



70+ tips to increase productivity with Unity 2020 LTS

Contents

Introduction	4
Editor workflows	5
The Package Manager	5
Shortcuts Manager	6
Focused Inspectors	7
Presets	8
SceneVisibility	9
Scene picking	11
Searching	11
Inspector Debug Mode	12
QuickSearch	12
10 small workflow tips	14
Artist workflows	15
2D art and workflow tips	
Sprite Atlas	16
10 2D workflow tips	17
Prefab workflows	21
TextMeshPro	24
Snapping	25
Animation workflow	27
Custom gizmos and icons	29
Progressive Lightmapper	30
Light Probes	31

Developer workflows	33
Attributes	33
Custom windows and Inspectors	35
Custom menus	36
Enter Play Mode settings	36
Script templates	37
Addressables	39
Preprocessor directives	42
ScriptableObjects	43
Managing assemblies	46
IDE support	47
Debugging	48
Additional debugging tips	50
Visual Studio shortcuts	52
Device Simulator	53
Console Log Entry	54
Custom Compiler status	54
Team workflows	55
Source control	56
Unity Accelerator	58

Resources for all Unity developers

Introduction

This guide helps Unity creators save time and boost productivity with over 70 tips on how to work faster with programmer and artist toolsets, individually or on a team.

The tips here work with new and existing features in [Unity 2020 LTS](#). Whether you are a new or experienced Unity developer, reference this guide to speed up workflows in every stage of your game development.

Many teams at Unity work to improve the quality of life for our creators, such as the [Accelerate Solutions team](#) who contributed their valuable knowledge to this guide. The Accelerate Solutions team supports a plethora of Unity customers to help them get the most out of the engine. They identify, and help to optimize, critical points in projects for speed, stability, and efficiency.

As Aras Pranckevičius, one of Unity's first engineers and the leader of the Quality of Life team, says, "Unity should be a joy to use. When millions of users repeat a task multiple times per day, every second or mouse click adds up. We want creators to waste less time and be more productive."



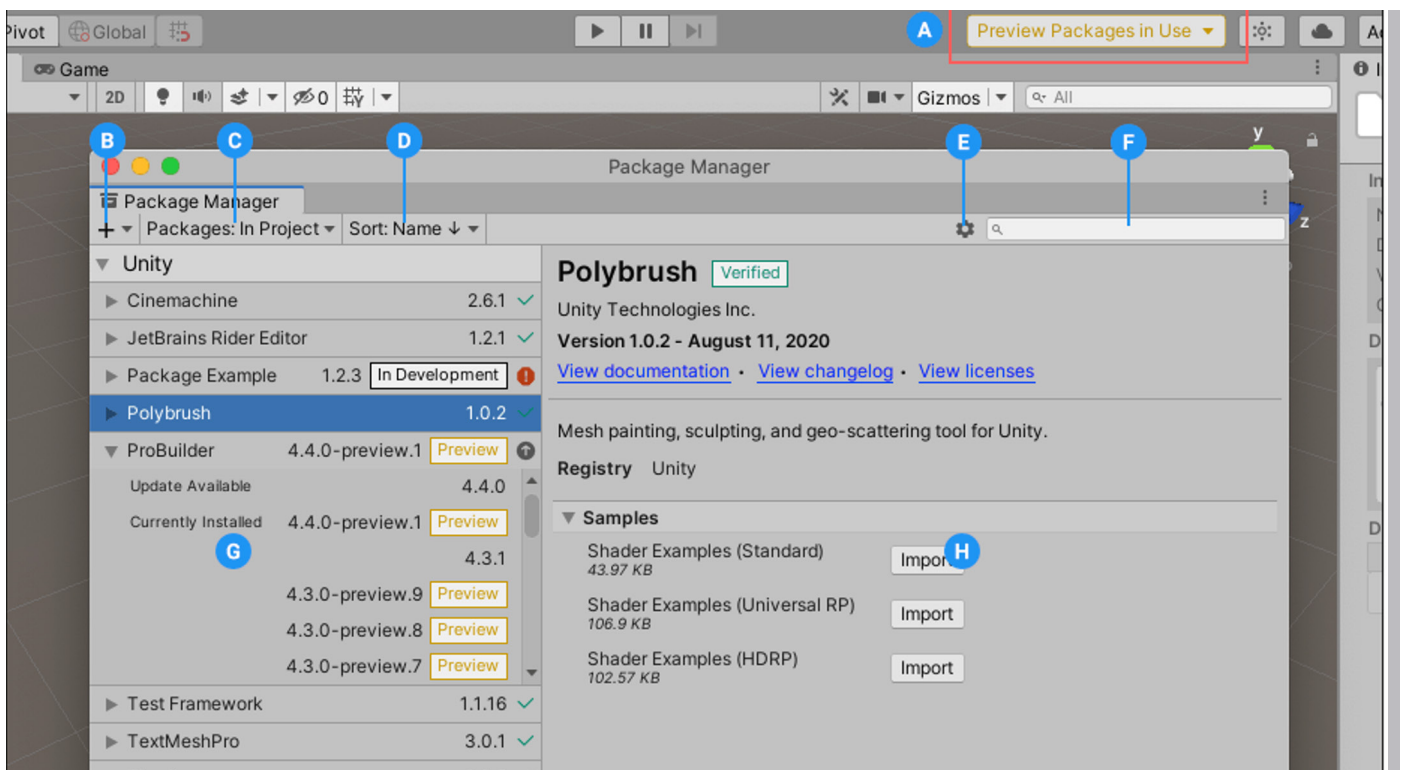
Editor workflows

Unity 2020 LTS includes multiple improvements that speed up Editor workflows, such as keyboard shortcuts to launch frequently used features, handy user interfaces to reduce repetitive tasks, improvements to the debugging workflow, and much more.

Collectively, these improvements can save you hours of work over days and weeks. Your quality of life with Unity improves because you can iterate faster and develop more efficiently. Try out these tips and shortcuts to go faster in Unity.

The Package Manager

The Package Manager has several design updates in 2020 LTS, including new user interface (UI) iconography, improved layout, and better distinctions between information for currently installed packages and for available updates.



The updated Package Manager interface.

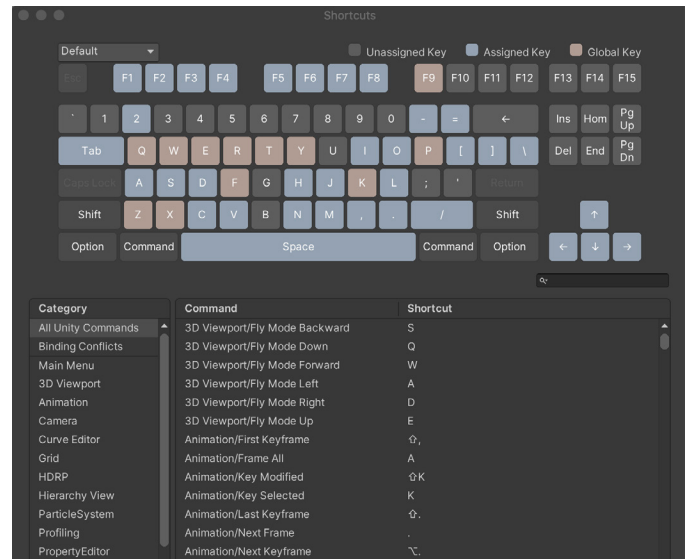
The docs provide an easy overview of the features in the [Package Manager window](#), handy for both new and experienced Unity developers who need to quickly understand the latest updates.

Shortcuts Manager

The [Shortcuts Manager](#) is an interactive visual interface to help you manage Editor hotkeys. Here, you can assign shortcuts to different contexts and visualize existing bindings for any tools that you use frequently.

Bind any key or combination of keys, to a Unity Editor command. For example, the R key is bound by default to the Scale tool in the Tools context.

The Binding Conflicts category also identifies if you have a shortcut assigned to two commands that can be executed at the same time. Use the interface to resolve such conflicts. Note: You *can* assign the same shortcut to multiple commands if they are in different contexts and cannot execute at the same time.



The Shortcuts Manager

Category	Command	Shortcut
All Unity Commands	HDRP/Decal: Handle swap between cropping and stretching UV	⌘W
Binding Conflicts	Main Menu/File/Close	⌘W
Main Menu		
3D Viewport		

Identify Binding Conflicts between shortcuts

Access the Shortcuts Manager from Unity's main menu:

- On Windows and Linux, select **Edit > Shortcuts**.
- On macOS, select **Unity > Shortcuts**.

Use the provided API in the [UnityEditor.ShortcutManagement](#) namespace to define custom shortcuts in your own scripts and packages.

Command	Shortcut
View	Q
Move	W
Rotate	E
Scale	R
Rect	T
Transform	Y
Toggle Pivot Position	Z
Toggle Pivot Orientation	X

Common Editor shortcuts

Common Shortcuts

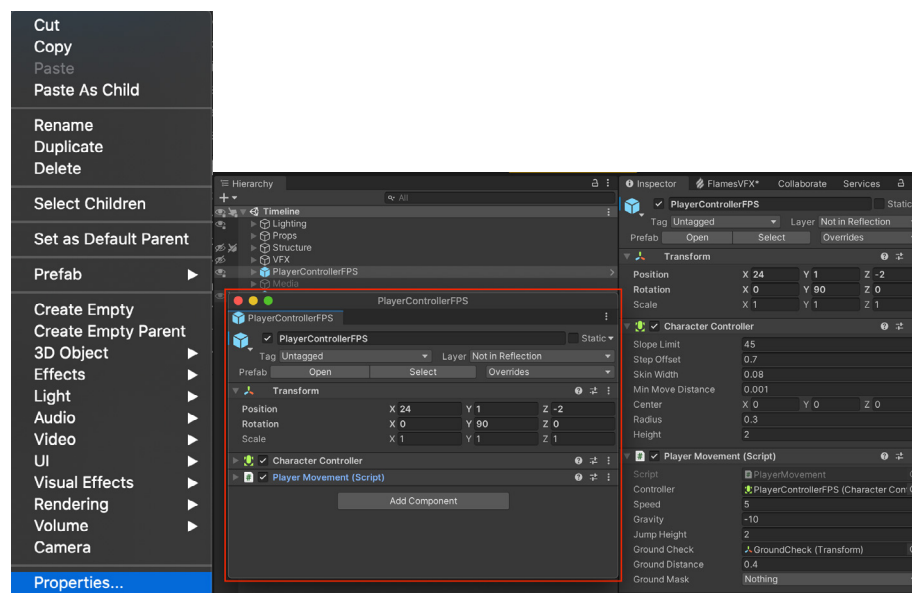
Here are some common default shortcuts:

Action	Windows	Mac
Frame Selected	F	F
Duplicate Items	Ctrl + D	Cmd + D
Delete GameObject	Shift + Del	Cmd + Delete
View/Move/Rotate/Rect/Transform	Q/W/E/R/T	Q/W/E/R/T
Toggle Pivot Mode	Z	Z
Toggle Pivot Rotation	X	X
Vertex Snap	V	V
Snap	Ctrl + LMB	Ctrl + LMB
Toggle Maximize	Shift + spacebar	Shift + spacebar
Edit Prefab in Context	P	P

Focused Inspectors

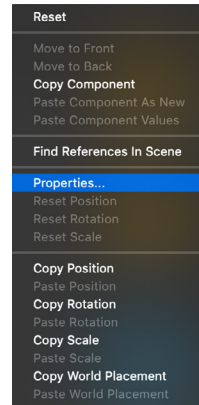
Unity 2020.1 introduced the [Focused Inspector window](#), which allows you to inspect the properties for a specific GameObject, component, or asset. It always displays the properties of the item you opened it for, even if you select something else in the Scene.

Right-click on a GameObject or Component and choose **Properties**. This reveals a floating Inspector window that you can reposition, dock, or resize like any other window.



A Focused Inspector comparing two GameObjects

Opening multiple Focused Inspectors at the same time allows you to reference multiple GameObjects while making changes to the Scene.



You can also focus on a specific Component of a GameObject, requiring less screen space.

Presets

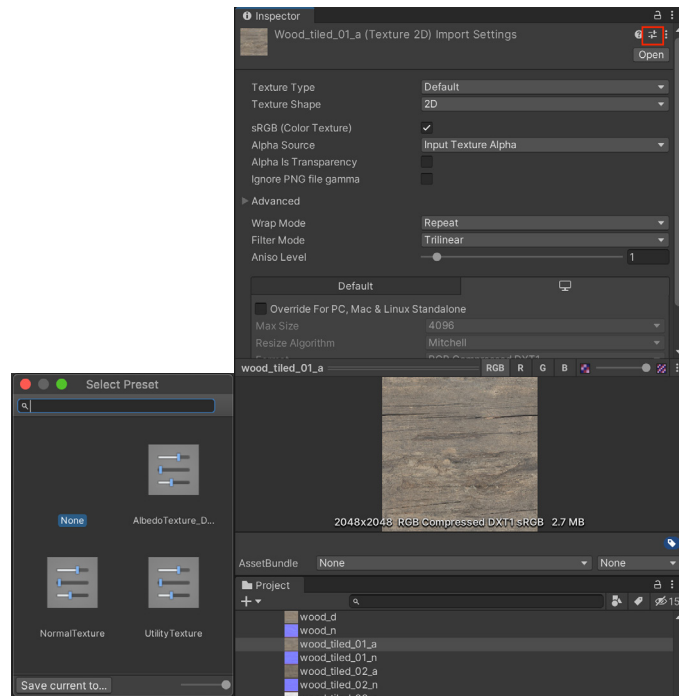
This feature allows you to customize the default state of anything in your Inspector. Creating a [Preset](#) lets you copy the settings of a component or asset, save it as an asset, then apply the same settings to another item later.

Use Presets to enforce standards or to apply reasonable defaults to new assets. This ensures consistent standards across your team, so commonly overlooked settings don't impact your project's performance.

Click the Preset icon to the top right of the component. Click **Save current to...** to save the Preset as an asset. Click one of the available Presets to load a set of values.



The Preset icon is highlighted here in red.



In this example, the Presets contain different Import Settings for 2D textures depending on usage (albedo, normal, or utility).

Other handy ways to use Presets:

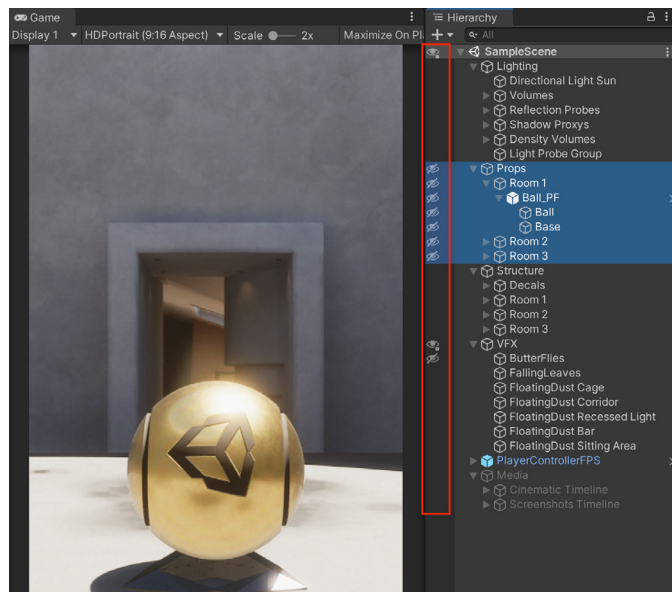
- **Create a GameObject with defaults:** Drag and drop a Preset asset into the Hierarchy in order to create a new GameObject with the corresponding component filled in with Preset values.
- **Associate a specific Type with a Preset:** In the Preset Manager (**Project Settings > Preset Manager**), specify one or more Presets per Type. Creating a new component will then default to the specified Preset values.
 - Pro tip: Create multiple Presets per Type, and rely on the Filter to associate the correct Preset by name.
- **Save and load manager settings:** Use Presets for a Manager window, so the settings can be reused; for example, if you plan to reapply the same Tags and Layers or Physics settings, Presets can reduce set up time for your next project.

SceneVisibility

As your Scene grows larger, you can temporarily hide specific objects so that you can select and edit your GameObjects with more ease.





Instead of deactivating the GameObjects (which can lead to unintended behavior), toggle the SceneVisibility controls. This allows you to hide and show objects in the Scene view, without changing their in-game visibility.

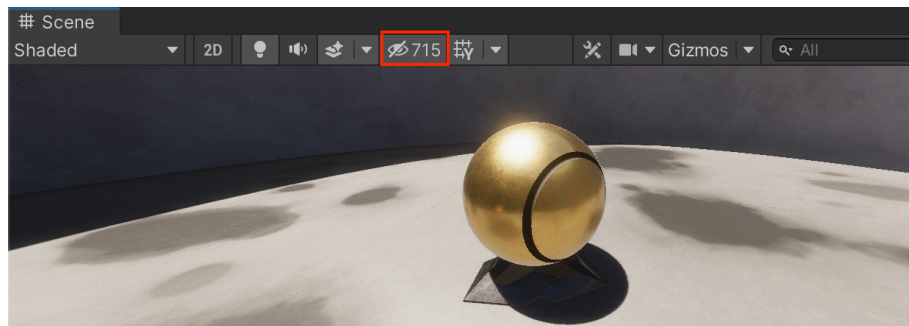
Use the toolbar in the Hierarchy window to enable or disable the SceneVisibility for GameObjects in the viewport.



Hide objects in the Scene view using SceneVisibility controls.

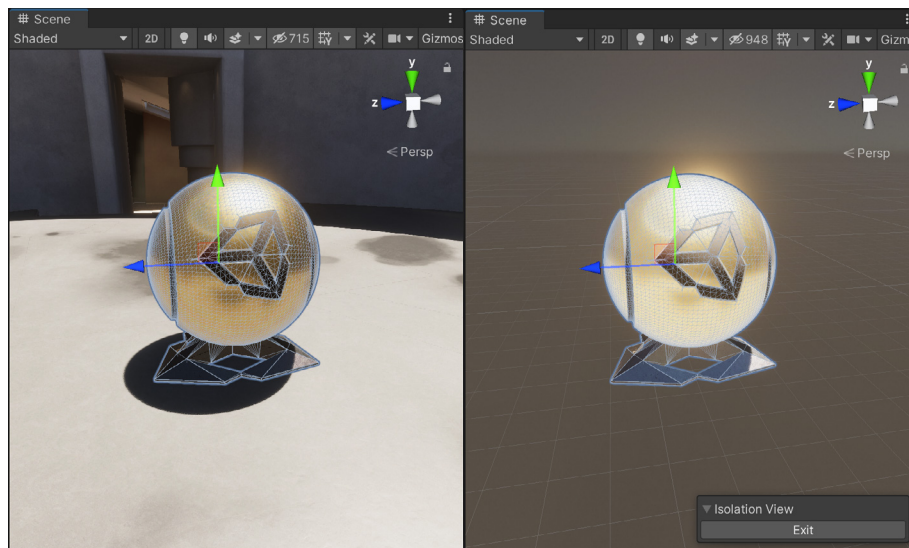
Note that the status icons may change in the Hierarchy, depending on whether parent or child objects become hidden.

Icon	Status
	The GameObject is visible, but some of its children are hidden.
	The GameObject is hidden, but some of its children are visible.
	The GameObject and its children are visible, but they only appear when you hover over the GameObject.
	The GameObject and its children are hidden.



Toggle the Scene view control bar on or off to override the global SceneVisibility.

Use Isolation View to concentrate on a specific object and its children. Select the GameObject in the Hierarchy window and press **Shift + H** to toggle it on and off. This overrides your other SceneVisibility settings until you exit.



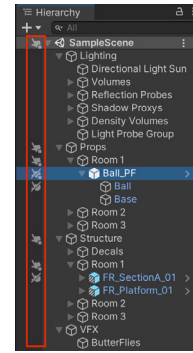
Isolation View allows you to edit a GameObject without distractions.

