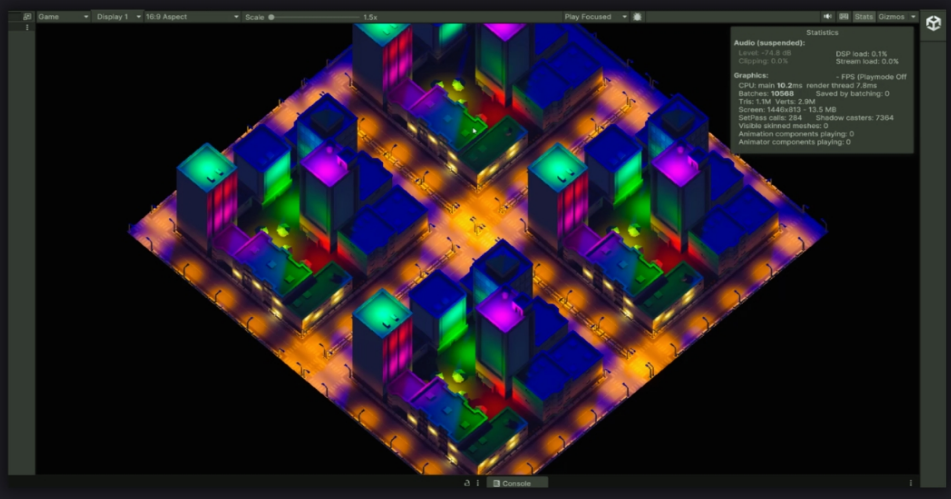
FIRST EDITION

UNITY 6 GAME OPTIMIZATION

Improve Performance and Create Smoother Gameplay



Daniel Buckley
Pablo Farias

ZENVA

Unity 6 Game Optimization: Improve Performance and Create Smoother Gameplay

Zenva Academy

© 2026 Zenva Pty Ltd. All rights reserved

Table of Contents

Introduction

Welcome. If you are reading this book, chances are you have built something in Unity that you are proud of, and now you want it to run better. Maybe your frame rate drops in a busy scene, or a build feels sluggish on target hardware and you are not sure where to look first. That is exactly the right reason to be here.

The good news is that optimization is not magic. It is a skill, and like any skill, it becomes much more approachable once you break it into clear, practical steps. This book is here to help you do exactly that.

Over the next several chapters, we will work through performance-focused ideas in a hands-on way, using real projects to explore how Unity behaves under pressure. Rather than treating optimization as a list of abstract rules, we will focus on how to observe problems, reason about them, and make improvements with purpose. The goal is not just to help you fix one scene or one bottleneck. It is to help you build instincts you can carry into every project you work on.

How to Use This Book

This book is designed to be used actively. Optimization clicks fastest when you can see the results of each change for yourself, so you will get the most from it if you follow along in Unity rather than just reading.

A few simple habits will make a big difference:

- Follow along in Unity as you read. Seeing the tools, settings, and profiler data in context will make the ideas stick.
- Experiment as you go. Try changing values, toggling features, or repeating a step in a slightly different way to see what changes and what does not.

- Pause to predict outcomes. Before checking a result, ask yourself what you expect to happen. This helps turn each exercise into real understanding.

- Do not worry about memorizing everything. What matters most is learning how to investigate performance problems and make thoughtful decisions.

You also do not need to read this book as if it were a test. Move steadily, revisit sections when needed, and give yourself room to explore. Optimization often becomes clearer the moment you try something with your own hands.

By the end, you should feel more comfortable profiling scenes, spotting common sources of cost, and making changes with a stronger sense of why they matter. That confidence is often the biggest win.

In the next chapter, we will get everything set up and ready so you can begin following along.

Getting Started

Before we dive into profiling and optimization, we need to make sure your environment is ready. This chapter walks through the required background knowledge, project files, and Unity installation so that everything runs smoothly once we start working with real scenes.

Prerequisites

Optimization builds on top of general Unity knowledge. Specifically, this book assumes you are comfortable with:

- **Unity Editor navigation**: opening windows, docking panels, switching between Scene and Game views, and using the Hierarchy and Project browsers
- **Scenes and GameObjects**: creating scenes, adding GameObjects, and understanding parent-child hierarchies
- **The Inspector**: viewing and modifying component properties, enabling checkboxes, and adjusting numeric fields
- **Basic components**: working with Transforms, Mesh Renderers, Lights, and Cameras
- **Materials and shaders**: understanding that materials control how objects look and that shaders define how rendering is performed
- **Prefabs**: knowing what prefabs are and how to replace or instantiate them in a scene
- **Play mode**: entering and exiting Play mode to test a scene at runtime

You do not need advanced C# scripting experience. While the book occasionally references MonoBehaviour methods like Update and

FixedUpdate, the focus is on Editor-side workflows and settings rather than writing code.

If you would like to brush up on any of these areas first, these resources cover the foundations well:

- Game Development with Unity: Learn the Foundations of Creating Games with Unity (https://academy.zenva.com/product/intro-to-unity-ebook/)

- C# for Unity Developers: Learn Programming Fundamentals Through Mini Projects (https://academy.zenva.com/product/unity-mini-projects-ebook/)

If you prefer course-based learning, these alternatives cover similar foundations:

- Intro to Game Development with Unity (https://academy.zenva.com/product/intro-to-game-development-with-unity/)

- Unity Mini-Projects – C# Fundamentals (https://academy.zenva.com/product/minigame-projects-explore-game-development-c/)

Beyond these resources, we do recommend having completed a few Unity games on your own. This will ensure you're well-prepared to discuss some of the more advanced systems we'll be featuring.

Regardless, even if it has been a while since you last used Unity, do not worry. This book explains each optimization topic step by step, so you can always revisit earlier material as needed.

Project Files

To follow along with the examples in this book, download the complete project here:

- Starter Project (https://academy.zenva.com/wp-content/uploads/2026/05/Starter-Project-Unity-Game-Optimization-v6.3.zip)

- Complete Project (https://academy.zenva.com/wp-content/uploads/2026/05/Complete-Project-Unity-Game-Optimization-6.3-LTS.zip)

Third Party Asset Licenses

Some of the assets provided in this course are subject to third-party licenses:

Assets by Kenney

City Models

URL: https://kenney.nl/

License Info: Creative Commons CC0 (https://creativecommons.org/publicdomain/zero/1.0/)

Assets by The Blender Foundation

Suzanne Monkey Head

URL: https://www.blender.org/

License Info: GNU General Public License (https://www.gnu.org/licenses/gpl-3.0.en.html)

Zenva-Created Assets

All other artwork and media assets included in the project files that are not listed above were created by **Zenva** and are released under:

Creative Commons CC0 (Public Domain)
https://creativecommons.org/publicdomain/zero/1.0/

Zenva-created source code included in the project files for this course is released under the MIT License (https://opensource.org/license/MIT). Third-party libraries and dependencies remain under their respective licenses.

Recommended Unity Version

Unity changes quickly, and small version differences can affect menus, workflows, and package behavior. For that reason, this book recommends using **Unity 6.3 LTS**.

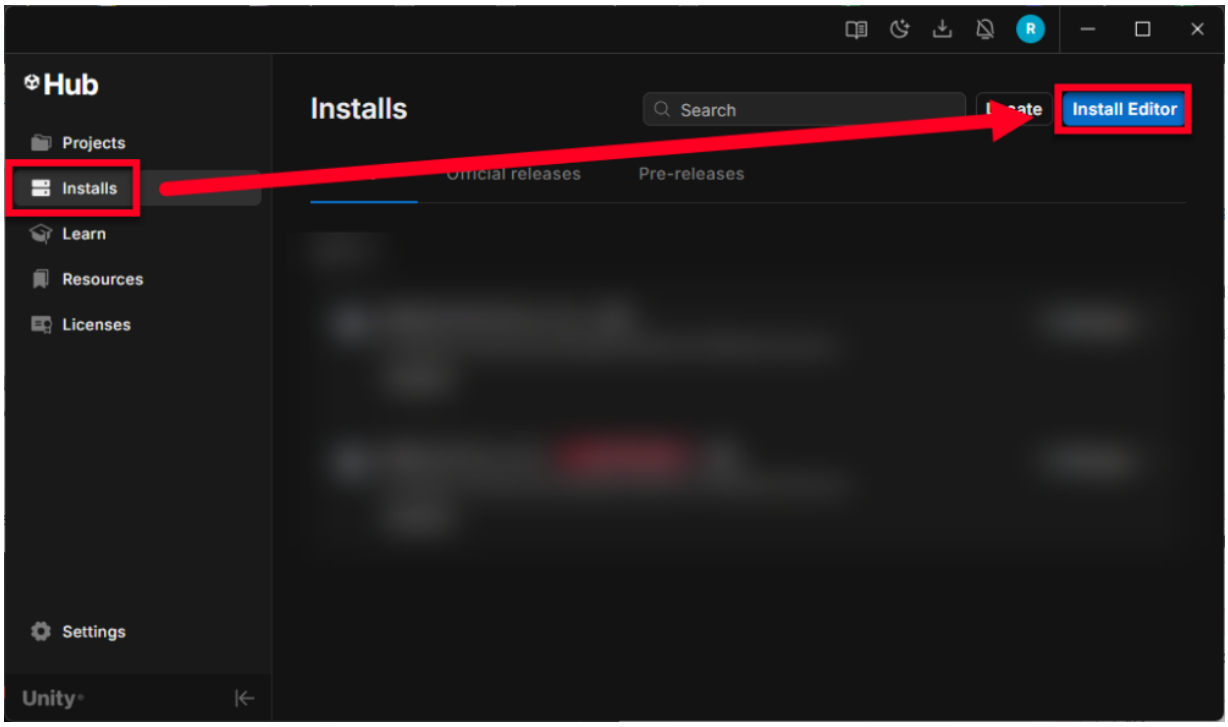
LTS stands for Long-Term Support. These releases are intended to provide a more stable foundation for learning and development, which makes them a strong choice for educational projects. You can learn more about Unity's LTS releases here: Unity LTS releases (https://unity3d.com/unity/qa/lts-releases).

It is best not to use a newer version than the one recommended here, since some project files or steps may behave differently.

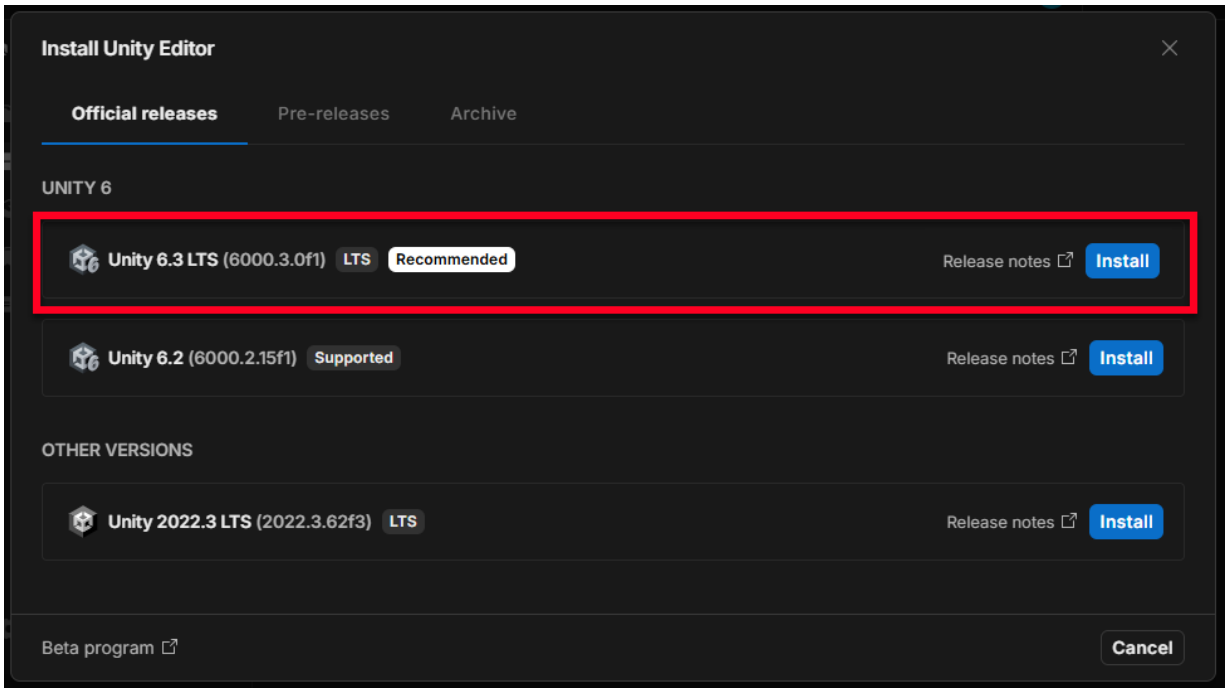
Installing Unity 6.3 LTS

If Unity is not installed yet, the easiest way to get started is through Unity Hub. This lets you manage editor versions and keep your setup aligned with the book.

To do so, open **Unity Hub** and go to the **Installs** section. From there, click the blue **Install Editor** button, as shown in the image below.



A new window will open with available Unity versions. In the **Official releases** list, look under the **Unity 6** section and select **Unity 6.3 [LTS]**.



Click **Install**, then follow the prompts to complete the setup.

Importantly, make sure you install Visual Studio Community (https://visualstudio.microsoft.com/vs/community/) along with Unity. This is usually bundled with Unity, but make sure to not uncheck it.

What This Book Covers

Every frame Unity renders involves a chain of decisions: what to draw, how to light it, and how much detail to include. When any part of that chain does more work than necessary, performance suffers. This book teaches you to spot those moments and fix them.

Here is the journey ahead:

- Unity analysis tools such as the Profiler and Frame Debugger, so you can see exactly where time is being spent
- Static batching to reduce draw calls and send geometry to the GPU more efficiently
- Real-time and baked lighting workflows, including when each approach makes sense
- Level of Detail (LOD) systems that swap in simpler meshes for distant objects
- Occlusion culling to skip rendering geometry the player cannot see

By the end, you will not only know how to apply these techniques, but also understand why they work, which is what lets you make good optimization decisions in your own projects.

In the next chapter, we will dive right into optimization fundamentals and get started on this exciting journey.

Optimization Fundamentals

This chapter introduces the core ideas behind performance optimization in Unity. Before applying specific techniques, it helps to understand the most common causes of slowdowns and how to measure them using Unity's built-in analysis tools.

We will begin by looking at several scene and rendering factors that frequently affect frame rate. Then we will move into the Profiler, which is one of the most important tools for identifying where performance problems actually come from.

Common Performance Pitfalls

Optimization is easier when you know what to watch for early. In Unity, several issues appear repeatedly across projects: excessive polygon counts, expensive real-time lighting, too many draw calls, too many unique materials, and inefficient UI setup.

These do not always cause problems on their own. The issue is usually scale. A few high-detail meshes or a small amount of UI is rarely a concern, but once these costs accumulate across an entire scene, performance can drop quickly.

High Polygon Count

Polygon count is one of the first things to check when a scene runs slowly, because every visible triangle adds GPU work that scales with camera view.

The number of triangles in a mesh directly affects how much work the GPU must do each frame. Higher-resolution meshes can look better, but they also cost more to render. If too many detailed meshes are visible at once, frame rate can suffer.

There are two common ways to manage polygon count:

- Lower the mesh resolution so the total triangle count is reduced.
- Use a Level of Detail (LOD) system so objects switch to simpler meshes when they are farther from the camera.

An LOD system is especially useful because it preserves visual quality where it matters most. Objects close to the camera can remain detailed, while distant objects use cheaper versions that are less noticeable to the player.

Real-Time Lighting

Lighting is often the single most expensive visual feature in a scene, which makes it one of the highest-impact areas for optimization. Real-time lighting can have a major impact on performance, especially when shadows are enabled. Real-time lights require ongoing calculations while the game is running, and shadows add even more cost. In many scenes, shadow settings alone can make a dramatic difference to frame rate.

To reduce the cost of real-time lighting, consider the following approaches:

- **Disable shadows**: This often provides the largest single improvement. In some camera perspectives, such as top-down games, the visual loss may be minimal.
- **Lower the shadow distance**: Unity only renders shadows within a configurable distance from the camera. Reducing this value prevents distant shadows from consuming resources unnecessarily.
- **Use shadow cascades**: Cascades keep shadows sharper near the camera and lower resolution farther away, which helps balance quality and cost.
- **Explore different rendering paths**: Unity supports forward and deferred rendering, and each has different trade-offs depending on scene complexity and light count.

- **Use baked lighting**: Baked lighting calculates lighting in advance instead of every frame. This usually gives much better runtime performance, although it is not suitable for lighting that must change dynamically.

Batching and Draw Calls

Draw calls are one of the most important rendering costs to monitor in Unity. Each draw call is a command sent from the CPU to the GPU telling it to render something. If a scene generates too many of these commands, the CPU can become a bottleneck even before the GPU is fully stressed.

A high batch count usually means the scene is sending too many separate rendering commands. To reduce this, Unity provides several useful strategies:

- **Use static batching**: Marking suitable objects as static allows Unity to batch them more efficiently when they share compatible meshes and materials.
- **Reduce shadow-casting lights**: Lights that cast shadows often increase draw calls significantly, so limiting them can help.
- **Combine meshes**: If many small objects never move independently, combining them into a single mesh can reduce the number of draw calls.
- **Reduce the number of materials**: Reusing materials across similar objects gives Unity more opportunities to batch them together.

Unity Insight: How draw call batching works

Unity offers several batching strategies, each suited to different situations. Understanding the differences helps you choose the right one for your scene.

Static batching pre-combines meshes that share the same material and are marked as static. It trades additional memory for fewer draw calls at

runtime, making it ideal for environment geometry that never moves.

Dynamic batching combines small moving meshes at runtime. It has strict limits on vertex count and is often less impactful than static batching, but it can help in scenes with many small, simple objects.

SRP Batcher (available in URP) takes a different approach. Instead of combining meshes, it reduces the cost of switching between materials that use the same shader variant. This makes it effective even when objects use different material instances, as long as the underlying shader is compatible.

Learn more: Draw call batching (https://docs.unity3d.com/6000.3/Documentation/Manual/DrawCallBatching.html)

High Number of Materials

Reducing material count is one of the simplest ways to help Unity batch objects more efficiently. Material count is closely related to draw calls. The GPU works most efficiently when many objects can be rendered using the same material state. If a scene contains many unique materials, Unity has fewer opportunities to batch objects together, which increases rendering overhead.

A good rule is to share materials whenever possible. If several objects are visually similar, avoid creating separate materials unless there is a clear reason to do so. Combined with static batching, this can significantly reduce batch count and improve rendering performance.

UI Optimization

UI is easy to overlook during optimization because it usually starts lightweight, but it can quietly become a bottleneck as interfaces grow in complexity. Unity's UI system is often lightweight, but large or frequently changing interfaces can still become expensive. Two practical optimizations are especially useful in most projects: splitting UI across multiple canvases and disabling unnecessary raycasting.

The first optimization is to use multiple canvases. When any element on a canvas changes, Unity may need to rebuild that entire canvas. If one large canvas contains the HUD, menus, inventory, and other interface elements, even a small update can trigger unnecessary work. Splitting the UI into separate canvases helps isolate updates so only the relevant section is rebuilt.

The second optimization is to disable raycasting on non-interactive UI elements. Buttons and other clickable controls need raycasting, but decorative images and static text usually do not. Leaving raycasting enabled on many non-interactive elements adds avoidable overhead, especially in complex interfaces.

Taken together, these are some of the most common performance issues to watch for in Unity projects. In later chapters, several of these topics will be explored in more depth.

Using the Profiler

Knowing the common causes of performance problems is useful, but optimization should always be guided by data. Unity's Profiler helps you measure what your game is actually doing so you can identify bottlenecks instead of guessing.

The Profiler tracks performance in real time while your game runs. It can report CPU usage, rendering cost, memory consumption, physics activity, audio load, and more.

Opening the Profiler

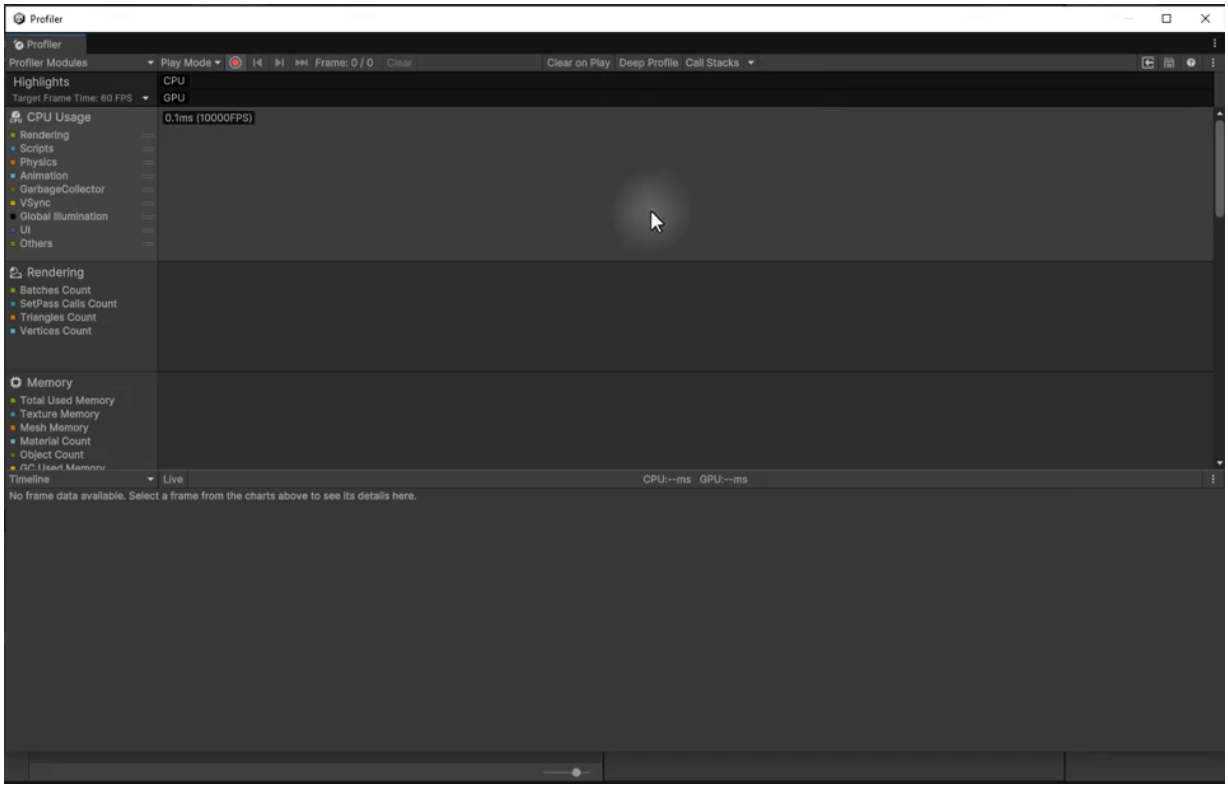
The first step is getting the Profiler window accessible in your layout so it is ready whenever you need to investigate a performance issue.

To open the Profiler, go to **Window > Analysis > Profiler**. This opens the Profiler window, which you can dock in your layout or leave floating depending on your workspace.

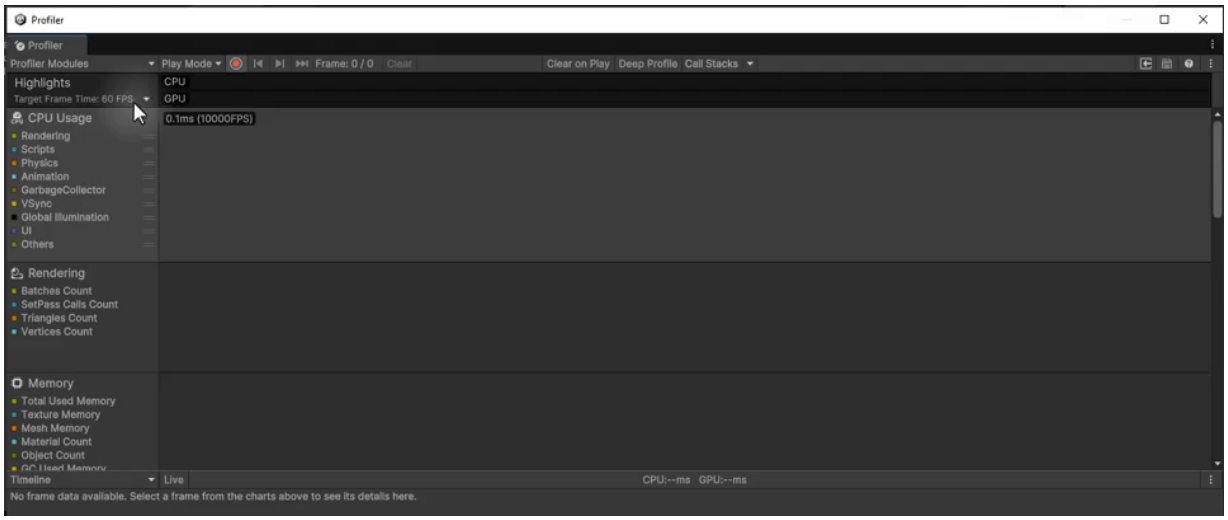
Understanding the Profiler Layout

Before capturing any data, it helps to understand what the Profiler is showing you so you can navigate it confidently during a real investigation.

The Profiler window is divided into two main areas. The upper section contains the available **Profiler Modules**, each of which tracks a different category of performance data over time. The lower section shows detailed information for whichever module is currently selected.



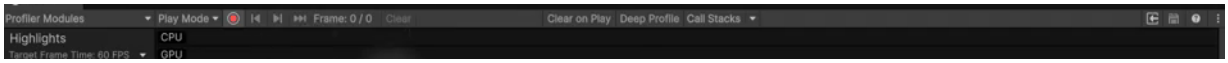
By default, Unity includes modules such as CPU Usage, Rendering, Memory, Audio, Video, and Physics. You can customize which modules are visible by clicking the **Profiler Modules** button in the top-left area of the window.



Starting a Profiling Session

Getting these settings right before you press Play ensures that the Profiler captures meaningful gameplay data from the start, rather than recording editor overhead or missing frames entirely.

Before entering Play mode, we need to verify two settings in the Profiler toolbar. First, make sure **Record** is enabled so the Profiler actually stores frame data as it arrives. When active, the button appears red. Second, confirm that **Play Mode** is selected so the Profiler captures data from the running game rather than focusing on editor activity.



Once those settings are correct, press **Play** in the Unity Editor. The Profiler will begin recording frame data immediately.

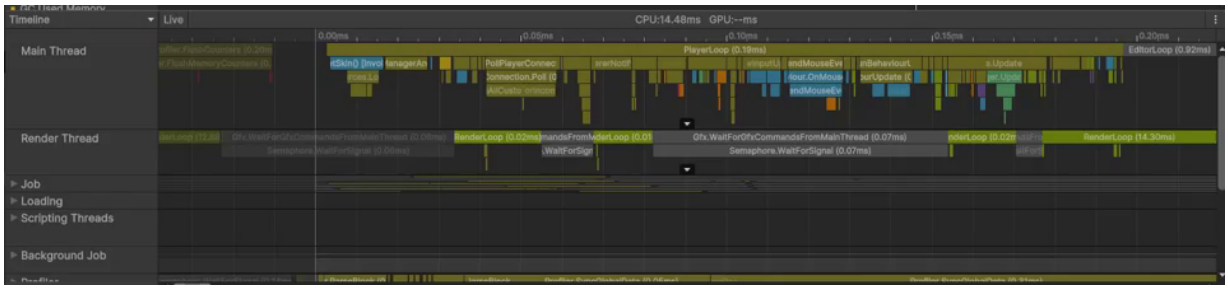
As your game runs, the graphs update in real time. You can then pause or stop the game and inspect individual frames to see what happened during specific moments.

To inspect a specific frame, click or drag across the timeline graphs. This lets you move through the recorded session and examine performance on a frame-by-frame basis.

Analyzing CPU Usage

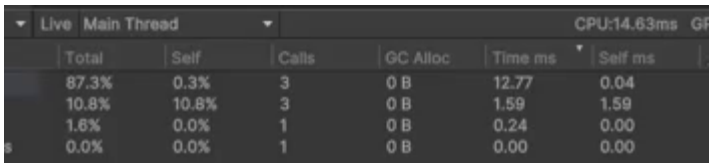
The CPU Usage module is one of the most useful places to start because it shows how much time the CPU spends on scripts, rendering preparation, physics, garbage collection, and other systems. If your frame time is too high, this module helps reveal where that time is going.

When you select a frame, the lower panel usually opens in **Timeline** view. This view shows activity across threads such as the Main Thread and Render Thread, with color-coded blocks representing different functions and systems.



Hovering over a block reveals how long that function took to execute. This is useful when you want to identify expensive calls or confirm whether a particular script event is actually significant.

If you want a more structured breakdown, switch the lower panel from **Timeline** to **Hierarchy** using the dropdown in the top-left corner of that panel. Hierarchy view groups calls under **PlayerLoop** and makes it easier to see which systems and scripts consumed the most time during the selected frame.



This view is especially helpful when tracking down expensive MonoBehaviour logic, because your own scripts appear alongside Unity's internal systems. It gives you a clearer sense of whether the problem comes from gameplay code, engine systems, or a combination of both.

Unity Insight: Custom profiler markers

The Profiler is not limited to Unity's built-in categories. You can add your own markers to any script using Profiler.BeginSample() and Profiler.EndSample(). When your code runs, these markers appear in the CPU Usage module alongside Unity's internal systems, making it much easier to identify which parts of your gameplay logic are consuming the most time.

This is especially useful when a MonoBehaviour contains several expensive operations in the same Update or FixedUpdate method. By wrapping each section in its own sample marker, you can see exactly where the cost is concentrated rather than treating the entire method as a single block.

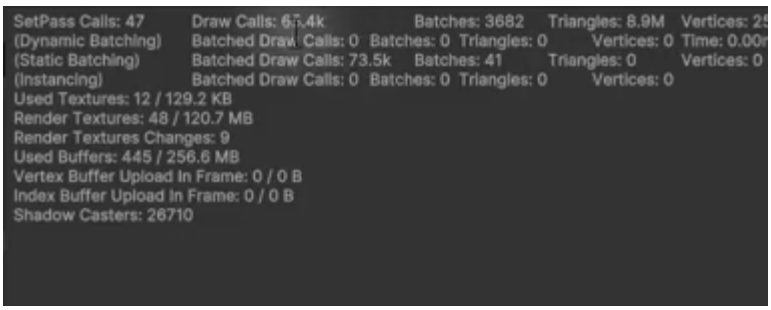
Learn more: Profiler (https://docs.unity3d.com/6000.3/Documentation/Manual/Profiler.html)

Optimizing Graphics with the Rendering Module

Once you can identify CPU-side costs, the next step is checking how much rendering work the GPU is handling. The Rendering module connects directly to the pitfalls discussed earlier in this chapter, such as polygon count, draw calls, and shadow casters.

If you suspect a rendering bottleneck, the Rendering module is the next place to look. This module reports how much geometry is being processed and how many rendering commands Unity is issuing each frame.

To check these metrics, select **Rendering** in the Profiler Modules list. This shows draw calls, batches, triangles, and vertices, along with how Unity is using batching methods such as dynamic batching, static batching, and instancing.



