UNITY FOR GAMES

E-BOOK

UNITY GAMEDEV FIELD GUIDE

2020 LTS EDITION

Unity

Contents

First steps 5

Key feature overview 6

Visual Editor 6

Component-based architecture 6

C# 6

Prefab workflow 6

Unity Hub 6

Render pipelines 7

Multiplatform development 7

Packages and extensibility 7

Worldbuilding 8

Active community 8

Unity Hub 9

Sample projects 11

Learn projects 11

Unity Asset Store assets 13

Unity GitHub repository 13

Editor interface 15

The main windows 15

Editor Play mode 17

Version control 18

Plastic SCM 19

Git 20

Perforce 21

Project organization 22

The Project view 22

Folder structure and naming 25

Scenes 25

The Hierarchy window 27

General tips for scenes and hierarchies 27

Naming standards 28

GameObjects and components **29**

The Inspector 30

The Transform component 31

Tips for working with transforms 31

Built-in components 32

Custom components 33

Prefabs **34**

Prefab usage 34

Creating Prefab assets and instances 34

Prefab mode 35

Prefab Variants and Nested Prefabs. 36

Package Manager **37**

Programming **38**

Custom components and Monobehaviours 38

Initializing objects. 39

Monobehaviour lifecycle and structure 40

More scripting tips 42

Common classes 43

Memory management 44

Value versus reference types. 44

Garbage collection 44

Multithreading: C# Job System and Burst compiler .46

Scripting backends in Unity. 47

Editor scripting 47

Odin Inspector and Serializer 48

Integrated development environment (IDE) support. 50

Script templates 50

Building and publishing **51**

Device Simulator. 53

Input **54**

Input System. 54

Built-in Input class 54

User interface **56**

Unity GUI. 56

Immediate Mode GUI 57

Profiling and optimization **58**

Unity Profiler 58

Memory Profiler 59

Profile Analyzer. 60

Debugging and playtesting **61**

Debugging gameplay 61

Additional debugging tips. 62

Unity Test Framework 63

Particle effects **64**

Physics **66**

3D physics 66

2D physics. 68

Audio **69**

Render pipelines and graphics **70**

3D pipelines 70

2D pipeline 73

Worldbuilding **74**

ProBuilder and Polybrush 74

Terrain tools 74

2D Tilemap and Sprite Shape 75

Unity Asset Store **76**

Importing an asset 76

Become a publisher 76

Learning resources **77**

Documentation 77

Unity Learn 77

Unity Blog 77

Professional training. 77

Next steps **78**

First steps

Shipping a game on a new platform and engine may seem like a journey of a thousand miles, but, like all worthwhile pursuits, it begins with a single step. And like every other journey, it helps to have a map to guide you.

As an industry veteran, you've faced the kinds of obstacles we'll cover here before: design and planning, development and testing, then launch and maintenance.

We've created this guide for experienced developers making the transition to Unity or returning from a long hiatus. Whether you're a solo indie developer or an established game studio, the Unity core platform can do the heavy lifting for you. This lets you focus on what you do best: building immersive and interactive real-time experiences.

Unity is better than ever, and we'd like to invite you to discover a host of new features, designed to make game development faster, more efficient, and more fun than ever before.

Over 1.5 million monthly active creators choose Unity to make amazing games and publish them to a wide range of devices. Our highly extensible platform brings artists, designers, and developers together.

This guide will help you jump-start your familiarity with Unity's rich feature set and intuitive workflows. Whether you're developing an action RPG for PC and console or the latest physics puzzler for mobile, Unity is here to help.

Key feature overview

Visual Editor

The Unity Editor includes a comprehensive interface that will help you iterate rapidly through cycles of prototyping and testing to quickly transform your proof of concept into a working build.

Component-based architecture

Everything in a Unity Scene is a **GameObject**. You'll combine these with **components**, modular and reusable parts that you can mix and match for functionality. A camera in Unity is simply a GameObject with a Camera Component. A light is a GameObject with a Light Component. This component-based system is simple, flexible, and highly scalable.

C#

Unity uses C#, an industry standard and one of the most widely used programming languages in the world. C# is an object-oriented and type-safe language with an extensive community and enterprise support.

If you're already familiar with C-based languages or Java, you'll have a low learning curve to get started in Unity scripting. If the Unity Editor doesn't fit your project requirements out-of-the-box, you can write your own scripted components that will work seamlessly with their built-in counterparts.

Unity includes support for a number of IDEs, including Visual Studio, VS Code, and JetBrains Rider. See Integrated development environment (IDE) support below for details.

Prefab workflow

Don't build something twice if you can build it once and reuse it – this will save you time and get better runtime performance. With **Prefabs**, you can set up GameObjects as assets in a library, then instance them at runtime. This workflow includes Variants, where you can "subclass" and override part of your "template" Prefab. You can also build complex Nested Prefabs out of smaller Prefabs, enabling you to propagate changes into your scene as soon as you make them.

Unity Hub

The Unity Hub is a front-end application that helps you install Unity and manage your engine versions, licensing, and projects. It ties your projects to specific versions of Unity, so that an engine update won't impede production or create local conflicts. The Hub also gives you easy access to valuable training materials and a vibrant Unity community.

Render pipelines

In Unity, you can choose between different prebuilt render pipelines to help you draw your graphics onscreen. The **Universal Render Pipeline (URP)** focuses on wide platform compatibility and maximum performance, while the **High Definition Render Pipeline (HDRP)** adds high-fidelity visuals for PCs and consoles. You can also create shaders either with HLSL or using the node-based **Shader Graph** tool, which doesn't require coding.

The HDRP is designed to create high-fidelity visuals.

Multiplatform development

Unity is guided by the ideal of helping developers build once, deploy anywhere. Our goal is to help your work reach the widest possible audience as you evolve your IP with the industry. We've enabled thousands of studios to achieve this, including Unknown Worlds, creators of the *Subnautica* series.

After developing your content in the Editor, you can deploy it across over 20 platforms, including PlayStation, Xbox, Nintendo Switch, PC, and mobile. With Unity, you can even create builds for WebGL and XR platforms like Oculus Quest.

Packages and extensibility

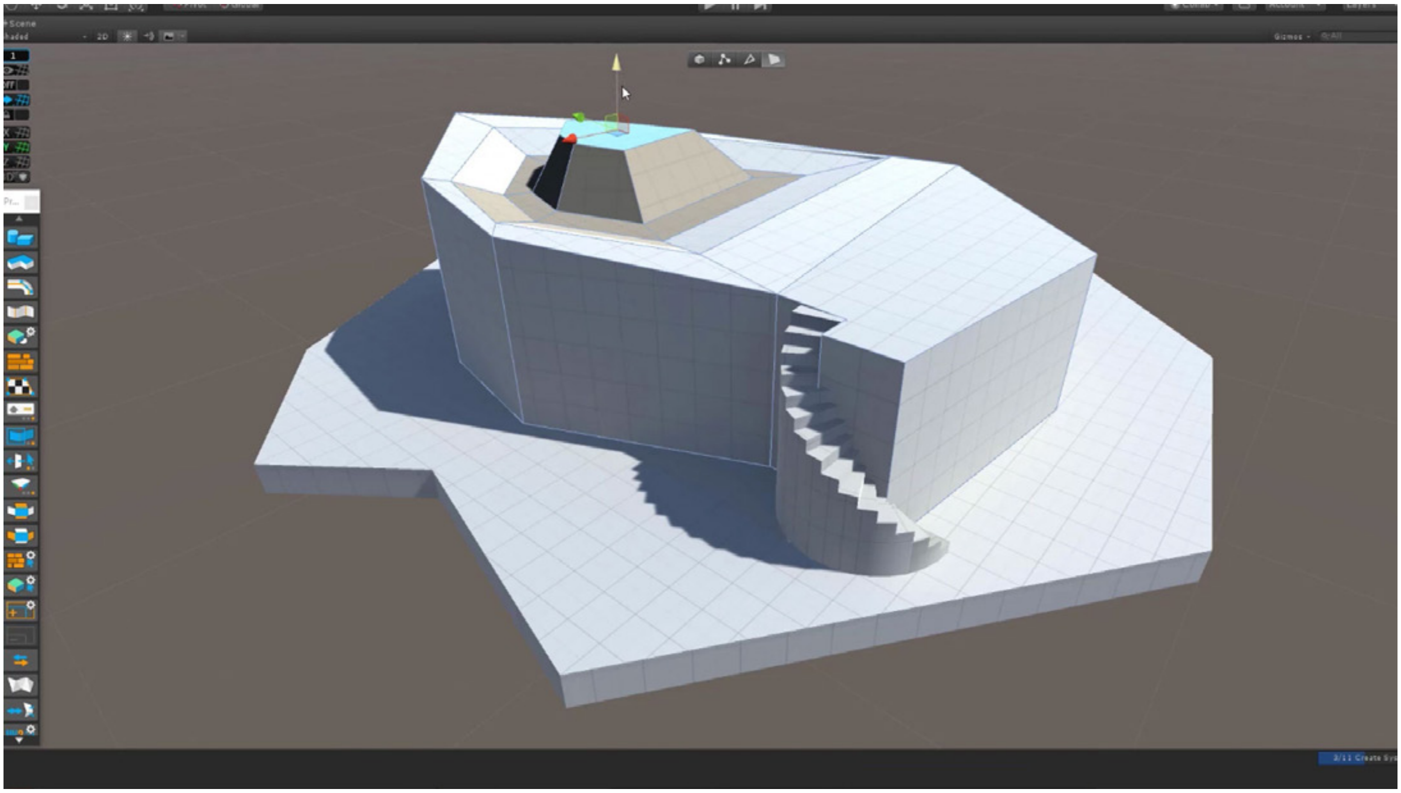
The **Package Manager** offers the ability to discover and update features independently of the Editor. Unity maintains a content repository of packages that extend its core functionality.

Keep your project lean and only import what you need. Packages allow you to explore Preview or Experimental features before they become production-ready, so you can plan ahead for new technologies that you might want to incorporate into future projects.

Worldbuilding

Unity includes a suite of tools for constructing environments. **ProBuilder**, a unique hybrid of 3D modeling and level-design tools, lets you model simple geometry and whitebox your game levels.

If you're working in **2D**, Unity includes tools such as SpriteShape and Tilemap to help you layout your 2D environments. Environment artists can generate natural landscapes using terrains, which you can paint using the included brush tools.



ProBuilder helps you prototype your levels.

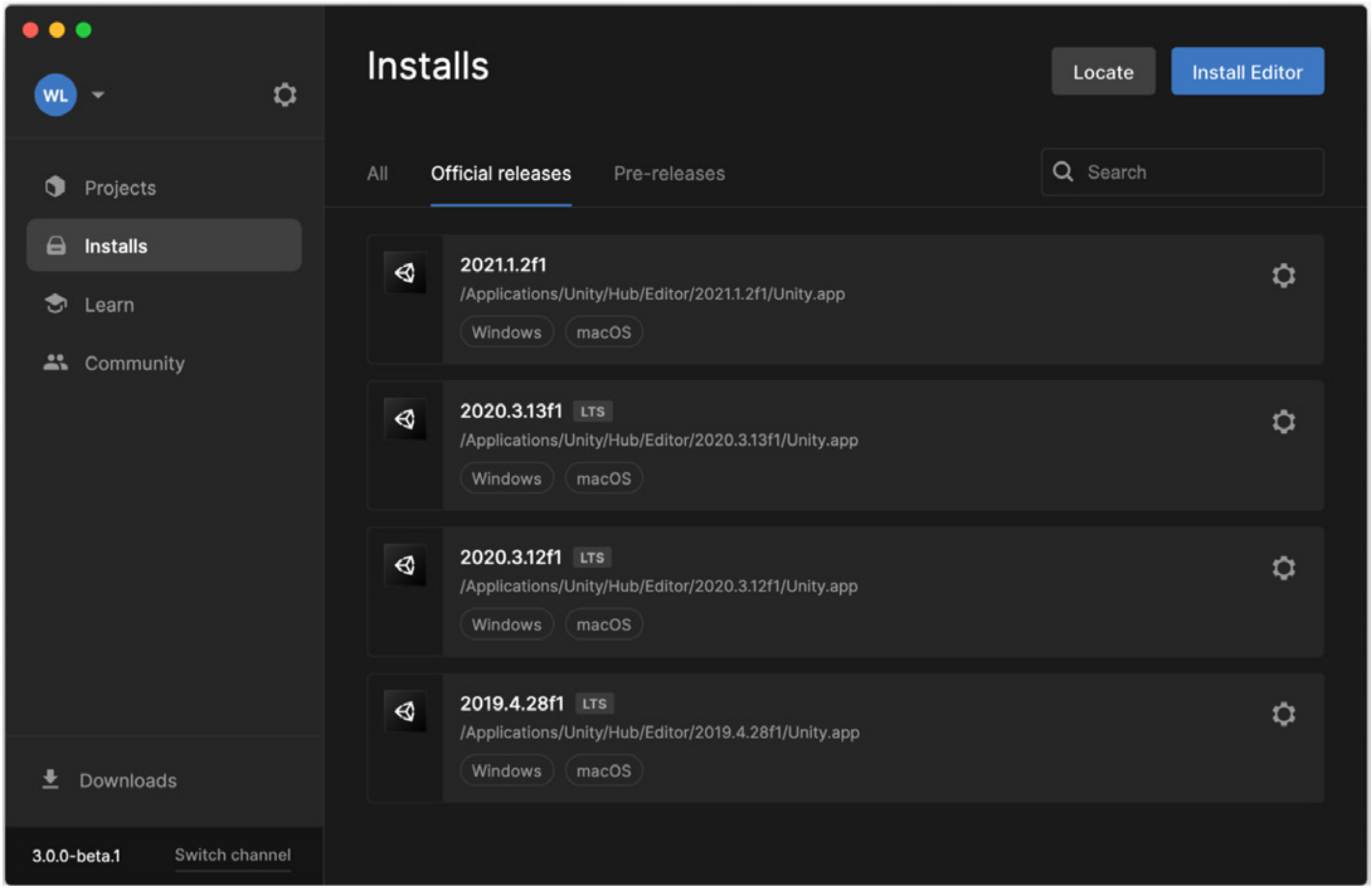
Active community

Unity has a broad reach and active user base. Developers used the Unity core platform to create and monetize over half of the top 1,000 mobile games on the Apple App Store and Google Play. With over a million active monthly creators, you can tap into an extensive knowledge base about development issues on the official Unity forums and other online communities.

Unity Hub

Begin exploring Unity with the Unity Hub, a tool that manages all of your Unity projects and installations. Use the Hub to install one or more versions of the Unity Editor, create new projects, or open existing ones.

To get the Hub, visit the Unity website and select **Download Hub**. Use the interface to install Unity, and activate an individual or team license before launching the engine. Unity Plus and Pro licenses also require a valid serial number.



2021.1.2f1
/Applications/Unity/Hub/Editor/2021.1.2f1/Unity.app
Windows macOS

2020.3.13f1 LTS
/Applications/Unity/Hub/Editor/2020.3.13f1/Unity.app
Windows macOS

2020.3.12f1 LTS
/Applications/Unity/Hub/Editor/2020.3.12f1/Unity.app
Windows macOS

2019.4.28f1 LTS
/Applications/Unity/Hub/Editor/2019.4.28f1/Unity.app
Windows macOS

The Unity Hub helps you manage projects and installed versions.

In the Unity Hub, navigate through the Projects, Installs, Learn, and Community screens.

The **Installs** screen lets you selectively install various versions of Unity, including both pre-releases and official releases. Add only the modules you need for fine-level control of platform-specific support or dev tools.

We recommend that you use the current Long Term Support, or LTS, version of Unity for production work. Unity 2020.3 LTS provides:

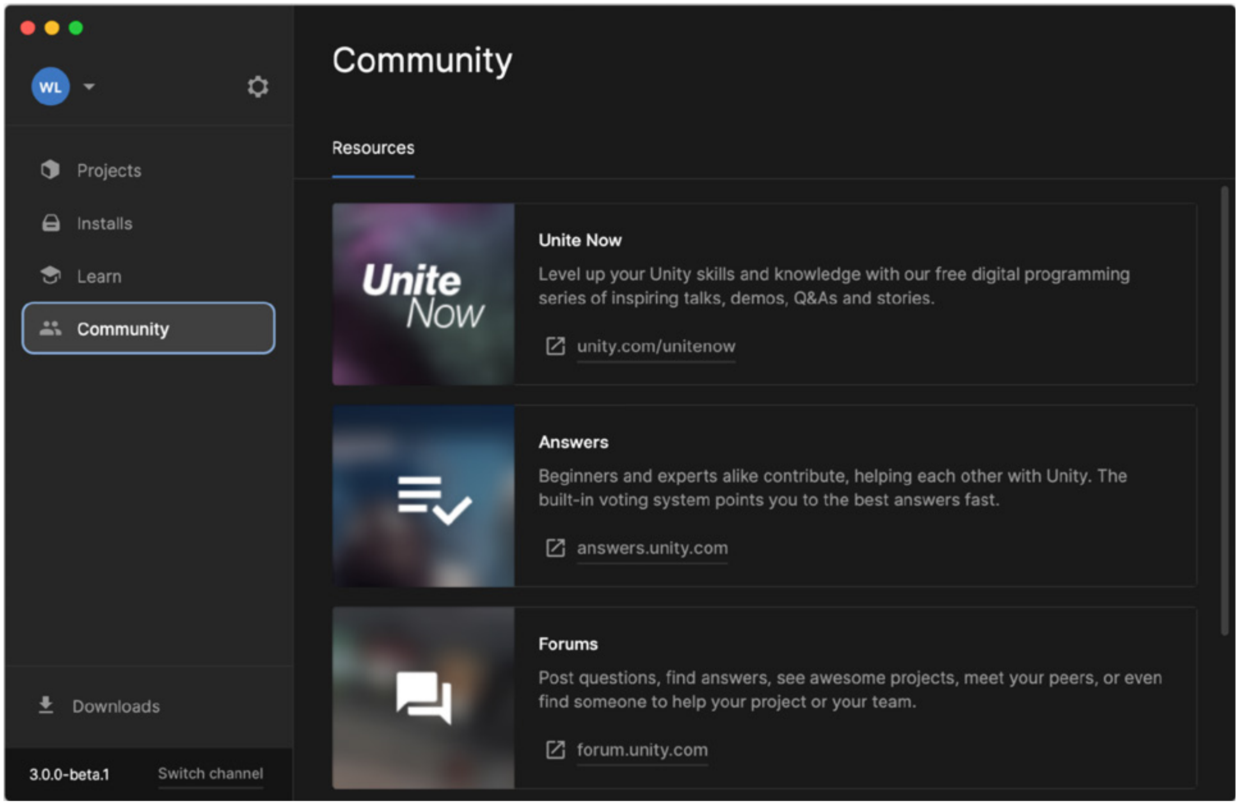
- a stable foundation for projects that are about to ship
- biweekly updates until mid-2022 with monthly updates thereafter until March 2023 (two years after the initial release date)

Updates to LTS versions only cover usability fixes aimed at improving the engine's stability.

Active projects appear in the **Projects** screen. Associate a specific target platform with each project and lock development to a specific Unity version to prevent an update from introducing work-stopping bugs.

The **Learn** screen gives you access to numerous educational resources and tutorials to help you get acclimated to the engine.

Follow the **Community** screen for other commonly available resources, such as the Unity Blog and Unite Now talks. If you have a technical question, you can ask a question or read a discussion on the Answers or Forums pages.



Community

Resources

Unite Now

Level up your Unity skills and knowledge with our free digital programming series of inspiring talks, demos, Q&As and stories.

unity.com/unitenow

Answers

Beginners and experts alike contribute, helping each other with Unity. The built-in voting system points you to the best answers fast.

answers.unity.com

Forums

Post questions, find answers, see awesome projects, meet your peers, or even find someone to help your project or your team.

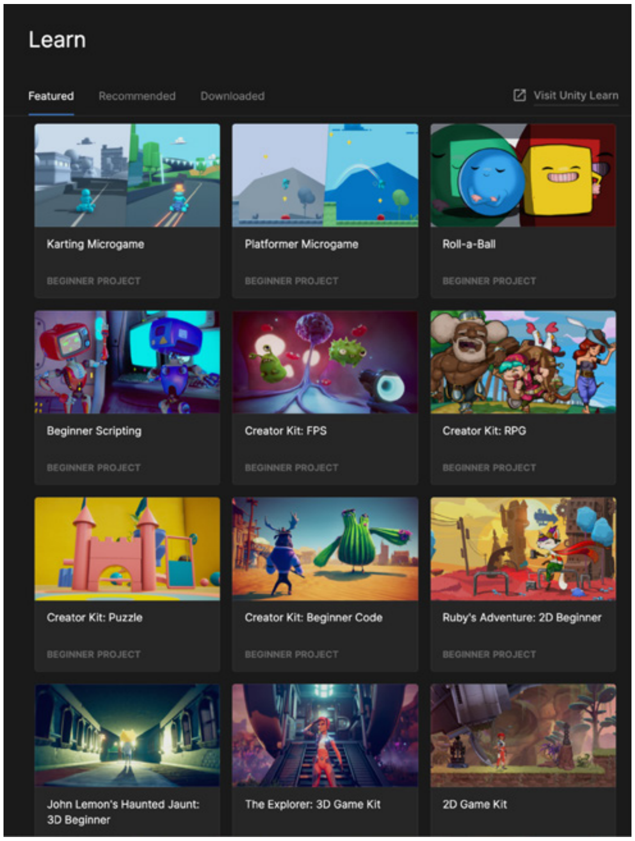
forum.unity.com

The Community screen links to several online resources.

Sample projects

Probably the best way to familiarize yourself with the development environment is to peek into a vertical slice of a game prototype. You can find these in a number of places to kickstart your exploration of Unity.

Learn projects
The Learn screen on the Hub gives you access to a variety of tutorials and learning resources, including numerous example projects that you can download and import directly into Unity.



Learn

Featured Recommended Downloaded Visit Unity Learn

Karting Microgame
BEGINNER PROJECT

Platformer Microgame
BEGINNER PROJECT

Roll-a-Ball
BEGINNER PROJECT

Beginner Scripting
BEGINNER PROJECT

Creator Kit: FPS
BEGINNER PROJECT

Creator Kit: RPG
BEGINNER PROJECT

Creator Kit: Puzzle
BEGINNER PROJECT

Creator Kit: Beginner Code
BEGINNER PROJECT

Ruby's Adventure: 2D Beginner
BEGINNER PROJECT

John Lemon's Haunted Jaunt: 3D Beginner

The Explorer: 3D Game Kit

2D Game Kit

Explore starter projects on the Learn screen.

Think of the Creator Kits and Microgames as mini-templates with documented code samples. While many of these projects are “code-free” for beginners, more experienced developers can dissect their scripts and assets to learn. Exploring and modifying Learn projects is a good way to get familiar with common production techniques in Unity development.

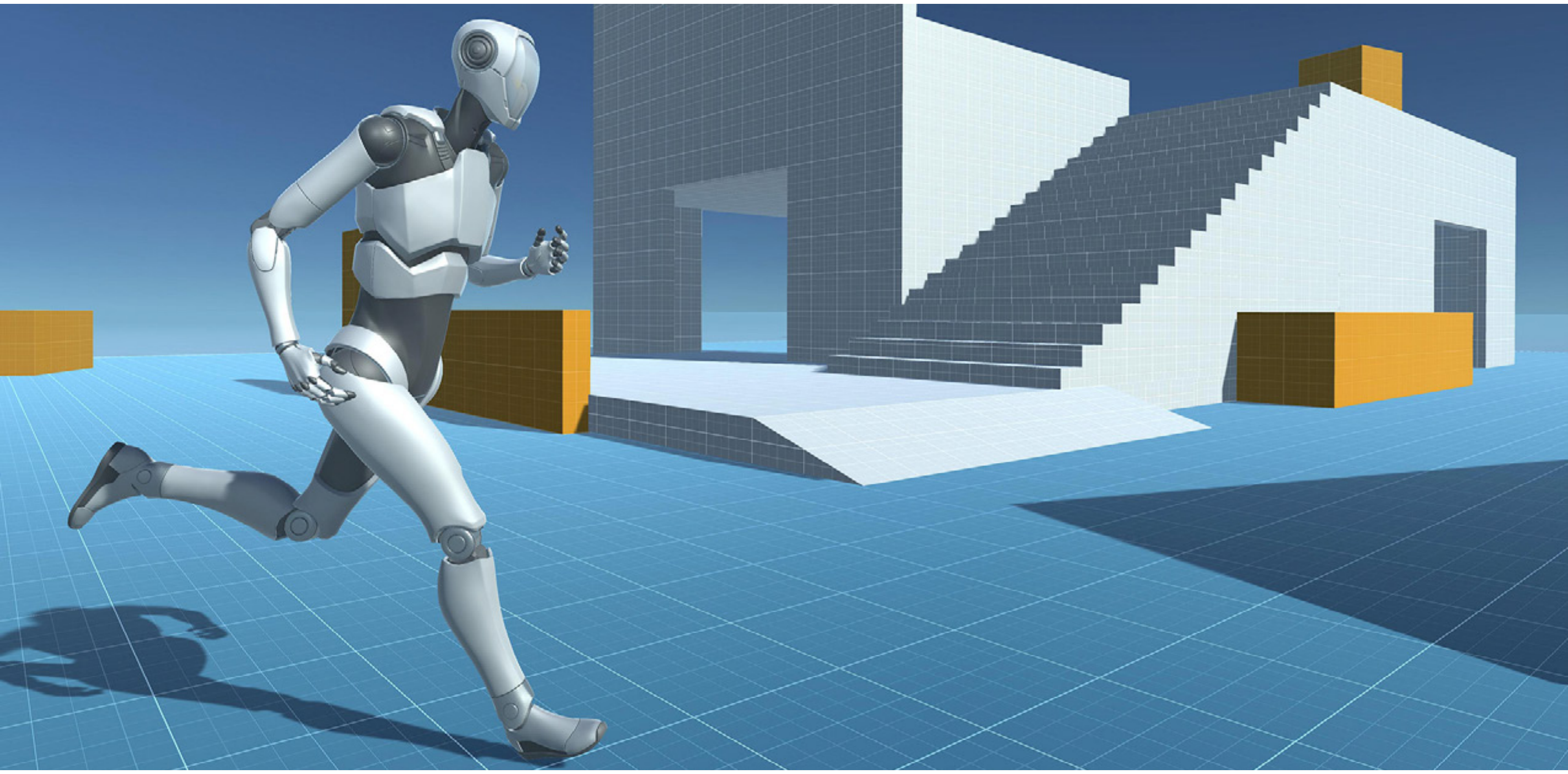
Unity gives you the option to save the modified project, then find it in Unity Hub to continue working on it later. The original, unmodified projects will remain available from the Learn screen.

