Community Experience Distilled

Unity 3.x Game Development Essentials

Game development with C# and Javascript

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

Foreword by
David Helgason, CEO and Co-founder, Unity Technologies

Will Goldstone

unity [PACKT] PUBLISHING

Unity 3.x
Game Development Essentials

Game development with C# and Javascript

Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!

Will Goldstone

BIRMINGHAM - MUMBAI

Unity 3.x Game Development Essentials
Game development with C# and Javascript

First published: October 2009

Second edition: December 2011

Production Reference: 1131211

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-144-4

www.packtpub.com

Cover Image by Will Goldstone (will@unity3d.com)

Credits

Author
Will Goldstone

Reviewers
Rune Skovbo Johansen
Mark Backler
David Fugère-Lamarre
Bastien Fontaine
Steffen Franz
Aaron Grove
Ben Lee

Acquisition Editor
Wilson D'souza

Development Editor
Maitreya Bhakal

Technical Editors
Aaron Rosario
Apoorva Bolar

Project Coordinator
Jovita Pinto

Proofreader
Aaron Nash

Indexer
Monica Ajmera Mehta

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

Foreword

When we began creating Unity, we were just three programmers working on a beautiful little game. There weren't any good game engines that one could license without putting down wads of cash, so we created our own. We eventually decided that we enjoyed the challenge of making great tools even more than making games, and after some soul searching we realized that the tools we had been creating—combined with a simple licensing structure and an open community—had the potential to change the way that developers create, distribute, and play games.

It wasn't always an easy road to where we are today. Ridiculously long days and late nights, gigs serving sandwiches and making websites for law firms, and general hardship. Once, we were told by a potential investor (he passed on the deal) that our dream of 'democratizing game development' had a 1 in 1000 chance of working out. We could think of nothing better to do than take on the odds!

Stuffing insanely complex technology into a polished package, and making it as simple as humanly possible was job one, and so we were thrilled to see the first book about our software, *Unity Game Development Essentials* released in 2009. The book helped many people get off the ground with Unity, and so when Will told me he was due to release an updated edition I was only too happy to be asked to write its foreword. A long standing member of the Unity community, we first met Will back in 2007 when he was teaching game development with Unity at UK based Bournemouth University. He went on to produce some of the first Unity video tutorials for his students and shared these on our forums, helping a generation of early adopters pick up Unity version 1.5 and upwards.

Now working with us at Unity Technologies, Will has retained much of his former career in teaching—helping us to grow adoption by new users through creating training materials, giving talks and keeping active in our community. The new *Unity 3.x Game Development Essentials* you hold in your hand (or read on your mobile or desktop!) is rewritten from the ground up—but holds on to everything that was so nice about the first edition: each part of the original has been expanded, improved or elaborated upon, and it also includes some of the many features we added to Unity since then. You will not only learn about new features however; Will thoroughly walks through the basics, through scripting, learning scripting, and even addresses that perpetual Unity conundrum: *Should I learn C# or Javascript?*—by covering both programming languages in parallel, his book lets you decide what makes the most sense for you.

Whether you are an artist, level designer, or simply a young person choosing game creation as a potential career, this book represents a fantastic start for learning Unity. Starting out by covering the essential elements of 3D, you'll learn everything from scratch—no prior knowledge is assumed, but the book moves at a pace that will keep you turning pages and writing code!

I'd like to personally welcome you to the Unity community, and hope you have as much fun reading this book as we do working on Unity.

David Helgason
CEO & Co-founder, Unity Technologies

About the Author

Will Goldstone is a longstanding member of the Unity community and works for Unity Technologies as a Technical Support Associate, handling educational content, marketing, community relations and developer support. With an MA in Creative Education, and many years experience as a lecturer in higher education, Will wrote the first ever Unity book, the original *Unity Game Development Essentials*, and also created the first ever external video tutorials for the package. Through his site `http://www.unity3dstudent.com`, Will helps to introduce new users to the growing community of developers discovering Unity everyday. He also blogs intermittently at `http://willgoldstone.com`.

I would like to thank the following parties for helping to produce this book, and being generally awesome—Teck Lee Tan (@LoTeKk) for creating the Art Assets in the book; all the Ninjas at Unity Technologies: Rune Skovbo Johansen, Nicholas Francis (@unitynich), David Helgason (@davidhelgason), Joachim Ante, Graham Dunnett, Andy Brammall (@andybrammall), Andy Stark, Charles Hinshaw, Roald Hoyer-Hansen (@brokenpoly), Carl Callewaert (@carlunity), Chris Pope (@CreativeChris1), Dave Shorter, Mark Harkness (@IAmRoflHarris), Ricardo Arango, Rob Fairchild (@robfairchild), Olly Nicholson, Cathy Yates, Adam Buckner, Richard Sykes, Emil Johansen (@AngryAnt), Ethan Vosburgh, Joe Robins (@JoeRobins) … and the many more awesome guys and girls I can't fit here!

Plus awesome Unity-powered friends Bob Berkebile (@pixelplacement), Tom Jackson (@quickfingerz), Thomas Pasieka (@thomaspasieka), Cat Burton (@catburton), Mike Renwick (@runonthespot), Mark Backler, Russ Morris (@therussmorris), Jasper Stocker (@jasperstocker), Paul Tondeur (@paultondeur), David Fugère-Lamarre, Benjamin Lee, Steffen Franz, Aaron Grove, Bastien Fontaine. And of course not forgetting Mum, Dad, Rach, Penny, and my awesome friends.

About the Reviewers

Rune Skovbo Johansen has been part of the development team at Unity Technologies since 2009, working on expanding the feature set of the editor and tightening the workflows and interface. He is based in Copenhagen, Denmark. Besides editor work he has developed procedural animation tools, written sections of the Unity documentation, and has been a programmer on several of the official Unity demos.

In general, Rune is passionate about creating solutions that make advanced and cool technology simple to use. He has a creative and cross-disciplinary approach to software development grounded in a Master's degree in Multimedia & Game Programming and an interest since childhood in graphics, animation, and coding.

Rune engages with the game development community online in various forums and blogs and offline through game jams and other events. He has been a speaker at the Game Developers Conference and Unity's own Unite Conference, and has helped organize the Nordic Game Jam.

In his spare time Rune enjoys the outdoor in parks and forests, daily biking, and reading. He also spends time working creatively with graphics and animation, and developing small games. He has a special interest in anything procedural and is persistently trying to find the best way to instruct his computer to generate giant sprawling worlds for him. He writes about his projects at `runevision.com`.

Mark Backler is a Game Designer who has been working in the games industry for over 5 years. He has worked at EA, Kuju and is currently at Lionhead Studios working on *Fable: The Journey*. He has worked on numerous games including *Harry Potter and the Order of the Phoenix*, *Milo and Kate* and the Bafta award winning *Fable 2*. He can be found on Twitter at @MarkBackler.

