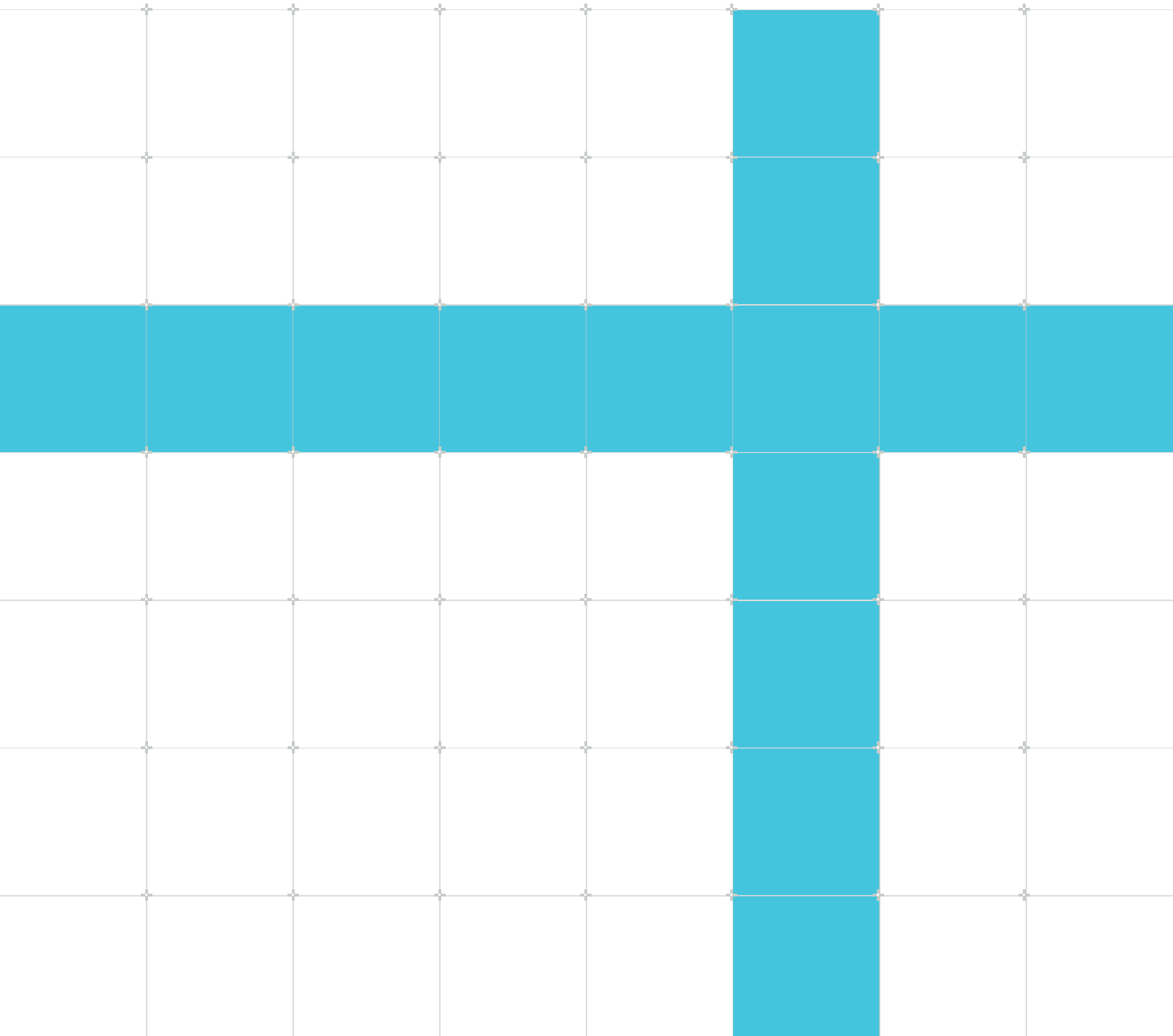
Arm Guide for Unity Developers - Optimization opportunities in Unity

Version 1.0

Issue 03
102314_0100_03_en



Arm Guide for Unity Developers - Optimization opportunities in Unity

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	12 January 2021	Non-Confidential	Initial release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Optimization is the process of taking an application and making it more efficient. For graphical applications, optimization typically means modifying the application to make it faster. For example, a game with a low frame rate might appear jumpy, which gives a bad impression and can make a game difficult to play. You can use optimization to improve the frame rate of a game, making it a better, smoother experience.

This guide introduces ways that you can optimize your Unity programs, especially their GPU usage. The guide groups optimizations into three chapters:

- Application processor optimizations
- GPU optimizations
- Asset optimizations

Before you begin

This guide is intended for experienced game developers who are already familiar with Unity.

This guide was last reviewed with Unity 2019.3

Understanding the optimization process

Before we learn about optimization techniques, let's review the general process you should follow when optimizing an application.

The optimization process is iterative. To find and remove performance problems, perform the following steps:

1. Take measurements of your application with a profiler. The profiler analyzes the measurements so that you can isolate and identify the source of any performance problem.
2. Locate the bottleneck by analyzing the profiler data
3. Determine the relevant optimization to apply
4. Verify that the optimization works
5. If the performance is still not acceptable, return to step 1 and repeat the process

Here is a brief example of the optimization process:

You have a game that does not have the performance you require. You can use a profiler to take measurements of your application. Profiling shows that the problem with the game is that it renders too many vertices, so you reduce the number of vertices in your meshes. Execute the game again to ensure that the optimization worked.

If the game is not performing as expected after you have completed the optimization process, you can restart the process by profiling the application again. This enables you to find out what else is causing problems.

The Ice Cave Demo

The Arm Guide for Unity Developers series often refers to the Ice Cave Demo. The Ice Cave demo is a demonstration application created by Arm that uses a number of optimized techniques to produce high-quality visual content for mobile devices.

You can see the Ice Cave demo in action on our YouTube channel, and read more about it on the blog post Virtual Reality: The Ice Cave and on our Unity Demos page.

Arm has started to release some of the effects used in our demos. You can find them in the Unity Asset Store.

2. Application processor optimizations

This section of the guide describes a few application processor optimizations that can improve the performance of your Unity programs.

Use coroutines

A coroutine is a function of type `IEnumerator` that can return control to Unity with a special `yield return` statement. You can call the function again later, and it resumes where it left off.

Use coroutines instead of Invoke()

The `Monobehaviour.Invoke()` method is a fast and convenient way to call a method in a class with a time delay. However, the method has the following limitations:

- `Monobehaviour.Invoke()` uses reflection in C# to find the method to call, which can be slower than calling the method directly.
- There are no compile-time checks on the method signature.
- You cannot supply more parameters.

The following code shows the `Invoke()` function:

```
public void Function()
{
        [...]
}
Invoke("Function", 3.0f);
```

An alternative method is to use coroutines. The following code shows calling coroutines through the `MonoBehaviour.StartCoroutine()` method:

```
public IEnumerator Function(float delay)
{
        yield return new WaitForSeconds(delay);
        [...]
}
StartCoroutine(Function(3.0f));
```

Changing from the `Monobehaviour.Invoke()` method to using coroutines provides more flexibility over the parameters that are passed to the functions that deal with animation states.

Use coroutines for relaxed updates

If your game requires a repeated action at a specific time interval, you can try launching a coroutine in the `MonoBehaviour.Start()` callback. Launching a coroutine is an alternative to performing an action in every frame through the `MonoBehaviour.Update()` callback. The following code shows an example of a coroutine:

```
void Update()
{
        // Perform an action every frame
}
```

```
IEnumerator Start()
{
        while(true)
        {
                // Do something every quarter of second
                yield return new WaitForSeconds(0.25f);
        }
}
```


Note

Another use of coroutines is to spawn enemies at irregular intervals rather than regular intervals. You can use an infinite loop inside the coroutine that spawns an enemy and generates a random number. Then pass the random number to the `WaitForSeconds()` function.

Avoid hard-coded values for tags

Avoid hard-coded values for tags. This is because hard-coded values restrict the scalability and robustness of your game. For example, with tag names, if you refer to the names directly by strings you cannot easily modify them. Therefore, you are potentially exposed to spelling errors. A hard-coded value for a tag is shown in the following code:

```
if(gameObject.CompareTag("Player"))
{
        [...]
}
```

You can improve the preceding code by implementing a special class for tags that exposes public constant strings. For example:

```
public class Tags
{
        public const string Player = "Player";
        [...]
}
if(gameObject.CompareTag(Tags.Player))
{
        [...]
}
```

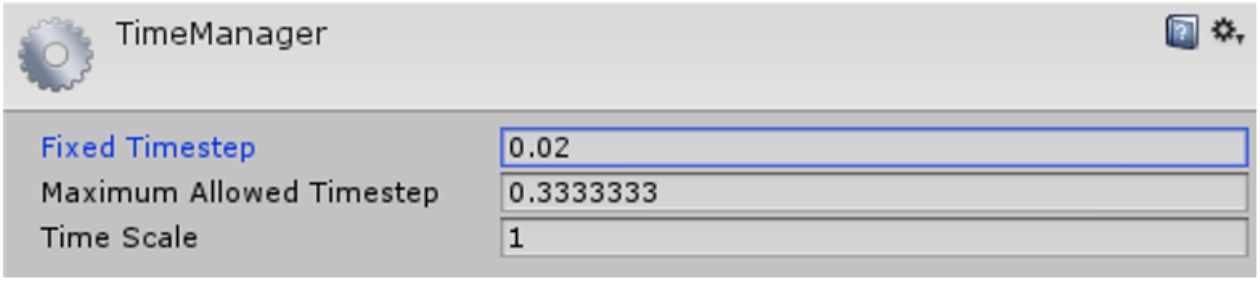
Reduce the number of physics calculations

Most physics calculations take place at a fixed time step. You can increase or decrease the length of this step to reduce computation load. Increasing the time step decreases the load on the application processor, but reduces the accuracy of physics calculations.

To access the time manager from the main menu, select Edit > Project Settings > Time.

The following screenshot shows the time manager:

Figure 2-1: Fixed timestep settings



Remove empty callbacks

If your code includes empty definitions for functions, like `Awake()`, `Start()`, or `Update()`, remove them. There is an overhead associated with the empty functions. This is because the engine attempts to access them even though they are empty. For example:

```
// Remove the following empty definition

void Awake()
{

}
```

Avoid using GameObject.Find() in every frame

`GameObject.Find()` is a function that iterates through every object in the scene. This function can cause a significant increase in the main thread size if it is used in an incorrect part of your code. For example:

```
void Update()
{
        GameObject playerGO = GameObject.Find("Player");
        playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

A better technique is to call `GameObject.Find()` on startup and cache the result, for example, in the `Start()` or `Awake()` function. The following code uses the `Start()` function:

```
private GameObject _playerGO = null;

void Start()
{
        _playerGO = GameObject.Find("Player");
}

void Update()
{
        _playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

The function `GameObject.FindWithTag()` is a faster alternative to `GameObject.Find()`. The following code shows `GameObject.FindWithTag()`:

```
void Update()
{
        GameObject playerGO = GameObject.FindWithTag("Player");
        playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```


Note

Use a dedicated class called `LocatorManager` that performs all the object retrievals immediately after the scene finishes loading. This allows other classes to use LocatorManager as a service, so that objects are not retrieved multiple times.

Use the StringBuilder class to concatenate strings

When concatenating complex strings, use the `System.Text.StringBuilder` class. This class is faster than the `string.Format()` method and uses less memory than concatenation with the plus operator. This code shows the plus operator concatenation and `string.Format()` methods:

```
// Concatenation with the plus operator
string str = "foo" + "bar";

// String.Format() method
string str = string.Format("{1}{2}", "foo", "bar");
```

To make the code faster, use the `System.Text.StringBuilder` class:

```
// StringBuilder class
using System.Text;

StringBuilder strBld = new StringBuilder();
strBld.Append("foo");
strBld.Append("bar");
string str = strBld.toString();
```

The following screenshot shows the difference in performance between using the `string.Format()` method, the concatenation method and the `StringBuilder` class:

Figure 2-2: Performance for the three methods

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
StringConcatenationTest.Start()	99.8%	0.0%	1	1.12 GB	2488.86	0.33
StringConcatenationTest.StringFormatMethod()	51.0%	0.2%	1	0.56 GB	1273.01	6.78
String.Format()	50.8%	0.3%	10000	0.56 GB	1266.22	7.51
StringConcatenationTest.PlusConcatenation()	48.4%	0.2%	1	0.56 GB	1208.21	6.53
String.Concat()	48.2%	0.5%	10000	0.56 GB	1201.68	13.15
StringConcatenationTest.StringBuilderClass()	0.2%	0.2%	1	256.2 KB	7.31	6.97
StringBuilder.Append()	0.0%	0.0%	10000	256.1 KB	0.32	0.12
StringBuilder..ctor()	0.0%	0.0%	1	0 B	0.00	0.00
StringBuilder.ToString()	0.0%	0.0%	1	0 B	0.01	0.00

Use the CompareTag() method

Use the `GameObject.CompareTag()` method instead of the `GameObject.tag` property. This is because the `CompareTag()` method is faster and does not allocate extra memory. These methods are shown in the following code:

```
GameObject mainCamera = GameObject.Find("Main Camera");

// Gameobject.tag property
if(mainCamera.tag == "MainCamera")
{
        // Perform an action
}