Unity Asset Store assets

The Unity Asset Store is a community marketplace filled with free and premium assets. You can find third-party tools and art assets for almost anything your team doesn't want to develop in-house. Assets range from production and prototyping art to Editor extensions and scripting tools to whole projects nearly ready to ship. Many asset packages include extensive templates that illustrate essential aspects of Unity development and can save days or weeks of dev work.

The Asset Store includes tens of thousands of assets, with more packages added each day. These packages from Unity offer a good entry point if you're just getting started:

— The Starter Assets package includes free and lightweight first- and third-person character base controllers that demonstrate how to control the camera using the Cinemachine and the Input System packages. Starter Assets – First Person Controller and Starter Assets – Third Person Controller show you how to prototype a character controller on a playground scene.



Starter Assets – Third Person Controller

— The *Lost Crypt* – 2D Sample Project is a 2D side-scrolling demo that showcases Unity's 2D-specific tools working in unison. The project walks you through features such as 2D Animation, 2D Sprite Shape, 2D Tilemaps, 2D Lights, Shader Graph for 2D, Secondary Textures, and Volume post-processing. See this blog post to learn more about the project.

Lost Crypt demonstrates Unity's 2D features.

If you're new to Unity, also check out the Asset Store team's top picks for new users.

Unity GitHub repository

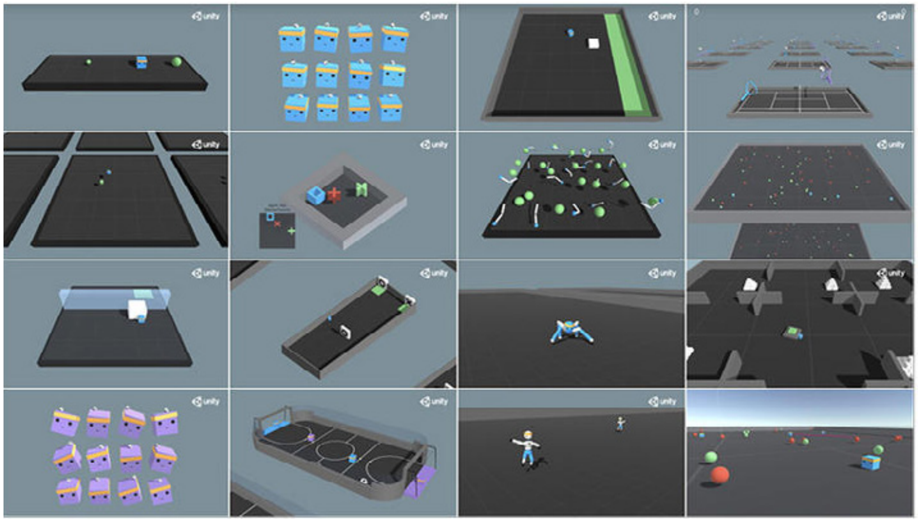
The Unity Technologies GitHub repository has a number of projects demonstrating specific features or topics. Not all projects run with the latest version of Unity, but you can use the Hub to install the matching Unity version. Many repositories are still Experimental or in Preview, so you can evaluate new features for your next production.

Here are few notable projects:

— *Boss Room* is a small-scale cooperative game sample project built on top of the new experimental netcode library and the MLAPI. It's designed to help you explore the concepts and patterns behind a multiplayer game flow with access to one dungeon level with four character classes to work on gameplay battling stylized enemies and the eponymous boss.

Boss Room

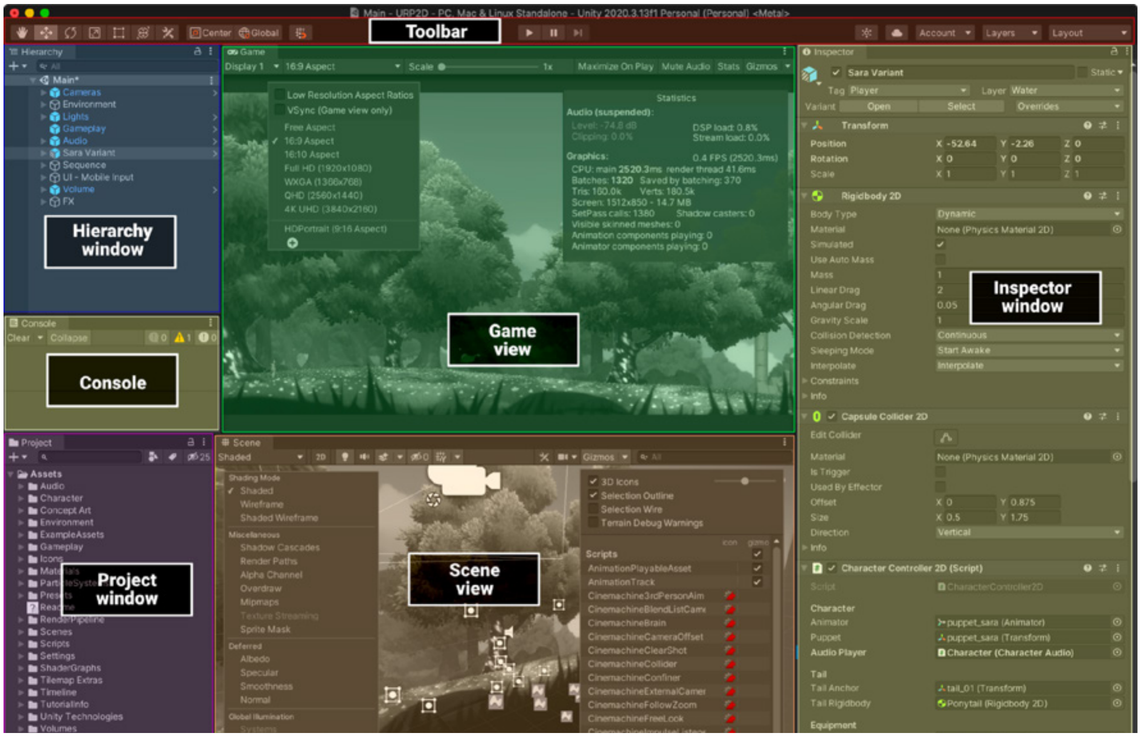
— The Unity Machine Learning Agents Toolkit (ML-Agents) is an open source project that enables games and simulations to train intelligent agents for 2D, 3D, and VR/AR applications. Researchers can also use the provided Python API to apply reinforcement learning, imitation learning, or other machine learning methods to in-game actors.



ML-Agents toolkit

Editor interface

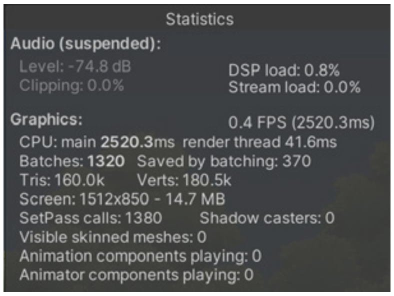
Unity has a flexible, modular user interface. If you've never used the Unity Editor before, refer to this quick overview of the main windows.



