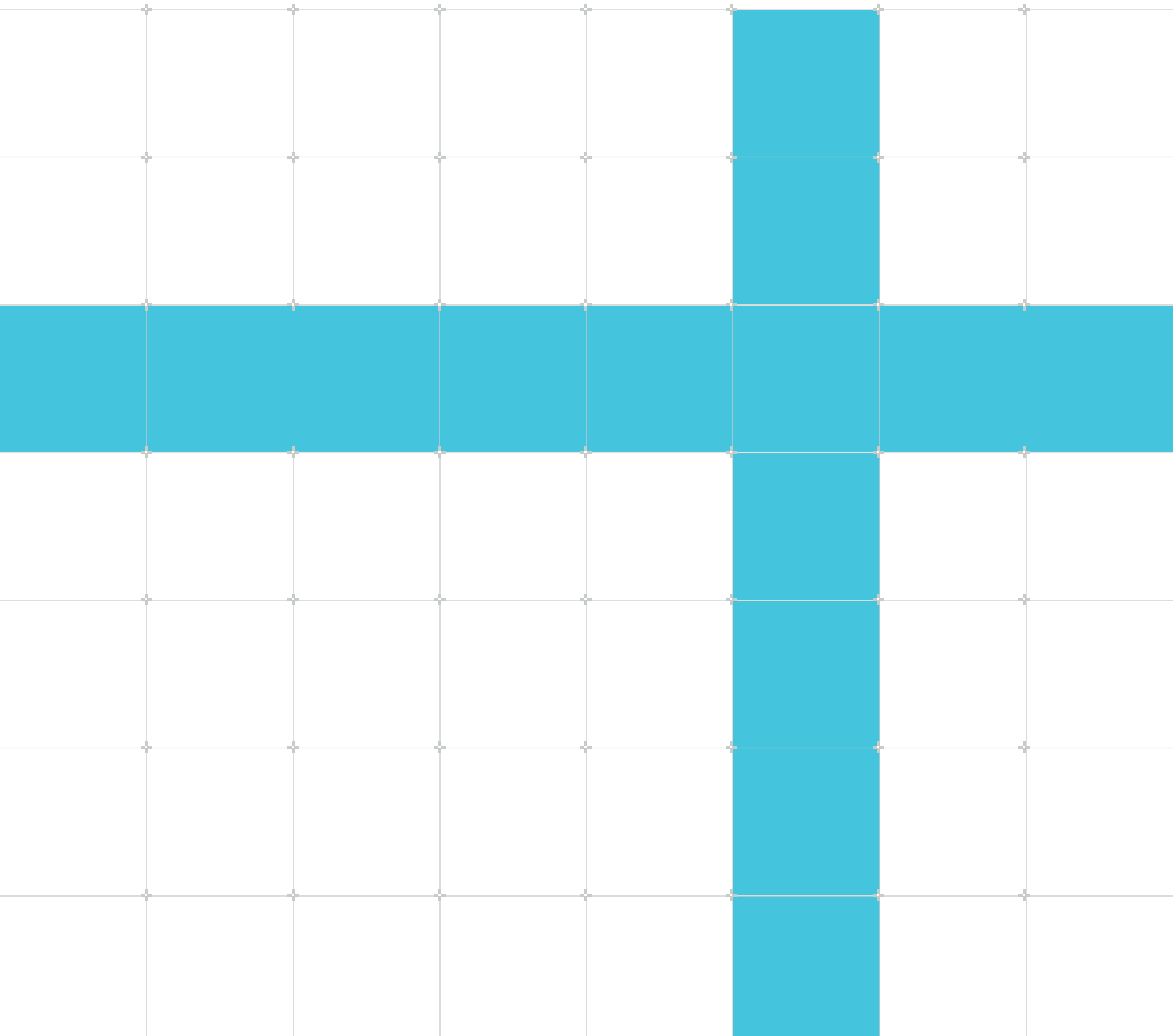
Arm Guide for Unity Developers - Real-Time 3D Art Best Practices - Lighting

Version 1.0

Issue 02
102109_0100_02_en



Arm Guide for Unity Developers - Real-Time 3D Art Best Practices - Lighting

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	5 May 2020	Non-Confidential	First release
0100-02	12 May 2020	Non-Confidential	Minor text clarifications

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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

Lighting is one of the most important aspects of a game. This is because lighting can set the mood, lead gameplay, and identify threats and objectives. Lighting can make or break the visuals of a game. For example, a simple model can look better in-game with good lighting techniques, and a detailed model can look worse with bad lighting techniques.

The Unity game engine makes working with lighting straightforward and simple to understand. The performance of a mobile game is affected by your lighting decisions, so lighting needs to be used efficiently.

This guide is also available in the format of a Unity Learn Course - Arm & Unity Presents: 3D Art Optimization for Mobile Applications.

At the end of this guide, you can Check your knowledge. In this guide, you will learn:

- The difference between static and dynamic light.
- Optimization of light.
- How to fake lighting.
- Real-time light and light types.

Arm Guide for Unity Developers: Korean

Arm Guide for Unity Developers: Chinese

Arm Guide for Unity Developers: Japanese

2. Render pipeline

Within Unity's Built-in Render Pipeline you are given two choices of render paths: Forward Rendering and Deferred Shading.

In forward rendering real-time lights are very expensive, but to offset this cost you can choose how many lights should be rendered per pixel.

Deferred shading requires GPU support, but on enabled hardware it can render a large number of real-time lights and requires a high level of lighting fidelity.

While deferred shading is very attractive for light intensive games for PC or console, it is not very performant on mobile GPUs. This is due to the lower bandwidth of mass market devices. When you are making a mobile title, it is important that the work you do can run smoothly on as many devices as possible. So to support its users, Unity has provided the Universal Render Pipeline (URP). This is a pre-built Scriptable Render Pipeline that is tuned for performance. URP has a slew of features to help your app be as performant as it can, which you can read more about here. It is because of those optimizations that we suggest lighting projects use URP.