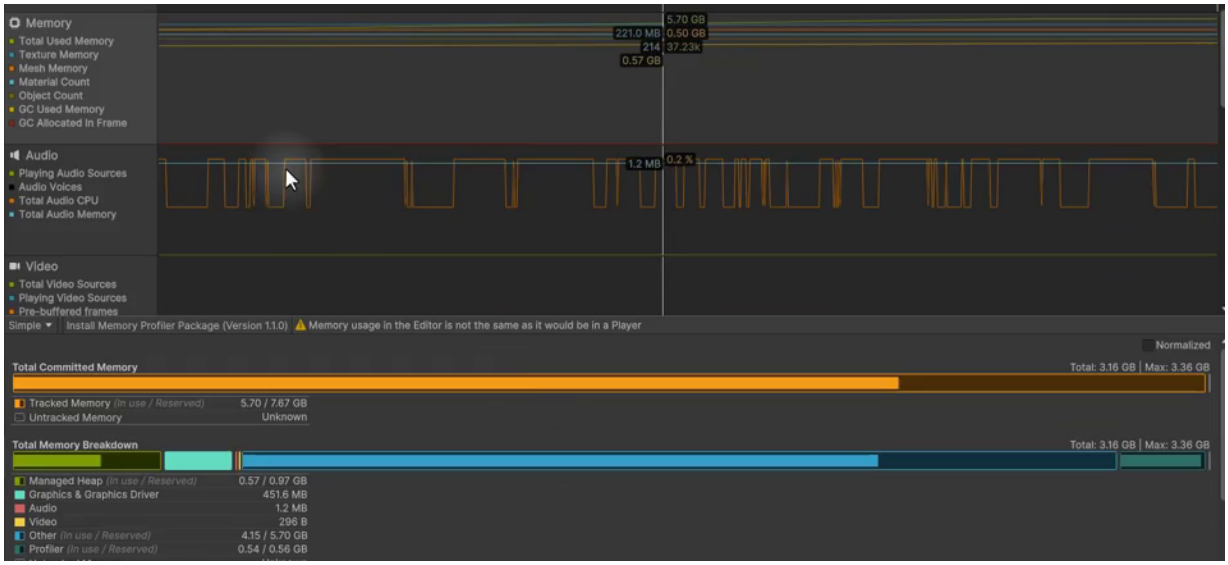
This module is particularly useful when evaluating the issues discussed earlier in the chapter, such as high polygon counts, excessive draw calls, or too many shadow casters. It helps connect scene setup decisions to measurable rendering cost.

Monitoring Memory Usage

Frame rate is only part of the picture. Some optimizations, such as static batching, trade memory for speed, so it is important to track memory alongside frame time to understand the full cost of your decisions.

Memory usage also matters especially in larger projects or on constrained hardware. The Memory module helps you see how much memory different parts of your project are consuming.

To investigate, select **Memory** in the Profiler to view a breakdown of allocations across categories such as textures, meshes, audio, graphics, and managed memory. This can help you identify whether a project is using more memory than expected and which systems are responsible.



One important detail is that the Profiler itself adds overhead. It uses memory and processing power while recording, so the numbers you see include some profiling cost. In the Editor, this overhead can be noticeable. Unity also warns that memory usage in the Editor is not the same as memory usage in a built player, so keep that in mind when interpreting results.

Exploring Additional Modules

The Profiler includes several other modules that are useful depending on the type of project you are building.

Some of the most relevant are:

- **Audio**: Useful when your project relies heavily on Unity's audio systems, including Audio Sources and listeners.
- **Video**: Relevant if your game includes video playback.
- **Physics**: Especially valuable in physics-heavy projects, where dynamic bodies and collision interactions may be consuming more time than expected.

These modules may not be central in every project, but they are worth checking when a specific system seems suspicious.

Summary

This chapter covered two essential parts of optimization work in Unity. First, you looked at common performance pitfalls such as high polygon counts, expensive lighting, excessive draw calls, too many materials, and inefficient UI setup. Second, you explored the Profiler, which provides the data needed to investigate those issues properly.

The practical takeaway is a workflow you can use throughout the rest of this book and in your own projects: identify a suspected bottleneck from the common pitfalls, measure it with the appropriate Profiler module, and then apply a targeted fix. The chapters ahead will follow this same pattern as they cover specific optimization techniques in greater depth.

Rendering Analysis

This chapter focuses on two practical rendering topics in Unity: the **Frame Debugger** and **static batching**. Together, they help you understand how Unity builds each frame and how to reduce the rendering overhead of large scenes.

The first half of the chapter examines the Frame Debugger, which lets you step through the rendering pipeline event by event. The second half applies that understanding to a common optimization technique: batching large numbers of stationary objects so the CPU sends fewer rendering commands to the GPU.

Using the Frame Debugger

Many rendering performance problems are difficult to solve without seeing what Unity is actually doing behind the scenes each frame. The Frame Debugger gives you that visibility by breaking down the rendering pipeline into individual steps you can inspect one at a time. Understanding this sequence is essential for optimization because it reveals exactly where rendering time is being spent and which features are adding cost.

The Frame Debugger shows the exact sequence of rendering events that produce the final image, which makes it valuable for diagnosing performance issues, checking render order, and understanding how the active render pipeline behaves.

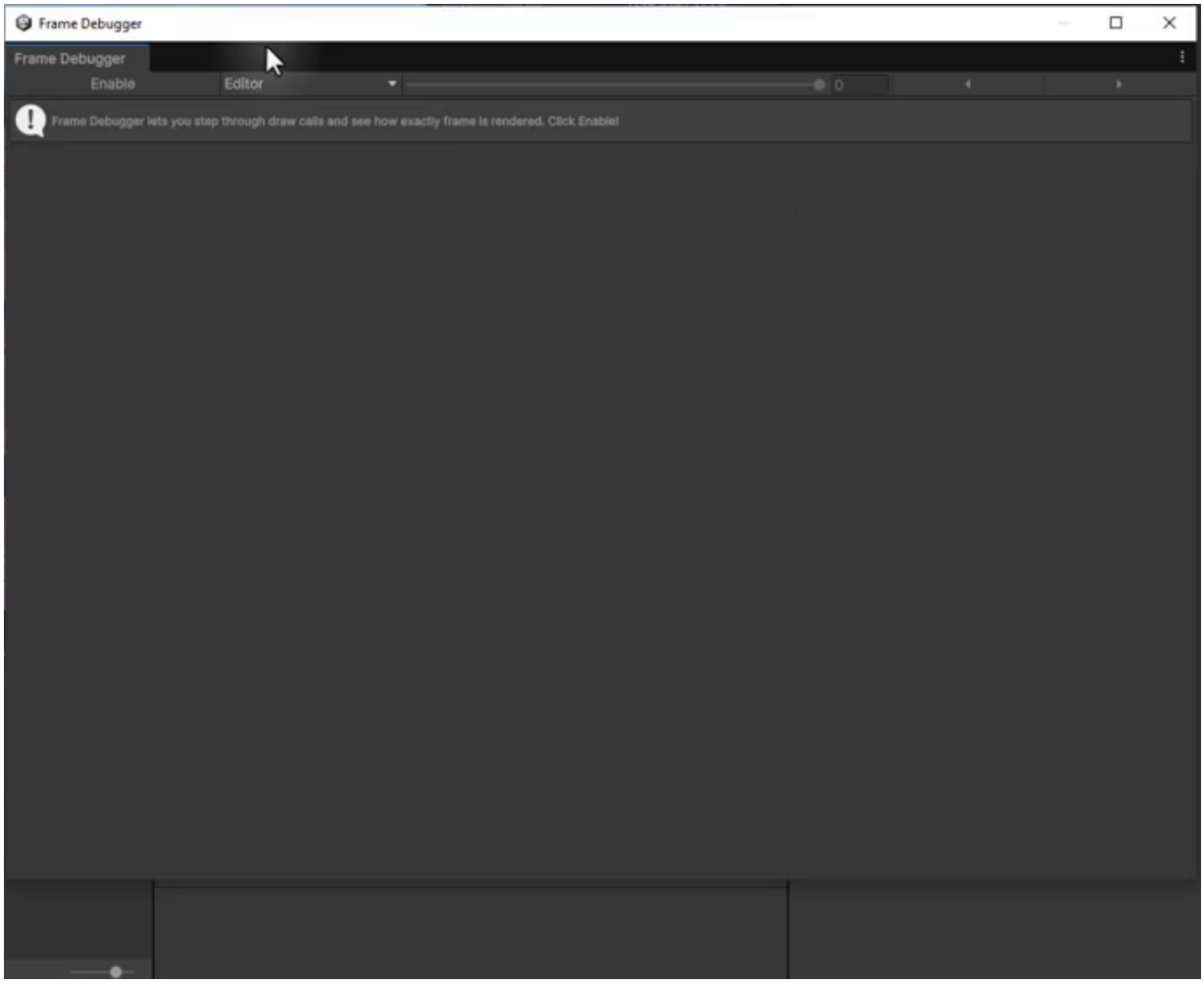
Opening the Frame Debugger

Before you can inspect rendering behavior, you need to open the Frame Debugger and capture a frame.

To open the Frame Debugger, go to **Window > Analysis > Frame Debugger**. This opens a dockable editor window that can be placed

anywhere in your layout.

With the window open, click **Enable** to capture the current frame. This freezes the rendering state so you can step through it without the scene continuing to update. Unity will then list everything being generated and sent to the GPU for that frame. You can also use the Frame Debugger during Play mode. In that case, pause the game first, then enable the debugger so you can inspect the frame at that exact moment.

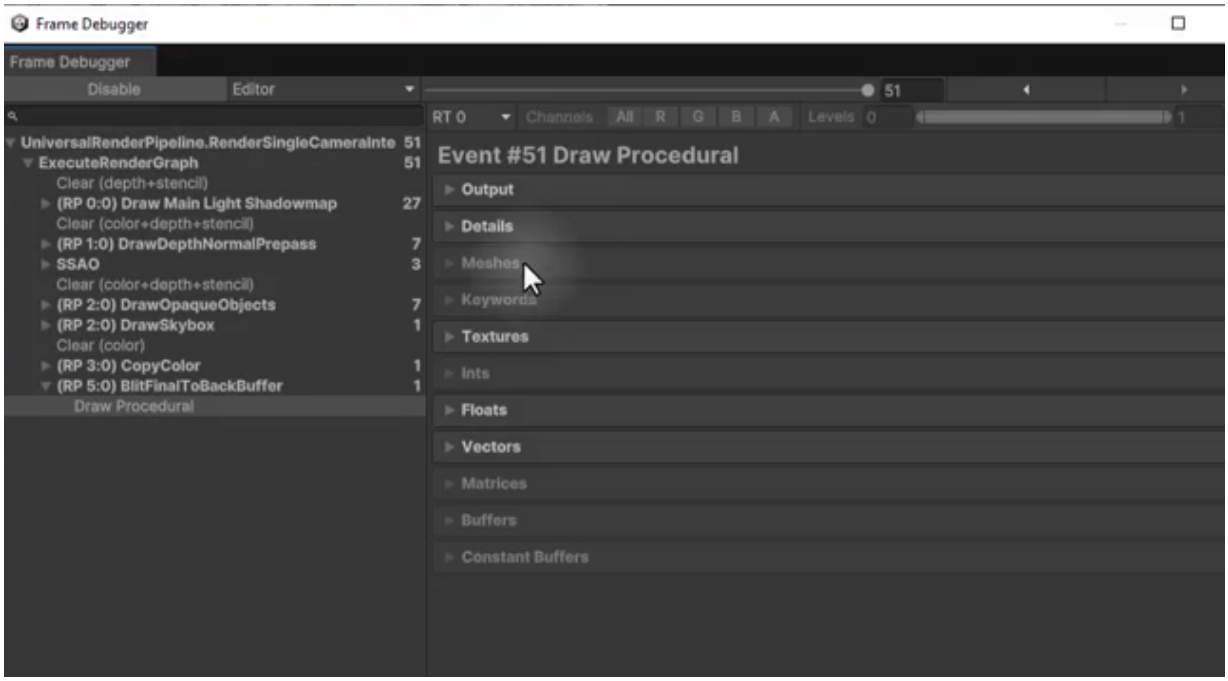


Understanding the Event List

Once enabled, the Frame Debugger displays the full order of rendering events for the selected frame. At the top of the window, Unity shows the

total number of events required to render the scene. In a URP scene, for example, this might be around 51 events.

You can step through these events one at a time to see how the final image is assembled. This is especially useful because rendering is not a single action. Unity performs a sequence of passes, intermediate texture operations, and draw calls before the frame is complete.

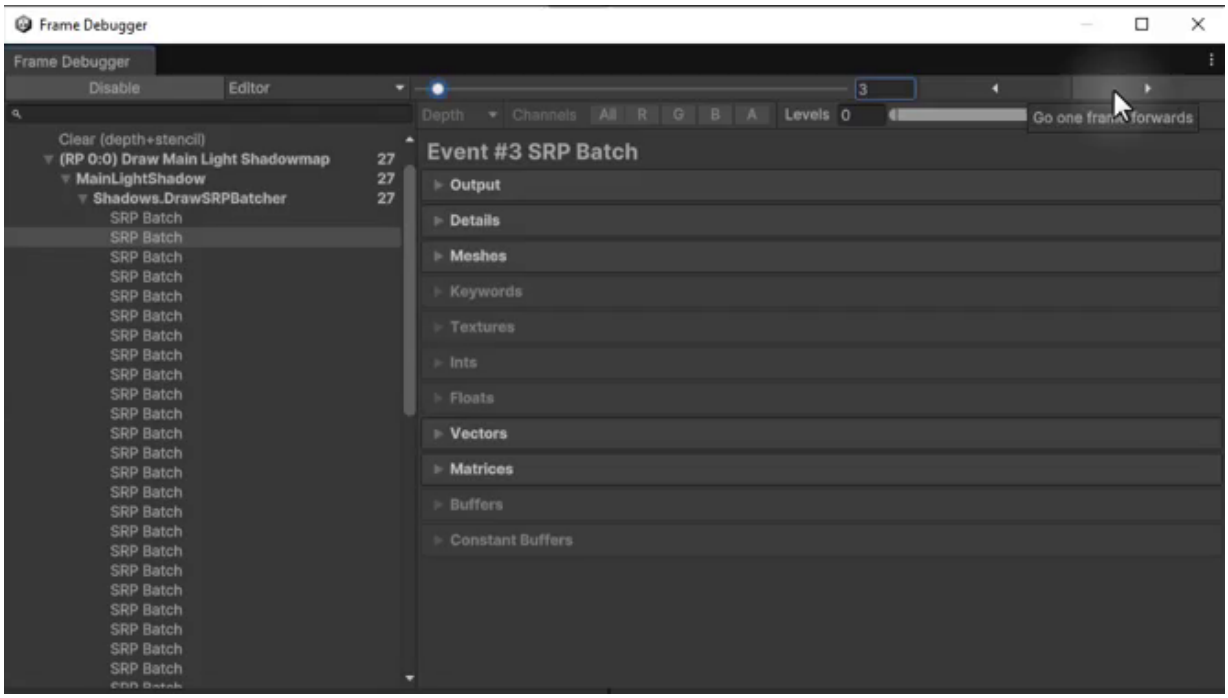


The exact event structure depends on the render pipeline in use. Unity's built-in pipeline, URP, and HDRP all produce different sequences. The examples in this chapter use **URP**, so the event names and ordering reflect that pipeline.

Shadow Maps, Depth Textures, and Early Passes

The first part of the rendering process often includes shadow map generation. For example, the main directional light may render a shadow map that will later be sampled when Unity draws shadows onto scene geometry.

While stepping through these early events, the Game view may look incomplete or unusually dark. That is expected. At this stage, Unity is still preparing data rather than drawing the final lit scene.



After shadow maps are created, Unity typically builds a **depth texture**. If **Screen Space Ambient Occlusion (SSAO)** is enabled, Unity then generates the data needed for that effect before moving on to the main scene geometry.

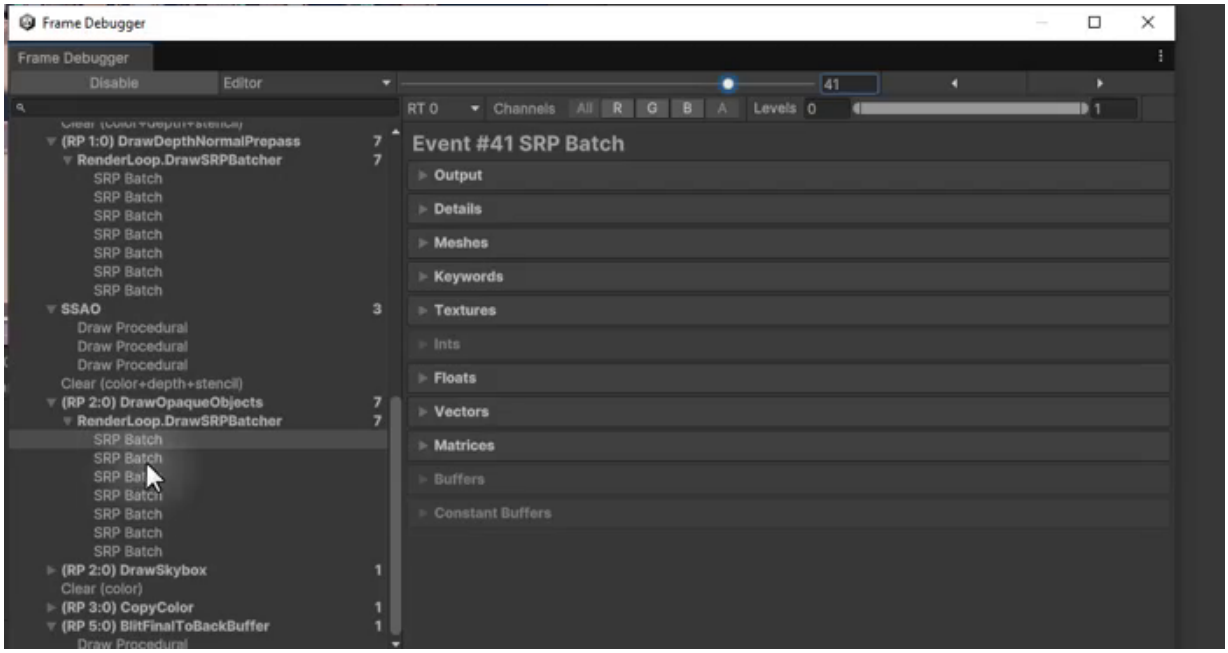
This sequence matters because each additional feature adds more rendering work. The Frame Debugger makes those extra steps visible, which helps explain why certain visual features increase frame cost.

SRP Batching in the Frame Debugger

When using the Scriptable Render Pipeline, Unity enables **SRP batching** by default. This optimization reduces rendering overhead by grouping compatible objects together so Unity can submit them more efficiently.

Without batching, many objects would require separate draw calls. With SRP batching, a large number of similar objects can be processed in a much

smaller number of batches. This does not eliminate rendering cost entirely, but it can reduce CPU-side overhead significantly.



The Frame Debugger makes this behavior easy to inspect. One batch may contain generic environment models, another may contain roads, and others may contain groups of buildings. As you step through the event list, you can see how Unity draws opaque geometry first, then transparent objects, then the skybox, followed by any remaining passes needed to complete the frame.

Unity Insight: How the SRP Batcher reduces CPU overhead

Traditional batching works by combining meshes into a single draw call. The SRP Batcher takes a different approach: it keeps individual draw calls but makes them cheaper to execute. It does this by caching material properties in GPU memory so Unity does not need to re-upload them every frame.

When two objects use the same shader variant, the SRP Batcher can render them back-to-back without expensive state changes, even if their material properties (colors, textures) differ. The key requirement is shader compatibility, not material identity.

In the Frame Debugger, SRP-batched draw calls appear as “SRP Batch” events. If you see many separate batches where you expected fewer, check whether the objects use incompatible shader variants, which prevents the batcher from grouping them efficiently.

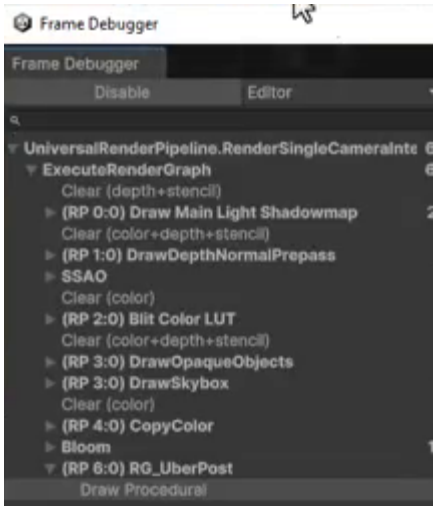
Learn more: [SRP Batcher](https://docs.unity3d.com/6000.3/Documentation/Manual/SRPBatcher.html) (https://docs.unity3d.com/6000.3/Documentation/Manual/SRPBatcher.html)

Analyzing Post-Processing Effects

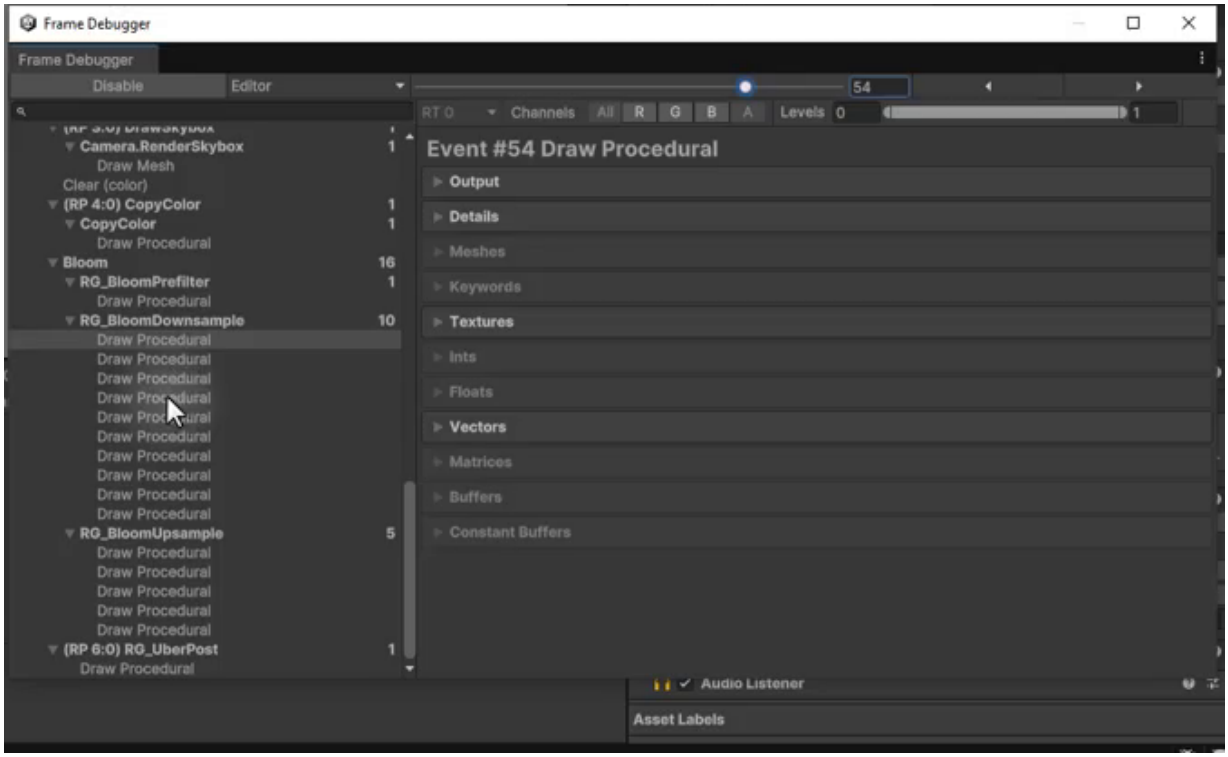
Post-processing effects can add a surprising number of rendering passes, so it is worth understanding their cost before committing to them in a project.

The Frame Debugger is especially useful when you want to understand the cost of post-processing. Enabling post-processing on a camera often increases the number of rendering events noticeably.

A scene that takes around 51 events without post-processing might rise to roughly 70 events once effects such as bloom are enabled. The debugger shows exactly where those extra passes appear in the pipeline.



Bloom is a good example because it is built from multiple passes rather than a single operation. Unity creates the glow effect by repeatedly downsampling and then upsampling the image, applying several processing steps along the way. In the Frame Debugger, each of those steps appears as its own event, making the cost much easier to see and reason about.



Inspecting Event Details

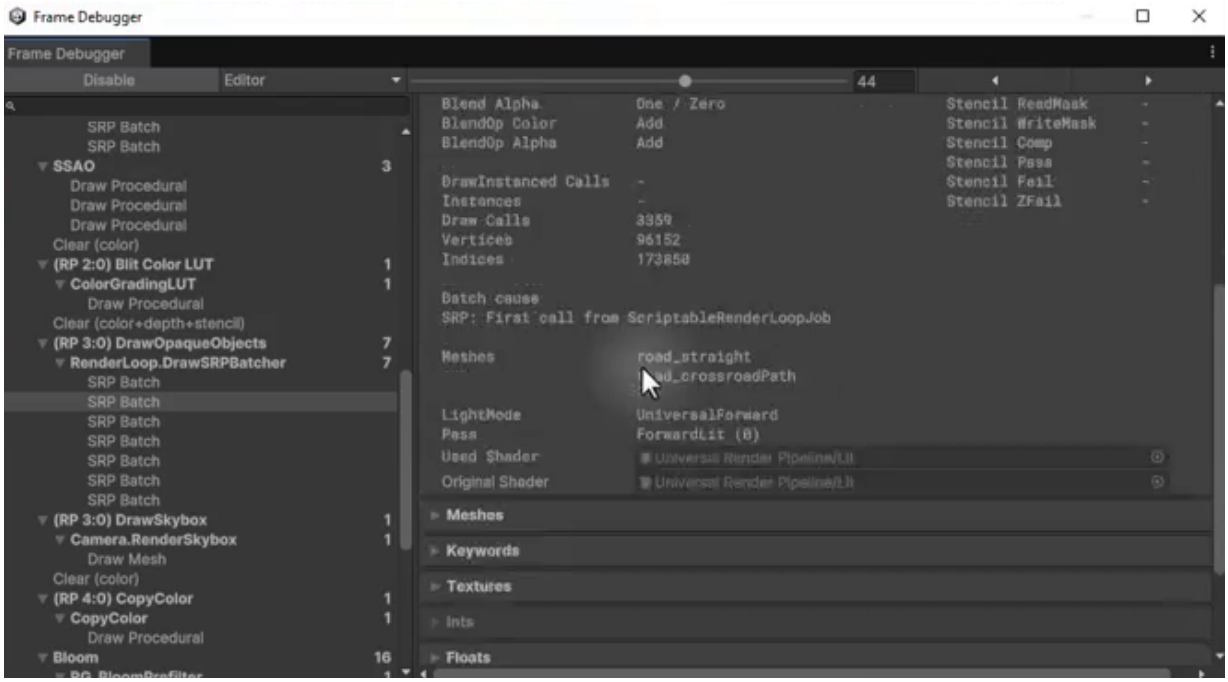
Knowing that an event exists is useful, but sometimes you need to see exactly what data Unity is sending to the GPU in order to diagnose why a batch is larger or more expensive than expected.

Beyond the event list itself, the Frame Debugger also lets you inspect the data associated with each rendering event. To do so, select an event and open **Details** to see information about what Unity is sending to the GPU.

This can include:

- Vertex counts

- Mesh names
- Shader-related values such as floats, vectors, and matrices
- Render state information for the selected draw call or batch



For example, a single SRP batch may contain close to 100,000 vertices. The **Meshes** section shows which models are included in that batch, such as road pieces or building geometry. The lower-level shader data is also available when needed, although many projects will not require that level of inspection.

Practical Uses of the Frame Debugger

In day-to-day development, you will not always need to step through every rendering event. Still, the Frame Debugger is extremely useful in several situations:

- **Performance optimization**: Identify expensive rendering steps and measure the cost of specific effects.

- **Render order debugging**: Confirm that objects are drawn in the correct sequence and detect cases where something renders too early or too late.
- **Effect analysis**: Understand how features such as bloom and SSAO add work to the pipeline.
- **Shader development**: Inspect how custom shaders appear in the rendering sequence and what data they receive.
- **Custom SRP work**: Analyze and verify rendering behavior when building or modifying a Scriptable Render Pipeline.

The key benefit is visibility. Instead of treating rendering as a black box, the Frame Debugger lets you inspect the frame as Unity constructs it.

Optimizing Scenes with Static Batching

In the previous chapter, you learned that high batch counts are one of the most common rendering bottlenecks and that static batching is one of the strategies Unity provides to address them. Now that you have seen how the Frame Debugger reveals individual batches and draw calls, you can apply that understanding to a concrete optimization. Static batching is a common technique for environments made of many stationary objects, such as roads, buildings, and background set pieces.

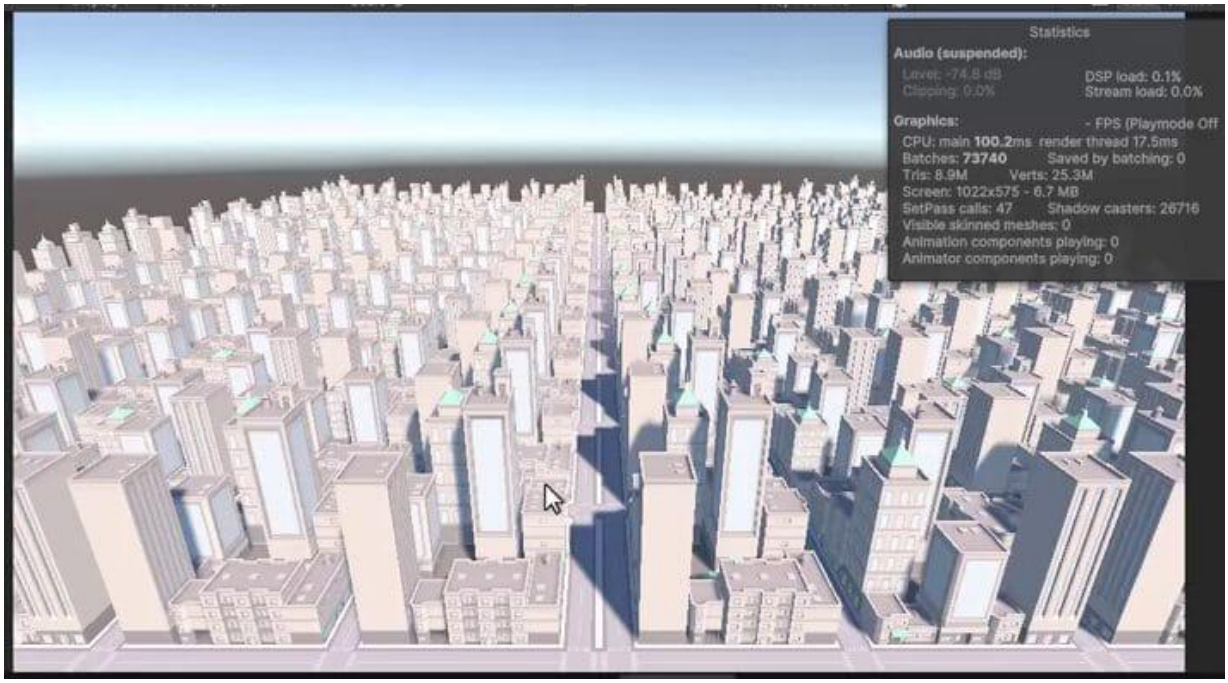
What Static Batching Does

Static batching reduces the number of batches Unity needs to submit by grouping compatible, non-moving objects together. This is most effective in scenes where many objects share meshes and materials and do not need to move during gameplay.

