

DEVELOPING VIRTUAL REALITY WITH **UE4**



UNREAL® ENGINE

VR

Cookbook



Mitch **McCAFFREY**

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Unreal[®] Engine VR Cookbook

Developing Virtual Reality with UE4

Mitch McCaffrey



◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corp-sales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016958916

Copyright © 2017 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

The following are registered trademarks of Google: Android™, Daydream™, Google Cardboard™.

Steam VR is a registered trademark of Valve Corporation.

Oculus Rift is a registered trademark of Oculus VR, LLC.

Samsung Gear VR is a registered trademark of Samsung Electronics Co., Ltd.

PlayStation VR is a registered trademark of Sony Interactive Entertainment Inc.

Unreal Engine is a registered trademark of Epic Games, Inc., in the United States of America and elsewhere.

ISBN-13: 978-0-13-464917-7

ISBN-10: 0-13-464917-6

Editor-in-Chief

Mark L. Taub **Executive Editor**

Laura Lewin **Development Editor**

Sheri Replin **Managing Editor**

Sandra Schroeder **Full-Service Production Manager**

Julie B. Nahil **Project Editor**

Anna Popick

Copy Editor

Barbara Wood

Indexer

Dick Evans

Proofreader

Anna Popick

Technical Reviewers

Luis Cataldi

Jeff Wilson

Clinton Crumpler **Editorial Assistant**

Olivia Basegio **Cover Designer**

Chuti Prasertsith **Compositor**

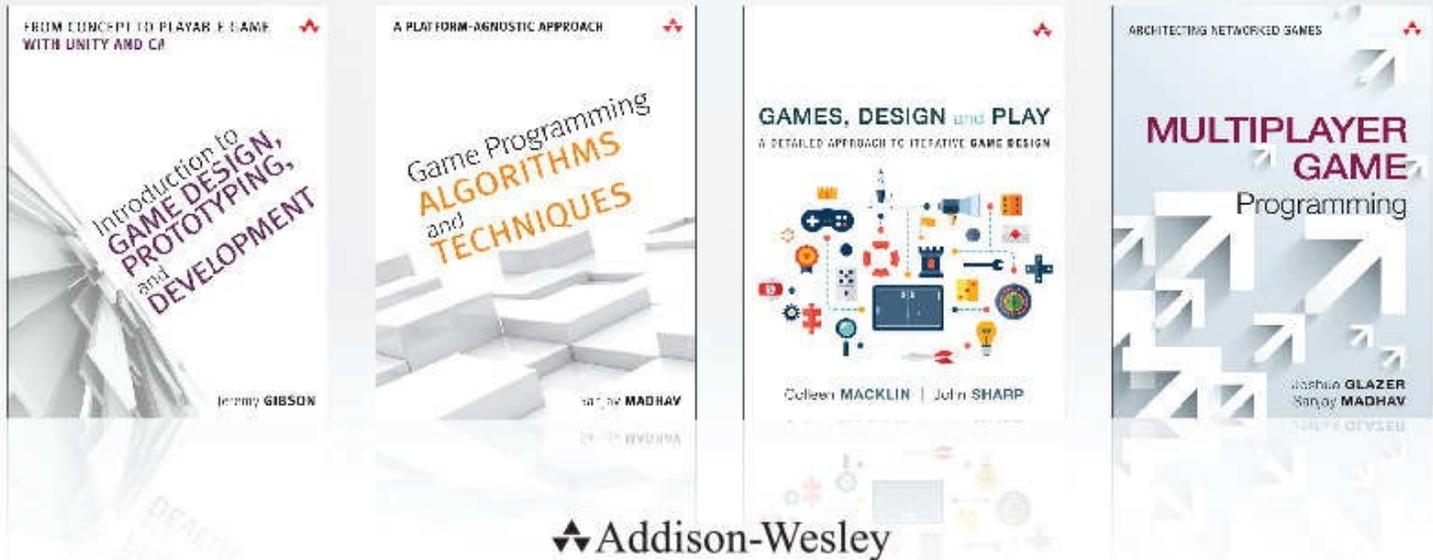
The CIP Group

Praise for *Unreal[®] Engine VR Cookbook*

“I’ve been a fan of Mitch’s work for quite some time. Back in early 2014, Mitch was exploring new locomotion mechanics in Unreal Engine for VR characters and sharing his findings, as well as numerous sample files, on the www.unrealengine.com forums in massive ongoing threads. Even in the early days of VR exploration, his work was helpful to many newcomers working to understand the issues of designing for a comfortable virtual experience. With his YouTube channel, *Mitch’s VR Labs*, he helped thousands of people understand the foundations of locomotion and interaction mechanics with clear and concise UE4 videos. I’m thrilled that he has taken the time to bring all his knowledge and experience in working with Unreal Engine and virtual reality to the *Unreal[®] Engine VR Cookbook*. With the current attention and appetite for understanding how best to work in the exciting medium of virtual reality with Unreal Engine, I think Mitch is uniquely qualified to share this book with the world.”

—**Luis Cataldi**, Unreal Engine Education, Epic Games, Inc.

The Addison-Wesley Game Design and Development Series



Visit informit.com/series/gamedesign for a complete list of available publications.

Essential References for Game Designers and Developers

These practical guides, written by distinguished professors and industry gurus, cover basic tenets of game design and development using a straightforward, common-sense approach. The books encourage readers to try things on their own and think for themselves, making it easier for anyone to learn how to design and develop digital games for both computers and mobile devices.



Make sure to connect with us!
informit.com/socialconnect

Contents

[Preface](#)

[Acknowledgments](#)

[About the Author](#)

[Part I: Getting Started](#)

[1 Terminology and Best Practices](#)

[Terminology](#)

[Devices](#)

[Software](#)

[Unreal Engine](#)

[Best Practices](#)

[Summary](#)

[2 Head Mounted Display Setup](#)

[Gear VR](#)

[Gear VR Project Setup](#)

[Gear VR Global Menu Setup](#)

[Gear VR Global Menu Progress Material](#)

[Rift and Vive](#)

[Rift and Vive Project Setup](#)

[Rift and Vive Tracking Origins](#)

[Summary](#)

[3 Toolkit](#)

[Generic Function Library](#)

[Oculus Function Library](#)

[Steam VR Function Library](#)

[Summary](#)

[Part II: Recipes](#)

[4 Trace Interaction](#)

[Understanding Trace Interaction](#)

[Understanding Interfaces](#)

[Setting Up Trace Interaction](#)

[Basic Project Setup](#)

[Interaction Interface Setup](#)

[Interaction Component](#)

[Interaction Pawn Setup](#)

[Setting Up a Basic Interactive Object](#)

[Summary](#)

[Exercises](#)

[5 Teleportation](#)

[Setting Up Teleportation](#)

[Parabolic Tracing](#)

[Visualizing the Teleport](#)

[Visualization Material](#)

[Visualization Actor](#)

[Simple Teleportation Volume](#)

[Summary](#)

[Exercises](#)

[6 Unreal Motion Graphics and 2D User Interfaces](#)

[Challenges with 2D UI in VR](#)

[History and Compatibility of UMG](#)

[Basic VR Menu](#)

[Menu Actor](#)

[Menu Pawn](#)

[Custom Menu Interaction](#)

[Implementing Custom Menu Interaction: Approach 1](#)

[Implementing Custom Menu Interaction: Approach 2](#)

[Summary](#)

[Exercises](#)

[7 Character Inverse Kinematics](#)

[Introduction to Inverse Kinematics](#)

[Setting Up Head IK](#)

[Mirror Creation](#)

[IK Pawn](#)

[Head IK Animation Blueprint](#)

[Setting Up Hand IK](#)

[Adding Motion Controllers to Your Pawn](#)

[Hand IK Animation Blueprint](#)

[Summary](#)

[Exercises](#)

[8 Motion Controller Interaction](#)

[Why Motion Controller Interaction Works](#)

[What to Look Out For: The Importance of Affordance](#)

[Shared Input of the Current Generation of Motion Controllers](#)

[Setting Up the World Interaction Project](#)

[Interacting with Objects](#)

[Creating the World Interaction Interface](#)

[Creating the Interactor Component](#)

[Adding Interaction to the Interaction Pawn](#)

[Creating the Interactive Objects](#)

[Creating an Interactive Static Mesh Actor](#)

[Creating an Interactive Button](#)

[Creating an Interactive Lever](#)

[Summary](#)

[Exercises](#)

[9 VR Locomotion](#)

[Simulator Sickness](#)

[Locomotion Types](#)

[Natural](#)

[Teleportation](#)

[Vehicle](#)

[Physical](#)

[Artificial](#)

[Locomotion Implementation](#)

[First Person Template for Snap Turning](#)

[First Person Template for Running in Place](#)

[Summary](#)

[Exercises](#)

[10 VR Optimization](#)

[Requirements of VR Rendering](#)

[Latency Mitigation](#)

[Performance Improvements](#)

[Forward versus Deferred](#)

[Instanced Stereo](#)

[Hidden Area Mesh Optimization](#)

[VR Project Settings](#)

[Summary](#)

[Exercises](#)

Part III: Appendices

A VR Editor

[Enabling the VR Editor](#)

[Controlling the VR Editor](#)

[Navigating the World](#)

[Interacting with Objects](#)

[Interacting with Menus](#)

[Summary](#)

B Resources

[Epic](#)

[Oculus](#)

[Valve](#)

[Google](#)

[Community](#)

[Physical Meetups](#)

[Conferences](#)

[Index](#)

Preface

The resurgence of virtual reality (VR) is upon us, and so an explosive demand for new and compelling VR experiences is coming. A growing billion-dollar medium, VR brings new and exciting challenges to the world of game development, while bringing previously uninitiated industries into the exciting realm of real-time rendering. Whether it's creating a classic-style arcade game or visualizing an exquisite villa in the grass-green hills of Tuscany, VR brings an unparalleled level of immersion to any experience. This immersion, however, brings with it a slew of challenges. When you develop for VR, you write the rulebook as you go.

In this book is a set of best practices and common interaction paradigms that have emerged over the past few years. Whether from the big players in the space or from a one-person team, the entire VR community is contributing to the compendium of VR knowledge at an astonishing pace. This book not only shows you how to implement these paradigms in Unreal Engine, it also shows you how to choose which one will suit your project.

Presented in cookbook style, this book takes a practical approach to learning the nuances of VR development. Each recipe shows you how to build a common system that is used in many VR games/experiences today. Whether you are building a first-person shooter or a relaxation simulator, each example keeps the content abstract enough that it applies to any genre, yet also mentions specific approaches that may work well for certain game types.

Who Should Read This Book?

This book is designed for people already familiar with navigating Unreal Engine 4 (UE4) and Blueprints. If you have little experience with either, visit the Unreal Engine documentation before reading this book. However, I explain most things when it comes to actual coding, and most of the mathematics is covered in sidebars and the main content. Therefore, a deep level of coding is not required.

How This Book Is Organized

This book is split into three parts:

- **Part I, “Getting Started”**: [Chapters 1](#) through [3](#) contain an introduction to some of the terminology used within this book and the VR industry. This part also contains instructions for creating basic projects for various VR headsets.
- **Part II, “Recipes”**: [Chapters 4](#) through [10](#) contain the main recipes for the book. This part covers everything from motion controller interaction to VR movement schemes.
- **Part III, “Appendices”**: This auxiliary information on the VR Editor and resources will help you on your journey in VR development.

Conventions Used in This Book

The following typographical conventions are used in this book:

- Monospaced text indicates blocks of code.
- *Italicized* text indicates a new key word or phrase.

Note

A Note signifies a tip, suggestion, or general note.

Sidebar

A sidebar contains auxiliary information for the main text, such as an explanation of the mathematical principles used or work related to the main content.

Warning

A Warning indicates a warning or caution.

Why Blueprints?

When programming in UE4, there are two main ways to implement logic in your games/experiences: the visual scripting language, Blueprints, and the more traditional coding language, C++.

Compared to Blueprints, C++ can be a little obscure because the required syntax can take a while to learn; however, C++ gives you greater access to some of the more hidden features of the engine. This, however, won't be a problem in this book; most of what is taught will be at a high enough level that Blueprints will suit your needs.

Blueprints also offers an easier method for migrating your work from one project to another, which will allow you to take any work done in this book and easily apply it to your existing content.

Companion Website

<https://github.com/mitchemmc/UE4VRCookbook> holds the various source files for every chapter that needs them. This website allows you to check your work for each recipe.

Register your copy of *Unreal® Engine VR Cookbook* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134649177) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

Acknowledgments

I am grateful to Laura Lewin, executive editor at Addison-Wesley, and Luis Cataldi, education evangelist at Epic Games, who gave me the opportunity to publish this book with Addison-Wesley. I thank the team from Addison-Wesley. Sheri Replin, the development editor, reviewed all the chapters and gave me valuable suggestions on the content presentation. Olivia Basegio helped me coordinate with the team at Addison-Wesley and organize the necessary paperwork.

This book could not have been published without feedback from technical reviewers Clinton Crumpler and Marty Murphy. They identified errors and provided valuable suggestions. Thank you.

Finally, I thank my significant other, Nicola, for being understanding and supportive while late nights were spent working on this book.

About the Author

Mitch McCaffrey is an independent game developer and creator of many community resources for Unreal Engine VR developers. He teaches game development best practices with his popular YouTube channel *Mitch's VR Lab* and demonstrates these in his community-driven VR Content Examples. His website is <http://mitchvr.com>.

Part I: Getting Started

Chapter 1. Terminology and Best Practices

The virtual reality (VR) development world can be daunting because of the sheer amount of competing hardware and software that is currently available. Also, because VR is such a new medium, many of the axioms that game developers take for granted may not work in the world of VR games and experiences.

If you're unsure of the differences among Oculus VR, OSVR, and OpenVR, or you're just looking for some common best practices as you start out with VR, this chapter is for you.

Terminology

Many technologies, pieces of software, and devices constitute the ever-growing VR ecosystem. To make sure that we are on the same page, let's look at a few key pieces of this ecosystem that any Unreal Engine 4 (UE4) VR developer should be aware of. If you are already familiar with the current state of the VR industry and the technologies involved, feel free to skip this chapter.

Devices

When creating your experiences, you can choose from various hardware devices, whether it's a certain VR Head Mounted Display (HMD) or a VR-ready controller. Unreal Engine supports most of them out of the box, which relieves developers of the burden of choosing the right device in the initial stages of a project. UE4 also has a nice VR abstraction layer, so deciding to change the device you're targeting (or targeting multiple devices at any stage in the pipeline) can be an easy task.

These natively supported HMDs are described in [Table 1.1](#).

HMD	Description
Samsung Gear VR	<p>The Gear VR is a hardware product compatible with Samsung smartphones that was developed in a partnership between Oculus and Samsung to bring Oculus's advances in software to mobile VR. When targeting the Gear VR, you can use the Oculus Mobile software development kit (SDK) to take advantage of these software features. The Gear VR currently supports only rotational tracking with a virtual head model for positional offset.</p>
HTC Vive	<p>The HTC Vive was developed as a result of a partnership similar to the Gear VR's but with HTC providing the hardware and Valve providing the software/SDK in the form of SteamVR/OpenVR. The Vive provides rotational tracking as well as one-to-one positional tracking for the headset and included motion controllers. To target the Vive, Valve provides the OpenVR SDK (for more information on OpenVR, see Table 1.3), which gives access to various sensor and setup data needed for a VR experience.</p>
Oculus Rift	<p>The Oculus Rift has gone through many iterations in the form of developer editions that were not meant for consumers. In 2016, however, Oculus released the Consumer Version 1 (CV1) that targeted consumers and allowed them to purchase the device. The CV1 is a fully integrated system with Oculus manufacturing the hardware and creating the software/SDK (see the Oculus SDKs in Table 1.3). The Rift provides rotational and positional tracking similar to the HTC Vive's. It also includes an Xbox One controller and Oculus Remote as the default input devices; the Oculus Touch motion controllers (see Table 1.2) ship separately.</p>

Google Cardboard	The Google Cardboard is a small, usually \$5 to \$20 piece of cardboard that includes a simple button and lenses. It allows you to attach your phone and view stereoscopic content. The Cardboard relies on the smartphone's built-in sensors for detecting the user's head rotation (which can be inaccurate if not calibrated). It also relies on native Android SDKs to provide a wide compatibility with devices (which can lead to higher latency rendering than other solutions). However, the Cardboard is by far the cheapest HMD.
Google VR/Daydream VR	Google VR or Daydream VR is a more tightly controlled software and hardware ecosystem than its alternative (Google Cardboard). For hardware, both the device (phone) and headset must be classified as "Daydream Ready," which allows Google to pursue more advanced software features. Daydream also has support for the Daydream controller (see Table 1.2), which provides a touchpad and rotational tracking. To target Daydream you use the Google VR SDK (see Table 1.3).
PlayStation VR	PlayStation VR is a peripheral for playing VR games on the PlayStation 4 and PlayStation 4 Pro game consoles. Positional tracking is obtained by using an external PlayStation Camera, and rotational tracking is enabled through the use of IMUs (inertial measurement units) similarly to the other headsets in this table. Players can also use PlayStation Move controllers (see Table 1.2) to take full advantage of motion controller experiences. The DualShock 4 can also be used; it has the capability of positional tracking as well.
OSVR	Open Source Virtual Reality (OSVR) is an initiative from several industry partners. Currently, OSVR offers the Hacker Development Kit (HDK) in the form of HDK1.3 and HDK2, developed by Razer. Both headsets offer rotational and positional tracking.

Table 1.1 Supported VR HMDs

In UE4, motion controllers are supported through a single Motion Controller Component that makes it easy to target multiple controllers. For a breakdown of the motion controllers that UE4 supports, see [Table 1.2](#).

Motion Controller	Description
Oculus Touch	The Touch controllers allow for both rotational and positional tracking through Oculus's Constellation tracking system. Each controller has two buttons (A, B, X, and Y across both controllers), an analog thumbstick, an index trigger, and a hand trigger. Support for the Touch controllers is enabled through the Oculus SDK.
Vive	The Vive controllers offer both rotational and positional tracking through Valve's Lighthouse tracking system. Each controller contains a circular trackpad that acts as an analog input as well as a button when pressed. The Vive controller also offers a trigger, menu button, and grip button. The Vive controller is enabled through the OpenVR SDK.
PlayStation Move	The Move controller offers both rotational and positional tracking through the PlayStation Camera and the RGB colored orb that sits on top of the controller. Each controller has a Cross, Circle, Triangle, and Square button like other PlayStation controllers. It also features Move, Start, and Select buttons as well as a trigger (or T button).
Daydream	The Daydream controller offers only rotational tracking. It also offers a touchpad, which can also act as a button, and an individual App button below that. The Daydream controller is enabled through the Google VR SDK.

Table 1.2 Supported Motion Controllers

Software

Many SDKs/libraries/APIs (application programming interfaces) help you interface with your VR hardware. UE4 tends to abstract these into singular interfaces or components that allow for easy interoperability; however, if you need to, you can still interact with the various SDKs manually. It is useful for you as a developer to understand the differing design philosophies in these pieces of software, because when developing your game or experience you may need to take advantage of specific features of a particular SDK. To interact with these SDKs manually, you don't need to download any separate files; UE4 includes them when you download the engine.

These natively supported SDKs are listed in [Table 1.3](#).

SDK	Description
Oculus PC SDK	The Oculus PC SDK provides the engine with information UE4 needs to render the appropriate view. This includes things like the user's head position and orientation as well as the user's IPD (interpupillary distance). It also allows UE4 to submit the render frame to the Oculus compositor and have it apply the distortion needed to correct the HMD's lenses. Luckily the engine handles this, and in most use cases you do not have to worry about it. The Oculus SDK also provides layers that give you the ability to draw different things to the screen with different resolutions; this is useful for user interfaces (UIs) where you may want to render text at a higher resolution than the background. Features such as ATW (Asynchronous Timewarp; see Table 1.4) are handled by the runtime automatically, and therefore you do not need to worry about enabling them.
Oculus Mobile SDK	The Oculus Mobile SDK gives you access to many of the same things as the PC SDK, including the head orientation and other user information needed for rendering. Similar to the PC runtime, the mobile runtime uses ATW; however, it also supports front buffer rendering, also called scanline racing (see Table 1.4).
Oculus Audio SDK	The Oculus Audio SDK provides a way to realistically spatialize and create 3D audio for VR. It allows you to apply an HRTF (head-related transfer function) to your audio sources to simulate directional audio (through the modelling of a human ear's geometry). It also simulates the distance from the sound to the ear by attenuating loudness. The Oculus Audio SDK's HRTF is implemented in UE4, and you have the ability to create distance curves manually in the engine; however, currently this implementation is only for DirectX-based projects (for example, PC).

OpenVR SDK	The OpenVR SDK is similar to the Oculus PC SDK in that it gives you access to the needed information of the HMD (such as position and orientation). The OpenVR SDK also has interfaces that give you access to the Chaperone (Valve’s virtual bounds that denote a user’s play space) and the Overlay to render 2D content to the compositor similar to Oculus’s layers. OpenVR also supports reprojection (similar to ATW) and prediction.
OSVR SDK	The OSVR SDK gives you access to a variety of interfaces for creating VR experiences. Because of its “support everything” nature, it includes the more essential interfaces for getting HMD information such as position and orientation, but there are also interfaces for eye tracking, omnidirectional treadmills, gestures, and the like. The OSVR SDK integration was added in Unreal Engine 4.12.
Google VR SDK	The Google VR SDK gives you access to head tracking data of the HMD as well as various features to optimize your experiences. It allows you to enable sustained performance mode, which is important on a phone because having the phone throttle from thermal limits can badly affect the user experience. It also gives you access to the scanline racing functionality, which optimizes latency in the rendering pipeline (see Table 1.4).

Table 1.3 Supported SDKs

Aside from the SDKs and libraries that UE4 uses to interact with the various bits of VR hardware, there are also specific software features that UE4 implements in the engine to enhance the VR experience. However, there are also other software features (such as ATW) that are implemented by the various runtimes and are therefore enabled by default and out of the control of developers. [Table 1.4](#) shows both types of software available in UE4.

Feature	Description
Instanced stereo rendering	Instanced stereo rendering works by rendering both the left and right eyes of the player's view simultaneously instead of sequentially. This can lead to a gain in both CPU and GPU render times. It is a feature that is supported natively by UE4 but is disabled by default because some rendering features do not work with it. In some cases it can lead to worse GPU performance, so it is important to test your own content.
Hidden and visible mesh optimization	The hidden area mesh optimizations in UE4 allow the renderer to reject the pixels that won't be seen because of the distortion of the HMD's lens, ultimately saving GPU time. These optimizations are enabled by default and nothing needs to be done to use them.
Timewarp/reprojection	Both timewarp (used with the Oculus runtime) and reprojection (used with the OpenVR runtime) allow the renderer to make up for a dropped frame (when the app doesn't render in time for the display's next refresh) instead of showing the same frame twice, causing judder. Both runtimes reproject the previous frame to the user's latest head rotation, providing a smoother experience (the differences between these two techniques are subtle and are explored in Chapter 10). As mentioned, these are handled by the respective runtimes, so nothing has to be done to enable them. Note, however, that developers should not rely on these features to fix bad frame rates and instead should use them as a fallback for performance issues that are out of their control.

Front buffer rendering/
scanline racing

When rendering to a display in regular 3D applications, to avoid an incomplete image being displayed to the user, images can be rendered to a back buffer before being swapped into the front buffer (for sending to the screen); however, this provides at minimum a frame of latency to your VR applications (for a deeper explanation see Chapter 10). To avoid this, the major (mobile, for the time being) SDKs render directly to the front buffer (literally race the pixel-by-pixel, line-by-line readout from the front buffer to the display and hence “race the scan”). This can be accomplished because of the predictability of some of the reprojection and prediction techniques implemented when rendering a VR frame. The Oculus Mobile SDK handles this natively, but it is an option you must enable for Google VR.

Deferred/forward rendering

There are two main techniques when rendering a 3D scene: forward and deferred. Initially most game engines were based on the forward approach because of its simplicity and efficiency for small scenes; however, many engines have transitioned to or now include a deferred approach because it allows for many more dynamic lights and some advanced screen space effects such as SSRs (screen space reflections). However, because performance is extremely important in VR, forward rendering can be a more compelling option. On mobile, forward rendering is the only option, so for Gear VR and Google VR you’re covered. As for desktop, UE4 is natively a deferred renderer. However, Epic is working on a forward renderer for desktop that will generate big performance enhancements when it is ready.

Table 1.4 Supported Software Features

Note

Many of these features are further explored in [Chapter 10](#), “[VR Optimization](#).”

Unreal Engine

Making games and experiences requires many different systems to work together; luckily, UE4 has some great tools to interact with these systems. [Table 1.5](#) describes the systems used in this book.

System	Description
Unreal Motion Graphics (UMG)	UMG is the data-driven UI system in UE4. To use UMG widgets, you can use the UMG UI Designer, which allows for the easy creation of UIs without using any code. UMG also supports key frame animation and dynamic UI scaling.
Blueprints	Blueprints is UE4's visual scripting language. It allows for very fast iteration and is a nice way to not get bogged down in language syntax (you will use Blueprints almost exclusively in this book).
Sequencer	Sequencer is UE4's advanced cinematics tool. It allows you to create sequences of animations through key framing and allows you to cut and move them in a similar style to a traditional NLE (nonlinear editor).
Persona	Persona is UE4's animation editor. It has loads of functionality, from allowing you to easily blend animations to creating inverse kinematics systems.

Table 1.5 Unreal Engine Systems Used in This Book

Best Practices

VR is an emerging industry. Much experimentation has to take place before we settle on “the” way to do things. A lot of the time, your initial great idea of how well a certain mechanic or functionality will work in VR falls flat, and at the same time things that you never thought would work well can work splendidly when you finally implement and try them.

That being said, however, through research and tests, both Oculus and Epic have found some things that the majority will find discomfoting when in VR. At the time of writing, these articles can be found here:

<https://developer.oculus.com/documentation/intro-vr/latest/>

<https://docs.unrealengine.com/latest/INT/Platforms/VR/ContentSetup/index.html>

The UE4 best practices guide is a good place to start for any Unreal Engine VR developer, because many of the topics covered are specific to UE4 and how to optimize your experience for VR. A lot of these, such as disabling HZB (Hierarchical Z-Buffer) and instanced stereo rendering, are covered in more detail in [Chapter 10](#). If you are looking for more general tips for VR experiences, these can be found in the Oculus best practices guide.

Here is a summary of these points:

- Avoid screen space effects when you can; they can look bad in the best case, and in the worst, they can cause stereo mismatch between the eyes.
- Aim for a frame rate slightly higher than the target display frame rate. This ensures that any hiccups won't cause major discomfort (don't rely on reprojection technology).
- Consider lowering the rendered resolution if the frame rate needs to be increased. This can be done with the console command `r.screenPercentage` in UE4 (see [Chapter 10](#)).
- The player's head motions should be in control of the camera at all times. Avoid using cinematic cameras, and try to use asynchronous loading in menus or during level loads. Avoid camera shake and other effects that move the camera without the user's control.

- Do not override the field of view of the player's view; this causes discomfort when rotating.
- Avoid accelerations; they create an ocular vestibular mismatch (a mismatch between what the players' eyes see and what their ears tell them) and cause nausea (for more on this, see [Chapter 9](#), "[VR Locomotion](#)").

As always, both documents contain great information for any VR developer. I recommend that you read them in their entirety.

Summary

You are now familiar with some of the terminology that will be used throughout this book, including the various VR hardware devices that UE4 supports and some of the libraries that are needed to create VR experiences (even if you won't be interfacing directly with them). You gained insight into some of the techniques used to optimize VR experiences. (If you want to learn more, check out [Chapter 10](#).) Finally, you looked at the current best practices from two of the major companies in VR.

Chapter 2. Head Mounted Display Setup

UE4 removes a lot of the pain of beginning to create your VR experiences, but as with a lot of things, it is good to have a basic understanding of how to set up your project to avoid some common pitfalls.

This chapter walks you through how to set up your project to work with your desired HMD as well as some basic VR functionality.

Gear VR

Because the Gear VR is on the Android platform, more steps are involved in getting your project set up and ready to deploy to your device. In addition, you must set up a few things in order for Oculus to accept the app into its store.

Epic's documentation offers the exact setup to make sure your engine can detect and launch to your phone, which is found here:

<https://docs.unrealengine.com/latest/INT/Platforms/GearVR/Prerequisites/index.html>

Make sure that you do the following:

1. Enable developer mode on your phone.
2. Enable USB debugging on your phone.
3. Install the Android development pack.
4. Create your Oculus Signature File (OSIG), which is packaged with your development builds and allows your app to run on your phone outside of the Oculus store.
5. Copy your OSIG into the Engine/Build/Android/Java/assets folder of your current engine.

Gear VR Project Setup

After your engine is set up to deploy to the phone, you need to set up your project to do the same. To do this, create a new blank Blueprint project: set the target device to Mobile/Tablet and scalability to Scalable 3D or 2D, and do not include starter content (see [Figure 2.1](#)). These settings ensure that some of the advanced graphical features of UE4 are off by default (to see the exact features, refer to [Chapter 10](#), “[VR Optimization](#)”).

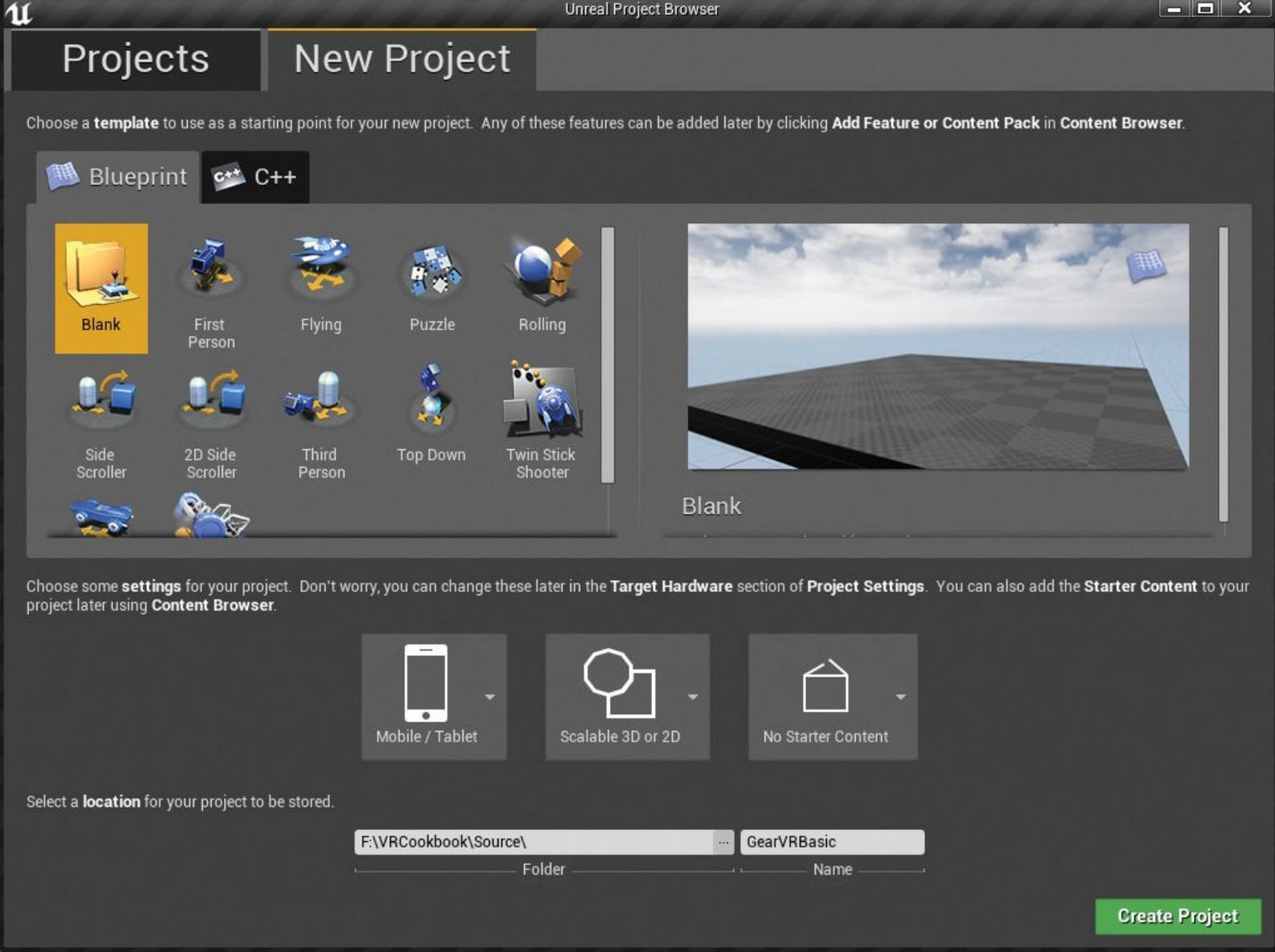


Figure 2.1 Gear VR initial project setup

After you create your project, you need to create the assets and folder structure required for a basic Gear VR game:

1. Create two folders under the root content folder of your project. Name them Blueprints and Materials.
2. Create one more folder named HUD (heads-up display) and place it under the Blueprints folder.
3. Create a new game mode: right-click the Blueprints folder, under Create Basic Asset select Blueprint Class, then in the dialog that pops up select Game Mode.
4. Name the new game mode GearVRGM.
5. Create a new HUD Blueprint by right-clicking the HUD folder and selecting Blueprint Class once more. This time in the dialog that pops up expand the All Classes drop-down and search for HUD. Select the result named solely HUD and then click the Select button (see [Figure 2.2](#)).

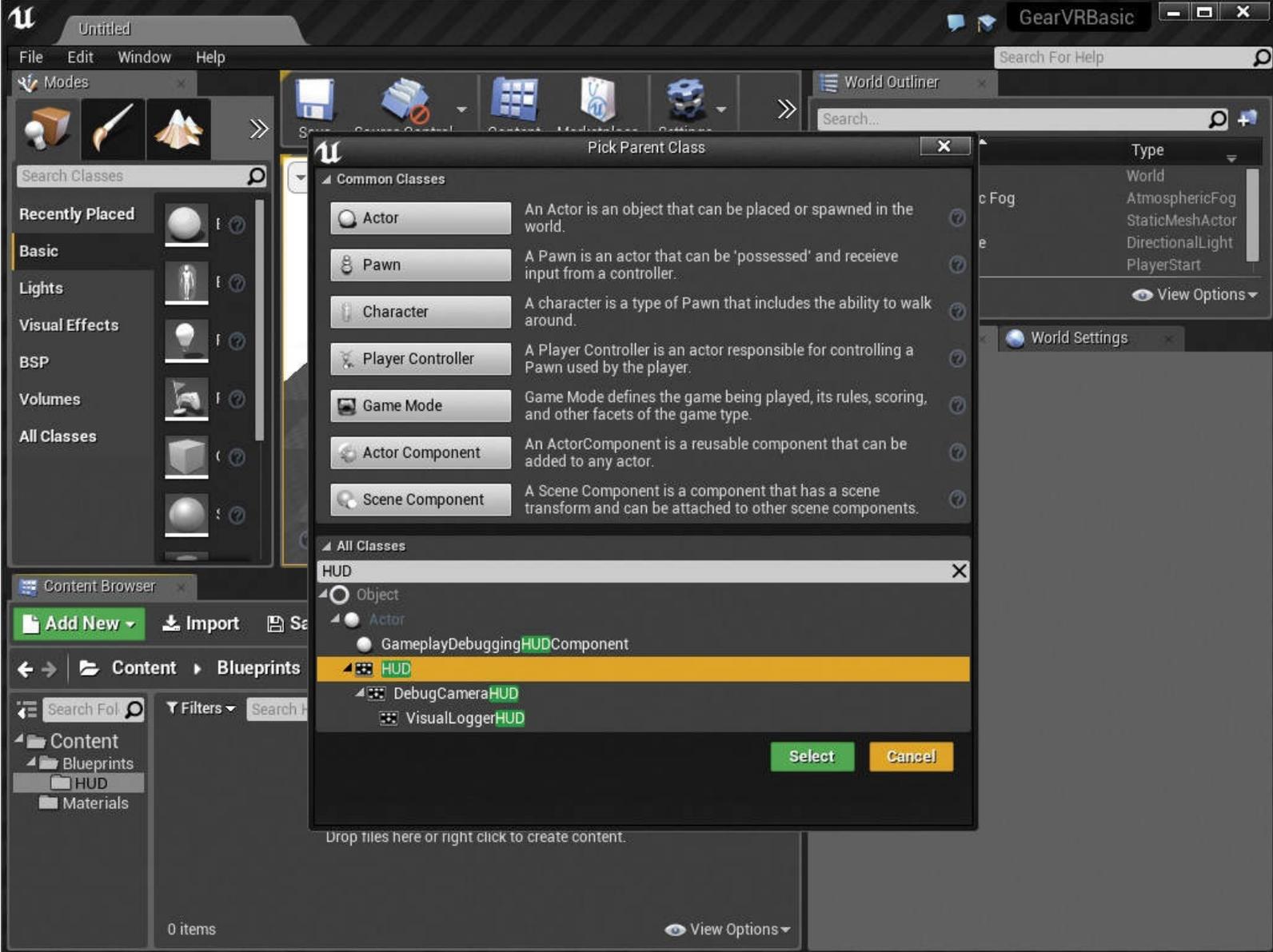


Figure 2.2 Gear VR HUD creation

6. Name this newly created HUD GlobalMenu. This will display your loading bar and detect when to open the Gear VR's global menu.
7. Right-click the Materials folder, create a new Material, and name it UICircle. This is the Material you will animate to create the menu progress circle. (Note that the Gear VR developer guidelines don't require the menu indicator to be circular; however, it looks good and is a good exercise to show the capabilities of UE4's Material Editor.)

Your project should now match [Figure 2.3](#).

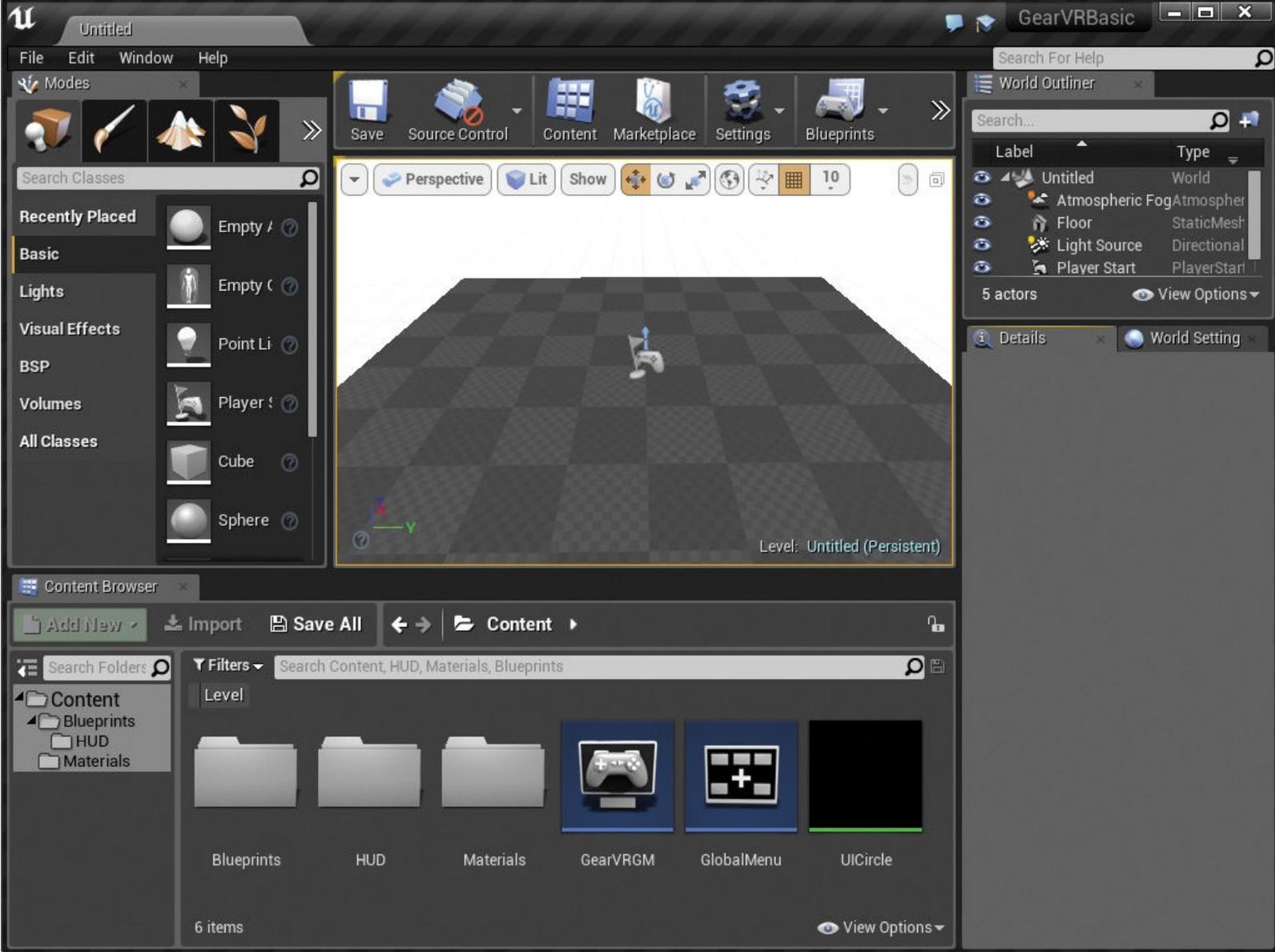


Figure 2.3 Gear VR project outline

Next, enter the Project Settings to make sure everything is set up properly to enable deploying to the Gear:

1. Open Project Settings, Edit → Project Settings, and head to the Android section under Platforms. This is where you'll configure your Android settings.
2. Click the Configure Now button on the banner that appears at the top of the window.
3. In the APKPackaging section, set the Minimum SDK and Target SDK Versions to 19.
4. In the same section, check "Package game data inside .apk?" This ensures that your .obb file (a file that contains large binary data such as textures) is packaged inside your .apk (Android Application Package) and is necessary because the Oculus store accepts only a single .apk file.
5. In the Advanced APKPackaging section check "Configure the AndroidManifest for deployment to GearVR"; this is needed to tell the phone to open your app as a Gear VR application. These settings are shown in [Figure 2.4](#).

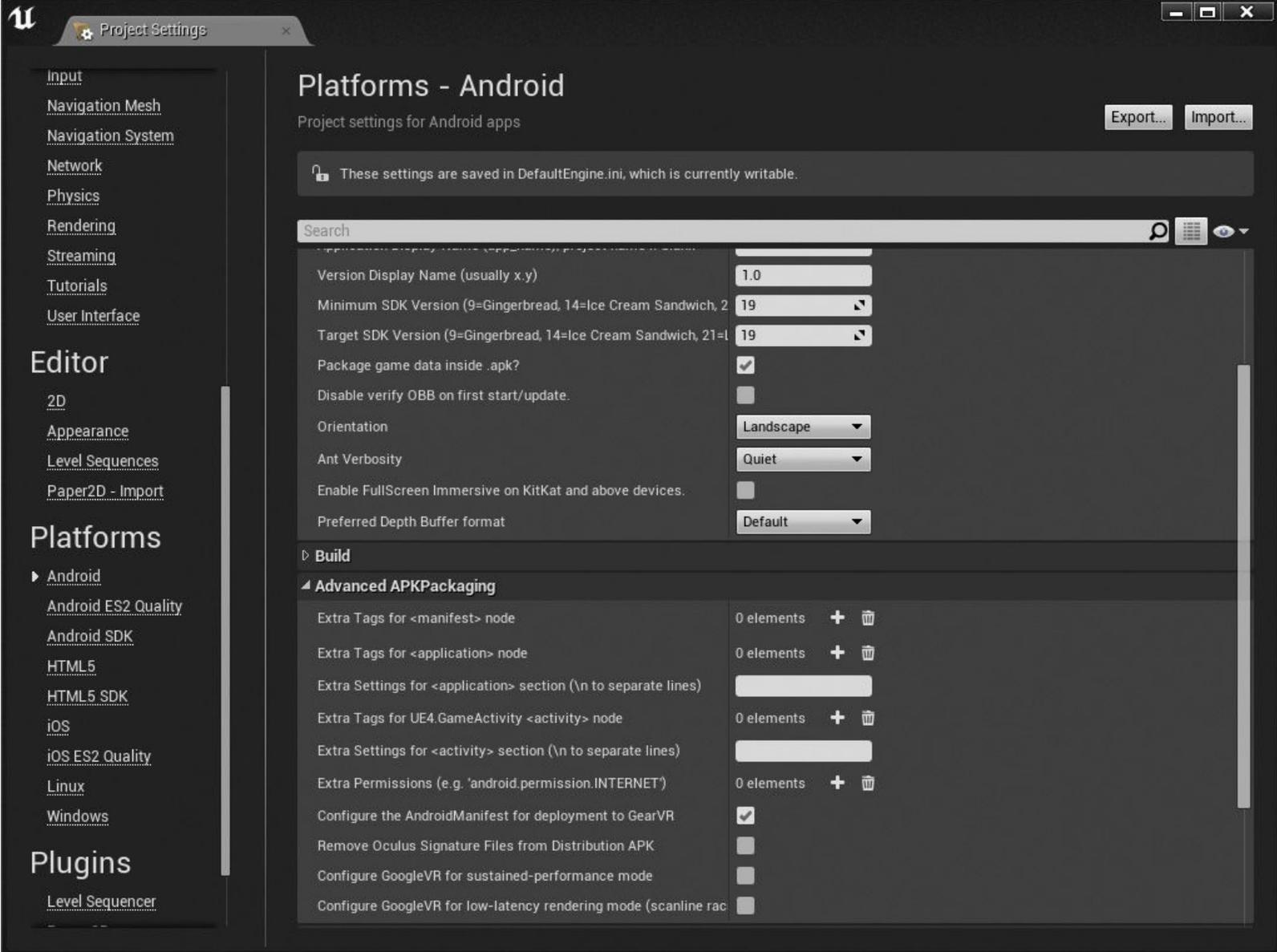


Figure 2.4 Gear VR: configuring the Android settings

6. To remove the default touch joysticks in UE4, head to the Input Settings under the Engine section in the Project Settings.
7. Click the drop-down for the Default Touch Interface and select the Clear option (see [Figure 2.5](#)).

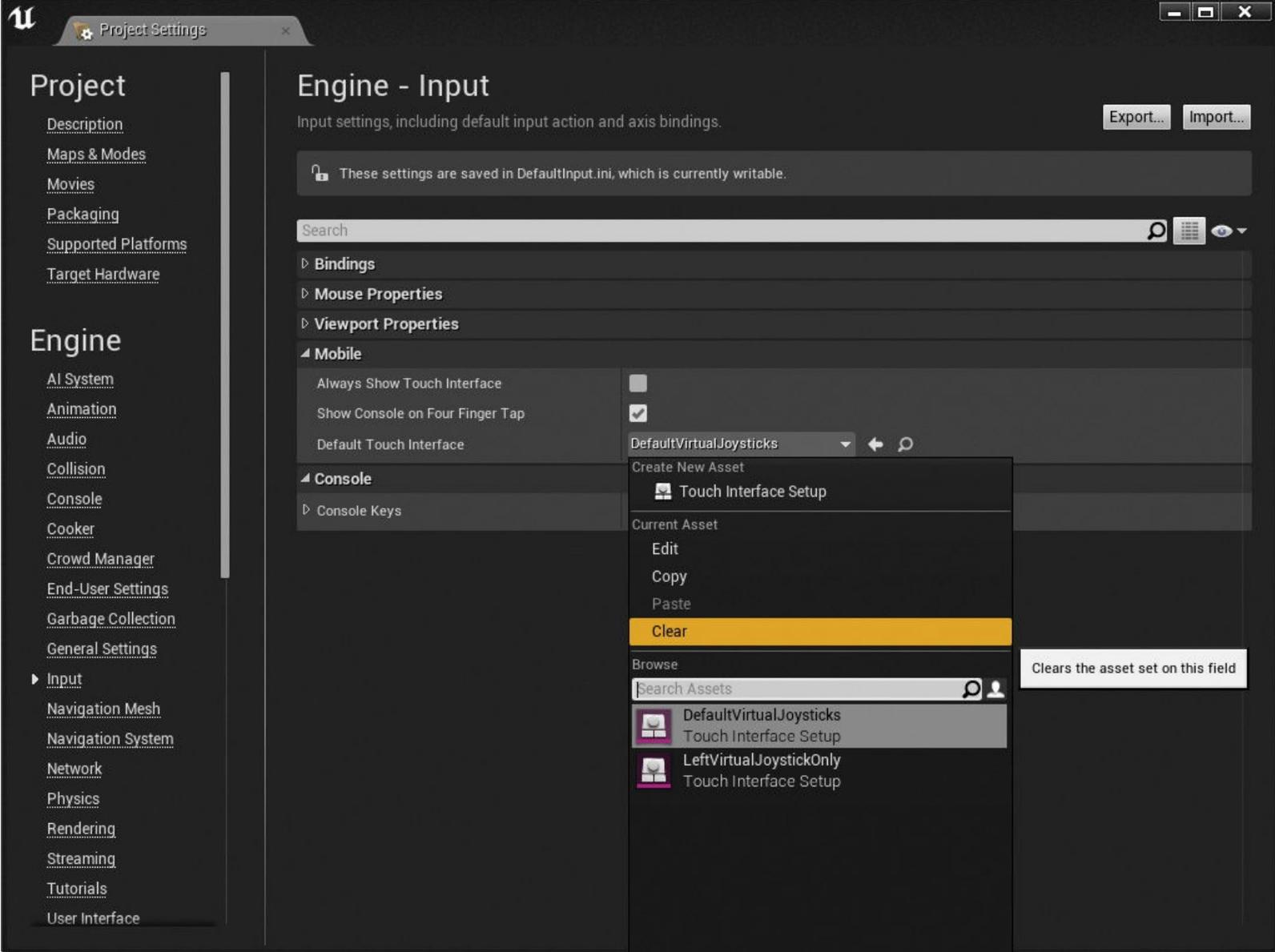


Figure 2.5 Gear VR: removing the default joysticks

8. To tell the project to use the custom game mode that you have created, still in the Project Settings head to the Maps & Modes section in the Project section.
9. Select the drop-down for the Default GameMode, and select GearVRGM which you set up previously (see [Figure 2.6](#)).

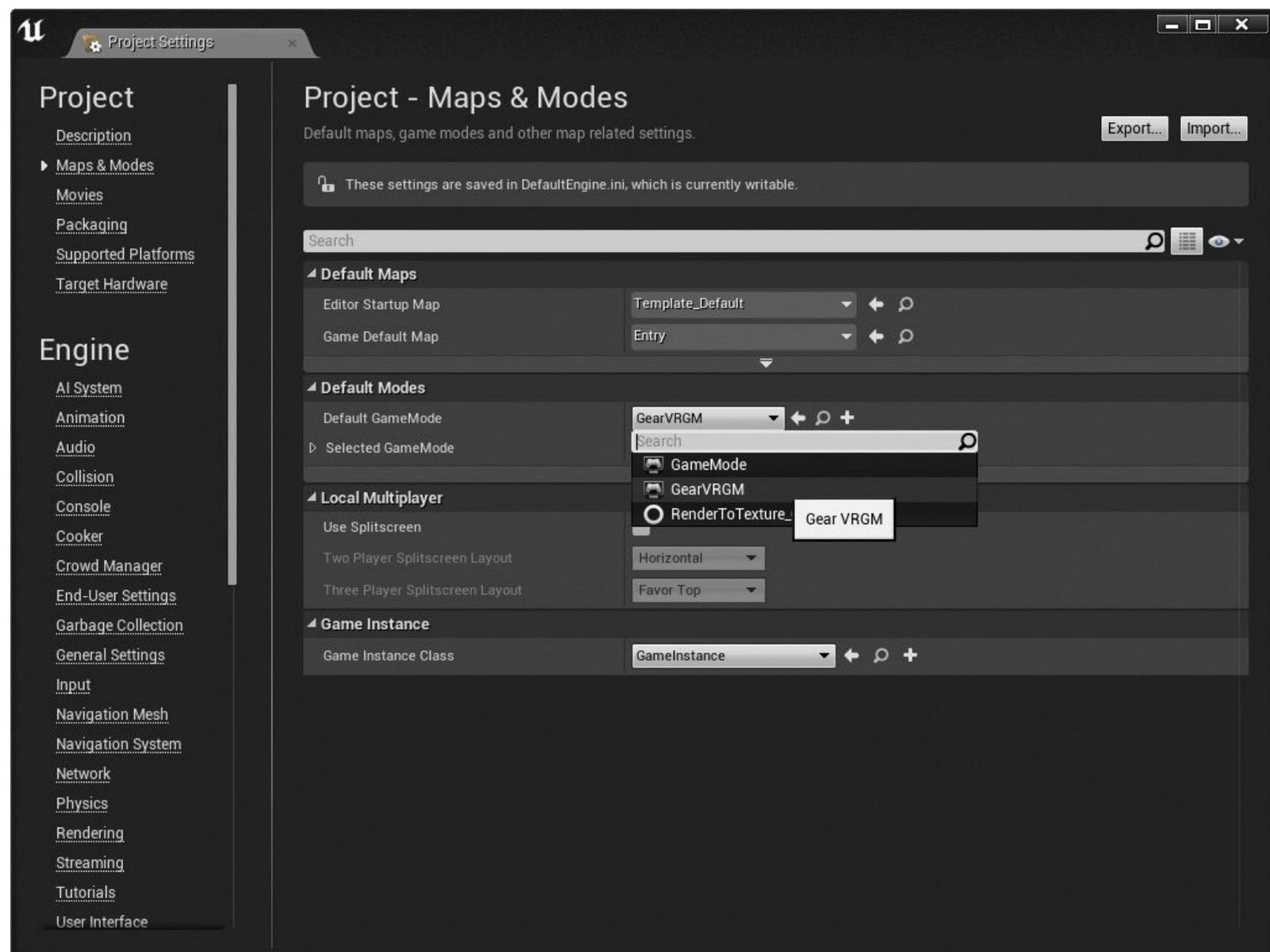


Figure 2.6 Gear VR: enabling the new default game mode

Gear VR Global Menu Setup

To get a Gear VR app accepted into the Oculus store, you need to give the app the functionality to open the Gear VR menu on a long press of the home button. By default a short click of the Back button asks users if they wish to exit the app, and a long press opens the global menu. However, if you want to create a transition to the global menu or create your own short-press functionality, you need override this functionality and create a custom event that opens the global menu on a long press of the home button.

Note

Here, you use a HUD Blueprint; they are convenient for this kind of user interface. However, in the initial version of Unreal Engine 4.13, Epic broke compatibility with HUD Blueprints and VR. If you are using this version of the engine, you can substitute a Widget Component inside a Pawn Class for the HUD Blueprint (see [Chapter 6](#), “[Unreal Motion Graphics and 2D User Interfaces](#)”).

1. Open your GearVRGM game mode Blueprint and set its HUD Class to be your GlobalMenu (see [Figure 2.7](#)). This tells the engine to use your custom HUD.

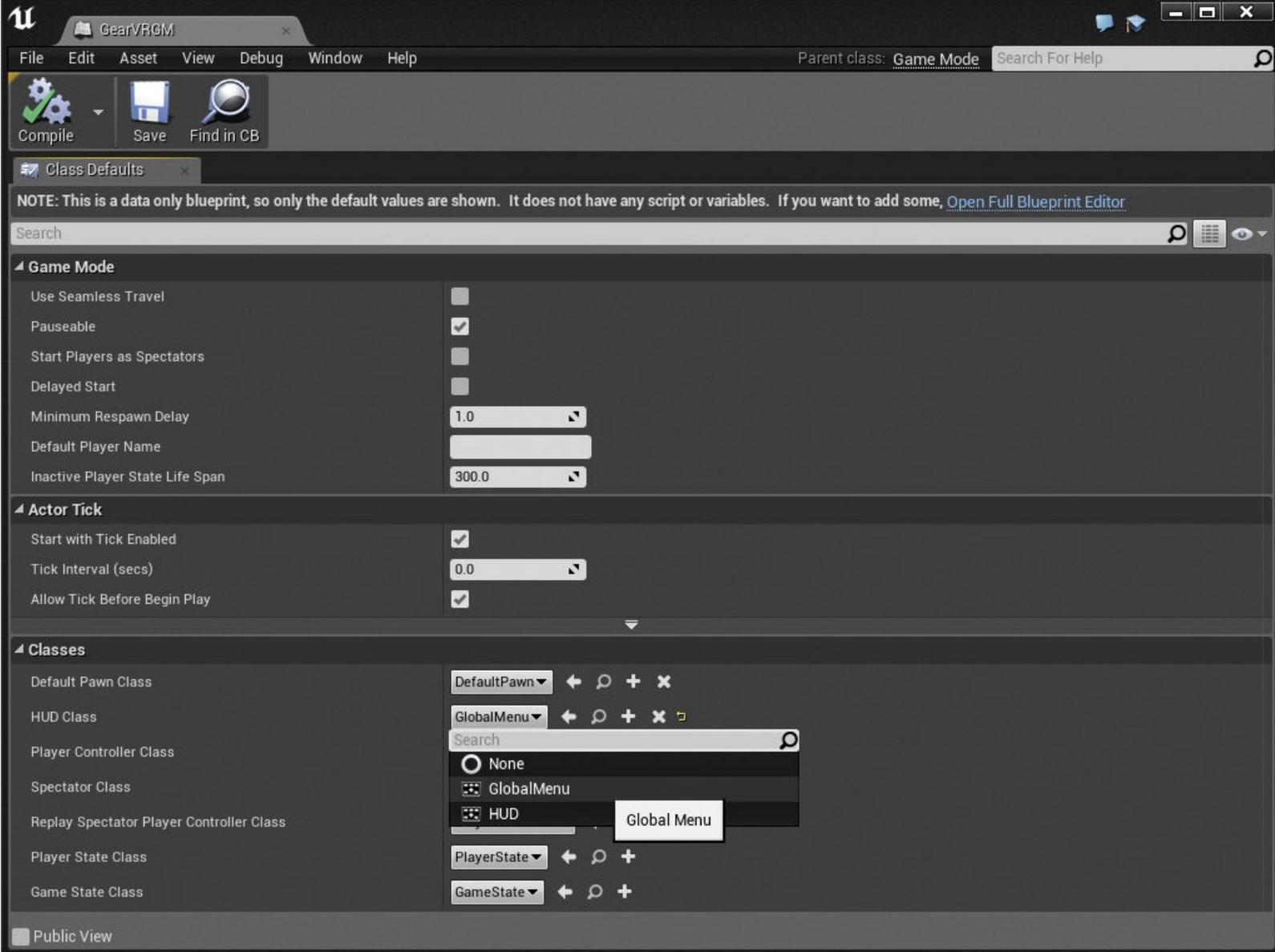


Figure 2.7 Gear VR: configuring the game mode to use the custom HUD

2. Open your GlobalMenu Blueprint and create a new function named OpenMenu. Inside the function, create a new ExecuteConsoleCommand node and hook it up to the function execution pin.
3. For the Command input pin of the ExecuteConsoleCommand, type OVRGLOBALMENU; this causes the Gear VR menu to open whenever this function is called (see [Figure 2.8](#)).

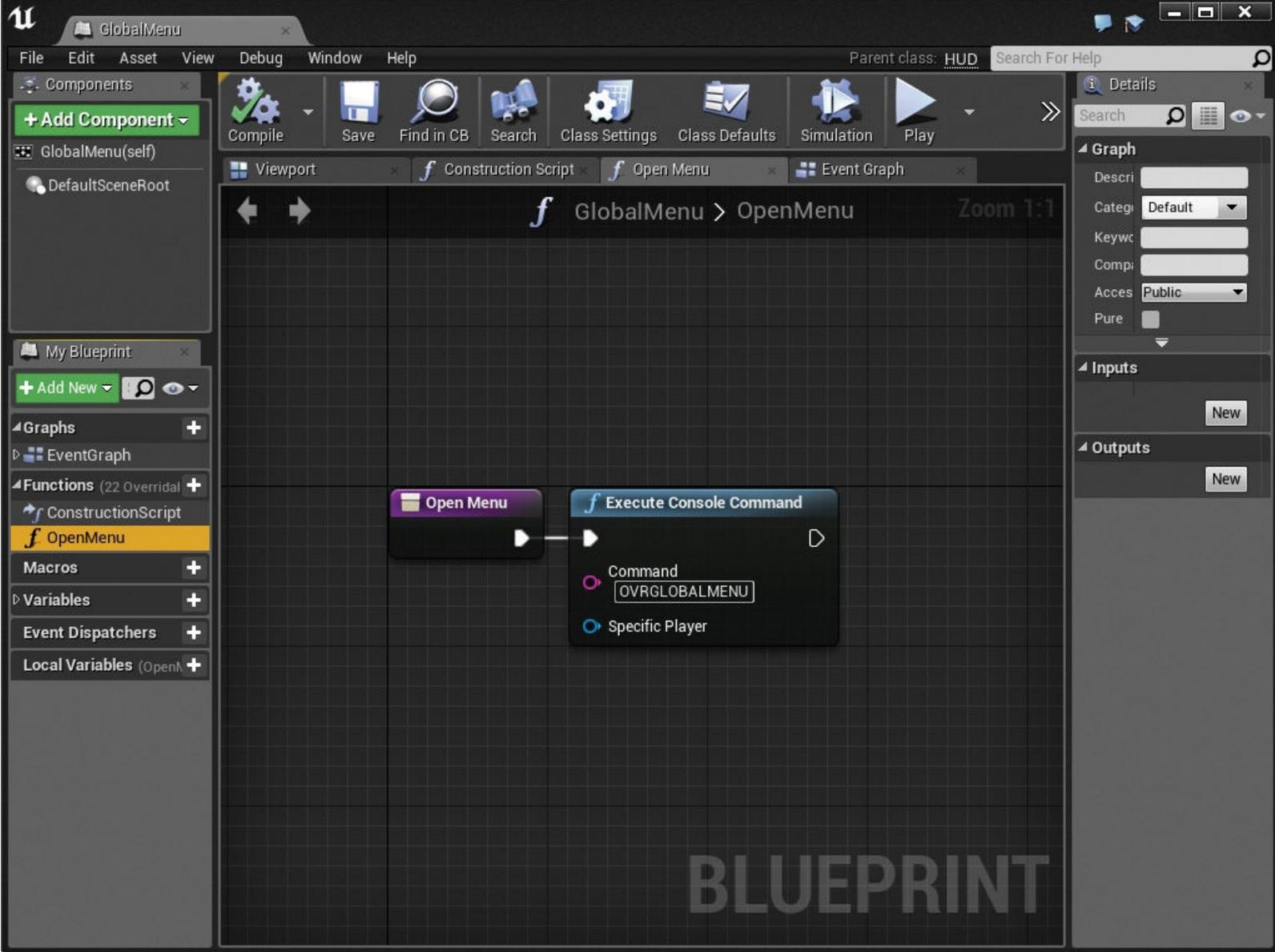


Figure 2.8 Gear VR: creating the OpenMenu function for the HUD

4. Create three new variables.
 - Make the first of type Timer Handle and name it BackButtonTimer.
 - Make the second of type Material Instance Dynamic and name it CircleMat.
 - Last, create a new Float variable named CircleRadius, setting its default value to 30.
5. Create a new AndroidBack input event by right-clicking the Event Graph and searching for AndroidBack. (Note that you may have to turn off context-sensitive searching.)
6. Drag off of the Pressed execution pin and create a new SetTimerByFunctionName node.
7. Set this node's Function Name input node to OpenMenu. This makes the timer call the function you set up earlier.
8. Set the Time input node to 0.75; this is the time required by Oculus in its developer guidelines.
9. Drag off of the Return Value pin of the SetTimerByFunctionName node and create a new setter node for the BackButtonTimer variable.
10. Drag in a new getter node for the BackButtonTimer variable, call the ClearTimerByHandle function, and connect its input execution pin to the Released execution pin of the AndroidBack event (see [Figure 2.9](#)). This creates a new timer that will count up to 0.75 when you press the AndroidBack button but cancels that timer when you release the button.

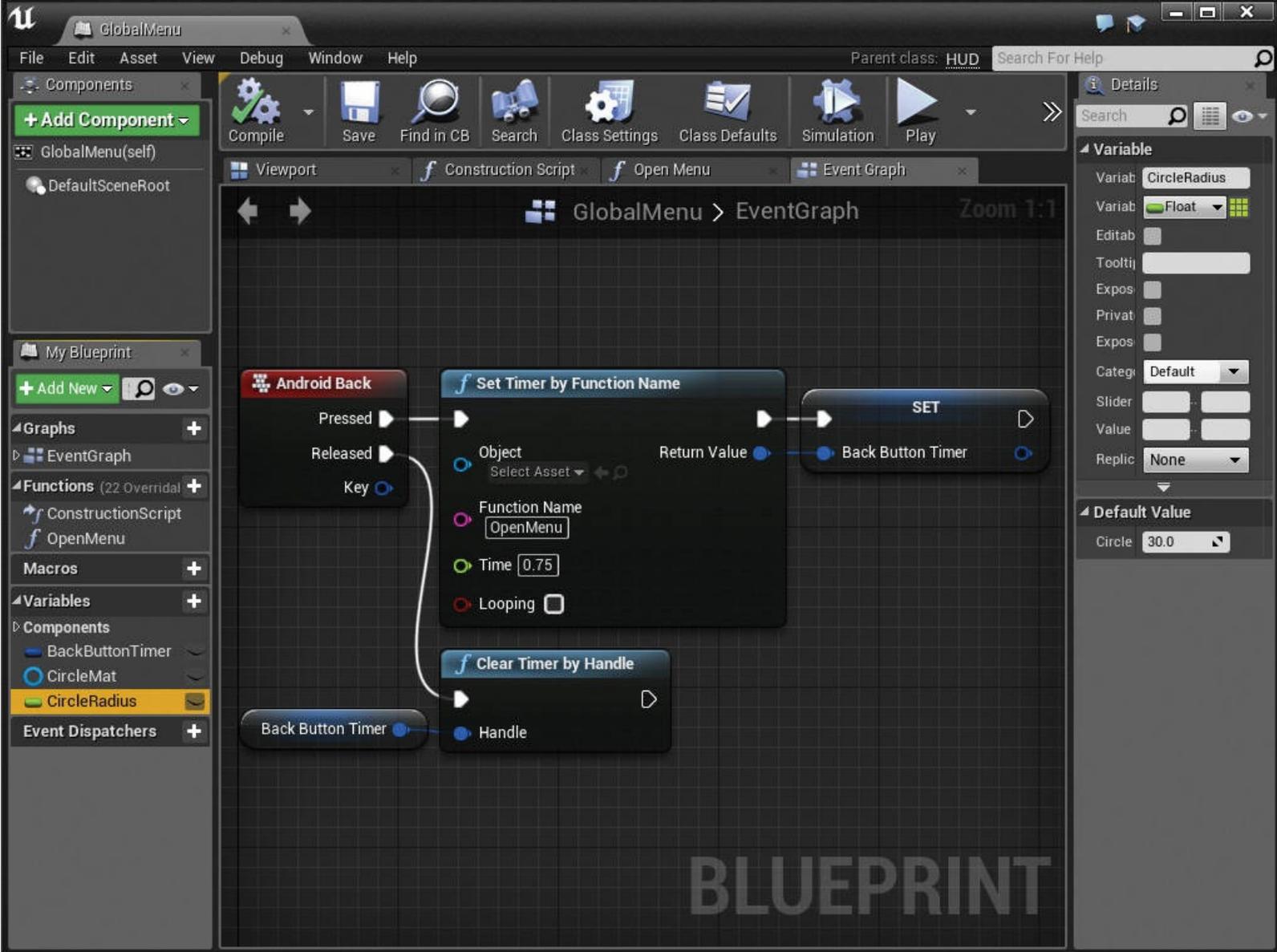


Figure 2.9 Gear VR: creating the timer to open the menu after a long press of the Back button

11. Create a new `GetOwningPlayerController` pure function call by right-clicking the Event Graph and selecting `GetOwningPlayerController`.
12. Drag off of the Return Value of this new node and call the `EnableInput` event, making sure that the `GetOwningPlayerController` node connects to the `EnableInput` event's Player Controller input pin.
13. Connect the output execution pin of the `EventBeginPlay` event to the `EnableInput` node. This ensures that the `AndroidBack` event will be able to capture the input event and fire when needed.
14. After the `EnableInput` event add a new call to the `ExecuteConsoleCommand` event, passing in `gearvr.handlebackbutton 0` as the command parameter. This ensures that the Gear VR runtime will allow you to program your own Back button functionality.
15. For the `Specific Player` input pin connect the `GetOwningPlayerController` node.
16. After the `ExecuteConsoleCommand` event add a new call to the `CreateDynamicMaterialInstance` function by clicking on the Parent pin's drop-down and selecting the `UICircle` Material that you created previously. This creates a dynamic Material that you can animate for the loading circle.
17. Drag off of the Return Value of the `CreateDynamicMaterialInstance` node and create a setter for the `CircleMat` variable (see [Figure 2.10](#)).

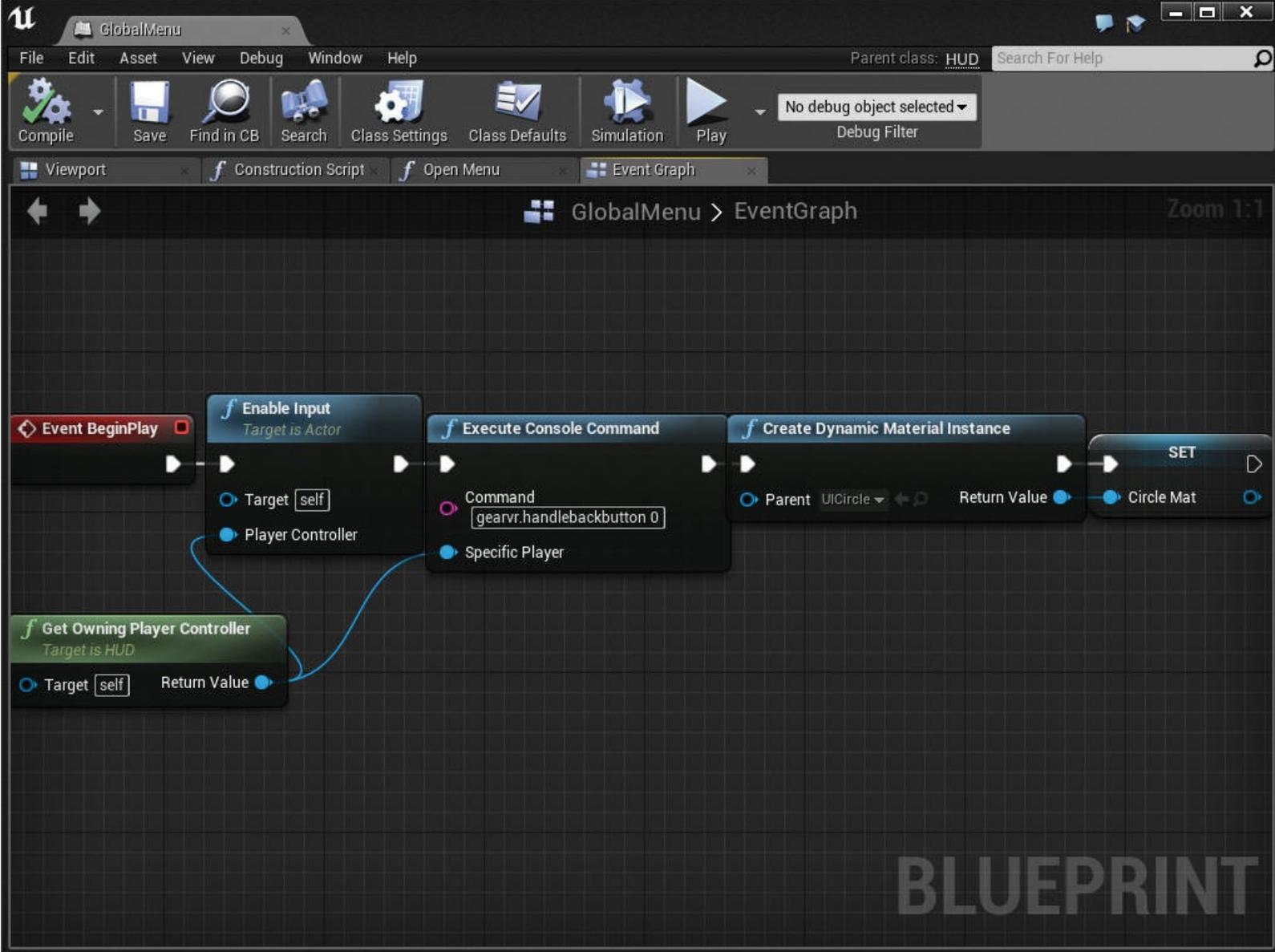


Figure 2.10 Gear VR: enabling input and setting up the dynamic Material for the HUD

The next task to take care of is animating and drawing the Material to the screen when the Back button is pressed:

1. Right-click the Event Graph and create a new EventReceiveDrawHUD node.
2. Drag this event's execution pin and create a new Branch node.
3. Drag in a new getter for the BackButtonTimer variable, call the IsTimerActiveByHandle pure function, and attach the Return Value to the Condition input pin of the Branch node. This causes the UI to draw only when the Back button is pressed.
4. Drag in another getter (this time for the CircleMat variable) and call the SetScalarParameterValue event on it.
5. In the Parameter Name input pin, type PercentComplete. This matches up with a scalar parameter in the UICircle Material that will be created soon.
6. Attach the True output pin of the Branch node to the input execution pin of the SetScalarParameterValue node.
7. Create another new getter for the BackButtonTimer variable and call the GetTimerElapsedTimeByHandle function.
8. Drag off of this function's Return Value and create a new Float / Float node. Enter 0.75 in the

- second pin. This normalizes the time elapsed value between 0 and 1.
9. Attach the output of this division to the Value input pin of the SetScalarParameterValue node.
 10. Drag in a new getter for the CircleMat variable and call the DrawMaterial function, connecting the output of the SetScalarParameterValue node to the input execution pin.
 11. Drag off of both the Size X and Size Y output pins and call the Int * Float function, passing 0.5 into the second pin. You will use this to calculate the middle of the screen.
 12. For both of these Multiply nodes, drag off of the output pin and create a new Float – Float node, passing in a new variable getter for the CircleRadius function to the second pin. This corrects for the size of the UI element and ensures that it is actually in the middle of the screen.
 13. Connect the calculation for the Size X into the Screen X input pin of the DrawMaterial event and the Size Y into the Screen Y.
 14. Drag in a new getter for the CircleRadius variable and call the Float * Float function, passing in 2 as the second pin, then pass this into both the Screen W and Screen H inputs of the DrawMaterial event.
 15. Set the Material UWidth and Material VHeight input pins of the DrawMaterial event to 1.

This event should now look like [Figure 2.11](#).

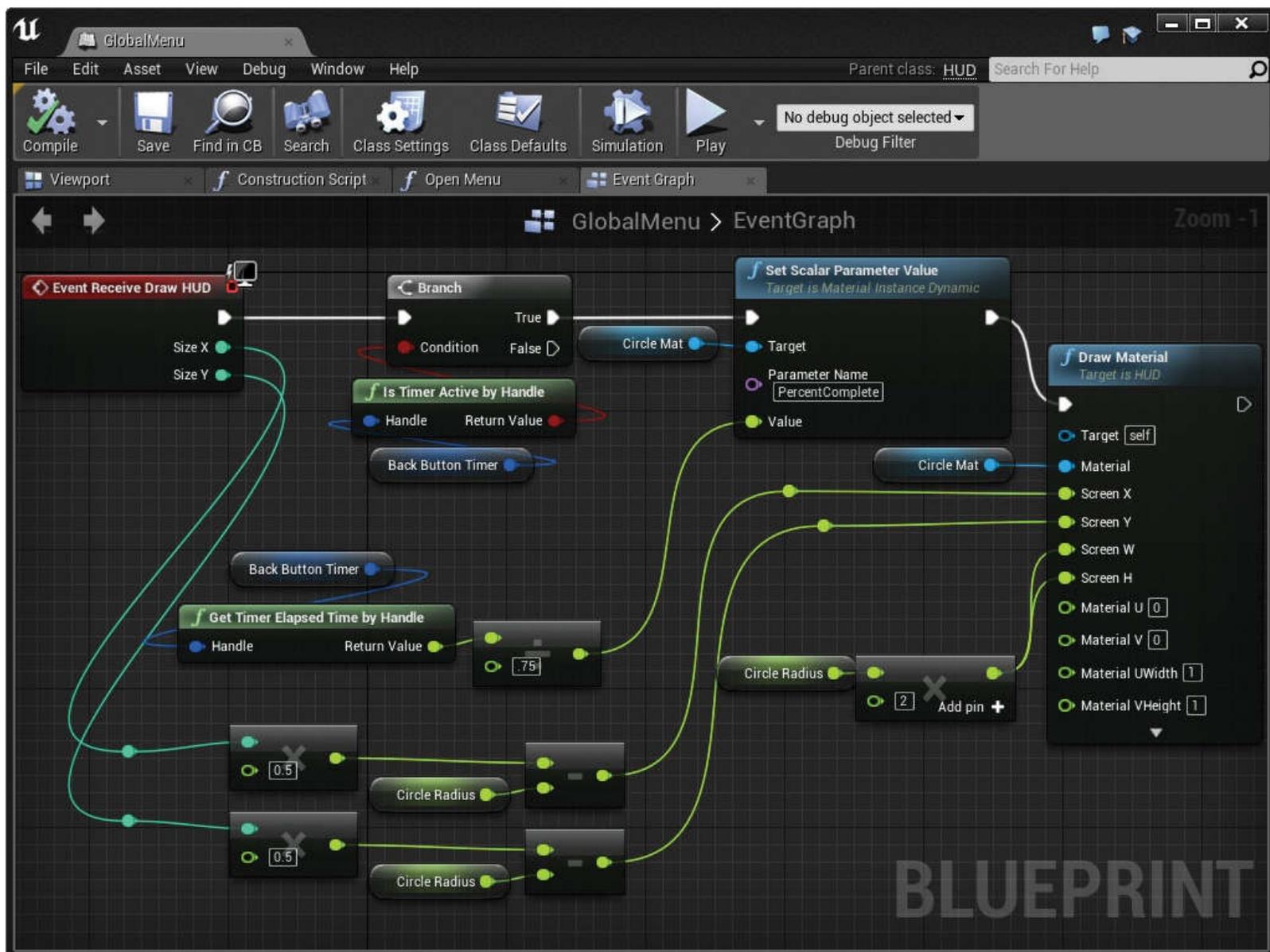


Figure 2.11 Gear VR: animating and drawing the HUD Material

Gear VR Global Menu Progress Material

When a user long-presses the Back button to open the Gear VR global menu, ideally there should be some visual feedback that lets the user know the state of the action.

To accomplish this, we will create a circular spinner that is similar to the one seen in Oculus's own experiences:

1. Open the UICircle Material from the Materials folder you created previously.
2. In the Details panel, set the Material Domain to User Interface; this limits the output Material pins severely, but you still have all you will need for this basic Material.
3. Still in the Details panel, set the Blend Mode to Translucent because you are going to have two rings: a full-opacity progress ring and a background transparent ring.
4. Create a new TextureCoordinate node by either right-clicking and searching for TextureCoordinate or U + clicking the graph.
5. Create a new VectorToRadialValue node; this will help you create the circle.
6. If you pass the output of the TexCoord into the Vector input of the VectorToRadialValue node and pass the Linear Distance output to the Final Color of your Material, you'll notice that it creates a circular gradient emanating from the top left corner. To translate this into the center of your Material, you need to scale and move the input you give the VectorToRadialValue node.
 - To do this, add a new ConstantBiasScale node to the graph, giving it a bias of -0.5 and a scale of 2.0 in the Details panel.
 - To actually see the effects of this node, connect the ConstantBiasScale between TexCoord and VectorToRadialValue.
 - You should now see a circle gradient in the center of the Material.
7. Disconnect the Linear Distance pin from the Final Color and connect the Vector Converted to Angle pin instead. You should now have something similar to [Figure 2.12](#).

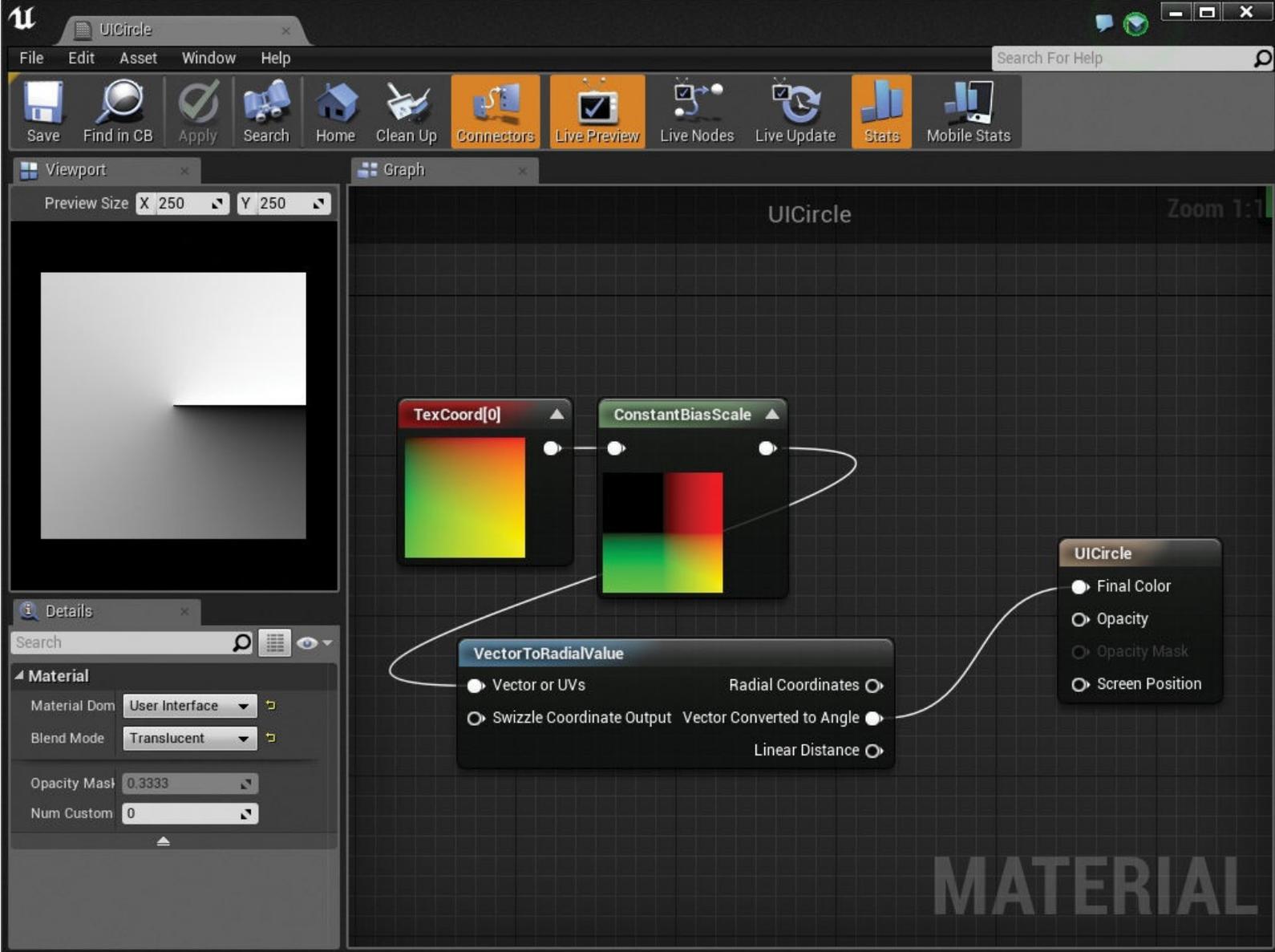


Figure 2.12 Gear VR HUD Material: basic gradient

Ideally, you can see how this new gradient may help you create a spinning circle.

The one problem with the current gradient is that it faces to the right when you would actually want it to face upward so that when animated, it moves more like an analog clock.

8. To rotate the Material, add a Swizzle node and connect it between ConstantBiasScale and VectorToRadialValue, making sure to choose the XY and YX paths.
9. Now that your Material is rotated, you will notice that it is upside down. To fix this, create a OneMinus node and place it between the TexCoord and ConstantBiasScale nodes. This flips the TexCoord and makes the gradient face the proper way.
10. To control the amount that the circle has completed, create a new scalar parameter node and name it PercentComplete in the Details panel.
11. Drag off of the PercentComplete node and create a new Add node, connecting the Vector Converted to Angle output of the VectorToRadialValue node to the second input.
12. Drag off of the Add node and create a new Floor node; this allows you to convert anything below 1 to black and anything at or above it to white. You will use this as one of your masks later on.
13. Now connect the output of the Floor to Final Color; you should have something resembling [Figure 2.13](#). (Change the PercentComplete value and notice how the output changes.)

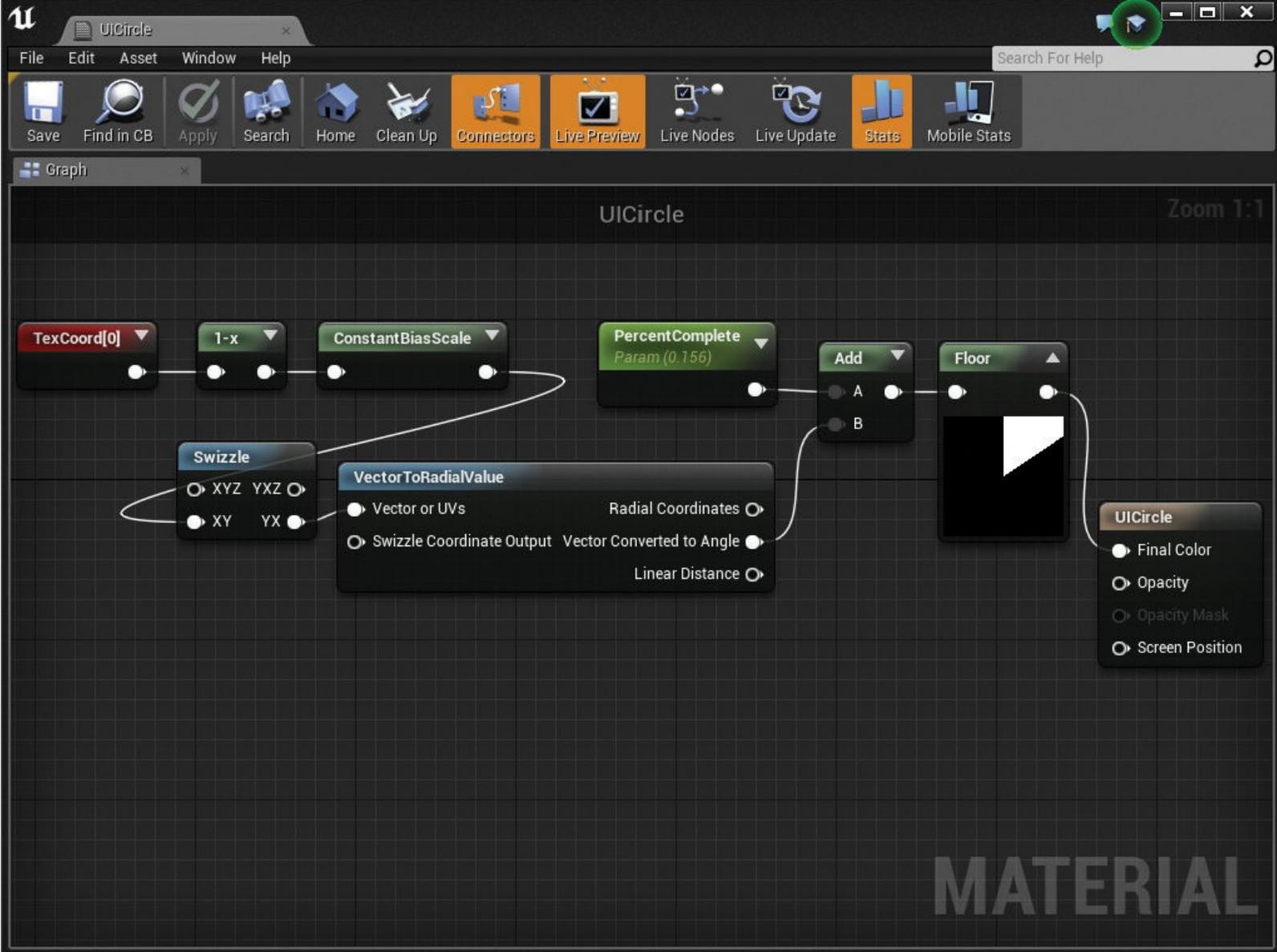


Figure 2.13 Gear VR HUD Material: animated rotating mask

Now that you have the first mask, it is time to actually create the circle. For this you need two circles, first an outer circle and then an inner circle that you will cut from the outer one to create a nice ring:

1. Drag off of the Linear Distance of the VectorToRadialValue node and create a new OneMinus node to invert the gradient.
2. Creating the inner circle first, drag off of the output of the OneMinus node and add a new Subtract node.
 - Create another scalar parameter, this time naming it Width and setting its default value to 0.4. This will be used to control the width of the ring.
 - Connect this Width parameter to the second input of the Subtract node.
3. After the Subtract node create a new Ceil node, which will round any value above zero up to 1.
4. Add a new Clamp node to ensure that the Ceil stays between 0 and 1.
5. To create the outer circle, simply drag off of the OneMinus node once more and add another new Ceil node to turn the gradient into black and white.
6. All that's left to do to create the ring is to grab the output of the Ceil you used to create the outer ring and subtract the output of the Clamp for the inner ring.
7. To finish off the ring mask, grab the output of the Floor from the PercentComplete mask and

multiply it by the output of the Subtract for the ring mask. The result is that as you change the PercentComplete value, the fill of the ring is animated.

8. Connect the output of this Multiply into the Opacity for the Material and disconnect anything that is connected to the Final Color pin. You should now have something that resembles [Figure 2.14](#).

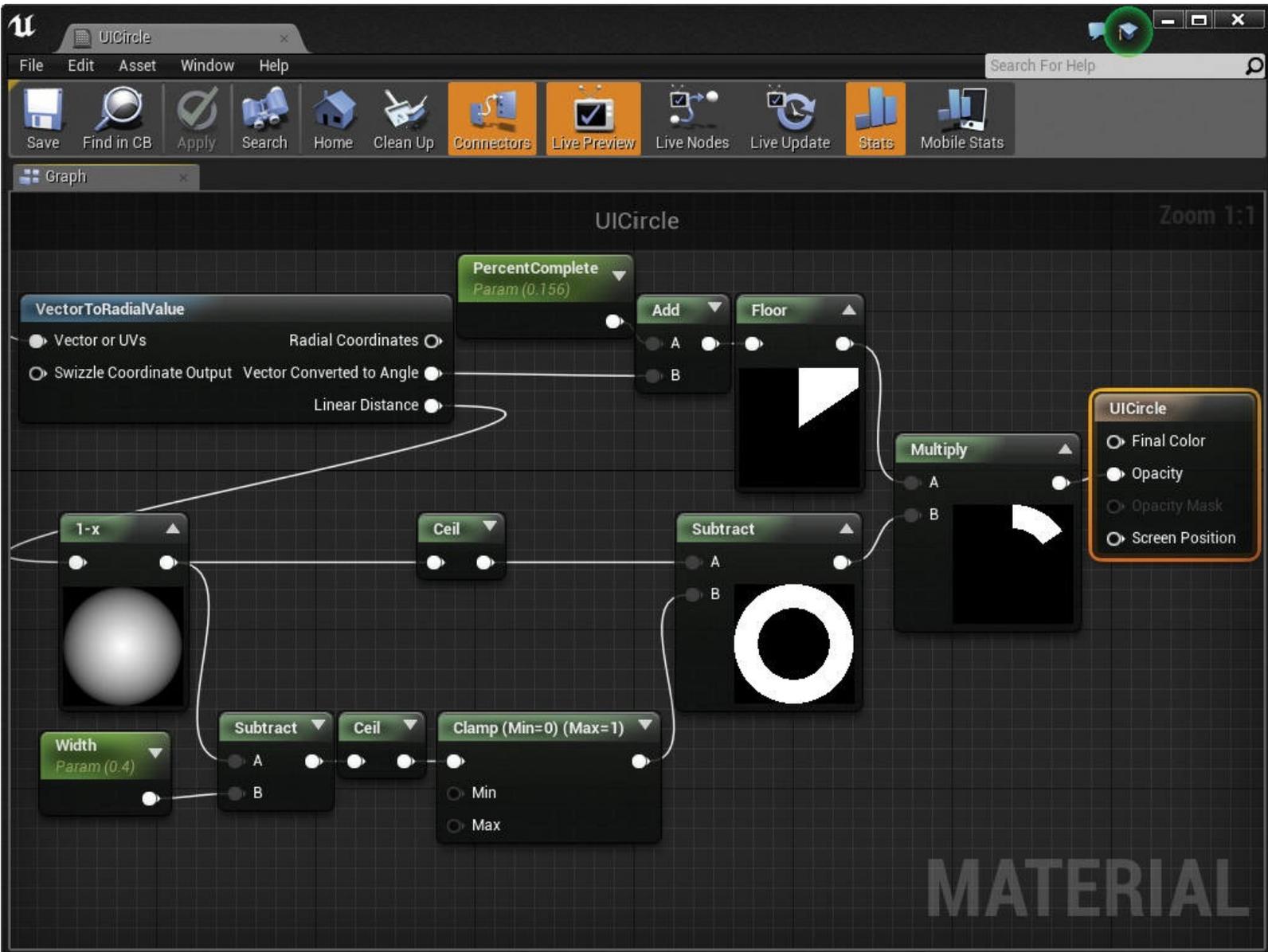


Figure 2.14 Gear VR HUD Material: animated mask and ring

9. To control the color of the ring, add a new vector parameter node to the graph and name it Color.
10. Set the default color of the vector parameter to whatever color you would like your ring to be (I opted for a nice blue of R = 0.266, G = 0.485, B = 0.896) and connect the white output pin of this node to the Final Color Material pin.
11. If you wish, you can stop here because the Material will be fully functional; however, it would also be nice if the background of the ring is always illuminated but at a more transparent level than the foreground.
 - To do this, create another new scalar parameter node named MinTransparency. Set its default value to 0.1.
 - Connected to this, add a new Add node, connecting the Floor of the PercentComplete path to the second pin.
12. Drag off of the new Add node and create another Clamp node to make sure that you are still

working with values between 0 and 1. Finally, connect the output of this node to the input of the Multiply that you used to combine the two masks. You should now have a Material that resembles [Figure 2.15](#).

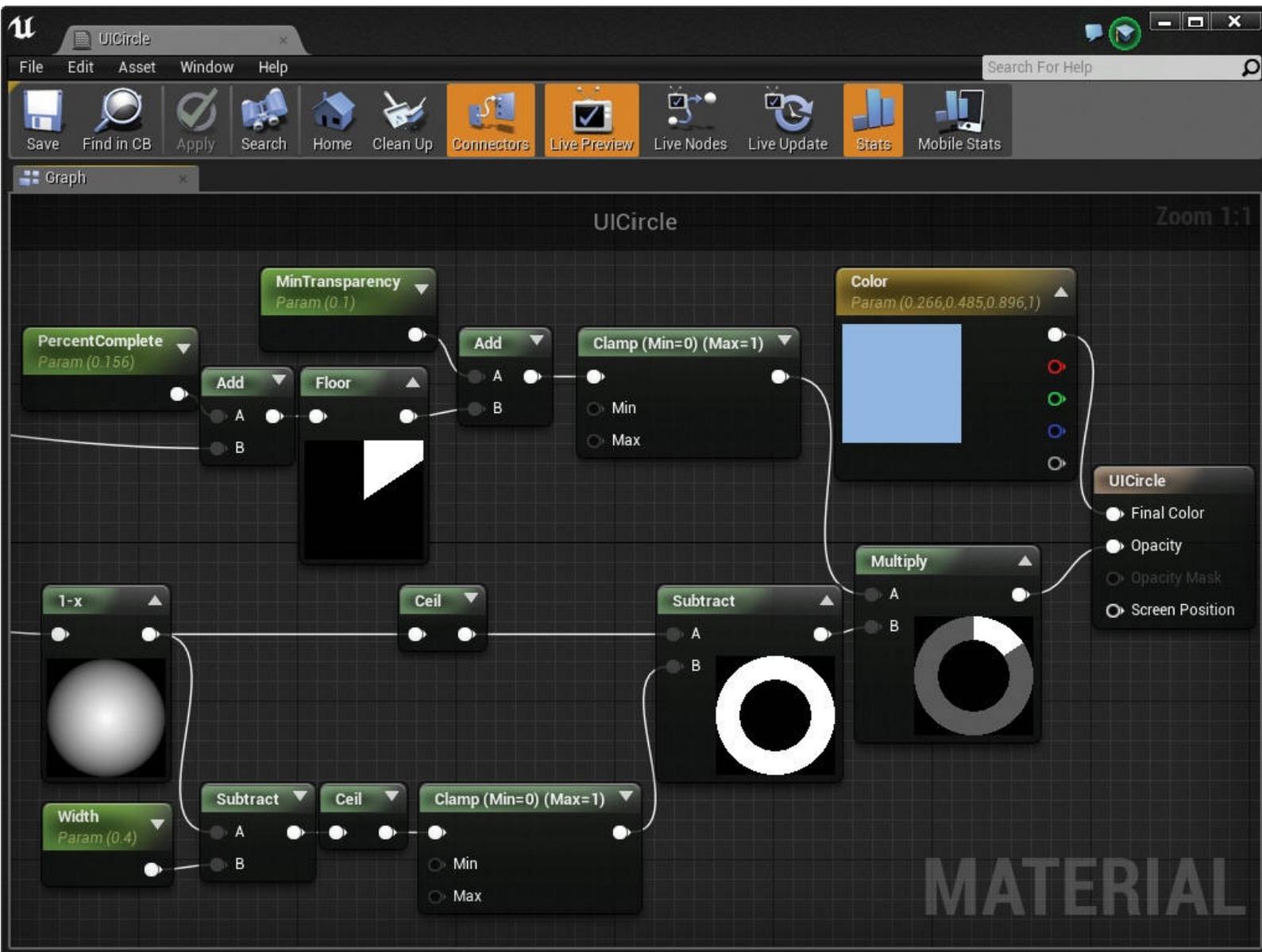


Figure 2.15 Gear VR HUD Material: animated mask with transparency

Now, if you connect your phone to your computer and click Launch, you should be able to hold the Back button on your Gear and see an animated circle that eventually takes you to the Gear VR Global Menu (see [Figure 2.16](#)).