3. Light mode

There are different modes for lights. These modes relate to the mobility of a light and how it is used within a scene. The modes differ in terms of performance, so light mode is important to consider when implementing lights. We will look at the advantages and disadvantages of using three different lighting modes: baked, mixed, and real-time.

Baked

Baked light mode provides static lighting: objects do not change their lighting during runtime. Baking lights is the process of storing the lighting data in texture maps prior to running the game.

The key features of baked light mode are as follows:

- Light cannot be modified at runtime. Lights and shadows are baked into lightmaps. This processing is done when lighting is created in Unity and does not affect run-time performance.
- Shadows are static, which can look odd with dynamic or moving objects during gameplay.
- Baked light mode is the least expensive computational method that we discuss in this guide.

Mixed

Mixed light mode provides stationary lights with moving objects. This can be considered as a mixture of the two other methods.

The key features of mixed light mode are as follows:

- Dynamic direct lighting and shadows
- Light can be included in lightmap calculations for static objects.
- Light affects dynamic objects, including generating shadows for those objects.
- Intensity can be changed at runtime, and only direct light is updated.
- Mixed light mode is an expensive computation method.

Real-time

Real-time light mode provides dynamic or movable lights, which is the most expensive and complicated way of working with lighting.

The key features of real-time light mode are as follows:

- Dynamic light and shadow, with properties that can be modified at runtime, rather than being baked into lightmaps.
- Real-time light mode is the most expensive computational method that we discuss in this guide.

You can learn more about the Unity lighting pipeline in the Unity documentation. There is a link in Related information.

4. Use static light where possible

Using static light is crucial when working on mobile devices. It is cheaper to run on the device and leads to a better experience for the end user of games.

Dynamic or real-time lighting is calculated and updated in every frame, which is an effective method for moving objects, raising interactivity, and giving emotion to a scene.

In contrast, static light information can be baked. Unity performs the calculations for baked lights in the Unity Editor and saves the results as lighting data. This process is called baking.

When the calculations are complete, the run-time values are only the ones that are needed for that scene. Static lights are always much less expensive than dynamic lights. This means that static lights should be your first choice to implement in a mobile game with more limited resources.

5. Bake as much as possible

Baking as much as possible should be your initial approach to lighting on a mobile platform using Unity. Lightmap baking is the process of calculating the effect of lights, like illumination and shadow, and storing this information in a separate texture called a lightmap. A lightmap can be used to augment the appearance of objects. By performing this baking process, you only need to do the offline computation once. There are no extra performance costs at runtime.

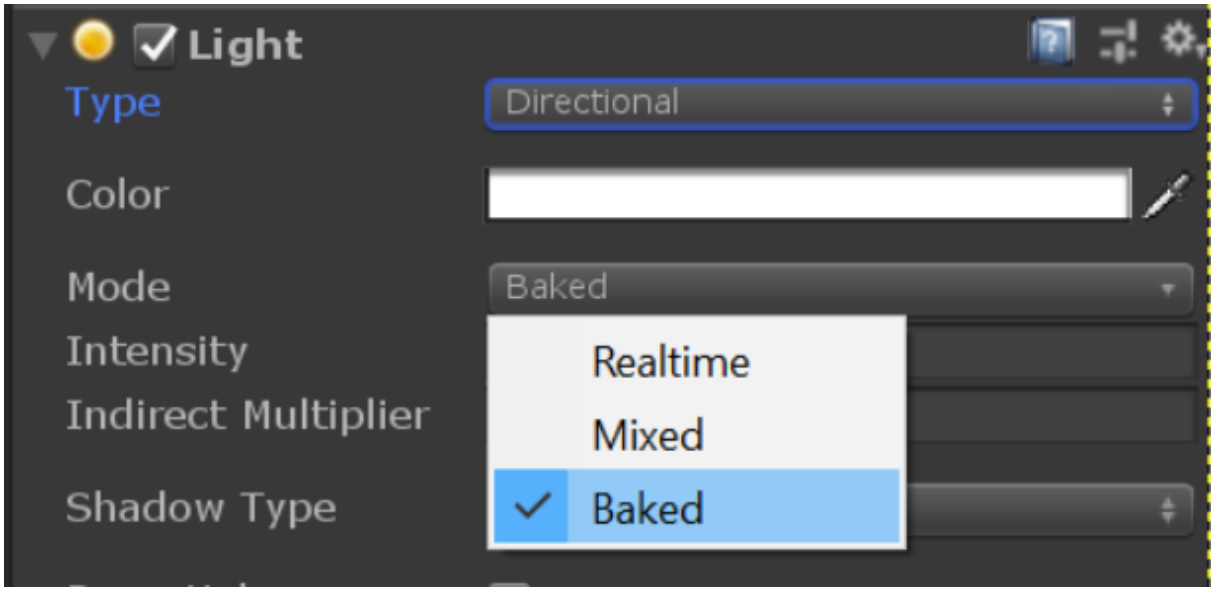
Pre-baked lighting does not cope with the dynamic, or moving, aspects of your scene. However, prebaked lighting does include global illumination for all the static elements. This means that each static element gets the indirect light that has bounced off objects and the direct light for the static lighting. The following image shows an example of a fully baked scene.

