

Learn. Create. Master.

First Edition

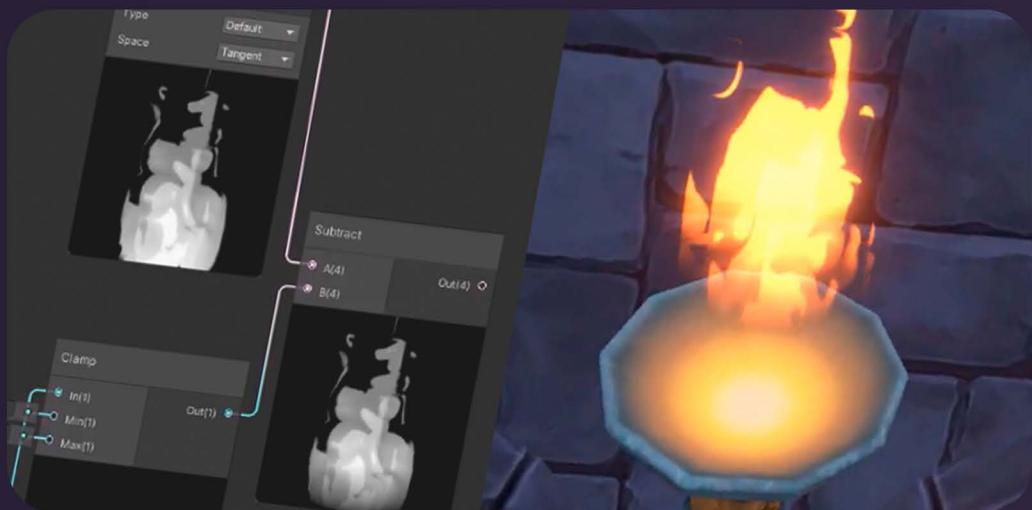
The Unity Shaders Bible.

The definitive book to learn shader in Unity.

Shaders

GameDev

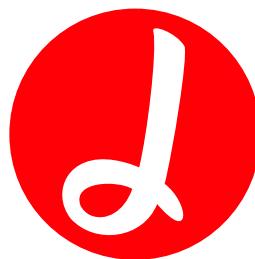
MadeWithUnity



Fabrizio Espíndola.



jettelly



The Unity Shaders Bible.

A linear explanation of shaders from beginner to advanced. Improve your game graphics with Unity and become a professional technical artist.

(First Edition)

Fabrizio Espíndola.

Game developer & Technical artist.

The Unity Shaders Bible, version 0.1.5b.
Jettelly © All rights reserved. www.jettelly.com
DDI 2021-A-11866
ISBN 979-883-3189-84-9

Credits.

Author.

Fabrizio Espíndola.

Design.

Pablo Yeber.

Technical Revision.

Daniel Santalla.

Translation, grammar, and spelling.

Martin Clarke.

About the author.

Fabrizio Espíndola is a Chilean video game developer, specialised in Computer Graphics. He has dedicated much of his career to developing visual effects and technical art, his projects include Star Wars - Galactic Defence, Dungeons and Dragons - Arena of War, Timenauts, Frozen 2, and Nom Noms. He is currently developing some independent titles together with his team at Jettelly.

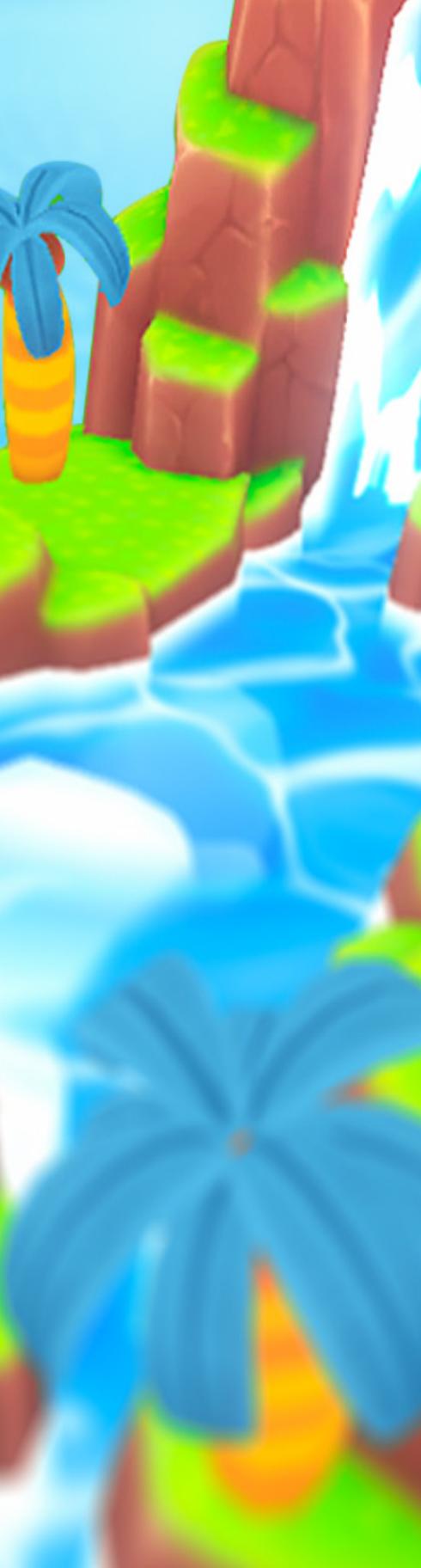
His great passion for video games was born in 1994 after Donkey Kong Country (Super Nintendo) appeared, a game that awoke a deep interest for him in the tools and techniques associated with this technology. To date, he has more than ten years of experience in the industry, and this book contains a part of the knowledge he has acquired during this time.

Years ago, while at university, his professor Freddy Gacitúa once told him:

*A person becomes a professional when
he or she contributes to others.*

These words were very significant for him in his career development and generated his need to bestow his knowledge on the international community of independent video game developers.

Jettelly was officially inaugurated on March 3, 2018, by Pablo Yeber and Fabrizio Espíndola. Together and committed, they have developed different projects among which the Unity Shaders Bible is one of the most important due to its intellectual nature.



Content.

Preface.	10
Chapter I Introduction to the shader programming language.	
1. Initial observations.	15
1.0.1. Properties of a polygonal object.	15
1.0.2. Vertices.	17
1.0.3. Normals.	18
1.0.4. Tangents.	18
1.0.5. UV Coordinates.	19
1.0.6. Vertex Color.	20
1.0.7. Render Pipeline Architecture.	20
1.0.8. Application stage.	22
1.0.9. Geometry processing phase.	22
1.1.0. Rasterization stage.	24
1.1.1. Pixel processing stage.	25
1.1.2. Types of Render Pipeline.	25
1.1.3. Forward Rendering.	27
1.1.4. Deferred Shading.	29
1.1.5. What Render Pipeline should I use?	29
1.1.6. Matrices and coordinate systems.	30
2. Shaders in Unity.	35
2.0.1. What is a shader?	35
2.0.2. Introduction to the programming language.	36
2.0.3. Shader types.	38
2.0.4. Standard Surface Shader.	39
2.0.5. Unlit Shader.	39
2.0.6. Image Effect Shader.	39
2.0.7. Compute Shader.	40
2.0.8. Ray tracing Shader.	40
3. Properties, commands and functions.	41
3.0.1. Structure of a Vertex-Fragment Shader.	41
3.0.2. ShaderLab Shader.	45



3.0.3. ShaderLab Properties.	46
3.0.4. Number and slider properties.	48
3.0.5. Color and vector properties.	49
3.0.6. Texture properties.	49
3.0.7. Material Property Drawer.	52
3.0.8. MPD Toggle.	53
3.0.9. MPD KeywordEnum.	55
3.1.0. MPD Enum.	57
3.1.1. MPD PowerSlider and IntRange.	59
3.1.2. MPD Space and Header.	60
3.1.3. ShaderLab SubShader.	61
3.1.4. SubShader Tags.	63
3.1.5. Queue Tag.	64
3.1.6. Render Type Tag.	67
3.1.7. SubShader Blending.	72
3.1.8. SubShader AlphaToMask.	77
3.1.9. SubShader ColorMask.	78
3.2.0. SubShader Culling and Depth Testing.	79
3.2.1. ShaderLab Cull.	82
3.2.2. ShaderLab ZWrite.	84
3.2.3. ShaderLab ZTest.	85
3.2.4. ShaderLab Stencil.	88
3.2.5. ShaderLab Pass.	95
3.2.6. CGPROGRAM / ENDCG.	96
3.2.7. Data types.	98
3.2.8. Cg / HLSL Pragmas.	103
3.2.9. Cg / HLSL Include.	104
3.3.0. Cg / HLSL Vertex Input & Vertex Output.	105
3.3.1. Cg / HLSL Variables and Connection Vectors.	109
3.3.2. Cg / HLSL Vertex Shader Stage.	110
3.3.3. Cg / HLSL Fragment Shader Stage.	113
3.3.4. ShaderLab Fallback.	114
4. Implementation and other concepts.	116
4.0.1. Analogy between a shader and a material.	116
4.0.2. Your first shader in Cg or HLSL.	116
4.0.3. Adding transparency in Cg or HLSL.	118
4.0.4. Structure of a function in HLSL.	119
4.0.5. Debugging a shader.	123



4.0.6. Adding URP compatibility.	126
4.0.7. Intrinsic functions.	131
4.0.8. Abs function.	131
4.0.9. Ceil function.	136
4.1.0. Clamp function.	141
4.1.1. Sin and Cos functions.	146
4.1.2. Tan function.	151
4.1.3. Exp, Exp2 y Pow functions.	155
4.1.4. Floor function.	157
4.1.5. Step and Smoothstep functions.	161
4.1.6. Length function.	165
4.1.7. Frac function.	168
4.1.8. Lerp function.	173
4.1.9. Min and Max functions.	176
4.2.0. Timing and animation.	177

Chapter II | Lighting, shadow and surfaces.

5. Introduction to the chapter.	182
5.0.1. Configuring inputs and outputs.	182
5.0.2. Vectors.	187
5.0.3. Dot Product.	189
5.0.4. Cross Product.	193
6. Surface.	195
6.0.1. Normal Maps.	195
6.0.2. DXT compression.	201
6.0.3. TBN Matrix.	206
7. Lighting.	208
7.0.1. Lighting model.	208
7.0.2. Ambient Color.	208
7.0.3. Diffuse Reflection.	211
7.0.4. Specular Reflection.	221
7.0.5. Environmental Reflection.	233
7.0.6. Fresnel Effect.	243
7.0.7. Structure of a Standard Surface.	251
7.0.8. Standard Surface Input & Output.	253

8. Shadow.	255
8.0.1. Shadow Mapping.	255
8.0.2. Shadow Caster.	256
8.0.3. Shadow Map Texture.	261
8.0.4. Shadow implementation.	266
8.0.5. Built-in RP Shadow Map optimization.	270
8.0.6. Universal RP Shadow Mapping.	274
9. Shader Graph.	281
9.0.1. Introduction to Shader Graph.	281
9.0.2. Starting in Shader Graph.	283
9.0.3. Analyzing its interface.	284
9.0.4. Your first shader in Shader Graph.	287
9.0.5. Graph Inspector.	294
9.0.6. Nodes.	296
9.0.7. Custom Functions.	298
Chapter III Compute Shader, Ray Tracing and Sphere Tracing.	
10. Advanced concepts.	303
10.0.1. Compute Shader structure.	304
10.0.2. Your first Compute Shader.	308
10.0.3. UV coordinates and texture.	321
10.0.4. Buffers.	325
11. Sphere Tracing.	336
11.0.1. Implementing functions with Sphere Tracing.	338
11.0.2. Projecting a texture.	347
11.0.3. Smooth minimum between two surfaces.	353
12. Ray Tracing.	358
12.0.1. Configuring Ray Tracing in HDRP.	359
12.0.2. Using Ray Tracing in your scene.	366
Index.	369
Special thanks.	372

One of the biggest problems that video game developers have when they start studying shaders, regardless of the rendering engine, is the lack of information for beginners on the web. Whether the reader is an independent developer or focused on AAA projects, it can be a little complicated breaking into this subject because of the technical knowledge required to develop these kinds of programs.

Despite this challenge, by being multi-platform, Unity offers a great advantage, the video game code only needs to be written once, then it can be exported to different devices, including consoles and smartphones. Likewise, once the adventure into the world of shaders starts, the code is written once and then the software takes care of its compilation for the different platforms (OpenGL, Metal, Vulkan, Direct3D, GLES 20, GLES 3x).

The Unity Shaders Bible has been created to solve most of the problems met when starting in this world. You will begin by reviewing the structure of a shader in the Cg and HLSL languages and then get to know its properties, commands, functions, and syntax.

Did you know that in Unity there are three types of Render Pipeline, each with its own qualities? Throughout the book, you will analyse them, verifying how Unity processes the graphics to project your video games onto the computer screen.

I. Topics we will see in this book.

The book is divided into three chapters, in which points are addressed as they are required linearly. All the code seen in this book has been tested using the Visual Studio Code editor and checked in Unity for the different types of Render Pipeline.

Chapter I: Introduction to the shader programming language.

This chapter looks at the base knowledge needed before starting, such as shader structure in ShaderLab language, the analogy between the properties and connection variables, SubShader and commands (ColorMask, Stencil, Blending, etc.), Passes and structure of Cg and HLSL languages, function structure, Vertex Input analysis, Vertex Output analysis, the analogy between a semantic and a primitive, Vertex Shader Stage structure, Fragment Shader Stage structure, matrices and more. This chapter is the starting point to understand fundamental concepts about how a shader works in Unity.

Chapter II: Lighting, shadow, and surfaces.

This section addresses highly relevant issues, such as: Normal Maps and their implementation, reflection maps, lighting and shadow analysis, a basic lighting model, surface analysis, mathematical functions, specularity, and ambient light. Furthermore, a review of Shader Graph and its structure, HLSL functions, nodes, properties and more. In this chapter, you will make your video game look professional with simple lighting concepts.

Chapter III: Compute Shader, Ray Tracing, and Sphere Tracing.

Here you will put into practice advanced concepts, such as: the structure of a Compute Shader, buffer variables, Kernel, Sphere Tracing implementation, implicit surfaces, shapes and algorithms, an introduction to Ray Tracing, configurations, and high-quality rendering. Your studies will conclude in this chapter by investigating GPGPU programming (general-purpose GPU), using .compute type shaders, trying the Sphere Tracing technique and using Direct Ray Tracing (DXT) in HDRP.

II. Recommendations.

It is essential to work with a code editor with **IntelliSense** in graphics language programming, specifically in Cg or HLSL. Unity has **Visual Studio Code**, which is a more compact version of Visual Studio Community. This editor contains some extensions that add IntelliSense to C#, ShaderLab, and HLSL.

For those using the Visual Studio Code editor, the installation of the following extensions is recommended: **C# for Visual Studio Code** (Microsoft), **Shader language support for VS Code** (Slevesque), **ShaderLab VS Code** (Amlovey), **Unity Code Snippets** (Kleber Silva).

III. Who this book is for.

This book has been written for Unity developers looking to improve their graphics skills or create professional-looking effects. It is assumed that the reader already knows, understands, and has access to the Unity interface; therefore, it will not be gone into in detail.

Having previous knowledge of C# or C++ would be a great asset in understanding the content presented in this book; However, it is not an exclusive requirement.

It is fundamental to have some basic knowledge of arithmetic, algebra, and trigonometry to understand more advanced concepts. All the same, mathematical operations and functions to fully understand what you are developing will be reviewed.

IV. Glossary.

Given its nature, there will be phrases and words that are distinguished from the rest in this book, they are easy to identify because they are highlighted to emphasise an explanation or concept. Likewise, there are blocks of code in HLSL to illustrate some functions.

Some words are shown in bold, which is also used to emphasise lines of code. Other technical definitions start with a capital letter (e.g., Vertex), while those of a constant nature will be presented entirely in the same style (e.g., RGBA).

In the function definitions arguments are shown with the acronym RG (e.g., N_{RG}) and space coordinates shown with AX (e.g., Y_{AX}). Also, there are blocks of code that include three periods (...), these refer to variables or functions included by default within the code.

V. Errata.

When writing this book, every precaution was taken to ensure the fidelity of its content. Even so, please remember that we are human beings, and it is very likely that some points may not have been explained well or may have incurred mistakes in the spelling/grammar correction.

If you find a conceptual error, code or otherwise, we would appreciate it if you could inform us by email at contact@jettelly.com indicating **USB Errata** in the subject field; in this way, you will be helping other readers reduce their level of frustration, by improvements being made in the following releases.

Furthermore, if you feel that this book is missing some interesting sections, you are welcome to email us, and we will include that information in future releases.

VI. Assets and donations.

This book has exclusive assets to reinforce its content that are included in the download. These have been developed using Unity 2020.3.21f1 and analyzed in both Built-in and Scriptable Render Pipeline. → learn.jettelly.com/usb-resources

All the work you see in Jettelly has been developed by its own members, this includes drawings, designs, videos, audio, tutorials, and everything you see with the brand.

Jettelly is an independent video game development studio, your support is critical to us. If you want to support us financially, you can do so directly through our PayPal account.

→ paypal.com/paypalme/jettelly

VII. Piracy.

Before copying, reproducing, or supplying this material without our consent, remember that Jettelly is an independent and self-financed studio. Any illegal practice could affect our integrity as a developer team.

This book is patented under copyright, and we will protect our licenses seriously. If you find this book on a platform other than Jettelly or detect an illegal copy, please contact us at contact@jettelly.com (attaching the link if necessary). In this way, we can find a solution. Thank you in advance.



Chapter I

Introduction to the shader programming language.

Initial Observations.

Years ago, when I was just starting to study Unity shaders, it was very difficult to understand much of the content I found in the books for various reasons. I still remember the day I was trying to understand the semantic function POSITION[n]. However, when I did finally manage to find its definition, I found the following statement:

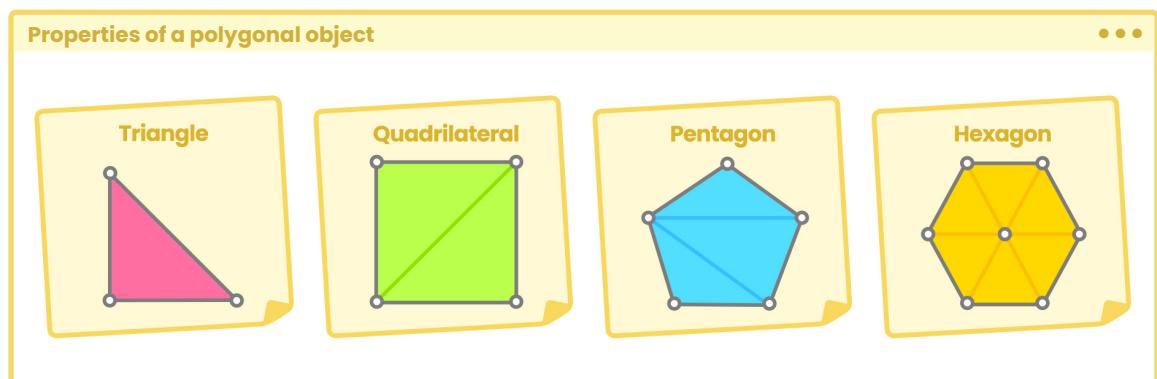
Vertex position in object-space.

At that moment, I asked myself, what is Vertex position in Object-Space? Then I understood that there was previous information that I had to know before starting to read about this subject. In my experience, I have been able to identify at least four fundamental areas that facilitate the understanding of shaders and their structure, such as:

- Properties of a polygonal object.
- Structure of a Render Pipeline.
- Matrices, and coordinate systems.

1.0.1. Properties of a polygonal object.

The word polygon comes from Greek πολύγωνος (polúgōnos) and is composed of poly (many) and gnow (angles). By definition, a polygon refers to a closed flat figure bounded by line segments.



(Fig. 1.0.1a)

A **primitive** is a three-dimensional geometric object formed by polygons and is used as a predefined object in different development software. Within Unity, Maya or Blender, you can find other primitives. The most common are:

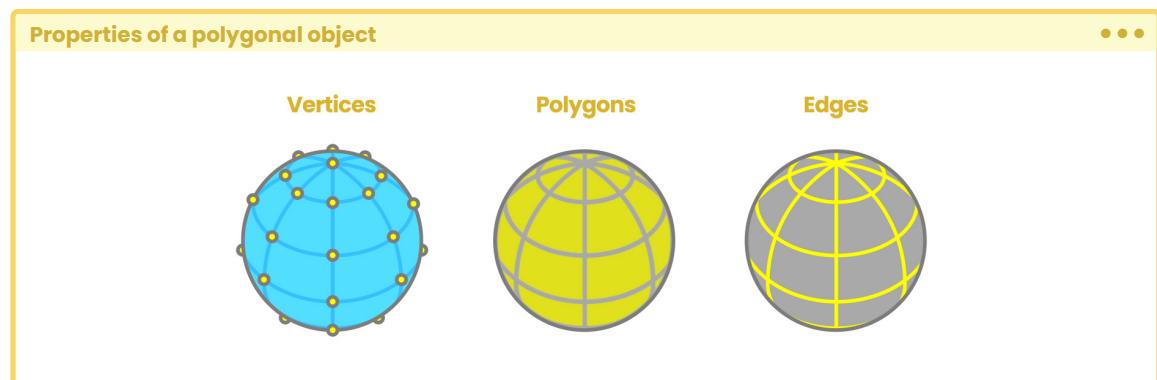
- Spheres.
- Boxes.
- Quads.
- Cylinders.
- Capsules.

All these objects are different in shape but have similar properties; all have:

- Vertices.
- Tangents.
- Normals.
- UV coordinates.
- Color.

Which are stored within a data type called **Mesh**.

All these properties can be accessed independently within a shader and kept in vectors. This is very useful because you can modify their values and thus generate interesting effects. To understand this concept better, here is a simple definition of the properties of a polygonal object.



(Fig. 1.0.1b)

1.0.2. Vertices.

The vertices of an object correspond to the set of points that define the area of a surface in either a two-dimensional or three-dimensional space. In Maya and Blender, the vertices are represented as the intersection points of an object mesh, similar to a set of atoms (molecules).

Two main things characterize these points:

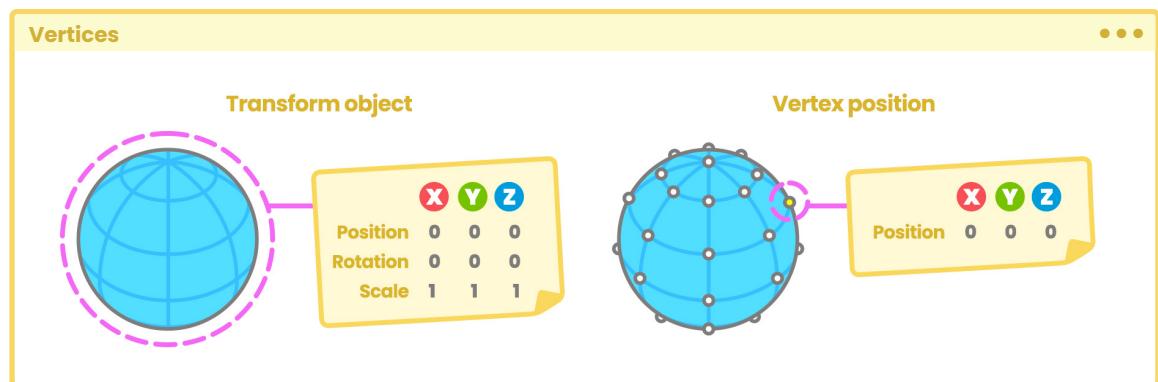
- 1 They are children of the Transform component.
- 2 They have a defined position according to the center of the total volume of the object.

What does this mean? Maya 3D has two default nodes associated to an object. These are known as **Transform** and **Shape**.

The Transform node, as in Unity, defines the position, rotation, and scale of an object in relation to the object's pivot. The Shape node, child of the Transform node, contains the geometry attributes, that is, the position of the object's vertices in relation to its volume.

This means the vertex set of an object can be moved, rotated, or scaled, and at the same time, the position of a specific vertex changed.

The POSITION[n] semantics in HLSL specifically gives access to the position of the vertices in relation to their volume, that is, to the configuration exported by the Shape node from Maya.



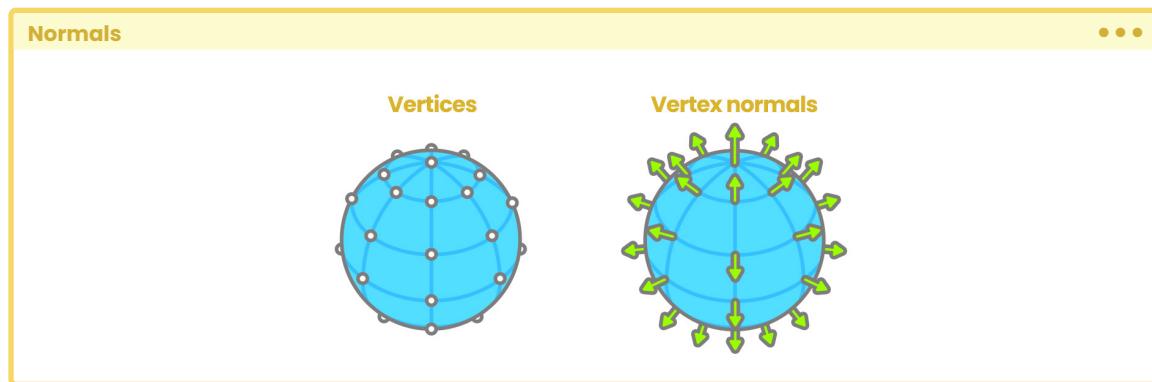
(Fig. 1.0.2a)

1.0.3. Normals.

Imagine that a friend is asked to draw on the front face of a blank sheet of paper. How could you determine which is the front face of a blank sheet if both sides are equal? This is why **Normals** exist.

A **Normal** corresponds to a perpendicular vector on the surface of a polygon which is used to determine the direction or orientation of a face or vertex.

In Maya, the object Normals are visualized by selecting the property Vertex Normals. This allows us to see where a vertex points in space and determines the hardness level between the different faces of an object.



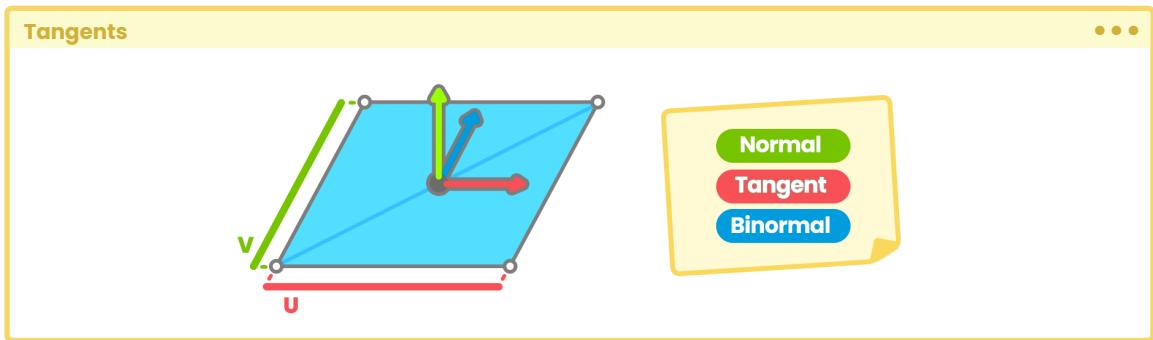
(Fig. 1.0.3a. Graphical representation of the Normals for each vertex)

1.0.4. Tangents.

According to official Unity documentation:

A tangent is a vector of a unit of length that follows the mesh surface along the direction of the horizontal texture.

What does this mean? Look at Figure 1.0.4a to understand its nature. The **Tangent** is a normalized vector that follows the U coordinate orientation of the UV on each geometry face. Its main function is to generate a space called Tangent-Space.



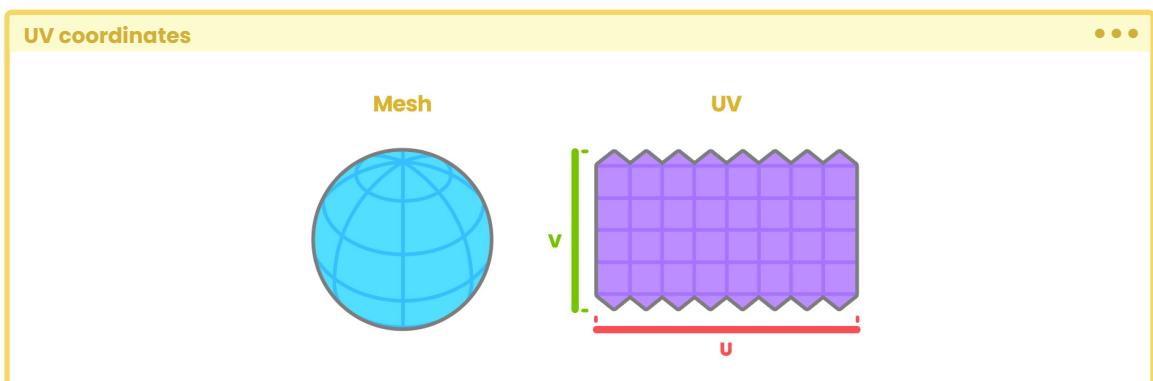
(Fig. 1.0.4a. By default, Binormals cannot be accessed in a shader. Instead, they need to be calculated in relation to the Normals and Tangents)

This property is reviewed in detail later, in Chapter II, section 6.0.1, as well as the **Binormals** for the implementation of a Normal Map over an object.

1.0.5. UV coordinates.

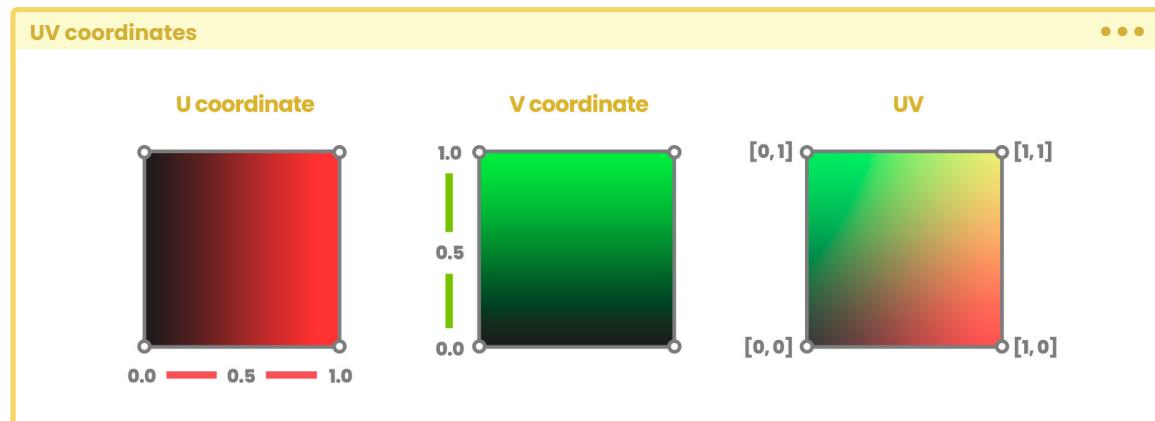
Everyone has changed the skin of their favorite character for a better one. UV coordinates are directly related to this concept since they allow you to position a two-dimensional texture on the surface of a three-dimensional object. These coordinates act as reference points, which control the corresponding texels in the texture map to each vertex in the mesh.

The process of positioning vertices over UV coordinates is called **UV mapping** and is a process by which UV, that appears as a flattened, two-dimensional representation of the object's mesh, is created, edited, and organized. You can access this property within your shader, either to position a texture on your 3D model or to save information in it.



(Fig. 1.0.5a. Vertices can be arranged in different ways within a UV map)

The area of the UV coordinates is equal to a range between 0.0f and 1.0f, where the first corresponds to the starting point and the second is the endpoint.



(Fig. 1.0.5b. Graphic reference to the UV coordinates in a cartesian plane)

1.0.6. Vertex Color.

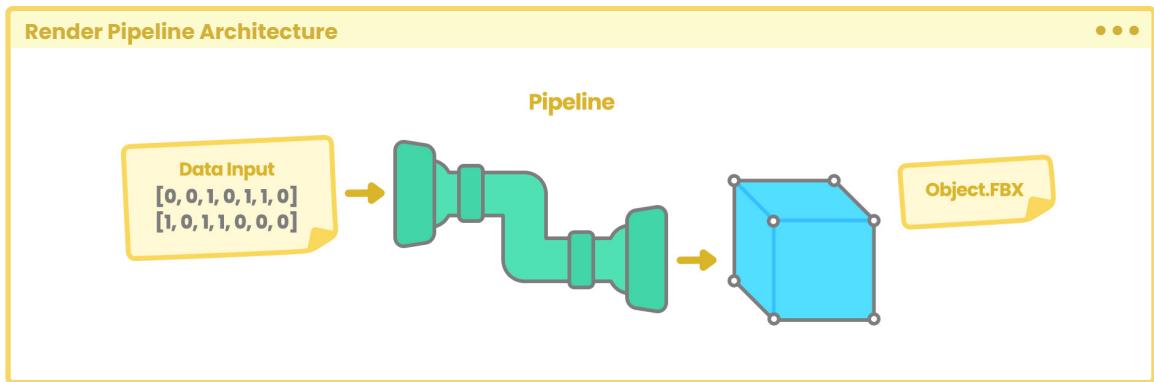
When you export an object from 3D software, it assigns a color to the object to be affected, either by lighting or multiplication of another color. Such color is known as **Vertex Color** and is white by default, with the values 1.0f in RGBA channels.

1.0.7. Render Pipeline Architecture.

In current versions of Unity, there are three types of Render Pipeline which are: **Built-in RP**, **Universal RP** (called **Lightweight** in previous versions), and **High-Definition RP**.

It is worth asking, what is a Render Pipeline? To answer this, the first thing to understand is the pipeline concept.

A pipeline is a series of stages that perform a bigger task operation. So, what does Render Pipeline refer to? This concept could be thought of as the complete process that a polygon object must take to be rendered onto our computer screen; it is like an object traveling through Super Mario pipes until it reaches its final destination.



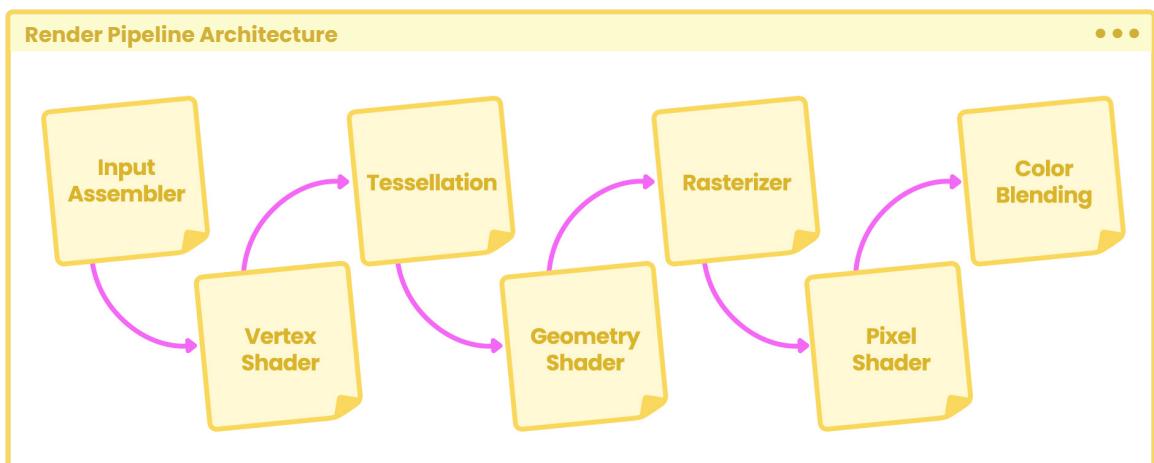
(Fig. 1.0.7a)

So, each Render Pipeline has its own characteristics, and depending on the type you are using, will affect the appearance and optimization of objects on the screen in terms of: material properties, light sources, textures, and all the functions that are occurring internally within the shader.

Now, how does this process happen? For this, you must learn about its basic architecture. Unity divides this architecture into four stages which are:

- Application stage.
- Geometry processing phase.
- Rasterization stage.
- Pixel processing stage.

Please note that this corresponds to the basic model of a Render Pipeline for real-time rendering engines. Each of the mentioned stages has threads that will now be defined.



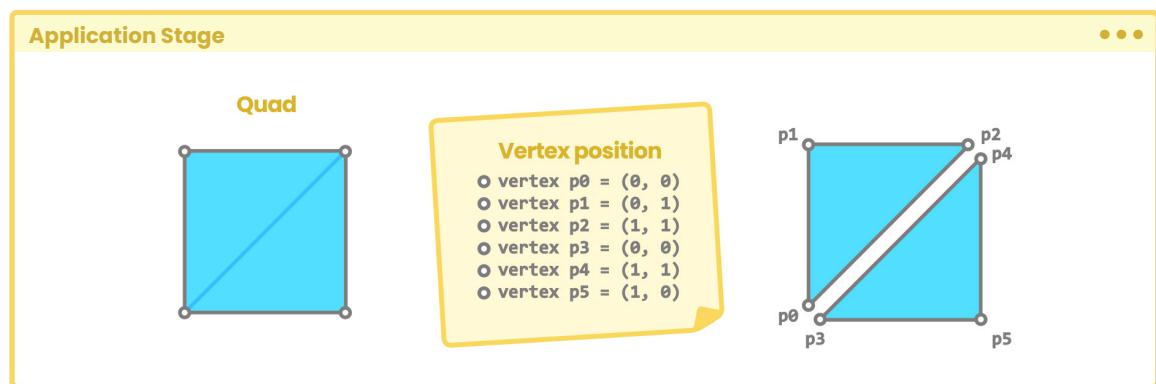
(Fig. 1.0.7b. Logic Render Pipeline)

1.0.8. Application Stage.

The application stage starts at the CPU and is responsible for various operations that occur within a scene, e.g.,

- Collision detection.
- Texture animation.
- Keyboard input.
- Mouse input, and more.

Its function is to read the stored memory data to later generate primitives (e.g., triangles, lines, vertices). At the end of the application stage, all this information is sent to the geometry processing phase to then generate the vertices' transformation through matrix multiplication.



(Fig. 1.0.8a. Local position of vertices in a Quad)

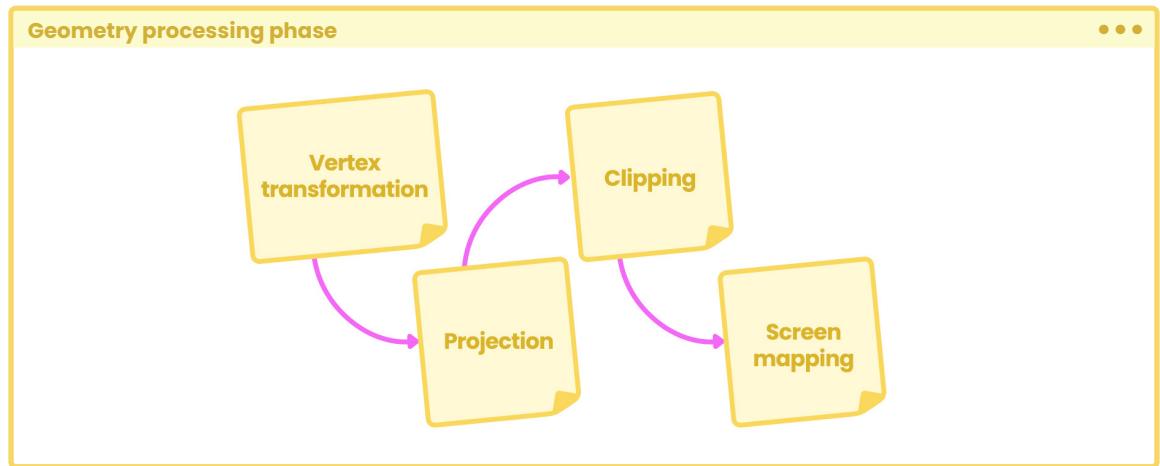
1.0.9. Geometry processing phase.

The images that you see on the computer screen are requested by the GPU for the CPU. These requests are carried out in two main steps:

- 1 The configuration of the render state, which corresponds to the set of stages from geometry processing up to pixel processing.
- 2 And then, the object being drawn on the screen.

The geometry processing phase occurs in the GPU and is responsible for the vertex processing of your object, which is divided into four subprocesses which are:

- Vertex Shading.
- Projection.
- Clipping.
- Screen mapping.



(Fig. 1.0.9a)

Once the primitives have been assembled in the application stage, the Vertex Shading, better known as the **Vertex Shader Stage**, carries out two main tasks:

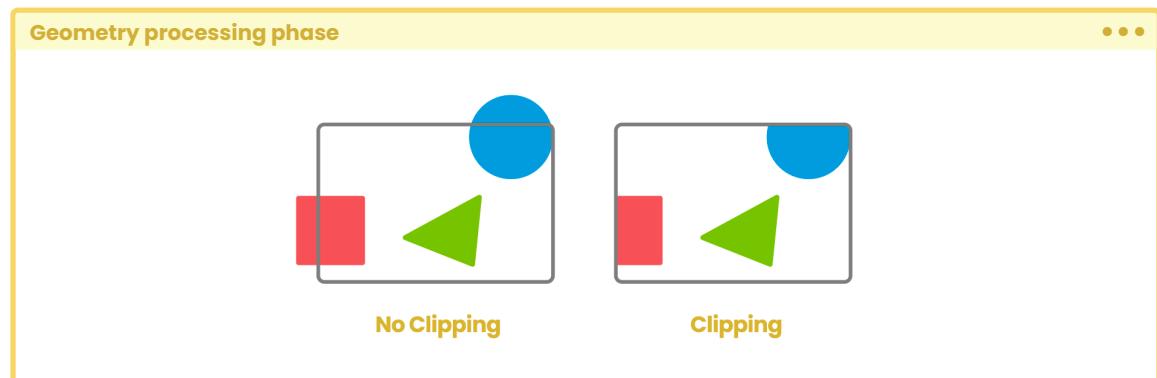
- 1 It calculates the object vertices position.
- 2 Then it transforms its position to different space coordinates so that they can be projected onto the computer screen.

Also, within this subprocess, you can select properties that you want to pass on to the following stages, for example, Normals, Tangents, UV coordinates, etc.

Projection and clipping occur in this process, which vary according to the camera properties in the scene: that is, if it is configured in perspective or orthographic (parallel). It is worth mentioning that the complete rendering process only occurs for those elements that are within the camera frustum, also known as the View-Space.

To understand this process, say there is a Sphere in the scene, where half of it is outside the frustum of the camera. Only the area of the Sphere that lies within the frustum will be projected

and subsequently clipped on the screen (clipping), while the area of the Sphere that is out of sight will be discarded in the rendering process.



(Fig. 1.0.9b)

Once the clipped objects are in the memory, they are then sent to the screen map. In this stage, the three-dimensional objects in the scene are transformed into 2D screen coordinates, also known as Screen or Window coordinates.

1.1.0. Rasterization stage.

The third stage corresponds to rasterization. At this point, the objects have 2D screen coordinates, so pixels in the projection area must be found. The process of finding all the pixels that surround an on-screen object is called **Rasterization**. This process can be seen as a synchronization point between the objects in the scene and the pixels on the screen.

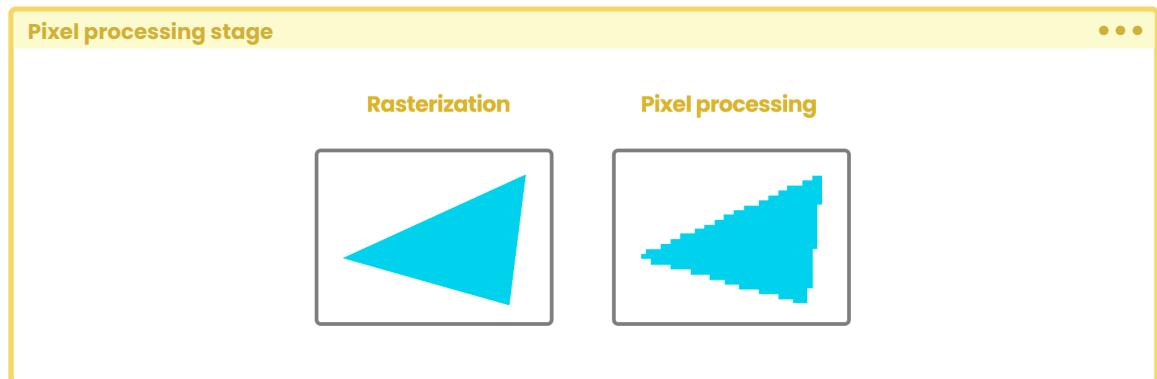
For each object, the **rasterizer** performs two processes:

- 1 Triangle Setup.
- 2 Triangle Traversal.

Triangle Setup generates the data that will be sent to **Triangle Traversal**. It includes the equations for the edges of an object on the screen. After this, Triangle Traversal lists the pixels that are covered by the area of the polygon object. In this way, it generates a group of pixels called fragments; and from this the word **Fragment Shader**, which is also used to refer to an individual pixel.

1.1.1. Pixel processing stage.

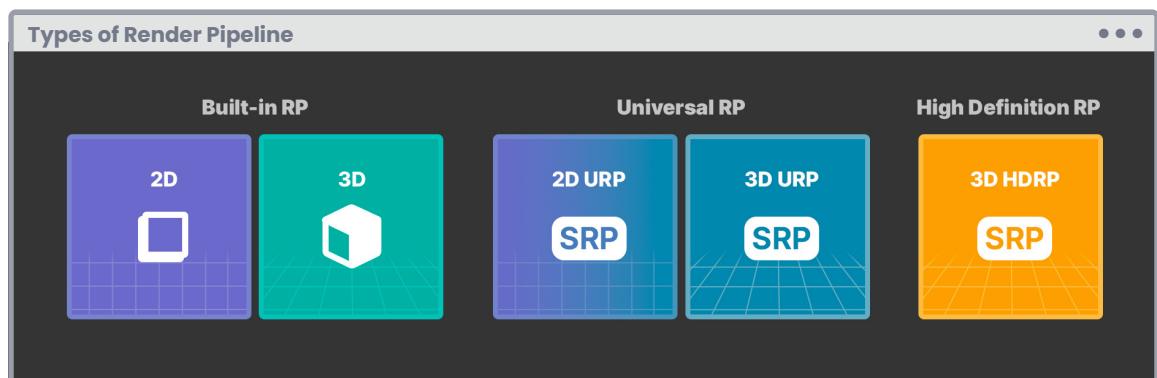
Using the interpolated values from the previous processes, this last stage starts when all the pixels are ready to be projected onto the screen. At this point, the **Fragment Shader Stage**, also known as a **Pixel Shader Stage**, begins and is responsible for the visibility of each pixel. What it does is process the final color of a pixel and then send it to the **Color Buffer**.



(Fig. 1.1.1a. The area covered by a geometry is transformed to pixels on the screen)

1.1.2. Types of Render Pipeline.

As you already know, there are three types of Render Pipeline in Unity. By default, there is the **Built-in** RP that corresponds to the oldest engine belonging to the software, in contrast, **Universal** RP and **High-Definition** RP belong to a type of Render Pipeline called **Scriptable** RP, which is more up-to-date and has been pre-optimized for better graphics performance.



(Fig. 1.1.2a. When you create a new project in Unity, you can choose between these three rendering engines. Your choice depends on the needs of the project at hand)

Regardless of the Render Pipeline , if you want to generate an image on the screen, you have to travel through the pipeline.

A pipeline can have different processing paths, known as **Render Paths**; as if the example pipeline in section 1.0.7 had more than one way to reach its destination.

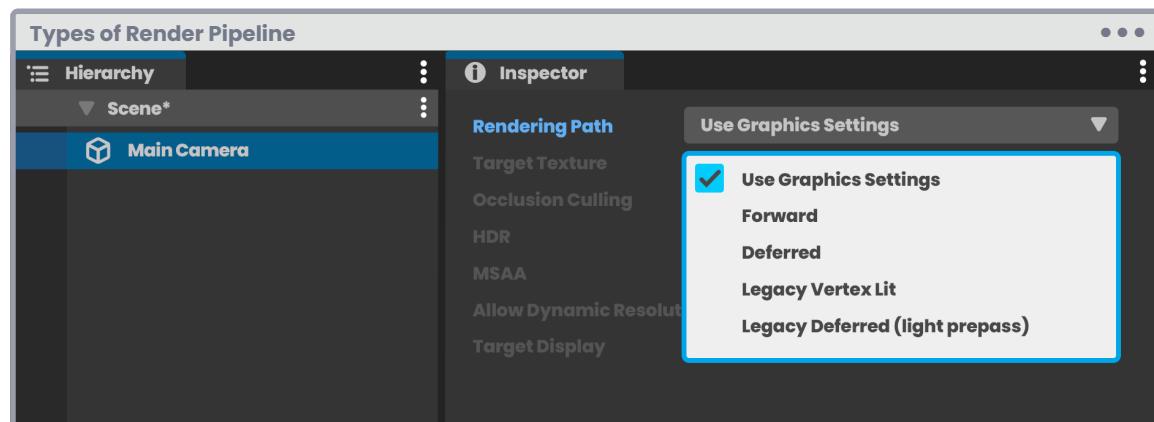
A Render Path corresponds to a series of operations related to lighting and shading objects. This allows you to graphically process an illuminated scene (e.g., a scene with directional light and a sphere).

Among them you can find:

- Forward Rendering.
- Deferred Shading.
- Legacy deferred.
- Legacy vertex lit.

Each of these has different capabilities and performance characteristics.

In Unity, the default Rendering Path corresponds to **Forward Rendering**; this is the initial path for the three types of Render Pipeline that are included in Unity (Built-in, Universal, and High-Definition). This is because it has greater graphics card compatibility and a lighting calculation limit, making it a more optimized process.



(Fig. 1.1.2b. To select a rendering path in Built-in Render Pipeline, you must go to the hierarchy, select the main camera and in the property Rendering Path you can change the configuration according to the needs of your project)

To understand this concept, imagine an object and a direct light in a scene. The interaction between them is based on the following two fundamental concepts:

- 1 Lighting characteristics.
- 2 Object material characteristics.

Such interaction is called the **lighting model**.

The basic lighting model corresponds to the sum of three different properties:

- Ambient color.
- Diffuse reflection.
- Specular reflection.

The lighting calculation is carried out within the shader and can be done by vertex or fragment. When the illumination is calculated by vertex it is called **Per-Vertex Lighting** and performed in the Vertex Shader Stage. Likewise, when it is calculated by fragment it is called **Per-Fragment** or **Per-Pixel Lighting** and is performed in the Fragment Shader Stage.

1.1.3. Forward Rendering.

Forward is the default Rendering Path and supports all typical features of a material including Normal Maps, individual pixel illumination, shadows, and more. This rendering path has two different code written passes that you can use in your shader, the first, **base pass** and the second **additional pass**.

In the base pass you can define **ForwardBase Light Mode** and in the additional pass, you can define **ForwardAdd Light Mode** for additional lighting calculations. Both are characteristic functions of a shader with lighting calculations. The base pass can process directional light Per-Pixel and will prioritise the brightest light if there are multiple directional lights in the scene. In addition, the base pass can process Light Probes, global illumination, and ambient illumination (sky light).

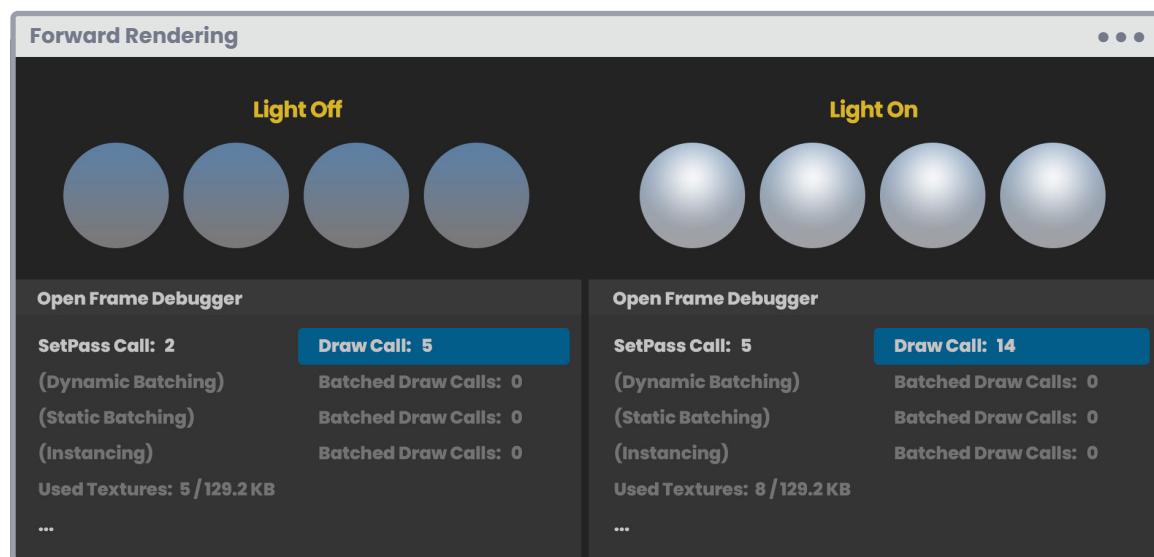
As its name says, the additional pass can process additional lights (Point Light, Spotlight, Area Light) or also shadows that affect the object. What does this mean? If there are two lights in the scene, your object will be influenced by only one of them. However, if you have defined an additional pass for this configuration, then it will be influenced by both.

A point to consider is that each illuminated pass will generate an independent **Draw Call**. What does this mean? By definition a Draw Call is a call graphic that is made in the GPU every time an element is drawn on the screen of the computer. These calls are processes that require a large amount of computation, so they need to be kept to the minimum possible, even more so if you are working on projects for mobile devices.

To understand this concept, suppose there are four spheres and one directional light in your scene. Each sphere, by its nature, generates a call to the GPU, this means that each of them will generate an independent Draw Call by default.

Likewise, the directional light influences all the spheres that are found in the scene, therefore, it will generate an additional Draw Call for each one.

This is mainly because a second pass has been included in the shader to calculate the shadow projection, therefore, four spheres, plus one-directional light will generate eight graphic calls in total.



(Fig. 1.1.3a. In the image above, you can see the increase in Draw Calls when there are light sources. The calculation includes the ambient color and the light source as the object)

Having determined the base pass, if another pass is added in the shader, then another Draw Call for each object will be added, and consequently, the graphic load will increase respectively.

1.1.4. Deferred Shading.

This rendering path ensures that there is only one lighting pass computing each light source in the scene, and only in those pixels that are affected by them, all this through the separation of the geometry and lighting. This figures as an advantage since a significant amount of light could be generated that influences different objects, thereby improving the fidelity of the final render but nominally increasing the Per-Pixel calculation on the GPU.

While Deferred Shading is superior to Forward when it comes to calculating multiple light sources, it comes with some restrictions, for example, according to official Unity documentation, Deferred Shading requires a graphics card with multiple Render Targets, Shader Model 3.0 or higher, and support for Depth render texture.

On mobile devices, this configuration works only on those that support at least OpenGL ES 3.0.

Another important consideration about this Rendering Path is that it can only be used in projects with a perspective camera. Deferred Shading does not have support for orthographic projection.

1.1.5. What Render Pipeline should I use?

In the past, there was only Built-in RP, so it was very easy to start a 2D or 3D project. However, nowadays, the project must be started according to its needs, so you may wonder, what does the project need? To answer this question, the following factors must be considered:

- 1** If a video game is being developed for a PC you can use any of the three Unity Render Pipelines available since generally, a PC has larger computing power than a mobile device or a console. Then, if the video game is aimed at a high-end device, does it need to graphically look realistic? If so, you could start in both High Definition and Built-in RP.
- 2** If the video game is wanted to be graphically in medium definition, you can use Universal RP or, as in the previous case, Built-in RP too. Now, why does Built-in RP appear as an option in both cases?

Unlike the previous ones, this Render Pipeline is much more flexible, hence, it is much more technical and does not have pre-optimization. High-Definition RP has been pre-optimized to generate high-end graphics, and Universal RP has been pre-optimized for mid-range graphics.

Another important factor when choosing the Render Pipeline is the shaders. Generally, in both High-Definition and Universal RP, shaders are created in **Shader Graph**, which is a package that brings an interface that allows the development of shaders through nodes.

This brings with it a positive and negative side. On the one hand, you can produce shaders visually through nodes without the need to write code in HLSL. However, if you want to upgrade the unity version to a higher version during production (e.g. from 2019 to 2022), it is very likely that the shaders will stop compiling because Shader Graph has independent versions and updates.

The best way to generate shaders in Unity is through HLSL, since this way you can ensure that your program compiles in the different Render Pipelines and continues to work regardless of the Unity update. This concept will be discussed later when reviewing the structure of a program in HLSL in detail.

1.1.6. Matrices and coordinate systems.

One of the concepts seen frequently in the creation of shaders is **matrices**. A matrix is a list of numeric elements that follow certain arithmetic rules and are frequently used in Computer Graphics.

In Unity the matrices represent a spatial transformation and among them are:

- UNITY_MATRIX_MVP.
- UNITY_MATRIX_MV.
- UNITY_MATRIX_V.
- UNITY_MATRIX_P.
- UNITY_MATRIX_VP.
- UNITY_MATRIX_T_MV.
- UNITY_MATRIX_IT_MV.
- unity_ObjectToWorld.
- unity_WorldToObject.

All of these correspond to four-by-four matrices (4x4), that is, each of them has four rows and four columns of numerical values.

They are conceptually represented is as follows:

```
UNITY_MATRIX
(
    Xx,     Yx,      Zx,      Tx,
    Xy,     Yy,      Zy,      Ty,
    Xz,     Yz,      Zz,      Tz,
    Xt,     Yt,      Zt,      Tw
);
```

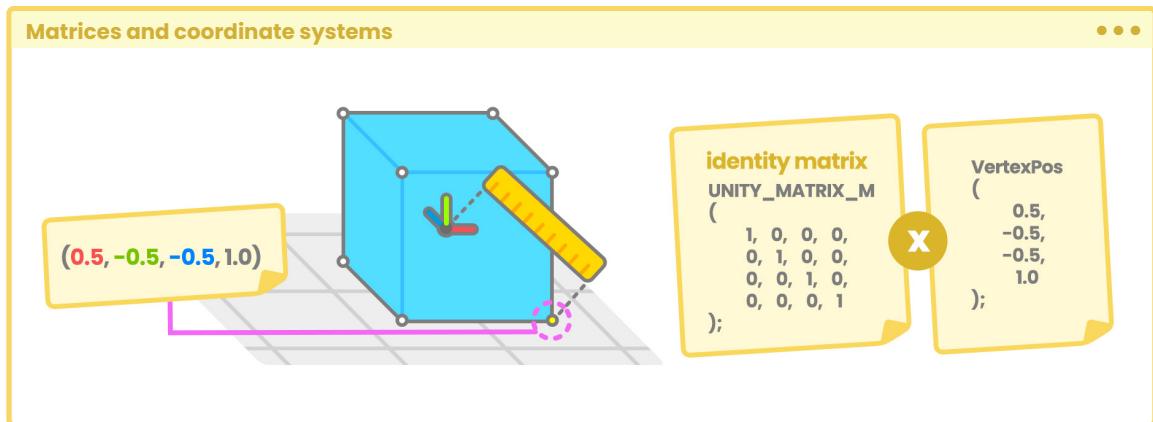
As previously explained in section 1.0.2 talking about vertices, a polygon object has two nodes by default. In Maya, these nodes are known as Transform and Shape, and both are in charge of calculating the position of the vertices in a space called Object-Space, which defines the position of the vertices in relation to the position of the object's center.

The final value of each vertex in the object is multiplied by a matrix known as the **Model Matrix** (**UNITY_MATRIX_M**), which allows you to modify its transformation, rotation and scale values. Every time the object is rotated, has its position or scale changed the Model Matrix is updated.

How does this process occur? To understand it try transforming a Cube in the scene. Start by taking a vertex of the Cube that is in the position $0.5_X, -0.5_Y, -0.5_Z, 1.0_W$ with respect to its center.

It is worth mentioning that the channel W corresponds to a coordinate system called **homogeneous**, which allows the uniform handling of vectors and points. In matrix transformations, the W coordinate can be equal to zero or one. When n_W equals 1, it refers to a point in space, while when it equals 0, it refers to a direction.

Later, this book talks about this system when vectors are multiplied by matrices and vice versa.



(Fig. 1.1.6a. Identity matrix refers to the matrix default values)

One thing to consider with respect to matrices is that a multiplication can be carried out only when the number of columns of the first matrix is equal to the number of rows of the second. As already known, this Model Matrix has a dimension of four rows and four columns, and the position of the vertices has a dimension of four rows and one column.

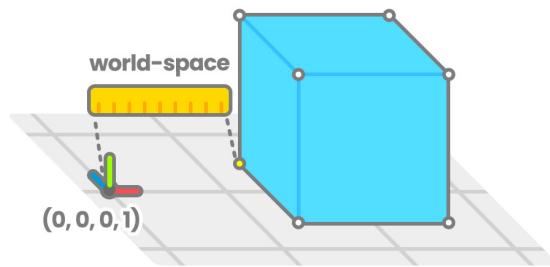
Since the number of columns in the Model Matrix is equal to the number of rows in the position of the vertices, they can be multiplied and the result will be equal to a new matrix of four rows and one column, which would define a new position for the vertices. This multiplication process occurs for all vertices in the object and is carried out in the **Vertex Shader Stage**.

Up to this point you already know that Object-Space refers to the position of a vertex according to its own center, so what does World-Space, View-Space or Clip-Space mean? The concept is basically the same.

World-Space corresponds to the position of a vertex according to the center of the world; to the distance between the starting point of the grid in our scene ($0_X, 0_Y, 0_Z, 1_W$) and the position of a vertex on the object.

If you want to transform a space coordinate from Object-Space to World-Space you can use the internal variable `unity_ObjectToWorld`.

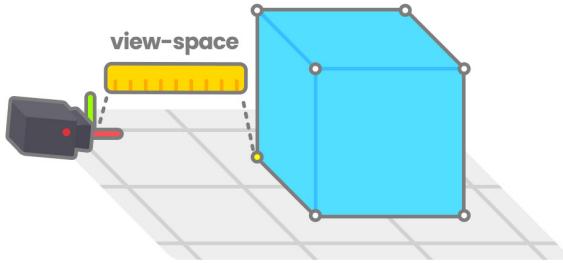
Matrices and coordinate systems



(Fig. 1.1.6b)

View-Space refers to the position of a vertex of our object relative to the camera view. If you want to transform a space coordinate from World-Space to View-Space, you can use the `UNITY_MATRIX_V` matrix.

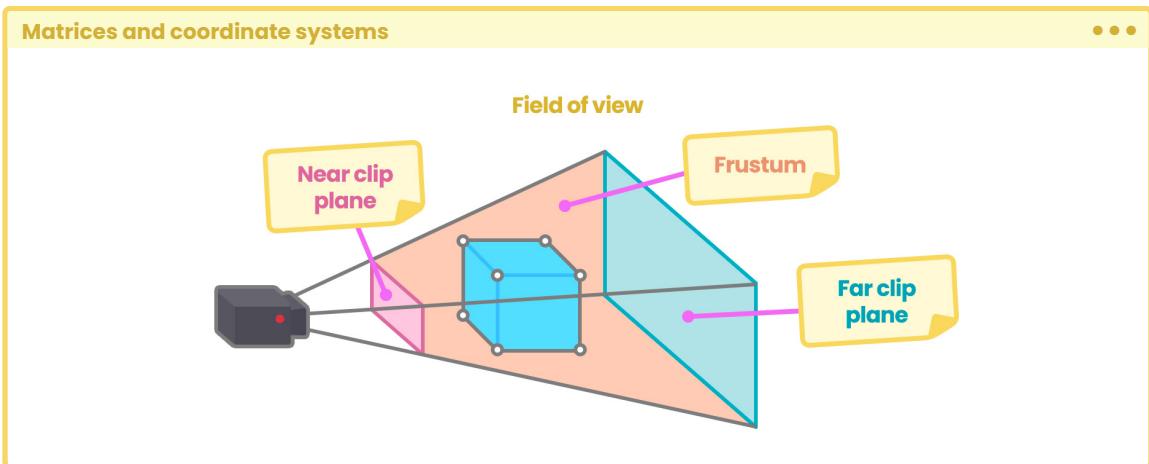
Matrices and coordinate systems



(Fig. 1.1.6c)

Finally, Clip-Space, also known as Projection-Space, refers to the position of an object vertex in relation to the camera's frustum. So, this factor will be affected by the **Near Clipping Plane**, **Far Clipping Plane** and **Field of View**.

If you want to transform a space coordinate from View-Space to Clip-Space, you can do it using the `UNITY_MATRIX_P` matrix.



(Fig. I.I.6d)

In general, the different space coordinates have been talked about at a conceptual level, but what the transformation matrices refer to, has not yet been defined.

For example, the Built-in shader variable `UNITY_MATRIX_MVP` refers to the multiplication of three different matrices. **M** refers to the **Model Matrix**, **V** the **View Matrix**, and **P** the **Projection Matrix**. This matrix is mainly used to transform object vertices from Object-Space to Clip-Space. Remember that the polygonal object has been created in a three-dimensional environment while the screen of the computer, where it will be projected, is two-dimensional, therefore you will have to transform the object from one space to another.

Later, this book will review these concepts in detail when using the `UnityObjectToClipPos(VRG)` function included in the shader, in the Vertex Shader Stage.

Shaders in Unity.

2.0.1. What is a shader?

Assuming having some previous knowledge, you can now go into the topic of shaders in Unity. A shader is a small program with a “.shader” extension which can be used to generate interesting effects for projects. Inside, it has mathematical calculations and lists of instructions (commands) that allow color processing for each pixel within the area covering an object on the computer screen.

This program allows you to draw elements using coordinate systems based on the properties of a polygonal object. The shaders are executed by the GPU since they have a parallel architecture that consists of thousands of small, efficient cores designed to solve tasks simultaneously, while the CPU has been designed for sequential serial processing.

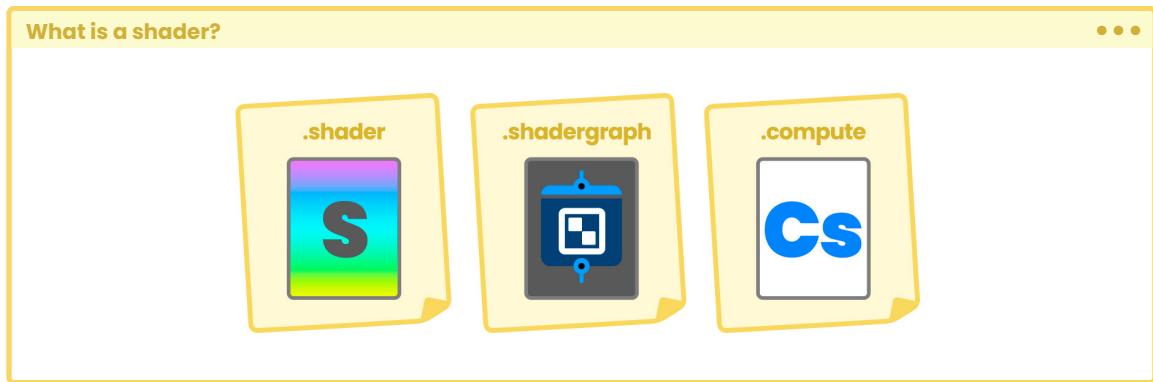
Note that Unity has three types of files associated with shaders. Firstly, there are programs with the **.shader** extension that are capable of compiling in the different types of Render Pipeline.

Secondly, there are programs with the **.shadergraph** extension that can only compile in either Universal RP or High Definition RP. Finally, there are files with the **.hlsl** extension that allow you to create customized functions; generally used within a node type called Custom Function, found in Shader Graph.

These files will be reviewed later on, including those with the extension **.cginc**. For now, just the following associations will be looked at:

- .cginc files are predominantly linked to .shader and CGPROGRAM.
- While .hlsl files are linked to .shadergraph and HLSLPROGRAM.

Knowing this analogy is fundamental because each extension fulfills a different function and is used in specific contexts.



(Fig. 2.0.1a. Reference icons for shaders in Unity)

In Unity, there are at least four types of defined structures to generate shaders, among which you can find:

- The combination of **Vertex Shader** and **Fragment Shader**.
- **Surface Shader** for automatic lighting calculation in Built-in RP.
- **Compute Shader** for more advanced concepts.

Each of these structures has previously described properties and functions that facilitate the compilation process; it is easy to define these operations, since the software adds these structures automatically.

2.0.2. Introduction to the programming language.

Before starting with code definition, you must take into consideration that in Unity there are three programming languages associated with shader development, these are:

- 1 **HLSL** (High Level Shader Language - Microsoft).
- 2 **Cg** (C for Graphics - NVIDIA) which still compiles into the shader but is no longer used in current versions of the software.
- 3 **ShaderLab** (declarative language - Unity) which acts as a link between the program and Unity.

The adventure starts with working with Cg and ShaderLab languages in Built-in RP and later gives way to the introduction of HLSL in Universal RP.

Cg is a high-level programming language designed to compile on most GPUs. It has been developed by NVIDIA in collaboration with Microsoft and uses a syntax very similar to HLSL. The reason why Unity shaders work with the Cg is that it can compile both HLSL and GLSL (OpenGL Shading Language), accelerating and optimizing the process of creating materials for video games.

When a .shader script is created, the code compiles in a field called **CGPROGRAM**. Unity is currently working on providing further support and compatibility between Cg and HLSL. Therefore, it is very likely that in the near future these fields will be replaced by **HLSLPROGRAM** and **ENDHLSL** since HLSL is the official shader programming language in current versions of Unity (version 2019 onwards).

The majority of the shaders in Unity (except Shader Graph, Compute and Ray Tracing) are written within a declarative language called **ShaderLab**. Its syntax can display shader properties in the Unity inspector which allows the manipulation of the values of variables and vectors in real-time, making it easier to get the desired result.

In ShaderLab several properties and commands can be defined manually, one of which is the **Fallback** field, which is compatible with the different types of existing Render Pipelines.

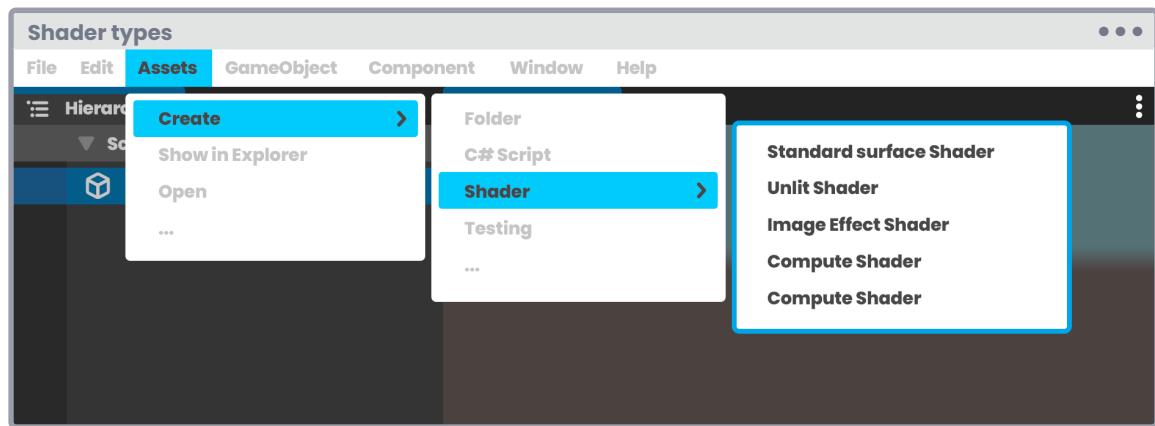
Fallback is a fundamental code field in multiplatform games. It can compile a different shader to one that has generated an error, what does this mean? Basically if the shader fails in its compilation process, Fallback returns a different shader, and so the graphics hardware can continue its work.

SubShader is another field in ShaderLab where you can declare commands and generate passes. When written in Cg/HLSL a shader can contain more than one SubShader or pass. However, in the case of Scriptable RP, a shader can only contain one pass per SubShader.

2.0.3. Shader types.

To start creating a shader, you must first create a new project in Unity. If you are using Unity Hub it is recommended to create the project in the most recent versions of the software, e.g., 2019, 2020 or 2021.

It is important that the project is a **3D template** with Built-in RP to facilitate the understanding of the graphics programming language. Once the project has been created, you must right-click on the **Project Window** (ctrl + 5 or cmd + 5), go to **Create** and select the **Shader** option.



(Fig. 2.0.3a. You can achieve the same result following the Assets / Create / Shader path)

As we can see, there is more than one type of shader, among them, we can find:

- Standard Surface Shader.
- Unlit Shader.
- Image Effect Shader.
- Compute Shader.
- Ray Tracing Shader.

The list of shaders is likely to vary depending on the version of Unity being used. **Shader Graph** could affect the number of shaders that appear in the list if the project were created in Universal RP or High-Definition RP.

It is necessary to understand some concepts before starting your first shader but more details about this subject will be left until later on. Only the Built-in RP default shaders will be used for now.

2.0.4. Standard Surface Shader.

This type of shader is characterized by its code writing optimization that interacts with a basic lighting model and only works in **Built-in RP**. If you want to create a shader that interacts with light, there are two options:

- 1 Use an **Unlit Shader** and add mathematical functions that allow lighting rendering on the material.
- 2 Or use a **Standard Surface Shader** which has a basic lighting model that includes albedo, diffuse, and in some cases specular.

2.0.5. Unlit Shader.

The Lit word refers to a material affected by light, and Unlit is the opposite. The **Unlit Shader** refers to the primary color model and is the base structure that is generally used to create your effects. This type of program, ideal for low-end hardware, has no code optimization; therefore, you can see its complete structure and modify it according to your needs. It works in both Built-in and Scriptable RP.

2.0.6. Image Effect Shader.

It is structurally very similar to an Unlit Shader. However, Image Effects are used mainly in Post-Processing effects in Built-in RP and require the function **OnRenderImage** to work.

```
Image Effect Shader
public Material mat;
void OnRenderImage(RenderTexture src, RenderTexture dest)
{
    Graphics.Blit(src, dest, mat);
}
```

2.0.7. Compute Shader.

This type of program is characterized by running on the graphics card, outside the normal Render Pipeline, and is structurally very different from the previously mentioned shaders.

Unlike a common shader, its extension is **.compute** and its programming language is HLSL. Compute Shaders are used in specific cases to speed up part of the game processing.

Chapter III, section 10.0.2 of this book reviews this type of shader in detail.

2.0.8. Ray Tracing Shader.

Ray Tracing Shader is a type of experimental program with the extension **.raytrace**. It processes Ray Tracing in the GPU and works only in High-Definition RP but has some technical limitations. If you want to work with DXR (DirectX Ray Tracing), you must have at least one GTX 1080 graphics card or equivalent with RTX support, Windows 10 version 1809+ and Unity 2019.3b1 onwards.

This kind of program can be used to replace the **.compute** type shader in processing algorithms for ray-casting, e.g., global illumination, reflections, refraction, or caustics.

Properties, commands and functions.

3.0.1. Structure of a Vertex-Fragment Shader.

Start by creating an **Unlit** shader called **USB_simple_color** in your project. You will use it to analyze its general structure, and not include custom functions for the moment. This process can be easily accomplished by following the detailed explanation in Section 2.0.3, which talked about shader types.

As you already know, this type of shader is a primary color model and has little optimization in its code. This will allow you to analyze its different properties and functions in depth.

When you generate a shader for the first time, Unity adds default code to facilitate the compilation process. Inside the program you can find blocks of code structured in such a way that the GPU can interpret them. If you open your previously created shader, its structure should look like this:

```
Structure of a Vertex-Fragment Shader ... ● ● ●

Shader "Unlit/USB_simple_color"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags {"RenderType"="Opaque"}
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog
        }
    }
}
```

Continued on the next page.

```
#include "UnityCG.cginc"

struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
};

struct v2f
{
    float2 uv : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float4 vertex : SV_POSITION;
};

sampler2D _MainTex;
float4 _MainTex;

v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o, o.vertex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

Continued on the next page.

