Ref: <https://www.studytonight.com/3d-game-engineering-with-unity/introduction>

What is a Game?

Well, you all have heard about the term **Game**. A Game can be defined as an activity enrolled for amusement as a prepared form of play that basically undertakes for enjoyment and/or as an educational tool.

## What is a Video Game?

A video game can be defined as a game that runs on electronic devices which involves humans interacting with a user interface(UI) for generating visual responses on any video device like TV screen or computer monitor. This electronic game when played by a user, deals with a set of graphics, images or even audio to make the game more interactive.

In simple words, video games are the digital entertainment platform (which is a form of software) that humans "play" via a computer, TV, smart phones, tablets or other gaming consoles.

Ref: https://glassyeyewear.com/blogs/article/a-guide-to-the-different-types-of-video-game-platforms

**Video games platforms**

Video games have evolved a lot since their inception—in both how they look and how they’re played. With every passing year, the variety of video game platforms expands. These platforms give gamers different experiences: some virtually catapult you into a fictional world, while others are simpler and better for gaming on the go. To gain a better understanding of the many gaming experiences now available, here’s a guide to some of the different types of video game platforms.

## PC

PC gaming is one of the best and oldest ways to experience video games. Playing on a PC has many advantages, including higher-quality visuals and more versatility. More often than not, game visuals are more superior on a PC than on a gaming console—PC gaming allows for more customization via [various accessories](https://glassyeyewear.com/blogs/article/12-essential-accessories-for-every-pc-gamer) and options that can enhance your experience. For example, you can use your keyboard and mouse for gameplay, but you also have the option to use a controller. PC gaming also gives you a wider variety of games to play, like the ones available through programs such as Steam. Certain games, such as World of Warcraft and League of Legends, are also only available on PCs.

## CONSOLES

Video game consoles from Microsoft, Sony, and Nintendo have been leading the way in gaming platforms for years now. In fact, a [2018 study](https://www.pewinternet.org/2018/05/31/teens-social-media-technology-2018/) from the Pew Research Center states that “84% of teens say they have… access to a game console at home.” Gaming consoles are a bit easier to use compared to PCs because console games are commonly played with controllers. While many companies are already developing next-gen consoles, Sony and Nintendo have also brought back classic consoles in miniature form, such as the NES Classic.

## MOBILE

A recent trend in gaming is the increased development of games for mobile phones and tablets. Most people today carry a smartphone with gaming capabilities; the games available on mobile devices range across all different types of genres. This is an area on which the gaming industry has recently capitalized with popular titles such as Clash of Clans, Fortnite, and Angry Birds. In recent years, mobile gaming has also taken advantage of augmented reality with interactive games such as Pokémon Go and the newly-released Harry Potter: Wizards Unite.

## VIRTUAL REALITY

Aside from mobile gaming, the gaming industry also continues to push virtual reality forward, making huge strides in the platform. VR puts you in the middle of a virtual world, and it’s one of the most immersive gaming experiences available at the moment. VR requires you to put on a headset, which allows you to see the virtual world; Oculus, Sony, HTC, Samsung, and more brands offer many different types of VR headsets. You’ll also use controls that allow you to “move around” in the virtual world. To use VR safely, make sure you have enough physical space. VR is a gaming platform that may have the most potential beyond gaming, and we’ll continue to see it move forward for years to come.

## ARCADE

Arcade games may seem like a dying gaming platform, but this is far from the case. The thirst for nostalgia has never been greater, and nothing quenches that thirst like arcade gaming. With games such as Pacman, Donkey Kong, Galaga, and more, this platform was the most popular way to game in the 80s. Back then, people typically had to visit local arcades, restaurants, or movie theaters to get their gaming in. Arcade games typically feature a large machine built for one game. These games are still in production today and are present in entertainment centers such as Dave & Busters and Main Event. You can also buy an arcade station for yourself and build a collection of your own.

## TELEVISION

An often-overlooked way to play video games is on a television. If you own a smart TV, you can actually download games and play without a console. While every smart TV is different, most of them offer app stores where you can download these games in the same way you would download a mobile game. To play these games, you may need to connect a gaming controller, but you get to enjoy some pretty cool games on a large TV display, which is more affordable than getting a console.

## WEB BROWSER

Another easy way to game is to simply use a web browser. Thousands of free games are available on the Internet, from games based on TV shows to classics such as RuneScape and FarmVille. There are also plenty of fun, educational games that can actually help with school, such as Lemonade Stand. The Internet might be the most accessible way for people of all ages to play games today.

## STREAMING

The final gaming platform in our list is one that is still a new, exciting concept. TV has been transitioning into streaming, and now video games are trying to do the same. In most cases, you have to download a game in order to get good quality—however, Google is looking to make gaming more like Netflix with its cloud gaming service, Stadia. Instead of housing games on one platform, Stadia allows you to game across many platforms by streaming the games in 4K. This would make video games the latest piece of technology to move into the cloud. This will also make gaming easier than ever before and allow you to play wherever you want with a monthly subscription.

The many platforms that produce video game titles are vastly different from one another. This has allowed the gaming industry to thrive and provide consumers with many different options. This progression will only continue as games continue to advance at a rate faster than in any other entertainment medium.

Ref: https://en.wikipedia.org/wiki/PC\_game

**PC Gaming in depth**

A **PC game**, also known as a **computer game** or **personal computer game**, is a type of [video game](https://en.wikipedia.org/wiki/Video_game) played on a [personal computer](https://en.wikipedia.org/wiki/Personal_computer) rather than a [video game console](https://en.wikipedia.org/wiki/Video_game_console) or [arcade machine](https://en.wikipedia.org/wiki/Arcade_cabinet). Its defining characteristics include: more diverse and user-determined gaming hardware and software; and generally greater capacity in input, processing, video and audio output. The uncoordinated nature of the PC game market, and now its lack of physical media, make precisely assessing its size difficult.[[1]](https://en.wikipedia.org/wiki/PC_game#cite_note-stuart10-1) In 2018, the global PC games market was valued at about $27.7 billion.[[2]](https://en.wikipedia.org/wiki/PC_game#cite_note-2)

[Home computer](https://en.wikipedia.org/wiki/Home_computer) games became popular following the [video game crash of 1983](https://en.wikipedia.org/wiki/Video_game_crash_of_1983), leading to the era of the "bedroom coder". In the 1990s, PC games lost [mass-market](https://en.wikipedia.org/wiki/Mass_market) traction to [console games](https://en.wikipedia.org/wiki/Console_game), before enjoying a resurgence in the mid-2000s through [digital distribution](https://en.wikipedia.org/wiki/Digital_distribution).[[1]](https://en.wikipedia.org/wiki/PC_game#cite_note-stuart10-1)[[3]](https://en.wikipedia.org/wiki/PC_game#cite_note-economist12-3)

[Newzoo](https://en.wikipedia.org/w/index.php?title=Newzoo&action=edit&redlink=1) reports that the *PC gaming sector* is the third-largest category (and estimated in decline) across all platforms as of 2016, with the *console sector* second-largest, and *mobile /*[*smartphone*](https://en.wikipedia.org/wiki/Smartphone)*gaming sector* biggest. 2.2 billion video gamers generate US$101.1 billion in revenue, excluding hardware costs. "Digital game revenues will account for $94.4 billion or 87% of the global market. [Mobile](https://en.wikipedia.org/wiki/Mobile_operating_system) is the most lucrative segment, with [smartphone](https://en.wikipedia.org/wiki/Smartphone) and [tablet](https://en.wikipedia.org/wiki/Tablet_computer) gaming growing 19% year on year to $46.1 billion, claiming 42% of the market. In 2020, mobile gaming will represent just more than half of the total games market. [...] China expected to generate $27.5 billion, or one-quarter of all revenues in 2017."[[4]](https://en.wikipedia.org/wiki/PC_game#cite_note-4)[[5]](https://en.wikipedia.org/wiki/PC_game#cite_note-5)

### Contemporary gaming**[[edit](https://en.wikipedia.org/w/index.php?title=PC_game&action=edit&section=5" \o "Edit section: Contemporary gaming)]**