Figure 5-1: Baked lights

Unity makes it easy to bake lights. There are two main steps that you must set up before you can bake your lights

1. Click Windows > Rendering > Lighting Settings and set the lights that you want to bake to either Mixed or Baked.

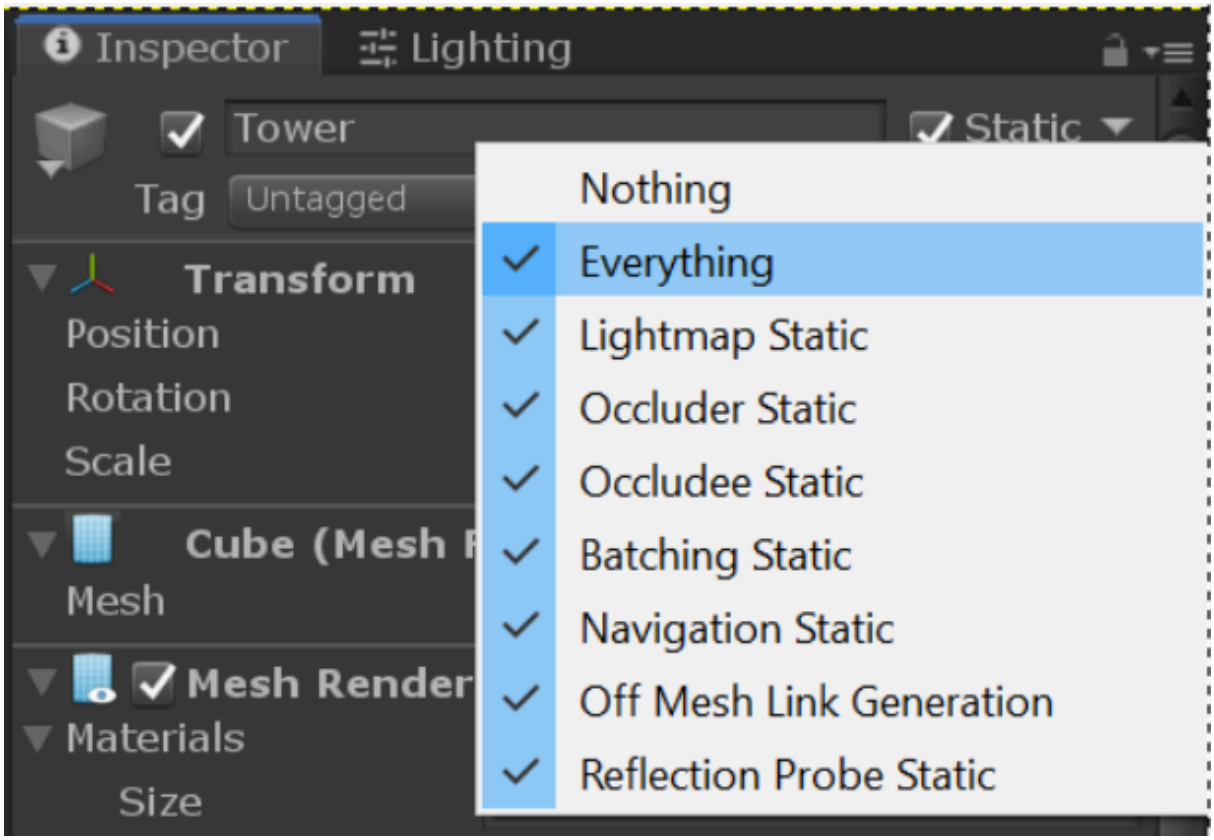
Figure 5-2: Baked setting



For mobile titles, use baked lights instead of mixed whenever possible. This is because baked is the least expensive of all the options.

2. Mark objects that receive the baked light as Static:

Figure 5-3: Inspector settings



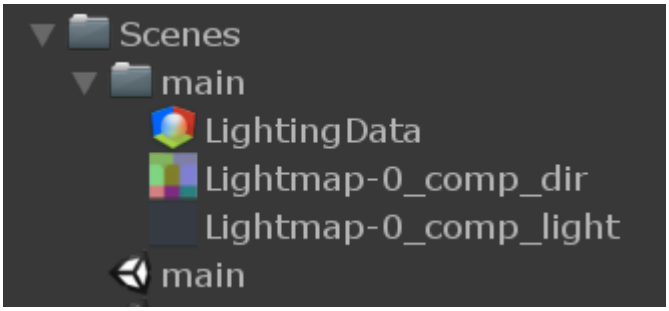
There are multiple possible optimizations for an object to be marked as static, but usually we have Everything selected in the settings. With the object marked as Static, Unity knows to include it in the light baking.

Note

With Batching Static enabled, you will not be able to move or animate the marked object. This is another optimization and should be left on wherever possible.

When you are baking your lights, remember that the data is saved based on the scene that was active when you started the bake. A folder is generated that has the same name as the scene that you just baked. This is where all the components for the lighting data will be stored. If your project uses multiple scenes that are loaded at one time, each scene needs to have its lights baked. If you adjust your scene, you need to rebake the lights. The following image shows where that folder will be created.