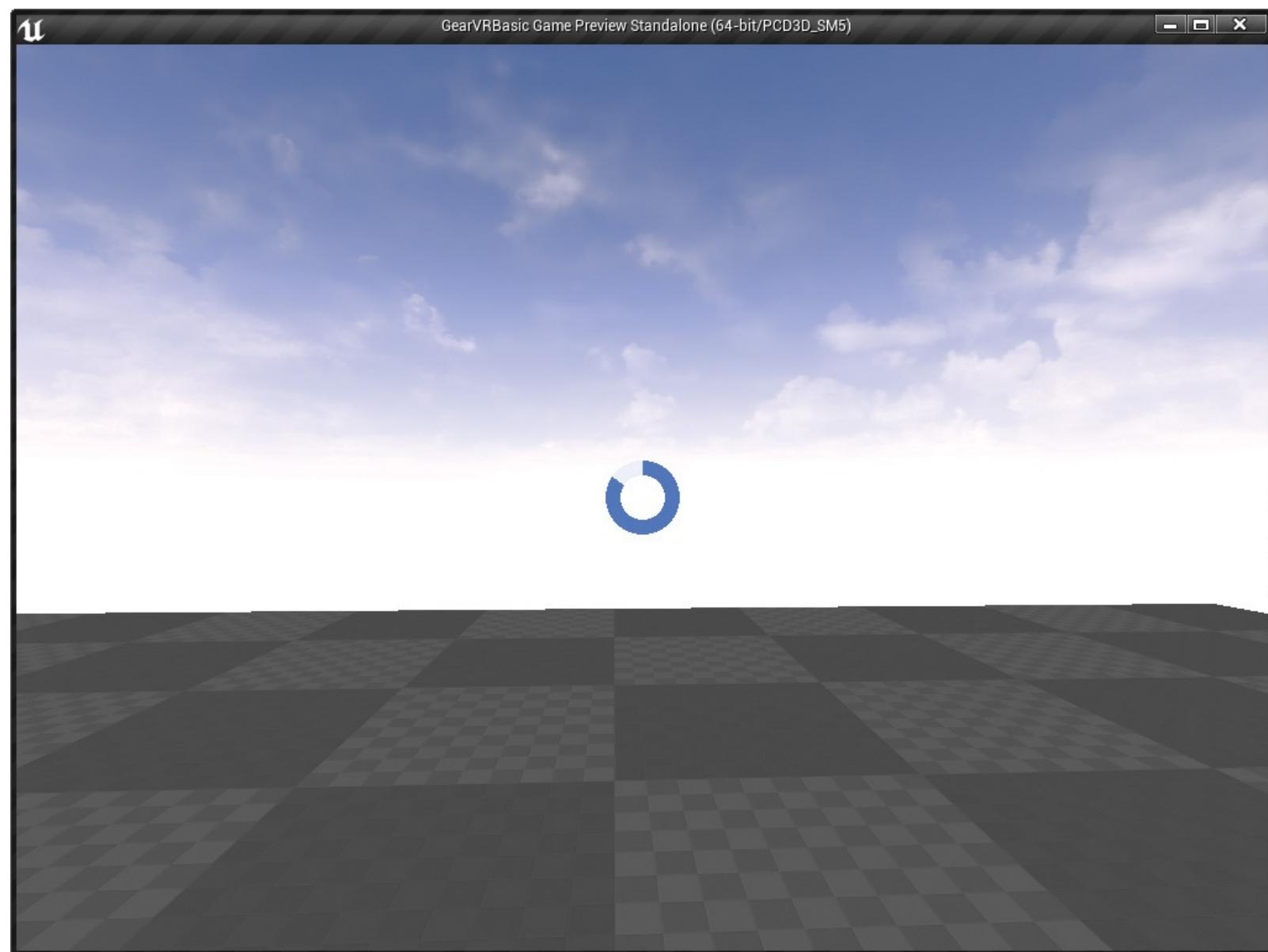


Figure 2.16 Gear VR HUD Material in-game

Rift and Vive

Unlike the Gear VR, both the Rift and Vive are easy to set up and get going. The main reason for this is that you will be developing on the system to which you eventually deploy (PC to PC), which eliminates the extra step of launching on a device, which can be a time-consuming process.

Similarly to the Gear VR, Epic has some nice documentation to take you through the setup of the Rift, which can be found here:

<https://docs.unrealengine.com/latest/INT/Platforms/Oculus/QuickStart/index.html>

Setup information for the Vive can be found here:

<https://docs.unrealengine.com/latest/INT/Platforms/SteamVR/QuickStart/index.html>

To develop for the Rift, all you need to do is

1. Install the Oculus runtime
2. Click the VR Preview button in the editor

To develop for the Vive, just

1. Install SteamVR and run Room Setup
2. Click the VR Preview button in the editor

Rift and Vive Project Setup

To set up a basic Rift or Vive project, the only two pieces you need are a Player Pawn and a game mode. (Technically, you could get away with only a Pawn.)

To do this, create a new blank Blueprint project, setting the target device to Mobile/Tablet and the scalability to Scalable 3D or 2D, and do not include any starter content (refer to [Figure 2.1](#)). These settings ensure that some of the advanced graphical features of UE4 will be off by default. (Again, if you want to see exactly which features, refer to [Chapter 10](#).)

It's a good idea to set up a basic folder structure to keep your project organized from the start. Once that is done, we will add the basic Blueprints you need to create a Rift/Vive experience:

1. Create a new folder under the root content folder and name it Blueprints.
2. Create two new Blueprints, one a game mode named VRGameMode and another a Pawn named VRPawn. You should now have a project similar to [Figure 2.17](#).

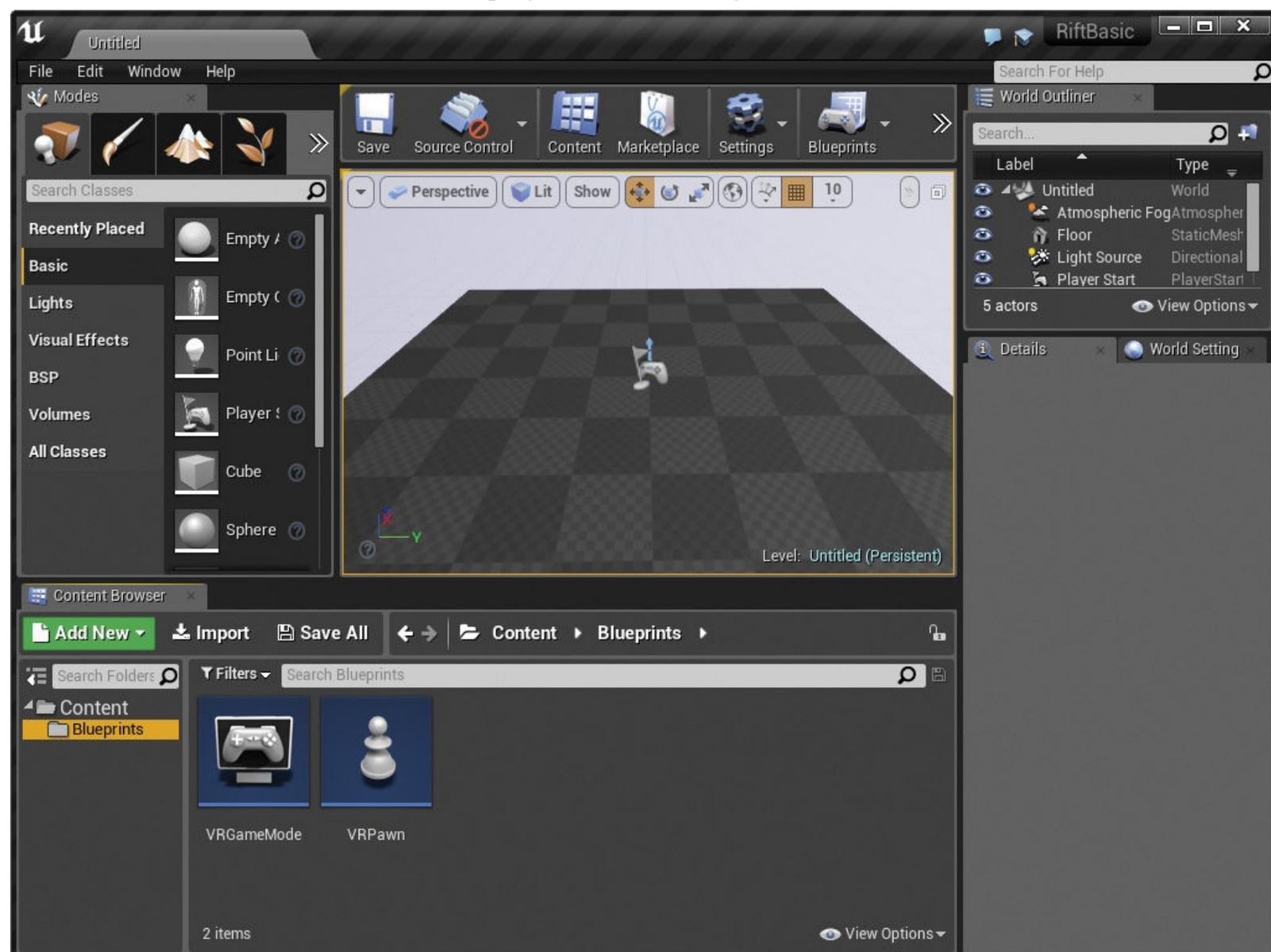


Figure 2.17 Basic desktop VR setup

3. Head to the Project Settings (Edit → Project Settings), select the Maps & Modes section, and change the Default GameMode to the VRGameMode you just created (see [Figure 2.18](#)).

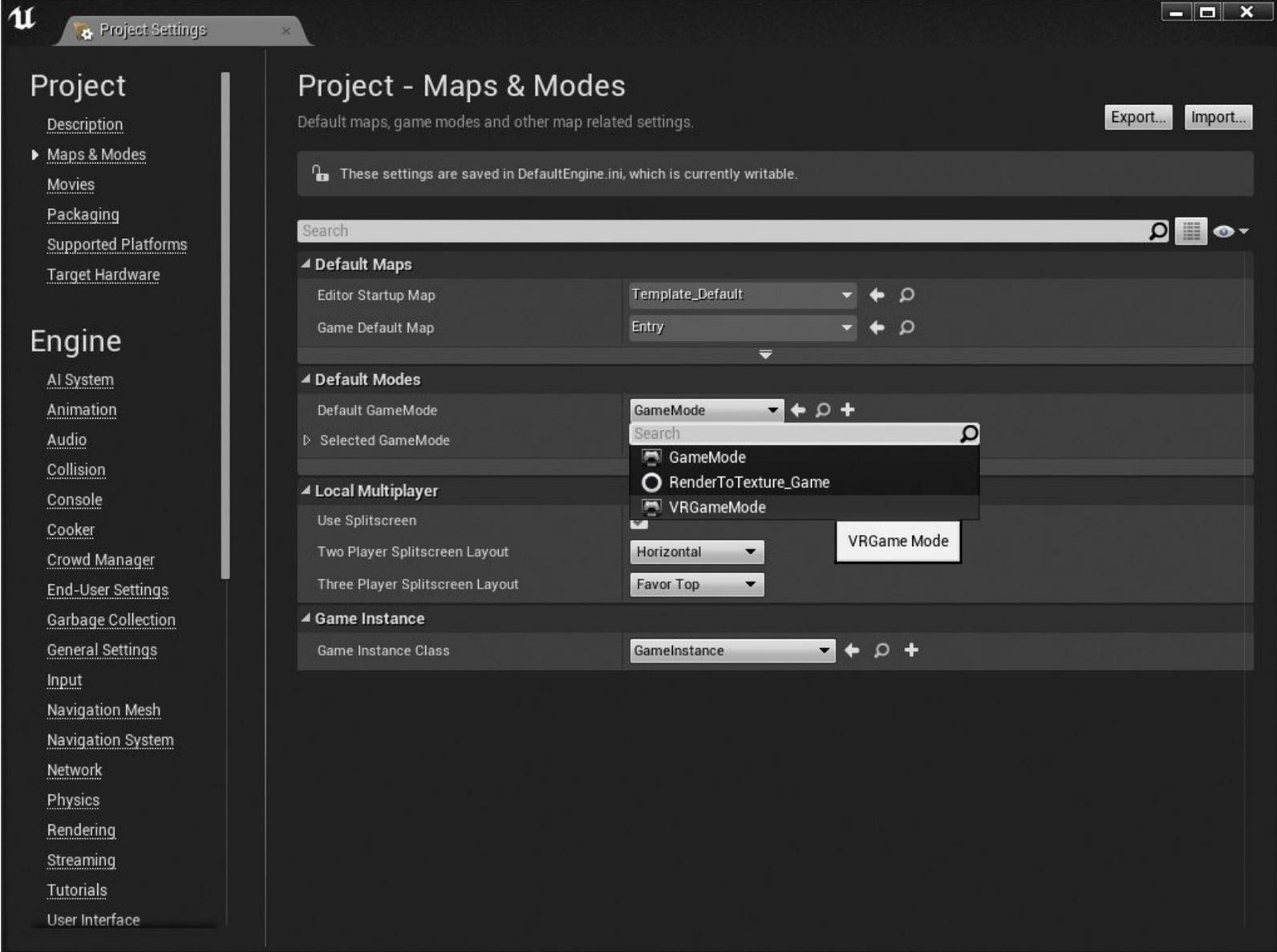


Figure 2.18 Desktop VR default game mode

4. Open your VRGameMode Blueprint and set the Default Pawn Class to the VRPawn you created (see [Figure 2.19](#)).

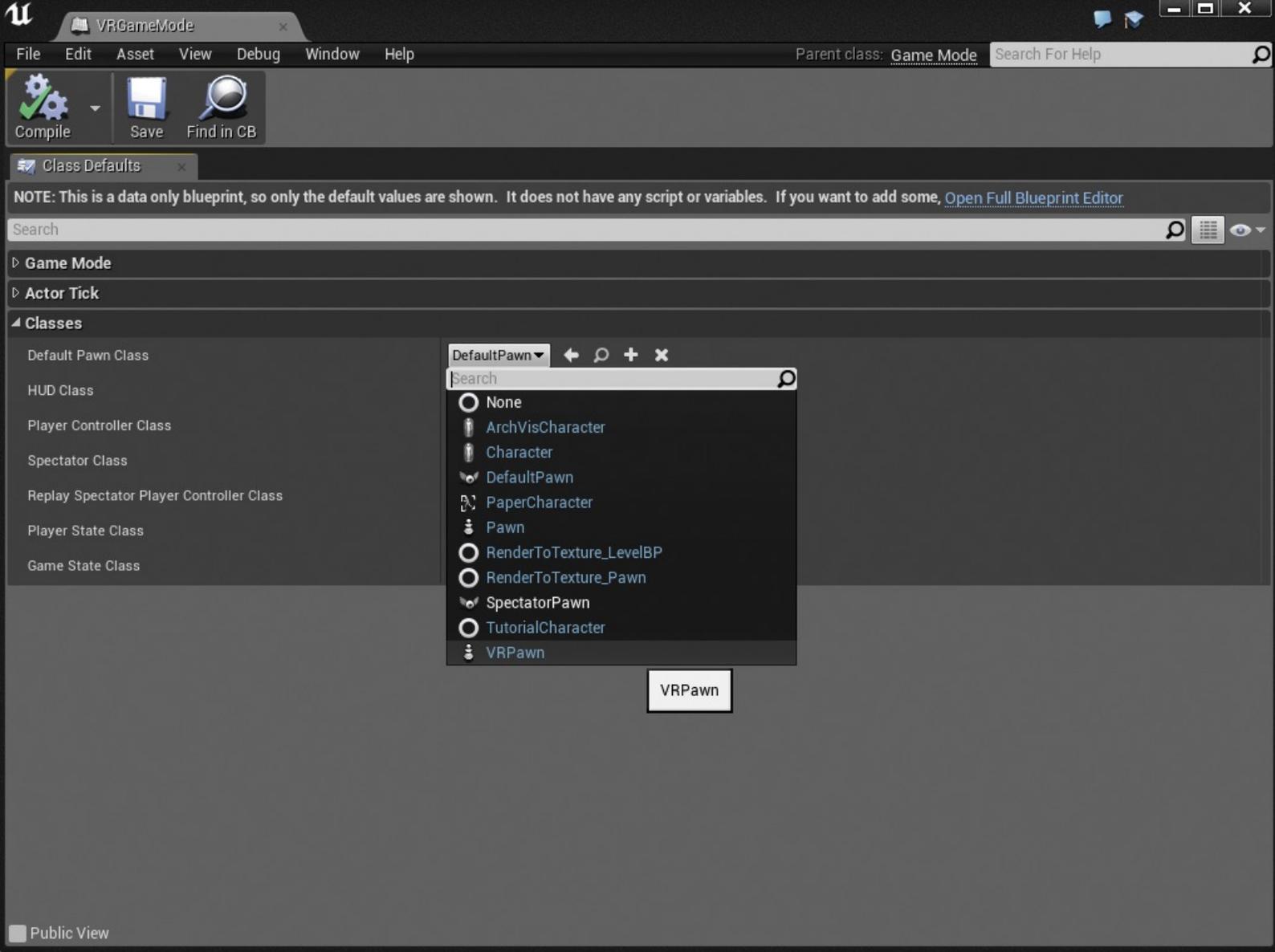


Figure 2.19 Desktop VR game mode

- 5. Open the VRPawn and add four new Components.
 - The first Component is a Scene named CameraRoot. This is the root of your Camera, and it allows you to position the Camera anywhere you wish in the Pawn.
 - Create a new Camera Component and make it the child of the CameraRoot by dragging it on top of the CameraRoot in the Components tab.
 - Create two Motion Controller Components; name the first MotionController_L and the second MotionController_R (see [Figure 2.20](#)).

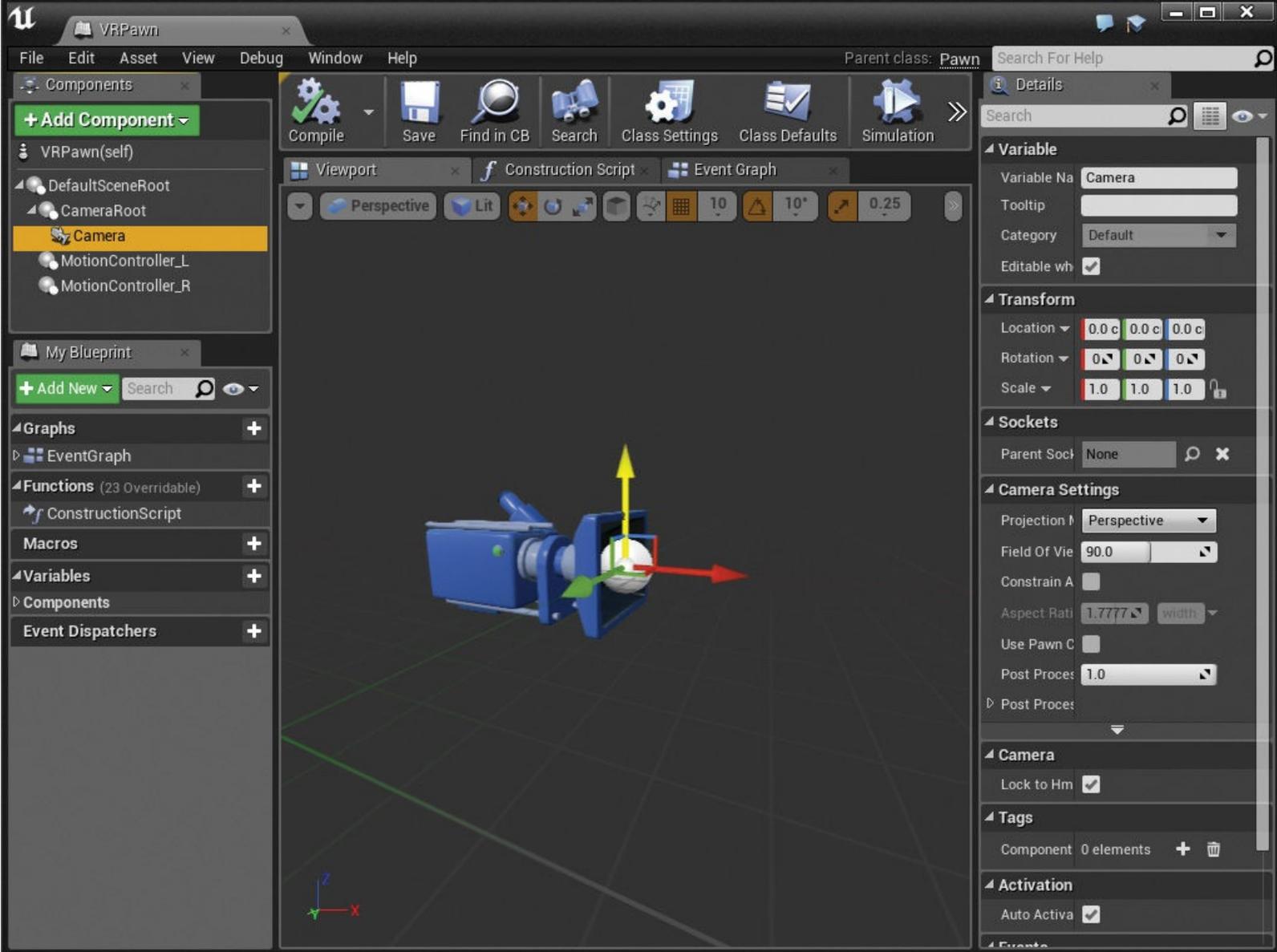


Figure 2.20 Desktop VR Player Pawn

- On the second motion controller, change the Hand variable in the Details panel to Right. This ensures that the motion controller is controlled by the right hand.
6. Open the Event Graph of the VRPawn. Drag off of the EventBeginPlay node and create a new SetTrackingOrigin node, making sure that the Origin input pin is set to Floor Level (see [Figure 2.21](#)). This sets up Rift and Vive for a standing experience; for different tracking origins, see the next section, “[Rift and Vive Tracking Origins](#).”

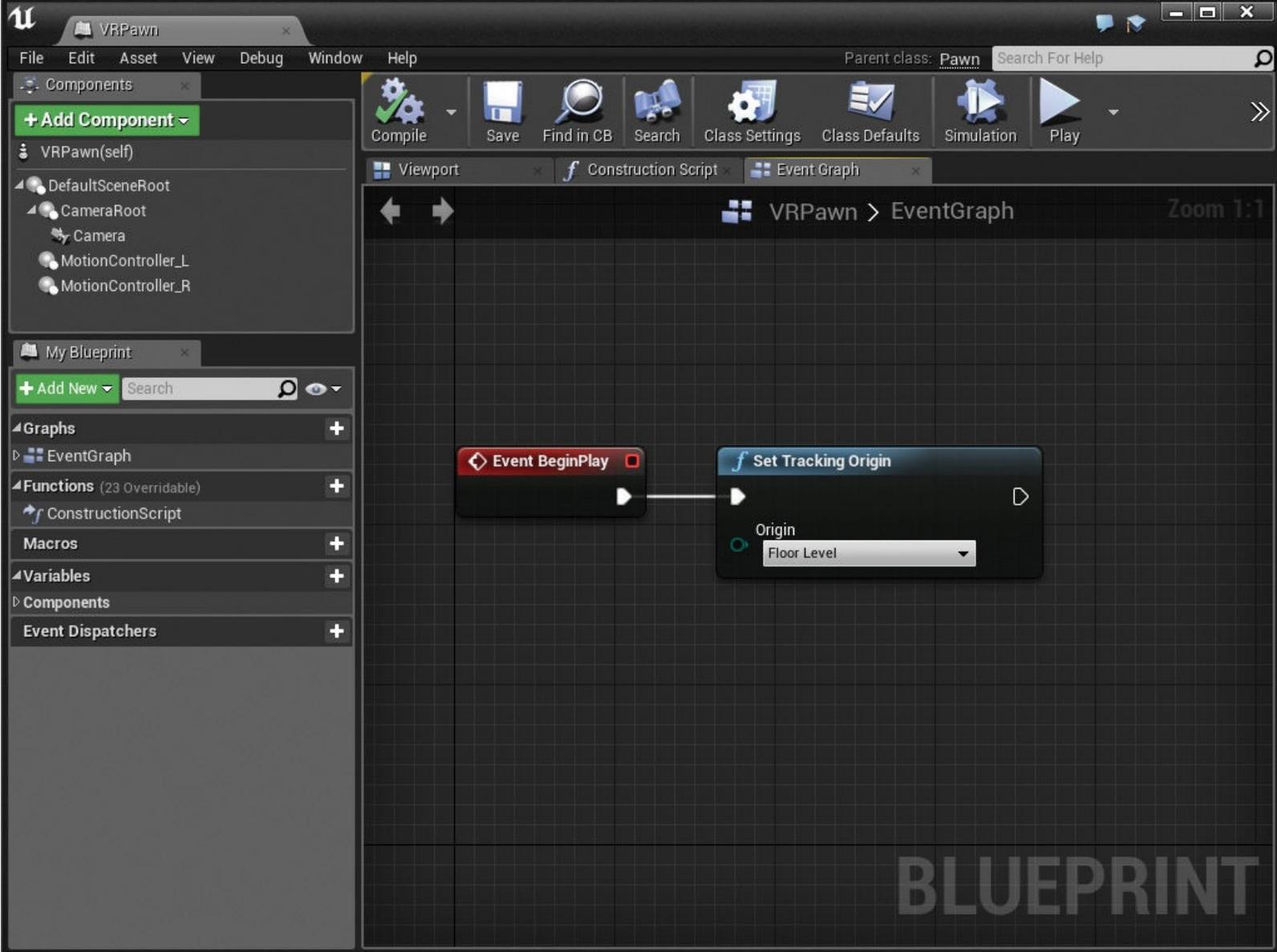


Figure 2.21 Desktop VR standing tracking origin

7. Now that you have set up the Pawn for a standing experience, you need to move the default Player Start location down to the floor because the HMD position is relative to the player's real-world floor. Grab the Player Start in the level and move its Z-axis to 20cm; this is the height of the default floor in the UE4 starter map (see [Figure 2.22](#)).

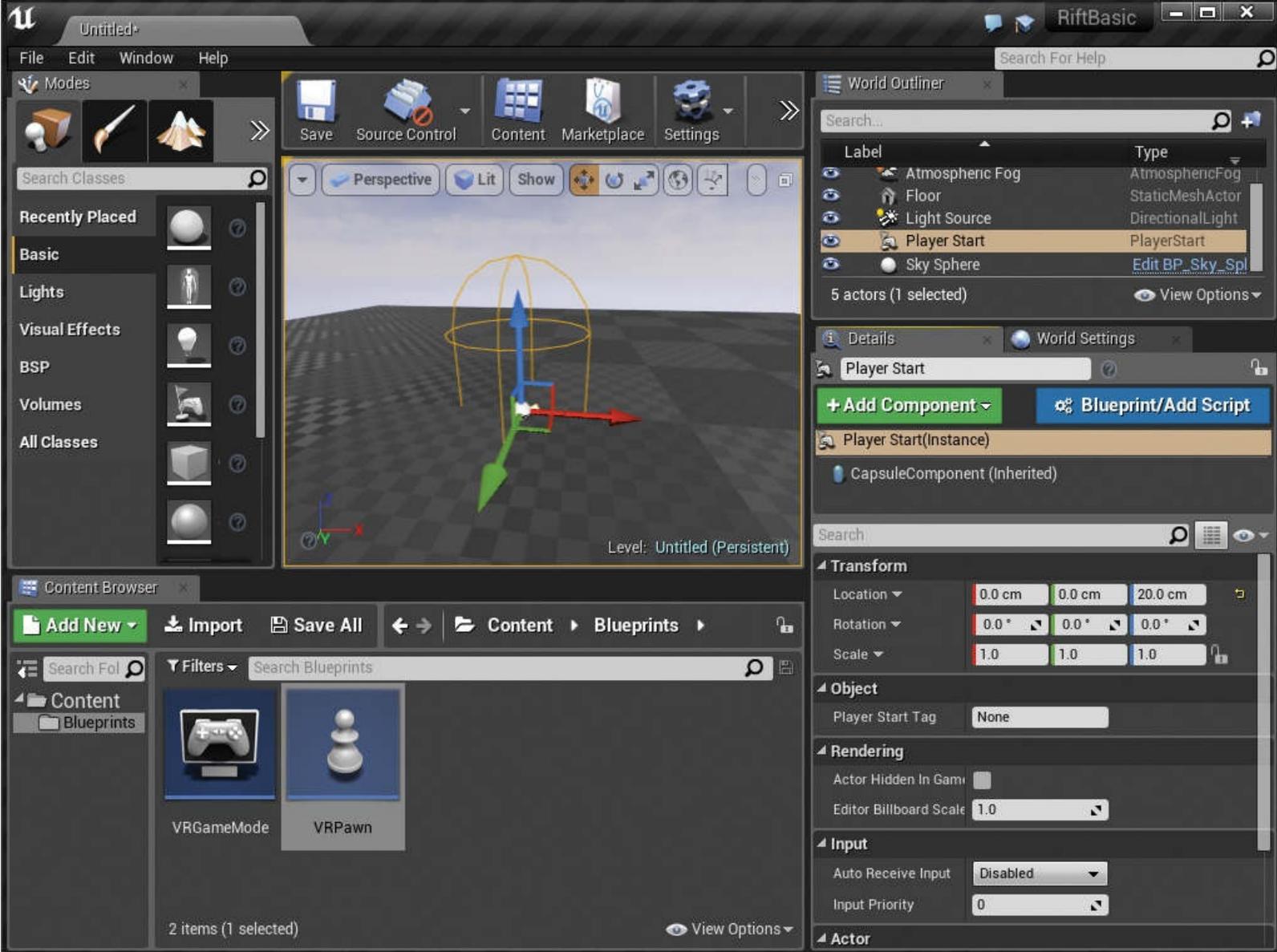
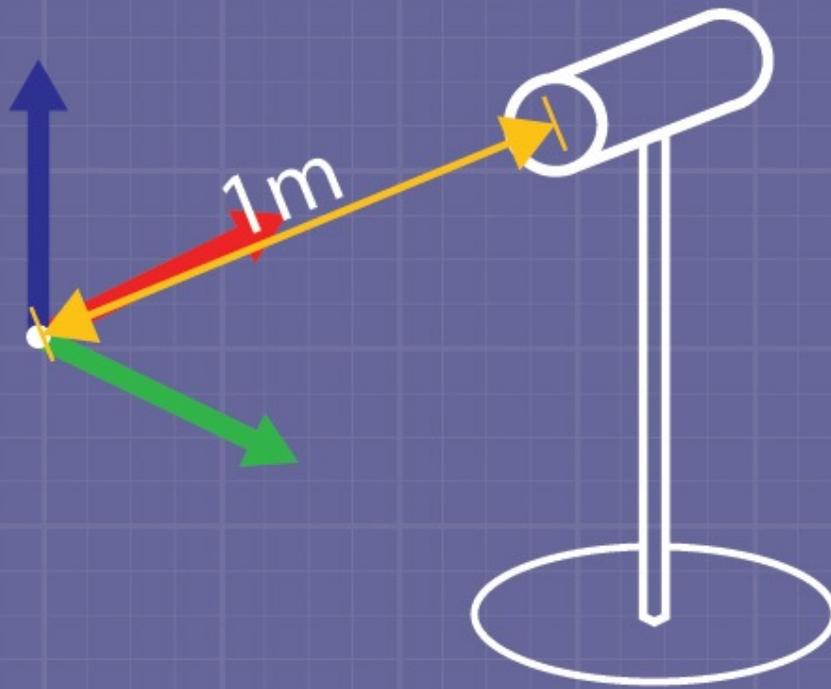


Figure 2.22 Desktop VR Player Start

Rift and Vive Tracking Origins

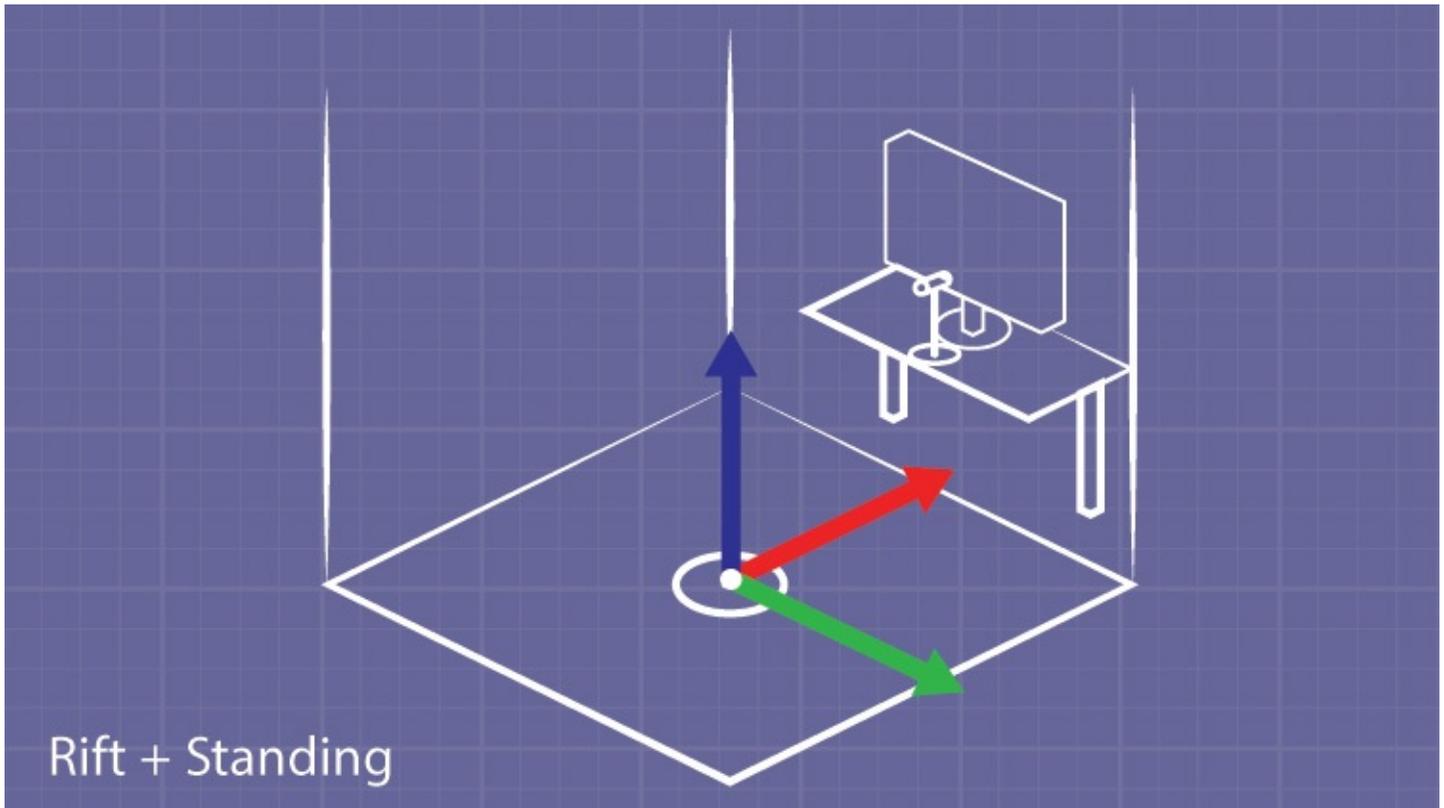
Both the Rift and the Vive offer the developer a way to target either a seated or a standing experience. To do this in UE4, you tell the engine what tracking origin you want to use (refer to [Figure 2.21](#)).

By default, the Rift targets a seated experience (or an Eye Level tracking origin) that places the VR camera's default location one meter horizontally in front of the available tracking camera (see [Figure 2.23](#)). However, if the tracking origin is set to Floor Level in UE4, the default camera location for the Rift is on the floor that the user calibrated when setting up the Oculus runtime (see [Figure 2.24](#)).



Rift + Sitting

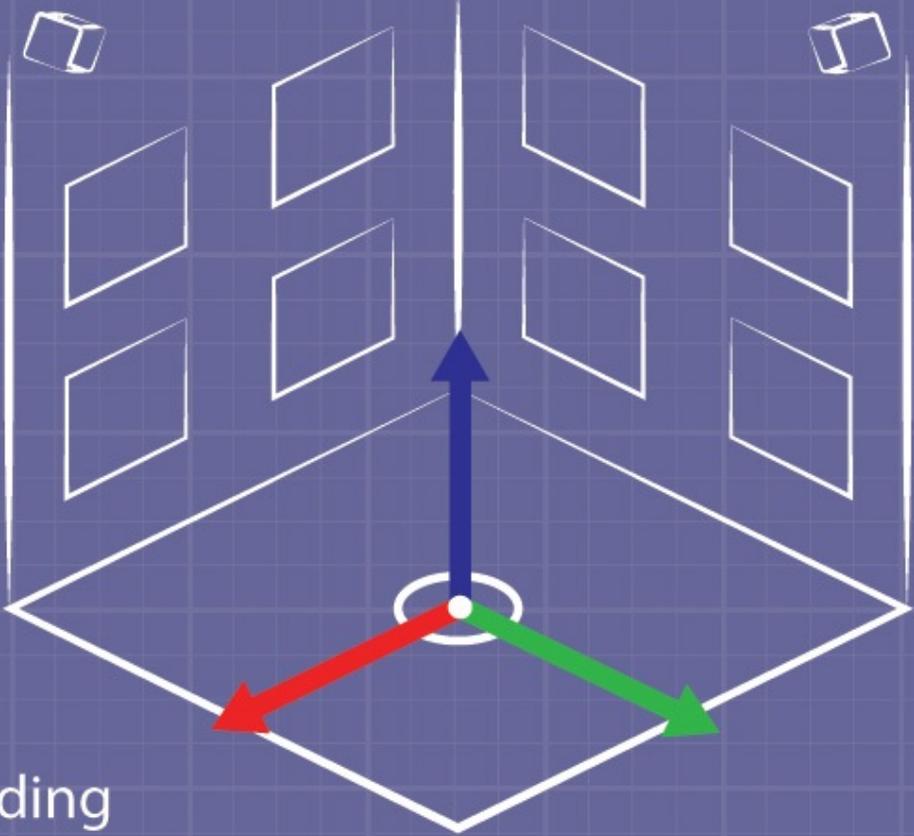
Figure 2.23 Rift Eye Level tracking origin



Rift + Standing

Figure 2.24 Rift Floor Level tracking origin

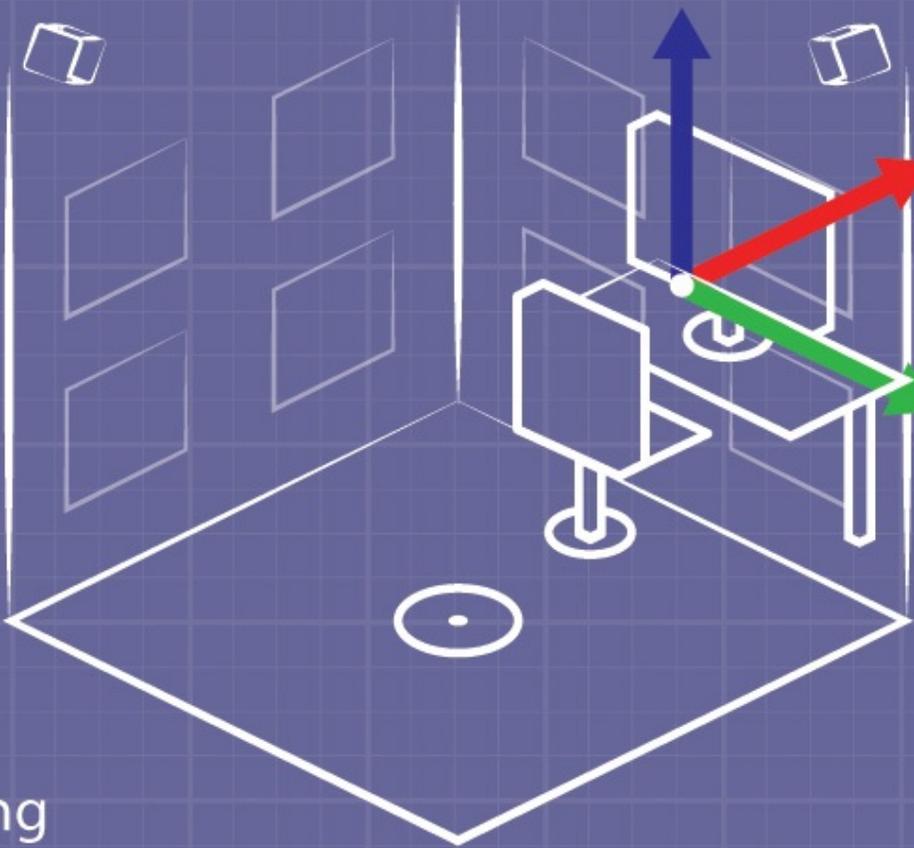
For the Vive, the default is to target standing experiences (Floor Level tracking origin). This locates the default camera position at the center of the user's play space and on the floor (see [Figure 2.25](#)). When targeting a seated experience, the Vive's tracking origin is, by default, in the middle of the monitor side of the user's play space (see [Figure 2.26](#)).



Vive + Standing

The diagram shows a 3D wireframe room with a grid floor. A white dot at the center of the floor represents the tracking origin. Three colored arrows originate from this dot: a blue arrow pointing vertically upwards, a red arrow pointing towards the bottom-left, and a green arrow pointing towards the bottom-right. Two small white cubes are positioned in the top corners of the room.

Figure 2.25 Vive Floor Level tracking origin



Vive + Sitting

The diagram shows a 3D wireframe room with a grid floor. A desk and chair are positioned on the right side of the room. A white dot on the desk represents the tracking origin. Three colored arrows originate from this dot: a blue arrow pointing vertically upwards, a red arrow pointing towards the top-right, and a green arrow pointing towards the bottom-right. Two small white cubes are positioned in the top corners of the room.

Figure 2.26 Vive Eye Level tracking origin

Summary

You now have a basis from which to create your own VR experiences. This chapter described mobile VR and creating the necessary menu-loading UI, as well as desktop VR and choosing whether you want your experience to target a seated or standing tracking origin (or both). You should now have everything you need to start looking at recipes and learning VR.

Chapter 3. Toolkit

UE4 provides a vast toolkit of Blueprint functions that you can use when creating your VR experiences. This chapter introduces some of the generic VR libraries that Unreal Engine offers, as well as some of the more vendor-specific ones.

Generic Function Library

When creating VR experiences in UE4, you have various helpers and functions available to you to query things from the current user's head position to the temperature of the HMD sensors.

Alongside the API-specific functions that Unreal Engine gives you to interact with an individual HMD's SDK, there is a generic Head Mounted Display function library that gives you access to various things common across multiple HMDs (see [Table 3.1](#)).

Function Name	Description	Compatibility
Get Positional Tracking Camera Parameters	Returns values such as Field of View for an HMD's tracking camera. Returns the position and rotation of the camera relative to the headset origin.	Supports only the Rift; the Gear VR has no positional tracking and the Vive does not use a camera. Deprecated in 4.13.
Get Tracking Sensor Parameters	Returns the values of a positional tracking sensor if it is connected.	Added in 4.13.
Get Orientation and Position	Returns the relative orientation and position of the headset from the HMD origin.	Works on all HMDs.
Is Head Mounted Display Enabled	Returns whether an HMD is connected and being rendered to.	Works on all HMDs.
Get Screen Percentage	Gets the current screen percentage/resolution scaling of the render target.	Will work only on Oculus devices; the Vive tracks its screen percentage through the <code>r.screenPercentage</code> console command, and the Rift through the <code>HMD SP</code> command.
Get Tracking Origin	Returns the current tracking origin for the HMD (either Floor or Eye Level).	Rift has a default of Eye Level and Vive has a default of Floor Level. Added in 4.11.
Get VR Focus State	Returns whether the engine has focus into the VR compositor.	Not implemented for the Vive. Added in 4.11.

Get World to Meters Scale	Returns the current World to Meters scale factor that allows for proportional scaling of the VR view.	Works on all HMDs.
Is in Low Persistence Mode	Returns whether the HMD display is in low persistence mode.	This is always True for the Rift and Vive.
Set Tracking Origin	Sets the tracking origin to either Eye or Floor Level.	Works on Rift and Vive. Added in 4.11.
Set Clipping Plane	Sets the VR specific camera clipping planes.	Not implemented for the Vive.
Set World to Meters Scale	Sets the World to Meters scale that allows for proportional scaling of the VR view.	Works on all HMDs.
Reset Orientation and Position	Resets the orientation and/or position of the HMD origin to the current HMD transform.	Works on all HMDs.
Enable HMD	Enables the HMD.	Works only in a stand-alone build of the game; otherwise the engine has priority over the VR compositor.
Get HMD Device Name	Returns the name of the currently connected HMD (for example, GearVR for the Gear VR).	Works on all HMDs. Added in 4.13.

Table 3.1 Generic Head Mounted Display Functions

Oculus Function Library

Along with the generic HMD function library, UE4 gives you access to SDK-specific libraries that allow you to access functions more specific to the HMD.

The Oculus Function Library allows you to have access to more low-level information about the HMD as well as information about the current Oculus user profile (see [Table 3.2](#)).

Function Name	Description	Compatibility
Get Base Rotation and Base Offset in Meters	Returns the offset of the current HMD's origin to the initial HMD origin of the game.	
Get Player Camera Manager Follow HMD	Returns whether Follow HMD is enabled on the camera manager.	Option was removed in 4.11.
Get Pose	Gets the HMD position and rotation as well as the neck position for the virtual head model.	
Get Raw Sensor Data	Returns the raw data from the headset's IMU; can be used for detecting little changes in motion.	
Get User Profile	Returns the profile information of the current user such as name, gender, and player height.	
Is Auto Loading Splash Screen	Returns whether the custom splash screen should automatically appear between level loads (defaults to True).	
Is Player Controller Follow HMD Enabled	Returns information on the player controller's Follow HMD status.	Option was removed in 4.11.
Add Loading Splash Screen	Allows you to specify a custom texture and transform for the custom splash screen.	
Show Loading Icon	Shows a single 2D texture similar to a splash screen without the custom transform.	Added in 4.13.

Hide Loading Icon	Hides a shown loading icon.	Added in 4.13.
Is Loading Icon Enabled	Returns whether a loading icon is being displayed.	Added in 4.13.
Set Position Scale 3D	Scales the positional tracking of the HMD.	Unimplemented.
Clear Loading Splash Screens	Clears any splash screen that is currently being displayed.	
Enable Auto Loading Splash Screen	Sets whether the custom async splash screen should automatically appear between level loads (defaults to True).	
Enable Player Camera Manager Follow HMD	Enables Follow HMD on the player camera manager.	Option was removed in 4.11.
Show Loading Splash Screen	Shows the custom splash screen.	
Enable Player Controller Follow HMD	Enables Follow HMD on the player controller.	Option was removed in 4.11.
Hide Loading Splash Screen	Hides the custom splash screen.	
Set Base Rotation and Base Offset in Meters	Sets a custom offset for the HMD origin.	

Table 3.2 Oculus Functions

Note

The functions in [Table 3.2](#) work only on the Gear VR and Rift headsets; they specifically target these headsets.

Steam VR Function Library

Steam VR has fewer functions than the Oculus Function Library but gives you access to the Vive Controllers and Base Stations.

Similarly to the Oculus Function Library, the functions in [Table 3.3](#) work only on Steam VR devices.

Function Name	Description	Compatibility
Get Hand Position and Orientation	Returns the position and orientation of either the left or right hand of a set of Steam VR controllers relative to the HMD origin.	Does not take into account the late update found on the motion controller components.
Get Tracked Device Position and Orientation	Given a device ID, this returns that device's orientation and position relative to the HMD origin.	
Get Valid Tracked Device IDs	Returns an array of IDs for a given device type.	

Table 3.3 Steam VR Functions

Summary

You now understand the types of functions that are available when creating your VR experiences, whether it is from the generic HMD function library or a device-specific one. You now have all that you need to try recipes and create your own experiences.

Part II: Recipes

Chapter 4. Trace Interaction

VR allows developers to tap into previously inaccessible natural forms of input. A major one of these is the player's head, and so this chapter discusses a way to use this input to interact with objects in your VR worlds.

Understanding traces is essential for nearly all VR games. In this chapter, you create a modular system for user interaction that can be easily expanded.

Understanding Trace Interaction

Trace interaction in VR is a common method of allowing the player to interact with the in-game world by detecting what object that player is looking (or, in the case of motion controllers, pointing) at.

A trace (or ray cast) has a broader definition in the context of UE4 and an even broader definition in the context of computer graphics in general. However, when I mention traces in this book, I refer to the set of functions inside Unreal Engine that allow you to set a start and end point and detect any objects that are between those two points.

Unreal Engine gives you the option of many different traces. They range from the simple line trace (see [Figure 4.1](#)), which detects whether the line that exists between a start point and an end point is blocked by any object, to more complicated shape traces, which allow you to define a box, sphere, or capsule and detect whether any object intersects that object as it is traced from the start to the end point (the sphere trace is shown in [Figure 4.1](#)).

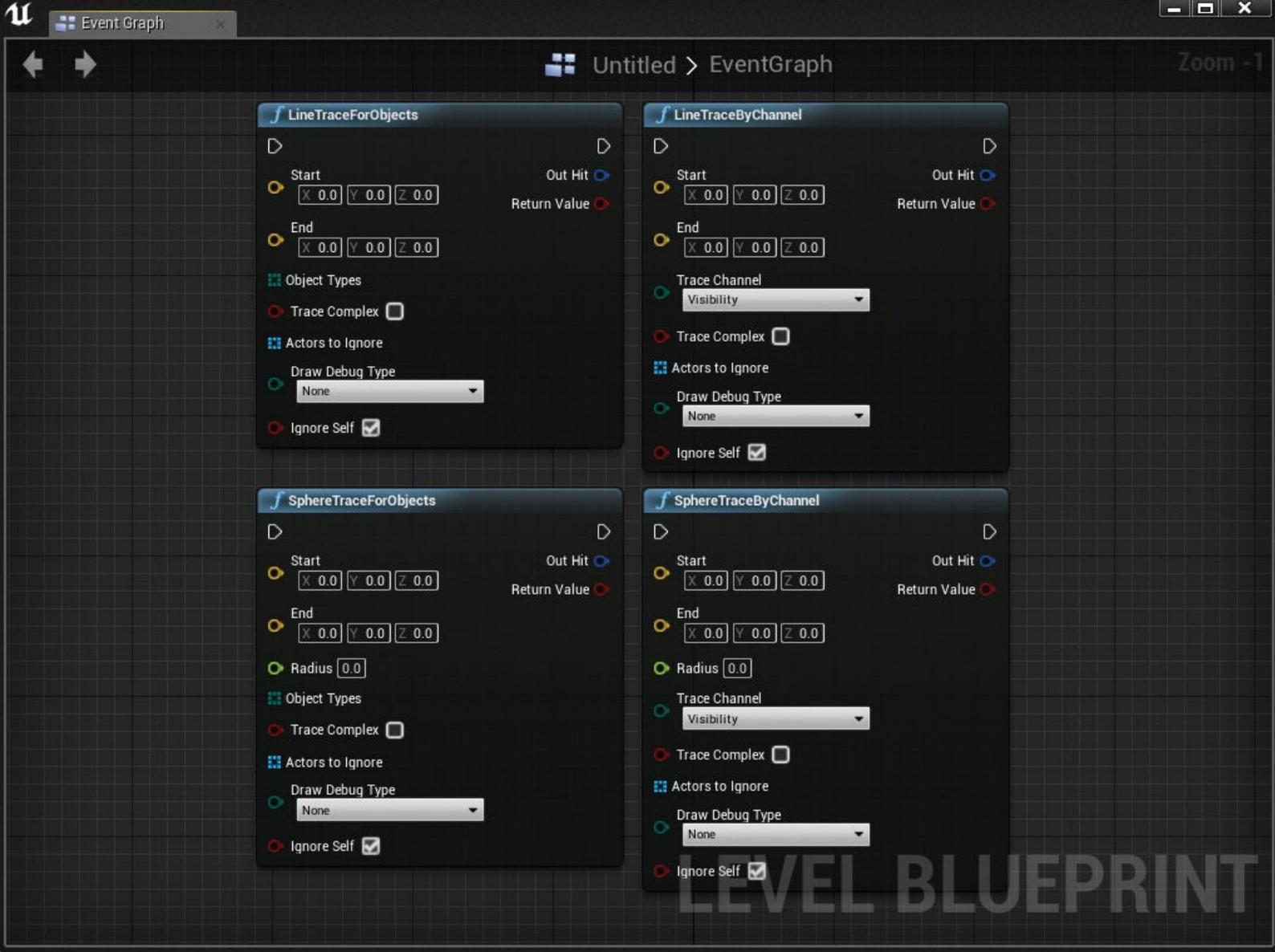


Figure 4.1 A sample of the trace functions offered in UE4

In addition to the different trace shapes (line, box, etc.), each shape has two distinct flavors: the Object trace and the Channel trace (shown in [Figure 4.1](#)). There are slight differences between them, and therefore it can be a bit difficult to know which one to choose if you're just starting out.

By dissecting them, you'll see that both flavors (Object and Channel) offer the same input parameters aside from one; that is, the Object Types parameter for Object traces and the Trace Channel parameter for Channel traces. Those who have worked with Unreal Engine's collision system before know that UE4 offers two types of collision responses: Object and Trace. These collision channels match perfectly to the two trace types. This means that by using the appropriate trace function, UE4 allows you to search (trace) for specific types of objects in your world while ignoring those that you don't care about. The one other difference between these trace types is that whereas Object traces allow you to define multiple types to trace against (which can be useful when you want to test for specific types of objects like players or vehicles), Channel traces allow only one trace channel to be selected. By default, these are Visibility and Camera; however, just as with custom Object collision responses, more can be added through the Projects Settings.

How do we actually set up one of these traces?

In your game, you have access to two main properties that allow you to calculate the start and end point of where the player is looking. Both properties are represented as vectors (a grouping of numbers that can be

used to represent a point or direction in a specific coordinate space). The first of these is the player's head location in world space, which means that the X, Y, and Z of this vector represent a forward, right, and up value relative to the in-game world's origin point. The second of these vectors is the player's head forward direction, which you obtain from the in-game camera representing the player's eyes. These two vectors are shown in [Figure 4.2](#).

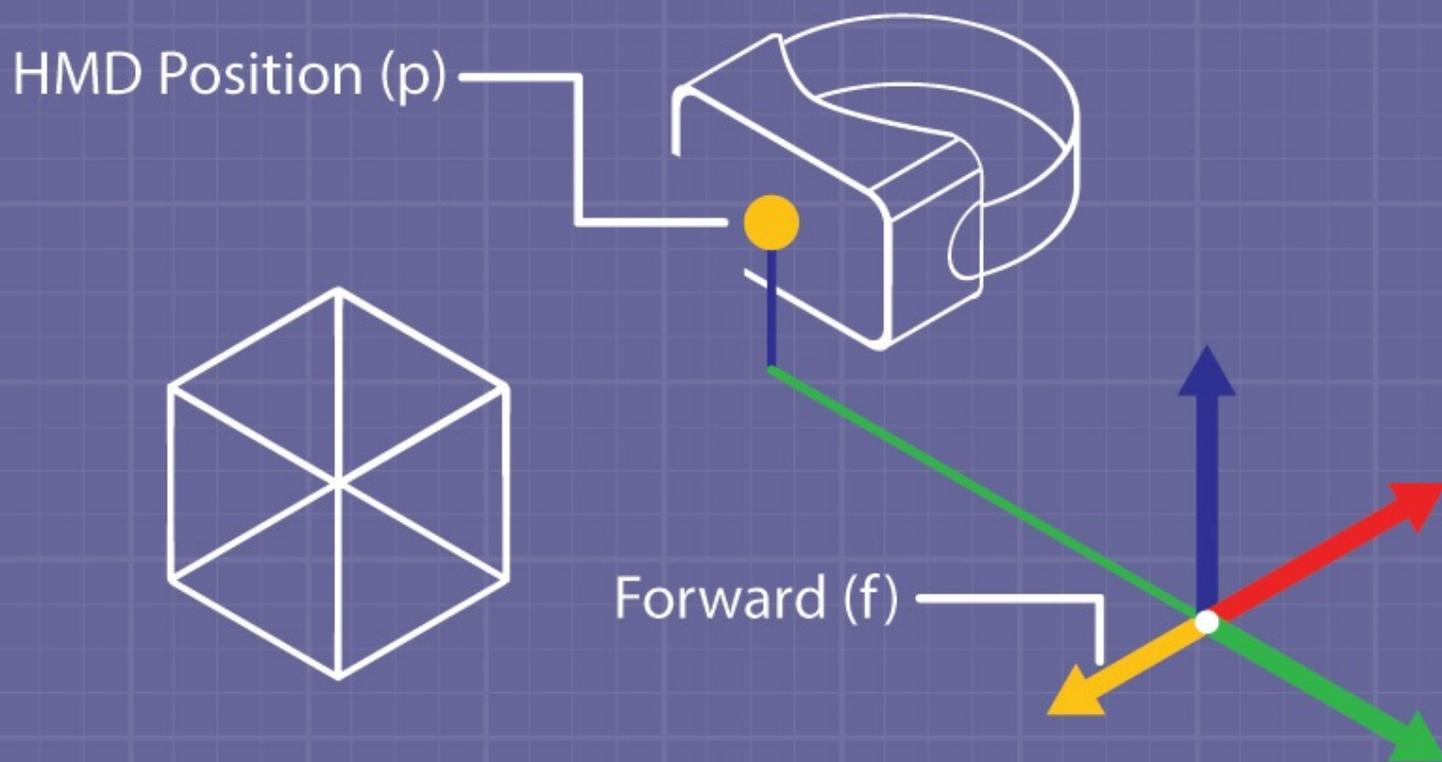


Figure 4.2 The HMD's position (p) and forward look (f) vectors represented in UE4

Note

In this example, you use the player's head as the place from which to trace. However, note that this also works with any other object in the game from which you want to trace (such as a motion controller).

To use these variables for tracing, you can first pass the world position of the HMD directly into the start position of the trace. Second, to get the end position of the trace, you need to do some simple math. Before we do this, however, a little background is needed. A direction vector in UE4 always has a length of 1; this has the nice property that scaling it to any other length is as simple as multiplying this direction vector by a number representing the length you want the vector to be. (In this case, this is how long you want the trace to be. Remember that 1UU [Unreal Unit] is equivalent to 1cm, so a length of 100UU is 1m.) Once this direction has been scaled to the length you want, you need to add it to the player's head location to position it relative to the player's head (see [Figure 4.3](#)) rather than to the world origin in which it was initially.

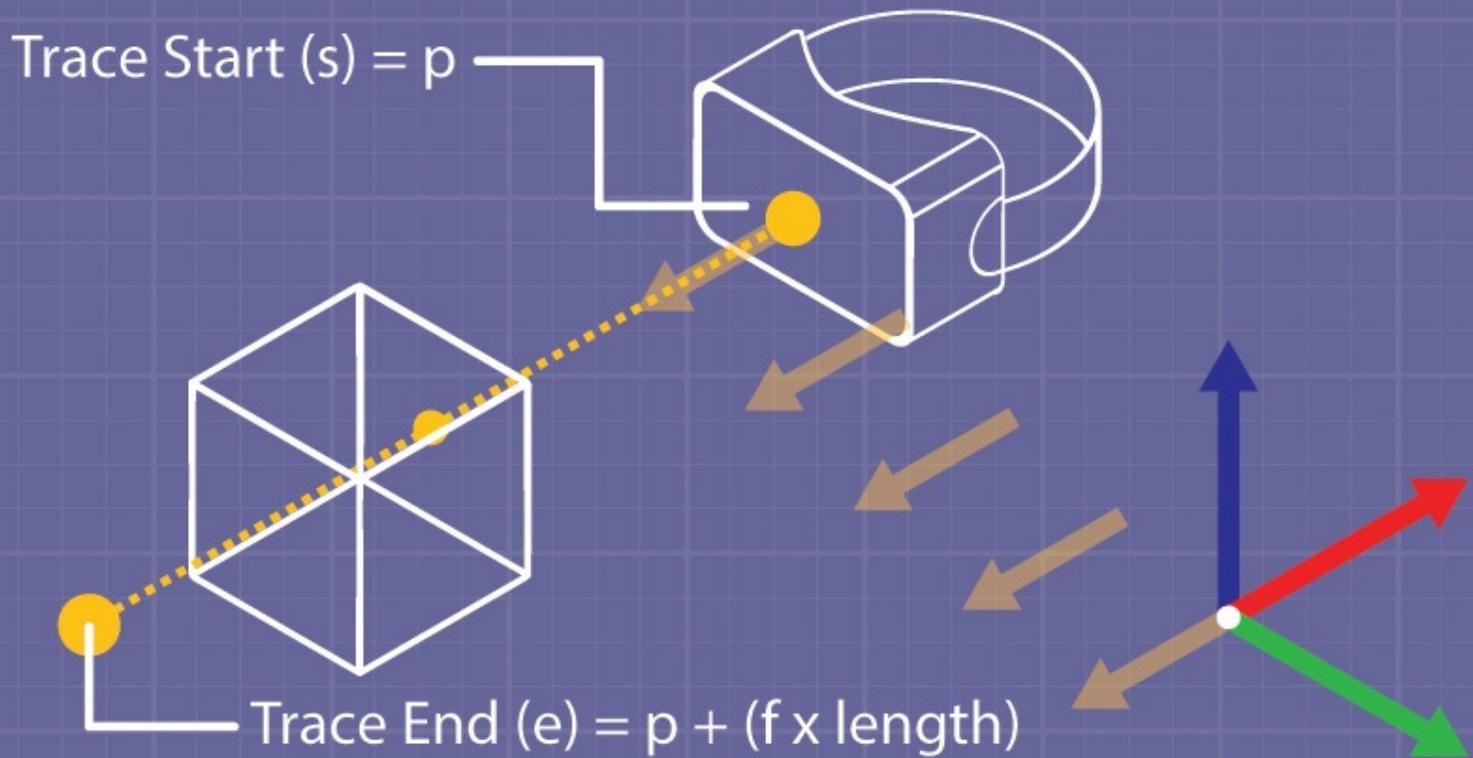


Figure 4.3 Calculating the start and end points for the trace (the larger orange circles) by scaling the look vector of the HMD (f) by the desired length, then adding it to the HMD's world position (p) to make sure the end point is in world space

To get the world origin and direction vectors in your game, use the `GetWorldLocation` and `GetForwardVector` Blueprint nodes (see [Figure 4.4](#)) from your desired Component, in this case a Camera Component.

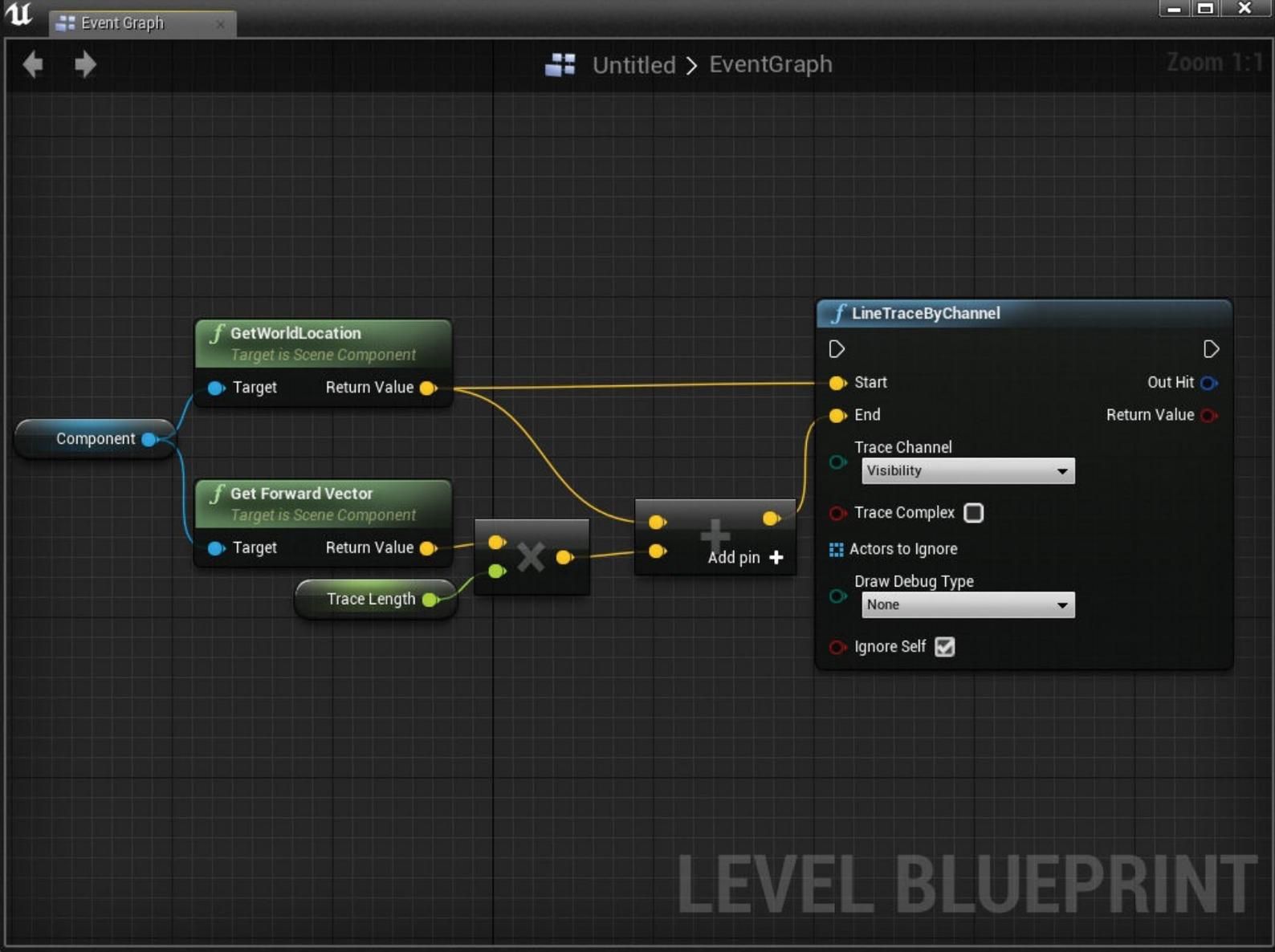


Figure 4.4 A simple trace function setup to find objects in front of a Component

Understanding Interfaces

An interface is one of the core objects you will use to create an interaction system. It allows your in-game character to communicate with other objects.

If you have programming experience and are familiar with the idea of interfaces, skip this section and move to the section [“Setting Up Trace Interaction.”](#)

In programming, an *interface* (also called a protocol) allows you to define a set of functionalities that any object can promise to implement and perform when called upon. This is useful for things like interaction systems where you first define a set of interactions that players can make. Then any object in the world can tell players that they are capable of handling these interactions. Note that the players do not need to know anything about how the implementing object will handle that interaction.

For example, you may have a ball that starts rolling when the player looks at it, and you may also have a cube that spins when looked at. Using a Look Interaction interface allows you to call the Look At functionality on both of these objects without the player knowing (or caring) that the ball needs to roll and the cube needs to spin.

This decoupling of trigger and action creates an interaction system that is easy to expand and to which you can add more interactive objects while having only one object (usually the player) that decides what

counts as a look and what does not.

Readers familiar with object-oriented programming may know that another way to give functionality to objects is for them to inherit functionality from their parent. This indeed works as an interaction system, because you can guarantee that if the object you are looking at has a parent that has the interaction functionality, you can call the correct functions when needed.

However, this has one major downside: in Blueprints, at least, an object can have only one parent. This means that if you wanted to have both a cube and a pig that can be interacted with, they would need to inherit from the same object, even though in your game it may make more sense for the pig to inherit from an Animal class and the cube to inherit from a Shape class. (This difference is shown in [Figure 4.5.](#))

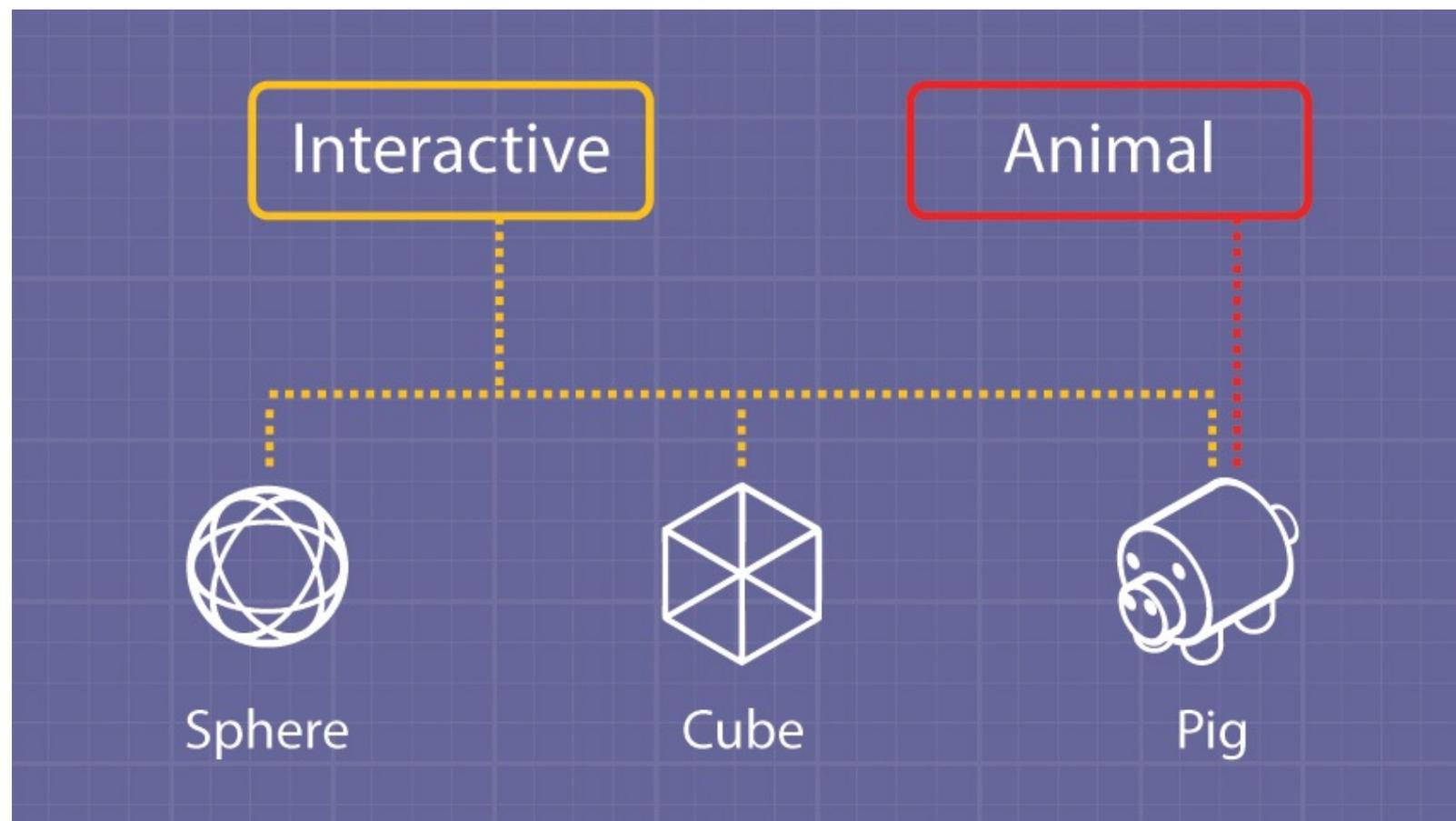


Figure 4.5 Interfaces versus classes. Using an interface (yellow) allows the cube, sphere, and pig to be interactive, but the pig can still have a parent class (red) of Animal; this would be impossible in Blueprints if the interface was a class because only single inheritance is allowed.

Setting Up Trace Interaction

Let's now implement trace interaction so you can use it in your games and experiences.

Basic Project Setup

To begin, you need a few objects to set up a trace interaction system:

1. Create a folder under the root content folder of your project and name it Blueprints.
2. Inside the Blueprints folder, create two more folders and name them Components and Interfaces.
3. Inside the Blueprints folder, create a new Pawn (select Add New, then select the Pawn Class).
4. Name this newly created Pawn TraceInteractionPawn.
5. Inside the Components folder, create a Scene Component named TraceInteractionComponent. This will hold the logic for the trace functionality. It would be easier from a programming perspective

not to separate the trace functionality from the Pawn, but when the TraceInteractionComponent is its own Component, you can give any object in your game the ability to perform trace interactions, which will ultimately make your code more flexible and easier to port to other projects.

6. Inside the Interfaces folder, create a new Blueprint interface called TraceInteractionInterface by selecting the Add New button, and under the Create Advanced Asset section hover over the Blueprints menu and select Blueprint Interface.

Your project should now resemble [Figure 4.6](#).

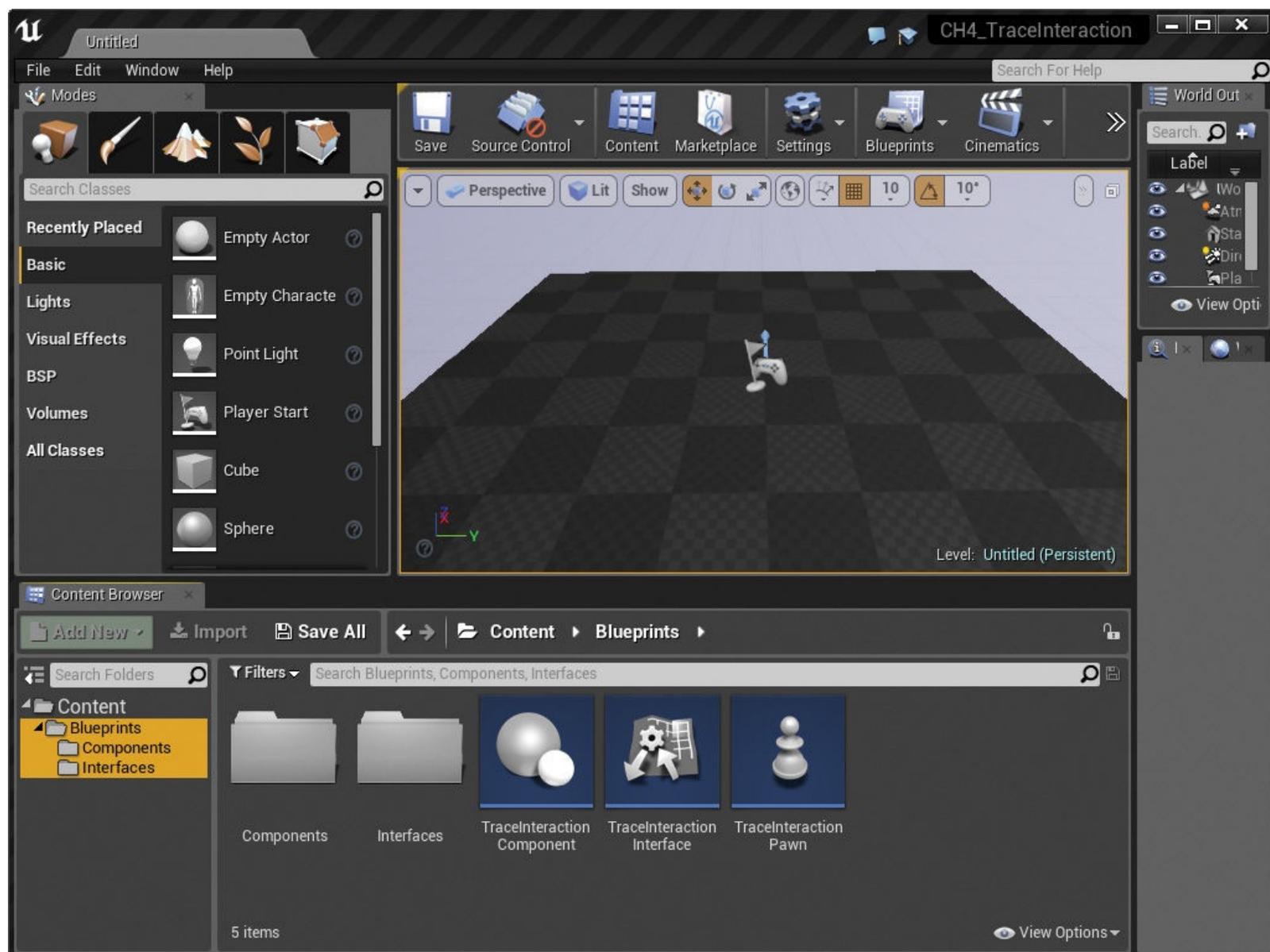


Figure 4.6 Trace interaction project setup

Interaction Interface Setup

To allow the Interaction Component to call functions on interactive objects, you need to define them in the interface you created in the initial project setup.

These functions allow for a basic trace interaction system that will handle most cases of detecting hits on objects and Components; however, if you need specific custom functionality, adding new functions to this Blueprint is usually your first step.

1. Open the TraceInteractionInterface Blueprint.
2. You need seven functions for this interface. Call the first two TraceHitObject and TraceLeaveObject. You use these to tell objects that a trace has hit or has left them.

3. For the inputs of these two functions, create an input of type Hit Result (name it Hit). Passing the Hit allows you to find out many useful things from the trace on the hit object.
4. Create two more functions, TraceHitComponent and TraceLeaveComponent. These act as the hit detection for objects on a Component level rather than on the object as a whole.
5. For the inputs of these functions, create the same Hit Result input as for the previous two, but this time add a new Component input of type Primitive Component. This gives you easy access to the Hit Component for the hit function while providing the Component that was left for the leave function, which would not be accessible in the Hit Result.
6. Create one more function, TraceMove. This function is called every frame when the trace is over the hit object.
7. Because this function is called every frame, we want it to be a little more optimized than the other functions in this interface. Therefore, add the Hit Result input variable as before, but this time expand this variable and check the Pass-by-Reference check box. This allows the Hit variable to not be copied every time this function is called and rather just passes in a reference to the existing variable in memory.
8. Using a pass-by-reference variable in Blueprints, however, means that you need to mark this function as const, which means that it is not allowed to directly edit the state of the Blueprint. You can do this by expanding the Graph tab in the Details panel and checking the Const option (an example of this can be seen in [Figure 4.7](#)).

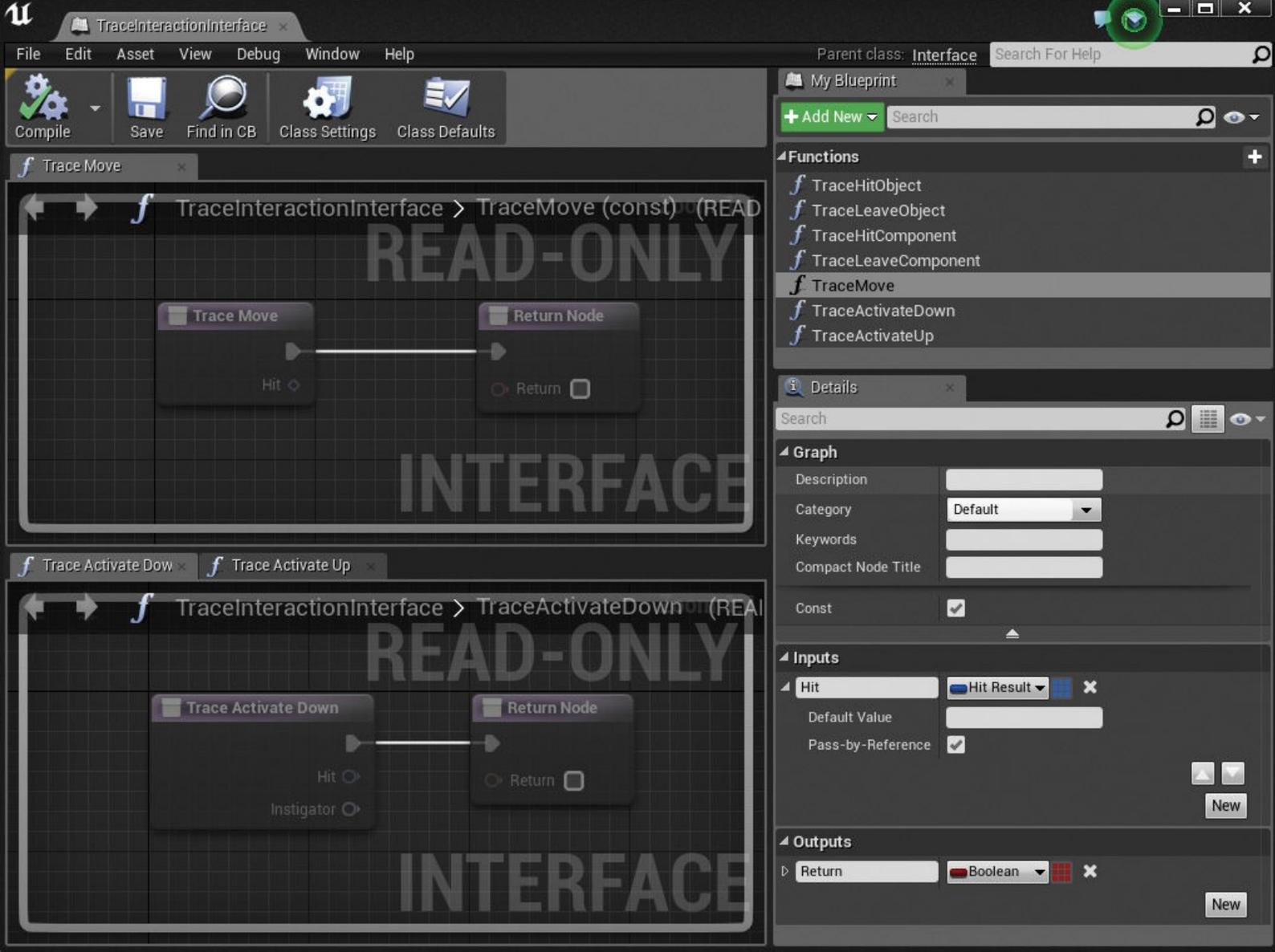


Figure 4.7 Trace Interaction Interface setup. Each function has a dummy Return Value to allow it to remain in one section on the implementing Blueprint. TraceMove has its Hit Result passed by reference to improve performance.

9. Create the last two functions, TraceActivateDown and TraceActivateUp. These are used to activate the hit object.
 10. Add the same Hit Result input variable that you added for the other functions, but this time add a new Pawn input variable, Instigator, that represents the Pawn that activated the object.
 11. Review all the functions and add a Boolean output variable named Return to all of them. This may seem like an odd thing to do, but because of the way interfaces work in Blueprints, if no output variable is given they will show up as event nodes in the Event Graph. Therefore, in an attempt to have all the functions in one place, you can add what essentially is junk output to each of them.
- All the functions in the interface should now look something like [Figure 4.7](#).

Interaction Component

The bulk of the interaction logic now needs to be handled in the Interaction Component. For this example, you set up a simple line trace and interact with the hit object from that. Later in this chapter, we will go over a few other methods that can be used for detecting with which objects to interact.

Setup

First, you need to do a few things before you actually handle the tracing and the interaction:

1. Open the `TraceInteractionComponent`.
2. Create two new functions, `LineTrace` and `InteractWithHit`. The first handles the trace itself, and the second takes the output of the line trace and decides how to interact with the hit.
3. Create three new variables, the first named `FocusedComponent` of type `Primitive Component`, the second `FocusedObject` of type `Actor`, and the last `CurrentHit` of type `Hit Result`. The first two hold the current hit `Component` and object respectively, and the last holds the hit information.
4. Mark these variables as `Private`. You don't want to give other Blueprints direct access to them.
5. In the Event Graph of the Blueprint, drag off of the `EventTick` node and add a new `Gate` node, making sure the `Start Closed` input pin is unchecked.
6. Create two new custom events, `Enable` and `Disable`. These allow you to enable and disable the trace functionality when needed.
7. Drag the output of the `Enable` event to the `Open` pin of the gate.
8. Drag the output of the `Disable` event to the `Close` pin of the gate.

This basic setup should now resemble [Figure 4.8](#).

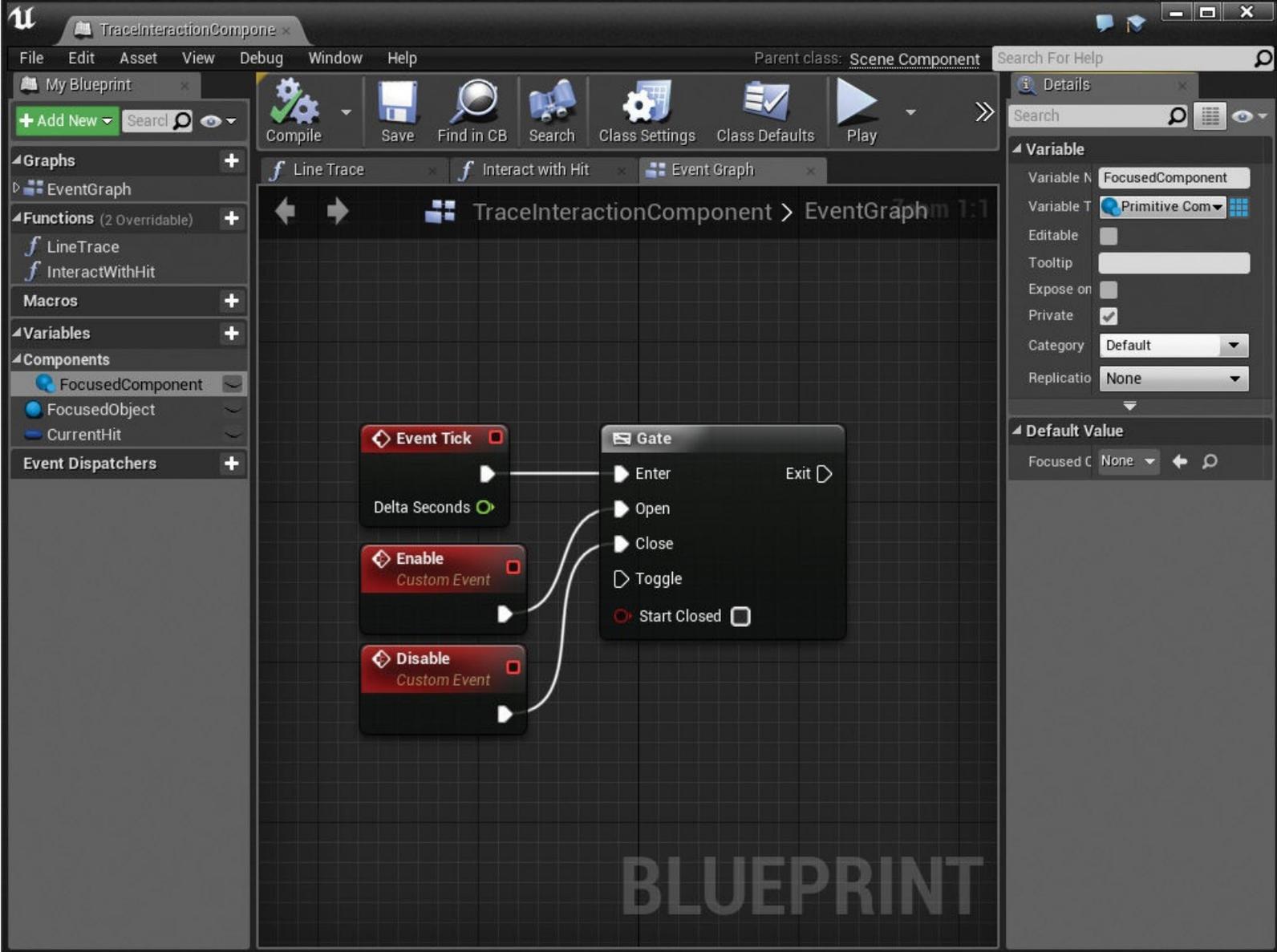


Figure 4.8 Trace Interaction Component basic setup. The Component is enabled by default; however, Enable and Disable events are available.

Line Trace Function

Now, you need to fill in the functionality of the functions you added in the preceding section. To get started, let's look at the LineTrace function:

1. Open the LineTrace function by double-clicking it.
2. Click the input node and create a new input variable named Distance of type Float. This is the distance you trace from the Component.
3. Create a new output variable named Hit of type Hit Result. This is the Hit Result of the trace.
4. Right-click the graph and add a new GetWorldLocation node by searching for it.
5. Right-click the graph again to add a GetForwardVector node. This gives you access to the forward direction of the Interaction Component and, in this case, the look direction of the player.
6. Create a new LineTraceByChannel node, connecting it between the input and the output nodes of the function.
7. Drag the output of the GetWorldLocation node to the Start input pin of the trace.
8. Drag off of the GetForwardVector node and call the Vector * Float function, putting the Distance

input pin of the function into the second slot. This scales the forward vector of the Component to the length we want.

9. Drag off of the GetWorldLocation node again, call the Vector + Vector function, and connect the output of the scaled forward vector to the second input pin.
10. Attach the output of the Vector + Vector function to the End input pin of the trace.
11. Connect the Out Hit output pin of the trace to the Hit pin of the Return node.

This function should now look similar to [Figure 4.9](#).

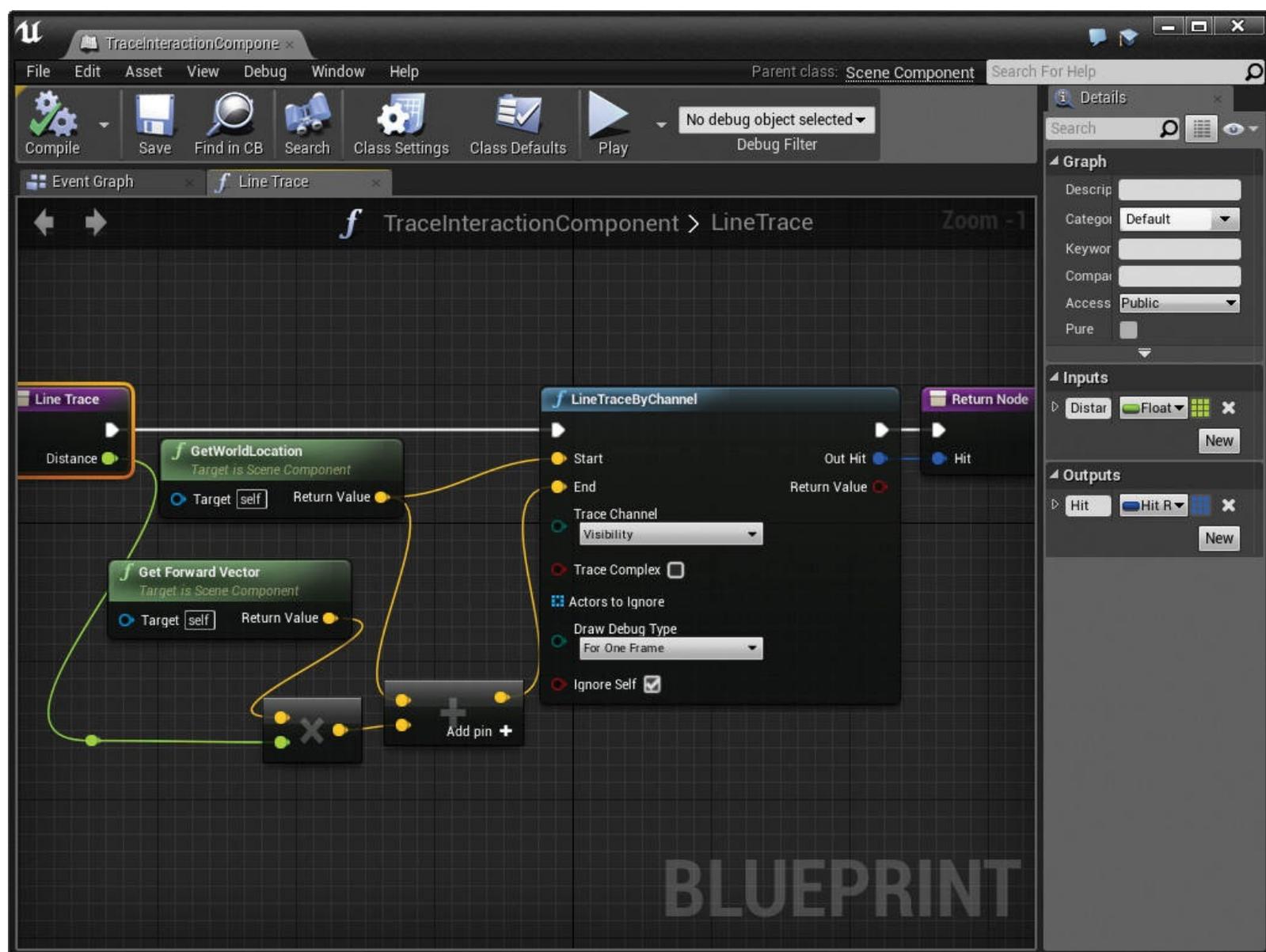


Figure 4.9 Trace Interaction Component LineTrace function

Interaction Function

Now that you have the hit that you want to interact with, you can add some logic to determine which interface messages to call on the hit objects:

1. Open the InteractWithHit function.
2. Add three local variables named InHit, InHitActor, and InHitComponent, of types Hit Result, Actor, and Primitive Component respectively. These act as temporary variables to give you easy access to parts of the input, ultimately allowing you to have a cleaner Blueprint graph.
3. Create a new input variable named Hit of type Hit Result (see [Figure 4.10](#)).

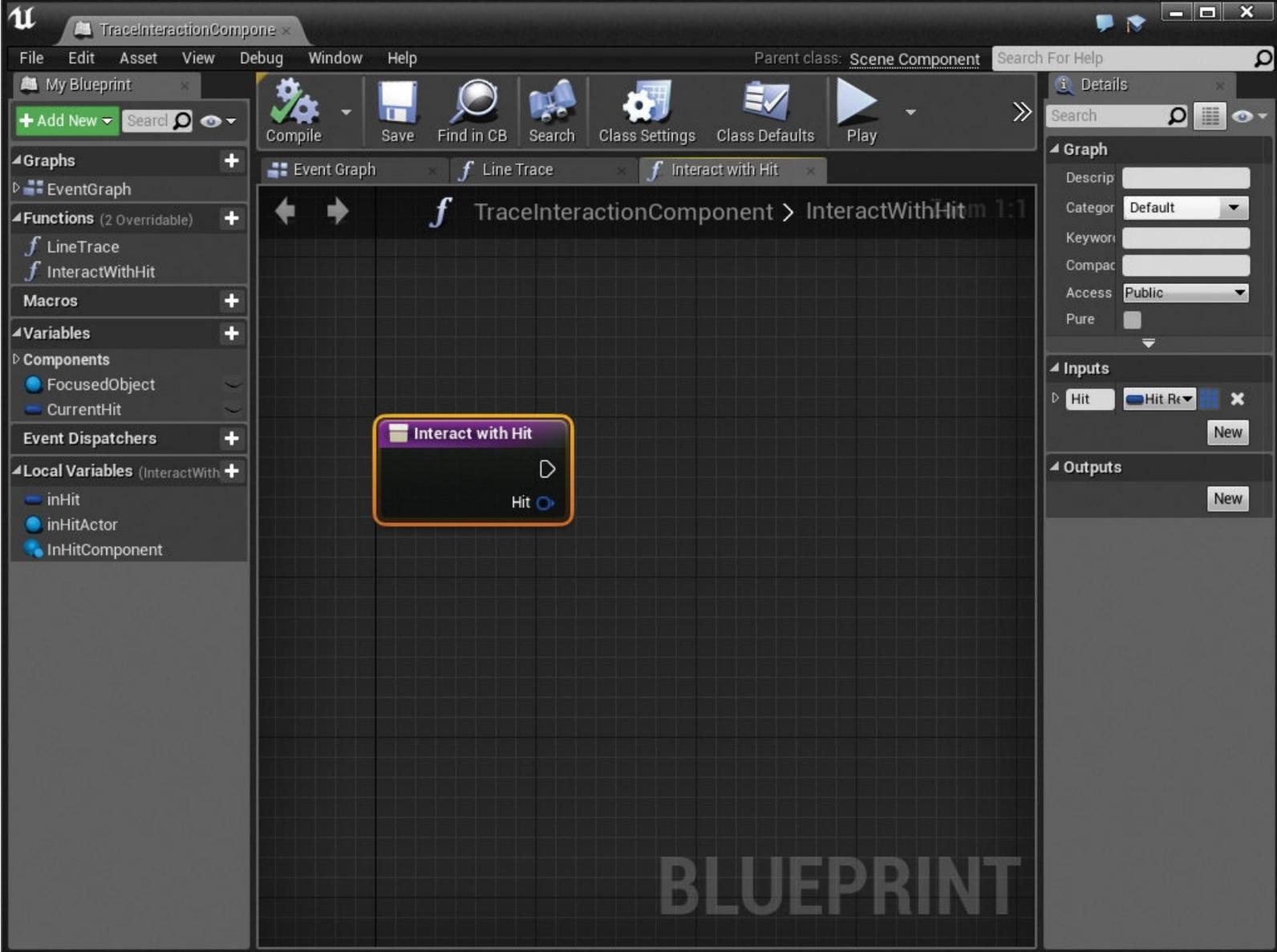


Figure 4.10 Trace Interaction Component interaction with Hit: local variables and inputs

4. Drag off of the input pin, create a setter node for the InHit variable, and connect the Hit pin of the input event to it.
5. Break the output of this setter by dragging off of its output and selecting BreakHitResult.
6. Create a setter for the InHitActor and InHitComponent variables, connecting the Hit Actor of the BreakHitResult node to the first setter and the Hit Component of the same break to the second setter.
7. Connect the output execution pin of the InHit setter to the InHitActor setter, then connect that InHitActor to the InHitComponent (see [Figure 4.11](#)).