```

        ENDCG
    }
}
}
```

Likely, you will not fully understand what is happening in the different blocks of code from the shader you just created. However, to begin your study, you must pay attention to its general structure.

Structure of a Vertex-Fragment Shader

```

Shader "InspectorPath/ShaderName"
{
    Properties
    {
        // properties in this field
    }
    SubShader
    {
        // SubShader configuration in this field
        Pass
        {
            CGPROGRAM
            // programma Cg - HLSL in this field
            ENDCG
        }
    }
    Fallback "ExampleOtherShader"
}
```



(The shader structure is the same in both Cg and HLSL, the only things that change are the program blocks in Cg and HLSL. Both compile in current versions of Unity for compatibility)

The shader starts with the **InspectorPath**, which refers to the Inspector in the Unity interface, where shaders are selected to apply to materials, and a **name** (ShaderName) to identify it. Then adds the properties (e.g., textures, vectors, colors, etc.), to the **SubShader** and finally there is the **Fallback**, which is optional.

You must remember that you cannot apply a shader directly to an object in your scene, instead, you must carry out the process through a previously created material. Your **USB_simple_color** shader has the default Unlit path, meaning that you will have to perform the following action from the Unity interface:

- 1 Select your material.
- 2 Go to the Inspector.
- 3 Look for the “Unlit” path.
- 4 Apply the USB_simple_color shader to the material.

A structural factor you must consider is that the GPU will read the program from the top down, in a linear fashion. This is directly related to the functions included in the shader, since, if a function is positioned below the code block where it will be used, the GPU will not be able to read it and generate a processing error. If this is the case, the Fallback will assign a different shader so that the graphics hardware can continue processing it.

Do the following exercise to understand this concept.

```
Structure of a Vertex-Fragment Shader
```

```
// 1. declare function
float4 ourFunction()
{
    // your code here ...
}

// 2. use the function
fixed4 frag (v2f i) : SV_Target
{
    // the function is being used here
    float4 f = ourFunction();
    return f;
}
```

The syntax of the above functions may not be fully understood. These have been created only to conceptualize the position of one function to another.

Section 4.0.4 talks in detail about the structure of a function. For now, the only important thing is that in the previous example the structure is correct because the function **ourFunction** has been written where the block of code is placed. The GPU will first read the function **ourFunction** and then it will continue to the fragment stage called **frag**.

Now look at a different case.

Structure of a Vertex-Fragment Shader

```
// 2. you use your function
fixed4 frag (v2f i) : SV_Target
{
    // the function is being used here
    float4 f = ourFunction();
    return f;
}

// 1 . declare the function
float4 ourFunction()
{
    // your code here ...
}
```

This structure will generate an error, because the same function has been written below the code block which it is using.

3.0.2. ShaderLab Shader.

Most Cg or HLSL shaders will start with the declaration of the **Shader**, then its path in the **Inspector** and finally its **name**.

Properties such as SubShader and Fallback are written inside the **Shader** field in **ShaderLab** declarative language.

```
ShaderLab Shader
Shader "InspectorPath/ShaderName"
{
    // write ShaderLab code here ...
}
```

Since **USB_simple_color** includes the path **Unlit** by default, if you want to assign it to a material, then go to the Unity Inspector, look for the **Unlit** path and then select the shader. Both the path and the name of the shader can be changed if necessary by the project organization.

```
ShaderLab Shader
// default value
Shader "Unlit/USB_simple_color"
{
    // write ShaderLab code here ...
}

// customized path to USB (Unity Shader Bible)
Shader "USB / USB_simple_color"
{
    // write ShaderLab code here ...
}
```

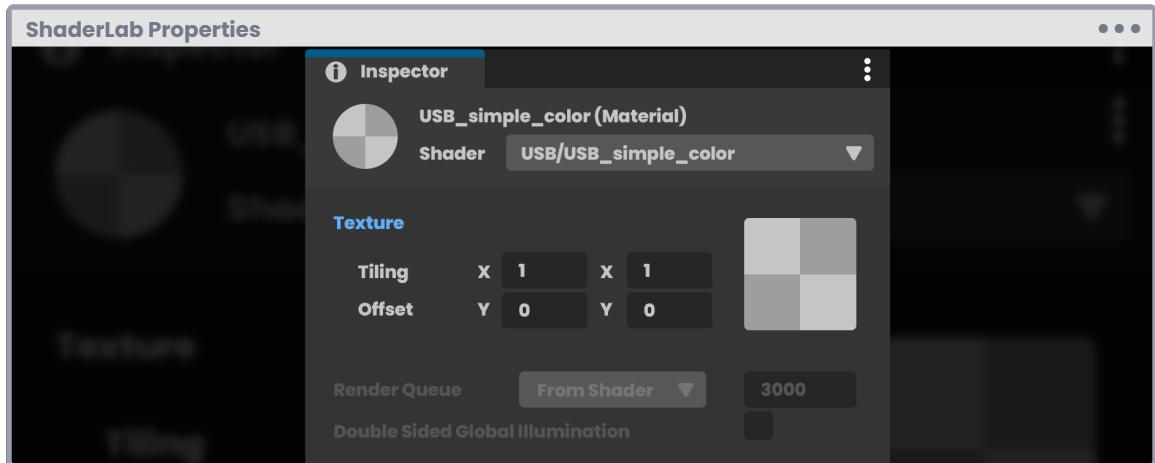
3.0.3. ShaderLab Properties.

The properties correspond to a list of parameters that can be manipulated from the Inspector. There are eight different properties both in value and usefulness. These properties are used in respect of the shader that you want to create or modify, either dynamically or at runtime.

The syntax for declaring a property is as follows:

```
ShaderLab Properties
PropertyName ("display name", type) = defaultValue.
```

PropertyName refers to the name of the property while **display name** corresponds to the name of the property that will be displayed in the Inspector. **Type** indicates the type of property, e.g., color, vector, texture, etc. And finally, as its name implies, **defaultValue** is the default value assigned to the property.



(Fig. 3.0.3a)

If you look at the properties of the new shader you will notice that there is a texture property that has been declared inside the field, this can be corroborated in the following line of code.



One factor to consider is that when a property is declared, it remains open within the property field, therefore you must not put a semicolon (;) at the end of the code line, otherwise the GPU will not be able to read the program.

3.0.4. Number and slider properties.

These types of properties add numerical values to the shader. Suppose that you want to create a shader with lighting functions, where 0.0f equals 0% illumination and 1.0f equals 100% illumination. A range can be created between the two values and then you can configure the minimum, maximum, and default values.

The following syntax declares numbers and sliders in our shader:

```
Number and slider properties ...
// name ("display name", Range(min, max)) = defaultValue
// name ("display name", Float) = defaultValue
// name ("display name", Int) = defaultValue

Shader "InspectorPath/ShaderName"
{
    Properties
    {
        _Specular ("Specular", Range(0.0, 1.1)) = 0.3
        _Factor ("Color Factor", Float) = 0.3
        _Cid ("Color id", Int) = 2
    }
}
```

Three properties were declared in the previous example:

- 1 A **floating range** type called **_Specular**.
- 2 Another **floating scale** type called **_Factor**.
- 3 And finally, an **integer** type called **_Cid**.

3.0.5. Color and vector properties.

Continuing with the previous analogy, suppose that the shader can change color at runtime. Now a color property can be added so you can modify its **RGBA** (Red, Green, Blue and Alpha) values.

Use the following syntax to declare colors and vectors in the shader:

```
Color and vector properties • • •

// name ("display name", Color) = (R, G, B, A)
// name ("display name", Vector) = (0, 0, 0, 1)

Shader "InspectorPath/ShaderName"
{
    Properties
    {
        _Color ("Tint", Color) = (1, 1, 1, 1)
        _VPos ("Vertex Position", Vector) = (0, 0, 0, 1)
    }
}
```

In this example two properties have been declared:

- 1 One an RGBA color type called **_Color**.
- 2 The other is a vector type called **_VPos** that includes four dimensions.

It is easy to see that the default color of the **_Color** property equals white, since 1.0f is the maximum lighting value for a pixel (I_R, I_G, I_B, I_A), and 0.0f is the minimum.

3.0.6. Texture properties.

If you want to place a texture on any object, e.g., a character, you would have to declare a 2D texture and then implement it through the function called **tex2D(S_{RG}, UV_{RG})**, which asks for two parameters by default:

- 1 A sampler2D type texture.
- 2 And the object UV coordinates.

A commonly used property in video games is the **Cube** which refers to a **Cubemap**. This texture type is extremely useful for generating reflection maps, e.g., reflections on a character's armor or on metallic elements in general.

Other types of textures that can be found are 3D type. They are used less frequently than the previous ones since they are volumetric and have an additional coordinate for their spatial calculation.

The following syntax declares textures in the shader:

```
Texture properties
{
    // name ("display name", 2D) = "defaultColorTexture"
    // name ("display name", Cube) = "defaultColorTexture"
    // name ("display name", 3D) = "defaultColorTexture"

    Shader "InspectorPath/ShaderName"
    {
        Properties
        {
            _MainTex ("Texture", 2D) = "white" {}
            _Reflection ("Reflection", Cube) = "black" {}
            _3DTexture ("3D Texture", 3D) = "white" {}
        }
    }
}
```

When declaring a property, it is very important to consider that it will be written in **ShaderLab** declarative language while the program will be written in either Cg or HLSL. As they are two different languages, you must create **connection variables**.

These variables are declared globally using the word **uniform**. However, this step can be skipped because the program recognizes them as global variables. So, to add a property to a .shader you must first declare the property in ShaderLab, then the global variable within the path using the same name in Cg or HLSL, and then it is ready.

• • •

Texture properties

```

Shader "InspectorPath/ShaderName"
{
    Properties
    {
        // declare the properties
        _MainTex ("Texture", 2D) = "white" {}
        _Color ("Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // add connection variables
            sampler2D _MainTex;
            float4 _Color;
            ...
            half4 frag (v2f i) : SV_Target
            {
                // use the variables
                half4 col = tex2D(_MainTex, i.uv);
                return col * _Color;
            }
            ENDCG
        }
    }
}

```

In the above example two properties have been declared:

- _MainTex for two-dimensional texture types.
- And _Color for the color.

Then two connection variables have been created inside our CGPROGRAM, these correspond to:

- `sampler2D _MainTex`
- `and float4 _Color.`

It is essential that both the properties and the connection variables have the same name, so that the program can recognize their nature.

Section 3.2.7 will detail the operation of a 2D sampler and talk about data types.

3.0.7. Material Property Drawer.

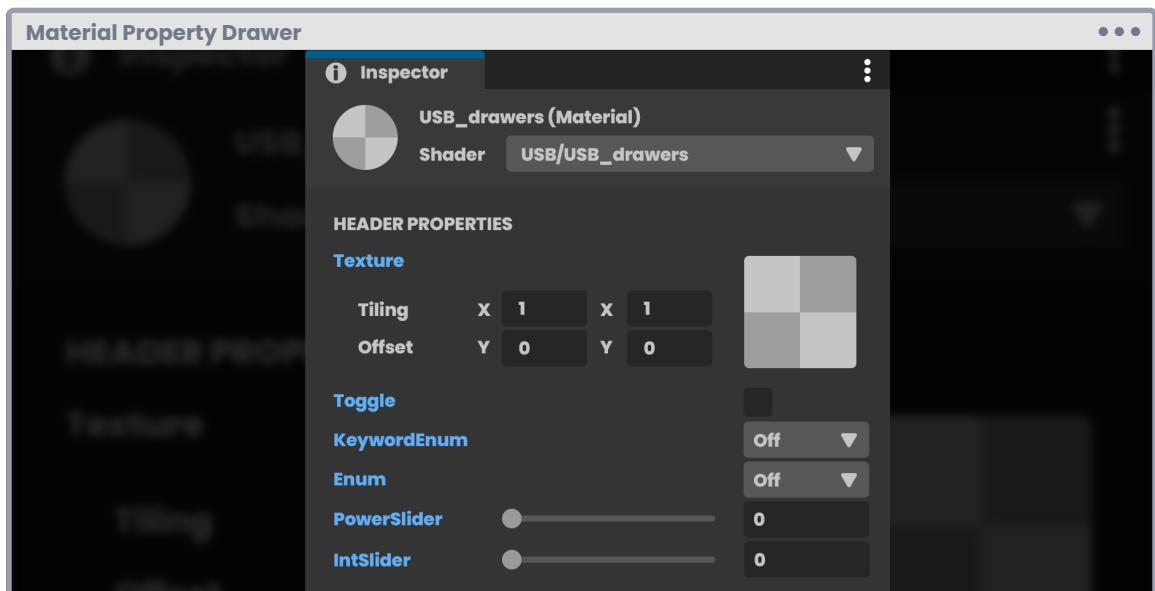
Another type of property that you can find in ShaderLab is **drawers**. This class allows you to generate custom properties in the Inspector, thus facilitating the programming of conditionals in the shader.

By default, these properties are not included in the shader, instead, you must declare them according to your needs. To date, there are seven different drawers:

- **Toggle.**
- **Enum.**
- **KeywordEnum.**
- **PowerSlider.**
- **IntRange.**
- **Space.**
- **Header.**

Each one of them has a specific function and is declared independently. Thanks to their particularity, multiple states can be generated within the program, allowing the creation of dynamic effects without the need to change materials at execution time. These drawers are generally used together with two types of **shader variants**, which refer to:

- **#pragma multi_compile.**
- **and #pragma shader_feature.**



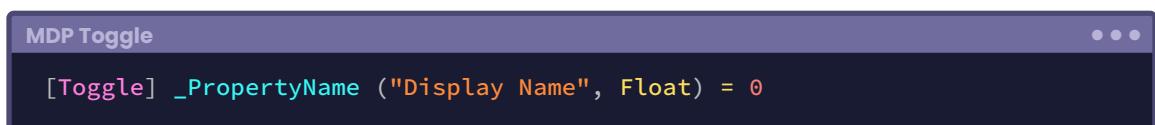
(Fig. 3.0.7a)

3.0.8. MPD Toggle.

ShaderLab does not support boolean type properties, instead, the **Drawer Toggle** fulfills the same function. This drawer allows switching from one state to another using a condition within the shader. To run it, you must first add the word **Toggle** between brackets and then declare the property.

Its default value must be an integer, either zero or one, why? Because zero symbolizes "Off" and one symbolizes "On."

Its syntax is as follows:



Something to consider when working with this drawer is that, if you want to implement it you have to use the #pragma **shader_feature**. This belongs to the Shader Variants and its function is to generate different conditions according to its state (enabled or disabled).

To understand its implementation, do the following operation:

```
MDP Toggle
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        _Color ("Color", Color) = (1, 1, 1, 1)
        // declare drawer Toggle
        [Toggle] _Enable ("Enable ?", Float) = 0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // declare pragma
            #pragma shader_feature _ENABLE_ON
            ...
            float4 _Color;
            ...
            half4 frag (v2f i) : SV_Target
            {
                half4 col = tex2D(_MainTex, i.uv);
                // generate condition
                #if _ENABLE_ON
                    return col;
                #else
                    return col * _Color;
                #endif
            }
            ENDCG
        }
    }
}
```

In the previous example, a Toggle-type property called `_Enable` has been declared. Then the Shader Variant (**shader_feature**) found in the CGPROGRAM added. However, unlike the property in the program, the Toggle has been declared as `_ENABLE_ON`, why is this? Shader Variants are **constants**, therefore they are written entirely in CAPITAL LETTERS.

In a different case, if you had named the property `_Change`, then it should be added as `_CHANGE` in the Shader Variant.

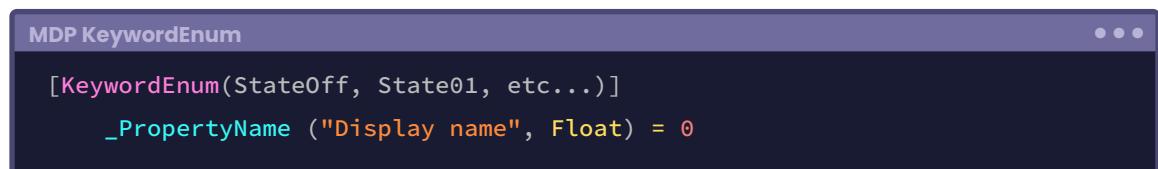
Continuing with the explanation, its position `_ON` corresponds to the default state of the Drawer, that is to say, if the property `_Enable` is active, the color of the texture will be the same as in the Fragment Shader Stage, otherwise, multiply `_Color` to it.

It is worth mentioning that **shader_feature pragma** cannot compile multiple variants for an application, what does this mean? Unity will not include variants that are not being used in the final build, which means that you will not be able to move from one state to another at execution time. For this, you must use the **KeywordEnum Drawer** that has the variant shader **multi_compile**.

3.0.9. MPD KeywordEnum.

Unlike a Toggle, this drawer allows you to configure up to nine different states generating a pop-up style menu in the Inspector. To execute it you must add the word **KeywordEnum** in brackets and then list the set of states that you want to use.

Use the following syntax:



To declare this drawer within the code, use the Shader Variant **shader_feature** or **multi_compile**. The choice will depend on the number of variants that are needed in the final build.

As you already know, **shader_feature** will only export the selected variant from the Material Inspector, whereas **multi_compile** exports all variants that are found in the shader, regardless of

whether they are used or not. Given this feature, `multi_compile` is ideal for exporting or compiling multiple states that will change at execution time (e.g. star status in Super Mario).

To understand its implementation, try the following operation:

```
MDP KeywordEnum ...
```

```
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        // declare drawer Toggle
        [KeywordEnum(Off, Red, Blue)]
        _Options ("Color Options", Float) = 0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // declare pragma and conditions
            #pragma multi_compile _OPTIONS_OFF _OPTIONS_RED _OPTIONS_BLUE
            ...
            half4 frag (v2f i) : SV_Target
            {
                half4 col = tex2D(_MainTex, i.uv);

                // generate conditions
                #if _OPTIONS_OFF
                    return col;
                #elif _OPTIONS_RED
                    return col * float4(1, 0, 0, 1);
                #elif _OPTIONS_BLUE
                    return col * float4(0, 0, 1, 1);
                #endif
            }
        }
    }
}
```

Continued on the next page.

```

    }
    ENDCG
}
}
}
}
```

In this example, a **KeywordEnum** type property called **_Options** was declared and three states configured for it (Off, Red and Blue). Later add them to the **multi_compile** found in CGPROGRAM and declare them as constants.

MDP KeywordEnum



```
#pragma multi_compile _OPTIONS_OFF _OPTIONS_RED _OPTIONS_BLUE
```

Finally, using the conditionals, define the three states for the shader that correspond to color changes for the main texture.

3.1.0. MPD Enum.

This drawer has some similarities to KeywordEnum. Its difference lies in its arguments, in that you can define more than one **value/id** and pass these properties to a command in the shader. In this way, they can be run dynamically from the Inspector.

Its syntax is as follows:

MDP Enum



```
[Enum(valor, id_00, valor, id_01, etc ... )]  
_PropertyName ("Display Name", Float) = 0
```

Enums do not use Shader Variants but are declared by command or function. Perform the following operation to understand how they work.

```
MPD Enum
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        // declare drawer
        [Enum(Off, 0, Front, 1, Back, 2)]
        _Face ("Face Culling", Float) = 0
    }
    SubShader
    {
        // pass the property to the command
        Cull [_Face]
        Pass { ... }
    }
}
```

As presented in this example, an **Enum** type property named **_Face** was declared, and the following **values/id** used as the argument:

- Off, 0.
- Front, 1.
- Back, 2.

Then, this property was added to the Cull command found in the **SubShader**, this way you will be able to control its values from the Inspector and thus modify the face you want to render on the object. Later, section 3.2.1 details the operation of the Cull command.

3.1.1. MPD PowerSlider and IntRange.

These drawers are very useful when working with numerical ranges and precision. On the one hand, there is the **PowerSlider**, which generates a non-linear slider with curve control.

Using the following syntax:

MPD PowerSlider and IntRange

```
[PowerSlider(3.0)] _PropertyName ("Display name", Range (0.01, 1)) = 0.08
```

On the other hand, there is the **IntRange** which, as its name mentions, adds a numerical range of integer values.

Its syntax is as follows:

MPD PowerSlider and IntRange

```
[IntRange] _PropertyName ("Display name", Range (0, 255)) = 100
```

Note that, if you want to use these properties within the shader, they must be declared within the CGPROGRAM in the same way as a conventional property.

To understand how to use it, do the following:

MPD PowerSlider and IntRange

```
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        // declare drawer
        [PowerSlider(3.0)]
        _Brightness ("Brightness", Range (0.01, 1)) = 0.08
        [IntRange]
        _Samples ("Samples", Range (0, 255)) = 100
    }
}
```

Continued on the next page.

```

SubShader
{
    Pass
    {
        CGPROGRAM
        ...
        // generate connection variables
        float _Brightness;
        int _Samples;
        ...
        ENDCG
    }
}
}

```

In the example above:

- A **PowerSlider** called **_Brightness** was declared.
- And then an **IntRange** called **_Samples**.

Finally, using the same names, the connection variables were generated within the CGPROGRAM.

3.1.2. MPD Space and Header.

These drawers are quite useful in organizing the properties you want to project in the Inspector. **Space** provides a given space between two properties, while **Header** refers to a header.

In the following, you will add ten points of space between two properties by implementing the Drawer Space.

```

MPD Space and Header
...
_PropertyName01 ("Display name", Float) = 0
// add space
[Space(10)]
_PropertyName02 ("Display name", Float) = 0

```

Continuing with the same analogy, add a header using the Drawer Header.

```
MDP Space and Header ...
// add the header
[Header(Category name)]
(PropertyName01 ("Display name", Float) = 0
(PropertyName02 ("Display name", Float) = 0
```

The Header is very useful when generating categories. In the example a small header was added before starting the properties, which will be visible only from the Inspector.

Next you will put into practice a real implementation case to understand both drawers.

```
MDP Space and Header ...
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        [Header(Specular properties)]
        _Specularity ("Specularity", Range (0.01, 1)) = 0.08
        _Brightness ("Brightness", Range (0.01, 1)) = 0.08
        _SpecularColor ("Specular Color", Color) = (1, 1, 1, 1)
        [Space(20)]
        [Header(Texture properties)]
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader { ... }
}
```

3.1.3. ShaderLab SubShader.

The second component of a shader is the **SubShader**. Each shader is made up of at least one SubShader for the perfect loading of the program. When there is more than one SubShader, Unity will process each of them and take the most suitable according to the hardware characteristics, starting from the first to the last on the list.

To understand this, assume that the shader is going to run on hardware that includes **Metal Graph API** (iOS). For this, the program will find the first SubShader that supports the application and run it. When a SubShader is not supported, Unity will try to use the Fallback component, mentioned in section 3.0.1, so that the hardware can continue with its task without graphical errors.

```
ShaderLab SubShader
{
    Shader "InspectorPath/ShaderName"
    {
        Properties { ... }
        SubShader
        {
            // shader configuration here
        }
    }
}
```

If you pay attention to the **USB_simple_color** shader, the SubShader will appear as follows in its default values:

```
ShaderLab SubShader
{
    Shader "USB/USB_simple_color"
    {
        Properties { ... }
        SubShader
        {
            Tags { "RenderType"="Opaque" }
            LOD 100

            Pass { ... }
        }
    }
}
```

3.1.4. SubShader Tags.

Tags are labels that show how and when our shaders are processed. Like a GameObject Tag, these can be used to recognize how a shader will be rendered or how a group of shaders will behave graphically.

The syntax for all Tags is as follows:

SubShader Tags

```
Tags
{
    "TagName1"="TagValue1"
    "TagName2"="TagValue2"
}
```

These can be written in two different fields, either within the **SubShader** or inside the **pass**. All this depends on the result you want to obtain. If a tag is written inside the SubShader it will affect all the passes that are included in it, but if it is written inside the pass, it will only affect the selected pass.

A frequently used Tag is **Queue**, which defines how the surface of the object will look. By default, all volume is defined as **Geometry**, that is, it does not have transparency.

If you look at the **USB_simple_color** shader, you will find the following line of code inside the SubShader, which defines an opaque surface for the render type.

SubShader Tags

```
SubShader
{
    Tags { "RenderType"="Opaque" "Queue"="Geometry" }
    LOD 100

    Pass { ... }
}
```

3.1.5. Queue Tag.

By default, this Tag does not appear graphically as a line of code, because it is compiled automatically in the GPU. Its function is directly related to the object processing order for each material.



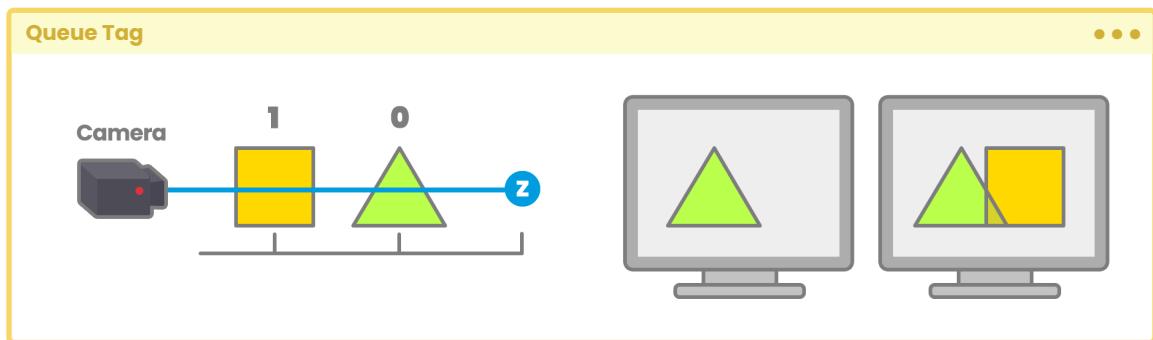
This Tag has a close relationship between the **camera** and the **GPU**. Every time an object is positioned in the scene, its information is passed to the computing units (e.g. position of vertices, Normals, color, etc.). In the case of **Game View**, it is exactly the same, with the difference that the information sent to the GPU corresponds to the object positioned inside the frustum of the camera.

Once this information is located in the GPU, the data is sent to the **VRAM** (Video Random Access Memory), also known as the **Frame Buffer**, which draws the object onto the computer screen.

The process of drawing an object is called a **Draw call**. While the more passes a shader has, the more draw calls there will be in the rendering. A pass is equivalent to a Draw Call, therefore, a multi-pass shader will reflect the same amount of Draw Calls depending on how many passes it has.

Now, how does the GPU perform such a process? For this you must understand how the **painting algorithm** works. This process takes as a reference the order of the objects in the scene, starting with the one farthest from the camera on its Z_{AX} axis, and ending with the one closest to the camera. Finally, the elements are drawn on screen following the same order.

It should be noted that Unity renders opaque objects in batch-optimized sorting mode, which minimizes transition states, i.e., shader execution, Vertex Buffer, textures, Buffer changes, and more. In this case, the Z-Buffer ensures that the out-of-depth-order draws objects correctly, from front to back, similarly to the behavior of the reverse painting algorithm, while for transparent objects, the conventional painting algorithm is used.



(Fig. 3.1.5a. Considering both transparent objects, the triangle is drawn first on the screen because it is farther away from the camera. The square is the last to be drawn, both generate two Draw Calls)

Every material in Unity has a processing queue called **Render Queue** where you can modify the processing order of objects on the GPU.

There are two ways to modify the Render Queue:

- 1 Through the properties of the material in the Inspector.
- 2 Or using the **Tag Queue**.

If the **Queue** value in the shader is modified, by default the material Render Queue applied by the program will also be modified.

Queue has number values ranging from 0 to 5000, where 0 corresponds to the farthest element and 5000 to the closest element to the camera. These order values have predefined groups, some important ones are:

- **Background.**
- **Geometry.**
- **AlphaTest.**
- **Transparent.**
- **Overlay.**

`Tags { "Queue"="Background" }`

goes from 0 to 1499, default value 1000.

`Tags { "Queue"="Geometry" }`

goes from 1500 to 2399, default value 2000.

`Tags { "Queue"="AlphaTest" }`

goes from 2400 to 2699, default value 2450.

`Tags { "Queue"="Transparent" }`

goes from 2700 to 3599, default value 3000.

`Tags { "Queue"="Overlay" }`

goes from 3600 to 5000, default value 4000.

Background is used mainly for elements that are very far from the camera (e.g. skybox).

Geometry is the default value in the Queue and is used for opaque objects in the scene (e.g., primitives and objects in general).

AlphaTest is used on semi-transparent objects that must be in front of an opaque object, but behind a transparent object (e.g., glass, grass, or vegetation).

Transparent is used for transparent elements that must be in front of others.

Finally, **Overlay** corresponds to those elements that are foremost in the scene (e.g., UI images).

```
Queue Tag
-----
Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader
    {
        Tags { "Queue"="Geometry" }
    }
}
```

High-Definition RP uses the Render Queue in a different way than Built-in RP since the materials do not directly show this property in the Inspector, instead it introduces two control methods which are:

- 1 Material order.
- 2 And Renderer order.

Together, HDRP uses these two order methods for object processing control.

3.1.6. Render Type Tag.

According to the official documentation in Unity,

*Use the **RenderType** tag to overwrite the behavior of a shader.*

What does the above statement mean? Basically, with this Tag, you can change from one state to another in the SubShader, adding an effect on any material that matches a given determined configuration (Type).

You need at least two shaders so it can carry out its function:

- 1 A replacement one (color or effect that you want to add at run time).
- 2 And another to be replaced (shader assigned to the material).

Its syntax is as follows:

Render Type Tag

```
Tags { "RenderType"="type" }
```

Like the Tag Queue, **RenderType** has different configurable values that vary depending on the task to be performed. Among them are:

- Opaque. Default.
- Transparent.
- TransparentCutout.
- Background.
- Overlay.
- TreeOpaque.
- TreeTransparentCutout.
- TreeBillboard.
- Grass.
- GrassBillboard.

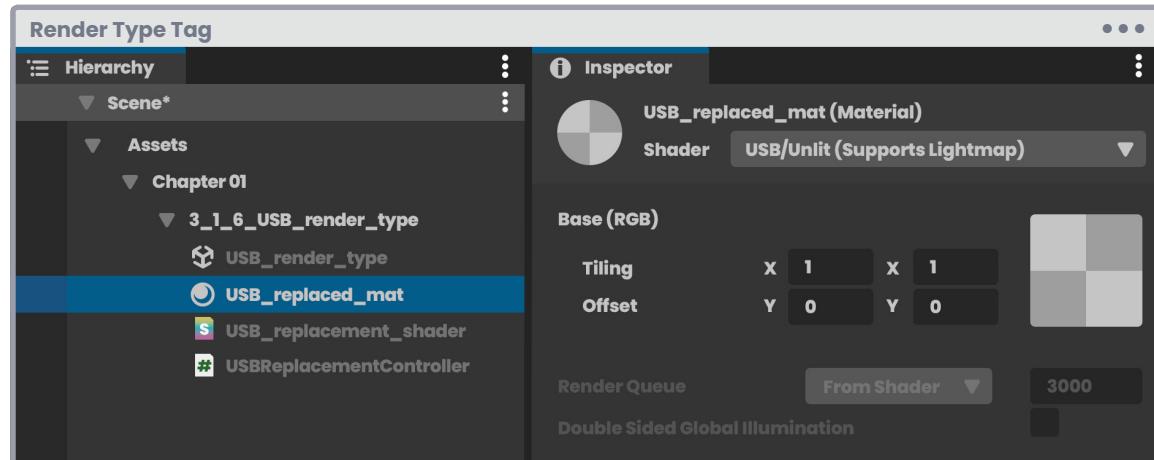
By default, the **Opaque** type is set every time a new shader is created. Likewise, most of the Built-in shaders in Unity come with this assigned value, since they do not have a configuration for transparencies. However, you can freely change this category; everything will depend on the effect you want to apply on a coincidence.

To understand the concept in depth, do the following. In the project,

- 1 Make sure to create some 3D objects in the scene.
- 2 Generate a **C# script** called **USBReplacementController**.
- 3 Then create a shader called **USB_replacement_shader**.
- 4 Finally, add a material called **USB_replaced_mat**.

A shader will be assigned dynamically over the material **USB_replaced_mat** using the **Camera.SetReplacementShader**. To perform the function the material shader must have a **Tag RenderType** equal to the replacement shader.

To illustrate this, assign the **Mobile/Unlit** shader to **USB_replaced_mat**. This built-in shader has a **RenderType** type **Tag** that equals **Opaque**. Therefore, the shader **USB_replacement_shader** must match the same **RenderType** for the operation to work.



(Fig. 3.1.6a. Unlit shader (Supports Lightmap) has been assigned over the material USB_replaced_mat)

The **USBReplacementController** script must be assigned directly to the camera as a component. This controller will be in charge of replacing a shader with one of a similar nature, as long as they have the same configuration in the **RenderType**.

• • •

Render Type Tag

```

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

[ExecuteInEditMode]
public class USBReplacementController : MonoBehaviour
{
    // replacement shader
    public Shader m_replacementShader;

    private void OnEnable()
    {
        if(m_replacementShader != null)
        {
            // the camera will replace all the shaders in the scene
            // with the replacement one the render type configuration
            // must match in both shaders
            GetComponent<Camera>().SetReplacementShader(
                m_replacementShader, "RenderType");
        }
    }

    private void OnDisable()
    {
        // reset the default shader
        GetComponent<Camera>().ResetReplacementShader();
    }
}

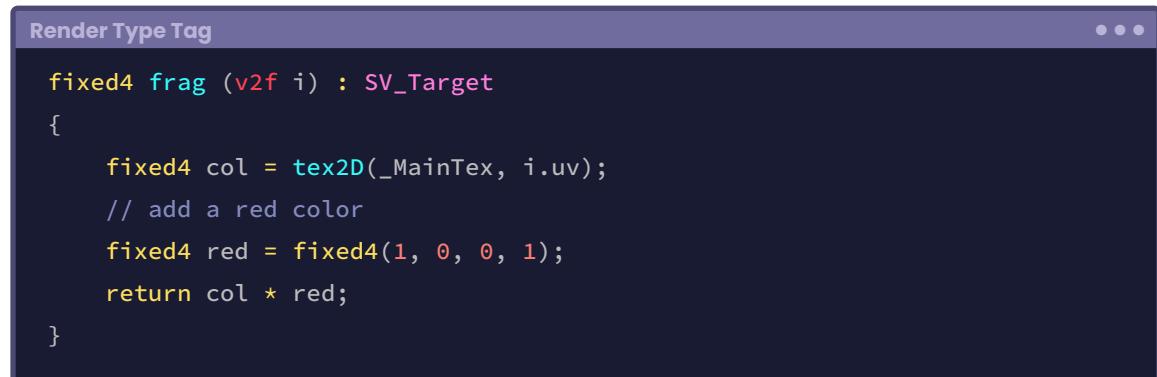
```

It is worth mentioning that the **[ExecuteInEditMode]** function has been defined over the class. This property allows changes to the preview in edit mode.

USB_replacement_shader will be used as a replacement shader.

As you already know, when a new shader is created, its RenderType is configured to Opaque. Consequently, **USB_replacement_shader** may replace the Unlit shader you previously assigned to the material.

To preview the changes clearly, go to the Fragment Shader Stage of **USB_replacement_shader** and add a red color, then multiply it by the output color.



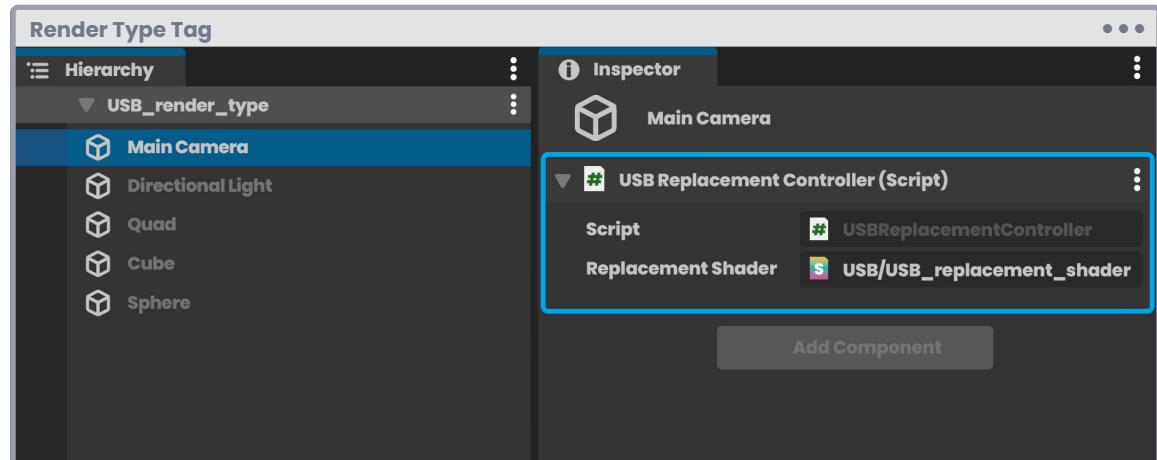
```

Render Type Tag

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // add a red color
    fixed4 red = fixed4(1, 0, 0, 1);
    return col * red;
}

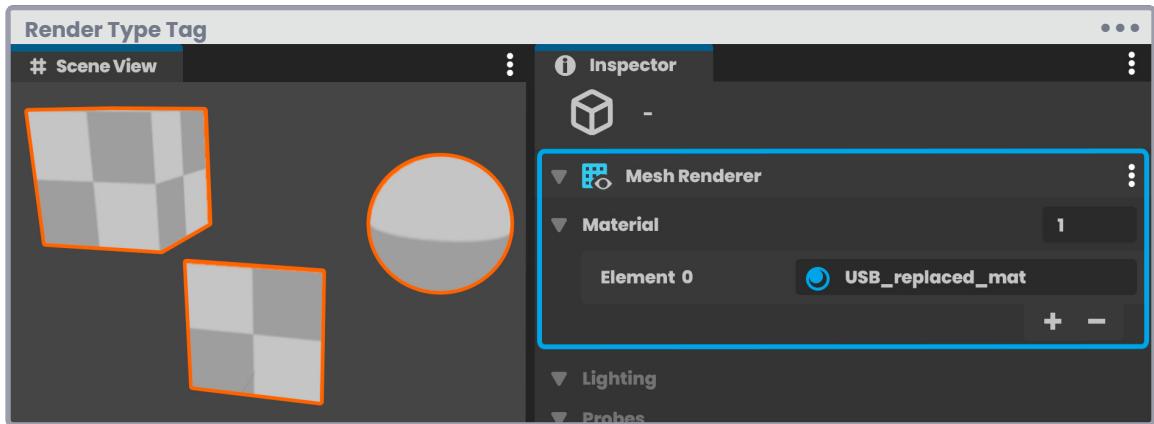
```

You must make sure to include **USB_replacement_shader** in the Shader type replacement variable found in the **USBReplacementController** script.



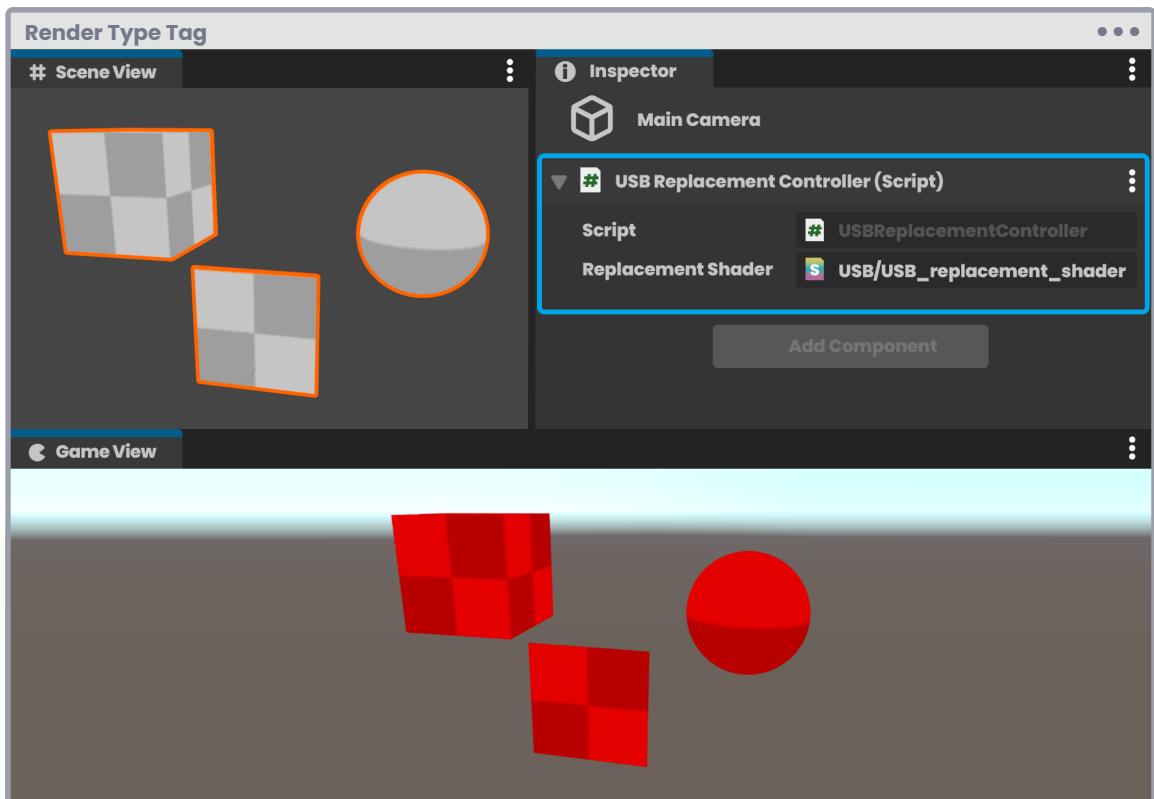
(Fig. 3.1.6b. The **USBReplacementController** has been assigned to the camera)

Also, those objects that were added previously in the scene must have the material **USB_replaced_mat**.



(Fig. 3.1.6c. The material `USB_replaced_mat` has been assigned to 3D objects; to a Quad, a Cube and a Sphere)

Since the `USBReplacementController` class has the `OnEnable` and `OnDisable` functions included, if you activate or deactivate the script, you can see how the built-in shader `Unlit` is replaced by `USB_replacement_shader` in edit mode, applying a red color in the rendering.



(Fig. 3.1.6d. The `USB_replacement_shader` has replaced the built-in shader `Unlit` in the final rendering)

3.1.7. SubShader Blending.

Blending is the process of mixing two pixels into one. Its command is compatible with both Built-in RP and Scriptable RP.

Blending occurs in a stage called **Merging** which combines the final color of a pixel with the Frame Buffer. This stage, which occurs at the end of the Render Pipeline; after the **Fragment Shader Stage**, is where the Stencil-Buffer, Z-Buffer and Color Blending are executed.

By default, the line of code associated with the command is not included in the shader, since it is an optional function and is mainly used when working with transparent objects, that is, with pixels of low opacity.

Its default value is **Blend Off** which results in an opaque surface. However, you can enable different types of blending and thus generate effects similar to those that appear in Photoshop.

Its syntax is as follows:

SubShader Blending



```
Blend [SourceFactor] [DestinationFactor]
```

Blend is a command that requires two values called factors for it to work. According to a simple equation, it will be the final color obtained on screen. According to official Unity documentation, such equation corresponds to the following operation:

$$B = \text{SrcFactor} * \text{SrcValue} [\text{OP}] \text{DstFactor} * \text{DstValue}.$$

To understand this operation, you must consider the following:

- The **Fragment Shader Stage** occurs first.
- And then, as an optional process; the **Merging Stage**.

SrcValue (Source Value), which has been processed in the **Fragment Shader Stage**, corresponds to the pixel's RGB color output.

DstValue (Destination Value) corresponds to the RGB color that has been written in the **Destination Buffer**, better known as **Render Target** (SV_Target). When the Blending options are not active in the shader, SrcValue overwrites DstValue. However, if you activate this operation, both colors are mixed to get a new color, which overwrites the previous DstValue.

SrcFactor (Source Factor) and DstFactor (Destination Factor) are three dimensional vectors that vary depending on their configuration. Their main function is to modify the SrcValue and DstValue values to achieve interesting effects.

Some factors that you can find in the Unity documentation are:

- **Off**, disables Blending options.
- **One**, (1, 1, 1).
- **Zero**, (0, 0, 0).
- **SrcColor** equals the RGB values of the SrcValue.
- **SrcAlpha** equals the Alpha value of the SrcValue.
- **OneMinusSrcColor**, 1 minus the RGB values of the SrcValue ($1 - R, 1 - G, 1 - B$).
- **OneMinusSrcAlpha**, 1 minus the Alpha of SrcValue ($1 - A, 1 - A, 1 - A$).
- **DstColor** equals the RGB values of the DstValue.
- **DstAlpha** equals the Alpha value of the DstValue.
- **OneMinusDstColor**, 1 minus the RGB values of the DstValue ($1 - R, 1 - G, 1 - B$).
- **OneMinusDstAlpha**, 1 minus the Alpha of the DstValue ($1 - A, 1 - A, 1 - A$).

It is worth mentioning that the Blending of the Alpha channel is carried out in the same way in which the RGB color of a pixel is processed, but it is done in an independent process because it is not used frequently. Likewise, by not performing this process, the writing in the Render Target is optimized.

Render Target is a feature of GPUs which allows a scene to be rendered in an intermediate memory buffer. **Forward Rendering** uses a Render Target by default, while **Deferred Shading** uses multiples of them.

Such a process can be understood as “multiple layers” of a frame of our composition, stored in intermediate buffers.

- Render Target Diffuse. Layer or diffusion buffer.
- Render Target Specular. Specularity layer or buffer.
- Render Target Normals. Normal layer or buffer.
- Render Target Emission. Emission layer or buffer.
- Render Target Depth. Depth layer or buffer.

To illustrate, assume you have an RGB pixel with color values $0.5_R, 0.45_G, 0.35_B$. Such a tint has been processed by the **Fragment Shader Stage**, consequently, it corresponds to the **SrcValue**. Continue by multiplying the previous result by the **SrcFactor One** which equals $1_R, 1_G, 1_B$. Every number multiplied by “one” results in the same value, so, the result between the **SrcFactor** and the **SrcValue** is equal to its initial value.

$$B = [0.5_R, 0.45_G, 0.35_B] [\text{OP}] \text{DstFactor} * \text{DstValue}.$$

OP refers to the operation about to be performed. By default, it is set to **Add**.

$$B = [0.5_R, 0.45_G, 0.35_B] + \text{DstFactor} * \text{DstValue}.$$

Once you have obtained the value of the first operation, the **DstValue** is overwritten, therefore, it is set to the same color $[0.5_R, 0.45_G, 0.35_B]$. So you need to multiply this color by the **DstFactor DstColor**, which is equal to the current **DstFactor** value.

$$\text{DstFactor } [0.5_R, 0.45_G, 0.35_B] * \text{DstValue } [0.5_R, 0.45_G, 0.35_B] = [0.25_R, 0.20_G, 0.12_B].$$

Finally, the output color for the pixel is.

$$\begin{aligned} B &= [0.5_R, 0.45_G, 0.35_B] + [0.25_R, 0.20_G, 0.12_B] \\ B &= [0.75_R, 0.65_G, 0.47_B] \end{aligned}$$

If you want to activate Blending in the shader, you must use the **Blend** command followed by **SrcFactor** and then **DstFactor**.

Its syntax is the following:

```
SubShader Blending
{
    Shader "InspectorPath/ShaderName"
    {
        Properties { ... }
        SubShader
        {
            Tags { "Queue"="Transparent" "RenderType"="Transparent" }
            Blend SrcAlpha OneMinusSrcAlpha
        }
    }
}
```

If you want to use Blending in the shader, it will be necessary to add and modify the **Render Queue**. As you already know, the default value of the Queue Tag is Geometry, which means that the object will appear opaque. If you want the object to look transparent, then you must first change the Queue to Transparent and then add some kind of blending.

The most common types of blending are the following:

➤ Blend SrcAlpha OneMinusSrcAlpha	Common transparent blending
➤ Blend One One	Additive blending color
➤ Blend OneMinusDstColor One	Mild additive blending color
➤ Blend DstColor Zero	Multiplicative blending color
➤ Blend DstColor SrcColor	Multiplicative blending x2
➤ Blend SrcColor One	Blending overlay
➤ Blend OneMinusSrcColor One	Soft light blending
➤ Blend Zero OneMinusSrcColor	Negative color blending

A different way to configure Blending is through the **UnityEngine.Rendering.BlendMode** dependency. This line of code allows you to change object Blending from the Inspector. It can be configured by initializing the property **Toggle Enum** and declaring both the **SrcFactor** and the **DstFactor** later in the Blend.

Its syntax is as follows:

```
SubShader Blending
[Enum(UnityEngine.Rendering.BlendMode)]
    _SrcBlend ("Source Factor", Float) = 1
[Enum(UnityEngine.Rendering.BlendMode)]
    _DstBlend ("Destination Factor", Float) = 1
```

```
SubShader Blending
Shader "InspectorPath/ShaderName"
{
    Properties
    {
        [Enum(UnityEngine.Rendering.BlendMode)]
        _SrcBlend ("SrcFactor", Float) = 1
        [Enum(UnityEngine.Rendering.BlendMode)]
        _DstBlend ("DstFactor", Float) = 1
    }
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
        Blend [_SrcBlend] [_DstBlend]
    }
}
```

Blending options can be written within the **SubShader** field or the **Pass** field. The position will depend on the number of passes and the final result needed.

3.1.8. SubShader AlphaToMask.

There are some types of Blending that are very easy to control, e.g., **SrcAlpha OneMinusSrcAlpha Blend**, which adds a transparent effect with Alpha channel included, but there are occasions where the Blending is not able to generate transparency for the shader. For this case use the **AlphaToMask** property, which applies a covering mask over the Alpha channel, generating an effect similar to discarded pixels.

Unlike Blending, a coverage mask can only assign values of one or zero to the Alpha channel, what does this mean? While Blending has the ability to generate different levels of transparency; levels ranging from 0.0f to 1.0f, AlphaToMask can only generate integers, no decimals. This translates to a harsher type of transparency, which will work in specific cases, e.g., it is very useful for general vegetation such as creating space portal effects.

- **AlphaToMask On**
- **AlphaToMask Off** Default value.

To activate this command, you can declare it in both the SubShader field and the pass field. It only has two values: On and Off, and is declared in the following way:

```
SubShader AlphaToMask
{
    Shader "InspectorPath/ShaderName"
    {
        Properties { ... }
        SubShader
        {
            Tags { "RenderType"="Opaque" }
            AlphaToMask On
        }
    }
}
```

It should be noted that, unlike Blending, in this case, it is not necessary to add Transparency tags or other commands. Just add AlphaToMask and automatically the fourth color channel "A" acquires the qualities of the mask in your program.

3.1.9. SubShader ColorMask.

Supported in both Built-in RP and Scriptable RP, this command allows the GPU to limit itself to writing one or several RGBA color channels when rendering an image.

When a shader is created, by default the GPU writes all the channels corresponding to the color. However, for some cases you may want to display only some of them (e.g. red channel or "R" of an effect).

- ColorMask R Our object will be red
- ColorMask G Our object will be green
- ColorMask B Our object will be blue
- ColorMask A Our object will be affected by transparency.
- ColorMask RG Can use two channel mixing.

As you already know, the acronym RGBA stands for Red, Green, Blue and Alpha, therefore, if you configure the mask with the value "G" you will only show the green channel as color output. This ShaderLab command is very simple to use, and you can use it in both the SubShader and the Pass.

Its syntax is as follows:

```
SubShader ColorMask
  Shader "InspectorPath/ShaderName"
  {
    Properties { ... }
    SubShader
    {
      Tags { "Queue"="Geometry" }
      ColorMask RGB
    }
  }
```

3.2.0. SubShader Culling and Depth Testing.

To understand both concepts, you must first know how **Z-Buffer** (also known as **Depth Buffer**) and **Depth Testing** work.

Before starting, you must consider that pixels have depth values.

These values are stored in the Depth Buffer, which determines if an object goes in front of or behind another on the screen.

On the other hand, Depth Testing is a conditional that determines whether a pixel will be updated or not in the Depth Buffer.

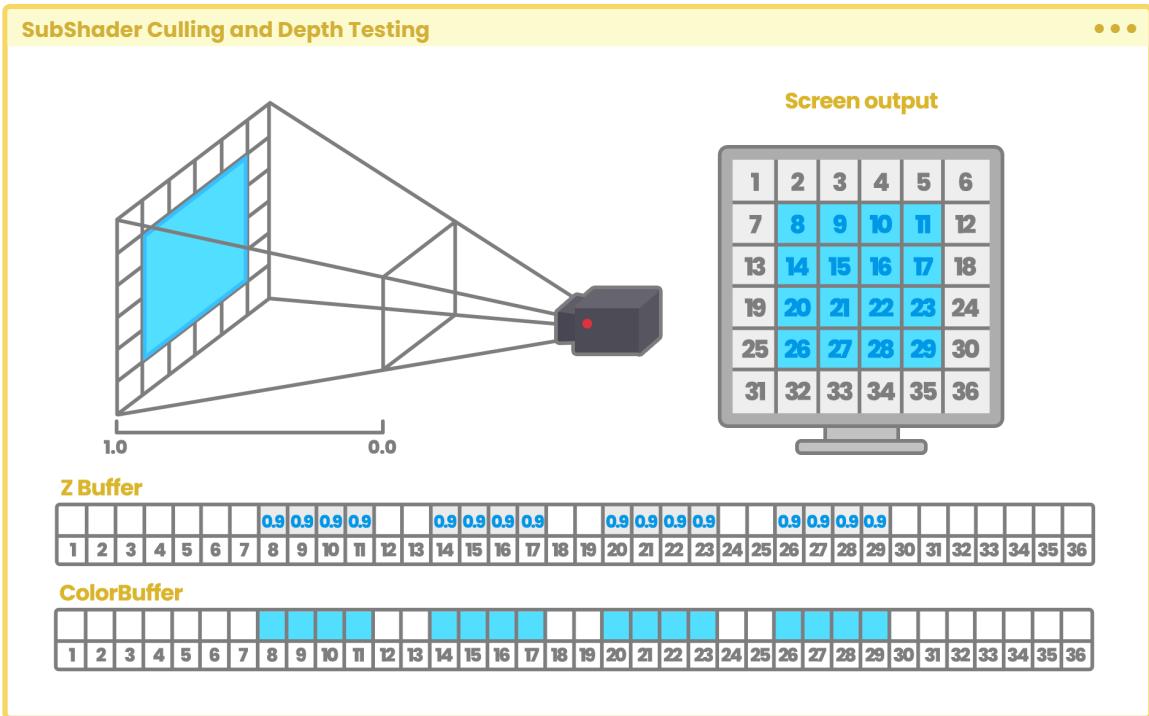
As you already know, a pixel has an assigned value that is measured in RGB color and stored in the **Color Buffer**. The **Z-Buffer** adds an extra value that measures the depth of a pixel in terms of distance to the camera, but only for those surfaces that are within its frustum, this allows two pixels to be the same in color, but different in depth.

The closer the object is to the camera, the lower the Z-Buffer value, and pixels with lower buffer values overwrite pixels with higher values.

To understand the concept, put a camera and some primitives in a scene; all positioned on the Z_{AX} axis. Now, why on the Z_{AX} axis? Such a result is obtained by measuring the distance between the camera and an object on the Z_{AX} space axis, while the values on X_{AX} and Y_{AX} measure the horizontal and vertical displacement on the screen.

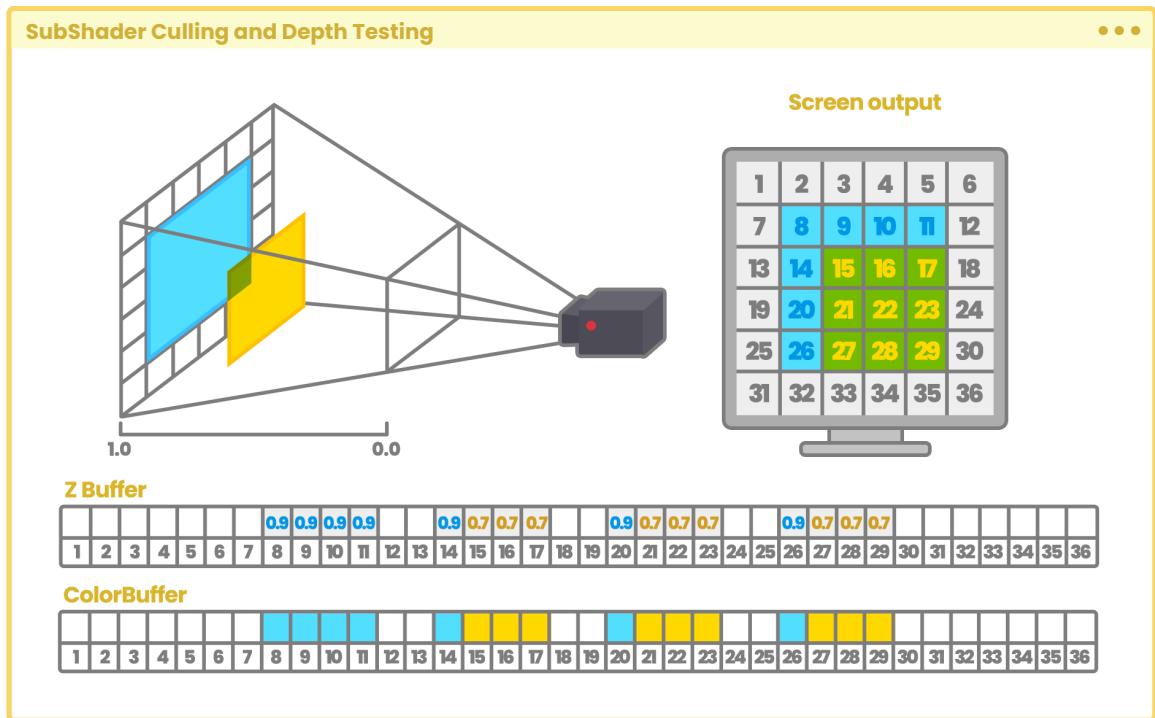
The word “buffer” refers to a **memory space** in which data is stored temporarily, therefore, Z-Buffer corresponds to those depth values between the objects in the scene and the camera, which are assigned to each pixel.

This concept can be illustrated using a screen of 36 total pixels. Place a transparent blue Quad in the scene. Given its nature, it will occupy pixel 8 to 29, therefore, all pixels within this area will light up and be painted blue. Likewise, such information will be sent to both the **Z-Buffer** and the **Color Buffer**.



(Fig. 3.2.0a. The Z-Buffer stores the depth of the object in the scene, and the Color Buffer stores the RGBA color information)

Position a new Quad in the scene, this time in yellow and closer to the camera. To differentiate it from the previous one, you will make this square smaller, occupying pixels 15 to 29. As you can see in Figure 3.2.0a, this area is already occupied by the information from the initial Quad (blue), so what happens here? Since the yellow square is at a shorter distance from the camera, this overwrites the values of both the Z-Buffer and the Color Buffer, activating the pixels in this area, replacing the previous color.



(Fig. 3.2.0b)

You can repeat this process depending on the object and its material. As already mentioned, the painting algorithm will work in a conventional way for transparent materials, while the process will be reversed for opaque ones.

One way to generate attractive visual effects is by modifying the Z-Buffer values. To do this there are three options included in Unity:

- **Cull.**
- **ZWrite.**
- and **ZTest.**

Like Tags, the **Culling** and **Depth Testing** options can be written in different fields: inside the SubShader or the pass. The position will depend on the results wanted and the number of passes you will work with.

To understand this concept, create a shader to represent the surface of a diamond. For this, you will need two passes:

- 1 Use the first for the background color of the diamond.
- 2 And the second for the brightness of the diamond's surface.

In this hypothetical case, since you need two passes to fulfill a different function, it will be necessary to configure the Culling options within each pass, independently.

3.2.1. ShaderLab Cull.

This property, compatible in both Built-in RP and Scriptable RP, controls which face of a polygon will be removed in pixel depth processing. What does this mean? Recall that a polygon object has inner faces and outer ones. By default, the outer faces are visible (Cull Back). However, the inner faces can be activated by following the scheme shown below.

- **Cull Off** Both faces of the object are rendered.
- **Cull Back** The back faces of the object are rendered.
- **Cull Front** The front faces of the object are rendered.

This command has three values, which are: **Back**, **Front** and **Off**. By default, each shader has been configured as **Back**. However, generally the line of code associated with Culling is not visible in the program for optimization reasons. If you want to modify these options, you must add the word Cull followed by the mode you want to apply.

```
ShaderLab Cull
Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader
    {
        // Cull Off
        // Cull Front
        Cull Back
    }
}
```

Culling options can also be dynamically configured in the Unity Inspector through the dependency **UnityEngine.Rendering.CullMode** which is declared from the **Enum Drawer** and passed as an argument in the function.

```
ShaderLab Cull

Shader "InspectorPath/ShaderName"
{
    Properties
    {
        [Enum(UnityEngine.Rendering.CullMode)]
        _Cull ("Cull", Float) = 0
    }
    SubShader
    {
        Cull [_Cull]
    }
}
```

Another very useful option is through the semantic **SV_IsFrontFace**, which allows projection of different colors and textures on both faces of the mesh. To do so, simply declare a boolean variable and assign such semantics as an argument in the **Fragment Shader Stage**.

```
ShaderLab Cull

fixed4 frag (v2f i, bool face : SV_IsFrontFace) : SV_Target
{
    fixed4 colFront = tex2D(_FrontTexture, i.uv);
    fixed4 colBack = tex2D(_BackTexture, i.uv);

    return face ? colFront : colBack;
}
```

Note that this option only works when the **Cull** command has been previously set to **Off** in the shader.

3.2.2. ShaderLab ZWrite.

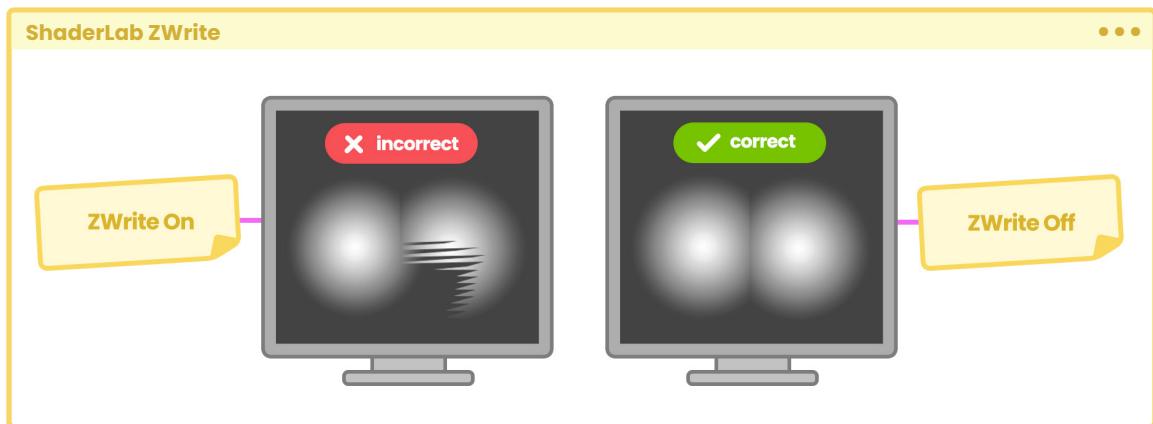
This command controls the writing of the surface pixels of an object to the Z-Buffer, that is, it allows you to ignore or respect the depth distance between the camera and an object. **ZWrite** has two values, which are: On and Off, where the first corresponds to its default value. This command is generally used when working with transparencies, e.g., when you activate the Blending options.

- **ZWrite Off** For transparency.
- **ZWrite On** Default value.

Why should you disable the Z-Buffer when working with transparencies? Mainly because of the translucent pixel overlay (**Z-fighting**). When working with semi-transparent objects, it is common that the GPU does not know which object lies in front of another, producing an overlapping effect between pixels when the camera is moved in the scene. To fix this problem, simply deactivate the Z-Buffer by turning the **ZWrite** command to **Off** as the following example shows:

```
ShaderLab ZWrite
Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off
    }
}
```

The Z-fighting occurs when there are two or more objects the same distance from the camera, causing identical values in the Z-Buffer.



(Fig. 3.2.2a)

This effect occurs when trying to render a pixel at the end of the Render Pipeline. Since the Z-Buffer cannot determine which element is behind the other, it produces flickering lines that change shape depending on the camera's position.

3.2.3. ShaderLab ZTest.

ZTest controls how **Depth Testing** should be performed and is generally used in multi-pass shaders to generate differences in colors and depths. This property has seven different values, which correspond to a comparison operation and are:

- **Less.**
- **Greater.**
- **LEqual.**
- **GEqual.**
- **Equal.**
- **NotEqual.**
- **Always.**

ZTest Less: ($<$) Draws the objects in front. It ignores objects that are at the same distance or behind the shader object.

ZTest Greater: ($>$) Draws the object behind. It does not draw objects that are at the same distance or in front of the shader object.

ZTest LEqual: (\leq) Default value. Draws the objects that are in front of or at the same distance. It does not draw objects behind the shader object.

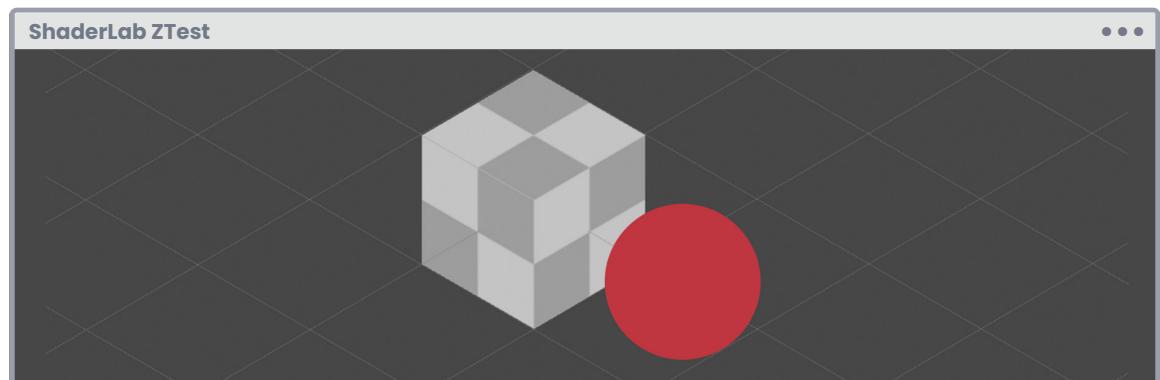
ZTest GEqual: (\geq) Draws the objects behind or at the same distance. Does not draw objects in front of the shader object.

ZTest Equal: ($=$) Draws objects that are at the same distance. Does not draw objects in front of or behind the shader object.

ZTest NotEqual: (\neq) Draws objects that are not at the same distance. Does not draw objects that are the same distance from the shader object.

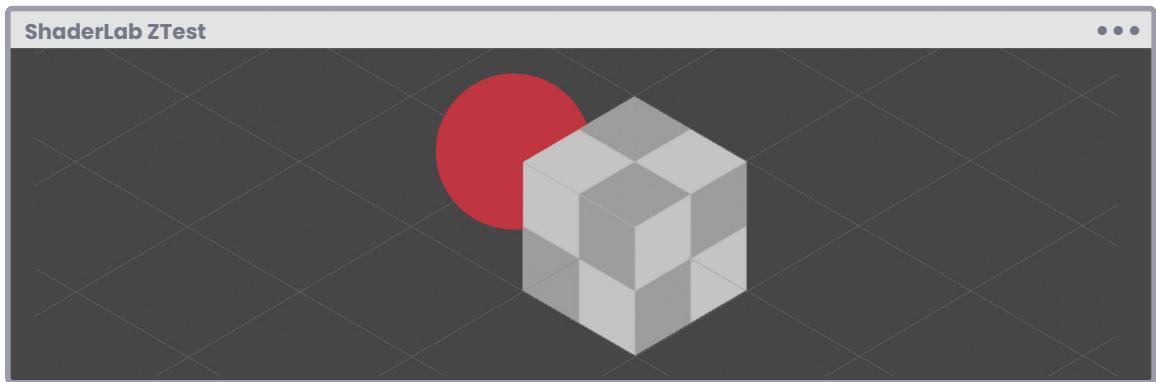
ZTest Always: Draws all pixels, regardless of the distance of the objects relative to the camera.

To understand this command, try the following exercise: Imagine you have two objects in the scene; a Cube and a Sphere. The Sphere is in front of the Cube relative to the camera, and the pixel depth is as expected.



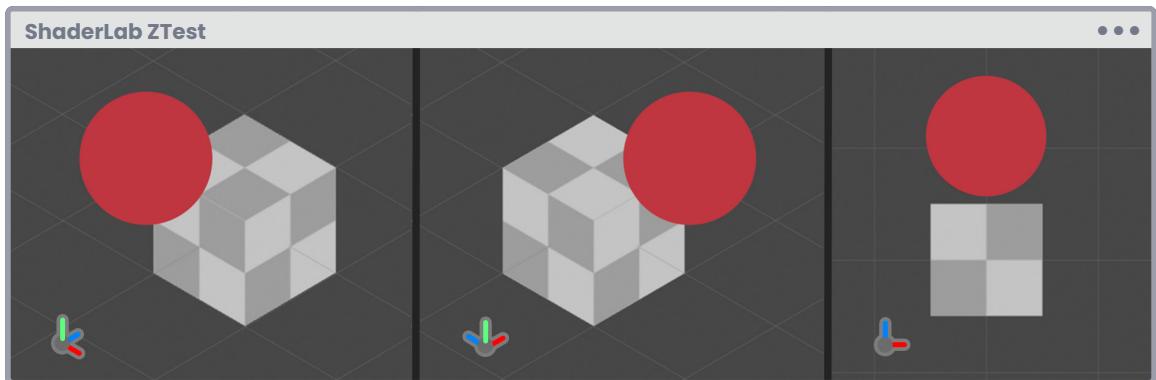
(Fig. 3.2.3a)

If the Sphere is now moved behind the Cube, the depth values will be as expected, why? Because the **Z-Buffer** is storing depth values for each pixel on the screen. The depth values are calculated concerning the proximity of an object to the camera.



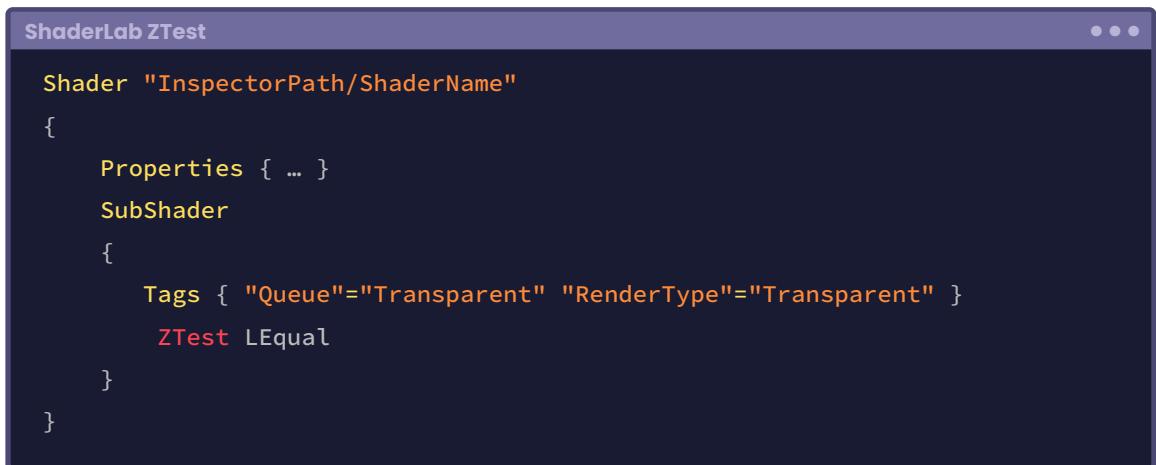
(Fig. 3.2.3b)

Now, what would happen if ZTest Always is activated? In this case, there would be no Depth Testing, therefore, all pixels would appear at the same depth on the screen.



(Fig. 3.2.3c)

Its syntax is the following:

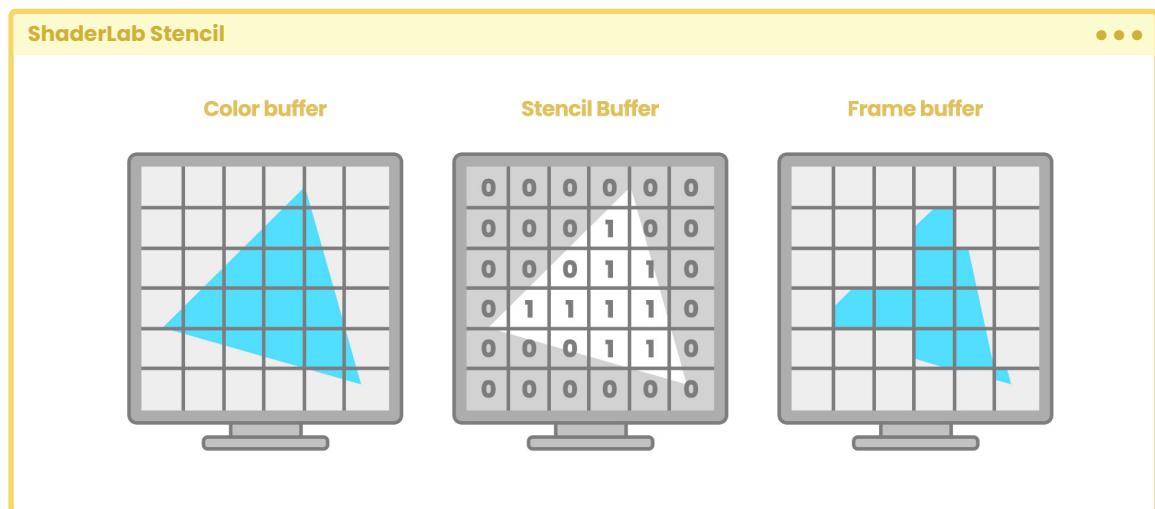


3.2.4. ShaderLab Stencil.

According to the official documentation in Unity:

The Stencil Buffer stores an integer value of eight bits (0 to 255) for each pixel in the Frame Buffer. Before running the Fragment Shader Stage for a given pixel, the GPU can compare the current value in the Stencil Buffer with a determined reference value. This process is called Stencil Test. If the Stencil Test passes, the GPU performs the Depth Test. If the Stencil Test fails, the GPU skips the rest of the processing for that pixel. This means that you can use the Stencil Buffer as a mask to tell the GPU which pixels to draw and which pixels to discard.

Begin by considering that the Stencil Buffer itself is a **texture** that must be created, and for this, an integer value from 0 to 255 is stored for each pixel in the Frame Buffer.



(Fig. 3.2.4a)

As you already know, when objects are positioned in the scene, their information is sent to the **Vertex Shader Stage** (e.g., vertex position). In this stage, the attributes of our object are transformed from Object-Space to World-Space, then View-Space, and finally Clip-Space. This process occurs only for those objects that are within the frustum of the camera.

When this information has been processed correctly, it is sent to the **Rasterizer** which then projects in pixels, the coordinates of your objects in the scene. However, before reaching this point it goes through a previous processing stage called Culling and Depth testing. Various processes occur here that can be manipulated within the shader, among them are:

- **Cull.**
- **ZWrite.**
- **ZTest.**
- and **Stencil.**

Basically, what the **Stencil Buffer** does is activate the **Stencil Test**, which discards **fragments** (pixels) so that they are not processed in the **Fragment Shader Stage**, thus generating a mask effect in the shader.

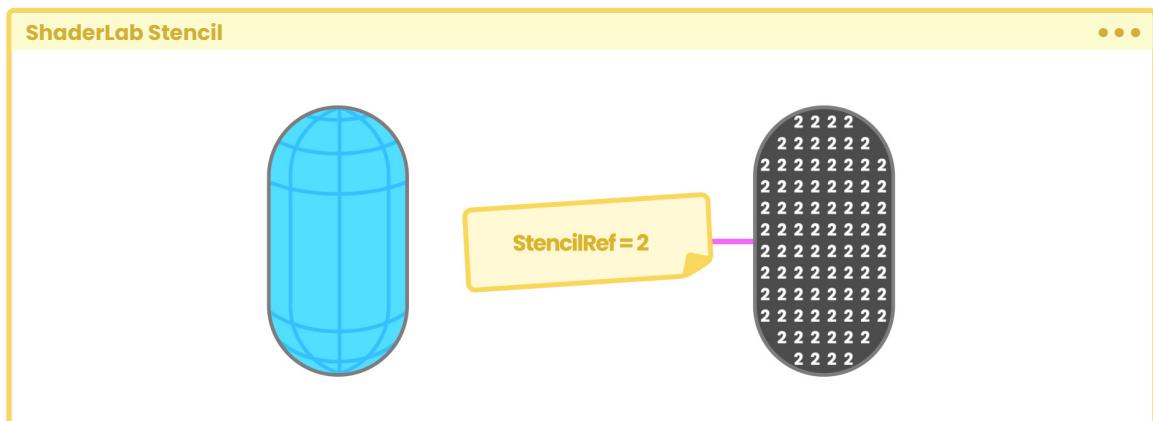
The function performed by the Stencil Test has the following syntax:

```
ShaderLab Stencil

if ( StencilRef & StencilReadMask [Comp] StencilBufferValue &
StencilReadMask)
{
    Accept Pixel.
}
else
{
    Discard Pixel.
}
```

StencilRef is the reference value to be passed to the Stencil Buffer, what does this mean? Remember that the Stencil Buffer is a texture that covers the area in the object's pixels. The StencilRef works as an "id" that maps all the pixels found in the Stencil Buffer.

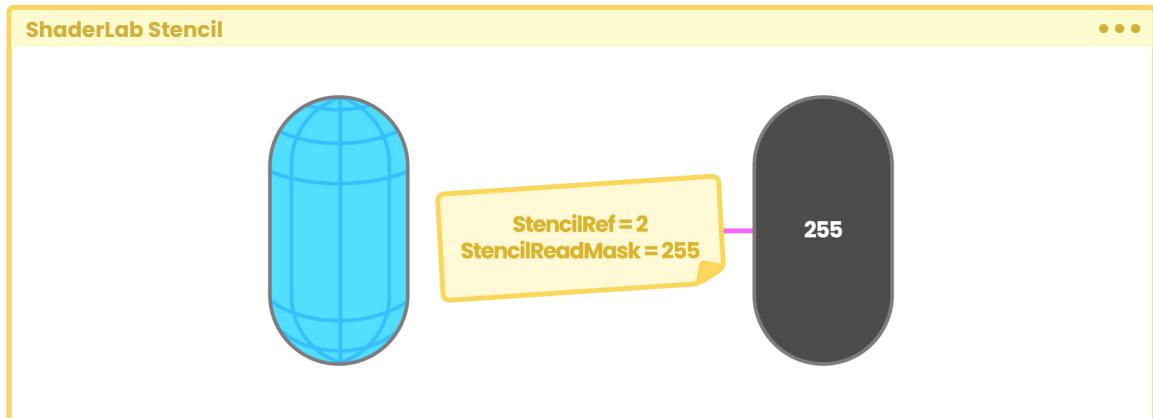
For example, make the StencilRef value 2.



(Fig. 3.2.4b)

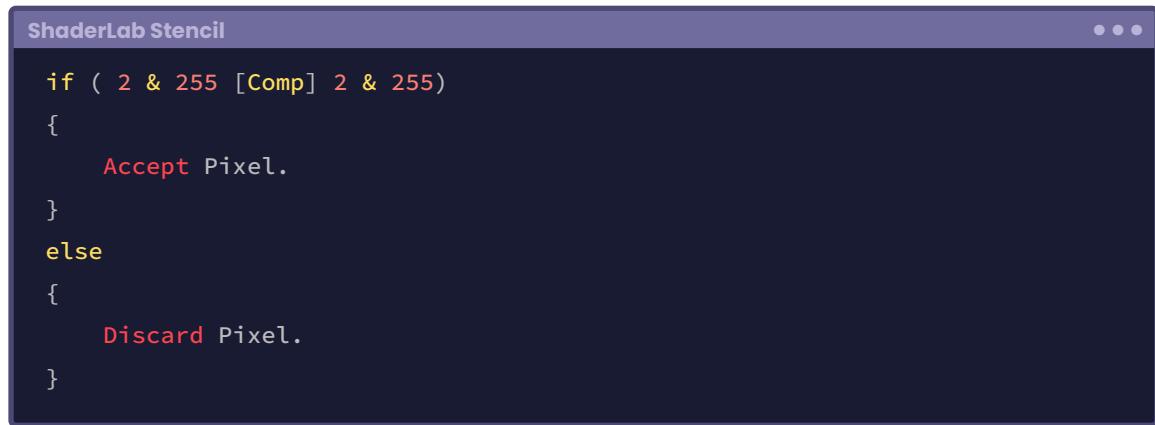
In the example above, all the pixels covering the capsule area have been marked with the value of the StencilRef, therefore now the Stencil Buffer equals 2.

Subsequently, a mask (StencilReadMask) is created for all those pixels that have a reference value that by default has the value 255.



(Fig. 3.2.4c)

Thus, the above operation is as follows.