Remember that you can always use the **Shift + spacebar** shortcut to maximize the viewport and hide the rest of the Editor as well.

Scene picking

You can modify the pickability state of GameObjects, similar to SceneVisibility. Use the toolbar to block specific GameObjects from being selected in the Scene view. This is useful to avoid selecting and editing an unintended GameObject in large scenes.

Because you can toggle pickability for a whole branch or a single object, some GameObjects may be pickable but have children or parents that are not. The following icons differentiate their status.

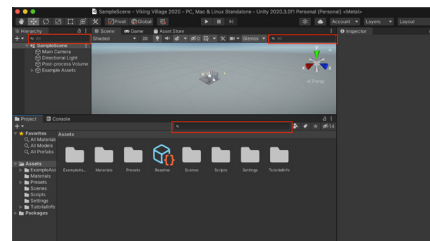


Hierarchy pickability

Icon	Status
	You can pick the GameObject, but you cannot pick some of its children.
	You cannot pick the GameObject, but you can pick some of its children.
	You can pick the GameObject and its children (only appears when you hover over the GameObject).
	You cannot pick the GameObject or its children.

Searching

The Editor contains search functionality for the Scene view, Hierarchy window and Project window.

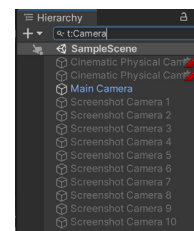


Search options in the Editor are highlighted in red.

In addition to searching for names, you can search by type. Use the dropdown to select **Type** or the **t:** shorthand syntax.

If you use [Asset Labels](#), you can also use the **l:** shorthand to filter for labels.

In this example, we search the scene for all objects of type Camera.

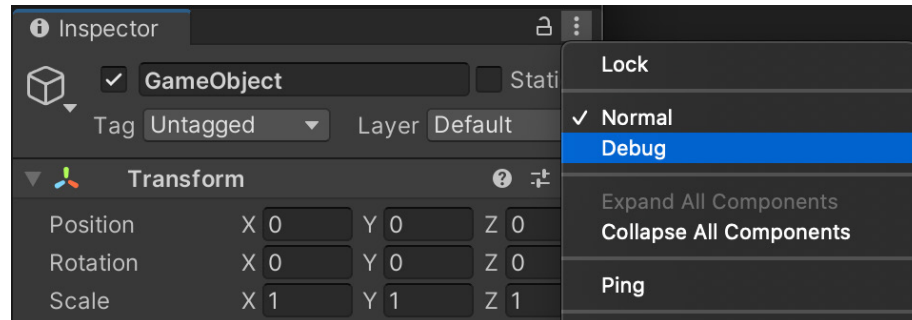


Filtering by Type

Inspector Debug Mode

You can toggle each GameObject's Inspector between Normal and Debug mode. Click the **More Items** (:) button to open the context menu and choose the desired mode.

Debug Mode only shows the selected component's properties and their values. It also displays private variables, although you cannot edit them.

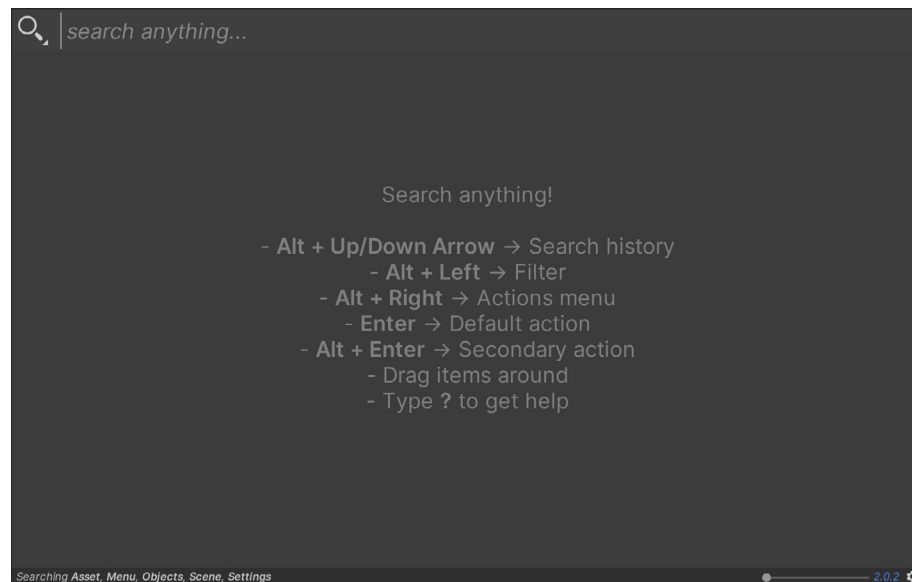


Inspector Debug mode

QuickSearch

If you want to extend your search beyond the windows discussed here, you can find anything in Unity using the [QuickSearch](#) package.

Unity 2021.1 incorporates this functionality into the Editor without requiring a separate package installation. Look for it under **Edit > Search All** (Ctrl + K on Windows / Cmd + K on macOS).

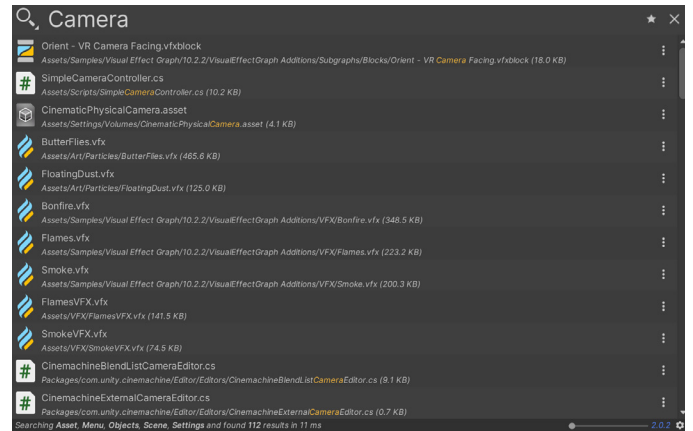


Use the hotkey or Help menu to launch QuickSearch.

Once installed from the PackageManager, activate QuickSearch from either **Help > QuickSearch** or use the **Alt + '** hotkey combination.

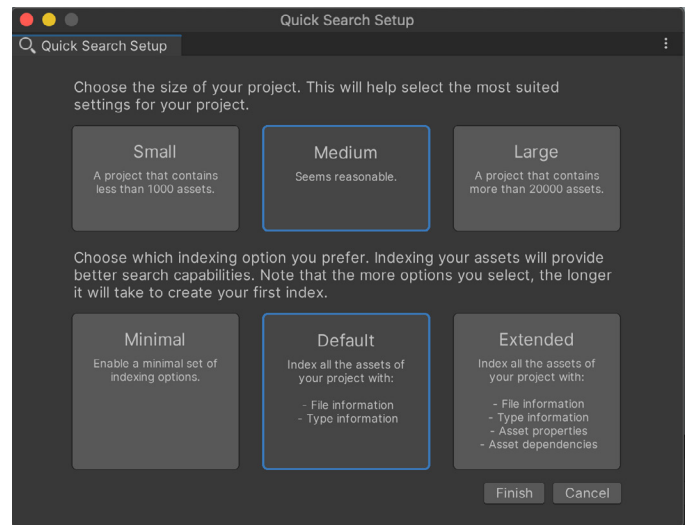
QuickSearch enables you to search multiple areas of Unity: assets, scene objects, menu items, packages, APIs, settings, etc.

The image to the right shows an example of a QuickSearch for “Camera.”



Comprehensive results from QuickSearch.

Make sure you run the setup wizard to configure the search settings for the best results.



Performance varies depending on the size of your project, so choose the best settings for your individual needs.

See the [QuickSearch guide](#) to learn more about searching both inside and outside of Unity.

10 small workflow tips

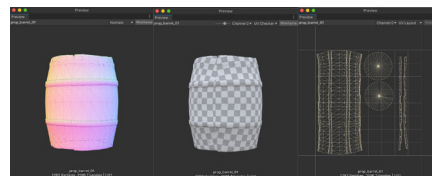
These 10 small but powerful enhancements will help you further speed things up in the Editor.

1. Cut and paste GameObjects in the Hierarchy window. You can also **Paste As Child** from the context menu.



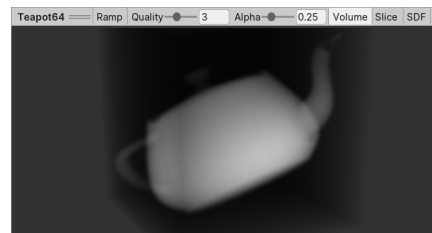
Paste As Child

2. Use the **F** shortcut to frame the selected object in Scene View. This now handles more object types and does a better job of framing them. In Play Mode, press **Shift + F** to lock onto a selected GameObject that is moving.
3. Display UVs, normals, tangents, and other Mesh information in the Inspector preview.



The Inspector preview

4. See improved Inspector previews for 3D textures, such as volumetric render, 3D texture slices, or a signed distance field.



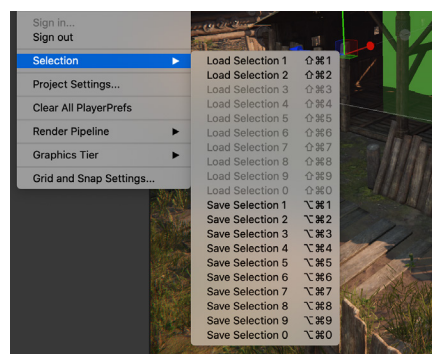
A volumetric render

5. Use the Layers menu to toggle off the visibility of any Layers (such as UI) that may obscure your Scene view. Lock a Layer to avoid changing its state accidentally.



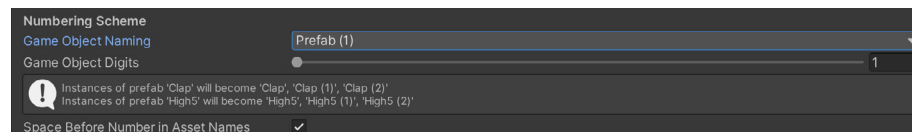
Toggle and edit Layers

6. If you frequently select the same objects in your scene, use the hotkey combos under **Edit > Selection** to quickly save or load a selection set.



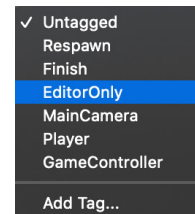
Load and Save Selections

7. Modify the **Numbering Scheme** for duplicate objects in **Project Settings > Editor**. Define options for the naming here as well as the padding and spacing of the instance number.



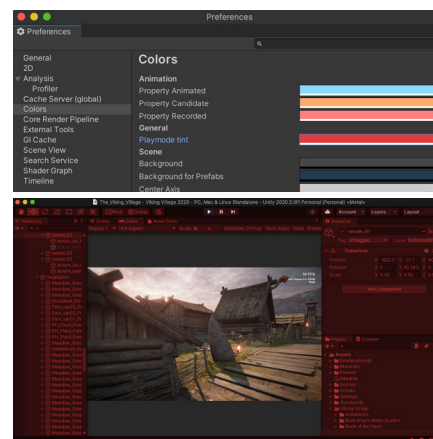
Numbering Scheme

8. Use the **EditorOnly** tag to designate GameObjects that will not appear in a build of the application.



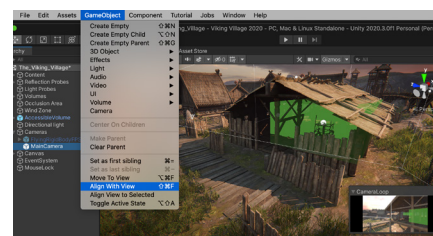
EditorOnly

9. Change colors in the Editor via **Unity > Preferences > Colors** to find certain UI elements or objects more quickly in Editor. Adjust the Playmode tint to remind yourself when Play Mode is active, so you don't lose any changes you intended to save on exit. In the event that you make a change in Play Mode that you want to keep, use the **More Items** (:) button. Copy the component or transform values while playing, then paste from the clipboard upon exiting Play Mode. Alternatively, if you have multiple component changes, drag out a temporary Prefab to save your work there.



Playmode tint

10. When you set up cameras, use **GameObject > Align With View** to line up your Game camera with the Scene camera. Or, if you're matching the other way around, use **Align View to Selected** to align the Scene camera with another camera in the Hierarchy.



Align With View

Artist workflows

This section highlights the improvements made to artist features in Unity 2020 LTS, making them even more friendly and intuitive to use. 2D and 3D artists, technical artists, animators, and level designers will benefit from shorter waiting times and more efficient in-context iteration.



Many of the 2D tips and improvements outlined here are used in [Dragon Crushers](#), a sample project that showcases Unity's native suite of 2D tools and graphics technology. The gameplay is a vertical slice of a side-scrolling idle RPG. Get it on the [Asset Store](#).

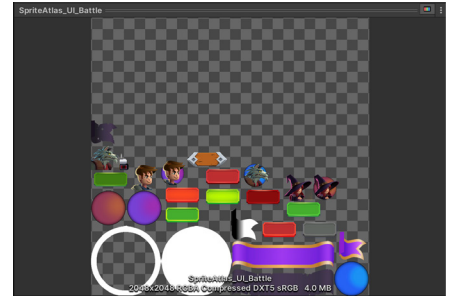


Screenshots from *Dragon Crashers*

Sprite Atlas

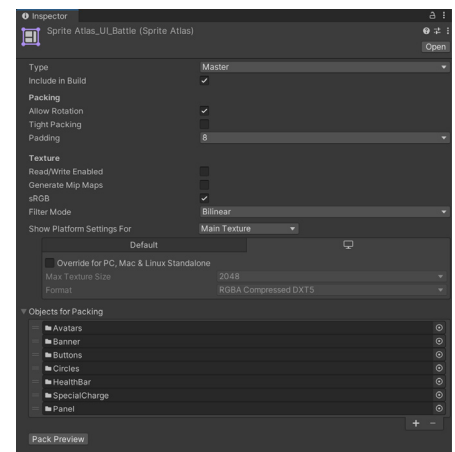
A 2D project uses Sprites to create its visuals. These potentially contain many Texture assets and may thus require many draw calls.

To optimize resources, use a **Sprite Atlas (Asset > Create > Sprite Atlas)** rather than rendering individual Sprites and Textures.



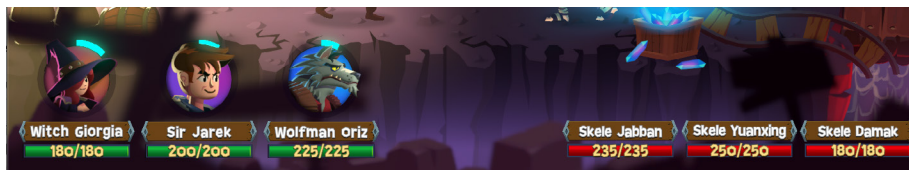
A Sprite Atlas packs several Sprites into a single combined Texture.

The Sprite Atlas (with the file extension **.spriteatlas**) appears in the Assets folder.



Sprite Atlas settings

Once the Textures are consolidated, Unity can issue a single draw call to access the packed Textures with a smaller performance overhead.



In Unity's UI system, atlasing and the GameObject structure matter for batching.

A SpriteAtlas can reduce draw calls if you organize the UI layout correctly. Unity scans the GameObjects' Hierarchy top-to-bottom in order to batch objects that use the same texture and material. See the best practice guide, [Optimizing Unity UI](#) (in particular, Section 4: Fill-rate, Canvases and input) for tips on how to structure your UIs for optimal results.

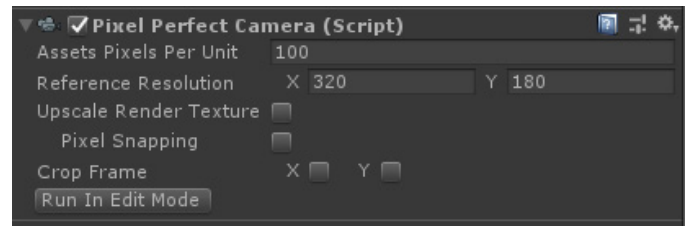
The [SpriteAtlas API](#) provides additional control at runtime. You can also create a [Variant Sprite Atlas](#) or prepare the Sprite Atlases for [an alternate form of distribution](#) with [Late Binding](#) in a script.

10 2D workflow tips

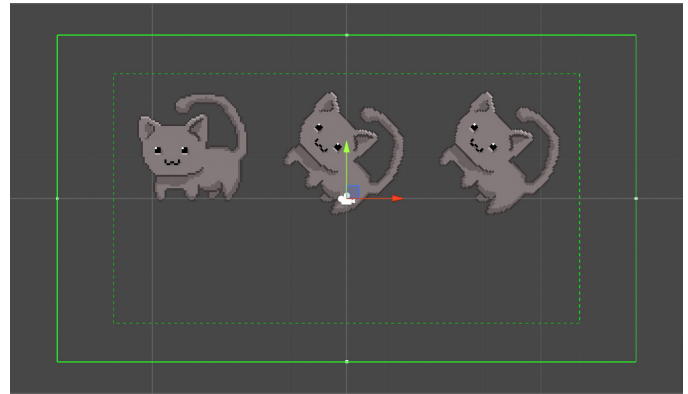
Unity has evolved its native 2D tools to help you develop faster. Save time with these helpful tips.

1. The [2D Pixel Perfect Package](#) contains a Pixel Perfect Camera that ensures your pixel art remains crisp and clear at different resolutions, so you can avoid manual scaling of your art assets.

Read how SouthPAW Games created their first pixel art game using Unity's 2D tools in [2D Pixel Perfect for a crisp conquest in Skul: The Hero Slayer](#).

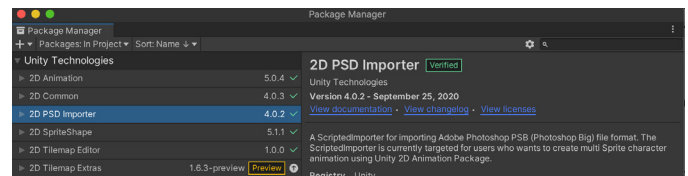


Pixel Perfect Camera



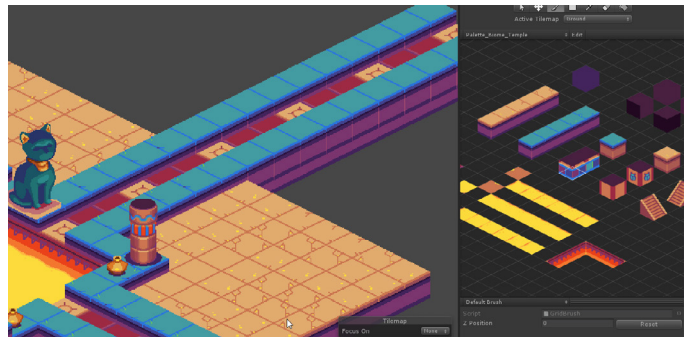
The 2D Pixel Perfect Camera

2. Use the [PSDImporter Package](#) if you want to work with Photoshop file assets. Skip exporting separate Sprites, and import a **.PSB** file (a **.PSB** file supports larger images than the more common **.PSD** file format but is functionally the same). This allows you to import multiple Sprites from the various layers and generate a [Sprite Sheet](#) or [2D Character Rig](#). See [this guide](#) on speeding up your 2D workflows with the PSD Importer.

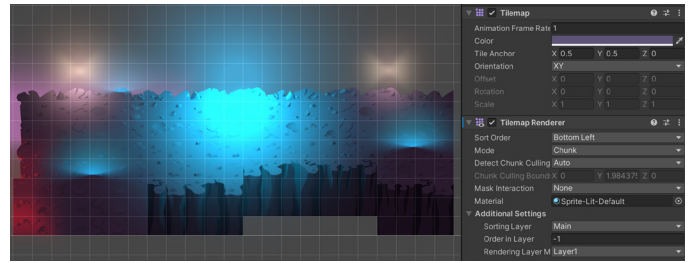


The 2D PSD Importer is available in the Package Manager.

