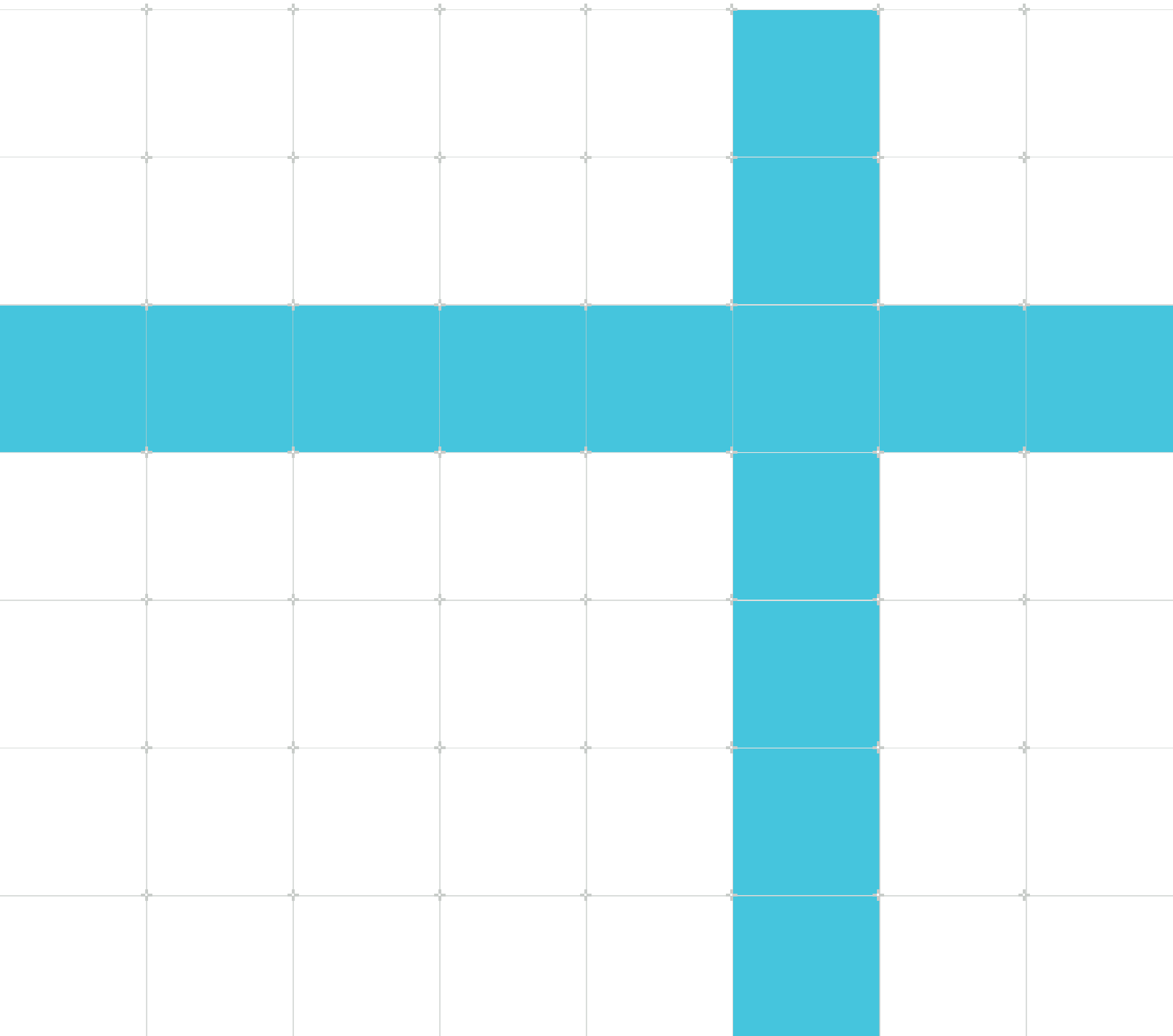
Real-time 3D Art Best Practices - Materials and Shaders

Version 1.0

Guide

Issue 02
102471_0100_02_en



Real-time 3D Art Best Practices - Materials and Shaders Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	16 March 2021	Non-Confidential	Initial release

Proprietary Notice

and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Materials and shaders determine how 3D objects appear on-screen, so it is important to know what they do, and how to optimize them.