I would like to thank Will for writing this book, which has helped me get up to speed with Unity so quickly, Cat for putting us into contact in the first place, the talented and creative people at Lionhead from whom I'm still learning every day, and my friends, especially Anish, Tom and Chuck, and my family for being all round awesome.

David Fugère-Lamarre holds a Computer Engineering degree from the École Polytechnique de Montréal and a Master's degree in Engineering Management from the New Jersey Institute of Technology. His video game development experience started in 2004 when he worked for Behaviour Interactive (Artificial Mind & Movement) in Montreal, Canada as a game programmer on various console titles. In 2007 he worked for Phoenix Studio in Lyon, France again as a game programmer for a console title. In 2009 he co-founded Illogika Studios (`http://illogika.com/`), an independent game development company in Montreal specializing in Unity game development. He also teaches Unity pro training classes at the Centre Nad in Montreal and his involved with local colleges in creating game programming courses.

Bastien Fontaine is a 25 year old French game designer/scripter. He passed a two-year diploma from a university institute of computer science (C++, Java, PHP, SQL, and so on) at Nice, France, then a 3-year diploma on ARIES private school on Game Design/Video game jobs formation. He learned software such as Virtools, Maya, 3DS Max, Photoshop. He finished his studies with a 1-year diploma from "Université Lyon 2" (Gamagora)where he learned Level Design and tools such as Unreal Engine, Unity, Sketch Up, and improved his game design skills.

He worked with Unity at Creative Patterns (Strasbourg, France) to develop for iPhone and at *Illogika Studio* (Montreal, Canada) to develop for the iPhone too.

He also worked at *Illogika Studio* with David Fugère-Lamarre, another reviewer of this book.

Steffen Franz is currently the Technical Director at *HiveMedia* (`www.hivemedia.tv`) a branded social game company in the San Francisco Bay Area. He is also the Lead Engineer on *Deadliest Catch – The Social Game*, a Facebook game based on Discovery Channel's hit show. Since earning his B.S. in Visual and Game Programming at *The Art Institute of California*—San Francisco, Steffen has been developing on Unity for over three years, working on titles such as *Globworld*, a child friendly virtual world, Disney Online *TRON Legacy*, and *Cordy*, a 3D platformer for Google's Android mobile platform.

I would like to thank the author for this opportunity to share my knowledge and professional experience of the Unity engine. Most of all I would like to thank my family, especially our two year old toddler taking the occasional nap time, allowing me to review this book and hopefully give you more insight on how challenging, yet fun game development can be.

Aaron Grove is an award winning Visual Effects Supervisor with over 10 years experience, creating high-end visual effects for television commercials and music videos in Australia, United Kingdom and United States of America. Aaron's creativity and knowledge of 3D & visual effects (technical and artistic) combined with his passion for games gives him the drive to create and craft visually stunning games. In 2010, Aaron was the Visual Effects Supervisor on the award winning (2010 D&D Yellow Pencil Award) music video *Two Weeks* by *Grizzly Bear*. Currently Aaron is the creative director and co-founder of Blowfish Studios which is solely focused on Unity game development. More information can be found at `www.blowfishstudios.com`.

Ben Lee is a software engineer and has been working in the computer game industry for over 13 years on projects involving EA, Intel, nVidia and 3M. He has extensive experience with designing and programming computer game engines and all other aspects of game software development for a variety of hardware platforms. Most recently Ben co-founded Blowfish Studios(`www.blowfishstudios.com`) and has since been focused solely on Unity game development.

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.

Table of Contents

This Glossary is not present in the book but is available as a free download from: `http://www.packtpub.com/sites/default/files/downloads/1444OT_Glossary_Final.pdf`

Preface

Game Engines such as Unity are the power-tools behind the games we know and love. Unity is one of the most widely-used and best loved packages for game development and is used by everyone from hobbyists to large studios to create games and interactive experiences for the web, desktops, mobiles, and consoles. With Unity's intuitive, easy to learn toolset and this book, it's never been easier to become a game developer.

Taking a practical approach, this book will introduce you to the concepts of developing 3D games, before getting to grips with development in Unity itself—prototyping a simple scenario, and then creating a larger game. From creating 3D worlds to scripting and creating game mechanics you will learn everything you'll need to get started with game development.

This book is designed to cover a set of easy-to-follow examples, which culminate in the production of a First Person 3D game, complete with an interactive island environment. All of the concepts taught in this book are applicable to other types of game, however, by introducing common concepts of game and 3D production, you'll explore Unity to make a character interact with the game world, and build problems for the player to solve, in order to complete the game. At the end of the book, you will have a fully working 3D game and all the skills required to extend the game further, giving your end-user - the player - the best experience possible. Soon you will be creating your own 3D games and interactive experiences with ease!

What this book covers

Chapter 1, *Enter the Third Dimension*: In this chapter, we will introduce you to the concepts of working in 3D and how game development works with Unity. Having covered how 3D development works, you will learn the core windows that make up the Unity Editor environment.

Chapter 2, *Prototyping and Scripting Basics*: In this chapter, we aim to get you started with a practical approach to learning Unity, by building a simple game mechanic prototype, and working with C# (pronounced C-Sharp) or Javascript scripting as you get to grips with your first Unity interaction.

Chapter 3, *Creating the Environment*: Now that you have a grasp on how to make use of Unity, and it's processes, in this chapter you will begin to design outdoor environments—using the Terrain toolset to create an island environment.

Chapter 4, *Player Characters and Further Scripting*: With our island environment completed, we will look at how Player Characters are constructed in Unity, looking at scripting for characters, and also learning further scripting concepts that will be essential as you continue to create further interactions in Unity.

Chapter 5, *Interactions*: This key chapter will teach you three of the most important processes in game design with Unity—interaction using Collisions, detection of objects using Ray casting, and collider intersect detection using Trigger areas.

Chapter 6, *Collection, Inventory, and HUD*: With a variety of interactions covered, we will put this knowledge into further practice as we learn how to make a simple inventory for the player, and an accompanying heads up display to record items collected.

Chapter 7, *Instantiation and Rigidbodies*: To put interaction into practice once more, we will make use of Unity's built-in physics engine, and learn how to combine this with animation to create a shooting mini-game.

Chapter 8, *Particle Systems*: Taking a break from scripting, we'll dive into some of Unity's visual effects to create a campfire using Particle systems-making systems for fire and smoke.

Chapter 9, *Designing Menus*: Every good game needs a user interface, and in this chapter we will take a look at two differing approaches to creating menus in Unity: with GUI Texture components and with the GUI scripting class.

Chapter 10, *Animation Basics*: To keep your Unity games looking dynamic, it's important to learn how to animate in Unity. In this chapter, we'll create a game ending sequence of titles and learn how to animate via scripting and also with Unity's Animation window.