```
ShaderLab Stencil
if ( 2 & 255 [Comp] 2 & 255)
{
    Accept Pixel.
}
else
{
    Discard Pixel.
}
```

Comp is a comparison function that gives a true or false value. If the value is true, the program writes the pixel in the Frame Buffer, otherwise, the pixel is discarded.

Within the comparison functions, you can find the following predefined operators:

- | | |
|-----------------|---|
| ➤ Comp Never | The operation will always return false. |
| ➤ Comp Less | <. |
| ➤ Comp Equal | ==. |
| ➤ Comp LEqual | ≤. |
| ➤ Comp Greater | >. |
| ➤ Comp NotEqual | !=. |
| ➤ Comp GEqual | ≥. |
| ➤ Comp Always | The operation will always return true. |

At least two shaders are needed to use the Stencil Buffer: One for the mask and one for the masked object.

Suppose you have three objects in the scene:

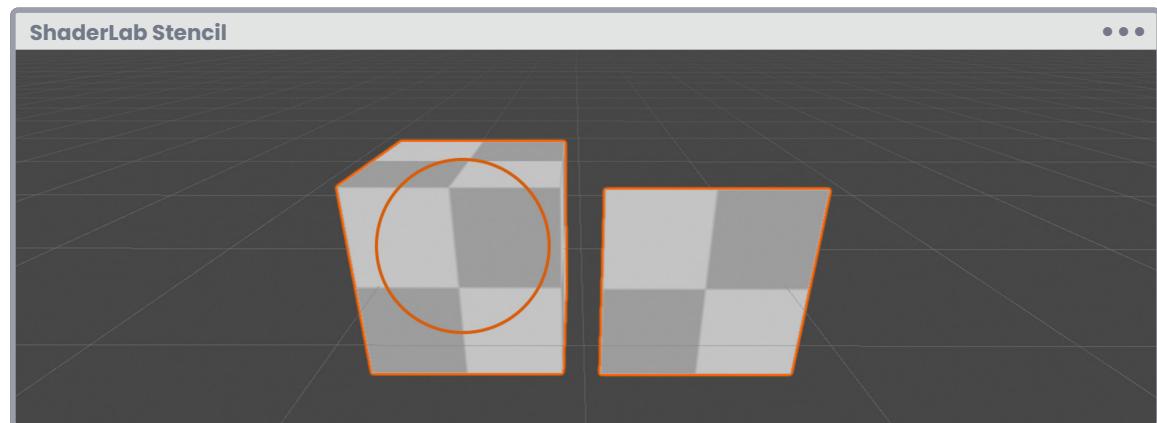
- A **Cube**.
- A **Sphere**.
- And a **Quad**.

The Sphere is inside the Cube, and you want to use the Quad as a “mask” to hide the Cube so that the Sphere inside can be seen.

Continue by creating a shader called **USB_stencil_ref** with the following syntax:

```
ShaderLab Stencil
SubShader
{
    Tags { "Queue"="Geometry-1" }
    ZWrite Off
    ColorMask 0

    Stencil
    {
        Ref 2 // StencilRef
        Comp Always
        Pass Replace
    }
}
```



(Fig. 3.2.4d)

In analyzing the above, the first thing you did was configure **Queue** equals **Geometry minus one**. Geometry defaults to 2000, therefore, Geometry minus 1 equals 1999, this will allow you to process the Quad (mask), to which this shader will be applied, first in the Z-Buffer. However, as you know, Unity by default processes objects according to their position in the scene in respect of the camera and painting algorithm, therefore if you want to disable this function you must set the property **ZWrite** to **Off**.

Then set **ColorMask** to zero so that the mask pixels are discarded in the Frame Buffer and appear transparent.

At this point, the Quad still does not work as a mask, therefore what you must do next is add the command **Stencil** so that it functions as such.

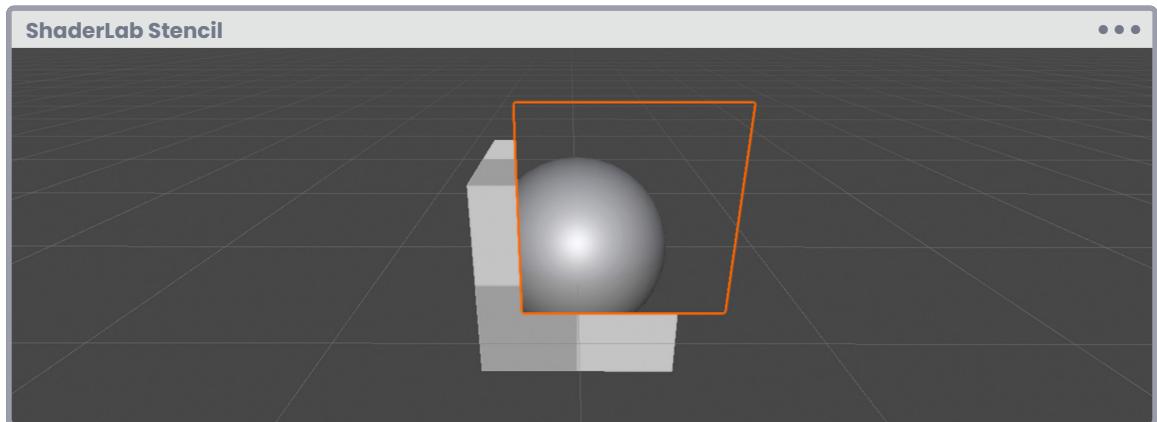
Ref 2 (StencilRef), your reference value, is compared in the GPU to the current content of the Stencil Buffer using the defined comparison operation.

Comp Always makes sure to set a 2 in the Stencil Buffer, taking into account the area covered by the Quad on the screen. Finally, **Pass Replace** specifies that the current values of the Stencil Buffer be replaced by the StencilRef values.

Next you will put emphasis on the object you want to mask. To do this, create a new shader called **USB_stencil_value**.

Its syntax is as follows:

```
ShaderLab Stencil
SubShader
{
    Tags { "Queue"="Geometry" }
    Stencil
    {
        Ref 2
        Comp NotEqual
        Pass Keep
    }
    ...
}
```



(Fig. 3.2.4e)

Unlike the previous shader, keep the Z-Buffer active so that it is rendered on the GPU according to its position relative to the camera. Then add the Stencil command so that your object can be masked. Add "Ref 2" again to link this shader to **USB_stencil_ref**.

Then, in the comparison operation, assign **Comp NotEqual**. This will allow the area of the Cube, which will be exposed around the Quad, to be rendered because the Stencil Test passes (no comparison or true).

On the other hand, if the Quad area is Equal, the Stencil Test will not pass and the pixels will be discarded. **Pass Keep** means that the object maintains the current content of the Stencil Buffer.

If you want to work with more than one mask, you pass a Ref property number different to "2."

3.2.5. ShaderLab Pass.

In this opportunity you will analyze the passes. There can be many of them within a shader. However, Unity by default, adds only “one” inside the SubShader field.

If you pay attention to **USB_simple_color**, created earlier, you will find the following pass included.

```
ShaderLab Pass • • •

Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    // make fog work
    #pragma multi_compile_fog

    #include "UnityCG.cginc"
    struct appdata { ... };

    struct v2f { ... };

    sampler2D _MainTex;
    float4 _MainTex;

    v2f vert (appdata v) { ... }

    fixed4 frag (v2f i) : SV_Target { ... }
}
```

A pass literally refers to a **Render Pass**. For those who have worked with rendering in 3D software, e.g., Maya or Blender, this concept will be easier to understand, since when a composition is being processed, different layers or passes can be generated separately.

Each Pass renders one object at a time, that is, if there are two passes in the shader, the object will be rendered twice in the GPU and the equivalent of that would be two Draw Calls. That's why you should use as few passes as possible, otherwise you could generate a significant graphic load.

```
Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        Pass
        {
            // first default pass
        }
        Pass
        {
            // second additional pass
        }
    }
}
```

3.2.6. CGPROGRAM / ENDCG.

All the sections reviewed above are written in the ShaderLab declarative language, the real challenge in the graphics programming language starts here with **CGPROGRAM** or **HLSLPROGRAM** declaration.

By default, Unity has included the word CG in the Program. Regarding this, the official documentation (version 2021.2) says that:

Unity originally used the Cg language, hence the name of some words and .cginc extensions, but the software no longer uses this language. However, the words are still included, and the code is still compiling for compatibility reasons.

As mentioned above, it is likely that in future versions, shaders will appear with HLSLPROGRAM declaration instead of CGPROGRAM since HLSL is currently the official graphics programming language (in fact, Shader Graph is based on it).

Anyway, you can update the shader simply by replacing the word CGPROGRAM for HLSLPROGRAM and ENDCG for ENDHLSL, and then the program will compile both in Built-in RP as in Universal RP and High Definition RP.

If you want to update the program to HLSL, you will have to change some settings and add some routes to the shader, which will make this more complex to understand.

```
CGPROGRAM / ENDCG

// CG version
Pass
{
    CGPROGRAM
    ...
    ENDCG
}

// HLSL version
Pass
{
    HLSLPROGRAM
    ...
    ENDHLSL
}
```

All the necessary functions for the shader to compile are written inside the CGPROGRAM and ENDCG field. Those functions outside this fragment will be taken by Unity as ShaderLab properties.

```
CGPROGRAM / ENDCG

Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        Pass
        {
            CGPROGRAM
            // write all functions here
            ENDCG
        }
    }
}
```

3.2.7. Data types.

Data types must be looked at before continuing to define the shader properties and functions, because there is a small difference between Cg and HLSL.

When a default shader is created in current versions of Unity, there can find floating-point numbers that differ in precision, among them:

- **Float.**
- **Half.**
- **Fixed.**

A shader written in Cg can compile perfectly in these three types of precision. However, HLSL is not capable of compiling **Fixed** data type, so, if you work with this language, you will have to replace all the variables and vectors of this type either with **Half** or **Float**.

Float is a 32-bit high-precision data type and is generally used in calculating positions in World-Space, UV coordinates, or scalar calculations involving complex functions such as trigonometry or exponentiation.

Half is 16-bit medium-precision and is mostly used in the calculation of low magnitude vectors, directions, Object-Space positions, and high dynamic range colors.

Fixed is 11 bit low precision, and is mainly used in the calculation of simple operations (e.g. basic color storage).

A question that commonly arises in vector use is, what would happen if you only used floating data type (Float) for all your variables? In practice this is possible. However, you must consider that float is a high-precision data type, which means that it has more decimals, therefore, the GPU will take longer to calculate it, increasing times and generating heat.

It is essential to use vectors and/or variables in their required data type, thus you can optimize the program, reducing the graphic load on the GPU.

Other widely used data types that are found in both languages are:

- **Int.**
- **Sampler2D.**
- And **SamplerCube.**

Sampler refers to the sampling state of a texture. This type of data can store a texture with its UV coordinates.

Generally, when working with textures in the shader, you must use the **Texture2D** type to store the texture and create a **SamplerState** to sample. The **Sampler** data type allows you to store both the texture and the sampling status in a single variable.

To understand in detail the function of a sampler, do the following:

Data types

```
// declare the _MainTex texture as a global variable
Texture2D _MainTex;
// declare the _MainTex sampler as a global variable
SamplerState sampler_MainTex;
// go to the fragment shader
half4 frag(v2f i) : SV_Target
{
    // inside the col vector sample the texture in UV coordinates.
    half4 col = _MainTex.Sample(sampler_MainTex, i.uv);
    // return the color of the texture.
    return col;
}
```

The above process can be optimized by simply using a **Sampler2D**.

Data types

```
// declare the sampler for _MainTex
sampler2D _MainTex;
// go to the fragment shader stage
half4 frag(v2f i) : SV_Target
{
    // sampler the texture in UV coordinates using the function tex2D().
    half4 col = tex2D(_MainTex, i.uv)
    // return the color of the texture.
    return col;
}
```

Both examples return the same value that corresponds to the texture that you assigned from the Unity Inspector.

You can use **scalar** values, **vectors**, and **matrices** in the program, which are differentiated by the number of dimensions they have.

Scalar values are those that return a real number, either an integer or decimal. To declare them in the shader you must first add the **data type**, then their **name** and finally initialize their **value**.

Its syntax is as follows:

Data types

```
float name = n; // e. g. float a = 0;
half name = n; // e. g. half b = 1.456;
fixed name = n; // e. g. fixed c = 2.5;
```

Vectors return a value with more than one dimension e.g., XYZW, and to declare them in the shader you must first add the **data type**, then the **dimension quantity**, then their **name** and finally initialize their **value**.

Its syntax is as follows:

Data types

```
float2 name = n; // e. g. float2 a = float2(0.5, 0.5);
half3 name = n; // e. g. half3 b = float3(0, 0, 0);
fixed4 name = n; // e. g. fixed4 c = float4(1, 1, 1, 1);
```

Finally, **matrices** have values stored in **rows** and **columns**, they have more than one dimension and are used mainly for shearing, rotation, and changing vertex position.

To declare a matrix in the shader, add the **data type**, then the **dimension quantity multiplied by itself**, then the **matrix name** and finally **initialize its default values** taking into consideration its identity X_x , Y_y & Z_z :

Its syntax is the following:

```
Data types
• • •

// three rows and three columns
float3x3 name = float3x3
(
    1, 0, 0,
    0, 1, 0,
    0, 0, 1
);

// two rows and two columns
half2x2 name = half2x2
(
    1, 0,
    0, 1
);

// four rows and four columns
fixed4x4 name = fixed4x4
(
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 0
);
```

If you are starting in the world of shaders, you may well not fully understand what has been explained right away. Don't worry, all these parameters will be reviewed in detail later in this book and there will be practical exercises to help see how they work.

3.2.8. Cg / HLSL Pragmas.

The shader contains at least three pragmas included by default. These correspond to processor directives and are included in Cg or HLSL. Their function is to help the program recognize and compile certain functions, which otherwise may not be recognized as such.

The **#pragma vertex vert** allows the **Vertex Shader Stage** called **vert** compile on the GPU as such. Without this line of code, the GPU would not recognize its nature, consequently, all the object information, including the transformation to screen coordinates, would be disabled.

If you look at the pass included in the **USB_simple_color** shader, you will find the following line of code related to the concept previously explained.

```
Cg / HLSL Pragmas
...
// allow to compile the "vert" function as a Vertex Shader
#pragma vertex vert

// use "vert" as vertex shader
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o, o.vertex);
    return o;
}
```

The **#pragma fragment frag** directive does the same function as the pragma vertex, with the difference that it allows the **Fragment Shader Stage** called **frag** to compile in the program.

Cg / HLSL Pragmas



```
// allow to compile the "frag" function as a fragment shader.
#pragma fragment frag

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    UNITY_APPLY_FOG(i.fogCoords, col);
    return col;
}
```

Unlike previous directives, the **#pragma multi_compile_fog** has a double function. Firstly, **multi_compile** refers to a **Shader Variant** that generates variants with different functionalities within the shader. Secondly, the word **_fog** enables the fog functionalities from the **Lighting** window in Unity, this means that, if you go to that tab, Environment / Other Setting, you can activate or deactivate the shader's fog options.

3.2.9. Cg / HLSL Include.

The directive **.cginc** (Cg include) contains several files that can be used in your shader to bring in predefined variables and auxiliary functions.

If you look again at **USB_simple_color**, you will find the following directives declared within the pass:

- **UNITY_FOG_COORDS(T_{RG}).**
- **UnityObjectToClipPos(V_{RG}).**
- **TRANSFORM_TEX(T_{RG}, S_{RG}).**
- **UNITY_TRANSFER_FOG(O_{RG}, C_{RG}).**
- **UNITY_APPLY_FOG(I_{RG}, O_{RG}).**

All these functions belong to **UnityCG.cginc**. If you remove this directive, the shader will not be able to compile because it will not have references to these.

Another defined function found in UnityCG.cginc is **UNITY_PI**, which mathematically equals 3.14159265359f. The latter is not included in the default shader because it is only used in specific cases, e.g., when calculating a triangle or sphere. In case you want to review the variables and functions that come in the “.cginc” files, follow the following path:

Windows → {unity install path} / Data / CGIncludes / UnityCG.cginc

Mac → / Applications / Unity / Unity.app / Contents / CGIncludes / UnityCG.cginc

In addition to the Unity default directives, you can create directives using the extension “.cginc,” to do this simply create a new document, save it with this extension and then start defining the variables and functions.

3.3.0. Cg / HLSL Vertex Input & Vertex Output.

A data type used frequently in the creation of shaders is **Struct**. For those who know C, C#, or C++ a struct is a compound data type declaration, which defines a grouped list of multiple elements of the same type and allows access to different variables through a single pointer. Structs will be used to define both **Inputs** and **Outputs** in the shader.

Its syntax is as follows:

```
Cg / HLSL Vertex Input & Vertex Output
...
struct name
{
    vector[n] name : SEMANTIC[n];
};
```

Inside the Struct field declare those vectors that will be used later between the Vertex Shader Stage and Fragment Shader Stage as a form of connection. The semantics have information that you will use from the object, i.e. UV coordinates, vertices, Normals, tangents and color.

By default, Unity adds two struct functions which are:

- **Appdata.**
- And **v2f.**

The **appdata** (application data) structure corresponds to the Vertex Input, i.e., the place where the object properties are stored, while “**v2f**” (vertex to fragment) refers to the Vertex Output, which allows you to pass properties from the Vertex Shader Stage to the Fragment Shader Stage.

Think of semantics as “access properties” of an object.

According to Microsoft’s official documentation:

A semantic is a string connected to a shader input or output that conveys information about the intended use of a parameter.

What does the above paragraph refer to? Using the POSITION[n] semantics will illustrate this.

The properties of a primitive were mentioned in section 1.0.1. A semantics allows access to such properties individually, i.e., if you declare a four-dimensional vector and pass the semantics POSITION[n] then that vector will contain its vertex position.

Do the following exercise:

Cg / HLSL Vertex Input & Vertex Output

```
struct appdata
{
    float4 pos : POSITION;
```

If you pay attention to the POSITION semantics, you will notice that the position of the object vertices are being stored in a four-dimensional vector called **pos**. Since the process is taking place in the **Vertex Input appdata**, it is assumed that the vertices are in Object-Space.

The most common semantics that you will use during the development of effects correspond to:

- POSITION[n].
- TEXCOORD[n].
- TANGENT[n].
- NORMAL[n].
- COLOR[n].

```
Cg / HLSL Vertex Input & Vertex Output
struct vertexInput // (e.g. appdata)
{
    float4 vertPos : POSITION;
    float2 texCoord : TEXCOORD0;
    float3 normal : NORMAL0;
    float3 tangent : TANGENT0;
    float3 vertColor: COLOR0;
};

struct vertexOutput // (e.g. v2f)
{
    float4 vertPos : SV_POSITION;
    float2 texCoord : TEXCOORD0;
    float3 tangentWorld : TEXCOORD1;
    float3 binormalWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 vertColor: COLOR0;
};
```

TEXCOORD[n] allows access to the primitive UV coordinates and has up to four dimensions (XYZW).

TANGENT[n] gives access to the primitive tangents. If you want to create Normal Maps, it will be necessary to work with a semantic that has up to four dimensions (XYZW).

Through **NORMAL[n]** you can access the primitive Normals, and it has up to four dimensions (XYZW). You must use this semantic if you want to work with lighting within the shader.

Finally, **COLOR[n]** allows access to the primitive vertex color and has up to four dimensions (RGBA). Generally, the default vertex color is white or gray.

To understand this concept, look at the structures that have been declared automatically within **USB_simple_color**, starting with **Vertex Input**.

Cg / HLSL Vertex Input & Vertex Output

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
};
```

On the one hand, “vertex” has the semantic POSITION, meaning that inside the vector the object’s vertices position is being stored in Object-Space. These vertices are then transformed to Clip-Space (screen coordinates) in the **Vertex Shader Stage** through the **UnityObjectToClipPos(V_{RG})** function.

On the other hand, the vector “uv” has the semantics TEXCOORD0, which gives access to the UV coordinates.

Why does the vertex vector have four dimensions? This is due to its coordinate W which is equal to one in this case, since XYZ mark a position in space.

If you look at both **v2f** and **appdata**, you will notice that in both cases they include the semantics POSITION, with the difference that the first one has the prefix **SV** which means **System Value**.

Cg / HLSL Vertex Input & Vertex Output

```
struct v2f
{
    float2 uv : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float4 vertex : SV_POSITION;
};
```

Note that these vectors are being connected in the vertex shader stage as follows:

Cg / HLSL Vertex Input & Vertex Output

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    ...
}
```

In the above example, the pointer **`o.vertex`** is equal to the **Vertex Output**, i.e. the “vertex” vector that has been declared in the **`v2f`** structure. On the other hand, **`v.vertex`** is equal to the **Vertex Input**, that is, to the “vertex” vector that was declared in the **`appdata`** structure. The same logic holds true for the **`uv`** vectors.

3.3.1. Cg / HLSL Variables and Connection Vectors.

Continuing with the **USB_simple_color** shader, notice that in the CGPROGRAM field there is a global variable of **sampler2D** type and a four-dimensional vector referring directly to the **_MainTex** property.

Cg / HLSL Variables and Connection Vectors

```
sampler2D _MainTex;
float4 _MainTex_ST;
```

The global variables are used to connect the values or parameters of the properties with the program variables or internal vectors. In the case of **_MainTex**, a texture can be assigned from the Unity Inspector and used as a texture in the shader.

To understand this concept better, assume that you want to create a shader that can change color from the Inspector. To do this, you would have to go to **properties**, create a **Color** type and then generate a connection variable within the CGPROGRAM field. This generates a link between them.

Cg / HLSL Variables and Connection Vectors



```

Properties
{
    // first declare the property
    _Color ("Tint", Color) = (1, 1, 1, 1)
}

...
CGPROGRAM
...
// then declare the connection variable
sampler2D _MainTex;
float4 _Color;
...
ENDCG

```

Generally, the global variables in Cg or HLSL are declared with the word **uniform**, e.g., **uniform float4 _Color**. However, Unity omits this step because this declaration is included internally in the shader.

3.3.2. Cg / HLSL Vertex Shader Stage.

This is a Render Pipeline programmable stage, where the vertices are transformed from a 3D space to a two-dimensional projection onto the screen. Its smallest unit of calculation corresponds to an independent vertex.

Inside the **USB_simple_color shader**, there is a function called **vert** which corresponds to the **Vertex Shader Stage**. This can be concluded due to the declaration in the pragma vertex.

Cg / HLSL Vertex Shader Stage



```

#pragma vertex vert
...
v2f vert (appdata v)
{
    ...
}

```

Before continuing remember that the shader **USB_simple_color** is an **Unlit** type, meaning it does not have light. That's why it includes a function for the **Vertex Shader** and another for the **Fragment Shader**.

In the opposite way, uniquely in Built-in RP, Unity provides a quick way to write shaders in the form of **Surface Shader**, this automatically generates Cg code, exclusively for materials that are affected by light. It allows optimization of development time, including internal program functions and calculations. However, due to its structure, it is not ideal for the explanation of concepts, which is why this book started with the creation of an Unlit shader; to understand its operation in detail.

When analyzing the structure of the Vertex Shader Stage, the function begins with the word **v2f** which means “vertex to fragment.” This name makes a lot of sense when you understand the internal process. Basically, V2f will be used later as an argument in the **Fragment Shader Stage**, hence its name.

Cg / HLSL Vertex Shader Stage

```
v2f vert (appdata v) { ... }
```

In this field the variables that have been declared in **appdata** or **v2f** can be transformed and connected. In fact, you will notice that the Vertex Output has been initialized with the letter “o” which refers to all the v2f internal variables.

Cg / HLSL Vertex Shader Stage

```
v2f o;
o.vertex ...
o.uv ...
return o;
```

Remember that the scene’s objects are within a three-dimensional space, while the screen is two-dimensional. Because of this, these coordinates need to be transformed. So the first operation that occurs in the Vertex Shader Stage is the transformation of the object vertices from Object-Space to Clip-Space through the function **UnityObjectToClipPos(v_{RG})**, which multiplies the current model matrix (`unity_ObjectToWorld`) by the factor of the multiplication between the view and the projection matrix (`UNITY_MATRIX_VP`).

Cg / HLSL Vertex Shader Stage

```
UnityObjectToClipPos(float3 pos).
{
    return mul(
        UNITY_MATRIX_VP,
        mul(unity_ObjectToWorld, float4(pos, 1.0)));
}
```

This operation has been declared in the **UnityShaderUtilities.cginc** file, which has been included as a dependency in **UnityCG.cginc**.

Cg / HLSL Vertex Shader Stage

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
}
```

A factor to consider is the number of dimensions that must contain variables when working with inputs or outputs, e.g., note that in **appdata** the vector **float4 vertex** has the same number of dimensions as its counterpart in the Vertex Output. If two vectors differ in dimension number when being connected, it is possible that Unity will return a transformation error.

Later you can find the **TRANSFORM_TEX** function, which asks for two arguments:

- 1 The input of the object UV coordinates (**v.uv**).
- 2 And the texture to be positioned over these coordinates (**_MainTex**). Basically, it fulfills the function of controlling the “tiling” and “offset” in the texture UV coordinates.

Finally, these values are stored in the UV output (**o.uv**) to be used later in the Fragment Shader Stage.

3.3.3. Cg / HLSL Fragment Shader Stage.

The next and last function in the pass corresponds to the **Fragment Shader Stage** which by default appears as **frag** in the program. Its purpose is obvious since it has been declared as such in pragma fragment.

Cg / HLSL Fragment Shader Stage

```
#pragma fragment frag

fixed4 frag (v2f i) : SV_Target
{
    // fragment shader functions here
}
```

The word **fragment** refers to a pixel on the screen; to an individual fragment or to a group that together cover an object area. The Fragment Shader Stage will process every pixel on the computer screen in relation to the object being viewed.

In analyzing its structure, you can see that the return will be equal to a four-dimensional vector due to the “**fixed4**” data type.

It is worth noting the Cg language in this case since the vector will only compile in Built-in RP. If you want the shader to work in Scriptable RP you will have to replace **fixed** by **half** or **float**, keeping the number of dimensions.

Cg / HLSL Fragment Shader Stage

```
// Cg language
fixed4 frag (v2f i) : SV_Target { ... }

// HLSL language
half4 frag (v2f i) : SV_Target { ... }
```

Unlike the Vertex Shader Stage, this stage has an output called **SV_Target** which corresponds to the output value that will be stored in the Render target. In previous versions of Direct3D (version 9 and before) the color output in the Fragment Shader appeared with the semantic **COLOR**. However, in modern GPUs (version 10 onwards) this semantic is updated by **SV_Target**, which

means “system value target” and can apply additional effects to the image before projecting them on the computer screen.

By default, inside the field there is a four-dimensional fixed type vector called **col**, which refers to the color of the **_MainTex** texture that is being applied through the **tex2D(S_{RG}, UV_{RG})** function, which in turn, receives as argument the property and the UV coordinates output.

This vector has four dimensions for two main reasons:

- 1 Because the frag function is a four-dimensional vector.
- 2 Because the tex2D function returns different values for each RGBA dimension.

```
Cg / HLSL Fragment Shader Stage
fixed4 frag (v2f i) : SV_Target
{
    // store the texture in the col vector
    fixed4 col = tex2D(_MainTex, i.uv);
    // return the texture color
    return col;
}
```

3.3.4. ShaderLab Fallback.

It is quite common to see magenta objects after a GPU compilation error. Fallback helps to avoid such conflicts by positioning a different shader than the one that generated the error.

Its syntax is as follows:

```
ShaderLab Fallback
Fallback "shaderPath"
```

To do this, simply declare a name and path, so the software will know where to get a new shader from in case of hardware failures.

This command can be omitted, leaving the space blank or declaring Fallback Off. Similarly, it is advisable to use a path and name of shaders that have been included in the software, e.g., **Mobile/Diffuse**. In this way you can be sure that the program will continue to work in case of failures.

```
ShaderLab Fallback
Shader "InspectorPath/ShaderName"
{
    Properties { ... }
    SubShader { ... }

    Fallback "Mobile/Unlit"
}
```

In the previous example, the Fallback will return an **Unlit** type shader belonging to the **Mobile** category in case the SubShader generates an error.

If you are developing a multiplatform game, it is advisable to declare a specific path for the Fallback, this way you can ensure that the program works on most devices.

Implementation and other concepts.

4.0.1. Analogy between a shader and a material.

According to its terminology,

Materials are definitions of how a surface should be rendered, including references to textures, tiling, offset, color and more. The options of a material depend on which shader is being used.

How can the above definition be translated on a practical level? Let's think of a material as "the container of a shader," this means that the program performs the surface calculations (shader) and the container (material) reads these calculations.

A material alone cannot perform any operation. If it doesn't have a shader, it will not know how it should be rendered. Likewise, a shader cannot be applied to an object if it is not through a material, therefore, the analogy between a material and a shader is a "graphical preview of mathematical calculations."

4.0.2. Your first shader in Cg or HLSL.

One of the simplest exercises that can be carried out in a shader is changing a color or tint of a texture. To explain this, you will continue working with **USB_simple_color** since, by default, it has a texture called `_MainTex`, which can be used to change its tint dynamically from the Inspector.

It is worth remembering that a shader cannot be directly applied on an object in a scene, instead, you need the help of a material. It is assumed that the reader of this book already knows the process of creating materials in the software, anyway, here is a little help to address the issue.

To generate a material;

- 1 Go to your project.
- 2 Right click on the folder in which you are working.
- 3 Go to the Create menu and select Material.

The Render Pipeline will determine the default configuration of the material just created. Generally, Built-in RP comes with a **Standard Surface** shader which allows you to visualize the lighting direction, shadows, and other associated calculations.

To preview the changes, navigate to the Hierarchy window in Unity and create a 3D object. You must make sure to assign the previously mentioned shader to the material, and then apply that material to the object in the scene. In the same way, to do this it is essential to apply a texture on the material.

Return to the shader and create a Color property, as shown in the following example:

```
Your first shader in Cg or HLSL
...
Shader "USB/USB_simple_color"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Color ("Texture Color", Color) = (1, 1, 1, 1)
    }
    SubShader { ... }
}
```

In the example, a color property was declared called Texture Color which will be used to modify the Texture tint. If you save and return to Unity you can see that the property appears in the Inspector. However, it has not been initialized inside CGPROGRAM, therefore it is not completely functional.

Next, add the connection variable `_Color` for its use inside the program. To do this, position yourself “over” the Vertex Shader Stage; where `_MainTex` has been declared as `sampler2D`, and create the variable as follows:

```
Your first shader in Cg or HLSL
...
uniform sampler2D _MainTex;
uniform float4 _MainTex_ST;
uniform float4 _Color;           // connection variable.
```

After declaring the global variable in the CGPROGRAM or HLSLPROGRAM, you can now use it in any of the functions in your shader, always considering that the GPU will read the program linearly, from top to bottom. To change `_MainTex` tint, go to the Fragment Shader Stage and perform the following exercise:

Your first shader in Cg or HLSL

```
// CGPROGRAM
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    return col * _Color;
}

// HLSLPROGRAM
half4 frag (v2f i) : SV_Target
{
    half4 col = tex2D(_MainTex, i.uv);
    return col * _Color;
}
```

As you can see, the variable `col` has been multiplied by `_Color`, which, as a result, modifies the `_MainTex` tint.

If you save and return to Unity, you will be able to change the texture tint from the material Inspector.

4.0.3. Adding transparency in Cg or HLSL.

In the previous section, RGBA color was added to the `USB_simple_color` configuration in order to change the tint of the `_MainTex` texture. However, if you modify the color Alpha channel from the Inspector, you will notice that it does not change the texture, why is this? As previously mentioned in section 3.1.7, it is essential to configure the **Blending** type, in addition to the **RenderType** and **Queue** to be transparent.

• • •

Adding transparency in Cg or HLSL

```
Shader "USB/USB_simple_color"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Color ("Texture Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" }
        Blend SrcAlpha OneMinusSrcAlpha
        LOD 100

        Pass { ... }
    }
}
```

If you save your shader and return to Unity, you can now modify the opacity of the texture since it is being affected by the fourth color channel.

4.0.4. Structure of a function in HLSL.

• • •

As in C#, in HLSL you can also declare void functions or those that return a value. Depending on the type of function you have to use “declarations.” These allow you to determine if a value corresponds to an **input** (in), **output** (out), **global variable** (uniform) or a **constant** (const) value.

Structure of a function in HLSL

```
void functionName_precision (declaration type arg)
{
    float value = 0;
    arg = value;
}
```

In the declaration of a void function, start with its nomenclature, then the function **name** along with its **precision** and finally its **arguments**.

Generally, in the arguments you must define whether they will be inputs or outputs, so, how can you know if they have a declaration? Everything will depend on the functions you want to pass as arguments. To understand the concept, create a simple function that is able to calculate the light in an object. For this, it will be necessary to use Normals, since, given their nature, they will allow you to determine the angle between the light source and the surface of the object. Therefore, the Normals would be an “input to calculate” within the function.

```
Structure of a function in HLSL
• • •
void FakeLight_float (in float3 Normal, out float3 Out)
{
    float[n] operation = Normal;
    Out = operation;
}
```

In a void type function you must always add the precision, otherwise it cannot be compiled into your program. It is worth mentioning that the previous operation does not have any functionality, however it will be used to understand the previously mentioned concepts.

FakeLight corresponds to the name of the function as such, and **_float** is its precision. The latter can be **float** or **half** type, since, as you know, these are the data types supported by HLSL.

Then in the arguments, notice that the object's Normals have been declared as input, since they have the declaration **in**. Also, there is an output called **out** which corresponds to the operation's final value.

Next, do the simulation of the FakeLight function inside the Fragment Shader Stage; as if it will really perform an operation.

Structure of a function in HLSL

```
// create function
void FakeLight_float (in float3 Normal, out float3 Out)
{
    float[n] operation = Normal;
    Out = operation;
}
```

Structure of a function in HLSL

```
half4 frag (v2f i) : SV_Target
{
    // declare normals.
    float3 n = i.normal;
    // declare the output.
    float3 col = 0;
    // pass both values as arguments.
    FakeLight_float (n, col);

    return float4(col.rgb, 1);
}
```

There are several situations that are occurring in the example above. First, the **FakeLight** function has been declared before the “frag” function because the GPU reads the code from top to bottom. Then, a three-dimensional vector called “n” and another of the same size called “col” have been generated in the Fragment Shader Stage, this is because you will use both vectors as arguments in the **FakeLight_float** function, which asks for both a three-dimensional vector type input and output. So, the first argument corresponds to the object Normals input and the second, to the result of the operation being performed within the function.

The vector “col” is black by default because all its channels have been initialized to “zero.” However, since it has been included as output it inherits the result being carried out inside in the **FakeLight_float** function.

Finally, the Fragment Shader stage returns a four-dimensional vector, where the first three values correspond to the vector “col” in RGB and one for A.

Why is it returning a four-dimensional vector? This is because the function "frag" is of **half4** type, that is, a half-precision vector with dimensions.

Having defined a void type function, you will now analyze the structure of a return function. In essence they are similar, with the difference that in the latter it will not be necessary to add the precision of the function nor to declare an output argument. To illustrate this use the same analogy through the `FakeLight` function, which, this time, will return a value.

Structure of a function in HLSL

```
// create our function
half3 FakeLight (float3 Normal)
{
    float[n] operation = Normal;
    return operation;
}
```

Structure of a function in HLSL

```
half4 frag (v2f i) : SV_Target
{
    // declare normals
    float3 n = i.normal;
    float3 col = FakeLight_float (n);

    return float4(col.rgb, 1);
}
```

In a return function simply add the argument without declaration since it does not require it. It should be noted that you can assign the result of the function directly to a variable or vector, just as you would in C#.

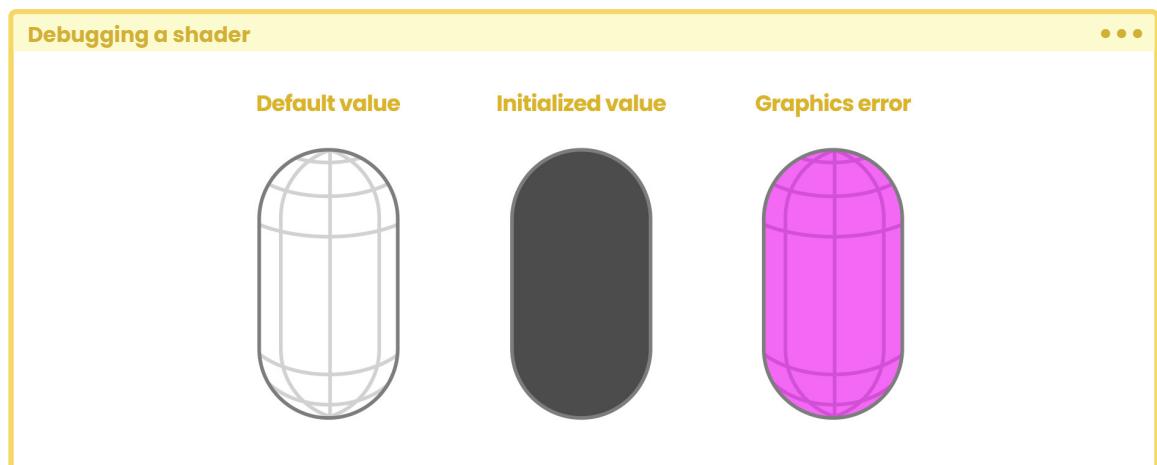
4.0.5. Debugging a shader.

It is very common in C# to use the **Debug.Log(Obj)** function to debug a program. This operation prints a message in the Unity console, making it easier to understand what you are developing. However, given its nature, this function is not available in Cg or HLSL. So, how can you debug a shader? To do this you need to consider some factors, among them, its colors.

In a shader there are three important colors to which you must pay attention, these are:

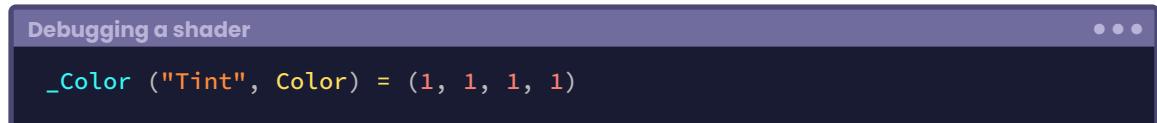
- White ($1_R, 1_G, 1_B, 1_A$).
- Black ($0_R, 0_G, 0_B, 1_A$).
- Magenta ($1_R, 0_G, 1_B, 1_A$).

White represents a default value, black refers to an initialized value, while magenta alludes to a graphical error. In fact, it is very common to import assets into Unity that look magenta in your scene.



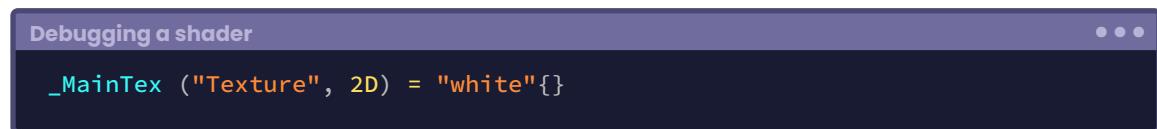
(Fig. 4.0.5a)

To address this concept, remember the Properties' declaration in ShaderLab, start by creating a color property.



In the example above, notice that the property has been declared **white** by default, this can be confirmed in the four-dimensional vector at the end of the operation. So this property as such will generate a “color selector” in the Unity Inspector, and its color and/or initialization value will be white. So, a color selector with the same value will be generated in the Inspector.

Now analyze another property.



As in the previous case, `_MainTex` is going to generate a selector in the Inspector, in this case, of texture, but what color will it be? It will be white, you can corroborate it in the declaration **white** at the end of the property. In the same way, you could use other colors to initialize your property, for example, “red,” “black” or “gray.”

You will see **black** frequently in vector declarations and internal variables in code, in fact it is very common to initialize vectors at “zero” to later add some operation.

It is worth mentioning that **white** represents the maximum pixel illumination value on the screen, while **black** equals the minimum. It is essential to consider this factor, since some operations will exceed such maximum ($x > 1$) and minimum ($x < 0$) values. These cases produce **color saturation**, so it will be necessary to use functions such as “clamp” to limit a value between two numbers; a minimum and a maximum.

As you already know, magenta represents a **graphic error**. Basically, when the GPU has not been able to perform the operations inside the shader, it gives magenta by default. In Unity, the reasons for this are mainly due to two factors:

- 1 That the Render Pipeline has not been configured.
- 2 Or that the shader has an error in its code.

When importing an asset into the software, the first thing you need to know is, what Render Pipeline you are working in. Remember that in Unity there are three types of Render Pipeline and each one has a different configuration.

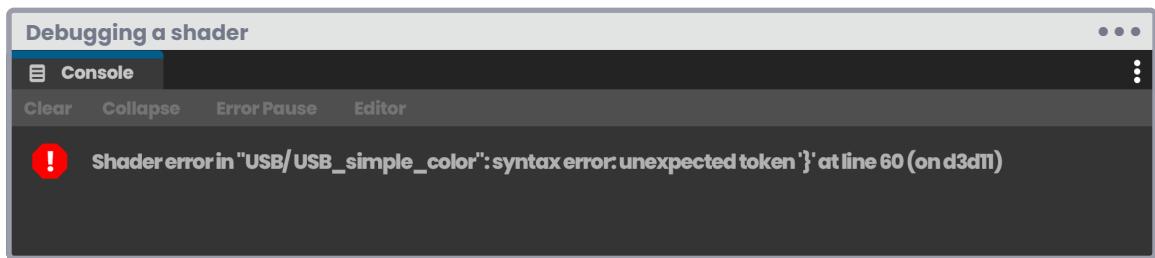
If the asset you import to Universal RP includes a material with a **Standard Surface** shader, given its configuration, it will appear magenta, since, as you know, this type of shader is only supported

in Built-in RP. To solve this graphical error, you have to select the material, go to the Inspector and change the shader type for one that is in the Shader Graphs menu.

Once you have identified the type of Render Pipeline being used you must proceed to the shader evaluation. If you check the console, you can find its definition. Some factors must be considered in solving graphical errors, among them:

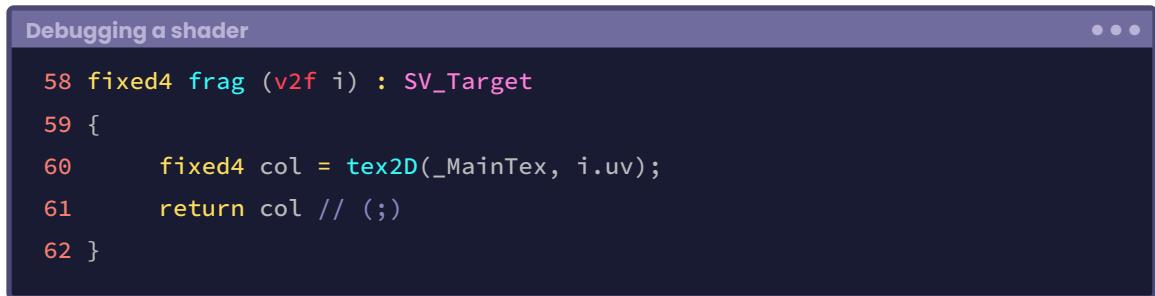
- Instruction or function term in HLSL and ShaderLab.

Most errors are generated by bad wording or syntax, below is a fairly common one.



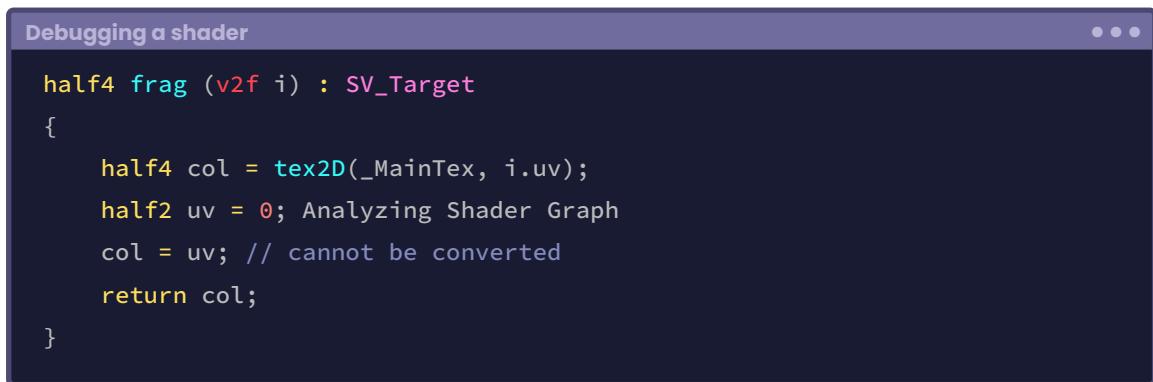
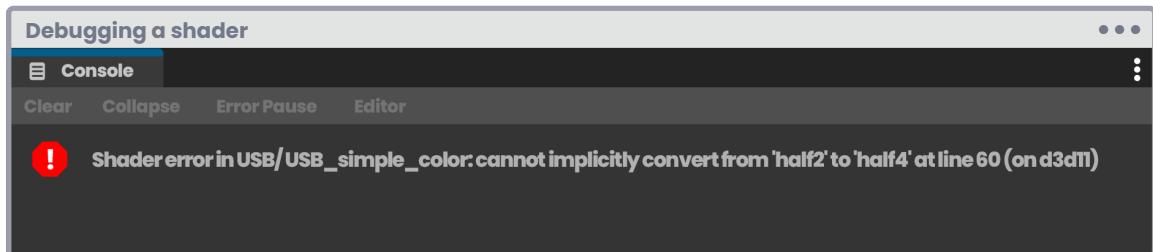
In the above error, the shader found in the **USB** menu, called **USB_simple_color**, has an error in line 60 of code and cannot compile in Direct3D 11 (d3d11).

To verify if that really is the shader problem, analyze the line of code that is generating the error.



According to the console, the error is being generated in code line number 60, but as you can see, there is no issue there. So, what is happening? The error is because the operation in code line number 61 is not closed. If you look at the **return** you will realize that the **col** vector is missing a semicolon, then the GPU thinks that the operation continues, so it can't compile it.

The following is another error that you see frequently:



In this case, the error has been generated because you are trying to convert a four-dimensional vector (col) into a two-dimensional one (uv). The col vector, being four-dimensional, has RGBA or XYZW channels, on the other hand, the "uv" vector, being two-dimensional, only has two channel combinations (RG, RB, GB, etc.), therefore it cannot be converted.

4.0.6. Adding URP compatibility.

Up to this point, most of the variables, functions, and vectors that you have implemented, work both for Cg and HLSL. However, there will be some cases where you will have to add URP compatibility so that your shader can compile.

If you want to create a shader via HLSL in Universal RP, you will have to add some dependencies so that the GPU can read the Render Pipeline correctly. Such dependencies can be found in different paths, including:

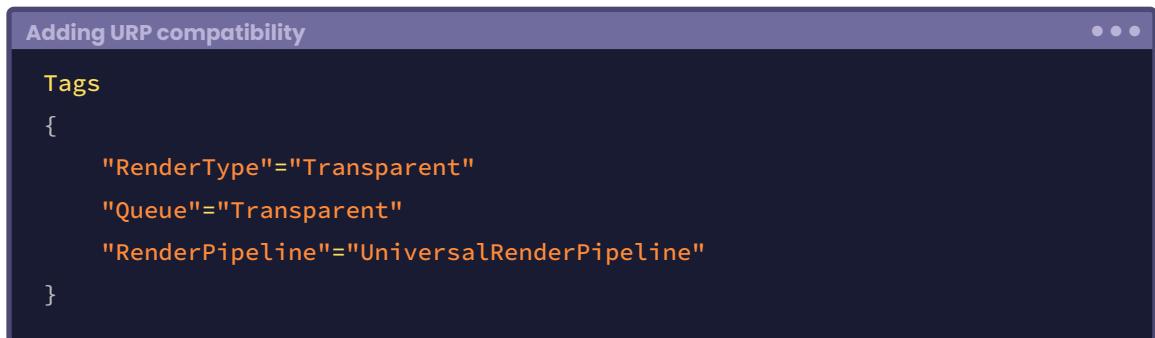
- Packages / Core RP Library / ShaderLibrary
- Packages / Universal RP / ShaderLibrary

The **Core RP Library** package is automatically included when you install **Shader Graph** in your project. Likewise, the **Universal RP** package is included when you choose this Render Pipeline as your rendering engine.

Both packages have files with .hsls extensions which are required by the program to compile shaders in HLSL.

To understand the concept, duplicate the **USB_simple_color** shader and rename it **USB_simple_color_UPR**. It is worth mentioning that the shader **USB_simple_color** (Cg), being a basic color model, already has Universal RP compatibility. However, you will generate a copy of it and modify it to review its implementation in HLSL in detail.

Start by declaring the **RenderPipeline Tags** in your URP shader and make it equal to **UniversalRenderPipeline**, this way the GPU will be able to deduce its compatibility with the Render Pipeline.



The screenshot shows a code editor window with a purple header bar containing three dots on the right. The main area is dark with yellow text. The title bar says "Adding URP compatibility". The code is as follows:

```
Tags
{
    "RenderType"="Transparent"
    "Queue"="Transparent"
    "RenderPipeline"="UniversalRenderPipeline"
}
```

As previously mentioned, Universal RP works with HLSL language, therefore the CGPROGRAM/ENDCG blocks in the pass will have to be replaced by HLSLPROGRAM and ENDHLSL respectively.

Adding URP compatibility

```

Pass
{
    HLSLPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        ...
    ENDHLSL
}

```

For perfect compilation and efficiency, it will be necessary to include the **Core.hsls** dependency, which contains functions, structures, and others that allow the program to work well in Universal RP. In itself, **Core.hsls** replaces **UnityCg.cginc**, therefore, it will be necessary to eliminate the latter from your shader.

Adding URP compatibility

```

HLSLPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    // #pragma multi_compile_fog
    // #include "UnityCg.cginc"

    #include "Package/com.unity.render-pipelines.universal/ShaderLibrary/Core.hsls"
    ...

ENDHLSL

```

Unlike UnityCg.cginc which comes included in the Unity installation, Core.hsls is added as a package once you install Universal RP, therefore, you will have to write its full location path in the project. Once these dependencies are replaced, two things will happen.

- 1 The GPU will not be able to compile the fog coordinates (UNITY_FOG_COORDS, UNITY_TRANSFER_FOR, and UNITY_APPLY_FOR) since these are included in UnityCg.cginc.
- 2 Nor will you be able to compile the **UnityObjectToClipPos(V_{RG})** function for the same reason.

Instead, you must use the **TransformObjectToHClip(v_{RG})** function which is included in the **SpaceTransforms.hlsl** dependency, which in turn, has been included in **Core.hlsl**.

TransformObjectToHClip(v_{RG}) performs the same operation as **UnityObjectToClipPos(v_{RG})**, that is, to transform the position of the vertices from Object-Space to Clip-Space. However, the execution process of the former is more efficient.

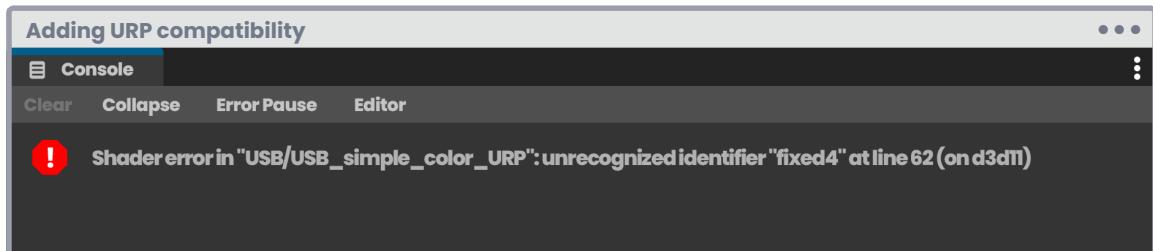
Adding URP compatibility

```
v2f vert (appdata v)
{
    v2f o;
    // o.vertex = UnityObjectToClipPos(v.vertex);
    o.vertex = TransformObjectToHClip(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    return o;
}
```

If at this point save your shader and go back to Unity, you can see that an **unrecognized identifier** error has been generated in the console, why is this? Remember that both the Fragment Shader Stage and the “col” vector, are by default, **fixed4** type. So you can check it in the function.

Adding URP compatibility

```
fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    // UNITY_APPLY_FOG(i.fogcoord, col);
    return col * _Color;
}
```



As you already know, Universal RP cannot compile “fixed” type variables or vectors, instead you will have to use “half or float.” Therefore, at this point, you can do two things.

- 1 Manually replace fixed type variables and vectors with half.
- 2 Or include the **HLSLSupport.cginc** dependency that adds helper macros and cross-platform definitions for shader compilation.

```

Adding URP compatibility

HLSLPROGRAM
#pragma vertex vert
#pragma fragment frag

// #pragma multi_compile_fog
// #include "UnityCg.cginc"

#include "HLSLSupport.cginc"
#include "Package/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"
...
ENDHLSL

```

Once this dependency is included you can work with fixed type variables and vectors. Now your program will recognize this type of data according to its precision and will replace them with either half or float, automatically.

4.0.7. Intrinsic functions.

In both Cg and HLSL, you can find intrinsic functions that will help program effects. These functions correspond to general mathematical operations and are used in specific cases depending on the result you wish to obtain. Among the most common you can find:

- | | | |
|---|--|--|
| <ul style="list-style-type: none"> ➤ Abs. ➤ Ceil. ➤ Clamp. ➤ Cos. ➤ Sin. ➤ Tan. | <ul style="list-style-type: none"> ➤ Exp. ➤ Exp2. ➤ Floor. ➤ Step. ➤ Smoothstep. ➤ Frac. | <ul style="list-style-type: none"> ➤ Length. ➤ Lerp. ➤ Min. ➤ Max. ➤ Pow. |
|---|--|--|

Which are defined below.

4.0.8. Abs function.

This function refers to the absolute value of a number, and as an argument, we can pass both a scalar value and a vector.

Its syntax is as follows:

```
Abs function
...
// return absolute value of n
float abs(float n)
{
    return max(-n, n);
}

float2 abs (float2 n);
float3 abs (float3 n);
float4 abs (float4 n);
```

An absolute value will always return to a positive number, and its mathematical symbology consists of two sidebars that frame a number.

$$|-3| = 3$$

absolute value of -3 equals 3

$$|-5| = 5$$

absolute value of -5 equals 5

$$|+6| = 6$$

absolute value of 6 equals itself

In your program you could use the **`abs(NRG)`** function for multiple effects, including recreating a kaleidoscope or generating triplanar projection. In fact, for the first case, you could perform such an effect by calculating the absolute value in the UV coordinates, while you could determine the absolute value of the mesh's Normals in triplanar projection to generate projections on both positive and negative axes.



(Fig. 4.0.8a. As you can see in section 1.0.5, UV coordinates start at 0.0f and end at 1.0f. In the previous figure, you can see the behavior of the U coordinate when you subtract 0.5f. The texture has been set in clamp from the Inspector)

If you pay attention to the starting point in the UV coordinates of Figure 4.0.8a above, you will notice that the U coordinate is centered on the Quad by subtracting 0.5f, and the minimum value becomes -0.5f. In the example, the texels get stretched because the texture has been configured in **Clamp** from the **Wrap Mode**. In this context, you could apply the absolute value to generate a mirror effect.



(Fig. 4.0.8b. The absolute value of U coordinate subtracting 0.5f)

Next, you will develop a kaleidoscope effect to fully understand the concept. Start by creating a new Unlit shader, called **USB_function_ABS**. Then, in its Properties declare a numeric range that you will use later to rotate the UV coordinates.

```
Abs function
Shader "USB/USB_function_ABS"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        // add a property to rotate the UV
        _Rotation ("Rotation", Range(0, 360)) = 0
    }
}
```

Since a complete rotation is 360 degrees, **_Rotation** is equal to a range between 0 and 360. Now continue with the global or connection variable for the property.

```
Abs function

Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Rotation;
    ...
    ENDCG
}
```

Both the U and V coordinates' absolute values can be calculated to generate the effect and use these new values for the output color in the Fragment Shader Stage.

```
Abs function

fixed4 frag (v2f i) : SV_Target
{
    // calculate the absolute value of U
    float u = abs(i.uv.x - 0.5);
    // calculate the absolute value of V
    float v = abs(i.uv.y - 0.5);

    fixed col = tex2D(_MainTex, float2(u, v));
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

Two main things could happen depending on the texture configuration you are assigning as `_MainTex`.

- 1 If the texture is set as **Repeat**, the negative area of the UV coordinates will be filled with a repetition of itself.
- 2 On the other hand, if the texture configuration corresponds to **Clamp**, the image texels will be stretched in the same way as in Figure 4.0.8a.

Regardless of the configuration, the mirror effect will be evident since each coordinate's "abs" function is used.

If you want to rotate the UV coordinates, you can use the **Unity_Rotate_Degrees_float** function, included in Shader Graph.

```
Abs function
void Unity_Rotate_Degrees_float
(
    float2 UV,
    float2 Center,
    float Rotation,
    out float2 Out
)
{
    Rotation = Rotation * (UNITY_PI/180.0f);
    UV -= Center;
    float s = sin(Rotation);
    float c = cos(Rotation);
    float2x2 rMatrix = float2x2(c, -s, s, c);
    rMatrix *= 0.5;
    rMatrix += 0.5;
    rMatrix = rMatrix * 2 - 1;
    UV.xy = mul(UV.yx, rMatrix);
    UV += Center;
    Out = UV;
}
```

Since it is a **void** function you will have to initialize some variables within the field of the Fragment Shader Stage and then pass them as arguments.

Abs function

```
Unity_Rotate_Degrees_float() { ... }

fixed4 frag (v2f i) : SV_Target
{
    float u = abs(i.uv.x - 0.5);
    float v = abs(i.uv.y - 0.5);
    // link the rotation property
    float rotation = _Rotation;
    // center the rotation pivot
    float center = 0.5;
    // generate new UV coordinates for the texture
    float2 uv = 0;

    Unity_Rotate_Degrees_float(float2(u,v), center, rotation, uv);
    fixed4 col = tex2D(_MainTex, uv);
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

The first argument in the function **Unity_Rotate_Degrees_float** corresponds to the UV coordinates that you want to rotate, next the center of rotation or pivot, then the number of degrees, and finally the output with the new coordinate values that will be used for the texture.

4.0.9. Ceil function.

According to NVIDIA's official documentation,

*Ceil returns the smallest integer
not less than its argument.*

What does this mean? The function **ceil(N_{RG})** will return an integer, that is, without decimals, closest to its argument, e.g., if the number equals 0.5f, ceil will return one.

```
ceil (0.1) = 1
ceil (0.3) = 1
ceil (1.7) = 2
ceil (1.3) = 2
```

All numbers between 0.0f and 1.0f will return one, since the latter would be the smallest integer value no less than its argument.

its syntax is the following:

Ceil function

```
// return an integer value
float ceil(float n)
{
    return -floor(-n);
}

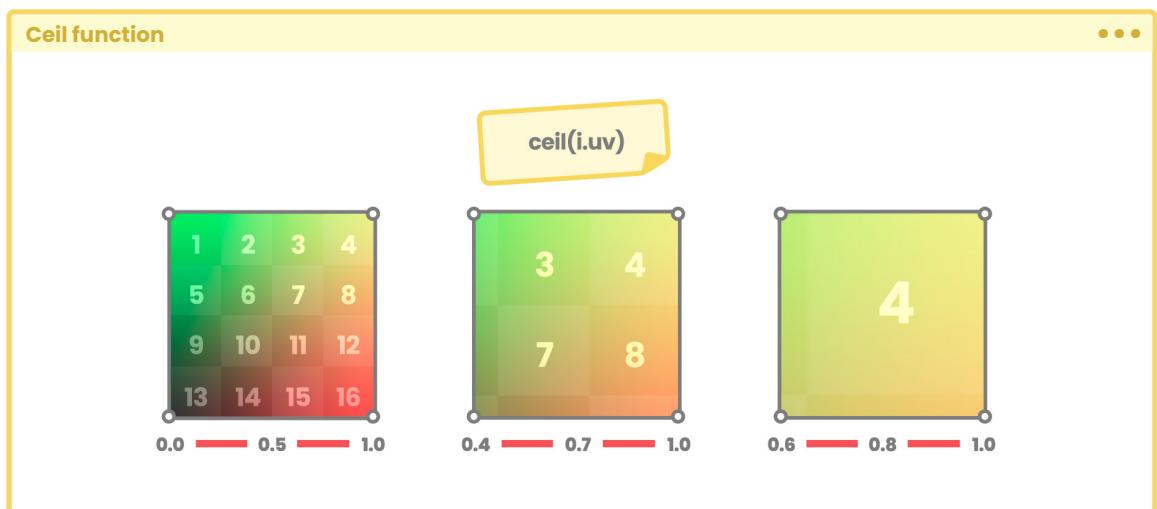
float2 ceil (float2 n);
float3 ceil (float3 n);
float4 ceil (float4 n);
```

This function is quite useful to generate video game zoom or magnifying glass effects. To do this simply calculate the value of **ceil(N_{RG})** for both the U and V coordinates, and multiply the result by 0.5f. Then, generate a linear interpolation between the default values of the UV coordinates and the resulting values of the function.

To understand the concept in depth, do the following: Create a new Unlit shader called **USB_function_CEIL**, and start by declaring the new UV coordinates for the _MainTex texture in the Fragment Shader Stage.

```
Cell function
fixed4 frag (v2f i) : SV_Target
{
    // ceil the U coordinate
    float u = ceil(i.uv.x);
    // ceil the V coordinate
    float v = ceil(i.uv.y);
    // assign the new values for the texture
    fixed4 col = tex2D(_MainTex, float2(u, v));
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

The operation performed above will return the texel found in position $1.0_U, 1.0_V$, why? Both the U and V coordinates start at 0.0f and end at 1.0f, so the function will only return the last texel found in the texture, and consequently, generate a zoom effect that will go from the upper right point of the texture to the lower left point.



(Fig. 4.0.9a. *Ceil is going to return the last texel found in the texture; which is in position $1.0_U, 1.0_V$*)

For this effect, a property is needed to increase or decrease the size of the texture. To do this, go to the Properties and declare a floating range and call it `_Zoom`.

Ceil function

```
Shader "USB/USB_function_CEIL"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Zoom ("Zoom", Range(0, 1)) = 0
    }
}
```

In the range, “0” represents zero percent zoom, and “1” one hundred percent. Then declare the connection variable within the program.

Ceil function

```
Pass
{
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Zoom;
    ...
}
```

Since you need the zoom point to start in the center of the texture, modify its position by multiplying the operation by 0.5f as follows:

Ceil function

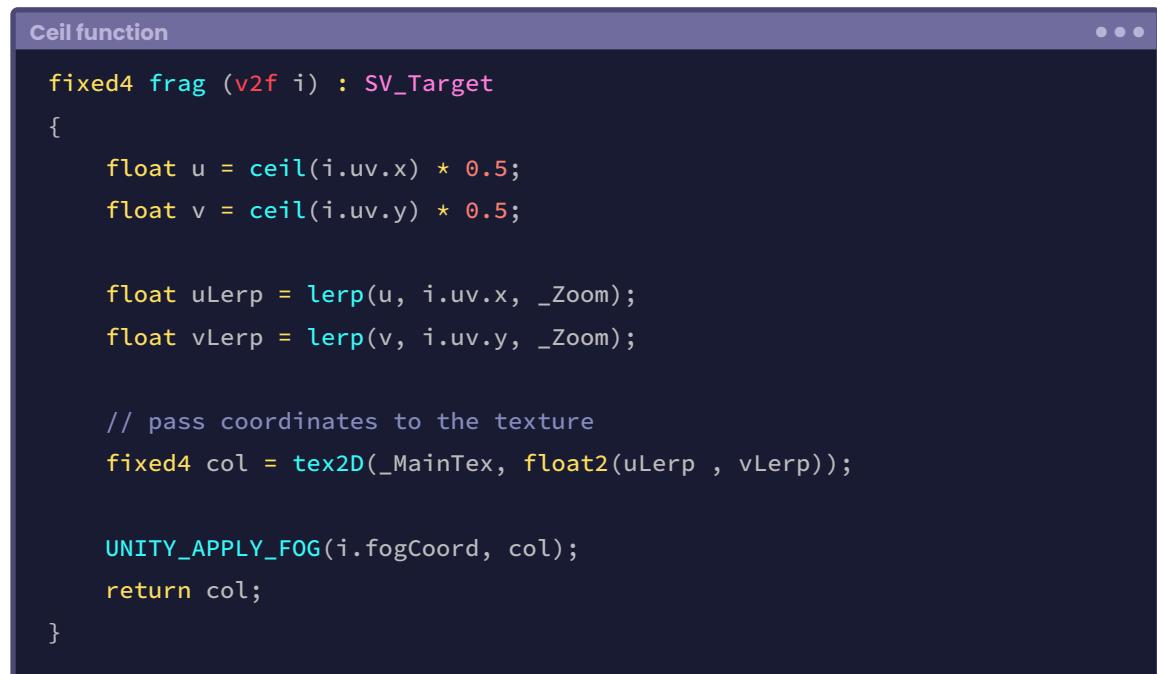
```
fixed4 frag (v2f i) : SV_Target
{
    // declare U coordinate
    float u = ceil(i.uv.x) * 0.5;
    // declare V coordinate
    float v = ceil(i.uv.y) * 0.5;
```

Continued on the next page.

```
// pass coordinates to the texture
fixed4 col = tex2D(_MainTex, float2(u, v));

UNITY_APPLY_FOG(i.fogCoord, col);
return col;
}
```

At this point, the texture is already expanding from its center. However, you still cannot see the zoom effect since the $\text{ceil}(N_{RG})$ function is still returning “one,” so you will just see a single fill color in the Quad area. What you need to do is generate a **linear interpolation** between the default UV coordinate values and those resulting from the $\text{ceil}(N_{RG})$ function.



The screenshot shows a code editor window with a purple header bar containing three dots on the right. The main area contains the following HLSL code:

```
Ceil function
fixed4 frag (v2f i) : SV_Target
{
    float u = ceil(i.uv.x) * 0.5;
    float v = ceil(i.uv.y) * 0.5;

    float uLerp = lerp(u, i.uv.x, _Zoom);
    float vLerp = lerp(v, i.uv.y, _Zoom);

    // pass coordinates to the texture
    fixed4 col = tex2D(_MainTex, float2(uLerp , vLerp));

    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

In the example above, two variables were created called **uLerp** and **vLerp** for the different coordinates being used. With these you are doing a linear interpolation through the “lerp” function belonging to Cg/HLSL. In addition, the property **_Zoom** was included, which has a range between 0.0f and 1.0f. If you modify the values of the property **_Zoom** from the Inspector, you can see how the texture increases or decreases its size, taking as a reference its central point.



(Fig. 4.0.9b. Ceil will return the texel found in the center of the texture)

4.1.0. Clamp function.

This function is very useful when you want to limit the result of an operation. By default, it allows you to define a value within a numerical range, setting a minimum and a maximum.

As you develop functions, you will encounter some operations that result in a number less than "zero" or greater than "one," e.g., in the calculation of the dot product between mesh Normals and illumination direction, you can get a range between -1.0f and 1.0f. Since a negative value would generate color artifacts in the final effect, with `clamp(ARG, XRG, BRG)` you can limit and redefine the range.

Its syntax is the following:

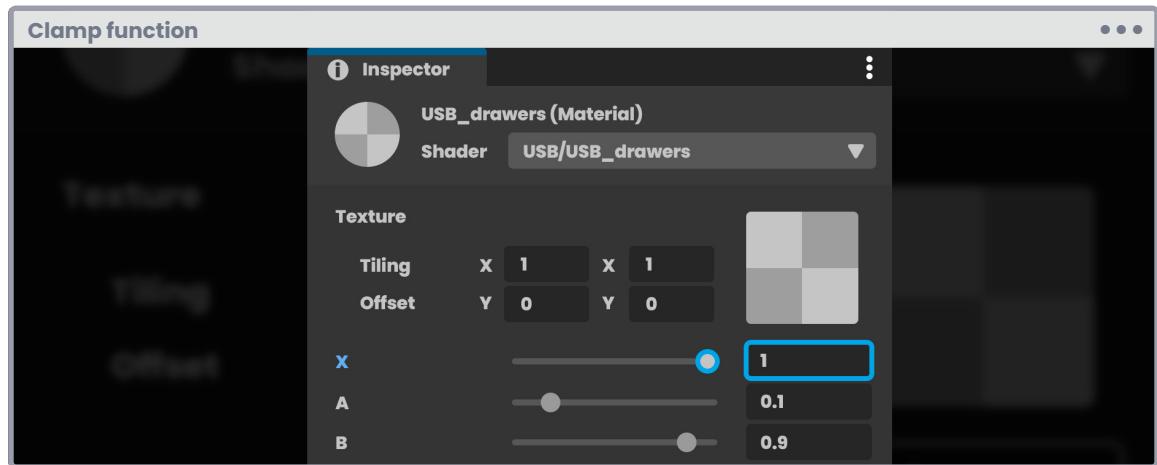
Clamp function

```
float clamp (float a, float x, float b)
{
    return max(a, min(x, b));
}

float2 clamp (float2 a, float2 x, float2 b);
float3 clamp (float3 a, float3 x, float3 b);
float4 clamp (float4 a, float4 x, float4 b);
```

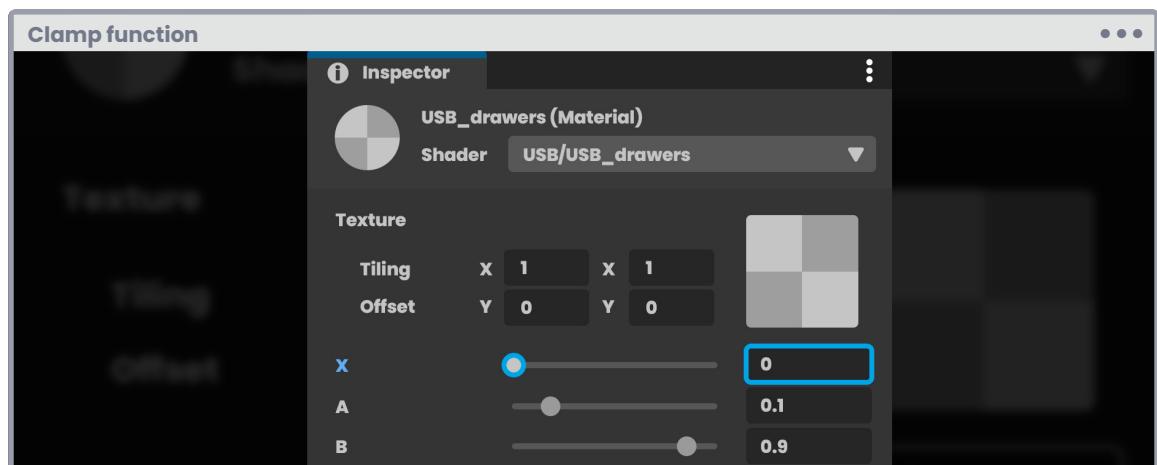
In the above function, the A_{RG} argument refers to the minimum return value, while B_{RG} refers to the maximum return value within the range. The argument $X_{RG'}$ corresponds to the value you want to limit according to A_{RG} and $B_{RG'}$, what does this mean? To see, create a range set as minimum and maximum, and a variable number for $X_{RG'}$:

When X_{RG} equals 1.0f; if the argument A_{RG} equals 0.1f and B_{RG} equals 0.9f, the maximum return value for $X_{RG'}$ will be equal to B_{RG} , why? Because the latter defines the highest limit value for the operation.