By 1996, the growing popularity of [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows) simplified device driver and memory management. The success of 3D console titles such as [*Super Mario 64*](https://en.wikipedia.org/wiki/Super_Mario_64) and [*Tomb Raider*](https://en.wikipedia.org/wiki/Tomb_Raider_(1996_video_game)) increased interest in [hardware accelerated 3D graphics](https://en.wikipedia.org/wiki/3D_acceleration) on PCs, and soon resulted in attempts to produce affordable solutions with the [ATI Rage](https://en.wikipedia.org/wiki/ATI_Rage), [Matrox Mystique](https://en.wikipedia.org/wiki/Matrox_Mystique" \o "Matrox Mystique), [S3 ViRGE](https://en.wikipedia.org/wiki/S3_ViRGE), and [Rendition Vérité](https://en.wikipedia.org/wiki/Rendition_V%C3%A9rit%C3%A9).[[50]](https://en.wikipedia.org/wiki/PC_game#cite_note-50) As 3D graphics libraries such as [DirectX](https://en.wikipedia.org/wiki/DirectX) and [OpenGL](https://en.wikipedia.org/wiki/OpenGL) matured and knocked proprietary interfaces out of the market, these platforms gained greater acceptance in the market, particularly with their demonstrated benefits in games such as [*Unreal*](https://en.wikipedia.org/wiki/Unreal_(1998_video_game)).[[51]](https://en.wikipedia.org/wiki/PC_game#cite_note-unreal-51) However, major changes to the [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows) operating system, by then the market leader, made many older DOS-based games unplayable on [Windows NT](https://en.wikipedia.org/wiki/Windows_NT), and later, [Windows XP](https://en.wikipedia.org/wiki/Windows_XP) (without using an [emulator](https://en.wikipedia.org/wiki/Emulator), such as [DOSbox](https://en.wikipedia.org/wiki/DOSbox" \o "DOSbox)).[[52]](https://en.wikipedia.org/wiki/PC_game#cite_note-dosincompatibility-52)[[53]](https://en.wikipedia.org/wiki/PC_game#cite_note-53)

**PC vs Other Platforms**

Ref: https://gamerant.com/pc-gamers-total-study/

 recent study by DFC Intelligence reveals that over 3 billion people on the planet play video games, which is about 40% of the world's population. Of those people, a whopping 1.5 billion are PC gamers, proving the enduring popularity of non-console and non-mobile play.

With almost [half the world playing video games](https://gamerant.com/3-billion-gamers-report/), it makes sense that there would be a variety of platforms that different people enjoy, given the variety in tastes and experiences in the global population. However, a significant portion of gamers, 48%, show their preference for PCs as their primary video gaming platform. That doesn't mean almost half of all gamers only play on PC, as this number includes people who also play on console or mobile.

According to the data, console-exclusive players make up a small minority of the worldwide gaming population, with only 8% of gamers primarily playing on console. However, PC gamers are more likely to be "casual" gamers, since [video game console sales makes more money](https://gamerant.com/most-popular-game-console-ps2-playstation-wii-nintendo/) as console gamers spend more on average. That explains why PC games, while popular, don't always get as much hype as new consoles or console games as the big money just isn't in PC gaming.

1.42 billion of the world's gamers are in Asia, and the [video games industry in Japan is growing](https://www.thegamer.com/the-japanese-game-industry-grows-for-third-straight-year-thanks-nintendo/) especially fast. However, a large portion of these gamers play primarily on mobile, taking up most of the world's population. Still, PC gaming has a significant holding in Asia. The next largest portions of gamers are in Europe, Latin America, MENA, and North America. Mobile gaming is the fastest-growing gaming demographic, and may even eventually overtake PC as the popular platform of choice.

Ref: <https://www.pcgamesn.com/pc-gaming-study>

**In a very reassuring finding for our line of work, a study by**[**DFC Intelligence**](https://www.dfcint.com/product/video-game-consumer-segmentation-2/)**has found that almost half of all players around the world are on PC. According to the report, over three billion people are gaming consumers, in one form or another.**

The research was published this August, and although it lives behind a paywall, stats gleamed by [IGN](https://www.ign.com/articles/three-billion-people-worldwide-now-play-video-games-new-report-shows) give a breakdown on its findings. It’s believed that 3.1 billion, some 40% of the population of the planet, consume video games, and of those nearly half are into mobile gaming, which is the fastest growing audience.

Onto the good stuff, though: 48% of the people of that 3.1 billion are on PC. That’s 1.5 billion if you don’t have a calculator near. Comparatively, only 8% are considered to be primarily console users, but they have the highest spending per player. Asia leads the regional breakdowns of the available data, contributing 1.42 billion gaming customers, and 53% of the mobile marketplace. Europe is next, with 668 million, and 17%, respectively. Latin America has 383 million and 11%, while North America has 261 million and 4%, and the last percentage is held by MENA, at 7% of the worldwide mobile players.

Mobile gaming continues to be one of the most important sectors in the industry. So much so, [Fortnite](https://www.pcgamesn.com/fortnite" \t "_blank) developer Epic and Apple are in the middle of [a large-scale legal tussle](https://www.pcgamesn.com/fortnite/ios-apple-epic-1984-free-fortnite) over how Apple runs the iOS marketplace.

So far this year, [Call of Duty: Modern Warfare is leading in sales](https://www.pcgamesn.com/call-of-duty-modern-warfare/sales-2020). Companies that make an effort with PC releases tend to see tangible benefits, like [SEGA with Persona 4: Golden this summer](https://www.pcgamesn.com/persona-4/ports). Many of [Steam’s top-sellers keep selling for years](https://www.pcgamesn.com/call-of-duty-modern-warfare/sales-2020), and some, like [Counter-Strike: Global Offensive](https://www.pcgamesn.com/counter-strike-global-offensive/csgo-update), continue to [outdo their own records](https://www.pcgamesn.com/counter-strike-global-offensive/csgo-player-count-record).

### **Game Development**

Game development is the procedure of creating video games which involves working on some platform/software to develop the game. The development is undertaken by a game developer which can be one person or a team of **Game Engineers** (for constructing the overall structure and game play), **Designers** (for designing the objects and animations), **Coders** (for coding and giving life to the game including logics, score and other calculations), **Project Manager** (for managing and promoting the game), **Game Tester** (who will test the game after it has been developed to find bugs in the game, if any). All of these human resources come under Game developers and the process of developing the complete game is termed as **Game Engineering**.

### **Game Development Tools**

Game development tools are specialized software solutions which allow and facilitate the development of a video game easier. In developing a game, the team of developers may need animation software to design a player, or trees, or any other object being used in the game; or a coder may need **Visual Studio** or **MonoDevelop** editor to inject specific code to move a game object, all these tools come under Game development tools. In other words we can say that the tools required to develop the complete game comes under the tag-line of game development tools. All these tools helps in engineering a game.

# **Game Development Concepts**

Game development is one of the most exciting fields of computer science and a major part of the software development industry. Computer games comprise of a large and ever expanding market world-wide.

Interactive digital media and games along with entertainment applications have an enormous craze amongst the daily users of PCs and play an important role in providing economical strength(as games are not free), a factor which cannot be easily neglected.

So game development will also continue giving birth to new ideas and interactive devices for making the game play more attractive and exciting. For this game developers need to be well trained and equipped with their development skills.

### **Game Development is Software Development**

The steps and stages required for a game's development follow similar stages like that of software development, along with a few extra stages for successful completion of the game. Since game development is a major part of the software development industry, therefore game development also goes through the various phases of **SDLC (Software Development Life Cycle)** along with some extra phases. Let's explain these phases in details.

Software Development Life Cycle is a well-structured and arranged sequence of phases in software engineering for developing the intended software product. Same structure needs to be followed by a game developer also. These stages are:

1. **Communication:**

Here, the user initiates the request to develop a desired game. Then he/she contacts the developer or project managing firm and tries to discuss the terms. Then after a successful agreement, stage 2 will begin. If the user himself is a game developer, then it starts from stage 2 directly.

1. **Requirement Gathering:**

In this stage, the game development team discusses the requirements to carry on the project and the project manager will decide the number of human resources required for the project. The team will carry out the discussion with varied stakeholders to discuss problems with various domains with a motive to bring out as much information as possible based on their requirements.

1. **System Analysis:**

In this stage, the developers will decide the roadmap of the plan for a successful game development till release and try to come up with the best software model (we will discuss about software engineering models in next chapter, in details) appropriate for the project. This phase also includes a proper understanding of the product's limitations or changes required in existing systems beforehand.

