стандартный игровой режим [*Game Mode*]

4. Откройте ваш *VRGameMode* и установите *Default Pawn Class* на *VRPawn* (рис. 2.19).

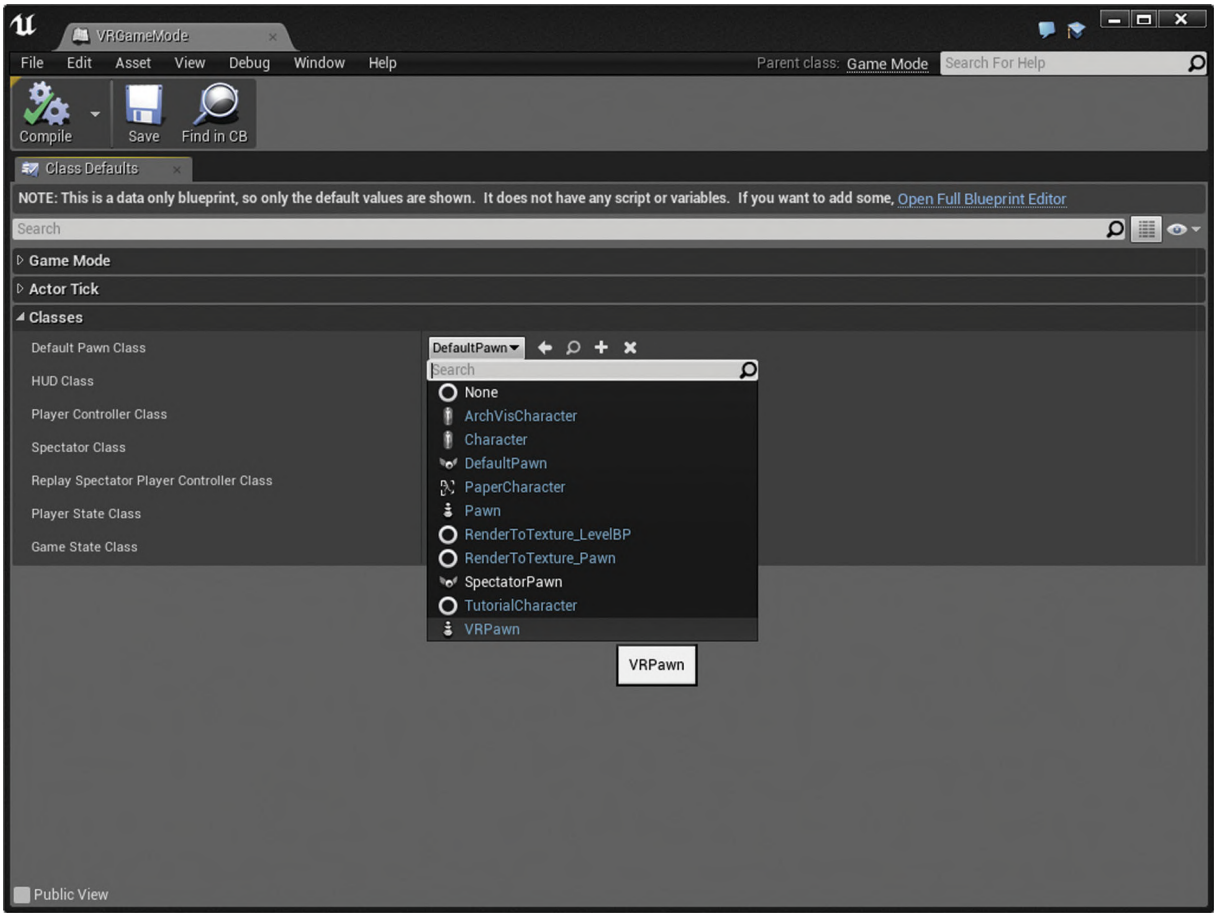

Рис. 2.19. Настольный *VR*: игровой режим

5. Откройте *VRPawn* и добавьте новые компоненты.

a. Первый компонент — это *Scene*, назовите его `CameraRoot`. Это будет основа для камеры, что позволит располагать камеру где вы хотите.

b. Создайте компонент *Camera* и сделайте его производным от `CameraRoot`, перенеся его на `CameraRoot` во вкладке *Components*.

c. Создайте два новых компонента *Motion Controller*; назовите их `MotionController_L` и `MotionController_2` (рис. 2.20.).

d. На втором компоненте *Motion Controller* измените переменную `Motion Source` на `Right`. Так мы укажем, что контроллер находится в правой руке.

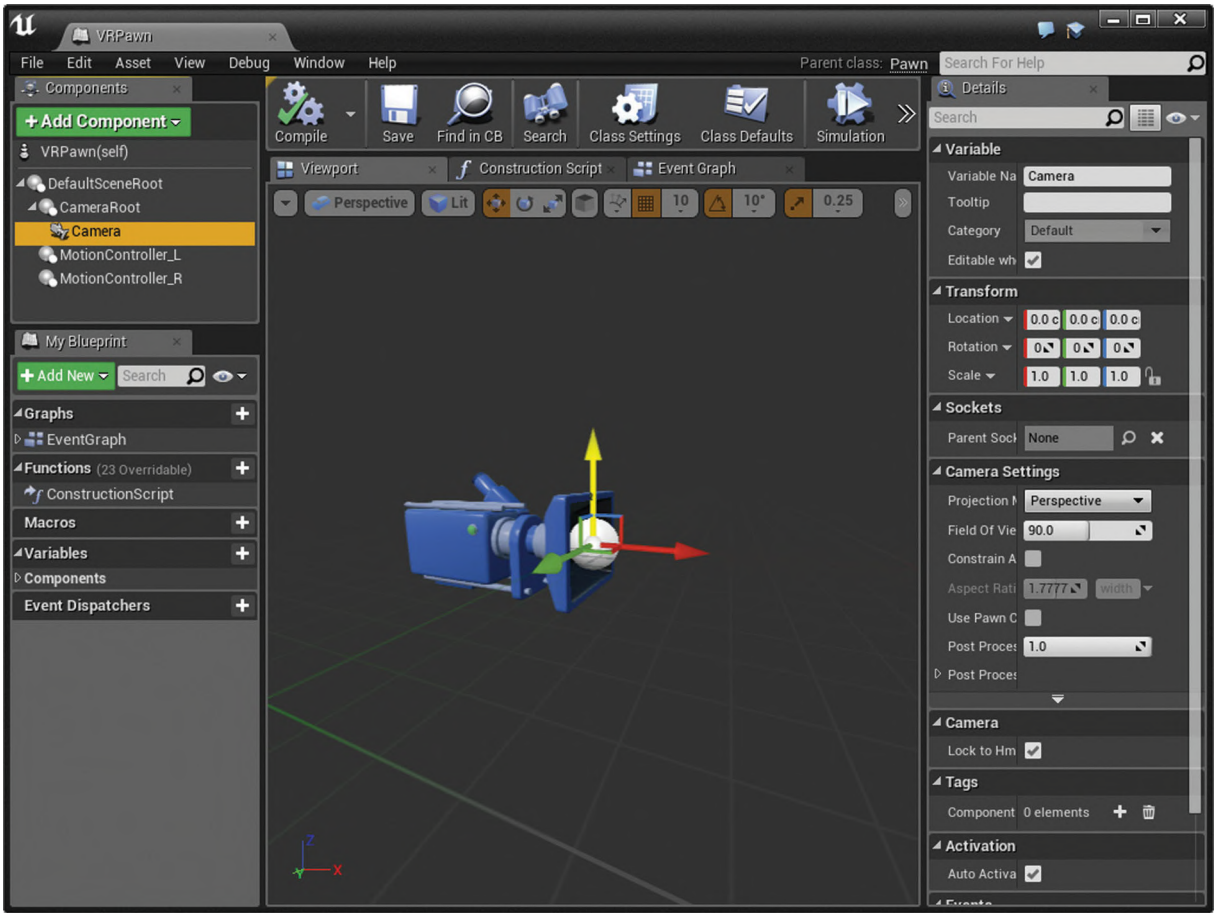

Рис. 2.20. Настольный *VR: Player Pawn*

6. Откройте *Event Graph* в *VRPawn*. Создайте новый узел `SetTrackingOrigin` и соедините его с `BeginPlay`, убедитесь, что в `Origin` установлено `Floor` *Level* (рис. 2.21). Эти настройки для использования *Rift* и *Vive* стоя; для других настроек отслеживания, см. следующий раздел «*Rift* и *Vive* — режимы отслеживания».

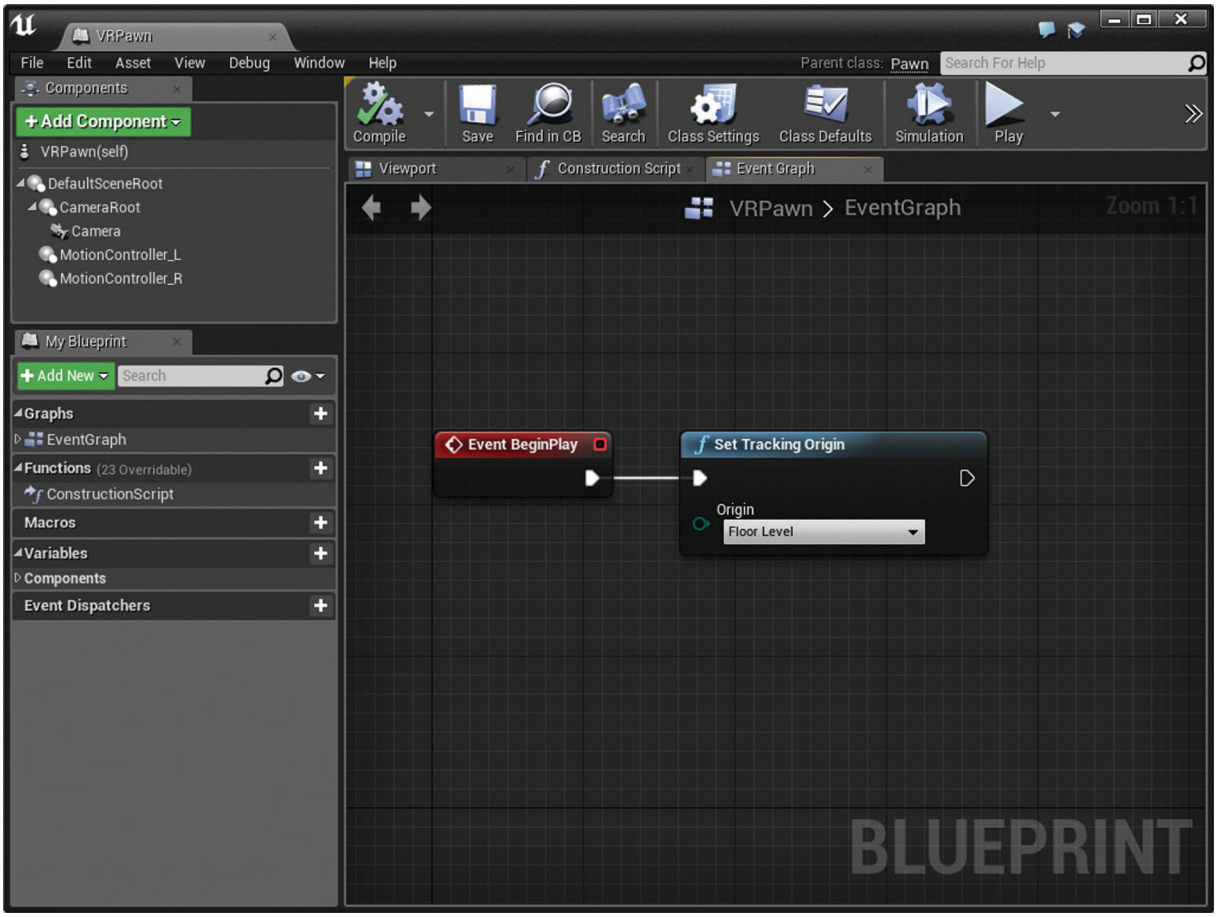

Рис. 2.21. Настольный *VR*: постоянное отслеживание источника

7. Настройте *Default Player Start Location* для использования стоя, так как позиция *VR*-шлема относительна физического пола в помещении. Захватите *Player Start* на уровне и сдвиньте по оси *Z* на 20 см; это высота стандартного пола в *UE*4 (рис. 2.22).

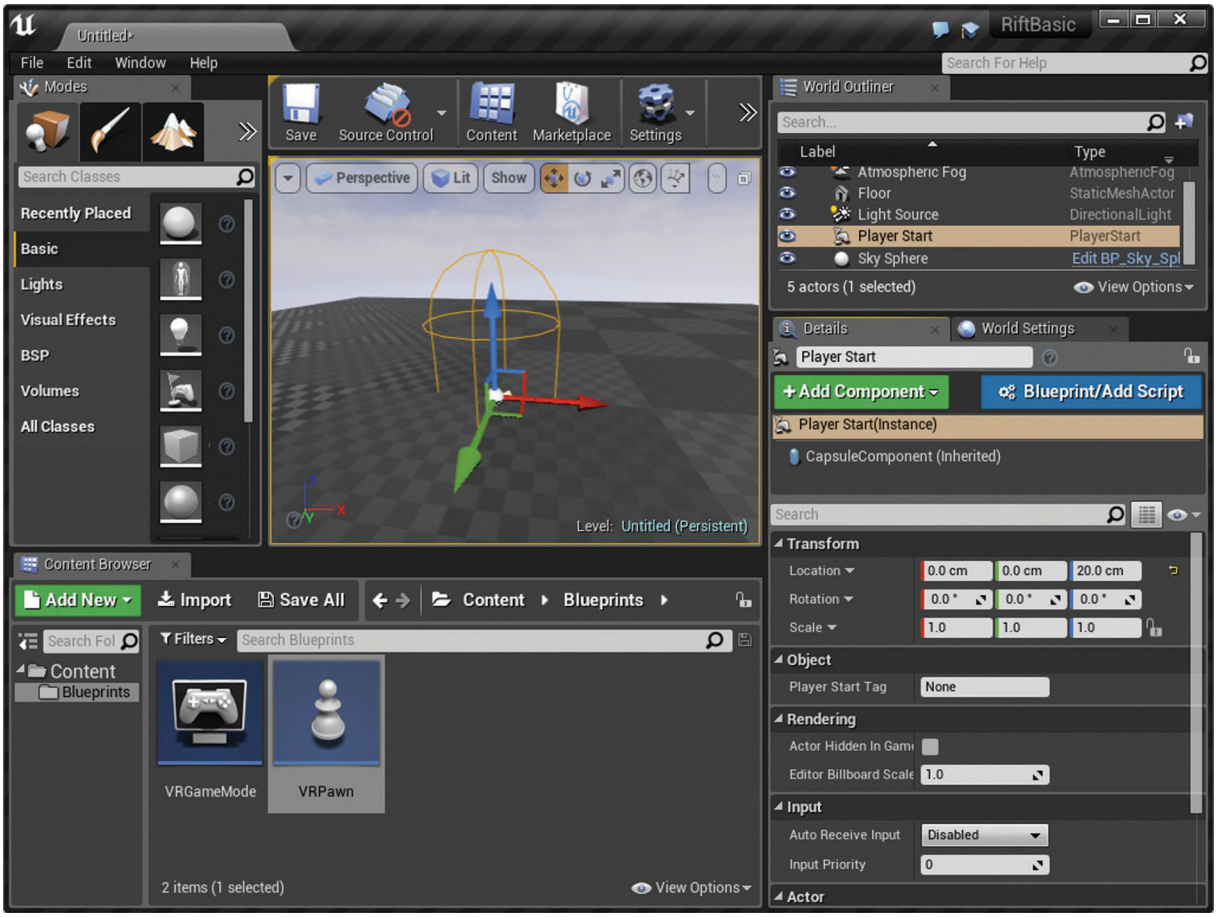

Рис. 2.22. Настольный *VR*: стартовая позиция

2.2.2. *Rift* и *Vive* — режимы отслеживания

Оба устройства позволяют пользоваться *VR* как стоя, так и сидя. Вы можете настроить, какую точку отсчета использовать при отслеживании движений шлема (рис. 2.21).

По умолчанию в *Rift* это сидячее положение. Из-за этого *VR*-камера помещается в одном метре по горизонтали от доступной камеры отслеживания (см. рис. 2.23). Однако если стоит отслеживание *Floor Level*, то позиция камеры будет определена при калибровке на этапе запуска *Oculus* (см. рис. 2.24).

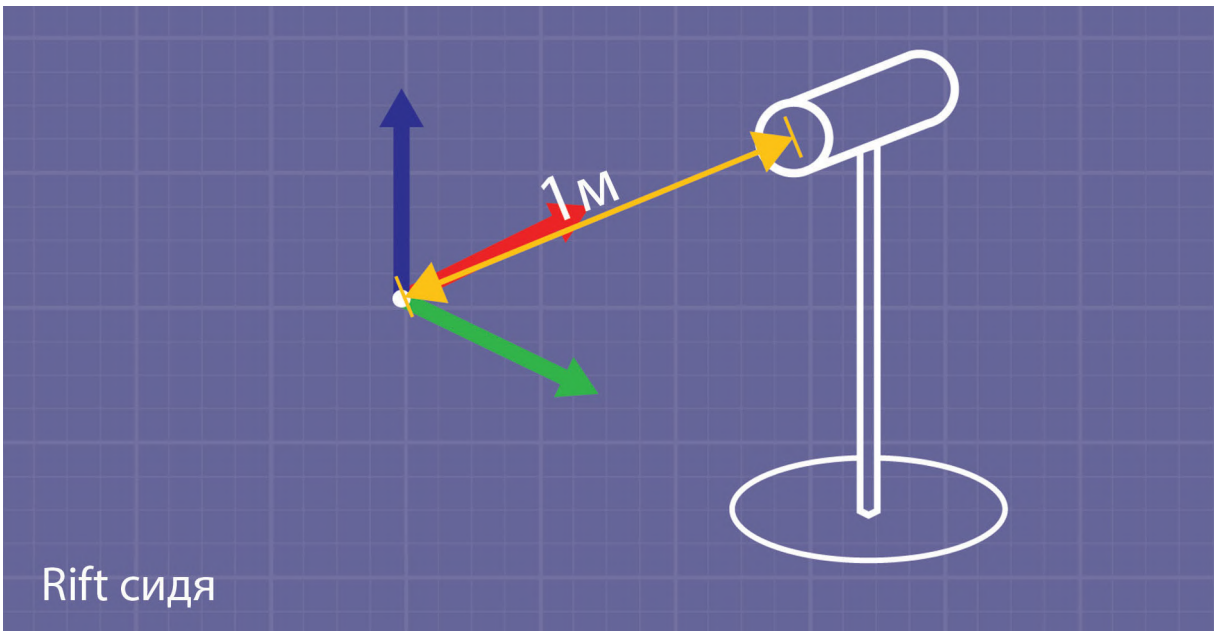

Рис. 2.23. *Rift* — точка отсчета на уровне глаз (сидя)

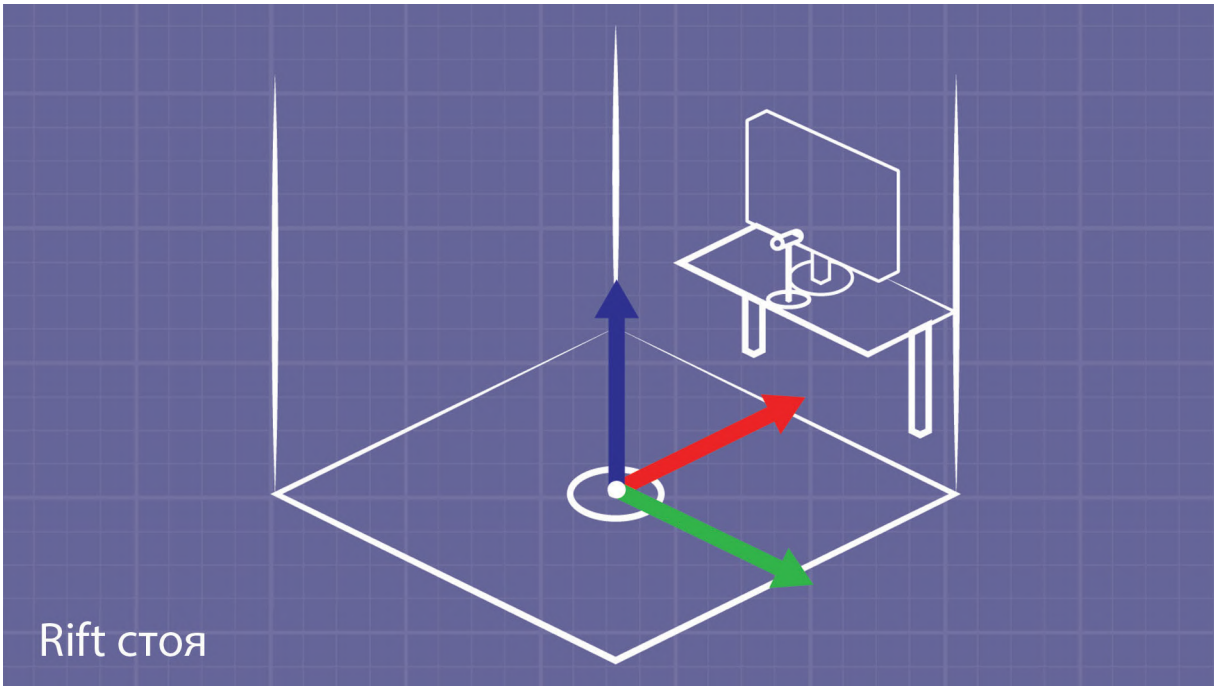

Рис. 2.24. *Rift* — точка отсчета на уровне пола (стоя)

У *Vive* по умолчанию установлено положение стоя. При этом камера помещается в центр игрового пространства игрока на уровне пола (рис. 2.25). Когда происходит переход в сидячий режим, *Vive* использует точку отсчёта в центре передней грани области перемещений головы, (обычно в плоскости дисплея) (рис. 2.26).

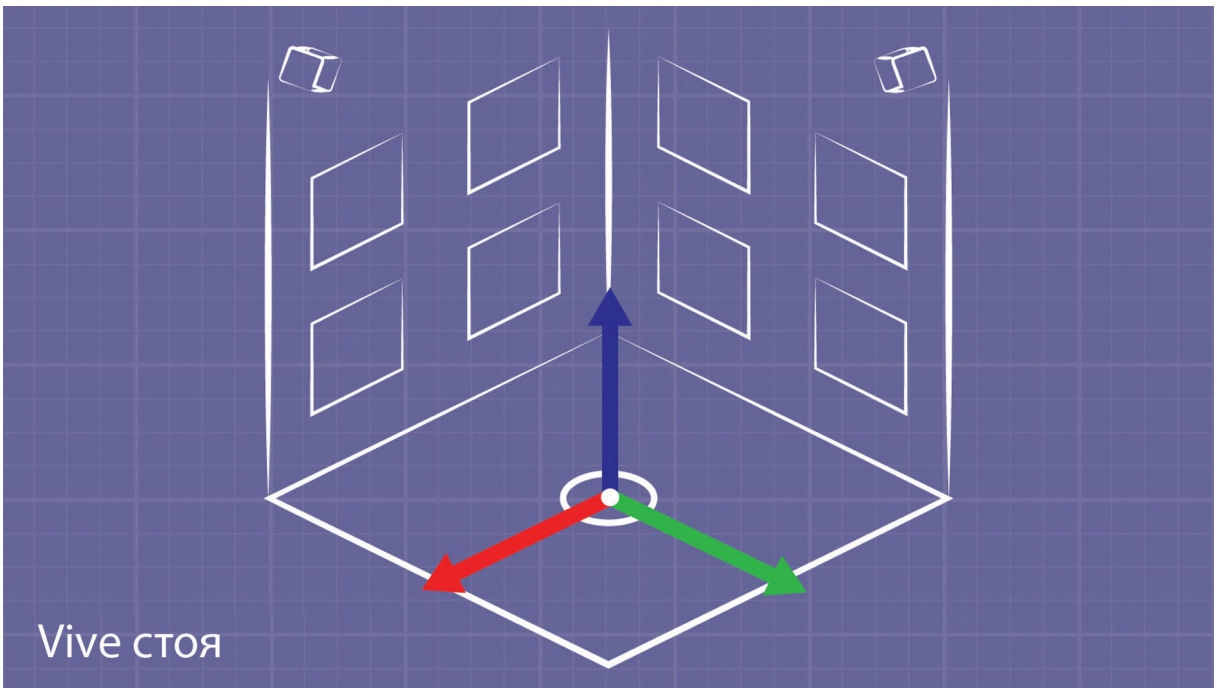

Рис. 2.25. *Vive* — точка отсчета на уровне пола (стоя)

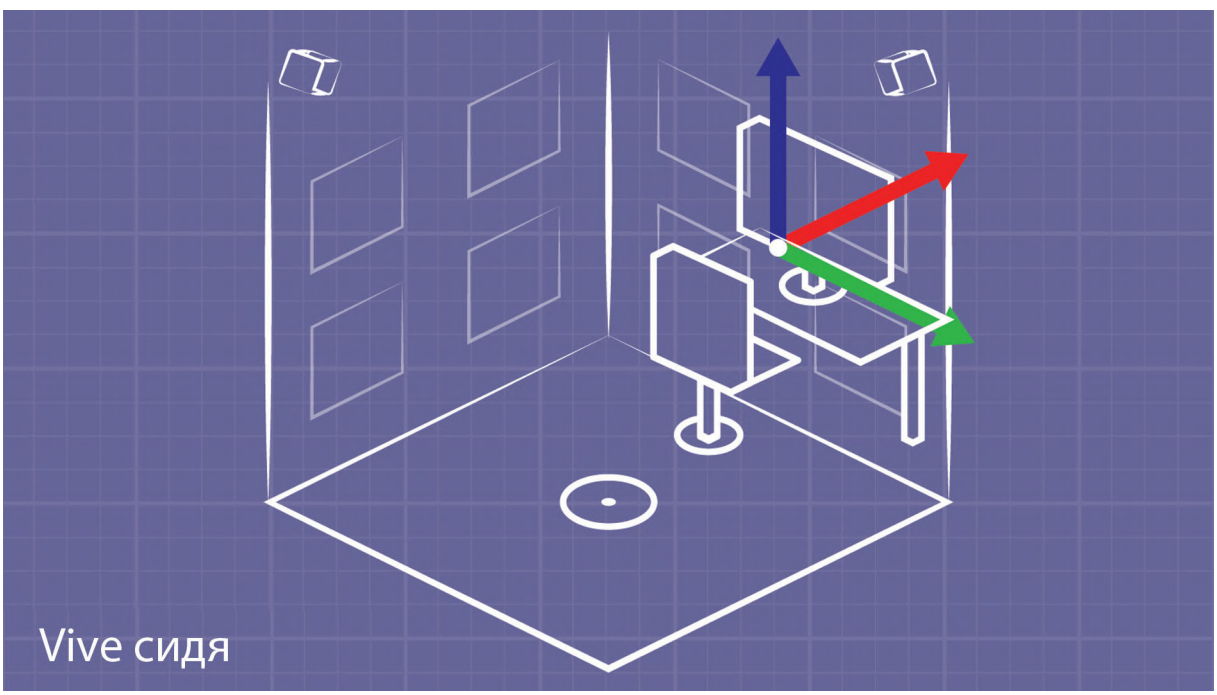

Рис. 2.26. *Vive* — точка отсчета на уровне глаз (сидя)

2.3. Заключение

Теперь вы имеете базу для создания своего проекта в *VR*. В главе были разобраны как мобильная *VR*, включая создание необходимого интерфейса с пользователем для меню загрузки, так и настольная *VR*, включая выбор режима отслеживания положения игрока. Теперь вы готовы приступить к изучению *VR*.

ГЛАВА 3

ИНСТРУМЕНТЫ

UE4 предоставляет широкий набор функций *Blueprint*, подходящих для создания *VR*-интерфейсов. В этой главе вы познакомитесь с некоторыми общими и специальными *VR*-библиотеками, входящими в *Unreal Engine*.

3.1. Библиотека обобщенных функций

При создании *VR*-сюжетов в *UE*4 у вас есть различные помощники и функции, доступные для получения разнообразных данных, начиная с положения головы пользователя и заканчивая температурой сенсоров *VR*-шлема.

Наряду с *API*-функциями, которые *Unreal Engine* предоставляет для взаимодействия с *SDK* конкретного *VR*-шлема, существует библиотека обобщенных функций *HMD*, которая дает вам доступ к возможностям, общим для нескольких *VR*-гарнитур (см. таблицу 3.1).

Таблица 3.1. Обобщенные функции *HMD*

Название функции	**Описание**	**Совместимость**
Get Positional Tracking Camera Parameters	Возвращает такие значения, как поле зрения камеры слежения за *VR*-шлемом. Возвращает положение и направление камеры относительно шлема	Поддерживается только *Rift*; *Gear VR* не отслеживает перемещения шлема, а *Vive* не использует камеру (устарело с версии 4.13)
Get Tracking Sensor Parameters	Возвращает данные датчика положения, если он подключен	Добавлено в 4.13
Get Orientation and Position	Возвращает относительную ориентацию и положение *VR*-шлема	Работает на всех *VR*-шлемах
Is Head Mounted Display Enabled	Сообщает о подключении *VR*-шлема и выводе изображения в него	Работает на всех *VR*-шлемах
Get Screen Percentage	Возвращает текущее относительное разрешение рендеринга	Работает только на устройствах *Oculus*; *Vive* отслеживает масштабирование через консольную команду `r.screenPercantage`, а *Rift* через команду *HMD SP*
Get Tracking Origin	Возвращает текущую привязку отслеживания для *VR*-шлема (уровень пола или глаз)	По умолчанию *Rift* настроен на уровень глаз, а *Vive* — на уровень пола. Добавлено в 4.11
Get VR Focus State	Возвращает, имеет ли движок фокус в *VR-compositor*	Не реализовано для *Vive*. Добавлено в 4.11
Get World to Meters Scale	Возвращает текущий масштабный коэффициент, позволяющий пропорционально масштабировать вид *VR*	Работает на всех *VR*-шлемах
Is in Low Persistence Mode	Возвращает, находится ли дисплей *VR*-шлема в режиме низкой персистентности	Всегда возвращает `True` для *Rift* и *Vive*
Set Tracking Origin	Устанавливает источник отслеживания на уровне глаз или пола	Работает на *Rift* и *Vive*. Добавлено в 4.11
Set Clipping Plane	Установить специальные *VR*-плоскости отсечения	Не реализовано для *Vive*
Set World to Meters Scale	Устанавливает текущий масштабный коэффициент, позволяющий пропорционально масштабировать вид *VR*	Работает на всех *VR*-шлемах
Reset Orientation and Position	Сброс ориентации и/или положения начала координат *HMD* к текущему *VR*-шлему	Работает на всех *VR*-шлемах

Название функции	Описание	Совместимость
Enable HMD	Включает *VR*-шлем	Работает только в автономной сборке игры, в противном случае движок имеет приоритет над *VR-compositor*
Get HMD Device Name	Возвращает имя текущего *VR*-шлема (например, *Gear VR* для *Gear VR*)	Работает на всех *VR*-шлемах. Добавлено в 4.13

3.2. Библиотека функций *Oculus*

Наряду с обобщенной библиотекой функций *HMD*, *UE*4 предоставляет доступ к *SDK*, которые позволяют получить доступ к функциям конкретного *VR*-шлема.

Библиотека функций *Oculus* дает вам доступ к более низкоуровневой информации о *VR*-шлеме и к информации о профиле пользователя (таблица 3.2).

> **Заметка**
>
> Функции из таблицы 3.2 специально ориентированы и работают только с *Gear VR* и *Rift*.

Таблица 3.2. Функции *Oculus*

Название функции	Описание	Совместимость
Get Base Rotation and Base Offset in Meters	Возвращает смешение текущего *VR*-шлема к исходному *HMD* игры	
Get Player Camera Manager Follow HMD	Возвращает признак включения функции *Follow HMD* в диспетчере камер	Удалена в версии 4.11
Get Pose	Возвращает позицию и поворот *VR*-шлема, а также позицию шеи для виртуальной модели головы	
Get Raw Sensor Data	Возвращает необработанные данные с *IMU* (*inertial measurement unit*) *VR*-шлема; может быть использован для определения небольших изменений в движениях	
Get User Profile	Возвращает информацию о профиле текущего игрока: имя, пол, рост и т. д.	
Is Auto Loading Splash Screen	Возвращает, должен ли пользовательский экран заставки автоматически отображаться между загрузками уровня (по умолчанию *True*)	
Is Player Controller Follow HMD Enabled	Возвращает информацию о состоянии контроллера *Follow HMD*	Удалена в версии 4.11
Add Loading Splash Screen	Позволяет указать пользовательскую текстуру и преобразование для пользовательского экрана заставки	

Название функции	Описание	Совместимость
Show Loading Icon	Показывает единственную 2*D*-текстуру, аналогичную заставке без пользовательского преобразования	Добавлена в версии 4.13
Hide Loading Icon	Скрывает иконку загрузки	Добавлена в версии 4.13
Is Loading Icon Enabled	Возвращает, отображается ли значок загрузки	Добавлена в версии 4.13
Set Position Scale 3*D*	Масштабирует позиционное отслеживание *VR*-шлема	Не реализована
Clear Loading Splash Screens	Очищает все заставки, отображаемые в данный момент	
Enable Auto Loading Splash Screen	Устанавливает, должна ли пользовательская асинхронная заставка автоматически отображаться между загрузками уровня (по умолчанию *True*)	
Enable Player Camera Manager Follow HMD	Включает *Follow HMD* на менеджере камеры игрока	Удалена в версии 4.11
Show Loading Splash Screen	Показывает пользовательский экран заставки	
Enable Player Controller Follow HMD	Включает *Follow HMD* на контроллере игрока	Удалена в версии 4.11
Hide Loading Splash Screen	Прячет пользовательский экран заставки	
Set Base Rotation and Base Offset in Meters	Устанавливает заданное смещения для *VR*-шлема	

3.3. Библиотека функций *Steam VR*

У *Steam VR* меньше функций, чем у *Oculus*, тем не менее они позволяют получить доступ к контроллерам *Vive* и базовым станциям.

Подобно библиотеке *Oculus*, функции из таблицы 3.3 работают только для устройств *Steam VR*.

Таблица 3.3. Функции *Steam VR*

Название функции	Описание	Совместимость
Get Hand Position and Orientation	Возвращает положение и ориентацию левого или правого контроллера *Steam VR* относительно источника *HMD*	Не учитывает последние обновления для компонентов контроллера
Get Tracked Device Position and Orientation	Учитывая идентификатор устройства, возвращает ориентацию и положение устройства относительно источника *HMD*	
Get valid Tracked Device IDs	Возвращает массив идентификаторов для данного типа устройства	

3.4. Заключение

Вы получили представление обо всех типах функций, доступных при создании *VR*-сюжетов как средствами обобщенной библиотеки *HMD*, так и из специализированных библиотек для различных устройств. Теперь у вас есть все, что нужно для изучения описанных в книге рецептов и накопления собственного опыта.

ЧАСТЬ II

РЕЦЕПТЫ

ГЛАВА 4

ВЗАИМОДЕЙСТВИЕ НА ОСНОВЕ ТРАССИРОВКИ

VR позволяет разработчикам использовать ранее недоступные естественные способы ввода данных при взаимодействии с пользователем. Важнейший из них — движение головы игрока, поэтому в этой главе обсуждается использование этого способа ввода для взаимодействия с объектами в ваших виртуальных мирах.

Понимание трассировки важно при разработке практически любых *VR*-игр. В этой главе вы создадите модульную систему взаимодействия с пользователем, которую можно легко расширить.

4.1. Понимание взаимодействия трассировки

4.1.1. Принципы взаимодействия на основе трассировки

Взаимодействие на основе трассировки лучей в *VR* является распространенным методом, позволяющим игроку взаимодействовать с виртуальным миром путем определения объекта, на который игрок смотрит (или указывает контроллером).

Трасса [*trace*] (или бросок [*cast*]) луча на самом деле определяет более широкое понятие в контексте *UE4* и еще шире трактуется в компьютерной графике в целом. Однако при упоминании трассировки лучей в этой книге мы имеем в виду набор функций *Unreal Engine*, которые позволяют установить начальную и конечную точки (а также некоторые настройки) и выявить объекты, находящиеся «между» этими двумя точками.

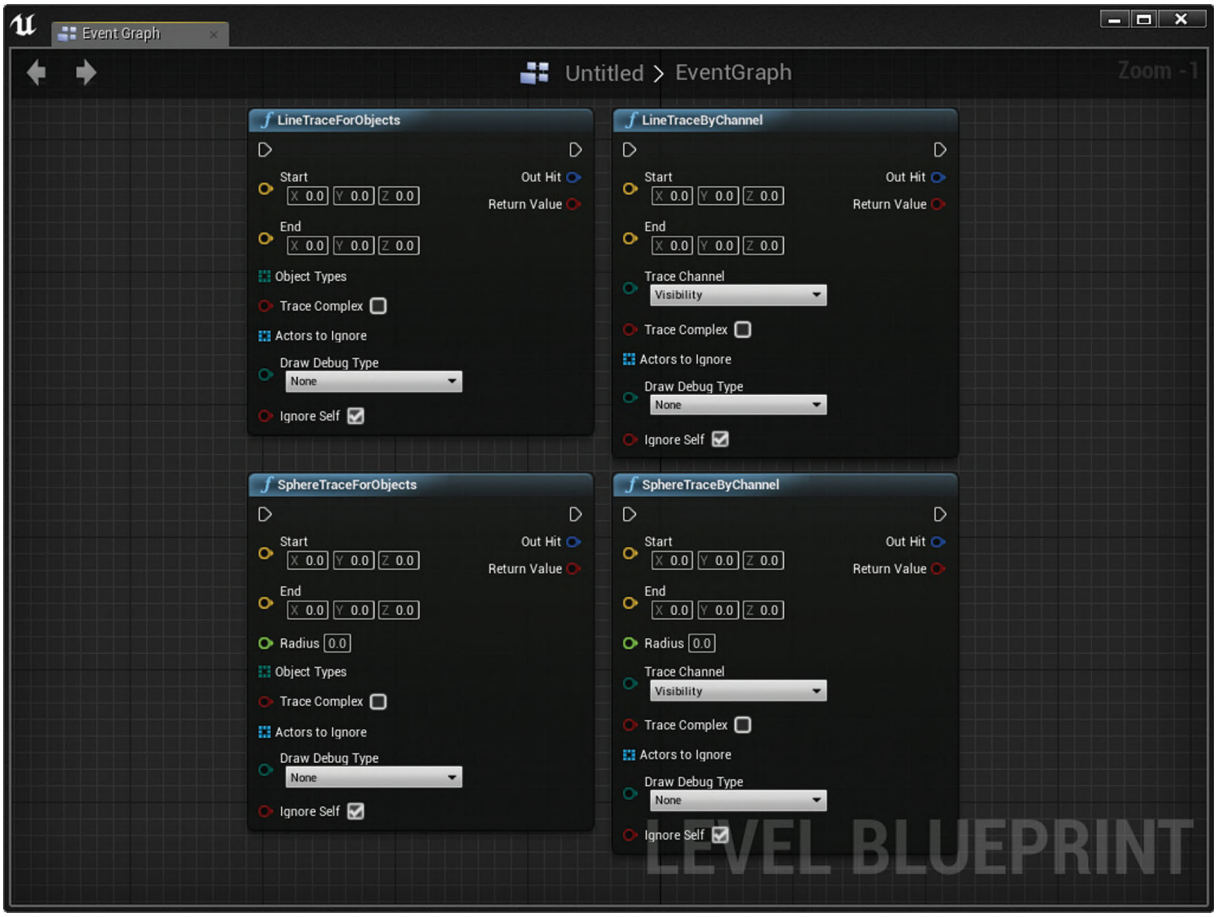

Рис. 4.1. Пример функций трейсинга, доступных в *UE4*

Unreal Engine дает вам возможность использовать различные виды трасс, отражающих специфику разнообразных задач. Простейшая **трасса линии** [*line trace*] (рис. 4.1) выявляет любые объекты, пересекаемые отрезком от начальной до конечной точки. Более сложные виды **трасс фигур** [*shape trace*], например

трассы куба, сферы или капсулы, выявляют объекты, которые соответствующая фигура пересекает во время перемещения от начальной до конечной точки (смысл трасс фигур иллюстрируется рис. 4.2).

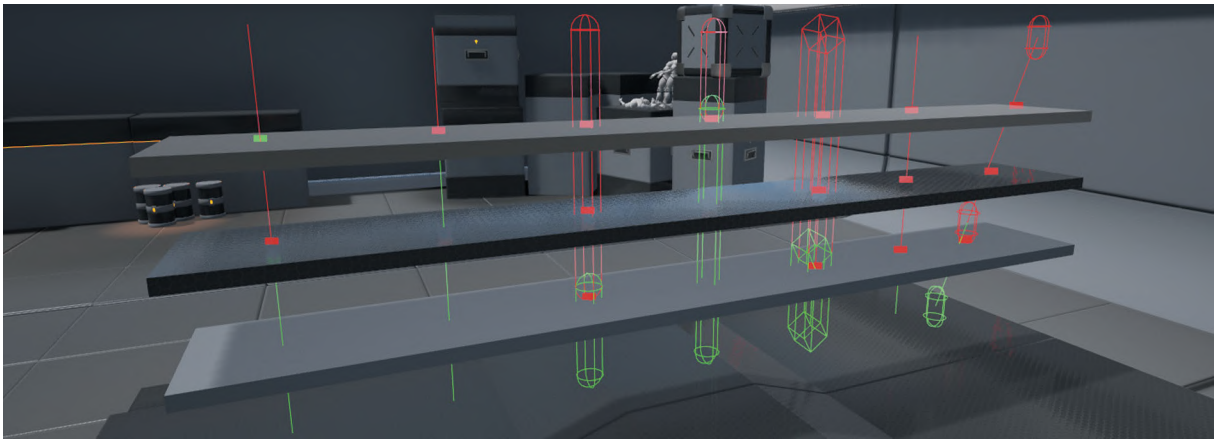

Рис. 4.2. Трассы линии, сферы, куба и капсулы

Помимо трассируемой фигуры (линия, куб и т. д.), у трассы есть другой критерий классификации: можно трассировать фигуру относительно **канала** [*channel*] или **объектов** [*object type*] (рис. 4.1). Различия между ними неочевидны начинающему, поэтому при выборе между каналом и типами объектов могут возникнуть трудности. Пытаясь различить эти варианты трасс, мы обнаруживаем, что оба варианта предлагают идентичные параметры за одним исключением: у трасс относительно канала присутствует параметр «Канал» [*Trace Channel*], который у трасс относительно объектов заменяется на параметр «Типы объектов» [*Object Types*]. Те, кто ранее работал с системой столкновений *Unreal Engine*, знают, что *UE*4 предлагает два типа ответов на столкновения: `Object` и `Trace`. Они идеально соотносятся с двумя типами трасс. Это означает, что с помощью подходящей функции трассировки *UE*4 позволяет искать (трассировать) определенные типы объектов в вашем мире, игнорируя те, которые вас не волнуют. Еще одно отличие между этими типами трасс состоит в том, что в то время, как трассировки объекта позволяют определить несколько типов объектов (что может быть полезно, когда вы хотите выполнить трассировку относительно нескольких типов одновременно, например игроков и транспортных средств), трассировка относительно канала разрешает выбрать только ***один*** канал. По умолчанию это `Visibility` или `Camera`; однако, как и в случае с ответами столкновений для настраиваемых объектов, в настройках проекта можно добавить дополнительные каналы.

Как нам правильно настроить одну из этих трасс для взаимодействия с пользователем в виртуальном мире?

В игре вы имеете доступ к двум основным свойствам, которые позволяют рассчитать начальную и конечную точки фокуса внимания, то есть области пространства, на которую смотрит игрок. Оба свойства представлены в виде векторов (три числа, которые используются для представления точки или направления в определенном координатном пространстве). Первый вектор — расположение головы игрока в мировом пространстве, что означает, что компоненты *X*, *Y* и *Z* этого вектора представляют собой расстояние «вперед», «вправо» и «вверх» относительно исходной точки игрового мира. Второй вектор — направление головы игрока вперед, которое вы получаете от игровой камеры, представляющей глаза игрока. Рис. 4.3 показывает эти два вектора.

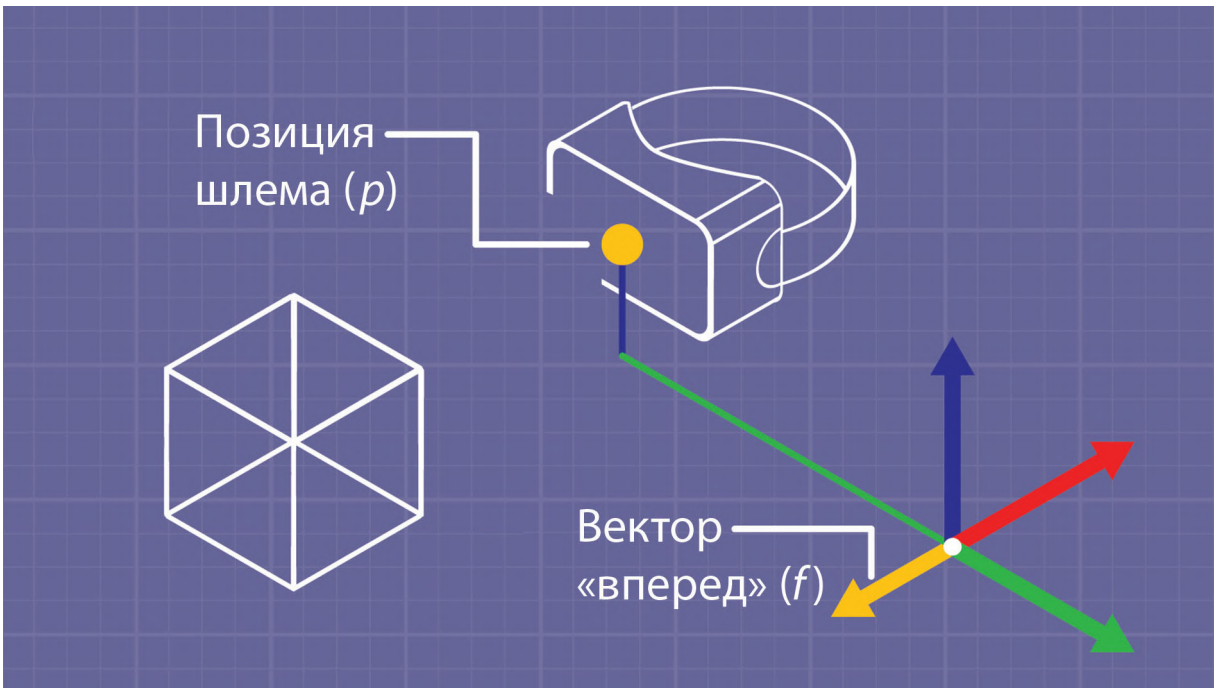

Рис. 4.3. Векторы положения *HMD* (*p*) и прямого взгляда (*f*), представленные в *UE*4

> **Заметка**
>
> В этом примере в качестве места для трассировки используется голова игрока. Тем не менее обратите внимание, что это также работает с любым другим объектом в игре, из которого вы хотите начать трассировать (например, контроллер движения).

Чтобы использовать эти переменные для трассировки, можно сначала передать мировые координаты *VR*-шлема как начальную точку трассы. Затем, чтобы получить конечную точку трассы, понадобится немного простой математики. Прежде чем перейти к вычислениям, нам понадобятся некоторые сведения. Вектор направления в *UE*4 всегда имеет длину, равную единице (1), из этого факта следует полезное свойство: чтобы масштабировать его на любую другую длину, надо умножать вектор направления на число, представляющее нужную длину. (В нашем случае эта длина определяет, насколько далекие объекты вы хотите трассировать. Напомним, что 1 *UU* [*Unreal Unit*] эквивалентна 1 см, поэтому длина 100 *UU* = 1 м.) Как только направление взгляда будет масштабировано до необходимой длины, вам нужно прибавить его к положению головы игрока для позиционирования относительно головы игрока (рис. 4.4), а не относительно мировой точки отсчета, в котором вектор направления взгляда был изначально.

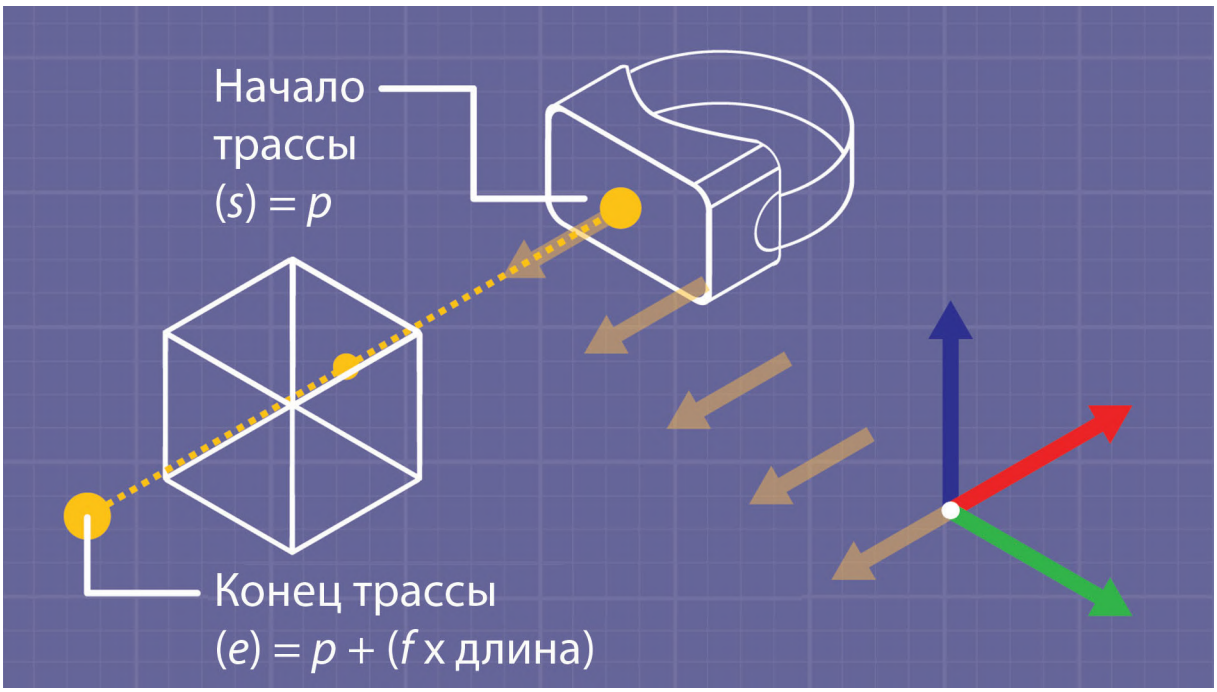

Рис. 4.4. Вычисление начальной и конечной точек трассировки (больших оранжевых кругов) путем масштабирования вектора направления взгляда *HMD* (*f*) на требуемую длину, а затем прибавления его к мировым координатам *HMD* (*p*), чтобы убедиться, что конечная точка находится в мировых координатах

Чтобы узнать векторы направления и положения мировой точки отсчета в игре, используйте *Blueprint*-узлы *GetWorldLocation* и *GetForwardVector* (рис. 4.5) из нужного компонента, в данном случае компонента *Camera*.

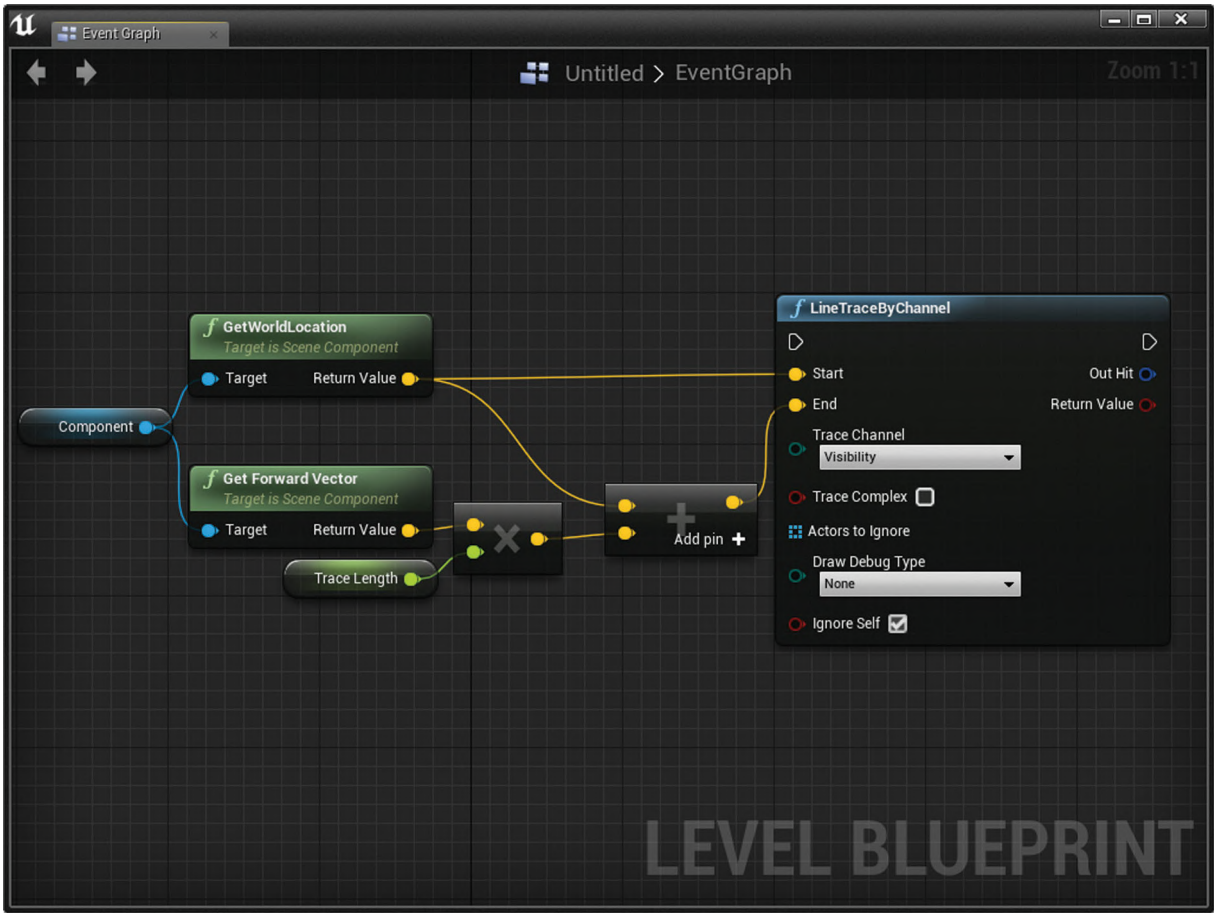

Рис. 4.5. Простейшая функция трассировки для нахождения объектов перед компонентом

4.1.2. Принципы взаимодействия с пользователем

Интерфейс с пользователем (ИП) является одним из базовых объектов, которые участвуют в создании системы взаимодействия. Это позволяет игровому персонажу воздействовать на другие объекты.

Если у вас есть опыт программирования и вы знакомы с идеей интерфейса, пропустите этот раздел и перейдите к разделу «Настройка взаимодействия трассировки».

В программировании интерфейс позволяет определить набор функций, которые любой объект может обещать реализовать и выполнить при вызове. Это полезно для таких вещей, как системы взаимодействия, где вы сначала определяете набор взаимодействий, которые могут сделать игроки. Тогда любой объект в мире может сказать игрокам, что они способны справиться с этими взаимодействиями. Обратите внимание, что игрокам не нужно ничего знать о том, как объект реализации будет обрабатывать это взаимодействие.

Например, у вас может быть шар, который начинает катиться, когда игрок смотрит на него, и у вас также может быть куб, который вращается, когда на него смотрят. Использование интерфейса взаимодействия «Взгляд» позволяет вам вызвать функциональность «Взгляд» на обоих из этих объектов без ведома игрока (или заботы) о том, что шарику нужно катиться, и кубу нужно закрутиться.

Это разделение триггера и действия создает систему взаимодействия, которая легко расширяется и позволяет добавить больше интерактивных объектов, имея только один объект (обычно игрок), который решает, что считается взглядом, а что нет.

Читатели, знакомые с объектно-ориентированным программированием, могут знать, что другим способом предоставления функциональности объектам является наследование функциональности от родительского объекта. Это действительно работает как система взаимодействия, потому что вы можете гарантировать, что, если объект, на который вы смотрите, имеет родителя, который имеет функциональность взаимодействия, вы можете вызвать правильные функции при необходимости.

Однако у этого есть один существенный недостаток: по крайней мере, в *Blueprints* объект может иметь только одного родителя. Это означает, что если вы хотите иметь и куб, и свинью, с которыми можно взаимодействовать, они должны будут наследовать от одного и того же объекта, даже если в вашей игре для свиньи разумнее наследовать от класса животных, а куб наследовать от класса геометрических фигур. (Это различие показано на рис. 4.6.)

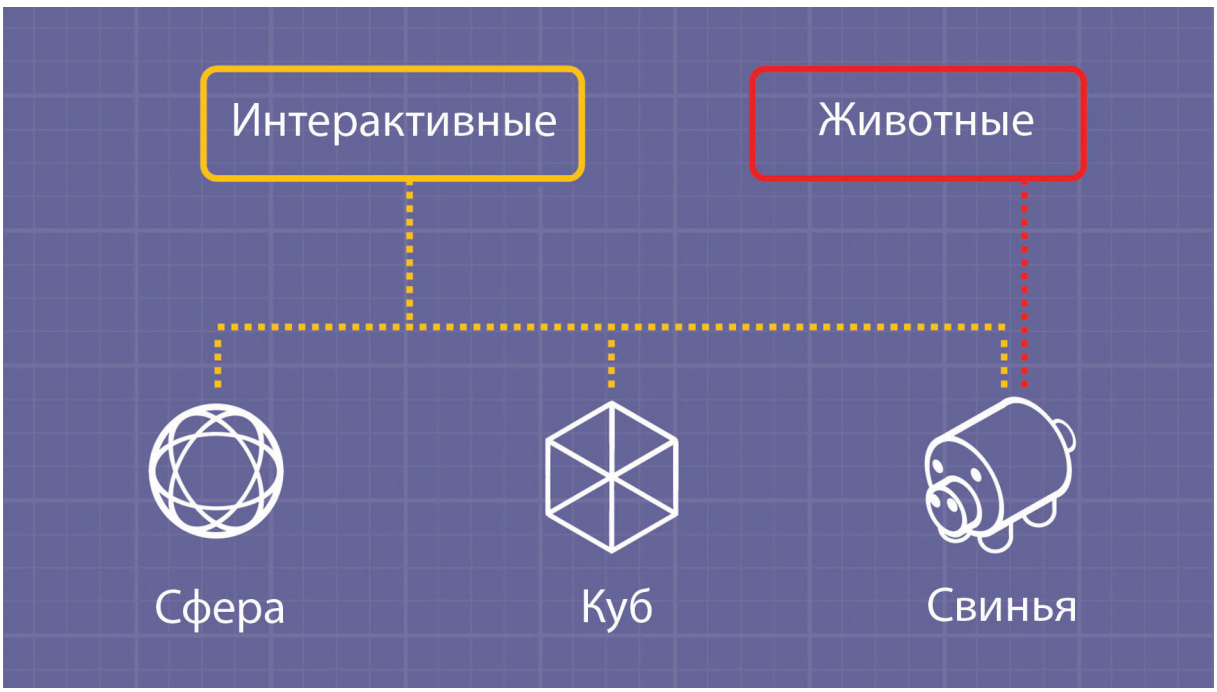

Рис. 4.6. Интерфейсы против классов. Использование интерфейса (желтый) позволяет кубу, сфере и свинье быть интерактивными, но у свиньи все еще может быть родительский класс (красный) животного; это было бы невозможно в *Blueprints*, если бы интерфейс был классом, потому что разрешено только одиночное наследование

4.1.3. Настройка взаимодействия трассировки

Теперь реализуем взаимодействие на основе трассировки так, чтобы вы смогли потом использовать эту реализацию в своем приложении.

4.1.4. Начальная настройка проекта

Для начала вам необходимо создать объект для настройки системы взаимодействия.

1. Создайте папку в корневом разделе проекта и назовите ее *Blueprints*.
2. В ней создайте еще две папки, *Components* и *Interfaces*.
3. В папке *Blueprints* создайте *Pawn* (выбрав *Add New* ⇒ *Pawn Class*).
4. Назовите его `TraceInteractionPawn`.
5. В папке *Components* создайте *Blueprint Scene Component* под названием `TraceInteractionComponent`. Он будет содержать логику работы трассы. Было бы проще с точки зрения программирования не отделять логику от *Pawn*, но, если она выделена в отдельный компонент, вы можете добавить любому объекту возможность взаимодействия на основе трассировки, что сделает ваш код гибче и упростит его добавление в другие проекты.
6. В папке *Interfaces* создайте новый *Blueprint interface* и назовите его `TraceInteractionInterface`. Для этого выберите *Add New* и в секции *Create Advanced Asset* раскройте меню *Blueprints* выберите *Blueprint Interface*.

Ваш проект должен выглядеть как на рис. 4.7.

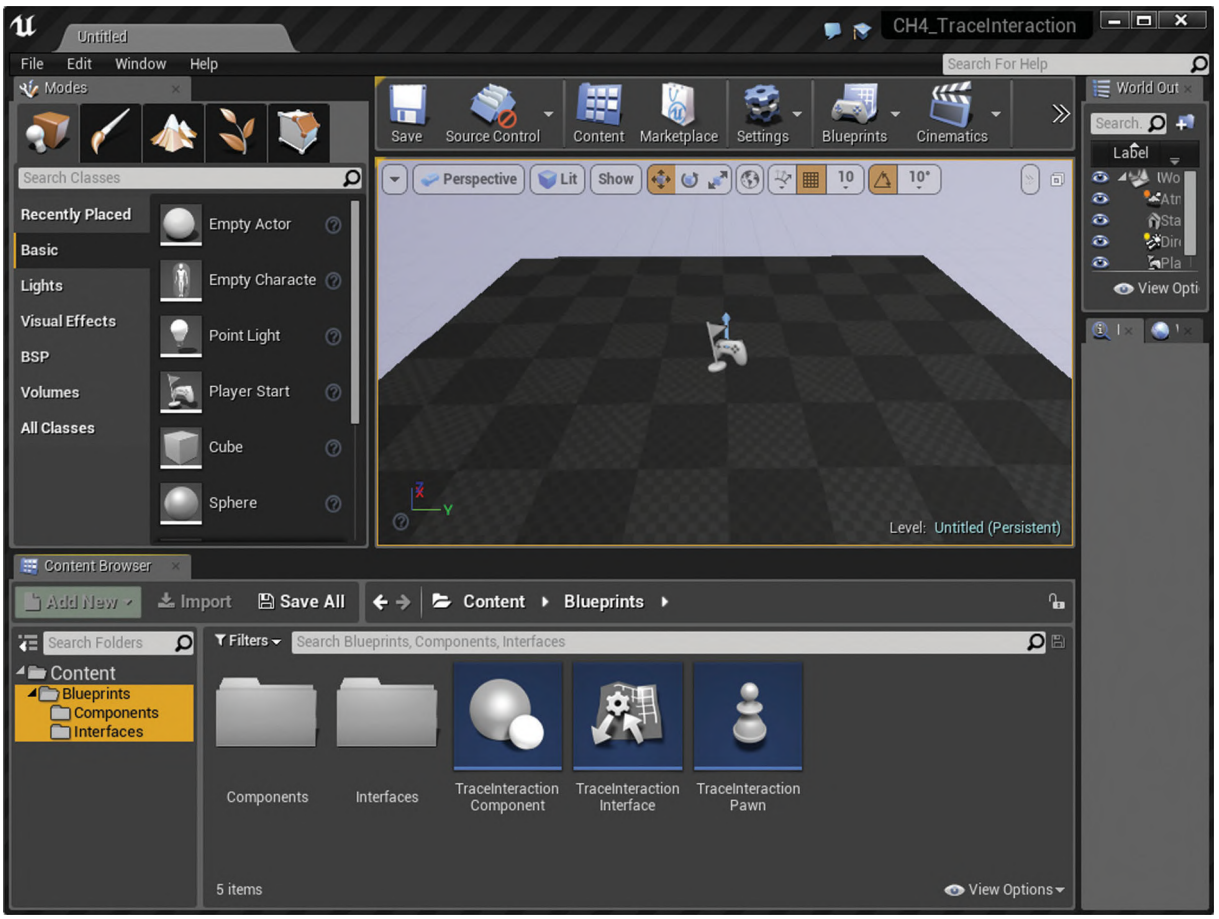

Рис. 4.7. Настройка проекта для отслеживания взаимодействия

4.1.5. Настройка интерфейса взаимодействия

Чтобы разрешить *Interaction Component* вызывать функции на интерактивных объектах, вам необходимо определить их в интерфейсе, который вы создали.

Следующие функции позволят создать основу системы взаимодействия, которая будет обрабатывать большинство случаев обнаружения кликов на объектах и компонентах; однако, если вам требуется специфическая функциональность, добавьте новые функции в этот *Blueprint*.

1. Откройте *Blueprint* `TraceInteractionInterface`.
2. Вам нужны семь функций для этого интерфейса. Назовите первые две `TraceHitObject` и `TraceLeaveObject`. Вы будете их использовать, чтобы сказать объекту, что отслеживание произошло или потеряно.
3. На вход (*Inputs*) в эти функции создайте ввод типа `Hit Result` (назовите его `Hit`). Передача столкновения позволяет вам узнать много полезного из следа, попавшего на объект.
4. Создайте еще две функции, `TraceHitComponent` и `TraceLevelComponent`. Они будут действовать как обнаружение попадания по объектам на уровне компонентов *Component level*, а не на объекте в целом.
5. На вход этим функциям подайте `Hit Result`, добавив этот раз еще переменную `Component` типа `Primitive component`. Это облегчит доступ к `Hit Component` функции попадания, которая не будет доступна в `Hit Result`.
6. Создайте функцию `TraceMove`. Эта функция вызывается каждый раз, когда трасса проходит через объект попадания.
7. Так как эта функция вызывается на каждой обработке кадра (*frame*), хотелось бы оптимизировать ее немного больше, чем другие функции интерфейса. Поэтому добавьте входную переменную `Hit Result`, как раньше, но в этот раз передадим ее по ссылке (*Pass-by-Reference* в развернутом описании входной переменной). Это позволит не копировать каждый раз переменную объекта столкновения, а обращаться к объекту в памяти по ссылке.
8. Использование передачи значений по ссылке в *Blueprint* означает, что вам потребуется обозначить функцию как константную*, то есть не способную напрямую изменять состояние объекта *Blueprint*. Вы можете сделать это, развернув вкладку *Graph* в *Details* и проверив параметр *Const* (рис. 4.8).

* Константная функция отсылает нас к пониманию концепций языка *C++*. Кратко ознакомиться с константными функциями *C++* можно в разделе *const-, volatile-, and ref-qualified member function* онлайн-справочника *Cppreference* (*https://en.cppreference.com/w/cpp/language/member_functions#const-.2C_volatile-.2C_and_ref-qualified_ member_functions*). — *Прим. пер.*

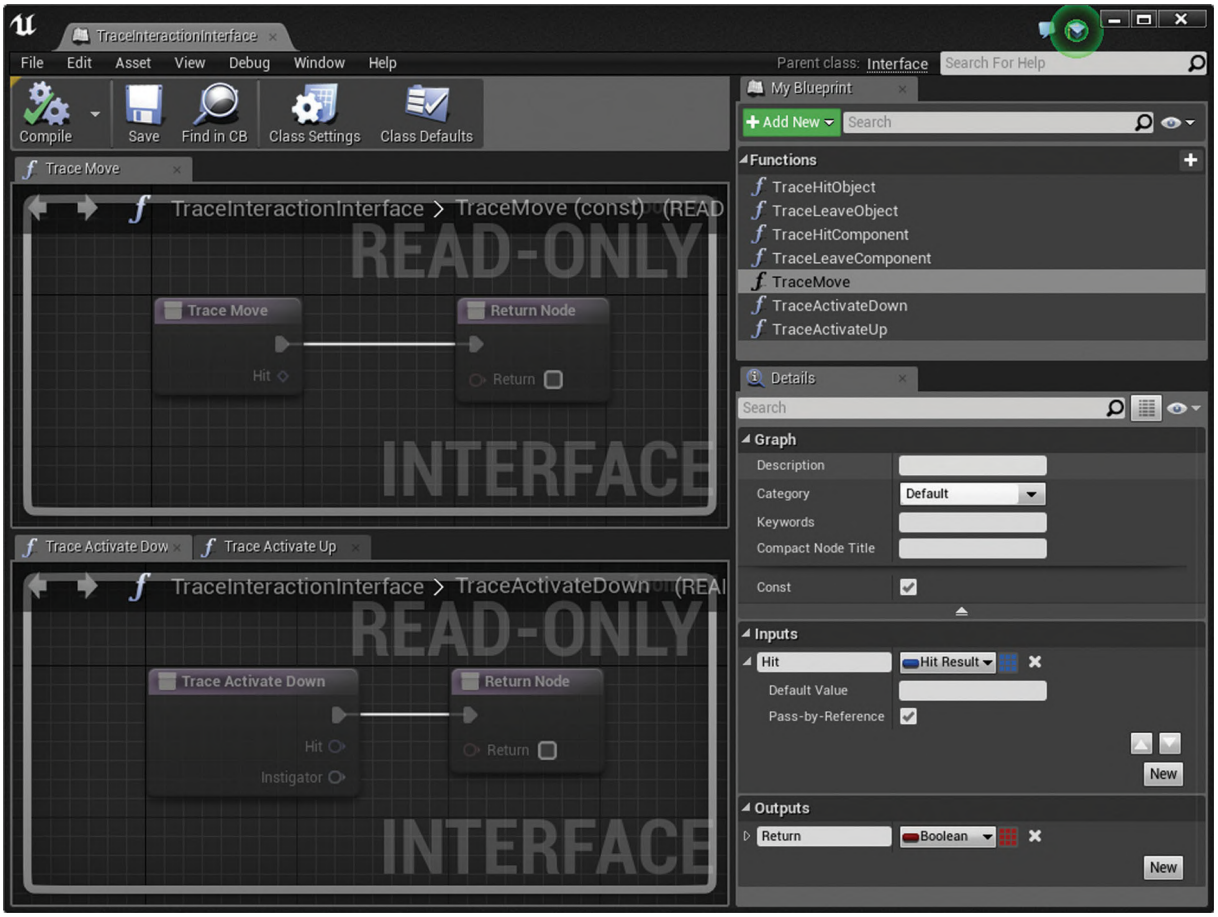

Рис. 4.8. Настройка интерфейса трейсинга взаимодействия: каждая функция имеет фиктивное возвращаемое значение, позволяющее ей оставаться в одном разделе реализующего *Blueprint*. Для улучшения производительности у *TraceMove* есть свой результат удара приведенный по ссылке

9. Создайте последние две функции `TraceActivateDown` и `TraceActivateUp`. Они используются для активации объекта попадания.
10. Как и в предыдущих функциях, подайте на вход переменную `Hit Result`, а также новую переменную *Pawn* с названием `Instigator`, в которую будет передаваться *Pawn*, активировавший объект.
11. Проверьте функции и добавьте в них выходную (*Output*) переменную с именем `Return` и типом `Boolean`. Это может показаться странным, но поскольку мы реализуем интерфейс на *Blueprints*, то при отсутствии переменных на выходе они будут отображаться как события в *Event Graph*. Поэтому при попытке разместить все функции в одном месте вы можете добавить выход, который на самом деле является мусором.

Все функции должны выглядеть как на рис. 4.8.

4.1.6. Компоненты взаимодействия

Основная часть логики интерфейса нужна нам для обращения в компонент *Interaction Component*.

В этом примере вы установите трассу в виде простой линии и с ее помощью организуете взаимодействие с объектом столкновения. Затем вы познакомитесь с несколькими методами, которые могут быть использованы для определения того, с какими объектами взаимодействовать.

4.1.6.1. Установка

1. Откройте `TraceIteractionComponent`.
2. Создайте две новые функции `LineTrace` и `InteractWithHit`. Первая отслеживает трассу, вторая берет выходное значение линии трассы и определяет, каким образом взаимодействовать с объектом попадания.
3. Создайте три новые переменные, `FocusedComponent` типа `Primitive Component`, `FocusedObject` типа `Actor`, `CurrentHit` типа `Hit Result`. Первые две содержат текущий компонент и объект, а третья содержит информацию о попадании.
4. Отметьте эти переменные модификатором `Private`. Вы не должны предоставлять доступ к ним другим *Blueprints*.
5. В *Event Graph* от узла `EventTick` создайте узел *Gate*, убедившись, что `Start Closed` не отмечено.
6. Создайте два новых события: `Enable` и `Disable`. Это позволит вам включать и отключать функции логики работы трассировки.
7. Соедините выход узла *Enable* с контактом `Open`.
8. Соедините выход узла *Disable* с контактом `Close`.

Начальная установка должна выглядеть как на рис. 4.9.

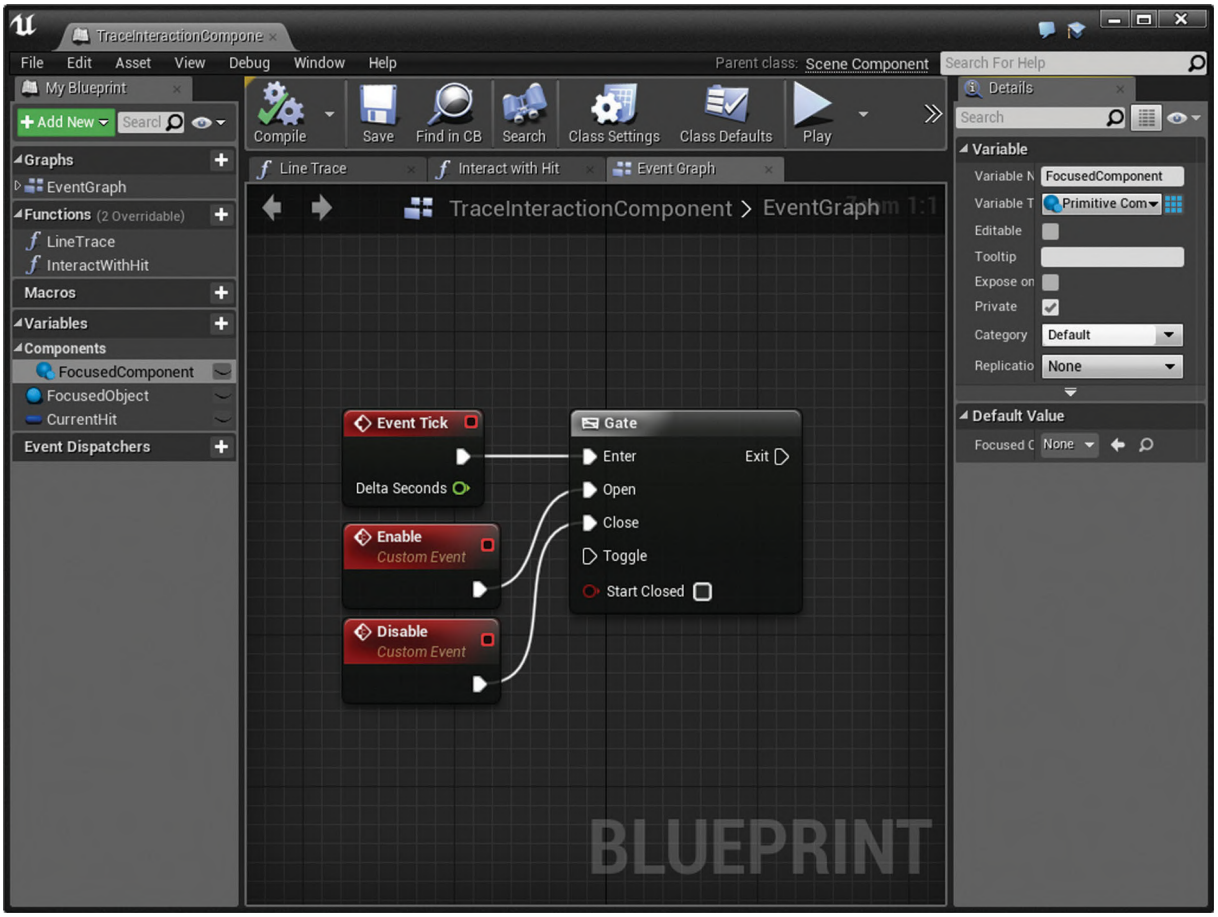

Рис. 4.9. Начальная установка компонента отслеживания. Компонент включен (*Enable*) по умолчанию, но доступны только события *Enable* и *Disable*

4.1.6.2. Использование трассы линии

Сейчас вам необходимо настроить функции, которые вы создали ранее.

Сначала посмотрим на функцию `LineTrace`.

1. Откройте функцию `LineTrace`.
2. Создайте новую переменную типа `Float`, поступающую на вход, и назовите ее `Distance`. Это расстояние отслеживания от компонента.
3. Создайте новую возвращаемую переменную типа `Hit Result` и назовите ее `Hit`.
4. Создайте узел *GetWorldLocation*.
5. Создайте узел *GetForwardVector*. Это даст вам прямое направление компонента взаимодействия и, в этом случае, направление взгляда игрока.
6. Создайте узел *LineTraceByChannel*, поставьте его между узлами входа и выхода функции.
7. Соедините выход узла *GetWorldLocation* с `Start` узла *LineTraceByChannel*.

8. Перетащите выход узла `GetForwardVector` и создайте узел `Vector * Float`, соедините `Distance` со вторым входом. Это умножит направляющий вектор на нужное нам расстояние.
9. Перетащите выход *GetWorldLocation* снова и создайте узел `Vector + Vector`, соединив второй вход с отмасштабированным направляющим вектором.
10. Соедините `Vector + Vector` с входом `End`.
11. Соедините выход `Out Hit` с узлом *Return*.

Результат должен быть как на рис. 4.10.

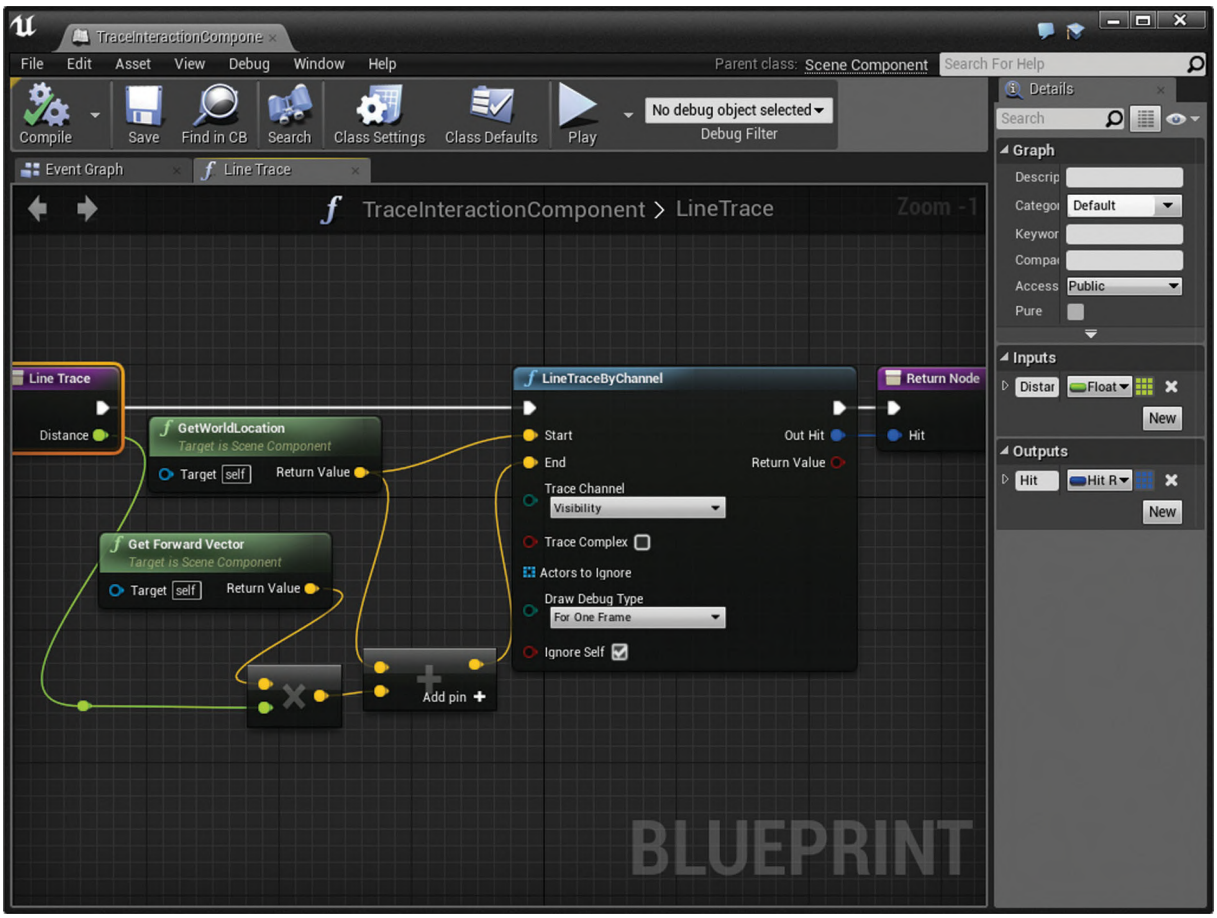

Рис. 4.10. Компонент отслеживания: функция *LineTrace*

4.1.6.3. Функция взаимодействия

Давайте добавим логику определения сообщений интерфейса для вызова на нашем объекте.

1. Откройте функцию `InteractWithHit`.
2. Добавьте три локальные переменные `InHit`, `InHitActor` и `InHitComponent` типов `Hit Result`, `Actor` и `Primitive Component`. Они действуют как временные переменные, чтобы предоставить вам легкий доступ к частям ввода и сделать ваш *Event Graph* чище.
3. Создайте входную переменную `Hit` типа `Hit Result` (рис. 4.11).

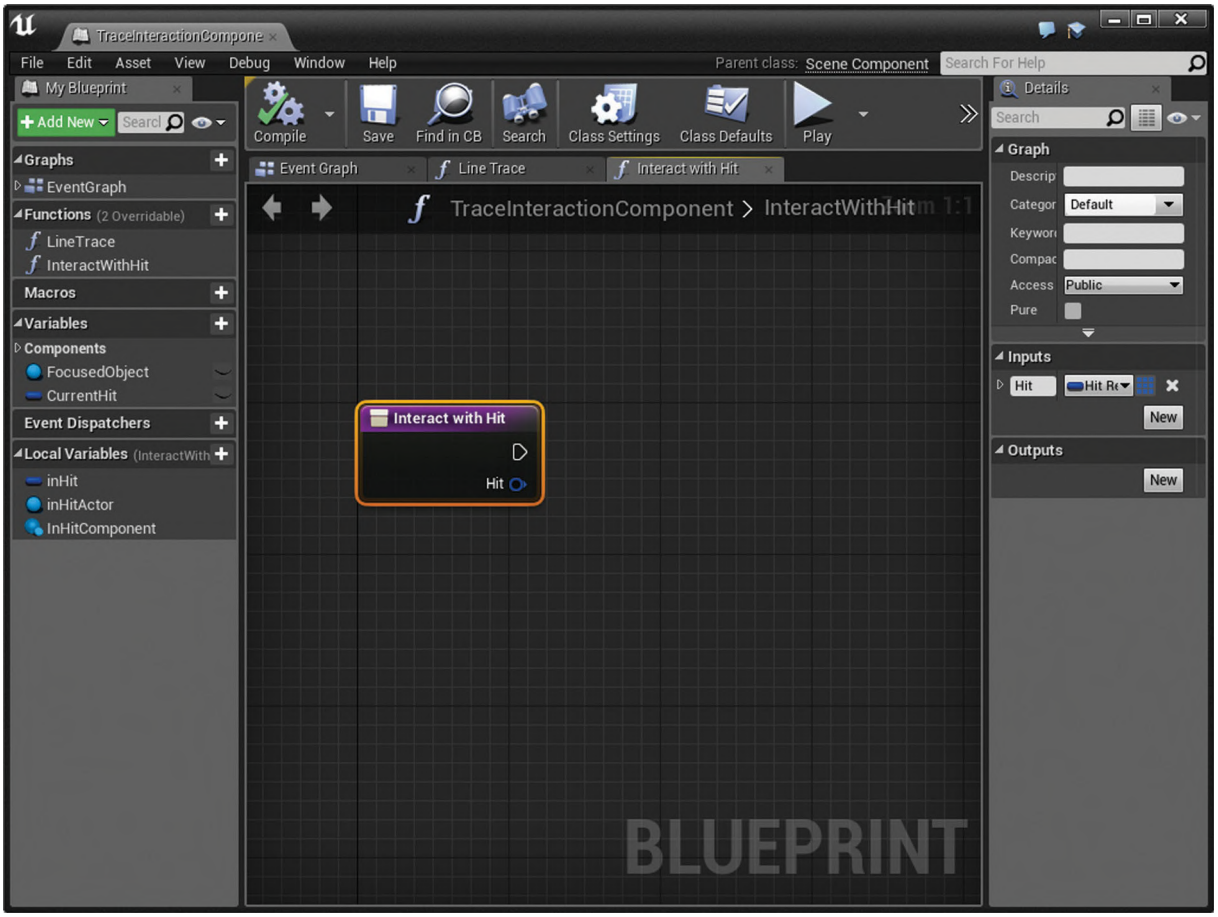

Рис. 4.11. Функция *InteractionWithHit*: локальные и входные переменные

4. Перетащите контакт выполнения, создайте сеттер для переменной `InHit` и соедините `Hit` с входным контактом.
5. Перетащите выход сеттер и создайте узел *BreakHitResult*.
6. Создайте сеттер для `InHitActor` и `InHitComponent`, соединив их с `Hit Actor` и `HitComponent` узла *BreakHitResult*.
7. Соедините контакт выполнения поочередно с `InHitActor` и `InHitComponent` (рис. 4.12).