This guide covers multiple different material and shader optimizations that can help your games to run more efficiently and look better.

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. You will have learned:

- How materials and shaders contribute to the performance of your game
- How to choose between different transparency implementations
- How features like texture packing and profiling can help to optimize performance

Arm Guide for Unity Developers: Korean

Arm Guide for Unity Developers: Chinese

Arm Guide for Unity Developers: Japanese

2. Materials and shaders

Materials and shaders determine how your 3D objects look, so it is important to know what they do, and how they can be optimized.

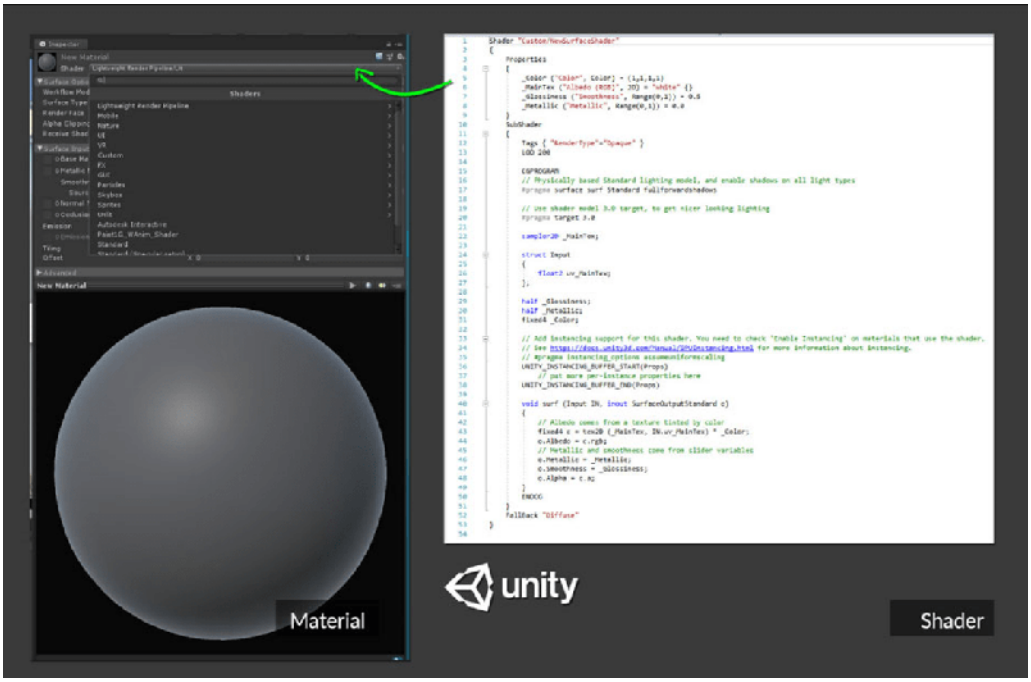
A shader is a small program that tells the GPU how to draw an object on the screen. A shader tells the GPU every calculation that must happen to the object. A shader can only be used when it is attached to a material.

For writing shaders, there are two common scripting languages: High-Level Shading Language (HLSL) and OpenGL Shading language (GLSL).

A material is something that can be applied to an object, or mesh, to define the visual look of that object. A material is used to set the specific value of parameters that are made available from a shader. Examples of these parameters include colors, textures, and number values.