Chapter 11, *Performance Tweaks and Finishing Touches*: Whilst it's important to get gameplay just right, it's also nice to make sure your game looks polished before it's ready to unleash on your audience. In this chapter, we'll look at further visual effects and optimization to make your game shine and perform well.

Chapter 12, *Building and Sharing*: In order to grow as a developer, it's really important to share your work with players, and get feedback. In this chapter, we'll learn how you can export your game as a standalone executable, and as a web player, so that you can do just that.

Chapter 13, *Testing and Further Study*: In this concluding chapter, we will look at ways of receiving feedback from your player, and give you some advice to stand you in good stead as you begin your career in development with Unity. This chapter also features some recommended further reading to cover as you progress from the confines of this book to becoming a fully-fledged game developer!

Glossary: The glossary contains descriptions of common terms that you might encounter and also serves as a handy reference.

The *Glossary* is not present in the book but is available as a free download from the following link:
`http://www.packtpub.com/sites/default/files/downloads/1444OT_Glossary_Final.pdf`

What you need for this book

For this book you will be required to download the free version of Unity, available at:

`http://www.unity3d.com/unity/download`

Your computer should also meet the following specifications, as stated on the Unity website:

- **Windows**: XP SP2 or later.
- **Mac OS X**: Intel CPU and "Leopard" 10.5 or later.
- Graphics card with 64 MB of VRAM and pixel shaders or 4 texture units. Any card made in this millennium should work.

For the latest requirements, see the Unity website:

`http://unity3d.com/unity/system-requirements.html`

Who this book is for

If you're a designer or animator who wishes to take their first steps into game development or prototyping, or if you've simply spent many hours sitting in front of video games with ideas bubbling away in the back of your mind, Unity and this book should be your starting point. No prior knowledge of game production is required, inviting you to simply bring with you a passion for making great games.

Getting help with the book and updates

This book is written with Unity version 3.4.2 in mind, and is tested for that version. However, in some rare cases, despite stringent checks of the text, book releases may feature errors that cause confusion to readers. Also, as Unity evolves and new versions of the software are released, parts of this book may need updating.

To keep you totally up to date, we're providing you with the following website in order to give you up to date changes to what is written in the book, and also any changes that are made to Unity itself - making sure that this book is always up to date with Unity's latest practices. So if you stumble upon a problem you think may have changed, or simply need help understanding the book, visit the following URL: `http://www.unitybook.net`

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 are shown as follows: "Set the `fps` variable to the Rounded value of frames."

A block of code is set as follows:

```
// Matches
private var haveMatches : boolean = false;
var matchGUIprefab : GUITexture;
```

Some code lines are long, and end up running onto new lines—make sure that you do not do this in your code. Remember that code lines are terminated by a semi-colon `;` so wherever possible, keep code on a single line. In the example below, space restraints forces this to be displayed on two lines but in your script editor, you should not place a new line to achieve this:

```
new Vector3(Mathf.Lerp(xStartPosition, xEndPosition, (Time.time-
startTime)*speed), transform.position.y,transform.position.z);
```

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: "Go to **File | Save** in the script editor, and switch back to Unity."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or 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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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.

Downloading the book asset bundle

You can download the required files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you. An updated repository of the asset bundle for this book is also available at `http://unitybook.net/book_assets.unitypackage.zip`.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/1444_Images.pdf`

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you 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/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support` and can also be viewed at `http://unitybook.net/book-errata/`.

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.

1
Enter the Third Dimension

Before getting started with any 3D package, it is crucial to understand the environment you'll be working in. As **Unity** is primarily a 3D-based development tool, many concepts throughout this book will assume a certain level of understanding of 3D development and game engines. It is crucial that you equip yourself with an understanding of these concepts before diving into the practical elements of the rest of this book. As such, in this chapter, we'll make sure you're prepared by looking at some important 3D concepts before moving on to discuss the concepts and interface of Unity itself. You will learn about:

- Coordinates and vectors
- 3D shapes
- Materials and textures
- Rigidbody dynamics
- Collision detection
- GameObjects and Components
- Assets and Scenes
- Prefabs
- Unity editor interface

Getting to grips with 3D

Let's take a look at the crucial elements of 3D worlds, and how Unity lets you develop games in three dimensions.

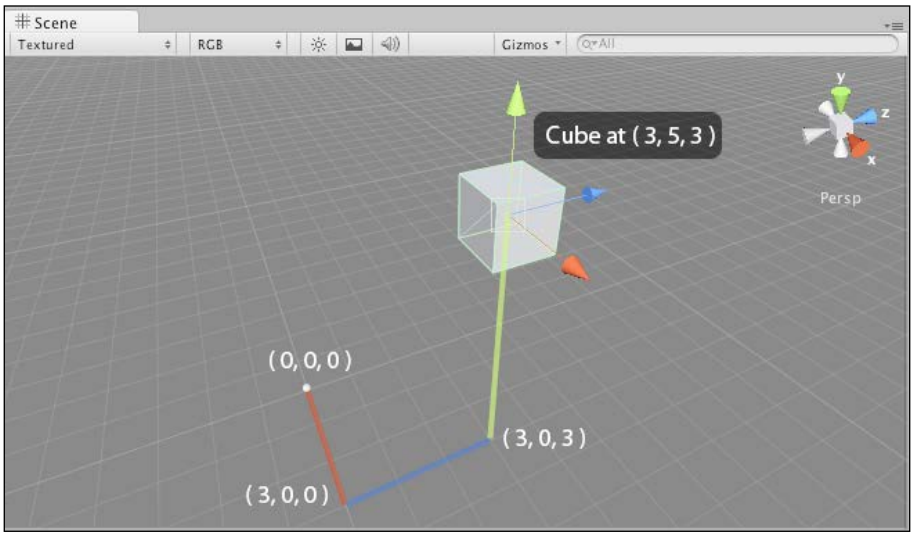
Coordinates

If you have worked with any 3D application before, you'll likely be familiar with the concept of the **Z-axis**. The Z-axis, in addition to the existing X for horizontal and Y for vertical, represents depth. In 3D applications, you'll see information on objects laid out in X, Y, Z format—this is known as the **Cartesian coordinate** method. Dimensions, rotational values, and positions in the 3D world can all be described in this way. In this book, as in other documentation of 3D, you'll see such information written with parenthesis, shown as follows:

(3, 5, 3)

This is mostly for neatness, and also due to the fact that in programming, these values must be written in this way. Regardless of their presentation, you can assume that any sets of three values separated by commas will be in X, Y, Z order.