(Fig. 4.1.0a)

The same happens in the opposite case. When X_{RG} equals 0.0f, the return value will be equal to A_{RG} , because this defines the minimum return value.



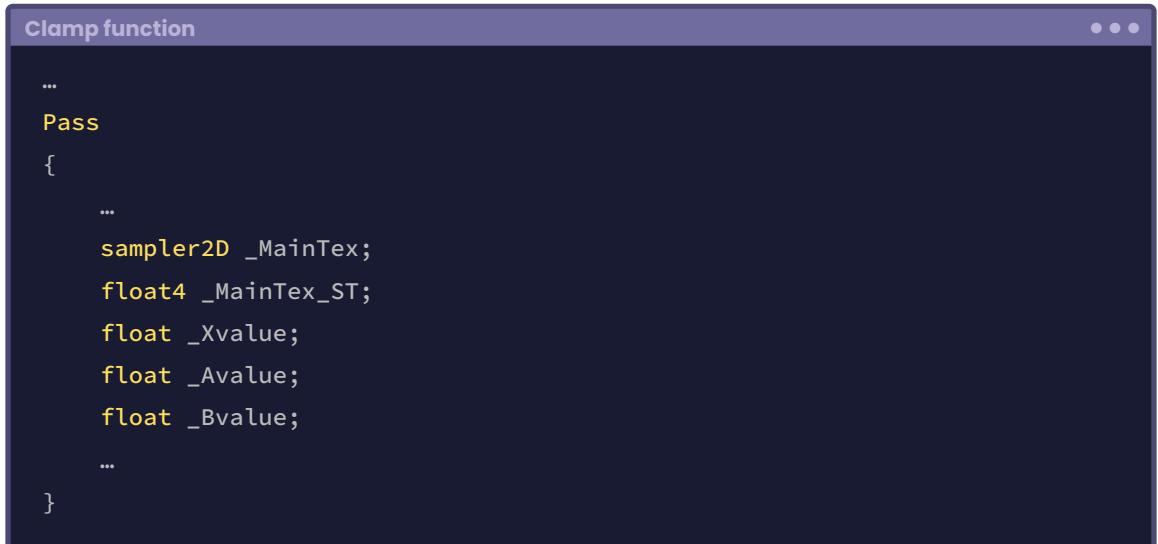
(Fig. 4.1.0b)

To understand this concept in-depth, do the following: First, create a new Unlit shader and call it **USB_function_CLAMP**, and in its Properties, declare three floating ranges; one for each argument.



```
Shader "USB/USB_function_CLAMP"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Xvalue ("X", Range(0, 1)) = 0
        _Avalue ("A", Range(0, 1)) = 0
        _Bvalue ("B", Range(0, 1)) = 0
    }
}
```

Then declare the global or connection variables to connect them to the program.



```
...
Pass
{
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Xvalue;
    float _Avalue;
    float _Bvalue;
    ...
}
```

You will use the shader you have just created to increase or decrease the gamma color of the main texture. Therefore, at this point, you can do two things:

- 1 Generate a simple function within your program that limits a value within a range.
- 2 Or use the included function **clamp** that is in Cg/HLSL.

Start by declaring a new function and call it **ourClamp**. To do this, go between the Vertex Shader and the Fragment Shader Stage.

Clamp function

```
v2f vert(appdata v) { ... }

float ourClamp(float a, float x, float b)
{
    return max(a, min(x, b));
}

fixed4 frag(v2f i) : SV_Target { ... }
```

Then, in the Fragment Shader Stage, create a floating variable called **darkness** and make it equal to your new function. As arguments, pass the properties declared above, following the same syntax order.

Clamp function

```
float ourClamp(float a, float x, float b) { ... }

fixed4 frag(v2f i) : SV_Target
{
    float darkness = ourClamp(_Avalue, _Xvalue, _Bvalue);
    fixed4 col = tex2D(_MainTex, i.uv);

    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

The **darkness** variable is of floating/scalar type, meaning that it has only one dimension. Therefore, if you multiply the “col vector” by the mentioned variable, the color RGBA channels will be affected.

Clamp function

```
float ourClamp(float a, float x, float b) { ... }

fixed4 frag(v2f i) : SV_Target
{
    float darkness = ourClamp(_Avalue, _Xvalue, _Bvalue);
    fixed4 col = tex2D(_MainTex, i.uv) * darkness;

    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

You can simplify the above operation by including the **clamp** function in the language you are applying.

Clamp function

```
// float ourClamp(float a, float x, float b) { ... }

fixed4 frag(v2f i) : SV_Target
{
    float darkness = clamp(_Avalue, _Xvalue, _Bvalue);
    fixed4 col = tex2D(_MainTex, i.uv) * darkness;

    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

In conclusion, you can now define a maximum and minimum gamma for the output color in the shader.

4.1.1. Sin and Cos functions.

These trigonometric functions refer to the sine and cosine of an angle, that is:

- The ratio between the adjacent leg and the hypotenuse, in the case of cosine.
- And the ratio between the opposite leg and the hypotenuse, in the case of sine.

Its syntax is as follows:

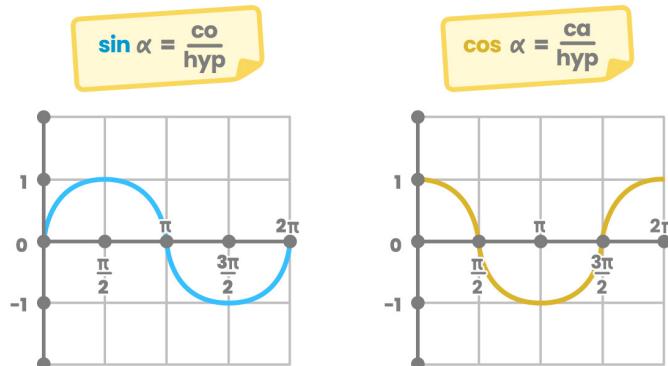
Sin and Cos functions

```
float cos (float n);
float2 cos (float2 n);
float3 cos (float3 n);
float4 cos (float4 n);

float sin (float n);
float2 sin (float2 n);
float3 sin (float3 n);
float4 sin (float4 n);
```

You can use both **$\cos(\mathbf{N}_{RG})$** and **$\sin(\mathbf{N}_{RG})$** on scalar values and vectors.

Sin and Cos functions



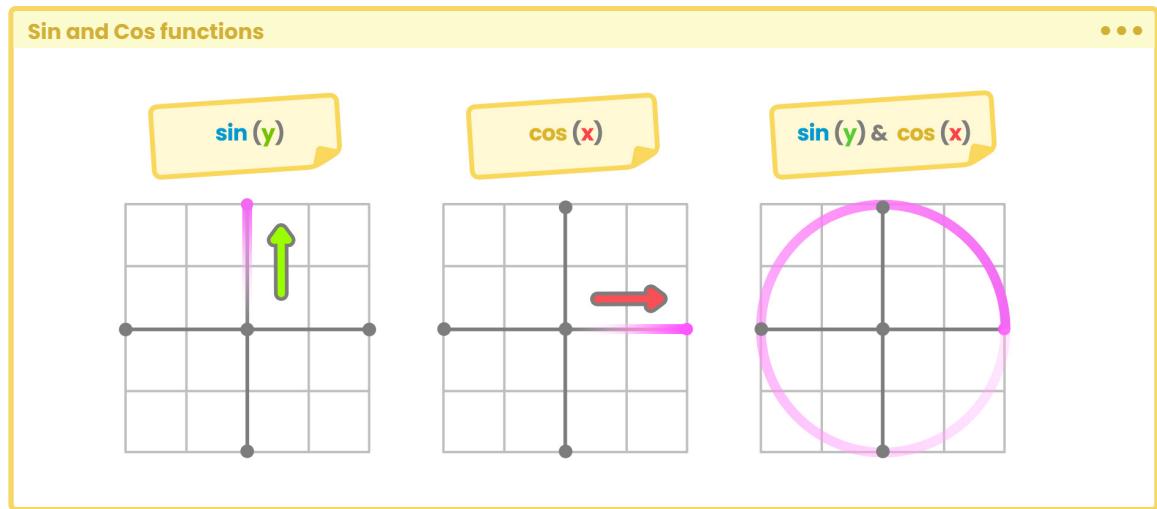
(Fig. 4.1.1a. Graphical representations of the functions on a Cartesian plane. On the left, you can see **$\sin(x)$** , and on the right, **$\cos(x)$**)

These functions also included in Cg/HLSL are very useful in Computer Graphics, with them you can generate multiple geometric figures and even matrix transformations.

A practical example of implementation is the rotation of vertices in an object.

As you already know, a vertex has three space coordinates (XYZ) considered as vectors. Given its nature, you can transform these values and generate the illusion of rotation from a matrix.

Imagine that you want to rotate a vertex in a two-dimensional space. Applying the "sin" function on its Y_{AX} axis will obtain a wave motion going from top to bottom. Using the "cos" function on its X_{AX} axis will reproduce a circular motion.



(Fig. 4.1.1b. Rotation is mainly caused by the time lag between sin and cos)

To understand the concept, do the following: create a new Unlit shader and call it **USB_function_SINCOS**. You will use this shader to generate a small rotation animation of the Cube's vertices. Start by declaring a floating range, which you will use later in the function to determine the rotation speed.

Sin and Cos functions

```
Shader "USB/USB_function_SINCOS"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Speed ("Rotation Speed", Range(0, 3)) = 1
    }
    SubShader { ... }
}
```

You will need the help of a matrix to rotate coordinates; in section 3.2.7, you were able to review its structure using matrices of different dimensions. This time, you will use a three times three-dimensional matrix to generate rotation of the object.

Sin and Cos functions

```
Pass
{
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Speed;

    // add our rotation function
    float3 rotation(float3 vertex)
    {
        // create a three-dimensional matrix
        float3x3 m = float3x3
        (
            1, 0, 0,
            0, 1, 0,
            0, 0, 1
        );
    }
}
```

Continued on the next page.

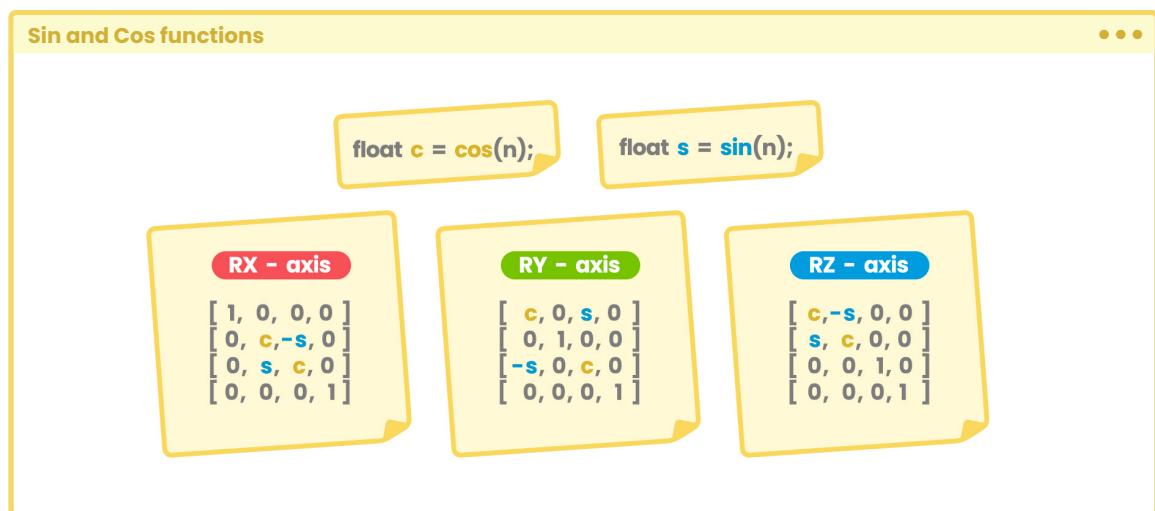
```

    // multiply the matrix by the Vertex Input
    return mul(m, vertex);
}
...
}

```

The **rotation** function returns a three-dimensional vector and alone, does not perform any specific action because the matrix **m** has only its identity values. However, you can use it later to transform the vertices of an object in the Vertex Shader Stage.

Start by selecting a rotation axis; paying attention to the following diagram.



(Fig. 4.1.1c. The rotation axis in a 4x4 matrix)

In the figure above, each of the matrices represents a rotation transformation axis for an object. This time, perform the exercise using the **RY axis** seen in Figure 4.1.1c, Then add two floating variables with the rotation method, applying the **sin(N_{RG})** and **cos(N_{RG})** trigonometric functions.

```

Sin and Cos functions

```

```

float3 rotation(float3 vertex)
{
    // let's add the rotation variables
    float c = cos(_Time.y * _Speed);

```

Continued on the next page.

```

float s = sin(_Time.y * _Speed);

// let's add the rotation variables
float3x3 m = float3x3
(
    c, 0, s,
    0, 1, 0,
    -s, 0, c
);
// let's multiply the matrix times the vertex input
return mul(m, vertex);
}

```

Later, in section 4.2.4, you will see the `_Time` property in detail, for now, only consider that this variable adds time to the operation, very similar to the behavior of `Time.timeSinceLevelLoad` in C#.

`_Time` will influence the Cube's vertices in the scene, so they will start to move according to their rotation axis.

Up to now the `rotation` function can already operate perfectly. Next, you must implement it in the Vertex Shader Stage. For this purpose, you must consider the function `UnityObjectToClipPos(vRG)` since, as explained in section 3.3.2, it allows the vertices of an object to be transformed from Object-Space to Clip-Space. Therefore, you must implement the vertex rotation before transforming their coordinates into screen position.

Sin and Cos functions

• • •

```

v2f vert (appdata v)
{
    v2f o;
    float3 rotVertex = rotation(v.vertex);
    o.vertex = UnityObjectToClipPos(rotVertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    return o;
}

```

In the previous exercise, a new three-dimensional vector called **rotVertex** was declared, in which the mesh vertices input was stored along with their rotation in Object-Space. This vector was then used as an argument in the **UnityObjectToClipPos(*v_{RG}*)** function.

If you go back to Unity and press the Play button, you will see the rotation of the Cube vertices as a unit.

4.1.2. Tan function.

This trigonometric function refers to the tangent of an angle, that is:

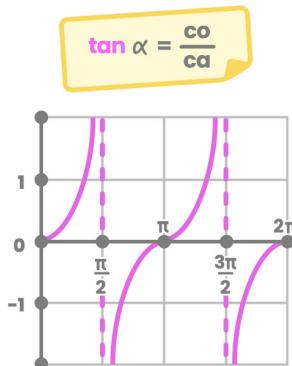
- The ratio of the opposite side to the adjacent side.

Its syntax is as follows:

Tan function

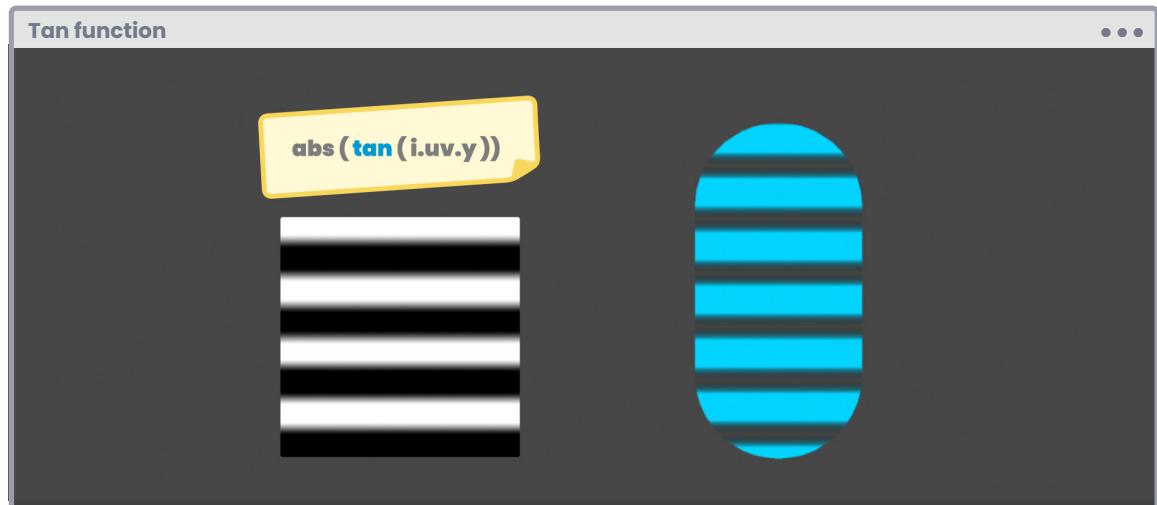
```
float tan (float n);
float2 tan (float2 n);
float3 tan (float3 n);
float4 tan (float4 n);
```

Tan function



(Fig. 4.1.2a. Graphical representation of the tan function on a Cartesian plane)

Like **sin** and **cos**, **tan(N_{RG})** is very useful in the calculation of geometric figures and repeating patterns. A practical example of implementation is the generation of a grid-like procedural mask, which you can use to generate a holographic projection effect on an object. To do this, simply calculate the absolute value of the tangent at one of the UV coordinates within the Fragment Shader Stage.



(Fig. 4.1.2b)

You can see this by generating a new Unlit shader and call it **USB_function_TAN**. Start by declaring a color property and a range to increase or decrease the number of lines you want to project.



It's worth mentioning that you can apply this shader to 3D objects included in the software. The reason for saying this is due to the operation you will perform on the V-coordinate of the UV.

As you can see in Figure 4.1.2b, such a visual effect has transparency; therefore, you will have to add Blending options in the SubShader so that black will be recognized as an Alpha channel.

```
Tan function

SubShader
{
    Tags {"RenderType"="Transparent" "Queue"="Transparent"}
    Blend SrcAlpha OneMinusSrcAlpha

    Pass { ... }

}
```

Make sure to declare the global variables and then go to the Fragment Shader Stage to add the functionality that will project the horizontal lines on the object.

```
Tan function

float4 _Color;
float _Sections;

fixed4 frag (v2f i) : SV_Target
{
    float4 tanCol = abs(tan(i.uv.y * _Sections));
    tanCol *= _Color;
    fixed4 col = tex2D(_MainTex, i.uv) * tanCol;

    return col;
}
```

In the previous example, you declared a four-dimensional vector called **tanCol**, and in it you stored the result of the absolute value for the calculation of the tangent; of the product between the V of the UV coordinates and the **_Sections** property. Then, the factor between the **texture** and the vector **tanCol** were stored in the four-dimensional vector named **col**. Therefore, the texture that you assign to the **_MainTex** property will be interlined.

A small detail from the previous operation is that the tangent of V returned a numerical range less than 0.0f and greater than 1.0f, therefore, the final color will be saturated on the computer screen. To solve this problem, use the **clamp(A_{RG}, X_{RG}, B_{RG})** function, limiting the values between “zero and one.”

```
Tan function
fixed4 frag (v2f i) : SV_Target
{
    float4 tanCol = clamp(0, abs(tan(i.uv.y * _Sections)), 1);
    tanCol *= _Color;
    fixed4 col = tex2D(_MainTex, i.uv) * tanCol;

    return col;
}
```

You can use the `_Time` variable again to generate movement in the spacing. To do this, you simply add or subtract this property from the V coordinate in the previous operation.

```
Tan function
fixed4 frag (v2f i) : SV_Target
{
    float4 tanCol = clamp(0, abs(tan((i.uv.y - _Time.x) * _Sections)), 1);
    tanCol *= _Color;
    fixed4 col = tex2D(_MainTex, i.uv) * tanCol;

    return col;
}
```

4.1.3. Exp, Exp2 and Pow functions.

These functions are characterized by using exponents in their operations, e.g., the function **exp(N_{RG})** returns the exponential of **base-e** in scalar and vector values, that is to say 2.7182828182846f raised to a number.

$$\exp(2) = 7.3890560986f \quad \text{It's the same as } 2.71828182846f^2$$

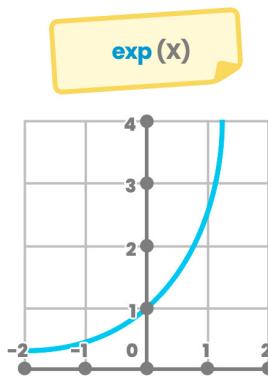
Its syntax is as follows:

Exp, Exp2 and Pow functions

```
float exp (float n)
{
    float e = 2.71828182846;
    float en = pow (e, n);
    return en;
}

float2 exp (float2 n);
float3 exp (float3 n);
float4 exp (float4 n);
```

Exp, Exp2 and Pow functions



(Fig. 4.1.3a. Graphical representation of $\exp(x)$ on a Cartesian plane)

Instead, **exp2(\mathbf{N}_{RG})** returns the **base-2** exponent in values of different dimensions, i.e., two raised to a number.

$\text{exp2}(3) = 8$	is the same as 2^3
$\text{exp2}(4) = 16$	
$\text{exp2}(5) = 32$	

Exp, Exp2 and Pow functions

```
float exp2 (float n);
float2 exp2 (float2 n);
float3 exp2 (float3 n);
float4 exp2 (float4 n);
```

On the other hand, **pow($\mathbf{x}_{RG}, \mathbf{N}_{RG}$)** has two arguments: The base number and its exponent.

$\text{pow}(3, 2) = 9$	is the same as 3^2
$\text{pow}(2, 2) = 4$	
$\text{pow}(4, 2) = 16$	

Exp, Exp2 and Pow functions

```
float pow (float x, float n);
float2 pow (float2 x, float2 n);
float3 pow (float3 x, float3 n);
float4 pow (float4 x, float4 n);
```

The usefulness of these functions will depend on the operation being performed. However, they are generally used to calculate noise, gamma increase in the output color, and repetition patterns.

4.1.4. Floor function.

This function returns an integer value not greater than its argument, i.e., a scalar or vector number without decimal places, rounded down, e.g., the floor of 1.97f returns “one,” why? Because this function subtracts the decimals of a number from its total.

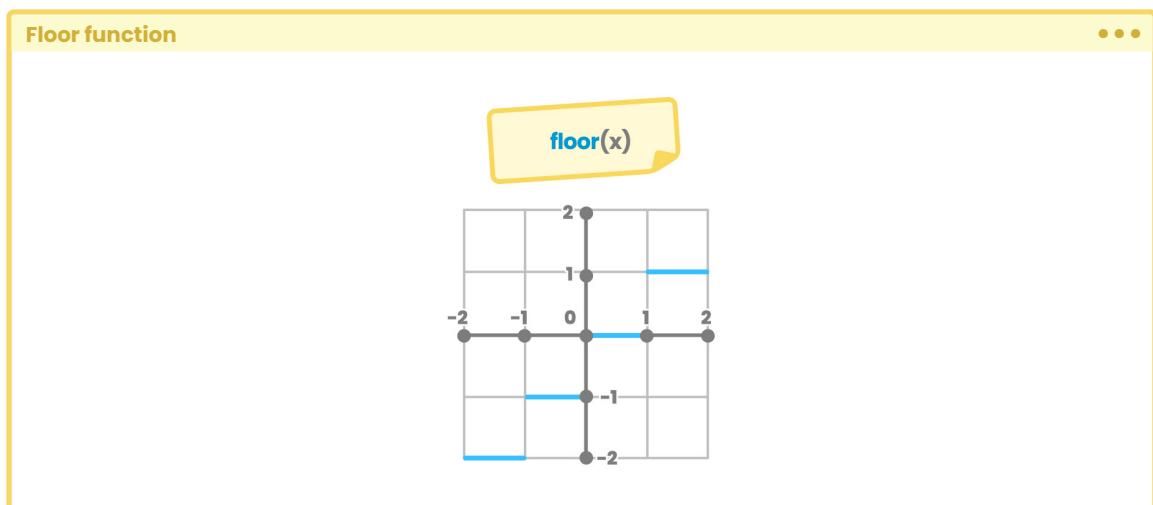
floor (1.56) = 1	is the same as 1.56f - 0.56f.
floor (0.34) = 0	
floor (2.99) = 2	

Its syntax is as follows,

Floor function

```
float floor (float n)
{
    float fn;
    fn = n - frac(n);
    return fn;
}

float2 floor (float2 n);
float3 floor (float3 n);
float4 floor (float4 n);
```



(Fig. 4.1.4a. Graphical representation of $\text{floor}(x)$ on a Cartesian plane)

Contrary to the $\text{ceil}(N_{RG})$ function, the $\text{floor}(N_{RG})$ is quite useful when creating visual effects with solid blocks of color, e.g., toon shader or repeating patterns in general.

To illustrate this, you need to understand the principle of implementing a toon shader. So, generate a new Unlit shader called **USB_functions_FLOOR**.

Start by adding two properties in your shader: one to generate multiple divisions and the second to increase the gamma in the output color.

```
Floor function
• • •
Shader "USB/USB_function_FLOOR"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        [IntRange]_Sections ("Sections", Range (2, 10)) = 5
        _Gamma ("Gamma", Range (0, 1)) = 0
    }
    SubShader { ... }
}
```

Since the sections must be added uniformly, **[IntRange]** has been defined for the **_Sections** variable. As in previous processes, you must include the global variables inside the pass so that ShaderLab and your program will communicate directly.

```
Floor function
Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Sections;
    float _Gamma;
    ...
    ENDCG
}
```

Next, go to the Fragment Shader Stage and declare a new variable, which will generate solid color blocks according to the V coordinate of the UV.

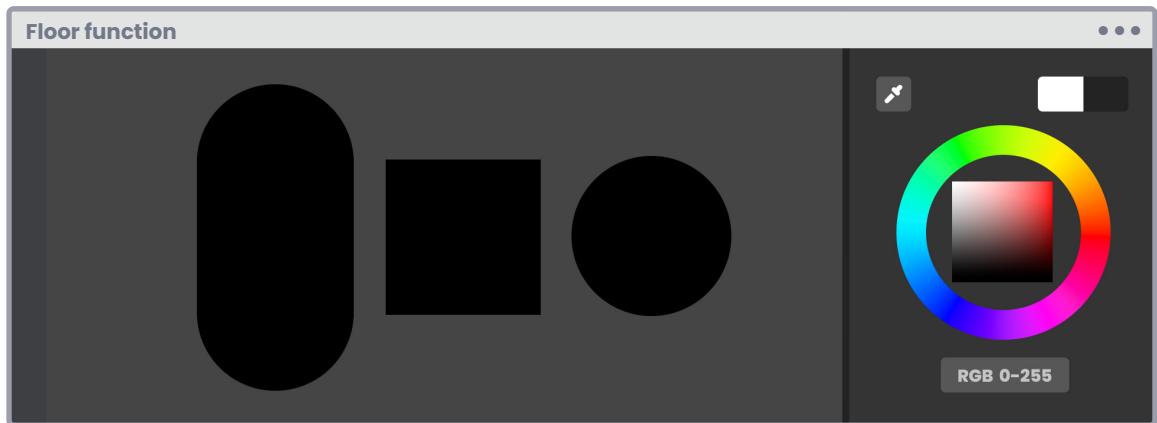
```
Floor function
fixed4 frag (v2f i) : SV_Target
{
    float fv = floor(i.uv.y);
    fixed4 col = tex2D(_MainTex, i.uv);
    return col;
}
```

In the previous operation, the variable **fv** was declared and initialized, with only one dimension. Its value is equal to the result of **floor(v)**, hence, it equals “zero.” This is because the UV coordinates start at 0.0f and end at 1.0f, and as you already know, this function returns an integer no greater than its argument.

You can check this by assigning a four-dimensional vector as output, where the first three are equal to the value of **fv**.

```
Floor function
fixed4 frag (v2f i) : SV_Target
{
    float fv = floor(i.uv.y);

    // fixed4 col = tex2D(_MainTex, i.uv);
    // return col;
    return float4(fv.xxx, 1);
}
```



(Fig. 4.1.4b)

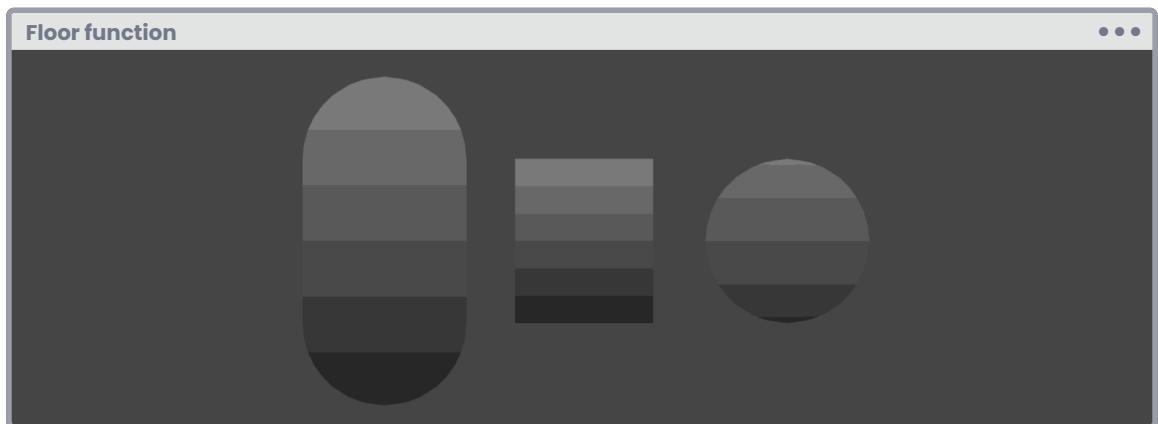
To add solid color blocks, simply multiply the operation by a certain number of sections and then divide by a decimal value.

As for gamma, you can add the property directly to the output color.

```
Floor function
fixed4 frag (v2f i) : SV_Target
{
    float fv = floor(i.uv.y * _Sections) * (_Sections/ 100.0);

    // fixed4 col = tex2D(_MainTex, i.uv);
    // return col;
    return float4(fv.xxx, 1) + _Gamma;
}
```





(Fig. 4.1.4c. The material has been configured with six sections and a gamma of 0.17f)

The implementation principle is the same for a toon shader, with the difference that you use global illumination in the calculation instead of the V-coordinate of UV.

The custom lighting implementation will be reviewed in detail in the Chapter II, section 7.0.3.

4.1.5. Step and Smoothstep functions.

Step and **smoothstep** are quite similar functions, in fact, both have an argument called “edge” in charge of differentiating the return between two values.

You will begin with **step(X_{RG} , E_{RG})** compression then the operation of **smoothstep(A_{RG} , B_{RG} , E_{RG})**, which has a more elaborate structure.

According to the official documentation at NVIDIA;

Step returns one for each component of X greater than or equal to the edge. Otherwise, it returns zero.

The syntax is as follows;

Step and Smoothstep functions

```
float step (float edge, float x)
{
    return edge >= x;
}

float2 step (float2 edge, float2 x);
float3 step (float3 edge, float3 x);
float4 step (float4 edge, float4 x);
```

To illustrate the previous statement from NVIDIA, perform a simple operation on the Fragment Shader Stage.

Step and Smoothstep functions

```
fixed4 frag (v2f i) : SV_Target
{
    // add the color edge
    float edge = 0.5;
    // return to RGB color
    fixed3 sstep = 0;
    sstep = step (edge, i.uv.y);

    // fixed4 col = tex2D (_MainTex, i.uv);

    return fixed4(sstep, 1);
}
```

You started by declaring a three-dimensional vector called **sstep**. The V coordinate of the UV and 0.5 as **edge** were used as the argument. You can see its graphical representation in Figure 4.1.5a. Finally, you have returned **sstep** in RGB and “one” for the Alpha channel.



(Fig. 4.1.5a. The step function is used on some primitive figures).

It is worth remembering that both the U and V coordinates start at 0.0f and end at 1.0f; therefore, all those less than or equal to E_{RG} will return “one” (white), and “zero” in the opposite case (black).

The behavior of the **smoothstep** function is not very different from the previous one; its only difference lies in the generation of a linear interpolation between the return values.

Its syntax is as follows;

Step and Smoothstep functions

```
float smoothstep (float a, float b, float edge)
{
    float t = saturate((edge - a) / (b - a));
    return t * t * (3.0 - (2.0 * t));
}

float2 smoothstep (float2 a, float2 b, float2 edge)
float3 smoothstep (float3 a, float3 b, float3 edge)
float4 smoothstep (float4 a, float4 b, float4 edge)
```

Going back to the operation in the Fragment Shader Stage, you could add a new variable to determine the amount of interpolation between the return values.

Step and Smoothstep functions

```

fixed4 frag (v2f i) : SV_Target
{
    // add the edge
    float edge = 0.5;
    // add the amount of interpolation
    float smooth = 0.1;
    // add the return value in RGB
    fixed3 sstep = 0;
    // sstep = step (edge, i.uv.y);
    sstep = smoothstep((i.uv.y - smooth), (i.uv.y + smooth), edge);

    // fixed4 col = tex2D (_MainTex, i.uv);

    return fixed4(sstep, 1);
}

```

In the previous exercises, you could modify the “smooth” value between 0.0f and 0.5f to obtain different levels of interpolation.

Step and Smoothstep functions



(Fig. 4.1.5b. Smooth equals 0.1f)

4.1.6. Length function.

As its name says, **length(N_{RG})** refers to the magnitude that expresses the distance between two points. This function is handy when creating geometric shapes, e.g., it can generate circles or polygonal shapes with rounded edges.

Its syntax is as follows:

```
Length function
float length (float n)
{
    return sqrt(dot(n,n));
}

float length (float2 n);
float length (float3 n);
float length (float4 n);
```

As usual, create a new Unlit shader called **USB_function_LENGTH**. This time you will use some functions to represent a circle in the program. Start by adding Properties that you will use later to enlarge, center, and smooth the shape.

```
Length function
Shader "USB/USB_function_LENGTH"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Radius ("Radius", Range(0.0, 0.5)) = 0.3
        _Center ("Center", Range(0, 1)) = 0.5
        _Smooth ("Smooth", Range(0.0, 0.5)) = 0.01
    }
    SubShader
    {
```

Continued on the next page.

```

Pass
{
    ...
    float _Smooth;
    float _Radius;
    float _Center;
    ...
}

}
}
}

```

To create a circle, simply calculate the magnitude of the UV coordinates and subtract the radius. Its representation would look like the following:

Length function

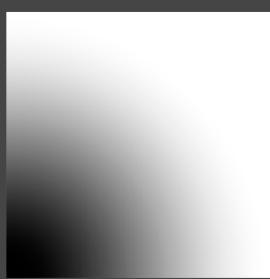
```

float circle (float2 p, float radius)
{
    // create the circle
    float c = length(p) - radius;
    return c;
}

```

However, as you can see in Figure 4.1.6a, the previous operation returned a blurred circle starting at the point 0.0f and ending at 1.0f, but what you are looking for corresponds to a centered circle and more compact.

Length function



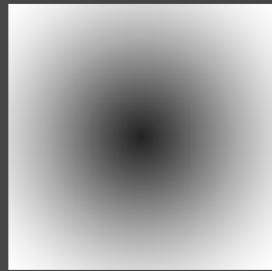
(Fig. 4.1.6a. The circle fades as the value of the UV coordinates increases)

To correct this add a new argument in the function that centers the circle on the object to which you are applying the material.

Length function

```
float circle (float2 p, float center, float radius)
{
    // the argument center is equal to a range
    // between 0.0f and 1.0f
    float c = length(p - center) - radius;
    return c;
}
```

Length function



(Fig. 4.1.6b. The center is equal 0.5f)

However, the circle will still be blurred. If you want to control the amount of blurring, you can use the **smoothstep** function, which, as you already know, generates a linear interpolation between two values.

Length function

```
float circle (float2 p, float center, float radius, float smooth)
{
    float c = length(p - center);
    return smoothstep(c - smooth, c + smooth, radius);
}
```

In the previous operation, a new argument was added called **smooth**, to control the amount of blurring. You can now apply this function in the Fragment Shader Stage as follows.

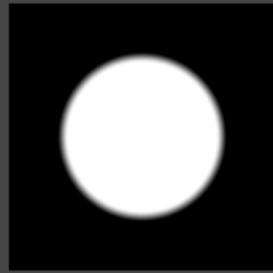
Length function

```
float circle (float2 p, float center, float radius, float smooth)
{ ... }

fixed4 frag (v2f i) : SV_Target
{
    float c = circle (i.uv, _Center, _Radius, _Smooth);
    return float4(c.xxx, 1);
}
```

As you can see, you have created a one-dimensional variable called "c" which has been initialized with the default values of the circle. It is essential to pay attention to the color output because the same channel (R) is being used for the three outputs (c.xxx).

Length function



(Fig. 4.1.6c. Radius 0.3f, Center 0.5f and Smooth 0.023f)

4.1.7. Frac function.

This function returns the fraction of a value, that is to say, its decimal values, e.g., **frac(1.534)** returns 0.534f, why? This is due to the operation performed in the function.

$\text{frac}(3.27) = 0.27f$ $\text{frac}(0.47) = 0.47f$ $\text{frac}(1.0) = 0.0f$	is the same as $3.27f - 3$.
---	------------------------------

Its syntax is as follows:

```
Frac function
float frac (float n)
{
    return n - floor(n);
}

float2 frac (float2 n);
float3 frac (float3 n);
float4 frac (float4 n);
```

frac(N_{RG}) is used in multiple operations like noise calculation, random repeating patterns, and much more.

To understand the concept, try the following:

- 1 Create a new Unlit shader and call it **USB_function_FRAC**.
- 2 Start by adding a property called **_Size** that you will use later to increase or decrease the size of a circle.

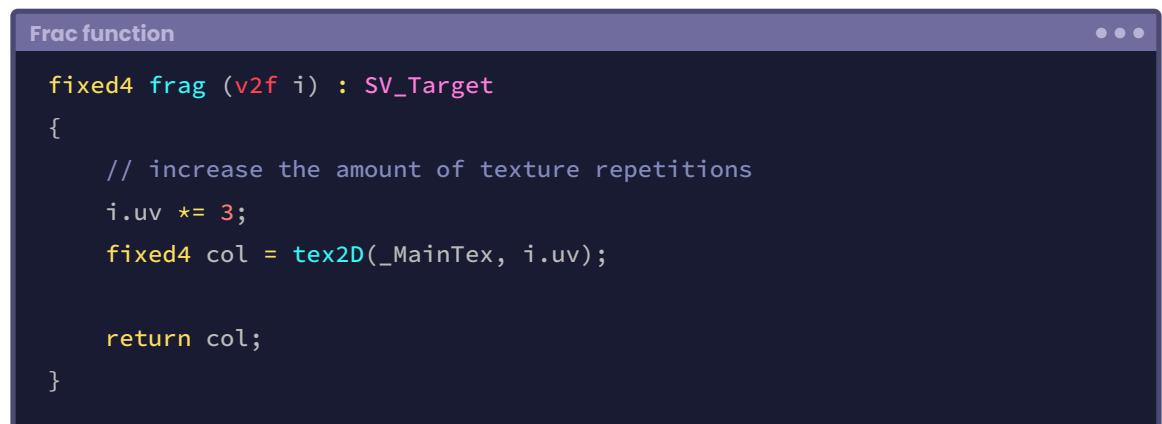
```
Frac function
Shader "USB/USB_function_FRAC"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Size ("Size", Range(0.0, 0.5)) = 0.3
    }
    SubShader
    {
        ...
        Pass
        {
            ...
        }
    }
}
```

Continued on the next page.

```

    ...
    float _Size;
    ...
}
}
}
}
```

The first operation to perform is to multiply the UV coordinates to obtain a repeating pattern in the Fragment Shader Stage.



```

Frac function

fixed4 frag (v2f i) : SV_Target
{
    // increase the amount of texture repetitions
    i.uv *= 3;
    fixed4 col = tex2D(_MainTex, i.uv);

    return col;
}
```

If you go back to Unity and assign a texture configured in **Clamp**, you will get a similar result to figure 4.1.7a. You can notice that the edge texels in the image are stretched along itself.



(Fig. 4.1.7a. The image on the left has default UV coordinates, while the images on the right have the exact coordinates multiplied by three. The Wrap Mode corresponds to Clamp)

In this case, you can use the **frac** function to return the fractional value of the UV coordinates to generate a defined repeating pattern.

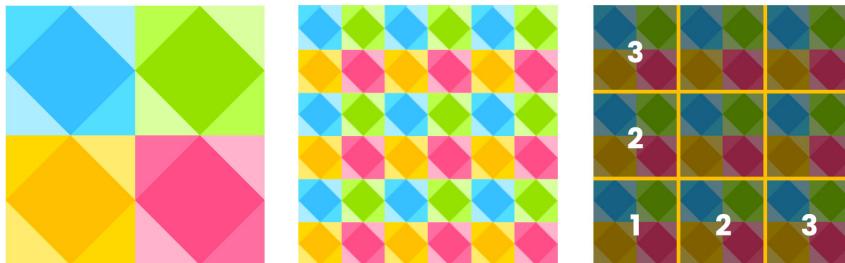
Frac function

```
fixed4 frag (v2f i) : SV_Target
{
    // increase the amount of texture repetitions
    i.uv *= 3;
    float2 fuv = frac(i.uv);

    fixed4 col = tex2D(_MainTex, fuv);

    return col;
}
```

Frac function



(Fig. 4.1.7b)

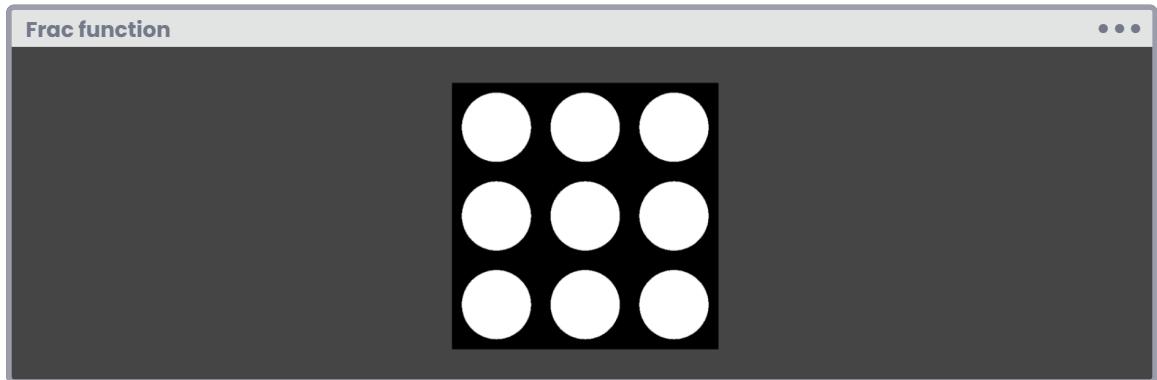
It is worth mentioning that performing this operation on a texture is not very useful, since you can easily set it to **Repeat** mode from the Inspector.

For a more practical example, this syntax will generate a pattern along the texture using a circle.

```
Frac function
fixed4 frag (v2f i) : SV_Target
{
    // increase the amount of texture repetitions
    i.uv *= 3;
    float2 fuv = frac(i.uv);
    // generate the circle
    float circle = length(fuv - 0.5);
    // flip the colors and return an integer value
    float wCircle = floor(_Size / circle);
    // fixed4 col = tex2D(_MainTex, fuv);

    return float4(wCircle.xxx, 1);
}
```

If you pay attention to the returned value, you will notice that the same channel is being used for the output colors in RGB (wCircle.xxx). If you modify the value of the **_Size** property, you can increase or decrease the size of the circles.



(Fig. 4.1.7c. The property **_Size** has been configured in 0.3f)

4.1.8. Lerp function.

Commonly used in color transitions, as the name suggests, this function allows a linear interpolation between two values, e.g., you can use **lerp(A_{RG}, B_{RG}, N_{RG})** on some of your characters to go from one skin to another through a crossfade.

Its syntax is as follows:

```
Lerp function
• • •

float lerp (float a, float b, float n)
{
    return a + n * (b - a);
}

float2 lerp (float2 a, float2 b, float2 n);
float3 lerp (float3 a, float3 b, float3 n);
float4 lerp (float4 a, float4 b, float4 n);
```

As you have done before, to demonstrate the function create a new Unlit shader called **USB_function_LERP**. Start by declaring two Textures that you will use later as “skins” in the effect, plus a numeric range to perform the crossfading.

```
Lerp function
• • •

Shader "USB/USB_function_LERP"
{
    Properties
    {
        _Skin01 ("Skin 01", 2D) = "white" {}
        _Skin02 ("Skin 02", 2D) = "white" {}
        _Lerp ("Lerp", Range(0, 1)) = 0.5
    }
    SubShader
    {
        ...
    }
}
```

Continued on the next page.

```

Pass
{
    ...
    sampler2D _Skin01;
    float4 _Skin01_ST;
    sampler2D _Skin02;
    float4 _Skin02_ST;

    float _Lerp;
    ...
}

}
}
}

```

Since you will use two textures for this effect, it is necessary to use UV coordinates in each case. These must be declared in both the Vertex Input and Output for two main reasons:

- 1 The textures will be affected by **Tiling** and **Offset** through the TRANSFORM_TEX function.
- 2 You will use them later in the Fragment Shader Stage.

Lerp function

```

struct appdata
{
    float4 vertex : POSITION;
    // create the UV coordinates for each case 01 and 02
    float2 uv_s01 : TEXCOORD0;
    float2 uv_s02 : TEXCOORD1;
};

struct v2f
{
    float4 vertex : SV_POSITION;
    // use the UV coordinates in the Fragment Shader Stage
    float2 uv_s01 : TEXCOORD0;
    float2 uv_s02 : TEXCOORD1;
}

```

Continued on the next page.

```

};

v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    // add tiling and offset for each case
    o.uv_s01 = TRANSFORM_TEX(v.uv_s01, _Skin01);
    o.uv_s02 = TRANSFORM_TEX(v.uv_s02, _Skin02);
    return o;
}

```

In the Fragment Shader Stage, you can declare two four-dimensional vectors, use the function **tex2D(S_{RG}, UV_{RG})** in each case, and then perform a linear interpolation between them using the function **lerp(A_{RG}, B_{RG}, N_{RG})**.

Lerp function

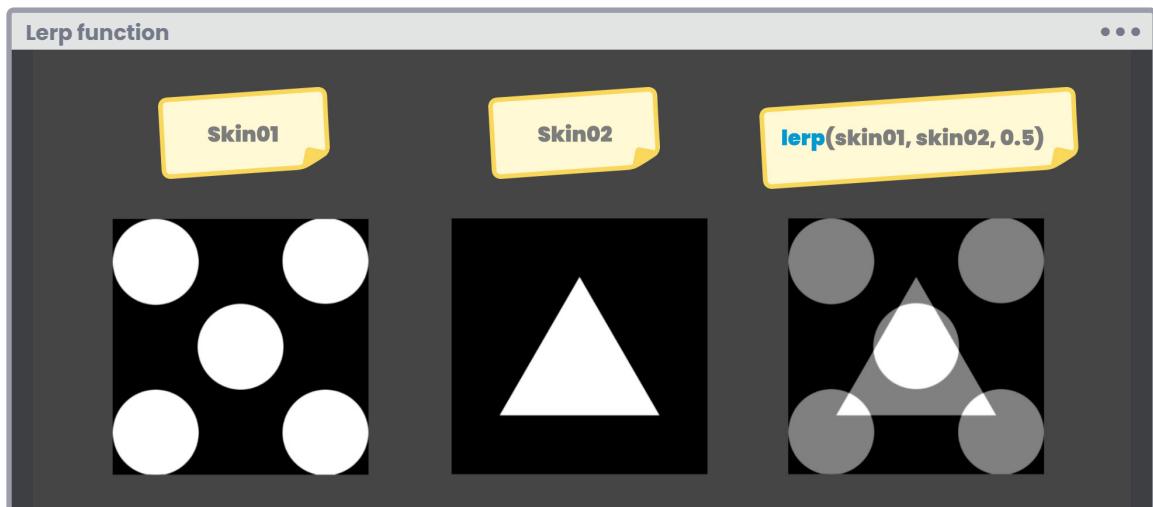
```

fixed4 frag (v2f i) : SV_Target
{
    // create a vector for each skin
    fixed4 skin01 = tex2D(_Skin01, i.uv_s01);
    fixed4 skin02 = tex2D(_Skin02, i.uv_s02);
    // make a linear interpolation between each color
    fixed4 render = lerp(skin01, skin02, _Lerp);

    return render;
}

```

If you look at the **_Lerp** property, you will notice that its value has been limited between 0.0f and 1.0f. By default, it has the value 0.5f; therefore, if you assign two different textures to each property of **_Skin[n]**, the result will equal a transparency or fading of 50 % in each case.



(Fig. 4.1.8a. Lerp between two textures)

4.1.9. Min and Max functions.

On the one hand, **min** refers to the minimum value between two vectors or scalars, while **max** is the opposite. You will use these functions frequently in different operations, e.g., you can use **max(A_{RG}, B_{RG})** to calculate the diffusion on an object, returning the maximum between "zero" and the dot product between the Normals of the Mesh and the light direction.

Its syntax is as follows:

Min and Max functions

```
// if "a" is less than "b," return "a," otherwise return "b"
float min (float a, float b)
{
    float n = (a < b ? a : b);
    return n;
}

float2 min (float2 a, float2 b);
float3 min (float3 a, float3 b);
float4 min (float4 a, float4 b);
```

Min and Max functions

```
// if "a" is greater than "b," return "a," otherwise return "b"
float max (float a, float b)
{
    float n = (a > b ? a : b);
    return n;
}

float2 max (float2 a, float2 b);
float3 max (float3 a, float3 b);
float4 max (float4 a, float4 b);
```

You will review this function in detail later in Chapter II, section 7.0.3, discussing diffuse reflection.

4.2.0. Timing and animation.

In Unity, there are three Built-in Shader Variables that you will frequently use in animating properties for effects. These refer to:

- **_Time**.
- **_SinTime**.
- And **_CosTime**.

Such variables are four-dimensional vectors, where each dimension represents a speed level, e.g., **_Time.y** is equal to the time (in seconds) that elapses since the scene or level has been loaded, similar to the **Time.timeSinceLevelLoad** function. In contrast, **_Time.x** corresponds to the same value, divided by twenty.

Its syntax is as follows:

Timing and animation

```
// "t" time in seconds.  
_Time.x = t / 20;  
_Time.y = t;  
_Time.z = t * 2;  
_Time.w = t * 3;
```

The `_CosTime` and `_SinTime` variables have the same behavior because they correspond to simplified functions of `_Time`, e.g., `_CosTime.w` is the same as the `cos(_Time.y)` operation; both return the same result, likewise for `sin(_Time.y)`.

Its syntax is as follows:

Timing and animation

```
_SinTime.x = t / 8;  
_SinTime.y = t / 4;  
_SinTime.z = t / 3;  
_SinTime.w = t;           // sin(_Time.y);  
  
_CosTime.x = t / 8;  
_CosTime.y = t / 4;  
_CosTime.z = t / 3;  
_CosTime.w = t;           // cos(_Time.y);
```

To go deeper into the concept, you could use the `_Time` function to generate the Super Mario icons' offset animation. For this purpose, you would have to add time to the U coordinate of the UV in the Fragment Shader Stage.

```
Timing and animation

fixed4 frag (v2f i) : SV_Target
{
    // add time to the U coordinate
    i.uv.x += _Time.y;
    fixed4 col = tex2D(_MainTex, i.uv);

    return col;
}
```



(Fig. 4.2.0a. U Offset on a Quad)

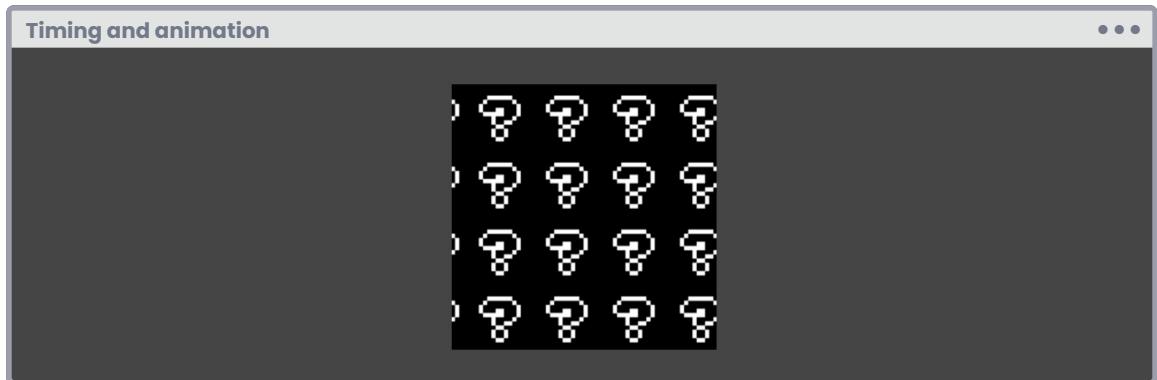
Or you could also generate a rotation animation using `_SinTime` and `_CosTime`, adding both U and V motion.

```
Timing and animation

fixed4 frag (v2f i) : SV_Target
{
    // add sine time to the U coordinate
    i.uv.x += _SinTime.w;
    // add cosine time to the V coordinate
    i.uv.y += _CosTime.w;

    fixed4 col = tex2D(_MainTex, i.uv);

    return col;
}
```



(Fig. 4.2.0b. Offset rotation in both U and V on a Quad. The tiling is equal to 4)



Chapter II
**Lighting, shadow
and surfaces.**

Introduction to the chapter.

One of the most complex concepts in Computer Graphics is the calculation of lighting, shadows, and surfaces. The operations we must perform to obtain good results depend on several functions and/or properties, that in most cases, are very technical and require a high level of mathematical understanding.

Before starting to create your functions, you must understand the theory behind each concept and then move on to their implementation in the HLSL.

It is worth mentioning that in Unity there are some predefined programs that facilitate lighting calculations for a surface (e.g. Standard Surface, Lit Shader Graph). However, it is essential to continue studying using **Unlit Shaders**, since being basic color models, they allow you to implement your own functions, and thus, obtain the necessary understanding to generate customized lighting and its derivatives, either in Built-in RP or Scriptable RP.

5.0.1. Configuring inputs and outputs.

In section 3.3.0 of the previous chapter, you learned the analogy between the property of a polygonal object and a semantic. Likewise, you could see how the latter is initialized within a **struct**.

This section will detail the steps necessary to configure your object Normals and transform its coordinates from Object-Space to World-Space.

As you already know, if you want to work with object Normals then you have to store the semantic **NORMAL[n]** in a three-dimensional vector. The first step is to include this value in the Vertex Input, as shown in the following example.

Configuring inputs and outputs

```
struct appdata
{
    ...
    float3 normal : NORMAL;
};
```

Remember that the fourth dimension of a vector corresponds to its W component, which, for a Normal, has a “zero” default value since it is a direction in space.

Once you have declared your object Normals as input, you must ask yourself if you need to pass these values to the Fragment Shader Stage? If the answer is yes, then you will have to declare the Normals once again, but this time as output.

This analogy applies equally to all the Inputs and Outputs that you use in your program.

Configuring inputs and outputs

```
struct v2f
{
    ...
    float3 normal : TEXCOORD1;
};
```

Why have you declared Normals in both **Vertex Input** and **Vertex Output**? This is because you will have to connect both properties within the Vertex Shader Stage, and then pass them to the Fragment Shader Stage. It should be noted that, according to the official HLSL documentation, there is no NORMAL semantic for the Fragment Shader Stage, therefore, you must use a semantic that can store at least three coordinates of space. That is why TEXCOORD1 has been used in the example above. This semantic has four dimensions (XYZW) and is ideal for working with Normals.

After having declared a property in both Vertex Input and Output, you can go to the Vertex Shader Stage to connect them.

Configuring inputs and outputs

```
v2f vert (appdata v)
{
    v2f o;
    ...
    // connect the output to the input
    o.normal = v.normal;
    ...
    return o;
}
```

In the example above you have connected the Normals output of **struct v2f** to the Normals input of **struct appdata**. Remember that v2f is used as an argument in the Fragment Shader Stage, this means that the object Normals can be used as a property in this stage.

To illustrate this concept, create an example function and call it **unity_light**, this will be responsible for calculating the surface lighting in the Fragment Shader Stage.

In itself, this function is not going to perform any actual operation, however, it will help you understand some factors that you should know.

Configuring inputs and outputs

```
void unity_light (in float3 normals, out float3 Out)
{
    Out = [Op] (normals);
}
```

According to its declaration, you can observe that **unity_light** is an empty function; that has two arguments: The object Normals and output value. Note that all lighting calculation operations require Normals as one of their variables, why? Because without this, the program will not know how the light should interact with the object surface.

Apply the **unity_light** function in the Fragment Shader Stage.

Configuring inputs and outputs

```
fixed4 frag (v2f i) : SV_Target
{
    // store the Normals in a vector
    half3 normals = i.normal;
    // initialize the light in black
    half3 light = 0;
    // initialize function and pass the vectors
    unity_light(normals, light);

    return float4(light.rgb, 1);
}
```

If you look closely at the previous exercise, you will notice that the Normals output that was declared in the previous stage (`i.normal`) is in Object-Space, how do you know this? You can easily determine this because there has not been any type of matrix transformation generated up to this point.

The operations for the lighting calculation must be in World-Space, why? Because incidence values are found in the world; within a scene. Likewise, the objects have a position according to the center of a grid, therefore, you will have to transform the space coordinates of the Normals in the Fragment Shader Stage.

To do this, do the following:

Configuring inputs and outputs

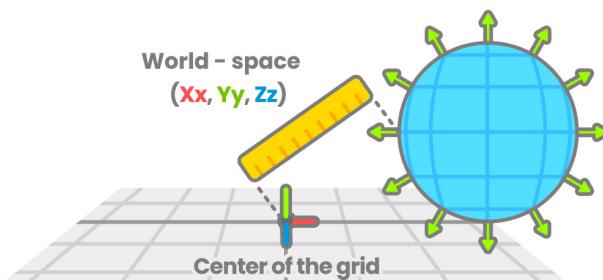


```
// create a new function
half3 normalWorld (half3 normal)
{
    return normalize(mul(unity_ObjectToWorld, float4(normal, 0))).xyz;
}

half4 frag (v2f i) : SV_Target
{
    // store the world-space normals in a vector
    float3 normals = normalWorld(i.normal);
    float3 light = 0;
    unity_light(normals, light);
    return float4(light.rgb, 1);
}
```

In the example above, the **normalWorld** function returns the space coordinate transformation for the Normals. In its process, it used the `unity_ObjectToWorld` matrix, which can transform coordinates from Object-Space to World-Space.

Configuring inputs and outputs



(Fig. 5.0.1a)

In the transformation function why are the Normals within a four-dimensional vector in the **normalWorld** multiplication function? As you already know these correspond to a direction in space, this means that their W component must equal “zero.” `unity_ObjectToWorld` is a four-by-four-dimensional matrix, as a result, you will obtain a new direction for the Normals in their four XYZW channels.

In the transformation you must make sure to assign the value “zero” to the W component, because, by the rules of arithmetic, any number multiplied by zero equals zero.

The process mentioned before can also be carried out in the Vertex Shader Stage. The operation is basically the same, except that if you do it at this stage, there will be a degree of optimization because the Normals will be calculated by vertices and not by the number of pixels on the screen.

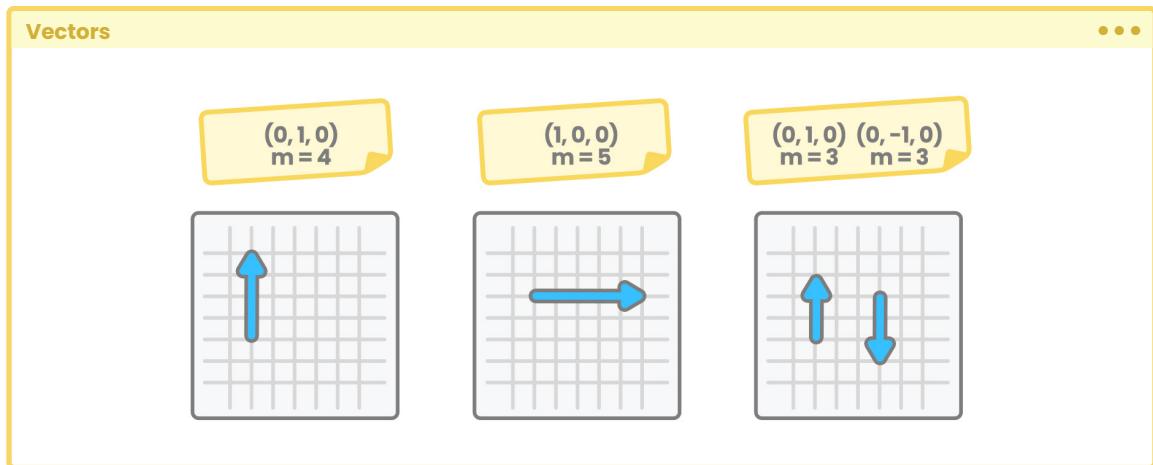
Configuring inputs and outputs

```
v2f vert (appdata v)
{
    ...
    o.normal = normalize(mul(unity_ObjectToWorld, float4(v.normal, 0))).xyz;
    ...
}
```

5.0.2. Vectors.

Before you start implementing lighting in your programs, you must first understand what a vector is and how it works in Computer Graphics.

A vector itself must be seen as a line or an arrow, possessing a magnitude and a direction.



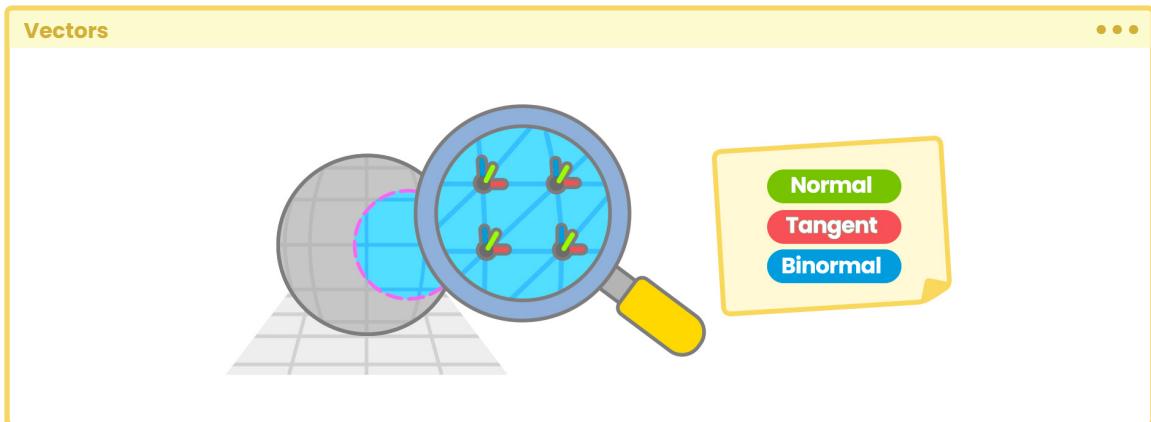
(Fig. 5.0.2a. The three-dimensional vector represents the direction, while "m" its magnitude.)

In the Figure above, a grid has been used as a measurement system to determine the vector magnitude. These values can be measured in both **scalar** and **vector magnitudes**.

Scalar magnitudes correspond to a single value; to a unit-value type number, e.g., [n] kilos, [n] hours, [n] degrees, etc. In this case, "n," which is a variable, represents a scalar magnitude because it contains a unique value of a specific unit. On the other hand, vector magnitudes; in addition to the unit-value, need a direction and a position in space, e.g., [n] km/h north, [n] N force down, etc.

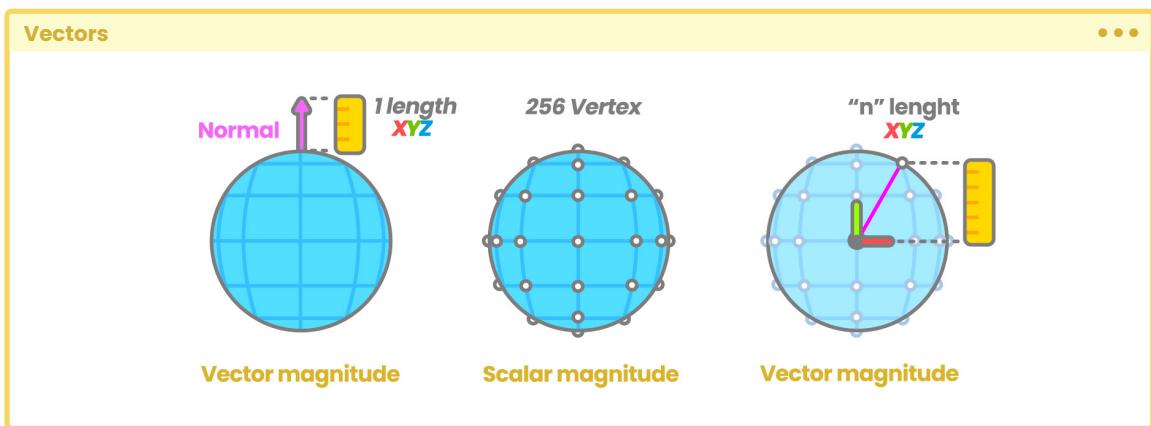
Because vector magnitudes have direction, you can conclude that they have a starting point and an endpoint and are also included within a two or three-dimensional coordinate system.

The object Normals are vector magnitudes, why? Because they have generally a standard length and a direction in space. Likewise, Tangents and Binormals as well because they have similar properties to a Normal.



(Fig. 5.0.2b)

Vertices, on the other hand, are scalar magnitudes since they represent points of intersection in geometry. Now, if you try to calculate the distance between the center of the object and the position of a vertex, you will obtain a vector magnitude, since, by definition, the unit-value would now have a direction and a position in space.



(Fig. 5.0.2c)

In Computer Graphics scalar magnitudes are represented as one-dimensional variables, e.g., float, half, fixed. While vector magnitudes are represented as variables of more than one dimension e.g., float2, float3, fixed4.

5.0.3. Dot Product.

The **dot(A_{RG}, B_{RG})** product is an operation that you will use frequently in the calculation of illumination and reflection since it allows you to determine the angle between two vectors and returns a scalar output value, that is, a one-dimensional variable. Generally, the resulting value will be normalized to ensure that the return range is between -1.0f and 1.0f.

$$a \cdot b = \|a\| \|b\| \cos \theta$$

(The “point” (\cdot) between a and b refers to the Dot Product.

The “ $\|x\|$ ” symbol refers to the vector magnitude.)

In the previous operation: when the angle between vectors “ a and b ” equals 0°, the Dot Product will return 1.0f. In turn, when the angle equals 90°, return 0.0f. Finally, when the angle equals 180°, the Dot Product will return -1.0f.

Its syntax is the following:

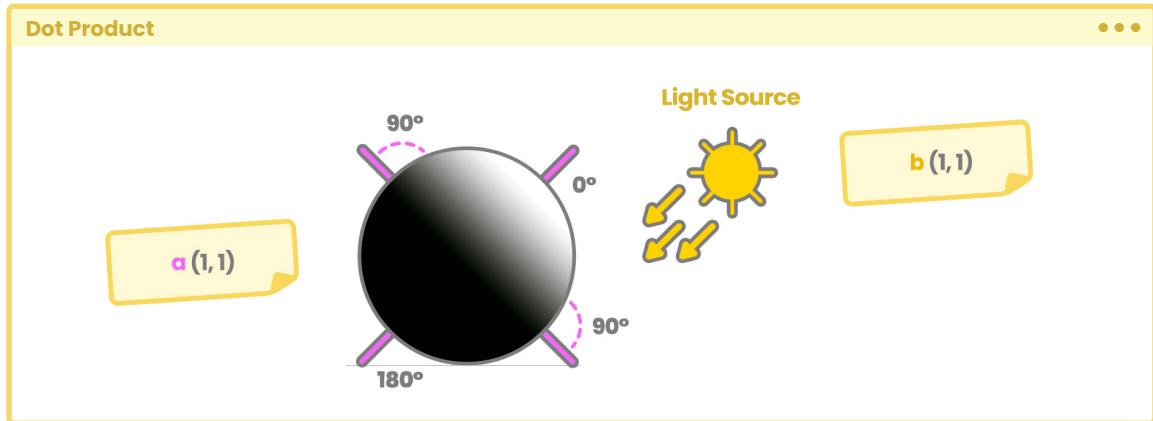
```
Dot Product
float dot(float4 a, float4 b)
{
    return a.x * b.x + a.y * b.y + a.z * b.z + a.w * b.w;
}
```

How does this operation work? To understand it you must calculate the Cosine of the angle between these two vectors.

$$\cos \theta = \frac{a \cdot b}{\|a\| \|b\|}$$

To understand the above function, assume you have two two-dimensional vectors with the following values.

- vector A (1, 1)
- vector B (1, 1)



(Fig. 5.0.3a Vector "A" corresponds to the direction of a Normal, while "B" refers to the direction of illumination)

Both vectors are equal in both direction and magnitude. To calculate the Dot Product between the two vectors you have to multiply A_x by B_x , then A_y by B_y and finally add the two values.

$$\begin{aligned} A_x * B_x &= 1 \\ A_y * B_y &= 1 \\ 1 + 1 &= 2 \end{aligned}$$

Consequently, the Product Point between A and B equals 2. To show this, replace these values in the function mentioned above.

$$\cos \theta = \frac{2}{\|a\| \|b\|}$$

Then calculate the magnitude of both A and B. To do this, perform the following operation.

$$\| v \| = \sqrt{vx^2 + vy^2}$$

So, for vector A.

$$\| A \| = \sqrt{1x^2 + 1y^2}$$

$$\| A \| = \sqrt{2}$$

Since vector B is exactly equal to vector A you can deduce that its value is the same.

$$\| B \| = \sqrt{1x^2 + 1y^2}$$

$$\| B \| = \sqrt{2}$$

If you multiply the magnitude of A by the magnitude of B, you get the following value.

$$\| A \| * \| B \| = \sqrt{2} * \sqrt{2} = \sqrt{2 * 2} = \sqrt{2^2} = 2$$

$$\| A \| * \| B \| = 2$$

As you can see, the factor between the magnitude of A and B equals two.

Again, replace this value in the previous function to understand it better.

$$\cos \theta = \frac{2}{2}$$

$$\cos \theta = 1$$

$$\theta = \cos^{-1}(1) = 0^\circ$$

$$\cos(0^\circ) = 1$$

So, the cosine of zero degrees equals one, therefore, if we carry out the initial operation ($a \cdot b = \|a\| \|b\| \cos \theta$) it would be as follows.

$$A \cdot B = 2 * 1 = 2$$

The result of the Product Point between A and B equals "two." Now, if you normalize this value, you will get a magnitude of one.

$$\text{normalize}(A \cdot B) = 1$$

What is the use of all this explanation? When you implement lighting in your shader, you have to use two vectors:

- One for the light calculation.
- And the other for the Normals calculation.

Then vector A could represent the global illumination and vector B represent the object Normals.

In its implementation, the lighting calculation is performed for each vertex in the object, so for those Normals that are in the same direction as the lighting, the Dot Product will return 1.0f, and for those that are on the opposite side, -1.0f.

5.0.4. Cross Product.

The **$\text{cross}(\mathbf{A}_{RG}, \mathbf{B}_{RG})$** function (also known as the Vector Product) is a function that, unlike the Dot Product, returns a three-dimensional vector that is perpendicular to its arguments.

Its syntax is as follows.

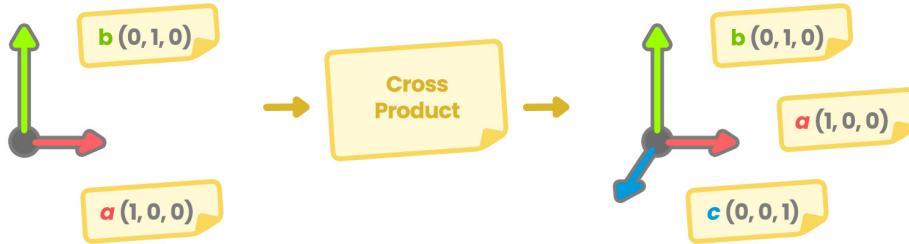
Cross Product

```
float3 cross(float3 a, float3 b)
{
    return a.yzx * b.zxy - a.zxy * b.yzx;
}
```

To understand the concept, take two vectors again: A and B, and position them in space as follows.

- vector A (1, 0, 0)
- vector B (0, 1, 0)

Cross Product



(Fig. 5.0.4a)

In the figure above, the Cross Product generates a third vector named "C" with new space coordinates. To understand how it works, pay attention to the next operation.

$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta$$

The magnitude of the resulting vector will be related to the function of sin .

Taking into consideration vectors A and B mentioned above, you can calculate the Cross Product from a determinant matrix between both vectors.

$$\text{Vector C} = X [(A_y * B_z) - (A_z * B_y)] Y [(A_z * B_x) - (A_x * B_z)] Z [(A_x * B_y) - (A_y * B_x)]$$

By replacing the values, you get.

$$\text{Vector C} = X [(0 * 0) - (0 * 1)] Y [(0 * 1) - (1 * 0)] Z [(1 * 1) - (0 * 0)]$$

$$\text{Vector C} = X [(0 - 0)] Y [(0 - 0)] Z [(1 - 0)]$$

$$\text{Vector C} = (0, 0, 1)$$

In conclusion, the resulting vector is perpendicular to its arguments.

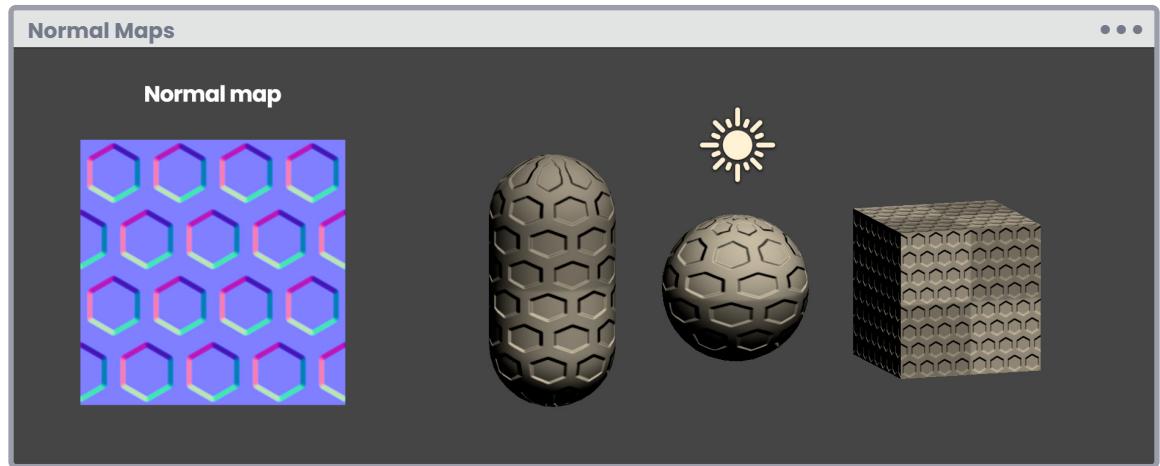
Later, you will use the $\text{cross}(A_{RG}, B_{RG})$ function to calculate the value of a Binormal in a Normals map.

Surface.

6.0.1. Normal Maps.

A Normal Map is a technique that allows you to generate details about a surface without the need to add more vertices to the object.

To perform this process, the Normals must change direction following a reference frame. To do this, each vertex is stored within a space coordinate called Tangent-Space. This type of space is used for the object surface lighting calculation.



(Fig. 6.0.1a)

To generate a Normal Map you have to use three normalized vectors, which together form a matrix called TBN:

- **Tangents.**
- **Binormals.**
- **Normals.**

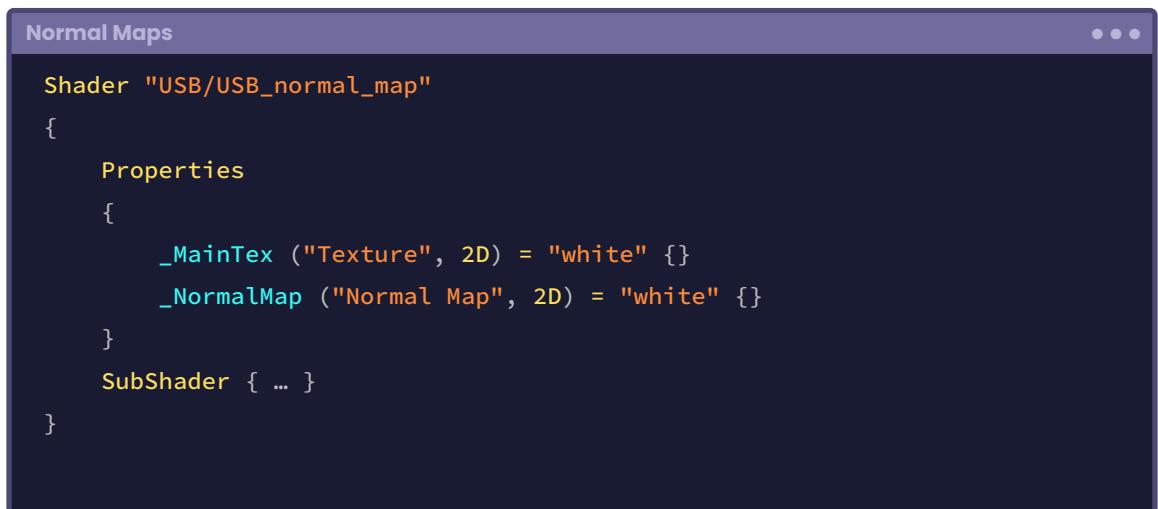
Section 5.0.1 talked about how to transform Normals into World-Space using the matrix **unity_ObjectToWorld**. Similarly, the TBN matrix is used to pass from one space to another, so you can transform both the lighting and Normal Map from World-Space to Tangent-Space.

The graphical representation of the TBN matrix is the following:

```
float4x4 TBN = float4x4
(
    Tx,      Ty,      Tz,      0,
    Bx,      By,      Bz,      0,
    Nx,      Ny,      Nz,      0,
    0,       0,       0,       0
);
```

In the matrix, the first row corresponds to Tangent values, the second row to Binormals and the third to Normals. This same order must be followed in its implementation.

To illustrate these concepts, create a new **Unlit** shader, and call it **USB_normal_map**. Start by adding a texture property to the program.



In the example above you have declared a 2D Texture Property called **_NormalMap**, this will be used to apply the Normal Map from the Inspector. Next add the connection Sampler corresponding to the declared property.

Normal Maps

```

Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    sampler2D _NormalMap;
    float4 _NormalMap_ST;
    ...
    ENDCG
}

```

Now, your Normal Map can be used inside the shader as a texture. Therefore, you must declare the TBN matrix. To do this, extract the Normals and Tangents from the object, go to the Vertex Input, and start the NORMAL and TANGENT semantics.

Normal Maps

```

Pass
{
    CGPROGRAM
    ...
    struct appdata
    {
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
        float3 normal : NORMAL;
        float4 tangent : TANGENT;
    };
    ...
    ENDCG
}

```

A factor to consider at this point is that both Normals and Tangents are in Object-Space and they need to be transformed to World-Space before being converted to Tangent-Space, therefore, you will have to connect them in the Vertex Shader Stage, and then, pass the Vertex Output to

the Fragment Shader Stage. For this, you must add the semantics corresponding to the Normals, Tangents and Binormals in the struct v2f for its calculation in World-Space.