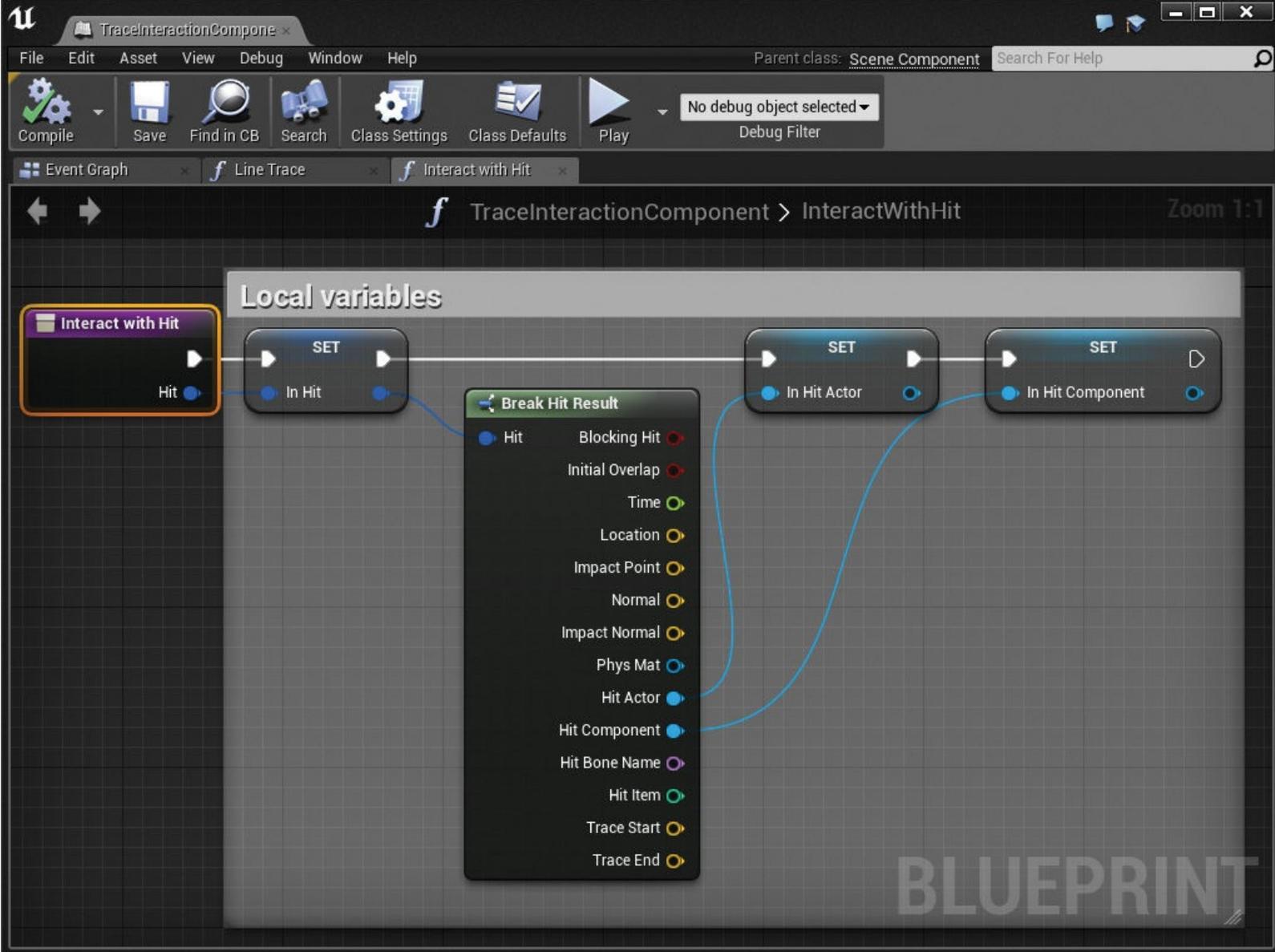


Figure 4.11 Trace Interaction Component interaction with Hit: local variable setup

8. Drag off of the InHitComponent setter and create a new Branch node.
9. Drag two variable getters for the InHitActor and FocusedObject variables.
10. Add an Equal (Object) node connecting both these getters to the two inputs.
11. Connect the output of the Equal (Object) node to the Condition input of the Branch (see [Figure 4.12](#)). This allows you to detect whether the player is looking at the same object or has started looking at a new one.

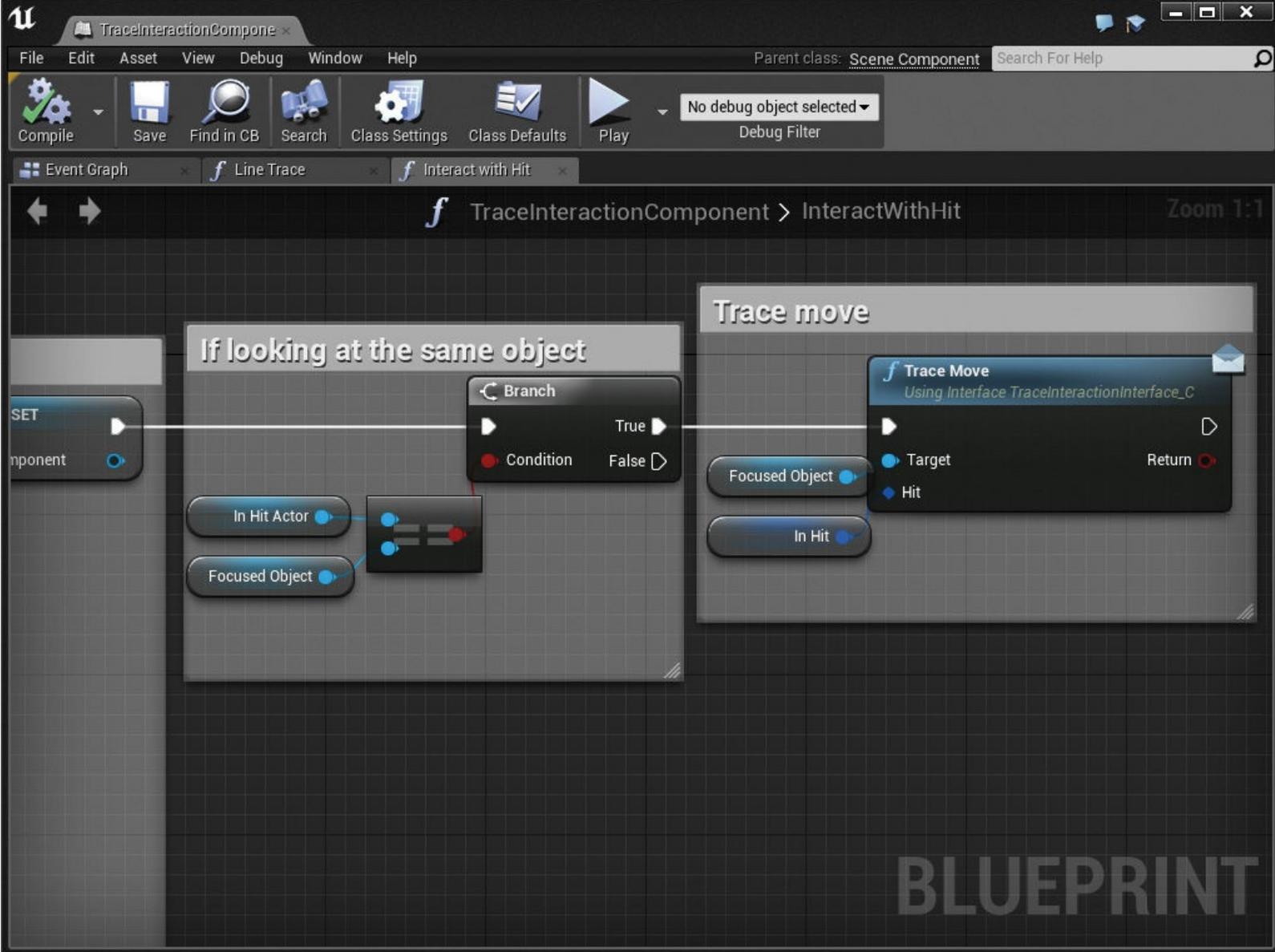


Figure 4.12 Trace Interaction Component interaction with Hit: looking at the same object and calling TraceMove

12. Create two new variable getters: one for the FocusedObject variable again and one for the InHit variable.
13. Drag off of the FocusedObject getter and call the TraceMove interface message. This calls the TraceMove function on the FocusedObject if it implements the interface you created in the first step.
14. Connect the InHit getter to the Hit input pin of the TraceMove message, and connect the True output of the Branch to the TraceMove node's input execution pin (see [Figure 4.12](#)).
15. Drag off of the output execution pin of the TraceMove message and create another Branch node.
16. This time, drag in two variable getters for the FocusedComponent and InHitComponent variables. Drag off of the first and add a new Equal (Object) node, and connect the second to the second input of this new Equal (Object) node.
17. Connect the output of the Equal (Object) node to the Condition input of the new Branch node. This allows you to detect when the player is looking at a new Component inside the currently looked-at object.
18. Create three new variable getters for the FocusedObject, InHit, and FocusedComponent variables.
19. Drag off of the FocusedObject getter and call the TraceLeaveComponent interface message,

connecting InHit to the Hit input pin and FocusedComponent to the Component input pin (see [Figure 4.13](#)). This ensures that you call leave on the currently focused Component before you set the new Hit Component.

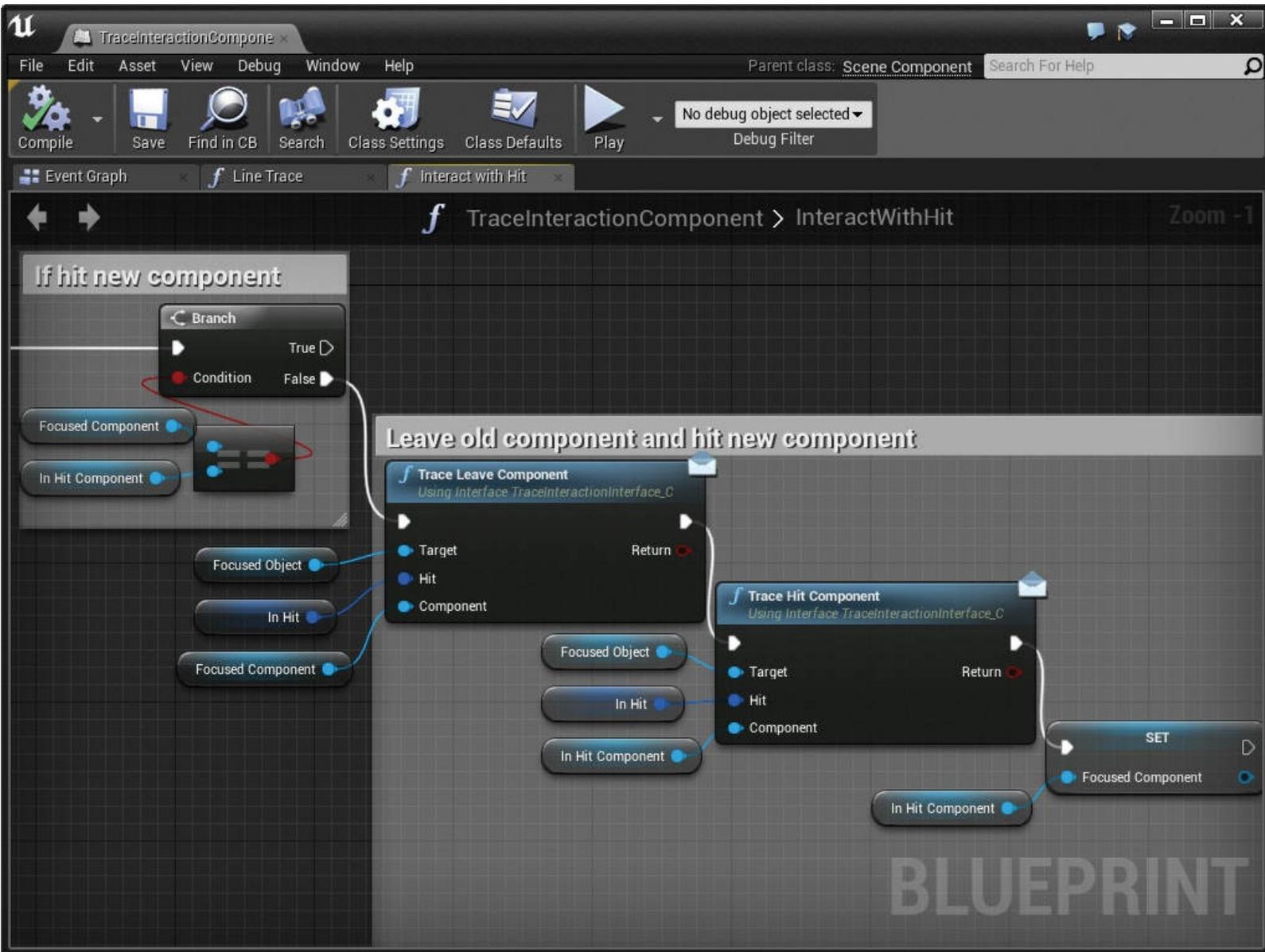


Figure 4.13 Trace Interaction Component interaction with Hit: calling Hit on a new Component

20. Connect the False output of the Branch node to the input execution pin of the TraceLeaveComponent message.
21. Create three more variable getters, this time for the FocusedObject, InHit, and InHitComponent variables.
22. Drag off of this new FocusedObject getter and call the TraceHitComponent interface message, connecting InHit to the Hit input pin and InHitComponent to the Component input pin.
23. Connect the TraceLeaveComponent message to the TraceHitComponent message.
24. Create a new variable setter for the FocusedComponent variable, and connect a new variable getter for the InHitComponent variable to its input pin.
25. Connect the output execution pin of the TraceHitComponent message to the new setter (see [Figure 4.13](#)). This sequence of calling leave on the current Component, then calling hit and remembering the new Component, is something you will use again for activating a hit on a new object.
26. Going back to the initial Branch after setting the local variables (from step 8), create a new

variable getter for the FocusedObject variable and call the TraceLeaveObject interface message from it, making sure to also connect the False output execution pin to the input execution pin of the message node.

27. Create a new variable getter for the InHit variable and connect it to the Hit input pin of the new message node.

28. Create two new variable setters for the FocusedObject variable and the FocusedComponent variable, connecting them in sequence after the interface message (see [Figure 4.14](#)). Leave their inputs blank because you want to clear their values.

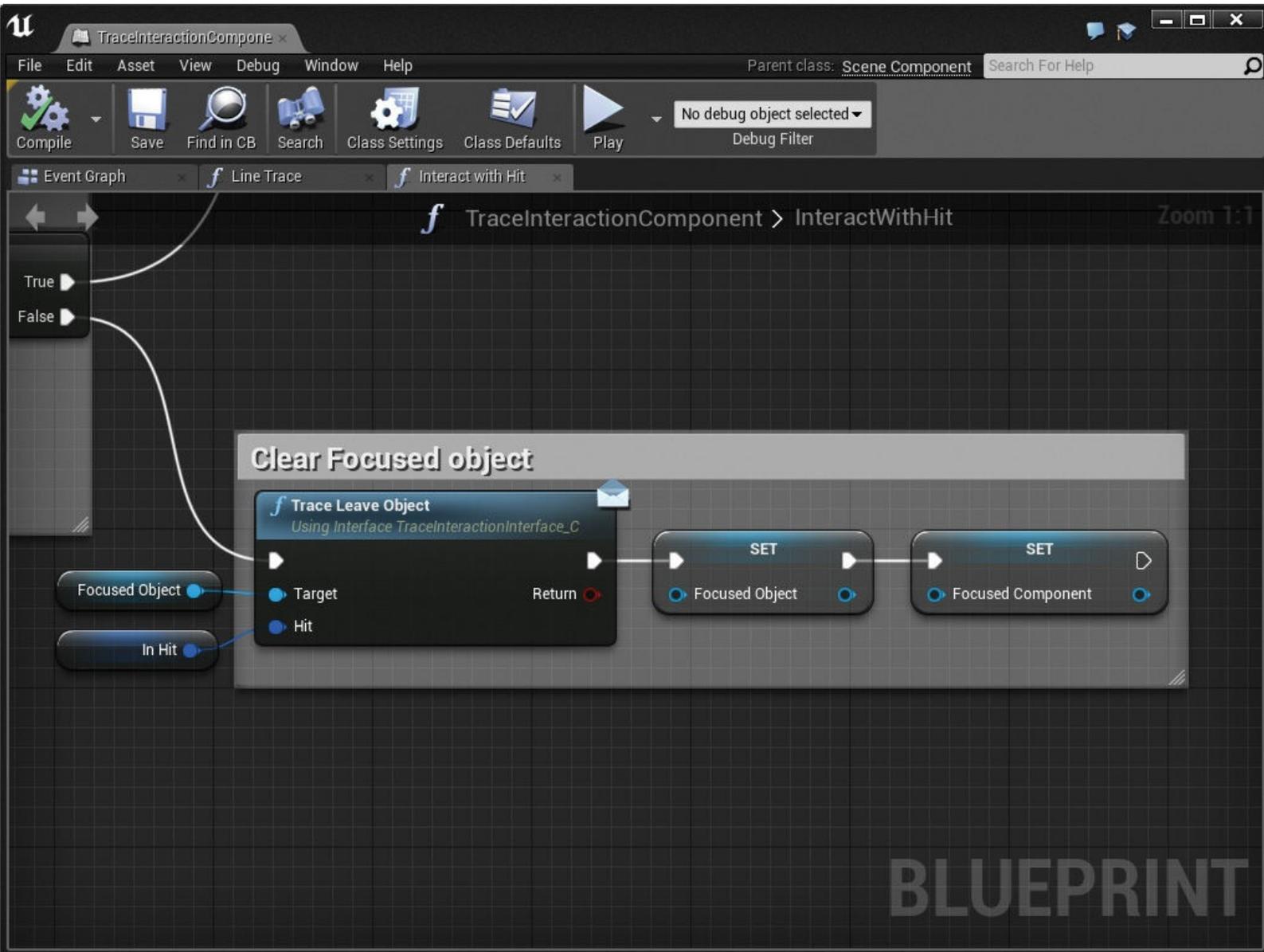


Figure 4.14 Trace Interaction Component interaction with Hit: clearing the FocusedObject

29. Drag off of this last setter and create a new Branch node.

30. Drag in a new getter for the InHitActor variable and call the DoesImplementInterface function, selecting TraceInteractionInterface as the interface class input pin. Connect this new node to the Condition input of the Branch node (see [Figure 4.15](#)). This ensures that the currently hit Actor uses the TraceInteractionInterface.

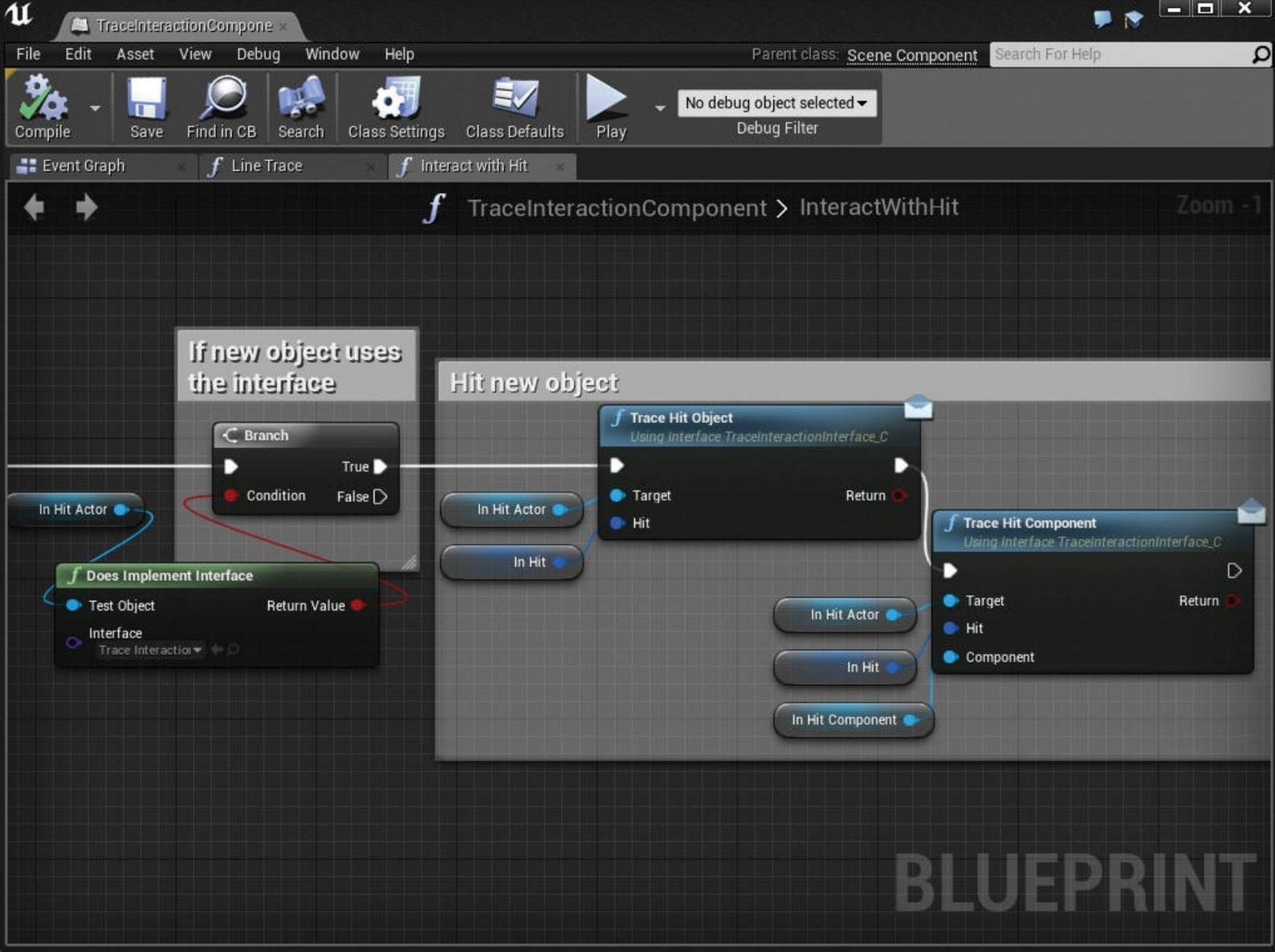


Figure 4.15 Trace Interaction Component interaction with Hit: calling Hit on a new object

31. Create two new variable getters for the InHitActor and InHit variables, and call the TraceHitObject interface message on the first while connecting the InHit variable to the input Hit pin.
32. Connect the True output execution pin of the Branch node to the input execution pin of the TraceHitObject message.
33. Drag in three new variable getters for the InHitActor, InHit, and InHitComponent variables, and call the TraceHitComponent interface message on InHitActor, connecting the other two variables to the remaining input pins.
34. Connect the output execution pin of the TraceHitObject message to the input execution pin of the TraceHitComponent message (see [Figure 4.15](#)); this ensures that you call hit on both the new object and the new Component when hitting a new object.
35. Create two new variable setters for the FocusedObject and FocusedComponent variables, connect them together, and connect the output of the TraceHitComponent message to the input execution pin of the first setter (see [Figure 4.16](#)).

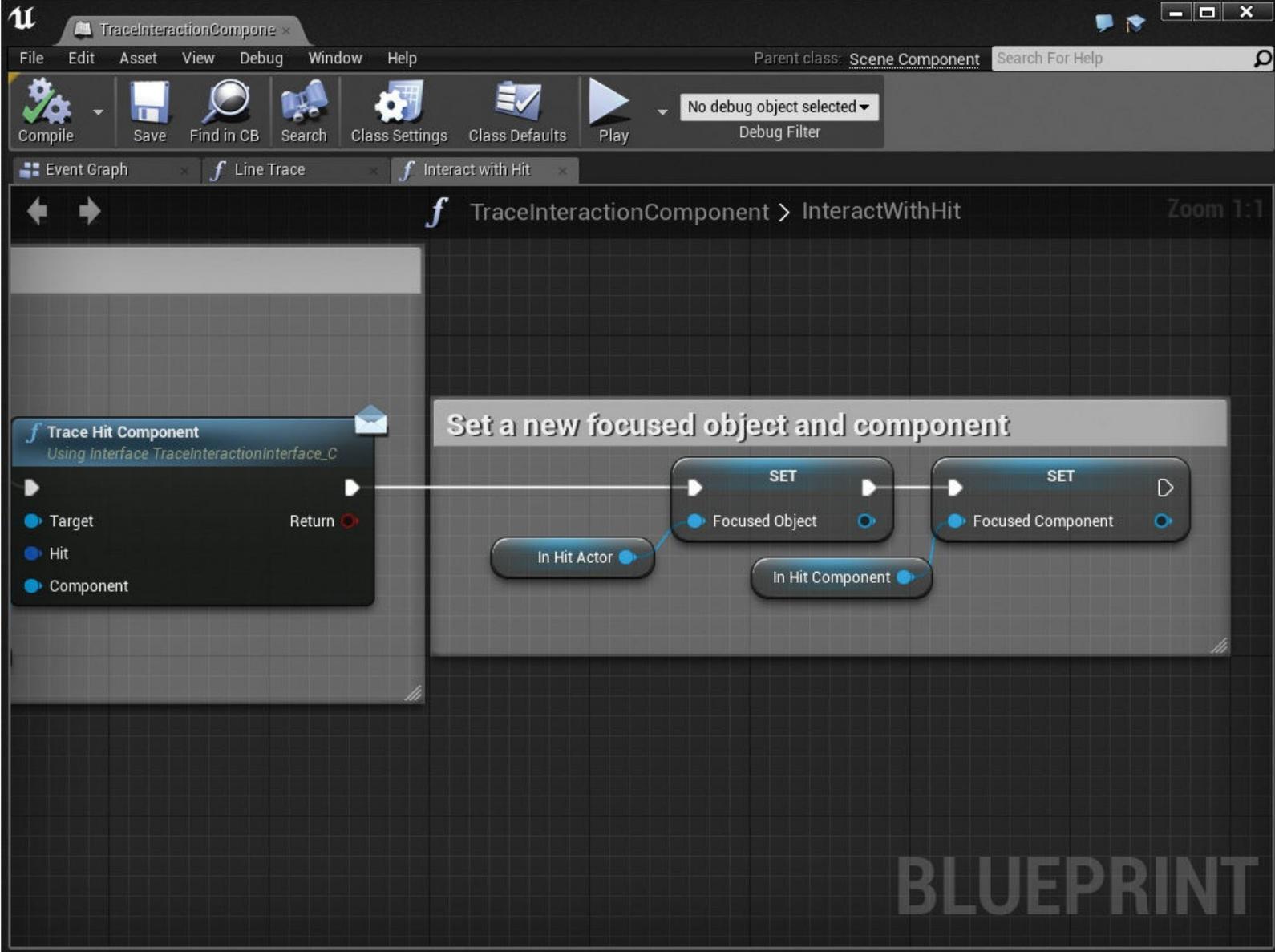


Figure 4.16 Trace Interaction Component interaction with Hit: setting the new FocusedObject

36. Create two new getters for the InHitActor and InHitComponent variables, and connect them to the inputs of the FocusedObject and FocusedComponent setters, respectively. This ensures that the FocusedObject and FocusedComponent variables are saved for the next time you call this function.

Putting the Functions Together

Now that you have created the functions you need for the basic trace interaction, you can put them together in the Event Graph and actually create a functioning Interaction Component:

1. In the Event Graph, find the EventTick where you set up the initial gate for controlling the Interaction Component.
2. Drag off of the output of the gate and call the LineTrace function, passing 1000 into the Distance input; this ensures that the trace will extend for 10m.
3. Drag off of the Hit output variable of the LineTrace function and create a new setter for the CurrentHit variable.
4. Drag off of the output of this setter and call the InteractWithHit function (see [Figure 4.17](#)). This creates a line trace every frame that the Component is enabled and then calls the appropriate interface messages to interact with that hit.

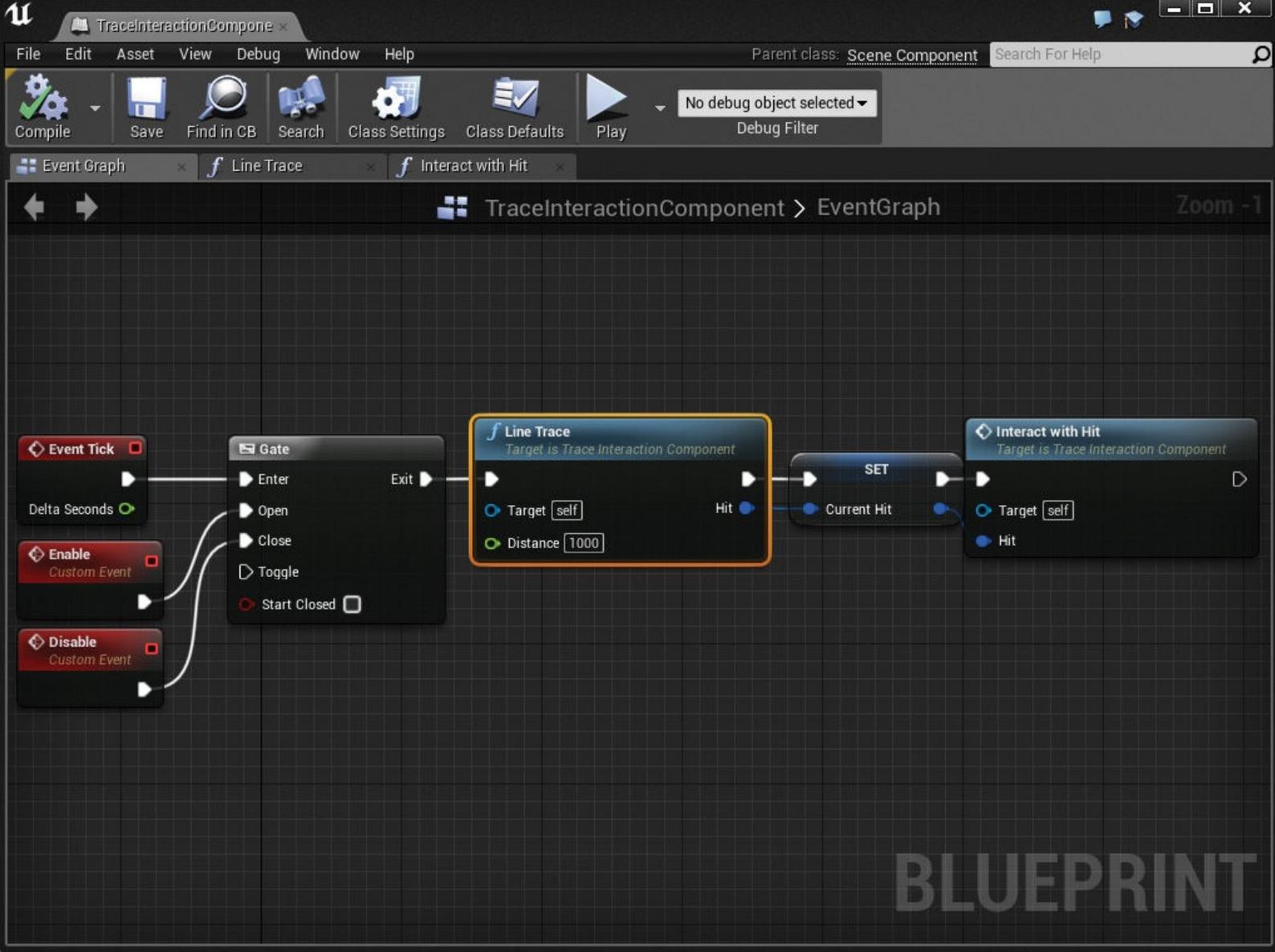


Figure 4.17 Trace Interaction Component: calling the LineTrace and Interact functions

5. The last thing you need to worry about is calling the TraceActivateDown and Up interface messages when needed. To do this, create two new custom events named ActivateDown and ActivateUp, giving both an input variable named Instigator of type Pawn.
6. Create two sets of two variable getters for the FocusedObject and CurrentHit variables and place them under the two custom events just created (see [Figure 4.18](#)).

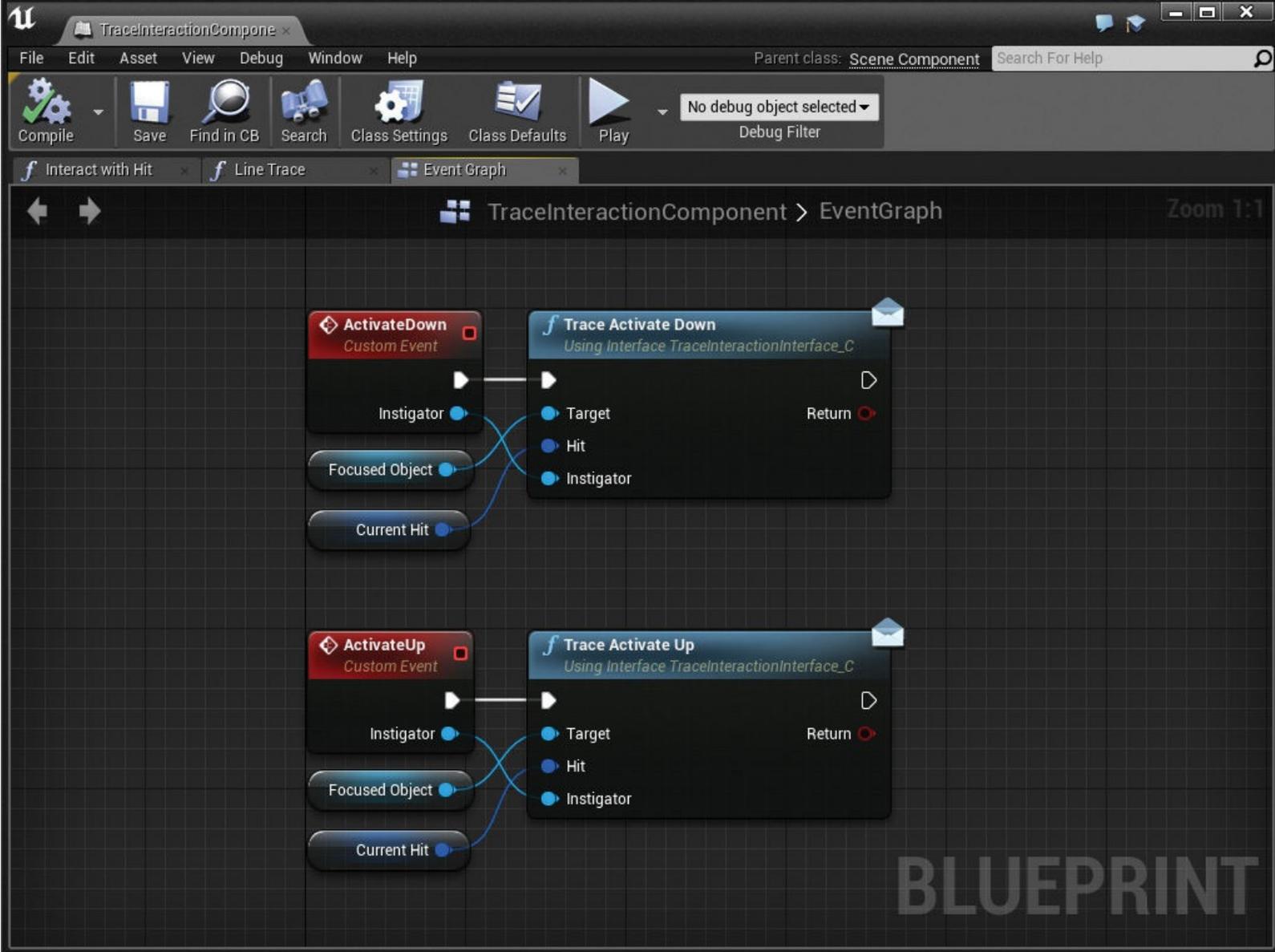


Figure 4.18 Trace Interaction Component: adding the activate events

7. On the FocusedObject variable under the ActivateDown event, call the TraceActivateDown interface message and connect the CurrentHit getter to the Hit input while also connecting the output execution pin of the event to the input execution pin of the message node and the output Instigator pin to the input Instigator pin.
8. Do the same thing for the ActivateUp event, but this time call TraceActivateUp instead (see [Figure 4.18](#)). You can now call these events from another Blueprint to activate the current FocusedObject.

Interaction Pawn Setup

Now that you have the Component and interface needed to set up an interaction system, it's time to add the Components needed for a basic VR Trace Interaction Pawn to your TraceInteractionPawn.

1. Open the TraceInteractionPawn Blueprint.
2. Add three new Components: a Scene, Camera, and TraceInteraction.
3. Name the Scene Component CameraRoot and attach the Camera to it by dragging the Camera Component over the top of the Scene Component and letting go.
4. Because you want the trace interaction to come from your player's look direction, attach the TraceInteraction Component to the Camera Component (see [Figure 4.19](#)). If you want the interaction to come from a motion controller, you can simply attach this Component to the controller instead.

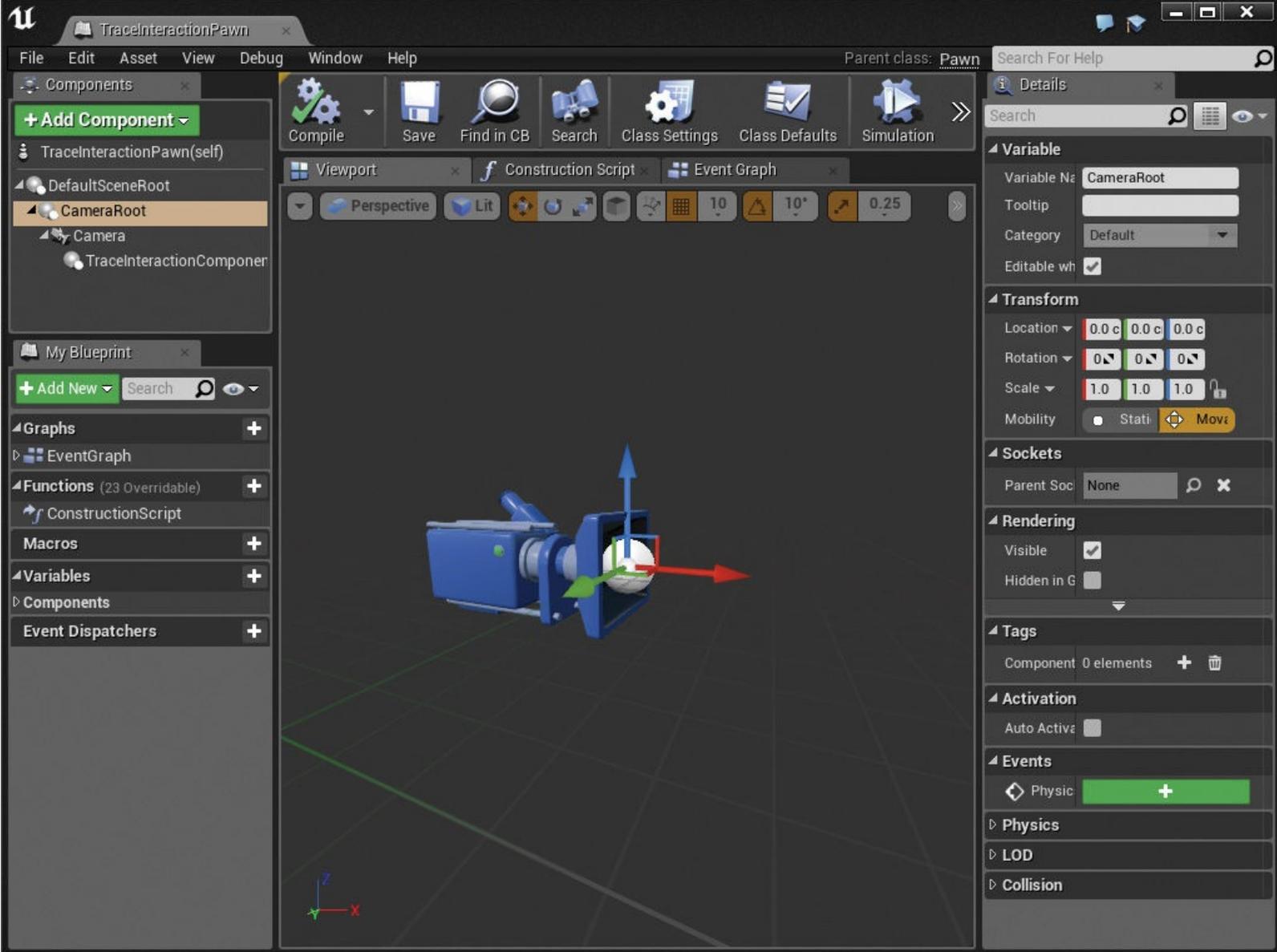


Figure 4.19 TraceInteractionPawn setup

5. The last thing needed to set up the Interaction Component is to call the ActivateDown and Up functions when needed. For this example, you use a simple LeftMouseButton node to activate the Interaction Component. Create a new LeftMouseButton event in the Event Graph.
6. Drag the TraceInteraction Component into the Event Graph and call the ActivateDown and ActivateUp events on the Pressed and Released outputs of the LeftMouseButton event, passing in Self as the Instigator (see [Figure 4.20](#)).

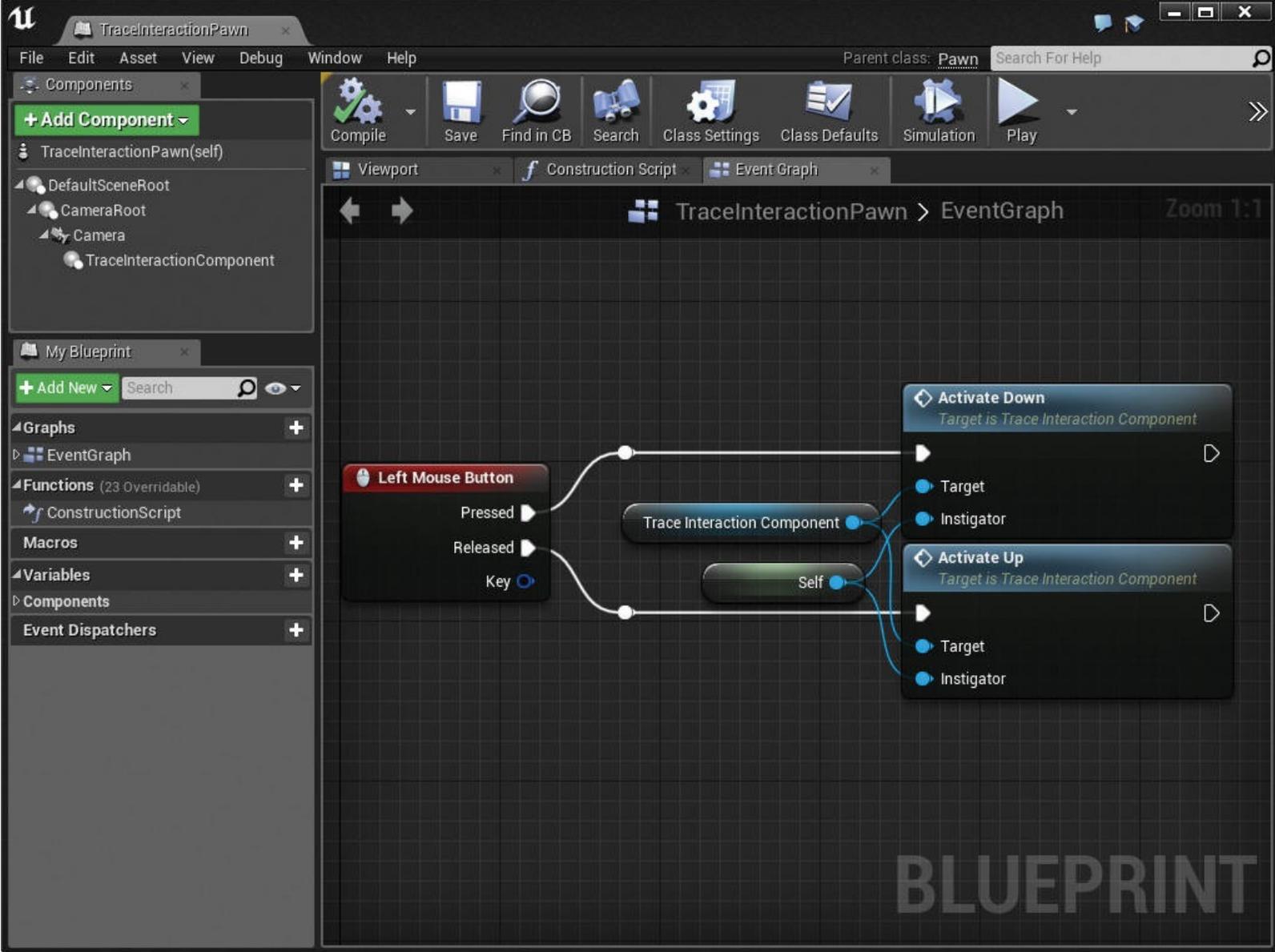


Figure 4.20 TraceInteractionPawn interaction activation

Setting Up a Basic Interactive Object

Now that the interaction system is set up, it is time to test its functionality on a basic example.

For this, we create a simple cube Blueprint that shows where the user is looking and reacts to the input by changing its color:

1. Create a new Actor Blueprint in the Blueprints folder named TraceInteractiveCube.
2. Open this Blueprint and create two new Components: the first a Cube Component and the second an Arrow Component.
3. Select the Arrow Component and set the Hidden in Game attribute to False (see [Figure 4.21](#)). This allows you to see the arrow in-game; this is the marker you will use to see where the player is looking at the cube.

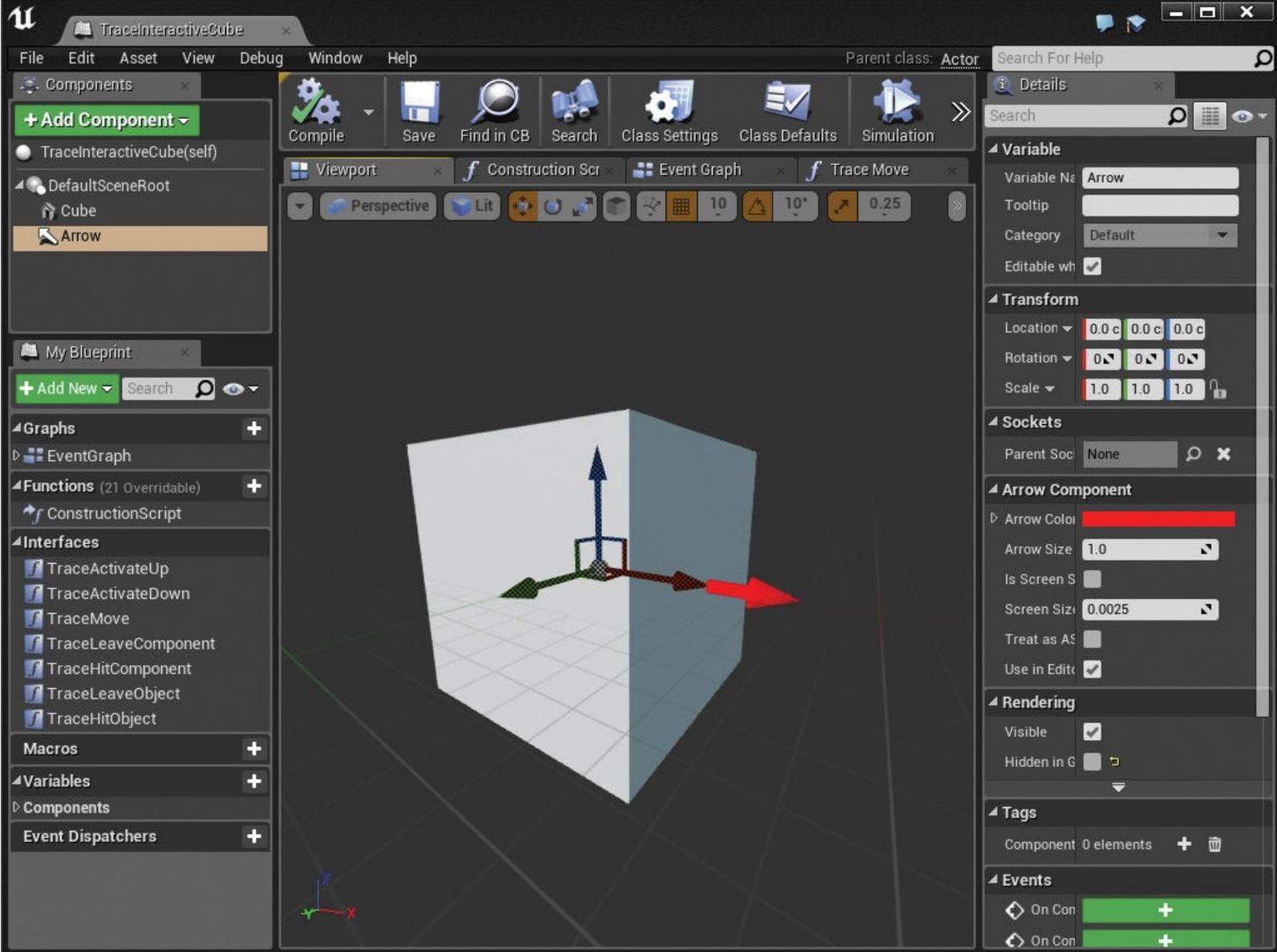


Figure 4.21 Trace Interactive Cube: Component setup

4. Click the Class Settings button in the top shelf of the Blueprint. Then, under the Interfaces tab in the Details panel, select the Add button and select your TraceInteractionInterface. After compiling the Blueprint, you will see the interface functions under the Interfaces section of the My Blueprint panel.
5. Open the TraceHitObject interface function and drag in a variable getter for the Cube Component.
6. Call the SetVectorParameterValueOnMaterials function on this Component, passing in the vector ($X = 1.0, Y = 0.0, Z = 0.0$). This turns the object red when a trace hits the object.
7. Attach the SetVectorParameterValueOnMaterials node between the input node and Return node of the function (see [Figure 4.22](#)).

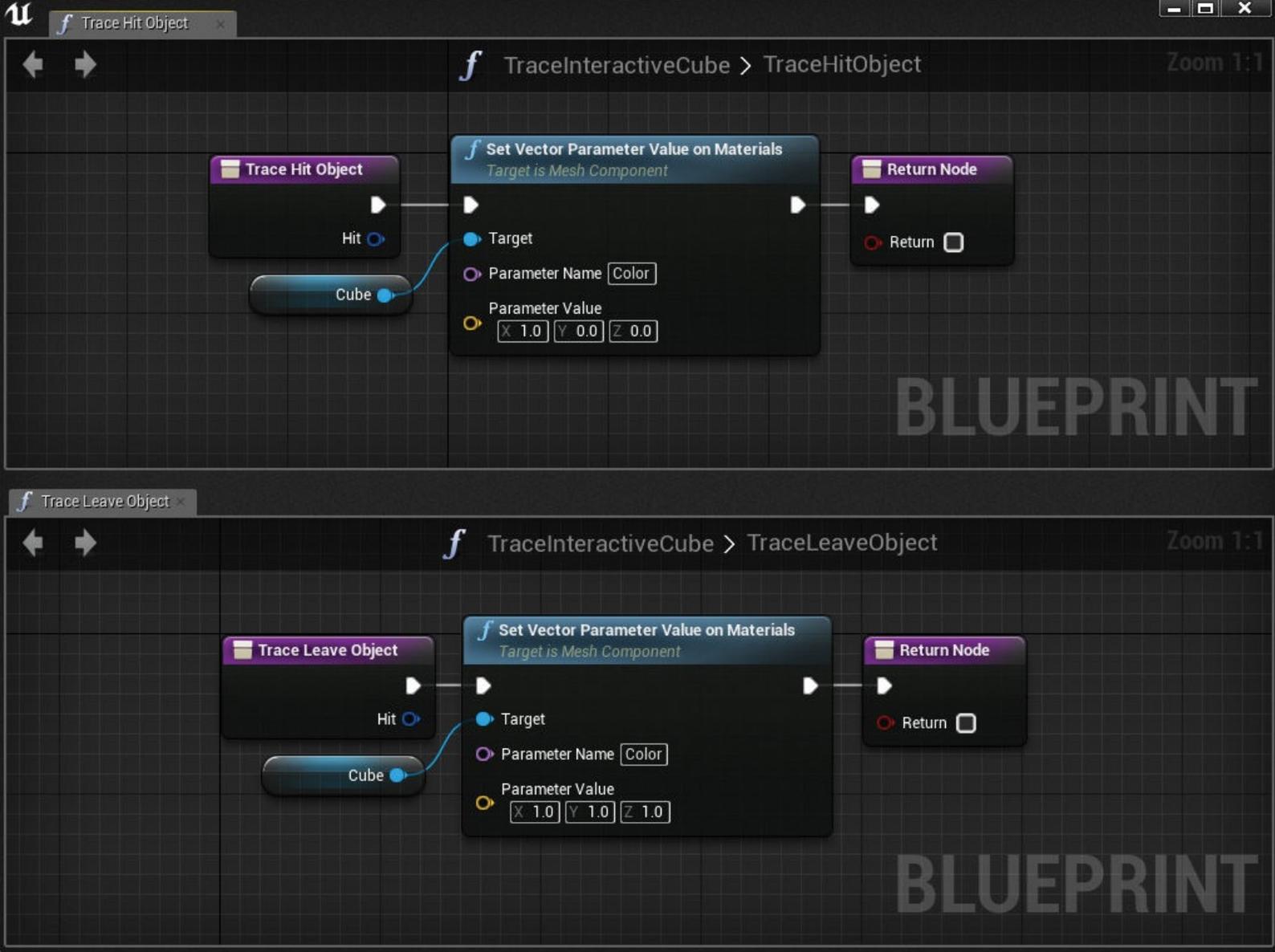


Figure 4.22 Trace Interactive Cube: hit and leave color change

8. Repeat steps 5 through 7 for the TraceLeaveObject function, but this time passing in the vector (X = 1.0, Y = 1.0, Z = 1.0) to the input of the SetVectorParameterValueOnMaterials node (see [Figure 4.22](#)). This sets the cube's color back to white when the player is no longer looking at it.
9. Repeat steps 5 through 7 again for the TraceActivateDown and TraceActivateUp functions, passing in the vectors (X = 0.0, Y = 1.0, Z = 0.0) and (X = 1.0, Y = 0.0, Z = 0.0), respectively, to the SetVectorParameterValueOnMaterials (see [Figure 4.23](#)). This sets the color of the cube to green when it is activated and back to red when deactivated.

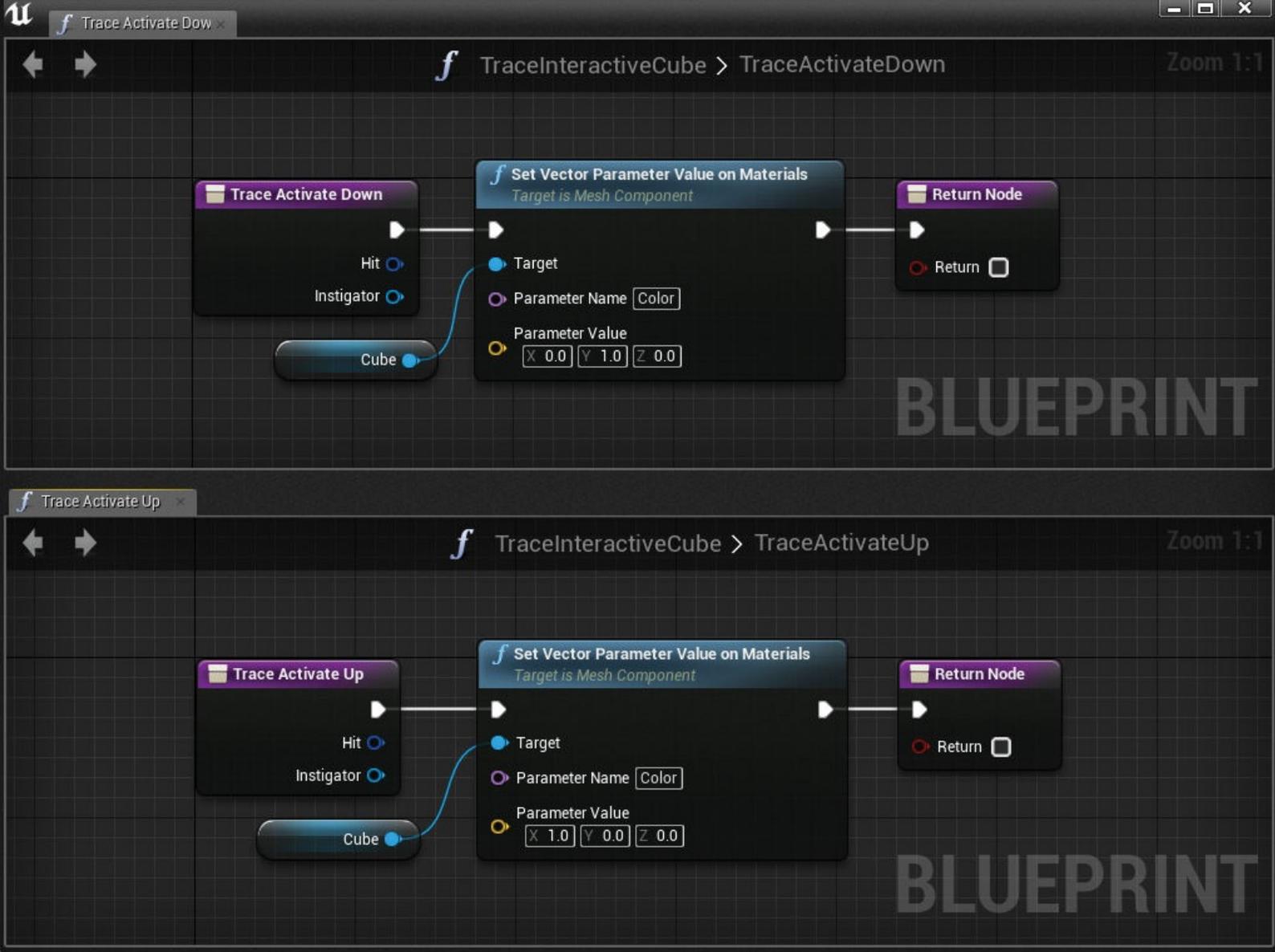


Figure 4.23 Trace Interactive Cube: activate up and down color change

10. To move the arrow to the position of the hit in every frame when the player is looking at the cube, open the TraceMove function.
11. Drag off of the Hit pin of the input node and break the Hit Result.
12. Create a new variable getter for the Arrow Component, then call the SetWorldLocationAndRotation function on it, connecting the input and Return nodes of the function.
13. Directly connect the Location output pin of the BreakHitResult to the New Location pin of SetWorldLocationAndRotation.
14. Dragging off of the Normal pin of BreakHitResult, call the MakeRotFromX function and connect its output to the New Rotation of SetWorldLocationAndRotation (see [Figure 4.24](#)). Because the Arrow Component points in the X direction, this will rotate the arrow so it faces outward from the hit normal.

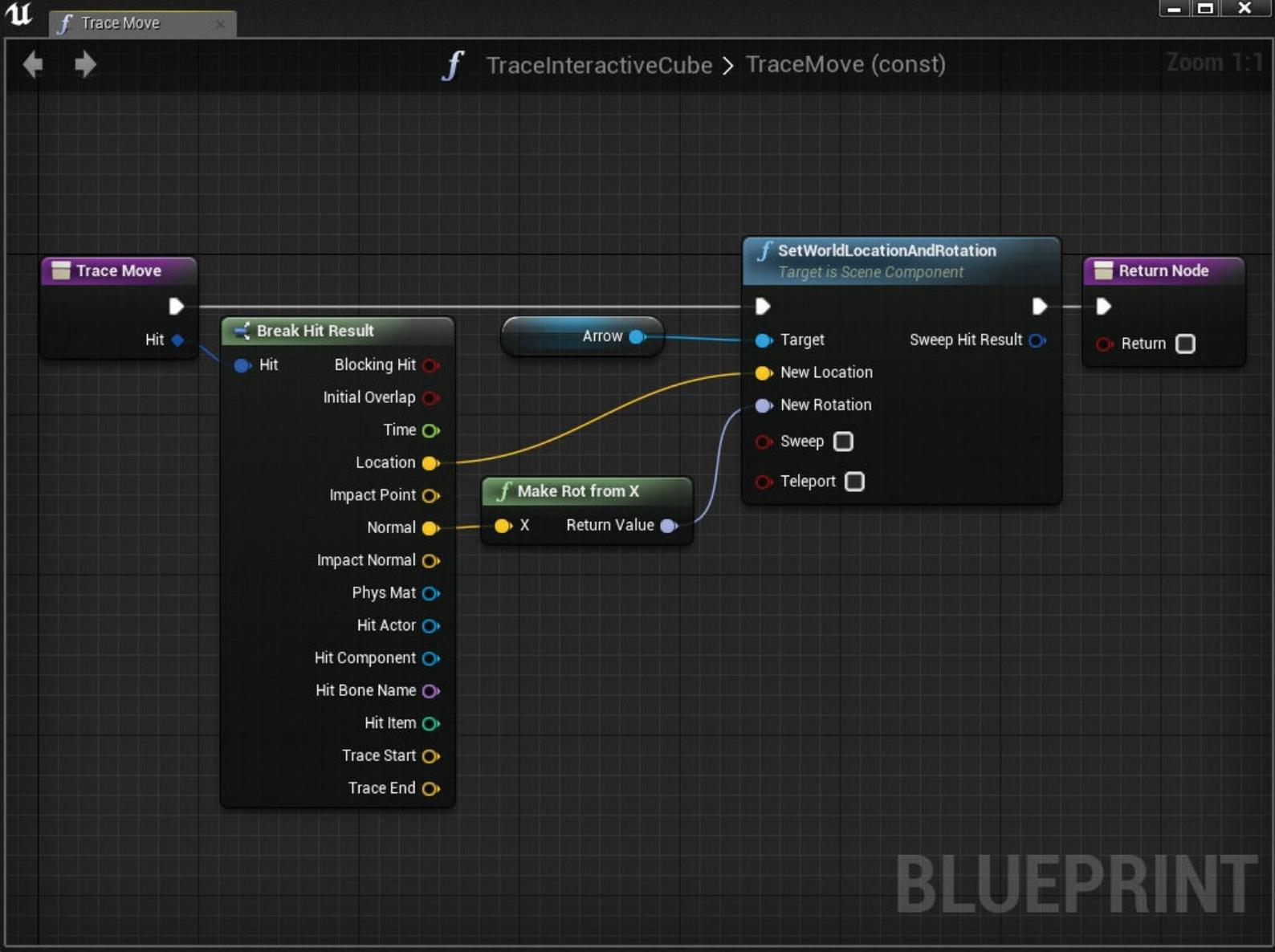


Figure 4.24 Trace Interactive Cube: TraceMove

Now with everything set up you should be able to add both your Pawn (by either creating a custom game mode and telling it to use your Pawn or by putting your Pawn in the level and enabling Auto Possess) and your interactive cube to the level and enjoy the Christmas spirit of a red and green cube.

Summary

This chapter discussed how traces work in UE4 and how to use them in VR games. You looked at interfaces and why they are a perfect fit for communicating between an interactor and interactive objects. You created a modular system for trace interaction that can be transferred and expanded upon in future projects.

Exercises

In this chapter, you created an interaction system that relies on a line trace to find objects with which to interact; this is good enough for a lot of cases, but it has its downsides. Mainly, when using a single line trace, it can be hard for users to interact exactly with the objects they want. This is especially noticeable at longer distances. Therefore, it may be useful to use some other metric to estimate where the user is looking and account for that in your interaction system. One common method for this is to use a little bit of vector math to find the difference between the user's

look vector and the vector from the player to the interactive object.

To do this, you can use the dot product, which is a form of multiplying two vectors together to return a single number. The dot product is calculated by taking each pair of numbers in the two vectors, multiplying them together, then taking the sum of all of those new numbers. It may sound a little complicated at first, but luckily UE4 takes care of the implementation and all you need to know is what the final number after the dot product represents.

For this example, the inputs to the dot product are normalized—that is, they have a length of 1. This basically means that both vectors represent directions and not actual locations or distances. Because the inputs have a length of 1, the dot product of these two vectors has the nice property that the end output is the cosine of the angle between them (see [Figure 4.25](#)). So, if you calculate the dot product between where the user is looking and the direction that some interactive objects are relative to the user’s head, you can get a value between -1 and 1 that represents how closely the user is looking at an object (with 1 being looking directly at and -1 being looking directly away from the object).

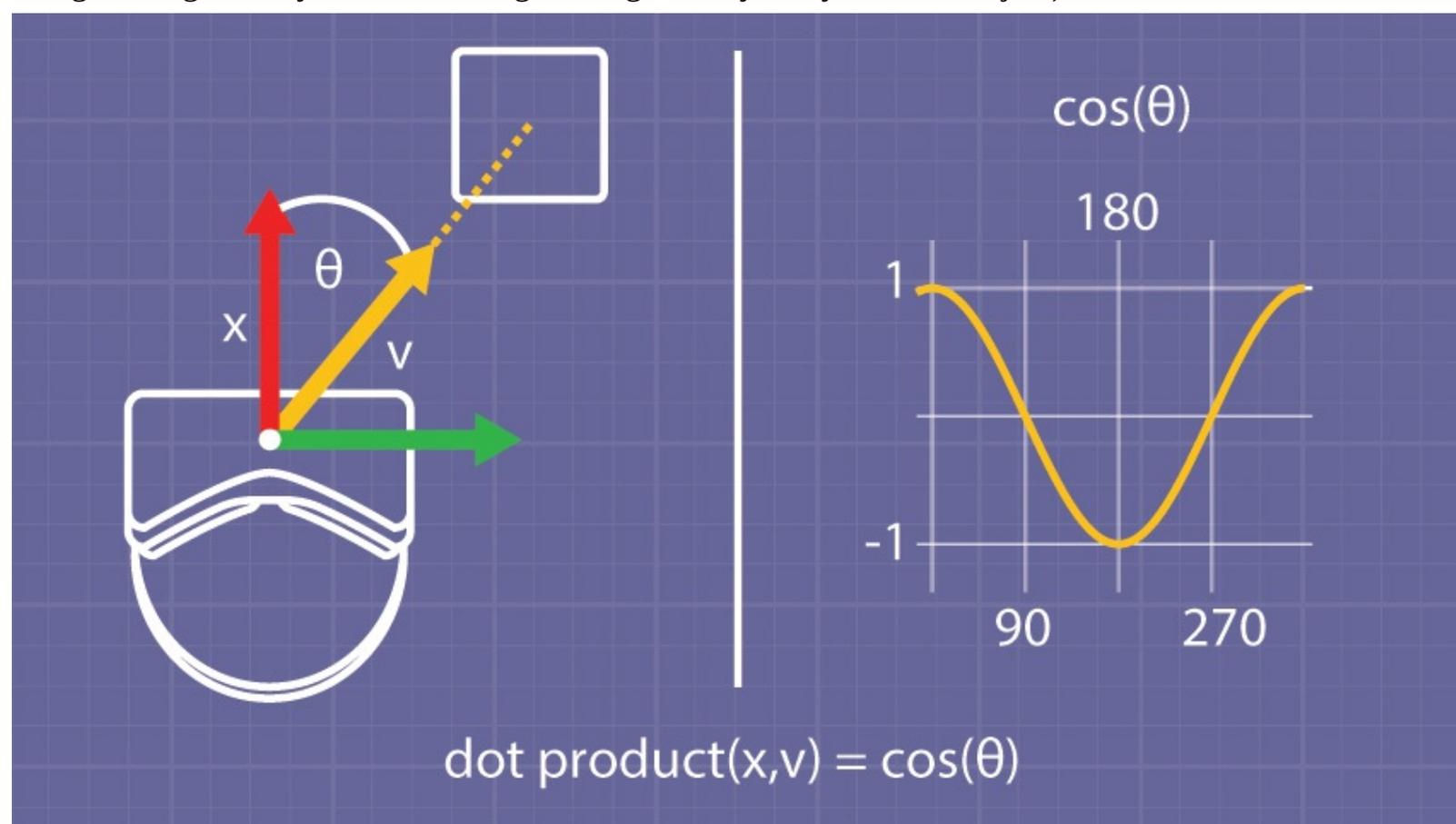


Figure 4.25 Dot product example. Both x and v are normalized vectors where x represents the player’s look direction and v represents the direction from the player’s head to the object. The angle θ represents the angle between x and v in degrees.

This happens to be a simple way to judge the intent of the user’s look and can be used if you need any kind of “aim assist” functionality in your game. To implement this, you can simply loop over all the objects you want to assist toward and find the one with the highest dot product value (or the object that the user is looking closest to).

One thing to look out for when using the dot product to find the user’s look intent is that normalizing these values takes any sense of distance out of the equation. The side effect is that if one object is close to the user and another object is far away, but the farther-away object has a shorter angle to the user’s look direction, it may be selected even if the user is trying to select the closer one.

One way to deal with this is instead of maximizing the dot product between the user and the interactive

objects you can minimize the arc length between the user and the objects. That is the distance that is created by the circle that is made by the user's look direction and the object's position (see [Figure 4.26](#)).

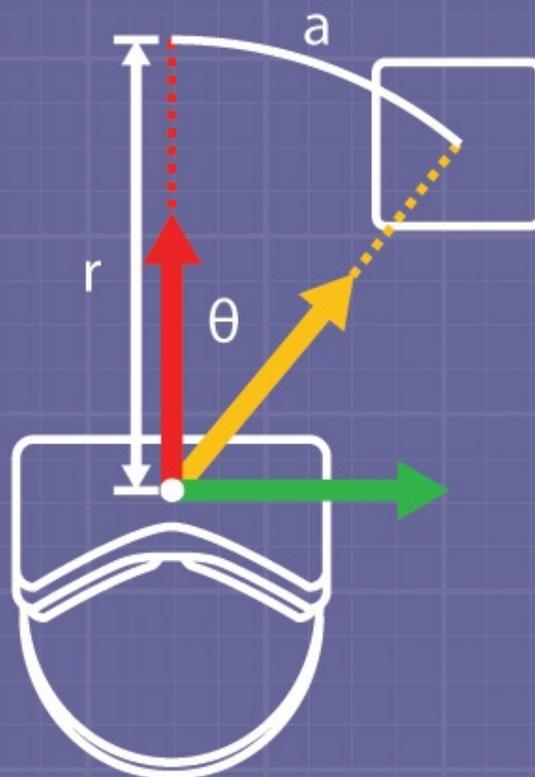


Figure 4.26 Arc length example. The arc length a is calculated by multiplying the radius r and the angle θ in radians.

[Figure 4.27](#) shows an example of how to calculate this and the dot product in UE4.

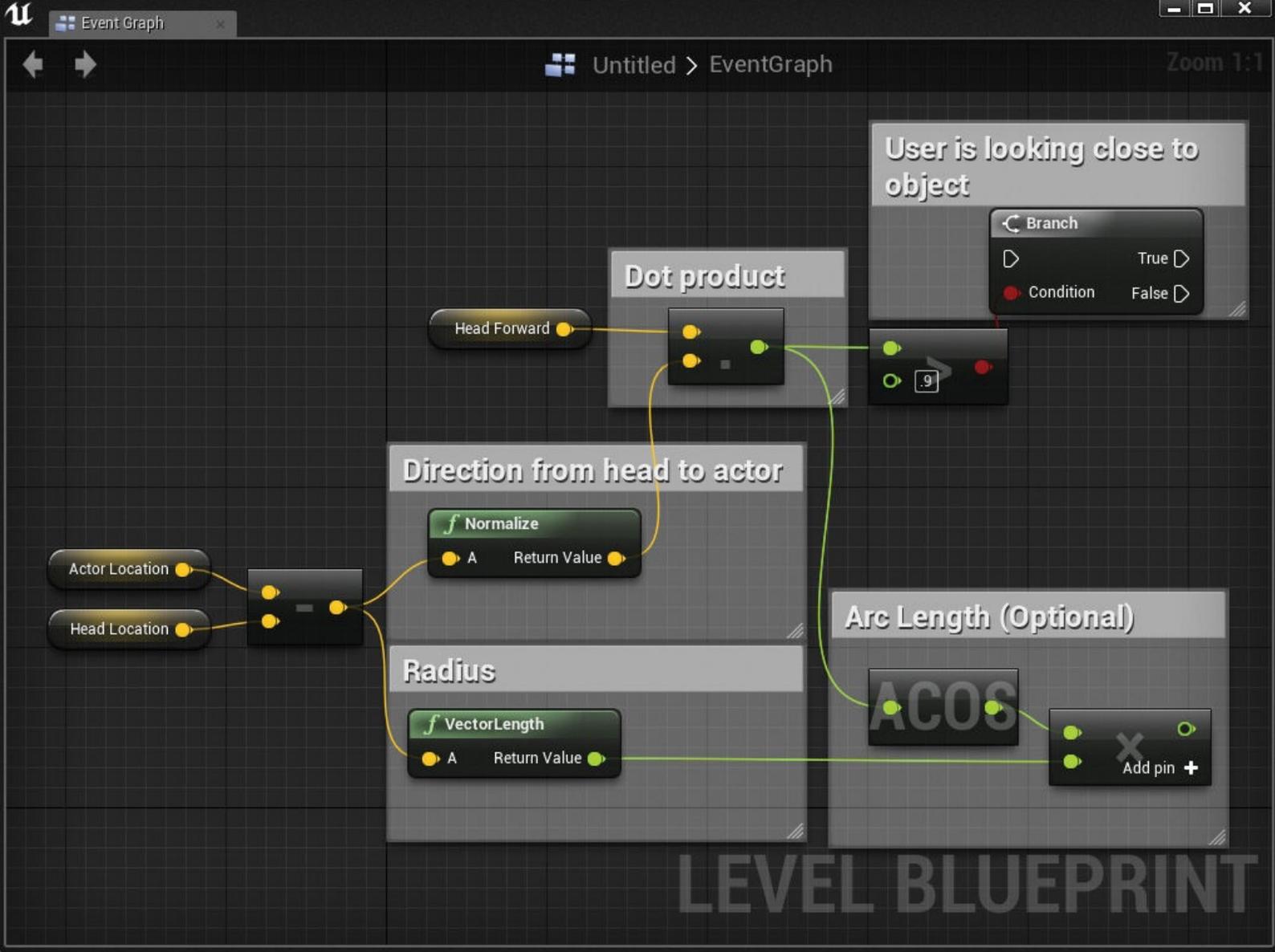


Figure 4.27 Calculating the dot product between the Actor location and head forward with an optional calculation of the arc length made between the Actor and the head forward

One downside of using a dot product method entirely to detect user interaction is that, by default, it won't handle collisions; this can lead to users being able to interact with objects through walls and other solid surfaces. One solution for this is to simply check the collision between the user and the objects you are testing by tracing to them. This, however, has the potential to be computationally expensive if there are many interactive objects in your scene because you would need a trace for every interactive object.

Another way to handle this problem is to simply make sure there is nothing that can obscure your interactive objects. This obviously can be guaranteed only in menus and other controlled scenarios; however, when this is the case, a dot product method gives you a fairly cheap and user-friendly way to detect interactions in your game.

If you want the benefits of the collision detection that line traces give you, but want more flexibility in the way elements are selected and you are able to spend a little more performance on your traces, using one of the shape traces can be a good idea. Creating a sphere trace with a radius greater than 1 will add a nice level of aim assist while still remaining relatively cheap to compute. To see an example of creating a sphere trace function that can be used in the Trace Interaction Component, see [Figure 4.28](#).

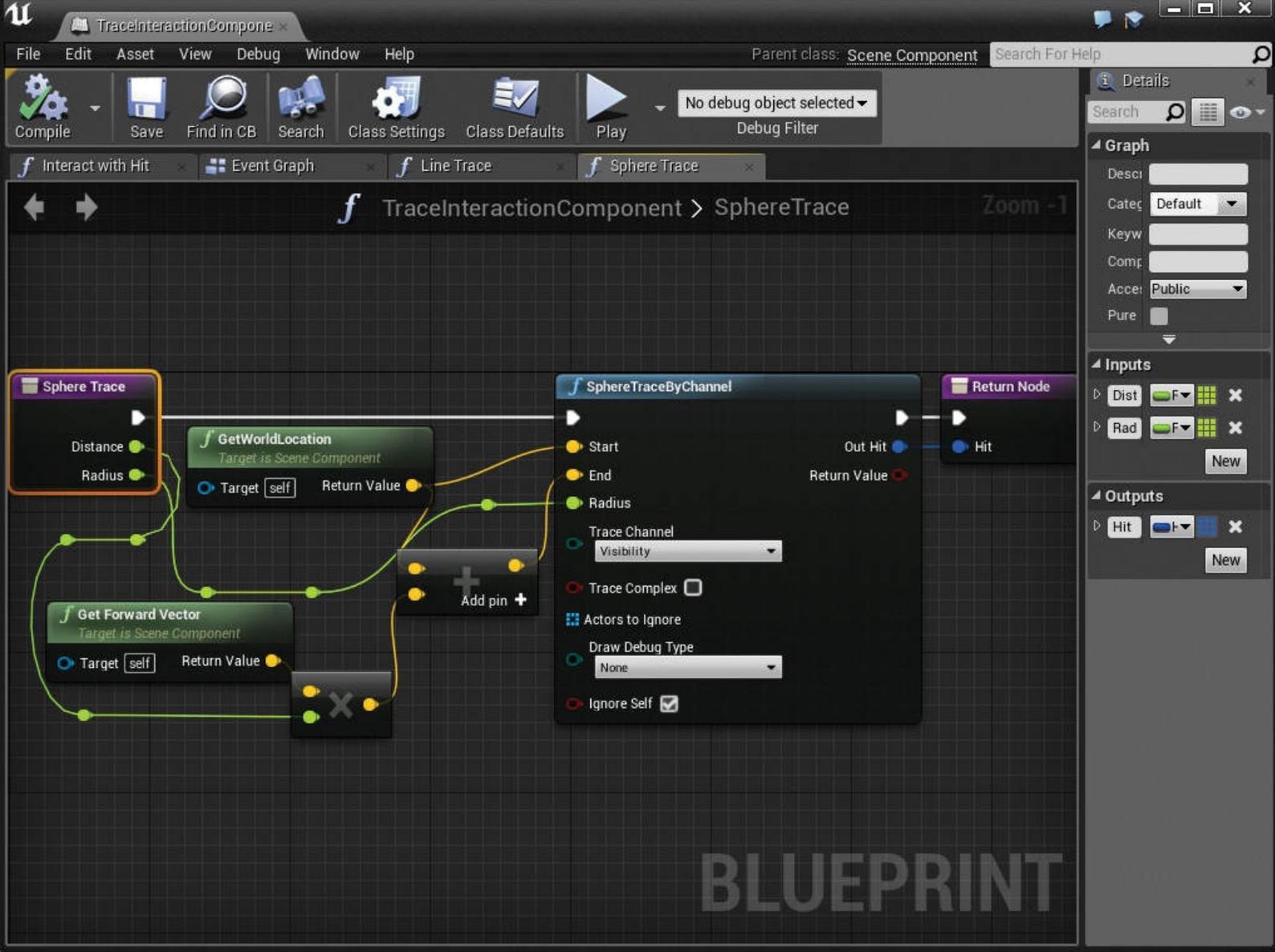


Figure 4.28 Simple sphere trace function for the Trace Interaction Component

As an exercise, add the sphere trace to your Trace Interaction Component and see if it also helps with usability.

As another exercise, add some aim assist using the dot product. To do this, you need a list of all the objects you want to interact with. Think about what the best way to get this list may be.

Chapter 5. Teleportation

Teleportation is one of the leading methods for artificial locomotion in the current state of VR. This chapter expands the trace interaction from [Chapter 4](#) into a simple teleportation mechanism.

The best form of locomotion in VR is undoubtedly to have no artificial locomotion. Instead, rely on a one-to-one mapping of real-world actions to your VR characters. However, this creates an incredibly limiting set of requirements, and thus it isn't possible for many games or even whole genres to rely on this method.

This need to have virtual players move farther than their physical bounds brings with it a host of challenges and design decisions that need to be made. The need for and reasoning behind those decisions are explored in [Chapter 9](#), “[VR Locomotion](#)”; however, this chapter introduces you to one of the more popular solutions to the locomotion problem: teleportation.

To avoid the ocular vestibular mismatch that can occur when players see their in-game character moving when they are, in fact, stationary in the real world, many developers rely on a teleportation system that moves the player from the current location to the desired location in an instant.

This single burst has proven to produce less sickness in the majority of VR users and so can be found in many VR games today. It does, however, have its downsides, mainly in the form of disorientation. As humans, we can't teleport (yet), and so in a VR game the user can be discombobulated for a moment when coming out of a teleport. There are various potential remedies for this, ranging from the simple, such as showing a trail from where the user came, to the more complicated, such as showing users a portal into the teleport location before they trigger the teleport. However, this chapter does not focus on these techniques. Instead, it focuses on getting a basic teleportation system functioning, leaving the (fun) experimentation up to the reader.

Setting Up Teleportation

To detect the teleportation, you will use traces; luckily for you, you have already set up a modular VR Trace Interaction Component and interface that can be adapted to fit your needs. If you haven't, review [Chapter 4](#), “[Trace Interaction](#),” which walks you through all the relevant information on creating a modular system for detecting trace interactions in your worlds.

Now that you have the base for a Trace Interaction level (from either completing the previous chapter or downloading its source code), it is time to extend it to accommodate teleportation.

Parabolic Tracing

When you open the Trace Interaction Component, notice that currently the only trace function creates a line trace emanating out of the Component. This is nice and would work perfectly well for teleportation; however, to take it a step further and allow the player to teleport vertically as well as horizontally, you will implement a parabolic trace method.

In this instance, a parabolic trace is based on the idea of projectile motion and can be thought of as shooting a ball from the Trace Interaction Component and following its path until it hits an object in the game. To accomplish this, you take the direction of the Interaction Component to be the direction in which the ball was fired and calculate a new position on a fixed time step, tracing between the previous location and this new position until the trace hits an object where it will stop and return the object that was hit.

To calculate this projectile arc, you use some basic physics formulas, which are shown in [Figure 5.1](#).

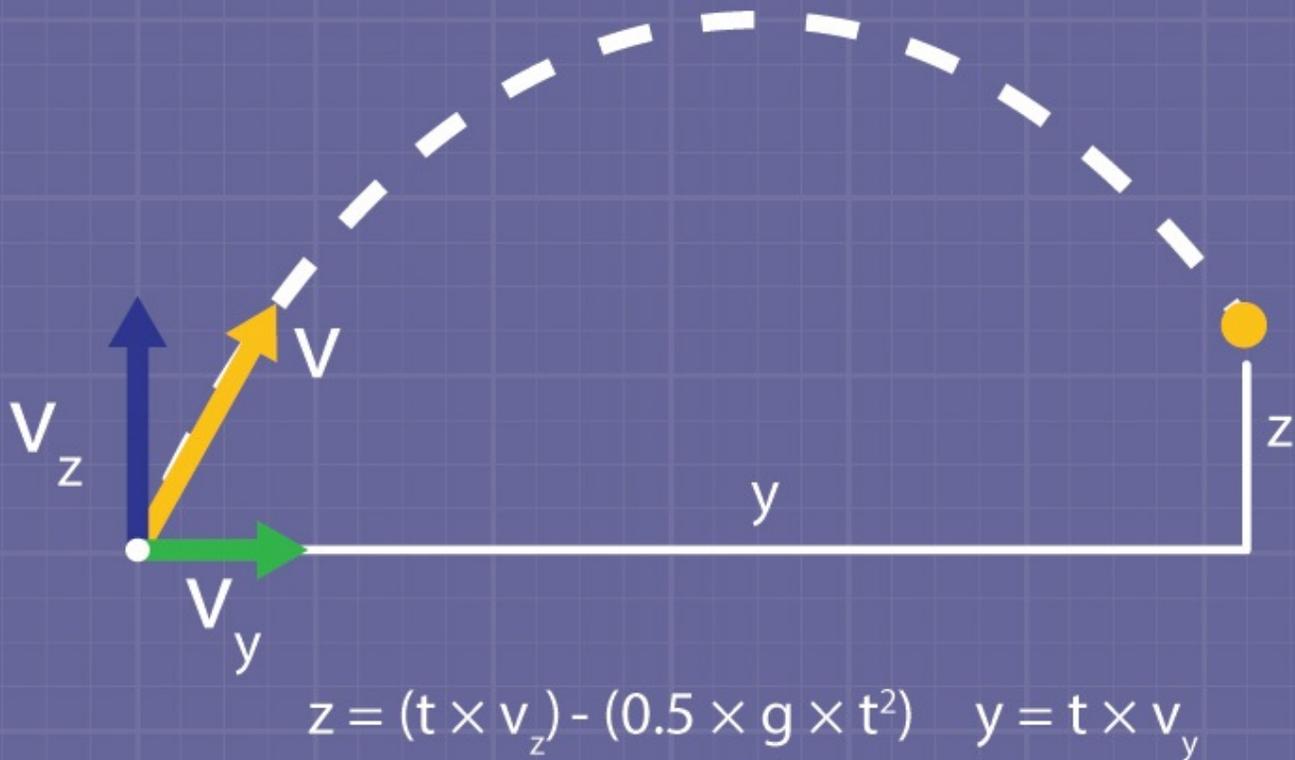


Figure 5.1 The projectile motion calculation used for parabolic traces. v represents the velocity of the trace, y represents the location of the trace in the Y-axis, and z represents the location of the trace in the Z-axis in relation to the time t and gravity g .

Now that you have an idea of how the trace function works, let's look at setting one up:

1. With your project open from [Chapter 4](#), open the Trace Interaction Component Blueprint from Blueprints → Components.
2. Create a new function named ParabolicTrace.
3. Add three input variables to the function named Steps, TimeStep, and Speed with types Integer, Float, and Float, respectively.
4. Create an output variable named Hit of type Hit Result.
5. Create three local variables named InitialLocation, PreviousLocation, and Velocity, all of type Vector.
6. Create two more local variables named InTimeStep and InSteps of types Float and Integer.
7. Create one last local variable named TmpHit. This will hold the temporary hit as you calculate the traces.
8. Create a variable setter for all these local variables, except for TmpHit; this will be set later.
9. Connect the output execution pin of the function input to the InitialLocation setter followed by the PreviousLocation setter, making sure to connect the yellow output pin of the first setter to the yellow input pin of the second (see [Figure 5.2](#)).

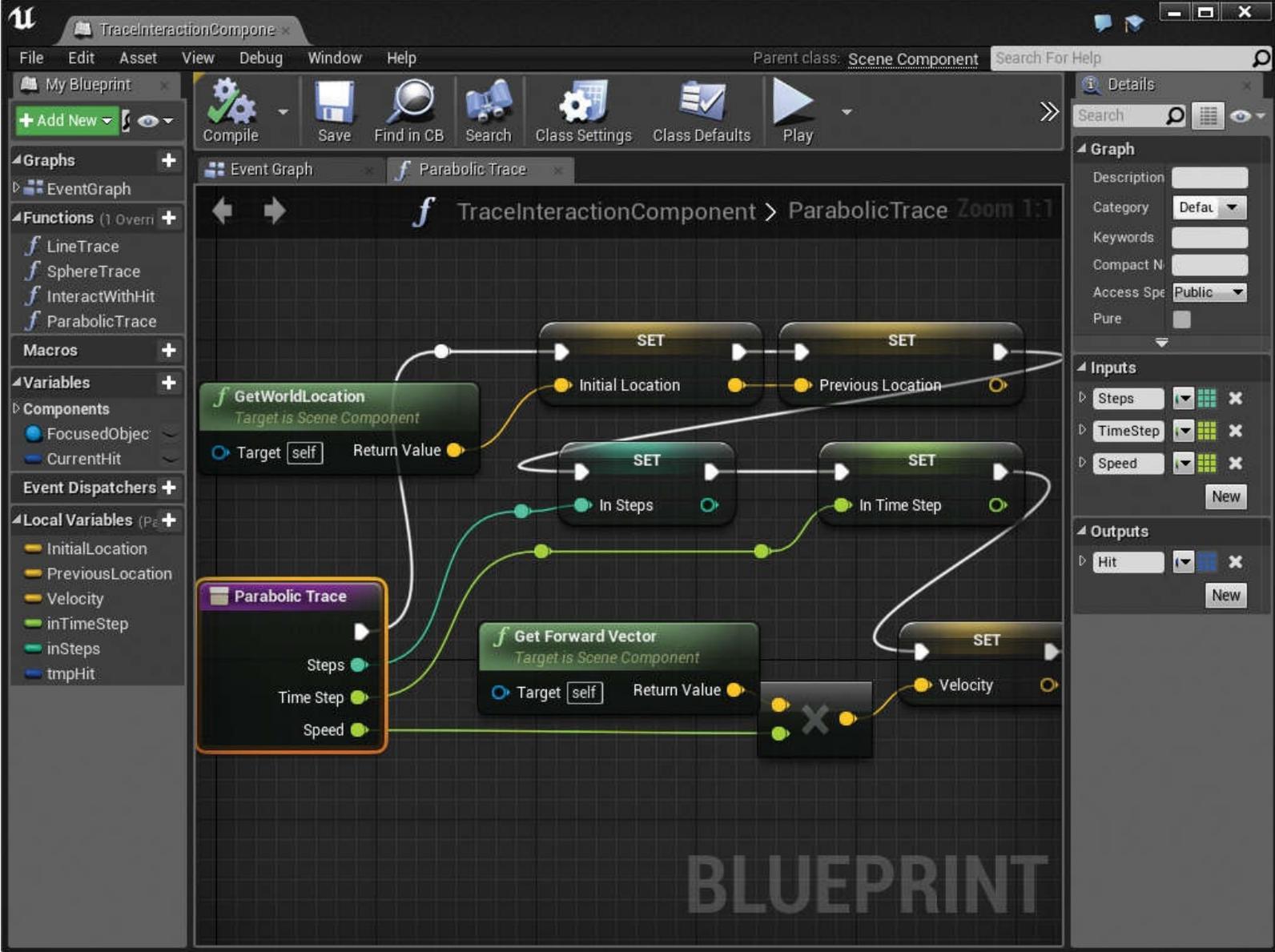


Figure 5.2 Parabolic trace function: basic setup of local variables and inputs

10. Create a `GetWorldLocation` node and connect its output to the `InitialLocation` setter.
11. Connect the `InSteps` setter and `InTimeStep` to their corresponding inputs to the function, and connect the `InSteps` output execution pin to the input execution pin for the `InTimeStep` setter (see [Figure 5.2](#)).
12. Connect the `InSteps` setter to the output execution pin of the `PreviousLocation` setter.
13. Create a `GetForwardVector` node and multiply it by the incoming `Float Speed`.
14. Attach the output of this `Multiply` to the input of the `Velocity` setter, and connect the input execution pin of the setter to the output execution pin of the `InTimeStep` setter (see [Figure 5.2](#)).

After creating the variables needed for the trace calculation, you need to calculate the parabolic arc, as shown in [Figure 5.1](#). To do this you will split the trace velocity into its separate axes and calculate the new position of the trace on a fixed time step: **1.** Drag off of the last variable setter from the function setup and create a new `ForLoopWithBreak` node.

2. For the `First Index` input put 1 and for the `Last Index` put an `InSteps` variable getter. In this example you start from the index 1 to avoid tracing to the same point twice because the previous location is initialized to the initial location.
3. Drag off of the `Loop Body` output pin and call the `LineTraceByChannel` function.
4. For the `Start` pin of this trace connect a new `PreviousLocation` variable getter.

15. Add a new InitialLocation variable getter to the second input pin of this Vector + Vector node. This turns the local calculation from the velocities into a world position.
16. Pass the output of this to the End input pin of the line trace.
17. If you want to visualize the trace, set the Draw Debug Type to For One Frame.
18. Attach the Return node of the function to the Completed pin of the loop and return the TmpHit variable. Don't worry about this too much, because you will set this variable next.

After calculating the trace, you will set the temp hit variable and break out of the loop early if a hit is detected to make sure no unnecessary calculations are made: **1.** Drag off of the LineTraceByChannel node, create a new PreviousLocation variable setter, and set the input to be the output of the Vector + Vector used for the End pin of the trace (see [Figure 5.4](#)).

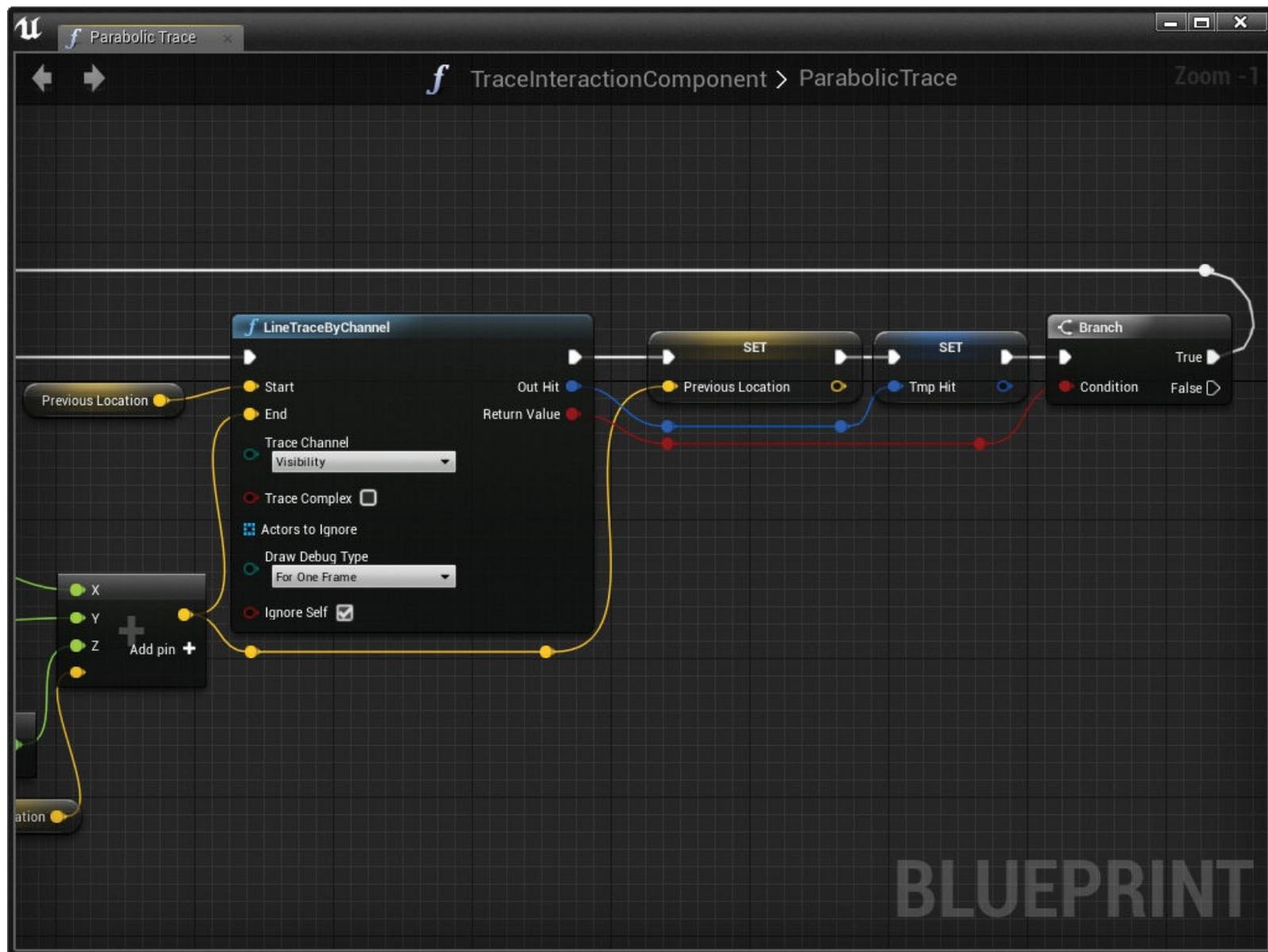


Figure 5.4 Parabolic trace: setting the temp hit variable and breaking out of the loop

2. Create a TmpHit variable setter and connect the Out Hit of the line trace to its input.
3. Create a new Branch node connecting the Return Value pin of the trace to its Condition input.
4. Connect the True output of the Branch node to the Break pin of the for loop. This ensures that when a hit is detected, the loop is broken out and the temp hit is returned.

Now that the trace function is complete, it is time to implement it in the Event Graph:

1. Moving back to the Event Graph of the Trace Interaction Component, delete the current function

call of the LineTrace function and replace it with the ParabolicTrace function.

2. For inputs to the ParabolicTrace function, a good starting point would be ten steps at a 0.1 time step and a speed of 500 (see [Figure 5.5](#)).