1. **Systems Design:**

In this phase of development, desired features and detailed work, which includes game-play, setting up of objectives and levels, screen layouts, player and game object model; creating animations in game, business rules, process diagrams (UML, DFD), pseudo-code and other documentations (GDDs) are done.

1. **Development Phase:**

The real code for your game will be written in this phase. This can be from a pseudo code or a set of algorithms written in the GDD (Game Design Document).

1. **Integration and Testing Phase:**

In this phase, a demo version of the game is released, with a trial period of 15 days or 1 month. This is done officially by the team itself to check for errors and popularity. With this partial release of game, the promotion is also started on websites like YouTube and other gaming sites. At the same time tester(s) are hired to check for bugs in the game.

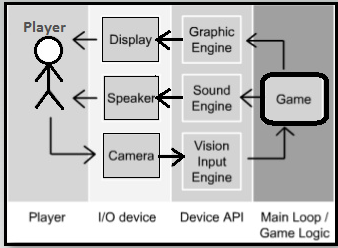
So all these stages must be followed to make a game and to make it successful.

### **Agile Model**

In this model, the product is broken down into a set of features, and hence it is used for quickly delivering a working product and so considered as a very rational development method. This model generates ongoing releases of your project, each having small and incremental changes updated from the previous released version. At each cycle, the project is tested and then released.

## Architecture of a Game

The architecture and structure of a game is similar to that of a software. But it does have some additional components which makes it different from a software. Every game has the following components:



* Graphics Engine
* Sound/Audio Engine
* Rendering & Vision-Input Engine
* I/O Devices (like, Mouse, keyboard, speaker, monitor etc)
* DLL files and Drivers/Device APIs

# **Different Genres of Game**

Genre of a game defines the exact category of the game and can be relayed through the similar gameplay characteristics, like - the type of objectives and storyline, the levels and camera point (i.e. FPS, TPS), the features and the storyline that the game is showcasing. Genre of a game is not defined by the content or the playing mode, but by the common challenges and characteristicss, that the game is having. For example, **FIFA** and **PES - Pro Evolution Soccer** have similar properties and objectives as both of them are soccer games and the player's get points or the levels of difficulty raises as the team wins the matches by scoring goals, like a real life football match. Genre of a game is decided by the resemblance found and listed under a common heading, here, FIFA and PES will come under *Sports-Genre Game*.

Ref: <https://en.wikipedia.org/wiki/Puzzle_video_game>

Puzzle games focus on logical and conceptual challenges. While many [action games](https://en.wikipedia.org/wiki/Action_game) and [adventure games](https://en.wikipedia.org/wiki/Adventure_game) include puzzle elements in level design, a true puzzle game focuses on puzzle solving as its primary gameplay activity.[[1]](https://en.wikipedia.org/wiki/Puzzle_video_game#cite_note-fundamentals-1)

Rather than presenting a random collection of puzzles to solve, puzzle games typically offer a series of related puzzles that are a variation on a single theme.

### Trial-and-Error**[[edit](https://en.wikipedia.org/w/index.php?title=Puzzle_video_game&action=edit&section=6" \o "Edit section: Trial-and-Error)]**

This sub-genre includes point-n-click games that often exhibit similarities with adventure games and walking simulators. Unlike logical puzzle games, these games generally require [inductive reasoning](https://en.wikipedia.org/wiki/Inductive_reasoning) to solve. The defining trait is that you must experiment with mechanisms in each level before you can solve them. Puzzle elements often do not have consistency throughout the game, and thus require guessing and checking.

These include *[Myst](https://en.wikipedia.org/wiki/Myst" \o "Myst)*, [*Limbo*](https://en.wikipedia.org/wiki/Limbo_(video_game)), [*The Dig*](https://en.wikipedia.org/wiki/The_Dig_(video_game)), [*Monument Valley*](https://en.wikipedia.org/wiki/Monument_Valley_(video_game)), and [escape room](https://en.wikipedia.org/wiki/Escape_the_room) games such as [*The Room*](https://en.wikipedia.org/wiki/The_Room_(video_game)).

Ref: https://www.britannica.com/topic/electronic-puzzle-game

# Electronic puzzle game

electronic game genre

**Electronic puzzle game**, [electronic game](https://www.britannica.com/topic/electronic-game) [genre](https://www.merriam-webster.com/dictionary/genre), typically involving the use of [logic](https://www.britannica.com/topic/logic), [pattern recognition](https://www.britannica.com/technology/pattern-recognition-computer-science), or [deduction](https://www.britannica.com/topic/deduction-reason).

Most popular puzzle games were made for [personal computers](https://www.britannica.com/technology/personal-computer), though some of them have been adapted for play on portable gaming systems and [mobile telephones](https://www.britannica.com/technology/mobile-telephone). Important games in this genre include Sokoban (1982), [*Tetris*](https://www.britannica.com/topic/Tetris) (1985), Windows Minesweeper (1990), and Lumines (2004).

Puzzles are frequently included in other electronic game [genres](https://www.merriam-webster.com/dictionary/genres), such as [electronic adventure games](https://www.britannica.com/topic/electronic-adventure-game), [electronic platform games](https://www.britannica.com/topic/electronic-platform-game), and [electronic role playing games](https://www.britannica.com/topic/role-playing-video-game) (RPGs). For example, Brøderbund Software’s *[Myst](https://www.britannica.com/topic/Myst)* (1993), originally released for [Apple Inc.](https://www.britannica.com/topic/Apple-Inc)’s [Mac OS](https://www.britannica.com/technology/Mac-OS), is an adventure game in which the plot is driven forward as the player gathers clues to solve a series of puzzles. Media Molecule’s *[LittleBigPlanet](https://www.britannica.com/topic/LittleBigPlanet)* (2008), released for [Sony Corporation](https://www.britannica.com/topic/Sony)’s [PlayStation 3](https://www.britannica.com/topic/PlayStation), is a platform game that features puzzle levels created by other players.

Ref: Transhistorical perspective of the puzzle video game genre August 2018

we are going to define puzzle video games as those video games where problem solving is the core and main mechanic and where the main source of satisfaction is solving these problems (Figure 1). To provide an illustrative example, a puzzle game such as Sokoban places the player in the first level, which contains the first puzzle, the first problem. Once solved, the player advances to the second problem and so on. There might be some narration or background inbetween puzzles, but it is usually minimal. Therefore, puzzle video games are designed to test the player’s ingenuity or knowledge, and the player extracts his or her pleasure from successfully passing the trial, often feeling clever on return. Non-puzzle games, on contrast, might contain puzzles, but players do other critical and meaningful actions before facing them. After solving the puzzle, they go back to other activities, which are usually more relevant to them. The puzzle feels like a nuisance, a roadblock in-between other activities. One could still argue as there are borderline cases (and player’s purpose should also be taken into account), but it is a starting point to understand the puzzle genre.

Inspirations:

Ref: https://en.wikipedia.org/wiki/Portal\_(video\_game)

***Portal*** is a 2007 [puzzle](https://en.wikipedia.org/wiki/Puzzle_video_game)-[platform game](https://en.wikipedia.org/wiki/Platform_game) developed and published by [Valve](https://en.wikipedia.org/wiki/Valve_Corporation). It was released in a [bundle](https://en.wikipedia.org/wiki/Product_bundling), [*The Orange Box*](https://en.wikipedia.org/wiki/The_Orange_Box)*,* for [Windows](https://en.wikipedia.org/wiki/Windows), [Xbox 360](https://en.wikipedia.org/wiki/Xbox_360) and [PlayStation 3](https://en.wikipedia.org/wiki/PlayStation_3), and has been since ported to other systems, including [Mac OS X](https://en.wikipedia.org/wiki/Mac_OS_X), [Linux](https://en.wikipedia.org/wiki/Linux), and [Android](https://en.wikipedia.org/wiki/Android_(operating_system)) (via [Nvidia Shield](https://en.wikipedia.org/wiki/Nvidia_Shield" \o "Nvidia Shield)).