```
Normal Maps
Pass
{
    CGPROGRAM
    ...
    struct v2f
    {
        float4 vertex : SV_POSITION;
        float2 uv : TEXCOORD0;
        float2 uv_normal : TEXCOORD1;
        float3 normal_world : TEXCOORD2;
        float4 tangent_world : TEXCOORD3;
        float3 binormal_world : TEXCOORD4;
    };
    ...
    ENDCG
}
```

As mentioned above, NORMAL or TANGENT semantics cannot be used within the Vertex Output, because they do not exist for this process according to the official HLSL documentation. In this a semantic is needed that can store up to four dimensions in each of its coordinates. This is the main reason why the semantic TEXCOORD[n] has been used in the previous example.

Now, if you pay attention, notice that each of the properties has a coordinate with a different ID, e.g., **uv_normal** has been assigned to TEXCOORD with its index at [1] while **binormal_world** has index [4]. It is essential that the IDs are different in value because, otherwise, you will be performing operations on a duplicated coordinate system.

To transform the properties from Object-Space to World-Space, go to the Vertex Shader Stage and perform the following operation:

Normal Maps

• • •

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);

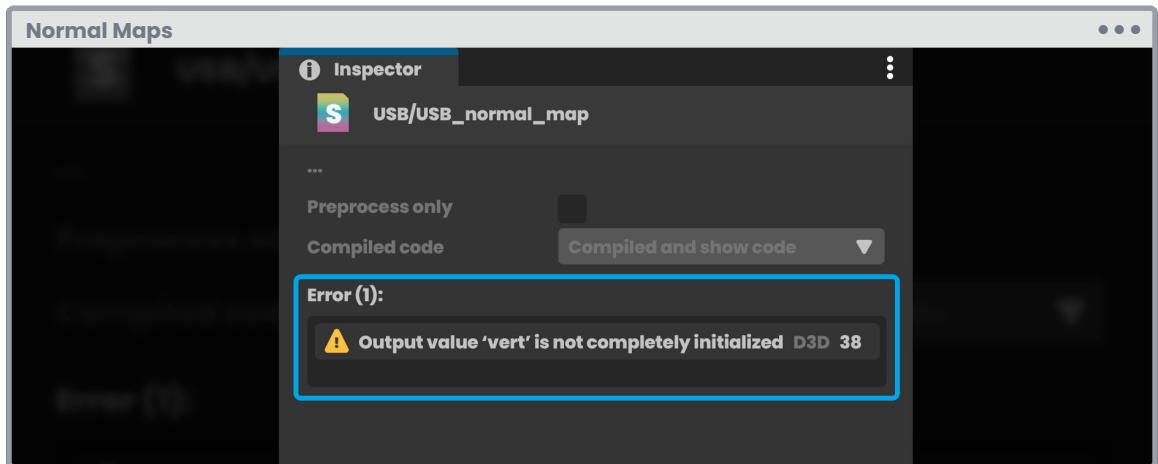
    // add tiling and offset to the normal map
    o.uv_normal = TRANSFORM_TEX(v.uv, _NormalMap);
    // transform the normals to world-space
    o.normal_world = normalize(mul(unity_ObjectToWorld, float4(v.normal, 0)));
    // transform tangents to world-space
    o.tangent_world = normalize(mul(v.tangent, unity_WorldToObject));
    // calculate the cross product between normals and tangents
    o.binormal_world = normalize(cross(o.normal_world, o.tangent_world) *
        v.tangent.w);

    return o;
}
```

In the example above, the operation started by adding **Tiling** and **Offset** to the Normal Map UV coordinates through the TRANSFORM_TEX function, which is included in UnityCg.cginc. Then, the four-dimensional matrix unity_ObjectToWorld was multiplied by the Normals input to transform its space coordinates from Object-Space to World-Space. The multiplication result was stored in the Normals output called **normal_world**, which you will use later for the per-pixel calculation in the Fragment Shader Stage.

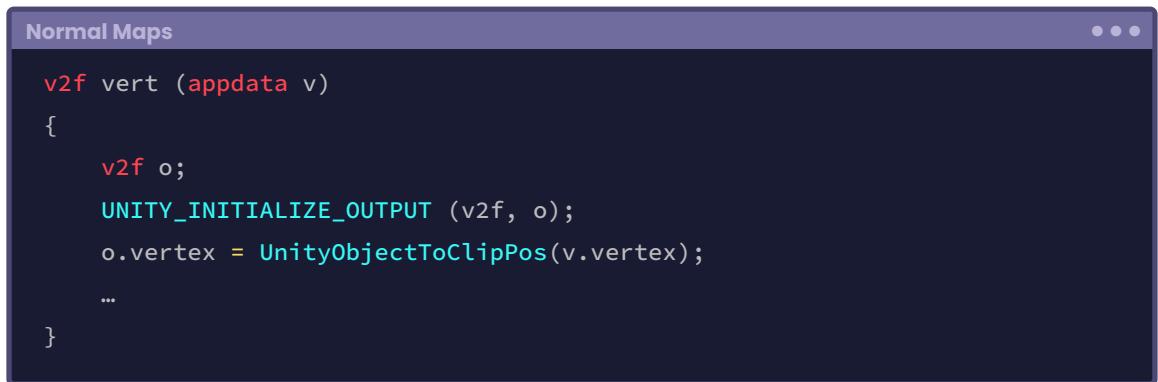
Next, the Tangents were multiplied by the matrix unity_WorldToObject to inversely transform their space coordinates to World-Space, and finally a perpendicular vector between the Normals and Tangents calculated using the **cross(A_{RG}, B_{RG})** function. The result gives Binormals and are stored in the output **binormal_world**.

It is possible that in Direct3D 11 it is necessary to initialize the Vertex Output **v2f** at "zero" to carry out the Normals calculation in the shader. To confirm this, a warning will appear in the Unity console, and the shader will show a small error related to this point.



(Fig. 6.0.1b)

In this case, you have to use the macro `UNITY_INITIALIZE_OUTPUT` within the Vertex Shader as follows:



Up to this point, your inputs and outputs are connected. Next, you must generate the TBN matrix to transform the coordinates of the Normal Map from World-Space to Tangent-Space. This process will be done in the Fragment Shader Stage.

One factor to consider when reading your Normal Map is that the XYZW coordinates are embedded within the RGBA channels, which have a range between 0.0f and 1.0f. It is fundamental to understand this concept since the Normal Map has a color range between -1.0f and 1.0f. Therefore, the first thing to do is to modify the numerical range using the following equation.

```
normal_map.rgb * 2 - 1;
```

To understand the above operation, do the following exercise: if you multiply a range between 0.0f and 1.0f, by “two,” then the new range value will be between 0.0f and 2.0f.

$0.0f * 2 = 0.0f$	Minimum color
$1.0f * 2 = 2.0f$	Maximum color

If you subtract “one” from this operation, then the range will be modified to a value between -1.0f and 1.0f.

$0.0f - 1 = -1.0f$	Minimum color
$2.0f - 1 = 1.0f$	Maximum color

Another factor to consider is that your Normal Map has more weight than a common texture, so it will be necessary to use DXT compression to reduce its graphic load on the GPU.

6.0.2. DXT compression.

Normal Maps are very useful when generating object detail; however, they are very heavy and produce a significant graphic load on the GPU. Likewise, if you are working on mobile devices, their processing will likely generate battery overheating, which could directly affect the user experience. For this reason, it will be essential to compress these textures within the shader.

DXT compression is one of the most commonly used to compress this type of image. When working with RGBA channels, each pixel needs 32 bits of information to be stored in the **Frame Buffer**. However, DXT compression divides the texture into blocks of “four by four” pixels, which are then minimized using only two of their AG channels, allowing the Normal Map to be reduced to $\frac{1}{4}$ resolution.

To understand this concept, you first have to calculate the Normal Map and its UV coordinates in the Fragment Shader Stage. Use the **tex2D(S_{RG}, UV_{RG})** function for this.

DXT compression

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    ...
}
```

As with the “col” vector in past exercises, a four-dimensional vector called **normal_map** has been generated and in it stored the Normal Maps and their UV coordinates. Its RGBA channels currently have a range between 0.0f and 1.0f which you will modify next.

To do this create a new function called **DXTCompression** and put it in the Fragment Shader Stage.

DXT compression

```
float3 DXTCompression (float4 normalMap)
{
    #if defined (UNITY_NO_DXT5nm)
        return normalMap.rgb * 2 - 1;
    #else
        float3 normalCol;
        normalCol = float3 (normalMap.a * 2 - 1, normalMap.g * 2 - 1, 0);
        normalCol.b = sqrt(1 - (pow(normalCol.r, 2) + pow(normalCol.g, 2)));
        ...
        return normalCol;
    #endif
}
```

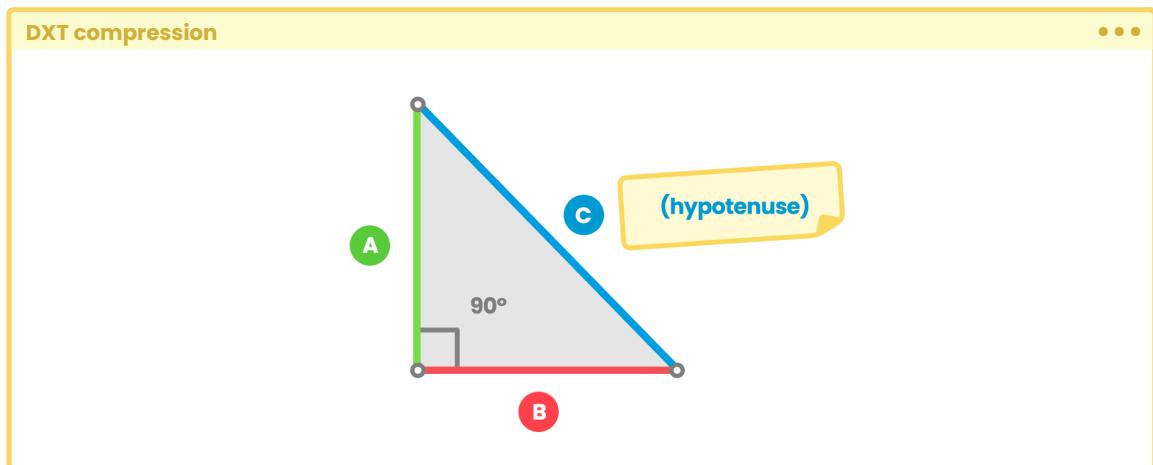
Four main things are happening in the previous function:

- 1 The function returns a three-dimensional vector according to a condition.
- 2 It has been called UNITY_NO_DXT5nm, which corresponds to a defined Built-In shader and its function is to compile shaders for platforms that do not support DXT5nm compression, which means that the Normal Maps will be encoded in RGB instead.
- 3 In the #else condition, you have generated DXT compression using only the two Alpha and Green channels. Notice that you have replaced the R channel with A, and then used the G for the second channel.
- 4 The third B channel in the vector has been discarded, but then calculated independently using the function:

```
sqrt(1 - (pow(normalCol.r, 2) + pow(normalCol.g, 2)))
```

Why are you using this feature? The answer lies in the Pythagoras theorem.

In every right triangle, the square of the hypotenuse is equal to the sum of the squares of the legs.



(Fig. 6.0.2a)

Consequently.

$$C^2 = A^2 + B^2$$

A vector in space generates a right triangle, so likewise, if you want to calculate the magnitude of a three-dimensional vector, you will have to do it through the Pythagoras theorem.

$$\| V \|^2 = A^2 + B^2 + C^2$$

When you transform the Normals, Tangent and Binormals space coordinates you use the **normalize** function which returns a vector with a magnitude of "one." Likewise, the magnitude of the vector that you use for the B or Z coordinate, which is the same, equals 1.0f, then the operation would be as follows:

$$1 = x^2 + y^2 + z^2 \quad \text{Or in its variation } R^2 + G^2 + B^2 \text{ which is the same.}$$

For the operation calculate the B or Z coordinate, then make the sum of X and Y equal 1.

$$1 - (x^2 + y^2) = z^2$$

Finally, by factorization, the above operation would equal:

$$Z = \sqrt{1 - (x^2 + y^2)}$$

Which in Cg or HLSL would be translated as:

```
normalCol.b = sqrt(1 - (pow(normalCol.r, 2) + pow(normalCol.g, 2)))
```

Why is this being done? Recall that the Normal Map has up to four channels and that only two of them are being used in DXT compression. The third channel was discarded in the **normalCol** vector, i.e. the B coordinate will not use its Normal Map values. But you must still calculate this coordinate, otherwise your texture will not work correctly, this is why the previous operation was carried out, where a new normalized vector has been calculated based on the AG coordinates.

Now you must compress the Normal Map, so go back to the Fragment Shader Stage and pass the texture as an argument in the **DXTCompression** function as follows:

```
DXT compression
float3 DXTCompression (float4 normalMap){ ... }

fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    fixed3 normal_compressed = DXTCompression(normal_map);
    ...
}
```

The exercise you have just performed is equivalent to the **UnpackNormal** function included in **UnityCg.cginc**, this means that you can use it to replace the DXTCompression and get the same result.

```
DXT compression
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    fixed3 normal_compressed = UnpackNormal(normal_map);
    ...
}
```

6.0.3. TBN Matrix.

As you already know, the TBN matrix is composed of the object Tangents, Binormals and Normals. In section 6.0.1 you looked at how to transform these properties from Object-Space to World-Space. What you will do next is create a new matrix to transform your Normal Map to Tangent-Space. To do this, go to the **Fragment Shader Stage** and create the matrix following the same order mentioned in the acronym TBN.

```
TBN Matrix
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    fixed4 normal_compressed = DXTCompression(normal_map);
    float3x3 TBN_matrix = float3x3
    (
        i.tangent_world.xyz,
        i.binormal_world,
        i.normal_world
    );
    ...
}
```

In the example above, you created a three by three dimension matrix called **TBN_matrix**, with all the Tangent and Binormal outputs and Normals in World-Space. Please note that in the case of tangents, you explicitly included their XYZ coordinates. This is because they were declared as a four-dimensional vector in the **struct v2f**, otherwise, if you do not specify the number of dimensions you want to use, the program will assume that it must use all of them (XYZW), and this could generate an error with the shader unable to compile them.

To conclude, all you have to do is multiply the Normal Map by the TBN matrix and return the result of the operation.

```

TBN Matrix

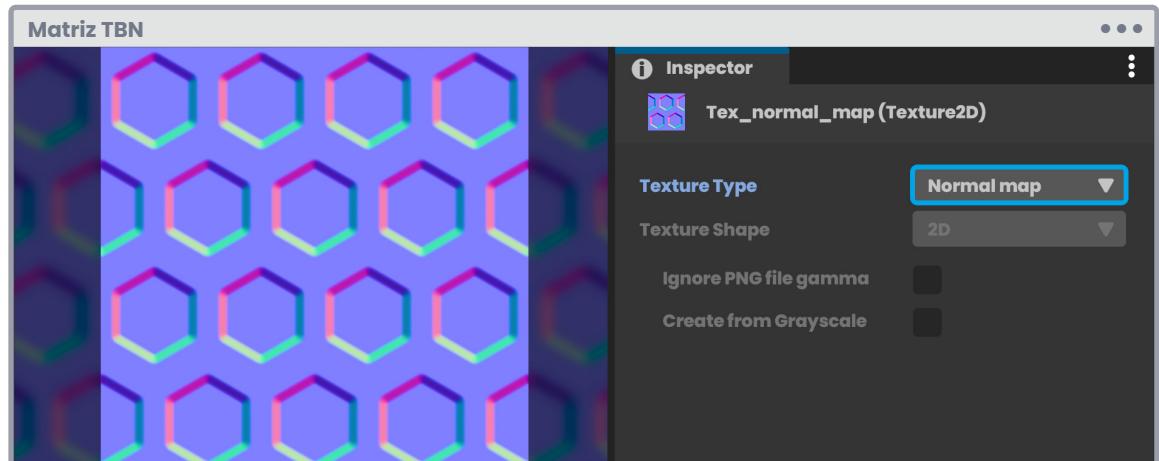
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    fixed3 normal_compressed = DXTCompression(normal_map);
    float3x3 TBN_matrix = float3x3
    (
        i.tangent_world.xyz,
        i.binormal_world,
        i.normal_world
    );
    fixed4 normal_color = normalize(mul(normal_compressed, TBN_matrix));
    return fixed4 (normal_color, 1);
}

```

The example above created a three-dimensional vector called **normal_color**. This vector has the result of the Normal Map and the TBN matrix. Finally, you returned the color of the Normals in RGB, and assigned the value “one” to the A channel.

It is important to mention that when you import a Normal Map to Unity, by default, it is configured in **Texture Type Default** in your project.

Before assigning the Normal Map to your material, you must select the texture, go to the inspector, and set it as **Texture Type Normal Map**, otherwise the program might not work correctly.



(Fig. 6.0.3a)

7.0.1. Lighting model.

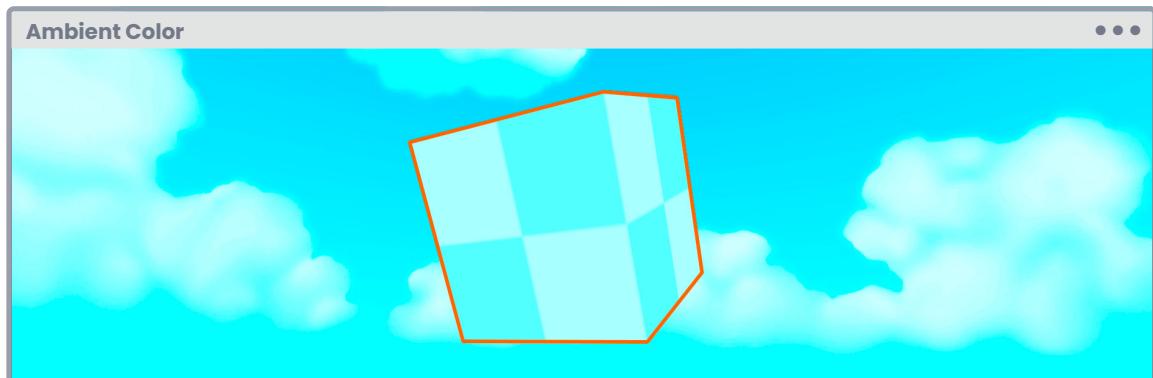
A lighting model refers to the result of the interaction between an object surface and the light source. By definition, it includes the light source properties like color, intensity, etc., and the assigned material properties.

In a shader, illumination can be calculated per-vertex or per-pixel. When calculated by vertex, it is called **per-vertex lighting** and the operation is performed in the **Vertex Shader Stage**, when it is calculated by pixel, it is called **per-fragment or per-pixel lighting** and is performed within the **Fragment Shader Stage**.

In the previous chapter, section 1.1.2, you conceptually defined the function of a Render Path, which corresponds to a series of operations related to object lighting and shading. You also illustrated two types of rendering, Forward Rendering and Deferred Shading. In this section you will recreate a shader using a basic lighting model, and learn about Ambient Color, Diffuse Reflection, and Specular Reflection.

7.0.2. Ambient Color.

A default value that you can find in real life is darkness. This is an interesting point because all objects in their nature are dark and the reason you can differentiate between one surface and another, is due to how lighting interacts with the properties of such a surface. This is why light properties start at "zero," because 0.0f equals "black," that is, their default value.



(Fig. 7.0.2a)

Ambient color refers to the hue of illumination that has been generated by the reflection of multiple light sources. In Computer Graphics, this property derives from a technique known as **Global Illumination**, which itself corresponds to an algorithm capable of calculating indirect lighting to simulate the natural phenomenon of reflected light.

In Unity 2020.3.2f1 you can easily access the Ambient Color from the menu,

➤ Window / Rendering / Lighting.

This path displays the **Lighting** window in which you can find the global lighting properties for your project. In its **Environment** tab there are two properties that are used directly in the shader, these refer to:

- 1 Source.
- 2 And Ambient Color.

To illustrate this, create a new Unlit shader and call it **USB_ambient_color**. This time you will focus on the compression of the internal variable **UNITY_LIGHTMODEL_AMBIENT**, which gives access to the Ambient Color.

Start by declaring a Property to increase or decrease the amount of ambient color in your shader. For this, use a range between 0.0f and 1.0f, where “zero” equals 0% lighting and “one” equals 100%.

```
AmbientColor
...
Shader "USB/USB_ambient_color"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Ambient ("Ambient Color", Range(0, 1)) = 1
    }
    SubShader
    {
        ...
        Pass
        {
            ...
        }
    }
}
```

Continued on the next page.

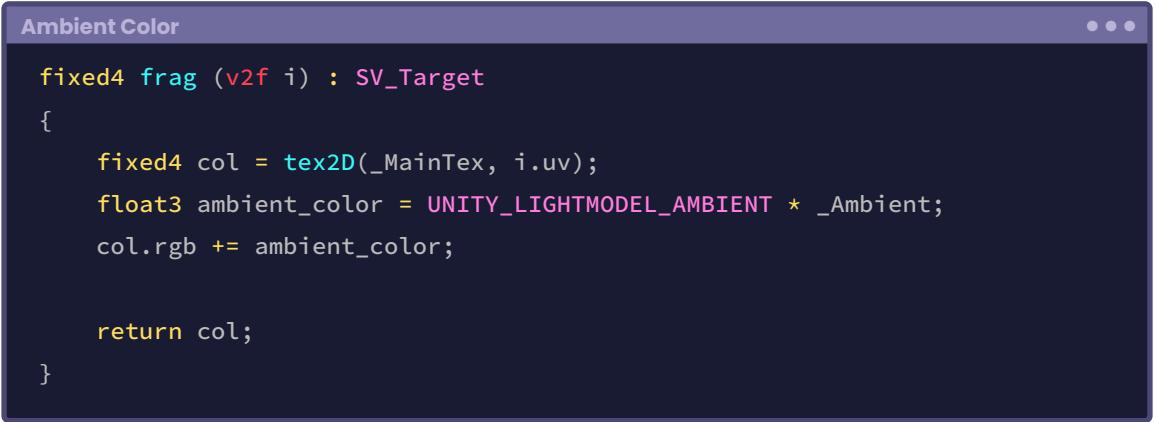
```

CGPROGRAM
...
sampler2D _MainTex;
float4 _MainTex_ST;
float _Ambient;
...
ENCG
}
}
}

```

The **_Ambient** Property controls the amount of ambient color in the program. Its initialization value equals 1.0f, so it will start affecting the entire surface color of the object in the scene.

You will perform the calculation for each pixel on the screen, therefore, go to the **Fragment Shader Stage** and use the internal variable as follows:



A screenshot of the Unity Editor showing the Fragment Shader stage. The code is as follows:

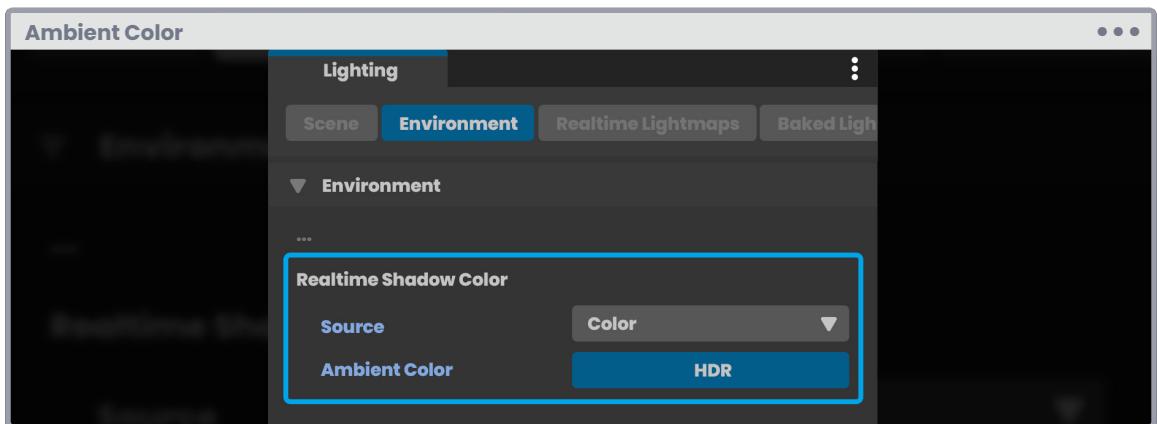
```

Ambient Color
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    float3 ambient_color = UNITY_LIGHTMODEL_AMBIENT * _Ambient;
    col.rgb += ambient_color;

    return col;
}

```

In the previous exercise, the factor between the ambient color and the **_Ambient** property was stored in the vector `ambient_color` and then finally, the RGB ambient color was added to the "col" texture. Up to this point your ambient color is already working. Now simply go to the **Lighting** window, **Environment** tab, and set the **Source** property to **Color**, and then select an ambient color from the **Ambient Color** property.



(Fig. 7.0.2b)

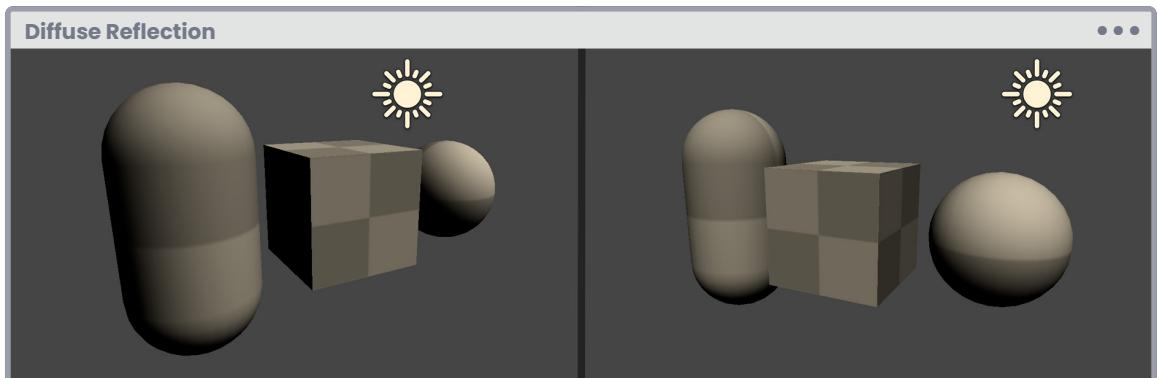
It is worth mentioning that the **Ambient Color** property corresponds to an HDR (High Dynamic Range) color by default, that is, high precision. That is why the three-dimensional vector **ambient_color** was declared as a “float3” type vector in the previous example.

7.0.3. Diffuse Reflection.

Generally, a surface can be defined by two types of reflection:

- Matte.
- Or gloss.

Diffuse Reflection obeys Lambert's cosine law, created by Johann Heinrich Lambert, who makes an analogy between illumination and the surface of an object, considering the light source direction and the surface Normal.



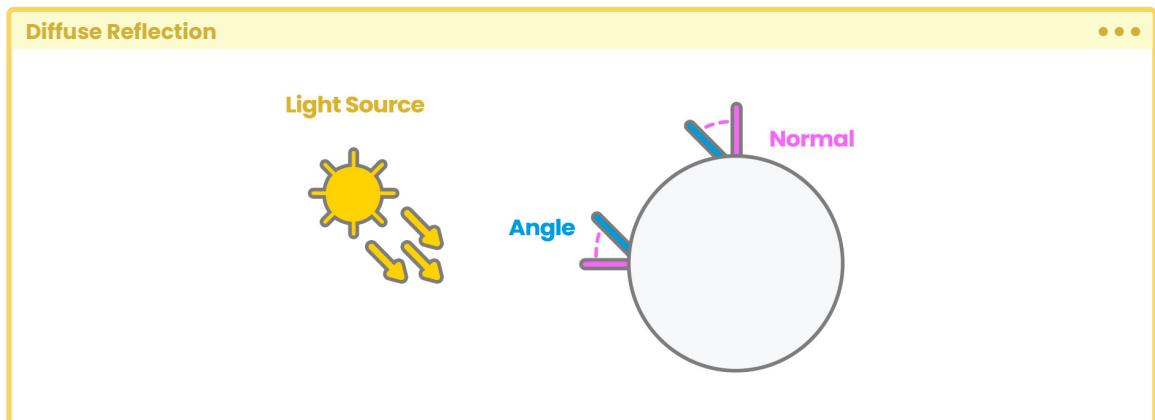
(Fig. 7.0.3a)

In Maya 3D you can find a material called Lambert which adds **Diffuse Reflection** by default. This same concept is applied in Blender, with the difference that the material is called Diffuse, however, both perform the same operation and the result is quite similar, with Render Pipeline architecture variations according to each software.

According to Johann Heinrich Lambert, to obtain a perfect diffusion you must carry out the following operation:

$$D = D_R \cdot D_L \max(0, N \cdot L)$$

How does this equation translate into code? To answer this question, you must first understand the analogy between the illumination direction and a surface Normal. Assume that you have a Sphere and a directional light pointing towards it as you can see in Figure 7.0.3b.



(Fig. 7.0.3b)

The diffusion is calculated according to the angle between the surface Normal [N] and the lighting direction [L], which in fact, corresponds to the dot product between these two properties. However, there are other calculations to consider given the nature of reflection, these refer to [D_R] and [D_L] which correspond to the amount of reflection in terms of color and intensity.

Consequently, the above equation can be translated as follows:

Diffusion [D] equals the multiplication of the reflection color of the light source [Dr] by its intensity [DI], and the maximum (max) between zero [0] and the result of the dot product between the surface normal and the lighting direction [N · L].

To understand this definition better, create a new Unlit shader and call it **USB_diffuse_shading**. Start by creating a function, calling it **LambertShading** and include the properties mentioned above, so it operates correctly.

```
Diffuse Reflection
...
Shader "USB/USB_diffuse_shading"
{
    Properties { ... }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            float3 LambertShading() { ... }
            ...
            ENDCG
        }
    }
}
```

Diffuse Reflection



```
// internal structure of the LambertShading function
float3 LambertShading
(
    float3 colorRefl, // Dr
    float lightInt,   // Dl
    float3 normal,    // n
    float3 lightDir   // l
)
{
    return colorRefl * lightInt * max(0, dot(normal, lightDir));
}
```

The **LambertShading** function returns a three-dimensional vector for its RGB colors. As an argument it has used,

- The light reflection color (`colorRefl`).
- The light intensity (`lightInt`).
- The surface Normals (`normal`).
- And the lighting direction (`lightDir`).

It is worth pointing out that, both the Normals and the lighting direction will be calculated in World-Space, therefore, you will have to transform them in the **Vertex Shader Stage**.

For lighting, it is not necessary to generate a transformation because you can use the internal variable `_WorldSpaceLightPos0` which refers to the direction of directional light in World-Space, included by default in Unity.

Because the intensity of light can be “zero” or “one,” start its implementation by going to Properties and declaring a range called `_LightInt`.

Diffuse Reflection

```
Shader "USB/USB_diffuse_shading"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _LightInt ("Light Intensity", Range(0, 1)) = 1
    }
    SubShader { ... }
}
```

You use **_LightInt** to increase or decrease light intensity in your **LambertShading** function. As part of the process, you must then declare an internal variable to generate the connection between the property and your program.

Diffuse Reflection

```
Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _LightInt;
    ...
    ENDCG
}
```

Now **_LightInt** is configured, next you must implement the **LambertShading** function in the Fragment Shader Stage. To do this, go to the stage, declare a new three-dimensional vector, call it “diffuse” and make it equal to the **LambertShading** function.

Diffuse Reflection

```

float3 LambertShading() { ... }

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // LambertShading(1, 2, 3, 4);
    half3 diffuse = LambertShading( 0, _LightInt, 0, 0);

    return col;
}

```

As you already know, the **LambertShading** function has four arguments, which are:

- 1 float3 colorRefl.
- 2 float lightInt.
- 3 float3 normal.
- 4 And float3 lightDir.

In this function you have initialized these arguments to “zero” except for “lightInt” which, given its nature, must be replaced by the property `_LightInt` in the second box.

Next continue with the reflection color, you can use the internal variable `_LightColor[n]` for this, which refers to the scene lighting color. To use it, you must first declare it as a uniform variable within the CGPROGRAM as follows:

Diffuse Reflection

```

Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _LightInt;

```

Continued on the next page.

```

float4 _LightColor0;

...
ENDCG
}

```

Now you can use it as the first argument in the **LambertShading** function. To do this, declare a new three-dimensional vector using only its RGB channels.

Diffuse Reflection

```

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    fixed3 colorRefl = _LightColor0.rgb;
    half3 diffuse = LambertShading( colorRefl, _LightInt, 0, 0);

    return col;
}

```

As has already been mentioned, **WorldSpaceLightPos[n]** refers to the lighting direction in World-Space. Unlike **_LightColor[n]**, it is not necessary to declare it as a uniform vector since it has already been initialized previously in **UnityCG.cginc**, so it can be used directly as an argument.

Diffuse Reflection

```

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    fixed3 colorRefl = _LightColor0.rgb;
    half3 diffuse = LambertShading( colorRefl, _LightInt, 0, lightDir);

    return col;
}

```

In the example above, a three-dimensional vector called **lightDir** has been declared to save the lighting direction values in its XYZ coordinates. In addition, the function has been normalized so that the resulting vector has a magnitude of "one." Finally, the lighting direction has been positioned as the fourth argument.

Only the third argument that corresponds to the World-Space object Normals is missing, for this you have to include this semantic in both the **Vertex Input** and the **Vertex Output**.

```
Diffuse Reflection
CGPROGRAM
...
// vertex input
struct appdata
{
    float4 vertex : POSITION;
    float2 UV : TEXCOORD0;
    float3 normal : NORMAL;
};

// vertex output
struct v2f
{
    float2 UV : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 normal_world : TEXCOORD1;
};

...
ENDCG
```

Remember that the reason why you are configuring the Normals in both the Vertex Input and Output is precisely because you will use them in the **Fragment Shader Stage**, where your LambertShading function has been initialized, however, their connection will be made in the **Vertex Shader Stage** to optimize the transformation process.

Diffuse Reflection

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    o.normal_world = normalize(mul(unity_ObjectToWorld,
        float4(v.normal, 0))).xyz;

    return o;
}
```

The normalized factor between the `unity_ObjectToWorld` matrix and the object Normal input has been stored in the vector **normal_world** and the object Normal input in World-Space.

Now you can go to the **Fragment Shader Stage**, declare a new three-dimensional vector, and pass the Normal output as the third argument in the **LambertShading** function.

Diffuse Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 normal = i.normal_world;
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    fixed3 colorRefl = _LightColor0.rgb;
    half3 diffuse = LambertShading(colorRefl, _LightInt, normal, lightDir);

    return col;
}
```

As you can see, you have declared a new three-dimensional vector called **normal** to which you have passed the object Normals. Then, the vector has been used as the third argument in the **LambertShading** function.

To configure the Render Path you only need to multiply the diffusion by the texture RGB color, and also; since your shader is interacting with a light source, include the **LightMode ForwardBase**.

Diffuse Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 normal = i.normal_world;
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    fixed3 colorRefl = _LightColor0.rgb;
    half3 diffuse = LambertShading(colorRefl, _LightInt, normal, lightDir);
    // diffuse is included in the texture
    col.rgb *= diffuse;
    return col;
}
```

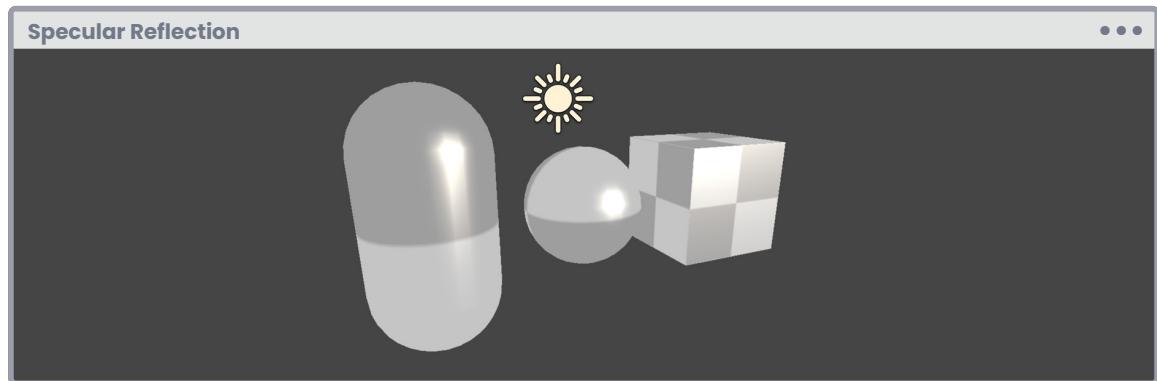
To finish, go to the **Tags** and configure the Render Path.

Diffuse Reflection

```
SubShader
{
    Tags { "RenderType"="Opaque" }
    Pass
    {
        Tags { "LightMode"="ForwardBase" }
    }
}
```

7.0.4. Specular Reflection.

One of the most common reflection models in Computer Graphics is the **Phong** model (Bui Tuong Phong), which adds specular brightness to a surface according to the position of its Normals. In fact, in Maya 3D there is a material with this name and its function is to generate shiny surfaces.



(Fig. 7.0.4a)

According to its author, to add specular reflection, you must carry out the following operation:

$$S = S_A \cdot S_P \max(0, H \cdot N)^2$$

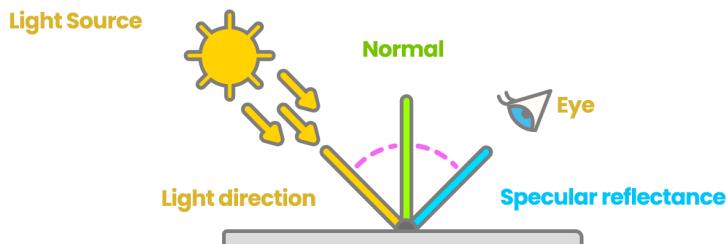
Note that this equation is very similar to the function that calculates diffuse reflection.

$$D = D_R \cdot D_L \max(0, N \cdot L) \text{ Diffuse reflection.}$$

The big difference lies in the vector [H] calculation which corresponds to a half vector called **halfway**. This allows us to appreciate the brightness of the reflection when it is close to [N]; where the latter corresponds to the surface Normals.

To understand the concept, begin your study by demonstrating Specular Reflection, take a flat surface and directional light pointing towards it as in Figure 7.0.4b.

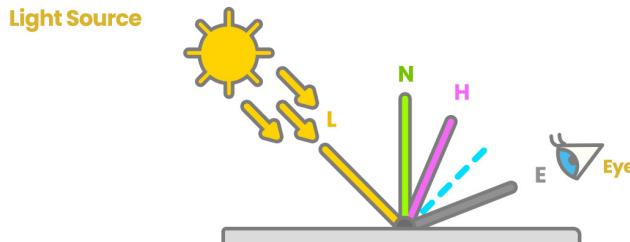
Specular Reflection



(Fig. 7.0.4b)

From the image above, you can deduce that Specular Reflection has the same angle as the light direction. This creates a problem that, if your eye/camera is not in the same direction of reflection, you will not be able to see it. To solve this, you can calculate an intermediate vector between the Normals and the light direction, following the view direction.

Specular Reflection



(Fig. 7.0.4c)

Vector [E] corresponds to the “view direction,” while vector [H] is the halfway value that you have calculated between the light direction and the surface normals. To determine the value of the vector [H] perform the following function.

$$H = \frac{L + E}{\| L + E \|}$$

It is worth mentioning that for your program you are going to use normalized vectors, this means that their magnitude will equal “one,” therefore, the previous operation can be reduced to the following function:

$H = \text{normalize}(L + E)$.

In the reflection calculation, there will be at least three variables that you will have to add in the code which are:

- 1 Lighting direction.
- 2 Surface Normals.
- 3 And the halfway value that includes the view direction.

Additionally, if you want to add Specular Maps, you will have to calculate the reflection color.

You will start a new program to review these concepts, for this, create an Unlit shader which you will call **USB_specular_reflection**. It is worth mentioning that many of the operations performed in this section are the same as those carried out in **USB_diffuse_shading**, so you can start from scratch or continue from the shader that you developed in the previous section.

Within your program, you will create a function called **SpecularShading**. Within this, you will include the properties mentioned above as follows:

```
Specular Reflection
Shader "USB/USB_specular_reflection"
{
    Properties { ... }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
        }
    }
}
```

Continued on the next page.

```

        // declare the function in the program
        float3 SpecularShading() { ... }

        ...
    ENDCG
}

}
}
```

Specular Reflection

```

// internal structure of the SpecularShading function
float3 SpecularShading
(
    float3 colorRefl, // Sa
    float specularInt, // Sp
    float3 normal, // n
    float3 lightDir, // l
    float3 viewDir, // e
    float specularPow // exponent
)
{
    float3 h = normalize(lightDir + viewDir); // halfway

    return colorRefl * specularInt * pow(max(0, dot(normal, h)), specularPow);
}
```

This time you have declared a function called **SpecularShading** that returns a vector of three dimensions for its RGB colors. Within its arguments you can find,

- 1 The reflectance color (`colorRefl`).
- 2 The specularity intensity (`specularInt`).
- 3 The surface Normals (`normal`).
- 4 The lighting direction (`lightDir`).
- 5 The view direction (`viewDir`).
- 6 And the specularity exponent (`specularPow`).

In the same way as the Diffuse Reflectance calculation, the Normals, view direction, and illumination will be calculated in World-Space, therefore, you will have to make some transformations in the Vertex Shader Stage.

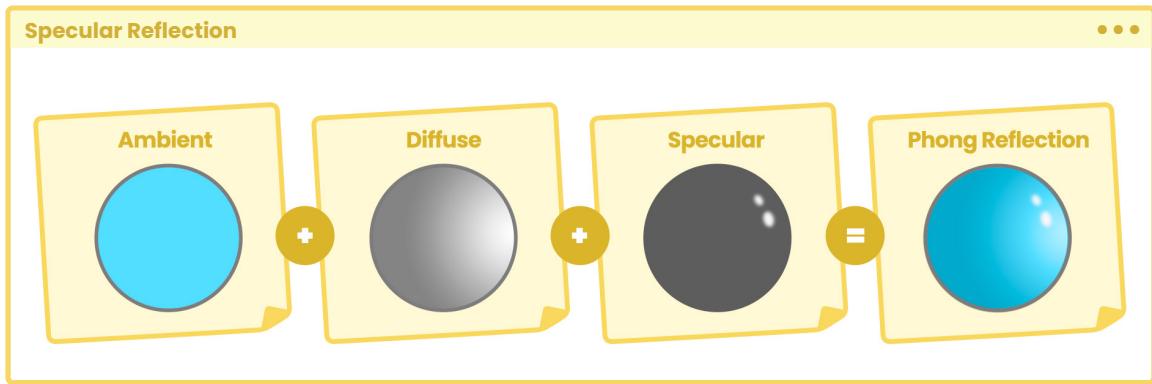
Start by configuring three properties for your shader,

- 1 A texture property for the specular map.
- 2 A range between 0.0f and 1.0f for the reflectance intensity.
- 3 And a new range between 1 and 128 for the specularity exponent.

```
Specular Reflection •••
Shader "USB/USB_specular_reflection"
{
    Properties
    {
        // mode "white"
        _MainTex ("Texture", 2D) = "white" {}
        // mode "black"
        _SpecularTex ("Specular Texture", 2D) = "black" {}
        _SpecularInt ("Specular Intensity", Range(0, 1)) = 1
        _SpecularPow ("Specular Power", Range(1, 128)) = 64
    }
}
```

Unlike the `_MainTex` property, `_SpecularTex` is color by default. This can be corroborated in the definition **black** found at the end of the declaration. Its operation is quite simple: If you do not assign a texture from the Inspector, then the object will look black.

Note that specularity is going to be added to the main texture, therefore, for this case black will not be visible graphically because, as you already know, black equals “zero,” and 0.0f plus 1.0f equals “one.”



(Fig. 7.0.4d)

Next, you must declare the connection variables for the three Properties that you have added.

```
Specular Reflection
...
Pass
{
    CGPROGRAM
    ...
    sampler2D _MainTex;
    float4 _MainTex_ST;
    sampler2D _SpecularTex;
    // float4 _SpecularTex_ST;
    float _SpecularInt;
    float _SpecularPow;
    float4 _LightColor0;
    ...
    ENDCG
}
```

Why has the variable `_SpecularTex_ST` been discarded in the above example? As you already know, the connection variables ending in the suffix `_ST` add Tiling and Offset to their texture. In the case of `_SpecularTex` it will not be necessary to add this type of transformation because, generally, textures or sSpecular Maps do not need them due to their consistent nature.

Another connection variable that you have generated is `_LightColor[n]`. This variable will be used to multiply the color result of `_SpecularTex`, in this way, the specular color will be affected by the light source color that you have in your scene.

The properties are now functional, so you can start implementing the **SpecularShading** function in the Fragment Shader Stage.

Start by calculating the reflection color.

```
Specular Reflection ...  
float3 SpecularShading() { ... }  
  
fixed4 frag (v2f i) : SV_Target  
{  
    fixed4 col = tex2D(_MainTex, i.uv);  
  
    fixed3 colorRefl = _LightColor0.rgb;  
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;  
  
    half3 specular = SpecularShading(specCol, 0, 0, 0, 0, 0);  
  
    return col;  
}
```

The first argument in the SpecularShading function corresponds to reflection color. To do this, you multiplied the texture `_SpecularTex` by the lighting color (`colorRefl`), and the factor was stored within a three-dimensional vector called **specCol**, which is assigned as the first argument.

The second argument corresponds to specular intensity, for this, simply assign the property `_SpecularInt`, which is a range between 0.0f and 1.0f.

Specular Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    fixed3 colorRefl = _LightColor0.rgb;
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;

    half3 specular = SpecularShading(specCol, _SpecularInt, 0, 0, 0, 0);

    return col;
}
```

The third argument in the function refers to the object Normals in World-Space. To do this, you have to add the corresponding semantics in both the **Vertex Input** and **Output** and then transform their space in the **Vertex Shader Stage**, in the same way as in the Diffuse Reflection calculation.

Specular Reflection

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Then you must assign the Normals in the Vertex Output. However, you must remember that the NORMAL semantic does not exist for this process, therefore, you must use TEXCOORD[n] as it has up to four dimensions.

Specular Reflection

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 normal_world : TEXCOORD1;
};
```

Before returning to the Fragment Shader Stage, create one last semantic in the Vertex Output. This will be used later as a reference point in the view direction calculation.

Specular Reflection

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 normal_world : TEXCOORD1;
    float3 vertex_world : TEXCOORD2;
};
```

As you can see, a new property called **vertex_world** has been included, which refers to the position of the object vertices in World-Space.

Then go to the **Vertex Shader Stage** to transform the coordinates' space, however, unlike the previous processes, now use the **UnityObjectToWorldNormal** function to transform the Normals from Object-Space to World-Space.

Specular Reflection

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o, o.vertex);
    o.normal_world = UnityObjectToWorldNormal(v.normal);
    o.vertex_world = mul(unity_ObjectToWorld, v.vertex);
    return o;
}
```

UnityCg.cginc includes the **UnityObjectToWorldNormal** function, which is equivalent to inversely multiplying the `unity_ObjectToWorld` matrix by the object Normals input. Next you can look at its internal structure.

Specular Reflection

```
inline float3 UnityObjectToWorldNormal(in float3 norm)
{
    #ifdef UNITY_ASSUME_UNIFORM_SCALING
        return UnityObjectToWorldDir(norm);
    #else
        return normalize(mul(norm, (float3x3) unity_WorldToObject));
    #endif
}
```

One factor to consider is that **normal_world** is normalizing the transformation operation. This is because normals are a direction of space; a three-dimensional vector returning a maximum magnitude of "one," while **vertex_world** remains a position in space, with the difference now being calculated in World-Space.

Continue with the SpecularShading function.

Specular Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // implement the Normals in World-Space.
    float3 normal = i.normal_world;
    fixed3 colorRefl = _LightColor0.rgb;
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;

    half3 specular = SpecularShading(specCol, _SpecularInt, normal, 0, 0, 0);

    return col;
}
```

As you can see, a new three-dimensional vector called **normal** has been created. This vector has its Normals output in World-Space, that's why it has been assigned as the third argument in the function.

It will not be necessary to generate any kind of lighting transformation because you can use the internal variable `_WorldSpaceLightPos[n]`, which refers to the light direction in World-Space.

Specular Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // let's calculate the light direction
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    float3 normal = i.normal_world;
    fixed3 colorRefl = _LightColor0.rgb;
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl
    half3 specular = SpecularShading(specCol, _SpecularInt, normal,
    lightDir, 0, 0);

    return col;
}
```

Now you only need to calculate the view direction since the last argument in the `SpecularShading` function corresponds to the exponential value which increases or decreases reflection.

To calculate the view direction, you must subtract the object vertices in World-Space from the camera also in World-Space. Unity has an internal variable called `_WorldSpaceCameraPos`, which gives you precise access to the scene's camera position.

Specular Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // calculate light direction
    float3 viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    float3 normal = i.normal_world;
    fixed3 colorRefl = _LightColor0.rgb;
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;
```

Continued on the next page.

```
// pass the view direction to the function
half3 specular = SpecularShading(specCol, _SpecularInt, normal,
lightDir, viewDir, _SpecularPow);

return col;
}
```

The only operation left is to add specularity to the main texture, to do this perform the following operation:

Specular Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
    half3 normal = i.normal_world;
    fixed3 colorRefl = _LightColor0.rgb;
    fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;

    half3 specular = SpecularShading(specCol, _SpecularInt, normal,
    lightDir, viewDir, _SpecularPow);
    // let's add the specularity to the texture
    col.rgb += specular;
    return col;
}
```

Remember that you cannot add a four-dimensional vector to a three-dimensional vector, so you must make sure that specular reflection RGB channels are only added to the main texture.

Given that reflection is a lighting pass, you must once again go to the **Tags** and configure the Render Path in the same way as you did for Diffuse Reflection.

Specular Reflection

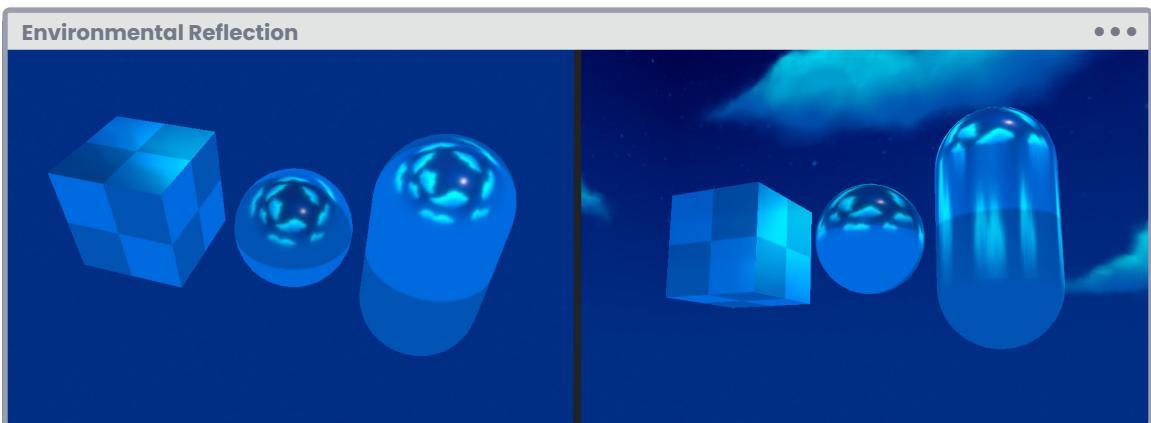
```

Shader "USB/USB_specular_reflection"
{
    Properties { ... }
    SubShader
    {
        Tags
        {
            "RenderType"="Opaque"
            "LightMode"="ForwardBase"
        }
    }
}

```

7.0.5. Environmental Reflection.

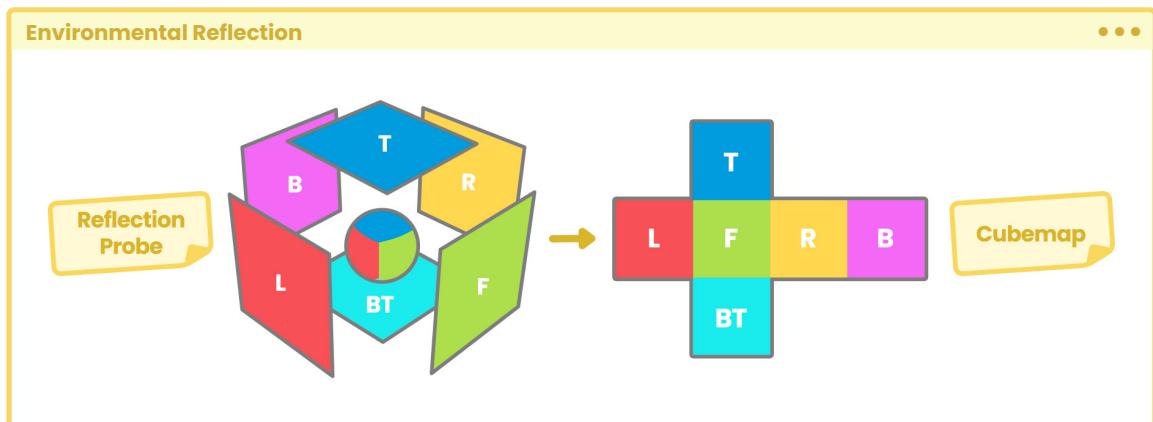
Ambient Reflection occurs similarly to Specular Reflection. Its difference lies in the number of light rays that affect a surface, e.g., in a generic scene, Specular Reflection is only generated by the main light source, while Ambient Reflection is generated by each light ray hitting a surface, including bouncing from all angles.



(Fig. 7.0.5a)

Given its nature, calculating this type of reflection in real time uses a lot of GPU power, instead, you can use a **Cubemap** type texture. In Chapter I, section 3.0.6, the **Cube** property was mentioned, which refers precisely to this type of texture.

In Unity, you can generate Cubemaps using the **Reflection Probe** component. This object works like a camera, which captures a spherical view of its surroundings in all directions and then generates a Cube-type texture that you can use as a reflection map.



(Fig. 7.0.5b)

To see its implementation in detail, create a new Unlit shader called **USB_cubemap_reflection**. Start by declaring a new function to use later for generating reflections.

```
Environmental Reflection
...
Shader "USB/USB_cubemap_reflection"
{
    Properties { ... }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            ...
            // add the function in the program
            float3 AmbientReflection () { ... }
            ...
            ENDCG
        }
    }
}
```

Environmental Reflection

• • •

```
// internal structure of the AmbientReflection function
float3 AmbientReflection
(
    samplerCUBE colorRefl,
    float reflectionInt,
    half reflectionDet,
    float3 normal,
    float3 viewDir,
    float reflectionExp
)
{
    float3 reflection_world = reflect(viewDir, normal);
    float4 cubemap = texCUBEElod(colorRefl, float4(reflection_world,
    reflectionDet));

    return reflectionInt * cubemap.rgb * (cubemap.a * reflectionExp);
}
```

The first argument of the **AmbientReflection** function corresponds to a sampler for a **Cube** type texture, which supposes the creation of a Property of this type. Then there is a variable called **reflectionInt**, which is later used to modify the reflection intensity within a range between “zero and one.” The third argument corresponds to a medium precision variable called **reflectionDet**, which is used to increase or decrease the samplerCUBE **texel** density.

If you look closely, you will notice that **reflectionDet** has been included in the **texCUBEElod(C_{RG}, S_{RG})** method as follows:

Environmental Reflection

• • •

```
texCUBEElod(colorRefl, float4(reflection_world, reflectionDet));

// float4 texCUBEElod(samplerCUBE samp, float4 s)
// s.xyz = reflection coordinates
// s.w = texel density
```

The **texCUBElod** method has two default arguments: the first refers to the samplerCUBE that you are going to use as a texture, and the second corresponds to a four-dimensional vector that has been divided into two parts; the first three channels correspond to the reflection coordinates in World-Space (XYZ), while the last value corresponds to the level of detail of the texels of the samplerCUBE (W).

In the same way as Specular Reflection, you must calculate both the Normals and the view direction in World-Space. This is why these vectors were included as arguments in the function.

To finish, as a final argument, there is a variable called **reflectionExp**, which refers to the Reflection Map color exposure.

A function that you will frequently use in the reflection calculation is **reflect(I_{RG}, N_{RG})**. This operation included in both Cg and HLSL is composed as follows:

Environmental Reflection

• • •

```
float3 reflect (float3 i, float3 n)
{
    return i - 2.0 * n * dot(n, i);
}

float3 reflection_world = reflect(viewDir, normal);
```

The first argument I_{RG} refers to the incidence value, that is, the view direction, while N_{RG} are the object Normals.

It should be noted that in the internal operation of the “reflect” function, the incidence value is being calculated in the direction of the reflection point, which will cause the Reflection Map to be graphically flipped; as if we were looking at the reflection through a concave lens. To solve this you have to make the incidence value negative.

Now that you understand a great deal of the operation, start implementing it in the Fragment Shader Stage. To do this create a three-dimensional vector and pass it through the **AmbientReflection** function as follows:

Environmental Reflection

```

float3 AmbientReflection() { ... }

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    half3 reflection = AmbientReflection(0, 0, 0, 0, 0, 0);

    return col;
}

```

As you already know, the first argument in the function corresponds to the Cube type reflection color, so you need to go to the shader Properties and declare the texture along with the intensity, detail, and exposure variables.

Environmental Reflection

```

Shader "USB/USB_cubemap_reflection"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}

        _ReflectionTex ("Reflection Texture", Cube) = "white" {}
        _ReflectionInt ("Reflection Intensity", Range(0, 1)) = 1
        _ReflectionMet ("Reflection Metallic", Range(0, 1)) = 0
        _ReflectionDet ("Reflection Detail", Range(1, 9)) = 1
        _ReflectionExp ("Reflection Exposure", Range(1, 3)) = 1
    }
    SubShader { ... }
}

```

Among the properties that have been declared, you can find **_ReflectionMet** which has not been included in the **AmbientReflection** function. As its name says, you use this property to control the reflection shininess and thus emulate a metal surface.

Continue by generating the connection variables for these properties.

Environmental Reflection

```
CGPROGRAM  
...  
sampler2D _MainTex;  
float4 _MainTex_ST;  
samplerCUBE _ReflectionTex;  
float _ReflectionInt;  
half _ReflectionDet;  
float _ReflectionExp;  
float _ReflectionMet;  
...  
ENDCG
```

Since texture sampling occurs within the `AmbientReflection` function, you can pass the `_ReflectionTex` property directly as the first argument in the function declaration.

Environmental Reflection

```
fixed4 frag (v2f i) : SV_Target  
{  
    fixed4 col = tex2D(_MainTex, i.uv);  
    // add the cubemap  
    half3 reflection = AmbientReflection(_ReflectionTex, 0, 0, 0, 0, 0);  
  
    return col;  
}
```

You can also add the second and third `reflectionInt` and `reflectionDet` arguments to the function.

Environmental Reflection

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // we add the intensity and detail of the reflection
    half3 reflection = AmbientReflection(_ReflectionTex, _ReflectionInt,
    _ReflectionDet, 0, 0, 0);

    return col;
}
```

The fourth and fifth arguments correspond to the Normals and the view direction, both in World-Space. For this, do exactly the same as you performed in the Specular Reflection, that is, include the Normals in the Vertex Input and Output, then declare the view direction in the Vertex Output.

Environmental Reflection

```
// vertex input
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};

// vertex output
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 normal_world : TEXCOORD1;
    float3 vertex_world : TEXCOORD2;
};
```

Now simply transform their space coordinates from Object-Space to World-Space in the Vertex Shader Stage.

Environmental Reflection

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    o.normal_world = normalize(mul(unity_ObjectToWorld,
        float4(v.normal, 0))).xyz;
    o.vertex_world = mul(unity_ObjectToWorld, v.vertex);

    return o;
}
```

Continuing with the **AmbientReflection** function, create a new vector in which you assign the Normals and then carry out the view direction calculation. However, this time you will use the **UnityWorldSpaceViewDir** function included in UnityCg.cginc, which is equivalent to the camera position and object vertices calculation.

Environmental Reflection

```
// included function in UnityCg.cginc
inline float3 UnityWorldSpaceViewDir( in float3 worldPos)
{
    return _WorldSpaceCameraPos.xyz - worldPos;
}

// our function
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    half3 normal = i.normal_world;
    half3 viewDir = normalize(UnityWorldSpaceViewDir(i.vertex_world));
    // we add the normals and view direction
    half3 reflection = AmbientReflection(_ReflectionTex, _ReflectionInt,
        _ReflectionDet, normal, -viewDir, 0);

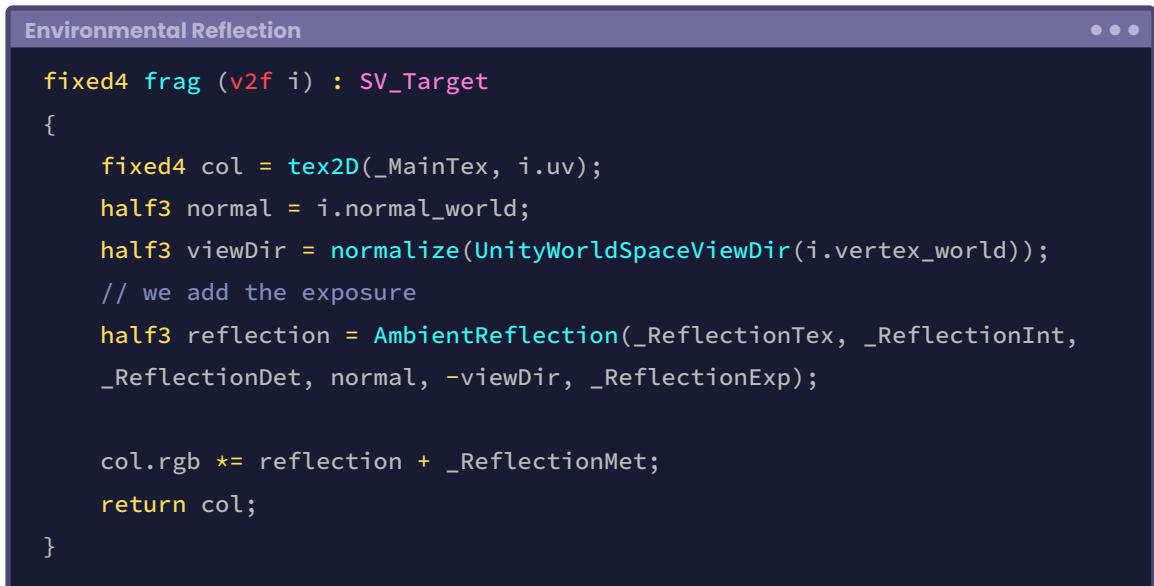
    return col;
}
```

Notice that the fifth argument; corresponding to the view direction is negative, basically, in the direction of the incidence vector, why is this? Making its value negative will allow the reflection to work perfectly for this case.



(Fig. 7.0.5c)

As a final operation, it is necessary to add the sixth argument corresponding to the reflection exposure, and in addition, add the total reflection to the RGB color of the main texture.



In the previous example, we multiply the RGB texture color by the reflection and then add the property **_ReflectionMet**, which corresponds to a range between 0.0f and 1.0f. To understand this operation, we must pay attention to the property **_ReflectionInt** which is a range as well.

Since the “reflection” vector is being multiplied by “col.rgb,” the resulting color will be that of a metallic surface. Now, add it to **_ReflectionMet** to lighten the final color of the surface and thus obtain reflection variations.

A different way of creating reflection is through the macro **UNITY_SAMPLE_TEXCUBE**. This automatically assigns the environmental reflection that is configured in your scene, this means that, if you have configured a **Skybox** from the Lighting Window then the reflection will be saved as a texture within the shader, and you can use it immediately without the need to independently generate a Cubemap texture.

```
Environmental Reflection ...  
fixed4 frag (v2f i) : SV_Target  
{  
    fixed4 col = tex2D(_MainTex, i.uv);  
    half3 normal = i.normal_world;  
    half3 viewDir = normalize(UnityWorldSpaceViewDir(i.vertex_world));  
    half3 reflect_world = reflect(-viewDir, normal);  
    // the process mentioned above is replaced by the function  
    // UNITY_SAMPLE_TEXCUBE  
    half3 reflectionData = UNITY_SAMPLE_TEXCUBE(unity_SpecCube0,  
    reflect_world );  
    half3 reflectionColor = DecodeHDR(reflectionData , unity_SpecCube0_HDR);  
    col.rgb = reflectionColor;  
    return col;  
}
```

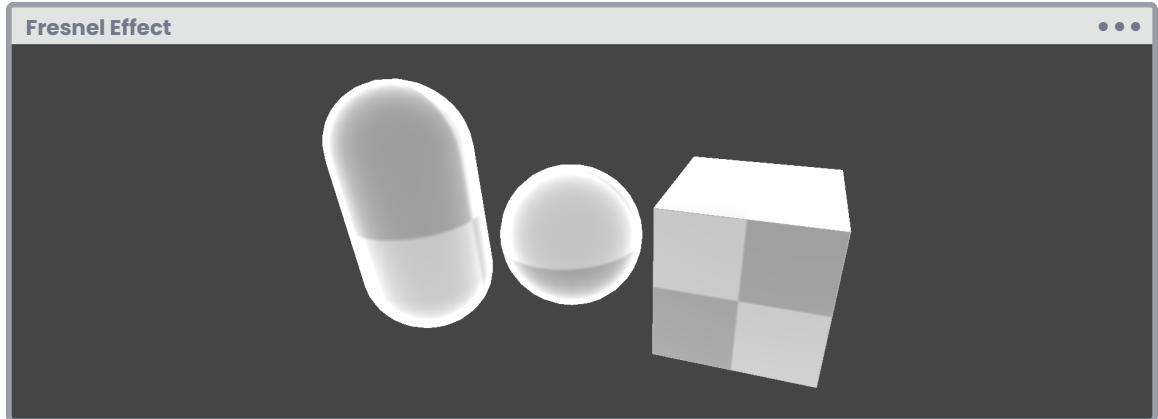
The internal variable **unity_SpecCube[n]** contains the data of the Unity default Reflection Probe object.

The **UNITY_SAMPLE_TEXCUBE** macro samples this data using the reflection coordinates and then decodes the colors in HDR through the **DecodeHDR(D_{RG}, I_{RG})** function, included in **UnityCg.cginc**.

This operation makes it easier to implement this type of reflection but gives less control over the final result.

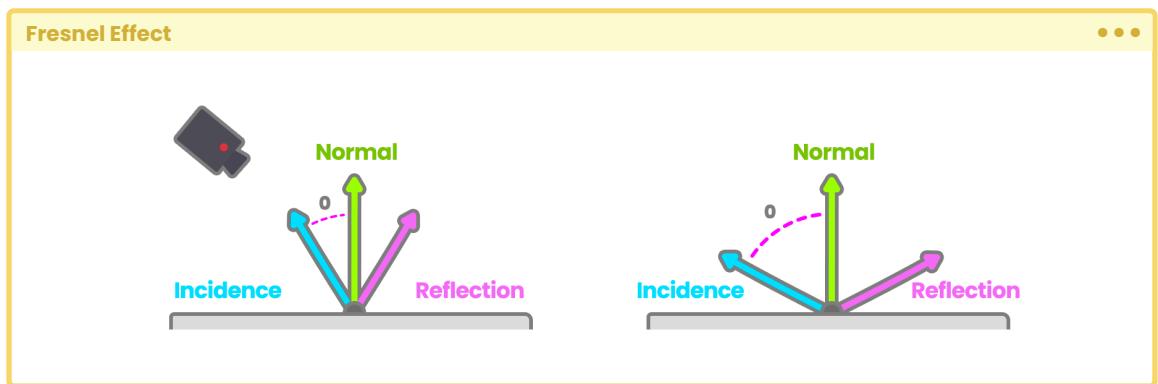
7.0.6. Fresnel Effect.

The **Fresnel** effect (after its creator Augustin Jean Fresnel), also known as the Rim effect, is a type of reflection where its size is proportional to the incidence angle, the angle between the object Normals and the camera direction.



(Fig. 7.0.6a)

The further the surface is from the camera, the more Fresnel reflection there will be because the angle between the incidence value and the object normals is greater.



(Fig. 7.0.6b)

There is no reflection when the angle between the incidence value and the Normals equals 0° because both vectors are parallel. On the other hand, when the angle equals 90° , the reflection is full, and the vectors are perpendicular. This is quite interesting because, when the reflection is null the program must return black. On the contrary, when it is full, it must return white, why? Because these correspond to the maximum and minimum lighting values of a pixel.

To understand this concept, analyze the following function from the **Fresnel Effect node** in the **Shader Graph**.

```
Fresnel Effect
void unity_FresnelEffect_float
(
    in float3 normal,
    in float3 viewDir,
    in float power,
    out float Out
)
{
    Out = pow((1 - saturate(dot(normal, viewDir))), power);
}
```

Several things are happening that will be seen in detail throughout this section, for now, just focus on the output's internal operation. This operation can be divided into three processes:

saturate(dot(normal, viewDir))

This operation determines the angle between the incidence vector and the object Normals, and as a result, returns a numerical range between 0.0f and 1.0f.

As you already know, the dot(A_{RG}, B_{RG}) function will return "one, zero or minus one" depending on the angle between its arguments. Since the reflection operation only requires a value between 0.0f and 1.0f the intrinsic function **saturate(X_{RG})**, has been added which limits the values between this range.

Fresnel Effect

```
// it only can return "0" as minimum and "1" as maximum
float saturate (float x)
{
    return max(0, min(1, x));
}

// it can modify the minimum and maximum range
float clamp (float x, float a, float b)
{
    return max(a, min(b, x));
}
```

Saturate fulfills the same function as **clamp**, with the difference that with the latter you can modify the minimum and maximum value to generate the limit.

Now continue with the operation “ $1 - x$.”

$(1 - x)$

To understand its nature, you must return to the previous exercise. $\text{dot}(\mathbf{A}_{RG}, \mathbf{B}_{RG})$ will return 1.0f when the view direction vector and the Normals are parallel and point in the same direction. This is a problem, because you need the operation to return 0.0f, which is equivalent to black.

Fresnel Effect

`dot (Normal, viewDir)`



(Fig. 7.0.6c)

The operation “ $1 - x$ ” has the function of flipping the result as follows.

Fresnel Effect

```
// if the normals and the view are parallel in the same direction  
saturate(dot(float3(0, 1, 0), float3(0, 1, 0))) = 1  
1 - 1 = 0  
  
// if the normals and the view are perpendiculars  
saturate(dot(float3(0, 1, 0), float3(1, 0, 0))) = 0  
1 - 0 = 1
```

Finally, in the function, you can find the operation $\text{pow}(X_{RG}, N_{RG})$ which allows you to increase or decrease the range of reflection.

To understand the Fresnel operation of Shader Graph in detail, start a new **Unlit** shader and call it **USB_fresnel_effect**. The first thing to do is to include this function in the program.

Fresnel Effect

```
Shader "USB/USB_fresnel_effect"  
{  
    Properties { ... }  
    SubShader  
    {  
        Pass  
        {  
            CGPROGRAM  
            ...  
            void unity_FresnelEffect_float() { ... }  
            ...  
            ENDCG  
        }  
    }  
}
```

It should be noted that the function `unity_FresnelEffect_float` is a **void** type. Section 4.0.4 of Chapter I reviewed the difference between implementing an empty function and one that

returns a value. In this case, you have to declare some variables and pass them as arguments, as appropriate.

Start by declaring the function in the Fragment Shader Stage.

```
Fresnel Effect
void unity_FresnelEffect_float() { ... }

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // initialize the void function
    unity_FresnelEffect_float(0, 0, 0, 0);

    return col;
}
```

The first argument in the function corresponds to the object Normals in World-Space, so you have to go to **Vertex Input** and use the **NORMAL** semantics, and then transform their space coordinates in the **Vertex Shader Stage**.

Because you are using the Normals in the **Fragment Shader Stage**, you have to declare a vector in the Vertex Output, this way you can store the result of the transformation.

```
Fresnel Effect
// vertex input
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Continued on the next page.