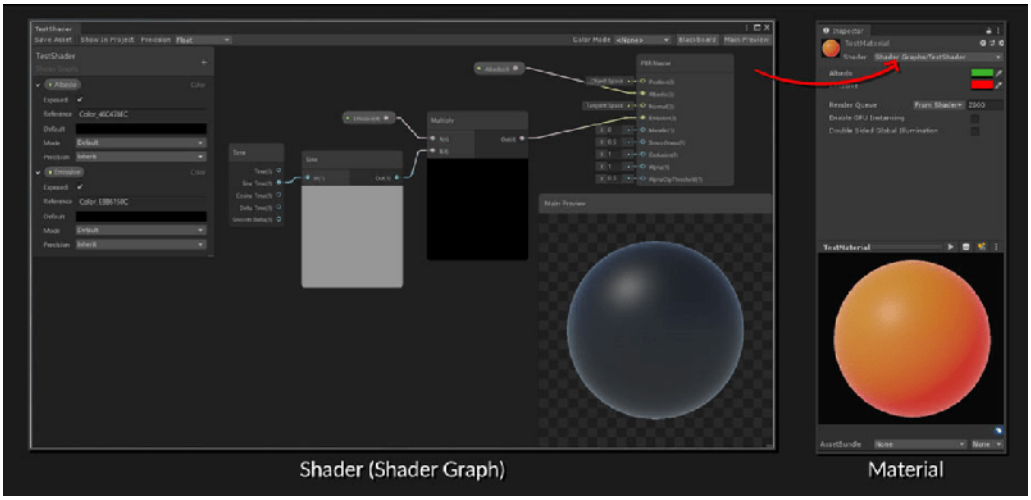
The following screenshot shows an HLSL shader applied to a material in Unity:

Figure 2-1: This is an image of the HLSL Unity shader.



Unity also provides shader graphs, which let you build your shaders visually instead of hand-writing code. The following screenshot shows how you can build a shader graph by creating and connecting nodes in a graph network, then apply that shader graph to a material:

Figure 2-2: This is an image of the Unity shader graph.



3. Use optimized shaders for mobile platforms

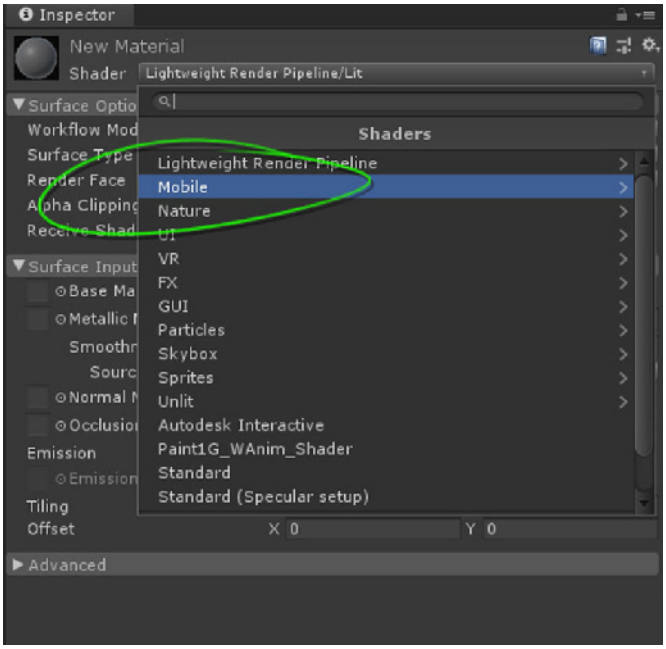
Unity provides a collection of shaders that are optimized for mobile platforms.

These shaders are located in the Mobile category within Unity.

While these optimized shaders contain simplified features, they have a better performance on mobile-based platforms. However, there are fewer features available, compared to standard, non-mobile, shaders. For example, these optimized shaders do not provide color tinting.

The following screenshot shows where you can locate the mobile-dedicated shaders, in the Mobile menu:

Figure 3-1: This is an image of the mobile menu.



Only use features that are needed. Unity creates an optimized runtime shader version. This version removes unused features from the shader that is chosen and configured in a material. This reduction in shader complexity helps with performance on mobile platforms.