In the following image, a cube is shown at location (3,5,3) in the 3D world, meaning it is 3 units from 0 in the X-axis, 5 up in the Y-axis, and 3 forward in the Z-axis:



Local space versus world space

A crucial concept to begin looking at is the difference between local space and world space. In any 3D package, the world you will work in is technically infinite, and it can be difficult to keep track of the location of objects within it. In every 3D world, there is a point of origin, often referred to as the **'origin'** or **'world zero'**, as it is represented by the position (0,0,0).

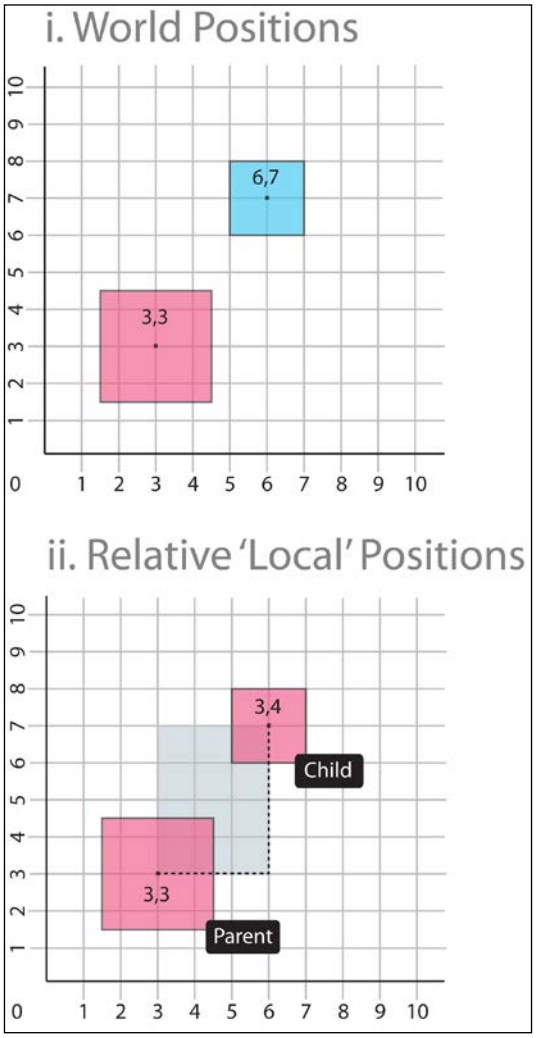
All world positions of objects in 3D are relative to world zero. However, to make things simpler, we also use local space (also known as **object space)** to define object positions in relation to one another. These relationships are known as **parent-child relationships**. In Unity, parent-child relationships can be established easily by dragging one object onto another in the Hierarchy. This causes the dragged object to become a child, and its coordinates from then on are read in terms relative to the parent object. For example, if the child object is exactly at the same world position as the parent object, its position is said to be (0,0,0), even if the parent position is not at world zero.

Local space assumes that every object has its own zero point, which is the point from which its axes emerge. This is usually the center of the object, and by creating relationships between objects, we can compare their positions in relation to one another. Such relationships, known as parent-child relationships, mean that we can calculate distances from other objects using local space, with the parent object's position becoming the new zero point for any of its child objects.

This is especially important to bear in mind when working on art assets in 3D modelling tools, as you should always ensure that your models are created at 0,0,0 in the package that you are using. This is to ensure that when imported into Unity, their axes are read correctly.

We can illustrate this in 2D, as the same conventions will apply to 3D. In the following example:

- The first diagram (**i**) shows two objects in world space. A large cube exists at coordinates(**3,3**), and a smaller one at coordinates (**6,7**).
- In the second diagram (**ii**), the smaller cube has been made a child object of the larger cube. As such the smaller cube's coordinates are said to be (**3,4**), because its zero point is the world position of the parent.



Vectors

You'll also see 3D vectors described in Cartesian coordinates. Like their 2D counterparts, 3D vectors are simply lines drawn in the 3D world that have a direction and a length. Vectors can be moved in world space, but remain unchanged themselves. Vectors are useful in a game engine context, as they allow us to calculate distances, relative angles between objects, and the direction of objects.

Cameras

Cameras are essential in the 3D world, as they act as the viewport for the screen.

Cameras can be placed at any point in the world, animated, or attached to characters or objects as part of a game scenario. Many cameras can exist in a particular scene, but it is assumed that a single main camera will always render what the player sees. This is why Unity gives you a Main Camera object whenever you create a new scene.

Projection mode—3D versus 2D

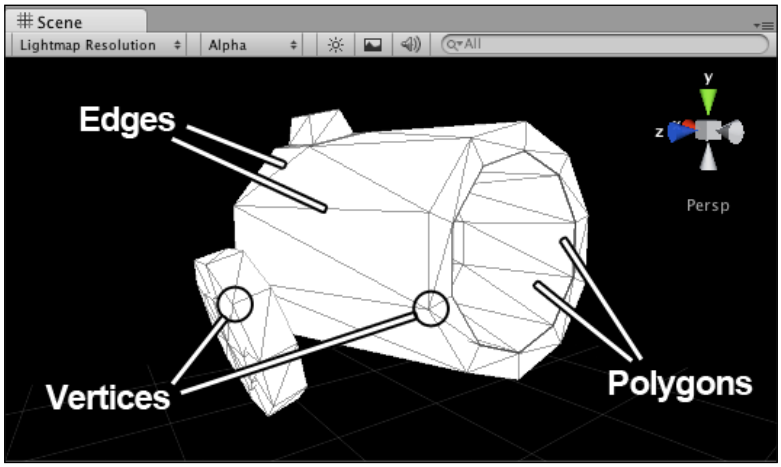
The Projection mode of a camera states whether it renders in 3D (Perspective) or 2D (Orthographic). Ordinarily, cameras are set to Perspective Projection mode, and as such have a pyramid shaped **Field of View** (**FOV**). A Perspective mode camera renders in 3D and is the default Projection mode for a camera in Unity. Cameras can also be set to Orthographic Projection mode in order to render in 2D—these have a rectangular field of view. This can be used on a main camera to create complete 2D games or simply used as a secondary camera used to render **Heads Up Display** (**HUD**) elements such as a map or health bar.

In game engines, you'll notice that effects such as lighting, motion blurs, and other effects are applied to the camera to help with game simulation of a person's eye view of the world—you can even add a few cinematic effects that the human eye will never experience, such as lens flares when looking at the sun!

Most modern 3D games utilize multiple cameras to show parts of the game world that the character camera is not currently looking at—like a 'cutaway' in cinematic terms. Unity does this with ease by allowing many cameras in a single scene, which can be scripted to act as the main camera at any point during runtime. Multiple cameras can also be used in a game to control the rendering of particular 2D and 3D elements separately as part of the optimization process. For example, objects may be grouped in layers, and cameras may be assigned to render objects in particular layers. This gives us more control over individual renders of certain elements in the game.