Figure 5-4: File structure

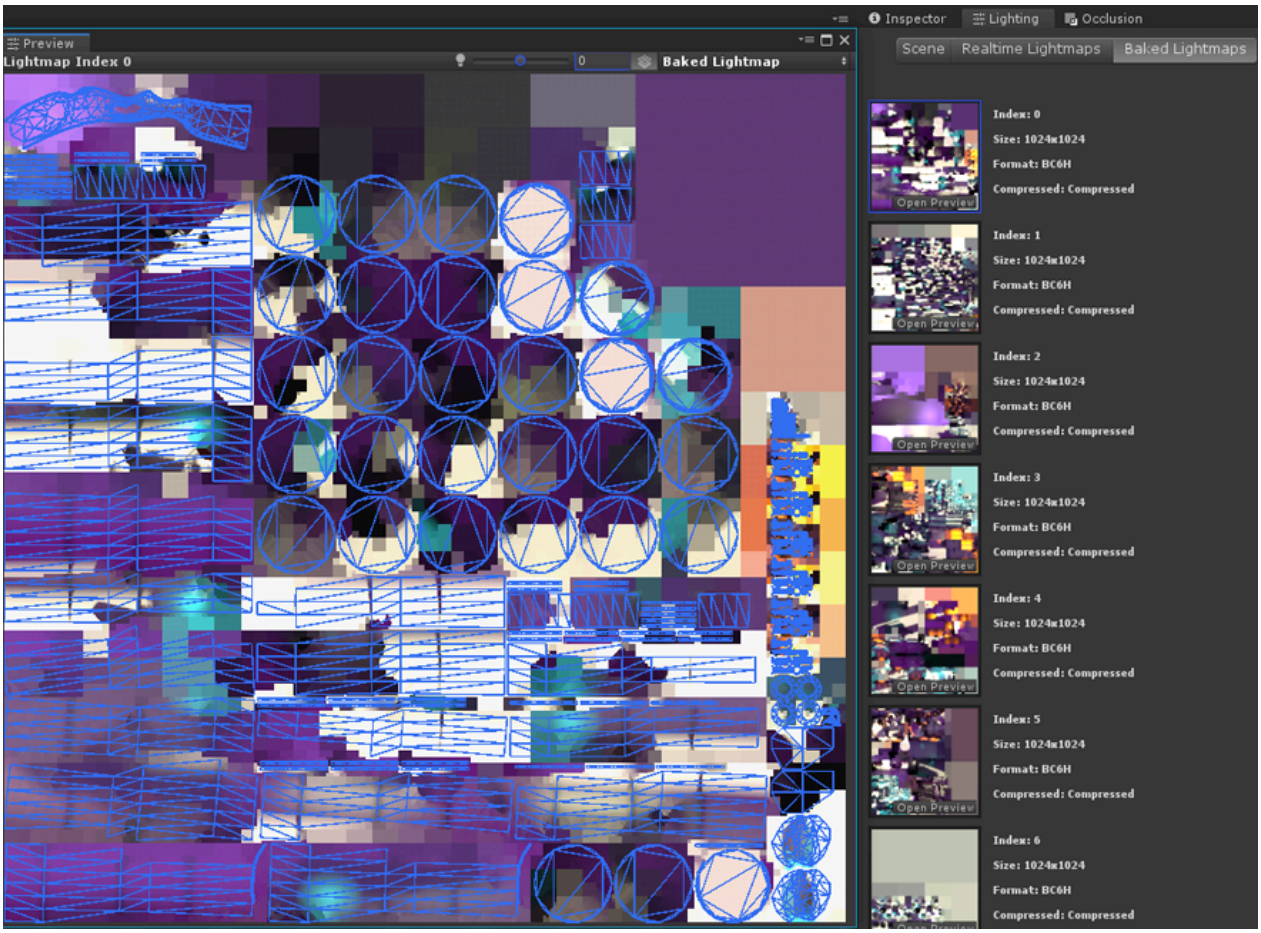


6. Optimize light maps

After you have configured lights to be baked, you should also make sure that the baked maps are optimized

Light maps vary in size depending on the settings they are baked with. We must minimize memory usage on mobile platforms, so lightmap size needs monitoring. In the example image, you can see that there are seven 1024x1024 pixel lightmaps.

Figure 6-1: Pixel lightmaps



In the preview of the map, you can see meshes laid on it, and selected meshes are highlighted.

Some settings in Lightmapping, and the size of the actual maps, determine how much space is used.

The most important settings are described in the following sections.

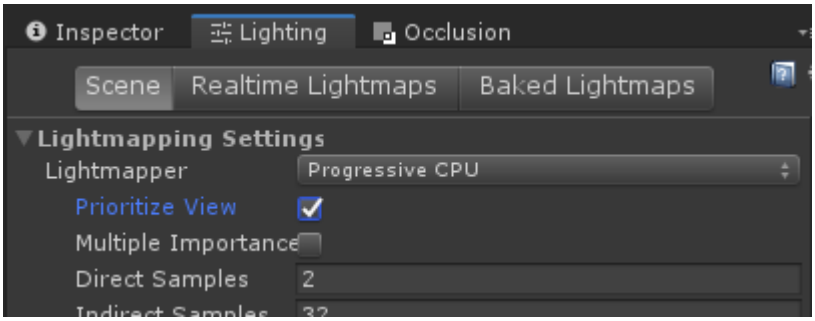
Light mappers

Light mappers in Unity offer three different methods to bake the lights in your scene:

- Enlighten
- Progressive CPU
- Progressive GPU

The following image shows this option.

Figure 6-2: Lighting options



In Unity, you should use one of the progressive options for any new work. The progressive light mappers offer a large time saving benefit because they incrementally create light maps. If the Prioritize View option is selected, areas that are in the scene view are prioritized. Prioritize view speeds up iteration times setting up your lighting for your scene.