3. Use [Tilemaps](#) to create large grid-based worlds, including hexagonal and isometric versions, optimized for size and performance. Get more tips on how to [optimize performance of 2D games](#) with Tilemaps.

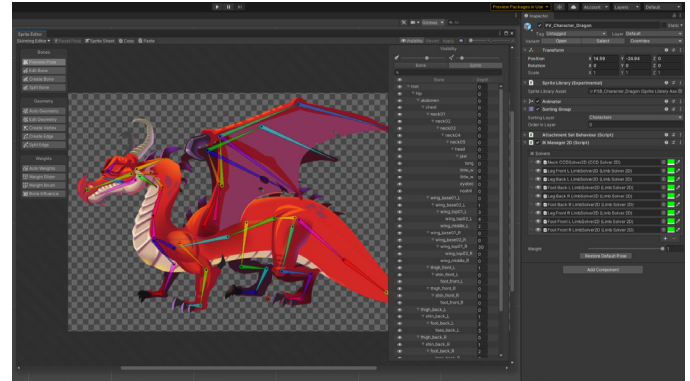


Tilemaps can describe isometric or other grid-like environments.



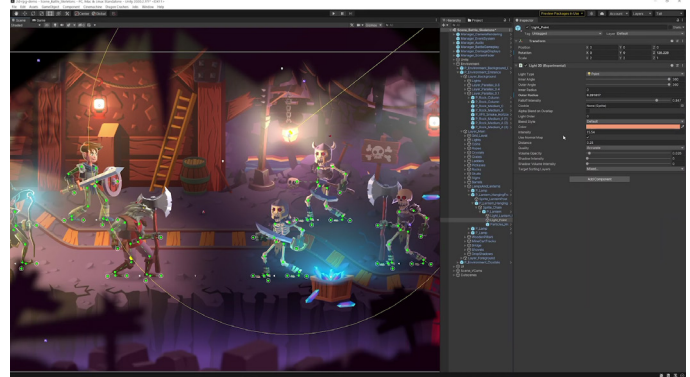
A Tilemap from *Dragon Crashers* sets the background.

4. Create smooth [2D skeletal](#) animation with rigging, tessellation, and bone creation. 2D Inverse Kinematics (IK) simplifies animation, calculating how your 2D bones can reach their target destination.



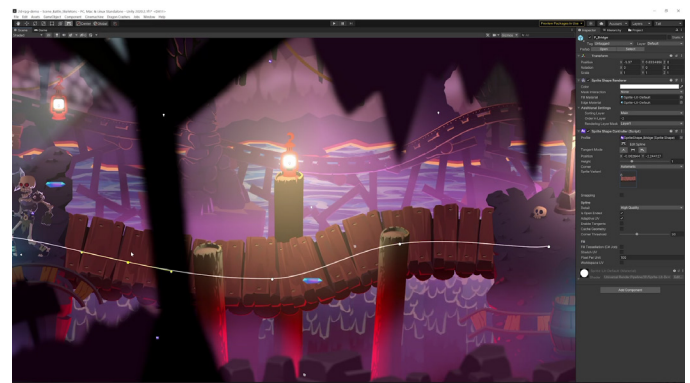
Animating the dragon character from *Dragon Crashers*

5. Enhance your visuals with [2D Lights](#). Lights feature easy-to-configure parameters like light colors, intensity, fall-off, and blending effects. Get additional tips on 2D lighting in [this article](#) by Martin Reinmann of Odd Bug Studio.



The lighting setup in *Dragon Crashers*

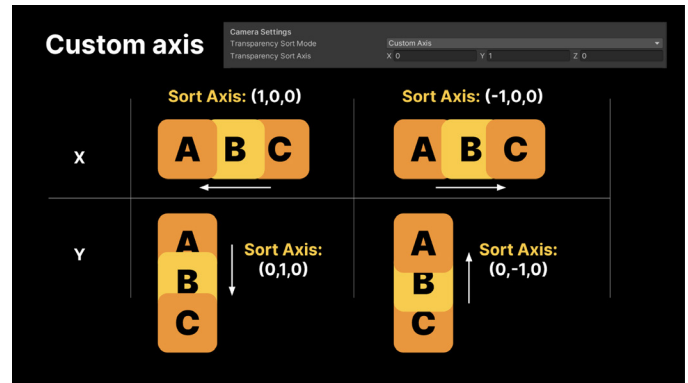
6. [2D Sprite Shape](#) gives you the freedom to create rich free-form 2D environments with a visual and intuitive workflow. It tiles Sprites along a shape's outline, automatically deforming and swapping them based on the outline angle.



One example of how 2D Sprite Shape is used in the latest 2D sample project

7. Sort your [Sprites](#) based on your preferred direction. This can be helpful if you have a number of Sprites within the same layer and sorting order (imagine a card game where the individual cards overlap a bit).

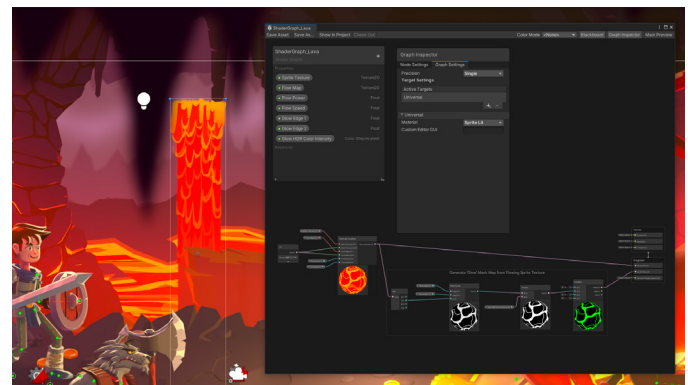
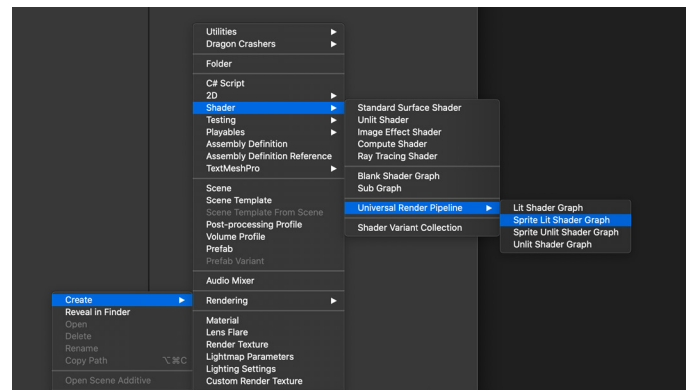
In the built-in render pipeline, look in **Edit > Project Settings > Graphics**. Choose **Custom Axis** for the **Transparency Sort Axis** for the **Transparency Sort Mode**. For example, use (0, 1, 0) for the **Transparency Sort Axis** to sort along the Y axis from top to bottom.



Transparency Sort Mode and Sort Axis

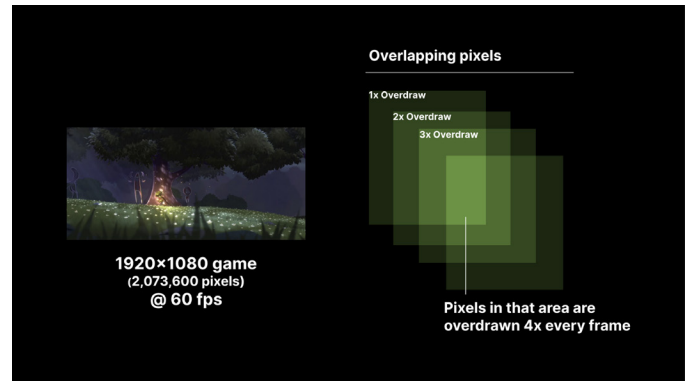
In the Universal Render Pipeline, set the [Camera.transparencySortMode](#) to [TransparencySortMode.CustomAxis](#), then set your axis using [Camera.transparencySortAxis](#).

8. Need custom shaders? Shader Graph includes two MasterNodes designed for 2D: **Sprite Lit** and **Sprite Unlit**. Create [2D shaders](#), and enhance your 2D project visually.



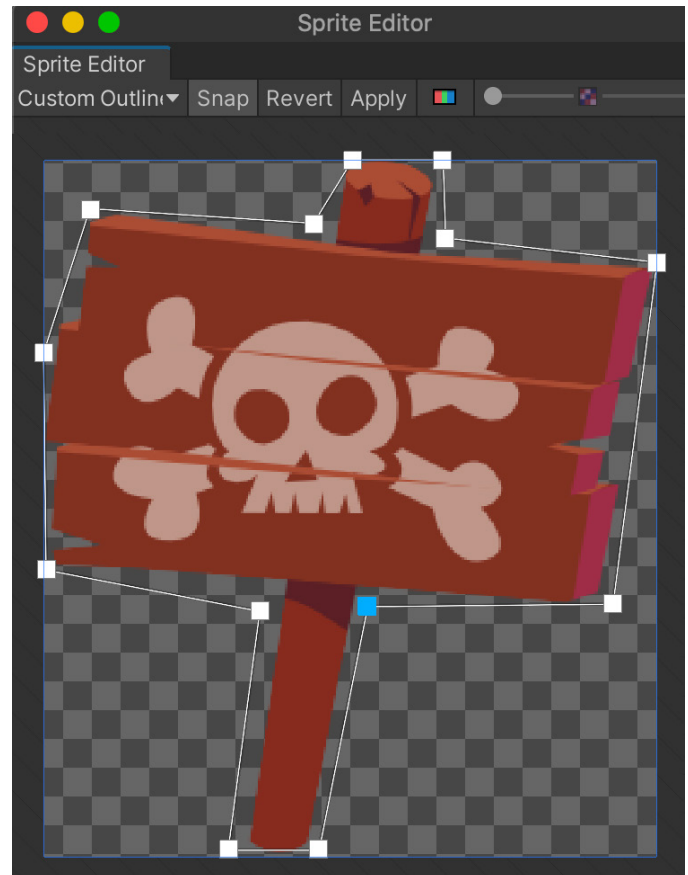
Shader Graph example from *Dragon Crashers*

9. Avoid overdraw to improve performance. Switch the **Mesh Type** to **Tight** in the [Import Settings](#) for each Sprite. Merge overlapping graphics in a single Sprite whenever possible, and try to disable Sprites that could be in a background layer with no use in the game. This reduces the overdraw area and potential overlap with neighboring Sprites.



Reduce overdraw between Sprites

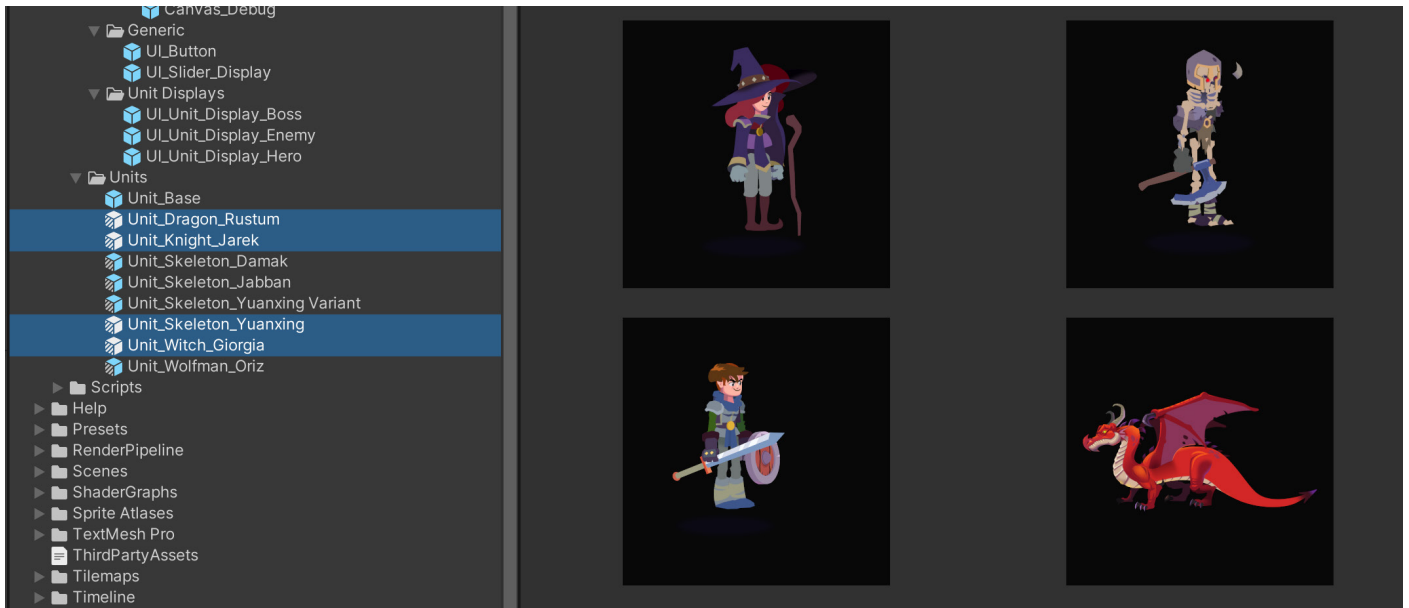
10. You can also consider defining a [custom outline](#) around each Sprite using the [2D Sprite Editor](#) to minimize the unused areas.



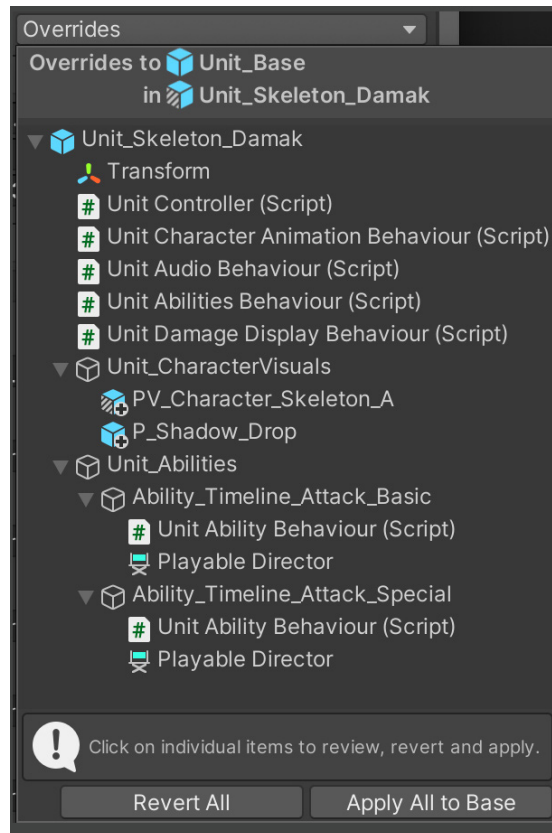
The Sprite Editor with a custom outline

Prefab workflows

Prefabs allow fully configured GameObjects to be saved in the project for reuse. The current workflow with Prefabs lets you build your scenes flexibly and efficiently.

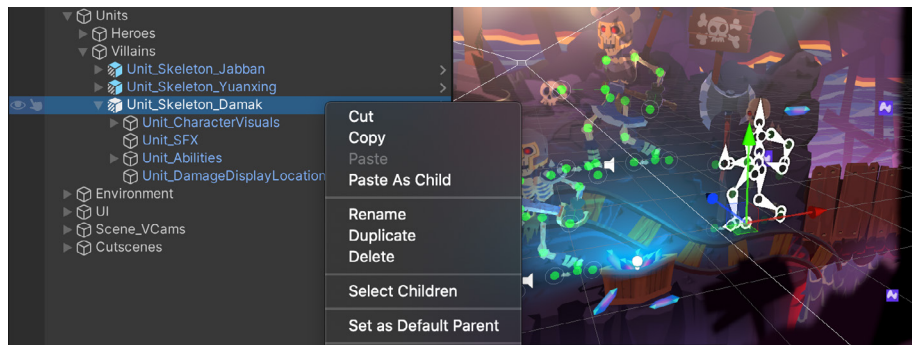


In *Dragon Crashers*, each unit overrides the base unit.

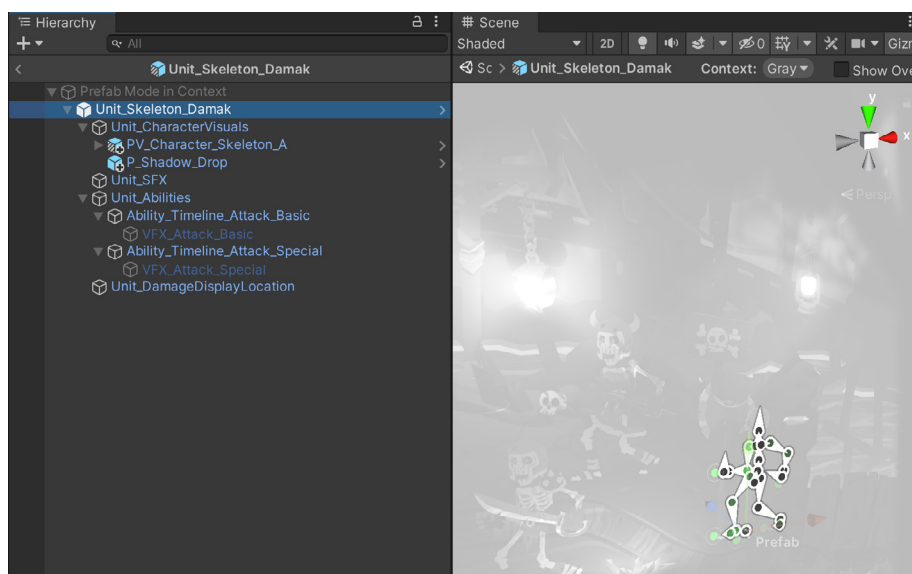


The Overrides dropdown shows how the Prefab differs from the original.

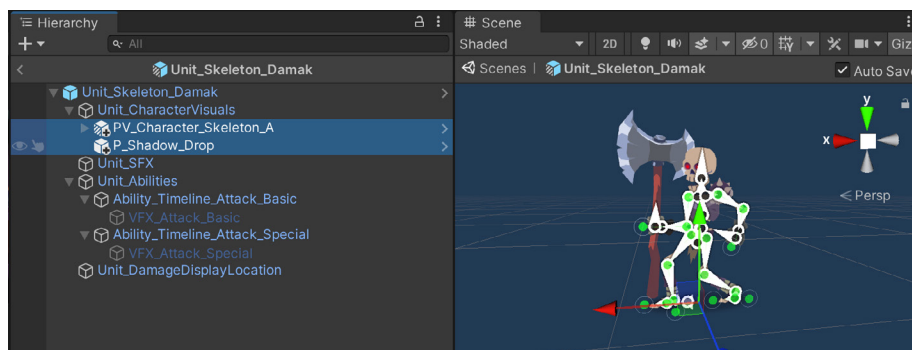
Create your Prefab as an Asset in the Project, then edit it in isolation in **Prefab mode**. Working with the Prefab by itself helps prevent applying unintended overrides. Make your changes with confidence with the background grayed out.



Edit each Prefab, either in Context or in Isolation.



Edit in Context mode to see the Prefab relative to the other objects in the Scene.