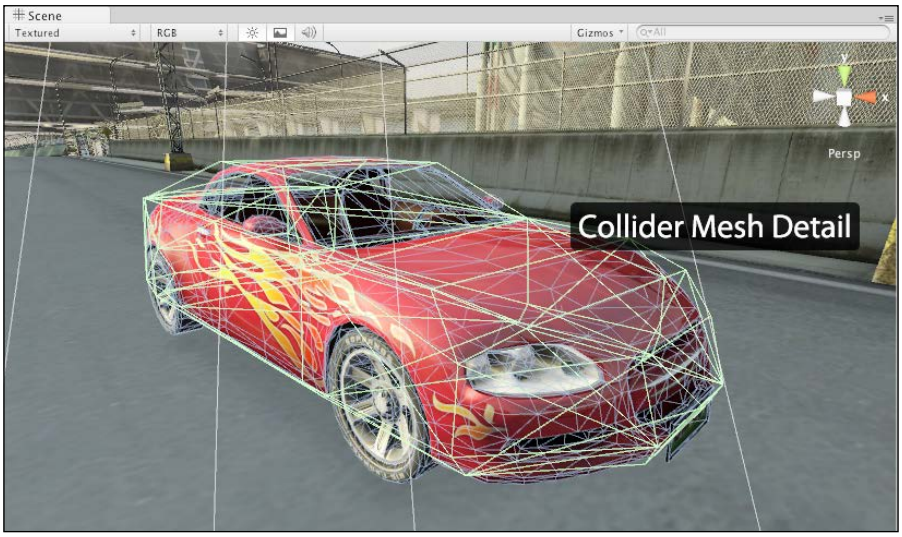
Polygons, edges, vertices, and meshes

In constructing 3D shapes, all objects are ultimately made up of interconnected 2D shapes known as **polygons**. On importing models from a modeling application, Unity converts all polygons to polygon triangles. By combining many linked polygons, 3D modeling applications allow us to build complex shapes, known as meshes. Polygon triangles (also referred to as faces) are in turn made up of three connected **edges**. The locations at which these **edges** meet are known as points or **vertices**.



By knowing these locations, game engines are able to make calculations regarding the points of impact, known as **collisions**, when using complex collision detection with **Mesh Colliders**, such as in shooting games to detect the exact location at which a bullet has hit another object. In addition to building 3D shapes that are rendered visibly, **mesh** data can have many other uses. For example, it can be used to specify a shape for collision that is less detailed than a visible object, but roughly the same shape. This can help save performance as the physics engine needn't check a mesh in detail for collisions. This is seen in the following image from the Unity car tutorial, where the vehicle itself is more detailed than its collision mesh:

In the second image, you can see that the amount of detail in the **mesh** used for the collider is far less than the visible **mesh** itself:



In game projects, it is crucial for the developer to understand the importance of the **polygon count**. The polygon count is the total number of polygons, often in reference to models, but also in reference to props, or an entire game level (or in Unity terms, 'Scene'). The higher the number of polygons, the more work your computer must do to render the objects onscreen. This is why we've seen an increase in the level of detail from early 3D games to those of today. Simply compare the visual detail in

a game such as id's *Quake*(1996) with the details seen in Epic's *Gears Of War* (2006) in just a decade. As a result of faster technology, game developers are now able to model 3D characters and worlds, for games that contain a much higher polygon count and resultant level of realism, and this trend will inevitably continue in the years to come. This said, as more platforms emerge such as mobile and online, games previously seen on dedicated consoles can now be played in a web browser thanks to Unity. As such, the hardware constraints are as important now as ever, as lower powered devices such as mobile phones and tablets are able to run 3D games. For this reason, when modeling any object to add to your game, you should consider polygonal detail, and where it is most required.

Materials, textures, and shaders

Materials are a common concept to all 3D applications, as they provide the means to set the visual appearance of a 3D model. From basic colors to reflective image-based surfaces, materials handle everything.

Let's start with a simple color and the option of using one or more images—known as **textures**. In a single material, the material works with the **shader**, which is a script in charge of the style of rendering. For example, in a reflective shader, the material will render reflections of surrounding objects, but maintain its color or the look of the image applied as its texture.

In Unity, the use of materials is easy. Any materials created in your 3D modeling package will be imported and recreated automatically by the engine and created as assets that are reusable. You can also create your own materials from scratch, assigning images as textures and selecting a shader from a large library that comes built-in. You may also write your own shader scripts or copy-paste those written by fellow developers in the Unity community, giving you more freedom for expansion beyond the included set.

When creating textures for a game in a graphics package such as Photoshop or GIMP, you must be aware of the resolution. Larger textures will give you the chance to add more detail to your textured models, but be more intensive to render. Game textures imported into Unity will be scaled to a power of 2 resolution. For example:

- 64px x 64px
- 128px x 128px
- 256px x 256px
- 512px x 512px
- 1024px x 1024px

Creating textures of these sizes with content that matches at the edges will mean that they can be tiled successfully by Unity. You may also use textures scaled to values that are not powers of two, but mostly these are used for GUI elements as you will discover over the course of this book.

Rigidbody physics

For developers working with game engines, physics engines provide an accompanying way of simulating real-world responses for objects in games. In Unity, the game engine uses Nvidia's *PhysX* engine, a popular and highly accurate commercial physics engine.

In game engines, there is no assumption that an object should be affected by physics—firstly because it requires a lot of processing power, and secondly because there is simply no need to do so. For example, in a 3D driving game, it makes sense for the cars to be under the influence of the physics engine, but not the track or surrounding objects, such as trees, walls, and so on—they will remain static for the duration of the game. For this reason, when making games in Unity a Rigidbody physics component is given to any object that you wish to be under the control of the physics engine, and ideally any moving object, so that the physics engine is aware of the moving object, to save on performance.

Physics engines for games use the Rigidbody dynamics system of creating realistic motion. This simply means that instead of objects being static in the 3D world, they can have properties such as mass, gravity, velocity, and friction.

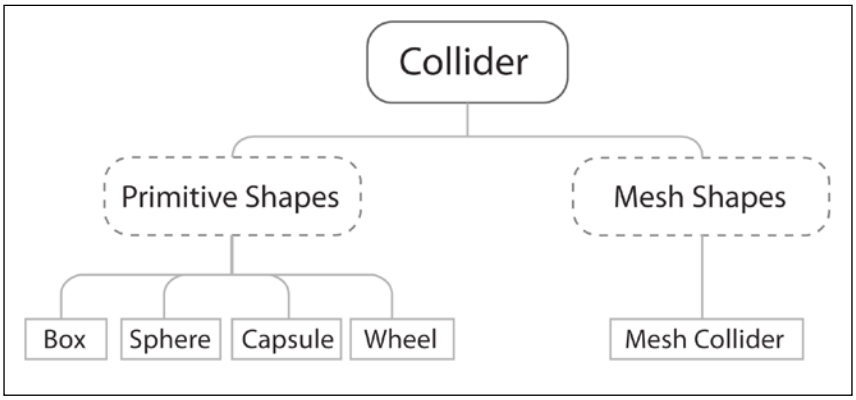
As the power of hardware and software increases, Rigidbody physics is becoming more widely applied in games, as it offers the potential for more varied and realistic simulation. We'll be utilizing rigid body dynamics as part of our prototype in this chapter and as part of the main game of the book in *Chapter 7*, *Instantiation and Rigid Bodies*.