*Portal* consists primarily of a series of puzzles that must be solved by teleporting the player's character and simple objects using "the Aperture Science Handheld Portal Device", a device that can create inter-spatial portals between two flat planes. The player-character, [Chell](https://en.wikipedia.org/wiki/Chell_(Portal)" \o "Chell (Portal)), is challenged and taunted by an [artificial intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) named [GLaDOS](https://en.wikipedia.org/wiki/GLaDOS" \o "GLaDOS) (Genetic Lifeform and [Disk Operating System](https://en.wikipedia.org/wiki/Disk_Operating_System)) to complete each puzzle in the Aperture Science Enrichment Center using the portal gun with the promise of receiving cake when all the puzzles are completed. The game's unique physics allows kinetic energy to be retained through portals, requiring creative use of portals to maneuver through the test chambers. This gameplay element is based on a similar concept from the game *[Narbacular Drop](https://en.wikipedia.org/wiki/Narbacular_Drop" \o "Narbacular Drop)*; many of the team members from the [DigiPen Institute of Technology](https://en.wikipedia.org/wiki/DigiPen_Institute_of_Technology" \o "DigiPen Institute of Technology) who worked on *Narbacular Drop* were hired by Valve for the creation of *Portal*, making it a [spiritual successor](https://en.wikipedia.org/wiki/Spiritual_successor) to the game.

Aggregate reviews for the standalone PC version of *Portal* gave the game a 90/100 through 28 reviews on [Metacritic](https://en.wikipedia.org/wiki/Metacritic" \o "Metacritic).[[72]](https://en.wikipedia.org/wiki/Portal_(video_game)#cite_note-metacritic-72) In 2011, Valve stated that *Portal* had sold more than four million copies through the retail versions, including the standalone game and *The Orange Box*, and from the Xbox Live Arcade version.[[77]](https://en.wikipedia.org/wiki/Portal_(video_game)#cite_note-4million-77)

***The Stanley Parable*** is an [interactive drama](https://en.wikipedia.org/wiki/Interactive_drama) and [walking simulator](https://en.wikipedia.org/wiki/Adventure_game#Walking_simulators) designed by American developer Davey Wreden. The game was originally released on July 31, 2011 as a free [mod](https://en.wikipedia.org/wiki/Mod_(video_gaming)) for [*Half-Life 2*](https://en.wikipedia.org/wiki/Half-Life_2).

The game has no combat or other action-based sequences. Instead, the gameplay involves guiding a [silent protagonist](https://en.wikipedia.org/wiki/Silent_protagonist) named Stanley alongside [narration](https://en.wikipedia.org/wiki/Narration) by British actor Kevan Brighting. As the story progresses, the player is confronted with diverging pathways. The player may disobey the narrator's suggestions, which will then be incorporated into the story. Depending on the choices made, the player will encounter different endings before the game restarts.

Within two weeks of its release, the mod was downloaded more than 90,000 times.[[3]](https://en.wikipedia.org/wiki/The_Stanley_Parable#cite_note-wired-5) Responses of most players were positive, and Wreden became "an overnight internet sensation among hardcore gamers."[[32]](https://en.wikipedia.org/wiki/The_Stanley_Parable#cite_note-34)

***Normality*** is a [3D](https://en.wikipedia.org/wiki/3D_computer_graphics) [graphical](https://en.wikipedia.org/wiki/Graphical_game) [adventure game](https://en.wikipedia.org/wiki/Adventure_game), released in June 1996 by [Gremlin Interactive](https://en.wikipedia.org/wiki/Gremlin_Interactive). All [cut-scenes](https://en.wikipedia.org/wiki/Cutscene) in the game are [pre-rendered](https://en.wikipedia.org/wiki/Pre-rendered). Normality has humor and is well paced, the puzzles can be hard for this post-syberia times, but it is just a matter of using lateral thinking and the 1984-style story is adequate enough to make you push the game forward.

"Normality" was a little sensation back then, being the first adventure to use a 3D engine to navigate your character through the game. And there was much to navigate. Many locations to visit, many objects to discover, many obstacles to overcome. The story and setting is still quite fun to follow and explore. Maybe the graphics are not that, what we are used to today, but from a gameplay standpoint "Normality" has aged very well.

# **Game Engines**

For every game, game engine plays a major role since the game engine helps the game designers to bring characters of the game to life, by helping in scenes, characters and graphic generation, sound, artificial intelligence, scripting animation, networking etc. Game Engine is like an integrated development environment, with a readymade suite of visual development tools and reusable software components. It turns the complex task of game development simple, by providing an abstraction layer, which makes a lot of big tasks look very easy, while the game engine does all the hardwork in the background. In other words, it is a framework that is designed specifically for the construction and development of video games. Developers use these game engines for creating games for consoles, mobile devices and personal computers.

A Game Engine is created to develop games, just like any other IDE for any particular language programming. All the components in the game engine are built and integrated to support the motive of game development.

Ref: <https://indiegamedev.net/2020/02/11/comparison-of-game-engines-2020/>

Choosing a game engine:

Choosing the right game engine comes down to personal preference. It depends on the answer of the question, what type of games do I want to make? For example, do I want to make a game on Roblox? Maybe, an RPG? If so, you can just use the OGRE engine, or RPG Maker engine. That’s not to say that it has to be those engines in order to make those games.

## [Godot](https://godotengine.org/)

Godot, is another great choice to choose from, it’s very flexible meaning that you could make 2d/3d games. It’s open source, and all the work you develop, is 100% yours. You own all the rights to it, and you won’t need to pay any fees/royalties in the future. This is a great choice for an indie game developer, but you need to take into account that there isn’t as much documentation, guides/videos as Unity or Unreal as the community is much smaller for this Engine.

|  |  |
| --- | --- |
| Pros | Cons |
| * Very flexible | * Not as beginner friendly |
| * Organized, great way to understand and see your work | * Small community |
| * Open source | * Documentation needs a lot of work |
| * MIT license, meaning that the games you develop are 100% yours, and you do not have to pay any sort of fees later on. | * Not as beginner friendly |
| * High Level Debugger | * Small community |
| * Engine is very actively developed | * Documentation needs a lot of work |

## [Unreal](https://www.unrealengine.com/)Engine

It’s true that you could get better graphics using the Unreal engine, but at the end of the day, it all depends on the assets, and visual artists. If you have great visual artists, you can make games on Unity look better than games made with Unreal. From personal experience, and others feedback, Unreal is not as beginner friendly. I’m not saying Unreal is bad, it is one of the best engines out there, and also, it’s backed with that Fortnite money.

One thing that Unreal does that not many other game engines provide, is grants. Unreal, is willing to support indie game developers with money to help fund their games. Which is really neat. Also, like Unity, Unreal has plenty of documentation, guides, and videos all over the internet.

|  |  |
| --- | --- |
| Pros | Cons |
| * Known for AAA game development | * Need a licensed copy for working |
| * Backed by Fortnite | * Not beginner friendly |
| * More tools | * More suitable for a team of developers |
| * Asset store | * Need a licensed copy for working |
| * More efficient rendering technology | * Not beginner friendly |
| * Offers grants | * More suitable for a team of developers |

## [Unity](http://unity3d.com/)Game Engine

It does get some hate, for having “bad graphics” compared to Unreal, but I believe you could make the same exact game on both engines, and have very similar graphics. In my opinion the Unity Game Engine, is very beginner friendly, and I highly recommend it as a first choice getting into indie game development, as there are plenty of documentation, guides, and videos all over the internet.

|  |  |
| --- | --- |
| Pros | Cons |
| * Beginner Friendly | * Unity is heavy, takes a lot of space on your hard drive |
| * Cross-platform friendly | * Even small games are built with a large .exe size |
| * Excellent asset store | * Does not offer grants |
| * Cheap cost / free to use | * If your games profit 100k/annually, you have to use Unity Pro (which is fair, but could be a con if you don’t want to pay a monthly fee) |
| * Endless possibilities | * Unity is heavy, takes a lot of space on your hard drive |

If you’re looking to get into indie game development, I recommend either Unreal, or Unity. For the main reasons that there are plenty of documentation, guides, videos, all over the internet, and it’s backed by great communities, and money. I personally like Unity more, but, at the end of the day, it all comes down to what type of game you want to make, and what your personal preferences are. Don’t be afraid to try out multiple engines at first. Find what works best for you, and try your best to stay motivated throughout the game development process. Good luck, with the indie game development journey.