The Editor's main windows

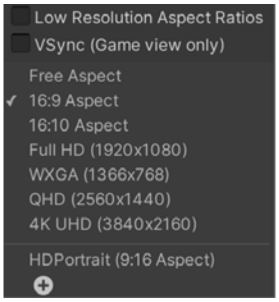
The main windows

Game view: This previews the game's main camera and how a player will interact with the game. This includes the Rendering statistics window in the top right, which shows graphics and audio stats to help you optimize your application. In the top left, Aspect Ratio allows you to preview multiple screen resolutions or define your own target resolution.



```
Statistics
Audio (suspended):
  Level: -74.8 dB              DSP load: 0.8%
  Clipping: 0.0%               Stream load: 0.0%

Graphics:                      0.4 FPS (2520.3ms)
  CPU: main 2520.3ms  render thread 41.6ms
  Batches: 1320   Saved by batching: 370
  Tris: 160.0k      Verts: 180.5k
  Screen: 1512x850 - 14.7 MB
  SetPass calls: 1380     Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Rendering statistics



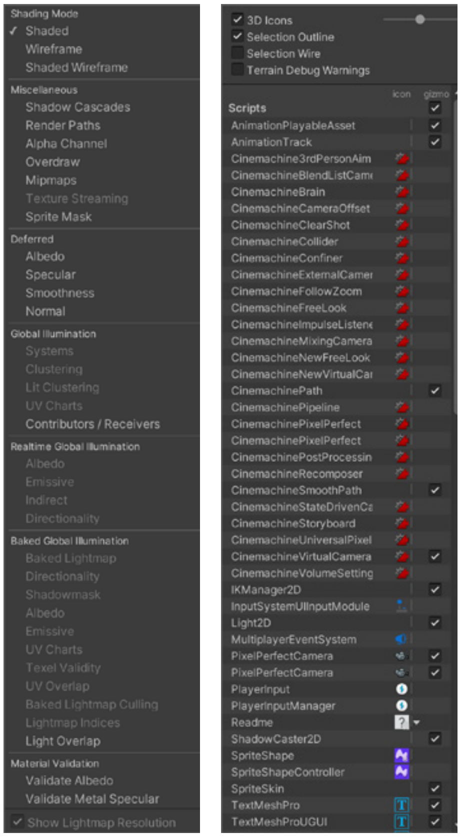
□ Low Resolution Aspect Ratios
□ VSync (Game view only)

Free Aspect
✓ 16:9 Aspect
16:10 Aspect
Full HD (1920x1080)
WXGA (1366x768)
QHD (2560x1440)
4K UHD (3840x2160)

HDPortrait (9:16 Aspect)
⊕

Aspect ratio

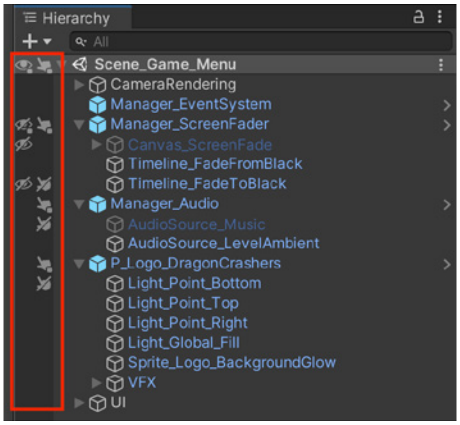
Scene view: This view shows the objects that comprise each scene. You will primarily work here in the Editor to place models, lights, physics bodies, and so on. Use the control bar to change the Draw Mode, useful for troubleshooting and quick diagnostics. The Gizmos menu controls icons and overlay graphics for certain GameObjects.



Draw Mode and Gizmos menus

Hierarchy window: This window shows objects in the currently active scene and how they relate to one another in parent-child relationships.

The Hierarchy includes a useful interface to override the visibility and pickability of objects in the Scene view. This UI does not affect the Game view. It's designed to help you navigate complex scenes with a large number of GameObjects.



Scene visibility and pickability overrides Scene view

Inspector window: This shows properties and settings for the selected objects and offers powerful options for customization. Focused Inspectors can help you open an Inspector window for any specific GameObject, asset, or component to simplify comparison of gameplay variables or scene objects. Inspectors allow you to adjust them at runtime without needing to go into the code each time, making playtesting more interactive and facilitating a faster workflow.

Project window: Unity stores all assets that create the game (Prefabs, Textures, models, animations, scripts) as files on disk. The Project window gives you easy access to these files – think of this as a content directory or library where your in-game resources reside. Use the Search bar to locate specific assets or types.

Toolbar: This appears at the top of the Editor, locked to the Application window's title bar. Use the buttons here to control the Editor Play mode and see how your published application plays. The Hand tool pans the scene, while the Move/Rotate/Scale/Rect Transform/Transform tools allow you to manipulate GameObjects.

The Console window: The Console window is used for debugging and shows errors, warnings, and other messages generated by Unity (see the Debug class). Everything written to the Console window, either by Unity or by your own code, also outputs to a Log File.

Editor Play mode

The Game view is rendered from the Camera(s) in your application. It represents your final, published application. In Play mode, any changes you make are temporary, and they are reset when you exit Play mode. The Editor UI darkens to remind you that any changes you make in Play mode are not saved.

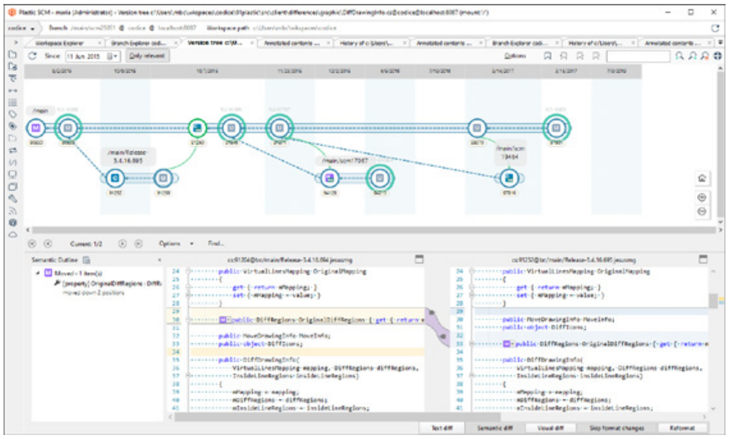
Remember that you will need to use one or more Cameras to control what the player sees at runtime. For more information, see the Camera component page.

For more specific information about each part of the interface, Unity Learn also includes an overview.

Version control

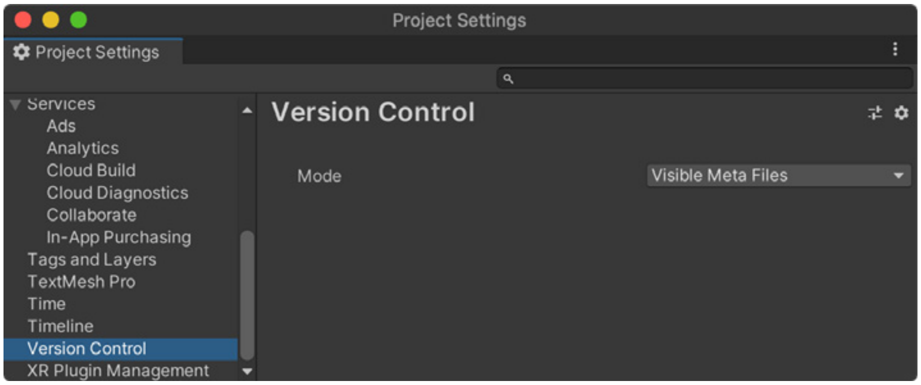
Whether it's Plastic, Git, Perforce, or another system, Unity allows you to choose which source control solution works best for you and your team.

Unity has in-Editor integrations with two industry-leading version control systems, Perforce and Plastic SCM. You must have either a Perforce or Plastic SCM server set up for your project to use with Unity.



PlasticSCM makes version control simple with its streamlined interface.

Before using source control, navigate to **Project Settings > Editor**. Confirm that **Asset Serialization Mode** is set to **Force Text**. Unity uses serialization to load and save Assets to and from disk. Force Text means that Unity will store the scene files in a text-based format to help with version control merges.



Version Control

Mode: Visible Meta Files

Version Control in the Project Settings

Navigate to **Project Settings > Editor > Version Control**. According to your version control system, switch to the mode **Plastic SCM**, **Perforce**, or **Visible Meta Files** (for external source control like Git).

Checkin 3 of 3 item(s)

Item	Status	Size	Extension	Type	Date modified	Repository
Changed items - 3 of 3 item(s) selected						
Assets\Timeline\Sequence_ArtefactExplosion.prefab +meta	Checked-out (changed)	33.29 KB	.prefab	txt	44 seconds ago (7/28/2020 7:22:27 PM)	LostCrypt@unity@cloud
Assets\Scenes\Main.unity +meta	Checked-out (changed)	342.53 KB	.unity	txt	1 minutes ago (7/28/2020 7:21:43 PM)	LostCrypt@unity@cloud
Assets\Environment\Animations\AnimClip_Artefact_Oscillate.anim +meta	Checked-out (changed)	42.29 KB	.anim	txt	13 seconds ago (7/28/2020 7:22:58 PM)	LostCrypt@unity@cloud

Plastic SCM

Plastic SCM

Plastic SCM is our recommended version control system for Unity game development. This solution offers the best experience when dealing with large binary files (>500 MB), so your art assets can have the same level of management as you would expect for your code.

Plastic SCM allows you to work knowing that both your art and programming assets are securely backed up. In addition, the intuitive visual interface simplifies branching and versioning.

Here are some of the key benefits of Plastic SCM:

— **Speed:** Creating branches in a large codebase is significantly faster in Plastic than in Perforce or Git.

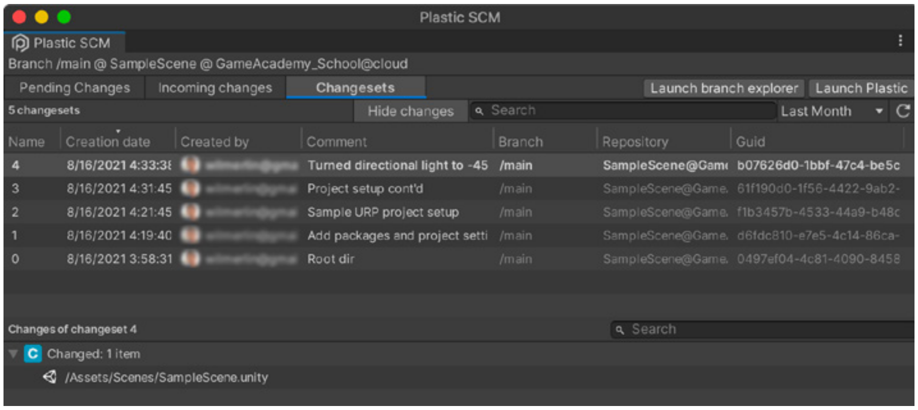
— **Simplicity for non-coders:** Artists on your team can work in Gluon mode. This simplified workflow allows them to check out their part of the project, work, and commit their changes, while removing irrelevant parts of the interface.

— **Cloud hosting:** Plastic offers a native solution called Plastic Cloud Edition, designed with fully distributed teams in mind. If your team doesn't want to consider running its own servers and prefers hosting everything online, try Plastic Cloud Edition.

— **Distributed Version Control:** Set up different repositories in remote offices so teams always work "locally." This is a Git-like workflow at a larger scale. Use the graphical interface to push and pull changes as well as solve remote conflicts.

— **Diff window:** Plastic features a complete interface to view changed, added, or deleted files. The GUI also includes a separate image diff tool to help you compare two revisions of a texture asset.

When using Plastic SCM, open the **Plastic SCM** window (**Window > Plastic SCM**) to view the files in your changelist.



Branch /main @ SampleScene @ GameAcademy_School@cloud

5 changesets

Name	Creation date	Created by	Comment	Branch	Repository	Guid
4	8/16/2021 4:33:3ε	[redacted]	Turned directional light to -45	/main	SampleScene@Gam	b07626d0-1bbf-47c4-be5c
3	8/16/2021 4:31:45	[redacted]	Project setup cont'd	/main	SampleScene@Game.	61f190d0-1f56-4422-9ab2-
2	8/16/2021 4:21:45	[redacted]	Sample URP project setup	/main	SampleScene@Game.	f1b3457b-4533-44a9-b48c
1	8/16/2021 4:19:40	[redacted]	Add packages and project setti	/main	SampleScene@Game.	d6fdc810-e7e5-4c14-86ca-
0	8/16/2021 3:58:31	[redacted]	Root dir	/main	SampleScene@Game.	0497ef04-4c81-4090-8458

Changes of changeset 4

C Changed: 1 item

/Assets/Scenes/SampleScene.unity

PlasticSCM window

The **Pending changes** tab lists all of the local changes that are pending a commit into version control. The **Incoming changes** tab allows you to view all incoming changes and conflicts and update your local project. Any changes made to your project prompts an "Incoming changes" notification at the top right of the Plastic SCM window.

Refer to the Version control integrations documentation for more information about the Version Control Window. To learn more, watch the Version Control for games with Plastic SCM video from Unite Now, or check out the Plastic SCM Book to get started.

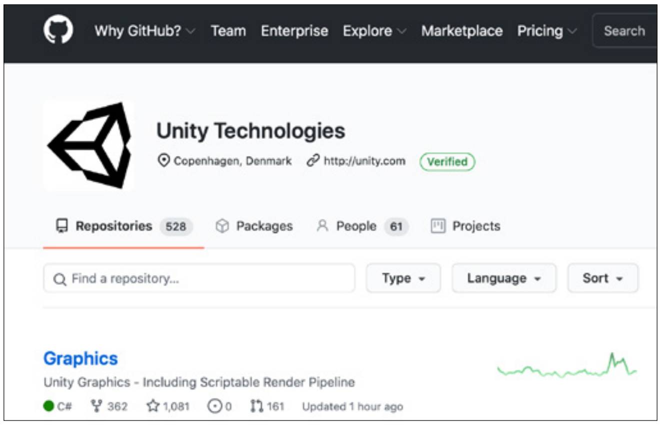
Git

Unity developers can also use external source control solutions such as Git. This, however, requires some initial manual setup of the project.

To use Git with Unity, create an empty repository on your local machine and optionally sync this to the cloud via GitHub. Be sure to include Git LFS (Large File Support) for more efficient version control of your larger assets, like graphics and sound resources. Unity maintains a .gitignore file that can help you decide what should and shouldn't go into the Git repository.

For the added convenience of working with the GitHub hosting service, install the GitHub for Unity plug-in. This open source extension allows you to view your project history, experiment in branches, commit your changes, and push your code to GitHub without leaving Unity.

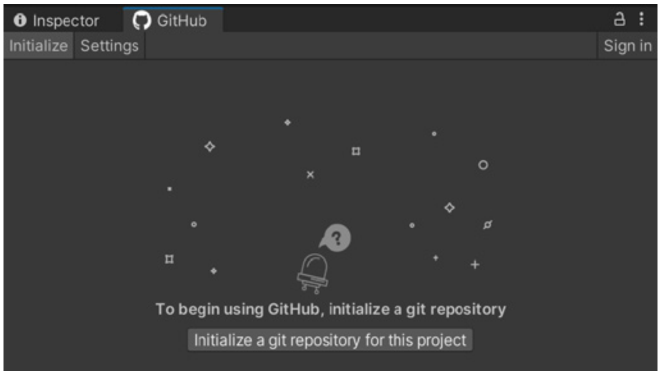
Also, consider any of the capable visual clients like GitKraken, SourceTree, or GitHub Desktop.



Unity Graphics - Including Scriptable Render Pipeline

Create a repository using GitHub.

For a walkthrough of setting up Unity with Git, watch the Introduction to Version Control video, which covers the basics of using a visual GitHub client with a sample project.



To begin using GitHub, initialize a git repository

The GitHub for Unity extension

Perforce

If Perforce is your source control of choice, use Helix Core to manage your code, artwork, and game engine assets. Helix Core is free for up to five users and 20 workspaces.

Perforce has good performance, even with remote teams distributed around the world. Many AAA or indie game dev studios use Perforce as their primary source control.

To use it:

- Follow the setup process described in the Unity Version Control documentation.
- Read how to set up Perforce Helix Core with Unity.
- Consult the Perforce documentation for more detail about Perforce Helix Core.

Project organization

As your project grows, you will need to maintain a level of organization so that it can scale with your team and application requirements. These general tips will help you to establish your basic project and scene structure.

The Project view

The Project window displays all of the files related to your project. This is the content directory where you will find assets and other project files in your application.

Unity stores the source files directly in the project, alongside individual .meta files. Meta files contain engine- and Editor-specific data for the associated asset.

Unity also imports each asset into an optimized format which the engine uses at runtime. These processed assets appear in the Library folder, which serves as a cache and does not need to be added to source control.

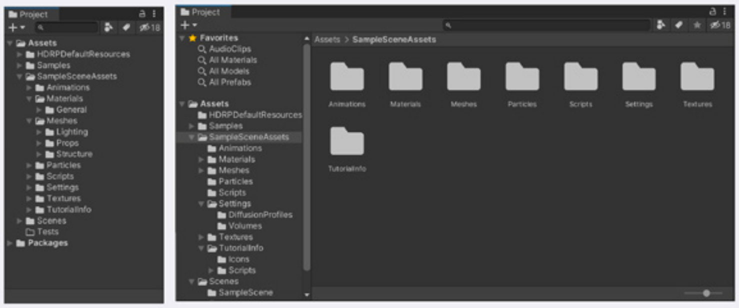
The project window has a few UI features to assist with navigation:

— **Right-click** to reveal the context menu for frequently used commands (creating/importing assets, revealing full path on disk, etc.).

— Use the **Search** field to locate assets as your project grows in size. If you're looking for a particular type of asset, filter by type using the t: syntax (e.g., **t:Material** will filter for all material assets in the project). This can help you to navigate large projects.

— Drag a frequently used folder into the **Favorites** field at the top of the interface. You can also save a search to the Favorites with the **Save Search** button.

— You can also change the layout of the window itself. Select the **More Items** (⋮) menu in the top right of the window, and choose from either **One Column Layout** or **Two Column Layout**. The two-column layout has an extra pane with a visual preview of each file.



one-column layout

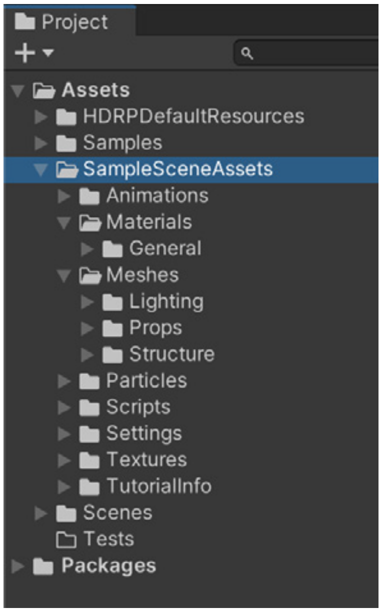
two-column layout

One-column vs two-column layout

Inside of the Project window is the **Assets folder**. This contains the assets used to build your game. If you've started your project with a template, you should see subfolders that represent several common assets. While most of these are user-defined, Unity does reserve a few folder names for specific purposes. Make sure you are aware of this list of Special folder names.

The following are some common subdirectories that you might use to organize your project, although these vary by team and project according to preferences. Above all, stay consistent – create a style guide and follow it.

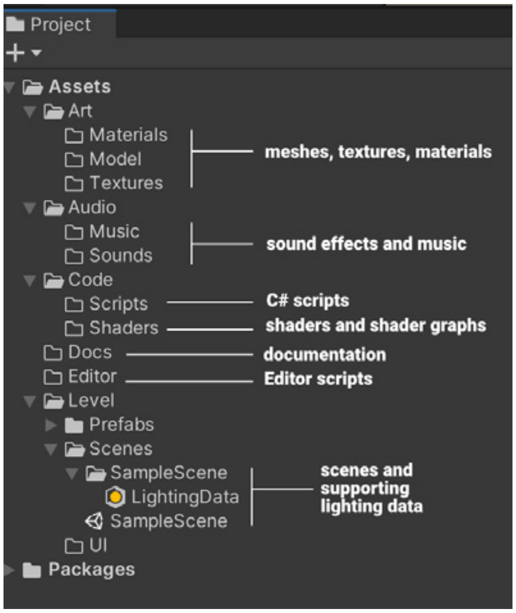
Animations	This folder contains animated motion clips and their controller files, as well as Timeline assets for in-game cinematics or rigging information for procedural animation.
Audio	Sound assets include audio clips as well as the mixers used for blending effects and music.
Editor	This folder contains scripted tools made for use with the Unity Editor but not appearing in a target build.
Fonts	The fonts used in the game.
Gizmos	Having a Gizmo icon can help visualize a GameObject in the Scene or Game view, especially if it does not have a mesh. Store the image files for these icons in the Gizmos folder.
Materials	These assets describe surface shading properties.
Meshes	Store models created in an external DCC application here.
Particles	Manage particle simulations in Unity, created either with the ParticleSystem or Visual Effect Graph.
Prefabs	These are reusable GameObjects with prebuilt components. Add them to a scene to build your levels and gameplay.
Scripts	All user-developed code for gameplay appears here.
Scenes	Unity stores small, functional portions of your project in scene assets. They often correspond to game levels or part of a level.
Settings	Assets store render pipeline settings for both HDRP and URP.
Shaders	These programs run on the GPU as part of the graphics pipeline.
Textures	Image files can consist of texture files for materials and surfacing, UI overlay elements for user interface, and lightmaps to store lighting information.
ThirdParty	If you have assets from an external source like the Asset Store, keep them separated from the rest of your project here. This makes updating your third-party assets and scripts easier. Third-party assets may have a set structure that cannot be altered.



The Sample Scene with the HDRP template includes several asset folders.

The Supported Asset Types manual page describes the most common assets in more detail. You can use the Template or Learn projects as an example of how to organize your folders effectively. While you're not limited to these folder names, this list should give you a good starting point that you can expand upon as your project scales up.

You are of course free to adapt the folder structure to your specific project's needs and team preferences, as in the image below.



Organize folders for your project needs, but stay consistent once you decide on a structure.

Folder structure and naming

While there’s no single way to organize your project, we recommend that you follow these best practices in general:

- **Document your naming conventions and folder structure**. A style guide and/or project template makes your files easier to find and organize.
- **Whatever naming convention you choose, make sure you remain consistent**. Don’t deviate from your chosen style guide or template. If you do need to amend your naming rules, parse and rename your affected assets all at once with a script.
- **Don’t use spaces in file and folder names**. Unity’s command line tools have issues with path names that have spaces.
- **Separate your testing or sandbox areas**. Create a separate folder for non-production scenes and experimentation. Subfolders with usernames can divide your work area by team member.
- **Avoid extra folders at the root leve**l. In general, store your content files within the Assets folder. Don’t create additional folders at the project’s root level unless absolutely necessary.

Scenes

Scenes are where you work with content in Unity. They contain the objects of your game and can be used to create a main menu, individual levels, and anything else. In each unique scene, you will place the environments, obstacles, and decorations that roughly translate into one level of your game. This enables you to design and build your application piece by piece, keeping it modular.

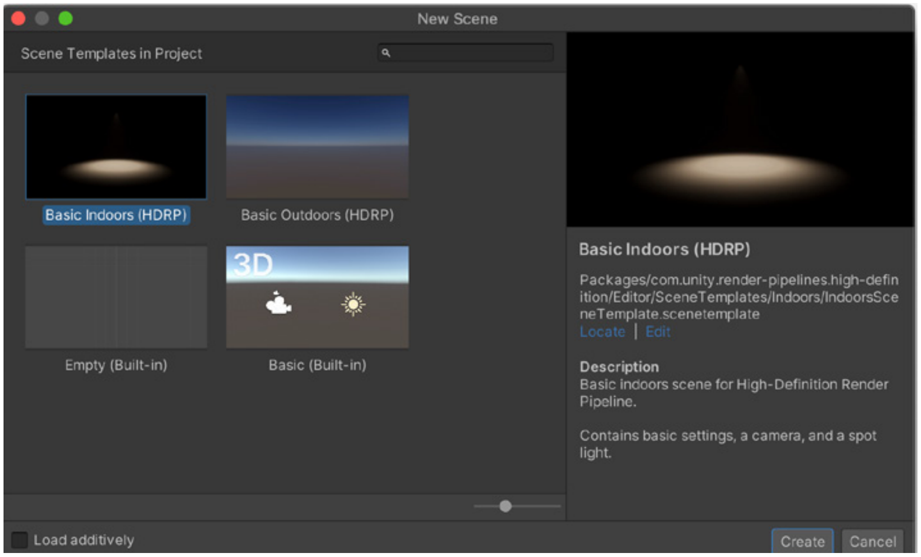
The scene files themselves are assets that are stored on disk. If you use Force Text Mode for Asset Serialization, they appear as text files; otherwise, they default to binaries.

Scenes often represent a level of your game or a portion of a level. While demos and simple games might occupy just a single scene, most commercial games might use one scene per level, each with its own environments, characters, UI, etc.

You can create any number of scenes in a project, but be aware that how you structure your scenes can have a significant performance impact. For more info on scene organization and performance, check out our mobile performance optimization guide.

You'll need to create, create, load, and save scenes to represent different portions of your game and flesh out your application. A typical "scene flow" involves triggering the loading of another scene with an event. For example, you may have a menu scene that loads up a main gameplay scene when the user clicks the interface. Note that you can load scenes one at a time, or separate elements and load the scenes additively.

When creating a new scene, Unity allows you to select from a set of Scene Templates. For example, the HDRP 3D Sample Scene comes with several templates. You can define your own scene templates to streamline your workflow and start everyone on your team with the same assets and configuration.



Basic Indoors (HDRP)

Packages/com.unity.render-pipelines.high-definition/Editor/SceneTemplates/Indoors/IndoorsSceneTemplate.scenetemplate

Description
Basic indoors scene for High-Definition Render Pipeline.

Contains basic settings, a camera, and a spot light.

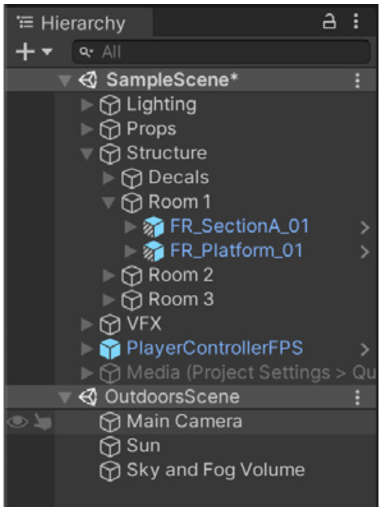
Scene templates in HDRP

Unity provides a SceneManagement API for loading or managing scenes from scripts. See this Learn tutorial on scene flow for more information.

The Hierarchy window

The Hierarchy window displays every GameObject in the currently loaded scene's assets. These include your models, Cameras, and Prefabs. Simply drag a GameObject to change its parenting.

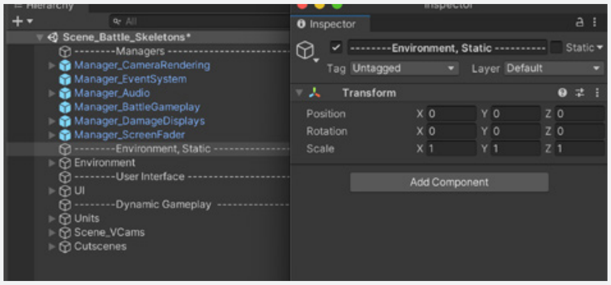
Adding or removing objects in the Scene view also adds or removes them from the Hierarchy window (and vice versa). The Hierarchy window can show more than one loaded scene at runtime, with each scene containing its own GameObjects.



The Sample Scene with the HDRP template includes several asset folders.

General tips for scenes and hierarchies

— **Use named, empty GameObjects as spacers.** Carefully organize your scenes to make it easy to find objects. Keep these to a minimum, as every GameObject matters. Balance your organizational needs with performance. Avoid unnecessary parenting for organization (see **Use proper parenting** below).



☑ --------Environment, Static ----------
Tag Untagged
Layer Default
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Caption: Empty game objects serve as spacers.

— **Put maintenance Prefabs and empty GameObjects at world origin.** If a transform is not specifically used to position an object, it should be at (0,0,0). This simplifies code and reduces issues converting between local and world space.

— **Put your world floor at y = 0**. This makes it easier to put objects on the floor. Treat the world as a 2D space along the xz-plane for game logic, AI, and physics.

— **Separate dynamic and static objects**. If you generate moving objects at runtime, consider keeping them organized under an empty placeholder object. Likewise, store non-moving level geometry in a different part of the hierarchy. This can help you apply the appropriate lighting techniques to your geometry (e.g., lightmapping versus probe lighting).

— **Use proper parenting**. Group related objects by function. Use common sense when creating hierarchies (e.g., parent the tires so that they are children of the car body). Avoid unnecessary parenting when possible, as a flatter hierarchy is more performant. Follow these guidelines for scene hierarchies.

Naming standards

While there is no definitive naming standard for GameObjects, consider the following standards and practices for your project.

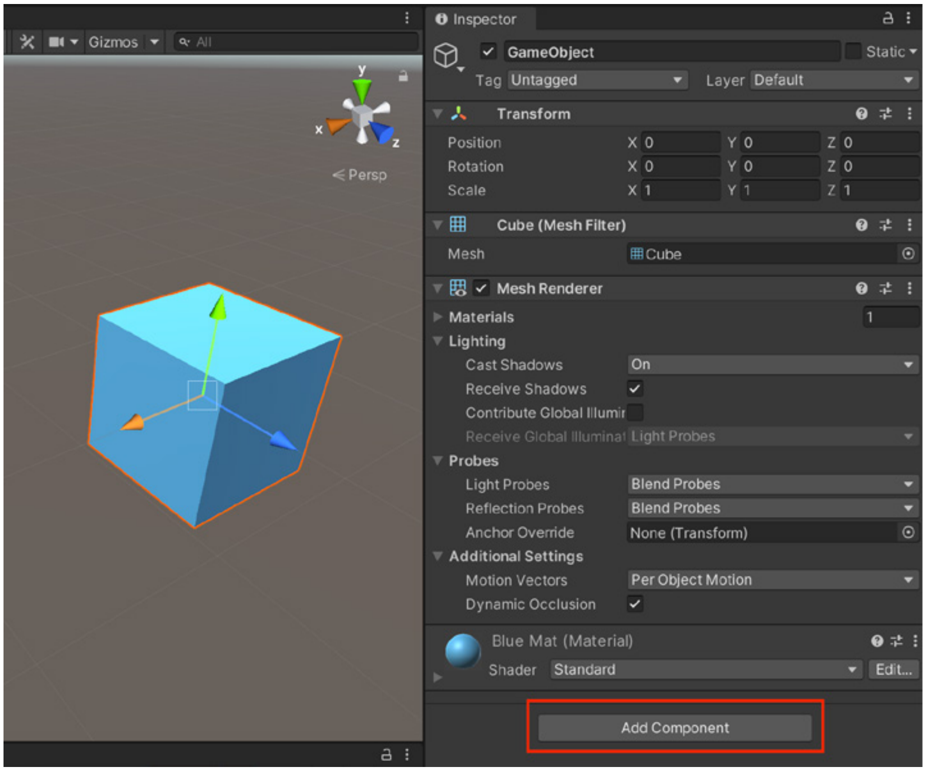
Standard	Example
Use descriptive names. Don't abbreviate. Use names that you will remember several months from now. Consider whether another person will understand your notation, and choose names that you can pronounce and remember. Be aware that abbreviations can create confusion.	**largeButton, LargeButton, or leftButton** NOT: lButton
Use Camel case/Pascal case. Avoid spaces in your object names. Camel case or Pascal case improve readability (and typing accuracy, according to this study).	**OutOfMemoryException, dateTimeFormat,** NOT: Outofmemoryexception, datetimeformat
Use underscores (or hyphens) sparingly. Avoid underscores and hyphens in general. However, they can be useful in certain circumstances. Prefixing a name with an underscore puts it alphabetically first. You can also use underscores to denote variants of a specific object.	Active states: **EnterButton_Active, EnterButton_Inactive** maps: **Foliage_Diffuse, Foliage_Normalmap** Level of detail: **Building_LOD1, Building_LOD0**
Use number suffixes to denote a sequence. Don't suffix with a number if it's not part of a list.	For a path, name the nodes: **Node0**, **Node1**, **Node2**, etc.
Follow the design document naming.	If your design document names locations like **HighSpellTower** or **RedDragonLair**, use those exact spellings.

As with all naming standards, decide on what works for your team and apply it *consistently*.

GameObjects and components

Gameplay and interactivity in Unity is constructed from **GameObjects** and **components**. These are the fundamental building blocks for your Unity project. Create them in the interface or using script.

Everything in your game hierarchy is a GameObject. GameObjects are essentially empty containers that don't do anything on their own. They can represent a wide range of things in your game, from a character to an environment to a particle effect.



☑ GameObject ☐ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cube (Mesh Filter)
Mesh Cube

☑ Mesh Renderer
Materials 1
Lighting
Cast Shadows On
Receive Shadows ☑
Contribute Global Illumir ☐
Receive Global Illuminat Light Probes
Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Additional Settings
Motion Vectors Per Object Motion
Dynamic Occlusion ☑

Blue Mat (Material)
Shader Standard Edit...

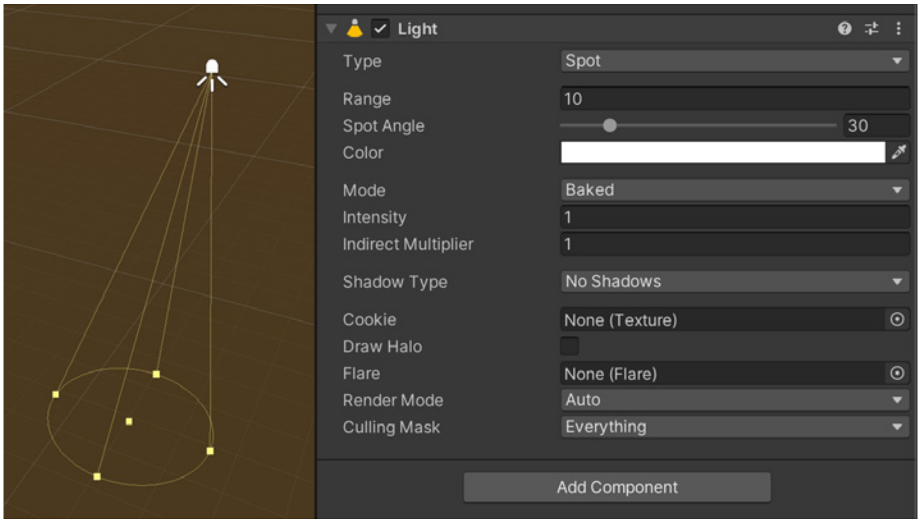
Adding a component changes its functionality.

You need to add special components to a GameObject before they can become a functional part of your game. Each GameObject can hold more than one component. Often components work together in concert or by communicating with components on other GameObjects.

Depending on what functionality you want to create, you will add different combinations of components to the GameObject. You can also make your own components using Scripts.

The Inspector

Components have any number of editable properties, or variables, that can be changed using the Inspector window. You can also modify these variables with your custom scripts.



Light

Type: Spot

Range: 10

Spot Angle: 30

Color

Mode: Baked

Intensity: 1

Indirect Multiplier: 1

Shadow Type: No Shadows

Cookie: None (Texture)

Draw Halo: □

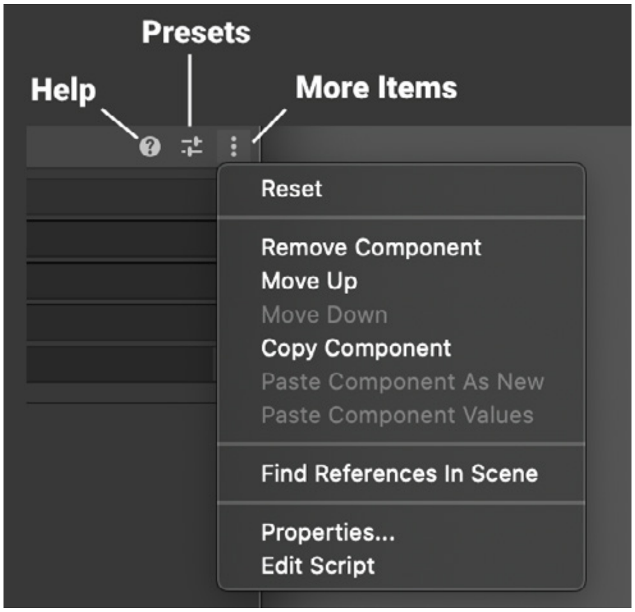
Flare: None (Flare)

Render Mode: Auto

Culling Mask: Everything

Modify an example light's range, color, and intensity in the Inspector window.

Use the Inspector to enter or reset values. The Help icon also gives you quick access to documentation, while the Presets feature lets you quickly save or load values. You can use the **More Items** (⁝) menu to reorder components or to copy and paste values.



Use the Help, Preset, and More Items menus to manage your components

The Transform component

Every GameObject in Unity has a Transform component for its position, rotation, and scale. You cannot remove this component. Transform values in the Inspector for any child GameObject are displayed relative to its parent's transform in local coordinates.

Transform scale determines the size of a mesh in Unity compared to the mesh size in your modeling application. This is important for the physics engine, which assumes one unit in world space is one meter.

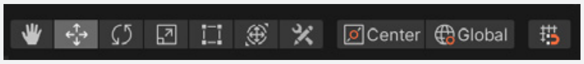
Like many 3D applications, Unity represents rotations as Quaternions internally, but it shows values of the equivalent Euler angles in the Inspector for easier editing.

Tips for working with transforms

- **Clear transformation values before adding the child.** Set the parent's Transform position and/or rotations to (0,0,0) when adding a transform as a child. Unity compensates for any existing values to keep the child object in the same world space position when reparenting.

  If the parent is untransformed, the local coordinates for the child will be the same as its global coordinates. This keeps the setup simple.

- **Use the correct Tool/Pivot mode.** When moving, rotating, and scaling, be aware of your tool settings in the Toolbar. Unity includes a toggle between Pivot and Center mode; this switches the Gizmo handle between the model pivot and the center of the selection, respectively. Likewise, be aware of toggling between Local and Global coordinate systems.



Tool settings

- **If an object needs to sit or stand on the floor, put its pivot at the base, not in the center.** This makes it easy to put characters and objects on the floor precisely. In a 3D project, you can work in a top-down view on the xz-plane more easily.
- **Make all meshes face in the same direction.** This applies to meshes such as characters and other objects that have the concept of facing a direction. Consistency makes many calculations at runtime much simpler.

- **Get the scale right from the beginning.** Make art assets so that they can all be imported at a scale factor of 1 with their x, y, and z scale axes at 1.

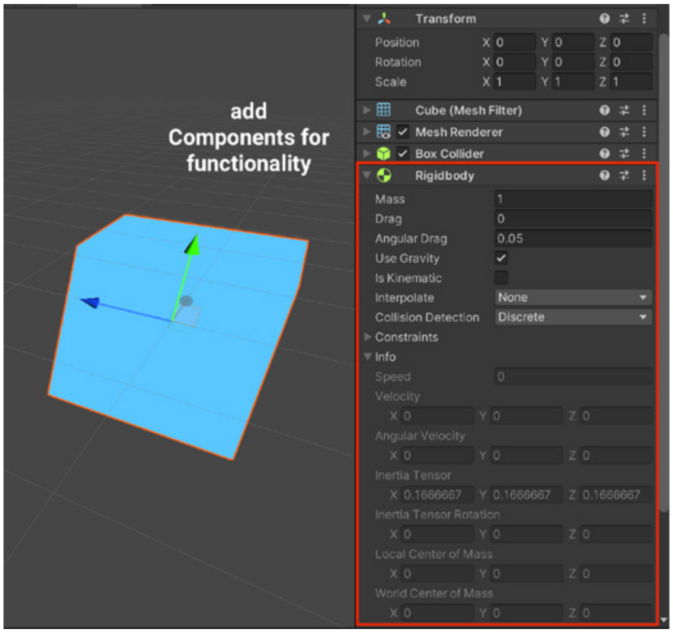
  Use a reference object (**GameObject > 3D Object > Cube**) to make quick scale comparisons. Avoid non-uniform scales on the root transform if you do need to scale. Otherwise, this can lead to unexpected results when adding child transforms.

- **If you are using Rigidbody components for physics simulation, then model at 1 unit = 1 meter.** Otherwise, compensate using the Mesh Scale Factor in the model's Import Settings. Large objects appear to fall in slow motion because we perceive speed relatively (e.g., a mouse and an elephant traveling the same speed in world units appear differently to the human eye because of how they move relative to their own body sizes).

  A physics simulation may be correct, even if the perceived speed does not appear to be. See the Rigidbody component reference page for more about how mesh scale affects physics

Built-in components

Add components to the selected GameObject using the Components menu. For example, if you add a Rigidbody (**Add Component > Physics > Rigidbody**) to a 3D cube, you will see the Rigidbody's properties appear in the Inspector.



Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cube (Mesh Filter)
✓ Mesh Renderer
✓ Box Collider

Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity ☑
Is Kinematic ☐
Interpolate None
Collision Detection Discrete
Constraints
Info
Speed 0
Velocity
X 0 Y 0 Z 0
Angular Velocity
X 0 Y 0 Z 0
Inertia Tensor
X 0.1666667 Y 0.1666667 Z 0.1666667
Inertia Tensor Rotation
X 0 Y 0 Z 0
Local Center of Mass
X 0 Y 0 Z 0
World Center of Mass
X 0 Y 0 Z 0

Add native components for greater functionality (Rigidbody in this example).

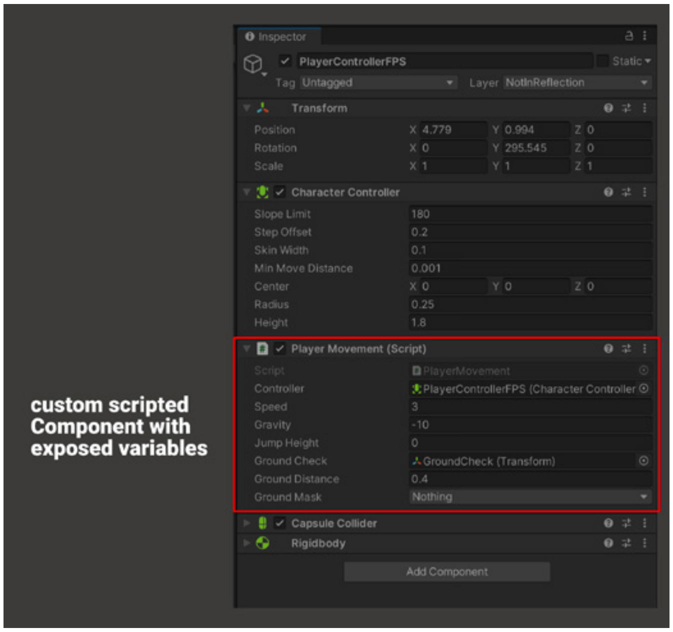
Press **Play** while the empty GameObject is still selected, and the physics engine in Unity causes the GameObject to fall under gravity. The Rigidbody has added functionality to the otherwise empty GameObject.

You'll use combinations of components to create unique gameplay. Built-in components give you access to sounds, rendering, AI navigation, and more. You'll uncover more features as you continue on your journey into the Editor.

Custom components

While built-in components add many abilities to your GameObjects, you will inevitably need to add your own behaviors using Unity's Scripting API. Custom scripted components can trigger game events, check for collisions, apply physics, respond to user input, and much, much more.

When you create a script and attach it to a GameObject, the script appears in the GameObject's Inspector, just like a built-in component, and you can tweak its values as you test your game design in the Editor. Scripts become components once they save and compile in your project.



custom scripted Component with exposed variables

☑ PlayerControllerFPS □ Static
Tag Untagged Layer NotInReflection

Transform
Position X 4.779 Y 0.994 Z 0
Rotation X 0 Y 295.545 Z 0
Scale X 1 Y 1 Z 1

☑ Character Controller
Slope Limit 180
Step Offset 0.2
Skin Width 0.1
Min Move Distance 0.001
Center X 0 Y 0 Z 0
Radius 0.25
Height 1.8

☑ Player Movement (Script)
Script PlayerMovement
Controller PlayerControllerFPS (Character Controller
Speed 3
Gravity -10
Jump Height 0
Ground Check GroundCheck (Transform)
Ground Distance 0.4
Ground Mask Nothing

☑ Capsule Collider
Rigidbody

An example of a custom scripted component

Scripts attached to GameObjects inherit from a built-in class called MonoBehaviour. Making **MonoBehaviours** interact with one another is core to creating gameplay in Unity.

The Unity Scripting API details every class available to the UnityEngine and UnityEditor. We recommend that you first familiarize yourself with the manual, then dive deeper as you begin to explore.

Prefabs

Unity's Prefab system allows you to create, configure, and store a GameObject as a reusable asset. This keeps all of its components, property values, and child GameObjects intact. The Prefab asset in a project acts as a template to create new Prefab instances in the scene.

When you want to reuse a GameObject configured in a particular way, convert it to a Prefab.

Prefab usage

You can use Prefab instances for many of the GameObjects in your hierarchy. A Prefab can be anything you want to replicate; it could be a character (either player or NPC), a prop, or part of the level geometry. Some common examples of Prefab use include:

- **Environmental assets:** If you want to reuse the same tree or building several times in a level, convert those meshes into Prefabs.
- **Non-player characters (NPCs):** A certain type of character may appear in your game several times, across multiple levels. Use overrides to make them differ by behavior, visuals, or sounds.
- **Projectiles or power-ups:** Use Prefabs when you want to instantiate GameObjects at runtime that did not exist in your scene at the start. For example, your laser gun might instantiate a particle effect each time it fires.
- **The player's main character:** The player Prefab might be placed at the starting point for each level of your game (or in a separate scene loaded additively).

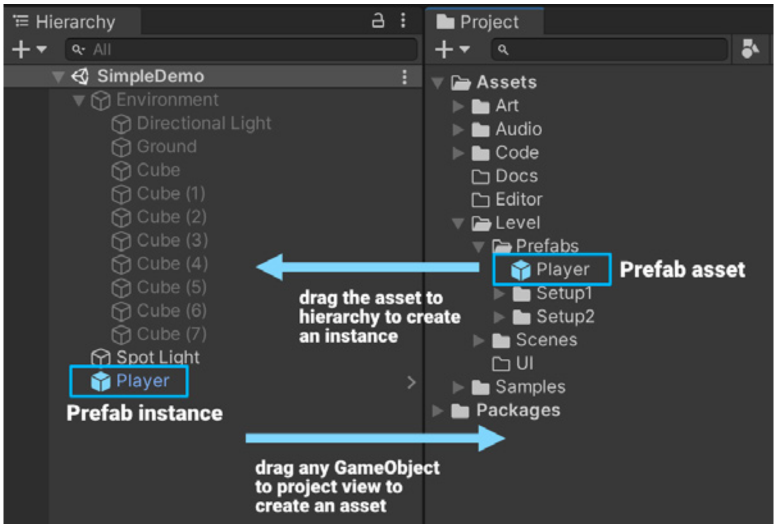
Changes can automatically sync to the Prefabs, simplifying the process of modifying your assets and propagating changes to your scene.

Creating Prefab assets and instances

To create a Prefab asset, drag a GameObject from the Hierarchy window into the Project window. The GameObject, including its components and child GameObjects, becomes a new asset in your Project window.

Prefabs assets appear either with a blue cube Prefab icon or thumbnail view of the GameObject, depending on whether your Project view uses a one-column or-two column layout.

Likewise, drag the Prefab asset from the Project view to the hierarchy or scene view to create a Prefab instance.



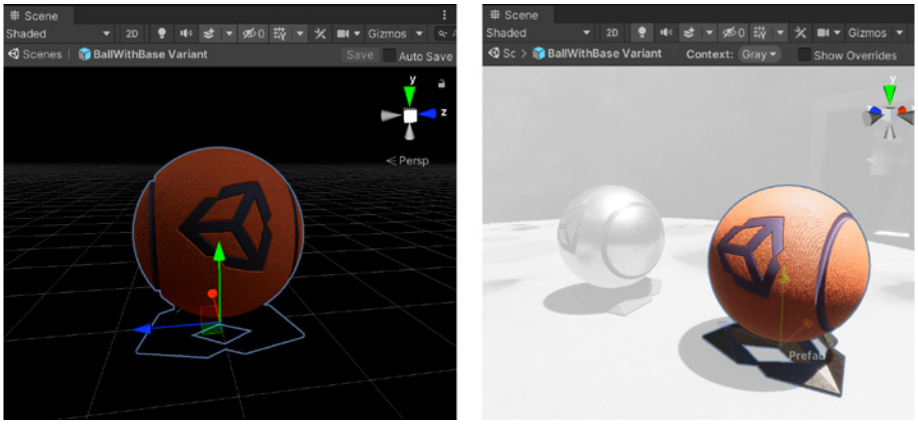
Creating a Prefab asset and instance

Any changes made to the Prefab asset on disk update automatically on the instances in the scene. This way you don't need to make the same edit to every copy of the asset.

This does not mean all Prefab instances have to be identical. You can override settings on individual prefab instances to make them differ from other instances or from the original asset.

Prefab mode

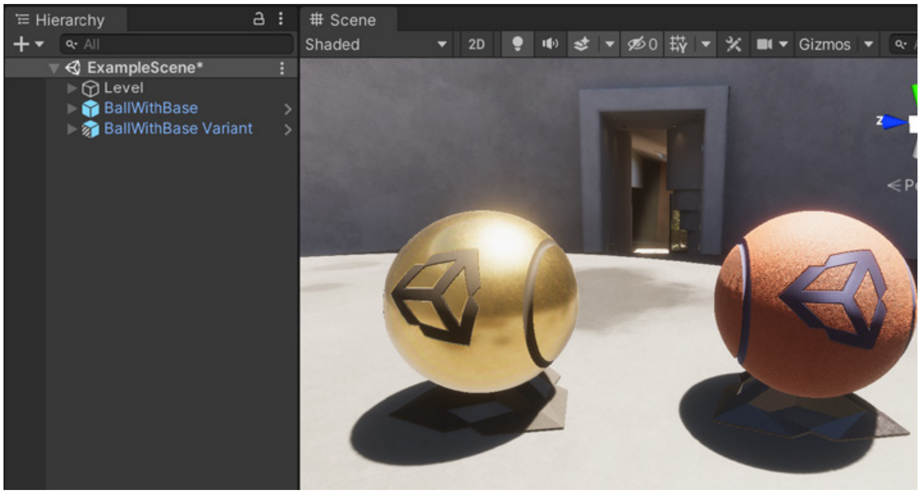
To edit a Prefab asset, you can open it in a special **Prefab Mode**. Prefab Mode allows you to view and edit the contents of the Prefab asset either separately in isolation or in the context of the other GameObjects in your scene. Changes that you make in Prefab Mode affect all instances of that Prefab.



Prefab Mode in isolation (left) or in context (right)

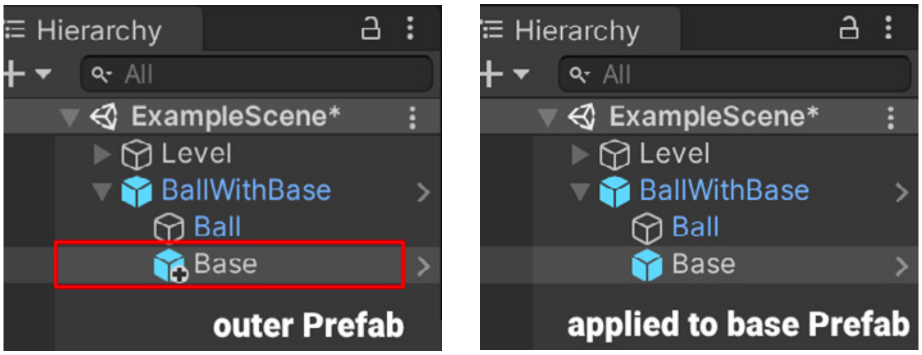
Prefab Variants and Nested Prefabs

You can create variants of Prefabs which allow you to design a meaningful modification of a Prefab. Think of it like inheriting from an archetypal asset and then overriding just the parts you want to change.



Variants of the same base Prefab

Nest Prefabs inside other Prefabs to create complex hierarchies of objects that are easy to edit at multiple levels. A Nested Prefab instance has a plus badge overlayed on its icon in the hierarchy. That indicates it overrides that specific instance of the outer Prefab.



Nested Prefabs, with the "outer" Prefab denoted by the plus badge

Edit the source Prefab assets independently, propagating changes into the Nested Prefabs.

Package Manager

To make development more flexible, Unity offers some features as optional add-on packages. These exist outside of the core Editor so that they can follow a more rapid release cycle. The package management system keeps your build small, as it allows you to only install the tools that you need for your specific project requirements. Packages themselves can vary in functionality. They can contain any combination of assets, shaders, textures, plug-ins, icons, and scripts to enhance various parts of your project.

Manage available packages using the **Package Manager** window (**Window > Package Manager**).

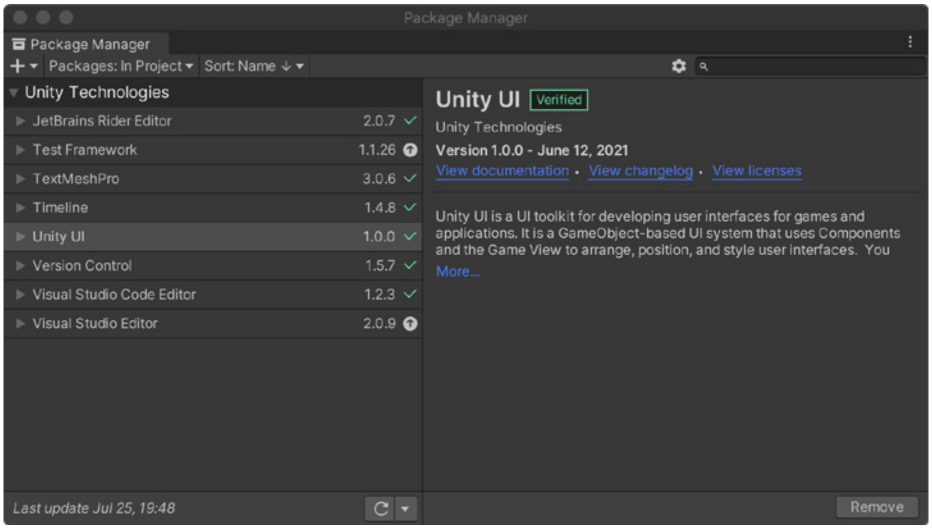
In addition to Unity-provided packages that are built-in or part of the Unity registry, you can also install custom packages from disk, GitHub URL, or the Asset Store:

— Click the + button to install a package directly from git URL or a local path.

— The **Packages** drop-down menu changes what appears in the list (Unity Registry/In project/My Assets/Built-in). Meanwhile, the Sort menu organizes packages by name or date.

— Asset Store packages from **My Assets** feature additional filtering options.

— The **Search** field allows you to locate a package by name. Use the Download, Import, and/or Update buttons to manage the packages in use by your project.



JetBrains Rider Editor	2.0.7 ✓
Test Framework	1.1.26
TextMeshPro	3.0.6 ✓
Timeline	1.4.8 ✓
Unity UI	1.0.0 ✓
Version Control	1.5.7 ✓
Visual Studio Code Editor	1.2.3 ✓
Visual Studio Editor	2.0.9

Unity UI

Unity Technologies

Version 1.0.0 - June 12, 2021

Unity UI is a UI toolkit for developing user interfaces for games and applications. It is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces. You

The Package Manager

Experienced developers can also leverage the PackageManager API to build their own shared libraries. Watch Creating Custom Packages in Unity for a quick introduction.

Programming

In Unity, you can use code to customize and control just about any part of your game, create visual tools for your team, and even change the way Unity itself works. Unity uses C#, a modern object-oriented language adopted widely in software industries.

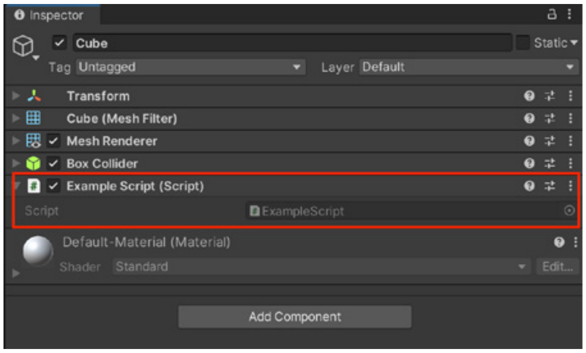
Custom components and MonoBehaviours

Every type of GameObject comes with a set of default components. For example, an empty GameObject starts with the Transform component. But you can extend this default collection with custom components that can tie your own game logic directly to objects your team uses in game scenes. You can even empower your designers to tweak values and behavior through values in your components.

To do this, create scripts, then add those scripts as components to GameObjects. Each script derives from the built-in class called **MonoBehaviour**.

Think of the MonoBehaviour class as a kind of blueprint for creating a new component type. Each time you attach a script component to a GameObject, the blueprint defines a new instance of that particular component.

Scripts are created directly within Unity. If you use the **Assets** menu (or **right-click**) **> Create > C# Script**, this generates a new C# script on disk. Name the filename to match your desired class name. Drag this onto an object in the hierarchy, and the inspector shows the ExampleScript appears as a component.



☑ Cube

Tag Untagged

Layer Default

Script ExampleScript

Shader Standard

A new C# script

Note: If you change the name of the class inside the file but not the filename, it can cause the script to not function properly when attached as a custom component. Unity will automatically set up a class named **ExampleScript** that inherits from MonoBehaviour.

Unity's current script template starts the class with two functions, **Start** and **Update**.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ExampleScript : MonoBehaviour
{
    // Start is called before the first frame update
   void Start()
   {
   }

// Update is called once per frame
   void Update()

  {
  }

}
```