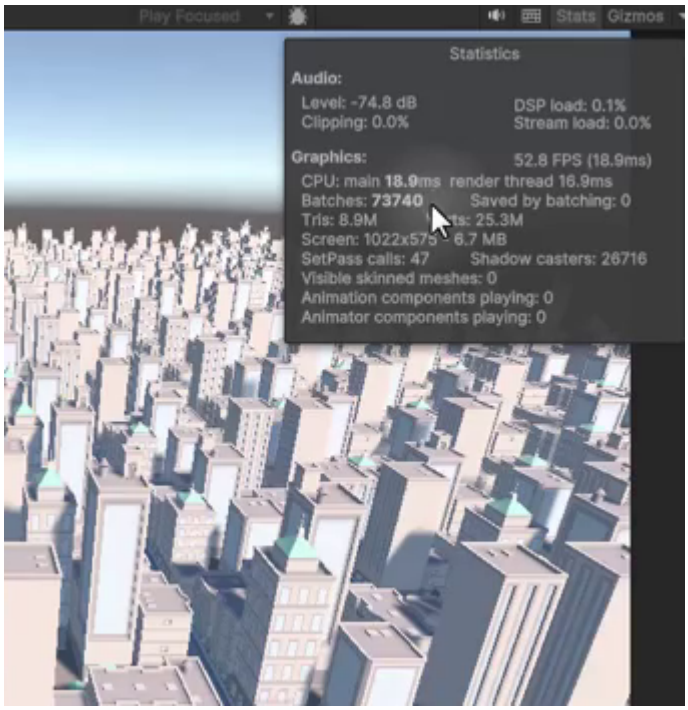
A large city scene is a good example. If every building and road segment is rendered individually, the CPU must issue a very large number of draw calls. Static batching helps reduce that overhead by consolidating those objects into fewer rendering batches.

Measuring the Scene before Optimization

Establishing a baseline before making changes is essential because it lets you measure the actual impact of an optimization rather than guessing whether it helped. In this example, the scene is a large city environment filled with repeated building types, roads, and other stationary geometry.

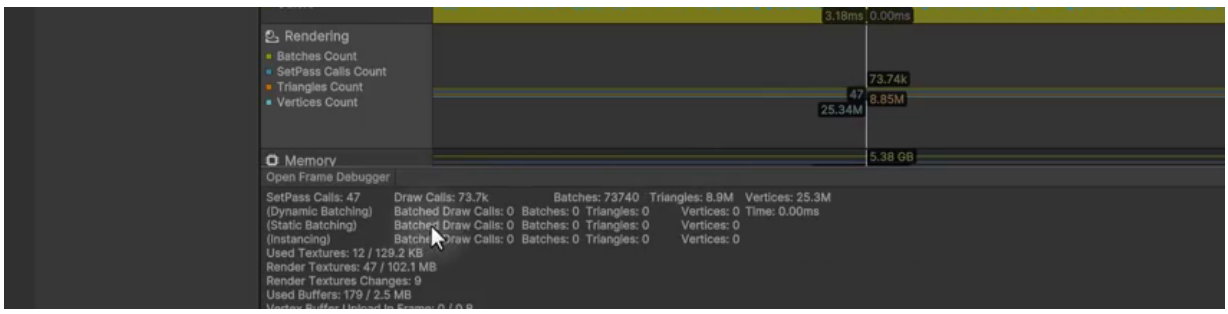


When the scene runs in Play mode, performance is poor. Frame rate drops to around 50 FPS, and the batch count rises above 70,000. That number is a strong sign that the scene is not being rendered efficiently, because Unity is issuing an enormous number of separate rendering commands.

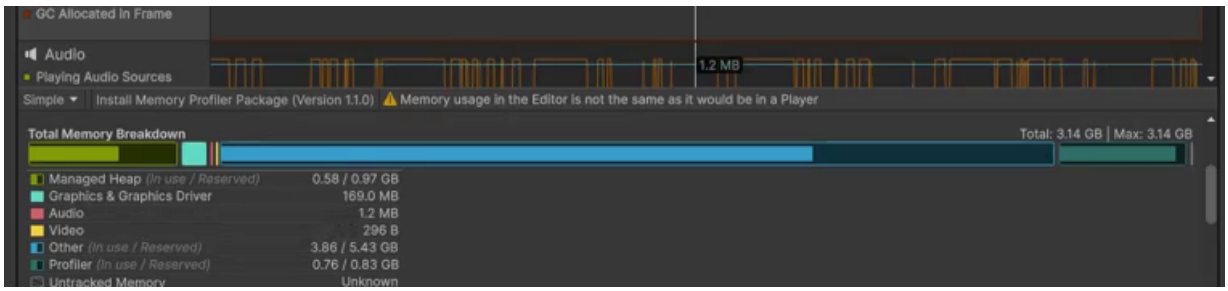


Confirming the Bottleneck in the Profiler

The Statistics overlay is useful for a quick glance, but the Profiler gives you the detailed breakdown needed to confirm exactly where the problem lies before you commit to an optimization strategy. In the **Rendering** module, the scene shows more than 73,000 draw calls and the same number of batches. This confirms that draw call overhead is a major bottleneck.



It is also useful to note the memory baseline before making changes. In the **Memory** module, the Graphics and Graphics Driver memory is roughly 170 MB. This matters because static batching improves rendering performance at the cost of additional video memory.



Enabling Static Batching

Unity cannot automatically determine which objects are stationary, so you need to mark them explicitly. This tells Unity it is safe to pre-combine their mesh data for more efficient rendering.

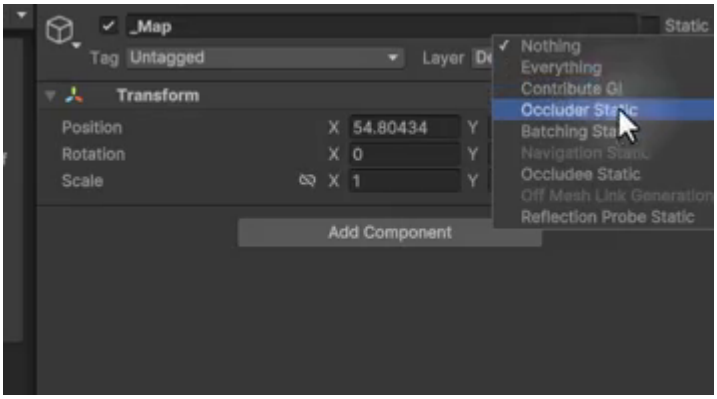
In this case, the entire environment map is selected because the buildings and roads are not expected to move. Follow these steps:

1. In the Hierarchy, select the models that should be included in static batching. Selecting a parent object lets you batch an entire group at once.

2. In the Inspector, find the **Static** checkbox near the top-right area and open the dropdown next to it. Unity separates static flags so you can enable only the optimizations that apply to each object.

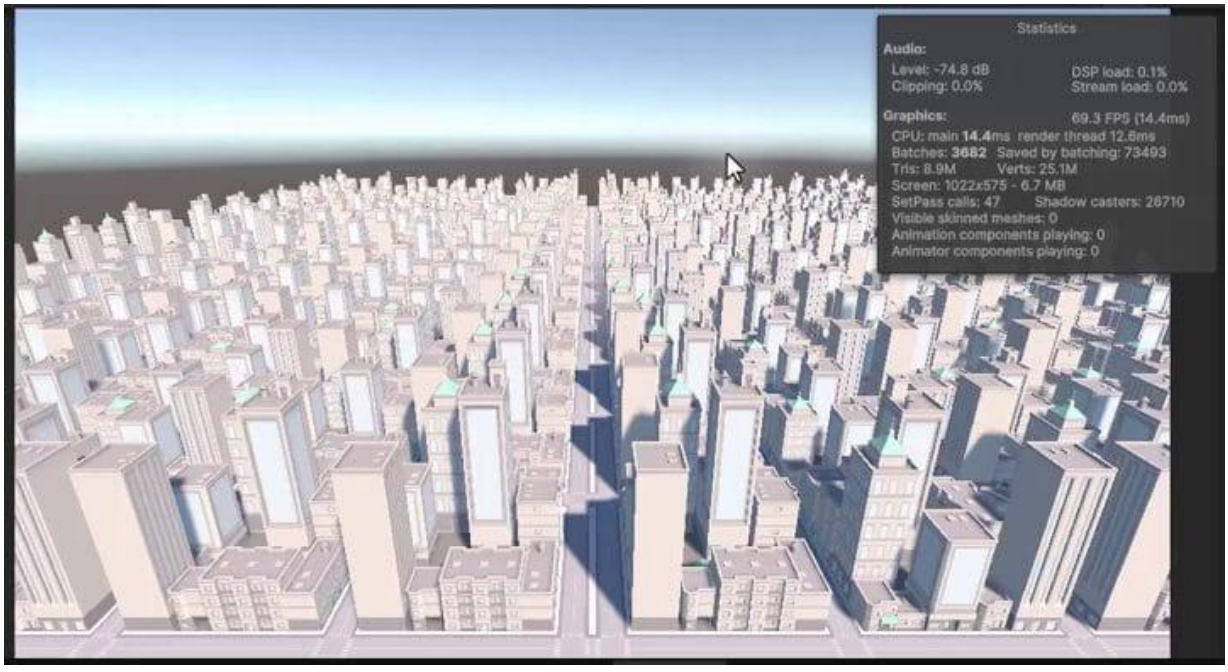
3. Choose **Batching Static** if you only want batching, or choose **Static** to enable all static-related flags.

4. When Unity asks whether to apply the change to child objects, click **Yes, change children**. This ensures that every mesh within the selected hierarchy is included in the batch.



Observing the Results

After enabling static batching and running the scene again, the improvement is immediate. The batch count drops from roughly 73,000 to around 3,000, and frame rate increases by about 20 FPS. Visually, the scene remains the same, which is why batching is such a useful optimization when it applies.



Reviewing the Profiler after Batching

The Profiler confirms what the Statistics overlay suggests. In the **Rendering** module, Unity now reports that tens of thousands of draw calls have been consolidated through static batching. The important result is not just the raw draw call number, but the fact that the scene is now being submitted in far fewer batches.

This is the core benefit of static batching: the CPU spends less time issuing individual rendering commands, which improves frame time in scenes with many stationary objects.

Monitoring the Memory Trade-off

Static batching is not free. The combined mesh data must be stored, which increases graphics memory usage. After enabling batching, the Graphics and Graphics Driver memory rises from roughly 170 MB to around 420 MB.

In many projects, this trade-off is acceptable. Modern GPUs usually have enough video memory that the increase is manageable, especially when the performance gain is substantial. Even so, you should always verify this against your target hardware rather than assuming the cost is harmless.

Unity Insight: Why static batching increases memory

When static batching is enabled, Unity creates a single combined vertex buffer for all batched objects at build time. Even if 500 buildings share the same mesh, the combined buffer stores each building's vertex data separately because each object occupies a different position in world space. The original shared mesh data remains in memory alongside the new combined buffer.

This is why memory usage can increase significantly in large scenes. The rendering improvement comes from reducing CPU-side draw call overhead, but the cost is paid in GPU memory. On platforms with limited video memory (mobile, older hardware), it is worth checking the Profiler's Memory module after enabling batching to confirm the increase is within budget.

Learn more: Draw call batching (https://docs.unity3d.com/6000.3/Documentation/Manual/DrawCallBatching.html)

When to use Static Batching

Static batching works best when the scene contains many objects that meet two conditions: they share compatible rendering data, and they do not move at runtime.

Keep the following guidelines in mind:

- Objects marked as static should not move during gameplay.
- Do not apply static batching to dynamic objects such as the player, enemies, or moving props.
- Use it primarily for environment geometry such as buildings, roads, obstacles, and background set pieces.
- Always check memory usage after enabling it, especially in large scenes.

Summary

This chapter covered two important parts of rendering analysis in Unity.

First, you explored the **Frame Debugger**, which reveals the exact sequence of rendering events used to build a frame. It is useful for understanding render order, inspecting batches, analyzing post-processing, and diagnosing rendering issues at a low level. When you encounter a rendering bottleneck in your own projects, the Frame Debugger should be one of the first tools you reach for.

Second, you applied that understanding to **static batching**, a practical optimization for scenes with many stationary objects. By reducing the number of batches Unity must submit, static batching can improve performance dramatically, although it does increase graphics memory usage. The key workflow to remember is: measure with the Profiler, apply the optimization, then measure again to confirm the improvement and check for trade-offs like increased memory.

In the next chapter, the focus will shift to another major rendering cost: lighting.

Lighting Performance

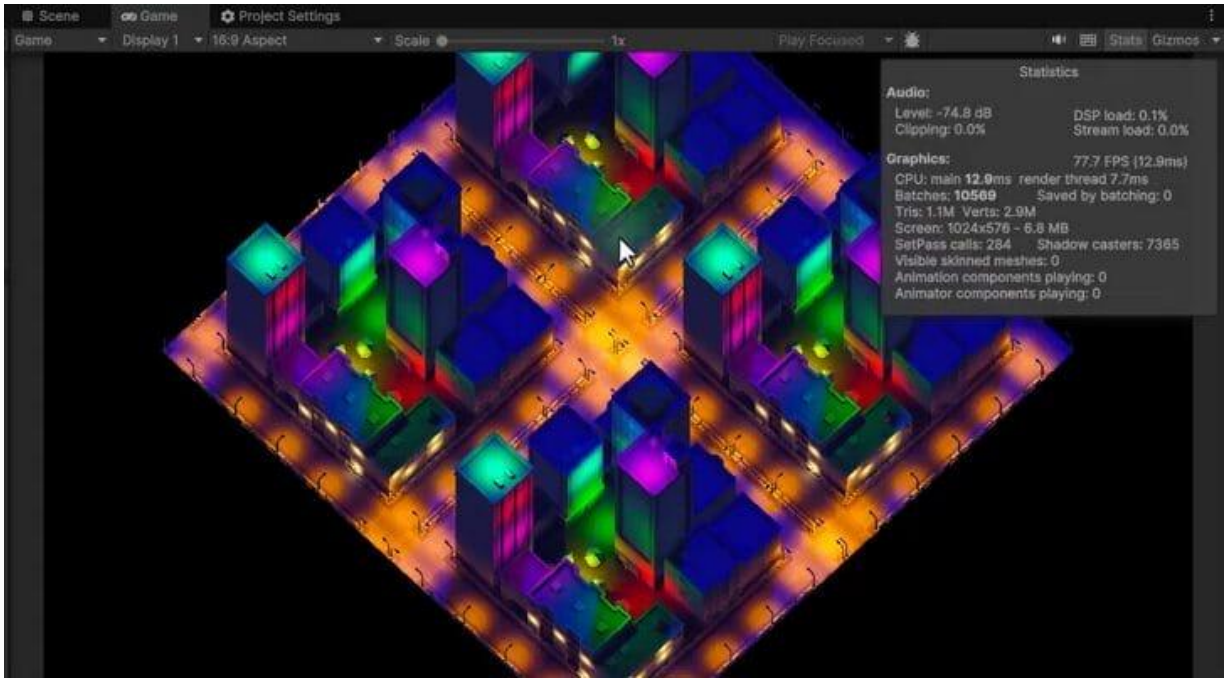
Lighting is one of the most visually important parts of a Unity scene, but it is also one of the easiest ways to lose performance. In this chapter, we will look at two sides of that problem: how to optimize **real-time lighting** when lights must remain dynamic, and how to use **baked lighting** when the scene can be precomputed ahead of time.

The examples in this chapter use the Universal Render Pipeline (URP) and focus on practical decisions you can apply immediately. We will begin with real-time lighting limits and shadow costs, then move into a full baked lighting workflow and the settings that matter most.

Optimizing Real-Time Lighting

Real-time lights are expensive because Unity must evaluate them continuously while the game runs. That cost increases quickly when many lights are visible at once, especially if they also cast shadows.

In this example, the scene is a small low-poly isometric city block with roughly one hundred real-time lights. The geometry itself is simple, but the number of active lights and shadow casters creates a heavy rendering load.

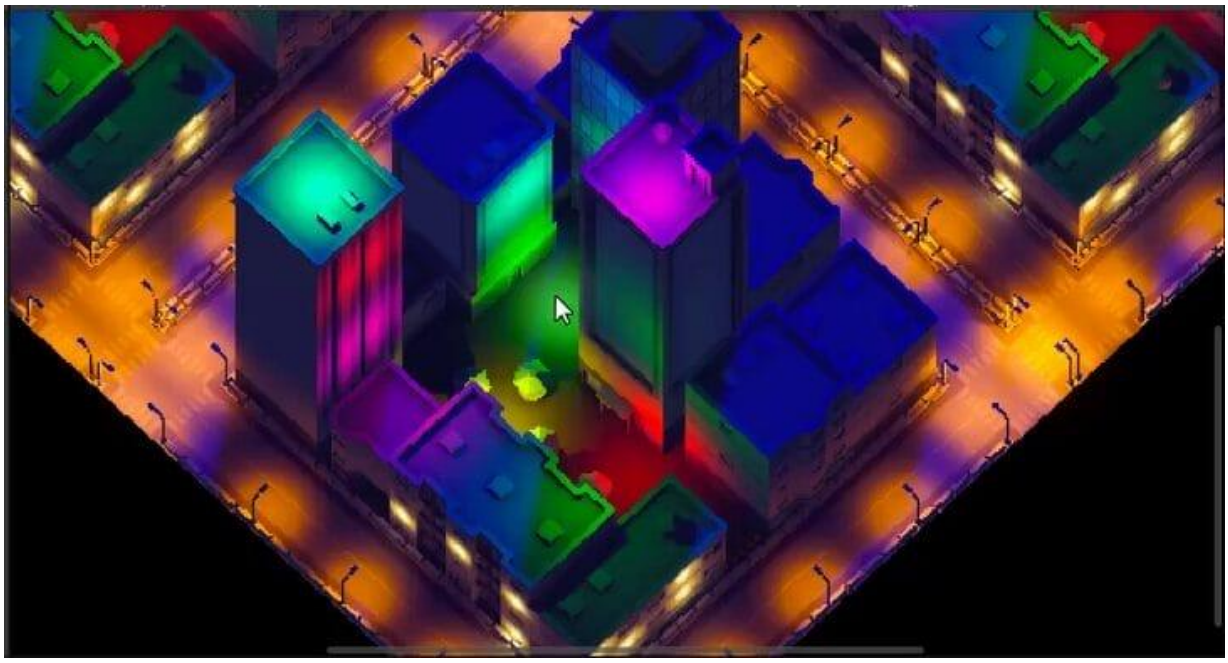


Even before measuring frame rate, scenes like this often reveal a warning sign: some lights appear to stop rendering entirely. That behavior is not random. It comes from Unity's handling of visible real-time lights.

Understanding the Real-Time Light Cap

Unity's default rendering behavior places a limit on how many real-time lights can affect the scene at once. When too many lights are visible to the camera, Unity prioritizes some of them and culls others.

As a result, you may notice that rooftop accent lights disappear, or that distant streetlights stop illuminating the environment even though the meshes themselves remain visible. As the camera moves, the set of rendered lights changes based on what Unity considers most important at that moment.



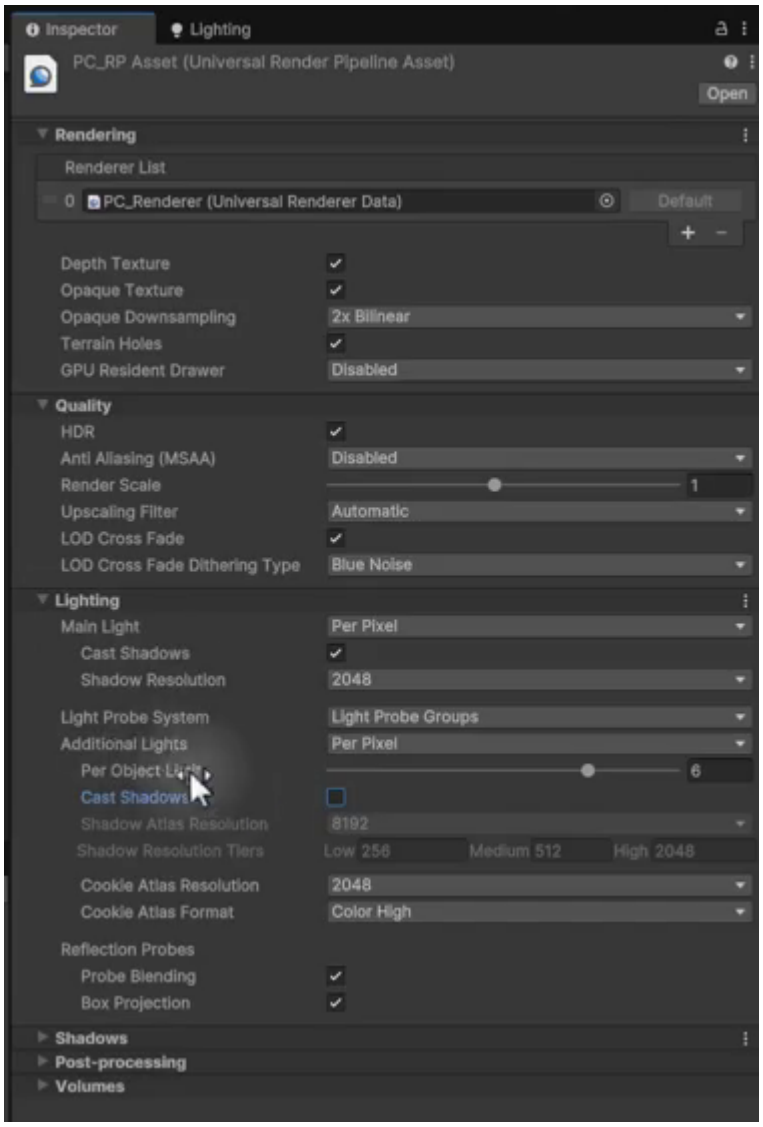
This cap exists to protect performance. Rendering hundreds of simultaneous real-time lights, especially with shadows, would be prohibitively expensive in most projects. The practical takeaway is simple: if a scene depends on a large number of visible dynamic lights, you need to manage them carefully or choose a rendering path that better fits the workload.

Disabling Shadows on Additional Lights

In most scenes, shadows are the single most expensive part of real-time lighting. Every shadow-casting light adds extra rendering passes, shadow map generation, and additional shading work.

One of the most effective optimizations is to disable shadows for **Additional Lights** globally. This removes shadow casting from non-directional lights such as point lights and spotlights, while still allowing the main directional light to keep its shadows.

To do this, select your **URP Render Pipeline Asset** in the Project window. In the Inspector, open the **Lighting** section, where you will find the **Cast Shadows** option under **Additional Lights**.

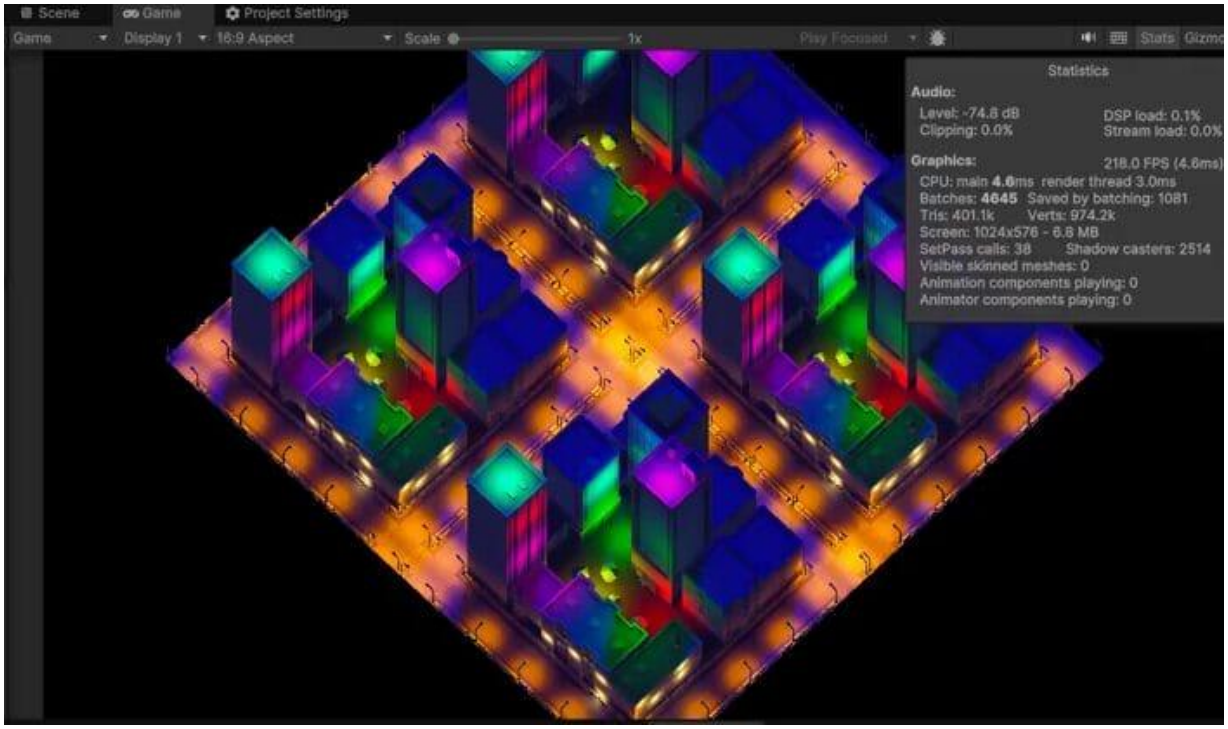


This setting is especially useful in top-down and isometric games. Because the camera is often far from the world, the visual difference between decorative lights casting shadows and not casting shadows is usually minor. The performance difference, however, can be substantial.

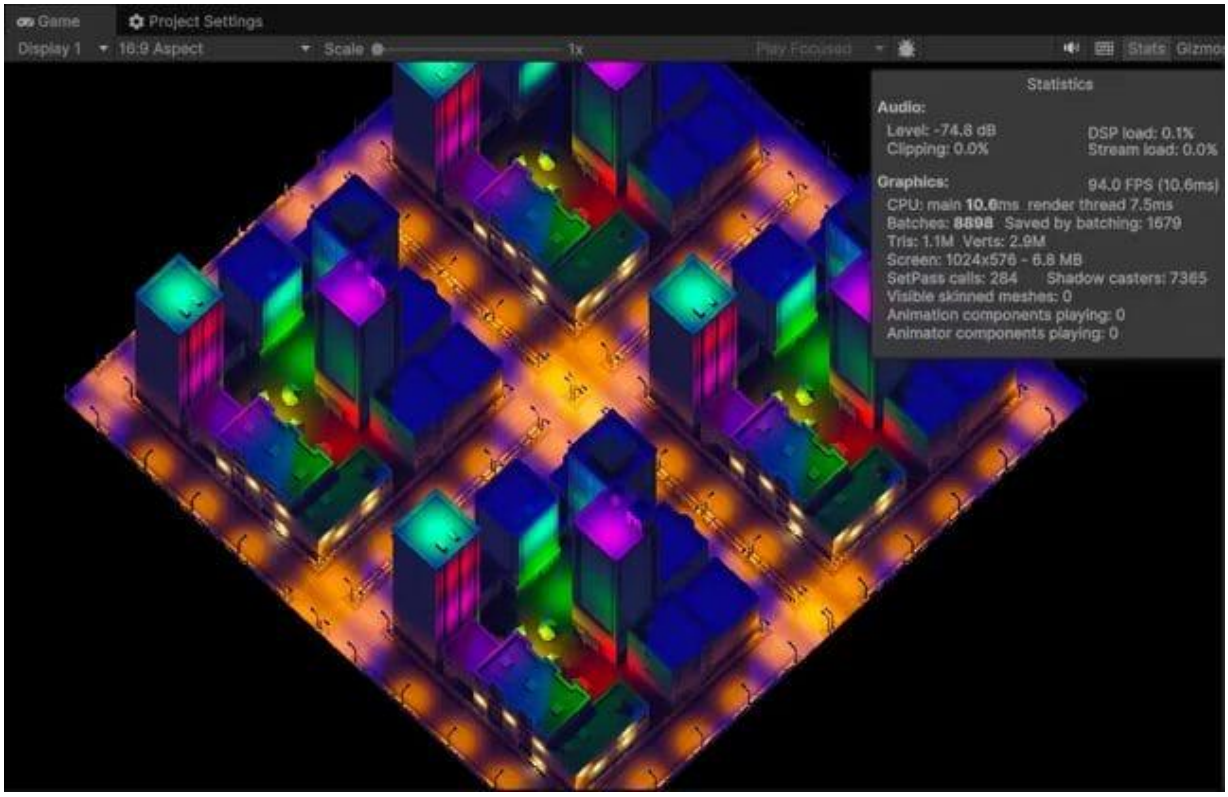
Comparing Performance with Shadows On and Off

It is always helpful to compare before and after results rather than assuming an optimization is worthwhile. In this scene, disabling shadows on additional lights produces a major improvement.

With additional light shadows disabled, the scene runs at roughly **218 FPS**.



With those same shadows enabled again, performance drops to roughly **94 FPS**.



That is close to a 50 percent reduction in frame rate from shadows alone. In scenes with many lights, this pattern is common.

A useful middle ground is to reserve shadows for lights that matter most to the player. For example, lights near the character or near important gameplay spaces may justify the cost, while decorative facade lights, rooftop accents, and distant streetlights usually do not.