It boiled down to either Unreal Engine or Unity Engine. And Unity was ultimately selected for the following project specific reasons:

* + Graphics are meant to be simple and cartoon-like for the purpose of the stylized game, therefore Unreal’s edge in ways of superior graphic quality was rendered irrelevant.
  + Beginner friendly: As the author both had some previous experience with Unity, and it is already more beginner friendly for learning purposes. The delay having to familiarize with UE’s environment was another setback against it.
  + Free to smalltime monetization:
  + Access to free assets and tutorials. As Unity is geared more towards a vaster audience it would be more comfortable for a quick development process.

Ref: https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp

# Unity : Developing Your First Game with Unity and C#

## What Unity Is

Unity is a 2D/3D engine and framework that gives you a system for designing game or app scenes for 2D, 2.5D and 3D. I say games and apps because I’ve seen not just games, but training simulators, first-responder applications, and other business-focused applications developed with Unity that need to interact with 2D/3D space. Unity allows you to interact with them via not only code, but also visual components, and export them to every major mobile platform and a whole lot more—for free. (There’s also a pro version that’s very nice, but it isn’t free. You can do an impressive amount with the free version.) Unity supports all major 3D applications and many audio formats, and even understands the Photoshop .psd format so you can just drop a .psd file into a Unity project. Unity allows you to import and assemble assets, write code to interact with your objects, create or import animations for use with an advanced animation system, and much more.

As **Figure 1** indicates, Unity has done work to ensure cross-platform support, and you can change platforms literally with one click, although to be fair, there’s typically some minimal effort required, such as integrating with each store for in-app purchases.

**Figure 1 Platforms Supported by Unity**

Perhaps the most powerful part of Unity is the Unity Asset Store, arguably the best asset marketplace in the gaming market. In it you can find all of your game component needs, such as artwork, 3D models, animation files for your 3D models (see Mixamo’s content in the store for more than 10,000 motions), audio effects and full tracks, plug-ins—including those like the MultiPlatform toolkit that can help with multiple platform support—visual scripting systems such as PlayMaker and Behave, advanced shaders, textures, particle effects, and more. The Unity interface is fully scriptable, allowing many third-party plug-ins to integrate right into the Unity GUI. Most, if not all, professional game developers use a number of packages from the asset store, and if you have something decent to offer, you can publish it there as well.

## What Unity Isn’t

I hesitate to describe anything Unity isn’t as people challenge that all the time. However, Unity by default isn’t a system in which to design your 2D assets and 3D models (except for terrains). You can bring a bunch of zombies into a scene and control them, but you wouldn’t create zombies in the Unity default tooling. In that sense, Unity isn’t an asset-creation tool like Autodesk Maya or 3DSMax, Blender or even Adobe Photoshop. There’s at least one third-party modeling plug-in (ProBuilder), though, that allows you to model 3D components right inside of Unity; there are 2D world builder plug-ins such as the 2D Terrain Editor for creating 2D tiled environments, and you can also design terrains from within Unity using their Terrain Tools to create amazing landscapes with trees, grass, mountains, and more. So, again, I hesitate to suggest any limits on what Unity can do.

Where does Microsoft fit into this? Microsoft and Unity work closely together to ensure great platform support across the Microsoft stack. Unity supports Windows standalone executables, Windows Phone, Windows Store applications, Xbox 360 and Xbox One.

## Getting Started

Download the latest version of Unity and get yourself a two-button mouse with a clickable scroll wheel. There’s a single download that can be licensed for free mode or pro. You can see the differences between the versions at [unity3d.com/unity/licenses](http://unity3d.com/unity/licenses). The Editor, which is the main Unity interface, runs on Windows (including Surface Pro), Linux and OS X.

I’ll get into real game development with Unity in the next article, but, first, I’ll explore the Unity interface, project structure and architecture.

## Architecture and Compilation

Unity is a native C++-based game engine. You write code in C#, JavaScript (UnityScript) or, less frequently, Boo. Your code, not the Unity engine code, runs on Mono or the Microsoft .NET Framework, which is Just-in-Time (JIT) compiled (except for iOS, which doesn’t allow JIT code and is compiled by Mono to native code using Ahead-of-Time [AOT] compilation).

Unity lets you test your game in the IDE without having to perform any kind of export or build. When you run code in Unity, you’re using Mono version 3.5, which has API compatibility roughly on par with that of the .NET Framework 3.5/CLR 2.0.

You edit your code in Unity by double-clicking on a code file in the project view, which opens the default cross-platform editor, Mono­Develop. If you prefer, you can configure Visual Studio as your editor.

You debug with MonoDevelop or use a third-party plug-in for Visual Studio, UnityVS. You can’t use Visual Studio as a debugger without UnityVS because when you debug your game, you aren’t debugging Unity.exe, you’re debugging a virtual environment inside of Unity, using a soft debugger that’s issued commands and performs actions.

To debug, you launch MonoDevelop from Unity. MonoDevelop has a plug-in that opens a connection back to the Unity debugger and issues commands to it after you Debug | Attach to Process in MonoDevelop. With UnityVS, you connect the Visual Studio debugger back to Unity instead.

When you open Unity for the first time, you see the project dialog shown in **Figure 2**.

**Figure 2 The Unity Project Wizard**

In the project dialog, you specify the name and location for your project (1). You can import any packages into your project (2), though you don’t have to check anything off here; the list is provided only as a convenience. You can also import a package later. A package is a .unitypackage file that contains prepackaged resources—models, code, scenes, plug-ins—anything in Unity you can package up—and you can reuse or distribute them easily. Don’t check something off here if you don’t know what it is, though; your project size will grow, sometimes considerably. Finally, you can choose either 2D or 3D (3). This dropdown is relatively new to Unity, which didn’t have significant 2D game tooling until fairly recently. When set to 3D, the defaults favor a 3D project—typical Unity behavior as it’s been for ages, so it doesn’t need any special mention. When 2D is chosen, Unity changes a few seemingly small—but major—things, which I’ll cover in the 2D article later in this series.

This list is populated from .unitypackage files in certain locations on your system; Unity provides a handful on install. Anything you download from the Unity asset store also comes as a .unitypackage file and is cached locally on your system in C:\Users\<you>\AppData\­Roaming\Unity\Asset Store. As such, it will show up in this list once it exists on your system. You could just double-click on any .unitypackage file and it would be imported into your project.

Continuing with the Unity interface, I’ll go forward from clicking Create in the dialog in **Figure 2** so a new project is created. The default Unity window layout is shown in **Figure 3**.

**Figure 3 The Default Unity Window**

Here’s what you’ll see:

1. Project: All the files in your project. You can drag and drop from Explorer into Unity to add files to your project.
2. Scene: The currently open scene.
3. Hierarchy: All the game objects in the scene. Note the use of the term GameObjects and the GameObjects dropdown menu.
4. Inspector: The components (properties) of the selected object in the scene.
5. Toolbar: To the far left are Pan, Move, Rotate, Scale and in the center Play, Pause, Advance Frame. Clicking Play plays the game near instantly without having to perform separate builds. Pause pauses the game, and advance frame runs it one frame at a time, giving you very tight debugging control.
6. Console: This window can become somewhat hidden, but it shows output from your compile, errors, warnings and so forth. It also shows debug messages from code; for example, Debug.Log will show its output here.

Of important mention is the Game tab next to the Scene tab. This tab activates when you click play and your game starts to run in this window. This is called play mode and it gives you a playground for testing your game, and even allows you to make live changes to the game by switching back to the Scene tab. Be very careful here, though. While the play button is highlighted, you’re in play mode and when you leave it, any changes you made while in play mode will be lost. I, along with just about every Unity developer I’ve ever spoken with, have lost work this way, so I change my Editor’s color to make it obvious when I’m in play mode via Edit | Preferences | Colors | Playmode tint.

## About Scenes

Everything that runs in your game exists in a scene. When you package your game for a platform, the resulting game is a collection of one or more scenes, plus any platform-­dependent code you add. You can have as many scenes as you want in a project. A scene can be thought of as a level in a game, though you can have multiple levels in one scene file by just moving the player/camera to different points in the scene. When you download third-party packages or even sample games from the asset store, you typically must look for the scene files in your project to open. A scene file is a single file that contains all sorts of metadata about the resources used in the project for the current scene and its properties. It’s important to save a scene often by pressing Ctrl+S during development, just as with any other tool.