// Gameobject.CompareTag() method
if(mainCamera.CompareTag("MainCamera"))
{
        // Perform an action
}
```

The following screenshot compares the use of `CompareTag()` and `GameObject`:

Figure 2-3: Compare tag and game object comparison

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ TagComparisonTest.Start()	97.2%	0.1%	1	3.6 MB	127.48	0.26
▼ TagComparisonTest.TagProperty()	68.0%	59.6%	1	3.6 MB	89.27	78.14
▼ GameObject.get_tag()	7.9%	1.0%	100000	3.6 MB	10.44	1.32
GC.Collect	6.9%	6.9%	4	0 B	9.12	9.12
▼ String.op_Equality()	0.5%	0.3%	100000	0 B	0.69	0.50
String.Equals()	0.1%	0.1%	100000	0 B	0.18	0.18
▶ TagComparisonTest.CompareTagMethod()	28.9%	28.4%	1	0 B	37.94	37.27
GameObject.Find()	0.0%	0.0%	1	0 B	0.00	0.00

Use object pools

If your game has many objects of the same kind that are created and destroyed at runtime, you can use the design pattern object pool. This design pattern avoids the performance penalty of allocating and freeing many objects dynamically. Using object pools for enemies and bombs restricts the allocation of those objects to the loading phase of the game.

If you know the total number of objects that you require, you can create them all immediately and disable the objects that are not immediately required. When a new object is required, search the pool for the first unused one and enable it.

When an object is not required anymore, you can return it to the pool by disabling it and resetting it to a default starting state.

You can use this technique with objects like enemies, projectiles, and particles. If you do not know the exact number of objects that you require, test to find out how many are used. So that you have a safety margin, create a pool that is slightly bigger than the number that you find. Without a safety margin, the player experience can be affected by an object that disappears as the game creates a new one. The game might even crash if it fails to create needed objects.

Cache component retrievals

Cache the component instance that `GameObject.GetComponent()` returns. The function call that is involved is quite expensive.

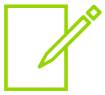
Properties like `GameObject.camera`, `GameObject.renderer`, or `GameObject.transform` are shortcuts to the corresponding `GameObject.GetComponent()`, `GameObject.GetComponent()`, and `GameObject.GetComponent()` . The following code shows the correct usage:

```
private Transform _transform = null;

void Start()
{
        _transform = GameObject.GetComponent();
}

void Update()
{
        _transform.Translate(Vector3.forward * Time.deltaTime);
}
```

Consider caching the return value of `Transform.position`. Even though the function is a C# getter property, there is overhead associated with running it. The overhead comes from iterating over the transform hierarchy to calculate the global position.


Note

In Unity 5 and newer, the transform component is automatically cached.

Use OnBecameVisible() and OnBecameInvisible() callbacks

Callbacks like `MonoBehaviour.OnBecameVisible()` and `MonoBehaviour.OnBecameInvisible()` notify your scripts if their associated game objects become visible or invisible on screen.

These calls enable you to, for example, disable computationally heavy code routines or effects when a game object is not rendered on screen.

Use sqrMagnitude for comparing vector magnitudes

If your application requires the comparison of vector magnitudes, use `Vector3.sqrMagnitude` instead of `Vector3.Distance()` or `Vector3.magnitude`.

`Vector3.sqrMagnitude` sums the squared components without calculating the root, but this sum is useful for comparisons. The other calls use a computationally expensive square root.

The following code shows the three different techniques that are used to compare two positions in space:

```
// Vector3.sqrMagnitude property
if ((_transform.position - targetPos).sqrMagnitude < maxDistance * maxDistance)
{
        // Perform an action
}
```

```
// Vector3.Distance() method
if (Vector3.Distance(transform.position, targetPos) < maxDistance)
{
        // Perform an action
}