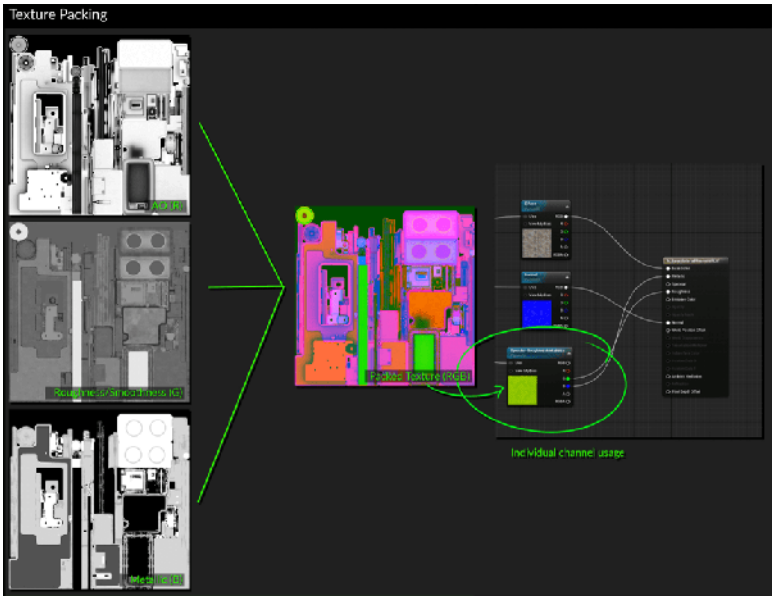
4. Optimize your textures

Try to use as few textures as possible on mobile platforms. This is because using more textures results in more texture fetches. More texture fetches mean that more bandwidth is used, affecting the battery life of the device.

Application size increases as more textures are saved in memory. Instead of using individual textures for the roughness texture and the metallic texture, you can pack these separate textures into the channels of a single texture. This technique is known as texture packing and it helps you to reduce the number of textures that are used.

The following image shows an example of how you can pack three separate textures into a single channel to save on bandwidth:

Figure 4-1: This is an image of three separate textures on a single channel.



You can also use a number instead of a texture for some parameters, for example metallic, roughness, or smoothness. If possible, you should try using an number instead of a texture and observe whether the visual quality is affected. Using a numerical value further reduces the number of textures that are used.

Note

In Unity, a value must be added within the shader.

Using an unlit shader also reduces the number of textures that are used. This is because light does not affect the material. This means that the material does not need roughness or metallic textures.

