Community Experience Distilled

Learning Unity Physics

Learn to implement Physics in interactive development using the powerful components of Unity3D

K. Aava Rani

[PACKT] PUBLISHING

Learning Unity Physics

Table of Contents

Learning Unity Physics

Learning Unity Physics

First published: October 2014

Production reference: 1221014

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78355-369-3

www.packtpub.com

Credits

Author

K. Aava Rani

Reviewers

Subbu (L BalaSubbaiah)

Bryan Wai-ching CHUNG

Thuan Do The

Jacob Williams

Commissioning Editor

Akram Hussain

Acquisition Editor

Llewellyn Rozario

Content Development Editor

Prachi Bisht

Technical Editors

Pankaj Kadam

Nikhil Potdukhe

Copy Editors

Deepa Nambiar

Rashmi Sawant

Project Coordinator

Sageer Parkar

Proofreaders

Maria Gould

Ameesha Green

Paul Hindle

Indexers

Mariammal Chettiyar

Tejal Soni

Graphics

Disha Haria

Production Coordinators

Arvindkumar Gupta

Conidon Miranda

Cover Work

Conidon Miranda

About the Author

K. Aava Rani is a cofounder of CulpzLab Pvt Ltd—a software company with 10 years of experience in game technologies. She is a successful blogger and technologist. She switched her focus to game development in 2004. Since then, she has produced a number of game titles and has provided art and programming solutions to Unity developers across the globe. She is based in New Delhi, India. Aava Rani has been the recipient of several prestigious awards, including Game Technology Expert (2012) from Adobe and recognition from SmartFoxServer for her articles. She has experience in different technologies.

Aava has also reviewed the book, *Creating E-Learning Games with Unity*, *David Horachek*, *Packt Publishing*.

I would like to extend my appreciation to the people at Packt Publishing for their wholehearted support in producing my book.

My thanks go to Prachi Bisht, the content development editor, for making the revision process work like a well-oiled machine; Llewellyn Rosario, the acquisition editor, for his advice on shaping the book. I want to thank the board of advisors who contributed with their feedback throughout the process.

I specifically want to thank Anil Kumar Sah for providing support and guidance; Dr. Sunil Kumar Sah, a Computer Science lecturer at TM University, who taught me how to love technology; Shova Saha, who is a teacher and has a Master’s degree in Computer Applications, for supporting me throughout my book; Vinod Gupta, my husband, who supported me; and my newly born son’s love and twinkling eyes, which give me energy and dedication.

Finally, I would like to express my greatest gratitude to my father, RK Rajanjan, who has helped and supported me throughout my life to be a better person and achieve great heights in my career.

About the Reviewers

Subbu has 5 years of experience in Unity3D. He has developed Physics games, puzzle games, challenge games, fighting games, and so on. His games can be found on iOS, Android, Facebook, and the Web.

He has been working at Credencys Solutions Inc. since September 2013 as a game tech lead. He is very happy to work there since all his team members are very good supporters and the work environment too is awesome.

Subbu has also written blogs on Unity3D tutorials.

I would like to thank Nihal Rama for her support, help, and valuable suggestions while I was reviewing this book.

Bryan Wai-ching CHUNG is an interactive media artist and design consultant. His artworks have been exhibited at World Wide Video Festival, Multimedia Art Asia Pacific, Stuttgarter Filmwinter festival, Microwave International New Media Arts Festival, and China Media Art Festival. In World Expo 2010 Shanghai, he provided interactive design consultancy to various industry leaders in Hong Kong and China. Chung studied Computer Science in Hong Kong, Interactive Multimedia in London, and Software Art in Melbourne. He also develops software libraries for the popular open source programming language, Processing. He is the author of the book, *Multimedia Programming with Pure Data, Packt Publishing*. Currently, he is an assistant professor at the Academy of Visual Arts, Hong Kong Baptist University, where he teaches subjects on Interactive Arts, Computer Graphics, and Multimedia. His website is http://www.magicandlove.com.

Thuan Do The is a passionate self-taught senior Flash developer who fell in love with programming in the age of MS-DOS and Turbo Pascal. He inadvertently discovered the beauty and power of Flash, started to learn 2D animation and ActionScript, and then got his first professional job in 2007.

During his career, he has built numerous interactive websites, media players, galleries, and rich Internet applications and started several open source projects on http://github.com/thienhaflash.

Thuan started to play with Unity when Unity 2.6 was released and never looked back. Being attracted by the power of the Unity platform, he stopped working on Flash to completely focus on Unity games. Right now, besides being a Unity team leader at his full-time job, Thuan is also an Asset Store author with some best-selling editor extensions such as Hierarchy2, Inspector2, and NGUI Depth.

I would like to give a special thanks to my wife, Toky, for supporting me so much in all my life.

Jacob Williams is a freelance software developer who has specialized in game development. He has been a Unity developer for over 6 years and has been a very active part of the independent game development community. He lives with his wife and two kids in rural Alabama and can usually be found in the middle of the night in front of his

computer, working on personal projects and tech demos. You can read more about Jacob at his personal blog, cheapdevotion.com.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Preface

In order to understand what it means to implement a Physics component of Unity3D successfully in games and applications, a developer must have a better understanding of the key elements of Physics and their behavior in the engine. I wrote this book to provide a detailed description with examples for learning a Physics component of Unity3D in a way that emphasizes the uniqueness of each component and their implementation.

The successful implementation of Physics in game or application development starts with a strong understanding of each component. When we look at game and application development in Unity3D, it is easy to see that Physics is the most important component. I have provided the conceptual foundation of a Physics component and introduced these using examples for a better understanding.

I had three main goals while writing this book:

- **Accuracy**: This book is the result of years of experience and research. This book focuses on the Physics components of Unity3D and their implementations.
- **Implementation**: My experience as a developer has taught me two things about examples: they need to be detailed, and they must help developers do their work. As a result, users of my book will always find abundant examples with every step carefully laid out and explained wherever necessary so that the reader can follow. To describe and explain the Physics of Unity3D, this book uses realistic examples to help developers get inside of what Physics is really like. In addition to the examples, I have provided detailed screenshots and figures for a better understanding. These examples will help developers in the successful implementation of Physics components in their game or application as well as help them identify the best practices they need to adopt to improve their game or application performance.
- **A structured approach**: I have tried to explain Physics and Unity3D in a reader-friendly way. Most books claim to do this, but my experience with a variety of books has proven otherwise. What students and developers will find in my book are short, precise explanations of terms and concepts that are written in an understandable language. For example, I have used more images rather than text to explain steps of implementations, which gives a better understanding for readers.

What this book covers

Chapter 1, *Introduction to Physics in Unity3D*, serves as a quick introduction to Physics, and specifically Physics in Unity3D. We will also learn about in-built Physics, as well as Unity3D and the uses of Physics in an interactive development.

Chapter 2, *Using Different Colliders for Interaction*, focuses on colliders, their types in Unity3D, and how we can define the collision shape of objects in a scene.

Chapter 3, *Overview of Collision Matrix*, explains matrices, their types, and how we can define a collision shape of objects in a scene explaining a matrix.

Chapter 4, *Rigidbody Types and Their Properties*, focuses on Rigidbodies and their types in Unity3D.

Chapter 5, *Joint Types and Their Properties*, describes joints and their types in Unity 3D.

Chapter 6, *Animation and Unity3D Physics*, gives in-depth knowledge on how to use Physics in animation in Unity3D. We will be developing complex animations and also see some examples of animation using Physics.

Chapter 7, *Optimizing Application's Performance Using Physics in Unity3D*, teaches you how you can optimize applications and games if you use Physics in Unity3D.

What you need for this book

The code examples in this book should work on most modern operating systems. To install Unity3D, go to https://unity3d.com/unity/download. For the system requirements, check out https://unity3d.com/unity/system-requirements.

The following is a list of software used for the examples and versions used for testing:

- Unity3D 4.1 and higher
- MonoDevelop (this normally comes with Unity3D software)

Who this book is for

If you are familiar with the fundamentals of Physics and have some basic experience of Unity game development, but have no knowledge of using these two together, this book is for you. Educators and trainers who want to use Unity in an e-learning setting will also benefit from the book. For further reading, the following books are recommended:

- *Unity 4.x Game Development by Example: Beginner's Guide*, *Ryan Henson Creighton, Packt Publishing*
- *Unity 4 Fundamentals: Get Started at Making Games with Unity, Alan Thorn, Focal Press*
- *Unity 4.x Game AI Programming, Aung Sithu Kyaw, Clifford Peters, and Thet Naing Swe, Packt Publishing*

Conventions

In this book, you will find a number of styles of text 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: “When a collision occurs, a trigger will call the `OnTriggerEnter` function on the trigger object’s scripts.”

