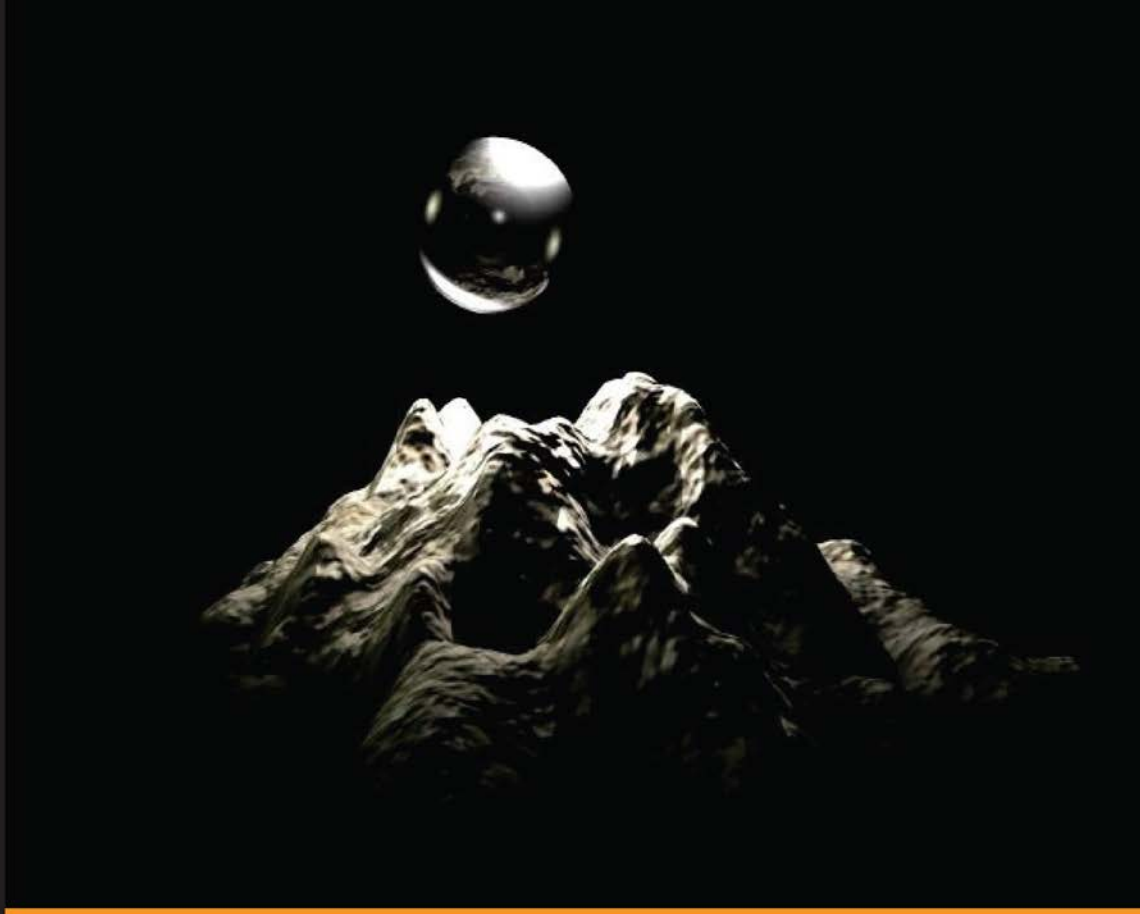


Community Experience Distilled

Unreal Engine Physics Essentials

Gain practical knowledge of mathematical and physics concepts in order to design and develop an awesome game world using Unreal Engine 4

Katax Emperore Devin Sherry

[PACKT] PUBLISHING

Unreal Engine Physics Essentials

Gain practical knowledge of mathematical and physics concepts in order to design and develop an awesome game world using Unreal Engine 4

Katax Emperore

Devin Sherry

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Unreal Engine Physics Essentials

First published: September 2015

Production reference: 1230915

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-490-5

www.packtpub.com

Credits

Authors
Katax Emperore
Devin Sherry

Reviewer
Abdullah Obaied

Commissioning Editor
Edward Bowkett

Acquisition Editor
Indrajit Das

Content Development Editor
Shubhangi Dhamgaye

Technical Editor
Siddhesh Ghadi

Copy Editor
Relin Hedly

Project Coordinator
Harshal Ved

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Authors

Katax Emperore was born in January 1974. Since his childhood days, he has loved board games, science magazines, comic books, and graffiti painting. Katax was introduced to the IT world when he got his first game platform, Fire Attack from the Game & Watch series by Nintendo, back in the 80's. While spending hours on it, he became curious about the process of creating games.

As a teenager, Katax owned Amiga 500 by Commodore on which he played hundreds of games. However, one stuck with him: Shadow of the Beast by Psygnosis. He was enamored by the quality of the graphics, music, and FX sounds involved in the game. Katax realized that he would like to learn to create such games, and this was the first step. Today, he can design and develop any game on various web pages and platforms alike.

The Amiga platform created a high-quality gaming experience supported by an advanced hardware architecture that was way ahead of its time. It was a high-profile computer with real stereo sound supported by the advanced Direct Memory Access technology for multiprocessing. On this platform, Katax learned many aspects of programming, multitasking, DMA, interactive applications, IO port mappings, graphic design, and 3D. When Microsoft introduced Windows 98, he got serious about programming, 3D, and graphic design, which led him to base his education, and later his career, in the IT industry.

Around this time, Katax experienced a live performance of a digital visual art improvisation over music known as a VJ performance. He was influenced by Jimi Hendrix, and he adopted his style of improvising each track on a live stage. His style involves visualizing forms, colors, and brightness of images and videos by playing live visual transitions over each pixel on the screen. Katax believes that it's necessary for each game designer/developer to be a part of some art movement or activity because it helps you in your career, both technically and spiritually.

Katax' favorite bands/artists include Klaus Schulze, Tangerine Dream, Hawkwind, and Jimi Hendrix.

I am grateful to John Carmack from id Software for his efforts and great work on 3D graphic programming. What he invented and developed back in the 90's was the beginning of the wonderful genre of first-person shooter games, which is my personal favorite. Also, I would like to thank Westwood Studios for introducing the Command and Conquer (C&C) series to the gaming world. This game pioneered many aspects of the modern real-time strategy games, which later powered many subgenres in this area as well. Great job, thank you!

Devin Sherry is originally from Levittown, Long Island in the state of New York, USA. He studied game development and game design at the University of Advancing Technology where he obtained a bachelor's degree of arts in game design in 2012.

During his time in college, Devin worked as a game and level designer with a group of students called Autonomous Games on a real-time, strategy style, third-person shooter game called The Afflicted using Unreal Engine 3/UDK. It was presented at the Game Developers Conference (GDC) in 2013 in the the GDC Play showcase.

Currently, Devin works as an independent game developer located in Tempe, Arizona, where he works on personal and contracted projects. His achievements include the title Radial Impact, which can be found in the Community Contributions section of the Learn tab of Unreal Engine 4's Launcher. You can follow Devin's work on his YouTube channel, Devin Level Design, where he educates viewers on game development within Unreal Engine 3, UDK, and Unreal Engine 4.

About the Reviewer

Abdullah Obaied is a self-taught software engineer with a long history of expertise in game programming. He has professionally worked on games such as Artifice: Faith in Chaos as well as on engines such as Unity and Unreal Engine 4. Furthermore, using the DirectX technology, Abdullah has developed his own engine labeled Nucleus Engine. He was also one of the developers selected to beta-test Unreal Engine 4 back in its early development stage.

Currently, Abdullah works at Mindvalley in Kuala Lumpur, Malaysia in the mobile app development team as an Android developer. You can follow him on his blog, `http://damngoodcode.blogspot.com`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Giving readers practical insight into the principles of mathematics and physics necessary to properly implement physics within Unreal Engine 4, this book covers everything one needs to know in order to create a game world.

Discover how to manipulate physics within Unreal Engine 4 by learning from scratch the basic real-world concepts in mathematics and physics that assist in the implementation of physics-based objects in your game world. Then, be introduced to PhAT (Physics Asset Tool) within Unreal Engine 4 to learn more about developing physics objects for your game world. Next, dive into Unreal Engine 4's collision generation, physical materials, blueprints, constraints, and more to get hands-on experience with the tools provided by Epic Games to create the effect of the real physical world in Unreal Engine 4. Lastly, you will create a working Vehicle Blueprint that uses all the concepts covered in this book, and also cover topics related to advanced physics.

What this book covers

Chapter 1, *Math and Physics Primer*, covers basic concepts in mathematics and physics that will assist your understanding of how the real-world, and Unreal Engine 4, works.

Chapter 2, *Physics Asset Tool*, discusses how to properly utilize the Physics Asset Tool to create physics assets to use with Skeletal Meshes.

Chapter 3, *Collision*, invites readers to learn and apply collisions to physics assets in order to experiment with physics simulations.

Chapter 4, *Constraints*, discusses how to implement constraints onto actors and blueprints using the Physics Asset Tool.

Chapter 5, *Physics Damping, Friction, and Physics Bodies*, defines physics damping, friction, and physics bodies in the context of Unreal Engine 4.

Chapter 6, *Materials*, discusses physical materials and how to utilize them to create a realistic game world.

Chapter 7, *Creating a Vehicle Blueprint*, takes a look at applying the skills learned in the previous chapters to create the physical body and blueprint of a working vehicle.

Chapter 8, *Advanced Topics*, covers advanced topics and troubleshooting techniques in physics.

What you need for this book

This book assumes that readers have access to Unreal Engine 4 and can utilize version 4.7.0 or higher. If you do not have Unreal Engine 4, you can visit `www.unrealengine.com/what-is-unreal-engine-4` to download the engine.

Furthermore, for those who are not familiar with how to use Unreal Engine 4, this text will walk you through some of the basics, but also assume that its readers have at least some experience with the engine.

Who this book is for

This book is intended for beginner to intermediate users of Epic Games' Unreal Engine 4 who want to learn more about how to implement physics within their game world.

No matter the knowledge base of Unreal Engine 4, this book contains valuable information on blueprint scripting, collision generation, materials, and the Physical Asset Tool (PhAT) for all users to create better games.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can name this parameter `Material Color`."

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/4905OT_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Math and Physics Primer

In this chapter, we will discuss and evaluate the basic 3D physics and mathematics concepts in an effort to gain a basic understanding of Unreal Engine 4 physics and real-world physics. To start with, we will discuss the units of measurement, what they are, and how they are used in Unreal Engine 4. In addition, we will cover the following topics:

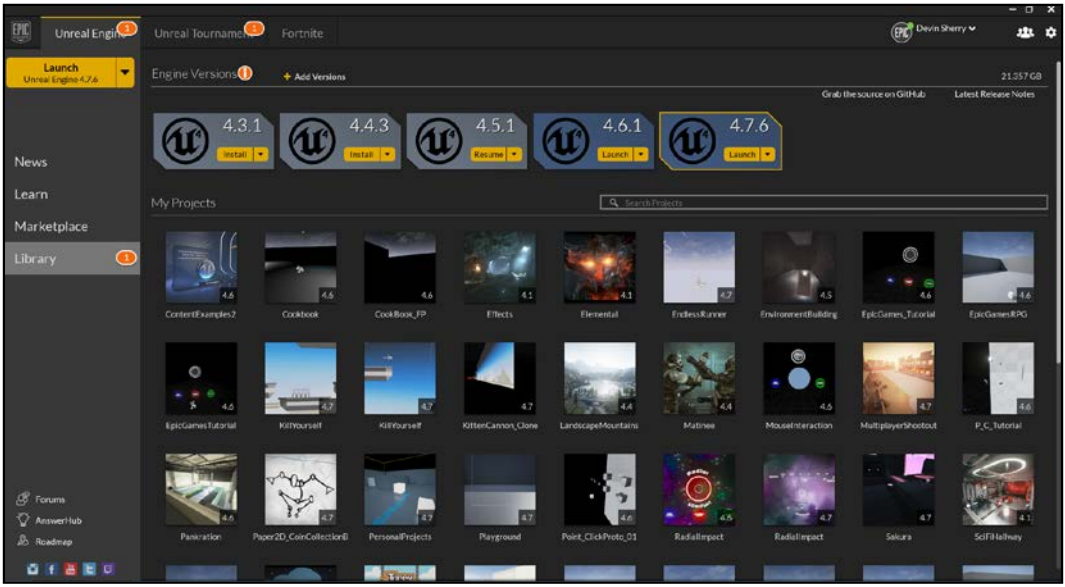
- The scientific notation
- 2D and 3D coordinate systems
- Scalars and vectors
- Newton's laws or Newtonian physics concepts
- Forces and energy

For the purpose of this chapter, we will want to open Unreal Engine 4 and create a simple project using the **First Person** template by following these steps.

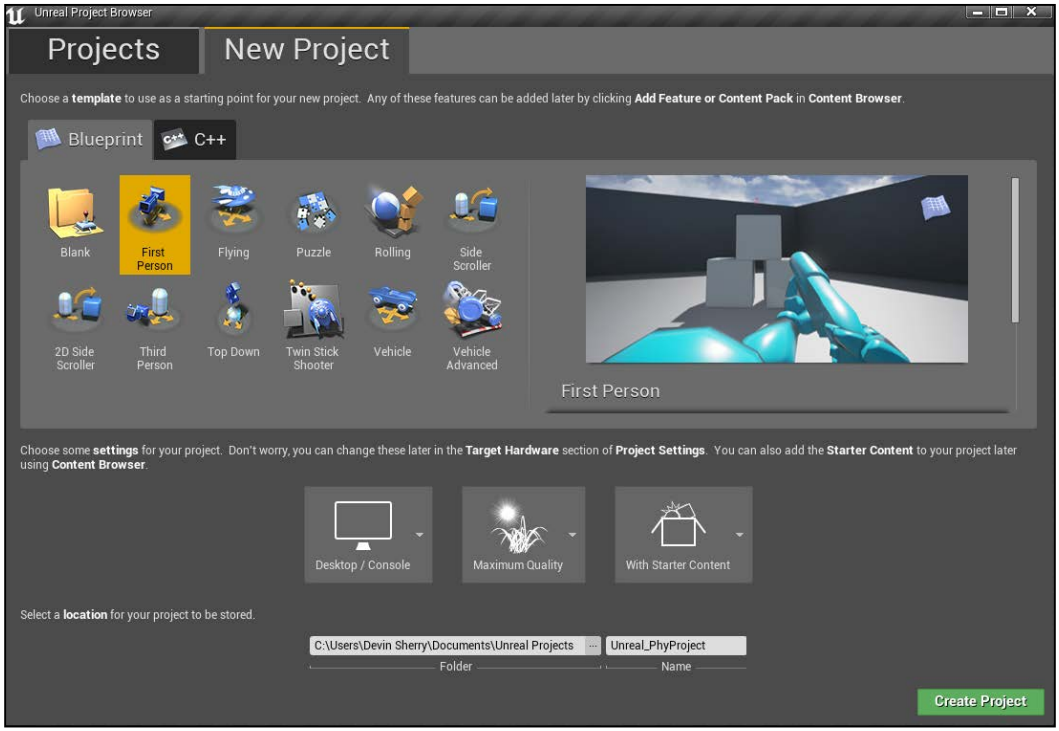
Launching Unreal Engine 4

When we first open Unreal Engine 4, we will see the Unreal Engine launcher, which contains a **News** tab, a **Learn** tab, a **Marketplace** tab, and a **Library** tab. As the first title suggests, the **News** tab provides you with the latest news from Epic Games, ranging from Marketplace Content releases to Unreal Dev Grant winners, Twitch Stream Recaps, and so on. The **Learn** tab provides you with numerous resources to learn more about Unreal Engine 4, such as written documentation, video tutorials, community wikis, sample game projects, and community contributions. The **Marketplace** tab allows you to purchase content, such as FX, weapons packs, blueprint scripts, environmental assets, and so on, from the community wikis and Epic Games. Lastly, the **Library** tab is where you can download the newest versions of Unreal Engine 4, open previously created projects, and manage your project files.

Let's start by first launching the Unreal Engine launcher and choosing **Launch** from the **Library** tab, as seen in the following image:



For the sake of consistency, we will use the latest version of the editor. At the time of writing this book, the version is 4.7.6. Next, we will select the **New Project** tab that appears at the top of the window, select the **First Person** project template with **Starter Content**, and name the project `Unreal_PhyProject`:



Now that we have the game engine open, we can now continue with our lesson.

Units of measurement

To begin this section, we want to first define measurement and what exactly we will measure in the context of Unreal Engine 4. In a general sense, the definition of measurement is determining the size, length, or the amount of something (such as distance), the length/width/height of an object, or the volume of a particular space. In the context of Unreal Engine 4, we will measure the lengths of each component of a 3D vector in the 3D space and the *X*, *Y*, and *Z* dimensions. For the 2D game world, we will measure the *X* and *Y* axes. In the real world, we can use the U.S. Standard and the European Standard units of measurement to measure distance.

In the U.S., we can use the standard of lengths that involve the use of **inches** (**in**), **feet** (**ft**), **yards** (**yd**), and miles, whereas in Europe, there is the standard of lengths in place that includes **millimeters** (**mm**), **centimeters** (**cm**), **meters** (**m**), and **kilometer** (**km**). For our convenience and as a point of reference, here are a set of conversion charts between the U.S. and the European units of measurements. For more conversions, refer to this free conversion website at `http://converter.eu/length/`.

The following table shows the U.S. Conversions:

1 ft	12 in
1 in	0.0833333 ft
1 yd	3 ft
1 yd	36 in
1 Mile	1,760 yd
1 Mile	5,280 ft
1 Mile	63359.999 in

The following table shows the European Conversions:

1 mm	0.1 cm
1 cm	10 mm
1 cm	0.0099999 m
1 m	100 cm
1 m	1000 mm
1 mm	0.000999999 m
1 km	1000000 mm
1 km	100000 cm
1 km	1000 m

The following table shows U.S. to European Conversions:

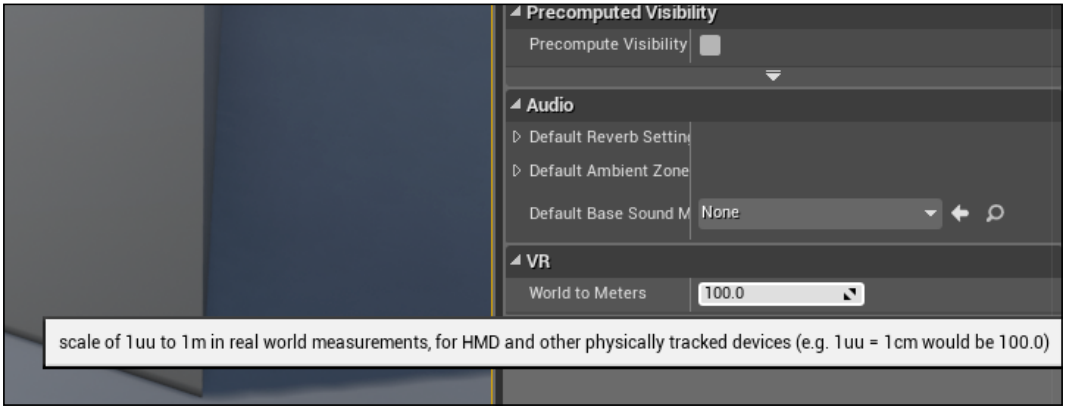
1 in	25.4 mm
1 in	2.54 cm
1 in	0.0254 m
1 in	0.0000254 km
1 ft	304.8 mm
1 ft	30.48 cm
1 ft	0.3048 m
1 ft	0.0003048 km
1 yd	914.4 mm
1 yd	91.44 cm
1 yd	0.9144 m
1 yd	0.0009144 km

1 Mile	1609344 mm
1 Mile	160934.4 cm
1 Mile	1609.3439999999998 m
1 Mile	1.609344 km

Now that we have a strong understanding of the real-world units of measurement, we are now ready to discuss how Unreal Engine 4 uses these units of measurement to determine distances and sizes of objects.

What is an Unreal Unit?

Back in the days of the UDK or Unreal Engine 3, the units of measurement were based on what was called **Unreal Units** (**uu**), where one uu equaled 0.75 in, or 16 units equaled 1 ft. In Unreal Engine 4, the measurement has changed to where 1 uu is equal to 1 cm by default, but the engine allows you to change the conversion ratio between an Unreal Unit and a meter in its **World** settings, as shown in the following screenshot:

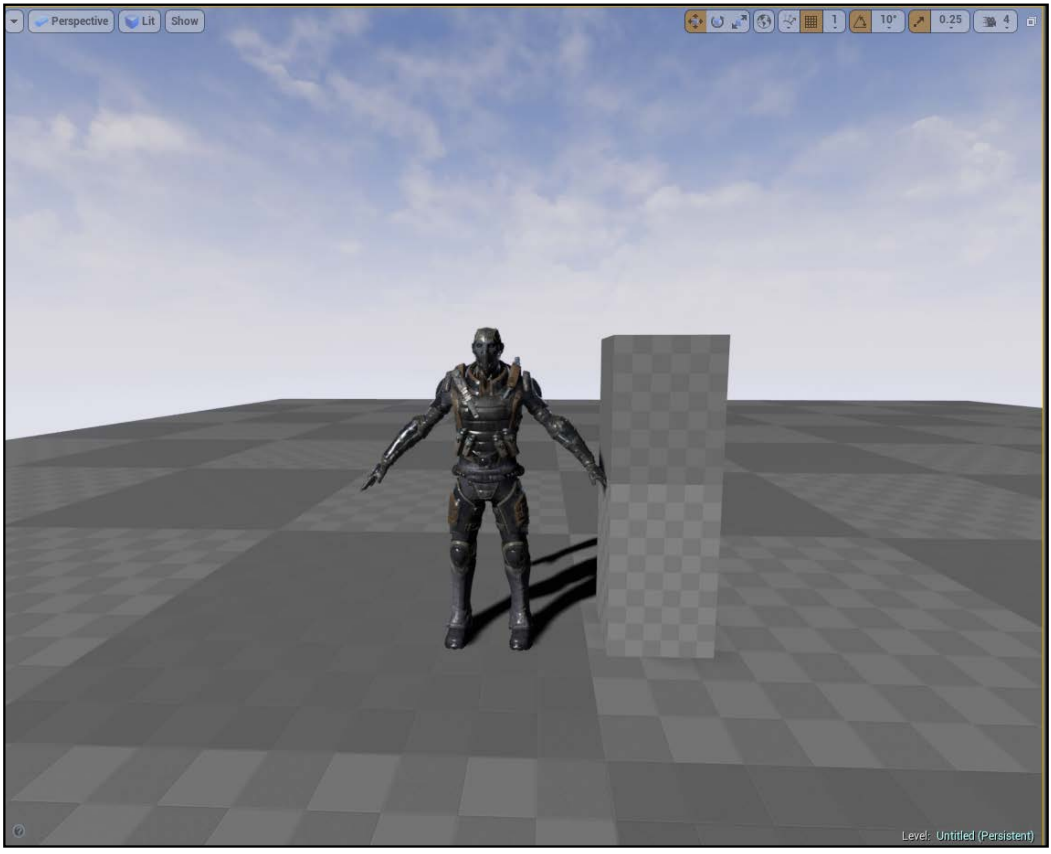


The value of `100.0` in this property equates an Unreal Unit to 1 cm. For example, by changing the **World to Meters** property to the value of 1 (as shown before), it will equate 1 uu to 1 m and a value of 1,000 will result in 1 uu equaling 1 mm. For the purposes of this project, we will leave the default value of 100 so that an Unreal Unit will equal 1 cm, but for future reference, this is the **World** settings property you would want to alter in order to change this conversion ratio.

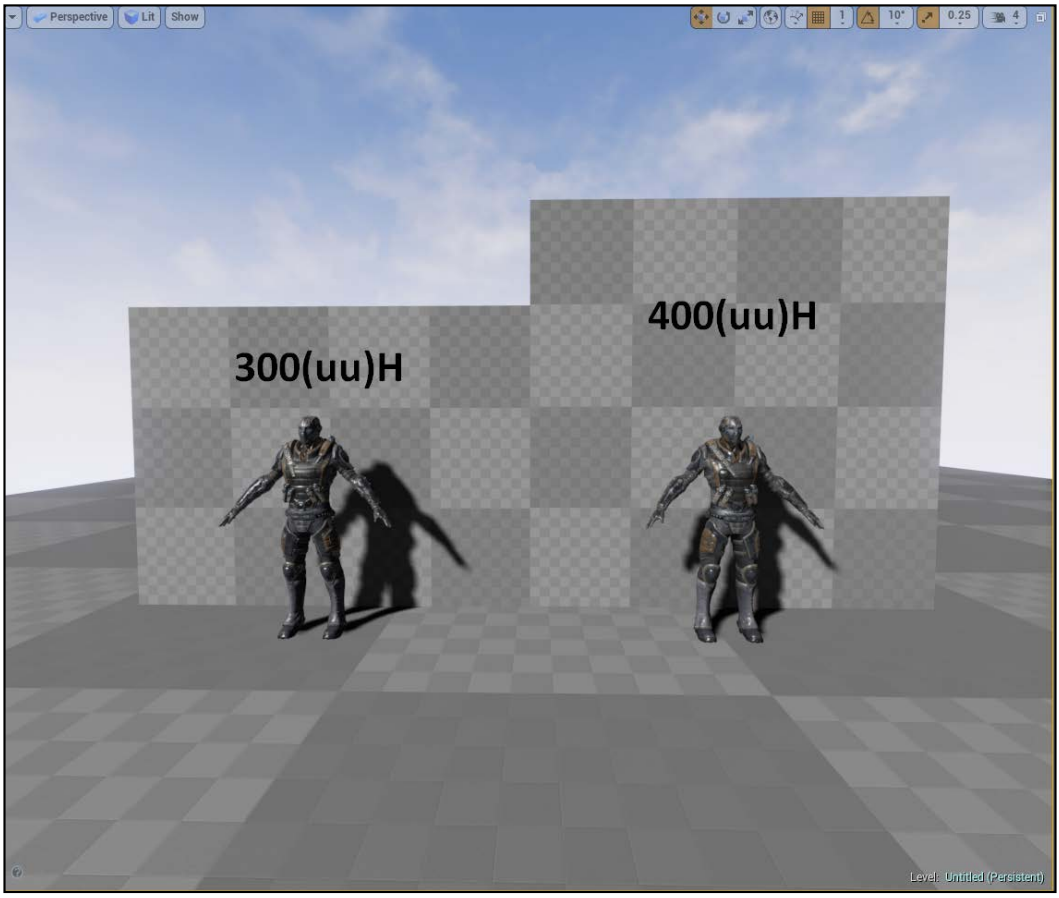
Common measurements in Unreal Engine 4

When you work on any game engine, it is very important and useful to know the common measurements that are used in your game world. Each game is different, and the scaling of that game world will be different depending on whether or not the developers are going for realistic scaling measurements, but for the purposes of this book, the following measurements will be for a game world that is going for realism. Remembering that by default, 1 uu is equal to 1 cm in Unreal Engine 4, here are some of the common measurements that you can implement in your game world. An additional note is that all the following dimensions are set under the assumption that your player character is roughly 6 ft tall or 180 uu.

The dimensions of a player character are 180(uu)H, 60(uu)W, 60(uu)D. These are dimensions for a larger character that is roughly 6 ft tall, so you may need to adjust these values accordingly based on your character's height.

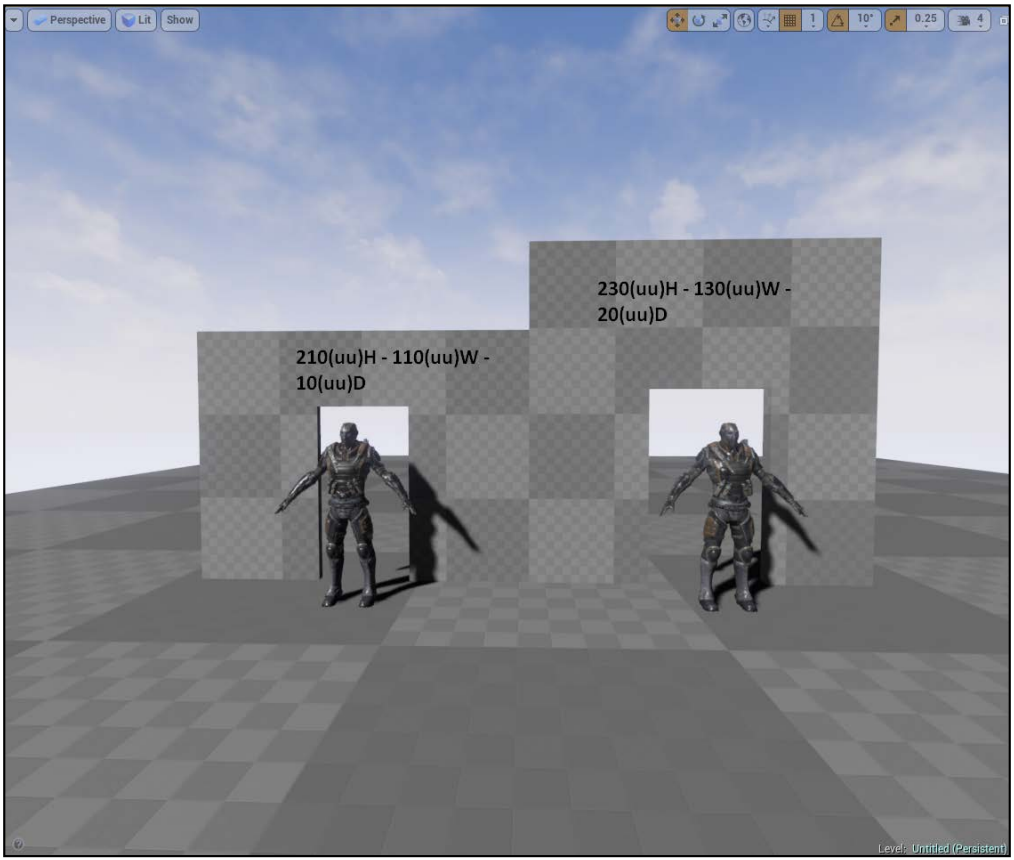


The wall height is 300(uu)H to 400(uu)H. A value of 400 uu will produce a slightly taller wall, whereas a value of 300 uu will result in a slightly shorter wall, but any value between 300 uu and 400 uu will work just fine.



The wall depth (thickness)is 10(uu)D to 20(uu)D. The value of the wall thickness depends greatly on the material that the wall is made of. For example, a brick wall would be thicker than a wall made of plaster.

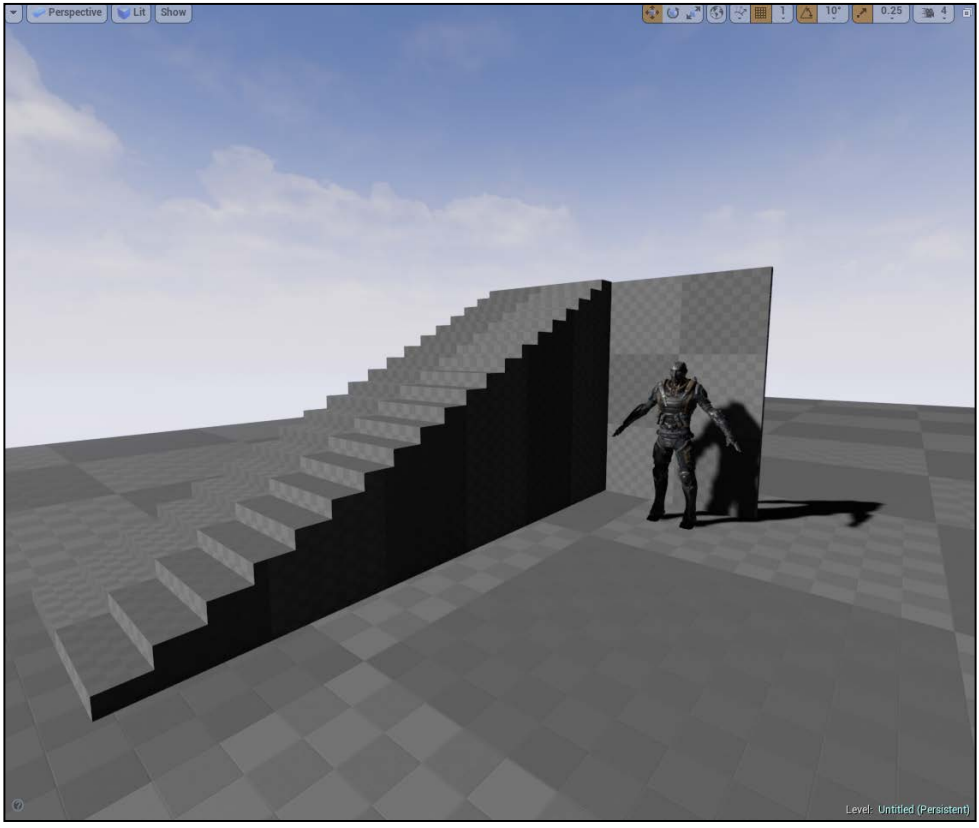
The dimensions of doors and doorways are 210(uu)H – 230(uu)H / 110(uu)W – 140(uu)W. The value of the door and the doorway depth (thickness) depends on the value of the wall thickness.



Staircases

- The step height is 15(uu)H
- The step length/depth is 30(uu)D

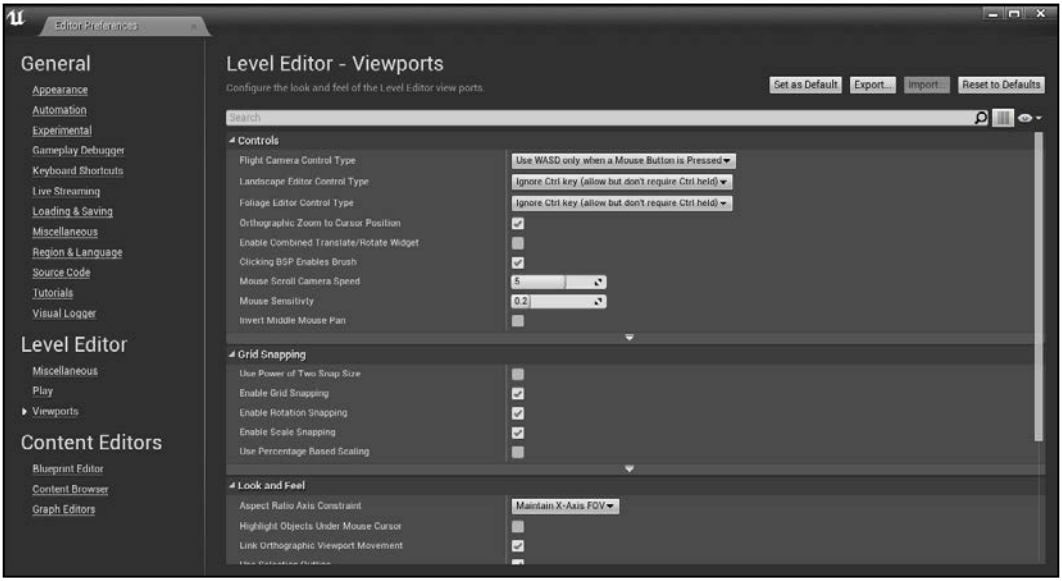
The value of the staircase width will depend on the area that the staircase is placed in, so the dimensional measurement of width will vary. The following image has a step length of 30(uu)L, a step height of 15(uu)H, a step width of 200(uu)W, and 20 steps in total:



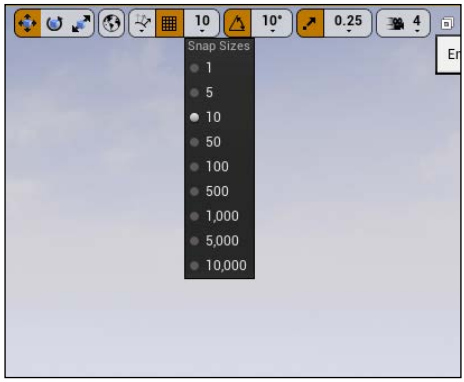
Unit snapping in Unreal Engine 4

If you open a blank map in Unreal Engine 4 or the different viewports in the editor, you will notice a grid. This grid can change dynamically depending on the current unit snapping measurement applied in the editor. The spacing between each grid square will determine the number of units an object will move when you place or transform objects in the editor. Back in Unreal Engine 3 or the UDK, the grid snapping would follow the power of 2 (2/4/8/16/32/64/128/256/512/1024/2048) , but in Unreal Engine 4, the grid snapping follows these values (1/5/10/50/100/5 00/1000/5000/10000). The main reason for this change is due to the fact that Unreal Engine 4 uses the value of an Unreal Unit equaling 1 cm instead of 1 uu equaling 0.75 in. This is similar to what it did in Unreal Engine 3 or the UDK.

By default, the unit snapping grid follows the notion that 1 uu equals 1 cm, but if we were to follow the power of 2 unit snapping scale, we have this option. In **Editor Preferences** under the **Level Editor** section, there is an option for **Viewports**. Under **Viewports**, there is a subsection labeled **Grid Snapping** and an option to enable/disable the **Use Power of Two Snap Size**, as shown in the following screenshot:



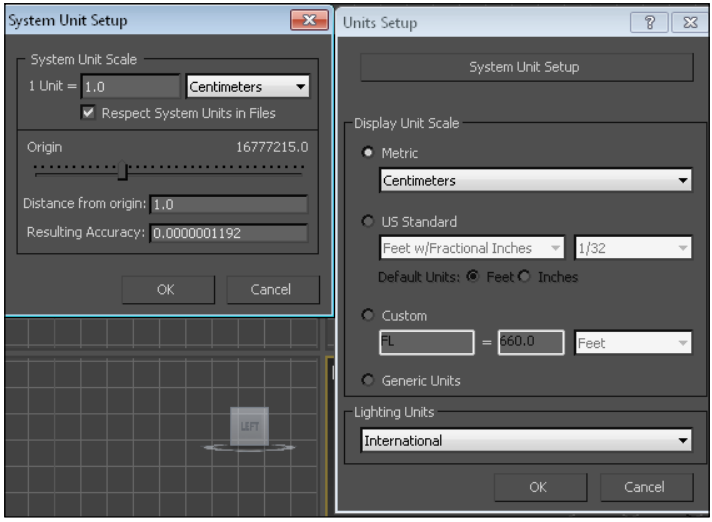
When it comes to unit snapping, follow the measurement that works best for you. Unit snapping is a very important aspect when it comes to placing assets in your game world. It can be a lifesaver when it comes to avoiding clipping or Z fighting between two objects. Unit snapping is also crucial when it comes to creating proper distances between objects, such as creating hallways or alleyways between buildings. In the end, it will save a lot of time and effort to take unit snapping into consideration at the beginning stages of level development and particularly during the white box stages of level design. There will also be specific instances when placing objects in our game world where unit snapping is not necessary, such as placing debris on the ground, placing paper on a desk, or any other objects that don't require specific distances between themselves and other in-game objects. When it comes to these instances, Unreal Engine 4 gives us the ability to toggle unit snapping on and off by clicking on the grid icon, as shown in the following screenshot. Lastly, we can also snap our objects to the grid or the floor of our environment by pressing the *End* key. Alternatively, we can press *Ctrl* + *End* to snap an actor to the grid. If we ever need to change the key bindings for these actions, we can navigate to the **Edit Menu** | **Editor Preferences** | **Keyboard Shortcuts** to make any changes.



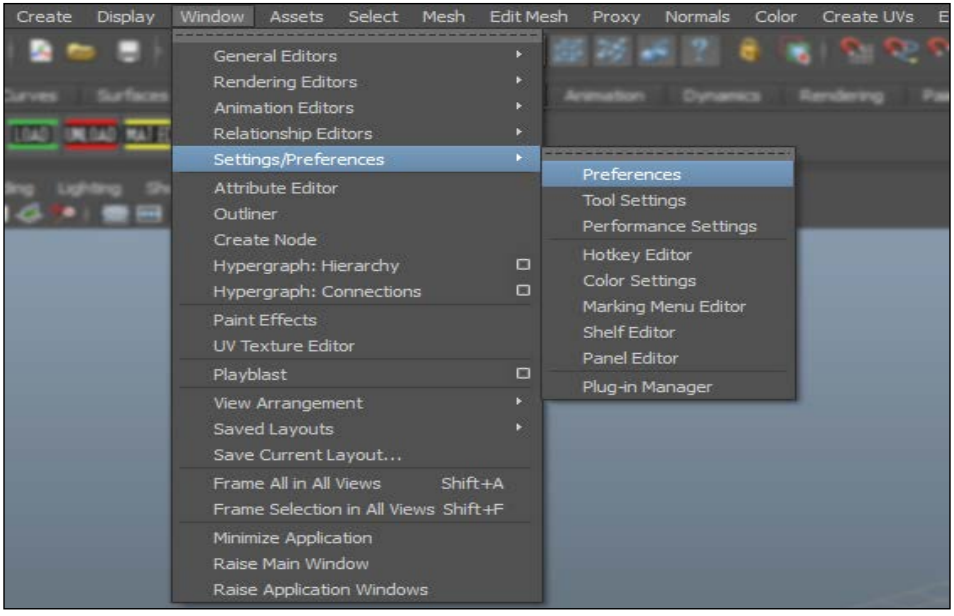
Changing units of measurement in 3ds Max and Maya

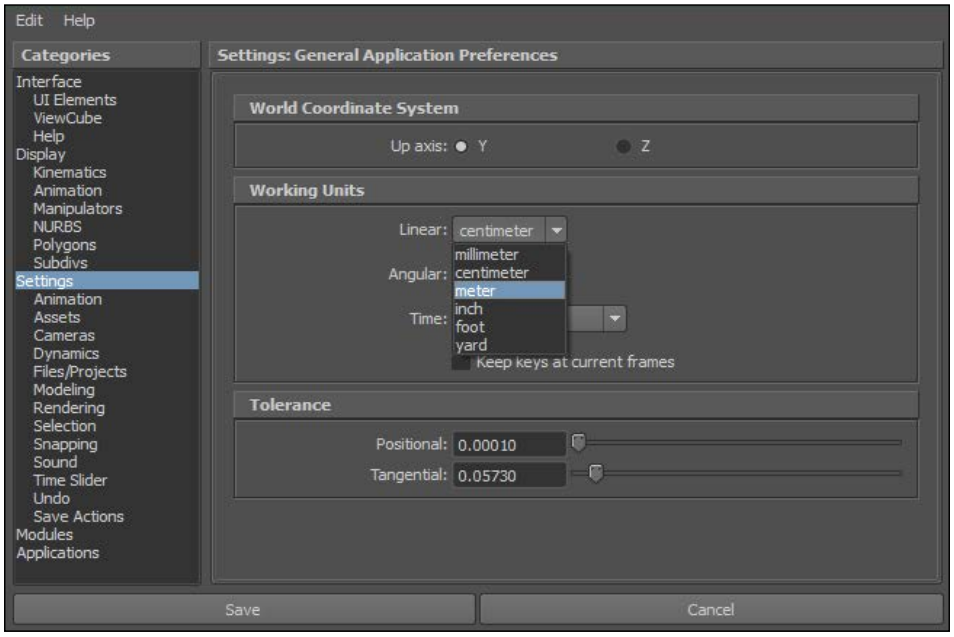
For both character and environmental artists, it is very important to know how to change the units of measurement in the third-party 3D modeling so that when assets are exported from the art program and then imported to Unreal Engine 4, the scale is correct and as intended by the artist. Keeping in mind that Unreal Engine 4 uses the measurement conversion of 1 uu equaling 1 cm by default, we want to make sure that the units of measurement in our 3D art program uses the same conversion.

To change the units of measurement in 3ds Max (2013 version), select the **Customize** option and then **Units Setup**. Here, click on the **System Unit Setup** button and change the units of measurement as follows:



In Maya, we can change the units of measurement by clicking on Window from the toolbar. Now, select **Settings/Preferences** from the drop-down window and then **Preferences**. In the **Preferences** dialogue box, select **Settings**. Under **Working Units**, we can change the linear units to **centimeter**.





Units of measurement – a section review

In this section, you learned about the basic unit conversions between the U.S. and European units and how these units translate into Unreal Engine 4's Unreal Units. Additionally, we briefly discussed the common measurements for our game world for our player character, walls, staircases, and doors/doorways. Moreover, we took an in-depth look at unit snapping in Unreal Engine 4 and the significance of the tool when it comes to object placement and creating our game world. Lastly, we looked at how to convert or change the units of measurement in 3D art programs, such as 3ds Max and Maya so that artists can ensure that their models are exported and imported to the correct scale when placed in Unreal Engine 4.

Now that we have a better understanding of the units of measurement and how they translate into Unreal Engine 4, we can now move forward to briefly discuss scientific notation.

The scientific notation

This is a method in which we can easily write very large or significantly small numbers without having to express the entire length of the number, meaning writing a bunch of zeroes. The use of scientific notation is not very common when you use Unreal Engine 4 as a designer or an artist, but as a programmer or a technical designer who uses blueprints or even C++ coding in the engine, the use of scientific notation can deem itself useful.

Let's take a look at some examples of both large and small numbers that are expressed in their scientific notation. To keep things as simple as possible, these examples will use the base of 10 for ease of clarity:

- 1,000 (1 thousand) – *1 * 10^3*
- 100,000 (1 hundred thousand) – *1 * 10^5*
- 1,000,000 (1 million) – *1 * 10^6*
- .01 (1 hundredth) – *1 * 10^-2*
- .001 (1 thousandth) – *1 * 10^-3*
- .0001 (1 ten thousandth) – *1 * 10^-4*

How to use scientific notation?

The main logic behind using scientific notation is to take a very large or small number and convert it to an easy to read/write expression. For an example that isn't a power of 10, the number 0.5 converted to scientific notation would read as *5 * 10^-1*. We reached this expression by moving the decimal point in 0.5 once to the right-hand side making the number into 5. The goal of using scientific notation is to reach the base number, meaning a number between 1 and 9. As we had to move the decimal point to the right-hand side, we know that the expression would read as a negative exponent, whereas if we were to move the decimal point to the left-hand side, the exponent would be positive. The number 5 is our base, and we multiply it by 10 with an exponent that is equal to the number of times we moved the decimal point to reach the said base. In our case, it would be 1. Lastly, we know that the exponent would be negative because we are dealing with 0.5, a number less than 1, and we had to move the decimal point to the right-hand side. As a result, our scientific notation of 0.5 would be *5 * 10^-1*. Here are a few more examples of large and small numbers as expressed in the scientific notation:

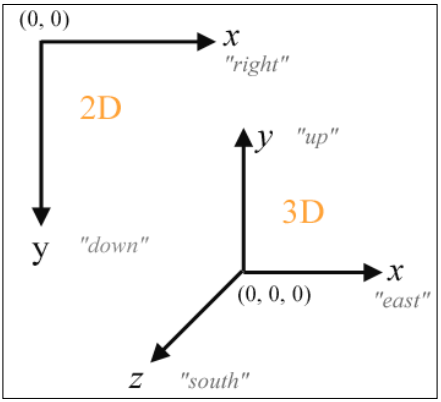
- 642,300,544,000 – *6.42300544 * 10^11*
- .00002055 – *2.055 * 10^-5*
- 8,549,248.5004 – *8.549285004 * 10^6*
- .0125174 – *1.25174 * 10^-2*

The scientific notation – a section review

In this section, we briefly looked at the scientific notation and how it's used, along with providing examples of large and small numbers that are expressed using its scientific notation. Now that we have discussed the scientific notation, let's go ahead and move on to our next topic: 2D and 3D coordinate systems.

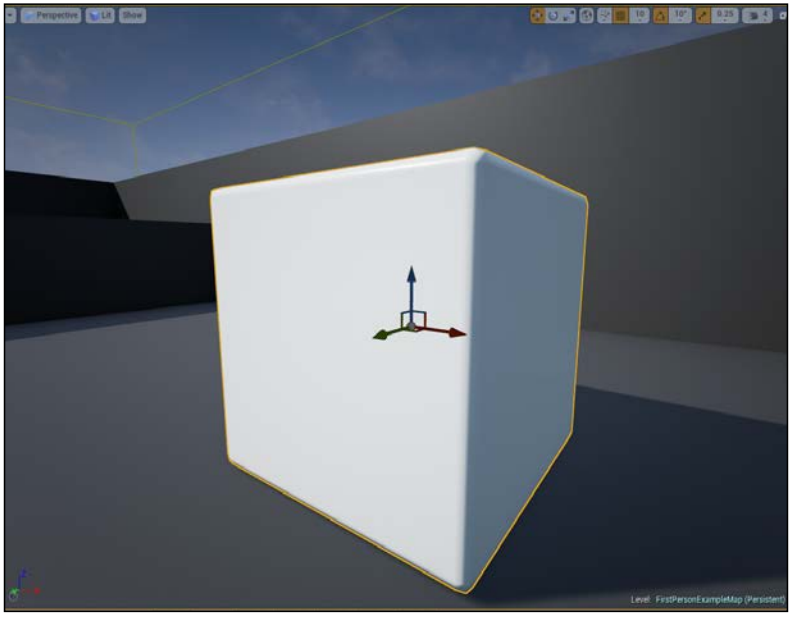
The 2D and 3D coordinate systems

In Unreal Engine 4, the use of 2D and 3D coordinate systems are used to determine the positions of actors in our game world. In a 2D coordinate system, we can determine an actor's position based on the *X* and *Y* axes, left-right, and up-down respectively. In a 3D coordinate system, along with the *X* and *Y* dimensions, we can determine the actor's position based on the *Z* axis: the inclusion of depth.



In Unreal Engine 4, the 3D axes are labeled differently, as displayed in the preceding image. Instead of the **"up"** axis being the **y** axis, in Unreal Engine 4, the **"up"** axis is labeled as the **z** axis. The **"forward"** axis is then the **y** axis instead of being the **z** axis, as depicted in the preceding image.

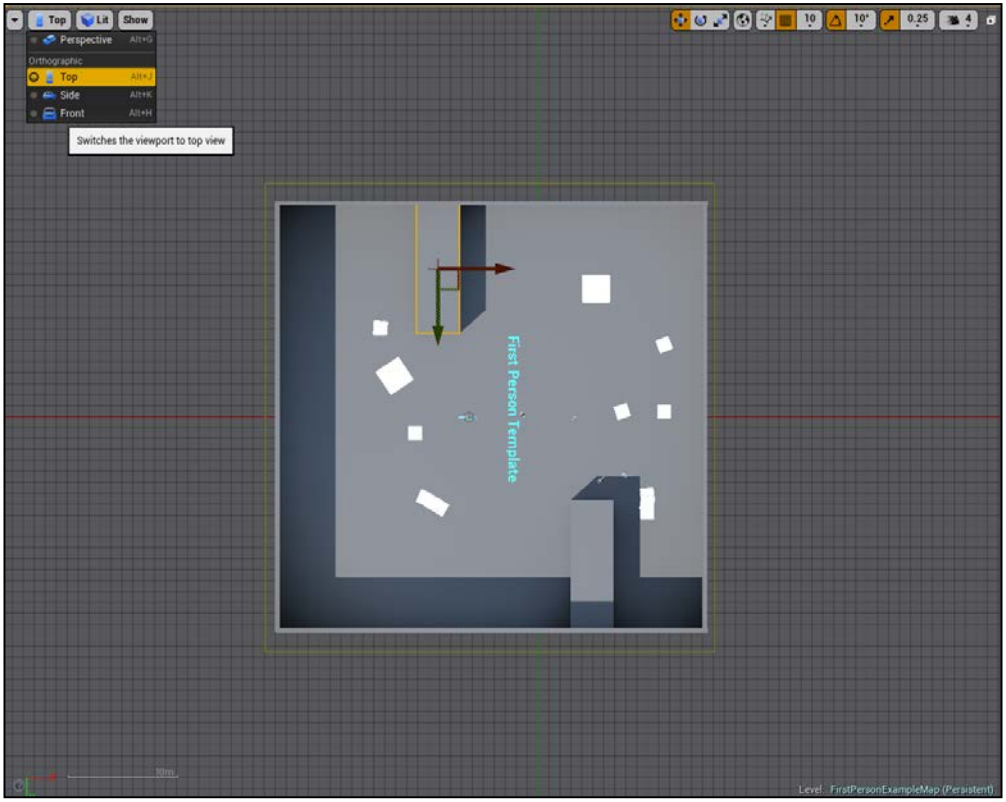
It should also be discussed that Unreal Engine 4 uses a left-handed coordinate system, which means that the positive direction for the *X* axis is on the right-hand side, the positive direction for the *Z* axis is upward, and the positive direction for the *Y* axis is forward. In the left-handed coordinate system, the positive rotation of an axis is always in the clockwise direction. We can see this reflected in the *transform* section of the details panel when an object is selected and is either moved or rotated.



In Unreal Engine 4, the *X* axis is labeled as a red-colored arrow; the *Y* axis is labeled as the green arrow, and the *Z* axis as the blue arrow. In the editor, you can toggle the transformation type of a selected object between translation, rotation, and scale by either repeatedly pressing on the spacebar or by toggling between the *W* (translation), *E* (rotation), and *R* (scale) keys. The viewport depicted in the preceding image is known as the perspective viewport. This is the only 3D viewport in Unreal Engine 4 and can be accessed using the *Alt* + *G* shortcut. When you work on a 3D game, Unreal Engine 4 offers three 2D viewports: the top, side, and front perspective viewports to take advantage of when you place objects in your game world.

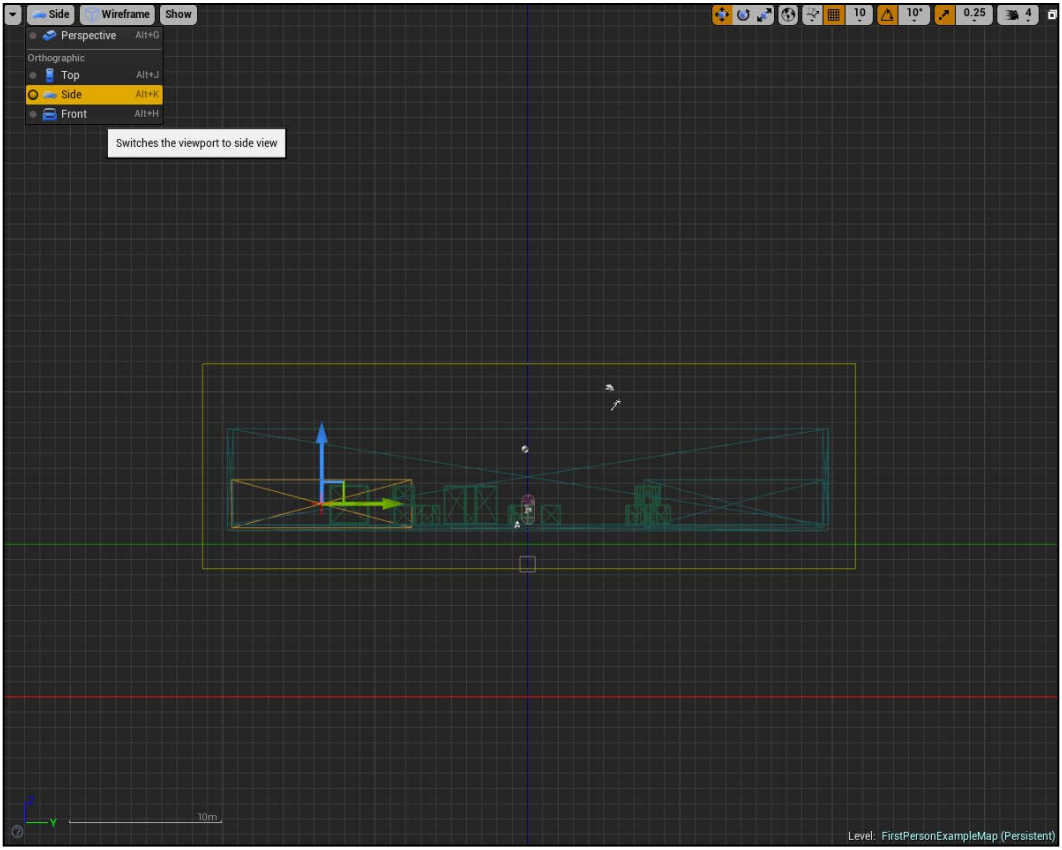
The top perspective

This perspective presents the 3D world from a top-down view (using the 2D coordinate system) with the *X* and *Y* axes that is similar to the previous image of the 2D coordinate system, where the *Y* axis represents up and down and the *X* axis represents left and right. This perspective can be accessed with the *Alt* + *J* shortcut or by clicking on the drop-down list labeled **Perspective** and selecting the **Top** option, as shown in the following screenshot:



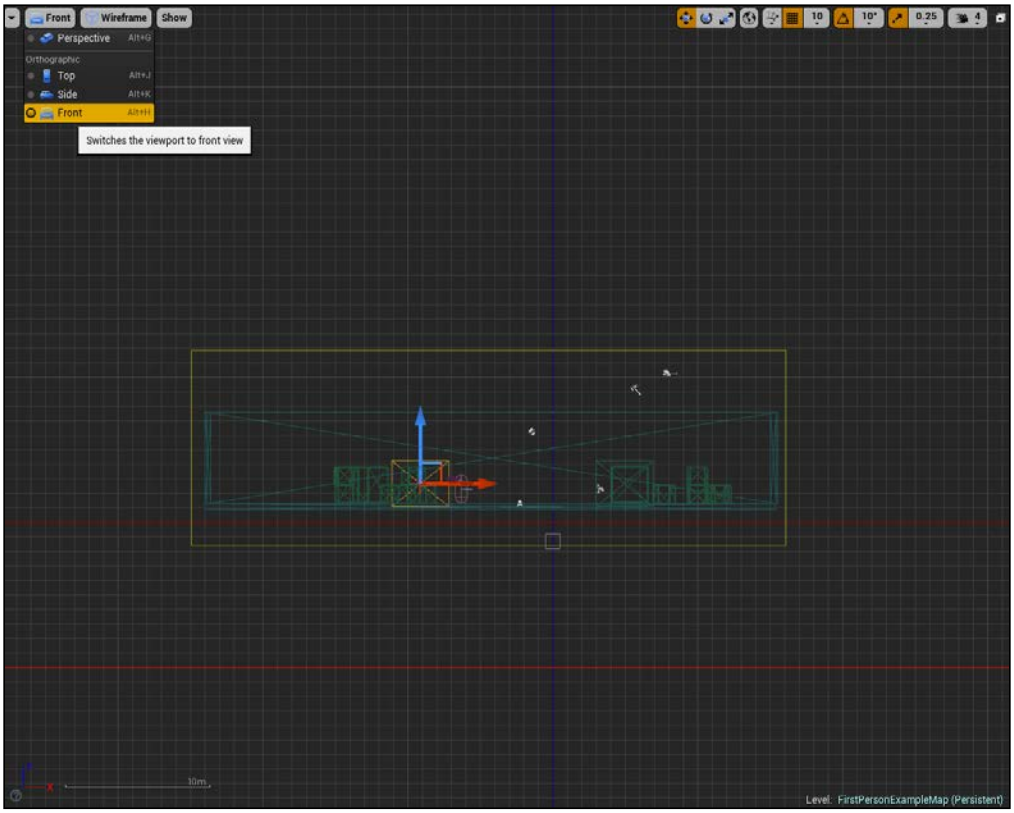
The side perspective

This perspective presents the 3D world from a side view perspective with the *Z* and *Y* axes using the 2D coordinate system. It effectively looks at the world from the left-hand side to the right-hand side. In this perspective, the *Z* axis represents up and down, whereas the *Y* axis represents left and right. This perspective can be accessed with the *Alt* + *K* shortcut or by the same method that is used to access the other three perspectives.