```
// vertex output
struct v2f
{
    float4 vertex : SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 normal_world : TEXCOORD1;
    float3 vertex_world : TEXCOORD2;
};
```

As you can see the vector **vertex_world** has been added to the Vertex Output because you need this variable to calculate the view direction. If you look carefully you will notice that the process is exactly the same as you have done in previous sections for the reflection calculation.

Fresnel Effect

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    o.normal_world = normalize(mul(unity_ObjectToWorld, float4(v.normal,
        0))).xyz;
    o.vertex_world = mul(unity_ObjectToWorld, v.vertex);
    return o;
}
```

Continuing with the implementation of the function, go to the Fragment Shader Stage and declare two vectors:

- 1 One for the Normals calculation.
- 2 And the other for the view direction.

Fresnel Effect

```

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 normal = i.normal_world;
    float3 viewDir = normalize(_WorldSpaceCameraPos -
    i.vertex_world);
    // assign the normals and view direction to the function
    unity_FresnelEffect_float(normal, viewDir, 0, 0);

    return col;
}

```

In the example above, a three-dimensional vector called **normal** was declared to store the Normals output value in World-Space. Then a new vector called **viewDir** was declared which contains the view direction calculation. Both vectors were assigned as the first and second arguments in the function **unity_FresnelEffect_float**, since they are required in its internal operation. For the third argument you must declare a property with a numerical range which will be used to modify the reflection range.

Fresnel Effect

```

Shader "USB/USB_fresnel_effect"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _FresnelPow ("Fresnel Power", Range(1, 5)) = 1
        _FresnelInt ("Fresnel Intensity", Range(0, 1)) = 1
    }
    SubShader { ... }
}

```

Use the property **_FresnelPow** as a third argument in the function, as an exponential value; while **_FresnelInt** will be used to increase or decrease the amount of effect in the object. As you already know, you must declare connection variables for both properties.

Fresnel Effect

```
CGPROGRAM  
...  
sampler2D _MainTex;  
float4 _MainTex_ST;  
float _FresnelPow;  
float _FresnelInt;  
...  
ENDCG
```

Once this process is done, the property will be connected to your program, this means that you can dynamically modify the reflection range from the Inspector. Now you can use **_FresnelPow** as the third argument in the function.

Fresnel Effect

```
fixed4 frag (v2f i) : SV_Target  
{  
    fixed4 col = tex2D(_MainTex, i.uv);  
  
    float3 normal = i.normal_world;  
    float3 viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);  
    // add the exponent value in the function  
    unity_FresnelEffect_float(normal, viewDir, _FresnelPow, 0);  
  
    return col;  
}
```

The fourth argument corresponds to the function output value, where you will save the color output. To do this, simply create and add a floating variable to the function.

Fresnel Effect

```
fixed4 frag (v2f i) : SV_Target  
{  
    fixed4 col = tex2D(_MainTex, i.uv);
```

Continued on the next page.

```

float3 normal = i.normal_world;
float3 viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
// initialize the color output in black
float fresnel = 0;
// add the output color
unity_FresnelEffect_float(normal, viewDir, _FresnelPow, fresnel);

col += fresnel * _FresnelInt;
return col;
}

```

In the previous example, a variable called **fresnel** was declared, which was initialized at “zero.” It was then included in the function as the fourth argument (as output), which means that within this variable lies the result of the final operation that occurs in the **unity_FresnelEffect_float** function.

At the end of the operation, you can see that the **fresnel** variable result, due to its intensity, was added to the base texture color called “col.” This adds reflection to the object in your scene and also allows you to modify its intensity.

7.0.7. Structure of a Standard Surface.

Before continuing to define some functions, take a look at the structure of a Standard Surface shader. This program, differently to the Unlit type, is characterized by having a simplified structure, which is configured to interact with lighting only in Built-in RP.

The reflection functions seen in previous sections are included internally in this program, this means that this shader by default has:

- Global lighting.
- Diffusion.
- Reflection.
- And Fresnel.

If you create a **Standard Surface**, you will notice immediately that its structure does not have a pass defined as such, but that the CGPROGRAM is written inside the SubShader field.

To understand how it works pay attention to the **surf** function that would be equivalent to the object surface color output. You can find two arguments inside this function,

- 1 **Input IN.**
- 2 And **inout SurfaceOutputStandard o.**

These refer to the shader Inputs and Outputs, and their semantics have been predefined internally in the code.

```
Structure of a Standard Surface
...
Shader "Custom/SurfaceShader"
{
    Properties { ... }
    SubShader
    {
        CGPROGRAM
        ...
        #pragma surface surf Standard fullforwardshadows
        ...
        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            ...
        }
        ENDCG
    }
}
```

The reason why you can determine that “surf” is the color output function is because it has been declared as such in the **#pragma surface surf**. This process is similar to a Vertex-Fragment Shader, the difference in this case is that you can declare other properties, e.g., the lighting model (Standard) and other optional parameters (fullforwardshadows).

By default, this type of program comes configured with a **Standard** type lighting model, which contains some predefined properties that you can use as color output.

Note that the defined properties in the “surf” function code are of **SurfaceOutputStandard** type, this means that the color output will be determined by the Standard lighting model.

• • •

Structure of a Standard Surface

```
#pragma surface surf Standard
...
struct SurfaceOutputStandard
{
    fixed3 Albedo;
    fixed3 Normal;
    half3 Emission;
    half Metallic;
    half Smoothness;
    half Occlusion;
    fixed Alpha;
};

void surf (Input IN, inout SurfaceOutputStandard o)
{
    fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

7.0.8. Standard Surface Input & Output.

As in a Vertex-Fragment Shader, by default you can find two “struct” type functions in a standard surface, these are:

- 1** Input.
- 2** SurfaceOutputStandard.

The **Struct Input** is different from the **Struct appdata** which was reviewed in the previous Chapter, why? In **appdata** you can define your object’s semantics as an input, while, in **Input** you can determine your shader’s predefined functions for the lighting calculation, what does this mean?

In appdata you can use the semantic POSITION[n] for the position of the mesh vertices in Object-Space, on the other hand, in **Input** you can get the input “viewDir” directly. This, as you already know, corresponds to the view direction in World-Space and is used for the calculation of different lighting functions.

Standard Surface Input & Output

```
struct Input
{
    float2 uv_MainTex;      // TEXCOORD0
    float3 viewDir;         // Standard Surface Input & Output
    float4 color : COLOR;   // vertex color
    float3 worldPos;        // world-space vertices
    float3 worldNormal;     // world-space normals
};
```

As in the reflection calculation in the previous sections, the Input **viewDir** is the same as the function used to determine the view direction in World-Space.

Standard Surface Input & Output

```
viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
```

The same goes for **worldPos** and **worldNormal**, which also refer to the position of the vertices and Normals in World-Space.

Standard Surface Input & Output

```
worldPos = mul(unity_ObjectToWorld, v.vertex);
worldNormal = normalize(mul(unity_ObjectToWorld, float4(v.normal, 0)))xyz;
```

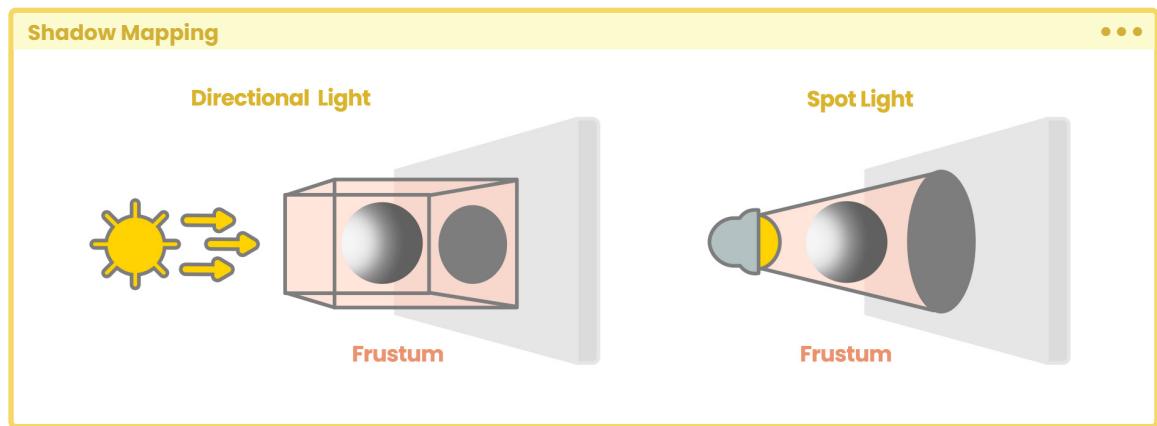
In conclusion, you can use the same properties as in a Vertex-Fragment Shader, their difference lies solely in the internal variables.

Shadow.

8.0.1. Shadow Mapping.

This technique generates Shadow Maps in a scene. Its concept is quite simple: the light and shadow areas are generated in relation to the frustum of illumination that you are using, that is, if the light source corresponds to a directional light, the shadow projection will be orthographic, while if the source corresponds to a point light, then the projection will be rendered in perspective.

This calculation is done by comparing whether a pixel is visible from the light source, as if the source were the projection point. This concept is shown in Figure 8.0.1a.



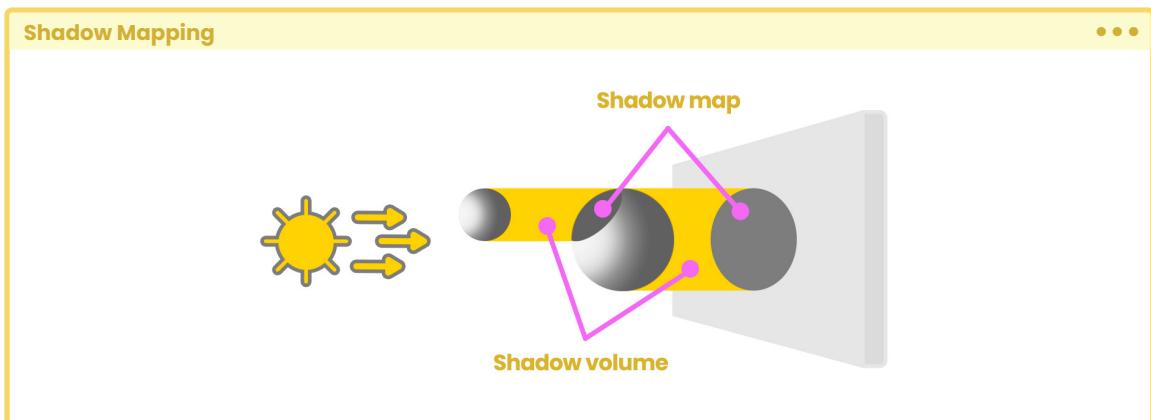
(Fig. 8.0.1a)

When you create a light source in your scene, the entire visible area, according to the light source viewpoint, will be the illuminated area, and all the area outside it will be the shadow zone. According to this logic, you can determine that this corresponds to a comparison operation, but how does this process work?

To do this, you must review two concepts:

- **Shadow Caster.**
- And **Shadow Map.**

The Shadow Caster corresponds to the shadow projection area, while the Shadow Map corresponds to the shadow cast on an object.



(Fig. 8.0.1b)

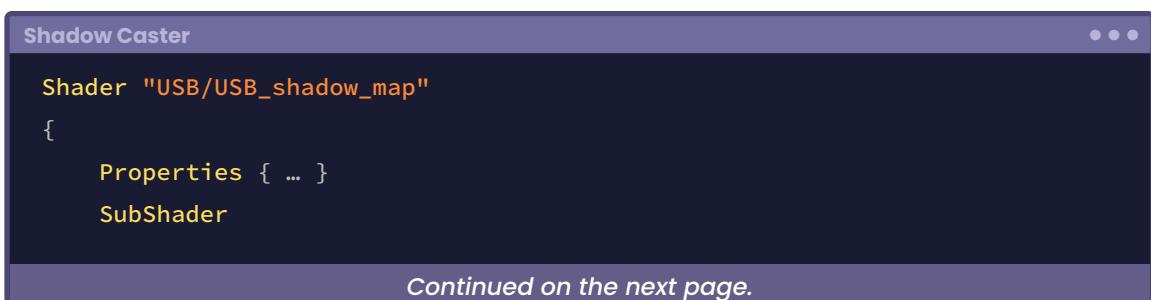
The Shadow Map is a texture; therefore it has UV coordinates and is calculated in two stages: First, the scene is rendered according to the light source viewpoint. During the process, the depth information is extracted from the Z-Buffer and then saved as a texture in the internal memory. In the second, the scene is drawn on the GPU in the usual way according to the camera viewpoint. This is where you must calculate the UV coordinates of the texture saved in memory to generate and apply shadows onto the object you are working with.

8.0.2. Shadow Caster.

Start with generating shadows. For this, create a new Unlit shader and call it **USB_shadow_map**. In the process, you need two passes:

- One to cast shadows (Shadow Caster).
- And another to receive them (Shadow Map).

Therefore, the first thing you must do is include a second pass, which will be responsible for the shadow projection.



```
{
    Tags { "RenderType"="Opaque" }
    LOD 100
    // shadow caster pass
    Pass { ... }
    // default color pass
    Pass { ... }
}
}
```

The color pass corresponds to the default **Pass** that is included every time you create a shader, whereas the new **Pass** will be responsible for generating the shadow projection, therefore, you have to declare the projection pass as **ShadowCaster**.

Shadow Caster

```
// shadow caster pass
Pass
{
    Name "Shadow Caster"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ShadowCaster"
    }
    ...
}
```

The Shadow Caster Pass begins with the declaration of its name (Name “Shadow Caster”) and continues with the **LightMode** Tag, which in this case, must equal **ShadowCaster** in order for Unity to recognize its nature.

Naming a Pass is a great help when you want to use its functionality dynamically in a shader. Later the **UsePass** command will be reviewed in detail, which is directly related to this concept.

It is worth mentioning that the **Name** only fulfills the function of naming in the shader and does not interfere with the process of calculating the projection. The name declaration eventually can be omitted, however this time it is being used to differentiate the two passes.

Because the ShadowCaster only corresponds to a shadow projection, you must pass the vertex position to the Vertex Shader Stage and return "zero" in the Fragment Shader Stage.

Shadow Caster

```
// shadow caster Pass
Pass
{
    Name "Shadow Caster"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ShadowCaster"
    }
    ZWrite On

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    struct appdata
    {
        // we need only the position of vertices as input
        float4 vertex : POSITION;
    };

    struct v2f
    {
        // we need only the position of vertices as output
        float4 vertex : SV_POSITION;
    };
}
```

Continued on the next page.

```

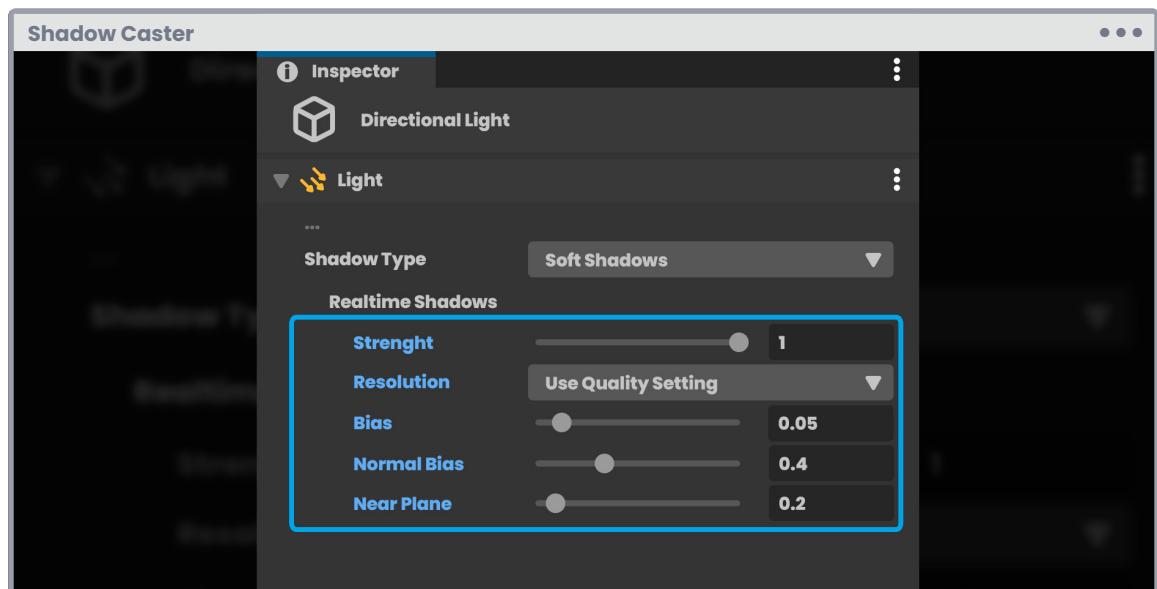
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    return 0;
}
ENDCG
}

```

Up to this point, the shadow caster is working correctly, however this configuration will not allow you to adjust the projection values since they have not been defined yet.

When you work with shadows you can determine the values of intensity, resolution, Bias, Normal Bias and Near Plane. These properties can be found in the lighting configuration.



(Fig. 8.0.2a)

Defining each property in your shader could take a long time, to avoid this you can work with the following macros that are included in UnityCG.cginc:

- 1 V2F_SHADOW_CASTER.
- 2 TRANSFER_SHADOW_CASTER_NORMALOFFSET(O_{RG}).
- 3 SHADOW_CASTER_FRAGMENT(I_{RG}).

V2F_SHADOW_CASTER contains several semantics for shadow calculation both in the interpolated vertices position and in the Normal Maps, this means that this macro has:

- A vertex position output (`vertex : SV_POSITION`).
- A Normal output (`normal_world : TEXCOORD1`).
- A Tangent output (`tangent_world : TEXCOORD2`).
- And a Binormals output (`binormal_world : TEXCOORD3`).

TRANSFER_SHADOW_CASTER_NORMALOFFSET(O_{RG}) is responsible for transforming the vertices position coordinates to Clip-Space and also calculates the Normal Offset, so you can include shadows in the Normal Maps.

Finally, SHADOW_CASTER_FRAGMENT(I_{RG}) is in charge of color output for shadow projection.

For Unity to compile these macros, you must make sure that you include the UnityCG.cginc directive as well as the `#pragma multi_compile_shadowcaster` to calculate its variants.

```
Shadow Caster
...
// shadow caster Pass
Pass
{
    Name "Shadow Caster"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ShadowCaster"
    }
}

Continued on the next page.
```

```

CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#pragma multi_compile_shadowcaster
#include "UnityCg.cginc"

struct v2f
{
    V2F_SHADOW_CASTER;
};

v2f vert (appdata_full v)
{
    v2f o;
    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    SHADOW_CASTER_FRAGMENT(i)
}
ENDCG
}

```

Now the Shadow Caster pass is ready, so continue looking at the included pass to generate a Shadow Map.

8.0.3. Shadow Map texture.

Continuing with the shader **USB_shadow_map**; in this section, you will define a texture to project shadows onto your object. To do this you have to include **LightMode** in the color pass and make it equal to **ForwardBase**, this way Unity will know that this pass will be affected by lighting.

Shadow Map texture

```
// default color Pass
Pass
{
    Name "Shadow Map Texture"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ForwardBase"
    }
    ...
}
```

Since these types of shadows correspond to a texture that is projected onto UV coordinates, you have to declare a sampler2D variable. Likewise, you must include coordinates to sample the texture. This process will be carried out in the Fragment Shader Stage because the projection must be calculated per-pixel.

Shadow Map texture

```
// default color Pass
Pass
{
    Name "Shadow Map Texture"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ForwardBase"
    }

    CGPROGRAM
    ...
    struct v2f
    {
        float2 uv : TEXCOORD0;
        UNITY_FOG_COORDS(1)
```

Continued on the next page.

```

float4 vertex : SV_POSITION;
// declare the UV coordinates for the shadow map
float4 shadowCoord : TEXCOORD1;
};

sampler2D _MainTex;
float4 _MainTex_ST;
// declare a sampler for the shadow map
sampler2D _ShadowMapTexture;
...
ENDCG
}

```

It is worth mentioning that the texture **_ShadowMapTexture** will only exist within the program, therefore, it should not be declared as Property in shader properties, nor will you pass any texture dynamically from the Inspector, instead, you will generate a projection which will work as a texture.

So, how do you generate a texture projection in the shader? To do this, you must understand how the **UNITY_MATRIX_P** projection matrix works. This matrix allows you to go from View-Space to Clip-Space, which clips objects on the screen.

Given its nature, it has a fourth coordinate already mentioned above, this refers to W, which; in this case, defines the homogeneous coordinates that allow such projection.

Orthographic projection space

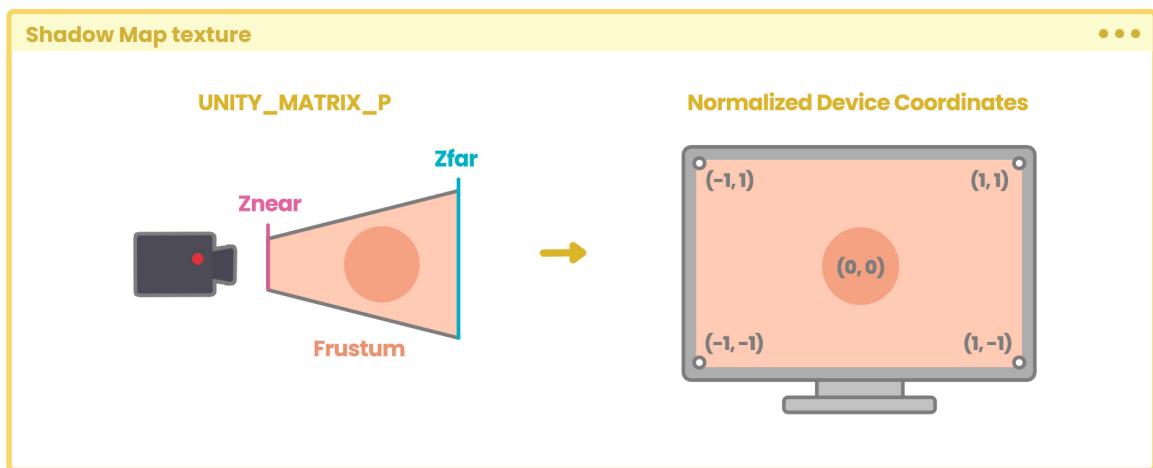
$$W = -((z_{\text{FAR}} + z_{\text{NEAR}}) / (z_{\text{FAR}} - z_{\text{NEAR}}))$$

Perspective projection space

$$W = - (2(z_{\text{FAR}} z_{\text{NEAR}}) / (z_{\text{FAR}} - z_{\text{NEAR}}))$$

As mentioned in section 1.1.6, `UNITY_MATRIX_P` defines the object vertex position in relation to the camera frustum. The result of this operation, which is carried out in the `UnityObjectToClipPos(VRG)` function, generates space coordinates called **Normalized Device Coordinates** (NDC). These types of coordinates have a range between -1.0f and 1.0f, and are generated by dividing the XYZ axes by the W component as follows:

$$\frac{\text{projection.X}}{\text{projection.W}} \quad \frac{\text{projection.Y}}{\text{projection.W}} \quad \frac{\text{projection.Z}}{\text{projection.W}}$$



(Fig. 8.0.3a)

It is essential to understand this process because you must use the `tex2D` function to position the shadow texture in the shader. This function, as you already know, asks us for two arguments: The first refers to its texture; to the texture `_ShadowMapTexture` that you declared before, and the second refers to the texture's UV coordinates. However, in the case of a projection, you must transform normalized device coordinates to UV coordinates, how do you do this? Remember that UV coordinates have a range between 0.0f and 1.0f and NDC; as just mentioned, between -1.0f and 1.0f. So, to transform from NDC to UV, you have to do the following:

$$\begin{aligned} \text{NDC} &= [-1, 1] + 1 \\ \text{NDC} &= [0, 2] / 2 \\ \text{NDC} &= [0, 1] \end{aligned}$$

This operation is summarized in the following equation.

$$\frac{(NDC + 1)}{2}$$

Now, what is the NDC value? As mentioned earlier, its coordinates are generated by dividing the XYZ axes by their component W, as seen in the following equation.

$$\begin{aligned} NDC.x &= \frac{\text{projection.X}}{\text{projection.W}} \\ NDC.y &= \frac{\text{projection.Y}}{\text{projection.W}} \end{aligned}$$

The reason this concept is mentioned is because, in Cg or HLSL, the UV coordinate values are accessed from the XY components or coordinates, what does this mean? Suppose, in the **float2 uv : TEXCOORD0** input, you can access the U coordinate from the **uv.x** component, likewise for the V coordinate which would be **uv.y**. However, in this case, you have to transform the coordinates from **Normalized Device Coordinates** to UV coordinates. Therefore, going back to the previous operation, the UV coordinates would obtain the following value:

$$U = (\frac{NDC.x}{NDX.w} + 1) * 0.5$$

$$V = (\frac{NDC.y}{NDX.w} + 1) * 0.5$$

8.0.4. Shadow implementation.

Now that you understand the coordinate transformation process, you can go back to the **shader USB_shadow_map** to generate a function that you will call **NDCToUV**. This will be responsible for transforming from Normalized Device Coordinates to UV coordinates.

Please note that the previously mentioned function refers to a simplified explanation of the **ComputeScreenPos(P_{RG})** function which has been included in the **UnityCG.cginc** dependency.

This function will be used in the Vertex Shader Stage, so it will have to be declared above that stage.

Shadow implementation

```
// declare NDCToUV above the Vertex Shader Stage
float4 NDCToUV(float4 clipPos)
{
    float4 o = clipPos * 0.5;
    o.xy = float2(o.x, o.y) + o.w;
    o.zw = clipPos.zw;
    return o;
}

// vertex shader stage
v2f vert(appdata v) { ... }
```

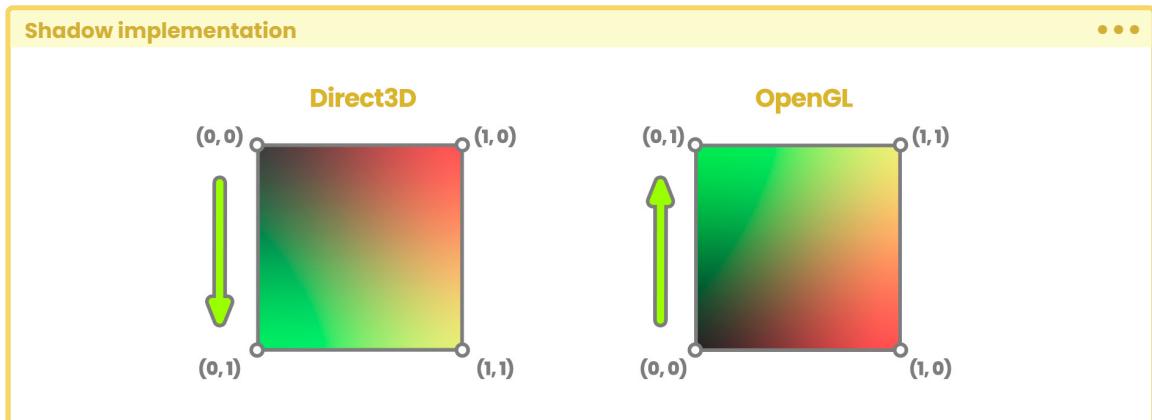
The previous example has declared the **NDCToUV** function to be type **float4**, that is, a four-dimensional vector.

A new four-dimensional vector called **clipPos** has been used as an argument referring to the vertices position Output; the result of the **UnityObjectToClipPos(V_{RG})** function.

The function then translates the mathematical operation mentioned in the previous section into code. Despite this, the operation is not complete because there are some factors that are not being considered in the implementation of the function. These factors refer to:

- The platform where you compile the code.
- And the half-texel offset.

It is necessary to mention that there is a small coordinate difference between OpenGL and Direct3D. In the latter, the UV coordinates start at the top left, while in OpenGL they start at the bottom left.



(Fig. 8.0.4a)

This factor generates a problem when implementing a function in the shader because the result could vary depending on the platform. To solve this issue, Unity provides an internal variable called **_ProjectionParams** which helps correct the coordinate difference.

`_ProjectionParams` is a four-dimensional vector that has different values, e.g., `_ProjectionParams.x` can be “one or minus one” depending on whether the platform has a flipped transformation matrix; i.e., Direct3D. `_ProjectionParams.y` possesses the Z_{NEAR} camera values while `_ProjectionParams.z` possesses the Z_{FAR} values and `_ProjectionParams.w` possesses the $1/Z_{\text{FAR}}$ operation.

In this case, you must use `_ProjectionParams.x` because you only need to turn the V coordinate depending on its starting point.

If you look at Figure 8.0.4a, you will notice that the U coordinate maintains its position regardless of the platform on which you are compiling, however the V coordinate is the one that changes.

Using the concept above, `_ProjectionParams.x` will equal `1.0f` when the shader is compiled in OpenGL, or `-1.0f`, if compiled in Direct3D. Taking this factor into consideration, the operation would be as follows:

Shadow implementation

```
float4 NDCToUV(float4 clipPos)
{
    float4 o = clipPos * 0.5;
    o.xy = float2(o.x, o.y * _ProjectionParams.x) + o.w;
    o.zw = clipPos.zw;
    return o;
}
```

As you can see, the V coordinate of the UV has been multiplied by `_ProjectionParams.x`, in this way you can flip the matrix in case the shader is compiled in Direct3D, now you simply need to add the Half-Texel Offset.

In Unity, there is a macro called `UNITY_HALF_TEXEL_OFFSET` which works on platforms that need mapping displacement adjustments, from textures to pixels. To generate the displacement, use the internal variable `_ScreenParams`, which, like `_ProjectionParams`, is a four-dimensional vector that has a different value for each coordinate.

The values that you are going to use are `_ScreenParams.zw` since,

- Z equals `1.0f + 1.0f / width`.
- While W equals `1.0f + 1.0f / height`.

Taking into consideration the Half-Texel Offset, the function will be as follows:

Shadow implementation

```
float4 NDCToUV(float4 clipPos)
{
    float4 o = clipPos * 0.5;
    #if defined(UNITY_HALF_TEXEL_OFFSET )
        o.xy = float2(o.x, o.y * _ProjectionParams.x) + o.w * _ScreenParams.zw;
```

Continued on the next page.

```

#else
    o.xy = float2(o.x, o.y * _ProjectionParams.x) + o.w;
#endif
    o.zw = clipPos.zw;
    return o;
}

```

Now the shadow UV coordinates are working perfectly, so next, you must declare them in the Vertex Shader Stage, for this, you will make the **shadowCoord** Output the same as the **NDCToUV** function and pass the vertices Output as an argument.

Shadow implementation

```

float4 NDCToUV() { ... }

v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    o.shadowCoord = NDCToUV(o.vertex);

    return o;
}

```

At this point, **shadowCoord** has the projection coordinates, and now you can use them as UV coordinates for the **_ShadowMapTexture**.

As you already know, the “tex2D” function asks for two arguments: the texture and its UV coordinates. You can use this function to generate the shadow in the Fragment Shader Stage.

Shadow implementation

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // create the UV coordinates for the shadow
    float2 uv_shadow = i.shadowMCoord.xy / i.shadowCoord.w;
    // save the shadow texture in the shadow variable
    fixed shadow = tex2D(_ShadowMapTexture, i.shadowCoord).a;

    col.rgb *= shadow;
    return col;
}
```

In the previous example, you created a one-dimensional variable called **shadow**, which saves the sampling values for the **_ShadowMapTexture**.

You can notice that in difference to the vector “col,” the variable “shadow” has only one dimension, why?

You must remember that a shadow texture only has the colors black and white, therefore, you only store the A channel within the shadow variable since it gives a range from 0.0f to 1.0f.

8.0.5. Built-in RP Shadow Map optimization.

The process of implementing shadows that you saw earlier for color passing, can be optimized using the following macros included in Unity:

- SHADOW_COORDS(N_{RG}).
- TRANSFER_SHADOW(O_{RG}).
- SHADOW_ATTENUATION(O_{RG}).

It is fundamental to include the file **AutoLight.cginc** in the program, in addition to **#pragma multi_compile_fwbbase** so its macros work correctly. The latter is responsible for compiling all the lightmap and shadow variants produced by directional lights for the ForwardBase pass.

Built-in RP Shadow Map optimization

• • •

```
// default color Pass
Pass
{
    Name "Shadow Map Texture"

    Tags { "LightMode"="ForwardBase" }

    CGPROGRAM
    ...
#pragma multi_compile_fowndbase nolightmap nodirlightmap nodynlightmap
novertexlight

#include "UnityCG.cginc"
#include "AutoLight.cginc"
...
ENDCG
}
```

The variables defined after the `#pragma`, correspond to optional parameters that you can define to add or remove functionality in the shadow behavior.

- `nolightmap`.
- `nodirlightmap`.
- `nodynlightmap`.
- `novertexlight`.

Note that, if you use the macros for the implementation of the shadow map, you must use some predefined definitions in the semantics that you use as input/output, otherwise your code will generate an error.

Built-in RP Shadow Map optimization

• • •

```
struct appdata
{
    float4 vertex : POSITION;
    // float2 uv : TEXCOORD0;
    float2 texcoord : TEXCOORD0;
};
```

By default, the code includes the vector **uv**, with its semantic **TEXCOORD[n]** for the Vertex Input. If you use the macros, you must replace the definition **uv** with **texcoord** otherwise the internal operation will not be able to read the UV coordinates to generate the Shadow Map.

The same thing happens in the case of Vertex Output. The **vertex** vector must be renamed **pos** so that the internal process can write the UV coordinates. Furthermore, you have to include the macro **SHADOW_COORDS(N_{RG})**, which includes the UV coordinates that you pass to the Fragment Shader Stage.

Built-in RP Shadow Map optimization

• • •

```
struct v2f
{
    float2 uv : TEXCOORD0;
    // store the shadow data in TEXCOORD1
    SHADOW_COORDS(1) // without ();
    // float4 vertex : SV_POSITION;
    float4 pos : SV_POSITION;
    ...
};
```

After having declared the Inputs and Outputs in the “Shadow Map Texture” pass, you can synchronize the values in the Vertex Shader Stage.

Built-in RP Shadow Map optimization

```
v2f vert (appdata v)
{
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    o.uv = v.texcoord;
    // transfer the shader UV coordinates to the fragment shader
    TRANSFER_SHADOW(o) // without ();
    return o;
}
```

The TRANSFER_SHADOW(O_{RG}) macro is the same as the **NDCToUV** operation performed in the previous section. Basically, it calculates the UV coordinates for the shadow texture. Now you can transfer the coordinates to the Fragment Shader Stage.

Built-in RP Shadow Map optimization

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // use the shadows
    fixed shadow = SHADOW_ATTENUATION(i);
    col.rgb *= shadow;

    return col;
}
```

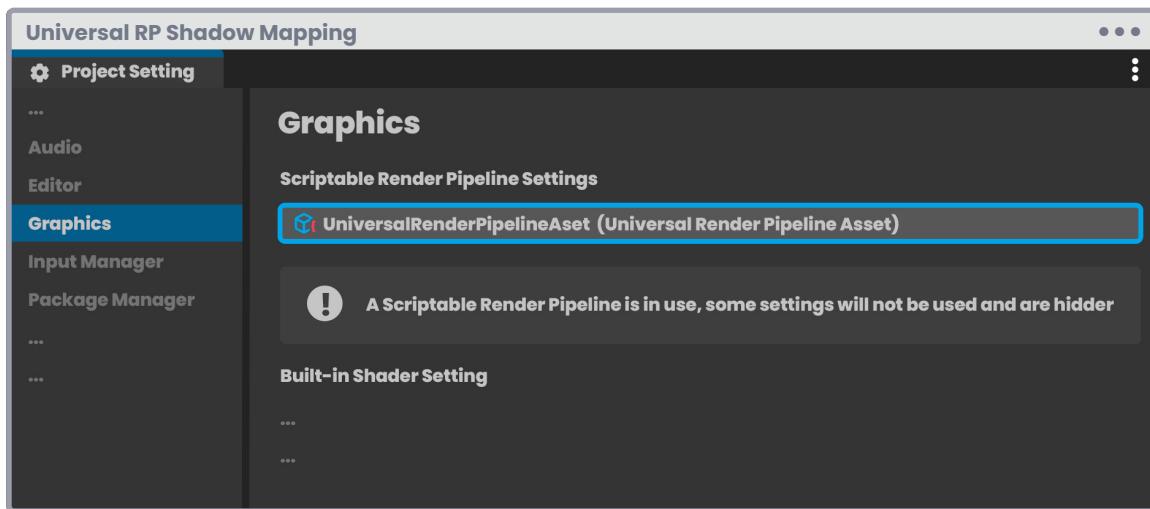
Finally, SHADOW_ATTENUATION(O_{RG}) contains the texture and its projection. This function can be saved as a one-dimensional vector, since it only occupies one channel in the texture projection.

8.0.6. Universal RP Shadow Mapping.

The process previously carried out is the optimal for Unity, however, its implementation may vary depending on the Render Pipeline chosen, e.g., the process for the shadow map implementation that was detailed in the previous sections, works only in Built-in RP.

If you want to generate shadows in Universal RP, you have to make use of the dependency **Lighting.hlsL**, as this package includes definitions for the coordinate calculations.

Start by configuring the Render Pipeline in Universal RP. To do this, you must make sure that you have assigned a **Render Pipeline Asset** in the **Scriptable Render Pipeline Settings** box, in the **Graphics** menu.



(Fig. 8.0.6a)

Once configured, create a new Unlit shader called **USB_shadow_map_UPR**. This shader will be used to implement both a Shadow Map and the Shadow Caster function in the rendering engine. Furthermore, it is necessary to review the dependencies needed so that the program can compile.

Given its nature, this shader is opaque and works in Universal RP, therefore, the program rendering values need to be defined.

Universal RP Shadow Mapping

• • •

```

Shader "USB/USB_shadow_map_URP"
{
    Properties { ... }
    SubShader
    {
        Tags
        {
            "RenderType"="Opaque"
            // add the rendering pipeline
            "RenderPipeline"="UniversalRenderPipeline"
        }
        ...
    }
}

```

As you already know, shadows are produced in two passes: one for shadow projection and one for texture. By default, Unity adds only one pass which can be used for the shadow map.

In Universal RP, there is a shader called **Lit**, which has been included in the **Universal Render Pipeline** category. This shader has a pass called **ShadowCaster** that is responsible for the calculation of coordinates for shadow projection.

In Unity, you can dynamically include passes using the **UsePass** command, what does this mean? You could either go to the Lit shader, copy the pass, and paste it into your shader, or you can simply include the route of the pass and make a call directly to its function.

Universal RP Shadow Mapping

```

SubShader
{
    Tags
    {
        "RenderType"="Opaque"
        // add the rendering pipeline
        "RenderPipeline"="UniversalRenderPipeline"
    }
    // default color Pass
    Pass { ... }
    // shadow caster Pass
    UsePass "Universal Render Pipeline/Lit/ShadowCaster"
}

```

The UsePass command is used when you want to include functionalities of a pass that is in a different shader to the one you are programming. Taking into consideration the previous example: the **ShadowCaster** pass located in the **Lit** shader, in the **Universal Render Pipeline** path, will contain the Shadow Caster calculations for your program.

Universal RP Shadow Mapping

```

// default shader included in Unity
// different from the one we are programming
Shader "Universal Render Pipeline/Lit"
{
    Properties { ... }
    SubShader
    {
        Pass
        {
            Name "ShadowCaster"
            Tags { "LightMode"="ShadowCaster" }
            ...
        }
    }
}

```

Since you have included this path, you must define the default pass as the one that will process the shadow map, that is, you have to include the **LightMode** for Universal RP.

Universal RP Shadow Mapping

```
// default color Pass
Pass
{
    Tags
    {
        "LightMode"="UniversalForward"
    }
    HLSLPROGRAM
    ...
    ENDHLSL
}
```

The **UniversalForward** property works similarly to **ForwardBase**, the difference is that the former evaluates all light contributions in the same pass.

Once the LightMode is defined, you must add some dependencies that will help you to implement the texture.

Universal RP Shadow Mapping

```
HLSLPROGRAM
#pragma vertex vert
#pragma fragment frag
#pragma multi_compile _ _MAIN_LIGHT_SHADOW

#include "HLSLSupport.cginc"

#include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Core.hlsl"
#include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Lighting.hlsl"
...
ENDHLSL
```

The dependencies **Core.hsls** and **Lighting.hsls** have several functions that you will use in the calculation, among them: the **GetVertexPositionInputs** function, which belongs to a sub-dependence called **ShaderVariablesFunctions.hsls** and **GetShadowCoord** that is included in a sub-dependence **Shadows.hsls**.

Because you are going to implement a texture, you will need a vector that can store its UV coordinates, for this create a four-dimensional vector in the Vertex Output.

Universal RP Shadow Mapping

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float4 shadowCoord : TEXCOORD1;
};
```

As detailed in section 8.0.4, the **shadowCoord** vector will store the result of the vertices' transformation from NDC to UV coordinates.

In Universal RP, use the **GetShadowCoord(V_{RG})** function that asks you for a **VertexPositionInputs** type object as an argument.

Universal RP Shadow Mapping

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = TransformObjectToHClip(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);

    // find VertexPositionInputs at Core.hsls
    // find GetVertexPositionInputs at
    // ShaderVariablesFunctions.hsls
    VertexPositionInputs vertexInput = GetVertexPositionInputs(v.vertex.xyz);
```

Continued on the next page.

```
// find GetShadowCoord at Shadows.hlsl
o.shadowCoord = GetShadowCoord(vertexInput);
return o;
}
```

At this point, the shadowCoord vector already has the coordinates for the shadow generation, now you simply have to pass it as an argument to the **GetMainLight(s_{RG})** function that contains the calculations for light direction, attenuation, and light color. This function has been included in the **Lighting.hlsl** dependency.



```
Universal RP Shadow Mapping
...
Light GetMainLight()
{
    Light light;
    light.direction = _MainLightPosition.xyz;
    light.distanceAttenuation = unity_LightData.z;
    light.shadowAttenuation = 1.0;
    light.color = _MainLightColor.rgb;
    return light;
}

Light GetMainLight(float4 shadowCoord)
{
    Light light = GetMainLight();
    light.shadowAttenuation = MainLightRealtimeShadow(shadowCoord);

    return light;
}
```

The **GetMainLight(s_{RG})** function has up to three variations, in the third you can include the shadowMask, in case you need it. Now you simply have to create a vector to store the shadow attenuation and multiply it by the texture RGB color.

Universal RP Shadow Mapping

```
fixed4 frag (v2f i) : SV_Target
{
    // find GetMainLight function at Lighting.hlsl
    Light light = GetMainLight(i.shadowCoord);
    float3 shadow = light.shadowAttenuation;

    fixed4 col = tex2D(_MainTex, i.uv);
    col.rgb *= shadow;

    return col;
}
```

Shader Graph.

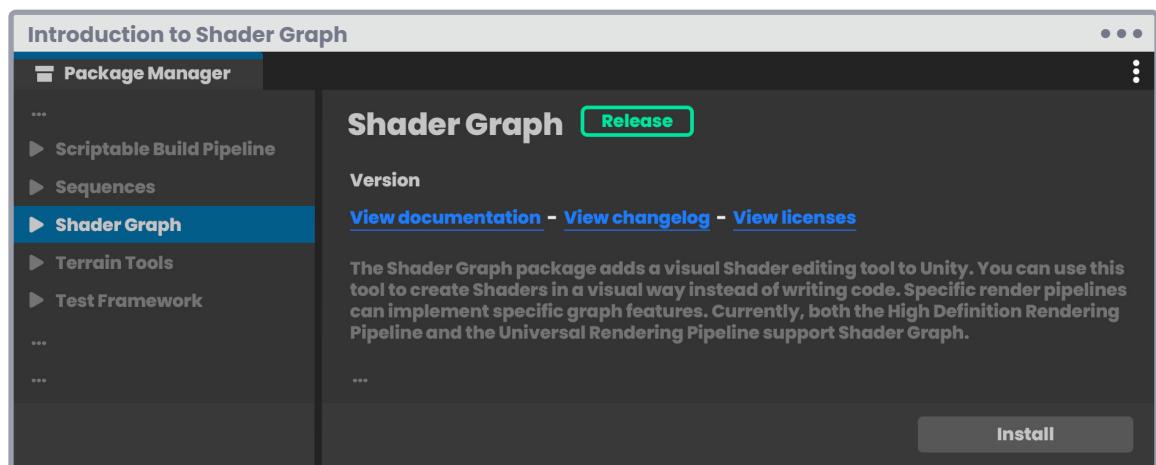
9.0.1. Introduction to Shader Graph.

Up to this point, much of the Render Pipeline structure has been reviewed, as well as understanding of how a Unity shader works. Now you will look at **Shader Graph**, since its structure and analogy are mainly based on all the previous knowledge acquired.

Shader Graph is a **package** that adds support for a visual node editing tool. Its interface is used by artists and developers to create custom shaders through **nodes** instead of having to write code. Even so, its **Custom Function** node has a high compatibility with HLSL, which allows you to generate specific functions within the program.

Currently, Shader Graph is available for two rendering modalities, these are:

- ▶ High Definition RP.
- ▶ And Universal RP.



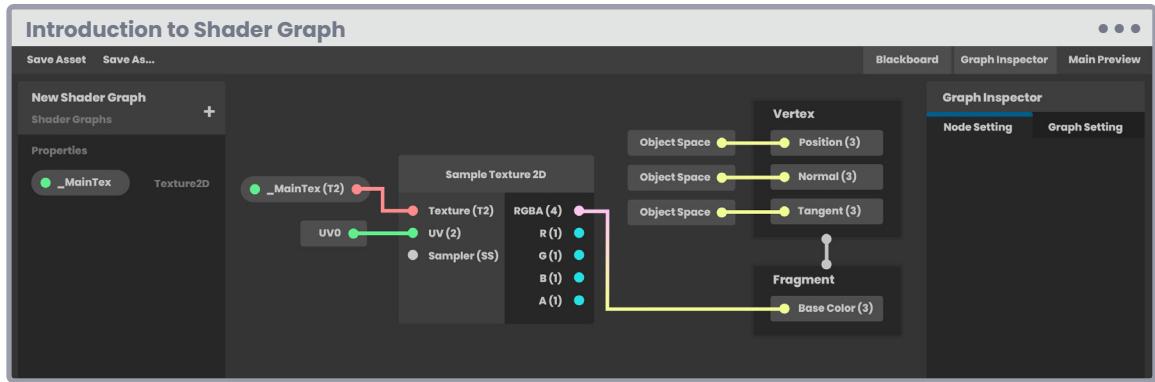
(Fig. 9.0.1a)

There are a few things to consider when working with Shader Graph, the versions developed for Unity 2018 are BETA versions and do not receive support, whereas the later versions are actively compatible and do receive support.

Another thing is that it is very likely that shaders created with this interface do not compile correctly in other versions. This is because new features are added in each update, sometimes affecting the node set, even more so if you are using custom functions.

So, is Shader Graph a good tool for developing shaders? The answer is yes, even more so for artists.

For those who have worked with 3D software such as Maya or Blender; Shader Graph will be very useful since it uses a system of nodes very similar to **Hypershader** and **Shader Editor** which allows for more intuitive shader creation.



(Fig. 9.0.1b)

Before introducing this topic, be aware that the Shader Graph interface has functional variations according to its version, e.g., at the time of writing this book, its most up-to-date version is **12.0.0**.

If you create a node with this version, you can see that the Vertex and Fragment Shader Stage appear separate and work independently. However, if you go to version **8.3.1**, both stages are merged within a node called **Master**, which refers to the final shader output color.

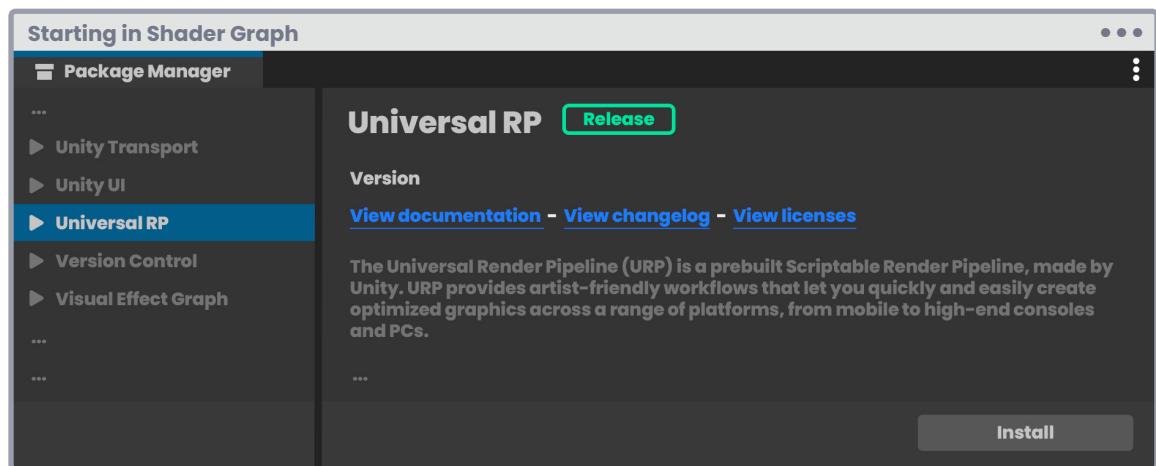
As you mentioned before, it is very possible that the shaders created in this interface do not compile in all its versions, in fact, if you create a shader in version 8.3.1 and update to version 12.0.0, there are probably functional changes that prevent its compilation.

9.0.2. Starting in Shader Graph.

There are two ways to include Shader Graph in your project:

- 1 From the default settings, when you start a project, either in Universal RP or High Definition RP.
- 2 Or similarly, installing from the Unity **Package manager**, which is located in the path: Window / Package Manager.

If you start a project in Universal RP or High Definition RP, this package is included by default, and you can use it immediately. This means that in the menu: Assets / Create / Shaders, you can find a new section of shaders that work exclusively for these types of rendering.



(Fig. 9.0.2a)

What if you have created a project in Built-in RP and want to upgrade it to either Universal RP or High Definition RP?

From Built-in RP, it is easy to upgrade to Universal RP. To do this, you must follow the second step that was described to add Shader Graph to the project. Likewise, it will be necessary to include the Universal RP package, which adds support to this type of Render Pipeline.

It should be noted that you cannot upgrade a project from Built-in RP or Universal RP to High Definition RP. High Definition RP is a Render Pipeline for high-end video games and can only be started as a project from the Unity Hub.

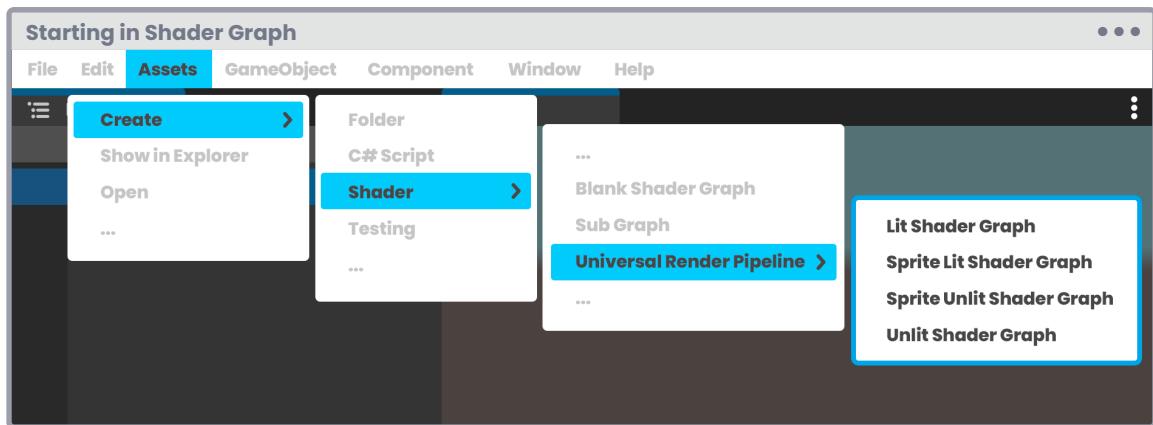
Once you have installed both packages, you will have to create a type of object called a **Pipeline Asset** and another called **Forward Renderer Data** from the menu: Assets / Create / Rendering.

Generally, when you create a Pipeline Asset, Unity creates a Forward Renderer Data file by default, which works as Render Path for Universal RP.

Having these files, you must go to the **Project Settings** window and in the submenu **Graphics**, assign the Pipeline Asset object in the **Scriptable Render Pipeline Asset** box, this way Unity will know that the Render Pipeline now corresponds to the Universal RP configuration.

To finish, go to **Quality**, which is inside the **Project Setting** and assign the Pipeline Asset again in the **Rendering** box, in this way all the properties associated to rendering can be modified from the same Render Pipeline Asset.

The process of creating a shader in Universal RP or High Definition RP is exactly the same as in Built-in RP, the only difference is that Shader Graph shaders are in an exclusive category that is added automatically when installing the package.



(Fig. 9.0.2b)

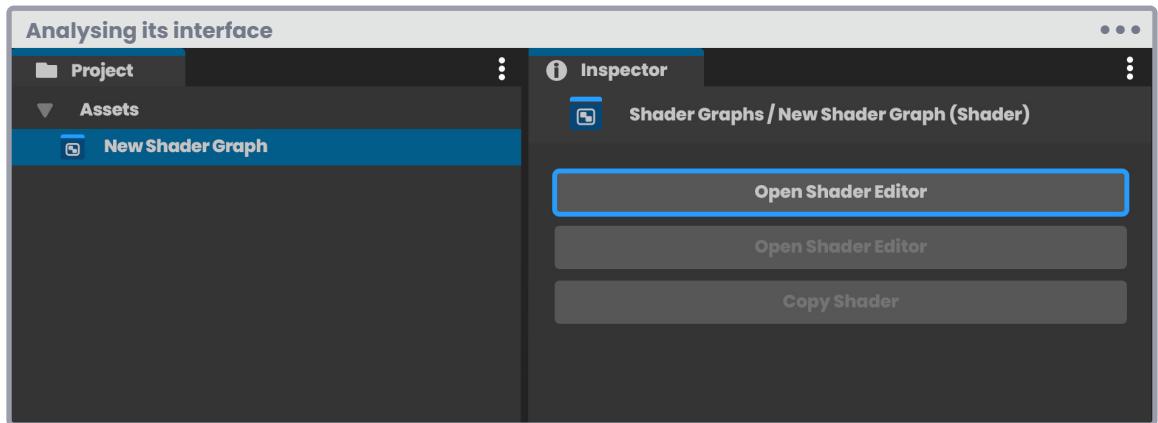
Please consider that the process and steps mentioned above may change depending on the version of Unity; however, their creation and installation analogy are the same.

9.0.3. Analysing its interface.

As you have seen already, the Shader Graph interface has structural changes depending on the version you are using. This section will detail update version 10.4.0 that corresponds to the package included in Unity 2020.3.1f1. Now, regardless of its reference, the analogy between the different versions will be the same, therefore, if you have reached this point in the book, you will be able to clearly understand its interface and different properties.

To start in the Shader Graph interface, you must have previously created a Universal RP or High Definition RP shader, as appropriate.

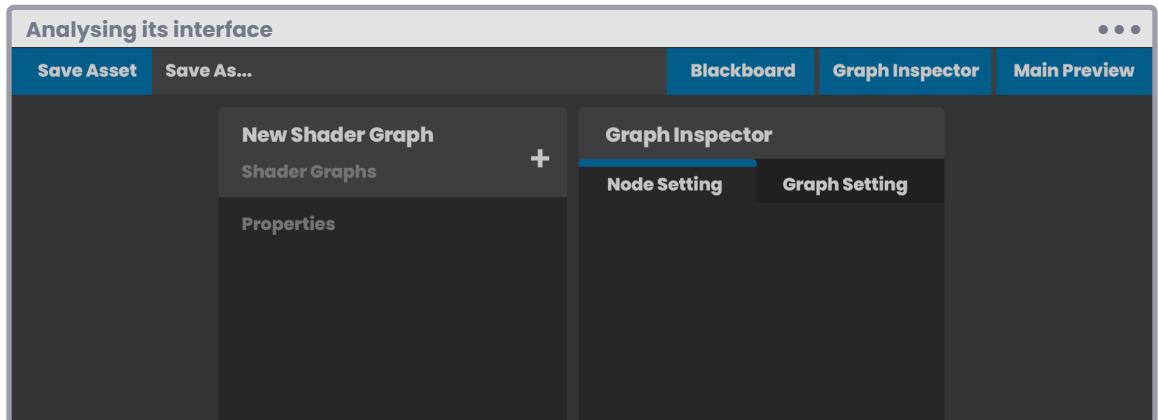
Once you have a shader in your Project, you can open it by double-clicking on it or by pressing the **Open Shader Editor** button found in the Inspector.



(Fig. 9.0.3a)

On its interface you can find four main buttons that define part of its functionalities, these buttons correspond to:

- **Save Asset.**
- **Blackboard.**
- **Graph Inspector.**
- And **Main Preview.**



(Fig. 9.0.3b)

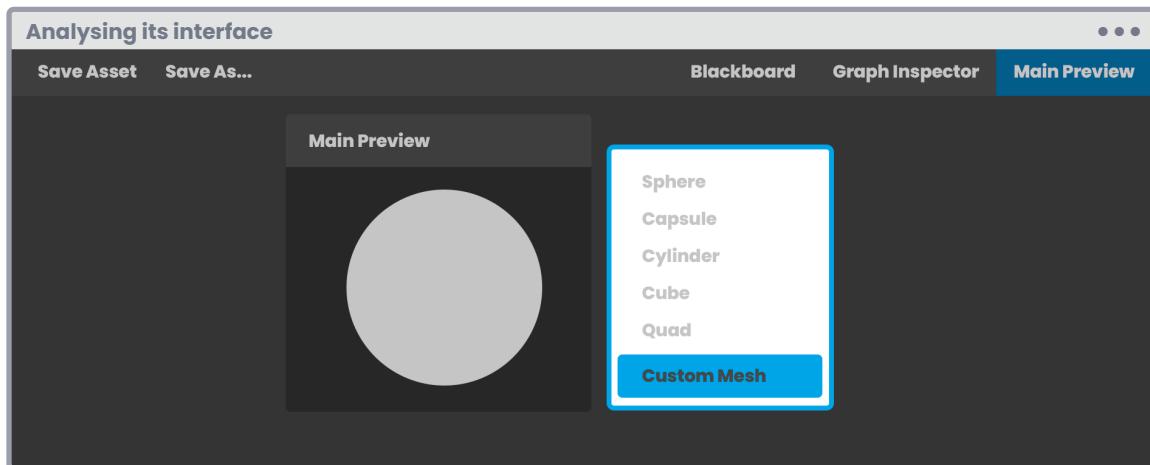
The Shader Graph interface looks like Figure 9.0.3b above. The **Save Asset** button allows you to save the internal configuration of the shader and works in the same way as the Ctrl+S command for Windows or Cmd+S in the case of a Mac.

Blackboard is the direct analogy of ShaderLab Properties. As you already know, if you want to create a property in a shader via code, you must add it within the Properties in declarative language. In Shader Graph it is practically the same except that they are added by pressing the **Plus** button located at the top of the menu.

Graph Inspector is a small panel that lets you modify the node and general configuration of the shader. This panel has two tabs which are: **Node Settings** and **Graph Setting**, each with different functionalities.

Node Settings allows you to name properties, references, default values, mode, precision, and others, while **Graph Settings** defines the general shader configuration, e.g., if your shader will be transparent, additive, or opaque.

Finally, **Main Preview** activates or deactivates the node configuration **preview**, ideal for previewing an effect that is being developed. An interesting feature of the preview pane is that it lets you import custom objects. To do this, simply right-click on the preview area and select **Custom Mesh**.



(Fig. 9.0.3c)

9.0.4. Your first shader in Shader Graph.

Here you will work with Universal RP to test your shaders. Start by going to the Unity Project window and creating an **Unlit Shader Graph** shader, using the following path:

- Create / Shader / Universal Render Pipeline / Unlit Shader Graph.

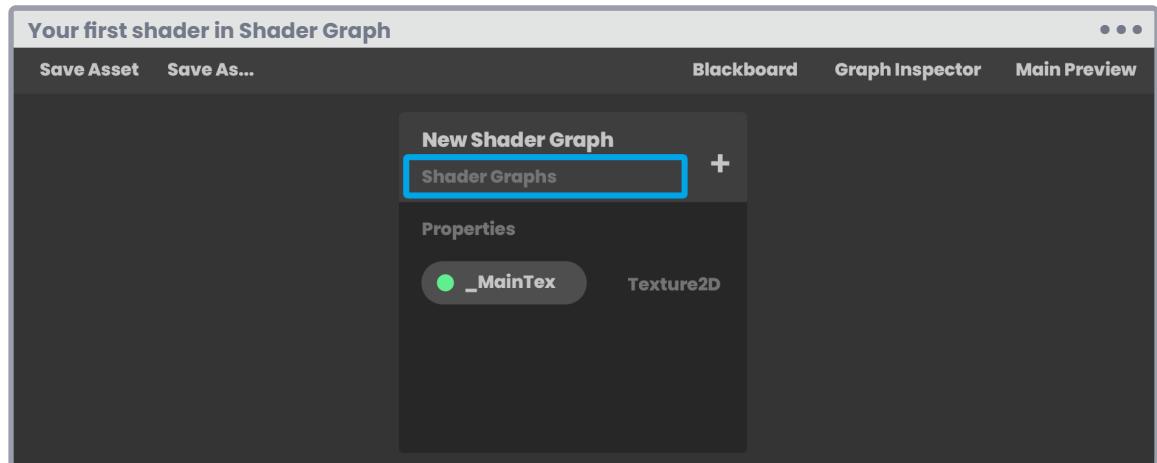
Please note: this route will only be visible if you have the Shader Graph package included in your project.

Next, in the Shader Graph, recreate the **USB_simple_color** shader that you started in section 3.0.1. To do this, name this new shader **USB_simple_color_SG** and double-click to start it.

At first glance, the first difference you see between both programs is that the first one has a **.shader** extension, while the second has **.shadergraph**. Likewise, the presentation icon in each case has a graphic variation that is used to differentiate their nature.

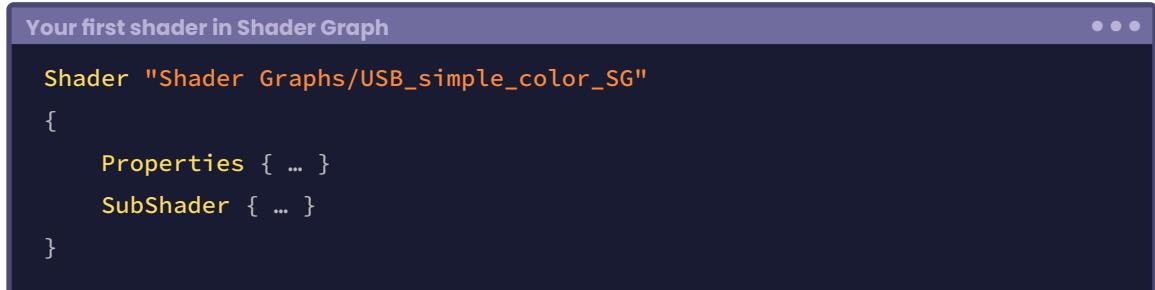
It should be noted that, as in previous programs, in Shader Graph you can also modify the path in the Inspector.

By default, all the shaders that you create with this interface will be saved in the **Shader Graphs** path, however, if you want to modify their destination, you can do it directly from the access to the path which is found in the Blackboard, under the shader name.



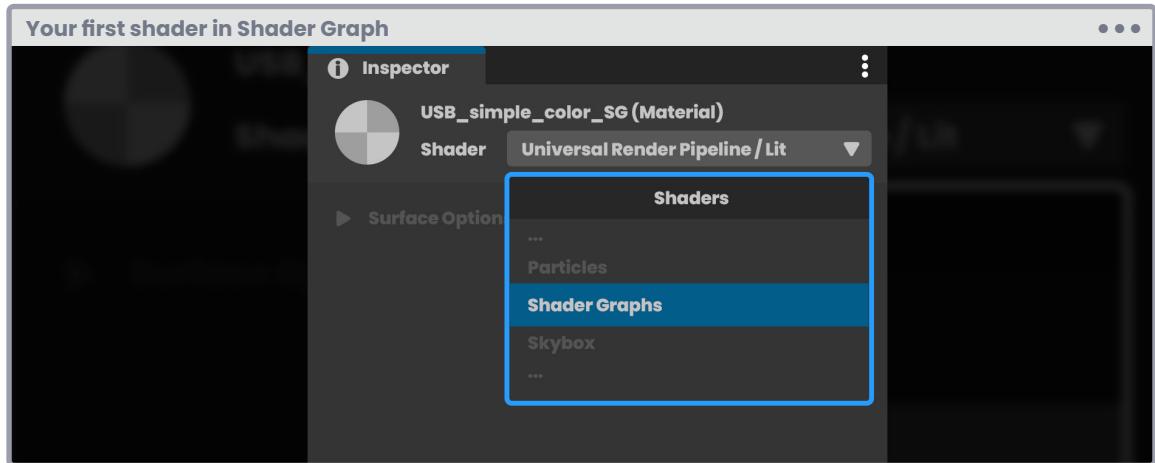
(Fig. 9.0.4a)

The above operation is the equivalent to manually changing the path of a program with the **.shader** extension.



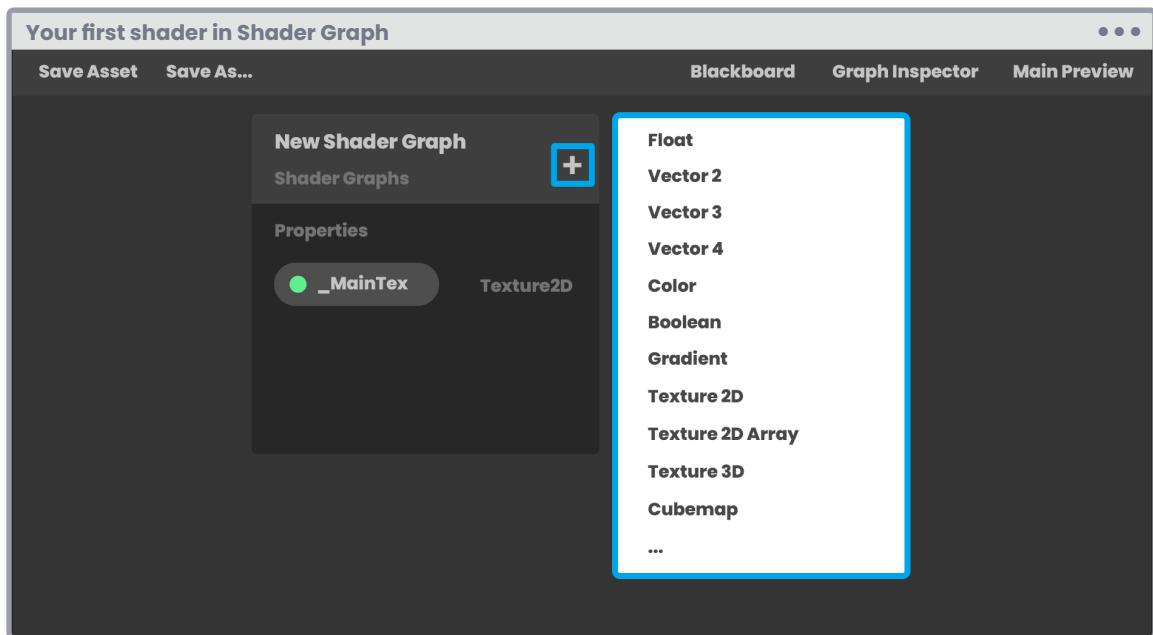
Your first shader in Shader Graph