5. Comparing unlit shaders to lit shaders

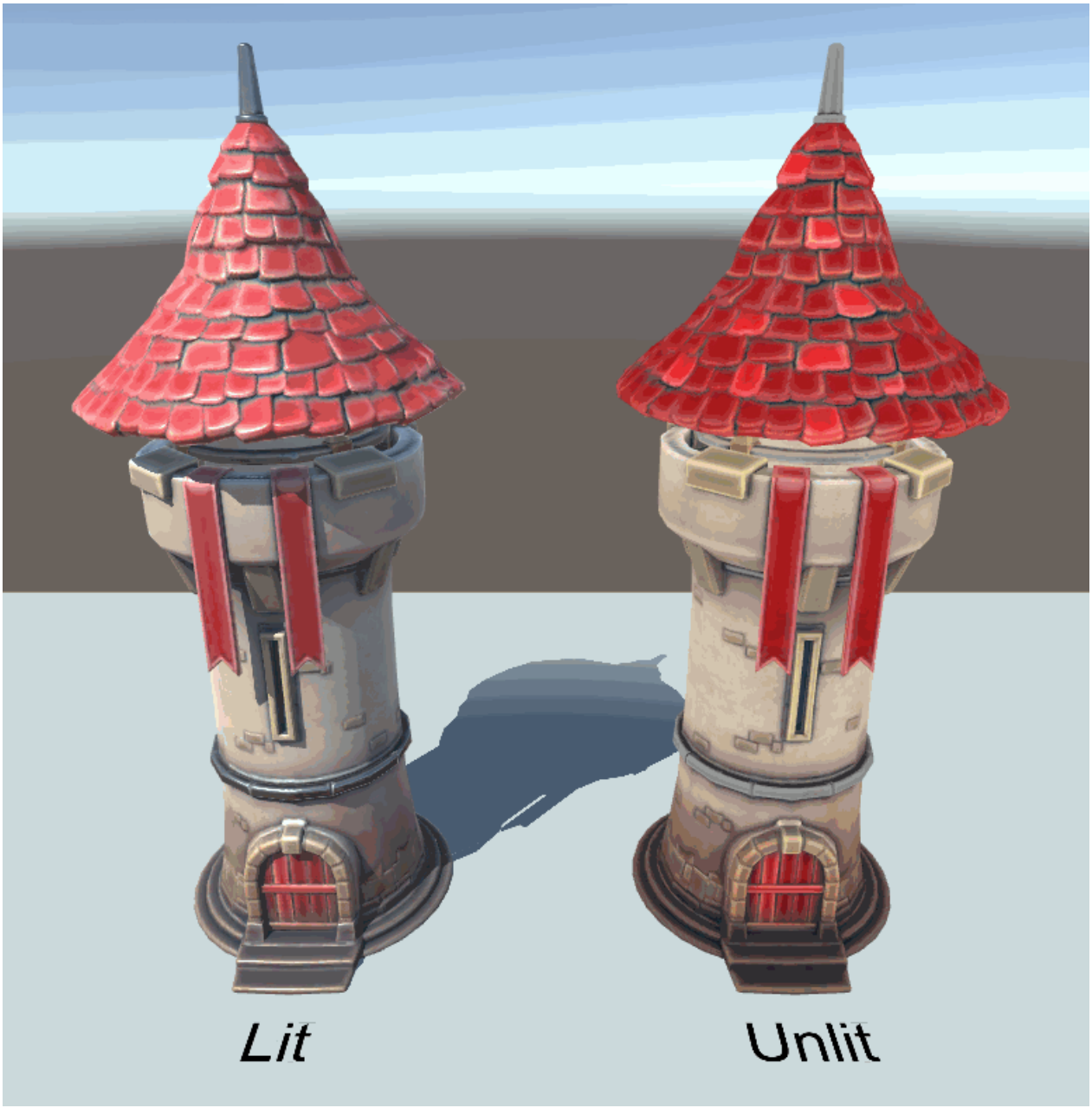
When creating a shader, you can decide how the material reacts to light. Most shaders that are used in mobile games are classified as either lit or unlit.

Unlit shaders are the fastest and cheapest shading models. Use an unlit shader if you are targeting a lower-end device. Several key points to consider include:

- Lighting does not affect an unlit shading model. This means that many calculations, for example specularity calculations, are not needed. The result is either cheaper or faster rendering.
- Using a stylized art direction that resembles a cartoon works well with unlit shading. This art style is worth considering when you are making games for mobile platforms. Lit shaders use more processing power than unlit shaders. However:
- Light affects a lit shader and it enables the surface to have specularity.
- This is probably the shading model that is most used in mobile games today.

The following image shows a comparison between a lit and an unlit object:

Figure 5-1: This gif shows the difference between a lit an unlit object.



The same tower mesh and texture set have been applied, but are using different shaders. Lights do not affect unlit shaders, so they need less computation. This results in a better gaming performance, especially on less-powerful devices.

6. Using transparencies

Be careful when using transparencies. Use an opaque material whenever possible. On mobile platforms, avoid using transparency unless it is necessary.

Rendering an object with transparency always uses more GPU resources than rendering an opaque object. Using many transparent objects can affect performance on mobile platforms. Performance can be a particular problem when transparent objects are rendered on top of one another multiple times.