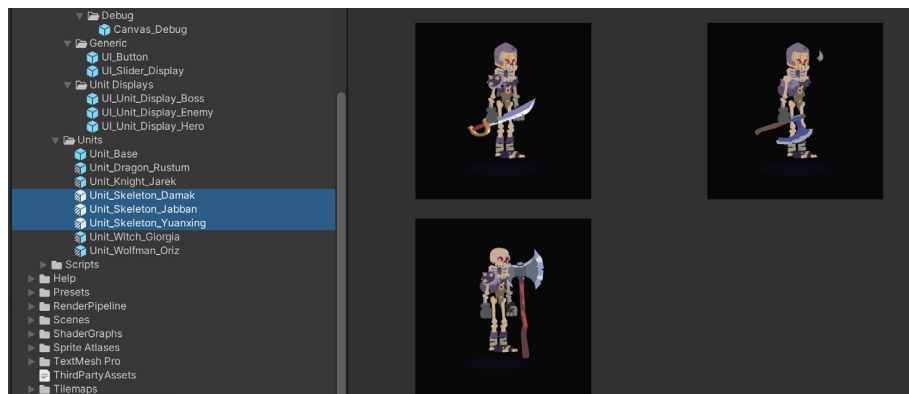


Isolate the Prefab in Prefab mode to avoid unintended overrides. Note how smaller prefabs make up this Nested Prefab.

Nested Prefabs allow you to parent Prefabs to one another. You can now create a larger Prefab, such as a building, composed of smaller Prefabs for the rooms and furniture. This makes it efficient to split development of your assets over a team of multiple artists and developers, who can all work on different parts of the content simultaneously.

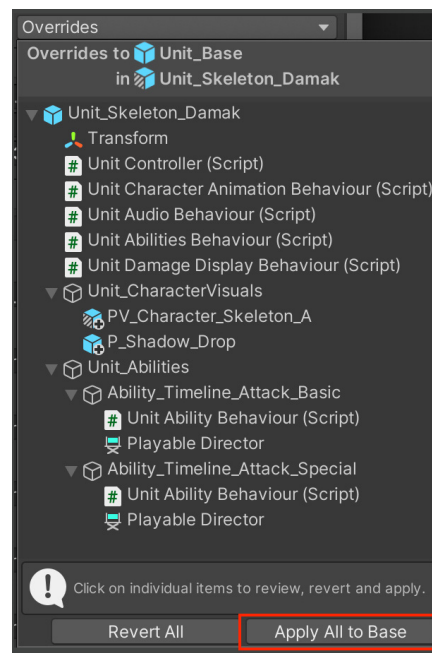
A **Prefab Variant** allows you to derive a Prefab from other Prefabs, much like inheritance in object-oriented programming. To change the Variant, just override certain parts without worry of impacting the original. You can also remove all modifications and revert to the base Prefab at any time.

Alternatively, if you want to change all of your Variants at once, apply changes directly onto the base Prefab itself.



In *Dragon Crashers*, these Prefab Variants have different weapons and abilities.

See [Prefab Workflow Improvements](#) for more information about working with Prefabs in Unity.

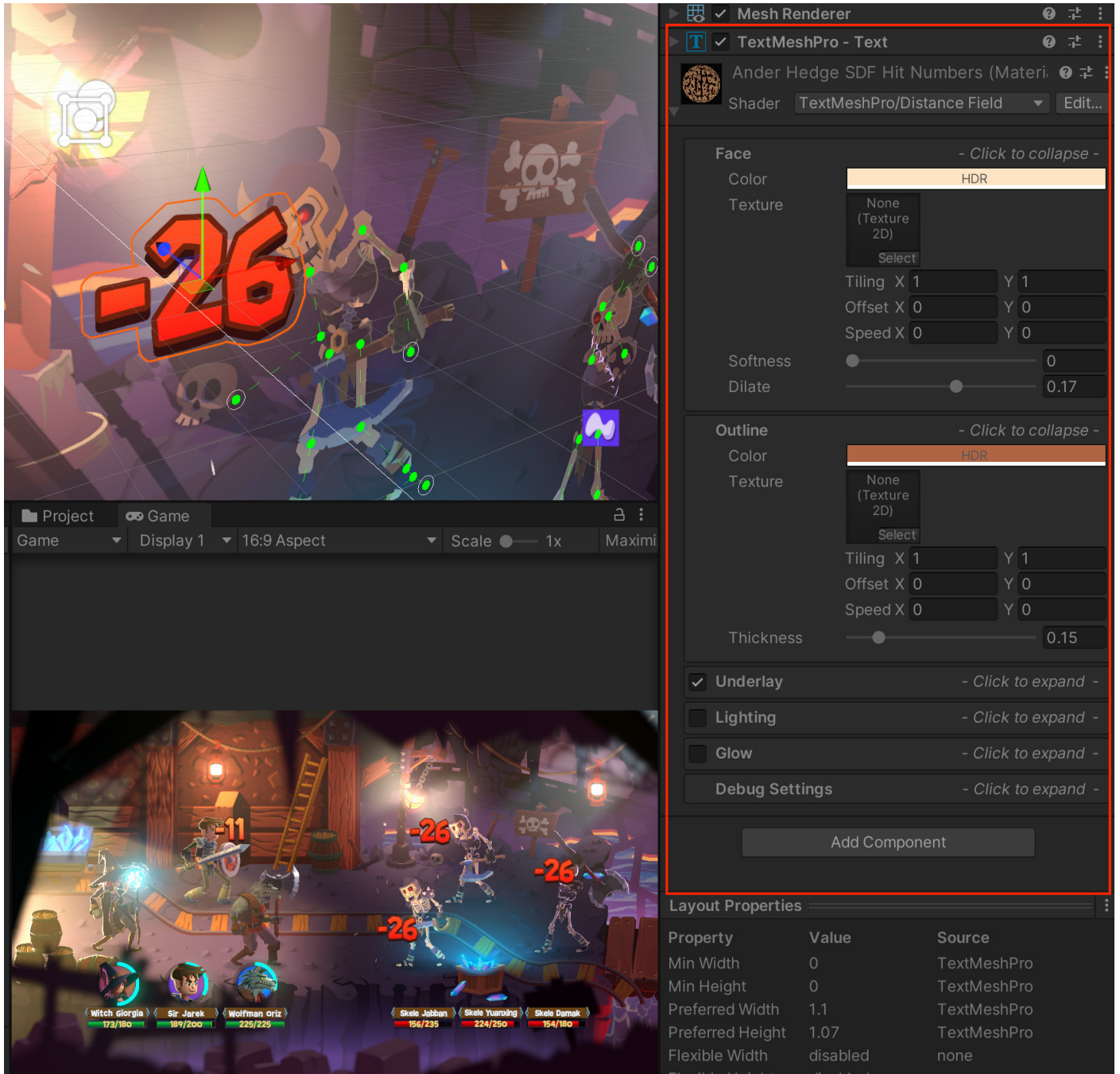


Use **ApplyAll to Base** to propagate changes to the Base object or **Revert All** to undo the overrides.

TextMeshPro

TextMeshPro replaces Unity's UI Text and the legacy Text Mesh. Installed via the PackageManager, [TextMeshPro](#) uses custom shaders and advanced text rendering techniques to deliver flexible text styling and texturing.

Use [TextMeshPro](#) to get access to features like character, word, line, and paragraph spacing, kerning, justified text, links, over 30 rich text tags available, support for Multi Font and sprites, custom styles, and more.

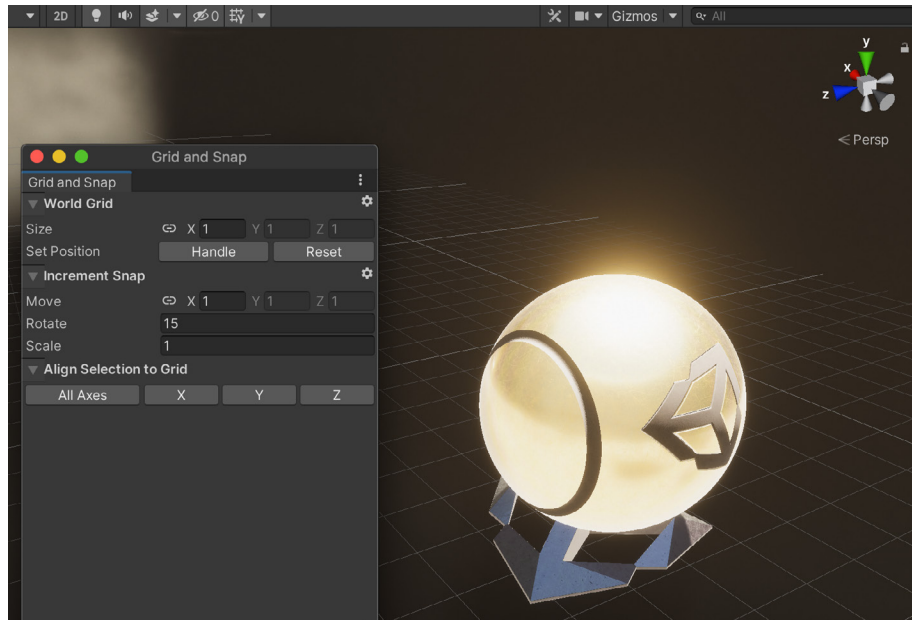


TextMeshPro example from *Dragon Crashers*

Snapping

Working on a grid helps you fit your Prefabs together with less guesswork and greater consistency. Design your level so the pieces connect at scale, making it easier to rearrange and reassemble them.

If you are constructing your Scenes from modular assets, use grid planes to align your GameObjects with each other. Rather than manually typing in round numbers into the Inspector, consider letting the grid snapping tools set your Transforms more quickly and precisely.



The Grid and Snap settings

Unity provides three types of snapping to help you assemble your scenes quickly:

- **World grid snapping:** Make sure the Move tool has the handle orientation set to Global. Hold **Ctrl (Windows)** or **Cmd (macOS)** to snap an object to the world grid increments set in **Edit > Grid and Snap Settings**.
- **Surface snapping:** Hold **Shift** and **Ctrl (Windows)** / **Cmd (macOS)** to snap an object to the intersection of any Collider.
- **Vertex snapping:** Hold down the **V** key while the Move tool is active. This moves the current GameObject to the vertex position of another mesh. Before moving, hover the mouse over one vertex of the active GameObject to make that vertex act as a pivot. **Shift-V** toggles **Vertex snapping** mode on/off.

Combine Vertex and Surface snapping for quick placement:

- Move the GameObject using **Vertex snapping** with the **V** key or **Shift-V**. Hover the cursor over a vertex as a pivot. Snap to another vertex as usual.
- Hold down the **Shift** and **Ctrl (Windows) / Cmd (macOS)** key combo to drag along the surface of the target Mesh.
- Release the mouse button and **V** key once the object is at the desired location.

Open the **Grid and Snap window** from either **Edit > Grid and Snap Settings** or from the grid visibility drop-down menu.

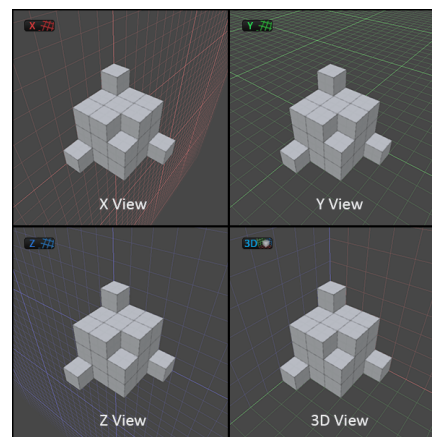


Grid and Snap settings

These grid snapping shortcuts are created by default.

Action	Default Shortcut
Increase Grid Size	Ctrl +] (Windows) or Cmd +] (macOS)
Decrease Grid Size	Ctrl + [(Windows) or Cmd + [(macOS)
Nudge Grid Backward	Shift + [
Nudge Grid Forward	Shift +]
Align Selection to Grid	Ctrl + \ (Windows) or Cmd + \ (macOS)

Need even more control? Consider using the [ProGrids](#) package for even finer control of your snapping and grid planes.

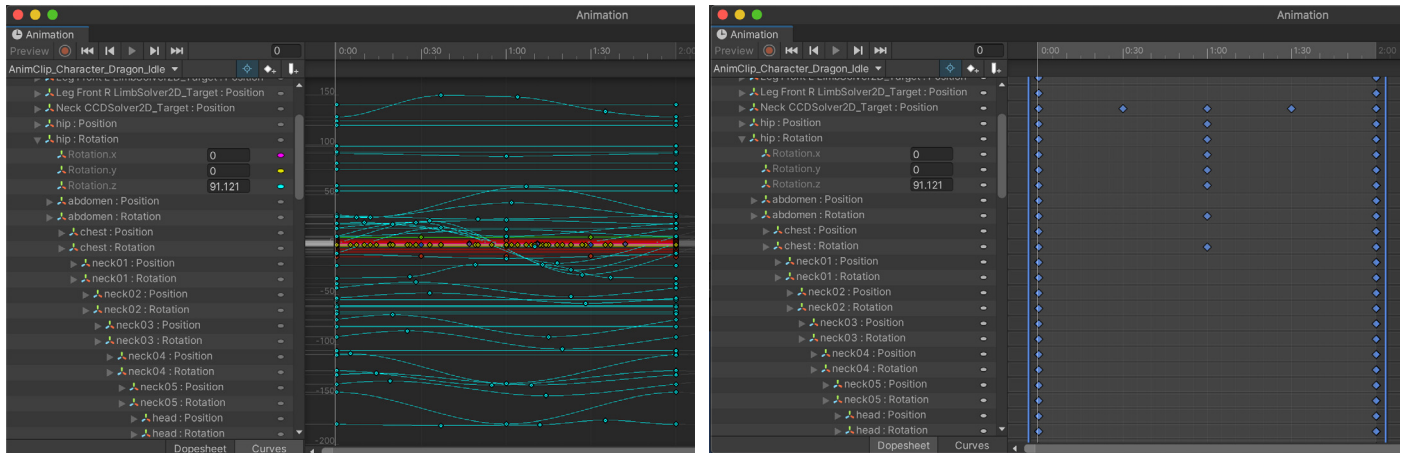


ProGrids

Animation workflow

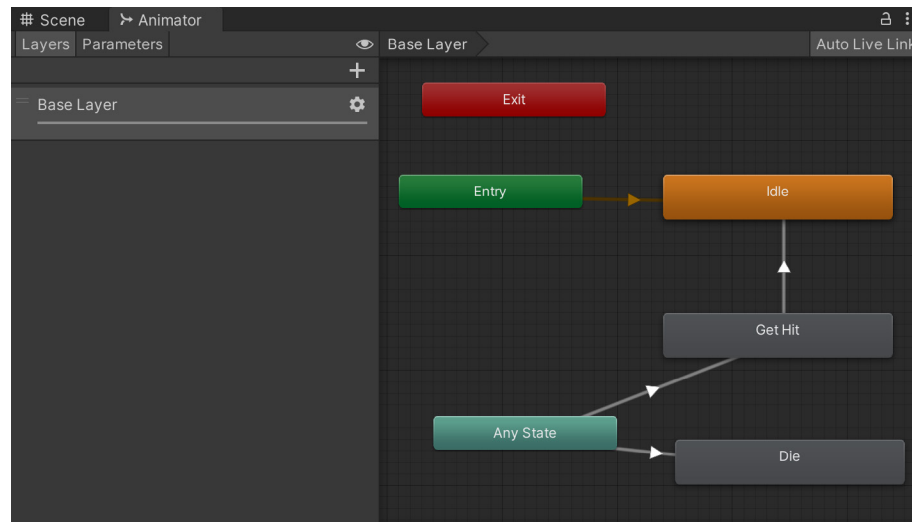
You can animate just about any property in Unity without writing a single line of code using the **Animation Window (Window > Animation > Animation)**. In addition to modifying movement, you can even drive parameters defined in your custom scripts.

Create Animation Clips here, or work in a third-party DCC package of your choice (Autodesk® Maya®, Blender, etc). Think of each clip as an individual unit of motion.



The Animation window can represent the same animation data as curves or a dopesheet.

Edit the AnimationClip Asset within the window in either **Dopesheet** or **Curve** mode. Use **K** or **C** shortcuts, respectively, to toggle between the two. Use standard shortcuts to frame all keyframes (**A**) or frame selected keyframes (**F**).

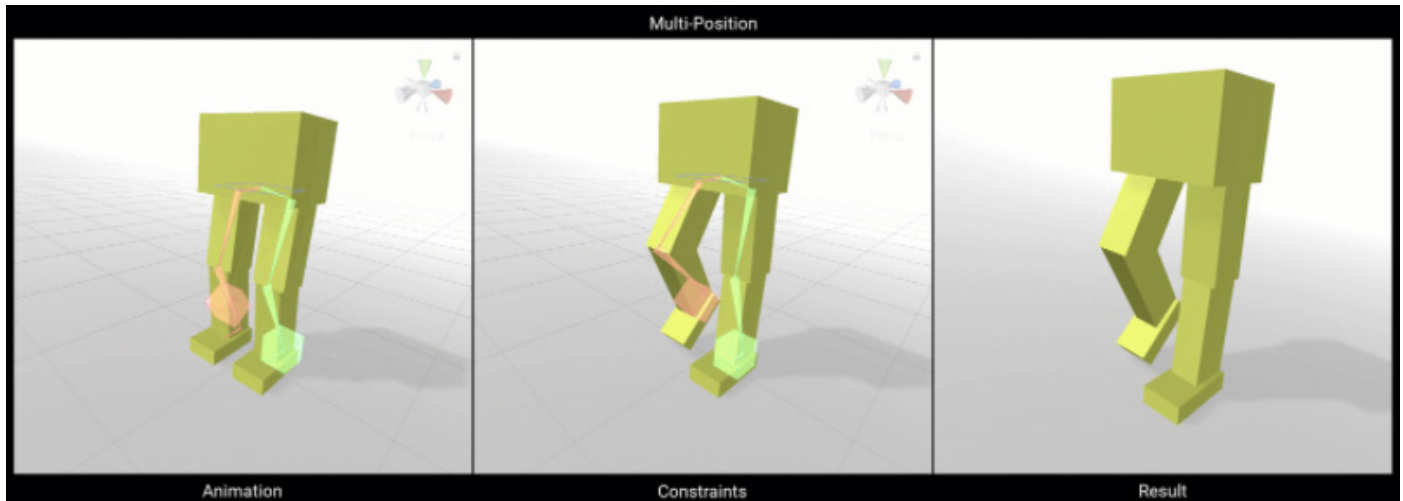


The AnimatorController links the Animation Clips in a visual graph.

Once you have several Animation Clips for your GameObject, the AnimatorController acts as a state machine to create a flowchart-like graph between them.

This allows artists to produce sophisticated animation with greater independence from programmers. If you're using a 2D or 3D rig, you can animate its body parts with different logic. Take advantage of the layering and masking features for greater control. Prototype your motions in a visual programming tool to fine-tune any transitions or interactions between your clips.

Extend this further using the [Animation Rigging](#) package. This package provides a library of rig and inverse kinematic constraints that can create procedural motion. Animated skeletons can thus interact with the environment with "runtime rigging," or physics-based constraints can add dynamic secondary motion.



Constraints can modify your animation at runtime.

Optimization tip

While AnimatorControllers offer convenience, be aware of a few caveats:

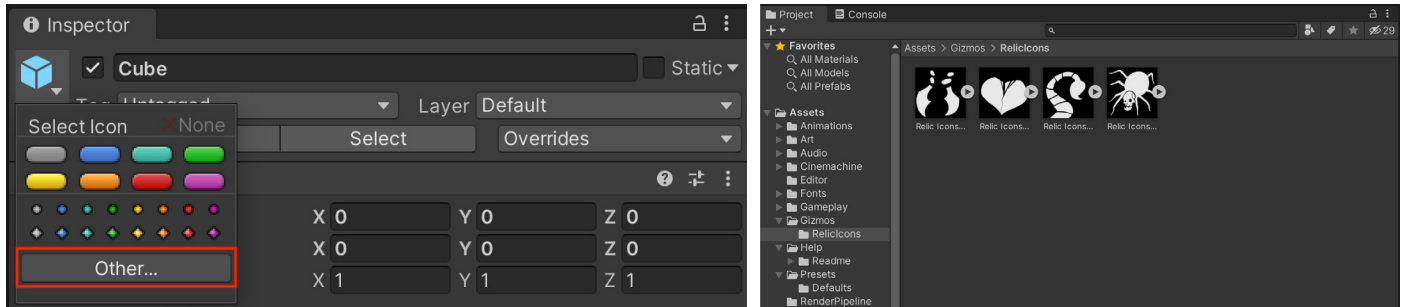
- [Avoid overusing Animators](#), particularly in conjunction with UI elements. Animators cause the UI Canvas to rebuild each frame, even if no animation is playing. Whenever possible, use the legacy Animation components for UI or simple animations. Also, consider creating [tweening](#) functions or using a third-party library (e.g., [DOTween](#)).
- By default, Unity [imports animated models](#) with the generic rig, but developers often switch to the humanoid rig when animating a character. A humanoid rig calculates inverse kinematics and animation retargeting each frame, even when not in use. If you don't need these specific features, save on CPU time and use the generic rig.