// Vector3.magnitude property
if ((_transform.position - targetPos).magnitude < maxDistance)
{
        // Perform an action
}
```

Use built-in arrays

If you know the size of an array in advance, use the built-in arrays.

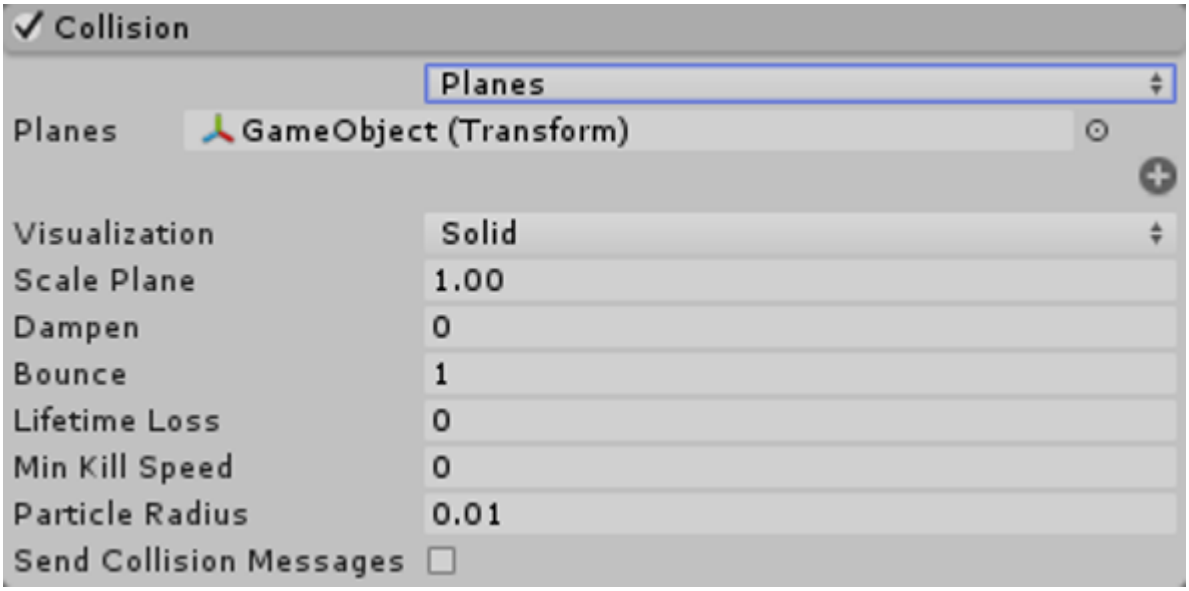
The `ArrayList` and `List` classes have more flexibility than built-in arrays. This is because they grow when you insert more elements. However, they are slower than the built-in arrays.

Use planes as collision targets

If your scene only requires particle collisions with planar objects like floors or walls, change the particle system collision mode to Planes to reduce the required computations. In this mode, you can provide Unity with a list of empty `GameObjects` to act as the collider planes.

The following screenshot shows collision settings for Planes mode:

Figure 2-4: Collision settings for Planes mode



Use compound primitive colliders

Mesh colliders are based on the real geometry of an object. Mesh colliders are accurate for collision detection but are computationally expensive.

You can combine shapes like boxes, capsules, or spheres into a compound collider that mimics the shape of the original mesh. Combining shapes provides similar results to mesh colliders, with a much lower computational overhead.

3. GPU optimizations

This section of the guide shows how you can use various graphics optimizations to reduce the time and effort that your CPU or GPU need to run your game.

Use static batching

Static batching is a common optimization technique that reduces the number of draw calls, and therefore application processor use.

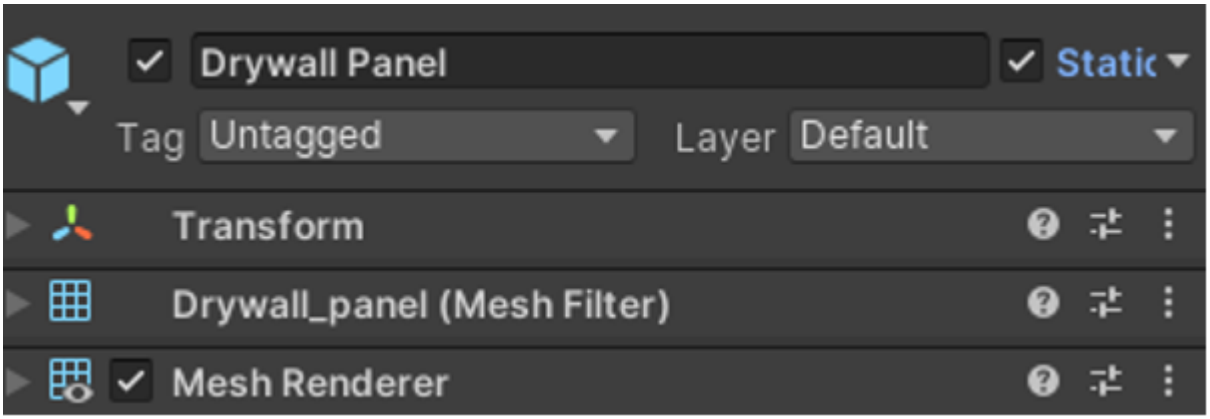
Unity performs dynamic batching transparently, but cannot apply it to objects that are made of many vertices. This is because the computational overhead becomes too large.

Static batching can work on objects that are made of many vertices, but the batched objects must not move, rotate, or scale during rendering.

To enable Unity to group objects for static batching, mark them as static in the Inspector window.

The following screenshot shows static batching settings:

Figure 3-1: Static batching settings

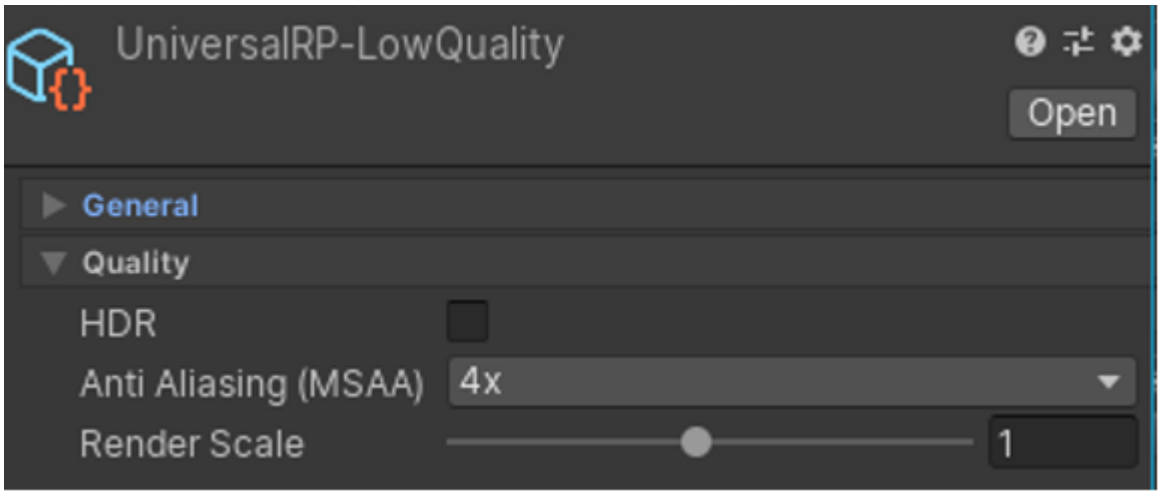


Use 4x MSAA

Arm Mali GPUs can do 4x Multi-Sample Anti-Aliasing (MSAA) with a low computational overhead.

You can enable 4x MSAA in:

- The Universal Render Pipeline (URP) settings if you are using the URP. The following screenshot shows the setting:



- The Unity Quality Settings if you are not using URP. Select Edit > Project Settings > Quality Settings.

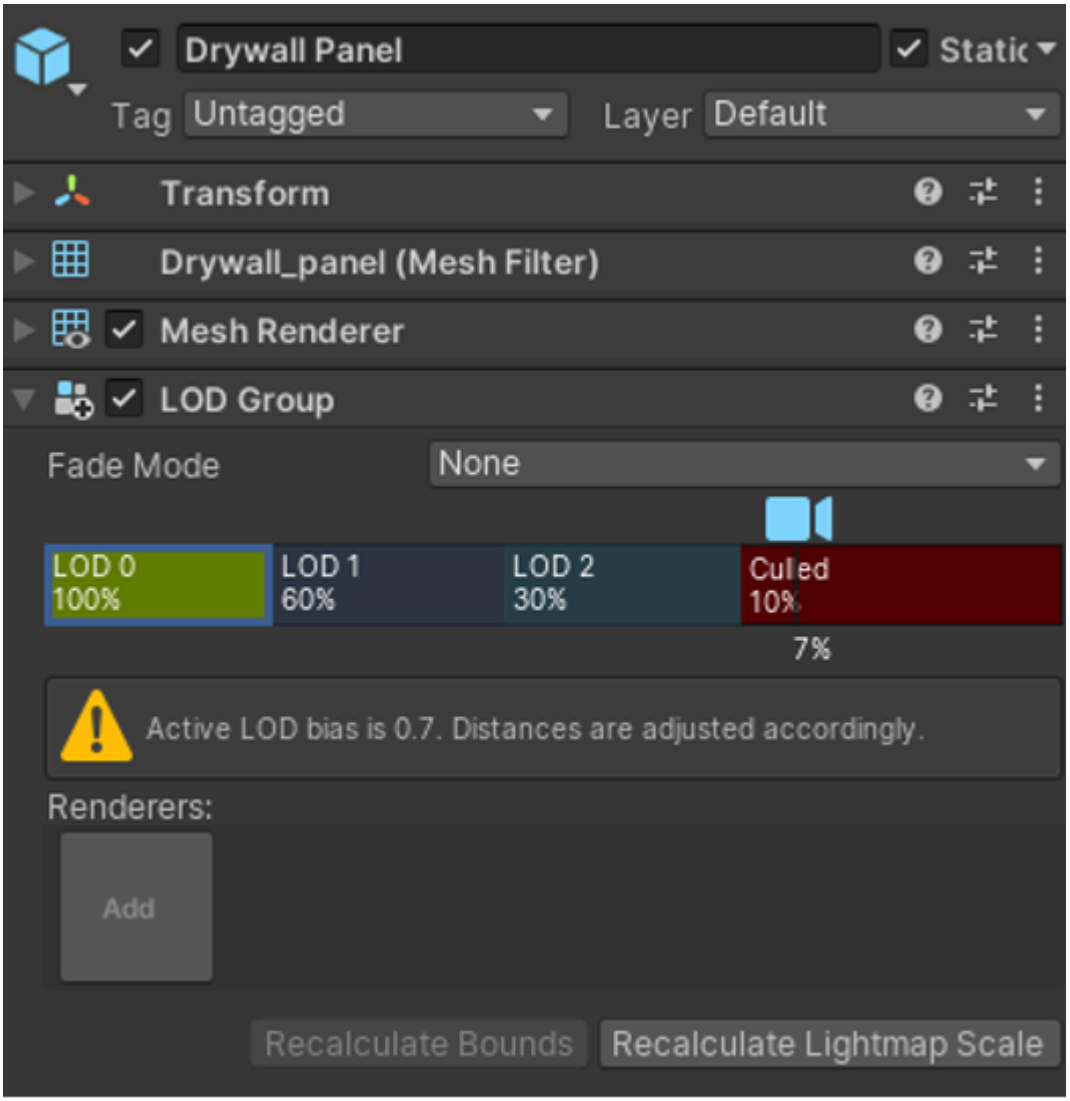
Use Level of Detail

Level of Detail (LOD) is a technique in which the Unity engine renders different meshes for the same object at different distances from the camera. Geometry is more detailed when the object is close to the camera. The LOD is reduced as the object moves away from the camera. At the furthest distance, you can use a planar-aligned billboard.

To set up LOD groups to manage the meshes that you use and the associated distance ranges, select Add Component > Rendering > LOD Group.

From Unity 5, you can set a Fade Mode for each LOD level to blend to contiguous LODs. Fade Mode smooths the transition between the LODs. Unity calculates a blending factor according to the screen size of the object and passes it to your shader for blending. You must implement the geometry blending in a shader. The following screenshot shows the LOD group settings:

Figure 3-2: Level of Detail group settings



Avoid mathematical functions in custom shaders

When writing custom shaders, minimize the use of expensive built-in mathematical functions. This is because these functions can cause the custom shader to take longer to run. Expensive functions include:

- `pow()`
- `exp()`
- `log()`
- `cos()`
- `sin()`
- `tan()`
- `sqrt()`

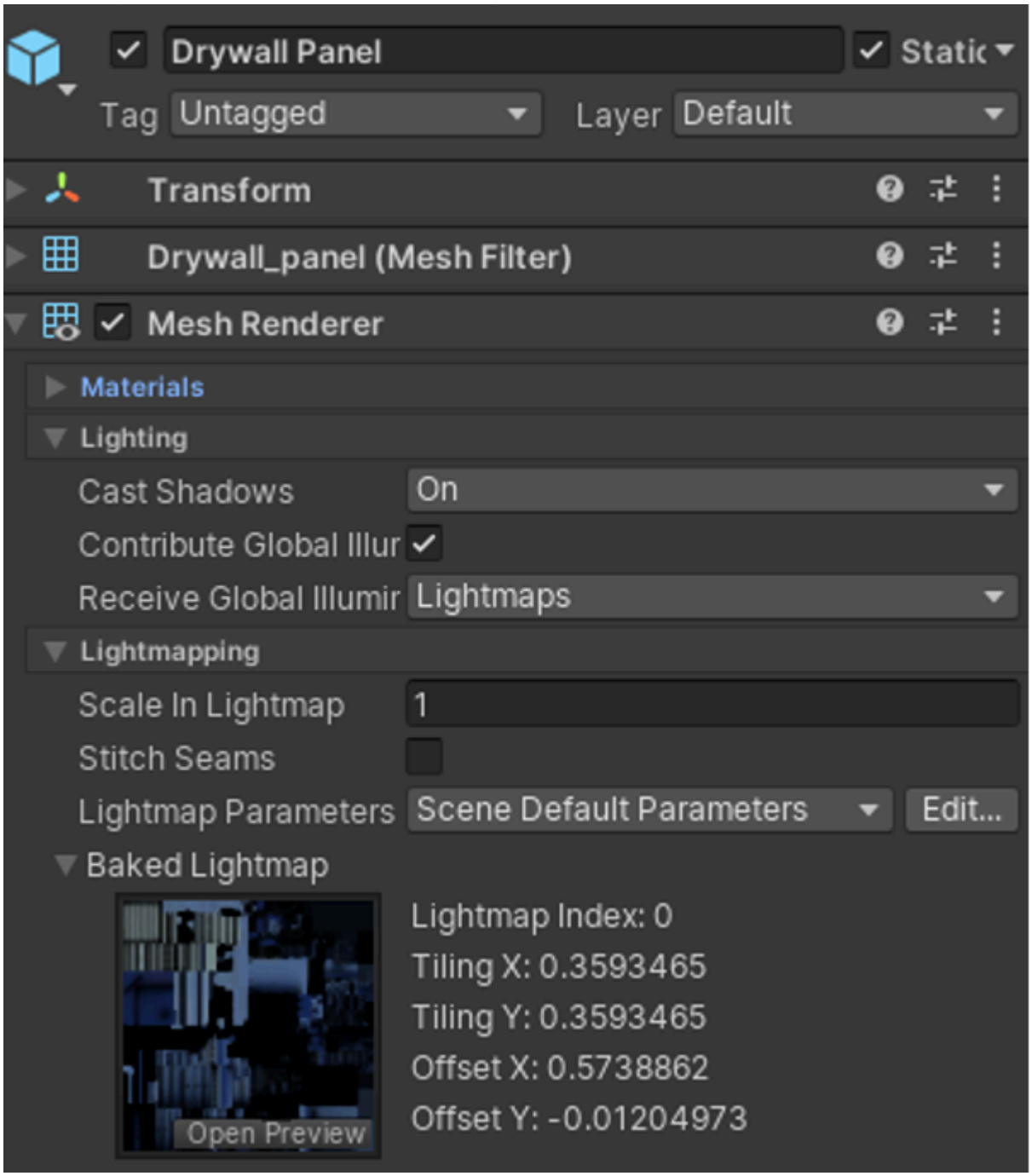
Use lightmaps and light probes

Runtime lighting calculations are computationally expensive. A popular technique to reduce the computational cost is called lightmapping. Lightmapping pre-computes the lighting calculations and bakes them into a texture called a lightmap. With a lightmap, you lose the flexibility of a fully dynamically lit environment. However, you get high-quality images without affecting performance.

To bake the resulting lighting in a static lightmap, follow these steps:

1. Navigate to the Inspector window for the geometry that is receiving the lighting.
2. Set the geometry to Static.
3. Navigate to Mesh Renderer > Lighting:
    a. Check the Contribute Global Illumination option.
    b. Set Receive Global Illumination to Lightmaps.

The Settings window is shown in the following screenshot:



4. Navigate to Window > Rendering > Lighting Settings > Scene tab select the Baked Global Illumination option. To see the resulting lightmap, select the geometry and do one of the following:

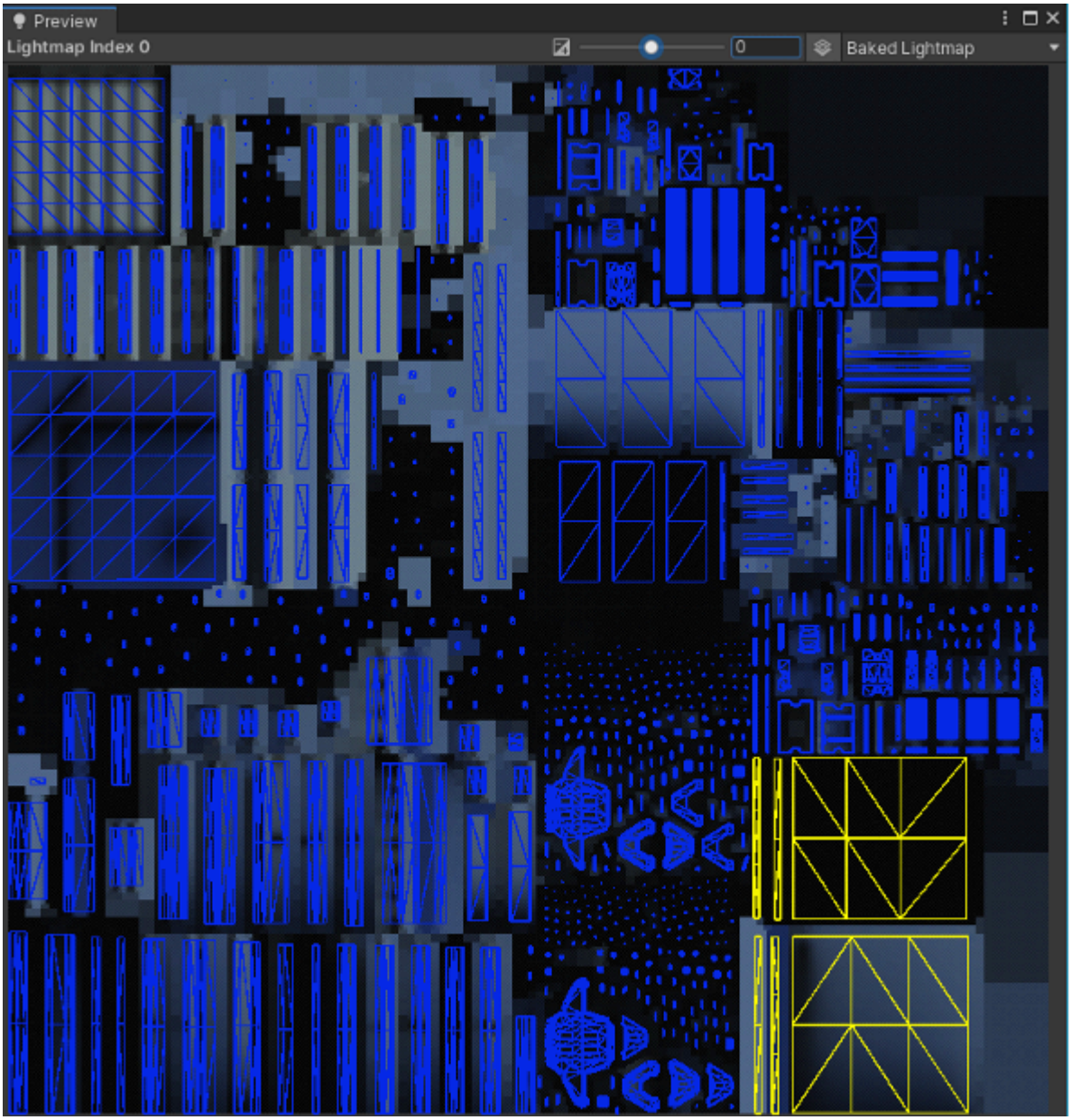
- Open the Lighting window and view the Baked Lightmaps.
- Open the Inspector window for the object and select Baked Lightmap > Open Preview.

If the Continuous Baking option is selected, Unity bakes the lightmap and updates the scene in the Editor window in seconds. If the Continuous Baking option is not selected, select Generate Lighting to update the scene.

A quick way to check that the lightmap has set up correctly is to run the game in the Editor window and disable the light. If the lighting is still there, the lightmap has been created correctly and is in use.

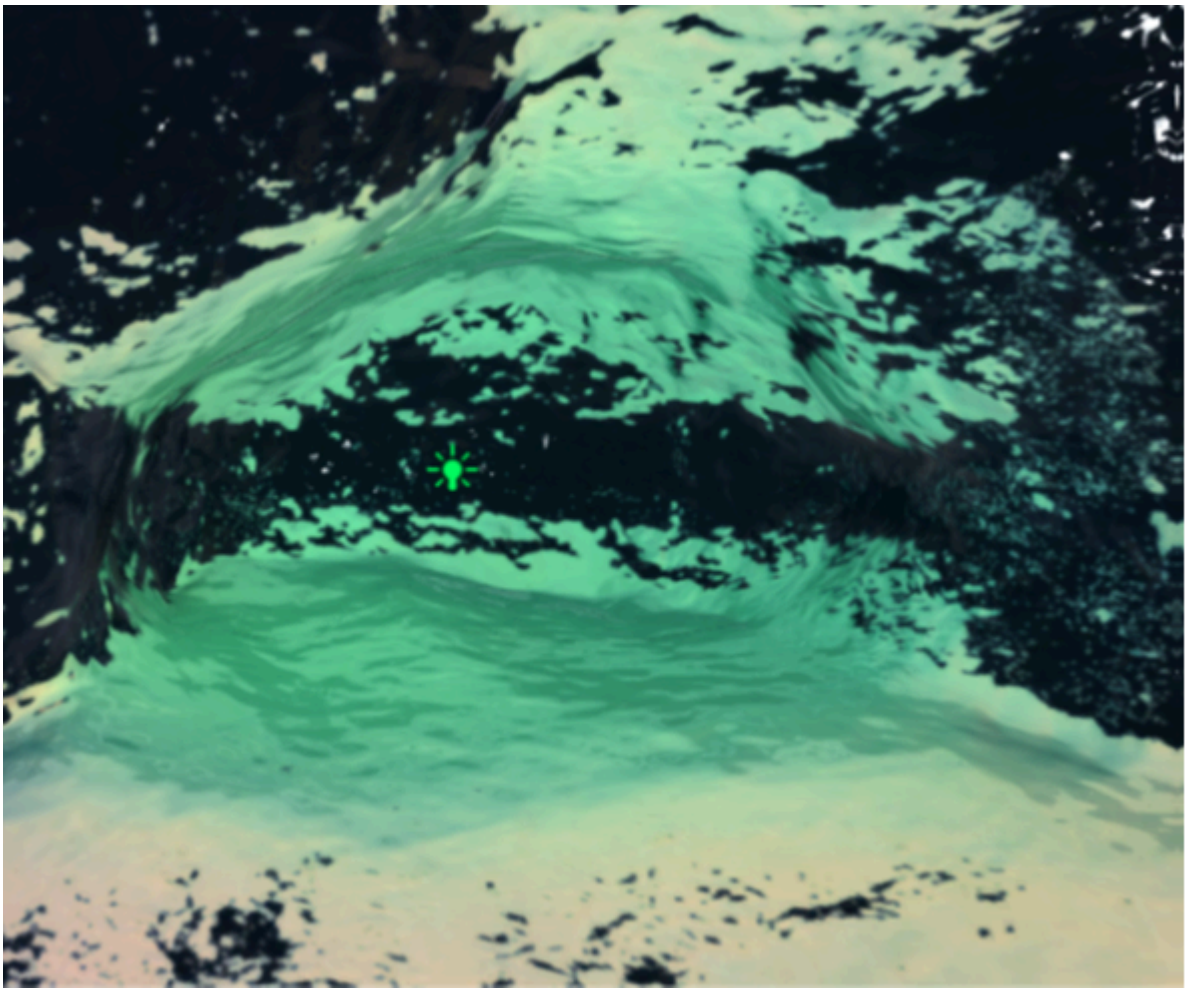
The following screenshot shows a baked lightmap as opened from either the Inspector window's Lightmapping section or the Lighting window:

Figure 3-3: The baked lightmap



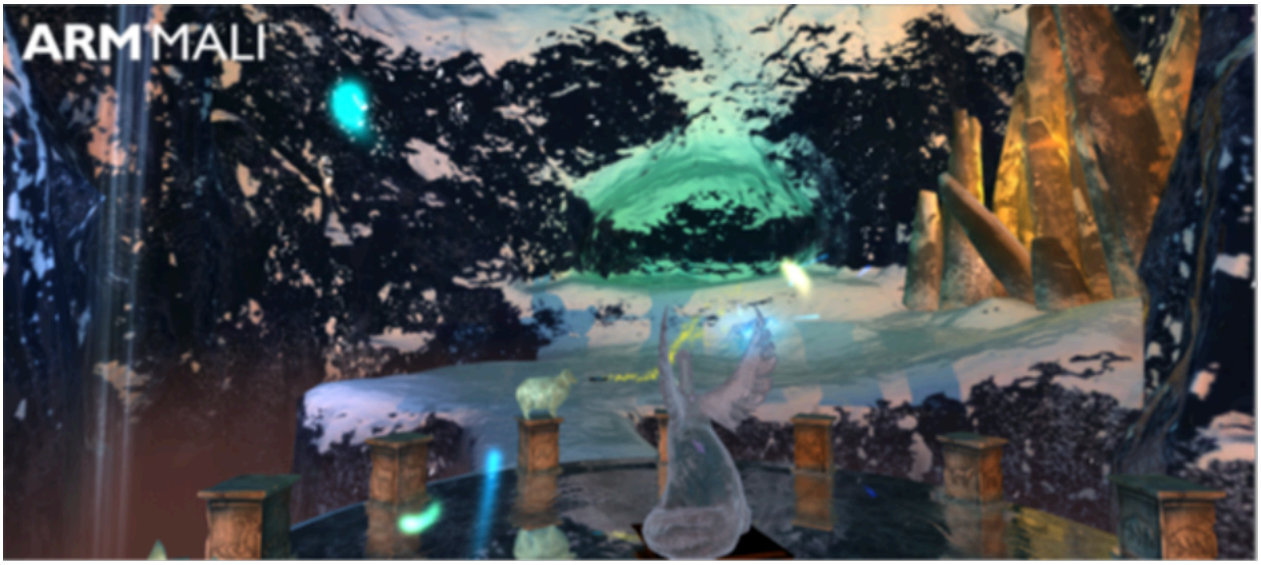
The following screenshot shows the Editor window displaying lighting from a green light at the end of a cave. The lighting is generated with a static lightmap:

Figure 3-4: Adding a light to bake a static lightmap



The following screenshot shows the result of the static lightmap in the Ice Cave demo:

Figure 3-5: Lightmapped cave



Setting up lightmapping

To prepare an object for lightmapping, you need:

- A model in your scene with lightmap UVs. You can do this by selecting Generate Lightmap UVs when importing the mesh.
- The model must be set as Lightmap Static. Objects that are not marked as static are not placed in the lightmap.
- There must be a light within range of the model. The Baking type of this light must be set to Baked.

These objects are not likely to be perfect, so experiment to see what works best for your game.

To set up lightmapping, follow these steps:

1. Navigate to the main menu and select Lighting > Window and Lightmapping.
2. Select one of the following options:
    - Scene
    - Baked Lightmaps
    - Real-time Lightmaps, if you are not using the URP

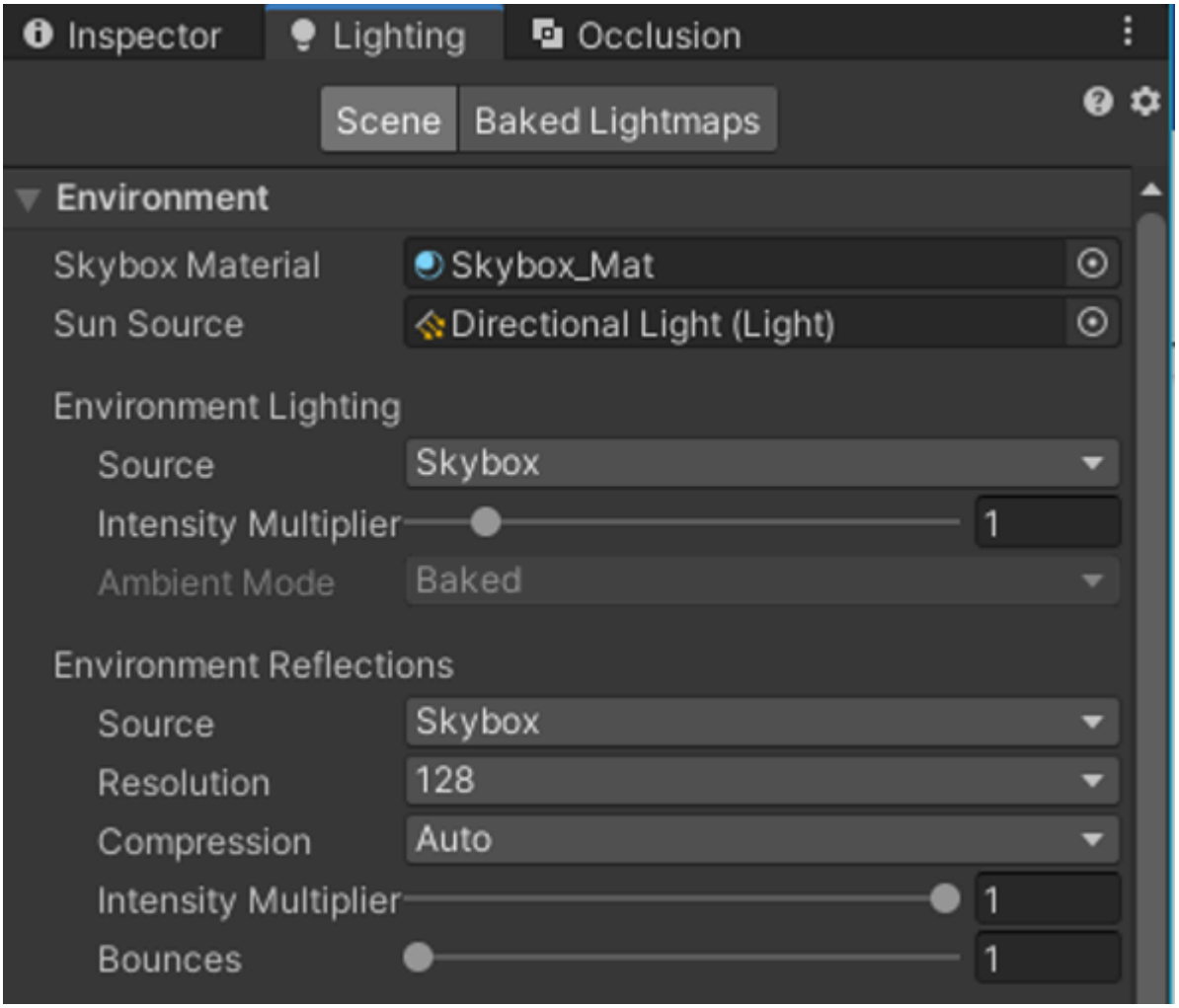
Scene-level lighting options

The options in the Lighting > Scene tab can have a big impact on performance. Here, we review a few of the important ones:

- In the Environment section, the Environment Reflections > Bounces option is the most important from the performance point of view. Reflection bounces defines the number of inter-reflections between reflective objects, that is, the number of bake times for the probe that sees the objects. If the reflection probes are updated at runtime, they can have a large, negative

impact on performance. Only set the number of bounces higher than one if the reflective objects are visible in the probes. The following screenshot shows the Environment section:

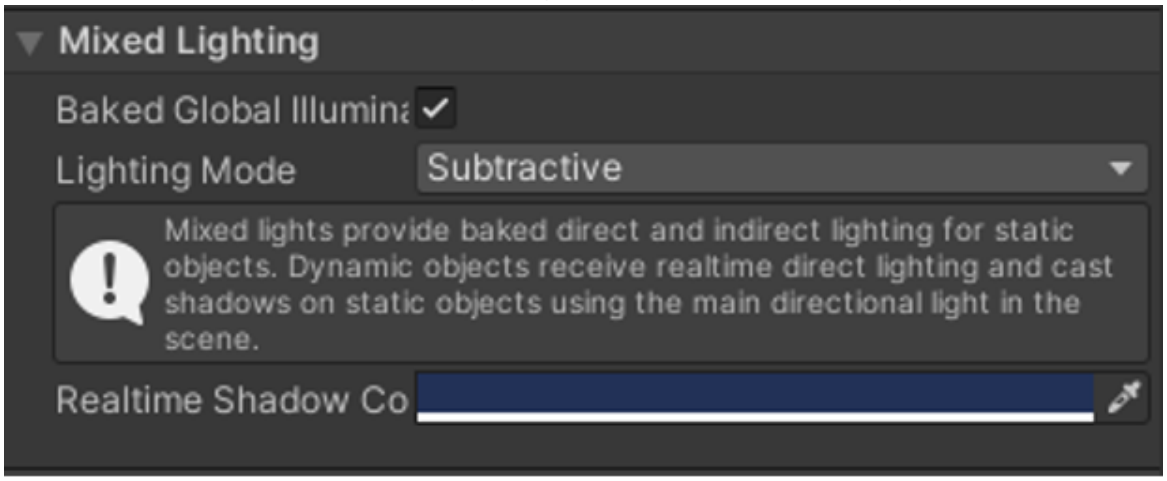
Figure 3-6: Environment settings



In the Mixed Lighting section:

- Select a Lighting Mode. The different modes have significant performance variations; you can review them on the Unity documentation site.

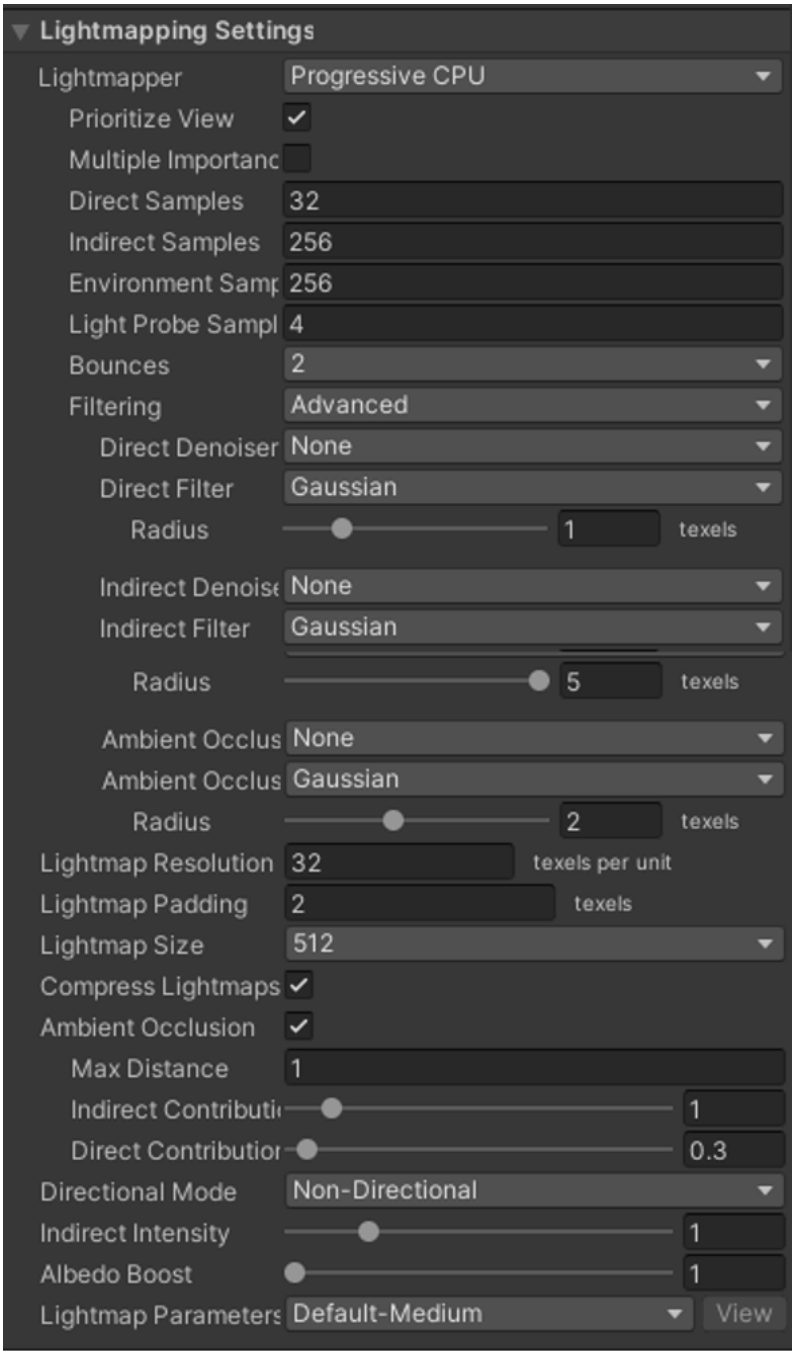
- In URP, there is no option for Real-time Global Illumination. However, you can select Baked Global Illumination to create lightmaps, as shown in the following screenshot:



- In the Lightmapping Settings:
  - Select the Lightmapper > Progressive GPU option for fast updates while setting up your scene. Select Progressive CPU for the final version.