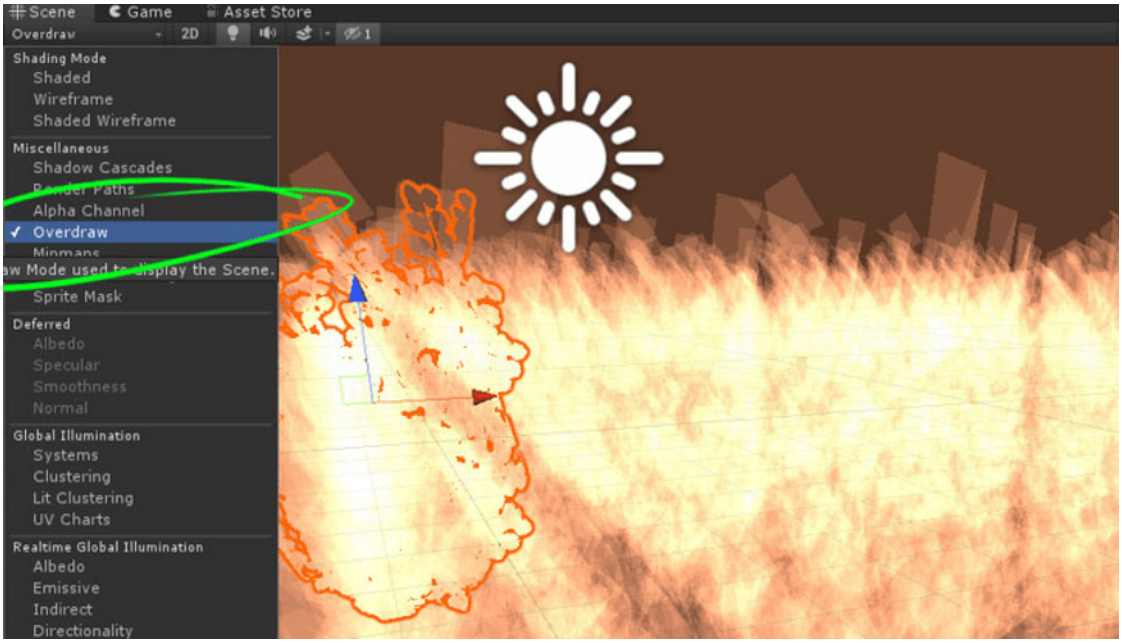
When the same pixel is drawn on-screen multiple times it is known as overdraw. Overdraw is a problem because the more layers of transparency that you have, the more expensive the rendering becomes. For mobile platforms, overdraw can severely affect performance.

The following screenshot shows how the blue lights could have had a transparency effect, but they still look good with an opaque material:

Figure 6-1: This is an image of blue lights and transparency effect.

When building levels, try to keep overdraw to a minimum. Unity lets you measure and visualize the amount of overdraw that is in a scene. This mode is highlighted in the following screenshot:

Figure 6-2: This is a screenshot of overdraw.



7. Profile and compare transparency implementations

There are several different ways to implement transparency in a shader, each with their own benefits and drawbacks.

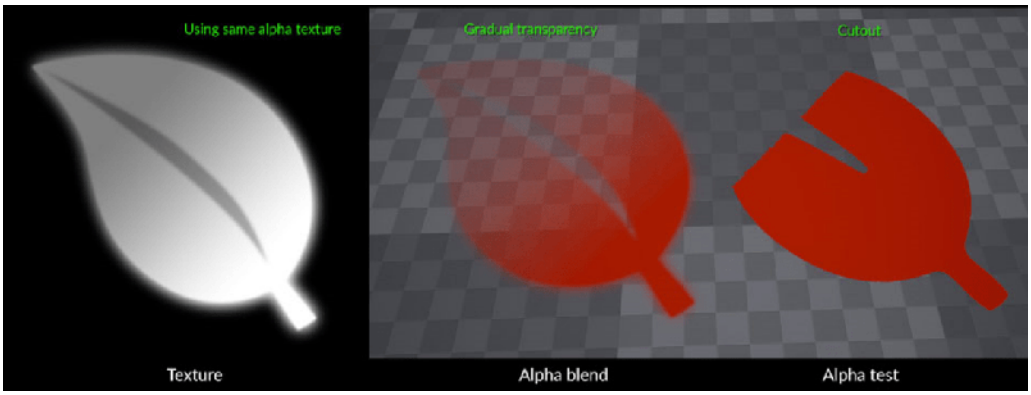
The most common used transparency implementations are alpha test and alpha blend:

- Alpha test, or cutout The alpha test implementation makes the object material look either 100% opaque, or 100% transparent. You can set the threshold of the cutout for the mask. Unity calls this type of transparency cutout.
- Alpha blend

Visually, alpha blend allows the material to have a range of transparency, making an object look partially transparent, instead of just opaque or completely transparent. Unity calls this blend mode transparent.