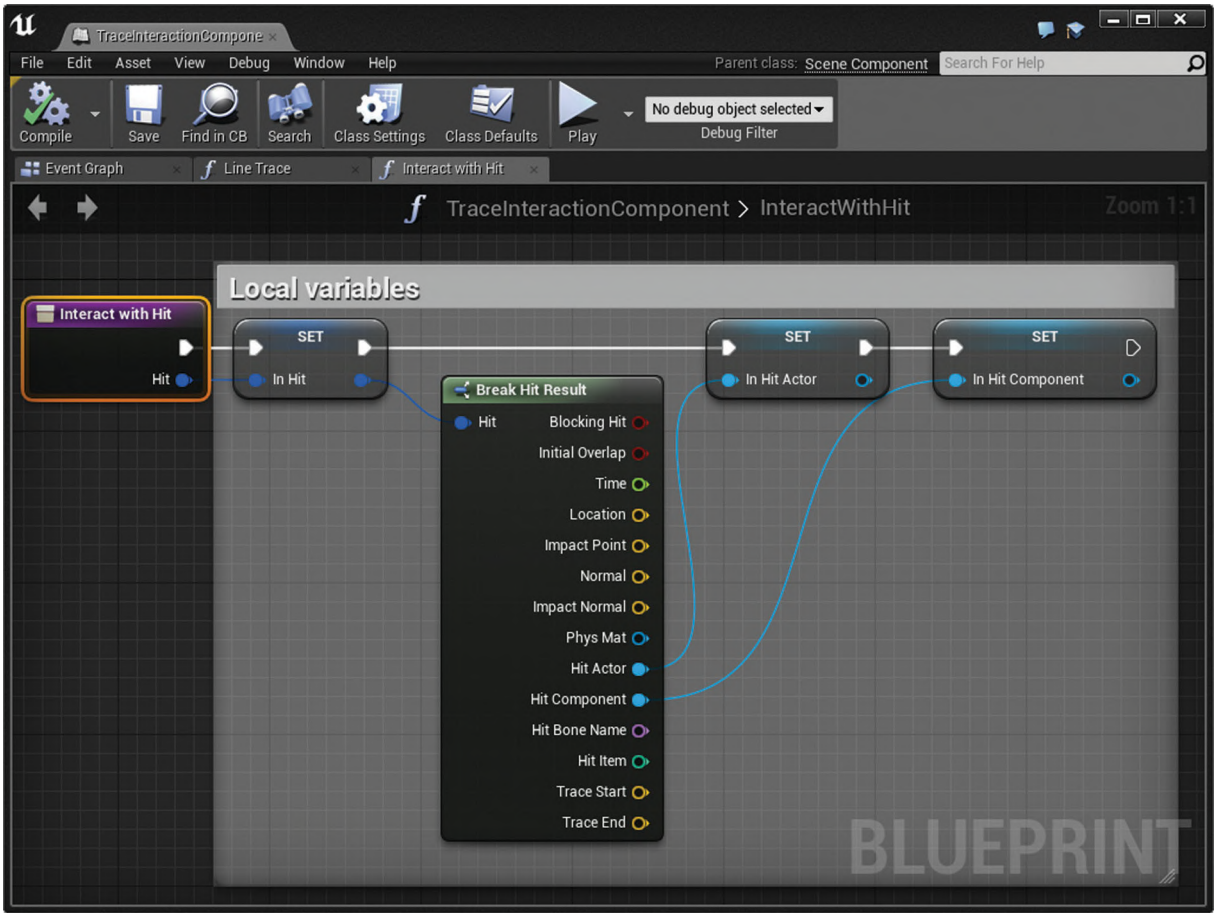

Рис. 4.12. Функция *InteractionWithHit*: установка локальных переменных

8. От сеттера `InHitComponent` создайте `Branch`.
9. Создайте геттер для `InHitActor` и `FocusedObject`.
10. Добавьте узел *Equal* (`Object`) и сравните оба геттера.
11. Соедините выход сравнения с *Condition* (рис. 4.13). Это позволит определить, смотрит ли игрок на тот же объект или начал смотреть на новый.

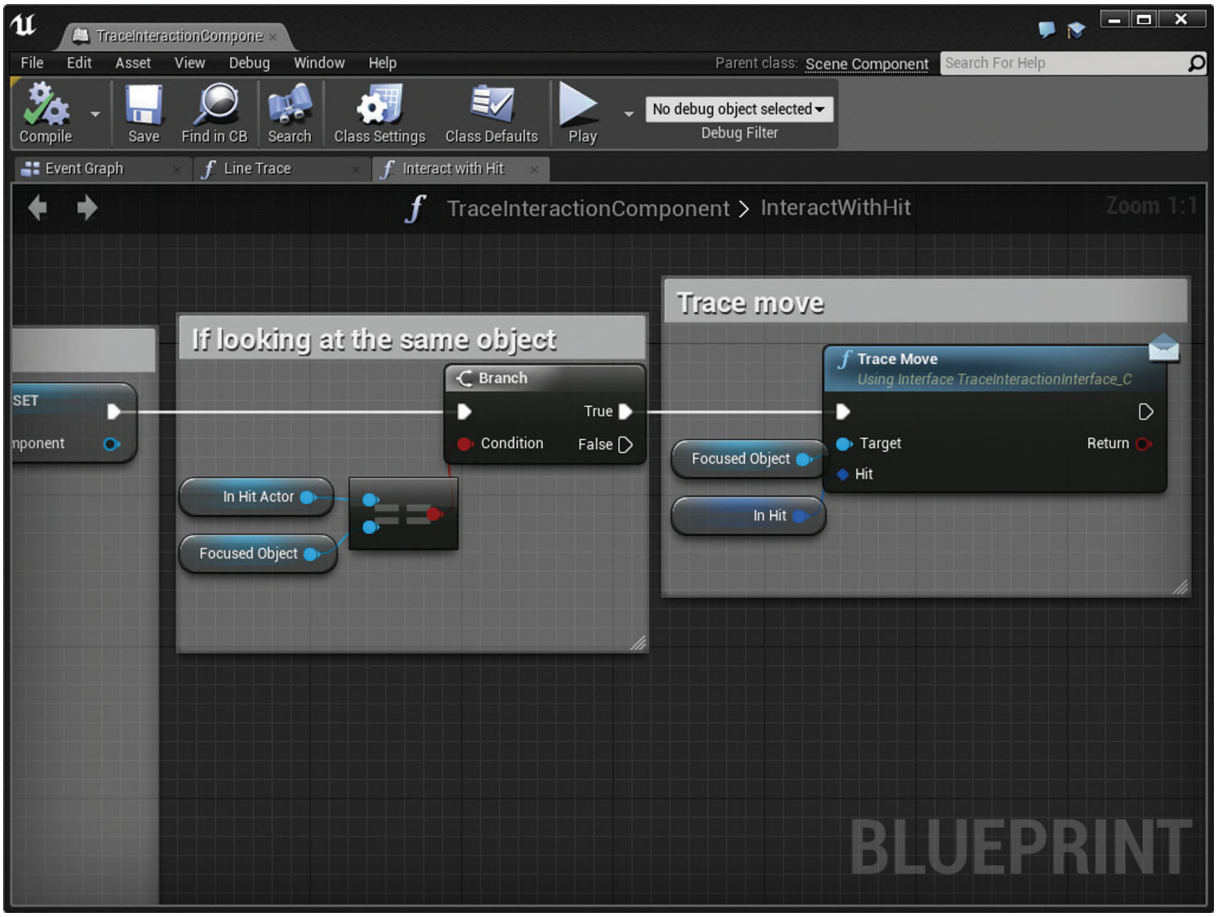

Рис. 4.13. Функция *InteractionWithHit*: вызов *TraceMove* при взгляде на тот же объект

12. Создайте два новых геттера: `FocusedObject` и `InHit`.
13. От геттера `FocusedObject` создайте вызов функции `TraceMove`. Это вызовет функцию `TraceMove`, если он реализует интерфейс, который вы создали на первом шаге.
14. Соедините геттер `InHit` с `Hit`. Соедините ветку `True` с узлом вызова функции `TraceMove` (см. рис. 4.13).
15. От вызова функции `TraceMove` создайте новый узел `Branch`.
16. В этот раз создайте два геттера для переменных `FocusedComponent` и `InHitComponent`. Добавьте узел *Equal* для них.
17. Соедините сравнение с контактом *Condition*. Это позволит вам определить, когда игрок смотрит на новый компонент текущего объекта.
18. Создайте три новых геттера для переменных `FocusedObject`, `InHit` и `FocusedComponent`.
19. От геттера переменной `FocusedObject` создайте вызов функции интерфейса `TraceLeaveComponent`, соедините `InHit` с контактом `Hit`, а `FocusedComponent` с `Component` (рис. 4.14). Это обеспечит, что вы отпустите текущий компонент, перед определением нового.

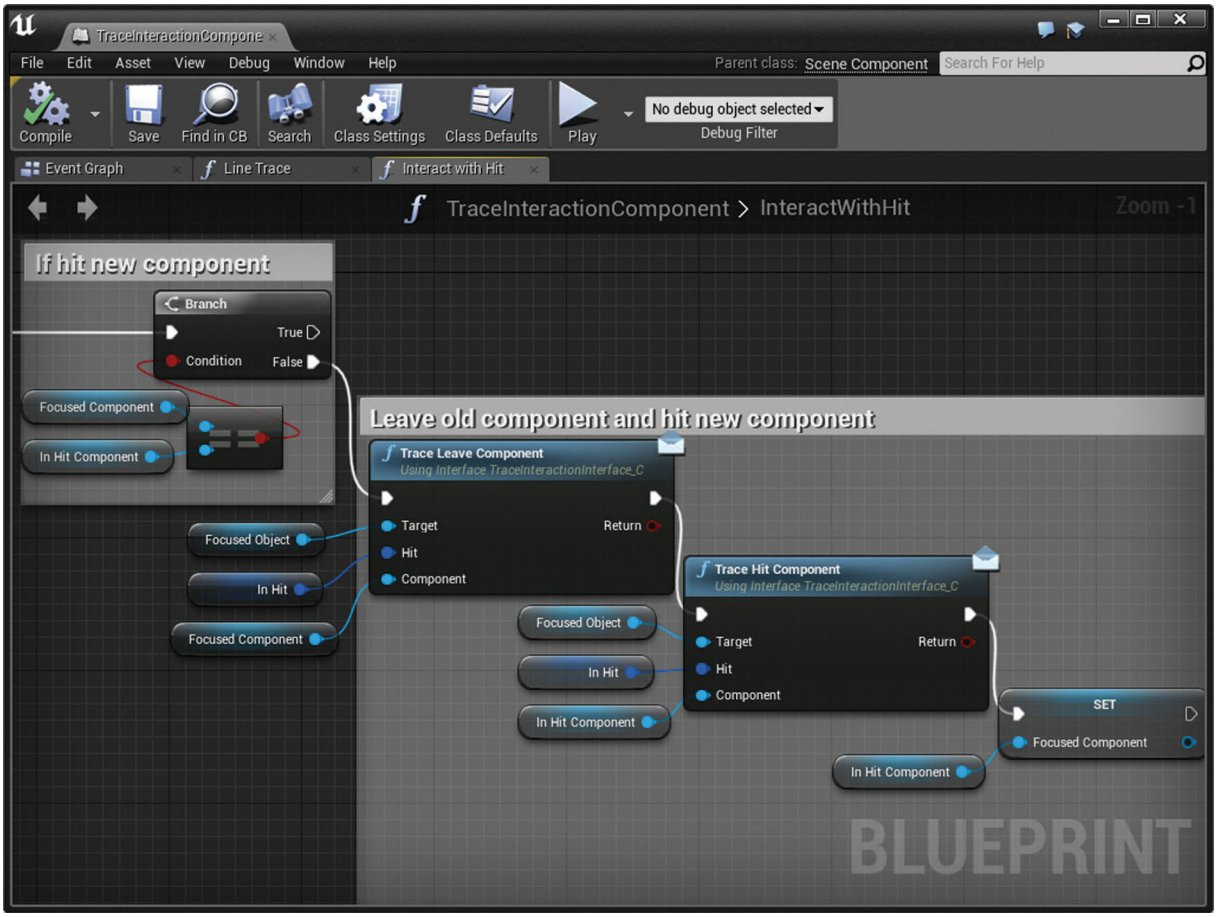

Рис. 4.14. Функция *InteractionWithHit*: вызов луча на новом компоненте

20. Соедините контакт вызова `False` с узлом `TraceLeaveComponent`.
21. Создайте снова три новых геттера для `FocusedObject`, `InHit`, `InHitComponent`.
22. От `FocusedObject` создайте вызов функции интерфейса `TraceHitComponent`, соединив `InHit` и `InHitComponent` с `Hit` и `Component`.
23. Соедините `TraceLeaveComponent` с `TraceHitComponent`.
24. Создайте новый сеттер для `FocusedComponent` и соедините его с новым геттером переменной `InHitComponent`.
25. Соедините контакт вызова `TraceHitComponent` с сеттером (рис. 4.14). Эта последовательность вызовов функций обеспечивает отпуск текущего компонента отслеживания, а затем определение и сохранение нового.
26. Вернитесь к узлу *Branch* после настройки локальных переменных (шаг 8), создайте новый геттер переменной `FocusedObject` и вызовите `TraceLeaveObject`. Соедините контакт вызова от `False` с этим узлом.
27. Создайте геттер переменной `InHit` и соедините его с `Hit`.

28. Создайте два новых сеттера переменных `FocusedObject` и `FocusedComponent`, соедините их последовательно вызовом после `TraceLeaveObject` (рис. 4.15). Оставьте контакты входов этих сеттеров пустыми, так как мы хотим освободить эти переменные.

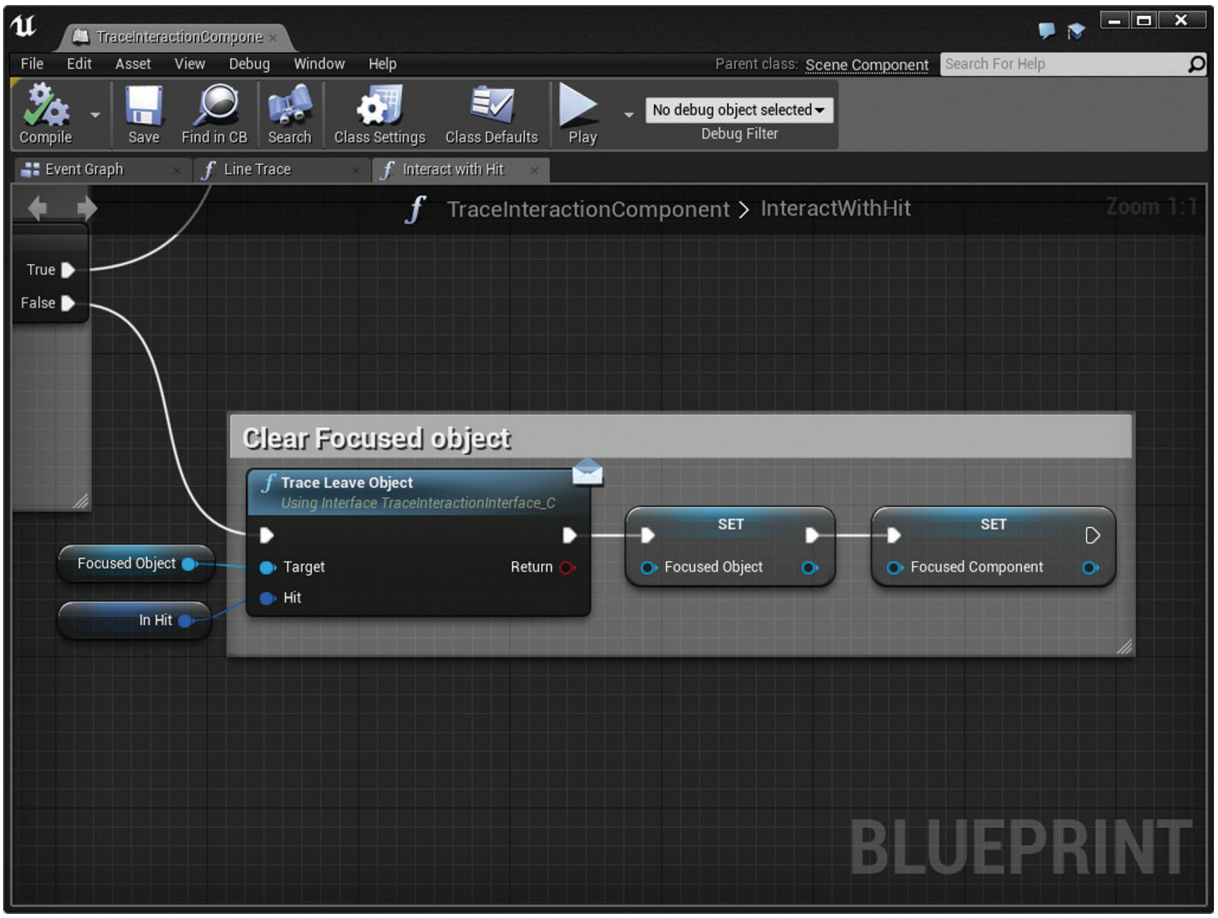

Рис. 4.15. Функция *InteractionWithHit*: освобождение переменной *FocusedObject*

29. От последнего сеттера создайте узел *Branch*.
30. Создайте новый геттер переменной `InHitActor` и вызовите от него `DoesImplementInterface`, выбрав `TraceInteractionInterface` как `Interface`. Соедините его с контактом *Condition* (рис. 4.16). Так мы обеспечим, что текущий пораженный лучом *Actor* реализует интерфейс.

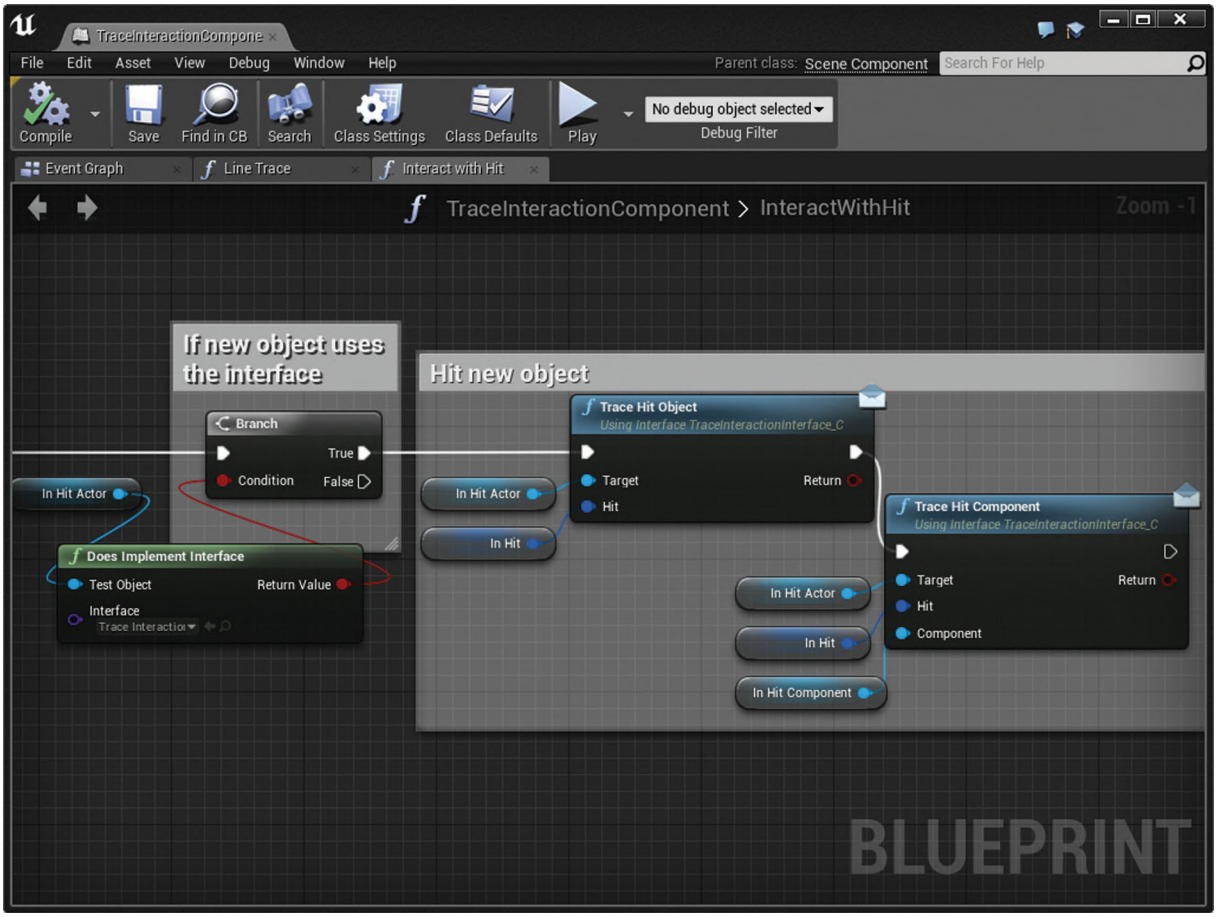

Рис. 4.16. Функция *InteractionWithHit*: вызов луча на новом объекте

31. Создайте два новых геттера для `InHitActor` и `InHit` и вызовите `TraceHitObject` от первого, после чего соедините `InHit` с `Hit`.
32. Соедините контакт вызова `True` с `TraceHitObject`.
33. Создайте три новых геттера для `InHitActor`, `InHIt` и `InHitComponent`, и вызовите `TraceHitComponnet` от `InHitActor`. Соедините два других геттера с нужными контактами.
34. Соедините `TraceHitObject` с `TraceHitComponnet` (см. рис. 4.15); это обеспечит, что вы вызовете луч на объект и компонент сразу.
35. Создайте два новых сеттера для `FocusedObject` и `FocesudComponnet`, соединив их друг за другом вызовом, а первый соедините с узлом `TraceHitComponent` (рис. 4.17).

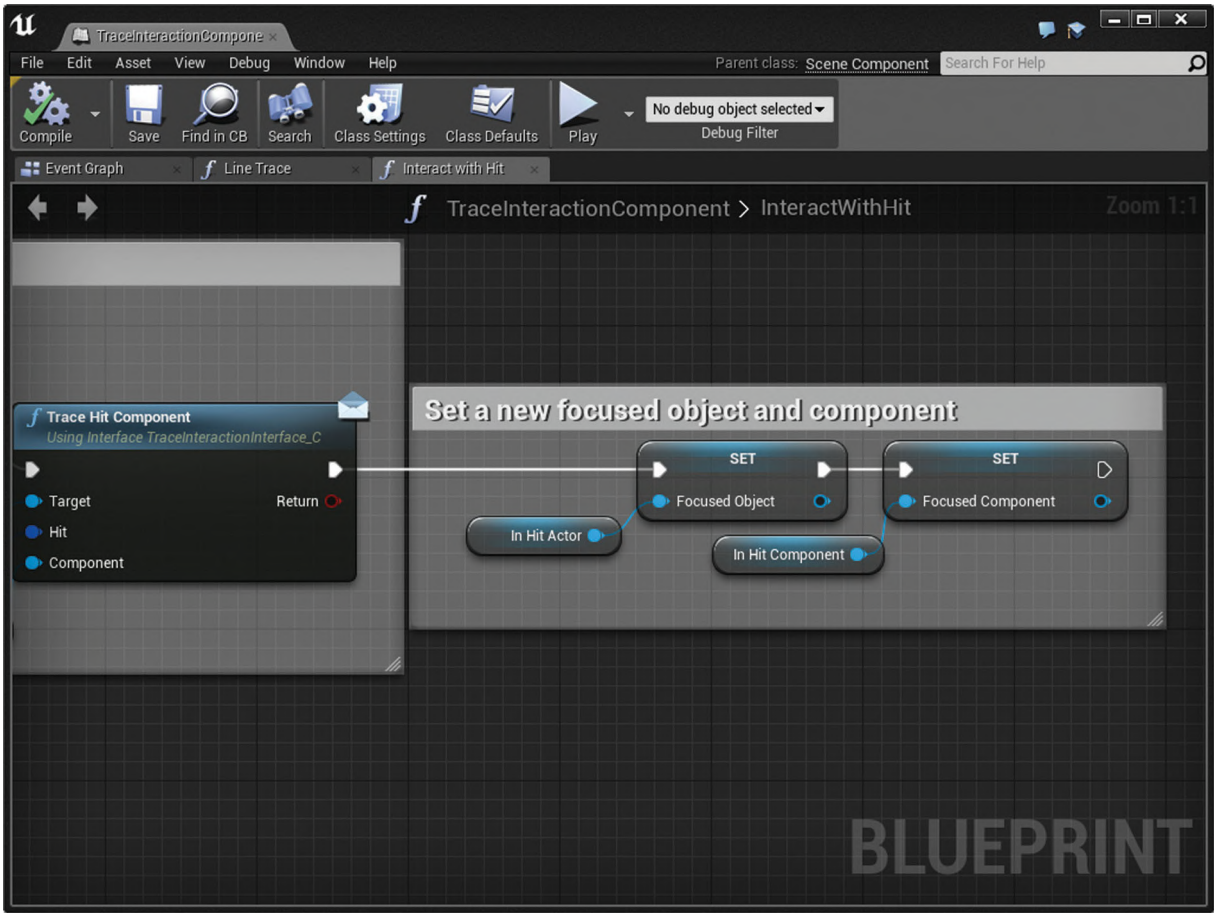

Рис. 4.17. Функция *InteractionWithHit*: установка нового объекта

36. Создайте два новых геттера для `InHitActor` и `InHitComponent`. Соедините их с сеттерами `FocusedObject` и `FocusedComponent`. Это сохранит текущие объект и компонент до следующего вызова функции.

4.1.6.4. Объединение функций

Сейчас, создав все функции для базовой трассировки взаимодействия, вам необходимо установить их в *Event Graph* и фактически создать функциональный компонент взаимодействия.

1. В *Event Graph* найдите `EventTick`, где вы настроили затвор для контроля этого компонента.
2. От узла *Gate* вызовите функцию `Line Trace`, установив `Distance` равный `1000`. Это обеспечит отслеживание с расстоянием в `10` метров.
3. От контакта `Hit` узла *Line Trace* создайте новый сеттер для `CurrerntHit`.
4. От сеттера вызовите `InteractWithHit` (рис. 4.18). Это будет создавать трассу при каждой обработке кадра, когда компонент включен, для взаимодействия с ним будут вызываться функции.

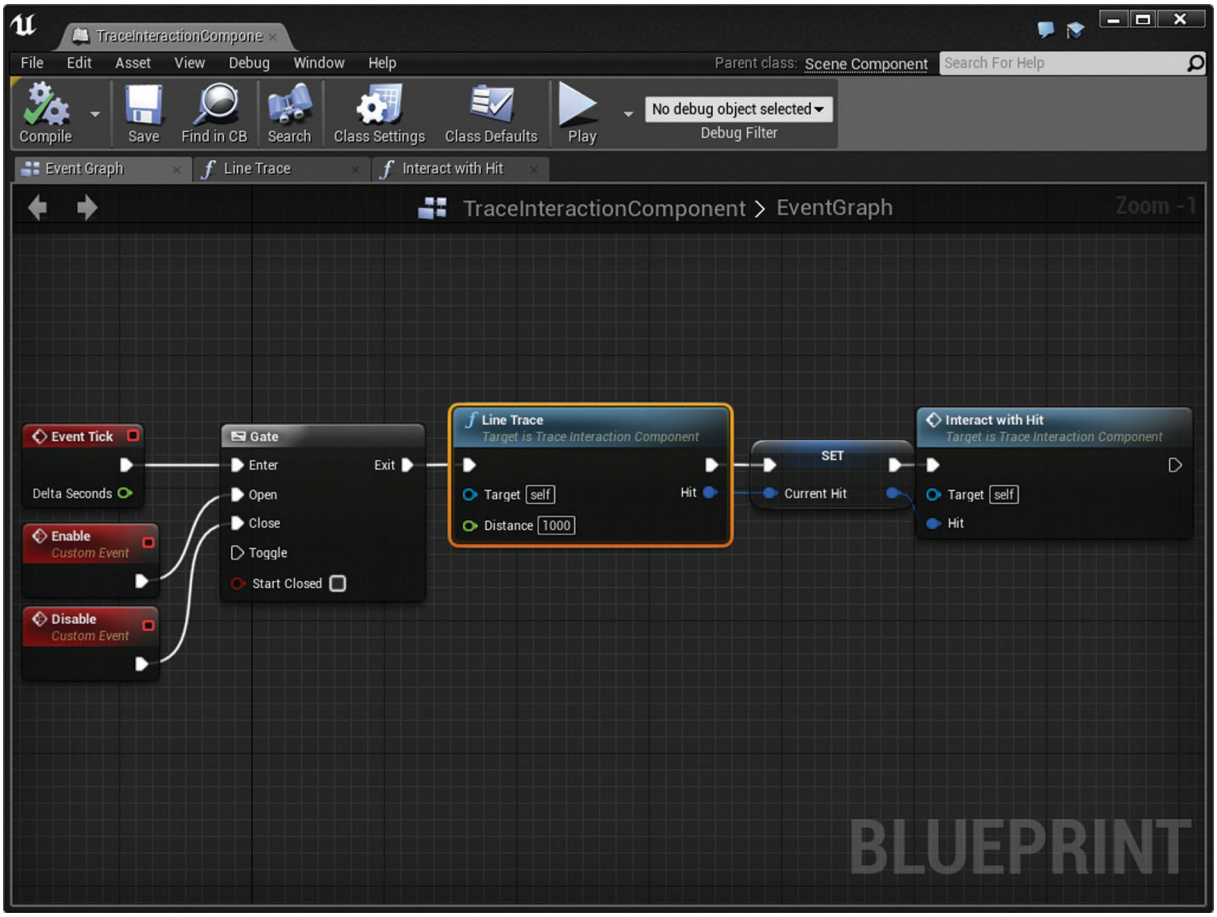

Рис. 4.18. Компонент отслеживания взаимодействия: вызов *LineTrace* и взаимодействие функций

5. Осталось побеспокоиться о *TraceActivateDown* и *TraceActiveteUp*. Создайте два события под названиями `ActivateDown` и `ActivateUp`, дав обоим на вход переменную `Instigator` типа `Pawn`.
6. Создайте по две пары геттеров для переменных `FocusedObject` и `CurrentHit` и разместите их под созданными событиями (рис. 4.19).

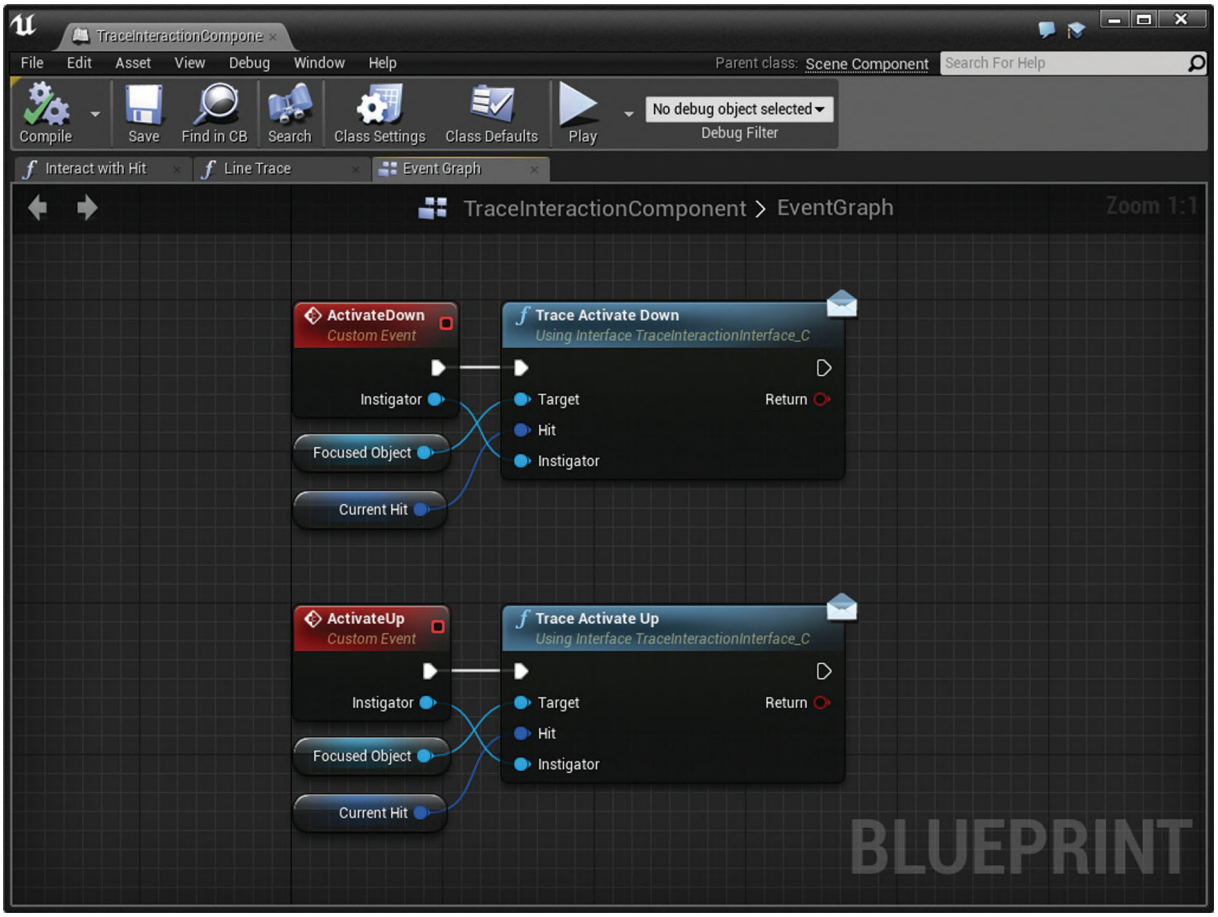

Рис. 4.19. Компонент отслеживания взаимодейсвтия: добавление событий *Activate*

7. От геттера переменной `FocusedObject` создайте узел *TraceActivateDown*, подсоедините к нему *CurrentHit*. От события `ActivateDown` соедините контакт выполнения и *Instigator*.
8. Сделайте то же самое от события `ActivateUp`, но с функцией *TraceActivateUp*. Теперь из других *Blueprint* вы можете активировать текущий `FocusedObject`.

4.1.7. Установка взаимодействия *Pawn*

Теперь у нас есть компонент и интерфейс, необходимый для системы взаимодействия. Пришло время добавить компоненты, необходимые для базового взаимодействия отслеживания *VR* к вашему *TraceInteractionPawn*.

1. Откройте *Blueprint TraceInteractionPawn*.
2. Добавьте три компонента: *Scene*, *Camera* и *TraceInteraction*.
3. Назовите *Scene* `CameraRoot` и сделайте *Camera* дочерним элементом (так как сам *Pawn* будет находиться на уровне земли, поднимете компонент *Camera* на 80 по оси *Z*, чтобы при проверке на компьютере ваша *Camera* была примерно на том же уровне, что и в *VR*).

4. Так как вы хотите, чтобы отслеживание приходило от направления взгляда игрока, поместите `TraceInteraction` в *Camera* (рис. 4.20). Если вы хотите, чтобы взаимодействие приходило с *Motion Controller*, просто подключите к этому компоненту контроллер.

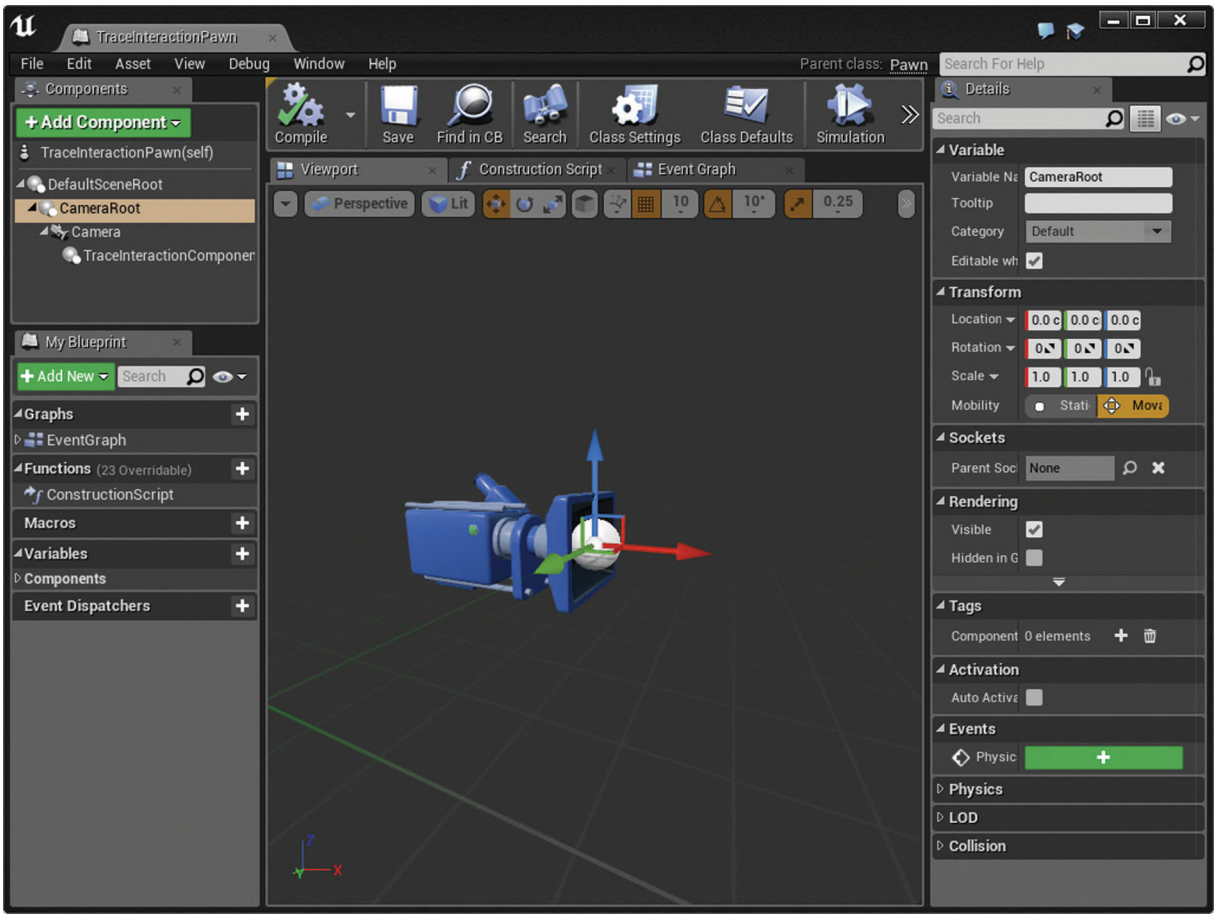

Рис. 4.20. Настройка *TraceInteractionPawn*

5. Осталось настроить *Interaction Component* на вызов `ActivateDown` и `ActivateUp`, когда нам нужно. Например, вы используете просто узел *LeftMouseButton* для активации. Создайте новое событие `LeftMouseButton` в *Event Graph*.
6. Переместите *TraceInteraction Component* в *Event Graph* и вызовите от него `ActivateDown` и `ActivateUp`, подключив к ним контакт вызова от `Pressed` и `Released` соответственно. Также установите `Self` в контакт `Instigator` (рис. 4.21).

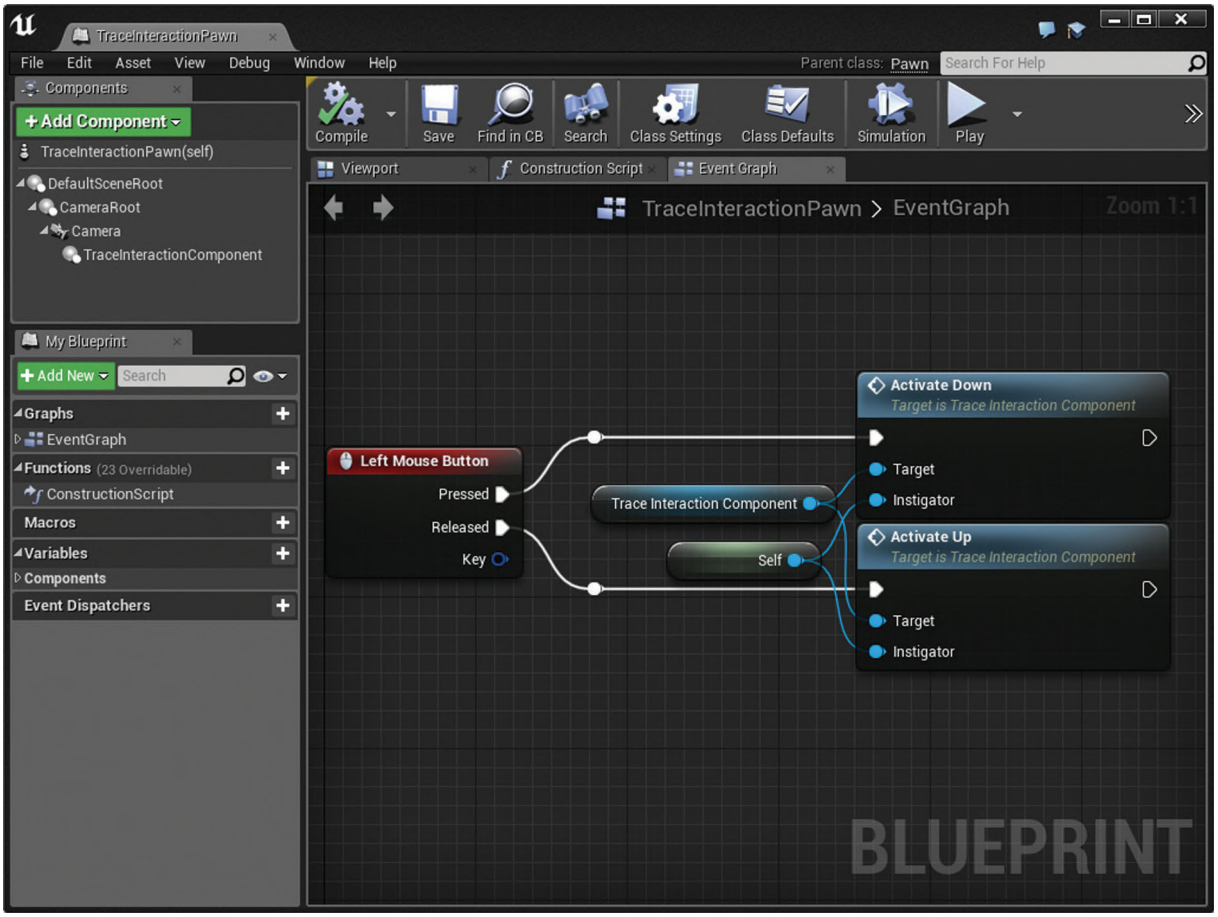

Рис. 4.21. Активация взаимодействия в *TraceInteractionPawn*.

4.2. Создание простого объекта взаимодействия

Сейчас есть система взаимодействия, теперь проведем ее тест на простом примере.

Для этого создадим простой *Blueprint* куба, который будет показывать, куда смотрит игрок, и реагировать изменением цвета.

1. Создайте *Actor Blueprint* и назовите его `TraceInteractionCube`.
2. Откройте *Blueprint* и создайте два новых компонента: *Cube* и *Arrow*.
3. Выберите *Arrow* и отключите `Hidden in Game` на вкладке *Details* (рис. 4.22). Это позволит вам видеть этот компонент в игре; это маркер, чтобы видеть, в какое место на кубе смотрит игрок.

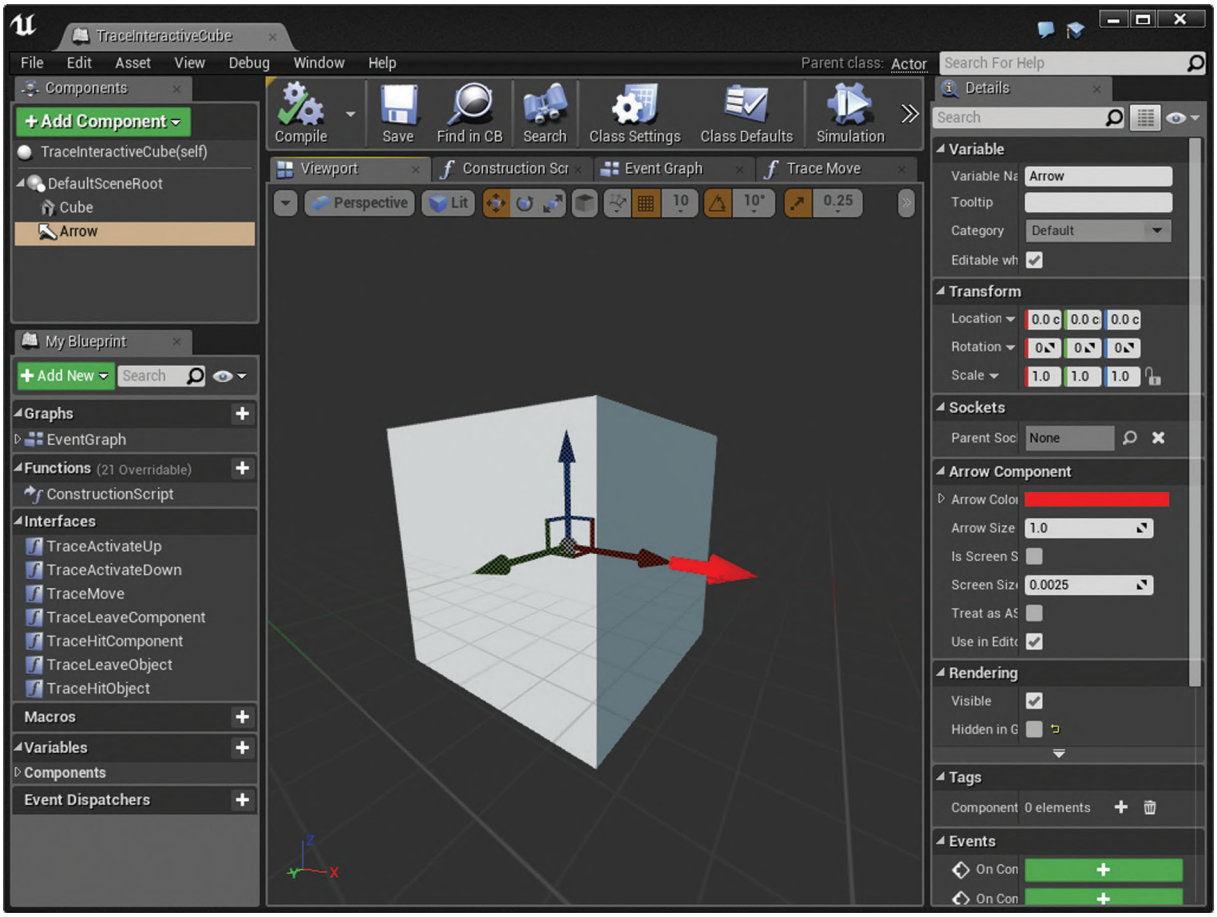

Рис. 4.22. *TraceInteractiveCube*: настройка компонента

4. Нажмите на *Class Settings*. Затем под панелью *Interfaces* на панели *Details* выберите кнопку *Add* и установите ваш *TraceInteractionInterface*. После компиляции *Blueprint* функции интерфейса появятся в секции *Interface* панели *My Blueprint*.
5. Откройте функцию `TraceHitObject` и переместите геттер для компонента *Cube*.
6. Вызовите функцию `SetVectorParametrValueOnMaterials` для этого компонента, установив вектор `(1.0, 0.0, 0.0)` и `Parameter Name Color`. Это сделает объект красным, когда отслеживание попадает в него лучом.
7. Поставьте этот узел между узлом вхождения и узлом *Return* (рис. 4.23).

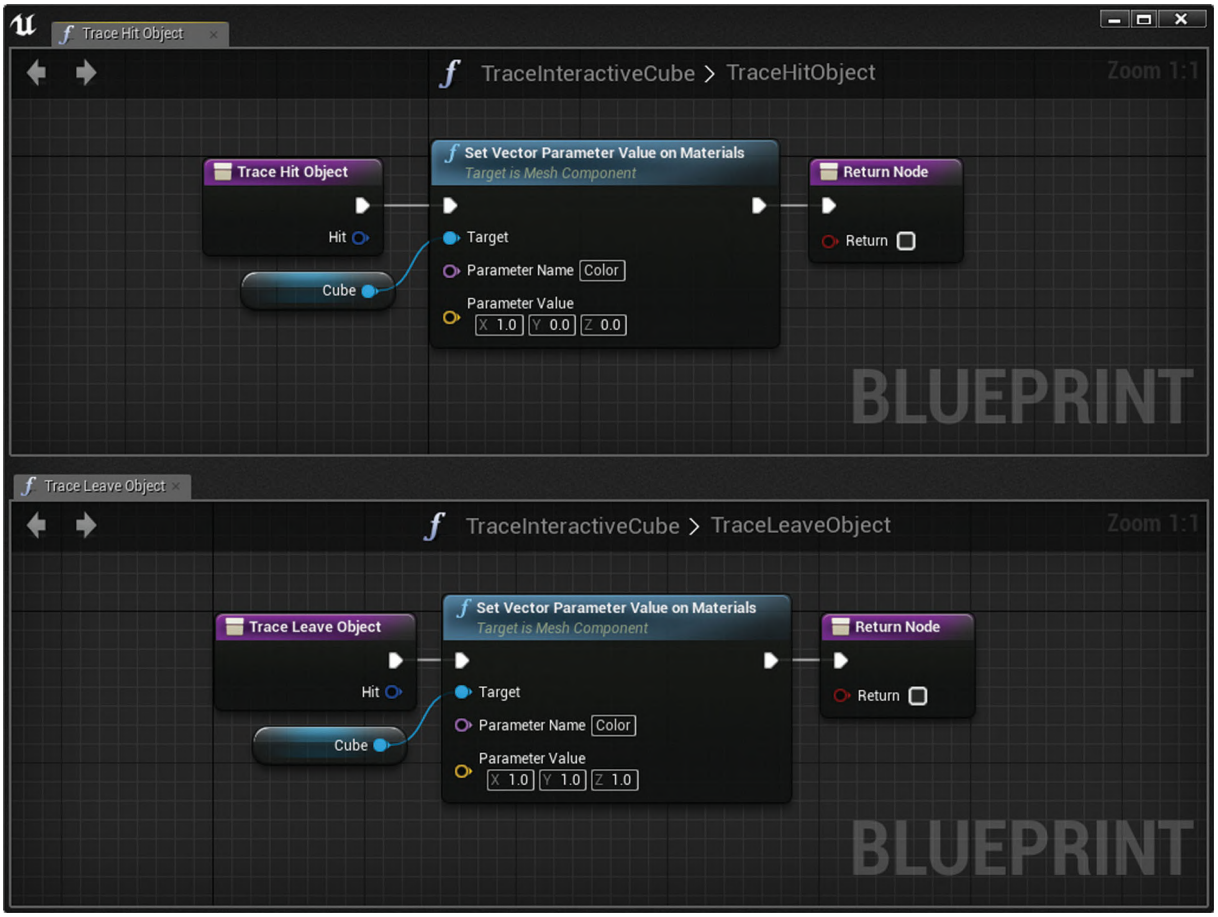

Рис. 4.23. *TraceInteractiveCube*: изменение цвета при столкновении

8. Повторите шаги 5 и 7 для функции `TraceLeaveObject`, но в этот раз поставьте вектор `(1.0, 1.0, 1.0)` (рис. 4.23). Это вернет кубу белый цвет, если игрок долго не смотрит на него.
9. Повторите шаги 5–7 для `TraceActivateDown` и `TraceActivateUp`, установив вектор `(0.0, 1.0, 0.0)` и `(1.0, 0.0, 0.0)` (рис. 4.24). Это установит зеленый цвет куба при активации и обратно красныйпри деактивация.

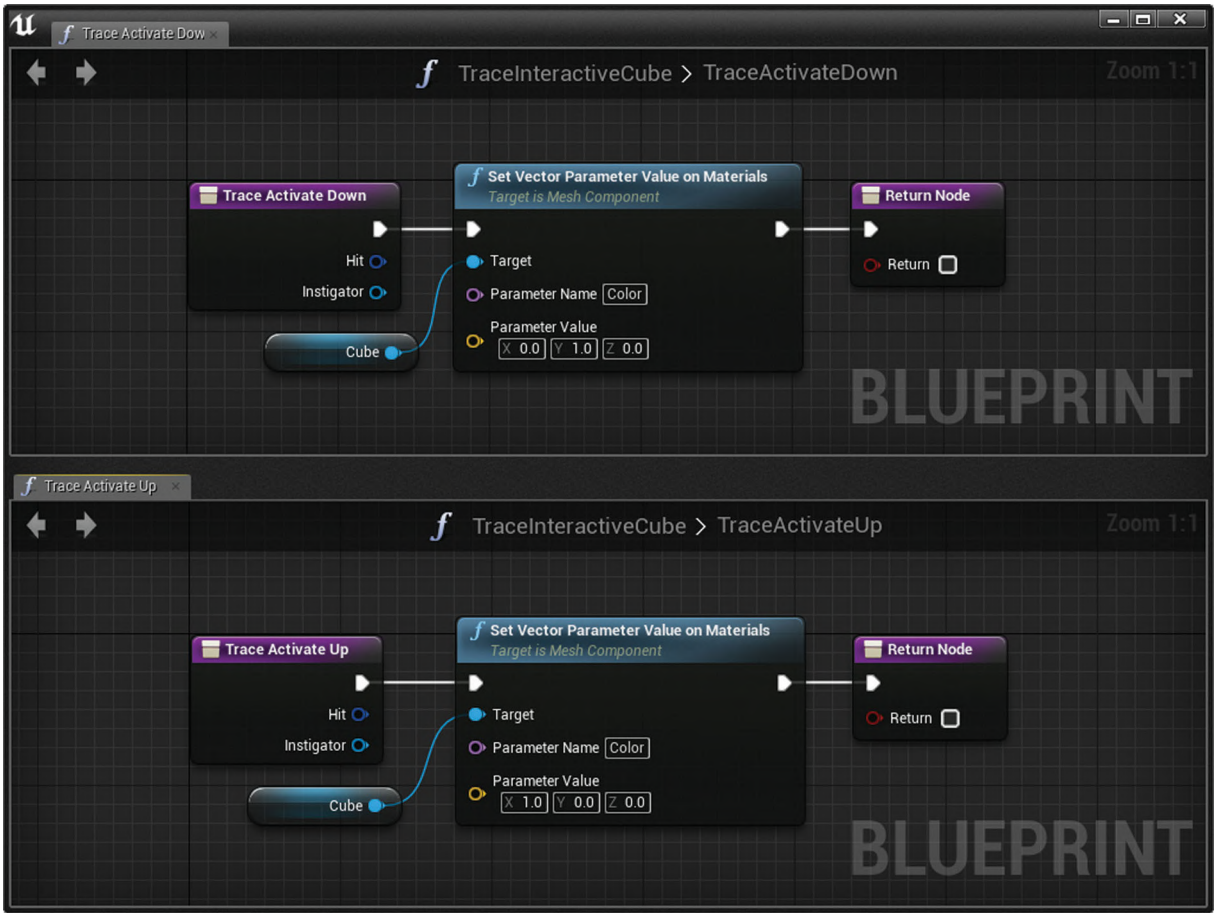

Рис. 4.24. *TraceInteractiveCube*: изменение цвета при *Activate Up* и *Down*.

10. Чтобы стрелка меняла свою позицию при каждой обработке кадра, когда игрок смотрит на куб, откройте `TraceMove`.
11. От контакта `Hit` создайте узел *BreakHitResult*.
12. Создайте геттер для компонента *Arrow*. Затем вызовите функцию `SetWorldLocationAndRotation`, соединив узлы вхождения в функцию и выхода.
13. Соедините `Location` узла *BreakHitResult* с `New Location` узла *SetWorldLocationAndRotation*.
14. От контакта `Normal` создайте узел *MakeRotFromX* и соедините его с `New Rotstion` узла *SetWorldLocationAndRotation* (рис. 4.25). Так как компонент *Arrow* смотрит в направлении *X*, это позволит перевернуть его лицом наружу от нормали.

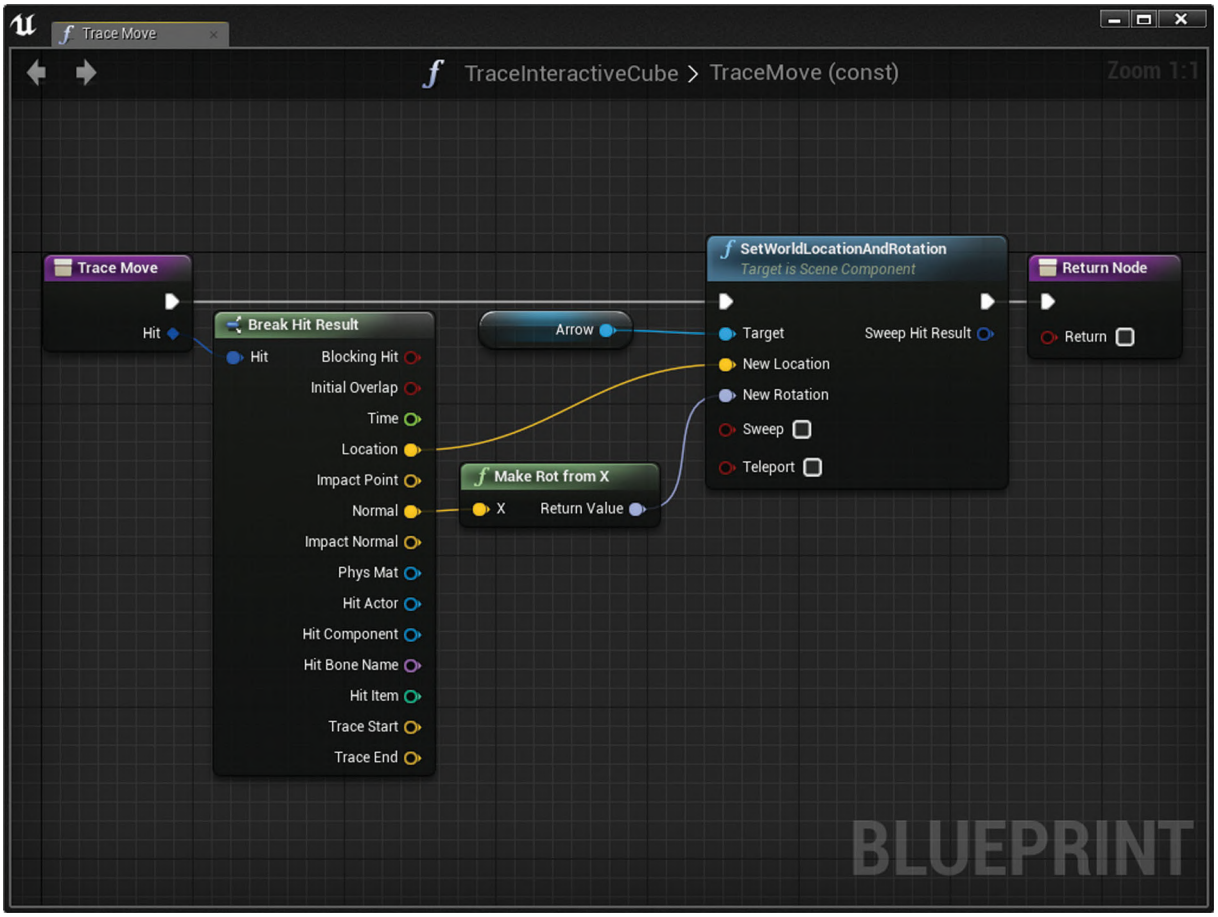

Рис. 4.25. *TraceInteractiveCube*: *TraceMove*

Настроив все необходимое, вы можете добавить оба ваших *Pawn* (либо создав *Game Mode* и сказать ему использовать ваш *Pawn*, либо установить ваш *Pawn* на уровень и включить *Auto Possess*) и ваш куб на сцену и наслаждаться рождественским духом красного и зеленого куба.

4.3. Заключение

Эта глава показала, как работает трассировка в *UE*4 и как использовать ее в *VR*-играх.

Вы познакомились с интерфейсами и узнали, как они справляются с взаимодействием объектов. Также вы воздали модульную систему трассировки, которую можете переносить и использовать в будущих проектах.

4.4. Упражнения

В этой части вы создали систему взаимодействия, которая использует трассировку линии для нахождения интерактивных объектов; она хороша для многих случаев, но имеет и недостатки.

В основном при использовании одиночной трассировки пользователям бывает трудно взаимодействовать именно с нужными объектами. Это особенно заметно на больших расстояниях. Поэтому может быть полезно использовать какую-либо другую метрику, чтобы оценить, куда пользователь смотрит, и учесть это в системе взаимодействия.

Один из распространенных методов для этого — использовать знания из векторной алгебры, чтобы найти разницу между вектором взгляда пользователя и вектором от игрока к интерактивному объекту.

Для этого можно использовать скалярное произведение, которое представляет собой форму умножения двух векторов, возвращающее в результате число. Скалярное произведение вычисляется путем взятия каждой пары координат в двух векторах, умножения их, а затем суммирования. Сначала это может показаться немного сложным, но, к счастью, *UE4* заботится о реализации, и все, что вам нужно знать, это то, что представляет собой конечное число после скалярного произведения.

Например, входные данные скалярного произведения нормированы, то есть имеют длину 1. Это означает, что оба вектора представляют направления, а не фактические местоположения или расстояния. Поскольку входные данные имеют длину 1, скалярное произведение этих двух векторов даст на выходе косинус угла между ними (рис. 2.26). Таким образом, если вы вычисляете скалярное произведение между тем, куда пользователь смотрит, и направлением, в котором находятся относительно его головы некоторые интерактивные объекты, вы можете получить значение из диапазона –1 и 1, которое показывает, насколько близко от объекта находится взгляд пользователя (1 смотрит прямо на и –1 — смотрит прямо от объекта).

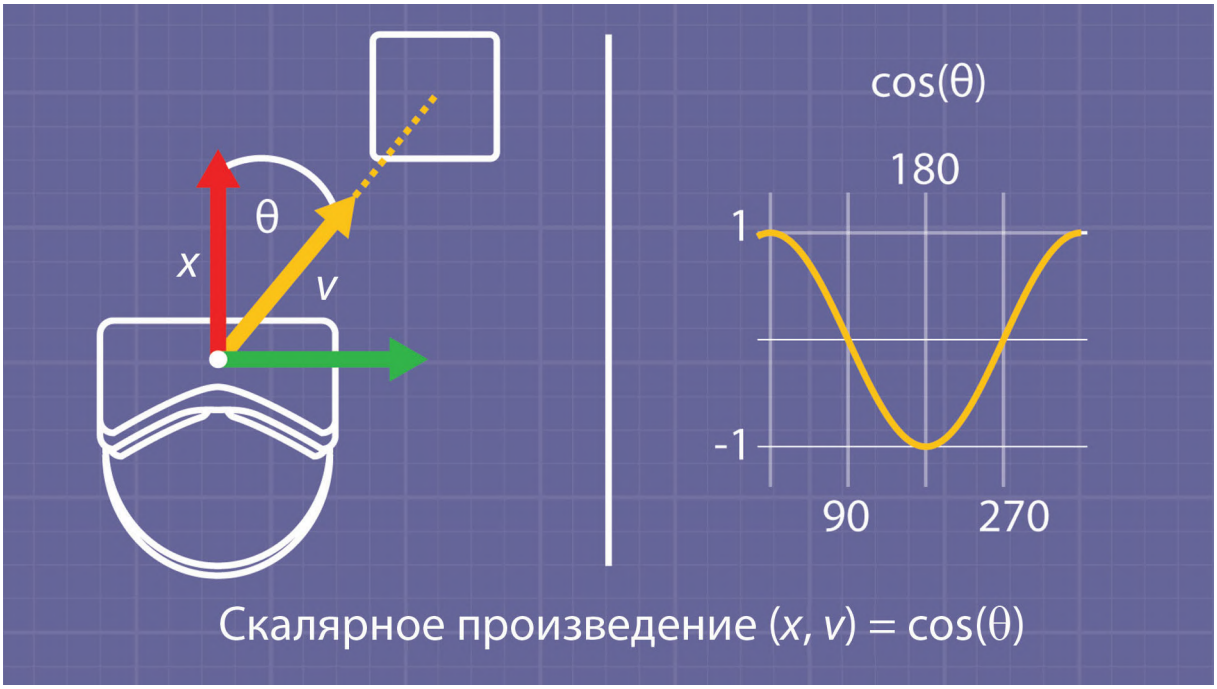

Рис. 4.26. Пример скалярного произведения. И *x*, и *v* являются нормализированными векторами, где *x* представляет направление взгляда игрока, а *v* — направление от головы игрока к объекту. Угол θ представляет собой угол между *x* и *v* в градусах

Это простой способ предугадывать действия пользователя, и он может быть использован, если нужно реализовать помощь в прицеливании (*aim assist*) в вашей игре. Для этого вы можете просто перебрать все объекты, которым хотите помочь, и найти тот, который имеет наибольшее значение скалярного произведения (или объект, к которому пользователь смотрит ближе всего).

Следует обратить внимание, что при использовании скалярного произведения для понимания намерения пользователя по его взгляду нормализация этих значений принимает собственную длину вектора. Побочный эффект заключается в том, что если один объект находится близко к пользователю, а другой — далеко, то более удаленный объект имеет более короткий угол к направлению взгляда пользователя, он может быть выбран, даже если пользователь пытается выбрать более близкий.

Один из способов справиться с этим — это вместо того, чтобы максимизировать скалярное произведение между пользователем и интерактивными объектами, минимизировать длину дуги между пользователем и объектами. Это расстояние, которое создается окружностью, заданной направлением взгляда пользователя и положением объекта (рис. 4.27 и 4.28).

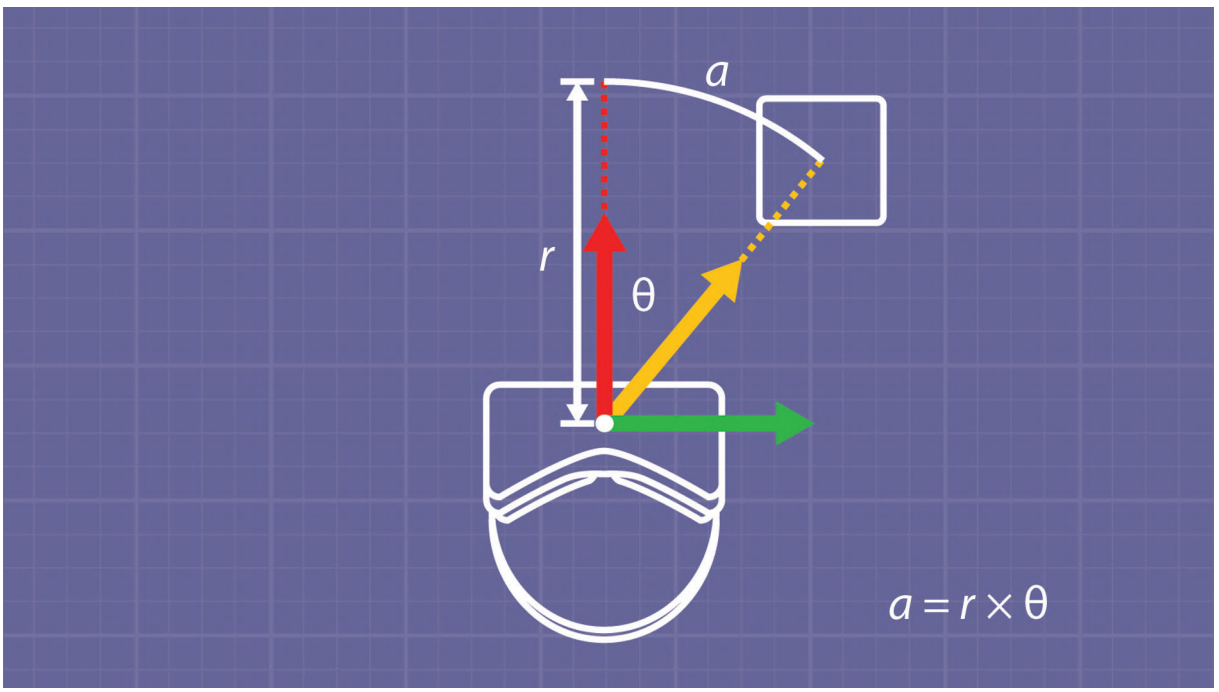

Рис. 4.27. Пример длины дуги. Длина дуги рассчитывается путем умножения радиуса *r* и угла θ в радианах

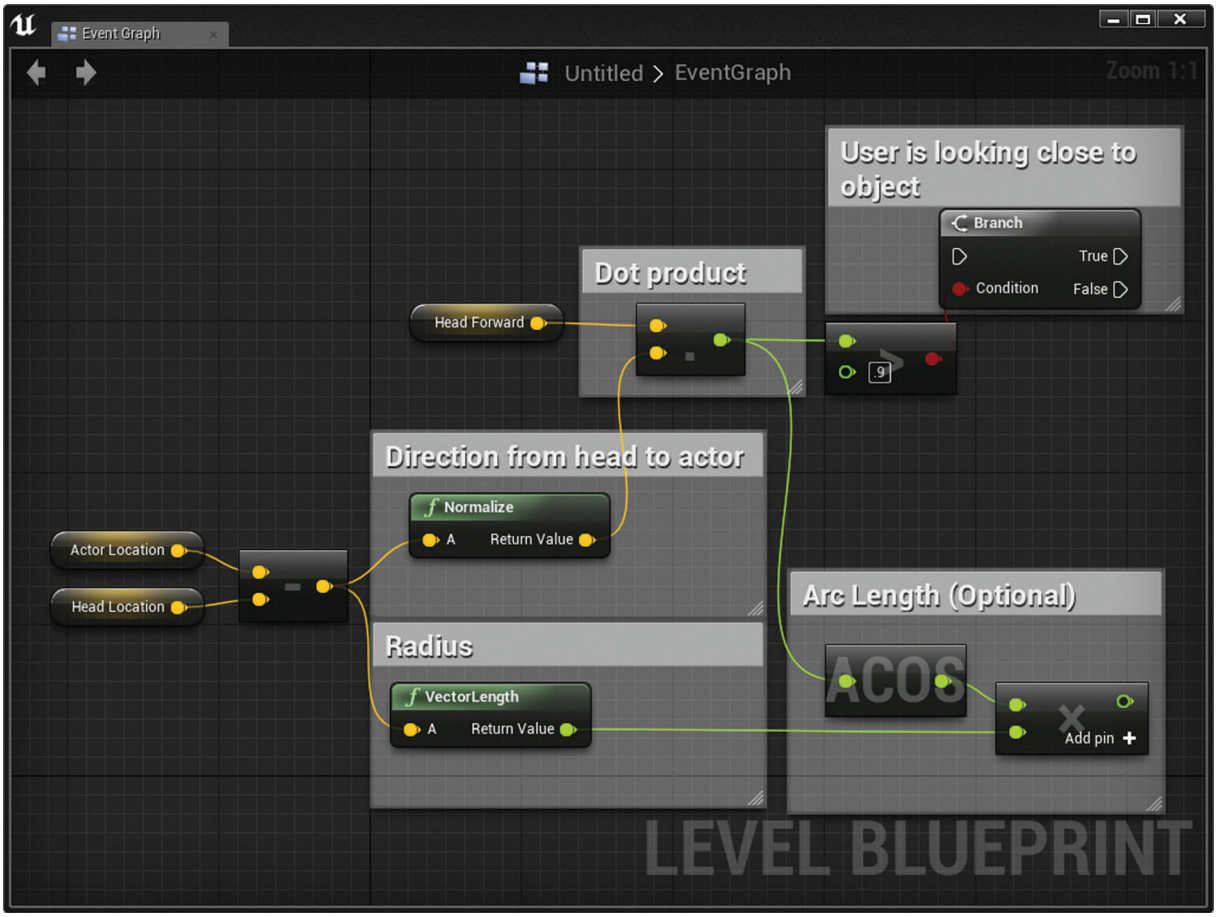

Рис. 4.28. Вычисление скалярного произведения между местоположением *Actor* и головой вперед с дополнительным расчетом длины дуги между актером и головой вперед

Одним из недостатков использования метода скалярного произведения для обнаружения взаимодействия с пользователем является то, что по умолчанию он не будет обрабатывать столкновения; это может привести к тому, что пользователи смогут взаимодействовать с объектами через стены и другие твердые поверхности. Одно из решений этой проблемы — проверить столкновение между пользователем и объектами, которые вы тестируете, отслеживая их. Это, однако, может иметь высокую вычислительную сложность, если на сцене много интерактивных объектов, для каждого из которых потребуется трассировка.

Другой способ справиться с этой проблемой — просто убедиться, что нет ничего, что может затенить ваши интерактивные объекты. Это, очевидно, может быть гарантировано только в меню и других контролируемых сценариях; однако, когда это так, метод скалярного произведения дает вам довольно дешевый и удобный способ обнаружения взаимодействий в игре.

Если вы хотите сохранить преимущества обнаружения столкновений, которые дает трассировка линии, но желаете большей гибкости в процессе выделения и согласны потратить немного больше вычислительных ресурсов на трассировку, используйте одну из трасс фигур. Создание трассы сферы с радиусом больше 1 позволит прицеливаться удобнее и останется относительно дешевой для вычисления.

На рис. 4.29 показан пример создания функции трассировки сферой, которая может использоваться в компоненте отслеживания взаимодействия.

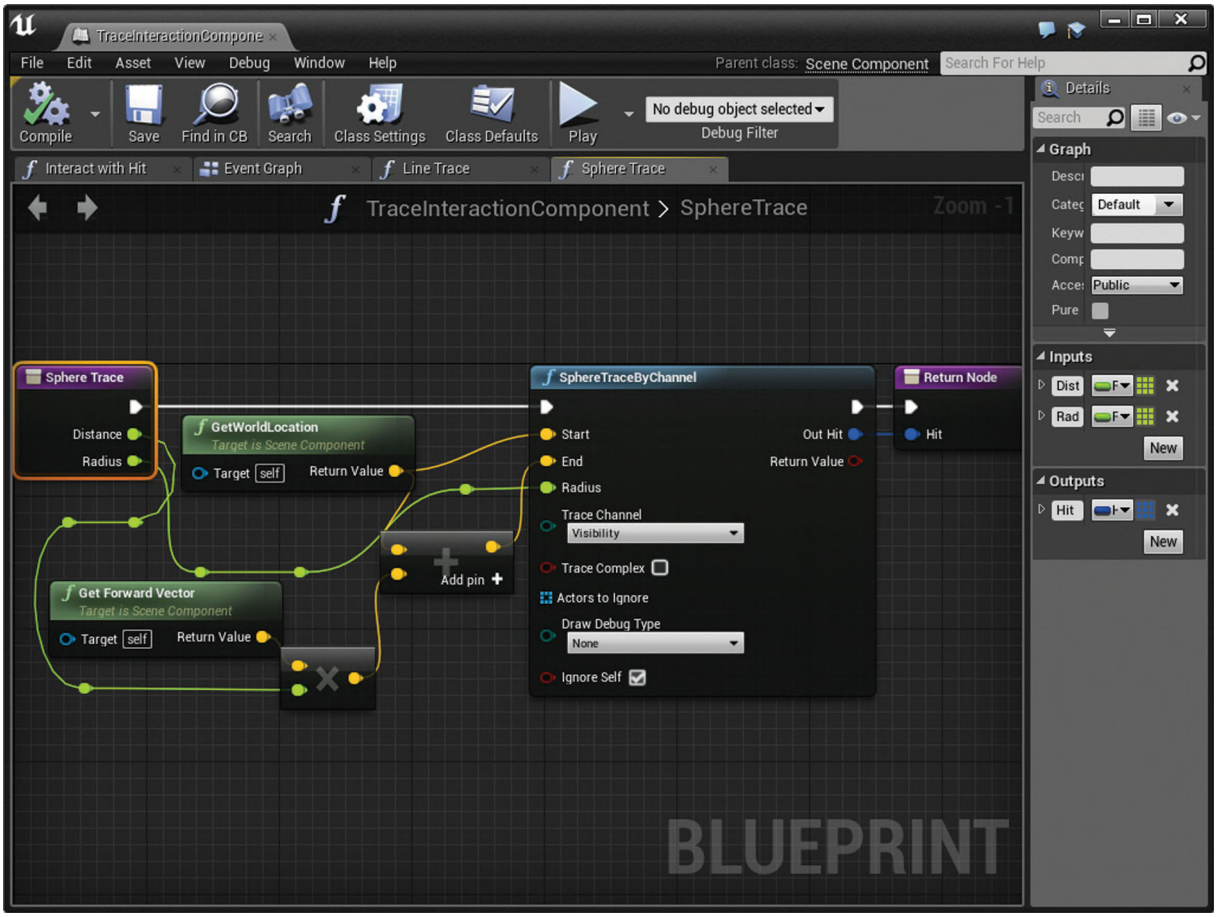

Рис. 4.29. Простая функция трассировки сферой для компонента отслеживания взаимодействия

В качестве упражнения добавьте трассу сферы в компонент взаимодействия трассировки и проверьте, упрощает ли она использование.

В качестве еще одного упражнения добавьте *aim assist* с помощью скалярного произведения. Для этого вам понадобится список всех объектов, с которыми вы хотите взаимодействовать. Подумайте, о лучшем способе получения этого списка.

ГЛАВА 5

ТЕЛЕПОРТАЦИЯ

Телепортация — один из основных методов искусственного перемещения на данном этапе развития *VR*. Эта глава расширяет трассировку, описанную в главе 4, до простого механизма телепортации.

Лучшая форма передвижения в *VR* — это, несомненно, отсутствие искусственного перемещения, вместо него лучше полагаться на индивидуальное сопоставление реальных действий с персонажами *VR*. Однако этот подход порождает ограничивающий набор требований, поэтому многие игры и даже целые жанры не могут полагаться только на него.

Потребность работать с персонажами, перемещающимися дальше, чем физические пределы движений пользователя, бросает вызов разработчику и требует оригинальных проектных решений. Необходимость и обоснование этих решений рассматриваются в главе 9 «Перемещение в *VR*». Возвращаясь к содержанию текущей главы, напомним, что она знакомит вас с одним из наиболее популярных решений проблемы передвижения в *VR* — телепортацией.

Чтобы избежать зрительного и вестибулярного несоответствия, которое может возникнуть, когда игроки видят, что их персонаж движется, в то время как они, по сути, неподвижны в реальном мире, многие разработчики полагаются на систему телепортации, которая мгновенно перемещает игрока из текущего местоположения в другое.

Доказано, что подобный одиночный «взрыв» менее болезнен для большинства пользователей *VR*, поэтому встречается в большинстве созданных в настоящее время *VR*-игр. Однако у него есть свои недостатки, главным образом связанные с дезориентацией. Физически мы не умеем телепортироваться (по крайней мере, пока), поэтому в *VR*-игре пользователь может быть на мгновение сбит с толку при выходе из телепорта. Существуют различные способы предотвратить этот эффект, начиная от простого — показывать след от того, откуда пришел пользователь, до более сложного — показывать пользователю портал в местоположение телепорта перед телепортацией. В этой главе, однако, эти методы не рассматриваются. Вместо этого мы сфокусируемся на создании простой системы телепортации и предложим читателю самому провести (забавные) эксперименты.

5.1. Настройка телепортации

Чтобы отследить телепортацию, вы будете использовать трассы; к счастью, вы уже создали модульный компонент взаимодействия трассировки *VR* и интерфейс, который может быть адаптирован в соответствии с вашими потребностями. Если вы этого не сделали, просмотрите главу 4 «Взаимодействие на основе трассировки», которая проведет вас через всю соответствующую информацию о создании модульной системы для обнаружения взаимодействий трассировки в ваших мирах.

Теперь, когда у вас есть база для взаимодействия трассировки (от завершения предыдущей главы или загрузки ее исходного кода), пришло время расширить ее для размещения телепортации.

5.1.1. Параболическая трассировка

При открытии компонента взаимодействия трассировки обратите внимание, что в настоящее время единственная функция трассировки создает трассировку линией, исходящую из компонента. Это будет отлично работать для телепортации; однако, чтобы сделать еще один шаг вперед и позволить игроку телепортироваться как вертикально, так и горизонтально, вы реализуете метод параболической трассировки.

В этом случае параболический след основан на идее движения снаряда и может рассматриваться как бросок мяча из компонента взаимодействия трассировки и следование по его пути, пока он не попадет

в объект в игре. Чтобы выполнить это, вы берете направление компонента взаимодействия в качестве направления, в котором был запущен мяч, и вычисляете новую позицию на фиксированном временном шаге, отслеживая между предыдущим местоположением и этой новой позицией, пока трассировка не коснется объекта, где она остановится и вернет объект, который был поражен.

Для расчета этой дуги снаряда используются физические формулы, которые изображены на рис. 5.1.

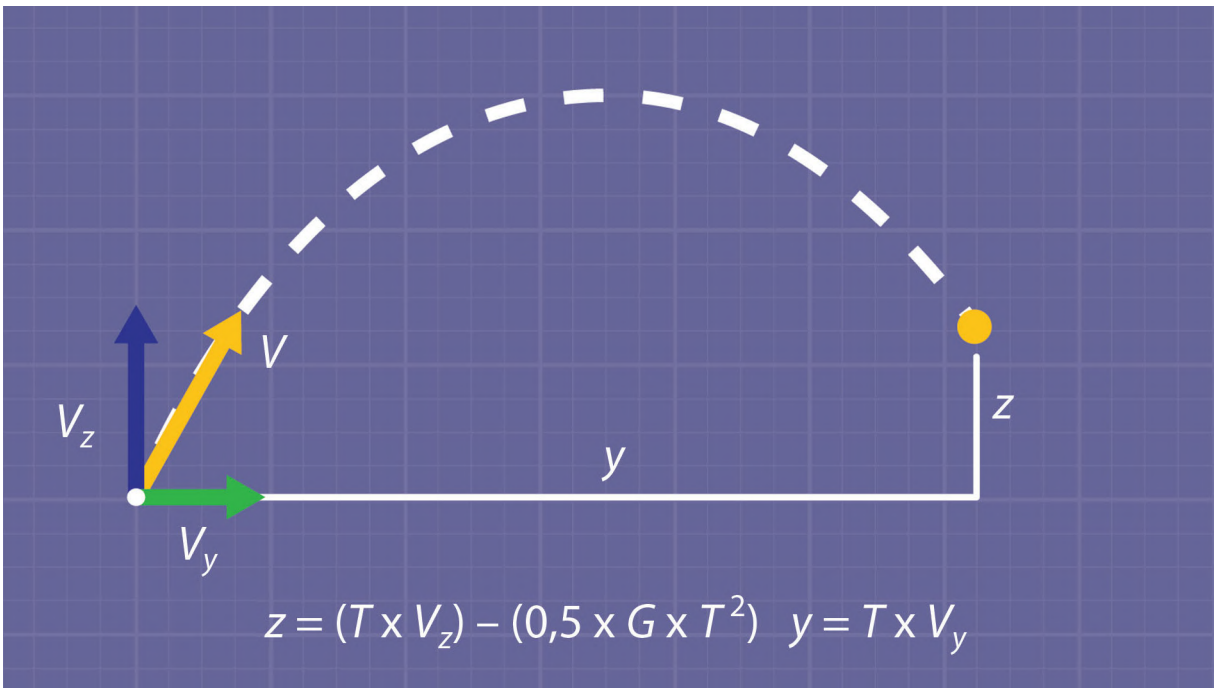

Рис. 5.1. Расчет движения снаряда, используемый для параболических трасс. *V* представляет скорость движения по трассе, *y* представляет расположение по оси *Y*, и *Z* представляет собой расположение по оси *Z* по отношению к времени *T* и тяжести *G*

Создадим функцию, реализующую параболическую трассировку.

1. Откройте ваш проект из главы 4 и в нем откройте *Blueprint TraceInteractionComponent*.
2. Создайте функцию `ParabolicTrace`.
3. Добавьте три входные переменные: `Steps`, `TimeStep`, `Speed` типов `Integer`, `Float`, `Float`.
4. Создайте возвращаемую переменную `Hit` типа `Hit Result`.
5. Создайте три локальные переменные: `InitialLocation`, `PreviosLocation`, `Velocity` типов `Vector`.
6. Создайте еще две переменные `InTimeStep`, `InSteps` типов `Float` и `Integer`.
7. Создайте еще одну переменную `TmpHit` типа `Hit Result`. Она будет хранить временное попадание при вычислении следа.
8. Создайте сеттеры для всех этих переменных, кроме `TmpHit`.
9. Соедините выход `InitialLocation` со входом `PreviousLocation` (рис. 5.2).

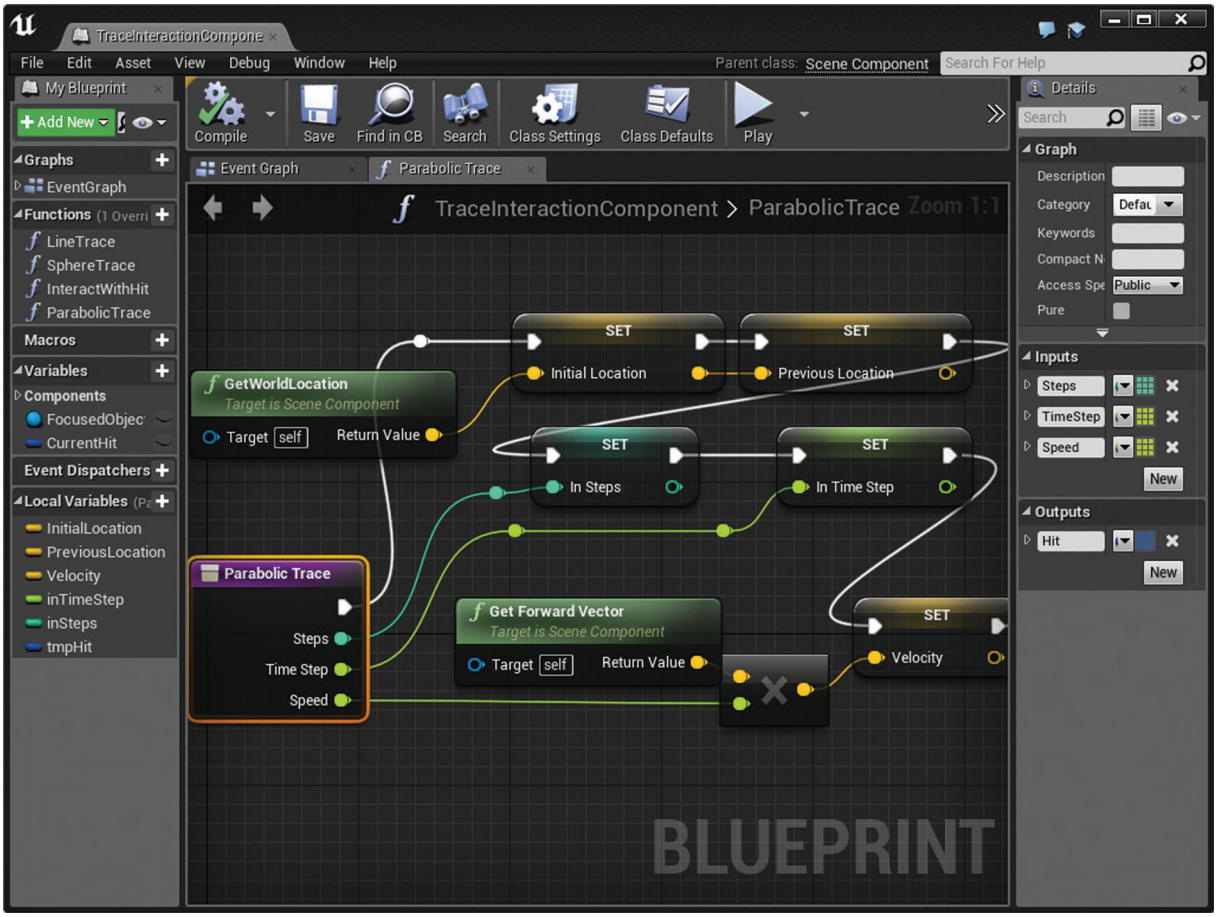

Рис. 5.2. Настройка локальных переменных для параболической трассировки

10. Создайте `GetWorldLocation` и соедините его с `InitialLocation`.
11. Соедините сеттеры `InStep` и `InTimeStep` с входными переменными `Steps` и `TimeStep` (см. рис. 5.2).
12. Соедините `InStep` с вызовом `PreviousLocation`.
13. Создайте `GetForwardVector` и перемножьте его значение с входной переменной `Speed`.
14. Соедините полученное значение с `Velocity` и соедините последний сеттер с входом вызова, как показано на рис. 5.2.