Unity calls the **Start** function before gameplay begins and before it calls the Update function for the first time. Thus, the Start function is an ideal place to set up variables, read preferences, and make connections with other GameObjects.

The **Update** function handles code that runs every frame. For example, this might include movement, triggering actions, and responding to user input.

Update and **Start** are only two of MonoBehaviour's event functions. These built-in methods run on a set order of execution. Overriding MonoBehaviour's event functions is how you will construct gameplay and build the main game loop.

To familiarize yourself with the basics of coding in Unity, check out our documentation here.

Initializing objects

Experienced programmers may be surprised that initializing an object is not done using a constructor. The Editor handles object construction, which does not take place at the start of gameplay. Attempting to define a constructor for a script component will interfere with the normal operation of Unity and can cause problems.

MonoBehaviour lifecycle and structure

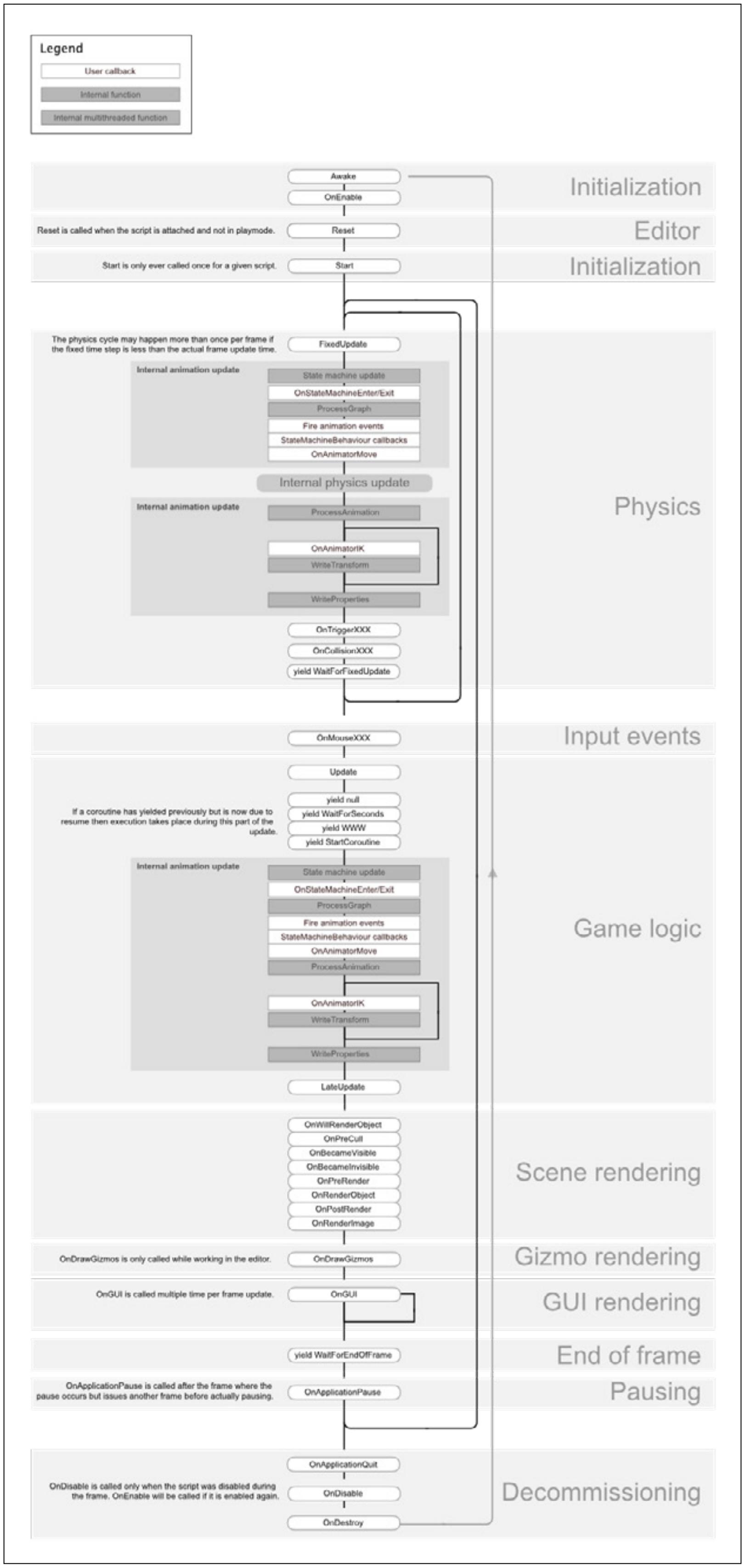
Game engines rely on an endless loop responsible for processing user input, updating game state, and rendering to the screen. In Unity, this **PlayerLoop** is a low-level class at the heart of the game engine. It controls a number of subsystems that handle initialization and per-frame updates.

To interface with the PlayerLoop, your scripts derive from the MonoBehaviour base class, and learning how this operates is key to creating gameplay. This flowchart shows MonoBehaviour's event functions and how they execute over a script's lifetime.

Here are some essential parts of the game loop when working with MonoBehaviours:

- First scene load
- Editor
- Before the first frame update
- In between frames
- Update order
- Animation update loop
- Rendering
- Coroutines
- When the object is destroyed
- When quitting

Experienced users can even build their own PlayerLoop and PlayerLoopSystems. However, we recommend that you begin by familiarizing yourself with the Monobehaviour class and the most common classes below.



Reset is called when the script is attached and not in playmode.

Start is only ever called once for a given script.

The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.

If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.

OnDrawGizmos is only called while working in the editor.

OnGUI is called multiple time per frame update.

OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.

OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.

MonoBehaviour lifecycle

More scripting tips

Unity's scripting backend is based on the .NET Framework. Take advantage of these C# features when scripting for Unity development:

- **Namespaces**: Classes in Unity must have unique names. When several programmers check their work into the same project, common names like "Controller" can create conflicts. Use namespaces to avoid this, and organize your data types. Apply the using directive to shorten the namespace prefix if desired.

  For example, you could create a namespace for the Player as well as for an Enemy. Then Player.Controller and Enemy.Controller could safely live in the same project.

  See the Namespaces manual page for more information.

- **Coroutines**: Sometimes you may want to trigger an action and have it take place over multiple frames. For example, imagine moving an object from A to B over a duration or slowly fading its color. Normally a function runs to completion and returns on the same frame, making it more difficult to perform logic over a timeframe.

  A coroutine has the ability to pause execution and return control to Unity, then continue where it left off on the following frame. You can use coroutines to apply game logic for a specified time. For example, pausing execution for a set time is often done via coroutine. You can even use coroutines to wait for other coroutines or combine it with a while loop to wait for a condition. Coroutines can behave like MonoBehaviour's Update event functions but with the added benefit of controlling the update interval.

  Refer to the Coroutine manual page for more information.

- **Attributes**: These are markers that can be placed above a class, property, or function to indicate special behaviour.

  For example, you can turn a numeric field into a slider in the Inspector using the Range attribute or add a floating Tooltip over a field in the Inspector. C# contains attribute names within square brackets.

  You can find a complete list of Attributes in the Scripting API.

Common classes

Once you start scripting in Unity, you should review some of the most important built-in classes. While this list is not exhaustive, it should help you start exploring Unity. See the Scripting API for a complete list of classes for more information.

Class	Description
GameObject	Represents the type of objects which can exist in a scene
MonoBehaviour	The base class from which every Unity script derives
Object	The base class for all objects that Unity can reference in the Editor
Transform	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	Classes for expressing and manipulating 2D, 3D, and 4D points, lines, and directions
Quaternion	A class which represents an absolute or relative rotation, and provides methods for creating and manipulating them
ScriptableObject	A data container that you can use to save large amounts of data
Time	This class allows you to measure and control time, and manage the frame rate of your project
Mathf	A collection of common math utilities, including trigonometric, logarithmic, and other functions
Random	Provides you with easy ways of generating various commonly required types of random values
Debug	Allows you to visualize 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	Allows you to draw lines and shapes in the Scene view and Game view, as well as interactive handles and controls

Memory management

Unity supports C#, an industry-standard language with some similarities to Java or C++. C# is a "managed language." It automatically handles memory management for you: allocating and deallocating memory, covering memory leaks, and so on.

In some languages like C++, the programmer is responsible for allocating and releasing these blocks of heap memory with the appropriate function calls. By contrast, automatic memory management in C# requires less coding effort than explicit allocation/release, while greatly reducing the potential for memory leakage (where memory is allocated but never subsequently released).

Value versus reference types

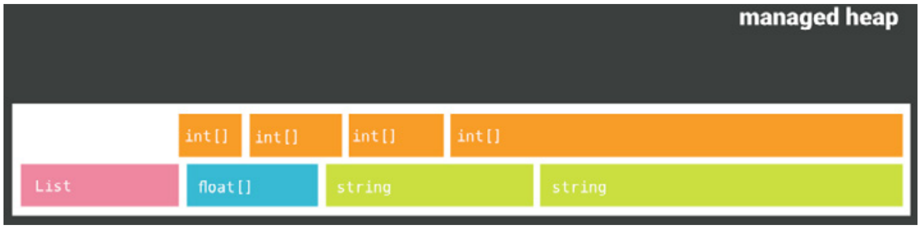
When you call a function, Unity reserves an area of memory for it and copies the values of the function's parameters as well. *Value types*, like integers, floats, and booleans, only occupy a few bytes. Unity stores value types directly and copies them during parameter passing.

Other data types (like objects, strings, and arrays) are *reference types*. They occupy more space and would be inefficient to copy on a regular basis. Instead, Unity stores their data in heap memory and accesses them via pointers. Thus, if you only need a struct (*value type*), using that can be more efficient than if you use a class (*reference type*) to hold the same data.