The front perspective

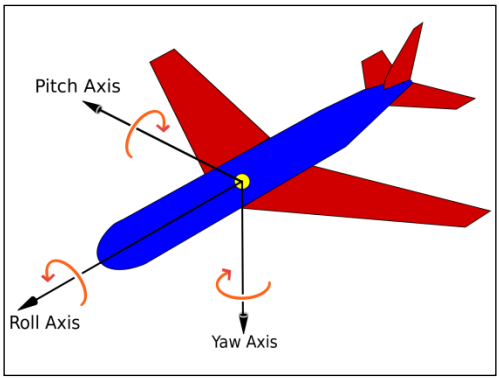
This perspective presents the 3D world from the front perspective with the *Z* and *X* axes using a coordinate system. It essentially views the world from the front to back side. In this perspective, the *Z* axis represents up and down, whereas the *X* axis represents left and right. This perspective can be accessed with the *Alt* + *H* short cut or by the same method that is used to access the other three perspectives.



Lastly, let's briefly discuss how rotation works in 3D programs, such as Unreal Engine 4. In the real world, the three different types of rotation of an object are Yaw, Pitch, and Roll. These rotations are defined as follows:

- **Pitch**: In Unreal Engine 4, this is defined as the rotation of an object about the *Y* axis.
- **Yaw**: In Unreal Engine 4, this is defined as the rotation of an object about the *Z* axis.
- **Roll**: In Unreal Engine 4, this is defined as the rotation of an object about the *X* axis.

In the real world, the pitch, yaw, and roll rotations of an object can be visualized by looking at how a plane can rotate, as shown in the following image:



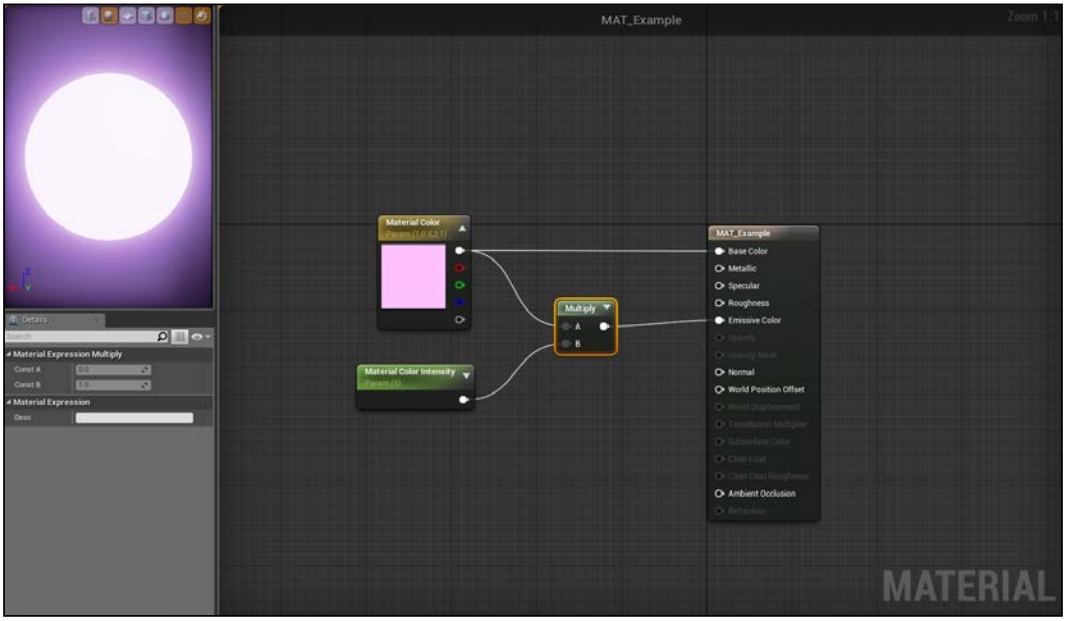
The 2D and 3D coordinate systems – a section review

In this section, we discussed the purposes of both 2D and 3D coordinate systems, their differences, and their uses in Unreal Engine 4. Furthermore, we took some time to go through the four perspectives offered in the engine: perspective, top, side, and front. Lastly, we looked at how to transform actors in our 3D world. With the concepts of 2D and 3D coordinate systems underneath our belt, you can now move forward and learn more about scalars and vectors.

Scalars and vectors

These are definitions to describe the motion of objects. Both are unique. In mathematics, scalars are defined as quantities that are described as a single numerical value, whereas vectors are defined as quantities that are described as a numerical value and direction. The examples of scalar quantities include length, area, volume, speed, mass, temperature, and power, whereas the examples of vector quantities include direction, velocity, force, acceleration, and displacement.

In Unreal Engine 4, the use of scalar and vector values is very common, especially in blueprints and materials. In the context of the material editor, scalar values are simply numerical values, whereas vectors are actually the colors of RGBA or red, green, blue, and alpha. In the **Material** editor, we can use both scalar and vector parameters to influence the color and intensity of the material itself.

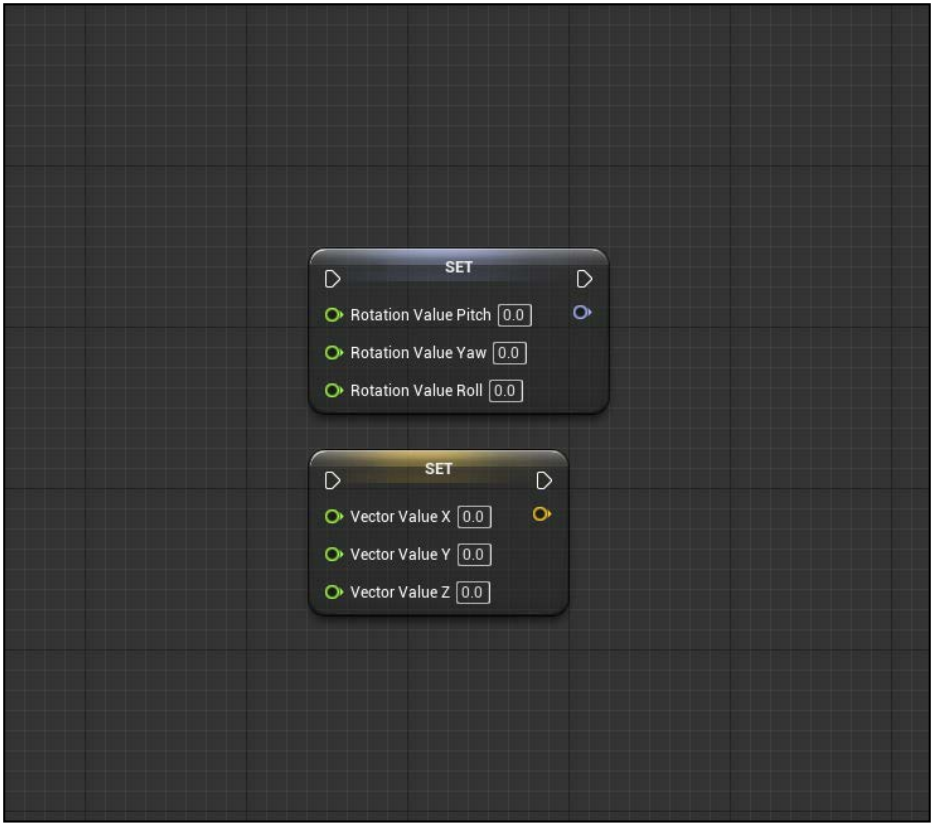


In the preceding screenshot, we are using a vector parameter node in our material to dictate the color of the material itself. By default, the vector parameter in the material editor contains the values for red, green, blue, and alpha; the alpha value controls the opacity of the color. In the material example, the scalar parameter controls the strength of the emissive value of the material. By increasing or decreasing this value, the material's brightness will get brighter or dimmer. To recreate this, we can right-click on our **Content Browser**, select **Material** and name this `MAT_Example`, and double-click on the material to open the **Material** editor. Perform the following steps:

1. Right-click on an empty space in the **Material** editor and search for the **Vector** parameter. Set its RGBA values to `1.0`, `0.5`, `2.0`, and `1.0` respectively. We can name this parameter as `Material Color`.
2. Next, let's right-click and search for the **Scalar** parameter. Then, set its numerical value to `5` and name this parameter as `Material Color Intensity`.

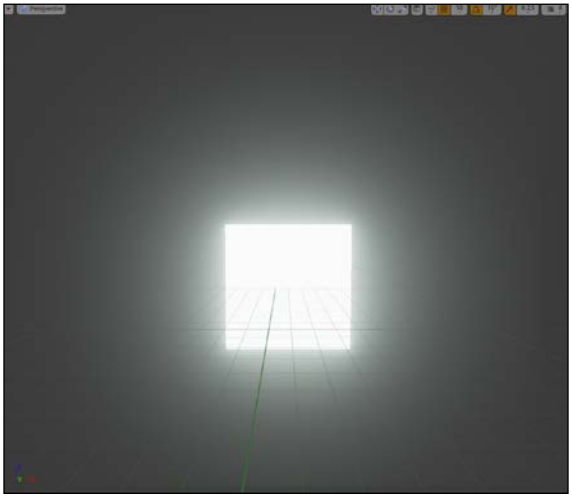
3. To create a multiply node, we can either right-click and search for this node, or just hold the *M* key and left-click on the blank space in the **Material** editor to create the multiply node.
4. Now, we can multiply the color output **Material Color** vector by the numerical output of the **Material Color Intensity** scalar and plug the result into the **Emissive Color** input of the material itself to create a bright and intensive purple material.
5. For additional color, we can plug the color output of the **Material Color** vector parameter into the **Base Color** input of the material as well.

In the Blueprints of Unreal Engine 4, the scalar and vector parameters serve similar purposes (as seen in the material editor). The vector variable in blueprint scripting holds the values for **X**, **Y**, and **Z** values and is used to dictate the location and direction, whereas the rotator variables holds the **Roll**, **Pitch**, and **Yaw** rotation values. When it comes to scalar variables in blueprints, there are many options to use (such as integers or floats) because scalar values are only numerical values with no direction associated to them.



As shown in the preceding image, we can split the structure pin for the rotator and vector variables by right-clicking on the vector values and selecting **Split Struct Pin**. So, we can edit each direction individually using float scalars to affect each. At the same time, instead of using individual scalar values by right-clicking on one of the split float values and selecting the **Recombine Struct Pin** option, we can also recombine the structure pin for these variables so that we can edit these values with vectors or rotators respectively.

Another interesting use of materials and blueprints is that you can dynamically change the value of the scalar and vector parameters in the event graph or the construction script of the blueprint.



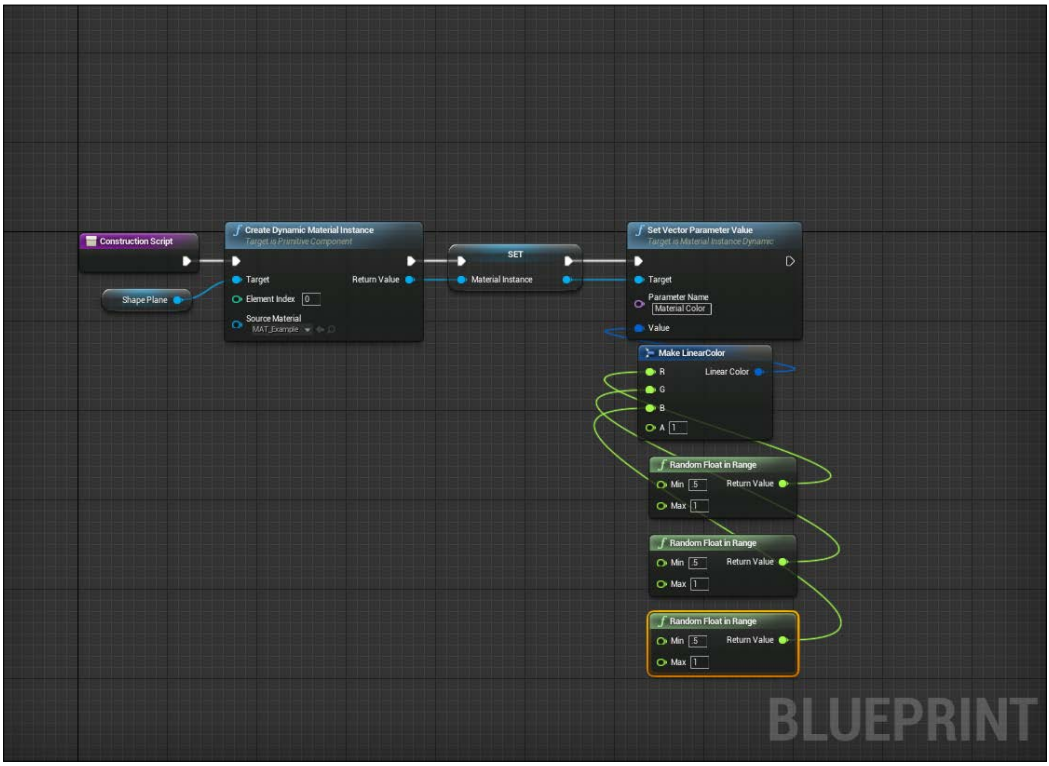
As shown in the preceding image, we can create a dynamic material instance from a static mesh in our blueprint that uses the material example we made earlier, which uses the vector and scalar parameters. Here, we can set the **Material Color** vector parameter, split the color structure into four unique float values of RGBA (red, green, blue, and alpha) and then use the random float in the range node to create random colors for the material.

To recreate this, we first need to create a new blueprint by right-clicking on our content browser and selecting the **Blueprint** class and then **Actor** to create an actor-based blueprint. Next, double-click on this new blueprint to open the **Blueprint** editor. Perform the following steps:

1. Select the **Viewport** tab at the top of the editor so that we can add our components for this example blueprint. For the base of this blueprint actor, we want to add a scene component to the root of the actor so that the other actors we add can be attached to this component.

2. From the **Add Component** tab, select the **Scene component** option and name it `ROOT`.
3. Now, we want to add the shape of a plane to our blueprint so that we can see our material on an object. Under the `StarterContent` folder in **Shapes**, select the **Shape_Plane Static** mesh so that it is highlighted in the content browser.
4. With the **Shape_Plane** mesh highlighted, let's go back to our blueprint. Under the **Add Component** tab, there will be an option for **Static Mesh** (**Shape_Plane**). Name this component whatever you like and rotate/orient the mesh in the 3D viewport as necessary.
5. Now, back in our **Content Browser**, let's select our material so that it is highlighted. Then, back in the blueprint, we can apply this material to our plane mesh by selecting the plane in the **Components** tab and clicking on the arrow next to the **Element 0** option in the **Materials** section.
6. With our material applied to our **Static Mesh**, we can now navigate to our **Construction Script** to script the behavior that will randomly change the color of this material each time the blueprint initializes.
7. In the **Construction Script** tab, let's grab the **Get** variable of our plane static mesh by keeping *CTRL* pressed and clicking and dragging the variable from the variables section to the left-hand side of the editor. From this variable pin, we can search for **Create Dynamic Material Instance**. Make sure that the material we created is selected for the **Source Material** variable in that node.
8. From the return value of the **Create Dynamic Material Instance** node, we can promote this value to a variable that we can then reference in the blueprint whenever we like. Name this variable whatever you like.
9. Now, we can drag the pin from the variable output of the newly promoted material instance variable and search for the **Set Vector Parameter** node. Here, we need to provide this node with the name of the vector parameter that we want to change and color values that we want to enter. If you remember, we named our vector parameter `Material Color`.
10. To randomize the color, we need to drag from the value input variable of the **Set Vector Parameter Value** node and search for `Make Linear Color`. For the RGB values, we can use **Random Float** in range nodes that have a minimum value of 0.5, a maximum value of 1.0, and a constant alpha value of 1 to randomize the color.

11. Now, when we repeatedly click on the **Compile** button at the top, we can see the color of the material change each time.



Scalars and vectors – a section review

In this section, we discussed the important differences between the values of vectors and scalars, their definitions in the case of real-world mathematical quantities and in the realm of Unreal Engine 4. In addition, we looked at some in-engine examples of how to use scalars and vectors in materials and blueprints. We also looked at how to incorporate scalars and vectors so that it can dynamically change the color of a material. With a base understanding of scalars and vectors in our pockets, we can now discuss Newton's laws of physics.

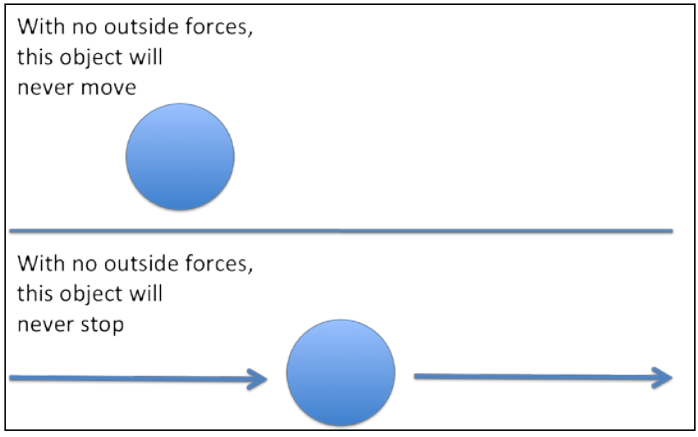
Newton's laws/Newtonian physics concepts

The base of all that we know about real-world physics comes from the principles developed by *Sir Isaac Newton*, also known as Newton's three laws of motion. When we recreate real-world physics in video games, it is very important that we understand these laws and how they affect objects in our game world. Keep in mind that not all games use realistic physics, but these laws of motion are still important to grasp when you develop any game world.

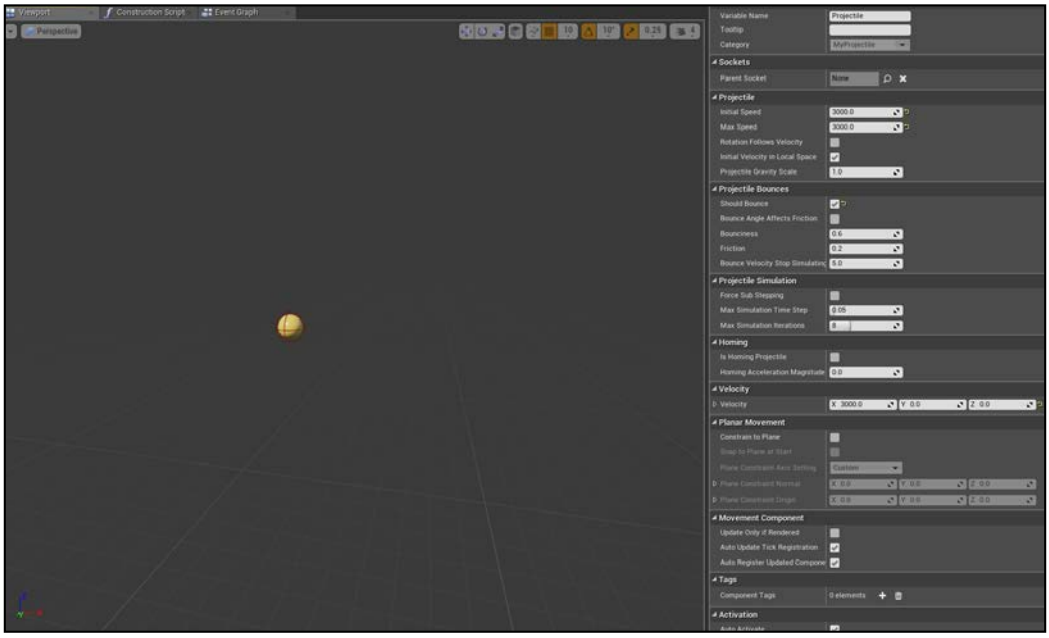
Newton's first law of motion

Isaac Newton's first law of motion, also known as the **Law of Inertia**, states that *every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it*. In other words, an object in motion tends to stay in motion unless acted upon by another force.

In the real world, there are external forces (such as ground and air friction) that act on objects in motion that eventually cause this object to stop completely, or forces such as a person pushing or pulling on an object that can cause acceleration, or for the move to increase in speed over time. In a vacuum, there is no friction. As a result, an object in motion in an infinite vacuum space would continue to move at the same rate unless it is acted on by some external force.



In Unreal Engine 4, the blueprint assets that utilize the Projectile component, such as bullets, rockets, or any other kind of projectiles used in our game, can edit the coefficient of friction and other physics-based properties. From the first person project that was created earlier in the chapter, we can navigate to the `Content` folder in the **Content Browser** and then to the `FirstPersonBP` folder and select the `Blueprints` folder. In this folder, we can select the **FirstPersonProjectile** blueprint. Then, in the **Viewport**, we can select the **Projectile** component to view some of the physics properties, as shown in the following screenshot:

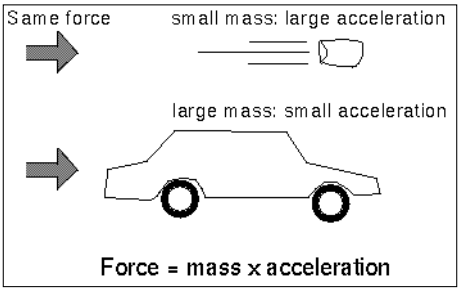


By increasing the **Friction** property, we can cause this projectile to come to a stop more quickly, whereas decreasing this property will result in the projectile coming to a stop over a longer period of time. We can alter this property until we can get the behavior we want. For more examples of physics-based properties featured in the Unreal Engine 4 blueprints, feel free to investigate the **FirstPersonCharacter** blueprint and select the **CharacterMovement** component.

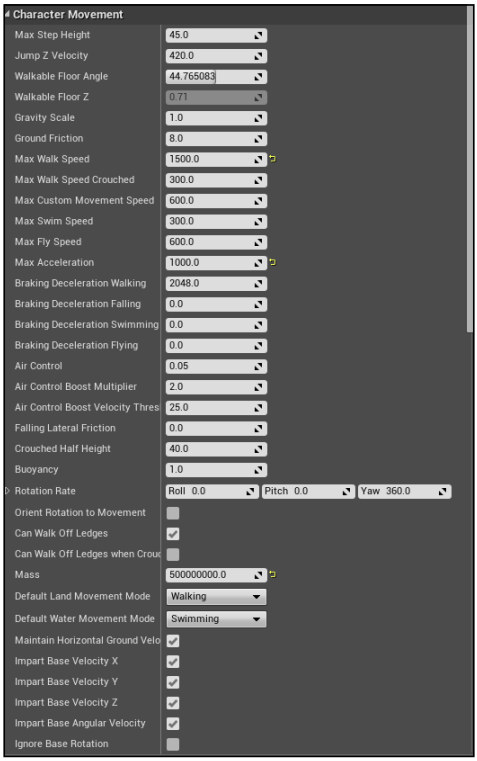
Newton's second law of motion

This states that *the relationship between an object's mass (m), its acceleration (a), and the applied force (F) is F = ma or an applied force is equivalent to the mass of the object and its applied acceleration*. Acceleration and force are vectors, (remember that a vector is both a numerical value and a direction). In this law, the directional force vector is the same as the direction of the acceleration vector.

In more simple words, this law focuses on the principle that a change in an object's velocity can only occur if this object is accelerating in a particular direction, and a positive or negative acceleration can only take place if an external force is acting on it.

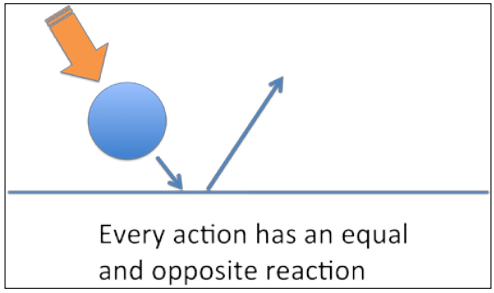


In Unreal Engine 4, we can use blueprints to apply forces to physics objects and override properties (such as acceleration and mass) through different available components. For example, the **CharacterMovement** component in the **FirstPersonCharacter** blueprint has a property labeled as **Max Acceleration**, and if we increase or decrease this property, we can see how quickly the player accelerates from a stationary position to its maximum walk speed.



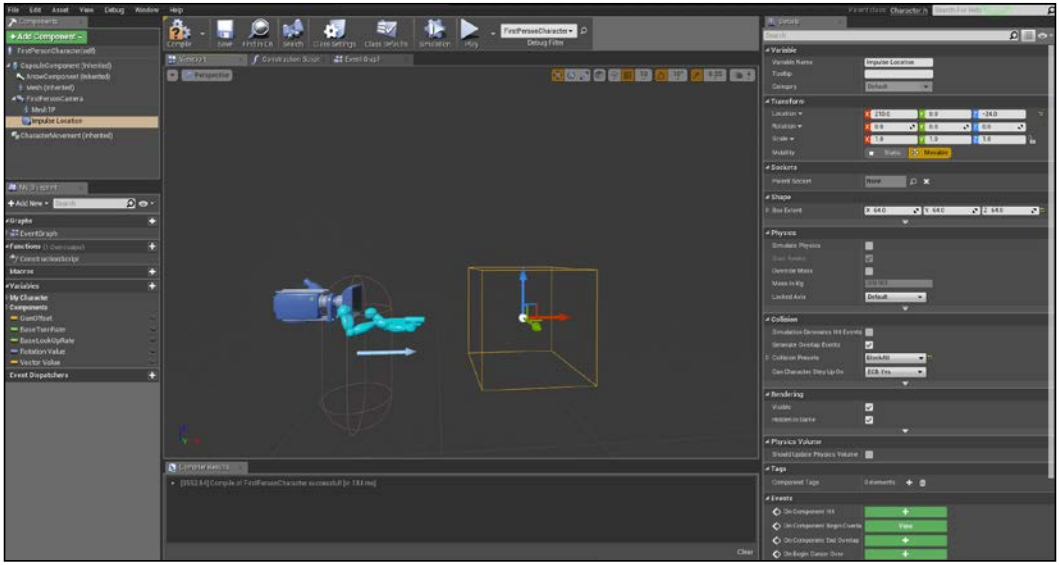
Newton's third law of motion

Isaac Newton's third law of motion and one of the more commonly known law states that *for every action there is an equal and opposite reaction*. In other words, when an object applies force to another object, an equal and opposite force is applied as well.



In Unreal Engine 4, we can see this law of motion in action by playing in the editor and clicking on the left mouse button while aiming at the ground to see the ball bounce in the opposite direction that it was fired at and moving at a speed equal to the one in which it was fired at, except that we have friction applied to this projectile, so it loses some of its initial velocity due to the friction.

To bring all of these laws of motion together, what we can do is add a box collision component to the **FirstPersonCharacter** blueprint, set its **Collision Presets** to **BlockAll**, and attach it to the **FirstPersonCamera** component by dragging it onto the **FirstPersonCamera** component in the **Components** tab.



Now if we play in the editor, we can start running into the physics cubes in the **FirstPersonExampleMap** (which is default to the **First Person** project template) and see forces applied to them, which is equal to the mass of the player multiplied by the players' current acceleration value.

Newton's laws of motion – a section review

In this section, we discussed the three laws of motion developed by *Sir Isaac Newton* and their application in the real world and in Unreal Engine 4. For each of these three laws, we examined each by providing working examples of each, using blueprints in Unreal Engine 4. Now that we have a strong understanding of these principles, we can tackle the last subject of this chapter: forces and energy.

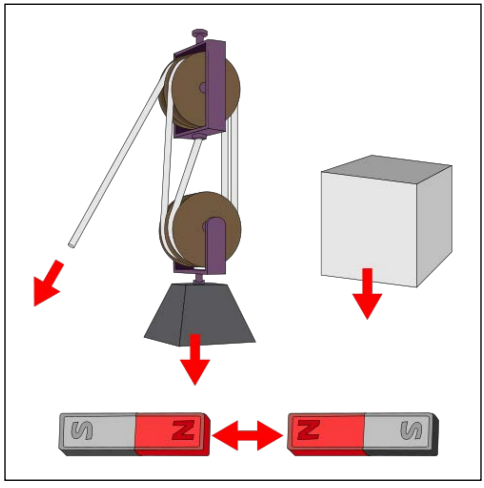
Forces and energy

One of the most important concepts regarding energy is that it's a property of objects that can be transferred from one object to another, but it cannot be created or destroyed. All forms of energy follow the conservation of energy aspect and can be converted to different types of energy. There are many types of energy that exist in the real world. Here are a few examples:

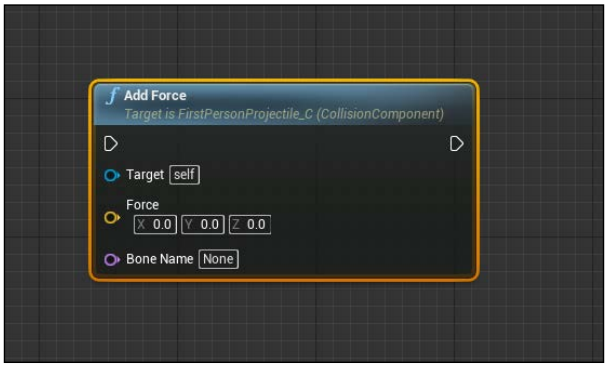
- **Kinetic Energy**: This specifies the motion of a moving body
- **Potential Energy**: This denotes the energy that an object has due to its location in the 3D space
- **Mechanical Energy**: This specifies the sum of kinetic and potential energies
- **Heat**: This denotes the amount of thermal energy being transferred in the direction of decreasing temperature

These are just a few examples of energy that exist in the real world, so feel free to perform additional research on the concept of energy because this section will only cover the surface of the topic. When it comes to the concept of energy and the conservation of energy, Unreal Engine 4 follows these properties as well through its built-in physics engine.

As discussed in the previous section of this chapter regarding the Newtonian principles, forces are any interaction that tends to change the motion of an object. This can also be referred to as concepts (such as pushing and pulling) and contains a magnitude and direction, making it a vector quantity. Forces can be caused by gravity, magnetism, wind, or even the pushing or pulling of an object by a person or machine.

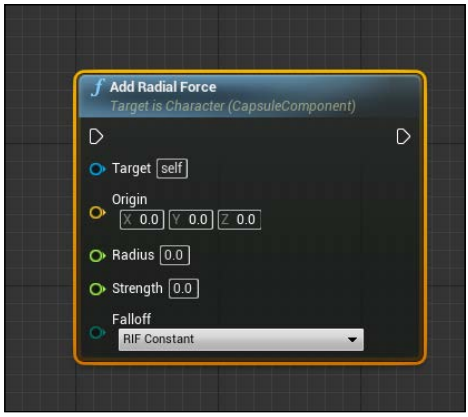


In Unreal Engine 4, we can add forces through blueprint scripting, and we can perform this in a few ways. The first method is a function called **Add Force**. This acts a lot like a thruster and adds a linear burst of energy in the specified direction.



You can add this type of force to any component associated with our blueprint, or you can apply this force to any component that is hit by a component of our blueprint. We can also see that there is a bone name property in this function. This means that we can apply force to a bone if one exists in our blueprint.

The second method of applying forces to our blueprint components or to components in our game world is through the **Add Radial Force** function:



This function allows you to specify a location in the 3D space. Here, the source of the radial force begins and then specifies a radius and strength using Float values. Lastly, we can apply one of the two methods of **Falloff** for the radius, which is either a constant fall-off or a linear fall-off. All bodies in this radius will be affected by this force, so make sure that you take this into consideration when you apply radius and strength values.

Forces and energy – a section review

In this section, we looked at forces and energy and how they are applied in the real-world and in Unreal Engine 4. Additionally, we investigated the different types of energy that exist in the real world. Furthermore, we looked at different examples on how to apply forces in blueprints with the **Add Force** and **Add Radial Force** functions. Lastly, we discussed the properties of these two functions and how they alter the force that is applied.

Summary

In this chapter, we discussed a handful of mathematical and physics-based concepts that are necessary to grasp in order to understand how physics works in Unreal Engine 4. We looked at the different units of measurement that exist in the American and European standards of length and how they convert from one to another. We also looked at **Unreal Units** (**uu**). Then, we discussed the common measurements for walls, doorways, characters, and stairs.

Next, you learned a little bit about scientific notation, how it works, and how it is used. We looked at some basic and advanced examples of conversions from numerical to scientific notations.

Additionally, you learned about the 2D and 3D coordinate systems and how they are used in the real world and in Unreal Engine 4. We also investigated the different 2D and 3D viewports that exist in Unreal Engine 4 and the important functions they serve when you create game worlds.

Furthermore, you learned about the scalar and vector properties and how they are applied in the real world and in Unreal Engine 4. We also looked at examples of how the values of scalars and vectors are used in the **Material** editor and in blueprints.

We also looked at each of the three Newtonian laws of motion and provided real-world and Unreal Engine 4 examples for each law.

Lastly, you learned about the different forces and energy that exist in the real-world and provided examples on how to apply forces in blueprints.

Now that we have a base understanding of real-world mathematics and physics concepts and how they are used in Unreal Engine 4, we can now move on to the **Physics Asset Tool** (**PhAT**) in Unreal Engine 4.

2
Physics Asset Tool

PhAT stands for Physics Asset Tool. Imagine you drop a dice. Based on the reality of the physical rules in the world we live, the dice drops differently on wood, stone, glass, and carpet. The same can be said for glass, plastic ball, and enemy body (more complex). Using PhAT, you will learn how to simulate reality based on the physical rules that exist in the game world. This is kind of animated, but with more logics and tools.

In this chapter, you will first learn about the most important items in PhAT and then use them in a practical example.

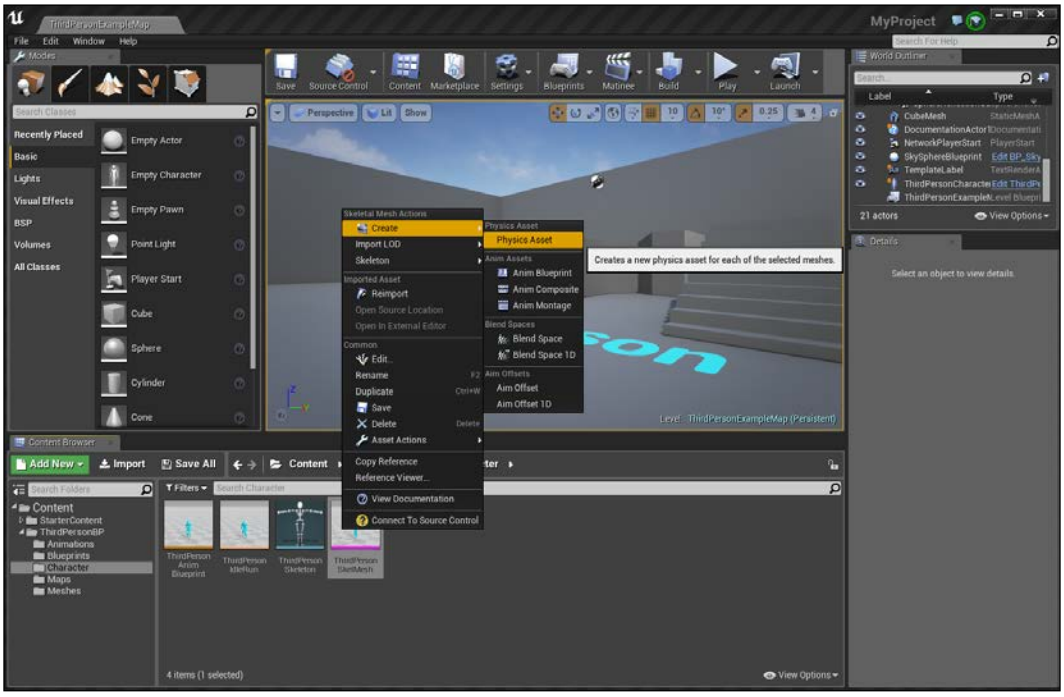
Navigating to PhAT

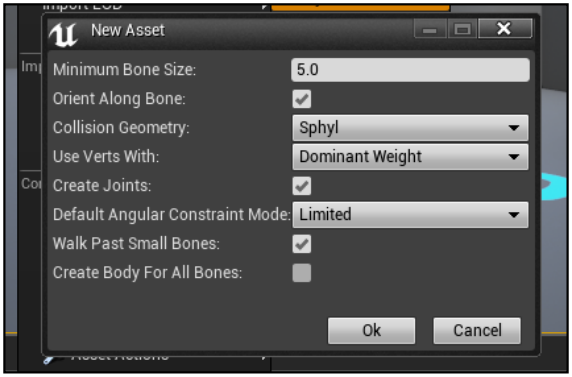
Before we start working in **Unreal Editor**, we will need to have a project to work with. Perform the following steps:

1. First, open **Unreal Editor** by clicking on the **Launch** button from Unreal Engine Launcher.
2. Start a new project from the **Project** browser by selecting the **New Project** tab. Then, select **Third Person** and make sure that **With Starter Content** is selected. Give the project a name (`phat_test`). Once you are finished, click on **Create Project**.

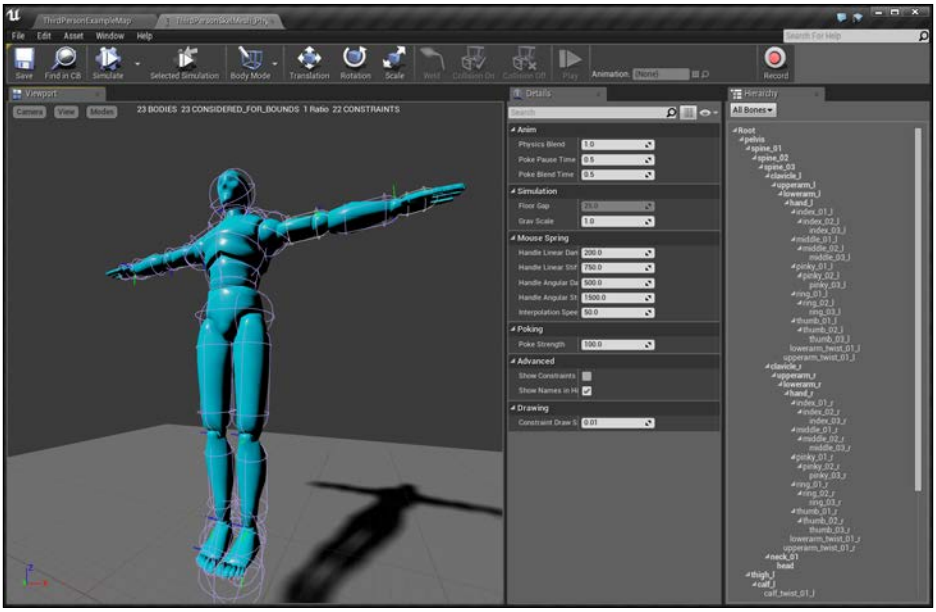
After everything comes up on your screen, in **Content Browser**, locate the **ThirdPersonBP** folder and click on the **Character** folder. Find **ThirdPersonSkelMesh** and then right-click on the list and select **Create | Physics Asset**, as shown in the following screenshot. Then, click on **Ok** in the **New Asset** window.

ThirdPersonSkelMesh can be found by the name **SK_Mannequin** in the newer version of Unreal Engine 4.

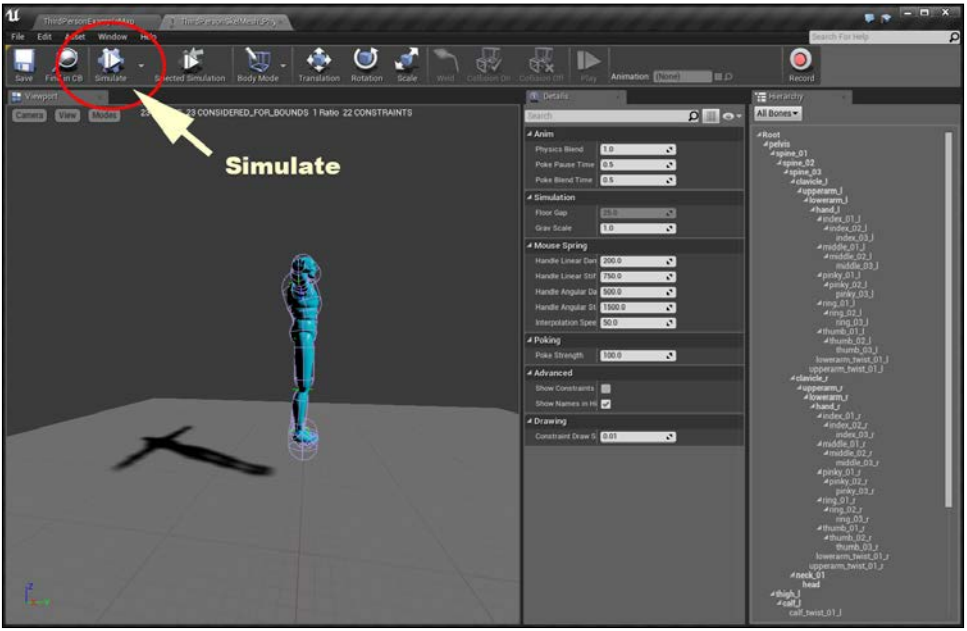




Now, if everything works fine, you will find your selected mesh in **Physics Asset Tool** or **PhAT**. This kind of editor allows you to put some custom controllers on parts of your character's mesh based on your bones. These controllers are sensitive to physical aspects, such as world gravity, movements of other parts, rotations on each part, and collisions between parts.

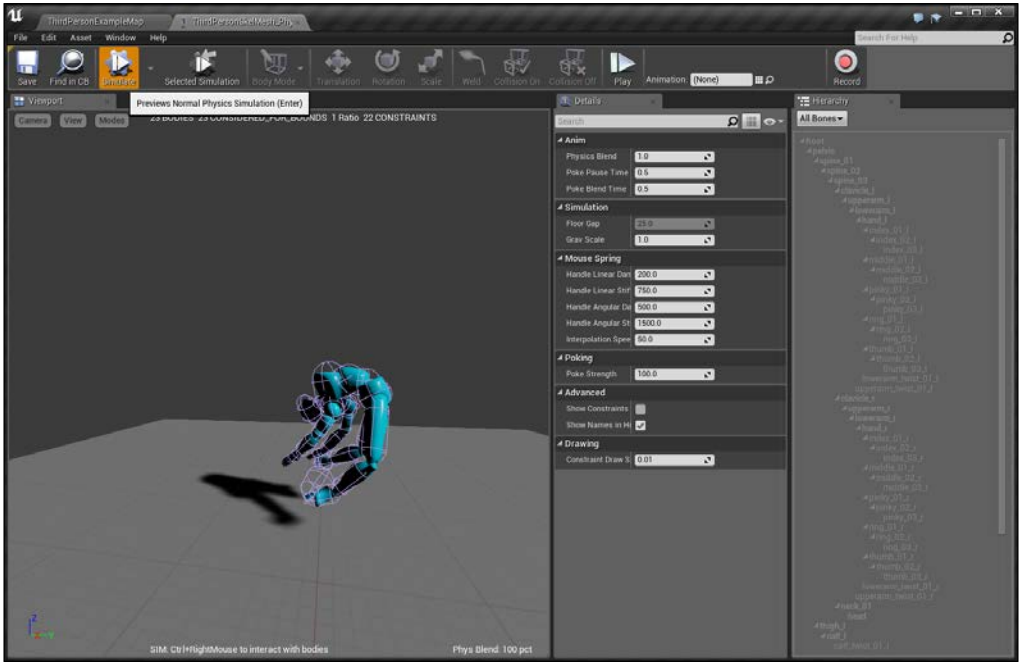


We will select a human like mesh. Imagine that this is an enemy and a player should shoot this object. After shooting, on hit, the enemy character falls on the surface. For simulating this scenario, set your camera as shown in the following screenshot, and click on **Simulate** from the top menu:



Then, set your camera to this view and click on **Simulate**.

In the preceding screenshot, you can see that the character falls, but the way it falls is not natural in many ways. The following image is a screenshot of this action. Here, you can see that the hands and backbones are in an unreal position. PhAT can solve all these problems by customizing each bone movement based on physical rules (such as gravity).



The problem persists on the back and hands as the mesh falls.

Click on **Simulate** again to go back to the normal mode.

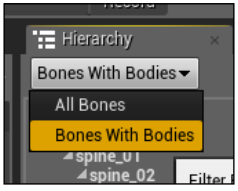
The PhAT environment

PhAT has a couple of gadgets and sections. Here, we will cover most of the commonly used sections:

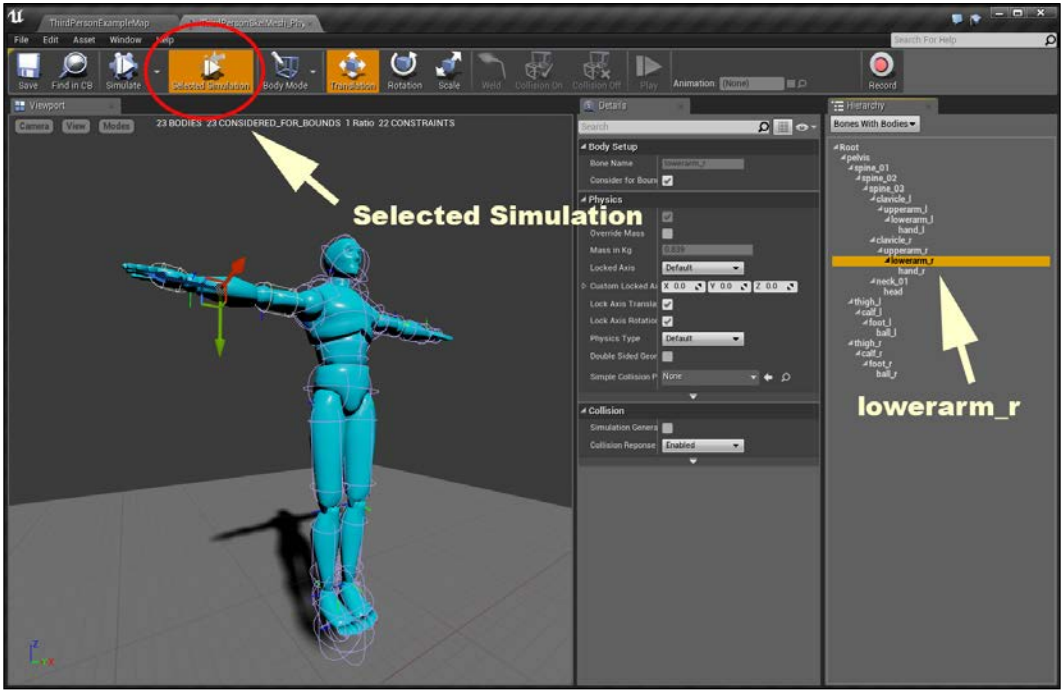
- **Hierarchy**: On the right-hand side of the stage, there is a list of bones entangled with meshes on the character. This is used for animation purposes. Some bones have a bold font, whereas some don't. When you assign a physical asset to a bone, Unreal Engine automatically makes it bold. This is useful for addressing the bones involved with a glance over them. You can turn it on/off using the **Windows** button from the top menu.

- **Save**: This saves the current asset on the model.
- **Find in CB**: This locates the current asset in **Content Browser**.
- **Simulate**: This runs the simulation.
- Little arrow in front of **Simulate**: This changes the simulation mode between the **Real** and **No Gravity** types of simulation.
- **Selected Simulation**: When you select one bone and then hit simulate, the only selected bone and its branch (or children) will react to simulation. Let's take a look at an example:

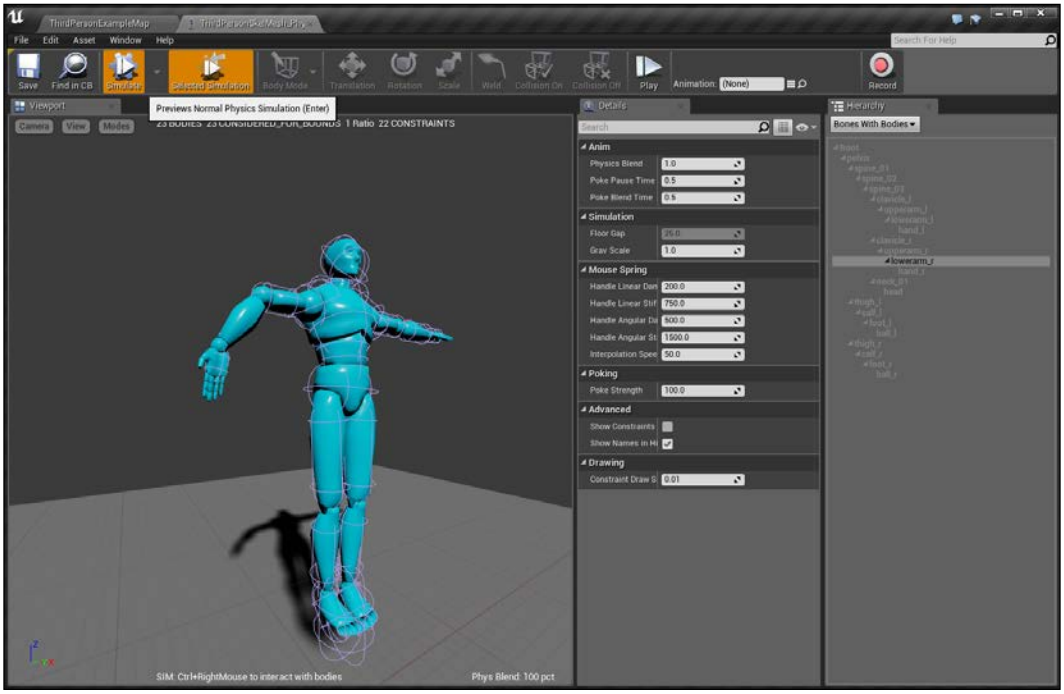
    1. Navigate to **Hierarchy** on the right-hand side and select **Bones with Bodies**. This just shows the bones with assets.



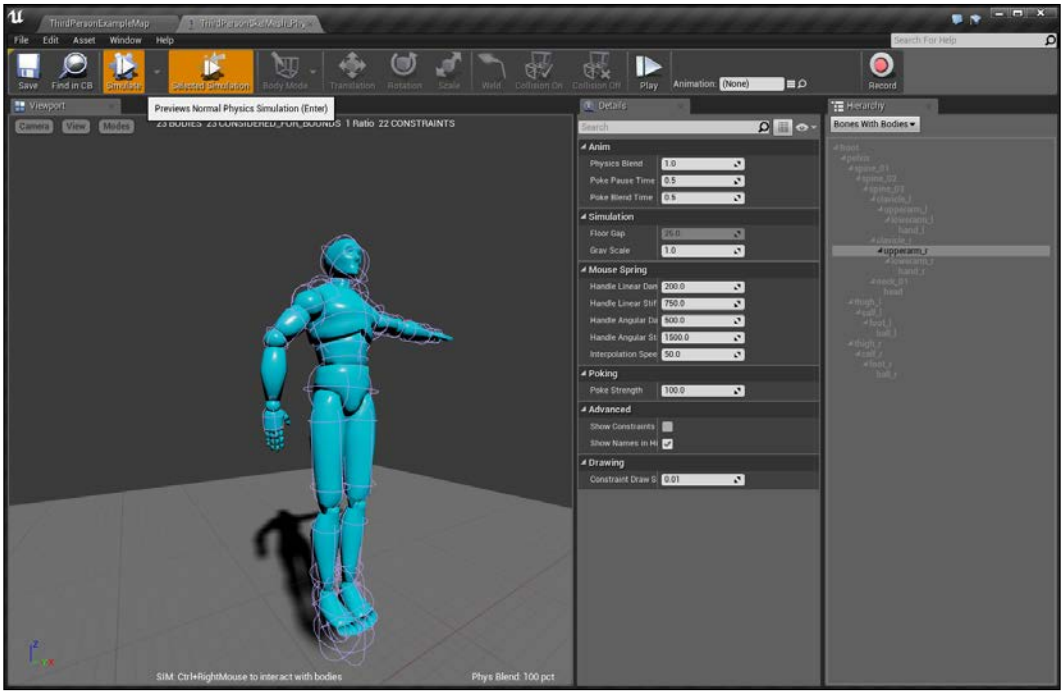
    2. Now, right-click on **lowerarm_r** in the bones list and then on **Selected Simulation** to make it active:



3. Then, click on **Simulate**. As you can see, only a part of the left hand shows some movement, but the rest of the body does not move:



4. Click again on **Simulate** to go back to the normal scene.
5. Now, click on **upperarm_r** in **Hierarechy**. Then, click on **Selected Simulation** again. As you can see, now the entire hand is part of the simulation, but the body is not moving:



6. Later when we start adding assets, this simulation type will be useful and handy. For now, just try experimenting with more bones and simulate your selection.

- **Body Mode**: This selects the different types of visual presentation on the body.
- **Translation**, **Rotate**, and **Scale**: These are tools for editing each asset.
- **Details**: In this section, you will find some related properties for each asset. When you select different assets in **Hierarchy**, the properties related to the selected asset are displayed here. You can turn this on/off using the **Window** button from the top menu:

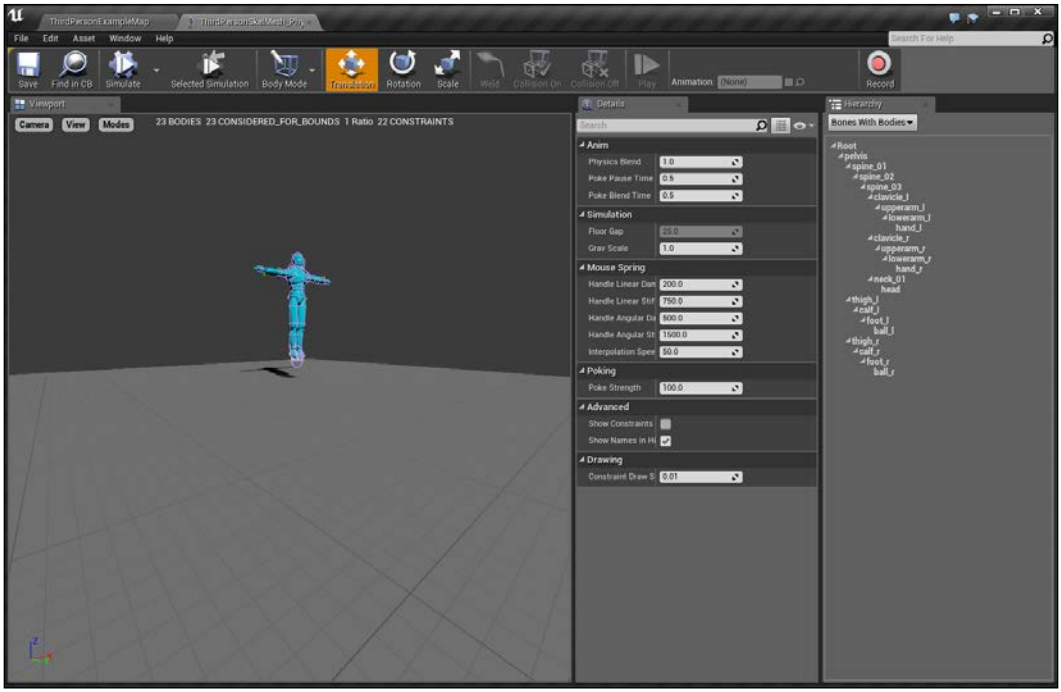
The PhAT example and experience

In order to create a new asset on each bone, it's better to clean our object from the previous ones. There is a quick method to perform this, that is, click on **Edit** from the top menu bar and then on **Select All Objects**. Now, click on *Delete* on your keyboard. Try this and then click on **Undo** from the same menu. You can also try another useful method, which is explained in the next section.