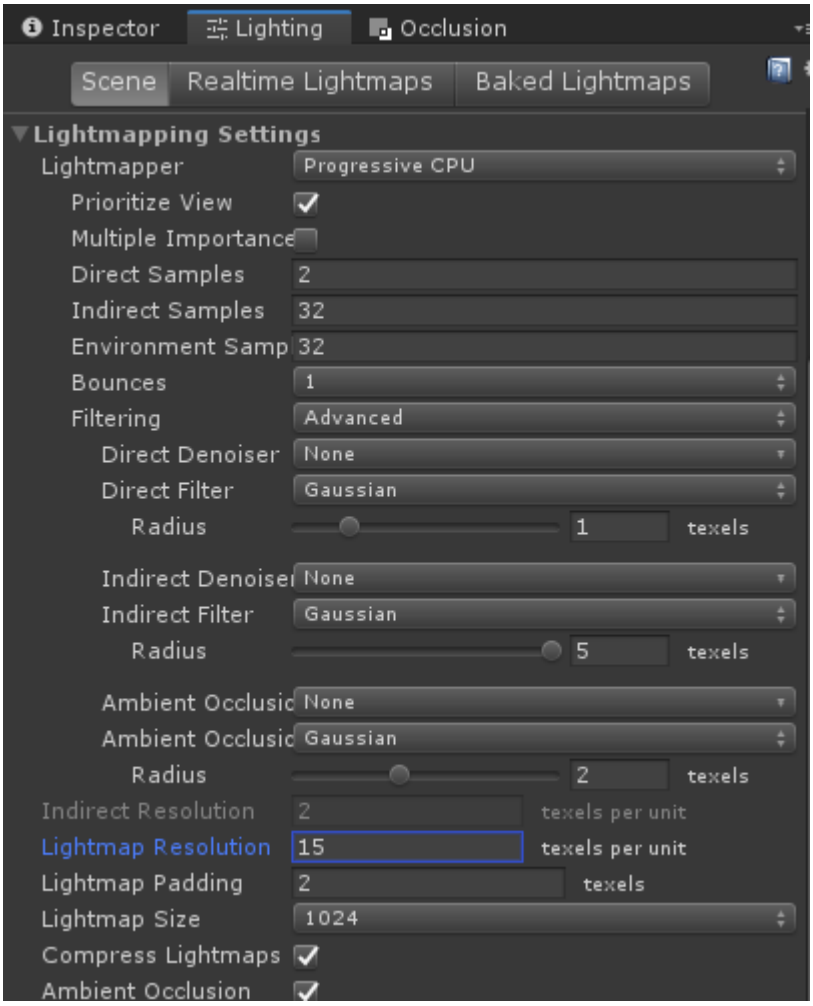
The major difference between the CPU and GPU progressive light mappers is whether lightmap generation is in the CPU or in the GPU. The results of the two options are the same, but if you have a powerful GPU that option is much faster. More requirements and set up steps for the GPU option, which can be found here.

Texels

A texel, or texture element, is the individual pixel in a texture map. Texels are used, for example, to store lighting information at each point where a light hits an object in a lightmap. We can measure the amount of work that is needed to bake in a light by counting the number of texels that are used. It is important to understand what a texel is, and how your control over them can influence the quality of your lighting, computation time for the bake, disk storage costs, and the VRAM cost of your lightmaps.

For the biggest impact on the amount of lightmap data that is required, you must adjust the number of texels for each unit of the bake. This can be done in the Lightmapping Settings. These settings give you control over the lightmaps, including how many texels each object uses in the bake as shown in the example image.

Figure 6-3: Scene settings



Lightmapping Settings includes an option called Lightmap Resolution. This option sets how many texels are used for each unit in the lightmaps.

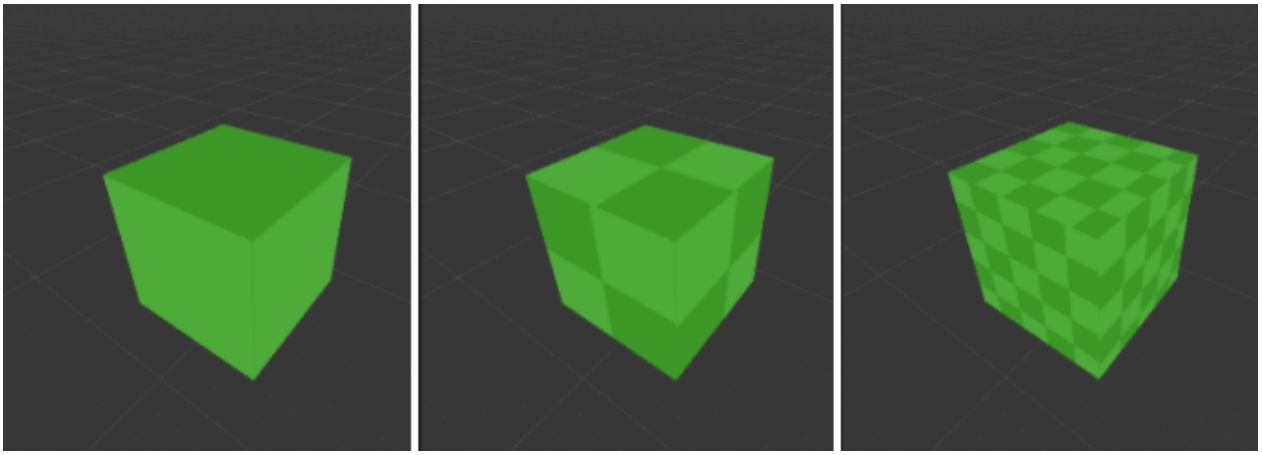
You can see how texels are laid out in your scene in the following ways:

- Click the draw mode drop down on the Scene view.
- Find and then click Lightmap Indices.

Bake objects will now be covered in a checkerboard overlay. This is how your texels are distributed when you bake the lights.

In the following screenshot, you can see an example of a cube with different Lightmap Resolution settings. The left-hand image has a setting of one, the middle image has a setting of two, and the right-hand image has a setting of five.

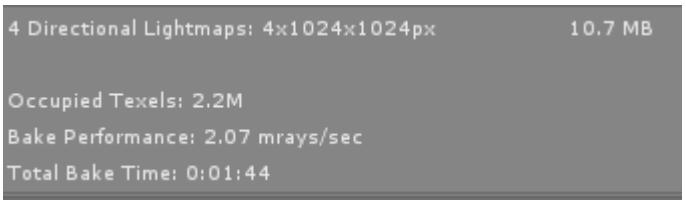
Figure 6-4: Three green cubes



You can see how a higher resolution quickly increases the amount of work that is needed. We recommend that you start with a low Lightmap Resolution, between five and ten, and scale up or down based on what your scene needs. Increasing the lightmap resolution causes the size to go up massively with each iteration.

For example, reducing the Lightmap Resolution from 15 to 12 in the example demo reduces the number of lightmaps that are needed from seven to four as shown in the proceeding image.