Refer to the manual pages about [AnimationClips](#) and [AnimationController](#) for more information about their usage. Read [Unity's evolving best practices](#) for more about optimizing your animation components.

Custom gizmos and icons

Gizmos are small overlay graphics associated with your GameObjects. Use them to navigate the viewport or locate specific objects.

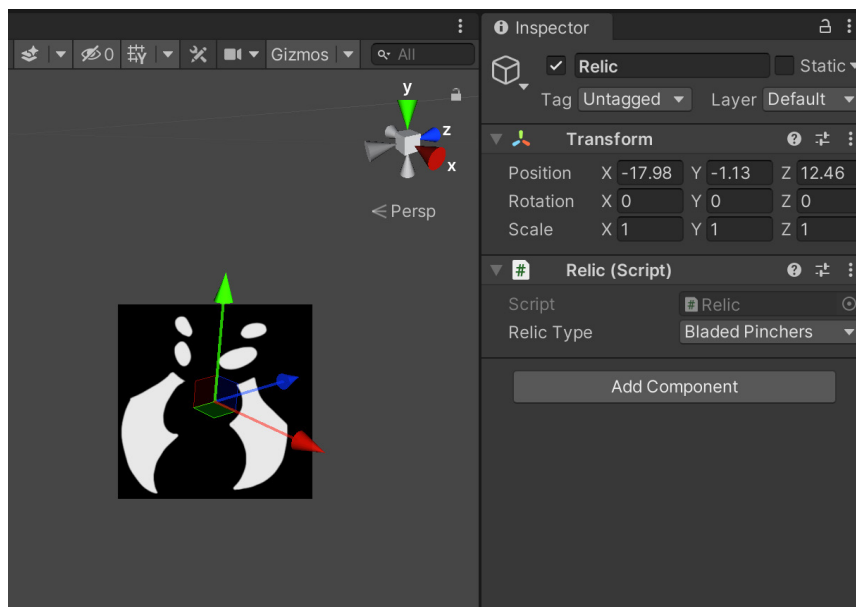
Modify the icons for a GameObject using the **Select Icon** menu. Choose **Other** to define your own icon.



Use the drop-down in the Inspector to switch gizmos.

Select a custom gizmo using the **Other...** option.

You can also create gizmos with scripts and make them interactive. For example, a gizmo could help you define a volume or area of influence for a custom component.



In this example, a script changes the gizmo based on a selection.

Use the **Gizmos** dialogue in the Scene control bar to toggle specific gizmos or globally enable/disable all of them.

See [Creating Custom Gizmos for Development](#) for usage examples. Also, review the APIs for [Gizmos](#) and [Handles](#).

Progressive Lightmapper

Lightmapping allows you to precalculate both direct and indirect lighting, then store the result in a Texture called a lightmap for later use. Unity offers a number of Global Illumination (GI) techniques to produce high-quality lighting and shadows. Though lightmapped geometry is performant at runtime, baking a lightmap has historically been expensive.



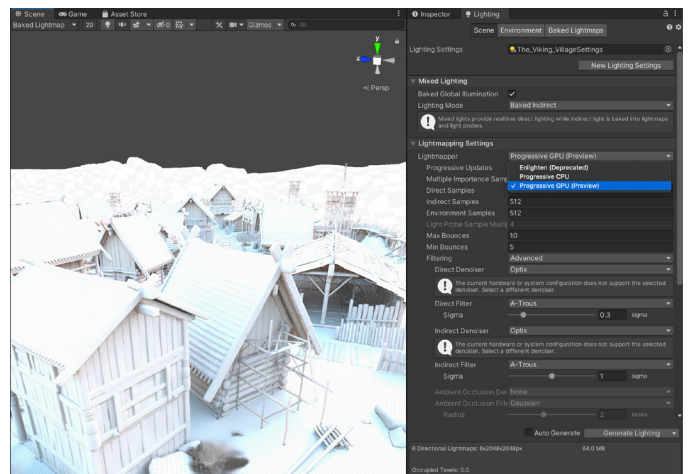
A scene with lightmaps applied.



The same scene without lightmapping.

The [Progressive Lightmapper](#) is a fast path tracer that produces a result quickly, then refines the render over time. You can thus interrupt the process to make changes without waiting for the final bake to complete, allowing you to iterate more rapidly. Here are some tips to speed up your lightmapping:

- Enable **Prioritize View** so the Progressive Lightmapper works on texels currently visible in the Scene view first before changing anything outside of view.
- Reduce unnecessary **Samples** (Direct and Indirect Samples) and **Bounces** (two is usually sufficient; only increase if necessary).
- Optimize the **Lightmap Resolution** and texel count for your lighting needs. The number of texels represents how much work your lightmapper needs to do. Because lightmaps are 2D textures, doubling the lightmap resolution quadruples the amount of work.
- Reduce texels on hidden surfaces, small or thin objects, or anything where lightmapping won't make much impact. Each MeshRenderer contributing to Global Illumination has a **Scale in Lightmap** option to reduce its relative UV size in the lightmap.
- Choose the proper **Lighting Mode**: [Baked Indirect](#), [Subtractive](#), [ShadowMask](#). You don't need to bake shadows if it's not required for your art direction.
- The current Progressive CPU Lightmapper uses your machine's CPU and RAM. The newer **Progressive GPU Lightmapper** (in Preview) uses your GPU and VRAM, potentially speeding up the bake considerably. If your computer meets the [hardware and software requirements](#), this can dramatically accelerate up your lighting workflow (tenfold in some cases).

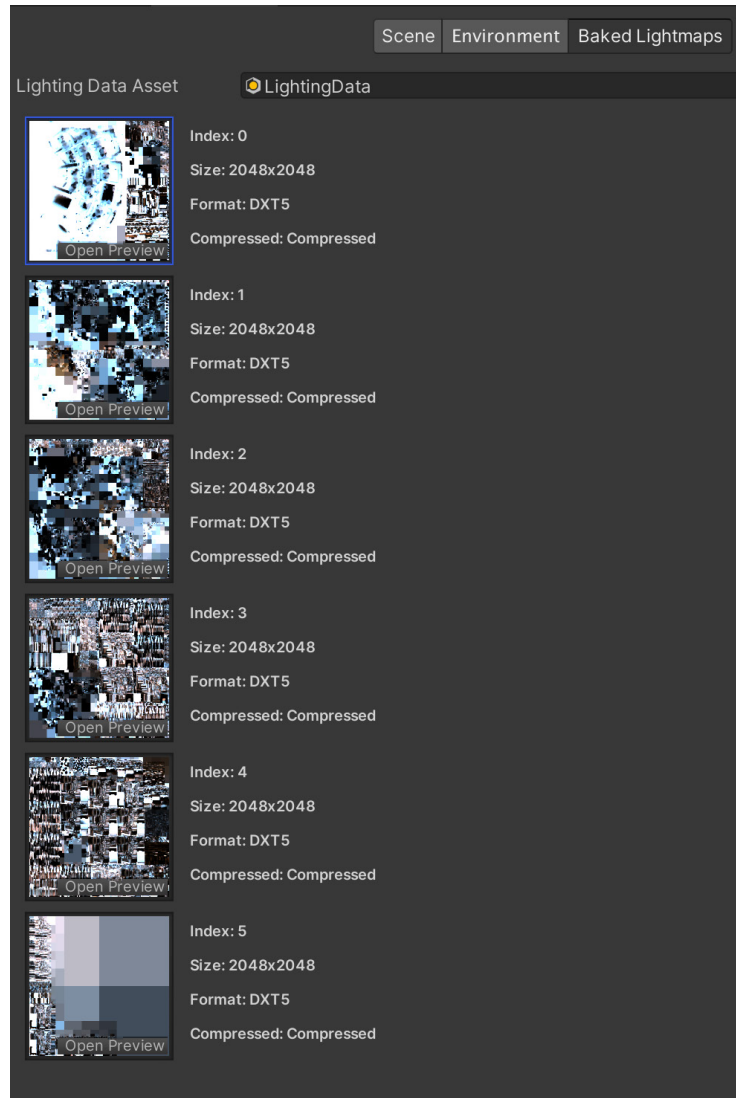


Preview of a baked Lightmap

Light Probes

Global Illumination produces beautiful indirect lighting, but this can be expensive to calculate and store on disk. If you have set dressing or other static meshes that don't absolutely require lightmapping, consider removing them from your lightmap bakes and use [Light Probes](#) instead.

In this example, Light Probes could approximate both direct and bounced lighting for the smaller objects, reserving the higher-quality lightmapping where it's more noticeable.



Lightmaps applied to the *Viking Village* project

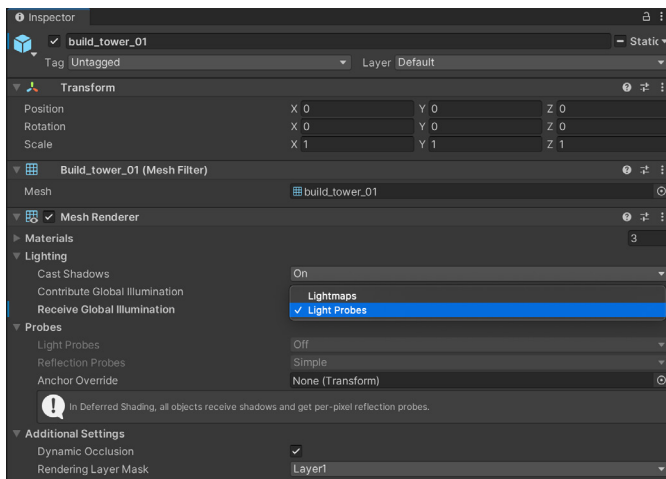


Use Light Probes for smaller details where lightmapping is less noticeable.

Formerly reserved for dynamic objects, Light Probes can apply to static meshes as well. In the MeshRenderer component, locate the **Receive Global Illumination** dropdown and toggle it from **Lightmaps** to **Light Probes**.

Light Probe illumination does not require proper UVs, saving you the extra step of unwrapping your meshes. The Spherical Harmonics basis functions used in probe lighting make it fast to calculate relative to lightmapping.

Arrange Light Probes and Light Probe Groups spatially in the scene. Probe lighting typically bakes faster than lightmapping.



Selecting Light Probes



A Light Probe Group with Light Probes spread across the level

See [Static Lighting with Light Probes](#) for information about selectively lighting scene objects with [Light Probes](#).

For more about lighting workflows in Unity, read [Making believable visuals in Unity](#).

Developer workflows

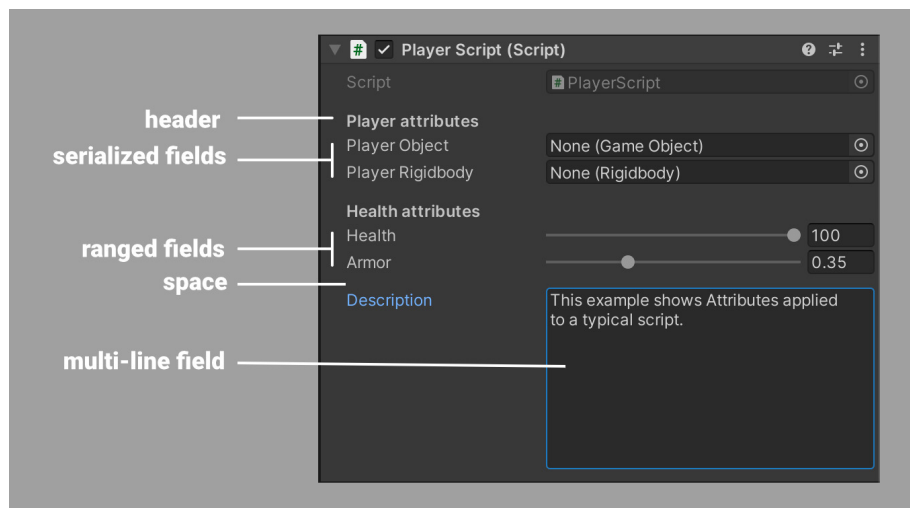
There are always small but helpful shortcuts and tips that even long-time Unity developers can benefit from. Whether it's a small Property Attribute attached to a script variable or a handy but often-overlooked Editor setting, we're sure that you'll find plenty here to speed up your workflows.

Attributes

Unity has a variety of Attributes that can be placed above a class, property, or function to indicate special behavior. C# contains attribute names within square brackets. These are some common Attributes you can add to your scripts.

Attribute	Description	Example
SerializeField	This forces Unity to serialize a private field and makes it visible in the Inspector. Note: You may experience a 0649 warning when applying [SerializeField] to a private variable. To prevent this, simply initialize the variable upon declaration. For convenience, use the default keyword.	<code>[SerializeField]</code> <code>private GameObject myObject = default;</code>
Range	This attribute takes a float or int variable restricted to a specific range. The field appears as a slider in the Inspector.	<code>[Range(1,6)]</code> <code>public int integerRange;</code> <code>[Range(0.2f, 0.8f)]</code> <code>public float floatRange;</code>
HideInInspector	This makes a variable not appear in the Inspector but be serialized.	<code>[HideInInspector]</code> <code>public int p = 5;</code>
RequireComponent	This automatically adds required components as dependencies to avoid setup errors. Note: This attribute only checks the moment that the Component is added to a GameObject.	<code>// PlayerScript requires the GameObject to have a Rigidbody</code> <code>[RequireComponent(typeof(Rigidbody))]</code> <code>public class PlayerScript: MonoBehaviour</code> <code>{</code> <code> private Rigidbody rBody;</code> <code> void Start()</code> <code> {</code> <code> rBody = GetComponent<Rigidbody>();</code> <code> }</code> <code>}</code>
Tooltip	This shows a tooltip when the user hovers a mouse over a field in the Inspector.	<code>public class PlayerScript: MonoBehaviour</code> <code>{</code> <code> [Tooltip("Health value between 0 and 100.")]</code> <code> int health = 0;</code> <code>}</code>
Space	This adds a small space between your fields (without any additional text) to create visual separation between your fields.	<code>[Space(10)] // 10 pixel of spacing added</code> <code>int p = 5;</code>

Attribute	Description	Example
Header	This adds some bold text and spacing to help organize your variables in the Inspector. Only add this to the first field that you want to belong to the group.	<pre>public class PlayerScript: MonoBehaviour { [Header("Health Settings")] public int health = 0; public int maxHealth = 100; [Header("Shield Settings")] public int shield = 0; public int maxShield = 0; }</pre>
Multiline	This makes the string editable with the multiline text field. Pass in an optional int to designate the number of lines. Tip: Use this for annotating scripts with notes to yourself or another user.	<pre>[Multiline] public string textToEdit; [Multiline(20)] public string moreTextToEdit;</pre>
SelectionBase	This is useful for selecting an otherwise empty GameObject whose <i>children</i> may contain meshes. Add the attribute to any component on the base object. When picking objects in the Editor, the GameObject containing the [SelectionBase] attribute gets selected rather than the children.	<pre>// add this to the base GameObject [SelectionBase] public class PlayerScript: MonoBehaviour { }</pre>



Attributes affecting the Inspector fields

This is just a small sample of the numerous Attributes. Do you want to [rename your variables without losing their values](#)? Or [invoke some logic without needing an empty GameObject](#)? See the Scripting API for a complete list of [Attributes](#) for everything that's possible.

You can even create your own [PropertyAttribute](#) to define custom Attributes for your script variables.

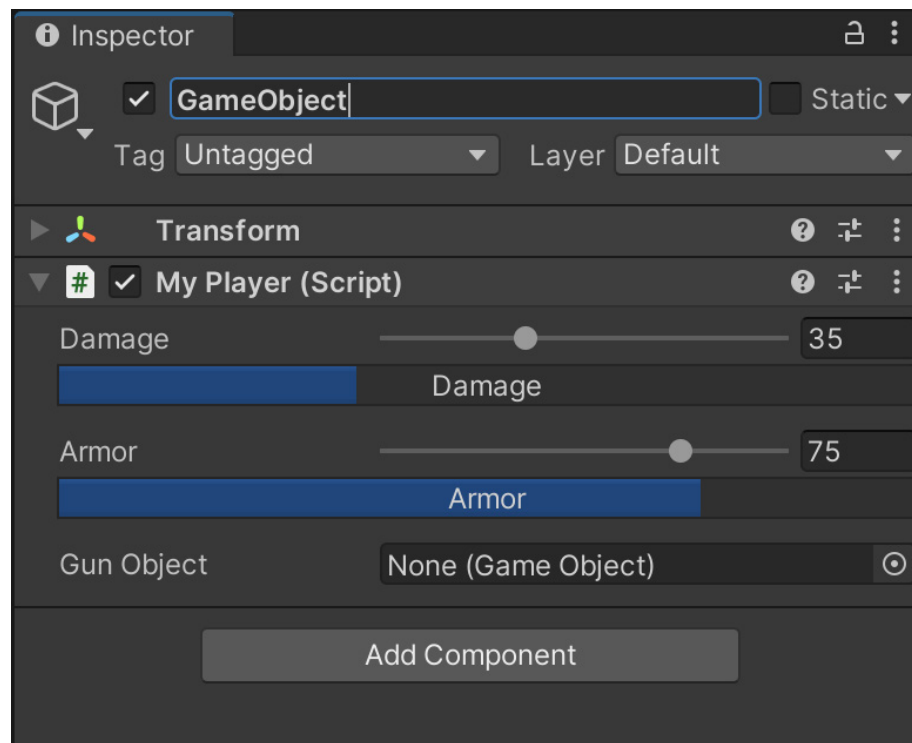
Custom windows and Inspectors

One of Unity's most powerful features is its extensible Editor. Use the [UI Toolkit](#) package or the immediate mode [IMGUI](#) to create Editor UIs such as custom windows and Inspectors.

UI Toolkit has a workflow similar to standard web development. Use its HTML and XML inspired markup language, UXML, to define user interfaces and reusable UI templates. Then, apply Unity Style Sheets (USS) to modify the visual style and behaviors of your UIs.

Alternatively, you can use immediate mode IMGUI. Derive from the [Editor](#) base class, then use the CustomEditor attribute.

Either solution can make a [custom Inspector](#).