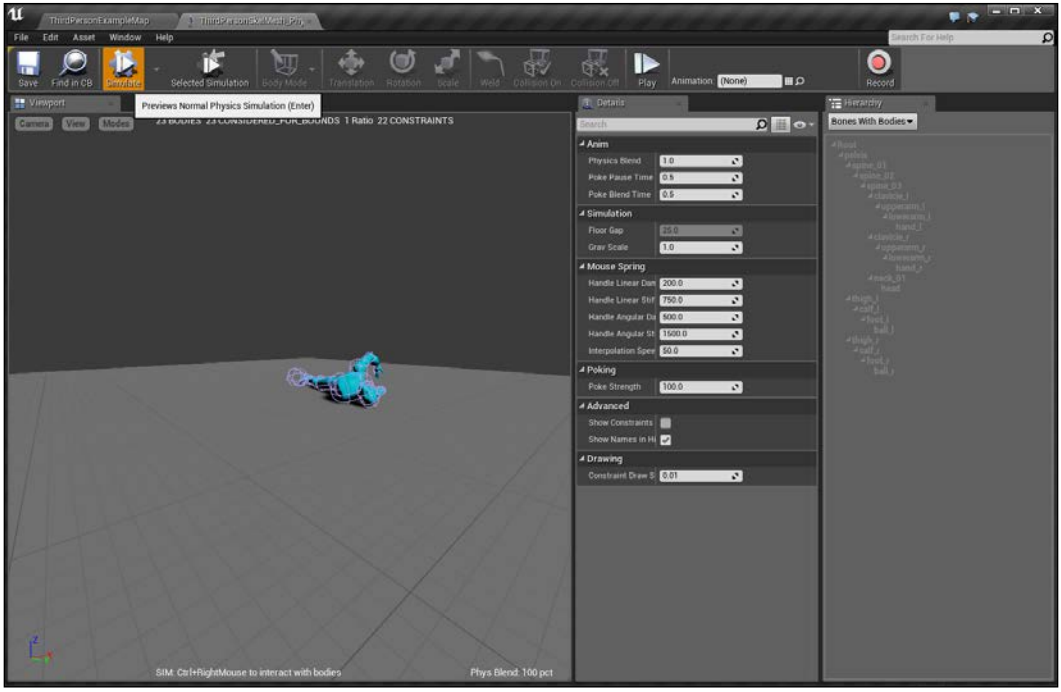
Deleting current assets

Perform the following steps to delete current assets:

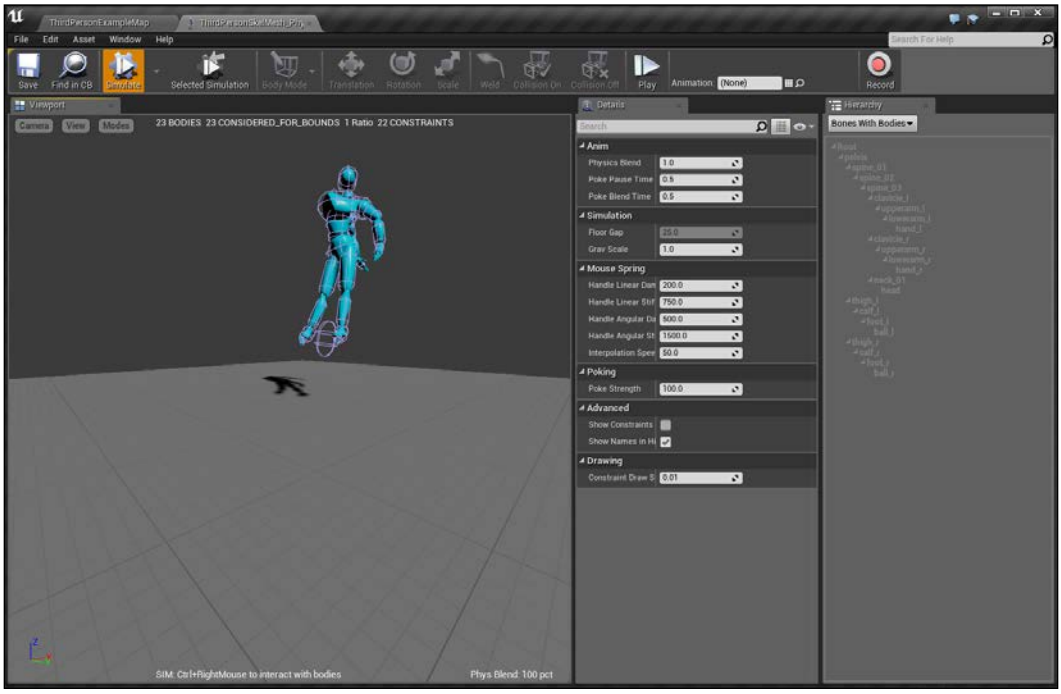
1. Set your camera as shown in the following screensahot:



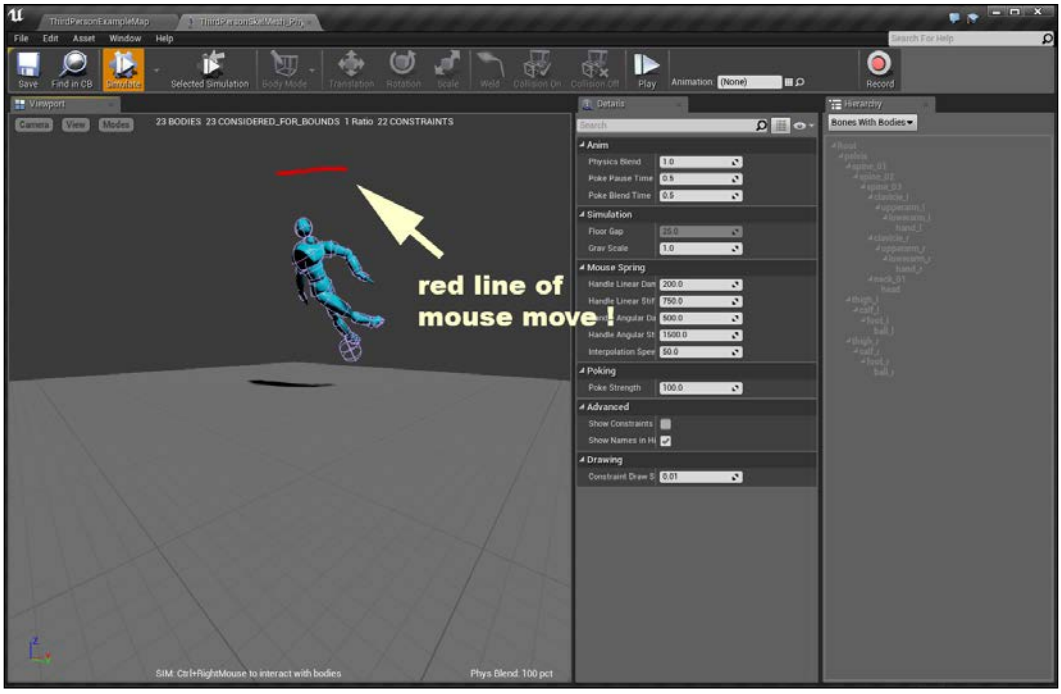
2. Now, click on **Simulate** and wait until the body fully lies down on the ground, as shown in the following screenshot. It is possible that your result would be a bit different, but no problem. You can try and click on **Simulate** to reach the closest form of the image:



3. Now, press *Ctrl* and right-click to hold the head and move your mouse up not fast, but not slow either. It looks like the mesh is hanging to your mouse pointer, as shown in the following screenshot:



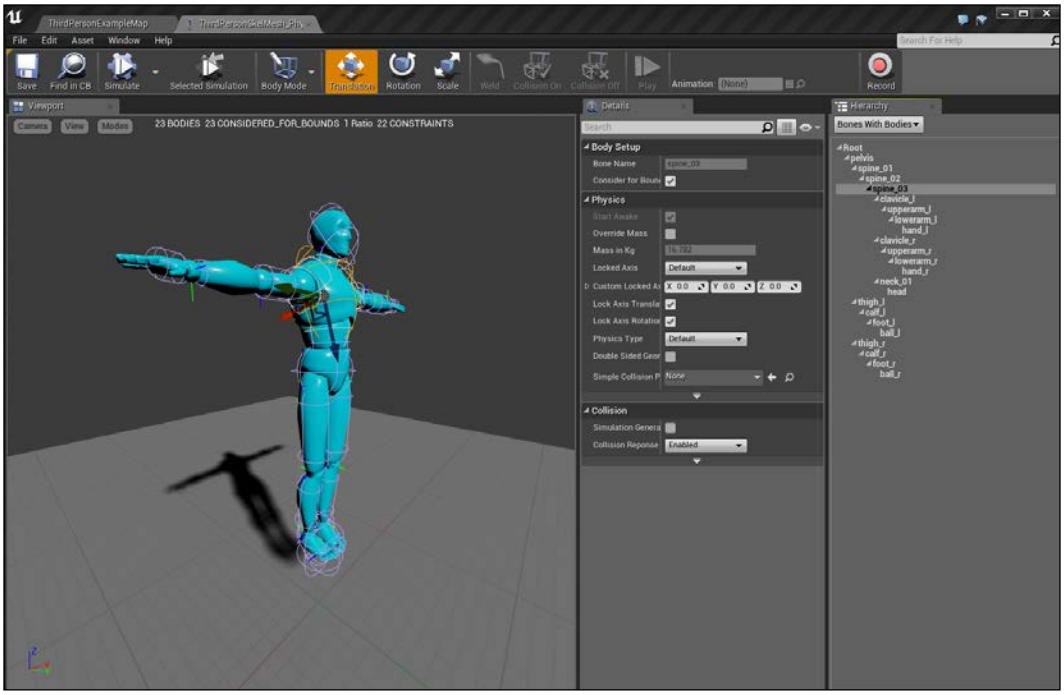
Without releasing the right mouse button, move your mouse in the left and right direction over the stage and watch the elements of the mesh connecting and moving with each other. You will soon discover a red line that indicates the path, in which your mouse just moved, as shown in the following screenshot:



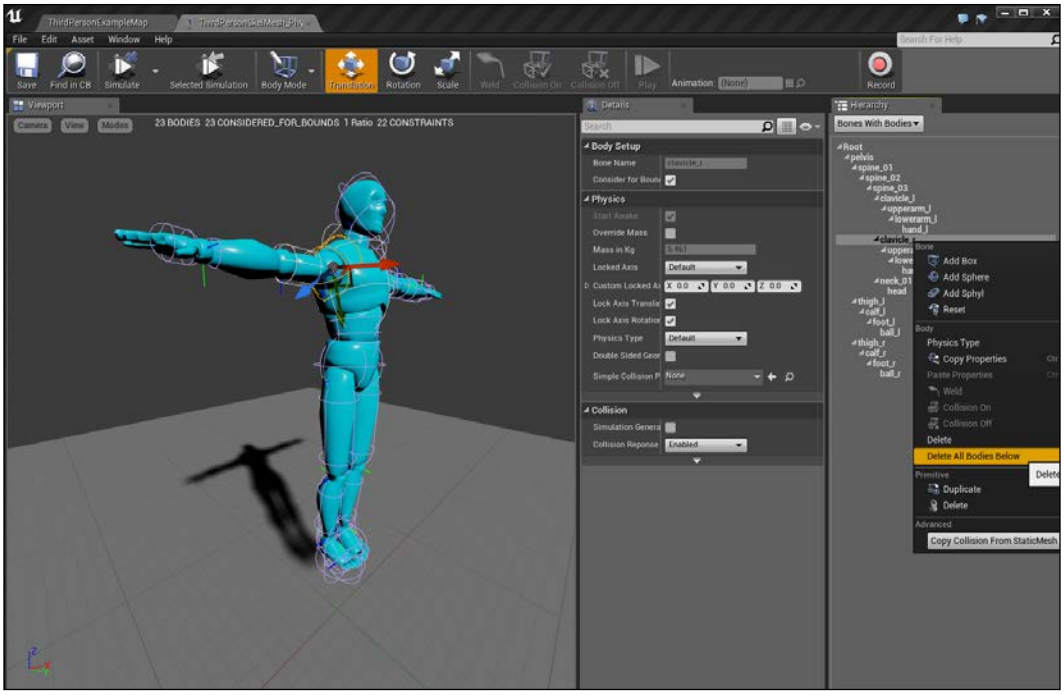
If you release the mouse button, you can use **Simulate** to switch back to the normal mode and create it back again.

4. Now leave the mouse, and the character will fall on the surface. Grab all the other parts of the character and experience the same movement with different mouse speed. This is kind of fun with PhAT and also shows how the body is connected with bones. You can track this in **Hierarchy**.

5. Click on **Simulate** to go back to the normal mode and right click on **spine_03** in **Hierarchy**. As you can see in the following screenshot, this part is selected in a different color. Now, click on **Simulate**. Then, grab this part (*Ctrl*, right-click and hold) and move it. Select **Simulate** again to return to normal. This time, select **foot_r** and try to simulate and grab it and move the character. As you can see, when you grab the body and move, other parts create a kind of rhythm, which depends on your mouse movement on the screen. This is caused by the default physic assets of each part.



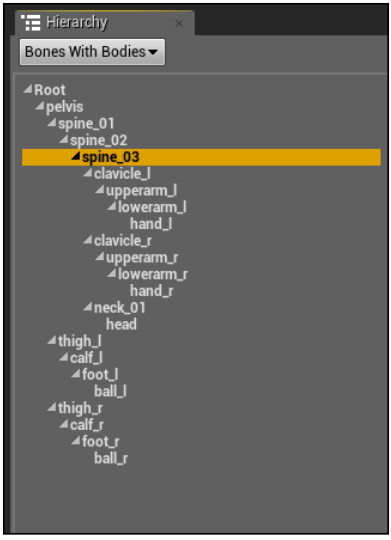
6. Now, right-click on **clavicle_r** and select **Delete All Bodies Below**, as shown in the following screenshot. It looks like nothing seriously happened here, but when you click on simulate, boom! The left arm remains still, and a totally different behavior is displayed. Stop the simulation, right-click on **clavicle_l**, select **Delete All Bodies Below**, and click on **Simulate** again. Now, both hands appear still, and it doesn't behave like before. Select **Undo** to track the difference between the previous and current behavior.



This practice is essential to understand what PhAT can provide us in the game. When you select **Delete All Bodies Below**, you can delete all the previous physic assets on the bones that will connect to your selected bone. You can track this in **Hierarchy**. For example, **clavicle_r** is connected to **upperarm_r**, same logic (image it as chain of bones), **upperarm_r** is connected to **lowerarm_r**. And, **lowerarm_r** is connected to hand_r. Here, hand_r is last member of these series of the bones. (this is important for the last bone). If you apply any physical rule to **clavicle_r**, the energy flows through the other bones until the end: **hand_r**. When you select **Delete All Bodies Below**, you can actually delete this path to navigate energy. This is why the hands look solid in simulation.

This is a basic understanding of what PhAT is used for. You will learn how to design certain assets that guide and reflect the physical input from the environment (such as gravity or collision) through the bones of the character.

To see how these assets will function and navigate energy, select **top bone to down bone (or bones)** in **Hierarchy** and check whether **clavicle_r** is connected to **spine_03**. The bigger bone sends/navigates the physic assets/ energy to **clavicle_r**.

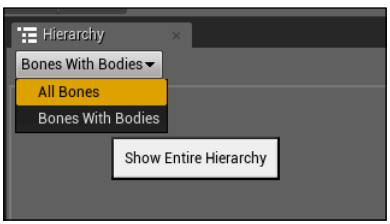


7. Try the same with **thigh_l** and **thigh_r**. Here, **pelvis** is the higher bone in **Hierarchy**.

Adding and customizing current assets

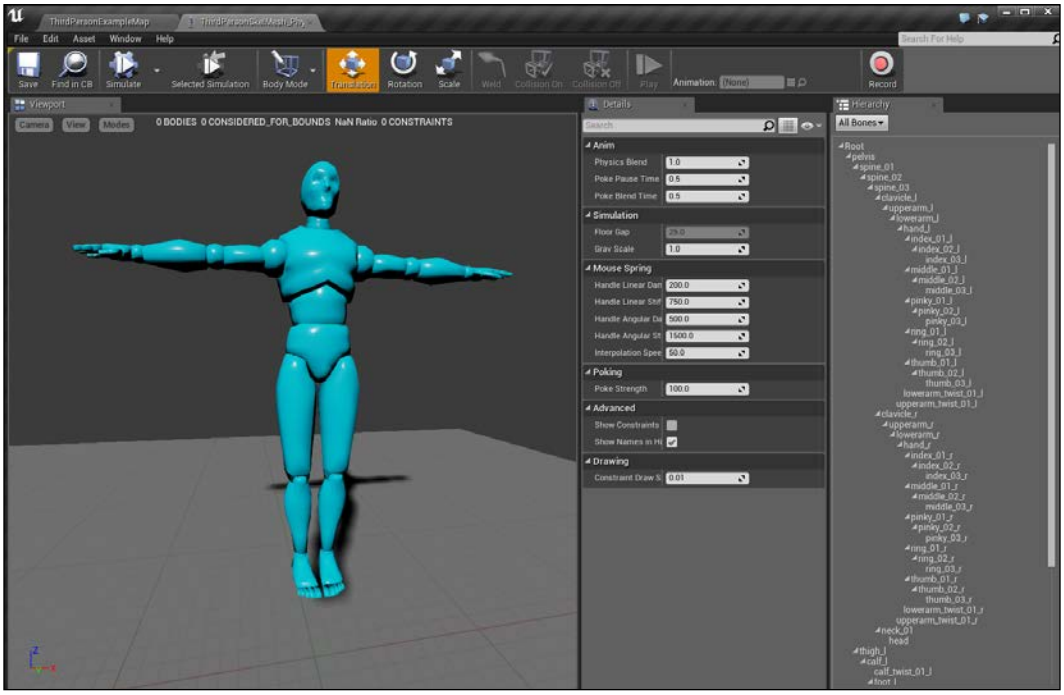
Now, as we experience the effect of PhAT assets on the bones and also have an understanding of how bones are connected and guide the energy, it's time to create our own physics assets over the body. Refer to the last part, try to delete the different bones asset, and simulate the scene. Then, select **Undo** and **Redo**. This is a creative practice for game designers.

Now, click on **Edit** from the top menu bar, select **All Objects**, and press *Delete* on your keyboard. Now, we can delete all the physic assets on the body. Then, select **All Bones** in **Hierarchy**:



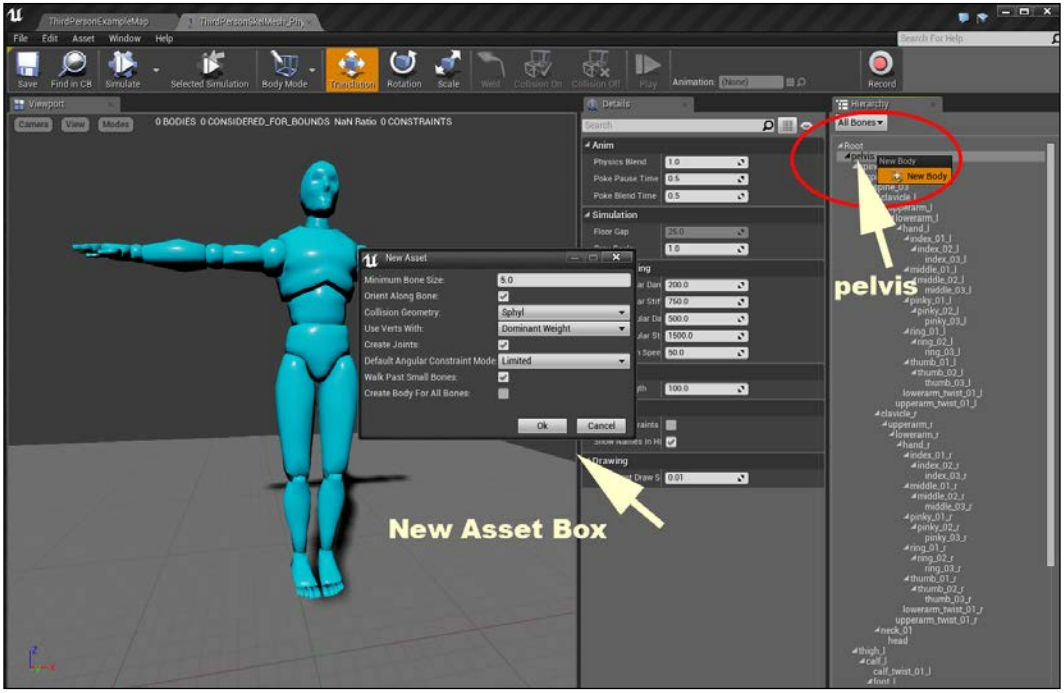
You will find a detailed list of all the bones in **Hierarchy**. It seems that it has more bones with more detailed names. We can select some of them and establish our physic assets over them in this chapter; you can experiment with different bones and get more realistic results.

1. Change your camera as follows. Then, click on **Simulation**.



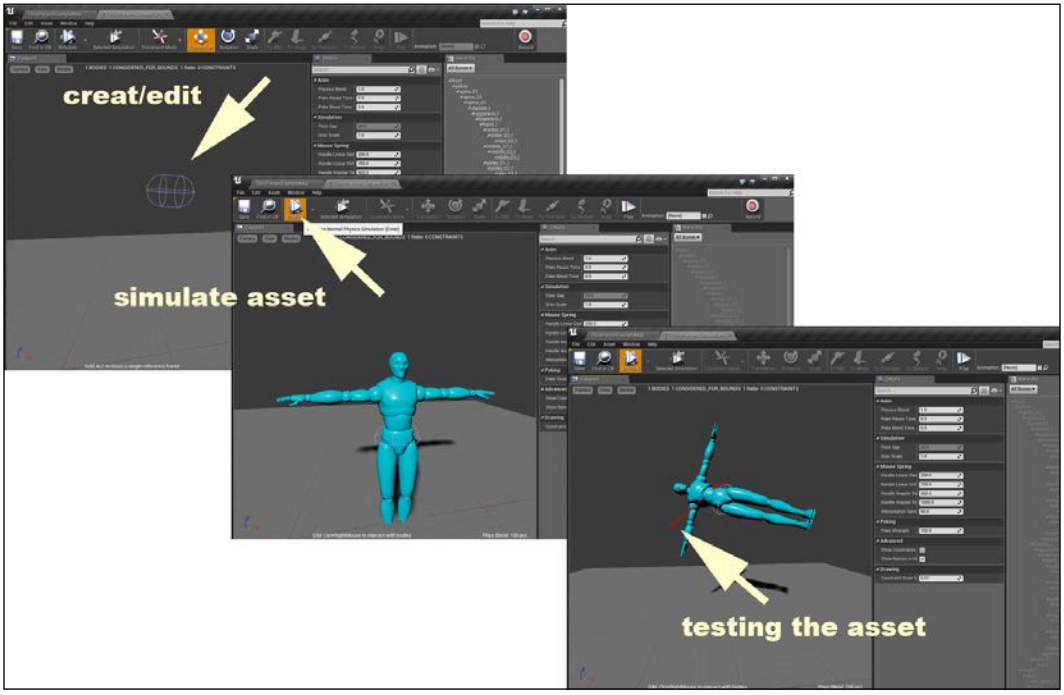
1. Nothing will happen. Now, click on **Body Mode** at the top and select **Constraint Mode**. The screen will go blank.

2. Navigate back to **Body Mode** by clicking once again on **Constraint Mode** and select **Body Mode**. You will see the body again. Now, click on **pelvis** in **Hierarchy**, right-click on it, select **New Body**, and click on **Ok** in the **New Asset** box:

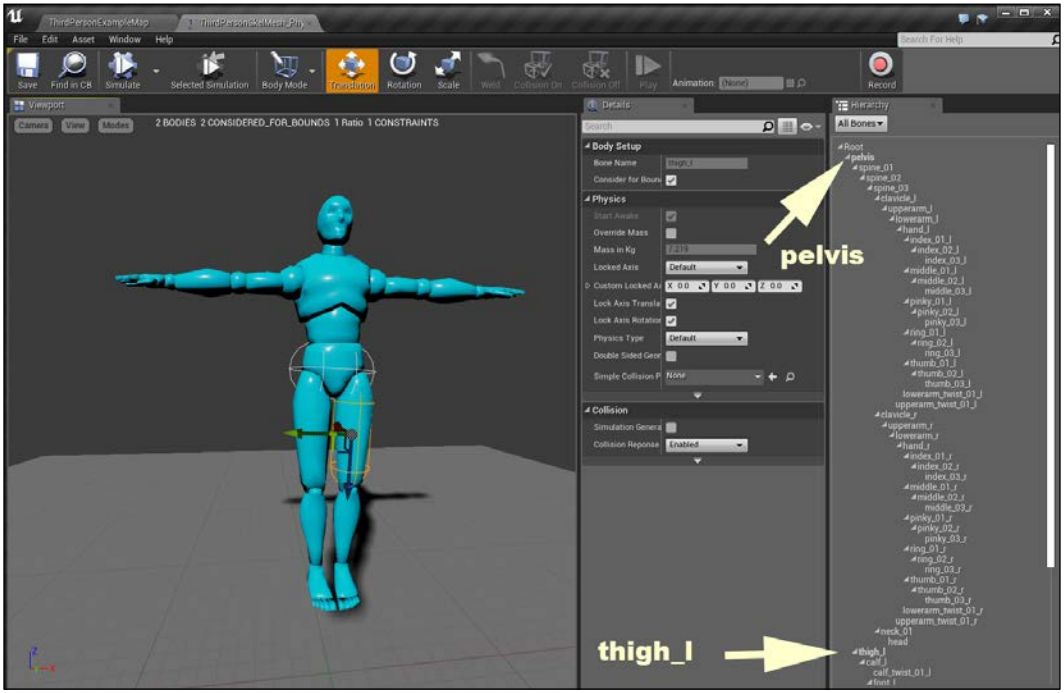


3. Now, click on **Body Mode** at the top and select **Constraint Mode**. Something will appear on the screen this time.

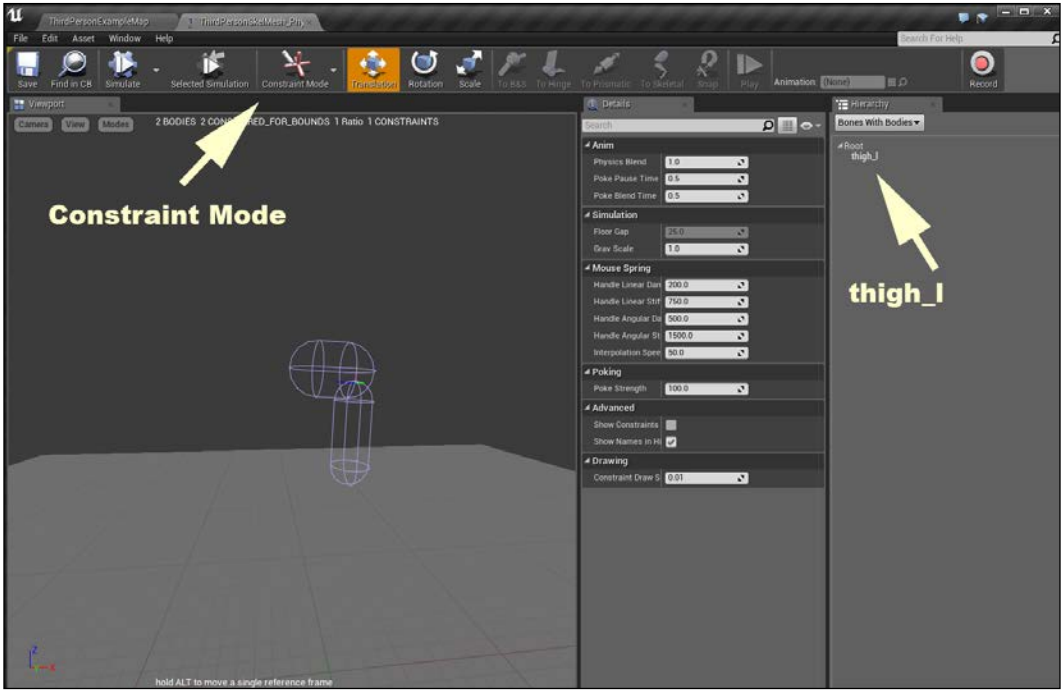
4. This is your first asset. Now, click on **Simulate** to try it in the real world. Also, drag and try to move it around (holding *Ctrl* and the right mouse button). This is mostly similar to how you check your asset functionality, create and edit, choose **Simulate** and then test with mouse.



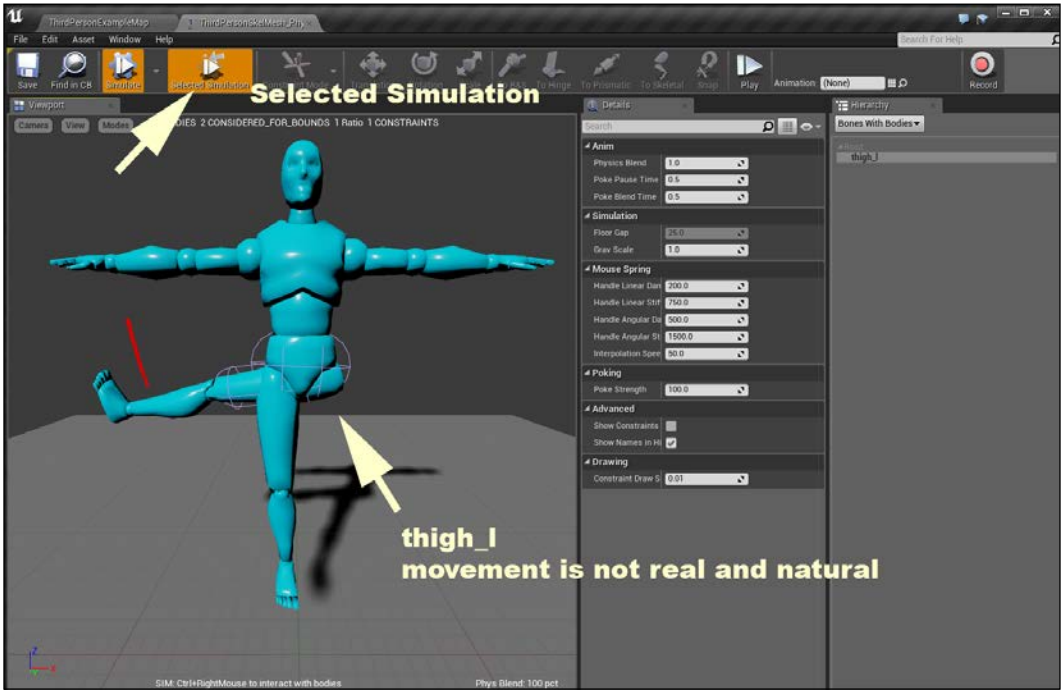
2. Switch to **Body Mode** at the top to view the entire body. Then, select **thigh_l** in **Hierarchy** and create a new body for it. Now, view the changes. Click on **Simulate** and try to move it with your mouse. As you can see, the movement is between two parts: **pelvis** and **thigh_l**, which is represented in bold fonts in **Hierarchy**.



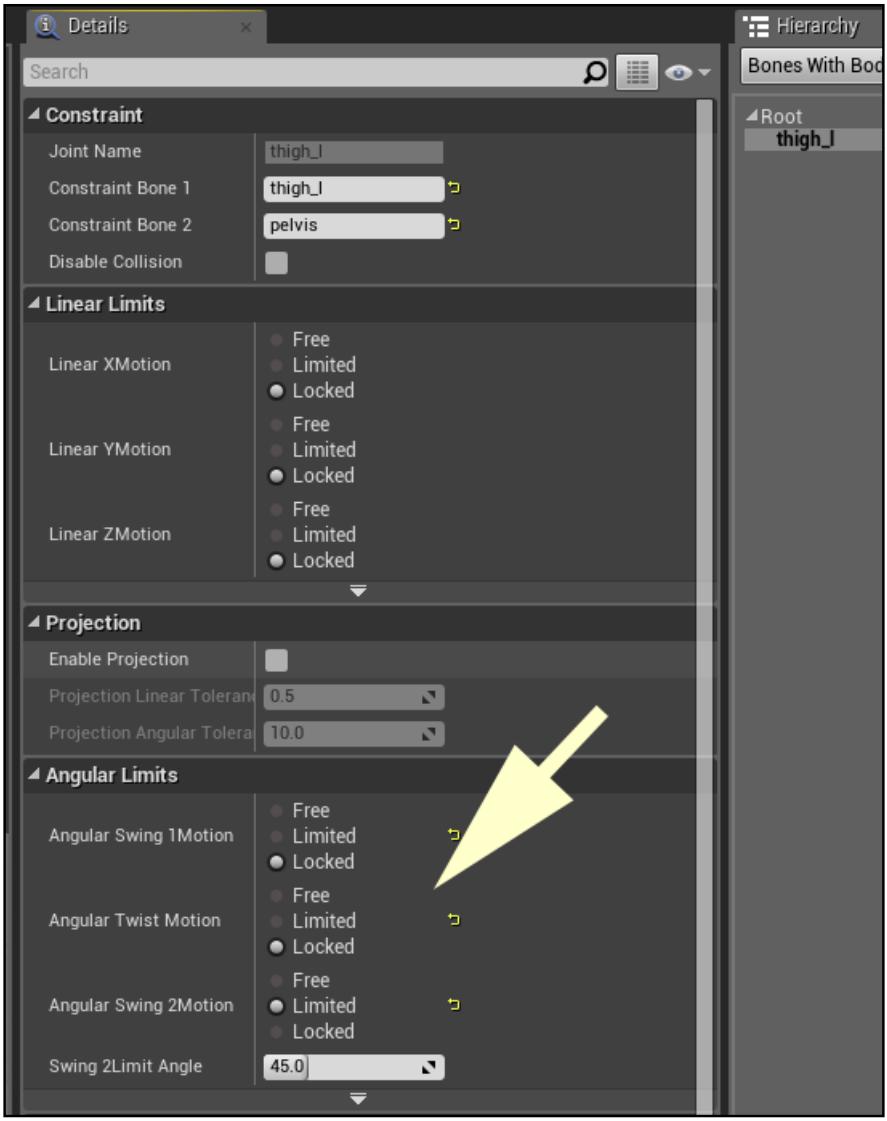
1. Switch to **Bones With Bodies** in **Hierarchy** to have just these bones in the list. Now, switch to **Constraint Mode** at the top. Now, you can see the new physic assets that you put over your bones. At this point, we have two.



2. Now, click on **thigh_l** and then on **Selected Simulation** at the top to make it active. Then, click on **Simulate**. As you can see, only the selected part will move. Test this with the mouse. It does not move naturally, and, compared to the movements of a real human body, it is absolutely unnatural.

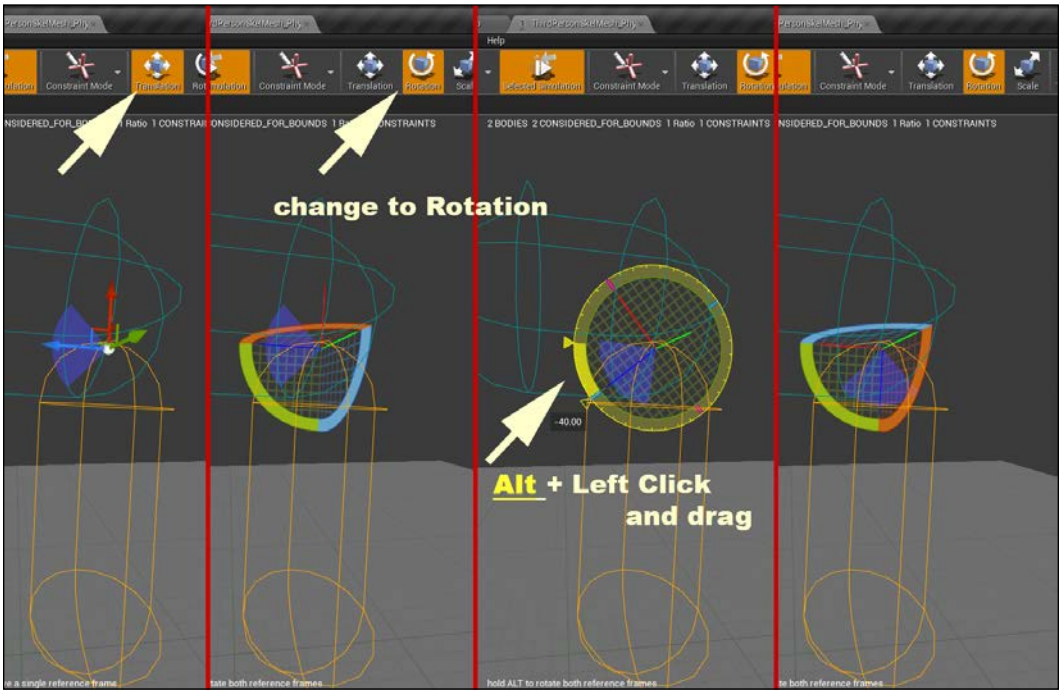


3. Now, we want to fix this so that it can move and work based on real physics. So, stop the simulation and click on **thigh_l** in **Hierarchy**. Under **Details**, locate **Angular Limits** and change the options, as shown in the following screenshot:

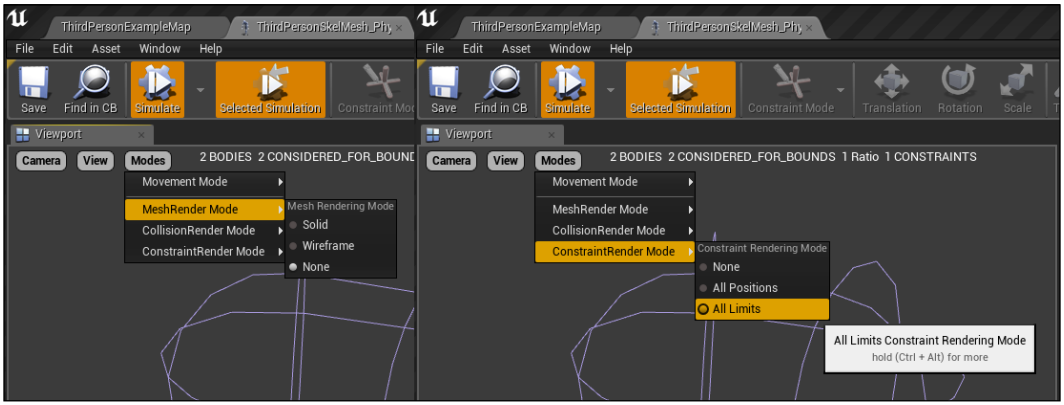


4. Now, change your camera view, as shown in the following screenshot, and click on **Rotation** at the top.
5. Once you select **Rotation**, you will see a colorful bold arc shape around your selection. Use this to click and rotate the physic asset. Also, a blue triangle like shape will appear. This indicates the maximum range in which your bone can play and move.

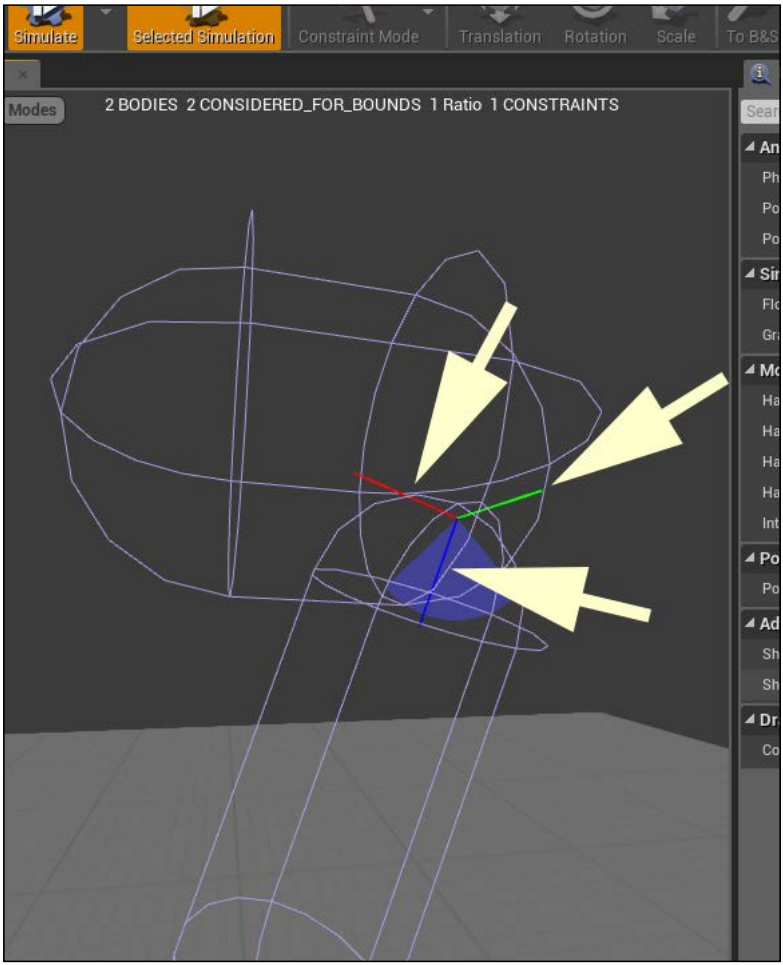
6. Click on the green bold arc around your selected point, press *Alt* (*very important*), and rotate the point so that it points to the ground, as shown in the following screenshot:



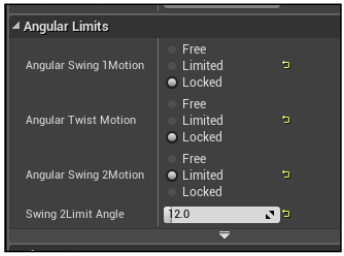
7. Now, click on **Simulate** and change your rendering options so that it looks similar to following screenshot:



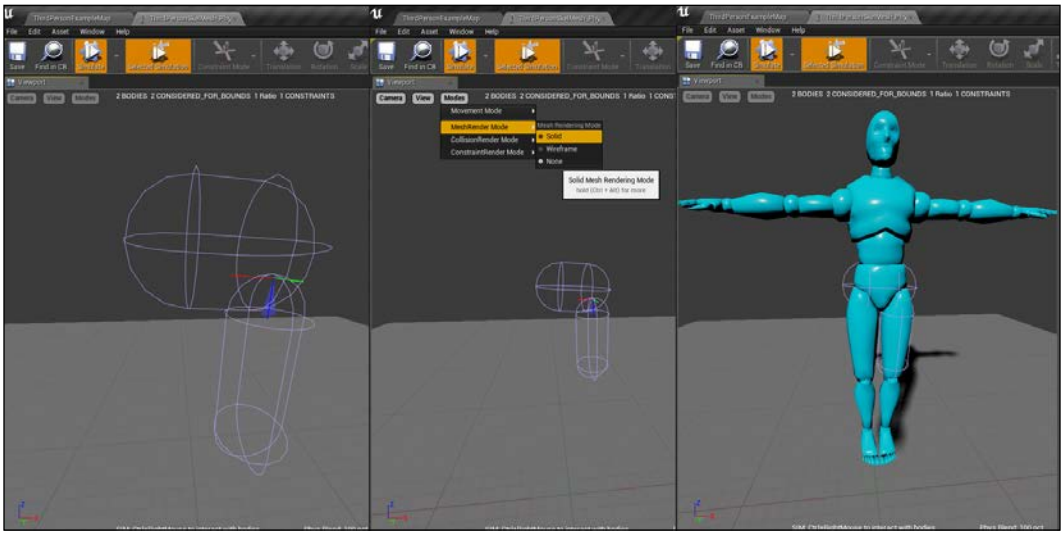
8. Basically, changing **MeshRender Mode** to **None** allows you to see the main physic assets behavior during the test. Also, setting **ConstraintRender Mode** to **All Limits** allows you to observe the movement of small tiny colorful lines (which are indicated by the arrows), as shown in the following screenshot. Understanding and controlling these lines are the key to the soft physic assets on your mesh. The way they move, the area they cover, and the other details are located in the **Detail** section.



9. Stay in the simulation mode and move the object a couple of times. It moves from left to right and always stays with the blue tiny line in the range of the blue triangular shape. Now, stop the simulation. In **Details**, inside **Angular Limits**, change **Swing2Limit Angle** from `45` (the current one) to `12.0` and run the simulation again:



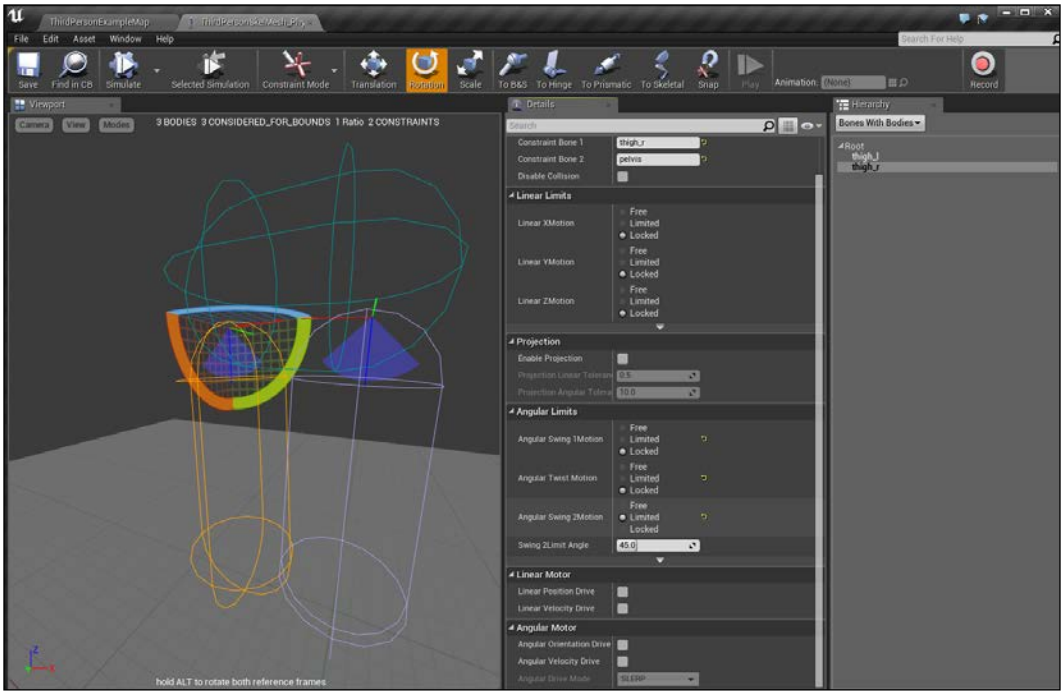
10. Now, the movements become tighter. Also, the blue triangle appears smaller. Change your view and then **MeshRender Mode** to **Solid**, as shown in the following screenshot. Then, check the movements of the left leg or the movement of **thigh_l** under your new physic asset (in a better way).



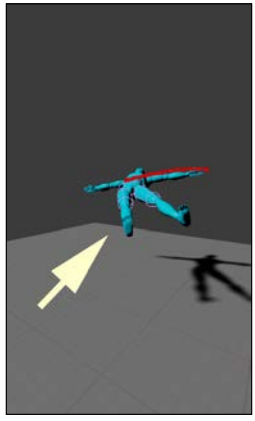
11. Now, let's change **Swing2Limit Angle** from `12` (the current one) to `45` and run the simulation again.

3. In **Hierarchy**, switch to **All Bones** and then to **Body Mode** from the top menu. Finally, select **thigh_r** from the bone list, right-click on it, and select **New Body**. This is the second leg.

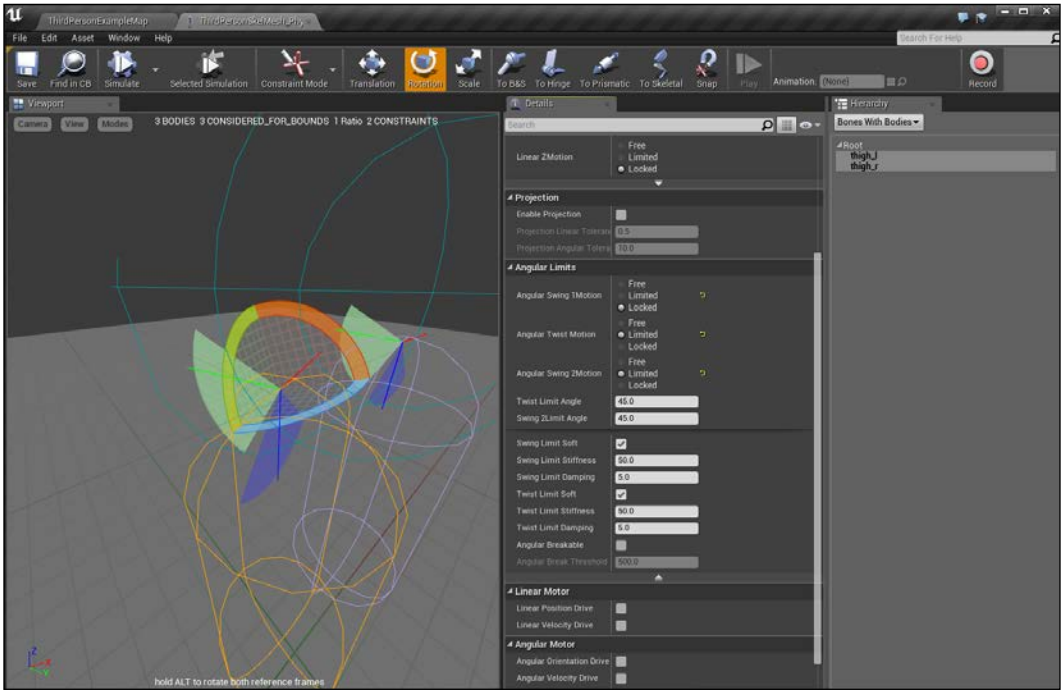
1. Switch to **Constraint Mode** at the top. Similar to the last bone, make the rotation and details look similar to the following screenshot. Also, turn off **Selected Simulation** from the top menu:



2. Change **MeshRender Mode** to **Solid** and check the behavior of the body when you move it on the stage. The legs move much more naturally than before, but it needs to move in different angles as well, as shown in the following screenshot:

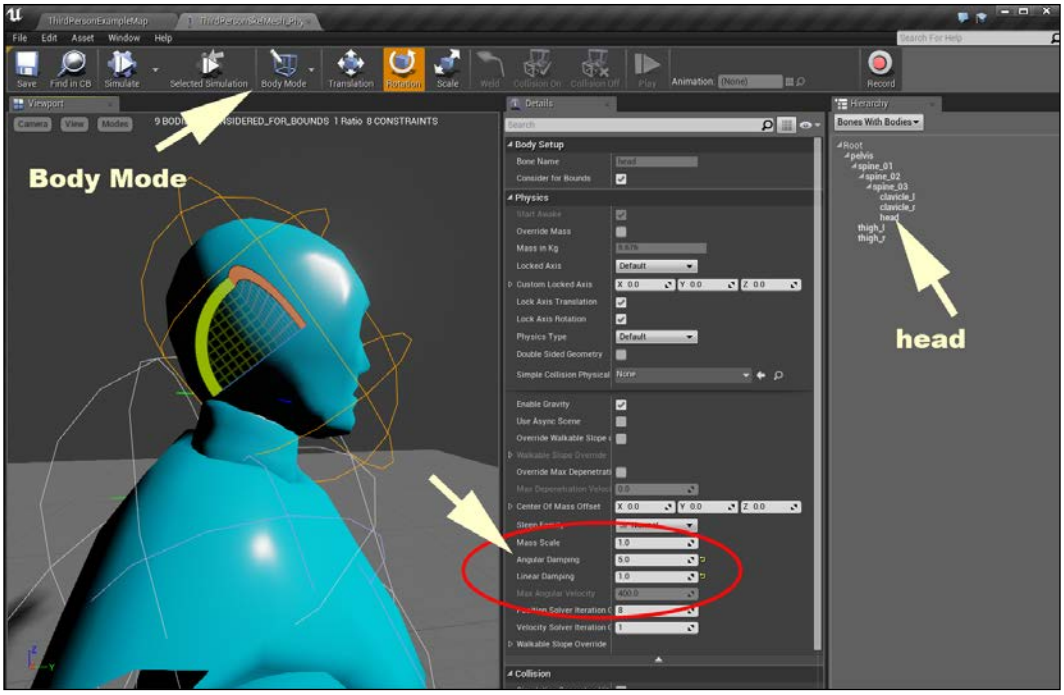


3. To apply more details, play with rotation and angles to reach the following screenshot:



4. Try the same process with the **spine_01**, **spine_02**, and **spine_03** bones. Then, fix the limitations, angles, and work on **clavicle_l** and **clavicle_r**. Finally, work on **neck_01** and **head**. This way, you can grow your physic assets on the body in a properly organized way. You can test the overall movement and, when everything looks fine, increase the assets on smaller bones (such as fingers).

4. As you go further, you should be aware of a correction that needs to be performed at the end of each series of bones. Similar to fingers, head, and toes, this is simply related to physical energy. When the body moves, based on the physic assets and the way you edit these, the body reflects and guides the movements from one bone to another. If there is no bone after the current one, such as **head** or the last part of the finger, it sometimes ends in an unusual vibrate-like movements over the bone during the simulation and the real game. To guide this energy in a controlled way, click on the bone in **Body Mode**, expand the **Physics** section in **Details**, set **Angular Damping** to `5` and **Linear damping** to `1`, and simulate. Now, check your movements and change these numbers until the problem is fixed.



Summary

Now, you can use PhAT to make your character move and reflect, which is similar to the real world. Keep in mind that working on PhAT is presenting the game story over the character behavior. It is an art because all the details and time that you spend here are directly monitored and tried by the player (perhaps hundreds of times while playing the game). It is good practice to imagine a character with some special behaviors, like movement, impact on other objects like walls, guns or fire, in real world; and then, try to recreate that behavior using Unreal Engine. It is a physical simulation with many details which we have gone through in this chapter. As a game designer, more practice will guarantee high quality results.

You can import the rigged mesh from the 3D software and apply the same physic assets to them. The mesh should be rigged in a proper way, which is similar to what we used as default in this chapter. Try the online sources by searching Unreal Engine 4 PhAT on YouTube. Also, if you practise the example, you can use the resources of Unreal Engine 3 and tuts in this area.

3
Collision

In this chapter, we will analyze collision in Unreal Engine 4, what it is, the different types of collision that exist in the engine, how to use it, and how to apply it to both static meshes and blueprints. To start with, we will first take an overview look of the different collisions that exist in Unreal Engine 4, but we will also cover the following topics:

- Simple versus complex collision
- Generating simple collision
- Creating complex and custom collision hulls
- Collision interactions
- Custom object and trace channels
- In-depth collision presets

For the purposes of this chapter, we will continue to work with Unreal Engine 4 using the **Unreal_PhyProject** that we created in the first chapter.

Collision and Trace Responses – an overview

In the real world and in Unreal Engine 4, we define collision as an overlap of two or more objects. In the context of Unreal Engine 4, **Collision** and **Trace Responses** lay the groundwork for how Unreal Engine 4 handles collision and ray casting during the game. Every object that is given collision gets an **Object Type** and a series of responses that describe how it interacts with the other object types. In the event of either a collision or an overlap of two or more objects, all objects involved can be set to affect or to be affected by blocking, overlapping, or ignoring one another.

Trace Responses describe how an object should react when you interact with a trace, which is done with a ray cast. An object can choose to block, overlap, or even ignore a trace from a particular source. By default, there are two different **Trace Responses**:

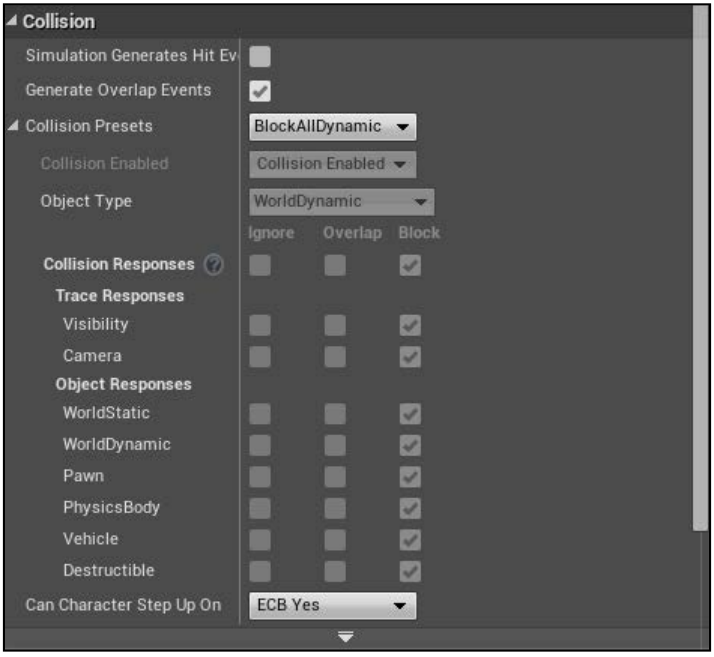
- **Visibility**: This specifies a trace from one position to another
- **Camera**: This is exactly similar to the **Visibility** trace response, but it should be used when you use a ray cast from the camera

Object Responses describe how an object should respond when you interact with other objects in our game world. Similar to **Trace Responses**, **Object Responses** offer the ability to choose whether or not an object will block, overlap, or ignore other objects when a collision occurs. By default, there are six different types of **Object Responses**:

- **WorldStatic**: This object response is for objects in our game world that are static, meaning that they do not and cannot be moved by any means. Objects such as volumes, world geometry, or any other meshes in the game world are associated to this object response.
- **WorldDynamic**: This object response is for objects in our game world that are moving actors, outside of player pawns, physics bodies, vehicles, and destructible actors. Examples of **WorldDynamic** objects would be an elevator, a door that can open and close, or a wheel that a player can turn.
- **Pawn**: This object response is for player characters in our game or any other character that can be possessed by the player.
- **PhysicsBody**: This object response is for any physics body or object that can be simulated with physics in our game world. An example of a **PhysicsBody** object would be a basketball that the player can pick up and throw; *Half-Life 2* is a great example of how physics body object collisions are used in games.

- **Vehicle**: Although this object response is labeled as **Vehicle**, what this response is useful for is to have player pawns jump into them, such as a vehicle.
- **Destructible**: This object response is for any actors that are destructible, meaning that they can break apart using the destructible mesh editor.

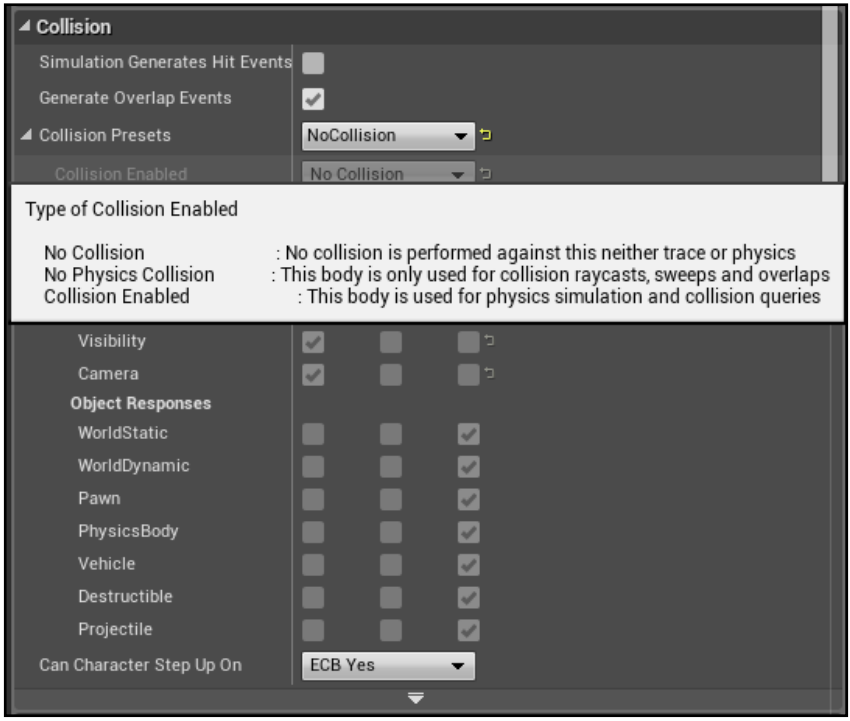
When you work on setting up collisions on an object or a component in Unreal Engine 4 blueprints, you will see the following properties:



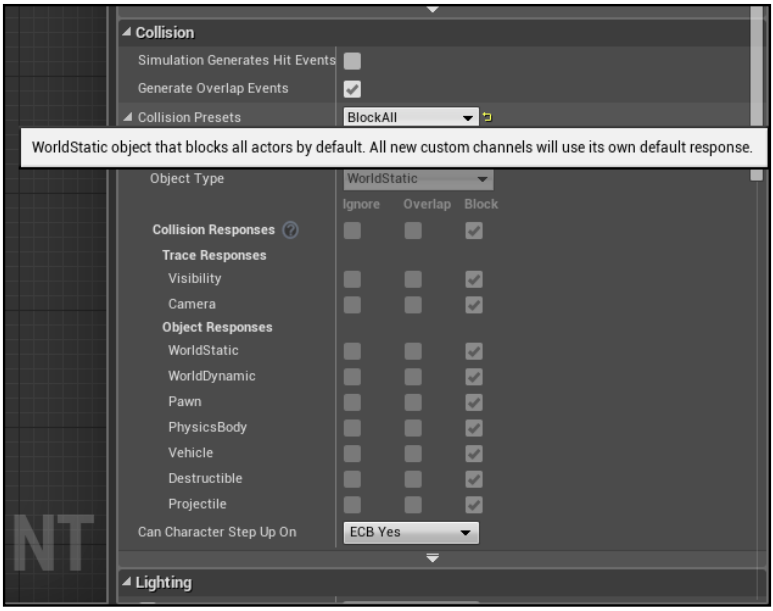
It is important to note that we want to make sure that our static mesh or blueprint component has collision generated before setting any collision presets to that object; otherwise, we will not receive any responses once a collision occurs. Later in this chapter, we will go into more detail on how to generate simple and complex collisions for our objects.

When it comes to setting up **Collisions** to an object, there are numerous collision presets that default to Unreal Engine 4 that either ignores, overlaps, or blocks a combination of trace and object responses. In addition to these presets, we do have the option to create a custom collision preset for certain circumstances in our blueprint. Feel free to explore some of the collision presets and how they differentiate from one another, but for the sake of this text, let's take a look at some of the more common presets. We will take an in-depth look at the following presets later on in this chapter:

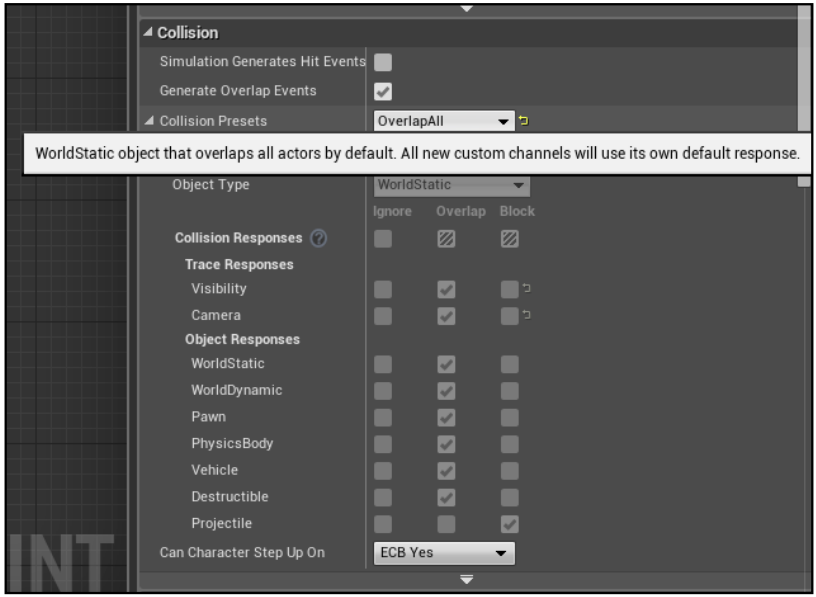
- **No Collision**: As the name suggests, this collision preset eliminates any collision responses by setting the **Visibility** trace response and the **Camera** trace response to ignore and sets **Collision Enabled** to **No Collision**. Typically, we would use this for blueprint components that we don't want to react to any collisions that may occur.