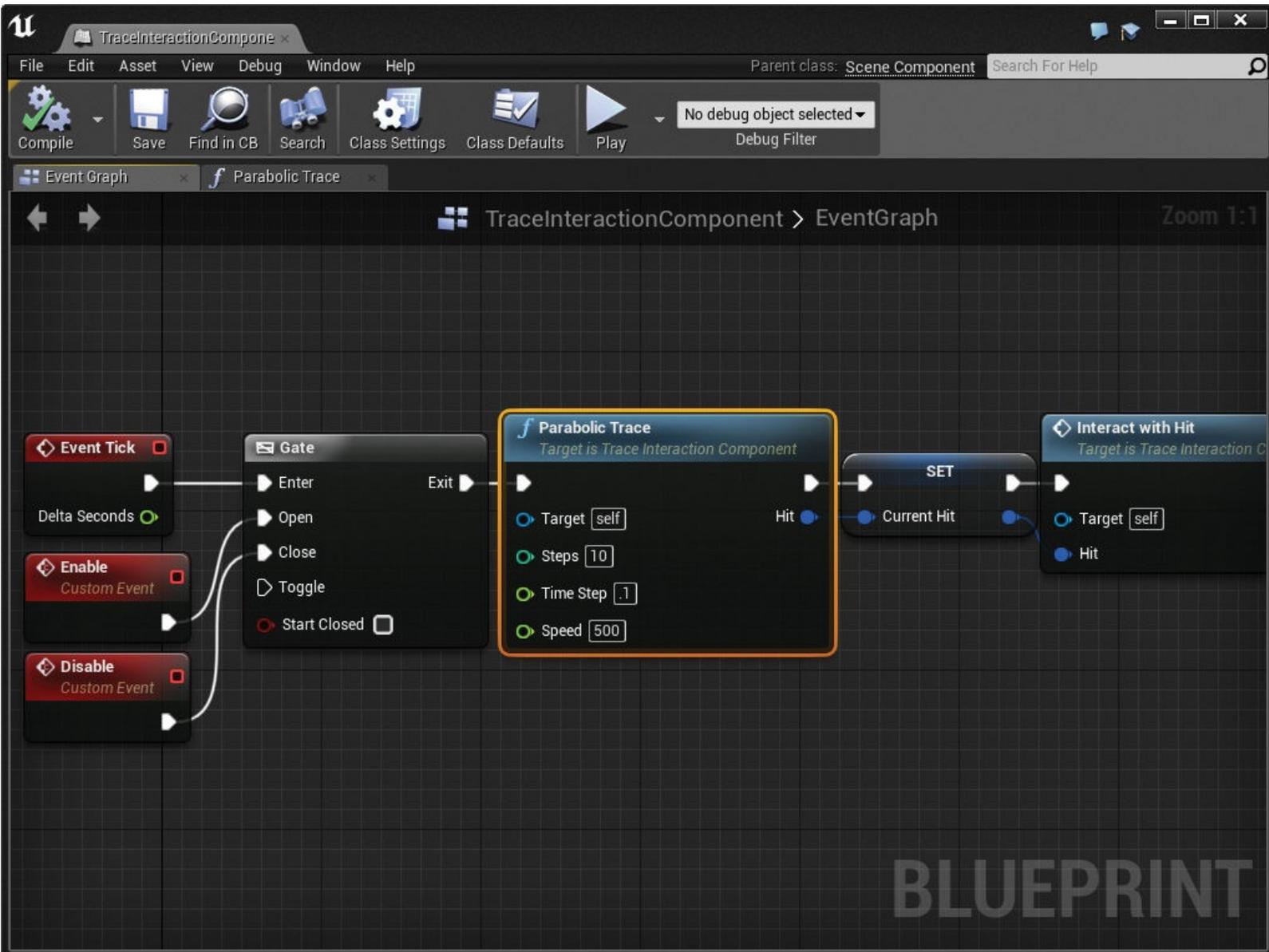


Figure 5.5 Parabolic trace: using the ParabolicTrace function in the Interaction Component

Visualizing the Teleport

When testing the teleport, you could use a simple debug shape to indicate to users where they will be teleporting; however, for this example you will create a separate Actor because it will allow you to expand it later to visualize more than just the trace location.

Visualization Material

The first step is to create the Material that will be used for the teleport visualizer. It is a simple gradient that trails off at both top and bottom.

1. If none exists, create a Materials folder in the Content Browser.
2. Create a new Material named TeleportVisualizer.
3. Open the Material, set the Blend Mode to Translucent and the Shading Model to Unlit, and check Two Sided in the Details panel.

4. Create a new VectorParameter node named Color and set its Default Value to (R = 0.25, G = 1.0, B = 0.5, A = 1.0).
5. To create the gradient, first create a new TextureCoordinate node by either searching for it or U + clicking in the graph.
6. Drag off of the TextureCoordinate and create a new Mask node, selecting the G field only. This will extract just the Y gradient from the TextureCoordinate.
7. To get the gradient at both ends, drag off of the Mask node and create a new Cosine node. This will take the input of a gradient of 0 to 1 to make it a 1 to -1 to 1 gradient.
8. To transform this gradient to all positive values, add a new OneMinus node after the Cosine node. The values now form a gradient of 0 to 2 back to 0.
9. To normalize these values, simply divide the OneMinus by 2 (see [Figure 5.6](#)).

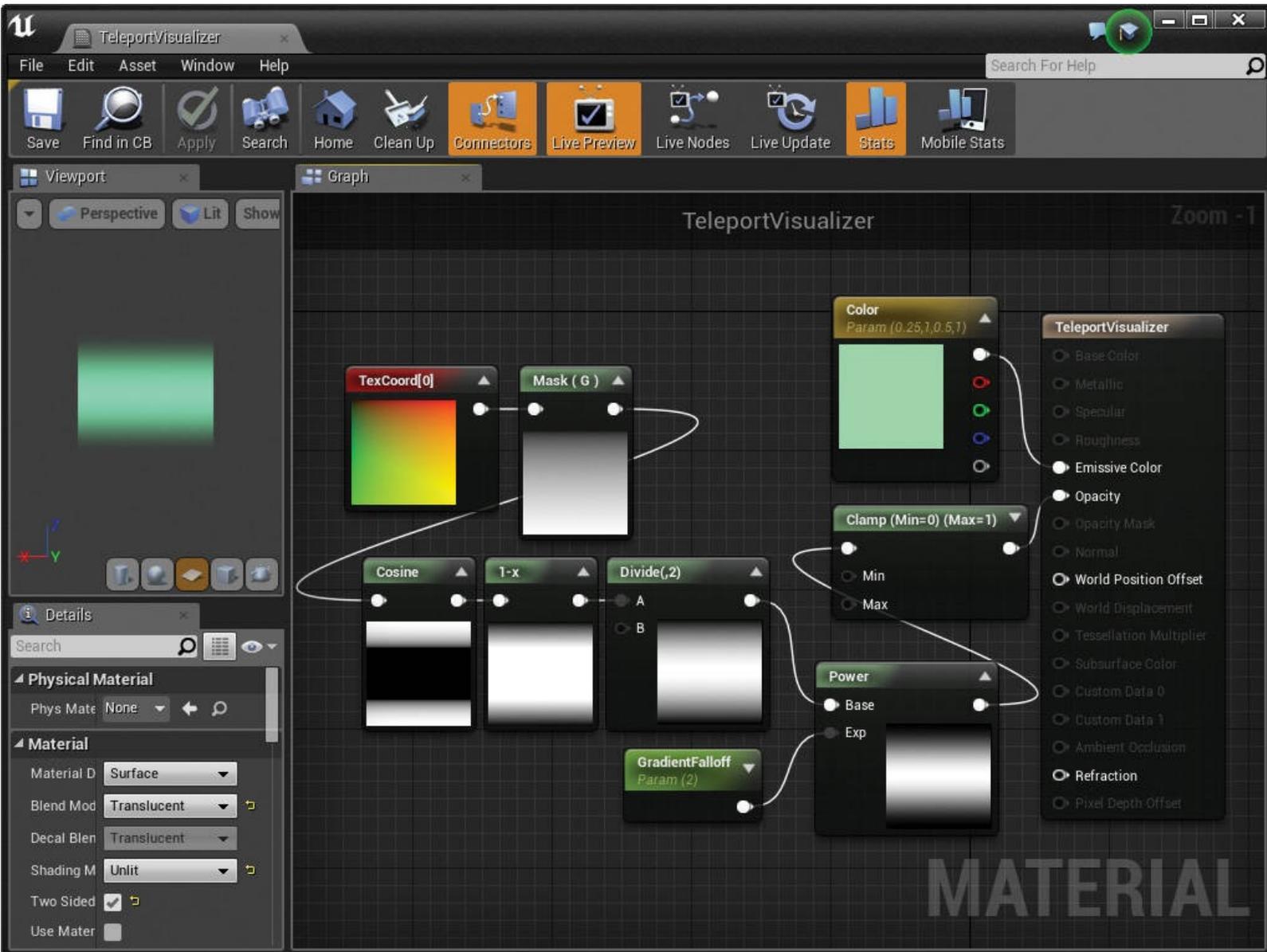


Figure 5.6 Teleport visualizer Material

10. To add some control to how sharply the gradient falls off, add a new Power node after the Divide, passing in a new scalar parameter called GradientFalloff with a default value of 2.
11. To ensure that this value remains in a 0 to 1 range, add a new Clamp node after the Power node.
12. Connect the output of the Clamp to the Opacity input of your Material. This creates a nice gradient.

Visualization Actor

Now you will create the Actor that will contain the visualization mesh:

1. In the Blueprints folder, create a new Blueprint Actor named TeleportVisualizer.
2. Open the new Blueprint and add a new Static Mesh Component named Visual.
3. Set the Component's Static Mesh property to S_EV_SimpleLightBeam_02. If you do not see it, enable Show Engine Content in the View options.
4. Set the Material's Element 0 to your new TeleportVisualizer Material.
5. Rotate the Component 180 degrees on the X-axis and scale it at 0.3 on the Z-axis to make it look more like a teleporter (see [Figure 5.7](#)).

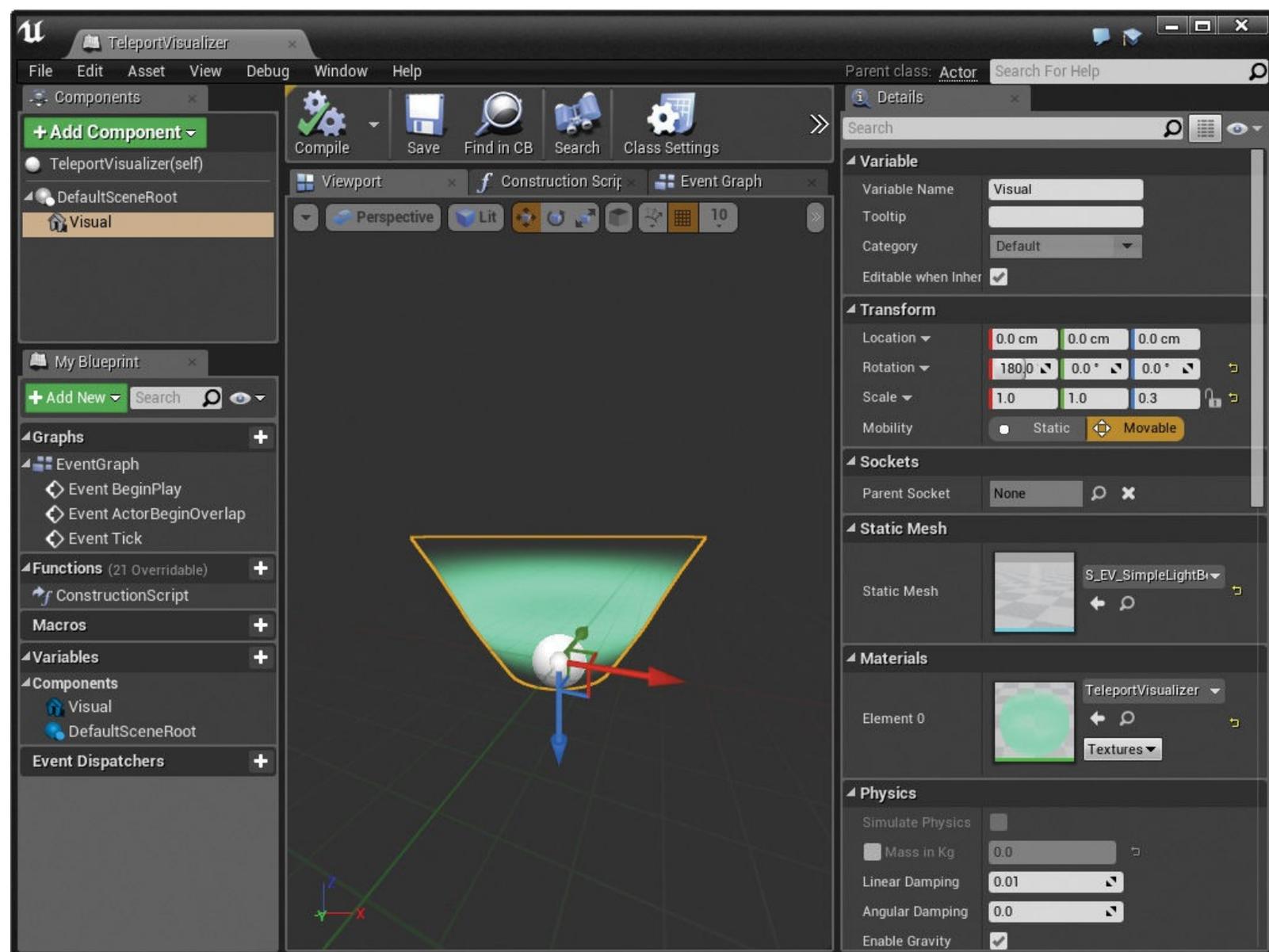


Figure 5.7 Teleport visualizer Actor

Simple Teleportation Volume

Now that you have a way to visualize your teleport, you can create a simple teleportation volume that allows the user to teleport onto its surface:

1. In the Blueprints folder, create a new Blueprint Actor named TeleportVolume.
2. Open the new Blueprint and add two new Components, a Box with its default name and a Child Actor named TeleportVisualizer.

3. Select the Box and set its Box Extent to (X = 200.0, Y = 200.0, Z = 1.0); see [Figure 5.8](#).

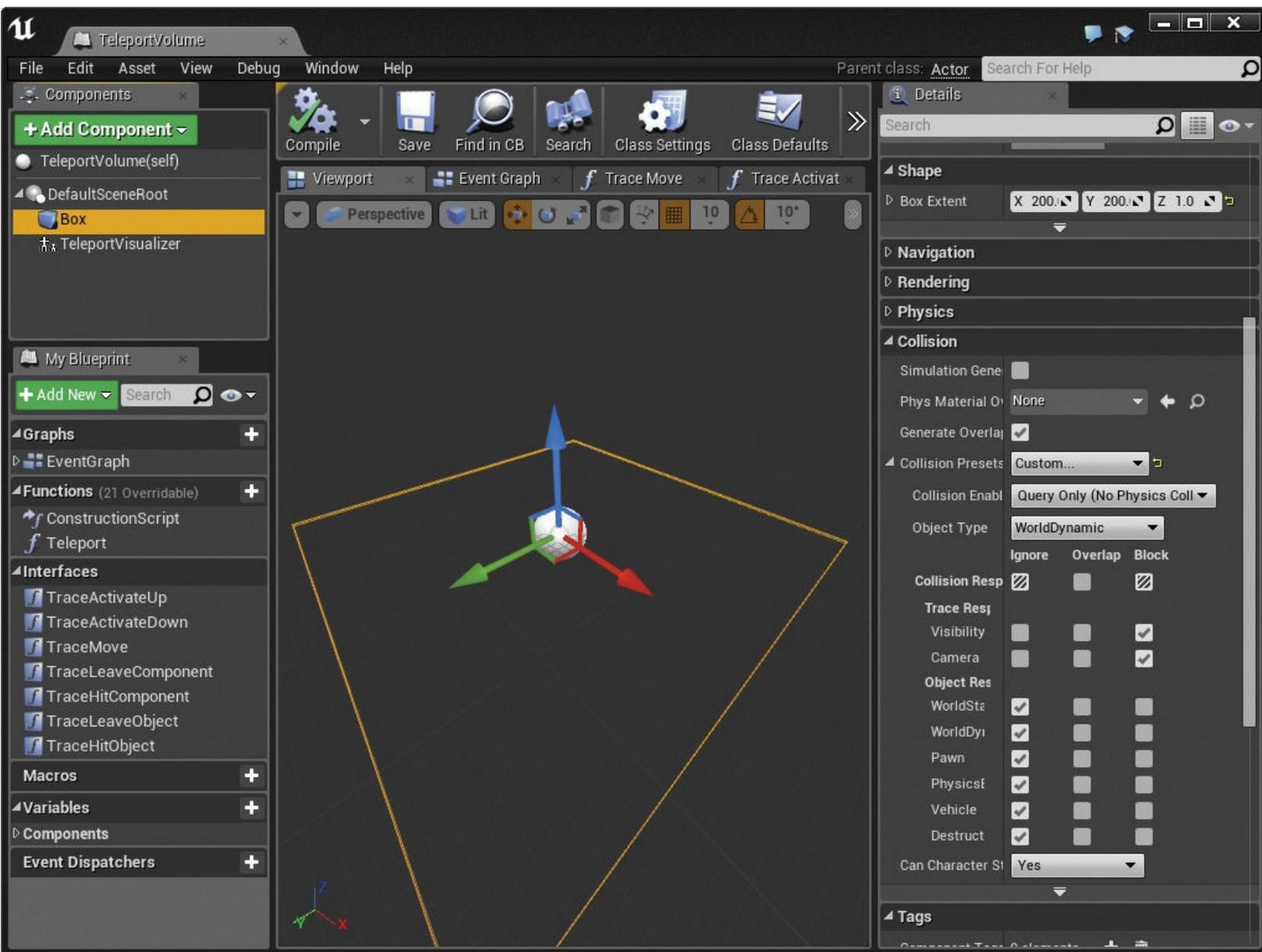


Figure 5.8 Teleport volume: Components

4. Set the Box's Collision Preset to Custom, making sure to set Collision Enabled to Query Only, and select Ignore for all collision responses apart from Visibility and Camera.
5. Select the Child Actor and set its Child Actor Class to TeleportVisualizer.
6. Uncheck the TeleportVisualizer's Visible flag; you will enable its visibility when the user looks at the volume.
7. Open the TeleportVolume's Class Settings and add the TraceInteractionInterface to the Implemented Interfaces section (remember to compile to get access to the interface functions).
8. Open the TraceMove function and break the Hit input.
9. Create a new variable getter for the TeleportVisualizer Component, and call SetWorldLocation on it.
10. Connect SetWorldLocation between the input and Return nodes of the function.
11. Connect the Location pin of the broken Hit Result to the New Location input pin in the SetWorldLocation node (see [Figure 5.9](#)). This moves the teleport visualizer every frame to the place where the player is interacting with the volume.

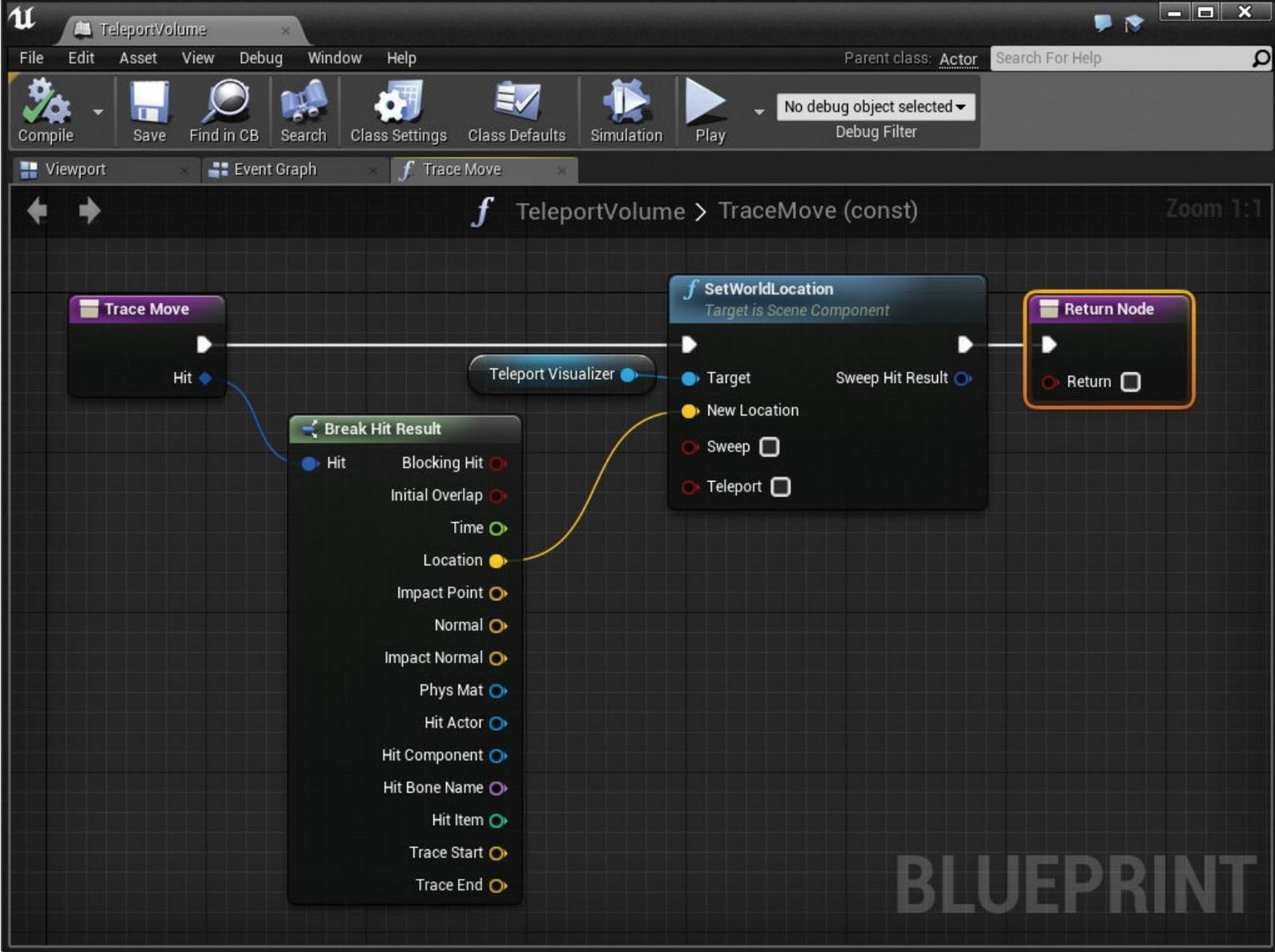


Figure 5.9 Teleport volume: TraceMove function

12. Open the TraceHitObject and TraceLeaveObject functions and add a new variable getter for the TeleportVisualizer Component.
13. Call SetVisibility on these Components, checking both boxes in the TraceHitObject function and checking only the last box in the TraceLeaveObject function. Then connect the input and Return nodes for the functions (see [Figure 5.10](#)). This shows and hides the visualizer when the user interacts with the volume.

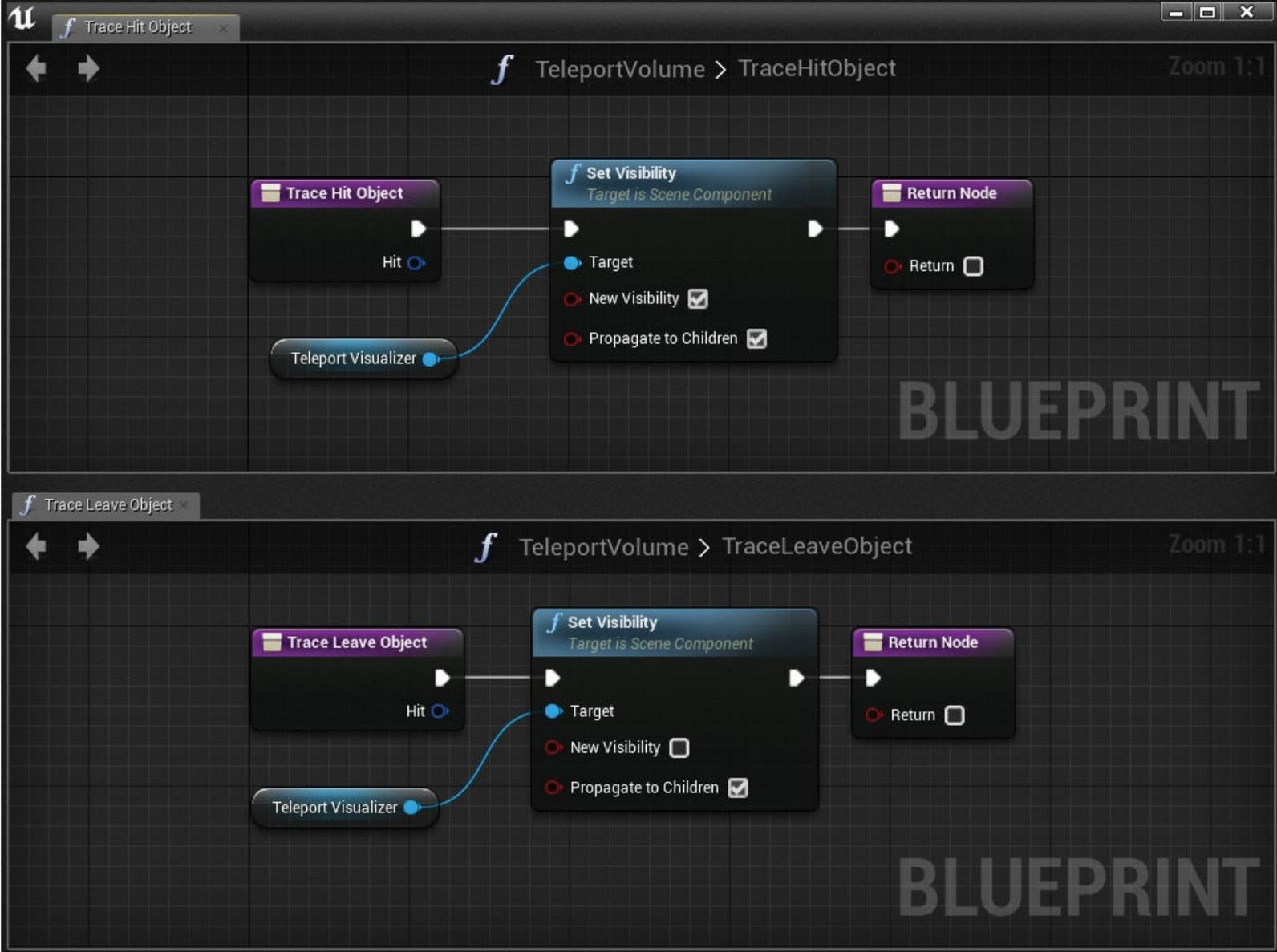


Figure 5.10 Teleport volume: trace hit and leave

14. Create a new function named TeleportPlayer, which takes in a Pawn named Player and a vector named Location. This will teleport the player while taking into account the position of the HMD.
15. Open the function, drag off of the Player input, and call the GetController function, then cast it to a PlayerController, connecting the input execution pin of the cast to the function input.
16. Drag off of the As Player Controller pin and get the PlayerCameraManager.
17. Drag off of the PlayerCameraManager getter and call the GetCameraLocation function (see [Figure 5.11](#)).

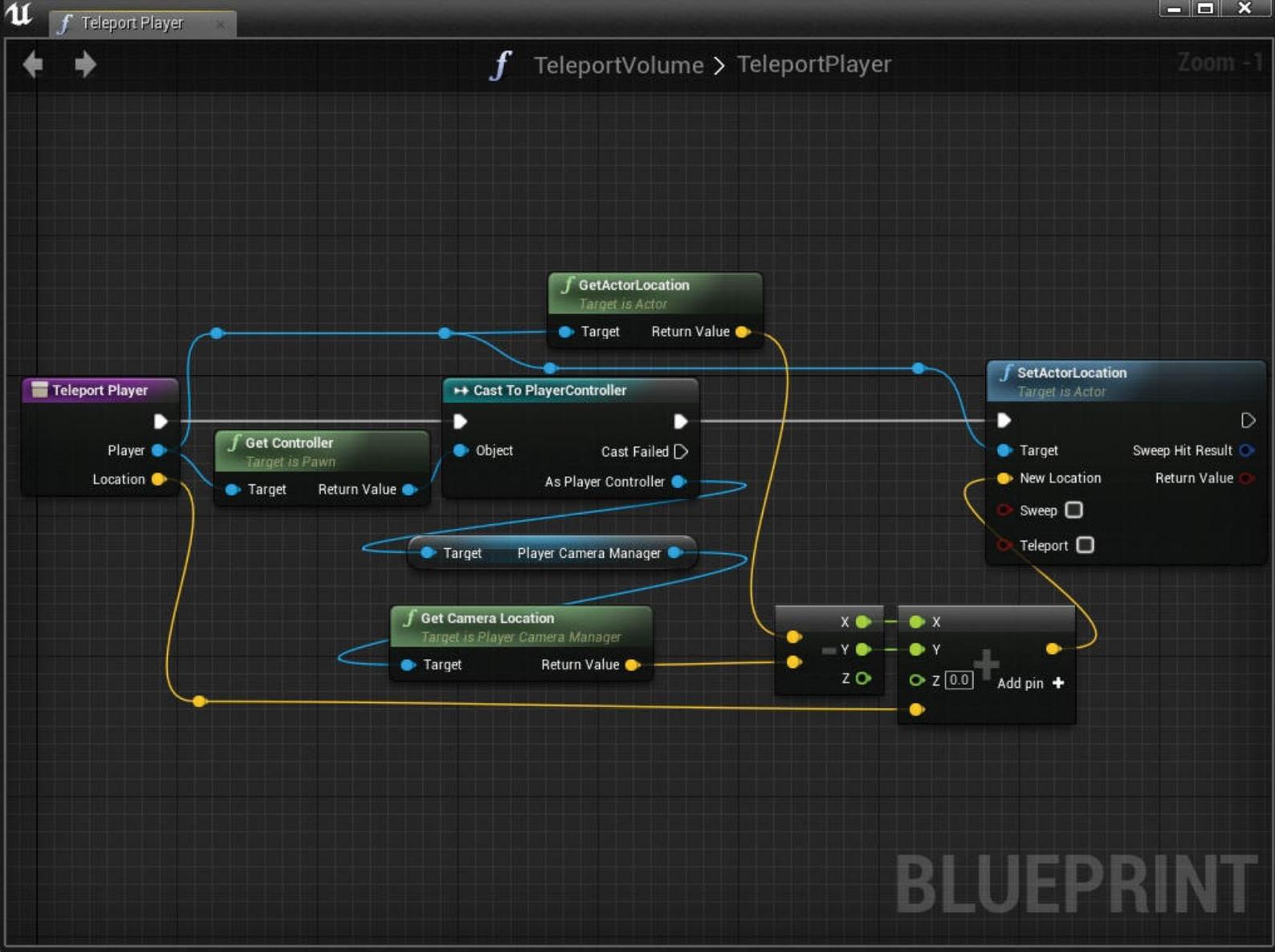


Figure 5.11 Teleport volume: teleport function

18. Drag off of the Player input once more and call the GetActorLocation function.
19. Drag off of this new function and subtract the camera location from GetCameraLocation.
20. Create a new Vector + Vector node and split the first input.
21. Split the output of the Actor – Camera location node and pass the X and Y into the corresponding inputs of the node from step 18.
22. Pass the input Location variable from the function into the second pin of the node from step 18. This will create a vector that offsets the player’s camera location to ensure that the actual location is in the center after the teleport.
23. Connect the output of this Vector + Vector node to the New Location input of SetActorLocation (see [Figure 5.11](#)).
24. Open the TraceActivateUp function and break the Hit Result input.
25. Call the TeleportPlayer function and connect it between the input and Return nodes.
26. Connect the Instigator input of the function to the Player input of the TeleportPlayer node.
27. Break the Location input of the TeleportPlayer node and break the Location pin of the Hit Result.
28. Connect Location X and Y to the corresponding inputs of the Location input of the TeleportPlayer node.

29. Create a new GetActorLocation node and split the Return Value.

30. Pass the Return Value Z into the remaining input of the TeleportPlayer node (see [Figure 5.12](#)). This teleports the player to the hit locations X and Y but the Actor's Z (to account for the teleport volume's size) whenever ActivateUp is pressed.

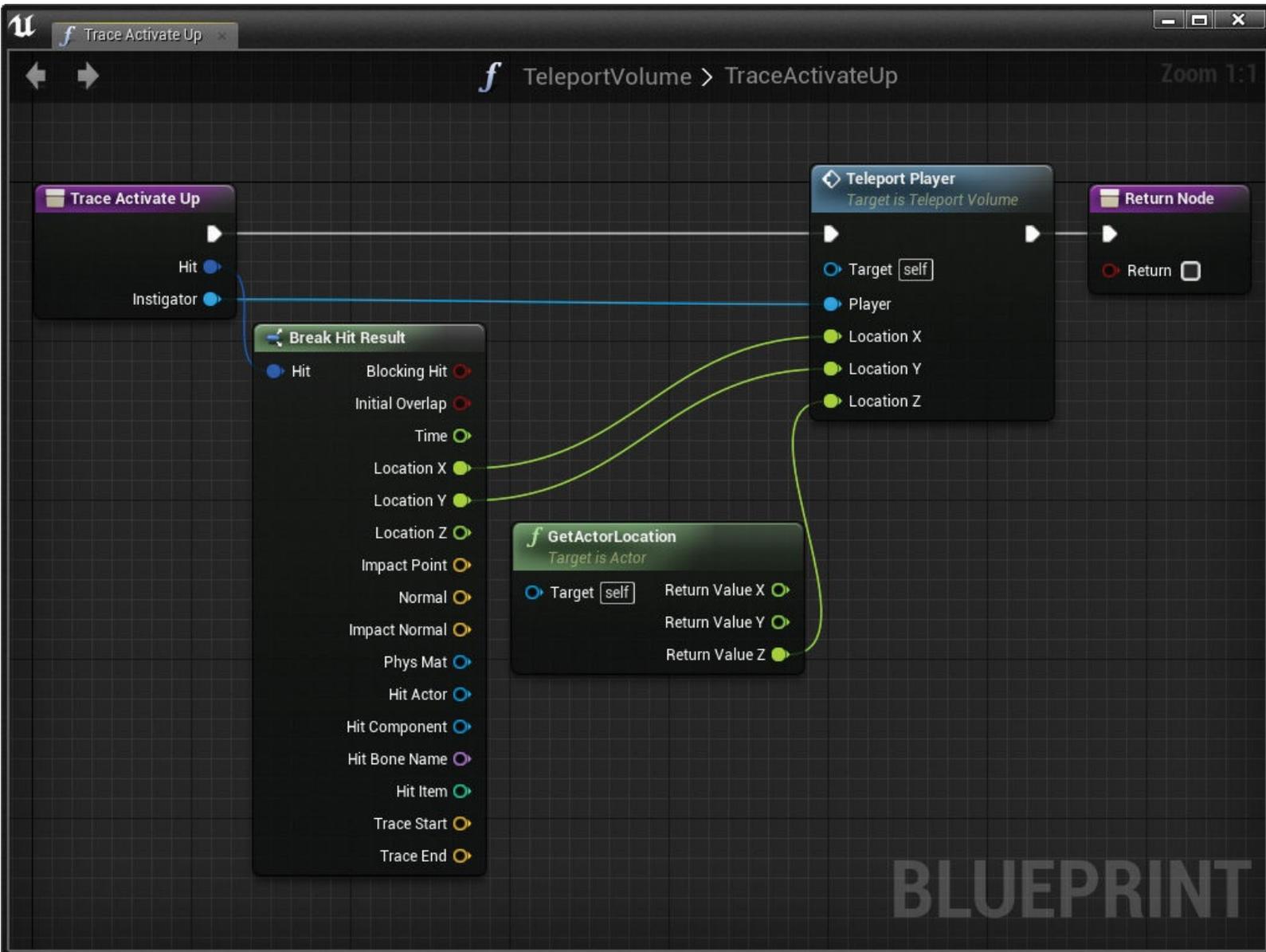


Figure 5.12 Teleport volume: activate

If you're using the files from [Chapter 4](#), just add this new teleportation volume to your level and hit Play to test it. If not, you may need to add in a game mode with your custom Pawn and the Interaction Component.

Summary

This chapter showed a way to extend the interactions from [Chapter 4](#) to add new parabolic trace functionality.

You also looked at how to implement a basic teleport volume that allows any user with the Trace Interaction Component to teleport to it.

Exercises

Can you think of any ways to expand this system? Some possibilities might include the following:

- Add the trail particle effect that shows where the user came from.
- Create a new teleport Actor that acts as a single point to which the player can teleport.
- Slow down the parabolic trace over a few seconds to stop the player from teleporting rapidly.
- Create a spline component that allows you to visualize your parabolic trace.

Chapter 6. Unreal Motion Graphics and 2D User Interfaces

Besides being primarily a 3D medium, a 2D UI still provides a convenient way to show users in VR relatively large amounts of information or allow for familiar interaction paradigms. This chapter looks at some of the challenges with 2D UI in VR and shows how to implement an Unreal Motion Graphics (UMG) interaction system from scratch. It also shows you how to integrate this system into other interaction systems.

Challenges with 2D UI in VR

Creating UIs in VR leads to some tricky problems that don't arise in traditional two-dimensional games. First, in traditional games, many UIs are head-locked (attached to the player's in-game head so wherever that moves, the UI follows). This behavior causes some major problems when ported to current VR systems.

Two main issues arise when trying to implement head-locked UIs in VR. The first of these is that traditionally head-locked UIs are orthographically projected to maintain clarity. This means the UI can be painted on top of all other elements in the game because it does not need to interact with the 3D game world.

However, if you were to project a user interface orthographically in VR, the player's eyes would treat it as if it were infinitely far away. As you can guess, this is a bad idea. It leads to a mismatch between the player's eyes and any object in the world that isn't at infinity (because something that the brain thinks is farther away is rendered on top of objects that are supposedly closer). This can cause discomfort.

Another reason that head-locked UIs are generally thought of as a bad idea in VR is that the lenses that are currently needed in HMDs to allow players to focus on a screen very close to their eyes aren't perfect and have the limitation that the clearest image is in the center of the player's vision. That is, as you transition to the periphery, the image gets less clear. Therefore, pushing UI elements to the edges of the player's vision results in less-readable UI than in the center. Another big issue is that because of differences in eye relief (distance from the player's eyes to the HMD's screen) among players, what may be in the periphery for some may be completely hidden for others!

How do you go about fixing these issues? A simple solution is to create a three-dimensional object (usually a quad) and place that in the game world, then render all the UI onto that. This solves the distance issue with orthogonal projection because all perspective projections from the game world are applied. It also solves the periphery issue with head-locked UI because the player is free to look at any part of the UI when needed. Indeed this is the solution that Unreal opts for.

However, this solution isn't problem free, and so this chapter discusses potential solutions for some of these issues.

Another issue with using traditional user interfaces in VR arises when you ask yourself if you actually need any UI. It may seem like a silly question, but because of the increased levels of immersion and the inherent stereo nature of the medium, most traditional two-dimensional UIs leave a lot to be desired. Often a less traditional UI creates a more immersive experience. For example, in a shooter game, it's more realistic to look into the cylinder to see how many bullets are left as opposed to just showing the number of bullets in the periphery. Of course, this all depends on the style and type of game you are going for; in some cases large text may actually suit the style of your VR world.

History and Compatibility of UMG

Unreal Motion Graphics (UMG) is Unreal Engine's WYSIWYG (what you see is what you get) tool for creating user interfaces. It allows you to hierarchically define menus, UI, and HUDs (heads-up displays) by placing widgets inside of a graphical editor and defining their look and functionality.

Integrated into the engine since version 4.4, UMG was traditionally conceived to create two-dimensional UIs, but it will serve our purposes. Originally, UMG suffered from the issues just outlined; however, since version 4.6, an experimental 3D UMG component has been added to the engine. This component allows you to place any UMG widget in 3D space and set up basic properties, like size and background color. However, it wasn't without its problems. The main problem facing VR developers was the limitation that there was no way to interact with these widgets once rendered in the world. To counteract this, many developers used tricks, such as creating custom widgets and detecting custom hits by doing the coordinate conversion from world units to relative UMG units manually, or, like me, adding the functionality in the form of a plugin that allowed developers to interact with 3D widgets manually with traces. (Refer to [Chapter 4](#), "[Trace Interaction](#).”)

However, as of engine version 4.13, these problems have been solved by the addition of the UMG Interaction Component, which we will use in this chapter.

Basic VR Menu

To create a basic VR menu, you need three main Actors:

- The traditional UMG widget that holds the actual UI widgets and defines their appearance and layout
- An Actor that holds the 3D Widget Component and displays that UMG widget in 3D space
- A player Pawn that holds the VR camera and the Widget Interaction Component that will allow you to interact with your widget in 3D space

First, let's create the UMG widget that resembles a typical menu in a VR game. It lets the player start the game, choose whether to enable any comfort modes, and change the difficulty of the game, and it allows the player to exit the game. Note that these options won't be functional, because the implementation depends on your game/experience.

1. Using Unreal Engine 4.13 or later, create a new blank Blueprint project without any starter content.
2. To create the folder structure, create two new folders under the Content folder. Name them Blueprints and UMG. Blueprints will hold any Blueprints you create, and the UMG folder will hold the UMG widgets.
3. Create a new Widget Blueprint in the UMG folder by clicking Add New → User Interface → Widget Blueprint. Name it MenuWidget.
4. Open this new widget, delete the Canvas Panel that already exists, and replace it with a Vertical Box. This creates an easy way to lay out the inner menu widgets.
5. Set the Fill Screen to Custom with a Width and Height of 200. This will be represented as 200UU (2m) in-game.
6. Add a new button to your UMG widget by dragging it on top of the Vertical Box panel. Name this button StartButton and set its Padding to (Left = 10.0, Top = 10.0, Right = 10.0, Bottom = 5.0).
7. Drag in a new Text widget, attach it to the StartButton widget, and change its text to "Start."
8. Drag in a new Check Box widget underneath the Start button and name it ComfortModeCheckBox, setting its state to Checked and Padding to (Left = 10.0, Top = 5.0, Right = 10.0, Bottom = 5.0).

9. Drag a new Text widget on top of this Check Box and set its Text to “Comfort Mode” and set its Font Size to 12.
10. Drag in a new Combo Box widget and place it under the Comfort Mode Check Box, adding three new strings to its Default Options (“Easy,” “Normal,” and “Hard”) and setting its Selected Option to Normal (see [Figure 6.1](#)).

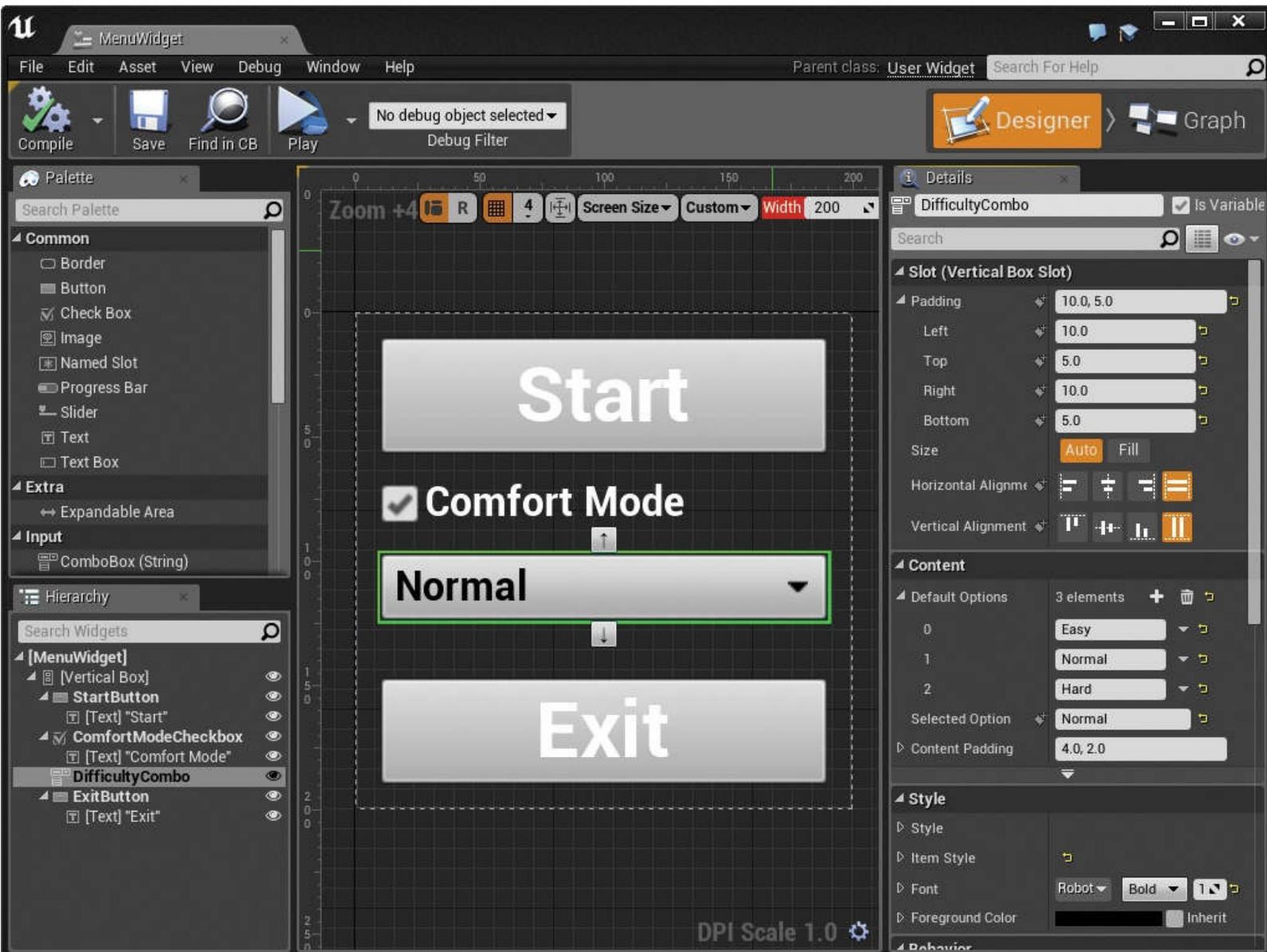


Figure 6.1 Basic VR menu with buttons to start and exit the game and some options for comfort mode and game difficulty

11. Change the Font Size for this Difficulty Combo to 12 and set the Padding to (Left = 10.0, Top = 5.0, Right = 10.0, Bottom = 5.0).
12. Copy the Start button by selecting it and pressing Ctrl + C.
13. Duplicate the now-copied widget by selecting the Vertical Box panel and pressing Ctrl + V.
14. Select this newly created widget, name it ExitButton, and then set its Vertical Alignment to Bottom, its Size to Fill, and its Padding to (Left = 10.0, Top = 0.0, Right = 10.0, Bottom = 10.0).
15. Select the text inside this button and set the Text property to “Exit.”

Menu Actor

After creating the 2D Menu widget, you need to create the three-dimensional Actor that can be placed in the world to hold a 3D representation of the widget:

1. Create a new Actor Blueprint in the Blueprints folder and name it MenuActor.
2. To add the menu to this Actor, add a new Widget Component.
3. With this new Widget Component selected, in the Details panel under User Interface, set the Widget Class to your just-created MenuWidget.
4. In the same tab, set the Draw Size to (X = 200, Y = 200). This matches the settings you used when creating the widget, but this does not have to be the case; UMG widgets can scale to arbitrary sizes.
5. Because a Draw Size of (X = 200, Y = 200) is still large in the world, set the Component's Scale to (X = 0.5, Y = 0.5, Z = 0.5). This has the nice side effect of making the text clearer because the widget is being rendered at a high resolution and then scaled down.
6. Still in the Details for the Widget Component, set the background color to (R = 0.0, G = 0.0, B = 0.0, A = 0.2) and the Blend Mode to Transparent. This causes a performance hit, but the transparent black background will help it blend with the world (see [Figure 6.2](#)).

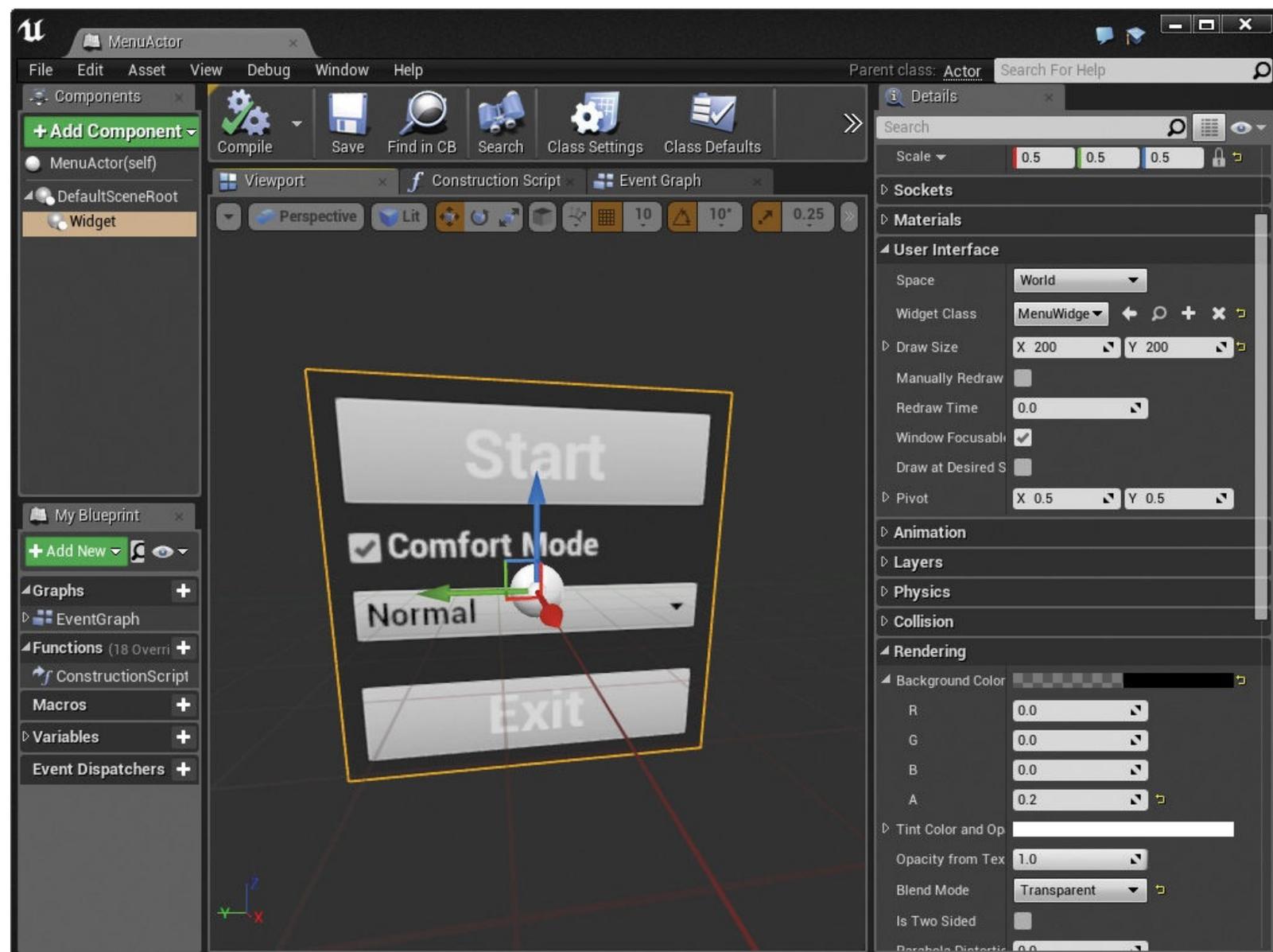


Figure 6.2 VR menu Actor containing the needed 3D Widget Component

7. To respond to user interaction with the widget, you will need to get access to the Widget Object. To

do this drag a getter for the Widget Component into the Event Graph and call GetUserWidgetObject (see [Figure 6.3](#)).

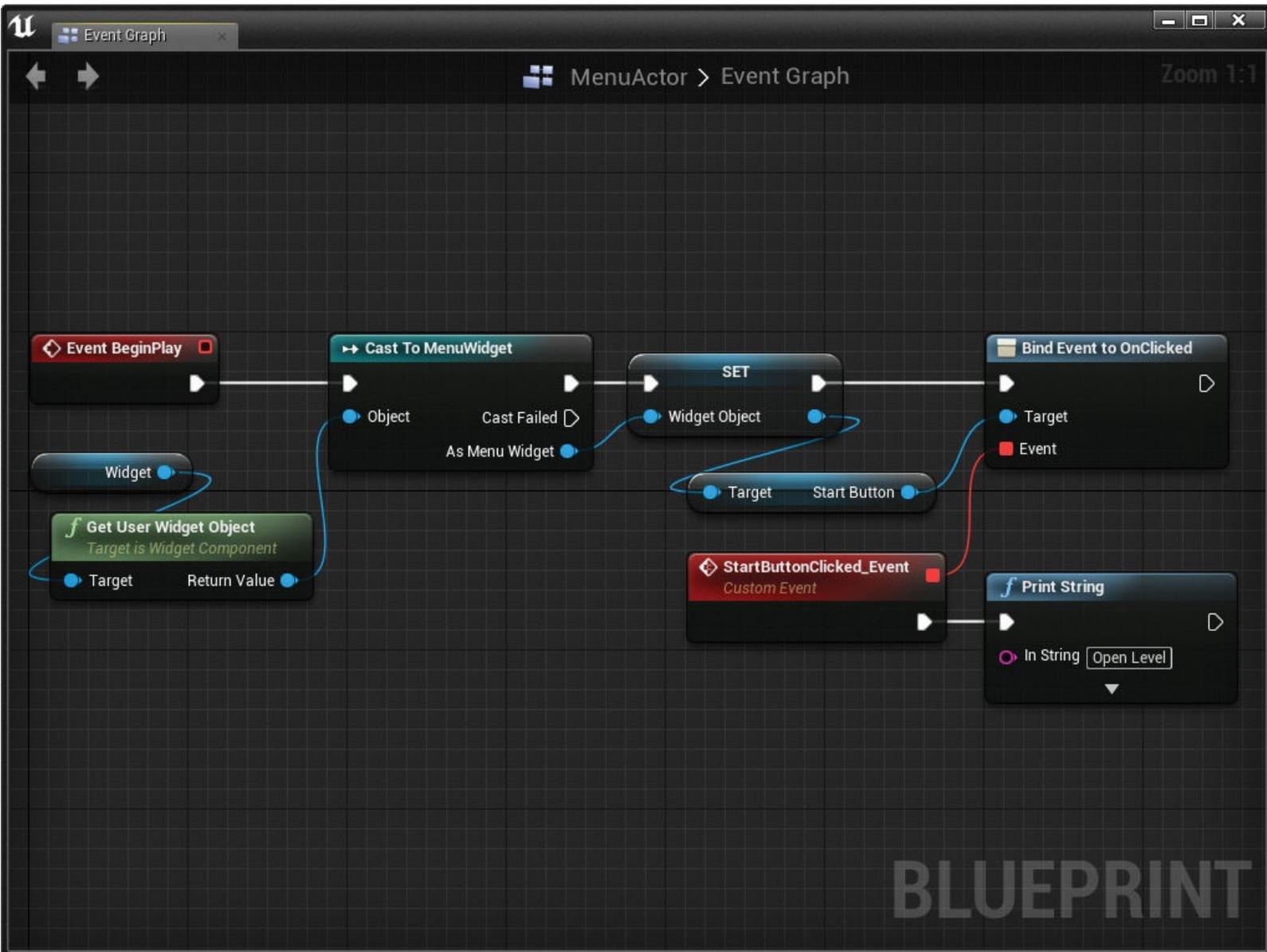


Figure 6.3 Getting the WidgetObject from a Widget Component and assigning an event to the Start button's clicked event dispatcher

8. To get access to the specific widgets inside your menu, drag off of the GetUserWidgetObject node and cast to your MenuWidget object, connecting the input execution pin of this cast to EventBeginPlay.
9. Drag off of the As Menu Widget output pin of the cast and create a new variable of type Menu Widget by clicking the Promote to Variable option. Name it WidgetObject. This makes it easier to access your widget whenever you need it later.
10. To respond to events from this widget, drag off of the blue output pin of the WidgetObject setter and get the StartButton variable, which has a reference to the Start button in your menu.
11. To respond to a click event for this button, drag off of the variable getter and select the Assign OnClicked option. This creates a new event and automatically binds it to the dispatcher from the button (see [Figure 6.3](#)).
12. Now you can do anything you wish on this event, and it will be called when the player presses the Start button (like opening a new level that contains your gameplay).

Menu Pawn

After creating the 2D widget and the 3D widget Actor, you need to create a Player Pawn that will hold the Widget Interaction Component, which allows a player to interact with the 3D menu:

1. Create a new Pawn Blueprint inside the Blueprints folder and name it MenuPawn.
2. Create three new Components: a Scene named CameraRoot, a Camera named Camera, and a Widget Interaction Component named WidgetInteraction.
3. Place the Camera Component on top of the CameraRoot and place the Widget Interaction Component on top of the Camera. This makes the Camera a child of the Scene Component and makes the Widget Interaction Component a child of the Camera, which allows a player to interact with widgets using the Camera's view (see [Figure 6.4](#)).

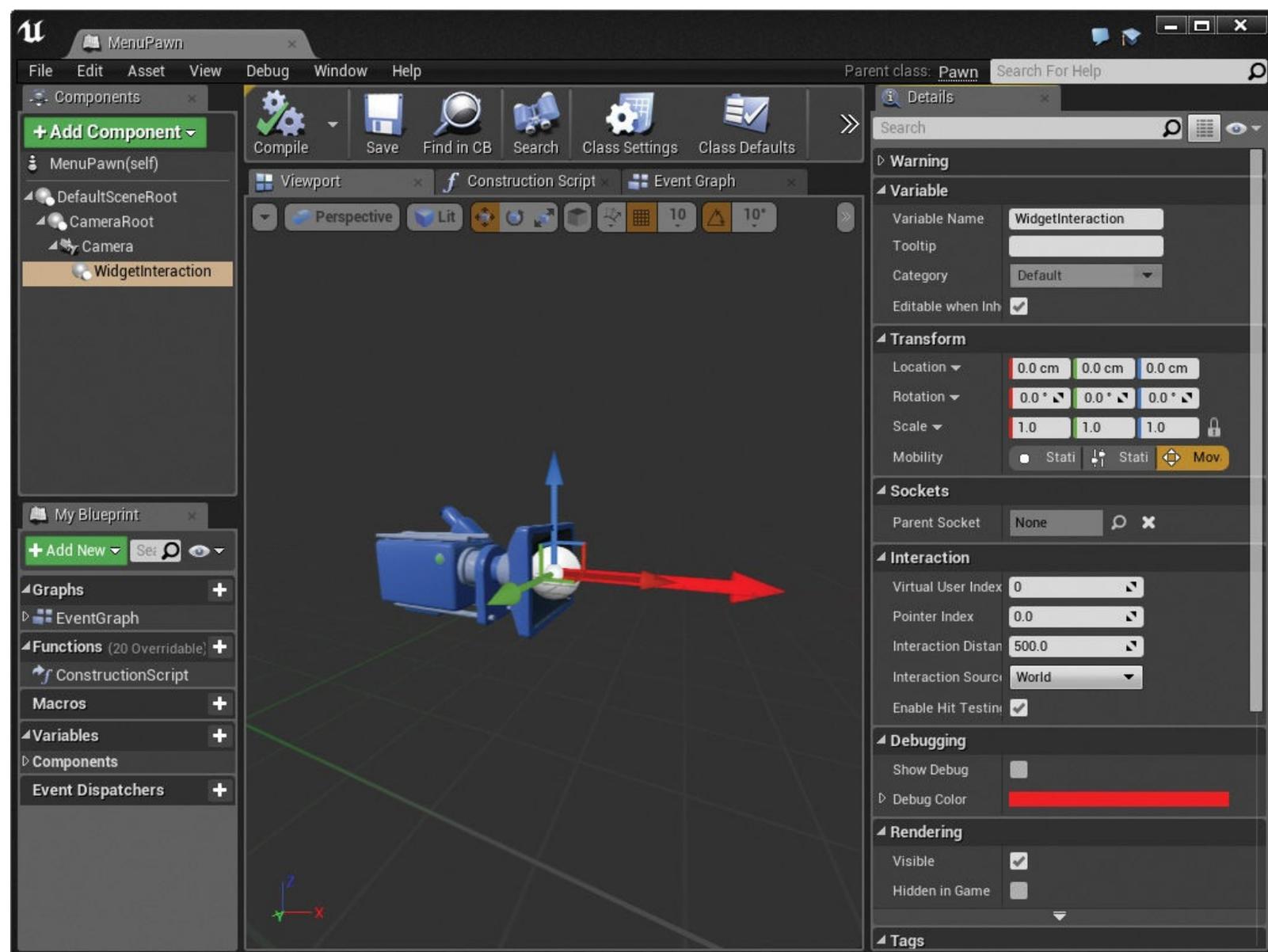


Figure 6.4 VR menu Pawn with a base Widget Interaction Component

4. The Widget Interaction Component has a few properties to note. Interaction Source allows you to tell the Component how it should interact with widgets; the default is World, which traces in the direction of the arrow and interacts with any Widget Component it hits. Interaction Distance is the maximum interaction distance the Component traces when using the World interaction method. Last, the Interaction Component has the notion of virtual users, which allows you to define different users and allows multiple players to interact with the widget. This can be set with the Virtual User Index property.

- The hover events are handled automatically because the interaction method for the Widget Interaction Component is World, so all you need to do to get a functioning interaction system is to activate the press events. In the Event Graph, create a new LeftMouseButton event (you can replace this with any interaction method you need such as a touchpad press).
- Drag in a variable getter for the Widget Interaction Component and call the PressPointerKey function, passing in the LeftMouseButton key and connecting the input execution pin to the Pressed pin of the LeftMouseButton event.
- Drag off of the Widget Interaction getter once more and call the ReleasePointerKey event, passing in the LeftMouseButton key and connecting the Released pin of the LeftMouseButton event to the input execution pin (see [Figure 6.5](#)). This simulates the mouse button on the current widget with which the Interaction Component is interacting.

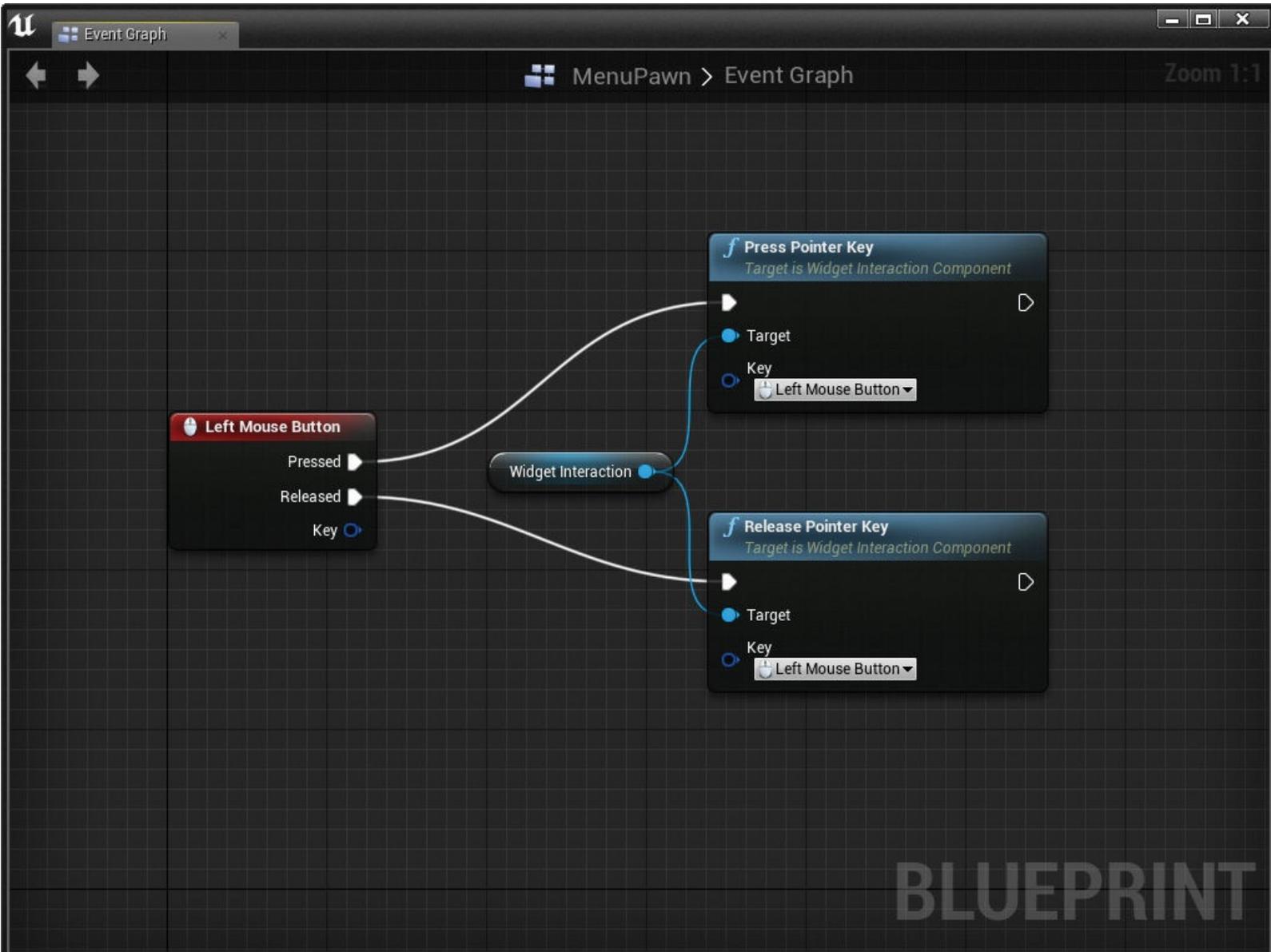


Figure 6.5 VR menu Pawn: activating click events for the Widget Interaction Component

Now that you have the basics for a simple menu interaction system, you can add the menu Actor and the Pawn to a level and test the interaction.

Custom Menu Interaction

In the previous section, you used the built-in menu interaction method that is enabled by default when using the Widget Interaction Component. However, this has some limitations, the primary one being that you are stuck with the line trace that the Component provides.

Luckily, it is easy to integrate this Component into a more complicated interaction system such as the one you developed in [Chapter 4](#) and [Chapter 5](#), “[Teleportation](#).” When doing so, there are two options: have one Widget Interaction Component next to your custom Interaction Component and make it aware of this Widget Interaction Component so it can call the needed functions to interact with any menus it hits, or have each menu Actor have its own Widget Interaction Component that calls the appropriate functions on the Component. The first approach is nice, because there is only one Interaction Component to deal with; however, the second approach is also nice because your custom Interaction Component remains independent of the experimental Widget Interaction Component. The next two subsections describe both approaches.

Implementing Custom Menu Interaction: Approach 1

This first approach adds menu interaction functionality to the Trace Interaction Component created in [Chapter 4](#):

1. Migrate the Trace Interaction Component from [Chapter 4](#) into this chapter’s project.
2. Open the Trace Interaction Component. Create a new variable named `WidgetInteraction` of type `Widget Interaction Reference` and set it to `Private`.
3. Unlike other variables, you will not be able to set this variable’s default value in the Components tab of its parent Blueprint. This is because the Widget Interaction Component won’t have been spawned until the level has started, so no reference is available until runtime. To get around this, create a new custom event named `Setup` that takes in a Widget Interaction Component named `WidgetInteraction` (see [Figure 6.6](#)).

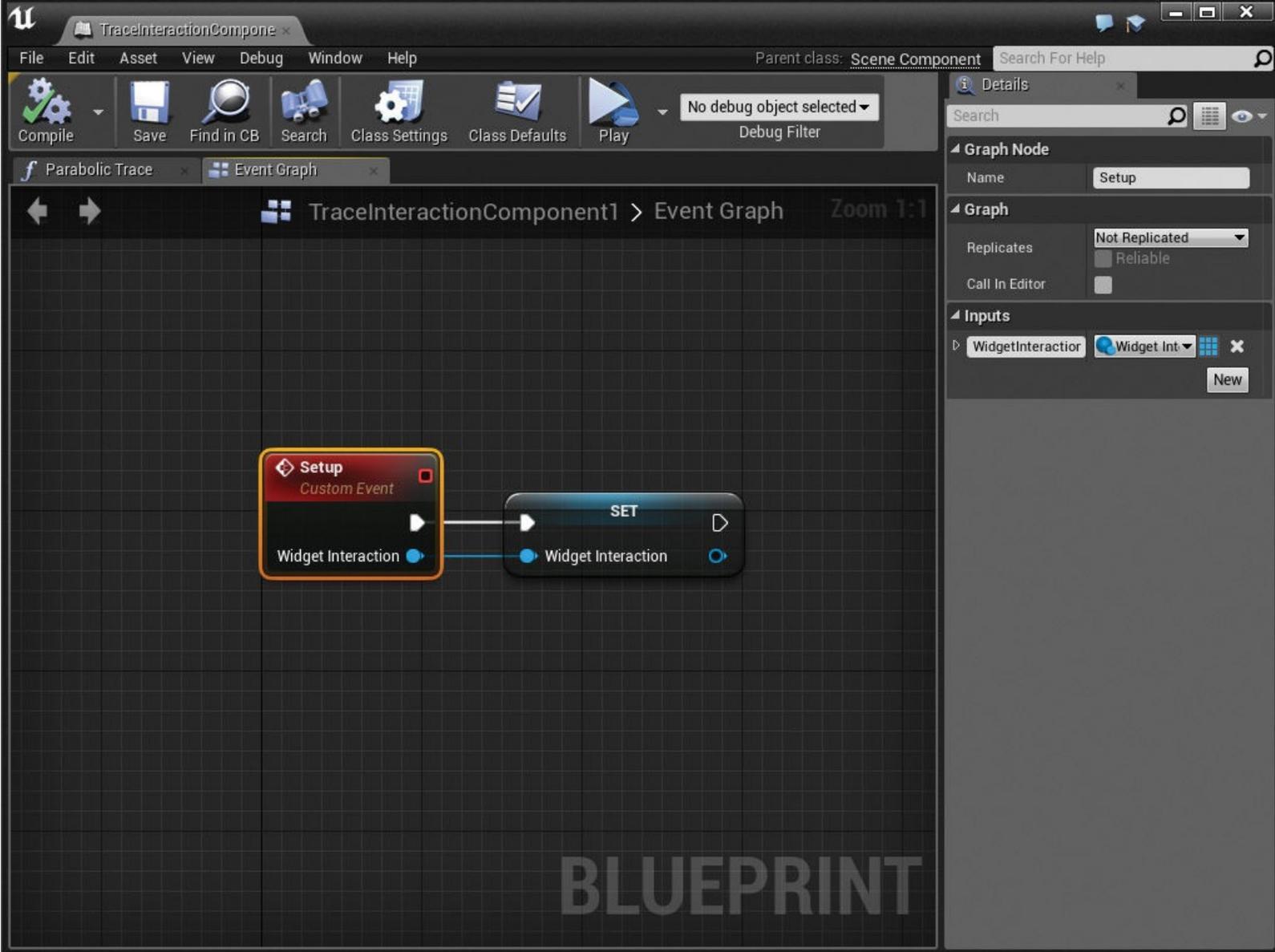


Figure 6.6 Dependency injection to the Trace Interaction Component through a Setup function

4. Create a variable setter for the WidgetInteraction variable and connect both pins to the Setup custom event. You will call this from the parent Blueprint and pass in the Widget Interaction Component that you want the Trace Interaction Component to use. Think of this as a way to pass in any dependencies that the Component might have in a clean way all at once.
5. Moving to the EventBeginPlay node, create a new variable getter for the WidgetInteraction reference and create a new IsValid node off of it. Connect the input execution pin for this node to the output of the InteractWithHit function call (see [Figure 6.7](#)).

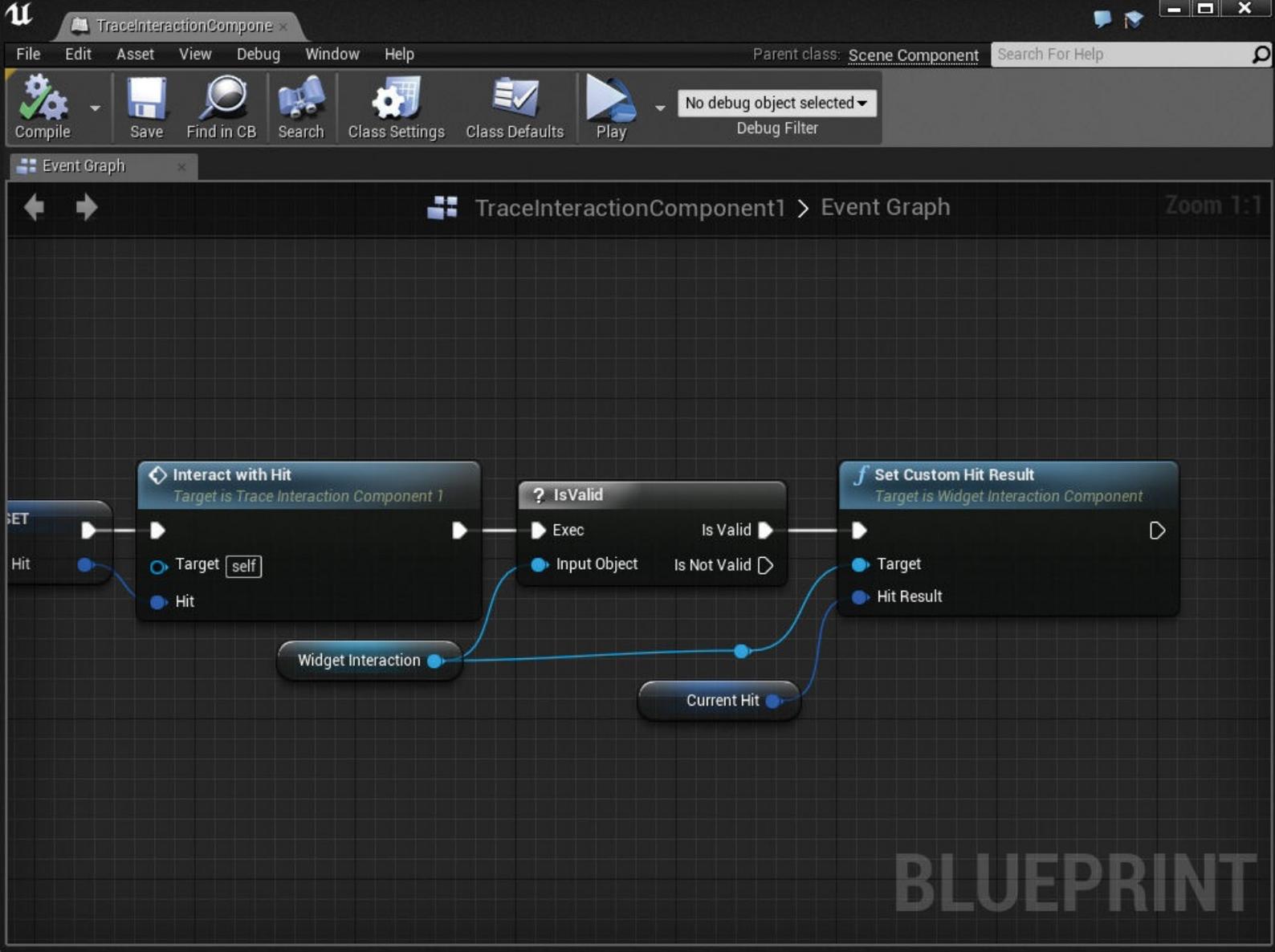


Figure 6.7 Setting the custom hit of a Widget Interaction Component on the Trace Interaction Component

6. Dragging off of the WidgetInteraction getter once more, call the SetCustomHitResult function, connecting a new Current Hit getter to its Hit Result input and connecting the IsValid output of the validity check to its input execution pin. This will set a custom hit for the given Widget Interaction Component, which will allow it to interact with what your Trace Interaction Component is interacting with.
7. Going down to the ActivateDown and ActivateUp custom event nodes, create a new WidgetInteraction variable getter for both and do the same validity check as in step 5 (see [Figure 6.8](#)).

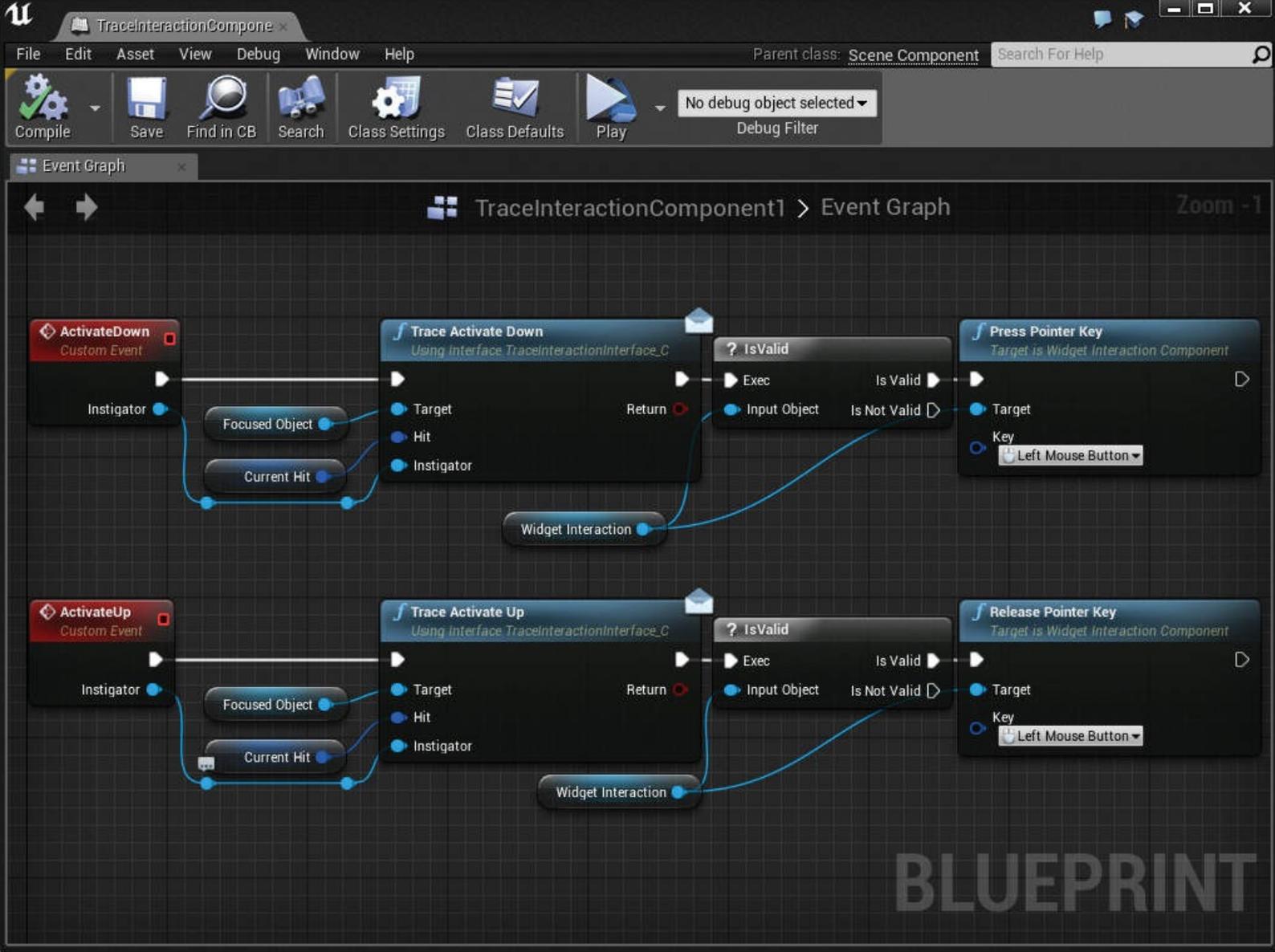


Figure 6.8 Simulating the pointer key on the currently hit widget in the Trace Interaction Component

8. After the IsValid nodes, call PressPointerKey and ReleasePointerKey on the ActivateDown and Up events respectively, making sure to pass in the LeftMouseButton key. This will activate any hovered menu when the Trace Interaction Component is told to activate.
9. Inside the Blueprints folder, create a new Pawn named TraceComponentPawn.
10. Create four new Components inside this Pawn: a Scene named CameraRoot, and Camera, TraceInteraction, and WidgetInteraction Components named with their default names.
11. Attach the Camera to the CameraRoot and attach the Trace Interaction Component to the Camera.
12. Select the Widget Interaction Component and set the Interaction Source to Custom (see [Figure 6.9](#)). This means that the default behavior of the Widget Interactor will be overridden and the SetCustomHitResult from step 6 will be used instead.

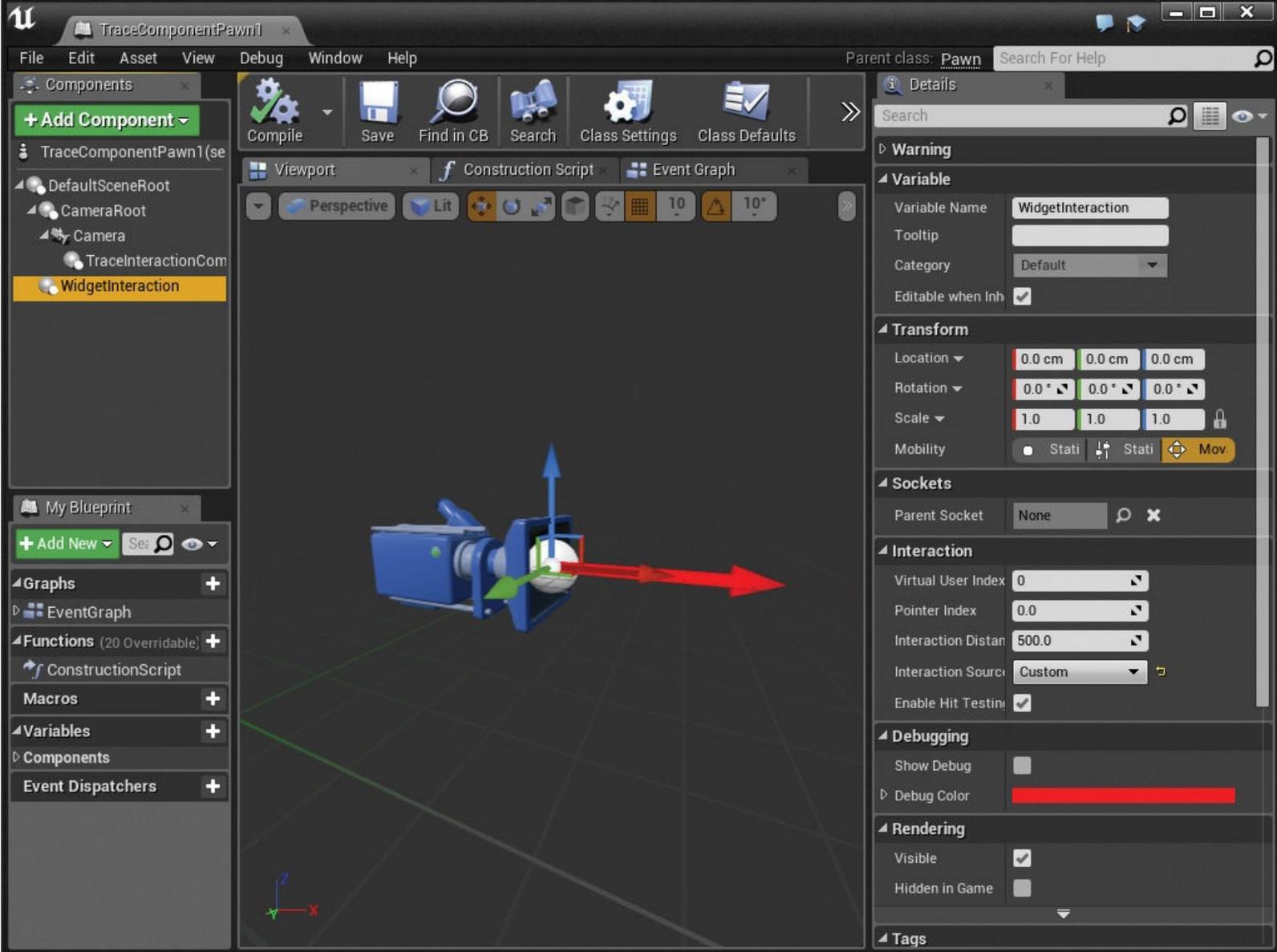


Figure 6.9 Trace Interaction Pawn with a Widget Interaction Component set to take in custom hit results

13. The last thing to do to get this Pawn working is to activate the Trace Interaction Component. First create a getter for the Trace Interaction Component and call the Setup function, passing in the Widget Interaction Component to the remaining input to the function.
14. Connect the Setup function call to the EventBeginPlay node.
15. Create a new LeftMouseButton event.
16. Create a new getter for the Trace Interaction Component and call the ActivateDown and ActivateUp functions, passing in a reference to Self as the Instigator input.
17. Connect the Pressed pin of the LeftMouseButton event to ActivateDown and the Released pin to ActivateUp (see [Figure 6.10](#)). This activates the Trace Interaction Component which in turn will activate the Widget Interaction Component.

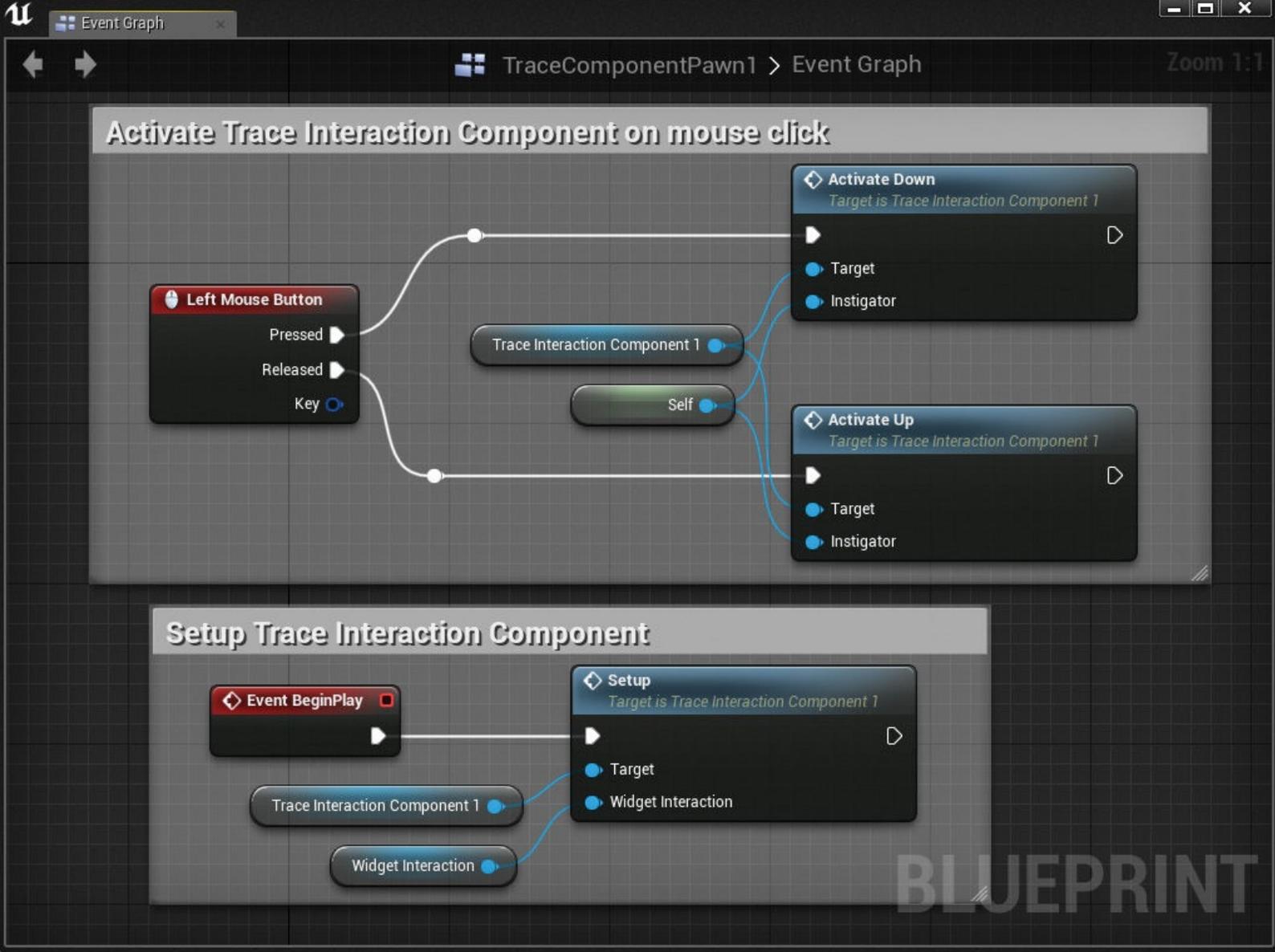


Figure 6.10 Activate the Trace Interaction Component with the left mouse button and pass any dependencies into the Trace Interaction Component on Begin Play.

You now have a generic Trace Interaction Component that works with UMG menus.

Implementing Custom Menu Interaction: Approach 2

The second approach to adding custom menu interaction is to take the Trace Interaction Component developed in [Chapter 4](#) and instead of expanding on it, using the already programmed functionality, push the job of interacting with the menu to the menu itself:

1. Migrate the Trace Interaction Component from [Chapter 4](#) into this chapter’s project.
2. Create a new Pawn Blueprint in the Blueprints folder. Name it TraceComponentPawn.
3. Open this Pawn and create three new Components: a Scene Component named CameraRoot, and Camera and TraceInteraction Components with their default names.
4. Attach the Camera to the CameraRoot and attach the TraceInteraction Component to the Camera. This allows anything that the player looks at to be interacted with.
5. In the Event Graph for the Pawn, create a new LeftMouseButton event.
6. Create a new variable getter for the TraceInteraction Component and call both the ActivateDown and ActivateUp functions, passing in Self as the Instigator input.

7. Attach the Pressed pin of the LeftMouseButton to ActivateDown and the Released pin to ActivateUp (see [Figure 6.11](#)).

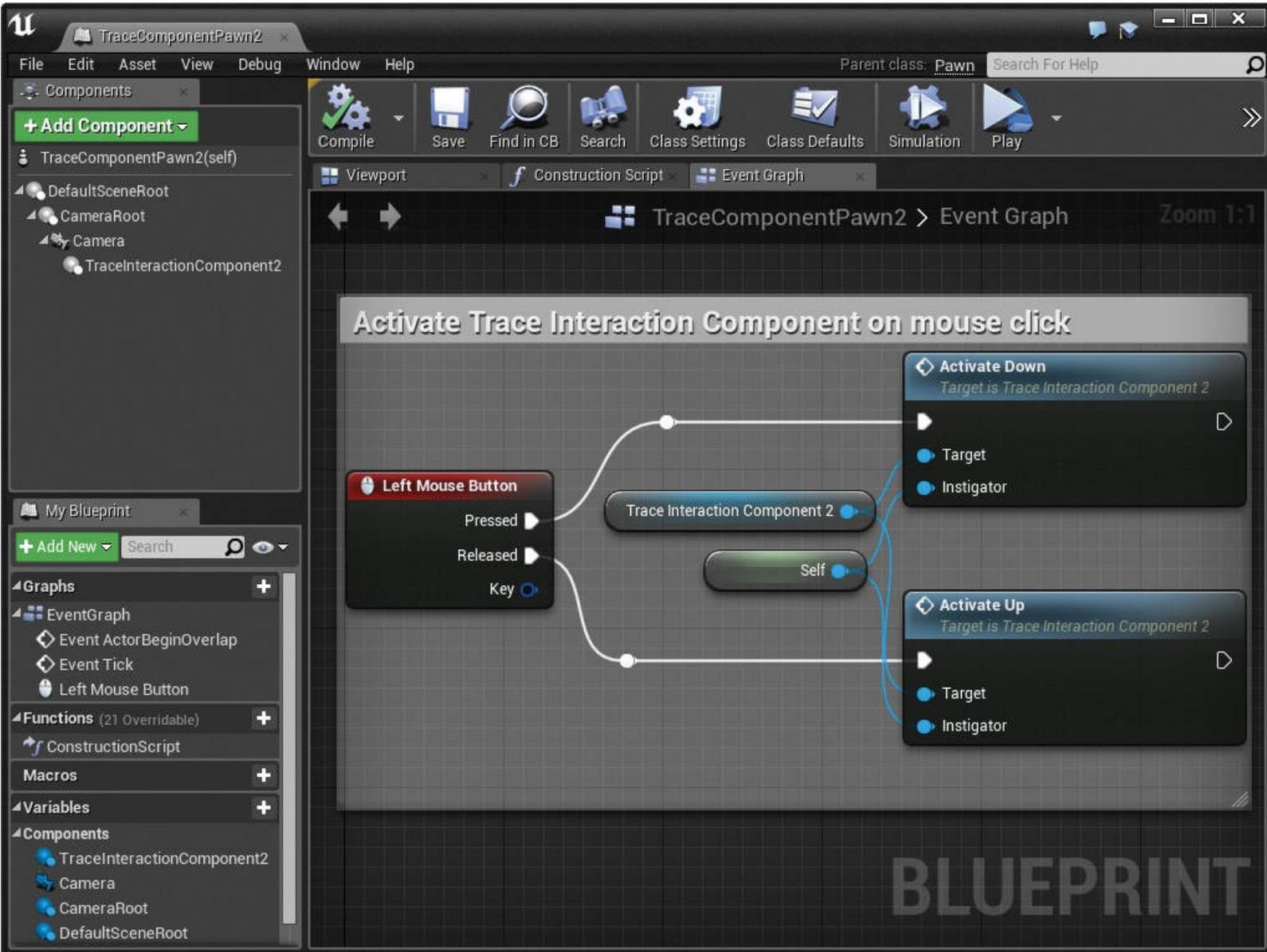


Figure 6.11 A Trace Interaction Pawn with no Widget Interaction Component, which will be handled on a per-menu basis

8. In the Blueprints folder, open the MenuActor Blueprint and add the Trace Interaction Interface in the Class Settings.
9. Add a new Widget Interaction Component to this Blueprint, setting its Interaction Source to Custom (see [Figure 6.12](#)).

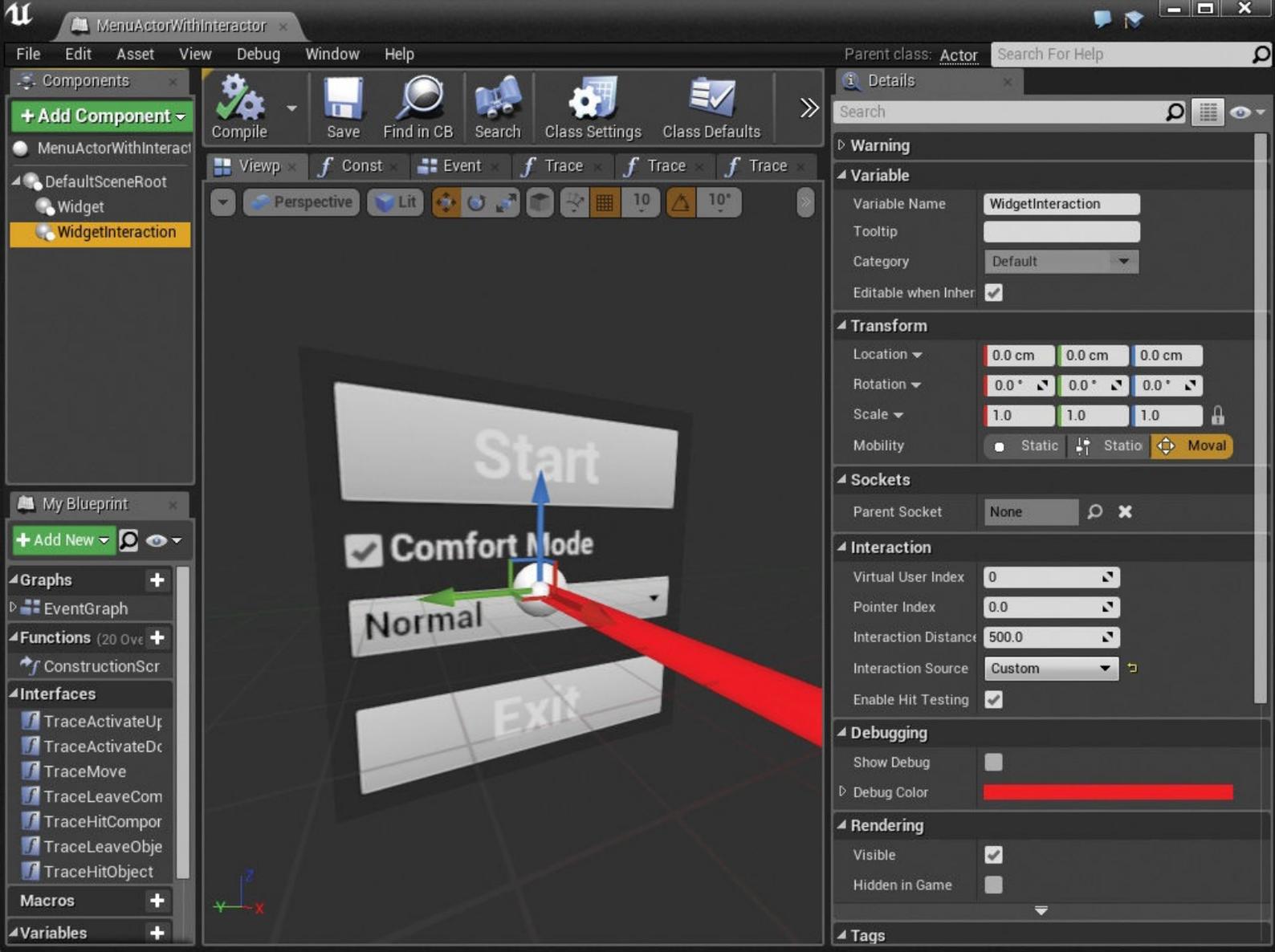


Figure 6.12 Widget Interaction Component added to a menu Actor with a custom Interaction Source

10. Open the TraceMove interface function. (If you don't see this function, compile your Blueprint.)
11. Create a new Widget Interaction variable getter and call the SetCustomHitResult function, connecting it between the function input and the Return node, making sure to connect the input Hit to the function's Hit Result input (see [Figure 6.13](#)). This sets the custom hit whenever the player is looking at the menu.