Collision detection

More crucial in game engines than in 3D animation, collision detection is the way we analyze our 3D world for inter-object collisions. By giving an object a Collider component, we are effectively placing an invisible net around it. This net usually mimics its shape and is in charge of reporting any collisions with other colliders, making the game engine respond accordingly.

There are two main types of **Collider** in Unity—**Primitives** and **Meshes**. **Primitive shapes** in 3D terms are simple geometric objects such as **Boxes**, **Spheres**, and **Capsules**. Therefore, a primitive collider such as a **Box collider** in Unity has that shape, regardless of the visual shape of the 3D object it is applied to. Often, **Primitive colliders** are used because they are computationally cheaper or because there is no need for precision. A **Mesh collider** is more expensive as it can be based upon the shape of the 3D mesh it is applied to; therefore, the more complex the **mesh**, the more detailed and precise the collider will be, and more computationally expensive it will become. However, as shown in the Car tutorial example earlier, it is possible to assign a simpler mesh than that which is rendered, in order to create simpler and more efficient mesh colliders.

The following diagram illustrates the various types and subtypes of **collider**:



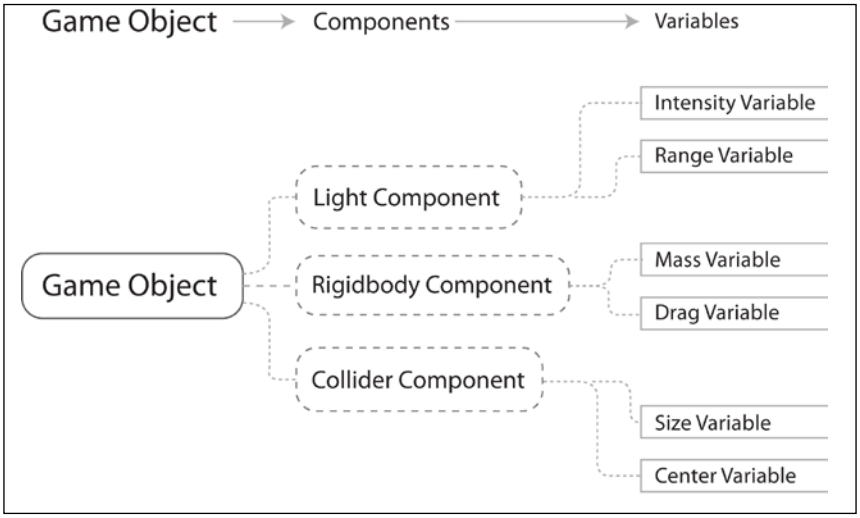
For example, in a ten-pin bowling game, a simple **Sphere collider** will surround the ball, while the pins themselves will have either a simple **Capsule collider**, or for a more realistic collision, employ a **Mesh collider**, as this will be shaped the same as the 3D mesh of the pin. On impact, the colliders of any affected objects will report to the physics engine, which will dictate their reaction, based on the direction of impact, speed, and other factors.

In this example, employing a **Mesh collider** to fit exactly to the shape of the pin model would be more accurate but is more expensive in processing terms. This simply means that it demands more processing power from the computer, the cost of which is reflected in slower performance, and hence the term expensive.

Essential Unity concepts

Unity makes the game production process simple by giving you a set of logical steps to build any conceivable game scenario. Renowned for being non-game-type specific, Unity offers you a blank canvas and a set of consistent procedures to let your imagination be the limit of your creativity. By establishing its use of the GameObject concept, you are able to break down parts of your game into easily manageable objects, which are made of many individual **Component** parts. By making individual objects within the game—introducing functionality to them with each component you add, you are able to infinitely expand your game in a logical progressive manner.

Component parts in turn have **Variables**—essentially properties of the component, or settings to control them with. By adjusting these **variables**, you'll have complete control over the effect that **Component** has on your object. The following diagram illustrates this:



In the following image we can see a **Game Object** with a **Light Component**, as seen in the Unity interface:

Now let's look at how this approach would be used in a simple gameplay context.

The Unity way—an example

If we wished to have a bouncing ball as part of a game, then we would begin with a sphere. This can quickly be created from the Unity menus, and will give you a new **GameObject** with a **Sphere mesh** (the 3D shape itself). Unity will automatically add a Renderer component to make it visible. Having created this, we can then add a Rigidbody component. A Rigidbody (Unity refers to most two-word phrases as a single word term) is a component which tells Unity to apply its physics engine to an object. With this comes properties such as mass, gravity, drag, and also the ability to apply forces to the object, either when the player commands it or simply when it collides with another object.

Our sphere will now fall to the ground when the game runs, but how do we make it bounce? This is simple! The collider component has a variable called **Physic Material**—this is a setting for the physics engine, defining how it will react to other objects' surfaces. Here we can select **Bouncy**—a ready-made Physic material provided by Unity as part of an importable package and voila! Our bouncing ball is complete in only a few clicks.

This streamlined approach for the most basic of tasks, such as the previous example, seems pedestrian at first. However, you'll soon find that by applying this approach to more complex tasks, they become very simple to achieve. Here is an overview of some further key Unity concepts you'll need to know as you get started.

Assets

These are the building blocks of all Unity projects. From textures in the form of image files, through 3D models for meshes, and sound files for effects, Unity refers to the files you'll use to create your game as assets. This is why in any Unity project folder all files used are stored in a child folder named `Assets`. This `Assets` folder is mirrored in the Project panel of the Unity interface; see *The interface* section in this chapter.

Scenes

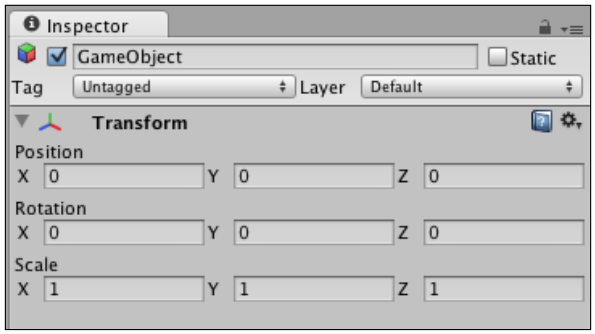
In Unity, you should think of scenes as individual levels, or areas of game content—though some developers create entire games in a single scene, such as, puzzle games, by dynamically loading content through code. By constructing your game with many scenes, you'll be able to distribute loading times and test different parts of your game individually. New scenes are often used separately to a game scene you may be working on, in order to prototype or test a piece of potential gameplay.