A block of code is set as follows:

```
function OnTriggerStay(col:Collider)
{
  isInRang = true;
}
function OnTriggerExit(col:Collider)
{
   isInRang = false;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: “Select the **Cube** option by navigating to **GameObject** | **Create Other**.”

Note

Warnings or important notes appear in a box like this.

Tip

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via 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 on 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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright 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

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Introduction to Physics in Unity3D

Physics has been used for a long time in different sectors such as scientific, study, and software. The use of Physics in interactive development and software is not new, but its use has been the focus of special attention in recent years.

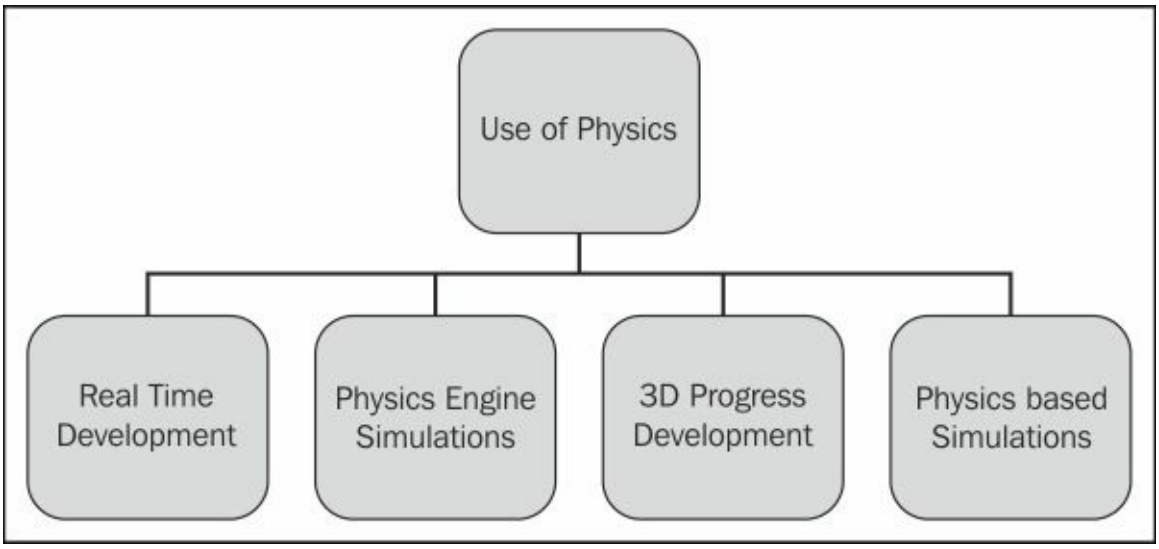
At first glance, Physics gives the impression that it’s all about long, webbed equations and calculations. However, the contributions of Physics are tremendous and have made developer’s life easy. The idea of gaming was possible only because of Physics. All of us remember our childhood games, which were facilitated by the use of Physics; for example, the Ping Pong game. Isn’t it amazing how much Physics has touched our lives and how we cannot escape its influence? There are many games that possess primitive Physics, used to calculate the movement and trajectory of a bouncing ball. If you look into the structure of Physics used for one of the earliest games, Pong, you get a brief insight into how Physics was used in these games. Game behavior, motion trajectory, and paddle movement were all handled by primitive Physics.

In this chapter, we will learn about the following topics:

- How Physics is used in interactive development
- The basic components of Physics for interactive development
- Physical simulation in Unity

In today’s interactive world, the use of Physics is changing day by day. In recent years, the use of Physics in games, software, and interactive development has increased drastically. Now, we are able to make games and other applications using Physics in different ways. In the modern era, the importance of in-game Physics and interactive development has increased.

If you look at the way Physics was used in older interactive applications or games, you will find that they were all based on some specific scenarios. The Physics code was written according to the scenario of interactive development. For example, if the Physics code is written for a ball to handle its trajectory motion, it only handles the trajectory motion and nothing else; this means that every time a developer has to write a new code for a new effect. Also, with the passage of time, if an update is required, the developer has to modify or even rewrite the code. Based on such cases, development became very complicated and time-consuming. Conditions became worse when the number of scenarios increased. As mentioned earlier, the way Physics is used is changing day by day, and it is required in various sectors. Let’s take a look at the areas where Physics is widely used these days:



Physics is useful in the following areas of interactive development:

- Real-time programming where we need knowledge of electronics, which is much related to Physics
- A Physics engine for graphics software or games, where Physics is heavily used
- Creating 3D programs for various software and game engines
- Physics-based simulations

The most common component of Physics used in interactive development

Rigidbody dynamics is the most common component used in interactive development. For Physics-based simulation development, we use algorithms of Rigidbody dynamics. Rigidbody dynamics is based on the Newtonian principle of movement and mass.

Now, the question that arises here is what is a Rigidbody? An idealized solid whose size and shape is fixed and remains unaltered when some external forces are applied and is used in Newtonian mechanics to model real objects is known as a Rigidbody. For example, a box, wall, and so on.

The use of accurate Physics in interactive development is not feasible as it has its own natural constraints. In interactive development, we cannot use accurate Physics due to the standard frame rate restrictions, but the physical accuracy of a simulation only needs to be believable.

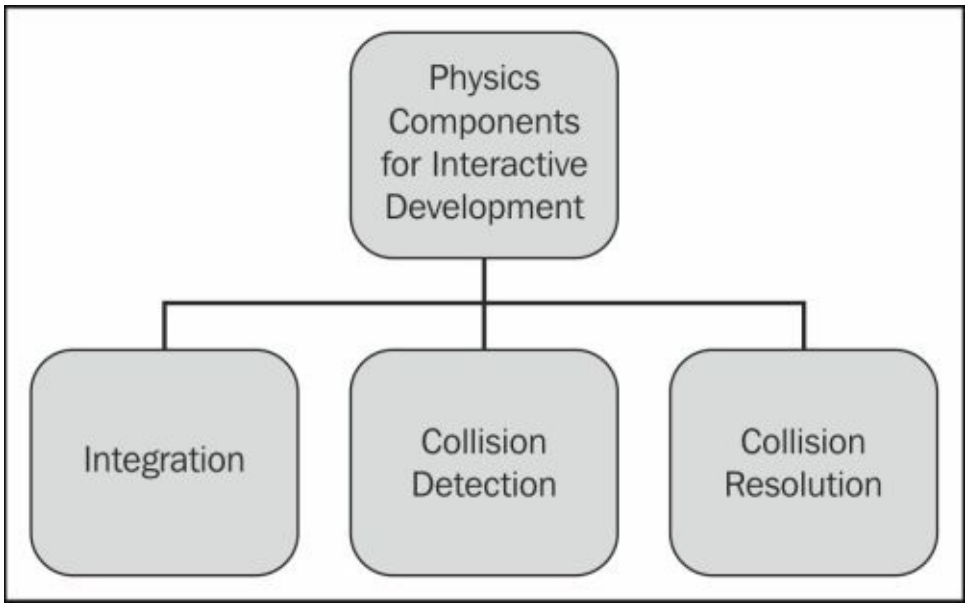
Use of Physics in simulation and frame rate

In order to use Physics for interactive development, there are many things that need to be taken care of. Physics simulations are related to the frame rate. The frame rate can be described as the number of frames that are displayed per second. This is relevant to animation, in which many images are displayed quickly to give the impression of movement. The standard frame rate for most PCs and console games is 60 fps.

The interval of moving objects with their trajectories is called Physics simulation. Almost every Physics simulation equation involves time, and the time required for solving this is determined by interactive development.

Basic components of Physics for interactive development

There are a few basic components that can be used for interactive development. Let’s take a look at them one by one:



Integration

There are multiple factors that have to be considered when we are talking about the integration of Physics. How we track objects that participate in simulation is of paramount importance in this regard.

To implement Physics in interactive development, we normally track all the objects that are simulating in a data structure.

To implement Physics for each object, we need to know some important information such as the object’s physical properties, that is, mass, current velocity, current position, orientation, the external forces acting on the object, and the future time of an object.

Note

What is future time?

Future time = the current time + the time slice for the frame

Collision detection

There is no collision detection if only one object is moving in a vacuum. However, most interactive developments involve more than one object and these objects move in an environment. Therefore, a situation arises where two objects are moving towards each other.

What will happen if no action is taken? These objects will just pass through each other. In most interactive developments, however, we don’t want the objects to pass through each other.

In order to handle those scenarios where collision occurs, the interactive development needs to know that two objects are colliding. One of Physics’ most important tasks is to identify these scenarios.

The collision detection code has to determine all such pairs of overlapping objects, collect some additional data such as how far they overlap and in which orientation, and provide this data to the interactive development for further processing.

In the later chapters, we will see how we can handle collision detection in Unity3D.

Collision resolution

What will happen after the collision? Let’s take a look at one such scenario where two or more objects overlap and see what can we do in this case. In many cases, some specific rules are added to the interactive development.

For example, in a shooting game, when a bullet hits the player’s ship, the game might decide to show an explosion animation. Following this, before removing the player’s ship, the game might start the level again and reduce the number of lives of the player. These effects of the collision are driven by the game itself and not by Physics because they are very specific to the game.

However, there are certain cases where the game doesn’t have to be involved.

We’ve covered the basics of all Physics components. Every simulation suitable for interactive development will have these components. Other than the covered topics, there are many more features to be included, such as joints, cloth simulations, Physics-based animations, and so on.

Physical simulation in Unity

We will now look at the physical simulations already available in Physics and how Unity uses them. This section will also give you an overview of the built-in Physics components in Unity3D. Let’s take a look at the Physics simulation in Unity:

- Unity is a powerful tool. It is able to take care of many problems involved with interactive physical simulations. It embeds a state-of-the-art Physics engine called PhysX.
- It’s Rigidbody is mostly targeted at rigid objects.

Using Unity we can make Physics approximations based on an object’s parameter values.

Built-in Physics in Unity3D

As I mentioned earlier, Unity is a powerful engine that has a number of built-in Physics components. It handles physical simulations. By adjusting a few parameter settings, we can create an object that behaves in a realistic way.

By controlling Physics from scripts, we can give an object the dynamics of a vehicle, machine, cloth, and so on. The built-in components are very useful in fast development. In most interactive developments, these simulations are required. By making Physics a built-in component, Unity3D has made developers' lives easier.

Note

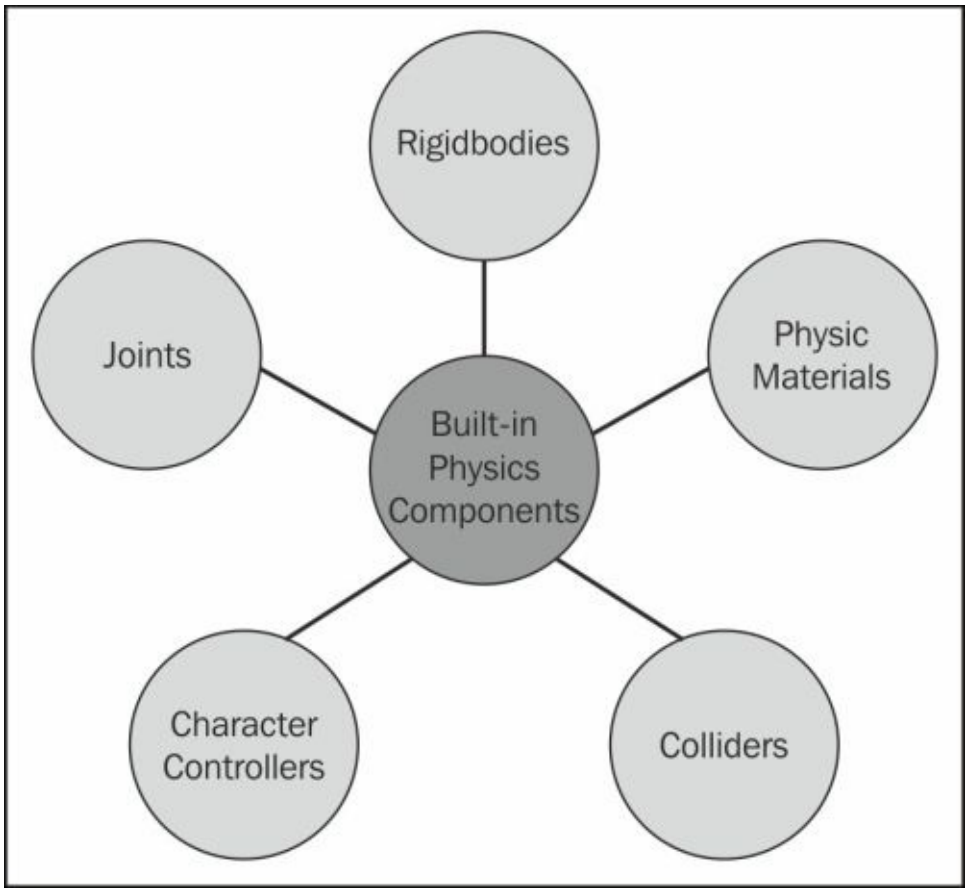
Unity has two separate Physics engines: one for 3D Physics and one for 2D Physics.

As such, there is a separate Rigidbody component for 3D Physics and an analogous Rigidbody2D component for 2D Physics.

Now, let's explore the built-in components in Unity3D.

Built-in Physics components in Unity3D

The following figure depicts the basic built-in Physics components in Unity3D, which help us in interactive development:



Rigidbodies

We have already discussed that in simulation or interactive development, the most important component is a Rigidbody. It enables the physical behavior of an object. The object to which a Rigidbody is attached can be made to respond to gravity. If we want to create a ball and want it to respond to gravity, we need to add a Rigidbody component to the object and gravity will be enabled by default.

A Rigidbody component takes the movement of the object to which it is attached; therefore, we shouldn't try to move it using a script by changing the position and rotation. Instead, we can apply forces to push the object and let the Physics engine calculate the results.

Kinematic motion and Rigidbodies

Sometimes, it is desirable for a Rigidbody object's motion to not be controlled by the Physics engine but by the script code instead. This type of motion produced from a script is known as kinematic motion.

Is Kinematic is one of the properties of a Rigidbody that will remove it from the control of the Physics engine and allow it to be moved using a script. We can change the value of

Is Kinematic from a script to switch this property on and off for an object by using both the script and the inspector.

Note

Is Kinematic is useful, but we should keep in mind that enabling it will affect performance. If enabled, the object will not be driven by the Physics engine and can only be manipulated by its transform, which is more performance consuming.

Colliders

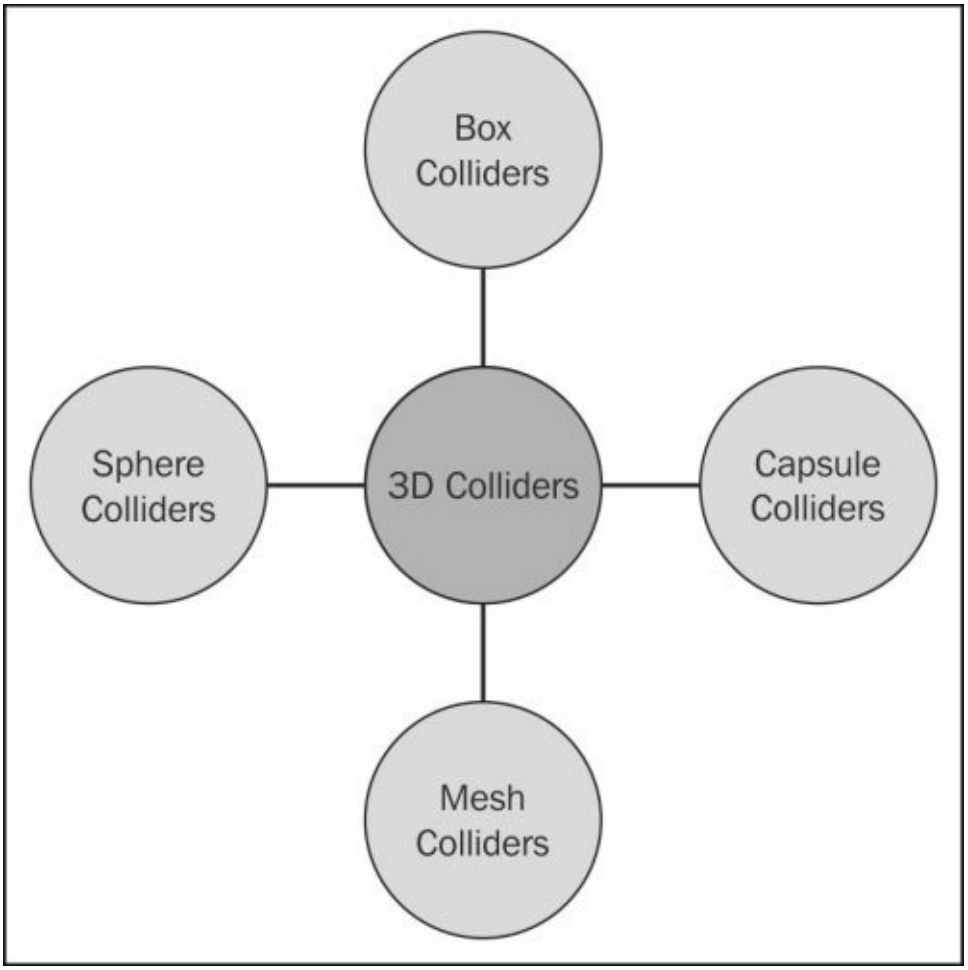
It’s one of the most important built-in components of Unity3D. A collider component is used to define the shape for the physical collision. We use different colliders according to the shape of the objects. A collider, which is invisible, need not be matched exactly to the shape of the object’s mesh.

In the later chapters, we will see how to use different colliders.

In 3D, the following are the basic colliders:

- Box Collider
- Sphere Collider
- Capsule Collider
- Mesh Collider

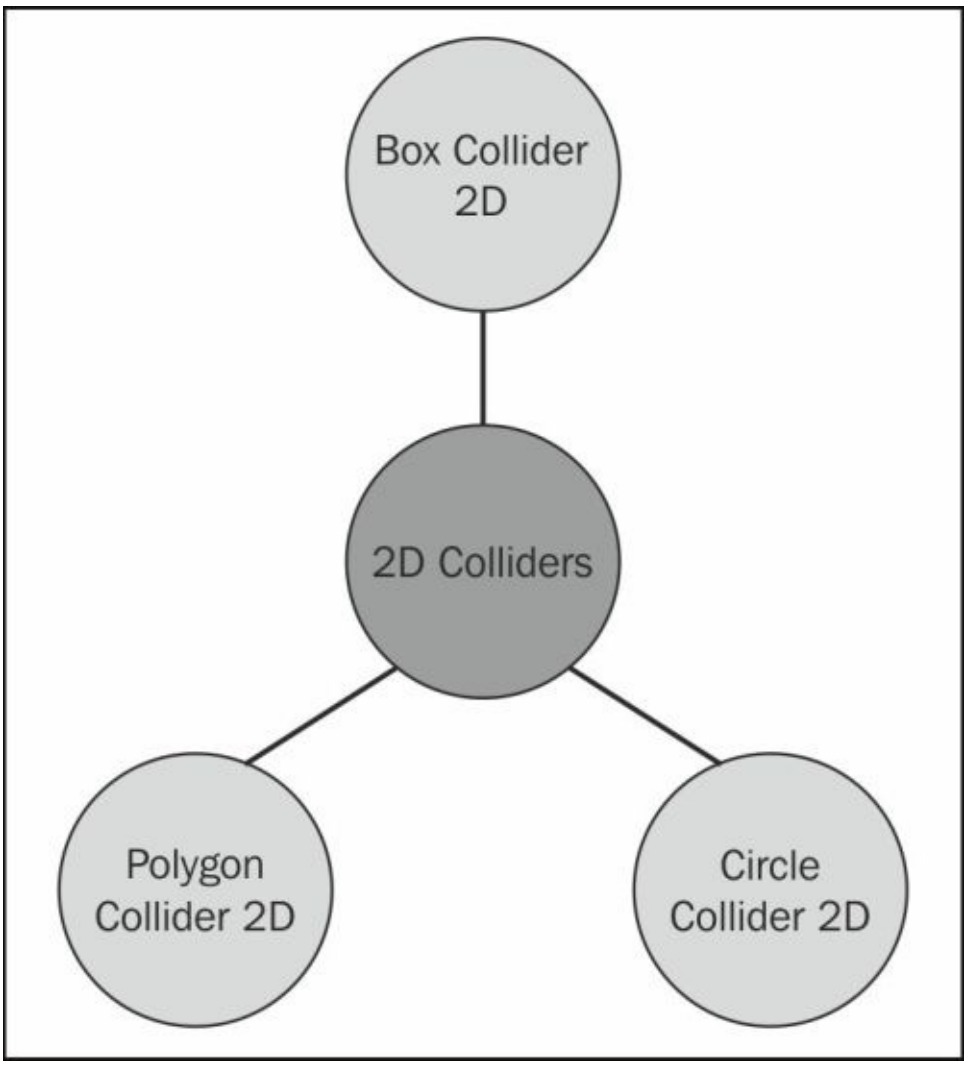
Take a look at the following figure:



There are differences between 3D colliders and 2D colliders. In 2D, the following are the basic colliders:

- Box Collider 2D
- Circle Collider 2D
- Polygon Collider 2D

Take a look at the following figure:



Apart from the aforementioned core colliders, there are a few important terms about the colliders that you need to know.

Static colliders

Colliders can be added to an object without a Rigidbody component in order to create floors, walls, and so on. These are referred to as static colliders. Repositioning static colliders by changing the transform position will impact the performance of the Physics engine heavily.

Note

To improve the performance, we should not reposition static colliders by changing the

transformation position.

Dynamic colliders

Colliders attached to a Rigidbody object are known as dynamic colliders. Static colliders do not respond to collisions with dynamic colliders with any movement. In the later chapters, we will learn about the aforementioned colliders and how to implement these colliders in detail.

Physic Materials

Different materials are used for different objects. As the colliders interact, their surfaces need to simulate the properties of the material that they are supposed to represent. We can configure the friction and bounce using Physic Materials.

Again, Physic Materials for 2D and 3D are different; they are called Physic Materials 3D and Physic Materials 2D.

Triggers

In scripting, we can detect when the collisions are going to occur and then we can initiate taking actions using the `OnCollisionEnter` function. We can configure a collider, which does not behave as a solid object, as a trigger using the `Is Trigger` property of Unity3D, and we will simply allow other colliders to pass through. When a collision occurs, a trigger will call the `OnTriggerEnter` function on the trigger object's scripts. Using these functions, we can handle a number of scenarios where an action after the collision is required.

Note

The following points will give you sources where you can find more details on this particular topic:

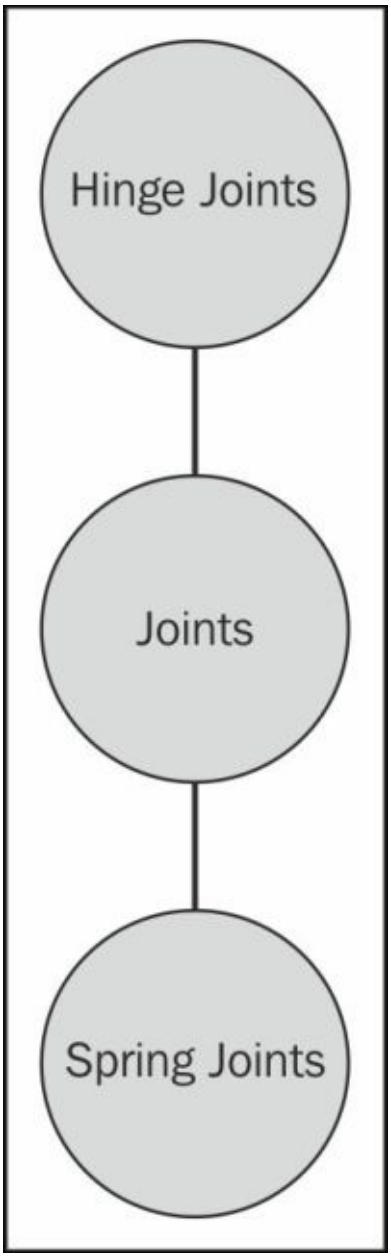
- `Collider.OnCollisionEnter(Collision)` at http://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html
- `Collider.OnCollisionStay(Collision)` at http://docs.unity3d.com/ScriptReference/Collider.OnCollisionStay.html
- `Collider.OnCollisionExit(Collision)` at http://docs.unity3d.com/ScriptReference/Collider.OnCollisionExit.html
- `Collider.OnTriggerEnter(Collider)` at http://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html
- `Collider.OnTriggerStay(Collider)` at http://docs.unity3d.com/ScriptReference/Collider.OnTriggerStay.html
- `Collider.OnTriggerExit(Collider)` at http://docs.unity3d.com/ScriptReference/Collider.OnTriggerExit.html

Joints

Often, one Rigidbody object is attached to another using joints. Unity provides different joints to help us in different scenarios. We can attach one Rigidbody object to another or to a fixed point in space using a joint component. If we want a joint to allow at least some

freedom of motion and so on, then Unity provides different joint components that enforce different restrictions.

The following figure depicts the types of joints:



Joints also have other options that enable specific effects; for example, we can set a joint to break when the force applied to it exceeds a decided limit.

We will learn more about joints in the later chapters.

Character controllers

In game development, often a character is required, and for this, a controller is always required. A character in a first- or third-person game will often need some collision-based Physics so that character doesn’t fall on the floor or walk over the walls.