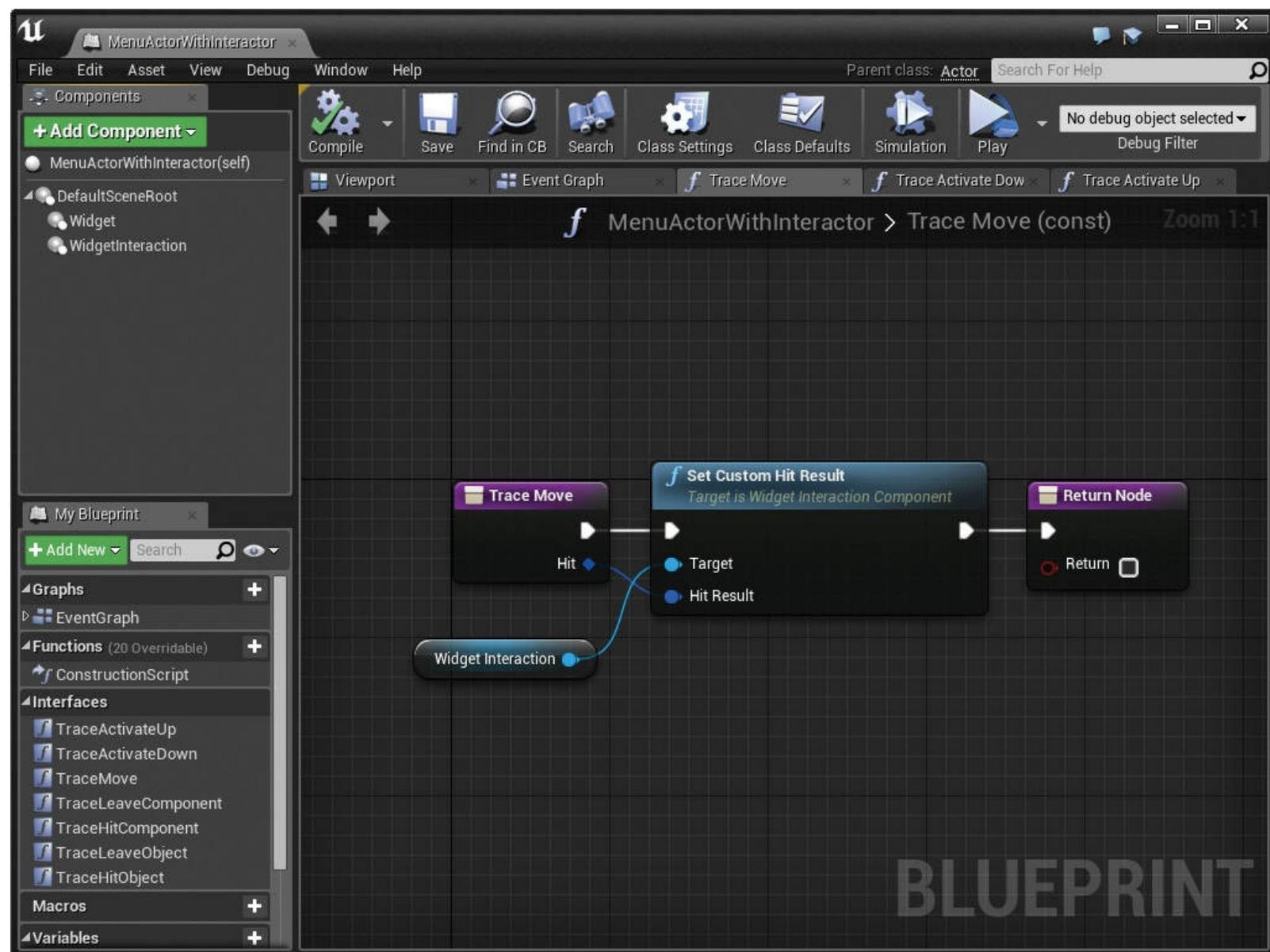


Figure 6.13 Setting the Custom Hit Result of the Widget Interaction Component in the menu Actor

12. Open the TraceActivateDown function and create another getter for the Widget Interaction Component.
13. Call the PressPointerKey function, passing in the LeftMouseButton as the key, and connect this function between the input to the function and the Return node (see [Figure 6.14](#)).

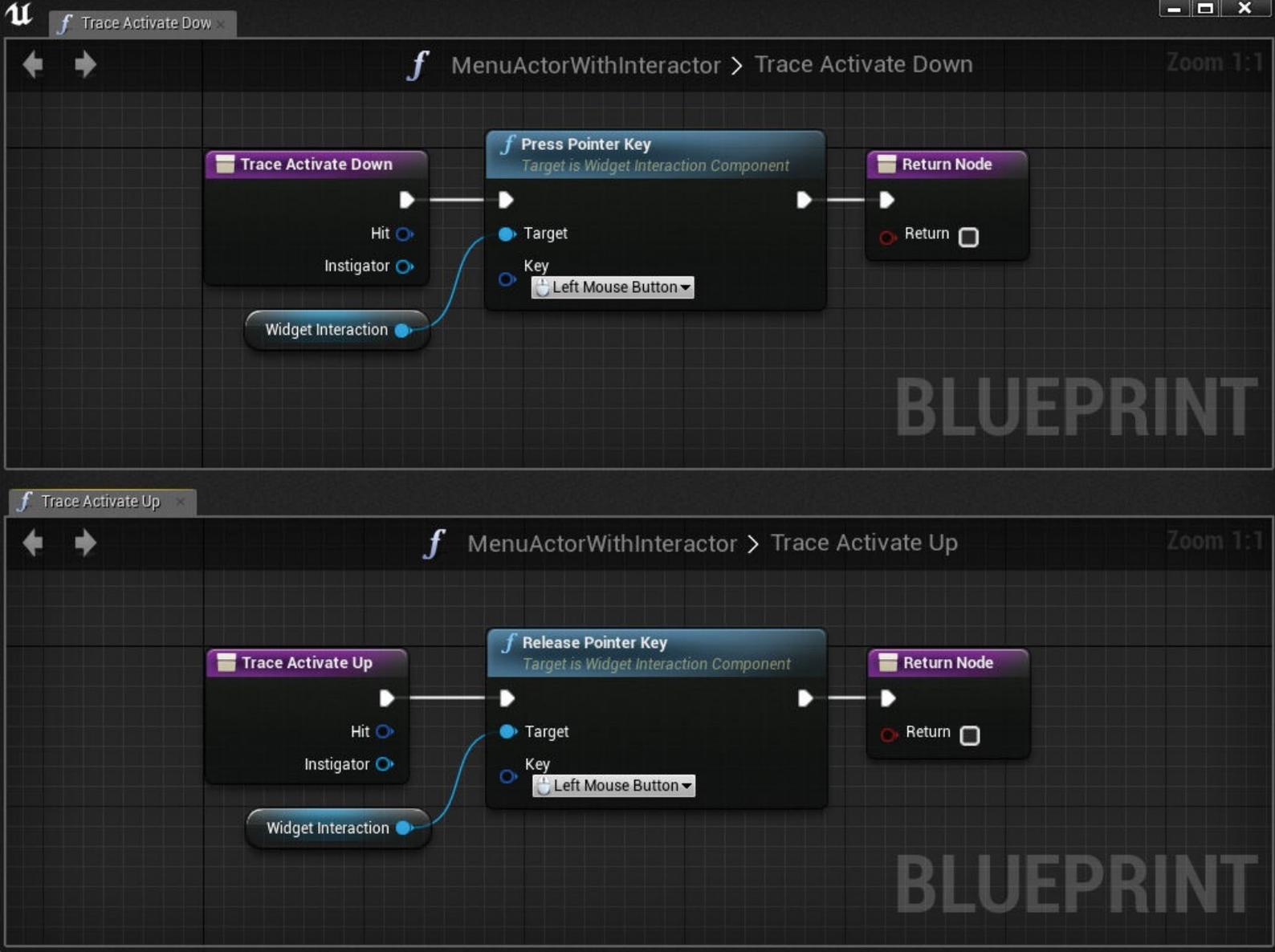


Figure 6.14 Calling the Press and Release functions on the Widget Interaction Component in the menu Actor

14. Repeat steps 12 and 13, but call the ReleasePointerKey function instead of the PressPointerKey.

You now have a functioning menu without even touching the Trace Interaction Component (yay, composability!), and the Trace Interaction Component isn't dependent on any other Component. The only drawback is the possible performance problems from having multiple Interaction Components in the same world and the harder task of handling multiuser interaction, because a single menu won't have multiple Interaction Components affecting it.

Summary

This chapter looked at the need for in-world UI instead of the alternative (head-locked). You looked at the history of Unreal Engine's implementation of this feature.

You also looked at how to create a basic menu and menu interaction system from scratch with the built-in Widget Interaction Component as well as how to integrate similar functionality into the Trace Interaction system developed in [Chapters 4](#) and [5](#).

Exercises

The approach presented in this chapter works well for menus and the like, because often they have a predictable location. However, for a player's HUD and other UI elements whose position the user can control, new problems arise when the menu exists in a 3D space.

The main issue that you will face is occlusion: what to do when the UI intersects with (or is even completely covered by) other in-game geometry.

There are some interesting possible solutions to this issue. You can disable the engine depth test altogether by creating a custom translucent Material and selecting Disable Depth Test in the Translucency tab so that your menus will always render on top of other elements. However, this may create a stereo mismatch.

You could make use of UE4's custom depth functionality and display a more translucent version of the UI when it intersects with other geometry.

Or you could—well, do whatever you want. This is where making games becomes fun! Start thinking of interesting solutions to these problems and choose the best for your game or experience.

Another popular menu choice, if you have motion controllers, is to attach a menu to one hand and interact with this menu with the other. To achieve this, attach your created menu Actor to one of your Pawn's motion controllers and your Widget/Trace interactor to the other. (You may need to enable self-collision in your trace function to get this to work.) Try this technique and see which better suits your game.

Chapter 7. Character Inverse Kinematics

VR allows players to inhabit a virtual character's body, and thus replicating players' real-world movements to their virtual characters can be an immersion multiplier if done right.

This chapter discusses techniques built into Unreal Engine for interpolating the user's current pose from known information about the player's location.

Introduction to Inverse Kinematics

As opposed to forward kinematics, where you define each bone's rotation to get your desired output, inverse kinematics (IK) allows you to define an end effector goal and let the system interpolate what the bone rotations need to be to get to that goal.

Many VR headsets give you access to precise head tracking so you know exactly where the user's head is in your virtual world. Without expensive motion capture equipment, however, you do not necessarily know where any other bone is (aside from hands in the case of motion controllers).

In these situations, inverse kinematics lets you extrapolate where the other bones in the skeleton are located because of the predictable way in which bones rotate. The simplest method of IK is two-bone IK (see [Figure 7.1](#)), because the bone rotation can be calculated analytically with some basic trigonometric identities; however, note that once a third dimension is introduced, as is the case in VR, there are an infinite number of possible solutions to a two-bone IK setup. To deal with this, Pole Vectors/Joint Targets are introduced. These allow developers to create a preference for how the bones move. In UE4, Joint Targets define a point in space that, along with the direction from the root bone to the IK target point, generates a plane (the normal of this plane is the cross product of these two directions). This plane is then used to simplify the IK problem back down to two dimensions.

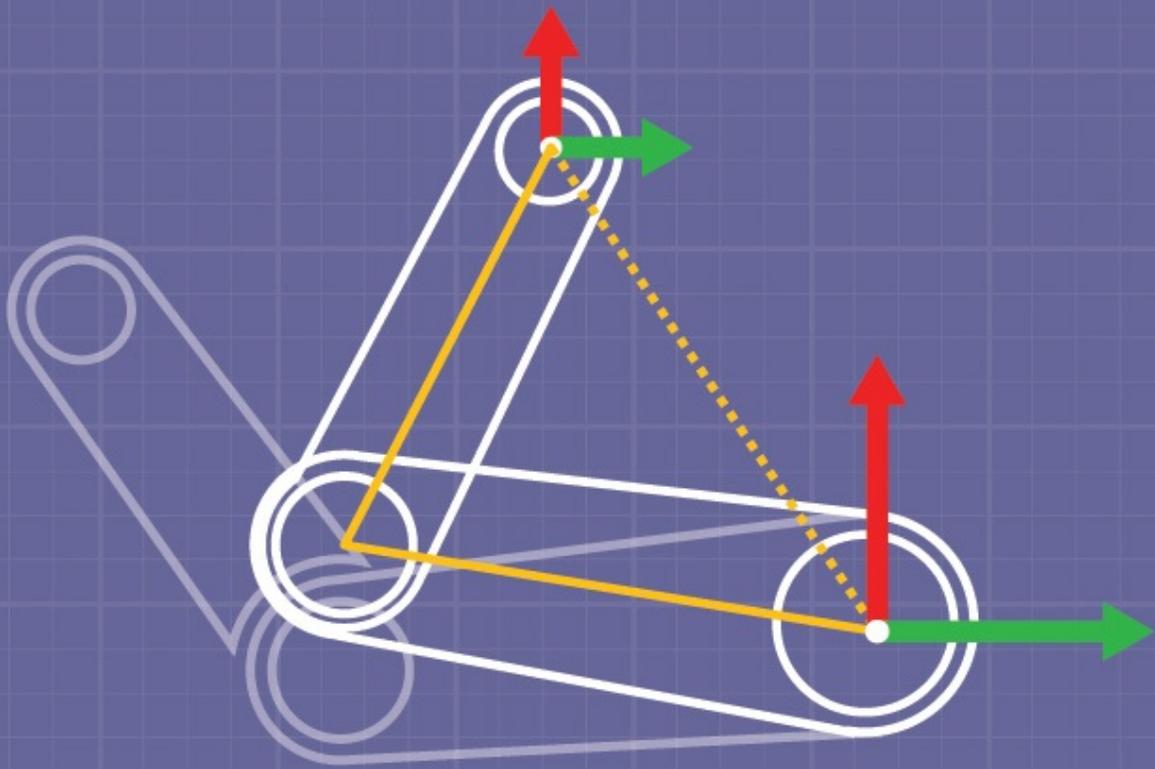


Figure 7.1 Two-bone inverse kinematics. Because the bone lengths are known, the delta from the previous bone rotations can be calculated through trigonometric identities.

UE4 offers another method for inverse kinematics that comes out of the box. This is Forward and Backward Reaching Inverse Kinematics (FABRIK). Unlike the two-bone IK implementation, FABRIK does not restrict the number of bones that can exist in the inverse kinematic chain. To do this, FABRIK takes a slightly different approach: rather than being analytical, like two-bone IK, FABRIK must be calculated iteratively by traversing up and down the chain of bones and converging toward a solution.

FABRIK first sets the last bone in the chain to match the position of the end goal, then works its way back up the chain, moving each bone toward the solution while attempting to maintain the bone's length and a straight line to the previous bone (see [Figure 7.2](#)). Once this loop reaches the very first bone, it will be moved out of place; however, because this is the root of the chain and shouldn't move from its original position, the algorithm is then run again, this time in reverse (hence Forward and Backward IK). By default, in UE4 this process is repeated ten times, but this iteration count can be changed on any FABRIK node in an animation graph.

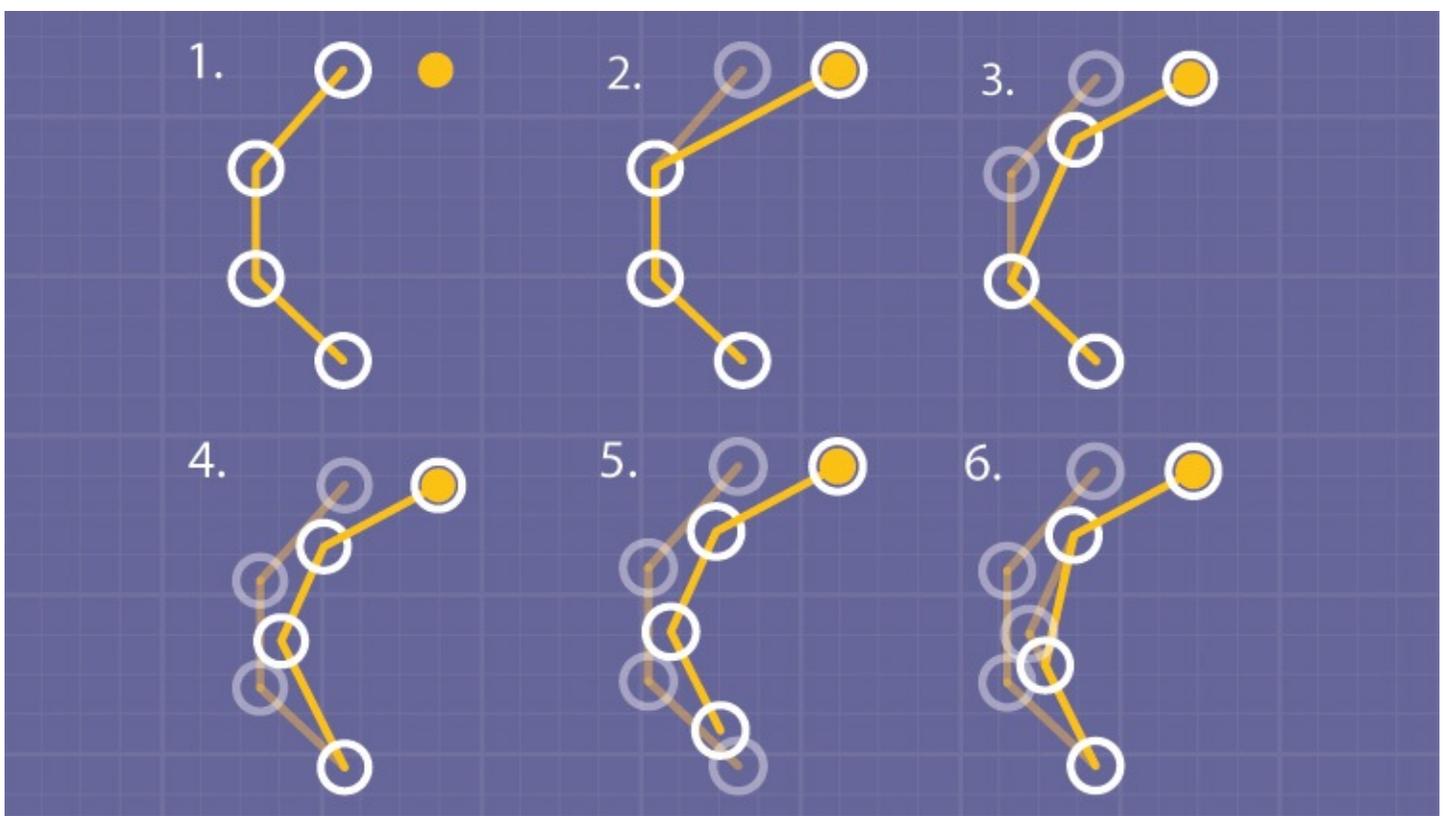


Figure 7.2 Forward and Backward Reaching Inverse Kinematics. 1: The initial bone state. 2: The end bone is moved toward the goal (yellow circle). 3–5: The next bone in the chain is then moved along the line connecting this bone to the previous bone while maintaining the initial bone length. 6: Because the root bone has been moved, it is moved back to its original position and the process is done in reverse.

Setting Up Head IK

To test out UE4's IK system, we will create a basic head IK system that allows the in-game character to position its head bone at the player's in-world head location as well as bend its torso to match. Note that this works only if a player is standing still and not walking around the room, because the player's torso is not tracked.

Before starting, create a new project based on the Third Person Template with no starter content. This allows you to use the player model that comes with that template to test your IK Blueprints.

Mirror Creation

To know if your IK system is working, it would be nice to see what your in-game character looks like while playing in VR. To do this, you can create a simple mirror Material and render Texture:

1. Create two new folders in the Content Browser, the first named Materials and the second named Textures.
2. Inside the Textures folder, add a new Render Target called MirrorRenderTarget (Add New → Materials & Textures → Render Target).
3. Set both Size X and Size Y to 512 (see [Figure 7.3](#)).

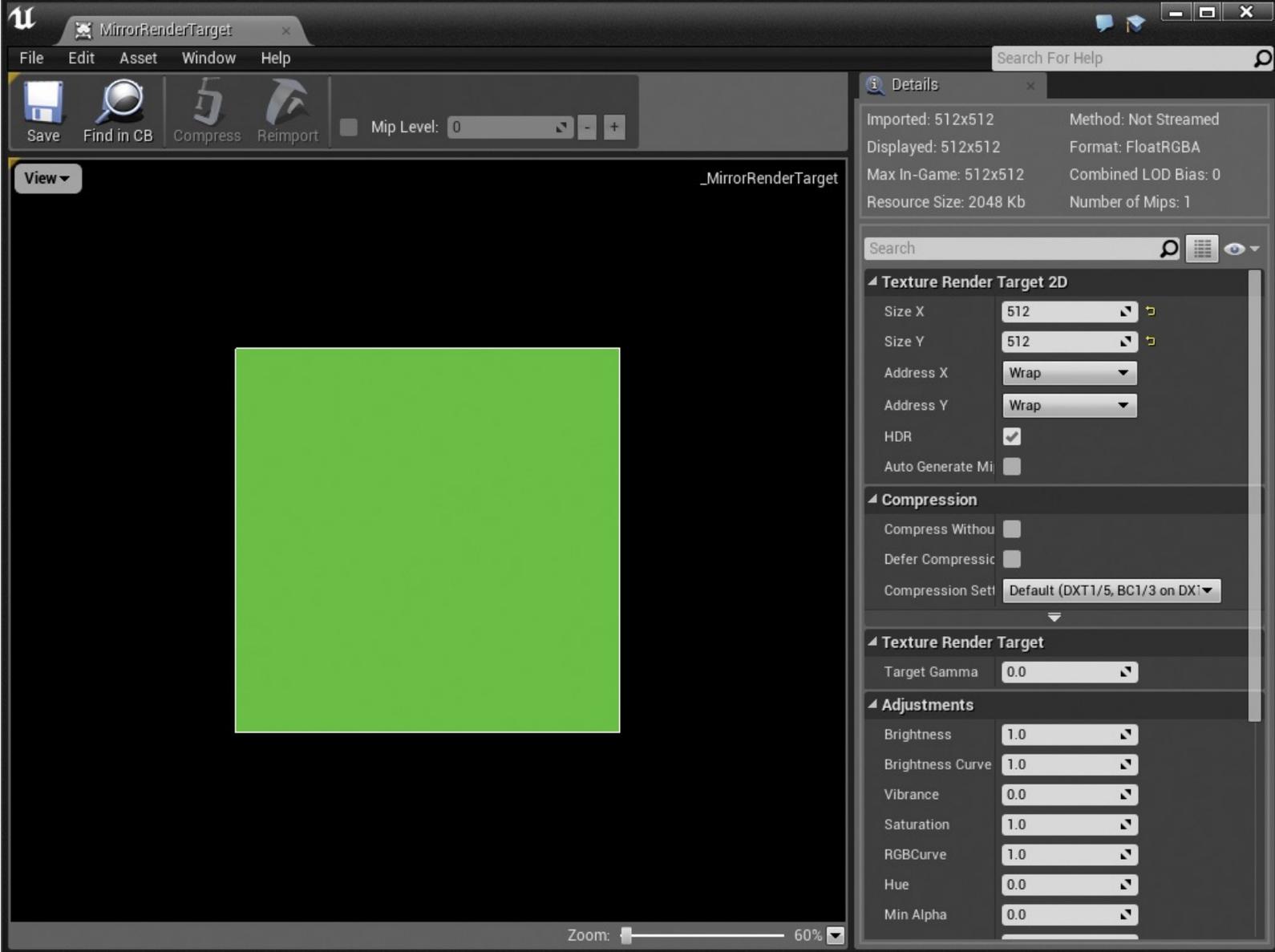


Figure 7.3 Mirror Render Target

4. Inside the Materials folder, create a new blank Material called MirrorMat.
5. Open this Material and set its Shading Model to Unlit.
6. From the Content Browser, drag the MirrorRenderTarget into your MirrorMat.
7. Create a new TextureCoordinate node (U + click the graph) and set its U Tiling to -1.
8. Pass this TextureCoordinate into the UVs input of your TextureSample (see [Figure 7.4](#)). This flips your Mirror Texture to make it more mirrorlike.

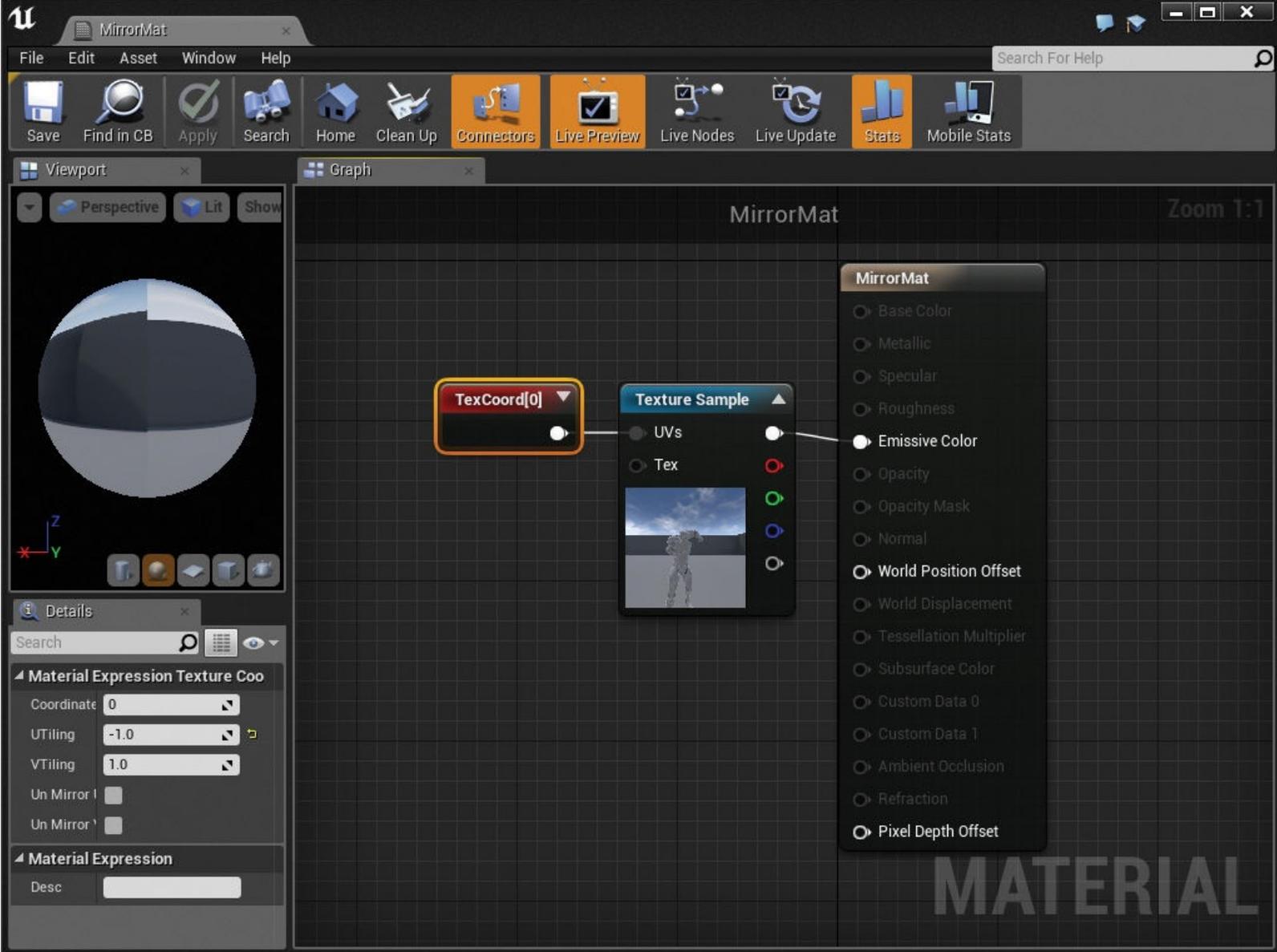


Figure 7.4 Mirror Material

9. Save and close this Material. Head back to the main engine window.
10. From the Modes tab, search and drag in a SceneCapture2D Actor, setting its Location to (X = -650.0, Y = 390.0, Z = 260.0) and Rotation to (Pitch = 0.0, Yaw = -180.0, Roll = 0.0). This positions it in front of the player.
11. Set this SceneCapture2D's Texture Target to the MirrorRenderTarget Texture. This puts whatever this camera captures into this Texture.
12. From the Modes panel again, drag in a basic Cube and set its Location to (X = -640.0, Y = 390.0, Z = 260.0) and Scale to (X = 0.01, Y = 2.0, Z = 2.0); see [Figure 7.5](#).

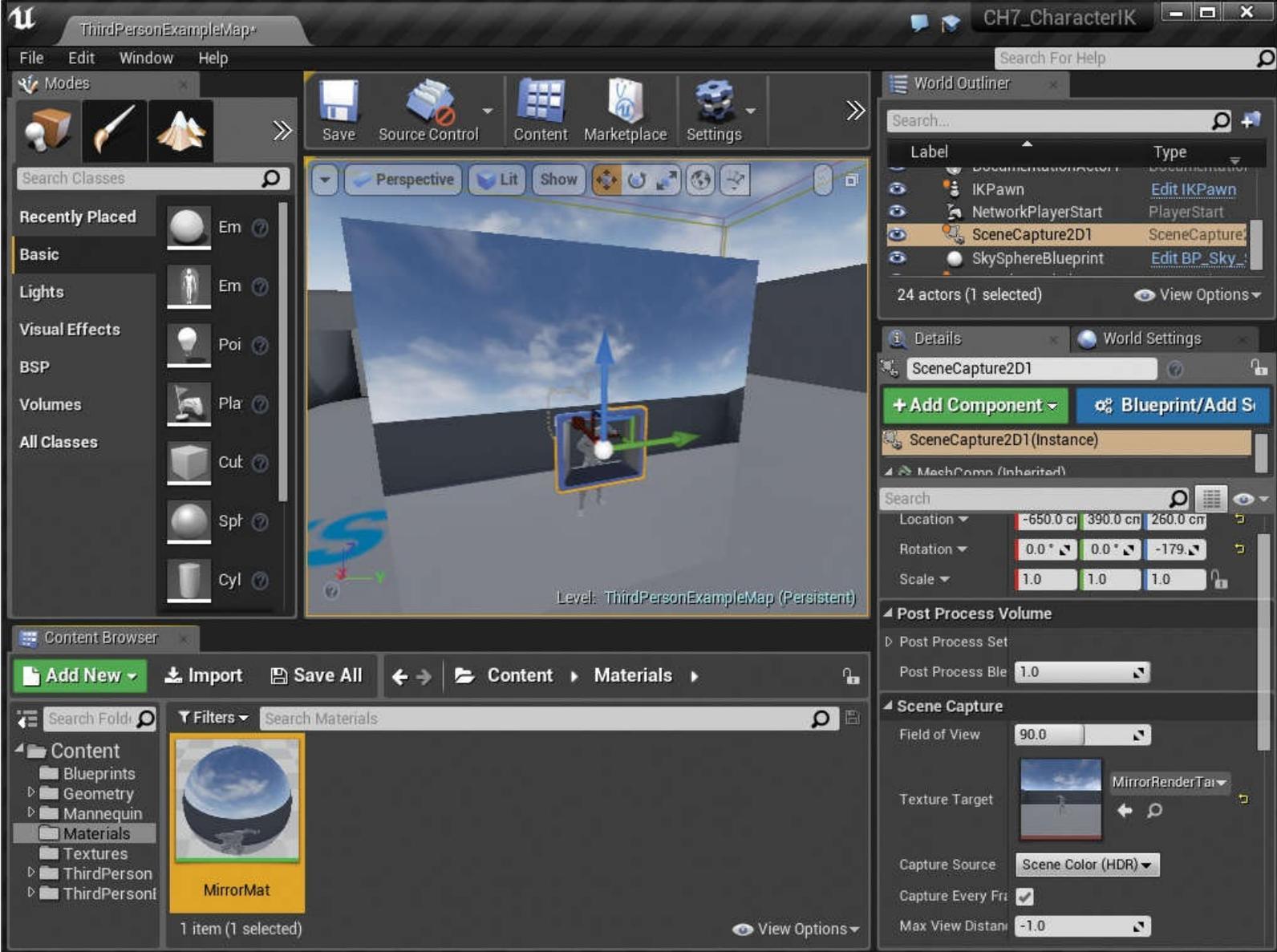


Figure 7.5 Scene capture and mirror mesh

13. Apply the MirrorMat to the Cube's Material Element 0.

IK Pawn

Before the IK is set up, you need a Pawn to hold the Components you reference in the Animation Blueprint:

1. Create a new folder in the Content Browser named Blueprints.
2. Inside the Blueprints folder, create a new Animation Blueprint (Add New → Animation → Animation Blueprint), then select the UE4_Mannequin_Skeleton option in the Animation Blueprint creation dialog (see [Figure 7.6](#)). This will hold all the IK logic that will be used later.

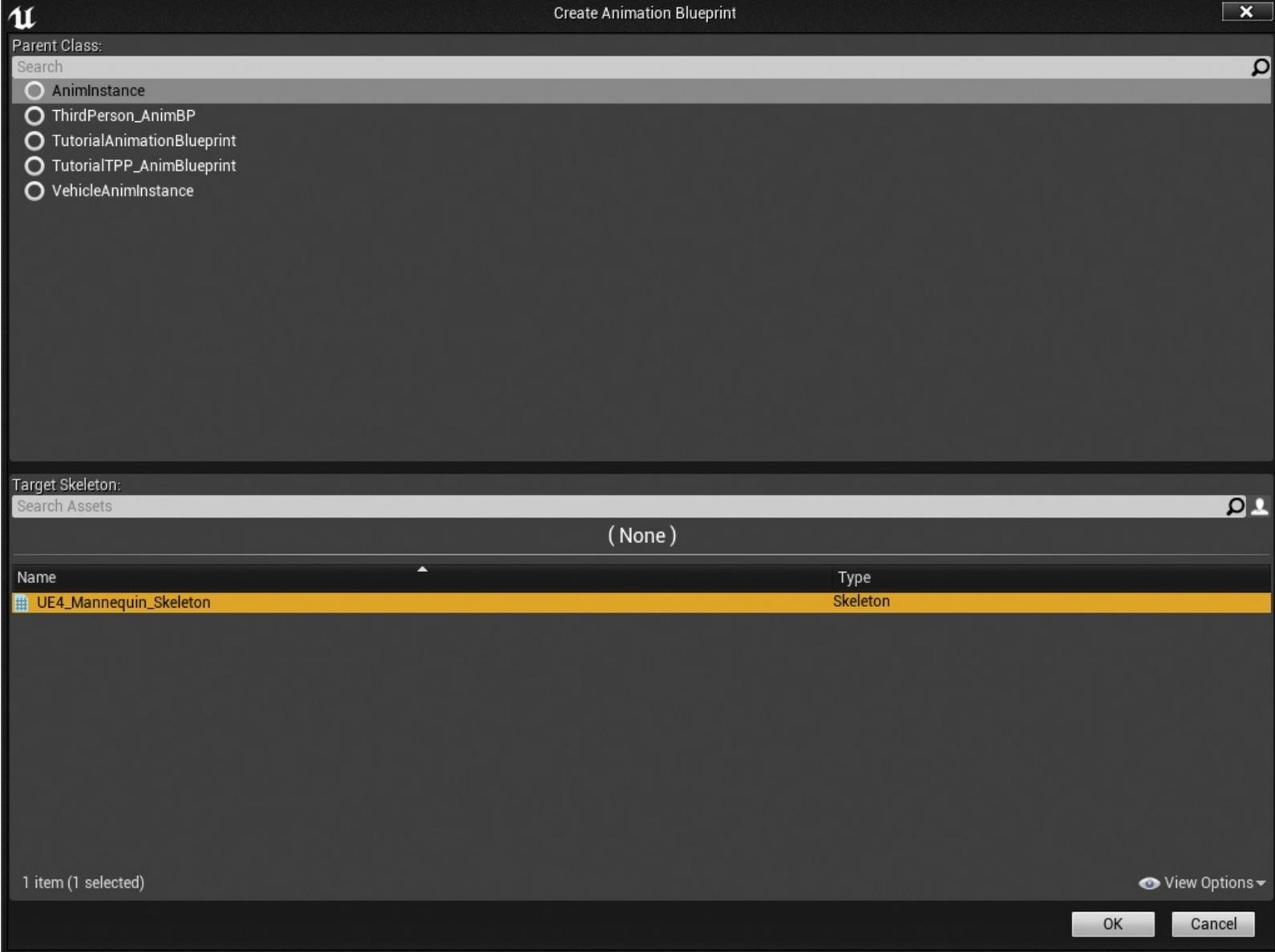


Figure 7.6 Animation Blueprint creation

3. Name this Animation Blueprint IKAnimBP.
4. Inside the Blueprints folder again, create a new Pawn named IKPawn.
5. Open this new Pawn and add three new Components: a SkeletalMesh, Camera, and Scene.
6. Name the Scene Component CameraRoot and attach the Camera to it.
7. Set the Camera to be at 170 Z height and 10 units forward in the X-axis (see [Figure 7.7](#)). This will affect the height of the Camera for testing with a monitor; however, because the Camera's position is reset to its root component when a VR game is launched, the Camera will be in the appropriate position for VR.

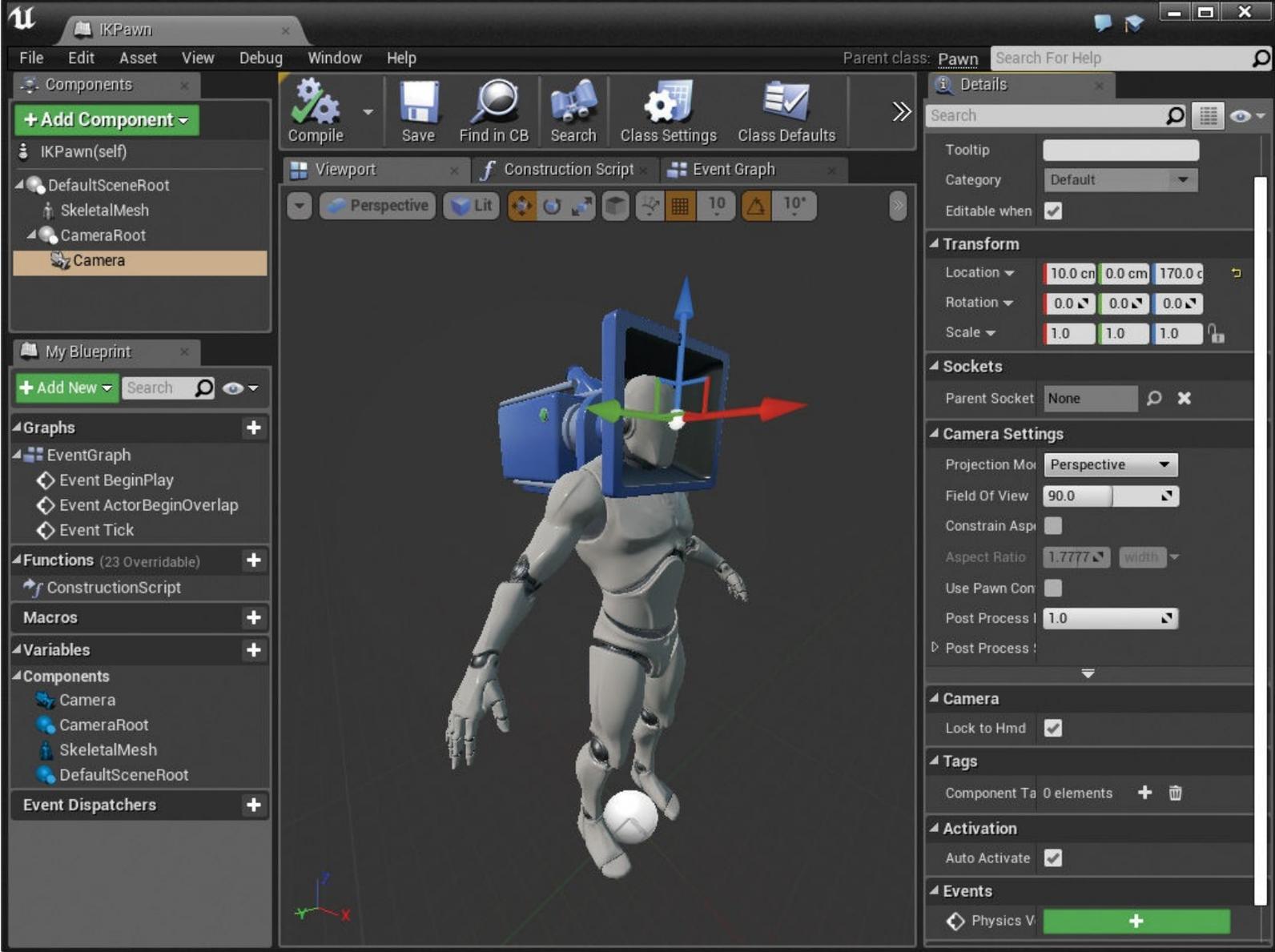


Figure 7.7 IKPawn simple Camera setup

8. Rotate the SkeletalMesh -90 on the Z-axis. This ensures that the mesh faces down the X-axis.
9. Set the SkeletalMesh's Skeletal Mesh property to the SK_Mannequin from the Third Person Template.
10. Set the SkeletalMesh's Anim Class to IKAnimBP.

Head IK Animation Blueprint

To apply the IK to the skeleton of the template character, you will use the FABRIK system because there are more than two bones from the head to the pelvis in this skeleton. Although you can chain two-bone IK nodes to get a similar effect, this can be a pain; the FABRIK node will work well for the head IK.

1. Open the IKAnimBP Animation Blueprint and create a new variable named HeadWorldTransform of type Transform.
2. Set the HeadWorldTransform's default Location value to 160 units in the Z-axis and the default Rotation to 90 degrees in the Z-axis. This ensures that the head faces forward in preview windows.
3. Drag off of the TryGetPawnOwner node in the Event Graph and cast to the newly created IKPawn.
4. Drag off of the As IKPawn blue output pin of the cast and get the Camera Component from the Pawn.

5. Create a setter for the HeadWorldTransform variable and attach it to the Cast Succeeded pin of CastToIKPawn.
6. Drag off of the Camera Component getter and call GetWorldTransform, passing in the output to the setter for HeadWorldTransform (see [Figure 7.8](#)). You are using world space transforms in this example because they are easier to work with.

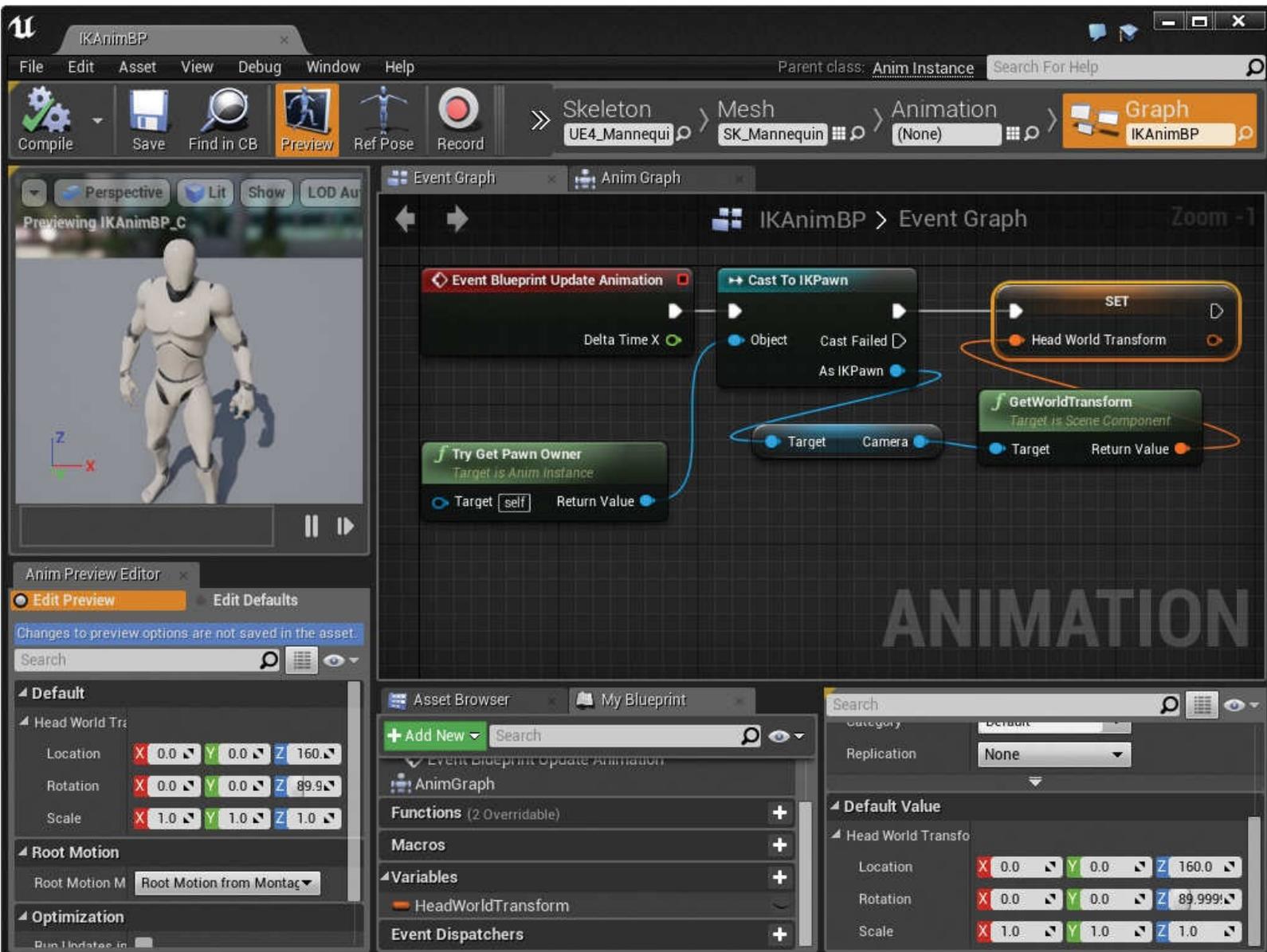


Figure 7.8 Animation Blueprint: head transform setup

7. Head into the Anim Graph and create a new PlayThirdPersonIdle node.
8. Create a new FABRIK node and connect the HeadWorldTransform variable to the Effector Transform input pin.
9. Select the FABRIK node and set the Effector Transform Space to World Space. This is because the Camera transform you are using is in world space.
10. Set the Tip Bone to “head” and the Root Bone to “spine_01” (see [Figure 7.9](#)). The FABRIK node will handle IK on all the bones between the two you just selected.

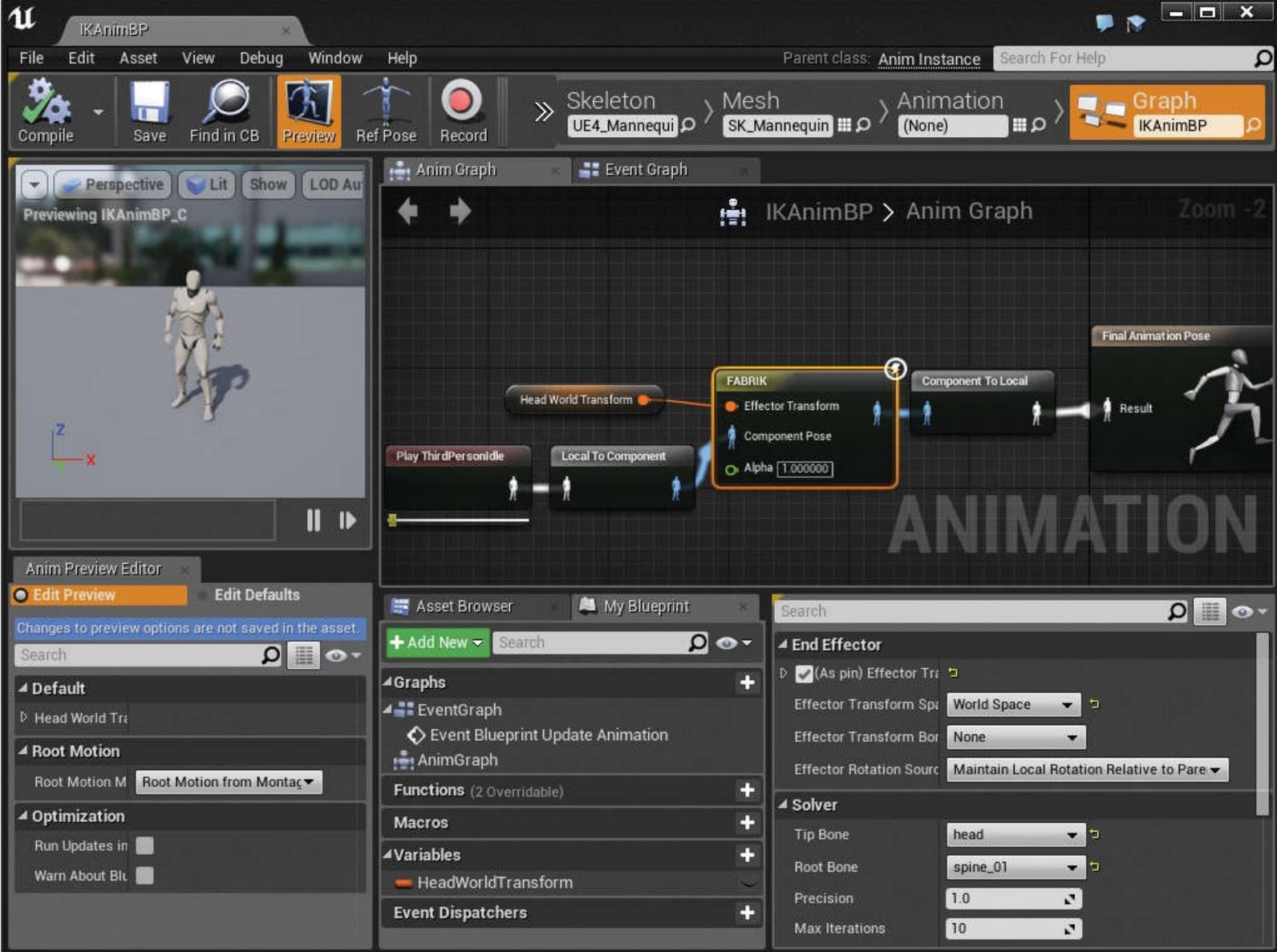


Figure 7.9 Animation Blueprint: simple head IK

11. Drag the output of the PlayThirdPersonIdle animation into the input of the FABRIK node. UE4 will automatically convert from Local to Component space for you.
12. Drag from the output of the FABRIK node into the input of the FinalAnimationPose node. Again, UE4 will automatically transform the coordinates for you. After this step the IK will work; however, it will not account for the rotation of the headset.
13. A quick note before the head rotation is added: In [Figure 7.10](#), notice that the head bone's forward axis (red) is facing up, which means that you need to take this into account when setting this bone's rotation. Create a new Transform (Modify) Bone node in the Anim Graph.



Figure 7.10 UE4 default skeleton head rotation

14. Set the transform node's Bone to Modify attribute to "head."
15. Set the Rotation Mode to Replace Existing and the Rotation Space to World Space (see [Figure 7.11](#)).

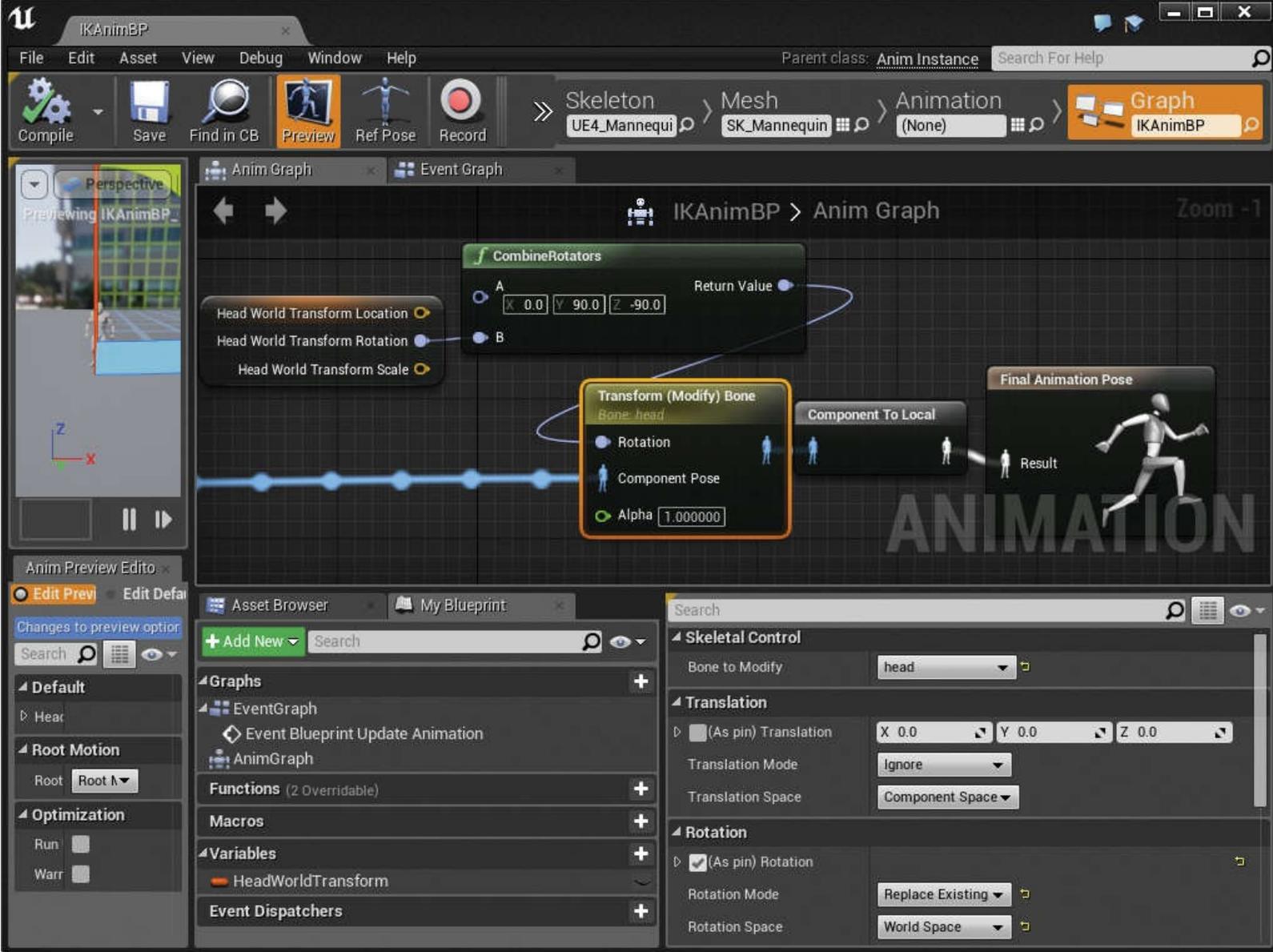


Figure 7.11 Animation Blueprint: head bone rotation

16. Create a new CombineRotators node, passing in the value (Pitch = 90.0, Yaw = -90.0, Roll = 0.0) in the first input pin to account for the initial bone rotation.
17. Create a variable getter for the HeadWorldTransform variable and pass its rotation to the CombineRotators by splitting its output pin, then passing in the blue Rotation pin as the second argument (see [Figure 7.11](#)).
18. Connect the output of CombineRotators to the Rotation pin of Transform (Modify) Bone.
19. Connect the Transform (Modify) Bone between the FABRIK and ComponentToLocal node.

Now that the Pawn and Animation Blueprint are complete, test out your IK by deleting the default Pawn that is in the Third Person Example Map and dragging in your custom IKPawn. Then, set Auto Possess Player to Player 0 (this allows you to use this Pawn in your level without having a game mode set up) and hit Play. If you are using an Oculus Rift, tracking will default to Eye Level, so you may need to set the tracking level origin to Floor (see [Chapter 2](#), “[Head Mounted Display Setup](#)”).

Setting Up Hand IK

If you tested the Pawn and IK from the previous section, you noticed that although the head IK works fairly well, if you have motion controllers, the temptation to move your hands is quite strong. To set it up so that your motion controllers control the hands of the UE4 mannequin, you will use the

simple two-bone IK system of UE4. This is because it is nice to have experience with both the FABRIK and two-bone systems, and the IK between the player's hands and shoulders works quite well with the two-bone system.

Before starting, we will work off of the previous section, so if you haven't completed that, please do so.

Adding Motion Controllers to Your Pawn

Before you set up the IK for your Pawn, you need to add the Motion Controller Components that will be used to track the player's hands:

1. Open the IKPawn from the Blueprints folder.
2. Add two new Motion Controller Components, the first named MotionController_L and the second MotionController_R.
3. For MotionController_R, set the Hand variable to Right.
4. Create two new Static Mesh Components to represent the controllers. These are mainly for debugging purposes.
5. If you are using Unreal Engine 4.13 or higher, you will have access to the motion controller meshes built into the engine (simply select Show Engine Content in the View options). If this is the case, select the appropriate Static Mesh property for your device for these two newly created Components. (For example, I am using the Oculus Touch, so I selected the OculusControllerMesh Static Mesh.) If you are not using 4.13 or above, simply select a sphere to represent your controllers or import the controller meshes from an external source.
6. Name these meshes appropriately for your controller (in this case, Touch_L and Touch_R were chosen) and attach these to the appropriate Motion Controller Component. Be sure to invert the Y scale on the right mesh if necessary (see [Figure 7.12](#)).

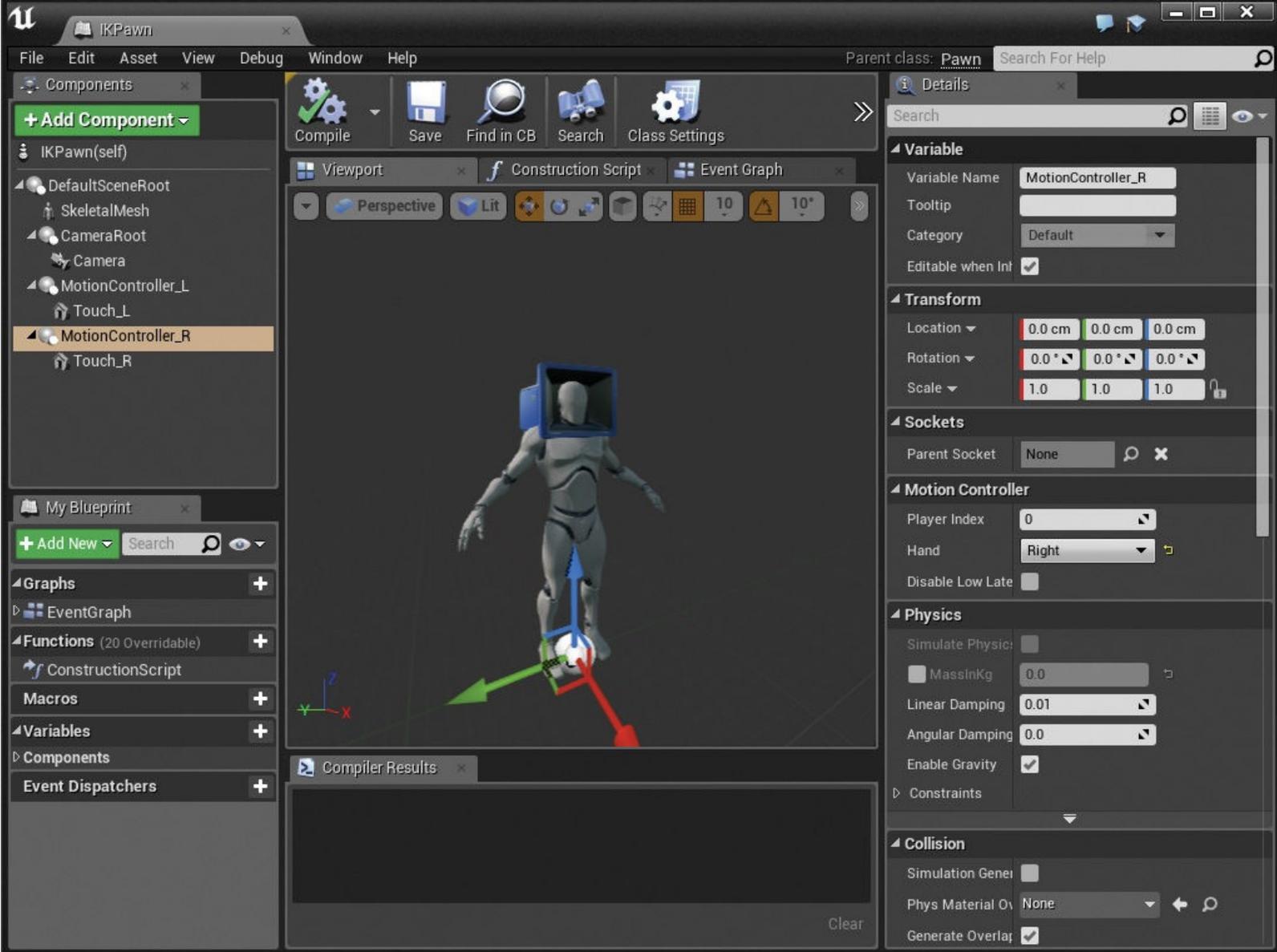


Figure 7.12 Adding motion controllers to the IKPawn

Hand IK Animation Blueprint

Your IKAnimBP class already implements a FABRIK system for the head and spine IK, so now it is time to implement a two-bone IK system for both arms of your skeleton:

1. Open the IKAnimBP Animation Blueprint and create two new variables, `LeftHandWorldPosition` and `LeftHandWorldRotation`, of types `Vector` and `Rotator` respectively. For `LeftHandWorldPosition`, set a default value of ($X = 40, Y = 20, Z = 100$). This ensures that the mesh has an appropriate preview pose.
2. Create two more variables named `RightHandWorldPosition` and `RightHandWorldRotation` of types `Vector` and `Rotator` again. For `RightHandWorldPosition` set a default value of ($X = -40, Y = 20, Z = 100$) this time (see [Figure 7.13](#)). These will hold the world position and rotation of both motion controllers.

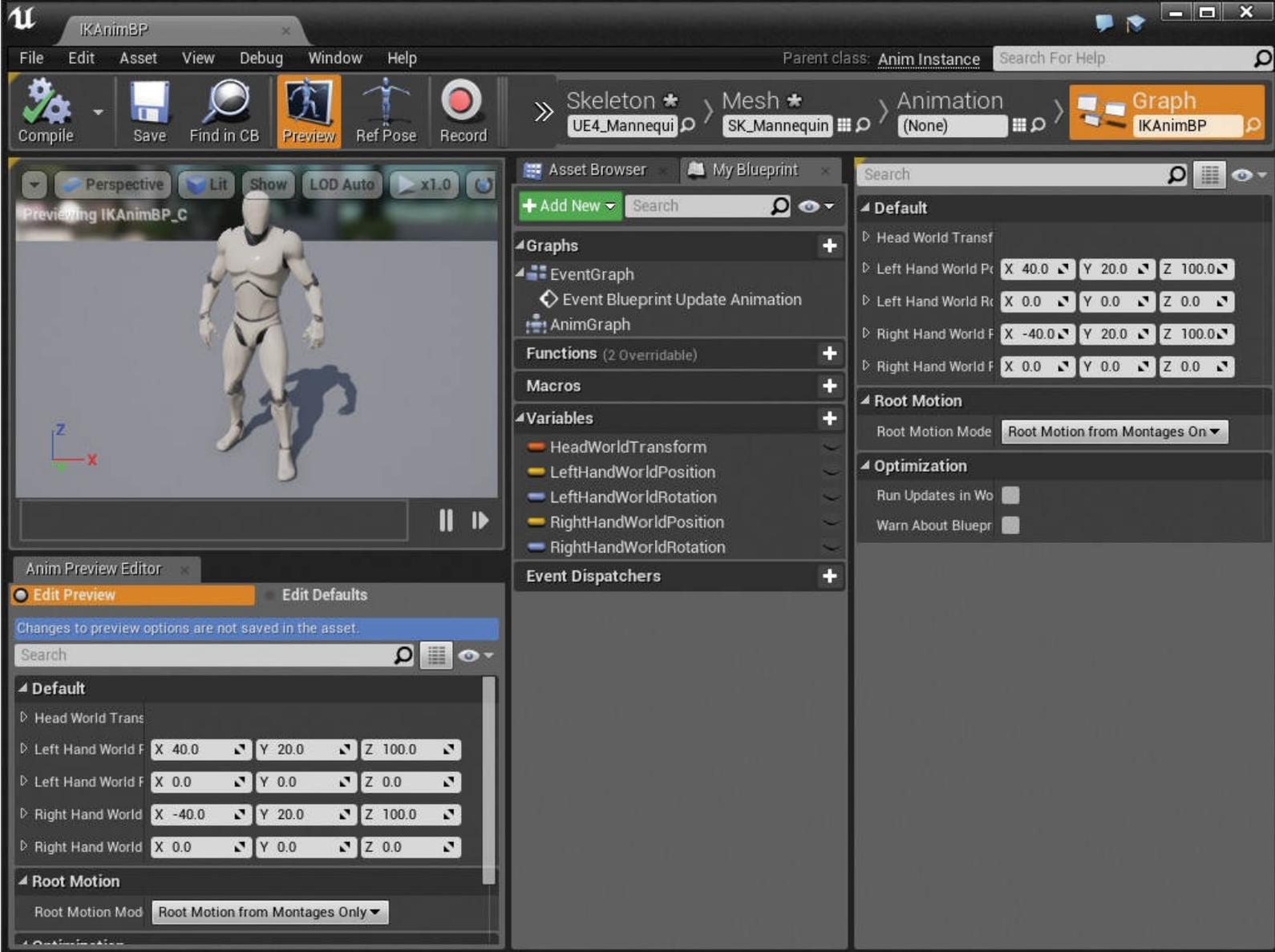


Figure 7.13 IKAnimBP: adding the hand IK variables

3. Before you start adding the IK setup, you will add two Sockets to the Skeleton. This is because the origin point of the motion controllers does not line up with the hand bone location. This means that you will need to account for an offset when you use IK to get to the motion controller position.
4. Open the Skeleton editor by clicking on the Skeleton tab inside Persona.
5. Right-click the hand_l bone in the hierarchy and select Add Socket. Name this Socket hand_lSocket because you will reference it later.
6. This newly created Socket will initially be at the same position as its parent bone. To change this, select it and in the Details panel change the Relative Location to (X = 13.0, Y = -6.0, Z = -3.5). This places the Socket in the palm of the Skeleton's hand.
7. With this new Socket still selected, set its Relative Rotation to 180 degrees in the X-axis.
8. To test the position of this Socket if you are using Unreal Engine 4.13 or above or have access to the controller mesh of your desired motion controller, right-click the Socket and add a Preview Asset of your controller (see [Figure 7.14](#)).

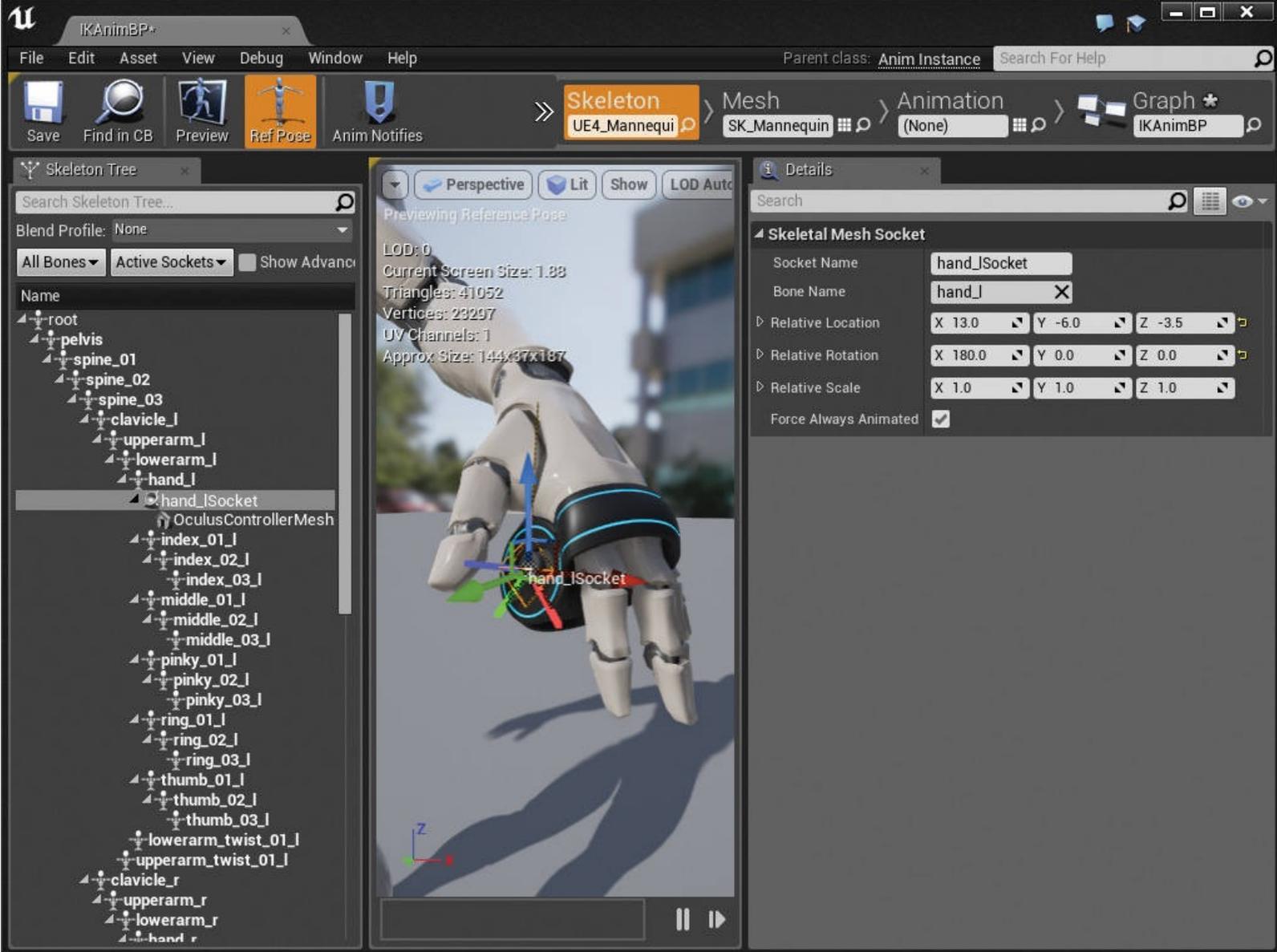


Figure 7.14 IKAnimBP: adding the controller offset Sockets to compensate for the hand bone placement

9. To make sure you have a Socket for both hands, repeat steps 5 through 8, but this time for the hand_r bone with a Relative Location of (X = -13.0, Y = 6.0, Z = 3.5) and a Relative Rotation of 180 degrees in the Z-axis.
10. To take advantage of these newly created Sockets, head back to the Animation Blueprint.
11. Create a variable setter for each of the left- and right-hand position and rotation variables.
12. Drag off of the CastToIKPawn node and get a reference to the MotionController_L, MotionController_R, and SkeletalMesh Components.
13. Drag off of the MotionController_L Component reference, call GetWorldRotation, and pass it directly into the LeftHandWorldRotation setter.
14. For the LeftHandWorldPosition you will need to do a little bit of vector math to subtract the difference between the Socket location and the actual bone location. Drag off of the SkeletalMesh reference and call GetSocketLocation twice.
15. For the first GetSocketLocation pass in the string hand_I Socket as the In Socket Name parameter, and for the second pass in hand_I (see [Figure 7.15](#)). This will get the world location of the hand bone and hand Socket.

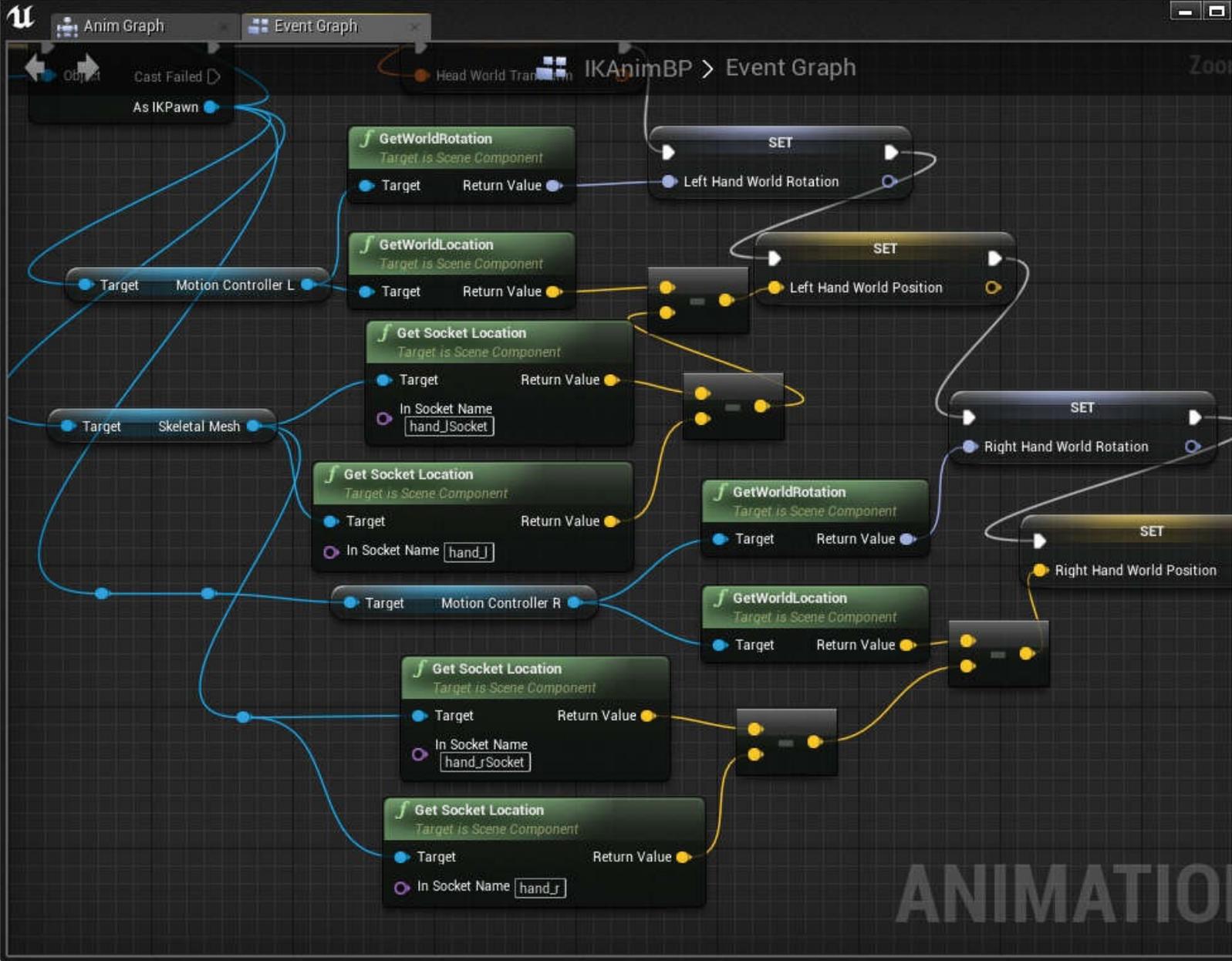


Figure 7.15 IKAnimBP: setting the hand variables and offsetting for controller positions in the IK

16. Drag off of the GetSocketLocation for the Socket and subtract the location of the hand bone. This will give you the relative distance from the hand to the Socket in world space.

17. Drag off of the MotionController_L reference, call GetWorldLocation, and subtract the relative distance calculated in step 16 from this world location.

18. Attach the output of this Subtract node to the LeftHandWorldPosition setter.

19. Repeat steps 13 through 18 but with the MotionController_R reference and the hand_rSocket and hand_r bone. If you need guidance, refer to [Figure 7.15](#).

20. Attach these setters to the HeadWorldTransform setter already in the graph.

To implement the two-bone IK nodes, you need to head to the Anim Graph:

1. Once in the Anim Graph, create a new TwoBoneIK node.
2. Create a getter for your LeftHandWorldPosition variable and attach it to the Effector Location of the two-bone IK.
3. Pass in ($X = 45$, $Y = -50$, $Z = 100$) to the Joint Target Location. This is a point in Component space toward which the IK rotates. This value was obtained through trial and error, so feel free to change it to suit your needs.

4. Select TwoBoneIK and change the IKBone to hand_l and the Effector Location Space to World Space (see [Figure 7.16](#)).



Figure 7.16 IKAnimBP: left-hand two-bone IK

5. Create a new Transform (Modify) Bone node, setting the Bone to Modify property to hand_l, the Rotation Mode to Replace Existing, and the Rotation Space to World Space. This will allow you to rotate the hand because the IK will not handle this.
6. Create a new variable getter for the LeftHandWorldRotation and connect it to the B input of a CombineRotators node.
7. Pass in 180 degrees to the X value of the A input of this CombineRotators node (see [Figure 7.16](#)). This value was obtained by looking at the Skeleton's initial bone rotation as you did in the "[Setting Up Head IK](#)" section.
8. Connect the output of CombineRotators to the Rotation input of Transform (Modify) Bone.
9. Repeat steps 2 through 8, replacing the LeftHandWorldRotation and Position with the right-hand variants, (X = -45, Y = -50, Z = 100) as the Joint Target Location, and 180 degrees in the Z for CombineRotators (see [Figure 7.17](#)).



Figure 7.17 IKAnimBP: right-hand two-bone IK

10. Connect these new IK nodes between the Transform (Modify) Bone of the Head IK and the Component to the Local transform and you should be able to play in VR and see a full IK upper body (see [Figure 7.18](#)).



Figure 7.18 Upper-body IK example

Summary

In this chapter, you looked at a simple way to get started working with IK for VR in Unreal Engine. You looked at the two different types of IK offered natively, and you created a simple head and hands IK system that allows players' upper-body movements to be replicated to their in-game characters.

Exercises

Now that you have a taste of what IK can do, experiment with the values provided in this chapter and tune them to better suit your content.

If you are feeling adventurous, research some full-body IK setups. The main challenge here is to find a heuristic to detect when a player is bending his or her legs rather than simply leaning over.

If you are feeling less adventurous but still would like a full-body IK solver, there are a few middleware companies that provide full-body IK setups for UE4 (such as IKinema or the community Full Body IK Plugin).

Chapter 8. Motion Controller Interaction

This chapter shows you a modular way to interact with game objects in your world with motion controllers. Whether it's picking up objects and throwing them around, pulling levers to open a door, or pushing buttons to trigger an in-game event, this chapter shows you how to create an easily expandable system for these interactions.

When you enter a VR experience with a headset capable of compelling VR, the sense of presence is astounding, yet without an input device that matches that experience it often feels as though you've entered a world governed by a "look but don't touch" mantra.

Luckily, motion controllers solve this problem in most cases; however, players having this freedom just made your job much more difficult. But with the appropriate systems in place, you can make an astoundingly compelling experience.

Why Motion Controller Interaction Works

Since the dawn of modern first-person gaming, the gaming industry has largely settled on an input paradigm that maps certain gamepad events onto the in-game character. The dual analog stick has been a staple for first-person interaction in games for many years and does a good job of translating real-world player actions to the in-game character.

In most cases, you can directly map the player's view to the right analog stick, you can map the player's legs to the left analog stick, and you can map the player's hand actions to buttons on the gamepad.

With VR, however, this paradigm no longer holds true. The right analog stick is taken away to allow the user's physical head motion to control the character's view. The player is, most of the time, no longer moved by an analog stick, because this can cause discomfort (see [Chapter 9, "VR Locomotion,"](#) for more information on this). Finally, many of the hardware providers in VR offer precisely tracked motion controllers, so you now have a way to physically represent the player's hands in your experiences, removing the need for abstract interaction systems that use face buttons on the gamepad because the player can physically interact with the objects in the game world.

What to Look Out For: The Importance of Affordance

Failing a player's expectations in a game is almost never a good thing; however, failing a player's expectations in a VR experience is completely immersion breaking. When a player reaches out to open a drawer but finds that it is in fact a static object, it's a disappointment. The incredible immersion of VR allows players to understand the *affordances* of the objects around them as if they existed in the real world. To make good experiences we should try not to disrupt these perceptions.

Affordance can be described as the properties of an object that suggest its use. As an example, if you were designing a door in a VR experience and gave that door a handle, that signifies to the player that the door can be opened. Blocking this expectation can be immersion breaking and is something to think about when designing your VR experiences.

Shared Input of the Current Generation of Motion Controllers

When creating motion controller interactions, try to consider the target platform to ensure that you fully utilize the available inputs. Each offering of motion controller has slightly different capabilities and best practices. However, luckily, three of the major consumer VR companies—HTC/Valve, Oculus, and PlayStation—have a few shared input paradigms that you can take advantage of to ensure cross-platform compatibility.

All three controllers share a trigger that allows for some sort of analog input (see [Figure 8.1](#)). A trigger can be useful for activating objects or firing weapons in a game. Also, both the Touch and Vive controllers offer a grip button, which can be useful as a dedicated pickup button.

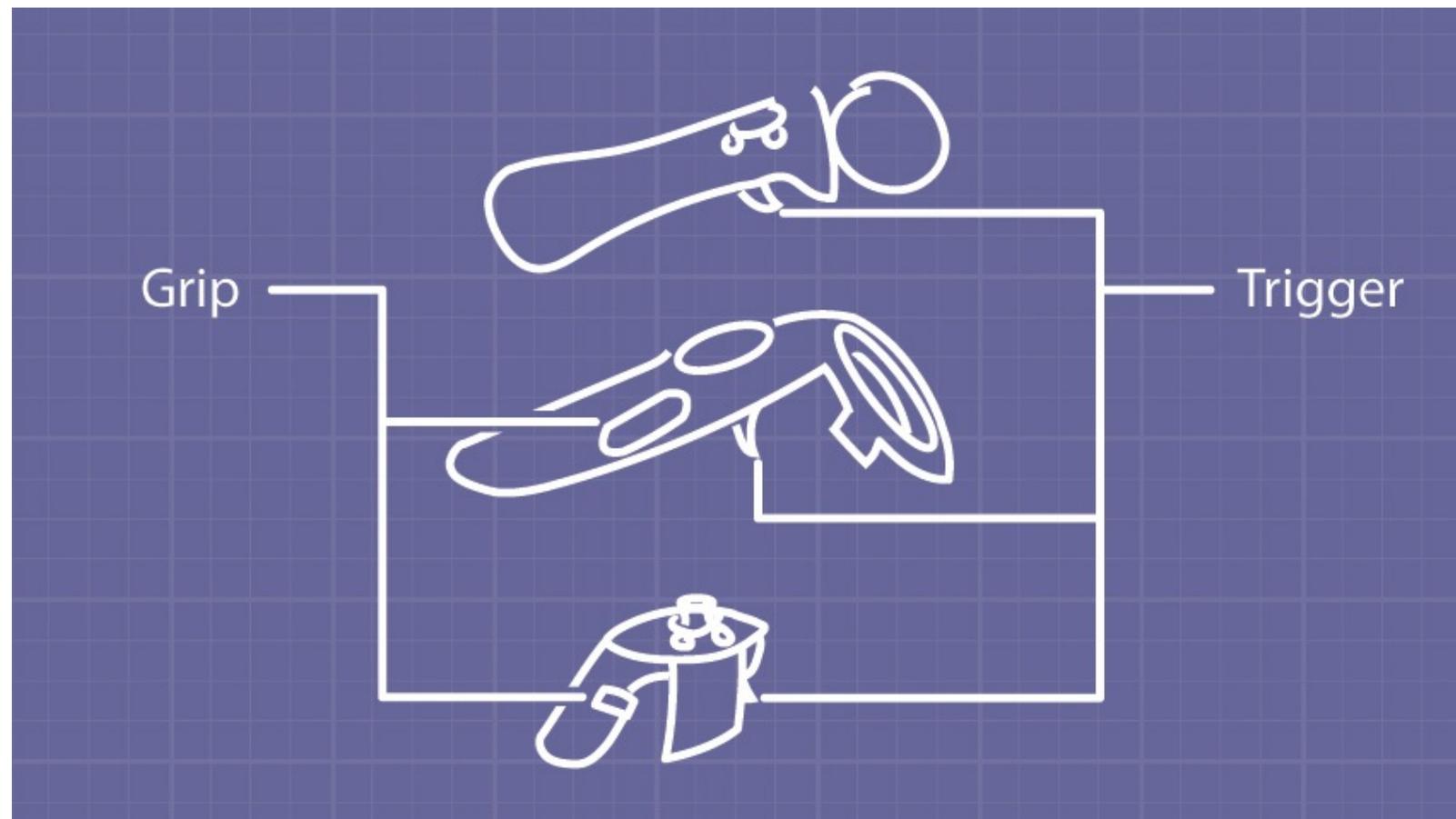


Figure 8.1 The current VR controller landscape’s shared functionality: top, PlayStation Move; middle, Vive; bottom, Oculus Touch

There are more motion controllers coming out frequently (such as the Google Daydream controller). Therefore, this chapter focuses on creating a system that works with the Vive and Oculus Touch. But because of the abstraction layer that UE4 offers developers (all of these controllers use the same Motion Controller Component), you can easily adapt the techniques in this chapter to any device.

Setting Up the World Interaction Project

To set up the project, you need a few objects:

1. Create a new blank Blueprint project with no starter content.
2. Create a new folder named Blueprints, and inside that folder create three more folders named Interfaces, InteractiveObjects, and Components.
3. Inside the Components folder create a new Scene Component named WorldInteractor by clicking Add New → Blueprint Class → Scene Component.
4. Inside the Blueprints folder create a new Pawn Blueprint named InteractionPawn.

5. Open the newly created Pawn and add a new Scene Component called CameraRoot.
6. Create a Camera Component and attach it to the newly created CameraRoot.
7. Create a MotionController Component and leave it at its defaults.
8. Create a Sphere Component and attach it to the MotionController Component.
9. Scale the Sphere to (X = 0.1, Y = 0.1, Z = 0.1). This will act as the visual representation of the controller.
10. Select the Sphere and change the Collision Preset to OverlapAllDynamic (see [Figure 8.2](#)). This stops the Sphere from interfering with the objects you pick up.

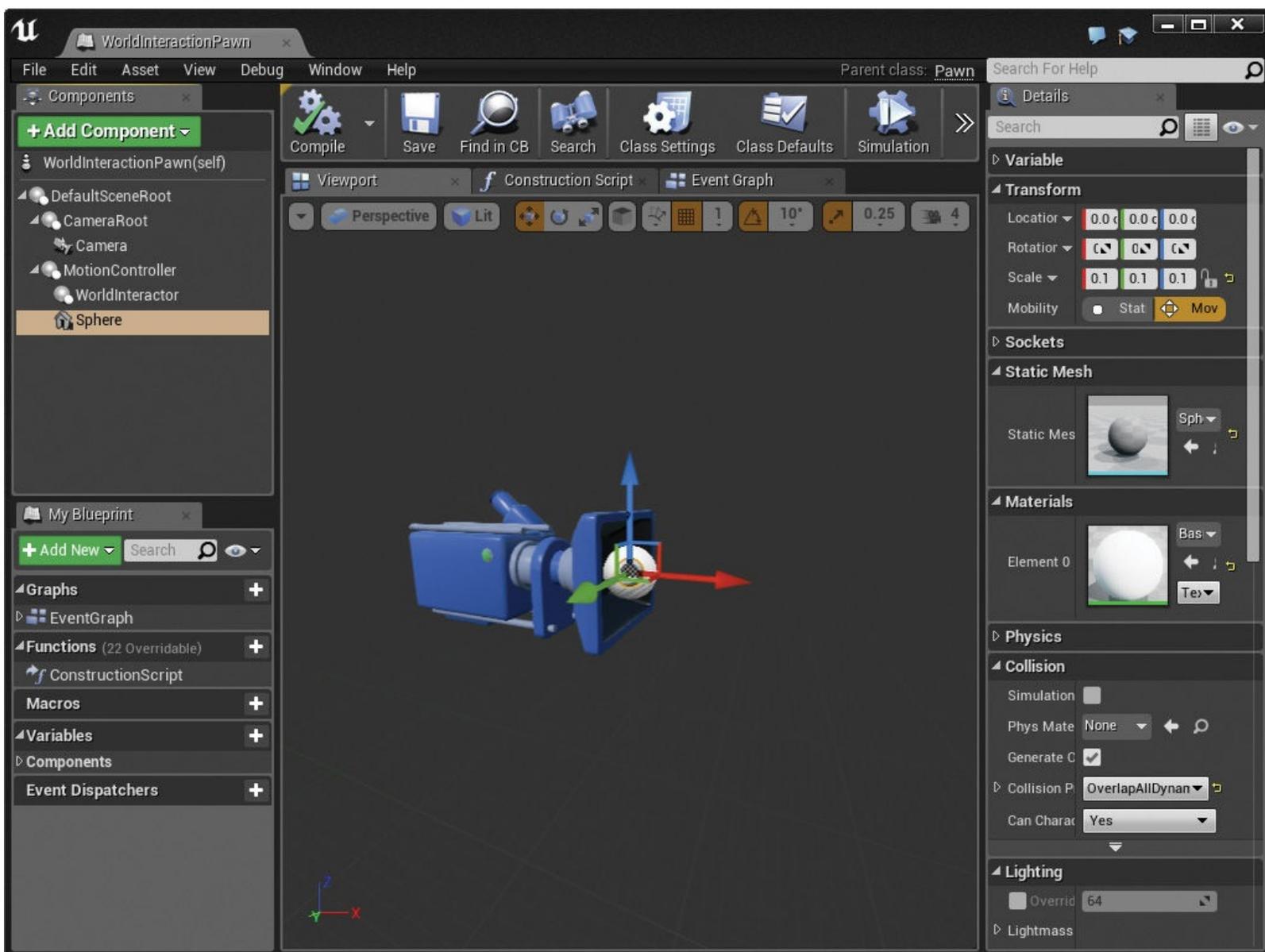


Figure 8.2 World Interaction Pawn Components

11. Create a new WorldInteractor Component and attach it to the MotionController Component.

Note

The MotionController Component defaults to the left-hand controller. To change which hand the Component represents, in the Details panel, change the Hand property under the Motion Controller tab.

Interacting with Objects

To interact with objects, you need two things: an interface (refer to [Chapter 4](#), “[Trace Interaction](#)”) that allows you to talk to objects you wish to interact with, and a Component to detect and manage those interactions.

Creating the World Interaction Interface

Interfaces allow you to communicate with any object that promises to implement the needed functionality without having to know (or care, for that matter) what the object is (see [Chapter 4](#)). This makes them a perfect fit for an interaction system, because the interactor should not care that a button needs to change color when it is pressed or that a certain drawer can open in only one axis. Rather, the interactor should just be told that it needs to interact with something and then trigger the interaction on the appropriate object.

1. In the Interfaces folder create a new Blueprint interface by selecting Add New → Blueprint → Blueprint Interface. Name the newly created interface TraceInteractionInterface.
2. Each function has one input named Interactor of type WorldInteractor and one output of type Boolean named Return. You won't be using this Return value, but it allows all the functions to be in the same section on the implementing objects.
3. Create OnHover and OnUnhover functions and set their inputs and outputs as in step 2. These will be triggered whenever the player hovers over an interactive object (see [Figure 8.3](#)).

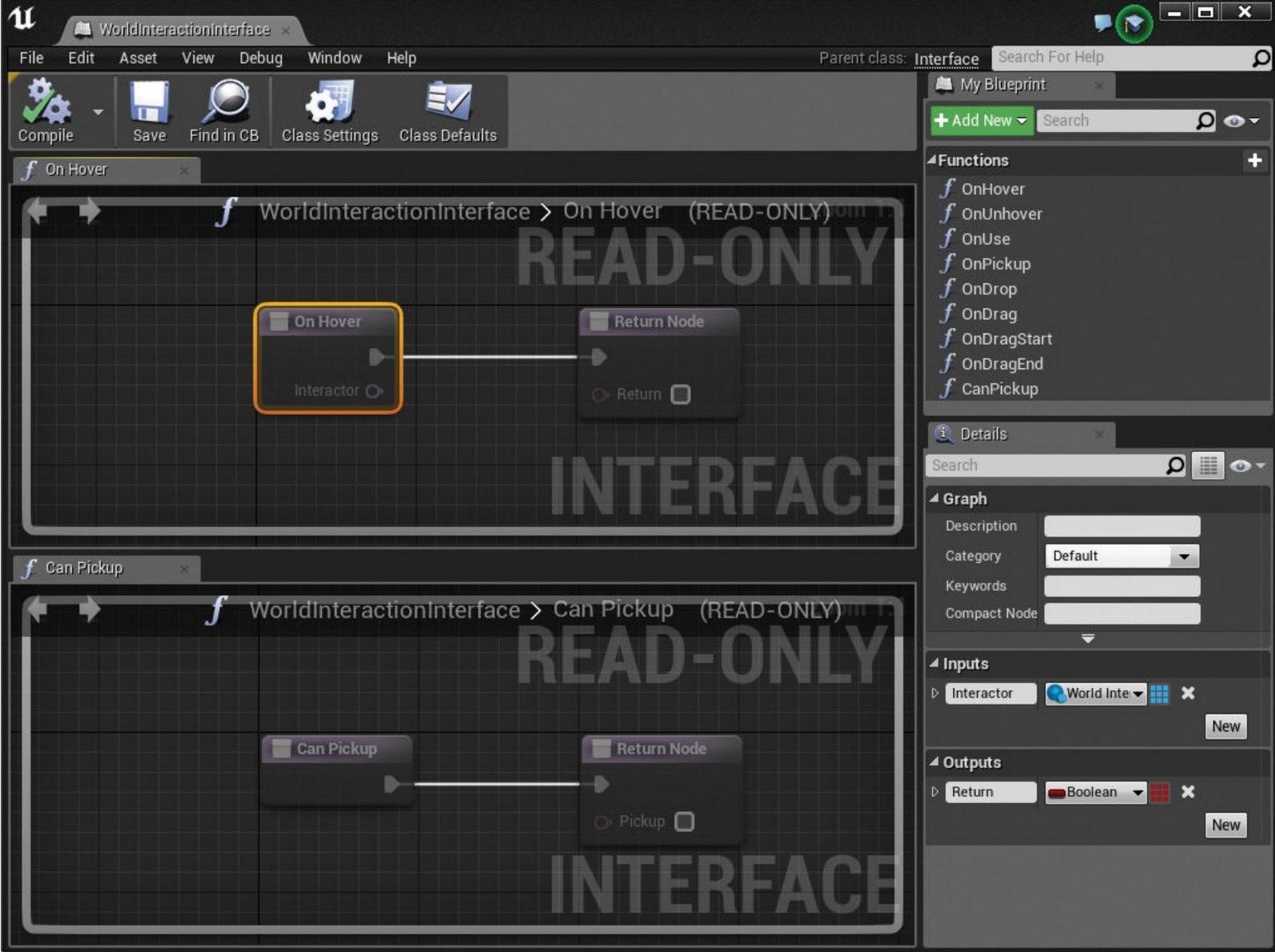


Figure 8.3 World Interaction Interface

4. Create an OnUse function with the same input and output. This is called whenever a player chooses to use a hovered-over object.
5. Create two functions, OnPickup and OnDrop, to detect when to pick up objects with the same inputs and outputs.
6. Create three functions, OnDrag, OnDragStart, and OnDragEnd, which allow you to detect when a player activates an object and then moves the controller while still activating it. Set their inputs and outputs to be the same as the rest.
7. Create a function CanPickup. However, give this function no input variables and a single Boolean output variable named Pickup (see [Figure 8.3](#)).

Creating the Interactor Component

To interact with objects, you need something to hold the logic for finding which object in the world to interact with and choosing which functions from the interactive interface to call. To do this in a modular way, you can use the Blueprint Scene Component created in the section “[Setting Up the World Interaction Project](#).”

Detecting Object Overlaps

First, you need to create the variables the interactor needs so it can remember hovered-over objects and set up a basic sphere overlap to detect objects around the interactor:

1. Open the WorldInteractor from the Components folder.
2. Create a new Float variable called Radius and set its default value to 10. This is the radius of the sphere that the interactor creates to check for interactive objects. If you can't set the default value, be sure to compile your Blueprint.
3. Create a new Actor reference variable called HoveredObject. This is a reference to the object that is currently hovered over.
4. Create another Actor reference variable called PickedUpObject. This is the object that is currently picked up.
5. Create a new Boolean variable called Dragging. The interactor uses this to know when it is dragging an object (see [Figure 8.4](#)).

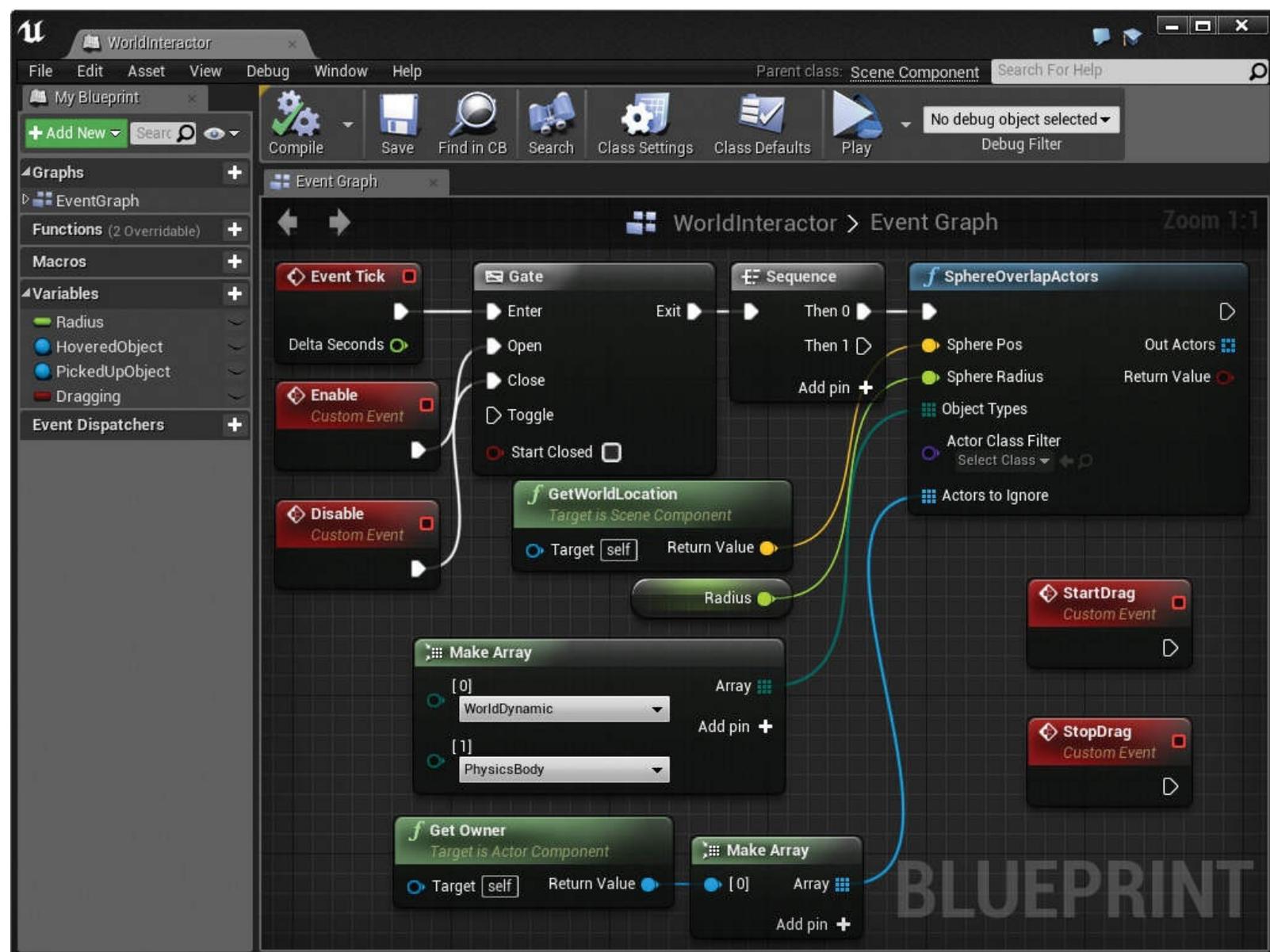


Figure 8.4 World Interactor Component: basic sphere overlap

6. Drag off of the EventTick node and add a new Gate node, and uncheck the Start Closed input pin because you want the Component to be enabled by default.
7. Next add two new custom event nodes. Name the first Enable and the second Disable.