```
Shader "Shader Graphs/USB_simple_color_SG"
{
    Properties { ... }
    SubShader { ... }
}
```



(Fig. 9.0.4b)

As you could see in previous sections, every time you created an Unlit shader, Unity added a default property corresponding to a texture called **_MainTex**. Shader Graph cannot do this. When you start the interface, the Blackboard, where properties are added, is empty by default. This means that you will have to add each property to the shader independently using the **Plus** button.



(Fig. 9.0.4c)

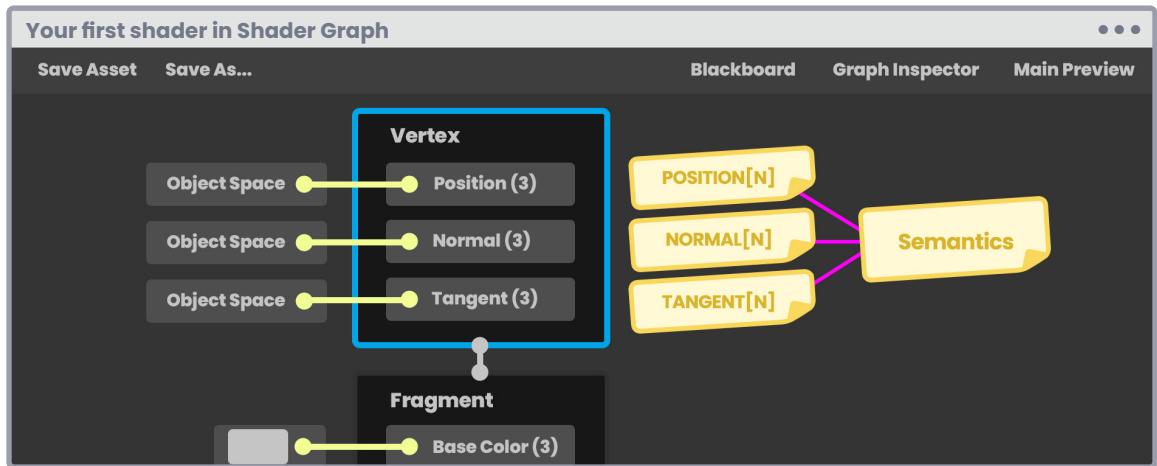
Before starting, here is a small introduction to the Vertex-Fragment Shader Stage to understand how it works with this interface.

As you can see, in the **Vertex Shader Stage** there are three defined inputs which are:

- Position(3).
- Normal(3).
- Tangent(3).

As in a Cg or HLSL shader, these inputs refer to the semantics that you can use in the **Vertex Input**, this means that,

- Position(3) equals POSITION[n].
- Normal(3) equals NORMAL[n].
- And Tangent(3) equals TANGENT[n].



(Fig. 9.0.4d)

The value that precedes such inputs refers to the number of dimensions that it possesses, therefore Position(3) equals **Position.xyz** in object-space.

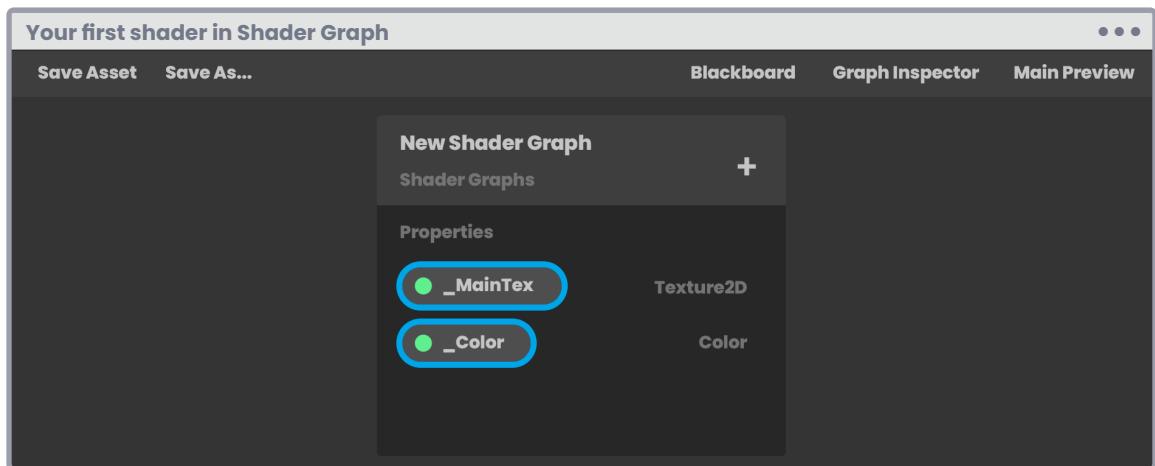
Why has Shader Graph got three dimensions but Cg or HLSL up to four? Recall that the fourth dimension of a vector corresponds to its W component, which, in most cases, is “one or zero.” When W equals **one**, it means that the vector corresponds to a position in space or a point. Whereas, when it equals **zero** the vector corresponds to a direction in space.

Considering the above explanation, you can conclude that, if the input Position had four coordinates, then these would be XYZ1, and for the Normals XYZ0, likewise for the Tangents that also correspond to a direction in space.

It is essential to understand this analogy since most of the functions, applications and properties in Shader Graph are based on the structure of a “.shader” in HLSL.

To start your program, the first thing you will do is go to **the Blackboard** and create two properties:

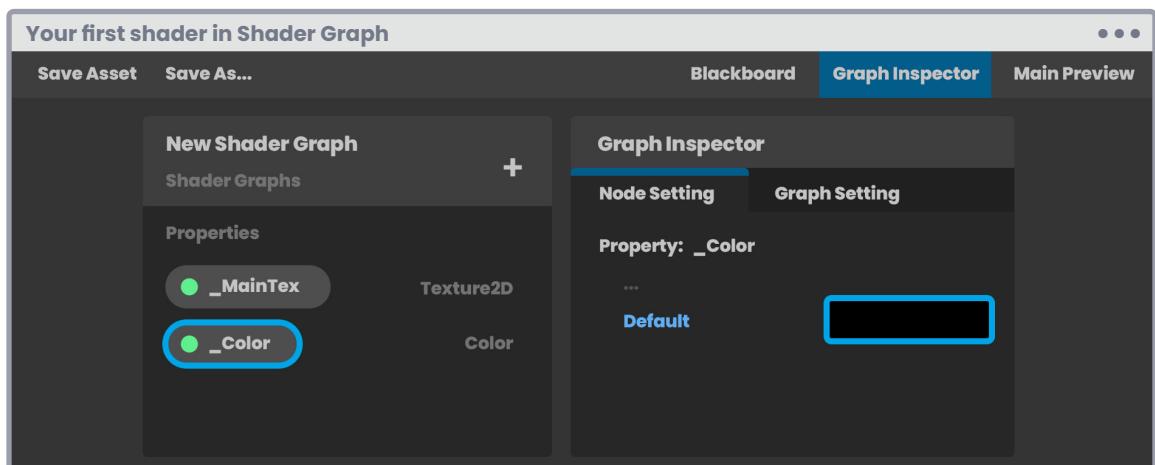
- 1 A color type, which you will call **_Color**.
- 2 And a Texture2D type, which you will call **_MainTex**.



(Fig. 9.0.4e)

A question that frequently arises is, what is the default color of our property `_Color` or even `_MainTex`?

When you declare a property in ShaderLab you can define its tonality by its value. However, in Shader Graph it is different. By default, your property `_Color` is black, this can be corroborated by going to the **Graph Inspector; Node Settings** tab. If you look at the "Default" property, you will see that it is set to black.



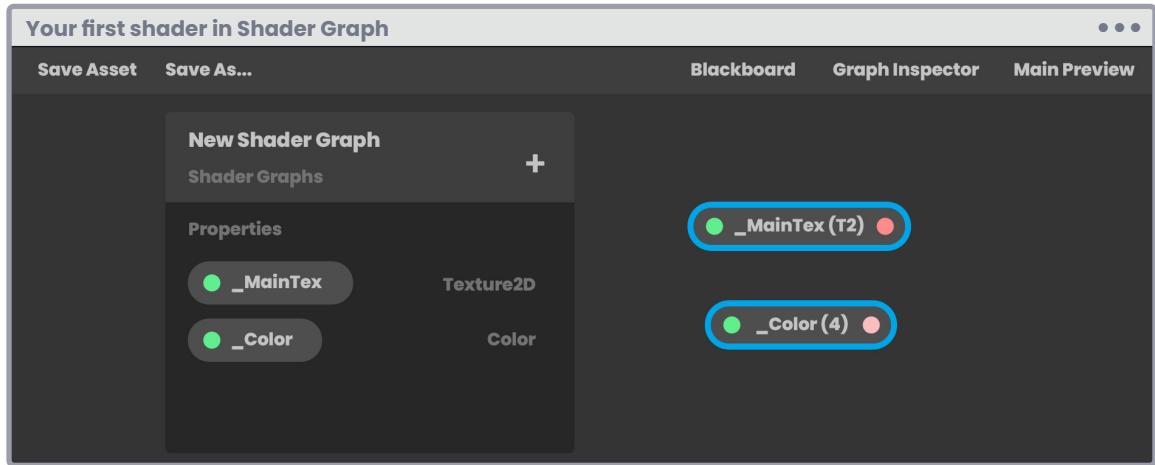
(Fig. 9.0.4f)

You can select a different color by pressing the tonality bar (black bar) to change the color.

For `_MainTex` it is exactly the same, within the Graph Inspector; Node Settings tab, you can select its **Mode**, which by default equals **white**.

To generate communication between the ShaderLab properties and your program, you must create connection variables within the CGPROGRAM field.

This process is different in Shader Graph. What you must do is drag the properties from the Blackboard to the node area.

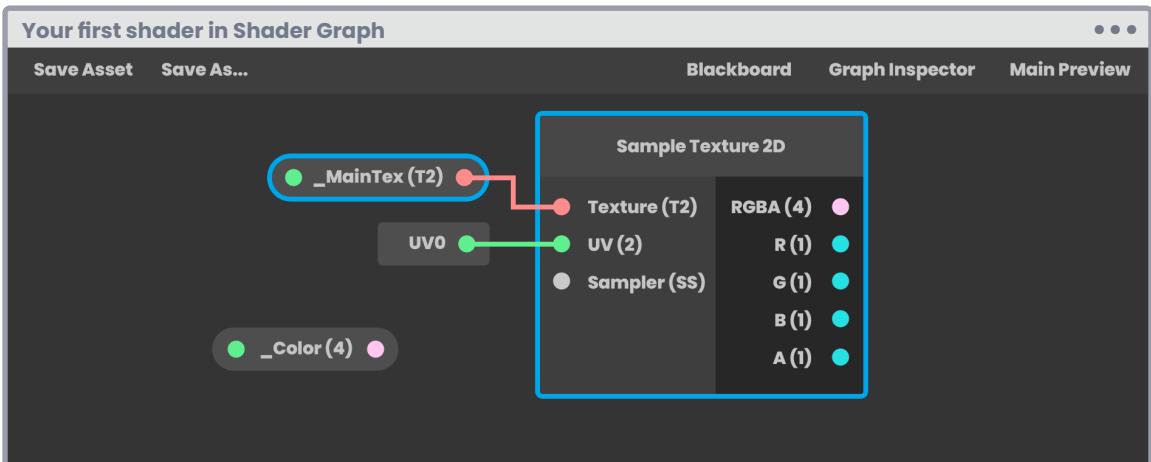


(Fig. 9.0.4g)

As you saw in Chapter I, section 3.2.7, you must generate a **sample** for a texture to be projected onto an object. For this, there is the **Sample Texture 2D** node, which fulfills the function of $\text{tex2D}(\mathbf{s}_{RG}, \mathbf{UV}_{RG})$. If you want to work with this node, you can do two actions:

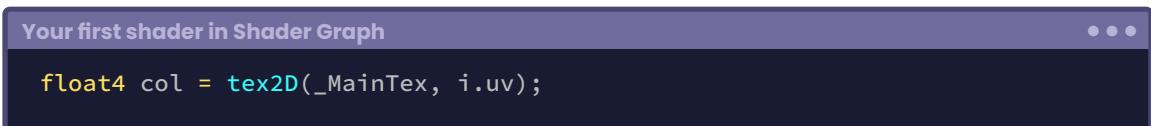
- 1 Press the **spacebar** on the node area and write the name of the node you are looking for; which in this case would be **Sample Texture 2D**,
- 2 Or right-click, select the **Create Node** option and find the node you want to work with.

To make the Texture2D type texture work in conjunction with the Sample Texture 2D node, connect the output of the `_MainTex` property to the **Texture(T2_{RG})** type input, which is included in Sample Texture 2D.

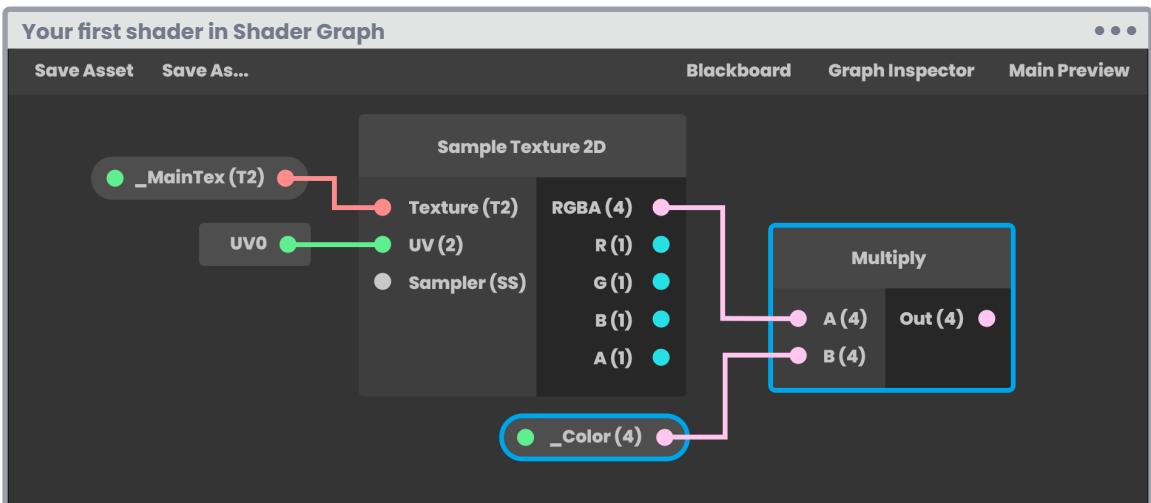


(Fig. 9.0.4h)

This operation is equivalent to creating a four-dimensional vector and passing both the texture and the UV coordinates input to it.

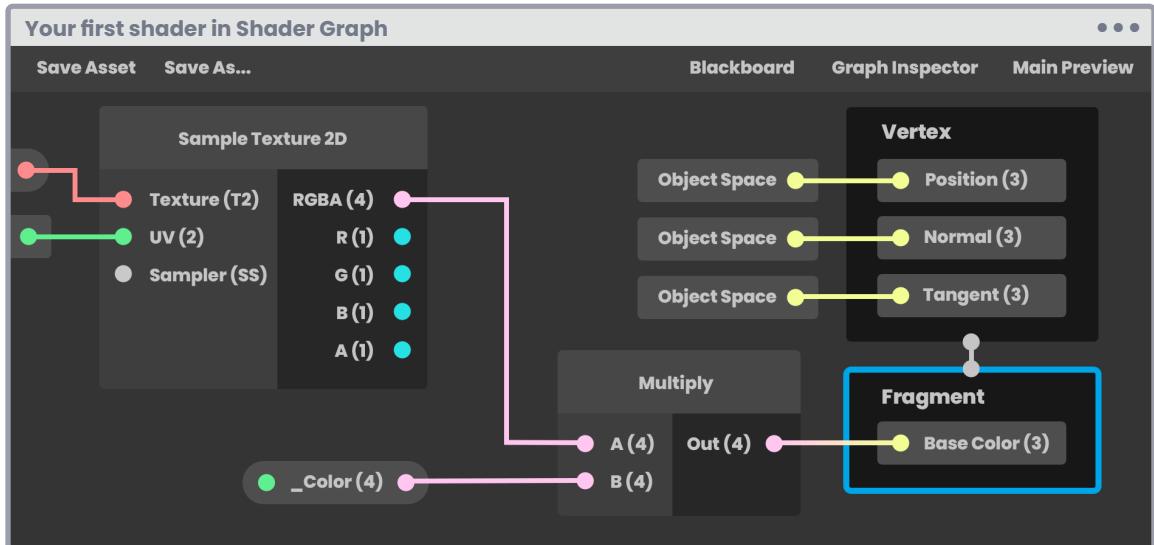


If you want to change the tint of the texture, multiply the 2D Sample Texture RGBA output by the `_Color` property output. In this way, when the color is black, the texture will be black, and when the color is white, the texture will be its default color.



(Fig. 9.0.4i)

To multiply both nodes, you must simply bring the **Multiply** node and pass both values as input. Finally, the color output (factor) of the previous operation must be connected to the **Base Color** found in the Fragment Shader Stage. Once the operation is complete, you must save your shader by pressing the **Save Asset** button that is located at the top left of the Shader Graph interface.



(Fig. 9.0.4j)

Something you must consider is that the **Base Color** is an input that corresponds to the final RGB color of the shader and is part of the operations that are occurring within the Fragment Shader Stage. This means that the final color output that includes alpha and blending, would be the same as the `SV_Target` semantic that you can find in a `.shader`.

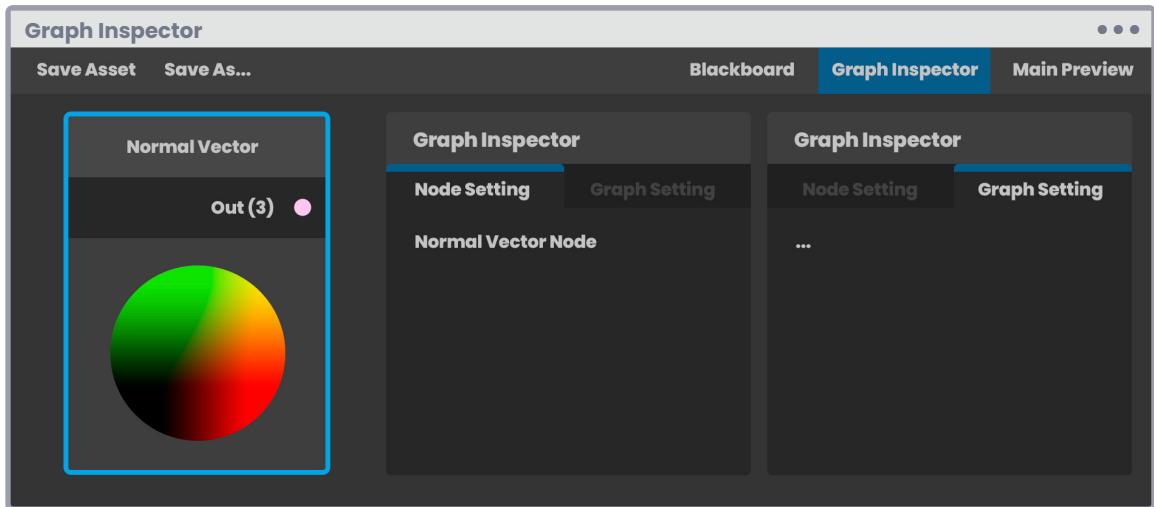
9.0.5. Graph Inspector.

According to official Unity documentation;

The Graph Inspector allows us to interact with any selectable element and settings in Shader Graph. We can use the Graph Inspector to edit attributes and values.

This panel may have some aesthetic variations depending on the Shader Graph version. Its version 10.6.0 looks like in figure 9.0.5a.

To summarize, this panel, divided into two sections called **Node** and **Graph**, has configurable properties that can modify the color output. Among them you can find Blending, Cull, and Alpha Clip. Also, you can adjust the properties of the nodes in your configuration.

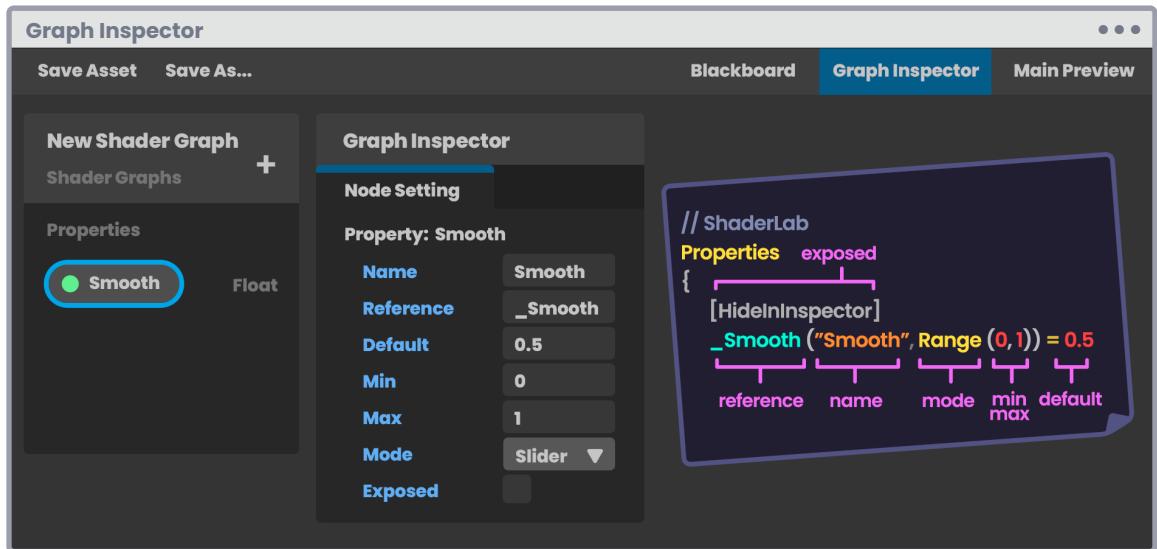


(Fig. 9.0.5a)

On the one hand, **Graph Settings** has the general properties of the node, in other words, the commands that you have used previously to define the behavior of the pixels on the screen, e.g., the **Blend [SourceFactor] [DestinationFactor]** command is enabled once you determine the type of surface in the shader. Likewise with the **Cull** command which you can enable through the **Two Sides** checkbox.

On the other hand, **Node Settings** contains the attributes of the selected elements, e.g., in the **Custom Function** node, appear the precision, and preview options, as well as those inputs and outputs that can be configured according to the operation being carried out. Only the precision and preview options appear for the remaining ones.

This same behavior is reflected in the properties you have added to the Blackboard. As you can see in Figure 9.0.5b, the tab shows the attributes that you added manually when writing the shader.



(Fig. 9.0.5b)

The **precision** attribute refers to the calculation to decimal places of a node on the GPU. By default, it is set to **inherit**. However, you can change it to **single** (float), which is precise to six decimal places, or to half, which is to three decimal places.

9.0.6. Nodes.

The first chapter reviewed the operations performed by some intrinsic functions, among them: $\text{clamp}(A_{RG}, X_{RG}, B_{RG})$, $\text{abs}(N_{RG})$, $\text{ceil}(N_{RG})$, $\text{exp}(N_{RG})$ and many others. The nodes in Shader Graph are the graphical representation of these functions; therefore, they fulfill the same functions, e.g., according to the official Unity documentation, the **Clamp** node is articulated as follows:



The following example code represents a possible outcome of this node.

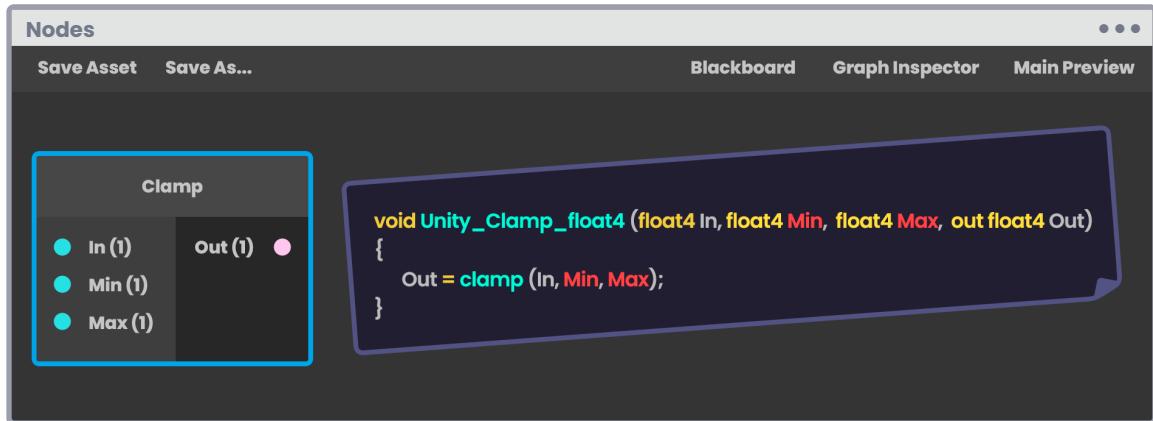


```
Nodes

void Unity_Clamp_float4(float4 In, float4 Min, float4 Max, out float4 Out)
{
    Out = clamp(In, Min, Max);
}
```

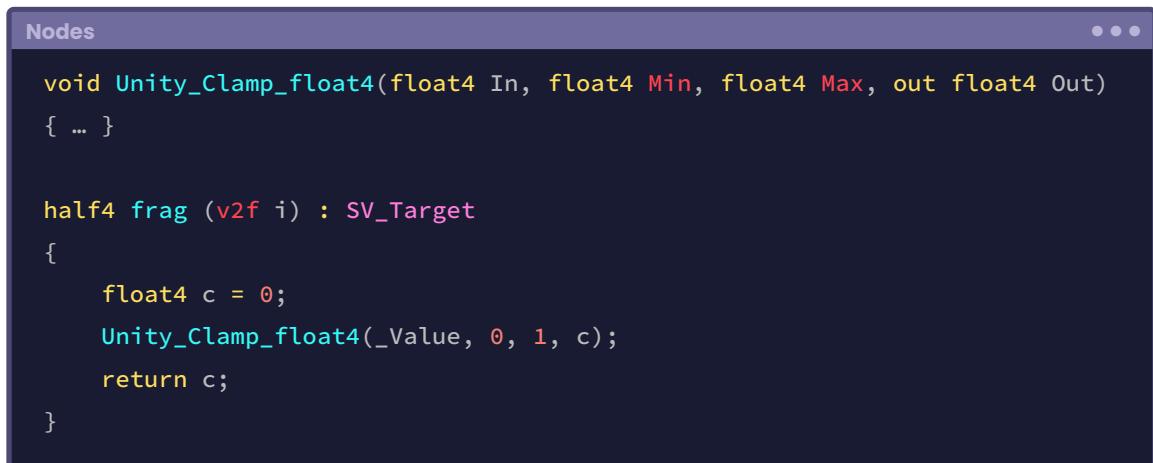
As you can see in this example, the **Unity_Clamp_float4** function of type **void** has three inputs corresponding to four-dimensional vectors. Among them, you can find **In**, **Min** and **Max**. These same values can be seen graphically in the node construction, as shown in Figure 9.0.6a.

The output called **Out** simply has the operation to be carried out.



(Fig. 9.0.6a)

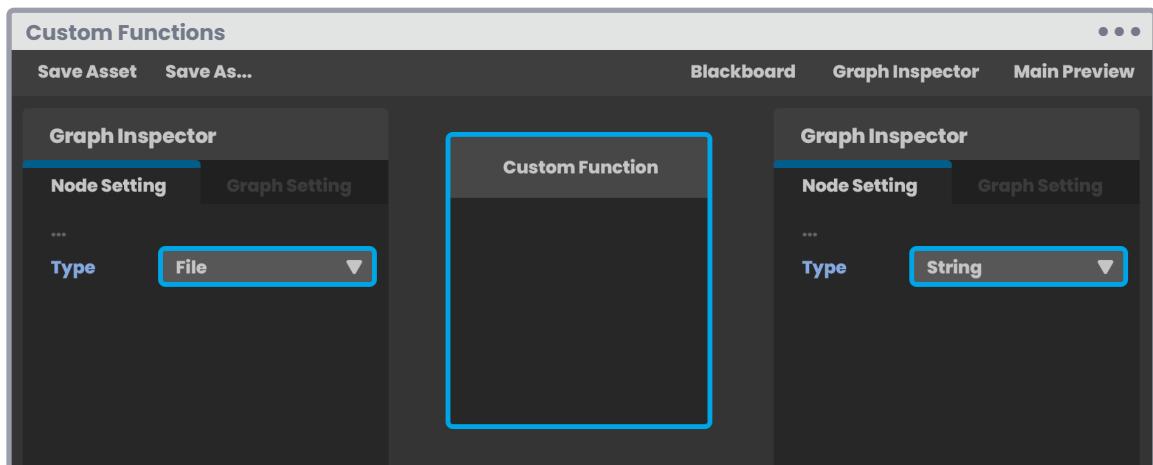
Since Shader Graph is based on HLSL, you can use these same functions inside a shader with **.shader** extension, following the points mentioned in section 4.0.4.



9.0.7. Custom Functions.

It is essential to know basic concepts about Computer Graphics and how to write functions in HLSL to work with this node. As its name mentions, **Custom Functions** lets you create your own functions and work with them in Shader Graph. It is very useful for using custom lighting, complex operations, or for optimizing long processes.

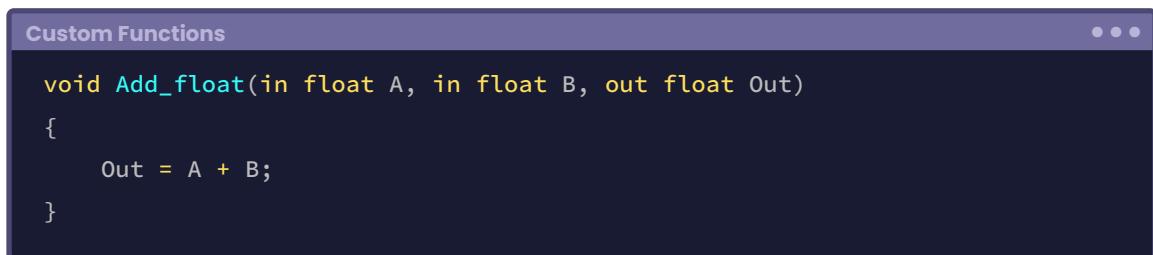
Given its structure, you can work with this node in two ways: from files with **.hsls** extension (type file) or by writing functions directly in its body (type string). However, it is advisable to use the first option because you can reuse the files later in other projects.



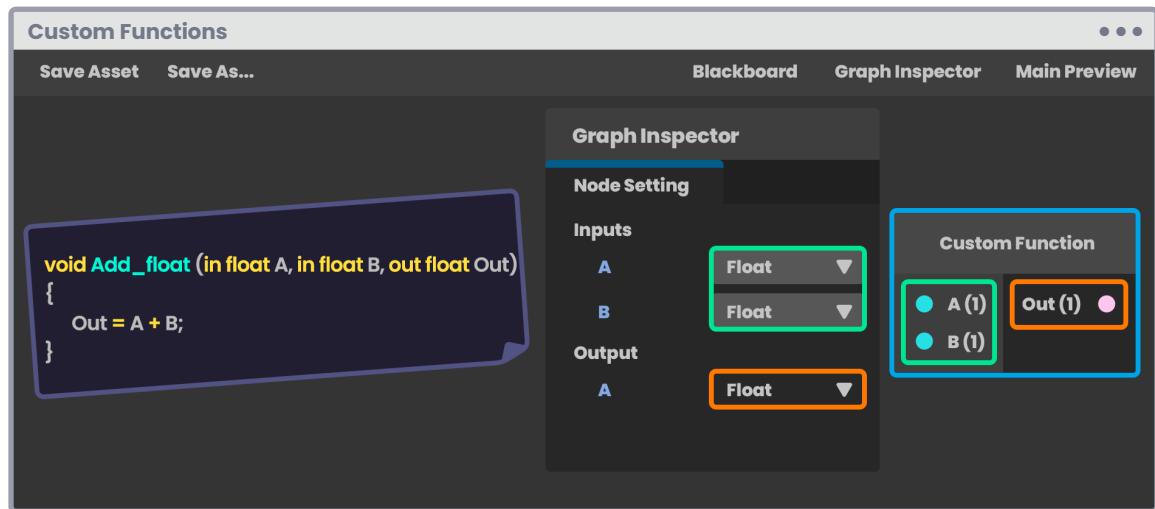
(Fig. 9.0.7a)

By default, Custom Functions has no input or output. Initially, you must go to the Graph Inspector and add the necessary properties for the function you want. This process may be different depending on the Unity version, e.g., since the 2019.3 version of the software does not have a Graph Inspector, you must press the configuration button (gear button) at the top right of the node to do the same thing.

Analyze this implementation of a simple function to understand the concept:



In the previous exercise, the **Add** function simply returns a scalar value (**Out**) equal to the sum of two numbers. The first input you can find corresponds to the floating type variable **A_{RG}** and the second is **B_{RG}**. Therefore, in Custom Function node, you should add two inputs and an output corresponding to scalar values.



(Fig. 9.0.7b)

The same analogy holds true for vectors and other data types, i.e., if **A_{RG}** were a three-dimensional vector, then such a vector would have to be added as input in the node configuration.

Next, recreate the behavior of the **_WorldSpaceLightPos** variable that was initially mentioned in section 7.0.3 talking about Diffuse Reflection, and use it for the same purpose. Start by creating a new Unlit Shader Graph type of shader and call it **USB_custom_function**.

First bring a Custom Function node to the node area. As mentioned above, you will need a script with the **.hlsl** extension. To generate it, simply create a file with the **.txt** extension and replace it with **.hlsl**, or create a script directly from the editor you are working with in Unity.

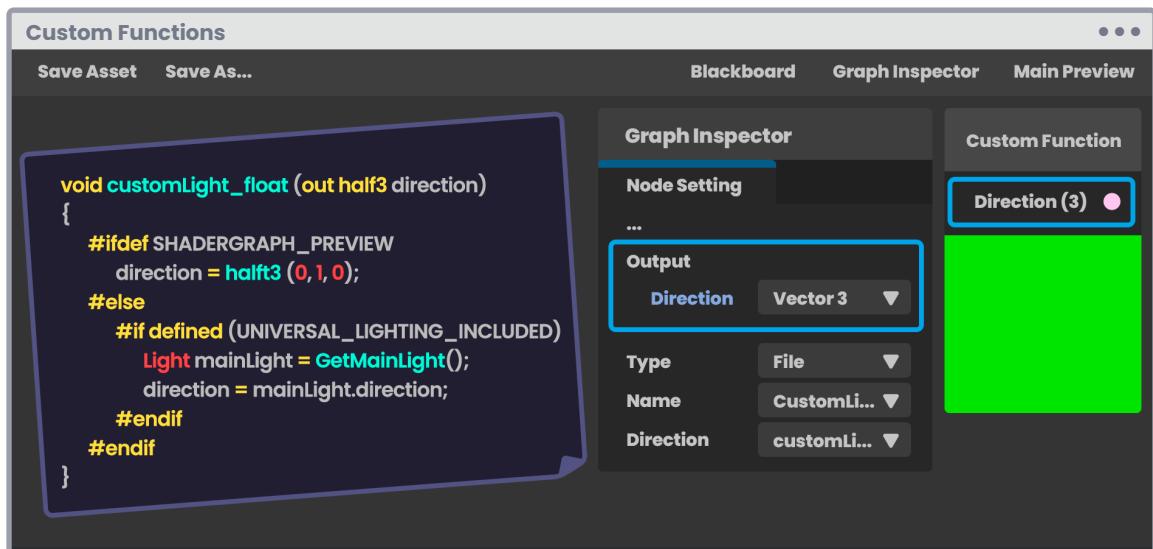
Call the .hsls file **CustomLight** and start adding your function as follows:

```
Custom Functions
...
CustomLight.hsls
void CustomLight_float(out half3 direction)
{
    #ifdef SHADERGRAPH_PREVIEW
        direction = half3(0, 1, 0);
    #else
        #if defined(UNIVERSAL_LIGHTING_INCLUDED)
            Light mainLight = GetMainLight();
            direction = mainLight.direction;
        #endif
    #endif
}
...
```

In the previous code block, you can deduce that if the preview in Shader Graph is enabled (SHADERGRAPH_PREVIEW), you will project lighting for ninety degrees on the **Y_{AX}**. Otherwise, if **Universal RP** has been defined, then the output **direction** will be the same as the direction of the main light, that is, to the directional light you have in the scene.

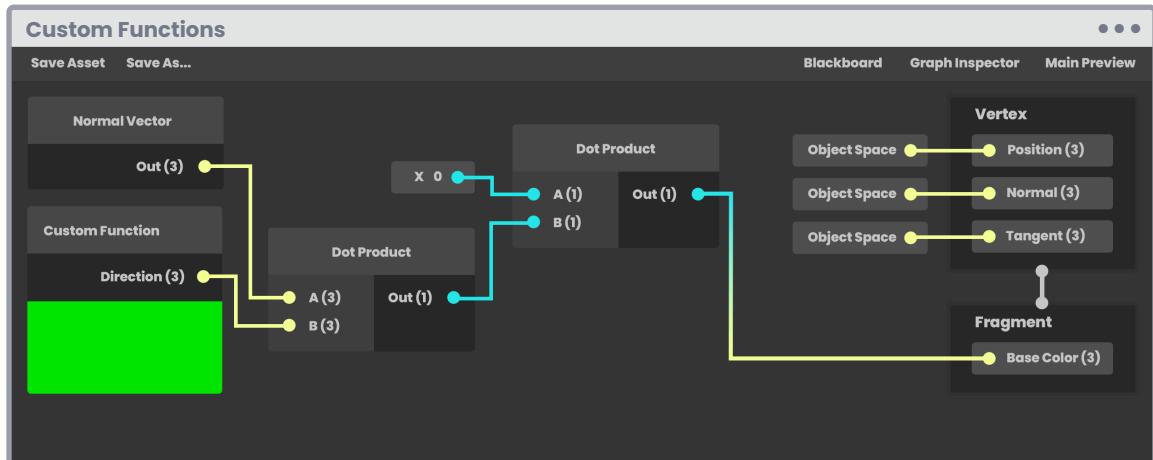
Unlike the previous case, the **CustomLight** function has no inputs; therefore, a three-dimensional vector will have to be added as output later in the node configuration. On the other hand, it's worth noting that such output is of **half3** type. Consequently, the accuracy of the node will be equal to a 16-bit value because, by default, it is configured as **inherit**.

Next, you must make sure to drag or select the **.hsls** file in the **Source** box located in the Graph Inspector **Node Settings** window. You must be sure to use the same name of the function in the **Name** box; that is, CustomLight, otherwise it could generate compilation errors.



(Fig. 9.0.7c. The Custom Function node is green due to the values of the direction vector)

Since the enclosed variable **_WorldSpaceLightPos** has the light position, you can use the node to replicate the same behavior; in fact, the operation **mainLight.direction** is the same as the variable **_MainLightPosition** included. You can check this by going to the project's **Lighting.hlsl** file.



(Fig. 9.0.7d. $D = \max(0, N \cdot L)$)

You can generate the same behavior through nodes in the Shader Graph as shown in Figure 9.0.7d, following the diffusion scheme seen in section 7.0.3.



Chapter III

Compute Shader, Ray Tracing and Sphere Tracing.

Advanced concepts.

In the previous chapters, the focus was on the understanding of three types of programmable shaders: Vertex, Fragment, and Surface Shader, whose properties, data types, structure, functions, among other topics were reviewed.

In this chapter, you will go deeper into more advanced concepts, and for this, investigate **.compute** type shaders. You will also practice two rendering techniques that require a high level of computing and mathematical understanding: Ray Tracing and Sphere Tracing.

Compute Shader is a program that lets you implement data algorithms directly into the compute units (graphic card), generating high-quality effects through parallel processes. Given its capacity, it is pretty helpful for general-purpose GPU programming (GPGPU) which refers to using functions for processing applications that are not necessarily of a graphic nature, e.g., calculations of millions of vertices or multiple instances of an object.

A fundamental feature of the Compute Shader is its Render Pipeline. It has its own graphical pipeline, which is not part of those mentioned in previous chapters. However, because this shader is part of the Direct3D API, it can link the output directly over the logical Render Pipeline mentioned in section 1.0.7.

Ray Tracing and Sphere Tracing refer to a set of techniques or algorithms for lighting rendering and physical materialization. Both work by tracing a ray; a line in the scene determining the object's distance according to the camera's position. This task is performed in three main parts:

- Ray generation (starting point).
- Ray intersection (the point where it hits the object).
- Shading (illumination, shadow, and surface).

In their implementation these algorithms increase the project's graphic quality, generating a very polished and realistic look; they also cause a significant load on the GPU, which makes their application incompatible with mid-range or low-end devices, e.g., mobile devices.

In this chapter, a factor to consider is the use of **High Definition RP** for the function and algorithm implementation in **.raytrace** type shaders. It is worth noting that DXR functions have technical

limitations; therefore, it is recommended that the reader has a high-end computer with the following technical characteristics for good graphics performance:

- Windows 10, 1809+ version.
- DirectX 12.
- NVIDIA series 20+ (2060, 2070, 2080 and its TI variants).

Ray Tracing also works on NVIDIA, Turing generation, and Pascal (GTX 1060+) cards. However, its performance is limited compared to those mentioned above.

10.0.1. Compute Shader structure.

Up to this point, the focus has been on studying **Unlit** and **Surface** shaders, which have a very similar structure; both are executed within the ShaderLab field, which, as you already know, is a declarative language that allows communication between the program and Unity. However, there is another type of shader called Compute, which has a similar structure to the above-mentioned but does not include Built-in shader variables that facilitate its programming.

You will begin this section creating a Compute Shader for it:

- 1 Go to your project folder in Unity.
- 2 Right-click.
- 3 Select Create / Shaders / **Compute shader**.
- 4 Call it **USB_simple_color_CS**.

Later you will work with this shader to understand the basic structure in color, UV coordinates, and texture implementation. Therefore, this will not be pretty at a functional level, but it will serve to illustrate the syntax behind the program.

Once opened, you will get a structure like the following:

```
Compute Shader structure • • •
```

```
// Each Kernel tells which function to compile;
// you can have many Kernels
#pragma kernel CSMain

// Create a RenderTexture with enableRandomWrite
// and set it with cs.SetTexture
RWTexture2D <float4> Result;

[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    // TODO: insert actual code here
    Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0
}
```

In the example, you can see a basic Unity default color structure. In it, you can find the following components:

- The **kernel** of the **CSMain** function.
- A 2D texture for writing and reading, called **Result**.
- The number of threads used to process each texture texel (**numthreads**).
- A function called **CSMain**, which includes a semantic as argument and an RGBA color output.

Such a structure has similarities to a Vertex-Fragment Shader because both programs are written in the HLSL. Therefore, to understand their basic setup, you will make an analogy using the CSMain Kernel.

As mentioned in Chapter I, section 3.3.2 the Vertex Shader Stage is configured like this when you determine the pragma associated with its function. This means, for example, that the **vert** function must be declared as **vertex** in the pragma so that the GPU can recognize its nature within the Render Pipeline.

Compute Shader structure

```
// declare the vert function as vertex shader stage
#pragma vertex vert

// initialize the vert function
v2f vert(appdata v) { ... }
```

You can find the same behavior in the Fragment Shader Stage. If you want the default function called **frag** to compile as a Fragment Shader, you must declare it in its respective pragma.

Compute Shader structure

```
// declare the frag function as fragment shader stage
#pragma fragment frag

// initialize the frag function
fixed4 frag (v2f i) : SV_Target { ... }
```

A Compute Shader is no exception, if you want to send the **CSSMain** function to the compute units (physical unit where it performs the computation), then it must be defined as **kernel** in the pragma.

Estructura de un Compute Shader

```
// declare the CSSMain function as kernel
#pragma Kernel CSSMain

// initialize the CSSMain function
[numthreads(8, 8, 1)]
void CSSMain (uint3 id : SV_DispatchThreadID) { ... }
```

The smallest unit that a Compute shader can process corresponds to an independent **thread**, and the attribute [**numthreads**(x_{RG}, y_{RG}, z_{RG})] is directly related to this.

Threads perform the computation of the operation that you want to carry out, e.g., in the case of a texture, they are in charge of processing each texel that the image has.

By default, the program has a group of 64 threads. How can you determine this? Basically, by multiplying the values in X_{RG} , Y_{RG} and Z_{RG} included in the numthreads attribute.

- **numthreads** (x, y, z).
- $8 * 8 * 1 = 64$ threads per group.

The above values can be translated as:

- Eight columns of threads in the X-axis.
- Eight rows of threads in the Y-axis.
- One set of threads in the Z-axis.

When working with threads, the hardware divides the groups into sub-blocks called **Warp**s. The total number of threads per group must be a multiple of the **Warp** size (32 threads per group on NVIDIA cards) or a multiple of the **Wavefront** size (64 threads per group on ATI cards). Unity defines eight threads in both X_{RG} and Y_{RG} precisely to ensure that the program runs on both NVIDIA and ATI cards. You will review this and other attributes in detail later in this chapter, since some semantics are associated with the threads you will operate with. For now, continue defining the internal structure of your program.

The **RWTexture2D** variable called **Result** refers to a two-dimensional RGBA texture with **reading/writing** capability (RW). This feature allows data to be sent from the CPU to the GPU, processed in parallel, and then returned.

If you want to add a variable that only has the capability to write, you would have to add it without the RW prefix, e.g., **Texture2D**. Now, how could you determine the need for a variable in your program? For this, you will have to implement some functions in the Compute Shader.

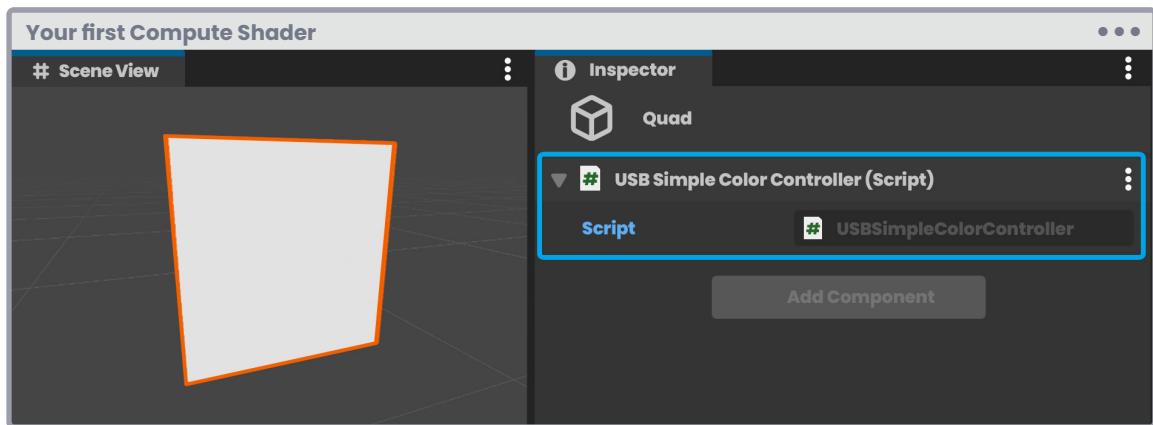
A Vertex-Fragment Shader requires the declarative language ShaderLab to communicate between Unity and the CGPROGRAM or HLSLPROGRAM. Analogously, a **.compute** shader requires a **C# script** for the same function. Every time you work with Compute Shaders, you will have to create and associate a C# script in the scene. This last one declares the global variables and buffers that you will connect later with the HLSL program.

A function that you will see recurrently throughout this chapter is **Dispatch**. This function is in charge of executing the **CSMain** Kernel, launching a certain number of thread groups in its XYZ dimensions. There are other concepts associated with the operation of these types of shaders that will be discussed later in this book.

10.0.2. Your first Compute Shader.

Continuing with **USB_simple_color_CS**, you will need to assign a color, texture, and UV coordinate to a 3D object. Add a **Quad** to the scene for this exercise and ensure it is centered in the grid, with its position and rotation set to "zero."

Create a C# script called **USBSimpleColorController.cs** and use it as a controller for the Compute shader. Since you want to write a texture on the object's material, you will have to assign the script directly to the 3D object.



(Fig. 10.0.2a. The *USBSimpleColorController* script has been assigned to the Quad you have in the scene)

Once the program is opened, you will see the following structure:

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

public class USBSimpleColorController : MonoBehaviour
{
    // Start is called before the first frame update.
    void Start()
    {
```

Continued on the next page.

```

}

// Update is called once per frame
void Update()
{
}

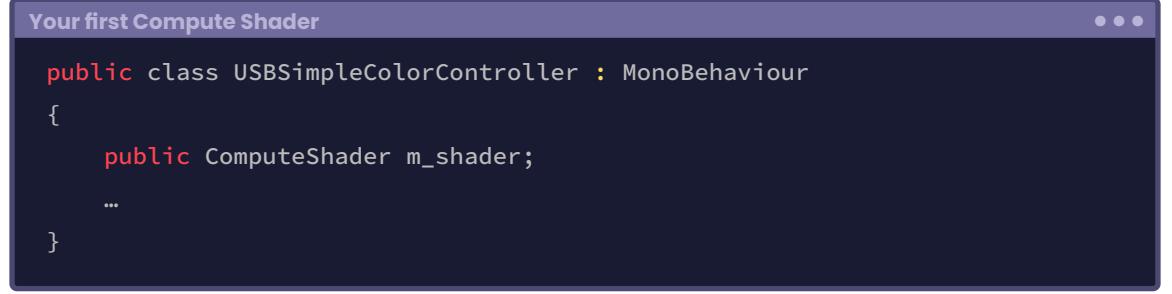
}

}

```

You can see that it corresponds to the default structure of a C# script. It includes start and update functions frame by frame to facilitate understanding.

Start by adding a global public variable to the program to connect the Compute shader. Call it **m_shader**.



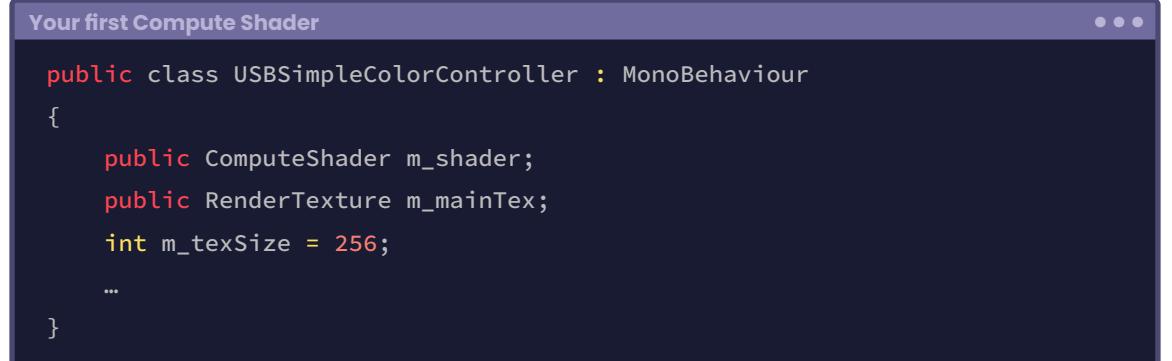
```

Your first Compute Shader

public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    ...
}

```

Assuming that you will write a **texture** on the Quad's material, you will need dimensions for width and height. The most common sizes for textures are values in powers of two, e.g., 128, 256, 512, 1024, etc. For that reason, declare a public variable of type **RenderTexture** for the texture and an integer value for its dimensions. Use 256 for both width and height.



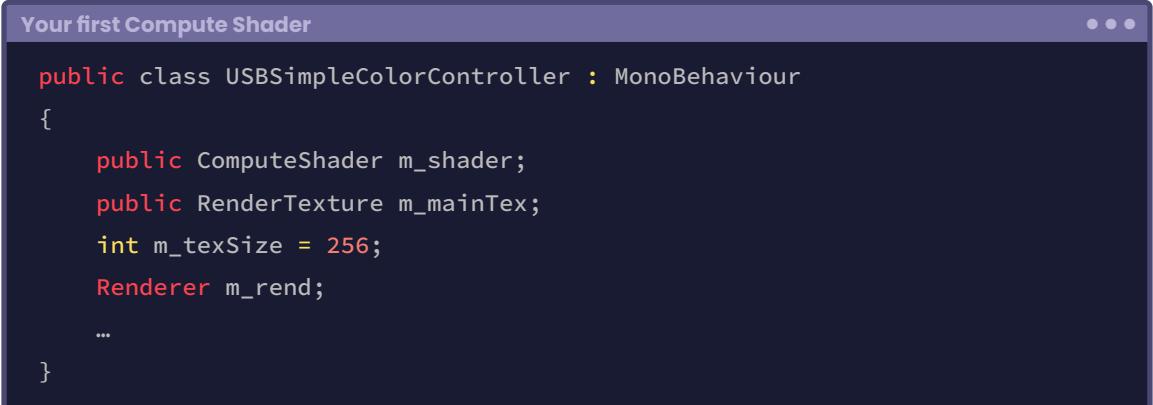
```

Your first Compute Shader

public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    ...
}

```

In the previous example, you declared a texture named **m_mainTex** and defined its dimension in the **m_texSize** variable. Its nature is associated with the **_MainTex** property that you frequently use to determine properties for the **.shader** shader. This suggests writing the variable **m_mainTex** on **_MainTex** later for color display. You now only need to define a variable to write the Quad material's texture.



```

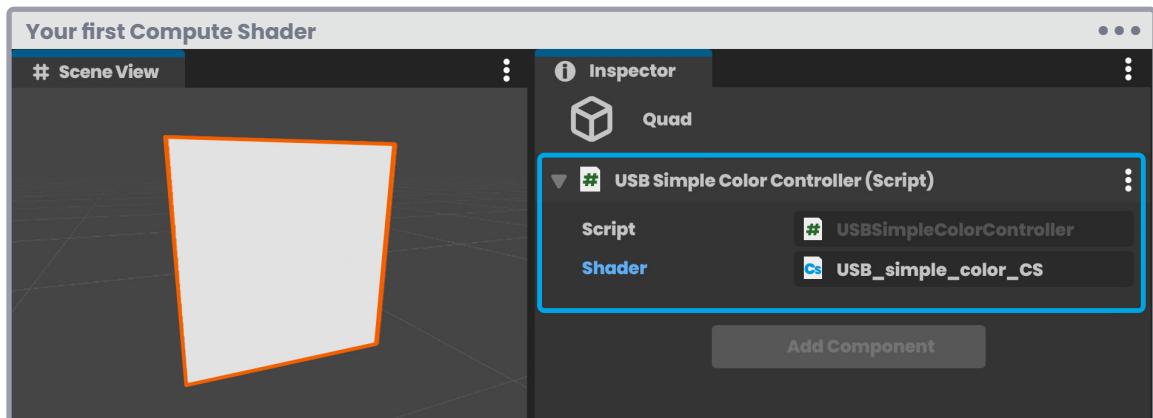
Your first Compute Shader

public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;
    ...
}

```

You will use the variable **m_rend** later to store the **Renderer** component of the material associated with the Quad, while the variable **m_mainTex** will be used to write the colors that you want to generate in the Compute shader.

Before continuing with the explanation, save the code you have added and go to the Unity Inspector to assign the Compute shader in its respective variable.



(Fig. 10.0.2b. The Compute shader has been assigned to the **m_shader** variable from the Unity Inspector)

Return to the script and initialize the texture in the **Start** method. For this, you need to use the variable **m_texSize** for both the texture's height and width.

Your first Compute Shader

```
public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;

    void Start ()
    {
        // initialize the texture.
        m_mainTex = new RenderTexture(m_texSize , m_texSize, 0,
        RenderTextureFormat.ARGB32);
    }
    ...
}
```

The `RenderTexture` class constructor has up to seven arguments; however, you only need four of them for the texture to work correctly. The first two arguments correspond to the texture's width and height, next the Depth Buffer, and finally, its configuration (32-bit RGBA). Now simply enable the random writing options and create the texture as such.

Your first Compute Shader

```
public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;

    void Start ()
    {
```

Continued on the next page.

```

m_mainTex= new RenderTexture(m_texSize , m_texSize, 0,
RenderTextureFormat.ARGB32);
// enable random writing
m_mainTex.enableRandomWrite = true;
// create the texture
m_mainTex.Create();
}
...
}

```

Since thread groups cannot synchronize with each other, you cannot determine which texel will be written to the texture first. For that reason, you must use the **enableRandomWrite** function before creating it. Finally, the **Create** function generates the texture; in fact, you can find the following text according to the official Unity documentation.

*The RenderTexture builder does not actually create the texture.
By default, the texture is created the first time it is activated. The texture is created in advance by calling the Create function.*



(Fig. 10.0.2c. Compute shaders can write texels arbitrarily on a texture)

The following process allows communication between the C# script and the Compute Shader. Start by saving the **Renderer** component (specific to the Quad material) in the **rend** variable that you created previously.

```
Your first Compute Shader

void Start ()
{
    m_mainTex = new RenderTexture(m_texSize, m_texSize, 0,
        RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    // get the material's renderer component
    m_rend = GetComponent<Renderer>();
    // make the object visible
    m_rend.enabled = true;
}
```

You now have the texture created, but it does not have any specific color, so what you must do next is to send it to the Compute Shader, assign it a color or design and then re-assign it to the material that the Quad is using so that it is visible in the scene. To do this, you can use the **ComputeShader.setTexture** function.

```
Your first Compute Shader

void Start ()
{
    m_mainTex = new RenderTexture(m_texSize, m_texSize, 0,
        RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    m_rend = GetComponent<Renderer>();
    m_rend.enabled = true;
    // send the texture to the Compute Shader
    m_shader.setTexture(0, "Result", m_mainTex);
}
```

The first argument in the function **SetTexture(K_{RG}, S_{RG}, T_{RG})** corresponds to the kernel index you are using in the Compute Shader.

Your first Compute Shader

```
// Each kernel tells which function to compile;  
// you can have many kernels  
#pragma kernel CSMain  
  
// Create a RenderTexture with enableRandomWrite and set it  
// with cs.SetTexture  
RWTexture2D <float4> Result;  
  
[numthreads(8, 8, 1)]  
void CSMain (uint3 id : SV_DispatchThreadID) { ... }
```

The **USB_simple_color_CS** shader has only one added default Kernel called **CSMain**. This occupies index “zero” since it is the only one in the program. A Compute Shader can have multiple Kernels, and each one has an **id** automatically assigned.

Your first Compute Shader

```
#pragma kernel CSMain           // id 0  
#pragma kernel CSFunction01     // id 1  
#pragma kernel CSFunction02     // id 2
```

The second argument in the function corresponds to the **name** of the buffer variable in the Compute Shader. By default, it is called **Result** and is a 2D RGBA texture with reading/writing capability.

Your first Compute Shader

```
RWTexture2D <float4> Result;
```

Finally, the third argument in the function corresponds to the texture that you will write on the buffer variable; in this case, called **m_mainTex**. You will process this variable in the Compute Shader within the CSMain method, you can corroborate this performing the operation in the function.

Your first Compute Shader

```
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    // TODO: insert actual code here
    Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0);
}
```

For the moment, the operation occurring in the CSMain method will not be gone into detail. Just continue implementing the texture on the material that the Quad currently has in the scene. To do this, return to the **USBSimpleColorController** script and pass the texture **m_mainTex** on the **_MainTex** property using the material's **SetTexture** function.

Your first Compute Shader

```
void Start ()
{
    m_mainTex= new RenderTexture(m_texSize , m_texSize, 0,
    RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    m_rend = GetComponent<Renderer>();
    m_rend.enabled = true;

    m_shader.SetTexture(0, "Result", m_mainTex);

    // send the texture to the Quad's material
    m_rend.material.SetTexture("_MainTex", m_mainTex);
}
```

By default, each Unity shader has the **_MainTex** property, so you can assume that the Quad material has it as well.

Now the process is almost ready; you only need to generate the groups of threads that will process each texel of the texture you are creating. For this, you must call the **Dispatch** function.

```
Your first Compute Shader

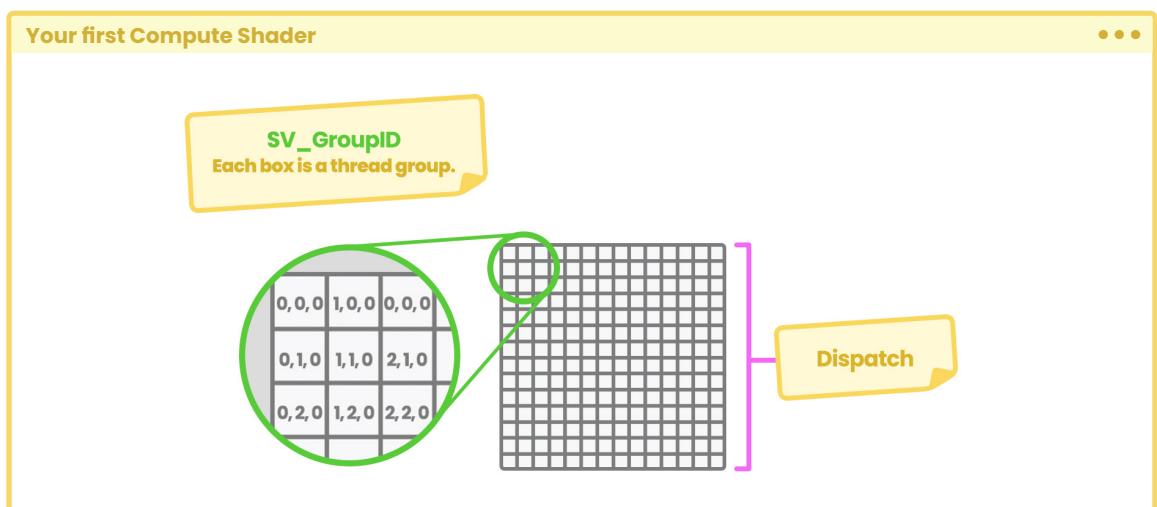
void Start ()
{
    m_mainTex = new RenderTexture(m_texSize, m_texSize, 0,
        RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    m_rend = GetComponent<Renderer>();
    m_rend.enabled = true;

    m_shader.SetTexture(0, "Result", m_mainTex);
    m_rend.material.SetTexture("_MainTex", m_mainTex);

    // generate the thread group to process the texture
    m_shader.Dispatch(0, m_texSize/8, m_texSize/8, 1);
}
```

The first argument in the function refers to the kernel; since you are only using CSMain, it will have to use the value “zero.” The following three values correspond to the grid (groups of threads) that you will generate to process the texture texels. The first value corresponds to the number of columns in the grid, then the rows, and finally the number of dimensions. As you already know, **m_texSize** equals 256; therefore, if you divide that value by 8, you will obtain a grid of 32 x 32 x 1.



(Fig. 10.0.2d. Each block in the grid represents a group of threads)

In GPU programming, the number of threads desired for execution is divided into a **thread group** grid, and it executes one thread group on each independent compute unit.

The operation of thread synchronization can occur only for those within the same group, generating more efficient parallel programming methods. Different groups of threads cannot be synchronized, in fact, you have no control over the order in which they will be processed, that is why they can be sent to different compute units.

Each of the blocks generated in the grid corresponds to a thread group. Now, how many threads does each group have? This value is determined by the **numthreads** attribute at the top of the CSMain function.

Your first Compute Shader

```
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID) { ... }
```

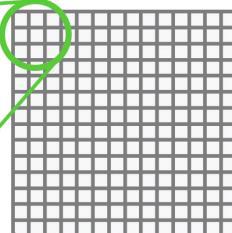
As you know, to calculate the number of threads within a group, you simply multiply the number of columns, by the number of rows, by the dimensions ($8 * 8 * 1$). In this configuration, there are 64 threads for each group.

Your first Compute Shader

numthreads (8, 8, 1)
SV_GroupThreadID
 Each box is a thread.

SV_GroupID
 Each box is a thread group.

0,0,0	1,0,0	0,0,0
0,1,0	1,1,0	2,1,0
0,2,0	1,2,0	2,2,0



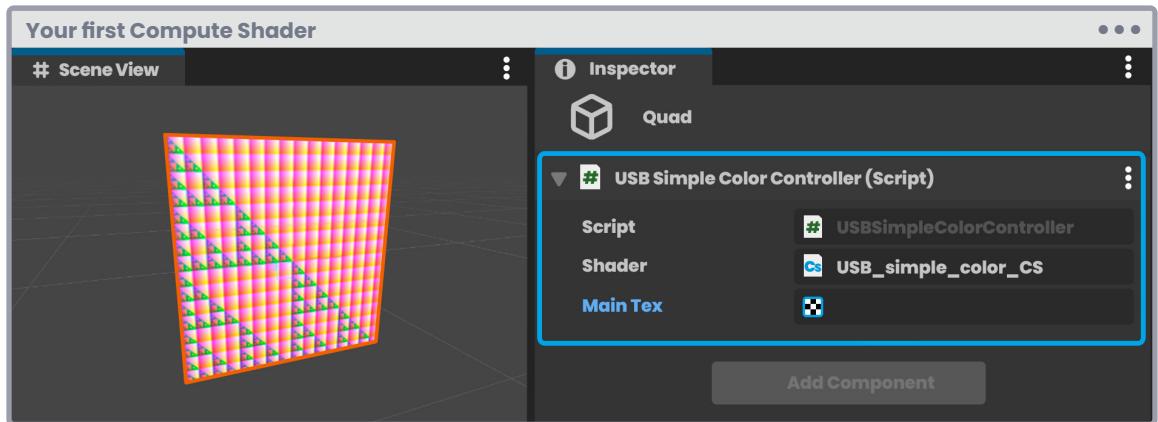
Dispatch

(Fig. 10.0.2e. The semantic **SV_GroupID** corresponds to the index of a group to be executed in the Compute Shader, **numthreads** refers to the total number of threads for each group, and **SV_GroupThreadID** refers to the identifier of each individual thread)

It is essential to delve into this process to understand the semantic `SV_DispatchThreadID`, which is found as an argument in the `CSMain` method. It corresponds to the sum of the number of threads for each group you are using, plus the index of each thread.

$$\text{SV_DispatchThreadID} = [(\text{SV_GroupID}) * (\text{numthreads})] + (\text{SV_GroupThreadID})$$

Going back to our **USBSimpleColorController.cs** program, if you save, go back to Unity and press the **play** button, you can see that a texture has been generated and is dynamically assigned to the Quad.



(Fig. 10.0.2f. The texture corresponds to the graphical representation of the Sierpinski fractal)

The texture you see in the figure above is being generated within the `CSMain` function, and its creation process is quite simple.

```
[Your first Compute Shader]
[ numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    // TODO: insert actual code here
    Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0);
}
```

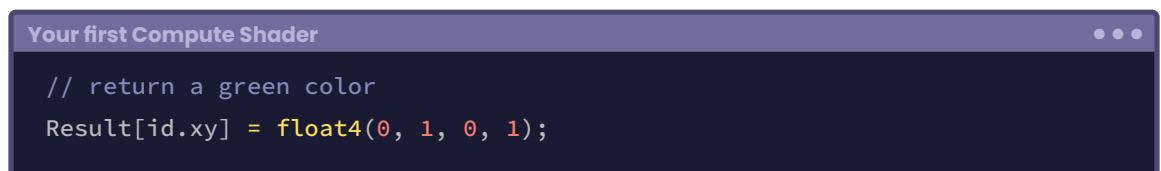
To understand, you should pay attention to the arguments of the CSMain function. The semantic **SV_DispatchThreadID** represents the indices of those combined threads and thread groups executed in the Compute Shader; this means that the identification of each thread is being stored in the **id** variable of type **uint3** (unsigned integer).

Unlike an **int** variable, **uint** variables only have positive numbers, starting at "zero." It makes sense since the indexes of each thread group begin at $0_X, 0_Y, 0_Z$ hence the **uint3** data type.

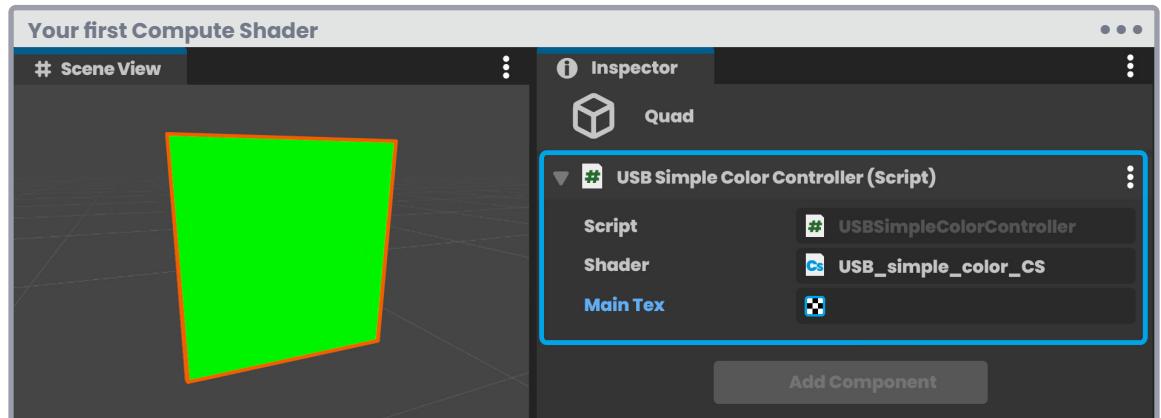


```
Your first Compute Shader
(uint3 id : SV_DispatchThreadID)
```

Each of the threads found in the **id** variable is processing the texels of the **Result** texture. Since it is a four-dimensional RGBA vector, you can return a solid color, e.g., green.



```
Your first Compute Shader
// return a green color
Result[id.xy] = float4(0, 1, 0, 1);
```



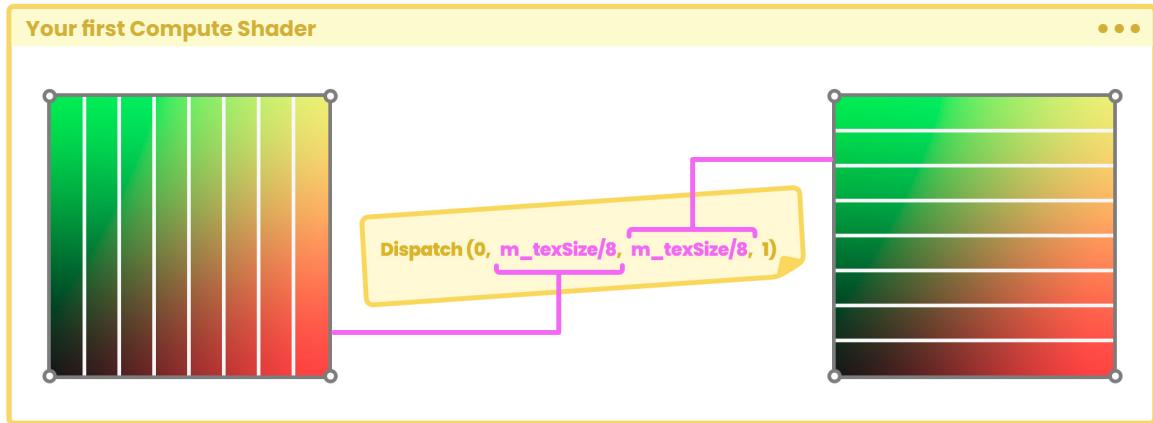
(Fig. 10.0.2g)

One factor to consider is the number of threads you will use in your Compute Shader, since it is directly related to the final result you get.

Earlier in the **Dispatch** function, you configured the width and height of the texture divided into eight to generate a grid of $32 \times 32 \times 1$ groups of threads. Why do you use the number eight as a

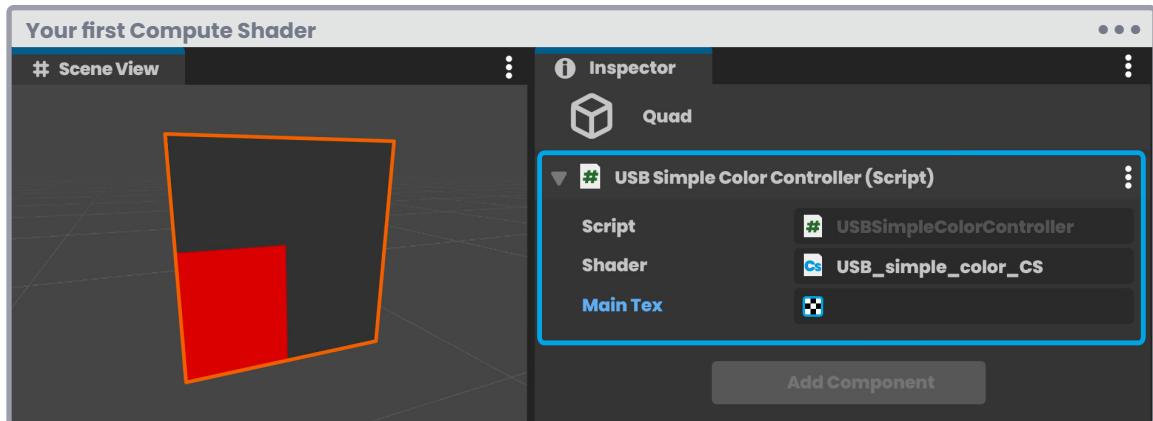
divisor in this operation? It is related to the configured number of threads in the **numthreads** component.

To understand the concept perform the mathematical operation: The texture you are creating has a width and height equal to 256 if you divide it by eight you get 32. Its graphic representation would equal eight rows or columns of thirty-two texels, each one above the texture.



(Fig. 10.0.2h)

To get 256 again, you must multiply by eight; this is precisely the number set as the total number of threads in both X_{RG} and Y_{RG} in the numthreads attribute. In fact, if you change the number of threads to 4 [`numthreads(4, 4, 1)`], you will notice that only $\frac{1}{4}$ of the texture is rendered in the Quad.



(Fig. 10.0.2i. Color (1, 0, 0, 1))

This factor occurs when multiplying 32 by 4, resulting in 128, precisely $\frac{1}{4}$ of the texture you are generating.

10.0.3. UV coordinates and texture.

The previous section defined a texture and its dimensions through the variables `m_mainTex` and `m_texSize` respectively. However, the final result corresponds to the graphical representation of the Sierpinski triangle.

Considering the script you have been writing up to this point, in this section, you will assign an imported texture to the effect, and for this, you will have to define UV coordinates.

Start by declaring a new public texture of type **Texture** called **`m_tex`** in your C# script.

```
UV coordinates and texture
public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public Texture m_tex;

    RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;
    ...
}
```

Since the data corresponds to a texture, later you will have to declare **Texture2D** and **SamplerState** variables in the Compute Shader.

```
UV coordinates and texture
RWTexture2D<float4> Result;

Texture2D<float4> ColTex;
SamplerState sampler_ColTex;
...
```

As you can see in the example above, the declaration for a texture has the same analogy as those seen in previous sections. In the same way, you will need UV coordinates to position the

texture on the Quad you are using. To do this, use the **GetDimensions(W_{RG}, H_{RG})** function within the **CSSMain** Kernel.

UV coordinates and texture

```
RWTexture2D<float4> Result;  
  
Texture2D<float4> ColTex;  
SamplerState sampler_ColTex;  
  
[numthreads(8, 8, 1)]  
void CSSMain (uint3 id : SV_DispatchThreadID)  
{  
    uint width;  
    uint height;  
    Result.GetDimensions(width, height);  
    ...  
}
```

The **GetDimensions(W_{RG}, H_{RG})** function is of type “void,” which means you are saving the dimensions of the Result variable in **width** and **height**, which correspond to the value you assigned to the variable “m_texSize” from **USBSimpleColorController**.

UV coordinates and texture

```
void GetDimensions(out uint width, out uint height);
```

You can now determine the values of the UV coordinates as follows:

```
UV coordinates and texture • • •
```

```
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width;
    uint height;

    Result.GetDimensions(width, height);

    float2 uv = float2(id.xy / float2(width, height));
    ...
}
```

Like width and height, the **id** variable is also an integer value, and the UV coordinates correspond to a range from 0.0f to 1.0f; that is why they have been declared as **float** type variables in the previous example.

It is worth noting that such an operation has a technical aspect that you must evaluate according to the texture **Wrap Mode** that you assign to the Quad.

The above operation works perfectly for those textures declared in **Clamp** mode. If the texture has been configured as **repeat**, you will need to add 0.5f to the **id** variable; otherwise, it will reflect part of the edges in its projection.

```
UV coordinates and texture • • •
```

```
// if the texture has been configured as Wrap Mode = clamp
float2 uv = float2(id.xy / float2(width, height));

// if the texture has been configured as Wrap Mode = repeat
float2 uv = float2((id.xy + float2(0.5, 0.5)) / float2(width, height));
```

Next, you can then use the **SampleLevel($S_{RG}, UV_{RG}, LOD_{RG}$)** function to determine the texture in the Kernel.

UV coordinates and texture

```
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width;
    uint height;

    Result.GetDimensions(width, height);

    float2 uv = float2(id.xy / float2(width, height));
    float4 col = ColTex.SampleLevel(sampler_ColTex, uv, 0);

    Result[id.xy] = col;
}
```

Such a function can return a scalar value or multidimensional vector. The previous exercise has been used to store the RGBA sampling values in the vector "col."

UV coordinates and texture

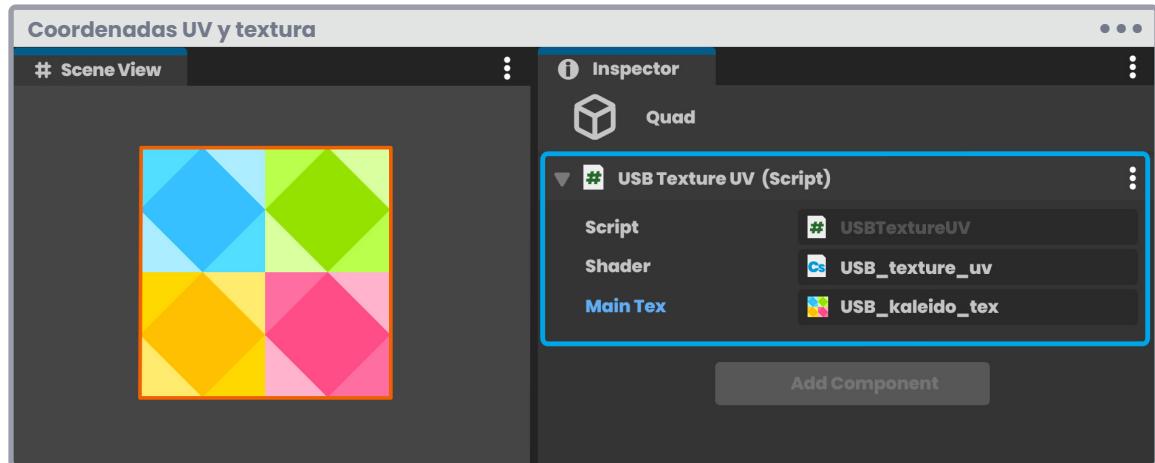
```
Object.SampleLevel(in SamplerState s, in float2 uv, in int LOD);
```

To conclude, you must return to the **USBSimpleColorController** script and send the texture to the Compute Shader through the **SetTexture** function, in the same way you did with the "m_mainTex" variable.

UV coordinates and texture

```
void Start()
{
    ...
    m_shader.SetTexture(0, "Result", m_mainTex);
    m_shader.SetTexture(0, "ColTex", m_tex);
    m_rend.material.SetTexture("_MainTex", m_mainTex);
    ...
}
```

If everything goes well, you will see the texture assigned in the **m_tex** field projected on your Quad.



(Fig. 10.0.3a. The texture has been configured in Clamp mode)

10.0.4. Buffers.

There are some cases where it will be necessary to process multiple data simultaneously, e.g., particle development, post-processing, Ray Tracing functions, simulations, and more. These are characterized by the computational units' extensive graphics load they generate. However, to your benefit, there are two associated data types that you can use in your program to speed up the reading and writing of values in the memory buffer:

- **ComputeBuffer.**
- And **StructuredBuffer.**

As its name mentions, **ComputeBuffer** corresponds to a buffer, which you can create and fill with a list of values from your C# script.

A **StructuredBuffer** is essentially the same, except that you declare it in the Compute Shader.

```
Buffers
```

```
// ----- C#
struct Properties
{
    Vector3 vertices;
    Vector3 normals;
    Vector4 tangents;
}
Properties[] m_meshProp;
ComputeBuffer m_meshBuffer;

// ----- Compute Shader
struct Properties
{
    float3 vertices;
    float3 normals;
    float4 tangents;
};

StructuredBuffer<Properties> meshProp;
```

You will create a new Compute Shader in the project to understand its implementation, which you will call **USB_compute_buffer**. In the same way, you will create a new C# script which you will call **USBComputeBuffer**.

Previously, in section 4.1.6, you created a simple method called **circle** that was used to reproduce a circle in a Quad. In this section, you will perform the same exercise with the difference that you will use the scripts previously created for this purpose.

Having studied part of the Compute Shader integration in Unity, you can deduce that **USBComputeBuffer** will handle the data configuration through the **SetFloat(S_{RG}, N_{RG})** and **SetTexture(K_{RG}, S_{RG}, T_{RG})** functions.

In the same way, using the **ComputeBuffer.SetBuffer** function, you will configure data that you will send later to the predefined list of values in the Compute Shader

Start by declaring the public variables associated with the “circle” function detailed in section 4.1.6.

Buffers

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

public class USBComputeBuffer : MonoBehaviour
{
    public ComputeShader m_shader;

    [Range(0.0f, 0.5f)] public float m_radius = 0.5f;
    [Range(0.0f, 1.0f)] public float m_center = 0.5f;
    [Range(0.0f, 0.5f)] public float m_smooth = 0.01f;
    public Color m_mainColor = new Color();

    private RenderTexture m_mainTex;
    private int m_texSize = 128;
    private Renderer m_rend;

    ...
}
```

If you pay attention to the example above, you will notice that the same properties (`m_radius`, `m_center`, and `m_smooth`) have been defined that you used previously for the generation of a circle. In addition, a color property has been created.

Such variables can be sent individually to the Compute Shader using the **ComputeShader.SetFloat** function or also create a buffer containing the complete list of values you want to assign.

To do this declare a structure and a buffer associated with the list of values you will use in the shader.

Buffers

```

public class USBComputeBuffer : MonoBehaviour
{
    public ComputeShader m_shader;

    [Range(0.0f, 0.5f)] public float m_radius = 0.5f;
    [Range(0.0f, 1.0f)] public float m_center = 0.5f;
    [Range(0.0f, 0.5f)] public float m_smooth = 0.01f;
    public Color m_mainColor = new Color();

    private RenderTexture m_mainTex;
    private int m_texSize = 128;
    private Renderer m_rend;

    // declare a struct with the list of values
    struct Circle
    {
        public float radius;
        public float center;
        public float smooth;
    }

    // declare an array type Circle to access each variable
    Circle[] m_circle;

    // declare a buffer of type ComputeBuffer
    ComputeBuffer m_buffer;

    ...
}

```

The variables have been declared inside the **Struct Circle** that you will use later in the ComputeShader; you will access each instance through “m_circle.” Given the nature of this exercise, you can deduce that the values of the global variables will be assigned to those defined in the struct. At this point, the **ComputeBuffer** enters the game since, once the list is filled with values, you must copy the data to the buffer and finally pass it to the shader.

Start by creating the texture before performing this process. To do so, declare a new method which you will call **CreateShaderTex**. This will contain the algorithm described in section 10.0.2 for the texture definition.

```
Buffers ...  

void Start()  

{  

    CreateShaderTex();  

}  

void CreateShaderTex()  

{  

    // first, create the texture  

    m_mainTex = new RenderTexture(m_texSize, m_texSize, 0,  

    RenderTextureFormat.ARGB32);  

    m_mainTex.enableRandomWrite = true;  

    m_mainTex.Create();  

    // then access the mesh renderer  

    m_rend = GetComponent<Renderer>();  

    m_rend.enabled = true;  

}
```

Next, declare a new function which you will use in the **Update** method for study purposes only.

```
Buffers ...  

void Update()  

{  

    SetShaderTex();  

}  

void SetShaderTex()  

{  

    // write the code here ...
}
```

Before continuing, go to **USB_compute_buffer**, since you must configure its structure before bringing the data from **USBCcomputeBuffer**. Add the **circle** function and then define its values in the CSMain Kernel.