Using SSAO to Restore Depth

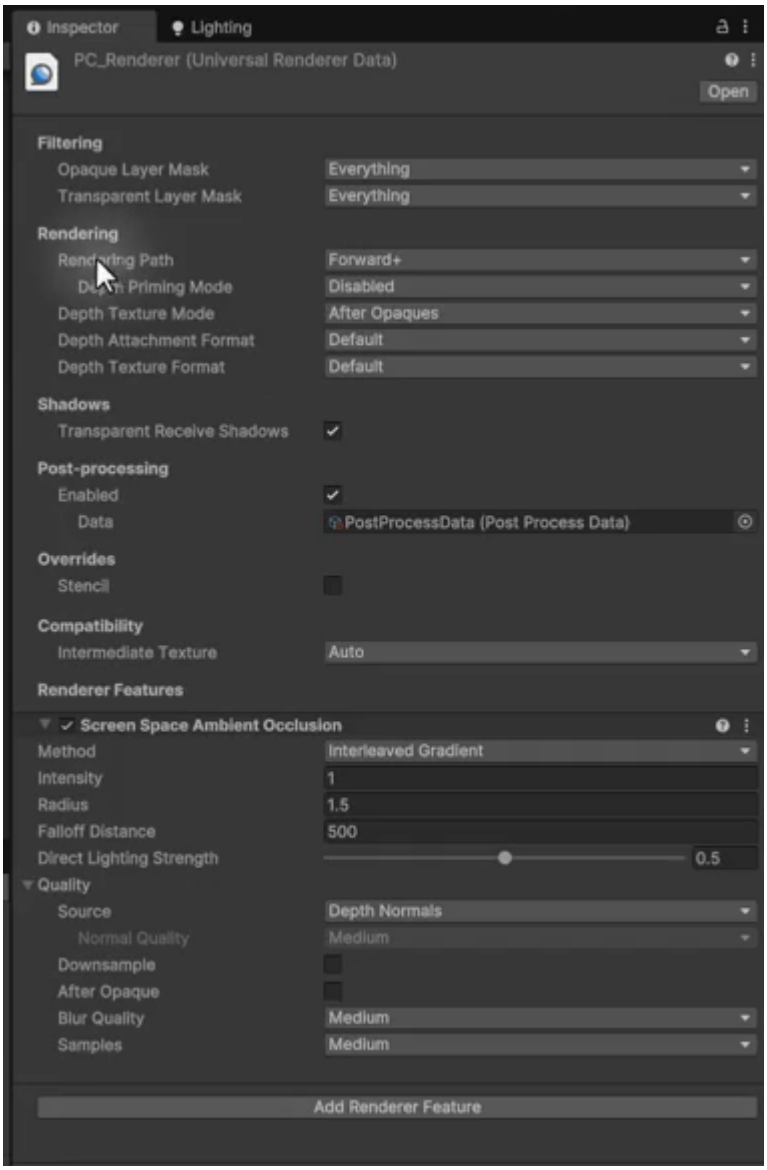
When you remove many real-time shadows, the scene can start to feel flatter. A practical way to recover some visual depth is to enable **Screen Space Ambient Occlusion (SSAO)**.

SSAO darkens creases, corners, and contact areas between surfaces. It does not replace true cast shadows, but it helps objects feel grounded and adds depth at a much lower cost than shadow maps. In shadow-light scenes, SSAO is often a good companion effect because it preserves some of the visual richness that would otherwise be lost.

Choosing a Rendering Path

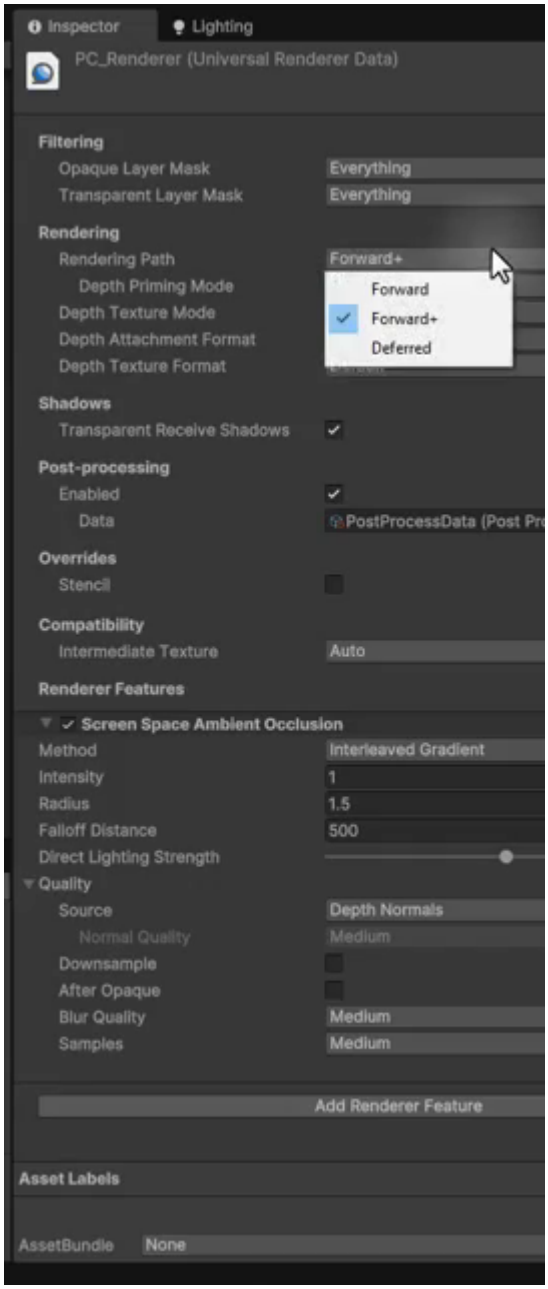
Even after reducing shadow cost, you may still run into the problem of lights disappearing. At that point, the next setting to evaluate is the renderer's **Rendering Path**.

In URP, this is configured on the **Universal Renderer Data** asset in the Inspector.

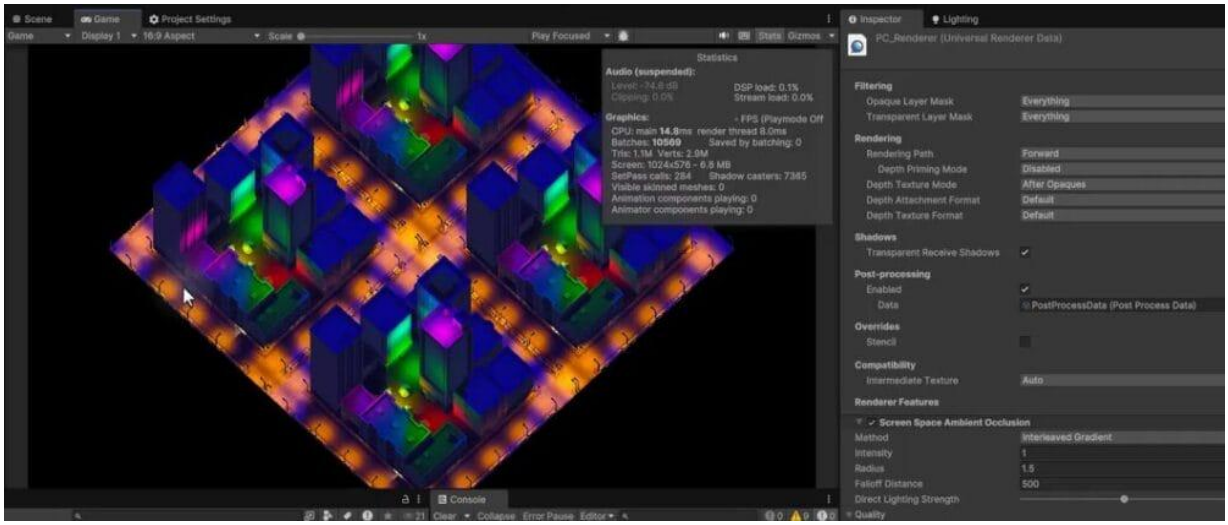


URP provides three main options:

- **Forward**: The most restrictive option. It supports the fewest visible real-time lights and is the most likely to cause lights to disappear in dense scenes.
- **Forward+**: A more flexible forward rendering mode that supports significantly more real-time lights. In most modern URP projects, this is the best default choice.
- **Deferred**: Best suited to scenes with very large numbers of lights. It removes most practical light count limits, but introduces its own overhead.


Inspector
Lighting
PC_Renderer (Universal Renderer Data)
Filtering
Opaque Layer Mask
Everything
Transparent Layer Mask
Everything
Rendering
Rendering Path
Forward+
Depth Priming Mode
Depth Texture Mode
Depth Attachment Format
Depth Texture Format
Forward
Forward+
Deferred
Shadows
Transparent Receive Shadows
Post-processing
Enabled
Data
PostProcessData (Post Pr
Overrides
Stencil
Compatibility
Intermediate Texture
Auto
Renderer Features
Screen Space Ambient Occlusion
Method
Interleaved Gradient
Intensity
1
Radius
1.5
Falloff Distance
500
Direct Lighting Strength
Quality
Source
Depth Normals
Normal Quality
Medium
Downsample
After Opaque
Blur Quality
Medium
Samples
Medium
Add Renderer Feature
Asset Labels
AssetBundle
None

The difference between **Forward** and **Forward+** is often immediately visible. In this example, switching to Forward causes many background building lights and rooftop accents to disappear.



Switching back to **Forward+** restores those lights. **Deferred** goes further by allowing a much larger number of lights to render, but that does not automatically make it the best option. In smaller scenes, or scenes that do not truly need that capacity, Deferred may perform worse than Forward+ because of its baseline overhead.

As a general guideline:

- Use **Forward** when the scene is simple and light count is low.
- Use **Forward+** for most URP projects that need a healthy number of dynamic lights.
- Use **Deferred** when the scene genuinely requires many simultaneous lights and the target hardware can handle the extra cost.

For mobile projects, **Forward** or **Forward+** are usually the better fit. Deferred is generally more appropriate for PC or console targets with stronger hardware.

Unity Insight: Rendering paths and light limits

The rendering path determines how Unity processes lights for each pixel on screen, and the differences go beyond just light count.

Forward rendering calculates lighting per-object. Each object is drawn once for the main directional light, then additional passes are added for each extra light that affects it. This means the cost scales with the number of lights multiplied by the number of affected objects, which is why the light cap exists.

Forward+ improves on this by using a per-tile light culling system. The screen is divided into tiles, and each tile tracks which lights affect it. Objects are still drawn in a single pass, but they can access many more lights without additional draw calls. This is why Forward+ can handle significantly more lights than Forward with similar performance.

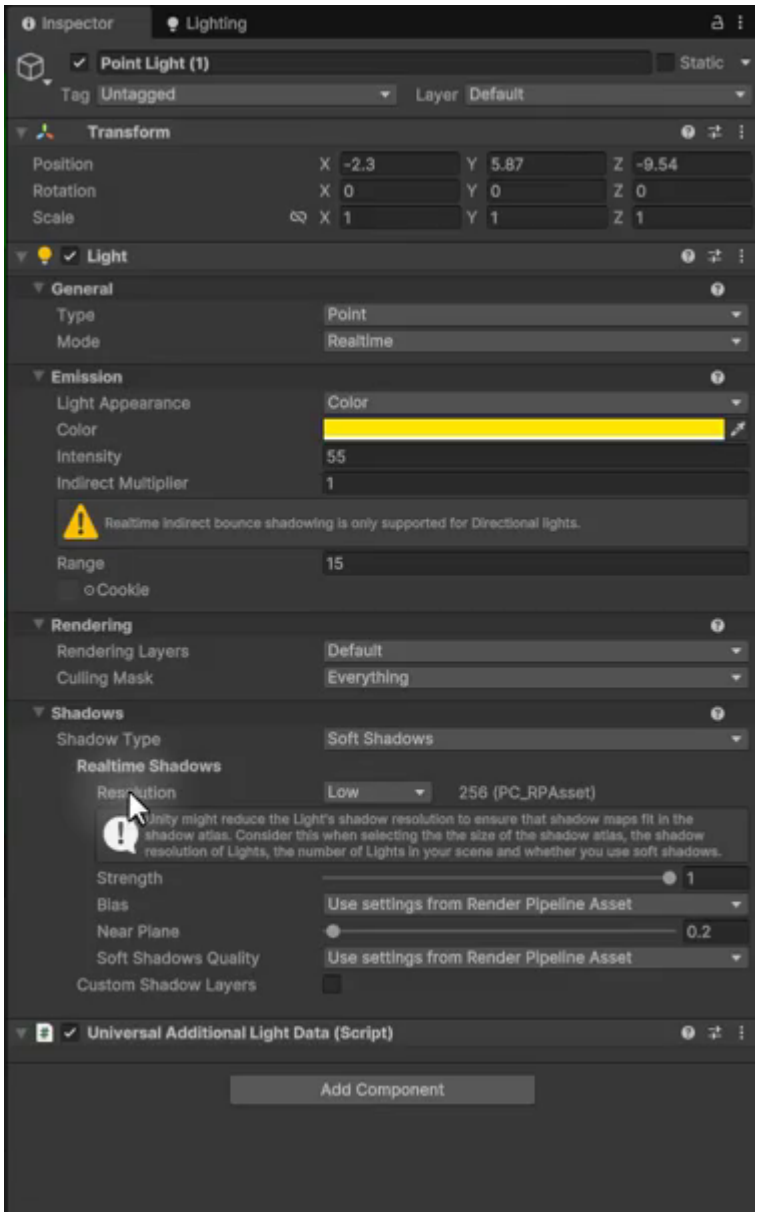
Deferred rendering decouples geometry from lighting entirely. It first renders all geometry into a set of screen-space buffers (the G-buffer), then applies lighting as a screen-space operation. This makes the cost of each light independent of scene complexity, but it cannot handle transparent objects and uses more memory for the G-buffer.

Learn more: Rendering paths (https://docs.unity3d.com/6000.3/Documentation/Manual/rendering-paths-introduction.html)

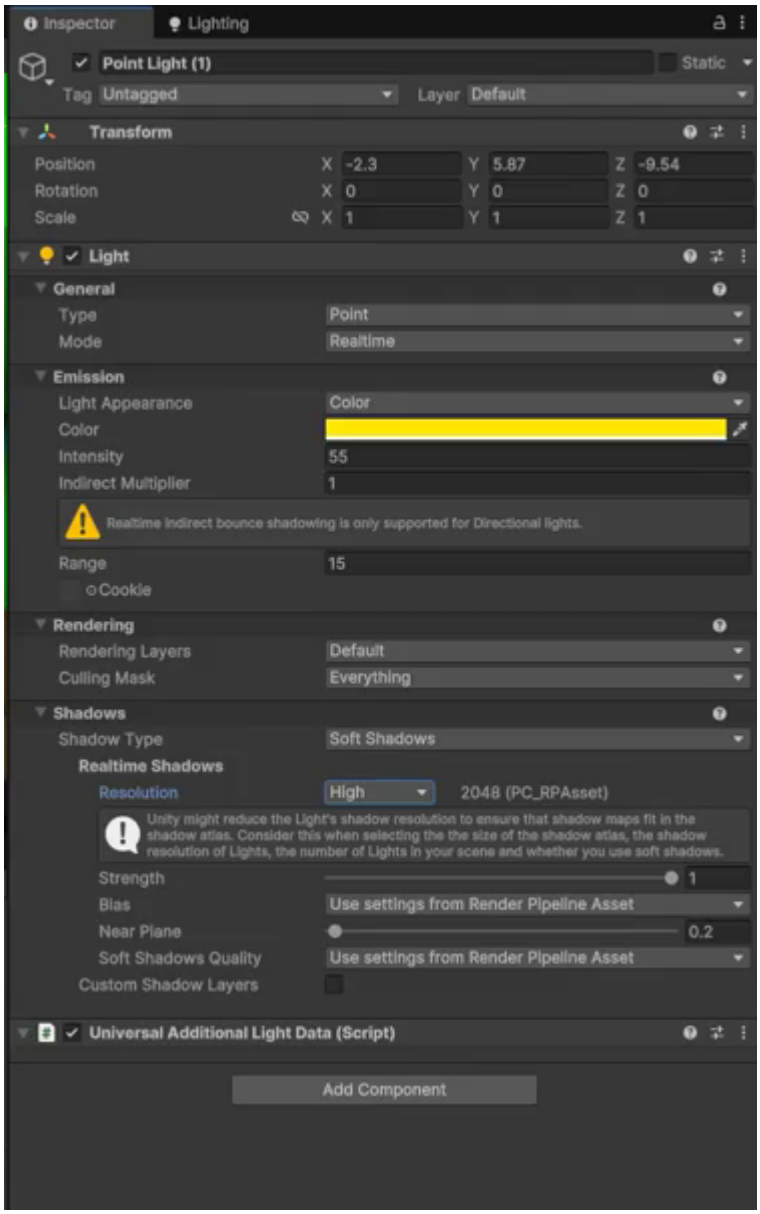
Adjusting Per-Light Shadow Resolution

When some real-time lights still need shadows, URP lets you control their shadow quality individually. This is useful because not every light deserves the same shadow budget.

Select a light in the scene, then expand its **Shadows** section in the Inspector. There you will find a **Resolution** setting.


Inspector
Lighting
Point Light (1)
Static
Tag Untagged
Layer Default
Transform
Position X -2.3 Y 5.87 Z -9.54
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Light
General
Type Point
Mode Realtime
Emission
Light Appearance Color
Color
Intensity 55
Indirect Multiplier 1
Realtime indirect bounce shadowing is only supported for Directional lights.
Range 15
Cookie
Rendering
Rendering Layers Default
Culling Mask Everything
Shadows
Shadow Type Soft Shadows
Realtime Shadows
Low 256 (PC_RPAsset)
Unity might reduce the Light's shadow resolution to ensure that shadow maps fit in the shadow atlas. Consider this when selecting the the size of the shadow atlas, the shadow resolution of Lights, the number of Lights in your scene and whether you use soft shadows.
Strength 1
Bias Use settings from Render Pipeline Asset
Near Plane 0.2
Soft Shadows Quality Use settings from Render Pipeline Asset
Custom Shadow Layers
Universal Additional Light Data (Script)
Add Component

Higher resolutions produce sharper shadows, but they also consume more space in the shadow atlas and increase rendering cost. If too many lights use high-resolution shadows, Unity may exceed the atlas budget and produce warnings or lower-quality results.



A practical approach is to use higher shadow resolution only on lights close to the player or near important surfaces where shadow detail is noticeable. Decorative lights often do not need shadows at all, and disabling them individually helps preserve the shadow budget for the lights that matter.

Working with Baked Lighting

Real-time lighting gives you full flexibility, but as we saw in the previous section, it comes at a steep runtime cost. When a scene is mostly static, baked lighting can provide both better performance and better visual quality. In this section, we will walk through the full baked lighting workflow in Unity and examine the settings that most directly affect quality, bake time, and file size.

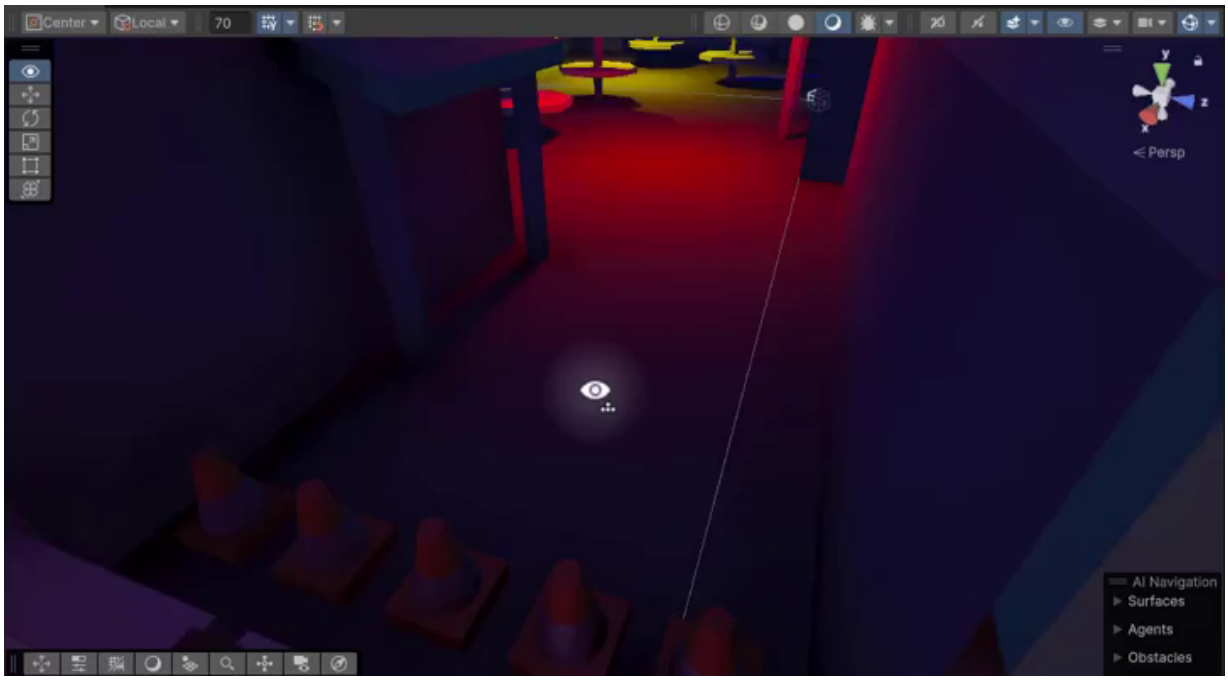
Why use Baked Lighting

Baked lighting precomputes light and shadow information ahead of time and stores the result in textures called **lightmaps**. At runtime, Unity reads those textures instead of recalculating the lighting every frame.

This has two major advantages:

- **Improved performance**: The game no longer needs to spend as much CPU and GPU time on lighting calculations during play.
- **Better indirect lighting**: Baked lighting can simulate bounced light, color bleeding, and softer illumination that real-time lighting often cannot reproduce as efficiently.

For example, if a red light shines onto the floor, baked lighting can capture the subtle red bounce onto nearby surfaces. That kind of indirect effect adds realism and depth with no runtime lighting cost.



If the scene's lighting does not need to change during gameplay, baking is often the best optimization available. Baking is ideal for static environments such as fixed nighttime streets, interior spaces, or levels with no day-night cycle. If the lighting must change dynamically, baking is not suitable and the real-time strategies covered above become more important.

Trade-offs of Baked Lighting

Baked lighting is powerful, but it is not free. The main trade-off is storage and memory use.

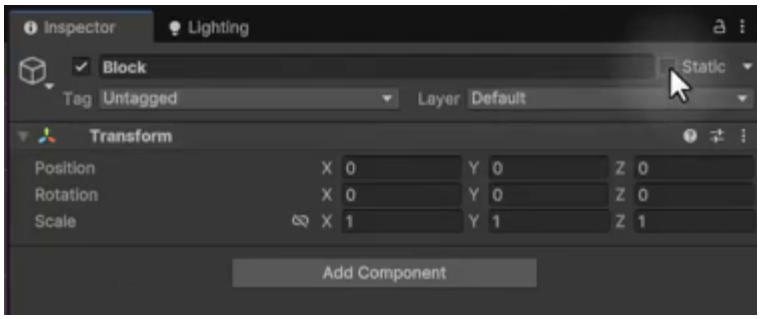
Because the lighting is stored in textures, large scenes or high-quality bakes can generate a significant amount of lightmap data. Depending on the scene, this may range from tens of megabytes to hundreds of megabytes or more. Those textures also occupy memory at runtime.

In many projects, that trade-off is worthwhile because the runtime performance gain is so large. Still, it is important to remember that baked lighting shifts cost away from frame time and into bake time, disk space, and memory usage.

Step 1: Marking Objects as Static

Before Unity can bake lighting onto scene geometry, it needs to know which objects will remain stationary. Only objects that never move should be included in the bake, because if a moving object is baked as static, the lightmap stays fixed on its surface as it moves, producing obviously incorrect results.

To prepare the scene, select the parent object that contains the environment models. In the Inspector, enable the **Static** checkbox. When Unity asks whether to apply the change to child objects, go ahead and confirm.



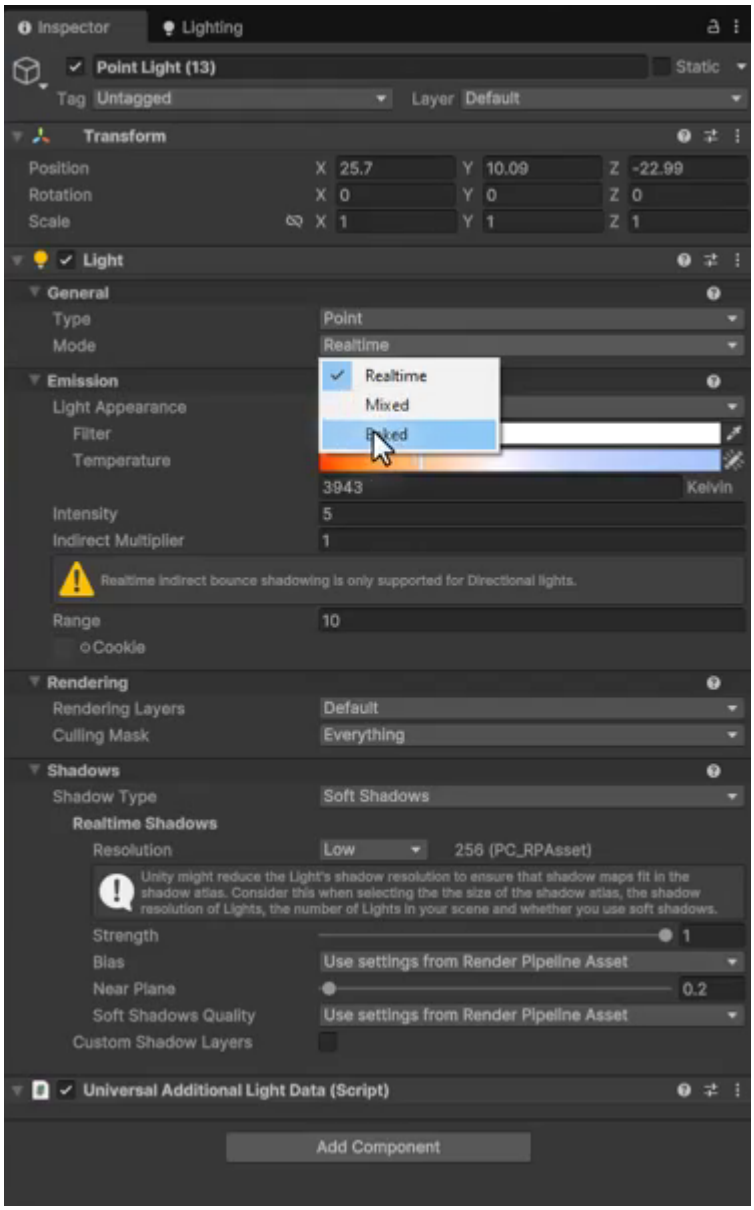
Step 2: Setting Light Modes

Once the geometry is marked correctly, we need to tell Unity how each light should contribute to the bake. Unity provides three light modes:

- **Realtime**: The light is calculated every frame and does not contribute baked data.
- **Baked**: The light contributes only to the bake and has no runtime lighting cost.
- **Mixed**: The light contributes baked lighting for static objects while still affecting dynamic objects in real time.

For static decorative lights, **Baked** is usually the right choice. For a directional sun or another important light that still needs to affect dynamic objects, **Mixed** is often more appropriate.

To change this, select a light and adjust the **Mode** field in the Light component.


Inspector
Lighting
Point Light (13)
Static
Tag Untagged
Layer Default
Transform
Position X 25.7 Y 10.09 Z -22.99
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Light
General
Type Point
Mode Realtime
Realtime
Mixed
Baked
Emission
Light Appearance
Filter
Temperature
3943 Kelvin
Intensity 5
Indirect Multiplier 1
Realtime indirect bounce shadowing is only supported for Directional lights.
Range 10
Cookie
Rendering
Rendering Layers Default
Culling Mask Everything
Shadows
Shadow Type Soft Shadows
Realtime Shadows
Resolution Low 256 (PC_RPAsset)
Unity might reduce the Light's shadow resolution to ensure that shadow maps fit in the shadow atlas. Consider this when selecting the the size of the shadow atlas, the shadow resolution of Lights, the number of Lights in your scene and whether you use soft shadows.
Strength 1
Bias Use settings from Render Pipeline Asset
Near Plane 0.2
Soft Shadows Quality Use settings from Render Pipeline Asset
Custom Shadow Layers
Universal Additional Light Data (Script)
Add Component

Go through the scene and assign each light to **Baked** or **Mixed** based on whether dynamic objects need to respond to it during gameplay.

Unity Insight: Choosing between light modes

The three light modes control a fundamental trade-off between visual flexibility and runtime cost.

Realtime lights are the most flexible but the most expensive. They update every frame, support fully dynamic shadows, and respond to moving objects. Use them only when the light itself must move or change intensity during gameplay.

Baked lights are the cheapest at runtime because their contribution is stored entirely in lightmaps. The trade-off is that they cannot affect dynamic objects at all. A character walking past a baked point light will not receive any illumination from it.

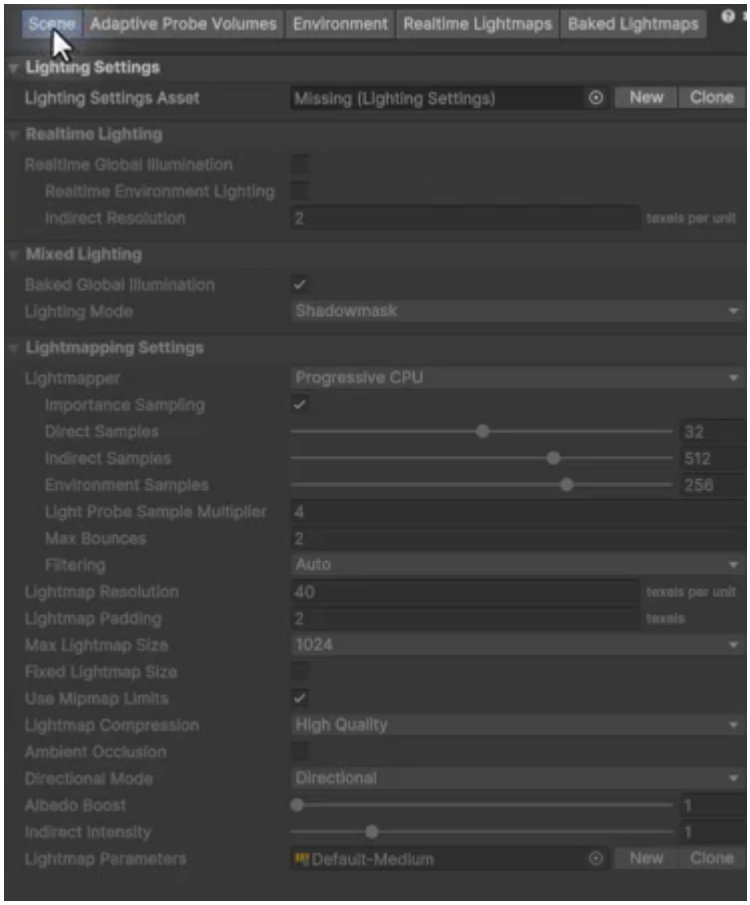
Mixed lights offer a middle ground. They bake indirect lighting onto static surfaces (saving runtime cost) while still casting real-time direct light and shadows onto dynamic objects. This makes them ideal for important scene lights like the sun, where both static and dynamic objects need to be lit correctly.