Figure 6-5: Lightmap resolution



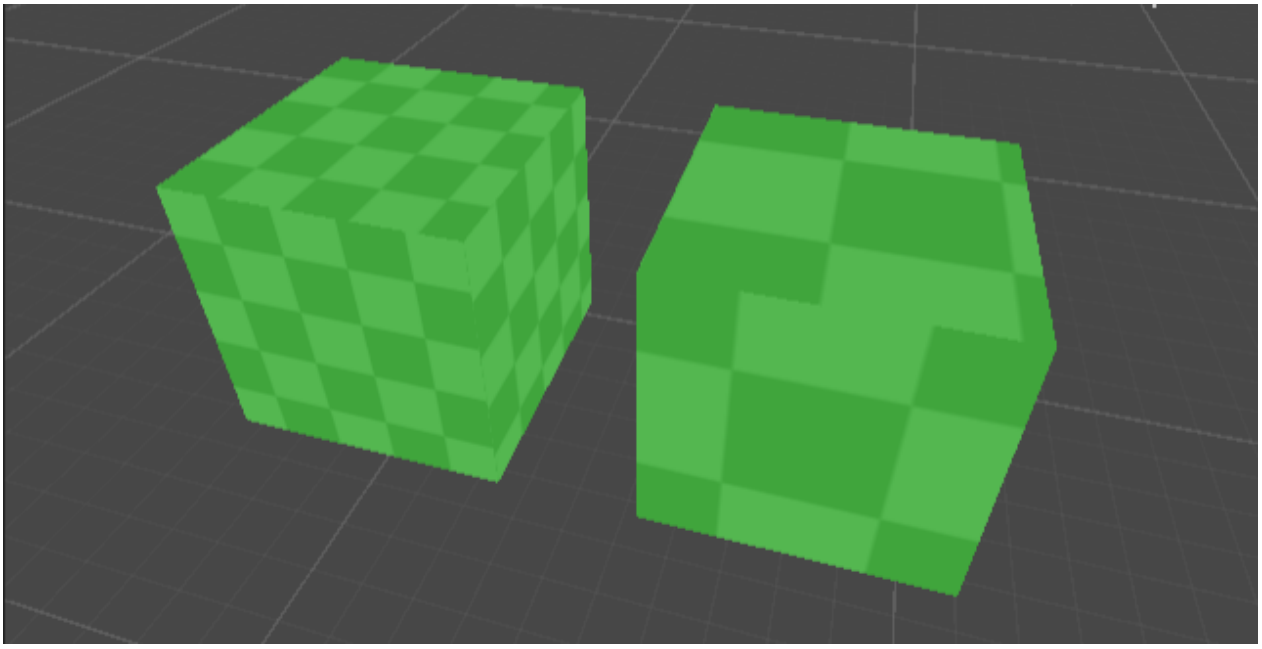
Texel usage

In the Lightmapping Settings, you set the number of texels that your scene uses per unit, but there are some objects that you do not want to use that many texels on.

Unity allows you to control how many texels each object can use. The Inspector > Mesh Renderer for an object includes a parameter called Scale In Lightmap. You can adjust Scale In Lightmap to change the amount of texels that this object uses in your lightmap.

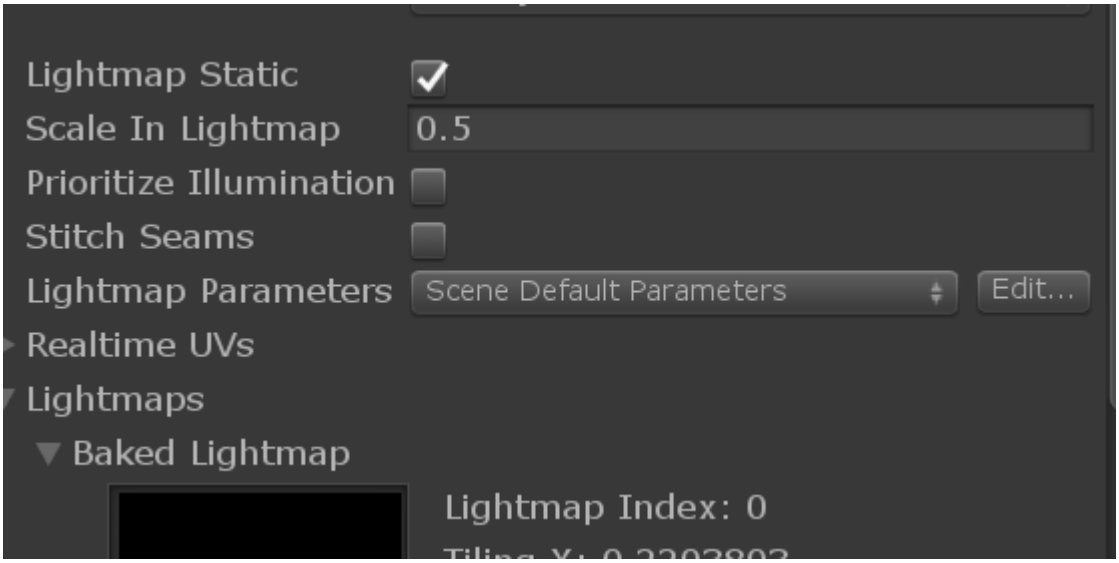
In the following screenshot, on the left-hand side is an average object getting five texels of lighting information for each baking unit, because Lightmap Resolution is set to five. On the right-hand side is a box with Scale In Lightmap set to 0.5:

Figure 6-6: Two green cubes



The right-hand box will use much less space in the lightmap than the left-hand box. In the proceeding screenshot you can see the settings for lightmaps.

Figure 6-7: Settings for lightmaps



Try to avoid spending texels on the following elements:

- Surfaces and objects that a player will not see. This avoids wasting memory on a larger lightmap for detail that is not seen on screen.
- Surfaces with little light variation on them, for example an object in a shadow, or an object that is only touched by a single light.