После создания нужных переменных для вычисления попадания вам необходимо найти параболическую дугу, как на рис. 5.1. Для этого вы разделите скорость на оси и рассчитаете новое положение попадания на фиксированном шаге.

1. От последнего сеттера создайте узел *ForLoopWithBreak*.
2. В `FirstIndex` укажите 1, а в `LastIndex` поместите геттер для `InStep`.
3. От *Loop Body* создайте узел *LineTraeByChannel*.
4. В `Start` поставьте геттер `PreviousLocation`.

5. Создайте геттер для `Velocity` и разбейте его на `X`, `Y` и `Z` (рис. 5.3).

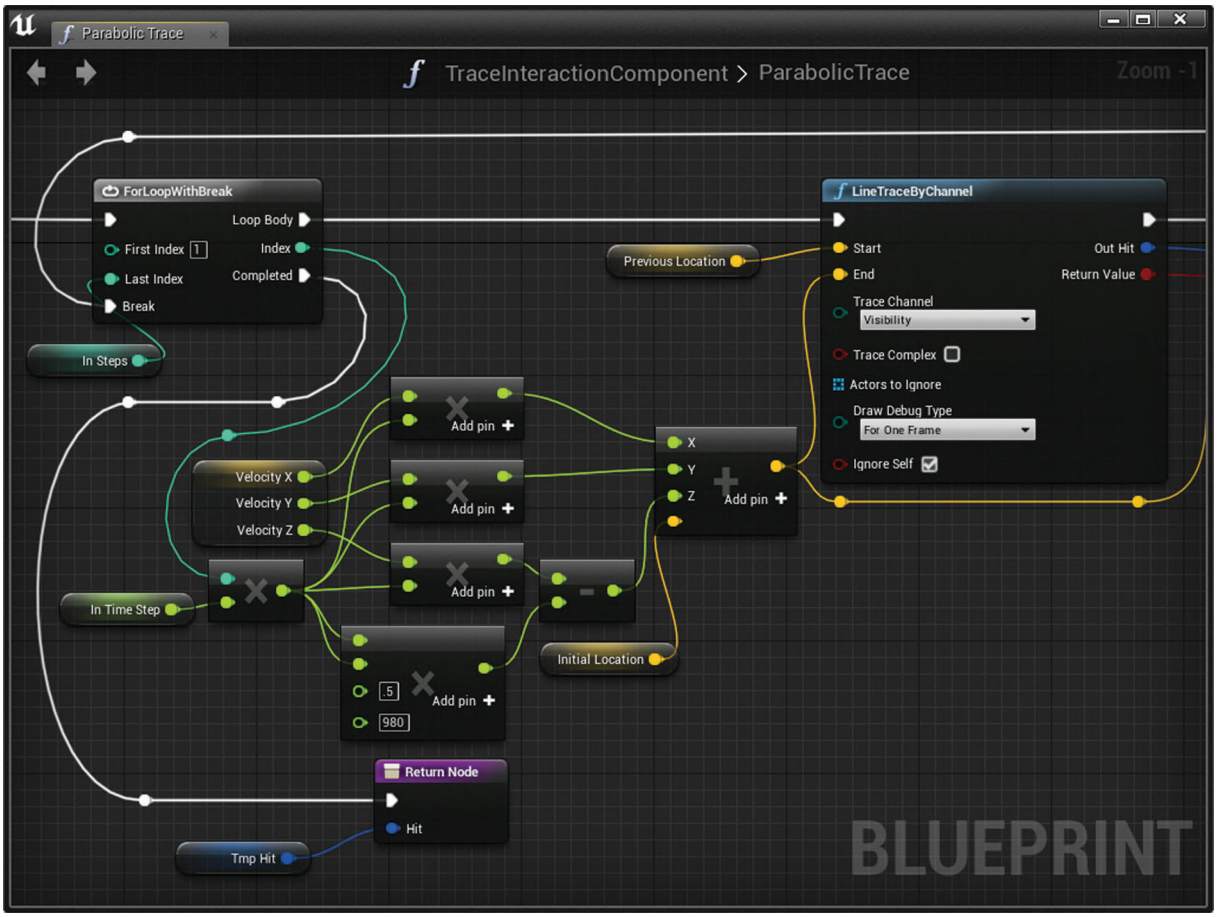

Рис. 5.3. *ParabolicTrace*: расчет позиции окончания отслеживания

6. Для каждой координаты вызовите функцию `Float * Float`.
7. От контакта `Index` создайте `Int * Float`, подключив второй контакт к геттеру для `InTimeStep`. Это позволит конвертировать каждый шаг во время `Float`.
8. Соедините выход последнего узла со вторыми контактами узлов `Float * Float`, созданных на шаге 6. Это посчитает соответствующие `X` и `Y`. Для расчета `Z` нужно учесть гравитацию.
9. От созданного узла на шаге 7 создайте еще один узел `Float * Float`, соединив оба контакта. Фактически это возведение в квадрат.
10. Добавьте еще два входных контакта (нажать правой кнопкой мыши по узлу и выбрать `Add Pin`) и установите в них значения 0.5 и 980. Это значения для расчета формулы движения снаряда.
11. От узла `Float * Float`, созданного от `Velocity Z`, создайте узел `Float – Float` и соедините второй контакт с узлом шага 10 (рис. 5.3).
12. Создайте узел `Vector + Vector`, разбив первый `Vector` на координаты.
13. Установите подсчитанные *X* и *Y* в соответствующие входы.

14. Установите `Float – Float` шага 11 в `Z`.
15. Добавьте новый геттер переменной `InitialLocation` во второй вектор суммы. Это вернет подсчитанный вектор в мировых координатах.
16. Соедините выход этого узла с контактом `End`.
17. Если вы хотите видеть след отслеживания, установите `For One Frame` в `Draw Debug Type`.
18. Присоедините узел `Return` к `Completed` узла *ForLoopWithBreak* и `TmpHit` в возвращаемом значении. Не беспокойтесь, позже мы установим эти переменные.

После расчета трассировки, вам необходимо настроить переменную `TmpHit` и прервать цикл раньше, если обнаружено столкновение, чтобы убедиться, что не сделано лишних вычислений.

1. От узла `LineTraceByChannel` создайте сеттер `PreviousLocation` и установите в нем выход узла `Vector+Vector` (рис. 5.4).

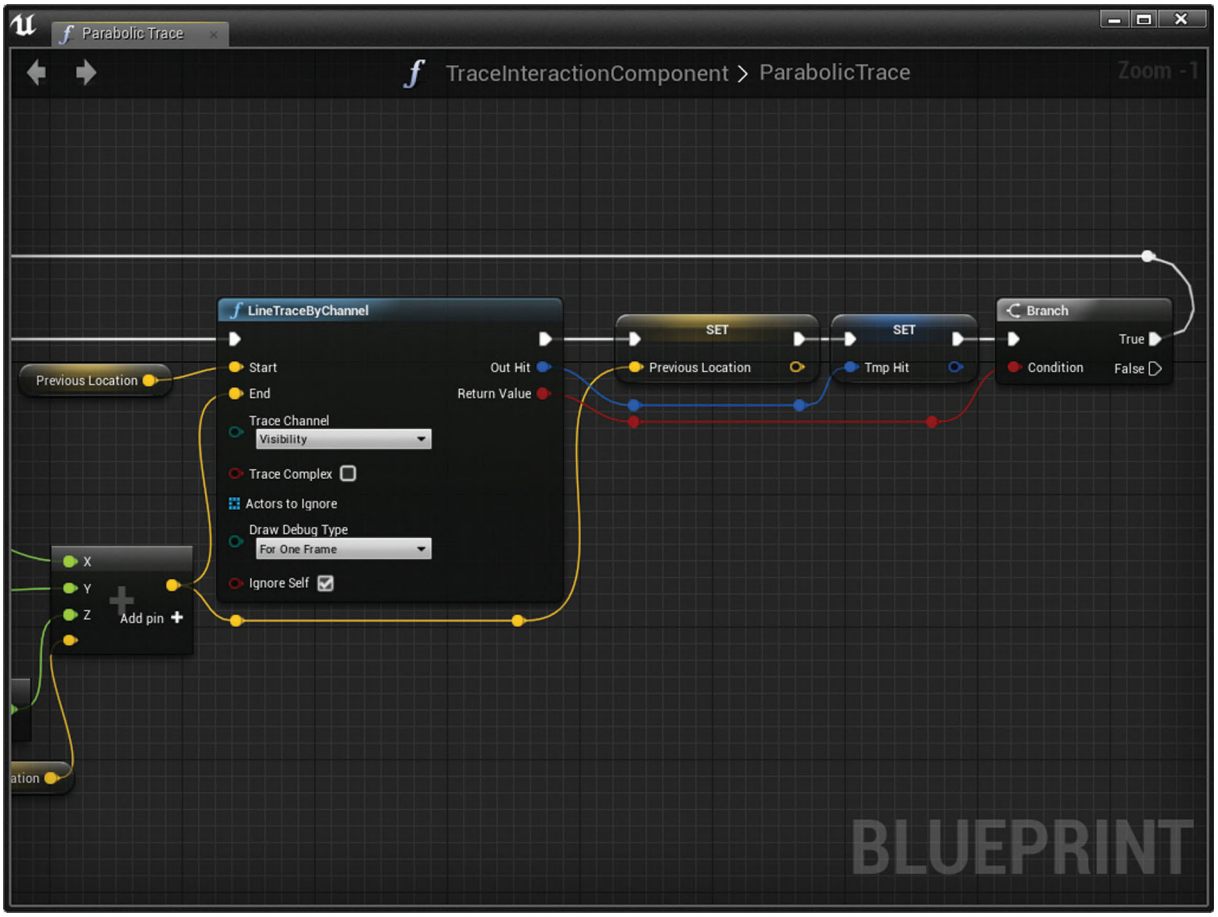

Рис. 5.4. *ParabolicTrace*: установка переменных и прерывание цикла

2. Создайте сеттер переменной `TmpHit` и соедините `Out Hit` с ним.
3. Создайте *Branch* и соедините `Return Value` с `Condition`.

4. Соедините ветку `True` с контактом `Break` в цикле, чтобы прервать цикл сразу после того, как произошло столкновение.

Теперь функция отслеживания завешена, пришло время вставить ее в *Event Graph*.

1. Вернитесь в *Event Graph*, удалите функцию `LineTrace` и замените ее на `ParabolicTrace`.
2. Во входных значениях укажите `Steps 10`, `Time Step 0.1` и `Speed 500` (рис. 5.5).

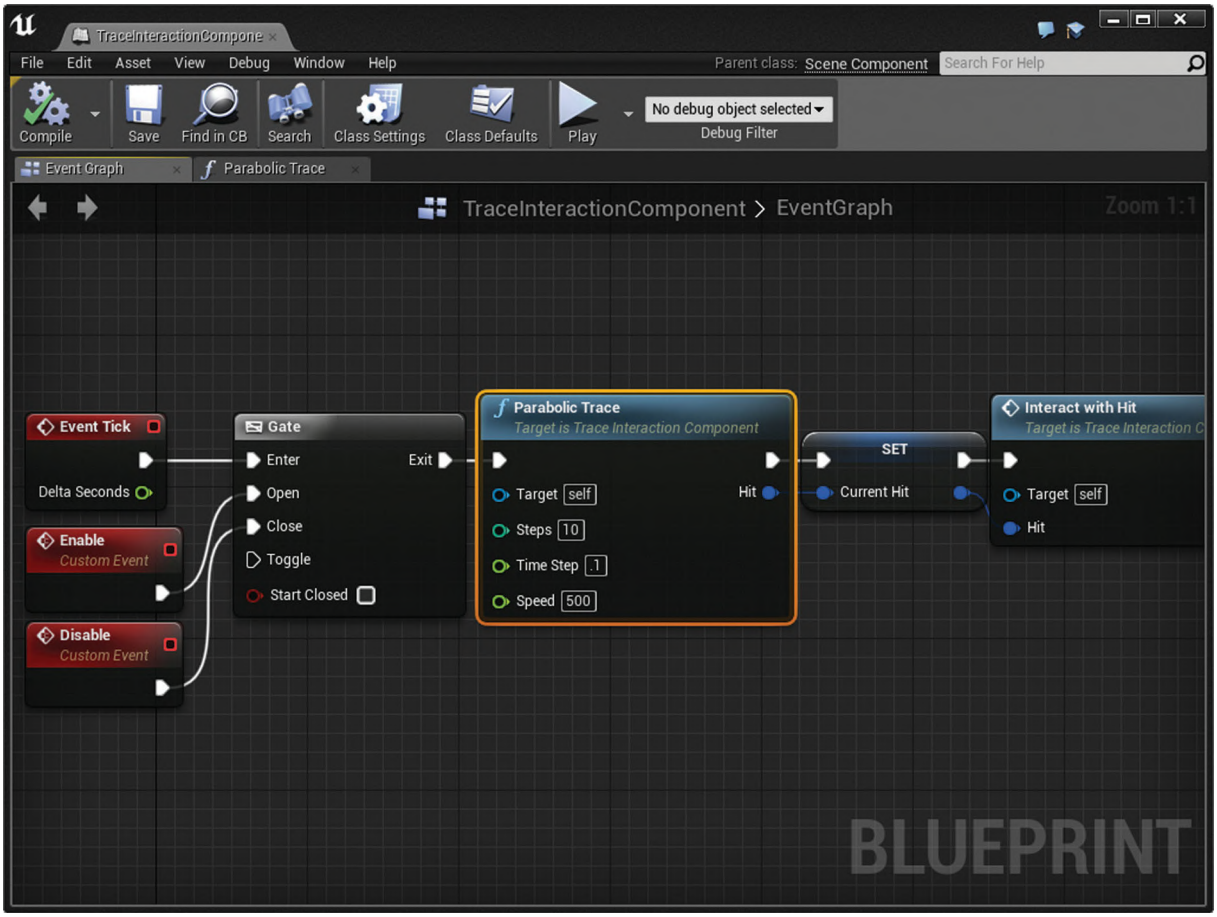

Рис. 5.5. Параболическая трассировка: использование *ParabolicTrace* в компоненте взаимодействия

5.2. Визуализация телепорта

При тестировании телепорта вы можете использовать простую отладку, чтобы видеть, куда будет телепортирован игрок. Однако для этого вы создадите отдельный *Actor*, потому что тогда сможете расширить его позже для визуализации большего, чем просто следа.

5.2.1. Визуализация материала

Сначала создадим материал, который будет использован для визуализации. Это простой градиент, затихающий наверху и внизу.

1. Создайте папку *Materials* в *Content Browser*, если ее нет.
2. Создайте в ней новый материал и назовите его `TeleportVizualizer`.
3. Откройте материал, установите на панели *Details Translucent* в *Blend Mode* и *Unlit* в *Shading Model* и включите *Two Sided*.
4. Создайте узел *VectorParameter* (можно нажать *V* и ЛКМ), назвав его `Color`, и установите значения `(R = 0.25, G = 1.0, B = 0.5, A = 1.0)`. Соедините его с контактом `Emissive Color`.
5. Чтобы создать градиент, создайте *TextureCoordinate* (можно нажать *U* и щелкнуть мышью).
6. От этого узла создайте узел *ComponentMask*, выбрав только *G*-область. Это извлечет только `Y` градиент из координат.
7. Чтобы получить градиент на обоих концах, от *Mask* создайте узел *Cosine*. Это возьмет градиент от 0 до 1, чтобы сделать его от 1 до –1 до 1.
8. Чтобы преобразовать его, добавьте *OneMinus* после *Cosine*.
9. Чтобы нормировать это значение, создайте *Divided* по 2 (рис. 5.6).

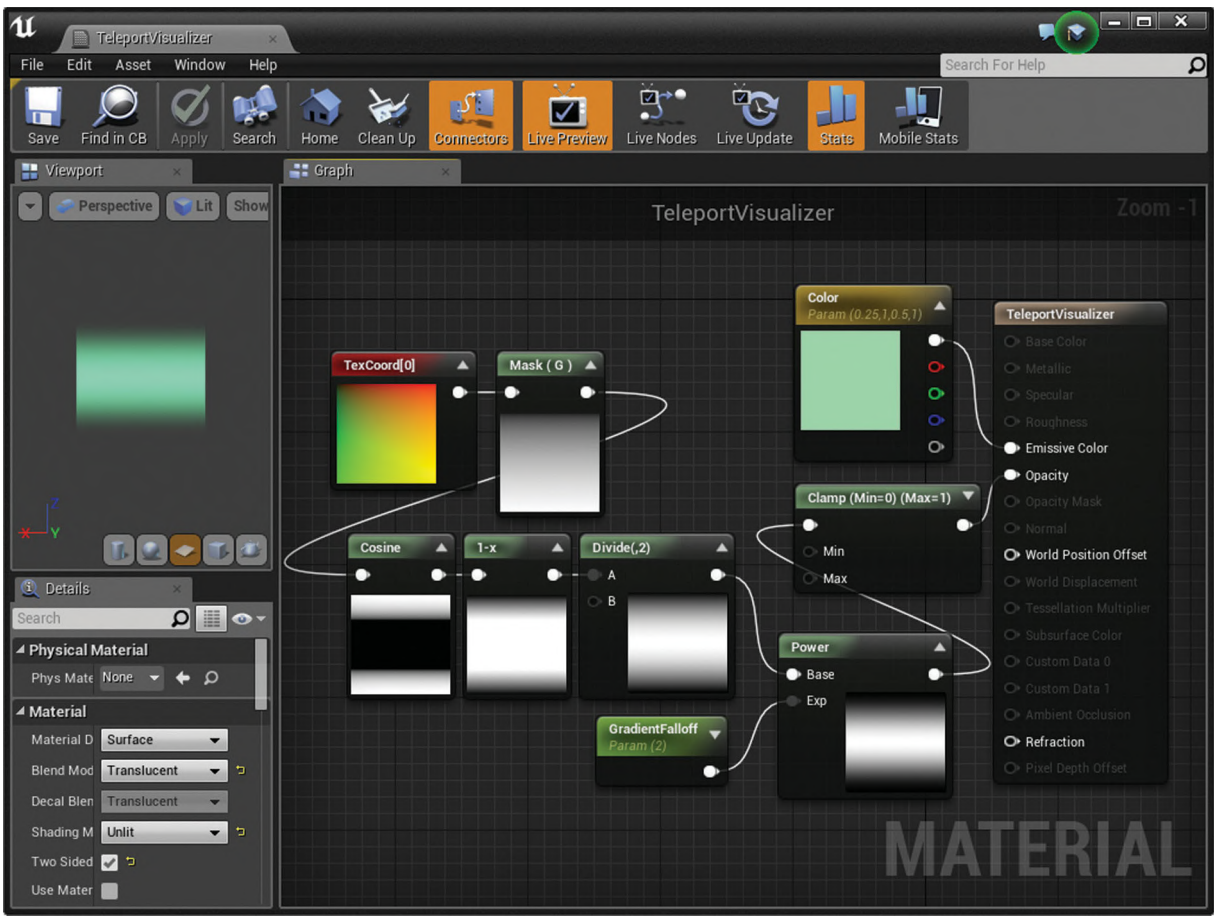

Рис. 5.6. Материал визуализации телепортации

10. Добавьте узел *Power*, установив во второй контакт новый скалярный параметр `GradientFallOff` со значением по умолчанию `2`.
11. Добавьте *Clamp* узел после *Power*.
12. Соедините выход `Clamp` с входом `Opacity` материала.

5.2.2. Визуализация актора

Теперь создадим актора (*Actor*), который будет содержать сетку (*mesh*) визуализации.

1. В папке *Blueprint* создайте новый *Blueprint Actor*, назвав его `TeleportVizualizer`.
2. Откройте его и добавьте новый компонент *Static Mesh*, назвав его `Visual`.
3. Установите у компонента на панели *Details* в свойстве `Static Mech S_EV_SimpleLightBeam_02`. Если вы его не видите, активируйте `Show Engine Content` в параметре `View`.
4. Установите в `Materials Element 0` ваш созданный материал.
5. Поверните компонент на `180` градусов по оси `X` и укажите `scale` на `0.3` по оси `Z`, чтобы сделать его похожим на телепорт (рис. 5.7).

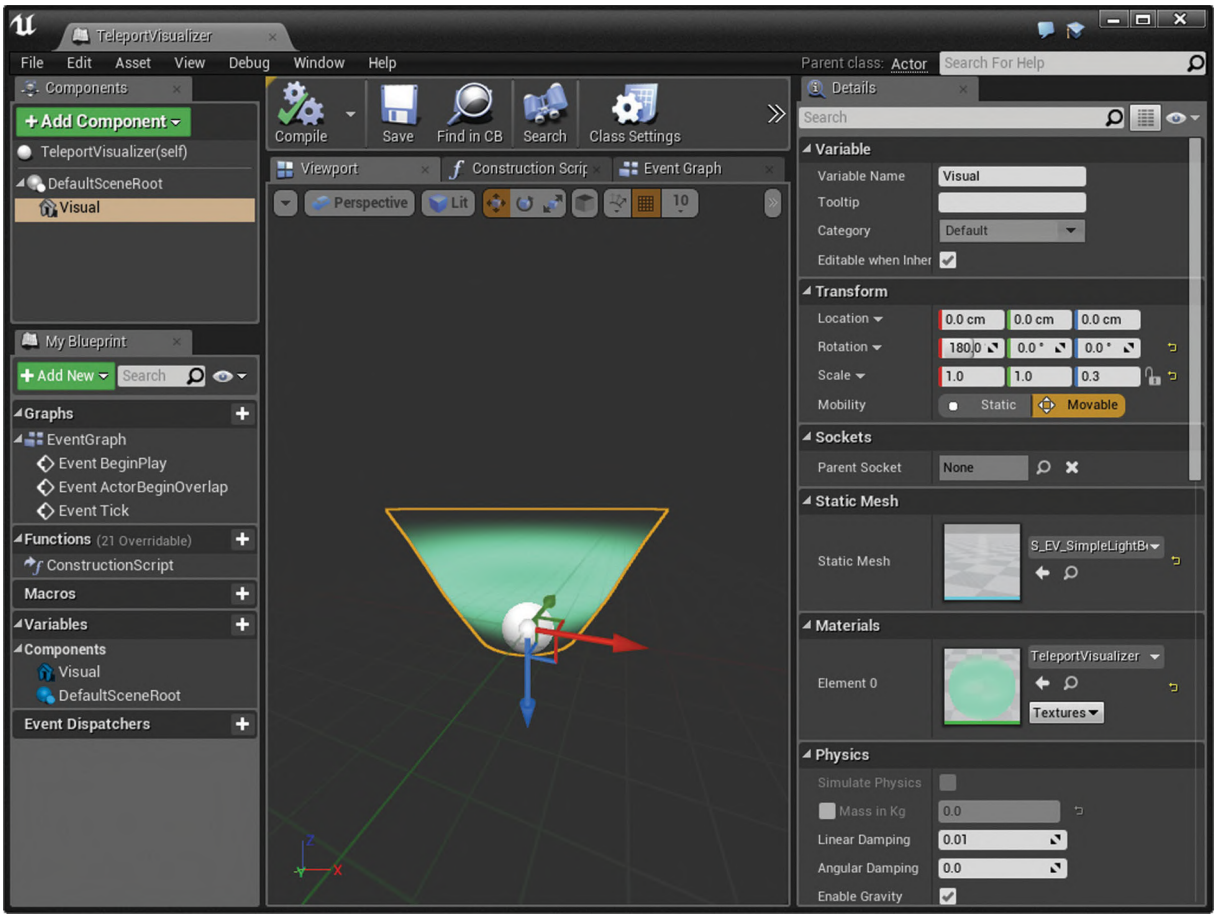

Рис. 5.7. *Actor* визуализации телепортации

5.3. Простой регулятор телепортации

Теперь у нас есть визуализация телепорта. Вы можете создать регулятор телепортации, чтобы пользователь мог выбирать куда телепортироваться.

1. В папке *Blueprint* создайте новый *Actor* под названием `TeleportVolume`.
2. Откройте его и добавьте два компонента *Box Collision* со стандартным названием и *Child Actor*, назвав его `TeleportVizualizer`.
3. Выберите `Box` и установите в значении `Box Extent (X=200, Y=200, Z=1)` (рис. 5.8).

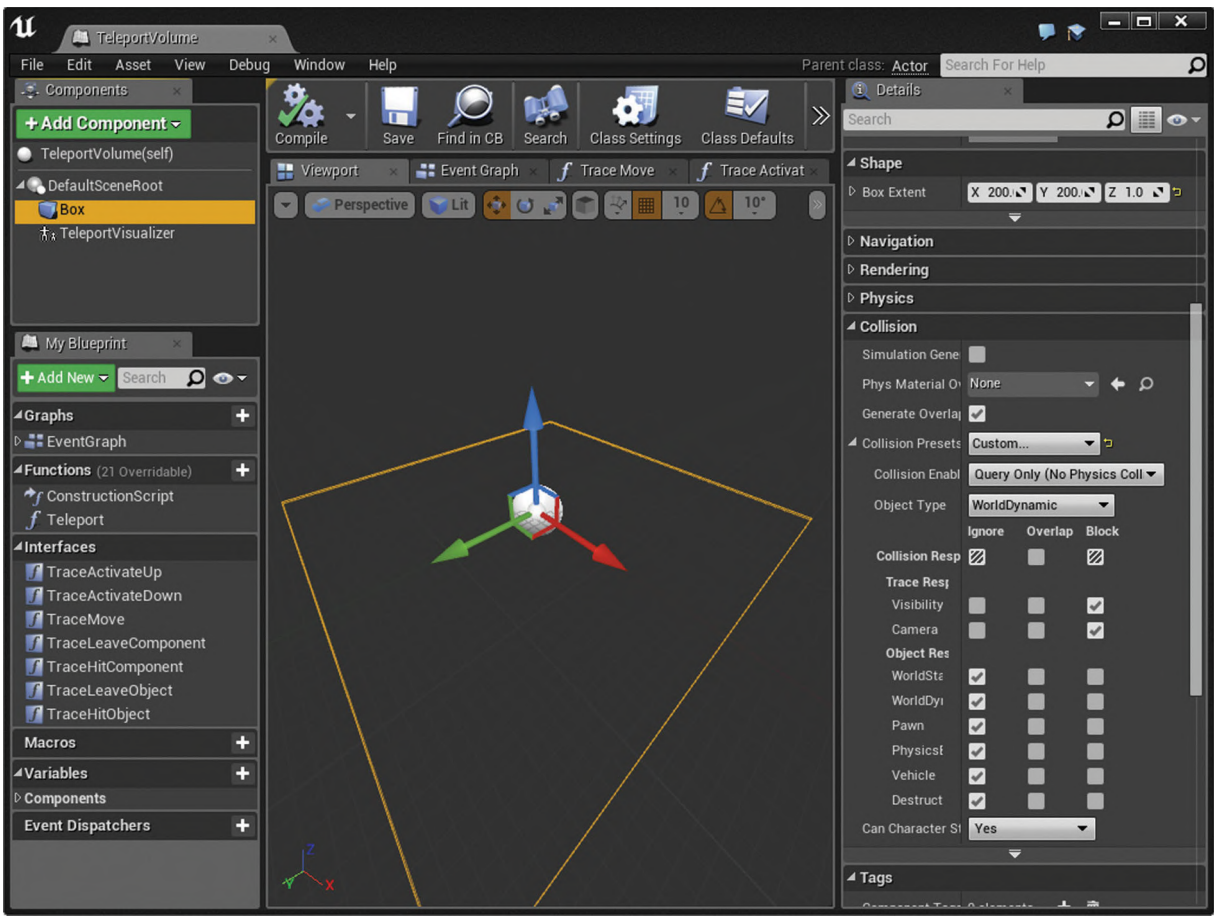

Рис. 5.8. Регулятор телепортации: компоненты

4. Установите `Collision Present` на значение *Custom*, убедившись, что в `Collision Enabled` стоит `Query Only`, и выберите `Ignore` для всех реакций на столкновение, кроме `Visibility` и `Camera`. Для них установите `Block`.
5. Выберите *Child Actor* и установите `Child Actor Class` на `TeleportVizualizer`.
6. Уберите флаги `Visible`.

7. Откройте *Class Settings* и добавьте *TraceInteractionInterface* в секцию *Implemented Interface* (не забудьте скомпилировать, чтобы получить доступ к функциям интерфейса).
8. Откройте функцию `TraceMove` и вызовите узел *Break Hit Result* от входной переменной `Hit`.
9. Создайте новый геттер для компонента `TeleportVilualizer` и создайте `SetWorldLocation`.
10. Соедините последний между входом и выходом из функции.
11. Соедините узел *Location* узла *Break* с `New Location` узла *SetWorldLocation* (рис. 5.9). Это будет двигать телепорт каждую обработку кадра в место, где игрок взаимодействует с регулятором.

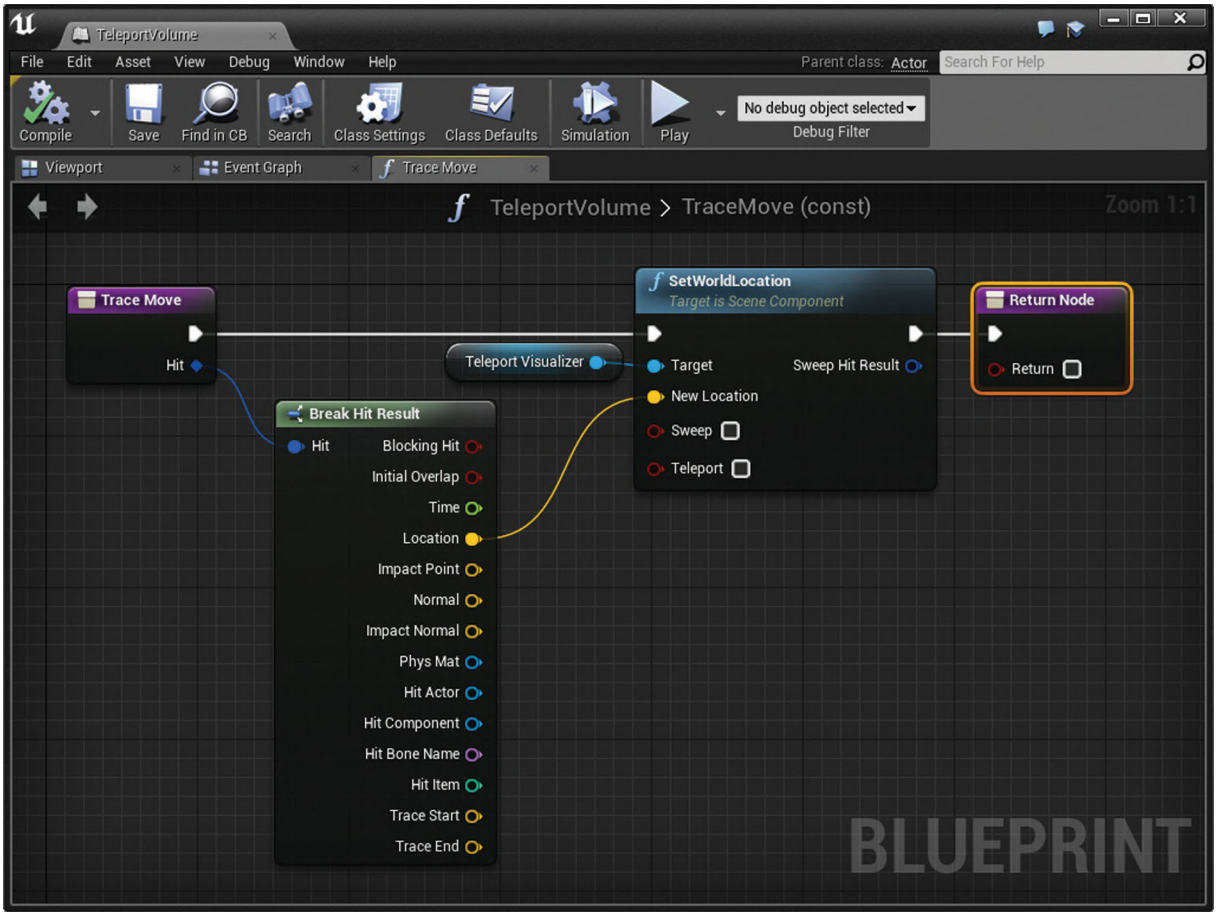

Рис. 5.9. Регулятор телепортации: функция *TraceMove*

12. Откройте функции `TraceHitObject` и `TraceLeaveObject` и добавьте новый геттер для компонента `TeleportVizualizer`.
13. От него вызовите `SetVisibility`. В функции `TraceHitObject` отметьте оба флажка, а в `TraceLeaveObject` только последний. Затем соедините их с выходом из функции (рис. 5.10). Это будет показывать и скрывать визуализацию при взаимодействии пользователя с регулятором.

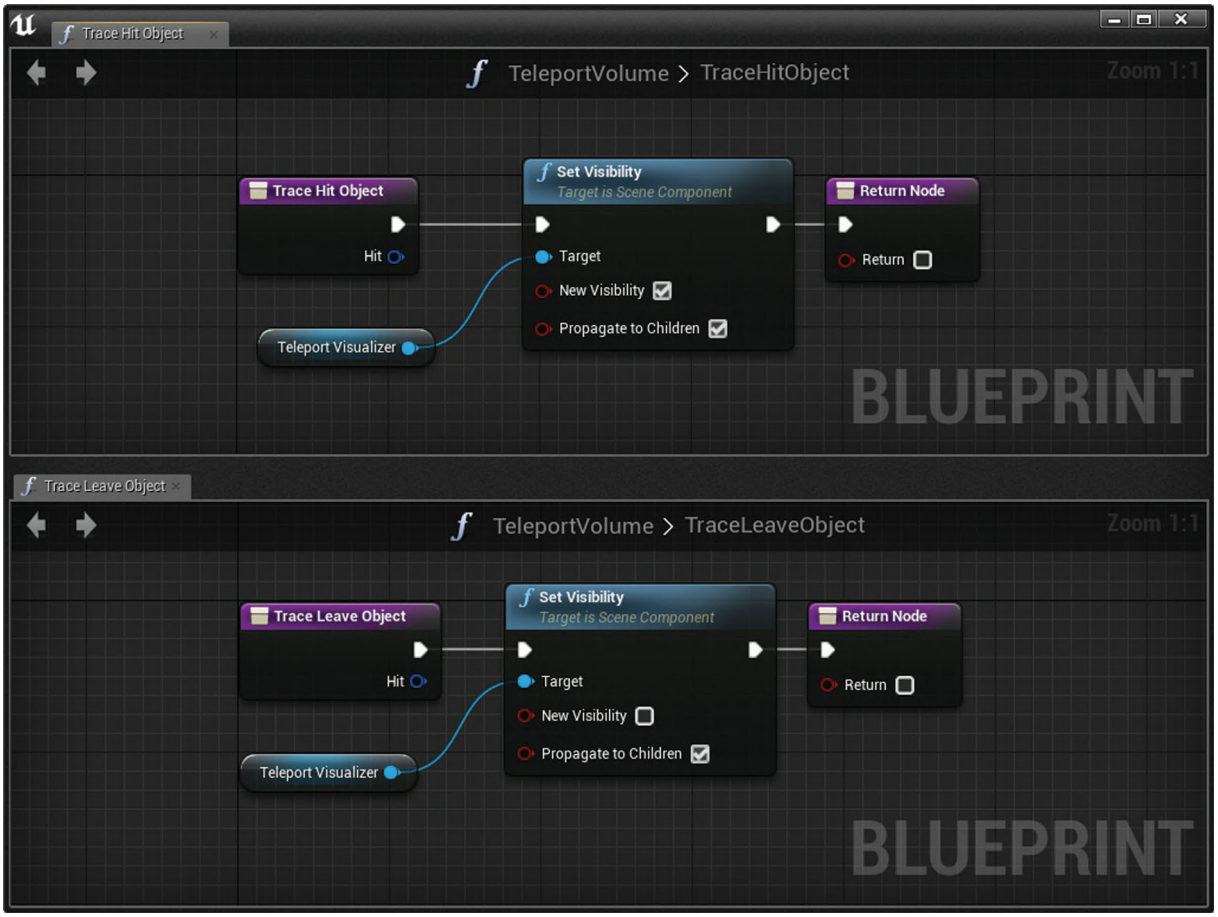

Рис. 5.10. Регулятор телепортации: попадание и выход

14. Создайте новую функцию под названием `TeleportPlayer`, которая принимает *Pawn* под названием `Players` и *Vector* под названием `Location`. Она телепортирует игрока, принимая во внимание позицию *HMD*.
15. Откройте функцию и от `Players` создайте узел *GetController*, прикастуйте его к *PlayerController*, соединив контакт выполнения с функцией `Cast`.
16. От `As Player Controller` получите *PlayerCameraManager*.
17. От этого узла вызовите функцию `GetCameraLocation` (рис. 5.11).

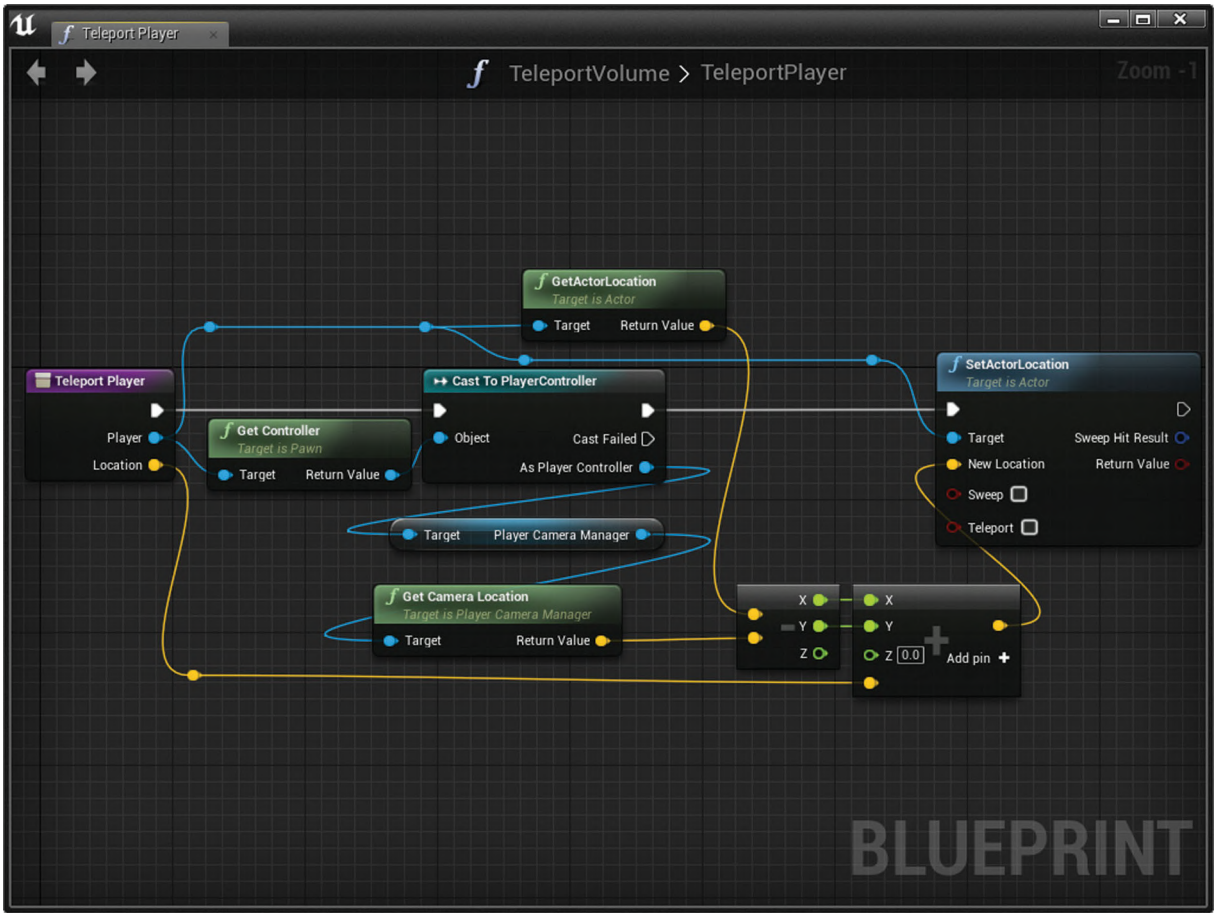

Рис. 5.11. Регулятор телепортации: функция телепорта

18. От входного значения *Player* создайте еще один узел *GetActorLocation*.
19. От него создайте узел `Vector - Vector`, присоединив во второй вход *GetCameraLocation*.
20. От разности создайте узел `Vector + Vector` и разбейте первый вход.
21. Разбейте выход узла `Actor - Camera` и оставьте только `X` и `Y` контакты с шага 18.
22. Во второй вход узла `Vector + Vector` присоедините входную переменную `Location`.
23. От выхода последнего узла вызовите `SetActorLocation`. В *Target* укажите входное значение *Player* (рис. 5.11).
24. Откройте функцию `TraceActivateUp` и вызовите узел *BreakHitResult* от `Hit`.
25. Вызовите функцию *TeleportPlayer* и соедините его между входом и выходом из функции.
26. Соедините *Instigator* с контактом *Player* узла *TeleportPlayer*.
27. Разбейте вход `Location` узла *TeleportPlayer* и выход `Location` узла *BreakHitResult*.
28. Соедините `X` и `Y` между ними.
29. Создайте *GetActorLocation*, разбейте его выход.

30. Соедините `Z` с входным контактом `Z` узла *TeleportPlayer* (рис. 5.12). Это телепортирует игрока по координатам `X` и `Y`, но по `Z` *Actor*-а, когда вызывается `ActivateUp`.

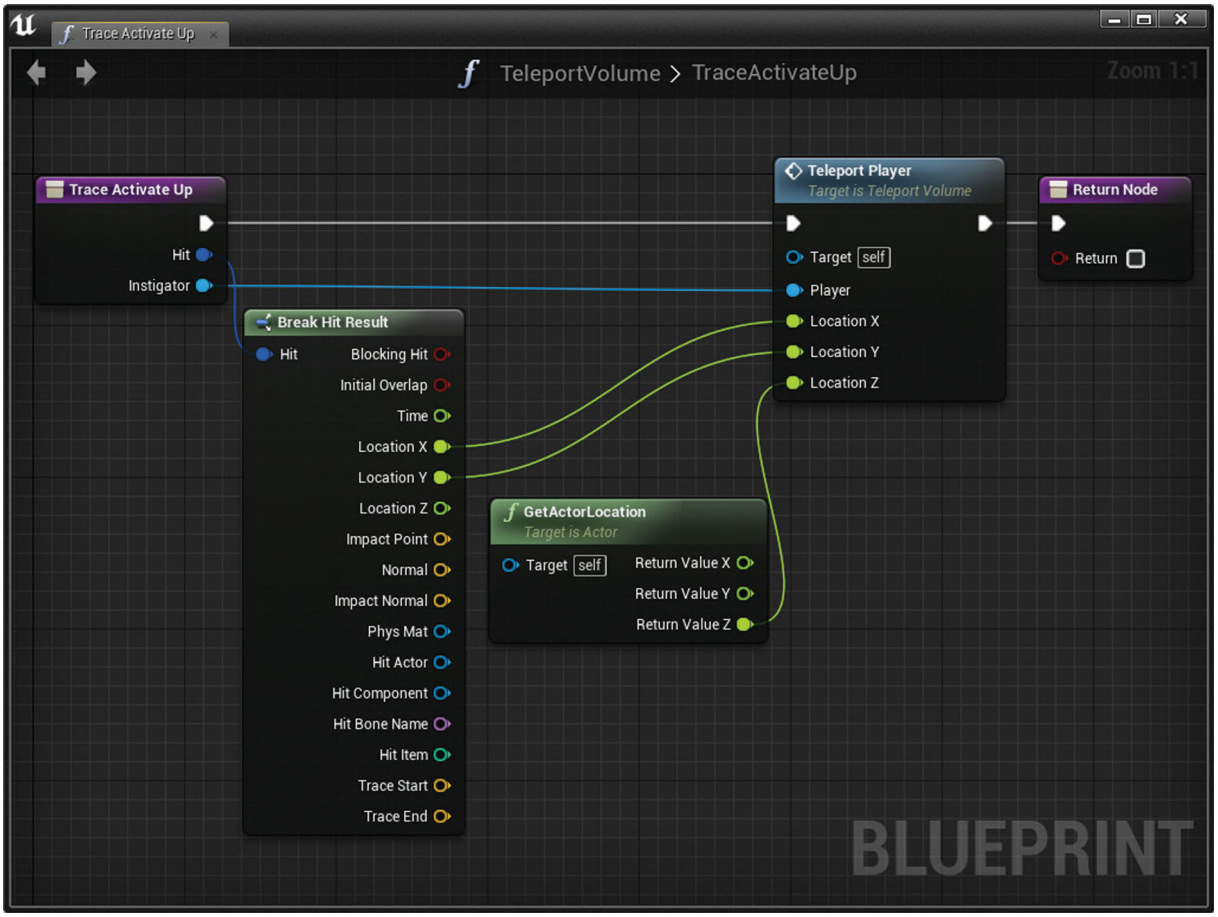

Рис. 5.12. Регулятор телепортации: активация

Если вы используете проект из главы 4, добавьте только новый `TeleportVolume` на ваш уровень и проверьте его. Иначе вам необходимо добавить *Game Mode* вместе с вашими *Pawn* и *InteractionComponent*.

5.4. Заключение

Эта глава показала вам, как усовершенствовать взаимодействие, разработанное в главе 4, и добавить параболические трассы.

Вы также реализовали базовый телепорт, который позволит любому пользователю с *Trace Interaction Component* телепортироваться к нему.

5.5. Упражнения

Попробуйте придумать, как расширить эту систему. Варианты могут включать:

1) добавление системы частиц, которая будет показывать, откуда пришел пользователь;
2) создание нового телепорта, действующего как единая точка, к которой может телепортироваться игрок;
3) замедление отслеживания на пару секунд;
4) создание компонента, который визуализирует вашу параболическую трассу.

ГЛАВА 6

ГРАФИКА ДВИЖЕНИЯ И 2*D*-ИНТЕРФЕЙСЫ С ПОЛЬЗОВАТЕЛЕМ В *UNREAL*

Несмотря на лидирующие позиции 3*D*-интерфейсов, 2*D* по-прежнему является удобным способом представлять в *VR* относительно большие объемы информации, реализуя известные шаблоны взаимодействия. В этой главе рассматриваются некоторые проблемы с 2*D*-интерфейсами в *VR*, предлагается реализовать систему взаимодействия *Unreal Motion Graphics (UMG)* с нуля, и разъясняется, как интегрировать эту систему в других системах взаимодействия.

6.1. Проблемы 2*D*-интерфейса с пользователем в *VR*

Создание интерфейса с пользователем в *VR* сопряжено с серьезными трудностями, которые обычно не возникают в обычных 2*D*-играх. В традиционных играх многие пользовательские интерфейсы «привязаны» к голове (прикреплены к виртуальной голове игрока, то есть если голова перемещается, пользовательский интерфейс следует за ней). Такое поведение вызывает основные проблемы при портировании на существующие *VR*-системы.

Две основные проблемы возникают при попытке реализовать в *VR* описанный выше интерфейс пользователя с привязкой к голове. Обычно для обеспечения четкости такой интерфейс проецируется ортогонально. Это означает, что пользовательский интерфейс может быть отрисован поверх всех других элементов в игре, и ему не нужно взаимодействовать с игровым 3*D*-миром.

Однако если вы спроецируете пользовательский интерфейс ортогонально в *VR*, глаза игрока увидят его так, как если бы он был бесконечно далеко. Как вы легко догадаетесь, это плохая идея, потому что приводит к несоответствию между зрительным восприятием и положением объектов, которые не находятся в бесконечности (потому что мозг недоумевает, почему самый дальний объект отображается поверх объектов, которые предположительно расположены ближе). Это может вызвать дискомфорт.

Еще одна причина, по которой фиксированные к голове интерфейсы с пользователем обычно являются плохим решением для *VR*, заключается в том, что линзы, которые в настоящее время необходимы в *VR*-шлемах, чтобы позволить игрокам сосредоточиться на экране, очень близко расположены к глазам, не идеальны и имеют ограничение, приводящее к тому, что самое четкое изображение находится по центру взгляда игрока. Как следствие, при переходе на периферию изображение становится менее четким. Таким образом, расположение элементов интерфейса с пользователем по краям его поля зрения делает их менее читаемыми, чем в центре. Еще одна проблема заключается в удалении выходного зрачка (расстояние от глаз игрока до экрана *VR*-шлема) игроков, и то, что на периферии для некоторых, может быть полностью скрыто для других!

Как поступить для решения этих проблем? Простое решение — это создать трехмерный объект (обычно четырехугольник) и поместить его в игровом мире, после чего на нем сгенерировать интерфейс. Это решает проблему расстояния с ортогональной проекцией, поскольку применяются все перспективные проекции игрового мира, и проблему периферии с заблокированным пользовательским интерфейсом, потому что игрок сможет при необходимости смотреть на любую часть пользовательского интерфейса. Именно так выглядит решение, которое выбирает *Unreal*.

Однако это решение не лишено проблем, поэтому в этой главе рассматриваются возможные решения некоторых из них.

Еще одна проблема с использованием традиционных пользовательских интерфейсов в *VR* возникает, когда вы задаетесь вопросом, а действительно ли вам нужен пользовательский интерфейс. Такой вопрос может показаться глупым, но из-за повышенного уровня погружения и присущей среде стереоскопичности большинство традиционных двумерных интерфейсов оставляют желать лучшего. Часто менее традиционный пользовательский интерфейс порождает более захватывающий сюжет. Например, в игре-шутере реалистичнее смотреть на виртуальное оружие, чтобы увидеть, сколько осталось патронов, а не просто показывать их количество сбоку. Конечно, все это зависит от стиля и типа игры, которую вы создаете; в некоторых случаях объемный текст может действительно соответствовать стилю вашего *VR*-мира.

6.2. История и совместимость *UMG*

Unreal Motion Graphics (*UMG*) — это инструмент, построенный по принципу *WYSIWYG* (то, что вы видите, это то, что вы получаете), для создания пользовательских интерфейсов в *Unreal Engine*. Он позволяет иерархически определять меню, пользовательский интерфейс и *HUD* (*heads-up displays*), размещая виджеты внутри графического редактора и определяя их внешний вид и функциональность.

Интегрированный в движок, начиная с версии 4.4, *UMG* раньше нужен был для создания двумерных пользовательских интерфейсов, но послужит и нашей цели. Первоначально *UMG* страдал от только что описанных проблем; однако начиная с версии 4.6 в движок был добавлен экспериментальный 3*D UMG* компонент. Этот компонент позволяет разместить любой виджет *UMG* в 3*D*-пространстве и настроить основные свойства, такие как размер и цвет фона. Однако и здесь не обошлось без проблем. Главная проблема заключалась в том, что не было возможности взаимодействовать с этими виджетами после рендеринга *VR*-мира. Чтобы избежать этого, многие разработчики использовали приемы, такие как создание пользовательских виджетов и обнаружение пользовательских попаданий, выполняя преобразование координат из координат игрового мира в относительные координаты *UMG* вручную или, как я, добавляя функциональность в виде плагина, который позволял разработчикам взаимодействовать с 3*D*-виджетами вручную при помощи трассировки. (Глава 4 «Взаимодействие на основе трассировки».)

Однако начиная с версии движка 4.13 эти проблемы решены добавлением компонента *UMG Interaction*, который мы будем использовать в этой главе.

6.3. Простое *VR*-меню

Чтобы создать простое меню, нам понадобятся три *Actors*.

1. Традиционный *UMG*-виджет, который хранит актуальные *UI*-виджеты, определяет их внешний вид и положение.
2. *Actor*, который хранит 3*D Widget Component* и показывает *UMG*-виджет в 3*D* пространстве.
3. *Pawn* игрока, который хранит в себе *Camera* и *Widget Interaction Component*, позволяющий ему взаимодействовать с вашими виджетами в 3*D*-пространстве.

Сначала создадим *UMG*-виджет, который будет похож на обычное меню в *VR*-играх. Это позволит игроку начать игру, выбрать комфортный режим и изменить сложность игры, а также выйти из игры. Заметим, что эти пункты не будут рабочими, так как реализация зависит от вашей игры.

1. Создайте новый пустой *Blueprint* проект.
2. Создайте в корневой папке две папки: *Blueprints* и *UMG*. В первой будут храниться *Blueprints*, во второй — *UMG*-виджеты.
3. Создайте новый *Widget Blueprint* в папке *UMG* (в меню выберите *Add new* ⇒ *User Interface* ⇒ *Widget Blueprint*). Назовите его `MenuWidget`.
4. Откройте его и удалите уже созданную `Canvas Panel`. Замените ее на `Vertical Box`. Это простой способ разместить внутренние виджеты.
5. Установите для свойства `Fill Screen` (*В верхнем правом углу* `Viewport`) `Custom` с высотой и шириной по 200.
6. Добавьте новый `Button`, перетащив его на `Vertical Box`. Назовите его `StartButton` и установите значения для `Padding` (`Left=10`, `Top=10`, `Right=10`, `Bottom=5`). Создайте новый `Text`, привязав его к `StartButton`. Измените отображаемый текст на `Start`.

7. Создайте новый `Check Box` под `StartButton` и назовите его `ComfortModeCheckBox`, настроив его состояние `Checked State` на `Checked` и `Padding` на `(Left = 10, Top = 5, Right = 10, Bottom = 5)`.
8. Создайте новый `Text`, привязав его к `Check Box` и установите текст `Comfort Mode` и `Font Size` на `12`.
9. Создайте `ComboBox` и поместите его под `ComfortModeCheckBox`, добавив три новые строки в `Default Options` (`Easy`, `Normal`, `Hard`) и установите `Normal` в `Selected Option` (рис. 6.1).

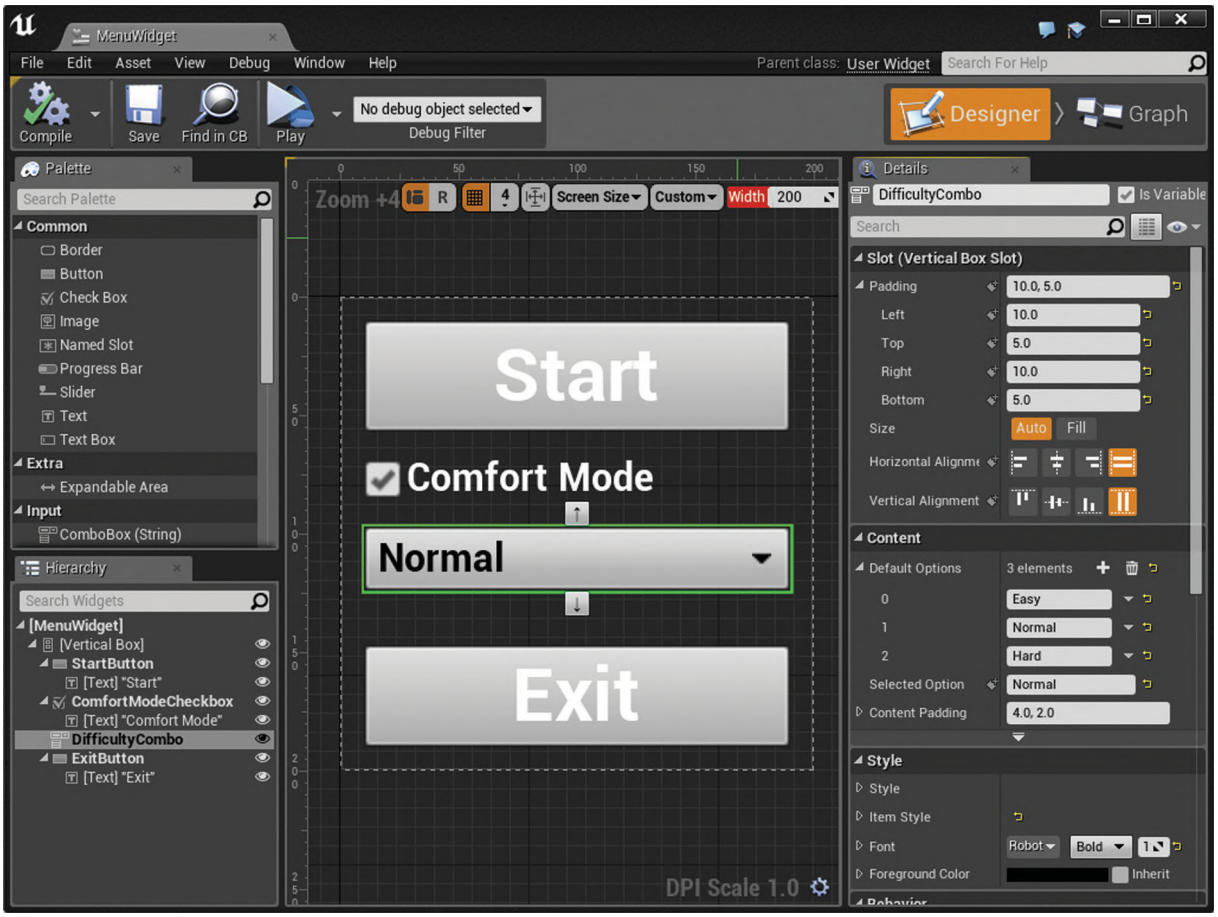

Рис. 6.1. Простое меню с кнопками «старт» и «выход», а также вариантами режима и сложности

10. Измените `Font Size для Difficulty Combo` на `12` и установите `Padding (Left = 10, Top = 5, Right = 10, Bottom = 10)`.
11. Скопируйте кнопку *Start*.
12. Вставьте ее, выбрав *Vertical Box*.
13. Выберите этот виджет и назовите его `ExitButton`, затем установите его `Vertical Alignment` на `Bottom`, `Size` на `Fill` и `Padding (Left = 10, Top = 5, Right = 10, Left = 5)`.
14. Выберите текст внутри кнопки и измените его на `Exit`.

6.3.1. Меню *Actor*

После создания 2*D*-меню, вам нужно создать трехмерный *Actor*, который будет размещен в мире и хранить 3*D*-представление виджета.

1. Создайте новый *Actor* в папке *Blueprints* и назовите его *MenuActor*.
2. Добавьте компонент *Widget*.
3. Выбрав этот компонент, на панели *Details* под *User Interface*, установите *MenuWidget* в *Widget Class*.
4. В этой же панели, установите `DrawSize (X=200, Y=200)`. Эти параметры вы использовали, когда создавали виджет, но это необязательно так. *UMG*-виджет может масштабироваться до произвольного размера.
5. Так как *Draw Size* все еще большой в мире, установим `Scale` у компонента `(X = 0.5, Y = 0.5, Z = 0.5)`. Это имеет хороший побочный эффект, делая текст лучше, так как виджет будет отрисован в высоком разрешении, а затем уменьшен.
6. На панели *Details* установите `Background Color` на `(R = 0.0, G = 0.0, B = 0.0, A = 0.2)` и `Blend Mode` на `Transparent`. Это снижает производительность, но прозрачный черный фон поможет ему слиться с местностью (рис. 6.2).

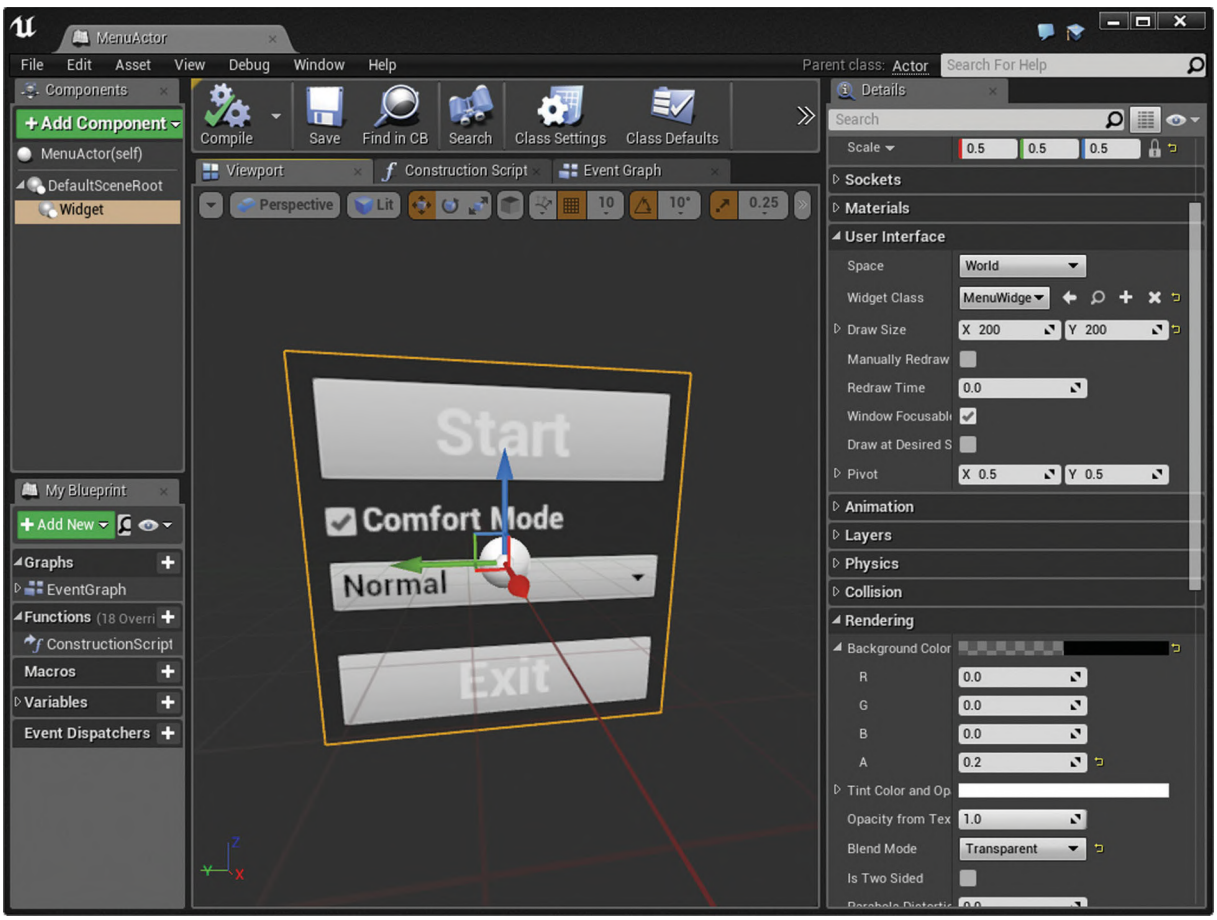

Рис. 6.2. *VR*-меню *Actor* для создания 3*D*-виджета

7. Чтобы отвечать на взаимодействия пользователя с виджетом, вам необходим доступ к `Widget Object`. Чтобы сделать это, перенесите *Widget Component* на *Event Graph* и вызовите `GetUserWidgetObject` (рис. 6.3).

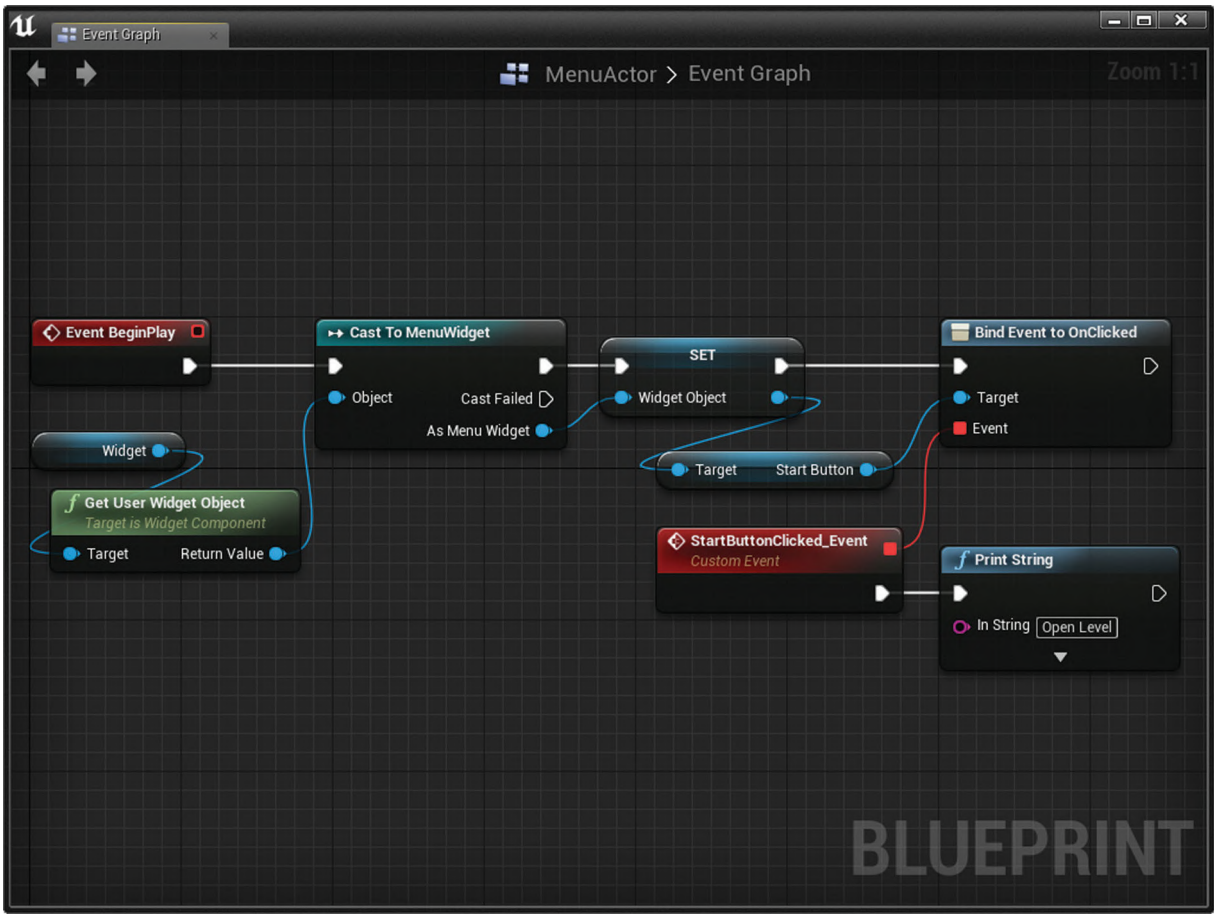

Рис. 6.3. Получение *WidgetObject* от *Widget Component* и назначение события на нажатие кнопки *Start*

8. Чтобы получить доступ к конкретным виджетам внутри меню, от `GetUSerWidgetNObject` создайте узел и соедините с `MenuWidget`, соединив вход выполнения к `EventBeginPlay`.
9. От *As Menu Widget* создайте новую переменную типа `Menu Widget`, нажав *Promote to Variable*. Назовите ее `WidgetObject`. Это сделает доступ легким для последующего использования.
10. От голубого выхода сеттера получите `StartButton`, которая связана со *Start button* в вашем меню.
11. От последнего геттера выберите `Assign OnClicked`. Это создаст событие и автоматически привяжет его к диспетчеру от кнопки (см. рис. 6.3).
12. Теперь вы можете делать что угодно с этим событием. Оно будет вызываться при нажатии игроком кнопки `Start`.

6.3.2. Меню *Pawn*

После создания актора для 2*D*-виджета и 3*D*-виджета, вам необходимо создать *Pawn*, который будет хранить в себе *Widget Interaction Component*, который позволит игроку взаимодействовать с 3*D* меню.

1. Создайте новый *Pawn* в папке *Blueprints* и назовите его `MenuPawn`.
2. Создайте три новых компонента: *Scene* под названием `CameraRoot`, *Camera* под названием `Camera` и *Widget Interaction* под названием `WidgetInteraction`.
3. Поместите `Camera` под `CameraRoot` и `WidgetInteraction` под `Camera`. Это сделает `Camera` дочерним элементом от *Scene* и `WidgetInteraction` от `Camera`, что позволит взаимодействовать с виджетом, используя вид из камеры (рис. 6.4).

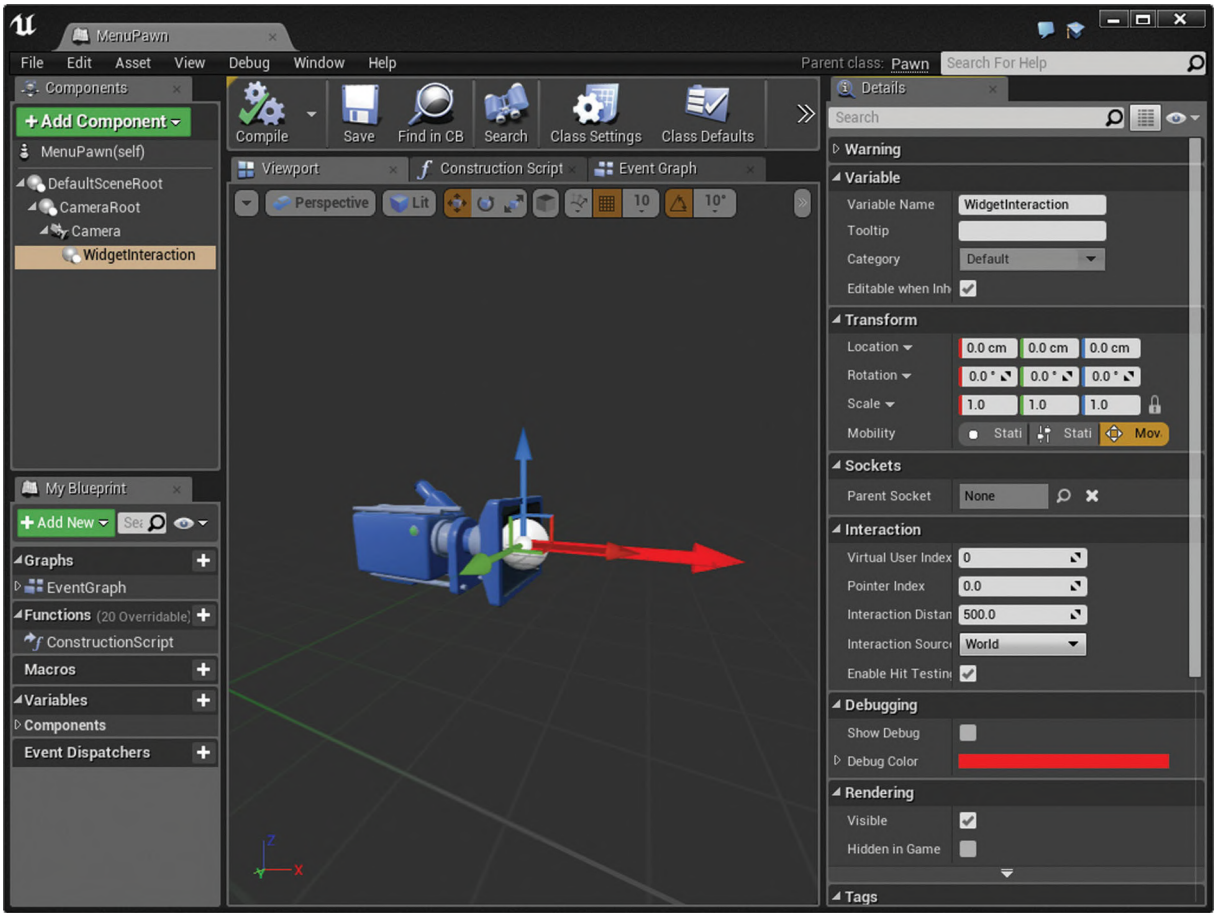

Рис. 6.4. *VR*-меню *Pawn*: базовый компонент *WidgetInteraction*

4. `WidgetInteraction` имеет несколько свойств. `Interaction Source` позволяет вам сказать компоненту, как следует взаимодействовать с виджетом; по умолчанию это `World`, который посылает следы по направлению стрелы и взаимодействует с компонентами, которые они поразили. `Interaction Distance` — это максимальная дистанция взаимодействия. Последнее, `Interaction Component` имеет понятие виртуальных пользователей, которые позволяют вам определять других пользователей и разрешать нескольким игрокам взаимодействовать с виджетом. Настраивается это при помощи свойства `Virtual User Index`.

5. Чтобы получить функциональное взаимодействие, вам необходимо просто активировать событие нажатия. В *Event Graph* создайте узел *LeftMouseButton* (позже вы можете изменить это на любое нажатие, например *Touchpad*).
6. Создайте геттер для `WidgetInteractionComponent` и вызовите от него узел *PressPointerKey*, соединив его с контактом `Pressed`.
7. От геттера создайте еще один узел *ReleasePointerKey*, соединив его с контактом `Released` (рис. 6.5). Это симулирует нажатие кнопки на текущем виджете.

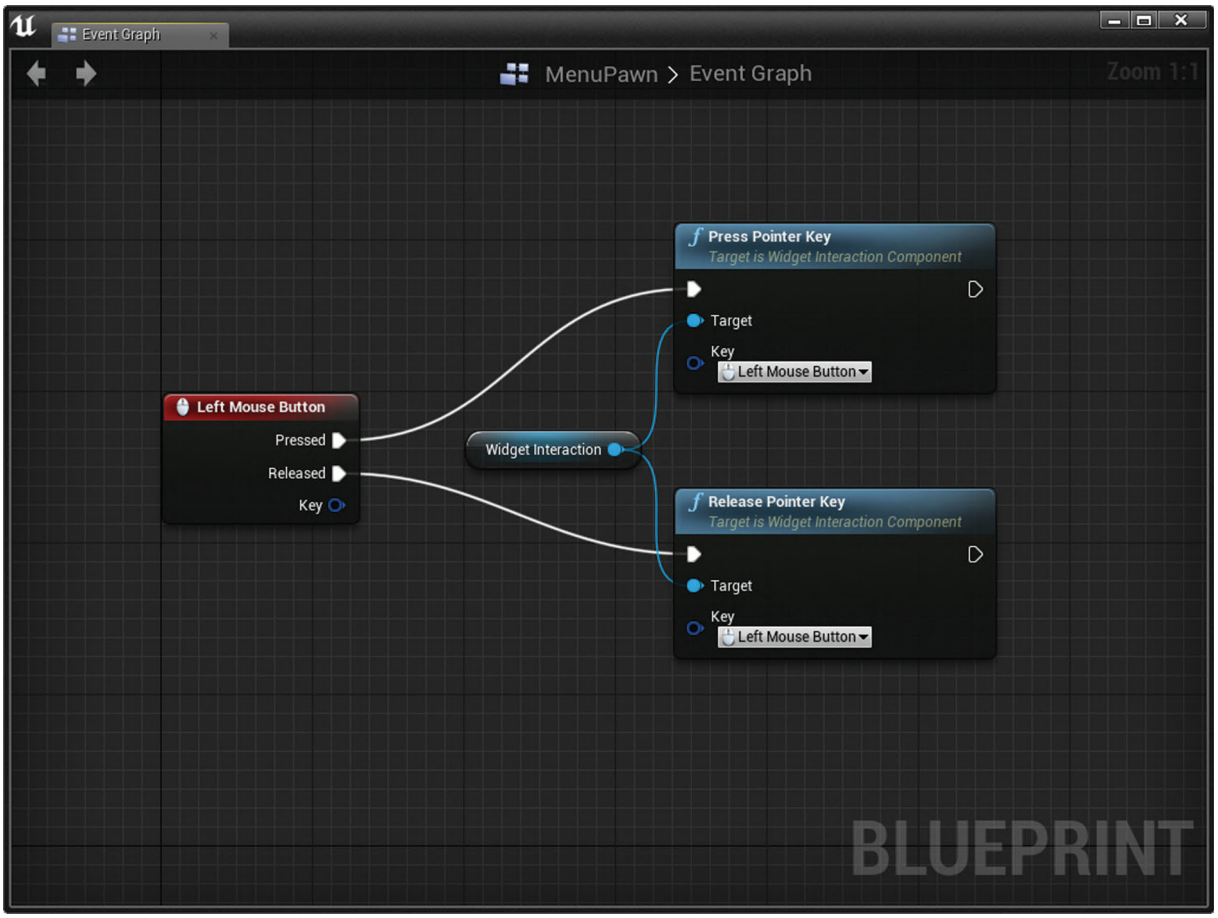

Рис. 6.5. *VR*-меню *Pawn*: активация события нажатия для *Widget Interaction*

Теперь у вас есть основа простого меню, и вы можете добавить *Actor* и *Pawn* на уровень и протестировать взаимодействие.

6.4. Взаимодействие с пользовательским меню

В предыдущем разделе вы использовали метод взаимодействия с меню, который включен по умолчанию, когда используется *Widget Interaction Component*. Однако это очень ограниченный метод, в первую очередь потому, что вы не можете сменить трассировку, предоставляемую компонентом. К счастью, легко интегрировать этот компонент в более сложные системы взаимодействия, такие как вам нужно было разработать в главах 4 и 5. При этом есть два варианта: иметь один *Widget Interaction Component* рядом

с компонентом взаимодействия и информировать об этом взаимодействии *Widget Interaction Component*, чтобы он вызывал нужные функции для взаимодействия с какими-либо меню, которые он поразил; либо разместить на каждом акторе его собственный *Widget Interaction Component*, который будет вызывать нужные функции компонента. Первый подход хорош, так как имеется только один *Interaction Component*; однако второй подход еще лучше, так как ваш собственный *Interaction Component* остается независимым от экспериментального *Widget Interaction Component*. Следующие две подсекции описывают оба способа.

6.4.1. Реализация взаимодействия с меню. Первый способ

Первый подход добавляет функционал взаимодействия в *Trace Interaction Component*, созданный в главе 4.

1. Перенесите *Trace Interaction Component* из главы 4 в этот проект.
2. Откройте его. Создайте новую переменную `WidgetInteraction` типа `Widget Interaction Reference` и сделайте его `Private`.
3. Вы не сможете задать значение по умолчанию для этой переменной в панели компонента его родительского *Blueprint*, потому что *Widget Interaction Component* еще не инициализирован, пока не загрузится уровень, поэтому ссылка будет доступна только в процессе выполнения. Чтобы обойти это, создайте новое событие, назвав его `Setup`, которое на вход принимает *Widget Interaction Component* под названием `WidgetInteraction` (рис. 6.6).

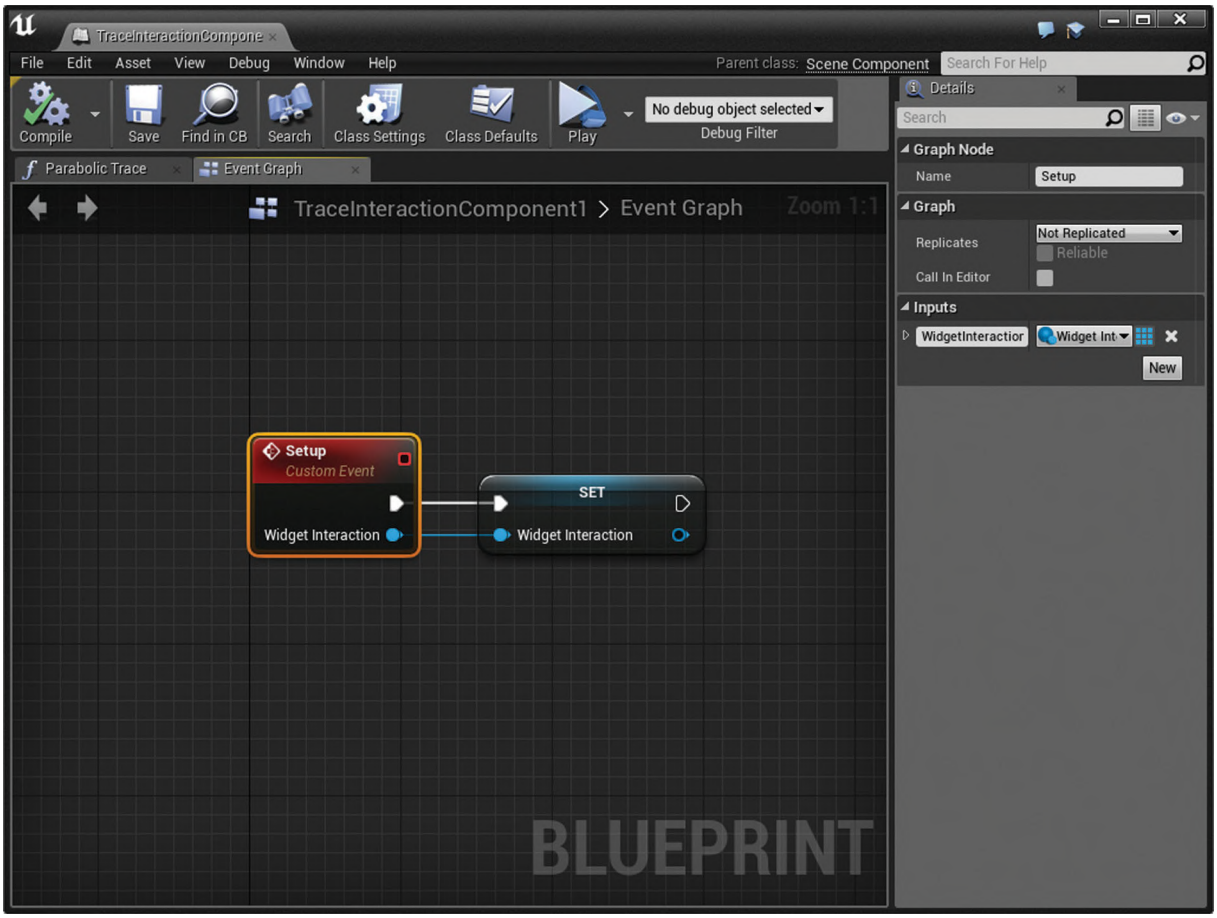

Рис. 6.6. Внедрение зависимостей в *Trace Interaction Component* с помощью *Setup*

4. Создайте сеттер для переменной `WidgetInteraction` и соедините оба контакта с событием. Вы будете вызывать это из *Blueprint* родителя и устанавливать в компоненте то, что вы хотите для использования в *Trace Interaction Component*.
5. Перейдите к `EventTick`, создайте геттер для ссылки `WidgetInteraction` и создайте узел *IsValid* после него. Соедините контакт выполнения с `IntertactWithHit` (рис. 6.7).

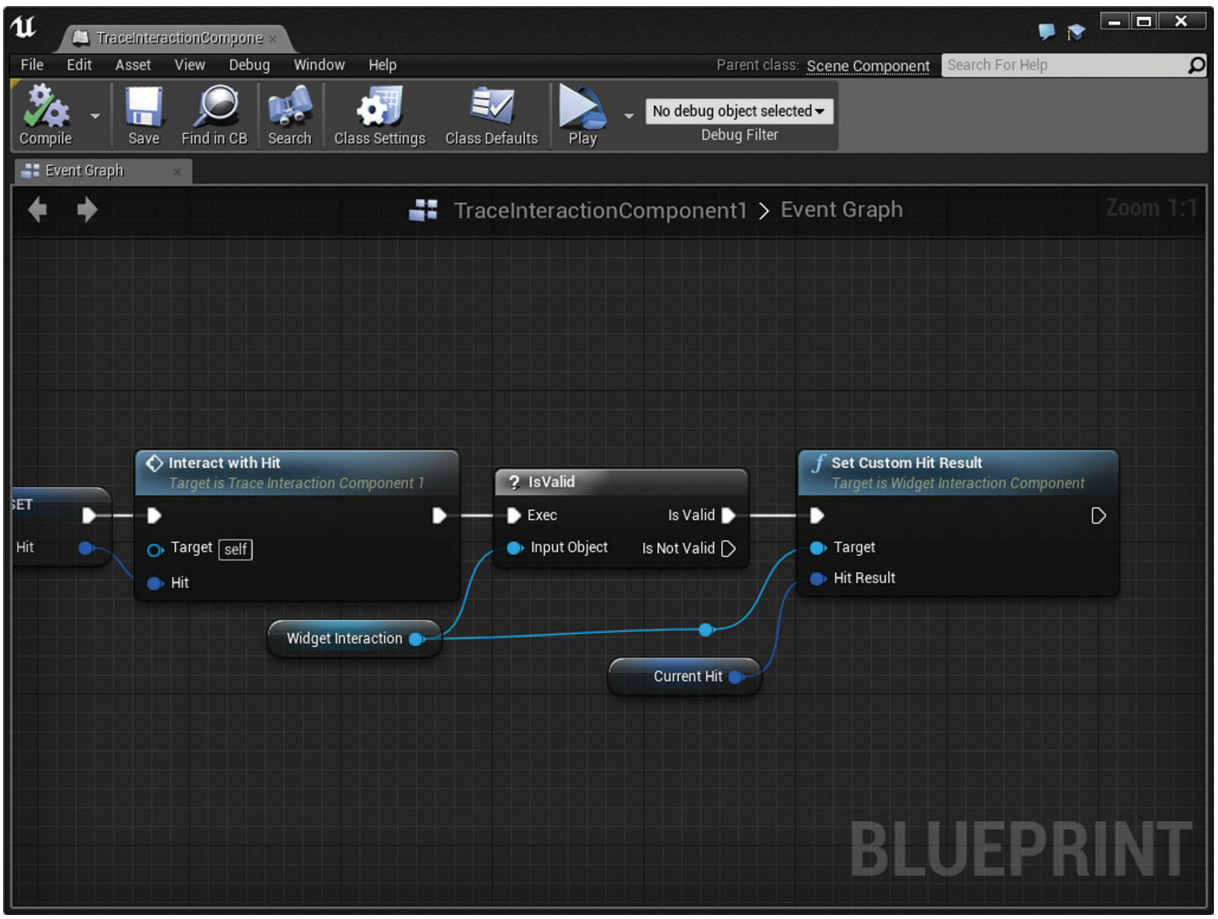

Рис. 6.7. Настройка попадания *Widget Interaction Component* на *Trace Interaction Component*

6. От последнего геттера создайте еще один узел *SetCustomHitResult*, соединив новый геттер для `CurrentHit` с `Hit Result` и контакт выполнения с *IsValid*. Это назначит попадание для данного `WidgetInteractionComponent`, который позволяет ему взаимодействовать с тем, с чем взаимодействует *Trace Interaction Component*.
7. Перейдите к `ActiveDown` и `ActiveUp`, создайте новые геттеры для `WidgetInteraction` для обоих событий и сделайте все, как в шаге 5 (рис. 6.8).