Learn more: Light modes (https://docs.unity3d.com/6000.3/Documentation/Manual/LightModes-introduction.html)

Step 3: Opening the Lighting Window

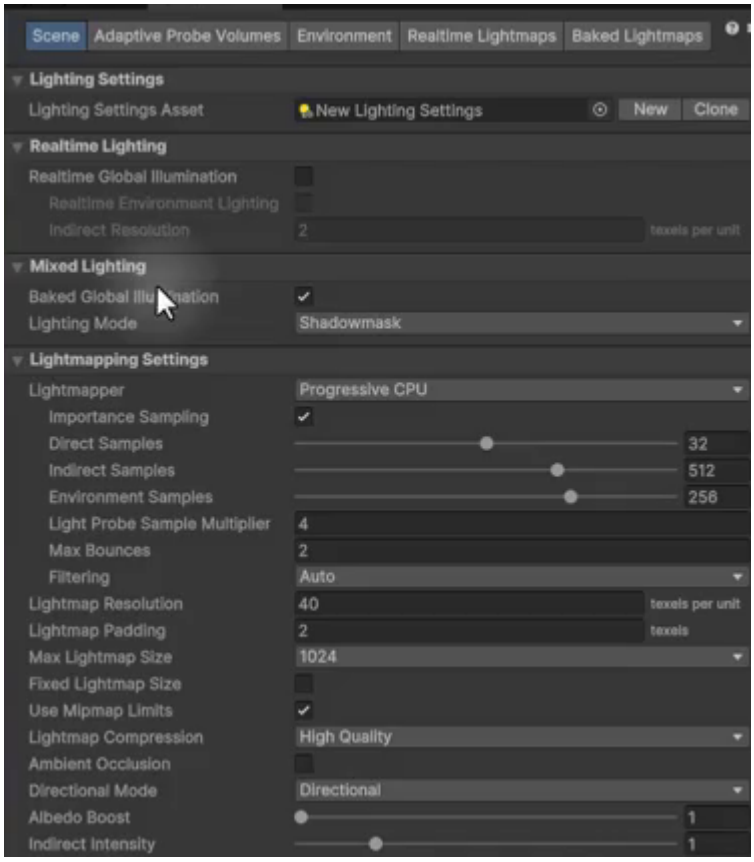
With the scene prepared, we can configure the bake itself. Open the Lighting window through **Window > Rendering > Lighting**, then switch to the **Scene** tab.

In the **Lighting Settings** section, we need to create a new settings asset by clicking **New**. This asset stores the bake configuration and can be reused across scenes if needed.



Step 4: Navigating the Lightmapping Settings

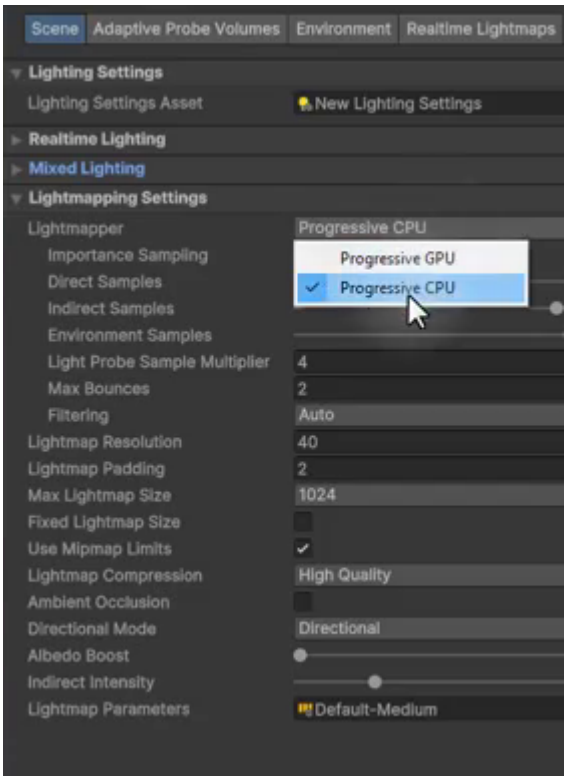
The Lighting window contains many options, but most of the important bake controls are grouped under **Lightmapping Settings**. The default values are a reasonable starting point, though different scene types (indoor, outdoor, large, small) benefit from different settings. Expect some experimentation here.



Step 5: Choosing a Lightmapper

The lightmapper determines which hardware generates the lightmap textures. Unity offers two backends:

- **Progressive CPU**: Uses the processor to generate lightmaps.
- **Progressive GPU**: Uses the graphics card and is usually faster on systems with capable hardware.



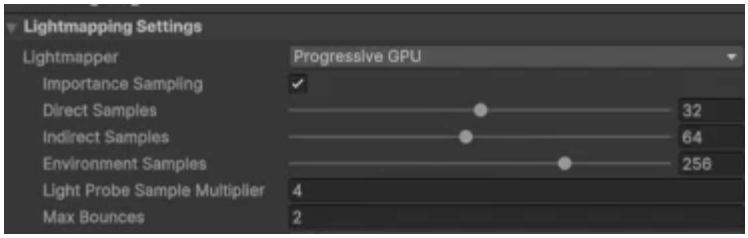
If your machine supports it well, **Progressive GPU** is generally the better choice because it reduces bake times. Even so, the exact behavior and limitations can differ between the two, so it is worth testing both if your project has unusual requirements.

Step 6: Configuring Sample Settings

The sample settings control how much work Unity does while calculating the bake. Higher values usually improve quality, but they also increase bake time.

The three most important sample settings are:

- **Direct Samples**: Controls the quality of direct lighting. The default value of 32 is often sufficient.
- **Indirect Samples**: Controls the quality of bounced light. This is usually the most important setting for smooth indirect illumination.
- **Environment Samples**: Controls how much the skybox or environment contributes to the bake.

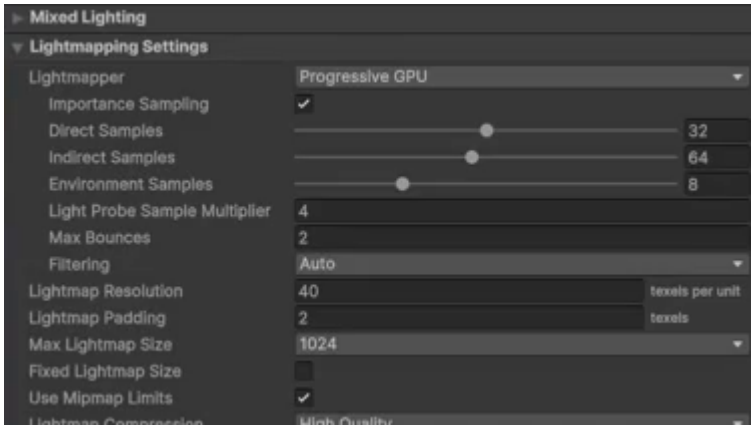


These values should reflect the scene. In a nighttime outdoor environment, for example, **Direct Samples** can often stay at 32, **Indirect Samples** can be reduced compared to a dense indoor scene, and **Environment Samples** can be lowered because the sky contributes less light.

Max Bounces

The **Max Bounces** setting controls how many times light is allowed to reflect between surfaces during the bake.

A value of **2** is a good default for most scenes. It allows light to bounce once or twice, which is enough to create believable indirect illumination without making bake times unnecessarily long.



Higher bounce counts can improve very small or enclosed scenes, but they also increase bake cost. In many projects, 2 or 3 bounces is enough.

Refining Baked Lighting Quality

A first bake rarely looks perfect. Once the basic setup is in place, the next step is to tune quality settings and generate improved lightmaps. This is where the trade-offs become more visible: sharper shadows and smoother indirect light usually require more texels, more samples, and larger lightmaps.

Understanding Lightmap Resolution

Lightmap Resolution controls how many texels per unit Unity uses when storing baked lighting. Since one Unity unit typically represents one meter, this setting determines the density of lighting detail across the scene.

Higher values produce sharper shadows and cleaner lighting transitions, but they also increase bake time and file size. Lower values are faster and smaller, but can lead to blotchy or blurry results.

For an outdoor scene viewed from a distance, around **30 texels per unit** is a reasonable starting point. If the camera is closer to the environment, **40**

texels per unit may be more appropriate.

Unity Insight: Lightmap resolution and scene scale

Lightmap resolution is measured in texels per world unit, which means the total lightmap size depends on both the resolution setting and the physical size of the scene. A small room at 40 texels per unit might produce a compact lightmap, while a large outdoor environment at the same setting could generate hundreds of megabytes of texture data.

One way to manage this is to use the **Lightmap Resolution** preview in the Scene view. With the Lighting window open, Unity can overlay a checkerboard pattern showing the actual texel density on each surface. This helps you identify which parts of the scene are consuming the most lightmap space and whether the resolution is appropriate for the camera distance.

If certain objects need more or less detail than the global setting provides, you can override resolution per-object using the **Scale in Lightmap** property on individual Mesh Renderers. This lets you allocate higher resolution to surfaces the player sees up close while keeping distant geometry lighter.

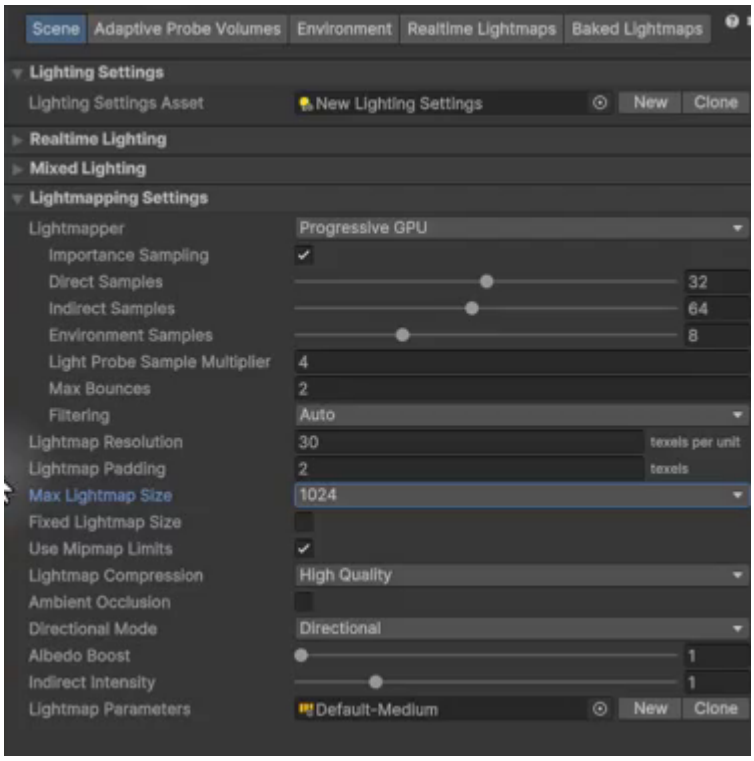
Learn more: Lightmapping (https://docs.unity3d.com/Manual/Lightmapping.html)

Lightmap padding and max lightmap size

Two other settings strongly affect bake quality and texture layout:

- **Lightmap Padding**: Controls the spacing between UV regions in the lightmap atlas. A value of **2** is usually enough to reduce light bleeding.
- **Max Lightmap Size**: Controls the maximum dimensions of each lightmap texture. The default of **1024** works well for many projects.

Keeping **Fixed Lightmap Size** disabled allows Unity to allocate textures more flexibly, which is usually the better choice during normal production.

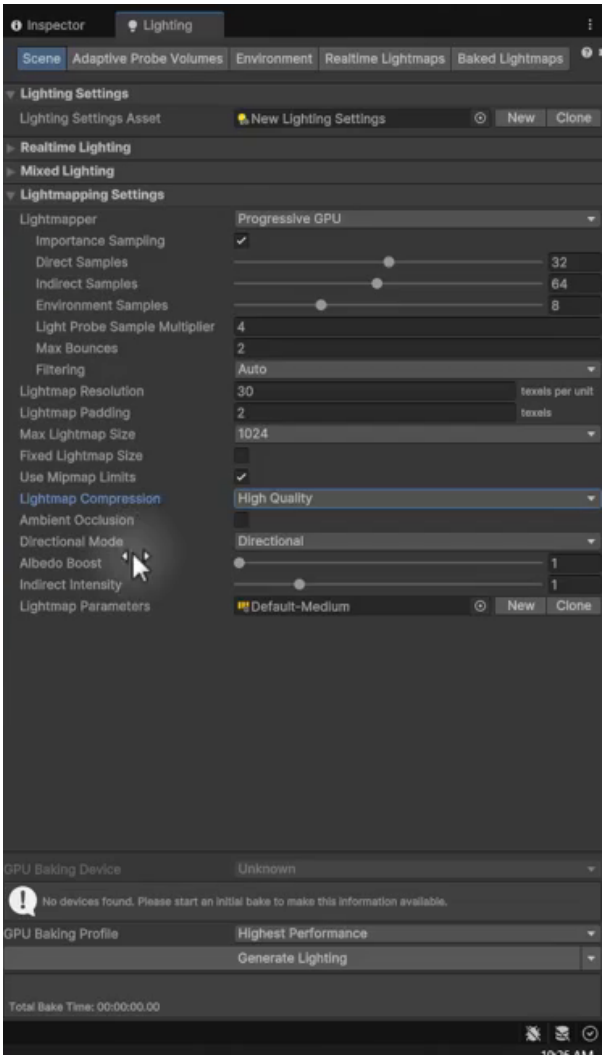


Lightmap Compression

Lightmaps can consume a large amount of storage, so Unity provides several compression options:

- **None**: Highest quality, largest file size.
- **High Quality**: A strong balance between quality and storage.
- **Normal Quality**: More compression with more visible artifacts.
- **Low Quality**: Smallest files, but the most noticeable degradation.

For most projects, **High Quality** is the best default because it keeps file size under control without introducing major visual problems.



Ambient Occlusion in the Bake

Unity can also bake **Ambient Occlusion** directly into the lightmap. This adds extra depth to corners, creases, and contact areas.

Even if your renderer already uses SSAO, baked ambient occlusion can still improve the final look because it becomes part of the static lighting itself. For an initial bake, the remaining settings can usually stay at their defaults.

Generating Lighting

With the settings configured, we can run the bake and see how the scene looks. To start the process, click **Generate Lighting** at the bottom of the

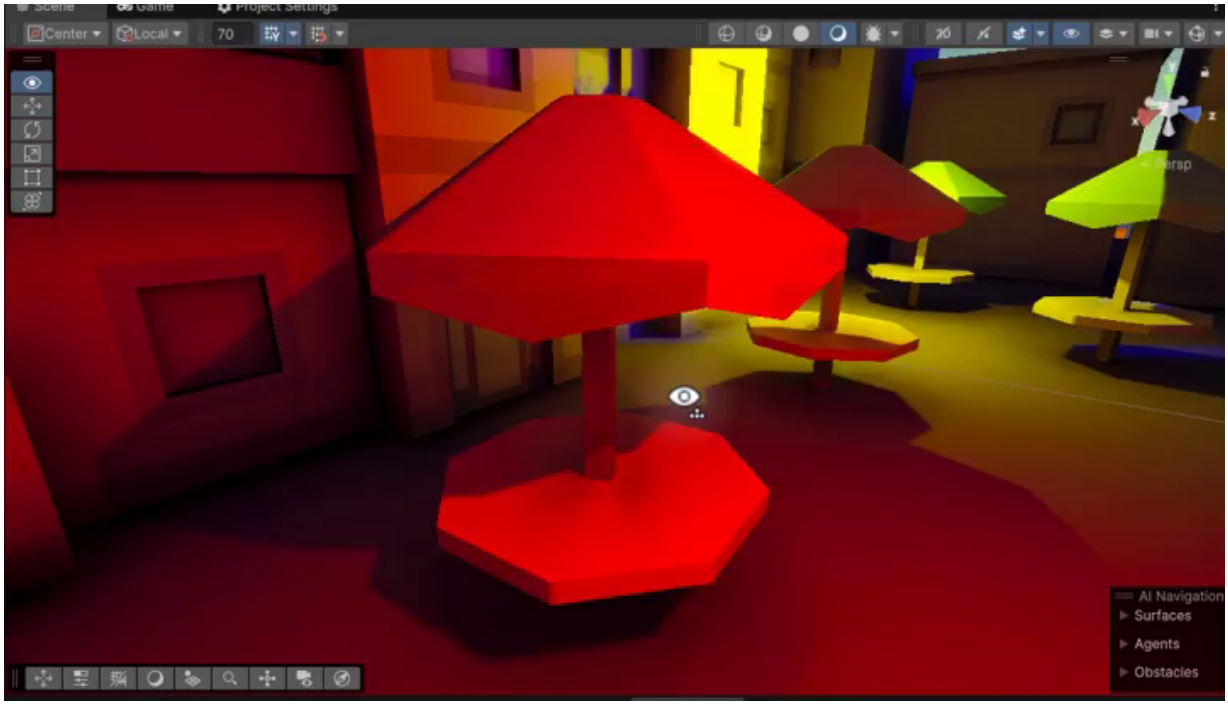
Lighting window.

Bake time depends on scene complexity and quality settings. Small scenes may complete in a few minutes, while larger scenes can take much longer. To monitor progress, click the **Global Illumination** status indicator at the bottom of the editor to open the Background Tasks window.

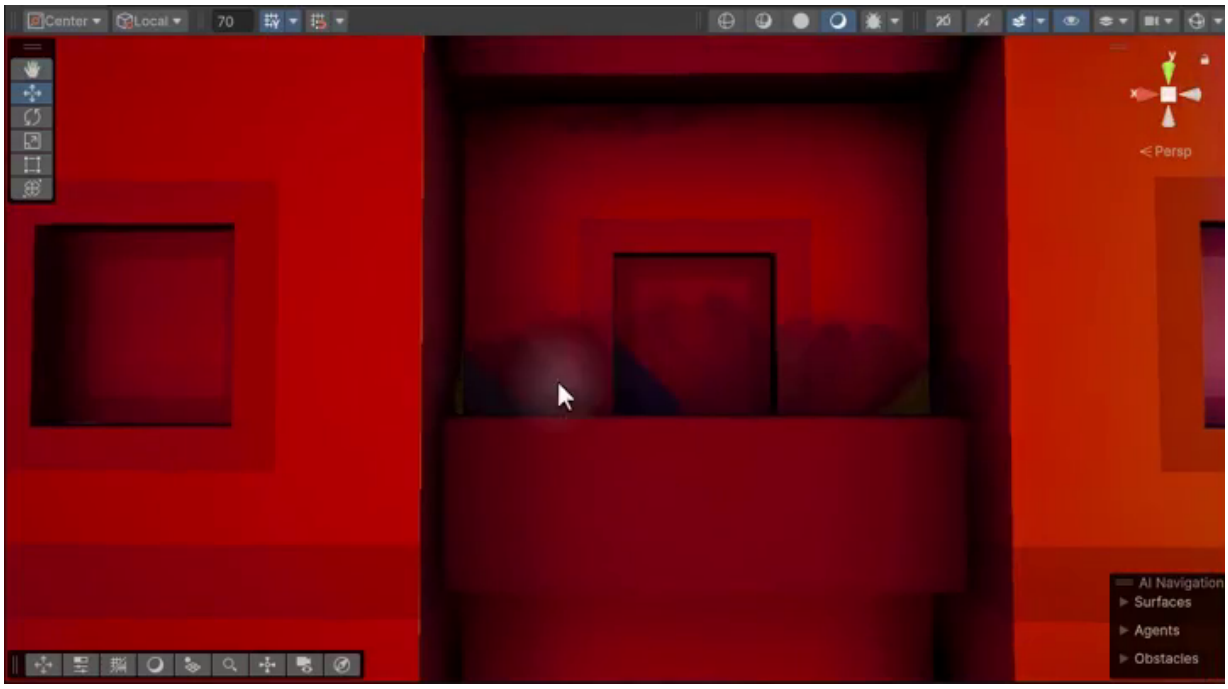
Reviewing the First Bake

After the first bake completes, the scene may not look dramatically different at a glance, but the improvements become clearer on closer inspection.

Direct light now reaches nearby surfaces more consistently, and indirect lighting fills areas that were previously too dark. Light bouncing from one surface to another creates subtle color transfer, which helps the scene feel more natural and grounded.



At the same time, a first bake often reveals quality issues. In lower-quality bakes, shadowed areas may look blotchy, and transitions between overlapping light sources may appear rough or imprecise.



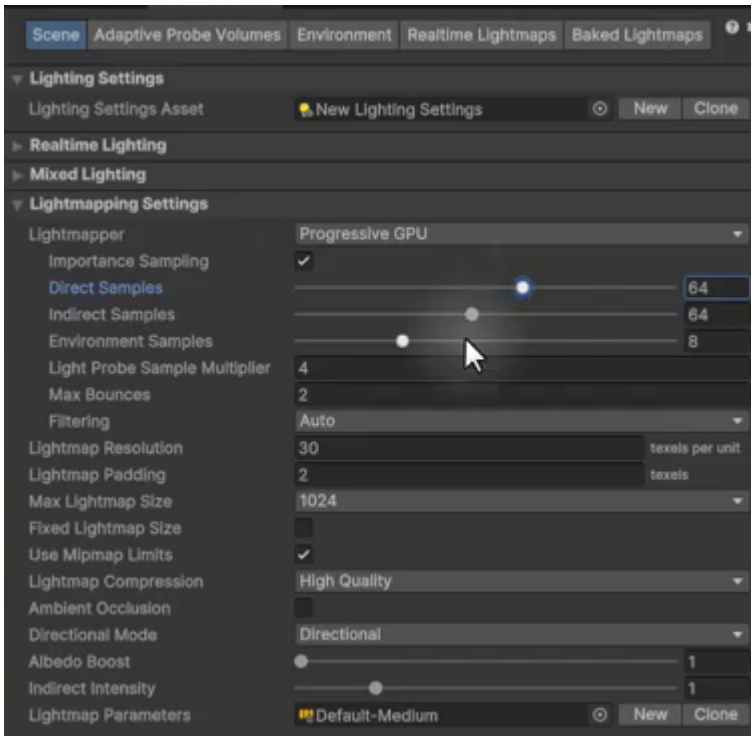
These artifacts are normal in an early pass. The purpose of the first bake is often to establish a baseline before investing in higher settings.

Improving Bake Quality

To reduce artifacts, we can increase several quality settings before baking again. In this example, the following adjustments are made:

- **Direct Samples**: Increased from 32 to 64
- **Indirect Samples**: Increased to 512
- **Environment Samples**: Increased slightly
- **Max Bounces**: Increased to 3
- **Lightmap Resolution**: Increased to 60 texels per unit
- **Max Lightmap Size**: Increased to 2048

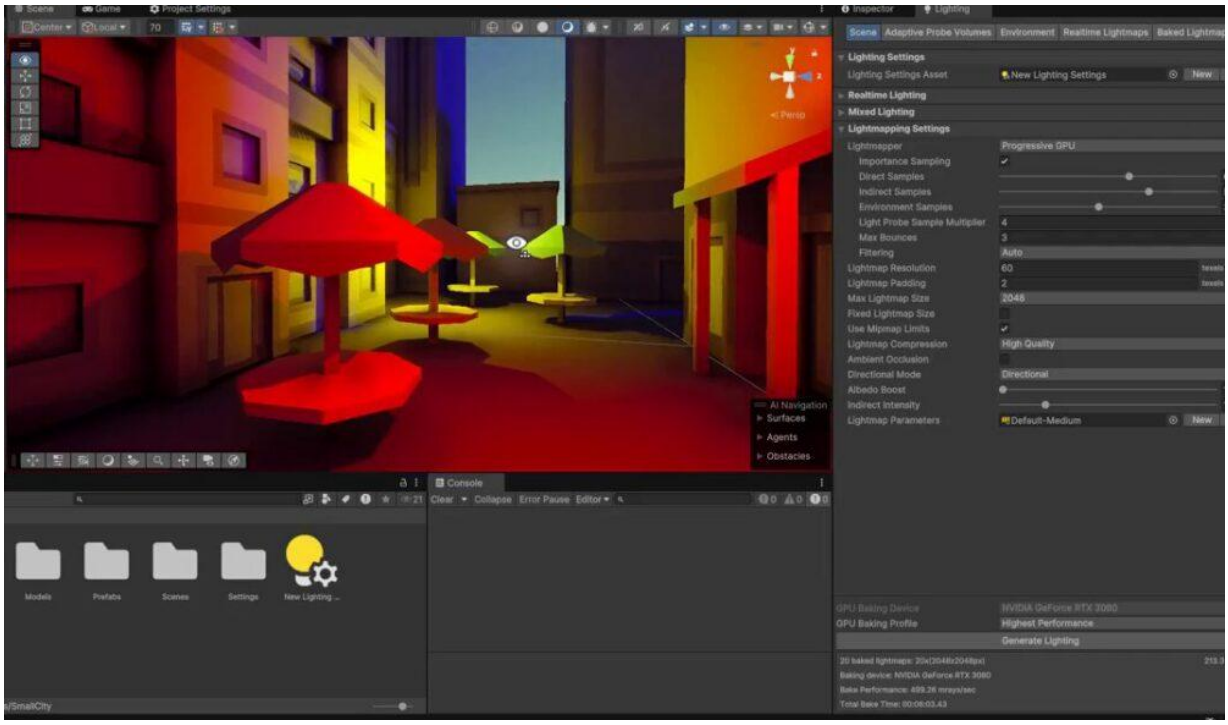
These changes improve shadow definition and smooth out indirect lighting, but they also increase bake time and file size.



Comparing the Second Bake

After the second bake, the difference is much easier to see. Shadows are sharper because of the higher lightmap resolution, and indirect lighting transitions are smoother because of the increased sample counts.

In this example, the bake statistics report **20 lightmaps at 2048 x 2048**, a total size of roughly **213 MB**, and a bake time of about **six minutes**.



Some artifacting may still remain on thin or flat geometry. That is a common limitation of baked lighting and can be difficult to eliminate completely without pushing settings even higher.

Troubleshooting Stray Light Artifacts

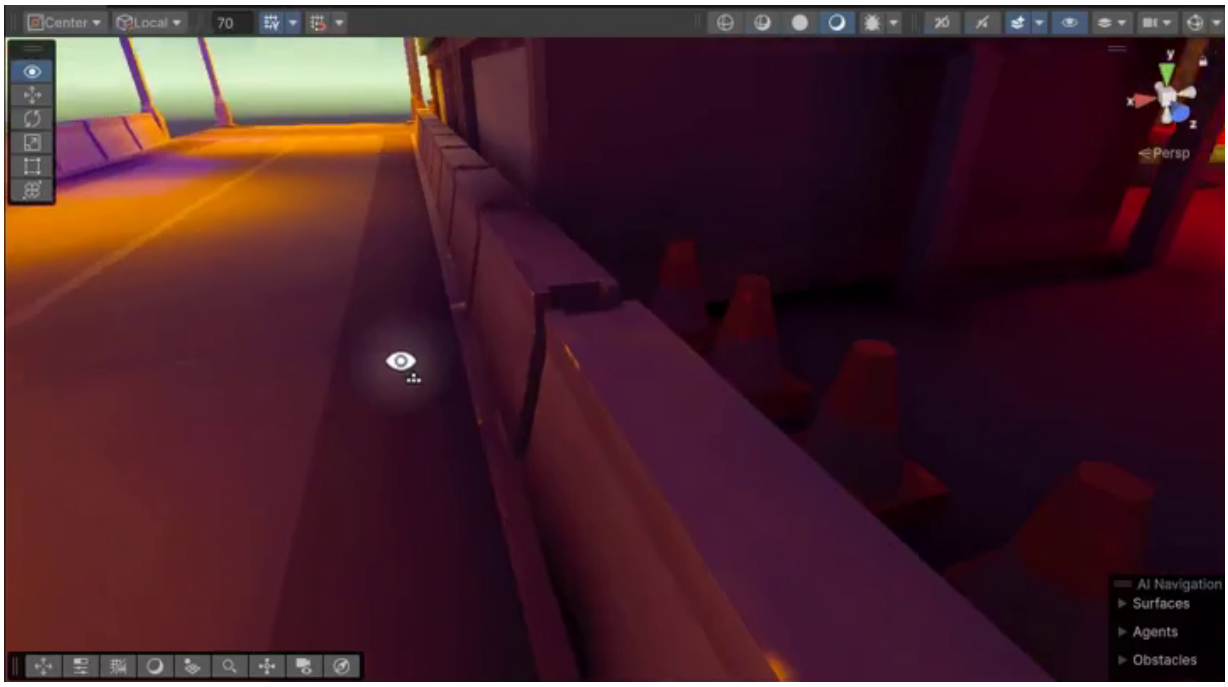
If your night scene shows bright patches or unexpected lighting on surfaces that should remain dark, the cause is usually insufficient sampling or light leaking between nearby regions in the lightmap atlas. The good news is that this is straightforward to address.

Try increasing any combination of the following:

- **Sample counts**, so the lightmapper calculates each texel more accurately
- **Lightmap resolution**, so there are more texels per surface to capture detail

- **Lightmap padding**, so adjacent UV regions in the atlas do not bleed into each other

A second bake after these adjustments will usually reduce or eliminate the stray patches.



Troubleshooting Incorrect Lightmap UVs

If baked shadows and lighting appear completely wrong (streaking up walls or coming from impossible directions), the most likely cause is incorrect **lightmap UVs** on one or more models. This is especially common with models imported from external tools, where the existing UV channels may not match what Unity's lightmapper expects.

The fix is simple: select the affected model in the Project window and open the **Model** tab in the Inspector. From there, enable **Generate Lightmap UVs** and click **Apply**. Unity will create a UV layout suited for lightmapping.


Inspector
Lighting
3 Model Import Settings
Open
Model
Rig
Animation
Materials
Scene
Scale Factor
10
Convert Units
1dm (File) to 0.09999999m (Unity)
Bake Axis Conversion
Import BlendShapes
Import Deform Percent
Import Visibility
Import Cameras
Import Lights
Preserve Hierarchy
Sort Hierarchy By Name
Meshes
Mesh Compression
Off
Read/Write
Optimize Mesh
Everything
Generate Colliders
Geometry
Keep Quads
Weld Vertices
Index Format
Auto
Legacy Blend Shape Normals
Normals
Import
Blend Shape Normals
Calculate
Normals Mode
Area And Angle Weighted
Smoothness Source
Prefer Smoothing Groups
Smoothing Angle
60
Tangents
Calculate Mikktspace
Swap UVs
Generate Lightmap UVs
Lightmap UVs settings
Strict Vertex Data Checks
Revert
Apply
Asset PostProcessors
Asset Labels

Runtime Performance Benefits

The main reason baked lighting is such a valuable optimization is that it removes most lighting work from runtime. Once the bake is complete, the GPU no longer needs to recalculate those lights and shadows every frame.

That can lead to a substantial frame rate improvement compared to a similar real-time setup, especially in scenes with many lights. For static environments, baked lighting is often one of the highest-impact optimizations available.

Summary

This chapter covered two major lighting workflows in Unity: optimizing **real-time lighting** and configuring **baked lighting**.

For real-time lighting, the most important techniques were:

- Disabling **Cast Shadows** for additional lights when shadow detail is not essential
- Using **SSAO** to restore some depth in lower-shadow scenes
- Choosing the right **Rendering Path** for the project's light count and target platform
- Adjusting **per-light shadow resolution** so shadow quality is reserved for the lights that matter most

For static scenes, baked lighting provides an even stronger optimization path. By precomputing light and shadow into lightmaps, Unity can deliver better indirect lighting and much lower runtime cost. The key setup steps were:

- Marking environment objects as **Static**
- Setting lights to **Baked** or **Mixed**

- Configuring the **Lighting Settings** asset
- Tuning sample counts, bounce count, resolution, padding, and compression
- Troubleshooting artifacts through better settings and proper **lightmap UVs**

In the next chapter, we will move from lighting to another important rendering optimization: reducing geometric cost with Level of Detail systems.

Scene Visibility

This chapter focuses on two techniques that reduce the cost of rendering large 3D scenes: **Level of Detail (LOD)** and **occlusion culling**. Both aim to avoid unnecessary rendering work, but they solve different problems.

LOD reduces the complexity of objects as they become less important on screen. Occlusion culling prevents Unity from rendering objects that are hidden behind other geometry. Used together, these techniques can dramatically lower triangle counts and improve frame rate in scenes with many visible assets.