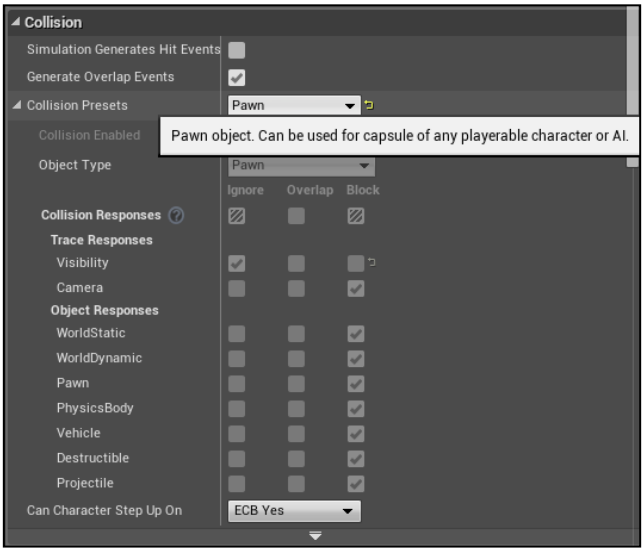
- **Block All**: This collision preset causes all the collisions with the associated component to result in a block. Alternatively, it causes all the objects involved in this collision to hit and bounce off one another if physics are applied. This is done by setting all the responses, including the trace and object responses, to **Block** under their **Collision Responses**, as shown in the following screenshot:



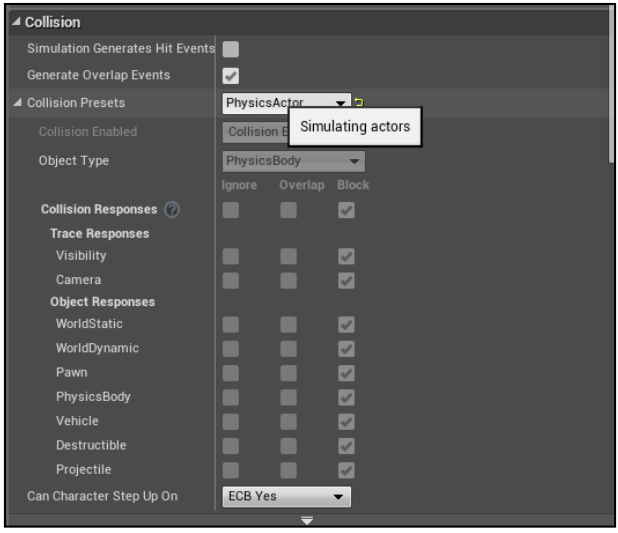
- **Overlap All**: This collision preset results in all the collisions to generate an overlap between all the objects involved in the collision. As long as the **Generate Overlap Events** property is checked, we can use the blueprint collision events to enable behaviors or events to occur once this type is involved in a collision.



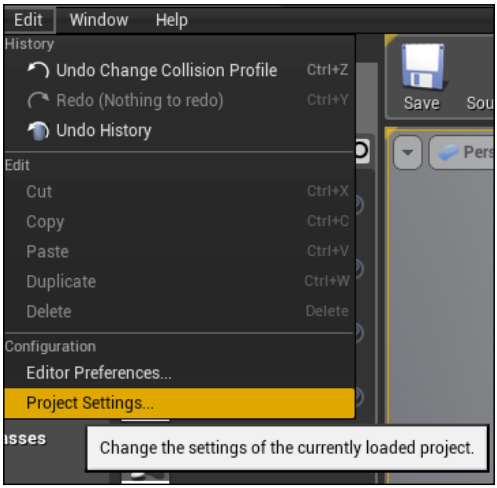
- **Pawn**: This collision preset is useful if it is used for a player pawn or character in our game. By default, it is set to block **Object Responses**, block the **Camera** trace response, and ignore the **Visibility** trace response:



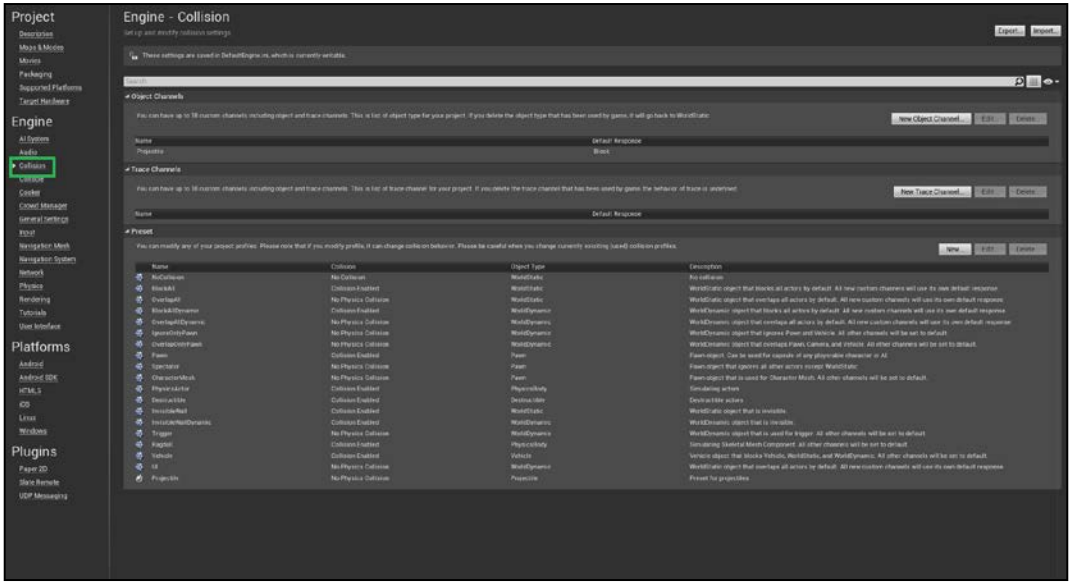
- **Physics Actor**: This collision preset is used for any actor or component that is a physics-based actor, meaning that the object has in-game physics (such as gravity) applied to it. In order for this preset to work properly, we want to make sure that the **Simulate Physics** property in the **Physics Tab** is checked. By default, all the **Trace Responses** and **Object Responses** are set to **Block**:



These are just a few of the different options that Unreal Engine 4 offers by default for collision, and we will cover the other options in more detail later on in this chapter. Although there are a handful of options when it comes to collision presets offered in Unreal Engine 4 by default, a really nice feature that is in place is the ability to create your own trace, object channels, and collision presets. To do this, we need to navigate to the **Edit** window and select **Project Settings**:



From here, we need to navigate to the **Collision** option in the **Engine** category:



In this menu, we can create custom collision presets, specify which object and trace channels to either ignore, overlap, or block, give it a specific name, and save it to the project file. For advanced needs, we can also create custom object and trace channels in this window. Later in this chapter, we will create our own custom collision preset and apply it to an object.

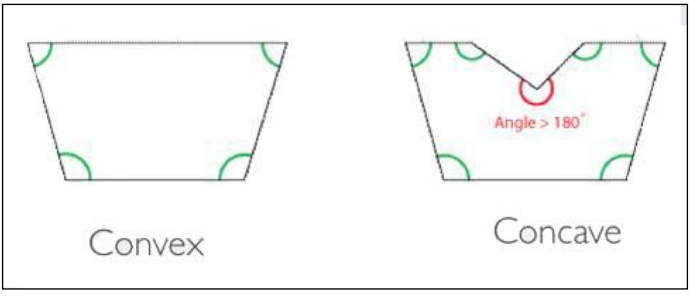
Collision and Trace Responses – a section review

In this section, we briefly looked at the different **Collision** and **Trace Responses** that exist in Unreal Engine 4 and defined a handful of these responses. We analyzed the different **Trace Responses** and **Object Responses** that default to Unreal Engine 4, and we also defined a limited number of collision presets that are provided. Now that we have a basic understanding of **Collision** and **Trace Responses**, we can move forward and learn more about simple and complex collision in Unreal Engine 4.

Simple versus complex collision

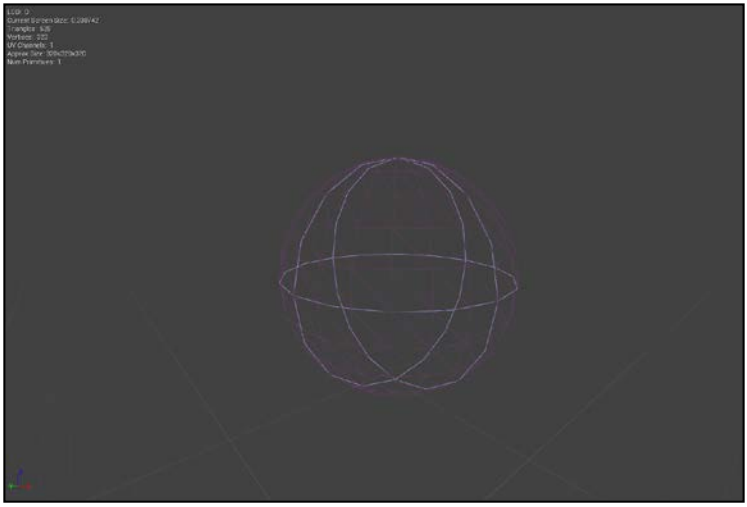
In Unreal Engine 4, we will be able to autogenerate collisions for our meshes that can be used in our game. There are two different types of collision that exist in Unreal Engine 4: simple and complex collision. Each type of collision serves its own unique purpose, and in this section, we will simply define each collision type and provide examples of each. Later on in this chapter, we will work on how to apply these collisions to our objects. We will also test these collisions in our game. Let's begin with simple collision.

A simple collision is a collision mesh that uses basic shapes, such as boxes, spheres, capsules, and convex shapes, to define the bounds of our object. Convex shapes are ones that have one or more interior angles that are less than 180 degrees, whereas concave shapes are ones that possess one or more interior angles that are more than 180 degrees, as shown in the following image:

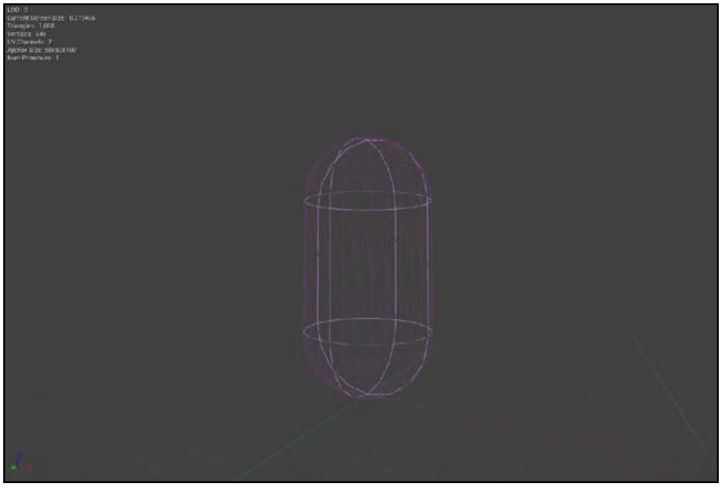


In addition to these basic shapes, we can generate a form of simple collision called **KDOP** or **K Discrete Oriented Polytope** (where K is the number of axis-aligned planes). What this option essentially does is that it takes the *K* axis-aligned planes and moves them as close as possible to the selected mesh. We will go into more detail on how to generate these different types of simple collision later on. Now, let's define the different types of simple collision here:

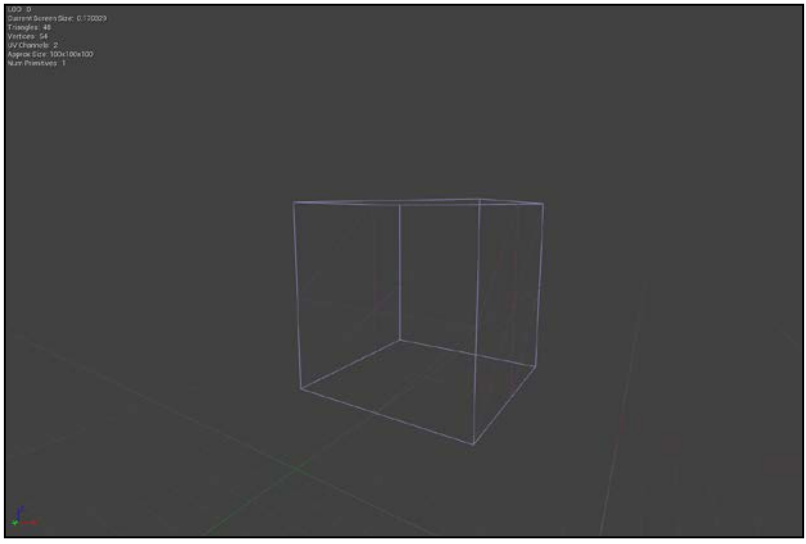
- **Sphere**: This creates a spherical bound mesh around the selected object. It can be used in physics objects and to apply collision to objects that are round.



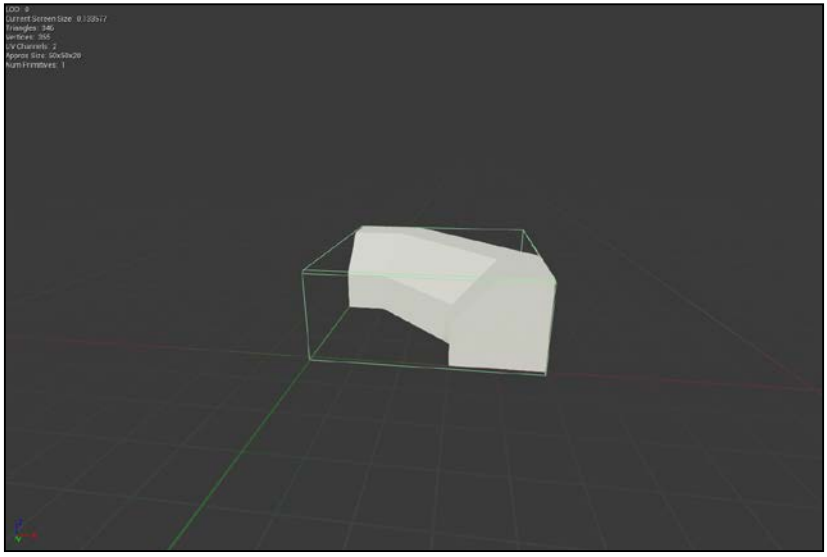
- **Capsule**: This creates a capsule bound mesh around the selected object and is typically used for character or pawn meshes:



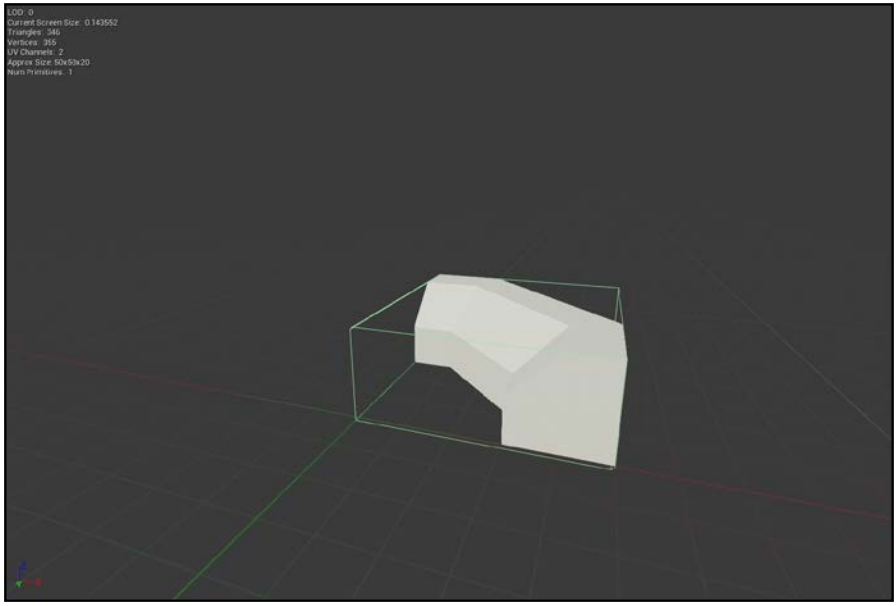
- **Box**: This creates a box bound mesh around the selected object. This type of simple collision is most commonly used for environment meshes:



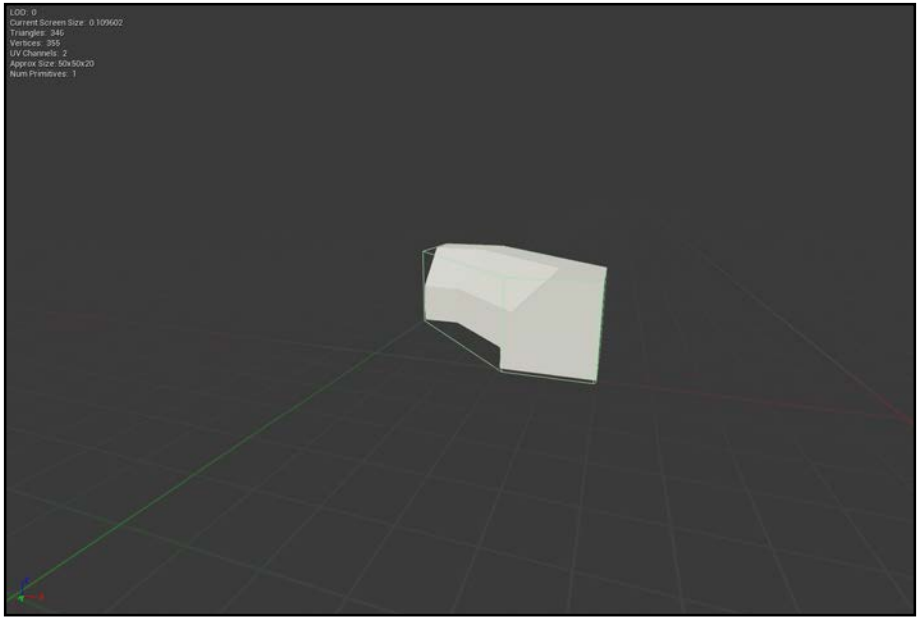
- **10DOP X**: This creates a box with four edges beveled in the *X*-aligned edges:



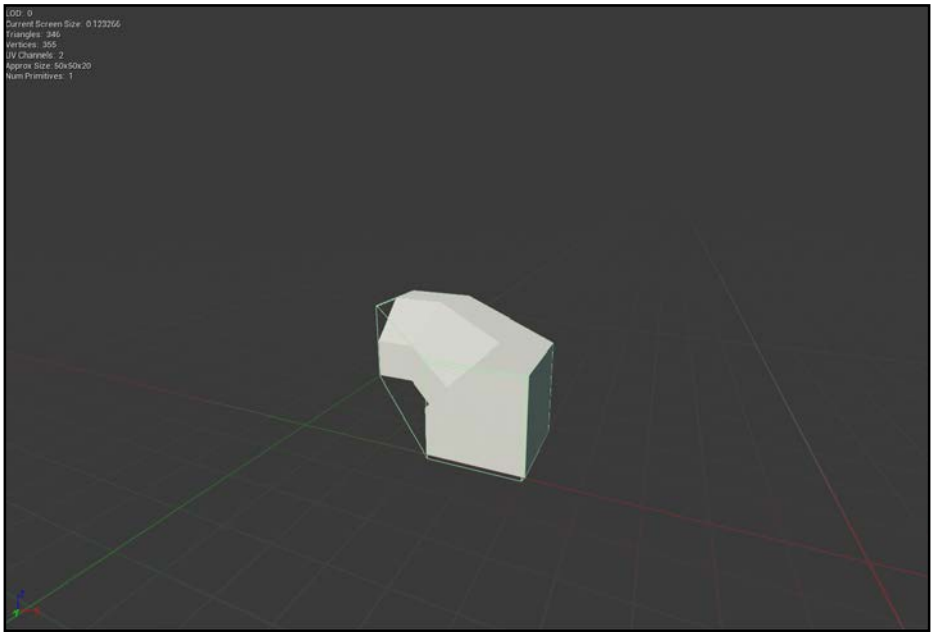
- **10DOP Y**: This creates a box with four edges beveled in the *Y*-aligned edges:



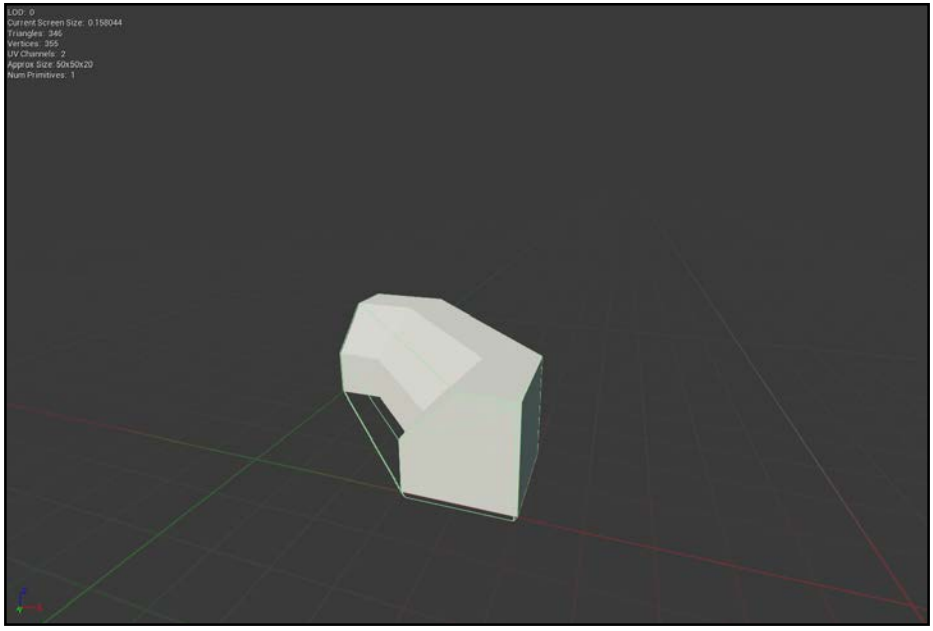
- **10DOP Z**: This creates a box with four edges beveled in the *Z*-aligned edges:



- **18DOP**: This creates a box with all of its edges beveled:

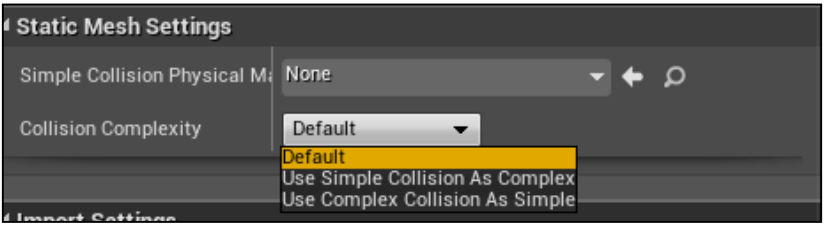


- **26DOP**: This creates a box with all of its edges and corners beveled:



The main advantage of simple collision is that it almost eliminates the possibility of an object getting stuck to a player or vice versa. An additional advantage is that the collision mesh is of a basic shape, which is less expensive to use in the game at runtime.

Complex collision is done for each polygon and is very expensive in Unreal Engine 4 as compared to simple collision. Moreover, complex collision is never used for an actor that is simulating physics, and it will just fall through the game world. In order to enable complex collision in the **Static Mesh** editor, we need to navigate to **Details Panel** and then to the **Static Mesh Settings** section. It is here that we can change the **Collision Complexity** parameter to **Use Simple Collision As Complex** or **Use Complex Collision As Simple**:



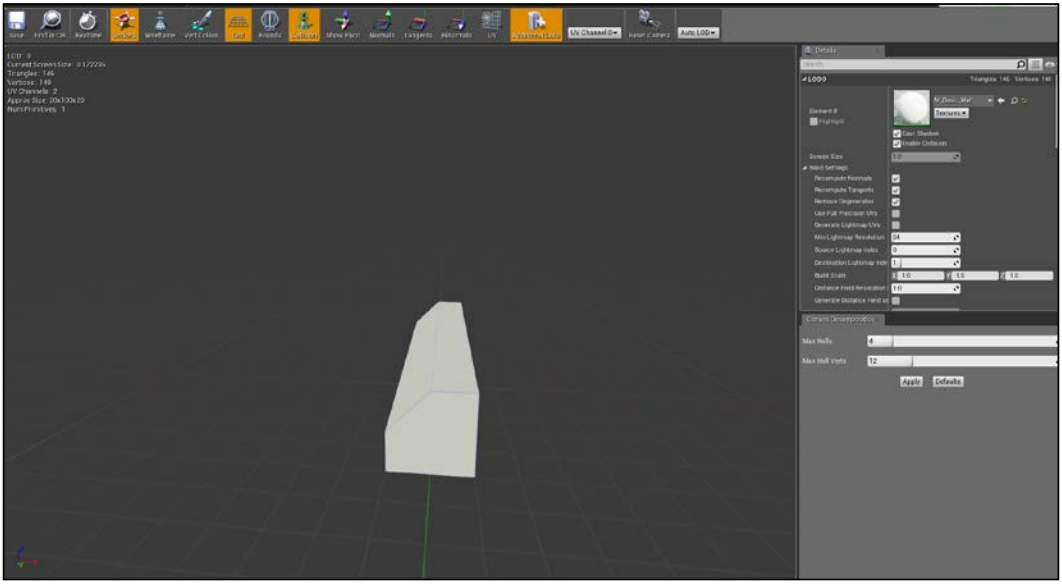
Simple versus complex collision – a section review

In this section, we looked at the different simple collisions offered by default in the **Static Mesh** editor in Unreal Engine 4. We also discussed the advantages and disadvantages of the simple and complex collision when it comes to game development and engine performance. Lastly, we briefly looked at how to create both these types of collision. With a basic understanding of simple and complex collision under our belts, we can now discuss how to create simple collision, and how to create collision hulls in Unreal Engine 4 later on.

Creating simple collisions

When it comes to creating collisions, there are many options that we can take advantage of to properly utilize collision and optimize game performance. As we discussed in the previous section, we have the option to create simple and complex collisions in the **Static Mesh** editor in Unreal Engine 4, but we can also use third-party art programs to create custom collision hulls. Let's first discuss how to create simple collisions in Unreal Engine 4, and in the next section, we will discuss how to create complex and custom collisions for our assets.

Let's begin by opening `StarterContent` and navigating to **Content Browser**. From here, let's go to the `StarterContent` folder and select the `Shapes` folder that contains multiple simple-shaped static meshes to select from. For this set of examples, we will choose the `Shape_Trim` mesh because it is a more complicated shape as compared to a sphere or box; this way, we can see the effects of different collision options. Double-click on the `Shape_Trim` asset to open the **Static Mesh** editor.



Navigating to the **Static Mesh** editor is very similar to moving around in the **Perspective** view mode in the main game editor of Unreal Engine 4. At the top of the **Static Mesh** editor is the main toolbar that provides a handful of useful options when you view your mesh and its collision.

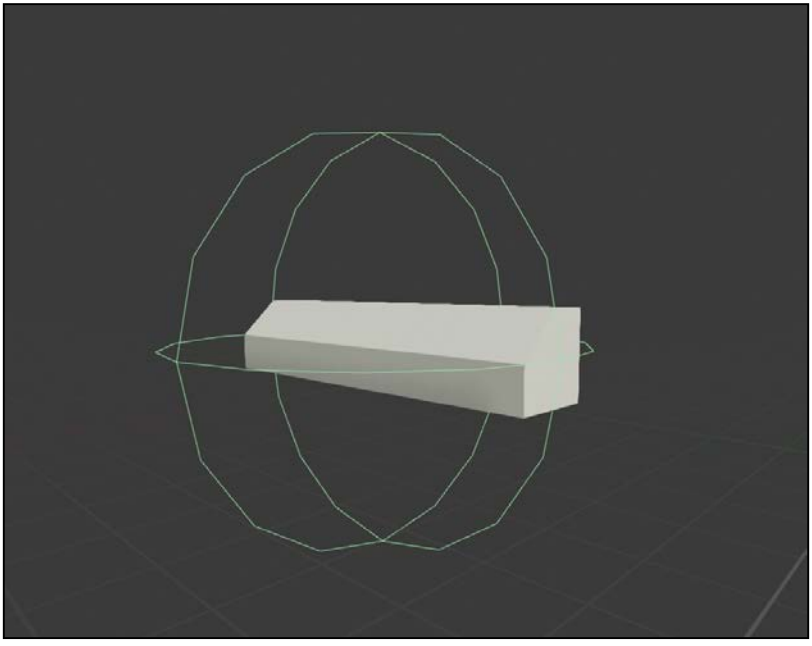
The toolbar provides us the options to save our mesh and its properties, to view the mesh in real time, which is useful if the mesh has an animated material applied to it, to view any applied **Sockets**, to toggle the **Wireframe** of the mesh, to view any **Vertex Colors** applied to the mesh, to toggle a background **Grid**, to toggle the **Bounds** of the mesh, and (most importantly) to toggle the **Collision** applied to the mesh. Additionally, we can view the mesh's **Pivot Point**, its **Normals**, **Tangents**, **Bi-Normals**, and **UV** sheet.

As we will work primarily with collisions, we will want to make sure that the **Collision** option is toggled on so that we can see the bounds of the bounding collision mesh. To do this, we can left-click on the **Collision** button to make sure that it's highlighted in orange, and if the mesh has any collision applied to it, we will see it in a light blue-colored wireframe around our object. By default, `Shape_Trim` does have a collision applied to it, so we first want to remove this collision so that we are able to apply only one collision mesh to the object at once for demonstration purposes.

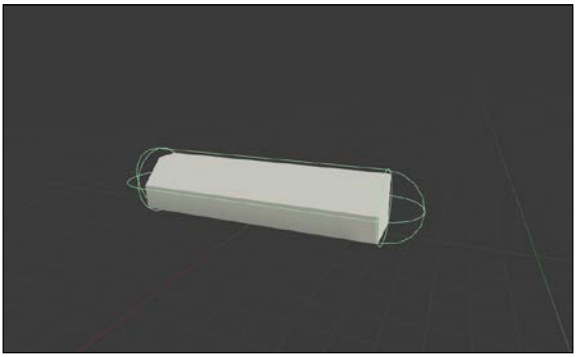
1. First, navigate to the **Collision** drop-down window at the very top of the **Static Mesh** editor located alongside the **File**, **Edit**, **Asset** window options.
2. Then, select **Remove Collision**.

Now, the light blue-colored wireframe outline mesh will disappear from our mesh, meaning that this asset no longer has any collision applied to it. It is also very important to keep in mind that we do not want more than one collision-bounding mesh applied to an object at once in order to keep our assets as optimized as possible, unless the shape of the mesh demands more than one collision mesh.

When it comes to generating simple collision in the **Static Mesh** editor, it is as easy as clicking on a few buttons in its interface. Let's start by creating a **Sphere** collision in our `Shape_Trim` mesh by clicking on the **Collision** drop-down menu and selecting **Add Sphere Simplified Collision**. Once complete, we should see a collision-bounding mesh that looks similar to the following screenshot:

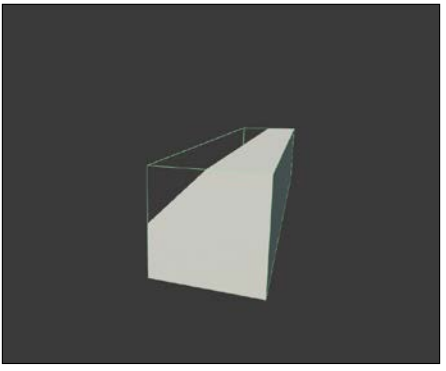


The **Sphere Simplified Collision** option sets the radius of the sphere that best matches the size and shape of the mesh that it is applied to. We should also note that the collision wireframe changed from light blue to green; this means that the collision will use a simple shape. Once a collision is generated, the shape can be moved, rotated, and scaled to the desired size and shape. For this shape, a sphere collision does not seem to work as we would like it to work, so let's select the **Remove Collision** option from the **Collision** drop-down list and then the **Add Capsule Simplified Collision** option.

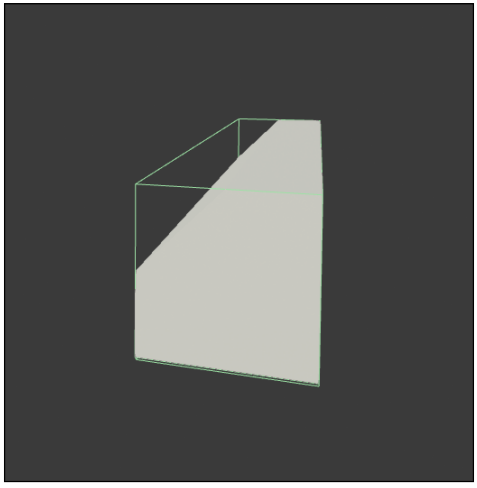


As we can see, the **Capsule Simplified Collision** option does a much better job of matching the size and shape of our mesh than the **Sphere Simplified Collision** option because it sets the capsule's height and radius as opposed to just setting the radius. We can still see that the collision-bounding mesh does not fit this shape as closely as we would like, so let's continue to add differently shaped collision meshes in order to find the best one.

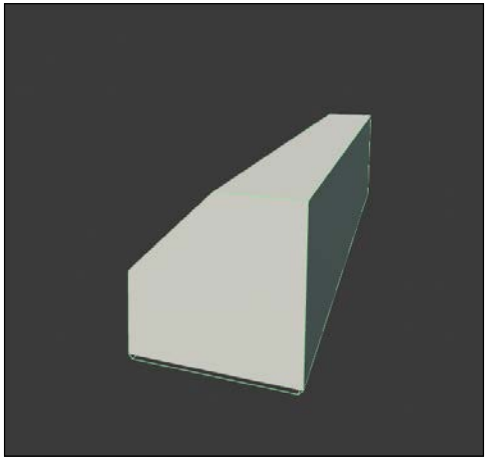
Let's remove the capsule collision-bounding mesh and instead select the **Add Box Simplified Collision** option to `Shape_Trim`. Here, we can see that the box shape does a really good job of matching the size and shape of the mesh, and in most situations, we would use this option for this asset for use in our game.



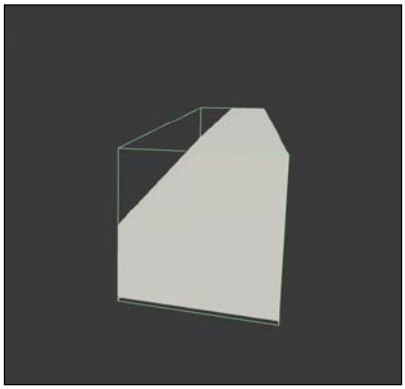
For the purposes of this chapter, we will continue to apply the **KDOP Collision** options to this mesh so that we have a better understanding of their purposes and the results that we can get from these options. Now, let's remove the **Box Simplified Collision** option and use the **Add 10DOP-X Simplified Collision** option. If you remember from the previous section, the **10DOP-X Simplified Collision** creates a box with four edges beveled in the *X*-aligned edges. Then, we get the following result:



As we can see, the **10DOP-X Simplified Collision** option generates a collision-bounding mesh identical to the **Box Simplified Collision** option. Now, let's try applying the **10DOP-Y Simplified Collision** option, which creates a box with four edges beveled in the *Y*-aligned edges by first removing the **10DOP-X Simplified Collision** option and then selecting the **Add 10DOP-Y Simplified Collision** option to obtain the following collision mesh:

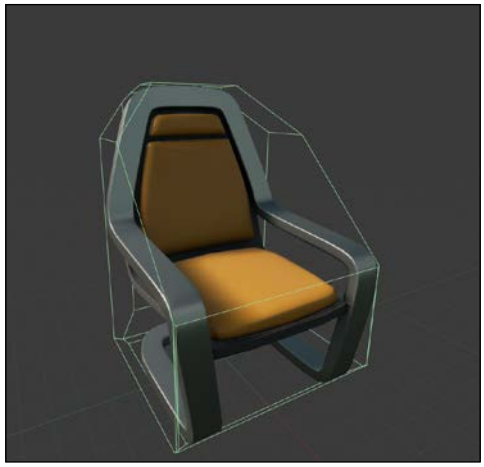


As we can see here, the **10DOP-Y Simplified Collision** option does an excellent job of almost exactly matching the size and shape of our `Shape_Trim` static mesh. This is definitely a viable option to select when you generate a collision for this asset. Lastly, let's apply the **10DOP-Z Simplified Collision** option to view how it generates a collision mesh around our asset. First, let's remove the **10DOP-Y Simplified Collision** option and then navigate to the **Collision** drop-down menu and select **Add 10DOP-Z Simplified Collision**, which creates a box with four edges beveled in the *Z*-aligned axis to obtain the following result:

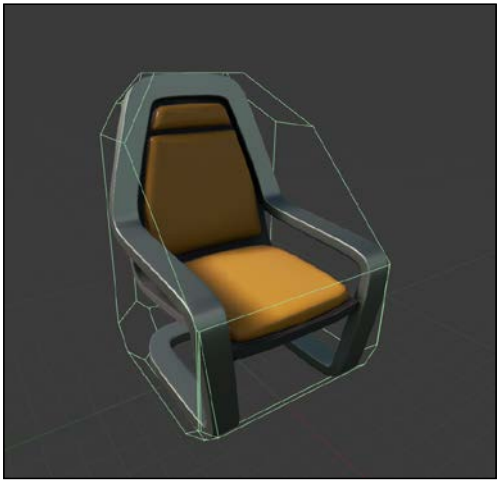


The result is identical to what we obtained when we applied the **Box Simplified Collision** and **10DOP-X Simplified Collision** options. Based on the results we received from these options, the best choices for this asset would either be **Box Simplified Collision** or **10DOP-Y Simplified Collision**. Due to the simplicity of this asset, the **18DOP** and **26DOP Simplified Collision** options won't produce unique options, so to properly demonstrate these choices, we need to choose a different asset.

To demonstrate this, let's close **Static Mesh** editor for the `Shape_Trim` asset and navigate to **Content Browser**. Here, under the `Starter Content` folder in the `Props` folder, we will double-click on the `SM_Chair` asset to open this mesh in **Static Mesh** editor. The `SM_Chair` asset does have the default collision applied to it, so before we apply our own, let's first remove its collision. Then, let's go ahead and select the **18DOP Simplified Collision** option and view how it generates a collision-bounding mesh for our chair:



If you remember, the **18DOP Simplified Collision** option creates a collision-bounding box with all of its edges beveled, creating a nice collision around our chair. Here, let's apply the **26DOP Simplified Collision** option by first removing our collision and then selecting the **Add 26DOP Simplified Collision** option:



As you can see, the **26DOP Simplified Collision** option creates a box that has all of its edges and corners beveled, creating a smoother and more rounded collision mesh around our asset.

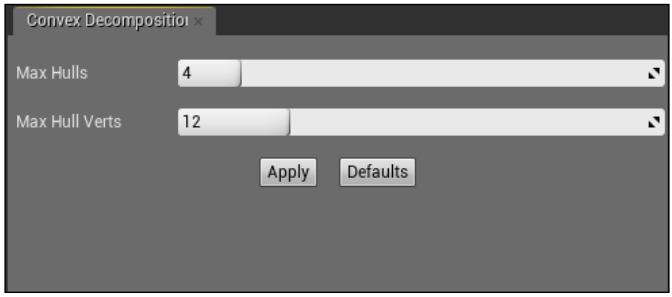
Creating simple collisions – a section review

In this section, we took a more in-depth look at the different types of simple collision that can be generated in the **Static Mesh** editor in Unreal Engine 4 and the pros and cons of each type. Using starter content assets as examples, we applied each type of simple collision to view how they are generated based on the size and shape of our asset to better understand how they work. Now that we have taken a deeper look at how to generate simple collisions in Unreal Engine 4, let's now move on and take a look at how to generate complex and custom collision hulls using Unreal Engine 4.

Creating complex and custom collision hulls

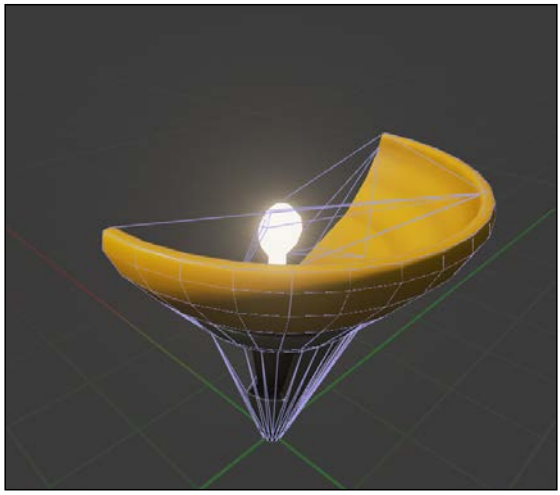
When it comes to creating complex collision in the **Static Mesh** editor, we can use the **Auto Convex Collision** tool to customize the number of hulls and hull vertices that the collision mesh will have. For the purposes of this section, we will need to continue using `Unreal_PhyProject` that we created, and we will use the `SM_Lamp_Wall` asset as an example of how to generate custom and complex collision hulls. To navigate to this asset, we need to go to **Content Browser** and then to the `Starter Content` folder. Now, under props, we will find the `SM_Lamp_Wall` asset. Double-click on this asset to open the **Static Mesh** editor. If this static mesh has any default collisions applied to it, make sure to remove the said collision by navigating to the **Collision** drop-down menu and selecting remove collision. Make sure that the **Collision Toggle** option is set to on so that we can view the collision mesh in the **Static Mesh** editor.

For this asset, we will use the **Auto Convex Collision** tool that provides us with a set of parameters to generate **Complex Collision**. To use this tool, we need to navigate to the **Collision** drop-down menu and select the **Auto Convex Collision** option. Once done, we will be provided with a submenu on the right-hand side under **Details Panel** labeled as **Convex Decomposition** with the following parameters:



- **Max Hulls**: This parameter determines the number of hulls that are created to best match the size and shape of the mesh.
- **Max Hull Verts**: This parameter sets the maximum number of collision hull vertices. By increasing this value, we can see how complex the collision hulls can be.
- **Apply**: This parameter generates a collision mesh based on the **Max Hulls** and **Max Hull Verts** parameters.
- **Defaults**: This parameter resets the **Max Hulls** and **Max Hull Verts** parameters back to their default values of `4` and `12` respectively (as seen in the previous image).

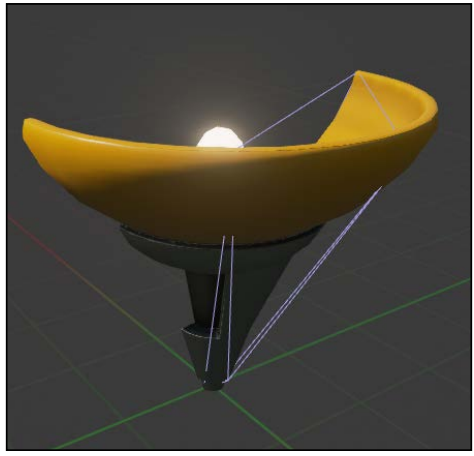
For the sake of providing examples, let's apply **Auto Convex Collision** to our `SM_Lamp_Wall` mesh and set the **Max Hulls** and **Max Hull Verts** parameters to their default values of `4` and `12` respectively:



To really view the power of this tool, let's try applying **Auto Convex Collision** to the following parameters:

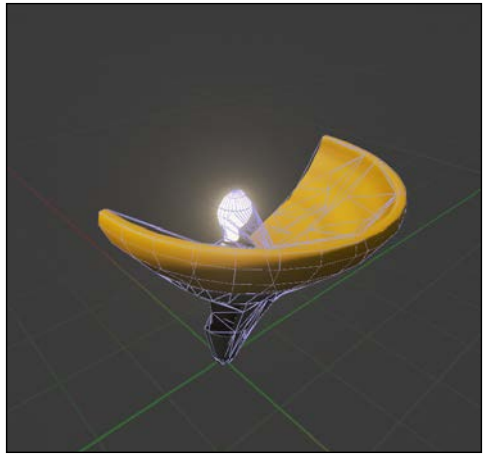
- **Max Hulls**: Set this parameter to `1`
- **Max Hull Verts**: Set this parameter to `6`

Then, we should see the following result:



As we can see, setting these parameters to the lowest values possible will result in a collision mesh that does the bare minimum and does not fit the size and shape of our lamp. Now, let's try applying **Auto Convex Collision** to the following parameters:

- **Max Hulls**: Set this paramerter to 24
- **Max Hull Verts**: Set this paramerter to 32



With the maximum settings applied to the convex collision mesh, we can see that it does a much better job of covering the lamp in terms of its size and shape. In the end, we would want to choose a setting somewhere between the lowest and highest values for the **Max Hulls** and **Max Hull Verts** parameters in order to create the most optimized collision possible for our assets.

Now that we have covered the methods of how to create collisions with the tools offered in the **Static Mesh** editor of Unreal Engine 4, we will now briefly discuss how to create and import collisions created in third-party art programs (such as 3ds Max or Maya).

The idea behind creating customized collision geometry is to make it as simple as possible in order to optimize collision detection when you play the game. The more complicated the collision geometry for an object, the more calculations are required by the engine to ensure that the collision is done correctly on that object. When you import the `.FBX` file to Unreal Engine 4, the collision meshes included in this file are identified by the importer based on their name. Here is the collision-naming syntax required to ensure proper collision when you import your assets to Unreal Engine 4:

- **UBX_[Mesh Name]**: This naming syntax is required when you import collision meshes that are box shaped, using either the box object type in 3ds Max or the cube primitive in Maya. Keep in mind that if you move any of the vertices of the box collision in the third-party art program or deform the shape in any way to make it anything other than a rectangular prism, the import will not work.
- **USP_[Mesh Name]**: This naming syntax is required when you import collision meshes that are sphere shaped, using the sphere object type in 3ds Max and Maya. The sphere itself in the third-party art program does not need to have a specific number of segments because it is converted to a true sphere for collision once it is imported to Unreal Engine 4.
- **UCX_[Mesh Name]**: This naming syntax is required when you import collision meshes that are convex shaped or a shape that is completely closed and does not have an interior angle of more than 180 degrees.

When you import the `.FBX` files that contain collision meshes, there are a few concepts that we have to keep in mind:

1. At the time of writing this book, spheres are only used for rigid-body collisions and Unreal's zero-extent traces, such as weapons, and not for instances (such as player movements).
2. In the naming syntax mentioned earlier, the **Mesh Name** component must be identical to the name of the mesh that the collision is associated with in the third- party art program. An example would be a box collision mesh for an object named `Chair_01` would be labeled as `UBX_Chair_01`, or if there are multiple collision meshes for this object, an additional collision mesh could be named `UBX_Chair_01_02`, and so on.

3. Once the collision meshes are created and named properly, we can export both the collision and the mesh that the collision is associated with in the `.FBX` file. Once imported, Unreal Engine 4 will find the collision, separate it from the actual mesh, and transform it into a collision model.
4. In the instance that an object has a collision composed of multiple shapes, the best results are found when the collision hulls do not intersect with one another.

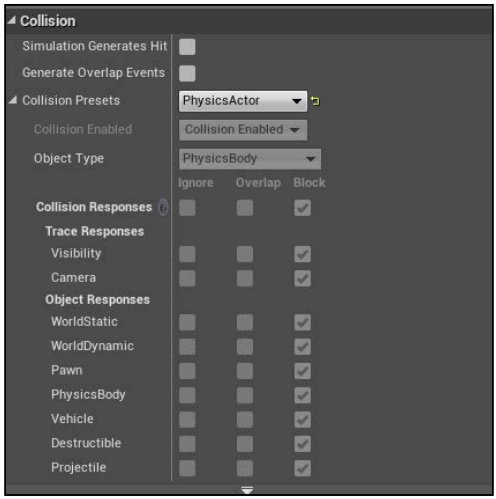
Creating complex and custom collision hulls – a section review

In this section, we took an in-depth look at how to create more complex collisions. We also looked at how to create custom collision hulls in third-party art programs (such as 3ds Max and Maya). Moreover, we analyzed the **Auto Convex Collision** tool in the **Static Mesh** editor of Unreal Engine 4. We also discussed how the **Max Hulls** and **Max Hull Verts** parameters affect the collision that is generated around our asset. Lastly, we looked at all the necessary naming conventions required in our art applications that ensure proper exporting and importing of our meshes and collisions to Unreal Engine 4. Now that we have a very strong understanding of how to generate collisions in Unreal Engine 4 and third-party art applications, we can now talk about about the different collision interactions that exist in Unreal Engine 4 in detail.

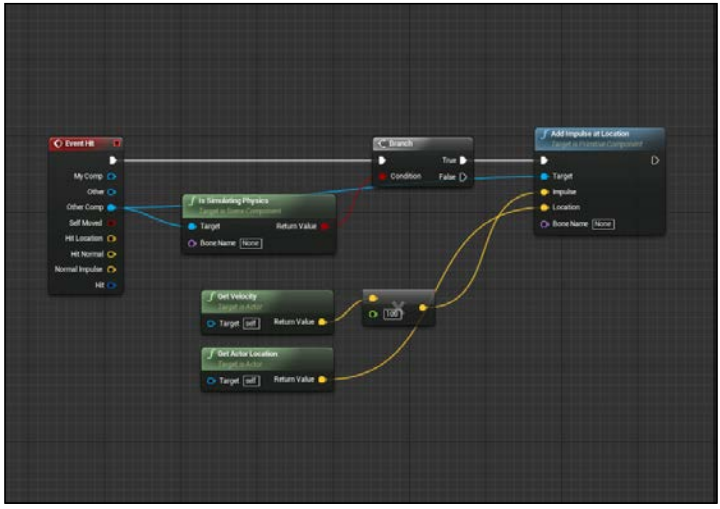
Collision interactions

After discussing a lot about what collision is and how to generate different types of collision, let's now talk about how the different collision responses function when you interact with the player and other objects in our game world. For the purposes of this section, we will want to have `Unreal_PhyProject` open, and we will work with the `FirstPersonExampleMap` level and use the default starter content to analyze these interactions.

In `FirstPersonExampleMap`, we will find numerous cube physics actors spread across the surface of the level (each starting awake and active at game time). If we select any of these cube actors in the editor by left-clicking on it, we will see the following **Collision** settings in its **Details Panel**:

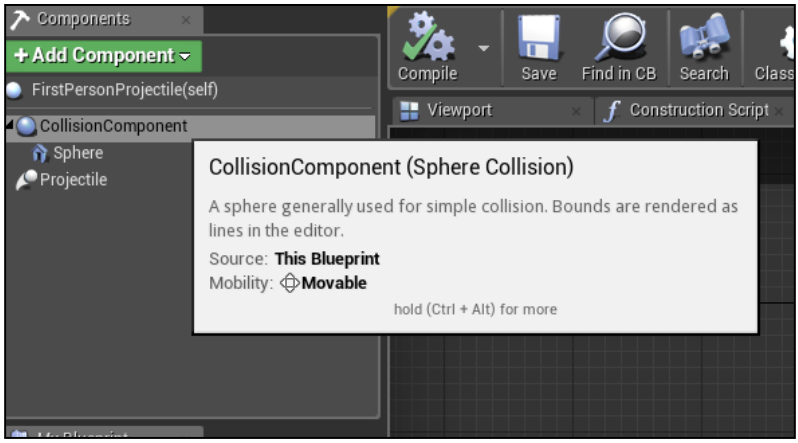


As we can see, these actors will use **Physics Actor** collision preset and have an **Object Type** of **PhysicsBody**. If we were to jump to the level with the first person project example (which we have in place) by pressing *Alt* + *P*, we can shoot these cubes with the **First-Person Projectile** blueprint by left-clicking on it. We can see that on colliding, there is an impulse created that causes the boxes to be pushed, and the collision itself causes the projectile to bounce off because it is also a physics object that has **Physics Body Object Response** set to **Block**. To get a better idea of what is happening, let's open the **First Person Projectile** blueprint by navigating to **Content Browser** and then to the `First Person BP` folder. In the `Blueprints` folder, we will find the **First Person Projectile** blueprint. Double-click on this asset to open its blueprint. It will bring us to the main **Event Graph**, as shown in the following screenshot:



If we are viewing the blueprint graph for the first time, this may be a little confusing, but we can easily break down the logic flow and understand exactly what the projectile will do once it's spawned and collides with an actor.

Let's first look at the main event of this graph: the **Event Hit** event node. What this event node checks for is whether or not the main root component of the blueprint is hit in a collision. In this case, the main root of this blueprint is the **Sphere Collision Component** option, labeled as **Collision Component**. Let's select this component by left-clicking on **Collision Component** in the **Components** tab in the top-left corner of the blueprint screen and then view its collision in **Details Panel**:

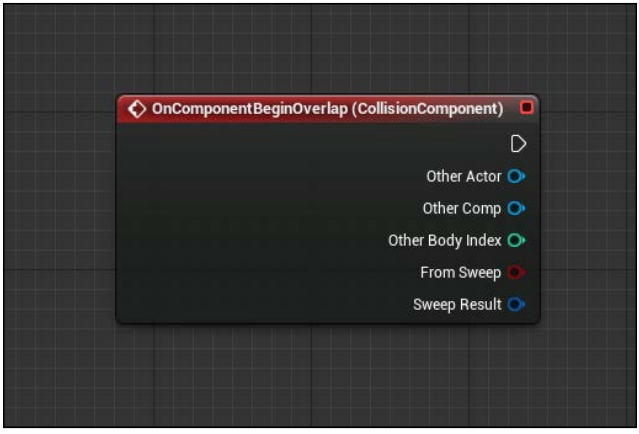


When you view its collision, you will see that it is set exactly similar to the physics cube actors in the level, possessing a **Physics Actor** collision preset and an **Object Type** set to **Physics Body**. What this means in terms of collision is that this projectile will act similar to a normal physics ball, such as a baseball or a basketball, when spawned into our game world. A ball in both the real world and our game world will more than likely end up hitting something, and when it does, our **Event Hit** node will be called.

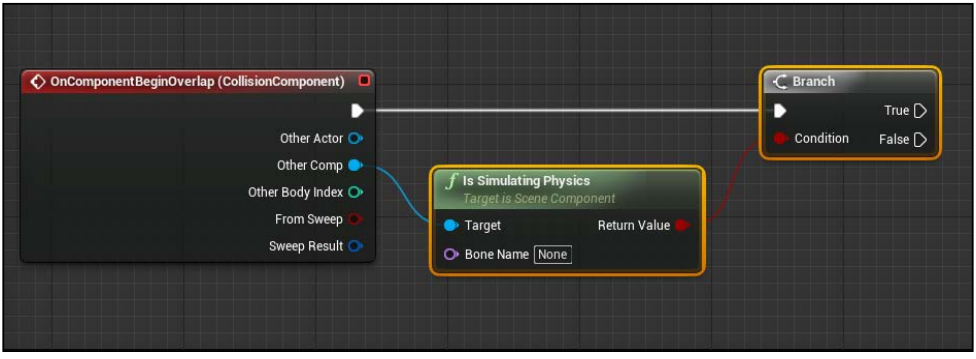
What happens next in the **First Person Projectile** blueprint is that it checks whether or not the other component that hits our projectile is a box, a wall, a player, or the floor. In particular, this blueprint will check whether the other hit component of any collision to this projectile is **Simulating Physics** or a physics actor. The **Is Simulating Physics** function node returns a **Boolean** value (**True** or **False**), irrespective of whether or not the other hit component is a physics actor. We then use a **Branch** node that uses this **True** or **False** condition from the **Is Simulating Physics** function to perform actions based on whether or not the hit component is a physics actor. We can see that from the **True** execution pin, we can add an impulse at the location of the projectile and use the **Other Component** of the **Hit** collision as our target to apply this impulse to. To determine the force of this impulse, we can perform a simple multiplication between the velocity vector of the projectile. We multiply it by a constant **Float** value. In addition to **Add Impulse at Location**, this math is what causes the physics cube to bounce or react to the projectile on collision, and **Is Simulating Physics** checks to ensure that no impulses are created when hitting the walls, the floor, or even the player. To have some fun with this blueprint, let's change the constant Float value from `100` to `1000` and see how it drastically changes the results when the projectile hits a physics object.

To change the way this projectile behaves in the game, we can change its collision preset from **Physics Actor** to **Custom** so that we can individually set how the collision interacts with the different object responses. For example, let's set the **Physics Body** object response from **Block** to **Overlap** and then compile the blueprint so that we can see the changes in the game. The result is that the projectile goes straight through the physics cube. However, it still reacts normally to the **World Static** object type (such as the floor and the walls). This is because we changed the object response to **Physics Body** from **Block** to **Overlap**. This causes the **Event Hit** event node to never get called.

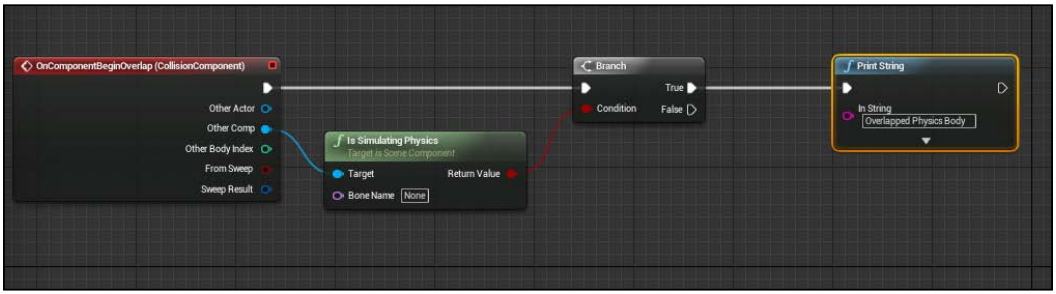
In the blueprints of Unreal Engine 4, there are event nodes we can use when objects overlap. This is called the **On Component Begin** overlap. As we made our projectile use the **Custom Collision** preset that overlaps the physics bodies in the game, we can use the **On Component Begin Overlap** event node to have any number of actions to take place during this collision. To set up a basic example in our projectile blueprint, select the **Collision Component** option in the **Components** tab so that it is highlighted. Next, right-click on an empty space of **Event Graph** and navigate to **Add Event** for **Collision Component** and then to **Collision**. Finally, select the **Add On Component Begin Overlap** event node.