Although you won't need to allocate and release memory explicitly, you *will* need to understand managed heap memory and how it affects the performance of your game application.

Garbage collection

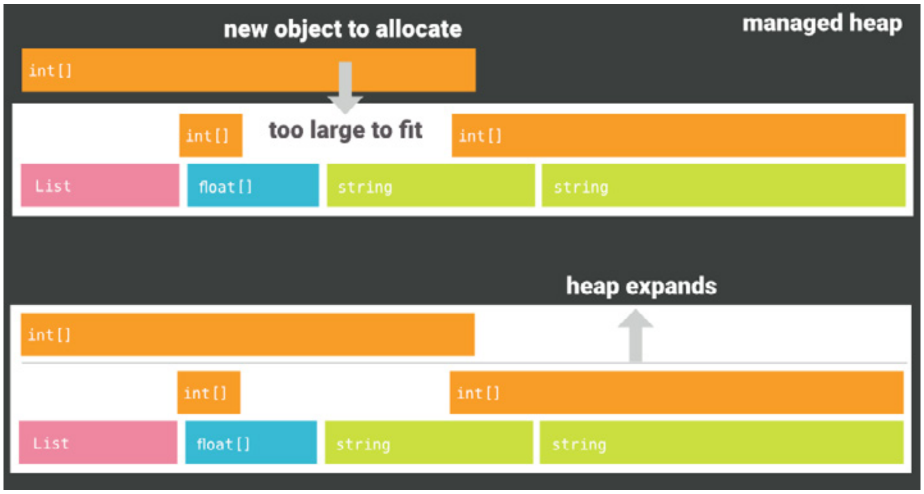
Blocks of heap memory are "live" if they are still in use and have active references.



The Managed Memory Allocator automatically allocates heap memory.

As your game runs, references to a block of memory may disappear (GameObjects get destroyed, variables get reassigned, etc.). Once all references to a memory block are gone, the **Managed Memory Allocator** can safely reuse the memory.

Periodically, the allocator searches the empty spaces between live blocks of memory. Locating and freeing up unused memory is known as garbage collection, or GC for short. When the game application requests new blocks of memory, the allocator draws from these unused blocks.



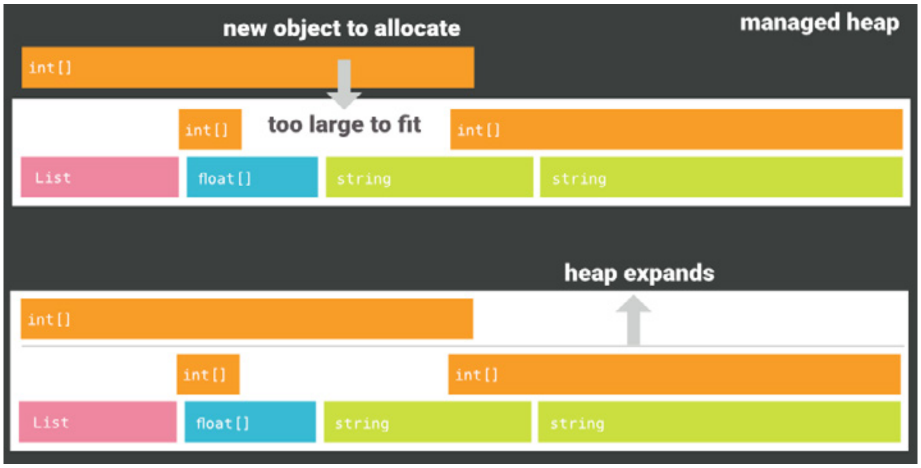
Garbage collection frees up unused memory but pauses execution of your script code.

Unity implements the Boehm–Demers–Weiser garbage collector. During **garbage collection**, Unity stops running your program code, and it only resumes normal execution when the garbage collector finishes.

This interruption can cause delays in the execution of your application, which depend on how much memory the garbage collector needs to process and the game's target platform. These can vary, anywhere from less than one millisecond to hundreds of milliseconds.

For real-time applications like games, this can become quite a big issue. Interruptions from garbage collection, called GC spikes, can cause game play to stutter. Even though garbage collection is invisible for the most part, the collection process actually requires significant CPU time behind the scenes.

Also, be aware that the Boehm GC algorithm is non-compacting. It does not move existing objects in memory to close the gaps between objects, which can lead to memory fragmentation. If you try to allocate a new object that does not fit within the existing gaps, the allocator may need to expand the size of the heap to accommodate it. Heap expansion can impact performance.



Be aware that the heap can expand.

In Unity, you need to avoid triggering the garbage collector more often than necessary. Otherwise, your application could freeze or stutter at runtime. Check out this blog post for optimization tips and tricks that can help reduce the impact of garbage collection.

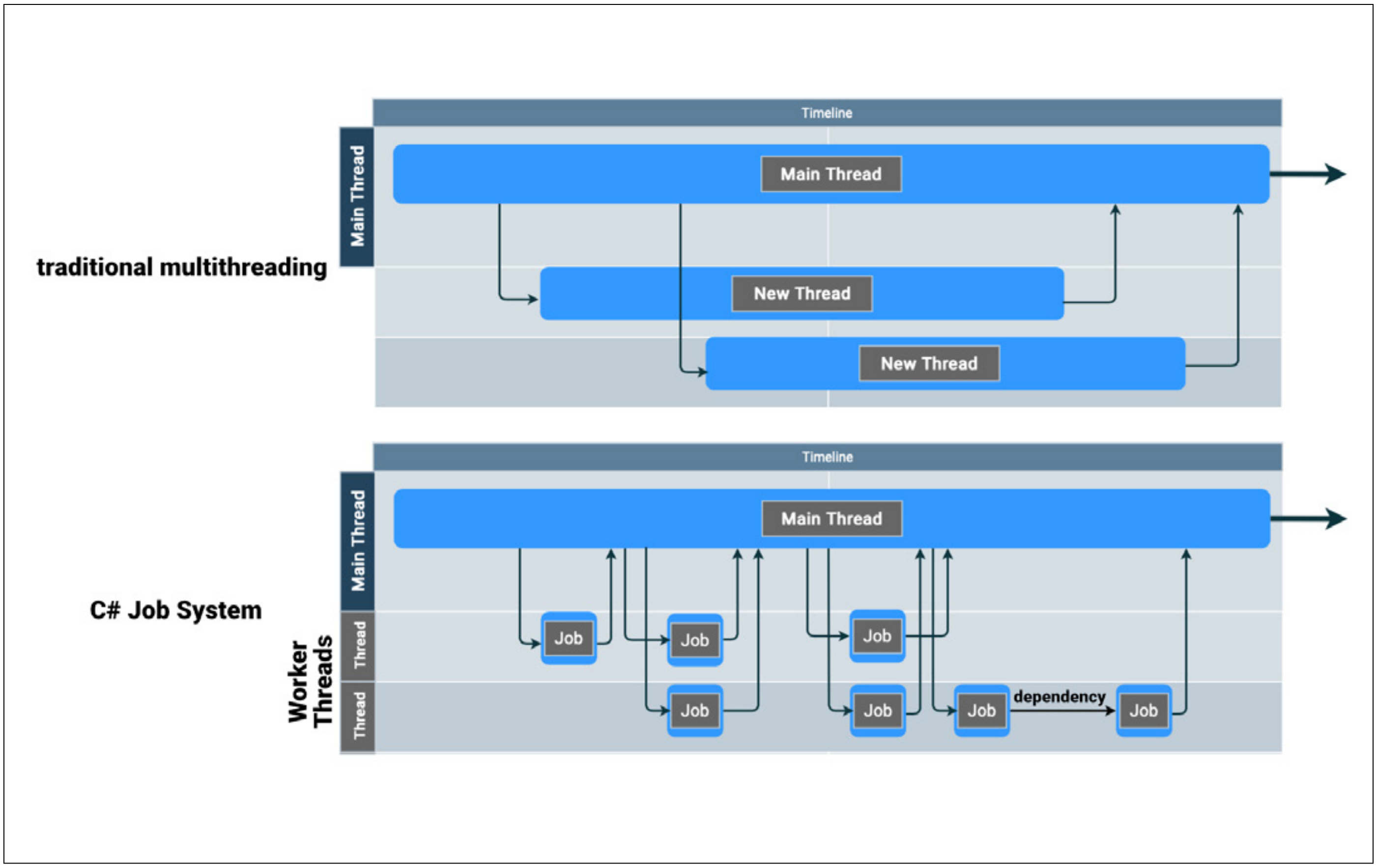
Unity also offers an optional **Incremental Garbage Collector** that splits GC over multiple frames. This feature is currently Experimental and detailed in this blog post.

See Understanding Automatic Memory Management and Understanding the managed heap in the Unity documentation for more information about memory management and garbage collection. You can also read the Memory Management in Unity guide from the Learn site.

Multithreading: C# Job System and Burst compiler

Modern CPUs have multiple cores, but your application needs **multithreaded code** to take advantage of them. Unity's Job System allows you to split large tasks into smaller chunks that run in parallel on those extra CPU cores. This can significantly improve performance.

Often in multithreaded programming, one CPU thread of execution, the *main thread*, creates other threads to handle tasks. These additional *worker threads* then synchronize with the main thread once their work completes.



In traditional multithreaded programming, threads are created and destroyed. In the C# Job System, small jobs run on a pool of threads.

If you have a few tasks that run for a long time, this approach to multithreading works well. However, it's less efficient for a game application, which typically must process many short tasks at 30–60 frames per second.

Thus, Unity uses a slightly different approach to multithreading called the C# Job System. Rather than generate many threads with a short lifetime, it breaks your work into smaller units called jobs.

These jobs go into a queue, which schedules them to run on a shared pool of worker threads. JobHandles help you create dependencies, ensuring the jobs run in the correct order. In order for a safety system to prevent race conditions, jobs work on a copy of the data. Then, Native Containers send the results back to the main thread.

Complementing the Job System is the **Burst compiler**. Burst translates IL/.NET bytecode into optimized native code using LLVM. To access it, simply add the Burst package from the Package Manager. Burst allows Unity developers to continue using a subset of C# for convenience while improving on performance.

Scripting backends in Unity

Unity has two scripting backends: Mono and IL2CPP (Intermediate Language To C++). Each uses a different compilation technique:

- **Mono** uses just-in-time (JIT) compilation and compiles code on demand at runtime.
- **IL2CPP** uses ahead-of-time (AOT) compilation and compiles your entire application before it is run.

IL2CPP is a Unity-developed scripting backend which you can use as an alternative to Mono when building projects for some platforms. It can improve performance and reduce build sizes, but this often comes with slower build time.

When you choose to build a project using IL2CPP, Unity converts IL code from scripts and assemblies into C++ code, before creating a native binary file (.exe, apk, .xap, for example) for your chosen platform.

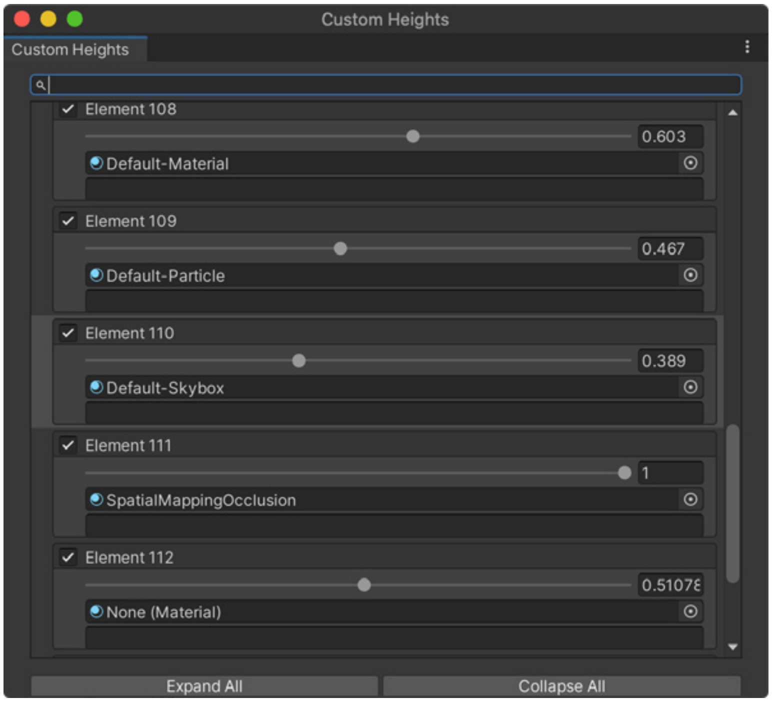
Note that IL2CPP is the only scripting backend available when building for iOS and WebGL.

For more information about using IL2CPP, refer to the The Unity IL2CPP blog series and the Building a project using IL2CPP page.

Editor scripting

You may want to tailor your development environment to the specific needs of your team and project to work more efficiently.

If you require a specialized workflow, you can extend the Editor with your own custom inspectors and windows. These can behave just like the Inspector, Scene, or other built-in windows. You can also define how properties appear with custom Property Drawers.



A custom Editor window

Odin Inspector and Serializer

You can reduce the time you spend in the EditorWindow API using Odin Inspector and Serializer, a third-party Unity Verified Solutions Partner tool that you can purchase on the Unity Asset Store. Odin provides over 100 building-block attributes that let you create custom editors without manually writing and maintaining custom GUI code.

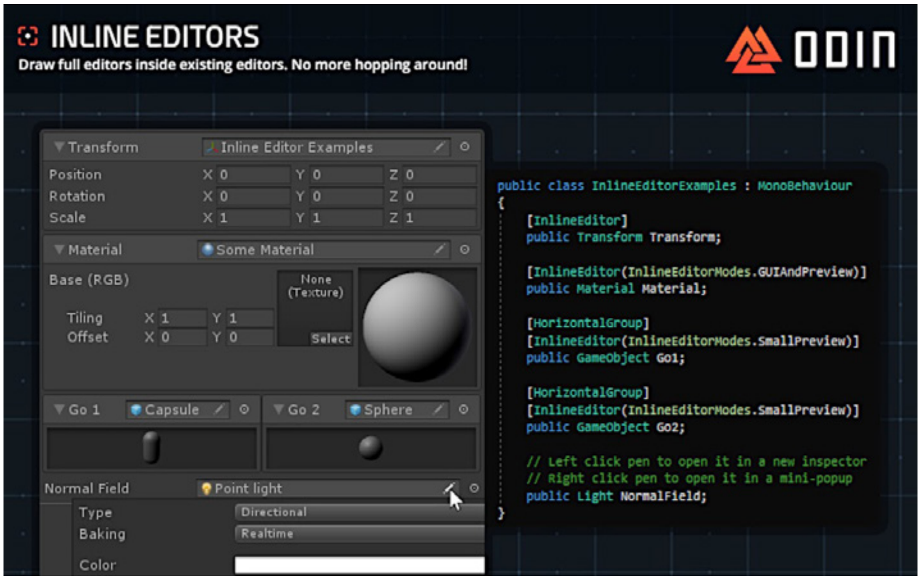
To create a custom editor window with Odin, simply inherit from the OdinEditor Window class, and populate your fields, properties, and methods with attributes.

These are just a few of the processes you can define with Odin:

- Customize layouts with group attributes such as TabGroup and ToggleGroup
- Serialize fields like dictionaries that are normally unavailable with the native Unity Inspector
- Easily create buttons in the Inspector window by adding button attributes to your methods
- Modify static members for testing and debugging; for example, invoke a static method with any arguments directly from the Inspector

- Create custom editors for the Inspector window using attributes
- Write snippets of C# code with attribute expressions directly inside the attributes to reduce boilerplate
- Validate user input with attributes such as Required, ValidateInput, ChildGameObjectsOnly

For example, you can generate an Inspector that looks like this, using a script:

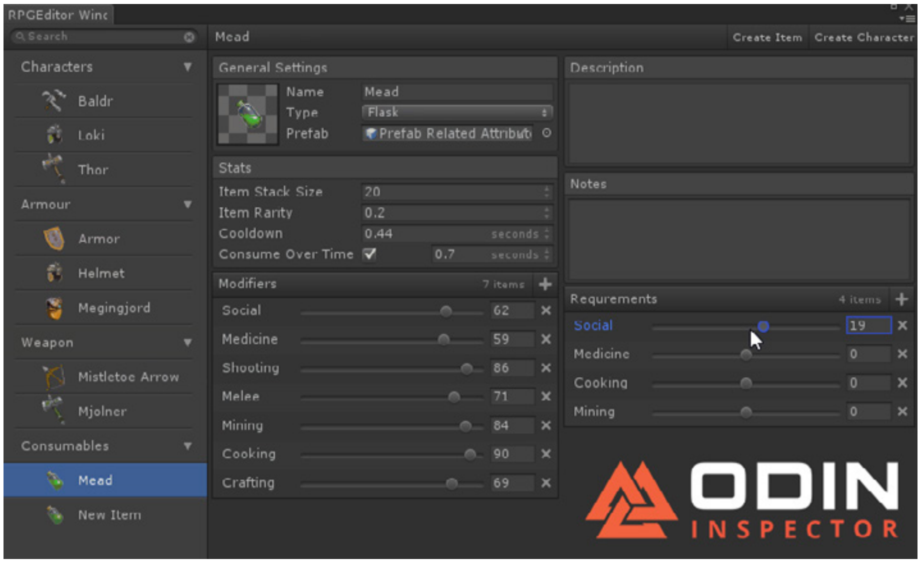

INLINE EDITORS
Draw full editors inside existing editors. No more hopping around!
ODIN

Transform: Inline Editor Examples
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Material: Some Material
Base (RGB): None (Texture) Select
Tiling X 1 Y 1
Offset X 0 Y 0
Go 1: Capsule
Go 2: Sphere
Normal Field: Point light
Type: Directional
Baking: Realtime
Color

```
public class InlineEditorExamples : MonoBehaviour
{
    [InlineEditor]
    public Transform Transform;

    [InlineEditor(InlineEditorModes.GUIAndPreview)]
    public Material Material;

    [HorizontalGroup]
    [InlineEditor(InlineEditorModes.SmallPreview)]
    public GameObject Go1;

    [HorizontalGroup]
    [InlineEditor(InlineEditorModes.SmallPreview)]
    public GameObject Go2;

    // Left click pen to open it in a new inspector
    // Right click pen to open it in a mini-popup
    public Light NormalField;
}
```

Example created in Odin Inspector

Here is an example of an editor window that was made using Odin:



Mead

Name: Mead
Type: Flask
Prefab: Prefab Related Attribut

Stats
Item Stack Size: 20
Item Rarity: 0.2
Cooldown: 0.44 seconds
Consume Over Time: ☑ 0.7 seconds

Modifiers 7 items
Social 62
Medicine 59
Shooting 86
Melee 71
Mining 84
Cooking 90
Crafting 69

Requirements 4 items
Social 19
Medicine 0
Cooking 0
Mining 0

RPG editor window created in Odin

The Odin Inspector is available in both Personal and Enterprise editions from the Unity Asset Store.

Integrated development environment (IDE) support

Unity supports several IDEs so that you can work in your preferred development environment. Visual Studio is installed by default with Unity on Windows and macOS. Select your script editor in Preferences (**Unity > Preferences > External Tools > External Script Editor**).

Unity supports the following IDEs out of the box:

- **Visual Studio** is the default IDE for Unity on Windows and macOS. On Windows, Unity also includes Visual Studio 2019 Community. On macOS, Unity includes Visual Studio for Mac.
- **Visual Studio Code** (Windows, macOS, Linux) is a free, lightweight, and customizable open source code editor known for speed and customizability. For information on using VS Code with Unity, see Unity Development with VS Code.
- **JetBrains Rider** (Windows, macOS, Linux) is built on top of ReSharper and includes most of its features. For more information, see the JetBrains documentation on Rider for Unity.

If your text editor of choice is not one of the above with built-in support, you may need to customize your text editor for Unity development. For example, many community members have created packages (plug-ins, extensions, and add-ons) to use Sublime Text with Unity.

Script templates

As you start creating your custom components, you may find that you make the same changes every time you create a new C# script. For example, you might want to delete the Update event function automatically or add a default namespace. Save yourself a few keystrokes and set up the script template so it fits the task at hand.

Unity uses templates stored in the **ScriptTemplates** resources:

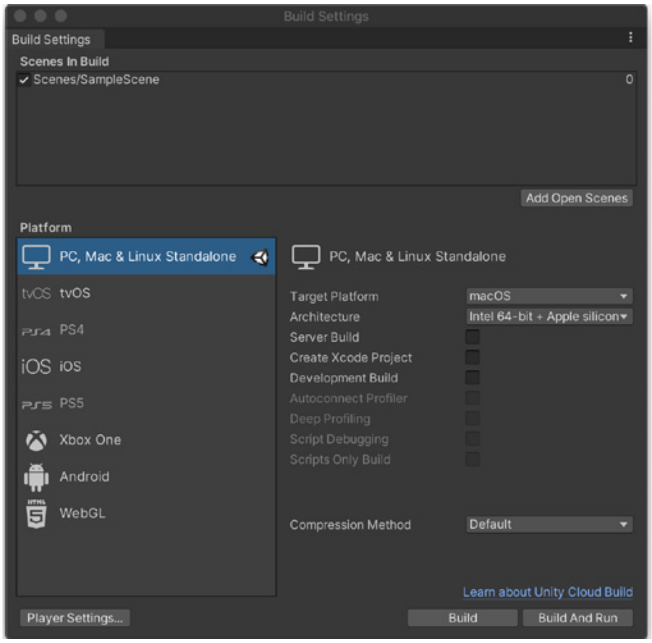
- **Windows:**
  C:\Program Files\Unity\Editor\Data\Resources\ScriptTemplates
- **Mac:**
  /Applications/Hub/Editor/[version]/Unity/Unity.app/Contents/Resources/ScriptTemplates

Open and edit these template files as needed, then relaunch the Unity Editor to apply your changes. Be sure to back up both your original template files and the modified ones.

Building and publishing

One of Unity's strengths is its ability to deploy on multiple platforms. To adjust the publishing settings for your application's build, go to **File > Build Settings**.

Add the relevant scenes to the **Scenes in Build**. The scene that acts as your entry point into the application will have an index of 0.



✓ Scenes/SampleScene 0

Target Platform: macOS
Architecture: Intel 64-bit + Apple silicon
Server Build □
Create Xcode Project □
Development Build □
Autoconnect Profiler □
Deep Profiling □
Script Debugging □
Scripts Only Build □
Compression Method: Default

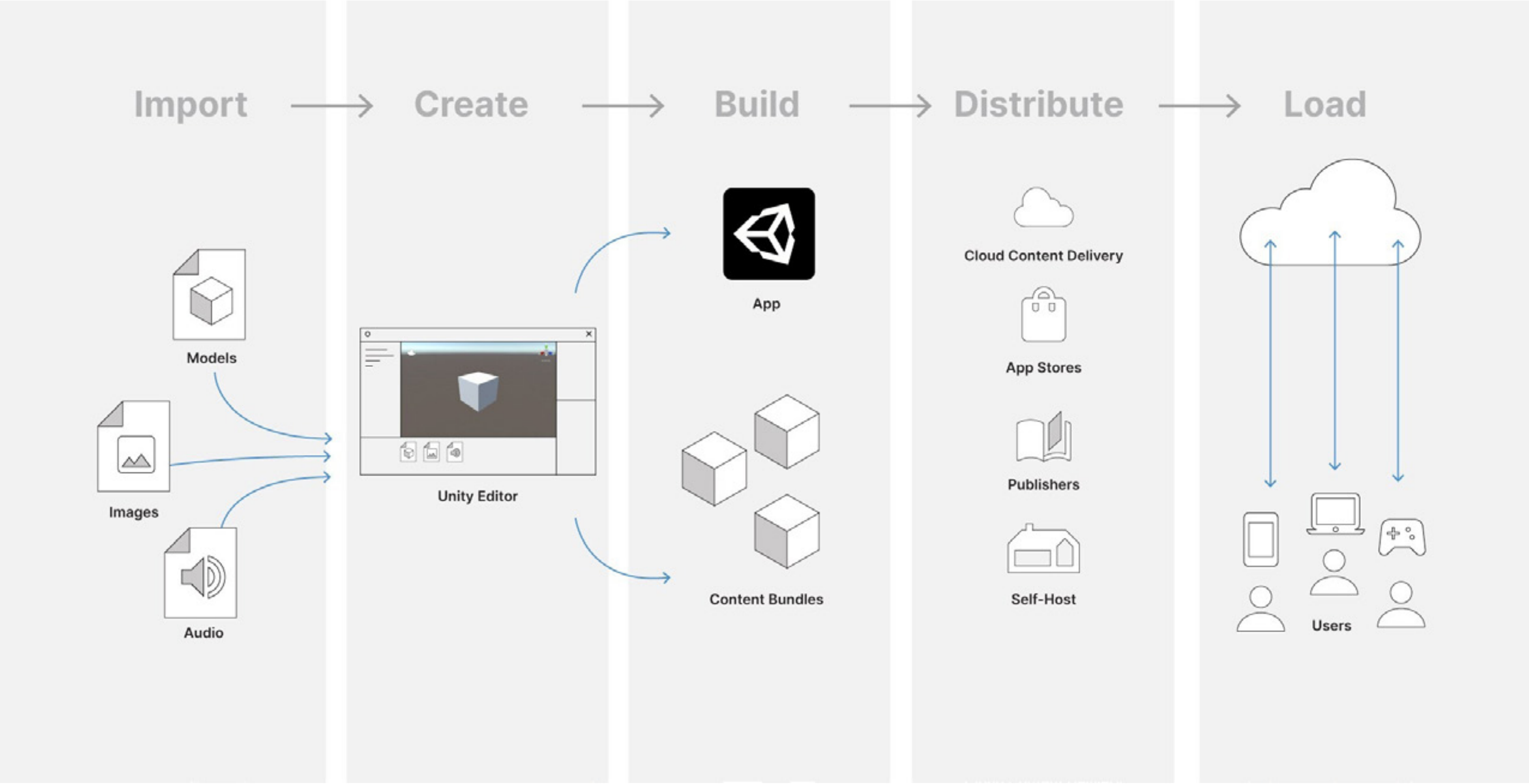
Build Settings

The available platforms appear at the lower left. Use the **Install with Unity Hub** option to install support for the other available target platforms. You can also go directly to the **Installs** screen of the Unity Hub and **Add Module** support for any version of Unity. Some consoles, such as PlayStation, Nintendo Switch, or Xbox, require additional modules from the relevant platform publisher.