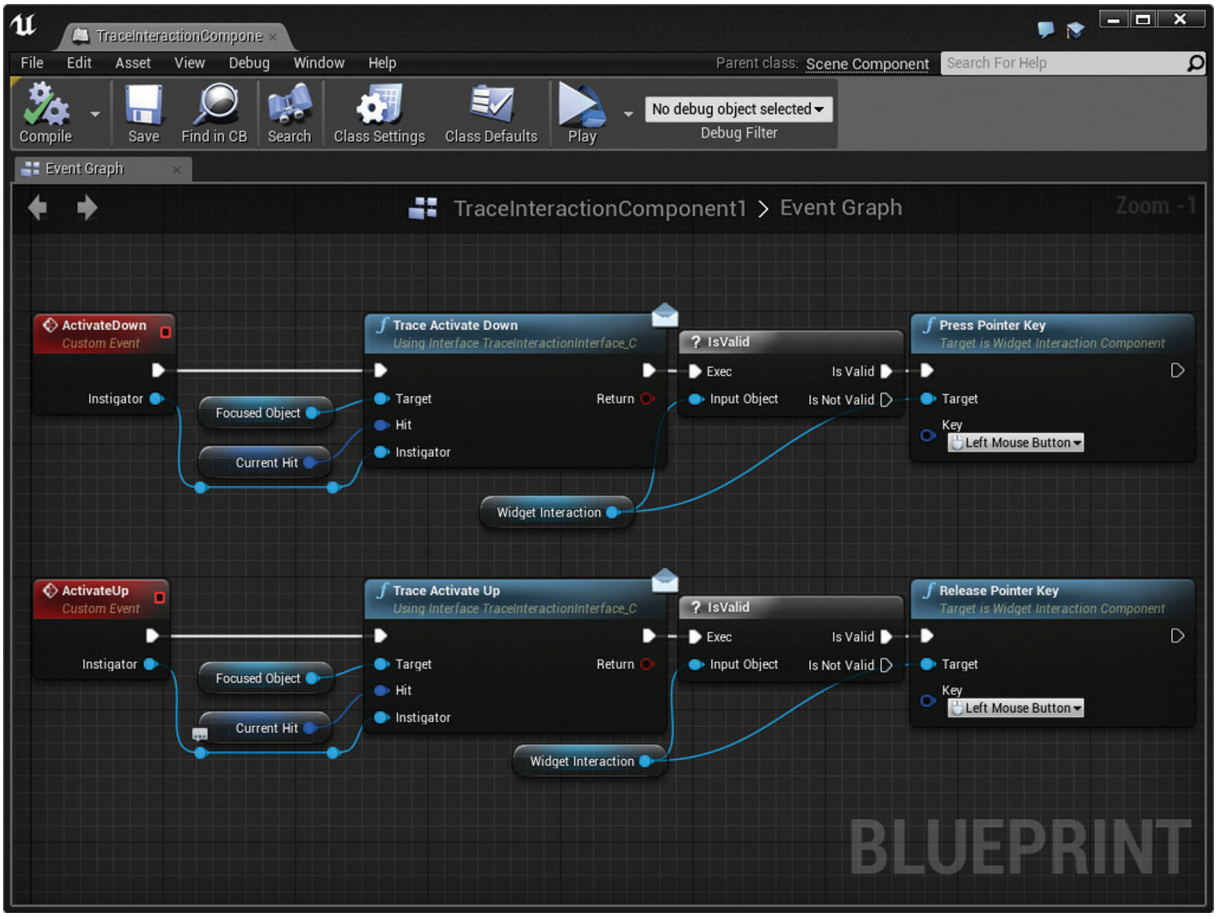

Рис. 6.8. Симуляция указателя на текущем виджете

8. После узлов *IsValid* создайте *PressPointerKey* и *ReleasedPointerKey*, убедившись, что установлено `LeftMouseButton`. Это активирует меню по команде от *Trace Interaction Component*.
9. В папке *Blueprint* создайте *Pawn* под названием `TraceComponentPawn`.
10. Создайте четыре новых компонента в нем: *Scene* под названием `CameraRoot`, *Camera*, *TraceInteraction* и *WidgetInterction*.
11. *Camera* перенесите в `CameraRoot`, а `TraceInteraction` в `Camera`.
12. Выберите `WidgetInteraction` и установите его `Interaction Source` на `Custom` (рис. 6.9). Это значит, что поведение по умолчанию компонента *WidgetInteraction* будет переопределено, и в итоге будет использоваться `SetCustomHitResult` из шага 6.

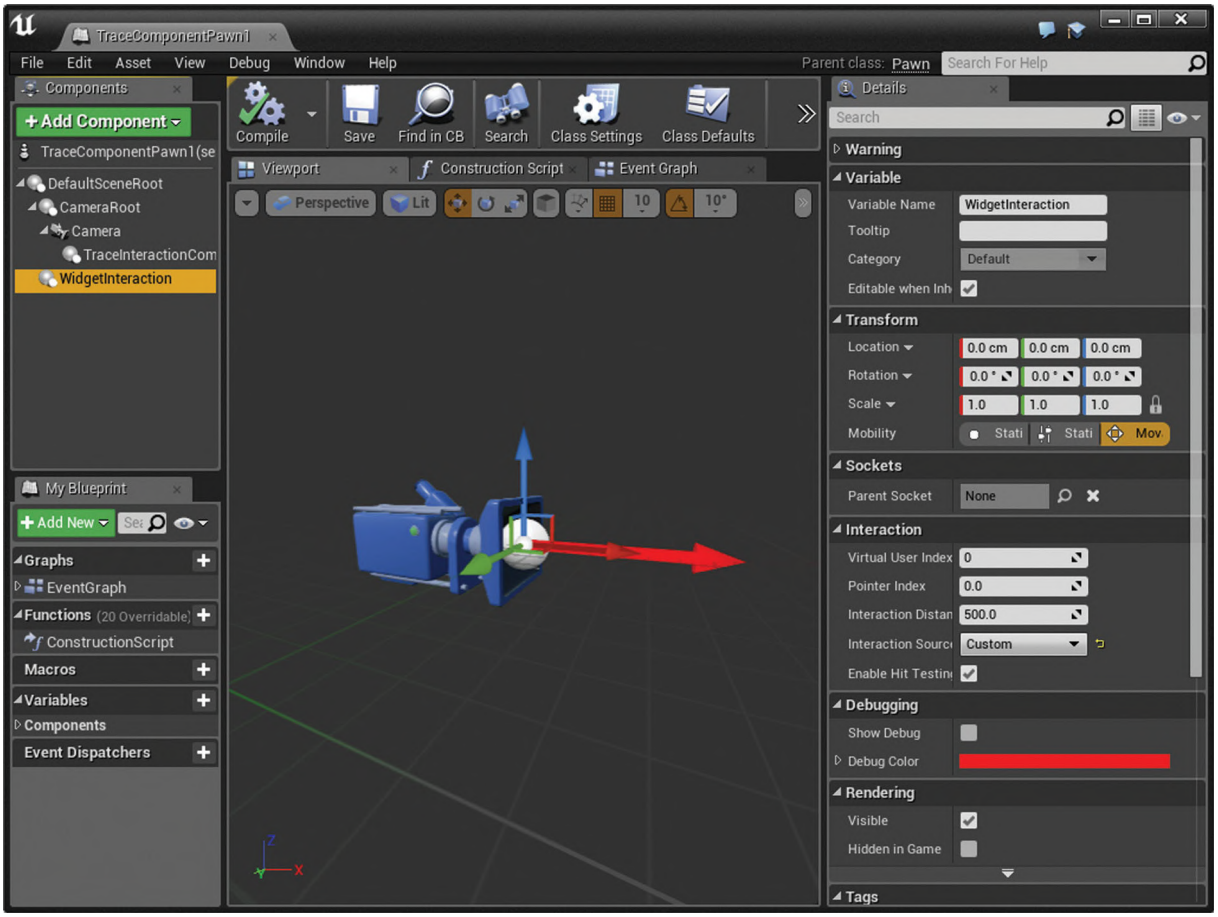

Рис. 6.9. Настройка *Trace Interaction Pawn* с *Widget Interaction Component* на использование результата столкновения

13. Последняя вещь, которую надо сделать для работы *Pawn*, это активировать *Trace Interaction Component*. Сначала в *Event Graph* создайте геттер для *Trace Interaction Component* и вызовите функцию `Setup`.
14. Соедините вызов функции с `EventBeginPlay`.
15. Создайте событие `LeftMouseButton`.
16. Создайте новый геттер для *Tracer Interaction Component* и вызовите `ActivateDown` и `ActivateUp`, установив в контакт `Instigator` узел `Self`.
17. Соедините `Pressed` с событием `ActivateDown` и `Released` с `ActivateUp` (рис. 6.10). Это активирует *Trace Interaction Component*, который, в свою очередь, активирует *Widget Interaction Component*.

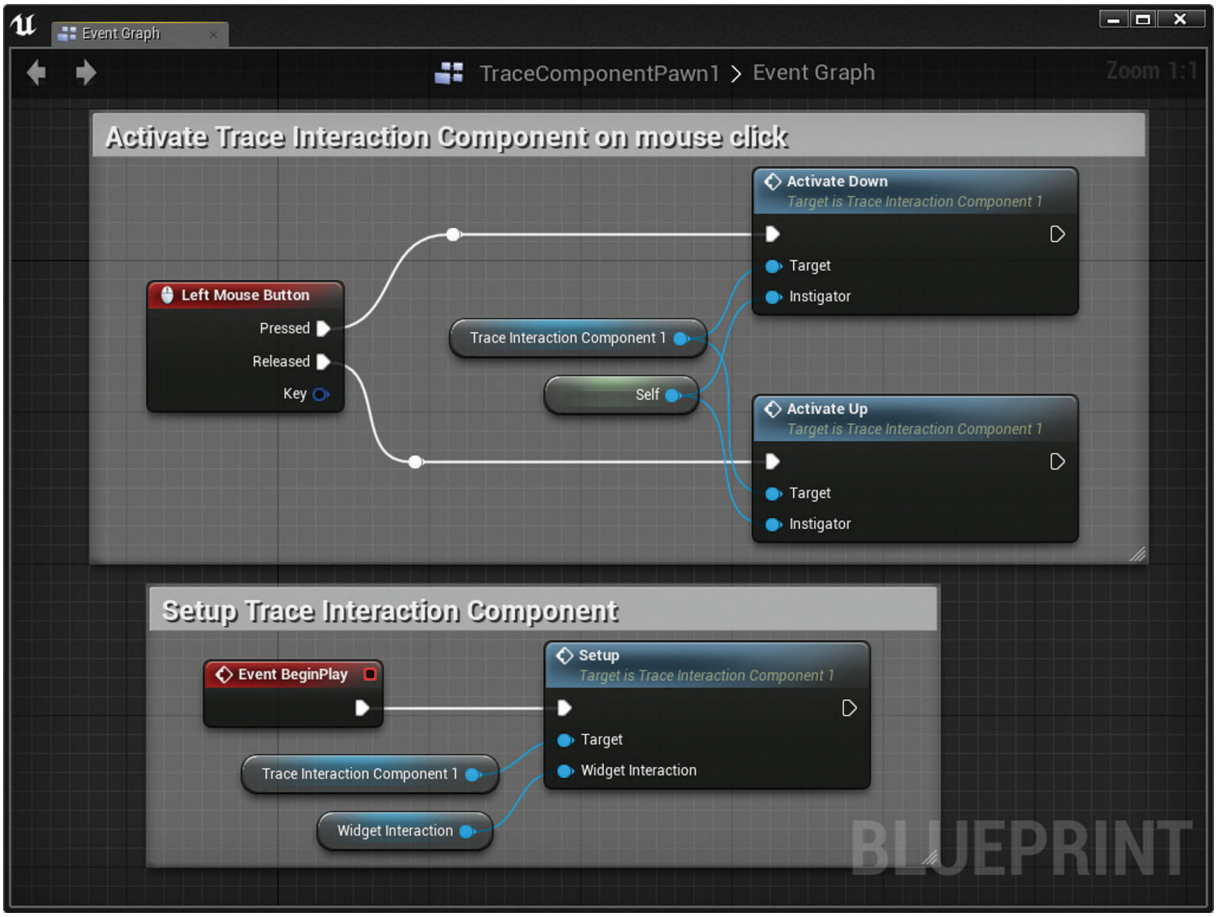

Рис. 6.10. Активация *Trace Interaction Component* нажатием левой кнопки мыши и установление зависимостей в *TraceInteractionComponent* от *BeginPlay*

Сейчас у вас есть общий *Trace Interaction Component*, который работает с *UMG* меню.

6.4.2. Реализация взаимодействия с меню. Второй способ

Второй подход добавления взаимодействие с меню основывается на применении *Trace Interaction Component*, созданного в главе 4, но вместо того, чтобы расширять его, используя уже запрограммированную функциональность, нужно передать работу по взаимодействию с меню на само меню.

1. Перенесите *Trace Interaction Component* из главы 4 в текущий проект.
2. Создайте *Blueprint Pawn* в папке *Blueprint*. Назовите его `TraceComponentPawn`.
3. Откройте *Pawn* и создайте три новых компонента: *Scene*, назвав его `CameraRoot`, *Camera* и *TraceInteraction*.
4. Прикрепите *Camera* к *CameraRoot* и *TraceInteraction* к *Camera*. Это позволит взаимодействовать со всем, на что смотрит игрок.
5. В *Event Graph* создайте событие `LeftMouseButton`.

6. Создайте новый геттер для *TraceInteraction* и вызовите `ActivateDown` и `ActivateUp`, установив в `Instigator` узел *Self*.
7. Соедините `Pressed` с `ActivateDown`, а `Released` с `ActivateUp` (рис. 6.11).

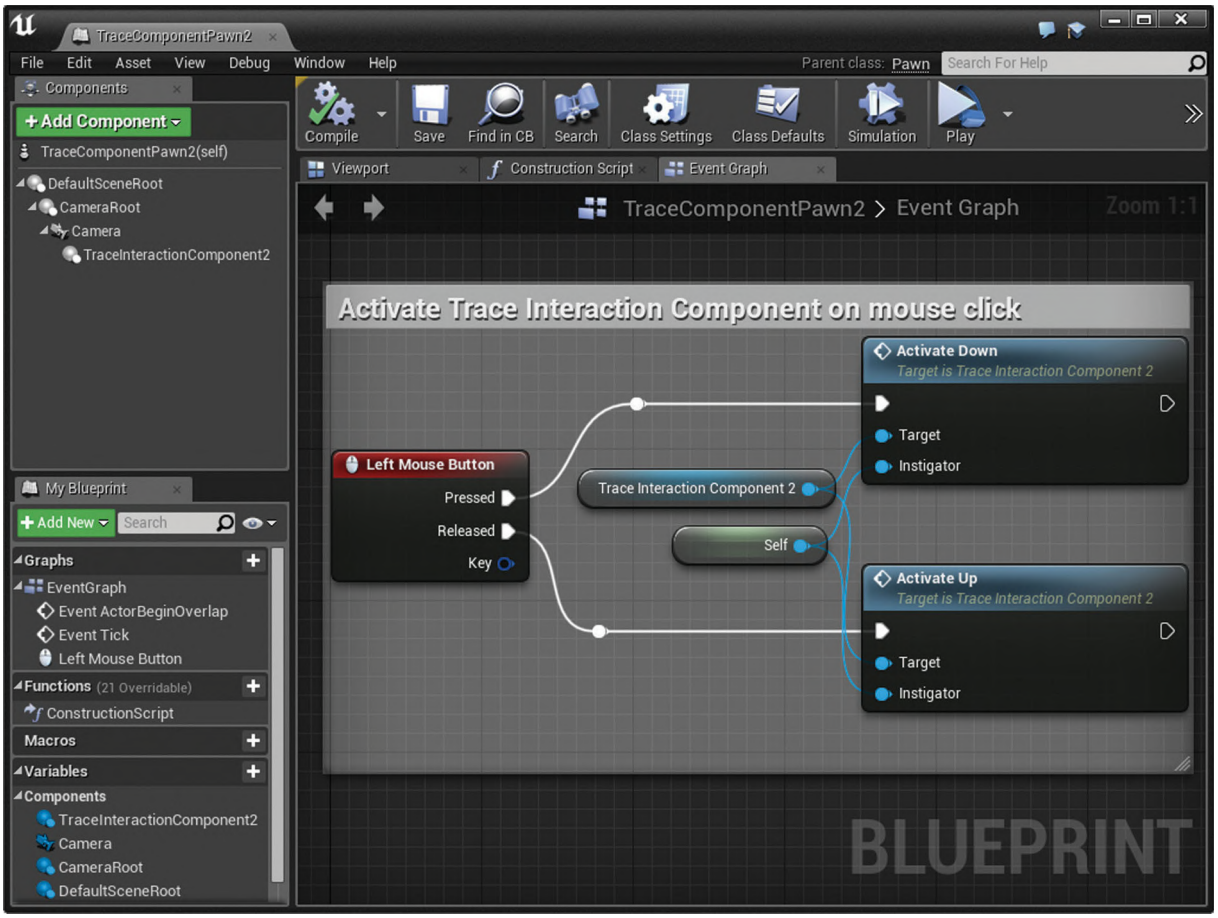

Рис. 6.11. *Trace Interaction Pawn* без *Widget Interaction Component*

8. В папке *Blueprint* откройте *MenuActor* и добавьте *Trace Interaction Interface* в настройках класса.
9. Добавьте новый *Widget Interaction Component* в этот *Blueprint*, установив в `Interaction Source` значение `Custom` (рис. 6.12).

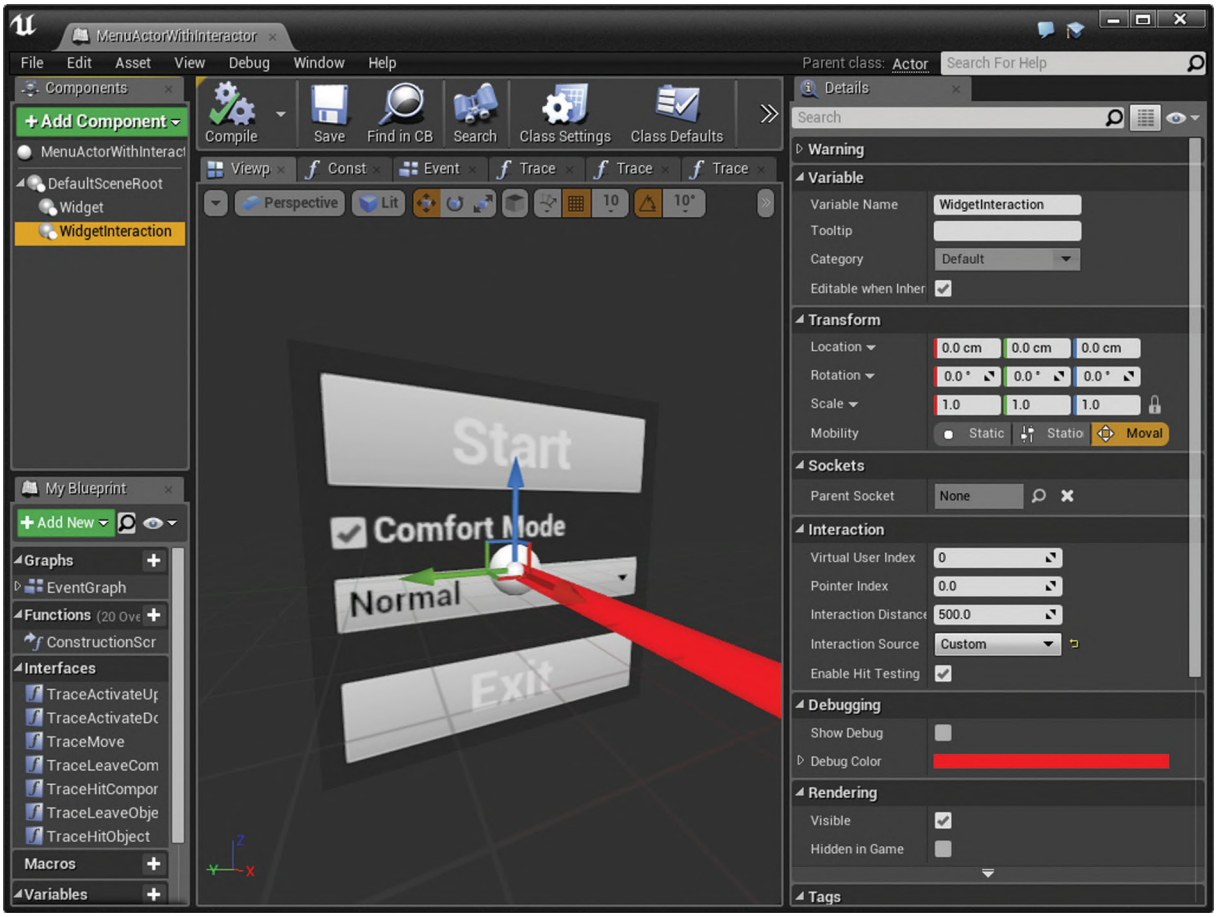

Рис. 6.12. Добавление *Widget Interaction Component* к *MenuActor*

10. Откройте функцию интерфейса `TraceMove` (если вы ее не видите, скомпилируйте *Blueprint*).
11. Создайте новый геттер для *Widget Interaction* и вызовите `SetCustomHitResult`, соединив его между точкой входа и выхода. Соедините `Hit` с `Hit Result` (рис. 6.13).

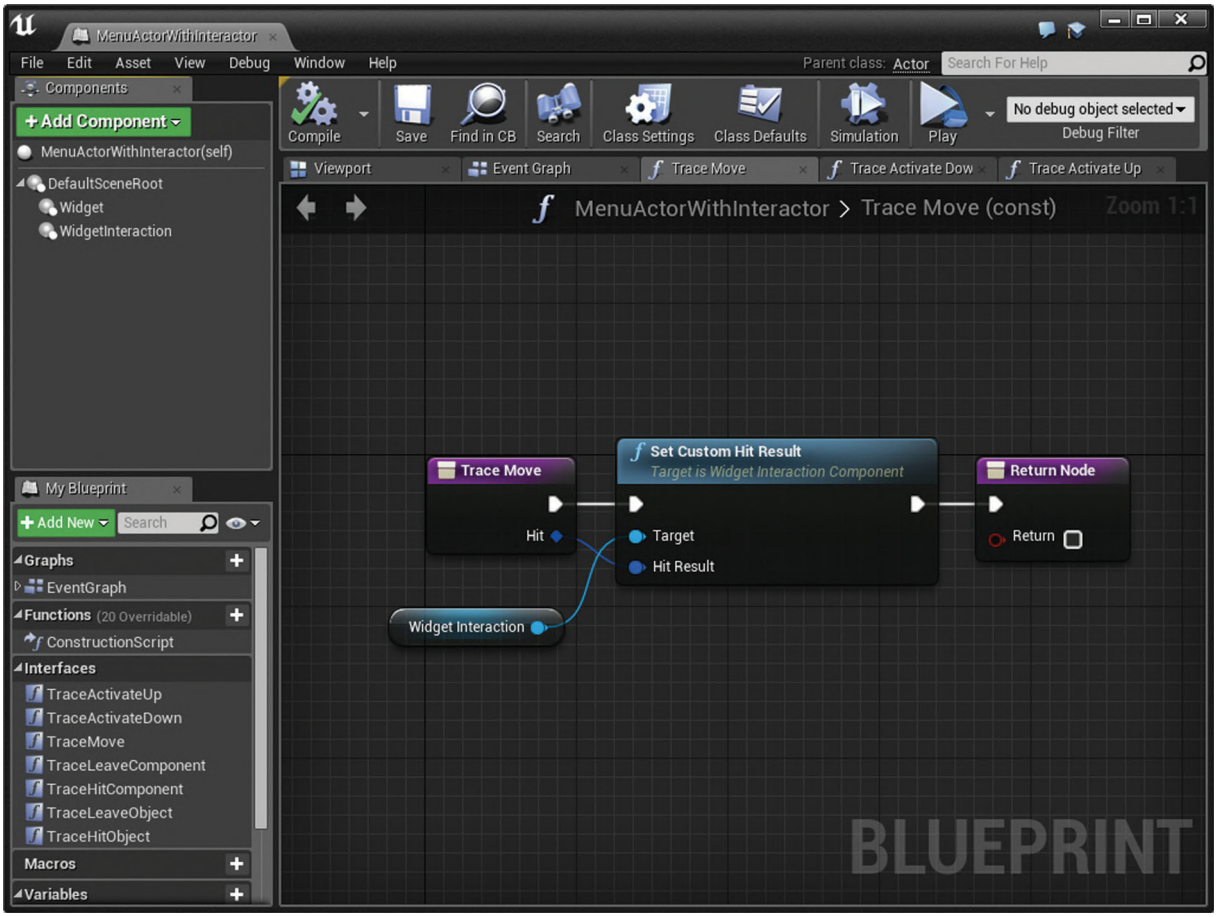

Рис. 6.13. Настройка результата столкновения для *Widget Interaction Component* в *MenuActor*

12. Откройте `TraceActivateDown` и создайте геттер для *Widget Interaction Component*.
13. Вызовите `PressPointerKey`, установив в `Key LeftMouseButton`, и соедините эту функцию между точкой входа и выхода (рис. 6.14).

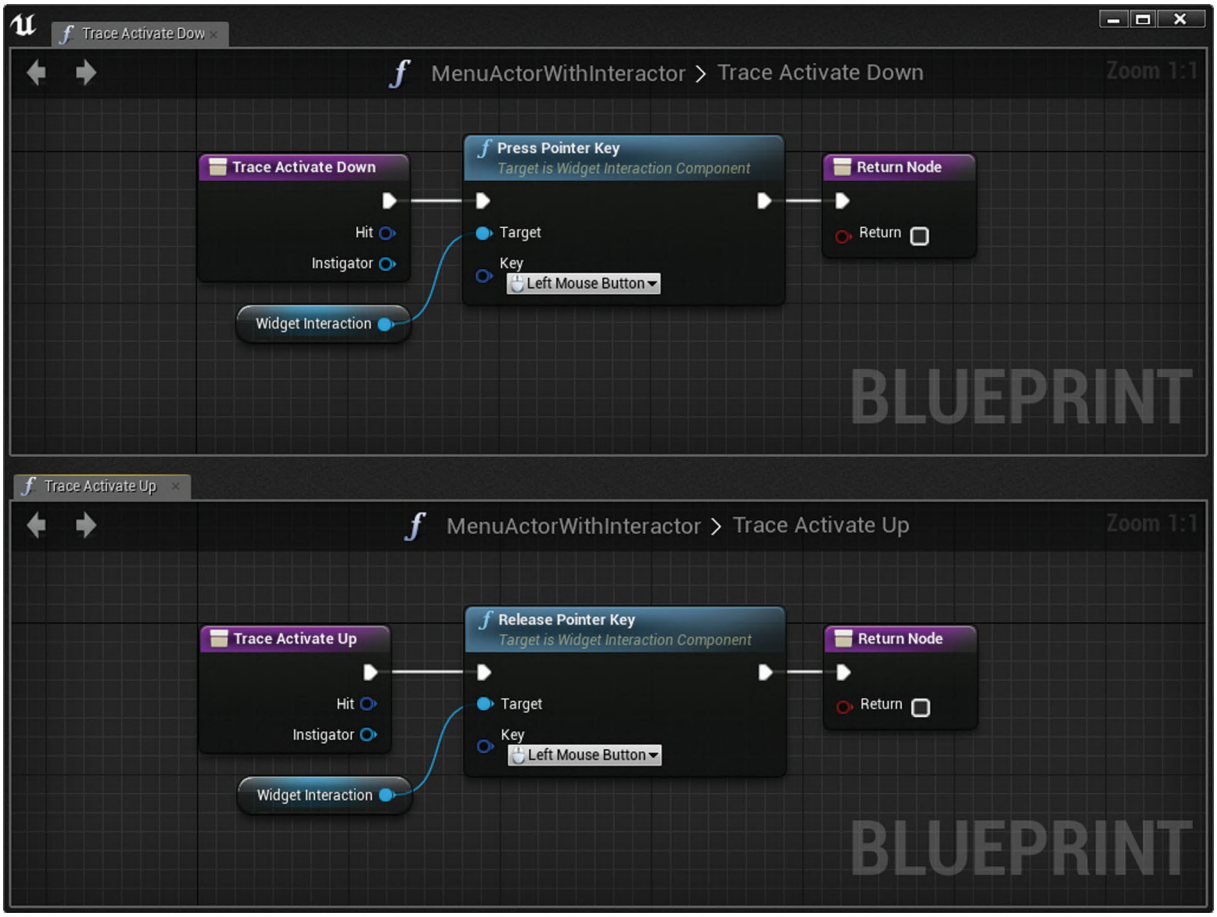

Рис. 6.14. Вызов функций нажатия и отпускания для *Widget Interaction Component* в *MenuActor*

14. Повторите шаги 12 и 13, но теперь вызовите `ReleasedPointerKey`.

Теперь у вас есть работающее меню без *Trace Interaction Component*, и *Trace Interaction Component* не зависит от других компонентов. Единственным недостатком являются возможные проблемы с производительностью из-за наличия нескольких *Interaction Component* в одном мире и более сложная задача обработки многопользовательского взаимодействия, потому что одно меню не будет иметь несколько *Interaction Component*, влияющих на него.

6.5. Заключение

В этой главе мы рассмотрели, как сделать меню в виртуальном мире, а не фиксированным относительно головы игрока. Также мы узнали историю реализации этой функции.

Мы рассмотрели, как с нуля создать простое меню и систему взаимодействия с ним при помощи встроенного *Widget Interaction Component*, а также как интегрировать схожий функционал в систему *Trace Interaction*, созданную в главах 4 и 5.

6.6. Упражнения

Подход, представленный в этой главе, хорошо работает для меню и тому подобных отображений, потому что часто они имеют предсказуемое местоположение. Однако для *HUD*-игрока и других элементов пользовательского интерфейса, положение которых пользователь может контролировать, возникают новые проблемы, когда меню существует в 3*D*-пространстве.

Основная проблема, с который вы столкнетесь, — это окклюзия: что делать, когда *UI* пересекается с (или даже полностью покрыт) другими геометриями в игре.

В этом вопросе возможны интересные решения. Можно полностью отключить проверку глубины, создав пользовательский полупрозрачный материал, и выбрать *Disable Depth Test* на вкладке *Transparency*, чтобы меню всегда отображалось поверх других элементов. Тем не менее это может создать стереонесоответствие.

Вы можете использовать пользовательские функции глубины *UE*4 и отображать более прозрачную версию пользовательского интерфейса, когда он пересекается с другими фигурами.

Вы можете делать все, что захотите! Делать игры должно быть весело! Начните думать об интересных решениях этих проблем и выбирайте лучшее для своей игры или сюжета.

Другой популярный вариант меню, если у вас есть контроллеры движения, чтобы прикрепить меню к одной руке и взаимодействовать с этим меню другой. Для этого прикрепите созданный актор меню к одному из контроллеров движения вашего *Pawn*, а ваш виджет/трассировку — к другому. (Вам может понадобиться включить столкновения с самим собой в функции трассировки, чтобы заставить это работать.) Попробуйте эти техники и посмотрите, какая лучше подходит для вашей игры.

ГЛАВА 7

ИНВЕРСНАЯ КИНЕМАТИКА ПЕРСОНАЖА

VR позволяет игрокам «обитать» в теле виртуального персонажа. Правильная репликация реальных движений игроков их виртуальными персонажами может значительно усиливать эффект погружения.

В этой главе рассматриваются методы, встроенные в *Unreal Engine* для интерполяции текущей позы пользователя на основе информации о местоположении игрока.

7.1. Введение в систему инверсной кинематики

В отличие от прямой кинематики, где вы определяете вращение каждой кости*, чтобы получить желаемый результат, обратная или инверсная кинематика (*inverse kinematics*, *IK*) позволяет определить конечное положение костей, подвергающихся воздействию, и позволить системе интерполировать их вращение для достижения цели.

Многие *VR*-шлемы предоставляют возможности по отслеживанию положения головы, чтобы вы точно знали, где находится голова пользователя в виртуальном пространстве. Однако без дорогостоящего оборудования для захвата движения вы, в действительности, не можете точно знать, где находятся другие кости (кроме рук при наличии контроллеров движения).

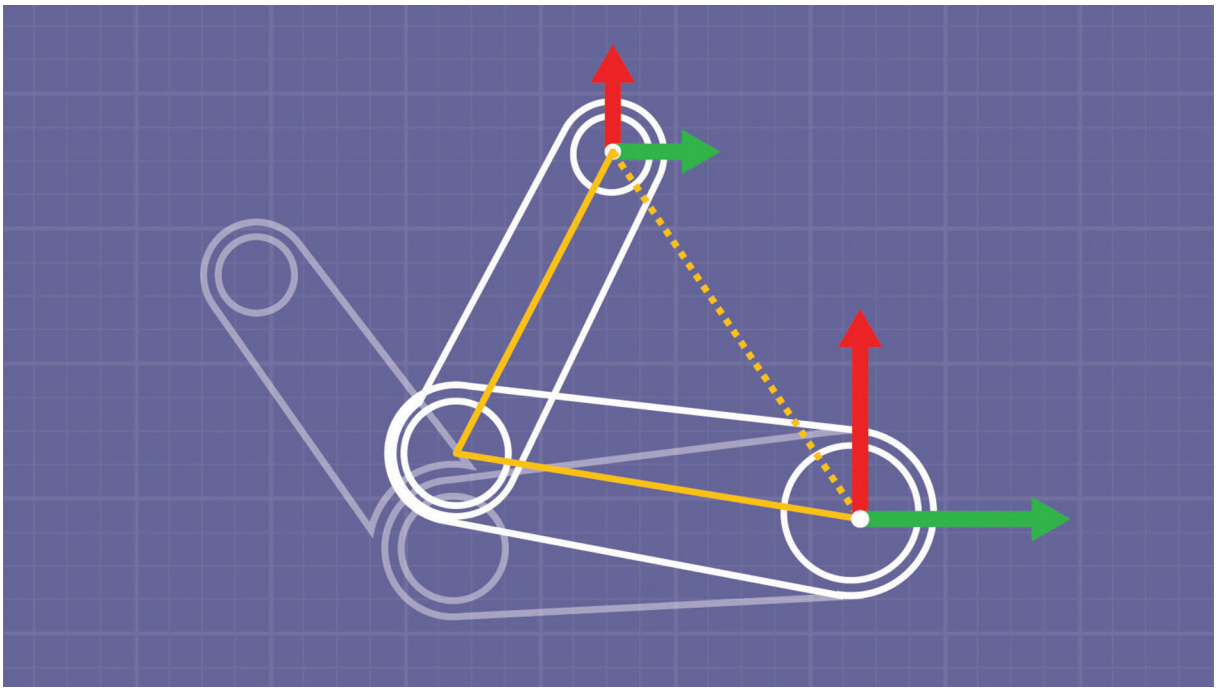

Рис. 7.1. Инверсная кинематика двух костей. Поскольку длины костей известны, дельта от предыдущих вращений костей может быть рассчитана с помощью тригонометрических идентичностей

В этих ситуациях инверсивная кинематика позволяет экстраполировать расположение других костей скелета на основе знаний о способе поворота костей. Двухкостная инверсная кинематика является простейшим ее методом (рис. 7.1), поскольку вращение кости может быть рассчитано аналитически на основе базовых тригонометрических тождеств; однако обратите внимание, что после введения третьего измерения, каковое имеет место в *VR*, существует бесконечное количество возможных решений задачи установки двух костей. Чтобы преодолеть эту сложность, используют полярные векторы (*Polar Vectors*) / совместные цели (*Joint Targets*). Они позволяют разработчикам задавать предпочтения для перемещения костей. В *UE*4 совместные цели определяют точку в пространстве, которая по вектору, направленному вдоль корневой

* Модели инверсной кинематики обычно создаются на основе скелетной анимации, где работают с гибко соединенными жесткими сегментами — костями. — *Прим. пер.*

кости к целевой точке, генерирует плоскость (нормаль к этой плоскости является векторным произведением этих двух направлений). Затем эта плоскость используется для снижения размерности инверсной кинематической задачи до двух измерений.

В *UE4* штатно доступен другой метод инверсной кинематики — метод прямого и обратного следования (*Forward and Backward Reaching Inverse Kinematics*, *FABRIK**). В отличие от реализации инверсной кинематики с двумя костями, *FABRIK* не ограничивает количество костей в инверсивной кинематической цепи. Для этого *FABRIK,* в отличие от двухкостной, модели предлагает не аналитическое, а эвристическое решение задачи инверсной кинематики. Алгоритм *FABRIK* проводит итеративные вычисления, проходя вверх и вниз по цепочке костей и сходясь к решению.

Сначала *FABRIK* устанавливает последнюю кость в цепочке в соответствии с положением конечной цели, затем прокладывает путь обратно по цепочке, перемещая каждую кость к решению, пытаясь сохранить длину кости и прямую линию к предыдущей кости (рис. 7.2). Как только эта итерация достигает самой первой кости и обнаруживает, что ее нужно переместить, алгоритм запускается снова, но в обратном направлении (именно поэтому в названии использованы слова *Forward* и *Backward*), поворот обосновывается тем, что корень цепи не должен двигаться из своего первоначального положения. По умолчанию в *UE4* этот процесс повторяется десять раз, но количество итераций можно изменить на любом узле *FABRIK* в графике анимации.

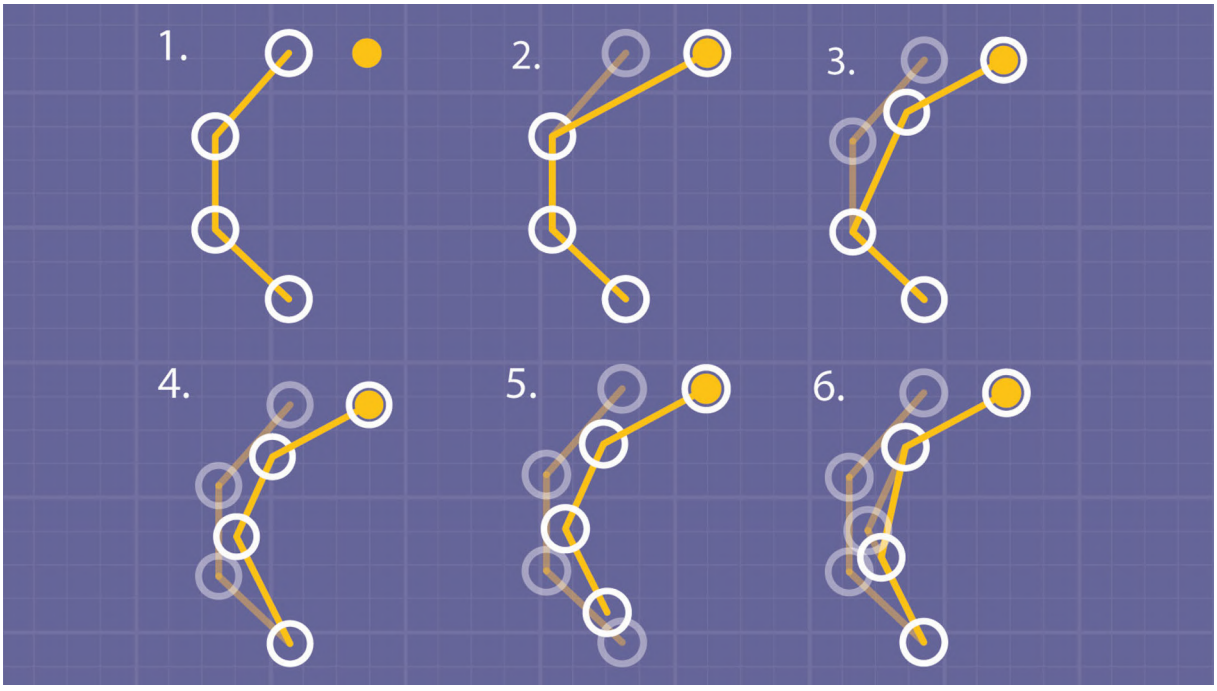

Рис. 7.2. *FABRIK*. 1: начальное состояние кости. 2: конечная кость перемещается к цели (желтый круг). 3–5: следующая кость в цепи перемещается вдоль линии, соединяющей эту кость с предыдущей костью, сохраняя при этом первоначальную длину кости. 6: потому что косточка корня была сдвинута, она двигается назад к своему первоначальному положению, и процесс сделан в обратном порядке.

* Подробнее можно прочитать в статье на Хабре «Инверсная кинематика: простой и быстрый алгоритм» (*https://habr.com/ru/post/*222689/). — *Прим. пер.*

7.2. Настройка инверсной кинематики головы

Чтобы протестировать инверсную кинематику в *UE4*, мы создадим базовую систему инверсной кинематики головы, которая позволит аватару позиционировать свою голову на месте головы игрока, а также сгибать туловище в соответствии с его движениями. Это работает, только если игрок стоит, а не ходит по комнате, потому что туловище человека не отслеживается.

Перед началом создайте проект, который основан на шаблоне *Third Person* без содержания. Это позволит вам протестировать *Blueprint* инверсной кинематики и использовать модель игрока, входящую в этот шаблон.

7.2.1. Создание зеркала

Чтобы знать, что ваша система работает, необходимо получить возможность видеть, куда смотрит игровой персонаж в процессе *VR*-игры. Чтобы это сделать, создадим материал простого зеркала и сгенерируем текстуру.

1. Создайте две новые папки: *Materials* и *Textures*.
2. В папке *Textures* добавьте новый `Render Target` под названием `MirrorRenderTarget` (*Add New* ⇒ *Materials* & *Textures* ⇒ *Render Target*).
3. Установите `Size X` и `Size Y` на `512` (рис. 7.3).

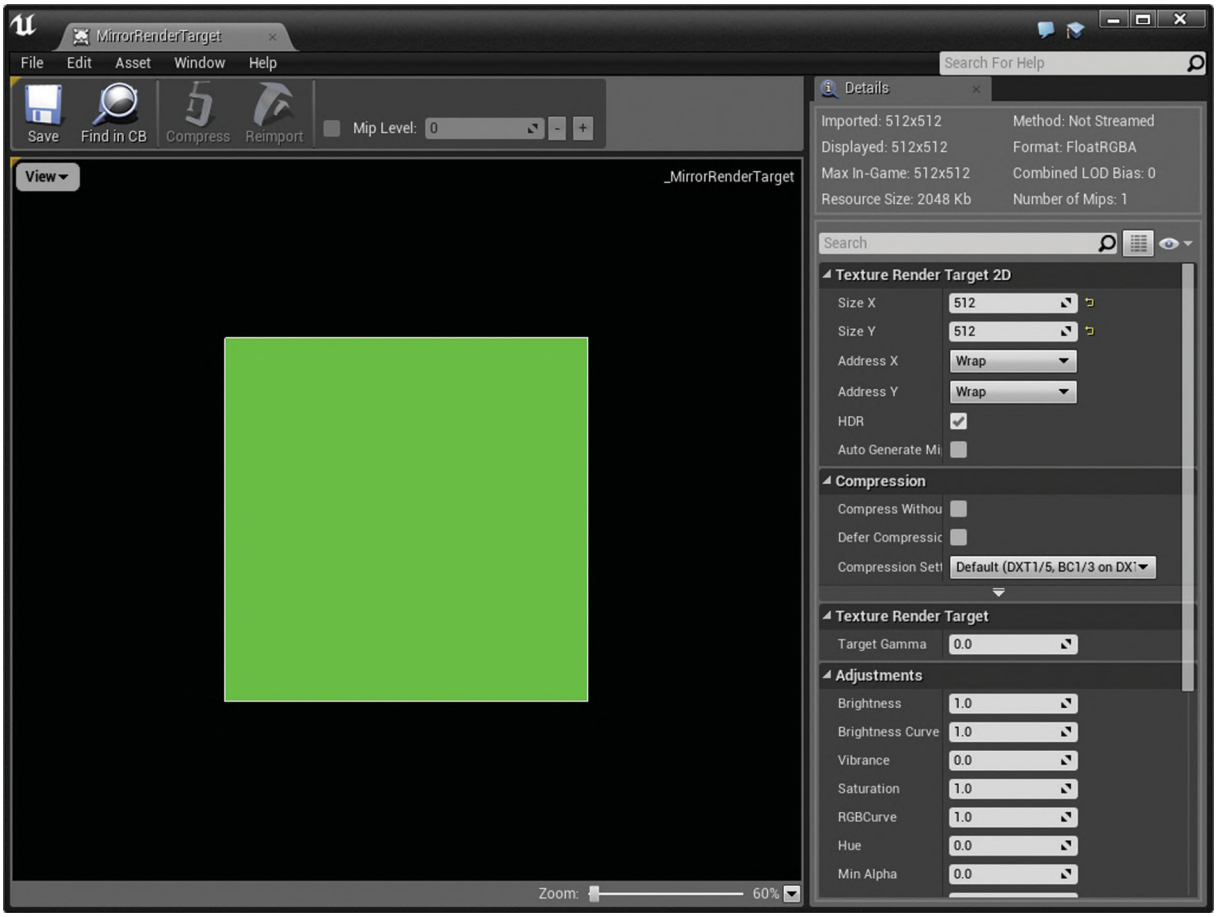

Рис. 7.3. *Mirror Render Target*

4. В папке *Materials* создайте новый материал под названием `MirrorMat`.
5. Отройте его и установите `Shading Model` на `Unlit`.
6. Из *Content Brower* перенесите `MirrorRenderTarget` в `MirrorMat`.
7. Создайте новый узел *TextureCoordinate* и установите `UTiling` на `–1`.
8. Соедините последний узел с `UVs` в `TextureSmple` (рис. 7.4). Это развернет вашу текстуру, что сделает ее более похожей на зеркало.

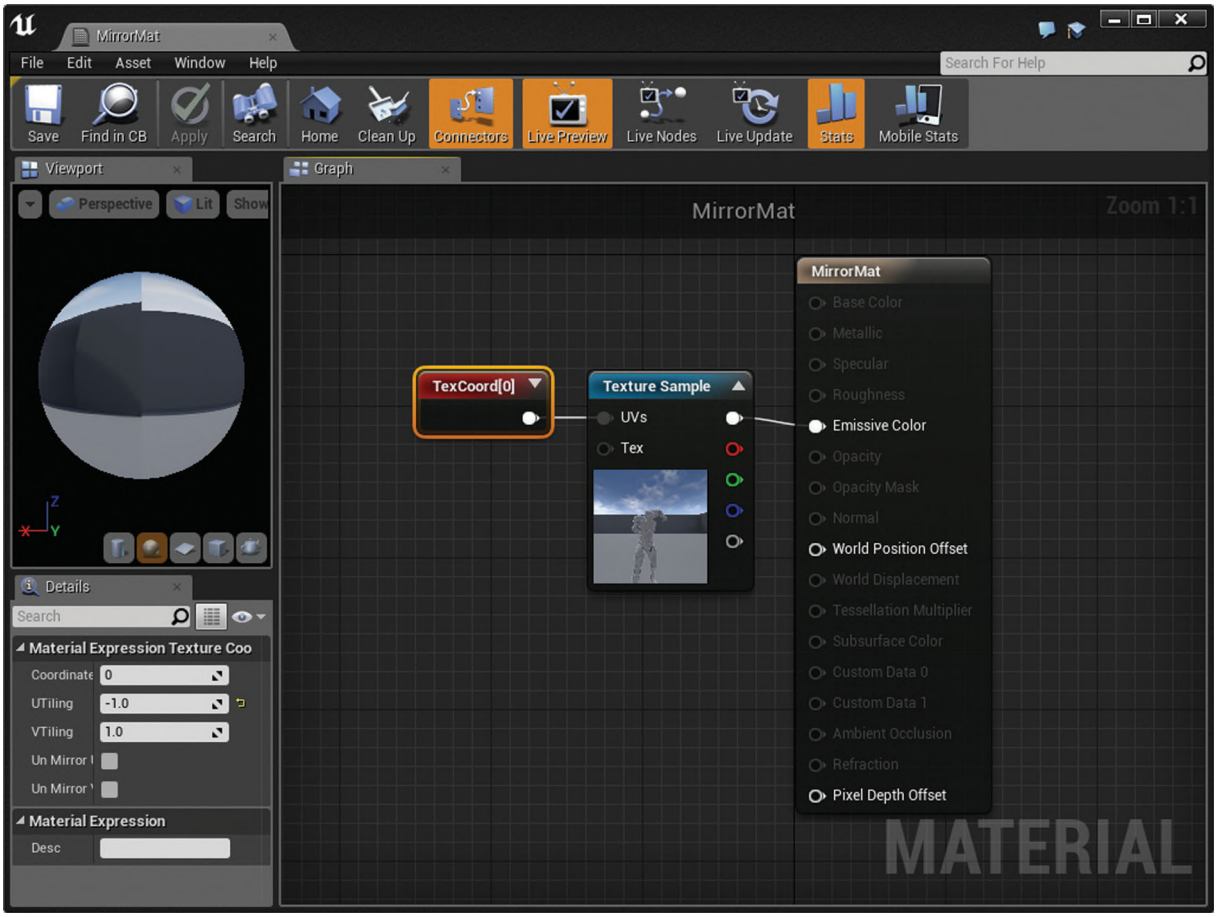

Рис. 7.4. *Mirror Material*

9. Сохраните и закройте этот материал.
10. Из панели *Modes* перенесите *SceneCapture2D Actor*, установите его `Location` на `(X = -650, Y = 390, Z = 260)` и `Rotation` на `(Pitch = 0, Yaw = -180, Roll = 0)`. Это позиция напротив игрока.
11. Установите `Texture Target` в `SceneCapture2D` на `MirrorRenderTarget`. Это помещает все, что захватила камера, в текстуру.
12. Из панели *Modes* перенесите `Cube` на уровень и установите его `Location` на `(X = -640, Y = 390, Z = 260)` и `Scale` на `(X = 0.01, Y = 2.0, Z = 2.0)` (рис. 7.5).

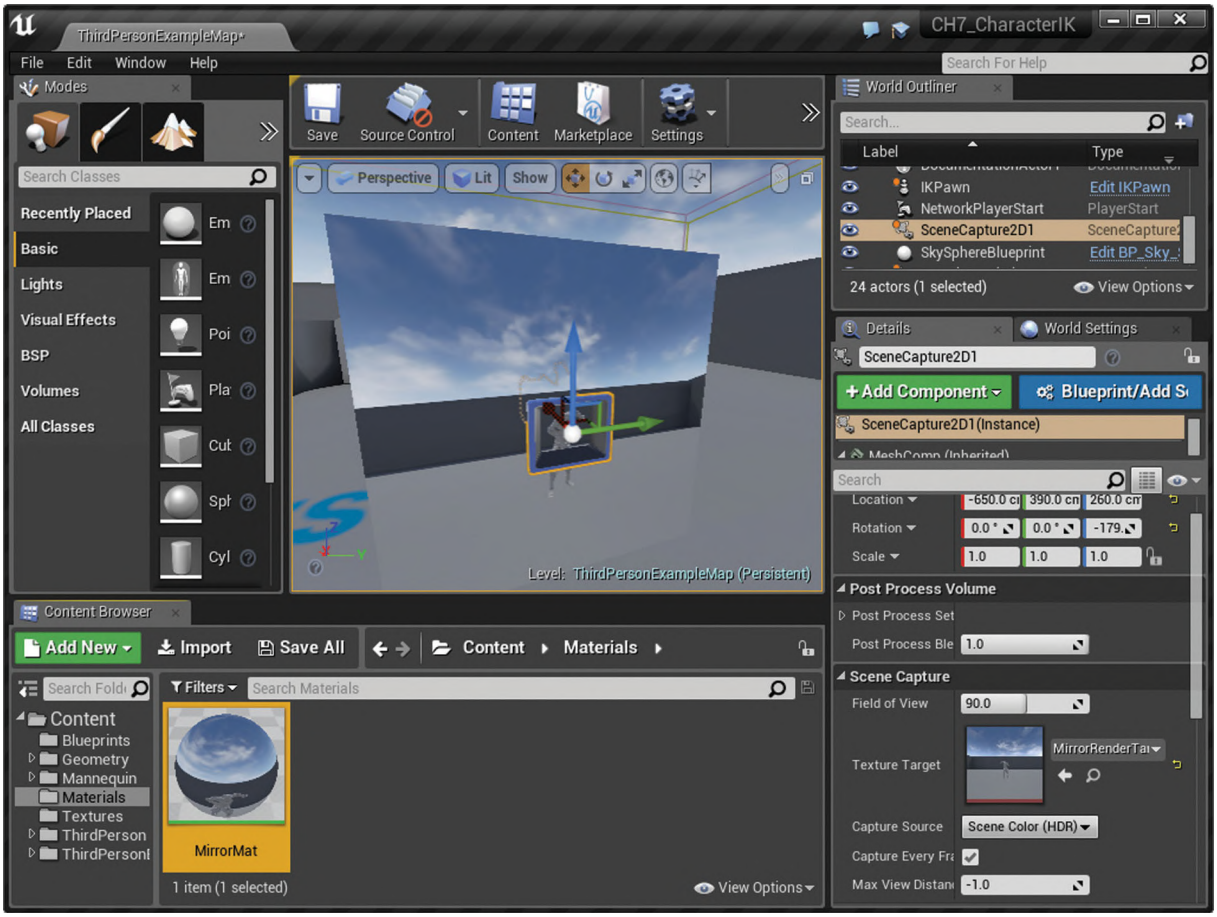

Рис. 7.5. *Scene Capture* и сетка зеркала

13. Установите в `Material Element 0` у `Cube` материал `MirrorMat`.

7.2.2. Pawn для инверсной кинематики

Перед настройкой инверсной кинематики необходимо создать *Pawn* для хранения компонентов, на которые вы ссылаетесь в *Animation Blueprint*.

1. Создайте новую папку *Blueprints*.
2. В этой папке создайте *Animation Blueprints* (в меню *Add new* ⇒ *Animation* ⇒ *Animation Blueprint*), затем выберите `UE4_Mannequin_Skeleton` в созданном диалоге у *Animation Blueprint* (рис. 7.6). Он станет хранить логику инверсной кинематики, которая будет использоваться позже.

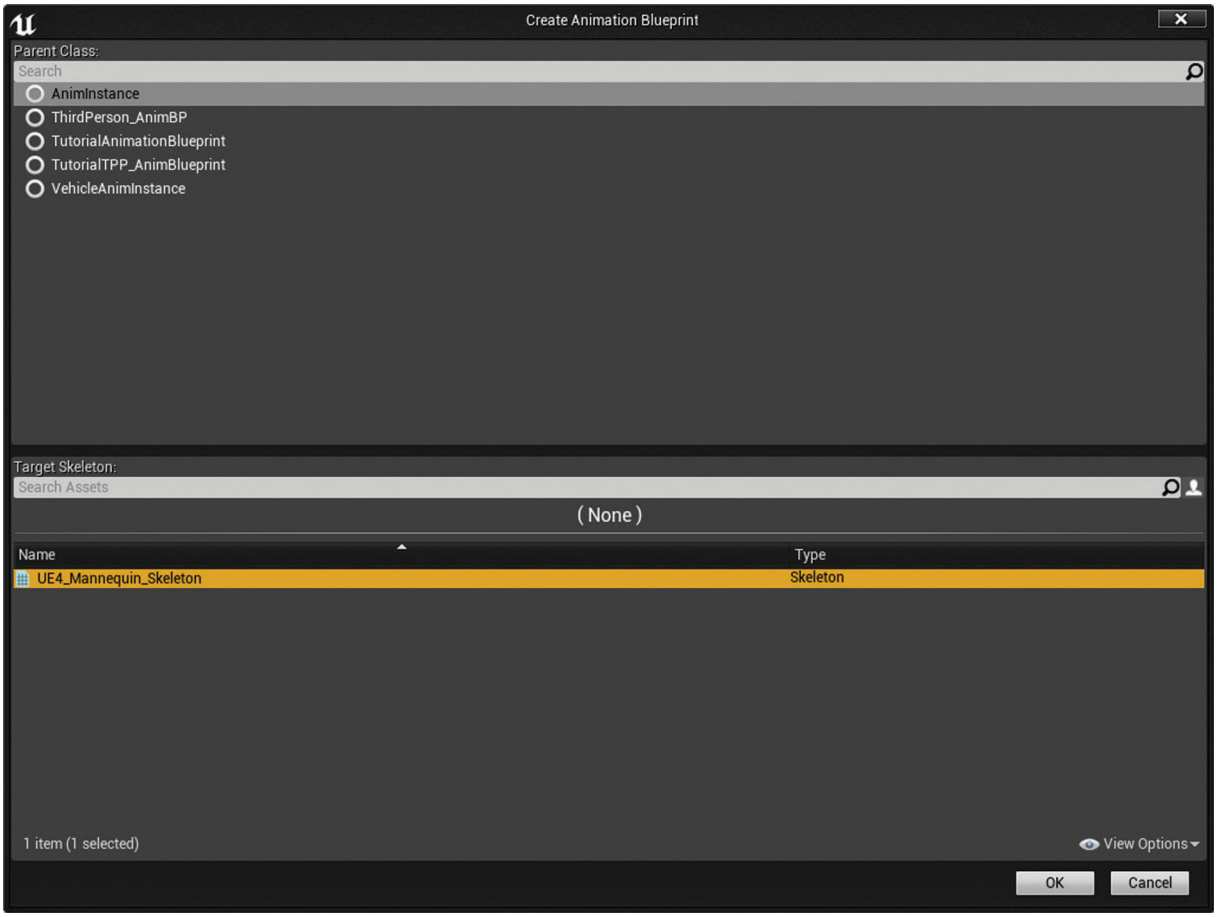

Рис. 7.6. Создание *Animation Blueprint*

3. Назовите этот *Blueprint* `IKAnimBP`.
4. В папке *Blueprint* создайте *Pawn* под названием `IKPawn`.
5. Откройте этот *Pawn* и добавьте три компонента: *SkeletalMesh*, *Camera* и *Scene*.
6. Назовите *Scene Component CameraRoot* и прикрепите *Camera* к нему.
7. Установите *Camera* на `170` на оси `Z` и `10` по оси `X` (рис. 7.7).

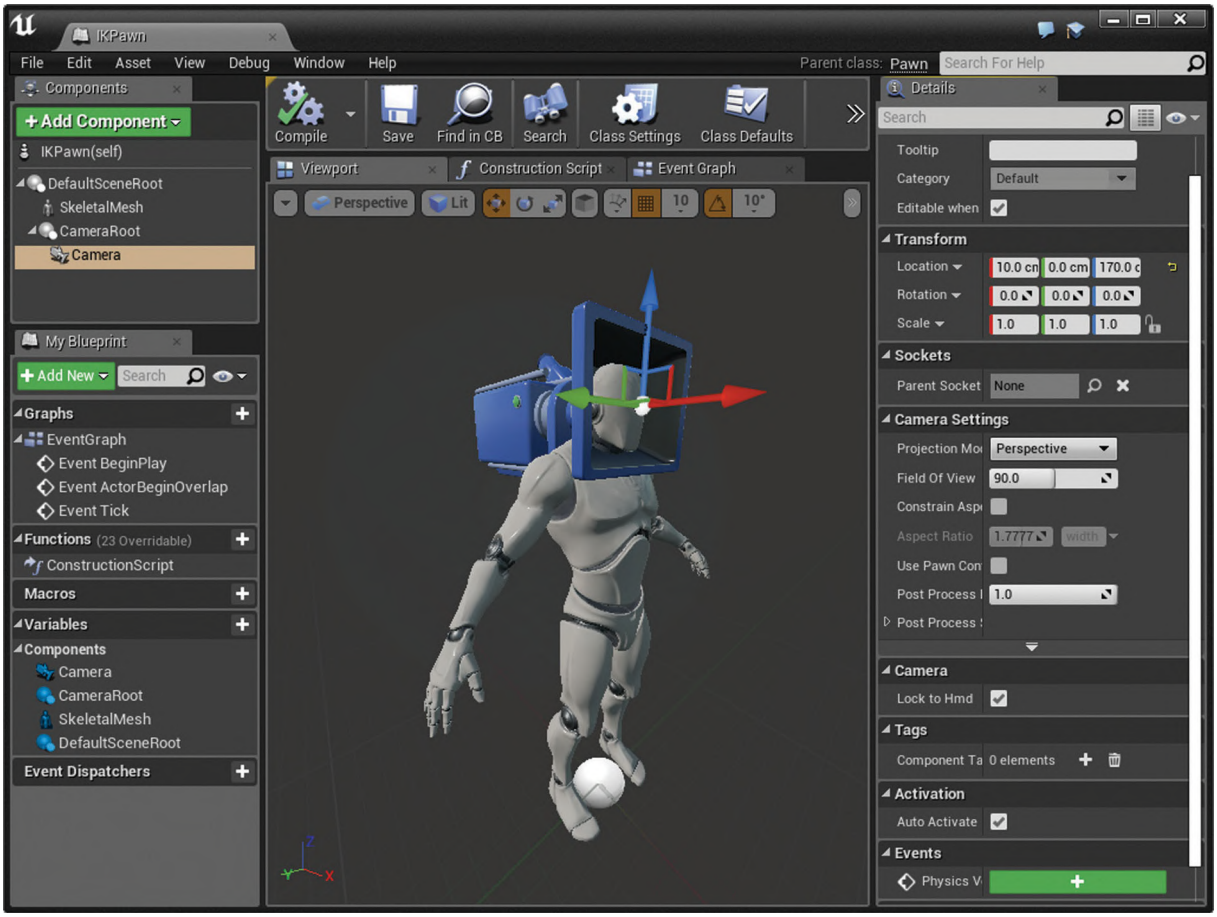

Рис. 7.7. Настройка камеры в *IKPawn*

8. Поверните `SkeletalMesh` на `-90` по оси `Z`. Так мы удостоверимся, что сетка обращена вниз по оси `X`.
9. Установите свойство `Skeletal Mesh` у *SkeletalMesh* на `SK_Mannequin` из *Third Person Template*.
10. Установите класс `Anim SkeletalMesh` на `IKAnimBP`.

7.2.3. Blueprint анимации инверсной кинематики головы

Применяя инверсную кинематику к скелету героя из шаблона, вы будете использовать *FABRIK*, потому что у этого скелета более чем две кости от головы до таза. Хотя вы можете собрать цепь из узлов двухкостной инверсной кинематической модели, чтобы получить похожий эффект, это потребует дополнительных усилий; готовый узел *FABRIK* будет работать хорошо для инверсной кинематики головы.

1. Откройте *Blueprint* `IKAnimBP` и создайте новую переменную типа `Transform` под названием `HeadWorldTransfom`.
2. Установите `Location` у этой переменной на `160` по оси `Z` и `Rotation` на `90` по оси `Z`. Это обеспечит нужное направление головы.
3. Создайте узел *TryGetPawnOwner* в *Event Graph* и прикастуйте к созданному `IKPawn`.

4. От контакта `As IKPawn` создайте геттер для `Camera Component` этого *Pawn*.
5. Создайте сеттер для переменной `HeadWorldTransform` и соедините его с входом вызова при удачном касте.
6. От геттера для `CameraComponent` создайте `GetWorldTransform`, установив выходное значение в сеттер из пункта 5 (рис. 7.8). Вы используете пространство мира в примере, потому в нем проще работать.

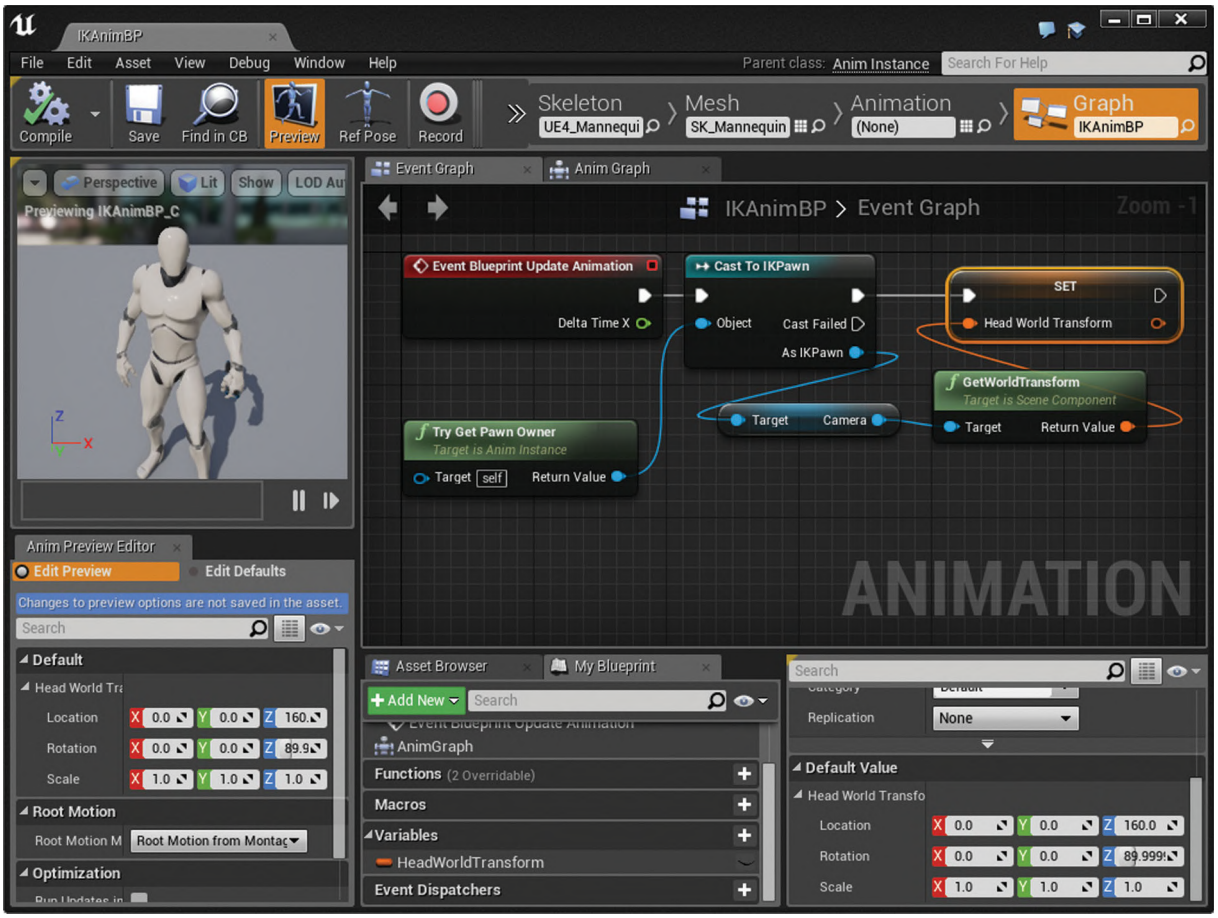

Рис. 7.8. *Blueprint* анимации: настройка позиции головы

7. Перейдите в *Anim Graph* и создайте узел `PlayThirdPersonIdle`.
8. Создайте новый узел *FABRIK* и соедините переменную `HeadWorldTransform` с контактом `EffectorTransform`.
9. Выберите узел *FABRIK* и установите `Effector Transform Space` на `World Space`
10. Установите `Tip Bone` на `head` и `Root Bone` на `spine_01` (рис. 7.9). Узел *FABRIK* будет обрабатывать инверсную кинематику на всех костях между парами, которые вы выберете.

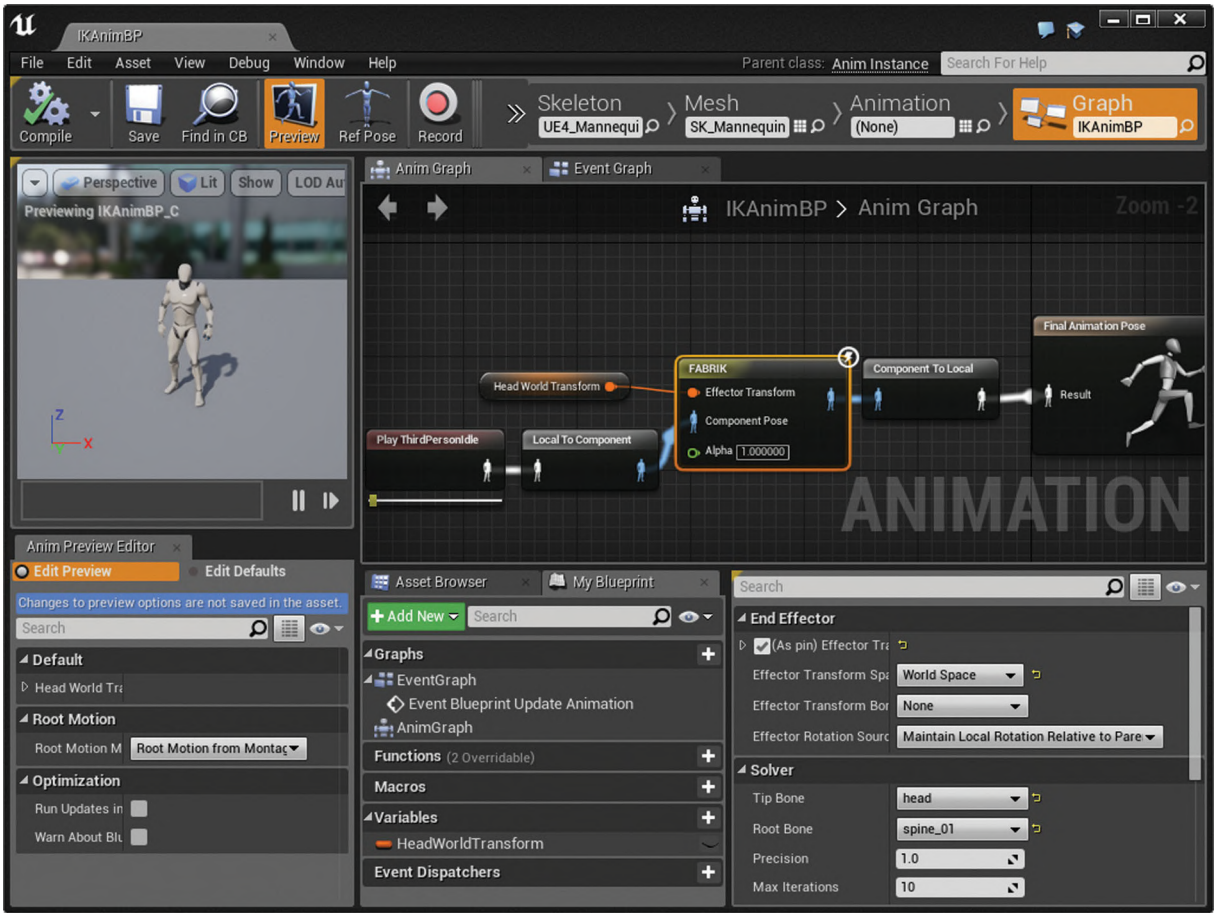

Рис. 7.9. *Blueprint* анимации: простая *IK* головы

11. Соедините `PlayThirdPersonIdle` с *FABRIK*. *UE* автоматически за вас конвертирует локального пространства в пространство компанента.
12. Узел *FABRIK* соедините с узлом `FinalAnimationPose`.
13. Перед вращением головы обратите внимание: на рис. 7.10 вы заметите, что передняя ось кости головы (красная) смотрит вверх. Это значит, что вам необходимо учесть это при настройке вращения. Создайте новый узел *Transform* (*Modify*) *Bone* в *Anim Graph*.

Рис. 7.10. Стандартное вращение головы в UE4

14. Установите атрибут `Modify` узла `Bone` на `head`.
15. Установите `Rotation Mode` на `Replace Existing` и `Rotation Space` на `World Space` (рис. 7.11).

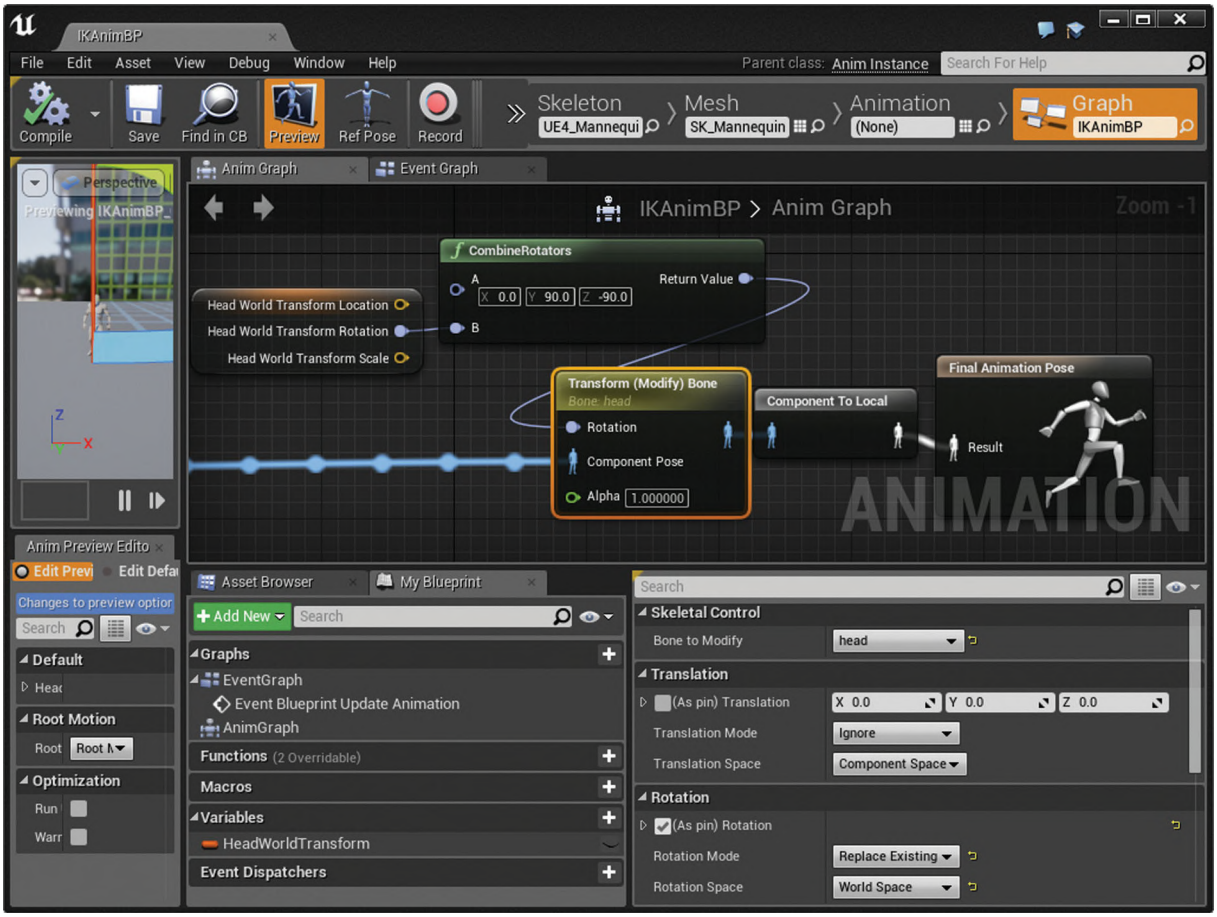

Рис. 7.11. *Blueprint* анимации: поворот кости головы

16. Создайте новый узел *CombineRotator*, установив значение `(Pitch = 90, Yaw = -90, Roll = 0)` в первом входе для инициализации кости.
17. Создайте геттер для переменной `HeadWorldTransfrom` и соедините `Rotation` с *CombineRotator*, разделив выход узла (см. рис. 7.11).
18. Соедините выход *CombiteRotators* с `Rotation` узла *Transform* (*Modify*) *Bone*.
19. Соедините `Transform Bone` между *FABRIK* и `ComponentToLocal`.

Закончите *Pawn* и *Animation Blueprint*, протестируйте вашу *IK*, удалив стандартный *Pawn*, который есть на уровне *Third Person Example* и перенесите на нее ваш *IKPawn*. Затем установите *Auto Prossess Player* на *Player* (это позволит вам использовать этот *Pawn* на уровне без *Game Mode*). Если вы используете *Oculus Rift*, отслеживание по умолчанию стоит на уровне глаз, поэтому вам необходимо установить отслеживание на *Floor* (см. главу 2).

7.3. Настройка инверсной кинематики руки

Если вы протестировали *Pawn* и инверсную кинематику в предыдущем разделе, то заметили, что она работает достаточно хорошо, и если у вас будут контроллеры, то возникнет соблазн перемещать ваши руки.

Чтобы настроить перемещение рук, вам необходимо использовать две простые кости инверсной кинематики потому, что приятно работать с двумя *FABRIK* и двумя системами костей, а также *IK* между руками и плечами должна работать хорошо.

Мы будем работать с кинематикой из предыдущей секции, поэтому, если вы ее не закончили, вернитесь к ней.

7.3.1. Добавление контроллеров к *Pawn*

Перед установкой инверсной кинематики для вашего *Pawn* необходимо добавить компоненты контроллеров, которые будут использоваться для отслеживания рук.

1. Откройте `IKPawn`.
2. Добавьте два компонента *Motion Controller*. Первый назовите `MotionController_L`, а второй — `MotionController_R`.
3. Для правого установите переменную `Hand` на `Right`.
4. Создайте два новых компонента *Static Mesh*, который будут представлять собой контроллеры.
5. Если вы используете *Unreal* 4.13 и выше, вам нужно разрешить мешам строиться в движке (выберите *Show Engine Component* в *View*). Если это сделано, выберите соответствующее свойство *Static Mesh* для вашего устройства для двух компонентов. (Например, я использую *Oculus*, поэтому я выберу *OculusControllerMesh Static Mesh*).
6. Назовите эти меши соответственно вашим контроллерам (в данном случае `Touch_L` и `Touch_R`) и прикрепите их к нужным компонентам. Обратите `Y scale` правого меша при необходимости (рис. 7.12).

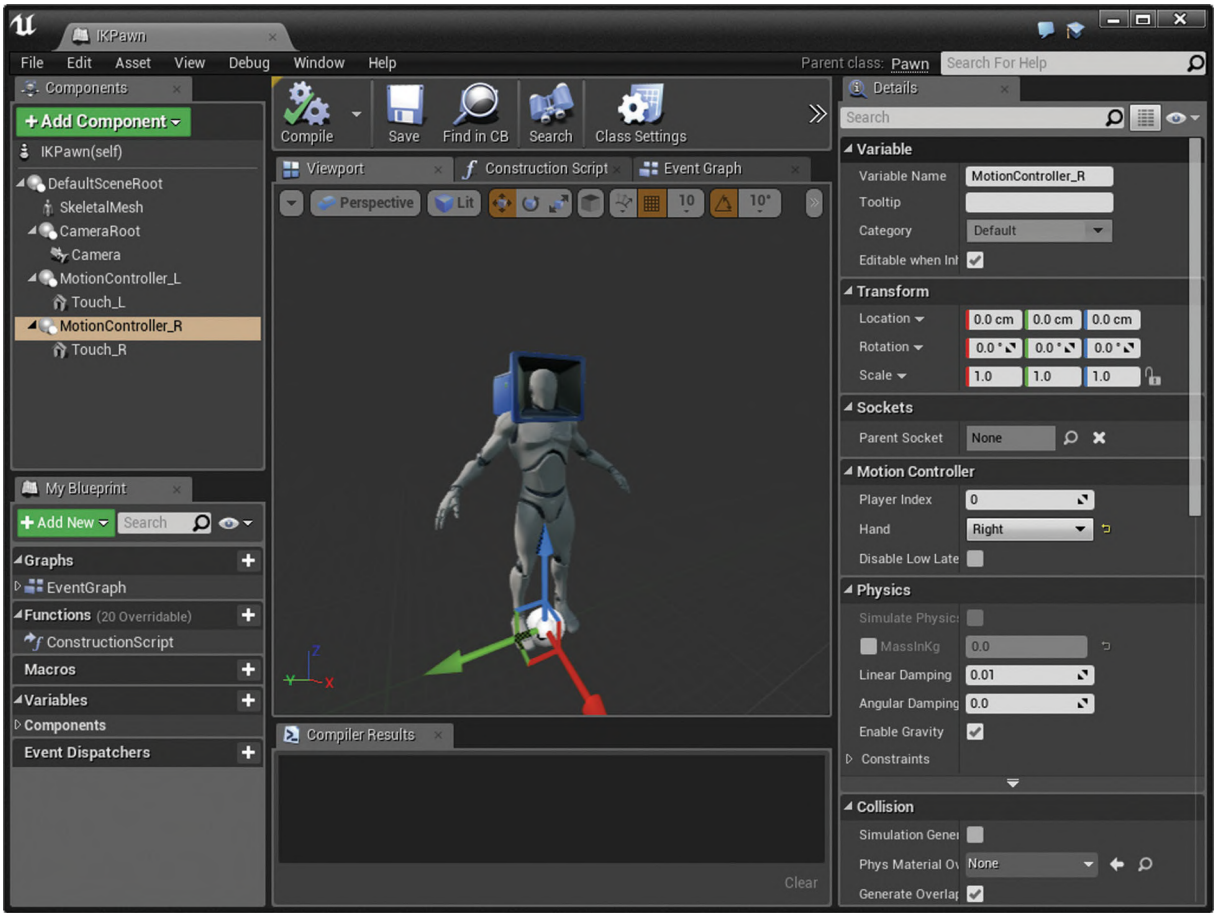

Рис. 7.12. Добавление контроллеров к *IKPawn*

7.3.2. Blueprint анимации инверсной кинематики руки

Ваш `IKAnimBP` уже реализует *FABRIK* для головы и позвоночника, поэтому сейчас пришло время для осуществления двухкостной инверсной кинематики для обеих рук вашего скелета.

1. Откройте `IKAnimBP` и создайте две новые переменные `LeftHandWorldPosition` и `LeftHandWorldRotation` типов `Vector` и `Rotator`. Для `Position` установите (`X = 40, Y = 20, Z = 100`). Это обеспечит, что меш соответствует позиции просмотра.
2. Сделайте то же самое для правой руки, только по `X = -40` (рис. 7.13).

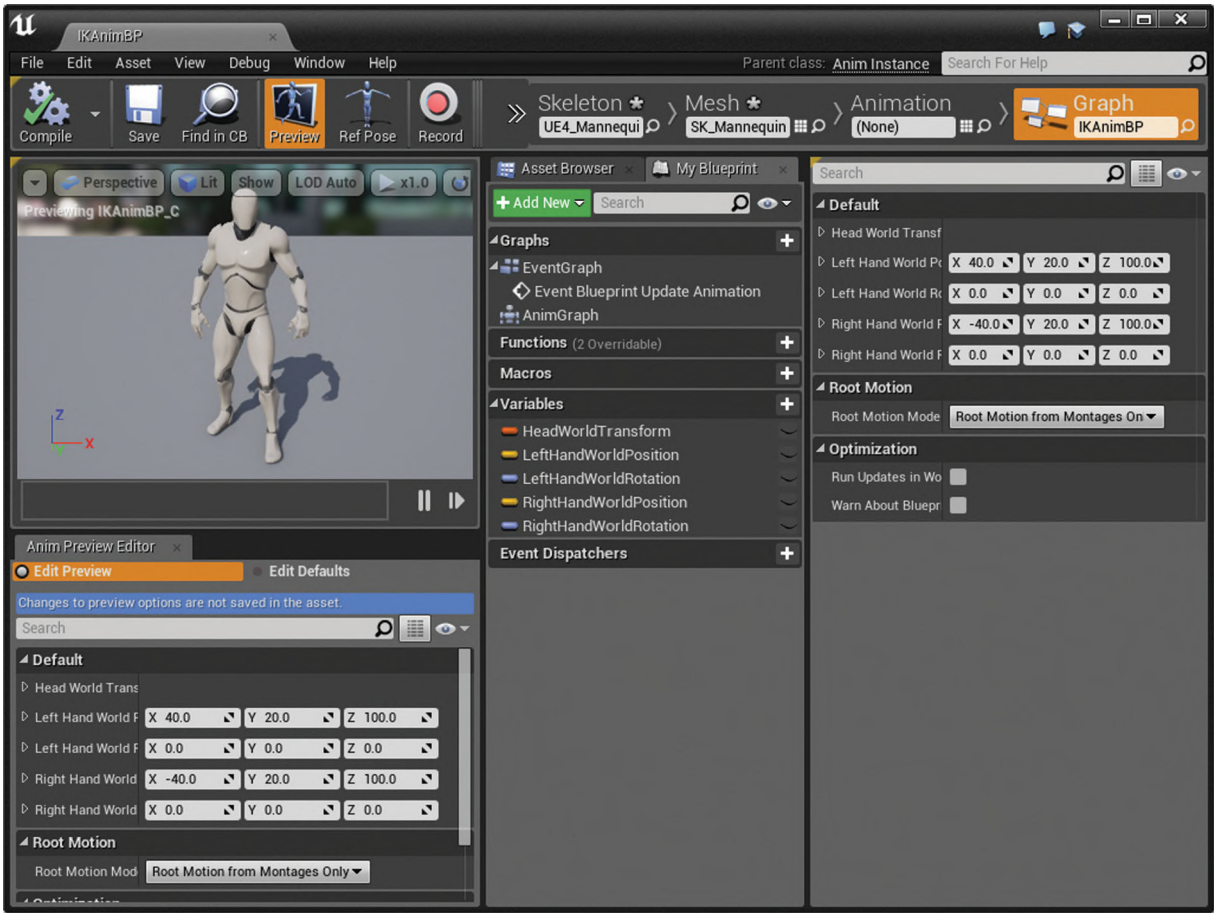

Рис. 7.13. *IKAnimBP*: добавление переменных

3. Перед началом настройки инверсной кинематики добавьте два новых *Socket* к вашему скелету. Это нужно, потому что оригинальная точка контроллеров не выравнивается с позицией кости руки. Это значит, что вам необходимо учитывать смещение, когда вы используете *IK* для получения позиции контроллера.
4. Откройте *Skeleton editor*, нажав на панель *Skeleton* в *Persona*.
5. Правым нажатием по `hand_L` в иерархии выберите *Add Socket*. Назовите его `hand_LSocket`.
6. Сейчас у *Socket* позиция родителя. Чтобы изменить это, выберите его и в панели измените `Relative Location` на `(X = 13, Y = -6, Z = -3.5)`. Это поместит *Socket* в ладонь руки.
7. Этот же *Socket* разверните по оси `X` на `180`.
8. Для проверки позиции *Socket*, правым нажатием по *Socket* добавьте *Preview Asset* вашего контроллера (рис. 7.14).