- Small or thin objects. The amount of lighting that small or thin objects receive does not add much to the final render of the scene.

7. Fake as much as possible

Real shadows are computationally expensive. We recommend that you implement fake shadow to show shadows on dynamic objects, without resorting to dynamic lights.

Real time shadows are most often generated using a technique known as a shadow map. The cost of scene geometry being rendered to the shadow map is comparable to the number of vertices drawn to it. So it is important to limit the number of shadow casting geometry, as well as real-time shadow casting lights.

Here are some ways to implement fake shadows:

- Use a 3D mesh, plane, or quad, that is placed under the character and applying a blurred texture to it.
- Use a Unity Built-in Render Pipeline feature to apply dynamic blob shadow using projector. This method is more expensive than using a quad under a character, and is not available in the Universal Render Pipeline that is recommended for mobile.
- Write custom shaders to create more sophisticated blob shadows.

In the proceeding screenshot of an example demo, Armies, shadow implementation is shown using shadow meshes.

Figure 7-1: Shadow implementation

Paint lighting information directly to textures. By painting some of the light shading into textures, we reduce the extra computation that extra lights require. Painting directly to textures also saves memory when baking the lights in your scene, because the scene needs less texture memory.

Use shaders or materials to simulate lighting. You can use custom materials to simulate light effects. For example, in the game level we might want our character to have rim lighting to improve their visibility and visual look. Instead of using lights to create this effect, you can use a shader effect to create the illusion of lights.

Shaders can provide lots of useful effects to add to the game. Look at the Best Practices For Shaders and Materials guide to learn more.

8. Light probes

Light Probes have two main uses: The primary use of light probes is to provide high quality lighting (including indirect bounced light) on moving objects in your scene. The secondary use of light probes is to provide the lighting information for static scenery when that scenery is using Unity's Level of Detail (LoD) system.

When using dynamic objects with baked lighting, light probes are not normally affected by the lightmaps. This may cause them to look odd, and to seem like they are not part of the scene.

A solution for this is Light Probes. Light probes have many of the same benefits as lightmaps, in that they store lighting data which can be calculated offline. Again this moves much of the computational costs to edit time, rather than run time. While a lightmap encodes lighting received at a given texel for surfaces in your scene, a light probe stores the light that passes through empty space in your scene. This data can then be used to light dynamic objects, which helps integrate them visually with lightmapped objects throughout your scene.

Light probes only store and show the lights and shadows of the static scene. This is because light probes are also pre-baked. They are not a solution for shade and light from dynamic objects, real-time lights, or self-shadowing. But light probes can provide most of the lighting for your scene.

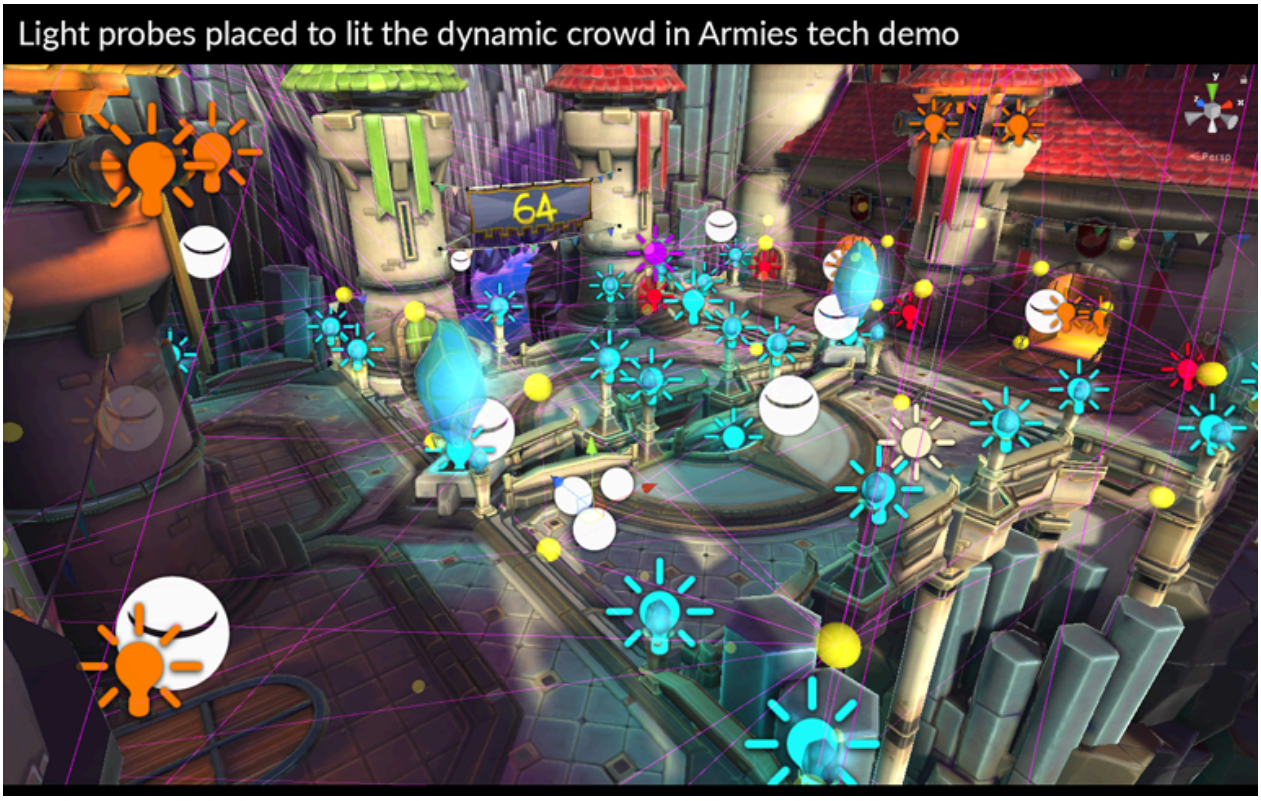
There are two main uses for light probes:

- To light moving objects in your scene. Light probes take advantage of your baked lights, so that your objects have the same lighting as the scene. Lighting dynamic objects with light probes is less expensive than using real-time lights.
- To provide lighting data to statically marked objects that use the LoD System.