Typically, Unity opens the last scene you’ve been working on, although sometimes when Unity opens a project it creates a new empty scene and you have to go find the scene in your project explorer. This can be pretty confusing for new users, but it’s important to remember if you happen to open up your last project and wonder where all your work went! Relax, you’ll find the work in a scene file you saved in your project. You can search for all the scenes in your project by clicking the icon indicated in **Figure 4** and filtering on Scene.

**Figure 4 Filtering Scenes in the Project**

In a scene, you can’t see anything without a camera and you can’t hear anything without an Audio Listener component attached to some GameObject. Notice, however, that in any new scene, Unity always creates a camera that has an Audio Listener component already on it.

## Project Structure and Importing Assets

Unity projects aren’t like Visual Studio projects. You don’t open a project file or even a solution file, because it doesn’t exist. You point Unity to a folder structure and it opens the folder as a project. Projects contain Assets, Library, ProjectSettings, and Temp folders, but the only one that shows up in the interface is the Assets folder, which you can see in **Figure 4**.

The Assets folder contains all your assets—art, code, audio; every single file you bring into your project goes here. This is always the top-level folder in the Unity Editor. But make changes only in the Unity interface, never through the file system.

The Library folder is the local cache for imported assets; it holds all metadata for assets. The ProjectSettings folder stores settings you configure from Edit | Project Settings. The Temp folder is used for temporary files from Mono and Unity during the build process.

I want to stress the importance of making changes only through the Unity interface and not the file system directly. This includes even simple copy and paste. Unity tracks metadata for your objects through the editor, so use the editor to make changes (outside of a few fringe cases). You can drag and drop from your file system into Unity, though; that works just fine.

## The All-Important GameObject

Virtually everything in your scene is a GameObject. Think of System.Object in the .NET Framework. Almost all types derive from it. The same concept goes for GameObject. It’s the base class for all objects in your Unity scene. All of the objects shown in **Figure 5** (and many more) derive from a GameObject.

**Figure 5 GameObjects in Unity**

A GameObject is pretty simple as it pertains to the Inspector window. You can see in **Figure 6** that an empty GameObject was added to the scene; note its properties in the Inspector. GameObjects by default have no visual properties except the widget Unity shows when you highlight the object. At this point, it’s simply a fairly empty object.

**Figure 6 A Simple GameObject**

A GameObject has a Name, a Tag (similar to a text tag you’d assign via a FrameworkElement.Tag in XAML or a tag in Windows Forms), a Layer and the Transform (probably the most important property of all).

The Transform property is simply the position, rotation and scale of any GameObject. Unity uses the left-hand coordinate system, in which you think of the coordinates of your computer screen as X (horizontal), Y (vertical) and Z (depth, that is, coming in or going out of the screen).

In game development, it’s quite common to use vectors, which I’ll cover a bit more in future articles. For now, it’s sufficient to know that Transform.Position and Transform.Scale are both Vector3 objects. A Vector3 is simply a three-dimensional vector; in other words, it’s nothing more than three points—just X, Y and Z. Through these three simple values, you can set an object’s location and even move an object in the direction of a vector.

## Components

You add functionality to GameObjects by adding Components. Everything you add is a Component and they all show up in the Inspector window. There are MeshRender and SpriteRender Components; Components for audio and camera functionality; physics-related Components (colliders and rigidbodies), particle systems, path-finding systems, third-party custom Components, and more. You use a script Component to assign code to an object. Components are what bring your GameObjects to life by adding functionality, akin to thedecorator pattern in software development, only much cooler.

I’ll assign some code to a new GameObject, in this case a simple cube you can create via GameObject | Create Other | Cube. I renamed the cube Enemy and then created another to have two cubes. You can see in **Figure 7** I moved one cube about -15 units away from the other, which you can do by using the move tool on the toolbar or the W key once an object is highlighted.

**Figure 7 Current Project with Two Cubes**

The code is a simple class that finds a player and moves its owner toward it. You typically do movement operations via one of two approaches: Either you move an object to a new position every frame by changing its Transform.Position properties, or you apply a physics force to it and let Unity take care of the rest.

Doing things per frame involves a slightly different way of thinking than saying “move to this point.” For this example, I’m going to move the object a little bit every frame so I have exact control over where it moves. If you’d rather not adjust every frame, there are libraries to do single function call movements, such as the freely available iTween library.

The first thing I do is right-click in the Project window to create a new C# script called EnemyAI. To assign this script to an object, I simply drag the script file from the project view to the object in the Scene view or the Hierarchy and the code is assigned to the object. Unity takes care of the rest. It’s that easy.

**Figure 8** shows the Enemy cube with the script assigned to it.

**Figure 8 The Enemy with a Script Assigned to It**

Take a look at the code in **Figure 9** and note the public variable. If you look in the Editor, you can see that my public variable appears with an option to override the default values at run time. This is pretty cool. You can change defaults in the GUI for primitive types, and you can also expose public variables (not properties, though) of many different object types. If I drag and drop this code onto another GameObject, a completely separate instance of that code component gets instantiated. This is a basic example and it can be made more efficient by, say, adding a RigidBody component to this object, but I’ll keep it simple here.

In code, I can get a reference to any component exposed in the editor. I can also assign scripts to a GameObject, each with its own Start and Update methods (and many other methods). Assuming a script component containing this code needs a reference to the EnemyAI class (component), I can simply ask for that component:

After you edit code in MonoDevelop or your code editor of choice and then switch back to Unity, you’ll typically notice a short delay. This is because Unity is background compiling your code. You can change your code editor (not debugger) via Edit | Preferences | External Tools | External Script Editor. Any compilation issues will show up at the very bottom status bar of your Unity Editor screen, so keep an eye out for them. If you try to run your game with errors in the code, Unity won’t let you continue.

## Writing Code

In the prior code example, there are two methods, Start and Update, and the class EnemyHealth inherits from the MonoBehavior base class, which lets you simply assign that class to a GameObject. There’s a lot of functionality in that base class you’ll use, and typically a few methods and properties. The main methods are those Unity will call if they exist in your class. There are a handful of methods that can get called (see [bit.ly/1jeA3UM](http://bit.ly/1jeA3UM)). Though there are many methods, just as with the ASP.NET Web Forms Page Lifecycle, you typically use only a few. Here are the most common code methods to implement in your classes, which relate to the sequence of events for MonoBehavior-derived classes:

Awake: This method is called once per object when the object is first initialized. Other components may not yet be initialized, so this method is typically used to initialize the current GameObject. You should always use this method to initialize a MonoBehavior-derived class, not a constructor. And don’t try to query for other objects in your scene here, as they may not be initialized yet.

Start: This method is called during the first frame of the object’s lifetime but before any Update methods. It may seem very similar to Awake, but with Start, you know the other objects have been initialized via Awake and exist in your scene and, therefore, you can query other objects in code easily, like so:

Update: This method is called every frame. How often is that, you ask? Well, it varies. It’s completely computation-dependent. Because your system is always changing its load as it renders different things, this frame rate varies every second. You can press the Stats button in the Game tab when you go into play mode to see your current frame rate, as shown in **Figure 10**.

**Figure 10 Getting Stats**

FixedUpdate: This method is called a fixed number of times a second, independent of the frame rate. Because Update is called a varying number of times a second and isn’t in sync with the physics engine, it’s typically best to use FixedUpdate when you want to provide a force or some other physics-related functions on an object. FixedUpdate by default is called every .02 seconds, meaning Unity also performs physics calculations every .02 seconds (this interval is called the Fixed Timestep and is developer-adjustable), which, again, is independent of frame rate.

## Unity-Generated Code Projects

Once you have code in your project, Unity creates one or more project files in your root folder (which isn’t visible in the Unity interface). These are not the Unity engine binaries, but instead the projects for Visual Studio or MonoDevelop in which you’ll edit and compile your code. Unity can create what might seem like a lot of separate projects, as **Figure 11** shows, although each one has a an important purpose.

**Figure 11 Unity-Created Projects**

If you have a simple Unity project, you won’t see all of these files. They get created only when you have code put into various special folders. The projects shown in **Figure 11** are broken out by only three types:

* Assembly-CSharp.csproj
* Assembly-CSharp-Editor.csproj
* Assembly-CSharp-firstpass.csproj

For each of those projects, there’s a dupli­cate project created with -vs appended to it, Assembly-CSharp-vs.csproj, for example. These projects are used if Visual Studio is your code editor and they can be added to your exported project from Unity for platform-specific debugging in your Visual Studio solution.