Any currently open scene is what you are working on, as no two scenes can be worked on simultaneously. Scenes can be manipulated and constructed by using the Hierarchy and Scene views.

GameObjects

Any active object in the currently open scene is called a **GameObject**. Certain assets taken from the Project panel such as models and prefabs become game objects when placed (or 'instantiated') into the current scene. Other objects such as particle systems and primitives can be placed into the scene by using the **Create** button on the Hierarchy or by using the **GameObject** menu at the top of the interface. All **GameObjects** contain at least one component to begin with, that is, the **Transform** component. **Transform** simply tells the Unity engine the position, rotation, and scale of an object—all described in X, Y, Z coordinate (or in the case of scale, dimensional) order. In turn, the component can then be addressed in scripting in order to set an object's position, rotation, or scale. From this initial component, you will build upon GameObjects with further components, adding required functionality to build every part of any game scenario you can imagine.

In the following image, you can see the most basic form of a **Game Object**, as shown in the **Inspector** panel:



GameObjects can also be nested in the Hierarchy, in order to create the parent-child relationships mentioned previously.

Components

Components come in various forms. They can be for creating behavior, defining appearance, and influencing other aspects of an object's function in the game. By attaching components to an object, you can immediately apply new parts of the game engine to your object. Common components of game production come built-in with Unity, such as the Rigidbody component mentioned earlier, down to simpler elements such as lights, cameras, particle emitters, and more. To build further interactive elements of the game, you'll write scripts, which are also treated as components in Unity. Try to think of a script as something that extends or modifies the existing functionality available in Unity or creates behavior with the Unity scripting classes provided.

Scripts

While being considered by Unity to be components, scripts are an essential part of game production, and deserve a mention as a key concept. In this book, we will write our scripts in both C Sharp (More often written as 'C#') and Javascript. You should also be aware that Unity offers you the opportunity to write in Boo (a derivative of the Python language).We have chosen to primarily focus on C# and Javascript as these are the main two languages used by Unity developers, and Boo is not supported for scripting on mobile devices; for this reason it is not advised to begin learning Unity scripting with Boo.

Unity does not require you to learn how the coding of its own engine works or how to modify it, but you will be utilizing scripting in almost every game scenario you develop. The beauty of using Unity scripting is that any script you write for your game will be straightforward enough after a few examples, as Unity has its own built-in `Behavior` class called `Monobehaviour`—a set of scripting instructions for you to call upon. For many new developers, getting to grips with scripting can be a daunting prospect, and one that threatens to put off new Unity users who are more accustomed to design. If this is your first attempt at getting into game development, or you have no experience in writing code, do not worry. We will introduce scripting one step at a time, with a mind to showing you not only the importance, but also the power of effective scripting for your Unity games.

To write scripts, you'll use Unity's standalone script editor, Monodevelop. This separate application can be found in the Unity application folder on your PC or Mac and will be launched any time you edit a new script or an existing one. Amending and saving scripts in the script editor will immediately update the script in Unity as soon as you switch back to Unity. You may also designate your own script editor in the Unity preferences if you wish to, such as Visual Studio. Monodevelop is recommended however, as it offers auto-completion of code as you type and is natively developed and updated by Unity Technologies.

Prefabs

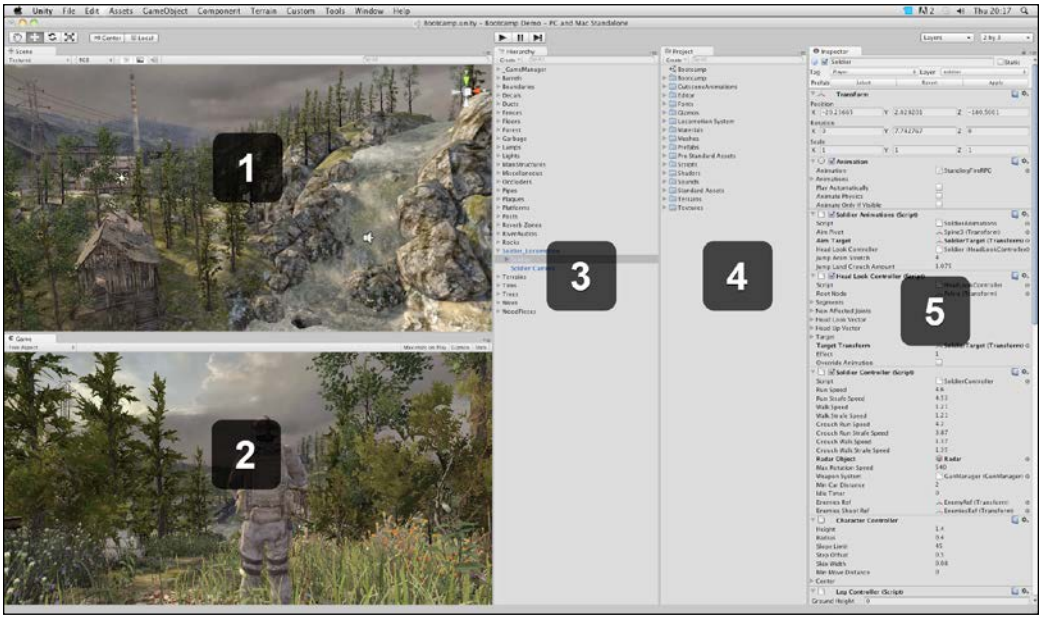
Unity's development approach hinges around the **GameObject** concept, but it also has a clever way to store objects as assets to be reused in different parts of your game, and then instantiated (also known as 'spawning' or 'cloning') at any time. By creating complex objects with various components and settings, you'll be effectively building a template for something you may want to spawn multiple instances of (hence 'instantiate'), with each instance then being individually modifiable.

Consider a crate as an example—you may have given the object in the game a mass, and written scripted behaviors for its destruction, and chances are you'll want to use this object more than once in a game, and perhaps even in games other than the one it was designed for.

Prefabs allow you to store the object, complete with components and current configuration. Comparable to the *MovieClip* concept in Adobe Flash, think of prefabs simply as empty containers that you can fill with objects to form a data template you'll likely recycle.

The interface

The Unity interface, like many other working environments, has a customizable layout. Consisting of several dockable spaces, you can pick which parts of the interface appear where. Let's take a look at a typical Unity layout:



This layout can be achieved by going to **Window | Layouts | 2 by 3 in Unity**.