Alpha blend allows partial transparency while alpha test results in a sharp cutout. The following image shows a comparison of alpha blend and alpha test:

Figure 7-1: This is an image of the alpha test.



Deciding whether to use alpha blend or alpha test

The performance of alpha blend and alpha test depend on your use case. To determine which transparency implementation to use, Arm recommends that you always profile the performance difference between the two. On a mobile platform, you can use Arm Streamline to collect the performance data of a device.

Alpha test

If you want to use alpha test in your game, then here are some tips to remember:

- Avoid the alpha test, or cutout, unless it is required.
- Alpha test means that a material is either 100% opaque, or 100% transparent.

- Using alpha test disables some optimization features within the GPU. Arm therefore recommends that you carefully examine the results of alpha test on the mobile platforms that you are targeting. Also, do not forget to profile and compare it against alpha blend for any differences in performance.

Alpha blend

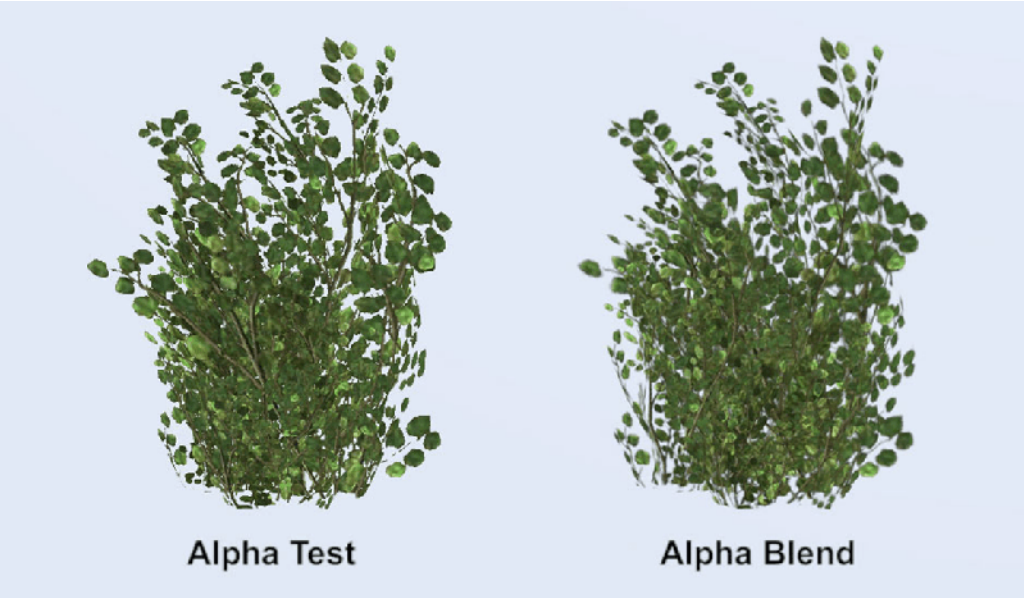
If you want to use alpha blend in your game, here are some tips to remember:

- For mobile platforms, Unity recommends using alpha blend instead of alpha test. In practice, you should profile and compare the performance between the alpha test and alpha blend. This is because the performance of alpha blend is content dependent and therefore needs measuring.
- In general, avoid using transparencies with alpha blend on mobile platforms.
- If alpha blend is needed, try to make the coverage of the blend area small.

Using alpha test transparency on foliage

In a static view of foliage, alpha blending can look better because of the soft edges. This is compared with the sharp cut in alpha test, as you can see in the following image:

Figure 7-2: This is an image of an alpha blend test.