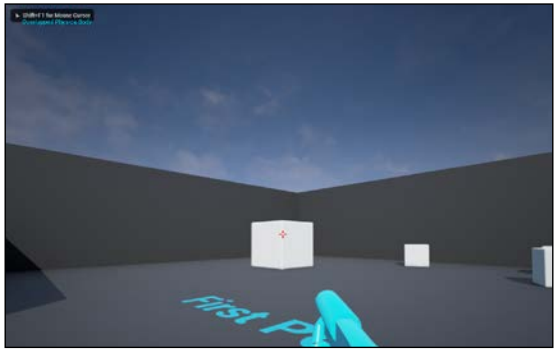
To have a similar behavior to what we had in the **Event Hit** node checking for objects that are physics actors, let's copy and paste the **Is Simulating Physics** function node and **Branch** that was used in the original blueprint logic and connect the nodes, as shown in the following screenshot:



So far, we had our projectile check for objects in our world that are physics actors once the collision is overlapped, but we have no actions taking place if this check is **True** or **False**. Instead of performing any kind of complicated actions, we will simply use the **Print String** function to print the dialogue to our console so that we know that the check is working. Let's right-click on the empty space of **Event Graph** and search for the **Print String** function node. In the **In String** parameter, enter `Overlapped Physics Body`, and connect it to the **True** executional output pin, as shown in the following screenshot:



If we click on the **Compile** button at the top of the blueprint and play the game, we can see that the projectile goes right through our physics objects, but we do not see our **Print String** outputted to the console. This is because both the physics cubes in the level and our projectile collision don't have the **Generate Overlap Events** parameter set to **True** by default, so let's select one or more of the cubes and navigate to their **Collision Settings** in **Details Panel** and make sure that **Generate Overlap Events** is set to **True**. Let's perform the same function to **Collision Component** in our projectile blueprint. Now, if we play again and shoot the physics cubes that we customized, we will now see our **Print String** outputted to the console.



From here, feel free to experiment and customize the collision presets that the projectile has, see how it changes and reacts in our game world, and add more blueprint functionalities to see what else is possible.

The last collision interaction that can exist between objects, apart from **Block** and **Overlap**, is the **Ignore** option. There is not much to this type of **Collision Response** because it will ignore the different **Object Responses** completely if it is set to **Ignore**. For the purpose of our example, in our **First Person Projectile** blueprint, let's change the **Physics Body** object response from **Overlap** to **Ignore**. If we play now, we can shoot at the physics cubes, but it will go completely through the object, and neither the **On Component Begin Overlap** nor the **Event Hit** event nodes will be called.

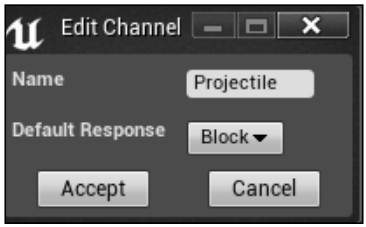
Collision interactions – a section review

In this section, we got our hands dirty by applying the different combinations of collision presets to the **First Person Projectile** blueprint to see how it interacts with the physics actors in our game world. By setting the **Physics Body** object response to **Block**, the **Event Hit** event node will be called. Also, an impulse will be created at the projectiles' location, resulting in a small push force applied to the physics actor the projectile collides with. By setting the **Physics Body** object response to **Overlap**, we can use the **On Component Begin Overlap Event** node to call different actions once the projectile overlaps with a physics actor. We just need to make sure that the physics actors in our level and our projectile have the **Generate Overlap Events** set to **True**. Lastly, we briefly discussed the results of when our projectile has the **Physics Body** object response set to **Ignore**. Like the name suggests, it ignores the object response, and no events are fired. Now that we talked more about collision interactions, let's move on and discuss how to create and use custom object and trace channel responses in Unreal Engine 4.

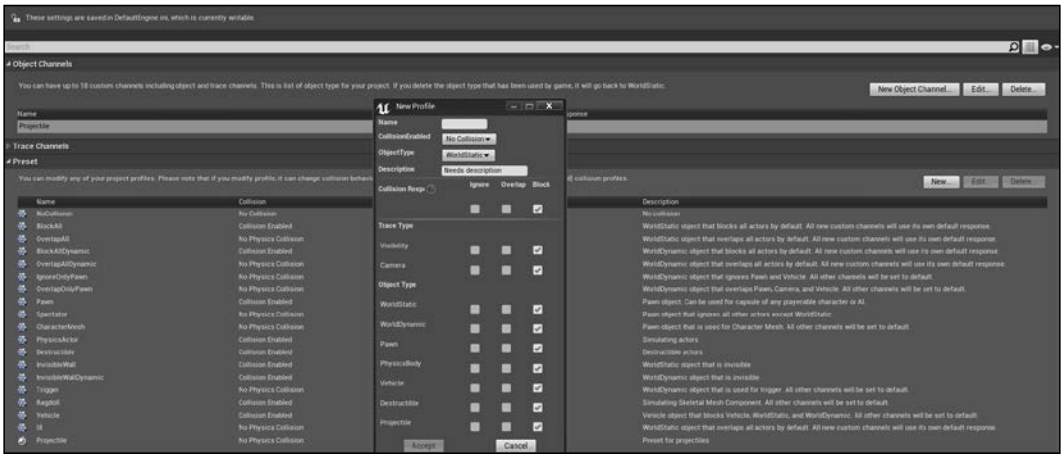
Custom object and trace channel responses

Sometimes, the default object and trace channel responses are not enough for what we want to do in our games. So, it may be necessary to create customized object and trace channel responses for certain assets and scenarios. To accomplish this, we can navigate to the **Edit** drop-down window at the top of the Unreal Engine 4 editor and select **Project Settings**. From here, select the **Collision** option under the **Engine** section. Here, we can create custom **Objects**, **Trace Channels**, and **Presets** that we can use when we apply collisions to our assets.

Let's start with creating a new **Object Channel** by selecting this option and clicking on the **New Object Channel** button. Here, a dialogue window pops up. Then, we can customize how **Object Channel** responds by default.



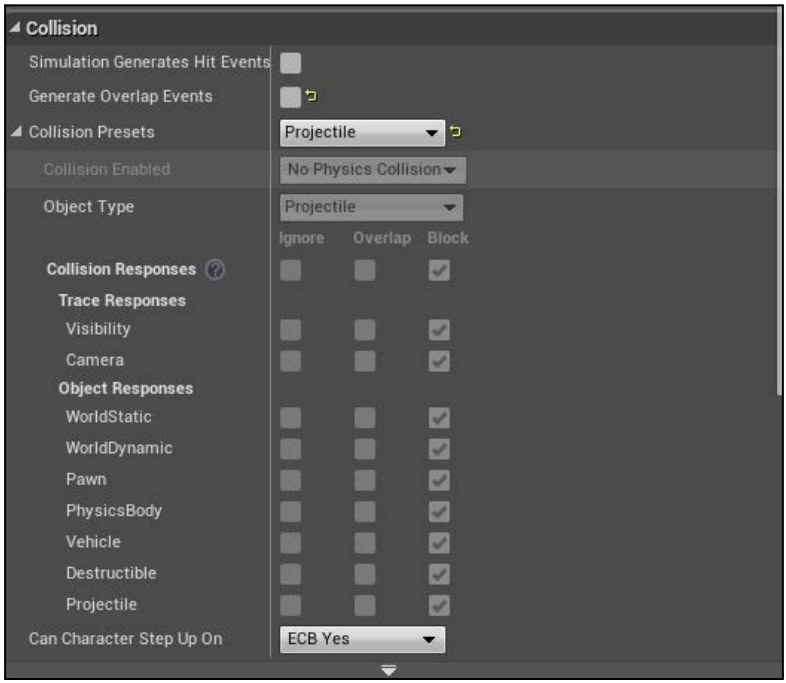
Name the **Object Channel** `Projectile`, and set its **Default Response** to **Block**. Now, let's create a custom **Collision Preset** by selecting this option and selecting the **New** button so that a dialogue window appears. Here, we can set the presets default values:



We can also name this new **Collision Preset Profile**, `Projectile`, set the **Collision Enabled** property to **Collision Enabled**, **Object Type** to `Projectile`, (the one that we have just created), and the **Description** property to anything that will remind us of what this **Collision Preset** is used for. Lastly, we can set all the **Trace** and **Object Channels** for this preset to **Block**.

For the purposes of this section's demonstration, we won't create a custom **Trace Channel**, but if we ever needed to, it works exactly similar to creating a custom **Object Channel**; select the **Trace Channels** option, left-click on the **New Trace Channel** button, name the channel, and set the **Default Response** to either **Block**, **Overlap**, or **Ignore**.

Now that we have created a custom **Object Channel** and a custom **Collision Preset**, let's apply these to the **Collision Component** of the **First Person Projectile** blueprint. Once you are in the **First Person Projectile** blueprint, select the **Collision** component from the **Components** tab in the top-left corner and navigate to the **Collision** section of the **Details Panel** in the bottom-right corner of the blueprint window. If we look at the **Collision Presets** drop-down menu, we will see our `Projectile` **Collision Preset** available, and when we select this option, we will see the default values that we set:



We can also see our `Projectile` **Object Channel Response** that we created earlier in the **Object Responses** section of the **Collision Component**. Now, if we ever need to, we can set all of our projectile assets to have the **Projectile Collision Preset** so that all of our projectiles behave the same during collisions. In addition to this, we can have other assets collide in a specific way to projectiles by setting the **Projectile** object response to **Block**, **Overlap**, or **Ignore** projectiles on colliding. Now, when we play the game, we can see that the **First Person Projectile** blueprint behaves exactly as intended when you fire the projectile.

Just as a reminder, we can only have up to 18 custom **Object Channels** and **Trace Channels**, and if we ever delete an **Object Type** that has been used in our game, it will revert back to **WorldStatic**, and if we delete a trace channel that has been used in our game, the behavior of the trace is undefined.

Custom object and trace channel responses – a section review

In this section, we took a deeper look at how to create and implement custom objects, trace channels, and custom collision presets. We then applied these customized channels and presets to the **First Person Projectile** blueprint and found that we can have the same collision behavior exist for the projectile when we use custom collision presets and **Object** channels. Now that we have created our very own custom object and trace channels and created our own collision preset, we can now move on and take an in-depth look at the additional default collision presets that exist in Unreal Engine 4.

In-depth collision presets

To conclude this chapter, let's briefly discuss the remaining collision presets available in Unreal Engine 4 that we have not gone through at this point:

- **Custom**: This collision preset allows you to fully customize how you want the collision to behave by selecting the **Collision Enabled** property, setting the **Object Type** property, and fully customizing how the **Trace** and **Object Responses** react to different types of collisions. This type of collision preset is useful when we need to customize an assets collision, and where the default collision presets does not fit the type of collision we need.
- **Block All Dynamic**: This collision preset blocks all the actors by default and makes the collision itself a **WorldDynamic** object. This type of collision preset is useful for dynamic objects or objects that can move in your game world that you want to block when you collide with other objects. Lastly, the **Collision Enabled** property is set to **Collision Enabled**.
- **Overlap All Dynamic**: This collision preset overlaps all the actors by default and makes the collision itself a **WorldDynamic** object. This type of collision preset is useful for dynamic objects that you want to overlap when you collide with other objects. Lastly, the **Collision Enabled** property is set to **No Physics Collision**, meaning that the assets' collision won't use game physics.
- **Ignore Only Pawn**: This collision preset blocks all the actors by default, but it ignores the **Pawn** and **Vehicle** object responses. This preset also sets the **Object Type** of the collision to **WorldDynamic** and is useful for assets that you want to ignore for **Pawns** and **Vehicles** in your game world. Lastly, the **Collision Enabled** property is set to **No Physics Collision**.

- **Overlap Only Pawn**: This collision preset blocks all the actors by default, but it overlaps the **Pawn** and **Vehicle** object responses. It also overlaps the **Camera** trace channel. This preset also sets the **Object Type** of the collision to **WorldDynamic** and is useful for assets that need overlap events to fire when it is overlapped during a collision with **Pawns**, **Vehicles**, and **Camera Traces**. Lastly, the **Collision Enabled** property is set to **No Physics Collision**.
- **Spectator**: This collision preset ignores all the actors by default, except the **WorldStatic** object responses. This preset also sets the **Object Type** of the collision to **Pawn** and is useful when you want players to see a game in multiplayer situations. Lastly, the **Collision Enabled** property is set to **No Physics Collision**.
- **Character Mesh**: This collision preset is a **Pawn Object Type** that is used for a **Character Mesh** when you create a player character. By default, this preset ignores the **Visibility** trace response and the **Pawn** and **Vehicle** object responses, although it blocks the remaining values. Lastly, the **Collision Enabled** property is set to **No Physics Collision**.
- **Destructible**: This collision preset is a **Destructible Object Type** that is used for assets that can be destructible in the game. By default, this preset blocks all the **Trace** and **Object Channels**. Its **Collision Enabled** property is set to **Collision Enabled**.
- **Invisible Wall**: This collision preset is a **World Static Object Type** that is used as an invisible wall that blocks all the **Trace** and **Object Responses**, except the **Visibility Trace Response**, which this preset ignores. This preset works exactly similar to a blocking volume. Lastly, its **Collision Enabled** property is set to **Collision Enabled**.
- **Invisible Wall Dynamic**: This collision preset is a **World Dynamic Object Type** that is used as an invisible wall that functions exactly similar to the **Invisible Wall** collision preset, in which it blocks all the **Trace** and **Object Responses**, except the **Visibility Trace Response**, which this preset also ignores. Lastly, its **Collision Enabled** property is set to **Collision Enabled**.
- **Trigger**: This collision preset is a **World Dynamic Object Type** that is used as a **Trigger**, meaning that it functions similar to a **Trigger Volume** so that we can use it to call all the events and functions in our game. By default, the **Trigger Collision Preset** overlaps all the **Trace** and **Object Channel Responses**, except the **Visibility Trace Channel**, which this preset ignores. Lastly, its **Collision Enabled** property is set to **No Physics Collision**.

- **Ragdoll**: This collision preset is used to simulate skeletal mesh components. Its **Object Type** is set to **Physics Body**. We can use this preset for character meshes that would turn rag doll when players are killed or lose control and would want the player character to be taken over by physics. By default, this collision preset blocks all the **Trace** and **Object Responses**, except the **Pawn Object Response**, which it ignores. Lastly, the **Collision Enabled** property is set to **Collision Enabled**.
- **Vehicle**: This collision preset is a **Vehicle Object Type** that is used for any moving vehicle assets in our game world. By default, this preset blocks all the **Trace** and **Object Responses**. Its **Collision Enabled** property is set to **Collision Enabled**.
- **UI**: This collision preset is a **World Dynamic Object Type** that is used for any **UI** assets (such as **UMG HUD** elements). By default, this preset overlaps all the **Trace** and **Object Responses**, except the **Visibility Trace Response**, which this preset blocks. Lastly, the **Collision Enabled** property is set to **No Physics Collision**.

In-depth collision presets – a section review

In this section, we took an in-depth look at all the collision presets that Unreal Engine 4 provides users by default, and by doing so, we analyzed the purpose and functionalities of each. Now that we have covered collision presets, we can now conclude this chapter and move on to discussing constraints in Unreal Engine 4.

Summary

In this chapter, we discussed how collision works and how it is implemented in Unreal Engine 4 by first analyzing the topics of trace and collision responses. We also discussed how these responses work, their parameter values, and how to implement these responses to our blueprint assets in detail.

Next, you learned about simple and complex collisions by defining what each type is and how they are used. We also looked at its pros and cons and how to generate the different types of simple collision.

Additionally, you learned more about complex collisions and how to generate these types of collisions in Unreal Engine 4. You also looked at how to create custom collision hulls in third-party art programs.

Furthermore, you learned about collision interactions. We used the **First Person Projectile** blueprint as an example of how these interactions are used when it comes to scripting different behaviors for our assets.

Moreover, we went through the purposes of custom objects and trace channels. We discussed how to create custom collision presets, including how to implement these customized parameters in blueprints.

Lastly, we discussed the different collision presets that exist in Unreal Engine 4, their purposes, and how they function in detail.

Now that we have a stronger understanding of how collision works and how to implement the different collisions for our assets, we can now dive deep into creating constraints in Unreal Engine 4.

4
Constraints

What are constraints?

Constraints are basic physical actors in Unreal Engine 4. Imagine that physical rules need something to be presented with, such as a tool, switch, calculator, or container. It's called **physic actor**, and physic actors that are responsible for simulating the physical behavior between two objects are known as physics constraint actor.

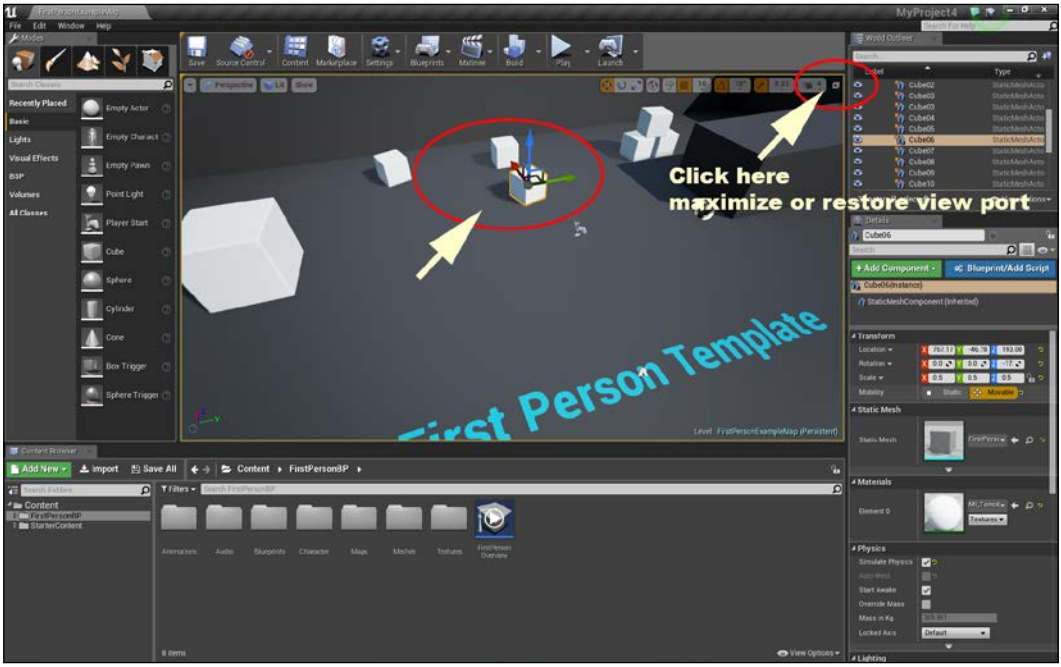
It works in a similar way to a minicomputer. You can connect two inputs to it, and it calculates how these two should play with the physical rules in the game. Finally, it applies the changes to the object as real-time processing during the game play.

The first physics constraint actor experience

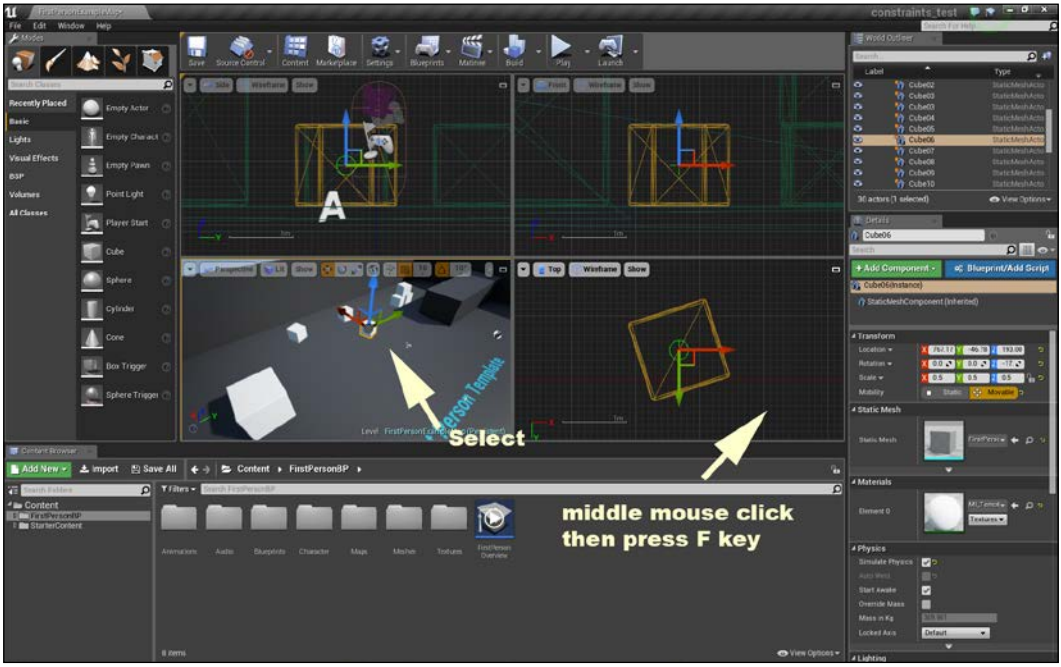
Before we start working in Unreal Editor, we will need to have a project to work with. Perform the following steps:

1. First, open Unreal Editor by clicking on the **Launch** button from Unreal Engine launcher.
2. Start a new project from **Project** browser by selecting the **New Project** tab. Select **First Person** and make sure that **With Starter Content** is selected and give the project a name (`constraints_test`).

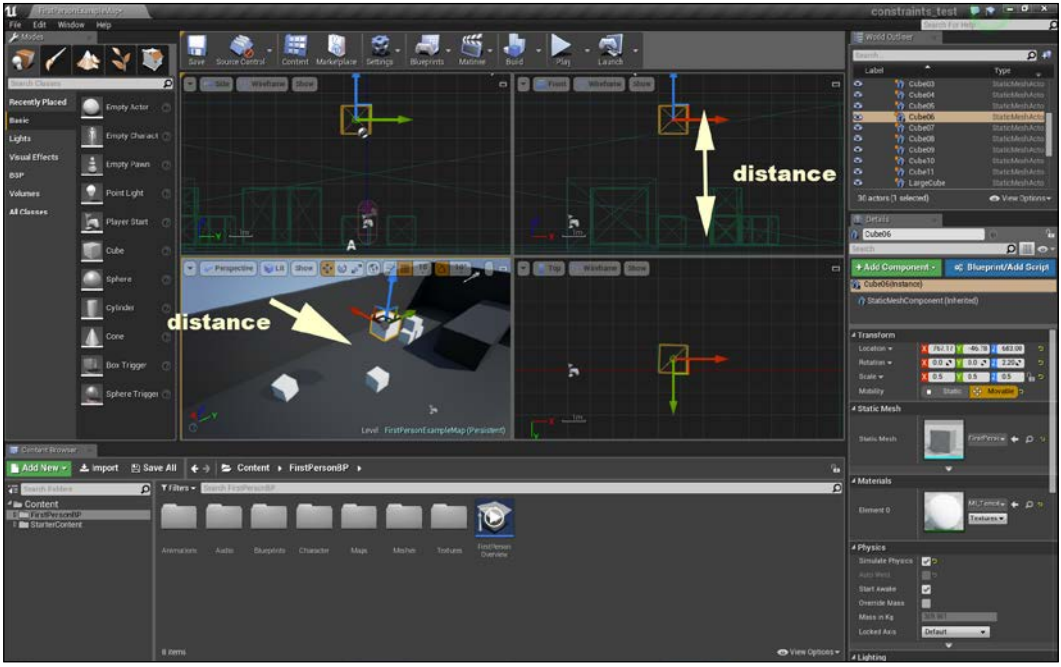
3. Once you are finished, locate the two cubes on your view. Select one and then click on the small icon in the top-right corner of your view:



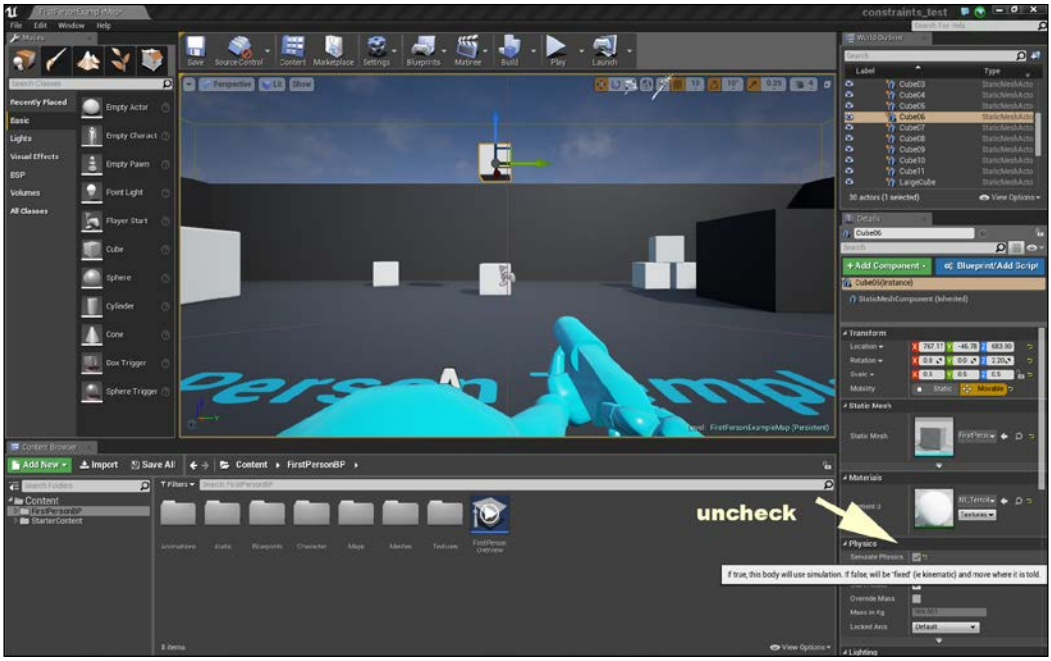
4. Now, you have four viewports on your screen. It's good practice when you are working with constraints to check your stage from four views. Now, select one of the cubes, middle-click on the top view, and press the *F* key. This automatically navigates all your viewports so that they are all focused on your actual selection.



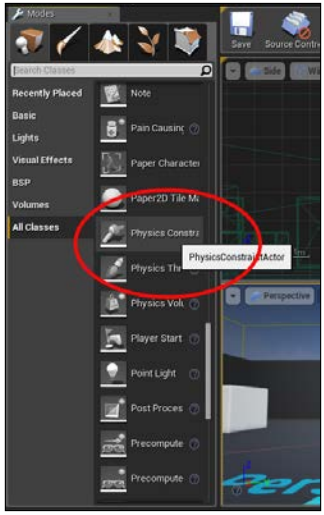
5. Now, we need to move one cube onto another one. Also, rotate the upper cube so that it fits the same angle as the one after it. Use the move and rotate tools to create your stage, as shown in the following screenshot. As you can see, the upper cube is about three times further away from the lower cube:



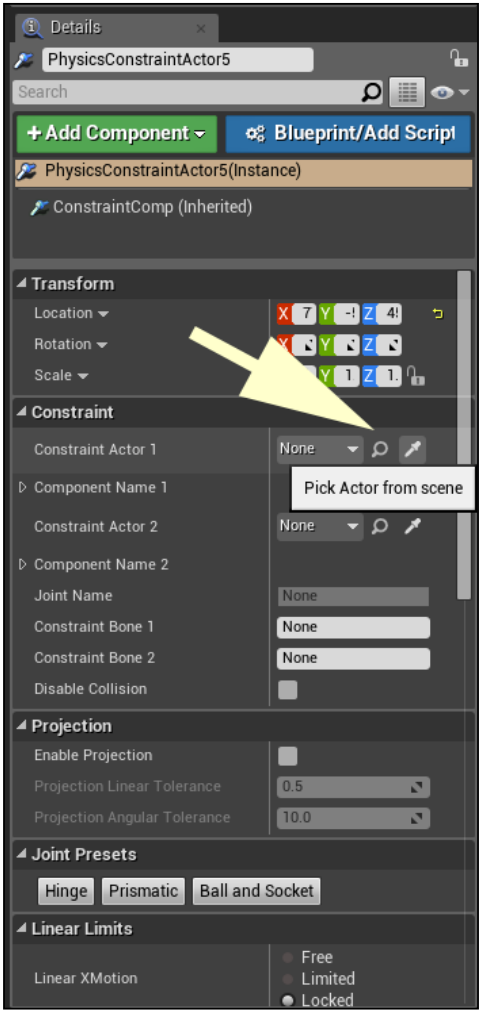
6. Let's click on the area in the top-right corner of your **Perspective** view to expand it and press **Play**. You will see one cube just fall on the lower cube. This is caused by gravity. Also, you can move the cubes by shooting balls at them. We need to disable this physic rules for the upper cube. Now, click on **Stop** to exit the play mode. Select the upper cube and uncheck **Simulate Physics** in **Details** on the right-hand side of your editor. Then, click on **Play** again.



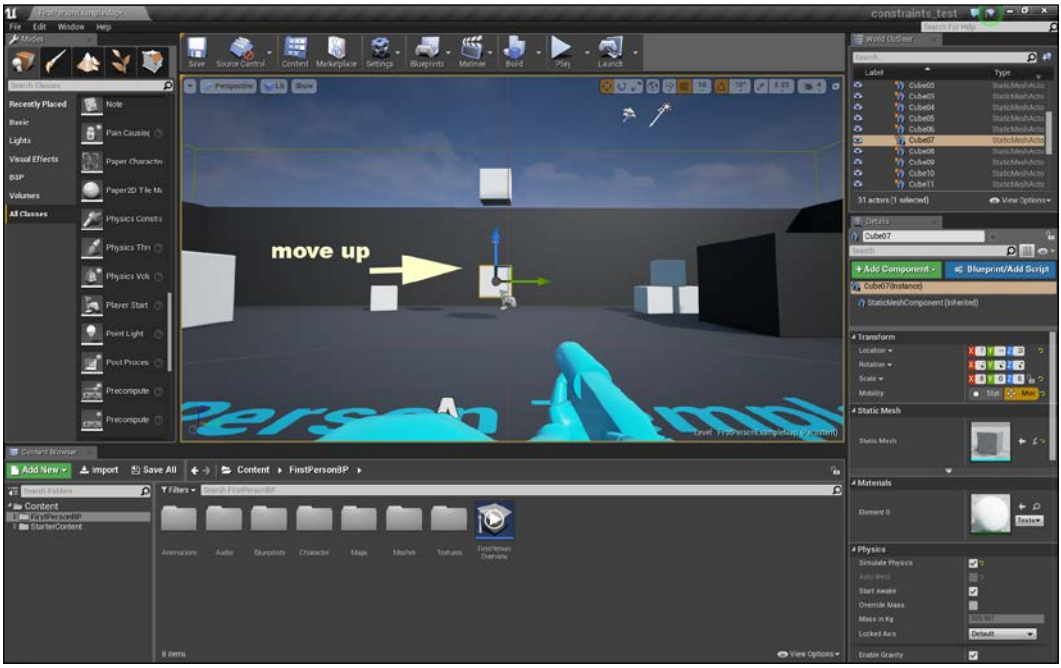
7. Now, the upper cube not only falls, but also, when you shoot at it, it doesn't move. This gives an ideal location to hang the other cube to this one. For this purpose, Unreal Engine defines some tools (known as **PhysX Constraint**). These tools allow you to simulate reality based on the physical behavior and the mode of the game world. Then, switch back to four views, click on **All Classes** in **Modes**, and locate **PhysicsConstraintActor** near **All Classes**:



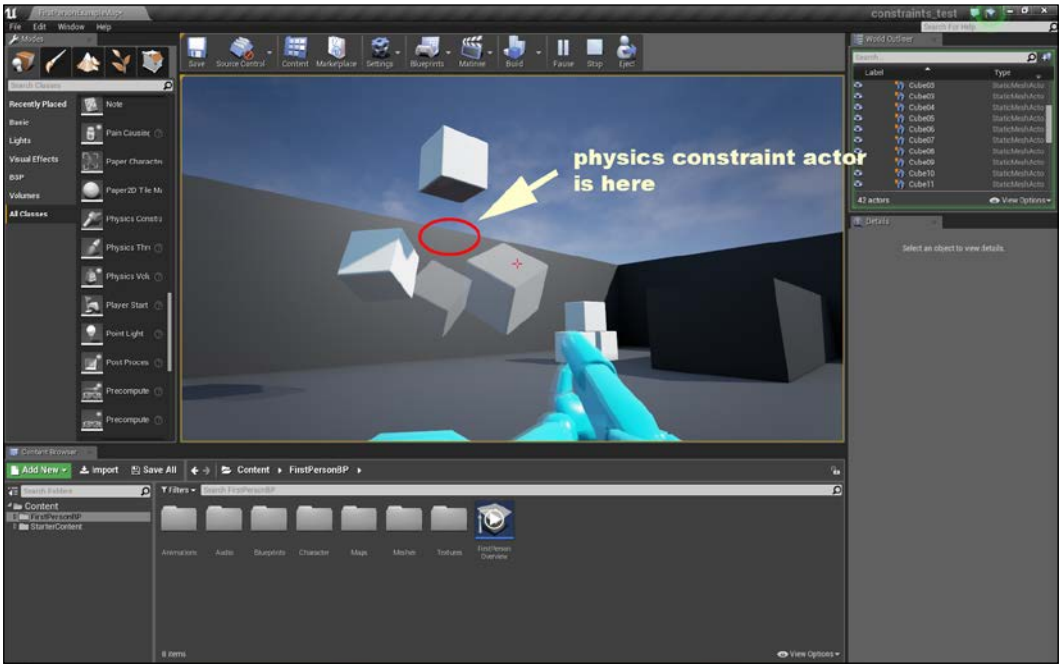
8. Drag and drop this actor and place it in the middle of the cubes. Locate the **Constraint** section from the right-hand side list. Each physics constraint actor needs two objects to operate with. This means that we need to give the actor object names and then this actor presents a physical-based action in the game world during the game play. How we do this? You will find two similar actors in the **Constraint** section: **Constraint Actor 1** and **Constraint Actor 2**. Here, we can define objects. Simply click on the picker icon and then on the upper cube for **Constraint Actor 1** and another cube for **Constraint Actor 2**, as shown in the following screenshot:



9. Now, switch to the **Perspective** view, move the lower cube a bit up, press **Play**, and shoot the lower cube from different angles and locations, as shown in the following screenshot:



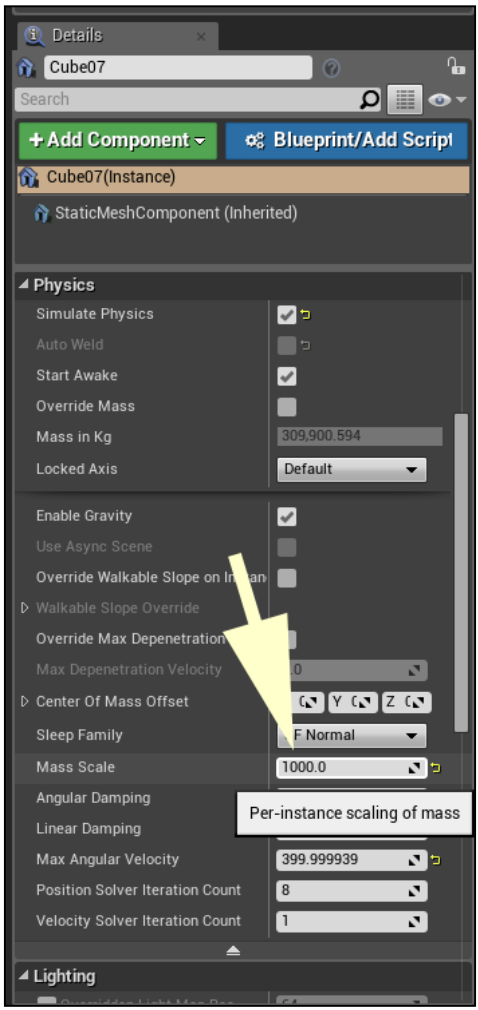
10. As you can see, it looks like something is grabbing the lower cube and forcing it to remain there and rotate around that point. Also, the cube shows normal behavior with gravity.



What you observe here is how physic constraints work in Unreal Engine. You can change the physical parameters related to each object and obtain different results. Let's take a look at some:

1. For the upper cube, check **Physics** in the **Details** window. When you play, it seems that all the cubes are connected to each other, whereas when you move one, others react to your interaction. Click on **Stop** and uncheck **Physics** for the upper cube.

2. Select the lower cube. Then, in **Physics**, change **Mass Scale** from `1.0` to `1000`. Now, play the game and try to move the cube. It looks like it's heavy, very heavy. So, click on **Stop** and change **Mass Scale** to `1.0`.

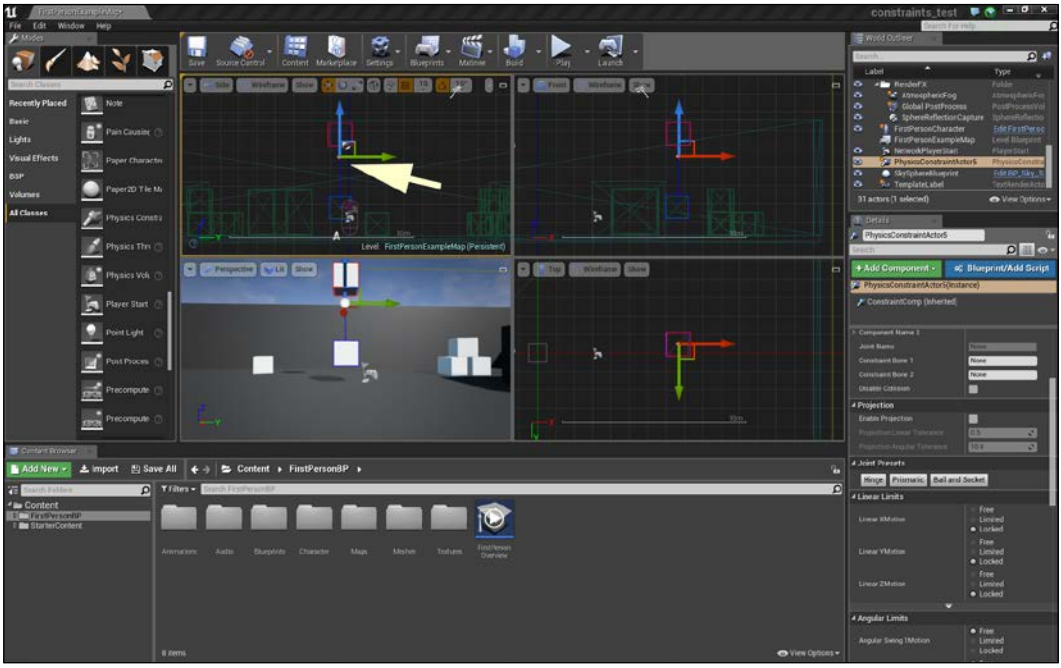


3. Select the lower cube, uncheck **Enable Gravity**, and click on **Play**. Try to shoot the cube and observe the difference. It saves its connection to other objects, but doesn't follow the gravity of the game world.

4. Then, press **Stop** and check **Enable Gravity**.

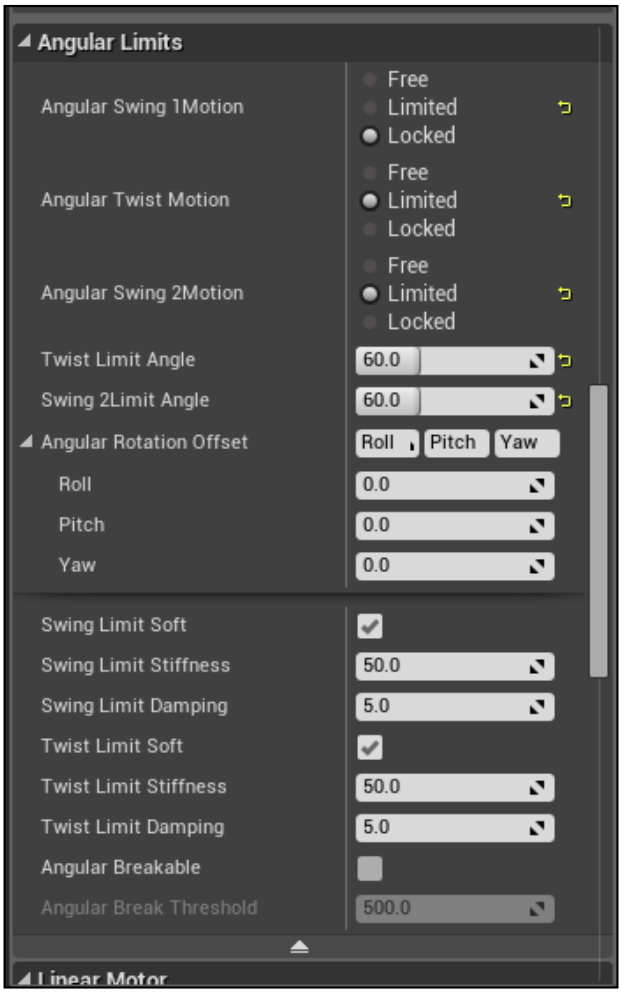
Customizing physics constraint actor

Now we know that this actor connects two objects to each other by supporting the gravity and physical aspects of each object. We also know that it is invisible to players. In addition, **Physics Constraint Actor** has some properties that strongly shape the physical behavior of objects. Let's start with position. Switch back to four views and move your **Physics Constraint Actor** close to the upper cube, as shown in the following screenshot:



Now, press **Play** and shoot the cube. As you can see, it even goes over the upper cube. It seems that the rotation point is going higher. This is correct. The exact description is that you move your physics constraint actor to the top. This basically changes the way these two objects behave based on physical rules.

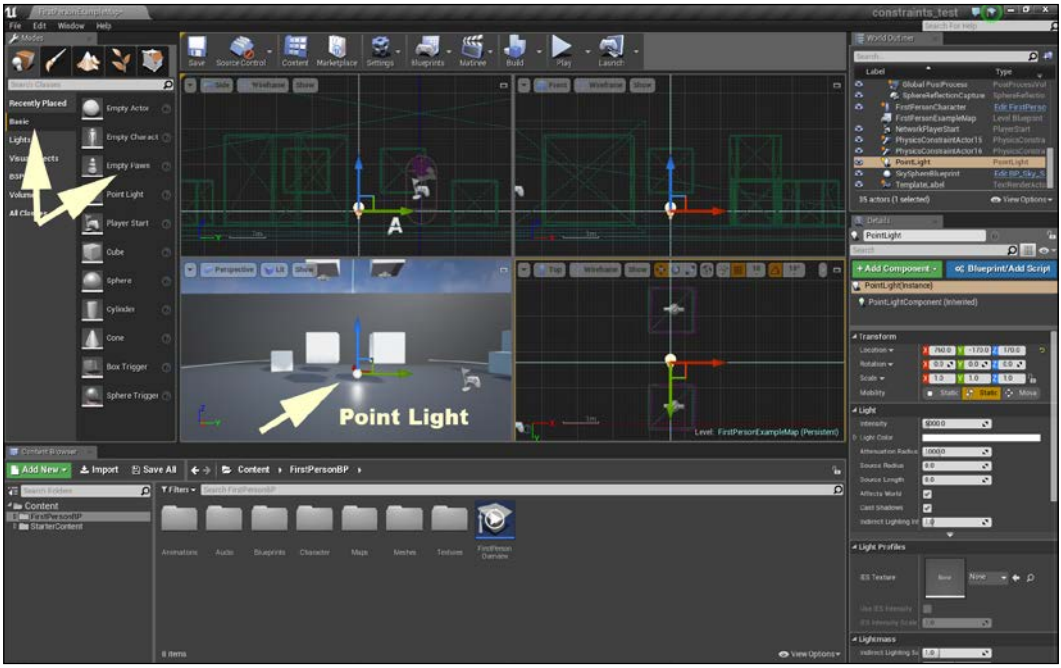
Now, click on **Stop**. Then, click on your physics constraint actor and press *F*. In **Angular Limits**, in the **Details** section on the right-hand side, set **Angular Swing 1Motion** to **Locked**, **Angular Twist Motion** to **Limited**, and **Twist Limit Angel** and **Angular Swing 2Motion** to **60**:



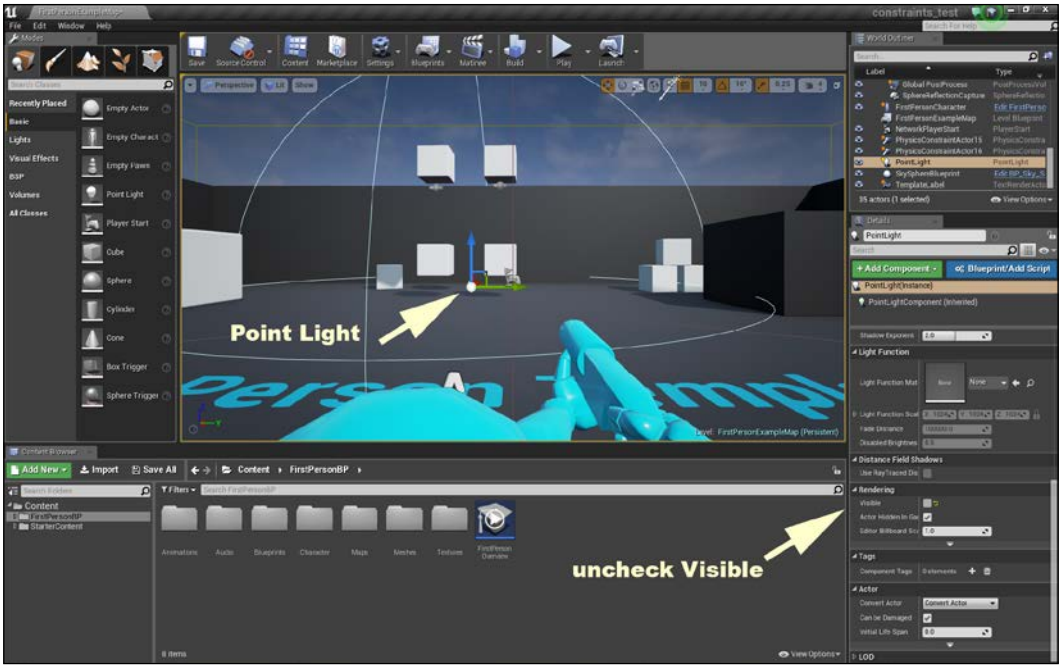
Now, click on **Play**. As you can see, the cube plays in a pyramid-like area between 60 degrees and nowhere else. Then, click on **Stop** and change **Angular Swing 1Motion** to **Free**. Finally, click on **Play**. As you can see, the cube has free rotation on its *z* axis.

A simple game with Blueprint

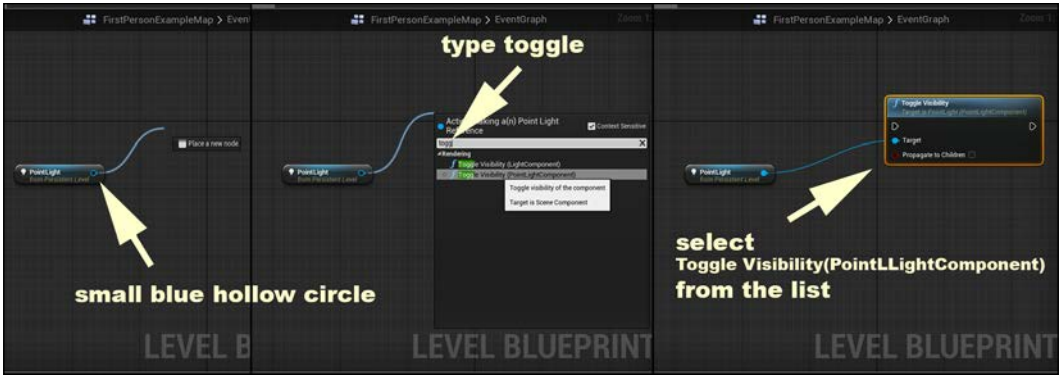
After you click on **Stop**, select the cubes and your actor. Then, hold *Alt* and create a copy of all by moving on the *Y* axis. Now, click on the second actor and pick the upper and lower cubes. Then, select one point light object from **Modes** | **Basic** on the left-hand side and place it close to the stage surface between the two cubes. Our goal is to turn on the light when the cubes experience collision and break one of the cube's physical constraints.



Now, click on point light and navigate to **Rendering** in the **Details** section and uncheck **Visible**:



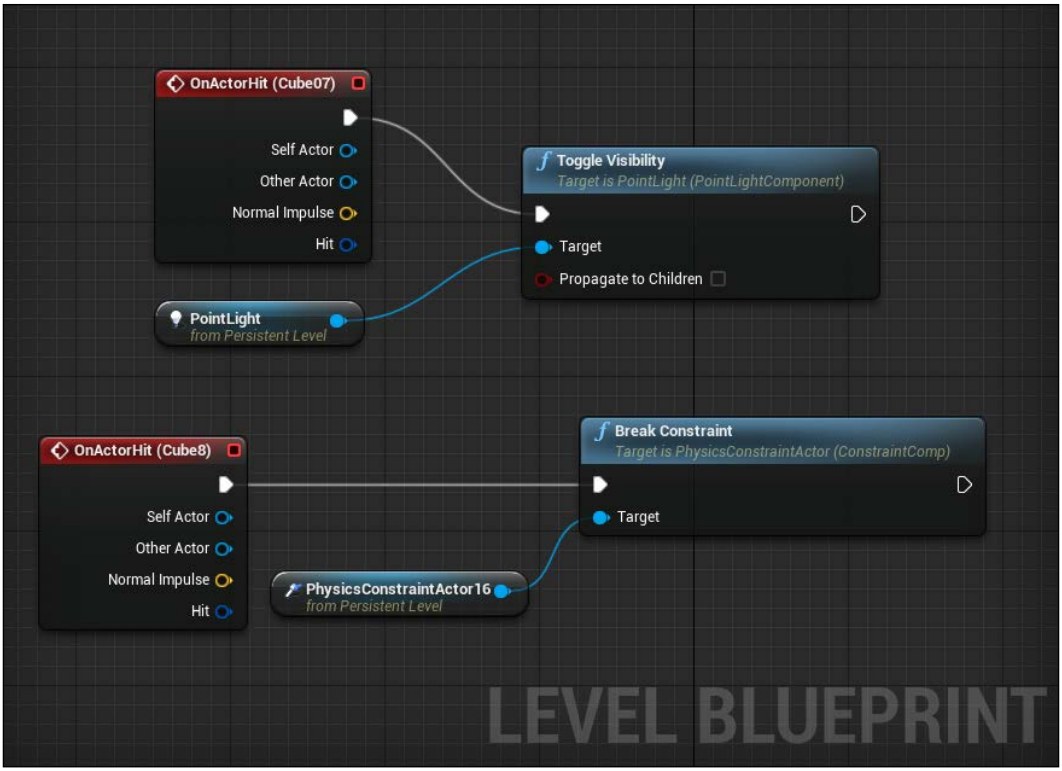
Now, click on **Blueprint** and select **Open Level Blueprint**. Navigate back to editor and select your point light. Now, go back to the **Blueprint** editor, right click on it, and select **Create a Reference to PointLight** from the list. You will have a curvy box with the **PointLight** title on your blueprint stage. Now, locate the small blue hollow circle in the top-right corner of your point light box. Then, click and hold the mouse and move the pointer to the right. You will find a wire-like line that follows your mouse. When you leave the mouse, a list will appear with a search area to type. Enter toggle and click on **Toggle Visibility (PointLightComponent)** from the names in the list, as shown in the following screenshot:



Now, navigate back to editor and select one of the lower cubes. Then, go back to **Blueprint** and use the same method to open the list, but this time, open **Add Event** for your cube name. Then, open **Collision** and select **Add On Actor Hit**. Now, connect a wire to the previous box you made, as shown in the following screenshot:

What this does is you first define the reference to your **PointLight** and then add the **Toggle** functionality to the visibility of the light. However, you need to turn the light on by selecting **Hit Event** for your cubes. So, we can add **Event** for collision and set the event to execute our toggle function. Click on **Play** and check how it works. As you can see, **PointLight** turns on and off on each hit to the selected cube.

Now, we need to define another rule for our game. We need to disable the physical actor on the hit event. Then, select your actor over the other cube. Create a reference for it as you did for the lamp. Then, create the **Break Constraint** function with the break keyword on the search area. Now, select the cube that responds to the actor and then create another hit event and connect it to this function. This is how the result should look:



Now, test the game. As a result of the heat effect, one cube turns the light on/off, whereas another falls on the game stage.

Summary

The way game designers and artists try to project the mode and presentation of objects in the game meets the frame by frame animation, or wisely use the physical aspects of Unreal Engine.

Twenty years ago, game designers animated the open and close movement of a simple door with the frame by frame method as an image-sequence file (such as GIF). Now, we can do the same in real time with light effects, materials, physical rules, and the blueprint code. Unreal Engine 4 provides the detailed properties and customizations for movements and dependency between objects in the game world. This is how creativity meets artistic details in a game design. Working with constraints is a kind of creative art.

5
Physics Damping, Friction, and Physics Bodies

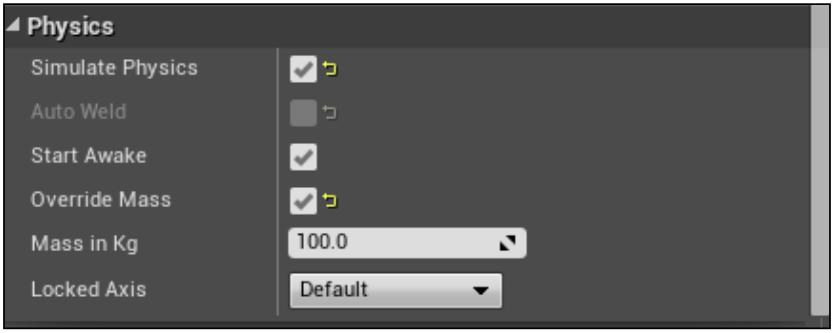
In this chapter, we will take a deeper look at **Physics Bodies** in Unreal Engine 4 and analyze how the engine uses physics properties, such as **Angular Damping** and **Linear Friction** to simulate real-world physics in our game. To start with, we will examine what **Physics Bodies** are. We will also look at some of the detailed properties available to these assets. In addition, we will discuss the following topics:

- Angular and linear friction
- Physical materials – an overview
- Physics damping

For the purposes of this chapter, we will continue to work with Unreal Engine 4 and the **Unreal_PhyProject** that we created in *Chapter 1*, *Math and Physics Primer* in Unreal Engine. Let's begin by discussing **Physics Bodies** in Unreal Engine 4.

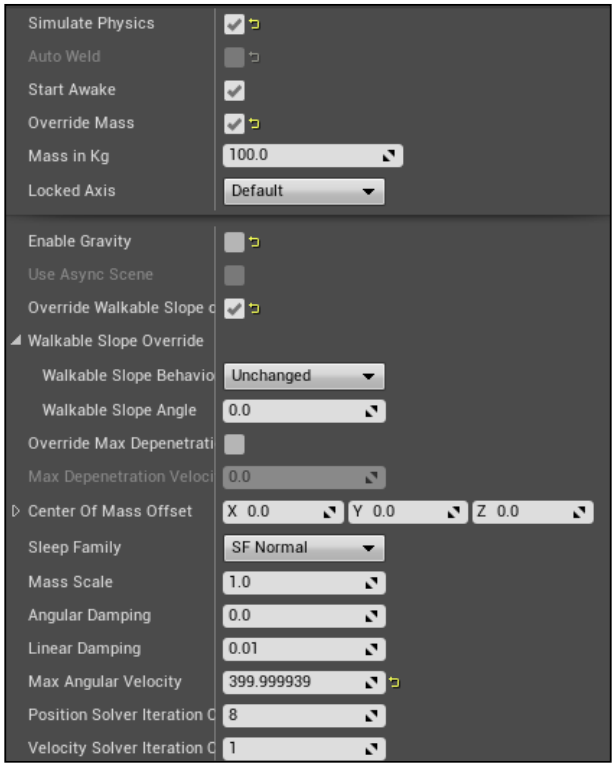
Physics Bodies – an overview

When it comes to creating **Physics Bodies**, there are multiple ways to go about it (most of which we have covered up to this point), so we will not go into much detail about the creation of **Physics Bodies**. We can have **Static Meshes** react as **Physics Bodies** by checking the **Simulate Physics** property of the asset when it is placed in our level:



We can also create **Physics Bodies** by creating **Physics Assets** and **Skeletal Meshes**, which automatically have the properties of physics by default. Lastly, **Shape Components** in blueprints, such as spheres, boxes, and capsules will automatically gain the properties of a Physics Body if they are set for any sort of collision, overlap, or other physics simulation events. As always, remember to ensure that our asset has a collision applied to it before attempting to simulate physics or establish **Physics Bodies**, otherwise the simulation will not work.

When you work with the properties of **Physics** on **Static Meshes** or any other assets that we will attempt to simulate physics with, we will see a handful of different parameters that we can change in order to produce the desired effect under the **Details** panel.



Let's break down these properties:

- **Simulate Physics**: This parameter allows you to enable or simulate physics with the asset you have selected. When this option is unchecked, the asset will remain static, and once enabled, we can edit the **Physics Body** properties for additional customization.
- **Auto Weld**: When this property is set to `True`, and when the asset is attached to a parent object, such as in a blueprint, the two bodies are merged into a single rigid body. **Physics** settings, such as collision profiles and body settings, are determined by **Root Component**.
- **Start Awake**: This parameter determines whether the selected asset will **Simulate Physics** at the start once it is spawned or whether it will **Simulate Physics** at a later time. We can change this parameter with the level and actor blueprints.
- **Override Mass**: When this property is checked and set to `True`, we can then freely change the **Mass** of our asset using **kilograms** (**kg**). Otherwise, the **Mass in Kg** parameter will be set to a default value that is based on a computation between the physical material applied and the mass scale value.

- **Mass in Kg**: This parameter determines the **Mass** of the selected asset using kilograms. This is important when you work with different sized physics objects and want them to react to forces appropriately.
- **Locked Axis**: This parameter allows you to lock the physical movement of our object along a specified axis. We have the choice to lock the default axes as specified in **Project Settings**. We also have the choice to lock physical movement along the individual *X*, *Y*, and *Z* axes. We can have none of the axes either locked in translation or rotation, or we can customize each axis individually with the **Custom** option.
- **Enable Gravity**: This parameter determines whether the object should have the force of gravity applied to it. The force of gravity can be altered in the **World Settings** properties of the level or in the **Physics** section of the **Engine** properties in **Project Settings**.
- **Use Async Scene**: This property allows you to enable the use of **Asynchronous Physics** for the specified object. By default, we cannot edit this property. In order to do so, we must navigate to **Project Settings** and then to the **Physics** section. Under the advanced **Simulation** tab, we will find the **Enable Async Scene** parameter. In an asynchronous scene, objects (such as **Destructible** actors) are simulated, and a **Synchronous** scene is where classic physics tasks, such as a falling crate, take place.
- **Override Walkable Slope on Instance**: This parameter determines whether or not we can customize an object's walkable slope. In general, we would use this parameter for our player character, but this property enables the customization of how steep a slope is that an object can walk on. This can be controlled specifically by the **Walkable Slope Angle** parameter and the **Walkable Slope Behavior** parameter.
- **Override Max Depenetration Velocity**: This parameter allows you to customize **Max Depenetration Velocity** of the selected physics body.
- **Center of Mass Offset**: This property allows you to specify a specific vector offset for the selected objects' center of mass from the calculated location. Being able to know and even modify the center of the mass for our objects can be very useful when you work with sensitive physics simulations (such as flight).
- **Sleep Family**: This parameter allows you to control the set of functions that the physics object uses when in a sleep mode or when the object is moving and slowly coming to a stop. The **SF Sensitive** option contains values with a lower sleep threshold. This is best used for objects that can move very slowly or for improved physics simulations (such as billiards). The **SF Normal** option contains values with a higher sleep threshold, and objects will come to a stop in a more abrupt manner once in motion as compared to the **SF Sensitive** option.