Note that you will need to appraise cross-platform considerations when building your application. For example, mobile devices have less storage, memory, and CPU power, so they are also more susceptible to spikes from garbage collection during automatic memory management. You won't be able to port a graphically taxing console game to iOS or Android – at least, not without a great deal of platform-specific optimization.

Also, be aware of input requirements when building for more than one platform. Mobile devices with touchscreens and accelerometers need different settings than platforms with controller or keyboard/mouse input. You can use the new **Input System** to process user input for multiple devices.

Platform-dependent compilation lets preprocessor directives partition your scripts to be specific to a target platform. Combine these with the #if compiler directive or ConditionalAttributeClass. If you wanted to add or remove a UI or object for a platform-specific release (e.g., omit a Quit button from an iOS application), then platform-dependent compilation simplifies that logic.



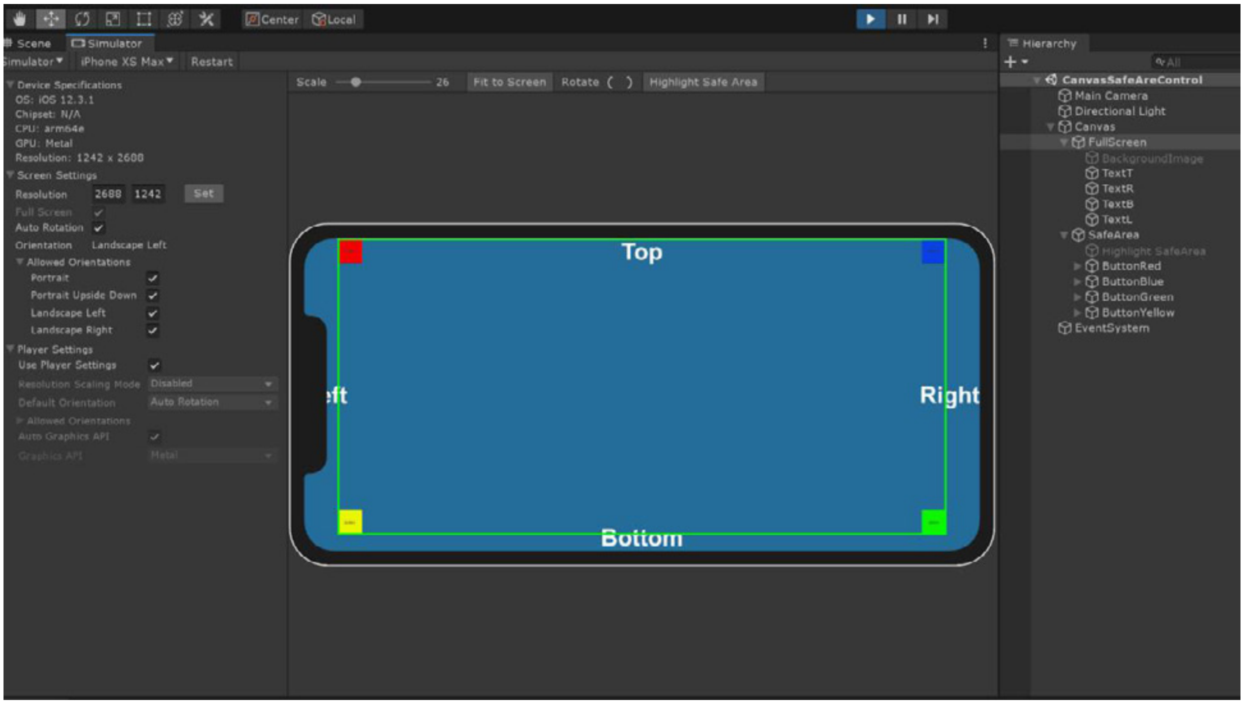
Building and publishing workflow

Device Simulator

If you are building for mobile, consider installing the Device Simulator using the Package Manager. This lets you view your application on a simulated mobile device to preview its screen shape, resolution, and orientation.

To use the Device Simulator view:

- In the Game view, in the top left corner, there is a drop-down menu. Use this to switch between the Game view and the Device Simulator view.
- Using the top menu, choose **Window > General > Device Simulator**.



Device Specifications
OS: iOS 12.3.1
Chipset: N/A
CPU: arm64e
GPU: Metal
Resolution: 1242 x 2688

Screen Settings
Resolution 2688 1242 Set
Full Screen ✓
Auto Rotation ✓
Orientation Landscape Left
Allowed Orientations
Portrait ✓
Portrait Upside Down ✓
Landscape Left ✓
Landscape Right ✓

Player Settings
Use Player Settings ✓
Resolution Scaling Mode Disabled
Default Orientation Auto Rotation
Allowed Orientations
Auto Graphics API ✓
Graphics API Metal

The Device Simulator

The Device Simulator is intended primarily for viewing the layout of your application and testing basic interactions.

Note that the Device Simulator view does not simulate the performance or rendering characteristics of a device, such as processor speed or available RAM.

Watch Simulate your Game with Device Simulator in Unity to get started.

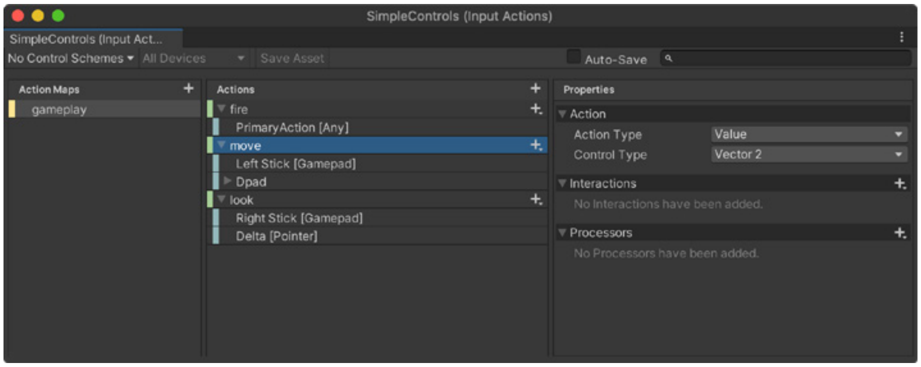
Input

Unity has two main systems for handling input, the Input System from the Package Manager and the built-in Input class.

Input System

The Input System is focused on ease of use, flexibility, and consistency across devices and platforms. It replaces the built-in Input class.

While you can still get input directly from a device, the Input System shines when you create a series of indirect actions to drive player interaction. Then you can create an actionMap, a collection of bindings and actions. The action map can then relate those actions to the actual devices.



Action Type: Value
Control Type: Vector 2

The Input System is available via the Package Manager.

If you want to adapt the input to more devices, you create more action maps rather than hard coding specific device buttons, keyboard keys, etc. Set up your callbacks and actions just once, then add additional mappings later to support more devices.

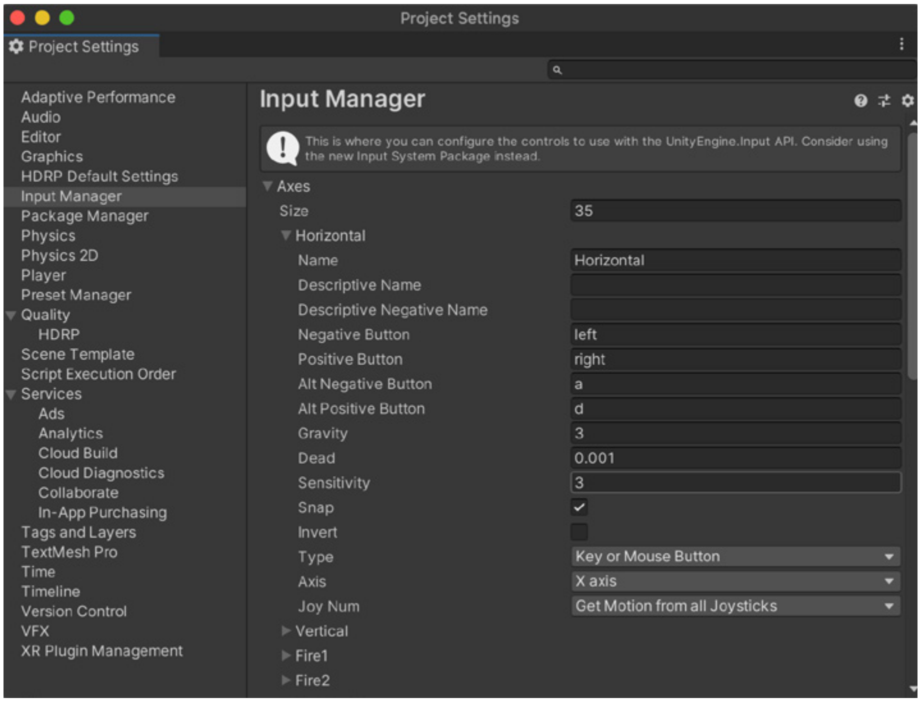
Action maps also assist with switching input depending on your in-game context. For example, your player input can behave differently when driving a vehicle versus walking or running.

Be sure to check out the Input System Quick start guide, and install some of the samples available through the Package Manager.

Built-in input class

You can alternatively use the original built-in Input class. This lets you utilize the Input Manager (**Edit > Project Settings > Input Manager**) to set up virtual axes for your keys, buttons, and other input devices. It also supports multitouch and accelerometer data on mobile devices.

Note that enabling the Input System package (above) disables this built-in Input Manager.



Input Manager

This is where you can configure the controls to use with the UnityEngine.Input API. Consider using the new Input System Package instead.

Axes

Size	35
Horizontal	
Name	Horizontal
Descriptive Name	
Descriptive Negative Name	
Negative Button	left
Positive Button	right
Alt Negative Button	a
Alt Positive Button	d
Gravity	3
Dead	0.001
Sensitivity	3
Snap	☑
Invert	☐
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

Vertical

Fire1

Fire2

The built-in Input Manager

Here are some useful classes and methods for input:

Input	Used to read the axes set up in the Conventional Game Input and access multitouch/accelerometer data on mobile devices
Input.GetAxis	Returns the smoothed value for the virtual axis identified by axisName (value between -1 and 1 for keyboard/joystick devices); for a mouse device, returns the current mouse delta multiplied by the axis sensitivity
Input.GetAxisRaw	Like Input.GetAxis without the smoothing filter
Input.GetButton	Returns true while the virtual button identified by buttonName is held down
Input.GetButtonDown	Returns true during the frame the user pressed down the virtual button identified by buttonName
Input.GetKey	Returns true while the user holds down the key identified by name
Input.GetKeyDown	Returns true during the frame the user starts pressing down the key identified by name
Input.touches	A read-only list of objects representing status of all touches during the last frame
Touch	Structure describing the status of a finger touching the screen
KeyCode	Lists all of the key press, mouse and joystick options

For an overview of the built-in Input system, see the Input Manual page.

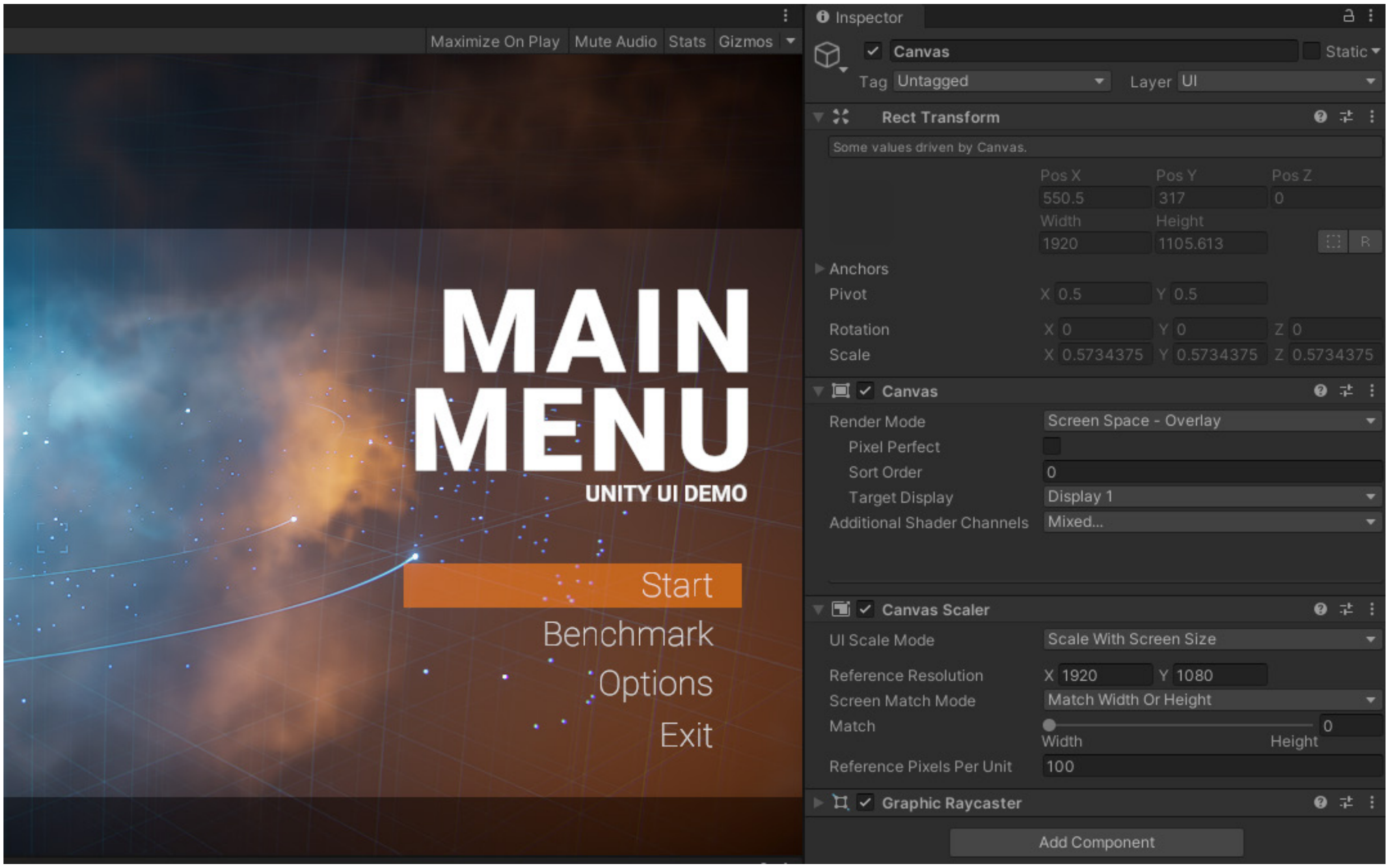
User interface

Unity implements several UI systems for either your in-game user interface or for Editor scripts. You may tap one or more of these, depending on your use case.

Unity GUI

Unity UI (sometimes referred to as UGUI) offers a GameObject-based approach for editing and positioning UI elements. Unity UI is a system for developing user interfaces for games and applications. It uses components and the Game view to arrange, position, and style user interfaces. You cannot use Unity UI to create or change user interfaces in the Unity Editor.

The Unity UI guide covers how to lay out your UI elements and create interactions.



☑ Canvas □ Static

Tag Untagged Layer UI

Rect Transform

Some values driven by Canvas.

Pos X 550.5 Pos Y 317 Pos Z 0

Width 1920 Height 1105.613

Anchors

Pivot X 0.5 Y 0.5

Rotation X 0 Y 0 Z 0

Scale X 0.5734375 Y 0.5734375 Z 0.5734375

☑ Canvas

Render Mode Screen Space - Overlay

Pixel Perfect □

Sort Order 0

Target Display Display 1

Additional Shader Channels Mixed...

☑ Canvas Scaler

UI Scale Mode Scale With Screen Size

Reference Resolution X 1920 Y 1080

Screen Match Mode Match Width Or Height

Match 0 (Width – Height)

Reference Pixels Per Unit 100

☑ Graphic Raycaster

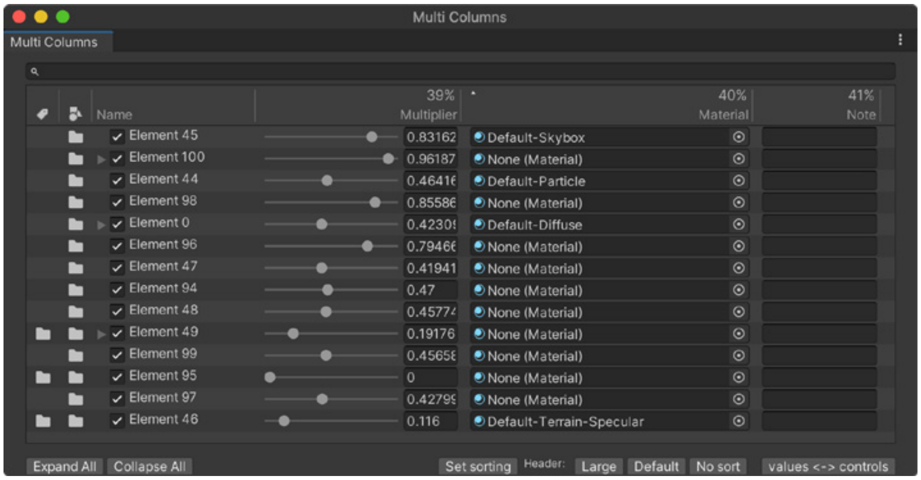
Unity UI offers a GameObject-based approach for UI elements.

Immediate Mode GUI

Immediate Mode GUI (IMGUI) can help you build custom inspectors and Editor windows. To create IMGUI elements, you write code within a special MonoBehaviour function named OnGUI. This system is not generally intended for normal in-game user interfaces.

The code displays the interface in "immediate mode," executed every frame. There are no persistent GameObjects other than the object where your OnGUI code is attached. The code produces GUI controls that are drawn and handled with a single function call.

Follow this scripting guide for when you need IMGUI.



		Name	39% Multiplier	40% Material	41% Note
	■	☑ Element 45	0.83162	Default-Skybox	
	■	▸ ☑ Element 100	0.96187	None (Material)	
	■	☑ Element 44	0.46416	Default-Particle	
	■	☑ Element 98	0.85586	None (Material)	
	■	▸ ☑ Element 0	0.42309	Default-Diffuse	
	■	☑ Element 96	0.79466	None (Material)	
	■	☑ Element 47	0.41941	None (Material)	
	■	☑ Element 94	0.47	None (Material)	
	■	☑ Element 48	0.45774	None (Material)	
■	■	▸ ☑ Element 49	0.19176	None (Material)	
	■	☑ Element 99	0.45658	None (Material)	
■	■	☑ Element 95	0	None (Material)	
	■	☑ Element 97	0.42799	None (Material)	
■	■	☑ Element 46	0.116	Default-Terrain-Specular	

An Editor UI created with Immediate Mode GUI

Profiling and optimization

Once you've designed a great game, your next task is to make it fast. Unity includes a suite of profiling and optimization tools to help you maximize the available resources for your target platform.

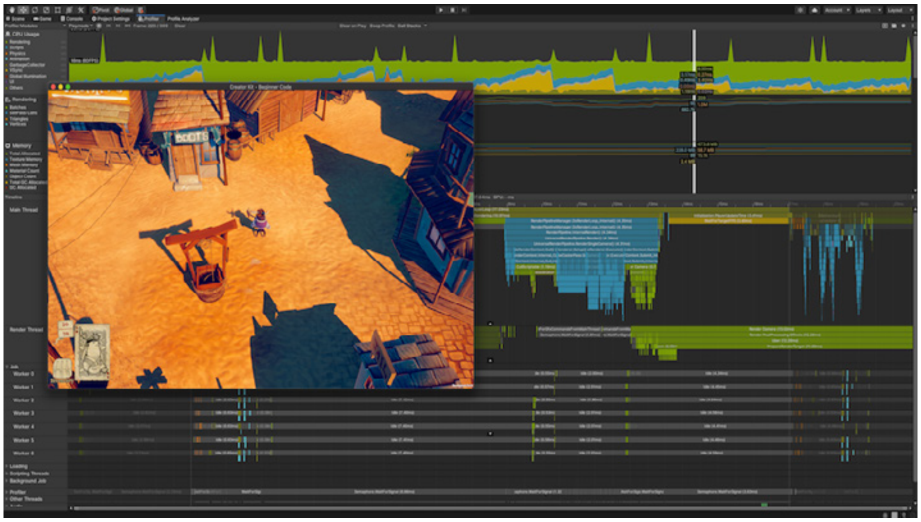
Unity Profiler

The Unity Profiler gives you performance information about the application. For best results, run it on the end platform where you intend to publish your build. This gives you accurate timings about what impacts the performance of your application.

The Profiler can also work directly from the Editor in Play mode. Just be aware that this setup sacrifices accuracy, so it's useful as a quick check but not for a final assessment.

The Profiler can help you identify areas for potential improvement in your application: the CPU, GPU, memory, renderer, physics, and audio. Iterate on those areas. You can pinpoint things like how your code, assets, scene settings, camera rendering, and build settings affect how well your application runs.

The Profiler displays the results in a series of graphs so you can visualize where spikes happen.



The Unity Profiler

Unity provides a number of Profiler markers to give you insight into what is taking up time in your application. The ProfilerRecorder can also use these markers to get useful timings for your frame.

Here are some of the most common markers:

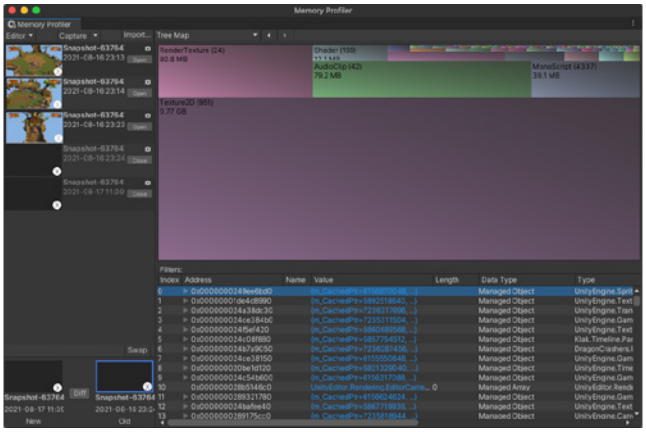
Profiler marker	**Summary**
Main thread base markers	Markers on the main thread containing samples from the main game loop, PlayerLoop, as well as the EditorLoop (present when you profile inside of the Editor)
Script update markers	Contain samples of MonoBehaviour.Update methods (Update/FixedUpdate/LateUpdate) and coroutines
Rendering and VSync markers	Contain the samples where the CPU spends time processing data for the GPU, or where it might be waiting for the GPU to finish
Back end scripting markers	Highlight Mono or IL2CPP scripting backend activities and can be useful for troubleshooting issues with garbage collection and allocation
Multithreading markers	Contain samples that do not measure the CPU cycles consumed, instead highlighting information that relates to thread synchronization and the Job System
Physics markers	Include high-level physics Profiler markers that sample Physics.Contacts, Physics.TriggerEnterExits, Physics.UpdateBodies, etc.
Performance warnings	Displayed when the Profiler detects some specific calls that you should avoid in performance-critical contexts

To access the Profiler window, navigate to the menu: **Window > Analysis > Profiler**. For a detailed overview of the window, see the Profiler window documentation.

Memory Profiler

The Memory Profiler is a tool you can use to identify areas in your Unity project and the Unity Editor where you can reduce memory usage. It gives you an overview of native and managed memory allocations, as well as the references between them that keep them in memory.

Use the Memory Profiler to capture, inspect, and compare memory snapshots to detect memory leaks and fragmentation.