Unity3D provides a component to create this behavior, which is called **CharacterController**. This component uses a Capsule Collider. The controller has

functions to set the object's speed and direction.

Note

You can find more details at http://docs.unity3d.com/ScriptReference/CharacterController.html.

Scripting based on collision

Let's see how we can handle collision and an after-collision effects with scripting. Scripts are attached to the objects in order to call some functions on collision. We can write any code in these functions to respond to the collision event. For example, we might play a sound effect when a ball hits an obstacle.

- `OnCollisionEnter`: This function indicates that the collision is detected in the first update
- `OnCollisionStay`: This function indicates that during the updates, the contact is maintained
- `OnCollisionExit`: This function indicates that the contact has been broken

Similarly, for Trigger Colliders, these functions are called `OnTriggerEnter`, `OnTriggerStay`, and `OnTriggerExit`.

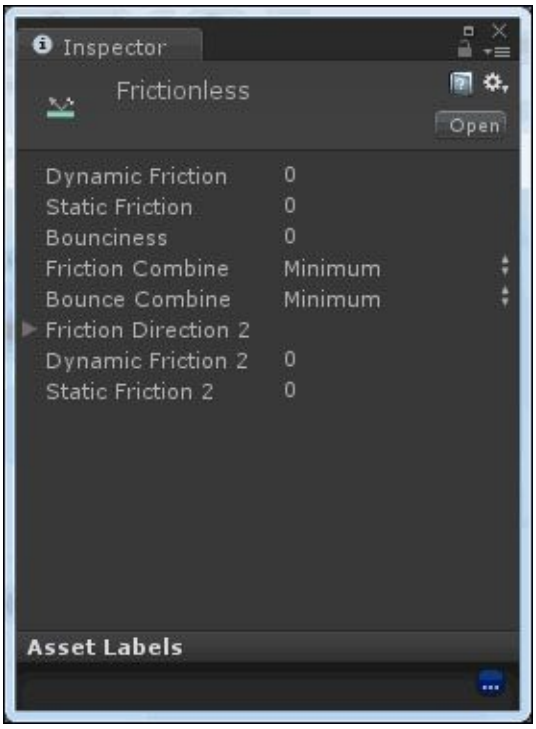
Note

For 2D Physics, there are equivalent functions with 2D appended to the name; for example, `OnCollisionEnter2D`.

With normal nontrigger collisions, there is an additional requirement that at least one of the objects involved must have a nonkinematic Rigidbody. To create a nonkinematic Rigidbody, we must set Is Kinematic to `off`.

Frictionless Physic Materials

Sometimes, during the development, we require frictionless Physic Materials. Let's try to create a frictionless material. In Unity3D, to create a completely frictionless material, we need to create a new Physic Materials asset in the **Project** view and set its properties as follows:



In the previous image, we can see various parameters, using which we can fulfil our frictionless game requirement.

Summary

This chapter is a quick introduction to Physics in Unity3D. In this chapter, we learned the use of Physics in an interactive world, built-in components of Unity3D, and frictionless material. In the next chapter, we will learn about colliders and their types, with examples. The chapter will also teach us how we can use different colliders for interaction.

Chapter 2. Using Different Colliders for Interaction

As mentioned in the previous chapter, colliders are among one of the main components of Unity3D. Using colliders, we define a shape for the object that helps in collision detection. In this chapter, we will learn about colliders. We will learn about the following topics:

- Primitive colliders and their implementation
- Nonprimitive colliders and their implementations
- Uses of Trigger Colliders
- Editing Polygon Collider 2D

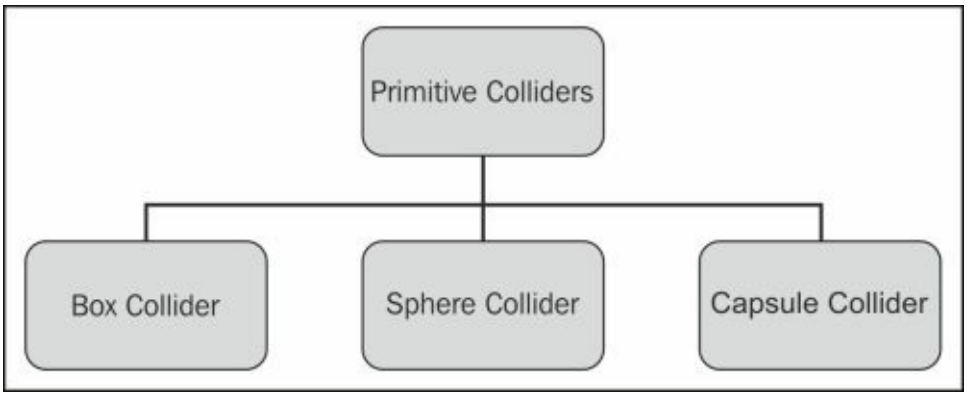
Let’s have a look at the primitive and nonprimitive colliders in detail.

Primitive colliders

Primitive colliders are basic colliders or, in other words, we can say that these colliders are the earliest of their kind. There are three primitive colliders in Unity3D, and apart from these three colliders, Unity also provides Mesh Collider, which helps us when we need to provide a collision shape for a complex shape.

Types of primitive colliders

As shown in the following figure, there are three primitive colliders: Box Collider, Sphere Collider, and Capsule Collider.



We will learn the uses and types of primitive colliders in detail with examples.

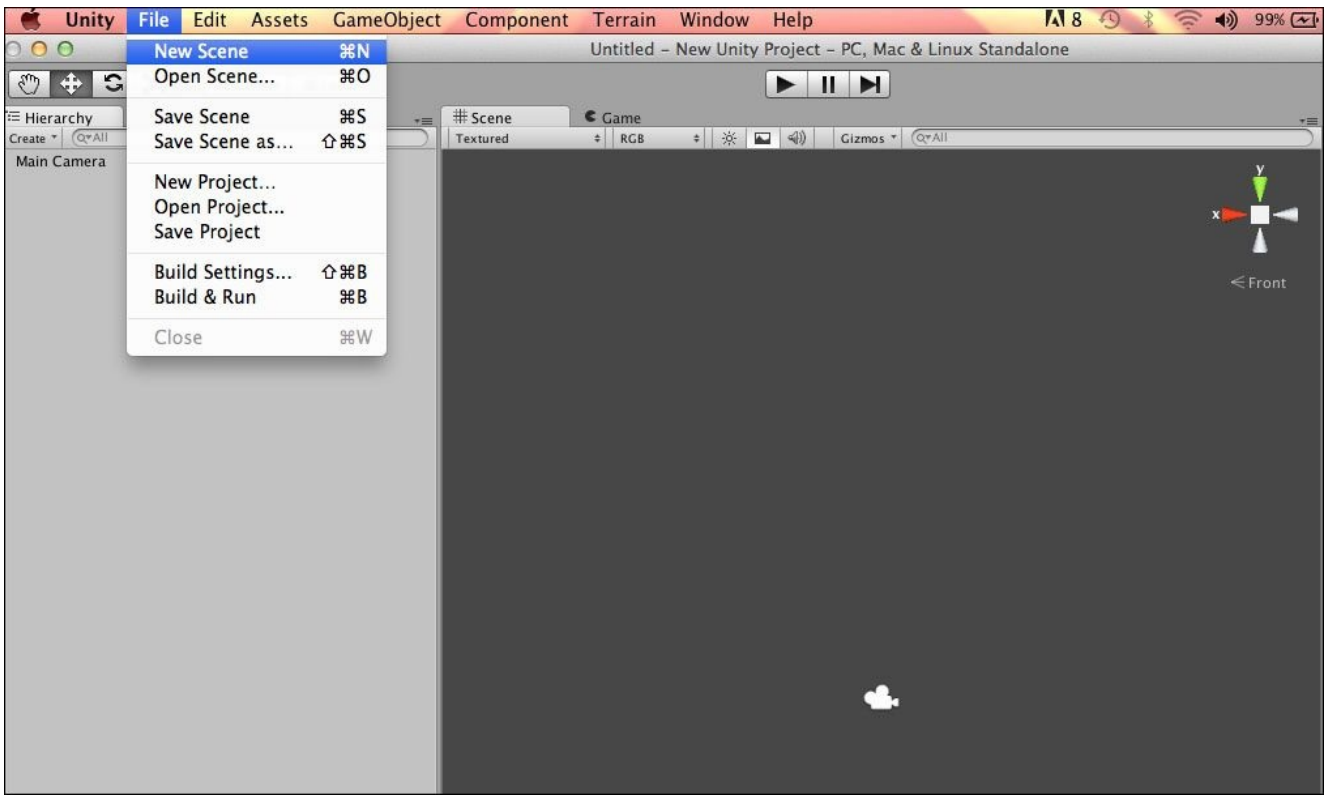
Box Collider 3D

Box Collider contains the cube shape and can be implemented for cube-shaped game objects, such as boxes, walls, and doors, that resemble the shape of a cube.

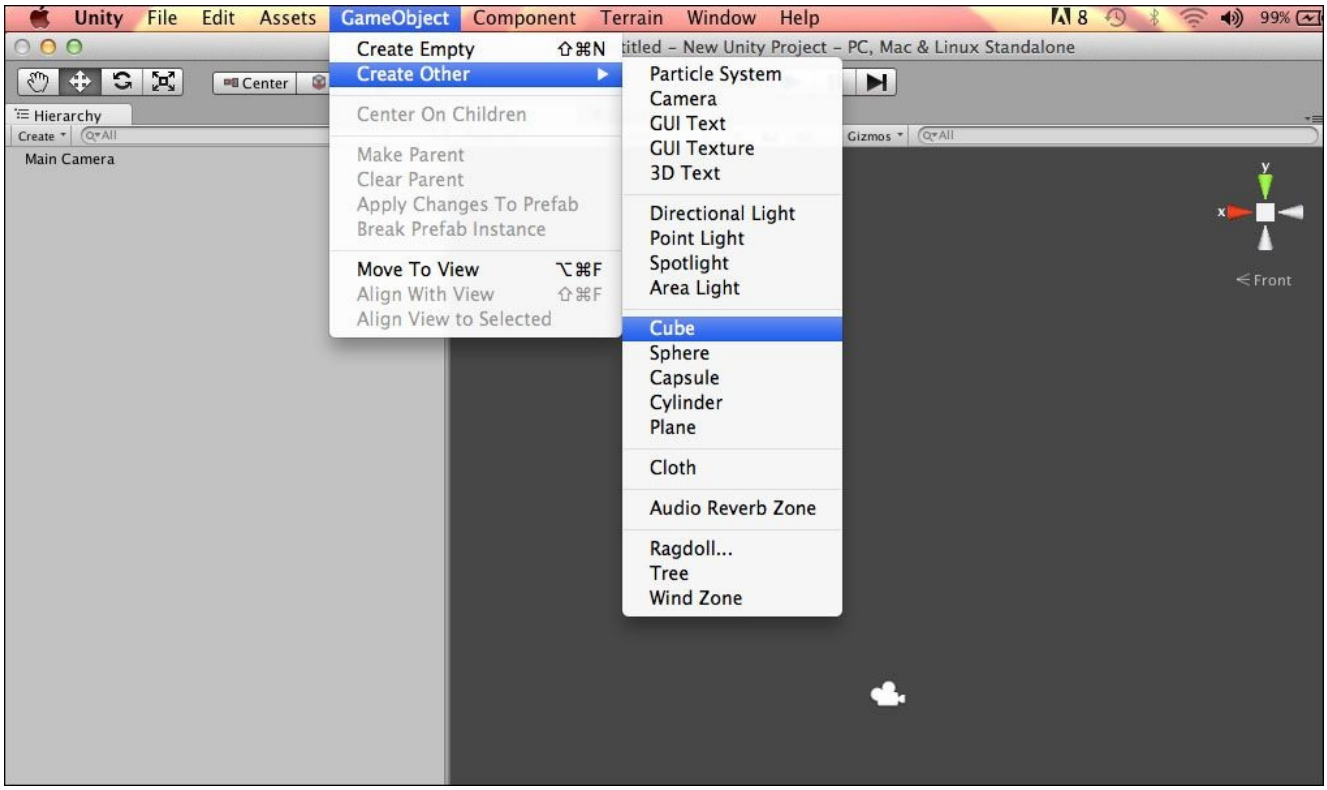
Example – implementation of Box Collider

Here, we will see how we can implement Box Collider in the following scene by performing the following steps:

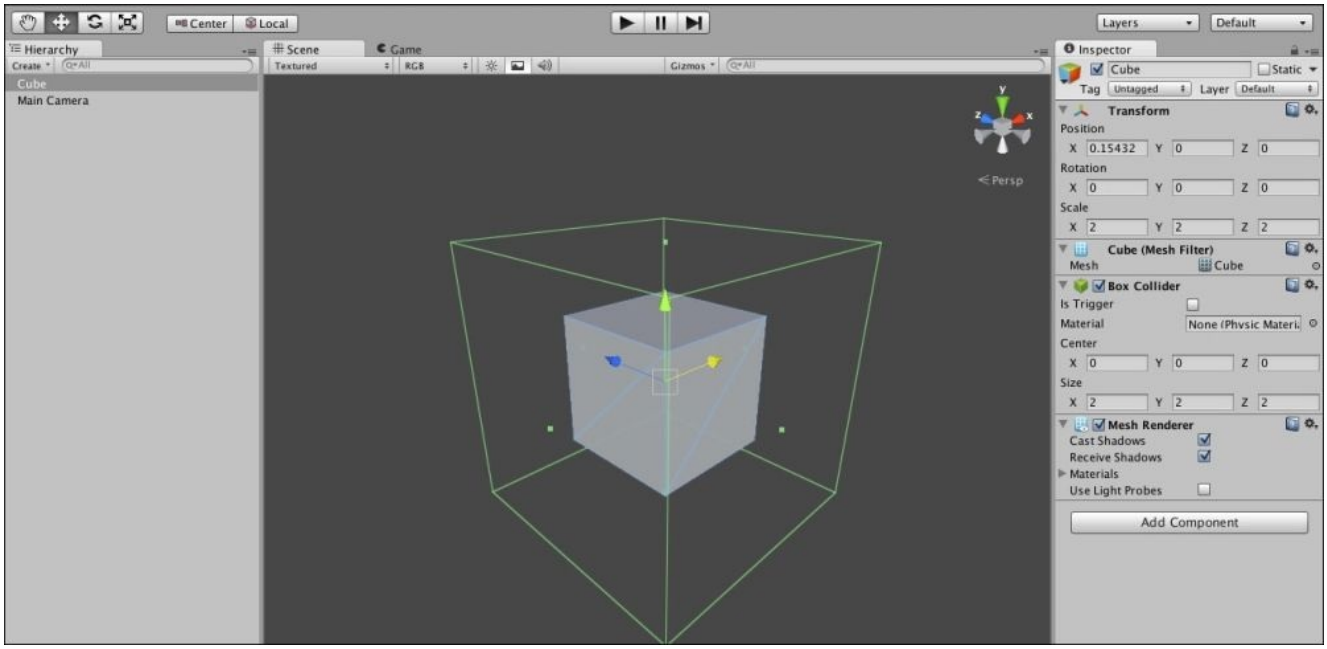
1. Let’s create a new scene as shown in the following screenshot:



2. Select the **Cube** option by navigating to **GameObject** | **Create Other** as shown in the following screenshot, and then open the **Inspector** panel:



3. In the following screenshot, you will see the **Box Collider** checkbox inside the **Inspector** panel; make sure it is checked:



4. In the **Inspector** panel, select the **Is Trigger** checkbox to make it trigger the collider and fire trigger events.
5. We can decide its shape and size using the dimension here; it should be relative to the

game object transform and size.

Note

Material

Box Collider has the Material property that determines the friction and bounciness of the game object, but it is irrelevant if we choose the Trigger Collider.

In the previous example, we learned how we can implement Box Collider for cube-shaped objects.

Box Collider 2D

For a 2D game object, Unity provides Box Collider 2D to handle the collision. This collider is of the rectangle shape and can be implemented on a sprite with given dimensions and coordinates. Like the 3D Box Collider, this collider too possesses the Is Trigger and Material properties along with the Size and Center properties.

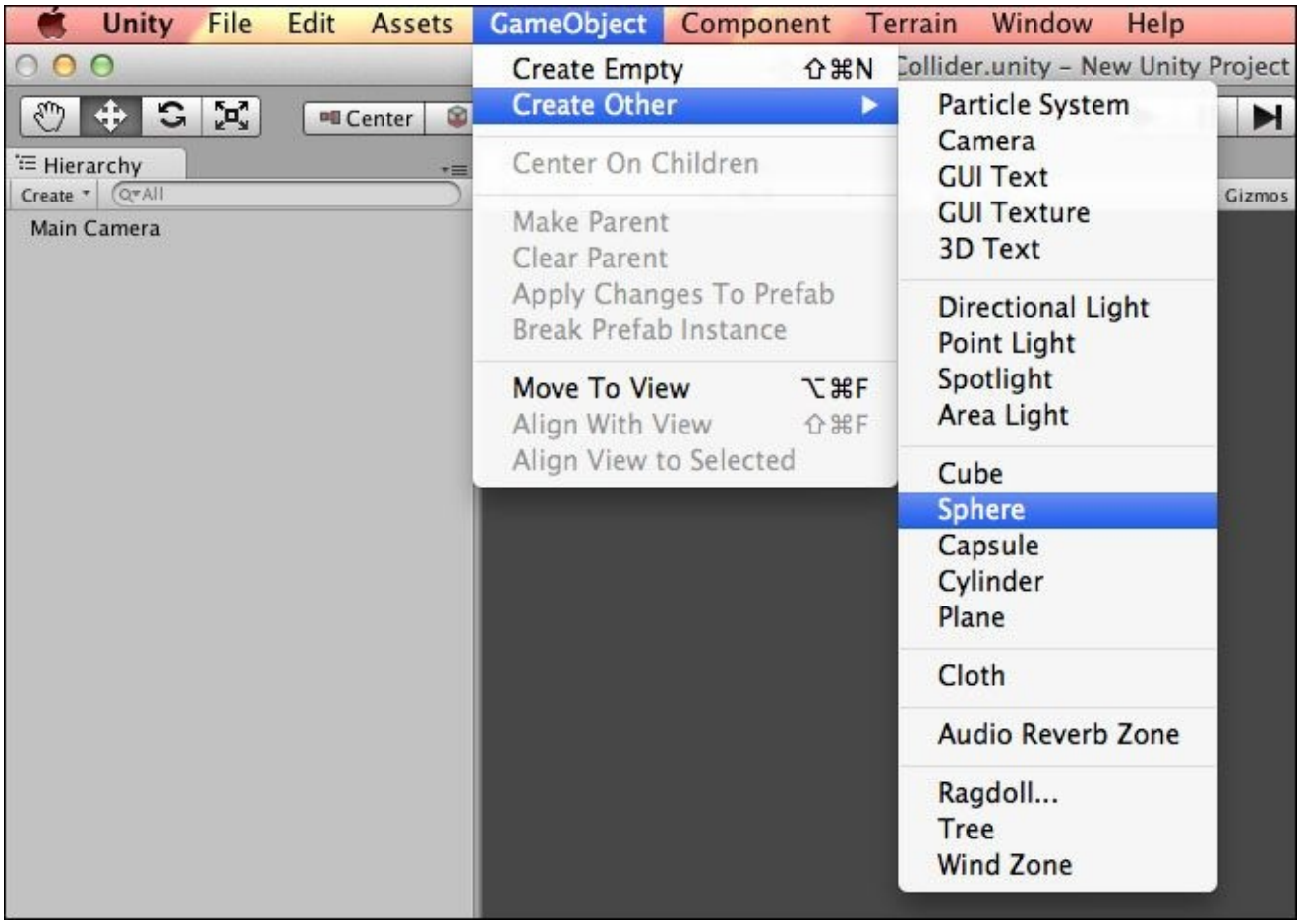
Sphere Collider 3D

For game objects that have the shape of a sphere, we use Unity3D's Sphere Collider.

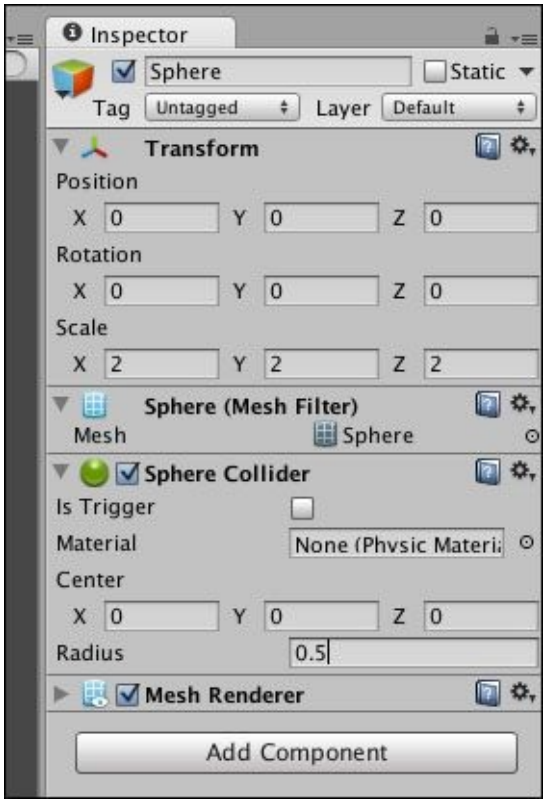
Example – implementation of Sphere Collider

Here, we will see how we can implement Sphere Collider in the scene using the following steps:

1. Create a new scene.
2. As shown in the following screenshot, go to **GameObject** and select **Create Other** where you will get a drop-down list; now, select a **Sphere** game object and open the **Inspector** panel:



3. Inside the **Inspector** panel, you will see **Sphere Collider**; make sure **Sphere Collider** is checked:



4. In the **Inspector** panel, check **Is Trigger** to make it trigger a collider and fire trigger events.
5. We can decide its shape and size using the dimension and radius; here, it should be relative to the game object transform and size.

Circle Collider 2D

For 2D objects, we implement Circle Collider instead of Sphere Collider for circle-shaped game object with a given dimension and coordinate. It has all properties which are in Sphere Collider such as Is Trigger and Radius.

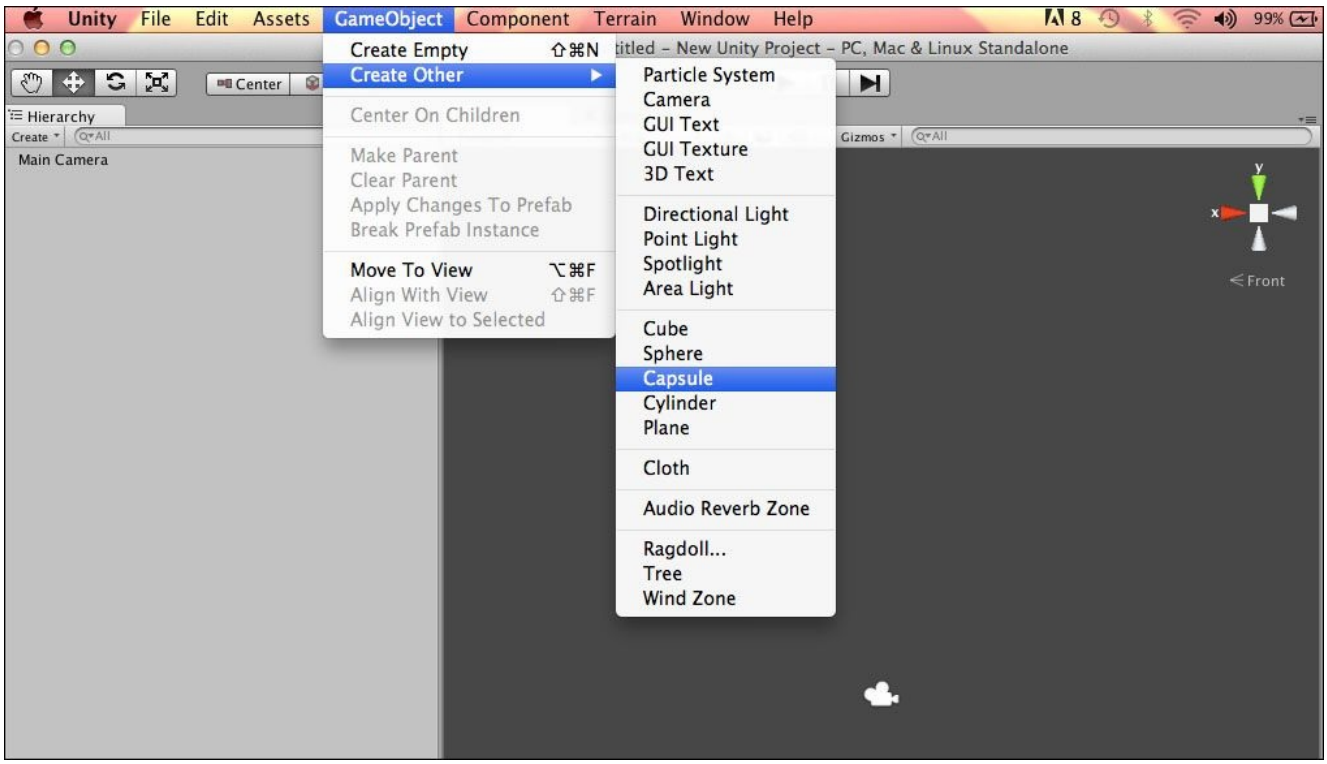
Capsule Collider 3D

Capsule Collider is used for capsule-shaped game objects. This is commonly used for creating a character game object.

Example – implementation of Capsule Collider

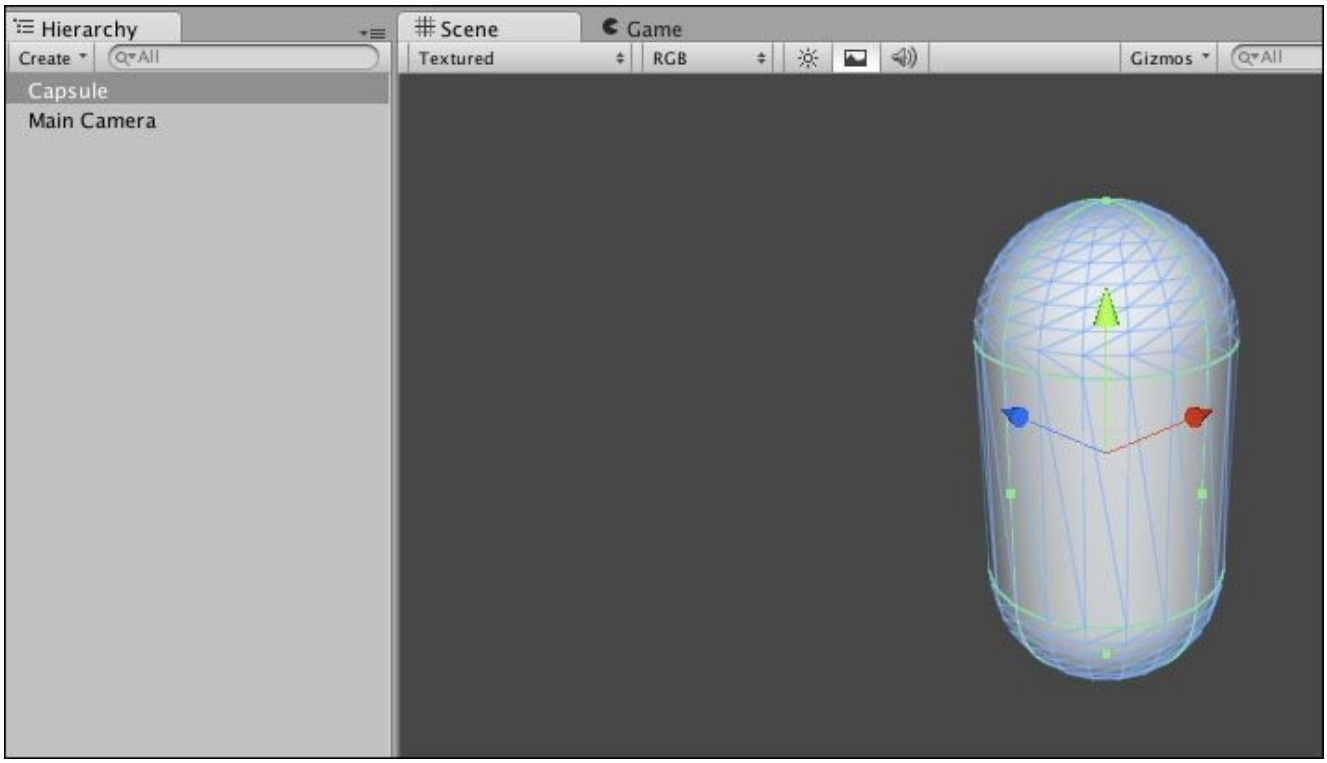
Here, we will see how we can implement Capsule Collider in the scene using the following steps:

1. Create a new scene.
2. As shown in the following screenshot, select the **Capsule** game object and then open the **Inspector** panel:

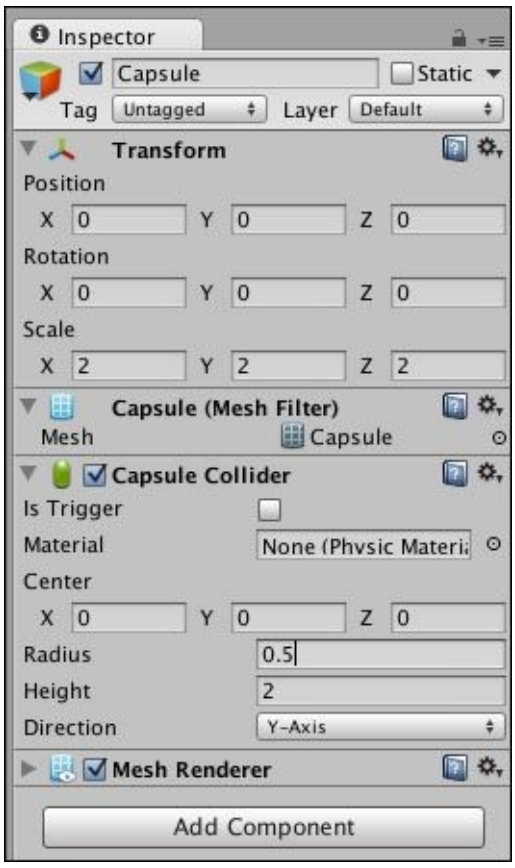


3. Inside the **Inspector** panel, you will see **Capsule Collider**; make sure that it is checked.
4. In the **Inspector** panel, check **Is Trigger** to make game object trigger a collider and fire trigger events.

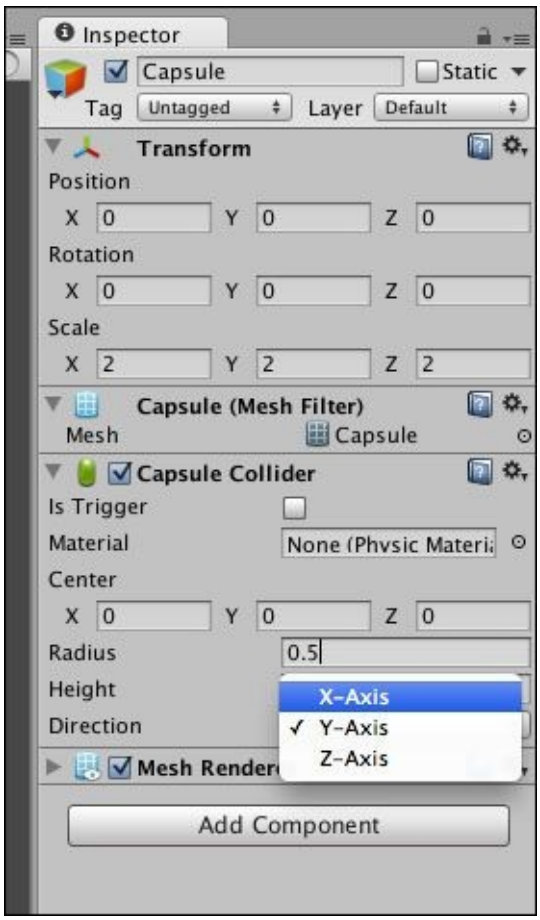
Our capsule object will look as shown in the following screenshot:



5. As shown in the following screenshot, we can decide its shape and size using the dimension and radius; here, it should be relative to game object transform and size:



6. Select **Height** as the length of the capsule body.
7. As shown in the following screenshot, give a direction relative to the game object:



By choosing **X-Axis**, **Y-Axis**, or **Z-Axis**, we give a direction to the game object.

Mesh Collider

Using mesh, Mesh Collider defines the shape of a collision. Although we use it for accurate shape definition for collision, this is expensive in terms of performance. We should avoid the use of Mesh Collider, and wherever we can, use primitive collider (Box, Sphere, and so on).

Note

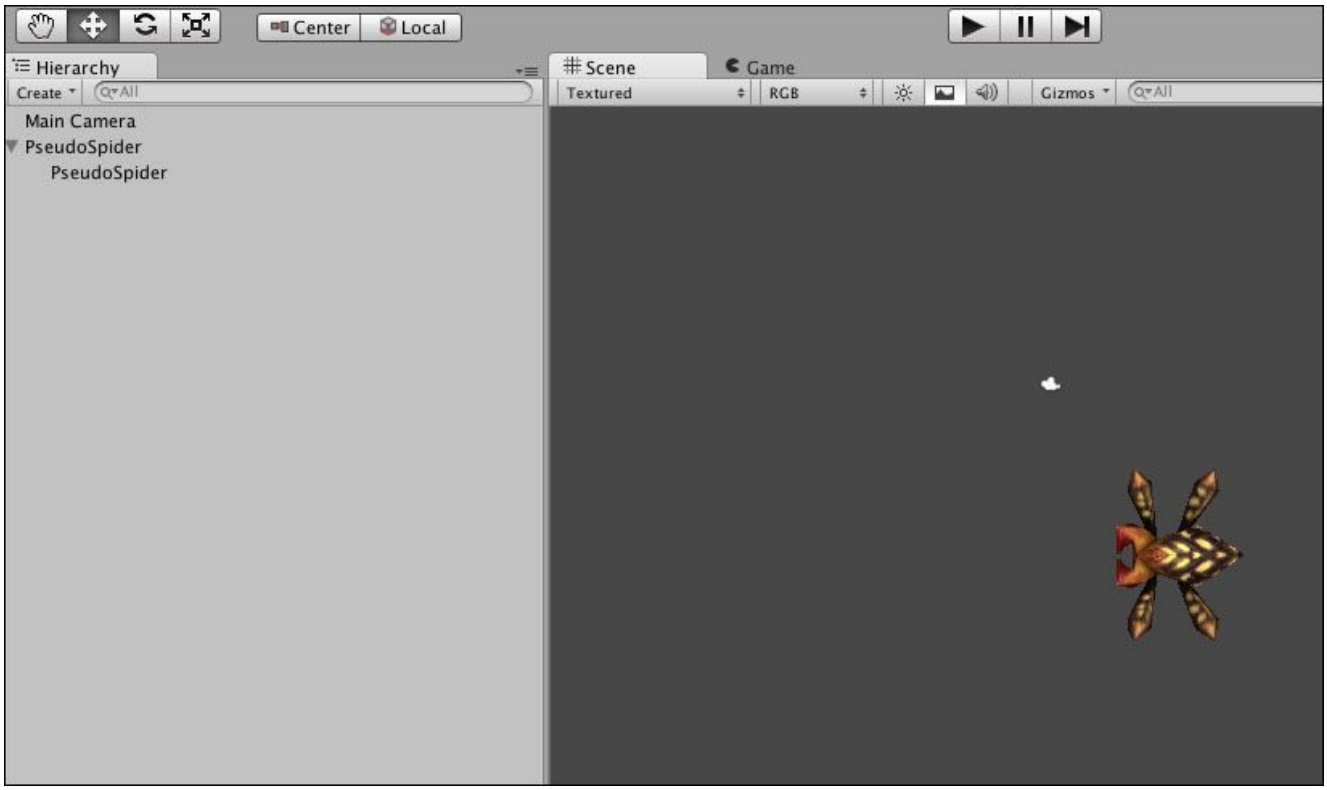
Mesh Collider should be avoided wherever possible by using complex colliders in order to optimize the performance.

Example – implementation of Mesh Collider

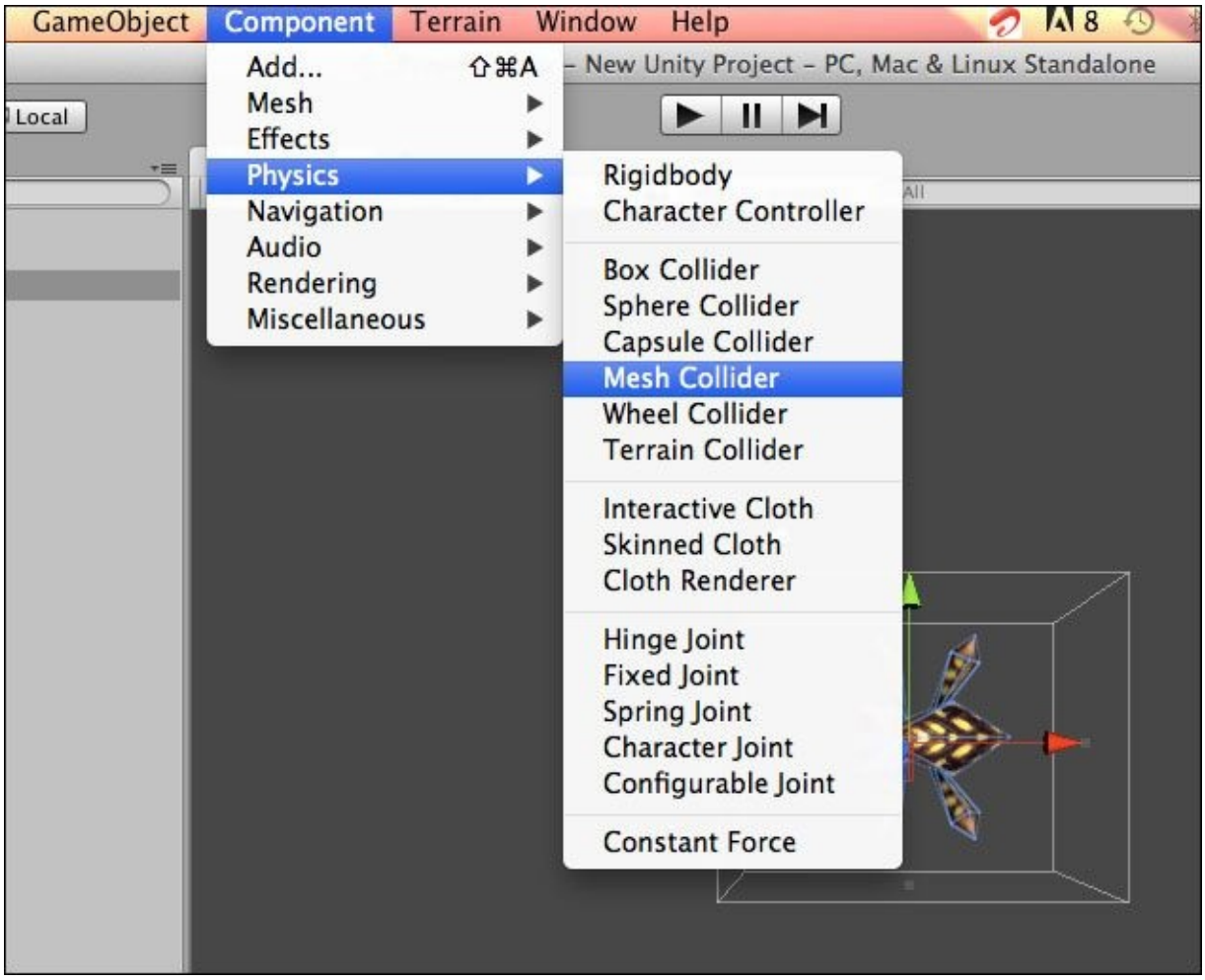
Here, we will see how we can implement Mesh Collider in the scene using the following steps:

1. Create a new scene.
2. Download or use any existing assets you have. I have downloaded a free spider asset from Unity3D's assets store.
3. Import the assets in the scene; and you will see them appear in the scene. As shown

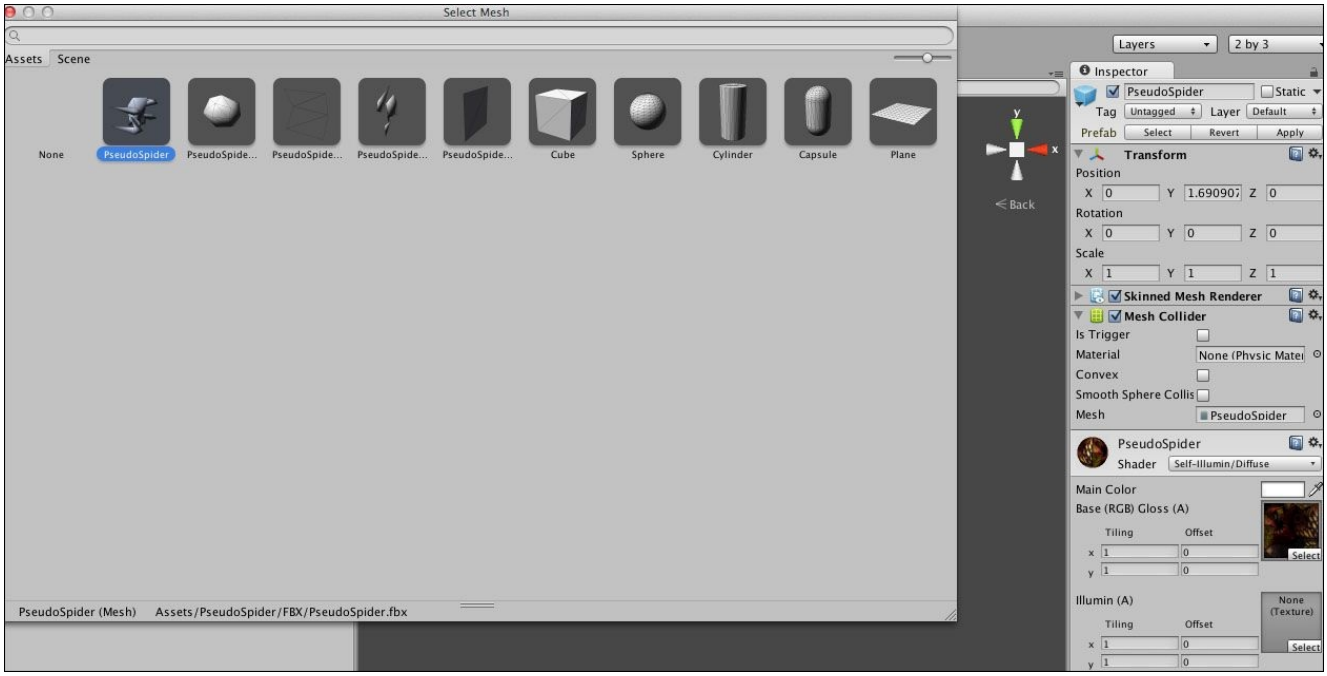
in the following screenshot, you can see your spider:



4. Now, as shown in the following screenshot, we will apply Mesh Collider. Navigate to **Component** | **Physics** | **Mesh Collider**:



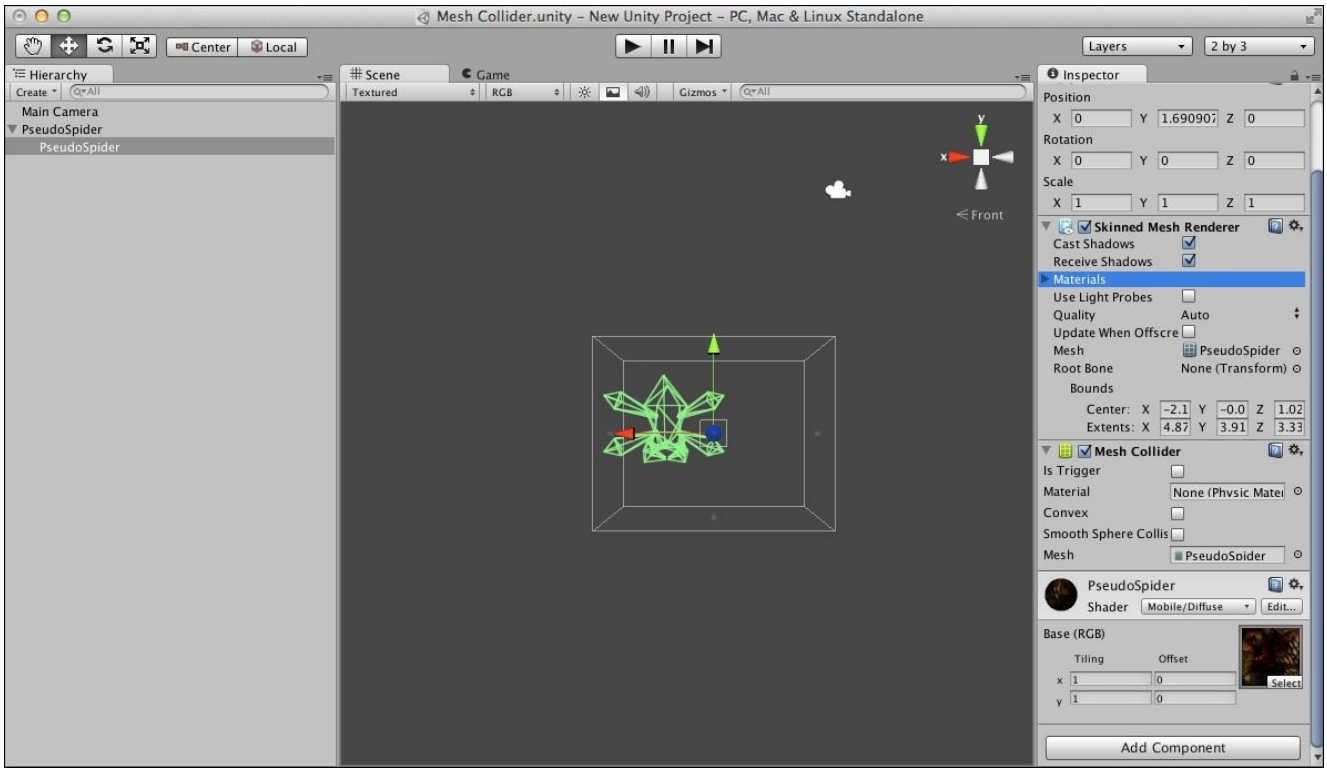
5. As shown in the screenshot, select the mesh where we have to implement Mesh Collider:



6. As shown in the following screenshot, in the **Inspector** panel, you will see a Is

Trigger checkbox, which is unchecked by default. Check **Is Trigger** to make it trigger a collider and fire trigger events:

7. We can mark **Convex** as true if we want this game object to collide with another Mesh Collider.
8. Give a direction relative to the game object:



As shown in the preceding screenshot, by selecting the game object, we open the **Inspector** panel and we can give a direction.

Polygon Collider 2D

For 2D objects where the shape of the game object is irregular, Polygon Collider 2D is used. To edit the collider shape, drag the sprite asset onto the Polygon Collider 2D component in the **Inspector** panel, hold the *shift* key and try to edit the vertex or edges.

Example – implementation of Polygon Collider 2D

The following steps will guide you through to implement Polygon Collider 2D:

1. Create a new scene and name it `Polygon Collider 2D example`.
2. In the **Hierarchy** pane, click on **Create** and select **Sprite** from the drop-down list to create an empty sprite renderer. Name this sprite renderer `Rock` in the **Inspector** window.
3. By default, it will be represented by a box that is not appropriate, so we will create a Polygon Collider instead.
4. Select **Sprite collection** and click on **Open Editor**.
5. Select the **Rock Sprite** option in the **Sprite Collection** editor.
6. Select **Collider Type** as **Polygon**.
7. Switch the **Sprite view** option to collider edit mode.
8. Double-click on an edge to add a control point. Click and drag this to position it.

Note

We can edit the polygon's shape directly by holding down the *shift* key as you move

the mouse over an edge or vertex in the **Scene** view. You can move an existing vertex by shift-dragging when the mouse is over that vertex. If you shift-drag while the mouse is over an edge, a new vertex will be created at the pointer's location. You can remove a vertex by holding down the *control*/*command* key while clicking on it.

9. Edit the collider until you get desired result.

Edge Collider 2D

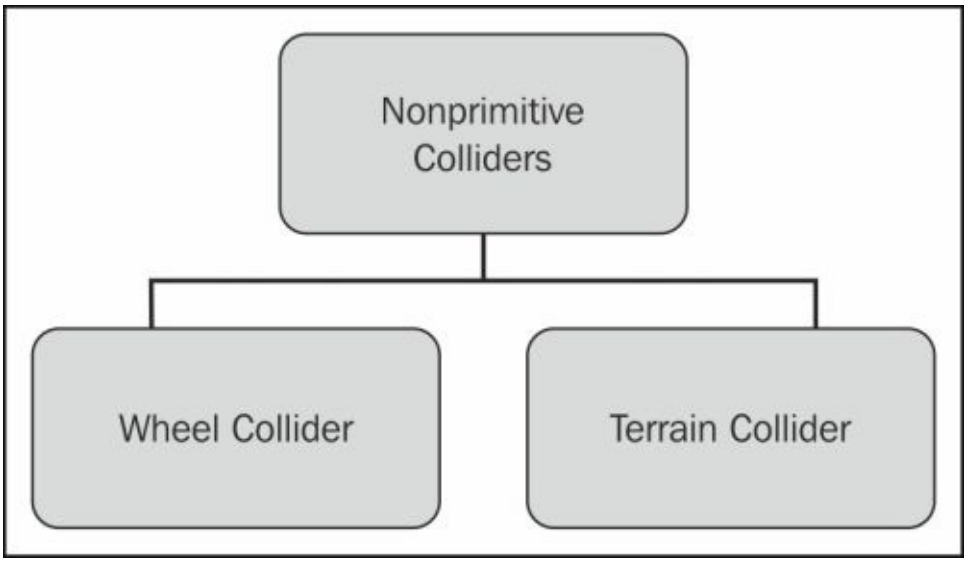
This 2D collider is used where the collision shape for a 2D game object requires precision. Using this collider, a shape is made of line segments. Using the *shift* and *control* keys, we can edit the shape of the collider. We can edit Edge Collider 2D in a similar way as shown in the preceding section on Polygon Collider 2D.

Nonprimitive colliders

Nonprimitive colliders originated from primitive colliders. Especially for vehicles, Unity provides Wheel Collider, and for terrains, a Terrain Collider component. By combining various primitive colliders, we create compound colliders.

Types of nonprimitive colliders

As shown in the following figure, nonprimitive colliders are specific for wheel objects and terrains. Using a Wheel Collider, we can easily provide a collision shape for vehicles. Similarly, using Terrain Collider collision implementation for different shapes becomes easy.



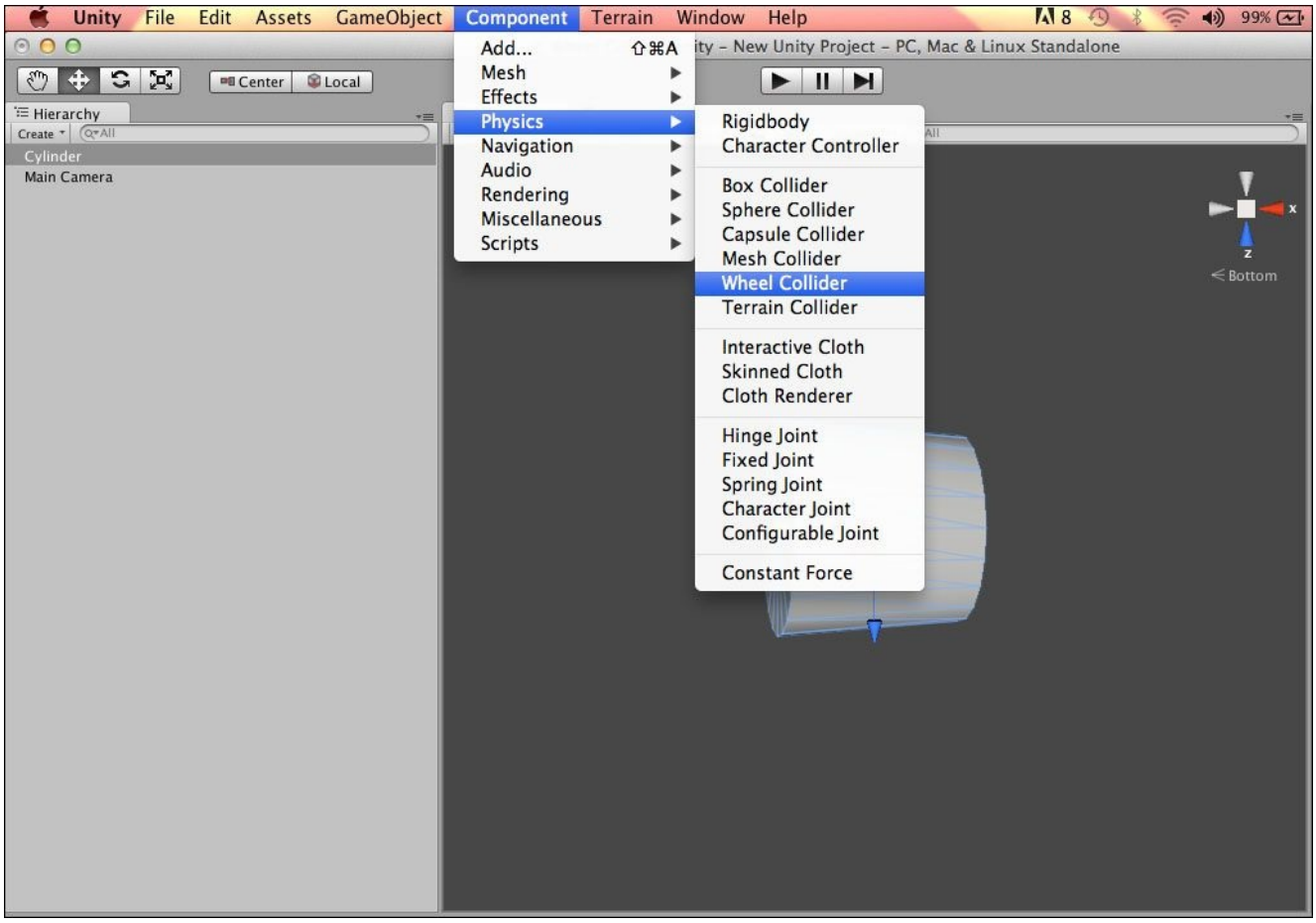
Wheel Collider

This is a nonprimitive collider used for wheels of a vehicle. Wheel Colliders have motorTorque, brakeTorque, radius, and steerAngle properties. Using a friction model, Wheel Colliders are able to give a realistic effect.

Example – implementation of Wheel Colliders

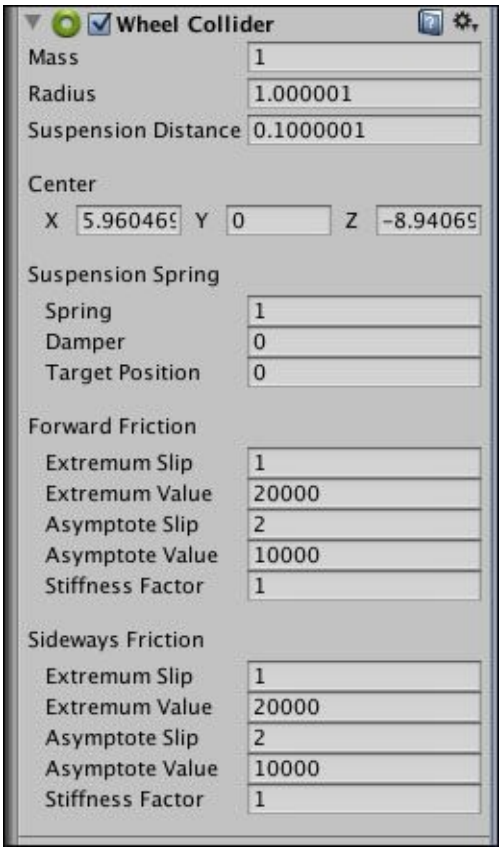
Here, we will see how we can implement a compound collider in the scene using the following steps:

1. Create a new scene.
2. In the **Hierarchy** pane, click on **Create** and select the **Cylinder** game object. Open the **Inspector** panel.
3. Add **Wheel Collider** from **Component**, as shown in the following screenshot:



4. Play with the properties to give a realistic effect.

Let’s open the **Inspector** panel. We will see some different properties; here, using **Mass**, we decide the mass of the wheel game object, which must be greater than 0. In this example, I have used `1`. We can set the radius by using the **Radius** property that defines the radius of the wheel, measured in local space. Have a look at the following screenshot:



As shown in the preceding screenshot, there are other properties that are also used to give a realistic wheel effect; for example, the wheel's collision detection is performed by casting a ray from the center (http://docs.unity3d.com/ScriptReference/WheelCollider-center.html) down the local *y* axis for which we use the Center property. We can extend the wheel's radius downwards by the `suspensionDistance` (http://docs.unity3d.com/ScriptReference/WheelCollider-suspensionDistance.html) amount. By changing `forwardFriction` (http://docs.unity3d.com/ScriptReference/WheelCollider-forwardFriction.html) and `sidewaysFriction` (http://docs.unity3d.com/ScriptReference/WheelCollider-sidewaysFriction.html) based on what material the wheel is hitting, we simulate different road materials.

Static collider

Static collider is used wherein a collision movement is not required. A static collider contains a nontrigger collider without a Rigidbody. This type of game object that does not have any Rigidbody will not get affected by Physics. Static colliders are mostly used to create boundaries or blockages.

Rigidbody Collider

The Rigidbody Collider is exactly the opposite of a static collider; it is a collider with Rigidbody components. Rigidbody Collider will collide with a static collider to create collision events and will get influenced by Physics.

Kinematic Rigidbody Collider

A Kinematic Rigidbody Collider is a collider that has the **Is Kinematic** flag `true`. Mostly, it is used where animation is required. Also, the Kinematic Rigidbody Collider is not influenced by the script.

Trigger Collider

The Trigger Collider is an invisible collider that fires events without any physical interaction. We can define nonphysical area where Trigger Collider will fire event on interaction with game object.

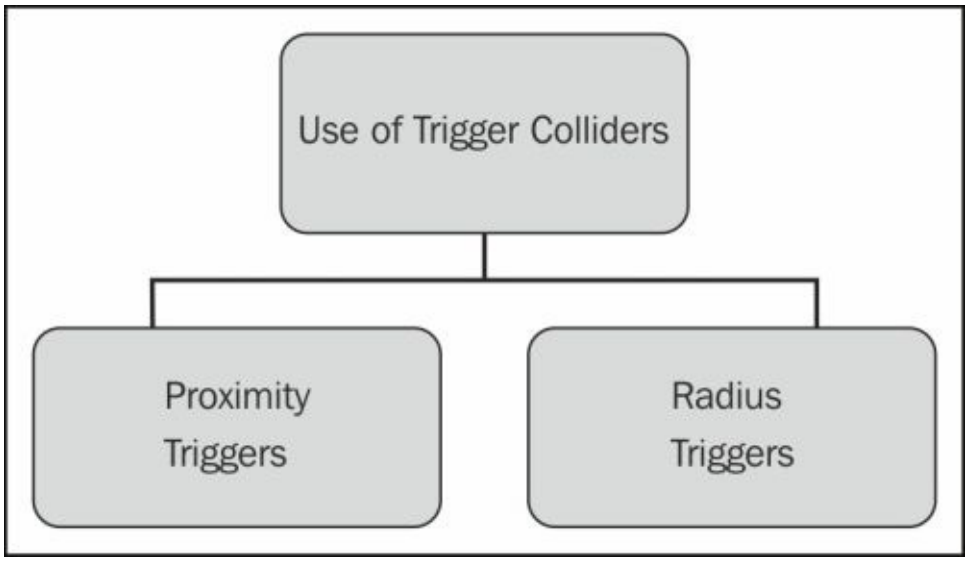
Note

Triggers should be used where a quick response is required. For games such as *Tower Defense* where a quick response is required, Trigger Colliders are very useful.

The following events are called:

- `OnTriggerEnter`: This event is called when the object just enters
- `OnTriggerStay`: This event is called when the object is inside the trigger
- `OnTriggerExit`: This event is called when the object leaves the trigger

When do we use Trigger Collider? As shown in the following figure, mainly, we use Trigger Collider for two scenarios:



An example of proximity triggers

During the development of one of my games, I had to create a game play where when a player reaches the door, the door should open. In this case, I needed to put a Trigger Collider in front of the door. When the player reaches the door, it fires the `OnTriggerEnter` event where I was executing game logic accordingly:

```
    var  isInRange:Boolean = false;

function OnTriggerEnter(col:Collider)
{
  if(col.CompareTag("Player"))
```

```
  {
      // Write here logic for open the door.
   }
}
```

Similarly, we can use `OnTriggerStay` and `OnTriggerExit`:

```
function OnTriggerStay(col:Collider)
{

  if(col.CompareTag("Player"))
  {
isInRange = true;
   }
}
function OnTriggerExit(col:Collider)
{
  if(col.CompareTag("Player"))
  {
isInRange = false;
   }
}
```

An example of radius triggers

In some games such as *Tower Defense*, we are required to spawn troops or shoot enemies in a range; hence, we create a radius. When the game object enters within the defined radius, `OnTriggerEnter` events get fired:

```
    var  isInRange:Boolean = false;
function OnTriggerEnter(col:Collider)
{
  var spawn:SpawnPoint = col.GetComponent(SpawnPoint);
  if(spawn){
    spawn.shootBullets();
  }
}
```

Similarly, we can use `OnTriggerStay` and `OnTriggerExit`:

```
function OnTriggerStay(col:Collider)
{
  isInRang = true;
}
function OnTriggerExit(col:Collider)
{
   isInRang = false;
}
```

Note

Warning

Trigger Colliders respond to raycasts. Make sure your triggers are set to the Ignore Raycasts layer.

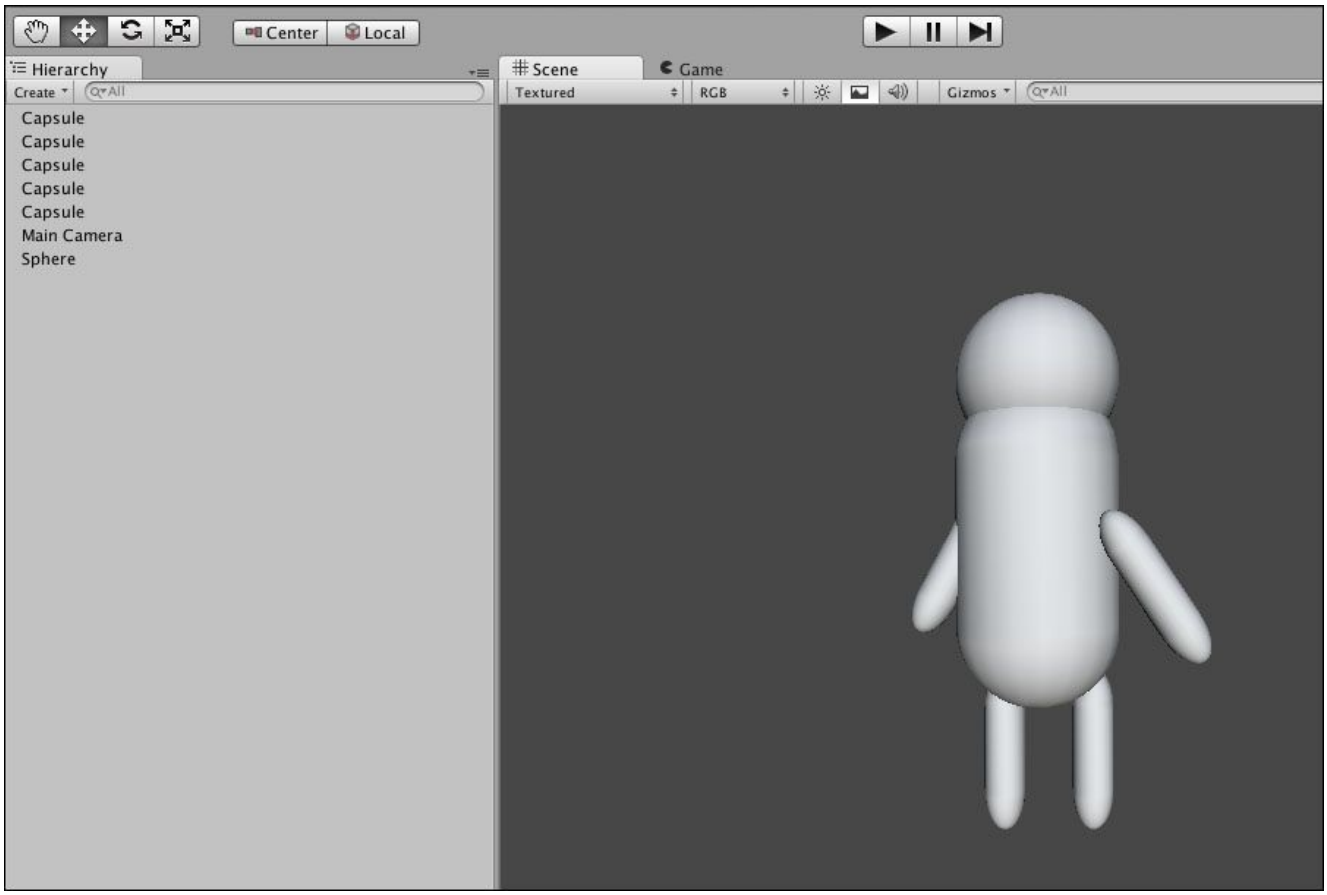
Compound colliders

Instead of using Mesh Collider, sometimes it is better to use combined primitive colliders. For this, we create a parent-child hierarchy of colliders. Let’s see this in the following example.

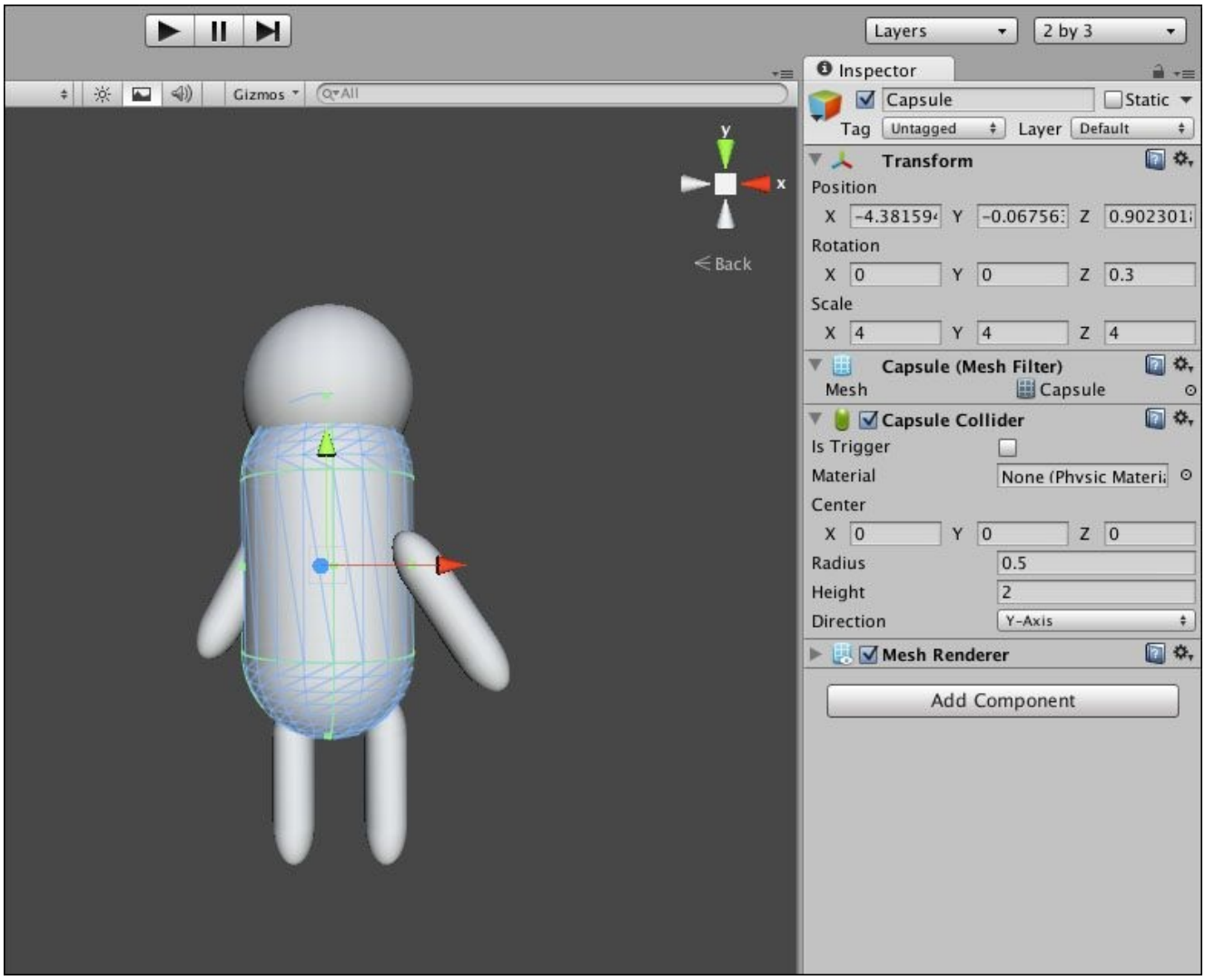
Example – implementation of compound colliders

Here, we will see how we can implement a compound collider in the following scene:

1. Create a new scene.
2. As shown in the following screenshot, select the **Capsule** and **Sphere** game objects and create a character shape:



3. Inside the **Inspector** panel, you will see **Sphere Collider** and **Capsule Collider**. Check the **Is Trigger** option for both to make it trigger a collider and fire trigger events.



In this section, we learned how we can use primitive colliders for a complex game object. We will learn more about animation and compound colliders in later chapters.

Summary

We learned about the different types of colliders and their implementations, how we can implement primitive and nonprimitive colliders, and what their uses are. We also learned the use of Trigger Colliders. In the next chapter, we will learn about the Collision Matrix of colliders. We will see how we can implement the Collision Matrix with nonscript-based and script-based examples. We will also learn about layer-based collision.

Chapter 3. Overview of Collision Matrix

In this chapter, you will learn types of different colliders in Unity3D, which define the collision shape of objects in scene and their Collision Matrix.

You will learn in detail about the following topics:

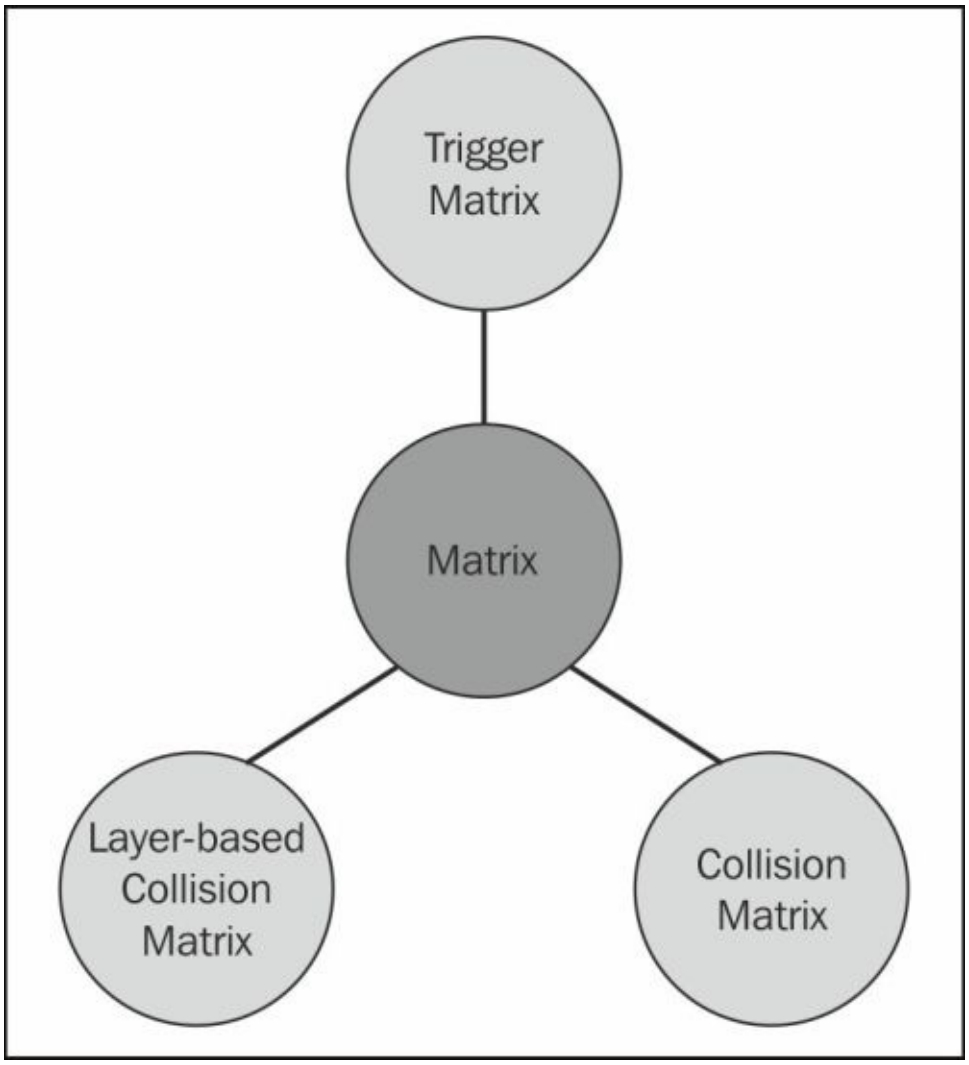
- Collision Matrix
- Trigger Matrix
- Layer-based Collision Matrix

By the end of this chapter, you will get knowledge about the matrix of colliders and their types. Using a matrix configuration of two colliding game objects, we will see how we can define different actions.

During the development with Unity3D, you will encounter with the following terms:

- **Collision Matrix**: This represents a matrix that shows which collider type will show a collision
- **Trigger Matrix**: This represents a matrix that shows which collider type will fire a trigger event
- **Layer-based Collision Matrix**: This represents a matrix that shows layer-based collision

The following figure will show you the different types of matrices:



Let’s take a look at them in detail.

Collision Matrix 3D

You will learn about Collision Matrix for 3D development; this matrix will define which collider type will show collision. We can define different actions by configuring two colliding game objects. The following table defines the collision status of two colliding game objects based on the attached components. To apply Physics, the game object must have Rigidbodies attached. Collision Matrix here represents a matrix that shows which collider type will show a collision.

On collision, the following events will be generated (for the description, please check the previous chapter):

- `OnCollsionEnter()`
- `OnCollisionStay()`
- `OnCollisionExit()`

Note

Only the collider that possesses a Rigidbody will show reaction on collision.

The following figure shows the table of the Collision Matrix for 3D objects:

	Static	Rigidbody	Kinematic Rigidbody	Static Trigger	Rigidbody Trigger	Kinematic Rigidbody Trigger
Static	No	Yes	No	No	No	No
Rigidbody	Yes	Yes	Yes	No	No	No
Kinematic Rigidbody	No	Yes	No	No	No	No
Static Trigger	No	No	No	No	No	No
Rigidbody Trigger	No	No	No	No	No	No
Kinematic Rigidbody Trigger	No	No	No	No	No	No

Let’s see how we read the preceding table with a few examples:

- A static collider will detect collision with a Rigidbody
- A Rigidbody Collider will detect collision with static, Rigidbody, and Kinematic Rigidbody Colliders
- A Kinematic Rigidbody will show reaction on collision with a Rigidbody

- In other conditions, no collision will be detected

Trigger Matrix

Trigger Matrix here represents a matrix that shows which collider type will fire trigger events. On trigger, the following events will be generated (for the description, please check the previous chapter):

- `OnTiggerEnter()`
- `OnTriggerStay()`
- `OnTriggerExit()`

Note

Only Trigger Collider will fire trigger events.

The following figure shows the table of the Trigger Matrix for 3D objects:

	Static	Rigidbody	Kinematic Rigidbody	Static Trigger	Rigidbody Trigger	Kinematic Rigidbody Trigger
Static	No	No	No	No	Yes	Yes
Rigidbody	No	No	No	Yes	Yes	Yes
Kinematic Rigidbody	No	No	No	Yes	Yes	Yes
Static Trigger	No	Yes	Yes	No	Yes	Yes
Rigidbody Trigger	Yes	Yes	Yes	Yes	Yes	Yes
Kinematic Rigidbody Trigger	Yes	Yes	Yes	Yes	Yes	Yes

Let’s see how we can read the preceding table with the following examples:

- The static collider will fire trigger events on collision with a Rigidbody Trigger Collider and Kinematic Rigidbody Collider
- The Rigidbody Collider will fire trigger events on collision with a static Trigger Collider, Rigidbody Trigger Collider, and Kinematic Trigger Collider
- The Kinematic Rigidbody Collider will fire trigger events on collision with a static Trigger Collider, Rigidbody Trigger Collider, and Kinematic Trigger Collider
- The static Trigger Collider will fire trigger events on collision with a Rigidbody Collider, Kinematic Rigidbody Collider, Rigidbody Trigger and Kinematic Trigger Collider

Matrix for 2D Objects

Although in Unity3D, Physics for 2D objects and 3D objects are very similar, there are some differences in sending messages during collision.

There are some important differences that we will see in the following table. The matrix given in this table is based on the assumption that we are using 2D variants of colliders and Rigidbodies.

The table is based on instances where collision detection occurs and Collision2D messages are sent.

	Static	Rigidbody	Kinematic Rigidbody	Static Trigger	Rigidbody Trigger	Kinematic Rigidbody Trigger
Static	No	Yes	No	No	No	No
Rigidbody	Yes	Yes	Yes	No	No	No
Kinematic Rigidbody	No	Yes	No	No	No	No
Static Trigger	No	No	No	No	No	No
Rigidbody Trigger	No	No	No	No	No	No
Kinematic Rigidbody Trigger	No	No	No	No	No	No

Let’s see how we can read the preceding table with the following examples:

- The static collider will fire collision events on collision with a Rigidbody only
- The Rigidbody will show collision with a static collider, Rigidbody, and Kinematic Rigidbody Collider
- The Kinematic Rigidbody will fire collision events with a Rigidbody

The following figure shows instances where Trigger2D messages are sent upon collision:

	Static	Rigidbody	Kinematic Rigidbody	Static Trigger	Rigidbody Trigger	Kinematic Rigidbody Trigger
Static	No	No	No	No	Yes	No
Rigidbody	No	No	No	Yes	Yes	Yes
Kinematic Rigidbody	No	No	No	No	Yes	No
Static Trigger	No	Yes	No	No	Yes	No
Rigidbody Trigger	Yes	Yes	Yes	Yes	Yes	Yes
Kinematic Rigidbody Trigger	No	Yes	No	No	Yes	No

Let’s see how we can read the preceding table with the following examples:

- The static collider will fire trigger events on collision with a Rigidbody only
- The Rigidbody will show collision with a static Trigger and Rigidbody Trigger
- The Kinematic Rigidbody will fire collision events with a Rigidbody Trigger

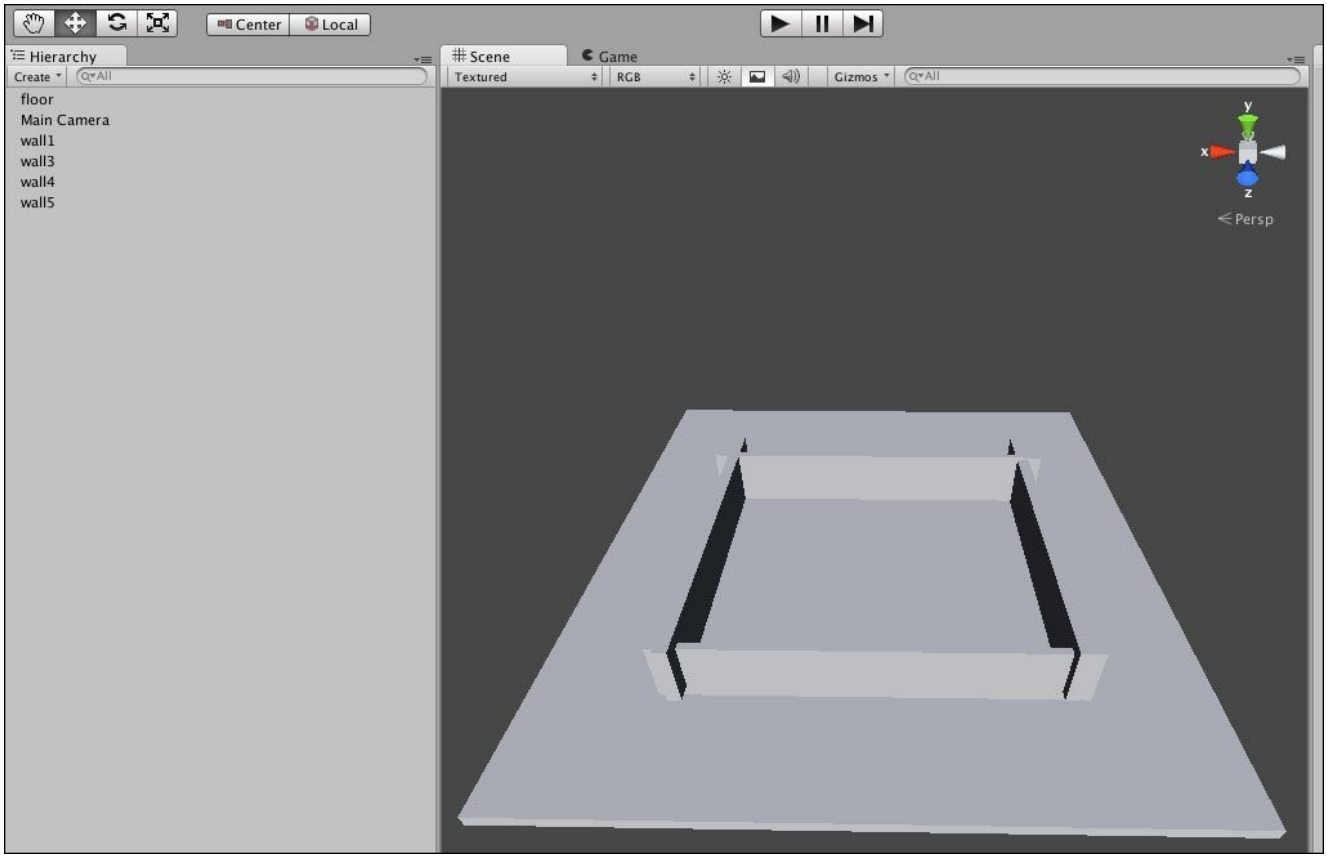
Layers and Collision Matrix

By default, if multiple game objects are on the default layer, they will all collide with each other. In that case, we can say everything will collide with everything. In game development, often we need to decide which object will collide with which one. For this, we need to define a layer-specific division of game objects. We put each type of object in different layers. In Collision Matrix, for each new layer, a new column and row are added. Matrix defines the interaction between the layers. By default, a new layer collides with all the other layers. A developer needs to set up the interaction between the layers. Using a correct setup, we can avoid unwanted collision.

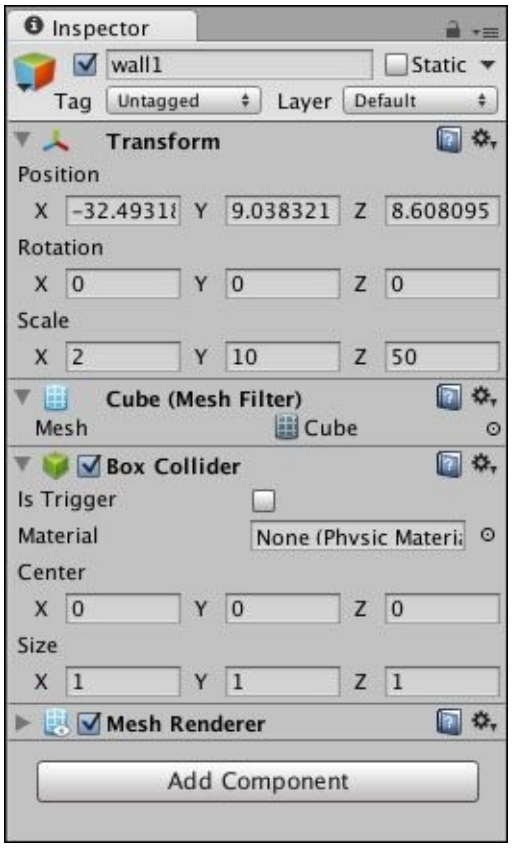
An example of a layer-based Collision Matrix

Let’s take a look at an example of setting Collision Matrix for layers:

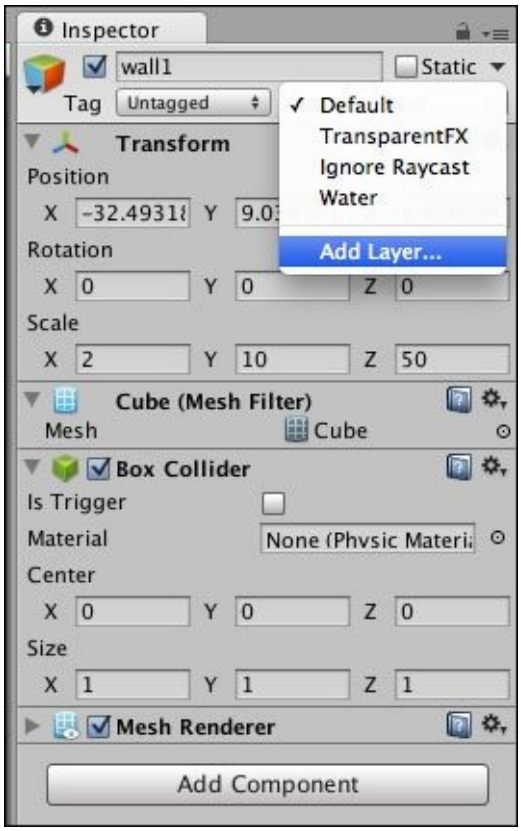
1. Create a new scene and name it `Layer Collision`.
2. Create a box container game object, as shown in the following screenshot:



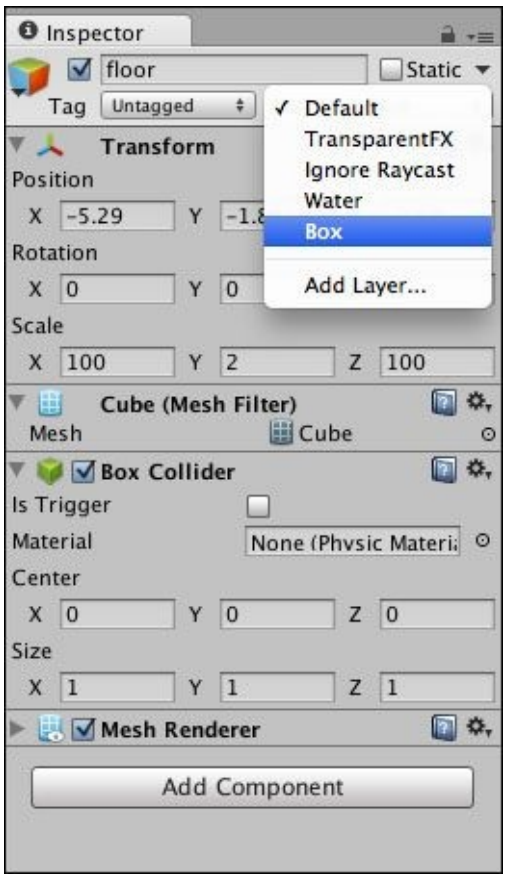
By default, all the objects will appear on the default layer, as shown in the following screenshot:



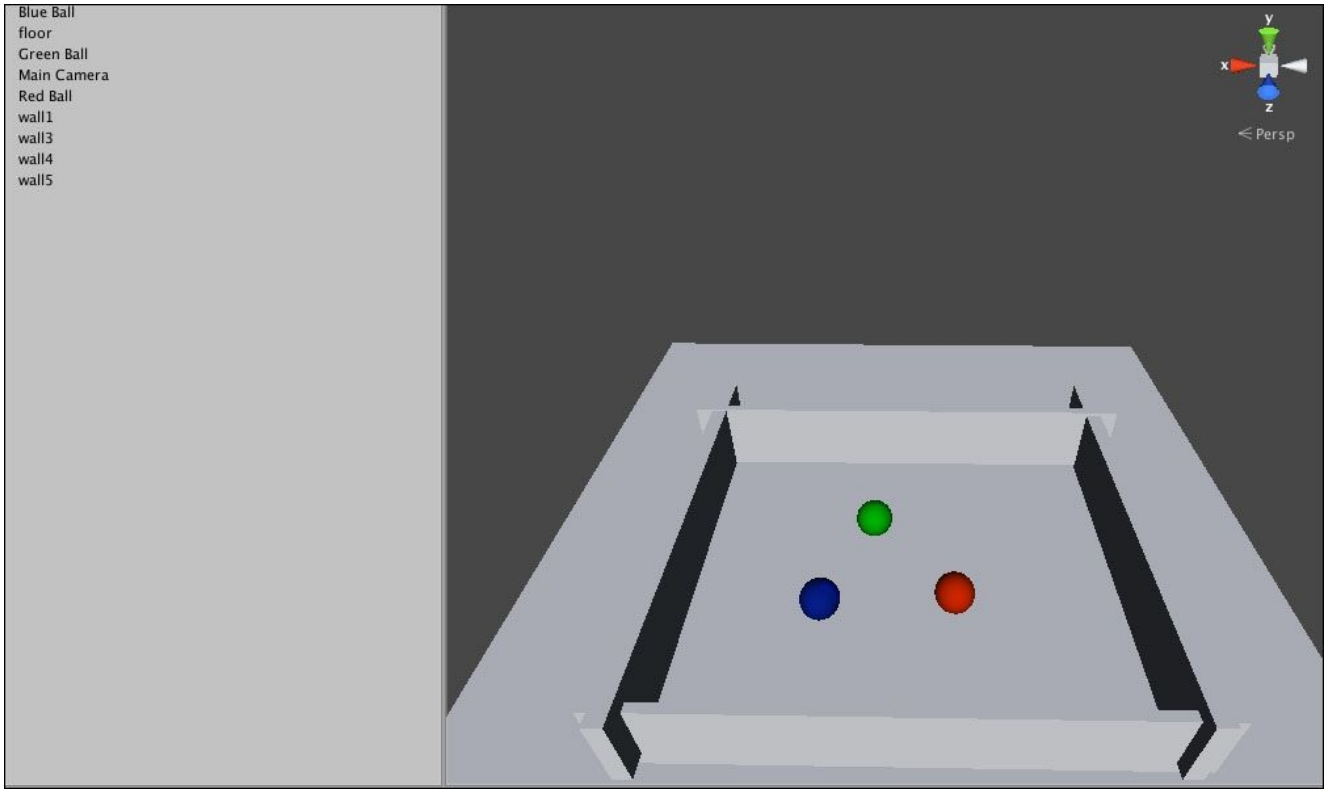
3. Add and set up the Box layer for floor and wall game objects. As shown in the following screenshot, by clicking on **Layer**, we will see a drop-down list of existing layers. We can add a new layer by clicking on **Add Layer**.



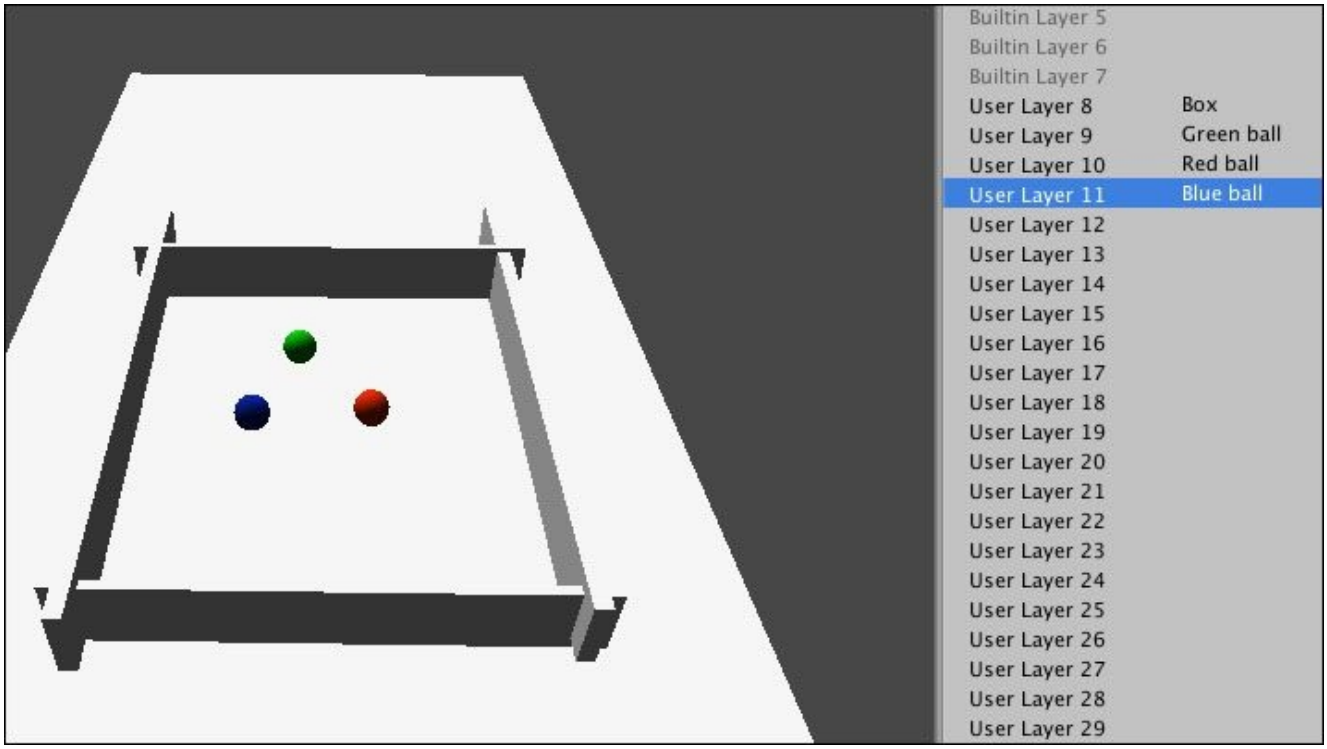
4. As shown in the following screenshot, we have created a new layer **Box**. By selecting **Box**, we can specify the layer for all the box game objects.



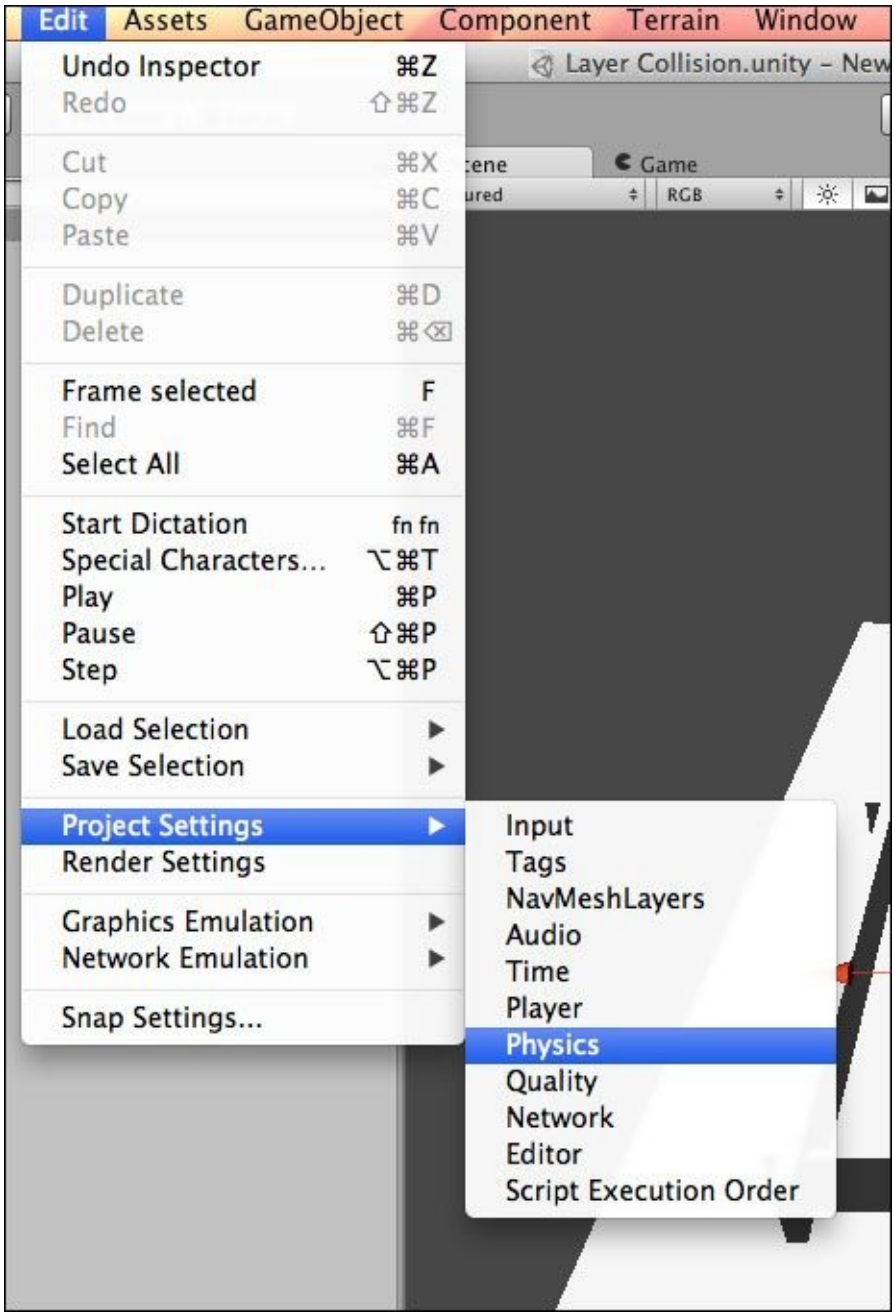
5. Inside the box container, add a red ball, green ball, and blue ball.



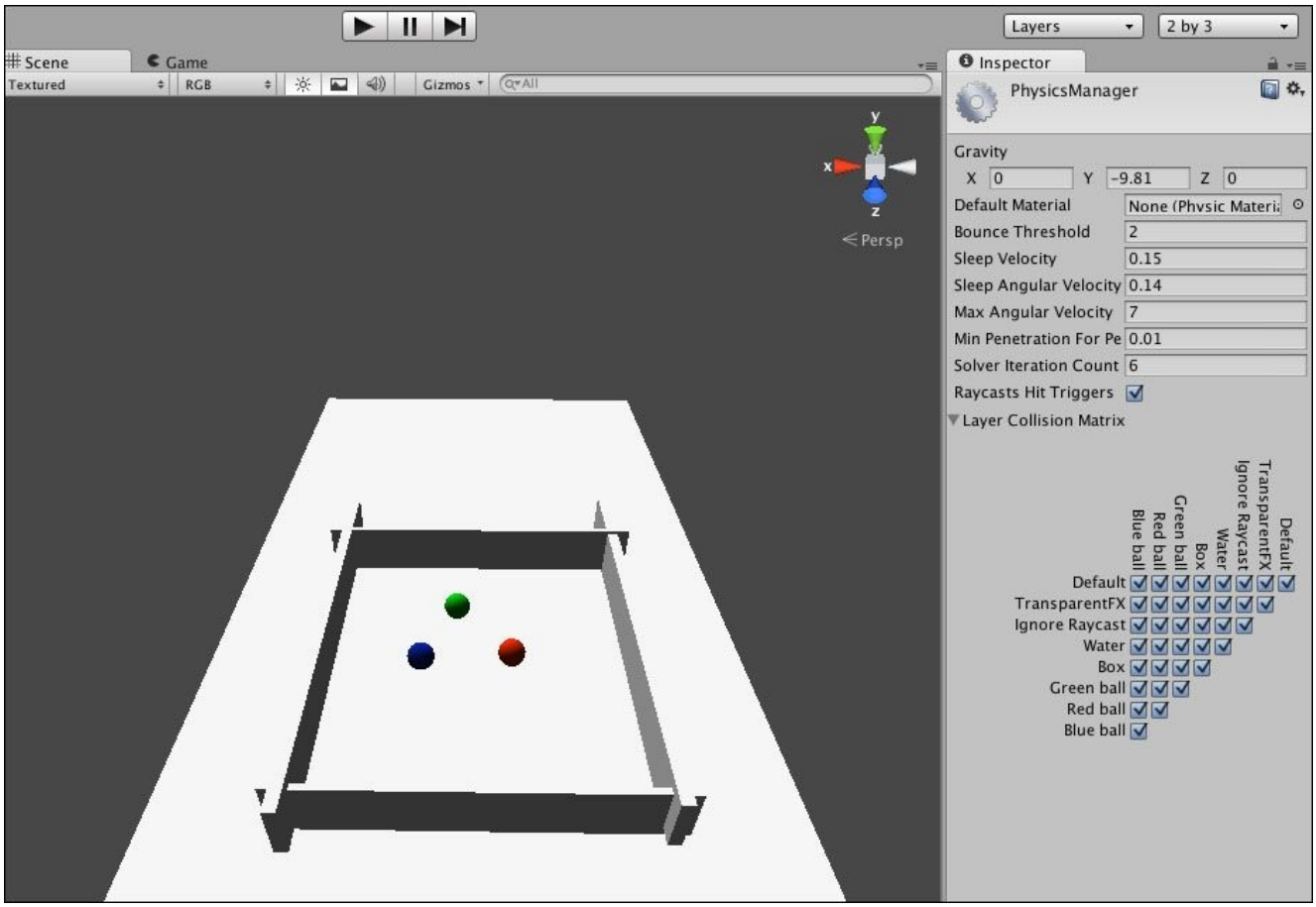
6. Similar to step 3, we add and set up different layers for the balls. As shown in the following screenshot, we have created layers for the red ball, green ball, and blue ball:



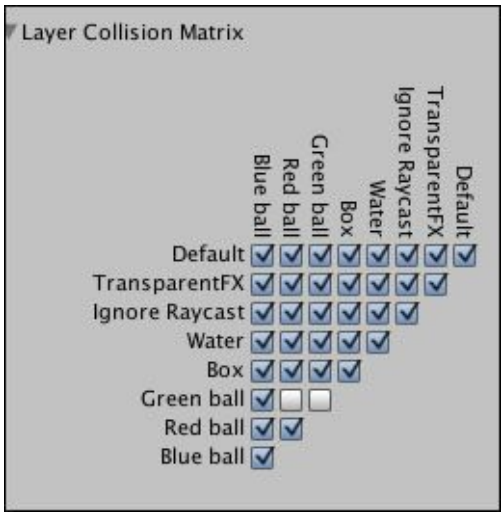
7. To set up collision rules for game objects, let's open **Matrix**. As shown in the following screenshot, navigate to **Edit** | **Project Settings** and select **Physics**:



8. As shown in the following screenshot, a collision table will be shown in the **Inspector** panel. By default, every object will collide with every another object.



9. Check and uncheck the checkboxes of matrix to set up rules, as shown in the following screenshot. We have set up Collision Matrix for a green ball, red ball, blue ball, and a box.



Using the preceding steps, we can set up a layer-based Collision Matrix.

Create a new JavaScript and attach it to the main camera and run the test.

Write the following code inside this JavaScript file:

```
  #pragma strict
  function Start () {
  }
  function Update () {
  }


function OnCollisionEnter(Collision collisionInfo)
{
  Debug.Log("Detected collision between " + gameObject.name + " and " +
collisionInfo.collider.name);

}

function OnCollisionStay(Collision collisionInfo)
{
  Debug.Log(gameObject.name + " and " + collisionInfo.collider.name + " are
still colliding");
}

function OnCollisionExit(Collision collisionInfo)
{
  Debug.Log(gameObject.name + " and " + collisionInfo.collider.name + " are
no longer colliding");
}
```

Using the preceding script, you can check which objects are colliding with one another and for how long.

Collision Matrix and a script

Let’s see how we can ignore collision for specific objects using a script. Often, in interactive development, we need to set up rules for collision between objects. For example, we need to set up a rule where projectiles should not collide with the objects shooting them. Although, using custom layer-based collision on GUI, we can define a layer-based game objects collision rule, we also can handle it using a script. We can use `IgnoreCollision` Physics of Unity3D for creating Collision Matrix.

Note

Limitation of IgnoreCollision

Collision Matrix does not store the state when saving the scene and we can apply only on active game objects.

An example of a script-based Collision Matrix

The following is the example of a script-based Collision Matrix:

1. Create a new scene and name it `Script-Collision-Matrix`.
2. Create a wall using **Plane**.
3. Create a **Bullet** game object.
4. Create a shooting object. Now, we need to set up a rule that a bullet should not collide with the shooting object.
5. Create a script and attach it to the main camera object. Add the following code to the script:

```
var bulletPrefab : Transform;
function Start () {
    var bullet = Instantiate(bulletPrefab) as Transform;
    Physics.IgnoreCollision(bullet.collider, collider);
}
```

Run and test the script, you will see that a bullet is ignoring collision with the shooting object.

Summary

In this chapter, you learned about different Collision Matrix and Trigger Matrix for 2D and 3D game objects. You also learned how we can set up a layer-based Collision Matrix and how we handle it with the scripts. In the next chapter, you will learn about Rigidbodies and their behaviors. The next chapter serves as a detailed description and will give you the uses and types of different Rigidbodies in Unity3D, which will covering topics such as Physics Rigidbody, Kinematic Rigidbody, and the properties of a Rigidbody.

Chapter 4. Rigidbody Types and Their Properties

In this chapter, we will learn about the types of Rigidbody components and their properties. To detect collision, add gravity, and several other Physics functionalities, a game object must have a Rigidbody component.

In this chapter, we will cover the following topics:

- Types of Rigidbody components
- Properties of Rigidbody components
- Example using Rigidbody components

We can handle a Rigidbody component using a script as well as by manually applying linear and angular velocity.

Note

For accurate Physics calculation, a Rigidbody component requires a collider component.

Types of Rigidbody components

We enable our game objects to act under the control of Physics using Rigidbodies. Here, we will learn how to use and handle a Rigidbody component. A Rigidbody can be handled in two ways:

- Physics Rigidbody
- Kinematic Rigidbody

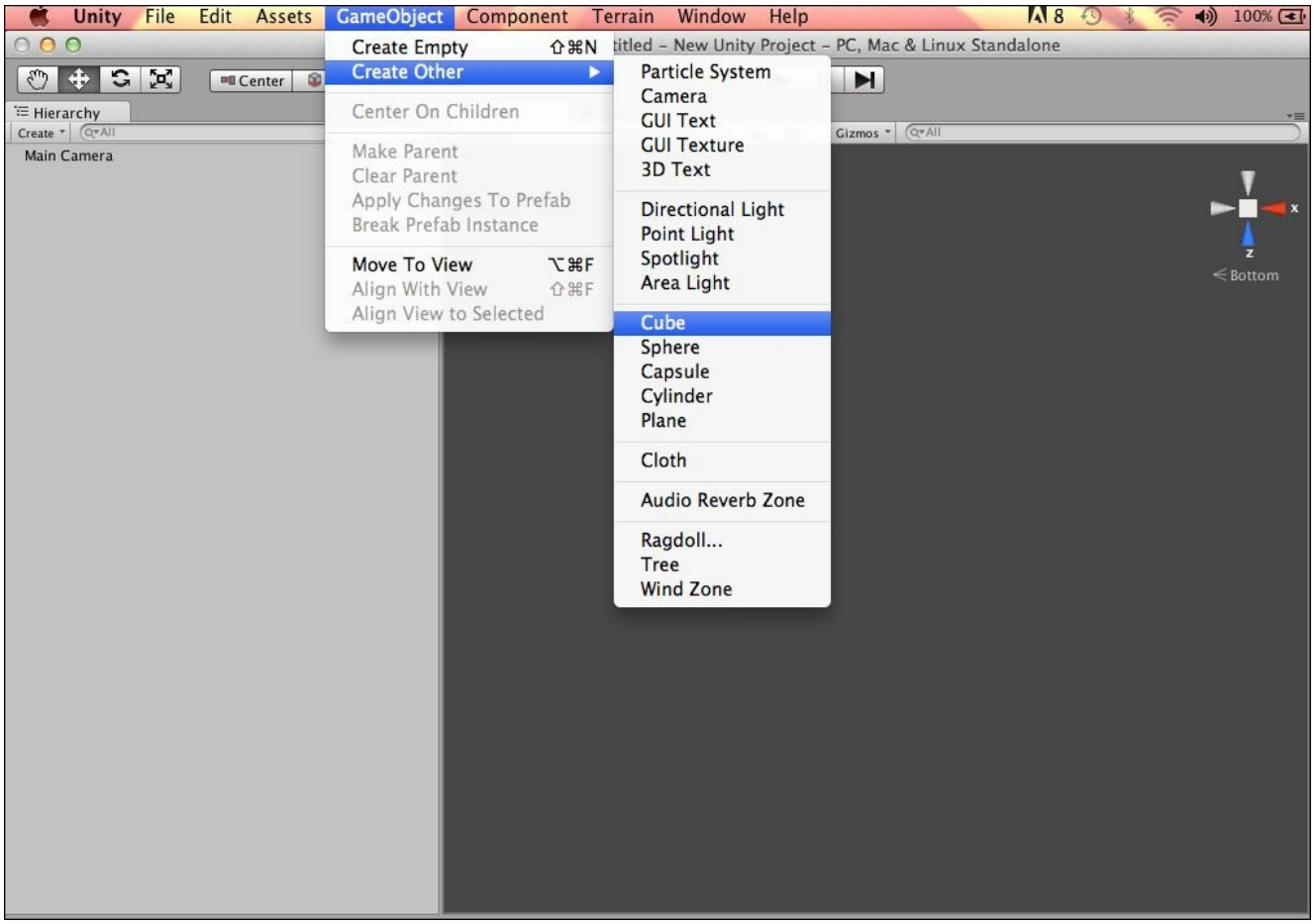
Physics Rigidbody

To influence a game object by gravity or forces, the game object must have a Rigidbody. When a Rigidbody is completely controlled by the engine, it is called a Physics Rigidbody. We can manually apply forces and torque to handle linear and angular velocity of a game object. Manual implementation of forces and torque gives you the freedom to get the desired effect. Let's look at an example of a Rigidbody.

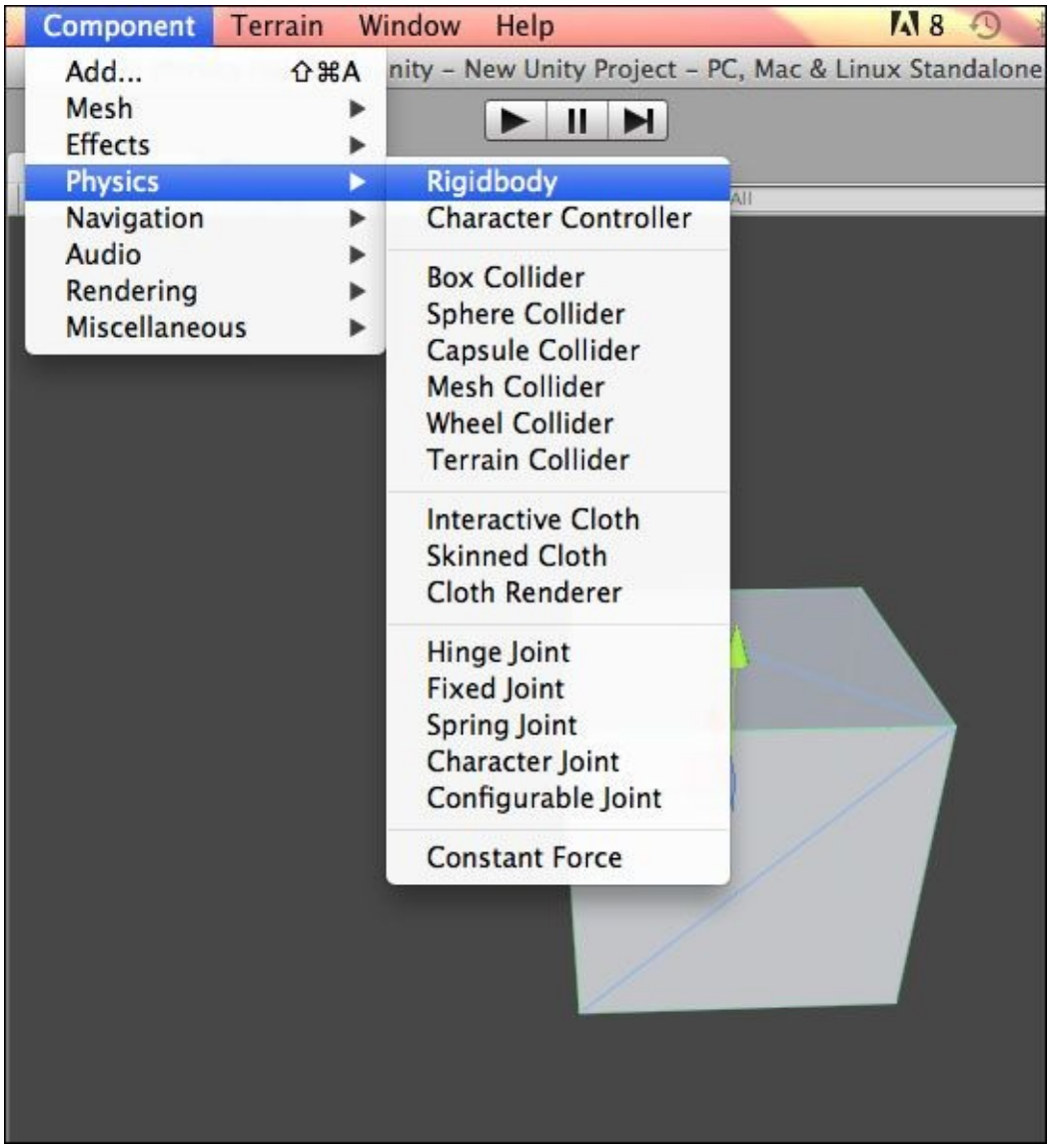
An example of creating a Physics Rigidbody

Use the following steps to apply a Physics Rigidbody:

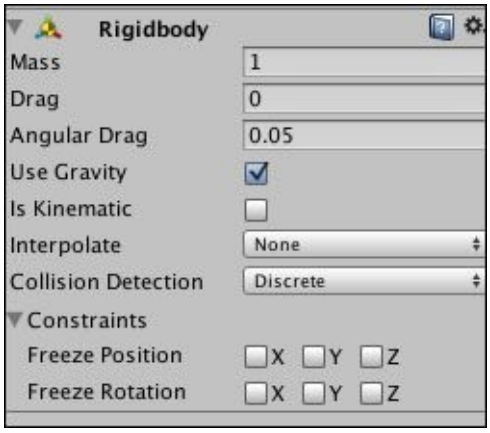
1. We will start by creating a new scene and save it as `Physics Rigidbody`.
2. Create a **Cube** game object, as shown in the following screenshot:



3. To enable our game objects to act under the control of Physics, we use Rigidbodies. Add the **Rigidbody** component to the cube game object, as shown in the following screenshot:



4. In the **Inspector** panel, make sure that the **Is Kinematic** property is unchecked. If we enable **Is Kinematic**, the object can only be manipulated by its Transform property. Generally, we use this for moving platforms or if we want to animate a Rigidbody that has a HingeJoint attached.



As shown in the previous steps, we can get the desired effect using the different properties.

Kinematic Rigidbody

When we check the **Is Kinematic** property of a Rigidbody, it is known as a Kinematic Rigidbody. We cannot apply forces or torque manually to a Kinematic Rigidbody. We can move a Kinematic Rigidbody by changing the values of a GameObject's Transform component. This is useful for moving platforms or animated HingeJoint GameObjects where the engine does not handle the object directly and where we can manipulate its Transform property as per our requirement.

We can use this property by checking it in the **Inspector** panel, as shown in the previous screenshot. Now, we will learn about the properties of a Rigidbody and their implementation.

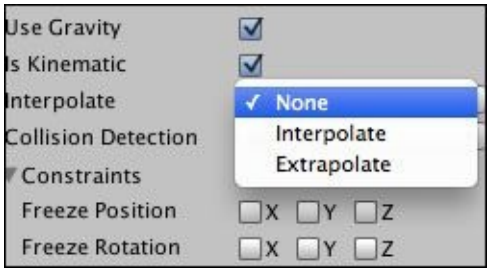
Properties of Rigidbody components

Let’s get familiar with the properties of a Rigidbody:

- **Mass**: If we look at the **Inspector** panel of a Rigidbody component, we will see **Mass** as one of the listed properties. This property of a Rigidbody defines the mass that should be based on the relative size and density of the object that is attached to it. The mass of a Rigidbody defines how much force is required to move the Rigidbody fast or slow. We can calculate the force using Newton’s law of motion as follows:

  $F=ma$, that is, Force = Mass x Acceleration

- **Drag**: This property defines the linear velocity due to air resistance. For example, if we need to add an outer-space behavior to a game object, we set this value as `0`. An object with mass `1` should have a **Drag** value of `998` to resist the force of gravity.
- **Angular Drag**: This property defines the angular velocity due to air resistance. For example, if we need to add an infinitely-spinning-unless-used external force behavior on a game object, we set this value as `0`.
- **Use Gravity**: If we want our game object to be affected by gravity, we use this behavior.
- **Is Kinematic**: To add kinematic behavior, we set this to true. By setting its behavior to true, we can directly change the position and orientation of the Rigidbody using its Transform properties.
- **Interpolate**: Using this property, we make a Rigidbody move smoothly. We can adjust the **Interpolate** method using **None**, **Interpolate**, and **Extrapolate**, as seen in the following screenshot:



  Using **None**, no interpolation is applied; using **Interpolate**, we smoothly transform based on its previous frame; and in **Extrapolate**, we smoothly transform based on its estimated next frame.

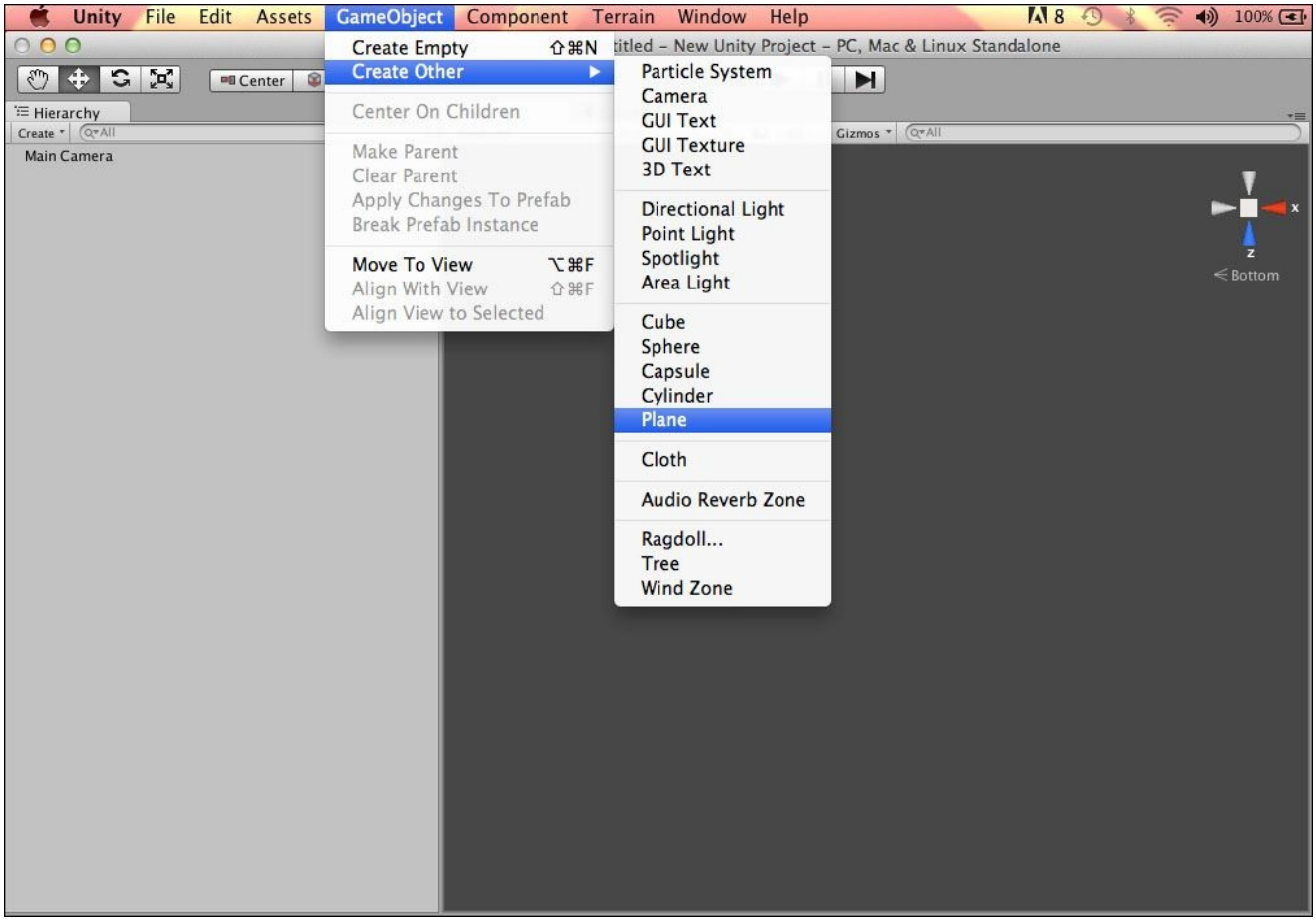
- **Collision Detection**: Using this property, we can determine how the Rigidbody will perform collision detection with other Rigidbodies. This property consists of the following subproperties:
  - **Discreet**: By setting this property, we use the simplest form of collision detection. At each frame, an intersection test is done. The drawback of this property is that small, fast moving objects will pass directly through solid objects.

    - **Continuous**: By setting this property, a Rigidbody performs continuous collision detection against all the other static colliders.
    - **Continuous Dynamic**: By setting this property, a Rigidbody detects the collision with fast moving objects.
- **Constraints**: This property is used for Physics Rigidbodies. Using this property, we can make the position and orientation of a Rigidbody constraint. This property consists of the following subproperties:
    - **FreezePosition**: By setting this property, a Rigidbody constrains the linear motion
    - **FreezeRotation**: By setting this property, the rotation of a Rigidbody is prevented

Example using a Rigidbody

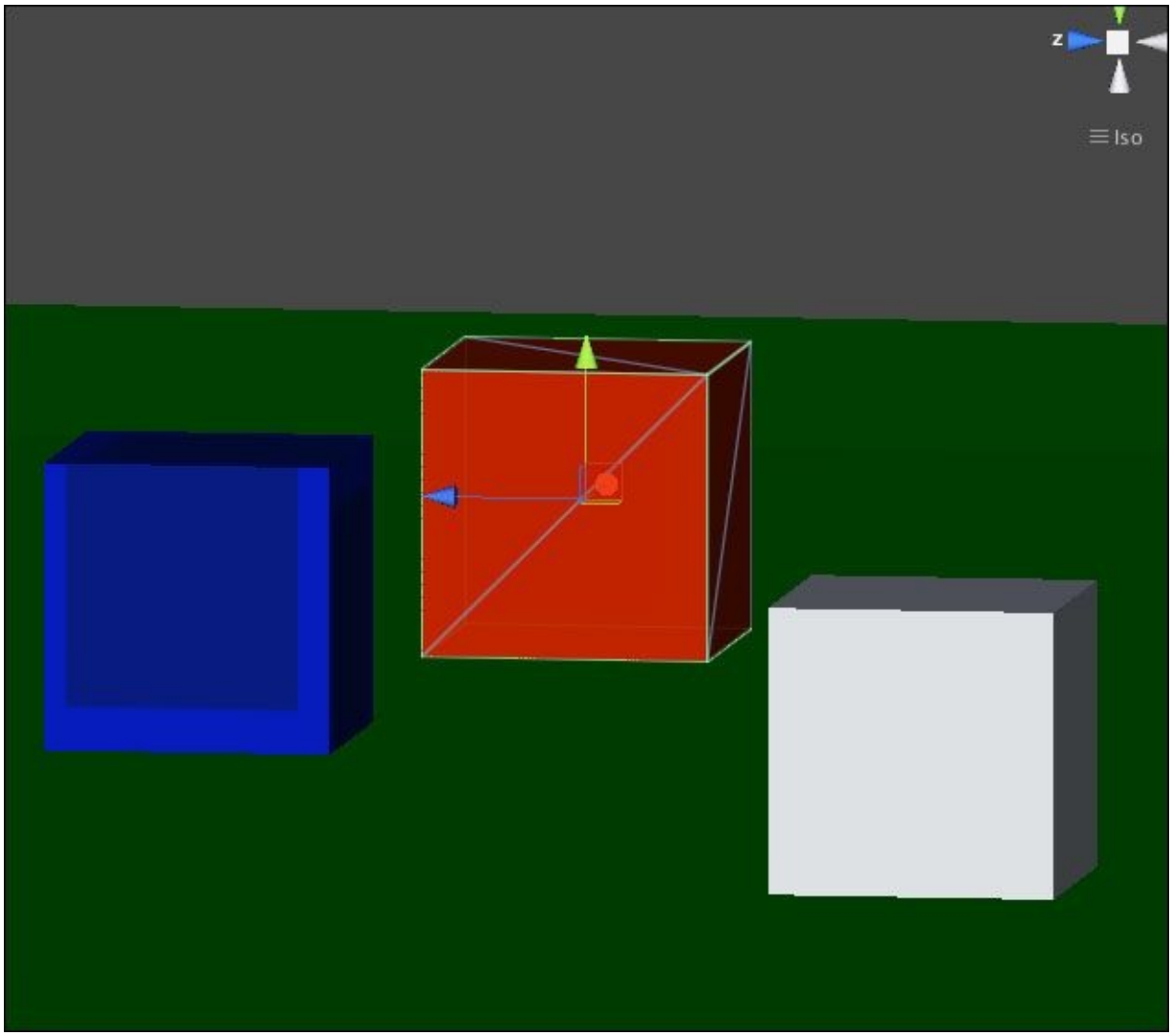
Let's assume that in our game, we require a cube to be our character. In this case, we won't be using the character controller. Let's follow these steps:

1. Create a new scene and save it as `Rigidbody example`.
2. As shown in the following screenshot, create a **Plane** game object and set its properties to position `0` (all axes), rotation `0` (all axes), and scale `20`, `1`, `20` for the *x*, *y*, and *z* axes respectively:



3. Add a **Cube** game object and a Rigidbody Collider. Place it above the plane.
4. As shown in the following screenshot, add material to the red cube:

5. Create another **Cube** game object. I have used a white cube without a Rigidbody.
6. Create another **Cube** object without a Rigidbody. I have used a blue cube, as shown in the following screenshot, and have set the **Is Trigger** property to true:



In the preceding screenshot, we have set the **Is Trigger** property in the blue cube.

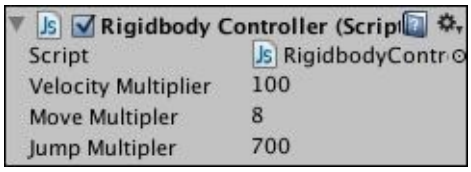
7. Create a new script and save it as `TriggerController.js`; add it to the **Trigger Collider** game object and put the following code inside it:

```
functionOnTriggerEnter( other : Collider )
{
  Debug.Log("OnTrigger Event");
}

functionOnTriggerExit( other : Collider )
{
  Debug.Log("OnTriggerExit Event");
}
```

   The preceding code will show the logon events in the console window: **OnTriggerEnter** and **OnTriggerExit**.

8. Let’s move to the next step. Create a new script and save it as `RigidbodyController.js`. Add it to the **Red Cube** game object, as shown in the following screenshot:



Add the following code to the `RigidbodyController.js` script file:

```
#pragma strict
@scriptRequireComponent(Rigidbody)
/******* RequirementComponent ensure there should be a Rigid body as it
asks for Rigidbody as required component/
privatevaronPlane = false;
// Multiply the velocity when using the velocity.
varvelocityMultiplier = 100.0f;
// Multiply the move distance.
varmoveMultiplier= 8.0f;
// Multiply the force to make the cube jump.
varjumpMultiplier= 700.0f;


functionFixedUpdate(){
// Calculate the velocity/move direction based on the user input.
  varmoveDirection = Vector3( Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical") ) * Time.deltaTime;
moveDirection = Camera.main.transform.TransformDirection( moveDirection );
// Here TransformDirection transforms direction from local space
// to world space.

  if ( rigidbody != null && !rigidbody.isKinematic )
    {
    rigidbody.MovePosition( rigidbody.position + ( moveDirection *
moveMultiplier ) );
rigidbody.velocity = Vector3( 0, rigidbody.velocity.y, 0 );

  }

if ( Input.GetButtonDown("Jump") &&onPlane )
  {

    rigidbody.AddForce( Vector3.up * jumpMultiplier );
     }

}



functionOnCollisionStay(collisionInfo : Collision)
{
  onPlane = true;
}
```

```
functionOnCollisionExit( collisionInfo : Collision )
{
onPlane = false;