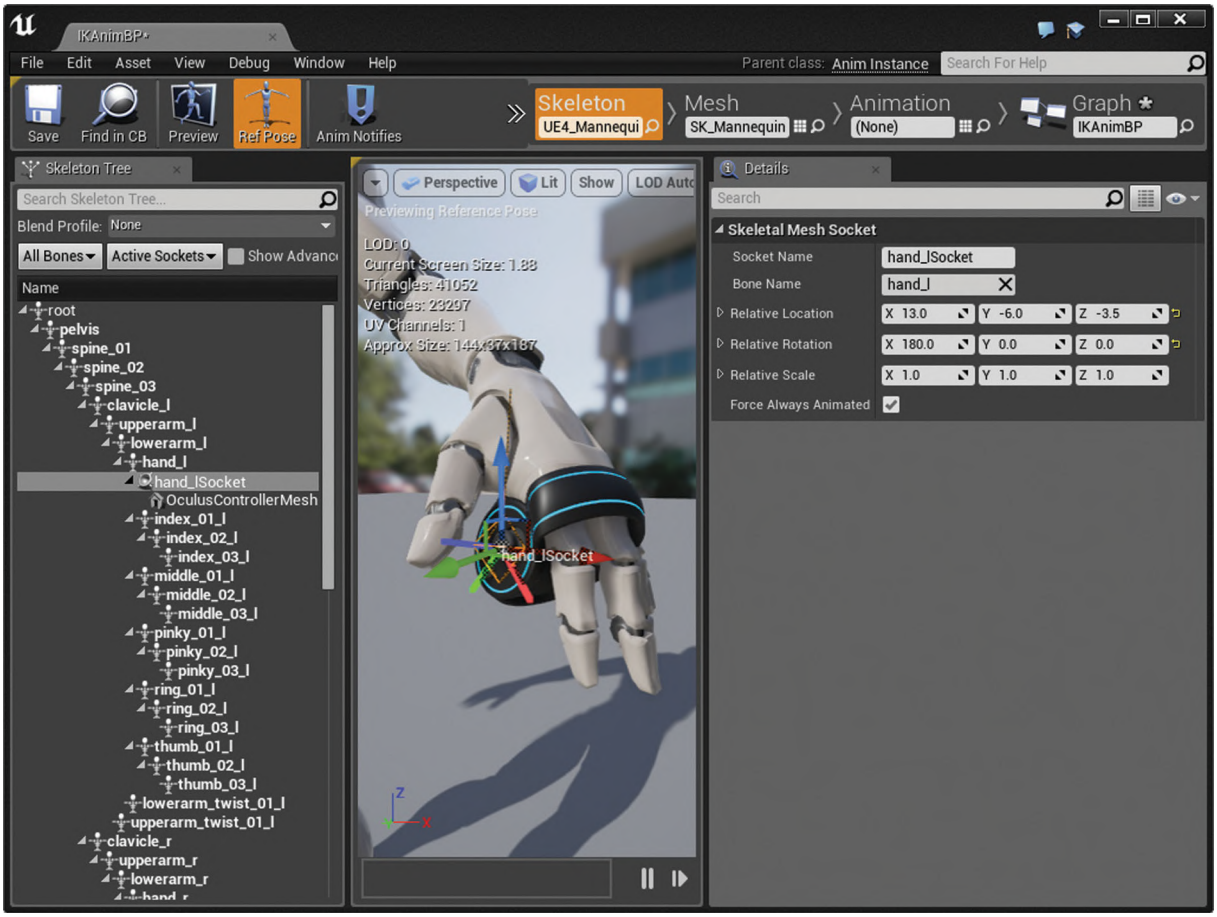

Рис. 7.14. *IKAnimBP*: добавление *Socket* для руки

9. Повторите шаги с 5 по 8 для правой руки с `Relative Location (X = -13, Y = 6, Z = 3.5)` и разворотом по оси `Z` на `180`.
10. Вернитесь к *Animation Blueprint*.
11. Создайте сеттер для позиции и поворота каждой руки.
12. От узла *CastToIKPawn* получите ссылки на `MotionController_L`, `MotionController_R`, `SkeletalMesh`.
13. От `MotionController_L`, вызовите `GetWorldRotation` и установите его в сеттер `LeftHandWorldRotation`.
14. Для *LeftHandWorldRotation* вам необходимо найти разницу между `Socket Location` и `location` кости. От `SkeletalMesh` вызовите `GetSocketLocation` дважды.
15. Для первого установите `hand_lSocket` в `In Socket Name`, во второй установите `hand_l` (рис. 7.15). Это позволит получить локацию кости и *Socket*.

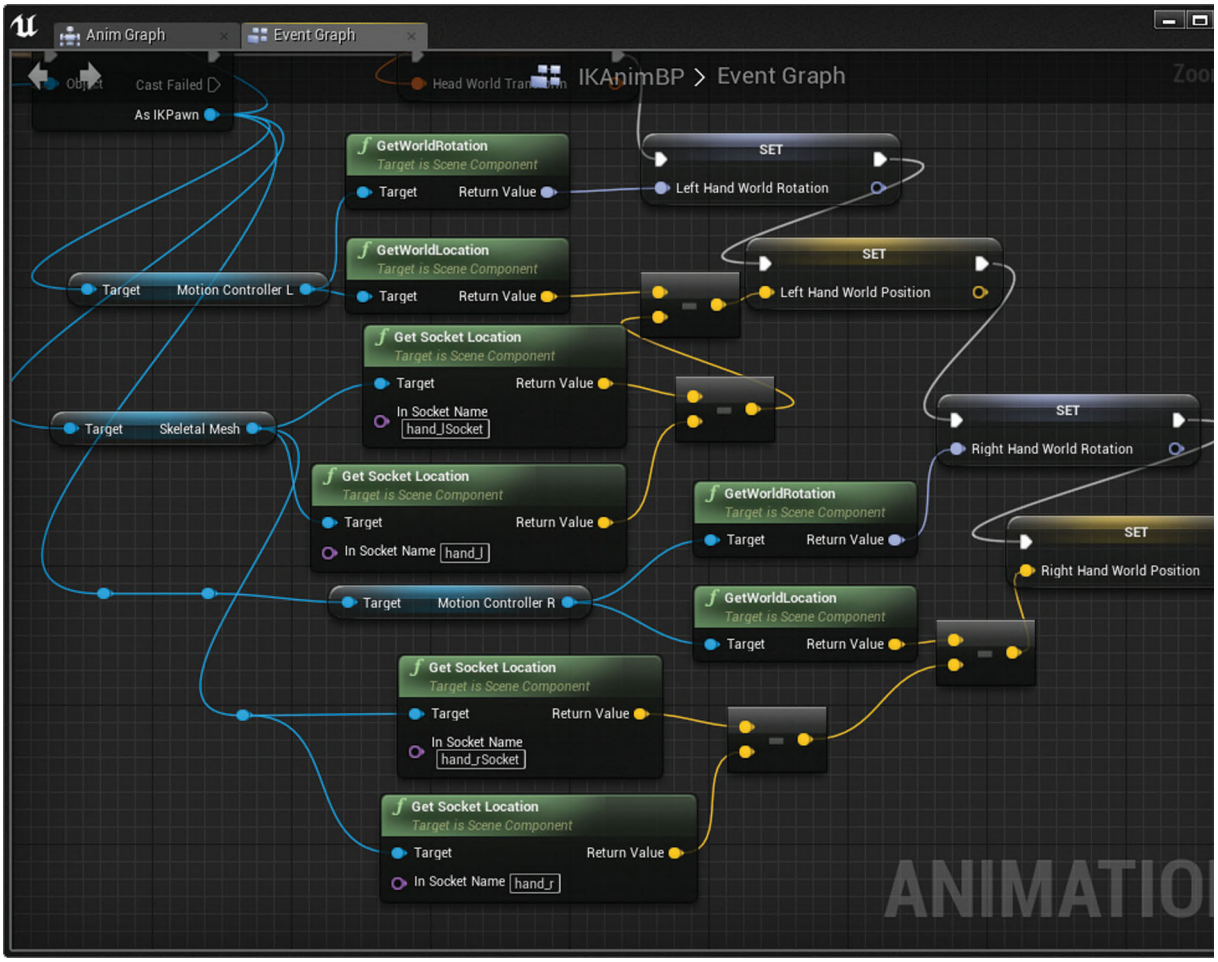

Рис. 7.15. *IKAnimBP*: настройка переменных и получение позиции

16. От `GetSocketLocation` вычтите позицию кости. Это даст вам дистанцию от руки до *Socket* в пространстве.
17. От `MotionController_L` вызовите `GetWorldLocation` и вычтите дистанцию, подсчитанную на шаге 16.
18. Соедините выход из узла разности с сеттером `LeftHandWorldPosition`.
19. Повторите шаги с 13 по 18, но с `MotionController_R` и `hand_r Socket` и костью `hand_r` (см. рис. 7.15).
20. Прикрепите сеттер к *HeadWorldTransform*, который уже есть в графе.

Для осуществления двухкостной *IK* вам необходимо выполнить следующие действия в *Anim Graph*.

1. Создайте узел *TwoBoneIK*.
2. Создайте геттер для `LeftHandWorldPosition` и прикрепите его к `Effector Location`.
3. Установите `(X = 45, Y = -50, Z = 100)` в `Join Target Location`. Можете поменять после при необходимости.

4. Выберите *TwoBoneIK* и измените `IKBone` на `hand_l` и `Effector Location Space` на `World Space` (рис. 7.16).

Рис. 7.16. *IKAnimBP*: двухкостная *IK* левой руки

5. Создайте новый узел *Transform Bone*, установив в свойстве `hand_l`, `Rotation Mode` на `Replace Existing` и `Rotation Space` на `World Space`. Это позволит вам поворачивать руку, потому что *IK* не сообщает об этом.
6. Создайте новый геттер для `LeftHandWorldRotation` и соедините его со входом `B` в узле *CombineRotator*.
7. Установите `180` в `X` входа `A` (см. рис. 7.16). Это значение получено просмотром кости скелета, как мы делали в секции *Setting Up Head IK*.
8. Соедините выход из *CombineRotators* со входом `Rotation` узла *Transform Bone*.
9. Повторите шаги 2–8, заменив `LeftHandWorldRotation` и `Position` с переменными правой руки и `(X = -45, Y = -50, Z = 100)` как `Join Target Location` и `180` градусами по `Z` (рис. 7.17).

Рис. 7.17. *IKAnimBP*: двухкостная *IK* правой руки

10. Соедините эти *IK* узлы между *Transform Bone IK* головы и компонента *Local Transform*, и вы можете играть в *VR* с полной системой *IK* (рис. 7.18).

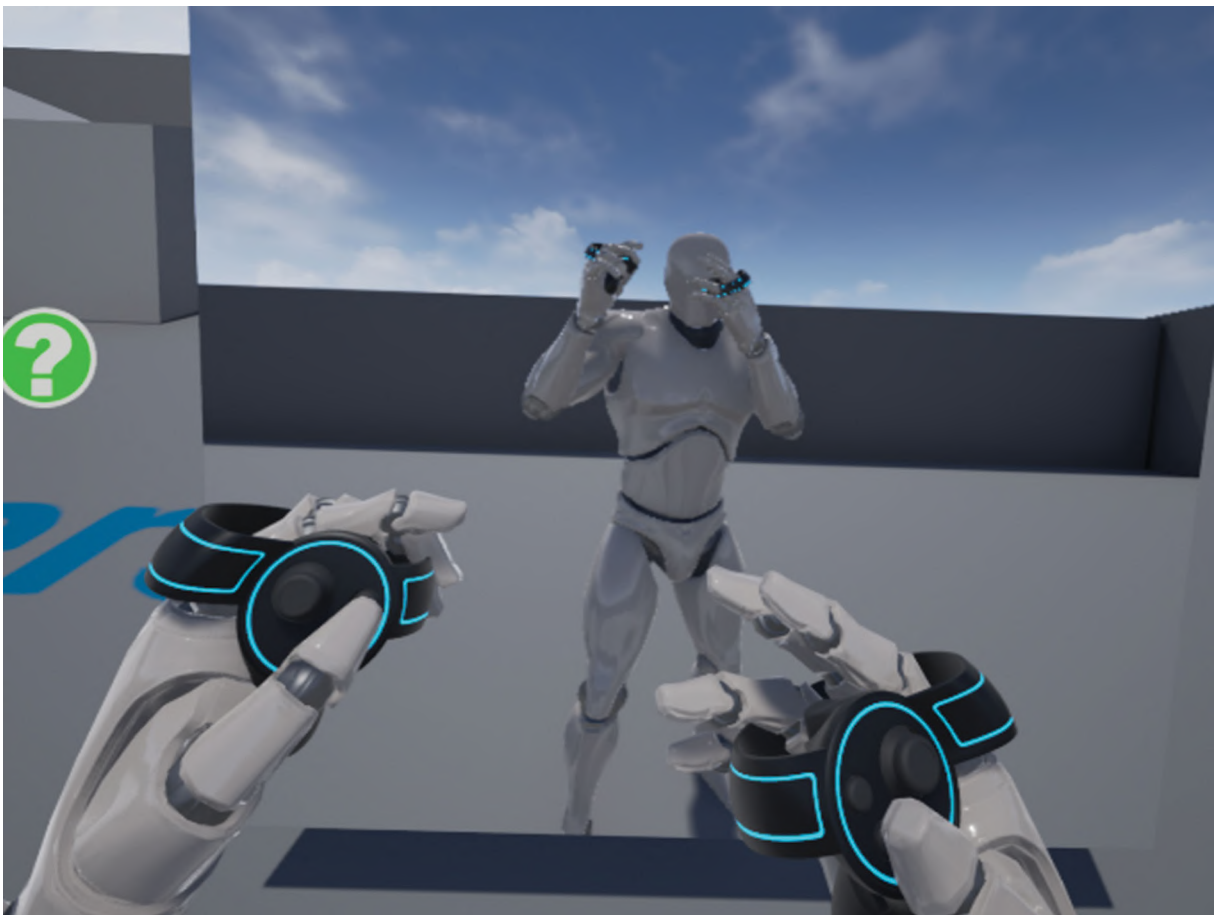

Рис. 7.18. Пример полной *IK* тела

7.4. Заключение

В этой главе вы увидели простой путь для создания инверсной кинематики (*IK*) в *VR*. Вы увидели два разных типа *IK* и создали простые системы *IK* головы и рук.

7.5. Упражнения

Теперь, когда у вас есть знания того, что может сделать *IK*, поэкспериментируйте со значениями, приведенными в этой главе, и настройте их в соответствии с вашим контентом.

Если вы чувствуете в себе достаточно креативности, исследуйте некоторые настройки *IK* для всего тела. Основная задача здесь состоит в том, чтобы найти эвристику, чтобы обнаружить, когда игрок сгибает ноги, а не просто наклоняется.

Если вы не авантюрист, но все же хотели бы полное *IK* решение, есть несколько компаний, которые обеспечивают полную *IK* тела для *UE*4 (например, *IKinema* или сообщество *Full Body IK Plugin*).

ГЛАВА 8

ВЗАИМОДЕЙСТВИЕ С КОНТРОЛЛЕРАМИ ДВИЖЕНИЯ

В этой главе показан модульный способ взаимодействия с объектами в игровом мире при помощи контроллеров движения. Независимо от того, собирает игрок предметы или бросает их, тянет рычаги, чтобы открыть дверь или нажимает кнопки, вызывая событие в игре, — эта глава научит вас создавать простую, легко расширяемую систему для широкого круга взаимодействий.

8.1. Движения

Когда вы погружаетесь в *VR*-сюжет с использованием шлема, возникающий эффект присутствия ошеломляет, но без устройства ввода, которое бы вписывалось в этот сюжет, часто кажется, что вы вошли в мир, управляемый по принципу «экспонаты трогать руками запрещается».

К счастью, контроллеры движения в большинстве случаев решают эту проблему; однако помните, что игроки, получившие такую свободу, сделали вашу работу значительно сложнее. Но используя подходящие системы, вы можете создать поразительно убедительные *VR*-сюжеты.

8.1.1. Почему работает взаимодействие с контроллерами движения?

Исторически сложилось, что в современных играх от первого лица индустрия сделала выбор в пользу парадигмы ввода, которая привязывает определенные события геймпада к игровому аватару. Геймпад с двумя аналоговыми джойстиками стал де-факто стандартом взаимодействия в играх от первого лица на многие годы.

В большинстве случаев вы можете напрямую сопоставить то, что видит аватар с правым джойстиком, ноги игрока с левым джойстиком, а действия рук аватара — с кнопками на геймпаде.

В *VR*, тем не менее, описанная парадигма не работает. Правый джойстик выводится из взаимодействия, что позволяет пользователю посредствам движения головы контролировать взгляд аватара. Аватар большую часть времени теперь не перемещается посредством джойстика, потому что такое управление может вызвать физический дискомфорт (чтобы узнать больше, см. главу 9 «Перемещение в *VR*»). Наконец, многие поставщики *VR*-оборудования предлагают точно отслеживаемые контроллеры движения, поэтому теперь у вас есть способ физически представить руки игрока в вашем сюжете и отпадает необходимость в абстрактных системах взаимодействия, которые используют кнопки на геймпаде. Контроллеры движения позволяют нашему игроку физически взаимодействовать с объектами игрового мира.

8.1.2. На что обратить внимание: важность доступности

Разумеется, несоответствие ожиданиям игрока всегда плохо для игры. В *VR*-сюжете провал ожиданий игрока к тому же полностью разрушает погружение. Когда игрок протягивает руку, чтобы открыть ящик, а он в действительности оказывается статическим объектом — это вызывает разочарование. Невероятное погружение в *VR* позволяет игрокам ощутить доступность окружающих их объектов, как если бы они существовали на самом деле. Чтобы сделать хороший сюжет, мы должны стараться не разрушать эти представления.

Доступность можно представить как свойство объекта, говорящее о возможности его использования. Например, если в вашем *VR*-сюжете есть дверь, на которой размещена ручка, для игрока это означает, что дверь может быть открыта. Если подобного рода ожидания не оправдываются, разрушается эффект погружения, это означает, что о доступности обязательно надо позаботиться на этапе проектирования вашего *VR*-мира.

8.1.3. Общие выходные данные текущего поколения контроллеров движения

При создании взаимодействий при помощи контроллера движения познакомьтесь с платформой, для которой ведется разработка, чтобы быть уверенными, что вы полностью реализуете доступные возможности. Каждый из предложенных на рынке контроллеров движения имеет слегка различные возможности и рекомендации по эксплуатации. К счастью, три из основных *VR*-компаний, нацеленные на широкого потребителя (*HTC/Valve*, *Oculus* и *PlayStation*) придерживаются в чем-то сходных парадигм организации ввода, то есть вы можете воспользоваться всеми возможными преимуществами кросс-платформенной совместимости.

Все три контроллера имеют кнопку под указательный палец («спусковой крючок», или триггер), позволяющую организовать некоторые виды аналогового ввода (рис. 8.1). Триггер может быть полезен для активации объектов или стрельбы из оружия в играх. Кроме того, как *Oculus Touch*, так и *Vive*-контроллеры имеют кнопку захвата, которая полезна как специализированная кнопка для подбора объектов.

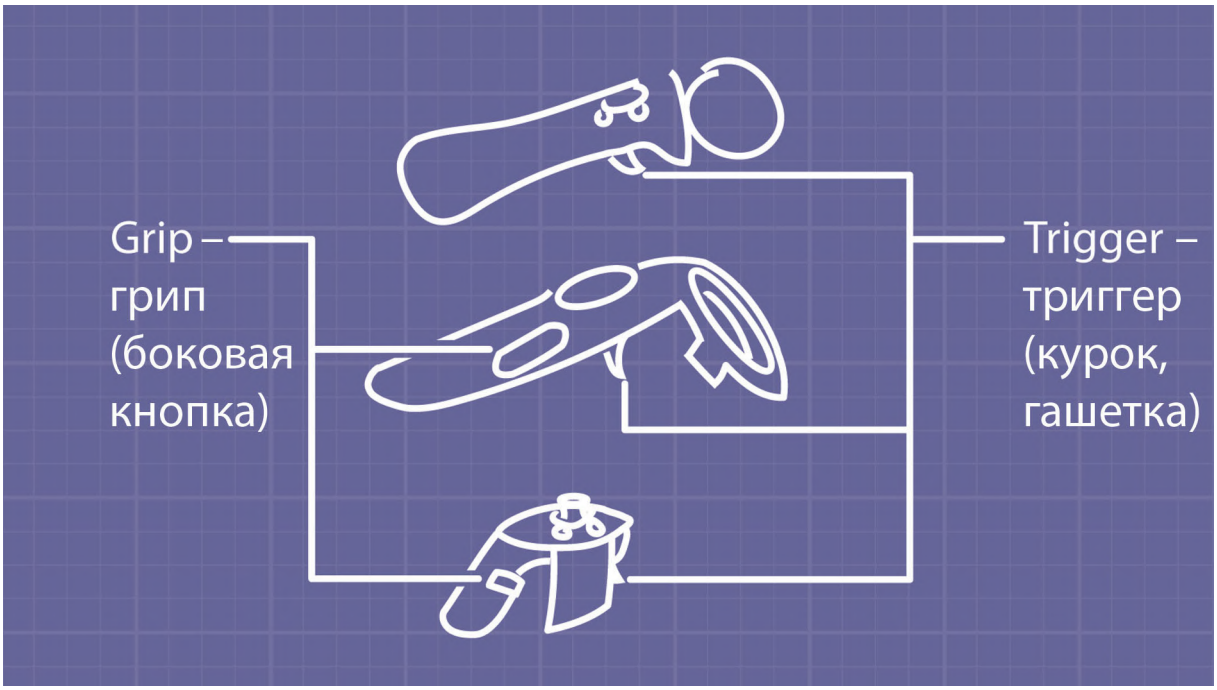

Рис. 8.1. Общие черты функционала *VR*-контроллеров: вверху: *PlayStation Move*; в центре: *Vive*; внизу: *Oculus Touch*

В то же время на рынок выходят новые контроллеры движения (например, контроллер *Google Daydream*). Поэтому в этой главе мы сосредоточимся на создании системы, работающей с *Vive* и *Oculus Touch*. Но высокий уровень абстракции, который предлагает разработчикам *UE*4 (все эти контроллеры используют один и тот же *Motion Controller Component*), дает вам возможность легко адаптировать методы этой главы к любому другому устройству.

8.1.4. Настройка проекта взаимодействия с миром

Для настройки проекта нам понадобятся следующие объекты.

1. Создайте пустой *Blueprint* проект без стартового контента.
2. Создайте папку *Blueprints*, а в ней еще три папки *Interface*, *InteractiveObjects*, *Components*.
3. В папке *Components* создайте новый *Scene Component* под названием `WordInteractor` (*Add New* ⇒ *Blueprint Class* ⇒ *Scene Component*).
4. В папке *Blueprints* создайте новый *Pawn* под названием `InteractionPawn`.
5. Откройте его и добавьте новый *Scene Component* под названием `CameraRoot`.
6. Создайте *Camera Component* и прикрепите его к последнему созданному компоненту.
7. Создайте *MotionController Component*.
8. Создайте *Sphere Component* и прикрепите его к *MotionController Component*.
9. Измените масштаб сферы на `(X = 0.1, Y = 0.1, Z = 0.1)`.
10. Выберите сферу и измените `Collision Present` на `OverlapAllDynamic` (рис. 8.2). Это предотвратит сферу от взаимодействия с объектами, которые вы будете поднимать.

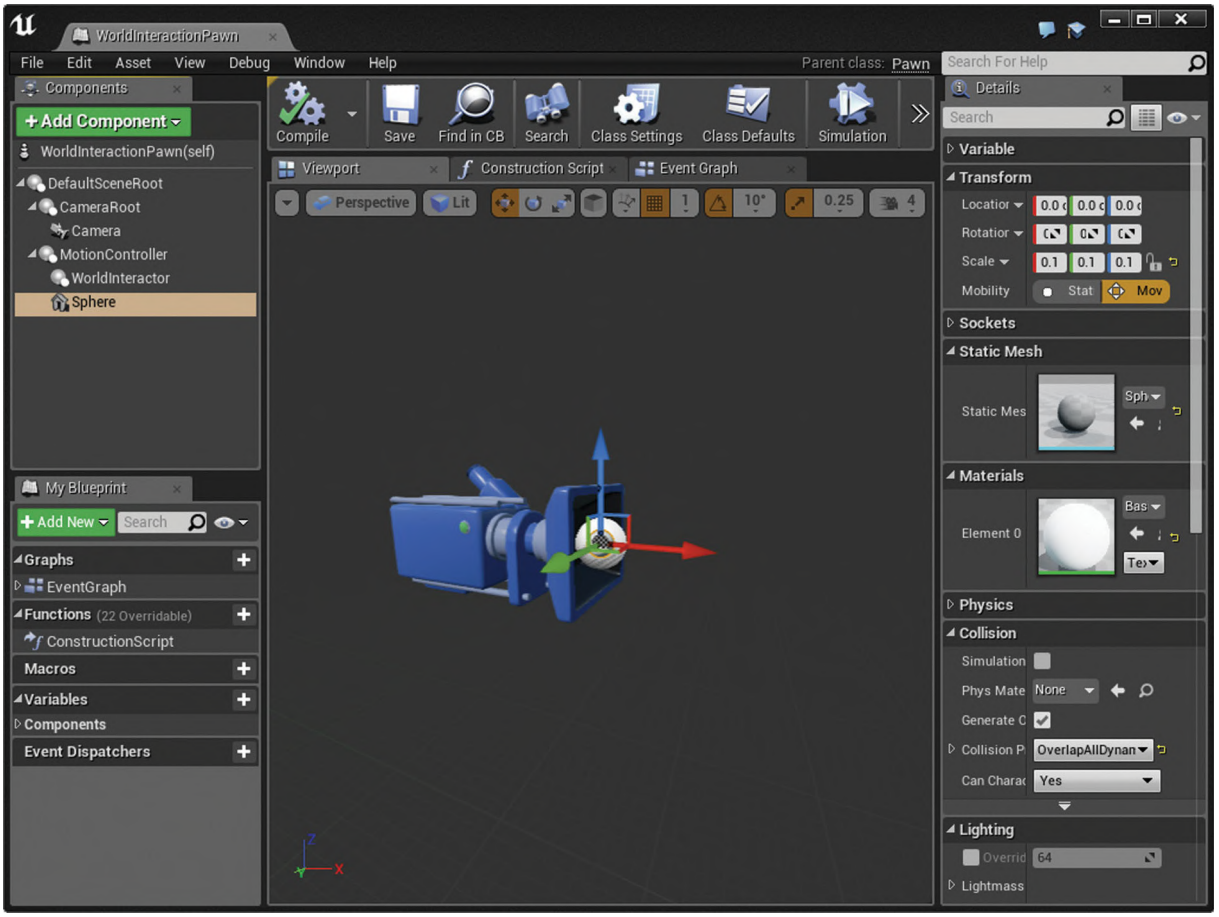

Рис. 8.2. *World Interaction Pawn Component*

11. Создайте новый *WorldINteraction Componnet* и прикрепите его к *MotionController Component*.

> **Заметка**
>
> *Motion Controller Component* по умолчанию направлен на левый контроллер. Чтобы это изменить, на панели деталей измените свойство *Hand*.

8.1.5. Взаимодействие с объектами

Чтобы взаимодействовать с объектами, вам нужны две вещи: интерфейс (см. главу 4), который позволяет говорить объектам, что вы взаимодействуете с ними, и компонент для определения и управления этими взаимодействия.

8.1.5.1. Создание интерфейса World Interaction

Интерфейс позволит вам общаться с объектами для осуществления необходимого вам функционала без знания, что это за объект (см. главу 4). Это сделает их пригодными для системы взаимодействия, потому что посреднику (интерактору) не следует беспокоиться, какую кнопку нужно нажать, чтобы изменить цвет. Посреднику нужно только сказать, что ему необходимо взаимодействовать с чем-то, а потом вызвать взаимодействие на требуемом объекте.

1. В папке *Interface* создайте новый интерфейс (в меню выберите *Add New* ⇒ *Blueprint* ⇒ *Blueprint Interface*). Назовите его `TraceInteractionInterface`.
2. Каждая функция должна иметь одну входную переменную `Interactor` типа `WorldInteractor` и одну выходную `Return` типа `Boolean`. Вы не будете использовать значение `Return`, но вы должны позволить всем функциям находиться в одном разделе вместе с реализующимся объектом.
3. Создайте `OnHover` и `OnUnhover` функции с переменными шага 2 (рис. 8.3). Эти функции будут вызываться каждый раз, когда игрок наводит курсор на объект.

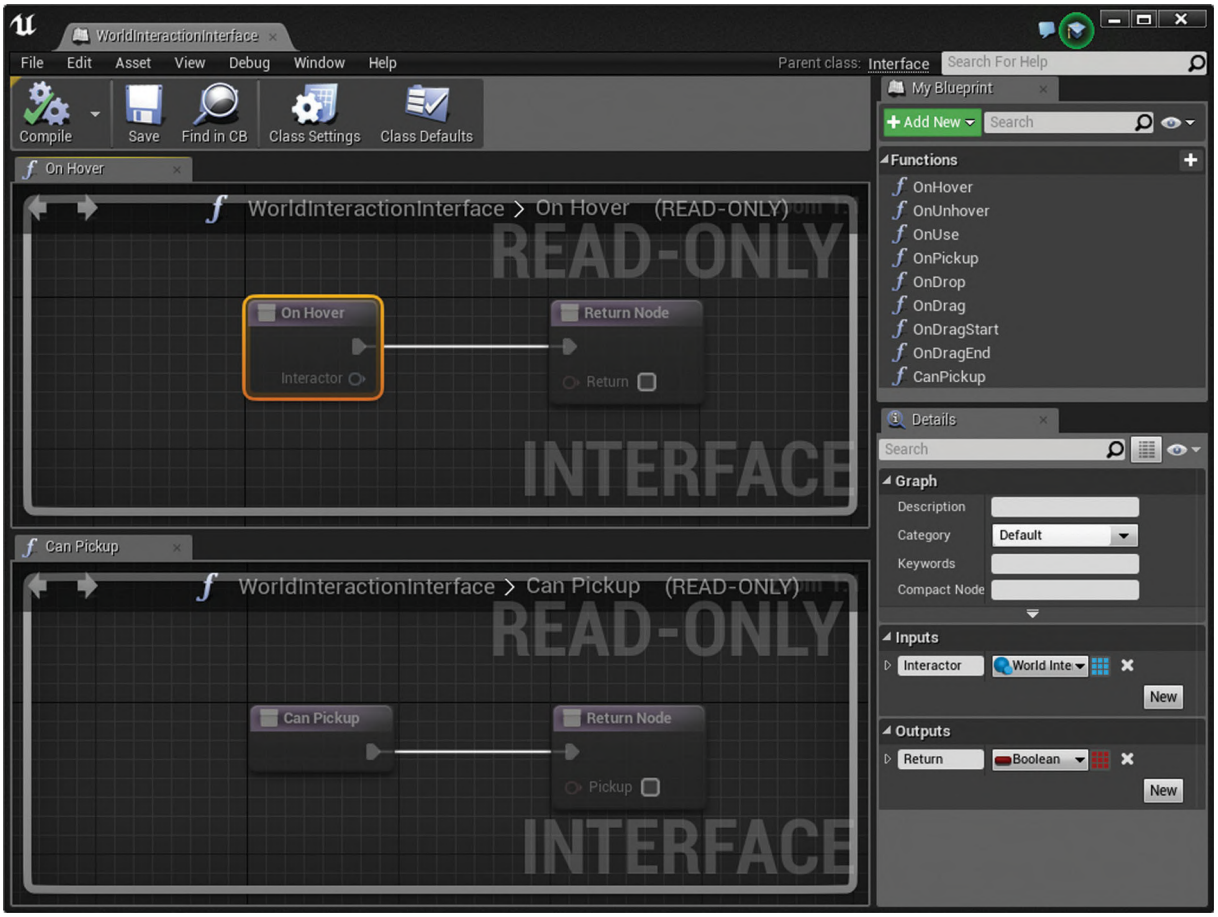

Рис. 8.3. Интерфейс *World Interaction*

4. Создайте `OnUse`-функцию. Она вызывается, когда игрок выбирает объект, на который наведен курсор.
5. Создайте две функции, `OnPickup` и `OnDrop`, для определения, когда выбирается объект с такими же переменными пункта 2.
6. Создайте три новые функции, `OnDrag`, `OnDragStart`, `OnDragEnd`, которые позволяют вам определять, когда игрок активирует объект и двигает контроллер во время активации.
7. Создайте функцию `CanPickup`. Однако дайте ей только выходную переменную под названием `Pickup` (рис. 8.3).

8.1.5.2. Создание Interactor Component

Для взаимодействия вам необходимо что-то для хранения логики нахождения объекта взаимодействия и выбора функций из интерфейса.

8.1.5.2.1. Определение соприкосновений объекта

Сначала вам необходимо создать переменные для запоминания наведенных объектов и настроить соприкосновение сферы для обнаружения объектов вокруг интерактора.

1. Откройте *WorldInteractor*.
2. Создайте переменную `Radius` типа `Float` и установите значение `10`. Это радиус сферы, которую создает интерактор для определения объектов.
3. Создайте новую ссылку на *Actor* под названием `HoveredObject`. Она ссылается на объект, который сейчас под курсором.
4. Создайте другую ссылку на *Actor* под названием `PickedUpObject`.
5. Создайте новую переменную `Dragging` типа `Boolean` (рис. 8.4).

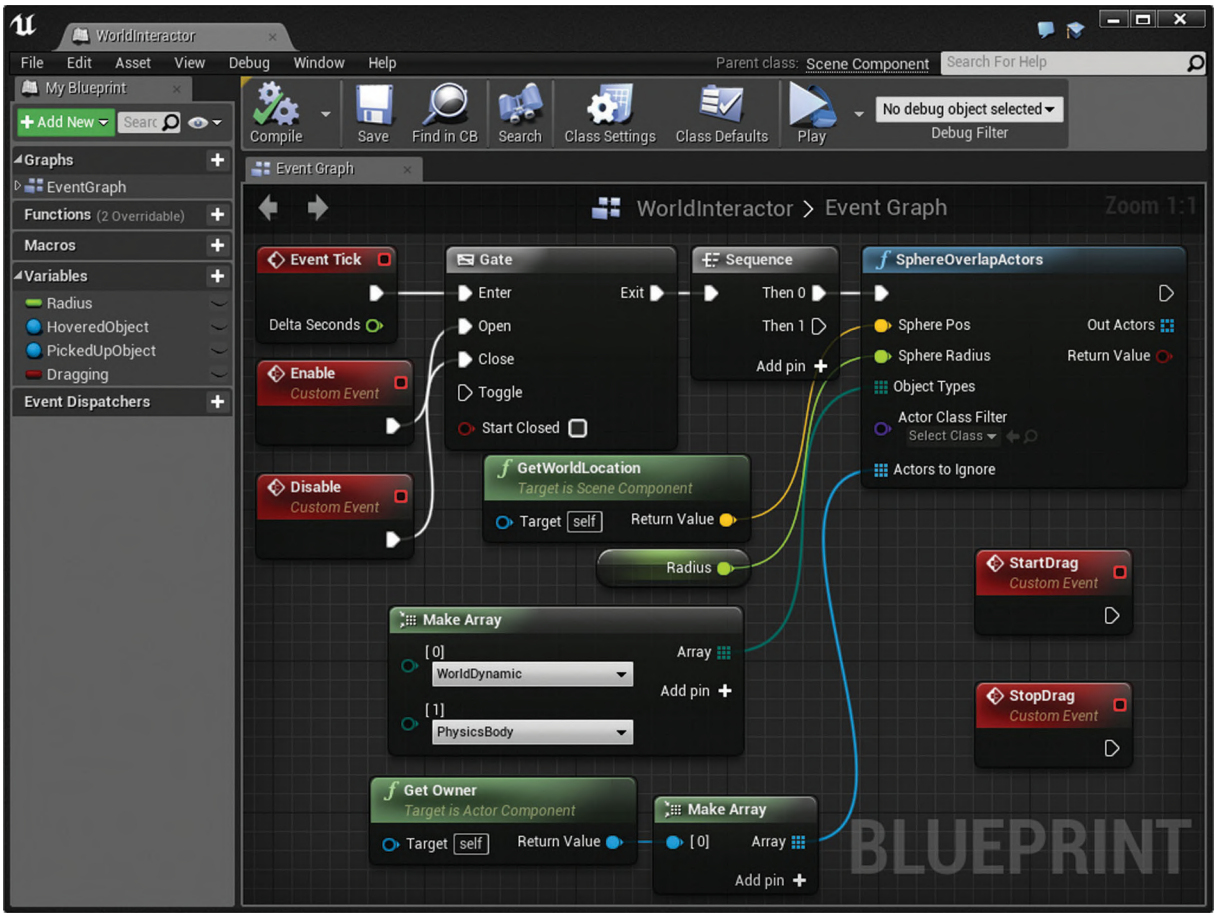

Рис. 8.4. Компонент *WorldInteractor*: базовое соприкосновение сферы

6. От `EventTick` создайте новый узел *Gate*, и отключите `Start Closed`, потому что вы хотите, чтобы компонент был доступен по умолчанию.
7. Добавьте два новых события: `Enable` и `Disable`.
8. Соедините событие `Enable` с `Open` узла `Gate`, а `Disable` — с `Close`. Это позволит вам активировать и деактивировать при необходимости.
9. Из `Exit` узла *Gate* создайте узел *Sequence*, потому что вам надо совершать два действия за ход, и это сделает код аккуратнее.

10. Соедините первый выход с новым узлом *SphereOverlapActors* (рис. 8.4). Это позволит вам определять объекты для взаимодействия по заданному радиусу.
11. Соедините `GetWorldLocation` с входом `Sphere Pos`.
12. Соедините новый геттер `Radius` с `Sphere Radius`.
13. Для `Object Types` создайте массив перечислений `EObjectTypeQuery`, добавив `WorldDynamic` и `PhysicsBody`.
14. Создайте массив `Actor`, присоединив его к `GetOwner` первым входом и прикрепите выход с `Actors` к пину `Ignore` перекрытия сферы.
15. Создайте два новых события `StartDrag` и `Stop Drag` (рис. 8.4). Они будут использованы позже.

АЛЬТЕРНАТИВА ПЕРЕКРЫТИЮ СФЕРЫ

Есть много способов понять намерения пользователя в процессе определения того, какой объект он хочет захватить.

Сфера дополняется обратной связью, визуальной (подсветка объекта) или физической (тактильный импульс), сигнализирующей, что игрок может подобрать объект, чего во многих случаях вполне достаточно для формирования этого намерения. Но в случае, если так не получается или обратная связь отсутствует, возможны альтернативы.

Одину из них *Epic Games* реализует как *Bullet Train*, создает вокруг контроллера сферу с исходящим вектором, направленным между векторами направления контроллера и вектором от шлема к контроллеру. Это обеспечивает хороший баланс между гибкостью и помощью игроку в подборе предметов.

Конечно, для решения данной задачи серебряной пули не существует, как и в большинстве других ситуаций с игровым дизайном, вы должны проверить, какая система взаимодействия лучше всего подходит вашему сюжету.

8.1.5.2.2. Вызов сообщения о наведении

Теперь вам нужно взять *Actor* из сферы соприкосновения и найти в нем объект для вызова логики наведения. В примере вы просто используете первый элемент в массиве; однако другие варианты, такие как нахождение ближайшего объекта из массива, тоже могут быть использованы.

1. От контакта `Out Actors` узла *SphereOverlapActors* вызовите узел *Get* и получите элемент `0`.
2. Создайте новый геттер для переменной `HoveredObject` и вызовите `NotEquals`, установив узел *Get* во второй контакт.
3. После узла *SphereOverlapActors* создайте ветку *Branch* и соедините *Conditions* с узлом *NotEquals*.
4. Вызовите сообщение `OnUnhover` и укажите в `Target` геттер переменной `Hovered Object`. Соедините вызов с веткой `True`. Это снимет наведение с предыдущего компонента.
5. От узла *OnUnhover* вызовите событие `StopDrag`; это остановит перетаскивание объекта, когда он будет слишком далеко.

6. После этого события создайте сеттер для переменной `HoveredObject`, поместив в него объект, полученный узлом *Get* из массива (рис. 8.5).

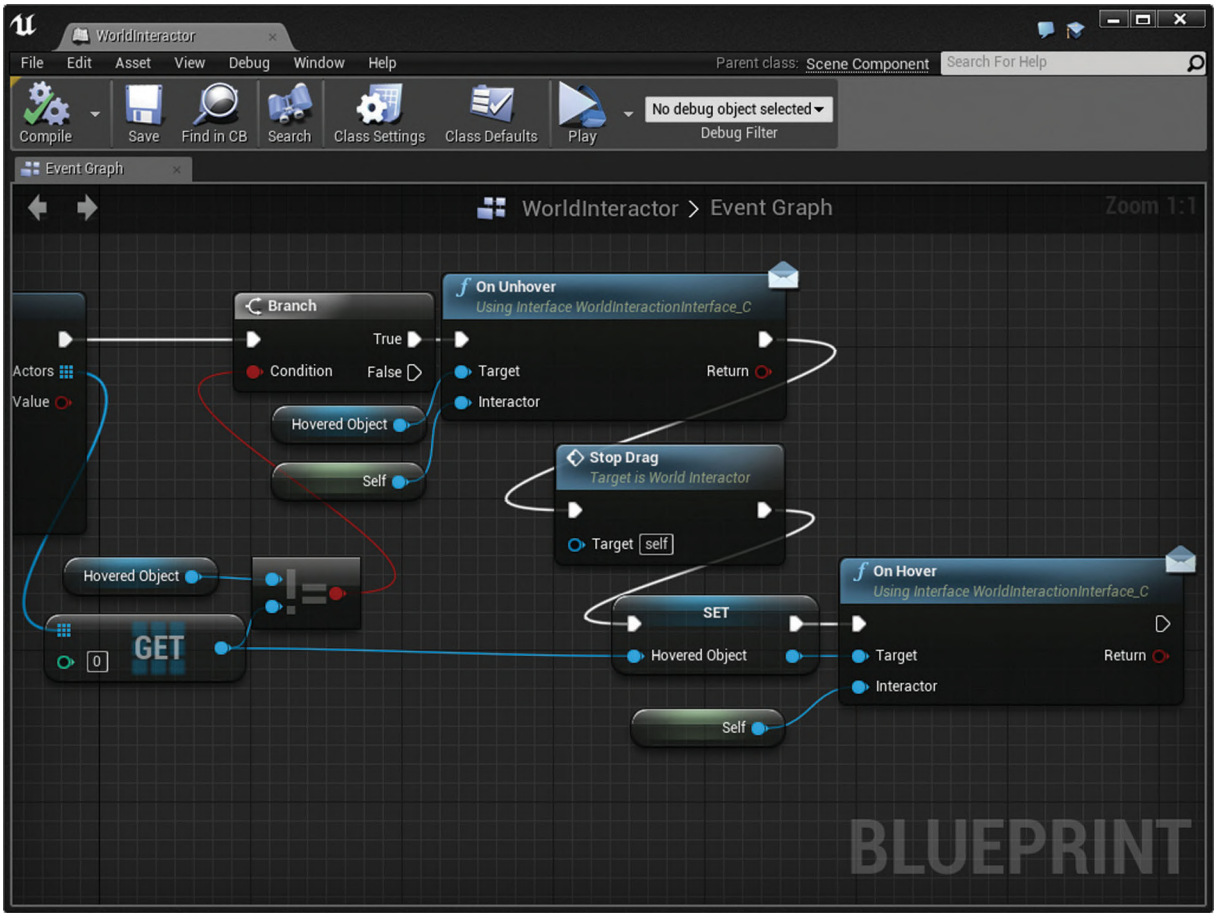

Рис. 8.5. Компонент *WorldInteractor*: наведение на объект соприкосновения

7. От сеттера вызовите `OnHover`, добавив в `Interactor Self` (рис. 8.5).

8.1.5.2.3. Реализация перетаскивания на объект

Теперь возьмем событие перетаскивания, созданное на первом шаге, и передадим его на текущий наведенный объект.

1. От свободного контакта узла *Sequence* добавьте другой узел *Gate*, соединив его с `Enter`. Это позволит вам активировать и деактивировать перемещение, когда это нужно.
2. От события `StartDrag` создайте сеттер для переменной `Dragging` и установите в нем `True`.
3. Создайте геттер для `HoveredObject`.
4. От геттера вызовите сообщение интерфейса `OnDragStart`, соединив контакт выполнения с сеттером (рис. 8.6)

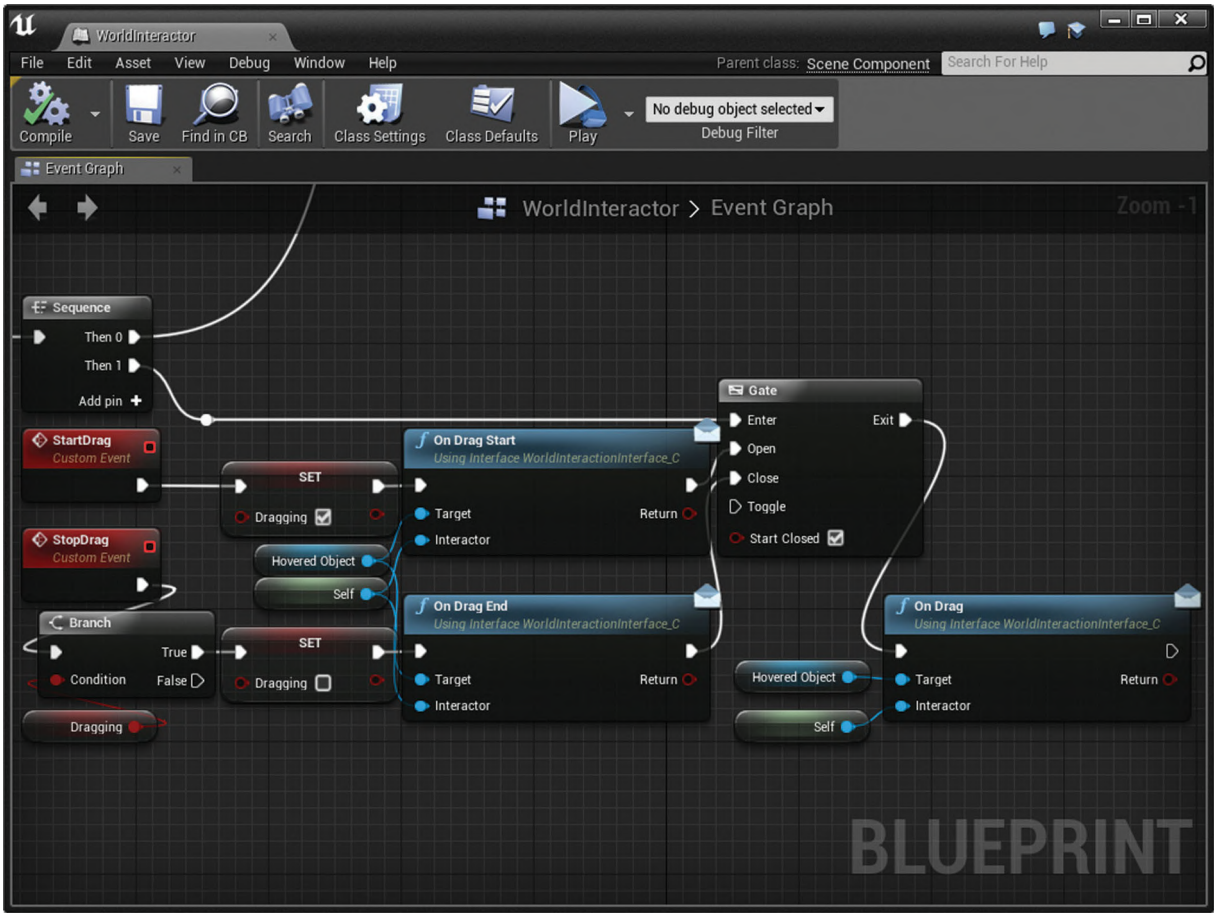

Рис. 8.6. Компонент *WorldInteractor*: обработка событий перемещения

5. Соедините выход обработчика `OnDragStart` с контактом `Open` узла *Gate*.
6. От события `StopDrag` создайте *Branch*. Для условия создайте геттер узла *Dragging*. Это обеспечит остановку перемещения только тогда, когда он уже передвинулся.
7. После ветки `True` создайте сеттер для *Dragging* и установите его на `False`.
8. После сеттера вызовите сообщение `OnDragEnd` на геттер `HoveredObject` из шага 3.
9. Соедините выход `OnDragEnd` с `Close` узла *Gate*.
10. После `Exit` вызовите сообщение `OnDrag` на `HoveredObject`, установив `Self` в `Interaction` (см. рис. 8.6)

8.1.5.2.4. Поднимаем и роняем объекты

Поднимать и ронять предметы в этом примере легко, потому что вы оставляете обработку физики и другие возможности за отдельными интерактивными объектами.

1. Создайте два новых узла событий и назовите их `Pick Up` и `Drop`. Владелец *Interactor* вызовет эти события для выбора текущего объекта.
2. От события `Pick Up` вызовите `IsValid`, указав во входном объекте геттер `HoveredObject`. Это гарантирует, что *Interactor* свяжется именно с подобранным объектом.

3. От этого же геттер вызовите сообщение `CanPickup`, соединив выполнения с выходом `Is Valid`.
4. После этого сообщения добавьте *Branch*, соединив вход *Condition* с выходом *Pickup*.
5. Для выхода `True`, связанного с *Branch*, создайте новый сеттер для переменной `PickedUpObject`. На вход этого сеттера подайте вновь созданный геттер `HoveredObject` (рис. 8.7).

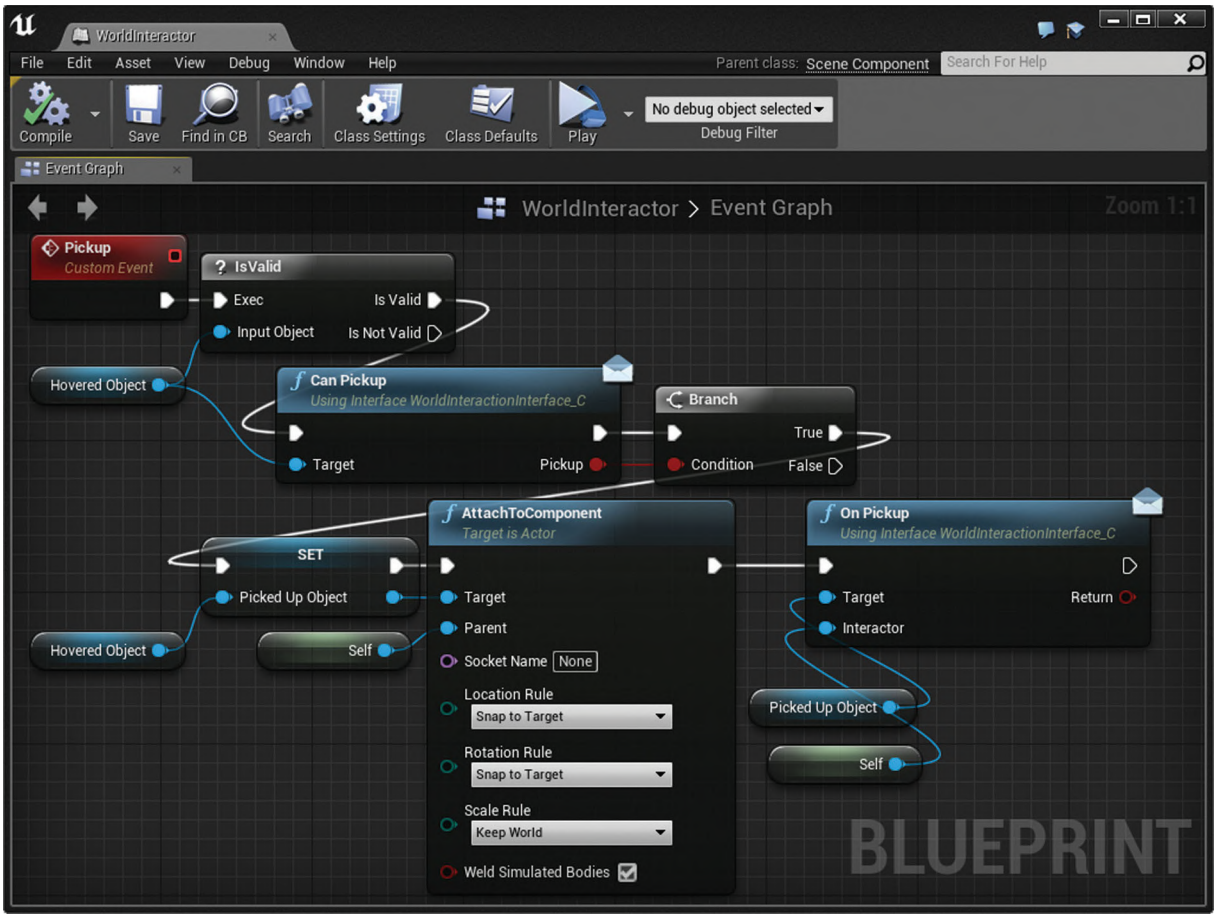

Рис. 8.7. Компонент *WorldInteractor*: выбор объекта

6. Свяжите голубой выход сеттера с новым узлом *AttachToComponent*.
7. Создайте ссылку `Self` и соедините ее со входом `Parent`.
8. Для обоих входов `Location Rule` и `Rotation Rule` установите значение `Snap To Target`.
9. Установите `Scale Rule` в значение `Keep World`, потому мы не хотим, чтобы размеры поднятого объекта подверглись воздействию *Pawn* или *Interactor*.
10. После *AttachToComponent* добавьте `CallPickup` на геттер `Picked Up Object` (рис. 8.7).
11. От события `Drop` вызовите `IsValid` и свяжите вход `Input Object` с новой переменной `PickedUpObject`.

12. От контакта `IsValid` вызовите `DetachFromActor`, установив новый геттер `PickedUpObject` как `Target`, и установите все на `Keep World` (рис. 8.8).

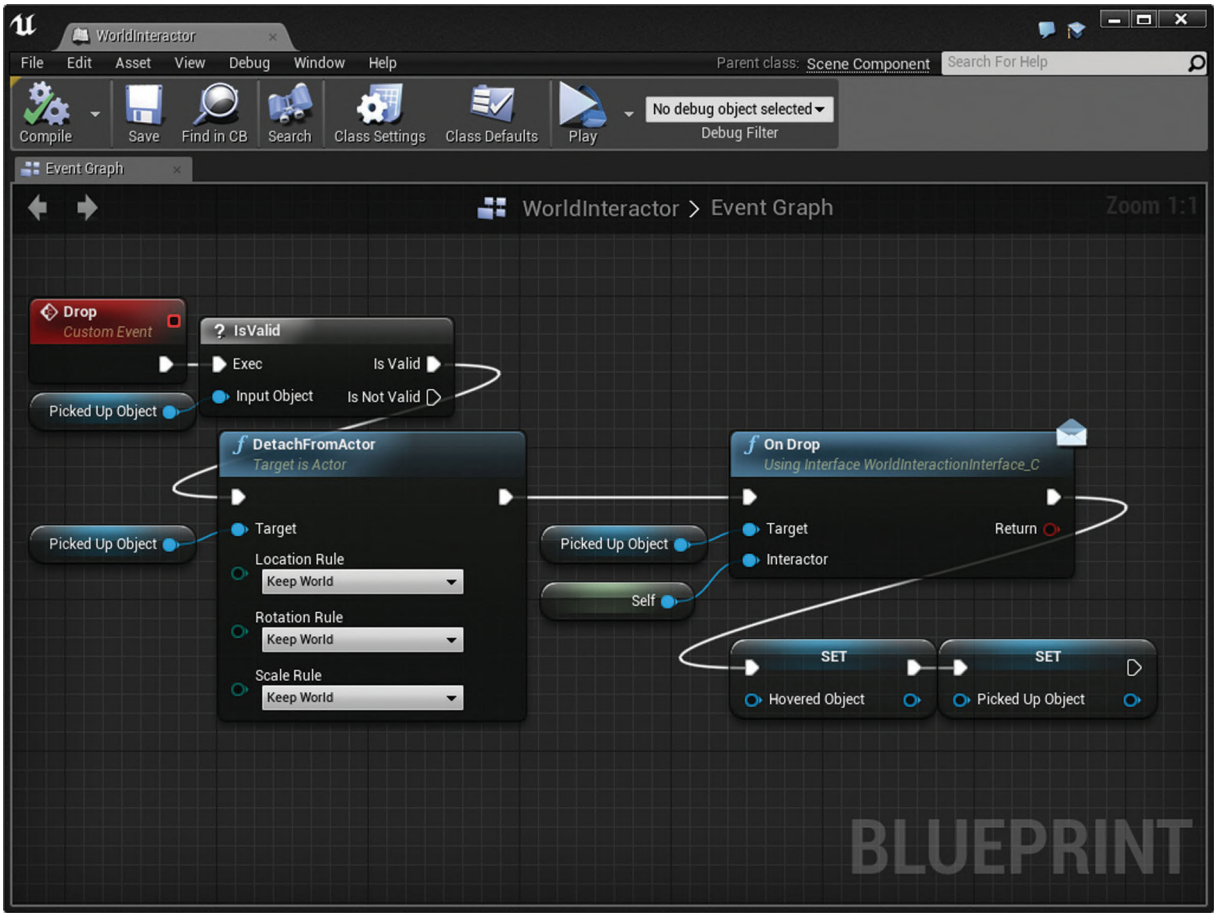

Рис. 8.8. Компонент *WorldInteractor*: бросок объекта

13. После узла *DetachFromActor* вызовите сообщение `OnDrop` на новом геттере `PickedUpObject`, установив `Self` в *Interaction*.
14. Теперь очистим `HoveredObject` и `PickedUpObeject` вызвав к ним пустые сеттеры (см. рис. 8.8).

8.1.5.2.5. Использование выбранных объектов

Добавление функциональности для использования выбранных объектов может быть полезно, если вы хотите сделать кнопки и другие объекты, которые могут быть использованы, когда игрок их выбирает.

1. Создайте новое событие под названием `UseHoveredObject`.
2. От события вызовите сообщение интерфейса `OnUse`, установив геттер `HoveredObject` в `Target` и `Self` в *Interactors* (рис. 8.9).

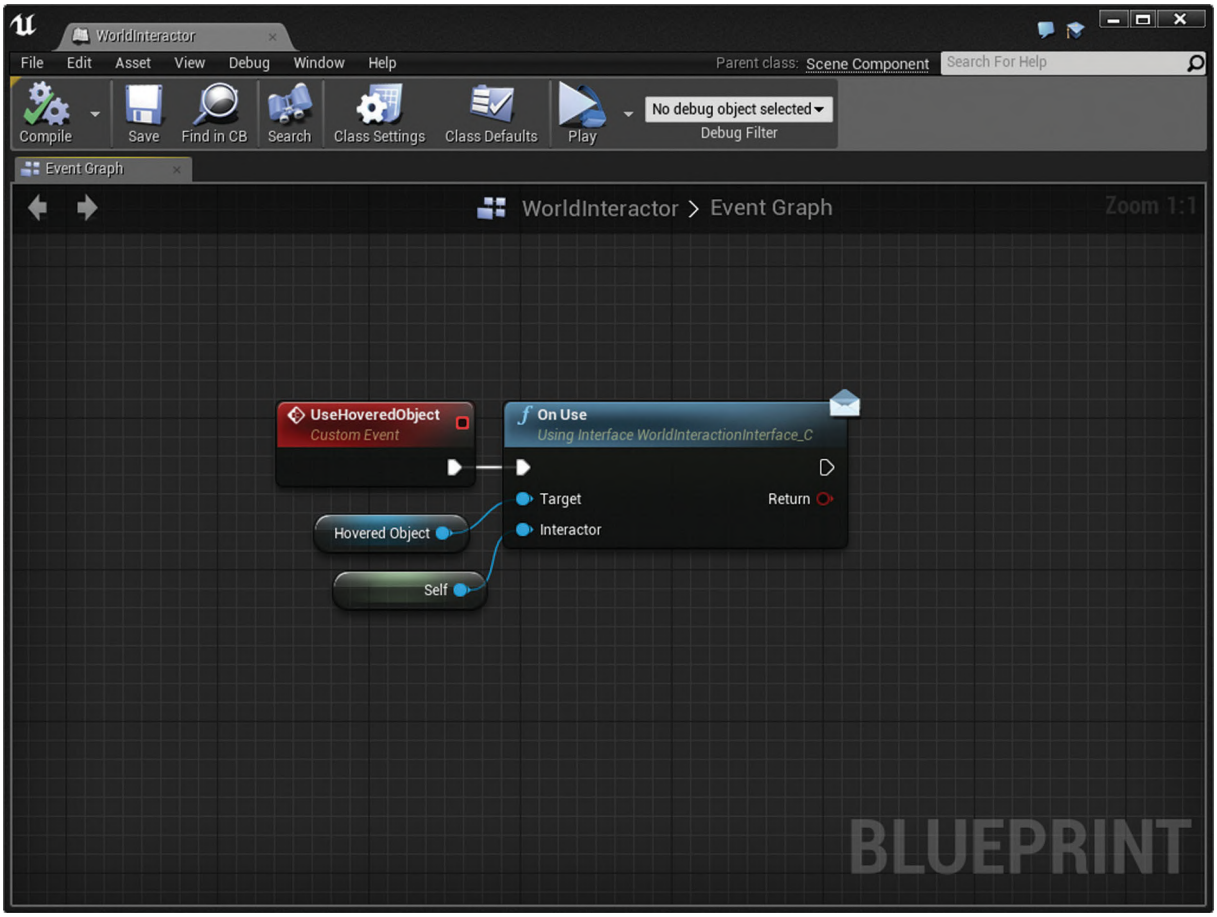

Рис. 8.9. Компонент *WorldInteractor*: использование выбранных объектов

8.1.5.3. Добавление взаимодействия к взаимодействующему Pawn

Для использования недавно созданного *Interaction Component*, вам необходимо вызвать некоторые недавно созданные события из *Interaction Pawn*.

1. Откройте *InteractionPawn Blueprint* из папки *Blueprint*.
2. Вы будете использовать события двух контроллеров. Создайте новые события *MotionController* (*L*) *Grip1* и *MotionController* (*L*) *Trigger* нажатием правой кнопкой мыши в *Event Graph* и найдя их по имени.

Событие `Grip` представляет боковую кнопку *grip* на контроллере *Vive* и ручной *trigger* на контроллере *Oculus*. Это хорошие кандидаты для подбора объектов.

3. Создайте новый геттер для *World Interaction Component* и вызовите события `Pickup` и `Drop` на нем.
4. Соедините контакт `Pressed` у `MotionController (L) Grip1` с контактом выполнения `Pickup`, а `Released` с `Drop` (рис. 8.10).

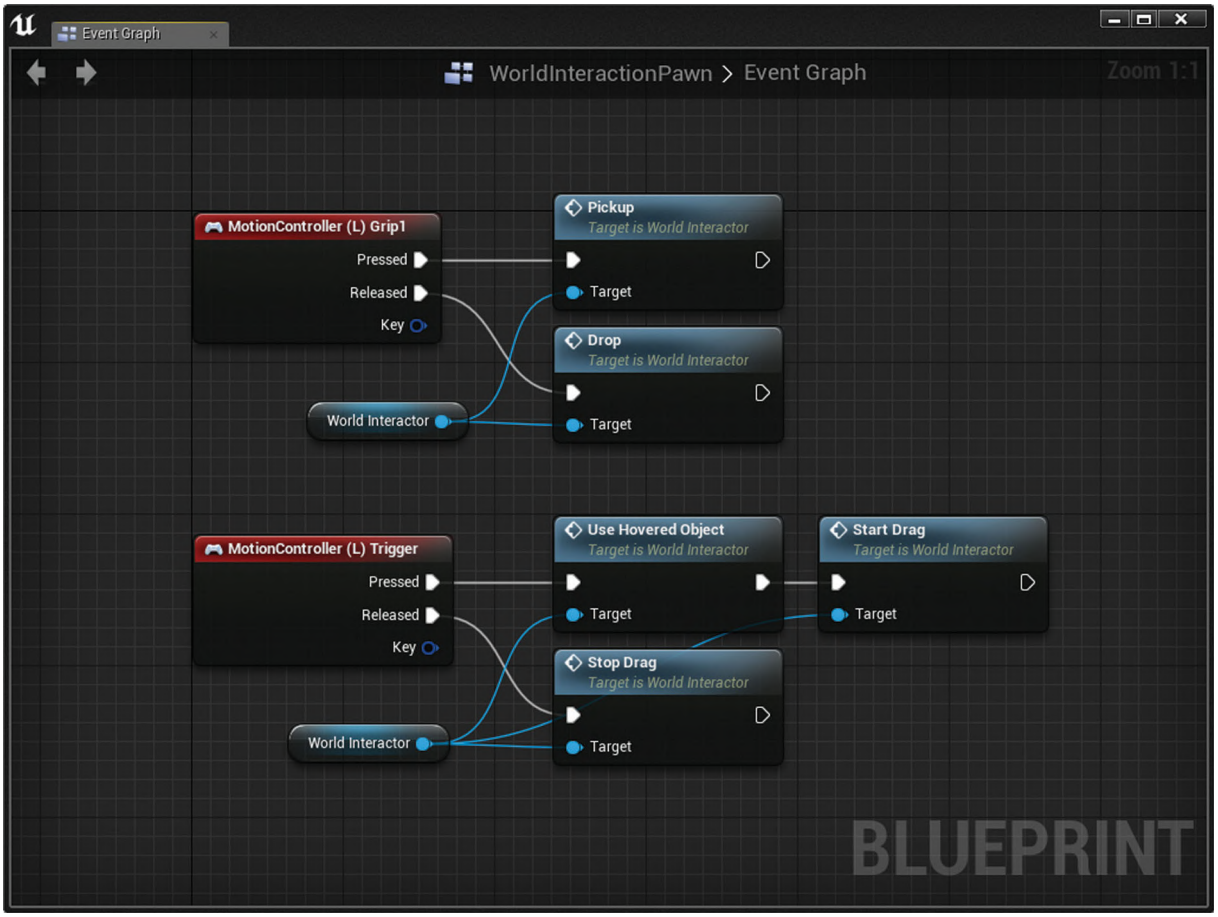

Рис. 8.10. *World Interaction Pawn*: вызов событий на *Interaction Component*.

Это позволит вам выбирать объекты по нажатию *Grid* и опускать их, когда отпустите ее.

5. Создайте новый геттер для *World Interaction Component* и вызовите от него `UseHoveredObject`, `StartDrag` и `StopDrag`.
6. Соедините `Pressed` у `MotionController (L) Trigger` с `UseNoveredObject`, а `UseNoveredObject` с `StartDrag`. Теперь соедините `Released` с `StopDrag` (рис. 8.10). Это позволит вам использовать объекты с помощью *trigger*, как и перетаскивать их, пока он нажат.

Заметка

Если у вас два *Motion Controller Components*, просто повторите эти шаги снова, но изменив контроллер с `L` на `R`, где это необходимо.

8.2. Создание интерактивных объектов

Поскольку у вас появилась система для работы с интерактивными объектами, настало время создать сами эти объекты.

8.2.1. Создание интерактивного *Static Mesh Actor*

Сначала мы создадим простой *Static Mesh Actor*, который позволит вам поднимать и бросать его. Этот *Blueprint* может использоваться затем для базовых декораций сцен, которые не несут какую-то специфическую логику, но должны позволять взаимодействовать с собой.

1. В папке *InteractiveObject* (*Blueprint* ⇒ *InteractiveObject*) добавьте новый *Blueprint* (*Add New* ⇒ *Blueprint Class* ⇒ *Search All* ⇒ *Static Mesh Actor*).
2. Назовите его `InteractiveStaticMesh`.
3. Откройте этот *Blueprint* и нажмите на кнопку *Class Settings*.
4. На панели *Details* добавьте новый *interface* (в меню откройте *Interfaces* ⇒ *Implemented Interface* ⇒ нажмите *Add* и выберите *World Interaction Interface*) (рис. 8.11).

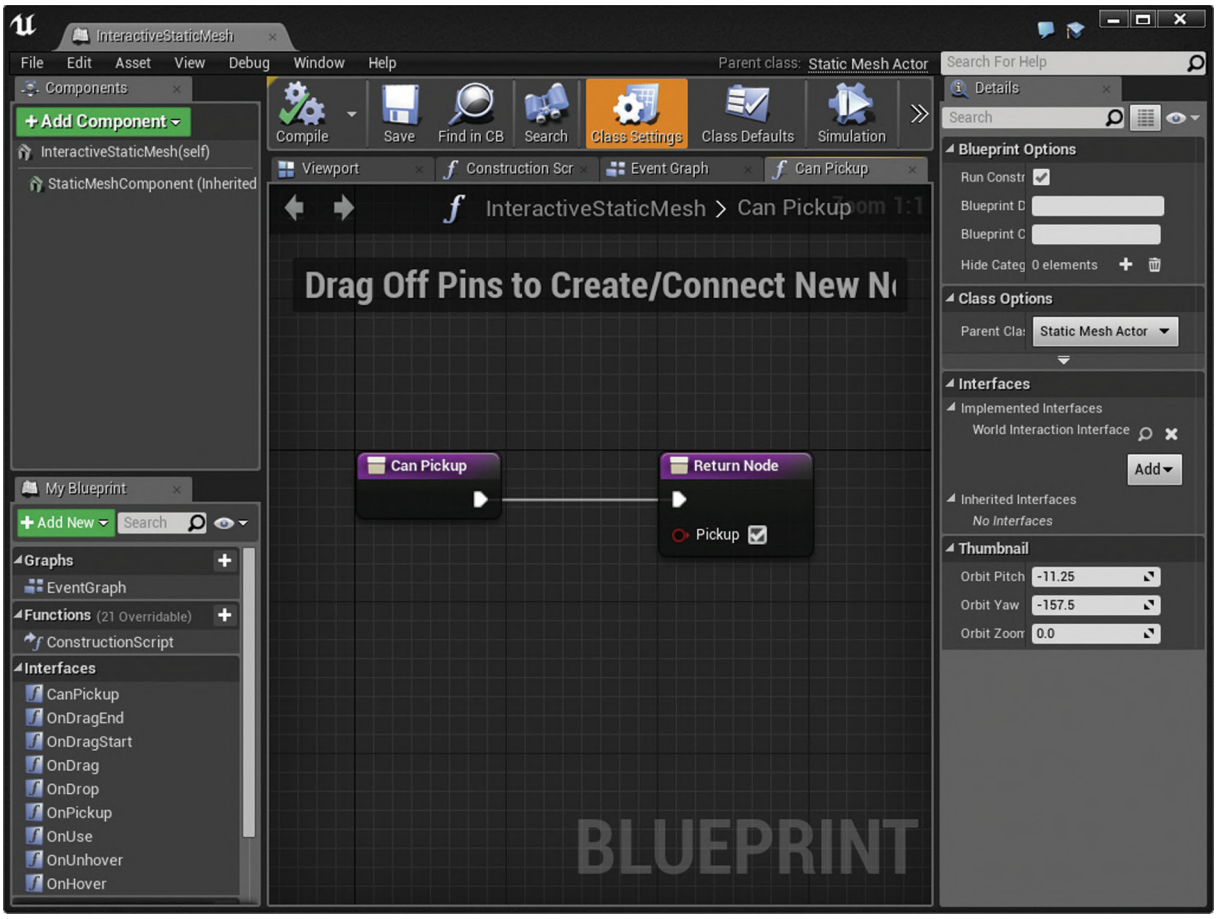

Рис. 8.11. Интерактивный *Static Mesh*: добавление интерфейса и разрешение на взаимодействие

5. Выберите *StaticMeshComponent* (*Inherited*) на панели *Components*. На вкладке *Details* установите *Mobility* на *Movable*. Установите в *Static Mesh Cube* (вам, возможно, понадобится открыть контент движка, чтобы найти *Cube*). Поставьте галочку у *Simulate Physics*. И последнее: разрешите *Generate Overlap Events* в *Collision*.
6. Скомпилируйте *Blueprint* для получения доступа к функциональности интерфейса.
7. Дважды щелкните по функции `CanPickup` и установите возвращаемое значение `Pickup` на `True`. Это позволит вам поднимать этот *Blueprint* (см. рис. 8.11)
8. Откройте функцию `Pickup`.
9. Создайте геттер для *StaticMeshComponent* (*Inherited*).
10. От геттера вызовите `SetSimulatePhysics` и установите в `Simulate` значение `False`. Установите этот узел между точкой входа в функцию и выходом (рис. 8.12). Это предотвратит падение объекта сквозь руки игрока, когда он поднимет что-либо.

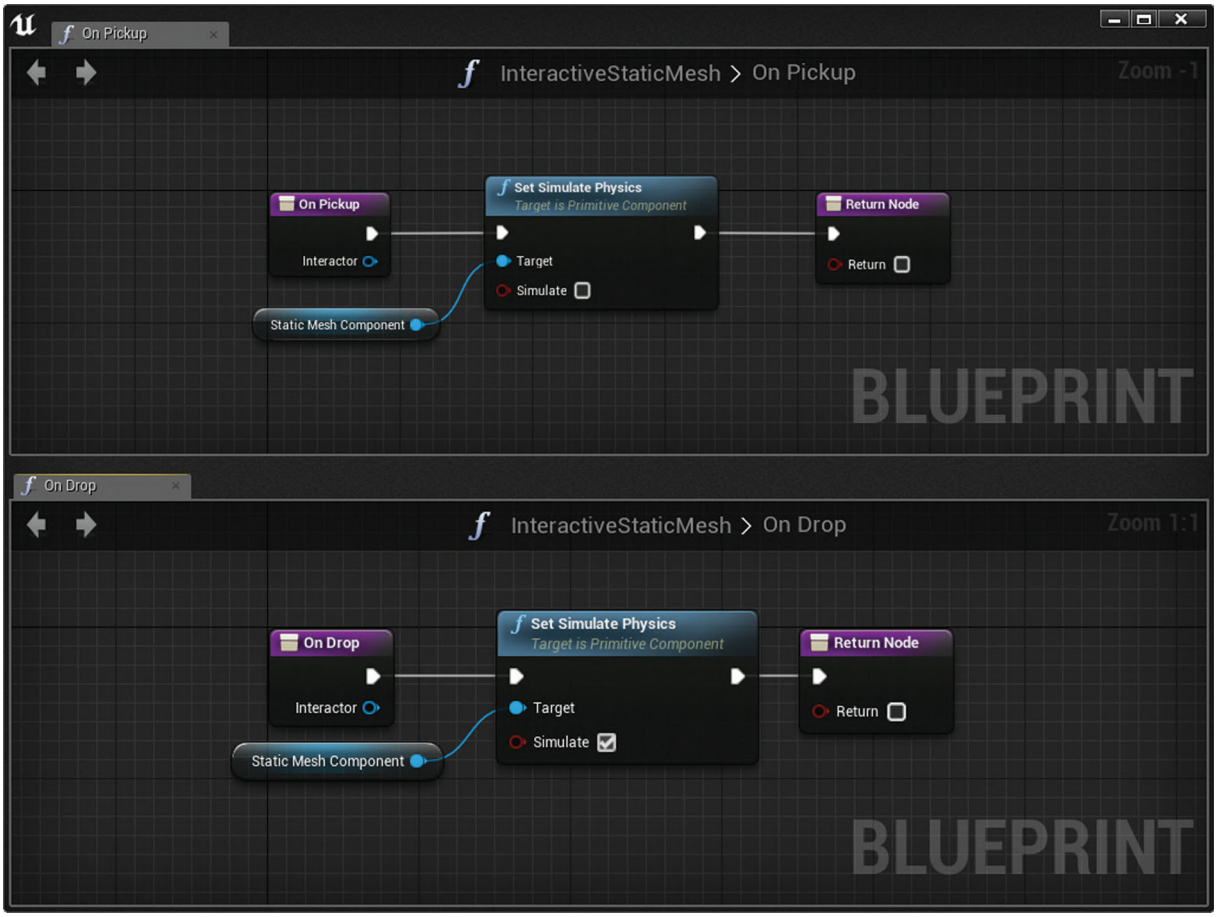

Рис. 8.12. Интерактивный *Static Mesh*: функции «Поднять» и «Выбросить».

12. Откройте функцию `OnDrop`.
13. Проделайте действия из пункта 10, только установите `True` в `Simulate`. Это снова включит физику, когда игрок отпустит предмет.

> **Заметка**
>
> Включение физики для *Component* из *Blueprint* выводит его из иерархии *Blueprint*. В данном случае, поскольку *StaticMeshComponent* является корнем нашего *Blueprint*, это ничего не изменит.

Однако если включить физику для *Component*, который не является корнем *Blueprint*, потребуется повторно присоединить его к корневому компоненту, прежде чем на него будет можно воздействовать преобразованием *Blueprint*.

Рекомендую вам сделать это позже в функции `OnPickup`.

СОХРАНЕНИЕ СТОЛКНОВЕНИЯ

При отключении моделирования физики объект перестает сталкиваться со статическими геометрическими объектами уровня, потому что теперь он изменяется непосредственно под воздействием контроллера движения, связанного с ним.

Если вы хотите сохранить столкновения, обходной путь (хотя и сложный) заключается в том, чтобы не отключать физику; вместо этого установите большие величины линейных и угловых значений демпфирования объекта. Это приведет к тому, что объект будет вести себя так, будто гравитация не влияет на него. Тогда он не станет выпадать из рук пользователя и продолжит сталкиваться со статическими геометрическими объектами.

Однако объект не возвращается в исходное положение после столкновения. Чтобы исправить это, вы можете отключить гравитацию объекта после того, как он поднят, и когда объект «отойдет» от своего первоначального положения, вы сможете имитировать гравитацию в противоположном направлении, чтобы объект, казалось, хотел вернуться в исходное положение.

8.2.2. Создание интерактивной кнопки

Общая система ввода для контроллеров — это обычная кнопка, реагирующая на нажатие пользователя. Первая приходящая в голову идея при желании получить интерактивный объект — это включить физику и добавить некоторые ограничения. Однако на практике такое решение плохо синхронизируется с неограниченным движением контроллера в ваших руках. Поэтому я рекомендую «подделать» ваше физическое взаимодействие, чтобы получить максимальный контроль над кнопкой.

Для создания кнопки не нужно создавать ваш собственный интерфейс. В идеале вы хотите кнопку, которую можно будет нажать любым объектом в руках игрока, а не только *Interactor Component*.

8.2.2.1. Создание кнопки и сетки

Перед тем как вы реализуете функциональность кнопки, вам следует создать визуальное представление из основных компонентов.

1. В папке *InteractiveObject* добавьте новый *Actor Blueprint*.
2. Назовите его `InteractiveButton` и откройте.
3. Создайте три компонента: *Scene Component* под названием `ButtonRoot` и два *Cylinder Component* под названиями `Button` и `Base`.
4. Перенесите `Button Cylinder` на `ButtonRoot`, чтобы сделать его наследником.
5. Установите позицию `ButtonRoot` на `(X = 0, Y = 0, Z = 5)`. Это сдвинет кнопку на 5 единиц вверх по оси `Z`.
6. Установите масштаб `Button Component` на `(X = 0.1, Y = 0,1 Z = 0.05)`. Это сделает ее более похожей на кнопку.
7. Установите масштаб `Base Component` на `(X = 0.2, Y = 0,2 Z = 0.05)`. Это сделает ее похожей на основу для кнопки (рис. 8.13).

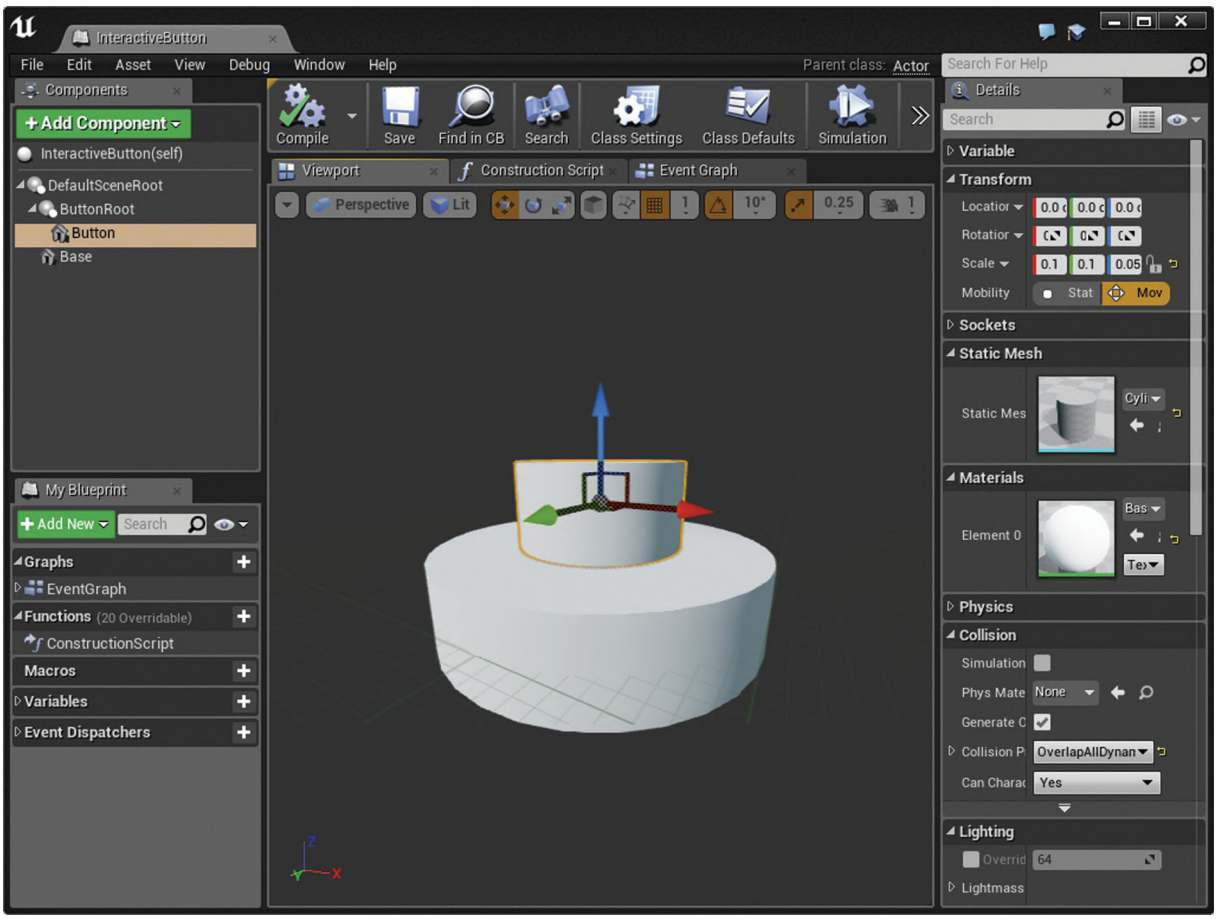

Рис. 8.13. Интерактивная кнопка: базовые компоненты

8. Выберите `Button Component` и установите `Collision Present` на `OverlapAllDynamic`. Вы будете использовать наложение кнопки для определения, нажата ли кнопка.
9. Выберите `Base Component` и установите `Collision Present` на `NoCollision`. Таким образом он не будет мешать кнопке.

8.2.2.2. Определение наложений кнопки

Мы настроили *Mesh*. Теперь необходимо определить, когда кнопка накладывается с *Actor*, чтобы вы могли применить математику нажатия.

1. Создайте новую переменную ссылку *Primitive Component* и назовите ее `OverlappedComponent`. Это позволит сохранять текущие компоненты.
2. Создайте новую переменную *Vector* и назовите ее `InitialOverlapLocation`. Она будет хранить начальные координаты наложения.
3. Создайте новую переменную `Float` и назовите ее `ButtonPressAmount`. Она будет хранить количество нажатий кнопок.
4. Создайте еще одну переменную `Float`, назовите ее `MaxButtonPressAmount` и установите в ней значение по умолчанию `4`. Это будет максимальное число нажатий на кнопку до вызова события *Presser*.
5. Создайте новый *Event Dispatcher* под названием `Pressed`. Вы будете использовать его, потому что это позволит другому *Actor* легко привязать события к нажатию на кнопку без необходимости знать об этом.
6. Выберите *Button Component* и на вкладке *Details* нажмите + у событий `OnComponentBeginOverlap` и `OnComponentEndOverlap`. Эти события показаны в *Event Graph* (рис. 8.14).