- **Mass Scale**: This parameter allows you to scale the mass of our object by multiplying a scalar value. The lower the number, the lower the mass of the object will become, whereas the larger the number, the larger the mass of the object will become. This property can be used in conjunction with the **Mass in Kg** parameter to add more customization to the mass of the object.
- **Angular Damping**: This property is a modifier of the drag force that is applied to the object in order to reduce angular movement, which means to reduce the rotation of the object. We will go into more detail regarding **Angular Damping** later in this chapter.
- **Linear Damping**: This property is used to simulate the different types of friction that can assist in the game world. This modifier adds a drag force to reduce linear movement, reducing the translation of the object. We will go into more detail regarding **Linear Damping** later in this chapter.
- **Max Angular Velocity**: This parameter limits **Max Angular Velocity** of the selected object in order to prevent the object from rotating at high rates. By increasing this value, the object will spin at very high speeds once it is impacted by an outside force that is strong enough to reach the **Max Angular Velocity** value. By decreasing this value, the object will not rotate as fast, and it will come to a halt much faster depending on the angular damping applied.
- **Position Solver Iteration Count**: This parameter reflects the physics body's solver iteration count for its position; the solver iteration count is responsible for periodically checking the physics body's position. Increasing this value will be more CPU intensive, but better stabilized.
- **Velocity Solver Iteration Count**: This parameter reflects the physics body's solver iteration count for its velocity; the solver iteration count is responsible for periodically checking the physics body's velocity. Increasing this value will be more CPU intensive, but better stabilized.

Now that we have discussed all the different parameters available to **Physics Bodies** in Unreal Engine 4, feel free to play around with these values in order to obtain a stronger grasp of what each property controls and how it affects the physical properties of the object. As there are a handful of properties, we will not go into detailed examples of each, but the best way to learn more is to experiment with these values. However, we will work with how to create various examples of physics bodies in order to explore **Physics Damping** and **Friction** later in this chapter.

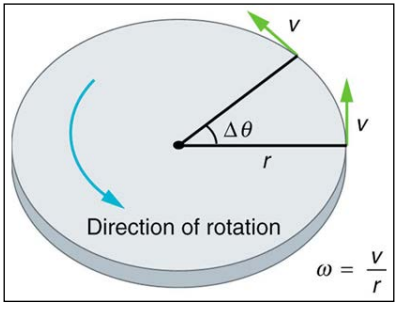
Physics Bodies – a section review

In this section, we discussed what **Physics Bodies** are and how to create them by using static meshes, skeletal meshes, shape components in blueprints, and physics assets. Additionally, you learned about all the different parameters that exist for **Physics Bodies** in Unreal Engine 4. We also looked at each property and how it affects the physics on the specified object. Now that we have a stronger grasp of what **Physics Bodies** are, let's move on and discuss **Angular** and **Linear Damping**.

Angular and Linear Damping

In this section, we will discuss **Angular** and **Linear Damping** in more detail, focusing on the friction properties of physics bodies. Further more, we will discuss physics damping and how this can be used when setting up the constraints for our blueprints. Let's begin by briefly discussing **Angular Damping** and **Angular Velocity/Momentum**.

In the realm of physics, **Angular Velocity** is defined as the rate of change of angular displacement, also known as a vector quantity, which specifies the angular speed or the rotational speed of an object and the axis in which the object is rotating.



In the preceding diagram, we can see that **w,** or the angular speed, is equal to the velocity divided by the radius of the object that is rotating.

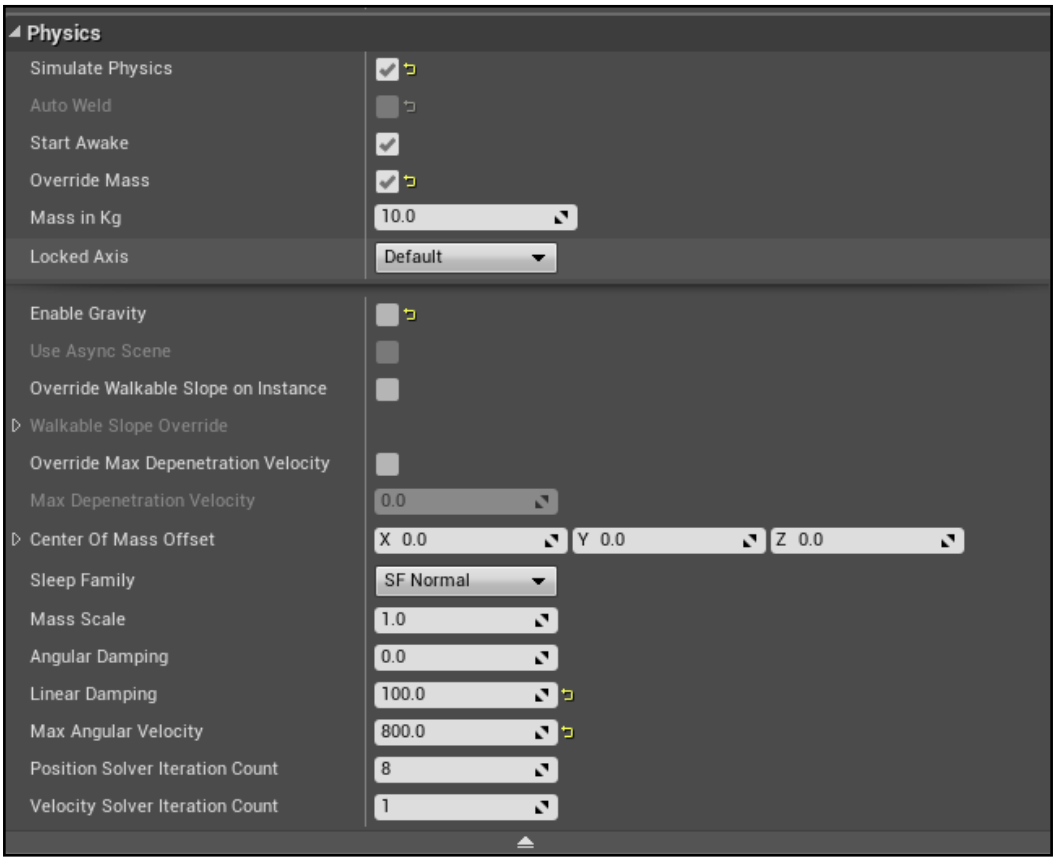
Linear Angular Momentum is proportional to **Moment of Inertia** (*I*) and **Angular Speed** (*w*), so the basic formula is *L = Iw*. We now know that *w* is equal to the velocity and radius of the object, so we can now write the expression as *L = I (v/r)*. Lastly, **Moment of Inertia** is equal to the radius of the object squared, multiplied by the object's mass, or *I = r^2 * (m)*. With this in mind, we can now rewrite the **Linear Angular Momentum** expression as *L = (r^2 * m) * (v/r)*, or more simply, we can write the expression as *L = rmv*, so the **Linear Angular Momentum** of an object is equal to the radius of the object multiplied by its mass and the velocity that is applied to it.

Now that we have a stronger understanding of **Angular Velocity** and **Angular Momentum**, let's apply this knowledge to the physics body with the **Unreal_PhyProject** game project that we created in the first chapter. By default, there are a bunch of physics body cubes placed throughout the **FirstPersonExampleMap** level that we can alter properties to in order to explore **Angular Damping** and **Angular Velocity**.

Let's start by selecting a random cube from **FirstPersonExampleMap** and viewing its **Physics** properties in **Details** panel, on the left-hand side. For the sake of an example, let's left-click on the down directional arrow in the **Physics** section to expand the advanced properties. The first property we will see is **Enable Gravity**, and by default, it is set to `True`; let's set this property to `False` so that the selected physics body doesn't have the force of gravity applied to it. With this change in place, we can now move the physics body upwards in the *Z* direction so that it floats in the level in mid-air. Now, if we play the game, the physics body will not move unless it is shot at by the player with the **FirstPersonProjectile** blueprint. Once shot, the physics body will start spinning and moving in the appropriate direction based on where it's shot and the impulse that is applied to it by the projectile blueprint.

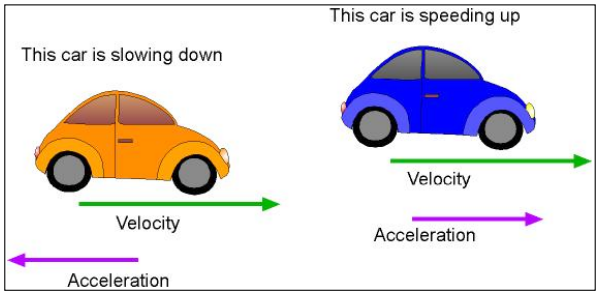
Back in the **Details** panel of **Physics Body**, we can now explore the **Angular Damping** property, which is set to `0` by default, meaning that the object has no additional drag force applied to it to reduce angular velocity. By increasing this value, we can see that when we shoot the object in the game, the angular velocity slows down in a stronger exponential value compared to how it behaved when the **Angular Damping** value was set to `0`. We can also set the limit on how fast the physics object can rotate when a force is applied to it by altering the value of the **Max Angular Velocity** parameter; the higher the value, the faster the object is able to rotate if enough force is applied to it.

For this example, let's set the **Angular Damping** value to `.01` and, the **Max Angular Velocity** parameter to `800`. We will override the **Mass in Kg** parameter and set a custom mass of `10.0`. We should now have the following properties in place for our physics body object:



Now, if we play the game and shoot at our **Physics Body** object, we will see that the object can rotate at a very quick rate when shot at by the player. To find out what else can affect the **Angular Velocity** and **Angular Damping** of an object, we can increase or decrease the values for the **Mass in Kg**, **Angular Damping**, and **Max Angular Velocity** parameters. With the knowledge of **Angular Damping** and **Angular Physics** under our belt, let's now discuss **Linear Damping** and **Linear Physics** in Unreal Engine 4.

In physics, **Linear Velocity** is defined as the rate of the change of linear displacement of time, also known as a vector quantity, which specifies the linear speed of an object and the direction in which the object is moving in.



The formula for basic **Linear Velocity** is $V = S/t$, or velocity equals the change in displacement of the object divided by the time the object takes to move this change in displacement. The concept of **Linear Damping** is the reduction of movement over time in order to have an object come to a complete stop once a force is applied to it that would cause displacement, also known as **Friction**. When it comes to **Linear Momentum**, the formula is $p = mv$, where p is the value of **Momentum**, m is the value of the object's mass, and v is the velocity of the object.

In Unreal Engine 4, we can damper the linear velocity of our physics body in two ways: by creating a **Physical Material** to apply to our physics body or by changing the **Linear Damping** property in the **Physics** section of the **Details** panel of our physics body. For the purposes of this section, we will only discuss how to change the **Linear Damping** property and we will go into more detail on how to create **Physical Material** later in this chapter. Working with **FirstPersonExampleMap** in **Unreal_PhyProject**, let's go ahead and create a working example of how to properly utilize the **Linear Damping** property for our physics bodies in order to recreate **Friction**. For this example, we can use the same physics body that we used to demonstrate angular velocity and angular damping, but we do want to make sure that we check the **Enable Gravity** parameter and set the rest of the parameters to their default values so that we can start this example from scratch.

By increasing the value of the **Linear Damping** property to a value such as `100` and playing the game, we will see that when we shoot the physics body, it does not translate as much as it would if the **Linear Damping** property was set to its default value of `.01`. With the value of `100`, the **Linear Damping** property causes the physics body to almost stay where it is, but the rotation of the object changes as normal because we did not change the properties involving angular damping. The use of **Linear Damping** is very straightforward, and there is not much customization passed (as discussed in the context of physics bodies in their **Physics** properties), but we can add additional properties using **Physics Materials**.

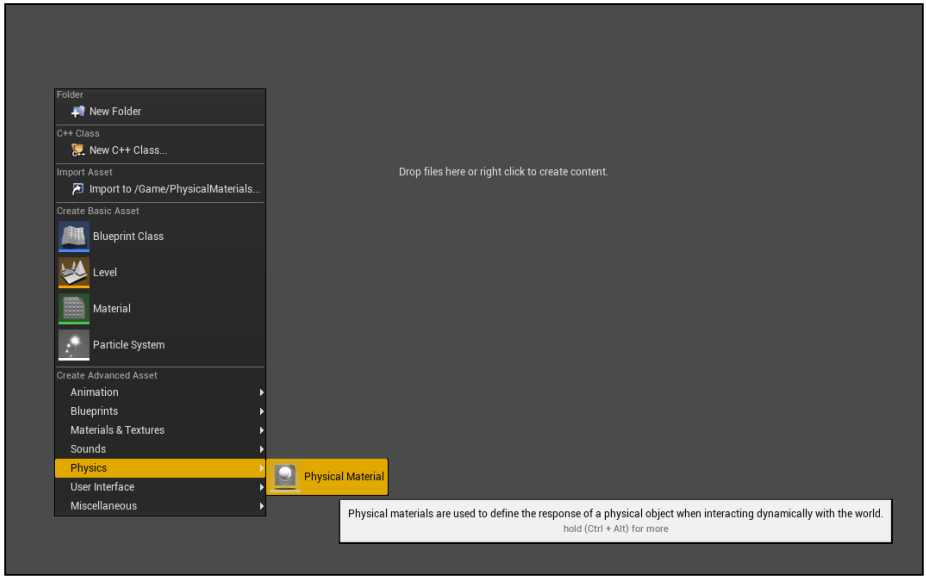
Angular and Linear Damping – a section review

In this section, we discussed the concepts of **Angular** and **Linear Damping** and **Angular** and **Linear Velocities** in the context of real-world physics in detail. Once we were able to grasp these concepts, we then applied what you learned about using physics bodies in Unreal Engine 4 and the **Physics** properties of these objects in order to explore how they affect the physics bodies in the game. Now that we have examined angular and linear damping, we can move on to briefly discuss what **Physics Materials** are and how to apply them to our physics bodies.

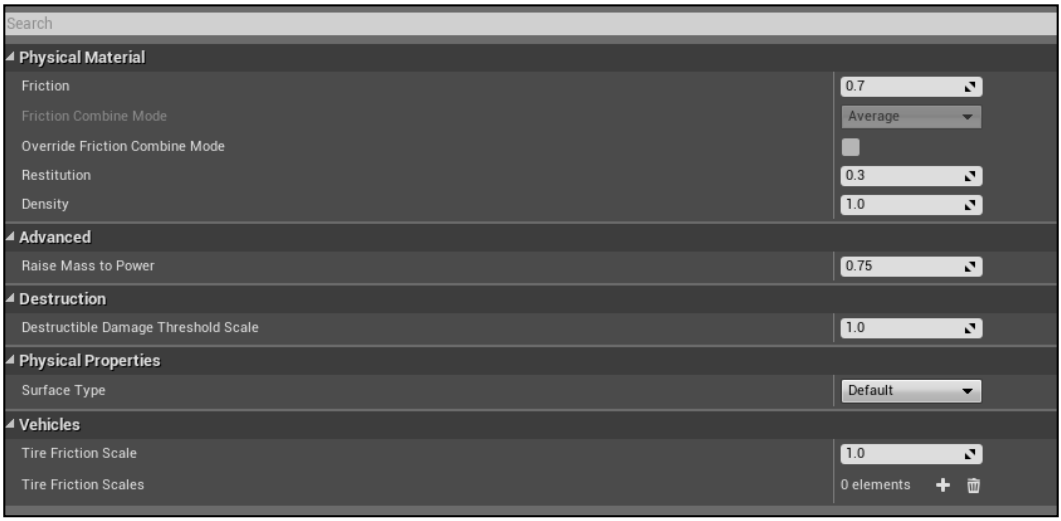
Physical Materials – an overview

Physical Materials are assets that are used to define the response of a physics body when you dynamically interact with the game world. When you first create **Physical Material**, you are presented with a set of default values that are identical to the default **Physical Material** that is applied to all physics objects.

To create **Physical Material**, let's navigate to **Content Browser** and select the **Content** folder so that it is highlighted. From here, we can right-click on the **Content** folder and select the **New Folder** option to create a new folder for our **Physical Material**; name this new folder `PhysicalMaterials`. Now, in the `PhysicalMaterials` folder, right-click on the empty area of **Content Browser** and navigate to the **Physics** section and select **Physical Material**. Make sure to name this new asset `PM_Test`.



Double-click on the new **Physical Material** asset to open **Generic Asset Editor** and we should see the following values that we can edit in order to make our physics objects behave in certain ways:



Let's take a few minutes to break down each of these properties:

- **Friction**: This parameter controls how easily objects can slide on this surface. The lower the friction value, the more slippery the surface. The higher the friction value, the less slippery the surface. For example, ice would have a **Friction** surface value of `.05`, whereas a **Friction** surface value of `1` would cause the object not to slip as much once moved.
- **Friction Combine Mode**: This parameter controls how friction is computed for multiple materials. This property is important when it comes to interactions between multiple physical materials and how we want these calculations to be made. Our choices are **Average**, **Minimum**, **Maximum**, and **Multiply**.
- **Override Friction Combine Mode**: This parameter allows you to set the **Friction Combine Mode** parameter instead of using **Friction Combine Mode**, found in the **Project Settings** | **Engine** | **Physics** section.
- **Restitution**: This parameter controls how bouncy the surface is. The higher the value, the more bouncy the surface will become.
- **Density**: This parameter is used in conjunction with the shape of the object to calculate its mass properties. The higher the number, the heavier the object becomes (in grams per cubic centimeter).

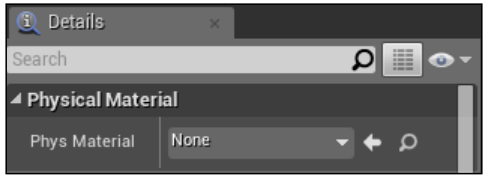
- **Raise Mass to Power**: This parameter is used to adjust the way in which the mass increases as the object gets larger. This is applied to the mass that is calculated based on a solid object. In actuality, larger objects do not tend to be solid and become more like shells (such as a vehicle). The values are clamped to `1` or less.
- **Destructible Damage Threshold Scale**: This parameter is used to scale the damage threshold for the destructible objects that this physical material is applied to.
- **Surface Type**: This parameter is used to describe what type of real-world surface we are trying to imitate for our project. We can edit these values by navigating to the **Project Settings** | **Physics** | **Physical Surface** section.
- **Tire Friction Scale**: This parameter is used as the overall tire friction scalar for every type of tire and is multiplied by the parent values of the tire.
- **Tire Friction Scales**: This parameter is almost identical to the **Tire Friction Scale** parameter, but it looks for a **Tire Type** data asset to associate it to. **Tire Types** can be created through the use of **Data Assets** by right-clicking on the **Content Browser** | **Miscellaneous** | **Data Asset** | **Tire Type** section.

Now that we have briefly discussed how to create **Physical Materials** and what their properties are, let's take a look at how to apply **Physical Materials** to our physics bodies. In **FirstPersonExampleMap**, we can select any of the physics body cubes throughout the level and in the **Details** panel under **Collision**, we will find the **Phys Material Override** parameter. It is here that we can apply our **Physical Material** to the cube and view how it reacts to our game world.

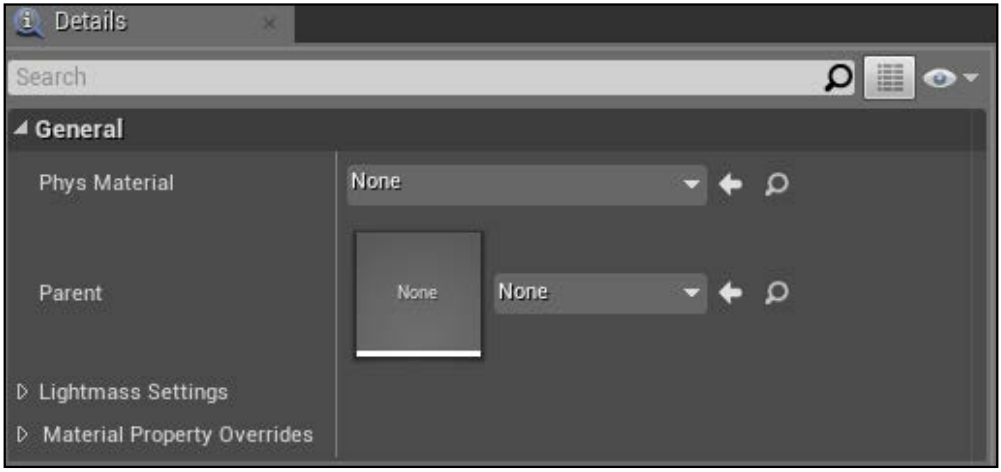
For the sake of an example, let's return to the **Physical Material**, `PM_Test`, that we created earlier, change the **Friction** property from `0.7` to `0.2`, and save it. With this change in place, let's select a physics body cube in **FirstPersonExampleMap** and apply the **Physical Material**, `PM_Test`, to the **Phys Material Override** parameter of the object. Now, if we play the game, we will see that the cube we applied the **Physical Material**, `PM_Test`, to will start to slide more once shot by the player than it did when it had a **Friction** value of `0.7`. We can also apply this **Physical Material** to the floor mesh in **FirstPersonExampleMap** to see how it affects the other physics bodies in our game world. From here, feel free to play around with the **Physical Material** parameters to see how we can affect the physics bodies in our game world.

Lastly, let's briefly discuss how to apply **Physical Materials** to normal **Materials**, **Material Instances**, and **Skeletal Meshes**.

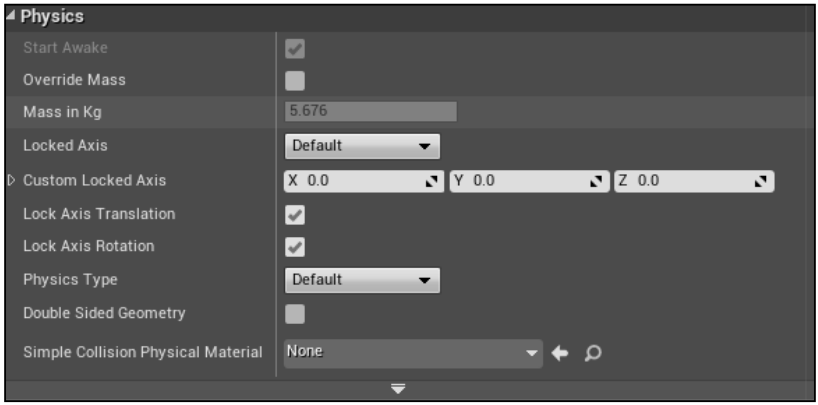
To apply **Physical Material** to a normal material, we first need to either create or open an already created material in **Content Browser**. To create a material, just right-click on an empty area of **Content Browser** and select **Material** from the drop-down menu. Double-click on **Material** to open **Material Editor**, and we will see the parameter for **Phys Material** under the **Physical Material** section of **Details** panel in the bottom-left of **Material Editor**:



To apply **Physical Material** to **Material Instance**, we first need to create **Material Instance** by navigating to **Content Browser** and right-clicking on an empty area to bring up the context drop-down menu. Under the **Materials & Textures** section, we will find an option for **Material Instance**. Double-click on this option to open **Material Instance Editor**. Under the **Details** panel in the top-left corner of this editor, we will find an option to apply **Phys Material** under the **General** section:



Lastly, to apply **Physical Material** to **Skeletal Mesh**, we need to either create or open an already created **Physics Asset** that contains **Skeletal Mesh**. In the **First Person Shooter Project** template, we can find **TutorialTPP_PhysicsAsset** under the **Engine Content** folder. If the **Engine Content** folder is not visible by default in **Content Browser**, we need to simply navigate to **View Options** in the bottom-right corner of **Content Browser** and check the **Show Engine Content** parameter. Under the **Engine Content** folder, we can navigate to the **Tutorial** folder and then to the **TutorialAssets** folder to find the **TutorialTPP_PhysicsAsset** asset. Double-click on this asset to open **Physical Asset Tool**. Now, we can click on any of the body parts found on **Skeletal Mesh** to highlight it. Once this is highlighted, we can view the option for **Simple Collision Physical Material** in the **Details** panel under the **Physics** section. Here, we can apply any of our **Physical Materials** to this body part.



Physical Materials – a section review

In this section, we discussed in detail what **Physical Materials** are and what their parameters mean when applying them to a physics body. Additionally, we explored how to apply **Physical Materials** to physics bodies, **Materials**, **Material Instances**, and **Skeletal Meshes** in **Physical Asset Tool**. Now that we have a better understanding of **Physical Materials**, we can now conclude this chapter by working with **Constraints** in blueprints to better understand **Physics Damping**.

Physics Damping

In order to gain a stronger grasp of **Physics Damping** in Unreal Engine 4, we will create a simple working example of **Constraints** with blueprints. To start with, let's continue our work in the **Unreal_PhyProject** project, navigate to **Content Browser** and then to the **Blueprints** folder, and right-click on it to create a new **Actor** blueprint. Let's name this blueprint `BP_Constraint` and double-click on it to open **Blueprint Editor**.

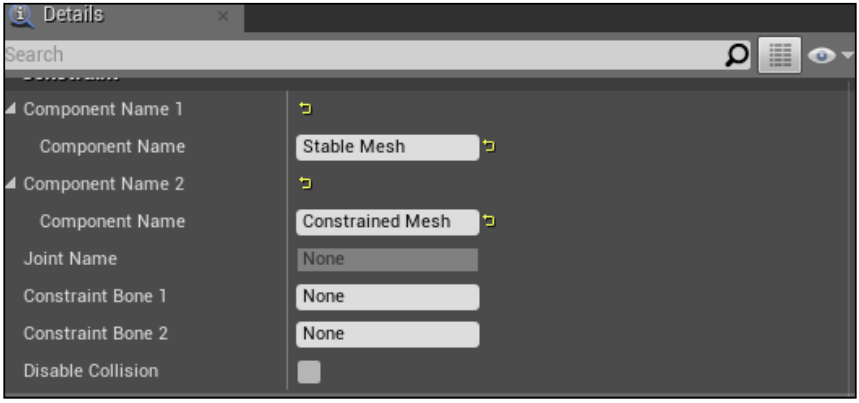
To begin this blueprint, let's navigate to the **Viewport** tab and then add **Scene Component** to the **Components** tab using the **Add Component** context sensitive drop-down menu. Name this component `ROOT`. Next, we will add two cube meshes from the **Basic Shapes** section to the **Add Components** menu. Name one `Stable Mesh` and the other `Constrained Mesh`. Lastly, let's add a **Physics Constraint** component from the **Physics** section and name it `Physics Constraint`. Now, we need to position these components in a manner that will better showcase how to use physics damping in Unreal Engine 4.

Set the position of our assets in the **Viewport** tab of our blueprint as follows:

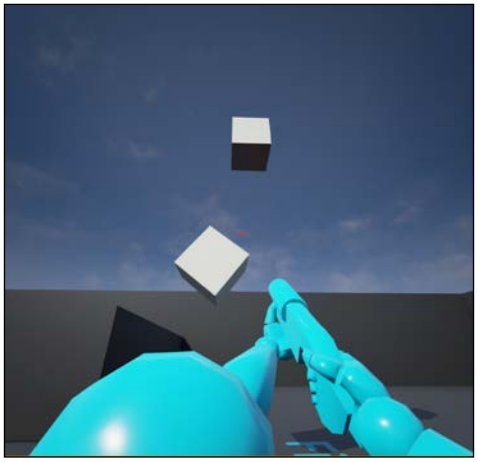
- **Stable Mesh**: `X - 0.0, Y - 0.0,` and `Z - 350.0`
- **Constrained Mesh**: `X - 120.0, Y - 0.0, Z - 0.0`
- **Physics Constraint**: `X - 0.0, Y - 0.0, Z - 340.0`

Now, we will need to set the default parameters to our **Physics Constraint** asset so that it recognizes the mesh that is the anchor. The other mesh is the free-hanging pendulum attached to it. To accomplish this, let's select the **Physics Constraint** component. Then, in its **Details** panel, we will find the parameters for **Component Name 1** and **Component Name 2** under the **Constraint** section. For these parameters, we will need to apply the names of the two meshes that we will use for our constraint: `Stable Mesh` and `Constrained Mesh`. Set these parameters as follows:

- **Component Name 1**: `Stable Mesh`
- **Component Name 2**: `Constrained Mesh`



With these parameters in place, we can now place our **BP_Constraint** in **FirstPersonExampleMap** to see the constraint in action. First, make sure to compile and save the blueprint before placing this blueprint in our level. Once this is placed in our level, make sure to lift the blueprint well off the ground so that there won't be any collision or clipping between the constrained assets and our game world, and when we can play the game to see how the constraint works, we will see the following result:



We will see that the swinging cube will continue to swing back and forth for eternity without any sign of damping or friction to slow it down. Let's change this. Then, navigate back to the **BP_Constraint** blueprint and select the **Physics Constraint** component. In the **Details** panel, we will see a handful of parameters that we can change to the constraint in order to affect its behavior. Let's briefly define each parameter found in the **Details** panel of the **Physics Constraint component**:

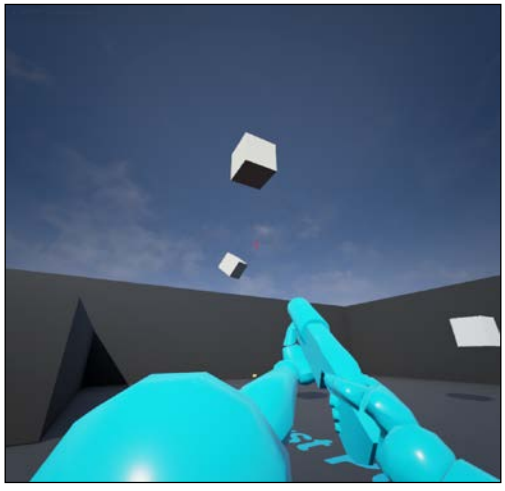
- **Component Name 1**: This parameter requires the name of the first component property to constrain. If this is left empty, the name parameter will search in its **Owner** for a parameter name. If **Owner** returns null, this parameter will use the **Root Component** of **Actor 1**.
- **Component Name 2**: This parameter requires the name of the second component property to constrain. If this is left empty, the name parameter will search in its **Owner** for a parameter name. If **Owner** returns null, this parameter will use the **Root Component** of **Actor 2**.
- **Joint Name**: This parameter is used when you work with **Skeletal Meshes** and requires the name of **Bone** that **Joint** is attached to.

- **Constraint Bone 1**: This parameter requires the name of the first bone (body) that this constraint is connecting to and would be the child bone in **Physics Asset**.
- **Constraint Bone 2**: This parameter requires the name of the second bone (body) that this constraint is connecting to and would be the parent bone in **Physics Asset**.
- **Disable Collision**: This parameter disables the collision between the bodies joined by this constraint.
- **Enable Projection**: This parameter ensures that all the bodies are projected so that both these bodies still appear attached to each other if a high enough linear or angular velocity is applied to each element. For example, a tether ball spinning too fast would cause the elements to look detached, but this parameter stops this from happening. If the distance error between the two bodies exceeds 0.1 units, or if the rotation error exceeds 10 degrees, the projection will correct this.
- **Projection Linear Tolerance**: This parameter represents **Linear Tolerance** in world units, and if the distance error exceeds this tolerance limit, the body will be projected.
- **Projection Angular Tolerance**: This parameter represents **Angular Tolerance** in world units, and if the angular distance error exceeds this tolerance limit, the body will be projected.
- **Linear X Motion**: This parameter indicates whether or not the linear motion along the **X** axis is allowed, blocked, or limited. If this is limited, the **Linear Limit** property will be used to determine whether a motion is allowed.
- **Linear Y Motion**: This parameter indicates whether or not the linear motion along the **Y** axis is allowed, blocked, or limited. If this is limited, the **Linear Limit** property will be used to determine whether a motion is allowed.
- **Linear Z Motion**: This parameter indicates whether or not the linear motion along the **Z** axis is allowed, blocked, or limited. If this is limited, the **Linear Limit** property will be used to determine whether a motion is allowed.
- **Linear Breakable**: This parameter defines whether or not the joint in the constraint is breakable based on the **Linear Break Threshold** property.
- **Linear Break Threshold**: This parameter defines the force threshold required to break this joint.
- **Angular Swing 1 Motion**: This parameter indicates whether or not the rotation around the **Z** axis is allowed, blocked, or limited. If this is limited, the **Angular Limit** property will be used to determine the range of motion.

- **Angular Twist Motion**: This parameter indicates whether or not the rotation around the **X** axis is allowed, blocked, or limited. If this is limited, the **Angular Limit** property will be used to determine the range of motion.
- **Angular Swing 2 Motion**: This parameter indicates whether or not the rotation around the **Y** axis is allowed, blocked, or limited. If this is limited, the **Angular Limit** property will be used to determine the range of motion.
- **Angular Breakable**: This parameter determines whether or not it is possible to break the joint with angular force.
- **Angular Break Threshold**: This parameter dictates the angular force necessary to break the joint.
- **Linear Position Drive**: This parameter enables/disables the linear position drive. Here, we can set the **Linear X**, **Y**, and **Z** axes and **Linear Position Target**.
- **Linear Velocity Drive**: This parameter enables/disables the linear velocity drive, where we can set the **X**, **Y**, and **Z** linear velocity targets for the constraint.
- **Linear Position Strength**: This parameter dictates the spring force applied to the linear drive.
- **Linear Velocity Strength**: This parameter determines the damping force applied to the linear drive.
- **Linear Drive Force Limit**: This parameter limits the force that can be applied to the linear drive.
- **Angular Orientation Drive**: This parameter enables the angular drive towards a target orientation along the **X**, **Y**, or **Z** axes.
- **Angular Velocity Drive**: This parameter enables the angular drive towards a target velocity along the **X**, **Y**, and **Z** axes.
- **Angular Drive Force Limit**: This parameter limits the force that the angular drive can apply.
- **Angular Position Strength**: This parameter applies a spring force value to the angular drive.
- **Angular Velocity Strength**: This parameter applies a damping value to the angular drive.
- **Angular Drive Mode**: This parameter determines the way the angular paths are estimated; we can either select **SLERP (Spherical Linear Interpolation)** or decompose it into **Twist and Swing**.

Now that we have briefly discussed the parameters available in the **Physics Constraint** component, let's apply angular damping to our constraint so that it can be slowed down and brought to a complete halt. In order to make this happen, we will need to apply **Angular Velocity Drive** and set **Angular Velocity Strength** to a value of `15.0`. Next, we will need to set **Angular Drive Mode** to **Twist and Swing**; the remaining parameters can be set to their default values. Once applied, let's compile, save, and jump back to **FirstPersonExampleMap**. We will see that over a short period of time, **Constrained Mesh** will slowly come to a halt. This is due to the high value of the **Angular Velocity Strength** parameter; the higher the value, the quicker **Constrained Mesh** will come to a halt. The lower the number, the longer it would take for **Constrained Mesh** to come to a halt all the way to the point where the mesh will not stop swinging.

What we can do now to conclude this chapter is set an angular force break so that if the constrained mesh moves with enough force, it will break from the constraint entirely and become its own unique physics body. To accomplish this, let's return to the **BP_Constraint** blueprint and select the **Physics Constraint** component from the **Components** tab. Now, in the **Details** panel under the **Angular Limits** section, we can set **Angular Swing 1 Motion**, **Angular Twist Motion**, and **Angular Swing 2 Motion** to **Limited** and leave their **Angle** values as their defaults. Lastly, make sure that the **Angular Breakable** parameter is checked and **Angular Break Threshold** is set to a value of `50`. If we compile and save the blueprint and jump back to the **FirstPersonExampleMap** level, we will see that when we shoot the constrained mesh enough times to cause its angular velocity to increase, once it reaches the threshold that we set, the mesh will separate entirely from the constraint. From here, we can interact with the separated mesh as its own physics body. This scenario is useful if you want to simulate physics for a tire swing, or for any type of a pendulum object that we wish to provide this behavior to. Feel free to experiment with the **Physics Constraint** values to see how else we can affect the behaviors of the constraint.



Physics Damping – a section review

In this section, we looked at how to explore **Physics Damping** using blueprints and the **Physics Constraint** component. By setting values in **Physics Constraint**, we are able to simulate a pendulum in our game world that would swing forever until we apply angular damping with the **Angular Velocity Strength** parameter. Lastly, we applied an angular threshold that would cause our constrained mesh to break from the joint and become its own separate physics body.

Summary

In this chapter, we discussed what **Physics Bodies** are and how they function in Unreal Engine 4. Moreover, we looked at the properties that are involved in **Physics Bodies** and how these properties can affect the behavior of these bodies in the game.

Next, we explored what **Angular** and **Linear Damping** are and how they can affect our **Physics Bodies**. We also discussed real-world physics when it comes to linear and angular momentum, apart from linear and angular velocities.

Additionally, we briefly discussed **Physical Materials**, how to create them, and what their properties entail when it comes to affecting its behavior in the game. We then reviewed how to apply **Physical Materials** to static meshes, materials, material instances, and skeletal meshes.

Lastly, we applied **Physics Damping** to **Physics Constraint** by creating a working blueprint example, where we constrained two cube meshes together and created a pendulum. Moreover, we applied angular damping and angular threshold breaks to slowly bring the constrained cube mesh to a halt. We also implemented the ability for the constrained mesh to break from the joint to become its own physics body.

Now that we have a stronger understanding of how **Physics Bodies** work in the context of angular and linear velocities, momentum, and the application of damping, we can move on and explore in detail how **Physical Materials** work and how they are implemented.

6
Materials

Every video game provides a kind of colorful presentation for the player (even in black and white). As most players experience memorable parts of the game, this is what a player hears and detects visually in the game. This detection can be about an image, some notes, or something known as texture.

Texture is a digital shell that covers objects in the game world. They can be unique or repetitive over a big area during the play in the different levels of the game. They can be interactive with the player or world, artistic, fantasy, or real and physical.

For example, sea mostly has a real texture of water. Some old engines just handle a simple image of water, whereas some modern engines (such as UE4) can provide the wave motion over the texture. Water is well known by everyone on our planet, and physical rules related to water have already been discovered by everyone during their lifetime.

On the other hand, *Alien* skin is something mostly related to art and fantasy of the game story. There is no experimental or proven physical aspect for such a thing. It can wave or vibrate with many different forms.

The art of creating materials in UE4 covers the physical and artistic features of design in detailed textures for the game.

What is physical material?

Unreal Engine 4 defines a new method for creating materials. This method is based on the physical aspects of the real world. Things such as light reflections or heaviness will directly affect your model in the game world as visual elements and interactive behavior. You can make an ice cube that moves on the surface like a real one, or you can make waves of the ocean that have their own movements and reflect sunlight.

Basically, the quality of material has a direct connection to the graphics card and technologies involved in the CPU and the memory speed of the machine of a user. This leads developers and art directors to answer some important questions: how does our game render materials? Is it necessary that a user should have a super good machine to get what designers plan to show, or can we solve the quality aspect in simple ways?

A material can be created in three different ways. The first method uses an image that is copied to memory. This image gets loaded several times from the memory and depends on the size and alpha channel. It also gets some processing time from the CPU. The other method is to use the shaders technology. Here, the graphics card and the CPU are involved in creating a fully calculated shell on the surface of the object that represents the material. The final one is a mix of both. Unreal Engine 4 has all the tools to develop materials of high quality. At the same time, artists/developers can include some Shader-based effect on the surface of object. These features can be customized. This means that you can allow the players to set the graphics options of the game depending on the machine they use.

Let's go through an example of visual and interactive elements and how to create a physics-based material in the next section.

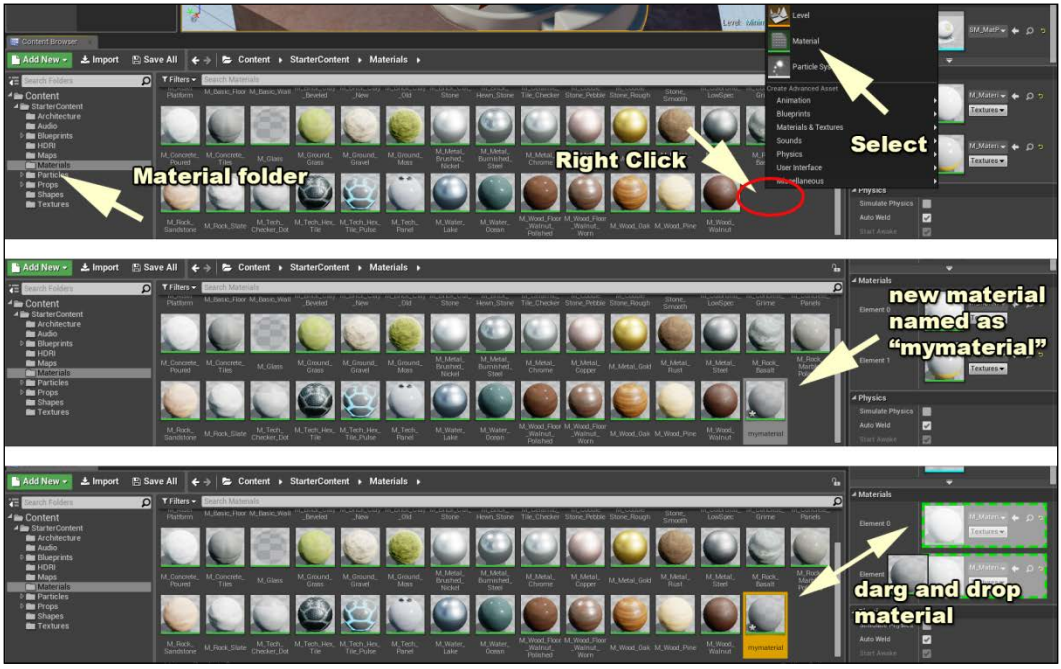
Creating the first material

Before we start working in Unreal Editor, we will need to have a project to work with. Perform the following steps:

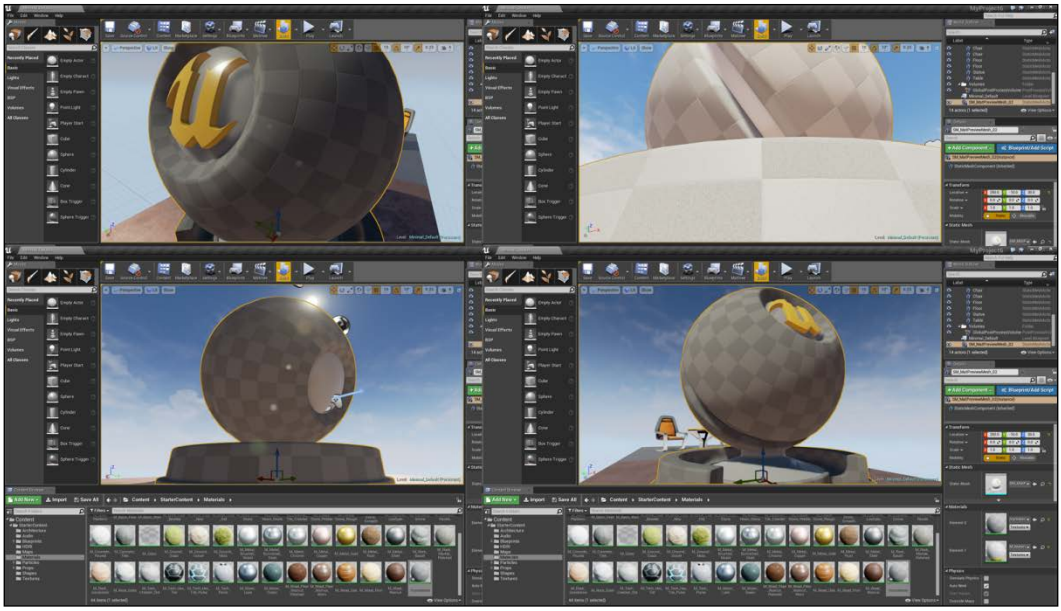
1. First, open Unreal Editor by clicking on the **Launch** button from the Unreal Engine launcher.
2. Then, start a new project from **Project** browser by selecting the **New Project** tab. Now, select **Blank** and make sure that **With Starter Content** is selected. Name the project `material_test`.

3. From **Content Browser**, click on the **Props** folder. You will see a series of models that you can drag and drop onto the stage. Select **SM_MatPreviewMesh_02** and drag an instance of it to the stage. Then, press the *F* key to focus the camera on your shape. We will use this shape to describe the physical aspects of materials in the game. You can choose other shapes and follow the next steps.

4. Now, click on the **Materials** folder and right click on an empty space in the preview section. From the menu, select **Material** and then enter `mymaterial` as the name for the new one. Now, drag and drop this material onto the **Materials** section in **Element0**, in **Details** menu. Now your new material is defined in the model:



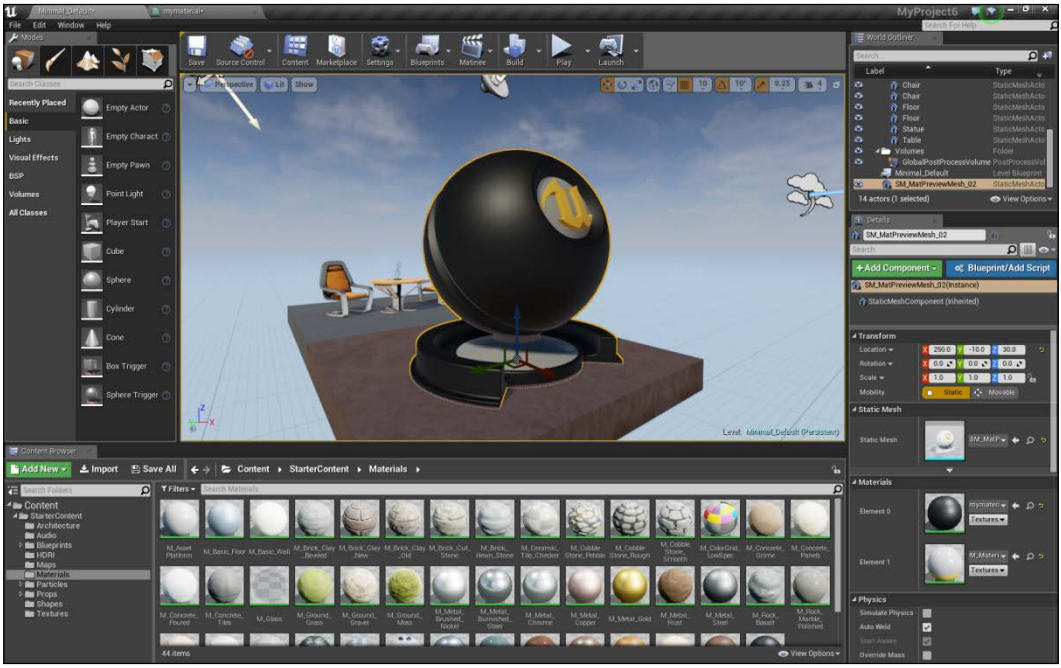
5. Rotate the shape and zoom in and out of the surface. It's important to learn how you can easily navigate your camera around the model when you create materials. Unreal Engine 4 provides the basic elements of physical light and ray processing in real time for the materials. It also calculates how rough the surface is and then dynamically renders this on the shape. All these are at the same time connected to world details, such as sunlight or any other source of light, gravity, and materials around the shape. Therefore, it is good practice to check your object from different angles and zoom value. Try to reach out to the angles, as shown in the following screenshot:



Also, the model that we are using has two elements: **Element0** and **Element1**. **Element0** is the surface or shell of the model, whereas **Element1** is similar to the core of the shape. Both can accept different materials from the **Material** folder.

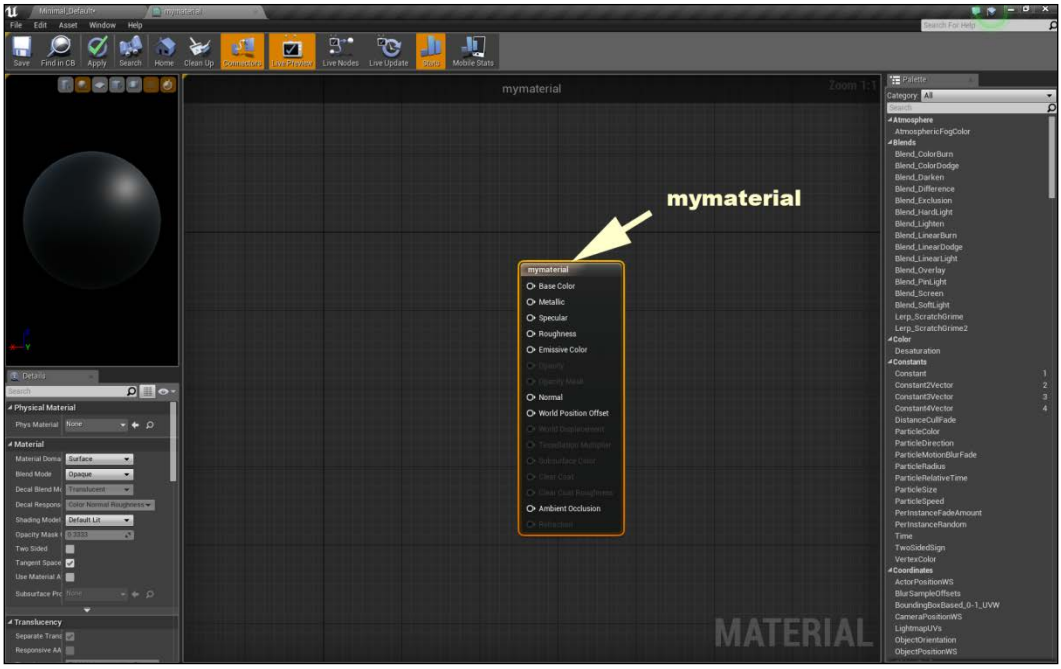
6. Double-click on **Element0** in the **Materials** section of the **Details** menu. This opens the blueprint and the material editor for your material. In the middle of the screen, there is a blueprint of the current material. This is completely new and is named `mymaterial`.

Before we go too far, click on **Apply** from the top menu and check your model. It's similar to a black surface that covers the shell; your material is simply applied to the model, as shown in the following screenshot:



From now on, you will learn how to change this layer on the shape using the blueprint in UE4.

7. Double-click on **Element0** to go back to the material blueprint. As you can see in the following screenshot, there is a box with your material name (`mymaterial`). This box has some features to connect the blueprint units:



Imagine that this box is kind of an output. You will send data to base color, metallic, and the rest of the inputs of this box. It creates a surface for your model and then outputs this data to your model's surface. As you can see, we don't have any input for this box at this stage. As a result, our material is simple and without any color or texture. You will have a sample of your material in the top-left corner of your screen that you can rotate or zoom with the help of your mouse.

The physics of materials

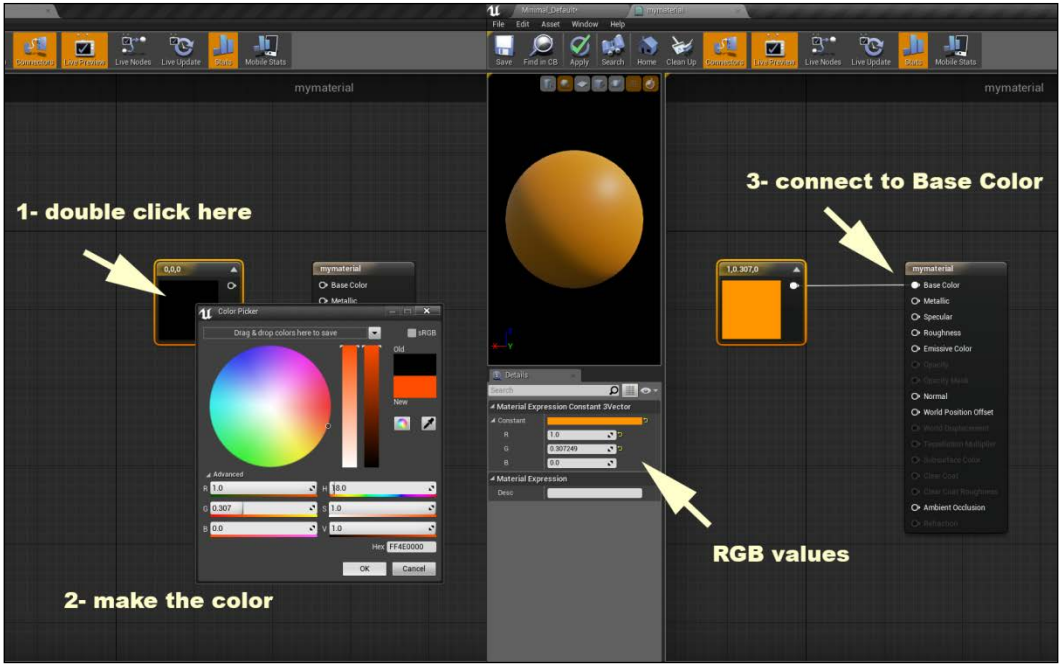
Each material in Unreal Engine 4 follows some physical properties to be defined by a designer. This is somehow different from the previous versions of the engine. Unreal Engine 4 gives developers and artists more options to create complex materials with higher performance compare to old versions. Also, you can invent or create dynamic materials with this structure.

As you can see in the preceding screenshot, there are a number of commands on the left-hand side of your material blueprint editor in the **Palette** section. Some have shortcuts to use. For example, you can drag and drop **Constant** from here onto your screen, or simply hold the *1* key on your keyboard and left-click your mouse on the screen. Both these methods give you a constant number to connect the input of the other box in the blueprint. Also, each box has some properties in the **Details** menu. These properties can be customized.

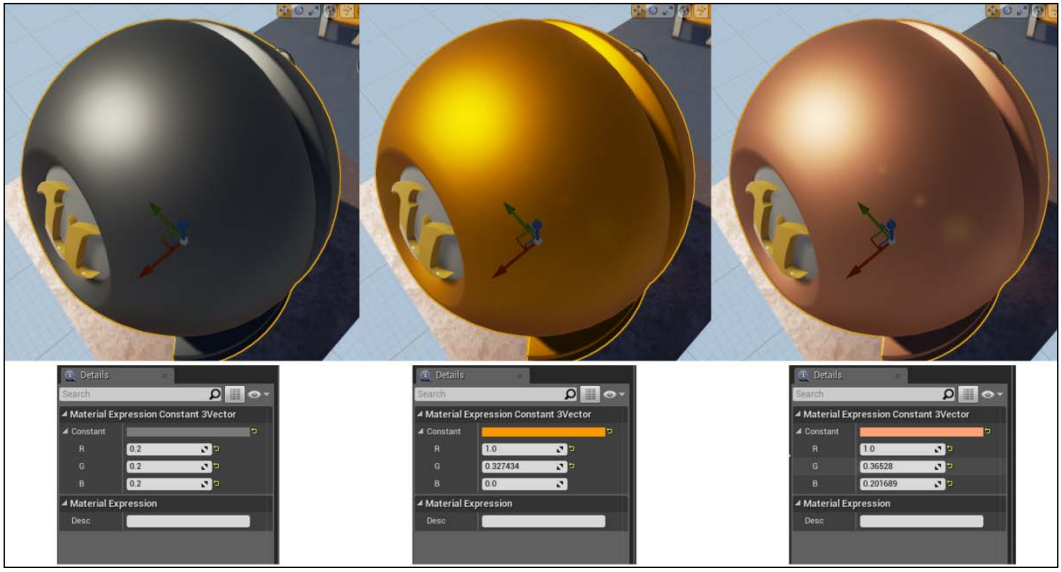
Now let's make a material, based on physical rules, inside the Unreal Engine 4:

1. At the top of the `mymaterial` box, locate **Base Color**. This property defines the color of the material and accepts numbers called vector numbers. Vectors in Unreal Engine 4 can handle 2, 3, and 4 numbers together.
    - This example describes the main concept of vectors. Now, imagine that you have three chocolate boxes named `2Vector`, `3Vector`, and `4Vector`. When you open `2Vector`, you will find two chocolates together in the box. When you open `3Vector`, you will find three chocolates together in the box, and so on. If you send one of these boxes to someone, it means that you have sent all the chocolates together in the box to the destination. Now, when it is opened at the destination, all the chocolates are there at the same time. In this scenario, the chocolate box is a vector and the chocolates are the members of the vector. The members of the vector have the same weight on interaction between the user and themselves.
    - We can use vectors when we need to use large amounts of data at the same time to apply physical properties in the engine. Color is a physical property made from three numbers for RGB values (red, green, and blue). We need to send three numbers, all together to this input, so the best choice is to send a vector number.

- On the right-hand side of the screen in **Palette**, locate **Constant2Vector**, drag and drop an instance of it onto the stage, or hold the *3* key and right-click on the stage, as shown in the following screenshot. Now, right-click on black area, set the color, and connect the box to **Base Color** on the `mymaterial` box. The values of RGB are shown in the **Details** menu as well. You can change them between `0` and `1`, so for example, pure yellow will be **R**: `1.0`, **G**: `1.0`, and **B**: `0.0`. Change this to **R**: `1.0`, **G**: `0.327`, and **B**: `0.0` to have an orange color on the model:

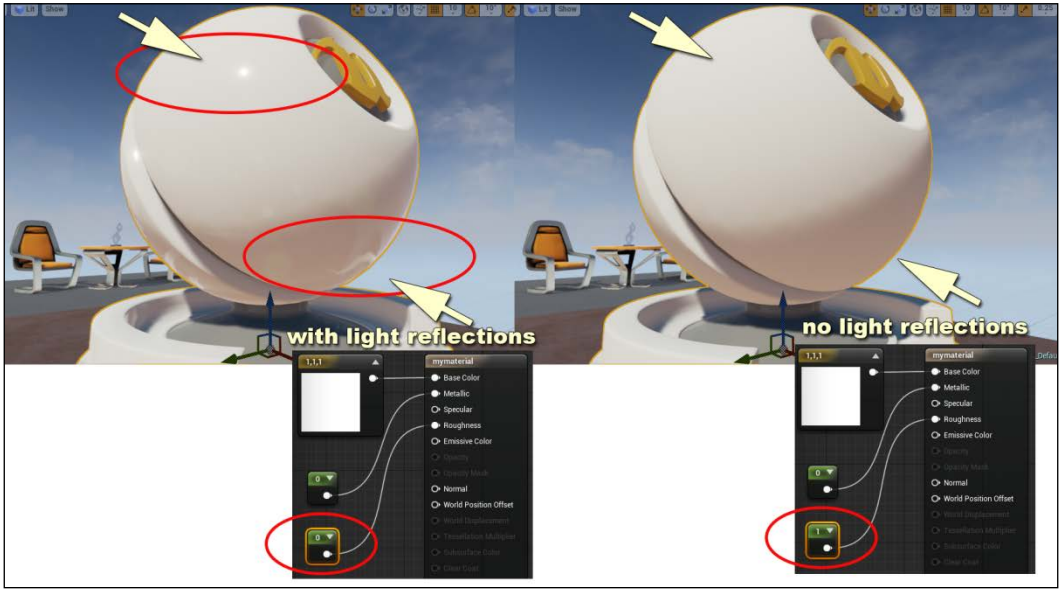


2. Metallic surface is another physical aspect of your material. It defines how metallic your object looks like in case of the shininess of lights around. Locate **Metallic** under **Base Color** on the `mymaterial` box, then drag and drop one **Constant** from **Palette** on the right-hand side. Change its value to `1.0` in the **Details** section. Click on **Apply** at the top and check your model. It looks similar to unpolished gold. Now, go back to the material editor and change the color to **R**: `0.2`, **G**: `0.2`, and **B**: `0.2`. Click on **Apply** at the top and check your model. Now, it looks similar to unpolished silver, as shown in the following screenshot:

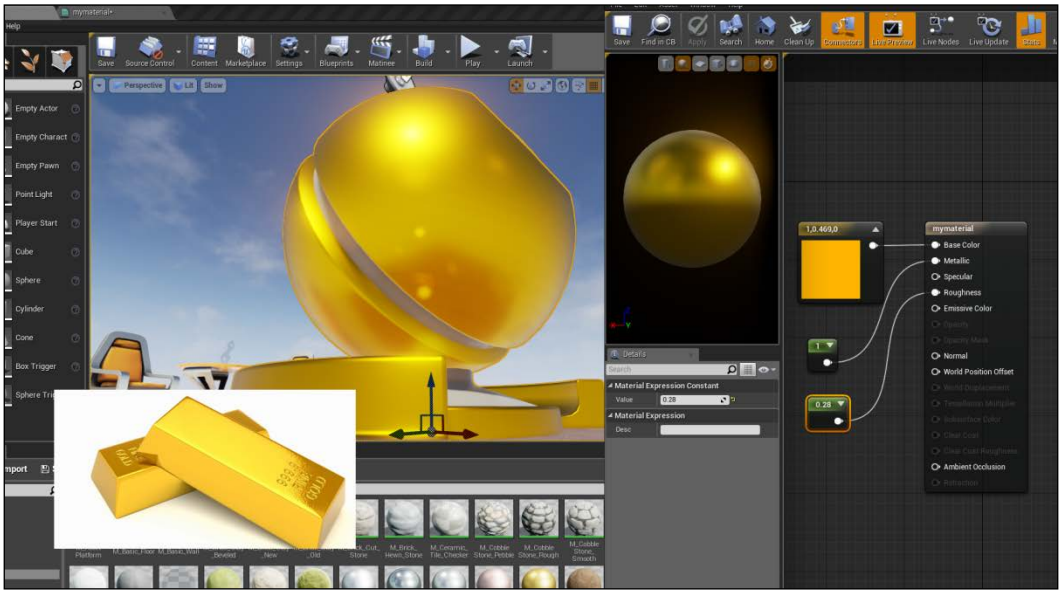


3. Each material can provide the physical behavior of soft or none of the soft surfaces. Imagine the difference between a mirror and the white surface of a wall made by chalk. The big difference is that the mirror reflects the light in a linear way, but the chalk breaks the light into many different angles. As a result, you can't see any reflection of the shape on the wall.