The other projects serve the same purpose but have CSharp replaced with UnityScript. These are simply the JavaScript (UnityScript) versions of the projects, which will exist only if you use JavaScript in your Unity game and only if you have your scripts in the folders that trigger these projects to be created.

Now that you’ve seen what projects get created, I’ll explore the folders that trigger these projects and show you what their purposes are. Every folder path assumes it’s underneath the /Assets root folder in your project view. Assets is always the root folder and contains all of your asset files underneath it. For example, Standard Assets is actually /Assets/Standard Assets. The build process for your scripts runs through four phases to generate assemblies. Objects compiled in Phase 1 can’t see those in Phase 2 because they haven’t yet been compiled. This is important to know when you’re mixing UnityScript and C# in the same project. If you want to reference a C# class from UnityScript, you need to make sure it compiles in an earlier phase.

Phase 1 consists of runtime scripts in the Standard Assets, Pro Standard Assets and Plug-ins folders, all located under/Assets. This phase creates the Assembly-CSharp-firstpass.csproj project.

Phase 2 scripts are in the Standard Assets/Editor, Pro Standard Assets/Editor and Plug-ins/Editor folders. The last folder is meant for scripts that interact with the Unity Editor API for design-time functionality (think of a Visual Studio plug-in and how it enhances the GUI, only this runs in the Unity Editor). This phase creates the Assembly-CSharp-Editor-firstpass.csproj project.

Phase 3 comprises all other scripts that aren’t inside an Editor folder. This phase creates the Assembly-CSharp-Editor.csproj project.

Phase 4 consists of all remaining scripts (those inside any other folder called Editor, such as /Assets/Editor or /Assets/­Foo/Editor). This phase creates the Assembly-CSharp.csproj project.

There are a couple other less-used folders that aren’t covered here, such as Resources. And there is the pending question of what the compiler is using. Is it .NET? Is it Mono? Is it .NET for the Windows Runtime (WinRT)? Is it .NET for Windows Phone Runtime? **Figure 12** lists the defaults used for compilation. This is important to know, especially for WinRT-based applications because the APIs available per platform vary.

**Figure 12 Compilation Variations**

|  |  |  |
| --- | --- | --- |
| **TABLE 1** | | |
| Platform | Game Assemblies Generated By | Final Compilation Performed By |
| Windows Phone 8 | Mono | Visual Studio/.NET |
| Windows Store | .NET | Visual Studio/.NET (WinRT) |
| Windows Standalone (.exe) | Mono | Unity - generates .exe + libs |
| Windows Phone 8.1 | .NET | Visual Studio/.NET (WinRT) |

When you perform a build for Windows, Unity is responsible for making the calls to generate the game libraries from your C#/UnityScript/Boo code (DLLs) and to include its native runtime libraries. For Windows Store and Windows Phone 8, it will export a Visual Studio solution, except for Windows standalone, in which Unity generates the .exe and required .dll files. I’ll discuss the various build types in the final article in the series, when I cover building for the platform. The graphics rendering at a low level is performed on the Windows platforms by DirectX.

Designing a game in Unity is a fairly straightforward process:

* Bring in your assets (artwork, audio and so on). Use the asset store. Write your own. Hire an artist. Note that Unity does have native support for Maya, Cheetah3d, Blender and 3dsMax, in some cases requiring that software be installed to work with those native 3D formats, and it works with .obj and .fbx common file formats, as well.
* Write code in C#, JavaScript/UnityScript, or Boo, to control your objects, scenes, and implement game logic.
* Test in Unity. Export to a platform.
* Test on that platform. Deploy.

## But Wait, I Want More!

This article serves as an overview of the architecture and process in Unity. I covered the interface, basics of assigning code, GameObjects, components, Mono and .NET, plus more. This sets us up nicely for the next article where I’ll dive right into assembling game components for a 2D game. Keep an eye on Microsoft Virtual Academy, as I’ll be doing a two-day Unity learning event late summer. And watch for local regional learning events at [unity3d.com/pages/windows/events](http://unity3d.com/pages/windows/events).

Ref: <https://docs.unity3d.com/Manual/>

# Unity architecture



The Unity engine is built with native C/C++ internally, however it has a C# wrapper that you use to interact with it. As such, you need to be familiar with some of the key concepts of scripting in C#. This section of the User Manual contains information on how Unity implements .NET and C#, and any exceptions you might encounter as you code.

For information on how to get started scripting in Unity, and the fundamentals you need to know, see the documentation on [Getting started scripting in Unity](https://docs.unity3d.com/Manual/VariablesAndTheInspector.html).

This section covers the following topics:

| **Page** | **Description** |
| --- | --- |
| [Overview of .NET in Unity](https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html) | How the Unity engine uses the .NET framework, and any differences you might encounter if you have used .NET outside of Unity before. This area also contains information on how Unity manages memory, and how to reference additional profiles in your Project. |
| [Scripting backends](https://docs.unity3d.com/Manual/scripting-backends.html) | Unity has two main scripting backends: Mono and **IL2CPP** . This section describes the differences between the backends and how and when to use them, plus their restrictions. It also contains information on managed code stripping, which removes unused code from your build. |
| [Code reloading in the Editor](https://docs.unity3d.com/Manual/code-reloading-editor.html) | Information on domain reloads and how they impact on the performance of your application. Also contains information on running code on Editor launch, and how to quickly enter and exit Play mode with Configurable Enter Play Mode. |
| [Script serialization](https://docs.unity3d.com/Manual/script-Serialization.html) | Serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. This contains information on how to use serialization in your Project in an effective way. |
| [Script compilation](https://docs.unity3d.com/Manual/script-compilation.html) | How Unity compiles your **scripts**  and in what order. Also contains information on Assembly Definitions and best practices around using them. |

# Important Classes

This section provides an overview of some of the most commonly used and important built-in classes in Unity that you may want to use when scripting.

These pages serve as a starting point for the discovery of scripting basics in Unity, and do not cover all classes in Unity, or even every member of the classes which are covered.

For a more complete reference of all the built-in classes and every member available, see the (Script Reference)[ScriptRef:index.html].

* [GameObject](https://docs.unity3d.com/Manual/class-GameObject.html)  
  : Represents the type of objects which can exist in a **Scene**  
  .
* [MonoBehaviour](https://docs.unity3d.com/Manual/class-MonoBehaviour.html): The base class from which every Unity script derives, by default.
* [Object](https://docs.unity3d.com/Manual/class-Object.html): The base class for all objects that Unity can reference in the editor.
* [Transform](https://docs.unity3d.com/Manual/ScriptingTransform.html): Provides you with a variety of ways to work with a GameObject’s position, rotation and scale via script, as well as its hierarchical relationship to parent and child GameObjects.
* [Vectors](https://docs.unity3d.com/Manual/VectorCookbook.html): Classes for expressing and manipulating 2D, 3D, and 4D points, lines and directions.
* [Quaternion](https://docs.unity3d.com/Manual/class-Quaternion.html)  
  : A class which represents an absolute or relative rotation, and provides methods for creating and manipulating them.
* [ScriptableObject](https://docs.unity3d.com/Manual/class-ScriptableObject.html): A data container that you can use to save large amounts of data.
* [Time (and framerate management)](https://docs.unity3d.com/Manual/TimeFrameManagement.html): The Time class allows you to measure and control time, and manage the framerate of your project.
* [Mathf](https://docs.unity3d.com/Manual/class-Mathf.html): A collection of common math functions, including trigonometric, logarithmic, and other functions commonly required in games and app development.
* [Random](https://docs.unity3d.com/Manual/class-Random.html): Provides you with easy ways of generating various commonly required types of random values.
* [Debug](https://docs.unity3d.com/Manual/class-Debug.html): Allows you to visualise information in the Editor that may help you understand or investigate what is going on in your project while it is running.
* [Gizmos and Handles](https://docs.unity3d.com/Manual/GizmosAndHandles.html): allows you to draw lines and shapes in the **Scene view**  
   and Game view, as well as interactive handles and controls.

# Physics

Unity helps you simulate physics in your Project to ensure that the objects correctly accelerate and respond to **collisions**  
, gravity, and various other forces. Unity provides different **physics engine** implementations which you can use according to your Project needs: 3D, 2D, object-oriented, or data-oriented. This page provides the links to their documentation.