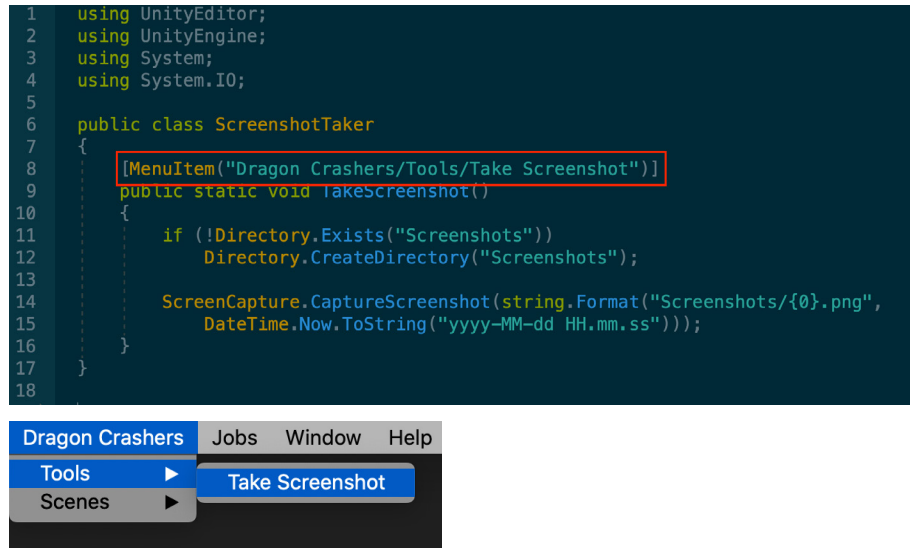
A custom Editor modifies how the MyPlayer script displays in the Inspector.

See [Creating user interfaces \(UI\)](#) for more detail on how to implement custom Editor scripts using either UI Toolkit or IMGUI. For a quick introduction to UI Toolkit, watch the [Getting Started with Editor Scripting](#) tutorial.

Custom menus

Unity includes a simple way to customize Editor menus and menu items, the **MenuItem** Attribute. You can apply this to any static method in your scripts.

If you have functions for your project that you will use frequently, organize them into menu items. This allows you to build a basic user interface with just a single `PropertyAttribute` modifier.



The `MenuItem` Attribute creates a simple interface to attach the static method (Take Screenshot).

Enter Play Mode settings

When you enter Play Mode, your project starts and runs as it would in a build. Any changes you make in the Editor during Play Mode reset when you exit Play Mode.

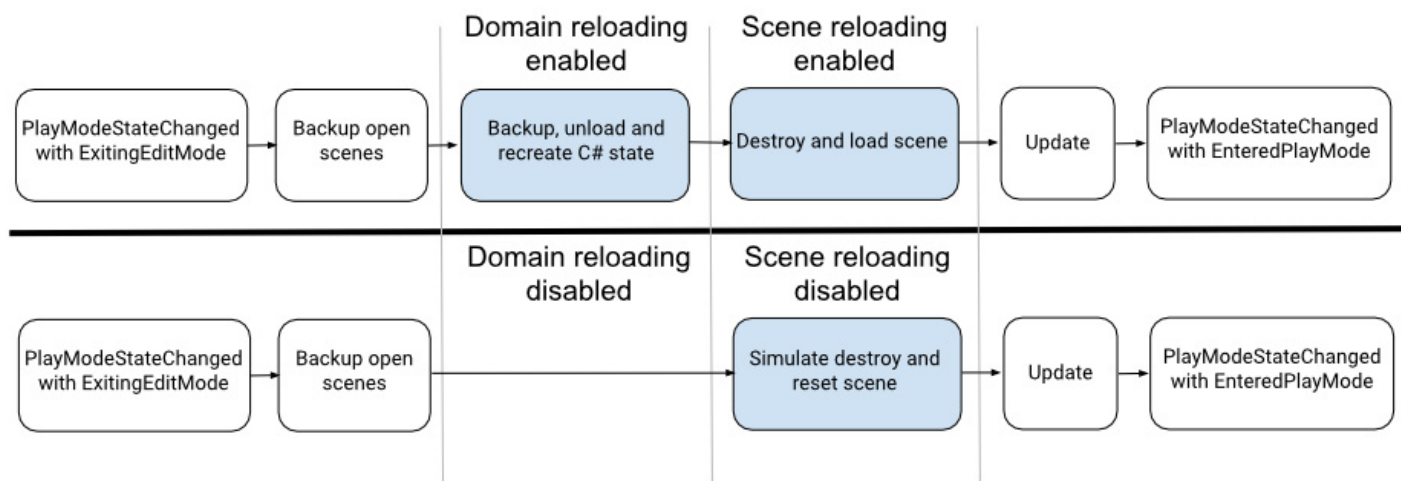
Each time that you enter Play Mode in the Editor, Unity performs two significant actions:

- **Domain Reload:** Unity backs up, unloads, and recreates scripting states.
- **Scene Reload:** Unity destroys the Scene and loads it again.

These two actions take more and more time as your scripts and Scenes become more complex.

If you don't plan on making any more script changes, the **Enter Play Mode Settings (Edit > Project Settings > Editor)** can save you a bit of compile time. Unity gives you the option to disable either Domain Reload, Scene Reload, or both. This can speed up entering and exiting Play Mode.

Just remember that if you *do* plan on making further script changes, you need to re-enable Domain Reload. Likewise, if you modify the Scene Hierarchy, you should re-enable Scene Reload. Otherwise, unexpected behavior could result.



The effects of disabling the Reload Domain and Reload Scene settings.

Script templates

Do you find that you make the same changes every time you create a new script? Do you instinctively add a namespace or delete the update event function? Save yourself a few keystrokes and create consistency across the team by setting up the script template for your preferred starting point.

Every time you create a new script or shader, Unity uses a template stored in **%EDITOR_PATH%\Data\Resources\ScriptTemplates:**

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

The default MonoBehaviour template is this one:

81-C# Script-NewBehaviourScript.cs.txt

There are also templates for shaders, other behavior scripts, and assembly definitions.

For project-specific script templates, create an **Assets/ScriptTemplates** folder. Copy the script templates into this folder to override the defaults.

You can also modify the default script templates directly for all projects, but make sure that you back up the originals before making any changes.

The original 81-C# Script-NewBehaviourScript.cs.txt file looks like this:

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

#ROOTNAMESPACEBEGIN#
public class #SCRIPTNAME# : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        #NOTRIM#
    }

    // Update is called once per frame
    void Update()
    {
        #NOTRIM#
    }
}
#ROOTNAMESPACEEND#
```

There are two keywords that may be helpful to you:

- **#SCRIPTNAME#** indicates the filename entered or the default filename (for example, NewBehaviourScript).
- **#NOTRIM#** ensures that the brackets contain a line of whitespace.

For example, you may want to set up the default MonoBehaviour to have default regions in order to stay organized:

```
/*
 * Modified template by Unity Support.
 */

using UnityEngine;

public class #SCRIPTNAME# : MonoBehaviour
{
    #region Public Fields
    #endregion

    #region Unity Methods
    void Start()
    {
    }

    void Update()
    {
    }
    #endregion

    #region Private Methods
    #endregion
}
```

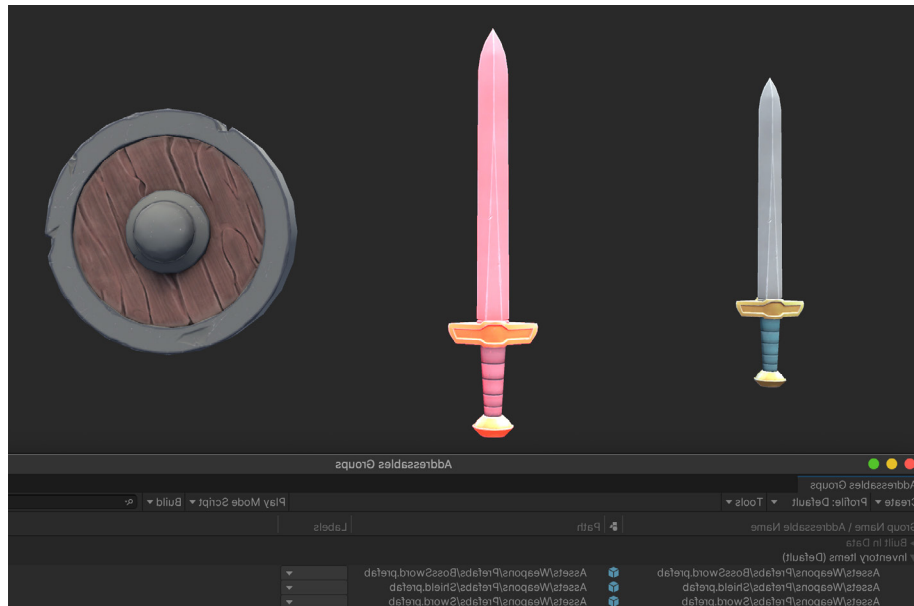
Relaunch the Unity Editor, and your changes should appear every time you create a custom MonoBehaviour.

Modify the other templates in a similar fashion. Remember to keep a copy of your original and modifications somewhere outside the Unity project for safekeeping.

Addressables

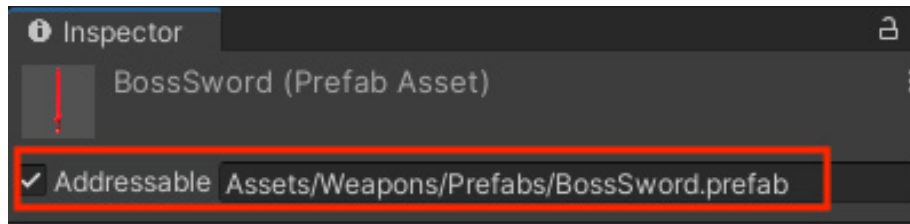
The [Addressable Asset System](#) simplifies how you manage the assets that make up your game. Any asset, including Scenes, Prefabs, text assets, and so on, can be marked as “addressable” and given a unique name. You can call this alias from anywhere.

Adding this extra level of abstraction between the game and its assets can streamline certain tasks, such as creating a separate downloadable content pack. This system makes referencing those asset packs easier as well, whether they’re local or remote.



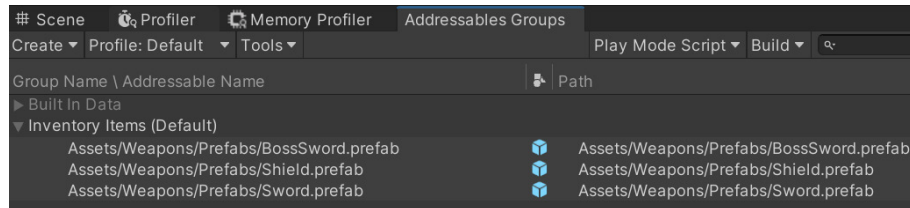
In this example, Addressables tracks the inventory of Prefabs.

To begin, install the Addressables package from the Package Manager, and add some basic settings to the project. Each asset or Prefab in the project should have the option to be made “addressable” as a result. Checking the option under an asset’s name in the Inspector assigns it a default unique address.



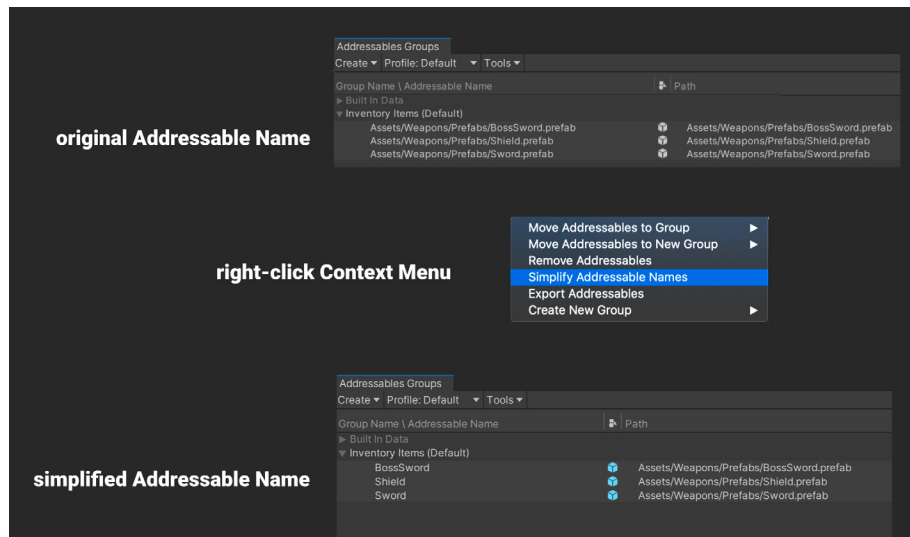
Addressable option enabled with default Addressable Name

Once marked, the corresponding assets appear in the **Window > Asset Management > Addressables > Groups** window.

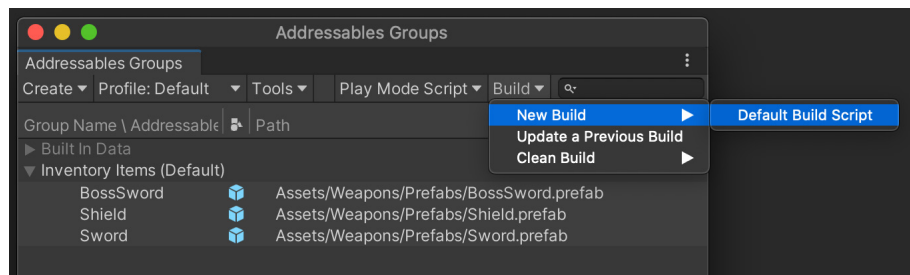


In Addressables Groups, you can see each asset's custom address, paired with its location.

For convenience, you can either rename each address in the asset's individual Address field or simplify them at once.



Simplify the Addressable Names with a single menu action, or rename them individually.



Use the default build script to generate an Addressable Group asset bundle.

Bundle these assets to be hosted on a server elsewhere or distribute them locally with your project. Wherever each asset resides, the system will locate it using the Addressable Name string.

You can now use your addressable assets using the Addressables API.

For example, without Addressables, if you wanted to instantiate a Prefab in your scene, you might do this:

```
public GameObject prefabToCreate;

public void CreatePrefab()
{
    GameObject.Instantiate(prefabToCreate);
}
```

The disadvantage here is that any referenced Prefab (like `prefabToCreate`) would load into memory, even if the scene didn't require it.

Using Addressables, you could instead do this:

```
public string prefabByAddress;
...
public void CreatePrefabWithAddress()
{
    Addressables.Instantiate(prefabByAddress, instantiationParameters, bool);
}
```

This loads the asset by its address string. The Prefab does not load into memory until it's needed (when we invoke `Addressables.Instantiate` inside `CreatePrefabWithAddress`). In addition, [Addressables](#) provides high-level reference counting and automatically unloads bundles and their associated assets when they're no longer in use.

[Tales from the Optimization Trenches: Saving Memory with Addressables](#) offers an example of how to organize your Addressable Groups to be more memory efficient. You can also check out the [Addressables: Introduction to Concepts](#) tutorial for a quick overview of how the Addressable Asset system can work in your project.

Operating live games: Cloud Content Delivery with Addressables

If you are operating a live game, then you might want to consider using Unity's Cloud Content Delivery (CCD) solution with Addressables. The Addressables system stores and catalogs game assets so that they can be automatically found and called, and then CCD pushes those assets directly to your players, completely separate from your code. This reduces your build size and eliminates the need to have your players download and install new game versions whenever you want to make an update. To learn more, [read this blog post](#) on the integration between Addressables and Cloud Content Delivery.

Preprocessor directives

The [Platform Dependent Compilation](#) feature allows you to partition your scripts to compile and execute code for a specifically targeted platform.

This example makes use of the existing platform **#define** directives with the **#if** compiler directive:

```
using UnityEngine;using System.Collections;

public class PlatformDefines : MonoBehaviour
{
    void Start ()
    {
        #if UNITY_EDITOR
            Debug.Log("Unity Editor");
        #endif

        #if UNITY_IOS
            Debug.Log("Iphone");
        #endif

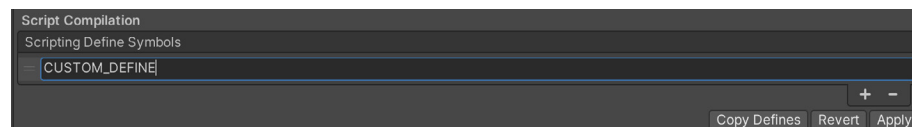
        #if UNITY_STANDALONE_OSX
            Debug.Log("Stand Alone OSX");
        #endif

        #if UNITY_STANDALONE_WIN
            Debug.Log("Stand Alone Windows");
        #endif
    }
}
```

Use the **DEVELOPMENT_BUILD** #define to identify whether your script is running in a player which was built with the **Development Build** option.

You can also compile selectively for specific Unity versions and/or scripting backends.

You can supply your own custom #define directives when testing in the Editor. Open the **Other Settings** panel of the [Player](#) settings, and navigate to **Scripting Define Symbols**.



See [Platform Dependent Compilation](#) for more information about Unity's preprocessor directives.

ScriptableObjects

A ScriptableObject is a data container that saves large amounts of data, independent of class instances. ScriptableObjects can reduce your project's memory usage by avoiding copies of values.

This is useful if your project has a [Prefab](#) that stores unchanging data in an attached MonoBehaviour script. Unlike with MonoBehaviours, the data saved to ScriptableObjects is written to disk as an asset and not attached to a GameObject. Thus, it can persist between sessions.

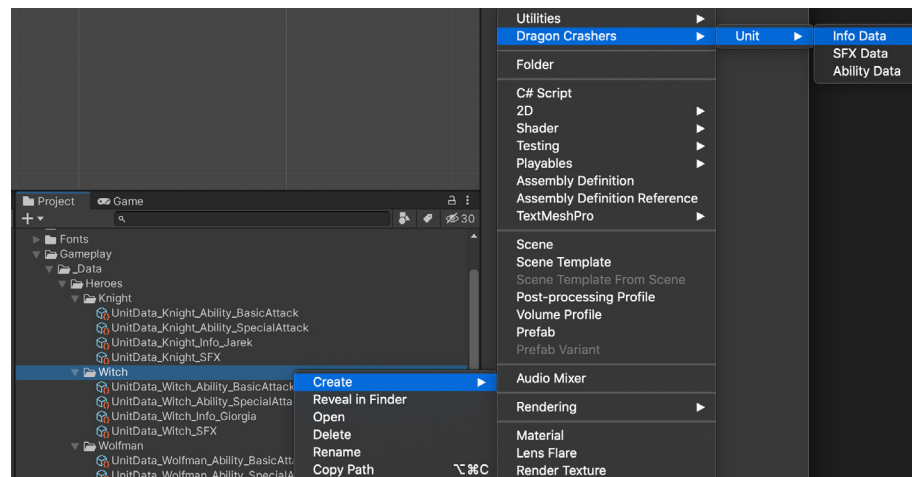
Dragon Crashers demonstrates a typical use case. A UnitInfoData class inherits from ScriptableObject. Each of its instances contains the unit's name, sprite, and health settings. This data remains constant over the course of gameplay, making it especially suitable for storage inside a ScriptableObject.