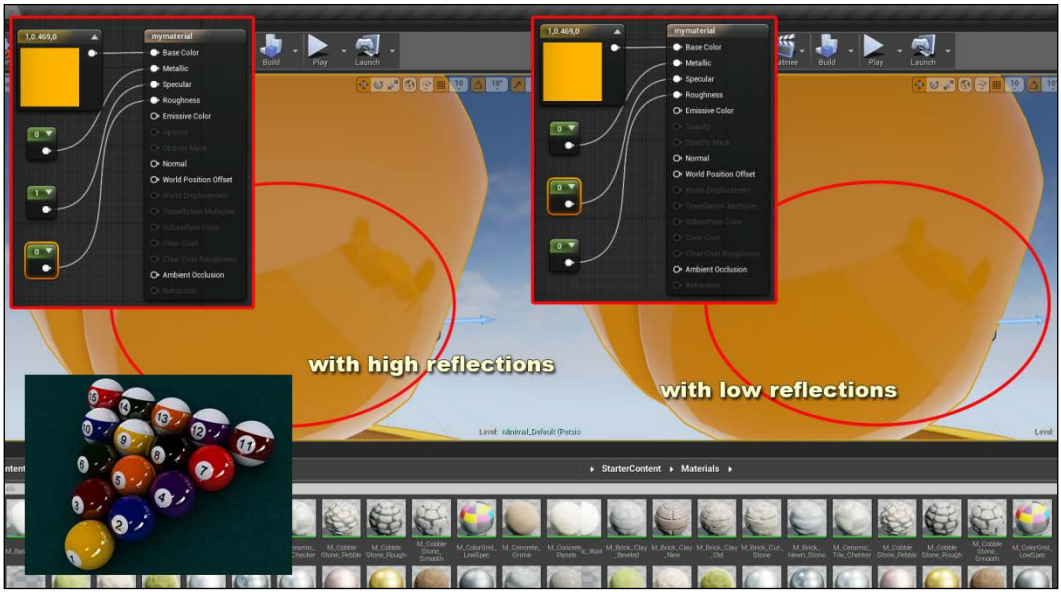
1. Let's try this by changing the color to white by setting **R**, **G**, and **B** to `1.0` for your **Base Color** property. Now, change the metallic value to `0.0` and then drag and drop one **Constant** from **Palette** on the right-hand side of your screen. Connect this to the **Roughness** property on the `mymaterial` box. Now, click on apply and check your model. It looks similar to a plastic with a mirror-like reflection of other objects on the surface. Now, go back back to the material editor and change its value to `1.0` in the **Details** section. Again, click on apply and check your model. Now, it looks similar a piece of chalk with absolutely no reflections of other objects around, as shown in the following screenshot:



2. Now, change the metallic value to `1.0` and click on **Apply**. The result is a nice reflective mirror surface around the object. Let's go back to the material editor. What you see is a fully metallic surface with maximum shininess of light and an absolute reflective surface that reflects light in the most linear and direct form. This is the physical descriptions of this material. So, if you wish to create a golden object similar to the following screenshot, you should change the values to `1.0`, `0.46`, and `1.0` for **Base Color**, `1.0` for **Metallic**, and `0.28` for **Roughness**:



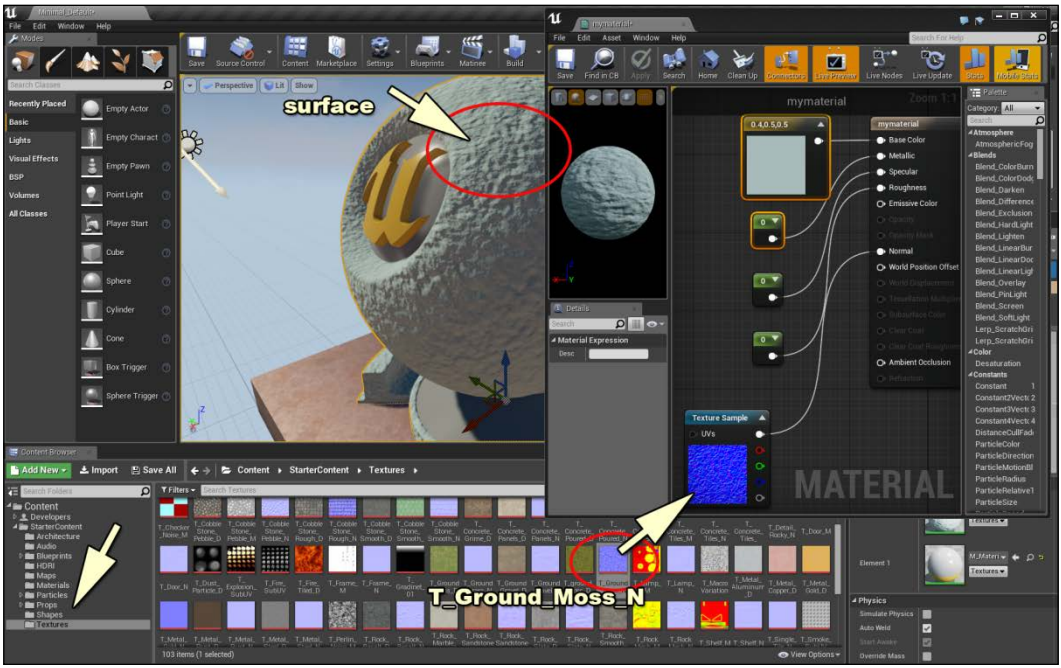
3. Sometimes, you need to create some shiny plastics (such as pool balls). For this purpose, Unreal Engine 4 defines some physical methods to simulate plastic-like materials. Change your metallic property to `0` and then change roughness to `0`. Now, click on **Apply** and check the model. As you can see, the model looks similar to a plastic shiny toy. Set the **Specular** property to `1.0` by adding one **Constant** to it. Similarly, change other properties as well. Now, click on **Apply** and check your model. The images of surrounding objects will become more visible in the plastic. Unreal Engine 4 simulates the differences between the shiny non-metal surface and the metal surface by including the **Specular** value of the surface as physical variable.



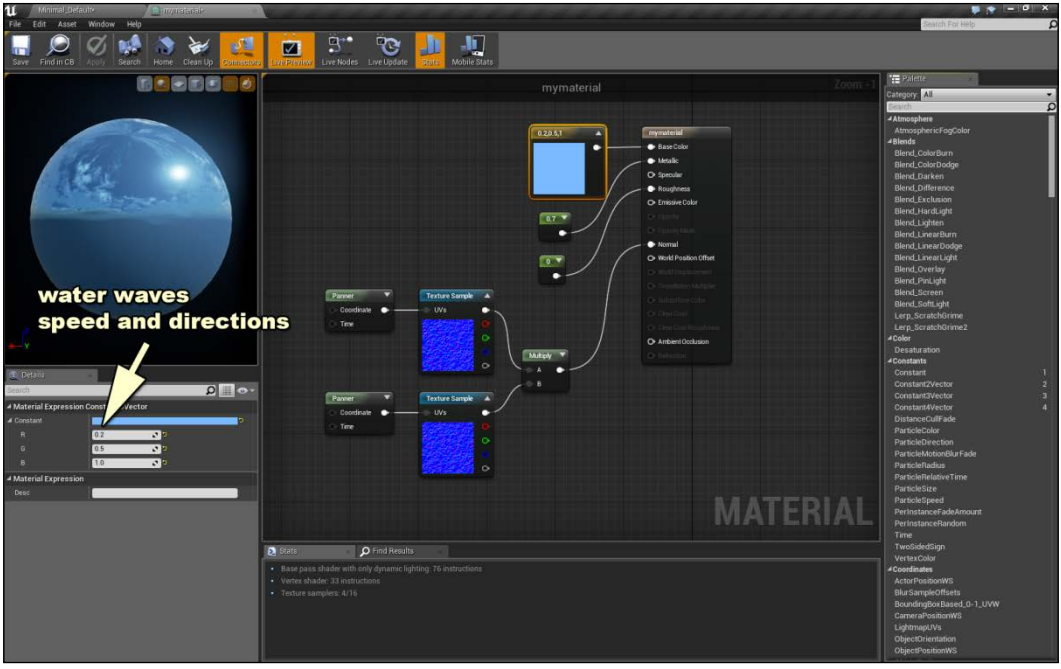
4. It is good practice to select a couple of real-world objects around your computer and create their material in the engine based on physical elements. This includes **Softness** (**Roughness** and **Specular** inside the engine) and **Shininess**, related to the **Metallic** property. For example, try to create green apple skin, shoes, cooking tools (such as a spoon or fork), and paper.

4. Some materials need more physical aspects than has been mentioned before. For example, water has some patterns similar to waves on its surface and transparency. Also, the way light reflects on the surface of water is different from metal or plastic. To create these kinds of materials, which mostly have patterns and shadows on the surface, Unreal Engine 4 uses some images known as **Normal Maps**. Click on the **Texture** folder in your editor, find `T_Ground_Moss_N` and drag it onto your material editor, and connect this to the **Normal** property of `mymaterial`. Now, change **Base Color** to `0.4`, `0.5`, and `0.5` for the values of RGB. Then, click on **Apply** and check your model.

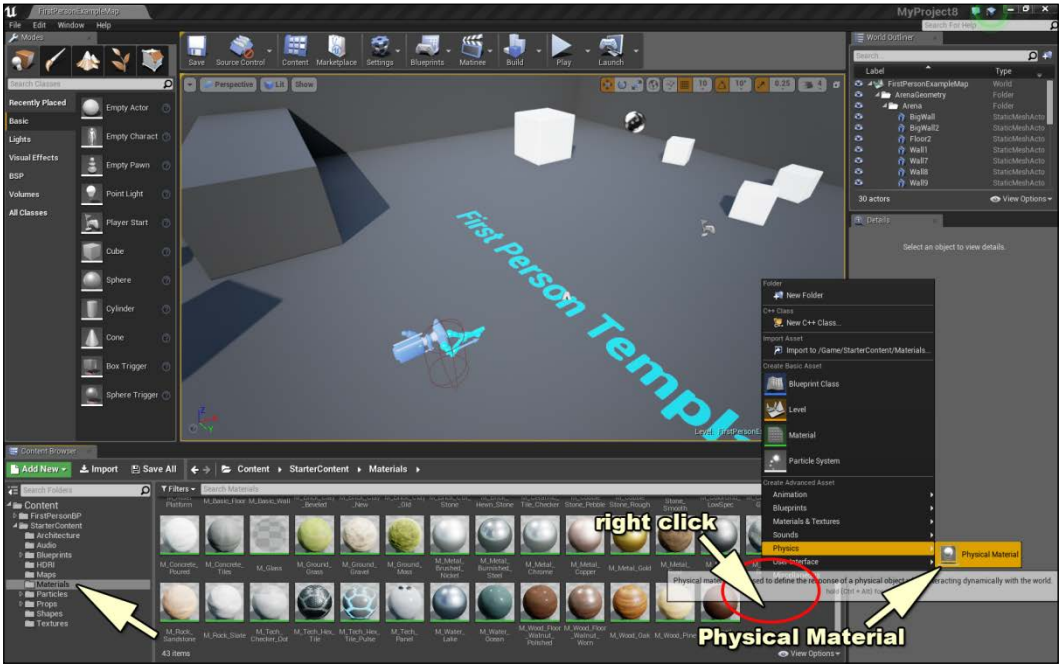
   As you can see in the following screenshot, your model will have a surface similar to a stone, which in different angles of light shows shadows.



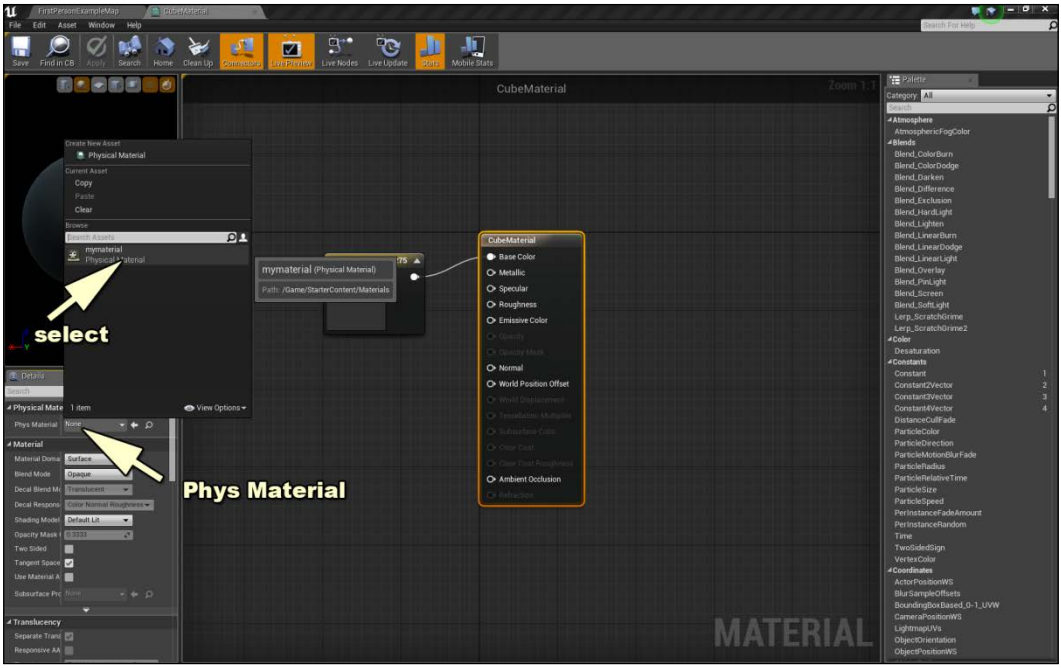
5. The texture on the surface of your model can also accept movements in the *X* and *Y* direction. Find **Panner** in **Palette** on the right-hand side of your material editor and drag it onto the stage and then connect this to the UV's input on your **Texture Sample**. Select both boxes and hit *Ctrl* + *W*. This shortcut duplicates your selection. Now, add the **Multiply** box and connect your boxes, as shown in the following screenshot. Then, change **Metallic** to `0.7` and **Roughness** to `0.0`. Now, change **Base Color** to `0.2`, `0.5`, and `1.0` for RGB to see the water. Change the values of **Panner** to `0.02`, `0.0`, `0.01`, and `-0.011` to generate water waves on your model.



6. Now, let's talk about other physical aspects. For this, close your project and create a new project from **Project** browser by selecting the **New Project** tab. Select **First Person** and make sure that **With Starter Content** is selected. Then, select the **Material** folder and right-click on an empty space, navigate to **Physics**, select **Physical Material**, and name it `mymaterial`:

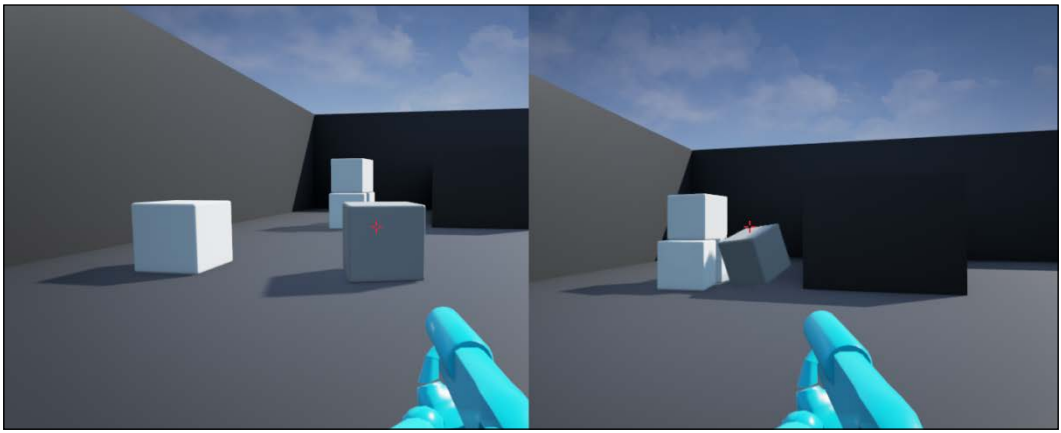


7. Now, select a cube, click on **Apply**, and double-click on **Materials** on the right-hand side to navigate to the material editor. In the **Details** section of **Phys Materials** on the left-hand side, click and select `mymaterial`; this will apply your physical material to the shape. Now, click on **Apply** at the top to go back to the material editor, as shown in the following screenshot:



8. As you can see, there isn't any change on the surface. We created a different kind of material that is responsible for the physical interaction of the shape, along with other shapes around. Let's click on **Play** and test how your cube responds to your shooting. Then, press **Stop**, double-click on `mymaterial`, and change **Density** to `20` in the new window. Now, click on **Play**. You will find that your cube appears to be heavier than the other cubes and moves very slowly in response to the shooting.
9. Click on **Stop** to go back to the editor, double-click again on `mymaterial`, and change **Density** to `1.0` (the default value). This time, change **Restitution** from `0.3` to `2.3`. Now, go back to the editor, click on **Play**, and shoot at the box. It seems that the box has some more elastic movement.
10. Now, click on **Stop** and again open the **Details** menu for `mymaterial` by double- clicking on it. Set **Restitution** to `0.3` (the default value). This time, change **Friction** to `-24` and play again. This property prepares an ice-like behavior for your shape.

You can create a simple game by changing **Density** to `10`, **Restitution** to `2.3`, and **Friction** to `-24`. Now, try to hit other cubes with the one that you defined your physical materials with.



Start a new project from **Project** browser by selecting the **New Project** tab. Select **First Person** and make sure that **With Starter Content** is selected and name the project `material_test`.

Summary

Making materials is an art in game design, and the new generation of Unreal Engine, which is 4, provides sufficient levels of detail design for the artists and developers in this area. Basically, the first step of making each material in Unreal Engine 4 is to address the physical features of this object. Features such as Metallic or *Color* and *Density* are some examples of how you can make a material inside your game world, which we went through in this chapter in detail.

7
Creating a Vehicle Blueprint

In this chapter, we will create a working **Vehicle Blueprint** from scratch using the default assets provided by Unreal Engine 4 with the **VehicleGame** project example as well as using assets which we will be creating by ourselves. First, we will start with an overview of what **Vehicle Blueprint** will be composed of and then move on to the specifics on how to create the different blueprints for all of their aspects. Following the overview, we will cover the following topics:

- Creating Vehicle Blueprints
- Editing Vehicle Blueprints
- Setting up user controls
- Testing the vehicle

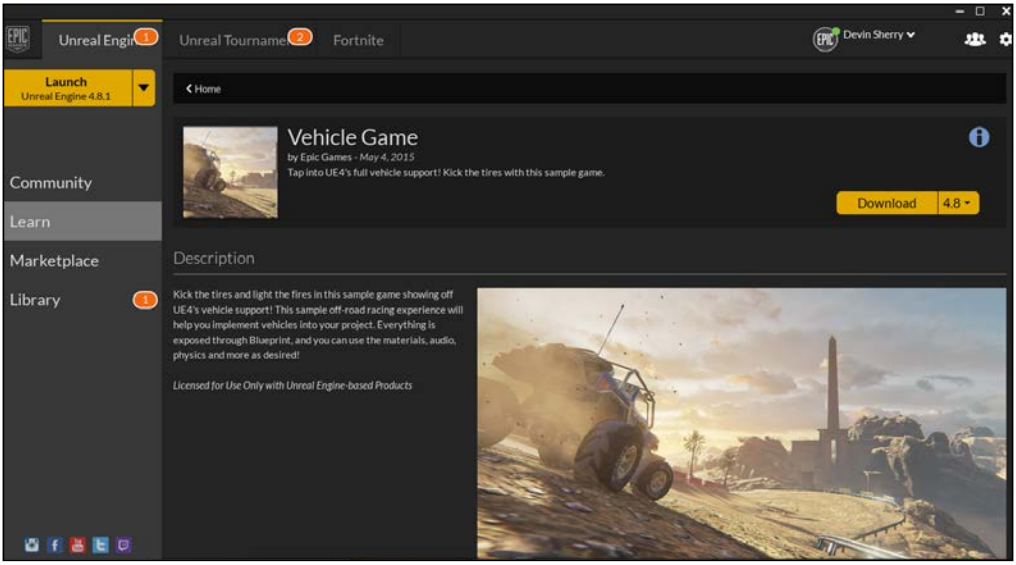
There is a lot of content to cover in this chapter, so let's get started.

Vehicle Blueprint – a content overview

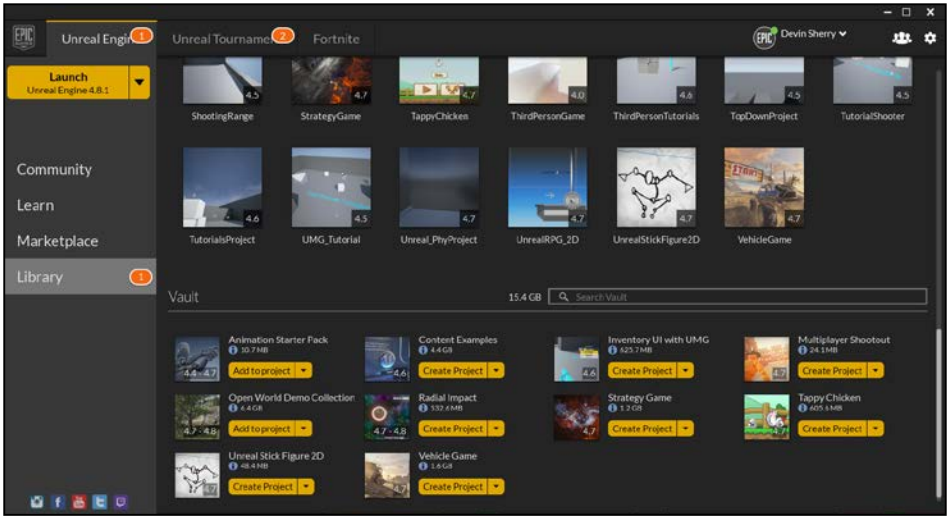
A vehicle in Unreal Engine 4 contains a number of different types of assets:

- A **Skeletal Mesh**
- A **Physics Asset**
- An **Animation Blueprint**
- A **Vehicle Blueprint**
- One or more **Wheel Blueprints**
- A **Tire Type Data Asset**

Let's start by creating the necessary game project so that we have access to a **Vehicle Skeletal Mesh** by default, and we don't have to create our own in a third-party 3D modeling program. To do so, let's open the Epic Games Unreal launcher and navigate to the **Learn** tab. Here, scroll down to the **Example Game Projects** section and find the **Vehicle Game** project template. Select this project and then the **Download** option:



Once successfully downloaded, we can create the project by navigating to the **Library** tab of the Unreal Engine launcher, scroll to the very bottom of the page to the **Vault** section, and select the **Create Project** option for **Vehicle Game**:

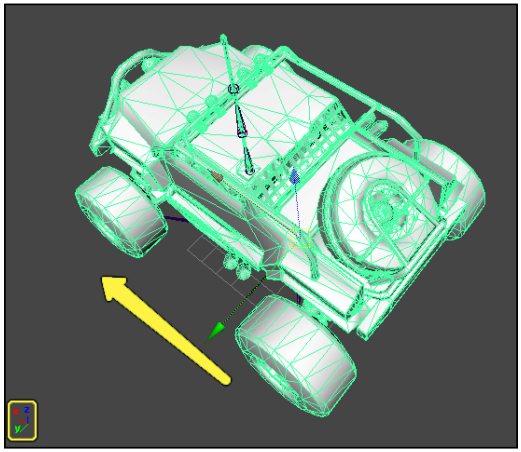


When we select the **Create Project** option, it will ask us for a name of the project. Let's call this project `Vehicle_PhyProject`. After the Unreal Engine launcher creates the vehicle project for us, we should see it available in our list of projects in the **My Projects** section of the **Library** tab. Now, double-click on the project image to open the Unreal Engine 4 editor for this project. By default, this project contains all the assets necessary to create a working **Vehicle Blueprint**, such as a **Skeletal Mesh**, **Wheel Blueprints**, a **Vehicle Animation Blueprint**, and so on, but we will only use the **Skeletal Mesh** that the project provides, and we will create every other aspect of the **Vehicle Blueprint** from scratch step by step.

Now that we have successfully created the **Vehicle Game Project**, feel free to explore some of the content that it contains and play the game in the **Desert Rally Race** level to see how our final result will be for this chapter. Once we are satisfied exploring the game project, let's begin by creating a new folder in **Content Browser** that will house all of our content for this chapter. First, navigate to **Content Browser** and highlight the **Content** folder at the very top of the content hierarchy in the top-left corner of the browser. Once highlighted, we can either right-click on the **Content** folder, select the **New Folder** option, or left-click on the **Add New** drop-down menu and select **New Folder**. Name this folder `VehicleContent`. With the folder in place, we will now navigate to the **Vehicles** folder, the **VH_Buggy** folder and then to the **Mesh** folder; this folder contains the **SK_Buggy_Vehicle Skeletal Mesh** that we need to begin this lesson. Let's left-click and drag the **SK_Buggy_Vehicle** asset to our **VehicleContent** folder and select it to create a copy. Name this copy `SK_Buggy_NewVehicle`. If we want to create our own **Skeletal Mesh**, here are some of the things we should keep in mind.

The basic, bare minimum, art setup required to create a proper vehicle is just a **Skeletal Mesh**. The type of vehicle will dictate how complicated an art setup we will need, and special considerations may need to be given to the suspension. For example, a tank does not require a special suspension setup, whereas a dune buggy (such as the one in the **Vehicle Game** project example) will require additional joints to make the exposed components move in a believable way.

Some of the more important basic information we need to know about setting up our vehicle in a third-party art program (such as 3ds Max or Maya) is that we want the vehicle mesh to point to the positive *X* direction. Next, we will need to measure the radius of our wheels in centimeters for use in Unreal Engine 4 because we had discussed earlier in this book that Unreal Engine 4 uses centimeters as its unit of measurement, where 1 **Unreal Unit** (**uu**) is equal to 1 **centimeter** (**cm**). The minimum number of **Joints** required for a four-wheeled vehicle is 5:1 and 4 wheels; this will change depending on the number of wheels the vehicle has (remember what we discussed in *Chapter 5*, *Physics Damping, Friction, and Physics Bodies*). The wheel and root joints should be aligned with the *X* direction looking forward and the *Z* direction looking upwards. By doing so, this will ensure that the wheel will roll on the *Y* axis and steer on the *Z* axis. All the other joints can be arranged as required, but it should be noted that things such as **Look At** nodes for the **Animation Blueprint** assume that the *X* direction is forward. To prevent visual oddities, the joints for our wheels should be accurately centered, as shown in the following screenshot:

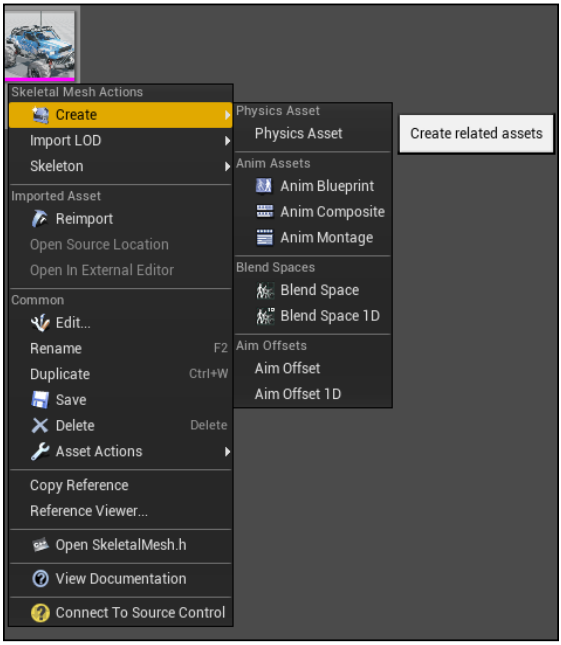


The visual mesh will not be used for collision detection; however, if the wheel mesh is off-center, it will look as if the wheel is broken and will be really noticeable due to motion blur.

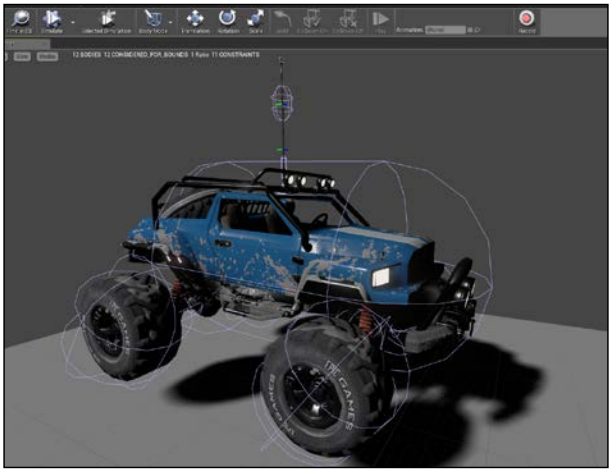
For binding purposes, we can use either the standard smooth bind for Maya or the skin modifier for 3ds Max. Wheels should only have weights on one joint so that they can spin free with no odd deformation. For shocks and struts, we can get away with some fancy skinning, but it will require more thought on the Unreal Engine Editor side.

Lastly, vehicles are simply exported as **Skeletal Meshes** with no special considerations when you import the asset to Unreal Engine 4.

Now that we have our own copy of the **Skeletal Mesh** vehicle, we can create our **Physics Asset** for this vehicle by right-clicking on **SK_Buggy_NewVehicle** from our **VehicleContent** folder, selecting the **Create** option from the drop-down menu and then **Physics Asset**. Then, name this asset `PA_Buggy_NewVehicle` and leave all the setup options to their default values, as shown in the following screenshot:

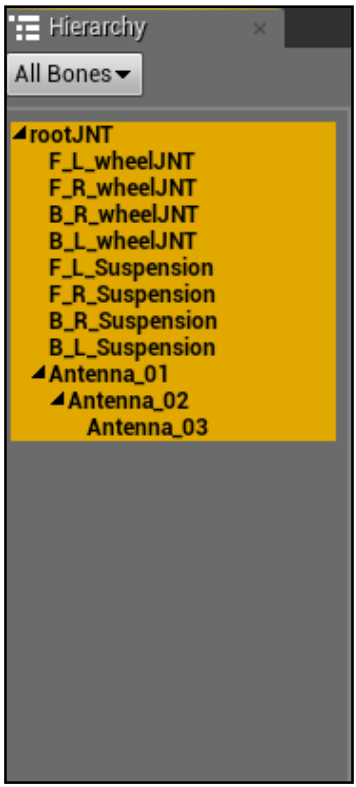


Now, double-click on the new **Physics Asset**, and we should see something similar to this screenshot in **PhAT**:



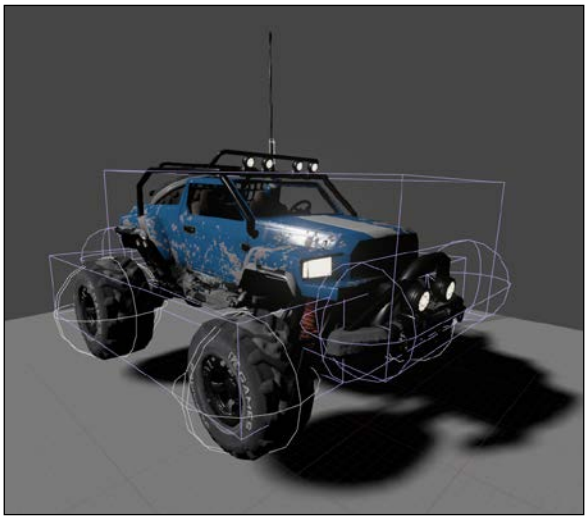
We obtain this result because the **Physics Asset Tool (PhAT)** in Unreal Engine 4 attempts to wrap the vertices that are skinned to a joint as best as it can. **PhAT** does not currently have a way to effectively handle the recreation of the constraints that hold all the **Physics Bodies** together, so what we need to do is delete all the existing **Physics Bodies** in **Hierarchy** so that we can start building them from the root joint. By doing so, all of our constraints will be created correctly.

To do this, navigate to **Hierarchy**, press *Shift* and left-click on all the options, and press the *Delete* key; this will remove all the **Physics Bodies** from the asset:



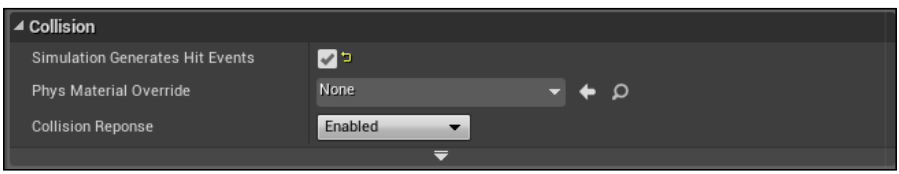
Starting with the root joint: **rootJNT**, let's create the **Physics Bodies** on the joints of our vehicles. Keep in mind that we only need a **Physics Body** on a joint that either needs to be physically simulated or affects the bounds of our vehicle. For our vehicle, a box shape for the root/main body and spheres for each of the wheels will serve us just fine, but we will add additional **Physics Bodies** to get the desired behavior that we want for other parts of the vehicle (such as the antenna).

For our **Buggy Vehicle**, we will have a total of 10 **Physics Bodies**. The end result should look similar to the following image:



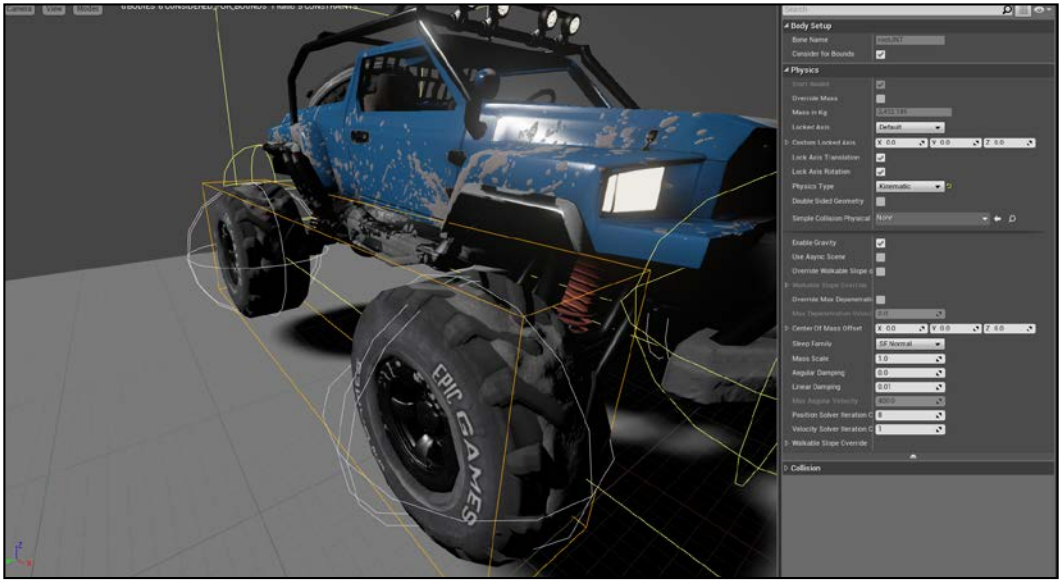
To accomplish this, we will first create the large bounding box that surrounds the main body of **Buggy Vehicle**. Select the **rootJNT** option from the **Hierarchy** panel on the right-hand side and then right-click and select the **Add Box** option. Make sure to use the **Translation** and **Scale** tools to shape the **Physics Body** box to match the shape of the body of the buggy as best as you can. Next, let's navigate to the **Collision** section of the **Details** panel and make sure that **Simulation Generates Hit Events** is set to `True`. Lastly, in the **Details** panel of the newly created box, **Physics Body**, make sure that **Physics Type** is set to `Default`.

Now, let's create the **Physics Bodies** for the four wheels of our vehicle. Each **Physics Body** will be of the same size and shape, and have the same properties associated to them. Let's select **F_L_wheelJNT** from the **Hierarchy** panel, right-click on this option, and select the **Add Sphere** option. Use the **Scale** and **Translation** tools to position the spherical **Physics Body** around the front-left wheel of the vehicle and change its **Physics Type** to **Kinematic**. Follow this process for the remaining three wheels: **F_R_wheelJNT**, **B_R_wheelJNT**, and **B_K_wheelJNT**. Lastly, let's navigate to the **Collision** section of the **Details** panel for each of the four wheels and make sure that **Simulation Generates Hit Events** is set to `True`.



Moving on, let's now create the **Physics Bodies** for the front and back bumpers of the vehicle by right-clicking on **rootJNT** and selecting the **Add Sphyl** option, which will create a capsule-shaped **Physics Body**. Use the **Scale** and **Translation** tools to position this **Physics Body** around the front bumper of the vehicle and set its **Physics Type** to `Default`. Repeat this process for the back bumper of the vehicle as well. Lastly, let's navigate to the **Collision** section of the **Details** option for both. Make sure that **Simulation Generates Hit Events** is set to `True`.

Before we move on to creating the **Physics Body** for the antenna, let's first create the two box shapes for the left-hand side and the right-hand side suspensions for the buggy. With the **rootJNT** bone joint selected in **Hierarchy** on the right-hand side, right-click and select the **Add Box** option. Lastly, let's navigate to the **Collision** section of the **Details** panel and make sure that **Simulation Generates Hit Events** is set to `True`. Then, the scale and position of the box should look similar to the following screenshot:

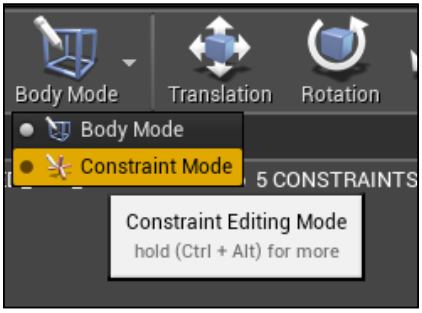


Now, create an additional **Box Physics Body** and shape and transform it so that it covers the other side of the buggy's suspensions. Then, set both their **Physics Type** parameters to `Default`. Next, let's navigate to the **Collision** section of the **Details** panel and make sure that **Simulation Generates Hit Events** is set to `True`.

Lastly, let's set up the **Physics Body** for the antenna of our vehicle so that we can simulate a responsive antenna when we move in our vehicle. To do this, select the **Antenna_01** option from the **Hierarchy** panel, right-click on it, and select the **Add Sphyl** option. Next, set **Physics Type** to `Default`, and set the following parameters in the **Details** panel for our antenna's **Physics Body**:

- **Mass Scale**: Set this parameter to `0.01`
- **Angular Damping**: Set this parameter to `10.0`
- **Linear Damping**: Set this parameter to `3.0`

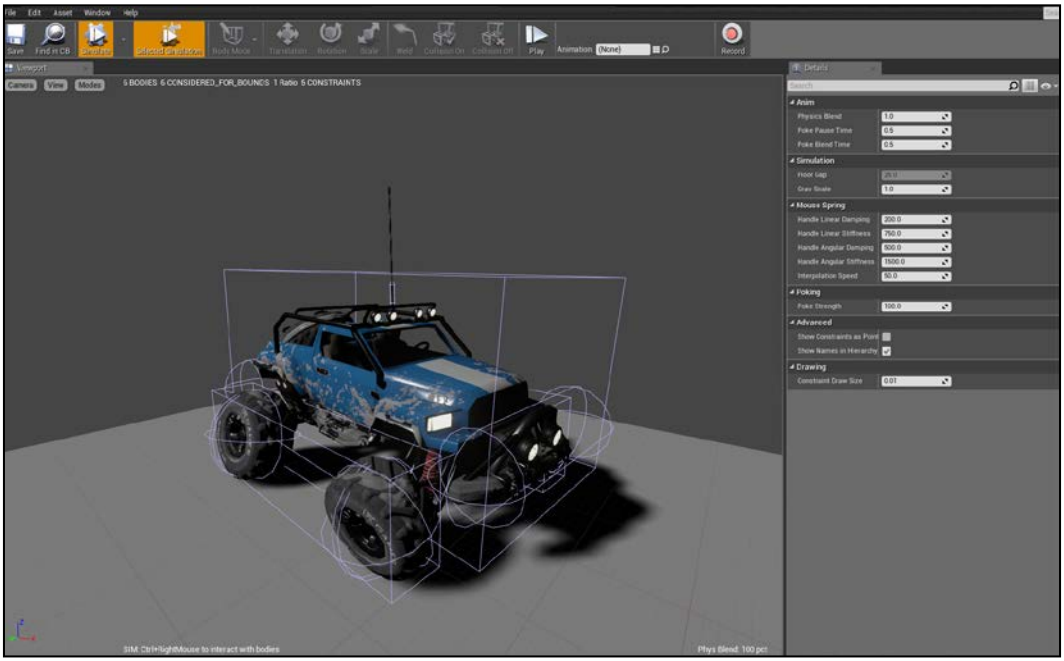
To finish off the **Physics Body** for our antenna, make sure that the **Antenna_01** joint is selected in the **Hierarchy** panel. Then, select **Constraint Mode** from the **Body Mode** drop-down menu:



While in **Constraint Mode** and with the **Antenna_01** joint selected, set the following parameters in the **Details Panel**:

- **Angular Swing 1 Motion**: Set this parameter to `Limited`
- **Angular Twist Motion**: Set this parameter to `Locked`
- **Angular Swing 2 Motion**: Set this parameter to `Limited`
- **Swing 1 Limit Angle**: Set this parameter to `1.0`
- **Swing 2 Limit Angle**: Set this parameter to `1.0`
- **Swing Limit Stiffness**: Set this parameter to `500.0`
- **Swing Limit Damping**: Set this parameter to `50.0`

With these changes in place, let's return to **Body Mode**, select **rootJNT** from the **Hierarchy** panel and then the **Selected Simulation** option, and select **Simulate** to see how our **Physics Bodies** are affected by the gravity of simulation:



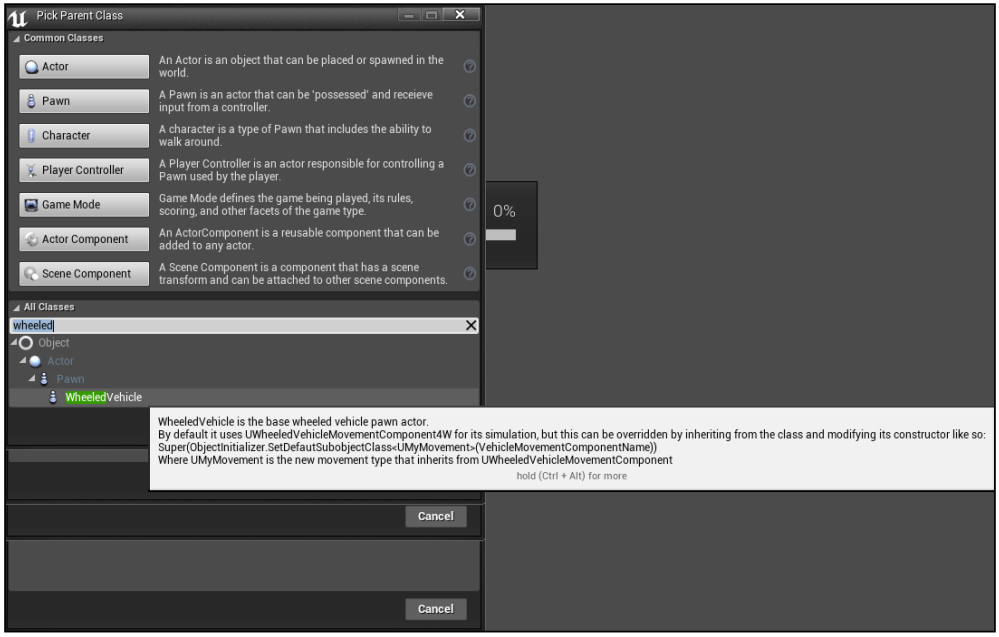
We can see that the wheels rotate in a strange manner that don't make too much sense in the context of a working vehicle for a racing game, but we will change this behavior later when we create the blueprints for the vehicle.

Vehicle Blueprints – a section overview

In this section, we discussed the necessary components that make up a working **Vehicle Blueprint**, and we looked at the necessary details of how to create the **Physics Bodies** for our vehicle. Lastly, we used **PhAT** in Unreal Engine 4 to recreate the necessary **Physics Body** components of the **Buggy Vehicle** to establish a working **Physical Body** for the vehicle. Now that we have created the **Physics Asset** from the premade **Skeletal Mesh** that is created by default when working with the **VehicleGame** project, we can now move on and work on **Vehicle Blueprints**.

Creating the Vehicle Blueprints

To create a new **Vehicle Blueprint**, let's first navigate to our **VehicleContent** folder in **Content Browser** and then right-click on an area that is empty. From the context drop-menu, we will select the **Blueprint Class** option, click on the drop-down **All Classes** menu, search for the **WheeledVehicle Pawn** class, and name this blueprint `BP_NewVehicle`.

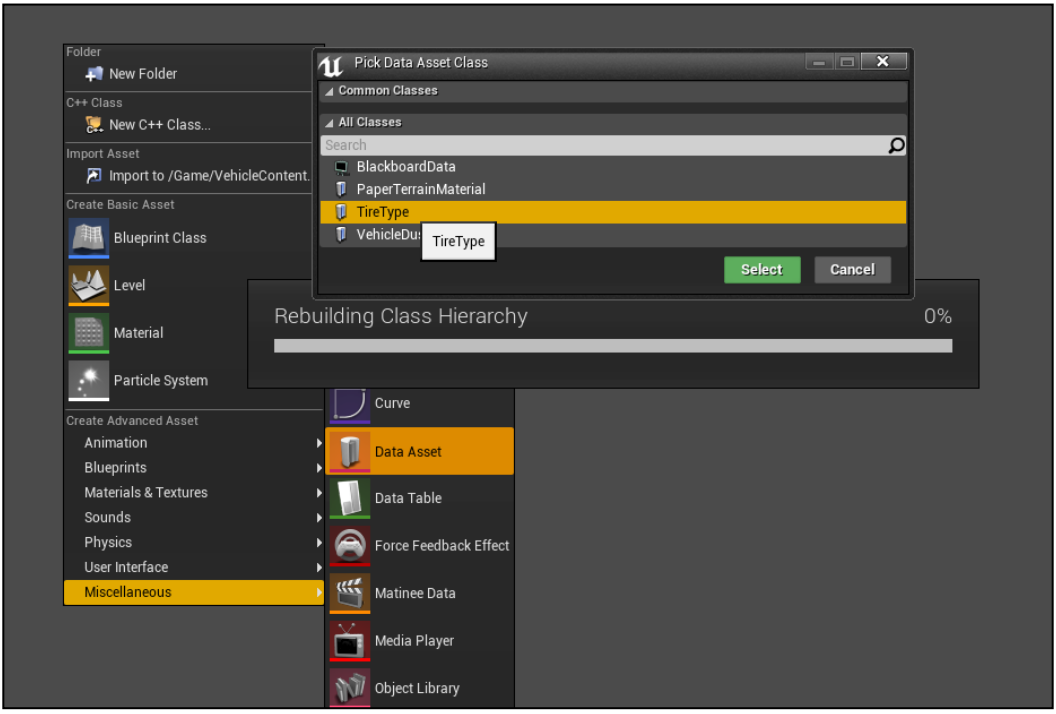


The **WheeledVehicle Pawn** blueprint contains an inherited component called **VehicleMovement**. This component allows you to have more control over the wheels and the overall behavior for the vehicle.

Next, we will need to create two different types of **Wheel Blueprints** for our vehicle (one for the front wheels and one for the back wheels). To get this going, let's navigate to the **VehicleContent** folder in **Content Browser**, right-click on an area of **Content Browser** that is empty, and select the **Blueprint Class** option. Now, in the context sensitive drop-down menu, enter `Vehicle Wheel` to locate the **VehicleWheel Blueprint Object** class. We will create two different **VehicleWheel Blueprint** classes (one named `BP_FrontWheel` and another named `BP_BackWheel`).

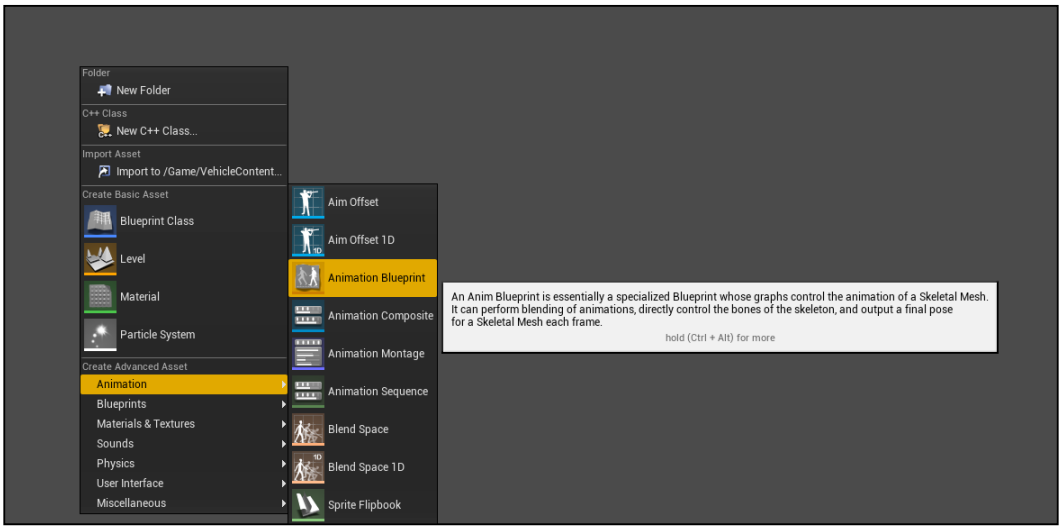
In most cases, we will want to have at least two wheel types: that is, one wheel type that is affected by steering and another that is affected by the vehicle handbrake. Additionally, we can set differing radii, mass, width, handbrake effect, suspension, and many other properties to give our vehicle the handling we desire.

Now, we can move on and create the **TireType** data asset that we will need for our **VehicleWheel** blueprint. To create a new **TireType** data asset in **Content Browser**, we need to right-click on an area of the **VehicleContent** folder that is empty, select the **Miscellaneous** option and then the **TireType** option from the context sensitive drop-down menu that appears:



Let's name this asset `DA_TireType` and then right-click on the asset to open **Generic Asset Editor**. The **TireType** data asset has only one single value: **Friction Scale**. This value not only affects the raw friction of the wheel, but also scales the value for how difficult or easy it is for a wheel to slide during a hard turn. There is a property slot in the **VehicleWheel** blueprint for the **TireType** data asset that we will use once the time comes.

Lastly, we have to create the **Animation** blueprint. We will use this to animate our **Buggy Vehicle**. To do this, navigate to **Content Browser** and then go to the **VehicleContent** folder. Now, in an empty area, right-click and select the **Animation** option from the drop-down menu and then select **Animation Blueprint**:



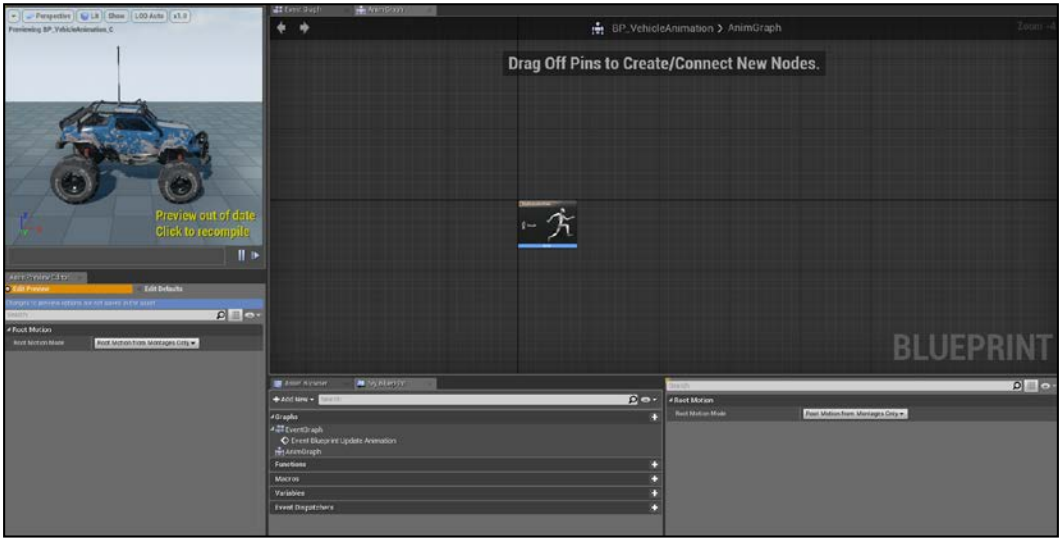
When we first create an **Animation Blueprint**, it will ask us for a **Target Skeleton** to use; we will select the **SK_Buggy_Vehicle_Skeleton** option from the **Target Skeleton** drop-down list. We also want to make sure that we select the **VehicleAnimInstance** option from the **Parent Class** context sensitive drop-down list and name this animation blueprint `BP_VehicleAnimation`. Before we move on and discuss how to edit the different blueprints and data assets we created to complete our vehicle, let's briefly discuss what animation blueprints are.

An **Animation Blueprint** is a specialized blueprint that contains graphs used to control the animation of a skeletal mesh. It can perform blending of animations, has direct control of the bones in a skeleton, and outputs a final pose for our skeletal mesh in each frame. The **Controller** directs the pawn or character to move based on the player input or decisions made based on what the game play dictates. Each pawn has a **Skeletal Mesh** component that references the **Skeletal Mesh** to animate and has an instance of an **Animation Blueprint**.

Through the use of its two graphs, the **Animation Blueprint** can access properties of the owning pawn, compute the values used for blending, state transitions, or driving **Anim Montages**, and can calculate the current pose of the skeletal mesh based on the blending of animation sequences and direct transformations of the skeleton from **Skeletal Controls**. When we work with animation blueprints, we have to keep in mind that there are two main components that work in correlation with one another to create the final animation for each frame. One is **Event Graph** that we can recognize from other blueprints. This is in charge of performing updates to values that can be used in **Anim Graph** to drive **State Machines**, **Blend Spaces**, or other nodes that allows you to blend between multiple animation sequences or poses that fire off notifications to other systems, thereby enabling dynamically-driven animation effects to take place.

There is one **Event Graph** in every **Animation Blueprint** that uses a collection of special animation-based events to initiate sequences of actions. The most common use of **Event Graph** is to update the values used by **Blend Spaces** and other blend nodes to drive animations in the **Anim Graph**.

The **Anim Graph** is used to evaluate the final pose of a skeletal mesh for the current frame. By default, each **Animation Blueprint** has an **Anim Graph**. This graph can contain animation nodes that are placed in it to sample animation sequences, perform animation blends, or control bone transformations using **Skeletal Controls**. The resulting pose is then applied to our **Skeletal Mesh** for each frame in the game.

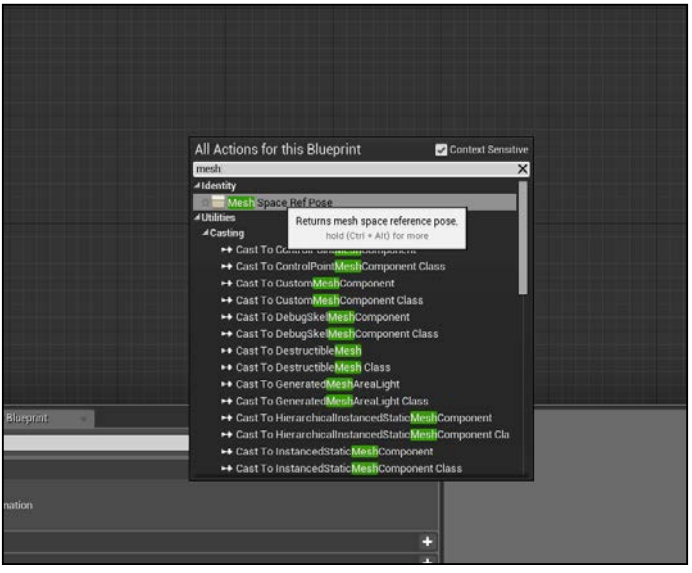


Creating the Vehicle Blueprints – a section review

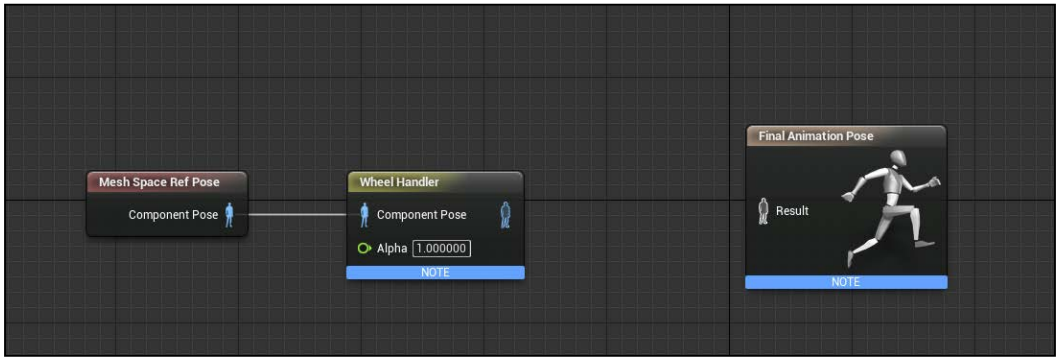
In this section, we looked at the necessary blueprints and data assets. We will move on and edit their properties to obtain the behaviors we need for our working **Buggy Vehicle**. First, we created the **WheeledVehicle** blueprint. This is the main blueprint for our vehicle. Then, we created two types of **Wheel Blueprints** (one for our front wheels and another for our back wheels). Further, we created the **TireType** data asset. This is necessary to control the **Friction** property for our wheels. Lastly, we created our animation blueprint for the **Buggy** skeletal mesh, and we discussed about **Animation Blueprints** and its functionalities in detail. Now that we have created the necessary blueprint and data assets for our vehicle, we can move on and edit the properties of these assets.