Filters:

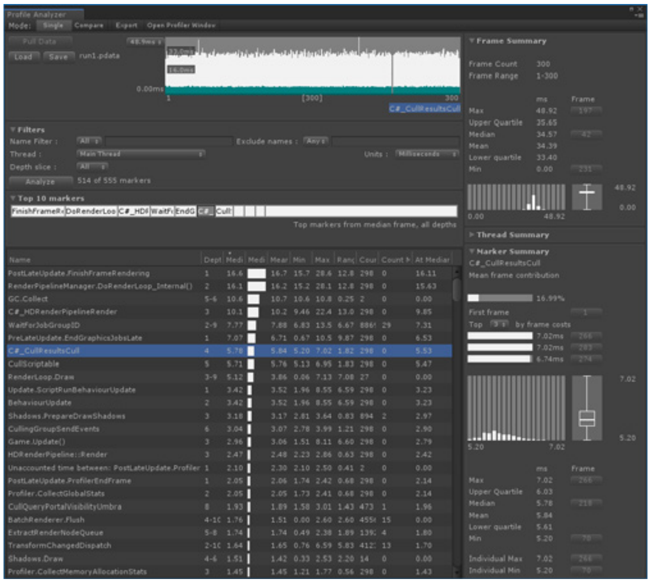
Index	Address	Name	Value	Length	Data Type	Type
0	▸ 0x0000000249ee6bd0		{m_CachedPtr=6156870048, ...}		Managed Object	UnityEngine.Sprit
1	▸ 0x00000001de4c8990		{m_CachedPtr=5892518640, ...}		Managed Object	UnityEngine.Text
2	▸ 0x000000024a38dc30		{m_CachedPtr=7236317696, ...}		Managed Object	UnityEngine.Tran
3	▸ 0x000000024ce384b0		{m_CachedPtr=7235311504, ...}		Managed Object	UnityEngine.Gam
4	▸ 0x000000024f5ef420		{m_CachedPtr=5860689568, ...}		Managed Object	UnityEngine.Text
5	▸ 0x000000024c08f880		{m_CachedPtr=5857754512, ...}		Managed Object	Klak.Timeline.Par
6	▸ 0x000000024b7a9050		{m_CachedPtr=7236287456, ...}		Managed Object	DragonCrashers.I
7	▸ 0x000000024ce38150		{m_CachedPtr=6155550848, ...}		Managed Object	UnityEngine.Gam
8	▸ 0x000000020be1d120		{m_CachedPtr=5821329040, ...}		Managed Object	UnityEngine.Time
9	▸ 0x000000024c54b600		{m_CachedPtr=6156317088, ...}		Managed Object	UnityEngine.Gam
10	0x000000028b5146c0		UnityEditor.Rendering.EditorCame...	0	Managed Array	UnityEditor.Rend
11	▸ 0x0000000289321780		{m_CachedPtr=6156624624, ...}		Managed Object	UnityEngine.Gam
12	▸ 0x000000024bafee40		{m_CachedPtr=5867719936, ...}		Managed Object	UnityEngine.Text
13	▸ 0x00000002891 75cc0		{m_CachedPtr=7235818944, ...}		Managed Object	UnityEngine.Cam

The Memory Profiler main view in tree map view

Profile Analyzer

The Profile Analyzer aggregates and visualizes frame and marker data from a set of Unity Profiler frames to help you understand their behavior. You can use the Profile Analyzer to compare two sets of data side-by-side, complementing the single-frame analysis already available in the Unity Profiler.

For information on how to use the Profile Analyzer, see the window's documentation.



Filters

Name Filter : All
Exclude names : Any
Thread : Main Thread
Units : Milliseconds
Depth slice : All
Analyze 514 of 555 markers

Top 10 markers

FinishFrameR | DoRenderLoo | C#_HDI | WaitF | EndG | C#_ | Cull

Top markers from median frame, all depths

Name	Dept	Medi	Medi	Mear	Min	Max	Ranç	Cour	Count ▸	At Mediar
PostLateUpdate.FinishFrameRendering	1	16.6		16.7	15.7	28.6	12.8	298	0	16.11
RenderPipelineManager.DoRenderLoop_Internal()	2	16.1		16.2	15.2	28.1	12.8	298	0	15.63
GC.Collect	5-6	10.6		10.7	10.6	10.8	0.25	2	0	0.00
C#_HDRenderPipelineRender	3	10.1		10.2	9.46	22.4	13.0	298	0	9.85
WaitForJobGroupID	2-9	7.77		7.88	6.83	13.5	6.67	886	29	7.31
PreLateUpdate.EndGraphicsJobsLate	1	7.07		6.71	0.67	10.5	9.87	298	0	6.53
C#_CullResultsCull	4	5.78		5.84	5.20	7.02	1.82	298	0	5.53
CullScriptable	5	5.71		5.76	5.13	6.95	1.83	298	0	5.47
RenderLoop.Draw	3-9	5.12		3.86	0.06	7.13	7.08	27	0	0.00
Update.ScriptRunBehaviourUpdate	1	3.42		3.52	1.96	8.55	6.59	298	0	3.23
BehaviourUpdate	2	3.42		3.52	1.96	8.55	6.59	298	0	3.23
Shadows.PrepareDrawShadows	3	3.18		3.17	2.81	3.64	0.83	894	2	2.97
CullingGroupSendEvents	6	3.04		3.07	2.78	3.99	1.21	298	0	2.90
Game.Update()	3	2.96		3.06	1.51	8.11	6.60	298	0	2.79
HDRenderPipeline::Render	3	2.47		2.48	2.23	2.86	0.63	298	0	2.42
Unaccounted time between: PostLateUpdate.Profiler	1	2.10		2.30	2.10	2.50	0.41	2	0	0.00
PostLateUpdate.ProfilerEndFrame	1	2.05		2.06	1.74	2.42	0.68	298	0	2.14
Profiler.CollectGlobalStats	2	2.05		2.05	1.73	2.41	0.68	298	0	2.14
CullQueryPortalVisibilityUmbra	8	1.93		1.89	1.58	3.01	1.43	473	1	1.96
BatchRenderer.Flush	4-1C	1.76		1.51	0.00	2.60	2.60	455	15	0.00
ExtractRenderNodeQueue	5-8	1.74		1.74	0.49	2.38	1.89	139	4	1.80
TransformChangedDispatch	2-1C	1.64		1.65	0.76	6.59	5.83	412	13	1.70
Shadows.Draw	4-6	1.51		1.42	0.33	2.53	2.20	14	0	0.00
Profiler.CollectMemoryAllocationStats	3	1.45		1.45	1.21	1.77	0.56	298	0	1.43

The Profile Analyzer

For a better understanding on optimization in Unity for mobile platforms, see this blog post, Optimize your mobile game performance. You can also download the full e-book outlining Unity experts' optimization tips for mobile games.

Debugging and playtesting

Unity is an excellent tool for tweaking and debugging. At any time, you can enter Editor Play mode and see all of the gameplay variables while the application runs in the Editor.

In Play mode, the Game view gives you a preview of your final, published application. You can alter the Unity scene on the fly and experiment, pausing any time or stepping through the code one statement at a time.

To assist with playtesting, you'll likely create cheats that will allow you to:

- Unlock levels, characters, items, etc.
- Disable enemies and/or gameplay
- Toggle GUIs
- Grant invincibility
- Add/subtract time, money, collectibles, etc.

Debugging gameplay

Playtesting isn't an exact science, but consider these suggestions:

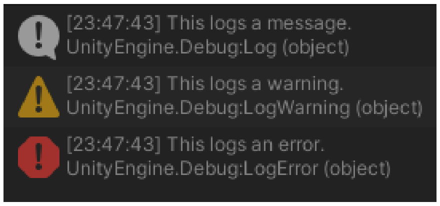
- Implement shortcuts for printing the player's world position. This helps you determine whether specific bugs happen at a particular place in the level.
- For small teams, make a test Prefab for each team member, and read settings and debug options from an uncommitted file. This way, team members won't accidentally commit testing options or change the production scene.
- Maintain a test/sandbox scene with all gameplay elements. For instance, create a scene with all enemies, all objects you can interact with, etc. This makes it easy to test functionality without having to play through the entire game.
- Write a set of in-Editor cheats. Attach a MenuItem attribute to a static method that can check if the Application.isPlaying, and then run some logic.

Additional debugging tips

Unity also includes a Debug class to help you visualize information in the Editor while it's running. Use this to print messages or warnings into the Console window, which shows errors, warnings, and other messages generated by Unity.

You can also use Debug to draw visualization lines in the Scene view and Game view, as well as to pause Editor Play mode from a script.

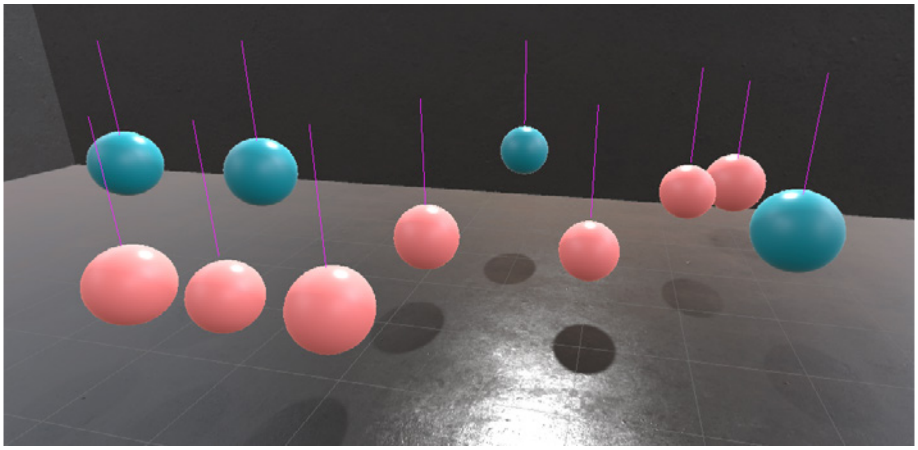
- Pausing execution with **Debug.Break** is useful if you want to check certain values in the Inspector when the application is difficult to pause manually.
- You can format your Console messages with different degrees of severity using **Debug.Log**, **Debug.LogWarning**, and **Debug.LogError**.



```
[23:47:43] This logs a message.
UnityEngine.Debug:Log (object)
[23:47:43] This logs a warning.
UnityEngine.Debug:LogWarning (object)
[23:47:43] This logs an error.
UnityEngine.Debug:LogError (object)
```

Log messages, warnings, and errors in the Console

- When using **Debug.Log**, you can pass in an object as the context. If you click on the message in the Console, Unity highlights the GameObject in the Hierarchy window.
- Use Rich Text to mark up your **Debug.Log** statements. This can help you enhance error reports in the Console.
- Troubleshooting physics? **Debug.DrawLine** and **Debug.DrawRay** can help you visualize ray casting.



Debug.DrawLine

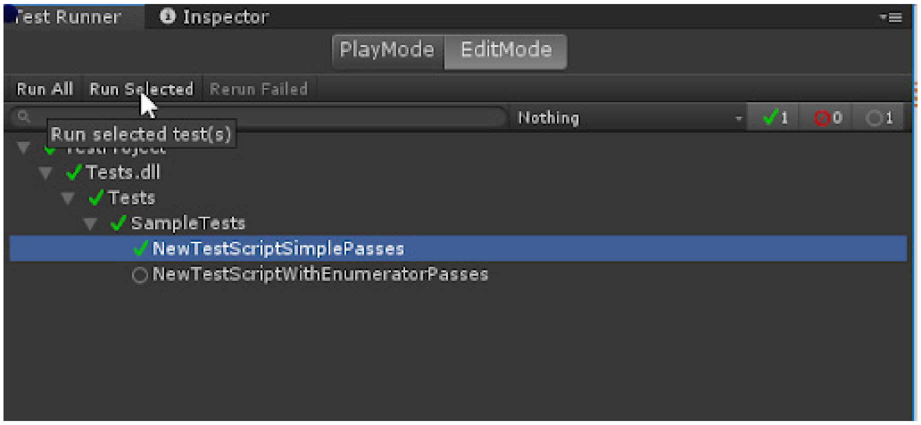
Unity Test Framework (UTF)

The Unity Test Framework (formerly known as the Unity Test Runner) allows you to create automated tests to make sure your code runs as intended. Create unit tests for any logical snippet of code and apply test-driven development while you develop the project.

UTF enables you to test your code in both Edit mode and Play mode. You can also run your tests on target builds such as Standalone, Android, iOS, and more. UTF extends the NUnit library, an open-source unit testing library for .NET languages.

To open the Test Framework, go to **Window > General > Test Runner**. Follow the instructions to set up a work folder and assembly definition, then add unit tests to your Test Assembly. This process can help you to isolate bugs faster and learn to write a testable way.

For more information about the Test Framework, see the Performance Benchmarking in Unity blog post and the Unity Test Framework documentation.



Unity Test Framework

Particle effects

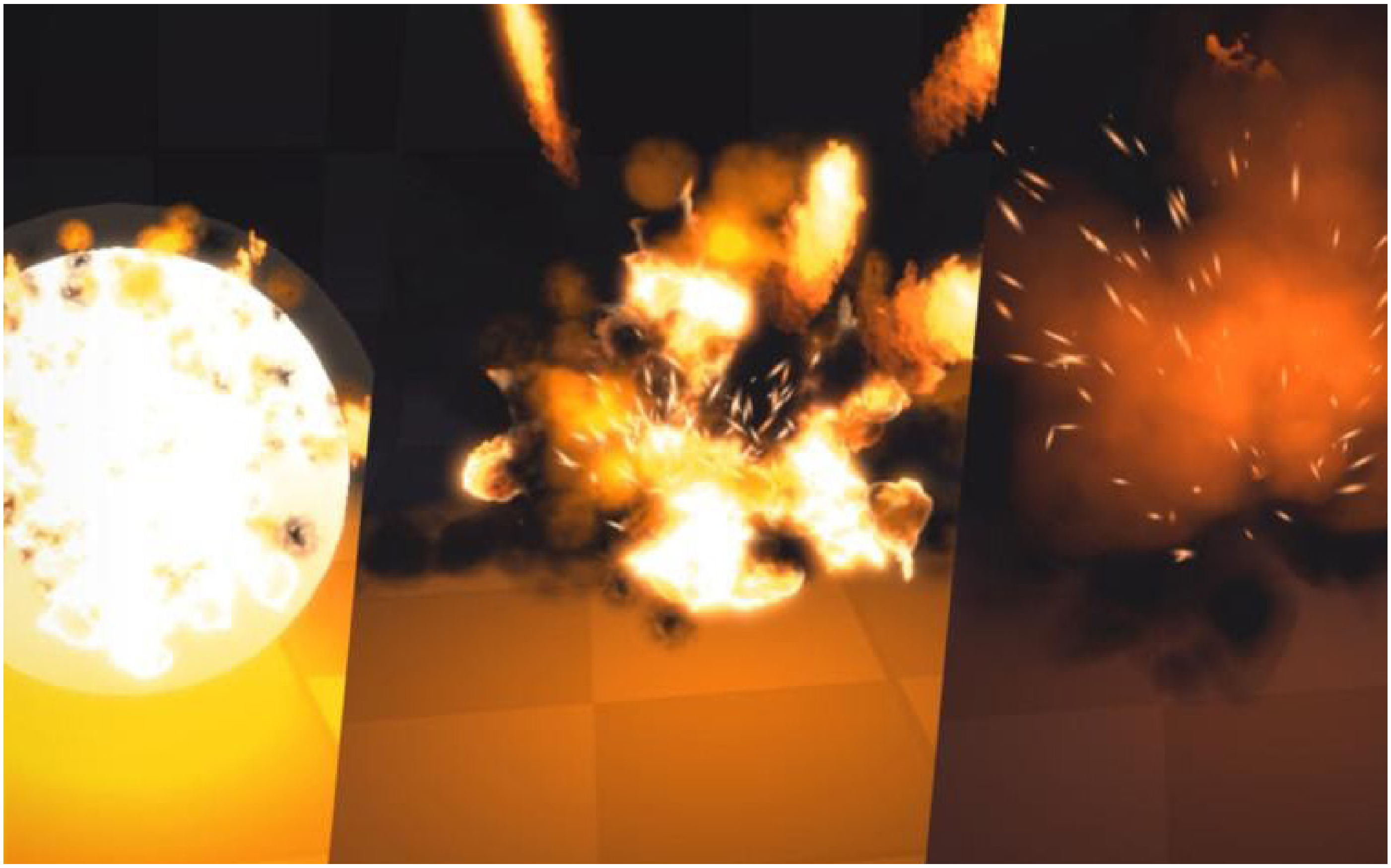
Unity includes two particle simulation solutions for your effects needs (smoke, liquids, flames):

— The **Particle System** can simulate thousands of particles on the CPU. Use the modules of the Particle System component to represent predefined behaviors. To create each effect, you will often layer several GameObjects with ParticleSystems together.

ParticleSystem supports the Built-in Render Pipeline and URP. You can access the ParticleSystem class to define a system and its individual particles via script.

Particle Systems can interact with Unity's underlying physics system and any Colliders in your scene, but cannot access frame buffers.

For examples of the built-in ParticleSystem, download the Particle Pack from the Asset Store.



A simple effects simulation using the Particle System

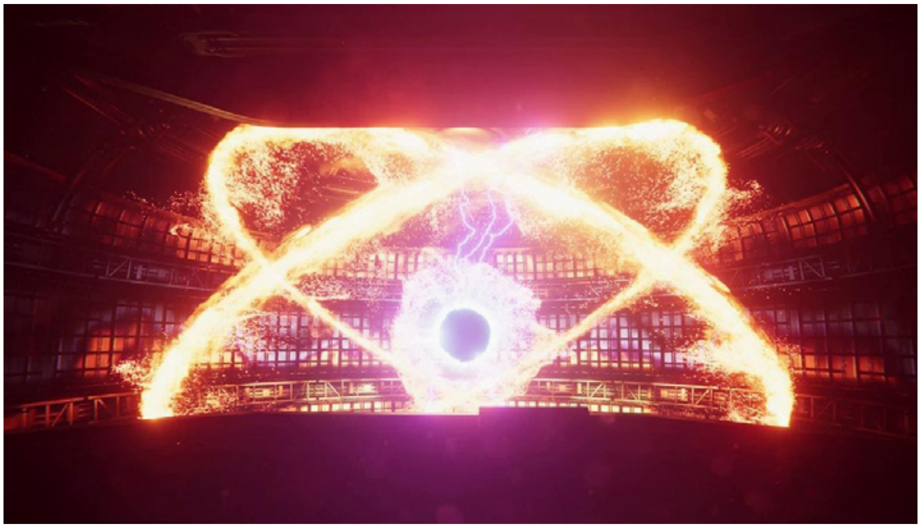
— The **Visual Effect Graph** moves calculations on the GPU using compute shaders. This can simulate millions of particles that can also interact with the color and depth buffer.

The workflow includes a highly customizable graph view.

While it does not have access to the underlying physics system, a Visual Effect Graph can interact with complex assets such as Point Caches, Vector Fields, and Signed Distance Fields.

Visual Effect Graph only works on platforms that support compute shaders and HDRP (support for URP is currently in Preview). You can use the event interface to send custom events and pass in attached data that the graph can process. The Visual Effect component also provides a playback control API.

Unity maintains two GitHub projects that demonstrate production examples of the Visual Effect Graph, the Visual Effect Graph Samples project and the Spaceship Demo. These both illustrate how the Visual Effect Graph can create a myriad of high-quality effects.



Millions of particles onscreen created with the Visual Effect Graph

When selecting one of the two systems, keep device compatibility in mind. Most PCs and consoles support compute shaders, but many mobile devices do not. We currently recommend the built-in ParticleSystem if your intended platform is mobile. If your target platform does support compute shaders, Unity allows you to use both types of particle simulation in your project.

Physics

Unity implements the NVIDIA PhysX engine for 3D projects and a 2D engine for 2D projects. The built-in physics engine in Unity provides industry standard solutions for Rigidbody interactions, joints, and forces. Use the Physics API for methods to ray cast or shape cast, test overlaps, and handle triggers/collisions.

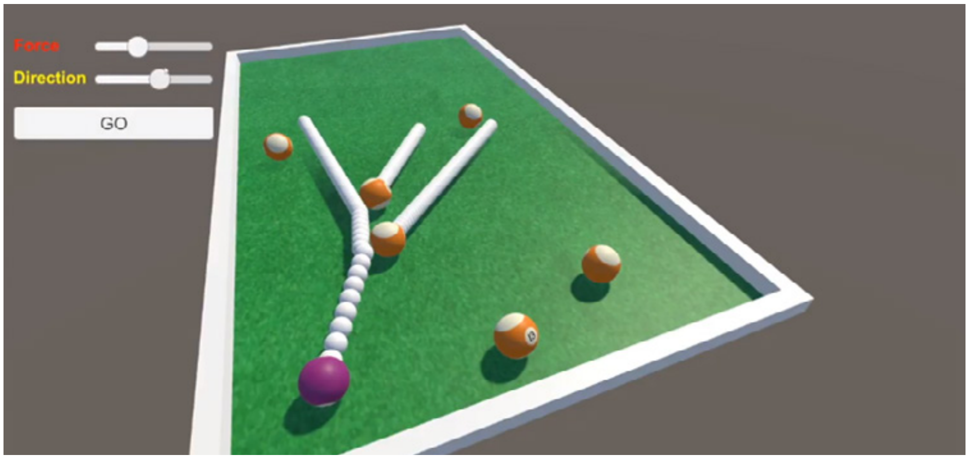
3D physics

Unity's built-in physics engine includes several useful components:

- **Rigidbody** allows the physics engine to control movement of the object. A Rigidbody will fall with gravity, and you can apply force or torque to it using the supplied methods. If a Collider component is active, the underlying GameObject will respond to physics collisions.
- **Colliders** (Box Collider, Sphere Collider, Capsule Collider, Mesh Collider) represent primitive or mesh volumes that define the boundaries for physics collisions.
- **Joints** (Fixed Joint, Hinge Joint, Spring Joint, Character Joint) allows you to connect two Rigidbody components to model a variety of motion constraints (e.g., breakable joint, one axis of rotation, elastic joint, ball and socket, etc.).
- **CharacterController** allows you to handle movement constrained by collisions without requiring a Rigidbody.

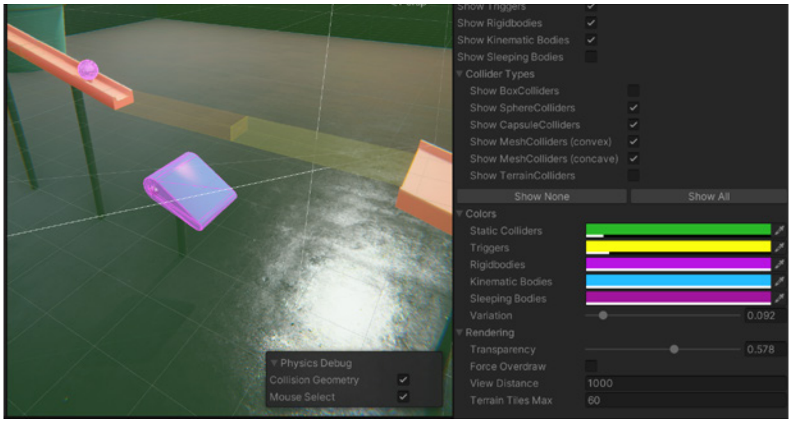
The Rigidbody component features speculative and sweep-based continuous collision detection for fast-moving objects.

Use multiphysics scenes to manage or work around complex physics contexts. For example, you could simulate multiple physics scenes in order to predict GameObject collisions and trajectories.



Use multiple scenes to predict collisions or trajectories.

Unity also provides a Debug Visualization window (**Window > Analysis > Physics Debugger**), an interface that allows you to color and toggle physics components by category. This helps quickly troubleshoot Colliders or other physics-based scenarios in your scene.



Show Triggers ✓
Show Rigidbodies ✓
Show Kinematic Bodies ✓
Show Sleeping Bodies □
▾ Collider Types
Show BoxColliders □
Show SphereColliders ✓
Show CapsuleColliders ✓
Show MeshColliders (convex) ✓
Show MeshColliders (concave) ✓
Show TerrainColliders □
Show None | Show All
▾ Colors
Static Colliders 🟩
Triggers 🟨
Rigidbodies 🟪
Kinematic Bodies 🟦
Sleeping Bodies 🟪
Variation 0.092
▾ Rendering
Transparency 0.578
Force Overdraw □
View Distance 1000
Terrain Tiles Max 60

▾ Physics Debug
Collision Geometry ✓
Mouse Select ✓

The Physics Debugger window

Apply physics updates inside of MonoBehaviour's FixedUpdate method, which is called on a fixed time step.The Physics class includes several static methods that you will use for physics interactions. Unity also provides several physics events on the Collider class to handle collisions and triggers. The following are some of the methods that you will use for common physics interactions.

Classes/Methods	**Description**
Monobehaviour.FixedUpdate	Message for physics calculations updates on a fixed frame-rate determined by **Edit > Settings > Time > Fixed Timestep**
Physics.Raycast, Physics.Linecast, Physics.BoxCast, Physics.CapsuleCast, Physics.SphereCast	Methods for determining if a collider falls within a ray, line, or shape
Physics.OverlapBox, Physics OverlapSphere, Physics OverlapCapsule	Methods for testing whether a collider falls within a given shape
Collider.OnCollisionEnter, Collider.OnCollisionStay, Collider.OnCollisionExit	Messages sent when an incoming collider makes contact/stays in contact/leaves contact with this object's collider
Collider.OnTriggerEnter, Collider.OnTriggerStay, Collider.OnTriggerExit	Messages sent when an incoming trigger makes contact/stays in contact/leaves contact with this object's collider (a trigger does not register an actual collision but sends messages)

2D physics

Unity has a separate physics engine optimized for handling 2D physics. Most 2D physics components are simply “flattened” versions of the 3D equivalents.

The corresponding components have “2D” appended to the name, e.g., Physics2D, Rigidbody2D, Collider2D, Joint2D, etc. Otherwise, they behave like their analogous 3D classes.