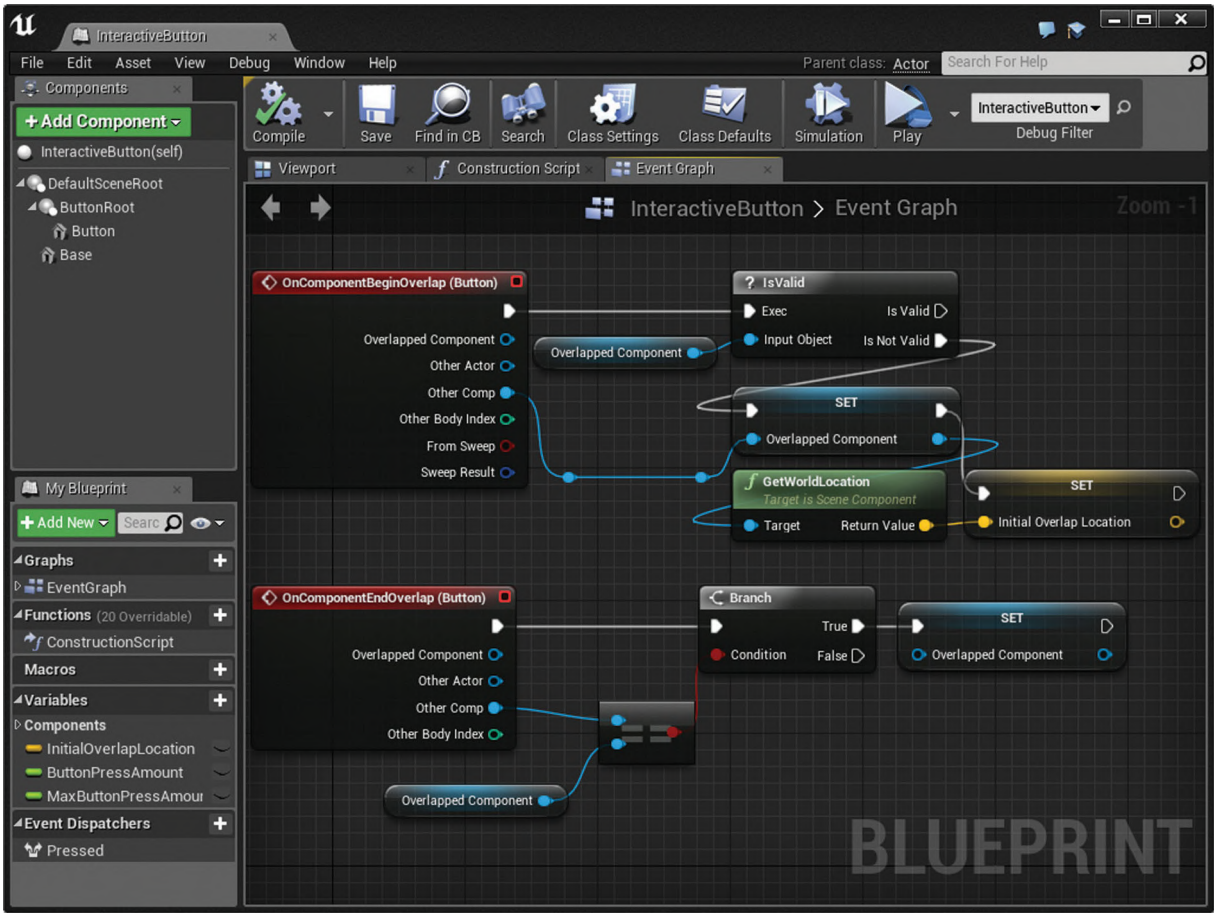

Рис. 8.14. Интерактивная кнопка: определение наложений с кнопкой.

7. После узла события `OnComponentBeginOverlap` добавьте узел `IsValid` и соедините его с новым геттером для `OverlappedComponent`.
8. Создайте новый сеттер для `OverlappedComponent` и соедините его с выходом `Is Not Valid`. После соедините контакт события `OtherComp` с входом нашего сеттера.
9. Создайте сеттер для переменной `InitialOverlappedcomponent` и соедините его с выходом сеттера `OverlappedComponent`.
10. От синего контакта выхода `OverlappedComponent` вызовите функцию `GetWorldLocation` и соедините ее выход с сеттером `InitialOverlappedcomponent` (рис. 8.14).
11. Создайте новый *Branch* и соедините его с событием `OnComponentEndOverlap`.
12. От контакта `Other Comp` события создайте узел *Equal* и создайте новый геттер для `OverlappedComponent`.
13. Создайте новый сеттер `OverlappedComponent` и соедините его с выходом `True`, убедившись, что сеттер остается пустым, потому что мы хотим очистить `OverlappedComponent`.

8.2.2.3. Нажатие на кнопку

Теперь мы можем обнаруживать соприкосновение с кнопкой, и нам надо нажать ее.

Для этого преобразуем начальную позицию *Overlapped Component* относительно кнопки, чтобы определитель, насколько далеко отодвинулся объект, пока он соприкасался с кнопкой. Это покажет, как вам надо нажимать на кнопку в относительном пространстве. Нажатие на кнопку — это все равно что вычитание этого значения из начальной позиции кнопки.

Вычисление по отношению к кнопке позволяет вам при необходимости повернуть кнопку в пространстве и убедиться, что она функционирует правильно.

1. От узла *Event Tick* создайте *Sequence*.
2. Добавьте новый узел *IsValid* и свяжите синий вход с геттером `OverlappedComponent`, соедините его с первым выходом `Sequence`. Таким образом, нажатие кнопки будет обрабатываться, только если у вас есть объект наложения.
3. Создайте новый геттер для компонента *ButtonRoot*.
4. От него создайте два узла *GetWorldTransform*.
5. От каждого выходного значения этих узлов вызовите `InverseTransformLocation`. Это преобразует любую координату относительно *ButtonRoot*.
6. Для `Location` первого `InverseTransformLocation` создайте геттер `InitialOverlapLocation`.
7. Для второго входа `Location` создайте новый геттер `OverlappedComponent` и вызовите от него `GetWorldLocation`, который соедините с `Location`.
8. Для нахождения дистанции возьмите выходное значение первого `InitialOverlapLocation` и вычтите из него второе.
9. Теперь у вас есть расстояние, пройденное наложенным компонентом в виде вектора. Разбейте выход на три контакта координат. Это позволит вам получить доступ к осям.
10. Так как кнопка будет двигаться по оси `Z`, вам надо изменять позицию кнопки вверх и вниз, чтобы компенсировать положение руки пользователя. От контакта `Z` вызовите новый узел `Clamp (float)`.
11. Добавьте новый геттер для `MaxButtonPressAmount` и соедините его с входом `Max`. В `Min` укажите `0`.
12. Создайте новый сеттер для `ButtonPressAmount` и соедините его с `Clamp`.
13. Соедините выход *Is Valid* с этим сеттером (см. рис. 8.15).

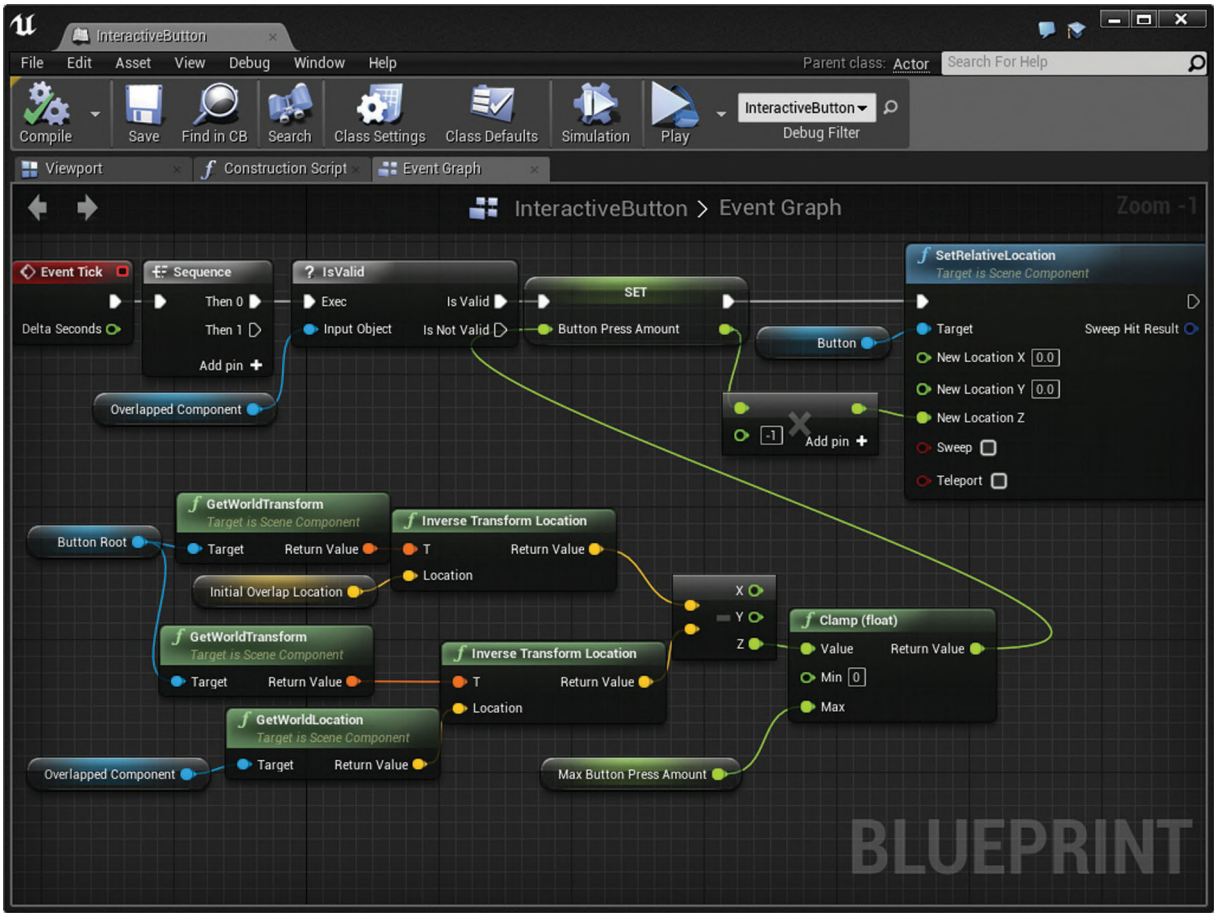

Рис. 8.15. Интерактивная кнопка: обработка нажатия

14. Создайте новый геттер для *Button Component* и вызовите функцию `SetRelativeLocation`.
15. Соедините контакт выполнения `SetRelativeLocation` с сеттером `ButtonPressAmount`.
16. Правым кликом по `New Location` разделите структуру.
17. Чтобы быть уверенным, что вы нажали на кнопку вниз, от зеленого контакта `ButtomPressAmount` вызовите функцию `Float * Float`, установив во втором входном значении `-1`.
18. После перемножения установите значение в `New Location Z` (см. рис. 8.15).

8.2.2.4. Возврат кнопки в исходное положение

После нажатия на кнопку нужно поднять ее, вернув, таким образом, в исходное положение. Давайте сделаем этот процесс автоматическим.

1. Создайте новый геттер для *Button Component* и добавьте `SetRelativeLocation` с ним в `Target`.
2. Соедините выход *Is Not Valid* с `SetRelativeLocation`.
3. Создайте новый узел *VInterpTo* и установите его выходное значение в `New Location`.

4. Добавьте новый геттер для *Button Component* и получите его `RelativeLocation`.
5. Соединит его с входом `Current`.
6. Установите в `Target (X=0, Y=0, Z=0)`; это начальная позиция кнопки.
7. Соедините `Delta Seconds` узла *Event Tick* с `DataTime` (рис. 8.16).

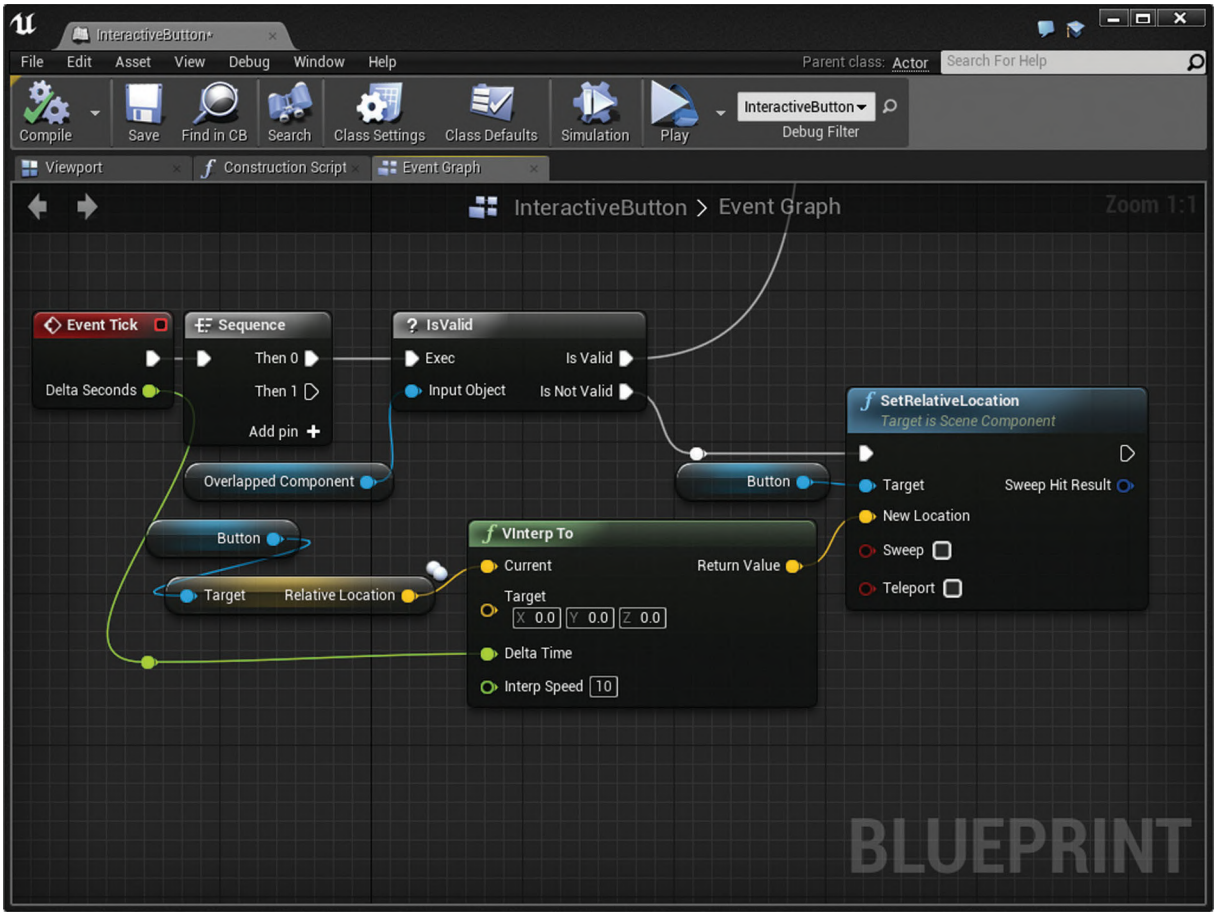

Рис. 8.16. Интерактивная кнопка: возвращение в исходное положение

8. Установите в `Interp Speed` на `10`. Это вернет кнопку к ее начальному положению, когда она не нажата.

8.2.2.5. Активация кнопки

У вас есть кнопка, которая нажимается, когда вы касаетесь ее, и поднимается обратно.

Завершающая вещь, которую вам надо сделать, это определитель, когда кто-то полностью нажал кнопку.

1. От второго выхода *Sequence* создайте узел *Branch*. Вы будете использовать его для определения, когда текущее нажатие равно максимальному, и вызывать `Pressed` в этом случае.
2. Создайте два новых геттера для `ButtonPressAmount` и `MaxButtonPressAmount`.

3. От `ButtonPressAmount` создайте новый `Float >= Float`, установив во второй контакт `MaxButtonPressAmount`.
4. Соедините это условие с *Branch*.
5. От выхода `True` вызовите `DoOnce`. А `False` соедините с `Reset` у `DoOnce`. Таким образом, мы будем вызывать `Pressed` только один раз при нажатии.
6. После узла `DoOnce` вызовите `Event Disputcher Pressed` и выберите `Call` из списка.
7. Соедините `Complete` с вызовом `Pressed` (рис. 8.17).

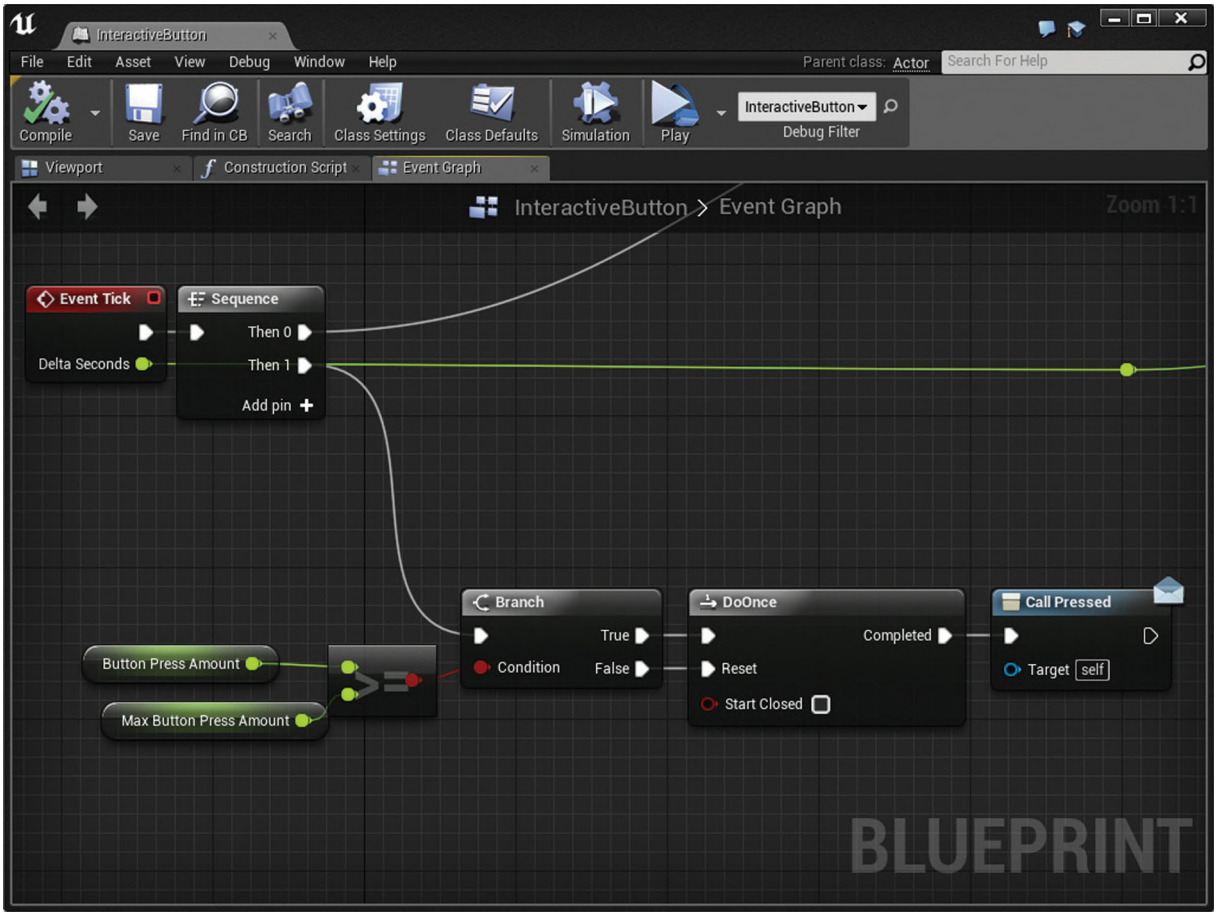

Рис. 8.17. Интерактивная кнопка: вызов *Pressed*

8.2.2.6. Использование кнопки

Вот мы и сделали полноценную кнопку. Давайте применим ее.

Простым способом реализовать вызов события при нажатии на кнопку (когда она находится на одном уровне с игроком) является ссылка на нее в *Blueprint* уровня и назначение события на ее диспетчер *Pressed*.

1. Откройте *Blueprint* уровня и создайте ссылку на эту кнопку.
2. Перетащите ссылку и вызовите макрос *AssignPressed*; он создаст два узла. Первый — это узел *BindEventToPressed*, который при вызове привязывает событие к диспетчеру событий, позволяя ему вызываться при каждом нажатии кнопки. Второе — это фактическое событие, которое будет вызвано при нажатии кнопки.
3. Прикрепите к контакту вызова узла *BindEventToPressed* событие `Event Begin Play`. Это позволит событию `Pressed_Event_0` быть вызванным при нажатии кнопки.
4. Делайте все, что вашей игре нужно, когда кнопка нажата (например, сыграйте последовательность *Sequencer*, чтобы открыть дверь, рис. 8.18).

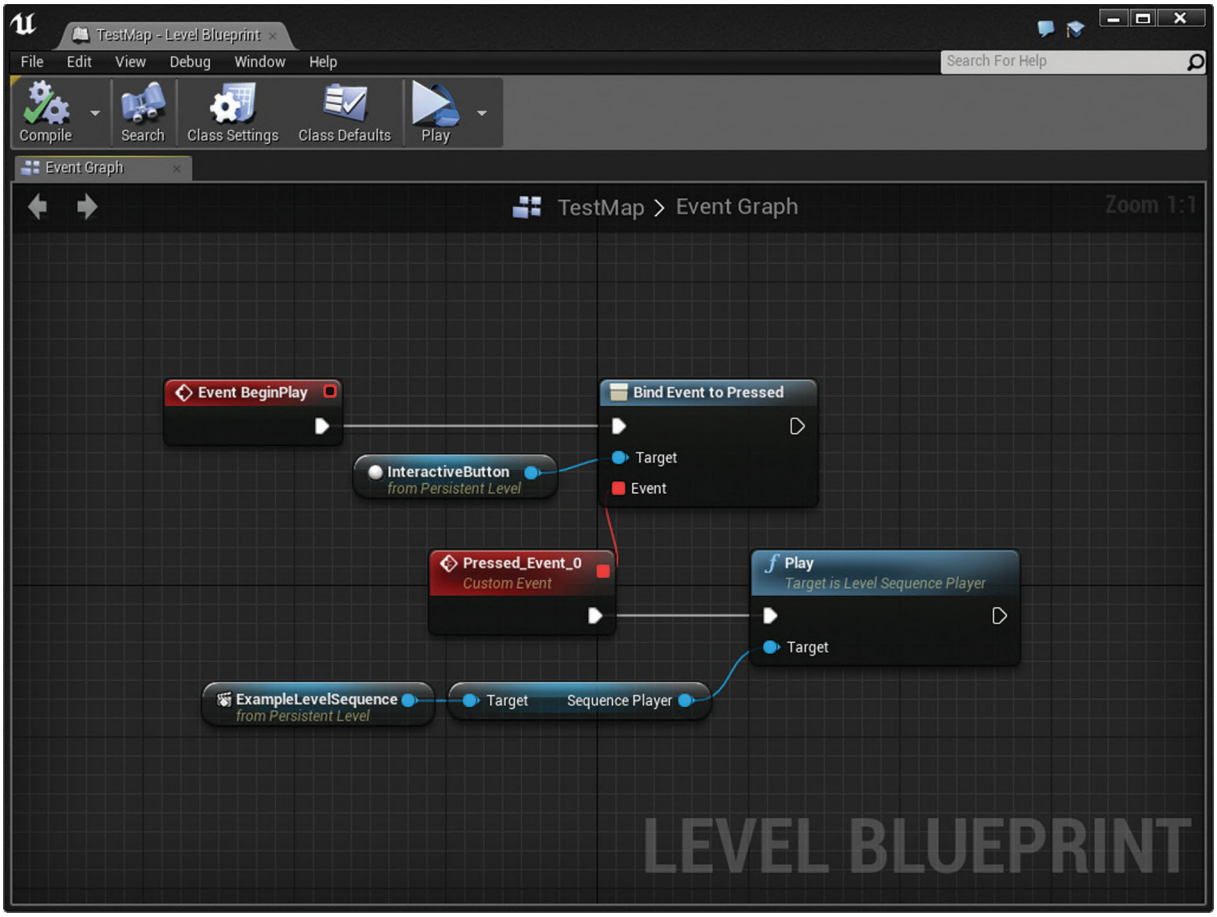

Рис. 8.18. Интерактивная кнопка: использование кнопки на уровне

8.2.3. Создание интерактивного рычага

В этом разделе мы создадим рычаг, который можно будет потянуть функцией захвата в *World Interactor*.

Как и у кнопки, у рычага будет одно выходное событие, которое говорит вам, что где-то был потянут рычаг, но в отличие от кнопки, тяга рычага — это большее физическое действие, поэтому оно должно использоваться для преднамеренных действий, которые используются реже.

8.2.3.1. Создание компонента рычага

Для настройки внешнего вида рычага вы будете использовать простой цилиндр.

1. В папке *InteractiveObjects* создайте новый *Actor Blueprint*.
2. Назовите его `InteractiveLever`.
3. Выберите *Class Settings*.
4. На панели *Details* добавьте *World Interaction Interface* (как мы делали для кнопки).
5. Скомпилируйте *Blueprint* для получения доступа к функциональности интерфейса.
6. Создайте два *Cylinder Component* и назовите их `LeverCylinder` и `LeverHandle`.
7. Создайте *Sphere Component* и назовите его `LeverEnd`.
8. Создайте *Scene Component* и назовите его `LeverRotate`. Этот компонент вы будете вращать.
9. Поверните *LeverCylinder Component* на `90` градусов по оси `X` и установите масштаб `(X = 0.3, Y = 0.3, Z = 0.1)`. Это сделает цилиндр более похожим на основу рычага.
10. Сдвиньте `LeverHandler` на `25` единиц по оси `Z`, а масштаб установите `(X = 0.05, Y = 0.05, Z = 0.5)`. Это даст вам цилиндр, похожий на ручку рычага.
11. Сдвиньте `LeverEnd` на `50` единиц по оси `Z`, а масштаб установите `(X = 0.1, Y = 0.1, Z = 0.1)`. Это даст вам сферу, похожую на конец рычага.
12. Прикрепите `LeverEnd` и `LeverHandle` к `LeverRotate`. Это позволит вам вращать рычаг при помощи `LeverRotate` (рис. 8.19).

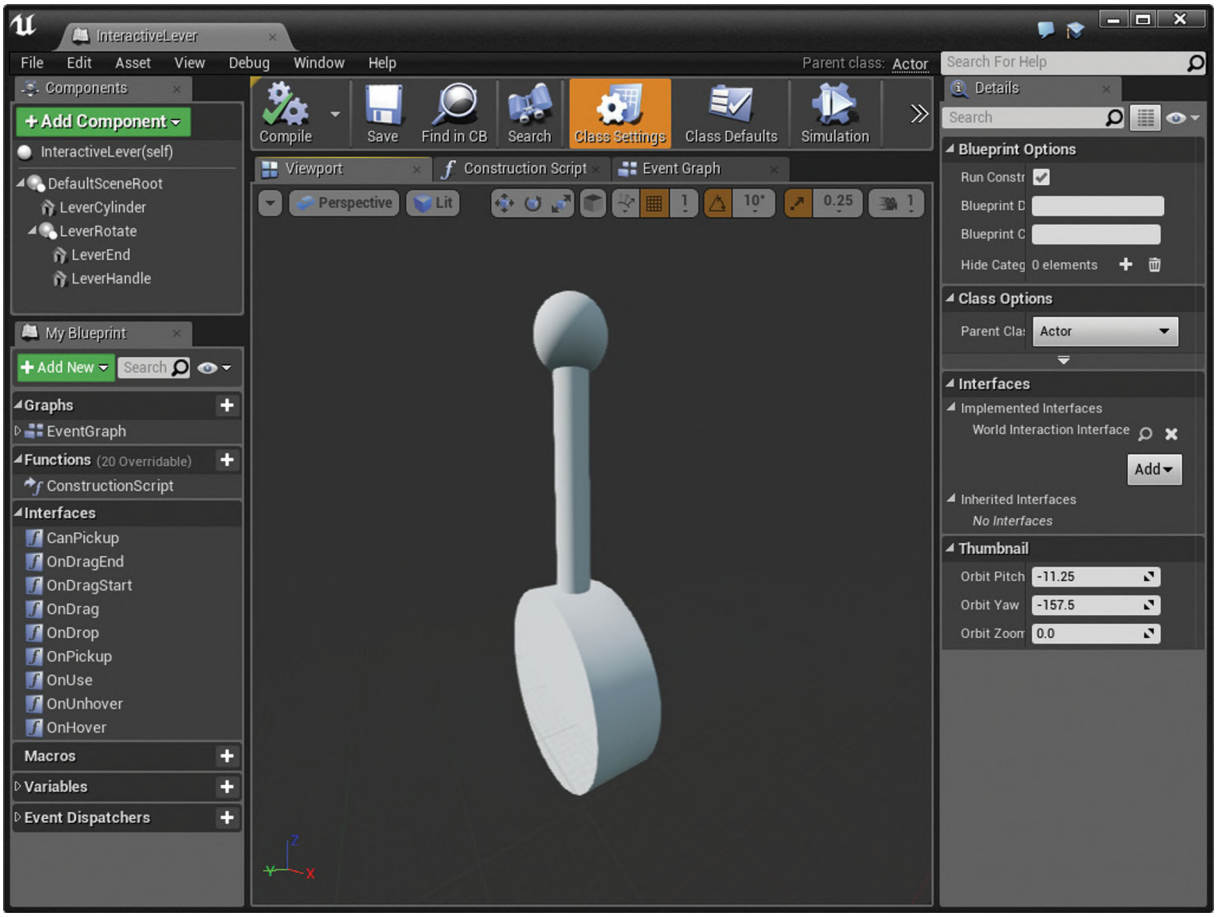

Рис. 8.19. Компоненты интерактивного рычага

8.2.3.2. Вращение рычага

Когда вы взаимодействуете с рычагом, вы хотите повернуть его *LeverRotate*. Это потребует использования тригонометрии, которую я привел ниже, если вы не сталкивались с ней раньше.

Во-первых, вам нужно настроить несколько переменных.

1. Создайте переменную `CurrentInteractor` типа `WolrdInteractor`. Это будет хранилище для взаимодействующих объектов с рычагом.
2. Создайте новую переменную `IsResettingRotation` типа `Boolean`. Ее вы будете использовать для контроля анимации возвращения рычага.
3. Создайте две новые переменные: `InitialResetRotation` и `InitialLeverRotation` типа `Rotator`. Первая будет хранить поворот перед анимацией возвращения рычага в начальное положение, а вторая — начальное положения.
4. Создайте четыре переменные типа `Float` и назовите их `MinPitch`, `MaxPitch`, `ActivationPitch` и `CurrentPitch`.

a. Первые две хранят нижнюю и верхнюю границы поворота рычага.

b. `ActivationPitch` хранит наклон, который активирует рычаг и возвращает в начальное положение.

c. `CurrentPitch` содержит текущий наклон рычага.

5. Установите `MinPitch` и `ActivationPitch` на `-90`.

6. Создайте новый `Dispatcher` под названием `Pulled`.

7. От события *Event* `BeginPlay` создайте сеттер для `InitialLeverRotation`.

8. Создайте новый геттер для *LeverRotation Component* и получите его `Relevant Rotation`, соединив его с сеттером (рис. 8.20).

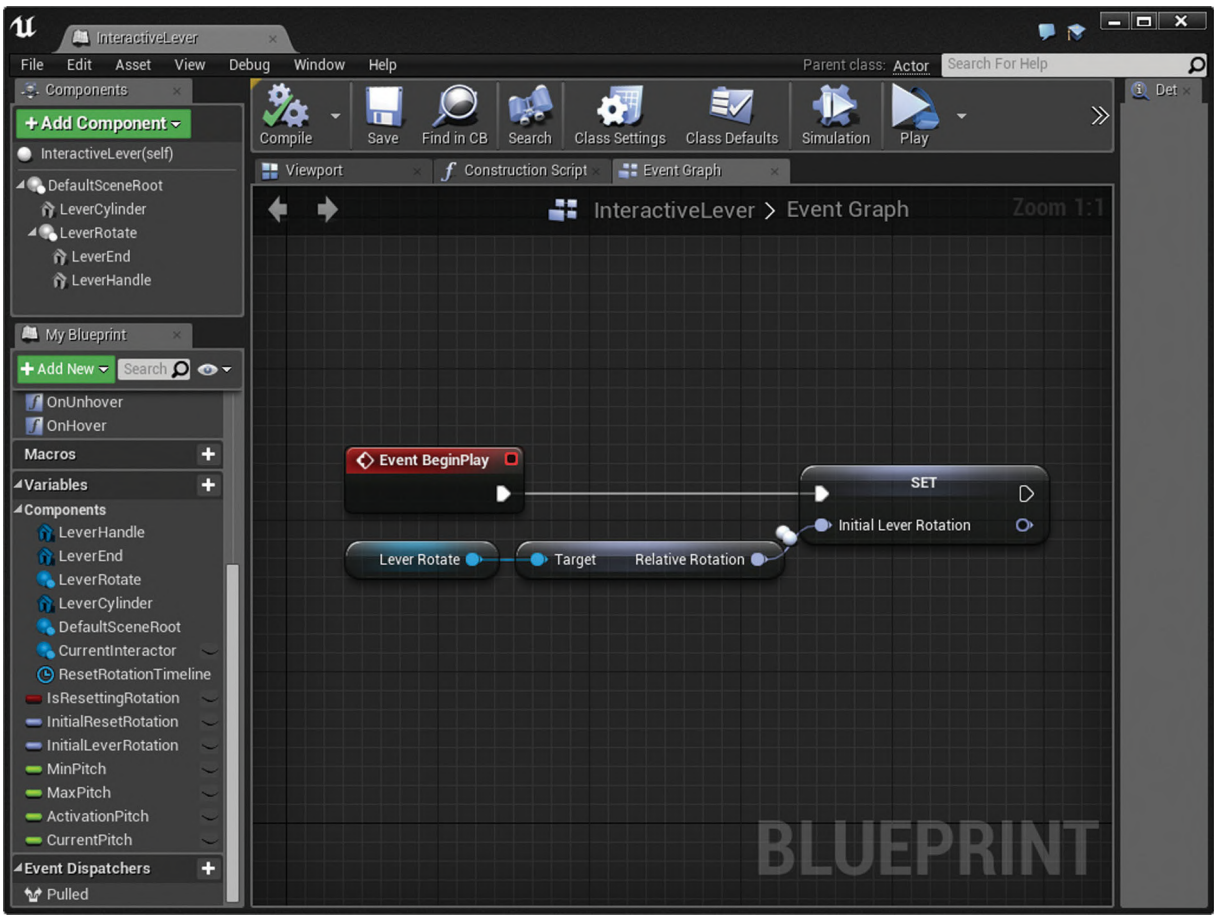

Рис. 8.20. Интерактивный рычаг: настройка переменных

> **Предупреждение**
>
> Ниже приводятся некоторые сведения из тригонометрии. Если вы с ними уже знакомы, можете их пропустить.

Мы определили все нужные переменные. Давайте рассчитаем поворот.

Для этого мы переведем координаты текущего объекта, взаимодействующего с рычагом, в относительные. Это корректно для некоторых преобразований мира, которые вы применяете на рычаге. Вы находите относительный наклон поворота для текущего положения через арктангенс координат *Z* и *X*.

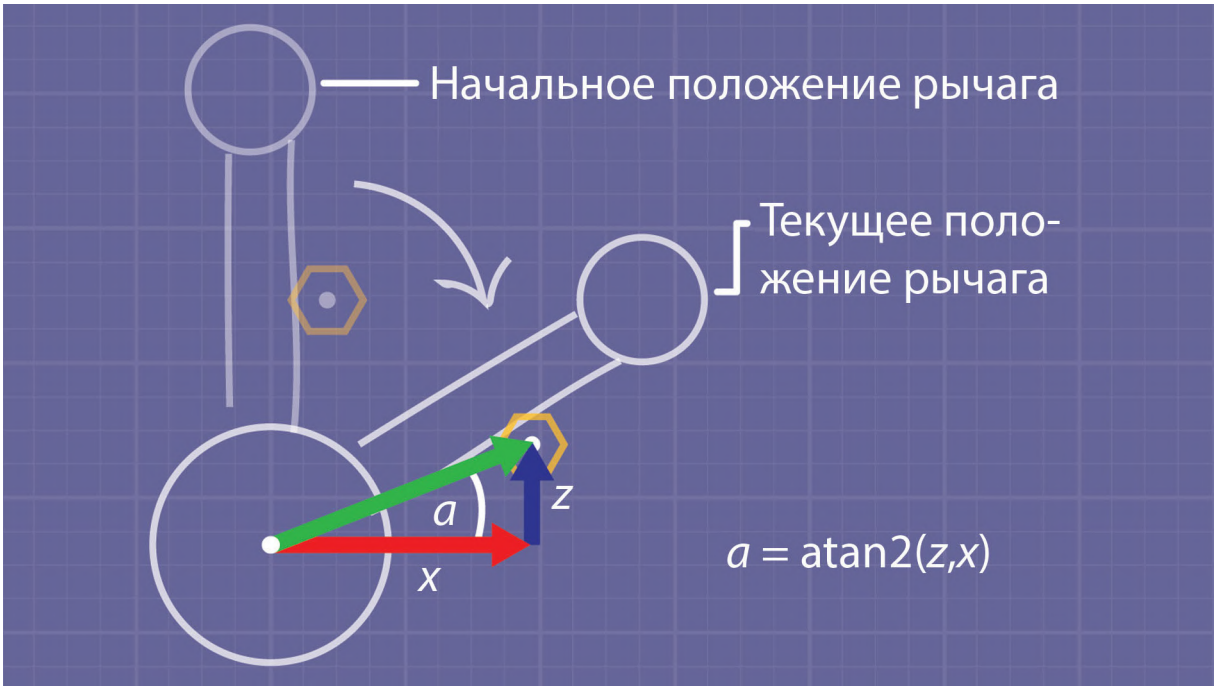

Рис. 8.21. Интерактивный рычаг; *x* — дистанция от центра рычага по оси *X*, *z* — дистанция по оси *Z* до точки на рукояти, а — угол между осью *X* и точкой на рукояти.

АРКТАНГЕНС И ФУНКЦИЯ ATAN2

В тригонометрии тангенс — это угол, равный длине стороны, противолежащей острому углу прямоугольного треугольника, разделенной на длину прилегающей к нему стороны.

tanangles = opposite/adjacent.

Арктангенс, или обратный тангенс, — это функция, обратная тангенсу. Арктангенс позволяет получить меру угла между противолежащей углу стороной и прилегающей к нему

atan(opposite/adjacent) = angle.

Наконец, atan2 — это функция расширения *atan*, которая принимает на вход (*x*, *y*). Это позволяет функции не получать ошибку деления на 0, а также собирать информацию о знаках сторон для возврата значения в определенном квадранте.

Не беспокойтесь, мы рассмотрим все подробно.

1. Откройте функцию `OnDrop`.
2. От точки входа в метод вызовите *Branch*.
3. От параметра *Interactor* создайте узел *Equal* (*Object*), установив во второй вход геттер для *CurrentInteractor*.
4. Соедините выход равенства с *Condition*.
5. От `False` создайте узел *Return*.
6. От `True` создайте еще один *Branch*.
7. Создайте геттер для `IsResettingRotation`.
8. От него вызовите узел `NOT Boolean`
9. Установите результат в *Condition* второго *Branch*. Таким образом, вы не сможете повернуть рычаг во время его анимации возврата в начальное положение.
10. Создайте узел *GetActorTransform*.
11. От него вызовите `InverseTransformLocation`.
12. Создайте геттер для *CurrentInteractor* и вызовите от него *GetWorldLocation*.
13. Соедините возвращаемое значение *GetWorldLocation* со входом `Location` узла *InverseTransformLocation*. Таким образом, мы преобразуем мировые координаты взаимодействующего в локальное пространство.
14. От узла *InverseTransformLocation* вызовите `UnrotateVector`, установив `90` градусов по оси `Y` во вход `B` (рис. 8.22). Это повернет вектор, потому что вы хотите посчитать наклон в относительном положении, которое смотрит вверх, а не вперед.
15. Нажатием правой кнопки по возвращаемому значению разбейте его.
16. От выхода `Z` вызовите функцию `Atan2 (Degrees)`.
17. На второй вход `B` подайте `X`.

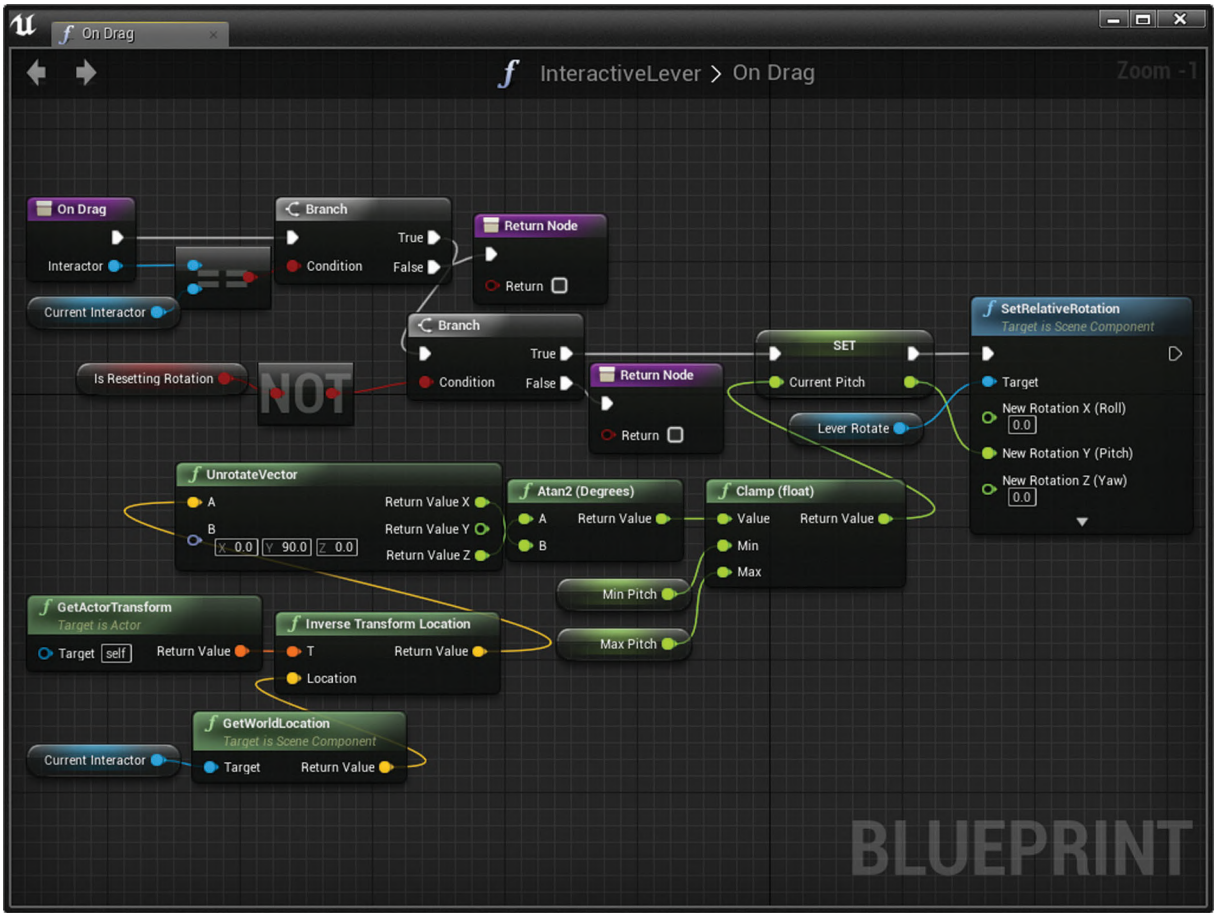

Рис. 8.22. Интерактивный рычаг: подсчет поворота

18. После узла `Atan2` вызовите `Clamp (float)`.
19. Создайте два новых геттера для `MinPitch` и `MaxPitch`, затем установите их в `Min` и `Max` входы `Clamp`.
20. Создайте новый сеттер для `CurrentPitch`, установив ему на вход значение из `Clamp`.
21. Соедините выход `True` с сеттером (рис. 8.22).
22. Создайте новый геттер для `LeverRotate` и вызовите от него `SetRelativeRotation`.
23. Разбейте вектор `NewRotation`.
24. Установите выходное значение из сеттера в ось `Y`, а также соедините выходы выполнения (рис. 8.22).

8.2.3.3. Возврат рычага

Мы создали вполне рабочий рычаг, но он вращается только туда и обратно. Чтобы сделать его более правдоподобным, вам надо вызывать его возвращение в исходное положение, когда он достиг конца или был отпущен.

1. Правым нажатием в *Event Graph* добавьте новое событие `ResetRotation`.
2. Создайте новый сеттер для `IsResettingRotation` и соедините его выполнение с событием. Установите его значение в `True`.
3. Создайте сеттер для `InitialResetRotation`. Соедините его выполнение с сеттером `IsResettingRotation`.
4. Создайте новый геттер для `LeverRotation` и получите его `RelativeRotation`, установив это на вход сеттера `InitialResetRotation` (рис. 8.23).

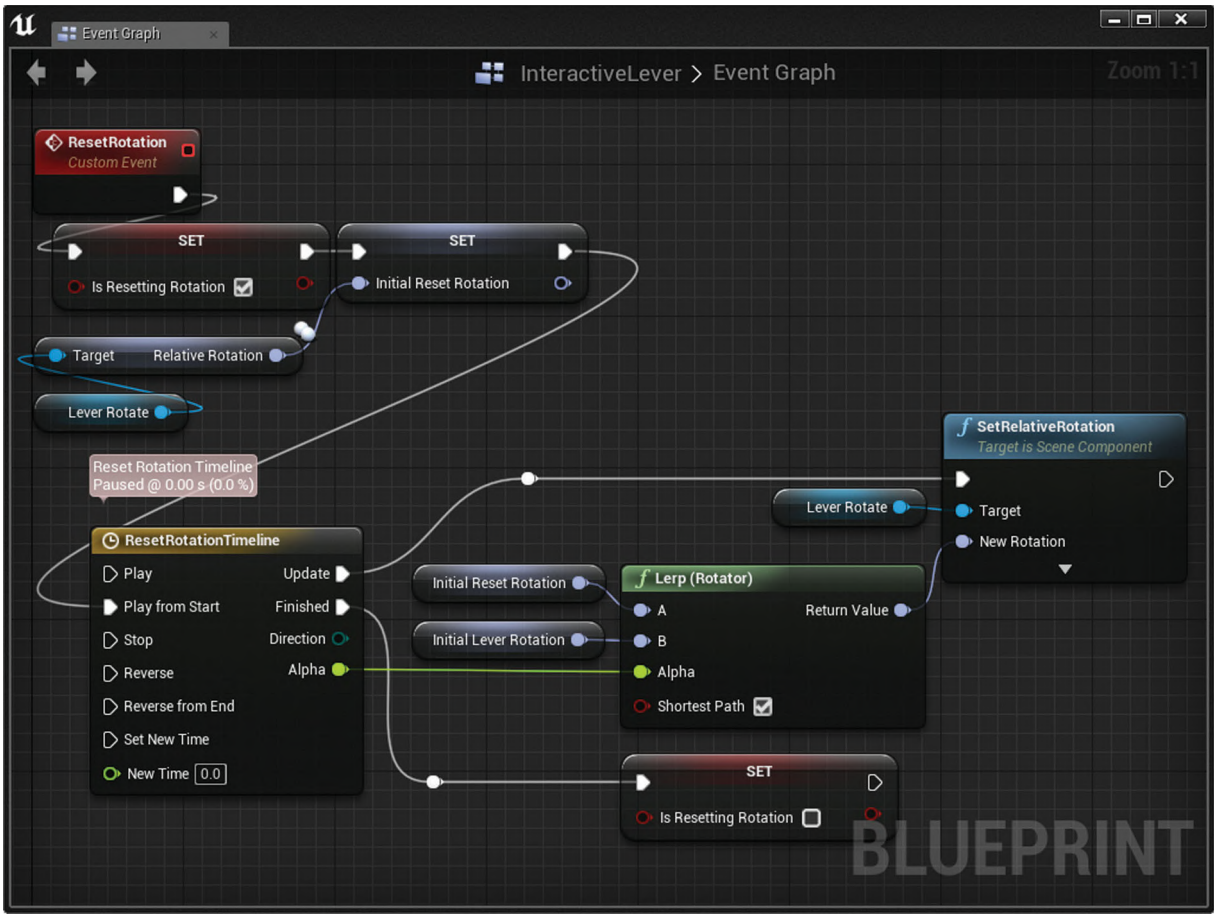

Рис. 8.23. Интерактивный рычаг: возврат в исходное положение

5. Чтобы повернуть рычаг, вы создадите *Timeline*. Правым нажатием в *Event Graph* выберите *Add* `Timeline` и назовите его `ResetRotationTimeline`.
6. Двойным нажатием откройте его.
7. Добавьте новую дорожку `Float` нажав *Add Float Track*. Назовите ее `Alpha`.
8. Создайте две точки на *Timeline*, кликая по площади сетки с нажатой клавишей *Shift*.
9. Установите на первой точке значение и время на `0`.
10. Установите у второй время на `0.5`, а значение на `1` (рис. 8.24).

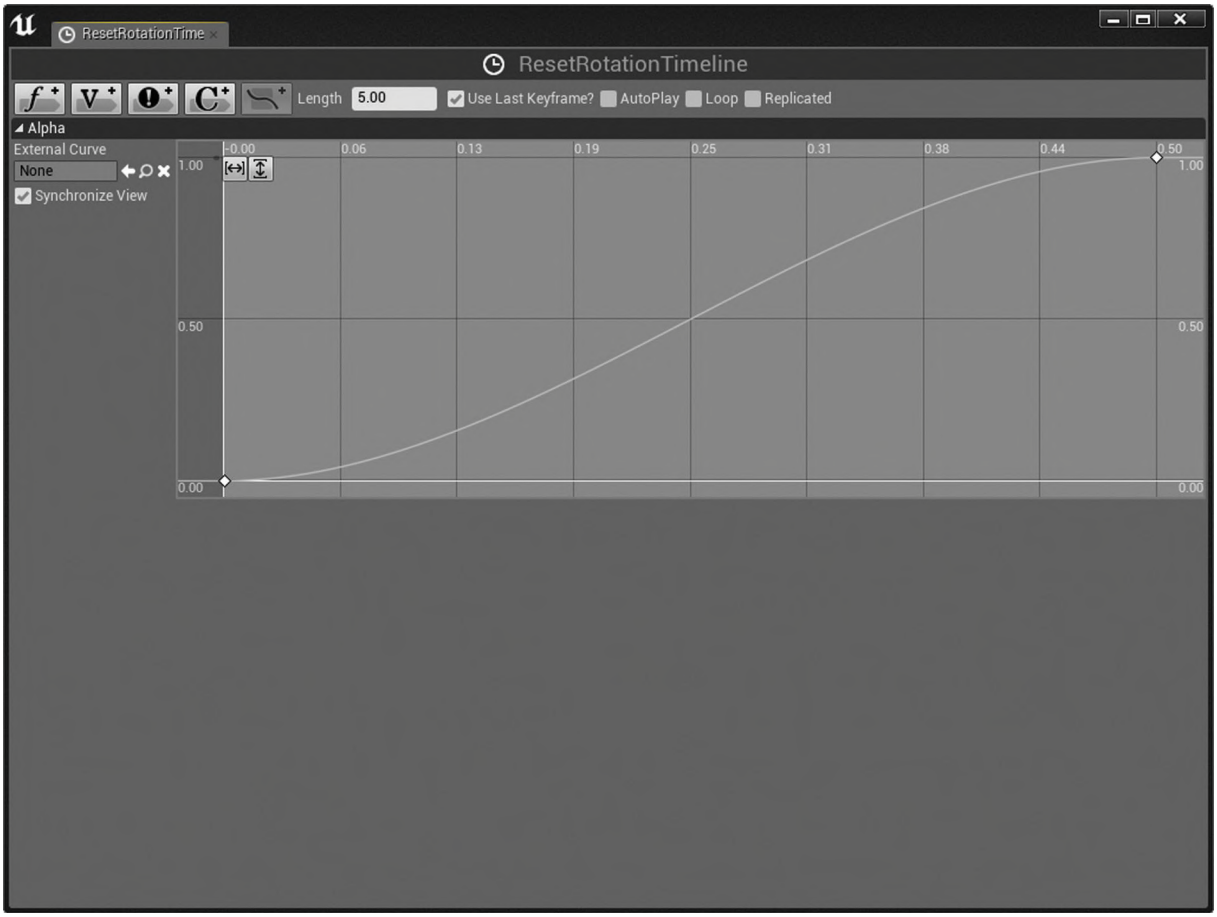

Рис. 8.24. Интерактивный рычаг: создание *timeline*

11. Выберите обе точки и нажмите правой кнопкой мыши. Выберите `Auto` под `Key Interpolation`. Это создаст линию между точками.
12. Проверьте, что включено `Use Last Keyframe`.
13. Соедините выполнения `InitialResetRotation` с `Play from Start` у *timeline*.
14. Создайте два новых геттера для `InitialResetRotation` и `InitialLeverRotation`.
15. От `InitialResetRotation` вызовите *Lerp* (*Rotator*).
16. Установите `InitialLeverRotation` во второй вход.
17. Соедините `Alpha timeline` с `Alpha` у *Lerp*.
18. Проверьте, что включено `Shortest Path Boolean` у *Lerp*. Этот узел позволит вам линейно интерполировать значения между двумя вращениями; хотя в нашем случае вы можете создать любую кривую интерполяции на *Timeline*.
19. Создайте новый геттер для *LeverRotate Component* и вызовите от него `SetRelativeRotation`.
20. В `New Rotation` установите значение из *Lerp*, а контакт выполнения соедините с `Update` узла *Timeline*.

21. Создайте новый сеттер для `IsResettingRotation` и соедините его с контактом выполнения `Finished` (см. рис. 8.22).
22. Откройте функцию `OnDropEnd`.
23. Создайте новый узел *Branch* и соедините его с контактом выполнения начала функции (рис. 8.25).

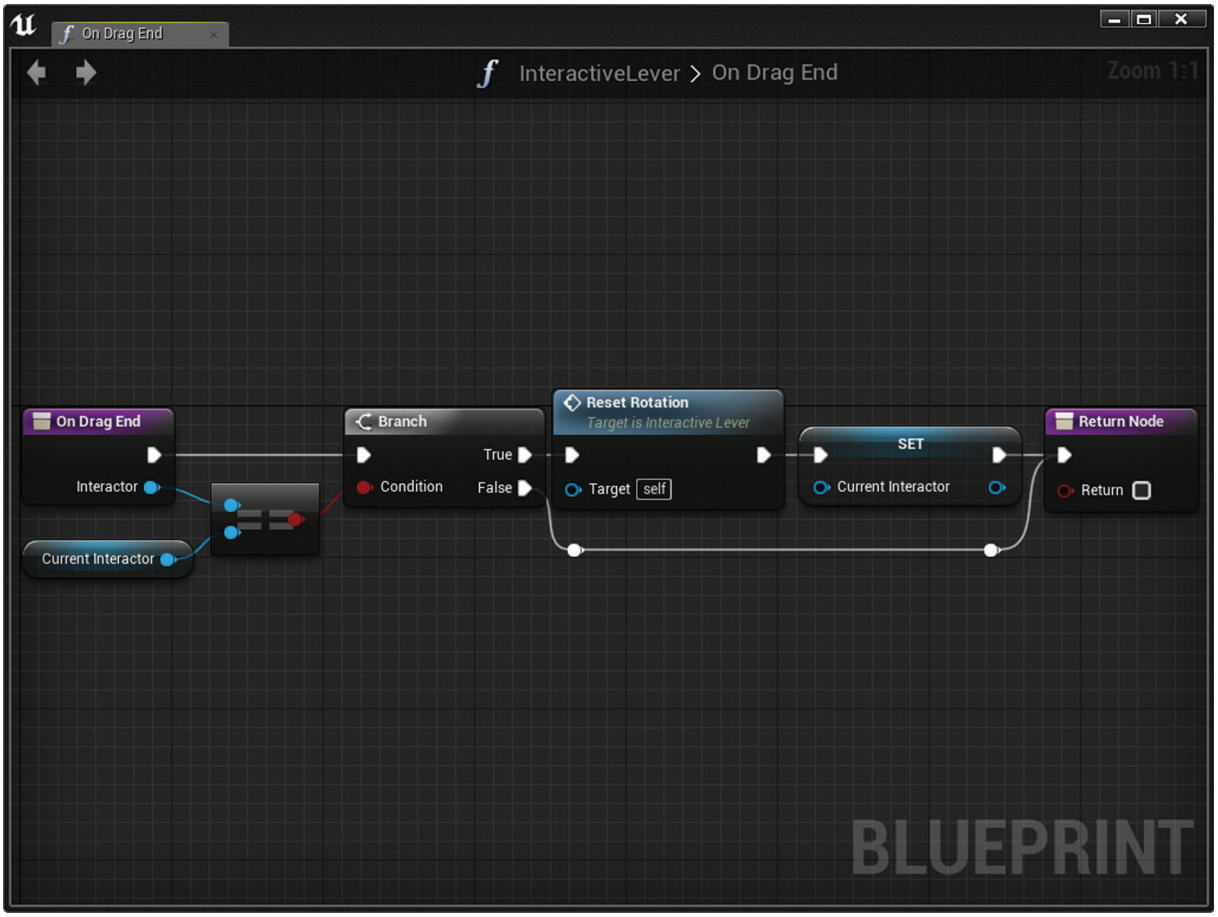

Рис. 8.25. Интерактивный рычаг: возврат рычага при прекращении тяги

24. От *Interactor* вызовите *Equal*, установив во второй вход геттер *Current Interactor*.
25. Соедините выходное значение с *Condition*. Это не даст пользователям возможностей тянуть рычаг разными объектами взаимодействия, кроме того, который взаимодействует с ним.
26. От выхода `True` вызовите событие `ResetRotation`.
27. После события создайте новый сеттер для *CurrentInteractor* и оставьте его пустым.
28. Соедините сеттер с `Return Node`.
29. Соедините `False` с `Return Node`.

8.2.3.4. Запуск рычага

Давайте добавим завершающий штрих: вызовем *Event Dispatcher Puller*, когда он достигнет шага активации.

1. Вернитесь к функции `OnDrop`. Создайте два новых геттера для `CurrentPitch` и `ActivationPitch`.
2. После `CurrentPitch` вызовите *Equal* (*float*), установив `ActivationPitch` во второй контакт.
3. Прикрепите новый узел *Branch* после `SetRelativeRotation` и соедините выход *Equal* с *Condition* (рис. 8.26).

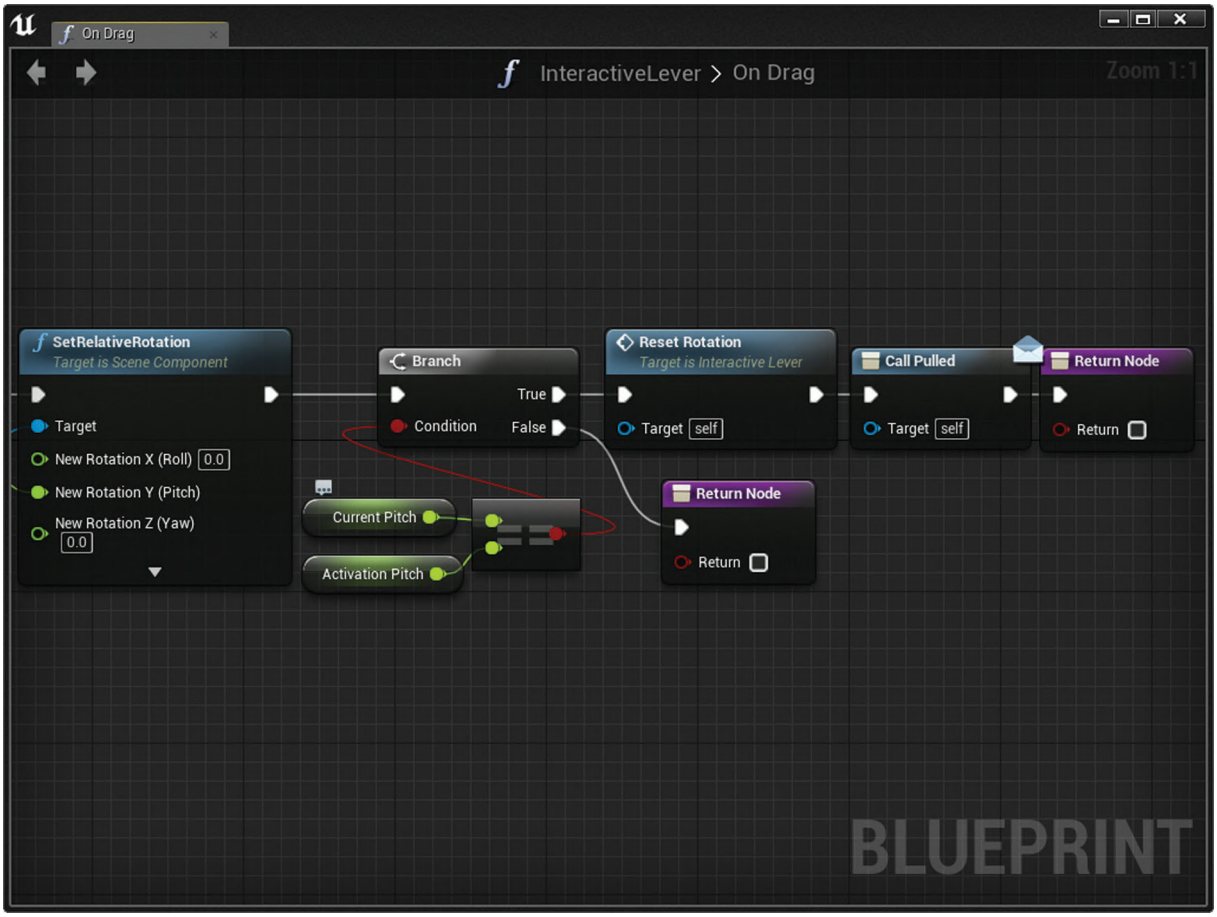

Рис. 8.26. Интерактивный рычаг: вызов *Pulled*

4. От выхода `True` вызовите событие `Reset Rotation`.
5. После него вызовите *Pulled*.
6. Прикрепите контакт выполнения *Pulled* и `False` из *Branch* с `Return Node`.

Вот мы и создали рычаг, который можно тянуть и активировать. Для подключения *Pulled* диспетчера, вы можете сделать то же самое, что и с кнопкой, и добавить его на уровень, а затем в *level Blueprint* привязать к событию тяги (рис. 8.27).

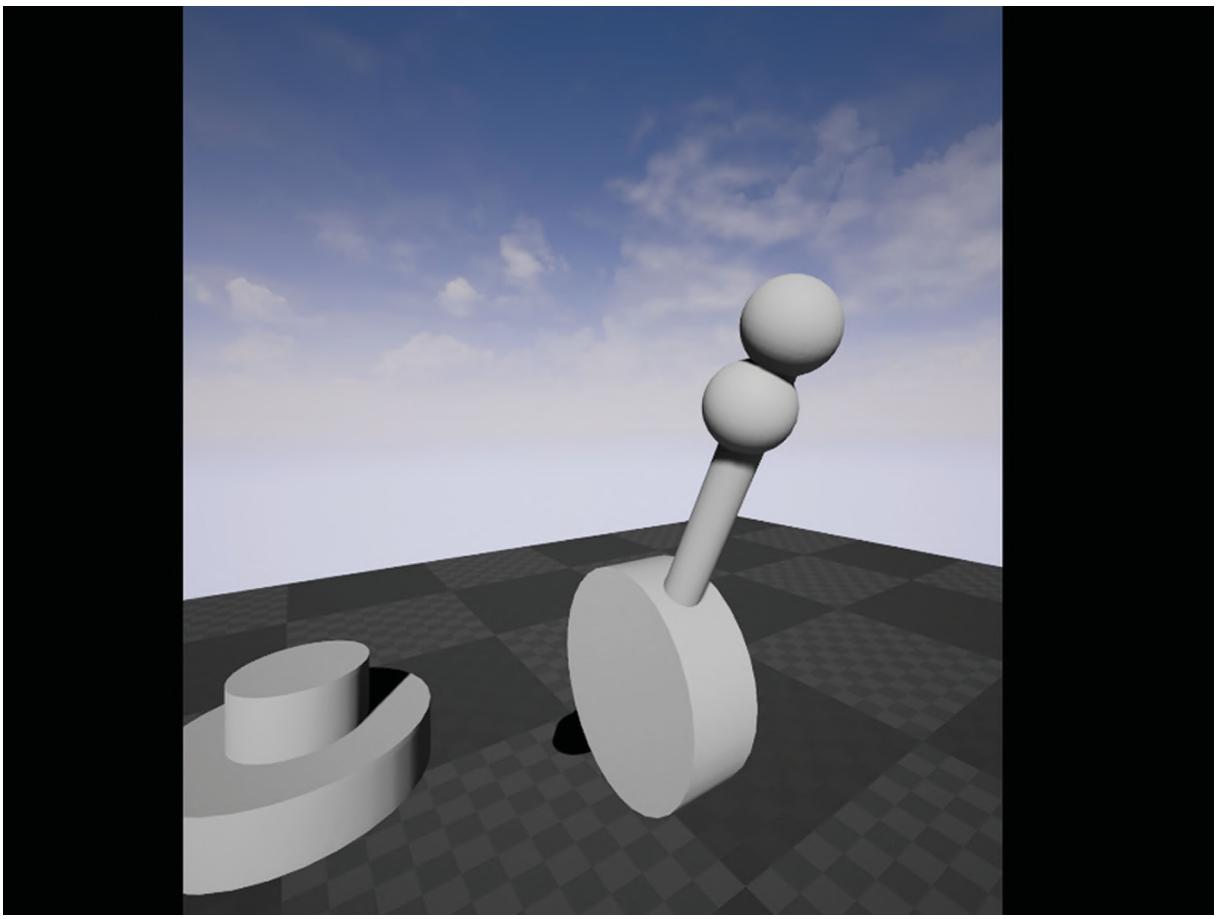

Рис. 8.27. Интерактивный рычаг

8.3. Заключение

В этой главе мы рассмотрели создание системы взаимодействия с объектами в виртуальном мире при помощи контроллеров. Кроме того, мы рассмотрели механизм работы объектов, которые часто встречаются в *VR*-играх и сюжетах, например кнопок и рычагов.

8.4. Упражнения

Рассмотрим следующие расширения для интерактивных объектов.

1. Добавьте эффект силовой обратной связи при взаимодействии с компонентами. Это может дать игроку достоверные ощущения физического присутствия виртуальных объектов.
2. Добавьте возможность для отдельных объектов переопределить привязку падения по умолчанию, так что игрок не должен держать кнопку захвата непрерывно для важных объектов. Один из способов сделать это — создать переопределенный *Actor* со специфической логикой.
3. Создайте собственные методы взаимодействия. Например, комод с ящиками, панель управления или циферблат.

ГЛАВА 9

ПЕРЕМЕЩЕНИЕ В *VR*

Разработчики все еще активно экспериментируют с перемещением в *VR*. В этой главе мы познакомимся с так называемой тренажерной болезнью и некоторыми причинами, влекущими дискомфорт в *VR*. Вы также познакомитесь с простыми механиками перемещения, которые в настоящее время применяются в *VR*-играх.

9.1. Тренажерная болезнь

В отличие от усталости, вызванной физическими движениями, тренажерная болезнь [*simulator sickness*] возникает как побочный эффект расхождения восприятия движения (в нашем случае в *VR*) с физически совершаемыми действиями (то есть теми, о которых наше тело информирует мозг).

Главной системой нашего организма, отслеживающей расхождения между действительными и виртуальными движениями, является вестибулярный аппарат. Задача вестибулярного аппарата — обнаружение ускорения (по изменению давления жидкости во внутреннем ухе) и отслеживание ориентации (по изменению вектора силы тяжести) тела в пространстве. Иллюзия движения, иногда говорят иллюзия восприятия пространства [*vection*], — термин, используемый в психологии восприятия для обозначения ощущения движения, которое пользователь воспринимает только зрительно (в случае использования *VR*). Иллюзия движения является причиной разногласия зрения и вестибулярного аппарата, что и приводит к дискомфорту пользователя. Есть множество теорий, объясняющих, почему это именно так, но опыт показывает, что множество иллюзий движения приводят к серьезному дискомфорту.

Вестибулярный аппарат, распознав ускорение, обращается к зрению пользователя, которое противоречит ощущениям, что и является причиной дискомфорта. Как следствие, ясно, что чем меньше искусственного ускорения будет в *VR*-сюжете, тем комфортнее себя будет ощущать пользователь.

Оптический поток [*Optic flow*] — любая визуализация движения на вашей сцене, сигнализирующая пользователю о том, что он перемещается. Чем сильнее оптический поток, тем больше информации у пользователя о перемещении. В общем, это приводит к еще большей иллюзии движения. Одним из возможных решений проблемы может стать использование кабины (кокпита) для уменьшения иллюзии движения и тренажерной болезни за счет общего снижения оптического потока на сцене. Помощь кабины в борьбе с иллюзией движения основана также на появлении у пользователя системы координат, привязывающей его к сцене.

Кабины, очевидно, неприменимы ко всем сюжетам, поэтому для обеспечения комфортного времяпрепровождения в виртуальной реальности могут использоваться другие техники. Наиболее известные из них рассмотрены в этой главе.

Общим у тренажерной болезни и иллюзии движения является то, что их воздействию подвержена только часть людей. Поэтому, вероятнее всего, угодить всем не получится, то есть придется все-таки выявить свою целевую аудиторию и определить наиболее подходящий для нее вариант перемещения. Уход в сторону комфорта, конечно, отличная идея, но будьте осторожны: множество факторов, увеличивающих комфорт, могут снизить эффект погружения, вследствие чего некоторые устойчивые к тренажерной болезни пользователи заходят либо отключить, либо уменьшить количество опций, влияющих на комфорт, чтобы добиться более глубокого эффекта погружения.

9.2. Типы перемещений

Результатом многочисленных исследований и экспериментов *VR*-сообщества стало появление различных методов искусственного перемещения, каждый из которых имеет преимущества и недостатки. Некоторые этих методов можно приспособить различным виртуальным сюжетам, другие существенно ограничены конкретным контекстом. Из-за молодости технологии *VR*, самые популярные методы все еще активно развиваются, поэтому, вы можете их комбинировать, приспосабливая для ваших виртуальных миров.

9.2.1. Естественное перемещение

Естественные методы перемещения, как правило, точно переводят движения в реальном мире в перемещения аватара, и поэтому такое перемещение практически не вызывает тошнотворного эффекта. Расплачиваться, как правило, приходится большим количеством ограничений, в первую очередь реальным пространством игрока и пространством для отслеживания шлема. Существуют, однако, остроумные решения для работы с ограниченным физическим пространством, например, перенаправленная ходьба [*redirected walking*]. При перенаправленной ходьбе виртуальный мир двигается в том направлении, в котором игрок никогда не покинет игровое пространство (например, медленно поворачивая мир, когда игрок идет вниз по коридору), перенаправленная ходьба обманывает игрока, заставляя его думать, что он проходит большое расстояние. Также вы можете преодолеть нехватку пространства, разрабатывая ваши уровни так, чтобы игрок никогда не добирался до краев игрового мира.

Преимущества:

- низкая вероятность вызвать тренажерную болезнь.

Недостатки:

- ограничение по физическому пространству пользователя;
- ограничение пространства для отслеживания шлема.

Разновидности:

- перенаправленная ходьба (передвижение виртуального мира для сохранения игрока внутри игрового пространства).

9.2.2. Телепортация

В этом разделе вы познакомитесь с еще одним методом перемещения «точка-точка» — телепортацией. Телепортация может быть объединена с естественными движениями для увеличения расстояния за границы физического пространства пользователей. Отметим, что при телепортации у пользователя могут проявляться такие побочные эффекты, как небольшая тошнота и дезориентация в пространстве, кроме того, она подходит не для всех игровых сюжетов. Известны различные варианты телепортации, некоторые позволяют пользователям видеть точку, в которой они находятся в реальном пространстве, прежде чем произойдет телепортация. Это позволяет пользователю спозиционировать себя наилучшим образом и обеспечить максимально естественное движение по завершении телепортации. Другие варианты позволяют управлять телепортацией физического объекта (например, мяча, который должен приземлиться там, где игрок телепортируется). Такая материальность позволяет предотвратить проблему микротелепортации, при которой пользователи телепортируются несколько раз настолько быстро, что испытывают иллюзию движения, поскольку в действительности движение не мгновенно и мозг успевает ощутить ускорение.

Если вы хотите узнать о способах реализации телепортации в *UE*4, см. главу 5 «Телепортация».

Преимущества:

- позволяет игроку преодолевать ограничения реального пространства;
- вызываемая телепортацией тошнота значительно слабее, чем во многих других вариантах перемещения.

Недостатки:

- может вызвать дискомфорт;
- дезориентация;
- цепь микротелепортаций может вызывать иллюзию движения.

Разновидности:

- визуализация игрового пространства перед телепортацией;
- телепортация относительно физических тел;
- привязка к базовым точкам (позволяет пользователю телепортироваться только на определенные локации).

9.2.3. Транспортные средства

Использование транспортных средств, как и телепортация, является методом перемещения «точка-точка», но основывается на использовании виртуального транспортного средства (машины, корабля и т. п.), задающего систему координат для пользователя.

Средства передвижения позволяют уменьшить иллюзию движения за счет снижения оптического потока и окружения пользователя стационарной системой. Наличие стационарной системы отсчета, привязанной к кабине транспортного средства, также снижает тренажерную болезнь, ведь пользователи подсознательно ощущают, что мир движется вокруг них, а не они перемещаются в нем, что согласуется с сигналами вестибулярного аппарата. Очевидно, средства передвижения не могут быть использованы в любых ситуациях. Как вариант, их замещающий, допустима демонстрация пользователю временной сетки, которая даст эффект привязки подобно средству передвижения (реализовано в *UE4 VR Editor*).

Преимущества:

- снижение оптического потока приводит к уменьшению иллюзии движения;
- привязка к системе координат снижает ощущение тошноты.

Недостатки:

- не подходит для многих ситуаций.

Разновидности:

- временная сетка, позволяющая пользователю получить координатную систему, что дает эффект привязки подобно транспортному средству.

9.2.4. Физическое перемещение

Физическое перемещение является новаторским методом, и именно оно позволяет спроектировать новые, захватывающие игровые механики. Обычно физическое передвижение представляет собой некоторое физическое действие из реального мира, которое транслируется в совсем другое движение в виртуальном мире. Например, движения как лыжными палками, осуществляемые контроллерами, для движения аватара в лыжной гонке, или бег трусцой для перемещения аватара боком в игре. С большинством физических перемещений возникают проблемы, связанные с иллюзией движения от искусственных переме-

щений, но дезориентация пользователей существенно меньше, потому что они ожидают определенных ощущений, двигаясь в реальном мире, и получают некую отдачу в виртуальном.

Преимущества:

- позволяет реализовывать необычные методы передвижения, которые хорошо подходят специфичным играм;
- может уменьшить иллюзию движения, благодаря присутствию пользовательских ожиданий от движения.

Недостатки:

- вероятно, привлекателен только внешне;
- у особо чувствительных пользователей возможна тренажерная болезнь.

Разновидности:

- горные лыжи;
- скалолазание;
- полет (взмахи крыльями);
- бег;
- Ввращение мира (вращение окружающего мира пользователем вокруг себя с помощью контроллеров).

9.2.5. Искусственное перемещение

Искусственное перемещение практически не переводит реальных движений пользователя в виртуальный мир. В его основе лежит более традиционный способ получения игровых входных данных (например, игровой контроллер). Из всех представленных нами видов перемещений, искусственное сильнее прочих провоцирует тренажерную болезнь, но при этом является самым простым способом транслировать («портировать») традиционные игры в *VR*. Некоторые вариации искусственных перемещений в играх-стрелялках (шутерах) от первого лица [*first person shooters*, *FPS*] позволяют снизить иллюзию движения за счет ограничения ускорения. Чтобы получить такой результат, ускорение вращения ограничивают, полностью отключают, либо разбивают поворот на отдельные шаги.

Преимущества:

- свобода перемещения вне зависимости от физического пространства вокруг пользователя;
- простота трансляции традиционных видеоигр в *VR*.

Недостатки:

- многие игроки ощущают тренажерную болезнь.

Разновидности:

- мгновенное вращение;
- вращение мира вокруг пользователя («эффект турели»).

9.3. Реализация передвижения

Давайте реализуем некоторые из методов перемещения. Естественное перемещение уже реализовано в движке *UE*4 через отслеживание положения шлема. Поскольку телепортация очень важна для *VR*, то ей посвящена целая глава (см. главу 5). Далее в этой главе мы рассмотрим физическое перемещение и разработаем бег на месте для контроля от первого лица (*First Person* шаблона) в *UE*4, но сначала настроим его для отслеживания поворота.

9.3.1. Настройка *First Person* шаблона для отслеживания поворота

Общий метод для искусственного перемещения — это контроль от первого лица [*first person*]. К счастью, встроенный *First Person* шаблон *UE*4 реализует его, поэтому мы легко добавим комфорта для пользователей, которые в этом нуждаются.

9.3.1.1. Создание проекта

Большинство необходимого уже есть в шаблоне *UE*4 с поддержкой *VR*. Однако давайте убедимся, что он полностью совместим. Вам необходимо применить некоторые изменения перед реализацией механизма поворота.

1. Создайте *Blueprint* проект на базе шаблона *First Person Template*.
2. Откройте *FirstPersonCharacter Blueprint* (в меню выберите *FirstPersonBP* ⇒ *Blueprints*).
3. Добавьте новый *Scene Component* под названием `CameraRoot`.
4. Установите его позицию на `-96` градусов по оси `Z` (рис. 9.1). Это гарантирует, что камера основана на точке соприкосновения персонажа с землей.

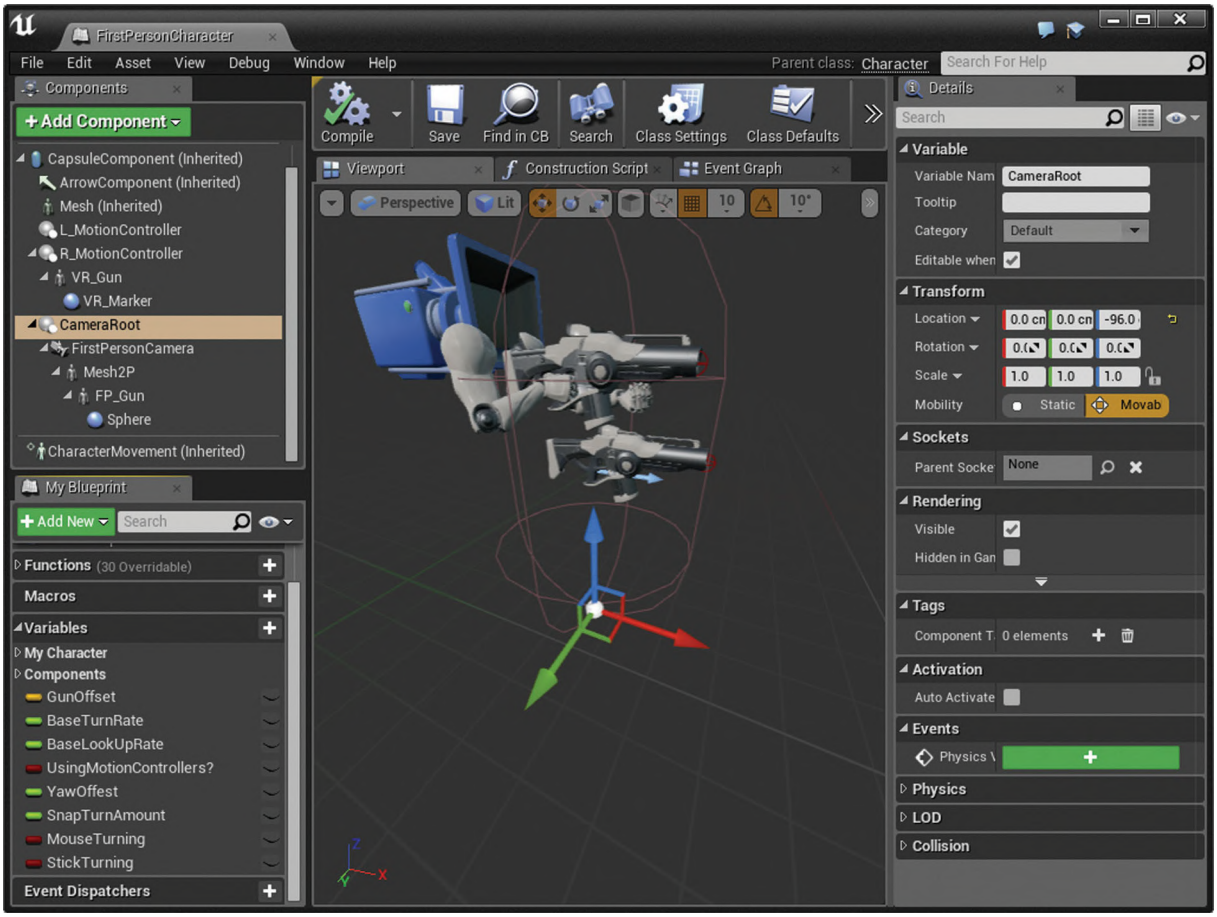

Рис. 9.1. *FirstPersonCharacter*: добавление *CameraRoot*

5. Перенесите *FirstPersonCamera* на `CameraRoot`, чтобы сделать его наследником.
6. Создайте две переменные `Float`: `YawOffset` и `SnapTurnAmount`.
7. Установите начальное значение на `15`. Это число градусов, на которое следует совершить поворот.
8. Создайте две переменные типа `Boolean`: `MouseTurning` и `StickTurning`. Они будут говорить нам, повернул ли пользователь мышкой или геймпадом.
9. В *Event Graph* найдите узел *EventBeginPlay* и затем, после узла *UseControllerRotationYaw* создайте геттер для *FirstPersonCamera Component* и вызовите `SetUsePswnControlRotation`. Установите `False` в нем, чтобы позволить персонажу использовать функции отслеживания положения шлема.
10. Создайте другой узел под названием `SetTrackingOrigin`, установив `Origin` на `Floor Level`. Это устраняет любые проблемы отслеживания со шлемами, которые по умолчанию устанавливают точку отсчета на уровень глаз.
11. Соедините два последних созданных узла между *UseControllerRotationYaw* и *Branch* (рис. 9.2).

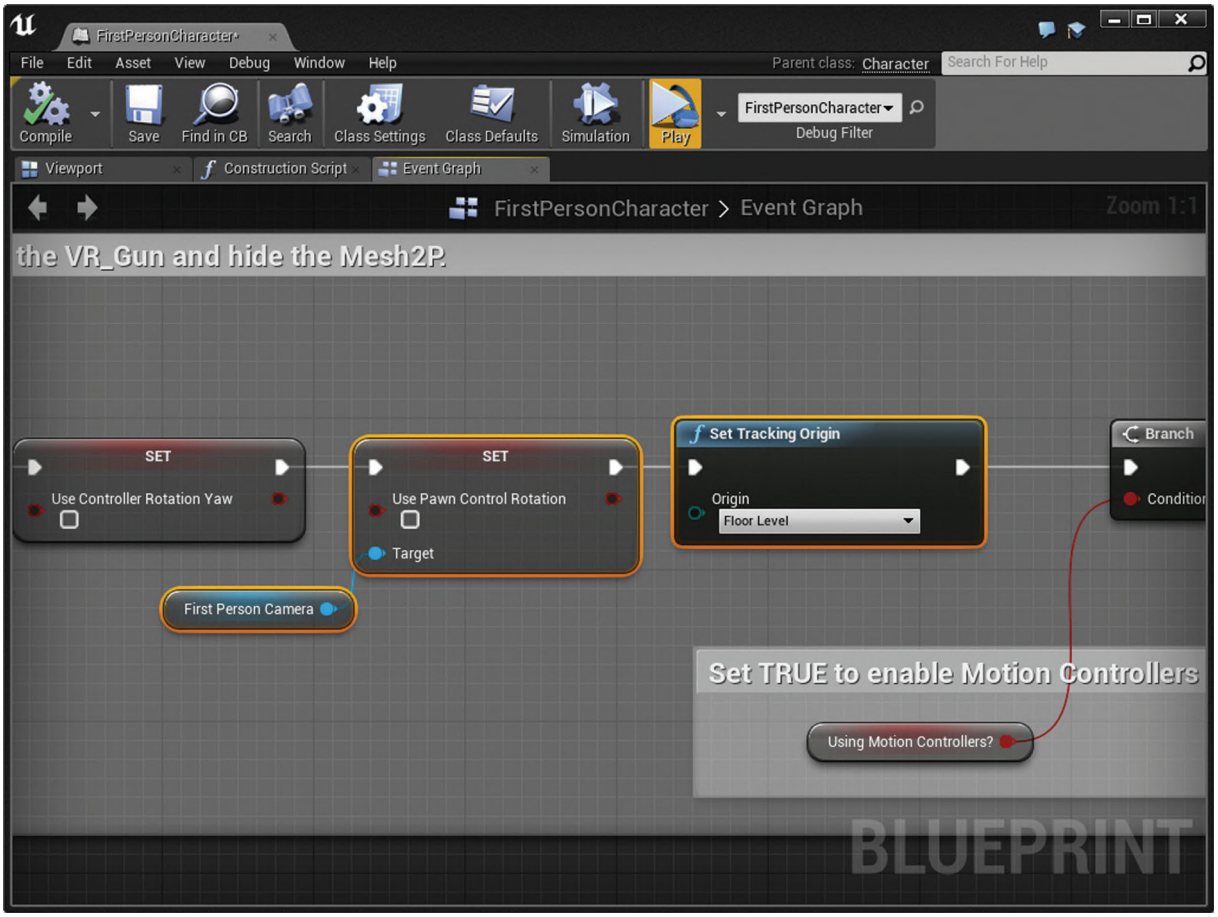

Рис. 9.2. *FirstPersonCharacter*: исправление отслеживания позиции и принуждение к новому отслеживанию

9.3.1.2. Реализация поворота

Чтобы реализовать поворот, необходимо создать новую функцию, которая позволит поворачивать камеру. Для этого используйте переменную *YawOffset* для хранения расстояния, на которое следует повернуть аватар, если используется нормальный метод поворота, и когда оно превышает пороговое значение, осуществить поворот.

1. Создайте функцию `AddControllerYaw` с двумя входными параметрами типа `Float`. `Float` и `SnapAmount` (рис. 9.3). Откройте ее.