Editing the Vehicle Blueprints

With the vehicle blueprints created, let's now move on and edit the properties of these blueprints in order to obtain the behaviors we want for our vehicle. We will begin by working with the **BP_VehicleAnimation** blueprint by double-clicking on our **Content Browser** and opening its **Anim Graph**; which opens by default. The first node we will create is the **Mesh Space Ref Pose** node, and this is used to return the mesh space reference pose for our skeletal mesh in the **Animation Blueprint**. To create this node, right-click on an area of the **Anim Graph** that is empty. Now, from the context menu, we will search for the **Mesh Space Ref Pose** node:



Next, we will need a **Wheel Handler** node. This is used to alter the wheel transformation based on the setup in the **Wheeled Vehicle** blueprint; keep in mind that this will only work when the owner is of the **Wheeled Vehicle** class. The **Wheel Handler** node also handles the animation needs of our wheels, such as the spinning, the steering, the handbrake, and the suspension. There is no additional setup required; this node obtains all the necessary information from the wheels and transforms it into animation on the bone that the wheel is associated with. To create the **Wheel Handler** node in the **Anim Graph** of our **Vehicle Animation** blueprint, we need to right-click on an area of the graph that is empty. Then, from the context-sensitive menu, we can search for **Wheel Handler**. Finally, we can connect the **Component Pose** output of the **Mesh Space Ref Pose** node to the **Component Pose** input of **Wheel Handler**, as shown in the following screenshot:

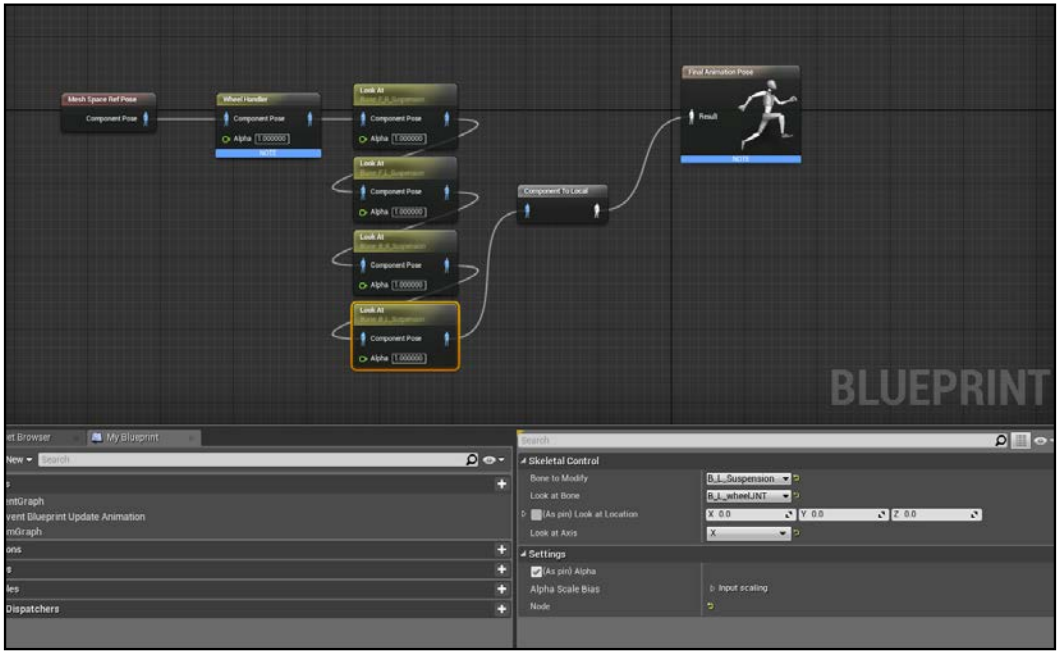


Unless we have additional struts or other suspension needs, we would connect the **Component Pose** output of the **Wheeler Handler** node to the **Result** output node of the **Final Animation Pose**; if we do this, a **Component to Local** node will automatically be generated between the **Wheel Handler** and the **Final Animation Pose** nodes so that it can convert **Component Space Pose** to **Local Space Pose**. As our **Vehicle Physics Asset** and **Vehicle Skeletal Mesh** contain bones for the vehicle suspension, we will want to create additional nodes to handle the joints that affect the suspension polygons. To do this, pull the output of the **Wheel Handler** node and use the context-sensitive drop-down menu; we will search for the **Look At** node. In the **Details** panel under the **Skeletal Control** section of the **Look At** node, we will want to edit the **Bone to Modify** and **Look at Bone** properties so that we can modify the four bones we have on our vehicle's skeletal mesh, and we have the **Look at Bone** look at our wheel joints. Let's create four different **Look At** nodes and set each individual property for the **Bone to Modify** and **Look at Bone** settings:

1. First node:
    - **Bone to Modify**: Select this property as `F_L_Suspension`
    - **Look at Bone**: Select this property as `F_L_wheelJNT`

2. Second node:
    - **Bone to Modify**: Select this property as `F_R_Suspension`
    - **Look at Bone**: Select this property as `F_R_wheelJNT`
3. Third node:
    - **Bone to Modify**: Select this property as `B_R_Suspension`
    - **Look at Bone**: Select this property as `B_R_wheelJNT`
4. Fourth node:
    - **Bone to Modify**: Select this property as `B_L_Suspension`
    - **Look at Bone**: Select this property as `B_L_wheelJNT`

With the four **Look At** nodes in place, we can now connect the output of the last **Look At** node to the **Result** input node of the **Final Animation Pose** node. Our final **Vehicle Animation** blueprint should look similar to the following screenshot:



With these nodes in place, we are done with the **Vehicle Animation** blueprint. We can now move on and edit our **Tire** data asset.

Similar to what we have discussed earlier, the **Tire** data asset only has one property value to edit: **Friction Scale**. Let's navigate back to **Content Browser** and to our **VehicleContent** folder. Now, we double-click on the **DA_Tire Tire Type** asset. In the **Generic Asset Editor**, let's change the **Friction Scale** property from its default value of `1.0` to a new value of `2.0`:

With this change to our **Tire** data asset in place, we can now move on and edit our **Wheel** blueprints. Navigate back to **Content Browser** and to our **VehicleContent** folder so that we can double-click our **BP_BackWheel** blueprint and edit its properties. As discussed previously, the properties of the front and back wheels will vary slightly because the front wheels will be responsible for steering, whereas the back wheels will be responsible for responding to the handbrake.

The properties that we need to initially set are as follows:

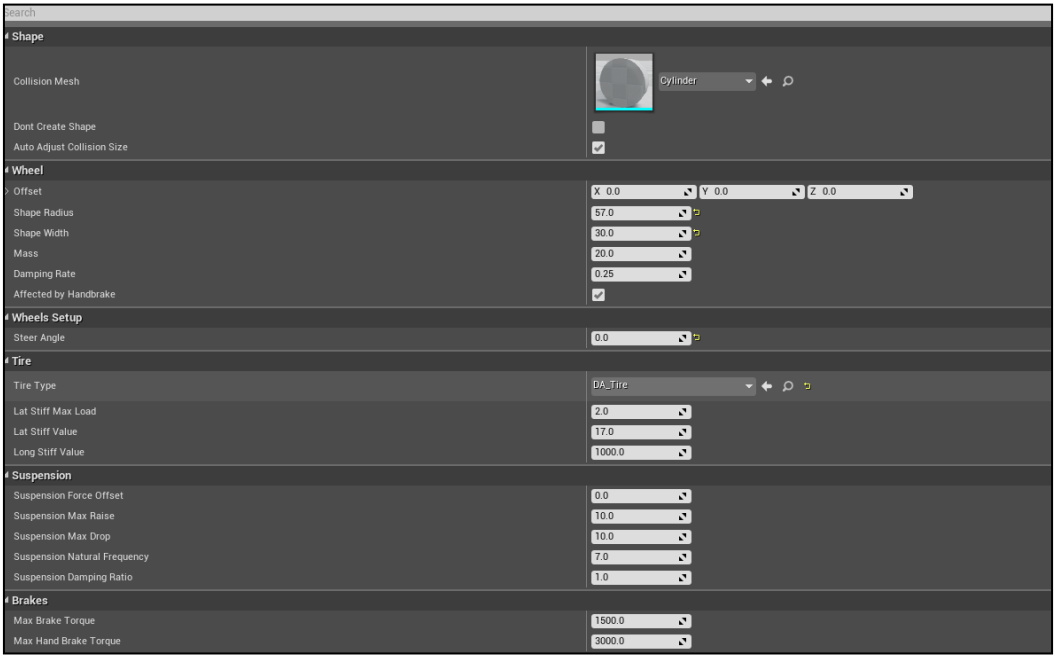
- **Shape Radius**: This property determines the radius of the shape used for the vehicle wheel.
- **Shape Width**: This property determines the width of the shape used for the vehicle wheel.
- **Affected by Handbrake**: This property determines whether or not the wheel is affected by the handbrake that the player uses to stop the vehicle. This parameter is typically used for back wheels only, not for the front wheels.
- **Steer Angle**: This specifies the maximum angle that the wheel can rotate in both the positive and negative directions, that is, turning left and right.
- **Tire Type**: This property determines the **TireType** data asset that the wheel will use for its **Friction Scale** property.

The **Shape Radius** and **Shape Width** properties are determined by the size of the wheels, and in this specific case, these are the back wheels on our vehicle, so for these settings, let's set the following parameters:

- **Shape Radius**: Set this parameter as `57.0`
- **Shape Width**: Set this parameter as `30.0`

Again, keep in mind that these values will change depending on the vehicle being used and the size of the wheels. Next, we will need to change the value of the **Steer Angle** property. As we will work with the **BP_BackWheel Wheel** blueprint, and the back wheel will not control the steering; we will set the **Steer Angle** property from its default value of `70.0` to a value of `0.0`.

Moving on, we need to make sure that the **BP_BackWheel Wheel** blueprint has the **Affected by Handbrake** property set to `True` so that these wheels are affected when the player uses the brakes of the vehicle to slow it down and allow it to stop. Lastly, we need to set **Tire Type** from its default value of `DefaultTireType` to `DA_Tire` from the drop-down menu so that this **Tire Type** is used by our **BP_BackWheel**.



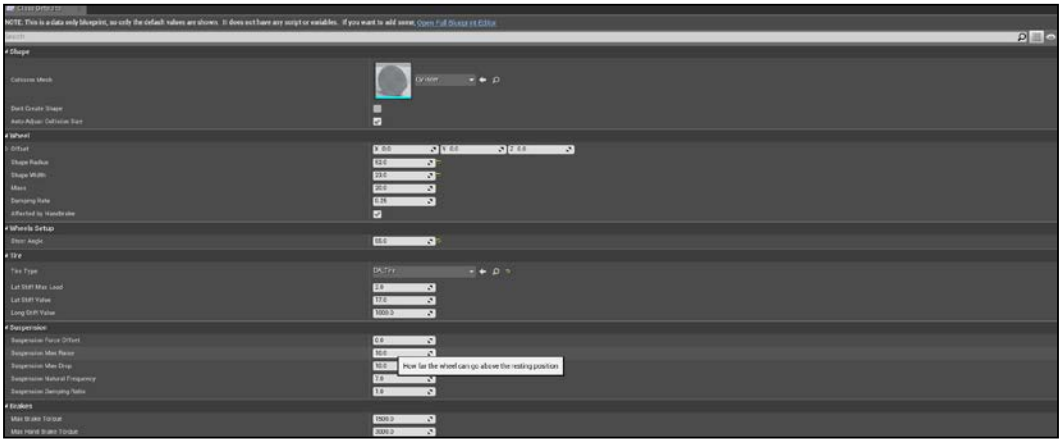
Now that we have completed the **BP_BackWheel** blueprint, let's navigate back to **Content Browser** and to our **VehicleContent** folder and then double-click and open the **BP_FrontWheel** blueprint. If we take a look at the skeletal mesh for our vehicle, we will see that the back wheels are slightly larger than the front wheels; this will be important when you set the values for the **Shape Radius** and **Shape Width** parameters of the **BP_FrontWheel** blueprint. Set the following values for the **Shape Radius** and **Shape Width** properties:

- **Shape Radius**: Set this parameter to `52.0`
- **Shape Width**: Set this parameter to `23.0`

We can see that the shape radius is only 5 units less than the **BP_BackWheel** value and the shape width is only 7.0 units less; this is the difference in **Unreal Units** (**uu**) between the two types of wheels.

As we will work with the **BP_FrontWheel** blueprint, we will want to uncheck the **Affected by Handbrake** property so that it is `False` because the front wheels of our vehicle should not react at all to the handbrake. Before we set the **Steer Angle** parameter, we have to understand that this angle is the max angle that the wheel can rotate in both the positive and negative directions, that is, turning left and right. For our **Buggy Vehicle**, any value between `50` and `60` works best, but for the sake of providing a value for testing purposes, let's set the **Steering Angle** value to `55`.

Last but not least, let's make sure that the **TireType** parameter is using our **DA_Tire** data asset.



Before we move on to editing the **BP_NewVehicle Vehicle** blueprint, let's take some time here to briefly define some of the parameters of our **Wheel** blueprints that we didn't edit so that we have a better understanding of the overall functionality of the **Wheel** blueprint. Here are the additional properties we can manipulate for our **Wheel** blueprint for further customization:

- **Lat Stiff Max Load**: This is the maximum normalized tire load, in which the tire can deliver no more lateral (sideways) stiffness, irrespective of how much extra load is applied to the tire.
- **Lat Stiff Value**: This determines how much lateral stiffness can be given to the lateral slip.
- **Long Stiff Value**: This determines how much longitudinal stiffness can be given to the longitudinal slip.

- **Suspension Force Offset**: This is the vertical offset from the vehicle center of mass where suspension forces are applied.
- **Suspension Max Raise**: This value determines how far the wheel can go above its resting position.
- **Suspension Max Drop**: This value determines how far the wheel can go below its resting position.
- **Suspension Natural Frequency**: This determines the oscillation frequency of suspension; most cars have values between `5` and `10`.
- **Suspension Damping Ratio**: This value is the rate at which energy is dissipated from the spring of the vehicle. Most cars have values between `0.8` and `1.2`; values less than `1` are more sluggish, whereas values greater than `1` is twitchier.
- **Max Brake Torque**: This sets the maximum brake torque of our vehicle in **Newton Meters** (**Nm**).
- **Max Hand Brake Torque**: This property sets the maximum handbrake torque for this wheel in Nm. A handbrake should have a stronger brake torque than the brake. This will be ignored for wheels that are not affected by the handbrake.

Now that we have a better understanding of **Wheel Blueprints**, let's move on to our **Vehicle Blueprint**. Navigate back to **Content Browser** and to our **VehicleContent** folder. Let's double-click on **BP_NewVehicle** and focus on the **Details** panel when we select the **Mesh (Inherited)** component from the **Components** tab in the top-left corner of our blueprint. Keep in mind that we need to click on the **Open Full** blueprint editor before viewing the **Viewport**, **Construction Script**, and **Event Graph** tabs in the blueprint.

The first thing we will do is select the **Mesh (Inherited)** component in the **Components** tab. Then, in its **Details** panel, we will change the **Anim Blueprint Generated Class** property and the **Skeletal Mesh** property. For the **Anim Blueprint Generated Class** property, we want to ensure that our `BP_VehicleAnimation` blueprint is selected from the context-sensitive drop-down menu; we do this because we want our vehicle to use the animation blueprint that we set up earlier. For the **Skeletal Mesh** property, we will use the `SK_Buggy_NewVehicle` skeletal mesh that we made a copy of in the **VehicleContent** folder. With these properties in place, we can now edit the parameters of the **VehicleMovement (Inherited)** component, where we will implement the **Wheel Blueprints** for our vehicle.

Select the **VehicleMovement (Inherited)** component from the **Components** tab. Then, in its **Details** panel, let's find the **Wheel Setups** parameters, where we can set **Wheel Class** and **Bone Names** that we want to use for each individual wheel for our vehicle. In the **Wheel Setups** section, set the following parameters for the four wheels:

1. **Wheel 0**:
    - **Wheel Class**: Set this parameter to `BP_FrontWheel`
    - **Bone Name**: Set this parameter to `F_L_wheelJNT`
2. **Wheel 1**:
    - **Wheel Class**: Set this parameter to `BP_FrontWheel`
    - **Bone Name**: Set this parameter to `F_R_wheelJNT`
3. **Wheel 2**:
    - **Wheel Class**: Set this parameter to `BP_BackWheel`
    - **Bone Name**: Set this parameter to `B_L_wheelJNT`
4. **Wheel 3**:
    - **Wheel Class**: Set this parameter to `BP_BackWheel`
    - **Bone Name**: Set this parameter to `B_R_wheelJNT`

Keep in mind that when we use a unique skeletal mesh, the **Bone Name** properties will be different depending on how they are named. We can also add more wheels to the vehicle setup by clicking on the **+** sign next to the **Wheel Setups** option.

The last thing we need to do for this **Vehicle Blueprint** is implement a third-person view camera position behind and slightly above our vehicle. To create a **Camera** component, we need to navigate to the **Components** tab. From the **Add Component** option, we can search the **Camera** component. Name this component `VehicleCamera` and set its position and rotation values as follows:

- **Location**:
    - **X**: Set value to `-490.0`
    - **Y**: Set value to `0.0`
    - **Z**: Set value to `310.0`
- **Rotation**:
    - **Roll (X)**: Set value to `0.0`
    - **Pitch (Y)**: Set value to `-10.0`
    - **Yaw (Z)**: Set value to `0.0`

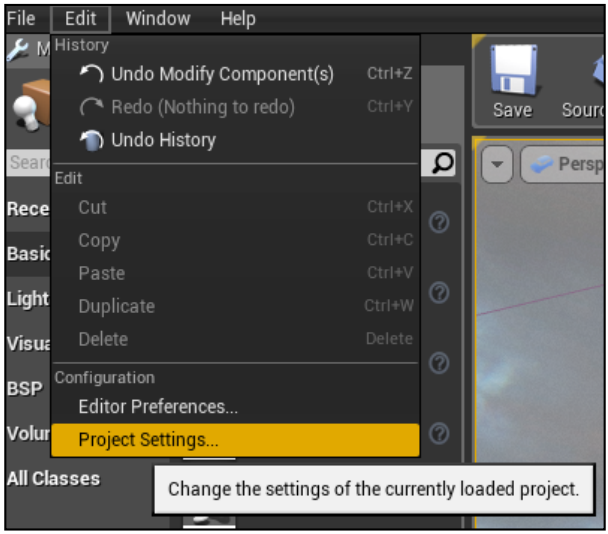
Finally, we want to make sure that the **Use Pawn Control Rotation** option from the **Details** panel of the **VehicleCamera** component is unchecked so that it is set to `False`. With these parameters in place, we are now ready to start setting up our **User Controls** so that we can begin to test our **Vehicle Blueprint**.

Editing the Vehicle Blueprints – a section review

In this section, we set up the basic functionality for all of our vehicle blueprints. First, we added the functionality to our **Vehicle** animation blueprint by creating a **Mesh Space Ref Pose** node, connected it to a **Wheel Handler** node, and implemented four different **Look At** nodes for each of our **Suspension Bones** that are attached to our vehicle. Next, we set up the **Friction Scale** value for our **Tire Type** data asset. Then, we set up the parameters required for our two different **Wheel Blueprints** so that we get the appropriate behaviors for our front and back wheels. Lastly, we set up the parameters for our **Vehicle Blueprint** by applying the necessary skeletal mesh and animal blueprint for the vehicle. We also took the time to implement the two **Wheel Blueprints** and associated them with the four wheel bones of the vehicle's skeletal mesh. With these blueprints in place, we can now implement the user controls for our vehicle so that the player can actually drive the vehicle in the game environment.

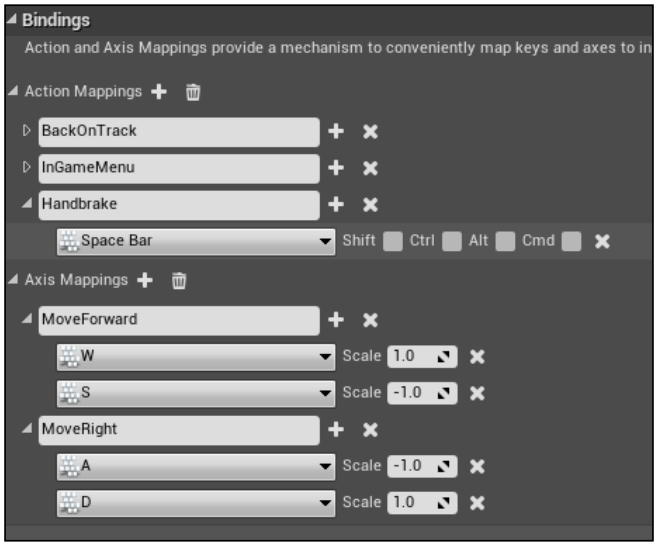
Setting up user controls

When we use the **Vehicle Game** project example, there are default **User Inputs** already in place that allows you to control the vehicle in the game, but we will take a look at the input settings so that we have a better understanding of what they are. To view the current input controls, let's navigate to **Project Settings** by first left-clicking on the **Edit** drop-down and selecting **Project Settings**; be sure to either be in a blueprint or a level to gain access to **Edit** options:



From **Project Settings**, navigate to the **Input** option in the **Engine** section so that we gain access to **Action Mappings** and **Axis Mappings** for our controls. By default, we already have the **MoveForward** mapping and the **MoveRight** mapping in place that utilize a combination of keyboard keys and gamepad buttons; for our purposes, we will only need to use a few of these buttons. Let's expand the **Axis Mappings** drop-down list and first view the **MoveForward** option; we will see multiple buttons that are used to move our vehicle forward, such as the *W*, *S*, up, and down keys for example. We will remove all the options, except the *W* and *S* keys to ensure that we don't have any unnecessary key bindings for our controls; to do this, just click on the **X** button located next to each option to remove it. We will also see a **Scale** value next to each key binding: `1` and `-1`. This refers to the direction that the **MoveForward** control will actually move the player or vehicle forward or backward; the same idea applies to the **MoveRight** option as well. Let's expand the **MoveRight Axis** mapping and remove all the key bindings, except the *A* and *D* keys.

The last thing we want to do here is evaluate **Handbrake Action Mapping**. By default, we have multiple key bindings, but we want to remove all of them, except the **Space Bar** option:



Before we move on, let's briefly discuss the differences between **Action Mappings** and **Axis Mappings**:

- **Action Mapping**: These mappings are for key presses and releases, such as the pressing and releasing of *spacebar*
- **Axis Mapping**: These mappings allow inputs that have a continuous range and direction

Overall, **Action** and **Axis Mappings** provide a mechanism to easily map keys and axes to input behaviors by inserting a layer of indirection between an input behavior and the keys or the game pad buttons that initiate it.

The final step is for us to create a **Game Mode** blueprint so that when we play in-editor, we are able to drive around in our vehicle. Let's navigate back to **Content Browser** and to our **VehicleContent** folder so that we can create the **Game Mode** blueprint. Once in the **VehicleContent** folder, let's right-click on an area of the **Content** folder that is empty, select the **Blueprint Class** option. Then, from **Common Classes**, select the **Game Mode** option and name this new blueprint `BP_VehicleGameMode`. Now, double-click on this new blueprint, and under the **Details** panel, we will find the section labeled as **Classes**. We will change the **Default Pawn** class from `DefaultPawn` to `BP_NewVehicle`. This ensures that when we play the game, by default, it will use our **BP_NewVehicle Pawn** class.

The last thing we need to do is apply this new **BP_VehicleGameMode** to our **Project Settings** by navigating back to **Project Settings**, and under the **Project** section, we will find the **Maps & Modes** option. It's here that we can apply **BP_VehicleGameMode** by expanding the **Default Modes** section. Now, from the **Default Game Mode** drop-down list, we can select the `BP_VehicleGameMode` option. For future reference, when we create levels, we can navigate to the **Settings** option while in the main **Level Editor** and select **World Settings**. This allows you to view your **World Settings** located next to the **Details** panel on the right-hand side of the screen. In **World Settings**, we will find the **Game Mode** section. Here, we can see the **Game Mode Override** parameter and select **BP_VehicleGameMode**. With this in place, we can play the game and see our vehicle in action, but we will see that we are unable to move our vehicle when we press the *W*, *A*, *S*, and *D* keys.

We can now move on and add the input action events in our **BP_NewVehicle** so that we are able to move around and control our vehicle.

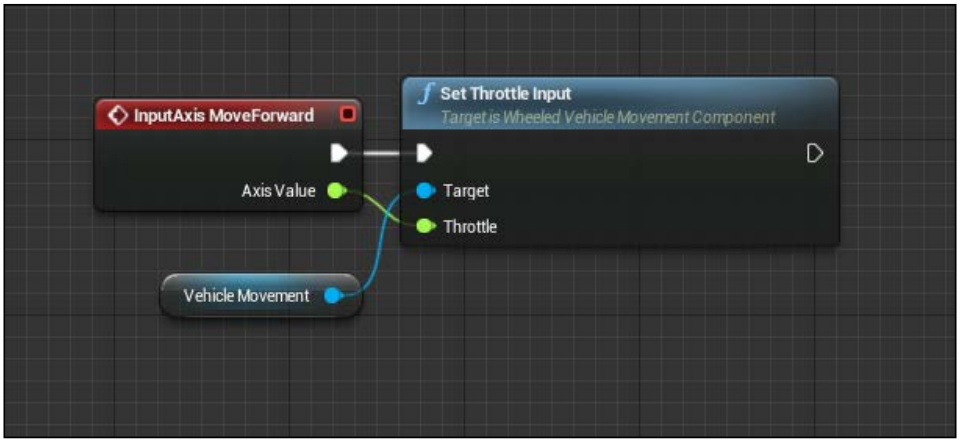
Setting up user controls – a section review

In this section, we created the **Axis Mappings** so that the vehicle could gain the ability to move forward/backward and right/left. We also implemented the ability to use the handbrake with the **Action Mapping** in **Project Settings**. Lastly, we created a new **Game Mode** blueprint and implemented the **Game Mode** in the **Project** and **World Settings** of our level. With these in place, we can move on and add behaviors to our **BP_NewVehicle Event Graph**. This allows you to control your vehicle.

Scripting movement behaviors

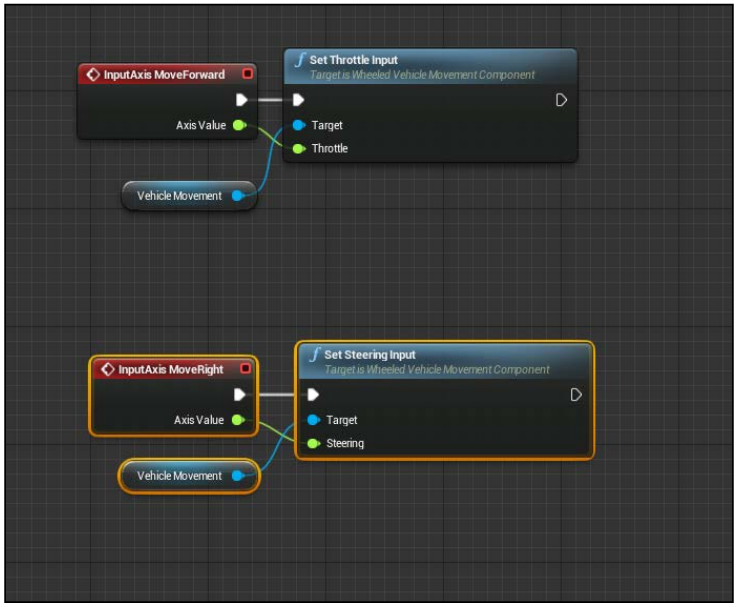
Before we can have our vehicle move through various player controls, we need to script the blueprint behaviors in **BP_NewVehicle Event Graph** by taking advantage of the **VehicleMovement (Inherited) Component** variable. To start with, let's navigate to **Content Browser** and to our **VehicleContent** folder so that we can double-click and open **BP_NewVehicle**.

In an empty area of **Event Graph**, let's right-click and use the context-sensitive drop-down menu to search for our **Input Axis MoveForward** event node so that we can control the forward and backward throttle of our vehicle. Next, we need to grab a **Get** variable of the **VehicleMovement (Inherited)** component. To do this, we have to hold down the *CTRL* key and then click and drag the **VehicleMovement** component from the **Components** tab onto our **Event Graph**. Then, we can pull the **VehicleMovement** variable and search for the **Set Throttle Input** action node from the context-sensitive drop-down menu that appears. Finally, we can connect the main executable pin of **Input Axis MoveForward Event** to the input executable pin of the **Set Throttle Input** node, and we need to connect the **Axis Value** float output of our event to the **Throttle** float value input, as shown in the following screenshot:

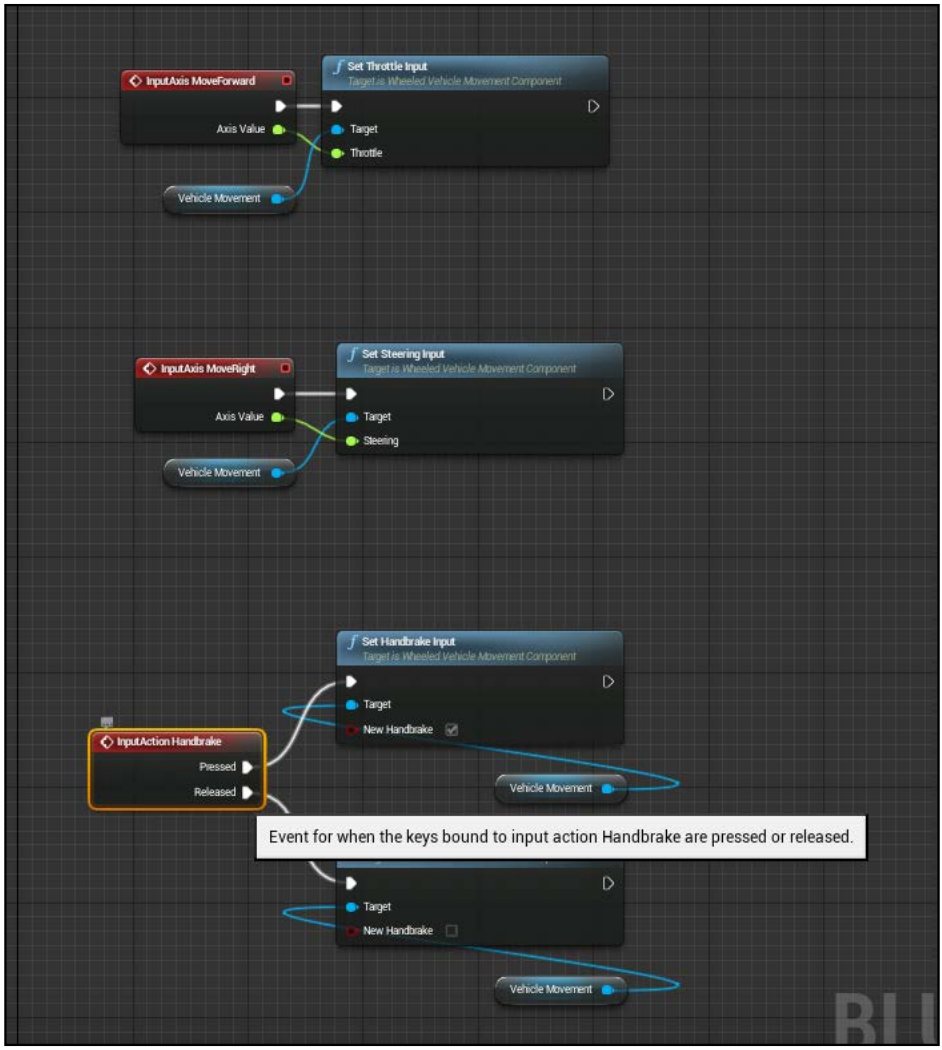


What this logic does is it uses **Axis Value** for our **Input Axis MoveForward** option, which will either be `1` or `-1` depending on the keys that are pressed, and applies this value to the **Throttle** of our vehicle, which results in our vehicle moving forward or backward.

Next, let's set up the logic to steer our vehicle by right-clicking on an empty area of our **Event Graph** and search for the **Input Axis MoveRight** event. We will also need a copy of the **Vehicle Movement Component** variable so that we can pull this copy and search for the **Set Steering Input** action node from the context sensitive drop-down menu. Connect the output executable pin of the **Input Axis MoveRight** event node to the input executable pin of the **Set Steering Input** node. Also, connect the **Axis Value Float** output to the **Steering Float** input, as shown in the following screenshot:



Lastly, we need to set up the logic for the handbrake so that when the player presses and releases the *spacebar*, the handbrake will react appropriately to the input. To begin with, let's right-click on an empty area of **Event Graph** and search for the **Input Action Handbrake** event node. Next, we will need to create a copy of the **Vehicle Movement Component** variable, and from this variable, we need to pull from it and search for the **Set Handbrake Input** action node from the context-sensitive drop-down menu. We then need to check the **New Handbrake Boolean** input variable of the **Set Handbrake Input** node so that it uses the handbrake of our vehicle to come to a halt. Next, we need to create a copy of the **Vehicle Movement Component** variable and the **Set Handbrake Input** node, but for this copy, we want to make sure that the **New Handbrake Boolean** input variable is unchecked. Lastly, we need to connect the **Pressed** output executable pin of the **Input Action Handbrake** node to the input executable pin of the **Set Handbrake Input** node that has its **New Handbrake Boolean** set to `True`. Then, connect the **Released** output executable pin of the **Input Action Handbrake** node to the input executable pin of the **Set Handbrake Input** node that has its **New Handbrake** set to `False`.



With the logic in place in our **BP_NewVehicle Vehicle** blueprint, we can compile and save the content and then navigate to the **DesertRallyRace** level so that we can play in-editor and test our vehicle. Again, make sure that **World Settings** has the **GameMode Override** parameter set to our **BP_VehicleGameMode** blueprint before the testing phase.

Now, when we play the game, we will be able to move our vehicle forward and backward with the *W* and *S* keys, steer the vehicle with the *A* and *D* keys, and use the handbrake with *spacebar* to have our vehicle come to a halt. We will also see that the wheels spin when it moves either forward or backward, the front wheels turn in the direction we press, and the physics of our vehicle work as expected.

Scripting movement behaviors – a section review

In this section, we worked on scripting the necessary behaviors of our vehicle so that when we play the game, we were able to move and steer our vehicle. First, we implemented **Set Throttle Input** in conjunction with our **Input Axis MoveForward** event node. Then, we used the **Set Steering Input** node with our **Input Axis MoveRight** event. Lastly, we associated the **Set Handbrake** node functionality with the **Input Action Handbrake** event node. Now that we are able to drive our vehicle in the game, we can evaluate its behavior and test how it feels.

Testing the vehicle

When we test our vehicle, we have to keep in mind the controls and the feel we are trying to create when we drive the vehicle. The behaviors of a vehicle will drastically differ depending on the type of gameplay we are going for, such as the drastic difference between the vehicles in Mario Kart as compared to the ones from the Forza series.

If tweaks or changes are necessary to obtain the desired behavior, the main aspect to view is the **VehicleMovement (Inherited)** component in the **BP_NewVehicle** blueprint, where it has various parameters in its **Details** panel that we can change in order to change the behavior of the vehicle, such as **Differential Setup** or **Transmission Setup**. We can also use **VH_Buggy** and the other default vehicle content that is provided by Epic Games when we use the **Vehicle Game Project** example as a reference point to change the way our vehicle behaves.

Use the vehicle we have created in this chapter as a stepping stone to create a unique vehicle that behaves in different ways. Also, feel free to play around with the settings in the **Animation**, **Wheel**, and **Vehicle Blueprints** to see what we can create.

Summary

In this chapter, we created our own working vehicle with the **Vehicle Game Project Example** template step by step from scratch. In the process of doing so, we accomplished certain tasks.

First, we downloaded and created a project using the **Vehicle Game Project Example** template so that we could have access to several resources and content available to create a basic vehicle and a template racing game. Then, we created our own **Physics Asset** using **PhAT** with the default buggy skeletal mesh as a base and implemented our own **Physics Bodies** to the vehicle.

Next, we created all the necessary blueprints and data assets required in constructing a working vehicle for our game. To begin with, we created a **Wheel Vehicle Blueprint** component that contained the **VehicleMovement (Inherited) Component** class. Then, we created two different types of **Wheel Blueprints** (one for the front wheels and another for the back wheels). Each has its own set of unique parameters. Lastly, we created the **Vehicle Animation Blueprint** component required to obtain the proper motion of our wheels when we drive.

Additionally, we then edited each of these blueprints so that we could obtain the proper behavior for our vehicle. We also set up the user controls for our vehicle by editing **Input Action** and **Axis Mappings** so that the appropriate key bindings were set for our vehicle to move forward/backward in order to use the handbrake and steer left/right.

Then, we implemented the **Blueprint** logic within our **BP_NewVehicle Wheeled Vehicle Blueprint** by implementing the **Input Action** and **Axis Mapping** event nodes to the appropriate **Vehicle Movement** actions such as setting the throttle value, and the steering input values.

Lastly, we set up our own **Game Mode Blueprint** that utilizes our **BP_NewVehicle Pawn Blueprint** class and implemented that **Game Mode** into the **Project Settings**, as well as to the **World Settings** of our level. From there, we were able to play in-game and drive our vehicle around the level, and we posed the challenge of changing the parameters of our **BP_NewVehicle** in order to obtain unique behaviors for our vehicle.

In the next chapter, we will be covering advanced physics topics and troubleshooting concepts like pragmatic physics.

8
Advanced Topics

Mixing multiple physical rules of an object and customizing physical properties is one of the new and powerful features in Unreal Engine 4. This allows designers and developers to apply physics on a larger scale. Things such as ocean water, sky, a grass field, and object destruction are examples of how to apply multiple physics for player interactions in the game world.

This chapter provides an example of various physics rules of a simple object.

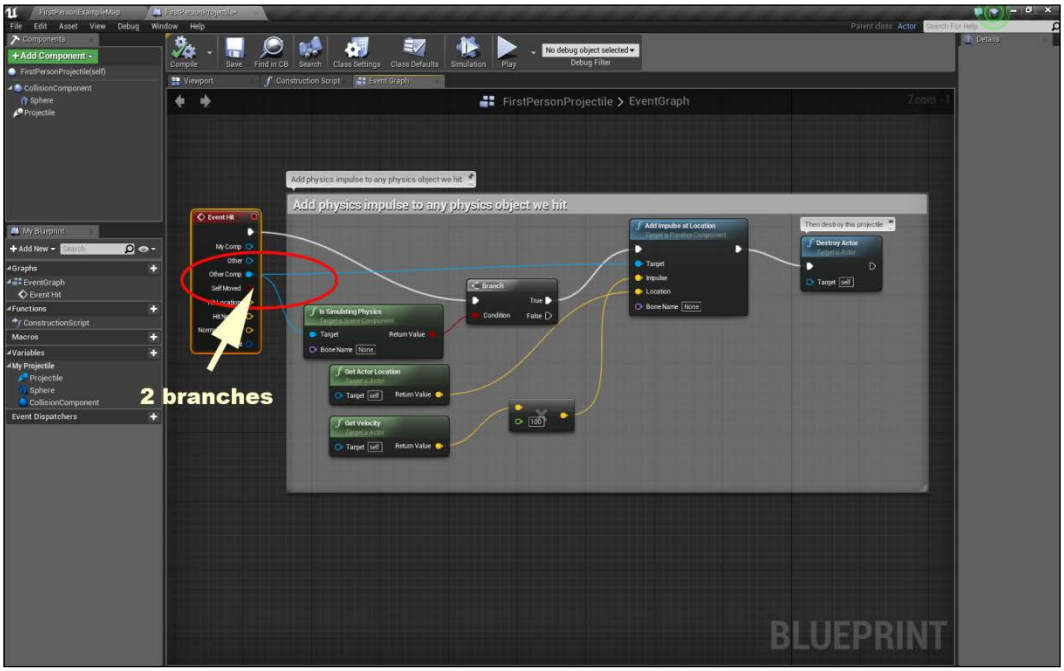
Simulating complex physics – destruction

If you want to destroy an object in the game into small pieces, you have to break it into small objects and save the model as a huge file that needs heavy processing by the machine to render, and it also takes time to make animation for each piece. Some designers have ignored this method by replacing objects with some particle system over the object position.

Today, Unreal Engine 4 not only solves this issue, but also provides features and properties to customize the destruction of an object. Depending on the artistic or reality-based features of the game, you can simulate the way the energy flows and destroys the object and control the physical aspects of the target at the same time. Each destruction is simply an interaction or collision between two objects. This displays special visual presentations during a specific period of time in the game world. In our example, we will simulate the bullet from the first person shooter map in UE4 over a simple cube. For this, first we need to define our bullet blueprint and then work on the cube object to display the destruction after the hit. Perform the following steps:

1. First, open Unreal Editor by clicking on the **Launch** button from Unreal Engine launcher.

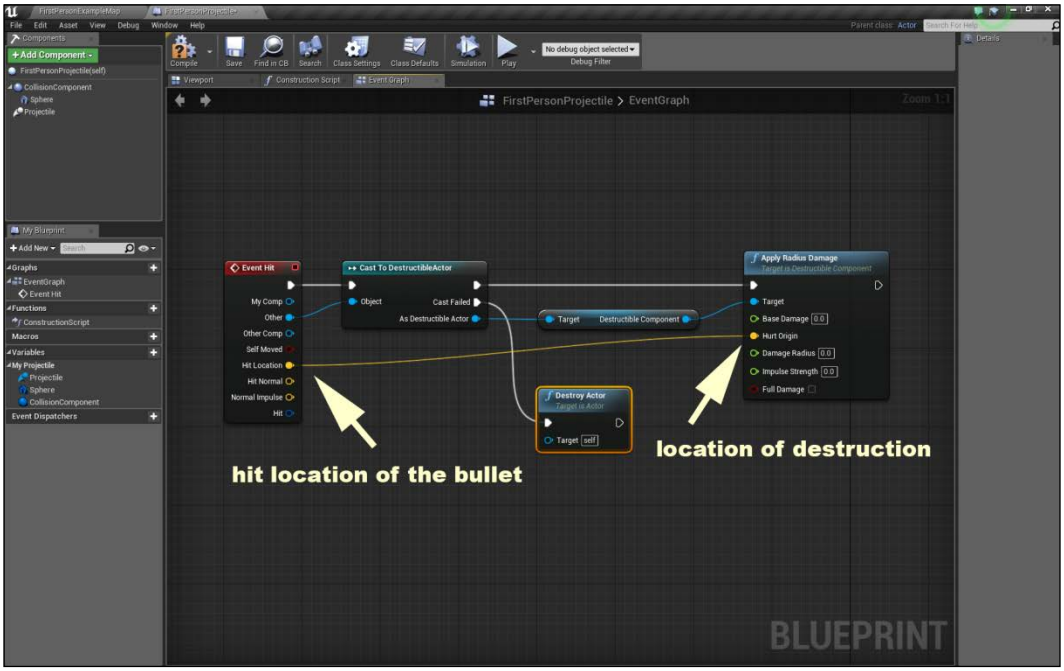
2. Then, start a new project from **Project** browser by selecting the **New Project** tab. Now, select **First Person** and make sure that **With Starter Content** is selected and name the project `dest_test`.
3. Now, open the **FirstPersonBP** folder and then in the **Blueprints** folder, double-click on **FirstPersonProjectile**. This opens the blueprint of your bullet. Check the titles of the box and the relations between each of them. This box simulates the default behavior of the bullets: hit the target and disappear and hit the wall and return at the opposite angle. These two have branches from the **Other Comp** output on the event box.



4. Disconnect **Event Hit** by pressing *Alt* + right-click on the outputs. Move it to an empty area on the blueprint screen and connect **Cast To Destructible Actor** to it. This means that the bullet will interact with the destroyable object. Click on the **Cast Failed** output and connect **Destroy Actor** to it. This means that if the bullet doesn't interact with any objects, it gets destroyed after a while.
5. Click on the **As Destructible Actor** output and connect the **Get Destructible** component to it and then connect the **Destructible Component** output of this box to **Apply Radius Damage**. By changing the properties of this box, you can control the physical simulation of the destruction of an object.

To create a new box in **Blueprint**, it's better to click and hold the mouse on the output of the box and then drag the wire, leave the mouse, and in the opening menu, enter the new box name.

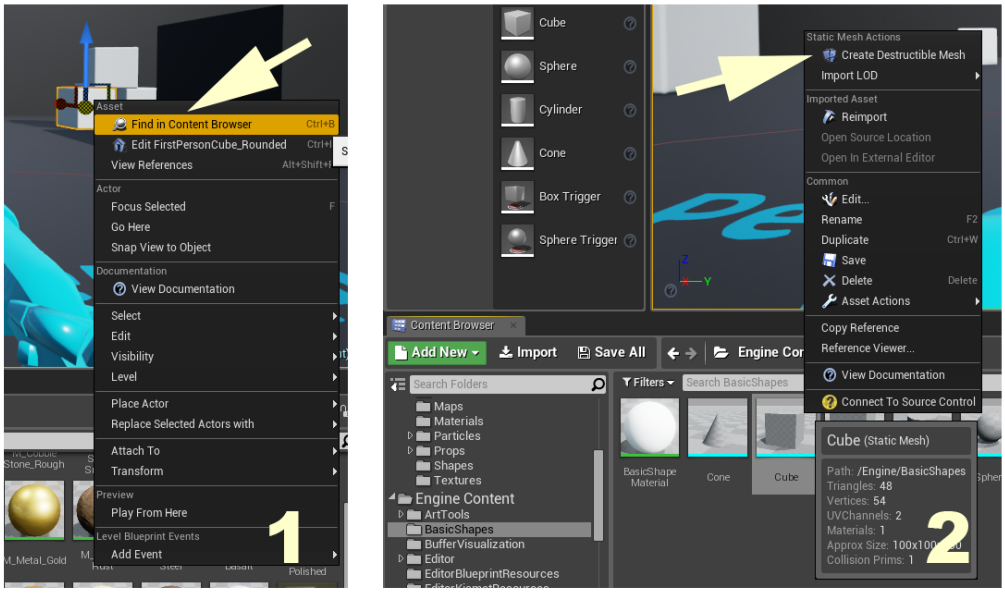
6. Now, add more connections between your boxes, as shown in the following screenshot:



As you can see in the preceding screenshot, the **Hit Location** output and the **Hit Origin** input are directly connected to each other. You can include more blueprint boxes between the **Hit Location** output and the **Hit Origin** input of your blueprint to support different scenarios. For example, imagine you want the player to shoot at an object in the sky and destroy another object located on the surface. For this scenario, there are a couple of ways to create a blueprint. This is not part of the physical rules and is mostly related to the controls and event handling in the blueprint. In our example, they are located at the same point, so the object gets destroyed by the bullet at the hit point.

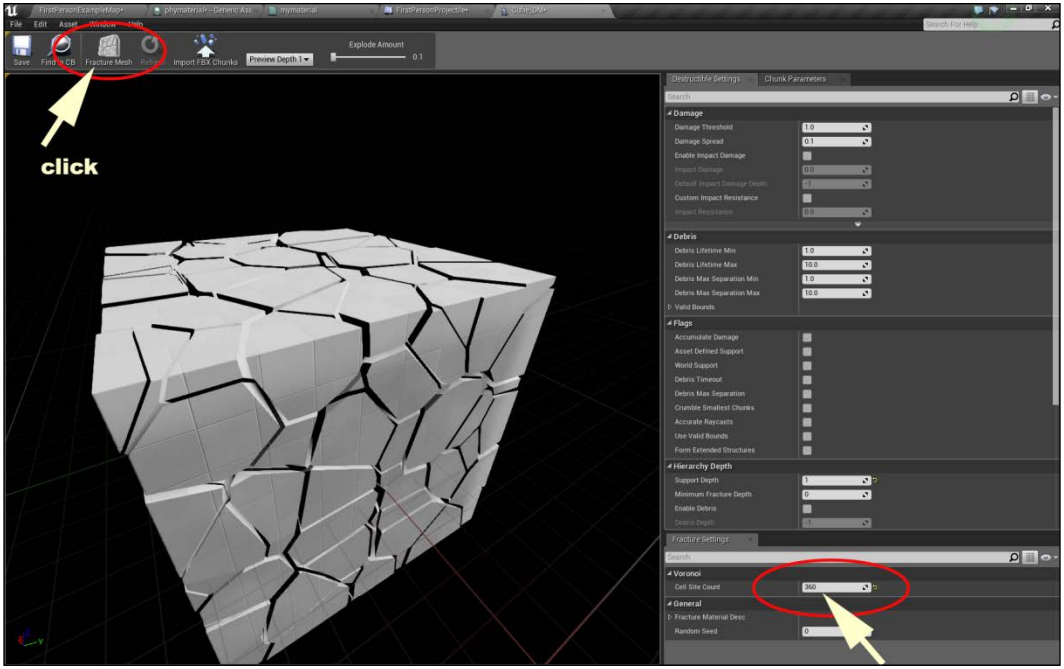
Now, let's create and customize the target by performing the following steps:

1. Go back to the editor and create a simple material with just the base color and one physical material in your material library. Then, drag and drop a cube from **Basic** in the **Modes** panel, right-click on it, and select **Find** in **Content Browser**. This will find the original object in the engine local library. Now, right-click on it in **Content Browser**. From the menu, select **Create Destructible Mesh**. This will create a new cube object with the properties to simulate destruction.

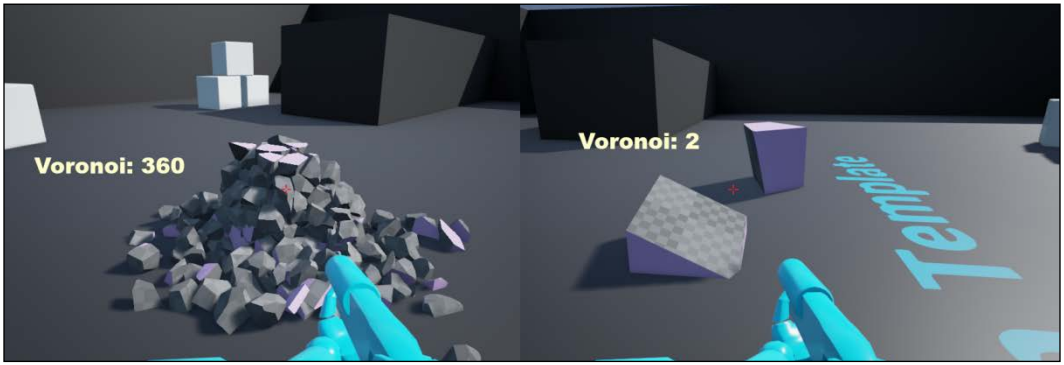


2. Drag an instance of this new cube onto the stage and apply your materials to this new object. Hit **Play** at the top and shoot at the cube. You will find that your bullets get projected from the surface of the cube. Now, click on **Stop** and navigate back to editor. Open the blueprint for your **First Person Projectile**, locate the **Apply Radius Damage** box, and change **Base Damage** to `1000`, as shown in the following screenshot:

3. In editor, double-click on the **Destructible Component** section in the **Details** panel of your cube. A new window will open. Here, you can define the properties related to destruction. Change **Support Depth** to `1` and then **Voronoi** to `360` (if your system is not very powerful, change it to `120`). Now, click on **Fracture Mesh** located at the top. With this action, you will be able to apply your setting to the mesh; otherwise, it will not work.

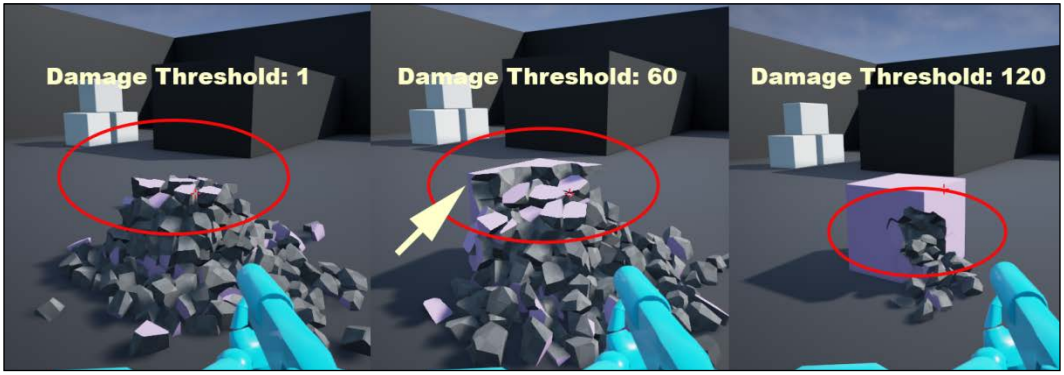


4. Again, let's navigate back to editor, hit **Play**, and shoot just one bullet at the object. As you can see, the cube breaks into many small particles. Now, hit stop and double-click on **Destructible Component** again. This time, change **Voronoi** to `2`, click on **Fracture Mesh** at the top, and play the level again (shoot just one bullet at the object).



As you can see in the preceding image, there is a big difference between the number of particles and the changes on **Voronoi**. This way, you can define the number of pieces for the object during the destruction period. Unreal Engine 4 automatically calculates the shape of pieces and renders them to the stage in real time based on physical rules. You can also import other meshes and apply the same.

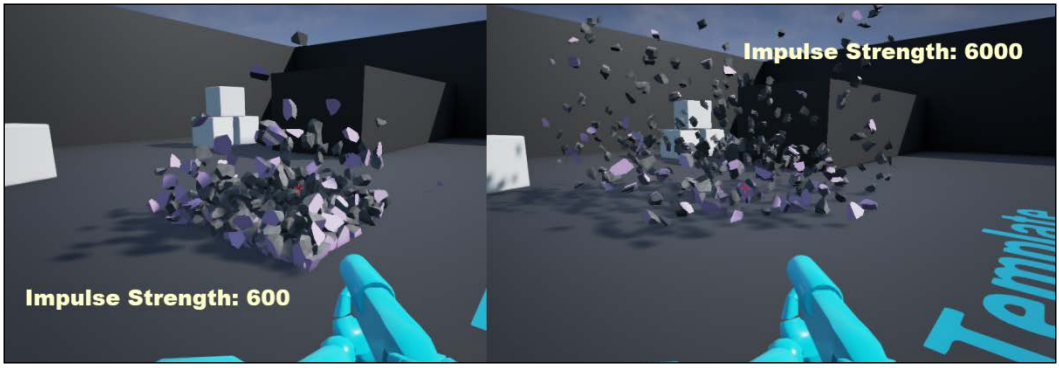
5. Change **Voronoi** back to `360` and then **Damage Threshold** to `40`. Click on **Fracture Mesh** and play the level (shoot just one bullet at the object).



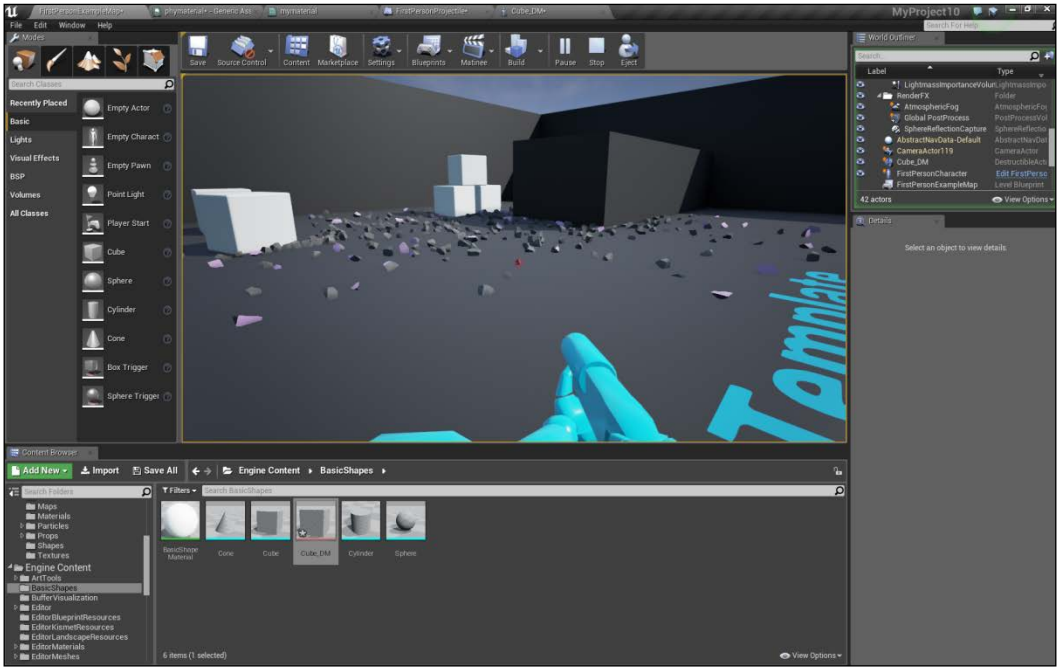
As you can see, the object provides some resistance against your bullet. If you increase **Damage Threshold** to `120`, you need to shoot more than one bullet to destroy the object and particles. This is similar to a rifle bullet destruction of a heavy material similar to stone.

6. Now, change **Damage Threshold** to `1` (the default value), click on **Fracture Mesh** and then switch back to the blueprint. We want to simulate the shotgun bullet's destructive power on the target. Change **Impulse Strength** to `600` and click on **Compile** at the top. Then, navigate back to editor and play the level by shooting one bullet at the object.

   As you can see in the following screenshot, it breaks with much more energy. Also, the visual simulation of physics is different. Now, navigate back to blueprint, change it to **6000**, click on **Compile** at the top, and play the game again. Bang! This looks similar to a heavy shotgun hit from close distance, isn't it?



7. You can also involve physical material features in the object. This means that with multiple physical rules, you can work at the same location. Now, double-click on your physical material and change **Friction** to `-60` and **Restitution** to `0`. Then, play the game. You can see in the following screenshot that the object breaks into particles similar to ice cubes and spreads around the game stage. Such a cool effect can be combined by changing **Restitution** to `1.8`.



There are many ways to combine the physical aspect of your object and the bullet preferences with the destruction properties of the object. For example, you can tune your bullet energy by changing **Impulse Strength** over a period of time. Also, as you know, each material can be customized by its own blueprint. For example, imagine that you have a ghost in your game; when you destroy it, it breaks into many particles and parts, and each part looks like rainwater. To create this scenario, try to apply some dynamic or random changes, such as waving, to your material and compare the results.

Sometimes, when you use a couple of physical customizations on an object, they tend to overlap with each other or remove each other's effect. To avoid such problems, it's good practice to understand the physical rules related to the instigator of the event (the bullet in the previous example) and then the physical rules of the target.

Summary

A simulation of mass change or a series of changes is a kind of art in the game world. A sufficient blueprint by developers can synchronize the physics behind the scenes. The presentation and quality of the simulation is based on the artist's efforts and madness.

Based on what you learned in this chapter, imagine that you (as a developer) try to put 50 objects on the stage. When you shoot at one, all get destroyed within a delay of a second. Also, you (as an artist) should design how and in which way the objects get destroyed and probably ask developers to make life easier with some blueprints. This great mix of developers and artists in the game design is one of the powerful aspects of Unreal Engine 4.

Index

Symbols

A

B

C

E

S

T

U

V

W