8. Connect the Enable event to the Open pin of the Gate node, and connect the Disable event to the Close pin. This will allow you to enable and disable the interactor whenever you need to do so.
9. After the Exit of the Gate add a Sequence node because you are going to need to do two things every tick and this will make your code neater.
10. Connected to the first pin of the Sequence node, create a new SphereOverlapActors node (see [Figure 8.4](#)). This will allow you to detect the objects you want to interact with within a given radius.
11. Connect a GetWorldLocation node to the Sphere Pos input pin.
12. Connect a new getter for the Radius variable to the Sphere Radius input pin.
13. Connected to the Object Types input pin, make a new EObjectTypeQuery enum array, adding WorldDynamic and PhysicsBody pins. If you want to detect any other types of objects to interact with, this is where you would add them.
14. Create a new Actor array, attaching a new GetOwner node as the first input pin and attaching the output of this array to the Actors to Ignore pin of the sphere overlap.
15. Create two new custom events: StartDrag and StopDrag (see [Figure 8.4](#)). These will be used later to control the dragging.

Alternative to Sphere Overlap

When trying to decide what object the user wants to pick up, there are many ways to get the intent of the user's action.

A sphere overlap with some visual feedback (object highlighting) or physical feedback (a haptic pulse) when the player is able to pick up an object in many cases is all you need for detecting this intent. However, when this fails or isn't an option, there are many alternatives.

One particular one that Epic Games's Bullet Train implements is creating a sphere cast from the controller outward by a vector that is between the controller's direction and the direction from the headset to the controller. This provides a good balance between flexibility and helping the player select the object to pick up.

However, there is no silver bullet with these types of design choices. As with most aspects of game design, you should test to see what interaction system works best for your experience.

Calling the Hover Interface Messages

Next, you will take the hovered Actors from the sphere overlap and find which object to call the hover logic on. In this example, you simply use the first element in the array; however, other techniques, such as finding the closest object within that array, may also be used.

1. Moving to the output of the SphereOverlapActors node, drag off of the Out Actors array pin and use a Get node to get the 0th element.
2. Create a new getter for the HoveredObject variable and call the NotEqual function, passing in the output of the Get node to the second input pin. This allows you to check whether the Component has hovered over a new object.
3. After the SphereOverlapActors node, create a new Branch node and connect it to the overlap's output execution pin, then connect the output of the NotEqual node to the Condition pin of the branch.

Picking Up and Dropping Objects

Picking up and dropping objects is simple in this example, because you leave it up to the individual interactive objects to handle physics and other possible functionality:

1. Create two new custom event nodes and name them Pickup and Drop, respectively. The owner of the interactor calls these events to tell it to pick up the currently hovered-over object.
2. Drag off of the output of the Pickup event and add an IsValid node with the Input Object pin connected to a new HoveredObject variable getter. This ensures that the interactor has a hovered-over object that you can actually pick up.
3. Drag off of the HoveredObject variable getter again, add a call to the CanPickup interface message, and connect the input execution pin of this function to the IsValid output pin of the validity check.
4. After the CanPickup message add a new Branch node, connecting the Condition input pin to the Pickup output pin.
5. Connected to the True output pin of the Branch node, create a new variable setter for the PickedUpObject variable. For the input of this setter attach a new HoveredObject getter (see [Figure 8.7](#)).

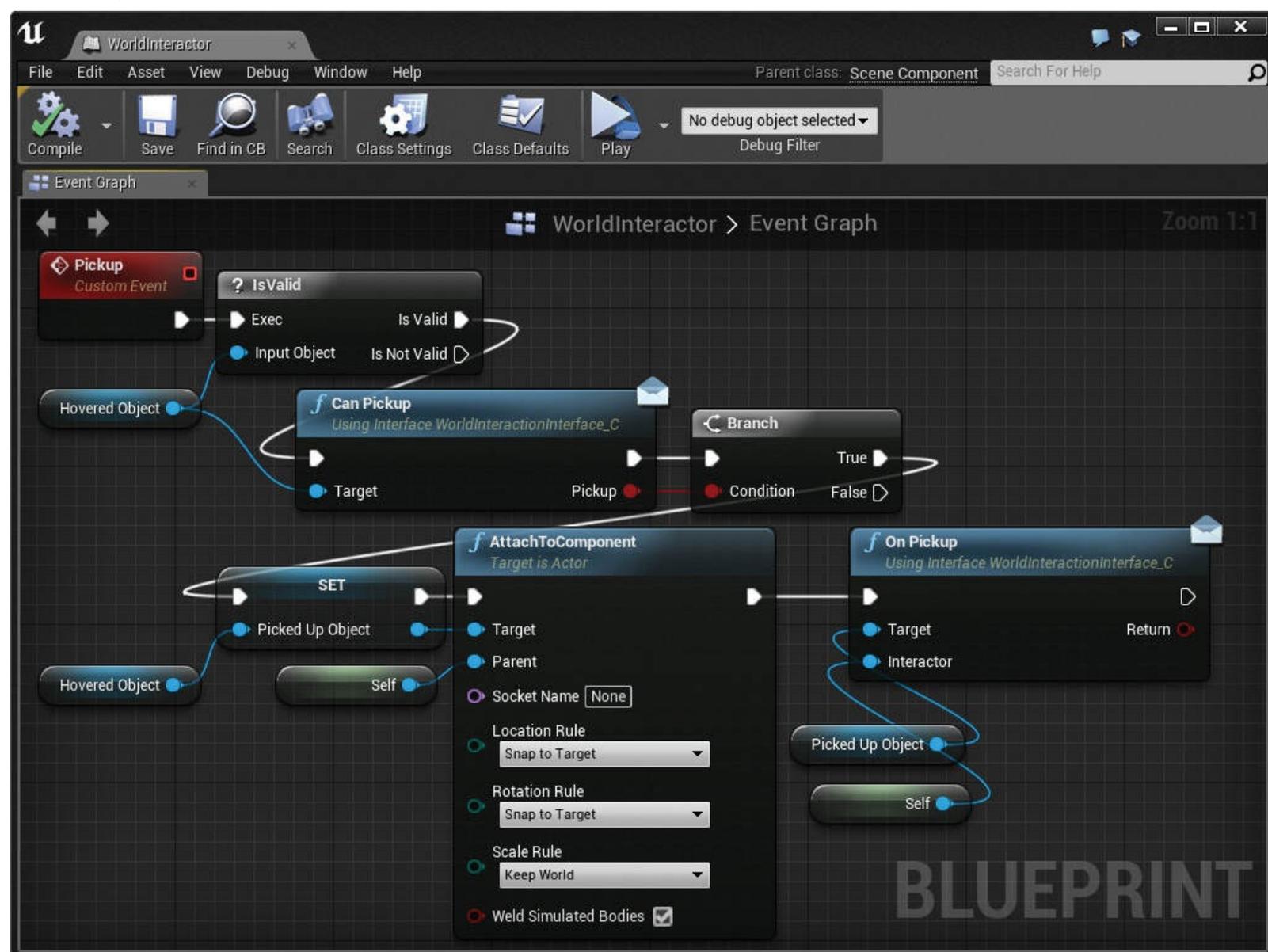


Figure 8.7 World Interactor Component: picking up objects

6. Drag off of the blue output pin of the PickedUpObject setter and add a new AttachToComponent

node.

7. Create a new Self reference getter and connect it to the Parent input pin.
8. For the Location Rule and Rotation Rule inputs, set both to Snap to Target. If you would like to add the ability to enable/disable snapping, you can change these rules.
9. Set the Scale Rule to Keep World, because you do not want the scale of the object that's picked up to be affected by the Pawn or Interactor.
10. After the AttachToComponent, call the OnPickup interface message on a new PickedUpObject variable getter (see [Figure 8.7](#)).
11. To set up the drop event, add a new IsValid check, passing a new PickedUpObject variable getter to the Input Object.
12. Connected to the IsValid output pin of the validity check add a new DetachFromActor node, passing in a new PickedUpObject variable getter as the Target and setting all the transform rules to Keep World (see [Figure 8.8](#)).

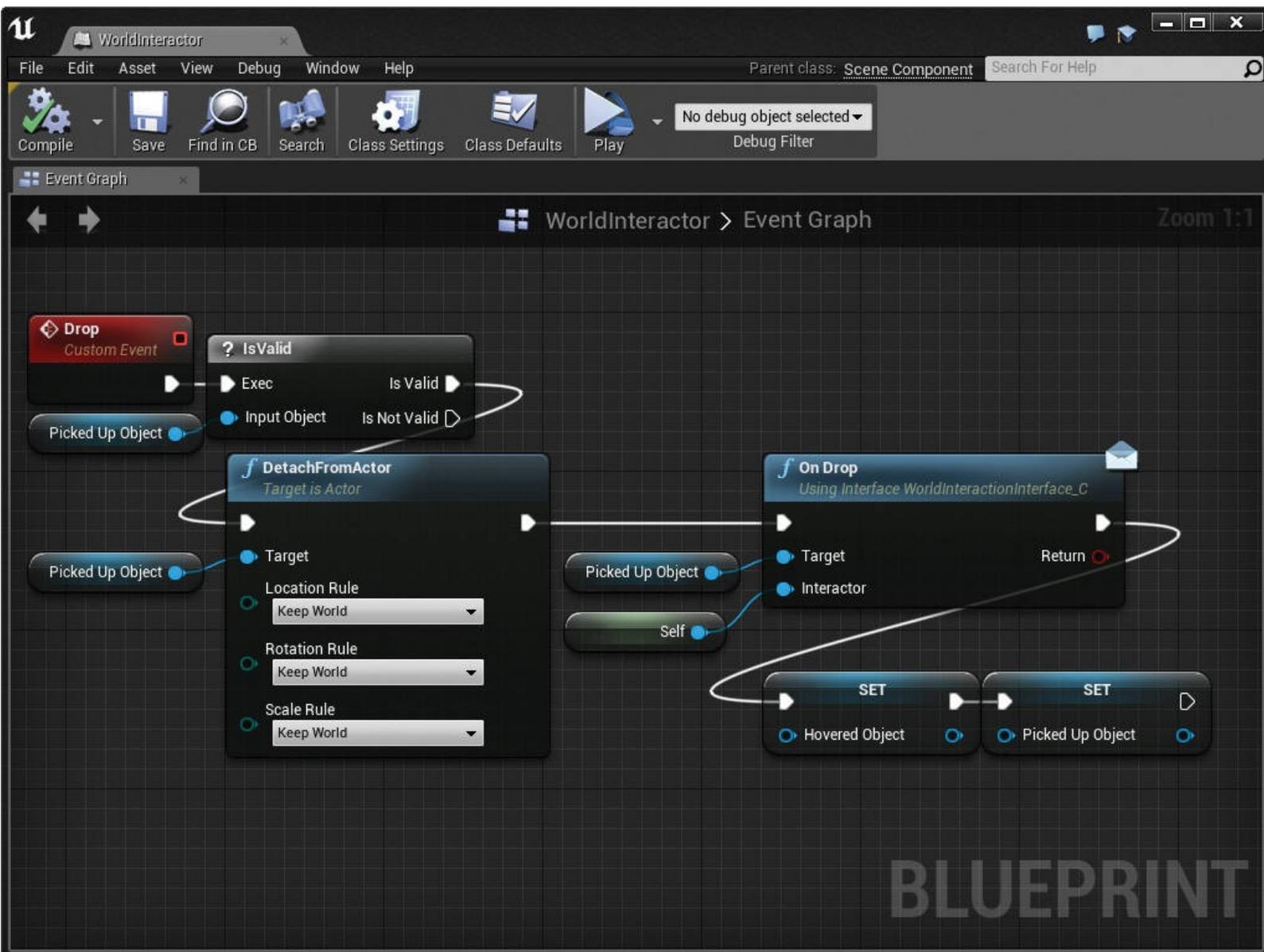


Figure 8.8 World Interactor Component: dropping objects

13. After the DetachFromActor node call the OnDrop interface message on a new PickedUpObject variable getter, passing in Self as the Interactor pin.
14. The last thing to do is clear the HoveredObject and PickedUpObject variables by adding two setter

nodes and leaving the blue input pin blank (see [Figure 8.8](#)).

Using Hovered Objects

Adding the functionality to use hovered objects can be useful if you want to have buttons and other things that can be used when the player hovers over them:

1. Create a new custom event node called UseHoveredObject.
2. Drag off of the output of the new event and call the OnUse interface message on a new HoveredObject variable getter, passing in Self to the Interactor pin (see [Figure 8.9](#)).

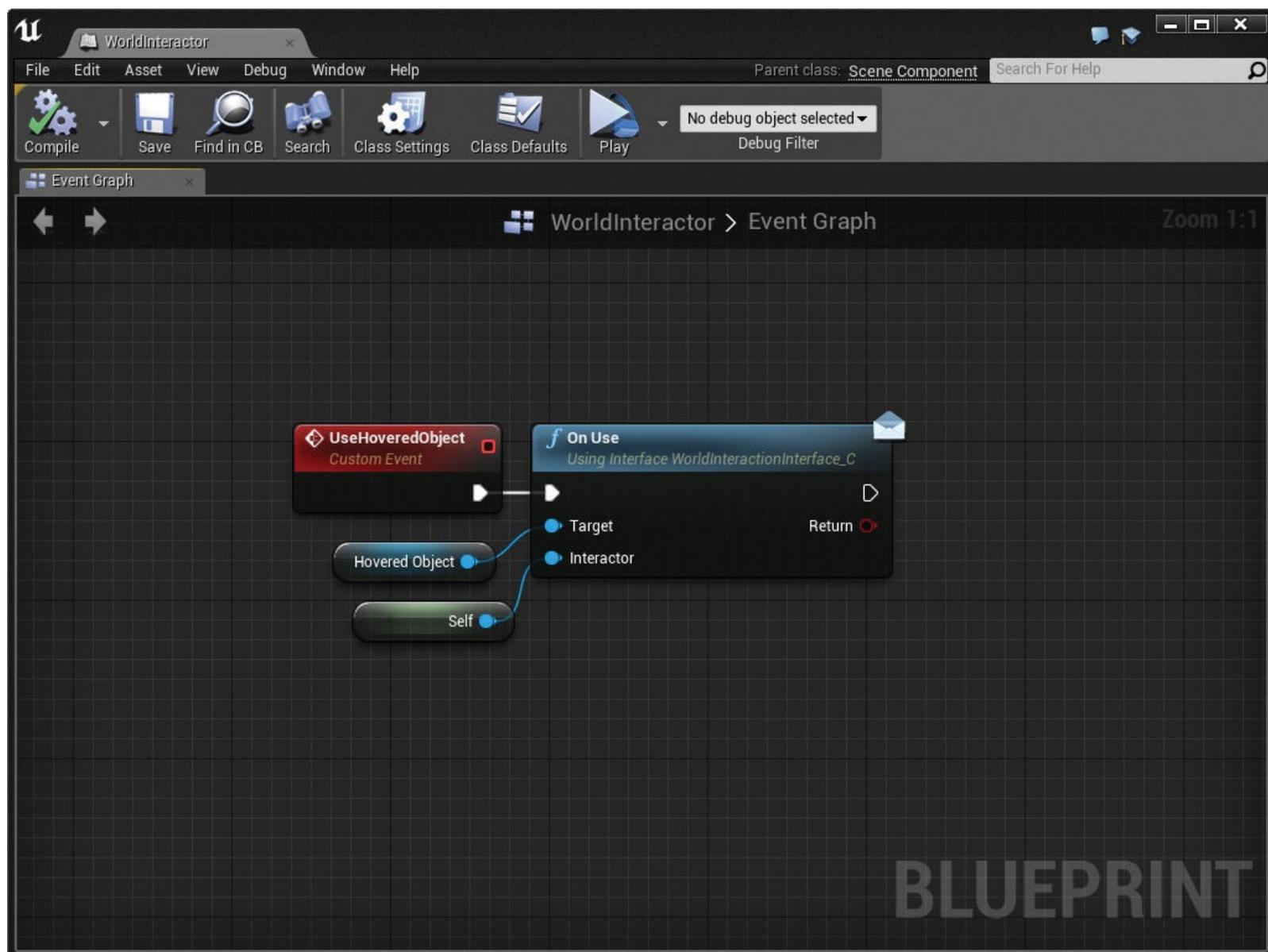


Figure 8.9 World Interactor Component: using hovered objects

Adding Interaction to the Interaction Pawn

To use the newly created Interactor Component, you need to call some of the newly created events from the Interaction Pawn:

1. Open the InteractionPawn Blueprint from the Blueprints folder.
2. You will use two motion controller events. Create new MotionController (L) Grip1 and MotionController (L) Trigger events by right-clicking the Event Graph and entering their names. The grip event represents the side grip button on the Vive controller and the hand trigger on the Oculus Touch controller. These are good candidates for picking up objects.

3. Create a new getter for the World Interactor Component and call the Pickup and Drop events on it.
4. Connect the Pressed execution pin of the MotionController (L) Grip1 event to the input execution pin of the Pickup event, and connect the Released execution pin to the input of the Drop event (see [Figure 8.10](#)).

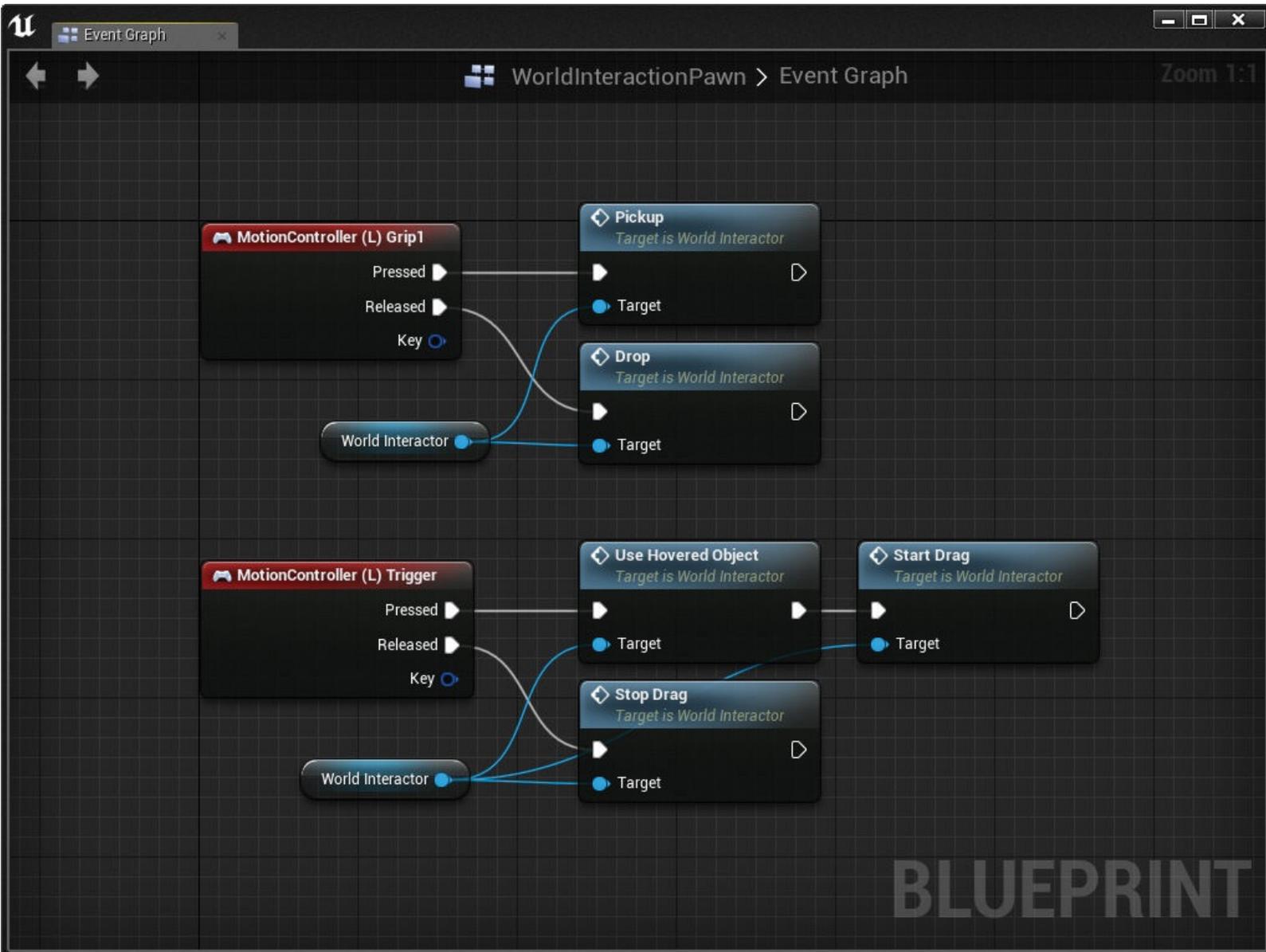


Figure 8.10 World Interaction Pawn: activating events on the Interactor Component

This will allow you to pick up an object with the grip button and drop it when you release the same button.

5. Create a new getter for the World Interactor Component again and drag off of its blue output pin to add a call to the UseHoveredObject, StartDrag, and StopDrag events.
6. Connect the Pressed execution pin of the MotionController (L) Trigger event to the input execution pin of the UseHoveredObject event, connect the out execution pin of the UseHoveredObject event to the input execution pin of the StartDrag event, and finally connect the Released execution pin of the MotionController (L) Trigger event to the input of the StopDrag event (see [Figure 8.10](#)). This allows you to use objects with the trigger but drag them with the trigger as well.

Note

If you have two Motion Controller Components, simply follow these steps again but change the MotionController (L) Grip1 and MotionController (L) Trigger events to MotionController (R) Grip1 and MotionController (R) Trigger events where appropriate.

Creating the Interactive Objects

Now that you have the base system in place to interact with objects, it's time to create the interactive objects themselves.

Creating an Interactive Static Mesh Actor

The first interactive object is a simple Static Mesh Actor that lets you pick it up and throw it. This Blueprint can then be used for basic set dressing objects in the scene that don't need any specific functionality but with which you would still like the player to be able to interact.

1. In the InteractiveObjects folder (Blueprints → InteractiveObjects) add a new Blueprint based on a Static Mesh Actor by selecting Add New → Blueprint Class → Static Mesh Actor, then selecting the Select button in the Pick Parent Class dialog.
2. Name the newly created object InteractiveStaticMesh.
3. Open the newly created Blueprint and click the Class Settings shelf button.
4. In the Details pane add the interactive interface to the Blueprint by clicking the Add button under Interfaces → Implemented Interfaces and choosing World Interaction Interface (see [Figure 8.11](#)).

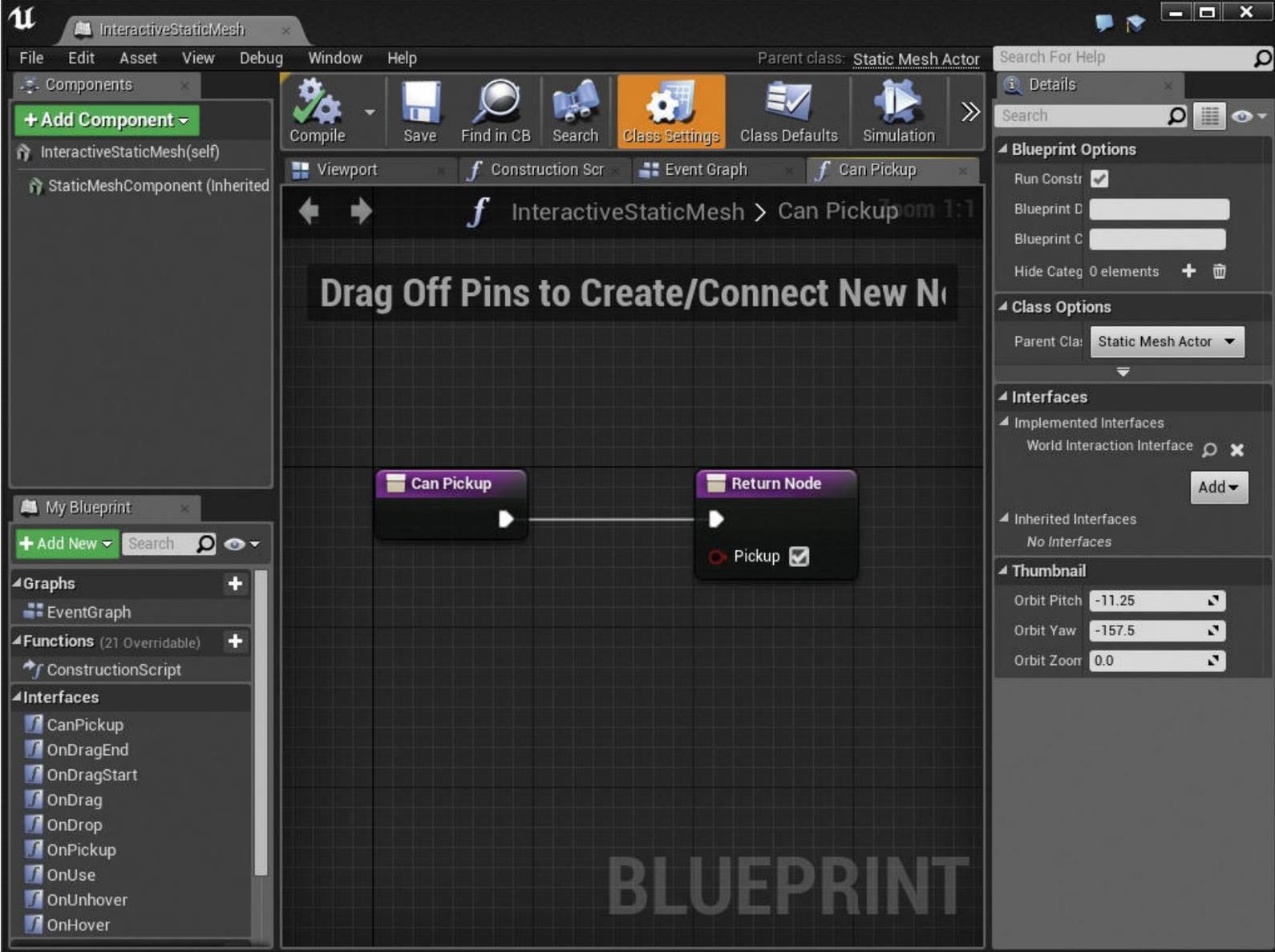


Figure 8.11 Interactive Static Mesh: adding the interface and allowing it to be picked up

5. Compile the Blueprint to get access to the interface functions.
6. Double-click the CanPickup function and set the Pickup return value to True. This will allow you to pick up this Blueprint (see [Figure 8.11](#)).
7. Select the StaticMeshComponent (Inherited) Component in the Components panel. In the Details panel on the right set its Mobility to Movable. Set its Static Mesh property to Cube (you may need to show the engine content to see this Cube). Set the Simulate Physics property to True. Last, enable Generate Overlap Events under Collision.
8. Open the OnPickup interface function.
9. Create a variable getter for the StaticMeshComponent (Inherited).
10. Drag off of the blue output pin of the StaticMeshComponent getter, call SetSimulatePhysics, and leave the Simulate input pin set to False, connecting it between the function input and Return value (see [Figure 8.12](#)). This prevents the object from falling through the user's hands when it's picked up.

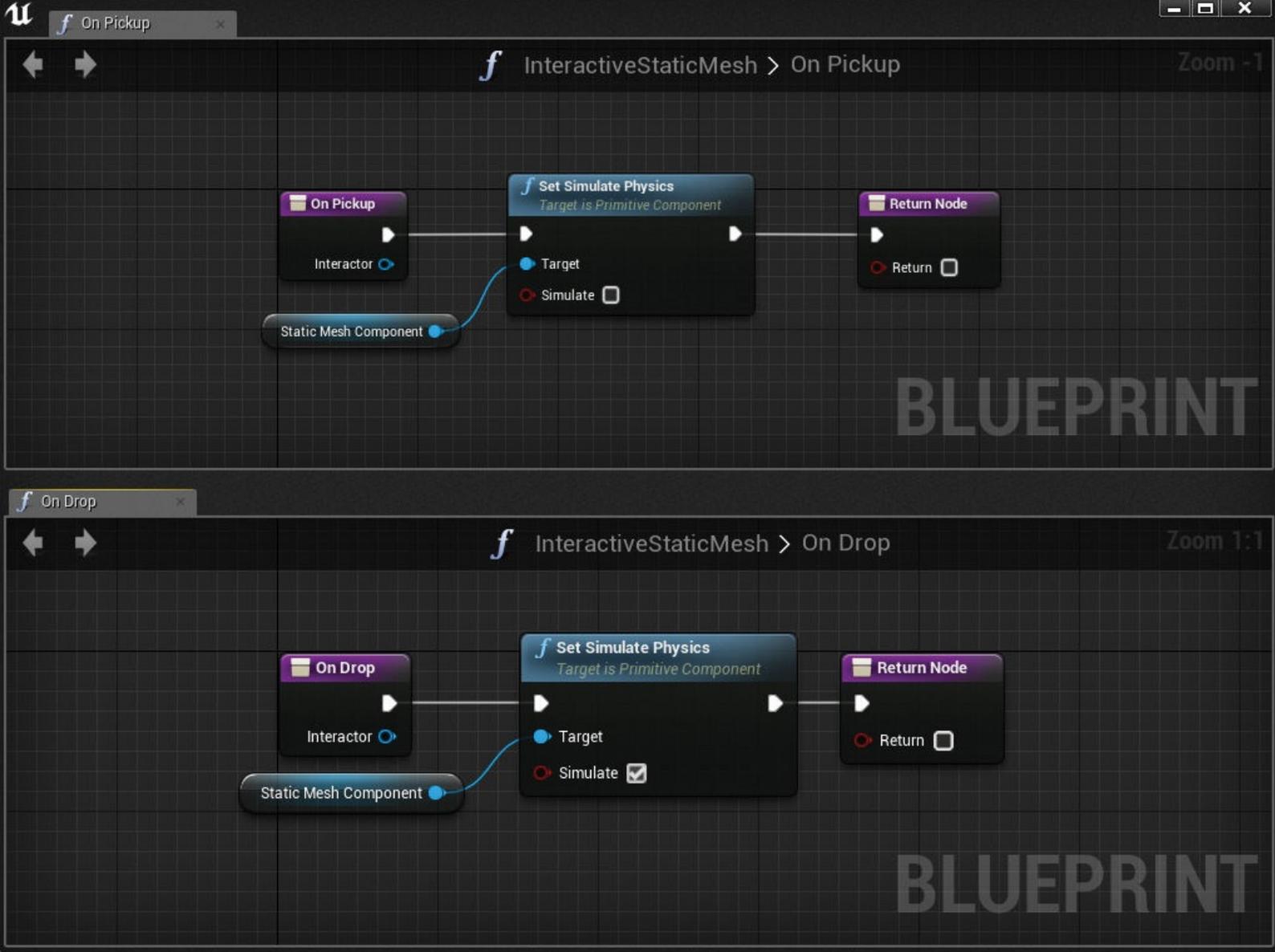


Figure 8.12 Interactive Static Mesh: Pickup and Drop functions

11. Open the OnDrop interface function.
12. Create a new getter for the StaticMeshComponent once more and call the SetSimulatePhysics function; this time, however, set the Simulate pin to True. This reenables physics when you drop the object.

Note

Enabling physics on a Component of a Blueprint takes that Component out of the Blueprints hierarchy. In this instance, because the StaticMeshComponent is the root of our Blueprint, this will have no effect.

However, if you enable physics on a Component that is not the root of the Blueprint, you will need to reattach it to its root Component before it can be affected by the Blueprint's transform.

A good place to do this is after the OnPickup function.

Conserving Collision

When you disable physics simulation, the object no longer collides with static geometry in your level because the object is now getting its transform directly from the motion controller to which it is attached.

If you want to retain the collision, a work-around (albeit a complicated one) is to not disable physics; instead, set the picked-up object's linear and angular damping values to a very high number. This causes the picked-up object to act as if gravity is no longer affecting it to any significant degree. This prevents it from falling out of the user's hands and allows the object to collide with static geometry.

However, the object does not return to its initial position after it has collided with some geometry. To fix this, you can disable the object's gravity after you pick it up, and when the object moves away from its original position, you can simulate gravity in the opposite direction to make the object seem to want to return to its original position.

Creating an Interactive Button

A common input system for motion controllers is a physical button that reacts to users pushing it. The first thing that comes to mind when wanting to have an interactive object like this is to enable physics and add some constraints. Unfortunately, in practice, this doesn't interact well with the unconstrained movement motion controllers give you. Therefore, as with many motion controller interactions, I suggest faking your own physical interaction to get the most control out of the button.

To make the button, you won't need to implement your fancy new interface. Ideally, you want the button to be able to be pressed by any object the player is holding and not just the Interactor Component.

Creating the Button and Button Mesh

Before you implement the functionality of the button, you should create the visual representation out of basic Components:

1. In the InteractiveObjects folder (Blueprints → InteractiveObjects) add a new Actor Blueprint.
2. Name the newly created Actor InteractiveButton and open it.
3. Create three new Components: a Scene Component named ButtonRoot, a Cylinder Component named Button, and another Cylinder Component named Base.
4. Drag the Button Cylinder onto the ButtonRoot to set it as its child.
5. Set the ButtonRoot Component's location to (X = 0.0, Y = 0.0, Z = 5.0). This will move the button 5 units up in the Z-axis.
6. Set the Button Component's scale to (X = 0.1, Y = 0.1, Z = 0.05). This will make it look more like a proper button.
7. Set the Base Component's scale to (X = 0.2, Y = 0.2, Z = 0.05). This will make it look more like the base of the button (see [Figure 8.13](#)).

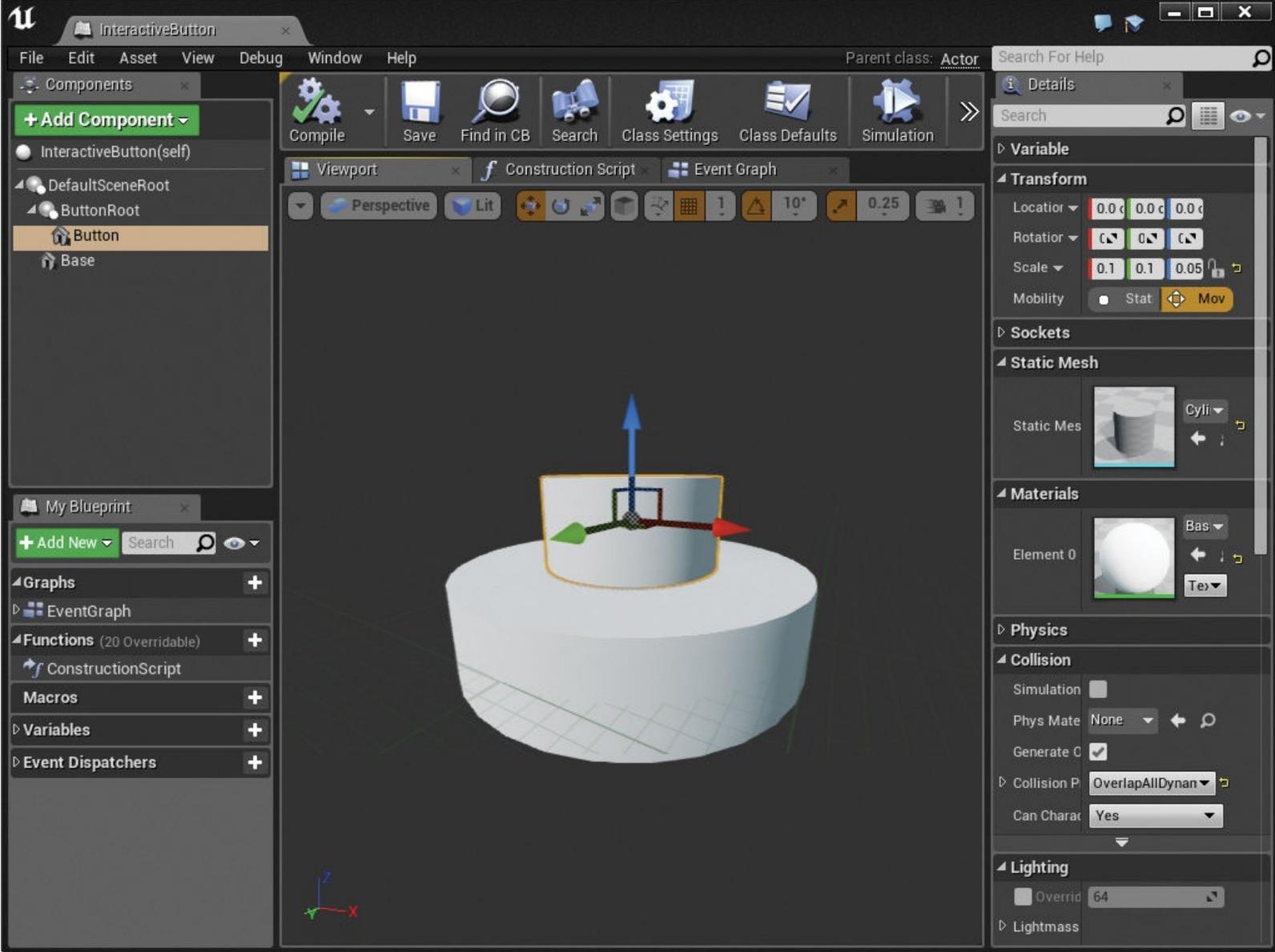


Figure 8.13 Interactive button: basic Components

8. Select the Button Component once more and set its Collision Preset to OverlapAllDynamic. You will use the button's overlap to detect when to press the button.
9. Select the Base Component and set its Collision Preset to NoCollision. This ensures that it will not interfere with the button.

Detecting Button Overlaps

Now that you have the meshes set up, you need to detect when the button is overlapped by an Actor so you can apply the math to press it:

1. Create a new Primitive Component reference variable and name it OverlappedComponent. This will hold the current overlapped Component.
2. Create a new Vector variable named InitialOverlapLocation, which stores the initial world location of the overlapped Component.
3. Create a new Float variable named ButtonPressAmount. This will store the current button pressed amount.
4. Create one more Float variable named MaxButtonPressAmount and set its default value to 4. This will be the maximum press amount for the button until it calls the Pressed event.

5. Create a new event dispatcher named Pressed. You are using an event dispatcher because it will make it easy for any other Actor to bind an event to the button's Pressed event without the button needing to be aware of it.
6. Click the Button Component in the Components panel, and then in the Details panel click the + button on the OnComponentBeginOverlap and OnComponentEndOverlap events. These events should now show up in the Event Graph (see [Figure 8.14](#)).

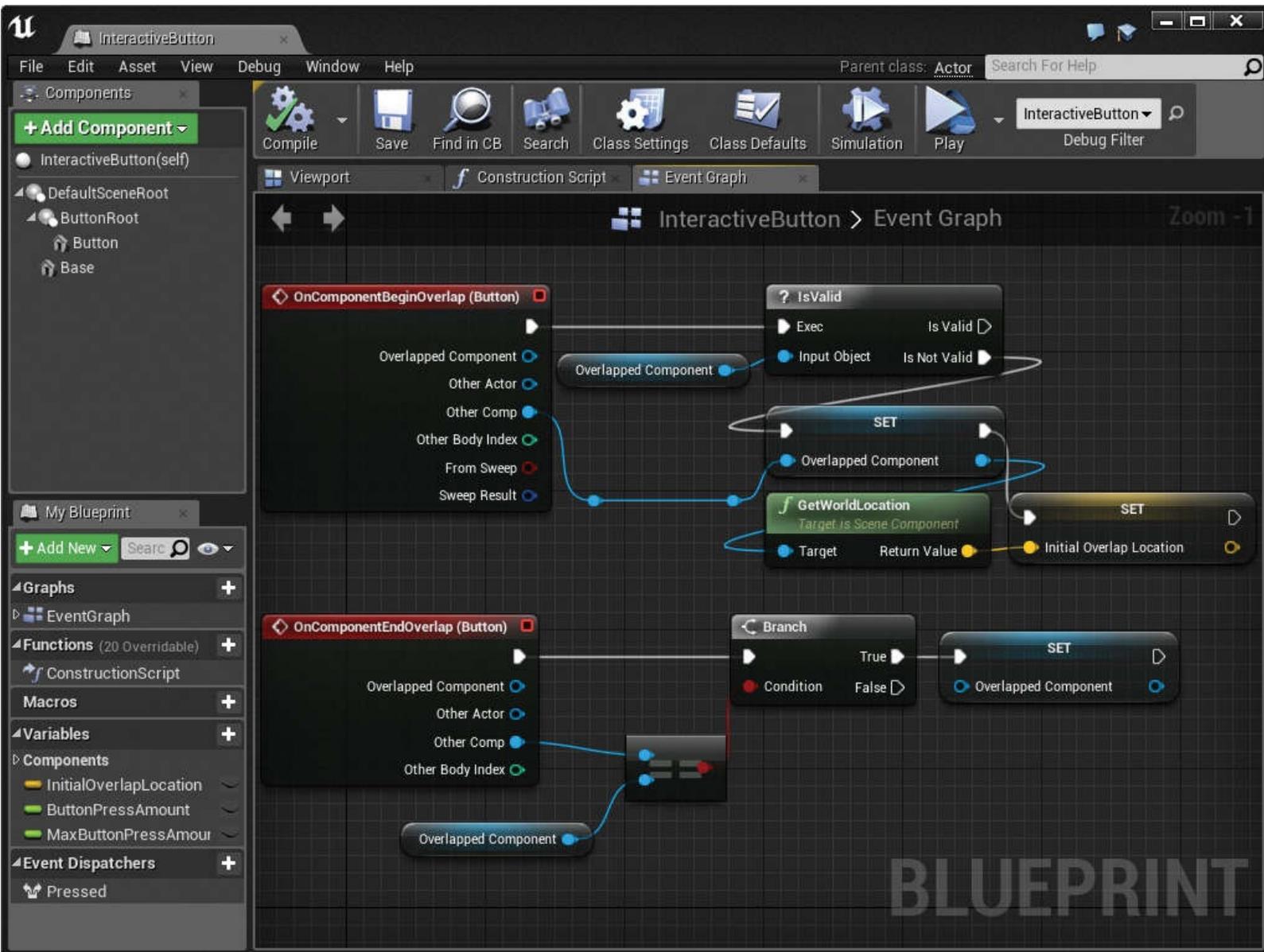


Figure 8.14 Interactive button: detecting button overlaps

7. After the output execution pin of the OnComponentBeginOverlap (Button) event add an IsValid node and connect the Input Object input pin to a new OverlappedComponent variable getter.
8. Create a new setter node for the OverlappedComponent variable and attach the input execution pin to the Is Not Valid output execution pin of the validity check. After that connect the blue input pin of the setter to the Other Comp output pin of the OnComponentBeginOverlap (Button) event.
9. Create a setter for the InitialOverlapLocation variable in the Event Graph and connect its input execution pin to the output execution pin of the OverlappedComponent setter.
10. Drag off of the blue output pin of the OverlappedComponent setter, call the GetWorldLocation function, and pass that into the InitialOverlapLocation setter (see [Figure 8.14](#)).
11. Create a new Branch node and connect it to the output execution pin of the OnComponentEndOverlap (Button) event.

12. Drag off of the blue output pin of the Other Comp output pin, add an EqualTo check, and connect the other input pin of the Equal node to a new OverlappedComponent getter.
13. Create a new OverlappedComponent setter and connect it to the True execution pin of the Branch, making sure to leave the setter blank because you want to clear the OverlappedComponent variable.

Pressing the Button

Now when you detect that an object has overlapped the button, you need to press it based on the overlapping object.

To do this, you will transform the initial overlapped Component's world position to be relative to the button mesh, then find out how far the object has traveled while it has been overlapping the button. This indicates how much you need to depress the button in its relative space. Pressing the button is as simple as subtracting this value from the button's initial position.

Doing this calculation relative to the button allows you to rotate the button however it is needed in the world and still ensure that it functions correctly.

1. Drag off of the EventTick node and create a new Sequence node.
2. Add a new IsValid node with an Input Object of an OverlappedComponent variable getter and attach it to the first execution pin of the Sequence. This ensures that you press the button only if you have an overlapped Component.
3. Create a new variable getter for the ButtonRoot Component.
4. Drag off of this getter and create two GetWorldTransform nodes.
5. Take the Return Value from each of these nodes and call the InverseTransformLocation function. This transforms any world input you give it and makes it relative to the button root.
6. For the Location input of the first InverseTransformLocation node attach a new getter for the InitialOverlapLocation variable.
7. For the second Location input create a new OverlappedComponent variable getter and call the GetWorldLocation function, attaching the Return Value pin.
8. To find the distance the overlapped Component has traveled relative to the button, take the output of the first InverseTransformLocation node and subtract the output of the second by right-clicking and creating a Vector – Vector node.
9. Now that you have the distance the overlapped Component has traveled as a vector, you can right-click the output of the Vector – Vector node and choose Split Struct Pin. This allows you to get access to the individual axis of the distance vector.
10. Because the button is going to be depressed in the local Z-axis, you need to modify this button's up and down position to compensate for the user's hand position. Drag off of the Z-axis and add a new Clamp (float) node.
11. Add a new getter for the MaxButtonPressAmount variable and drag its output pin to the Max input of the Clamp (float). Leave the Min pin at 0.
12. Create a new setter node for the ButtonPressAmount and connect the output of the Clamp (float) to it.
13. Connect the IsValid output execution pin of the validity check to the new setter node (see [Figure 8.15](#)).

3. Create a new VInterpTo node and attach its Return Value pin to the New Location pin of the SetRelativeLocation node.
4. Add a new variable getter for the Button Component and get its Relative Location.
5. Connect this to the Current pin of the VInterpTo node.
6. Leave the Target input pin as (X = 0.0, Y = 0.0, Z = 0.0); this is the button's initial position.
7. Drag the Delta Seconds pin from EventTick into the Delta Time input of the VInterpTo node (see [Figure 8.16](#)).

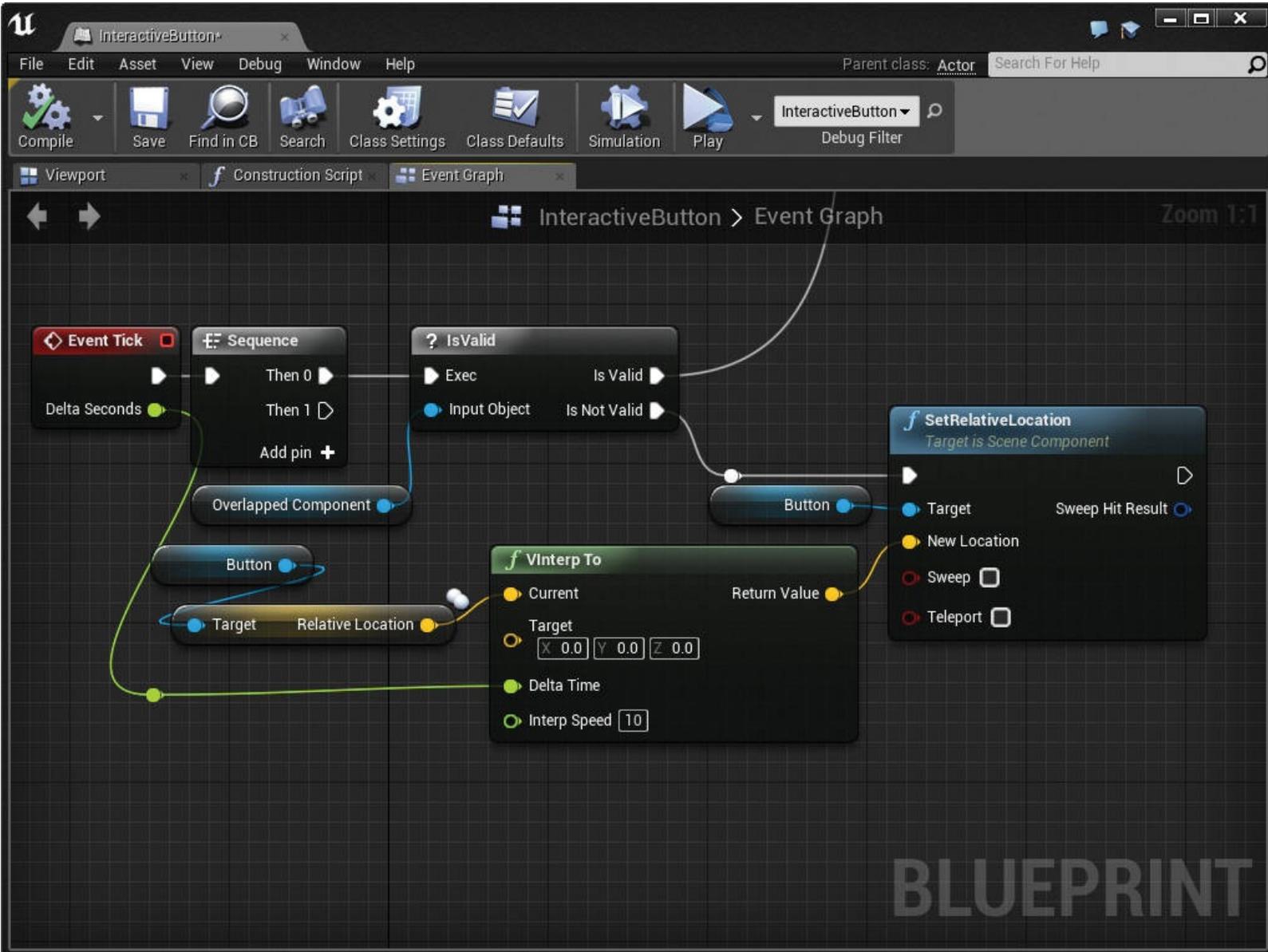


Figure 8.16 Interactive button: raising the button

8. Set the Interp Speed to 10. This will interpolate the button back to its initial position whenever it is not being pressed.

Activating the Button

You now have a button that is depressed when you overlap it and raised when you move away from it. The final thing you need to do is to detect when someone has fully pressed the button:

1. Drag off of the next execution pin of the Sequence node and add a new Branch node. You will use this to detect when the current press amount equals the maximum press amount and if so, call Pressed.

2. Create two new variable getters for ButtonPressAmount and MaxButtonPressAmount.
3. Drag off of the ButtonPressAmount variable getter and add a new Float >= Float node, passing in the MaxButtonPressAmount getter to the second input pin.
4. Connect the output of the Float >= Float node to the Condition input of the Branch.
5. Drag off of the True output execution pin of the Branch, add a DoOnce node, and connect the False output pin of the Branch to the Reset input execution pin of the DoOnce node. This ensures that you call Pressed only once per press of the button.
6. After the DoOnce node drag the Pressed event dispatcher from the My Blueprint panel and select Call from the context menu.
7. Attach the Completed output execution pin of the DoOnce node to the input of the Pressed call (see [Figure 8.17](#)).

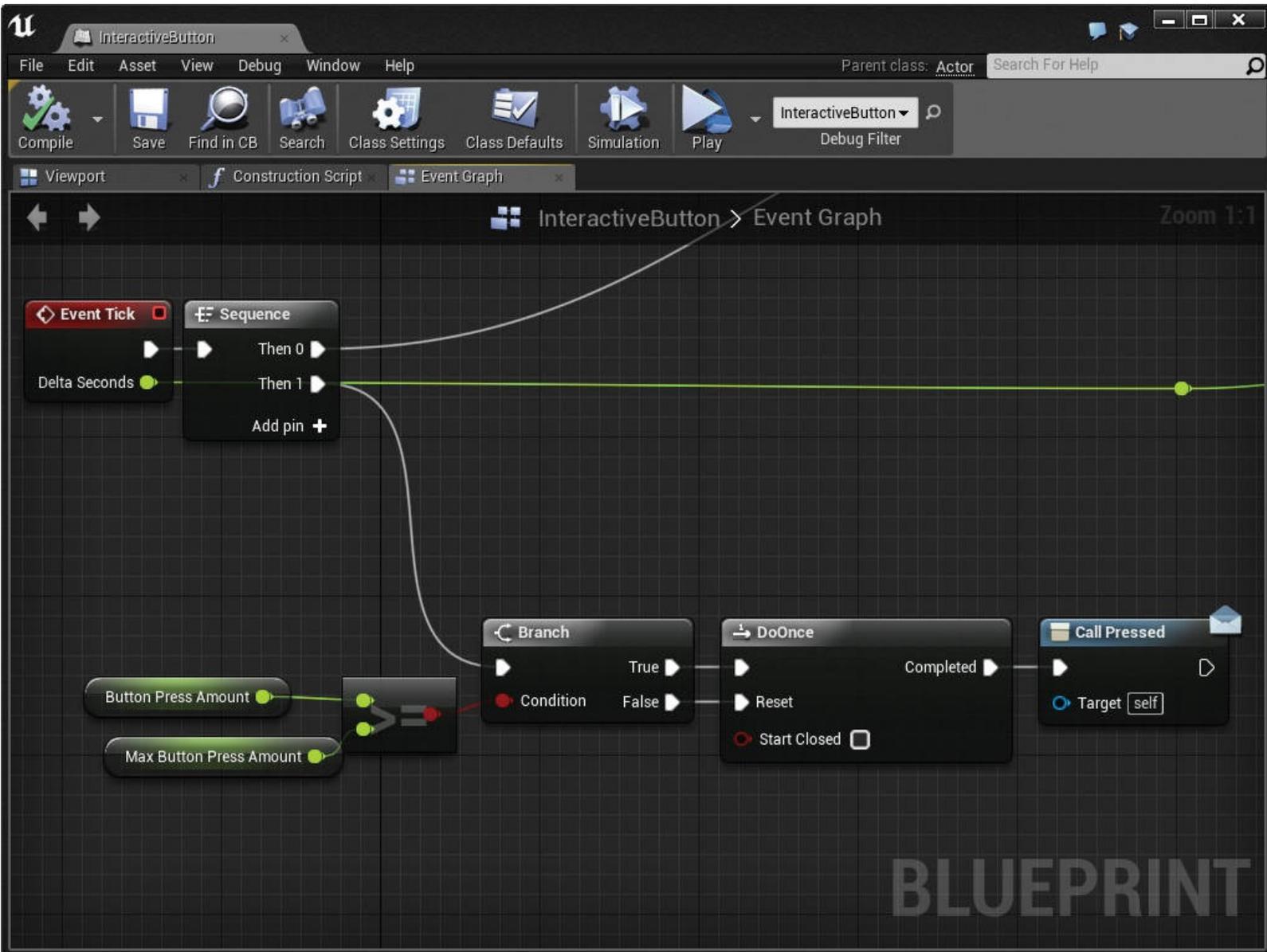


Figure 8.17 Interactive button: calling the Pressed event dispatcher

Using the Button

Now that you have a functioning button, all you need to do is use it.

One easy way to have the button trigger events in the level (once the button is in the level) is to create a reference to it in the level Blueprint and assign an event to its Pressed dispatcher:

1. Open the level Blueprint and with the button selected in the level create a reference to it.
2. Drag off of the reference and call the AssignPressed macro; it will create two nodes. The first is the BindEventToPressed node, which when called will bind an event to the event dispatcher, allowing it to be called whenever the button is pressed. The second is the actual event that will be called when the button is pressed.
3. Attach the input execution pin of the BindEventToPressed node to EventBeginPlay. This allows the Pressed_Event_0 event to be called anytime during play.
4. Do whatever your game needs to do when the button is pressed (for example, play a Sequencer sequence to open a door [see [Figure 8.18](#)]).

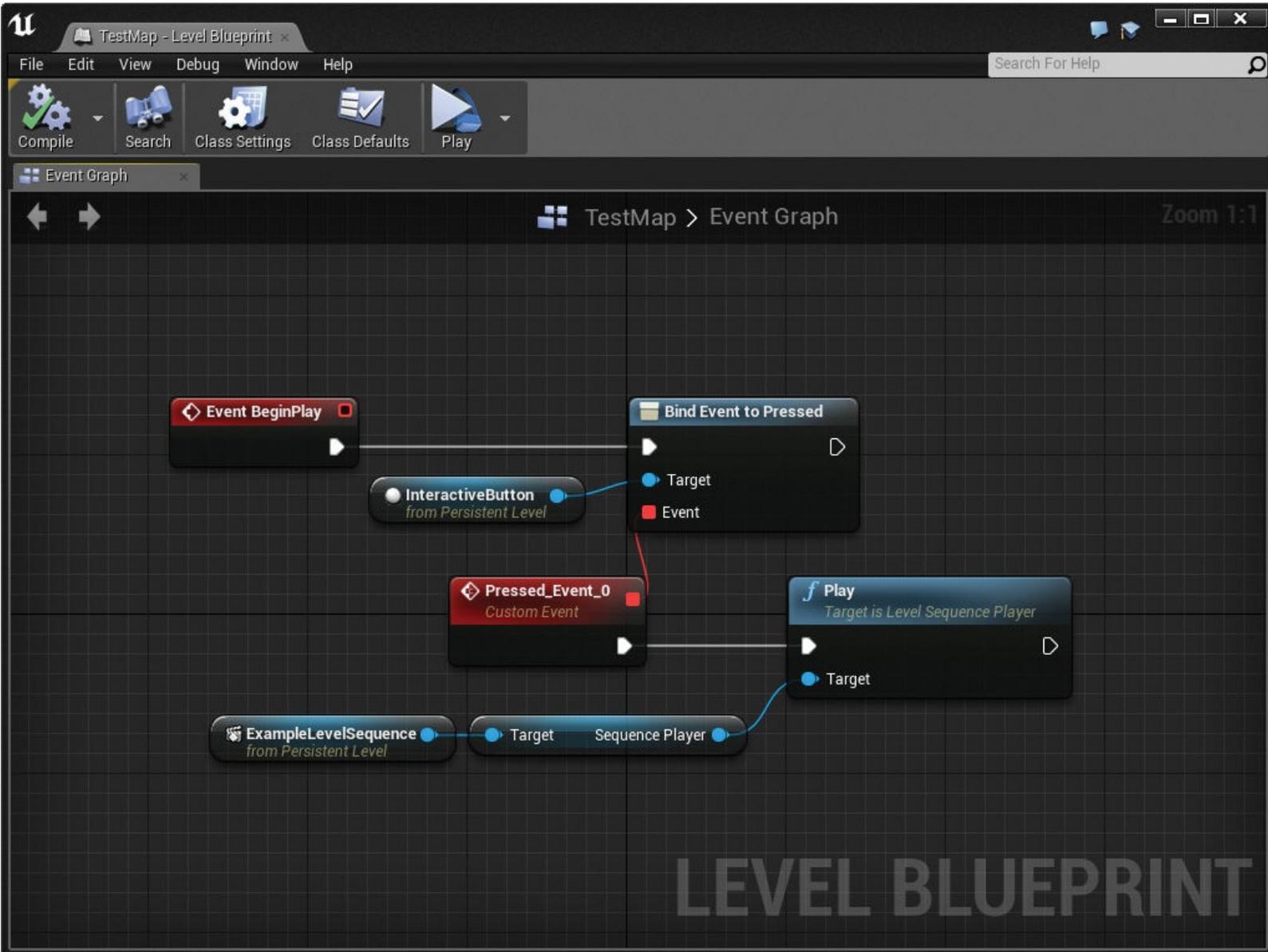


Figure 8.18 Interactive button: using the button in the level

Creating an Interactive Lever

In this section, you will create a lever that can be pulled with the grab functions set up in the World Interactor.

Like the button, the lever has a single output event that tells you whether the lever has been pulled, but unlike the button, pulling the lever is a much more physical action so it should be used for very deliberate actions that happen less often.

Creating the Lever Components

To set up the visual look of the lever you will use simple cylinders:

1. In the InteractiveObjects folder (Blueprints → InteractiveObjects) create a new Blueprint based on Actor.
2. Name the newly created Actor InteractiveLever and open it.
3. Click the Class Settings shelf button.
4. In the Details pane add the interactive interface by clicking the Add button under the Interfaces → Implemented Interfaces section and choose World Interaction Interface.
5. Compile the Blueprint to get access to the interface functions.
6. Create two Cylinder Components and name them LeverCylinder and LeverHandle respectively.
7. Create a Sphere Component and name it LeverEnd.
8. Create a Scene Component and name it LeverRotate. This is the Component you will use to rotate the lever.
9. Rotate the LeverCylinder Component 90 degrees on the X-axis, giving it a rotation of (Pitch = 0.0, Yaw = 0.0, Roll = 90.0), and set its Scale to (X = 0.3, Y = 0.3, Z = 0.1). This should make the cylinder look like the base of a lever.
10. Move the LeverHandle Component 25 units up on the Z-axis, giving it a location of (X = 0.0, Y = 0.0, Z = 25.0), and set its Scale to (X = 0.05, Y = 0.05, Z = 0.50). This should give you a cylinder that looks like the handle of a lever.
11. Move the LeverEnd Component 50 units up on the Z-axis, giving it a location of (X = 0.0, Y = 0.0, Z = 50.0), and set its Scale to (X = 0.1, Y = 0.1, Z = 0.1). This should give you a sphere that looks like the end of a lever.
12. Attach both the LeverEnd and LeverHandle Components to the LeverRotate Component. This allows you to rotate LeverRotate (see [Figure 8.19](#)) and have the rest follow.



Figure 8.19 Interactive lever Components

Rotating the Lever

When you drag the lever, you want to rotate the LeverRotate Component; this requires some basic trigonometry, which I will break down for you here if you haven't done anything similar before.

First, you need to set up some variables:

1. Create a new WorldInteractor reference variable named CurrentInteractor. This will store the interactor that is dragging the lever.
2. Create a new Boolean variable named IsResettingRotation. You will use this to control when to animate the lever back to its initial rotation.
3. Create two new Rotator variables named InitialResetRotation and InitialLeverRotation. The first will hold the rotation before you animate the lever back to its base rotation. The second will hold the initial base rotation from when the game was started.
4. Create four new Float variables and name them MinPitch, MaxPitch, ActivationPitch, and CurrentPitch.
 - MinPitch and MaxPitch will hold the lower and upper bounds of the lever's rotation.
 - ActivationPitch will hold the pitch that once the lever reaches it will activate and animate back to its original rotation.

- CurrentPitch will contain the current pitch of the lever.
5. Set MinPitch's default value to -90 , and set the ActivationPitch's default value to -90 as well.
 6. Create a new event dispatcher named Pulled.
 7. Drag off of EventBeginPlay and create a setter for the InitialLeverRotation variable.
 8. Create a getter for the LeverRotate Component and get its Relative Rotation, then connect it to the InitialLeverRotation setter input pin (see [Figure 8.20](#)).

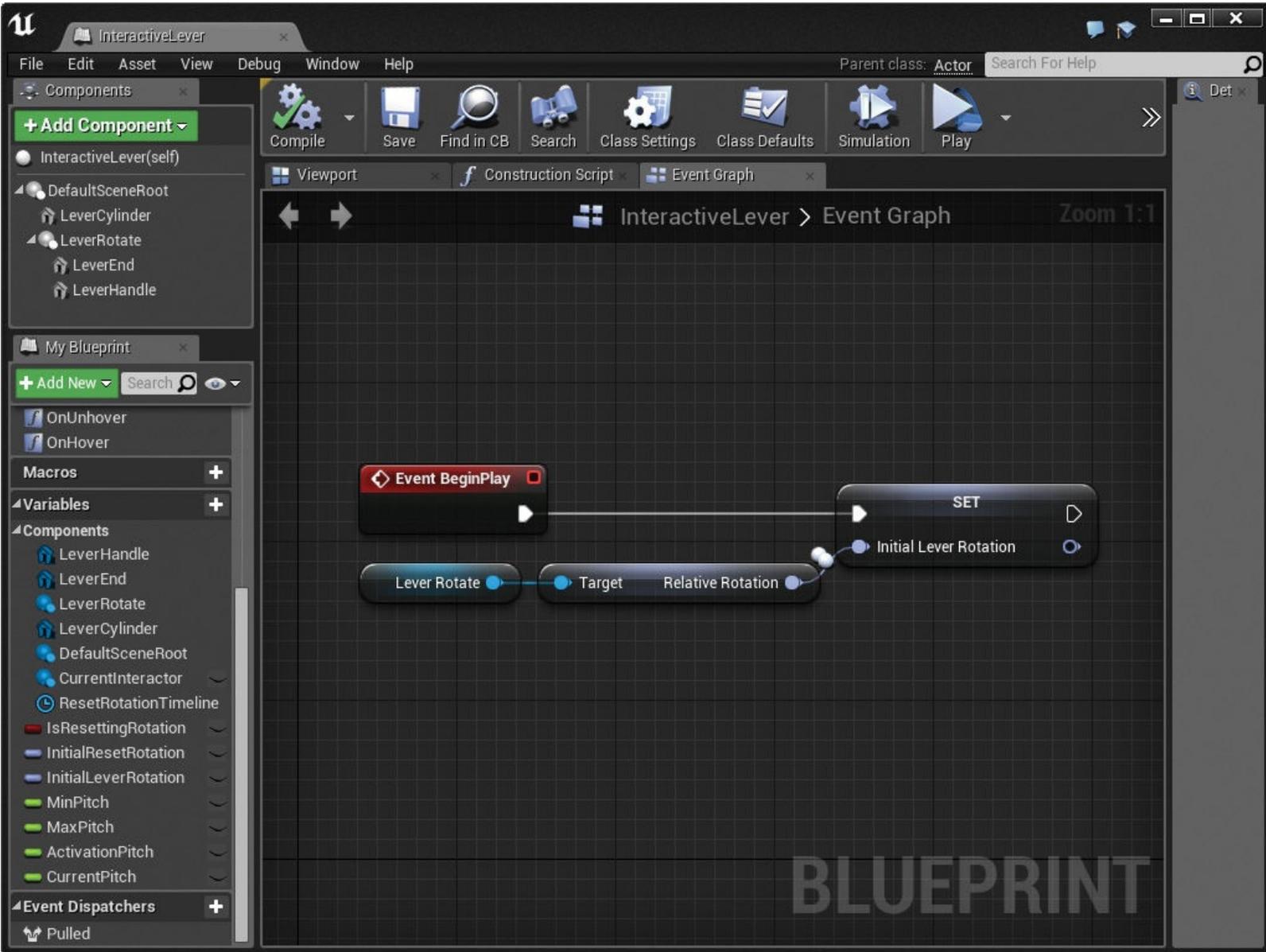


Figure 8.20 Interactive lever: variables and setup

Warning

MATH AHOY We are about to get into some trigonometry. I'll try to explain everything, but if it doesn't make sense feel free to skip to the implementation—you won't need to understand it to implement the functionality.

Now that you have the variables you need, it's time to calculate the rotation.

To do this, you will convert the current interactor's world location to a relative one. This corrects for any of the world transforms you apply to the lever in the level. You find the relative pitch rotation of this location by calculating the *arc tangent* (also called the *inverse tangent*) of the Z- and X-axes (see [Figure](#)

8.21). Last, because you want the rotation from the Z-axis and not the X-axis, you need to unrotate the relative location by 90 degrees to swap the axes.

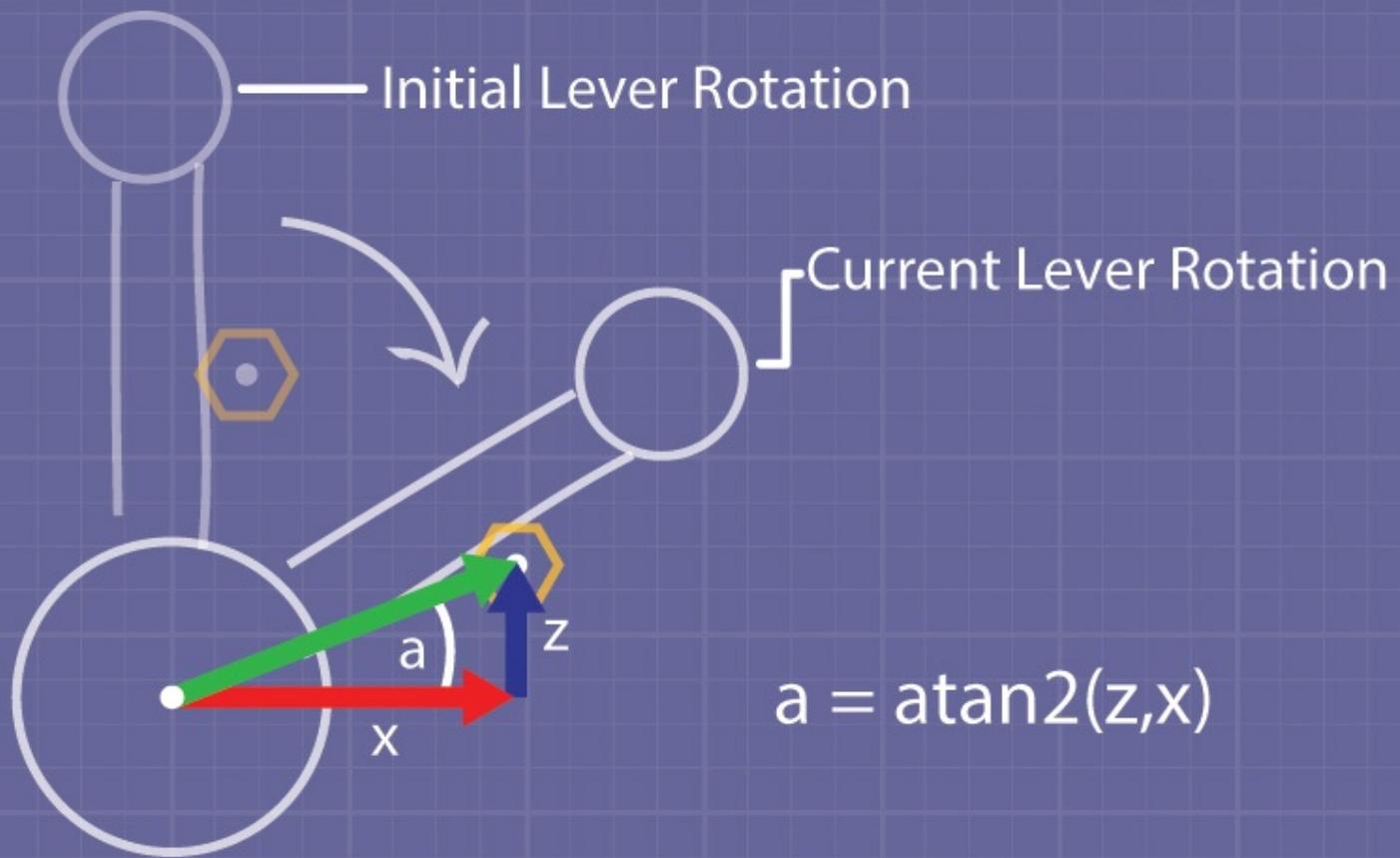


Figure 8.21 Interactive lever. x is the distance from the lever center in the X-axis to the grip point, z is the distance from the lever center in the Z-axis to the grip point, and a is the angle between the X-axis and the grip point.

Arc Tangent and Atan

In trigonometry the tangent of an angle is equal to the length of the opposite side of a right-angled tri-angle divided by the length of the adjacent side.

$$\tan(\text{angle}) = \frac{\text{opposite}}{\text{adjacent}}$$

The arc tangent or inverse tangent is simply the reverse of this. Given the result of the previous division, the arc tangent will give us the angle between the opposite side and the adjacent side.

$$\text{atan}\left(\frac{\text{opposite}}{\text{adjacent}}\right) = \text{angle}$$

Finally, the atan2 function is just an extension to the atan function that takes two arguments (x, y). This allows the function to internally recover from the problem of a divide-by-zero error (when the length of the adjacent side is zero) and also gather information about the signs of the sides to return a value in the appropriate quadrant.

Don't worry; I'll break it down step by step if you're lost.

1. Open the OnDrag interface function.
2. Drag off of the output execution pin of the event and create a new Branch node.
3. Drag off of the Interactor pin and create a new Equal (Object) node, passing in a new getter for the

CurrentInteractor variable to the second pin.

4. Connect the output of the Equal (Object) to the Branch node.
5. Drag off of the False pin and create a new Return node; leave it blank.
6. Drag off of the True output pin of the Branch and create a new Branch node.
7. Create a variable getter for the IsResettingRotation variable.
8. Drag off of its pin and add a new NOT Boolean node.
9. Connect the output of this NOT Boolean node to the Condition input of this new Branch. This will ensure that you don't rotate the lever when you're trying to animate it back to its initial rotation.
10. Create a new GetActorTransform node.
11. Drag off of the GetActorTransform node and call InverseTransformLocation.
12. Create a new getter for the CurrentInteractor variable and call GetWorldLocation on it.
13. Connect the Return Value of GetWorldLocation to the Location input pin of the InverseTransformLocation. This will transform the current interactor's world position into local space.
14. Drag off of the Return Value of InverseTransformLocation and call the UnrotateVector function, passing in 90 in the Y-axis to the B input (see [Figure 8.22](#)). This will rotate the vector because you want the pitch calculation to be relative to the lever, which is facing up and not forward.

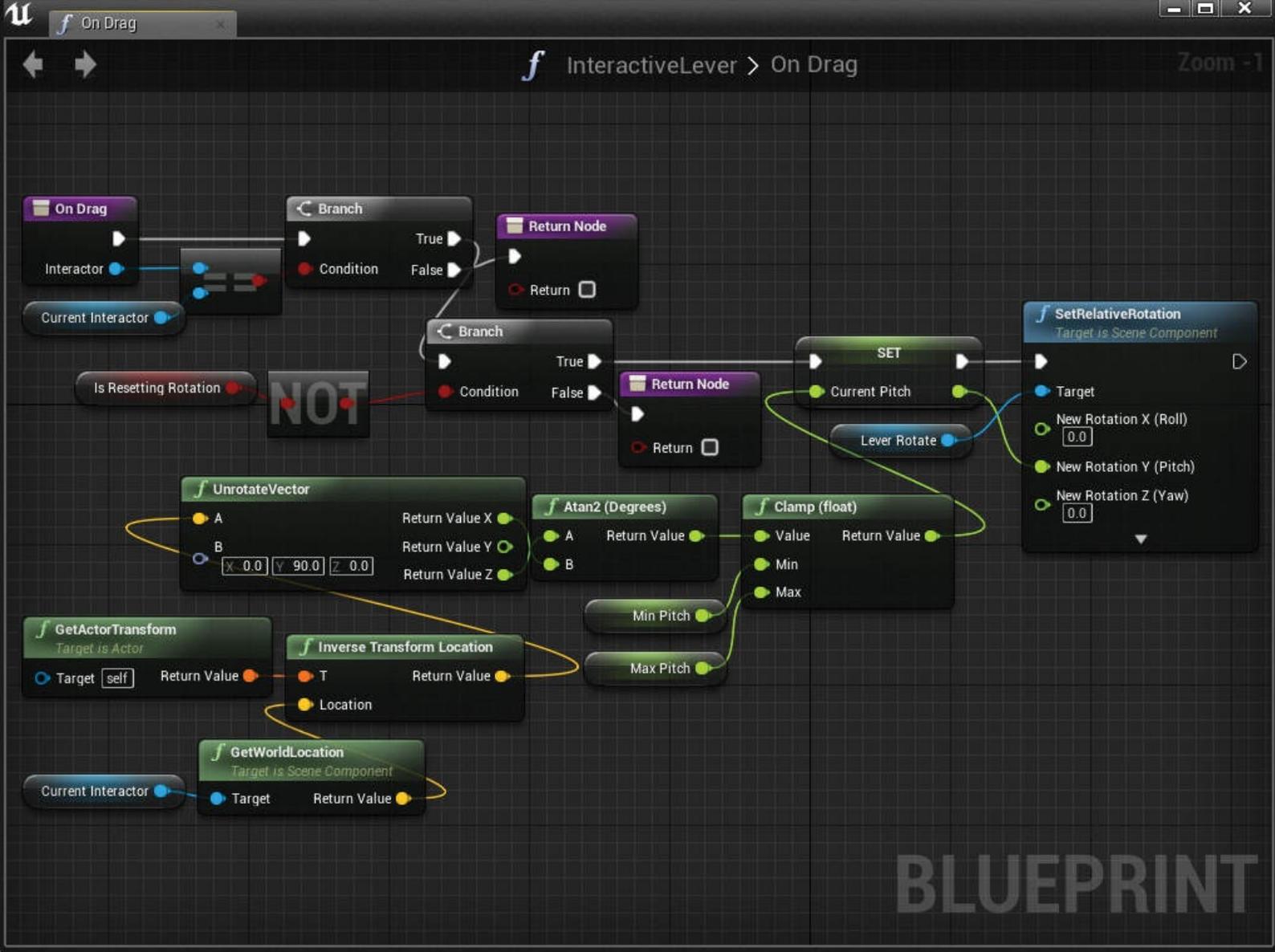


Figure 8.22 Interactive lever: rotation calculation

15. Right-click the Return Value pin of the UnrotateVector node and split the struct pin.
16. Drag off of the Return Value Z pin and call the Atan2 (Degrees) function.
17. Connect the Return Value X pin to the B input pin of Atan2 (Degrees). This will calculate the angle between the current interactor and the handle.
18. Drag off of the Return Value pin of the Atan2 (Degrees) node and call the Clamp (float) function.
19. Create two new getter nodes for the MinPitch and MaxPitch variables, then connect the first to the Min input pin of the Clamp (float) and the second to the Max input pin.
20. Create a new setter node for the CurrentPitch variable, and connect the output of Clamp (float) to the green input pin.
21. Connect the True output pin of the Branch node to the input execution pin of the setter node (see [Figure 8.22](#)).
22. Drag in a new getter node for the LeverRotate Component and call SetRelativeRotation on it.
23. Split the rotator struct pin of the SetRelativeRotation node by right-clicking the NewRotation pin and choosing Split Struct Pin.
24. Drag the output execution pin of the current pitch setter to the input execution pin of SetRelativeRotation, and connect the green output pin of the same setter to the New Rotation Y

(Pitch) input pin of the SetRelativeRotation node (see [Figure 8.22](#)).

Resetting the Lever

You now have a working lever, but so far it just rotates back and forth. To make it seem more lively, you will interpolate its rotation back to the lever's original rotation whenever it either reaches the end or is let go of:

1. Right-click the Event Graph, add a new custom event, and call it ResetRotation.
2. Create a new variable setter for the IsResettingRotation variable, and connect the output of the new event to the input execution pin of the setter. Check the input of the setter to make it True.
3. Create another variable setter for the InitialResetRotation variable. Connect the output execution pin of the IsResettingRotation setter to the input execution pin of the new node.
4. Create a new getter for the LeverRotate Component and get its RelativeRotation, connecting it to the input pin of the InitialResetRotation setter (see [Figure 8.23](#)).

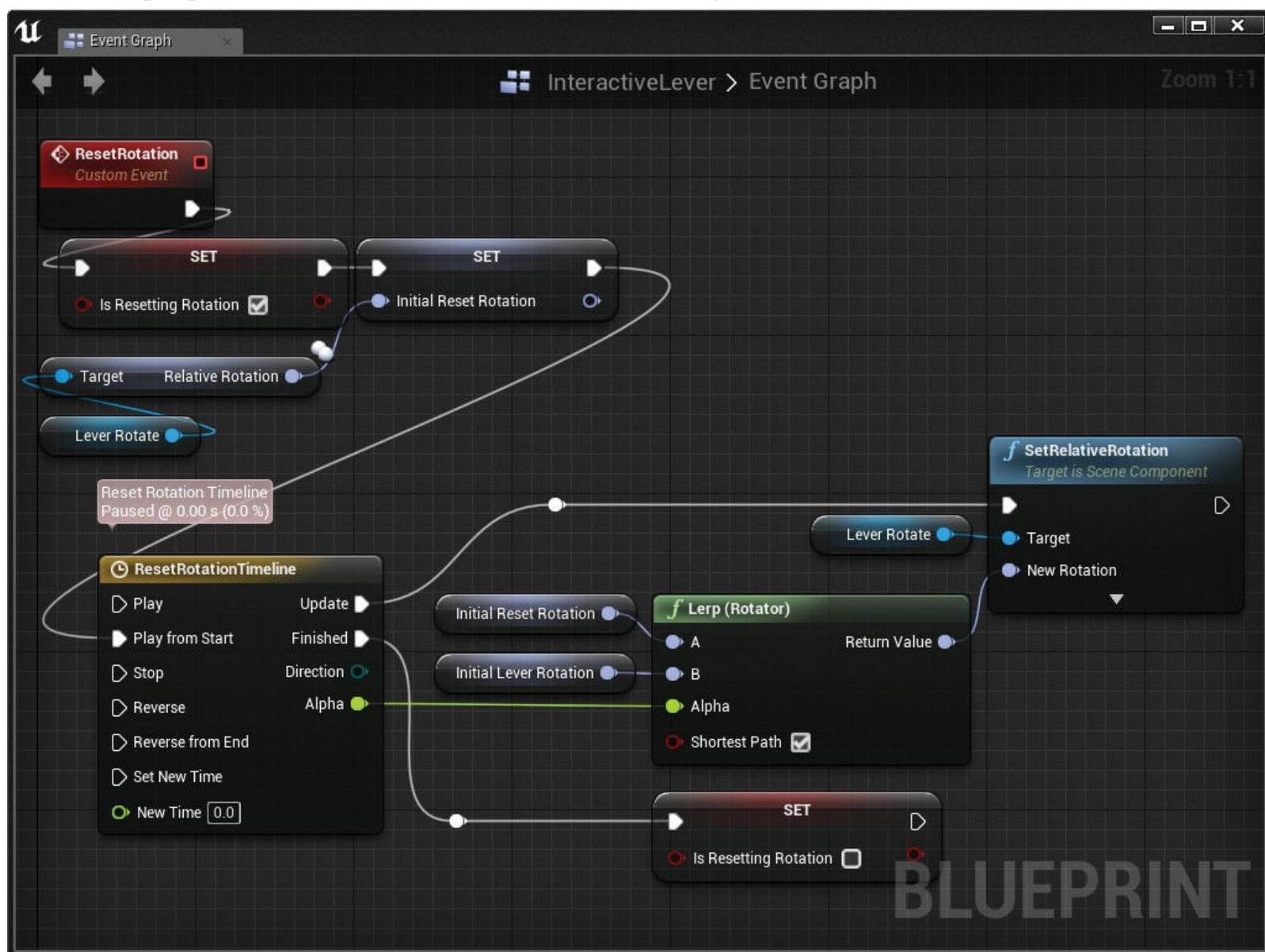


Figure 8.23 Interactive lever: resetting the lever rotation

5. To rotate the lever, you will create a new timeline. Right-click the Event Graph, choose Add Timeline, and name it ResetRotationTimeline.
6. Double-click the new timeline to open the timeline editor.

7. Add a new Float track by clicking the Add Float Track button on the shelf; name it Alpha.
8. Create two points on the timeline by Shift + clicking in the gridded area.
9. Set the first point to have a Time and Value of 0.
10. Set the second point to have a Time of 0.5 and a Value of 1 (see [Figure 8.24](#)).

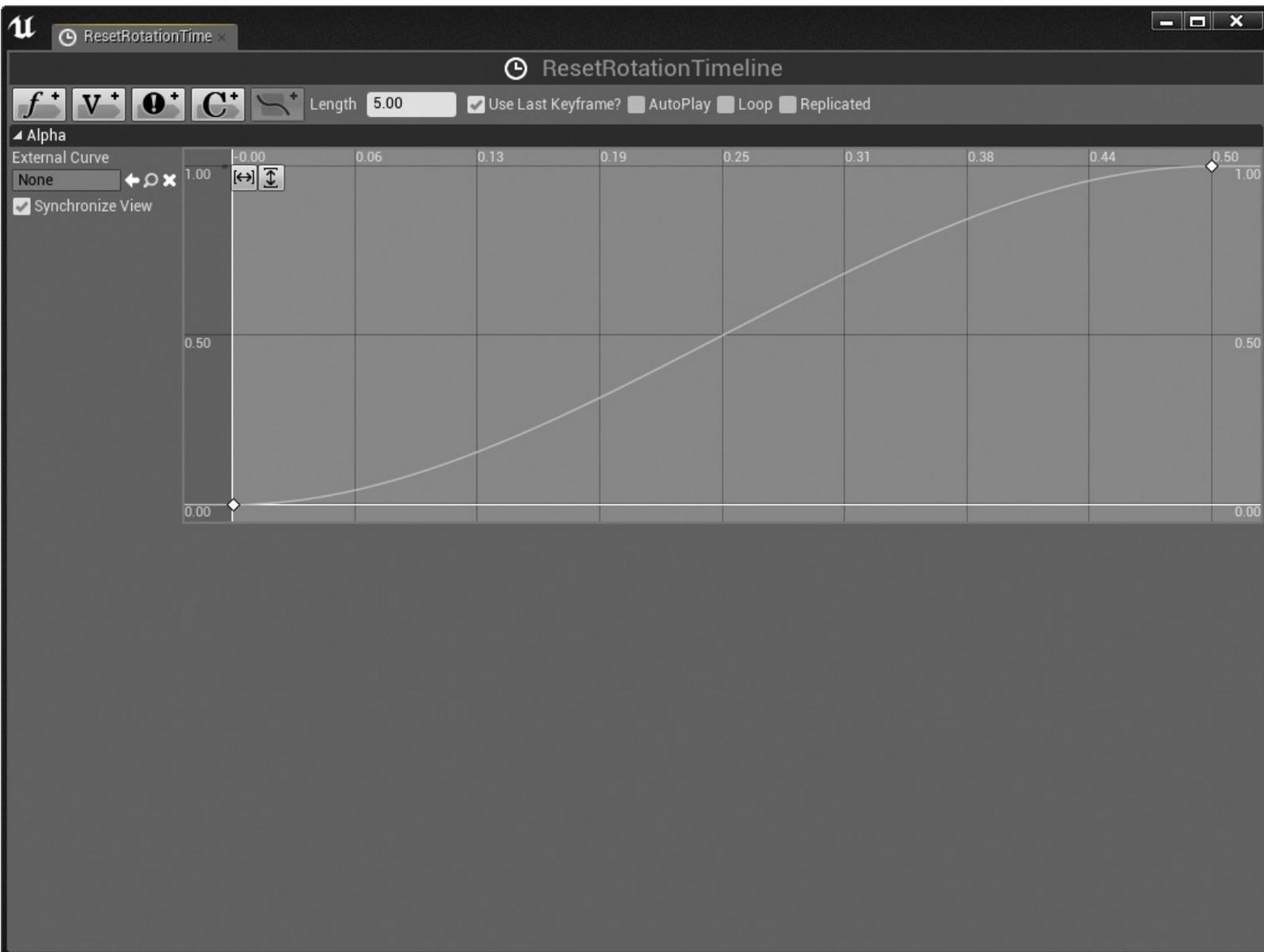


Figure 8.24 Interactive lever: resetting the lever rotation timeline

11. Select both points by dragging over both of them, then right-click one of the points and choose Auto under Key Interpolation. This will give the line between the points a nice curve.
12. Check the Use Last Keyframe? variable in the shelf. This will ensure that the length of the timeline is set to the last key frame.
13. Connect the output execution pin of the InitialResetRotation setter to the Play from Start input pin of the timeline.
14. Create two new variable getter nodes for the InitialResetRotation and InitialLeverRotation variables.
15. Drag off of the InitialResetRotation getter node and call the Lerp (Rotator) function.
16. Connect the InitialLeverRotation getter to the B input pin of the Lerp (Rotator) node.
17. Connect the Alpha pin of the timeline to the Alpha pin of the Lerp (Rotator).

18. Check the Shortest Path Boolean on the Lerp (Rotator) node. This node allows you to linearly interpolate between two rotations; however, in this case because you are driving it with a timeline, you can add any kind of interpolation curve you would like in the timeline.
19. Create a getter for the LeverRotate Component and call the SetRelativeRotation function on it.
20. For the New Rotation input pin attach the Return Value of Lerp (Rotator), and for the input execution pin attach the Update execution pin of the timeline.
21. Create a new setter for the IsResettingRotation variable, and connect the input execution pin to the Finished pin of the timeline (see [Figure 8.22](#)). This allows you to pull the lever again once the animation has finished.
22. Open the OnDragEnd interface function.
23. Create a new Branch node and connect it to the input execution pin of the function (see [Figure 8.25](#)).

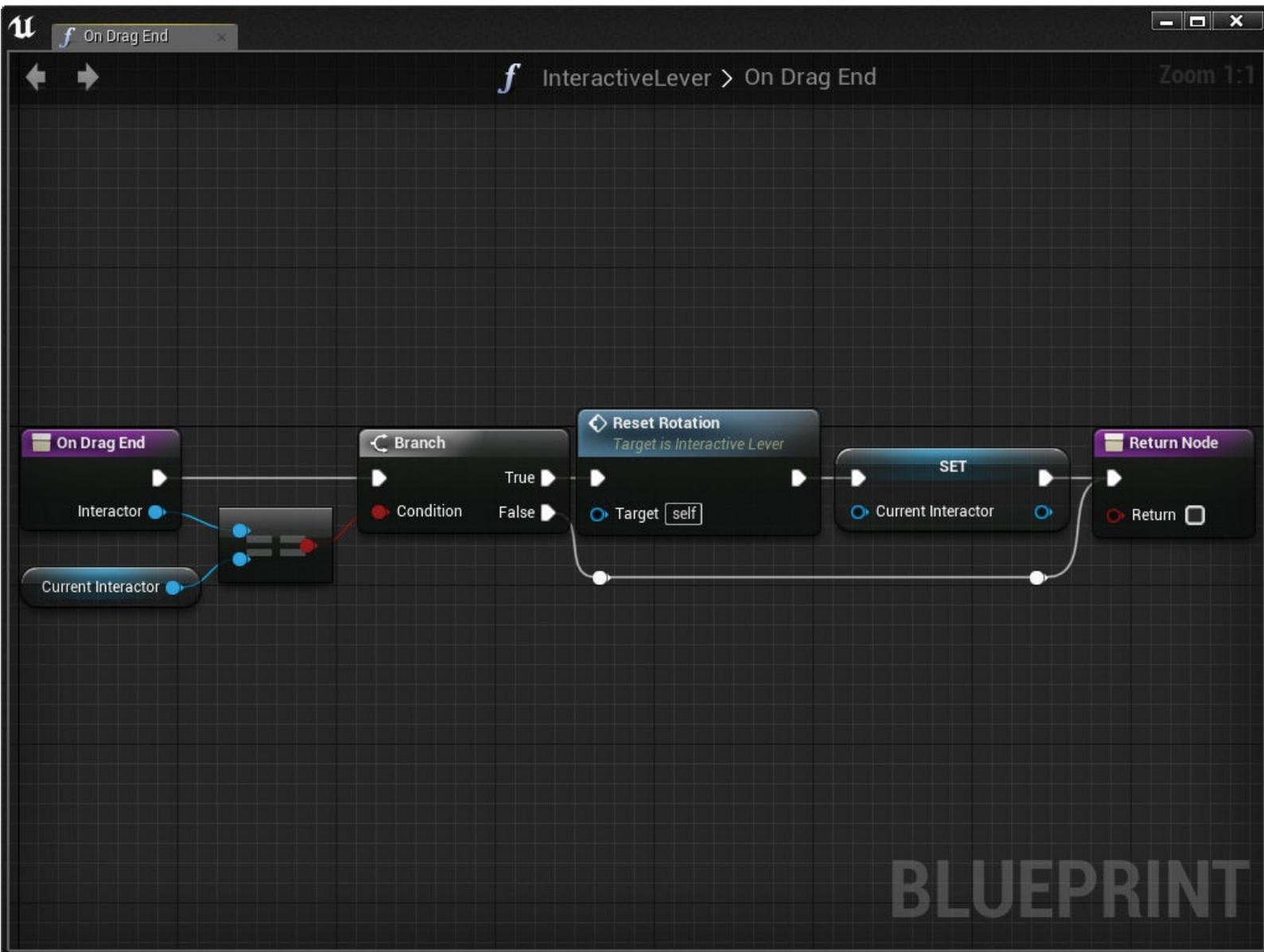


Figure 8.25 Interactive lever: resetting the lever rotation on drag end

24. Drag off of the Interactor input pin and create an Equal (Object) node, connecting a new variable getter for CurrentInteractor to the second input.
25. Connect the output of this Equal (Object) node to the Condition input of the Branch. This prevents users from stopping the dragging on the lever with a different interactor from the one that is

currently dragging it.

26. Drag off of the True pin of the Branch node and call the ResetRotation event.
27. After the ResetRotation call, create a new CurrentInteractor variable setter and leave it blank to clear the variable.
28. Connect the output of the setter to the Return node.
29. Connect the False output of the Branch to the Return node.

Triggering the Lever

Let's add the last piece of the puzzle: calling the Pulled event dispatcher when it reaches the activation pitch.

1. Back in the OnDrag function, create two new variable getter nodes for CurrentPitch and ActivationPitch.
2. Drag off of the CurrentPitch getter and call Equal (float), passing in the ActivationPitch getter as the second green pin.
3. Attach a new Branch node to the end of the SetRelativeRotation node, and connect the output of the Equal (float) to the Condition input of this Branch (see [Figure 8.26](#)).

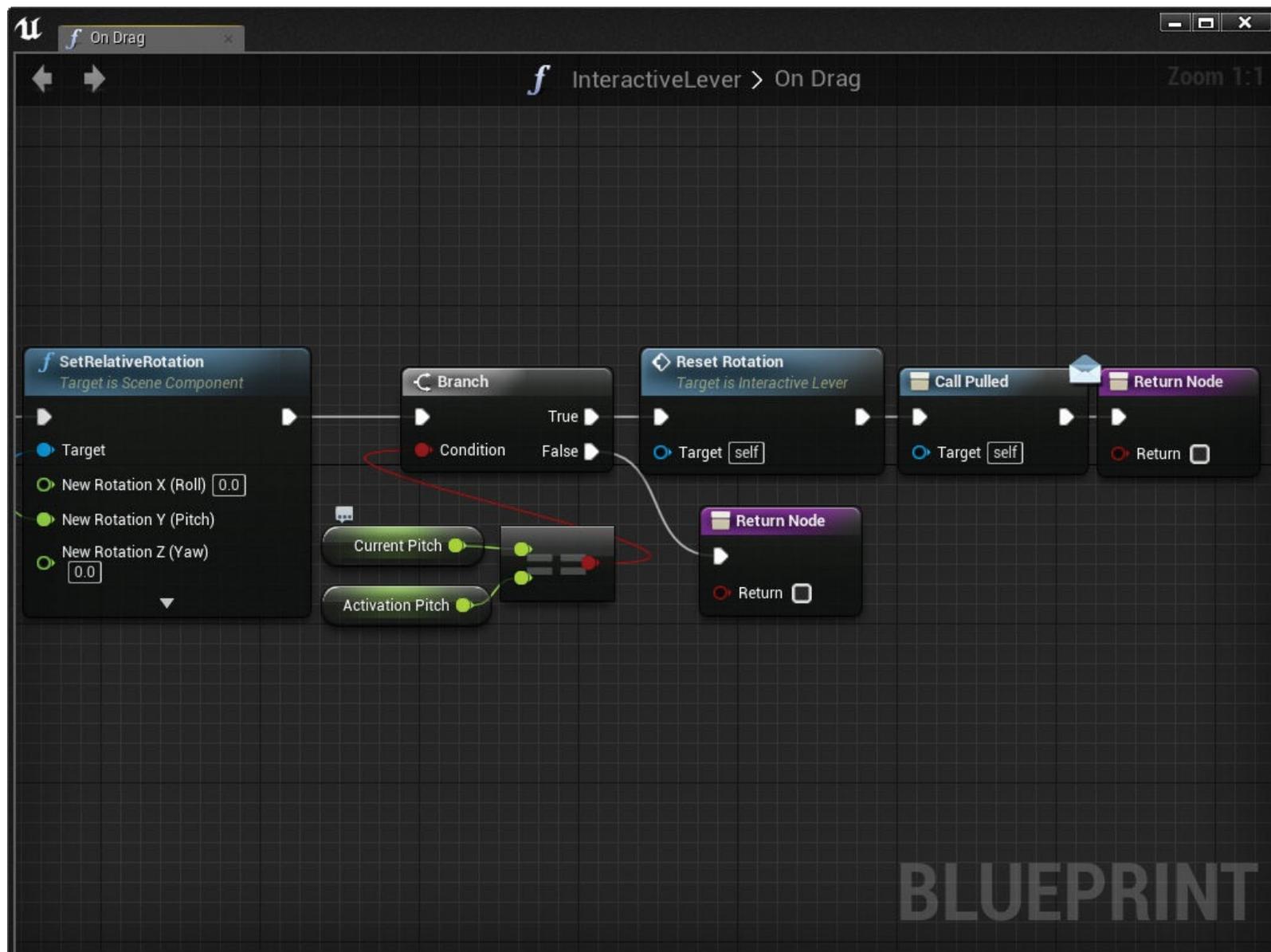


Figure 8.26 Interactive lever: calling the Pulled event dispatcher

4. Drag off of the True pin of the Branch and call the ResetRotation custom event.
5. After the ResetRotation call, call the Pulled event dispatcher.
6. Attach the output of the Pulled call to the Return node, and connect the False output of the Branch to another Return node.