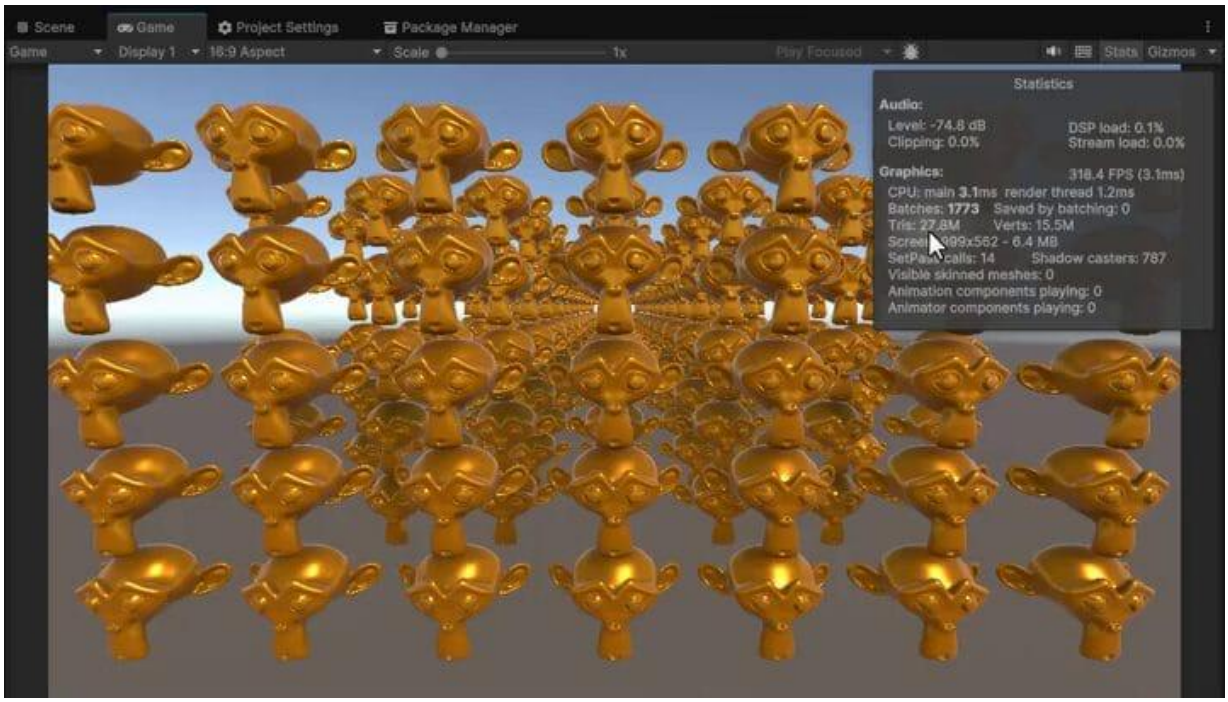
Working with Level of Detail

In most 3D games, the camera can see objects at a wide range of distances. A building right next to the player needs full geometric detail, but a building on the horizon might occupy only a handful of pixels. Rendering both at the same fidelity wastes GPU resources that could be spent elsewhere.

Level of Detail (LOD) solves this by letting each object swap to simpler mesh versions as it gets farther from the camera. It is one of the most widely used optimizations in 3D games, and the results can be dramatic.

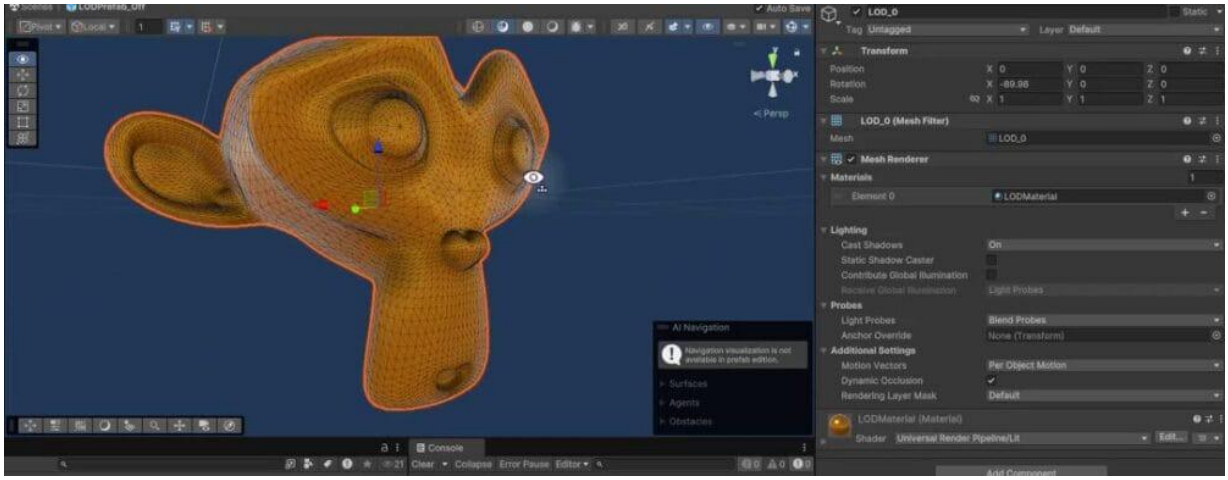
Why LOD Matters

To see the problem clearly, consider a scene filled with hundreds of identical high-polygon monkey head models. Each model contains thousands of triangles, and when all of them are rendered at full quality regardless of distance, the total triangle count becomes extremely high.



In this example, the Statistics panel reports roughly **27.8 million triangles** being drawn each frame. The frame rate may still appear acceptable because every object shares the same mesh and material, which allows Unity to batch them efficiently. In a real project, however, scenes usually contain many different meshes and materials. Under those conditions, rendering nearly 30 million triangles would be far more expensive.

The underlying issue is that distant objects are still being drawn at full fidelity even though they contribute very little to the final image. Looking at the mesh in wireframe makes that density much easier to appreciate.



How Unity’s LOD System Works

Unity’s LOD system lets you assign multiple versions of the same model, each with a different polygon count. Unity then switches between them automatically based on how large the object appears on screen.

A typical setup looks like this:

- **LOD 0**: The highest-detail version, used when the object is close to the camera
- **LOD 1**: A reduced-detail version for medium distance
- **LOD 2**: A much simpler version for far distance
- **Culled**: The object is not rendered at all once it becomes too small to matter

At runtime, Unity renders only one of these versions at a time for each object. As the object occupies less screen space, Unity swaps to a cheaper mesh.

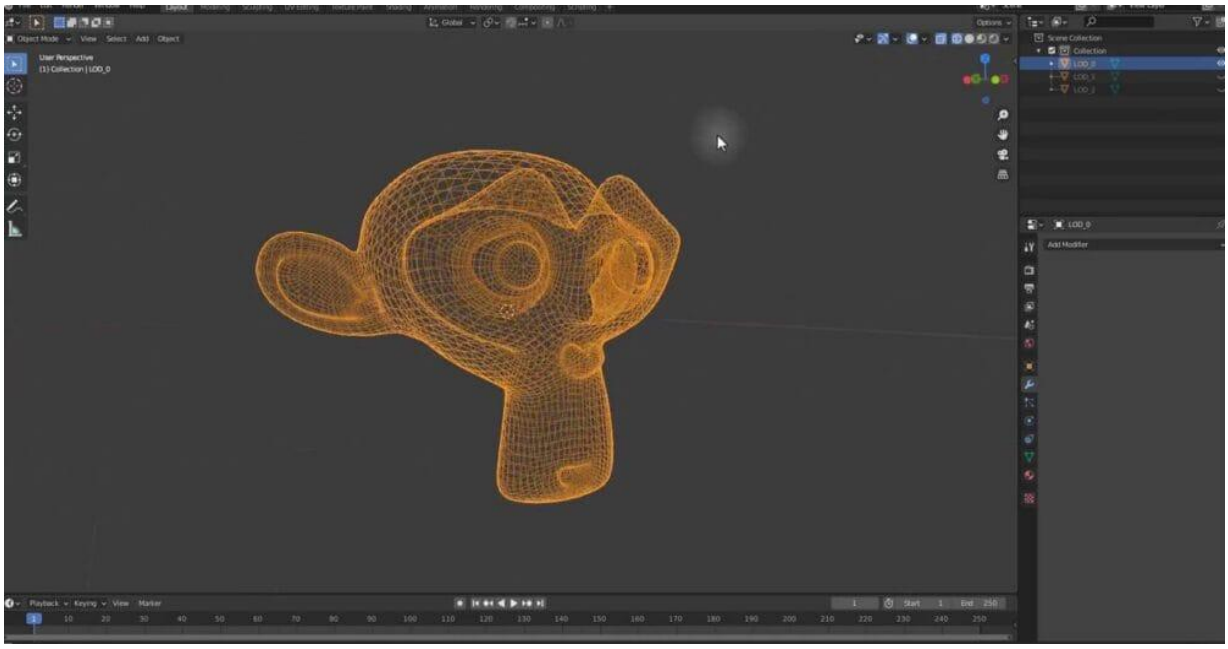
Creating LOD Meshes in Blender

LOD does not begin in Unity. Before you can configure LOD groups, you need multiple versions of the same model at different levels of complexity.

These are typically created in a modeling tool such as Blender, either by hand or with a decimation modifier.

In this example, the Blender file contains three mesh variants:

- **LOD_0**: The full-detail version
- **LOD_1**: A medium-detail reduction
- **LOD_2**: A minimal-detail version for far distances



The difference between these versions can be substantial. The highest-detail mesh may contain thousands of triangles, while the far-distance version may use only a small fraction of that geometry. Because the simplest version is only shown when the object is far away, the visual loss is usually negligible.

Setting up the LOD Group Component

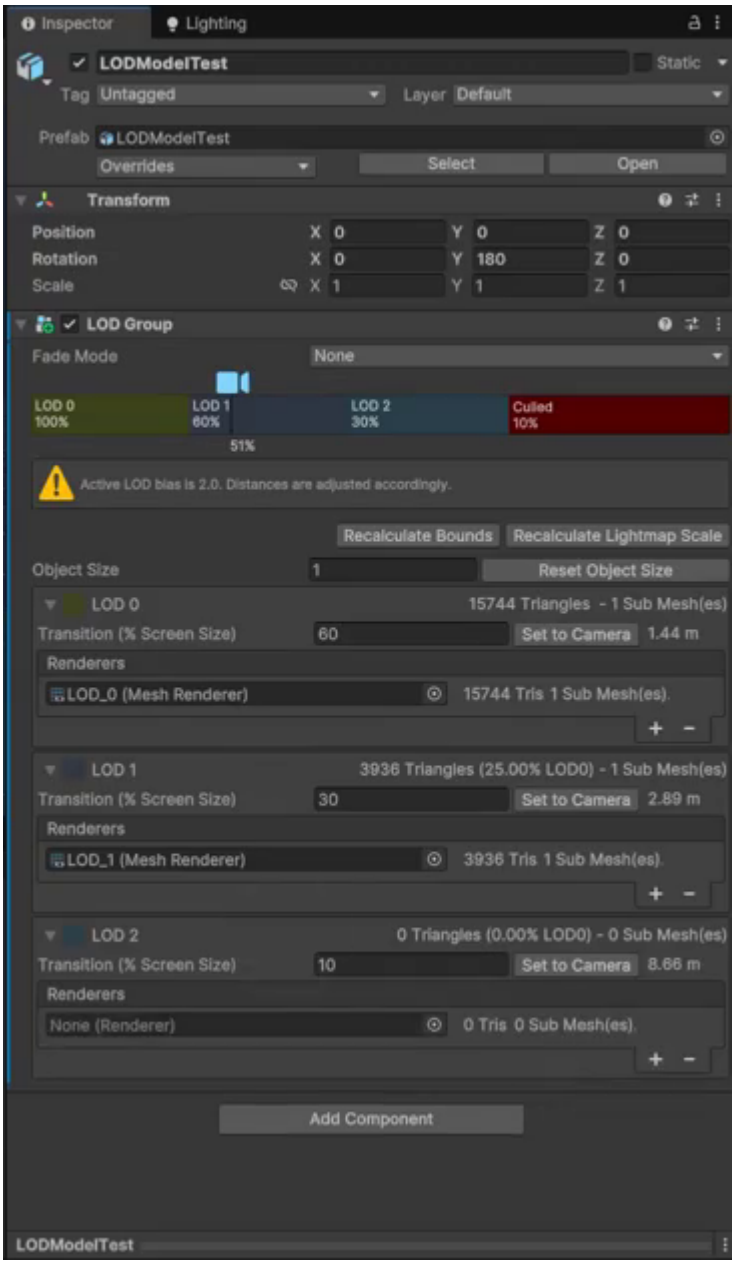
With the model variants imported into Unity, we need to tell Unity how to use them. The **LOD Group** component connects each mesh variant to a distance threshold so Unity knows when to swap between them.

Select the prefab and click **Add Component** to add an **LOD Group**. Unity will display a horizontal LOD timeline in the Inspector, with sections for each level and a final culled region.

Assigning Renderers to Each LOD Level

The LOD Group only works once each level has the correct renderer assigned. In this case, we want the high-detail renderer in LOD 0, the medium-detail renderer in LOD 1, and the low-detail renderer in LOD 2.

Drag the appropriate mesh renderer into each **Renderers** field in the LOD Group. Unity also shows the triangle count for each assigned renderer, which makes it easy to confirm that each level is actually cheaper than the previous one.


Inspector
Lighting
LODModelTest
Static
Tag Untagged
Layer Default
Prefab LODModelTest
Overrides
Select
Open
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 180 Z 0
Scale X 1 Y 1 Z 1
LOD Group
Fade Mode None
LOD 0 100%
LOD 1 60%
LOD 2 30%
Culled 10%
51%
Active LOD bias is 2.0. Distances are adjusted accordingly.
Recalculate Bounds
Recalculate Lightmap Scale
Object Size 1
Reset Object Size
LOD 0 15744 Triangles - 1 Sub Mesh(es)
Transition (% Screen Size) 60
Set to Camera 1.44 m
Renderers
LOD_0 (Mesh Renderer) 15744 Tris 1 Sub Mesh(es)
LOD 1 3936 Triangles (25.00% LOD0) - 1 Sub Mesh(es)
Transition (% Screen Size) 30
Set to Camera 2.89 m
Renderers
LOD_1 (Mesh Renderer) 3936 Tris 1 Sub Mesh(es)
LOD 2 0 Triangles (0.00% LOD0) - 0 Sub Mesh(es)
Transition (% Screen Size) 10
Set to Camera 8.66 m
Renderers
None (Renderer) 0 Tris 0 Sub Mesh(es)
Add Component
LODModelTest

If your model needs more than three levels, you can right-click the timeline and choose **Insert Before** to add another LOD section.

Unity Insight: LOD bias and quality settings

The LOD Group transition thresholds define when Unity swaps between levels, but there is a global multiplier that shifts all of those thresholds at once: the **LOD Bias** setting in **Quality Settings**.

An LOD bias of **2.0** (the default in many quality presets) tells Unity to use higher-detail LOD levels for longer. The object must be twice as far away before Unity switches to a cheaper mesh. A bias of **1.0** uses the thresholds exactly as configured, while a bias of **0.5** switches to lower detail sooner.

This is important because the LOD Group Inspector shows distances adjusted by the active bias. If your transitions look correct in the Editor but behave differently on a target device, check whether that device uses a different Quality Settings level with a different LOD bias.

Learn more: LOD Group (https://docs.unity3d.com/6000.3/Documentation/Manual/class-LODGroup.html)

Once the renderers are assigned, move the camera in the Scene view to preview the behavior. As the camera pulls back, Unity switches from LOD 0 to LOD 1, then to LOD 2, and eventually to the culled state.

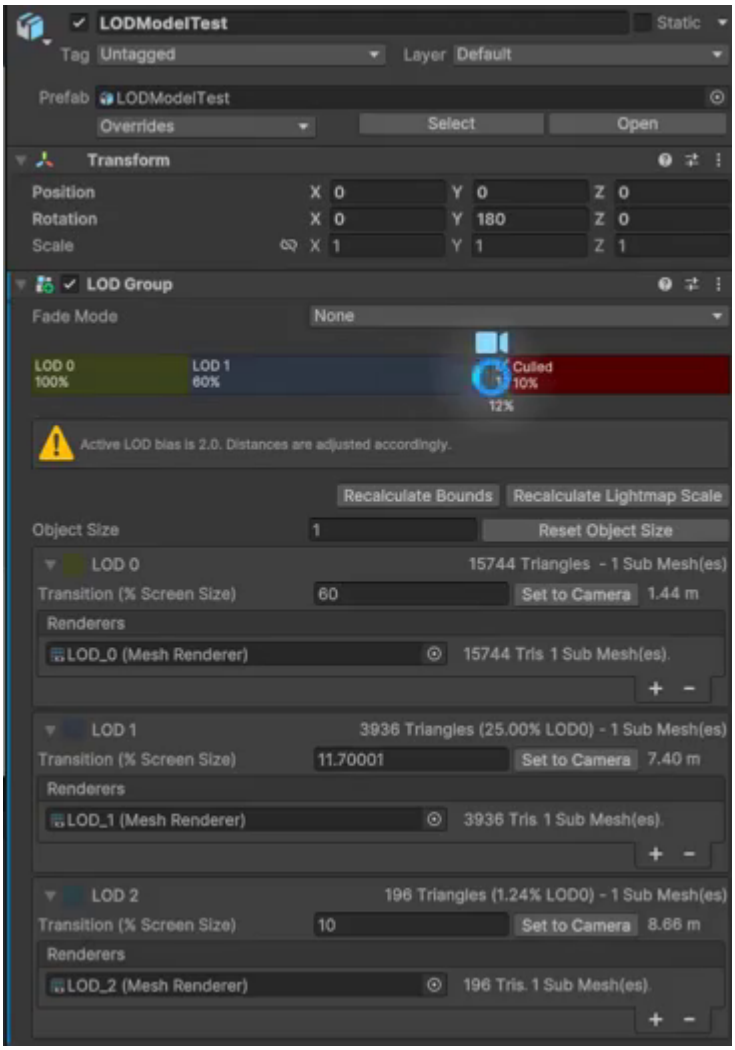
The culled state is especially useful for small props or repeated environment objects. If an object is so far away that it contributes nothing meaningful to the image, there is no reason to render it at all.

Adjusting Transition Points

The default transition thresholds are only a starting point. In practice, you will often need to tune them so the swaps happen at more natural distances.

You can drag the borders between LOD sections in the Inspector to change when each level becomes active. The small camera icon in the LOD Group

preview is also useful because it lets you scrub through the transitions and see exactly where each swap occurs.

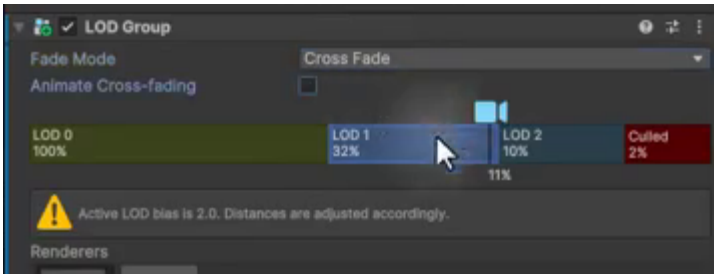


This tuning step matters because poor thresholds can make the object change too early, too late, or too noticeably.

Enabling Cross Fading

By default, LOD transitions happen instantly. That means the mesh simply pops from one version to another. Depending on the model and material, this can be obvious.

To make the change less noticeable, set **Fade Mode** to **Cross Fade** in the LOD Group. You can also enable **Animate Cross-fading**, which lets Unity handle the fade automatically as the camera moves.

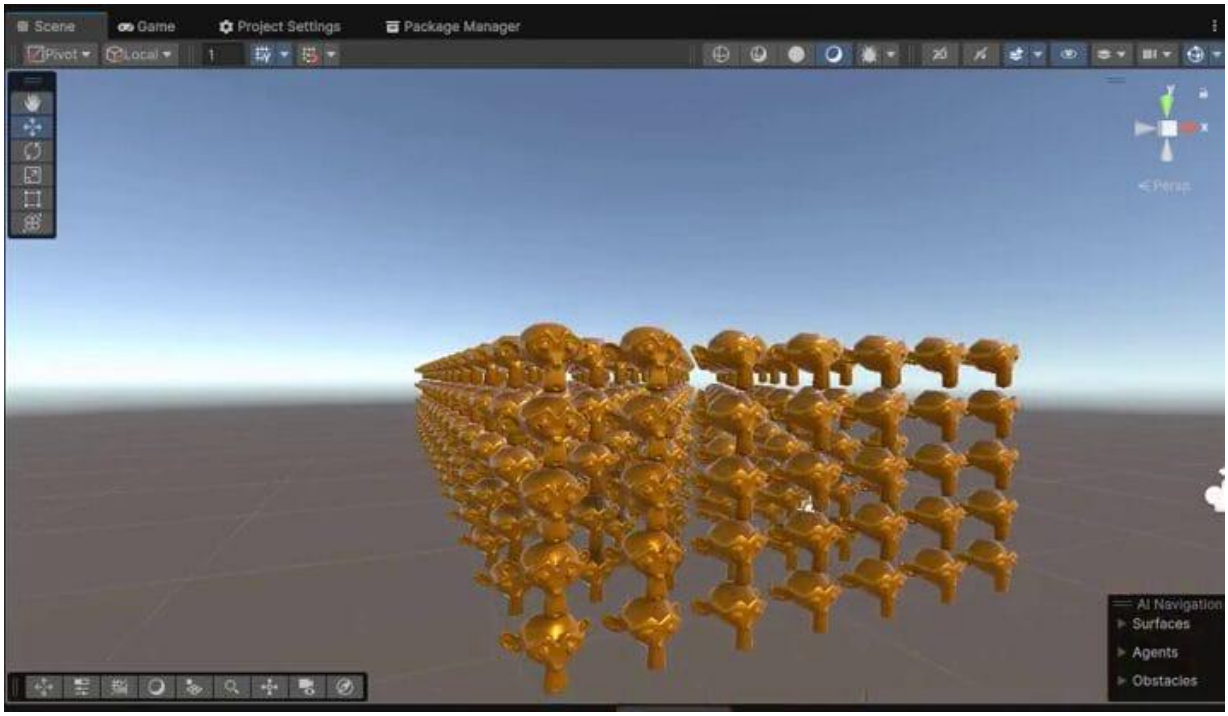


Cross fading is especially helpful when the object has reflective materials or a silhouette that changes noticeably between levels. In foggy scenes, or with more matte materials, LOD transitions are often much harder to detect.

Replacing Existing Scene Objects

Once the prefab is configured, the final step is to replace the original non-LOD versions in the scene. This gives you the optimization benefit without changing the visible layout of the level.

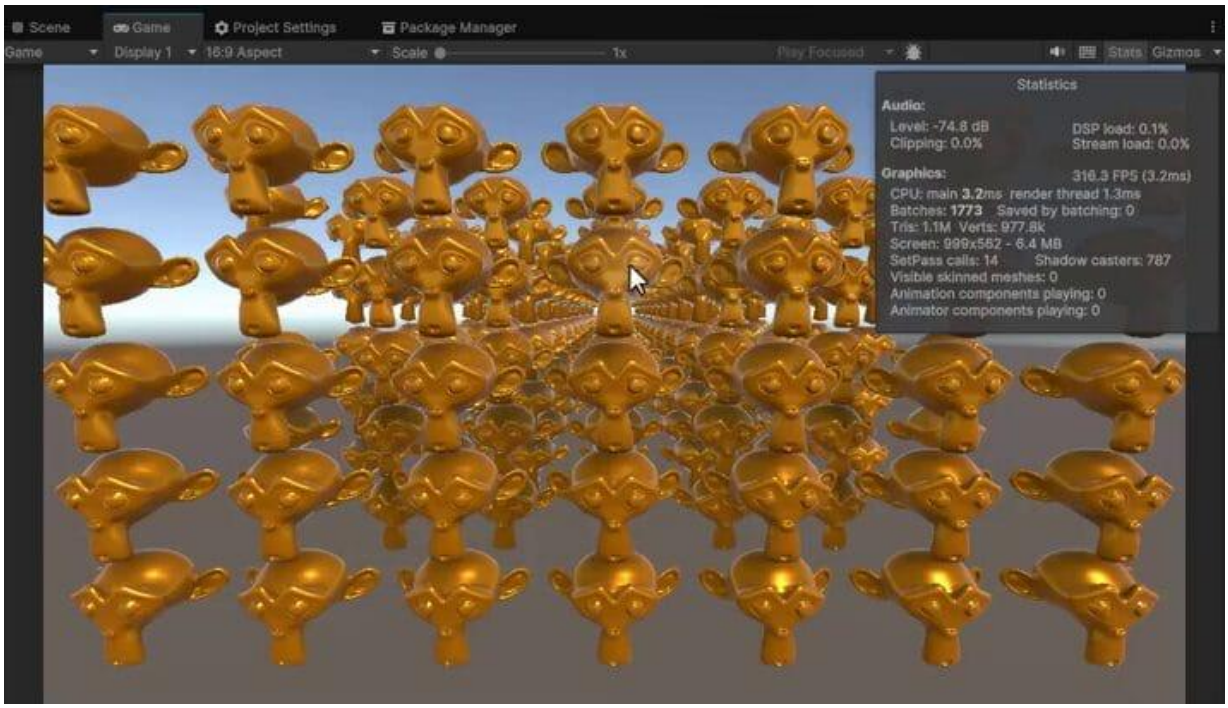
Let’s replace the old prefab instances in the Hierarchy by using **Prefab > Replace** to swap them for the LOD-enabled prefab.



Visually, the scene should look almost identical. The difference is that each object now adjusts its complexity based on its screen size.

Measuring the Performance Improvement

As with any optimization, it is important to compare before and after results. In this example, replacing the original prefabs with LOD-enabled versions reduces the triangle count from nearly **28 million** to about **1.1 million**.



That is roughly a **30x reduction** in triangle count while preserving almost the same visual result from the player's perspective. This is why LOD is such a standard technique in 3D games, especially in open environments.

Benefits and Trade-offs of LOD

LOD groups are widely used because they provide a strong balance between visual quality and rendering cost. Close objects remain detailed, while distant objects become cheaper to draw.

There is, however, an important trade-off: all LOD mesh versions remain loaded in memory. Only one is rendered at a time, but Unity still needs access to every level so it can switch between them. In other words, LOD reduces GPU workload but increases memory usage.

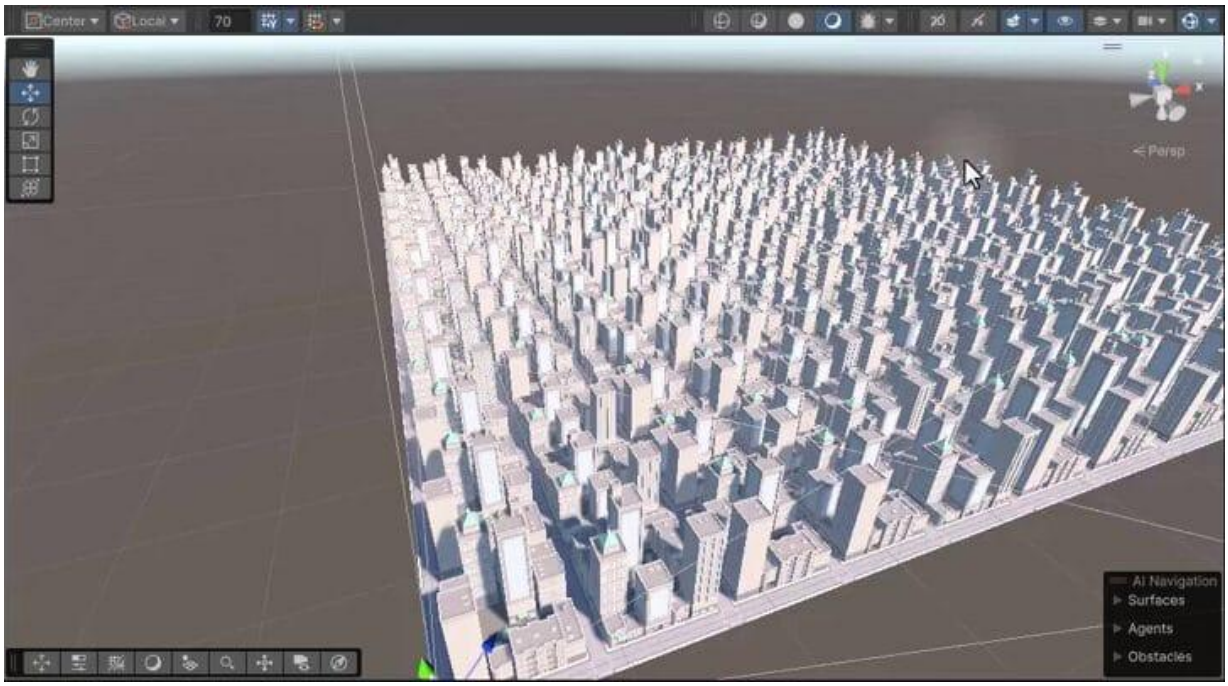
That trade-off is often worthwhile, but it should still be measured. As with static batching and baked lighting, the best optimization is the one that fits your target hardware and your actual bottleneck.

Using Occlusion Culling

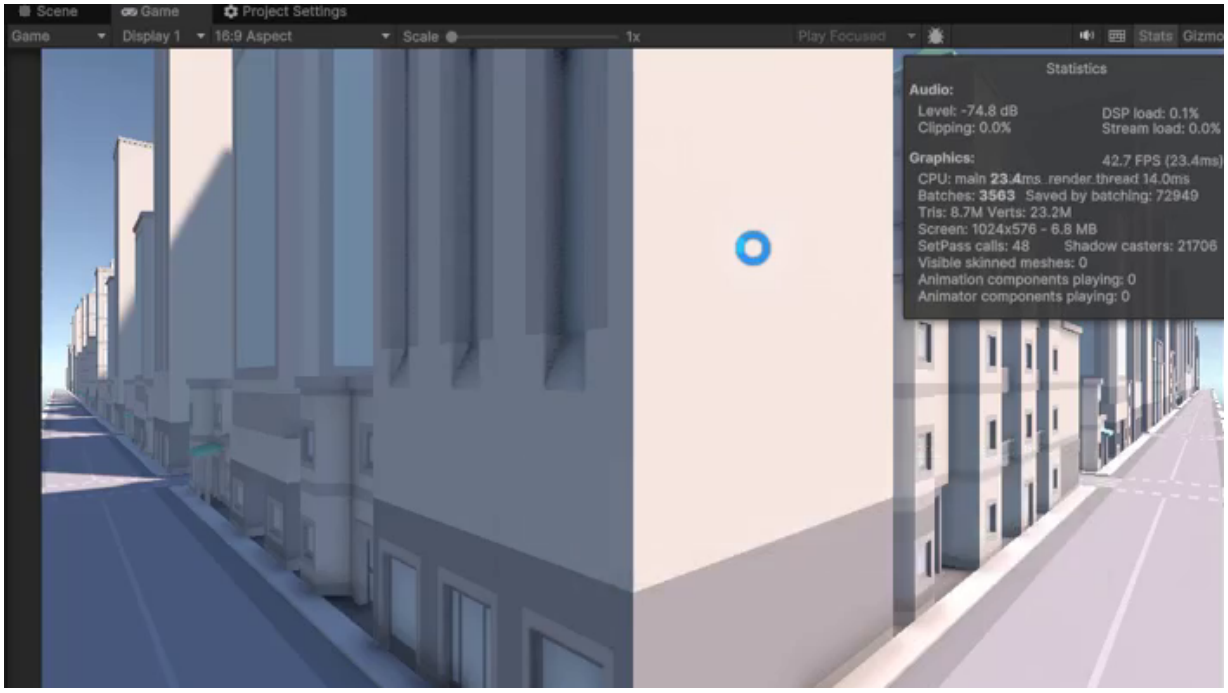
LOD reduces the cost of visible objects by simplifying distant meshes. Occlusion culling solves a different problem entirely: objects that are inside the camera's viewing cone but completely hidden behind other geometry. Unity already skips objects outside the camera's view (frustum culling), but it does not automatically skip objects that are blocked by a wall or building. Occlusion culling fills that gap by precomputing visibility data so Unity can skip hidden objects at runtime.