  - Set the lightmap texture to be compressed by selecting the Compress Lightmaps option. Compressing lightmap textures requires less storage space and less bandwidth at runtime. However, the compression process can add artifacts to the texture.
  - Be careful with the Directional Mode option. When Directional Mode is set to Directional instead of Non-Directional, an extra lightmap is created to store the dominant direction of incoming light. As a result, Directional mode requires about twice as much storage space and video memory, compared to Non-Directional. If you cannot use deferred lighting with dual lightmaps, another technique you may want to use is directional lightmaps. Directional lightmaps enable you to use normal mapping and specular lighting without real-time lights. Use directional lightmaps if normal mapping must be preserved but dual lightmaps are not available. This situation is typical for mobile devices.
  - Reduce Lightmap Resolution, Lightmap Padding, and Lightmap Size to a level just above the level that causes artifacts.

The following screenshot shows lightmapping settings in line with the recommendations in this section of the guide:

Figure 3-7: Lightmapping Settings

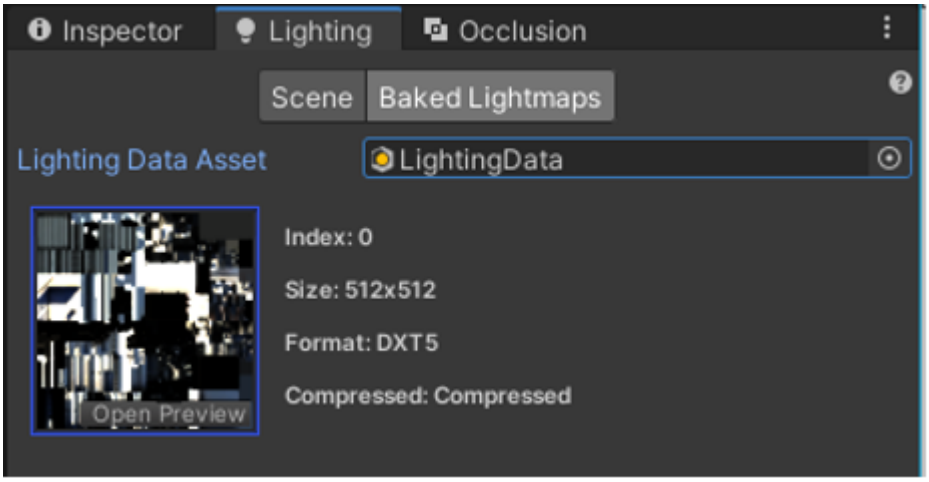


Review the Baked Lightmaps

Use the Baked Lightmaps tab to preview the impact of the lightmap settings you selected in the Scene tab.

The following screenshot shows lightmaps in the Lighting tab:

Figure 3-8: Lightmaps in the Lighting tab



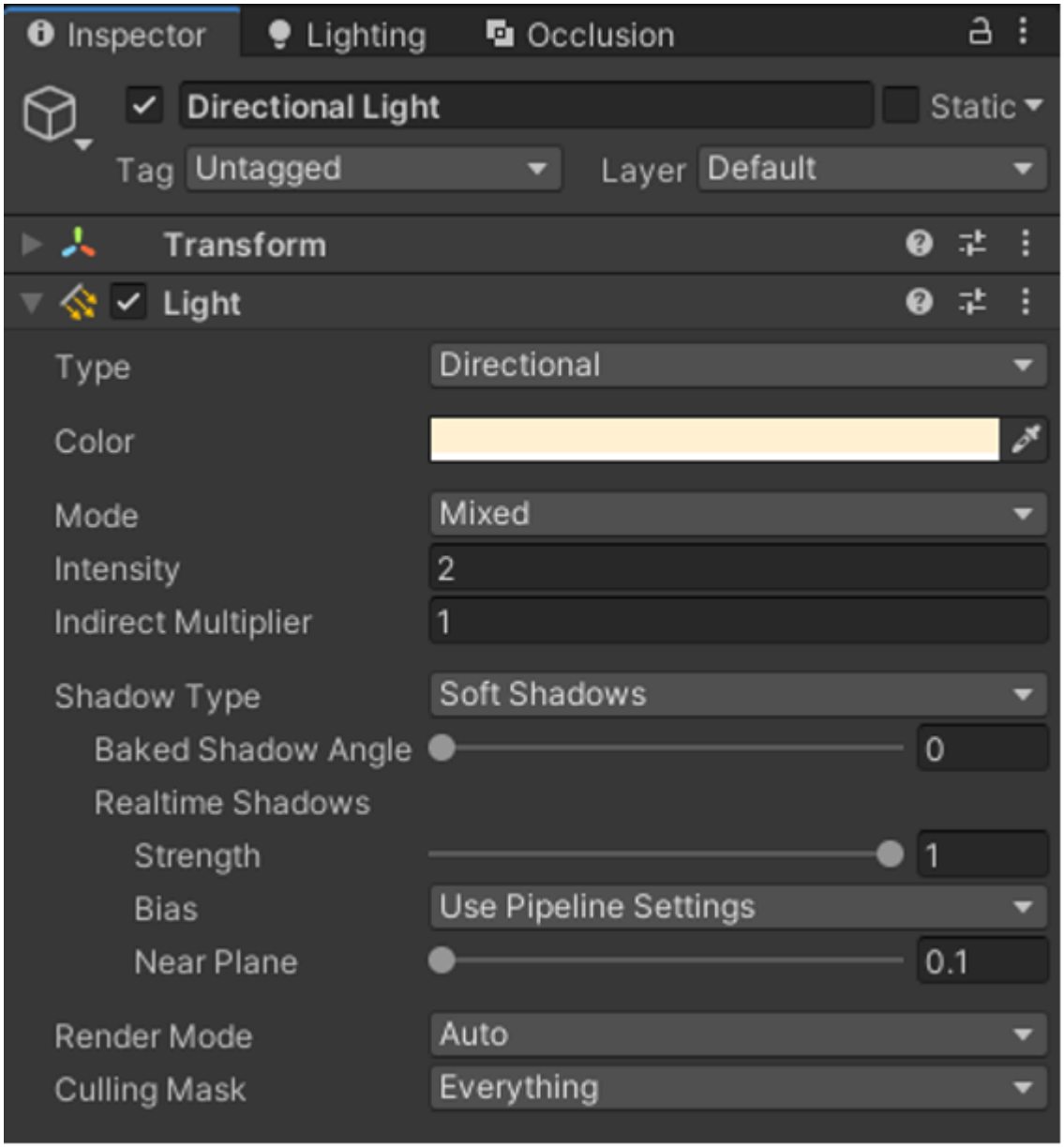
Object-level lighting options

You can modify the object settings that impact the lightmapping process in the Inspector window.

You can change the following options for your object:

- If you select a light, set Mode to Baked, Real-time, or Mixed. Setting most lights to Baked, or at least Mixed, ensures that the number of calculations at runtime is relatively low. The mode is a top-level lighting setting and has a huge impact on performance. The following screenshot shows the Light settings with a mixed mode:

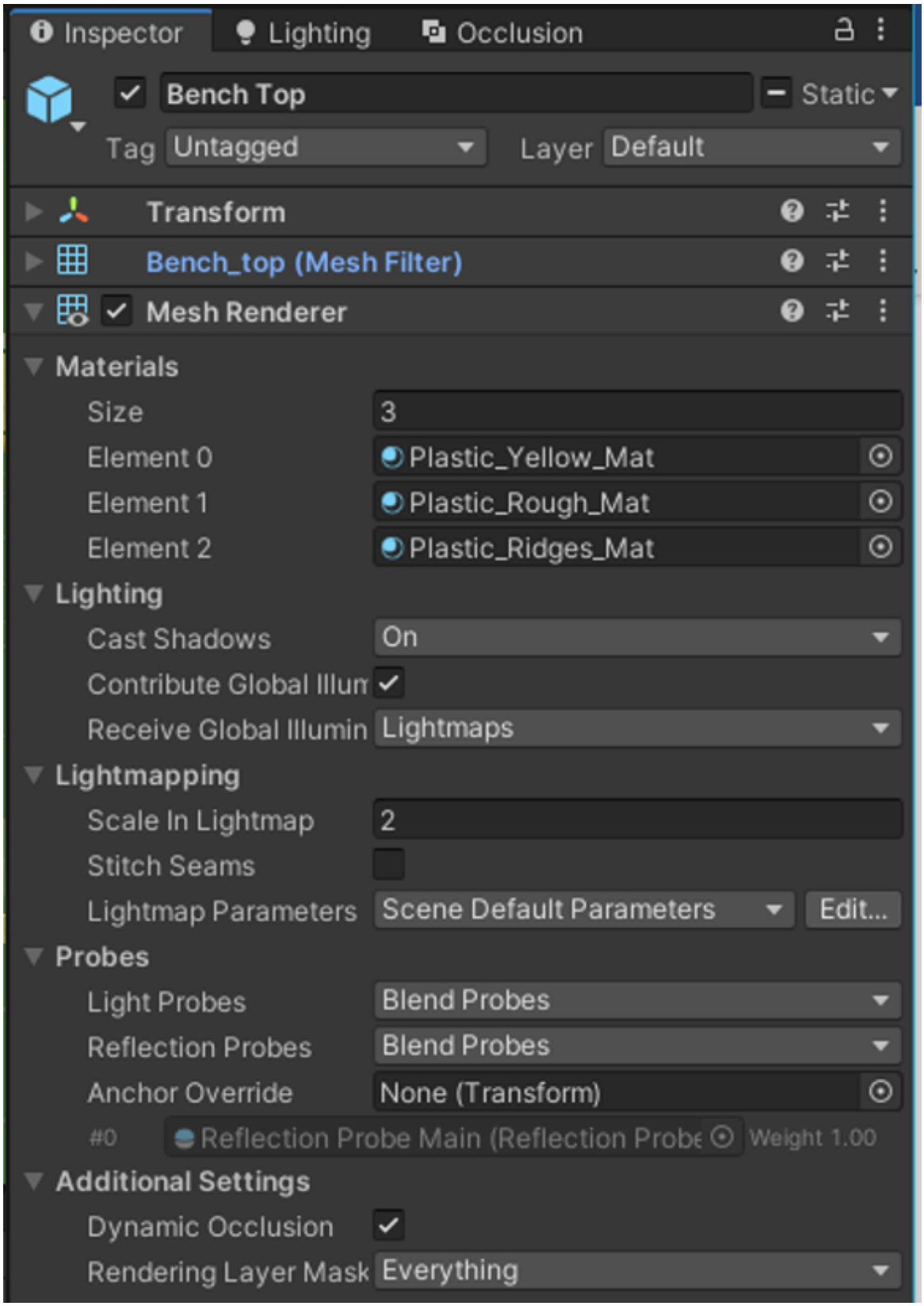
Figure 3-9: Light settings with Mixed Mode



Shadow options also need careful consideration. To learn more about shadows, see Limit shadow complexity.

- If you select an object with geometry, the two most important options for performance are:
  - Lightmapping > Stitch seams, which you will often want to use for getting correct lightmapping on complex objects.

- Lighting > Contribute Global Illumination and Receive Global Illumination set to Lightmaps for a static geometry, as shown in the following screenshot:



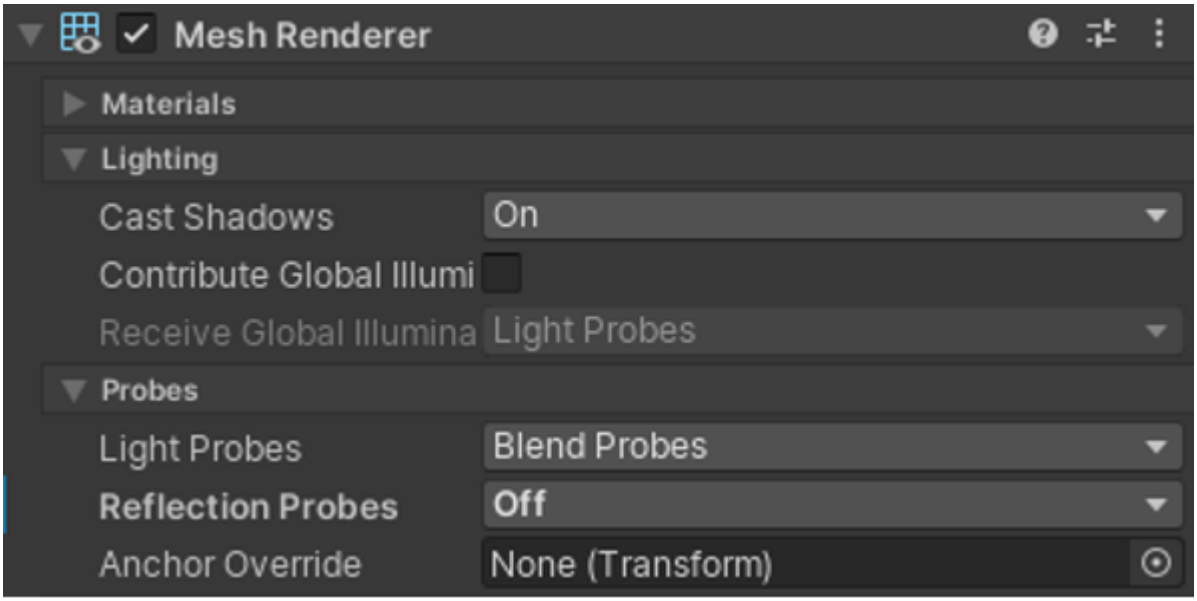
Use light probes for dynamic objects in your game

Light probes add dynamic lighting to lightmapped scenes. Light probes take a sample, or probe, of the lighting in an area. If the probes form a volume, or cell, the lighting is interpolated between these probes, depending on their position within the cell.

The more probes there are, the more accurate the lighting is. You do not typically require many light probes. This is because there is interpolation between probes. This means that the lighting at any position can be approximated. Approximation is done by interpolating between the samples that are taken by the nearest probes. You require more light probes in areas where there are large changes in light color or intensity.

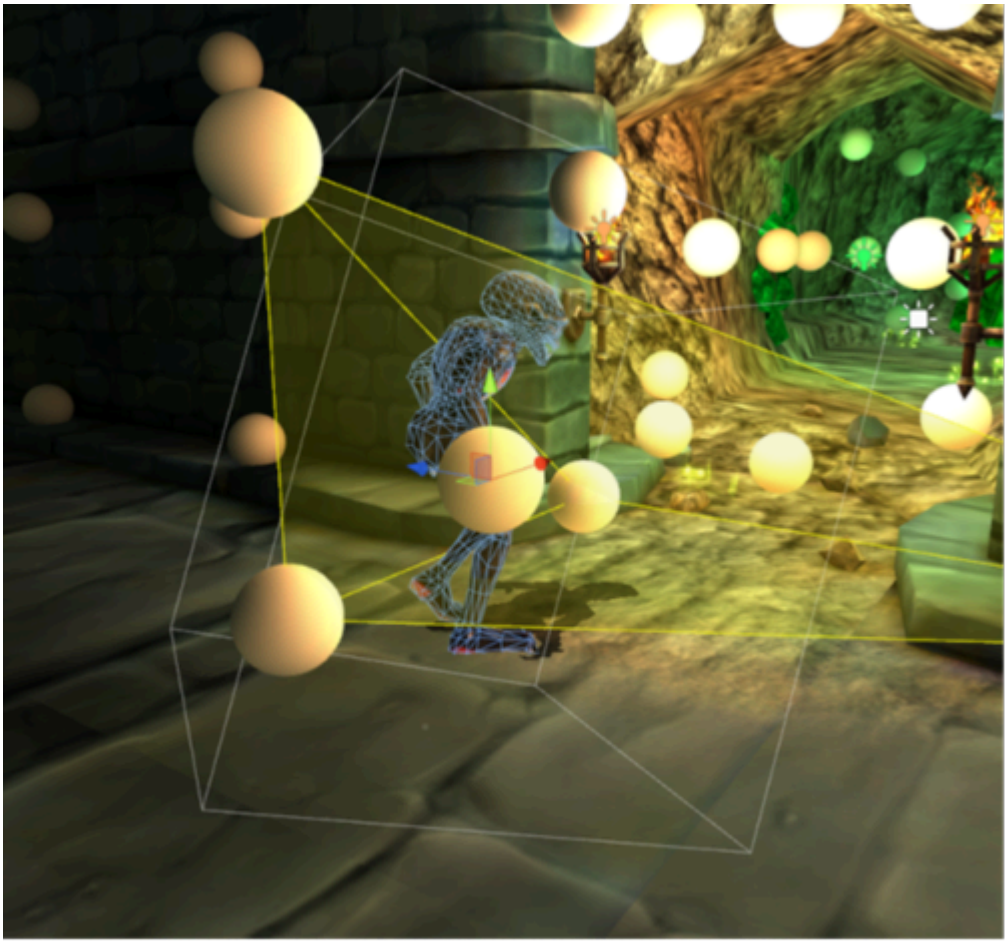
Be careful when you are placing the light probes. For the meshes that you want to be influenced by the probes, select Receive Global Illumination and Light Probes. The following screenshot shows light probe settings for an object with a mesh, in Object Settings:

Figure 3-10: Probes settings



The following screenshot shows multiple light probes:

Figure 3-11: Multiple light probes in a scene



Use ASTC texture compression

Adaptive Scalable Texture Compression (ASTC) is an official extension to the OpenGL and OpenGL ES graphics APIs, and is a core feature in the Vulkan API. ASTC can reduce the memory requirements of your application, and the memory bandwidth use of your GPU.

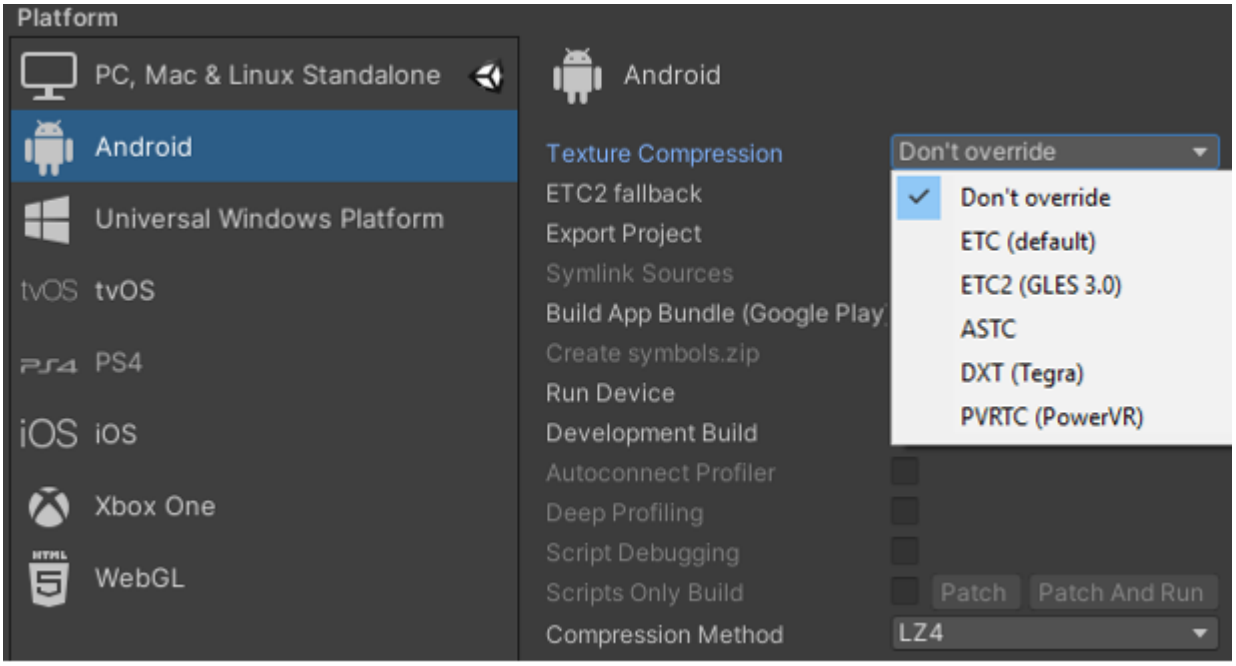
ASTC offers texture compression with high quality and low bitrate, and has many options. ASTC includes the following features:

- Bit rates range from 8 bits per pixel (bpp) to less than 1bpp. This enables you to fine-tune the trade-off of file size against quality.
- Support for 1-4 color channels
- Support for both Low Dynamic Range (LDR) and High Dynamic Range (HDR) images
- Support for 2D and 3D images
- Support for selecting different combinations of features

You can turn on ASTC for all textures in File > Build Settings. But we suggest leaving Texture Compression set to Don't override and giving texture-specific settings. This compression setting is global for all the textures. The Don't override option means that the default format will be used.

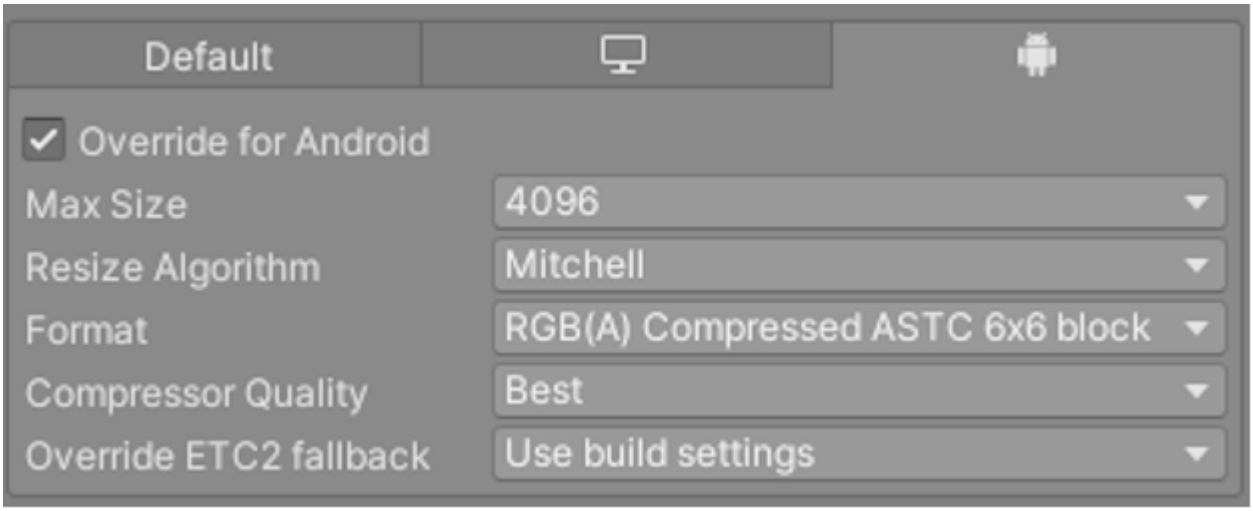
The following screenshot shows a setting of Don't override, rather than ASTC:

Figure 3-12: Texture Compression set to Don't override



The following screenshot shows the section in Inspector > Texture Settings that deals with platform-specific formatting:

Figure 3-13: Platform-specific texture settings

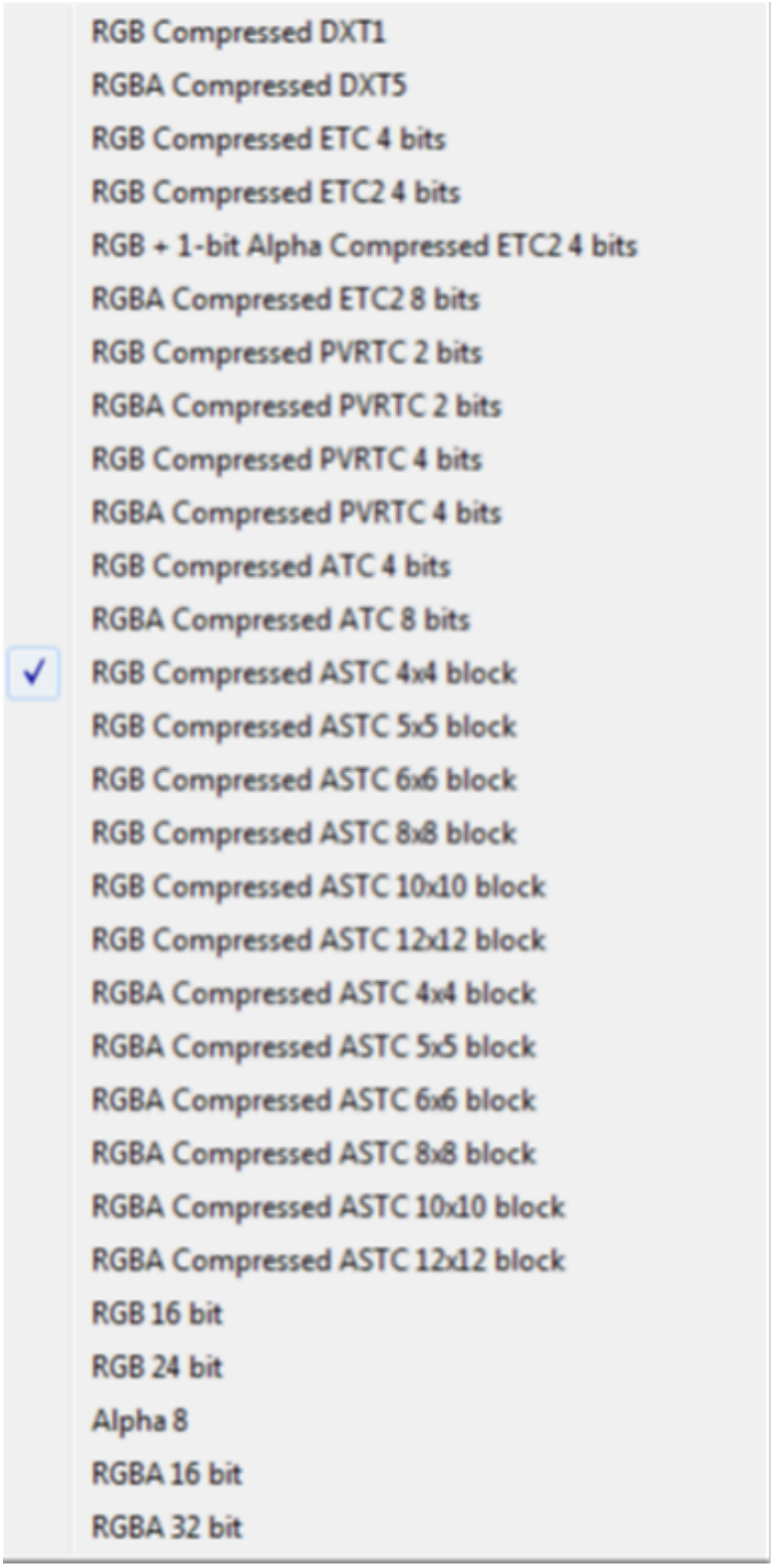


If you select Best for Compressor Quality, the compressor will try a lot of different ASTC block options to find the best result for quality and size. If you select Fast, the compressor will try only the most promising options, which will often not be the best ones. This means that Best provides a better quality than Fast, at the cost of being slower. You may want to select Fast when you are experimenting with your scene and want to iterate quickly, and Best for your final version.

There are several block sizes available in the ASTC settings window. The larger block sizes provide higher compression. Select large block sizes for textures that are not shown in detail, for example, objects far away from the camera. Select smaller block sizes for textures that show more detail, for example those closer to camera.

The following screenshot shows the block sizes for different texture compression formats:

Figure 3-14: Texture compression block sizes




Note

Compression in mobile devices:

- OpenGL ES 2 does not support ASTC. Therefore, if you want to target OpenGL ES 2 devices and use texture compression, you may want to build a separate Android Application Package (APK) for those devices and use ETC encoding.
- Most Android devices now support ASTC, so normally you should use ASTC to compress the textures in your 3D content. If you are targeting devices that do not support ASTC, try using ETC2.
- You must differentiate between textures that are used in 3D content from textures that are used in the Graphical User Interface (GUI) elements. This differentiation lets you select different compressions for each texture use. In some cases, you might want to leave the GUI textures uncompressed to avoid unwanted artifacts, but must still compress the 3D elements. Leaving 3D elements uncompressed can affect the game performance, especially on mobile devices. You may also want to differentiate between your 3D background and 3D hero character, so that you can choose a different compression for the background.

Each ASTC texture type needs a different compression format to achieve the best possible results.

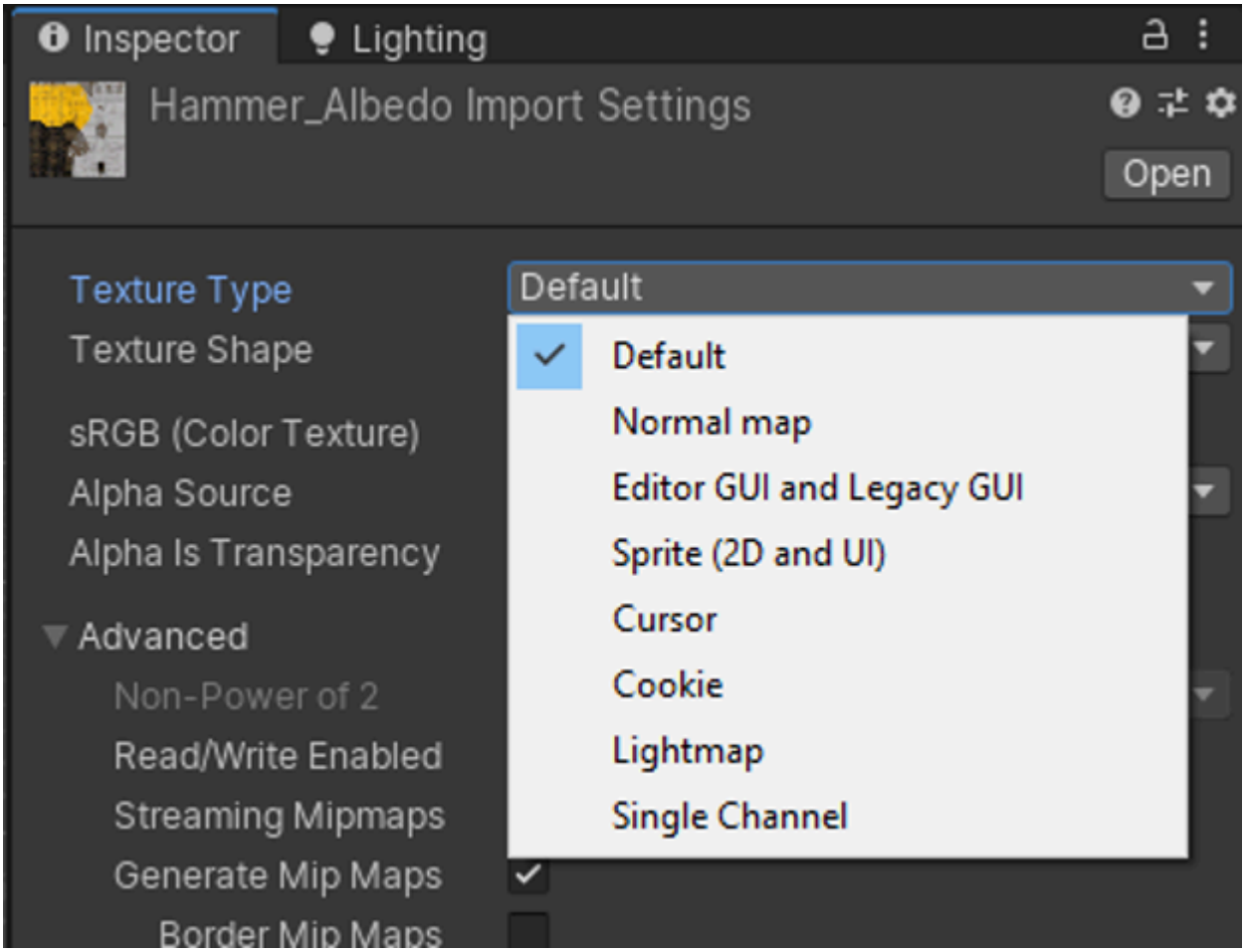
Texture compression algorithms have different channel formats, typically RGB and RGBA. ASTC supports several other formats, but these formats are not exposed within Unity.

Each texture type is typically used for a different purpose, for example:

- Standard texturing
- Normal mapping
- Specular
- HDR
- Alpha
- Look up textures

The following screenshot shows the texture types in the Inspector window:

Figure 3-15: Texture Type in Import Settings



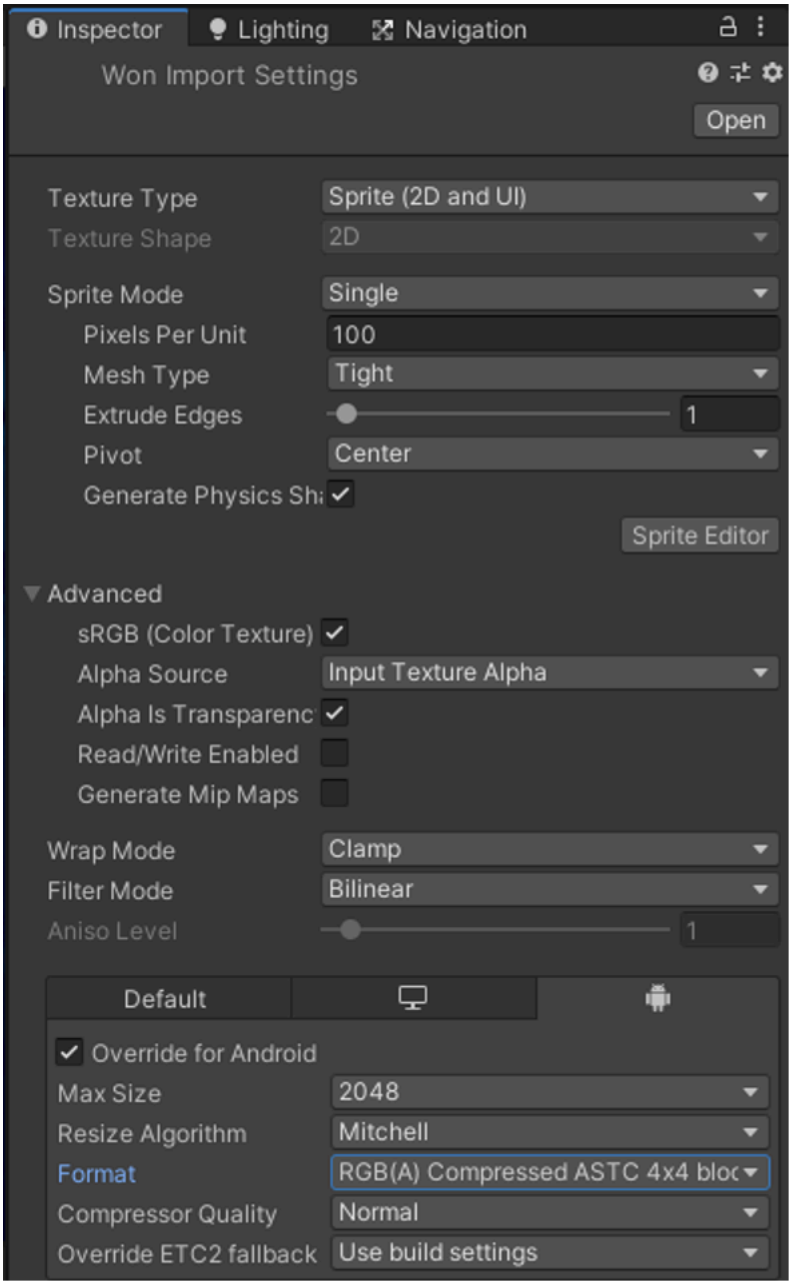
Unity typically imports your texture as the default type, which is suitable for most images, but uses more specialist ones where appropriate. For example, it is important to specify when a normal map is used. This is because ASTC needs to compress differently for normal maps, where the RGB components do not correlate with color.