Now you have a lever that can be pulled and activated. To hook into the Pulled event dispatcher, you can do exactly the same as with the interactive button and add it to the level, then in the level Blueprint bind to the Pulled event (see [Figure 8.27](#)).

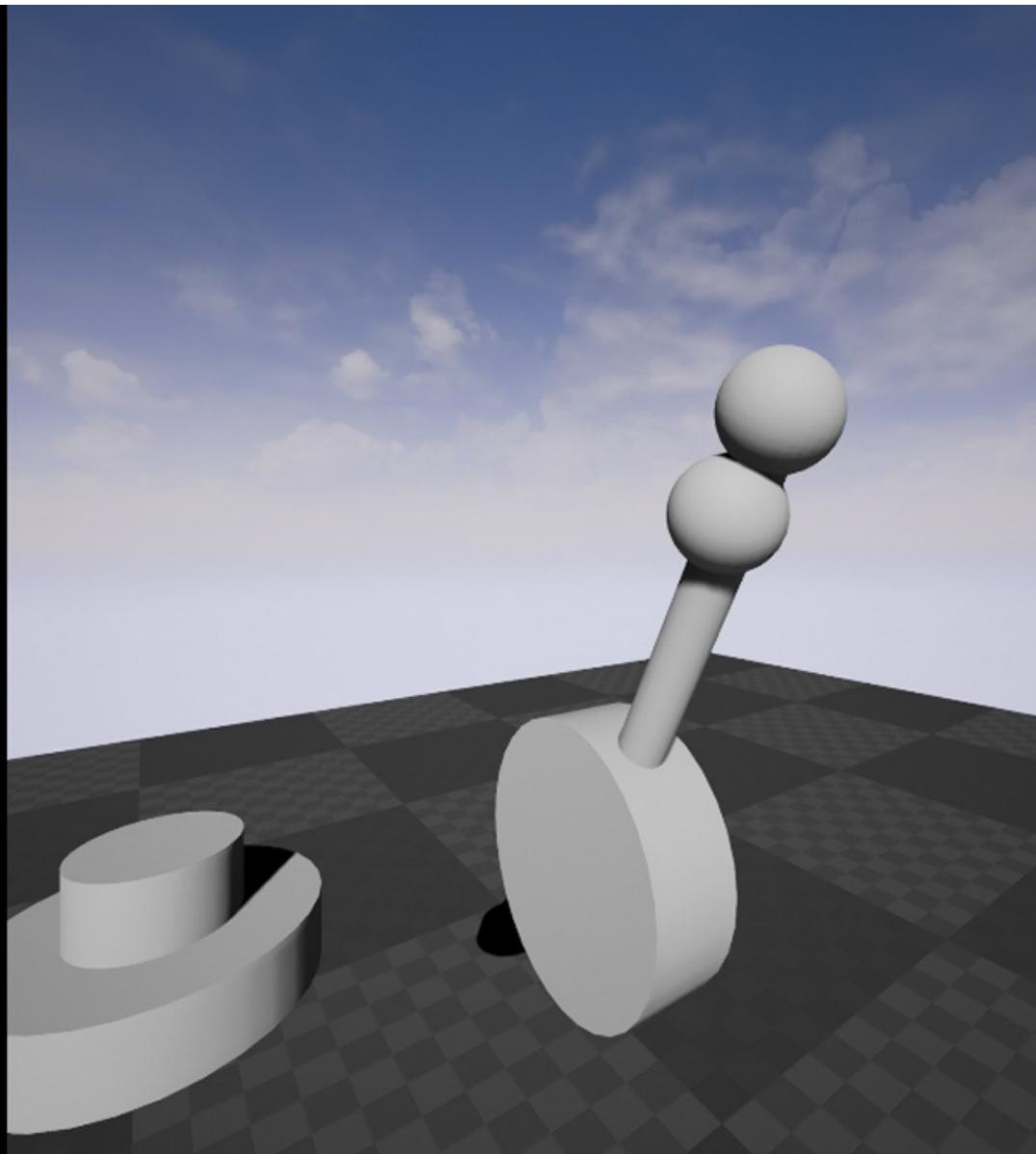


Figure 8.27 Interactive lever being pulled

Summary

This chapter looked at creating an extensible world interaction system that allows you to interact with objects in the world with motion controllers. You also tested that system with a few objects that can commonly be found in VR games and experiences such as a button and a lever.

Exercises

Consider these extensions to the interactive objects:

- Add force feedback effects when the player interacts with the Components. This can give the player a great sense of what the objects are made of (you may need to pass what hand is controlling the

interactor into the interactor itself to target your feedback effects).

- Add the ability for individual objects to override the default drop binding so the player does not have to hold the grab button continuously for important objects. One way to do this would be to create an input overriding Actor that you can spawn to override a specific hand's controls without affecting the other hand.
- Create new interaction methods; some examples are a drawer, a segmented control, and a dial control.

Chapter 9. VR Locomotion

Movement in VR is still an experimental field. This chapter introduces you to the concept of simulator sickness and some of the possible causes of discomfort in VR. You are also introduced to some of the more prominent movement mechanics currently used in VR games.

Simulator Sickness

Unlike motion sickness, which arises from physical motion, simulator sickness is the by-product of subjecting users to a perceived motion (in VR, in our case) that differs from what their bodies tell them. The main detector of these discrepancies between real-world motion and artificial motion is the vestibular system. The vestibular system's job is to detect acceleration (through fluid in the inner ear) and balance/orientation (through the acceleration of gravity). *Vection* is a term perceptual psychologists use to refer to the feeling of motion that a player gets from the visual system alone (as is the case in VR). This vection causes a disagreement with the player's vestibular system and can lead to discomfort. There are many theories as to why this is, but for our purposes, just know that more vection equals more discomfort. The vestibular system detects acceleration, and subjecting your users to visuals that disagree with this system causes discomfort. Therefore, it stands to reason that the less (artificial) acceleration there is inside your VR experiences, the more comfortable users will be.

Optic flow refers to any motion in your scene that signals to players that they are moving. The more optic flow, the more information players have that they are artificially moving. In general, this leads to more vection. Cockpits can therefore be one way to reduce vection/simulator sickness because they reduce the total optic flow in a scene. Cockpits also help with vection by providing players with a frame of reference, grounding them to a scene.

Cockpits are obviously not applicable to all experiences, and therefore other techniques can be used to ensure that your users have the most comfortable experience. The most prominent techniques are covered later in this chapter.

Finally, simulator sickness is similar to motion sickness in the way it affects some parts of the population more than others. Therefore, it's difficult to accommodate all tolerance levels, so know your target audience for your game/experience, and target your locomotion design choices to them. Erring on the side of caution is a good idea; however, be aware that many factors that increase the comfort of your experience can also decrease the immersion, and some players who are more tolerant of simulator sickness may want the option to turn off or reduce these comfort settings to gain a more immersive experience.

Locomotion Types

Through experimentation and research, the VR community has developed many different artificial locomotion methods; each has its advantages and disadvantages. Some can be fitted to many types of experiences, and others are restricted to a specific context. Because of the young age of the VR medium, the most popular techniques are still in flux; therefore, you may mix and match to best fit your experience.

Natural

Natural locomotion methods are generally one-to-one in their translation of real-world movements into player movements and, as such, they can lead to the least nausea-inducing experiences. In exchange, however, they are generally the most limiting of the locomotion methods. By default, you are restrained by the player's real-world/HMD's tracking space. Smart solutions to combat the requirements on physical space exist, and variations on natural locomotion introduce the concept of redirected walking. *Redirected walking* is the idea of tricking players into thinking that they are walking long distances by moving the virtual world in such a way that the players never leave their play space (for example, slowly rotating the world as a player walks down a corridor). You can also combat the space requirements by designing your levels so that a player never gets near the edges of the play space.

Advantages:

- Least likely to induce simulator sickness

Disadvantages:

- Limited by the player's physical space
- Limited to the HMD's tracking space

Variations:

- Redirected walking (moving the virtual world to keep the player inside the play space)

Teleportation

Teleportation is another one-to-one locomotion method that can be combined with natural locomotion to extend the range of the method beyond the player's physical space. It has few side effects in the way of nausea, but it can be disorienting to a user and may not suit all game themes. Many variations exist for teleporting; some allow users to see where they are in their physical room before they teleport, so that they can position themselves in the best possible place to allow for the most natural movement once they teleport. Other variations allow the teleportation to be controlled by a physical object (like a ball that needs to land where the player teleports). This physicality can prevent the issue of micro-teleporting, where users teleport multiple times in such quick succession that they experience vection as their brain senses small accelerations instead of instantaneous movement.

If you want to know how to implement teleportation in UE4, see [Chapter 5, "Teleportation."](#)

Advantages:

- Allows the player to move past physical boundaries
- Little nausea induced

Disadvantages:

- Can cause discomfort
- Disorienting
- Can still cause vection through micro-teleporting

Variations:

- Visualizing the play space before teleporting
- Physical object teleporting beacon (can eliminate micro-teleporting)
- Node based (allowing the player to teleport only to fixed locations)

Vehicle

Vehicle locomotion is a one-to-one method with the virtual vehicle (car, ship, or otherwise) as the reference frame for the user. Vehicles can reduce vection because they reduce the optic flow by encapsulating the user in stationary geometry. Having the stationary reference frame of the inside of the vehicle can also reduce simulator sickness, because players subconsciously feel as if the world is moving around them rather than that they are moving through the world, which is consistent with their vestibular system. Vehicles are not applicable to all experiences, but a variation may allow a temporary grid to be shown when players move, giving them the same grounding benefits as a vehicle (done in the UE4 VR Editor).

Advantages:

- Less optic flow means less vection
- Grounding decreases the potential for nausea

Disadvantages:

- Not suitable for many experiences

Variations:

- Temporary grid allowing the player to get the same grounding as a vehicle

Physical

Physical locomotion tends to be more novel than the other locomotion methods, and it can lead to exciting new game mechanics. Generally, physical locomotion involves some sort of physical action in the real world that gets translated into another motion in the virtual world. Examples are a skiing motion with some motion controllers to control your player's movement in a skiing game, or jogging on the spot to have the player move laterally in the game or experience. With most physical locomotion methods, there are still issues with vection caused by artificial movement, but players find this less disorienting because of what they expect to happen when they perform that action in the real world and the similar repercussions in the virtual one.

Advantages:

- Leads to novel locomotion methods that can suit specific games nicely
- Can reduce vection because of the user's expectation of the motion

- Disadvantages:
- Possibly gimmicky
 - Chance for simulator sickness in sensitive users

Variations:

- Skiing
- Climbing
- Flying (wing flapping)
- Running
- World pulling (pulling the world around the player with motion controllers)

Artificial

Artificial locomotion has little real-world player movement translated into the virtual world. Instead, it relies on more traditional game input (such as a gamepad thumbstick). Out of the locomotion methods presented, artificial causes the most simulator sickness, but it may also be the easiest method to use when porting a traditional game to VR. Some variations of artificial FPS (first person shooter) locomotion try

to reduce vection by limiting acceleration. Taking away the rotational acceleration by limiting the use of artificial turning (by either snapping the turning to increments or disabling it altogether) can help with this.

Advantages:

- Freedom to move around without dependencies on a player's physical space
- Quick to port non-VR games to VR

Disadvantages:

- Causes simulator sickness in many players

Variations:

- Snap turning
- Turret movement

Locomotion Implementation

Now, let's implement some locomotion methods. Natural locomotion is already implemented at an engine level by virtue of UE4's support of positional tracking in headsets that support that movement, so it's not covered here. Teleportation has an entire chapter dedicated to it, because it's so important to VR (refer to [Chapter 5](#)). For physical locomotion, we will look at how to create a basic running-in-place locomotion for the UE4 First Person Template later in this chapter, but before that, let's look at adapting the First Person Template for snap turning.

First Person Template for Snap Turning

A common method for artificial locomotion is traditional first-person controls. Luckily, UE4's default First Person Template has already implemented this, so it will be fairly easy to add some comfort features (like snap turning) for users who need it.

Project Creation

Some preliminary work has already been done to the First Person Template that comes with UE4 to support VR. However, to make sure it is fully compatible, you will need to make a few changes before implementing the snap-turning functionality:

1. Create a new Blueprint project based on the First Person Template.
2. Open the FirstPersonCharacter Blueprint (FirstPersonBP → Blueprints).
3. Add a new Scene Component named CameraRoot.
4. Set the newly added Component's Location to -96 in the Z-axis (see [Figure 9.1](#)). This ensures that the Camera is based on the character's contact point with the ground.

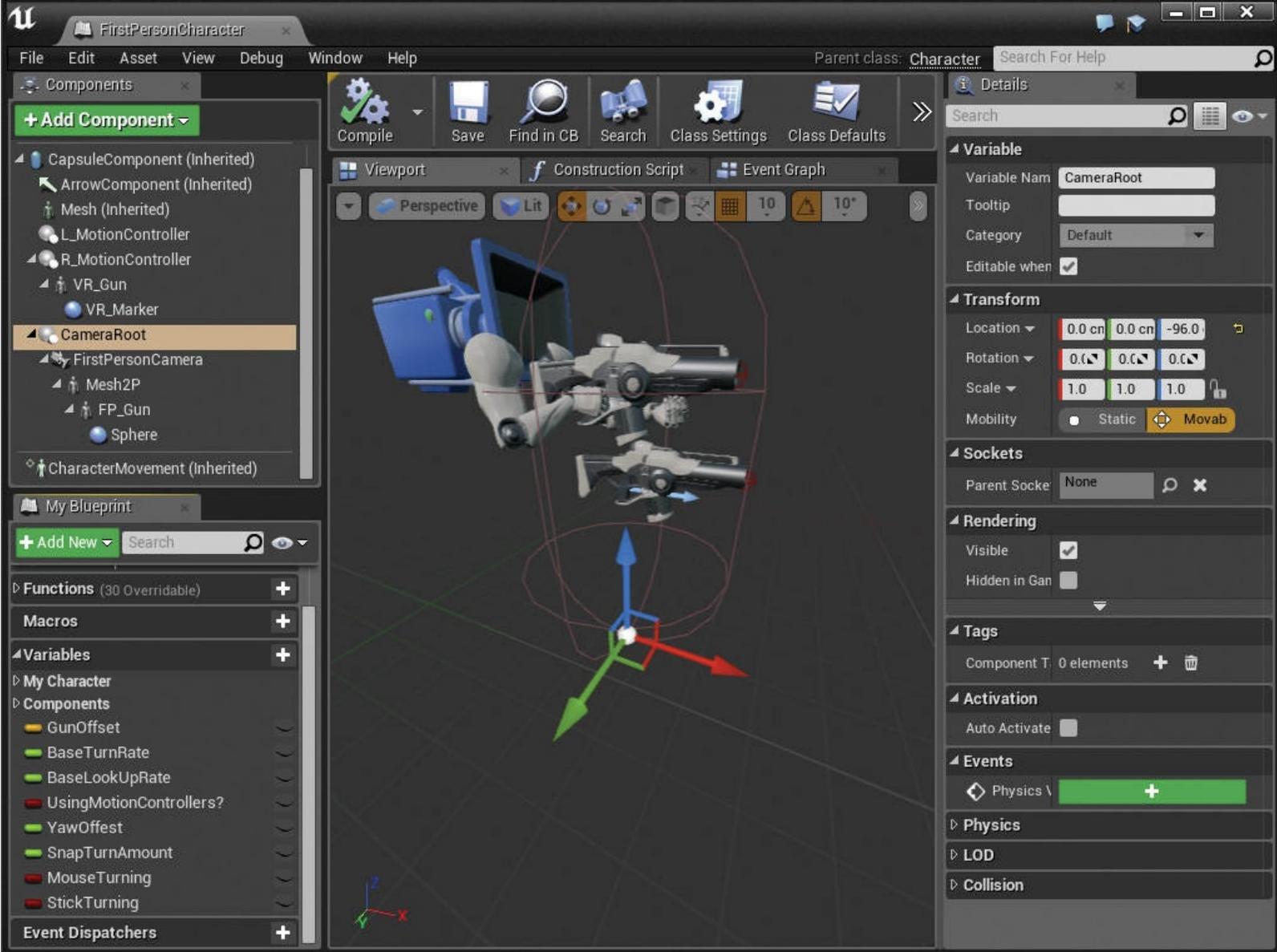


Figure 9.1 FirstPersonCharacter: adding a new CameraRoot

5. Drag the FirstPersonCamera Component on top of the CameraRoot to make it the parent.
6. Create two Float variables: YawOffset and SnapTurnAmount.
7. Set the default value for SnapTurnAmount to 15. This is how many degrees the snap turning should snap.
8. Create two new Boolean variables: MouseTurning and StickTurning. These will tell us whether the user is turning with the mouse or the gamepad later on.
9. In the Event Graph, find the EventBeginPlay node and, after the UseControllerRotationYaw node, create a new variable getter for the FirstPersonCamera Component and call the SetUsePawnControlRotation on it. Setting this to False allows the character to use the positional tracking of the headset (setting this variable to True makes it easier to rotate the Camera, so you will leave it as True when the player is using a monitor).
10. Create another new node, this time a SetTrackingOrigin function call, setting its Origin input to Floor Level. This fixes any tracking issues with headsets that default to Eye Level tracking origin.
11. Connect the two newly created nodes between the UseControllerRotationYaw and the Branch node (see [Figure 9.2](#)).

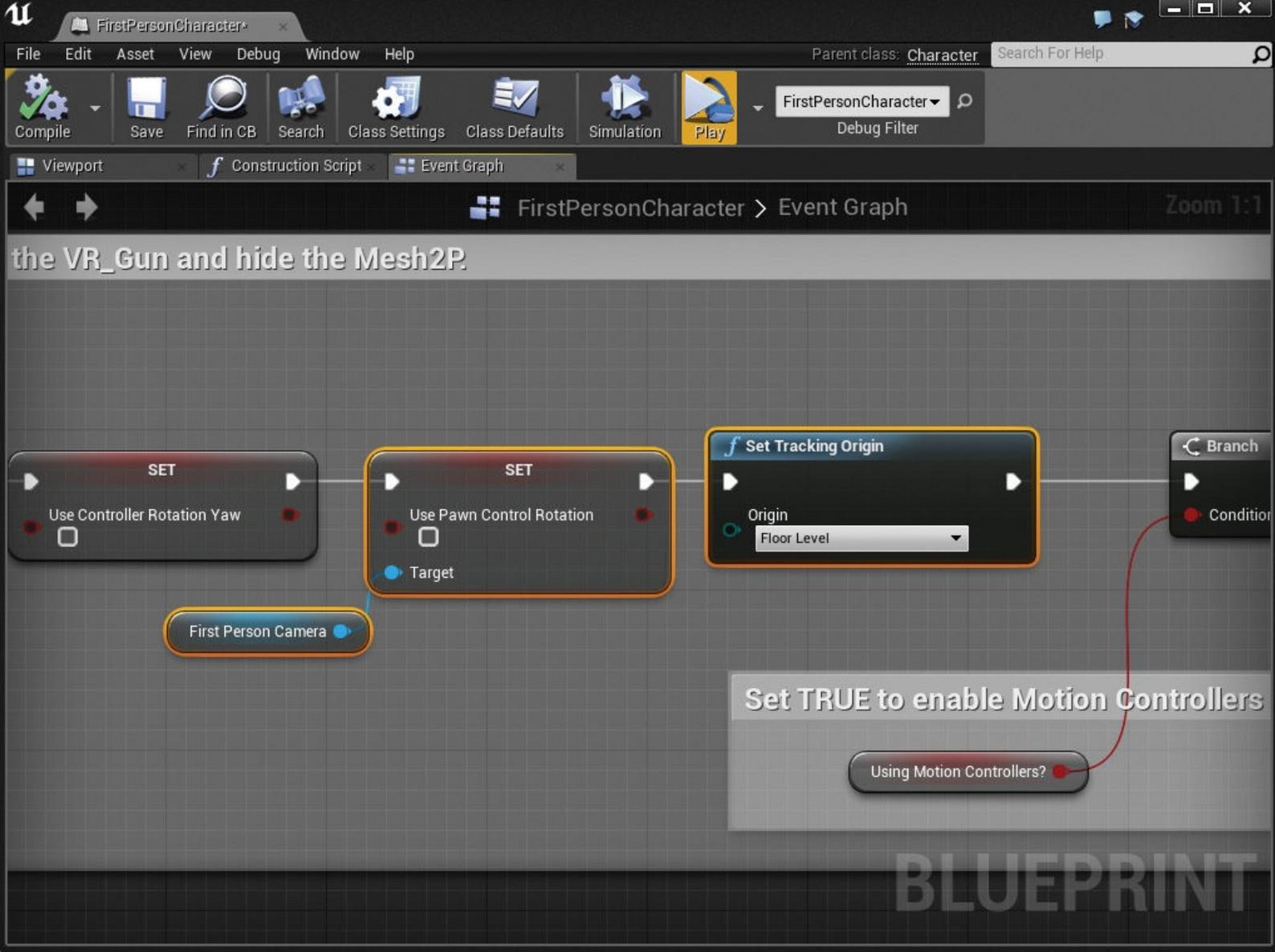


Figure 9.2 FirstPersonCharacter: fixing positional tracking and forcing a new tracking origin

Implementing Snap Turning

To implement snap turning, you will create a new function that allows you to rotate the Camera in fixed increments. To do this, you use the YawOffset variable to hold how far the character would have turned if using the normal turning method, and when that exceeds the turning threshold, snap to that turning amount.

1. Create a function called AddControllerYaw with two Float inputs, Float and Snap Amount (see [Figure 9.3](#)). Open it.

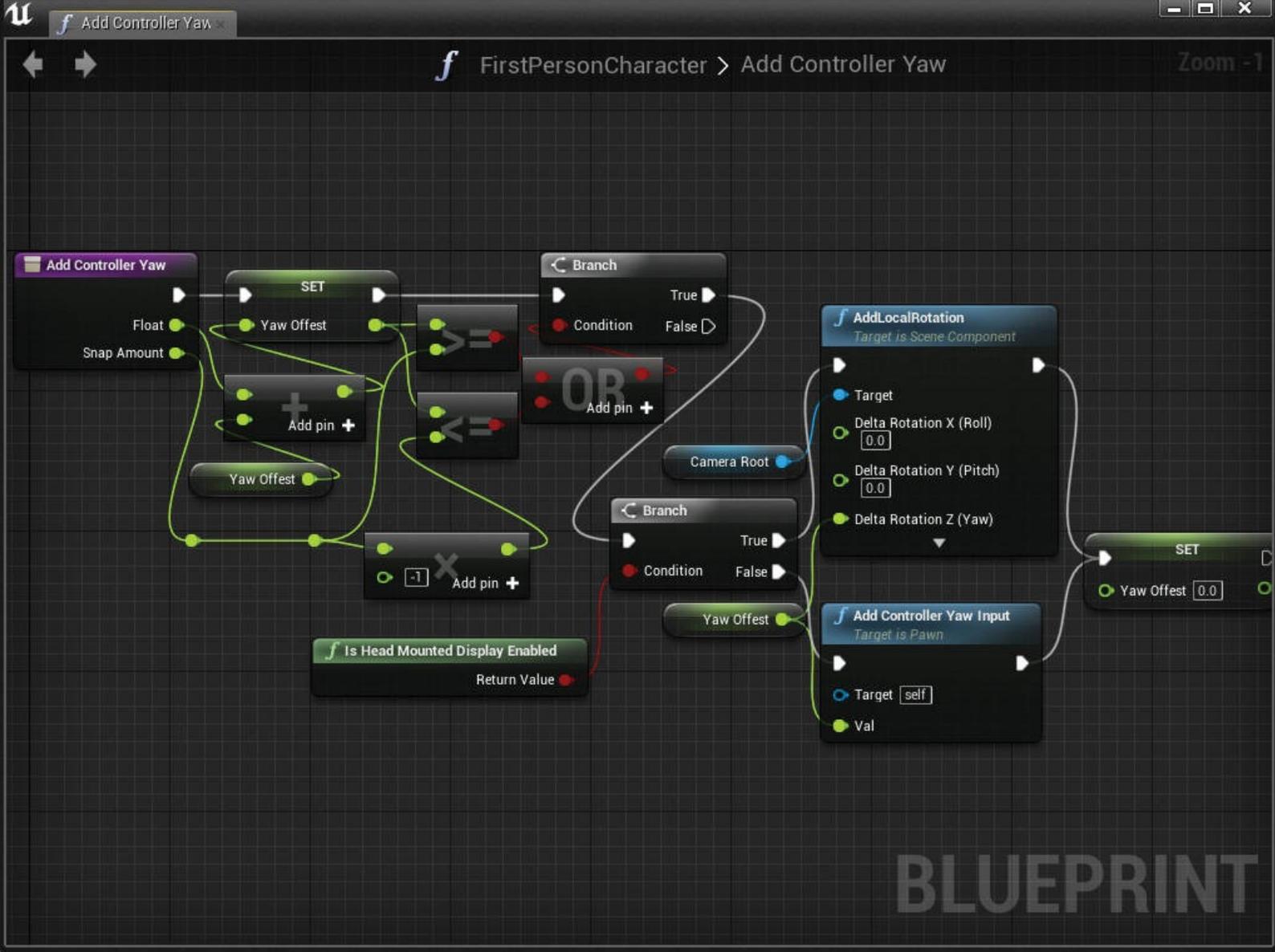


Figure 9.3 FirstPersonCharacter: custom rotation function for snap turning

2. Create a new getter and setter for the YawOffset variable, connecting the input execution pin of the setter to the input pin of the function.
3. Drag off of the Float input pin of the function and add a new Float + Float node, passing in the YawOffset getter to the second pin.
4. Connect the output of this Float + Float to the YawOffset setter.
5. Drag off of the output of the YawOffset setter and call the Float >= Float function, passing in the Snap Amount input pin to the second pin of this function.
6. Drag off of the output of the YawOffset setter again and call the Float <= Float function, this time passing in a Float * Float of the Snap Amount input and -1 (see [Figure 9.3](#)) as the second pin.
7. Connect the output of both the >= and <= functions with an OR gate. This means that the rest of the function is triggered when the input is either greater than the snap amount or less than the negative of the snap amount.
8. Create a new Branch node, connecting the output of this OR gate to the Condition input.
9. Connect the input execution pin of the Branch to the output execution pin of the YawOffset setter.
10. After this Branch's True output create a new Branch node, connecting an IsHeadMountedDisplayEnabled pure function call to the Condition input (see [Figure 9.3](#)).

11. Create a new CameraRoot getter and call the AddLocalRotation function on it.
12. Break the input of the Delta Rotation input and pass in the YawOffset variable to the Delta Rotation Z (Yaw). This will rotate the Root Component of the Camera; you will not be able to rotate the Camera directly while using an HMD.
13. Connect this AddLocalRotation node to the True output of the Branch.
14. Create a new setter for the YawOffset once more, this time leaving the input at 0.0 and connecting it to the output of the AddLocalRotation.
15. Drag off of the False output of the Branch and call the AddControllerYawInput function, passing in the YawOffset and connecting its output to the same YawOffset setter as in the previous step.
16. Back in the Event Graph find the comment labeled “Stick input” and replace the call to AddControllerYawInput with your custom AddControllerYaw function, passing in the SnapTurnAmount variable as the Snap Amount (see [Figure 9.4](#)).

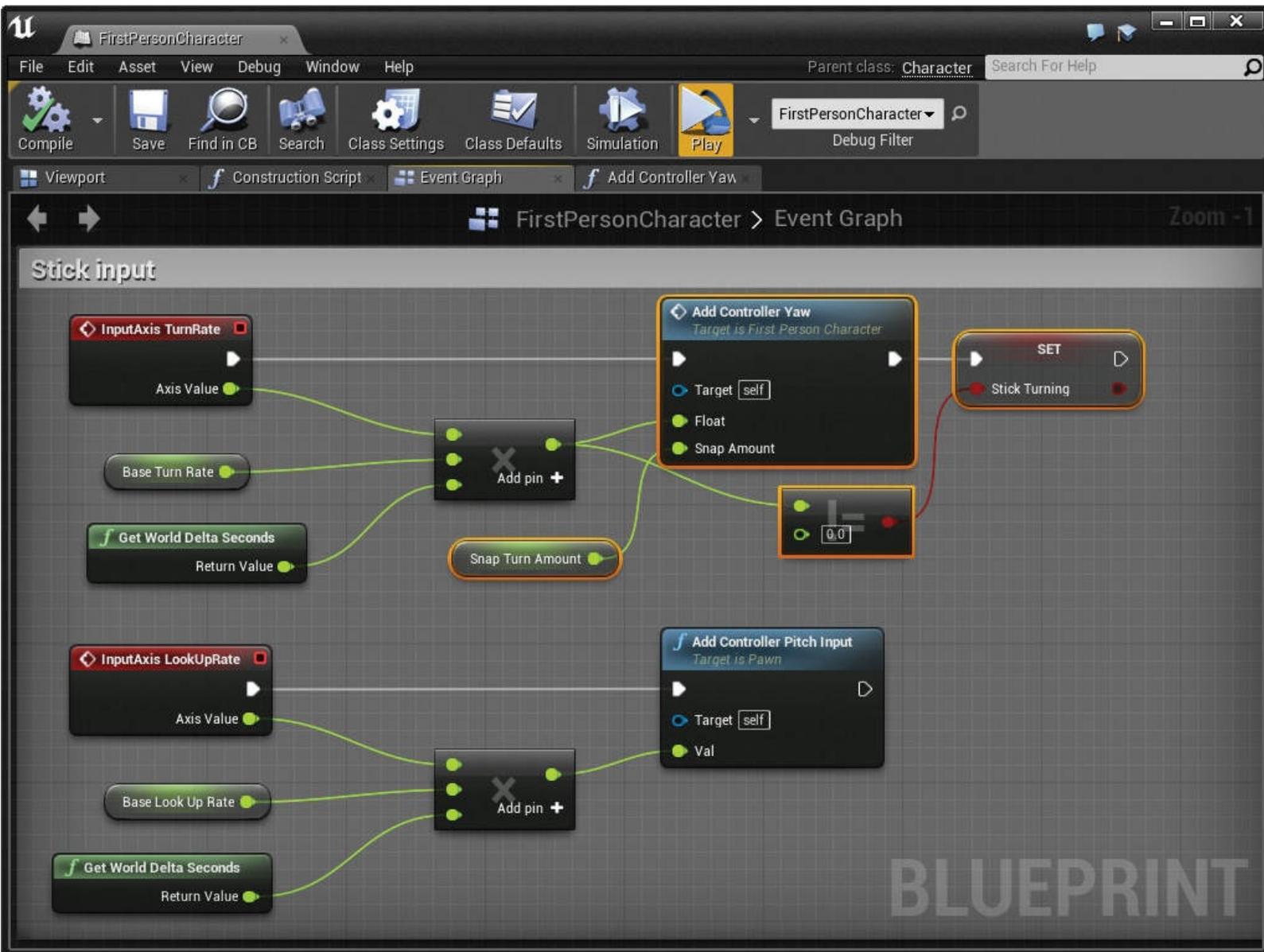


Figure 9.4 FirstPersonCharacter: replacing the default stick yaw turning with the snap-turning function

17. Drag off of the three-way Multiply, create a new Float != Float node, and pass it into a new setter for the StickTurning variable, connecting it to the end of the AddControllerYaw function (see [Figure 9.4](#)).
18. Find the comment labeled “Mouse input” and replace the call to AddControllerYawInput once

more with your custom function, passing in the same SnapTurnAmount variable to the second input again (see [Figure 9.5](#)).

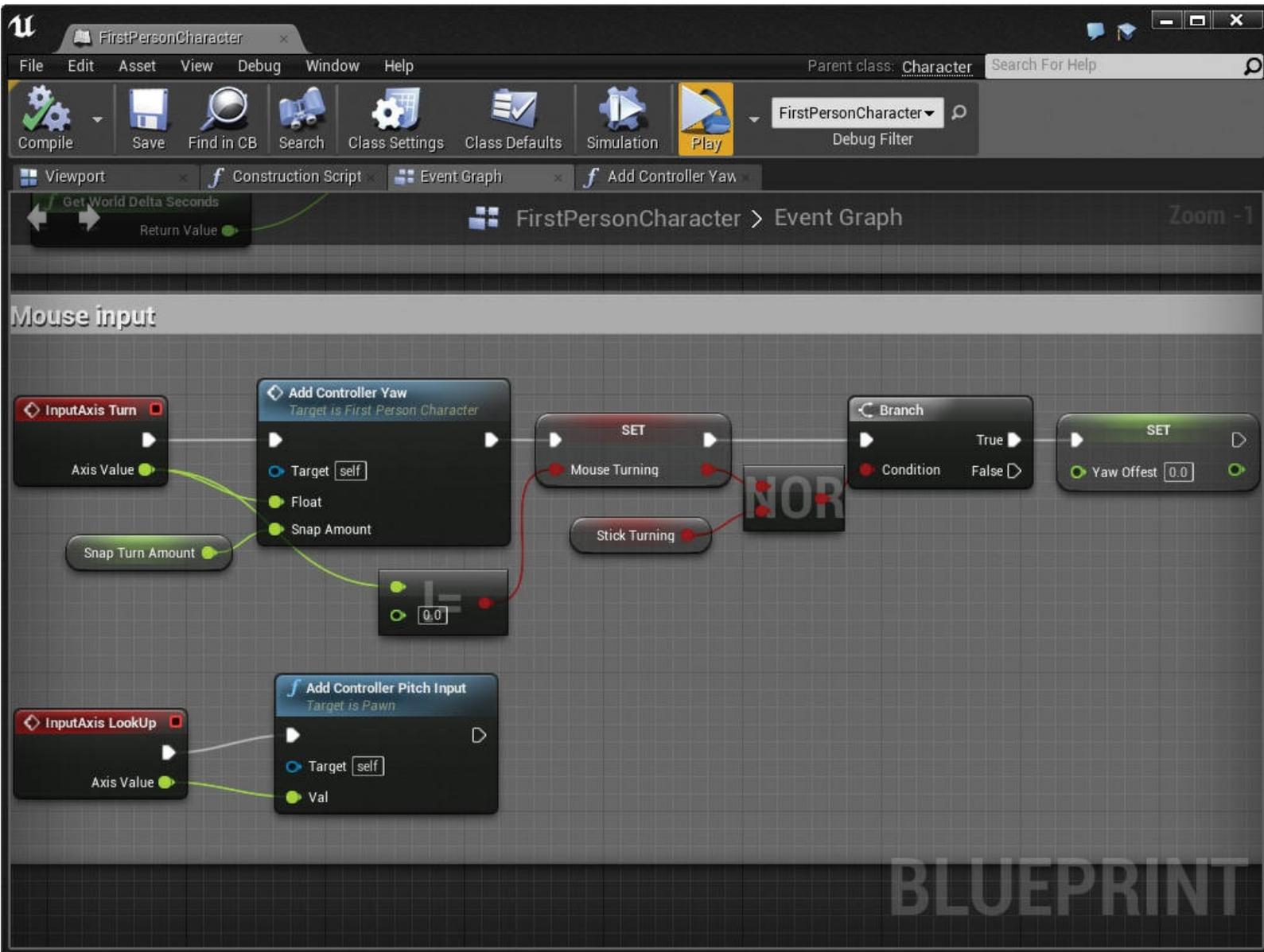


Figure 9.5 FirstPersonCharacter: replacing the default mouse yaw turning with the snap-turning function

19. Drag off of the InputAxisTurn's Axis Value, create a new Float != Float node, and pass the output to a new setter for the MouseTurning variable, connecting it to the end of the AddControllerYaw function call.
20. Drag off of the red output pin of this setter call and create a new NOR gate, passing in the StickTurning variable to the second input.
21. Create a new Branch node connected to the output of the setter from step 19, passing in the output of the NOR to the Condition input. This allows you to reset the YawOffset value to 0 when the player is using neither the mouse nor the joystick.
22. Drag off of the True output and create a new setter for the YawOffset variable, leaving the input as 0.0 (see [Figure 9.5](#)).
23. To add support for using the shoulder buttons on the gamepad to snap the user, create two new input events for GamepadRightShoulder and GamepadLeftShoulder (see [Figure 9.6](#)).



Figure 9.6 FirstPersonCharacter: snap turning with the gamepad shoulder

24. On the GamepadRightShoulder event, call the AddControllerYaw function, passing in the SnapTurnAmount to both the Float and Snap Amount inputs.
25. On the GamepadLeft Shoulder event call the AddControllerYaw function once more, this time passing in a Float * Float with the SnapTurnAmount and -1 into the Float input pin (see [Figure 9.6](#)).

First Person Template for Running in Place

An easy first physical locomotion system to implement is a basic “running in place” system; in this system when the player wearing the headset jogs up and down on the spot, the virtual player moves forward in the game world.

To implement this running motion, we continue from the previous section where you added snap turning to the default UE4 First Person Template: **1.** Open the FirstPersonCharacter Blueprint (FirstPersonBP → Blueprints).

2. Create two new Vector variables named HMDVelocity and PreviousHMDPosition. These hold the current HMD velocity and the previous frame’s head position, which will help you calculate that velocity.
3. Create another new Vector variable, this time making it an Array, and name it PreviousVelocities. You use this to smooth out any high-frequency motion that may come from the user’s head.

4. Create three new Float variables: RunningSpeed, RunningThreshold, and RunningMultiplier. RunningSpeed is the current target running speed based on the player's head movement, RunningThreshold is the amount RunningSpeed must surpass to be added to the player's movement, and RunningMultiplier is used to scale the player's head velocity, so that you can scale the player's head velocity to sensible distances for moving (see [Figure 9.7](#)).

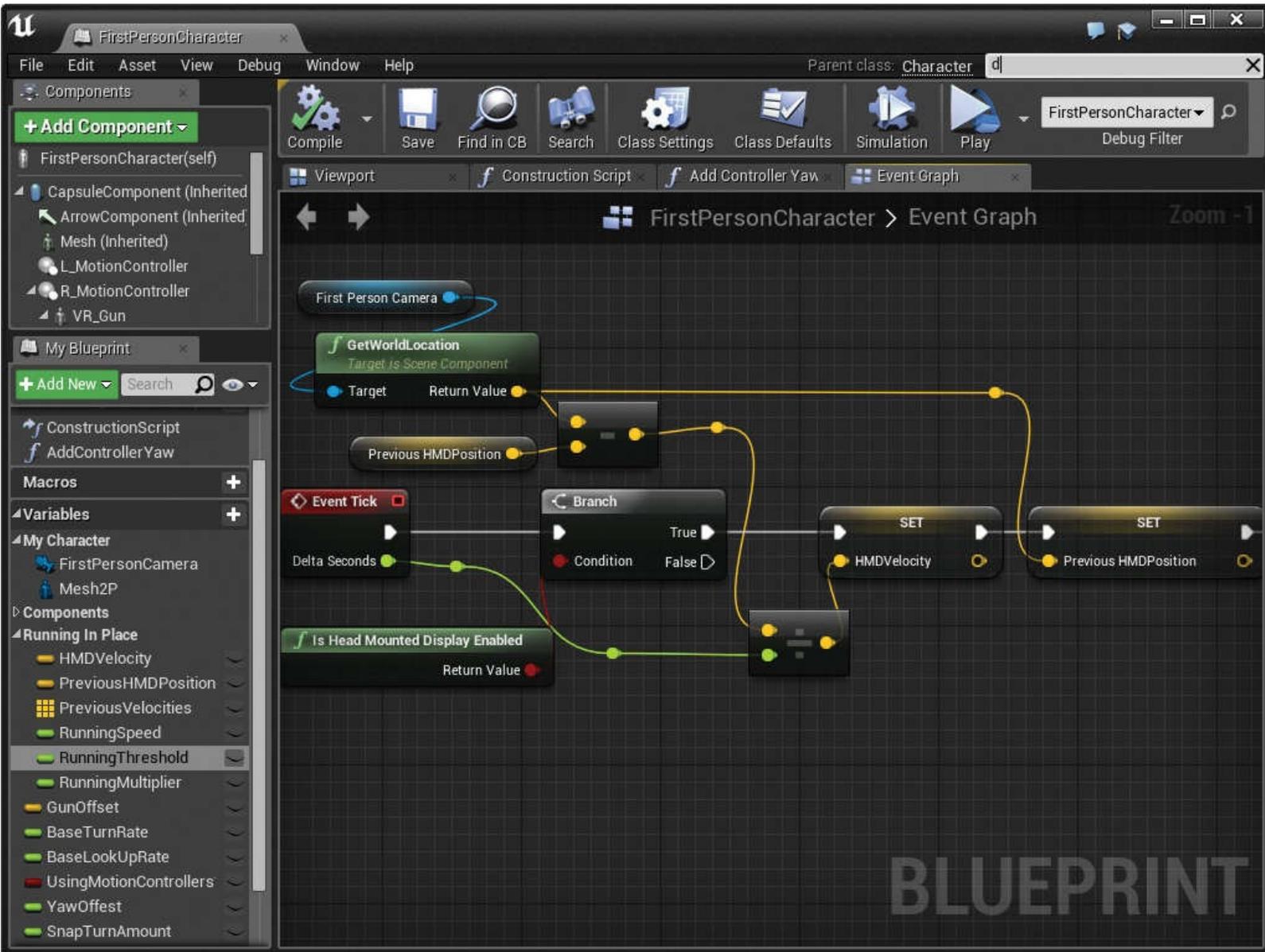


Figure 9.7 FirstPersonCharacter: running-in-place variables and initial velocity calculation

5. Set the default value of RunningThreshold to 1.5, and the default value of RunningMultiplier to 0.05.
6. In the Event Graph drag off of the EventTick and create a new Branch node, connecting a new IsHeadMountedDisplayEnabled function call to the Condition input.
7. Connected to the True output pin of this Branch, create a new setter for the HMDVelocity variable.
8. Create a new getter for the FirstPersonCamera Component and call the GetWorldLocation function.
9. Drag off of the Return Value of this function call and create a new Vector – Vector node, passing in a getter for the PreviousHMDPosition variable to the second input. This will calculate the distance between the previous HMD position and the current one.
10. Drag off of the output of this Vector – Vector node, create a new Vector / Float node, and pass in the Delta Seconds from the EventTick (see [Figure 9.7](#)). This will convert the difference from the Vector – Vector in centimeters into a velocity in centimeters per second.

11. Connect the output of the Vector / Float into the input of the HMDVelocity setter.
12. Create a new setter for the PreviousHMDPosition variable after the HMDVelocity setter and pass in the GetWorldLocation from step 8.
13. After the PreviousHMDPosition setter create a new getter for the PreviousVelocities array and call the Insert function.
14. Create and pass in a new getter for the HMDVelocity variable to the input pin of the array insert. This will add the current HMD velocity to the first section of the PreviousVelocities array.
15. Create a new getter for PreviousVelocities once more and call the Resize function, passing in the number 5 to the input pin and connecting it to the output execution pin of the array insert (see [Figure 9.8](#)). This will cause the array you are about to average over to hold the last five frames.

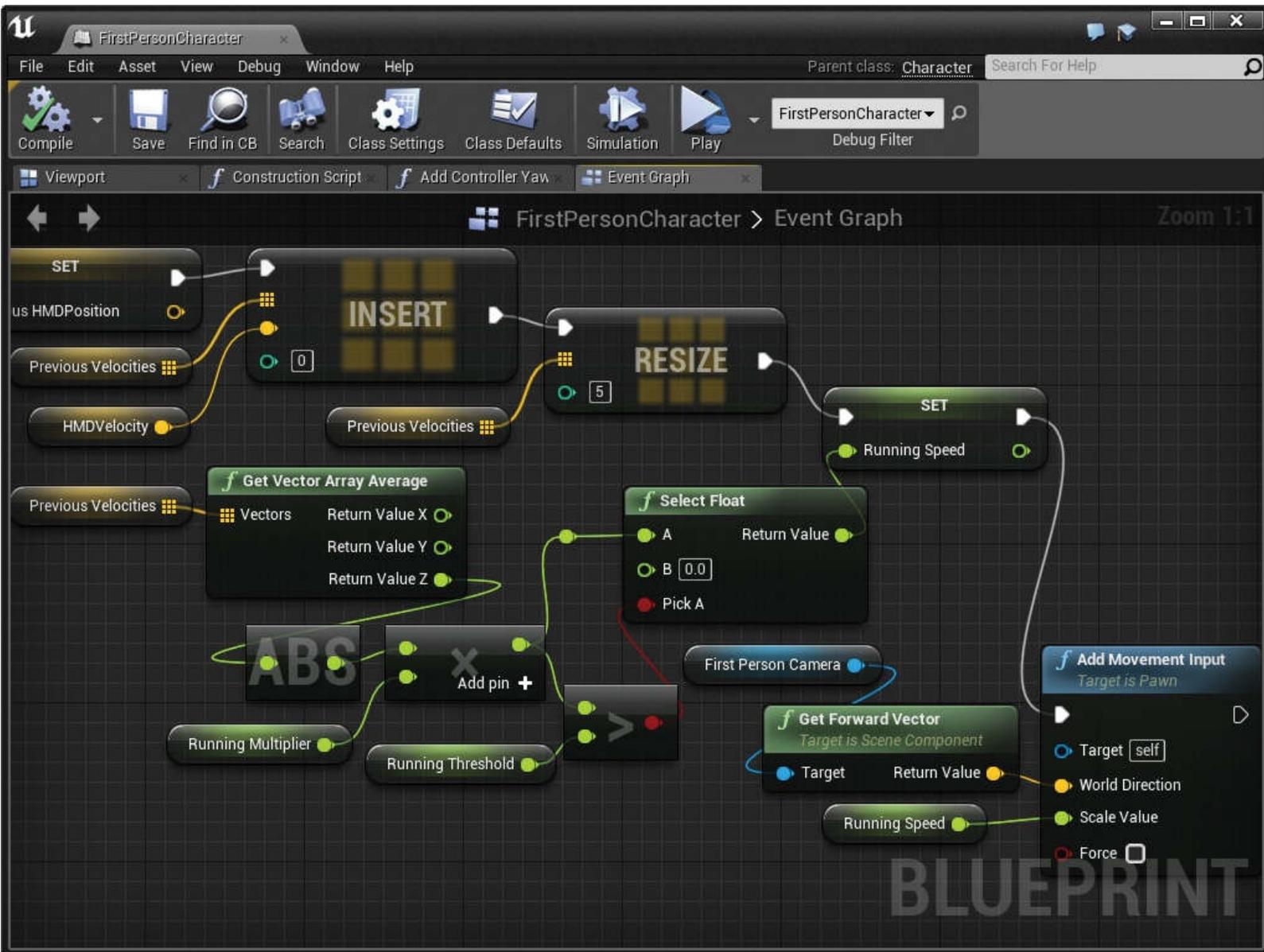


Figure 9.8 FirstPersonCharacter: running-in-place five-frame velocity averaging and adding movement input

16. Create one last getter for the PreviousVelocities variable and call the GetVectorArrayAverage function.
17. Right-click the Return Value of this function and split the struct pin.
18. Drag off of the Return Value Z and call the Absolute (float) function; this allows you to disregard whether the player is jogging up or down.

19. Connected to the output of this Absolute create a new Float * Float node, passing in a new getter for the RunningMultiplier variable to the second pin.
20. Drag off of the output of the Float * Float node and call the SelectFloat function. This ensures that the head movement is over the threshold.
21. Drag off of the Float * Float node once more and create a new Float > Float node, passing in the RunningThreshold to the second pin.
22. Attach this Float > Float node to the Pick A input pin of the SelectFloat; this ensures that the running speed will be 0 so long as the player's head velocity is under the threshold.
23. Create a new setter for the RunningSpeed variable and connect it to the output execution pin of the Resize function.
24. Pass in the output of the SelectFloat to the green input pin of this setter.
25. To actually move the player, first call the AddMovementInput function.
26. Create a new getter for the FirstPersonCamera Component and get its ForwardVector (see [Figure 9.8](#)), passing it into the World Direction of the AddMovementInput function.
27. Create a new getter for the RunningSpeed variable and pass that into the Scale Value of AddMovementInput.

You can now hit Play with your favorite HMD connected and, if it supports positional tracking, jog on the spot to see your character move.

Summary

You now understand simulator sickness and some of its potential causes. You also understand some of the VR locomotion methods that exist today and how they are implemented in UE4.

Exercises

The only locomotion method not covered so far is vehicle. And that's for a good reason: Epic already has a good driving template that fully supports VR. Create a new project with this template and see if the reduced optic flow helps with simulator sickness.

Chapter 10. VR Optimization

This chapter covers rendering requirements for VR. These requirements are different from those for traditional monitors, but some tricks can be implemented to ease the burden of this demanding medium.

Requirements of VR Rendering

Rendering a VR scene poses many new issues and challenges that do not exist when trying to render a traditional game or experience. VR has a strict need for low-latency rendering to avoid simulator sickness; however, in the past this part of the rendering pipeline has often been sacrificed because of the race for higher and higher performant engines.

An ideal VR experience has a motion-to-photon time (time from the sensor detecting motion to the user seeing it displayed on the screen) of less than 20ms. This stringent requirement has led to the use of high-refresh-rate panels (from 60Hz to 90Hz to even 120Hz). However, just having a high-refresh-rate panel isn't enough. Because a VR headset's display is close to the user's eyes (literally centimeters away and with the lenses acting like a magnifying glass), the display must have a high enough resolution that it does not detract from the experience. Therefore, we see refresh rates from 60Hz to 120Hz with display resolutions from 1920x1080 to 2560x1440. These numbers are high from a regular gaming perspective, but they don't tell the whole story.

To get an idea of how demanding VR rendering can be, you need to know how and why VR lenses work. In a nutshell, to focus on objects in the real world, the muscles in your eyes push and pull on your eyes' lenses and allow certain beams of light to converge on your retina. However, these muscles are unable to focus on objects close to your eyes. Unfortunately, for a head-mounted display this is bad news, because to get a high FOV (field of view), the display needs to be close to the eyes. To counteract this, you can put lenses between the user's eyes and the display; they bend the light coming from the display and trick the eyes into thinking that the screen is farther away than it actually is, hence solving this focus problem. But these lenses aren't perfect, and they actually distort the image coming from the display. To counteract this, you must render your virtual images with the opposite distortion so that your worlds feel as accurate as possible (see [Figure 10.1](#)).

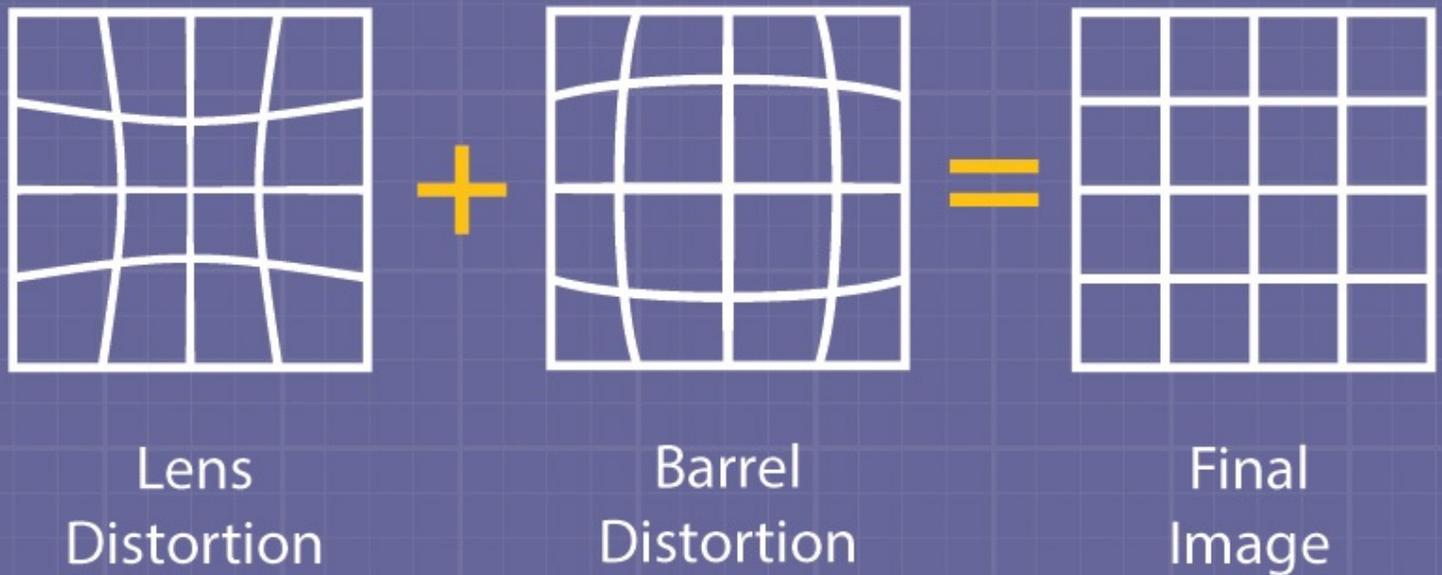


Figure 10.1 Correcting for VR lens distortion

This is all well and good, but barrel distortion actually causes a loss of some perceived resolution on your image. To make up for this loss in quality, many VR games and experiences render 1.3 to 1.5 times the display's native resolution. (In UE4, this is referred to as *screen percentage*, and it is an important value in tuning the performance of your experiences.)

Therefore, if you look back at your calculation of how demanding the VR experience is, you have to multiply everything by 1.5; so for the Rift CV1 and Vive (both headsets having two screens at 1080x1200 each), the total resolution is now $2160 \times 1200 \times 1.5$, so 3240×1800 and at 90 frames per second, or around 350 million pixels per second! In contrast, a 1080p game at 60 frames per second is only 124 million pixels per second.

Latency Mitigation

As you've just seen, latency in a VR headset can have some dire consequences for user comfort and whether or not your VR experiences are compelling. This requirement is quite a lot more important to VR than to any other medium that has come before it, but there are various techniques that can help reduce or eliminate major causes of latency in the rendering pipeline. This chapter looks at two main techniques that aim to address this: timewarping/reprojection and front buffer rendering.

Before we get into what exactly timewarping/reprojection and front buffer rendering are, let's take a quick trip down the rendering pipeline to gain a better understanding of how these features fit into your experiences. (Although developers not focused on rendering rarely have to deal with these details, and UE4 does a great job of implementing them, it is still important to learn the nitty-gritty to help in diagnosing problems that may arise.)

To render a frame in a game, many pieces work together to produce the final frame. On the CPU side, it polls the user input, does work on the game (simulates physics, calculates the artificial intelligence movement, etc.), and ultimately submits the draw calls to be carried out on the GPU. Traditionally after

the GPU has finished rendering those draw calls, it presents them to the display for scan-out, which then (from left to right, top to bottom) displays the image to the screen for the user to see. A simple version of this is shown in [Figure 10.2](#).

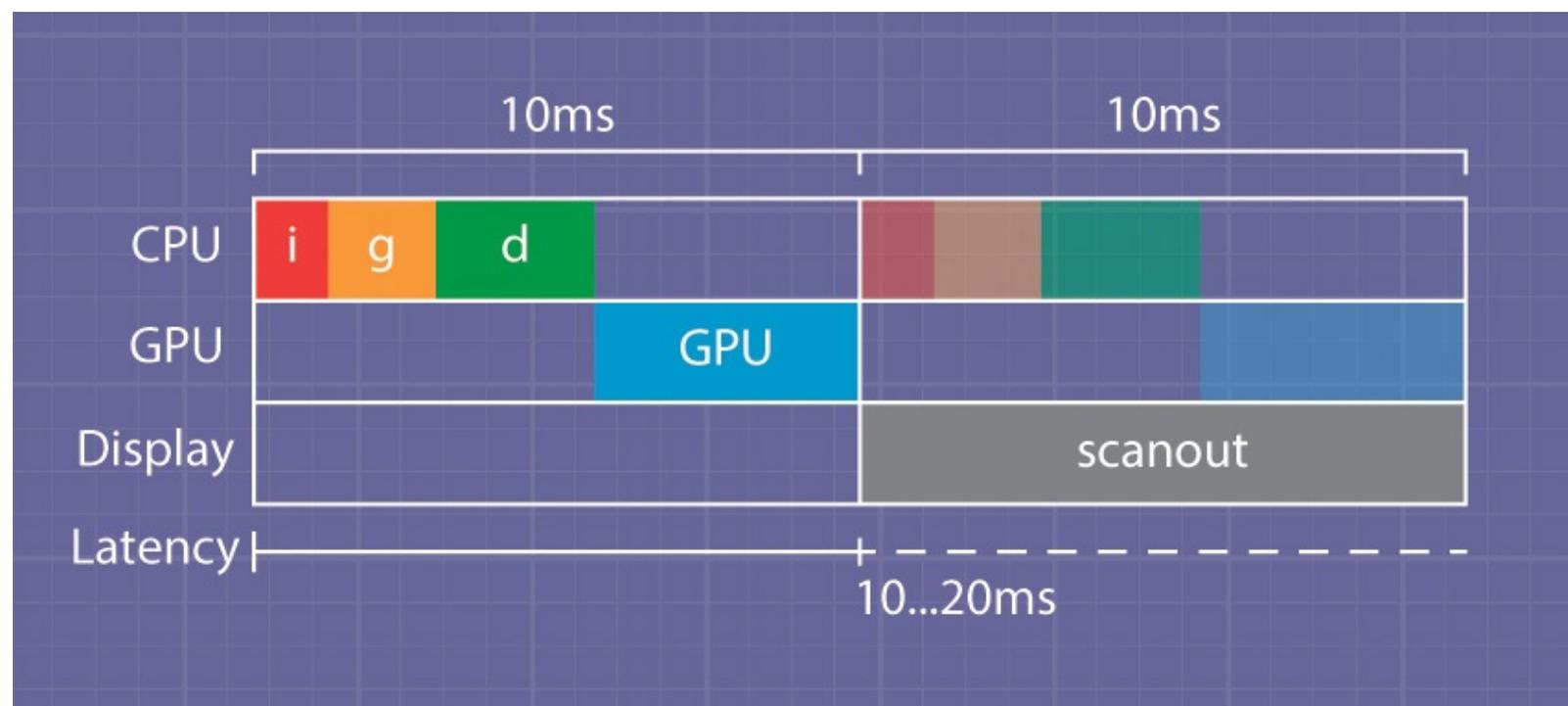


Figure 10.2 Simple rendering pipeline with one frame of latency; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

Note

Most VR headsets implement low-persistence technology that skips displaying the image to the user until a fraction of a second before the next display refresh to prevent the user from viewing “bad” information. However, the rest of this section assumes a traditional display for simplicity.

Notice that [Figure 10.2](#) is not the most optimal setup in terms of performance. There are long down periods when the GPU and CPU are doing nothing while they wait for each other to finish. Also notice that there is an entire frame of latency from the input (i) and the scan-out.

To fix these issues, we can first try to fix the performance bubbles (sections of inactivity) by introducing a pipelined architecture, as shown in [Figure 10.3](#).

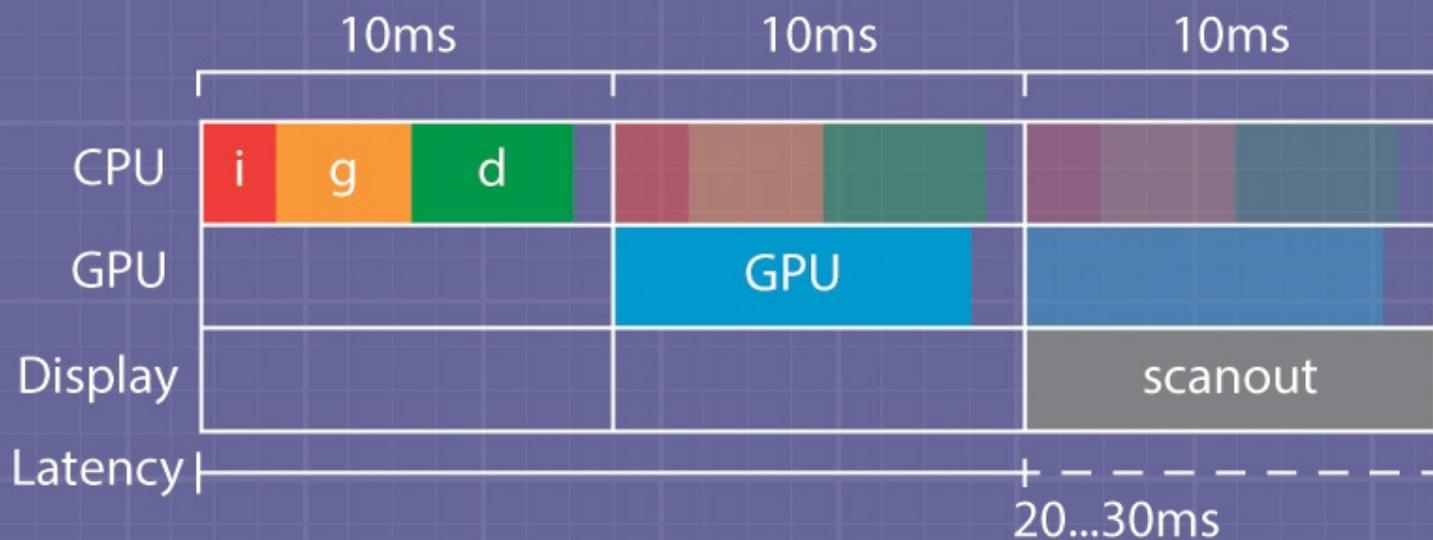


Figure 10.3 Simple rendering pipeline with two frames of latency and a pipelined architecture; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

Compared to [Figure 10.2](#), [Figure 10.3](#) now has better utilization of both CPU and GPU; they don't have to share the same 10ms interval because the GPU now renders the previous frame while the CPU is free to start its work on the next frame. However, you will notice that there is now another entire new frame of latency.

Another issue arises when that pesky draw task begins to take up too much of the 10ms window you have to render. Luckily, we now have multicore CPUs, and if you're willing to give up another frame of latency (and in fact many game engines make this trade-off), you can run the draw task on a separate core while the next frame's input and game calculations are made (see [Figure 10.4](#)).

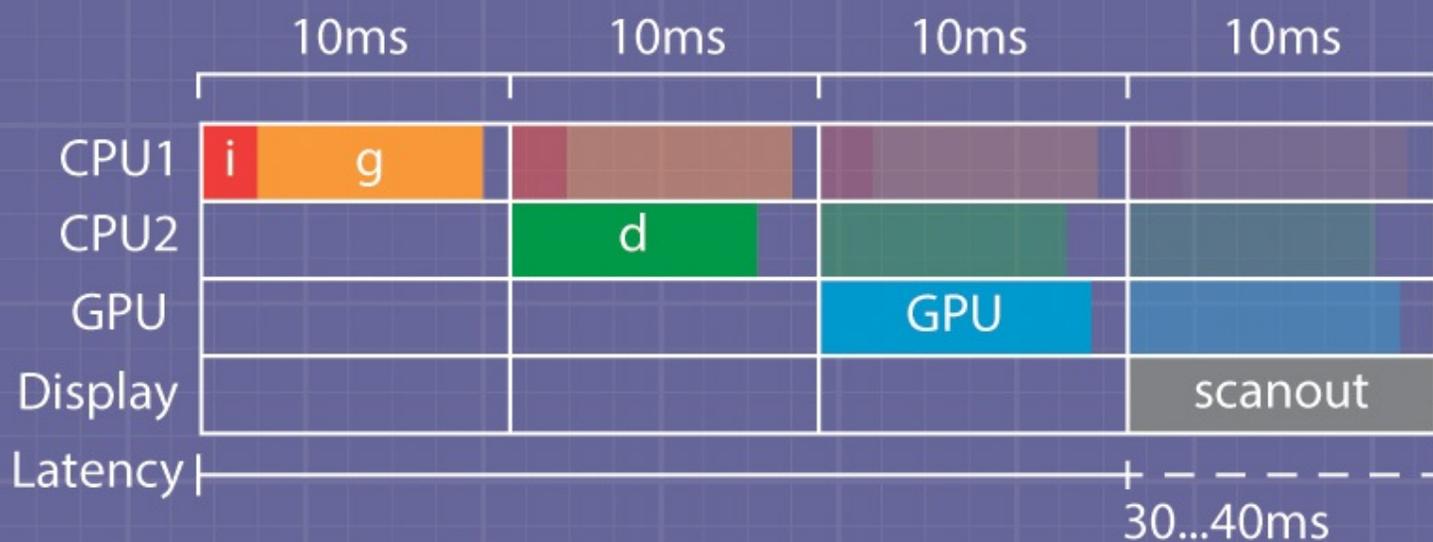


Figure 10.4 Simple rendering pipeline with three frames of latency and a pipelined architecture with a frame for draw tasks; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

However, it turns out that this needn't be the case. Much of the time, you can reduce the three frames of latency from [Figure 10.4](#) to only two by giving the draw tasks enough of a head start on the GPU that the CPU can start feeding the GPU data with only a small delay (a nice trade-off to get a whole frame of latency back; see [Figure 10.5](#)).

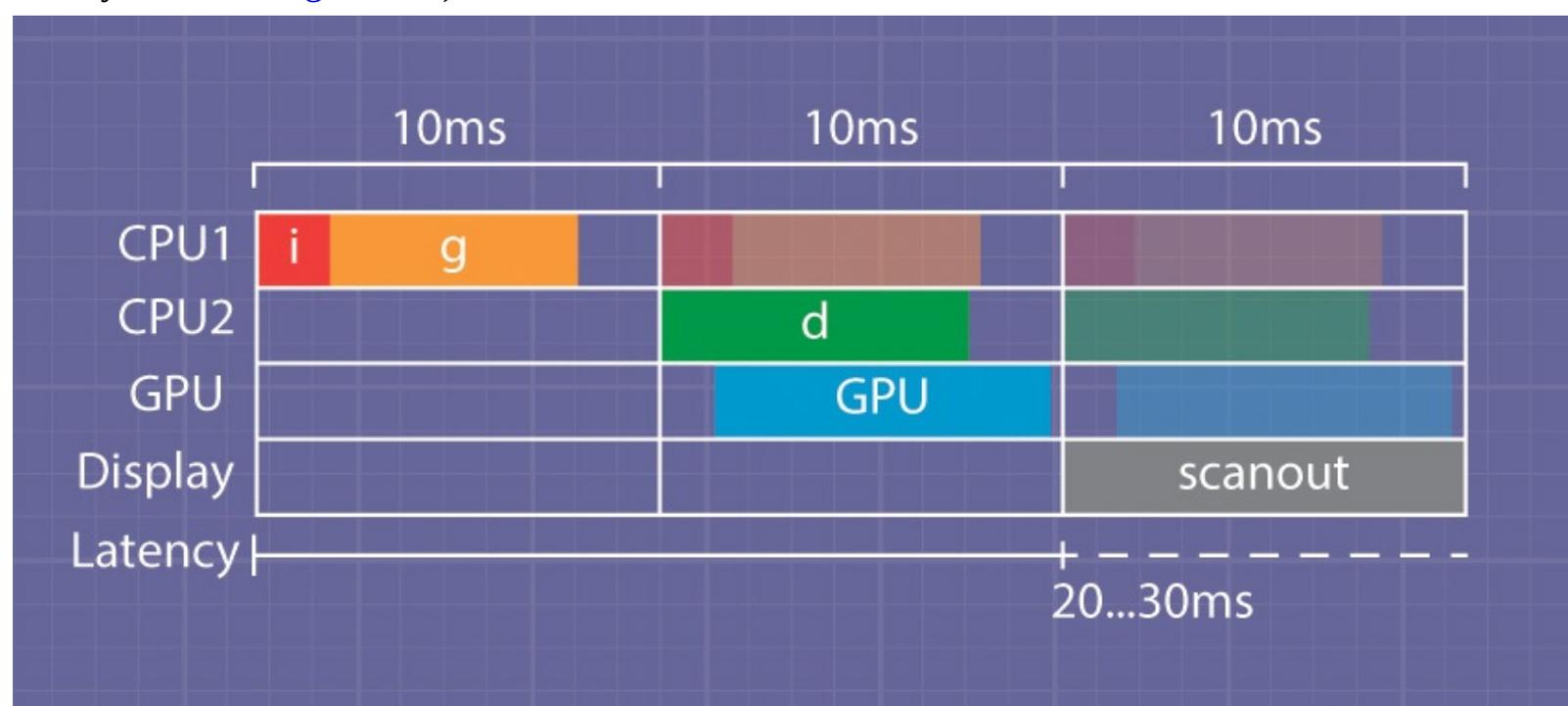


Figure 10.5 Simple rendering pipeline with two frames of latency and a pipelined architecture running the draw and GPU in parallel; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

[Figure 10.5](#) shows a nice balance between performance and latency, but is there more we can do? Of course! The keen-eyed among you will have noticed that bubble of inactivity while the GPU waits to be fed draw calls and may have been wondering if there was something you could do about it. Well, yes, there is, and in fact both Oculus and Valve handle this with what they call Adaptive Queue Ahead and Running Start. Basically, in terms of [Figure 10.5](#), they simply move the green draw task to before the second frame to give the GPU the most time to process all the draw calls.

This is all well and good for making sure you optimize CPU and GPU usage, but what about input latency? After all, VR is all about reducing latency, isn't it? Well, yes, and there are a few things you can do to help with this. First, notice that the input is being polled at the start of each frame. So, even in the best case, with no pipelining, there is still an entire frame of latency. But why can't you just move the input polling to the end of the frame? Well, you can! In fact, that is exactly what UE4 does with the Camera Component and the Lock to HMD flag and the Motion Controller Component with the Low Latency Update flag. This sort of input late updating is shown in [Figure 10.6](#).

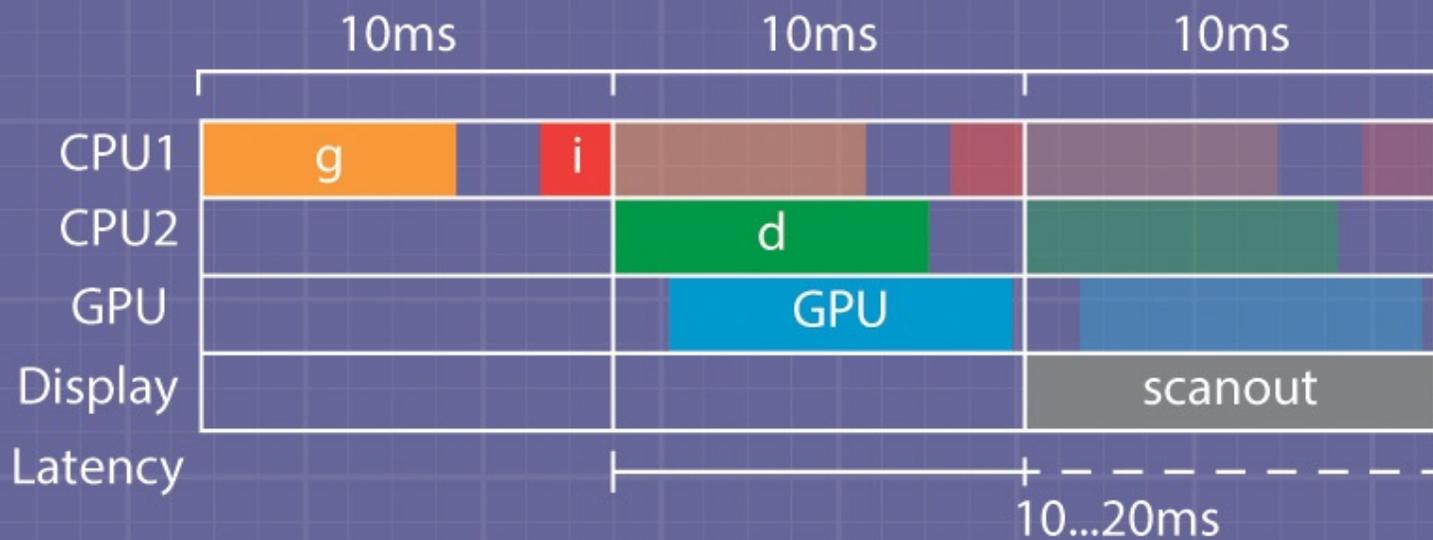


Figure 10.6 Simple rendering pipeline with one frame of latency and a pipelined architecture running the draw and GPU in parallel; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

This looks better, but there is one more thing you can do: timewarping. The basic idea of timewarping is to take the render data from the GPU, then transform that image to rotate the camera based on a newer input. As shown in [Figure 10.7](#), you can now reduce the rotational latency of the headset to only 2ms for the first part of the render. (Note that not all SDKs/runtimes implement timewarping, and those that do implement it differently. Therefore, be aware of those differences when targeting your experiences. Also note that this feature is enabled through the various runtimes and not the engine, so the developer does not need to do anything to enable it.)

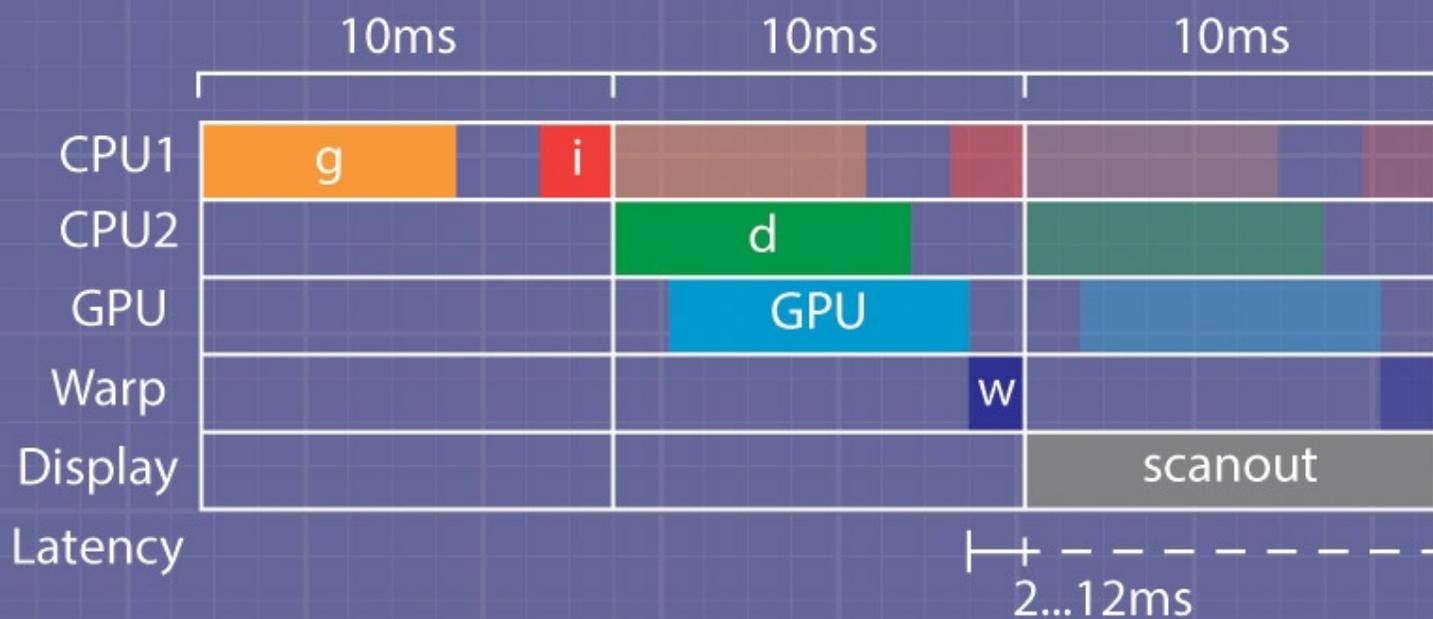


Figure 10.7 Simple rendering pipeline with less than one frame of rotational latency due to timewarp; i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

One last point, however. Previously I said that the GPU presents the rendered frame to the display, but that

isn't entirely true. In fact, until now what the GPU has been doing is rendering to what is called a back buffer, whereas what the display is actually reading from (the front buffer) is not directly touched by the GPU. This is because of the nature and disparity of how a display refreshes itself (remember, left to right, top to bottom) and how a GPU renders the geometry in the scene. A GPU doesn't neatly render from top left to bottom right like a display does; it instead renders each draw call it gets from the CPU in turn, one on top of the other, like a painter laying down strokes on a canvas. Therefore, you run into this pesky problem when rendering directly into the buffer from which the display is reading; that is, if the display begins to show a new frame before you are finished rendering it, it may in fact display an incomplete image to the screen with whole geometry from your game missing. To solve this, a back buffer is introduced; the GPU renders to it, and when the display is ready for a new image, it swaps the buffers if the frame is done. If the frame is not done just yet, the GPU can simply display the old image again, never showing an incomplete image.

Why is this important? Well, as it turns out, the timewarp operation does produce an image from left to right, top to bottom, and in fact when timewarping is in effect there is no real need for this double buffering because you can ensure that you start your timewarp with enough time to literally race the display refresh until it's finished. This is what is called *front buffer rendering* or *scanline racing*; it is currently implemented on both Gear VR and Daydream VR, and it allows for some pretty neat things. Because you are warping the image milliseconds before it is displayed on the screen, there is actually a chance that you can update the inputs to your warp as the display refreshes, and on a nearly line-by-line (if you time it well enough) basis you can warp the latest information to the display. This is in fact how the Gear VR renders and performs two warps each refresh (one for each eye), which is displayed in [Figure 10.8](#).

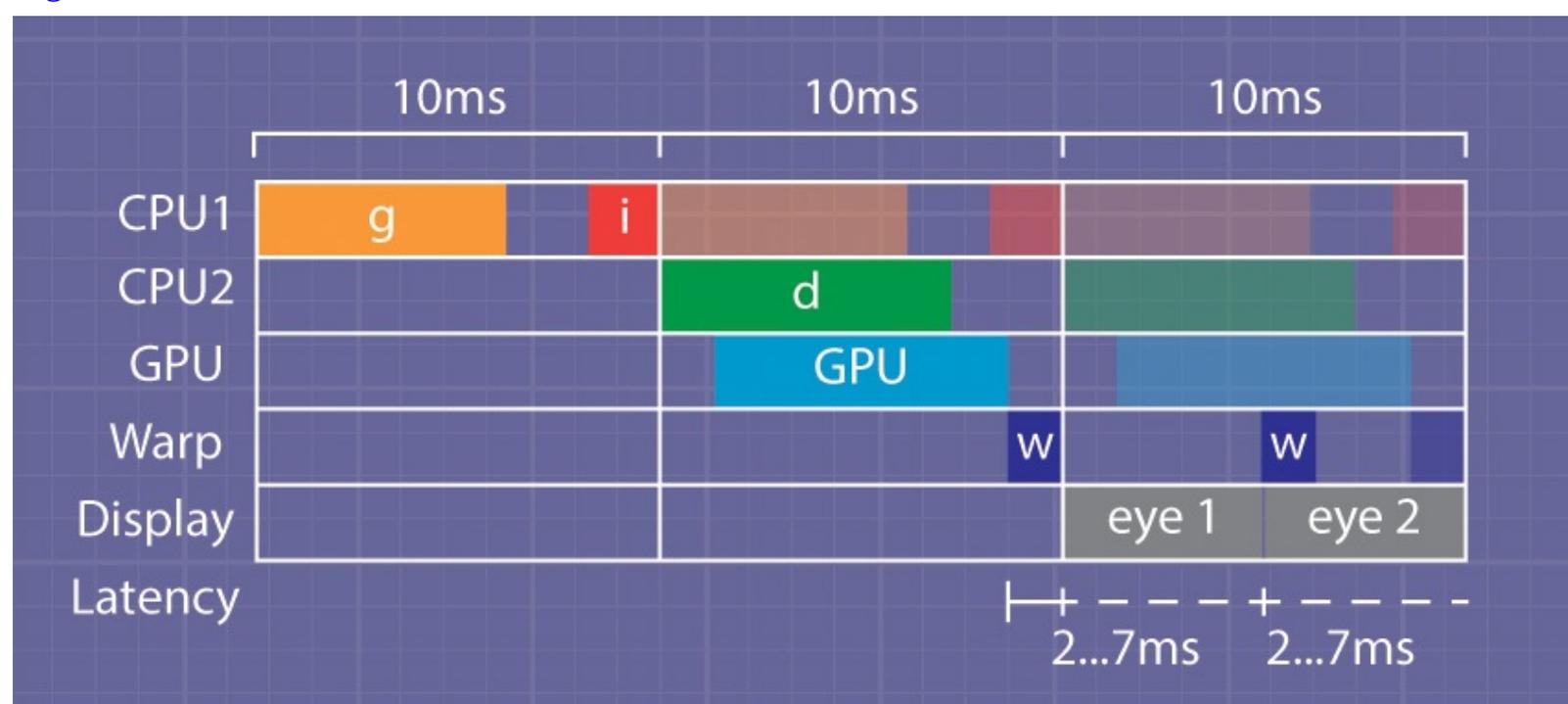


Figure 10.8 Simple rendering pipeline with less than one frame of rotational latency due to timewarp and front buffer rendering. The warp is updated twice per frame. i = input, g = game, d = draw. Each 10ms column represents a single refresh of a 100Hz display.

This front buffer rendering is enabled by default on Gear VR, but it needs to be manually activated on Daydream VR (see [Figure 10.9](#)).

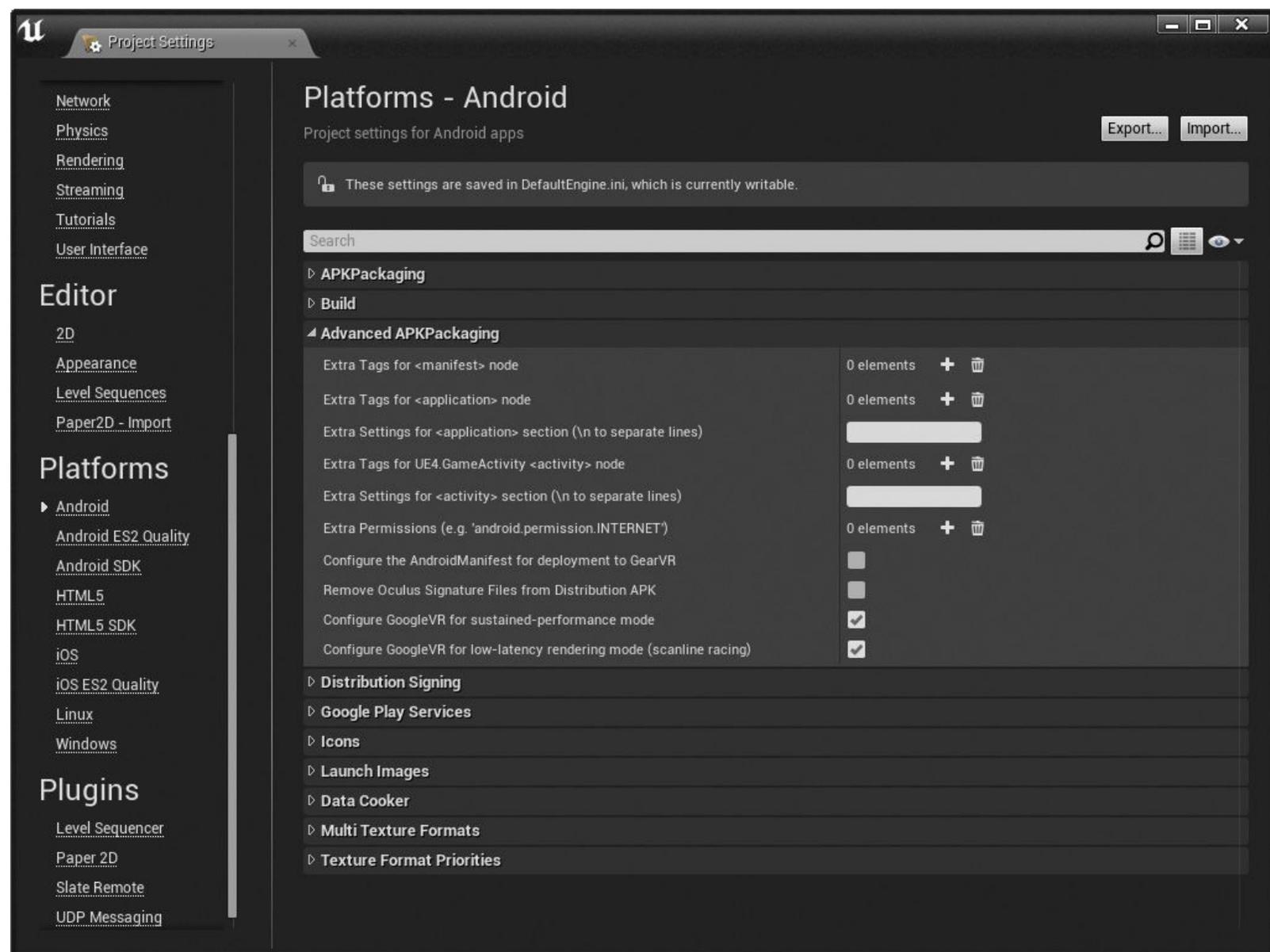


Figure 10.9 Configuration settings for enabling scanline racing on Daydream VR

Performance Improvements

The other pillar of rendering techniques aims to help your VR experiences render in less time or using fewer resources to get the same result.

These optimizations are important because not only do they allow you to spend more time rendering other aspects of your game (such as shadows or fancy Materials) that can make your worlds look more realistic, but they also allow you to run your current experiences on lower-end hardware. This can be very important for mobile VR but also for lowering the barrier to entry for VR, bringing high-quality experiences to more people.

We have looked at how a frame is timed and when is the best time to call rendering options on CPU and GPU to minimize latency; now we will look at how a frame is actually rendered to see what kind of performance savings you may be able to get from simply changing the way your frame renders.

Forward versus Deferred

A traditional game world in its barest form consists of thousands and thousands of individual one-dimensional points in space (known as *vertices*). However, these points on their own aren't conducive to creating a realistic 3D world. To convey to the player the sense of viewing a fully three-dimensional world, these points are first grouped into two-dimensional triangles. Combining these triangles in a 3D environment gives the user an idea of the shape of objects in your world. Although these shapes may convey a limited view of an object's form (through the perspective projections that are usually applied), shading is used to get a true sense of how the object exists in the 3D world. Shading aims to show to the player how light interacts with objects in your game world. Before we get into how this lighting is accomplished in gaming, we need to look at the process that leads up to the lighting stage.

Rendering in games works on what is called a *programmable pipeline*. This means that (as opposed to the fixed pipelines of yesteryear) the graphics APIs allow game developers to control how the vertices and fragments (data that can be used to produce pixels) are processed to produce the final image on the player's screen.

Current graphics APIs allow the chaining of many different shaders before the final pixel values of your game world are shown to your user; however, for simplicity we will focus on the two most important shaders, the vertex shader and the fragment/pixel shader.

In a simple application the vertex shader takes as input the vertices of an individual piece of geometry in your game world and does all the necessary transformations to turn those points into the appropriate form for the fragment shader (such as converting from object space to screen space and applying the perspective projection of the virtual camera).

After this stage, however, there are a few options. The two options that categorize the main rendering techniques applicable to your UE4 applications are forward rendering and deferred shading.

Forward Rendering

In a traditional forward rendering setup, the next stage is quite simple. The fragment shader takes as input the output of the vertex shader (in practice there is an intermediary shader called the geometry shader, but this is left out for simplicity). Using this vertex data, you can then loop over all the potential pixels that will end up being displayed to the player. These potential pixels are called fragments, and the job of this shader is to take these fragments and shade them or rather calculate the final color of the pixels that will be shown on the screen (see [Figure 10.10](#)).

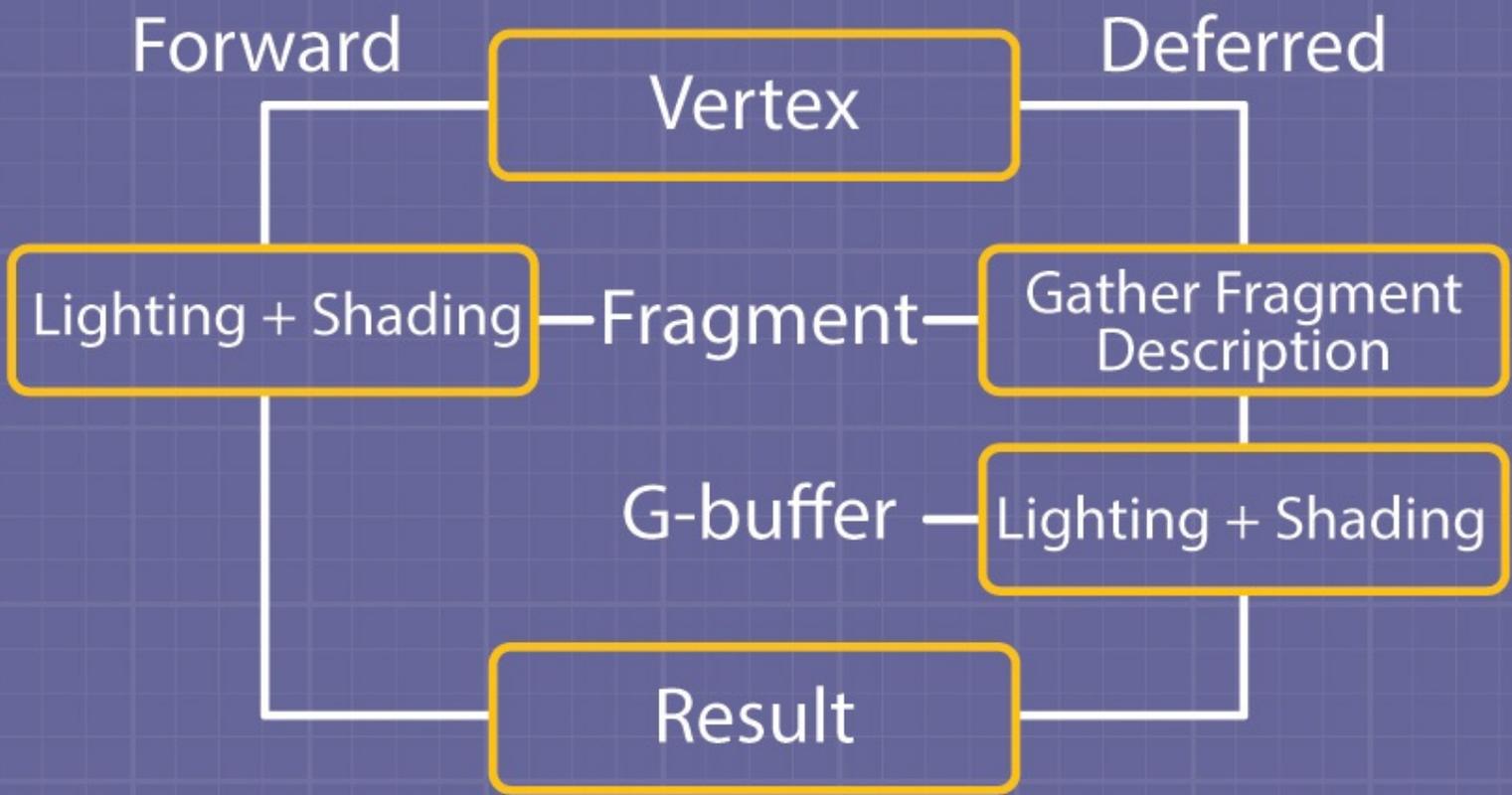


Figure 10.10 Forward rendering versus deferred shading. The forward approach does all lighting in the fragment shader, whereas the deferred approach puts descriptions of each fragment into buffers that can be used to light the scene at a later time.

Many techniques can be used to actually shade these fragments; however, as a base you will need a few things. First, you can use the vertex data to calculate the outward direction of the triangle to which a fragment belongs (referred to as the normal). This allows you to then loop over all the lights in your scene and compare the outward angle of your triangle to the direction of the light source. You can thus calculate how intensely a light will affect your particular surface. You can then use the vertex data once more to find the world space position of your fragment and use this to scale how much any light affects it by how far away that light is.

Note

The process of projecting the vertices that exist in your world to the screen to be shaded is called *rasterization* and will come up again in a little while.

Those familiar with forward rendering will notice that there are a few nonoptimal things about this approach. First, you have to loop over all the lights in your scene for every fragment that is shaded, even if that fragment is hardly affected (if at all) by a particular light. This can be partially avoided by checking the radius of influence (attenuation) of each light to see whether it reaches the fragment and if it doesn't, skipping over that light; however, as we will discuss later, there may be better ways to handle this. Another issue with this basic forward rendering is that because of the way triangles are rendered in your scene, you may fully light and rasterize a piece of geometry only to have it completely covered by another piece of geometry later in your rendering, hence wasting GPU time on something that the user will not see. This phenomenon is known as *overdraw*, and because of this inefficiency the performance of basic forward rendering is limited by the amount of geometry in your world multiplied by the number of

dynamic lights.

Note

Overdraw can be pretty much eliminated with an Early Z-Pass, which renders a depth buffer that can be tested against to cull pixels that will be hidden in the final image (before any lighting is done on them).

However, this comes with the cost of needing to rasterize your scene twice, which depending on your use case may lead to worse performance.

As the quest for realism in games continued, this limit on the number of dynamic lights began to become a bottleneck, and in the mid-2000s a deferred approach to shading became popular because it removed the dependency between the number of lights and the geometric complexity of a game world.

Deferred Shading

As opposed to forward rendering, deferred shading takes the fragment shader from the forward example and instead of outputting the final pixel to be displayed, it outputs a description of the fragment into render targets or buffers which will then be combined later to shade, light, and output the final pixel information (hence deferring the lighting/shading step). See [Figure 10.10](#). These buffers are known as *geometric buffers* (G-buffers); an example of a G-buffer produced by UE4 can be seen in [Figure 10.11](#).

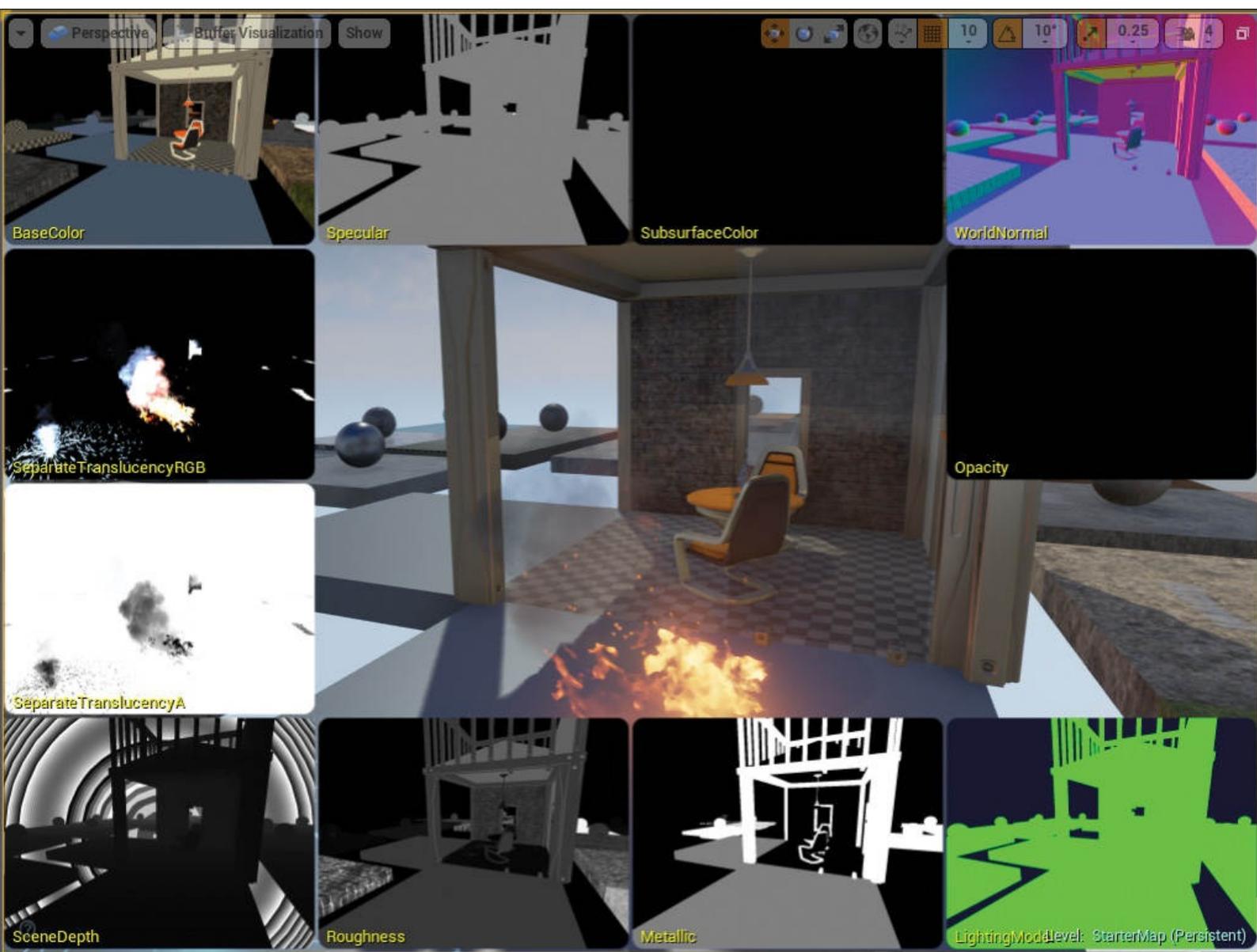


Figure 10.11 Sample G-buffer from an example Unreal Engine level

Deferred rendering isn't without its problems, however, though it may eliminate overdraw for the usually expensive lighting operations (by producing a depth map of the scene before any lighting is done, allowing the lighting calculations to be done only on the nearest fragments). The depth buffer that allows this also makes the rendering of transparent objects hard. Because the depth buffer can represent only a single depth value, objects that may need multiple depths to calculate lighting (such as semitransparent objects) cannot be represented easily in deferred rendering.

To get around this, many deferred rendering implementations may need to fall back to forward rendering for transparent objects, meaning you will need to rasterize your geometry twice, once for opaque objects and a second time for transparent objects. (It is out of scope for this book, but UE4 actually creates a spherical harmonic for each transparent object that allows it to approximate the scene lighting and compose these translucent objects into the scene with post processing.)