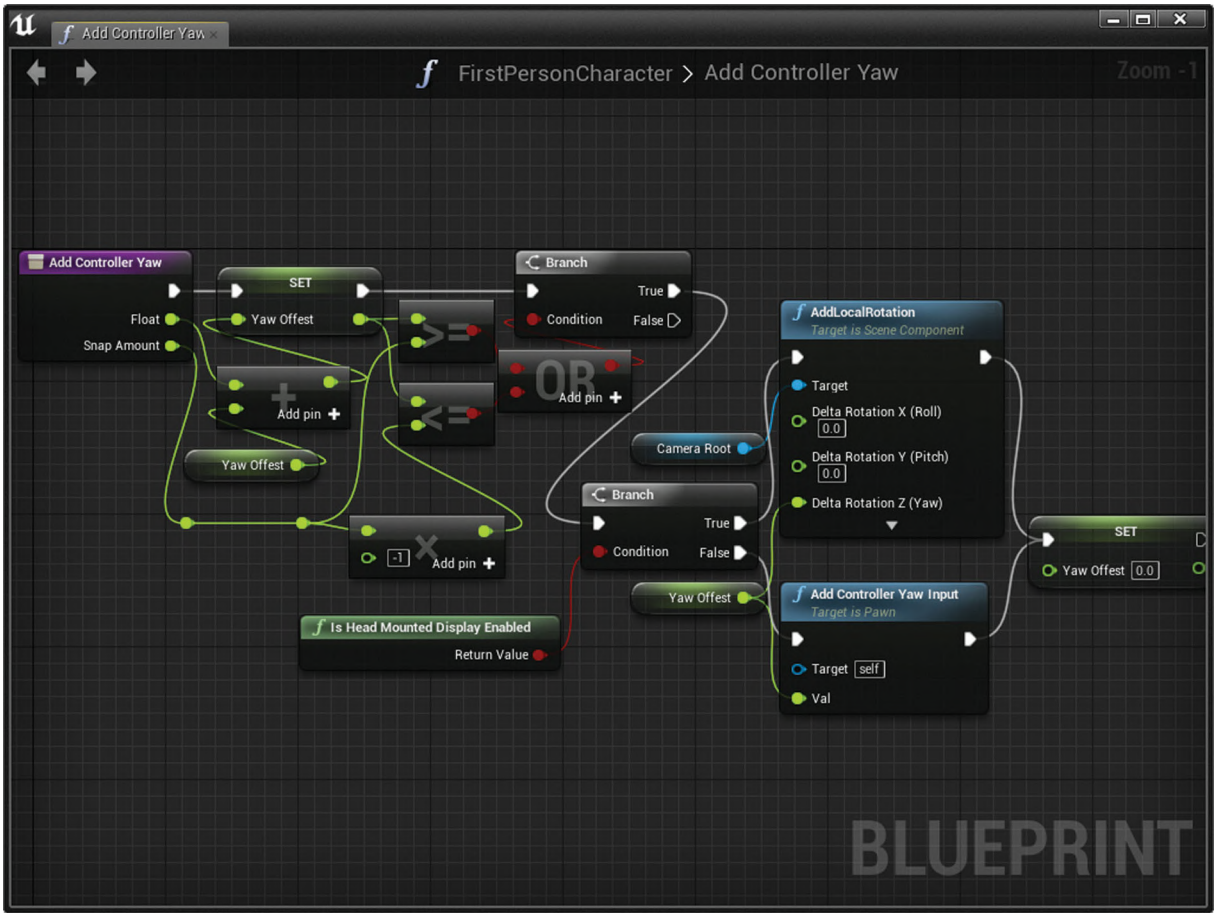

Рис. 9.3. *FirstPersonCharacter*: настройка функции поворота

2. Создайте новый геттер и сеттер для *YawOffset*, соединив контакт выполнения сеттера с началом функции.
3. От входного параметра `Float` вызовите функцию `Float + Float`, установив во второй вход *YawOffset*.
4. Соедините результат этой суммы с сеттером *YawOffset*.
5. После сеттера вызовите функцию `Float >= Float`, установив `SnapAmount` как второй параметр.
6. От сеттера вызовите еще одну функцию `Float <= Float`, в этот раз установив `Float * Float` между `SnapAmount` и `–1` во второй вход (рис. 9.3).
7. Соедините выход обоих функций с `OR`. Это означает, что остальное вызывается, только если входной параметр больше поворота или меньше его отрицательной величины.
8. Создайте новый узел *Branch*, соединив *OR* с *Condition*.
9. Соедините контакт выполнения *Branch* с сеттером *YawOffset*.
10. После `True` создайте еще один узел *Branch*, соединив вызов `MountDisplayEnabled` с *Condition* (рис. 9.3).

11. Создайте новый геттер `CameraRoot` и вызовите от него *AddLocalRotation*.
12. Разбейте вход *Delta Rotation* и установите *YawOffset* в `Z`. Это позволит вращать компонент. При этом вы не сможете поворачивать камеру, используя *VR*-шлем.
13. Соедините *AddLocalRotation* с веткой `True`.
14. Создайте новый сеттер для *YawOffset*, оставив в этот раз `0.0` в значении, и соедините его с концом *AddLocalRotation*.
15. От ветки `False` вызовите *AddControllerYawInput*, установив *YawOffset* в *Val* и соедините его выполнение также с сеттером *Yaw Offset*.
16. Вернитесь к *Event Graph*, найдите комментарий *Stick input* и замените вызов *AddControllerYawInput* на ваш метод *AddControllerYaw*, установив *SnapTurnAmount* в `Snap Amount` (рис. 9.4).

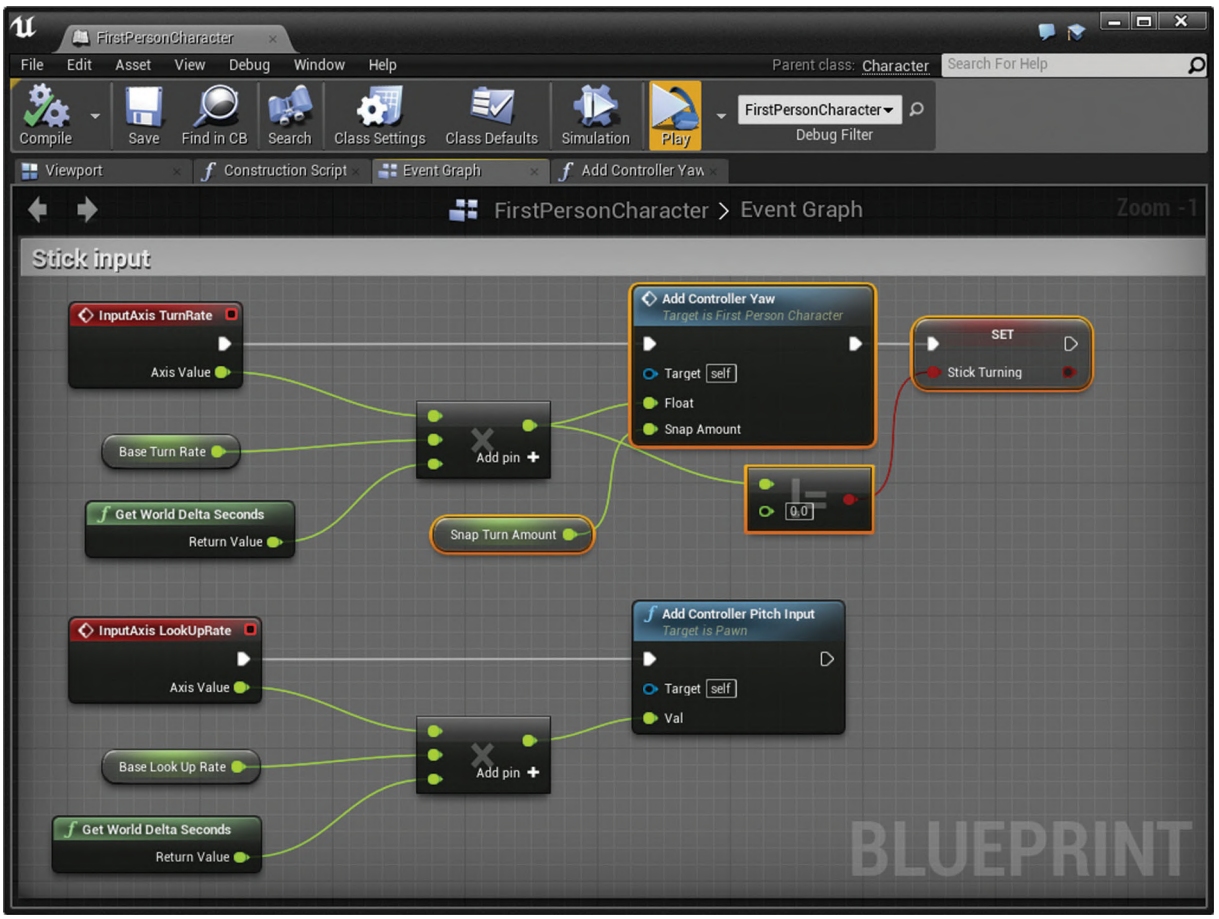

Рис. 9.4. *FirstPersonCharacter*: замена дефолтного метода на новый

17. От перемножения трех элементов вызовите узле `Float != Float` и установите его результат в новый сеттер для *StickTurning*, который поставите после *AddControllerYaw* (рис. 9.3).
18. Найдите комментарий *Mouse input* и замените *AddControllerYawInput* снова на наш метод, установив *SnapTurnAmount* в `Snap Amount` (рис. 9.5).

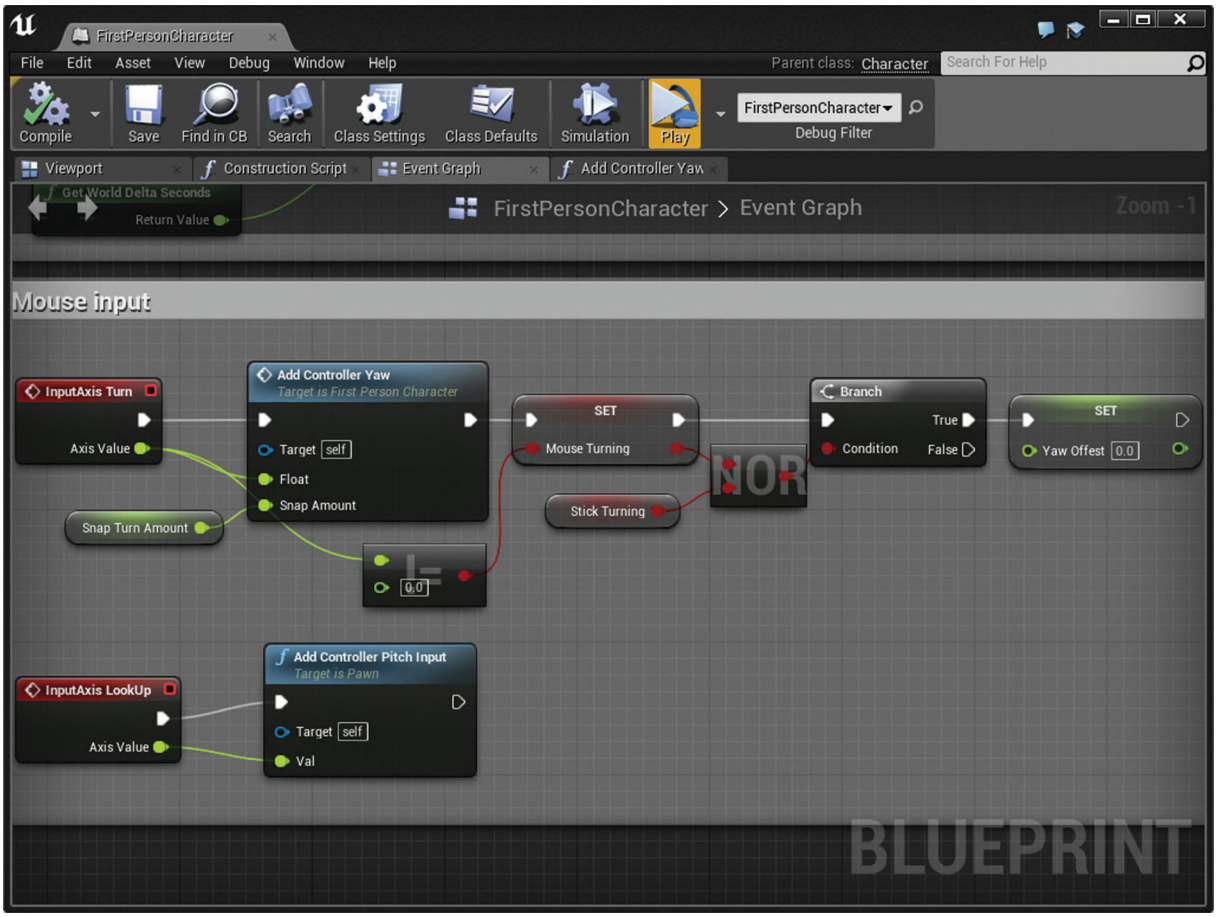

Рис. 9.5. *FirstPersonCharacter*: замена метода поворота камеры по умолчанию

19. От параметра `Axis Value` вызовите `Float != Float` и установите его результат в новый сеттер для *MouseTurning*, который поставите после *AddControllerYaw*.
20. От красного выхода сеттера вызовите `NOR`, установив во второй вход *StickTurning*.
21. Создайте *Branch* после сеттера из пункта 19, установив в *Condition* результат `NOR`. Это позволит вам обнулить *YawOffset*, когда игрок не использует ни мышь, ни геймпад.
22. От ветки `True` создайте новый сеттер *YawOffset*, оставив в значении `0.0` (рис. 9.3).
23. Чтобы добавить поддержку и использовать плечевые кнопки геймпада для поворота игрока, создайте два новых события `GamepadRightShoulder` и `GamepadLeftShoulder` (см. рис. 9.6).

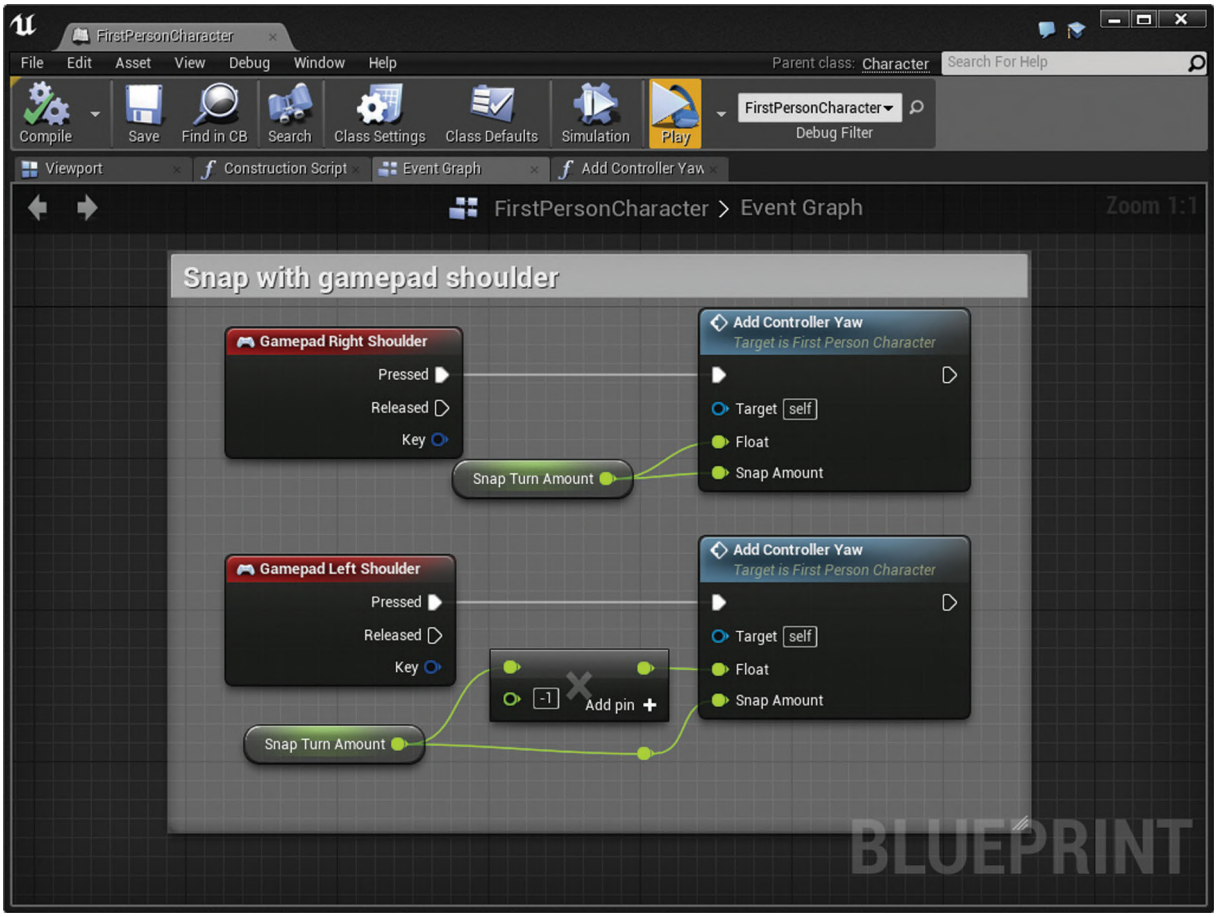

Рис. 9.6. *FirstPersonCharacter*: поворот с использованием плечевых кнопок геймпада

24. После события `GamepadRightShoulder` вызовите *AddControllerYaw*, установив `SnapTurnAmount` в `Float` и `Snap Amount`.
25. После события `GamepadLeftShoulder` вызовите *AddControllerYaw*, установив `SnapTurnAmount` в `Snap Amount` и умножив его на `-1` перед этим во `Float`.

9.3.2. *First Person* шаблон для реализации бега на месте

Простейшая система физического перемещения — это реализация бега на месте. Предполагается, что пользователь в шлеме «трусит» на месте, стоя в одной точке, а его аватар при этом перемещается вперед в виртуальном мире.

Для реализации бега мы продолжим работать с модернизированным *First Person* шаблоном, к которому до этого добавили мгновенное вращение.

1. Откройте *FirstPersonCharacter Blueprint* (*FirstPersonBP* ⇒ *Blueprints*).
2. Создайте две новые переменные типа `Vector` с именами `HMDVelocity` и `PreviousHMDPostion`. Они будут хранить текущую скорость *VR*-шлема и позицию головы на предыдущем кадре, необходимую для вычисления скорости.

3. Создайте еще одну переменную типа `Vector`, на этот раз в виде массива `Array` с названием `PreviousVelocities`. Вектор предыдущих значений скорости будет использоваться для фильтрации высокочастотных колебаний скорости из-за дрожания головы пользователя.
4. Создайте три новые переменные типа `Float`: `RunningSpeed`, `RunningThreshold` и `RunningMultiplier`. `RunningSpeed` — текущая скорость, связанная с перемещениями головы пользователя, `RunningThreshold` — конкретное значение, которое должно быть превышено в `RunningSpeed`, чтобы аватар переместился, `RunningMultiplier` — для изменения скорости движения головы аватара и нужна, чтобы иметь возможность настраивать скорость в зависимости от расстояния перемещения (рис. 9.7).

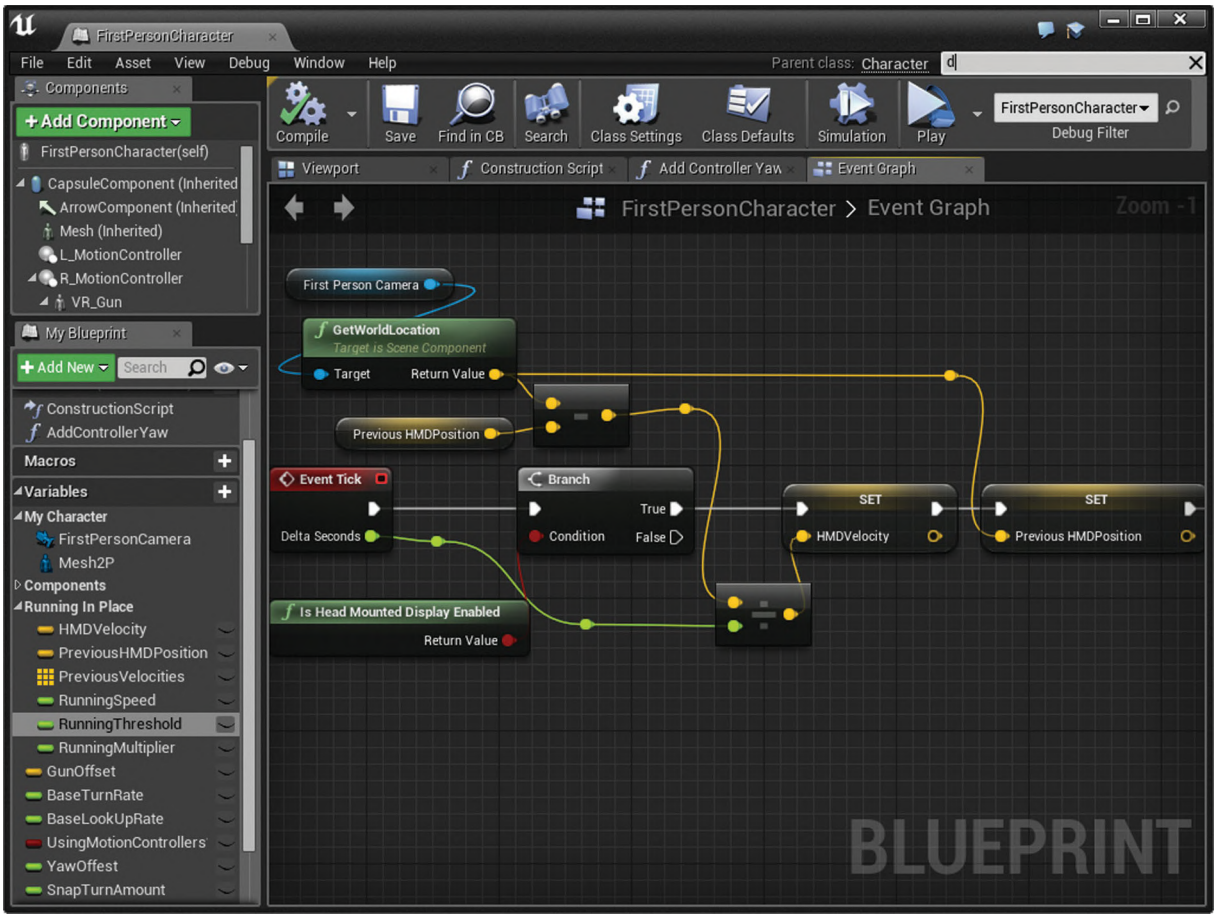

Рис. 9.7. *GirstPersonCharacter*: переменные для бега на месте и расчет скорости

5. Установите для `RunningThreshold` и `RunningMultiplier` значения по умолчанию, равные `1.5` и `0.5` соответственно.
6. В *Event graph* от *EventTick* создайте *Branch*, соединив новый вызов функции `IsHeadMountedDisplayEnabled` в *Condition*.
7. От ветки `True` вызовите сеттер `HMDVelocity`.
8. Создайте новый геттер для `FisrtPersonCamera` и вызовите `GetWorldLocation`.

9. От выхода этого узла создайте узел `Vector – Vector`, установив геттер `PreviousHMDPosition` во второй контакт, чтобы вычислять расстояние между предыдущим и текущим значением `HMD`.
10. Соедините выход вычисления разности из п. 9 с вызовом `Vector/Float`, установив во второй контакт `Delta Seconds` из *Event Tick* (рис. 9.3), чтобы из разности `Vector – Vector`, выраженной в сантиметрах, получить значение скорости в сантиметрах в секунду.
11. Соедините выход `Vector/Float` с сеттером `HMDVelocity`.
12. Создайте новый сеттер для `PreviousHMDPosition` после `HMDVelocity` и установите в него значение из `GetWorldLocation` шага `8`.
13. Теперь создайте новый геттер для массива `PreviousVelocities` и вызовите функцию `Insert`.
14. Создайте новый геттер `HMDVelocity` и установите его в `Insert`. Это добавит текущую скорость в первую секцию массива.
15. Создайте новый геттер для *PreviousVelocities* еще раз и вызовите `Resize`, установив `5` во второй контакт (рис. 9.8). Это приведет к тому, что массив будет содержать последние пять кадров.

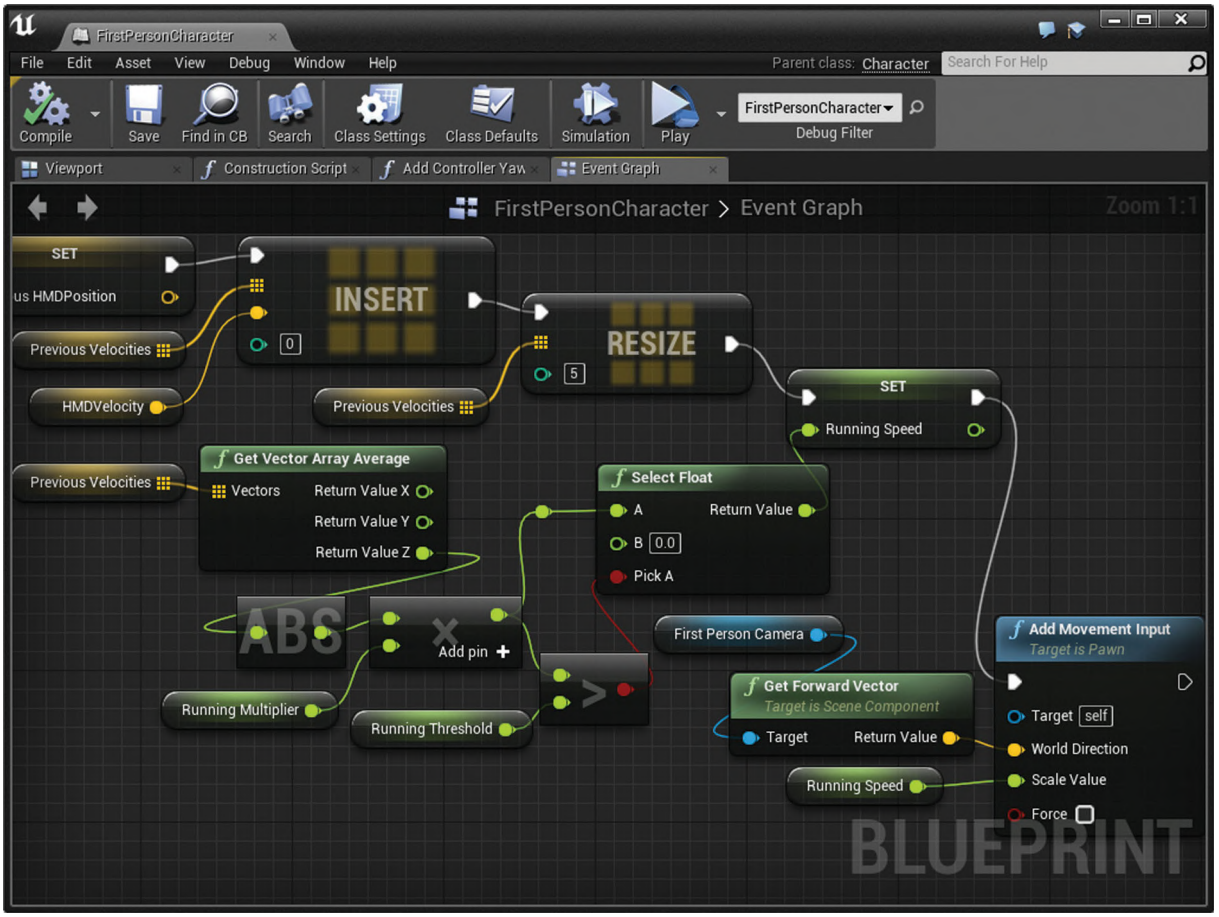

Рис. 9.8. *FirstPersonCharacter*: хранение пяти кадров для бега на месте и добавление ввода

16. Создайте еще один геттер для *PreviousVelocities* и вызовите `GetVectorArrayAverage`.
17. Правым кликом по выходному значению разделите его.
18. От координаты `Z` вызовите `Absolute (float)`; это позволит вам игнорировать направление бега трусцой.
19. От выхода `Absolute` вызовите `Float * Float`, установив во второй контакт новый геттер для *RunningMultiplier*.
20. После него вызовите *SelectFloat*. Таким образом, мы можем быть уверены, что голова движется быстрее порогового значения.
21. От `Float * Float` создайте еще один узел `Float > Float`, установив `RunningThreshold` во второй контакт входа.
22. Установите выходное значение `Float > Float в Pick A` узла *SelectFloat*; это гарантирует, что скорость бега будет `0`, пока скорость головы игрока находится ниже порогового значения.
23. Создайте новый сеттер для *RunningSpeed* и соедините вызов с `Resize`.
24. Установите *SelectFloat* в этот сеттер.
25. Для фактического перемещения игрока, после сеттера вызовите *AddMovementInput*.
26. Создайте новый геттер для *FirstPersonCamera* и получите его *ForwardVector* (рис. 9.3), установив его в `World Direction`.
27. Создайте новый геттер для *RunningSpeed* и установите его в `Scale Value`.

Теперь вы можете подключить ваш *VR*-шлем и нажать *Play*. Если ваш *VR*-шлем поддерживает отслеживание положения, пробегитесь на месте и увидите, как двигается ваш персонаж.

9.4. Заключение

В этой главе вы узнали, что такое тренажерная болезнь и в результате чего она может возникать. Вы также познакомились с существующими сейчас методами перемещения в *VR* и способами их реализации в *UE*4.

9.5. Упражнения

Мы не рассмотрели на практике использование транспортных средств для перемещения. И тому есть объяснение: в *Epic* есть отличный шаблон вождения, который полностью поддерживает *VR*. Создайте новый проект с этим шаблоном, поэкспериментируйте и проверьте, помогает ли сниженный оптический поток бороться с тренажерной болезнью.

ГЛАВА 10

ОПТИМИЗАЦИЯ *VR*

В этой главе рассматриваются требования к устройствам рендеринга и отображения для поддержки *VR*. Эти требования гораздо выше технических требований, предъявляемых к традиционным дисплеям, но некоторые хитрости могут облегчить нагрузку на оборудование в этом новом медиапространстве.

10.1. Технические требования для рендеринга *VR*

Рендеринг *VR*-сцен ставит много новых проблем и задач, не возникающих при рендеринге традиционных сцен или игр. У *VR* есть строгая необходимость в рендеринге с малой задержкой, чтобы избежать тренажерной болезни; однако в прошлом этой части рендеринга часто не уделяли достаточно внимания из-за бесконечной погони за все большей и большей общей производительностью графического движка.

У идеального *VR*-сюжета время *motion-to-photon* (время между обнаружением движения сенсором и отображением его пользователю) должно быть менее 20 миллисекунд. Это строгое требование привело к использованию ЖК-дисплеев с высокой частотой обновления (от 60 Гц до 90 Гц и 120 Гц). Однако недостаточно просто иметь панель с указанными характеристиками. Поскольку дисплей *VR*-шлема расположен очень близко к глазам пользователя, у панели должно быть достаточно большое разрешение, чтобы оно не мешало восприятию виртуального мира. Поэтому у дисплеев *VR*-шлемов частота обновления кадров от 60 Гц до 120 Гц и разрешение от 1920×1080 до 2560×1440. Эти цифры довольно высоки с точки зрения современных игроков, но они не отражают все сложности проблем и решений.

Чтобы осознать насколько требовательным может быть рендеринг в *VR*, вы должны понимать принцип работы *VR*-линз. Кратко: для фокусировки объектов в реальном мире, мышцы глаз толкают и притягивают хрусталик, позволяя определенным лучам света собираться на сетчатке. Однако эти мышцы не могут долго фокусировать взгляд на близко расположенных объектах. К сожалению, это плохо с точки зрения использования *VR*-шлемов, потому что для получения большого угла обзора [*FOV*] дисплей должен находиться близко к глазам. Для обхода этого противоречия можно расположить линзы между глазами пользователя и дисплеем. Линзы отклоняют лучи света от дисплея и «обманывают» глаза, заставляя нас думать, что экран находится гораздо дальше, чем на самом деле, тем самым решая проблему фокусировки. Другими словами, каждая линза «выпрямляет» лучи от глаза до некоторого объекта, обеспечивая ощущение оптического фокуса на бесконечности. Но эти линзы не идеальны и заметно искажают поступающую с экрана картинку. Для предотвращения этого вы должны рендерить вашу виртуальную картину с учетом этого искажения («бочки» [*barrel distortion*]), чтобы ваш мир казался настолько точным, насколько это возможно (рис. 10.1).

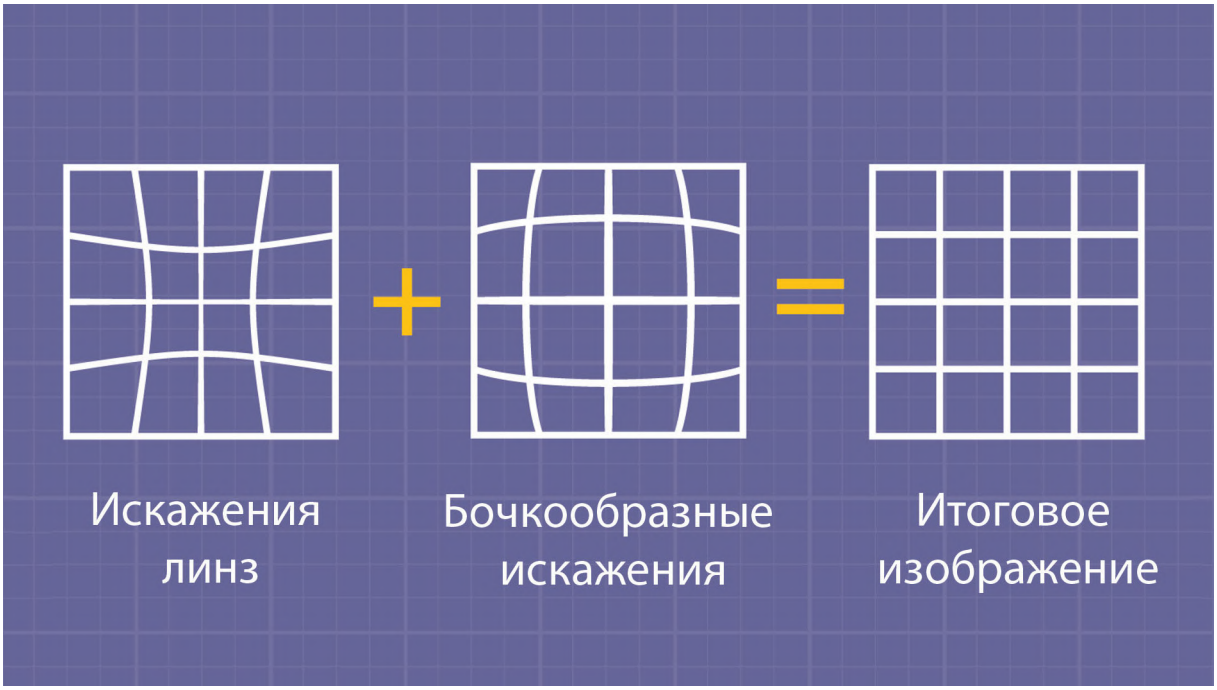

Рис. 10.1. Коррекция искажения *VR* линз

При том что коррекция искажения позволяет решить проблему фокусировки, искажение «бочки» приводит к уменьшению воспринимаемого разрешения дисплея. Чтобы избежать потери качества, многие *VR*-игры рендерят картинку с разрешением в 1,3–1,5 раза больше, чем «родное» разрешение экрана. В *UE*4 введено понятие относительного разрешения рендеринга [*screen percentage*] в процентах, и это очень важный параметр при оптимизации качества и производительности. Относительное разрешение может быть как меньше 100% (для повышения частоты кадров), так и больше 100% (для повышения качества за счет избыточной выборки, или суперсэмплинга [*supersampling*]).

Поэтому первоначальные технические требования для рендеринга в *VR* нужно умножить в полтора раза. Таким образом, для *Rift CV*1 и *Vive* (у обоих шлемов два экрана с разрешением 1800×1200 каждый), общее разрешение рендеринга получается 2160×1200×1,5, а именно 3200×1800 при 90 *fps*, или, другими словами, 350 млн пикселей в секунду! В противовес этому, в 2*D*-игра с разрешением 1080*p* при 60 *fps* требует обработки всего лишь 124 млн пикселей в секунду.

10.2. Уменьшение задержки

Как мы уже убедились, увеличение задержки обновления изображения в *VR*-шлеме имеет исключительно негативные последствия для пользователя, ставя под сомнение убедительность взаимодействия с виртуальной реальностью. Требование минимизации задержки вывода намного важнее для *VR*, чем для других способов визуализации. К счастью, существуют различные методы, уменьшающие или даже устраняющие основные причины задержек в конвейере рендеринга. В этой главе рассматриваются два основных способа борьбы с задержками: трансформация шкалы времени [*timewarping*] и рендеринг в передний (кадровый) буфер [*front buffer rendering*].

Прежде чем мы перейдем к тому, что такое трансформация шкалы времени (или перепроецирование) и рендеринг в передний (кадровый) буфер, давайте быстро пройдемся по конвейеру рендеринга, чтобы лучше понять, как эти возможности сочетаются с предыдущим опытом. (Хотя прикладные разработчики не сосредоточены на особенностях рендеринга и им редко приходится иметь дело с этими подробностями, поскольку *UE4* скрывает большинство деталей реализации, по-прежнему важно знать эти нюансы, как минимум для того, чтобы правильно диагностировать возникающие проблемы.)

Для построения одного кадра игры несколько компонентов работают сообща, чтобы создать окончательное изображение. Центральный процессор (ЦП или *CPU*) опрашивает пользовательский ввод, совершает игровые действия (обсчитывает физическую модель мира, шаги искусственного интеллекта и т. д.) и в итоге отправляет вызовы отрисовки, которые будут выполняться на видеочипе (*GPU*). Традиционно после того, как *GPU* закончил рендеринг этих вызовов отрисовки, он отправляет их на дисплей для сканирования, который затем (построчно слева направо, сверху вниз) переводит изображение на экран для пользователя. рис. 10.2 иллюстрирует это.

> **Заметка**
>
> Большинство современных *VR*-шлемов реализуют технологию *low-persistence*, которая позволяет значительно сократить размытость изображения при движении головой. Смысл заключается в том, чтоб задержать вывод изображения до момента за доли секунды до следующего обновления дисплея, чтобы предотвратить показ пользователю «плохой» (рваной/размытой) картинки. Тем не менее эта книга для упрощения предполагает работу с традиционным дисплеем.

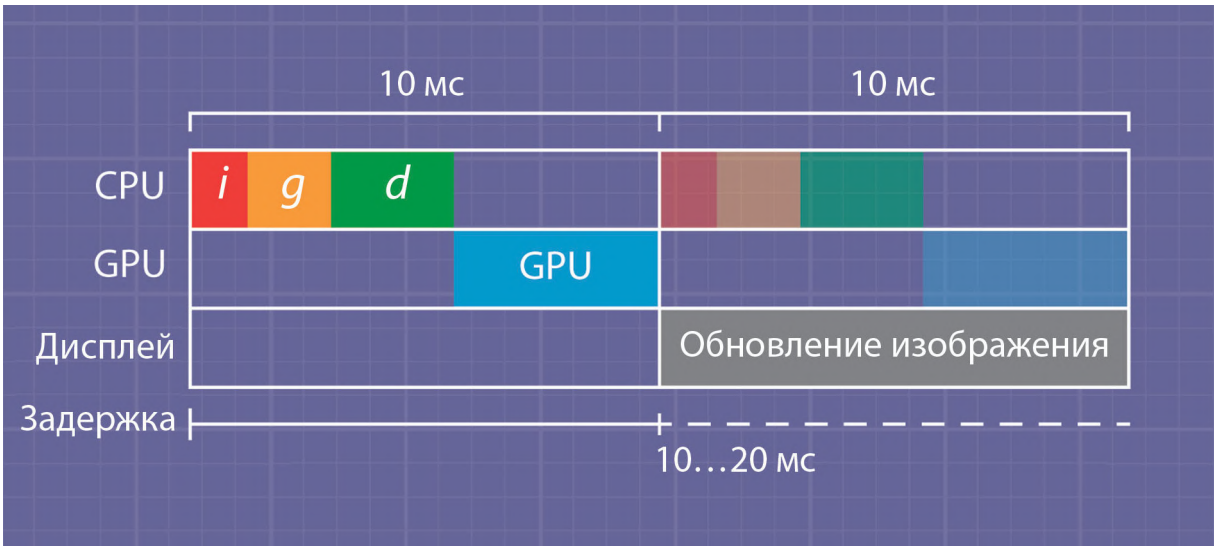

Рис. 10.2. Упрощенное представление конвейера рендеринга одного кадра; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Обратите внимание, что рис. 10.2 является не самым оптимальным вариантом настройки с точки зрения производительности. Отметим длительные периоды, когда *GPU* или *CPU* ничего не делают, ожидая окончания работы друг друга. Также отметим целый кадр задержки от ввода (*i*) до обновления изображения.

Чтобы устранить эти недостатки, мы можем для начала попробовать убрать паузы в работе процессоров путем введения параллельной конвейерной архитектуры рендеринга (рис. 10.3).

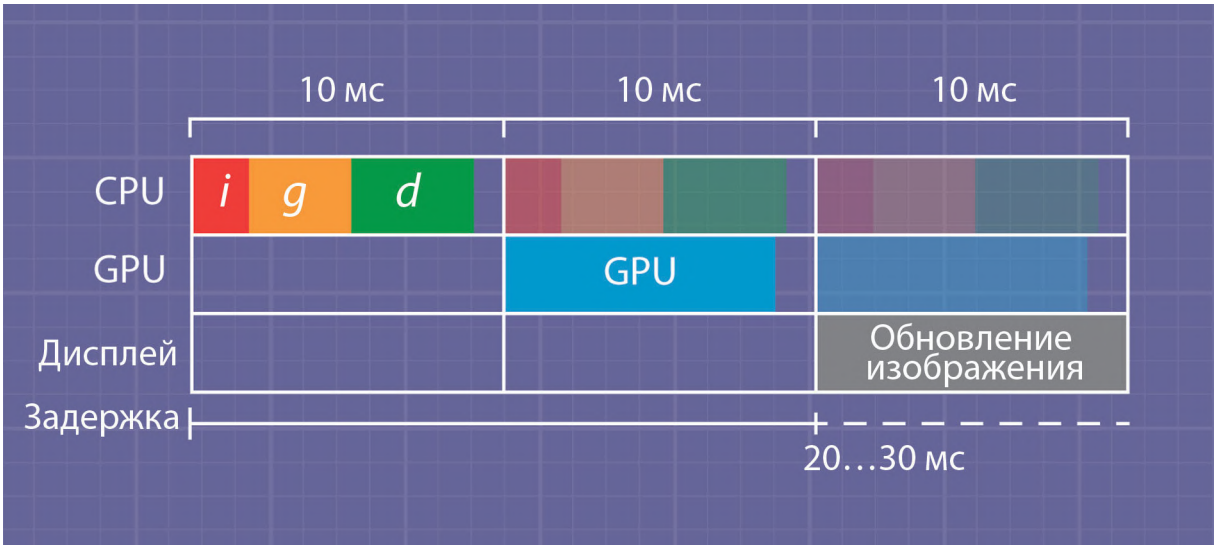

Рис. 10.3. Упрощенное представление конвейера рендеринга с двумя кадрами задержки и новой параллельной архитектурой; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

По сравнению с предыдущим рисунком, рис. 10.3 демонстрирует лучшую загрузку *CPU* и *GPU*; у них нет паузы в 10 мс, потому что графический процессор теперь отображает предыдущий кадр, в то время как процессор начинает работу над следующим кадром. Однако легко заметить увеличение задержки еще на целый кадр.

Другая проблема возникает, когда надоедливая задача вызова команд отрисовки начинает занимать слишком большую часть времени работы CPU и не помещается в 10 мс, отведённые на рендеринг сцены. К счастью, теперь у нас есть многоядерные процессоры, и если вы готовы смириться еще с одним кадром задержки (и на самом деле многие игровые движки идут на такой компромисс), вы можете запустить задачу на отдельном ядре, в то время как обработка пользовательского ввода и игровые вычисления выполняются на другом (рис. 10.4).

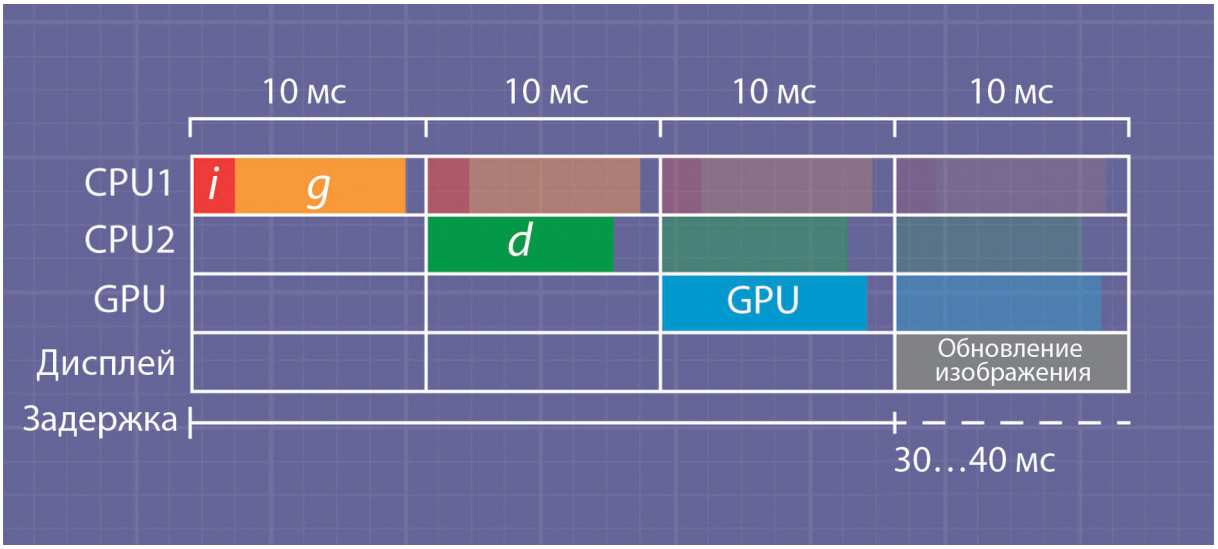

Рис. 10.4. Упрощенное представление конвейера рендеринга с тремя кадрами задержки и параллельной архитектурой с выделением кадра на задачи визуализации; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Однако оказывается, что можно пойти дальше. Вы можете уменьшить три кадра задержки (рис. 10.4) до двух, если *GPU* не будет дожидаться окончания вызова *GPU* всех команд отрисовки, а лишь предоставит центральному процессору достаточную фору, чтобы *CPU* мог начать подачу данных *GPU* с небольшой задержкой (хороший компромисс, чтобы отыграть обратно один кадр задержки; рис. 10.5).

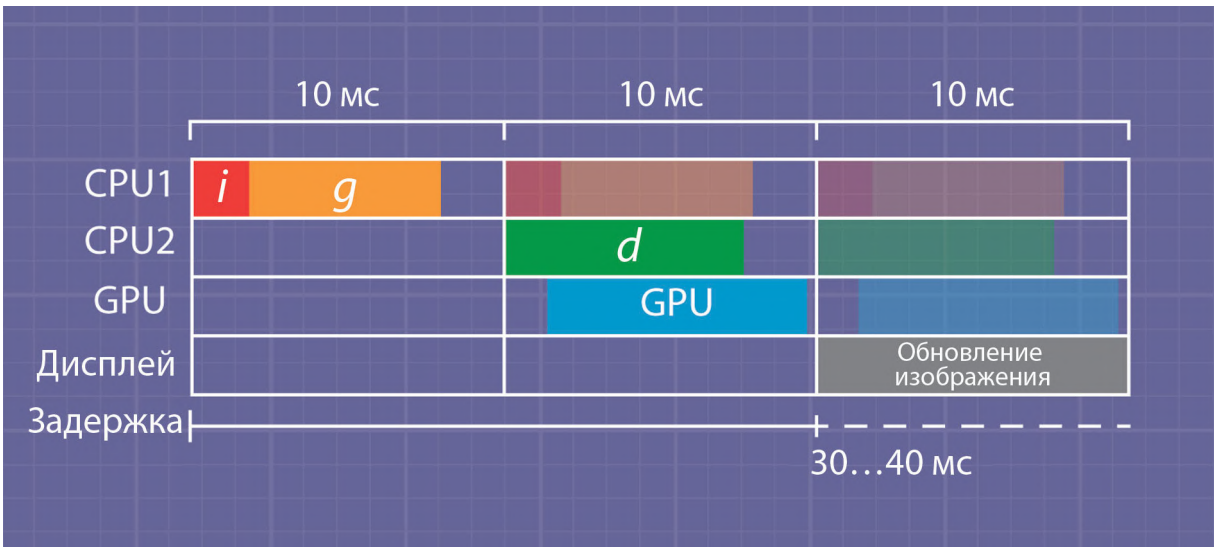

Рис. 10.5. Упрощенное представление конвейера рендеринга с тремя кадрами задержки и параллельной архитектурой с запуском отрисовки и *GPU* параллельно; *i* — ввод, *g* — игра, *d* — отрисовка. Каждый столбец — это 10 миллисекунд

Рис. 10.5 демонстрирует удачный баланс между производительностью и задержкой. Можно ли сделать лучше? Конечно! Самые внимательные заметят пузырь бездействия, когда *GPU* ждет подачи вызовов отрисовки, и, возможно, задавались вопросом, как его уменьшить. Да — как *Oculus*, так и *Valve* справляются с тем, что они называют адаптивной очередью с опережающим запуском (*Adaptive Queue Ahead and Running Start*). В принципе с точки зрения рис. 10.5 они просто перемещают задачу рисования зеленого цвета до второго кадра, чтобы дать *GPU* наибольшее время обработки всех вызовов отрисовки.

Эти принципы позволяют значительно оптимизировать использование *CPU* и *GPU*. Можем ли мы уменьшить задержку ввода? Во-первых, обратите внимание, что входные данные опрашиваются в начале каждого кадра, поэтому даже в лучшем случае, без конвейерной обработки, все еще существует целый кадр задержки. Но почему вы не можете просто переместить опрос пользовательского ввода в конец кадра? Вы можете! Фактически это именно то, что *UE*4 делает с *Camera Component* и флагом *Lock to HMD* и *Motion Controller Component* с флагом *Low Latency Update*. Этот вид позднего обновления входных данных показан на рис. 10.6.

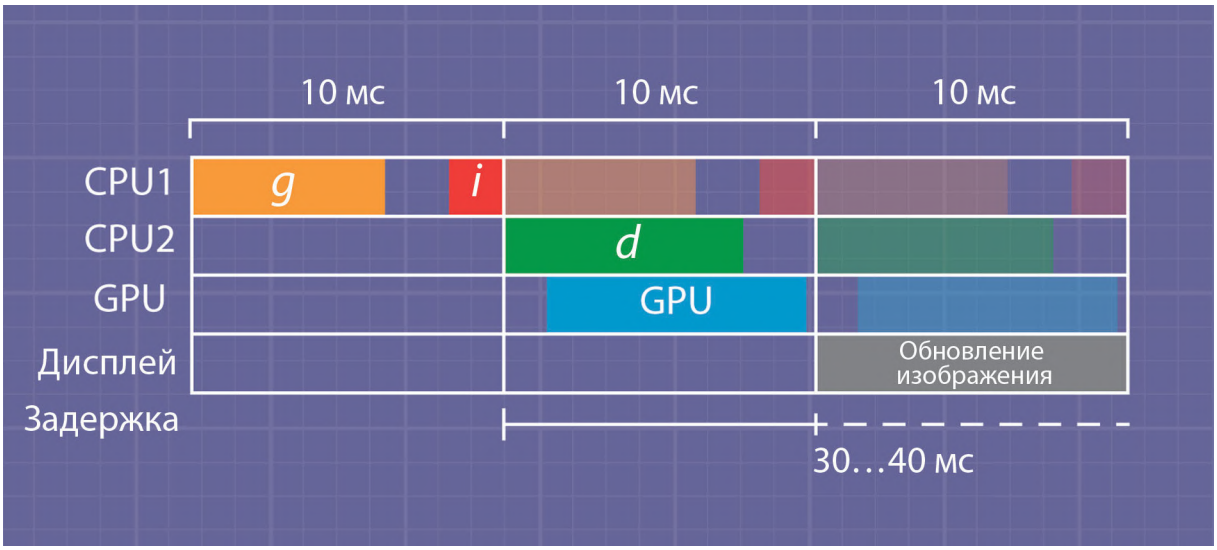

Рис. 10.6. Простая визуализация конвейера рендеринга с одним фреймом задержки и новой архитектурой с запуском отрисовки и *GPU* параллельно; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Это выглядит лучше, но есть еще одна вещь, которую вы можете сделать: временной скачок [*timewarping*]. Основная идея временн*о*го скачка состоит в том, чтобы взять данные рендеринга из *GPU*, а затем преобразовать это изображение, чтобы имитировать поворот камеры на основе нового положения шлема. Как показано на рис. 10.7, теперь можно уменьшить задержку вращения шлема всего до 2 мс для первой части рендеринга. (Обратите внимание, что не все пакеты *SDK*/среды выполнения реализуют временной скачок, а те, которые реализуют, реализуют его по-разному. Помните об этих различиях. Также обратите внимание, что эта функция включается через различные среды выполнения, а не через движок, так что разработчику не нужно ничего делать, чтобы включить его.)

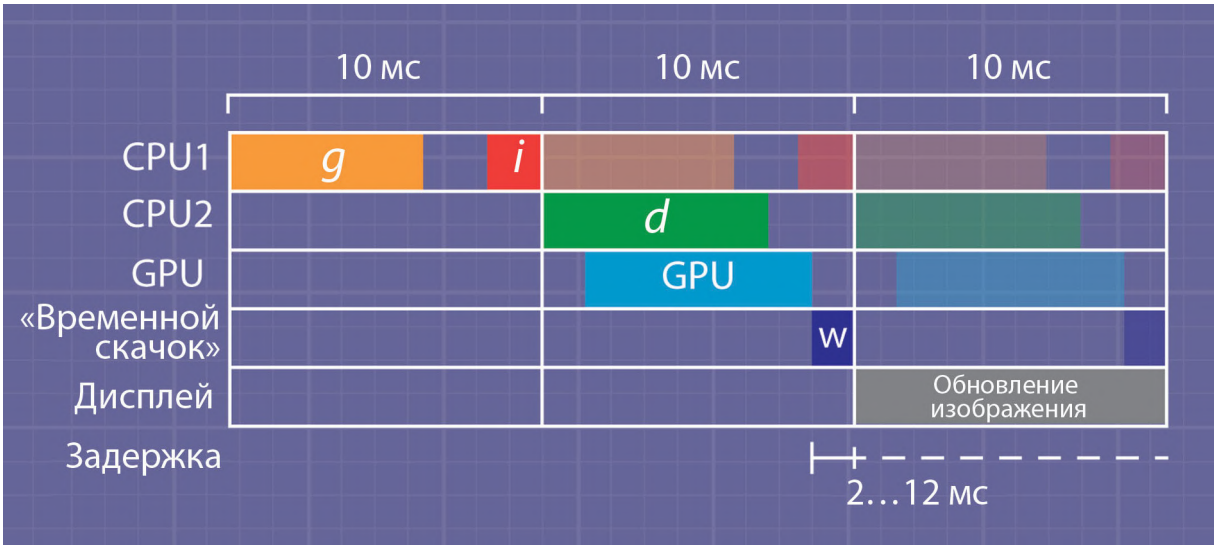

Рис. 10.7. Простая визуализация линии рендеринга с менее одного кадра ожидания из-за временного преобразования; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

Ранее я сказал, что *GPU* выводит отрендеренный кадр на дисплей, но это не совсем так. На самом деле то, что делает графический процессор, это отрисовка того, что называется задним буфером (*back buffer*), в то время как дисплей фактически читает из переднего (или кадрового) буфера (*front buffer*), непосредственно не затрагивает *GPU*. Это происходит из-за несоответствия того, как дисплей обновляется (помните, слева направо, сверху вниз) и как графический процессор отображает геометрию в сцене, а также механизма этого взаимодействия. Графический процессор не выполняет аккуратную отрисовку сверху и слева к вниз и направо, как это делает дисплей; вместо этого он отображает каждый вызов рисования, который он получает от процессора по очереди, один поверх другого, подобно художнику, накладывающему штрихи на полотно. Таким образом, вы сталкиваетесь с этой досадной проблемой при рендеринге непосредственно в буфер, из которого дисплей читает; то есть, если дисплей начинает показывать новый кадр, прежде чем вы закончите его рендеринг, он может фактически отображать на экране неполное изображение. Для решения этой проблемы существует механизм заднего буфера; графический процессор рендерит кадр в него, и когда дисплей готов к новому изображению, он меняет буфер, если кадр готов. Если кадр еще не готов, *GPU* просто отображает старое изображение снова, не показывая неполное изображение.

Почему это так важно? Оказалось, операция временного преобразования действительно создает изображение слева направо, сверху вниз, и на самом деле, когда временное преобразование действует, нет никакой реальной необходимости для этого двойной буферизации, потому что вы можете гарантировать, что вы начинаете преобразование с достаточным времени для буквально гонки с обновлением дисплея, пока он не закончил. Это то, что называется рендерингом переднего (кадрового) буфера или масштабированием; в настоящее время он реализован как на *Gear VR*, так и на *Daydream VR*, открывая широкие возможности. Поскольку вы искажаете изображение за миллисекунды до его отображения на экране, на самом деле существует вероятность того, что вы можете обновить входные данные до своей деформации по мере обновления дисплея, и на почти построчной (если вы достаточно хорошо его размечаете) основе вы можете деформировать последнюю информацию на дисплее. Именно таким образом *Gear VR* оказывает и выполняет два перекоса каждое обновление (по одному для каждого глаза), что показано на рис. 10.8.

Этот рендеринг переднего (кадрового) буфера включен по умолчанию на *Gear VR*, но он должен быть активирован вручную на *Daydreeam VR* (рис. 10.9). Если вы планируете использовать позиционный трекинг, выбирайте версию *Daydream* и добавляйте поддержку *Google Cardboard* в случае использования.

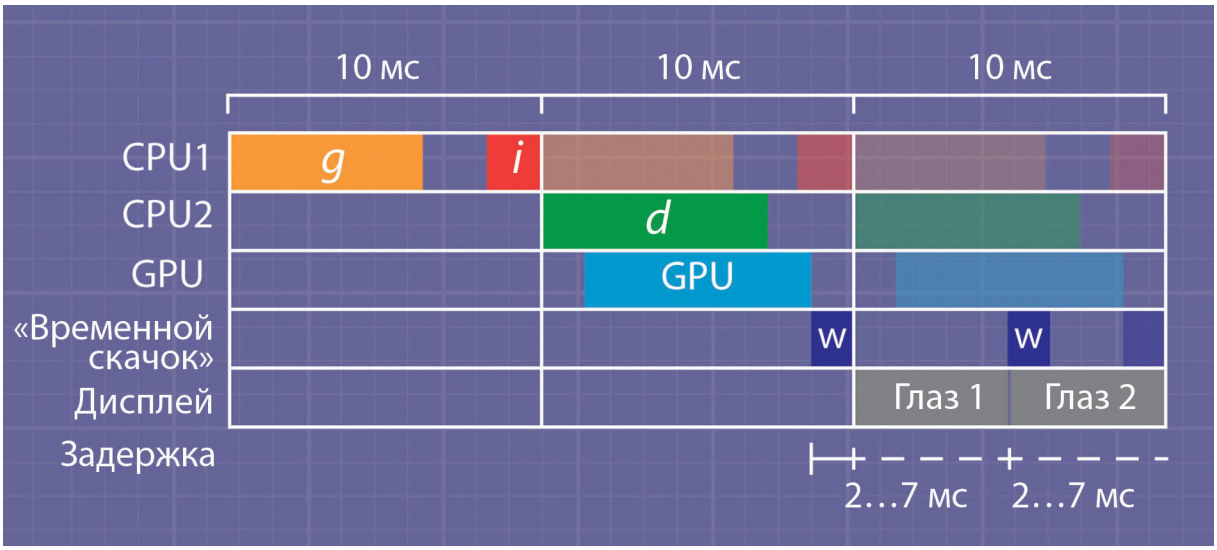

Рис. 10.8. Простая визуализация линии рендеринга с задержкой менее одного кадра из-за временного преобразования и рендеринга из переднего буфера; *i* = ввод, *g* = игра, *d* = отрисовка. Каждая колонка — это 10 миллисекунд, за которые происходит обновление изображения на дисплее с частотой 100 Гц

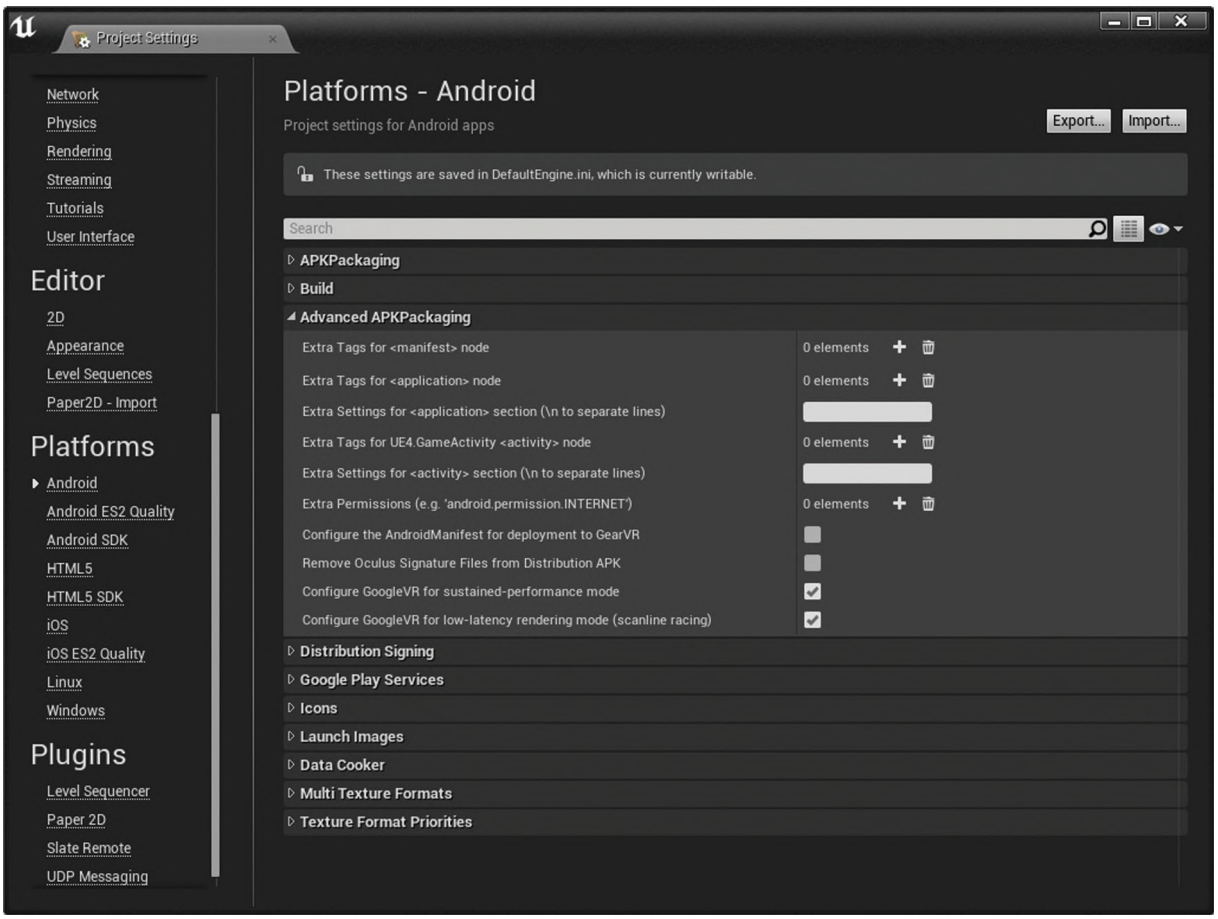

Рис. 10.9. Параметры конфигурации для включения масштабирования на *Daydream VR*

10.3. Повышение производительности

Другой столп методов рендеринга направлен на то, чтобы помочь вашему опыту *VR* визуализироваться («рендериться») за меньшее время или использовать меньше ресурсов для получения того же результата.

Эти оптимизации важны не только потому, что они позволяют вам отводить больше времени на рендеринг других аспектов вашей игры (например, теней или необычных материалов), которые могут сделать ваши миры более реалистичными, но и позволяют запускать *VR* на менее производительном оборудовании. Это может быть очень важно для мобильного *VR*, а также для снижения порога входа в *VR*.

Мы рассмотрели, сколько кадру требуется времени и как управлять механизмом отрисовки на *CPU* и *GPU*, чтобы уменьшить время ожидания. Давайте рассмотрим, как на самом деле визуализируется кадр, чтобы увидеть, какую производительность вы можете получить, просто изменив способ его рендеринга.

10.3.1. Методы визуализации

Традиционный игровой мир в чистом виде состоит из тысяч и тысяч отдельных одномерных точек в пространстве (известных как вершины). Однако эти точки сами по себе не способствуют созданию реалистичного 3*D*-мира. Чтобы передать игроку ощущение просмотра полностью трехмерного мира, эти точки сначала группируются в двухмерные треугольники. Сочетание треугольников в 3*D*-среде дает пользователю представление о форме объектов в мире. Хотя эти формы могут передавать ограниченный вид объекта (через перспективные проекции, которые обычно применяются), затенение используется, чтобы получить истинное представление о том, как объекты существуют в 3*D*-мире. Затенение призвано показать игроку, как свет взаимодействует с объектами в игровом мире. Прежде чем мы перейдем к тому, как освещение выполняется в играх, нам нужно посмотреть на процесс, который предшествует стадии освещения.

Рендеринг в играх работает по принципу программируемого конвейера. Это означает, что (в отличие от фиксированных конвейеров прошлых лет) графические *API* позволяют разработчикам игр контролировать каким образом вершины и фрагменты (данные, которые могут быть использованы для создания пикселей) обрабатываются для создания окончательного изображения на экране игрока.

Современные графические *API* позволяют изменять множество различных шейдеров до того, как конечные значения пикселей игрового мира будут показаны пользователю. Однако для простоты мы остановимся на двух наиболее важных шейдерах: вершинном шейдере и шейдере фрагментов / пикселей.

В простом приложении вершинный шейдер берет в качестве входных данных вершины отдельного фрагмента геометрии в игровом мире и выполняет все необходимые действия, чтобы преобразовать эти точки в данные для шейдера фрагментов (например, переход из пространства объектов в пространство экрана и применение перспективной проекции виртуальной камеры).

После этого этапа возможны два варианта. Этими двумя вариантами, которые классифицируют основные методы визуализации, применимые к приложениям *UE*4, являются прямой рендеринг (*forward rendering*) и отложенное затенение (*deferred shading*).

10.3.1.1. Прямой рендеринг

В традиционной настройке прямой визуализации следующий этап довольно прост. Шейдер фрагментов принимает на вход выход из вершинного шейдера (в практике существует посредник шейдер под названием «геометрия шейдеров», но это для простоты). Используя данные вершин, вы можете зациклить все потенциальные пиксели, которые в итоге будут отображаться игроку. Эти потенциальные пиксели называются фрагментами, и задача этого шейдера состоит в том, чтобы взять эти фрагменты и затенить их или, скорее, вычислить окончательный цвет пикселей, которые будут показаны на экране (рис. 10.10).

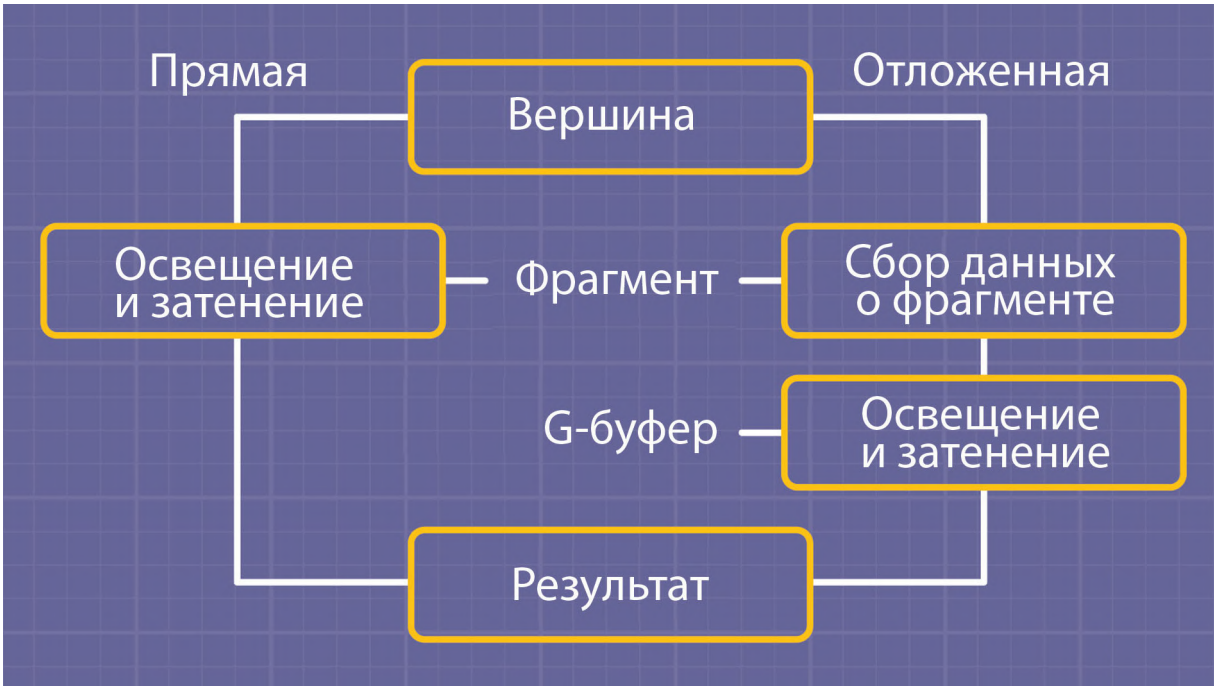

Рис. 10.10. Прямой рендер и отложенное затенение. Прямой подход вычисляет весь свет в шейдере фрагментов, в то время как отложенный подход помещает описания каждого фрагмента в буферы, которые могут быть использованы для освещения сцены в другое время.

Для затенения этих фрагментов можно воспользоваться разными приемами. В качестве основы нам понадобится следующее. Во-первых, можно использовать данные вершины для вычисления внешнего направления треугольника, к которому принадлежит фрагмент (перпендикуляр). Это позволяет затем зациклить все источники света в сцене и сравнить внешний угол треугольника с направлением источника света. Таким образом, вы можете рассчитать, насколько сильно свет будет влиять на конкретную поверхность. Затем вы можете еще раз использовать данные вершин, чтобы найти положение фрагмента в мировом пространстве и использовать его для вычисления освещенности в зависимости от удаленности от источника света.

> **Заметка**
>
> Процесс проецирования вершин, которые существуют в вашем мире, на экран для затенения называется *растеризацией*.

Заметим, что, по сравнению с прямым рендерингом, в этом подходе есть несколько неоптимальных черт. Во-первых, вам придется перебрать все источники света в вашей сцене для каждого фрагмента затенения, даже если этот фрагмент почти не затронут каким-либо источником света. Этого можно частично избежать, проверив радиус влияния (затухания) каждого источника света, чтобы увидеть, достигает ли он фрагмента, и если нет, то пропустить этот источник света. Однако, как мы обсудим позже, возможно, есть более эффективные способы оптимизации. Еще одна проблема такого простого прямого рендеринга заключается в том, что из-за способа визуализации треугольников на сцене вы можете полностью осветить и растеризировать кусок геометрии только для того, чтобы он был полностью закрыт другим куском геометрии позже в отрисовке, следовательно, тратя время *GPU* на то, что пользователь не увидит.

Это явление известно как перерисовка (*overdraw*), и оно ограничивает производительность базового прямого рендеринга количеством геометрии в вашем мире, умноженным на количество динамических источников света.

> **Заметка**
>
> Перерисовку (*overdraw*) можно в значительной степени устранить с помощью *Early Z-Pass*, визуализирующего буфер глубины, который можно протестировать против отбраковки пикселей, которые будут скрыты в конечном изображении (до того, как на них будет наложено освещение).

Тем не менее это связано с необходимостью растеризации сцены дважды, что, в зависимости от ситуации, может привести к снижению производительности.

Развитие алгоритмов трехмерной визуализации довольно долго упиралось в это ограничение на количество динамических огней. В середине 2000-х годов стал популярен отложенный подход к затенению, потому что он убрал зависимость между количеством источников света и геометрической сложностью игрового мира.

10.3.1.2. Отложенное затенение

В отличие от прямого рендеринга, отложенное затенение берет фрагмент шейдера из прямого примера (*forward example*) и вместо вывода окончательного пикселя, который будет отображаться, выводит описание фрагмента в рендеринговые цели или буферы, которые затем будут объединены, чтобы затенить, осветить и вывести окончательную информацию о пикселях (следовательно, откладывая шаг освещения/затенения) (рис. 10.10). Эти буферы известны как геометрические буферы (*G*-буферы); пример *G*-буфера в *UE*4, можно увидеть на рис. 10.11.

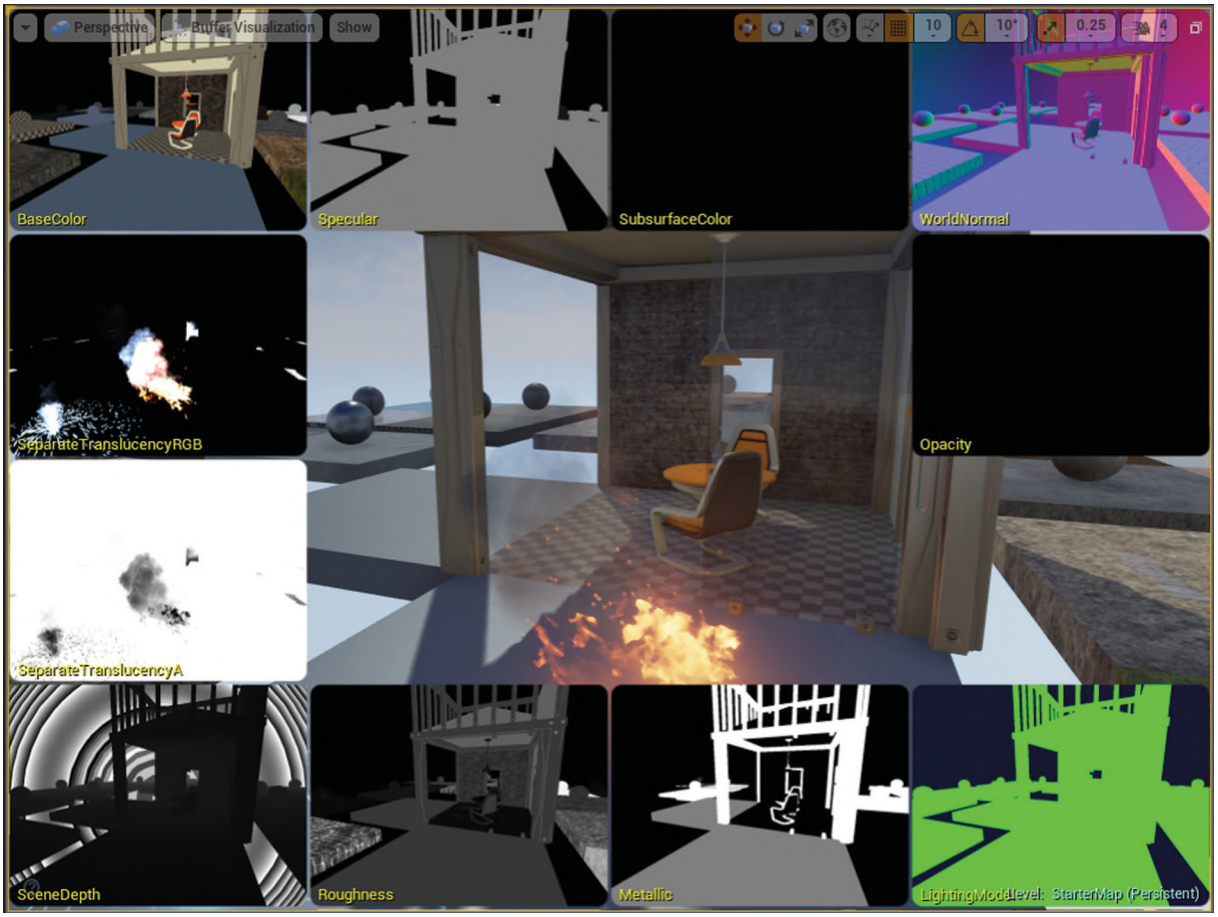

Рис. 10.11. Иллюстрация работы *G*-буфера на примере уровня *UE*4

Однако отложенный рендеринг не лишен проблем, хотя он может устранить перерисовку (*overdraw*) для обычно дорогостоящих операций освещения (путем создания карты глубины сцены до того, как будет выполнено какое-либо освещение, что позволит выполнять вычисления освещения только на ближайших фрагментах). Буфер глубины, задействованный в работе механизма, также делает трудным визуализацию прозрачных объектов. Поскольку буфер глубины может представлять только одно значение глубины, объекты, которым может потребоваться несколько глубин для вычисления освещения (например, полупрозрачные объекты), не могут быть легко представлены в отложенном рендеринге.

Чтобы обойти это, многим реализациям отложенного рендеринга приходится вернуться к прямому рендерингу прозрачных объектов. Таким образом, вы должны растрировать «геометрию» дважды: один раз для непрозрачных объектов и второй раз — для прозрачных. (Это выходит за рамки этой книги, но *UE*4 фактически создает сферическую функцию для каждого прозрачного объекта, что позволяет ему приблизить освещение сцены и составить из этих просвечивающих объектов сцену с последующей обработкой).

10.3.1.3. Какой подход выбрать: прямой или отложенный

Для настольных приложений *UE*4 изначально реализует отложенный рендеринг, потому что *G*-буфер позволяет вам делать много причудливых трюков рендеринга, которые сделают ваши игры очень реалистичными (например, отражения пространства экрана [*screen space reflection — SSR*], рассеяние подповерхностного пространства экрана [*screen space subsurface scattering — SSSS*], окклюзия окружающего пространства экрана [*screen space ambient occlusion — SSAO*] и многие другие). Однако эти эффекты могут быть дорогостоящими с точки зрения вычислений, и поскольку требования к рендерингу *VR* настолько высоки, в большинстве случаев эти эффекты не могут быть использованы. Кроме того, не наилучшим образом подходят для *VR*, потому что рассчитываются в пространстве экрана, что из-за стереоскопической природы *VR* может создать стереонесоответствия и вызвать дискомфорт для ваших пользователей.

Прямой рендеринг также позволяет упростить реализацию аппаратного сглаживания *MSAA* (*multi-sample anti-aliasing*) по сравнению с постпроцессным сглаживанием *TAA* (*temporal anti-aliasing*), которое *UE*4 реализует по умолчанию. Технология *MSAA* может лучше подходить для *VR*, поскольку может привести к более четкому изображению — эффект *MSAA* влияет только на края геометрии, таким образом, для той же четкости изображения необходимо обсчитывать меньше пикселей на экране, что в целом снижает стоимость визуализации.

В некоторых случаях, однако, ограничения стандартного прямого рендеринга могут быть болезненным моментом для разработчиков (в основном неэффективность работы со многими динамическими источниками света). Для противодействия этому существует много методов как прямого, так и отложенного типа, которые управляют миллионами динамических источников света. Двумя ведущими методами являются плиточный рендеринг и кластерный рендеринг. Также стоит отметить, что оба метода могут быть реализованы с использованием прямого рендеринга или отложенного затенения. (На самом деле, если в сцене достаточно света, *UE*4 перейдет к плитко-отложенной отрисовке вместо традиционной отложенной).

Основная идея плиточного подхода к прямой отрисовке заключается в ограничении количества источников света, по которым каждый фрагмент шейдера должен циклически вычислять свое освещение. Это делается путем разделения вида на квадраты (плитки) и нахождения того, какие огни влияют на каждую конкретную плитку. Во время рендеринга фрагмент должен зацикливаться только на подмножестве всех источников света в сцене. Тем не менее этот подход имеет потенциальный недостаток. Если одна плитка содержит два объекта, на которые влияют разные огни (что обычно происходит с объектами, находящимися далеко друг от друга), нет никакого способа разделить их, потому что объекты сгруппированы в двух измерениях. Чтобы решить эту проблему используют кластерный рендеринг, который не только разбивает экранное пространство на плитки, но и делит их на кластеры в направлении усечения камеры (рис. 10.12).

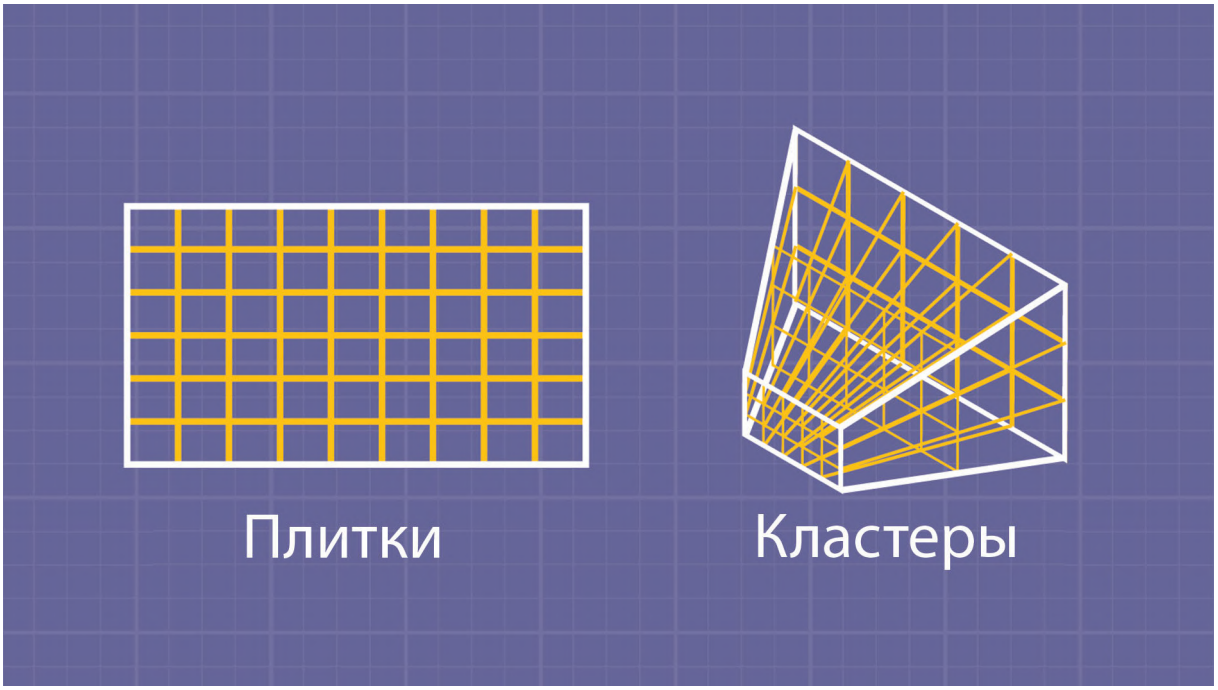

Рис. 10.12. Плиточная и кластерная отрисовка

На момент написания этой книги, *UE*4 реализует отложенный рендеринг на ПК проектах, в то время как прямой рендеринг должен использоваться на мобильных устройствах (из-за болевых точек обработки необходимых целей рендеринга для отложенного рендеринга на мобильных устройствах). Тем не менее из-за строгих требований к рендерингу *VR* сегодня и необходимости высококачественного, но дешевого сглаживания, прямой рендеринг вполне может быть лучшим вариантом для текущего состояния *VR*-игр. Компания-разработчик *Epic* внедрила прямой рендер для ПК приложений в версиях *UE*4 4.14–4.15, что потенциально позволит разработчикам *VR* предоставлять пользователям более качественный опыт. Обратите внимание, что в *UE*4 4.12 многие функции отрисовки *UE*4 (например, *SSR* и *SSAO*) недоступны при использовании прямой отрисовки, так как *G*-буфер, необходимый для использования таких функций, не создается. При этом, однако, могут быть и другие преимущества от расчета освещения на геометрическом уровне (например, плоские отражения на уровне каждого объекта), которые перевешивают этот недостаток.

10.3.2. *Instanced stereo*

Как показано на рис. 10.2–10.8, поток отрисовки, представленный зеленым цветом — это место, где движок отправляет вызовы рендеринга графическому процессору и сообщает ему, что отрисовывать. В отличие от обычного монитора, в *VR* вы должны подготовить изображения для обоих глаз; в самом простом случае вы можете просто визуализировать оба глаза полностью независимо друг от друга. Однако если вы задумаетесь об этом, вы можете рисовать оба глаза одновременно и сократить работу процессора. Фактически, в *UE*4 4.11 *Epic* представила функцию *Instanced Stereo Rendering*, которая делает именно это. Она не включена по умолчанию в версии 4.11, так как она не работает со всеми функциями отрисовки, но ее можно включить в настройках проекта.