Be sure to download the PhysicsExamples2D project for some example use cases, and watch this accompanying introductory video.

Read more about the built-in 3D and 2D physics implementations in the Physics documentation. For tips about optimizing 3D physics and its settings, watch Physics Performance Optimization.

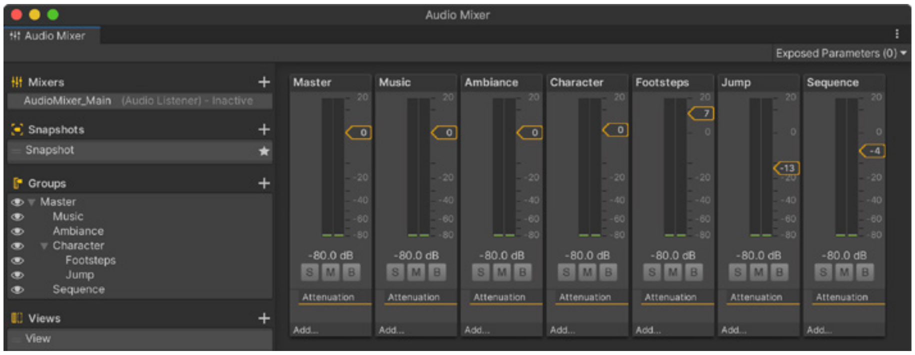
Audio

Unity's audio system has sophisticated features for playing sounds in 3D space. Unity can also record audio with any available microphone for use during gameplay or for storage and transmission.

Audio components mimic their real-life counterparts. Attach AudioSources to your GameObjects so they can play back an audio file called an AudioClip. An AudioListener somewhere else in the scene (usually attached to your main Camera) then hears this clip.

In this way, the audio system can replicate the Doppler effect if the source and listener move relative to one another. Simulate other effects, like echoes, using Audio Filters.

An AudioMixer blends various audio sources, applies effects to them, and performs mastering.



Caption: The AudioMixer

These are the most common audio component classes.

Class	Description
AudioClip	Any Audio file imported into Unity is available from scripts as an AudioClip. This provides a way for the audio system to access the encoded audio data at runtime. Meta-information from the AudioClip may load even before the actual audio data does.
AudioSource	An AudioSource attached to a GameObject plays back sounds in a 3D environment.
AudioListener	An AudioListener represents a microphone-like device that hears all sounds around it. You have one listener per scene.
AudioMixer	An Audio Mixer group is essentially a signal chain which allows you to apply volume attenuation and pitch correction; it allows you process the audio signal with effects and perform mastering.

Render pipelines and graphics

No game is complete without graphics. Before you begin lighting your scenes, you will need to choose between one of the different render pipelines. A render pipeline performs a series of operations that take the contents of a scene to display them on-screen.

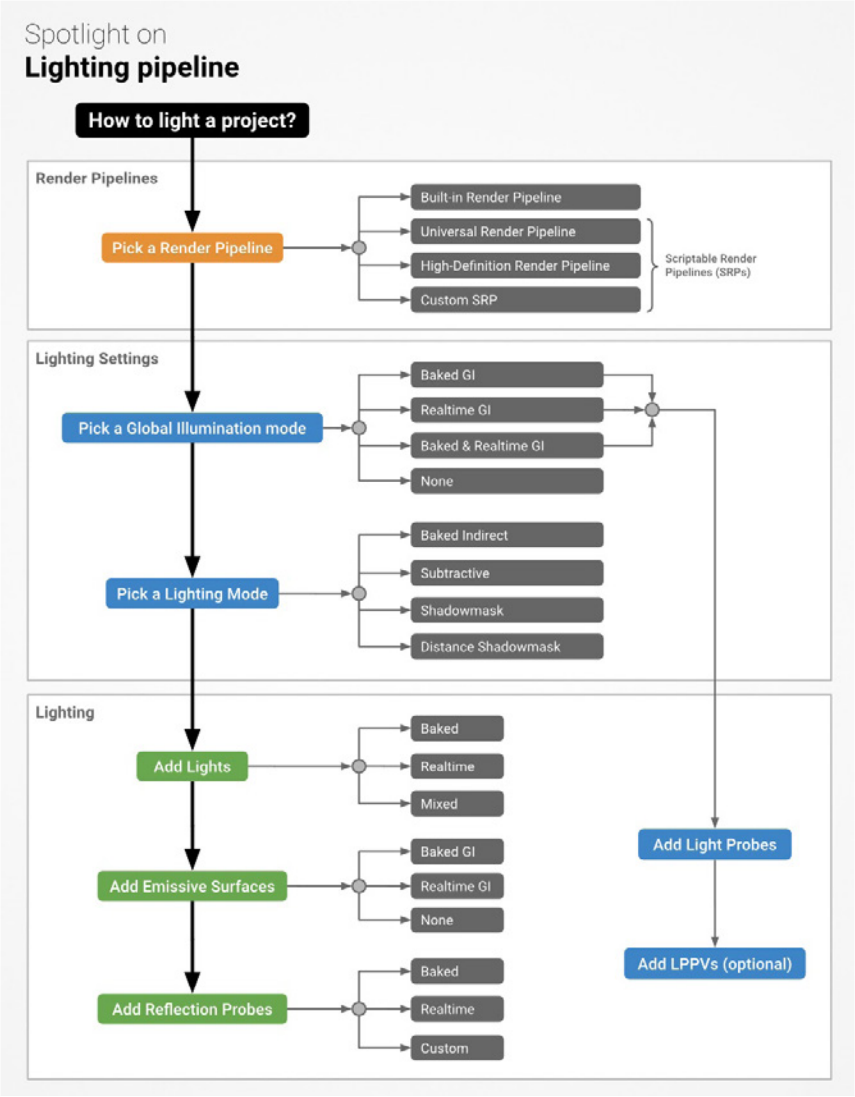
3D pipelines

Unity provides three prebuilt render pipelines with different capabilities and performance characteristics. You can also create your own.

- The **Built-in Render Pipeline** is a general purpose render pipeline with limited customization. This is the default.
- The **Universal Render Pipeline (URP)** is a prebuilt Scriptable Render Pipeline (SRP). URP provides artist-friendly workflows to create optimized graphics. Consider URP if you want to target multiple platforms – from mobile to desktop to consoles – with a single Unity project.

  URP adds graphics and rendering features unavailable to the Built-in Render Pipeline. In order to maintain performance, it makes tradeoffs to reduce the computational cost of lighting and shading. URP supports and scales to all Unity supported platforms. Choose URP if you want to build for mobile, Nintendo Switch, or Oculus Quest.
- The **High Definition Render Pipeline (HDRP)** is another prebuilt Scriptable Render Pipeline, designed for high-end hardware such as PC, Xbox, and PlayStation. HDRP supports the highest-quality physically based lighting and rendering available in Unity. Choose HDRP if photorealism is your goal.

  HDRP features different light types (Spotlight Pyramid/Box, Realtime Tube, and Realtime Rectangular) and more advanced Reflections (Probes with blending, Planar reflections, Screen space reflections). Fine-tune your materials with high-end shaders and effects, and use the pipeline debugging tools to troubleshoot renders.



Choose a render pipeline early when planning your project.

Both HDRP and URP include:

- Shader Graph, a tool that allows you to create shaders using a visual node editor instead of writing code in HLSL. This allows developers to offload some shader work to artists or technical artists.
- Sample scenes, which show examples of how to set up lighting settings, materials, and shaders.

  Several preconfigured Render Pipeline Assets let you quickly swap between graphics quality levels with settings that are already optimized for their respective pipeline.
- The latest Post Processing Stack – both URP and HDRP include their own post-processing systems optimized for their respective pipelines. These enable you to apply full-screen filters using an artist-friendly interface.

- A Render Pipeline Debug utility with visualization tools that you can use to solve issues with lighting, shading, shadows, etc.

URP and HDRP work on top of the Scriptable Render Pipeline, a thin API layer that lets you schedule and configure rendering commands using C# scripts. This flexibility allows you to customize virtually every part of the pipeline. You can also create your own custom render pipeline based on SRP.

While the Built-in Render Pipeline is not as customizable as URP or HDRP, it does support a wide number of platforms. To use it, you'll need to configure the different rendering paths and extend its functionality with command buffers and callbacks.

The Heretic short film showcases HDRP's high-end graphical capabilities.

See Render pipelines in Unity for a more detailed comparison of the available pipelines.

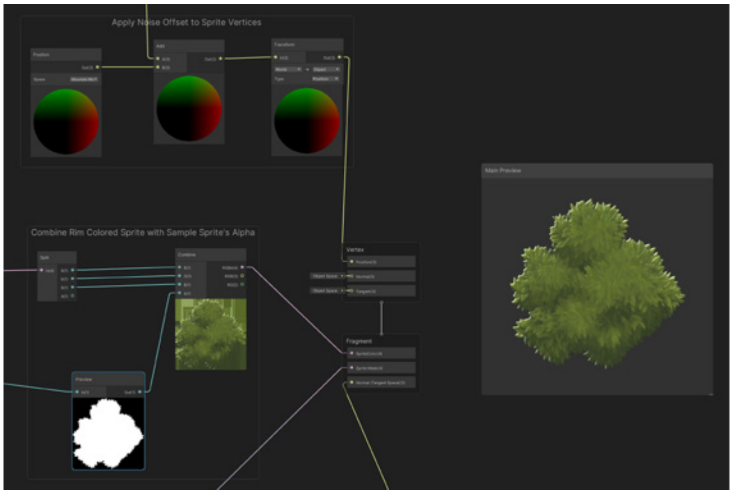
2D pipeline

The URP also includes a **2D Renderer**. If you're building a sprite-based game, this includes tools specifically designed to help you create a 2D experience:

- **2D Lights** can take a variety of shapes (Freeform, Sprite, Parametric, Point, and Global). These can interact with normal map and mask Textures when applied as a Secondary Texture in the Sprite Editor. This helps to create advanced lighting effects.

2D Lights in the *Lost Crypt* sample project

- **Shader Graph** includes Lit and Unlit Sprite Masternodes. These streamline the node input, so you can pass in the appropriate Sample Texture2D nodes in order to render in 2D.


Apply Noise Offset to Sprite Vertices
Combine Rim Colored Sprite with Sample Sprite's Alpha
Main Preview

2D Shader Graph

- The **2D Pixel Perfect package** contains the Pixel Perfect Camera component, which ensures your pixel art remains crisp and clear at different resolutions, and stable in motion.

We recommend that you start with one of the available 2D demos, such as the *Lost Crypt* and *Dragon Crashers* 2D sample projects. These should inspire you on how to assemble the 2D tools into a real-time experience.

Worldbuilding

Your games will need some levels. These tools can help.

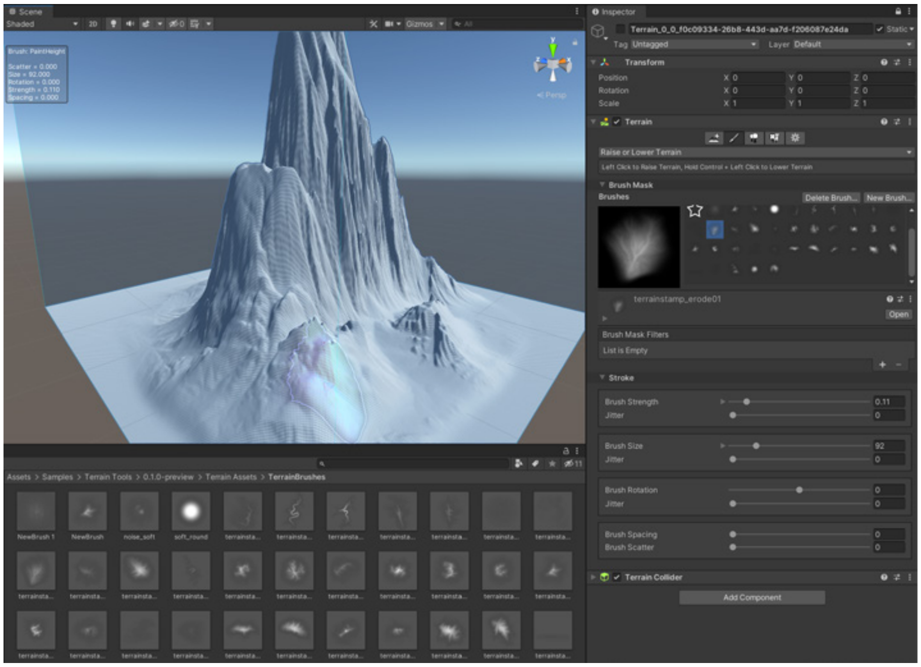
ProBuilder and Polybrush

Unity includes an add-on package called ProBuilder, a streamlined 3D modeling and level design tool. Use ProBuilder to quickly prototype structures or to make custom collision geometry, trigger zones, or nav meshes. This is optimized for building simple geometry and greyboxing your game levels.

Polybrush gives you the ability to sculpt meshes, blend textures, paint vertex colors, and scatter objects over your levels. Together with ProBuilder, you have a complete suite of tools for level design. Roundtrip with your DCC package of choice (Maya, Blender, etc.) to further refine your models.

Terrain tools

For complex natural 3D environments, the Unity Editor includes a built-in set of Terrain tools (available as a package in Unity 2021.2 and above) that allow you to add landscapes to your game. In the Editor, you can create multiple Terrain tiles, adjust the height or appearance of your landscape using brush based tools, and add trees or grass to it.



Brush: PaintHeight

Scatter = 0.000
Size = 92.000
Rotation = 0.000
Strength = 0.110
Spacing = 0.000

Terrain_0_0_f0c09334-26b8-443d-aa7d-f206087e24da ☑ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ Terrain
Raise or Lower Terrain
Left Click to Raise Terrain, Hold Control + Left Click to Lower Terrain

Brush Mask
terrainstamp_erode01
Brush Mask Filters
List is Empty

Stroke
Brush Strength 0.11
Jitter 0
Brush Size 92
Jitter 0
Brush Rotation 0
Jitter 0
Brush Spacing 0
Brush Scatter 0

☑ Terrain Collider

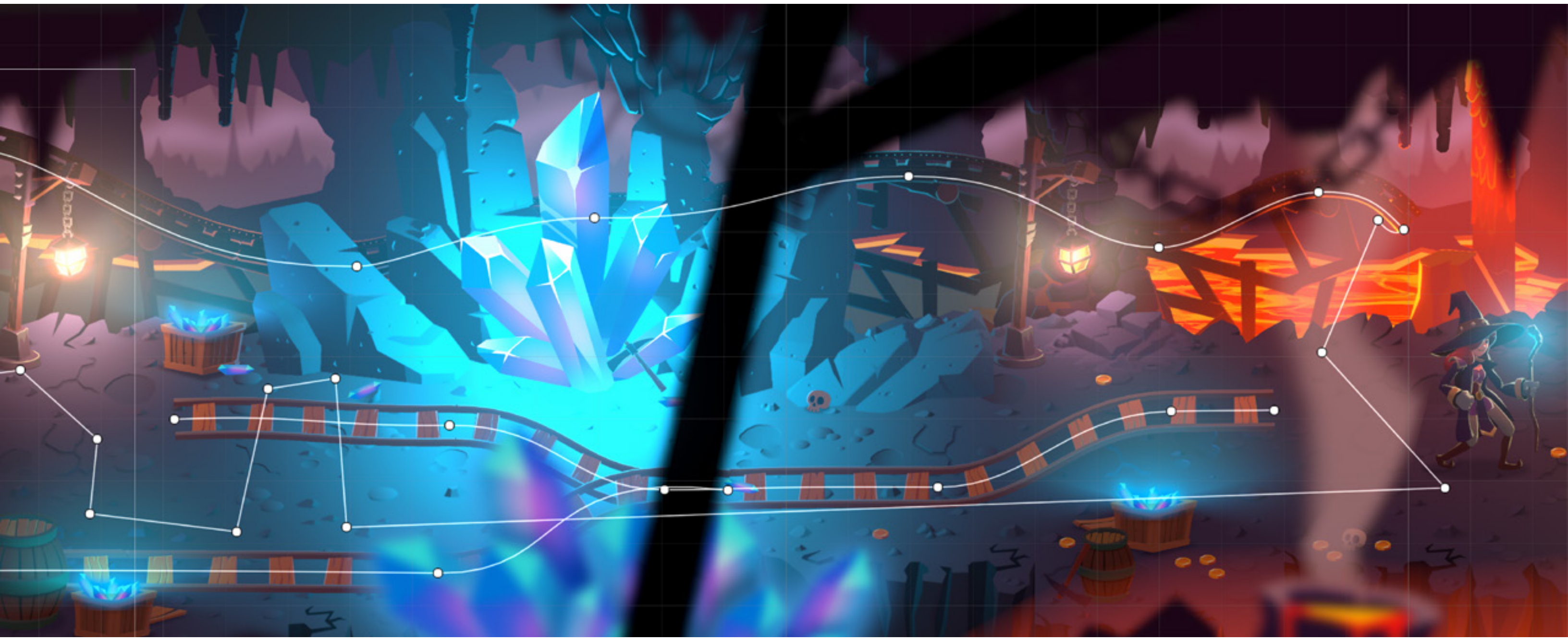
Assets > Samples > Terrain Tools > 0.1.0-preview > Terrain Assets > TerrainBrushes

Unity Terrain tools

2D Tilemap and Sprite Shape

Working in 2D? Create large grid-based worlds, including hexagonal and isometric environments, with Tilemap. This system stores and handles Tile Assets for creating 2D levels. Then, you can set up a Tile Palette and use a variety of Brushes to paint your 2D Tilemaps.

Sprite Shape gives you the freedom to create rich free-form and organic 2D environments with a visual and intuitive workflow. The tool works by dynamically tiling sprites along spline paths based on a given set of angle ranges. Sprites tiling on the spline automatically deform, and the tool swaps sprites based on the outline angle.



The SpriteShape tool lets you create organic 2D environments.

Unity Asset Store

The Unity Asset Store offers a variety of assets, from textures, animations, and models to entire project examples, tutorials, and Editor extensions. Both Unity Technologies and members of the community create assets and publish them to this active marketplace. You can download asset packages directly into your Unity project with the Package Manager.

To access the Asset Store online, log into your Unity account, then shop a variety of 2D, 3D, and scripting assets:

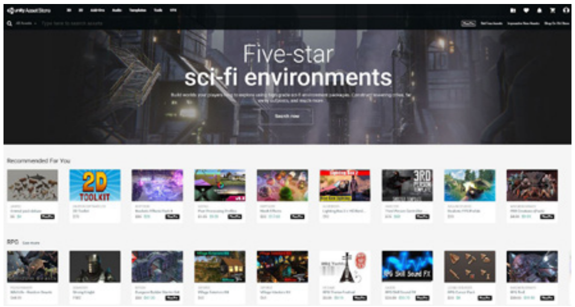
- The **Templates** section enables you to download various tutorials and starter packs.
- **Audio** has a library of sound files that you can use to enrich the user experience of your project, including ambient, music, voice, and sound effects.
- The **2D** and **3D** sections feature plenty of environments, props, and characters in all art styles and genres to fill out your game levels.
- Browse for **Tools and Editors** if you're short on development time.
- Discover systems for enhanced functions, from AI to Visual Scripting.

Importing an asset

Once you purchase a free or paid asset, it should appear on your Unity account. In the Package Manager (**Window > Package Manager**), filter for **My Assets**, then download and import any assets associated with your account into the current project.

Become a publisher

You can become a publisher on the Asset Store to sell your creations (3D models, Editor extensions, audio, and more) and add a new stream of revenue while you're working on your game. To get started, create an Asset Store Publisher account, check out the Submission Guidelines, and follow the Asset Store Provider Agreement. Submit your content using the Sell Assets page.



The Asset Store third-party asset marketplace

Learning resources

Take advantage of Unity's extensive documentation and learning resources to help you dive deeper into development.

Documentation

The Unity Manual and Scripting API are the most complete guides to Unity. The Manual documentation covers virtually every feature and tool, and we're constantly updating.

Packages have their own documentation microsites. You can always access the latest version from the Unity Manual or access a specific package version's documentation using the Package Manager window.

Unity Learn

Everyone has access to over 750 hours of free on-demand and live learning resources at Unity Learn. There, you can ask questions, get tips, and work through real-world projects with help from Unity experts and fellow creators.

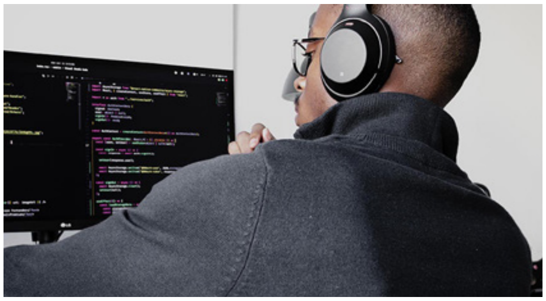
Unity Blog

The Unity Blog is a source for timely news, updates, and software releases, with short, readable articles covering everything from emergent technology to community events to intimate chats with Unity game creators. Whether you're an artist, programmer, technical artist, or designer, the blog offers tips and insights from the community to help you upskill.

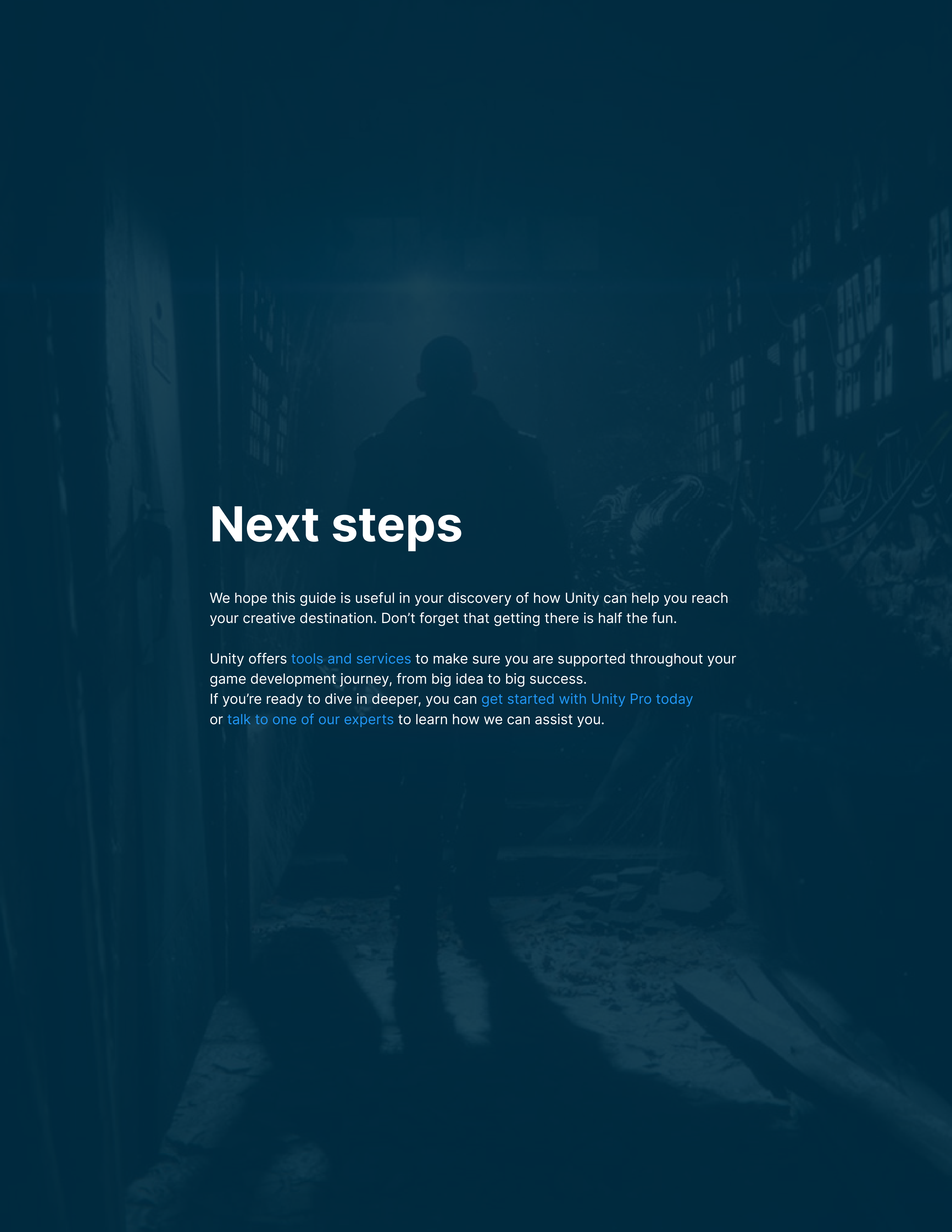
Professional training

Unity Professional Training now offers access to over 200 hours of training content with On-Demand Training. An annual subscription unlocks full access to curated courses, bite-sized video content, assessments, course challenges, and certificates.

Our experienced instructional designers created these materials in partnership with our engineers and product teams. This means that your team is always receiving the most up-to-date training on the latest Unity tech. See the course catalog to get started.



Unity Professional Training



Next steps

We hope this guide is useful in your discovery of how Unity can help you reach your creative destination. Don't forget that getting there is half the fun.

Unity offers tools and services to make sure you are supported throughout your game development journey, from big idea to big success.
If you're ready to dive in deeper, you can get started with Unity Pro today or talk to one of our experts to learn how we can assist you.

Unity®

unity.com