To learn more about light probes, see Related information.

The following screenshot shows an example of light probes being used.

Figure 8-1: Light probes

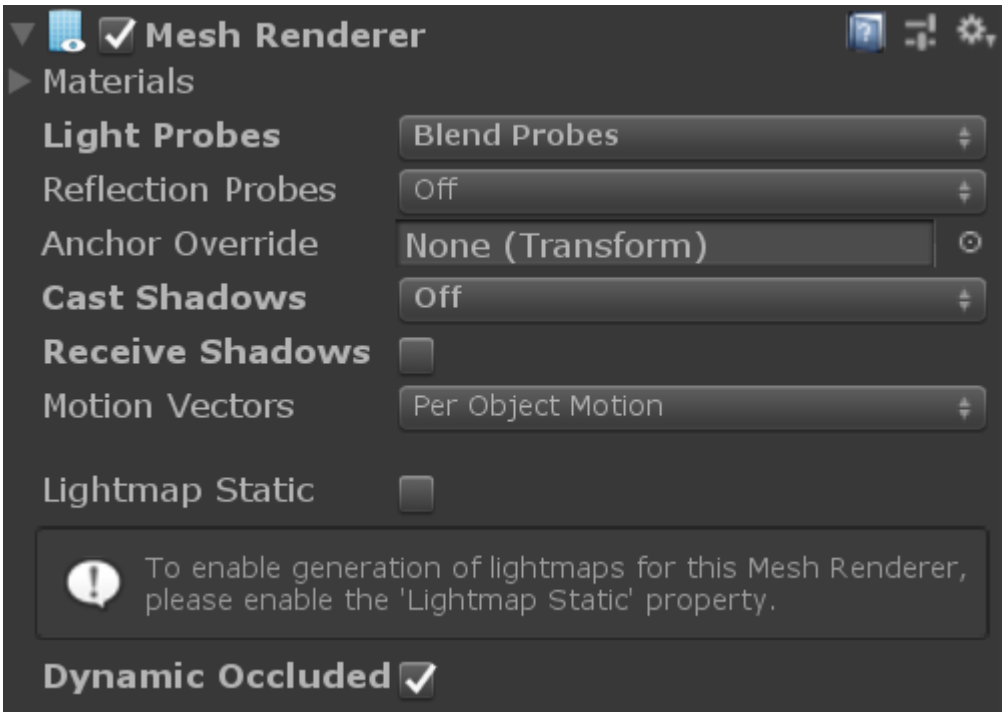


9. Mesh renderer settings

Regardless of what lighting your scene uses, it is important to make sure your mesh renderer settings are correct.

A good principle is to turn off anything that you are not going to use. Settings like Cast Shadows can add extra cost to rendering the scene, even if the object is unlit. The following screenshot shows an example of the Mesh Renderer settings for a scene:

Figure 9-1: Mesh renderer



The options are set to get lighting, but not reflections, from blending the lighting information from the nearest probes smoothly as they move between them. Cast Shadows is turned off, because we are using the blob method. Receive Shadows is also turned off, because the scene is baked, and nothing is casting a real-time shadow.

10. Realtime lights and light types

You should try to handle all your lighting with baked lighting, light probes, and material effects. Sometimes you need to use a real-time light. If you do, you must consider which type of real-time light to use.

Each type of real-time light has a different cost to calculate:

- Directional - With a uniform direction and no fall off, directional light is the cheapest real-time lighting for what it achieves. You usually only need one directional light, because a directional light can light the whole scene. This means that with forward rendering, Unity will always render one direction light. This is true even if there is no directional light in the scene.
- Point – A point light is located at a point in space and sends light out in all directions equally.
- Spot - Because a spot light culls more objects than a spherical point light, a spot light is the next cheapest type of real-time lighting. Keep the cone width tight, and only have it hit selected objects, to get the best performance from these lights.

While lighting in every direction is helpful, it is also quite expensive. Directional lights have a relatively cheap calculation everywhere. Spot lights can be confined to only be expensive for a small area, and point lights are expensive across a wider region. Also, shadow calculation can be the most expensive part of lighting, so casting light in all directions increases expense.

Dynamic lights are expensive to render, and it is best to avoid them in mobile games. Sometimes there are limits put on their use, depending on the device and graphics API used. For example, in the Unity Universal Render Pipeline forward renderer with OpenGL ES 2.0, there is a limit of four lights per object.

11. Check your knowledge

The following questions will help you test you knowledge:

Why use static lighting over dynamic lighting?

Static lighting is much cheaper in computational terms than dynamic lighting.

Why should you fake lighting in your scene?

Each shadow has to be applied to a scene individually which requires a lot more computation per scene. Faking these effects lowers the overall cost of the game.

What are the three types of Real-time lighting?

Directional, spot and point.

12. Related information

Here are some resources related to material in this guide:

- Arm Streamline
- Mali Offline Shader Compiler
- OpenGL Shading Language
- Programming guide for HLSL
- Material and Shader guide from Arm

External Resources:

- Unity's lighting pipeline
- Unity progressive lightmapper
- Unity class projector
- Unity Static lighting documentation
- Unity Light probe documentation

13. Next steps

This guide covers texture optimizations that can help your games to run more smoothly and look better.

You can continue learning about best practices for game artists and how to get the best out of your game on mobile by reading the other guides in our series:

- Real-time 3D Art Best Practices: Geometry
- Real-time 3D Art Best Practices: Materials and Shaders
- Real-time 3D Art Best Practices: Textures