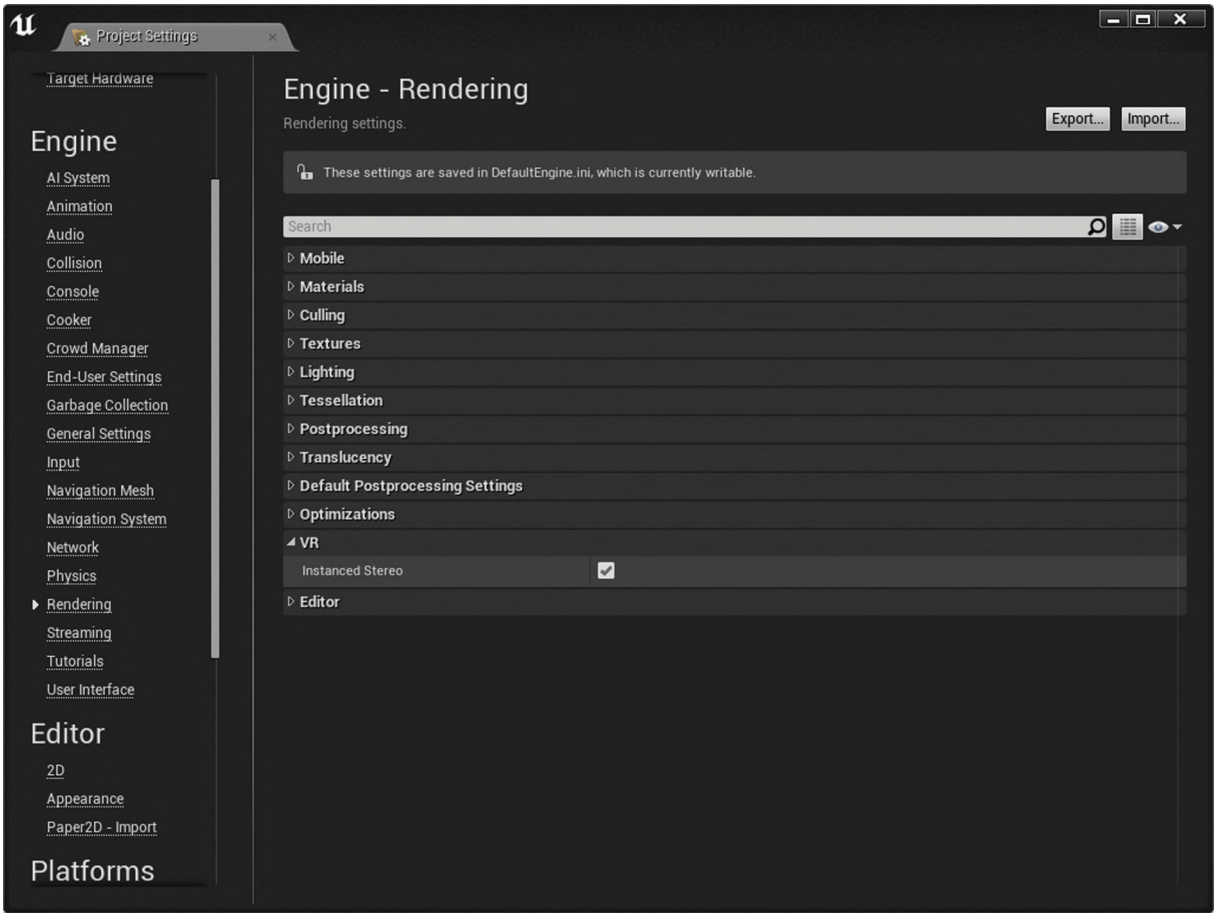

Рис. 10.13. Настройка *Instanced Stereo Rendering*

10.3.3. Оптимизация сетки скрытой области

Поскольку оптика в *VR*-шлеме имеет круглое поле зрения, то рендеринг на весь экран дисплея излишен, так как пользователь никогда не видит его целиком. Таким образом, *UE4* (начиная с версии 4.10) реализует скрытую область сетки, автоматически отбраковывая детали, которые никогда не будут видны. Этот метод также гарантирует, что эффекты постобработки обрабатываются только на тех деталях, которые увидит пользователь. Эта функция включена по умолчанию.

10.4. Настройка проекта *VR*

При создании нового проекта *UE4* по умолчанию включает все свои функции, а это означает, что производительность может пострадать от количества включенных механизмов.

Чтобы помешать этому, есть много вариантов, которые разработчики могут сами настраивать, а также несколько шаблонов (пресетов), способные помочь вам удовлетворить требования визуализации *VR*.

Во-первых, в диалоговом окне Создания проекта (рис. 10.14) и в Настройках проекта (рис. 10.15) представлены параметры целевого оборудования и производительности графики.

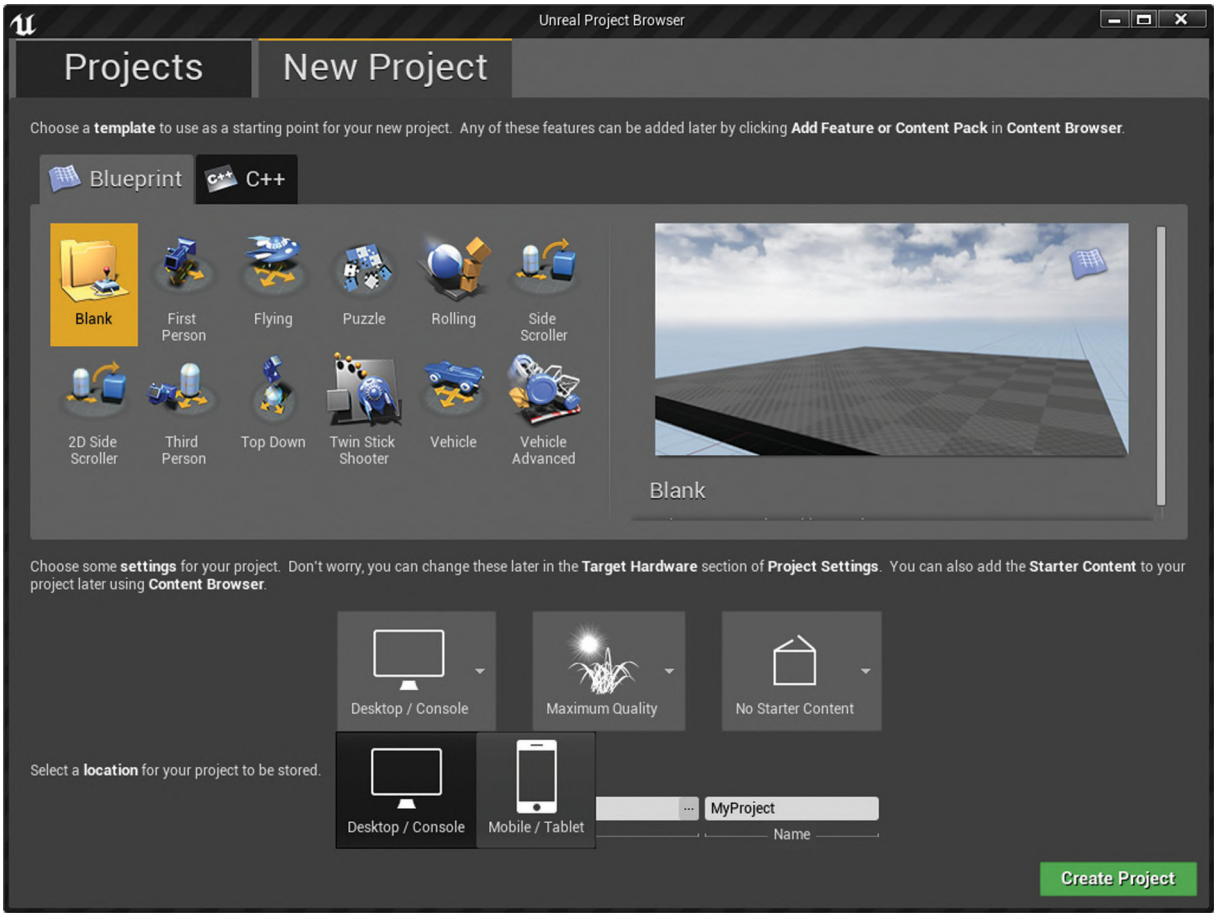

Рис. 10.14. Параметры целевого оборудования и производительности графики в диалоговом окне Создания проектов

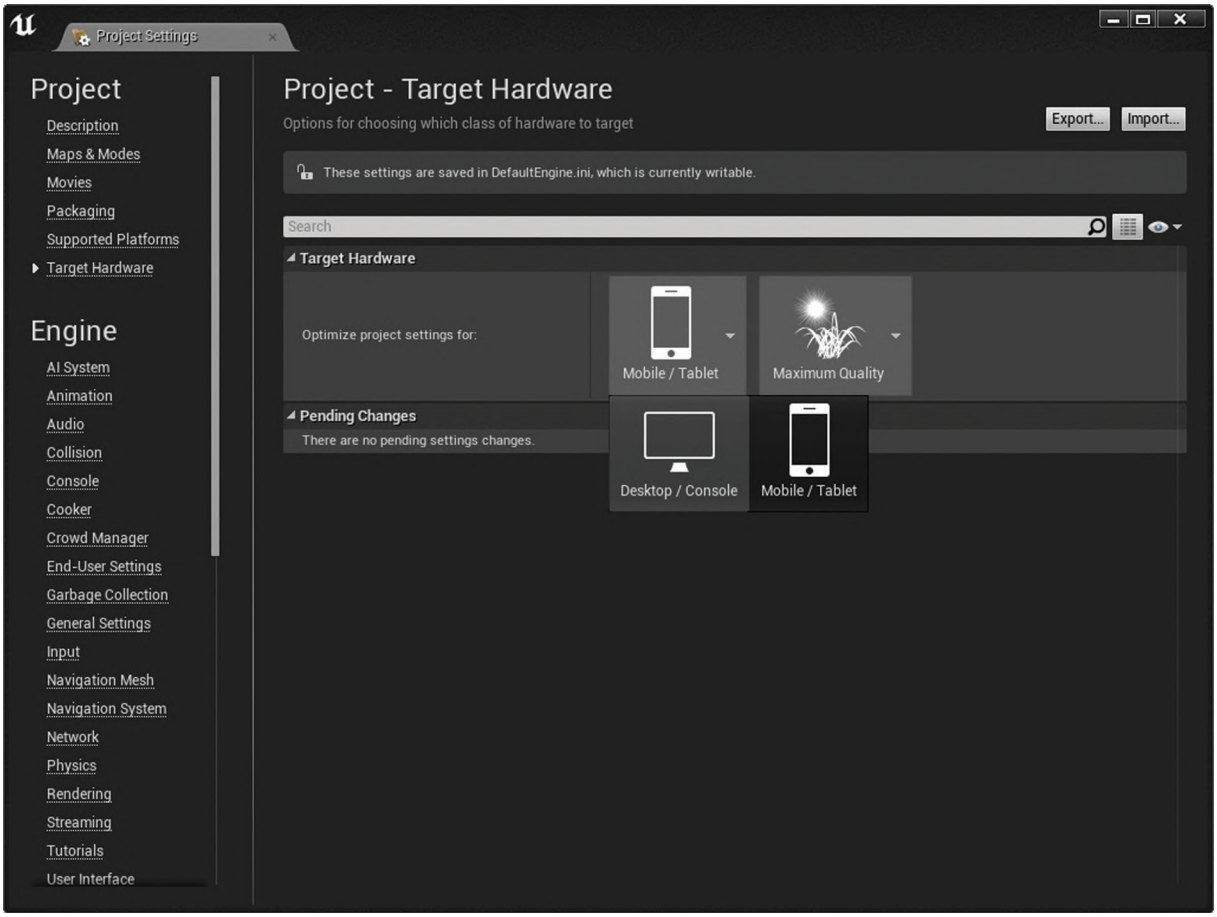

Рис. 10.15. Параметры целевого оборудования и производительности графики в настройках проекта

Чтобы увидеть различия между настройками (рис. 10.15), на котором показаны параметры целевого оборудования с соответствующим качеством графики. Как вы можете видеть, настройка масштабируемого качества графики удаляет *Auto Exposure* и *Motion Blur* из эффектов постобработки по умолчанию. Она также отключает *Anti-Aliasing*, когда целевое оборудование настроено на ПК-версию. Однако установка масштабируемого качества графики, когда целевое оборудование является мобильным, просто отключает *Bloom* и *Mobile HDR*; автоматическая экспозиция и размытие движения будет отключены, если целевое устройство установлено как мобильное.

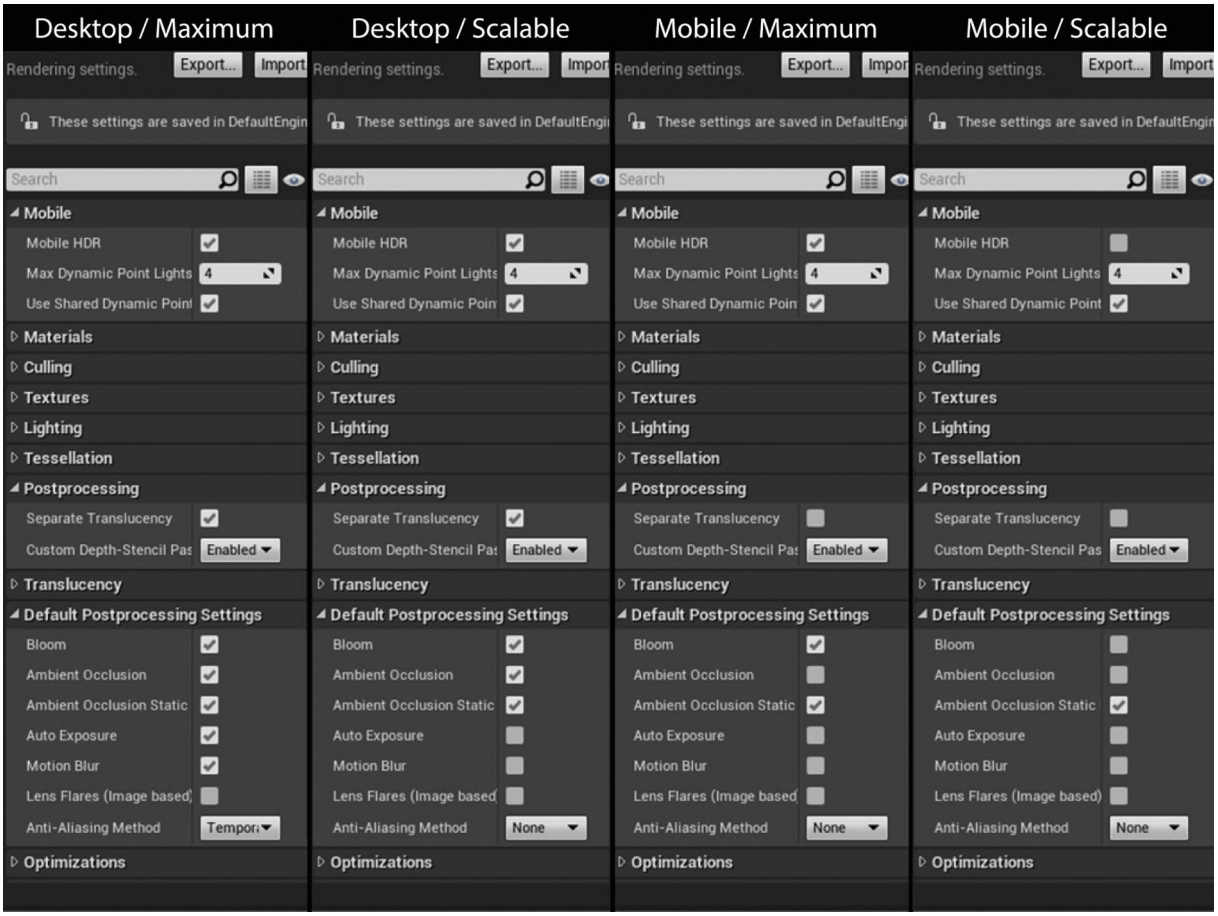

Рис. 10.16. Параметры визуализации для целевого оборудования и графики

Так как эффекты *Motion Blur*, *Auto Exposure*, *Bloom* и *Ambient Occlusion* не нужны в *VR*, вы можете просто отключить их все. Выбор мобильного устройства при создании проекта способен ускорить этот процесс.

Обратите внимание, что переход к мобильному целевому оборудованию отключает *Separate Translucency*, что приведет к повышению производительности, но может привести к некоторым артефактам в перекрывающихся прозрачных поверхностях (рис. 10.17). Рекомендую отключить его, чтобы получить производительность, и включить его, только если вы столкнетесь с проблемами.

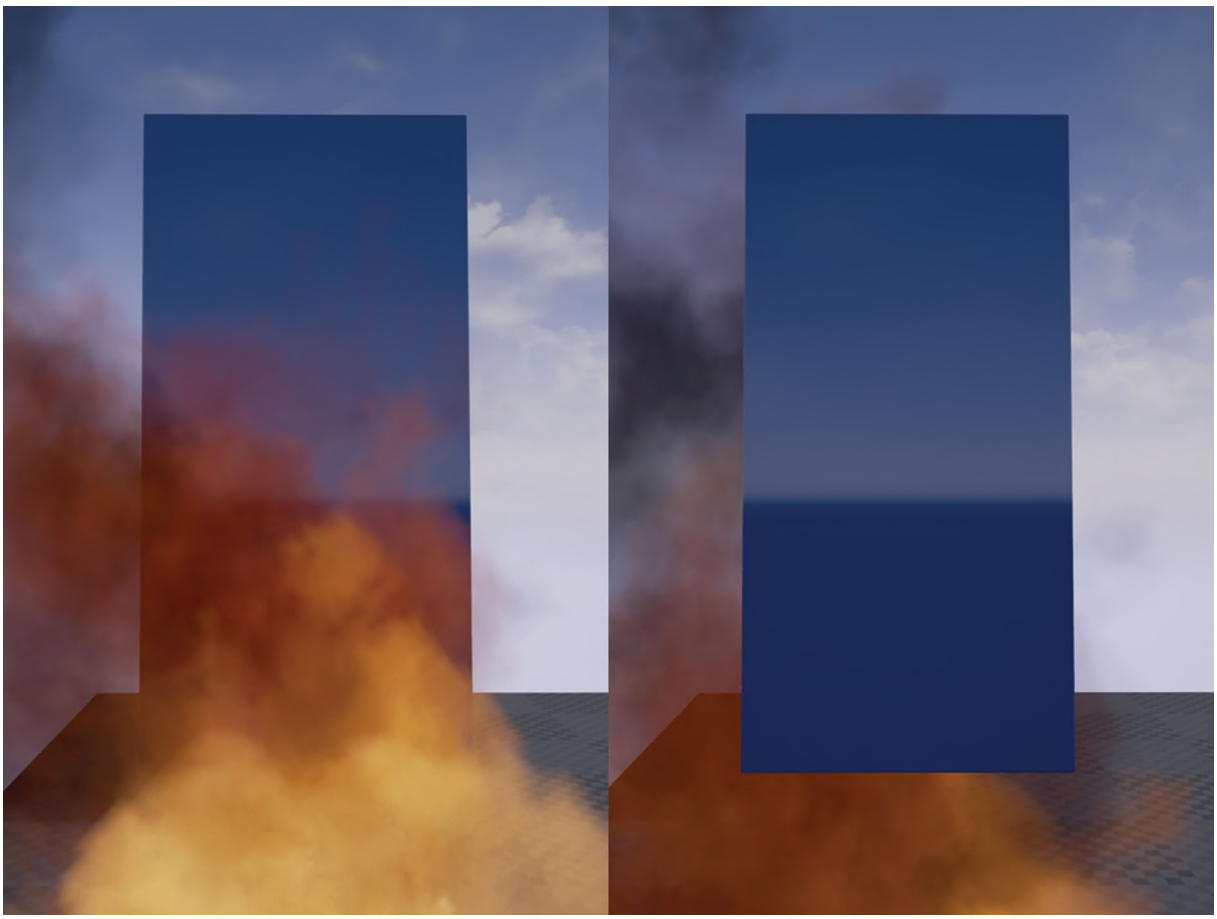

Рис. 10.17. Два отдельных полупрозрачных объекта, перекрывающих друг друга: слева *Separate Translucency* включено, а справа нет

При выборе параметров, которые следует отключить в соответствии с вашими требованиями к производительности, вы могли столкнуться с параметрами масштабируемости ядра (меню *Settings* ⇒ *Engine Scalability Settings*) (см. рис. 10.18). Они на самом деле являются только настройками ядра и не влияют на любой проект, который вы будете создавать за пределами движка.

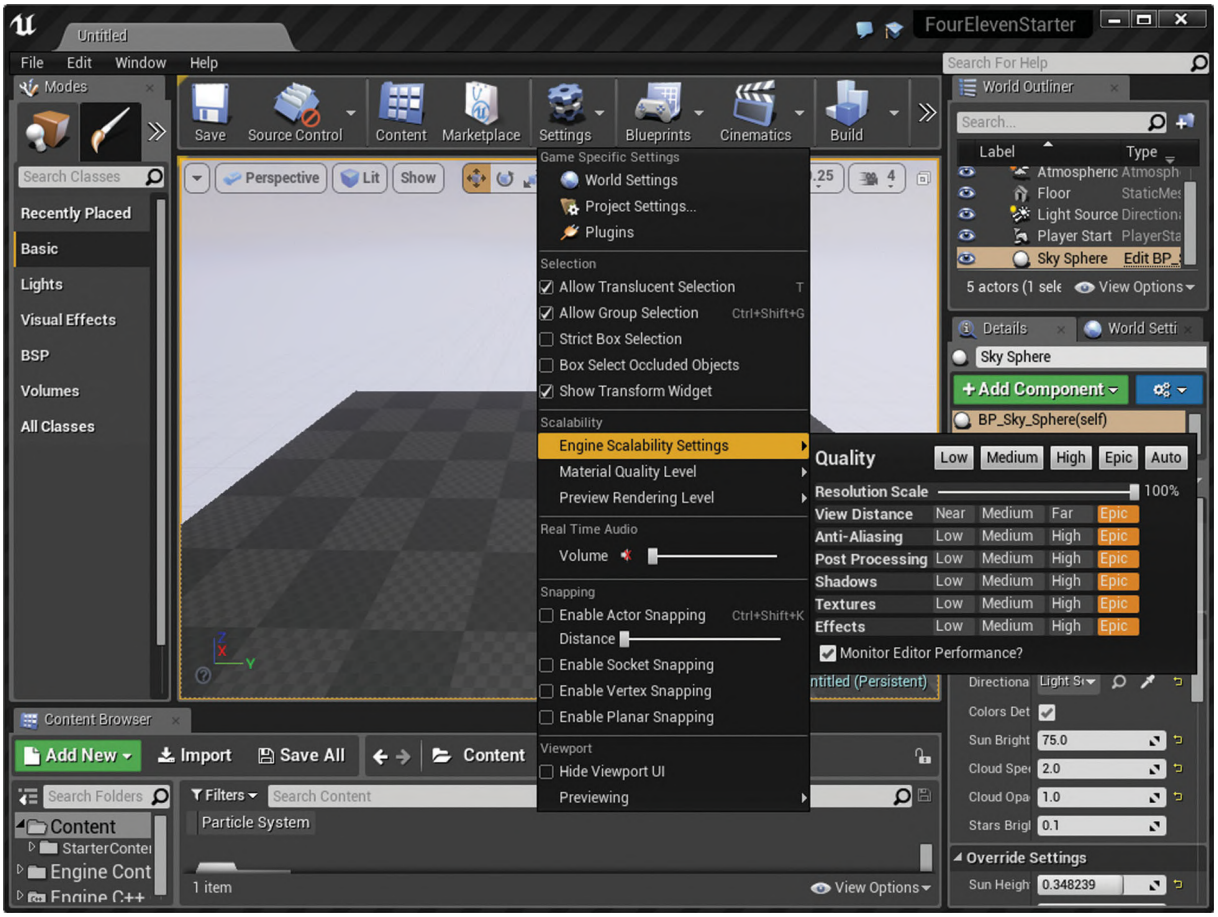

Рис. 10.18. Параметры масштабируемости ядра

Чтобы изменить настройки масштабируемости и повлиять на финальную игру пакета, поместите все настройки в файле *DefaultEngine.ini* (находится *<ProjectName>/Confid/DefaultEngine.ini*) в виде консольных команд в разделе [*SystemSettings*].

В качестве примера консольная команда для изменения относительного разрешения рендеринга `r.screenpercentage=x`, где `x` — значение от `30` до `300%` разрешения экрана.

В *DefaultEngine.ini*, для установки относительного разрешения в 130%, это будет выглядеть так:

```
[SystemSettings]

r.screenpercantage=130
```

При этом вы можете изменить значение по умолчанию почти для всех функций движка (например, сглаживание, `r.PostProcessAAQuality=x`) или даже отключить такие функции, как *HZB occlusion* (`r.HZBOcclusion=0`), которая имеет высокую первоначальную стоимость, но масштабируется лучше, чем *occlusion* по умолчанию.

Теперь мы знаем, как отключать функции ядра. Но как нам выяснить, что отключить?

Прежде чем ответить на этот вопрос, нужно выяснить, что вас больше ограничивает — графический или центральный процессор, то есть *GPU* ли виноват в слишком медленной смене кадров?

Для прояснения ситуации используйте консольную команду `Stat Unit`*, которая покажет вам общее ремя подготовки кадра [*Frame*], время, затраченное потоками центрального процессора (обсчета игровых событий [*Game*] и вызовов отрисовки [*Draw*]) и время, затраченное графическим процессором [*GPU*] (рис. 10.19). В принципе наибольшее из трех значений (игра, отрисовка, и *GPU*) определяет, ограничены вы *CPU* (либо логикой игрового процесса, либо вызовами отрисовки) или *GPU*.

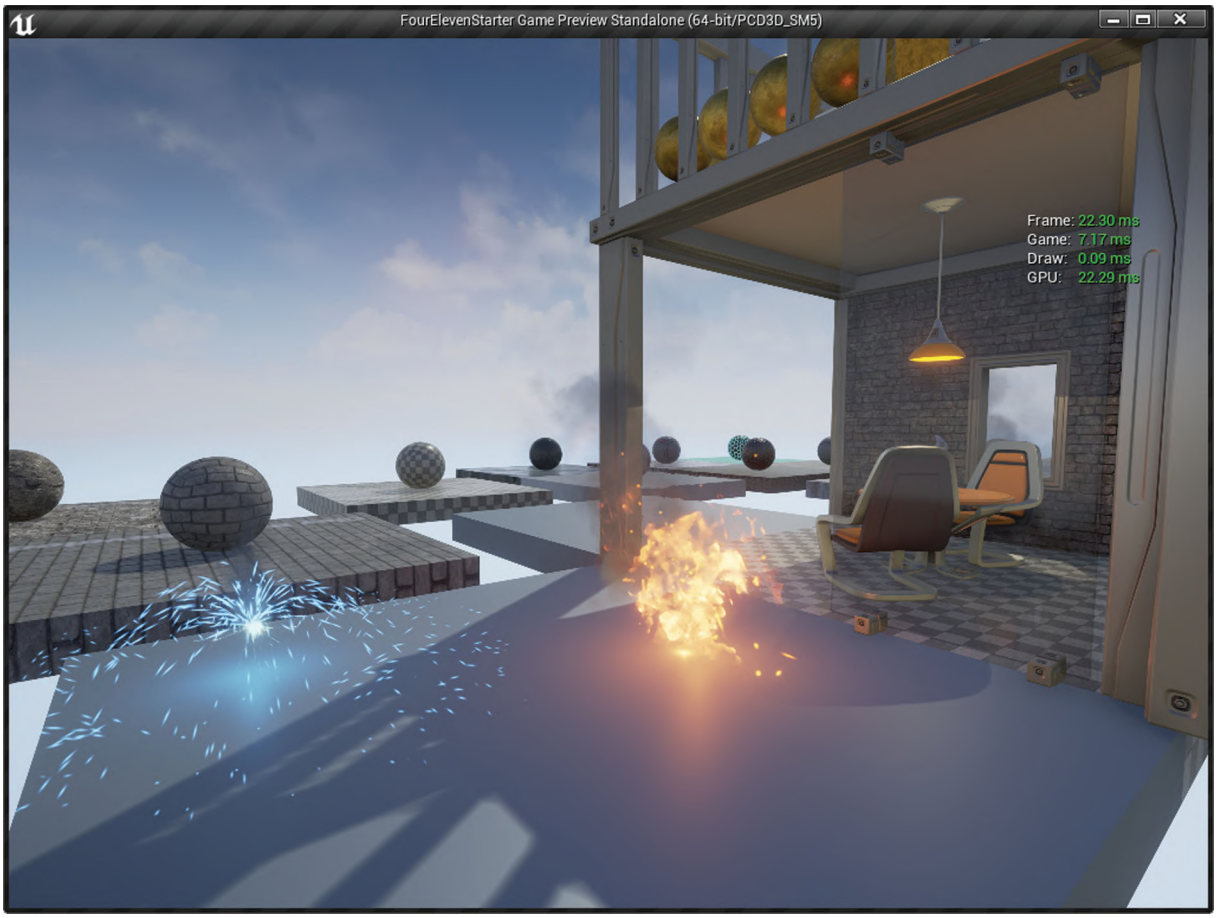

Рис. 10.19. Синхронизация *Stat Unit*

Узнав, что вас больше ограничивает, вы можете профилировать *GPU* или *CPU* с помощью инструментов профилирования *UE*4.

Если вы ограничены *GPU* (как показано на рис. 10.19), вы можете использовать визуализатор *GPU* (рис. 10.20) с помощью консольной команды `ProfileGPU` или сочетания клавиш *Ctrl*+*Shift*+,. После того как визуализатор открыт, вы можете найти время рендеринга для всего кадра и точно узнать, что занимает больше времени, чем должно.

---

* Чтобы ввести консольную команду в режиме *Preview*, используйте клавишу тильда «~» — *Прим. пер.*

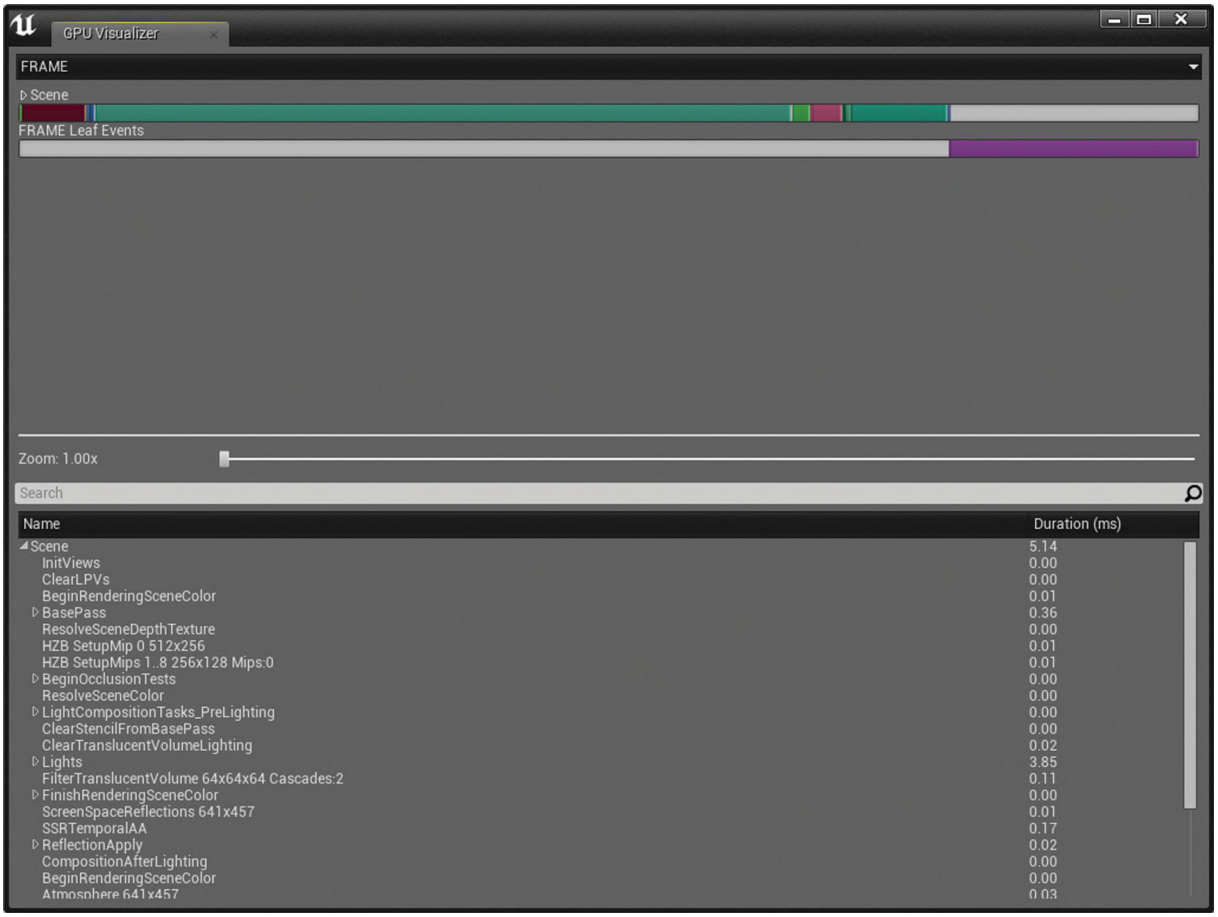

Рис. 10.20. Визуализатор работы *GPU*

Если вы ограничены потоком отрисовки *CPU*, скорее всего, вы достигаете предела вызовов отрисовки и, возможно, придется ограничить количество объектов в сцене или объединить некоторые объекты в одну сетку. Чтобы проверить вызовы отрисовки, используйте консольную команду `Stat SceneRendering`.

Если вы ограничены потоком обсчета игровых событий *CPU*, посмотрите, что может вызвать проблему с помощью консольной команды `Stat Game`.

10.5. Заключение

В этой главе мы рассмотрели причины особых требований к рендерингу *VR*-сцены, связанных в основном с особенностями ресурсоемкого оборудования текущего поколения *VR*-шлемов. Вы узнали о хитроумных способах оптимизации, которые применяются в *UE*4 и других движках, позволяющих гарантировать пользователю комфортную игру с максимальным качеством изображения и минимальными задержками.

Вы познакомились с базовыми настройками проекта, которые *UE*4 позволяет как задать при создании проекта, так и изменить в дальнейшем, причем с дополнительными ключами. Наконец, вы разобрались, какие параметры нужно настроить в первую очередь в случае проблем с производительностью.

10.6. Упражнения

Чтобы отточить навыки, основанные на материале этой главы, оптимизируйте собственные игры или приложения, либо поиграйте с настройками заготовок игр, идущих в комплекте с *UE*4.

ЧАСТЬ III

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

VR-РЕДАКТОР

С помощью *VR*-редактора вы сможете увидеть, как ваши игры выглядят в *VR*. Используя контроллеры, вы можете полноценно взаимодействовать с объектами в виртуальном мире, перемещать их, масштабировать и вращать, создавая правдоподобный виртуальный мир. В этом приложении вы узнаете, как использовать *VR*-редактор.

A.1. Включение *VR*-редактора

Редактор появился в экспериментальной версии 4.12.

Чтобы редактор стал доступен, откройте настройки редактора (меню *Edit* ⇒ *Editor Preferences*). На вкладке *Experimental* включите параметр *Enable VR Editor* (рис. A.1). Параметр ниже называется *Enable VR Mode Auto-Entry*; это автоматический вход в *VR*-редактор при активации датчика приближения в шлемах, которые его поддерживают.

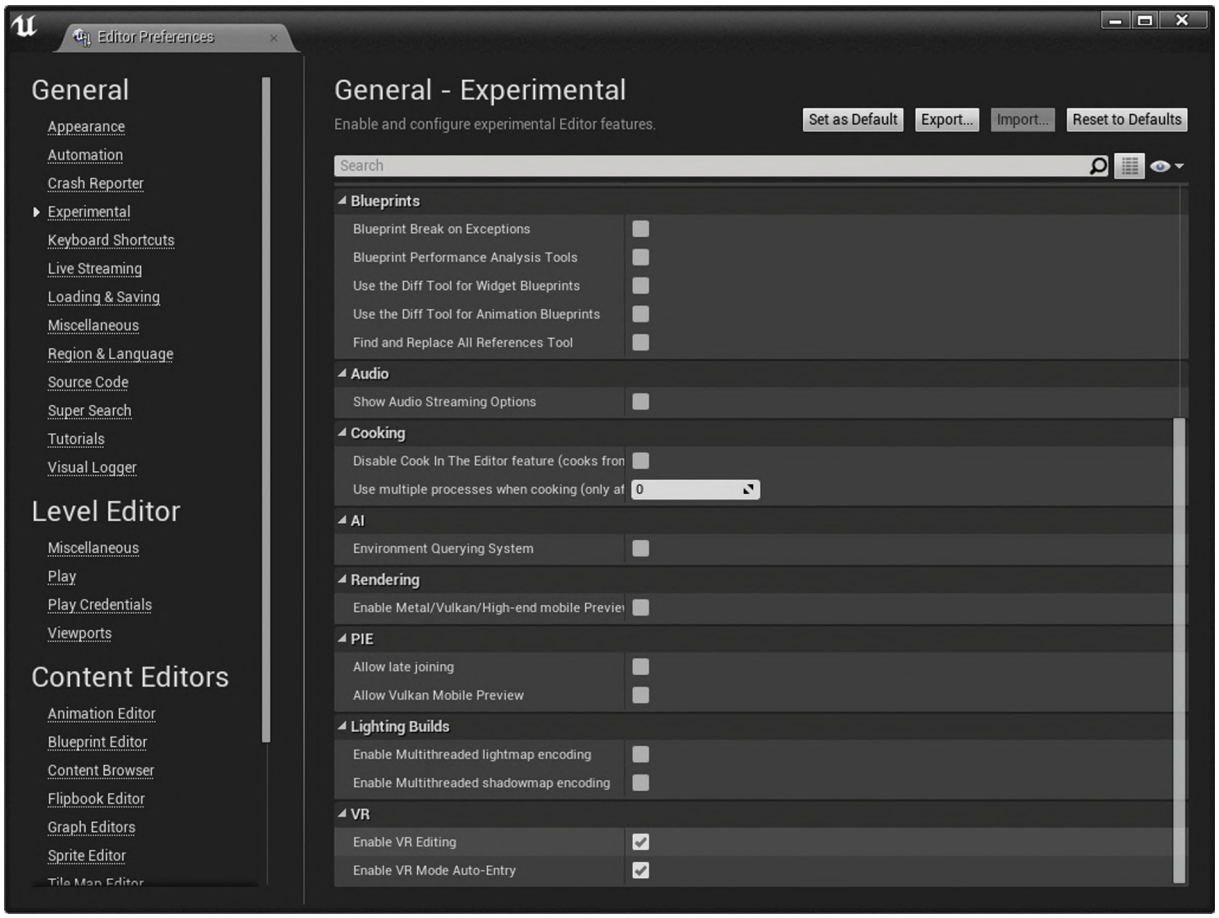

Рис. A.1. Включение *VR*-редактора в настройках редактора

Чтобы запустить *VR*-редактор, закройте настройки редактора и нажмите на появившуюся кнопку *VR* на панели инструментов (рис. A.2).

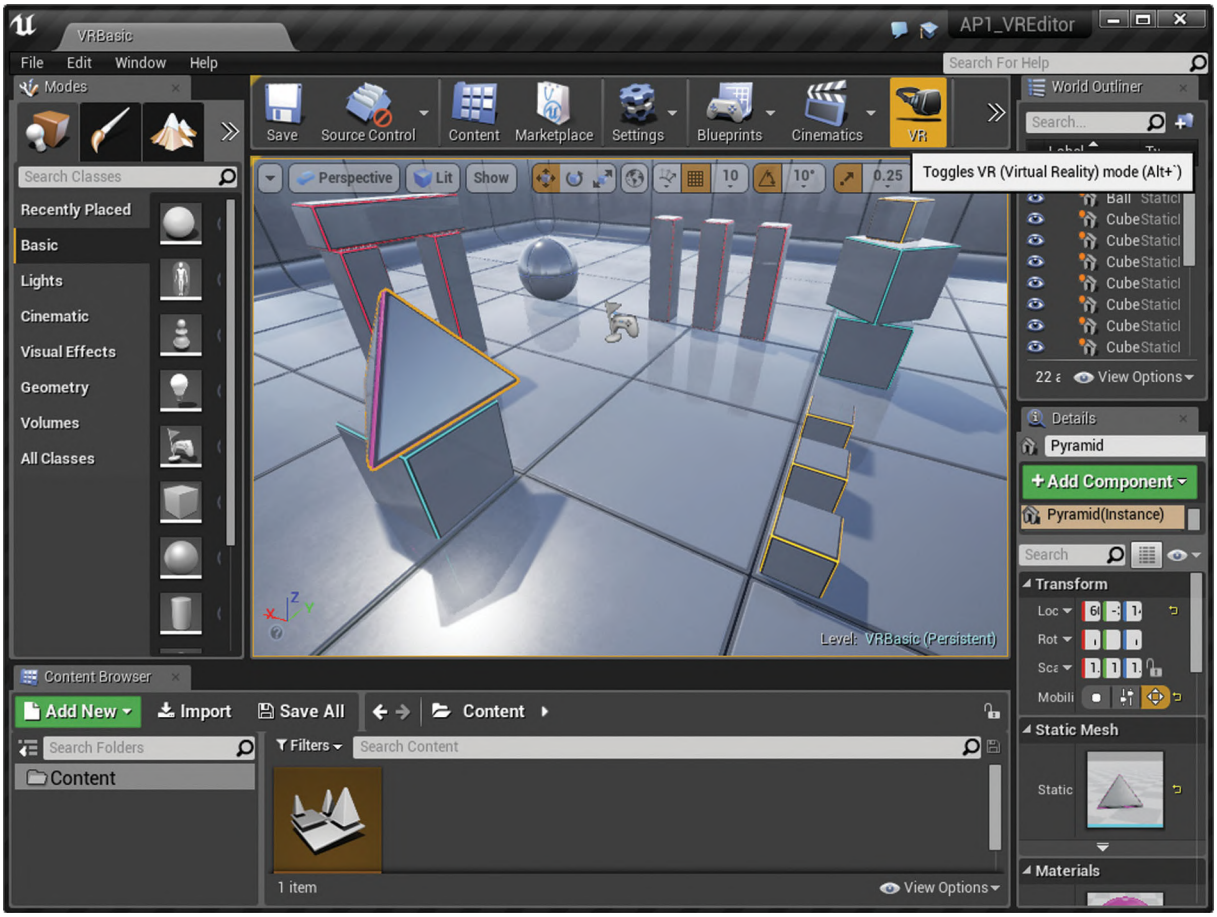

Рис. А.2. Включение *VR*-редактора новой кнопкой

А.2. Управление *VR*-редактором

VR-редактор позволяет выполнить большинство действий, которые доступны в обычном 2*D*-редакторе. Но в *VR* есть несколько оригинальных элементов управления, которые стоит рассмотреть.

VR-редактор полностью использует контроллеры и не реализует весь свой потенциал в их отсутствии. В данный момент редактор поддерживает контроллеры *Oculus Touch* и *HTC Vive*, задействуя триггер [*trigger*] (для выбора объектов), грип и боковые кнопки [*grip/side buttons*] (для навигации в виртуальном мире), тачпад или джойстик [*touchpad/joystick*] (для вызова радиального меню и масштабирования выбранных объектов), и кнопки вызова меню и модификаторов [*menu/modified*] (для выполнения вспомогательных действий). Эту схему можно увидеть, если расположить контроллер вертикально и посмотреть на него, находясь в *VR*-редакторе (рис. А.3).

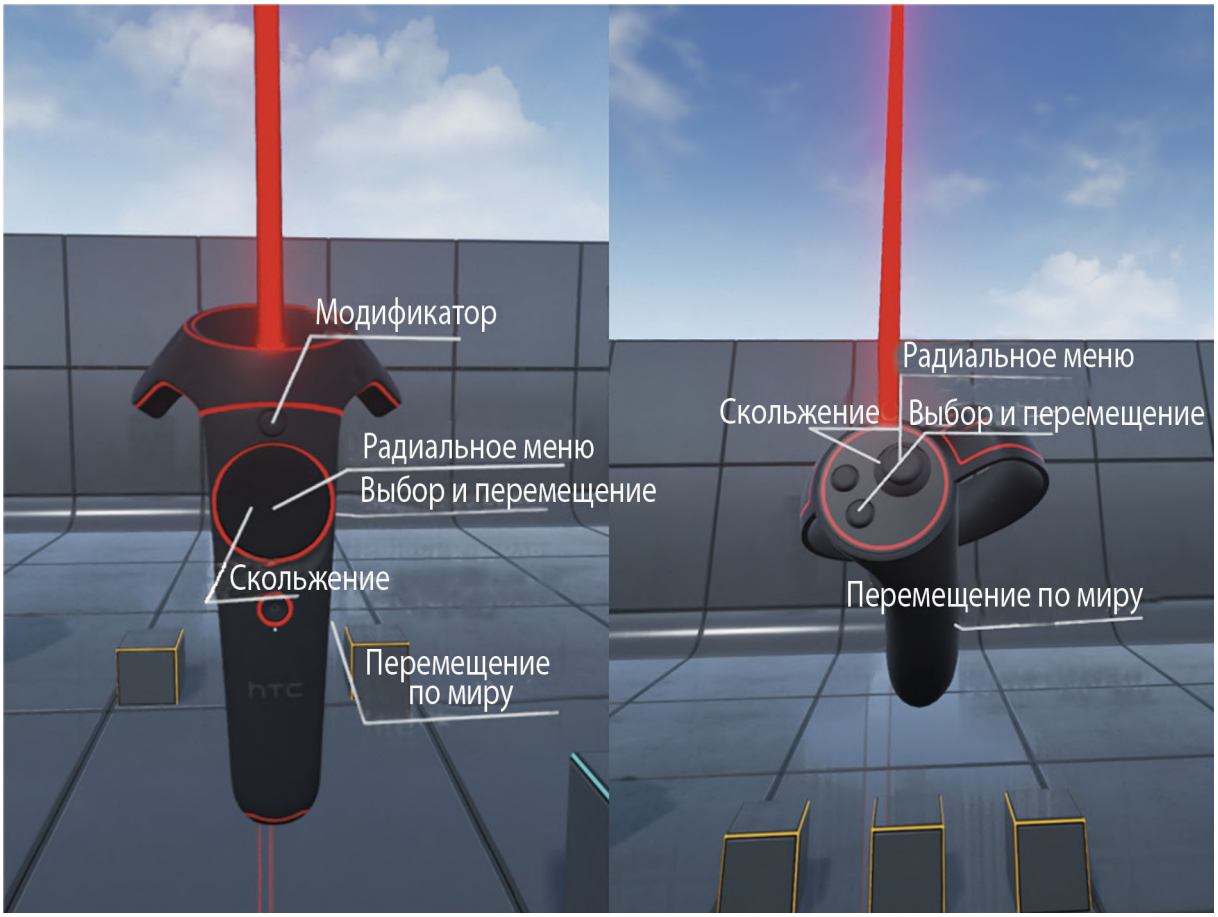

Рис. А.3. *VR*-редактор: схема использования контроллера (стандартная привязка действий)

А.2.1. Навигация в виртуальном мире

Для навигации в мире используйте кнопку *Move World* (обычно она привязана к *side button*). Зажав эту кнопку на одном контроллере и двигая его так, будто вы тянете себя сквозь мир, можно переместиться в любое место (рис. А.4). Нажатие этой кнопки на обоих контроллерах позволяет вам масштабировать мир (перемещением контроллеров друг относительно друга; рис. А.4) и вращать мир вокруг вертикальной оси (поворотом контроллеров относительно общей оси).

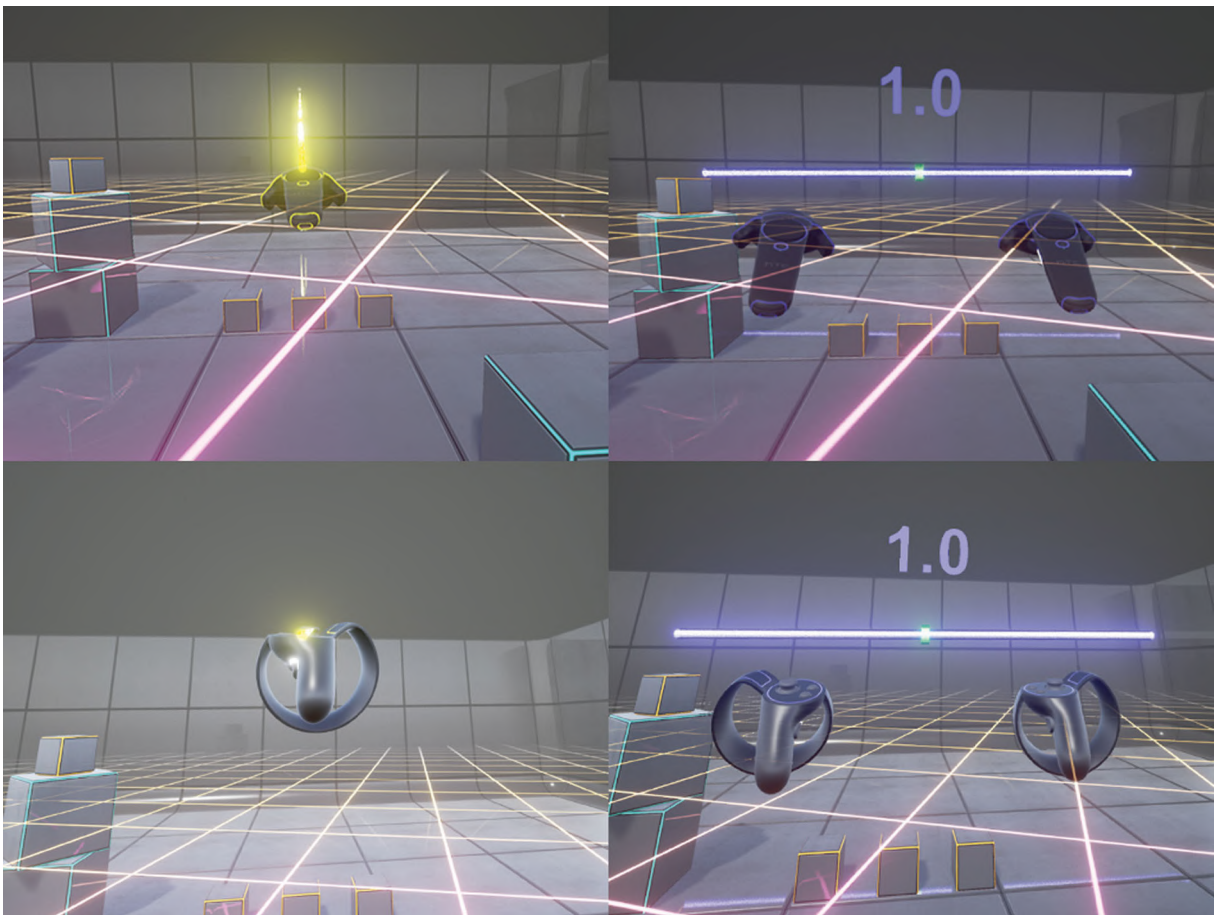

Рис. А.4. *VR*-редактор: навигация в виртуальном мире

А.2.2. Взаимодействие с объектами

Для взаимодействия с объектами в *VR*-редакторе выберите *Select* и *Move button* (обычно они привязаны к триггеру). Нажмите эту кнопку на контроллере наполовину для выбора объекта, с которым хотите взаимодействовать; для выбора нескольких объектов наполовину нажмите кнопку на одном контроллере и выберите объекты другим контроллером. Выбранный объект помечается виджетом (рис. А.5) и вы можете его перетаскивать, поворачивать и масштабировать, потягивая соответствующие ручки виджета. Другой способ перемещения объектов в редакторе — это войти в режим *Freeform* полным нажатием на триггер (см. рис. А.5), после чего объект входит в режим *Freeform*. В этом режиме вы можете перемещать объект, двигая контроллер или отдалять/приближать его к вам с помощью тачпада или джойстика на контроллере. Также вы можете выбрать объект одним контроллером, но масштабировать и вращать его, используя оба контроллера, подобно навигации в виртуальном мире.

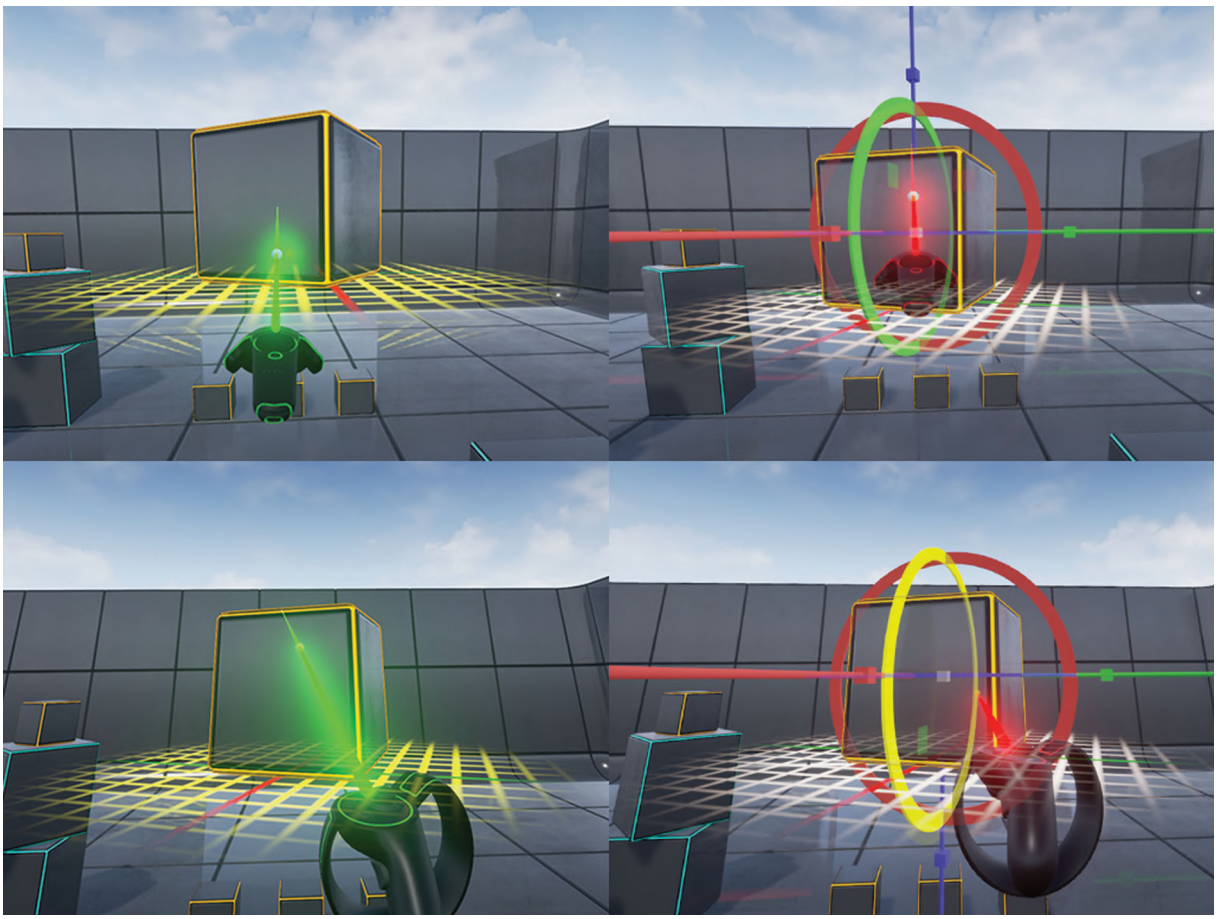

Рис. А.5. *VR*-редактор: взаимодействие с объектами

А.2.3. Взаимодействие с меню

VR-редактор имеет один тип меню (до версии 4.17 их было два): радиальное меню. Радиальное меню может быть открыто путем скольжения пальцем по тачпаду или движением джойстика, в нем объединили два старых меню (*Modes Panel*, *Asset Editors* и т. д.) (рис. А.6).

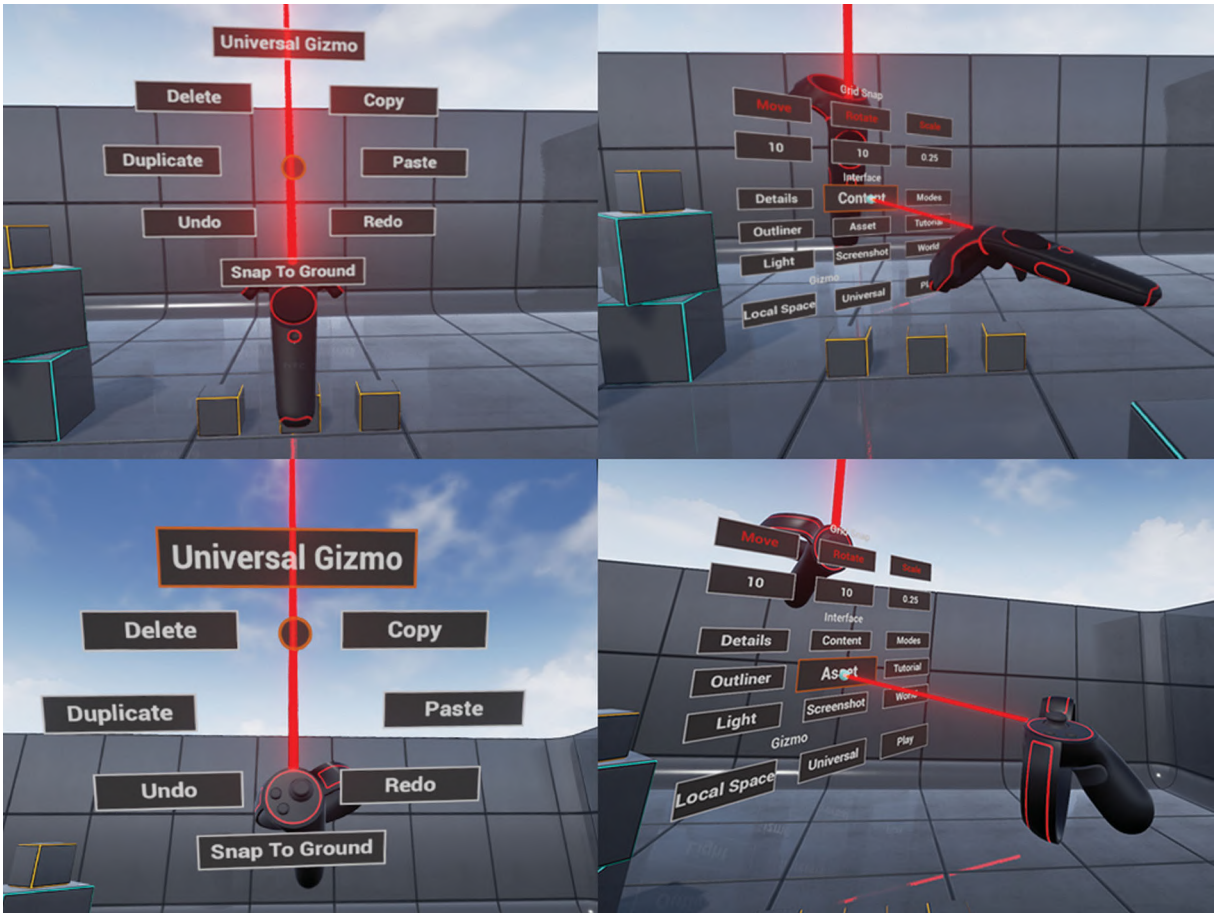

Рис. А.6. *VR*-редактор: доступ к радиальному меню

При выборе какого-либо параметра в разделе «Интерфейс» выбранный элемент 2*D*-интерфейса закрепляется на ваш контроллер (рис. А.7) и позволяет взаимодействовать с этим элементом другим контроллером при помощи лазерного указателя. Чтобы открепить элемент, выберите серый прямоугольник внизу элемента при помощи триггера и перенесите его с его позиции (рис. А.8). После того как элемент разблокирован, он остается в этом положении и позволит захватить больше элементов из меню быстрого доступа. Для быстрого доступа к общим элементам интерфейса с пользователем зажмите в центре тачпада или джойстика для переключения ключевых окон.

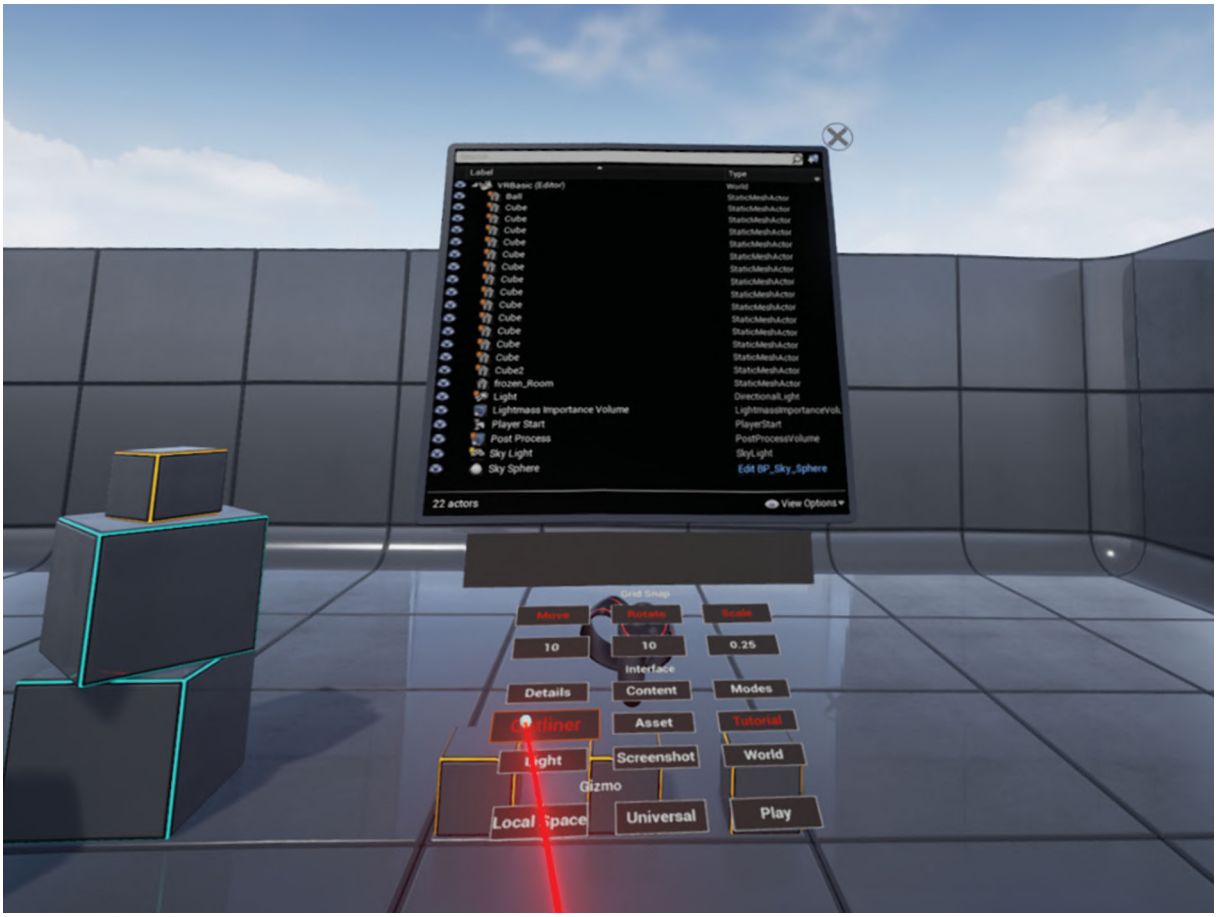

Рис. А.7. *VR*-редактор: взаимодействие с традиционным *UI*

Рис. А.8. *VR*-редактор: откреплены элементы *UI*

А.3. Заключение

В этом приложении мы рассмотрели *VR*-редактор, который позволяет создавать и редактировать виртуальные миры. Теперь у вас есть необходимые знания, чтобы начать изучение новых парадигм взаимодействия и создавать собственные *VR*-миры.

ПРИЛОЖЕНИЕ B

РЕСУРСЫ

Вам необходимо больше актуальной информации о *VR*-разработке? Это приложение поможет найти сообщества и ресурсы, которые станут прекрасным подспорьем на вашем пути к созданию совершенного *VR*-приложения.

B.1. *Epic*

Компания *Epic Games* предлагает много ресурсов, которые помогают преодолевать трудности и расширять знания при погружении в *UE*4 и *VR*. Документация поможет разобраться с особенностями движка, которые не освещены в этой книге, а разделы *AnswerHub* и *Forum* позволяет задать вопросы сообществу, участвующему в развитии *VR*-технологий. *Epic* также проводят еженедельные тренировочные (учебные) стримы на *Twitch/YouTube*, которые часто затрагивают тему *VR*.

1. Документация (*http://docs.unrealengine.com/en-us/Platforms/VR*).
2. *AnswerHub* — вопросы и ответы (*http://answers.unrealengine.com*).
3. Официальный форум о *VR*- и *AR*-разработке (*http://forums.unrealengine.com/development-discussion/vr-ar-development*?27-*VR-Development*).
4. *YouTube*-страница (*http://www.youtube.com/user/UnrealDevelopmentKit*).
5. Видео: начинаем разработку в *VR* (*http://www.youtube.com/watch*?*v=afodIcU_vK*4).

B.2. *Oculus*

Oculus предоставляет широкий спектр документации для своей платформы и *VR*-шлемов. Некоторые подробности могут быть излишними, потому что многие разработчики полагаются на реализацию различных *SDK* от *Epic*. Тем не менее доступное руководство по лучшим практикам [*Best Practices Guide*] проливает свет на многие важнейшие концепции для *VR* в целом. *Oculus* также проводит ежегодную конференцию разработчиков (*Oculus Connect*), на которой представляются интересные доклады ведущих *VR*-разработчиков. Записи докладов размещаются в интернете после окончания конференции (страница *Oculus* на *YouTube*).

1. Документация (*http://developer.oculus.com/documentation/*).
2. Лучшие практики (*http://developer.oculus.com/design/latest/concepts/book-bp/*).
3. Форум (*http://forums.oculusvr.com/community/discussions*).
4. *YouTube*-канал (*http://www.youtube.com/user/oculusvr*).
5. Доклады *Oculus Connect* 2017 (*http://www.youtube.com/watch*?*v=QAa*1*GjiLktc*&*list=PLL*2*xVXGs*1*SP*4 50*I0CPTIyIYX*3*yHCY*9*GQs*).
6. Доклады *Oculus Connect* 2018 (*http://www.youtube.com/watch*?*v=o*7*OpS*7*pZ*5*ok*&*list=PLL*2*xVXGs*1*SP* 6*KJXPd*8*n*1*Sxj*8*IpEQpOyy_*)

B.3. *Valve*

Valve еще не выпустила никаких руководств по лучшим практикам, но у нее есть сообщество *Steam* для разработчиков *Steam VR*, которое регулярно пополняется сообщениями в блоге. У *Valve* также есть почти ежегодная конференция (*Steam Dev Days*), которая позволяет разработчикам обсуждать все вопросы разработки игр с другими разработчиками.

1. Документация (*http://steamcommunity.com/steamvr*).
2. *Forums* (*http://steamcommunity.com/app/*358720*/discussions*).

3. *YouTube* (*http://www.youtube.com/user/SteamworksDev*).
4. *Steam Dev Days* 2017 (*http://www.youtube.com/watch?v=mJqo7XzwJd0&list=PLckFgM6dUP2ihiMeKHoyIdHvhRSyqwQsp*).

В.4. *Google*

У компании *Google* есть несколько страниц, которые документируют различные *SDK для VR*, и канал *YouTube*, на котором регулярно публикуют информацию и советы по разработке на *Daydream/Cardboard* для различных платформ. *Google* также проводит ежегодную конференцию разработчиков (*Google I/O*), на которой рассказывается о *VR*; записи докладов конференции загружаются на *YouTube*.

1. Документация (*http://developers.google.com/vr/develop/unreal/get-started*).
2. *YouTube*-страница разработчиков *Google* (*http://www.youtube.com/user/GoogleDevelopers*).
3. *Google I/O* 2018 (*http://www.youtube.com/watch?v=ogfYd705cRs*).

В.5. Сообщества

Бывает, что официально поддерживаемые ресурсы не содержат необходимой информации. Возможно, в этом случае на ваши вопросы смогут ответить в сообществах *VR*-разработчиков.

1. *General UE4 Developer Reddit* (*http://www.reddit.com/r/UE4Devs*).
2. *Oculus Developer Reddit* (*http://www.reddit.com/r/oculusdev*).

В.6. Офлайн-встречи

Будь то налаживание контактов, общение или изучения новых тем путем обсуждения докладов приглашенных спикеров, встречи лицом к лицу могут дать очень много как начинающему, так и профессиональному разработчику на *Unreal Engine*.

1. *UE4 Meetups* (*http://meetup.com/pro/UnrealEngine*).
2. *SVVR — Silicon Valley Virtual Reality* (*http://svvr.com*).

В.7. Конференции

Если вы ищете встречи уровнем выше, то посмотрите в сторону ежегодных технологических конференций, которые могут быть очень привлекательны для *VR*-разработчиков.

1. *GDC (VRDC) — Game Developers Conference* (*http://gdconf.com*).
2. *Oculus Connect* (*http://www.oculusconnect.com*).
3. *Steam Dev Days* (*http://steamcommunity.com*).
4. *Google I/O* (*http://events.google.com/io*).

В.8. Русскоязычные ресурсы и сообщества

1. *YouTube*-канал *Unreal Engine Rus* (*http://www.youtube.com/channel/UCLbkGIcYJxxL0tciH9RVebg*).
2. Русскоязычное сообщество *Unreal Engine* 4 (*http://uengine.ru*).
3. ВКонтакте: *Unreal Engine* 4 *(UE4)* Уроки, форум, документация (*http://vk.com/uengine*).
4. *Moscow Official Unreal Engine* 4 *User Group Meetup* (*http://www.meetup.com/ru-RU/Moscow-Official-Unreal-Engine-4-User-Group-Meetup/*)

ПРИЛОЖЕНИЕ С

ГЛОССАРИЙ

C.1. Основные понятия

1. Виртуальная реальность [*Virtual Reality*] — по возможности полная замена окружения пользователя на мир, генерируемый компьютером. Следует отличать от дополненной реальности, когда генерируемые компьютером объекты встраиваются в реальный мир.
2. Рендеринг [*rendering*] — процесс построения изображения по модели мира.
3. Тренажерная болезнь [*simulator sickness*] — разновидность морской болезни (укачивания), которая вызывается различием между симулируемым движением в тренажере (или виртуальном мире) и сигналами от вестибулярного аппарата человека.
4. Трассировка [*tracing*] — определение точек пересечения луча (некоторой полупрямой) с поверхностями объектов виртуального мира.
5. Отслеживание положения [*positional tracking*] — точное определение положения в пространстве головы (*VR*-шлема), рук (контроллеров) и других частей тела человека для совмещения движений полозователя и его автара в *VR*.

C.2. Термины *Unreal Engine*

1. Блюпринт [*Blueprint*] — визуальный язык среды *UE*4.
2. [*Motion Controller Component*] — контроллер движения.
3. [*VR Head Mounted Display* (*HMD*)] — *VR*-шлем.
4. Трасса [*trace*] — сущность, используемая при задании параметров трассировки. Применяется, например, при определении объекта в фокусе внимания игрока.
5. Мировая точка отсчета [*world origin*] — начало координат в системе координат виртуального мира (обычно вместе с базовыми ортами).
6. Виджет [*Unreal widget/gizmo/manipulator/transforming tool*] — специальный элемент интерфейса с пользователем, служащий для прямого манипулирования 3*D*-объектом, состоящий из визуализатора текущего положения объекта и «ручек», которые используются для перемещения, вращения, масштабирования и проведения других трансформаций объекта.

ПРИЛОЖЕНИЕ D

ОТЛАДКА ПРИЛОЖЕНИЙ БЕЗ ШЛЕМА

В этом приложении мы сделаем так, чтобы движение мышкой считывалось как движение головы в гарнитуре. Все входные действия, такие как нажатие *Grip* или *Trigger*, можно привязать к событиям клавиш клавиатуры или мыши, заменив одни события на другие.

D.1. Отладка

Для начала необходимо указать, какой контроллер поворота камеры нам использовать. В вашем проекте откройте *Pawn*, который выполняет роль игрока на вашем уровне и перейдите в *Event Graph*.

1. Добавьте геттер для компонтента *Camera* вашего *Pawn*.
2. От него вызовите сеттер для `UsePawnControlRotation`. Этот параметр у *Camera* отвечает за то, надо ли привязать вращение камеры к *Pawn*.
3. Соедините сеттер контактом выполнения с узлом *EventBeginPlay* (если у вас с этим узлом уже что-то используется, то просто поместите сеттер в конец вашей цепочки вызовов).
4. Создайте узел *IsHeadMountedDisplayEnabled* и вызовите от него `NOT`.
5. Последний созданый узел установите в `UsePawnControlRotation` (рис. D.1). Это обеспечит, что если *HMD* не доступен, то мы будем вращать *Pawn* с *Camera*.
6. После сеттера вызовите узел *SetTrackingOrigin* и в `Origin` укажите `Floor Level`.

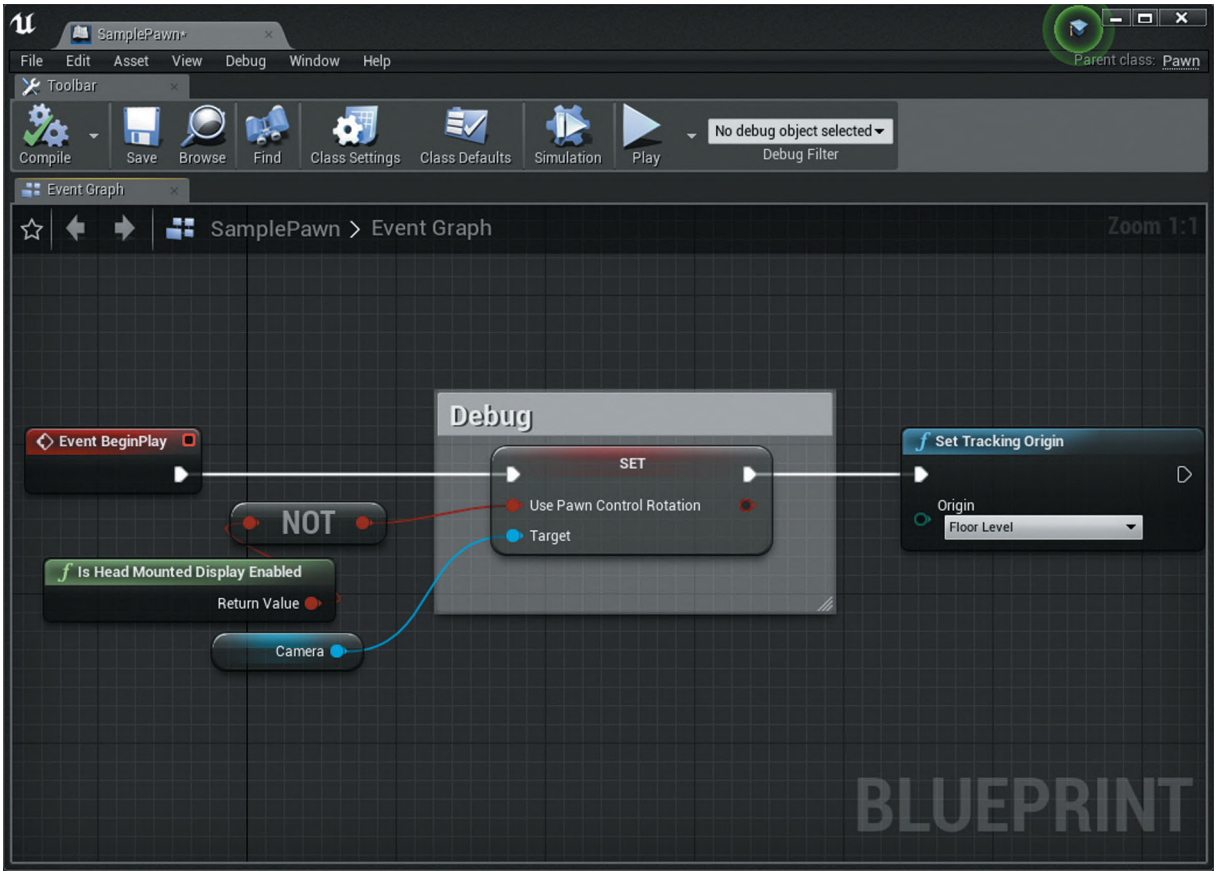

Рис. D.1. Установка контроллера вращения камеры

Теперь привяжем вращение камеры к движениям мыши. Для этого мы будем записывать движения мышки по оси `X` и `Y` вращения по соответствующим осям.

1. Добавьте два события: `MouseX` и `MouseY`.
2. От события `MouseX` создайте *Branch*.
3. Создайте узел *IsHeadMountedDisplayEnabled* и вызовите от него `NOT`.
4. Поместите выходное значение `NOT` в *Condition Branch*.
5. От ветки `True` вызовите *AddControllerYawInput*, указав в `Val Axis Value` события (рис. D.2).
6. Проделайте пункты 2–4 для `MouseY`.
7. От ветки `True` вызовите *AddControllerPitchInput*.
8. Умножьте `Axis Value` узла `MouseY` на `-1`.
9. Поместите полученный результат в `Val` узла *AddControllerPitchInput* (рис. D.2).

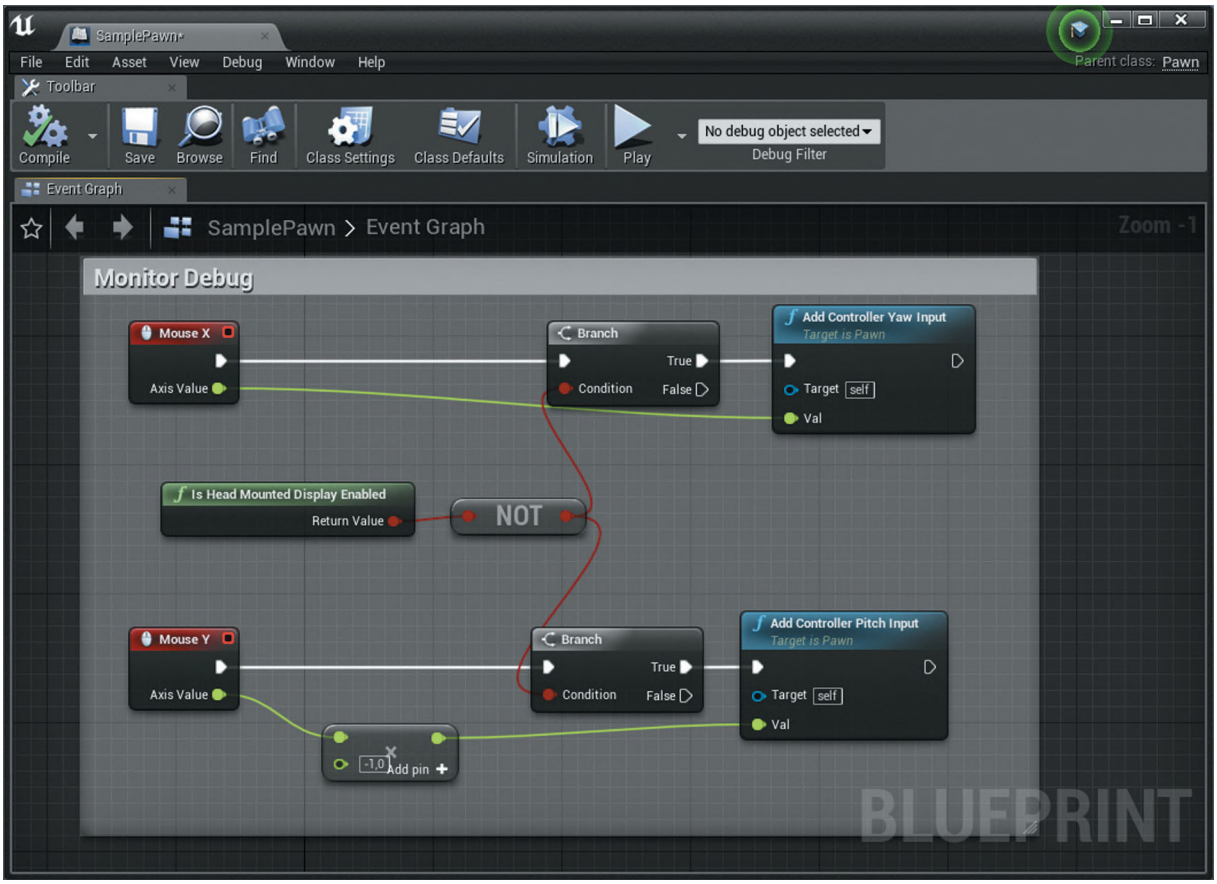

Рис. D.2. Установка вращения игрока с помощью мышки

Теперь у вас есть все, чтобы управлять игроком при помощи мыши так, как если бы вы делали это в шлеме. Осталось привязать события контроллеров к клавиатуре, чтобы отлаживать действия с ними. Единственное, что вы не можете сделать, — это заменить движение контроллеров, для этого уже придется подключать гарнитуру.

D.2. Заключение

В этом приложении мы рассмотрели способ отладки части действий без подключения *HMD*, а именно вращение камерой при помощи мыши.

D.3. Задания

Сейчас вы можете вращать ваш *Pawn* мышкой, но двигаться как в *VR*-шлеме, у вас не получится. Попробуйте привязать движение к клавишам клавиатуры самостоятельно.

ПРИЛОЖЕНИЕ Е

СПИСОК СОКРАЩЕНИЙ

1. *API (Application Programming Interface)* — интерфейс прикладного программирования.
2. *GPU (Graphics Processing Unit)* — графический процессор видеоадаптера.
3. *HMD (Head-Mounted Display)* — *VR*-шлем, шлем виртуальной реальности.
4. *HRTF (Head-related transfer function)* — передаточная функция, описывающая положение источника звука относительно слушателя.
5. *HUD (Head-up Display)* — индикатор на лобовом стекле (ИЛС); часть интерфейса с пользователем «на забрале виртуального шлема».
6. *OSIG (Oculus Signature File)* — файл подписи *Oculus*.
7. *SDK (Software Development Kit)* — комплект для разработки ПО.
8. *UI (User Interface)* — интерфейс с пользователем.
9. ОО [*OO*] — Объектно-ориентированный (как часть аббревиатуры).
10. ОП [*MM*] — оперативная память.
11. ОС [*OS*] — операционная система.
12. ПК [*PC*] — персональный компьютер.
13. ПО [*SW*] — программное обеспечение.
14. ПП [-] — программный продукт.
15. ЦП [*CPU*] — центральный процессор.

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Митч Макеффри

UNREAL ENGINE VR ДЛЯ РАЗРАБОТЧИКОВ

Главный редактор *Р. Фасхутдинов*
Ответственный редактор *В. Обручев*
Литературный редактор *С. Ульянов*
Младший редактор *Д. Атакишиева*
Художественный редактор *Е. Пуговкина*
Компьютерная верстка *Е. Матусовская*
Корректор *Р. Болдинова*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz
Интернет-дүкен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 18.04.2019. Формат 84x108 1/16.
Печать офсетная. Усл. печ. л. 26,88.
Тираж экз. Заказ

12+

BOOK24.RU
ИНТЕРНЕТ-МАГАЗИН
ИНТЕРНЕТ-МАГАЗИН
BOOK24.RU

ISBN 978-5-04-101419-3

EKSM0.RU
новинки издательства

КОГДА ВЫ ДАРИТЕ КНИГУ, ВЫ ДАРИТЕ ЦЕЛЫЙ МИР

ХОТИТЕ ЗНАТЬ БОЛЬШЕ?

Заходите на сайт:
https://eksmo.ru/b2b/

Звоните по телефону:
+7 495 411-68-59, доб. 2261

ВАШ ЛОГОТИП НА ОБЛОЖКЕ

ВАШ ЛОГОТИП НА КОРЕШКЕ

ОБРАЩЕНИЕ К КЛИЕНТАМ НА ОБЛОЖКЕ

«Восхищаюсь, что Митч смог найти время и перенести знания и опыт работы с Unreal Engine и виртуальной реальностью в книгу «Unreal® Engine VR для разработчиков»... Митч обладает уникальной квалификацией, чтобы поделиться этой книгой с миром».

ЛУИС КАТАЛЬДИ, Unreal Engine Education, Epic Games, Inc.

Митч Маккефри проделал отличную работу! Книга очень легко читается, каждая глава содержит всю достаточную и необходимую теоретическую информацию по рассматриваемым в книге топикам, не скатываясь только до банальных пошаговых инструкций «сделай так!». А подробные иллюстрации помогают понять, что происходит, даже если вы начали читать ее не с самого начала. Благодаря тому что весь материал максимально структурирован и проиллюстрирован, книга отлично подойдет как в качестве методических указаний для занятий в учебных заведениях, так и для самостоятельного обучения!

АНДРЕЙ КАРСАКОВ, к.т.н., руководитель магистерской программы «Технологии разработки компьютерных игр», Университет ИТМО

VR – новый, удивительный рубеж для разработчиков игр и специалистов по визуализации. А Unreal® Engine 4 – идеальная платформа для этого. ***«Unreal® Engine VR для разработчиков»*** – это исчерпывающее руководство по созданию потрясающих приложений на любых VR-устройствах, совместимых с Unreal Engine 4.

Известный VR-разработчик Митч Маккефри собрал вместе лучшие практики, общие парадигмы взаимодействия и руководства по реализации этих парадигм в Unreal Engine и практические советы по выбору решений, максимально подходящих для ваших проектов. «Рецепты» Митча представляют собой пошаговые инструкции и расширяют ваше понимание теории и математики.

Методики, изложенные в этой книге, помогают вам быстро получить результат в любой области применения – будь то создание шутера от первого лица или симулятора релаксации. Используя эти знания, вы сможете добиться успеха в любом жанре или проекте.

ВЫ ОСВОИТЕ:

- Понимание базовых концепций и терминологии VR
- Реализацию VR-логики на визуальном программировании Blueprint
- Создание базовых VR-проектов под Oculus Rift, HTC Vive, Gear VR, PSVR и других платформ
- Распознавание и управление различиями между сидячим и стоячим VR-опытом
- Настройку взаимодействия и телепортации
- Работу с UMG и 2D UI
- Реализацию инверсной кинематики для головы и рук
- Определение эффективного взаимодействия с помощью контроллеров
- Помощь пользователям, чтобы они могли избегать тренажерной болезни
- Оптимизацию VR-приложений
- Описание VR-редактора
- Ресурсы комьюнити и многое другое

Дополнительные материалы доступны по ссылке https://eksmo.ru/files/ue4_vr_projects.zip

ISBN 978-5-04-101419-3

Addison-Wesley

БОМБОРА

Бомбора — это новое название Эксмо Non-fiction, лидера на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks
www.bombora.ru

МИТЧ МАККЕФРИ создал сообщество VR Template for UE4 и популярную серию туториалов на YouTube Mitch's VR Lab. В качестве активного члена сообщества UE4 VR, он активно преподает лучшие методики VR, используя посты на форумах, сообщество VR Template и канал на YouTube.