}
```

Run the project and you will see the Rigidbody actions on the red cube by clicking on the **Jump** button.

In the preceding example, we have learned how to apply actions on a Rigidbody using scripts.

Summary

In this chapter, we have learned about the types of Rigidbodies and their properties. We have seen the properties and uses of the Physics Rigidbody and Is Kinematic Rigidbody. We have learned how to implement actions on them using scripts. In the next chapter, we will learn about the types of joints and their properties. We will also see how we can implement joints on the game objects using different examples.

Chapter 5. Joint Types and Their Properties

In this chapter, we will learn about types of joints components and their properties. Joints are important components of Unity3D and provide different types of joints. We will learn about all these joints with interesting examples, that is, by creating a door animation with a hinge joint, a ball’s spring movement using a spring joint, and bone movement using character joints.

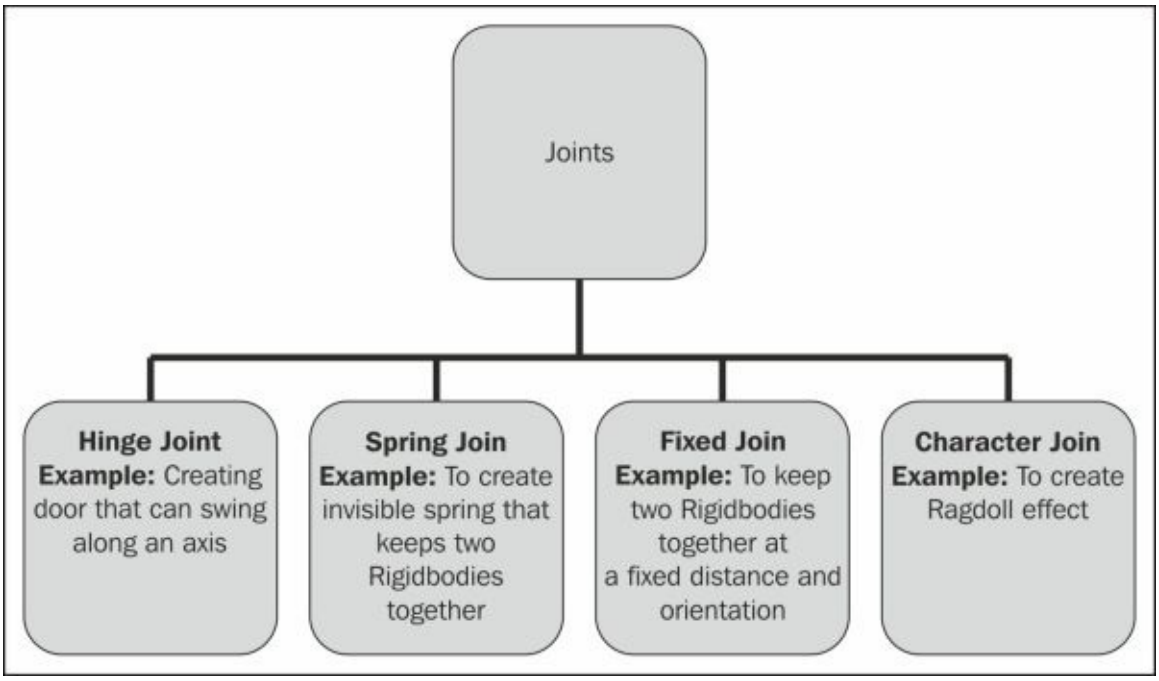
In this chapter, we will learn more about the following joints:

- Types of joints
- Configurable joints
- Handling movement and motion of configurable joints

Now, we will learn about different types of joints.

Types of joints

Joints are one of the most important components of Unity3D. In this chapter, we will learn more about joints.

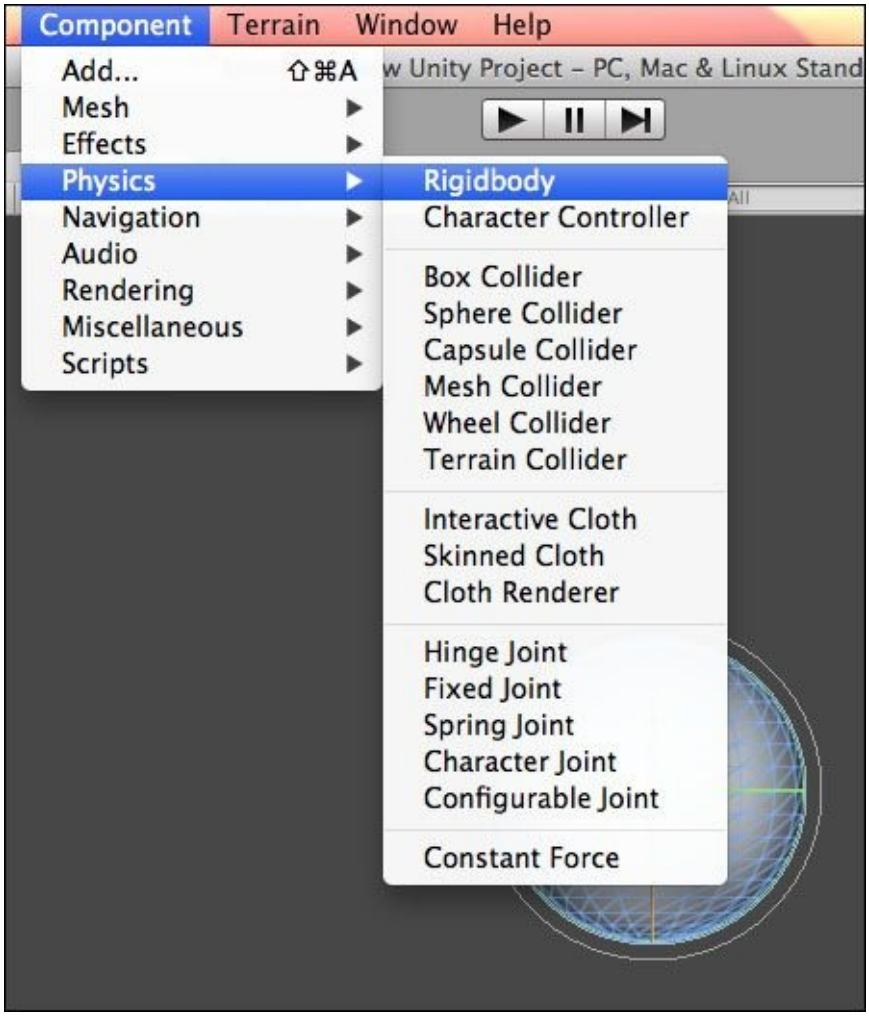


Apart from the previously mentioned joints in the preceding diagram, there is also a configurable joint that allows you to create complex joint configuration using different joints. Let’s take a look at the different types of joints with examples.

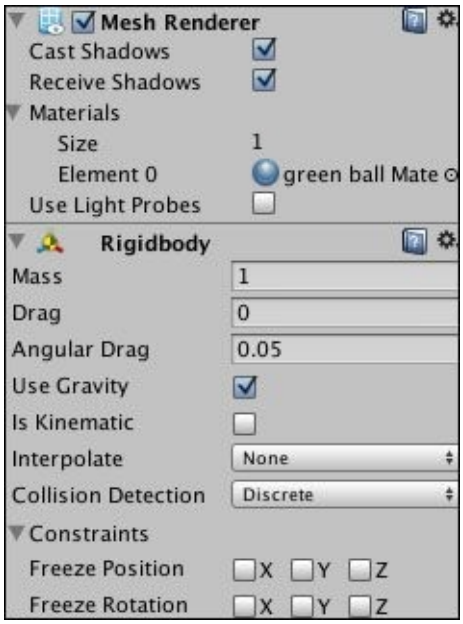
Fixed joint

The fixed joint is used to group two Rigidbodies together in their bound position. We use a fixed joint to parent the object in the hierarchy. By using this joint, we can lock the game object in the world. In this example, we are going to learn how we can implement a fixed joint. Let's get started:

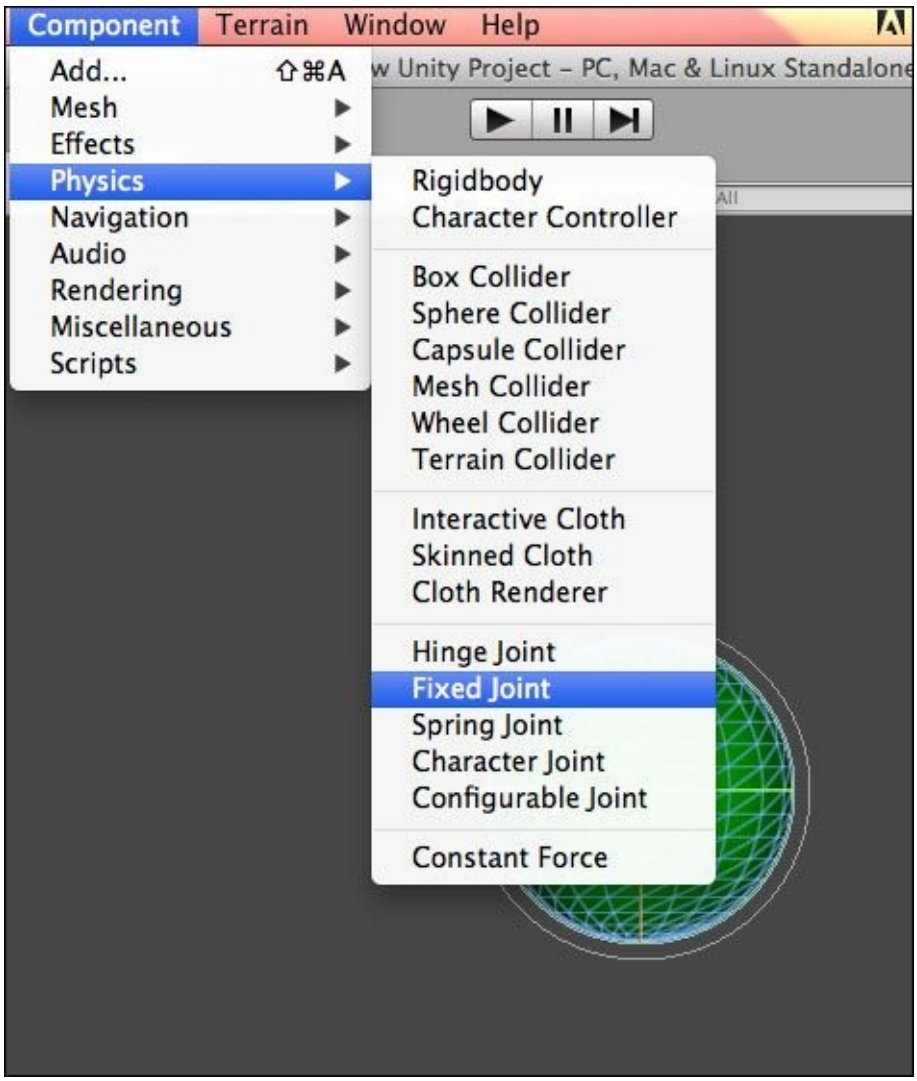
1. Create a new scene and save it as `Fixed Joint Example`.
2. As shown in the following screenshot, create a **Sphere** game object and assign a **Rigidbody** component to it.



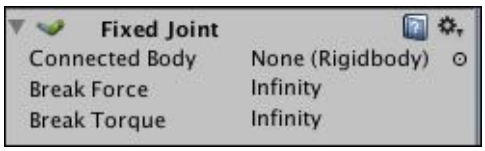
3. As you can see in the following screenshot, apply the **green ball Material** option and the **Use Gravity** option from **Rigidbody** for the sphere game object.



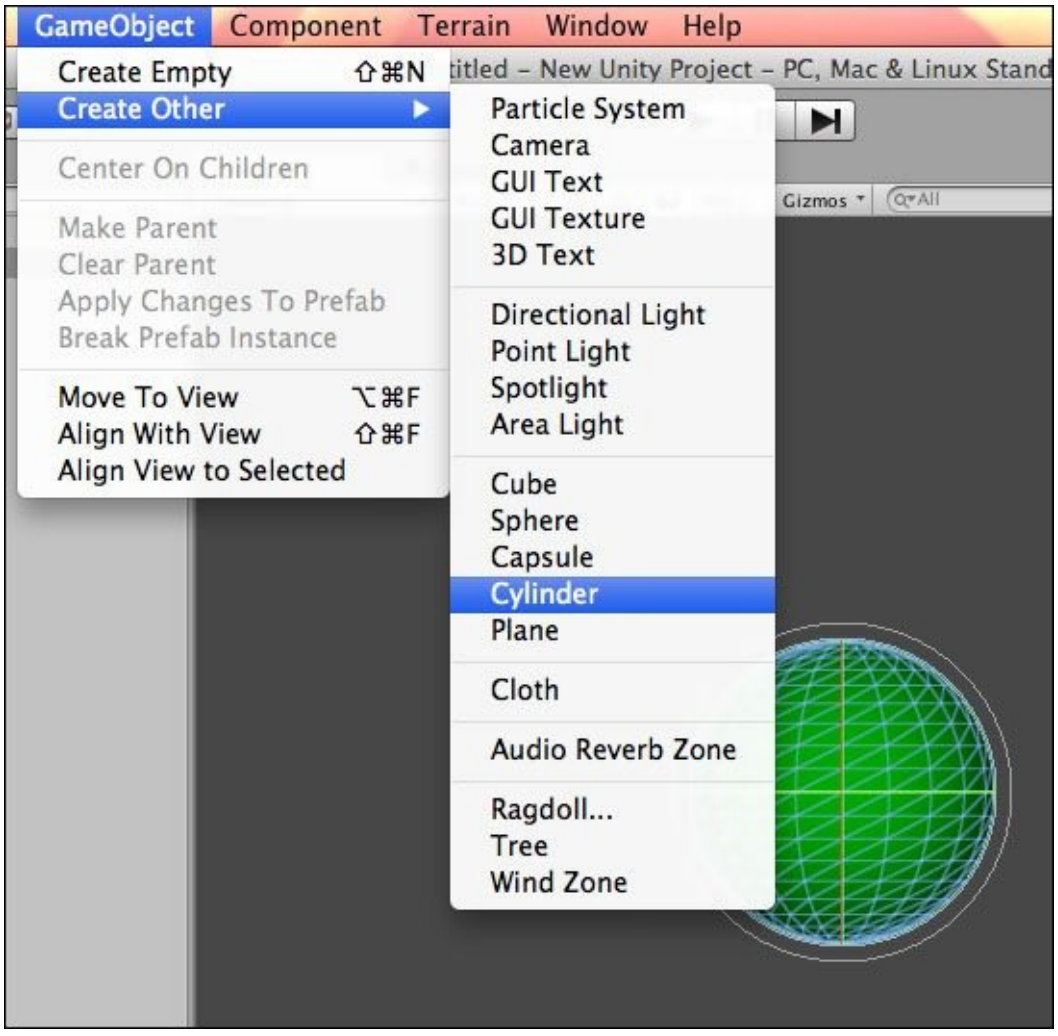
4. To apply a fixed joint as shown in the following screenshot, click on **Component** and select **Physics**. From the **Physics** menu, select **Fixed Joint**.



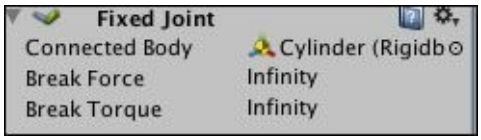
5. After applying **Fixed Joint**, look at the **Inspector** panel; you will see the options with the components, as shown in the following screenshot:



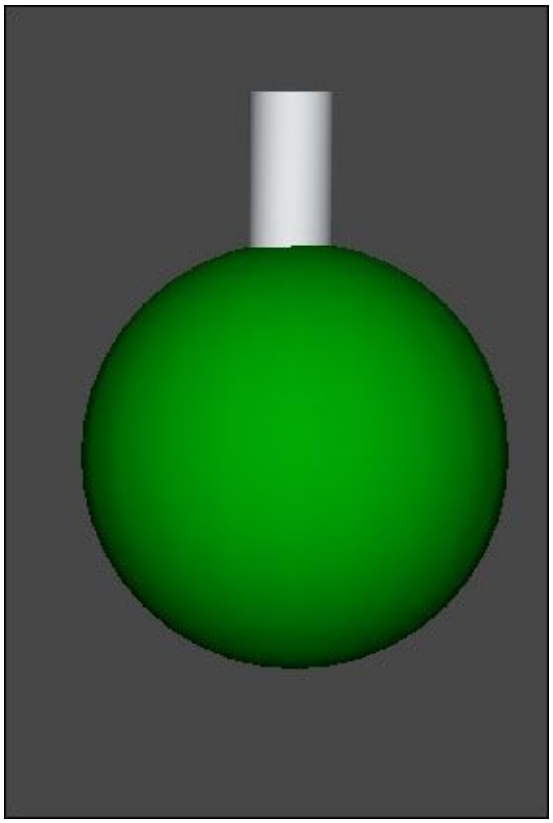
6. **Connected Body** refers to the other Rigidbody on which **Fixed Joint** is dependent upon. Now we need to create a connector for it. Let's create a **Cylinder** game object that will work as a connector to the sphere. As shown in the following screenshot, click on **GameObject** and then select **Create Other** to create the **Cylinder** game object:



7. Now drag **Cylinder** to **Connected Body**. This will create the connector for the **Sphere** game object. The other options **Break Force** and **Break Torque** are required amount of force and torque which can break the joint. For now, we have set it to **Infinity** to make it nonbreakable.



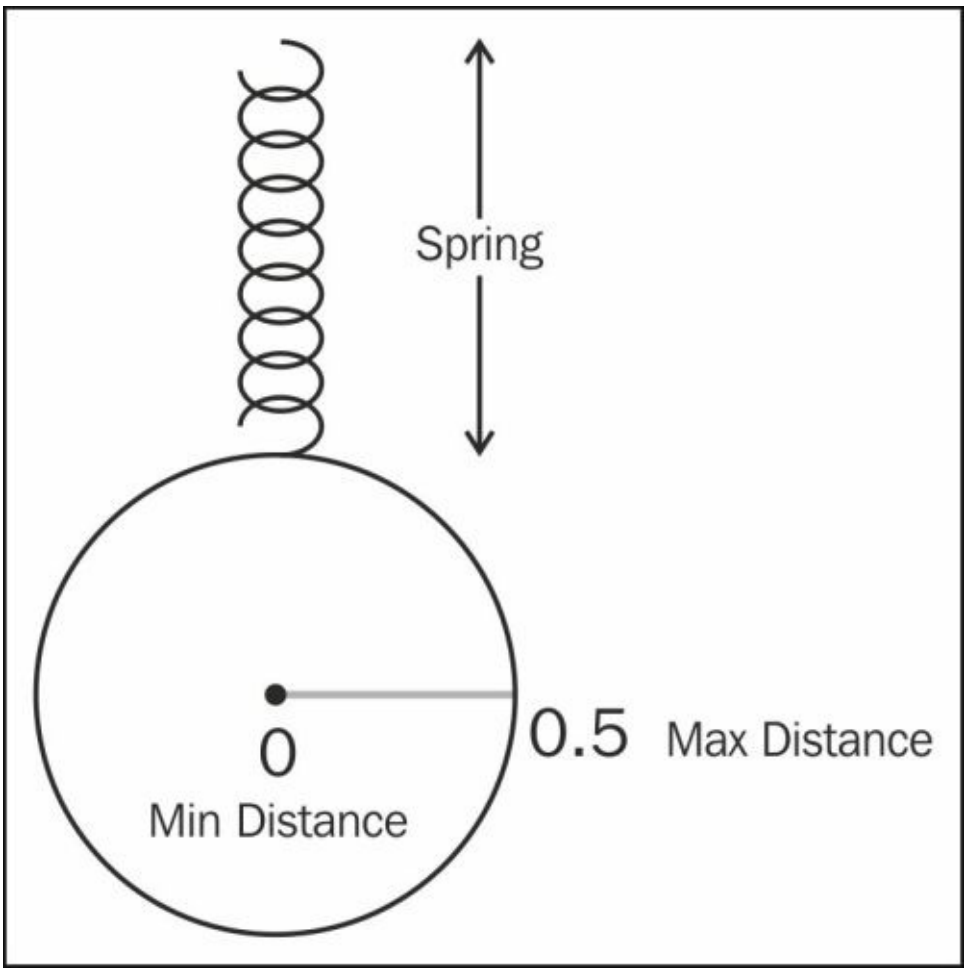
8. Run the project and you will see that the sphere falls with the cylinder because it is fixed to the cylinder.



In the preceding example, we have learned how we can implement the fixed joints in development.

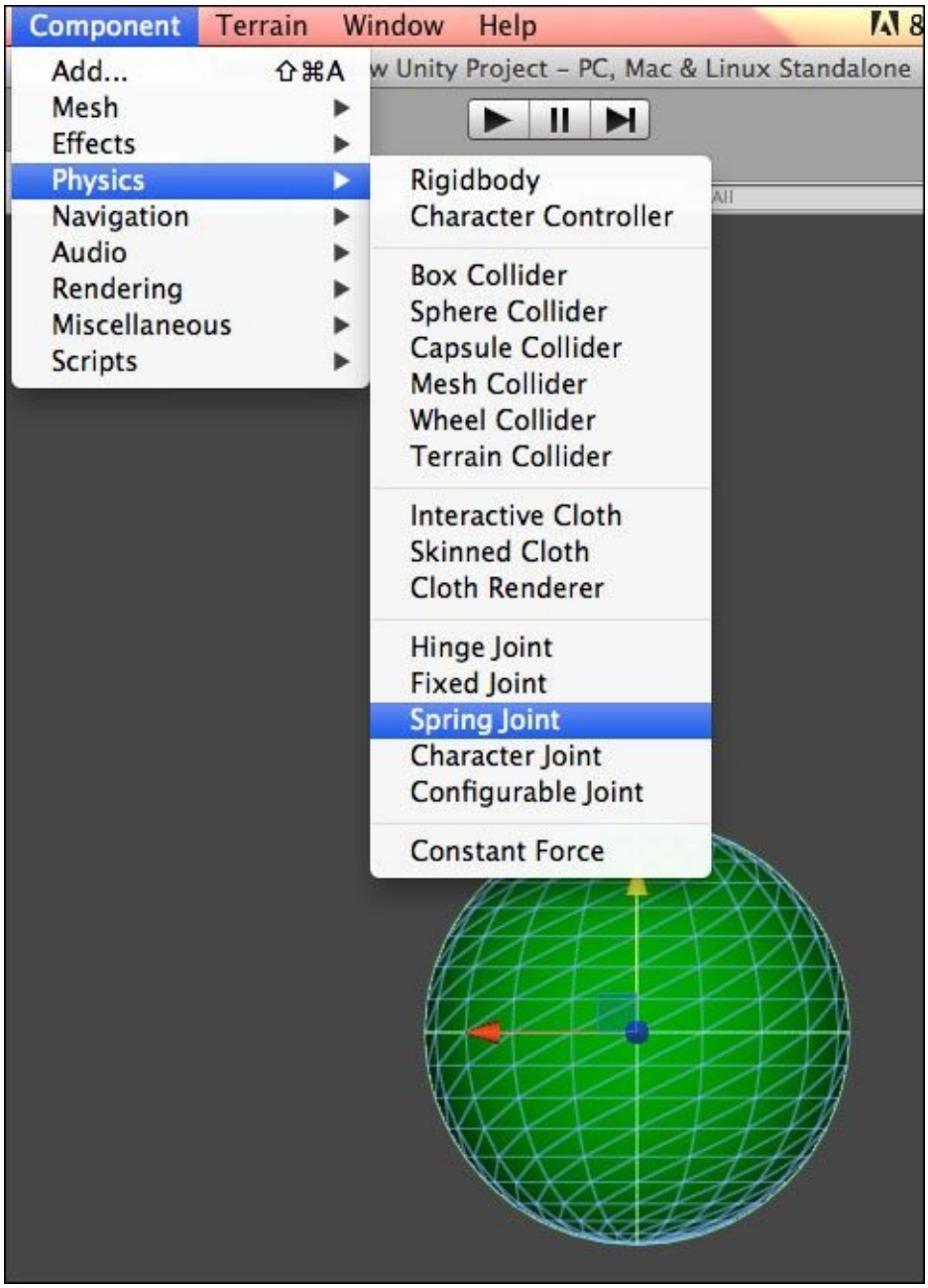
Spring joint

Let’s move to the next joint, which is a spring joint. A spring joint makes the Rigidbodies move like a spring. The applied game object gets pulled towards a particular position because of this joint. Spring joints, as their name implies, work on the principle of a spring where the specified game object tries to reach a target position, which we set in the scene view, while the attached Rigidbody will pull it away from the target position. As shown in the following diagram, if we apply a spring joint on a sphere, it will show the behavior of a spring and try to move toward the particular position. Here, **Min Distance** will be the center of sphere while end of radius will work as **Max Distance**. We can modify the value of minimum and maximum distance as per our requirements. The working of this joint is shown diagrammatically, as follows:

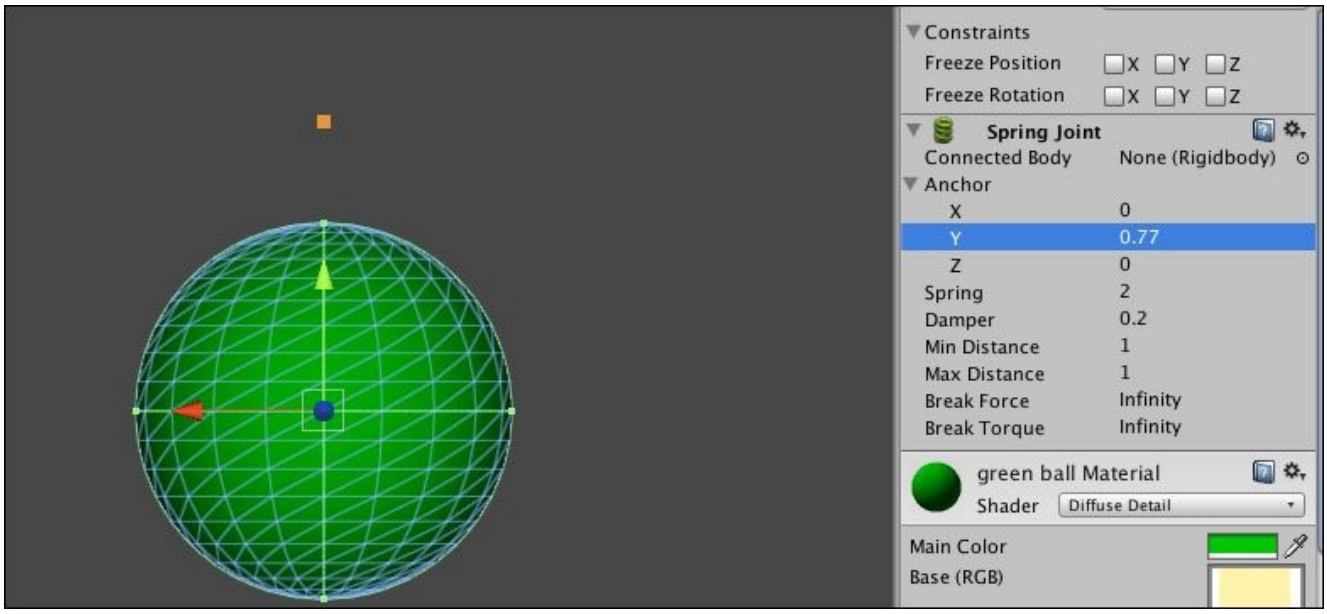


In the following example, we are going to create a sphere that will have a spring joint:

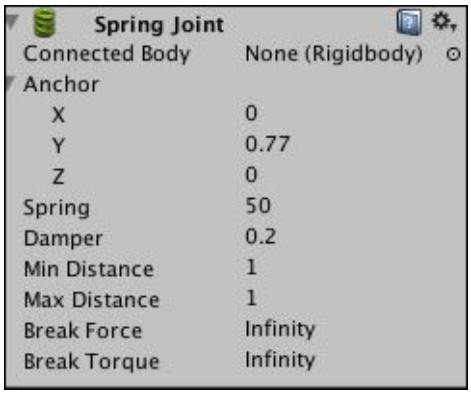
1. As shown in the following screenshot, let’s create a sphere with a Rigidbody and apply **Spring Joint**:



2. As shown in the following screenshot, we can set an **Anchor** value. An orange square above the sphere represents the anchor. Let's reset the **Y** value and run the project. You will see the sphere's movement with the spring.



3. We can set the number of springs by putting values for the **Spring** option. We have set it at `50` to increase the bounciness. Similar to other joints, **Break Force** and **Break Torque** are the force and torque required to break the joint.

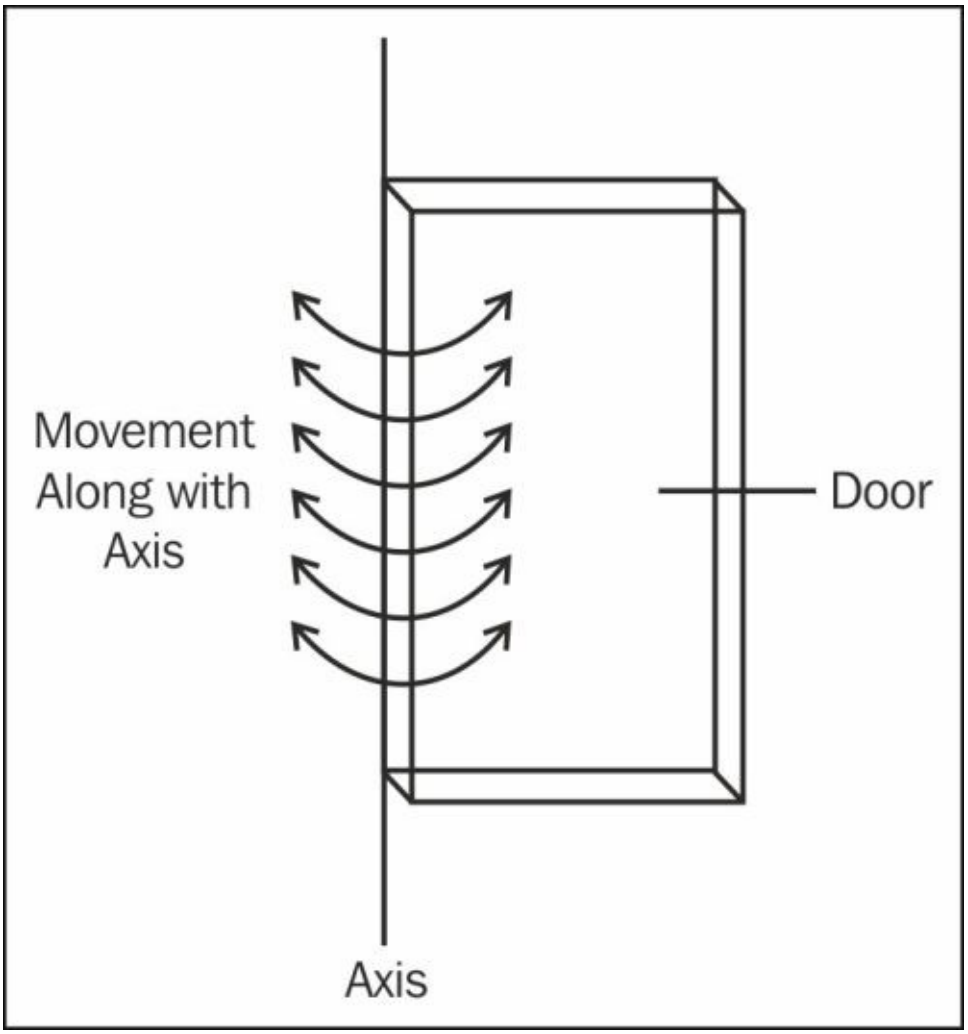


In the preceding example, we can see the different properties of a spring joint and their uses.

Hinge joint

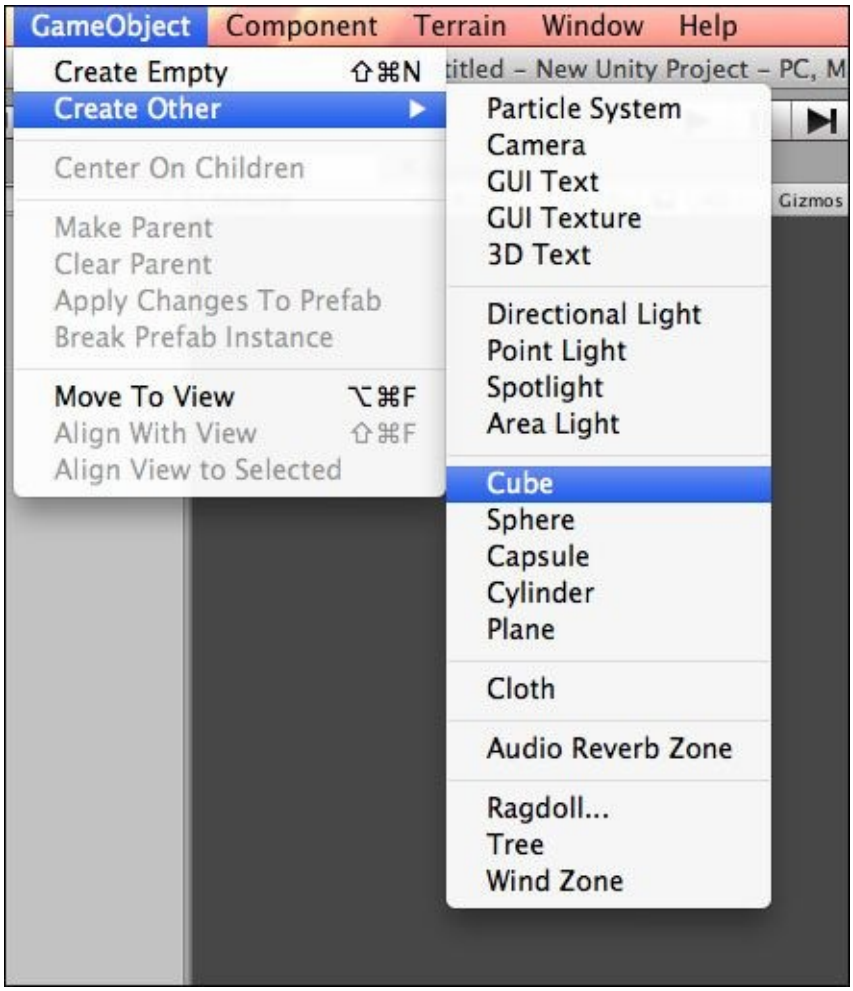
We have learned about a fixed joint and a spring joint. Now, let’s learn about a hinge joint.

Normally, hinge joints are used to create a swinging door and a wrecking ball. The hinge joints move along with the axis, as shown in the following diagram. If we apply a hinge joint on a door, it will move along with the axis.

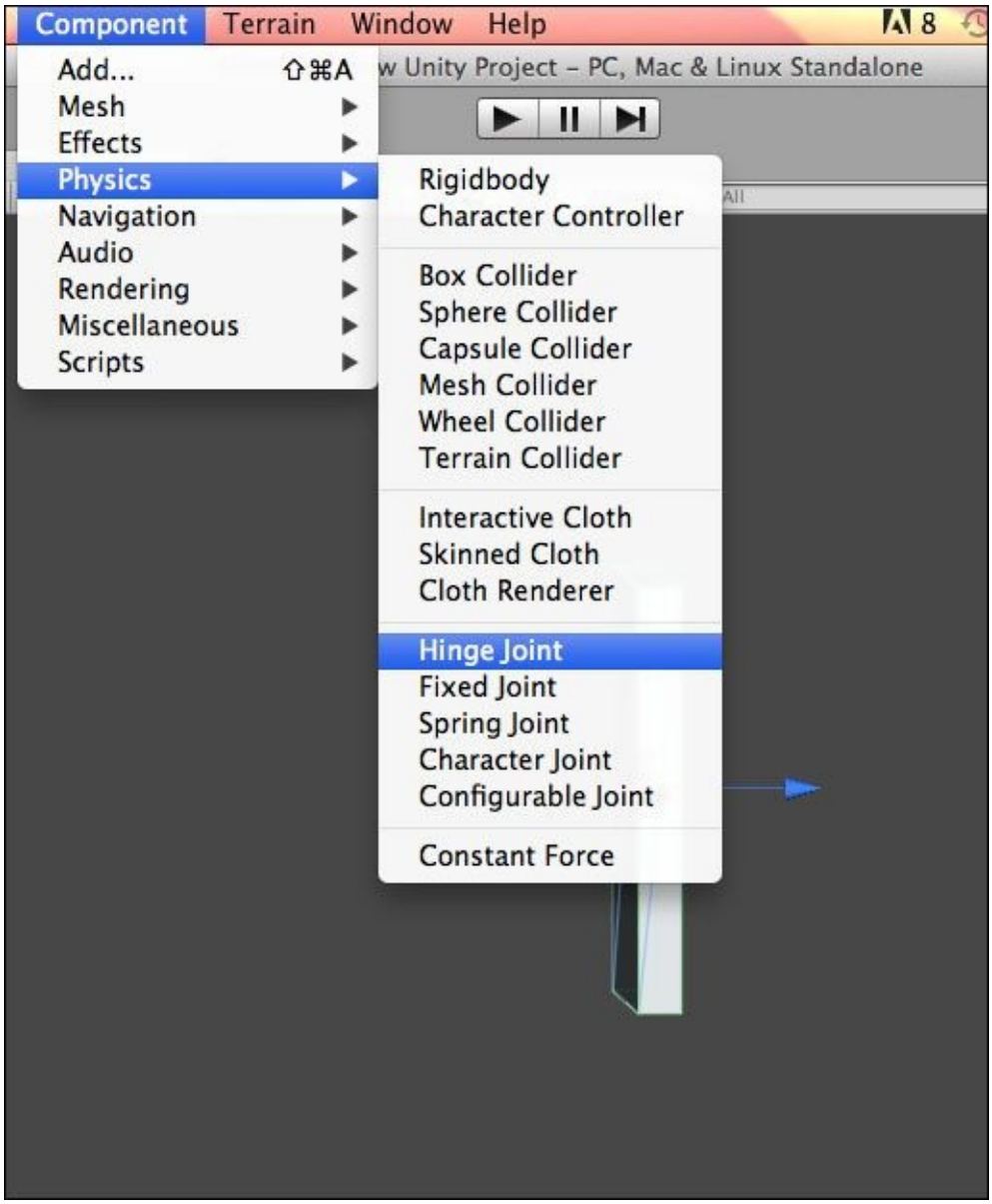


Let’s learn to apply a hinge joint to a game object and also see which properties are associated with hinge joints:

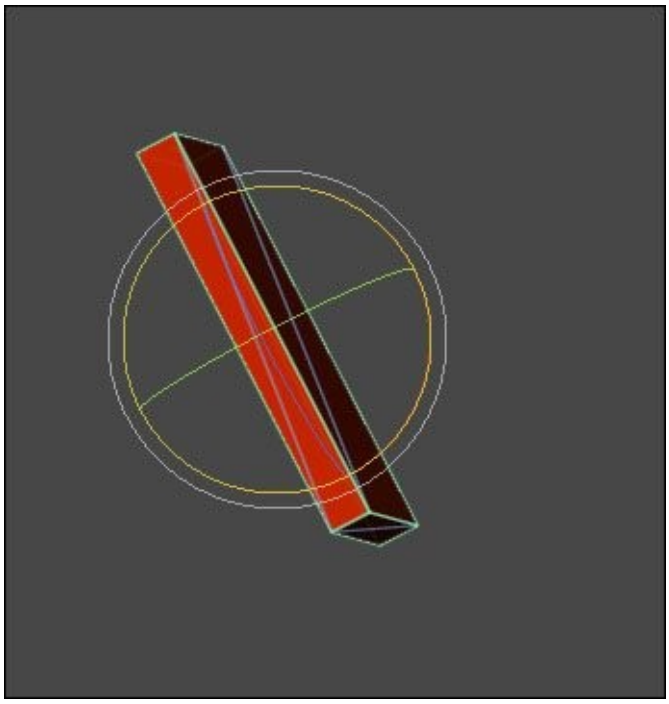
1. Create a new scene and save it as `Hinge Joint`.
2. Create a **Cube** game object and apply a Rigidbody component to it. Scale it to give it the shape of a door.



3. Now apply **Hinge Joint**, as shown in the following screenshot:



4. After applying material to it, give it a color, rotate it, and run and test the movement of the door.



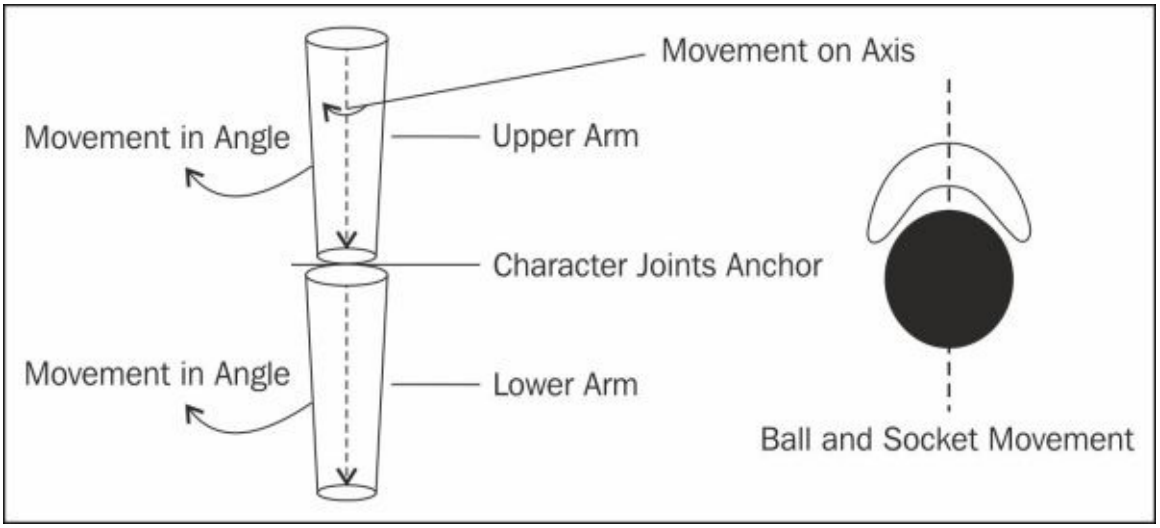
The hinge joint possesses several properties as options. We can use the properties of **Hinge Joint** as per our requirements. Here, **Connected Body** refers to the other Rigidbody with which the joint is dependent. **Anchor** refers to the point at which the connected bodies can rotate. **Axis** defines the axis of rotation. To apply spring force to a target angle, we apply **Spring**. To slow the spring recoil, we set the **Damper** value; **Motor** to apply motor force; and **Limits** to specify the angular limit.

Character joints

The character joints work on the principle of a ball and socket movement with limit, which are normally used for Ragdoll effects. This consists of variables that handle the effects or movement.

Note

Ragdoll Physics is a type of procedural animation that is often used in video games and animated films. You can find details of using Ragdoll Wizard of Unity3D at http://docs.unity3d.com/Manual/wizard-RagdollWizard.html.

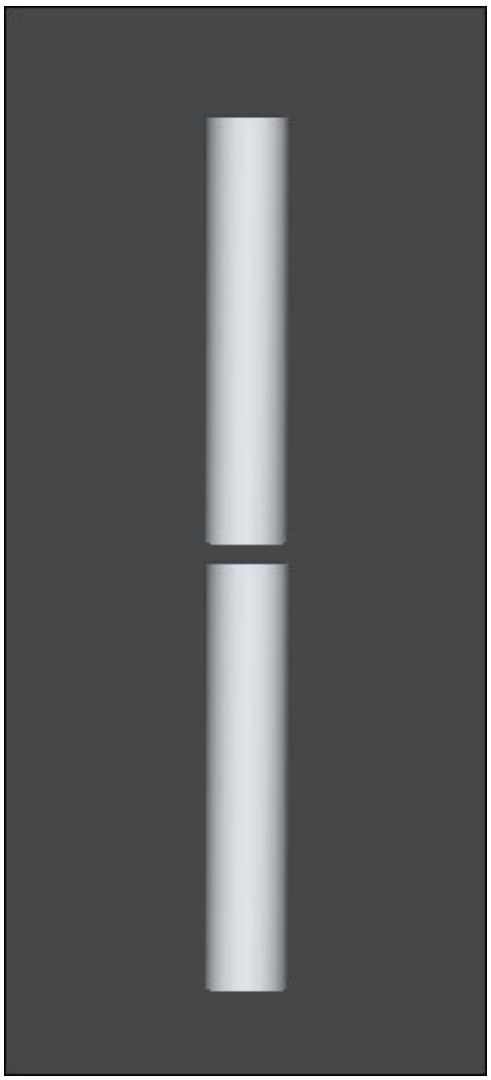


We are going to create a character joint for an arm. There are two types of movement, as shown in the previous diagram. We divide this arm in two parts—one is **Upper Arm** and the second is **Lower Arm**. We will see that there are two types of movement in this case —one is movement with the axis and the other is movement in angle or we can say swinging movement.

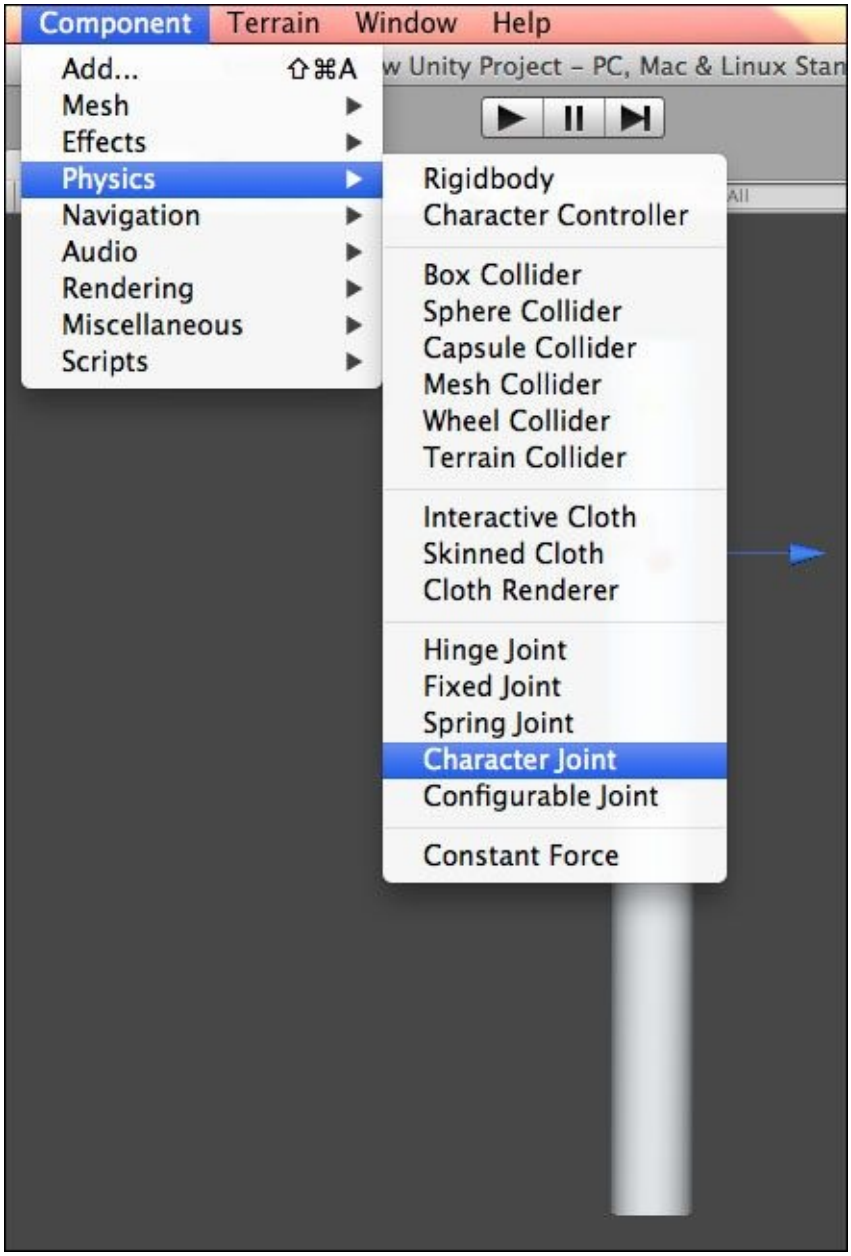
Let’s handle the arm animation with Unity Physics character joint:

1. Create a scene and name it `Character Joint example`.
2. Now create an empty game object and name it `Arm`:

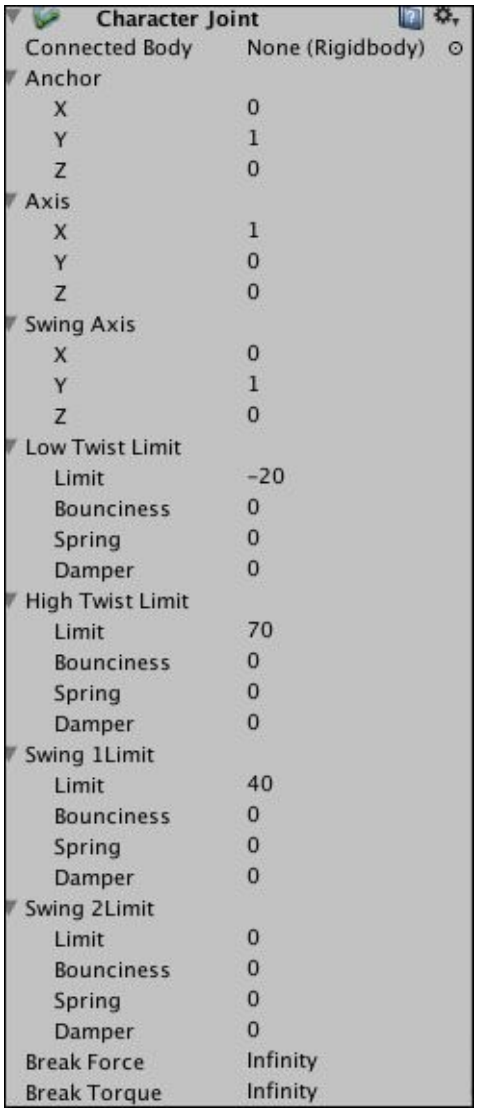
3. Now, create a **Cylinder** object and apply **Rigidbody** on it. Name it `Upper Arm`.
4. Click on **Component** and apply **Character Joint** on it.
5. Now, create another **Cylinder** object and again apply Rigidbody on it. Name it `Lower arm`:



6. Click on **Component** and then apply **Character Joint** on it:



7. Now let’s play with the properties of character joints to get the required effect:



8. By default, **Connected Body** connects to the world. It represents the Rigidbody on which the joint is dependent. Let's drag the **Upper arm** Rigidbody to the connected body option of **Lower arm**.

In the following screenshot, **Upper arm** is **Connected Body** for the lower arm:

Anchor is where a joint rotates around. **Axis** which have twist axis and swing axis; using twist axis, we handle twisting while using swing axis, we handle swinging. **Low Twist Limit** is for the lower limit of the joint and similarly, **High Twist Joint** is

for the higher limit of the joint. Again, **Swing 1Limit** is for the lower limit around the swing axis and **Swing 2Limit** is for the upper limit around the swing axis.

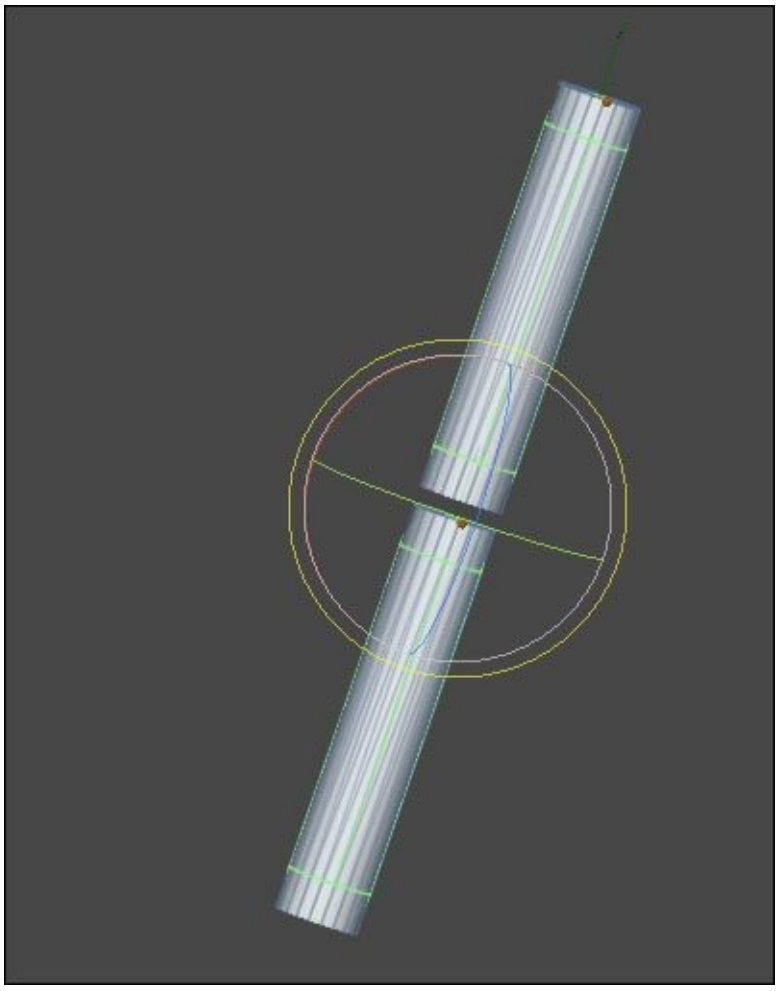
Note

The value for **Low Twist Limit** is -30 and for **High Twist Limit**, it is 60.

The **Swing 1Limit** limits the rotation between -30 and 30.

The **Swing 2Limit** will limit the rotation around that axis between -40 and 40 degrees.

9. Rotate the **Arm** game object and test the scene.



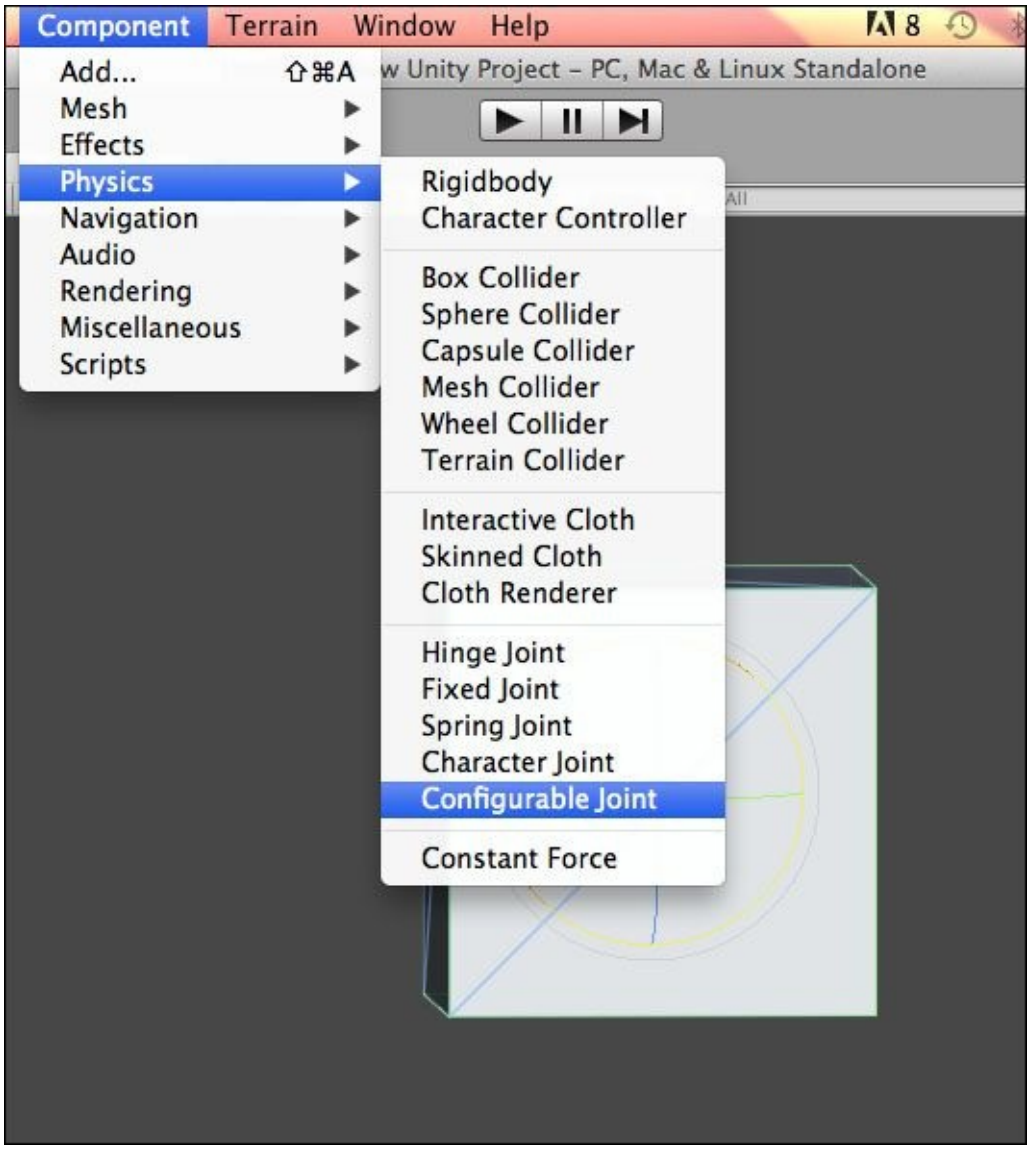
In the previous example, we have learned how we can use the character joint.

Configurable joints

Configurable joint possesses features of all the joints. It works on the principle of two primary functions which are movement/rotation restriction and movement/rotation acceleration. We can see that there are multiple interdependent properties in the configurable joints. To get the required effect, we will need to play with the values of its different properties.

Let’s see what do we mean by movement/rotation restriction. For this, perform the following steps:

1. Create a new scene and name it `Configurable Joints Example`.
2. As shown in the following screenshot, create a **Cube** game object and apply **Configurable Joint**:



3. In the **Inspector** panel, we will see the following properties:

Using the previously shown different properties, we can achieve the desired effect for your project. Now, let's see how we can handle the movement and rotation of different properties in the project.

Handling movement/rotation restriction

We can restrict movement as per the axis. **XMotion**, **YMotion**, and **ZMotion** allow you to do that and while using **Angular XMotion**, **Angular YMotion**, and **Angular ZMotion**, we can define the rotation. These properties can be set to **Free** which means unrestricted, **Limited** which means restricted based on limits, or **Locked** which means restricted to zero movement.

Limiting motions

We can use the Limit properties to restrict the motion. Using **Linear Limit**, we can define the maximum distance from the origin point. Using **Bounciness**, **Spring**, and **Damper**, we can define the behavior of the object when it reaches the limit on any of the limited motion axes. To stop the object's movement on the border, we set all these values to `0`. Using bounciness, we set the bounce back behavior of the object from the border similarly. **Spring** and **Damper** will create springing forces to apply the spring effect border.

Limiting rotation

We can use the Limit properties to restrict the rotation. Limiting rotation works almost the same as limiting motion but the differences are the **Angular Limit** properties. We can restrict translation along all three axes by defining the **Linear Limit** property, and we can also restrict rotation along each of the three axes by defining the **Angular Limit** property per axis.

Using the **Angular XMotion** limitation, we can define a **Low Angular XLimit** and a **High Angular XLimit**. For the *y* and *z* axes, the low and high rotation limits will be the same, which we set using the Limit property of **Angular YLimit** or **Angular ZLimit**.

Handling movement/rotation acceleration

Using drive properties, we define the acceleration by defining the **Target** value. To specify object movement or rotation in terms of moving the object toward a particular position or rotation, we need to define the **Target** value to move toward, and using a drive to provide acceleration that will move the object toward that target.

Handling translation acceleration

Using the **XDrive**, **YDrive**, and **ZDrive** properties, we define movement along that axis. **Spring** value defines the object's motion toward the **Target** position. Similarly, if we are using velocity in its mode, its maximum force value defines acceleration toward the velocity.

Handling rotation acceleration

Rotation acceleration properties contain the **Angular XDrive**, **Angular YZDrive**, and **Slerp Drive** function similar to the translation **Drives**. There is one small difference. **Slerp Drive** behaves differently. It has different functionality than the **Angular Drive** functionality so we cannot use both together.

Summary

In this chapter, we have learned about joints and their properties. We have learned how we can apply a fixed joint to fix two Rigidbodies together, a spring joint for spring-like behavior, a hinge joint to create door animation, and a character joint for a Ragdoll effect.

In the next chapter, we will learn about animation and Unity3D Physics. It will focus on the different animations, for example, rope animation, use of add force and add torque, and constant force.

Chapter 6. Animation and Unity3D Physics

In this chapter, you will learn to use Physics in animation creation. We will see that there are several animations that can be easily handled by Unity3D's Physics. During development, you will come to know that working with animations and Physics is easy in Unity3D. You will find the combination of Physics and animation very interesting.

We are going to cover the following topics:

- Interpolate and Extrapolate
- The Cloth component and its uses in animation
- ConstantForce
- AddForce
- AddTorque
- An example of creating a rope animation using different joints

Developing simple and complex animations

As mentioned earlier, you will learn how to handle and create simple and complex animations using Physics, for example, creating a rope animation and hanging ball. Let’s start with the Physics properties of a Rigidbody component, which help in syncing animation.

Interpolate and Extrapolate

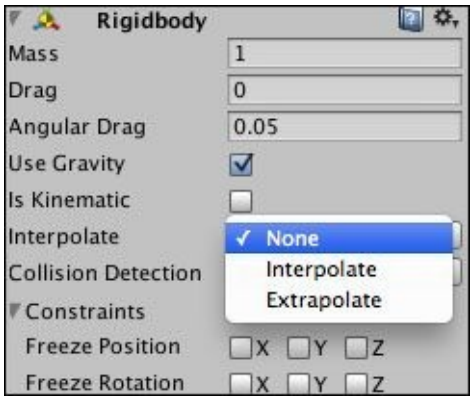
Unity3D offers a way that its Rigidbody component can help in the syncing of animation. Using the interpolation and extrapolation properties, we sync animations.

Note

Interpolation is not only for animation, it also works with Rigidbody.

Let’s see in detail how interpolation and extrapolation work:

1. Create a new scene and save it.
2. Create a **Cube** game object and apply **Rigidbody** on it.
3. Look at the **Inspector** panel shown in the following screenshot. On clicking **Interpolate**, a drop-down list that contains three options will appear, which are **None**, **Interpolate**, and **Extrapolate**. For details, refer to the previous chapter. By selecting any one of them, we can apply the feature.



In interpolation, the position of an object is calculated by the current update time, moving it backwards one Physics update delta time.

Note

Delta time or **delta timing** is a concept used among programmers in relation to frame rate and time. For more details, check out http://docs.unity3d.com/ScriptReference/Time-deltaTime.html.

If you look closely, you will observe that there are at least two Physics updates, which are as follows:

- Ahead of the chosen time
- Behind the chosen time

Unity interpolates between these two updates to get a position for the update position. So, we can say that the interpolation is actually lagging behind one Physics update.

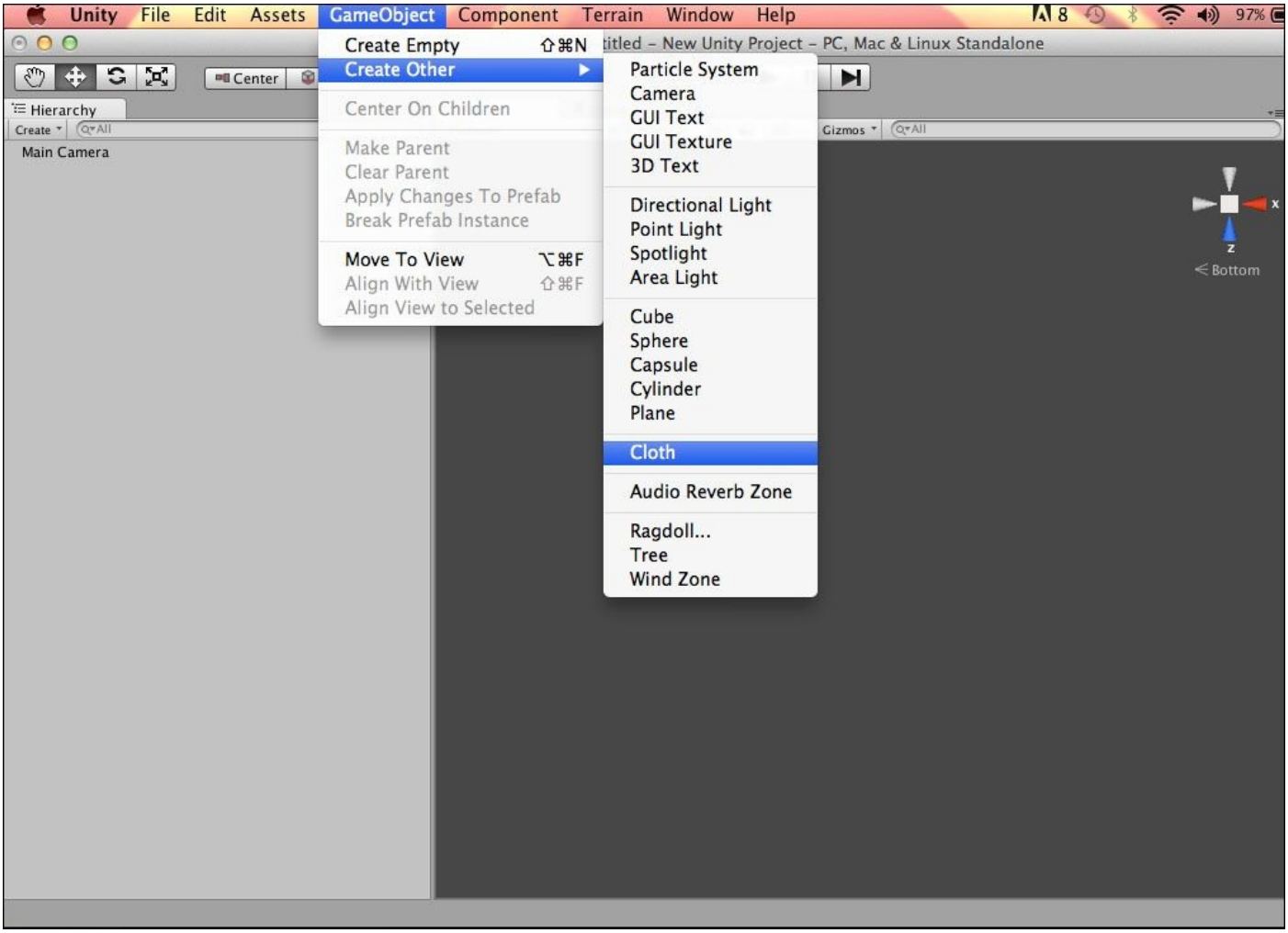
The second option is **Extrapolate**, which is to use for extrapolation. In this case, Unity predicts the future position for the object. Although this does not show any lag, incorrect

prediction sometime causes a visual jitter.

One more important component that is widely used to animate cloth is the **Cloth component**. Here, you will learn about its properties and how to use it.

The Cloth component

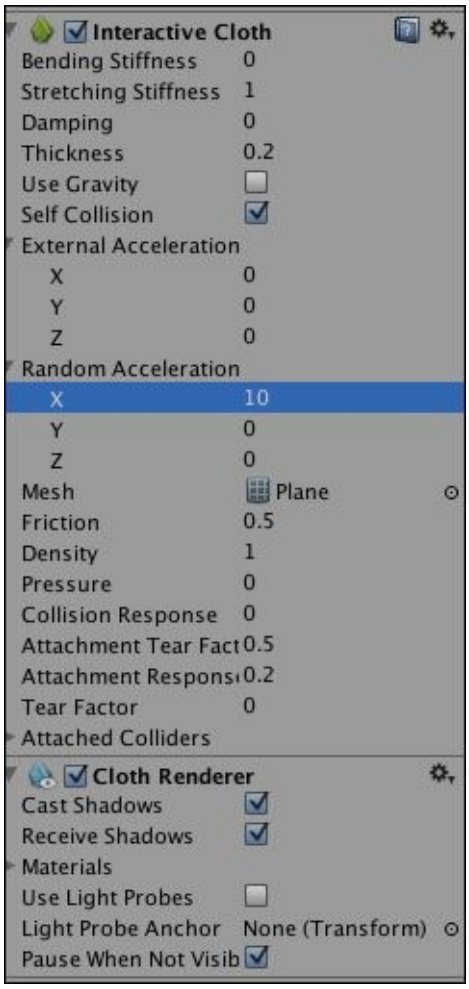
To make animation easy, Unity provides an interactive component called **Cloth**. In the **GameObject** menu, you can directly create the **Cloth** game object. Have a look at the following screenshot:



Unity also provides Cloth components in its Physics sections. To apply this, let's look at an example:

1. Create a new scene and save it.
2. Create a **Plane** game object. (We can also create a **Cloth** game object.)
3. Navigate to **Component** | **Physics** and choose **InteractiveCloth**.

   As shown in the following screenshot, you will see the following properties in the **Inspector** panel:



Let’s have a look at the properties one by one. **Blending Stiffness** and **Stretching Stiffness** define the blending and stretching stiffness of the Cloth while **Damping** defines the damp motion of the Cloth. Using the **Thickness** property, we decide the thickness of the Cloth, which ranges from 0.001 to 10,000. If we enable the **Use Gravity** property, it will affect the Cloth simulation. Similarly, if we enable **Self Collision**, it allows the Cloth to collide with itself. For a constant or random acceleration, we apply the **External Acceleration** and **Random Acceleration** properties, respectively.

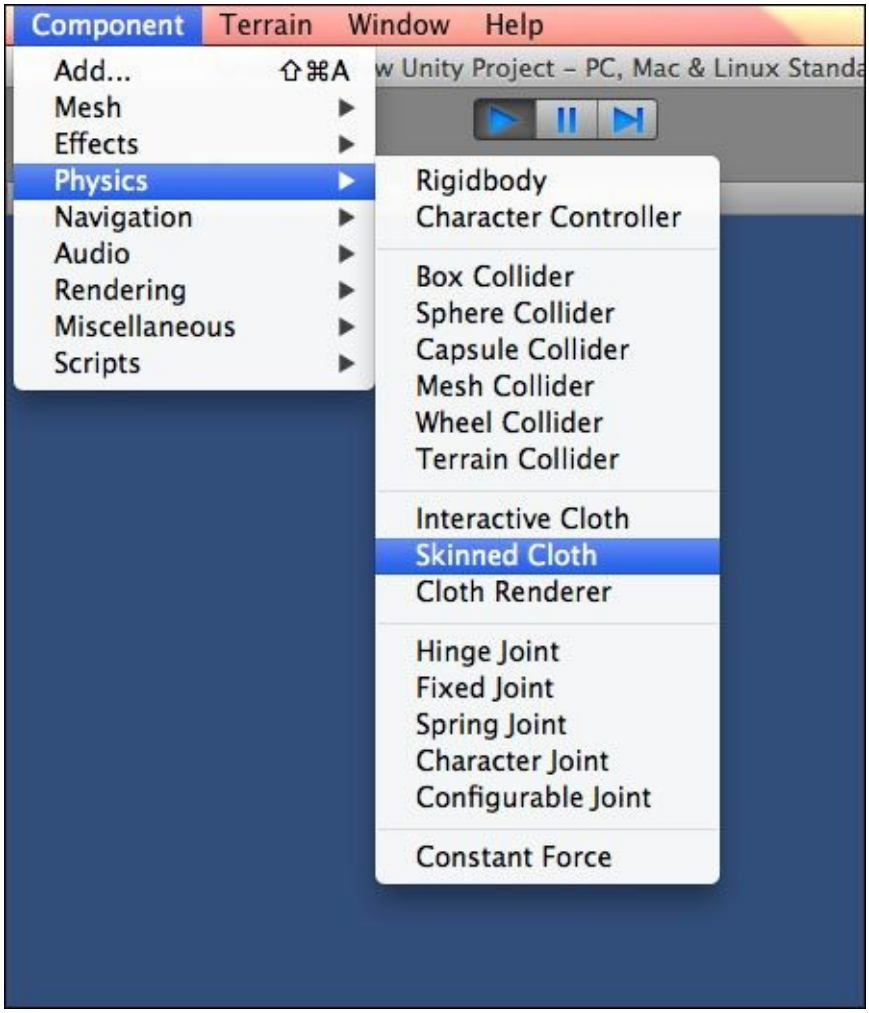
World Velocity Scale decides the movement of the character in the world, which will affect the Cloth vertices. The higher the value, the more movement of the character will affect. **World Acceleration** works similarly. The **Interactive Cloth** component depends on the **Cloth Renderer** components. Lots of Cloth components in a game reduces the performance of game. To simulate clothing in characters, we use the **Skinned Cloth** component.

Important points while using the Cloth component

The following are the important points to remember while using the Cloth component:

- Cloth simulation will not generate tangents. So, if you are using a tangent dependent shader, the lighting will look wrong for a Cloth component that has been moved from its initial position.

- We cannot directly change the transform of moving the Cloth game object. This is not supported.
- Disabling the Cloth before changing the transform is supported.
- The **SkinnedCloth** component works together with **SkinnedMeshRenderer** to simulate clothing on a character. As shown in the following screenshot, we can apply **Skinned Cloth**:



As you can see in the following screenshot, there are different properties that we can use to get the desired effect:

We can disable or enable the **Skinned Cloth** component at any time to turn it on or off.

ConstantForce

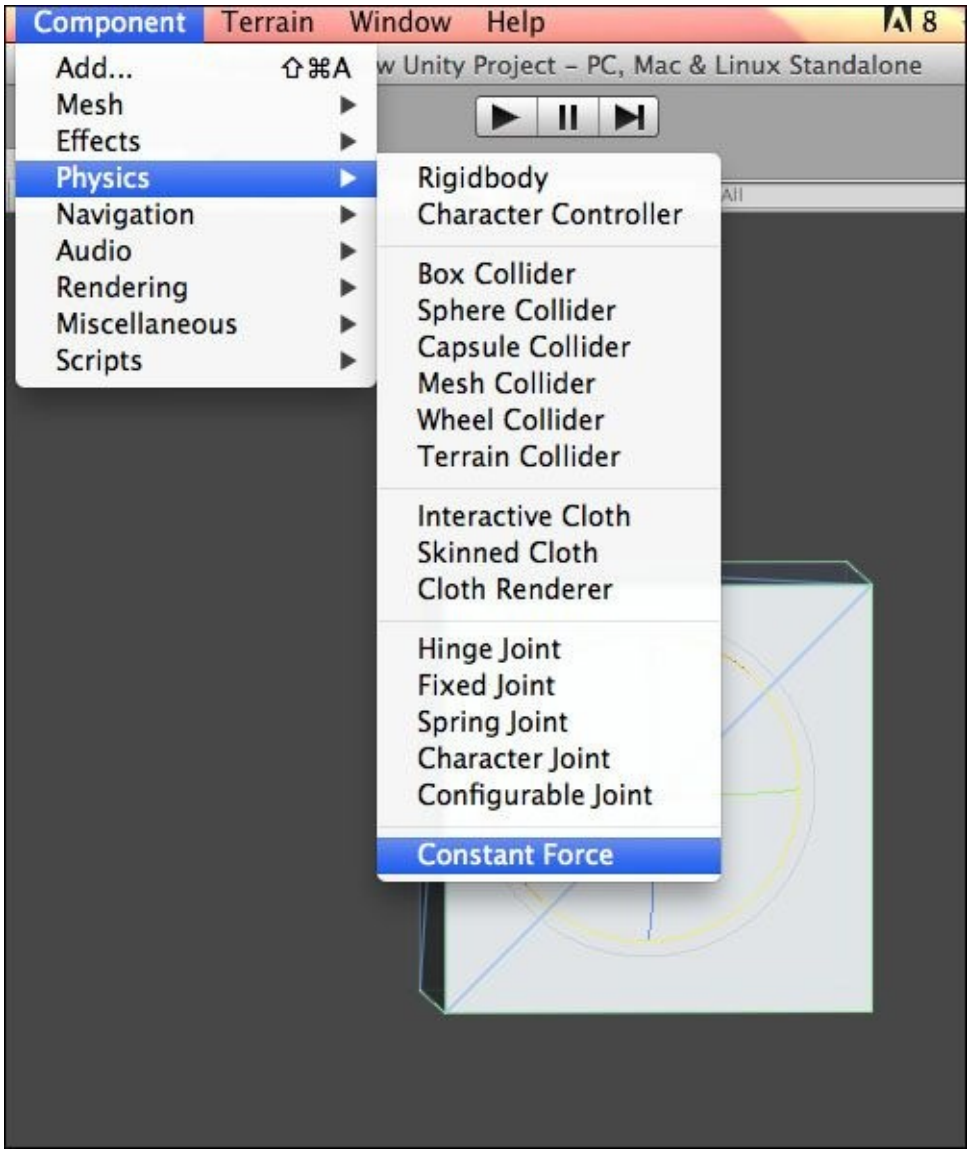
When a force is applied constantly on a game object, it is known as constant force. We use the `ConstantForce` Physics utility class to apply constant force on a game object. `AddForce` applies a force to the Rigidbody only for one frame whereas `ConstantForce` applies a force in every frame until we change the force or torque to a new value.

We use this for one-shot objects such as rockets.

An example of animation using ConstantForce

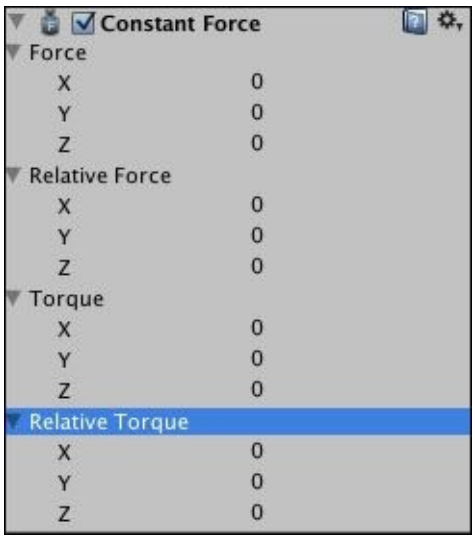
We will look at the example of animation using ConstantForce with the help of the following steps:

1. Create a new scene and save it as `Constant Force Example`.
2. Create a **Cube** game object.
3. As shown in the following screenshot, apply **Constant Force** on it:



4. To make a cube that accelerates forward, we set **Relative Force** to be along the positive *z* axis.

5. Then, use the Rigidbody's **Drag** property to set the maximum velocity and turn off gravity so that the game object will always stay on its path.



6. Run the scene and you will see that the cube game object moves constantly.

Till now, you have learned different Physics components. Now, you'll learn how we can use Physics in scripting. In this chapter, we will learn `AddForce` and `AddTorque`. By applying `AddForce` on a Rigidbody, we make it move while `AddTorque` adds a torque to the Rigidbody to make it spin around the torque axis.

An example of animation using AddForce

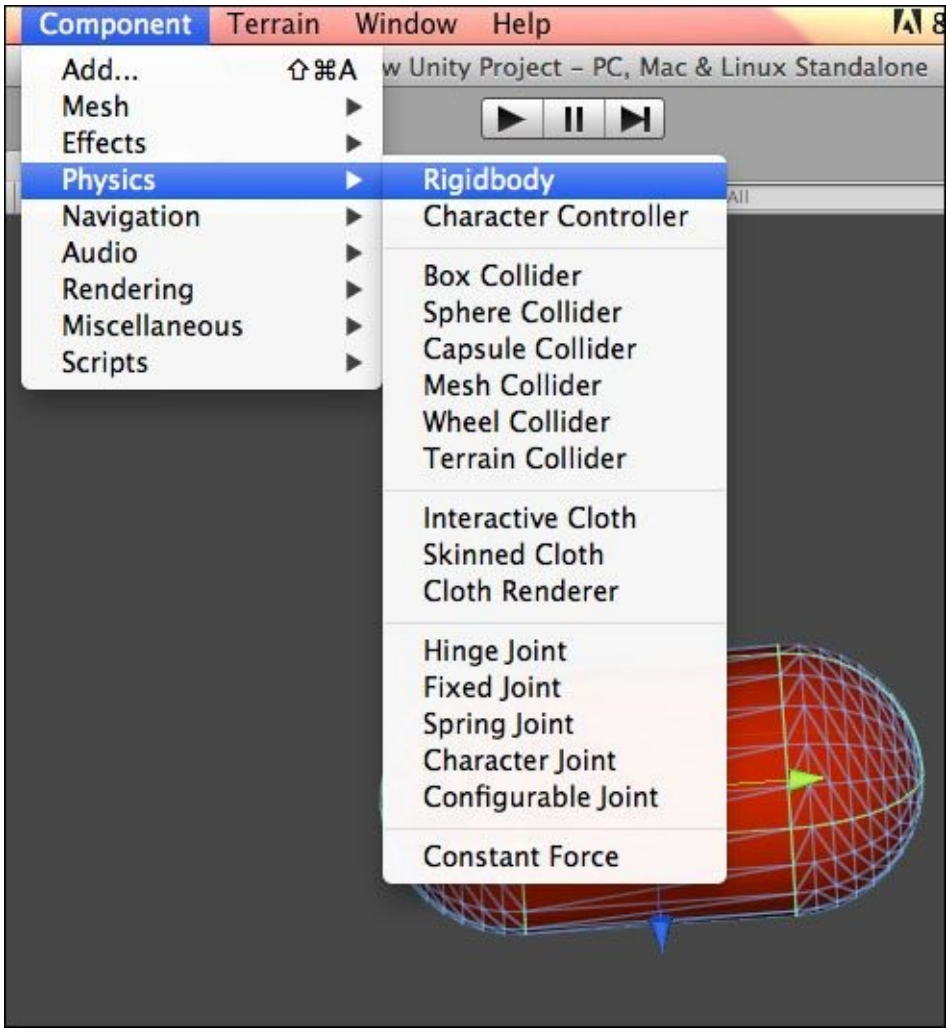
`AddForce` adds force to the Rigidbody.

Note

In Physics, a **force** is any interaction that tends to change the motion of an object. In other words, a force can cause an object with mass to change its velocity (which includes beginning moving from a state of rest), that is, to accelerate. Force can also be described by intuitive concepts such as a push or a pull.

In this example, we will create a projectile using `AddForce` Physics:

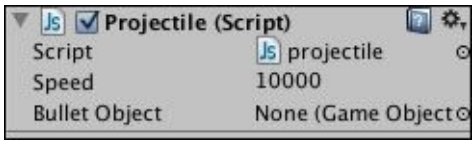
1. Create a new scene and save it as `addForce example`.
2. Use a **Plane** game object to create ground.
3. Apply a green material on it.
4. Create a **Capsule** object and apply **Rigidbody** on it as shown in the following screenshot. Now, name it `bullet`.



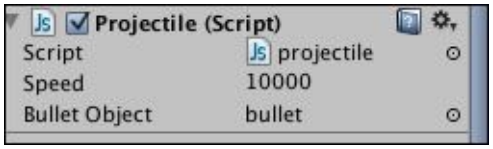
5. Create a new JavaScript or UnityScript and name it `projectile`. Write the following code inside it:

```
var speed:int=10000;
var bulletObject: GameObject;
private var hitPoint:Vector3;
private var hit : RaycastHit;
function Update ()
{
  var ray = Camera.main.ScreenPointToRay (Input.mousePosition);
  if (Input.GetMouseButtonDown (0))
  {
    if (Physics.Raycast (ray, hit, 300.0f))
    {
      if (hit.collider.tag == "ground")
      {
// Add "ground " tag to the plane game object.
        var bullet = Instantiate(bulletObject, Vector3(0,1.5,-12),
transform.rotation);
        bullet.rigidbody.AddForce( Vector3(0,0.5,0.5) * speed);
      }
    }
  }
}
```

6. As shown in the following screenshot, you will need to specify **Bullet Object**:



7. Drag the **bullet** game object and put it on the **Bullet Object** section as shown in the following screenshot:



8. Now, test the scene and you will see that on a click, the bullet is fired and it is moved in a projectile.

In this example, we saw how to use `AddForce` to create a projectile animation. Similarly, in the next example, we will see how to use `AddTorque` to create an animation.

An example of animation using AddTorque

`AddTorque` adds torque to the Rigidbody.

Note

Torque is a measure of the turning force on an object such as a bolt or a flywheel. For example, pushing or pulling the handle of a wrench connected to a nut or bolt produces a torque (turning force) that loosens or tightens the nut or bolt.

In this example, we will use `AddTorque` to create an animation:

1. Create a new scene and save it as `AddTorque example`.
2. Create a **Cube** game object and name it `box`.
3. Add **Rigidbody** on a **Box** game object.
4. Create a new JavaScript and use the following code:

```
// Spins the rigidbody around the global y-axis
  var box:GameObject;
  var speed:int=10;
  function FixedUpdate () {
    box.rigidbody.AddTorque (Vector3.up * speed);
  }
```

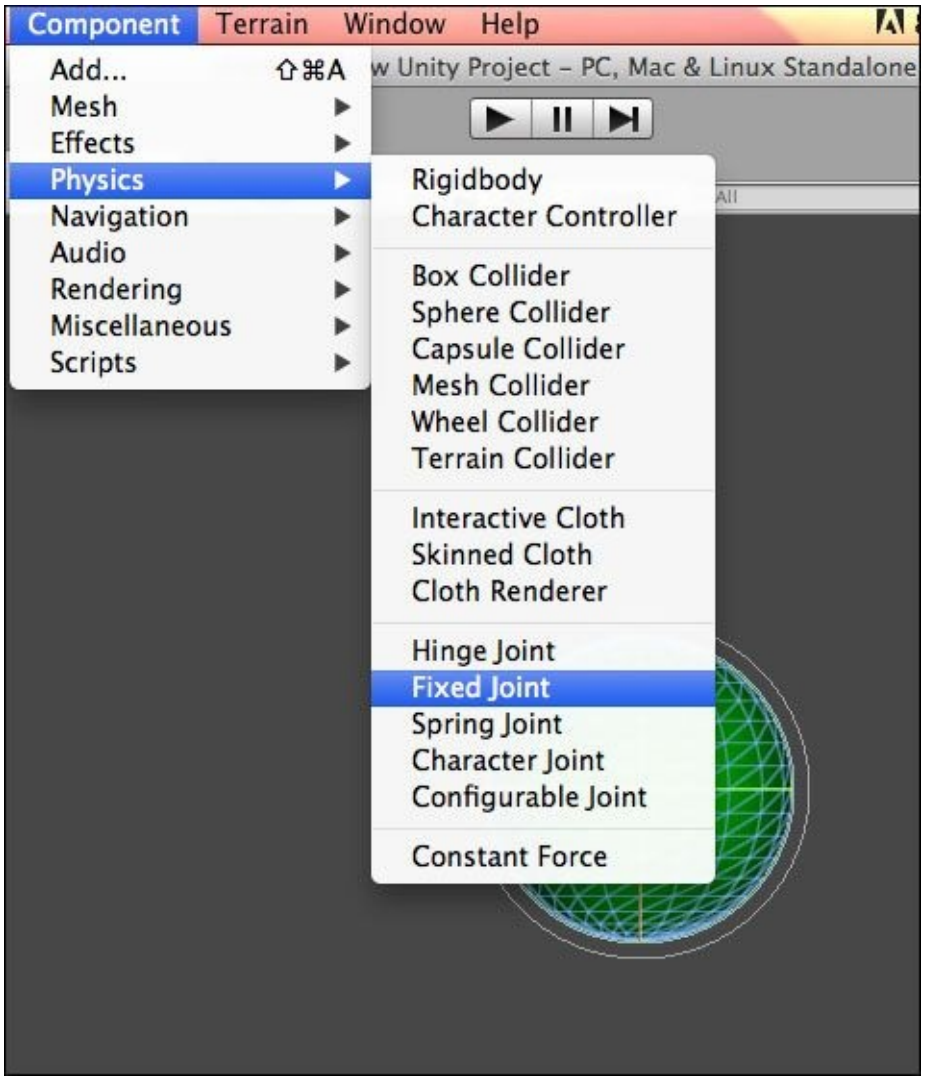
5. Add the script file to **Main Camera** and drag the game object box to the box variable.
6. Run the scene and you will see the spin movement of the box.

In another example, we will create an animation using different joints. In the previous chapter, you learned different joints, including the character joint, which is used for a Ragdoll effect. Now, we will create a rope animation using **Fixed Joint** and **Hinge Joint**.

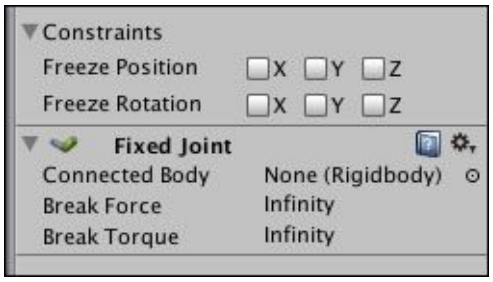
An example of rope animation using different joints

In the following example, we will create a rope using different joints. Joints help to create flexible animation, and this is why we are going to use joints for the rope animation.

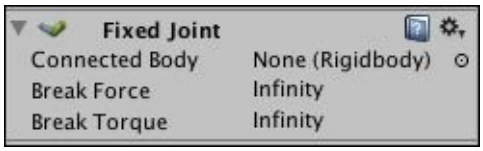
1. Create a new scene and save it as `Rope Animation example`.
2. Create a **Sphere** game object and apply a material to make it colorful.
3. Now, apply **Rigidbody** and **Fixed Joint** on it as shown in the following screenshot:



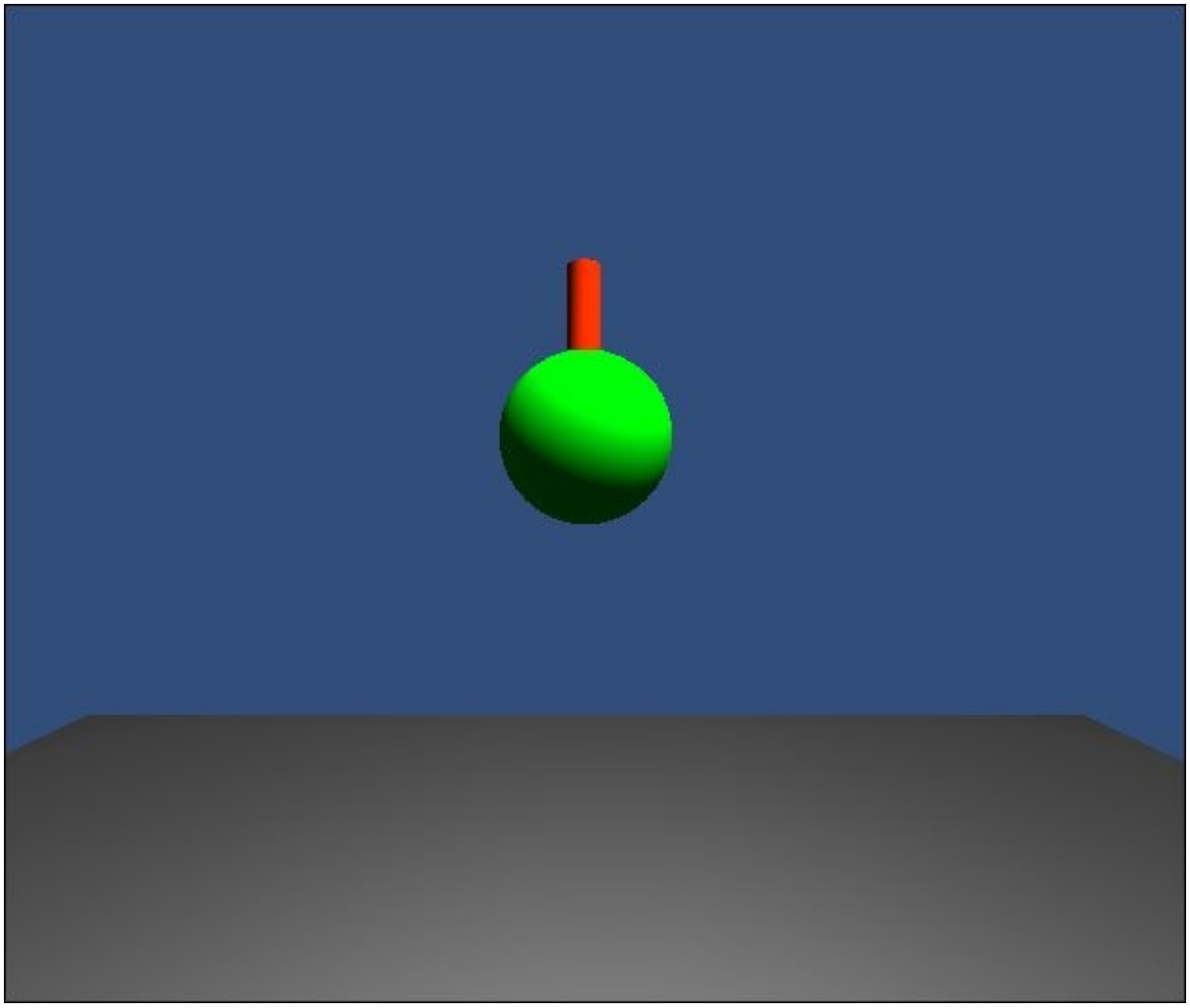
4. Create a **Cylinder** game object and name it `Chain 1`.
5. Now, apply **Rigidbody** and **Hinge Joint** on it. As shown in the following screenshot, you will see the **Connected Body** property:



6. In the **Connected Body** property of the **Sphere** game object, drag the **Chain 1** game object.

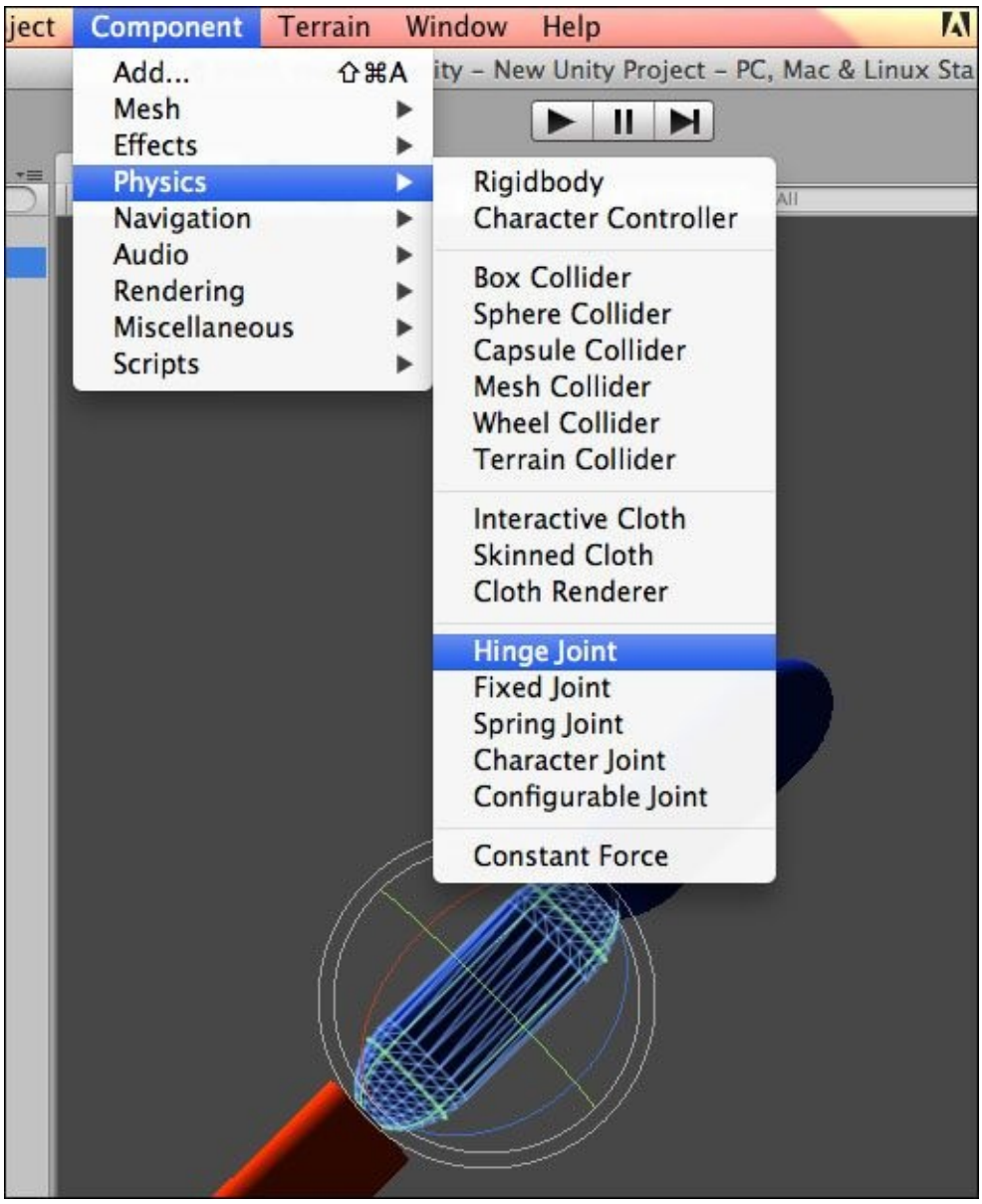


7. Now, run the scene; you will see that the sphere falls along with the **Cylinder** game object as shown in the following screenshot:

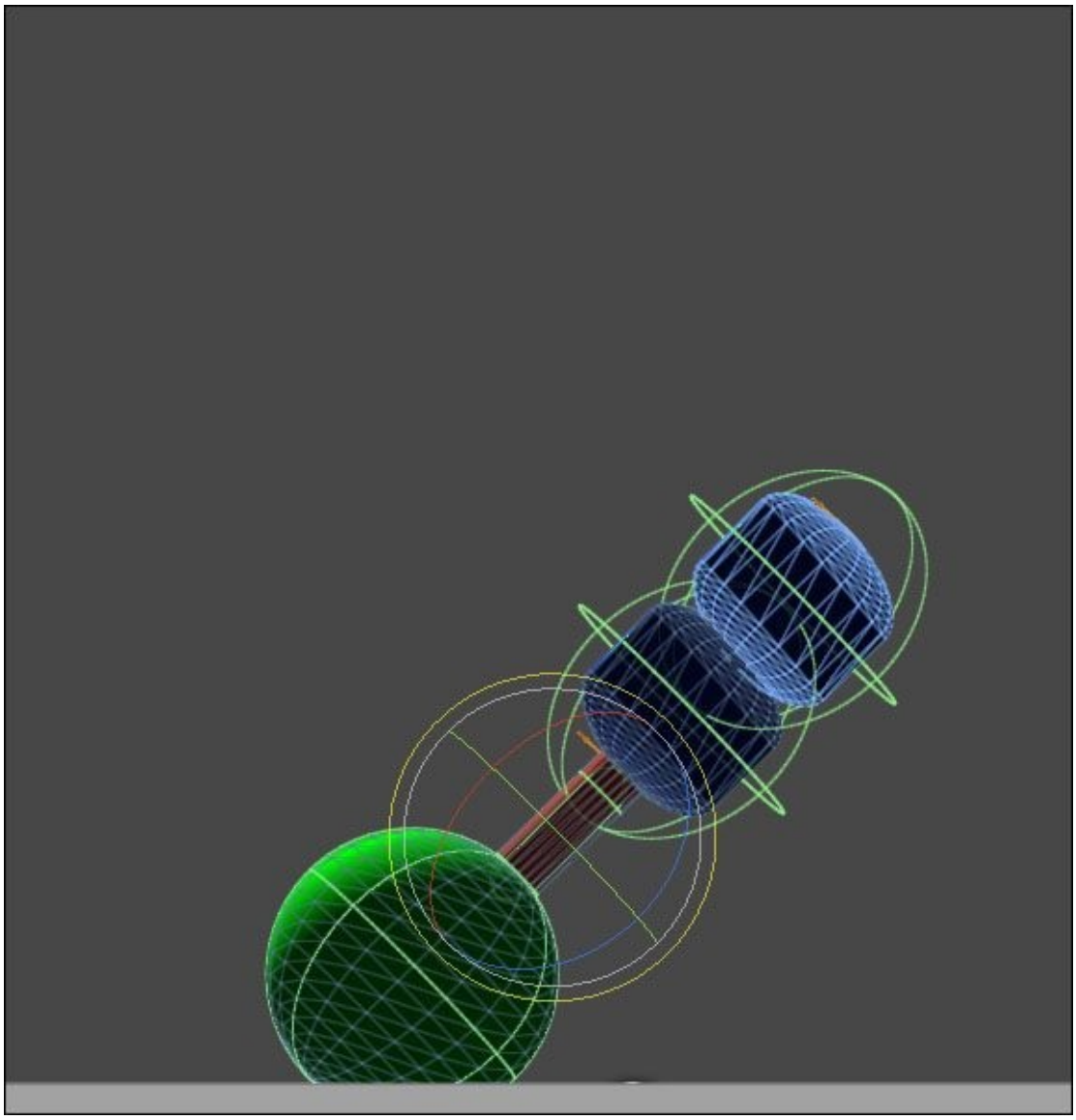


8. As shown in the following screenshot, now create the `Chain 2` game object and apply

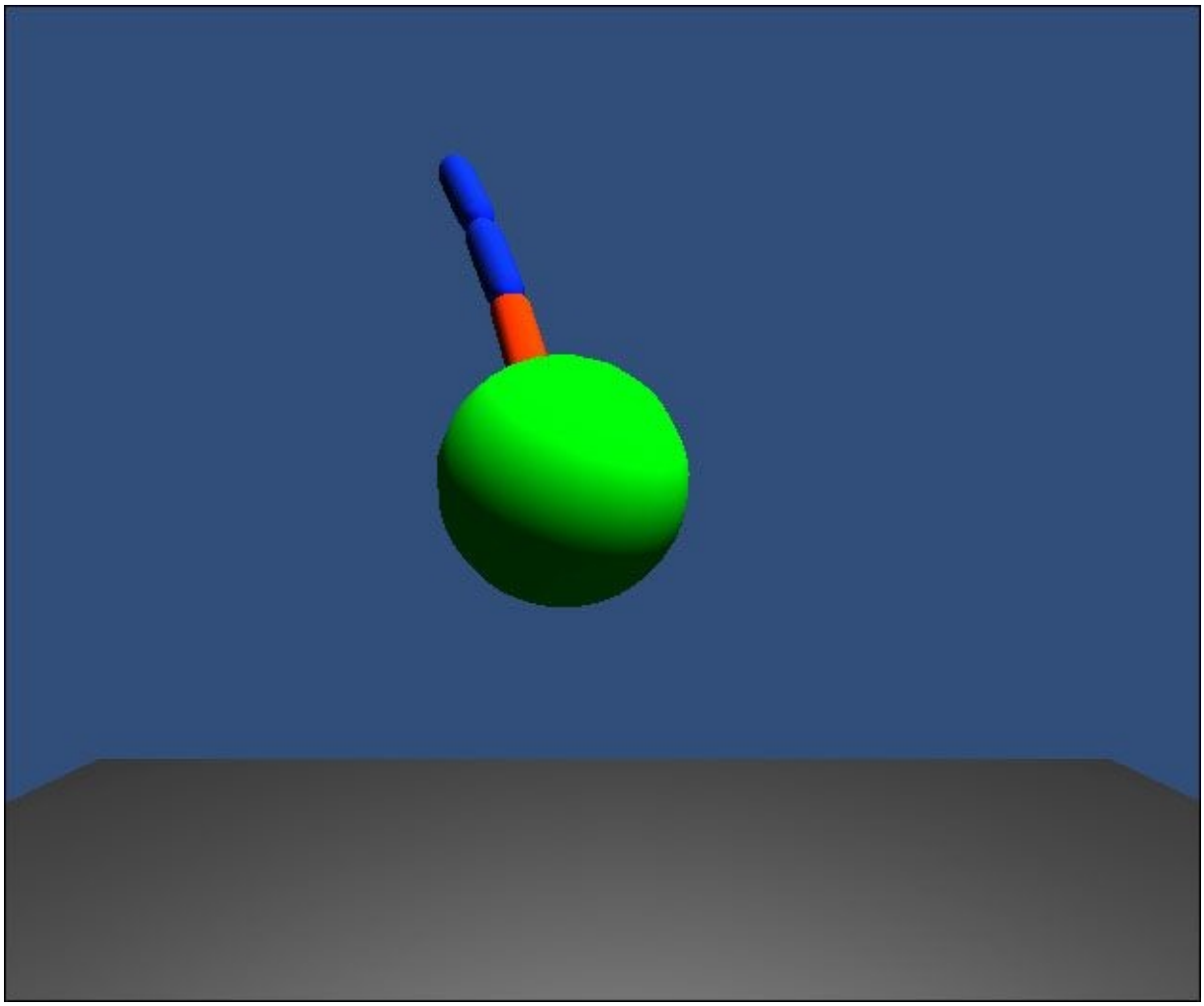
Rigidbody and **Hinge Joint**. Specify its position above the **Chain 1** and drag **Chain 2** as connected body for **Chain 1**.



9. Similarly, create **Chain 3** and **Chain 4**. Now, rotate all the game objects as shown in the following screenshot:



10. Test this application; you will see a sphere hanging from a rope:



In the preceding steps, you learned how to create an animation using different joints.

Summary

In this chapter, you learned about the different Physics components of Unity, using which we can create different types of animation. You learned how to use the Cloth component and `AddForce` to create the projectile animation of a bullet. You learned how to create the wrecking animation of ball using different joints. In the next chapter, you will learn how to create a smooth game play using different Physics components and how we can handle the game play performance.

Chapter 7. Optimizing Application’s Performance Using Physics in Unity3D

In this chapter, you will learn how to optimize your game or application during development with Unity3D. There are several factors that need to be kept in mind to run the app or game smoothly. We will handle optimization using Physics best practices. Although, this chapter will cover Physics tricks primary to handle the performance optimization, you will learn other Unity3D tricks for performance handling as bonus topics.

We will cover the following topics:

- Developing an optimized application and game
- Checking performance
- Moving the static collider
- Mesh Colliders
- The complex collider shape
- Rigidbodies
- Joints
- The Cloth component
- Optimized graphics
- Low timestep
- Pros of performance optimization

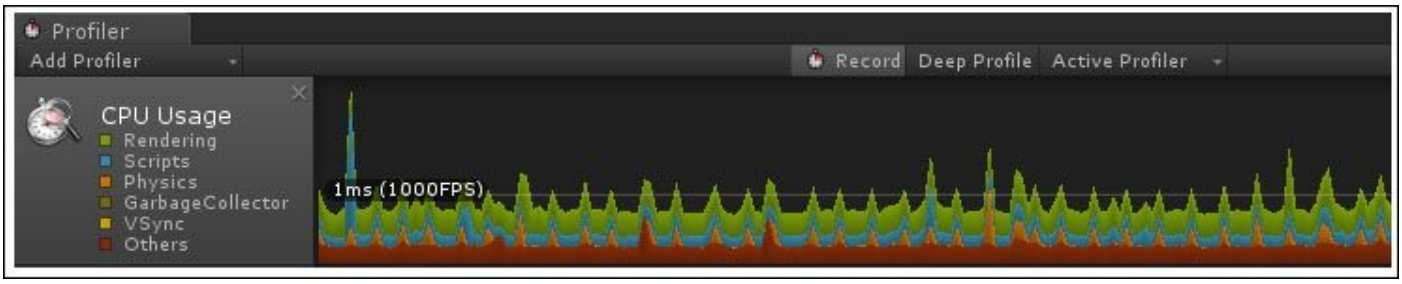
Developing an optimized application and game

Performance is a critical factor for games and applications, and for a fast-paced action game, it becomes the key point. For those features where fast Physics combined with fully animated characters and a 3D world are required, performance optimization is the most important factor. Any game or application needs 60 frames per seconds of performance, and so we will need to optimize our game for target devices to achieve that.

Checking performance

Unity Profiler is the first thing that we should use to check the performance of a game or application. Profiler is a great tool that comes with Unity Pro, using which we determine where any frame rate issues are coming from.

Profiler uses a graph to show the CPU usage while we play the game. Profiler is divided in categorizes such as **Rendering**, **Scripts**, **Physics**, **Garbage Collector**, **VSync**, and others. This is how it looks:



Now, let’s see how we handle optimization during the Physics implementation.

Moving static colliders

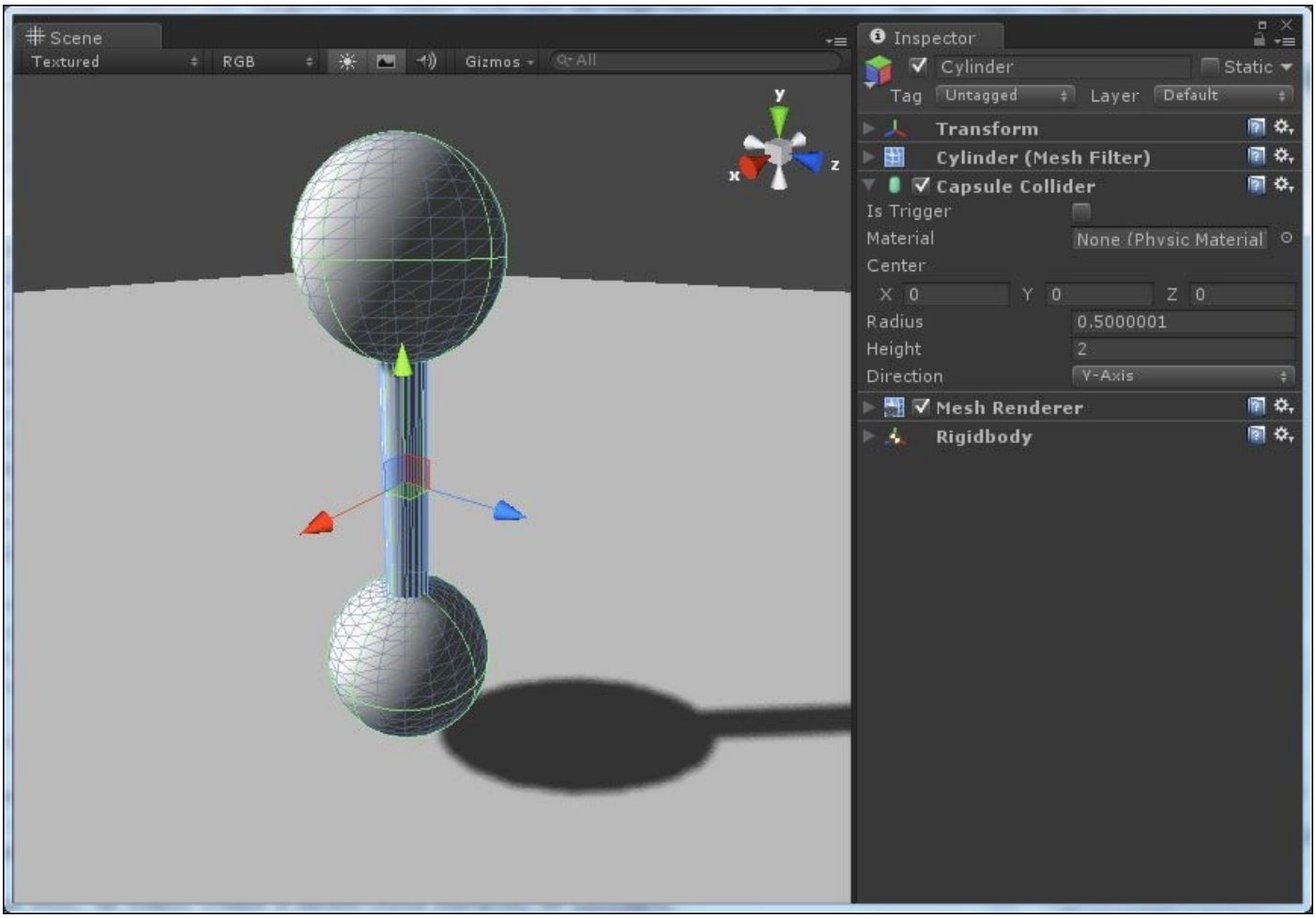
To make less expensive performance, we should avoid moving static colliders. You learned in previous chapters that a static collider is a game object with a collider component on it; however, it does not have a Rigidbody component. Moving static colliders is one of the top causes of performance issues in Unity games and it is expensive. If we need to create them with codes, we should add the collider and Physic Materials *after* its positioning.

Mesh Colliders

Mesh Colliders are slow compared to the primitive Box/Sphere Collider. A sphere has many more vertices than a cube but the uniform distance from the center makes the calculation much easier in comparison to the many individual triangles. Mesh colliders have a much higher performance overhead than primitive colliders.

The complex collider shape

To get better performance for more complex shapes, we should combine primitive colliders. Let’s have a look at an example. If we have a parent object with a Box Collider and Rigidbody component, we should add child objects with just a Box Collider. The collision for an entire object will have one multipart object. Rather than having several objects linked together, we can add more child objects with Rigidbodies and colliders and use joints to connect them to the parent object. As shown in the following screenshot, we used a Capsule Collider for the complex object:



Rigidbodies

Less use of Rigidbodies and materials saves performance. Again, the use of interpolation and extrapolation on Rigidbodies is discouraged all together. The total amount of Physics calculation depends on the number of nonsleeping Rigidbodies and colliders in the scene and the complexity of the colliders. We should handle performance by reducing calculation as much as possible.

Joints

We cannot use multiple basic joints on one game object as it is not supported, but we can use multiple configurable joints, which helps a lot in performance optimization. Rather than having a network of joint objects, we should use configurable joints wherever possible to avoid unnecessary memory consumption.

The Cloth component

The use of multiple Cloth components in one game is very expensive so should minimize the use of multiple Cloth components.

Lower timestep

A lower frame rate gives breathing room and helps in memory optimization. I personally find a 0.03 fixed timestep with a maximum of about 0.05 to be good for better performance. We can reduce the time spent on Physics updates by adjusting the fixed timestep setting. Increasing the timestep will reduce the CPU overhead but sometimes, the accuracy of Physics gets affected.

Precalculation

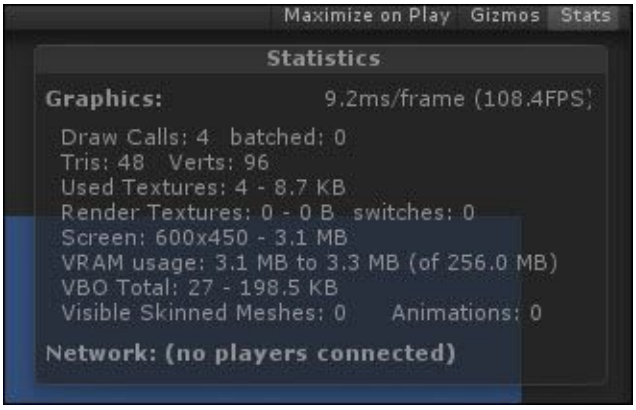
Precalculation during development can be very important to achieve high performance and make certain effects possible. Taking the approach of calculating as much as possible at the start of execution can have a great impact on performance.

Apart from Physics-based performance optimizations, there are other factors too that we should use to optimize our game or app. Let’s have a few examples.

Optimizing graphics

Performance optimization depends upon how fast we can render by GPU, which is mostly limited by the number of pixels rendered and by the memory bandwidth. The CPU performance is also limited by the amount of draw calls processed. We can use GPU Profiler to find out how much time and how many draw calls are in the scene. To save rendering time, we should remove as many draw calls as possible.

As shown in the following screenshot, by clicking on **Stats**, we can see the **Statistics** window:



To improve CPU performance, we should take into account the following points:

- For high performance, don’t use more than a few hundred draw calls per frame while draw call counts vary for older devices
- By combining nearby objects into single meshes, we can reduce draw calls
- By using automatically Unity’s draw call batching, we can reduce draw calls
- Using fewer different materials enables better batching of meshes
- By using a textures atlas where possible, we can reduce draw calls

To improve GPU performance, we should note the following points:

- By reducing the texture quality in the **Quality** settings, we can make the game run faster; we limit memory bandwidth by applying this
- We can reduce shader complexity using mobile GPUs and avoid alpha-testing shader.
- Use texture compression or 16-bit textures
- Reduce the texture size

Script call optimization for an iOS build

To make a big impact on project completion, from the initial phase of project, we should use slow but safe option of **Script Call Optimization**. Using this unhandled exception will crash the Unity build but if handled, we can get a higher performance at the end of the project. To apply this, navigate to the **Player** setting and select **iOS** as shown in the following screenshot:

There are some simple rules for handling performance such as:

- Pixel lights are too expensive, so we should avoid them to get high performance
- Keep your draw call count low because graphics rendering is CPU expensive
- We should mark the objects that don’t move as static
- For moving objects, try to keep the vertex count below 300
- We should avoid instantiating or destroying objects in runtime as the memory is slow
- Avoid alpha because it rejects pixels on iOS devices and it is slow; use alpha blend instead

Pros of performance optimization

For a successful game or application, the most important and required key point is good performance. Smooth game play attracts players; similarly, a better and fast user experience allows to capture a large audience.

A great game or application but with a slow speed or that crashes often never succeeds. Thus, the most important advantage of performance optimization is that using it, we provide fast and smooth gameplay and better user experience, and avoid lags or crashes.

Summary

In this chapter, you learned the different ways to handle performance. You learned about different Physics components and the ways they can be used for better performance. You learned how to use a single collider for a complex object, that we should not use multiple cloth components in one game, and that we can optimize graphics by reducing the number of draw calls.

This book is all about learning Physics with Unity3D. We saw that Unity has a powerful Physics engine. You learned about the different Physics components provided by Unity3D to make a developer’s life easier. You learned how to use Physics when creating animations. Now, you are set to learn about networking and multiplayer game and application using Unity3D.

Index

A

B

C

    - about / Collision Matrix 3D
    - examples / Collision Matrix 3D
- collision resolution
    - about / Collision resolution
- compound colliders
    - about / Compound colliders
    - implementing / Example – implementation of compound colliders
- configurable joints
    - about / Configurable joints
    - properties / Configurable joints
    - implementing / Configurable joints
    - movement/rotation, restricting / Handling movement/rotation restriction
    - movement/rotation, restricting with Linear Limit / Limiting motions
    - movement/rotation, restricting with Angular Limit / Limiting rotation
    - movement/rotation acceleration, handling / Handling movement/rotation acceleration
    - translation acceleration, handling / Handling translation acceleration
    - rotation acceleration, handling / Handling rotation acceleration
- ConstantForce
    - about / ConstantForce
    - used, for developing animation / An example of animation using ConstantForce

D

E

F

H

I

J

- joints
  - about / Joints
  - types / Types of joints
  - fixed joint / Fixed joint
  - spring joint / Spring joint
  - hinge joint / Hinge joint
  - character joints / Character joints
  - configurable joints / Configurable joints
  - used, for developing animation / An example of rope animation using different joints

K

- Kinematic motion
  - about / Kinematic motion and Rigidbodies
- Kinematic Rigidbody
  - about / Kinematic Rigidbody
- Kinematic Rigidbody collider
  - about / Kinematic Rigidbody Collider

L

M

N

- nonprimitive colliders
  - about / Nonprimitive colliders
  - types / Types of nonprimitive colliders
  - Wheel Collider / Wheel Collider
  - static collider / Static collider
  - Rigidbody collider / Rigidbody Collider
  - Kinematic Rigidbody collider / Kinematic Rigidbody Collider
  - Trigger Collider / Trigger Collider

O

P

R

S

T

U

W