Why Occlusion Culling Matters

Consider a large city scene with hundreds of buildings. From a street-level camera, only a small portion of those buildings may be visible, yet Unity can still render a large amount of hidden geometry if it falls inside the camera frustum.



In play mode, the cost becomes easier to measure. Even though only a limited number of buildings are visible to the player, the scene may still render millions of triangles.

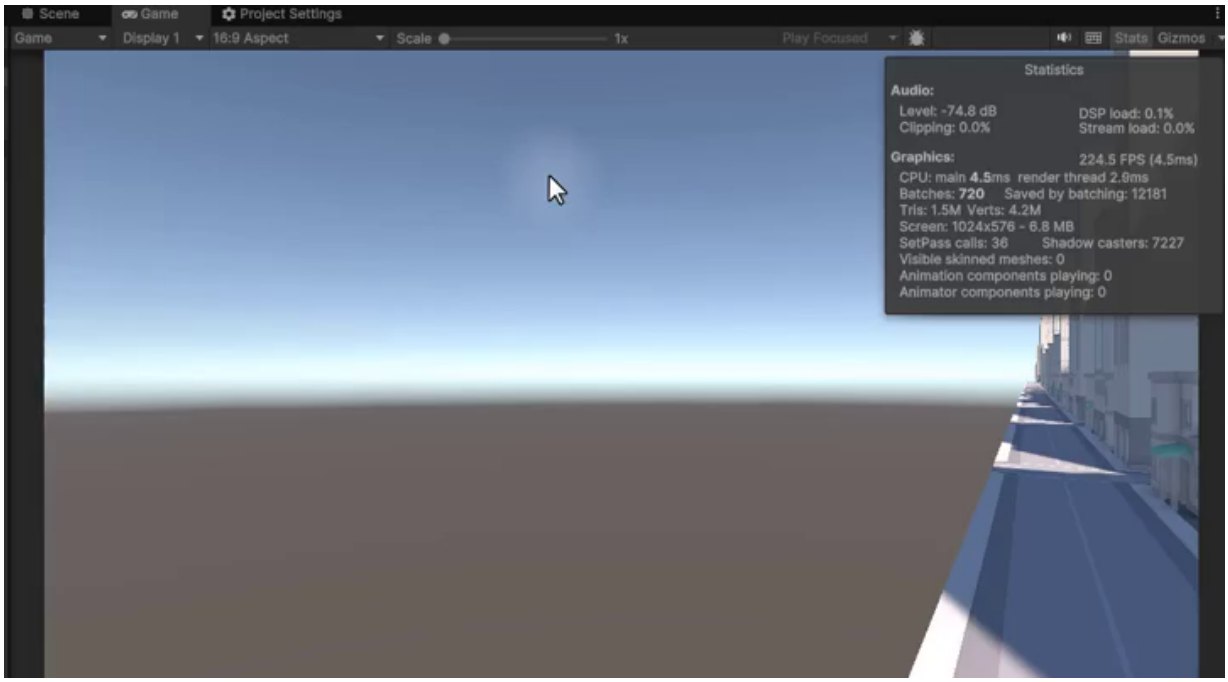


In this example, the scene renders about **8.7 million triangles** while running at roughly **42 FPS**, even though most of that geometry is blocked from view.

Frustum Culling and its Limits

To understand what occlusion culling adds, it helps to first see what Unity already does on its own. Unity performs **frustum culling** automatically, which means any object outside the camera's viewing volume is not rendered.

If the camera turns away from the city, the triangle count drops because the buildings are no longer inside the frustum.



Frustum culling is useful, but it only checks whether an object is inside the viewing cone. It does not check whether something else is blocking it. At street level, for example, a nearby building can hide dozens of buildings behind it, yet Unity still processes all of those hidden meshes along with their lighting and shadows.

The reason Unity does not handle this automatically is cost. Determining whether one object is hidden behind another is significantly more expensive than simply checking whether it falls inside the camera frustum, which is why it requires a precomputed bake step.

What Occlusion Culling Does

Occlusion culling extends Unity's visibility system by skipping objects that are hidden behind other objects. The concept is similar in spirit to baked lighting: instead of baking light data, Unity bakes visibility data. At runtime, it can then quickly look up which objects should be rendered from a given area of the scene, without performing expensive per-frame visibility calculations.

Unity Insight: How occlusion culling works internally

Unity's occlusion culling system divides the scene into a grid of cells during the bake step. For each cell, it precomputes which objects are visible from that region of space. At runtime, Unity looks up the cell that contains

the camera and uses the precomputed data to decide which objects to render.

This is why the bake settings matter. **Smallest Occluder** controls the minimum size of objects that can block visibility. If this value is too large, small walls or props will not act as occluders, and Unity will render geometry that could have been hidden. If the value is too small, the bake takes longer and the data becomes larger.

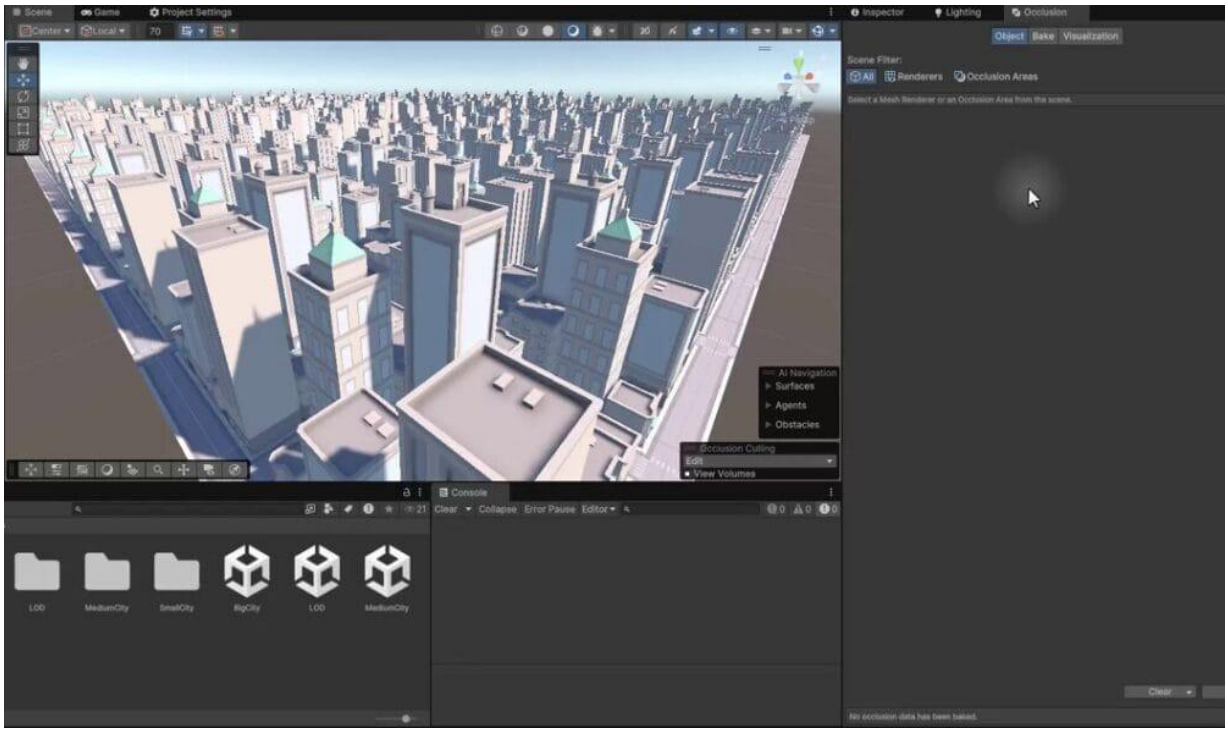
Smallest Hole determines the minimum gap the system accounts for. Smaller values mean tighter culling (fewer false positives), but also slower bakes. In practice, a value of 0.25 works well for most architectural scenes.

The baked data is stored as part of the scene and loaded at runtime. Unlike frustum culling (which is purely geometric and automatic), occlusion culling requires this precomputed data to function.

Learn more: Occlusion culling (https://docs.unity3d.com/6000.3/Documentation/Manual/OcclusionCulling.html)

Opening the Occlusion Culling Window

To set up occlusion culling, open **Window > Rendering > Occlusion Culling**. This window contains the tools needed to mark objects, configure bake settings, and visualize the results.



The window is divided into three main tabs:

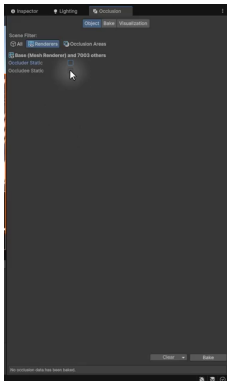
- **Object**
- **Bake**
- **Visualization**

Each tab serves a different part of the workflow.

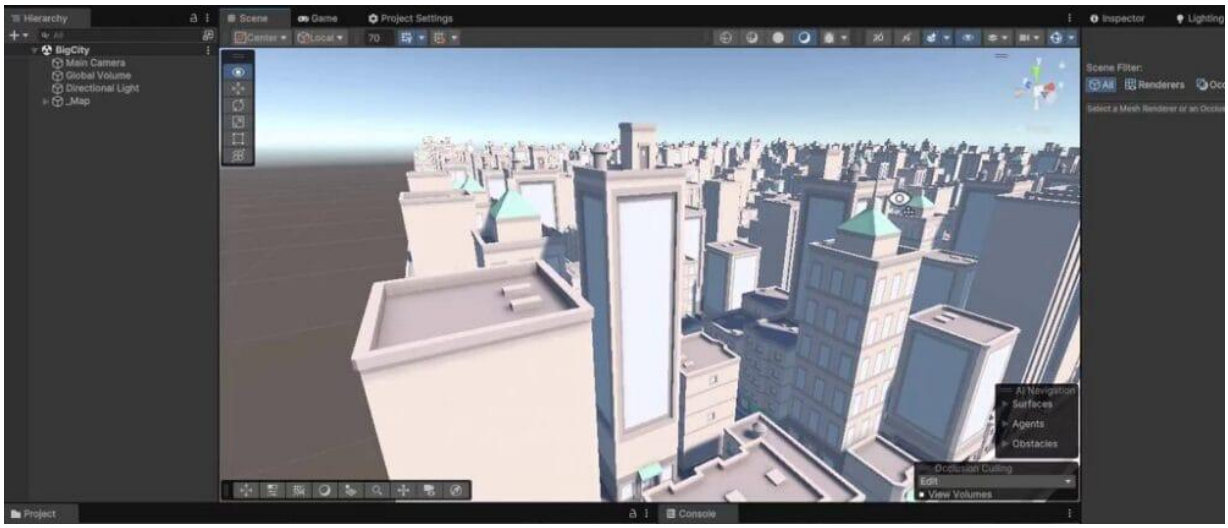
Configuring Objects for Occlusion Culling

Before baking, Unity needs to know which objects should participate. The **Object** tab controls this. We want to set the Scene Filter to **Renderers**, then select the relevant objects in the Hierarchy. For those selected objects, enable the following options:

- **Occluder Static**: The object can block other objects from view.
- **Occludee Static**: The object can itself be hidden by other objects.



In a city scene, most buildings should have both enabled. That way, a building can hide other buildings and also be hidden when another structure blocks it.



Configuring Bake Settings

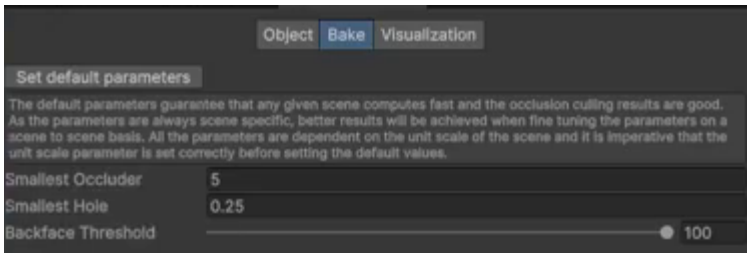
The **Bake** tab controls how finely Unity divides the scene when computing visibility. Getting these values right affects both the accuracy of culling and the stability of the result, so expect some experimentation.

The three main settings are:

- **Smallest Occluder**: The minimum size an object must be to block other objects

- **Smallest Hole**: The minimum gap the camera should be able to see through
- **Backface Threshold**: Controls how aggressively backfaces are removed during the bake

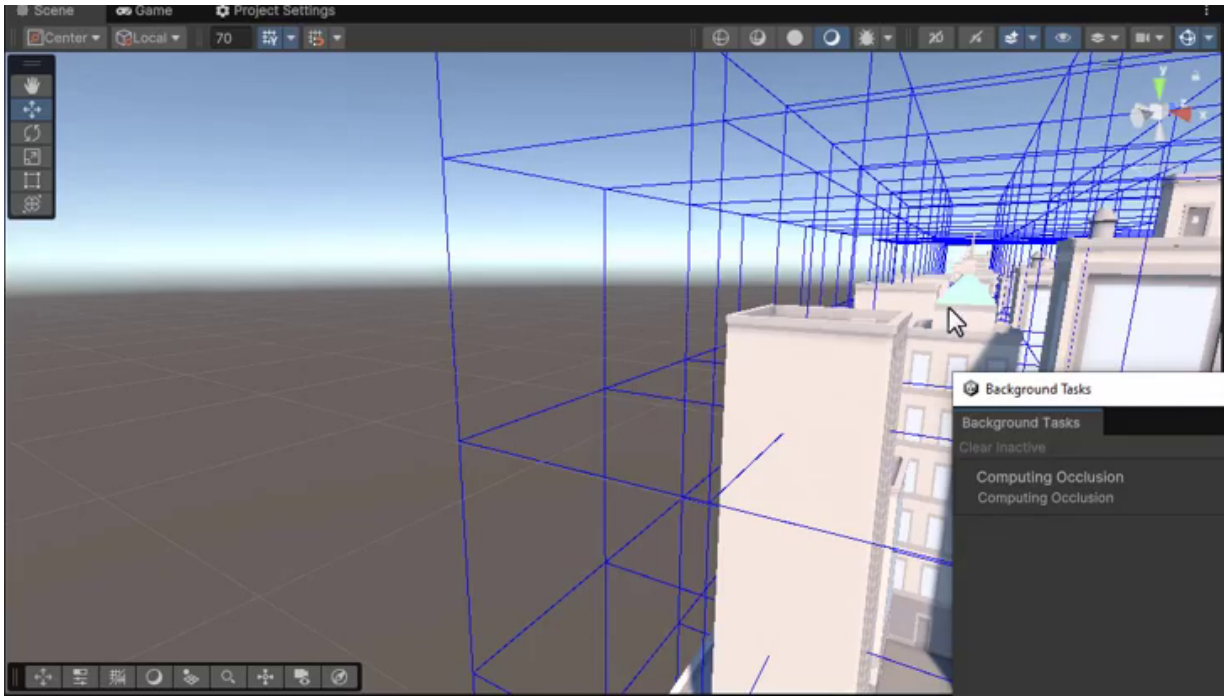
For a city scene, a **Smallest Occluder** value of **5** is a reasonable starting point. This prevents very small objects from acting as occluders, which could otherwise cause distracting pop-in.



Baking Occlusion Data

Once the objects and bake settings are ready, click **Bake** at the bottom of the Occlusion Culling window. Unity will begin generating visibility cells across the scene.

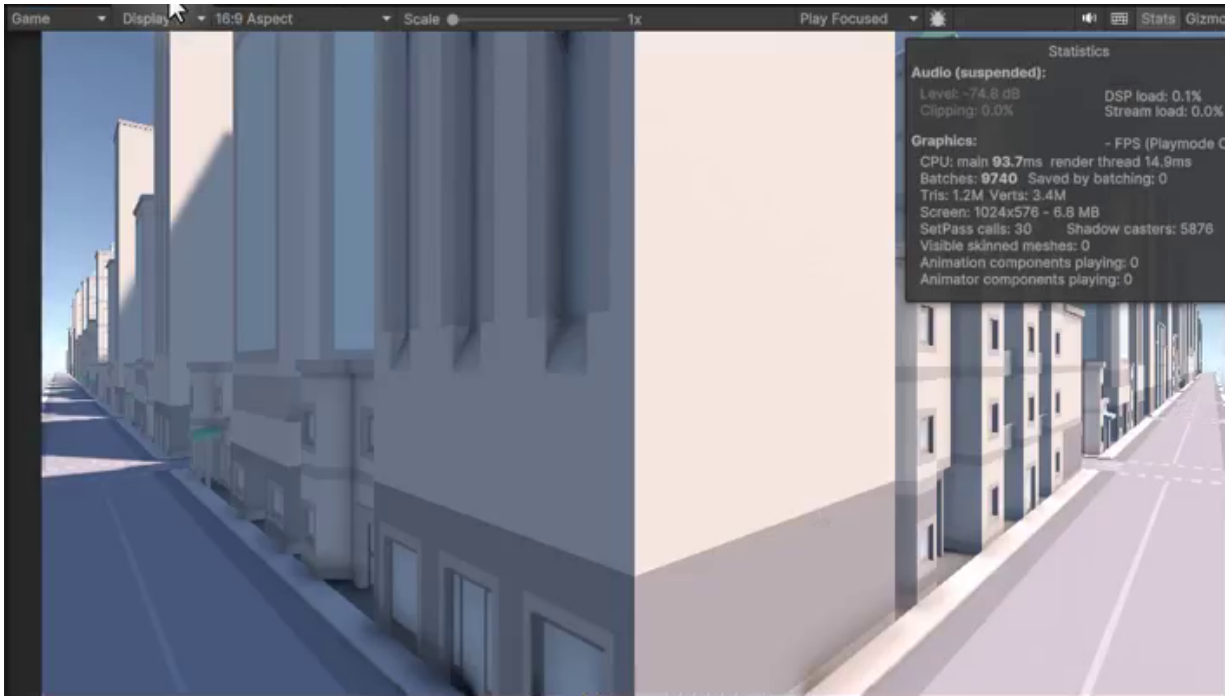
During the bake, you can see the process visualized in the Scene view. Larger and more complex scenes take longer, while smaller scenes finish more quickly.



Viewing the Baked Result

After the bake completes, the Scene view updates to reflect the visibility data. Objects that are hidden from the current viewpoint may disappear in the Scene view because Unity is now showing the occlusion result.

The Game view still shows the final camera output as normal. If you want to see the full scene again in the Scene view, disable the occlusion visualization from the Scene view controls.



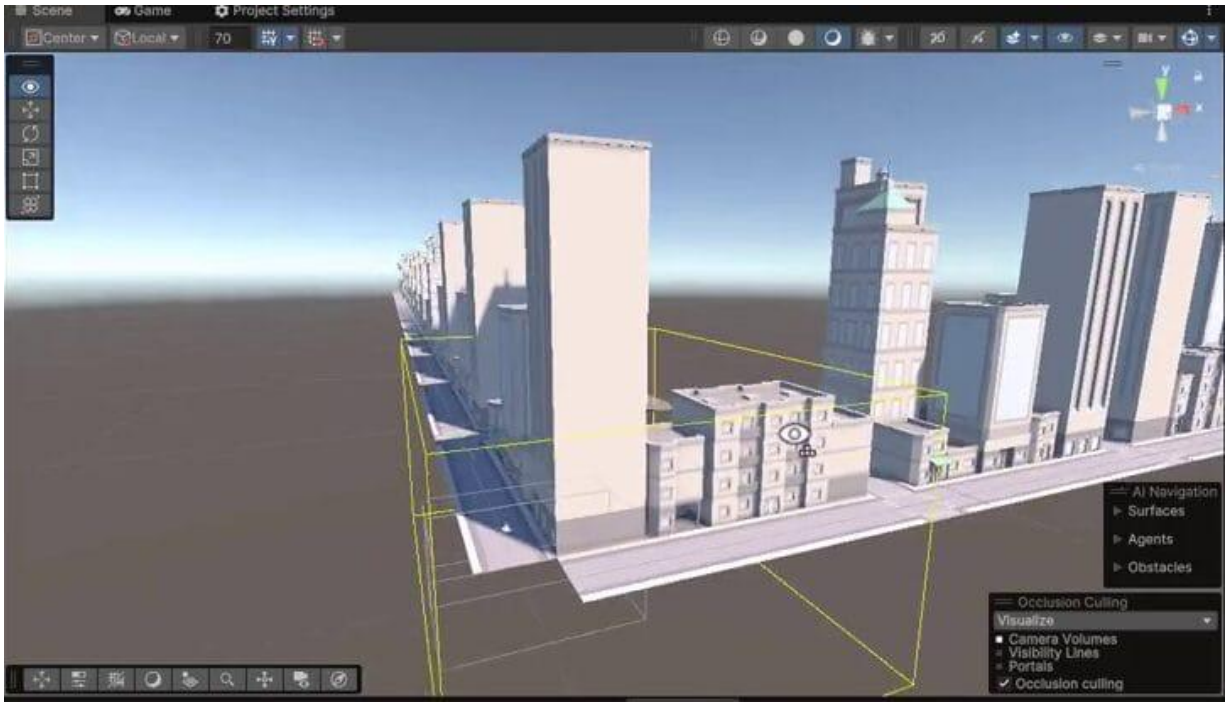
Using the Visualization Tab

The **Visualization** tab helps you inspect how the baked occlusion system behaves. This is useful when you want to confirm that Unity is culling the right objects and not introducing obvious errors.

The available visualization options include:

- **Camera Volumes**
- **Visibility Lines**
- **Portals**

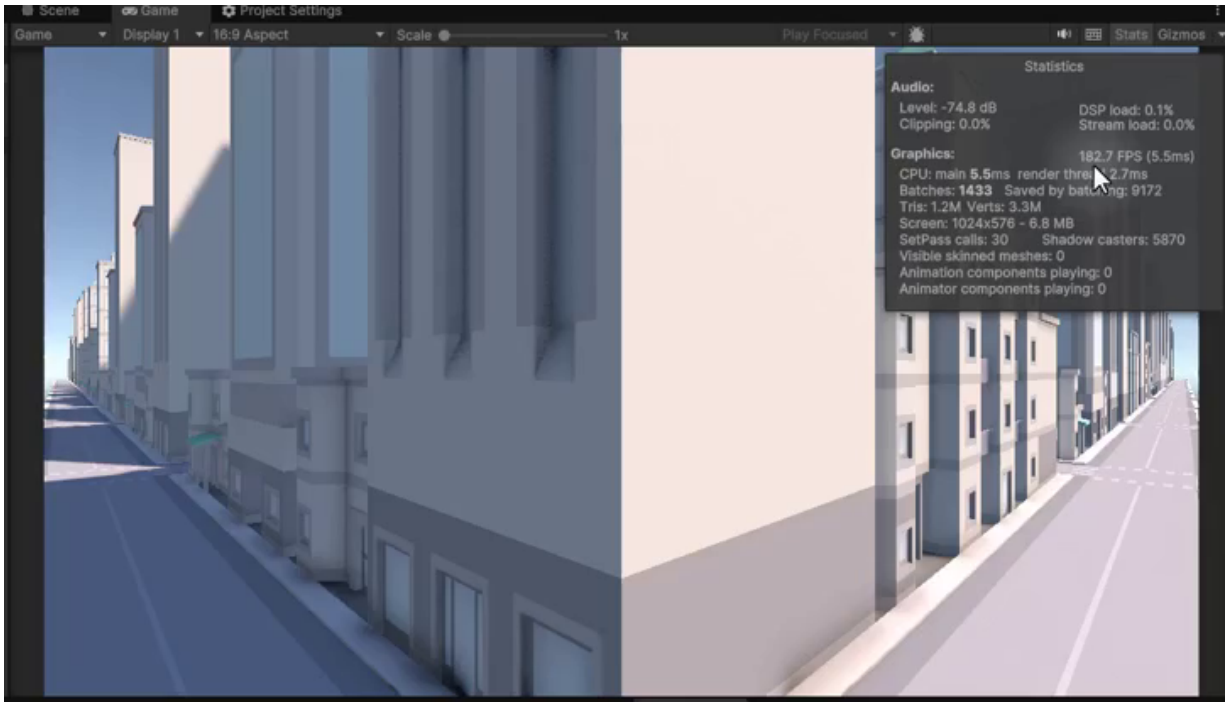
By selecting the camera and moving it through the scene, you can watch how the visible set of objects changes based on position.



Measuring the Performance Improvement

Once occlusion culling is enabled and baked, the performance gain can be substantial in dense scenes.

In this example, the scene improves from roughly **70 FPS** while rendering about **8 million triangles** to around **180 FPS** while rendering only **1.2 million triangles**.



As the camera rises and more of the city becomes visible, performance drops again because more geometry genuinely needs to be rendered. When the camera returns to street level, performance improves because more of the scene is hidden behind buildings.

Reducing Pop-In

If objects appear to pop in and out too aggressively while the camera moves, the bake settings are likely too coarse. Two adjustments usually help:

- Decrease **Smallest Occluder** so smaller objects can participate as occluders, giving the system more geometry to work with when deciding what is hidden.
- Decrease **Smallest Hole** so the system becomes more sensitive to narrow gaps, reducing cases where it incorrectly hides visible objects.

After changing these values, bake the scene again and test the result. As with LOD thresholds, this usually requires some iteration to find the best balance between performance and visual stability.

When Occlusion Culling is Most Useful

Occlusion culling works best in scenes where the player's view is frequently blocked by solid geometry.

Good candidates include:

- Dense city scenes
- Indoor environments
- Corridor-based levels
- Room-based layouts

It is less useful in wide open environments where most of the world is visible at once. In those cases, LOD is usually the more important optimization because the problem is not hidden geometry, but the cost of rendering distant visible geometry.

Summary

This chapter covered two major scene-visibility optimizations in Unity: **Level of Detail** and **occlusion culling**.

With **LOD**, the key idea is to reduce mesh complexity as objects become smaller on screen. By preparing multiple mesh versions and configuring them with an **LOD Group**, you can preserve close-up quality while dramatically lowering triangle counts at distance. The trade-off is increased memory usage, since all LOD versions remain loaded.

With **occlusion culling**, the goal is to stop rendering objects that the player cannot see because other geometry blocks them. After marking objects correctly and baking visibility data, Unity can skip large amounts of hidden geometry at runtime. This is especially effective in dense city scenes and enclosed environments.

Together, these techniques help ensure that Unity spends rendering time only where it matters: on objects that are both visible and important to the final image. In the next chapter, we will bring the book to a close by reviewing the broader optimization workflow and how these techniques fit together in real projects.

Optimization Challenge

Throughout this book, each chapter focused on a single optimization technique in isolation. That is a good way to learn individual tools, but real-world optimization rarely works that way. In practice, you need to combine multiple techniques, apply them in the right order, and measure their combined effect.

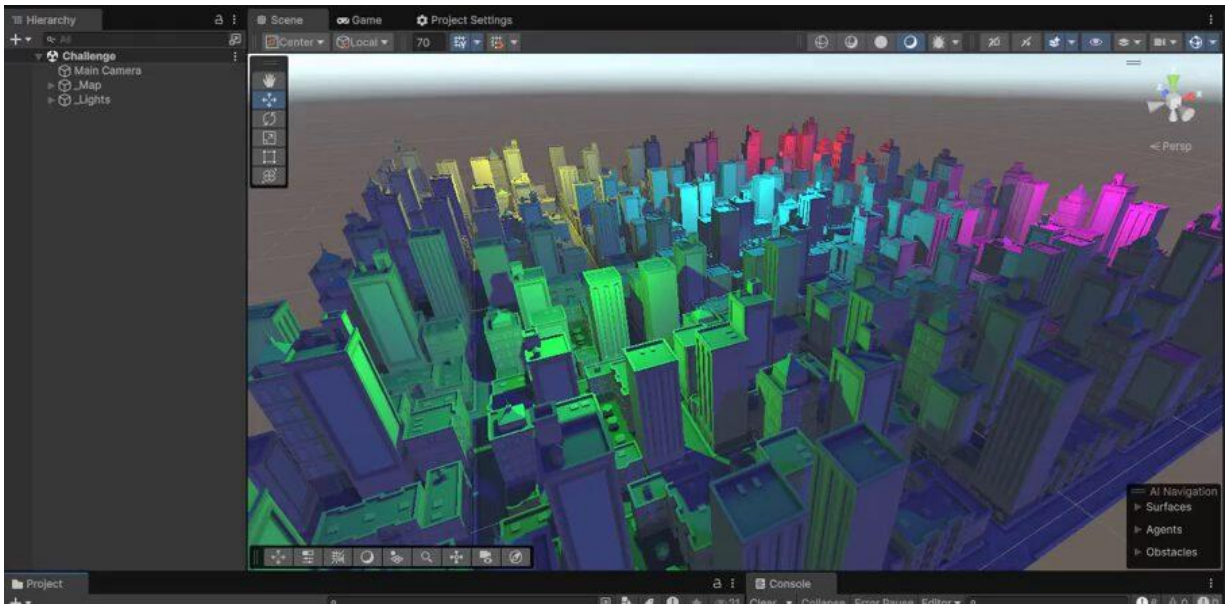
This chapter brings everything together in a single capstone exercise. You will start with an intentionally inefficient Unity scene, identify its major bottlenecks, and improve performance through a series of targeted changes. The goal is to practice the full optimization workflow from measurement to resolution, just as you would in a real project.

The Challenge Scene

Open the **Challenge** scene in the project (which you can download from the introductory part of this book). It contains a dense city environment with many colorful buildings and a lighting setup designed to create a clear rendering bottleneck.