Selecting individual settings for all your textures improves the visual quality of your project and avoids unnecessary texture data at compression time.

For example, the following screenshot shows settings for a GUI texture with some transparency. Because the texture is for a GUI, sRGB and Mip Maps are disabled. To include transparency, you need the alpha channel. To improve performance with an alpha channel, select Override for Android and choose an appropriate ASTC block size and compression format.

The following screenshot shows the Inspector with Advanced options:

Figure 3-16: Default texture settings



The following table shows the compression ratio for the available ASTC block sizes in Unity for an RGBA 8 bits per channel texture with a 1024x1024 pixel resolution at 4 MB in size:

ASTC block size	Size	Compression ratio	Bits per texel
4x4	1 MB	4.00	8.00
5x5	655 KB	6.25	5.12
6x6	455 KB	9.00	3.56
8x8	256 KB	16.00	2.0

ASTC block size	Size	Compression ratio	Bits per texel
10x10	164 KB	24.97	1.28
12x12	144 KB	35.93	0.89

Mipmapping

Mipmapping is a texturing technique that can enhance both the visual quality and performance of your game.

Mipmaps are pre-calculated versions of a texture at different sizes. Each texture that is generated is called a level, and it is half as wide and half as high as the preceding level. Unity can automatically generate the complete set of levels. The levels start at the original size and go down to a 1x1 pixel version.

If a texture does not have mipmap levels, and is bigger than the area, in pixels, that it covers, the GPU scales the texture down to fit the smaller area. Scaling the texture down increases the load on the GPU, and leads to inaccuracies that damage the quality. If a texture does have mipmap levels, the GPU fetches pixel data from the level that is closest to the object size to render the texture. Fetching the correct level ensures a higher quality image, compared to not using mipmaps. Fetching also reduces the GPU workload, because the GPU fetches a level that it produced earlier, rather than scaling down while displaying the game.

To generate the mipmaps, follow these steps:

1. In the Project window, select a texture and for Texture Type, select Advanced.
2. In the Inspector window, select Generate Mip Maps.

The following screenshot shows mipmap settings:

Figure 3-17: Mipmap settings

The disadvantage of mipmapping is that it requires 33% more memory to store the texture data.

Note

Textures that are used in a 2D UI do not usually need mipmapping. UI textures are typically rendered on screen without scaling, so that they only use the first level in the mipmap chain. To change this setting, in Inspector window of the setting, either:

- For Texture Type select Editor GUI and Legacy GUI.
- For Texture Type select Default, and clear the Generate Mip Maps checkbox.

Use cubemaps for skyboxes

Games and other applications often use skyboxes to generate backgrounds. There are several methods to implement skyboxes. One method is drawing the skybox by rendering the background of the camera using a single cubemap.

Drawing a skybox with a cubemap requires one cubemap texture and one draw call. Compared to other methods, a cubemap uses less memory and memory bandwidth, and fewer draw calls.

Limit shadow complexity

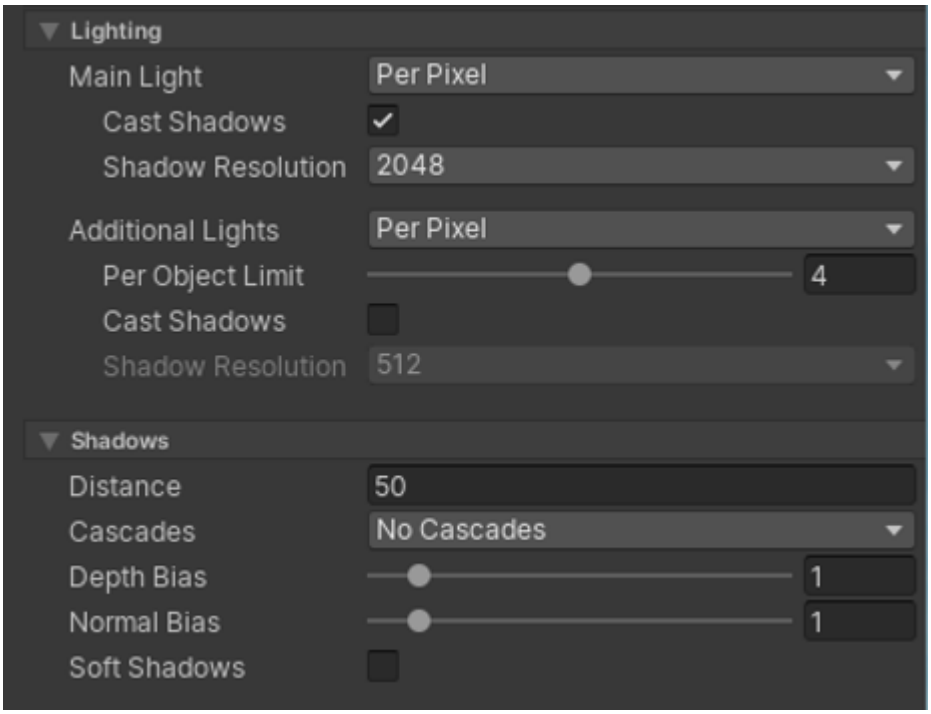
Shadows add perspective and realism to your scenes. Without shadows, it can sometimes be difficult to understand the depth of objects, especially if they look like other surrounding objects.

Shadow algorithms can be complex, especially when rendering accurate, high-resolution shadows. Ensure that you use the simplest shadowing possible to enhance performance.

For example, the Ice Cave demo implements custom shadows: shadows based on local cubemaps are combined with shadows that are rendered at runtime.

Unity has several options for shadows in the URP object that can impact the performance of your game. The URP options are shown in the following screenshot:

Figure 3-18: URP shadow options



If you are not using URP, the shadow options are at Edit > Project Settings > Quality.

Two of the URP shadow options that impact performance are soft shadows and shadow distance:

- Soft shadows look more realistic but take longer to calculate than hard shadows.
- The Shadow Distance option defines the distance from the camera that shadows appear in. Increasing the shadow distance increases the number of visible shadows, which increases the computational load. Increasing the shadow distance also increases the number of texels that are available for the shadows in the shadow map. This increase in the number of texels passively increases the resolution of your shadows.

You can use hard shadows with a small shadow distance and a high resolution. This combination produces reasonable quality shadows that are not too close to the camera and not too complex.

Lightmapped objects do not produce real-time shadows, so the more static shadows you can bake into the scene, the fewer real-time calculations the GPU does.

The following screenshot shows an alien character with a shadow:

Figure 3-19: Alien casting shadow

Use real-time shadows sparingly

Real-time shadows can dramatically enhance the realism of a scene, but they are computationally expensive, so use them sparingly.

On mobile devices in particular, try to limit the number of lights that include real-time shadows. Try to use lightmapping instead. Usually, the moving characters should cast real-time shadows, but not receive them. Instead, moving characters should receive the baked shadows from static objects. Static objects, on the other hand, should:

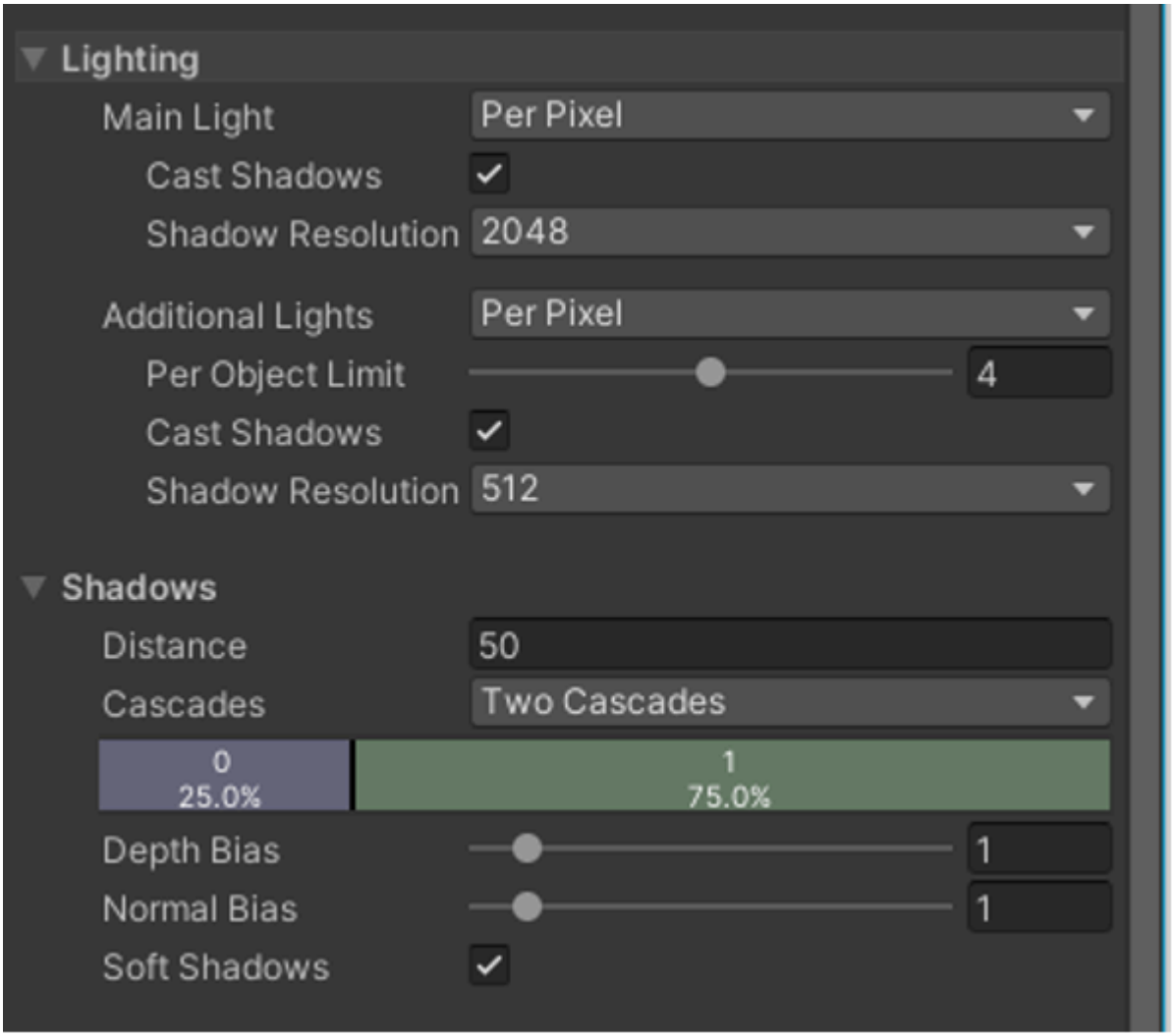
- Receive real-time shadows from the characters
- Only cast shadows to the baked lightmaps, not real-time

Consider whether, in the URP object, additional lights need to cast shadows. You can set this option for each individual light.

For the URP object, consider the following settings:

- Use Shadow Resolution to balance quality and processing time. There is a range of options between 256 and 4,096 pixels, and your choice will depend on the number of objects and lights in the scene.
- Use Shadow Cascades to balance quality and processing time. You can set the shadow cascades to zero, two, or four. Cascaded Shadow Maps are used for directional lights to achieve good shadow quality, especially for long viewing distances. A higher number of cascades produces better quality but increases processing overhead. The following sreenshot shows lighting and shadows settings for good performance:

Figure 3-20: Lighting and shadow settings



Set up occlusion culling

Occlusion culling disables the rendering of objects when they are obscured from the view of the camera. This process saves GPU processing time by rendering fewer objects.

Unity automatically performs frustum culling when objects exit the camera frustum completely. However, the style of your application might mean that there are other objects that cannot be seen and do not need to be rendered.

Unity includes an occlusion culling system with most settings at Window > Rendering > Occlusion Culling.

The settings that you use for occlusion culling depend on the style of your game. You must be careful picking settings. This is because incorrect settings can degrade performance.

Use OnBecameVisible() and OnBecomeInvisible() callbacks

If you use the callbacks `MonoBehaviour.OnBecameVisible()` and `MonoBehaviour.OnBecameInvisible()`, Unity notifies your scripts when their associated game objects move in or out of a camera frustum.

You can use these callbacks to optimize the rendering process. For example, rendering reflections on a pool with a second camera and render targets involves rendering geometry and combining textures off screen before rendering to the final screen surface. This technique is relatively expensive, so only use it when necessary. You are only required to render a reflection when it is visible. For example, you do not need a reflection if:

- The reflection surface is not in the camera frustum.
- An opaque object is in front of the surface.

The following code checks conditions with the `OnBecameVisible()` and `OnBecameInvisible()` callbacks from the reflective surface:

```
void OnBecomeVisible()
{
enabled = true;
}
void OnBecomeInvisible()
{
enabled = false;
}
```

Even with these checks in place, sometimes a reflection is rendered off screen even though it is not visible onscreen. To avoid this situation, you can add another condition, for example, that the camera must be inside the room of the reflective surface. The following code shows another condition being added to avoid a reflection rendering offscreen:

```
void OnBecomeVisible()
{
        if (inside == false)
        {
        return;
        }
        enabled = true;
```

```
}
void OnBecomeInvisible()
{
        if (inside == false)
        {
        return;
        }
        enabled = false;
}
void OnTriggerEnter()
{
        inside = true;
}
void OnTriggerExit()
{
        inside = false;
}
```

The preceding conditions restrict the rendering of reflections to specific areas of the game. This means that you can add effects in other, less compute intensive areas of the game.

Specify the rendering order

In a scene, the object rendering order is important for performance. If objects are rendered in random order, an object might be rendered and then be occluded by another object in front of it. In this situation, all the computations to render the occluded object are wasted.

Various software and hardware techniques exist to reduce the amount of wasted computation for occluded objects. However, you can manually improve this process because you know how a player explores the scene.

Early-Z is a hardware technique for reducing wasted computation. It is one of the techniques that is available on the Arm Mali GPUs from the Mali-T600 series onwards. Early-Z is a system that performs a Z-test before the fragment shader is processed. If the GPU cannot enable Early-Z optimization, the depth test is executed after the fragment shader. This can be computationally expensive, and the computations can be wasted if the fragment is occluded. The Early-Z system checks that the depth of the pixel being processed is not already occupied by a nearer pixel. If the pixel is occupied, the system does not execute the fragment shader.

The Early-Z system provides performance benefits, but sometimes it is automatically disabled. For example, Early-Z is disabled if the fragment shader modifies the depth by writing into the `gl_FragDepth` variable. This means that the fragment shader calls `discard`.

To help this system achieve maximum efficiency, ensure that opaque objects are rendered from front to back. This rendering helps to reduce the overdraw factor in scenes with only opaque objects.

Ordering the rendering of each frame from front to back can be expensive and incorrect if you render transparent objects in the same pass. However, Arm Mali GPUs from T620 onwards provide a mechanism called Forward Pixel Kill (FPK). Mali GPUs are pipelined so that multiple threads can be concurrently executing for the same pixel. If a thread completes its execution, the FPK system stops all other threads for that pixel if the current one covers them. The effect is a reduction of wasted computations.

Unity provides queue options inside the shader to specify the order of rendering, so that rendering does not rely entirely on hardware inference. If you set the order of rendering in the shader, objects that have a material that uses this shader are rendered together. Inside this rendering group, the order of rendering is random. However, this is not always the case. For example, transparency affects rendering order.

You can override a rendering group for a material in the material script. By default, Unity provides some standard groups that are rendered from first to last in the order shown in the table:

Name	Value	Notes
Background	1000	-
Geometry	2000	Default used for opaque geometry.
AlphaTest	3000	The AlphaTest is drawn after all opaque objects, for example, foliage.
Transparent	4000	This group is also rendered in back to front order to provide the correct results.
Overlay	5000	Overlay effects, for example, user interface, lens flares, dirty lens.

The integer values can be used instead of their string names. Integer values are not the only values available. You can specify other queues using an integer value between those that are shown. The higher the number, the later it is rendered. For example, you can use one of the following instructions to render a shader after the Geometry queue, but before the AlphaTest queue:

```
Tags { "Queue" = "Geometry+1" }
Tags { "Queue" = "2001" }
```

In the Ice Cave demo, the cave covers large parts of the screen and its shaders are expensive. Where possible, parts were not rendered. This is because rendering them could degrade performance.

Rendering order optimization was included in the Ice Cave demo after looking at the composition of the framebuffers. The tools that were used to look at the composition of the framebuffers include the Unity Frame Debugger and tools like the Graphics Analyzer. These tools show the rendering order.

Use the Unity Frame Debugger to debug in play mode and get the sequence of drawings that Unity executes.

In the Ice Cave demo, scrolling down the draw calls shows that the cave is rendered first. This means that the objects are rendered into the scene, occluding the parts of the cave that are already rendered. Another example is the reflective crystals that in some scenes are occluded by the cave. In these cases, setting a higher rendering order means fragment shaders are not executed for the occluded crystals, resulting in a reduction in computations.

Consider using depth prepass

Setting the rendering order for objects to avoid overdraw is useful, but it is not always possible to specify the rendering order for each object. For example, if you have a set of objects that the camera can rotate around freely, you cannot specify their order ahead of time. This is because objects that were previously at the back can now appear at the front. In this case, if there is a static

rendering order set for these objects, some objects might be drawn last, even if they are occluded. You also cannot specify rendering order if an object can cover parts of itself. In cases like this, a depth prepass can sometimes reduce overdraw.

Usually, depth prepass is an optimization that reduces performance, rather than improves it. But it can be worth evaluating whether depth prepass will help if your game:

- Is fragment bound
- Has spare capacity on the CPU and vertex shader
- Has significant transparency parts in scenes

Make sure to check the effect on performance and remember that Forward Pixel Kill (FPK) will already reduce a lot of the wasted computation.

In Unity, add an extra pass to your shaders, if you want to do a rendering prepass for objects with custom shaders.

The following code shows an extra pass that renders to the depth buffer only:

```
// extra pass that renders to depth buffer only
Pass {
ZWrite On
ColorMask 0
}
```

After adding this pass, the frame debugger shows that the objects are rendered twice. The first time that they are rendered there are no changes in the color buffer. This is because the depth prepass renders the geometry without writing colors in the frame buffer.

This process initializes the depth buffer for each pixel with the depth of the nearest visible object. After this prepass, the geometry is rendered as usual. However, using the Early-Z technique, only the objects that contribute to the final scene are rendered.

Extra vertex shader computations are required for this technique. This is because the vertex shader is computed two times for each object, one time for filling the depth buffer and another time for the actual rendering.


Note

You can see the depth buffer by choosing it in the top-left menu of the frame debugger.

4. Asset optimizations

This section of the guide describes some asset optimizations that you can use in Unity as a game developer. Other asset optimizations are reviewed in the artists guides that we list in Related information.

Disable read or write for static textures

If you do not dynamically modify a texture, uncheck the Read or Write Enabled option in the Inspector window.

Combine meshes to reduce draw calls

You can combine several meshes into one with the `Mesh.CombineMeshes()` method. If the meshes all share the same material, set the `mergeSubMeshes` argument to `true`. The mergeSubMeshes argument then generates a single sub-mesh out of each mesh in the combined group.

Combining several meshes into a single larger mesh helps:

- Create more effective occluders
- Turn tile-based assets into a single, large, seamless, solid asset

The mesh combine script can be useful for performance optimization, but this depends on the makeup of your scene. Large meshes tend to stay in view longer than smaller meshes, so experiment to get the correct size.

You can combine some meshes directly, if they do not need to stay separate for programming or while the game is played. If you have meshes that need to stay separate, you can group the meshes rather than combine them. Follow these steps:

1. Create an empty game object in the hierarchy.
2. Make the object the parent of all the meshes that you want to combine.
3. Attach the parent to a script that includes the `Mesh.CombineMeshes()` method.

Do not import animations data on FBX mesh models unnecessarily

By default, Unity creates animation data when importing an FBX mesh. This process increases the size of the imported object. If your object is not animated, it does not need this data. In Import Settings > Rag, for Animation Type select None so Unity does not generate the animation data.

Another way that you can stop the animation is in the Animation tab, where you can clear the Import Animation checkbox.

Avoid read or write meshes

By default, Unity keeps a second copy of model mesh data in memory to modify while preserving the original. Unity does this because it assumes that all models are modified at runtime. If your model is not modified at runtime, even to be scaled, in Import Settings > Model clear the Read or Write Enabled checkbox. Unity will not save a second copy, so memory use will be lower.

Use profiling to focus your optimizations

Profiling lets you focus your asset optimizations and compromises on the places that matter.

It may not be obvious which part of your scene needs optimization. For example, look at the following screenshot from Spellsouls, by Nordeus. This image was reviewed in a case study we published on our blog: Post-processing Effects on Mobile: Optimization and Alternatives:

Figure 4-1: Example frame to profile

The heaviest fragment shading is the terrain, because it covers the whole screen. We can see what optimizations make the most difference to this frame, for example a lower resolution lightmap, rendering the whole terrain at 720p instead of 1080p and blitting it in before the characters. The characters could continue to have high-resolution visuals, because they were not using most of the frame time.

5. Related information

Here are some resources related to material in this guide:

- Arm based demos made with Unity
- Arm Community - Ask development questions and find articles and blogs on specific topics from Arm experts
- Arm Guide for Unity developers:
  - Advanced graphic techniques – Getting started
  - Real-time 3D art best practices: texturing
  - Real-time 3D art best practices: geometry
  - Real-time 3D art best practices: materials and shaders
  - Real-time 3D art best practices: lighting
  - Key project-wide settings in Unity
- Post processing effects on mobile
- Unity demos
- Unity documentation
- Unity Manual Lighting Pipeline Best Practice
- Virtual Reality: The ice cave demo blog post and video

6. Next step

This guide has introduced to you some optimization techniques that you can implement in your Unity programs. We have looked at some application processor optimizations like using built-in arrays and remove empty callbacks. We have also looked at GPU optimizations like using static batching and Level of Detail.

After reading this guide, you will be ready to implement some of the techniques into your own Unity programs. If you want to learn more about Unity, you can read our Arm Guide for Unity developers.