```
using UnityEngine;

namespace DragonCrashers
{
    [CreateAssetMenu(fileName = "Data_Unit_", menuName = "Dragon Crashers/Unit/Info Data", order = 1)]
    public class UnitInfoData : ScriptableObject
    {
        [Header("Display Infos")]
        public string unitName;
        public Sprite unitAvatar;

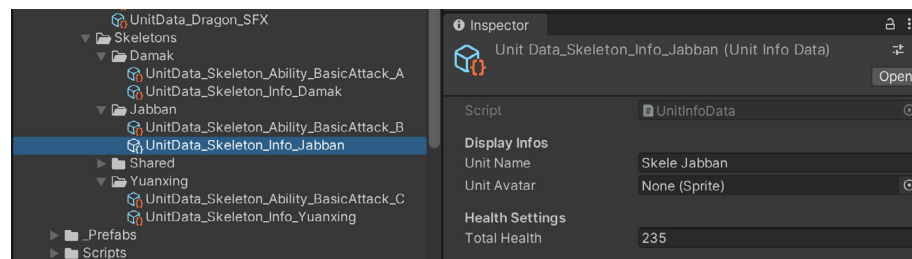
        [Header("Health Settings")]
        public int totalHealth;
    }
}
```

A ScriptableObject defines a data container object.



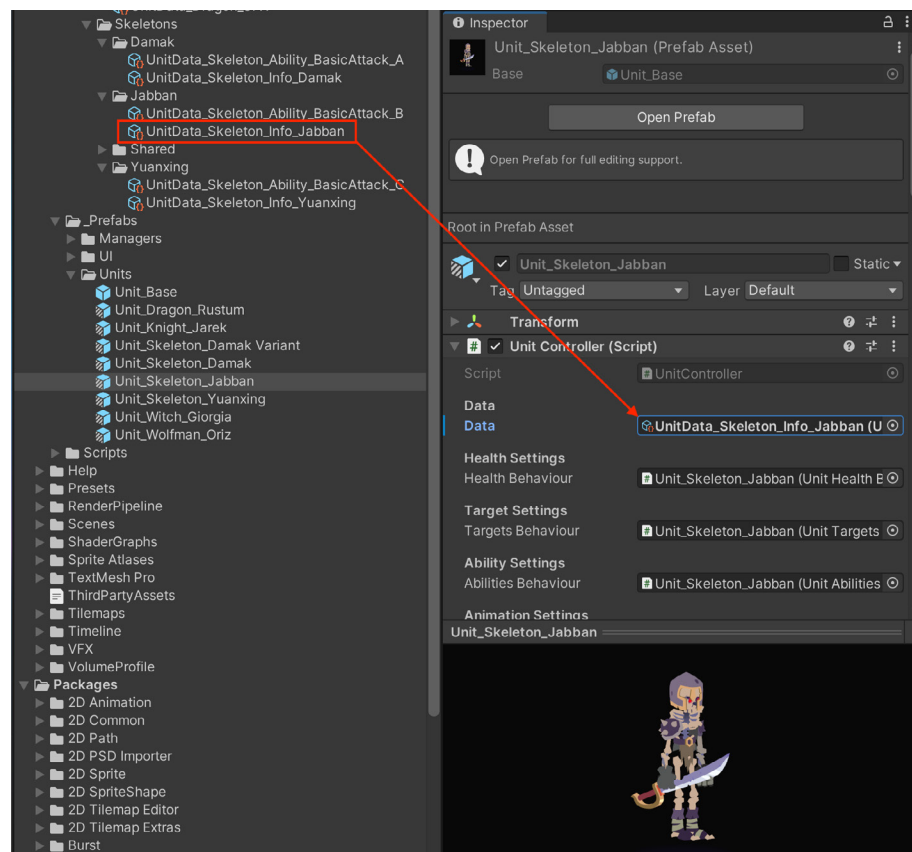
The CreateAssetMenu attribute generates a context menu item to help you generate an asset on disk. Each unit has additional ScriptableObjects for sound effects and special abilities.

With the assets created in the project window, you can fill in the correct values using the Inspector: Unit Name, Unity Avatar (Sprite), and Total Health.

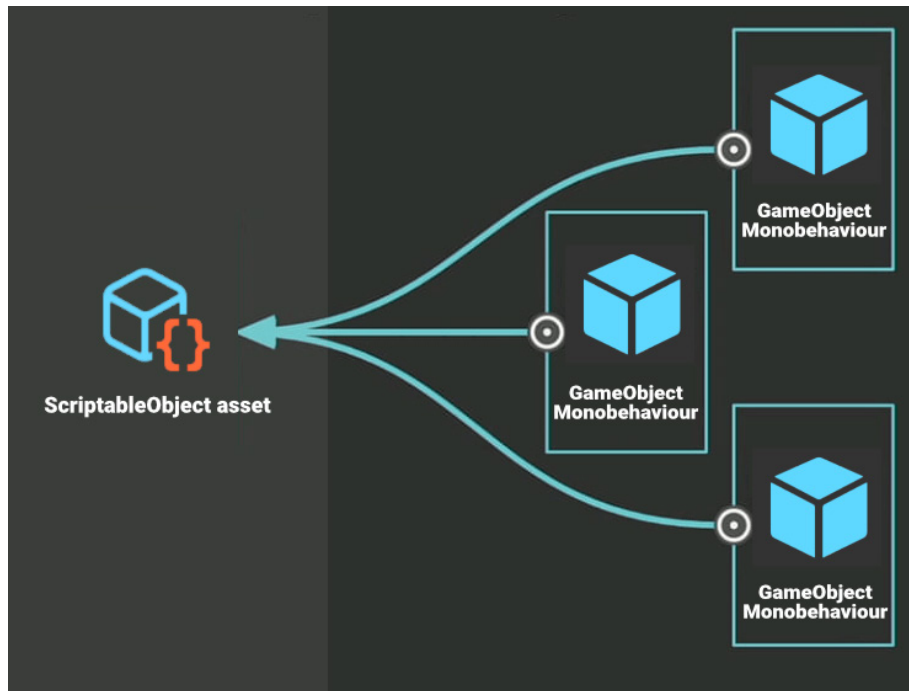


Use the Inspector to fill out values for the ScriptableObject asset. These values won't change during gameplay.

A **GameObject** (like the **UnitController** in this case) can then reference the **ScriptableObject** asset. If the Scene suddenly fills with units, the data on the **ScriptableObject** asset does not duplicate, saving memory.



The **Monobehaviour** object (**UnitController**, shown above) refers to the **ScriptableObject** data asset in the project.



Save memory and stay organized with ScriptableObjects. Set static data and settings in the asset in the project just once, even if you have lots of GameObjects.

Even if you add a thousand instances of a Prefab to your Scene, they still refer to the same data stored in your asset. Setting up the set of values just once guarantees consistency.

As your game scales up with more unit types, simply create more ScriptableObject assets and swap them out appropriately. Maintain your gameplay data just by tweaking the centrally stored assets.

ScriptableObjects don't replace keeping [persistent data](#) for the rest of your application's save files, where the data may change during gameplay. It's a workflow suited more for storing your static gameplay settings and values. Unlike parsing data from JSON or XML, reading a ScriptableObject asset won't generate garbage (and, as a bonus, it's faster).

Refer to [ScriptableObject documentation](#) for more information about using ScriptableObjects in your application. Also, watch [Better Data with Scriptables Objects in Unity](#) for a quick introduction, and see how they can help with Scene management in [Achieve Better Scene Workflow with ScriptableObjects](#).

Optimization tip

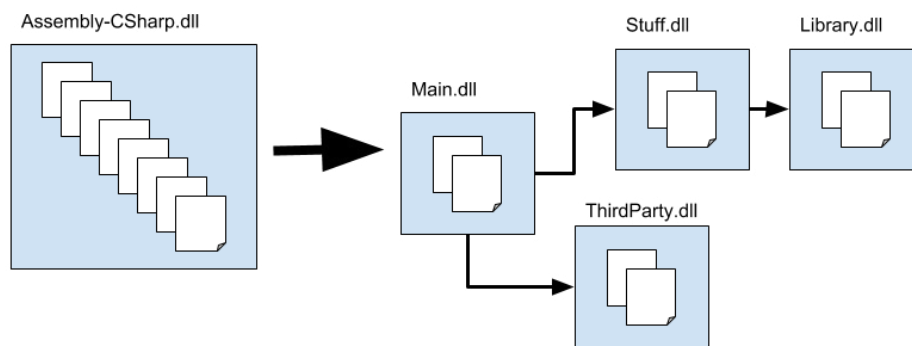
We recommend binary serialization formats such as [MessagePack](#) or [Protocol Buffers](#) for save data, rather than text-based ones such as JSON or XML. In Project Reviews, these binary serialization formats reduce the memory and performance issues associated with the latter.

Managing assemblies

An assembly is a C# code library, a collection of types and resources that are built to work together and form a logical unit of functionality. By default, Unity compiles almost all of your game scripts into the predefined assembly, **Assembly-CSharp.dll**. This works for small projects, but it has some drawbacks:

- Every time you change a script, Unity recompiles all other scripts;
- Any script can access types defined in any other script;
- All scripts are compiled for all platforms.

Organizing your scripts into custom assemblies promotes modularity and reusability. It prevents them from getting added to the default assemblies automatically and limits which other scripts they can access.



The project split into multiple assemblies

You might split up your code into multiple assemblies, as shown in the diagram above. Here, any changes to the code in Main cannot affect the code in Stuff. Similarly, because Library doesn't depend on any other assemblies, you can more easily reuse the code in Library in another project.

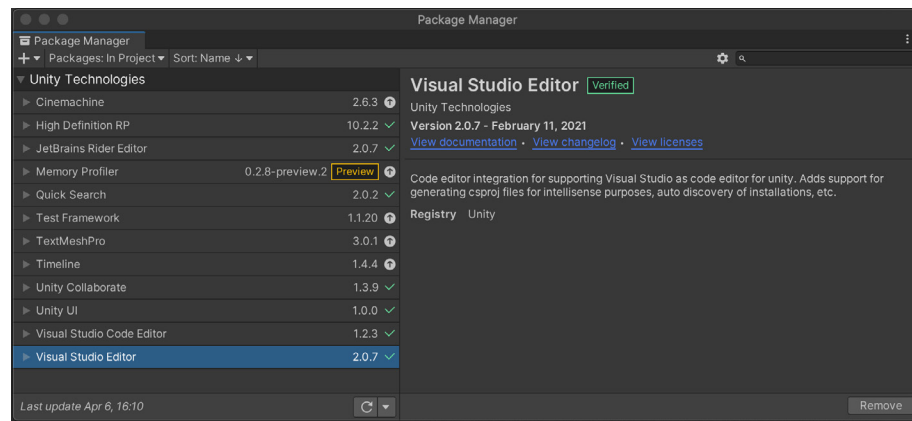
[Assemblies in .NET](#) has general information about assemblies in C#. Refer to [Assembly definitions](#) in the Unity documentation for more information about defining your own assemblies in Unity.

IDE support

Unity offers support for the following integrated development environments ([IDEs](#)):

- **Visual Studio** (default IDE on Windows and macOS)
- **Visual Studio Code** (Windows, macOS, Linux)
- **JetBrains Rider** (Windows, macOS, Linux)

IDE integrations for all three of these environments appear as packages in the Package Manager.



IDE integrations as packages

Visual Studio is installed by default when you install Unity on Windows and macOS. If you want to use another IDE, simply browse for the editor in **Unity > Preferences > External Tools > External Script Editor**.

Rider is built on top of [ReSharper](#) and includes most of its features. It supports C# debugging on the .NET 4.6 scripting runtime in Unity (C# 8.0). For more information, see JetBrains's documentation on [Rider for Unity](#).

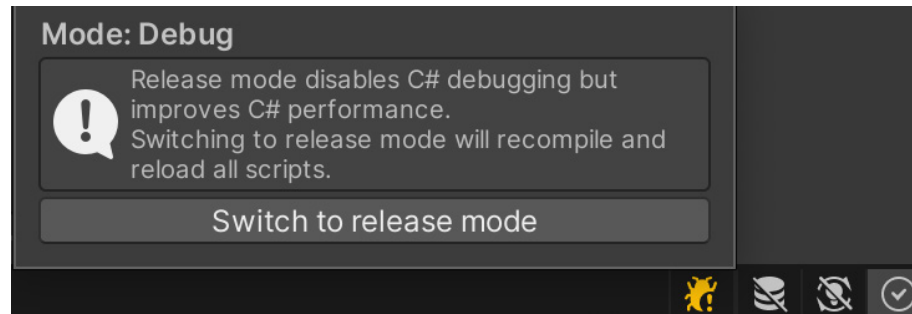
VS Code is a free, streamlined code editor with support for debugging, task running, and version control. Note that Unity requires [Mono](#) (macOS and Linux), [Visual Studio Code C#](#), and [Visual Studio Code Debugger for Unity](#) (not officially supported) when using VSCode.

Each IDE has its own productive merits. See [Integrated development environment \(IDE\)](#) support for more information about choosing an IDE.

Debugging

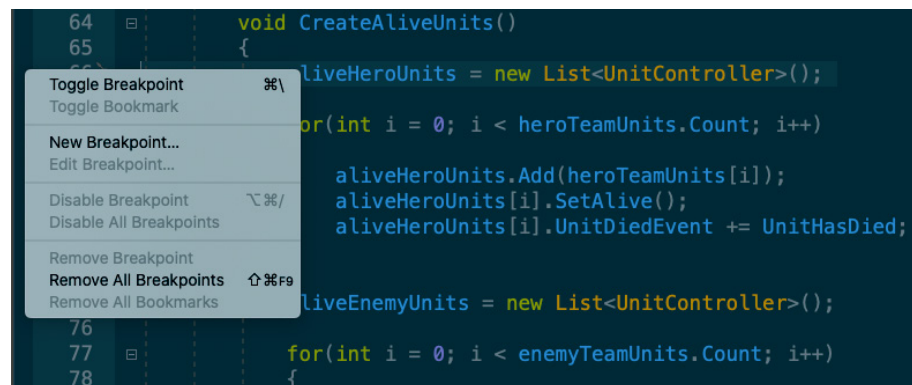
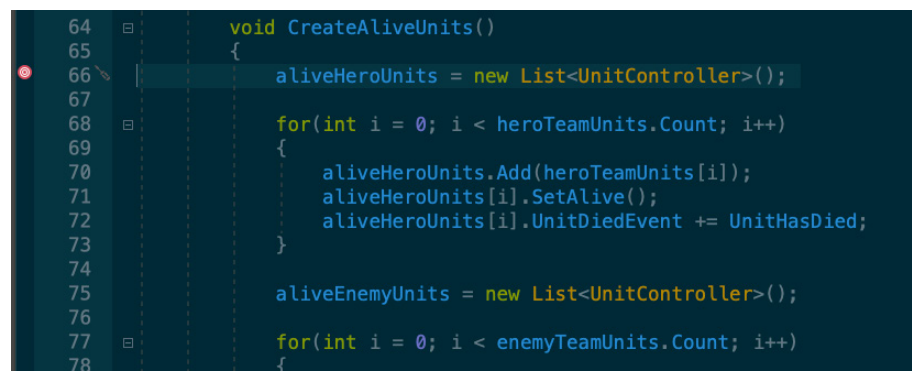
The Unity Debugger allows you to debug your C# code while the Unity Entity is in Play Mode. You can attach breakpoints within the code editor in order to inspect the state of your script code and its current variables at runtime.

Set the Code Optimization mode to **Debug** in the bottom right of the Unity Editor Status Bar. You can also change this mode on startup at **Edit > Preferences > General > Code Optimization On Startup**.



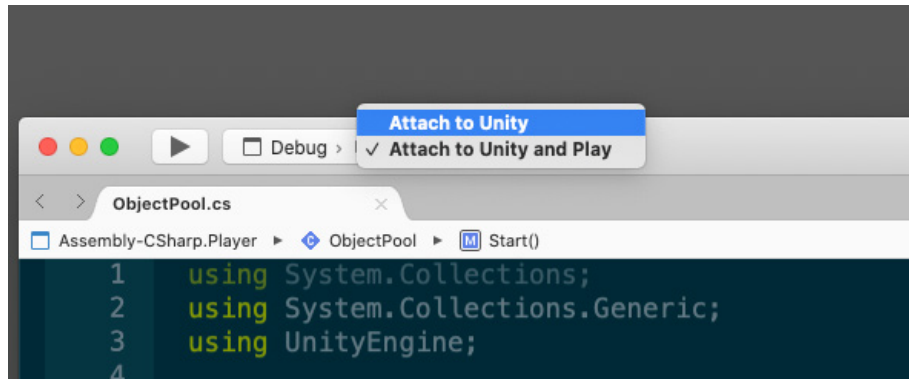
Debug Mode

In the code editor, set a breakpoint where you want the debugger to pause execution. Simply click over the left margin/gutter area where you want to toggle a breakpoint (or right-click there to use the context menu for more options). A red circle appears next to the line number of the highlighted line (see image below).



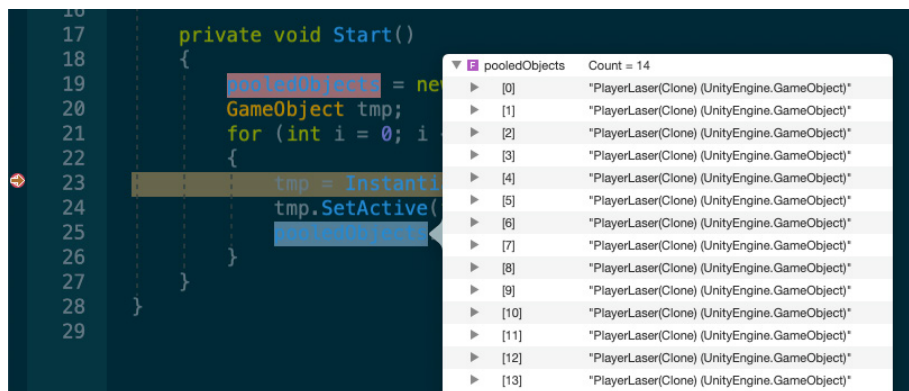
Toggling a breakpoint

Select **Attach to Unity** in your code editor. In the Unity Editor, run the project.



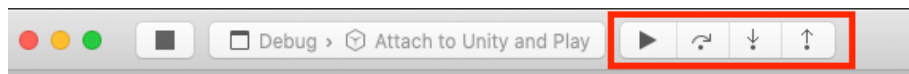
Attaching the Debugger to Unity

In Play Mode, the application will pause at the breakpoint, giving you time to inspect variables and investigate any unintended behavior.



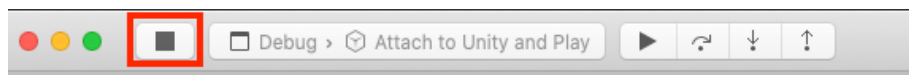
Debugging variables

In the example above, you can inspect the variables when debugging, watching the list build up one step at a time during execution.



Debug controls: Continue Execution, Step Over, Step Into, and Step Out

Use the Continue **Execution**, **Step Over**, **Step Into** and **Step Out** controls to navigate the control flow.



Debug controls: Stop

Press **Stop** to discontinue debugging and resume execution in the Editor.

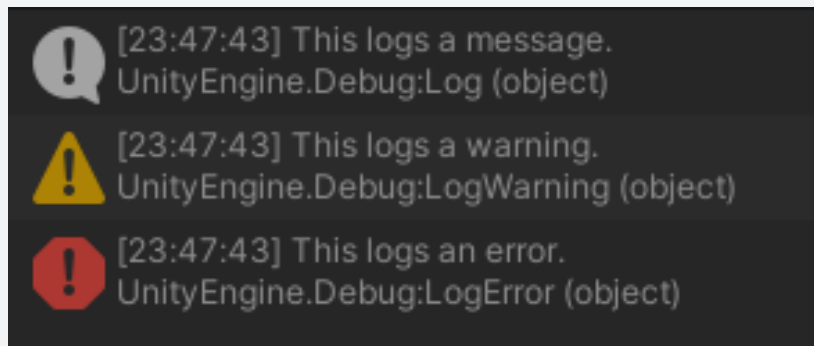
You can debug script code in a Unity Player as well. Just make sure that **Development Build** and **Script Debugging** are both enabled in the **File > Build Settings** before you build the Player. Check **Wait for Managed Debugger** to wait for the debugger before the Player executes any script code.

To attach the code editor to the Unity Player, select the IP address (or machine name) and port of your player. Then proceed normally in Visual Studio with the Attach To Unity option.

Additional debugging tips

Unity also includes a [Debug](#) class to help you visualize information in the Editor while it is running. Use it to print messages or warnings into the Console window, draw visualization lines in the Scene view and Game view, and pause Play Mode in the Editor from script.

1. Pause execution with [Debug.Break](#). This is useful if you want to check certain values in the Inspector when the application is difficult to pause manually.
2. You should be familiar with [Debug.Log](#), [Debug.LogWarning](#), and [Debug.LogError](#) for printing Console messages. Also handy is [Debug.Assert](#), which asserts a condition and logs an error on failure (only works if UNITY_ASSERTIONS symbol is defined).



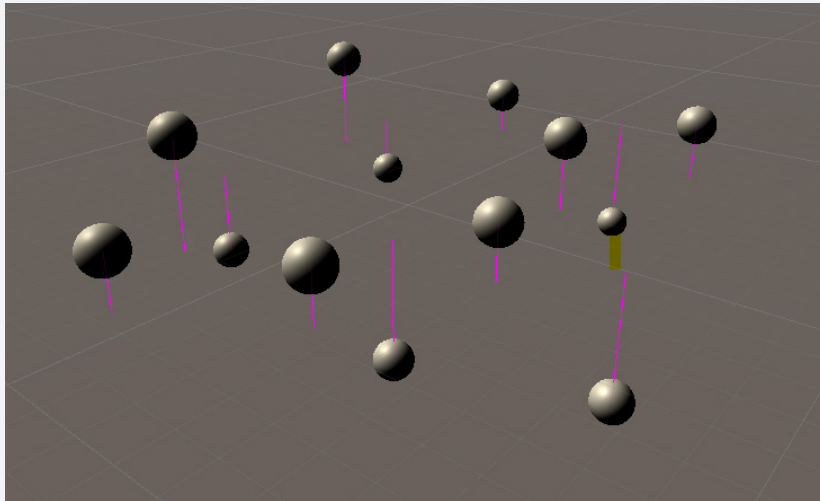
Log messages, warnings, and errors in the Console

3. 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.
4. Use [Rich Text](#) to mark up your [Debug.Log](#) statements. This can help you enhance error reports in the Console.
5. Unity does not strip the [Debug](#) logging APIs from non-development builds automatically. Wrap your **Debug Log** calls in custom methods and decorate them with the **[Conditional]** attribute.

Removing the corresponding **Scripting Define Symbol** from the Player Settings compiles out the Debug Logs all at once. This is identical to wrapping them in **#if...#endif** preprocessor blocks.

See this [General Optimizations](#) guide for an example.

6. Troubleshooting physics? [Debug.DrawLine](#) and [Debug.DrawRay](#) can help you visualize raycasting.



Debug.DrawLine

7. If you only want code to run when **Development Build** is enabled, check if [Debug.isDebugBuild](#) returns true.
8. Use [Application.SetStackTraceLogType](#) or the equivalent checkboxes in PlayerSettings to decide which kinds of log messages should include stack traces. Stack traces can be useful, but they are slow and generate garbage.

Stack Trace*			
Log Type	None	ScriptOnly	Full
Error	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Assert	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Warning	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Log	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Exception	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Visual Studio shortcuts

If you use Visual Studio as your IDE of choice, these shortcuts may prove useful.

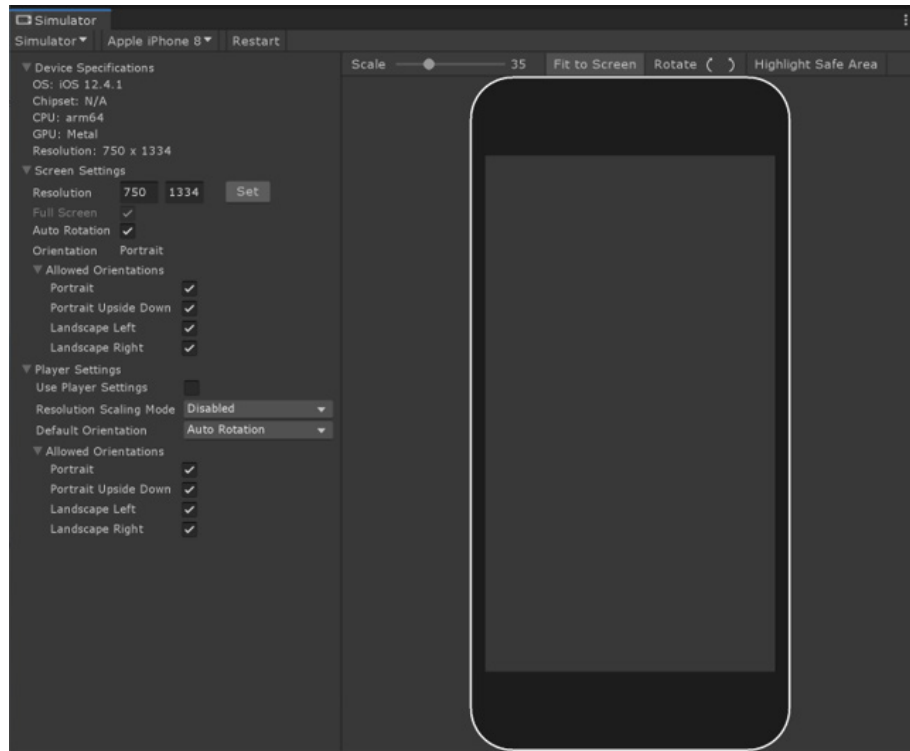
Action	Windows	Mac
Search your entire project for anything.	Ctrl + T	Cmd + .
Implement Unity Messages (boilerplate code)	Ctrl + Shift + M	Cmd + Shift + M
Comment out code blocks	Ctrl + K / Ctrl + C	Cmd + /
Uncomment blocks of code	Ctrl + K / Ctrl + U	Cmd + /
Copy from clipboard history	Ctrl + Shift + V	
View task list	Ctrl + T	No default keybinding, but you can bind it.
Insert a surrounding snippet such as namespace	Ctrl + K + S	No default keybinding, but you can bind it.
Rename a variable while updating all references	Ctrl + R	Cmd + R
Compile the code	Ctrl + Shift + B	Cmd + Shift + B

Watch [Visual Studio tips and tricks to boost your productivity](#) for more workflow improvements with Visual Studio.

Interested in exploring JetBrains Rider? Check out [Fast C# Scripting in Unity with JetBrains Rider](#) or [tips](#) for using JetBrains Rider as your code editor.

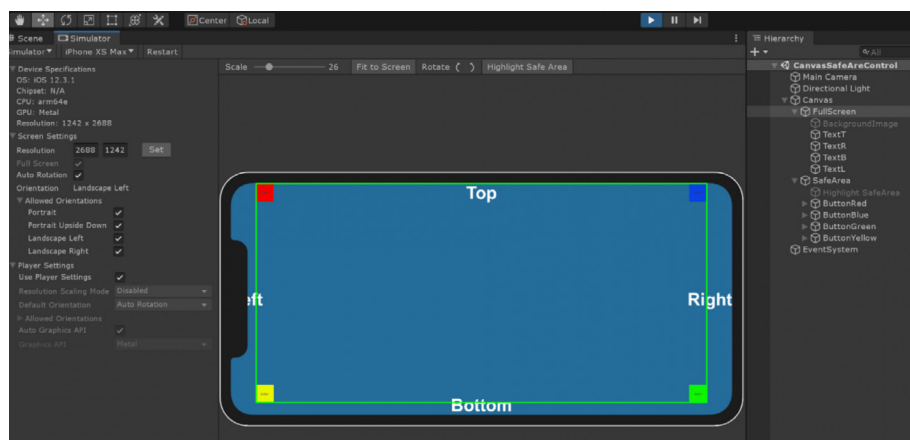
Device Simulator

If you're developing for mobile, the **Device Simulator** (currently a Preview Package) can help you simulate your application on different devices. Even if you have physical access to all of your targeted hardware, building the content for each device can be time consuming.



The Device Simulator

Use the Device Simulator to run a quick preview before you need to make an actual build. You can simulate specific resolutions or hardware conditions and adjust your UI to the physical notch/cutouts in Game view.



Adjusting the UI to the device's physical screen

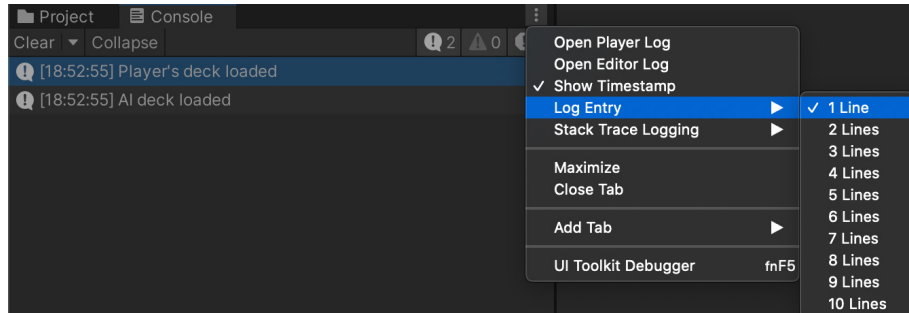
The Device Simulator also approximates the hardware's performance. Adjust quality settings based on the RAM, chipset, and other hardware specs, and your game code will simulate playback on the platform accordingly.

A list of predefined phones and tablets comes with the package (in the **com.unity.device-simulator/com.unity.device-simulator** folder). Device definitions are stored in JSON files, and the list of devices regularly expands through package updates.

Read [Speed up mobile iteration with the Device Simulator](#) for more tips or watch this [demo of the Device Simulator in action](#).

Console Log Entry

By default, the Console Log Entry shows two lines. For improved readability, you can configure this to be more streamlined with one line (see image).



The Console Log Entry options

Alternatively, you can also use more lines if you want longer entries.

Custom Compiler status

When Unity compiles, the icon in the lower right corner is hard to see. Use this custom Editor script to call [EditorApplication.isCompiling](#). This makes the Compiler status more visible in a floating window.

Launch the MenuItem to initialize the window. Optionally, you can modify its appearance with a new GUIStyle to suit your preferences.