As the previous image demonstrates (Mac version shown), there are five different panels or views you'll be dealing with, which are as follows:

Scene [1]—where the game is constructed.

Game [2]—the preview window, active only in play mode.

Hierarchy [3]—a list of GameObjects in the scene.

Project [4]—a list of your project's assets; acts as a library.

Inspector [5]—settings for currently selected asset/object/setting.

The Scene view and Hierarchy

The **Scene** view is where you will build the entirety of your game project in Unity. This window offers a perspective (full 3D) view, which is switchable to orthographic (top-down, side-on, and front-on) views. When working in one of the orthographic views, rotating the view will display the scene isometrically. The Scene view acts as a fully rendered 'Editor' view of the game world you build. Dragging an asset to this window (or the Hierarchy) will create an instance of it as a **GameObject** in the Scene.

The **Scene** view is tied to the **Hierarchy**, which lists all GameObjects in the currently open scene in ascending alphabetical order.

Control tools

The **Scene** window is also accompanied by four useful control tools, as shown in the following image:

Accessible from the keyboard using keys *Q*, *W*, *E*, and *R*, these keys perform the following operations:

- **The Hand tool** [*Q*]: This tool allows navigation of the Scene window. By itself, it allows you to drag around in the Scene window with the left mouse button to pan your view. Holding down Alt with this tool selected will allow you left click to orbit your view around a central point you are looking at, and holding the Alt key with right click will allow you to zoom, as will scrolling the mouse wheel. Holding the Shift key down also will speed up both of these functions.
- **The Translate tool** [*W*]: This is your active selection tool. As you can completely interact with the **Scene** window, selecting objects either in the **Hierarchy** or **Scene** means you'll be able to drag the object's axis handle in order to reposition them.
- **The Rotate tool**[*E*]: This works in the same way as Translate, using visual 'handles' to allow you to rotate your object around each axis.
- **The Scale tool** [*R*]: Again, this tool works as the Translate and Rotate tools do. It adjusts the size or scale of an object using visual handles.

Having selected objects in either the **Scene** or **Hierarchy**, they immediately get selected in both. Selection of objects in this way will also show the properties of the object in the **Inspector**. Given that you may not be able to see an object you've selected in the **Hierarchy** in the **Scene** window, Unity also provides the use of the *F* key, to focus your **Scene** view on that object. Simply select an object from the **Hierarchy**, hover your mouse cursor over the **Scene** window, and press *F*. You can also achieve this by double-clicking the name of a game object in the Hierarchy.

Flythrough Scene navigation

To move around your Scene view using the mouse and keys you can use Flythrough mode. Simply hold down the right mouse button and drag to look around in first-person style, then use *W*, *A*, *S* and *D* to move and *Q* and *E* to descend and ascend (respectively).

Control bar

In addition to the control tools, there is also a bar of additional options to help you work with your Unity scenes, which is shown as follows:

Known as the Scene View Control Bar, this bar allows you to adjust (left to right):

- Draw mode (default is 'Textured')
- Render mode (default is 'RGB')
- Toggle scene lighting
- Toggle overlays—shows and hides GUI elements and Skyboxes and toggles the 3D grid
- Toggle audition mode—previews audio sources in the current scene
- Gizmos—use this pop-out menu to show or hide Gizmos, the 2D icons of cameras, lights, and other components shown in the scene.

Search box

While the **Scene** view is intrinsically linked with the Hierarchy, often you may need to locate an item or type of item in the **Scene** view itself by searching. Simply type the name or data type (in other words, an attached component) of an object into the search, and the **Scene** view will grey out other objects in order to highlight the item you have searched for. This becomes very useful when dealing with more complex scenes, and should be used in conjunction with *F* on the keyboard to focus on the highlighted object in the **Scene** window itself.

Create button

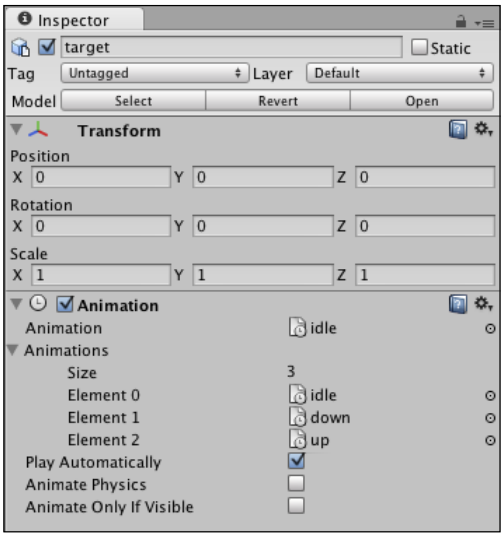
As many of the game assets you'll use in Unity will be created by the editor itself, the **Hierarchy** has a **Create** button that allows you to create objects that are also located within the top **GameObject** menu. Similar to the **Create** button on the Project panel, this drop-down menu creates items and immediately selects them so that you may rename or begin working with them in the **Scene** or **Inspector**.

The Inspector

Think of the **Inspector** as your personal toolkit to adjust every element of any GameObject or asset in your project. Much like the *Property Inspector* concept utilized by Adobe in Flash and Dreamweaver, this is a context-sensitive window. All this means is that whatever you select, the **Inspector** will change to show its relevant properties—it is sensitive to the context in which you are working.

The **Inspector** will show every component part of anything you select, and allow you to adjust the variables of these components, using simple form elements such as text input boxes, slider scales, buttons, and drop-down menus. Many of these variables are tied into Unity's drag-and-drop system, which means that rather than selecting from a drop-down menu, if it is more convenient, you can drag-and-drop to choose settings or assign properties.

This window is not only for inspecting objects. It will also change to show the various options for your project when choosing them from the **Edit** menu, as it acts as an ideal space to show you preferences—changing back to showing component properties as soon as you reselect an object or asset.



In this image, the **Inspector** is showing properties for a **target** object in the game. The object itself features two components—**Transform** and **Animation**. The **Inspector** will allow you to make changes to settings in either of them. Also note that in order to temporarily disable any component at any time, which will become very useful for testing and experimentation—you can simply deselect the checkbox to the left of the component's name. Likewise, if you wish to switch off an entire object at a time, then you may deselect the checkbox next to its name at the top of the **Inspector** window.

The Project window

The **Project** window is a direct view of the `Assets` folder of your project. Every Unity project is made up of a parent folder, containing three subfolders—`Assets`, `Library`, and while the Unity Editor is running, a `Temp` folder. Placing assets into the `Assets` folder means you'll immediately be able to see them in the **Project** window, and they'll also be automatically imported into your Unity project. Likewise, changing any asset located in the `Assets` folder, and resaving it from a third-party application, such as Photoshop, will cause Unity to reimport the asset, reflecting your changes immediately in your project and any active scenes that use that particular asset.