The scene is organized around three root objects:

- Main Camera
- _Map
- _Lights

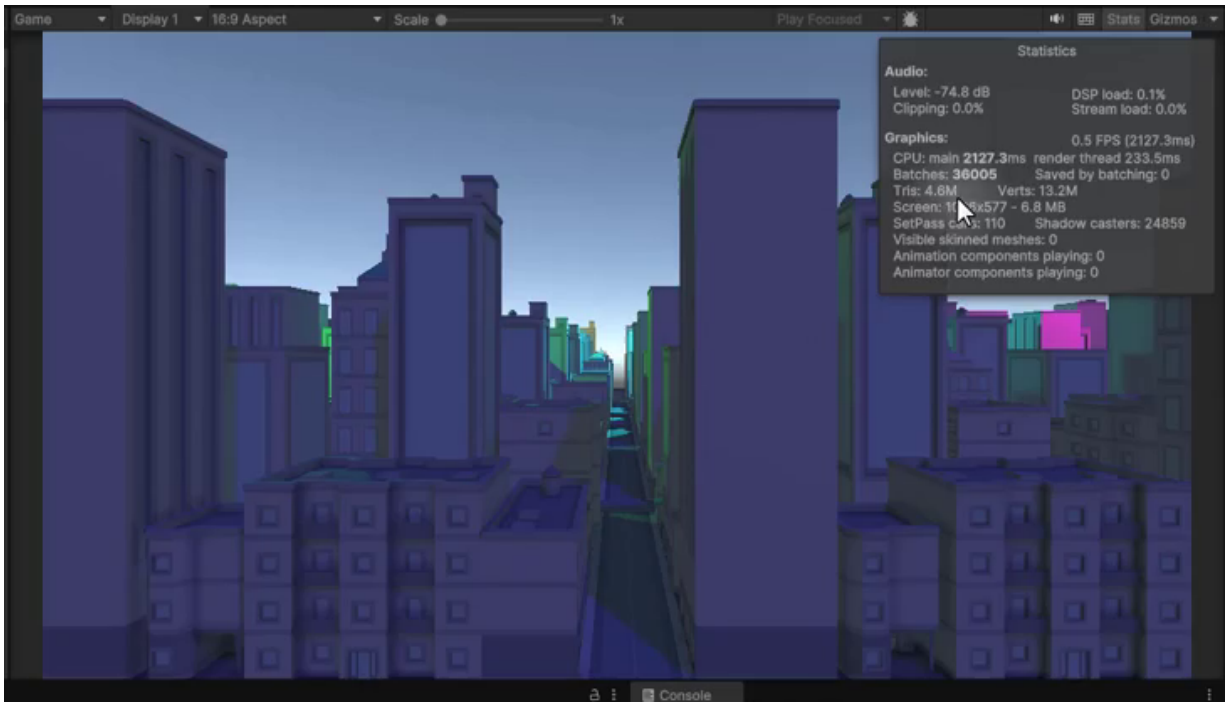


The scene is visually simple enough to read clearly, but dense enough to expose several common rendering problems at once. That makes it a good summary exercise for the book.

Establishing the Baseline

Before changing anything, it is important to measure the scene in its original state. This gives us a baseline so we can judge whether each optimization is actually helping.

Enter Play mode and open the **Statistics** panel in the Game view. The scene begins in a very inefficient state, with roughly **36,000 batches** being rendered per frame.



That number is the clearest warning sign. A batch count this high usually means the CPU is spending far too much time submitting rendering work to the GPU. The frame rate may still appear somewhat usable depending on the machine, but the scene is clearly leaving a large amount of performance on the table.

At this stage, several possible optimizations should come to mind:

- Reducing draw calls and batches through static batching or GPU instancing
- Adjusting the lighting setup to reduce shadow cost
- Using occlusion culling so hidden objects are not rendered
- Applying LOD where distant geometry does not need full detail

In this chapter, we will walk through a practical solution using three of those techniques: **static batching**, **lighting adjustments**, and **occlusion culling**.

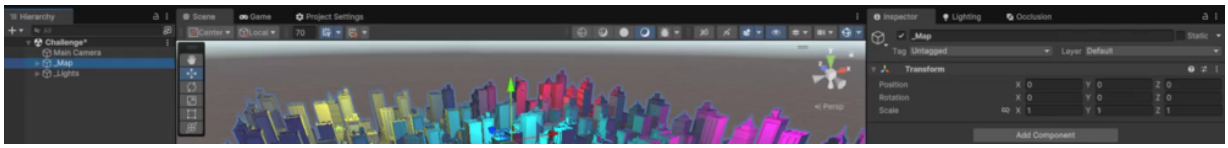
Feel free to tackle this on your own first if you'd like to truly make this a challenge. In the next section, though, we'll start going over how we can apply what we've learned to improve the scene.

Reducing Draw Calls with Static Batching

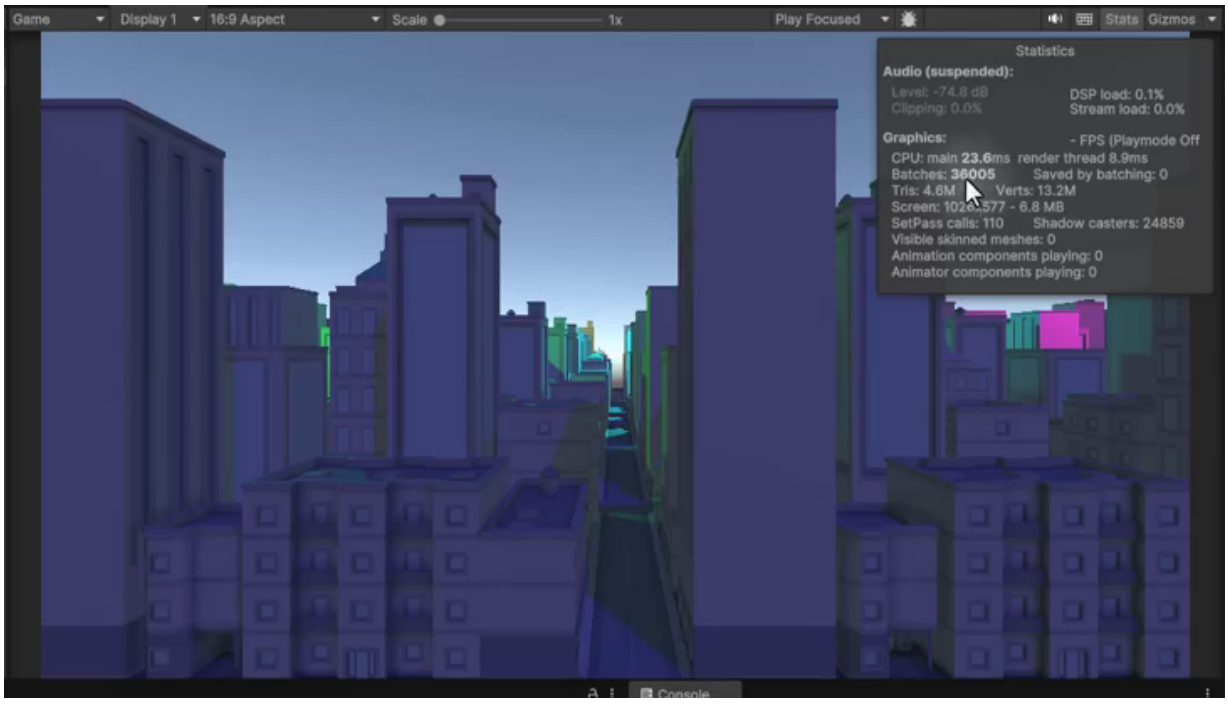
The most obvious bottleneck is the extremely high batch count. As you learned in the Static Batching chapter, Unity can combine compatible static meshes into larger batches, reducing the number of rendering commands the CPU must send each frame. Since the city geometry is made of stationary environment objects, static batching is a natural first optimization.

In this scene, the objects under **_Map** do not move during gameplay. That makes them good candidates for static batching.

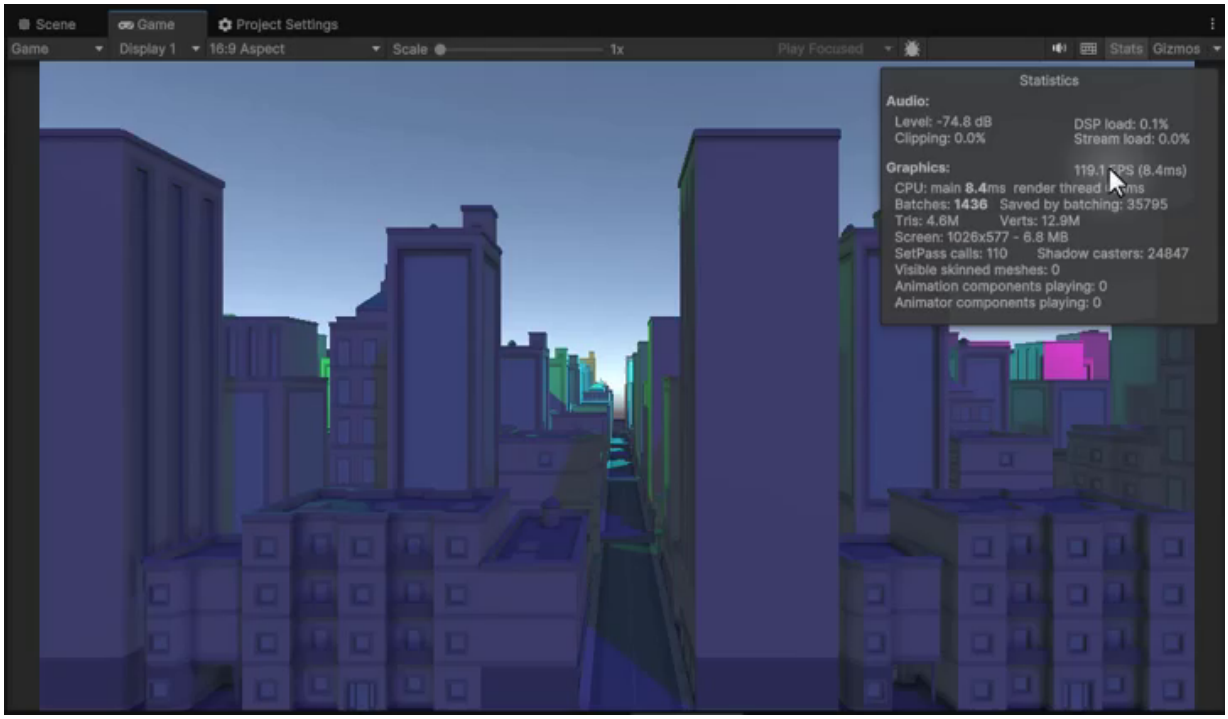
To enable it, select the models contained within the **_Map** object in the Hierarchy. In the Inspector, enable the **Static** checkbox near the top-right area. When Unity asks whether to apply the change to child objects, go ahead and confirm.



Before this change, the scene reports about **36,000 batches** per frame.



After entering Play mode again with static batching enabled, the batch count drops to roughly **1,400**.



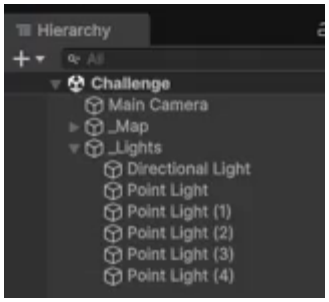
This is a dramatic improvement from a single change. In practical terms, it produces a gain of roughly **20 to 30 FPS**. More importantly, it confirms that draw call overhead was one of the scene's main bottlenecks.

Reducing Lighting Cost

Once the batch count is under control, the next major cost comes from lighting. As you saw in the Lighting chapter, real-time lights with shadow casting can add significant rendering overhead. The purpose of this step is not to remove lighting entirely, but to reduce the cost of shadows where they contribute the least visual value.

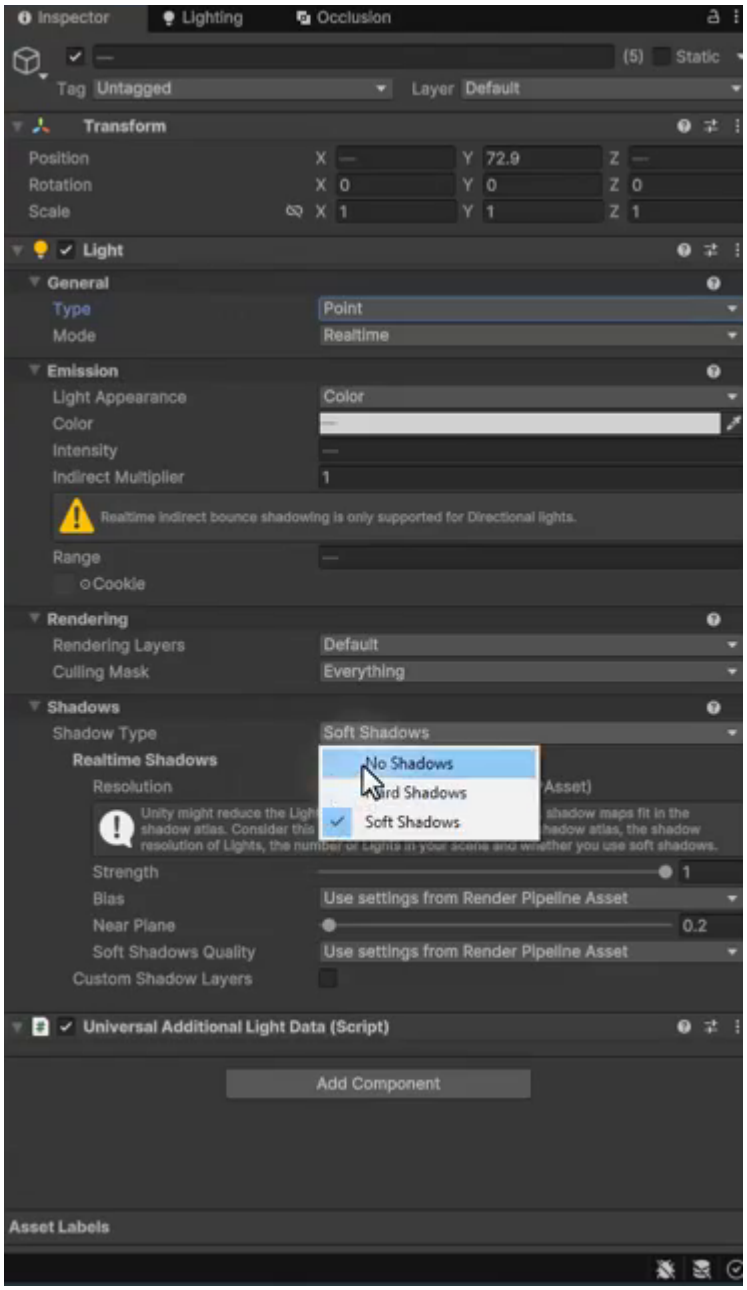
Under the **_Lights** object, the scene contains six real-time lights:

- One **Directional Light**
- Five **Point Lights**



By default, the Point Lights cast shadows, and their shadow resolution is set high enough to create unnecessary cost for this camera view. Since the scene is viewed from a distance, we can usually reduce or remove those shadows with little visible impact.

Select all of the Point Lights in the scene, then inspect their **Shadows** settings in the Inspector.


Inspector
Lighting
Occlusion
(5)
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Light
General
Type
Point
Mode
Realtime
Emission
Light Appearance
Color
Intensity
Indirect Multiplier
1
Realtime indirect bounce shadowing is only supported for Directional lights.
Range
Cookie
Rendering
Rendering Layers
Default
Culling Mask
Everything
Shadows
Shadow Type
Soft Shadows
Realtime Shadows
Resolution
No Shadows
Hard Shadows
Soft Shadows
Strength
1
Bias
Use settings from Render Pipeline Asset
Near Plane
0.2
Soft Shadows Quality
Use settings from Render Pipeline Asset
Custom Shadow Layers
Universal Additional Light Data (Script)
Add Component
Asset Labels

There are two sensible options here:

- Lower the shadow resolution from **High** to **Medium** or **Low**
- Disable shadows entirely by setting **Shadow Type** to **No Shadows**

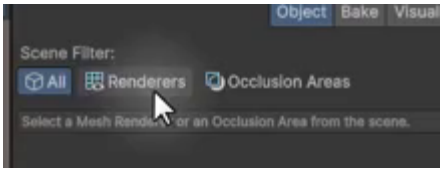
Lowering resolution preserves some shadow detail while reducing cost. Disabling shadows removes that cost more aggressively. In this challenge scene, disabling shadows on the Point Lights provides a strong performance gain while keeping the overall look acceptable.

This is a common trade-off in optimization work. Decorative or secondary lights often do not need to cast real-time shadows, especially when the player views the scene from farther away.

Reducing Hidden Rendering with Occlusion Culling

After improving batching and lighting, the next opportunity is visibility. As you learned in the Occlusion Culling chapter, Unity normally renders every object within the camera frustum, even if other geometry completely hides it. Occlusion culling solves that by baking visibility data ahead of time, allowing Unity to skip objects that are fully hidden behind other geometry at runtime.

Let's open the **Occlusion Culling** window. In the **Object** tab, set the **Scene Filter** to **Renderers**.



Then select the mesh objects in the scene. We need both of the following options enabled:

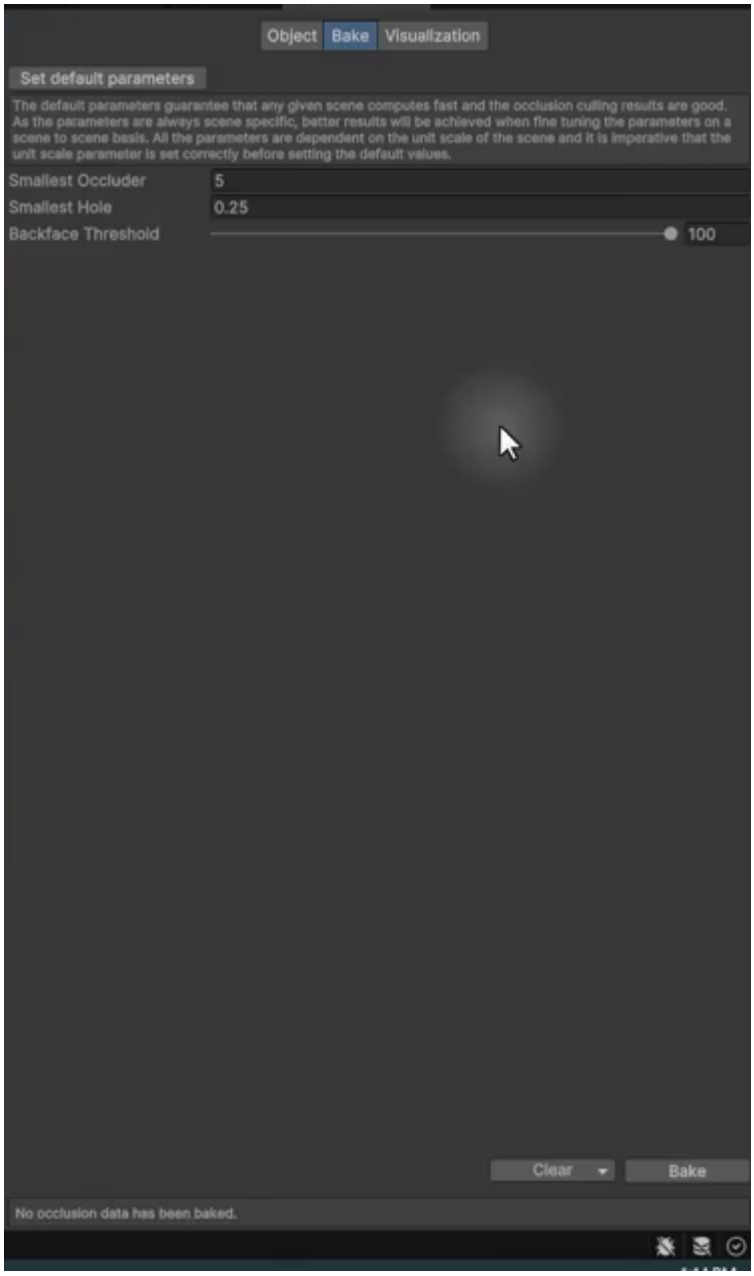
- **Occluder Static**
- **Occludee Static**

This tells Unity that the objects can both block visibility and be hidden by other objects.

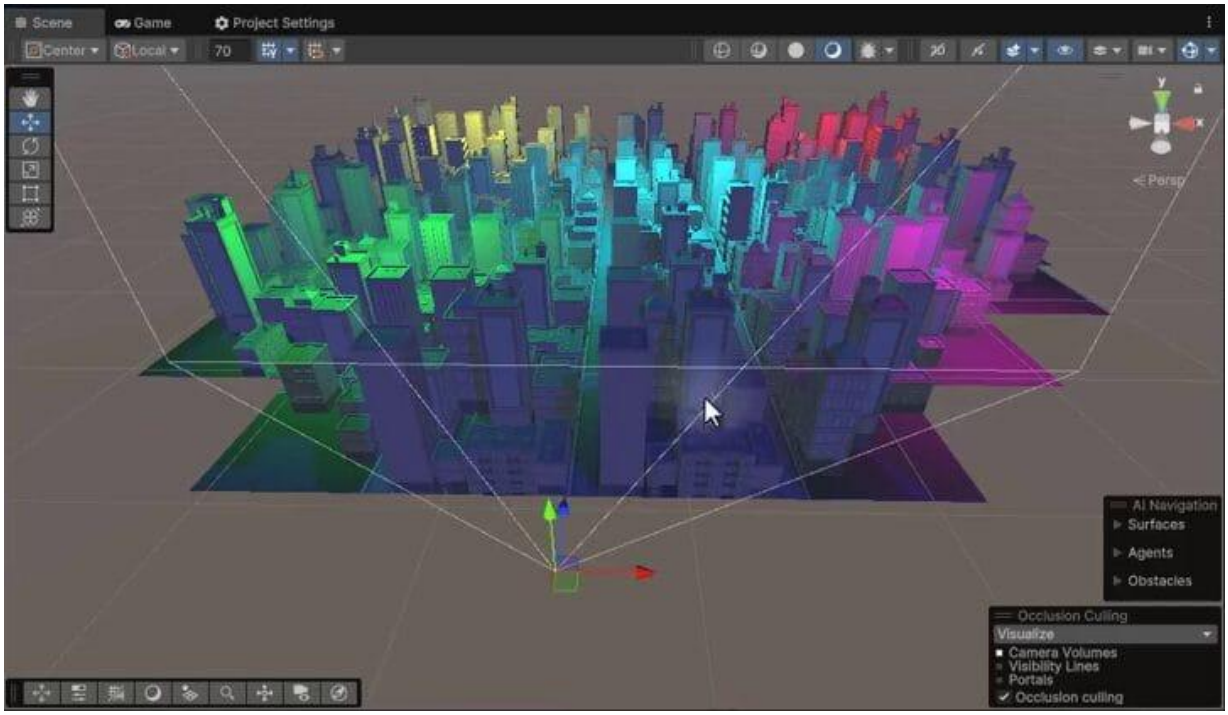
Next, switch to the **Bake** tab, which contains the main bake parameters:

- **Smallest Occluder**
- **Smallest Hole**
- **Backface Threshold**

For this scene, the default values are sufficient, so no special tuning is required. Click **Bake** to generate the occlusion data.


Object
Bake
Visualization
Set default parameters
The default parameters guarantee that any given scene computes fast and the occlusion culling results are good. As the parameters are always scene specific, better results will be achieved when fine tuning the parameters on a scene to scene basis. All the parameters are dependent on the unit scale of the scene and it is imperative that the unit scale parameter is set correctly before setting the default values.
Smallest Occluder
5
Smallest Hole
0.25
Backface Threshold
100
Clear
Bake
No occlusion data has been baked.

Once the bake completes, Unity can visualize which objects are currently considered visible from the camera's position.



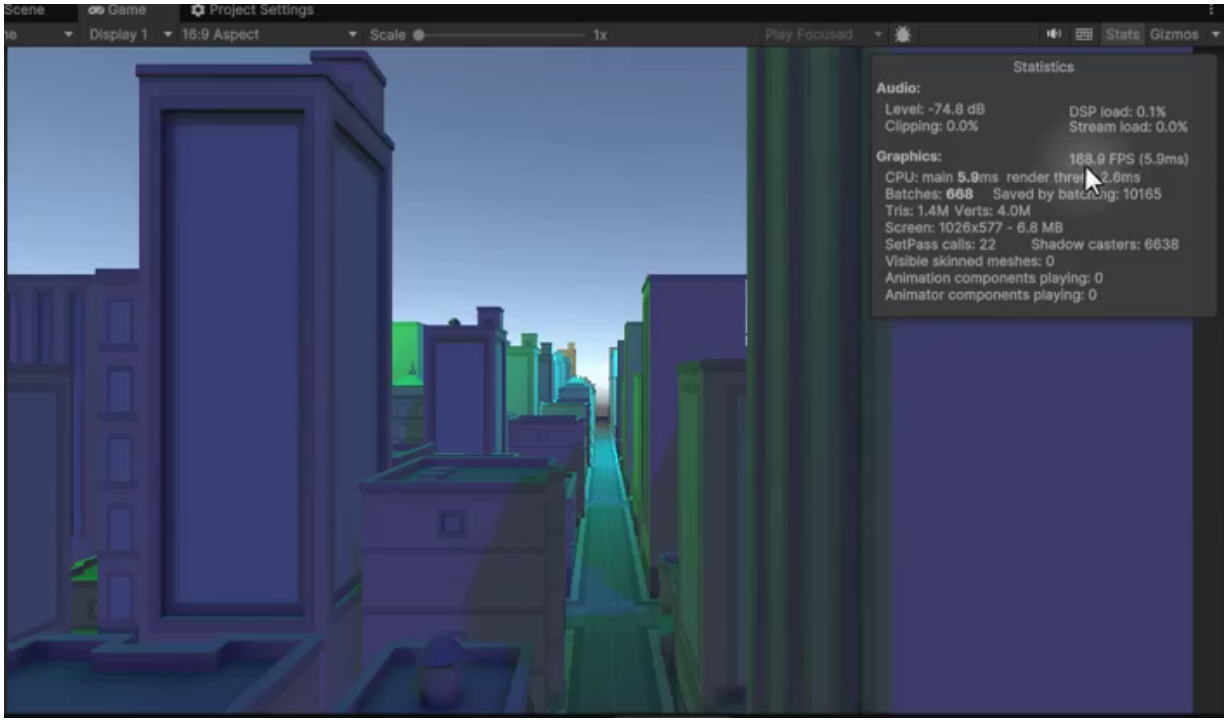
This is where occlusion culling becomes especially valuable in urban scenes. When the camera moves near a large building, the geometry behind that building can be excluded from rendering, which reduces GPU workload and often lowers the number of visible vertices and triangles substantially.

Final Results

With all three optimizations applied:

- Static batching on the environment
- Reduced or disabled shadows on the Point Lights
- Baked occlusion culling

… the scene performs dramatically better than it did at the start.



The optimized scene reaches close to **200 FPS**, with the batch count reduced to under **900** in the tested view. Compared to the original state, this is a major improvement in both CPU and rendering efficiency.

Just as importantly, the scene still looks broadly similar from the player's perspective. That is the ideal outcome of optimization: remove unnecessary cost while preserving the visual result as much as possible.

What this Challenge Demonstrates

This challenge reflects how optimization actually happens in real projects. There is rarely one magical setting that fixes everything. Instead, performance improves through a series of targeted decisions, each addressing a different bottleneck.

In this example, each technique addressed a distinct layer of the rendering pipeline:

- **Static batching** reduced CPU overhead from excessive draw calls, the same technique you first practiced in isolation earlier in the book

- **Lighting adjustments** lowered the cost of real-time shadows, applying the trade-offs you explored in the Lighting chapter
- **Occlusion culling** prevented hidden geometry from being rendered, using the baking workflow you learned in the Occlusion Culling chapter

Together, these three changes produced a much larger improvement than any one of them would have alone. That layered approach is the core skill this book has been building toward: measuring a scene, identifying where time is being spent, and choosing the right tool for each problem.

Summary

This chapter brought together static batching, lighting adjustments, and occlusion culling in a single practical scene. Starting from roughly 36,000 batches and under 1 FPS, you improved the scene to nearly 200 FPS and under 900 batches while maintaining a similar visual result.

The broader takeaway is the workflow itself. Effective optimization in Unity means measuring the scene, identifying the real bottleneck, and choosing the technique that addresses that specific cost. When those decisions are made carefully and supported by Unity's analysis tools, even a heavily unoptimized scene can improve dramatically.

Closing Words

Congratulations. You made it all the way through, and that matters.

Finishing a book on Unity optimization takes more than curiosity. It takes patience with Profiler data, willingness to experiment with settings, and the discipline to measure before and after every change. You put in that effort. You tested techniques, compared results, and built a practical understanding of how to make Unity projects run better. That is real progress, and you should feel proud of it.

What you Accomplished

Before looking ahead, it is worth pausing to recognize what you now know how to do. You did not just follow steps. You built a practical optimization skill set that many Unity developers take much longer to develop.

You learned how to use Unity's Profiler to measure CPU and GPU performance, and how to read the data it provides to find real bottlenecks instead of guessing. You used the Frame Debugger to step through individual rendering events and understand exactly what Unity draws each frame and why.

You practiced reducing draw calls with static batching, turning thousands of individual rendering commands into a fraction of that number. You explored the difference between real-time and baked lighting, learning when each approach makes sense and how shadow settings affect performance. You set up LOD groups so distant objects use simpler geometry, and you baked occlusion culling data so hidden objects stop consuming rendering resources.

Most importantly, you practiced the mindset behind optimization. Profiling, measuring, identifying bottlenecks, and making targeted improvements are valuable skills in every project, whether you are building a small prototype or a larger production. Optimization is not only about speed. It is about

understanding where your project spends time and resources, then making better decisions because of that knowledge.

By the end of this book, you have done something more important than learning isolated features: you have started thinking about performance as part of the development process. You can now look at a scene and consider not just how it looks, but how efficiently it renders. That is a meaningful shift, and it is one of the clearest signs that you are growing as a developer.

Where to go From Here

This final section is about momentum. The best next step is not to wait for the perfect idea. It is to keep building while what you learned is still fresh.

A great place to start is by applying what you learned to your own projects. Open a scene you have been working on, run the Profiler, and see what the data tells you. Try enabling static batching on your environment objects, or experiment with baked lighting instead of real-time lights. The goal is to practice making optimization decisions without a tutorial guiding every step.

You can also build a small test scene from scratch specifically to stress-test what you learned. Create a dense environment, add multiple light sources, and then optimize it using the full workflow from this book: profile, identify the bottleneck, apply the right technique, and measure again.

It is also a good time to spend more time with the official Unity documentation. Tutorials help you get moving, but documentation helps you become independent. When you read about a component, system, or API directly from the source, you start building the habit of finding answers on your own. The Unity Manual is here: Unity Manual (https://docs.unity3d.com/Manual/index.html). The scripting reference is here: Unity Scripting API (https://docs.unity3d.com/ScriptReference/).

You may also want to join the wider Unity and game development community. Seeing how other developers solve problems can accelerate your growth. Share your work, ask questions, read postmortems, and take

part in game jams if that sounds fun. Communities like Unity forums, local meetups, Discord servers, and online dev communities can make the learning process feel much less solitary.

As you continue, try to keep a few habits:

- Profile before optimizing. Always measure first.
- Test one change at a time so you know what actually helped.
- Revisit old projects and optimize them with fresh eyes.
- Build small test scenes to experiment with new techniques.
- Treat unexpected results as learning opportunities, not failures.

Those habits will carry you much farther than any single technique.

Given that Unity remains an industry standard, Zenva Academy (https://academy.zenva.com) provides an ongoing resource for mastering the engine. By utilizing a project-led methodology across 300+ courses and 40+ Learning Pathways, developers can continue to build their technical foundation and explore complex development techniques.

FinalThought

Optimization is not something you learn once and move on from. It is a skill that grows with every project.

Every time you open the Profiler, you get a little faster at reading the data. Every time you compare a before and after measurement, your instincts for spotting bottlenecks get sharper. You do not need to memorize every setting or technique. You just need to keep the habit of measuring, questioning, and improving.

You have already done the hardest part: you started, and you stayed with it. Now take what you have learned and apply it to the projects that matter to you. The next scene you optimize will go smoother than the last one did.