## Built-in physics engines for object-oriented projects

If your project is object-oriented, use the Unity’s built-in physics engine that corresponds to your needs:

* [Built-in 3D physics](https://docs.unity3d.com/Manual/PhysicsOverview.html) (Nvidia PhysX engine integration)
* [Built-in 2D physics](https://docs.unity3d.com/Manual/Physics2DReference.html) (Box2D engine integration)

## Physics engine packages for data-oriented projects

# Rigidbody overview

**Rigidbodies** enable your **GameObjects**  
 to act under the control of physics. The Rigidbody can receive forces and torque to make your objects move in a realistic way. Any GameObject must contain a Rigidbody to be influenced by gravity, act under added forces via scripting, or interact with other objects through the NVIDIA PhysX **physics engine**  
.

A **Rigidbody**  
 is the main component that enables physical behaviour for a **GameObject**  
. With a Rigidbody attached, the object will immediately respond to gravity. If one or more **Collider**  
 components are also added, the **GameObject** is moved by incoming **collisions**  
.

Since a Rigidbody component takes over the movement of the GameObject it is attached to, you shouldn’t try to move it from a script by changing the [Transform](https://docs.unity3d.com/Manual/class-Transform.html) properties such as position and rotation. Instead, you should apply **forces** to push the GameObject and let the **physics engine**  
 calculate the results.

There are some cases where you might want a GameObject to have a Rigidbody without having its motion controlled by the physics engine. For example, you may want to control your character directly from script code but still allow it to be detected by triggers (see Triggers under the [Colliders](https://docs.unity3d.com/Manual/CollidersOverview.html) topic). This kind of non-physical motion produced from a script is known as kinematic motion. The Rigidbody component has a property called **Is Kinematic** which removes it from the control of the physics engine and allow it to be moved kinematically from a script. It is possible to change the value of **Is Kinematic** from a script to allow physics to be switched on and off for an object, but this comes with a performance overhead and should be used sparingly.

# Colliders

Collider components define the shape of a **GameObject**  
 for the purposes of physical **collisions**. A **collider**, which is invisible, does not need to be the exact same shape as the **GameObject**’s **mesh**  
. A rough approximation of the **mesh** is often more efficient and indistinguishable in gameplay.

The simplest (and least processor-intensive) colliders are **primitive** collider types. In 3D, these are the [Box Collider](https://docs.unity3d.com/Manual/class-BoxCollider.html)  
, [Sphere Collider](https://docs.unity3d.com/Manual/class-SphereCollider.html)  
 and [Capsule Collider](https://docs.unity3d.com/Manual/class-CapsuleCollider.html)  
. In 2D, you can use the [Box Collider 2D](https://docs.unity3d.com/Manual/class-BoxCollider2D.html) and [Circle Collider 2D](https://docs.unity3d.com/Manual/class-CircleCollider2D.html). You can add any number of these to a single GameObject to create **compound colliders**.

## Compound colliders

Compound colliders approximate the shape of a GameObject while keeping a low processor overhead. To get further flexibility, you can add additional colliders on child GameObjects. For instance, you can rotate boxes relative to the local axes of the parent GameObject. When you create a compound collider like this, you should only use one **Rigidbody**  
 component, placed on the root GameObject in the hierarchy.

Primitive colliders do not work correctly with shear transforms. If you use a combination of rotations and non-uniform scales in the Transform hierarchy so that the resulting shape is no longer a primitive shape, the primitive collider cannot represent it correctly.

## Mesh colliders

There are some cases, however, where even compound colliders are not accurate enough. In 3D, you can use [Mesh Colliders](https://docs.unity3d.com/Manual/class-MeshCollider.html)  
 to match the shape of the GameObject’s mesh exactly. In 2D, the [Polygon Collider 2D](https://docs.unity3d.com/Manual/class-PolygonCollider2D.html) does not match the shape of the **sprite**  
 graphic perfectly but you can refine the shape to any **level of detail**  
 you like.

These colliders are much more processor-intensive than primitive types, so use them sparingly to maintain good performance. Also, a mesh collider cannot collide with another mesh collider (i.e., nothing happens when they make contact). You can get around this in some cases by marking the mesh collider as **Convex** in the **Inspector**  
. This generates the collider shape as a “convex hull” which is like the original mesh but with any undercuts filled in.

The benefit of this is that a convex mesh collider can collide with other mesh colliders so you can use this feature when you have a moving character with a suitable shape. However, a good rule is to use mesh colliders for **scene**  
 geometry and approximate the shape of moving GameObjects using compound primitive colliders.

## Static colliders

You can add colliders to a GameObject without a Rigidbody component to create floors, walls and other motionless elements of a Scene. These are referred to as static colliders. At the opposite, colliders on a GameObject that has a Rigidbody are known as dynamic colliders. Static colliders can interact with dynamic colliders but since they don’t have a Rigidbody, they don’t move in response to collisions.

## Physics materials

When colliders interact, their surfaces need to simulate the properties of the material they are supposed to represent. For example, a sheet of ice will be slippery while a rubber ball will offer a lot of friction and be very bouncy. Although the shape of colliders is not deformed during collisions, their friction and bounce can be configured using **Physics Materials**. Getting the parameters just right can involve a bit of trial and error but an ice material, for example will have zero (or very low) friction and a rubber material with have high friction and near-perfect bounciness. See the reference pages for [Physic Material](https://docs.unity3d.com/Manual/class-PhysicMaterial.html) and [Physics Material 2D](https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html) for further details on the available parameters. Note that for historical reasons, the 3D asset is actually called **Physic Material**  
 (without the S) but the 2D equivalent is called **Physics Material 2D**  
 (with the S).

## Triggers

The scripting system can detect when collisions occur and initiate actions using the OnCollisionEnter function. However, you can also use the **physics engine**  
 simply to detect when one collider enters the space of another without creating a collision. A collider configured as a **Trigger** (using the **Is Trigger** property) does not behave as a solid object and will simply allow other colliders to pass through. When a collider enters its space, a trigger will call the OnTriggerEnter function on the trigger object’s **scripts**  
.

## Collision callbacks for scripts

When collisions occur, the physics engine calls functions with specific names on any scripts attached to the objects involved. You can place any code you like in these functions to respond to the collision event. For example, you might play a crash sound effect when a car bumps into an obstacle.

On the first physics update where the collision is detected, the OnCollisionEnter function is called. During updates where contact is maintained, OnCollisionStay is called and finally, OnCollisionExit indicates that contact has been broken. Trigger colliders call the analogous OnTriggerEnter, OnTriggerStay and OnTriggerExit functions. Note that for 2D physics, there are equivalent functions with **2D** appended to the name, eg, OnCollisionEnter2D. Full details of these functions and code samples can be found on the Script Reference page for the [MonoBehaviour](https://docs.unity3d.com/ScriptReference/MonoBehaviour.html) class.

With normal, non-trigger collisions, there is an additional detail that at least one of the objects involved must have a non-kinematic Rigidbody (ie, Is Kinematic must be switched off). If both objects are kinematic Rigidbodies then OnCollisionEnter, etc, will not be called. With trigger collisions, this restriction doesn’t apply and so both kinematic and non-kinematic Rigidbodies will prompt a call to OnTriggerEnter when they enter a trigger collider.

## Collider interactions

Colliders interact with each other differently depending on how their [Rigidbody components](https://docs.unity3d.com/Manual/RigidbodiesOverview.html) are configured. The three important configurations are the Static Collider (ie, no Rigidbody is attached at all), the Rigidbody Collider and the Kinematic Rigidbody Collider.

### Static Collider

A static collider is a GameObject that has a Collider but no Rigidbody. Static colliders are mostly used for level geometry which always stays at the same place and never moves around. Incoming Rigidbody objects collide with static colliders but don’t move them.

In particular cases, the physics engine optimizes for static colliders that never move. For instance, a vehicle resting on top of a static collider remains asleep even if you move this static collider. You can enable, disable, or move static colliders in runtime without specially affecting the physics engine computation speed. Also, you can safely scale a static Mesh Collider as long as the scale is uniform (not skewed).

### Rigidbody Collider

This is a GameObject with a Collider and a normal, non-kinematic Rigidbody attached. Rigidbody colliders are fully simulated by the physics engine and can react to collisions and forces applied from a script. They can collide with other objects (including static colliders) and are the most commonly used Collider configuration in games that use physics.