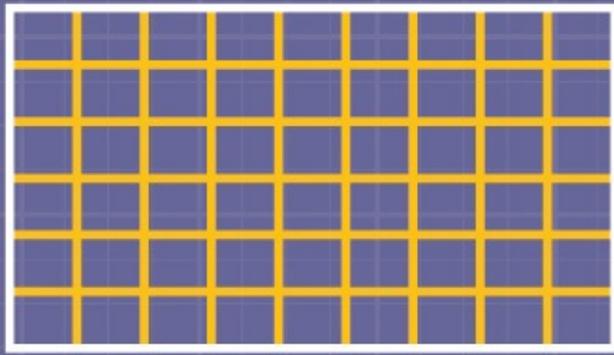
Which Approach to Choose: Forward or Deferred?

For desktop applications, UE4 natively implements a deferred renderer because the G-buffer allows you to do a lot of fancy rendering tricks that can make your games look very realistic (such as screen space reflection [SSR], screen space subsurface scattering [SSSS], screen space ambient occlusion [SSAO], and many others). However, these effects can be computationally expensive, and because the rendering requirements of VR experiences are so high, in most cases these effects cannot be used. Also, these effects may not be the best fit for VR because they are calculated in screen space, which due to the stereo nature of VR may create stereo disparities and cause discomfort to your users.

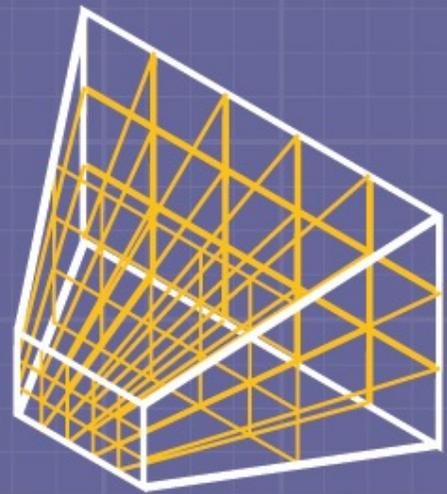
Forward rendering also allows easier implementation of hardware-based MSAA (multi-sample anti-aliasing) compared to the post-process TAA (temporal anti-aliasing) that UE4 implements by default. MSAA can be a better fit for VR because it can lead to a sharper image—it affects only the edges of geometry, meaning that a lower screen percentage can be used for the same image sharpness, overall reducing the render cost.

In some cases, however, the limits of standard forward rendering may be a pain point for developers (mainly the inefficiencies of dealing with many dynamic lights). To counteract this, there are many techniques for both forward and deferred that aim to efficiently handle millions of dynamic lights. The two leading techniques are tiled rendering and clustered rendering. It is also worth noting that both techniques can be implemented with forward rendering or deferred shading. (In fact, if there are enough lights in your scene, UE4 will move to a tiled-deferred render instead of its traditional deferred.)

The basic idea behind a tiled approach to forward rendering is to limit the number of lights over which each shader fragment must loop to calculate its lighting. This is done by splitting the view into many different tiles and finding which lights affect each specific tile. Then, when it comes time to render, your fragment needs to loop over only a subset of the total lights in your scene. However, there is still one potential inefficiency with this approach, mainly that if a single tile contains two objects that are affected by very different lights (which usually happens with objects that are far away from each other), there is no way of separating them because the objects are grouped in two dimensions. To solve this, there is clustered rendering, which not only splits the view into screen space tiles but also divides them into clusters in the direction of the camera frustum (see [Figure 10.12](#)).



Tiled



Clustered

Figure 10.12 Tiled versus clustered rendering

At the time of writing, UE4 implements a deferred renderer on the desktop whereas a forward renderer must be used on mobile (because of the pain points of handling the needed render targets for deferred rendering on mobile). However, because of the stringent rendering requirements of VR today and the need for high-quality but cheap anti-aliasing, forward rendering may very well be a better option for the current state of VR games. Epic is planning to introduce a forward renderer for desktop in the UE4 4.14–4.15 timeline which will potentially allow VR developers to provide higher-quality experiences for their users. Note that in the current implementation many of the rendering features of UE4 (such as SSR and SSAO) are not available when using forward rendering because the G-buffer that is required to use such features is not created. That being said, however, there may be other benefits from lighting being calculated on a geometric level (such as planar reflections on a per-object level) that outweigh this disadvantage.

Instanced Stereo

As shown in [Figures 10.2](#) through [10.8](#), the draw thread represented in green is where the engine submits the draw calls to the GPU and tells it what to render. Unlike a normal monitor in VR, you have to render both eyes separately; in a naïve approach you can simply render both eyes completely independently of each other. However, if you're smart about it, you can draw both eyes at the same time and cut down on work done on the CPU. In fact, in Unreal Engine 4.11 Epic introduced a feature called Instanced Stereo Rendering that does exactly that. It is not enabled by default, because it doesn't work with all rendering features currently, but you can enable it in the Project Settings (see [Figure 10.13](#)).

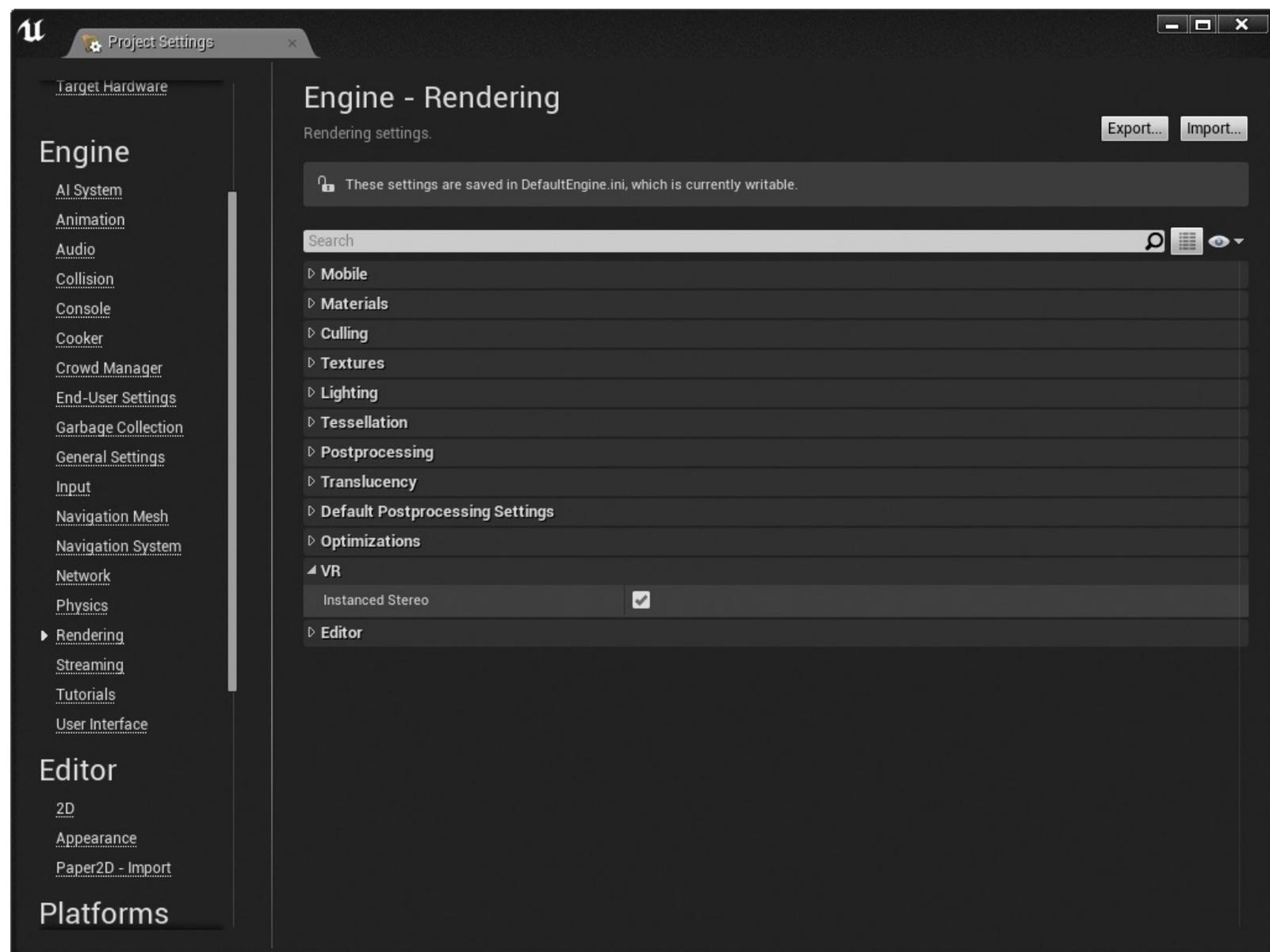


Figure 10.13 Instanced Stereo Rendering Project Settings

Hidden Area Mesh Optimization

Because the lenses in VR headsets have a circular field of view, there can be some waste in rendering to the entire screen of the display if the user will never see it. Therefore, UE4 (since version 4.10) implements a hidden area mesh that automatically culls the parts that will never be seen. This technique also ensures that post-processing effects are processed only on the parts that the user will see. This feature is enabled by default.

VR Project Settings

When creating a new project, UE4 by default throws everything it's got at you, meaning that performance may suffer from the number of features that the engine has.

To thwart this, there are many options that developers can toggle as well as a few presets that can help you meet the hearty requirements of rendering a VR experience.

First, there are the Target Hardware and Graphics Performance options presented in the project creation dialog (see [Figure 10.14](#)) and in the Project Settings (see [Figure 10.15](#)).

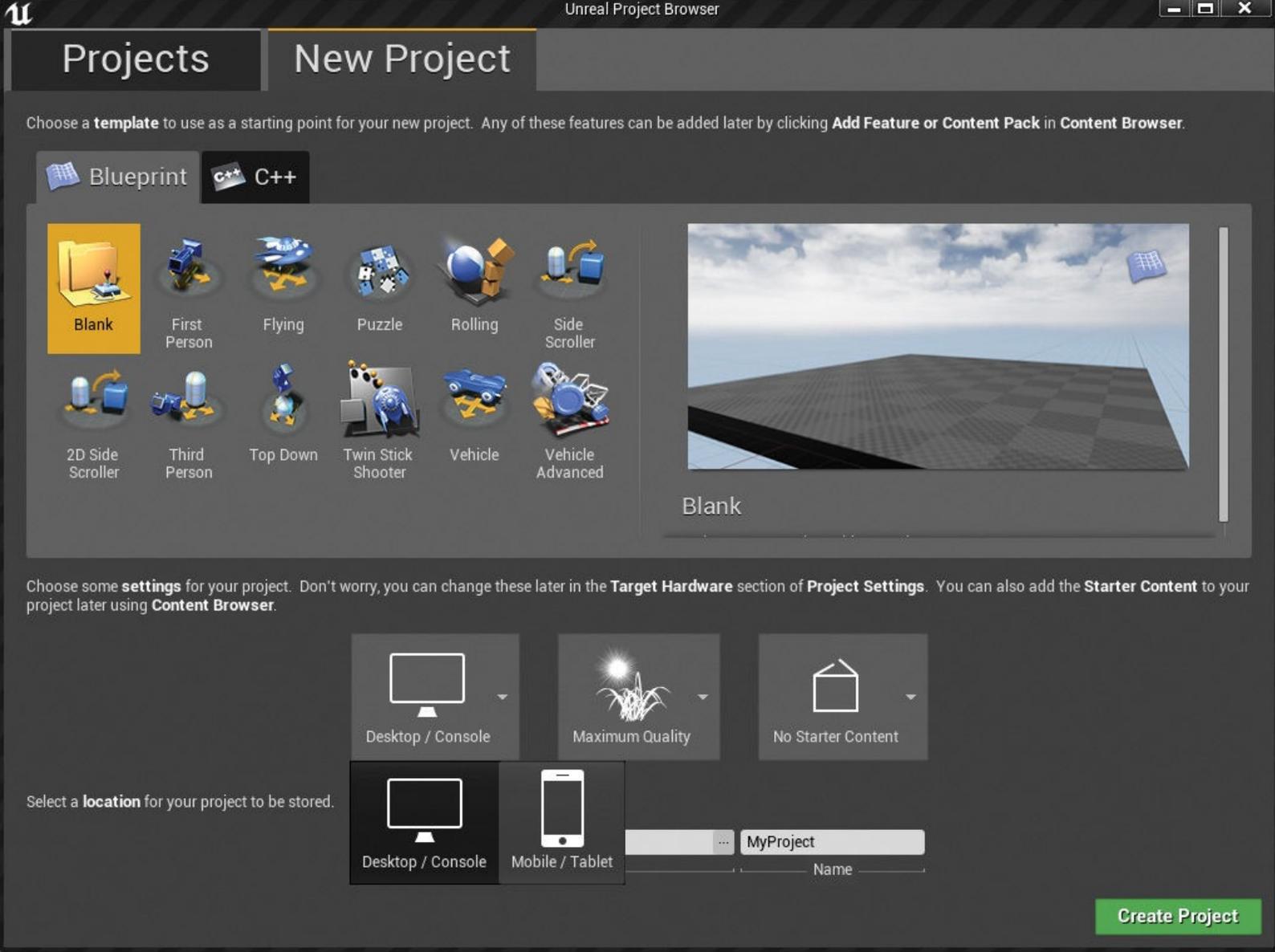


Figure 10.14 Target Hardware and Graphics Performance options in the project creation dialog

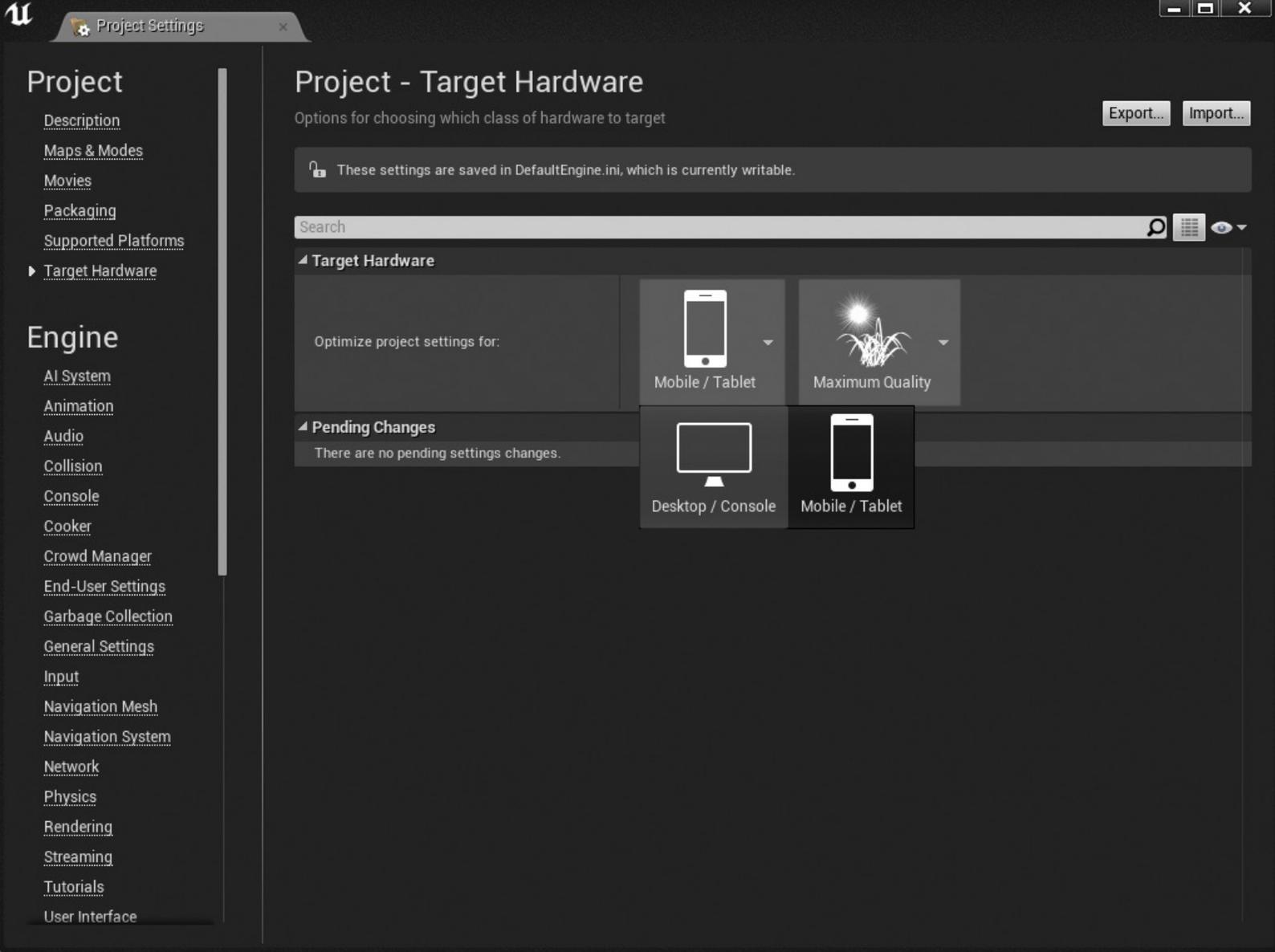


Figure 10.15 Target Hardware and Graphics Performance options in the Project Settings

To see the differences among the settings, refer to [Figure 10.16](#), which shows Target Hardware options with their respective graphics quality. As you can see, setting the graphics quality to Scalable removes Auto Exposure and Motion Blur from the default post-processing effects. It also disables Anti-Aliasing while the target hardware is set to Desktop. However, setting the graphics quality to Scalable when the Target Hardware is Mobile simply disables Bloom and Mobile HDR; Auto Exposure and Motion Blur have already been disabled by setting the target device to Mobile.

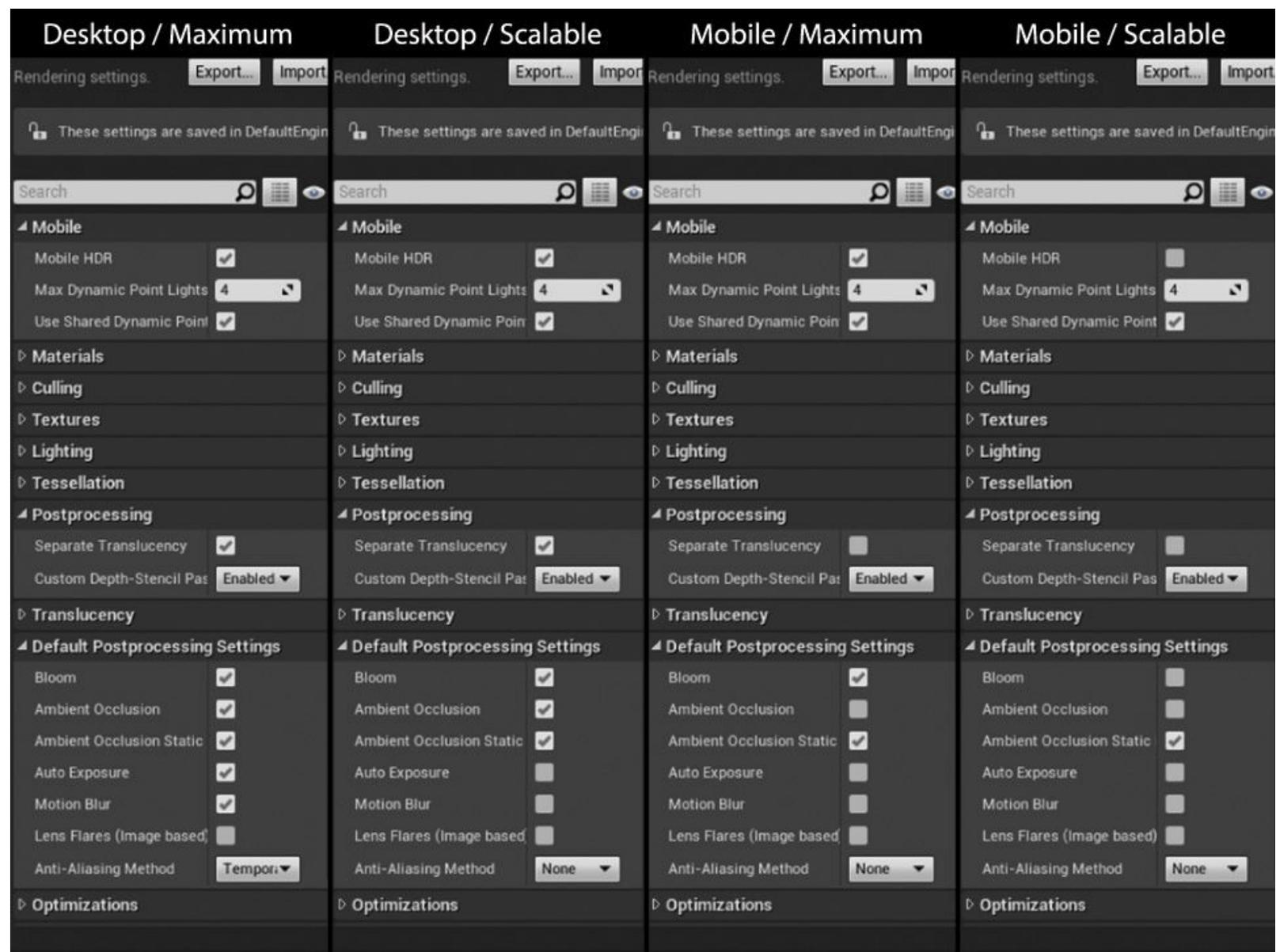


Figure 10.16 Rendering settings for target hardware and graphics

Because Motion Blur, Auto Exposure, Bloom, and Ambient Occlusion are all effects that are generally not needed in VR games, you can generally gain performance by turning them off. Selecting the Mobile Scalable settings during project creation can be a fast way to do so.

Notice that going to the Mobile Target Hardware disables Separate Translucency, which will result in a gain in performance but may lead to some artifacts on overlapping transparent surfaces (see [Figure 10.17](#)). I recommend disabling it to gain performance and enabling it only if you run into problems.

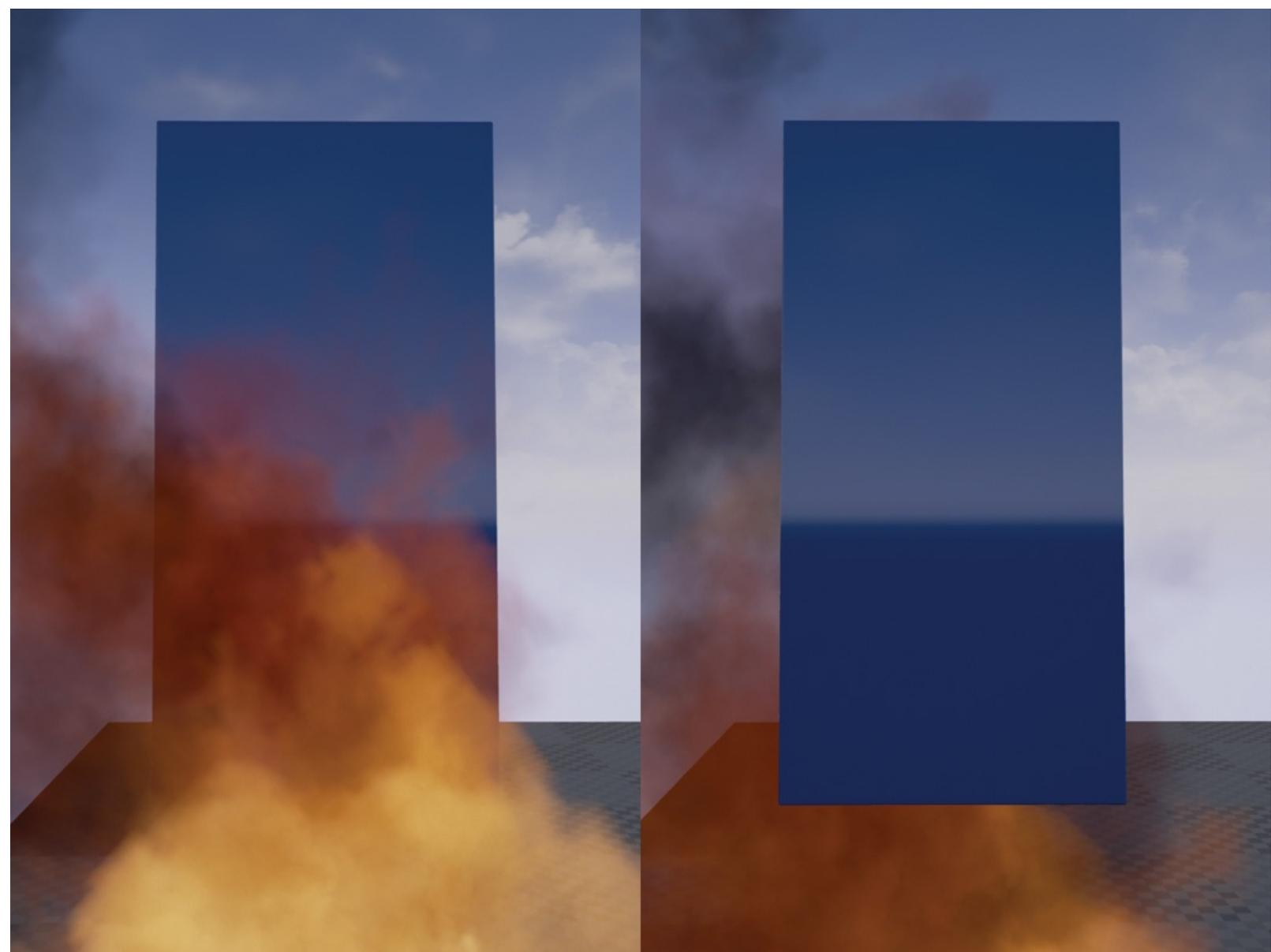


Figure 10.17 Two separate translucent objects overlapping each other: left, Separate Translucency on; right, Separate Translucency off

While choosing which settings to turn down to meet your performance requirements, you may have come across the Engine Scalability Settings (Settings → Engine Scalability Settings); see [Figure 10.18](#). They are in fact just that, engine settings, and do not affect any project that you package outside of the engine.

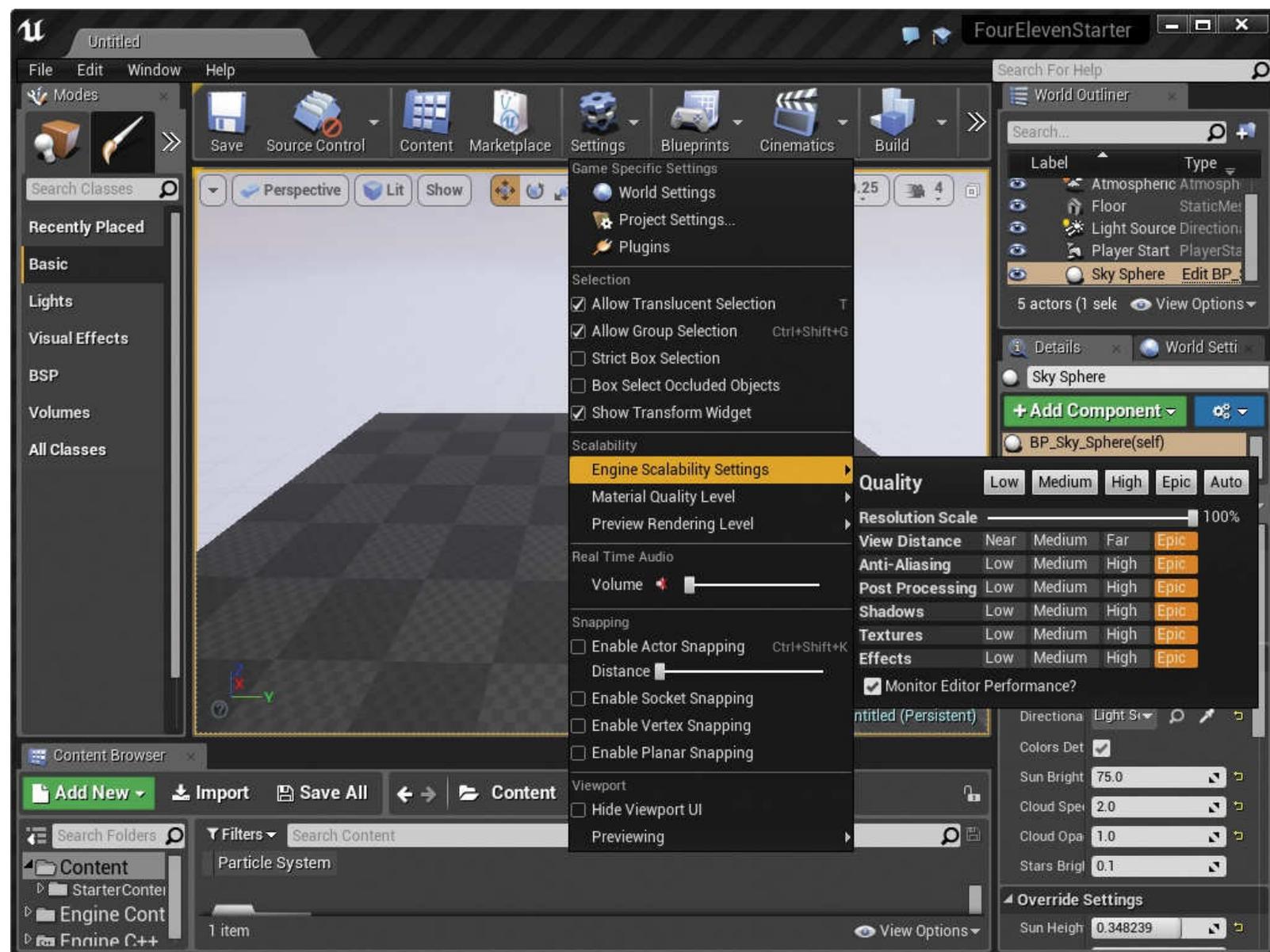


Figure 10.18 Engine Scalability Settings

To change scalability settings and have them affect your final packaged game, place all of your settings in the `DefaultEngine.ini` (`<ProjectName>/Config/DefaultEngine.ini`) in the form of console commands under the `[SystemSettings]` banner.

As an example, the console command to change the screen percentage is `r.screenpercentage=x` where `x` is a value from 30 to 300 of the screen percentage.

In the `DefaultEngine.ini`, it would look something like this for a screen percentage of 130:

```
[SystemSettings]
r.screenpercentage=130
```

With this, you can change the default value for nearly all engine features (such as anti-aliasing, `r.PostProcessAAQuality=x`) or even disable features like HZB occlusion (`r.HZBOcclusion=0`), which has a high up-front cost but scales better than the default occlusion.

Knowing how to turn engine features off is all well and good, but how do you figure out what to turn off? Before you answer that question, you need to find out whether you are CPU bound or GPU bound—is it your GPU that is causing the rendering to be too slow or is it the CPU?

To answer that, use the console command `stat unit`, which shows you frame times for your total frame as well as the game, and draw (CPU) threads, and finally the GPU (see [Figure 10.19](#)). Basically, the

highest of the three (game, draw, and GPU) determines whether you're CPU bound (by either your gameplay logic with the game thread or your draw calls with the draw thread) or GPU bound.

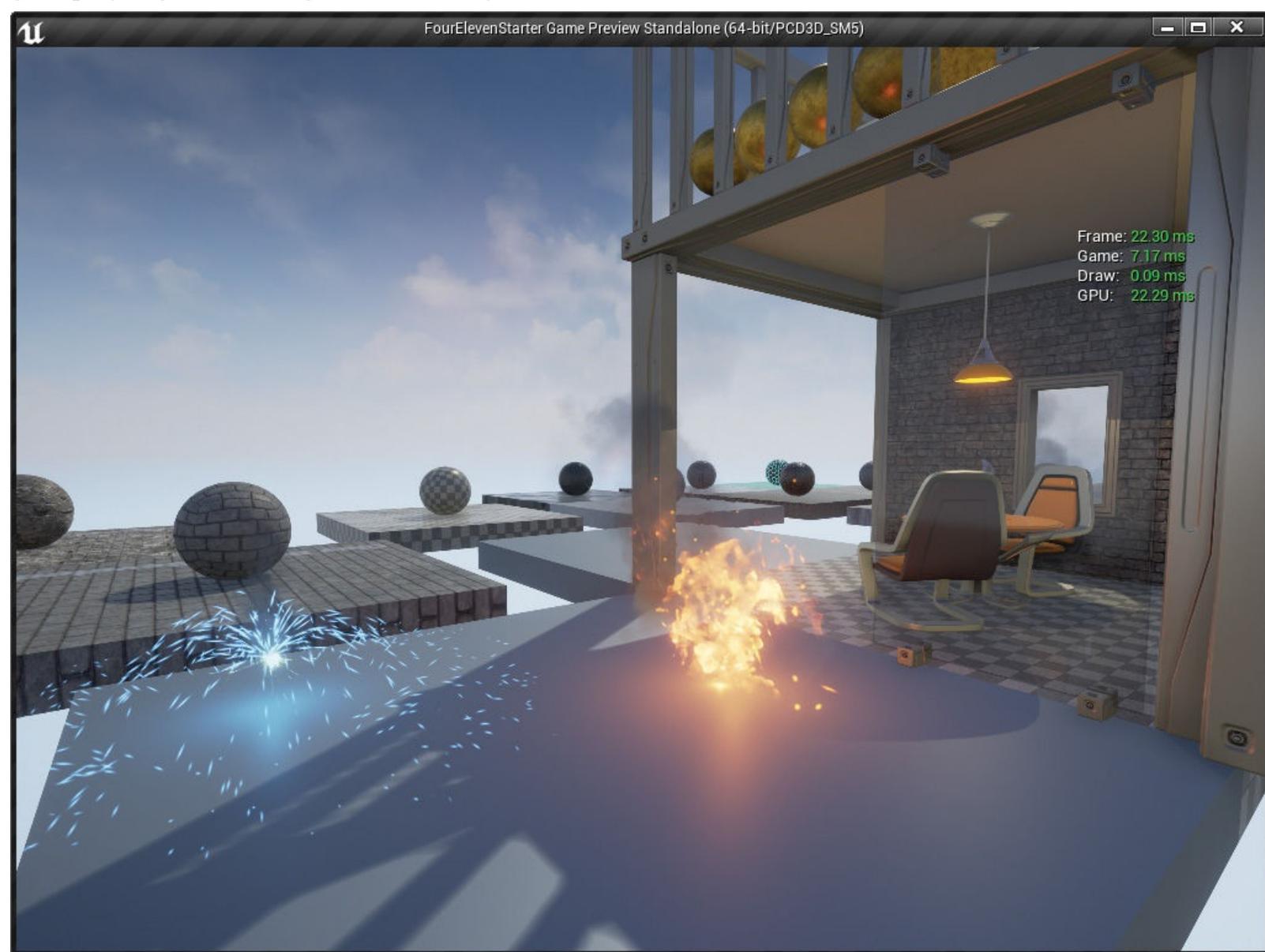


Figure 10.19 Stat Unit timings

Once you know which you are bound by, you can then profile using UE4's GPU or CPU profiling tools. If you are GPU bound (as in [Figure 10.19](#)), you can use the GPU Visualizer (see [Figure 10.20](#)) by either using the console command `ProfileGPU` or the shortcut `Ctrl + Shift + ,`. Once the Visualizer is open, you can find the render times for your entire frame and can find out exactly what is taking longer than it should.

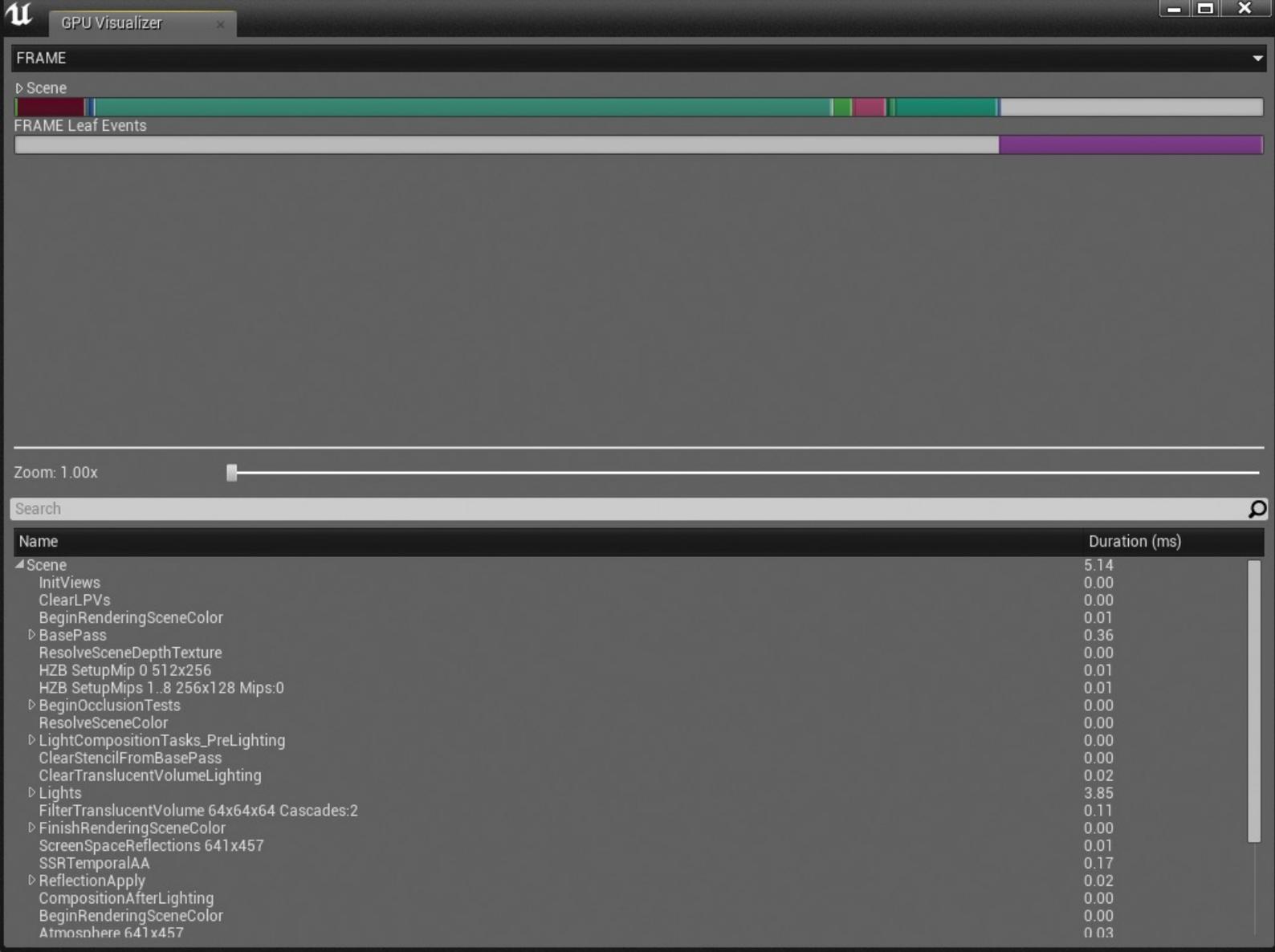


Figure 10.20 GPU Visualizer

If you are CPU bound by the draw thread, you are most likely hitting a draw call limit and may have to limit how many objects are in your scene or combine some objects into one mesh. To check your draw calls, use the console command `Stat SceneRendering`.

If you are CPU bound by the game thread, look at what may be causing the issue by using the `Stat Game` console command.

Summary

This chapter looked at why rendering a VR scene is a challenge because of the demanding specs of current VR HMDs. You looked at some clever optimizations that UE4 and others use to make sure your experiences are rendered at the highest fidelity and lowest latency.

You looked at some of the basic project settings that UE4 lets you easily change, whether it's when the project is created or later with deeper, more specific settings. Finally, you looked at ways to determine what settings may be causing you grief when trying to hit your performance targets.

Exercises

To hone the skills you learned in this chapter, optimize your own games/experiences or deconstruct the sample games Epic provides with UE4.

Part III: Appendices

Appendix A. VR Editor

With the VR Editor, you can see how your games/experiences will look in VR as you create them. Using motion controllers, you are able to richly interact with objects in your world, translating, scaling, and rotating them to create compelling visual experiences. This appendix introduces you to using this editor.

Enabling the VR Editor

The VR Editor is currently an experimental feature (as of 4.13) and must be enabled in the Editor Preferences before use.

To enable the editor, open the Editor Preferences (Edit → Editor Preferences). In the Experimental tab, check the Enable VR Editing option (see [Figure A.1](#)). Underneath this option, notice the Enable VR Mode Auto-Entry option; this automatically enters the VR Editor when the proximity sensor is activated on headsets that support it.

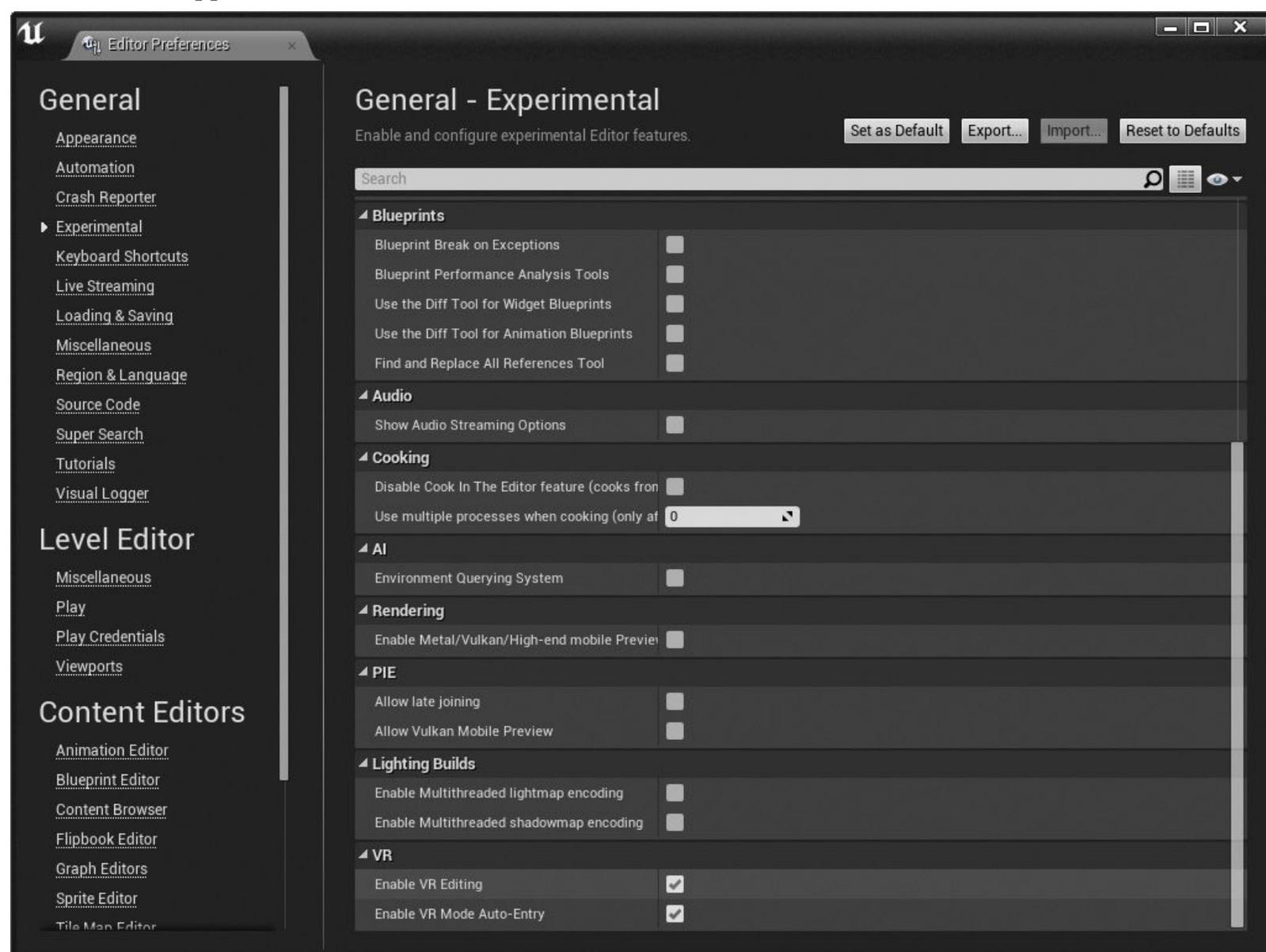


Figure A.1 Enabling the VR Editor in the Editor Preferences

To actually open the VR Editor, close the Editor Preferences and, in the shelf, click the now-visible VR button (see [Figure A.2](#)).

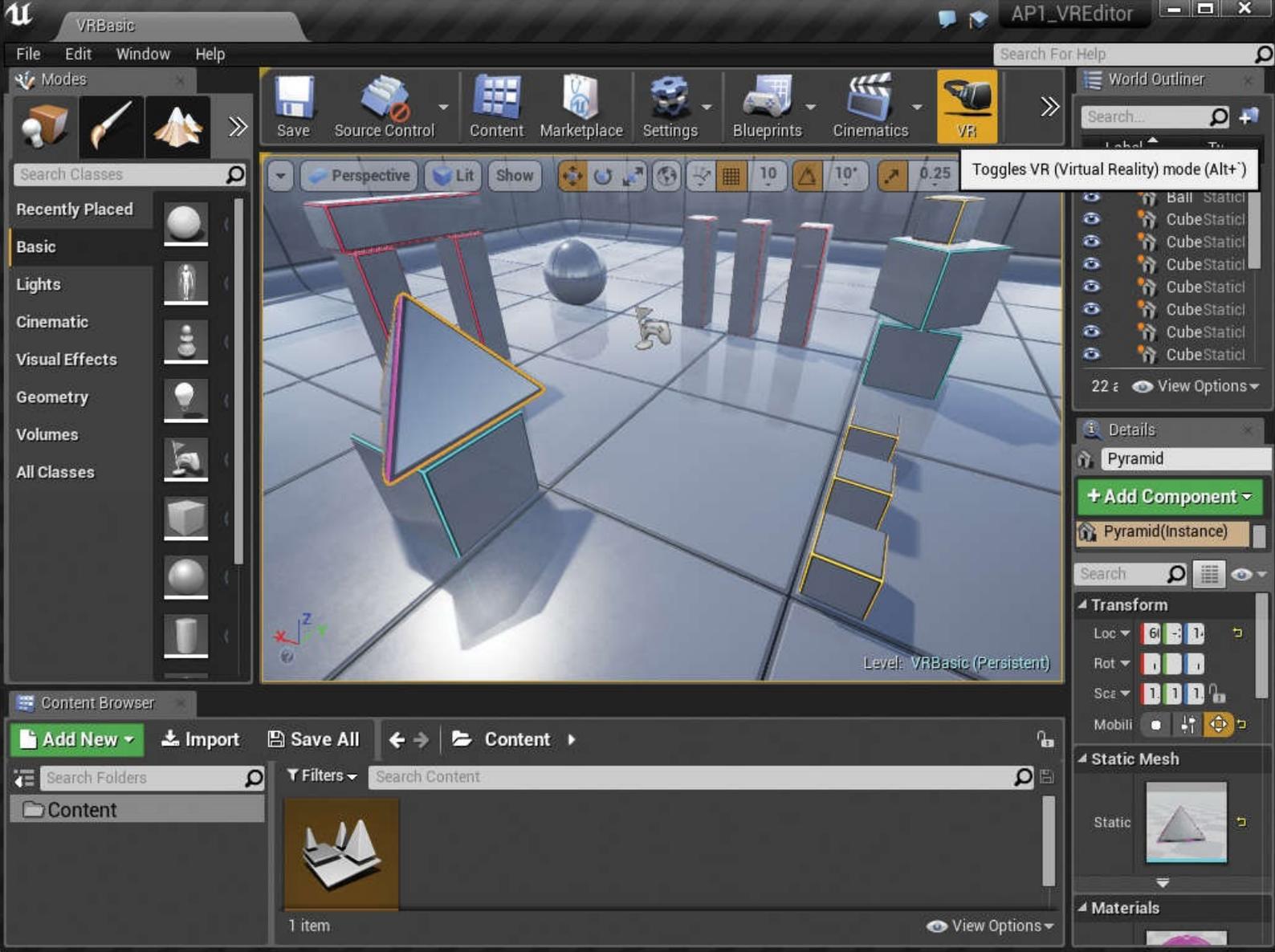


Figure A.2 Toggling the VR Editor with the new shelf button

Controlling the VR Editor

The VR Editor allows you to do most of the actions that can be performed in the traditional 2D editor. But in VR, a few controls need to be learned.

The VR Editor makes full use of motion controllers and won't work to its full potential without them. Currently, the editor supports both the Oculus Touch and HTC Vive controllers and makes use of the trigger (to select objects), the grip/side button (to navigate around the world), the touchpad/joystick (to activate the Radial Menu and to scale selected objects), and the menu/modifier button (to perform auxiliary actions). These mappings can be seen by placing the controller upright and looking at it with the VR Editor active (see [Figure A.3](#)).

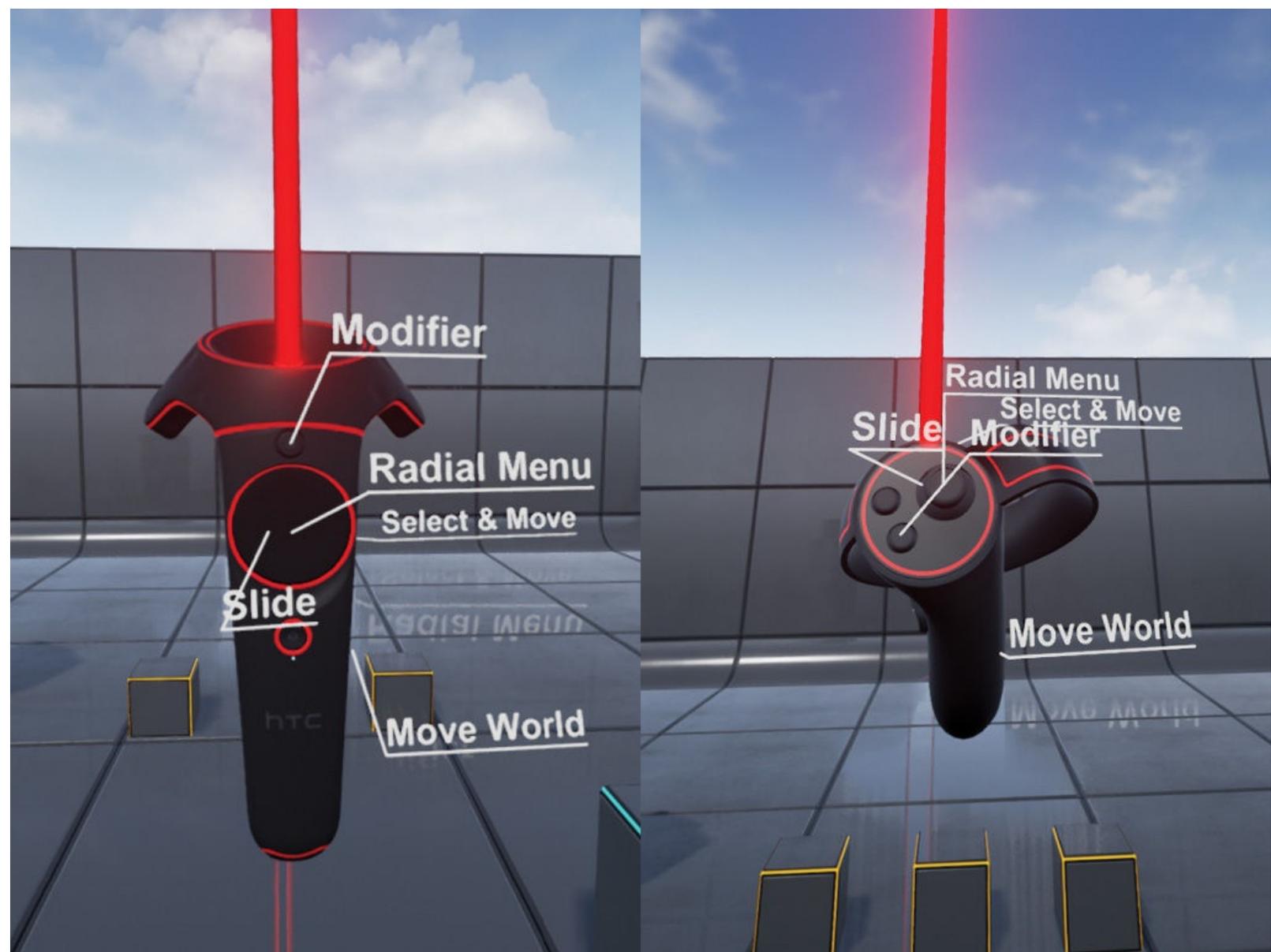


Figure A.3 VR Editor controller mappings

Navigating the World

To navigate the world inside the VR Editor, use the Move World button (usually mapped to the side button). Pressing this button on a single controller and moving it makes it seem as if you are pulling yourself through the world, allowing you to position yourself anywhere within it (see [Figure A.4](#)). Pushing this button on both controllers allows you to scale the world down and up (by moving the controllers toward and away from each other; see [Figure A.4](#)) and rotate the world in the yaw axis (by rotating the controllers around each other).

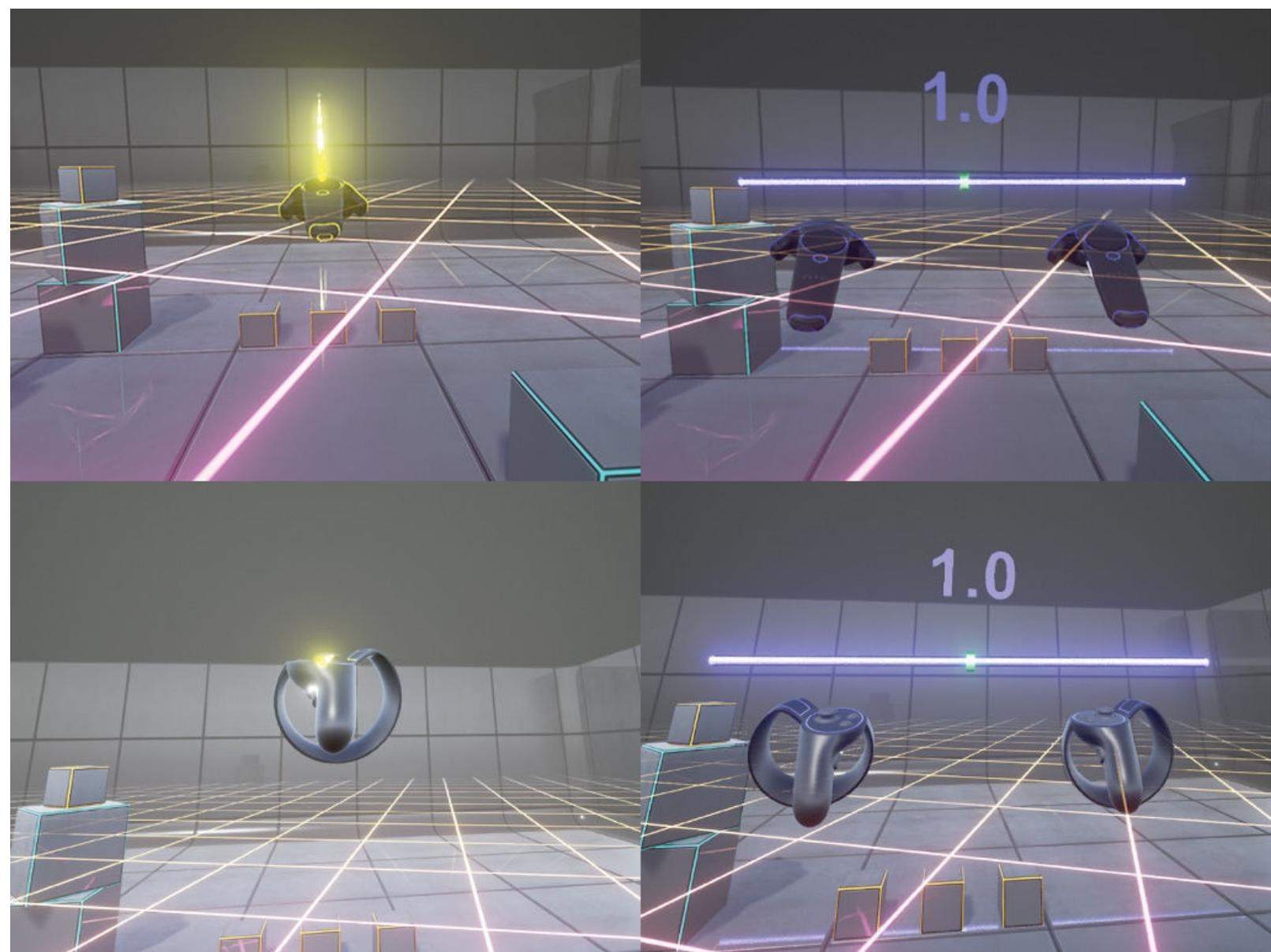


Figure A.4 VR Editor navigating the world

Interacting with Objects

To interact with objects when inside the VR Editor, use the Select and Move button (usually mapped to the trigger). Half-pressing this button on a controller selects the object that is intersecting with that controller's laser pointer; to select multiple objects, continue half-pressing this button on one controller while the other controller selects other objects. Selecting an object in this way shows the object Gizmo (see [Figure A.5](#)) and allows you to translate, scale, and rotate the selected object(s) by grabbing the appropriate Gizmo controls. Another way to transform objects in the VR Editor is to enter Freeform mode by fully pressing the trigger down on a controller (see [Figure A.5](#)), where the selected object enters Freeform mode. In this mode, you can translate the object by moving the motion controller or push it toward/away from you by sliding on the touchpad or joystick. Once in Freeform mode, you can also select the object with a second controller, then scale the object by moving the controllers toward/away from each other or rotate the object by rotating the controllers.

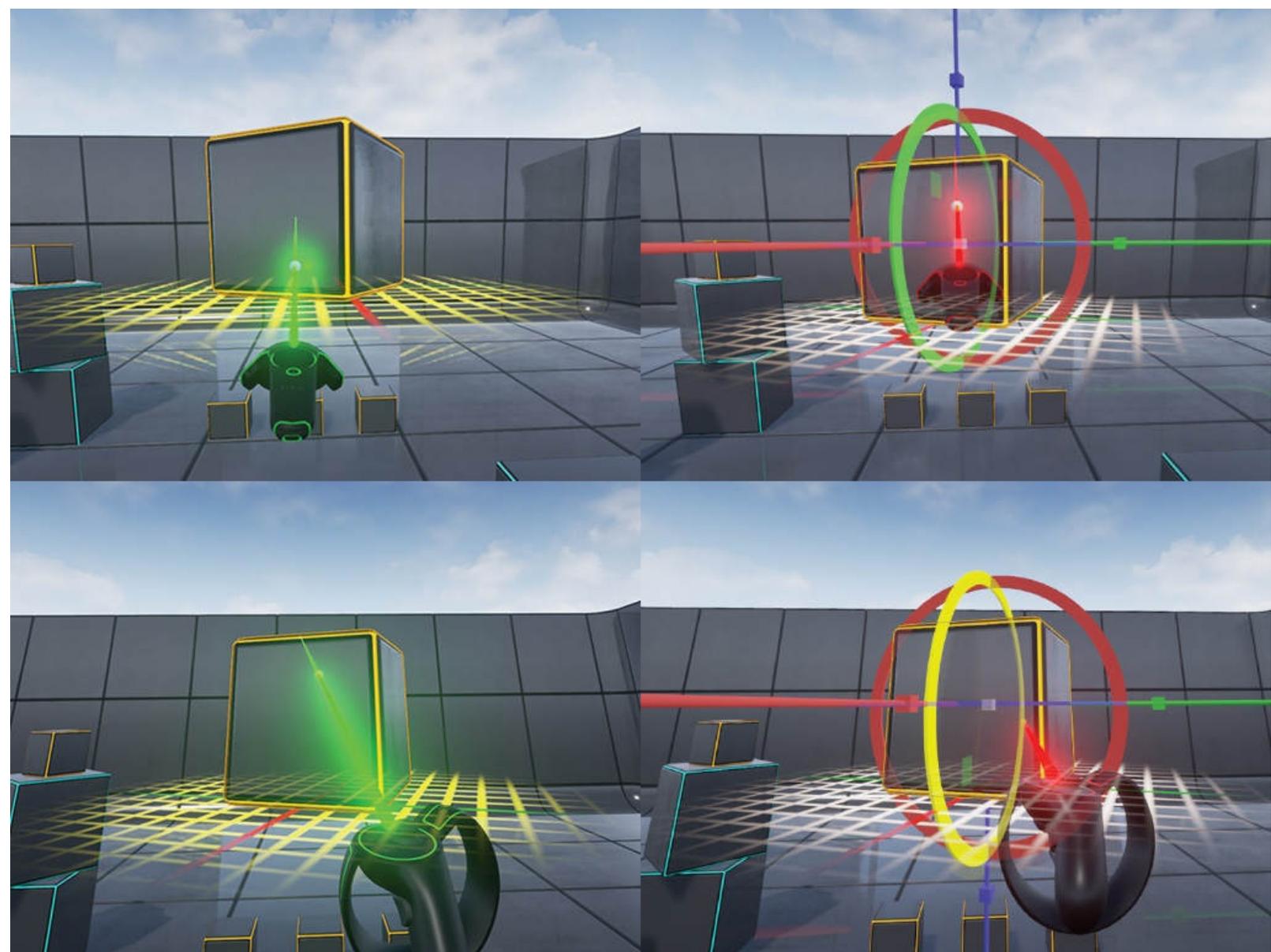


Figure A.5 VR Editor interacting with objects

Interacting with Menus

The VR Editor currently has two types of menus: the Radial Menu and the Quick Select Menu. The Radial Menu can be accessed by sliding your finger on the touchpad or moving the joystick and contains frequently used shortcuts like Undo/Redo and Copy/Paste (see [Figure A.6](#)). The Quick Select Menu can be accessed by pointing one motion controller at the other. It contains four sections (Grid Snap, Interface, Gizmo, and Play) that allow you to change snapping options, access 2D interface elements from the traditional editor, control the Gizmo, and play the game (see [Figure A.6](#)).

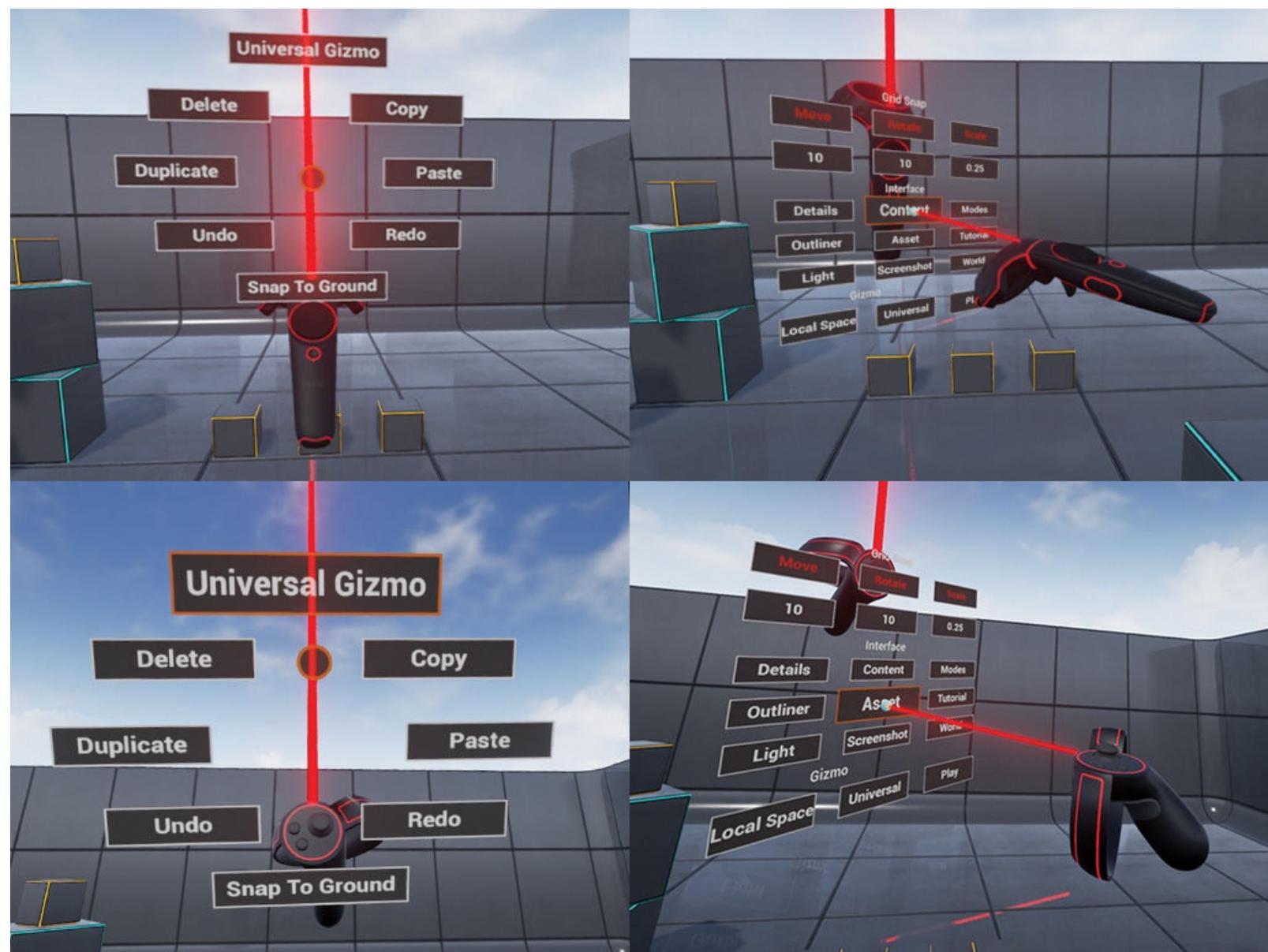


Figure A.6 VR Editor accessing the Radial Menu (left) and the Quick Select Menu (right)

When selecting an option from the Interface section, the selected 2D UI element docks to the end of your controller (see [Figure A.7](#)) and allows you to interact with that element with the other controller through the laser pointer. To undock the UI element, select the gray rectangle underneath the element with the trigger and move it away from its docked position (see [Figure A.8](#)). Once an element is undocked, it remains in that position and allows you to grab more elements from the Quick Select Menu. To quickly access common UI elements, press down in the middle of the touchpad or joystick to toggle key windows.



Figure A.8 VR Editor undocking the UI elements

Summary

This appendix looked at the experimental VR Editor, which allows you to create and edit your VR worlds in VR. You should now have all the necessary knowledge to start learning this new interaction paradigm and start designing and creating VR content in VR.

Appendix B. Resources

Looking for even more on VR? This appendix helps you find the right communities and resources that can assist you on your journey to create the perfect VR experience.

Epic

Epic has a good amount of resources to help with overcoming problems and expanding your knowledge when it comes to UE4 and VR. The documentation helps you with any engine features that are not covered in this book, and the AnswerHub and Forum allow you to ask any questions you might have or simply participate in the growing VR community. Epic also runs a weekly live training stream on Twitch/YouTube that often covers topics on VR.

- **Documentation:** <https://docs.unrealengine.com/latest/INT/Platforms/VR/index.html>
- **AnswerHub:** <https://answers.unrealengine.com>
- **Forum:** <https://forums.unrealengine.com/forumdisplay.php?27-VR-Development>
- **YouTube:** www.youtube.com/user/UnrealDevelopmentKit
- **VR Live Training:** www.youtube.com/playlist?list=PLZlv_N0_O1gY7G589Z3I5-Dz7AdFSIWaG

Oculus

Oculus provides a wide variety of documentation for its platform and headsets. Some of the details may be redundant, because a lot of developers rely on Epic's implementation of the various SDKs. However, the provided Best Practices Guide explores a lot of important concepts for VR in general. Oculus also hosts a yearly developer conference (Oculus Connect) that includes presentations from VR developers. These talks are hosted online after the conference (found on Oculus's YouTube page).

- **Documentation:** <https://developer3.oculus.com/documentation/>
- **Best Practices:** https://developer3.oculus.com/documentation/intro-vr/latest/concepts/bp_intro/
- **Forums:** <https://forums.oculus.com/community/discussions>
- **YouTube:** www.youtube.com/user/oculusvr
- **Oculus Connect 2014:** www.youtube.com/playlist?list=PLL2xVXGs1SP5CVE6X3GMSroO2cfHxzeCz
- **Oculus Connect 2015:** www.youtube.com/playlist?list=PLL2xVXGs1SP4kxSKQsG40PCsItIujwznZ

Valve

Valve hasn't produced any best practices guides yet, but it does have a Steam community for Steam VR developers, which is frequently updated with blog posts. Valve also has an almost-yearly conference (Steam Dev Days) that allows developers to attend talks and discuss all things game development with other developers.

- **Documentation:** <http://steamcommunity.com/steamvr>
- **Forums:** <https://steamcommunity.com/app/358720/discussions/>
- **YouTube:** www.youtube.com/user/SteamworksDev
- **Steam Dev Days 2014:** www.youtube.com/playlist?list=PLckFgM6dUP2hc4iy-

Google

Google has a few pages that document its various VR SDKs and a YouTube channel that fairly regularly contains information and tips on developing for Daydream/Cardboard among its various other platforms. Google also hosts a yearly developer conference (Google I/O) that features some talks concerning VR, which are uploaded to YouTube.

- **Documentation:** <https://developers.google.com/vr/unreal/>
- **YouTube:** www.youtube.com/user/GoogleDevelopers
- **Google I/O 2016:** www.youtube.com/playlist?list=PLOU2XLYxmsILe6_eGvDN3GyiodoV3qNSC

Community

Sometimes the officially supported channels may not have all the information you need. In that case, VR developer communities may be able to answer your questions:

- **Oculus Developer Reddit:** www.reddit.com/r/oculusdev
- **General UE4 Developer Reddit:** www.reddit.com/r/UE4Devs/

Physical Meetups

Whether it's just for networking or for learning new topics from guest speakers, there are various physical meetups for Unreal Engine or VR in general:

- **UE4 Meetups:** www.meetup.com/pro/UnrealEngine/
- **SVVR (Silicon Valley Virtual Reality):** <http://svvr.com>

Conferences

If you want a more formal meetup of developers, many conferences are held each year that appeal to VR developers:

- **GDC (VRDC) (Game Developers Conference):** www.gdconf.com
- **Oculus Connect:** www.oculusconnect.com
- **Steam Dev Days:** www.steamdevdays.com
- **Google I/O:** <https://events.google.com/io2016/>

Index

Numbers

- 2D texture splash screen, [46](#)
- 2D UIs (user interfaces)
 - custom menu interaction, [113–122](#)
 - issues with, [104](#)
 - readability, [104](#)
- 2D UIs (user interfaces), creating a basic VR menu
 - menu Actor, [108–111](#)
 - menu Pawn, [111–113](#)
 - UMG widget, [105–107](#)
- 3D scenes, rendering, [9](#)

A

- Acceleration, detecting, [190](#)
- Add Loading Splash Screen function, [46](#)
- Affordances, [148](#)
- Aim assist, [81](#), [83](#)
- Ambient Occlusion, disabling, [224](#)
- Animations
 - arm movement, [138–144](#)
 - blending, [9](#)
 - creating inverse kinematics, [9](#)
 - editing. *See* [Persona](#).
 - hand movement. *See* [Hand IK](#).
 - head movement. *See* [Head IK](#).
 - interpolating joint movement. *See* [IK \(inverse kinematics\)](#).
 - player movement. *See* [Locomotion](#).
- Arm movement, inverse kinematics, [138–144](#)
- Artificial locomotion, [193](#)
- Auto Exposure, disabling, [224](#)

B

- Barrel distortion, [208–209](#)
- Base rotation and offset, getting, [46](#)
- Best practices, [10](#)
- Bloom, disabling, [224](#)
- Blueprints
 - animation for hand IK, [138–144](#)
 - vs. C++, [xiv](#)

description, [xiv](#)

enabling physics on Components, [166](#)

Gear VR global menu setup, [21](#)

GetForwardVector node, [55](#)

inheritance, [56](#)

native support for, [9](#)

Boxes, defining, [52](#). *See also* [Traces](#).

Bullet Train, [155](#)

Button mesh, creating, [167–168](#)

Buttons

activating, [173–174](#)

pressed, detecting, [173–174](#)

pressing, [170–174](#)

raising, [172–173](#)

triggering events with, [175](#)

uses for, [175](#)

Buttons, creating

button mesh, [167–168](#)

overlaps, detecting, [167–168](#)

visual representation, [168–170](#)

C

C++ vs. Blueprints, [xiv](#)

Camera managers, enabling Follow HMD, [46–47](#)

Cameras

cinematic, best practices, [10](#)

clipping plane, setting, [45](#)

controlling with player's head motions, [10](#)

shake, avoiding, [10](#)

Capsules, defining, [52](#). *See also* [Traces](#).

Cardboard, native support for, [5](#)

Channel traces, [53](#)

Circular spinner, creating, [26–32](#)

Classes vs. interfaces, [56](#)

Clear Loading Splash Screens function, [46](#)

Clustered rendering, [219–220](#)

Collision conservation, [166](#)

Collision responses, [53](#), [82](#)

Commands, console, [226–228](#)

Community resources, [245](#)

Configuring VR. *See* [VR project settings](#).

Conserving collision, [166](#)

Console commands, [226–228](#)
CPU bound vs. GPU bound, [227–228](#)
Culling unseen objects, [221](#)
Customizing VR. See [VR project settings](#).

D

Daydream motion controller, native support for, [6](#)
Daydream VR, native support for, [5](#)
Deferred rendering, native support for, [9](#)
Deferred shading vs. forward rendering, [215–220](#)
Deployment to a phone, readying for, [14](#)
Devices. See [Hardware](#).

Direction vectors

- getting, [55](#)
- length of, [52](#)

Disorientation, from teleportation, [86](#)

Documentation, Epic

- Gear VR project setup, [14](#)
- Rift setup, [32–41](#)
- Vive setup, [32–41](#)

Dot products, [80–83](#)

Dragging, hovered-on objects, [157–158](#)

Draw calls, checking, [228](#)

Dropped frames, compensating for, [8](#)

Dropping, objects, [158–160](#)

E

Editors

- animation. See [Persona](#).
- VR. See [VR editor](#).

Enable Auto Loading Splash Screen function, [46](#)

Enable HMD function, [45](#)

Enable Player Camera Manager Follow HMD function, [47](#)

Enable Player Controller Follow HMD function, [47](#)

Engine feature defaults, changing, [226–228](#)

Engine Scalability Settings, [226](#)

Epic

- Bullet Train, [155](#)
- resources, [244](#)

Epic, documentation

- Gear VR project setup, [14](#)
- Rift setup, [32–41](#)

Vive setup, [32–41](#)

F

FABRIK (Forward and Backward Reaching Inverse Kinematics), [126–127](#)

Field of view, best practices, [10](#)

First Person Template

running in place, [201–205](#)

snap turning, [194–201](#)

Focus state, getting, [44](#)

Follow HMD, enabling for HMDs, [46](#)

Forward kinematics, vs. inverse kinematics, [126](#)

Forward rendering

vs. deferred shading, [215–220](#)

native support for, [9](#)

FOV (field of view), [208–209](#)

Fragment shaders, [215–219](#)

Frame rates, best practices, [10](#)

Front buffer rendering, [8](#), [213–214](#)

Functions. *See also specific functions.*

generic, [44–45](#)

Oculus, [45–47](#)

Steam VR, [47](#)

G

G-buffers, [217–219](#)

Gear VR

circular spinner, creating, [26–32](#)

global menu setup, [20–26](#)

native support for, [4](#)

progress indicator, creating, [26–32](#)

project setup, [14–20](#)

readying for deployment, [14](#)

Geometric buffers, [217–219](#)

Get Base Rotation and Base Offset in Meters function, [46](#)

Get Hand Position and Orientation function, [47](#)

Get HMD Device Name function, [45](#)

Get Orientation and Position function, [44](#)

Get Player Camera Manager Follow HMD function, [46](#)

Get Pose function, [46](#)

Get Positional Tracking Camera Parameters function, [44](#)

Get Raw Sensor Data function, [46](#)

Get Screen Percentage function, [44](#)

Get Tracked Device Position and Orientation function, [47](#)

Get Tracking Origin function, [44](#)

Get Tracking Sensor Parameters function, [44](#)

Get User Profile function, [46](#)

Get Valid Tracked Device IDs function, [47](#)

Get VR Focus State function, [44](#)

Get World to Meters Scale function, [44](#)

GetForwardVector node, [55](#)

Google

- Cardboard, native support for, [5](#)

- resources, [245](#)

- VR, native support for, [5](#)

- VR SDK, native support for, [7](#)

GPU bound vs. CPU bound, [227–228](#)

Gravity, disabling, [166](#)

H

Hand IK, setting up. *See also* [Head IK](#); [IK \(inverse kinematics\)](#).

- adding motion controllers, [137–138](#)

- animation Blueprint, [138–144](#)

Hardware, natively supported. *See also* [Software, natively supported](#).

- Daydream motion controller, [6](#)

- Daydream VR, [5](#)

- Gear VR, [4](#)

- Google Cardboard, [5](#)

- Google VR, [5](#)

- HMDs, [4–5](#)

- motion controllers, [5–6](#)

- Oculus Touch, [5](#)

- OSVR (Open Source Virtual Reality), [5](#)

- PlayStation Move, [6](#)

- PlayStation VR, [5](#)

- Rift, [4](#)

- Vive, [4, 6](#)

Head IK, setting up. *See also* [Hand IK](#); [IK \(inverse kinematics\)](#).

- animation Blueprint, [132–137](#)

- Mirror creation, [128–130](#)

- Pawn creation, [130–132](#)

Head tracking. *See* [Head IK, setting up](#).

Hidden area mesh, [221](#)

Hidden mesh optimization, native support for, [8](#)

Hide Loading Icon function, [46](#)

Hide Loading Splash Screen function, [47](#)

HMD SP command, [44](#)

HMDs (Head Mounted Displays)

base rotation and offset, getting, [46](#)

detecting small motion changes, [46](#)

enabled, testing for, [45](#)

enabling, [45](#)

Follow HMD, enabling, [46](#)

generic functions, [44–45](#)

getting raw data from the IMU, [46](#)

Is Head Mounted Display Enabled function, [44](#)

name, getting, [45](#)

native support for, [4–5](#)

neck position for virtual head model, getting, [46](#)

position and rotation, getting, [46](#)

scaling positional tracking, [46](#)

testing for, [44](#)

Hovered-on objects. *See* [Objects, hovered on](#).

HTC/Valve

motion controllers, [149](#)

resources, [244–245](#)

I

IK (inverse kinematics)

arm movement, [138–144](#)

creating, tool for, [9](#)

FABRIK (Forward and Backward Reaching Inverse Kinematics), [126–127](#)

vs. forward kinematics, [126](#)

hand movement, [137–144](#)

head movement, [128–137](#)

introduction, [126–127](#)

Joint Targets, [126](#)

multi-bone method. *See* [FABRIK](#).

previewing a character, [128–130](#)

testing, [128–130](#)

two-bone method, [126](#), [138–144](#)

Inheritance

Blueprints, [56](#)

interfaces, [56](#)

Instanced Stereo rendering

native support for, [8](#), [220–221](#)

optimization, [220–221](#)

Interaction function, setting up, [64–71](#)

Interaction interface, setting up, [58–61](#)

Interaction Pawn

adding interaction to, [161–163](#)

setting up, [73–75](#)

Interactor component, creating, [153–163](#)

Interfaces, [55–56](#)

Is Auto Loading Splash Screen function, [46](#)

Is Head Mounted Display Enabled function, [44](#)

Is In Low Persistence Mode function, [45](#)

Is Loading Icon Enabled function, [46](#)

Is Player Controller Follow HMD Enabled function, [46](#)

J

Joint movement, interpolating. *See* [IK \(inverse kinematics\)](#).

Joint Targets, [126](#)

Judder, [8](#)

L

Latency mitigation

front buffer rendering, [213–214](#)

overview, [209–213](#)

scanline racing, [213–214](#)

a simple rendering pipeline, [209–214](#)

timewarping, [212–213](#)

Lens distortion, [208–209](#)

Level loads, best practices for asynchronous loading, [10](#)

Levers

components of, creating, [176–177](#)

resetting, [182–185](#)

rotating, [177–181](#)

triggering, [186–187](#)

visual representation of, [176–177](#)

Light intensity, calculating, [216–220](#)

Lines, traces

calculating a start and end point, [53–55](#)

downsides of, [80–83](#)

example, [52](#)

LineTrace function, setting up, [62–63](#)

Locomotion

acceleration, detecting, [190](#)

First Person Template, [194–201](#), [201–205](#)

optic flow, [190](#)

running in place, [201–205](#)

snap turning, [194–201](#)

vection, [190](#)

visual sensation of movement, [190](#)

Locomotion, types of. *See also* [Teleportation](#).

artificial, [193](#)

natural, [191](#)

physical, [192–193](#)

vehicle, [192](#)

Low persistence mode, testing for, [45](#)

M

Menu Actor, creating, [108–111](#)

Menu interaction, VR editor, [239–241](#)

Menu Pawn, creating, [111–113](#)

Menus

asynchronous loading, best practices, [10](#)

custom interaction, [113–122](#)

Gear VR global menu setup, [20–26](#)

global setup, [20–26](#)

Menus, creating

menu Actor, [108–111](#)

menu Pawn, [111–113](#)

UMG widget, [105–107](#)

Mirror creation, [128–130](#)

Mobile Scalable settings, [224](#)

Motion Blur, disabling, [224](#)

Motion changes, detecting, [46](#)

Motion controllers

adding to hand IK, [137–138](#)

affordances, [148](#)

HTC/Valve, [149](#)

introduction, [148](#)

natively supported, [5–6](#)

Oculus, [149](#)

picking up objects, [155](#)

player intent, determining, [155](#)

PlayStation, [149](#)

shared input paradigms, [149](#)

sphere overlap, alternative to, [155](#)

triggers, [149](#)

Motion controllers, interactive objects. *See also* [Buttons](#); [Levers](#).

conserving collision, [166](#)

gravity, disabling, [166](#)

physics simulations, disabling, [166](#)

Static Mesh Actor, [163–166](#)

Motion controllers, world interaction project

dragging on hovered objects, [157–158](#)

dropping objects, [158–160](#)

hover interface messages, changing, [155–156](#)

hovered-on objects, remembering, [153–156](#)

hovered-on objects, uses for, [161](#)

interacting with objects, [151–163](#)

Interaction Pawn, adding interaction to, [161–163](#)

interactor component, creating, [153–163](#)

interface, creating, [151–153](#)

object overlaps, detecting, [153–156](#)

picking up objects, [158–160](#)

setting up, [149–151](#)

with two controllers, [163](#)

Motion sickness. *See* [Simulator sickness](#).

MotionController Component, changing hand setting, [151](#)

Motion-to-photon time, [208](#)

Move, native support for, [6](#)

Movement. *See* [Locomotion](#).

N

Natural locomotion, [191](#)

Navigation, with the VR editor, [237](#)

Neck position for virtual head model, getting, [46](#)

O

Object traces, [53](#)

Object Types parameter, [53](#)

Objects

affordances, [148](#)

blocking lines, detecting, [52](#)

dropping, [158–160](#)

interacting with, [151–163](#). *See also* [Motion controllers](#).

interaction with the VR editor, [238–239](#)

interactive, setting up trace interaction, [75–79](#). *See also* [Motion controllers, interactive objects](#).

intersecting with other objects, [52](#)

overlaps, detecting, [153–156](#)

picking up, [158–160](#)

of player's attention, traces, [52–55](#)

players interacting with. *See* [Trace interaction](#).

properties that suggest use. *See* [Affordances](#).

searching for specific types of, [53](#)

selecting with the VR editor, [238–239](#)

transforming with the VR editor, [238–239](#)

between two points, detecting, [53–55](#). *See also* [Traces](#).

unseen, culling, [221](#)

Objects, hovered on

dragging, [157–158](#)

hover interface messages, changing, [155–156](#)

remembering, [153–156](#)

uses for, [161](#)

Occlusion, [123](#), [224](#)

Oculus. *See also* [Rift](#).

Audio SDK, native support for, [7](#)

functions, [45–47](#)

Mobile SDK, native support for, [7](#)

motion controllers, [149](#)

PC SDK, native support for, [6](#)

resources, [244](#)

Online resources. *See also* [Resources](#).

companion website for this book, [xv](#)

Epic documentation, [14](#)

practices that discomfort most players, [10](#)

Rift projects, [33](#)

Vive projects, [33](#)

OpenVR SDK, native support for, [7](#)

Optic flow, [190](#)

Optimization

hidden mesh, native support for, [8](#)

latency mitigation, [209–214](#)

project settings. *See* [VR project settings](#).

rendering. *See* [Rendering, optimization](#).

visible mesh, native support for, [8](#)

Origins

custom HMD, setting offset for, [47](#)

resetting orientation and position, [45](#)

Set Base Rotation and Base Offset in Meters function, [47](#)

Orthographical projection of user interfaces, [104](#)

OSVR (Open Source Virtual Reality), native support for, [5](#)

OSVR SDK, native support for, [7](#)

Overdraw, eliminating, [217](#)

Overlapping objects, detecting, [153–156](#)

Overlapping translucency, [225](#)

P

Parabolic tracing, [86–92](#)

Pawn creation, head IK, [130–132](#)

Performance improvements. *See* [Latency mitigation](#); [Rendering, optimization](#); [VR project settings](#).

Persona, native support for, [9](#)

Physical locomotion, [192–193](#)

Physics simulations, disabling, [166](#)

Picking up objects, [158–160](#)

Pixel shaders. *See* [Fragment shaders](#).

Player controller

- Follow HMD, enabling, [46](#)

- Follow HMD status, getting, [46](#)

Player discomfort

- common discomforting practices, [10](#)

- disorientation, [86](#)

- nausea, [10](#)

- orthographical projection of UIs, [104](#)

- simulator sickness, [190](#), [208](#)

- teleportation, [86](#)

Players

- aim assist, [81](#), [83](#)

- focus of attention, estimating, [80–83](#). *See also* [Trace interaction](#); [Traces](#).

- head motion, best practices, [10](#)

- intent, determining, [81](#), [155](#)

- detecting interaction with objects, [80–82](#). *See also* [Trace interaction](#).

- movement. *See* [Locomotion](#).

- previewing a character, [128–130](#)

- profiles, getting, [46](#)

- selecting objects, [81–83](#)

PlayStation

- motion controllers, [149](#)

- Move, native support for, [6](#)

- VR, native support for, [5](#)

Points, detecting objects between. *See* [Traces](#).

Points in space. *See* [Vertices](#).

Position, getting, [46](#)

Positional tracking, scaling, [46](#)

Pressing, buttons, [170–174](#)

Previewing a character, [128–130](#)

ProfileGPU command, [227](#)

Programmable pipeline, [215](#)

Progress indicator, creating, [26–32](#)

Project setup. *See specific projects.*

Proportional scaling of the VR view

World to Meters scale factor, getting, [44](#)

World to Meters scale factor, setting, [45](#)

Protocols. *See Interfaces.*

Q

Quick Select Menu, [239–241](#)

R

Radial Menu, [239–241](#)

Raising buttons, [172–173](#)

Rasterization, [217–218](#)

Ray casts. *See Traces.*

Registering this book, [xv](#)

Remembering, hovered-on objects, [153–156](#)

Rendering, natively supported

3D scenes, [9](#)

deferred, [9](#)

forward, [9](#)

front buffer, [8](#)

Instanced Stereo, [8](#), [220–221](#)

Rendering, optimization barrel distortion, [208–209](#)

clustered rendering, [219–220](#)

culling unseen objects, [221](#)

deferred shading, [215–220](#)

forward rendering vs. deferred shading, [215–220](#)

FOV (field of view), [208–209](#)

fragment shaders, [215–219](#)

G-buffers, [217–219](#)

geometric buffers, [217–219](#)

hidden area mesh, [221](#)

Instanced Stereo, [220–221](#)

light intensity, calculating, [216–220](#)

motion-to-photon time, [208](#)

native resolution, [209](#)

overdraw, eliminating, [217](#)

programmable pipeline, [215](#)

rasterization, [217–218](#)

screen percentage, [209](#)

shaders, [215–219](#)

tiled rendering, [219–220](#)

transparent objects, [218](#)

vertex shaders, [215–219](#)

VR lens distortion, [208–209](#)

VR project settings, [222–225](#)

Rendering pipeline, example, [209–214](#)

Reprojection, native support for, [8](#)

Reset Orientation and Position function, [45](#)

Resetting levers, [182–185](#)

Resolution, getting, [10](#)

Resources. *See also* [Online resources](#).

community, [245](#)

conferences, [246](#)

Epic, [244](#)

Google, [245](#)

HTC/Valve, [244–245](#)

Oculus, [244](#)

physical meetings, [245](#)

Rift

low persistence mode, [45](#)

native support for, [4](#)

tracking origins, [39–41](#), [44](#)

tracking screen percentage, [44](#)

Rift projects

documentation, [32–33](#)

setting up, [33–39](#)

Rotating levers, [177–181](#)

Rotation, getting, [46](#)

`r.screenPercentage` command, [44](#)

Running in place, [201–205](#)

S

Samsung Gear VR. *See* [Gear VR](#).

Scaling

Engine Scalability Settings, [226](#)

Get World to Meters Scale function, [44](#)

Mobile Scalable settings, [224](#)

positional tracking, HMDs, [46](#)

proportional scaling of the VR view, [44–45](#)

Set Position Scale 3D function, [46](#)

Set World to Meters Scale function, [45](#)

the world with the VR editor, [237](#)

Scanline racing, [8](#), [213–214](#)

Screen percentage

changing, [226](#)

definition, [209](#)

getting, [10](#)

Screen space effects, best practices, [10](#)

SDKs, native support for, [6–7](#)

Searching. *See* [Traces](#).

Seated origin vs. standing, [40–41](#)

Selecting objects, by player attention, [81–83](#)

Selecting objects with the VR editor, [238–239](#)

Separate Translucency, disabling, [225](#)

Sequencer, native support for, [9](#)

Set Base Rotation and Base Offset in Meters function, [47](#)

Set Clipping Plane function, [45](#)

Set Position Scale 3D function, [46](#)

Set Tracking Origin function, [45](#)

Set World to Meters Scale function, [45](#)

Setting up VR. *See* [VR project settings](#).

Shaders, [215–219](#)

Shapes, traces for, [52–53](#), [83](#)

Show Loading Icon function, [46](#)

Show Loading Splash Screen function, [47](#)

Simulator sickness, [190](#), [208](#)

Snap turning, [194–201](#)

Software, natively supported. *See also* [Hardware, natively supported](#).

animation editor. *See* [Persona](#).

Blueprints, [9](#)

cinematics tools. *See* [Sequencer](#).

Google VR SDK, [7](#)

hidden mesh optimization, [8](#)

Oculus Audio SDK, [7](#)

Oculus Mobile SDK, [7](#)

Oculus PC SDK, [6](#)

OpenVR SDK, [7](#)

OSVR SDK, [7](#)

Persona, [9](#)

reprojection, [8](#)

scanline racing, [8](#)

SDKs, [6–7](#)

Sequencer, [9](#)

timewarp, [8](#)

UMG (Unreal Motion Graphics), [9](#)

Unreal Engine systems, [9](#)

user interface system, [9](#)

visible mesh optimization, [8](#)

visual scripting language. *See* [Blueprints](#).

Sphere overlap, alternative to, [155](#)

Spheres, defining, [52](#). *See also* [Traces](#).

Splash screens between level loads

2D texture, [46](#)

adding, [46](#)

alternative 2D display, [47](#)

auto loading, detecting/enabling, [46](#)

clearing, [46](#)

customizing, [46](#)

enabling, [46](#), [47](#)

hiding, [47](#)

loading icon, showing/hiding, [46](#)

loading icon, testing for, [46](#)

turning on/off, [46](#), [47](#)

Standing origin vs. sitting, [40–41](#)

Stat Game command, [228](#)

Stat SceneRendering command, [228](#)

Stat Unit command, [227](#)

Static Mesh Actor, [163–166](#)

Steam VR controllers

device IDs, getting, [47](#)

device orientation and position relative to HMD origin, getting, [47](#)

hand position and orientation, getting, [47](#)

Steam VR functions, [47](#)

T

Target Hardware and Graphics Performance settings, [222–223](#)

Teleportation

description, [191–192](#)

downsides, [86](#)

parabolic tracing, [86–92](#)

pros and cons, [191–192](#)

setting up, [86–92](#)

vertical, [86–92](#)

Teleportation, visualizing

visualization Actor, [94–95](#)

visualization Material, [92–94](#)

Teleportation volume, [95–100](#)

Testing inverse kinematics, [128–130](#)

3D scenes, rendering, [9](#)

Tiled rendering, [219–220](#)

Timewarp, native support for, [8](#)

Timewarping, [212–213](#)

Tips. *See* [Best practices](#).

Touch, native support for, [5](#)

Trace Channel parameter, [53](#)

Trace interaction

overview, [52–55](#)

parabolic tracing, [86–92](#)

Trace interaction, setting up

basic interactive objects, [75–79](#)

basic project, [57–58](#)

interaction function, [64–71](#)

interaction interface, [58–61](#)

Interaction Pawn, [73–75](#)

LineTrace function, [62–63](#)

Trace Interaction Component, [61–73](#)

Trace Interaction Component

custom menu interaction, [113–122](#)

setting up, [61–73](#)

Traces

Channel, [53](#)

definition, [52](#)

direction vectors, getting, [55](#)

Object, [53](#)

for object of player's attention, [52–55](#)

setting up, [53–55](#)

shapes, [52–53](#), [83](#). *See also specific shapes.*

for specific types of objects, [53](#)

types of, [52–53](#)

world origin, getting, [55](#)

Traces, lines

calculating a start and end point, [53–55](#)

downsides of, [80–83](#)

example, [52](#)

Tracking head movement. *See* [Head IK, setting up](#).

Tracking origins

Eye Level, setting, [45](#)

Floor Level, [40–41](#)

Floor Level, setting, [45](#)

Get Orientation and Position function, [44](#)

Get Positional Tracking Camera Parameters function, [44](#)

Get Tracking Origin function, [44](#)

Get Tracking Sensor Parameters function, [44](#)

Rift, [39–41](#), [44](#), [45](#)

seated vs. standing, [40–41](#)

Set Tracking Origin function, [45](#)

Vive, [39–41](#), [44](#), [45](#)

Tracking origins, defaults

Rift, [40](#), [44](#)

Vive, [41](#), [44](#)

Translucency, overlapping, [225](#)

Transparent objects, rendering, [218](#)

Triggering

events with buttons, [175](#)

levers, [186–187](#)

Triggers, motion controllers, [149](#)

Trigonometry of rotating levers, [179–181](#)

2D texture splash screen, [46](#)

2D UIs (user interfaces)

custom menu interaction, [113–122](#)

issues with, [104](#)

readability, [104](#)

2D UIs (user interfaces), creating a basic VR menu

menu Actor, [108–111](#)

menu Pawn, [111–113](#)

UMG widget, [105–107](#)

Two-bone method of inverse kinematics, [126](#), [138–144](#)

U

UIs (user interfaces)

accessing with VR editor, [240–241](#)

determining the need for, [104](#)

head locked. *See* [2D UIs](#).

intersection with game geometry. *See* [Occlusion](#).

native support for, [9](#)

orthographical projection, [104](#)

two-dimensional. *See* [2D UIs](#).

UMG (Unreal Motion Graphics)

compatibility, [105](#)

history of, [105](#)

native support for, [9](#)

widget for basic menu, creating, [105–107](#)

Unreal Engine systems, native support for, [9](#)

Users. *See* [Players](#).

V

Valve

motion controllers, [149](#)

resources, [244–245](#)

Vection, [190](#)

Vectors, multiplying together. *See* [Dot products](#).

Vehicle locomotion, [192](#)

Vertex shaders, [215–219](#)

Vertices, definition, [215](#)

Vestibular system, [190](#)

Visible mesh optimization, native support for, [8](#)

Visual sensation of movement, [190](#)

Visualizing teleportation

visualization Actor, [94–95](#)

visualization Material, [92–94](#)

Vive

low persistence mode, [45](#)

native support for, [4](#), [6](#)

tracking origins, [39–41](#)

tracking origins, defaults, [41](#), [44](#)

tracking screen percentage, [44](#)

Vive projects

documentation, [32–33](#)

setting up, [33–39](#)

VR editor

accessing UI elements, [240–241](#)

controller, [235–236](#)

enabling, [234](#)

menu interaction, [239–241](#)

navigation, [237](#)

object interaction, [238–239](#)

opening, [235](#)

Quick Select Menu, [239–241](#)

Radial Menu, [239–241](#)

scaling the world, [237](#)

selecting objects, [238–239](#)

transforming objects, [238–239](#)

VR lens distortion, [208–209](#)

VR project settings

Ambient Occlusion, disabling, [224](#)

Auto Exposure, disabling, [224](#)

Bloom, disabling, [224](#)

console commands, [226–228](#)

CPU bound vs. GPU bound, [227–228](#)

draw calls, checking, [228](#)

engine feature defaults, changing, [226–228](#)

Engine Scalability Settings, [226](#)

Mobile Scalable settings, [224](#)

Motion Blur, disabling, [224](#)

overlapping translucency, [225](#)

ProfileGPU command, [227](#)

rendering optimization, [222–225](#)

screen percentage, changing, [226](#)

Separate Translucency, disabling, [225](#)

Stat Game command, [228](#)

Stat SceneRendering command, [228](#)

Stat Unit command, [227](#)

Target Hardware and Graphics Performance, [222–223](#)

W

Websites. See [Online resources](#).

World origin, getting, [55](#)

World to Meters scale factor

getting, [44](#)

setting, [45](#)

WYSIWYG (what you see is what you get), [105](#)



REGISTER YOUR PRODUCT at informit.com/register Access Additional Benefits and SAVE 35% on Your Next Purchase

- Download available product updates.
- Access bonus material when applicable.
- Receive exclusive offers on new editions and related products. (Just check the box to hear from us when setting up your account.)
- Get a coupon for 35% for your next purchase, valid for 30 days. Your code will be available in your InformIT cart. (You will also find it in the Manage Codes section of your account page.)

Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com you can

- Shop our books, eBooks, software, and video training.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletters (informit.com/newsletters).
- Read free articles and blogs by information technology experts.
- Access thousands of free chapters and video lessons.

Connect with InformIT—Visit informit.com/community

Learn about InformIT community events and programs.



informIT.com
the trusted technology learning source

Addison-Wesley • Cisco Press • IBM Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • VMware Press