```
using UnityEditor;
using UnityEngine;

public class CustomCompileWindow : EditorWindow
{
    [MenuItem("Examples/CustomCompileWindow")]
    static void Init()
    {
        EditorWindow window = GetWindowWithRect(typeof(CustomCompileWindow), new Rect(0, 0, 200, 200));
        window.Show();
    }
    void OnGUI()
    {
        EditorGUILayout.LabelField("Compiling:", EditorApplication.isCompiling ? "Yes" : "No");
        this.Repaint();
    }
}
```

Team workflows

Building games is a collaborative art. Unity can help your team create together, faster by using Source Control to integrate everyone's changes and updates as they work.

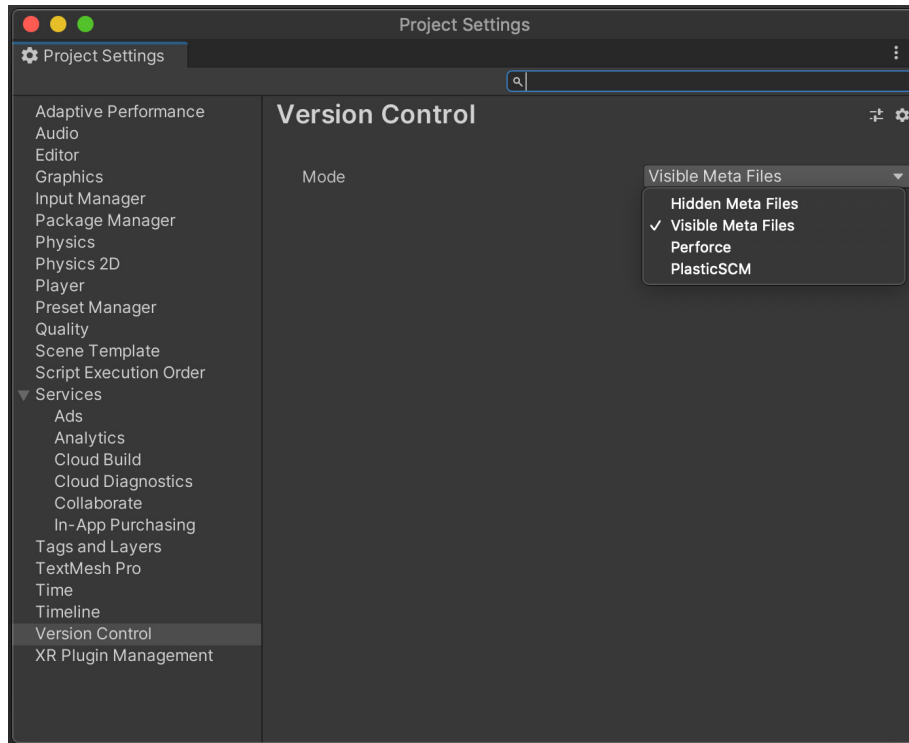
The Unity Teams and Unity Accelerator services can also assist you with wrangling your developers and artists, either locally or on the cloud.



Caption: Unity helps your team collaborate.

Source control

Unity has integrations with two version control systems: [Perforce](#) and [Plastic SCM](#). Set the Perforce or Plastic SCM servers for your Unity project in **Project Settings > Editor**. Configure the server (and your user credentials for Perforce) under **Version Control**.

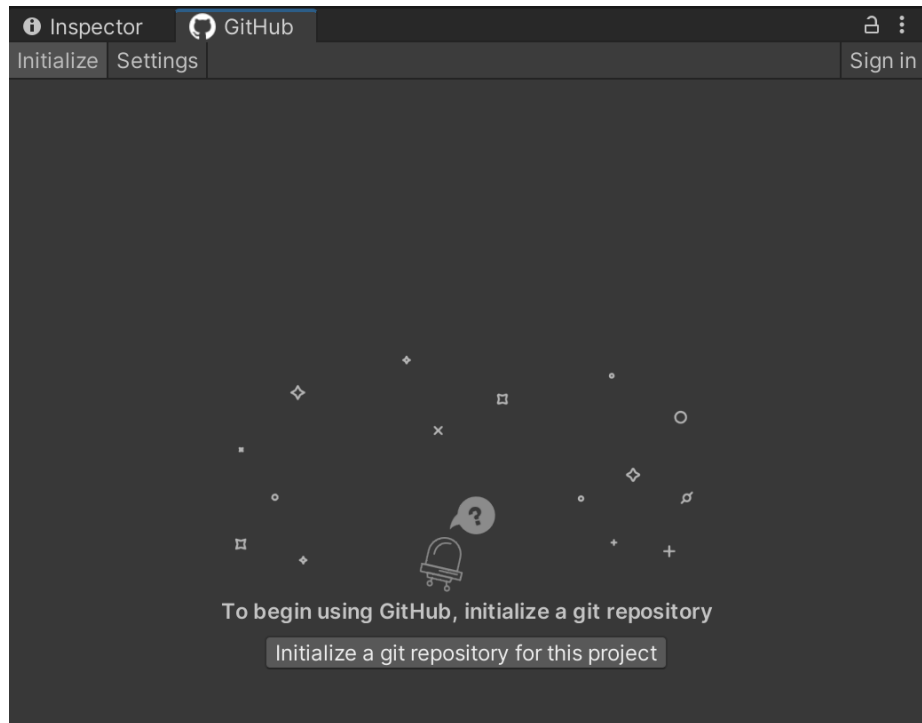


Configure your project to use Version Control.

You can also use an external system, such as Git, including [Git LFS \(Large File Support\)](#) for more efficient version control of your larger assets, like graphics and sound resources.

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.

Unity maintains a [.gitignore](#) file. This can help you decide what should and shouldn't go into the git repository and enforce those rules.



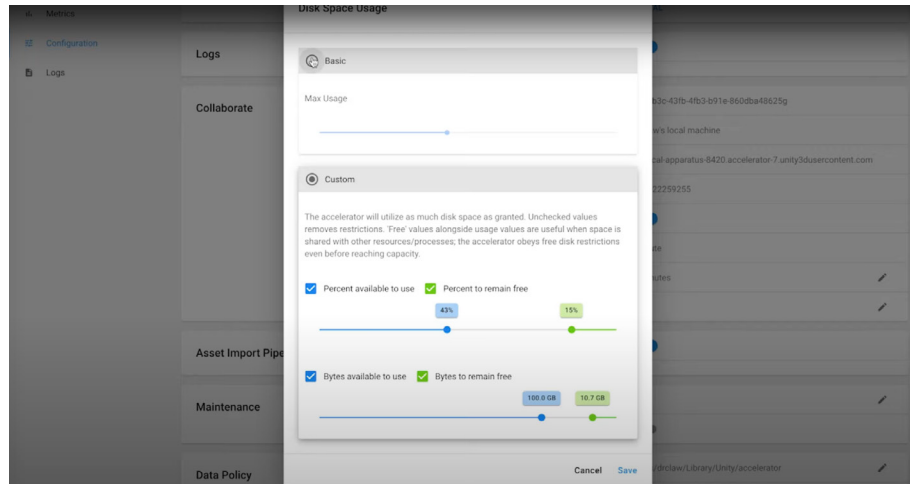
The GitHub for Unity extension

[Unity Teams](#) is another option for streamlining your team workflows. Unity Teams allows you to store your entire project in the cloud, so it's backed up and accessible anywhere. This makes it simple to save, share, and sync your Unity projects with anyone.

Unity Accelerator

The [Unity Accelerator](#) removes waiting time by caching copies of your team's assets. This means that only one person needs to perform the actual import, and the results will automatically be cached to the Unity Accelerator. The next time a team member goes to import the same version of the asset, the Unity Editor first checks the cache before starting the import process on their local machine.

In Unity 2020 LTS, you now have a local administrator dashboard for the Accelerator that enables you to configure the tool, see statistics like disk space usage or how much time you've saved, and diagnose issues with logs.



Unity Accelerator

Optionally, Accelerator can also be used with Unity Teams Advanced to share Collaborate source assets, which significantly reduces download time from the [Collaborate](#) service.

See the [requirements and installation procedure](#) for more information about the Unity Accelerator.

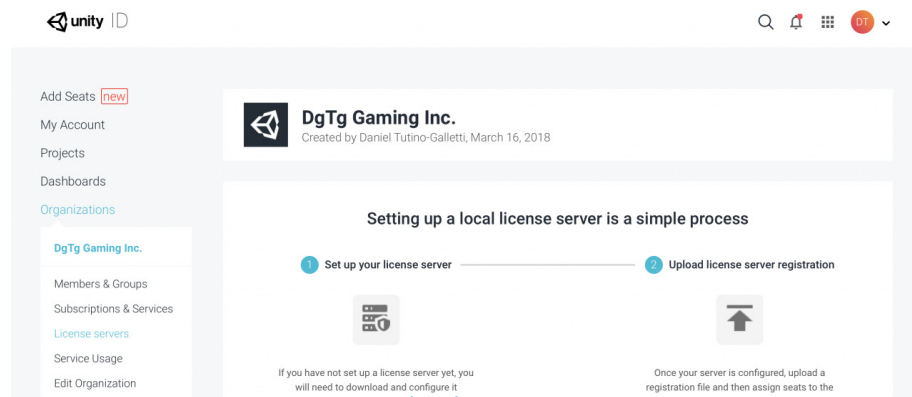
Unity Build Server

Consider enhancing your team's productivity by offloading the building process to network hardware using [Unity Build Server](#). This will help your creative team build the project as often as needed, allowing them to iterate more autonomously.

As your Unity project grows in size and complexity, generating a build consumes more and more time. If you're using your development workstations to build a project, you will lose productivity while your team waits for the build to complete.

Unity Build Server runs Unity in batch mode, exclusively for building Unity projects. Team members can request builds on demand at their own pace. This reduces wait time for bug fixes and releasing new features for testing. Building on separate machines reduces each developer's downtime and allows everyone to iterate more quickly.

Both Unity Pro and Unity Enterprise subscribers can get access to Unity Build Server. [Unity Pro](#) customers can get add-on packs, while Unity Enterprise customers receive a number of Build Server licenses based on their existing Enterprise licenses.



Setting up Unity Build Server



Resources for all Unity developers

You can find additional productivity tips, best practices, and news on the [Unity Blog](#), by searching the **#unitytips** hashtag, on [Unity community forums](#), and on [Unity Learn](#).

The [Unity Developer Tools](#) microsite makes it easy for you to find some of the best resources for developing with Unity, including documentation, the Knowledge Base, Issue Tracker, as well as our latest roadmap and release information.

For more information on improving and optimizing your workflows using Unity, [get in touch with our experts today](#).



unity.com