However, when in motion, the use of alpha blend looks wrong and the leaves do not render in the correct order. Alpha test handles the transparency, and the order of the leaves, much better. However, the edges are harsher, or aliased, when compared with alpha blend.

Usually, the alpha test visual quality is acceptable, because the aliased edges are not as obvious in motion. But the optical illusion is broken when the leaves and branches jump back and forward from the use of alpha blend.

8. Additional material and shader best practices

In this section of the guide, we will describe some best practice tips. You can use these tips during the development of your game.

Keep the shader simple

In a situation where overdraw is unavoidable, make the shader as simple as possible. Keep these principles in mind:

- Use the simplest shader possible, like unlit, and avoid using unnecessary features.
- Unity has a shader that was designed specifically for particles, which can be a good implementation to start with.
- Try to minimize overdraw. Reducing the number and size of particles can help with this.

Profile shader complexity

Adding more texture samplers, transparency, and other features makes a shader more complex and can affect the rendering. We recommend that you profile your shaders often.

Arm provides tools to perform profiling, for example Mali Offline Shader Compiler and Streamline. However, these tools require a higher level of graphics knowledge to understand more about how the GPU works. If you are editing shaders, then this is an area worth studying more deeply.

Do as many operations as possible in a vertex shader

It is common in projects to achieve a specific look through the combination of a vertex shader and a pixel, or fragment, shader. Vertex shaders work on every vertex, and pixel shaders run on every pixel.

Usually, there are more pixels that are being rendered than there are vertices on screen. This means that the pixel shader runs more often than the vertex shader. We recommend that you move computation from the pixel shader to the vertex shader whenever possible.

Moving operations to a vertex shader usually means moving the processed data onto the pixel shader, through varying variables. So, even though moving operations to a vertex shader is generally a good idea, you must pay attention to the tiler in case it now becomes the limiting factor in performance. As usual, after working on optimizations, you must do further profiling.

Avoid using complicated math operations where possible

You use mathematical operations within the shader to customize its look and behavior.

These math operations are not equal in terms of performance cost. Therefore, you must pay attention to their usage. Some of the more complicated operations include sin, pow, cos, divide, and noise.

Basic operations, like additions and multiplications, are faster to process, so try to keep the number of slower math operations as small as possible. You should minimize the amount of complicated math that you use must be kept lower on older devices, for example devices that use GLES 2.0.

Always do performance profiling

To understand where the real bottlenecks are occurring in your application, profile the performance. Profiling is also recommended to compare the before and after effect of any optimization that you use.

9. Check your knowledge

The following questions will help you test your knowledge.

Why are shader graphs useful?

Shader graphs let you build your shaders visually instead of hand-writing code, creating and connecting nodes in a graphical environment within Unity.

Texture packing lets you bundle multiple individual textures into the different channels of a single texture. What benefits does texture packing provide?

Texture packing reduces application size, and reduces the number of texture fetches. The reduced size, and the reduced number of fetches both result in lower bandwidth usage and battery power consumption.

When targeting a lower-end device, should you use a lit shader or an unlit shader?

An unlit shader if you are targeting a lower-end device. Unlit shaders are the fastest and cheapest shading models because they do not perform lighting calculations.

Which are the two most common transparency implementations, and which is usually preferred for mobile platforms?

The two most commonly used transparency implementations are alpha test and alpha blend. For mobile platforms, Unity recommends using alpha blend over alpha test. In practice, you should profile and compare the performance between alpha test and alpha blend.

10. Related information

Here are some resources related to material in this guide:

- Arm Streamline
- Mali Offline Shader Compiler
- OpenGL Shading Language
- OpenGL Software Development Kit Varying Variables
- Programming guide for HLSL
- Unity Manual: Meshes, Materials, Shaders and Textures
- Unity Manual: Practical guide to optimization for mobiles
- Unity Manual: Scene view control bar (used to display overdraw silhouettes)

11. Next steps

This guide introduces concepts relating to different material and shader optimizations that can help your games to run more efficiently 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 Best Practices: Textures