```
Buffers ...
```

```
#pragma kernel CSMain

RWTexture2D<float4> Result;

// declare function
float CircleShape (float2 p, float center, float radius, float smooth)
{
    float c = length(p - center);
    return smoothstep(c - smooth, c + smooth, radius);
}

[numthreads(128, 1, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width;
    uint height;
    Result.GetDimensions(width, height);
    float2 uv = float2((id.xy + 0.5) / float2(width, height));

    // initialize the values to zero
    float c = CircleShape(uv, 0, 0, 0);

    Result[id.xy] = float4(c, c, c, 1);
}
```

In the previous exercise, you defined the **CircleShape** method, which is the same as the **circle** function. Such a function has been initialized in the CSMain Kernel, with its values set to "zero." Consequently, its output corresponds to black by default.

Note that there are 128 threads in \mathbf{X}_{RG} for the operation this is mainly due to two factors:

- The texture size of the variable **m_texSize** is equal to 128.
- You only need one dimension to run the previously defined **m_circle** list.

Next, define the buffer that will contain the variables necessary for the correct operation of the CircleShape method.

```
Buffers ...
```

```
#pragma kernel CSMain
RWTexture2D<float4> Result;
float4 MainColor;

// declare the array of values
struct Circle
{
    float radius;
    float center;
    float smooth;
};

// declare the buffer
StructuredBuffer<Circle> CircleBuffer;

// declare the function
float CircleShape (float2 p, float center, float radius, float smooth)
{
    float c = length(p - center);
    return smoothstep(c - smooth, c + smooth, radius);
}

[numthreads(128, 1, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width;
    uint height;
```

Continued on the next page.

```

Result.GetDimensions(width, height);

float2 uv = float2((id.xy + 0.5) / float2(width, height));

// access the array values
float center = CircleBuffer[id.x].center;
float radius = CircleBuffer[id.x].radius;
float smooth = CircleBuffer[id.x].smooth;

// initialize the values
float c = CircleShape(uv, center, radius, smooth);

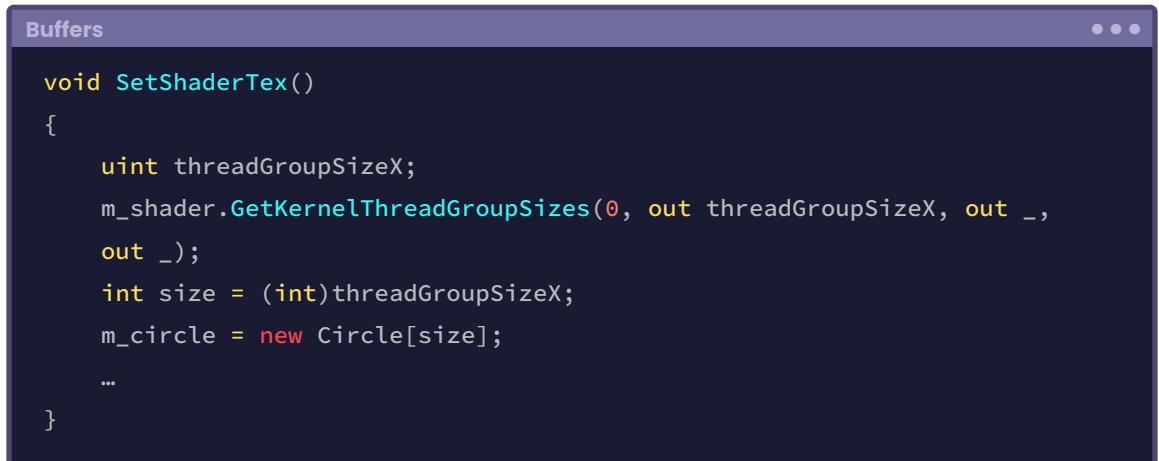
Result[id.xy] = float4(c, c, c, 1);
}

```

As in **USBComputeBuffer**, it has specified a list of scalars inside a Struct called Circle. These variables match quantity and data type with those declared in the C# script.

Subsequently, it has declared a **StructuredBuffer** called CircleBuffer. This will handle storing the values you send from USBComputeBuffer.

You only need to complete the **SetShaderTex** function's operations and send them to the Compute Shader. To do this, you must return to **USBComputeBuffer**.



```

Buffers

void SetShaderTex()
{
    uint threadGroupSizeX;
    m_shader.GetKernelThreadGroupSizes(0, out threadGroupSizeX, out _, 
    out _);
    int size = (int)threadGroupSizeX;
    m_circle = new Circle[size];
    ...
}

```

In the example the exercise was started by declaring a variable of type “unsigned integer” called **threadGroupSizeX**. This is due to the Void type function called **GetKernelThreadGroupSizes**,

which takes the group of threads that have been configured in the Kernel, that is, the variable mentioned above will be 128.

```
Buffers ...
```

```
// compute shader
[numthreads(128, 1, 1)]

// C#
GetKernelThreadGroupSizes(kernel, 128, 1, 1);
```

Finally, this value has been added to the **m_circle** list which you will use as **data** for the buffer. Next, assign the public variables to those declared within the list. You can simply initialize a loop and pass the values to each variable separately.

```
Buffers ...
```

```
void SetShaderTex()
{
    uint threadGroupSizeX;
    m_shader.GetKernelThreadGroupSizes(0, out threadGroupSizeX, out _, out _);
    int size = (int)threadGroupSizeX;
    m_circle = new Circle[size];

    for(int i = 0; i < size; i++)
    {
        Circle circle = m_circle[i];
        circle.radius = m_radius;
        circle.center = m_center;
        circle.smooth = m_smooth;
        m_circle[i] = circle;
    }
    ...
}
```

Once the information is stored in the list, you can declare a new ComputeBuffer, configure the information, and then send the data to the Compute Shader.

```
Buffers ...
```

```

for(int i = 0; i < size; i++)
{
    Circle circle = m_circle[i];
    circle.radius = m_radius;
    circle.center = m_center;
    circle.smooth = m_smooth;
    m_circle[i] = circle;
}

int stride = 12;
m_buffer = new ComputeBuffer(m_circle, stride, ComputeBufferType.Default);
m_buffer.SetData(m_circle);
m_shader.SetBuffer(0, "CircleBuffer", m_buffer);
...

```

By default, the ComputeBuffer builder contains three arguments:

- The number of elements in the buffer.
- The size of the elements.
- And the type of buffer you are creating.

As you can see in the previous example, the data stored in the variable **m_circle** is used as the first argument. The second one (**stride**) is equal to the number of dimensions of those scalars you are passing, multiplied by the floating variable's bytes.

```
Buffers ...
```

```

struct Circle
{
    float radius; // One-dimensional float - 4 bytes
    float center; // One-dimensional float - 4 bytes
    float smooth; // One-dimensional float - 4 bytes
};

int stride = (1 + 1 + 1) * 4

```

The type of buffer being used in the third argument corresponds to the **StructuredBuffer** a Circle type that you declared earlier in the Compute Shader.

```
Buffers
StructuredBuffer<Circle> CircleBuffer;
```

After passing the data to the buffer through the **SetData** function, send the information to the **StructuredBuffer CircleBuffer** through the **SetBuffer** function.

Finally, you must configure the texture and pass the color **m_mainColor** to the four-dimensional vector **MainColor** found in the Compute Shader.

At the end of the process, when the buffer will no longer be used, you can call the **Release** or **Dispose** function, which manually frees the buffer.

```
Buffers
void SetShaderTex()
{
    ...
    int stride = 12;
    m_buffer = new ComputeBuffer(m_circle, stride,
        ComputeBufferType.Default);
    m_buffer.SetData(m_circle);
    m_shader.SetBuffer(0, "CircleBuffer", m_buffer);
    m_shader.SetTexture(0, "Result", m_mainTex);
    m_shader.SetVector("MainColor", m_mainColor);
    m_rend.material.SetTexture("_MainTex", m_mainTex);

    m_shader.Dispatch(0, m_texSize, m_texSize, 1);
    m_buffer.Release();
}
```

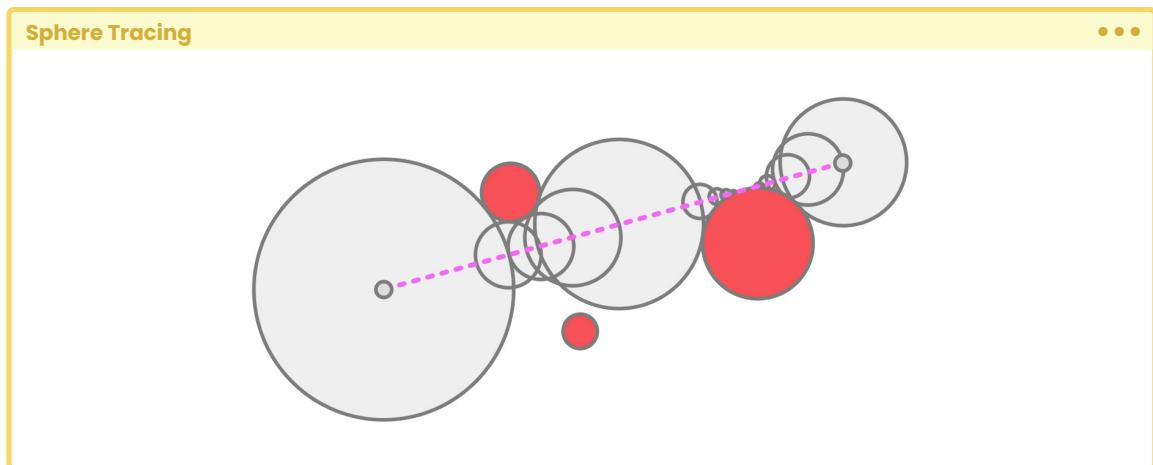
Sphere Tracing.

According to the publication in The Visual Computer (1995) by John C. Hart;

Sphere tracing is a technique for rendering implicit surfaces that use geometric distance.

What does the above statement refer to? Before going into detail, you will address some fundamental points to understand the concepts related to its function.

Sphere Tracing, Sphere Casting or Ray Marching refer to the same concept. Basically it is the process of “marching” along a ray, which is divided by points in space. This method is often used for volume rendering, where there is no specific surface; instead, you have to find the intersection between the ray and a surface defined by an “implicit distance” equation.



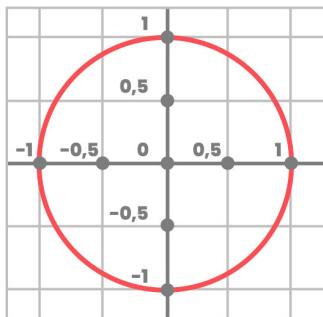
(Fig. 11.0.0a. Sphere Casting over three red spheres)

In differential calculus, you can find explicit and implicit algebraic and transcendental equations, e.g., the following implicit equation generates a three-dimensional sphere:

All its variables form part of the equation

$$x^2 + y^2 + z^2 - 1 = 0$$

Sphere Tracing



(Fig. 11.0.0b. With "Z" equal to 0.0f)

Which is the same as saying,

$$\| X \| - 1 = 0$$

Therefore,

Sphere Tracing

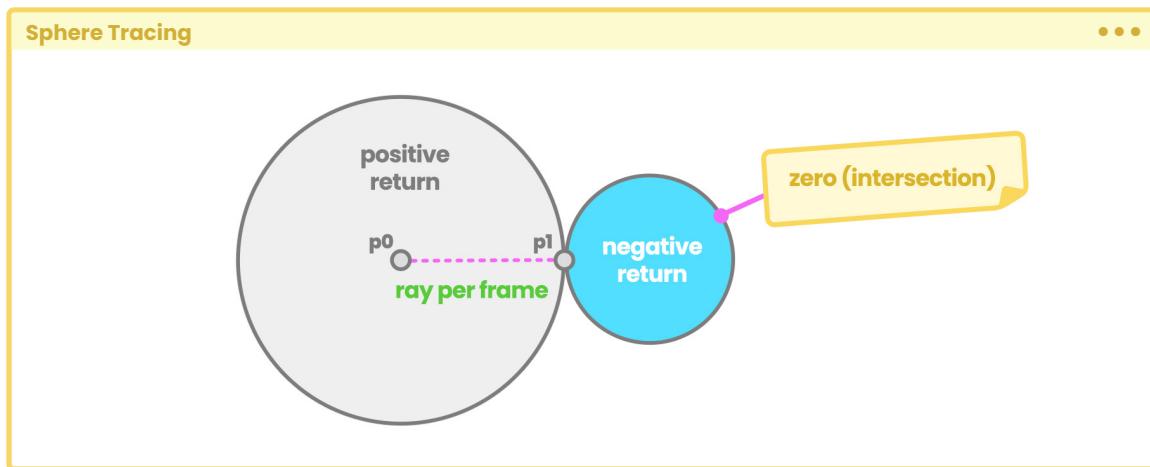
```
float sphereSDF(float3 p, float radius)
{
    float sphere = length(p) - radius;
    return sphere;
}
```

An implicit surface is defined by a function that, given a point in space, indicates that said point is inside or outside the surface.

A ray travels from the camera through a pixel until it hits a surface to achieve the objective. This concept is called Ray Casting, which is the process of finding the closest object along the ray, hence the name "sphere casting."

As you already know, a ray or line is composed of two points in space: a starting point and an endpoint. In this technique, the starting point corresponds to the camera's three-dimensional position, and the endpoint is the intersection of the surface you are hitting.

You have to consider the shape of the surface you will generate to determine the ray's endpoint. In this context, SDF (Signed Distance Functions) functions take a point as input and return the shortest distance between that point and the surface of a figure. If the return value is positive, the ray continues its path, while if the value is zero, the ray collides with a surface.



(Fig. 11.0.0c. The point "p0" represents the camera's position, while "p1" represents the collision ray point)

11.0.1. Implementing functions with Sphere Tracing.

It will be necessary to define at least two shader functions for this technique to work correctly. For this, you will have to consider:

- 1 An SDF function to determine the type of surface.
- 2 Another function to calculate the Sphere Casting.

In Unity, create a new Unlit shader and call it **USB_SDF_fruit**. To understand this concept, you will develop an effect called "sliced-fruit." You will divide a sphere into two parts to show its inside.

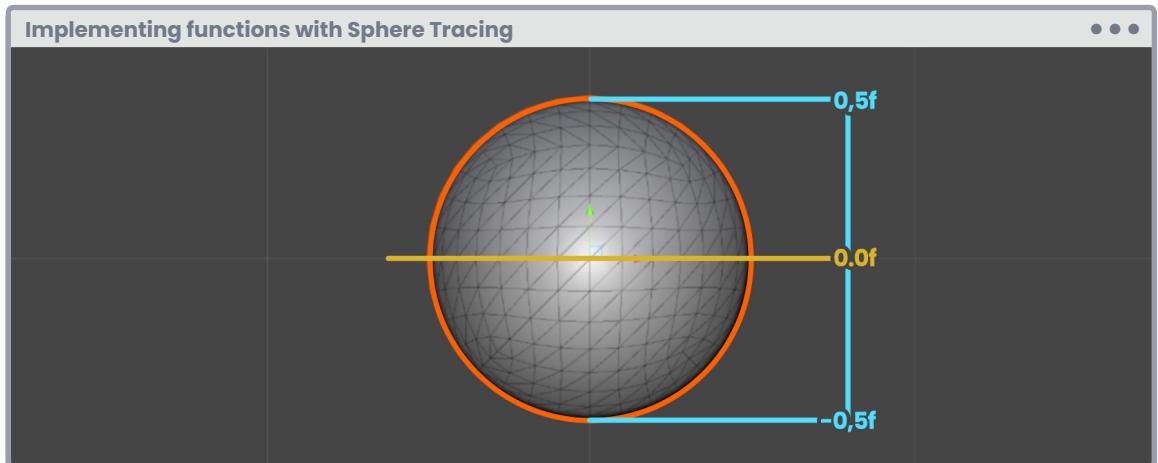
Note that you will apply this effect to a primitive sphere, ideally the one included in the 3D objects in Unity; why? Such a sphere has its pivot at the center of its mass and has a circumference equal to one. Therefore, it fits perfectly inside a grid block in our scene.

Start by declaring a new property in your shader, which will define the effect's border or division.

```
Implementing functions with Sphere Tracing • • •
```

```
Shader "USB/USB_SDF_fruit"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Edge ("Edge", Range(-0.5, 0.5)) = 0.0
    }
    SubShader
    {
        Pass
        {
            ...
            float _Edge;
            ...
        }
    }
}
```

If you pay attention to the **_Edge** property defined previously, you will notice that its range corresponds to a value between -0.5f and 0.5f. This is due to the volume or scale of the sphere you are working with.



(Fig. 11.0.1a)

You will use two primary operations to divide the sphere: discard the pixels on the **_Edge** and project a plane across its center. Discarding the pixels will optimize the effect and visualize the plane that you will generate later.

Continue by declaring an SDF function for calculating the plane.

Implementing functions with Sphere Tracing

```
Pass
{
    ...
    // declare the function for the plane
    float planeSDF(float3 ray_position)
    {
        // subtract the edge to the "Y" ray position to increase
        // or decrease the plane position
        float plane = ray_position.y - _Edge;
        return plane;
    }
    ...
}
```

Since **_Edge** is subtracting the position in Y_{AX} of the plane, the three-dimensional object, graphically, will have the ability to go up or down in space according to the value of the Property. Its implementation will be looked at later in this section. For now, some constants will be defined that will be used in the Sphere Casting calculation; determining the surface of the plane.

Implementing functions with Sphere Tracing

```
float planeSDF(float3 ray_position) { ... }

// maximum of steps to determine the surface intersection
#define MAX_MARCHIG_STEPS 50
// maximum distance to find the surface intersection
#define MAX_DISTANCE 10.0
// surface distance
#define SURFACE_DISTANCE 0.001
```

The **#define** directive lets you declare an identifier you can use as a global constant. The values associated with each macro have been determined according to their functionality, e.g., **MAX_DISTANCE** is equal to ten meters or ten grid blocks in the scene, while, **MAX_MARCHING_STEPS** refers to the number of steps you will need to find the plane's intersection.

Continue by declaring this function to perform Sphere Casting.

Implementing functions with Sphere Tracing

• • •

```
float planeSDF(float3 ray_position) { ... }

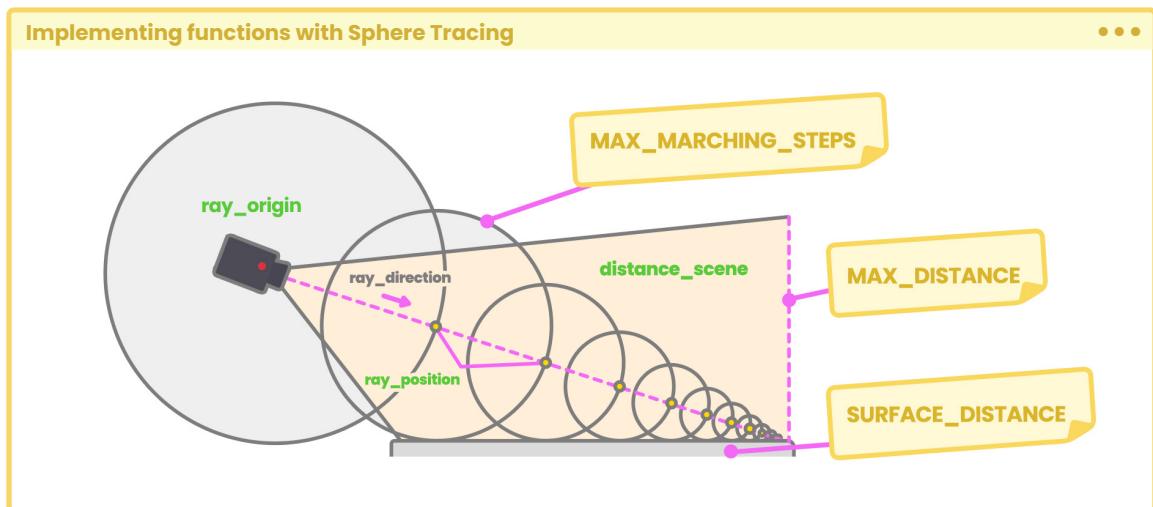
#define MAX_MARCHIG_STEPS 50
#define MAX_DISTANCE 10.0
#define SURFACE_DISTANCE 0.001

float sphereCasting(float3 ray_origin, float3 ray_direction)
{
    float distance_origin = 0;
    for(int i = 0; i < MAX_MARCHIG_STEPS; i++)
    {
        float3 ray_position = ray_origin + ray_direction * distance_origin;
        float distance_scene = planeSDF(ray_position);
        distance_origin += distance_scene;

        if(distance_scene < SURFACE_DISTANCE || distance_origin >
MAX_MARCHIG_STEPS);
            break;
    }

    return distance_origin;
}
```

At first glance, the operation in the above example looks challenging to understand. However, it is not that complex. Initially, pay attention to its arguments. The **ray_origin** vector corresponds to the ray's starting point, that is, the camera's position in World-Space, while **ray_direction** is the same as the position of the mesh vertices, in other words, as the position of the sphere you are working with, why? Since you will generate a division in the primitive according to an edge, you will need the position of the SDF plane to be equal to the position of the 3D object.



(Fig. 11.0.1b)

As mentioned above, it will be necessary to discard the pixels of the sphere that lie on the plane. To perform such an evaluation, you will need the Object-Space position of the vertices on the Y_{AX} axis of the sphere. Using the **discard** declaration, you can discard those pixels that lie on edge, as shown in the following operation.

Implementing functions with Sphere Tracing

```
if (vertexPosition.y > _Edge)
    discard;
```

For this exercise, it will be necessary to declare a new three-dimensional vector in the Vertex Output, why? Because of their nature, pixels can only be discarded in the Fragment Shader Stage. Therefore, you will have to take the values from the Vertex Shader to the Fragment Shader Stage.

Create a new vector called "hitPos."

Implementing functions with Sphere Tracing

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 hitPos : TEXCOORD1;
};
```

This vector will have a double functionality in your effect. On the one hand, you will use it to define the position of the Mesh vertices; and, on the other, to calculate the spatial position of the plane so that both surfaces are located at the same point.

Implementing functions with Sphere Tracing

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    // assign the vertex position in Object-Space
    o.hitPos = v.vertex;

    return o;
}
```

Finally, you can discard pixels that lie in the `_Edge` property.

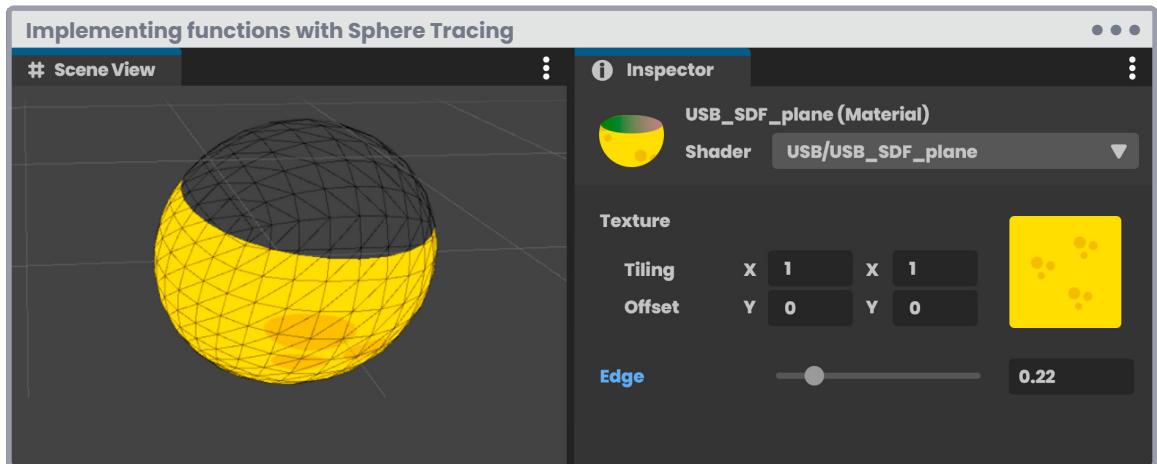
Implementing functions with Sphere Tracing

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    // we discard the pixels that lie on the _Edge
    if (i.hitPos > _Edge)
        discard;

    return col;
}
```

If everything before has been done correctly in Unity, you will see something similar to that of Figure 11.0.1c. The `_Edge` property will dynamically modify the discarded pixel border.



(Fig. 11.0.1c. _Edge equal to 0.251)

Note that the effect is “opaque,” that is, it has no transparency. Those discarded pixels will not be executed; therefore, they will not be sent to the output color.

As you can see in Figure 11.0.1b, the ray’s origin is given by the camera position, while the ray direction can be calculated by following the position of the vertices in the same space.

The declaration of these variables can be easily done in the Fragment Shader Stage using the **_WorldSpaceCameraPos** variable, and then passing the vertices to the function as shown in the following example:

```
Implementing functions with Sphere Tracing
```

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    // transform the camera to local-space
    float3 ray_origin = mul(unity_WorldToObject, float4(
        _WorldSpaceCameraPos, 1));

    // calculate the ray direction
    float3 ray_direction = normalize(i.hitPos - ray_origin);
```

Continued on the next page.

```

// pass the values to the function
float t = sphereCasting(ray_origin, ray_direction);

// calculate the spacial point of the plane
float3 p = ray_origin + ray_direction * t;

if (i.hitPos > _Edge)
    discard;

return col;
}

```

Looking at the previous example, you have stored the plane's distance in respect to the camera in the "t" variable and then stored each point in the variable "p." It will be necessary to project the SDF plane onto the front face of the sphere. Consequently, you will have to disable Culling from the SubShader.

Implementing functions with Sphere Tracing



```

SubShader
{
    ...
    // project both faces of the sphere
    Cull Off
    ...
    Pass
    {
        ...
    }
}

```

Using the **SV_isFrontFace** semantic, you can project the pixels of the sphere on its back face, and the plane on the front face.

```
Implementing functions with Sphere Tracing
```

```

fixed4 frag (v2f i, bool face : SV_isFrontFace) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);

    float3 ray_origin = mul(unity_WorldToObject, float4(
        _WorldSpaceCameraPos, 1));

    float3 ray_direction = normalize(i.hitPos - ray_origin);

    float t = sphereCasting(ray_origin, ray_direction);

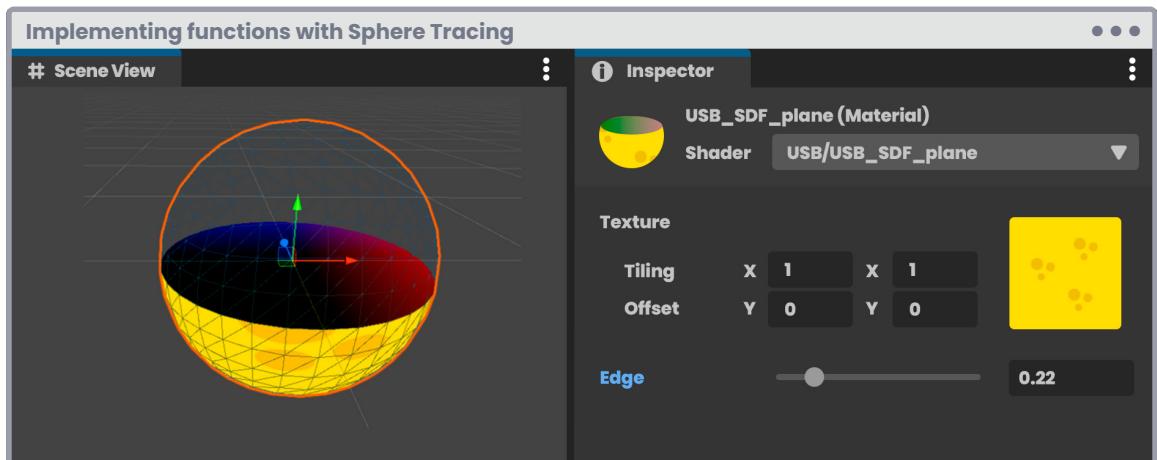
    float3 p = ray_origin + ray_direction * t;

    if (i.hitPos > _Edge)
        discard;

    return face ? col : float4(p, 1);
}

```

If you return to the scene, you can determine the position on the Y_{AX} axis of the SDF plane using the `_Edge` property, as shown in Figure 11.0.1d.



(Fig. 11.0.1d)

11.0.2. Projecting a texture.

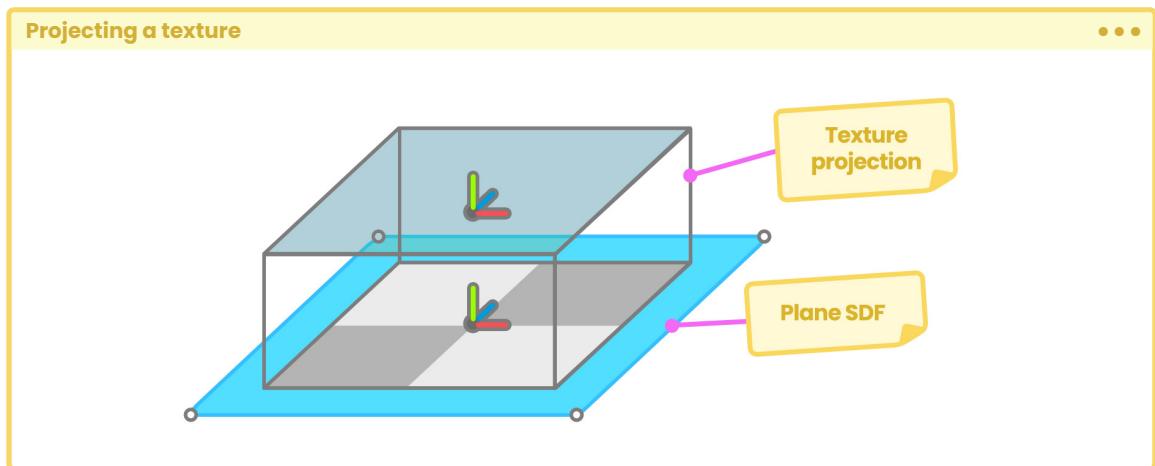
Continuing with the **USB_SDF_fruit** shader, this time, you will project a texture over the SDF plane that you generated previously. Start by adding some properties that you will use later in effect.

Projecting a texture



```
Shader "USB/USB_SDF_fruit"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        // plane texture
        _PlaneTex ("Plane Texture", 2D) = "white" {}
        // edge color projection
        _CircleCol ("Circle Color", Color) = (1, 1, 1, 1)
        // edge radius projection
        _CircleRad ("Circle Radius", Range(0.0, 0.5)) = 0.45
        _Edge ("Edge", Range(-0.5, 0.5)) = 0.0
    }
    SubShader
    {
        Pass
        {
            ...
            sampler2D _MainTex;
            sampler2D _PlaneTex;
            float4 _MainTex_ST;
            float4 _CircleCol;
            float _CircleRad;
            float _Edge;
            ...
        }
    }
}
```

For this case, if you want to project a texture on the SDF plane, you will have to consider the position and direction of the back face of the SDF plane; why? Remember that the plane is pointing towards the positive Y_{AX}. Therefore, the texture should be oriented in the opposite direction.



(Fig. 11.0.2a)

To do this, you can calculate UV coordinates inside the maximum Sphere casting area, using the point "p.xz."

```
Projecting a texture
```

```
fixed4 frag (v2f i, bool face : SV_isFrontFace) : SV_Target
{
    ...
    float t = sphereCasting(ray_origin, ray_direction);

    if(t < MAX_DISTANCE)
    {
        float3 p = ray_origin + ray_direction * t;
        float2 uv_p = p.xz;
    }

    if (i.hitPos > _Edge)
        discard;

    return face ? col : float4(p, 1);
}
```

Now you can use these coordinates in the **tex2D(S_{RG}, UV_{RG})** function, the same way as it has been done throughout the book.

Projecting a texture

```
fixed4 frag (v2f i, bool face : SV_isFrontFace) : SV_Target
{
    ...
    float t = sphereCasting(ray_origin, ray_direction);
    float4 planeCol = 0;

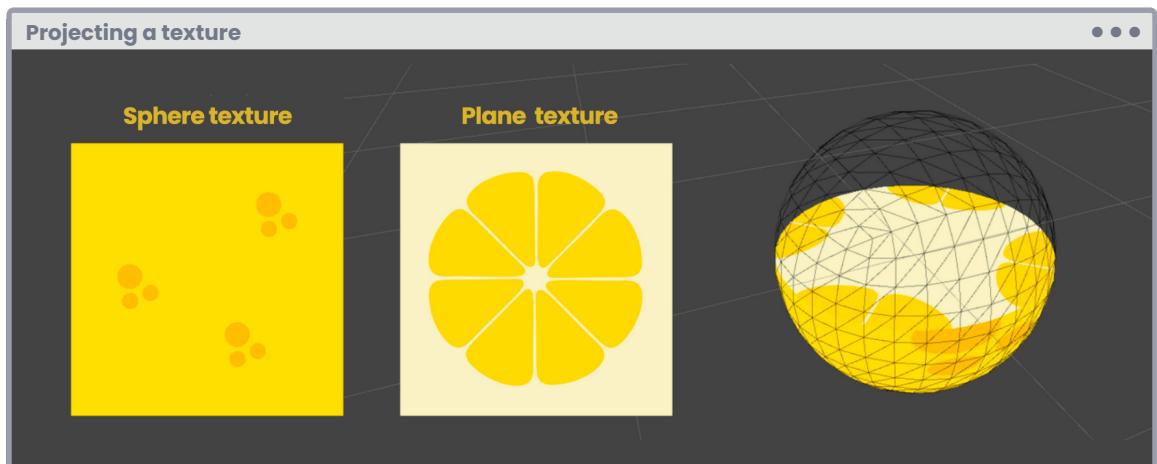
    if(t < MAX_DISTANCE)
    {
        float3 p = ray_origin + ray_direction * t;
        float2 uv_p = p.xz;

        planeCol = tex2D(_PlaneTex, uv_p);
    }

    if (i.hitPos > _Edge)
        discard;

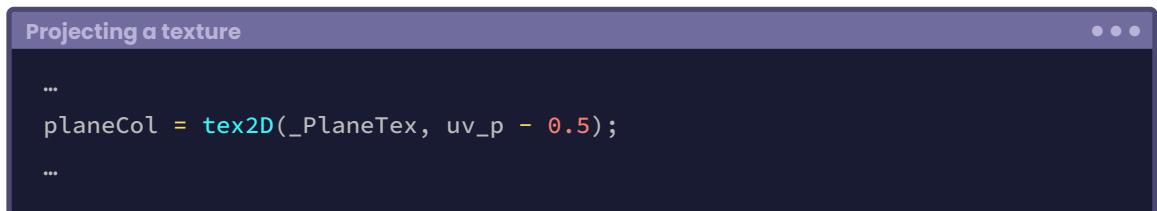
    return face ? col : planeCol;
}
```

In the previous exercise a new four-dimensional vector called **planeCol** has been declared. In this vector, you have stored the projection of the texture for the SDF plane, oriented on the Y_{Ax} axis. If everything went well, you would see textures on both the sphere and the plane in your scene.



(Fig. 11.0.2b)

Note that the starting point of the coordinate “uv_p” is at the center of the grid $0_X, 0_Y, 0_Z$ so it will be necessary to subtract 0.5f to center the plane’s projection.



If you modify the value of the `_Edge` property from the material Inspector, you can notice that the effect is working. However, it will be necessary to find an operation that lets you maintain the size of the projection on the plane when `_Edge` is equal to 0.0f, and decrease its size when it is in the range between 0.5f and -0.5f.

The following operation shows how to easily solve this exercise,

$$(-\text{abs}(x))^2 + (-\text{abs}(x) - 1)^2$$

Thus,

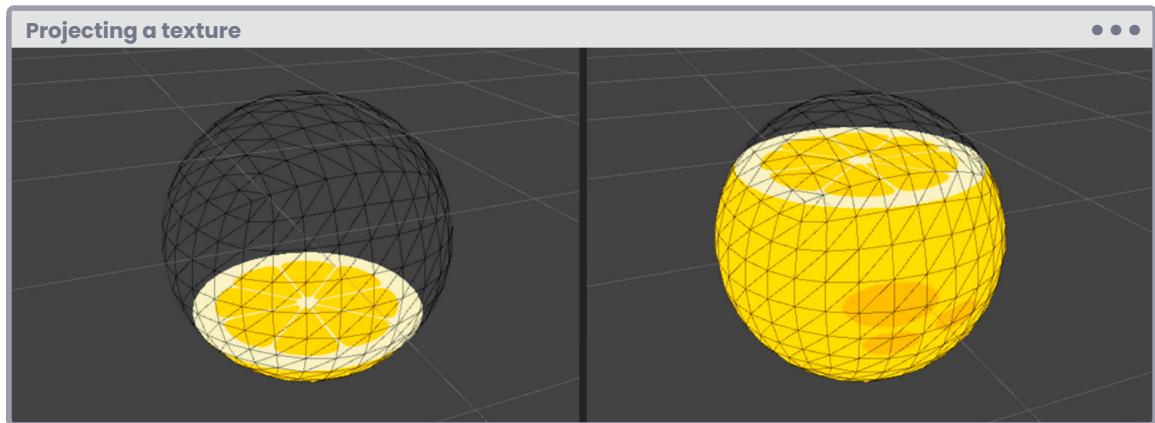
```
Projecting a texture ...
if(t < MAX_DISTANCE)
{
    float3 p = ray_origin + ray_direction * t;
    float2 uv_p = p.xz;

    float l = pow(-abs(_Edge), 2) + pow(-abs(_Edge) - 1, 2);

    planeCol = tex2D(_PlaneTex, (uv_p(1 - abs(pow(_Edge * l, 2)))) - 0.5);
}
...

```

You will see the plane's texture projection decreasing according to the sphere's volume when you modify the `_Edge` property's value from the Material Inspector.



(Fig. 11.0.2c. On the left, `_Edge` is equal to -0.246, while on the right, it equals 0.282)

You can stylize the effect by projecting a circle on the plane which follows the sphere's circumference. To do this, perform the following operation,

• • •

Projecting a texture

```

...
float4 planeCol = 0;
float4 circleCol = 0;

if(t < MAX_DISTANCE)
{
    float3 p = ray_origin + ray_direction * t;
    float2 uv_p = p.xz;

    float l = pow(-abs(_Edge), 2) + pow(-abs(_Edge) - 1, 2);

    // generate a circle following the UV plane coordinates
    float c = length(uv_p);

    // apply the same scheme to the circle's radius
    // this way, you can modify its size
    circleCol = (smoothstep(c - 0.01, c + 0.01, _CircleRad -
        abs(pow(_Edge * (1 * 0.5), 2))));

    planeCol = tex2D(_PlaneTex, (uv_p(1 - abs(pow(_Edge * l, 2)))) - 0.5);

    // delete the texture borders
    planeCol *= circleCol;

    // add the circle and apply color
    planeCol += (1 - circleCol) * _CircleCol;
}

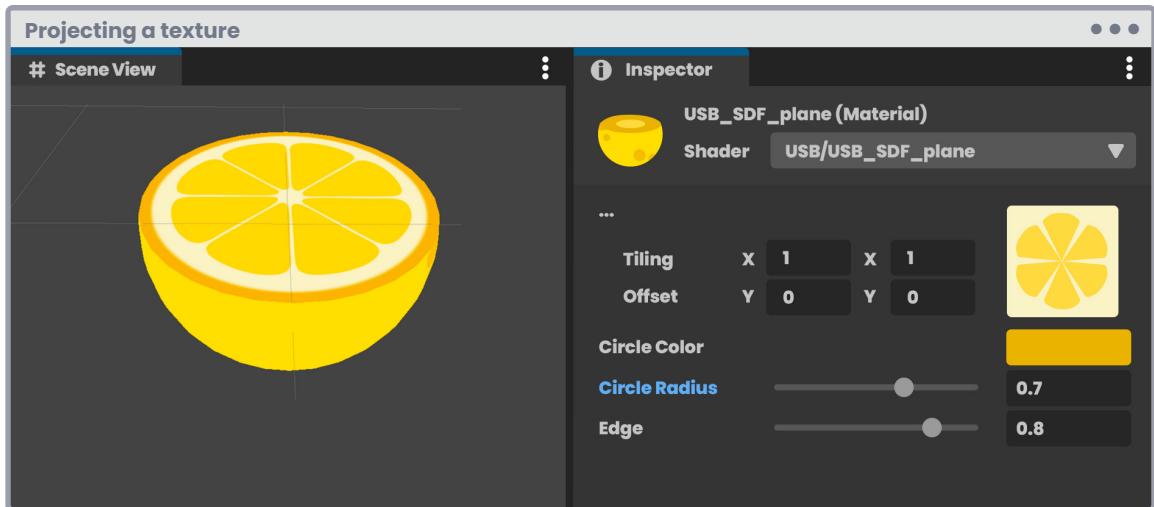
...

```

In the previous exercise, you generated a circle following the plane's UV coordinates projection and saved it in the "c" variable.

Then, the same mathematical scheme was used to increase or decrease the circle's radius. Finally, considering that the circle is only black and white, the edges of the texture projection have been removed. This was due to the sum of these values at the end of the operation.

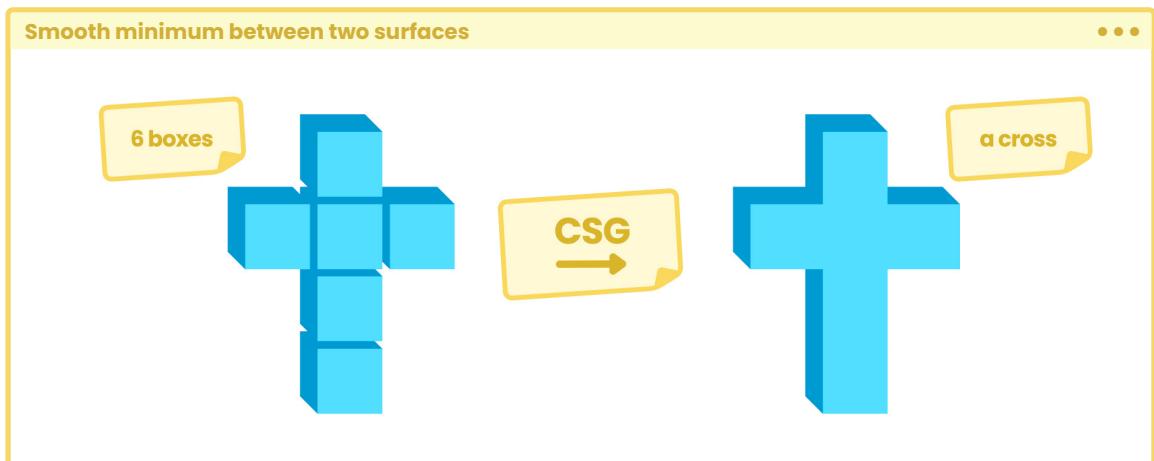
The circle radius can be modified dynamically through the `_CircleRad` property from the material Inspector.



(Fig. 11.0.2d)

11.0.3. Smooth minimum between two surfaces.

When working with Sphere Tracing, it is common to use operators (union, intersection, difference) for the generation of elaborate objects, e.g., if you want to create a cross in your shader, you could use six cubes, join them and simulate the shape, as shown in Figure 11.0.3a. This technique is called **constructive solid geometry** (CSG) and consists of creating complex bodies from primitive structures, i.e., cubes, cylinders, spheres, etc.



(Fig. 11.0.3a)

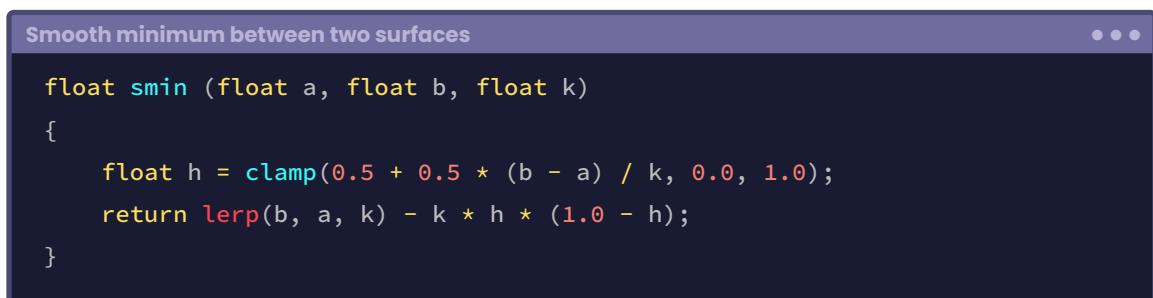
One of the most used operators in this technique corresponds to the **union between two surfaces**. To calculate it, you can simply use the “min” function detailed in section 4.1.9. However, given its nature, the result will generate sharp lines between both surfaces of implicit distance, maintaining the general shape being developed.



(Fig. 11.0.3b. Union between two circles)

If you want to mix both surfaces, you can use the solution of Íñigo Quilez, who created a function called **polynomial smooth minimum**, which uses linear interpolation to approximate the minimum between A_{RG} and $B_{RG'}$ where each one refers to a specific shape.

Its syntax is as follows:



To understand its implementation, create a new Unlit shader and call it **USB_function_SMIN**. Basically, this will make a smoothed union between two circles, which you will project onto a Quad in the scene.

Start by defining some properties. You will use these later in the effect's development.

```
Smooth minimum between two surfaces • • •
Shader "USB/USB_function_SMIN"
{
    Properties
    {
        _Position ("Circle Position", Range(0, 1)) = 0.5
        _Smooth ("Circle Smooth", Range(0.0, 0.1)) = 0.01
        _K ("K", Range(0.0, 0.5)) = 0.1
    }
    SubShader
    {
        Pass
        {
            ...
            float _Position;
            float _Smooth;
            float _K;
            ...
        }
    }
}
```

The **_Position** Property is directly related to changing the position of “one” of the circles. As mentioned above, you will create two circles. One of them will remain static, while the other will be moved from one side to the other to appreciate how the **smin(A_{RG}, B_{RG}, K_{RG})** function works.

_Smooth will be used to smooth the edges, while **_K** will be used to calculate the interpolation between the two circles.

Declare a function to generate a circle for the exercise.

```
Smooth minimum between two surfaces
float circle (float2 p, float r)
{
    float d = length(p) - r;
    return d;
}
```

As you can see, this function has the same structure seen previously. Using the **smin** function, you can calculate the minimum smoothing between two circles in the Fragment Shader Stage.

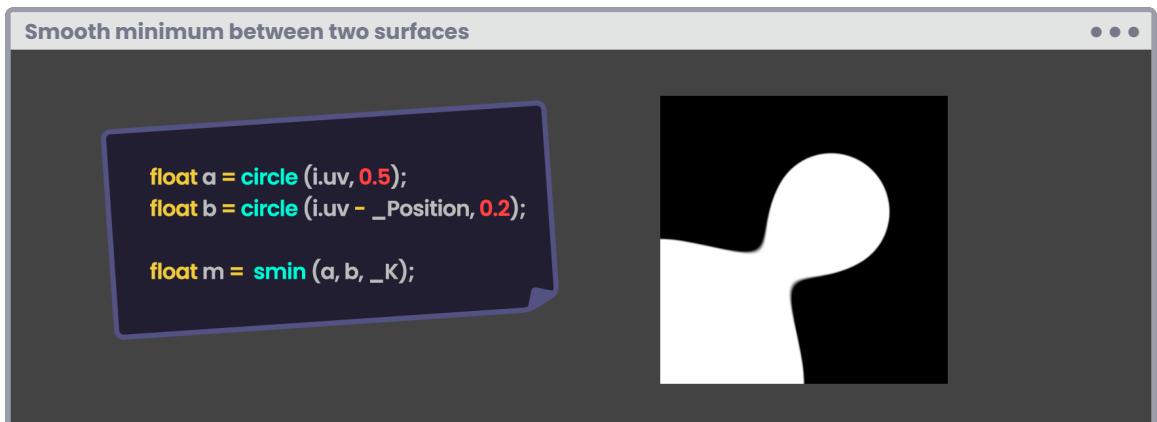
```
Smooth minimum between two surfaces
float circle (float2 p, float r) { ... }

float smin (float a, float b, float k) { ... }

fixed4 frag (v2d i) : SV_Target
{
    float a = circle(i.uv, 0.5);
    float b = circle(i.uv - _Position, 0.2);

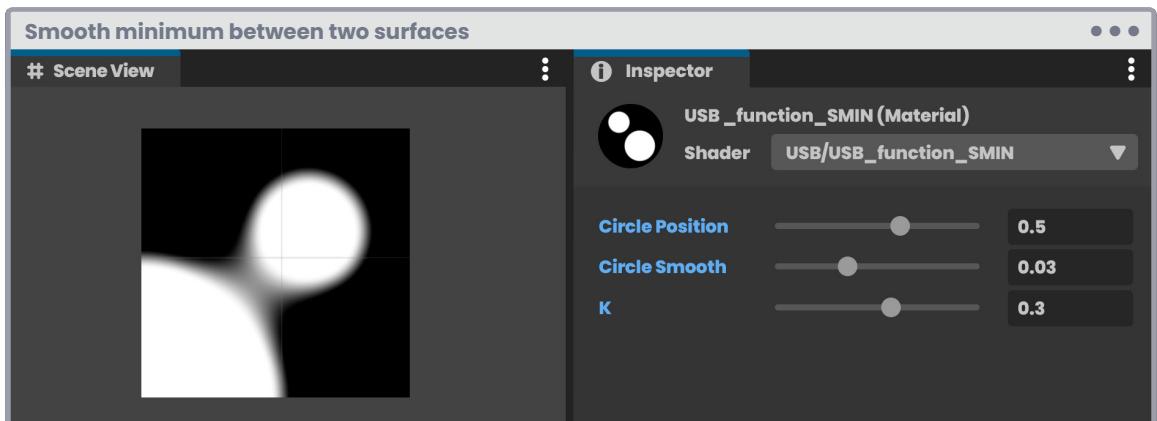
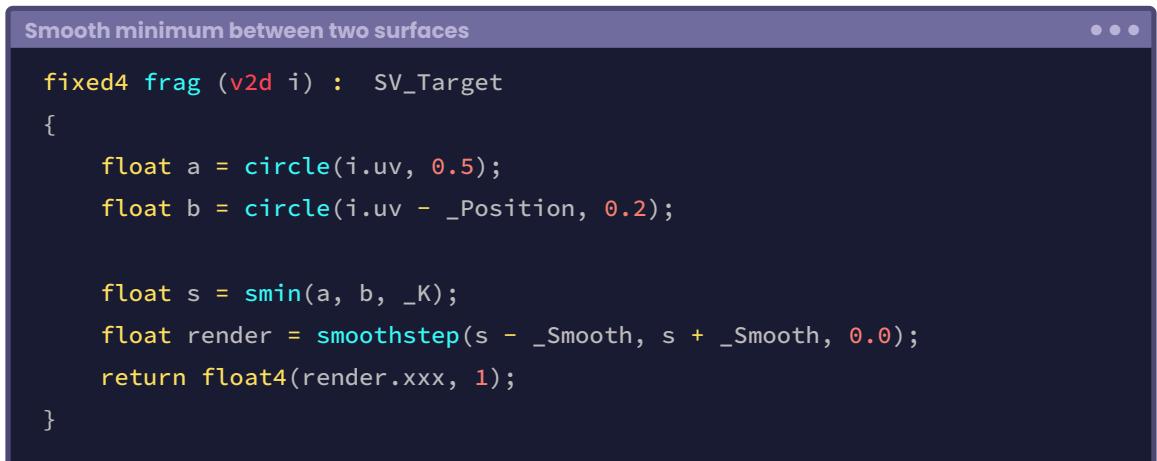
    float s = smin(a, b, _K);
    return float4(s.xxx, 1);
}
```

This created two scalar variables called “a” and “b.” These are used in the function “smin,” to return the smoothed union between the two shapes.



(Fig. 11.0.3c. Smooth union according to "k")

By incorporating the **smoothstep** function into the operation, you can generate smoothed edges for the overall composition.



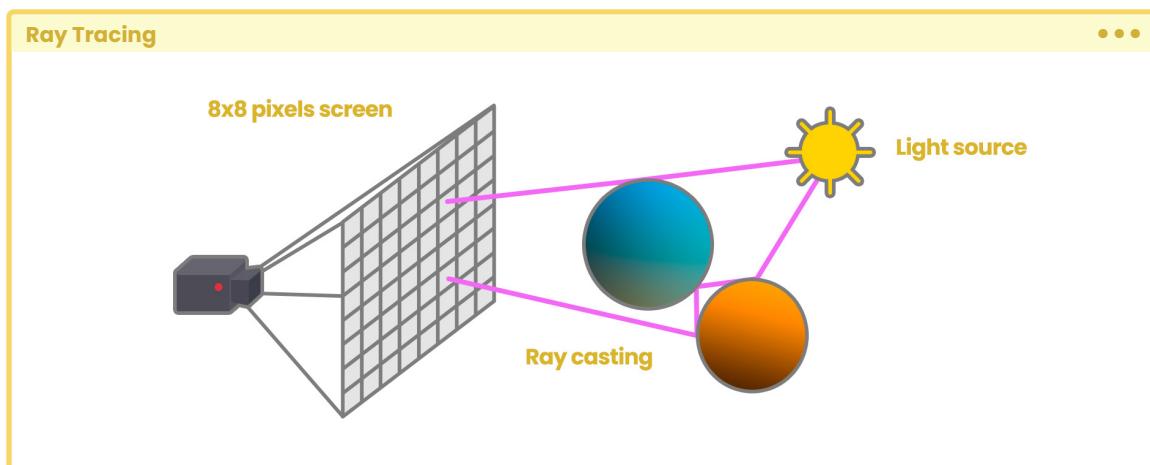
(Fig. 11.0.3d)

Ray Tracing.

At this point, you have reviewed an essential part of the necessary knowledge for shader development in Unity, both in Built-In and Universal RP. However, in this last section, you will focus on understanding High Definition RP and its Ray Tracing configuration.

To start, ask the following question, what is Ray Tracing? To understand the concept, you must pay attention to the behavior of lighting in the physical world.

Ray Tracing, like Sphere Tracing, uses the **Ray Casting** technique. However, in this particular case, it concentrates on obtaining lighting contributions from reflective or refractive objects in real-time, meaning that you can achieve a more realistic composition by sending rays through each pixel on the screen, which collide and bounce off each object in the scene.



(Fig. 12.0.0a)

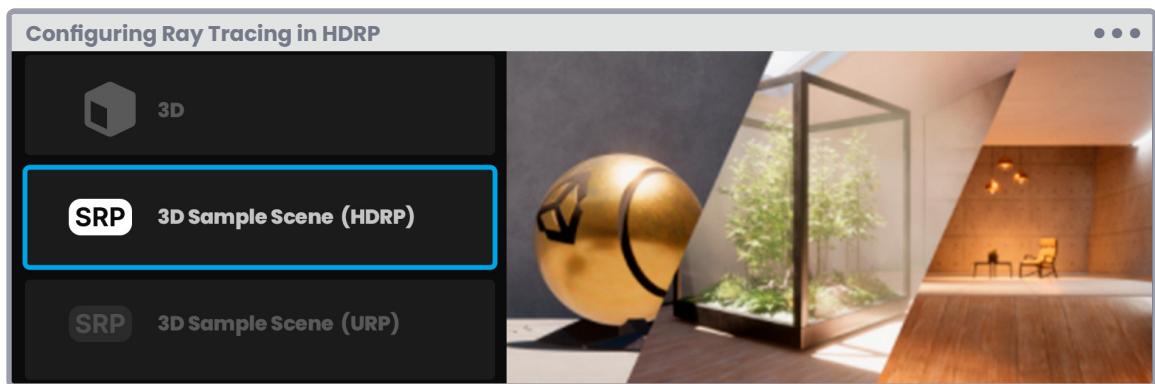
To accomplish a realistic composition you need to understand the following characteristics.

- Global illumination.
- Reflections.
- Refractions.
- Ambient occlusion.
- Shadows.

Before Ray Tracing, such properties were calculated in the software through **Lightmaps** which you can get from the Windows / Rendering / Lighting panel. However, given the nature of this technique, the elements in the scene had to remain static, not allowing the possibility of real-time calculations.

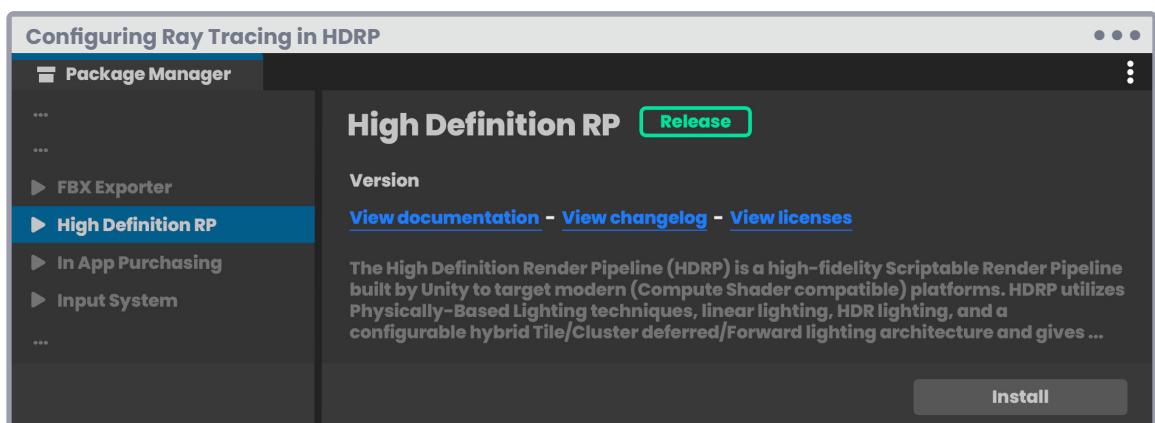
12.0.1. Configuring Ray Tracing in HDRP.

You will start this section using a default template from Unity Hub, version 3.0.0-beta.6. Such a template looks like this.



(Fig. 12.0.1a)

As mentioned at the beginning of this chapter, you need High Definition RP to perform these exercises. You can check its configuration by going to the Windows / Package Manager menu and ensuring that the **High Definition RP** package is installed in your project (as shown by Figure 12.0.1b).

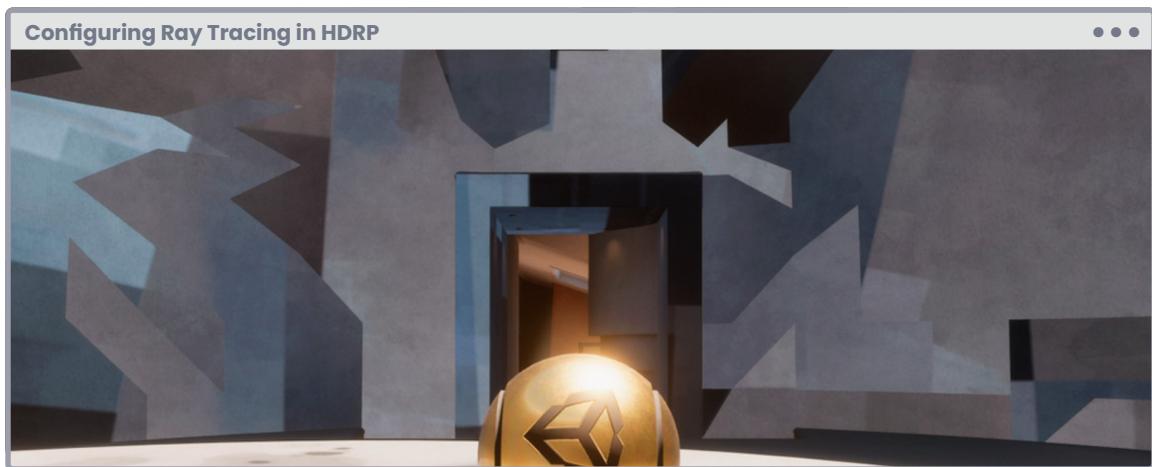


(Fig. 12.0.1b)

High Definition RP is characterized by its quality rendering and compatibility with high-end platforms, i.e., **PC**, **PlayStation 4**, or **Xbox One** (onwards).

It also supports **DirectX 11** and later versions and **Shader Model 5.0**, which introduces Compute Shaders for graphics acceleration.

When opening a scene, it is very common for some textures to look like those in Figure 12.0.1c. This is due to a configuration error in the generated **Lightmaps** when creating the project.



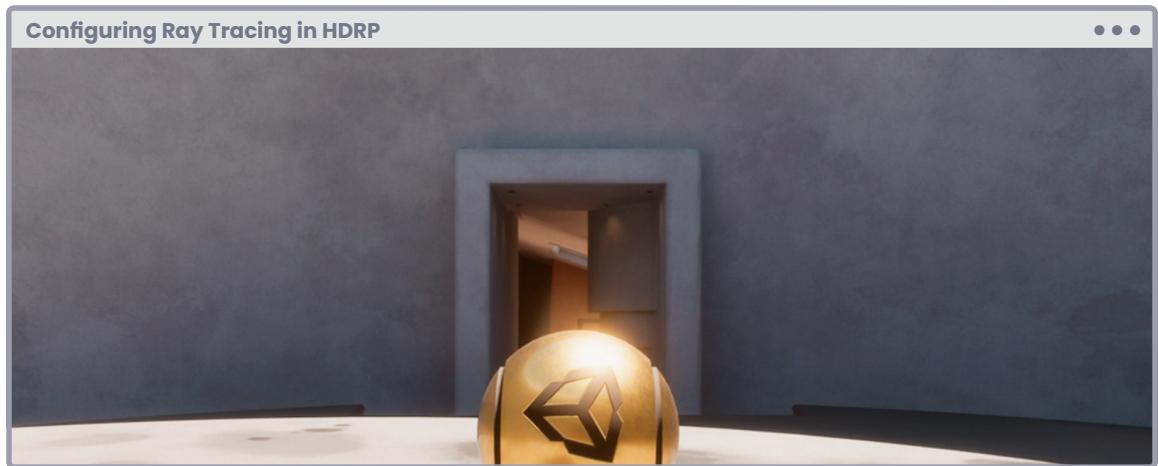
(Fig. 12.0.1c. The walls have errors in their lightmaps)

To solve this problem, you must pay attention to the configuration of the objects. If you select any object, e.g., **FR_SectionA_01_LODO** (wall), you will notice that it has been marked as "static" from the Unity Inspector.

So you must go to the menu Windows / Rendering / Lighting and perform the following operation:

- 1 Press the dropdown belonging to the Generate Lighting button and select Clear Baked Data. By doing this action, all the Lightmaps will be eliminated and you will see the default lighting.
- 2 Next, you must press the Generate Lighting button.

The process may take a few minutes depending on your computer's capacity. However, the textures and lighting will finally be displayed correctly.



(Fig. 12.0.1d. The lightmaps have been corrected)

Note that the global illumination and other properties, such as ambient occlusion, are being **burned** onto each texture. Therefore, if you change the position of an object, its lighting properties will keep their shape and will not be recalculated.

The only way to perform this process in real-time is by activating Ray Tracing. For this, you will have to consider several configurations, including DirectX 12 (DX12). The whole process can be summarized in three main steps;

- 1 Render Pipeline Asset.
- 2 Project Settings.
- 3 DirectX 12.

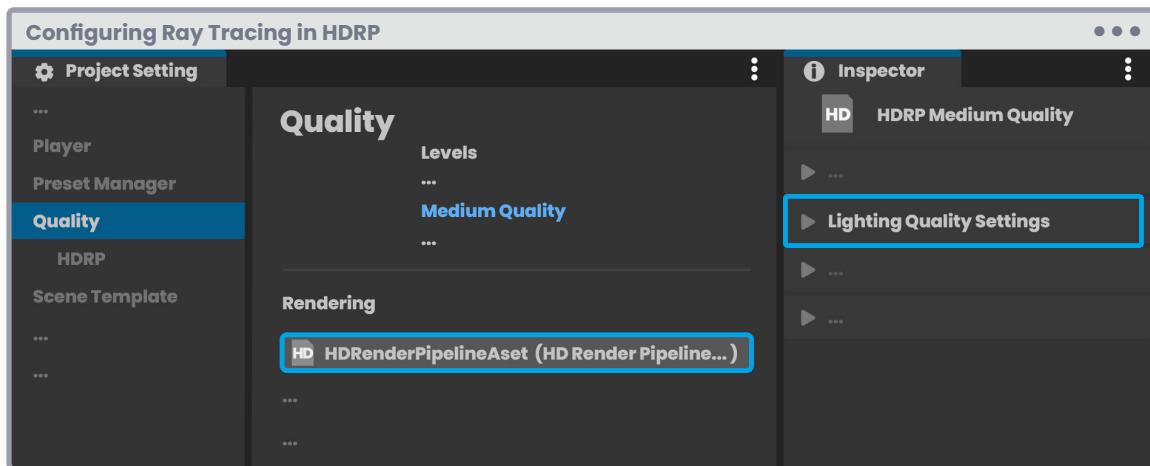
Start by going to the menu Windows / Panels / Project Settings, paying attention to the following categories:

- Quality.
- Graphics.
- HDRP Default Settings.

Note that Unity creates a different rendering configuration for each quality level in the project, e.g., our project has three default quality levels, which can be found in the **Quality** tab.

- High Quality.
- Medium Quality.
- Low Quality.

Each of these levels has a different **HD Render Pipeline Asset**, therefore, if you want to work with Ray Tracing, you will have to enable its options from the previously configured Asset; in this particular case, Medium Quality.



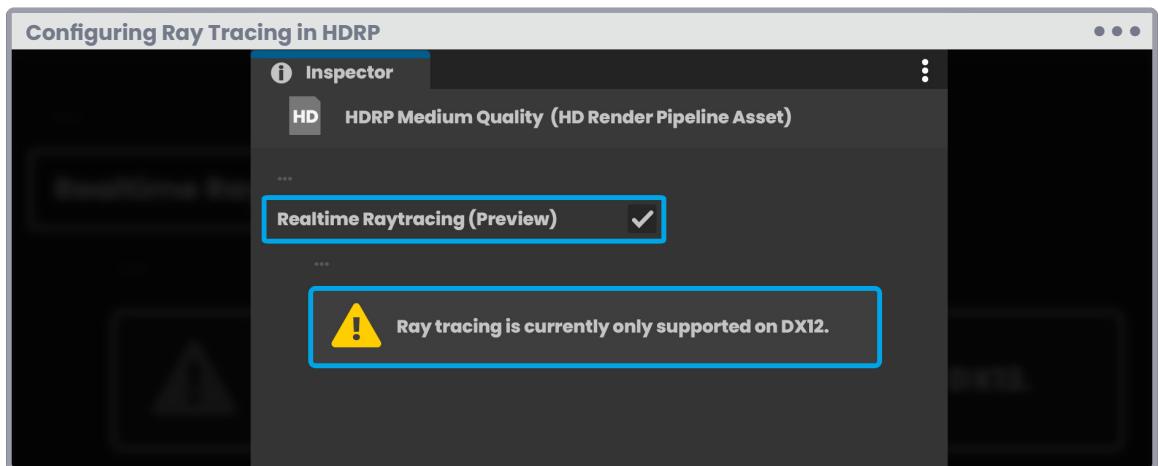
(Fig. 12.0.1e. Medium Quality configuration)

After selecting the HD Render Pipeline Asset from your project, you must pay attention to the **Rendering** and **Lighting** menus found in the Unity Inspector. Starting with the dropdown in the first one, you will have to activate the **Realtime Ray Tracing (Preview)** option, as shown in Figure 12.0.1f.

If your project is using a DirectX configuration other than version 12, the following message will appear:

Currently, Ray Tracing is only compatible with DX12.

Since Ray Tracing does not work in versions lower than DirectX 12, you will have to change your project configuration later. For now, you will continue to enable some options that let you perform global illumination calculations and other features.



(Fig. 12.0.1f)

Next, go to the **Lighting** menu and enable the following options:

- Screen Space Ambient Occlusion.
- Screen Space Global Illumination.
- Screen Space Reflection.
- Screen Space Shadows.

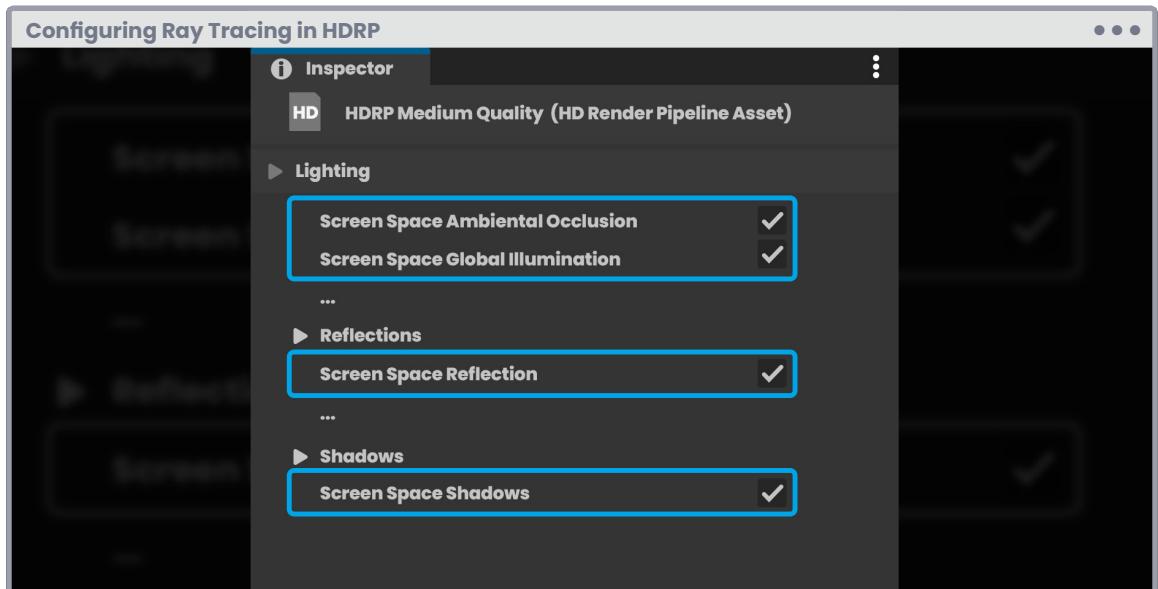
After the Clip-Space continues Screen-Space which refers to the transformation of coordinates between -1.0f to 1.0f to screen coordinates. Therefore, you can deduce that the higher the resolution, the bigger the Ray Casting calculation and therefore, the more power you will need in the GPU.

Screen Space Ambient Occlusion (SSAO) corresponds to an image effect capable of reproducing an approximation of ambient occlusion in real-time.

Screen Space Global Illumination (SSGI) lets you calculate the light bounce in real-time, generating a more accurate light representation of the composition in your scene.

Screen Space Reflection lets you calculate reflections in real-time.

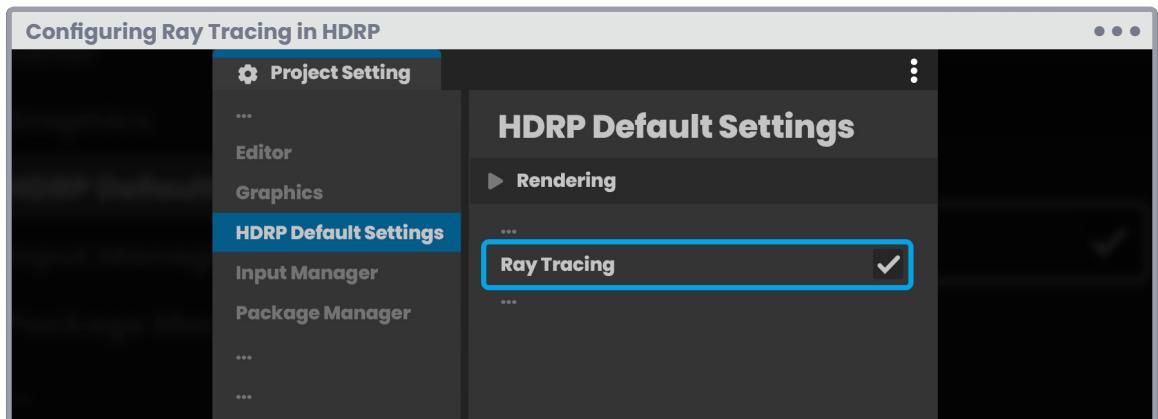
The same analogy is valid for **Screen Space Shadows**, which improves shadow projection.



(Fig. 12.0.1g. Different options have been enabled from the Lighting dropdown)

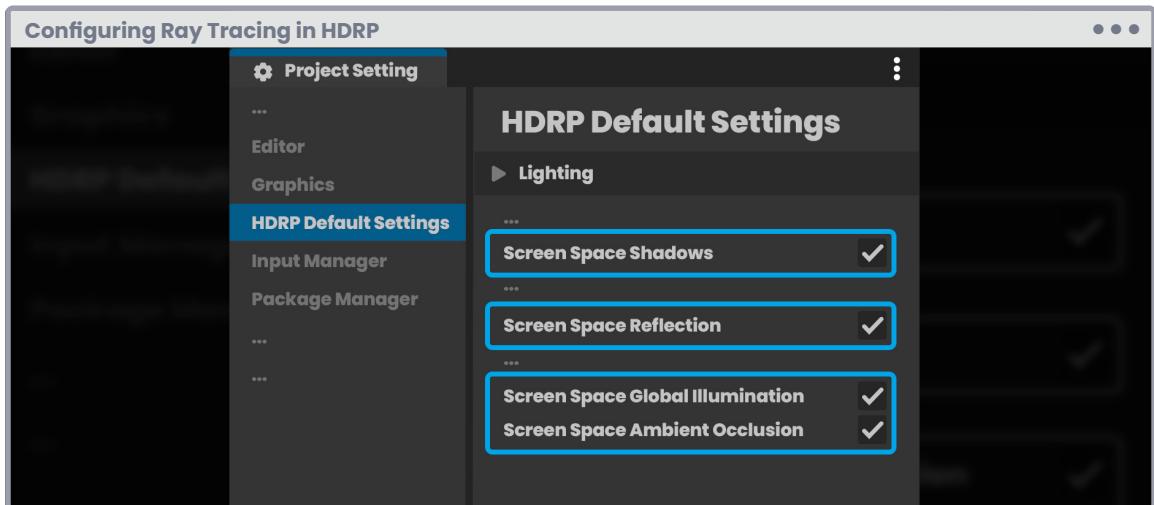
Now, Ray Tracing and its properties are enabled in the Render Pipeline Asset. You must configure your project so that these properties can perform their operations.

Again, go to Windows / Panels / Project Settings, **HDRP Default Settings** menu, and make sure it has the **Ray Tracing** option active from the **Rendering** dropdown in the **Frame Settings**.



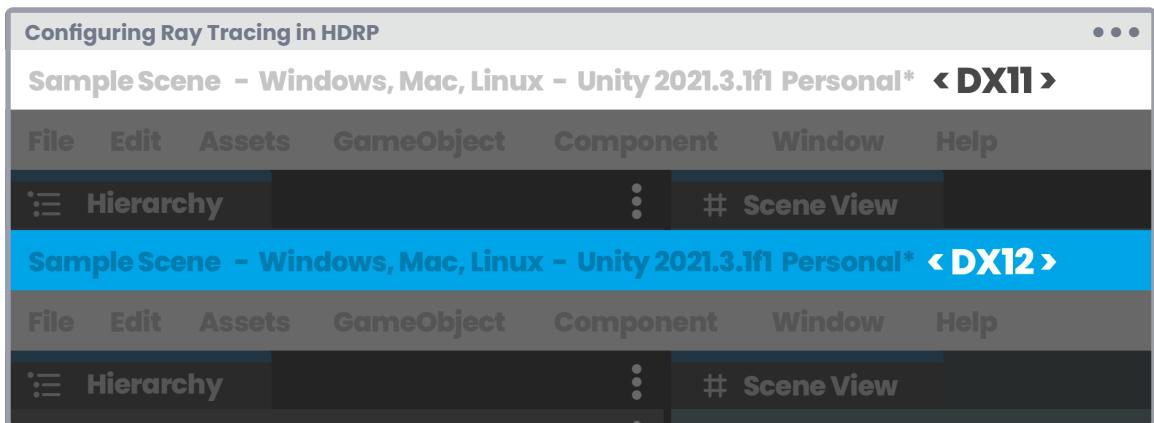
(Fig. 12.0.1h. Enabling Ray Tracing for scene's camera)

Then, from the **Lighting** menu, you must enable the same options that you activated in the Render Pipeline Asset: **Screen Space Shadows**, **Screen Space Reflection**, **Screen Space Global Illumination**, and **Screen Space Ambient Occlusion**.



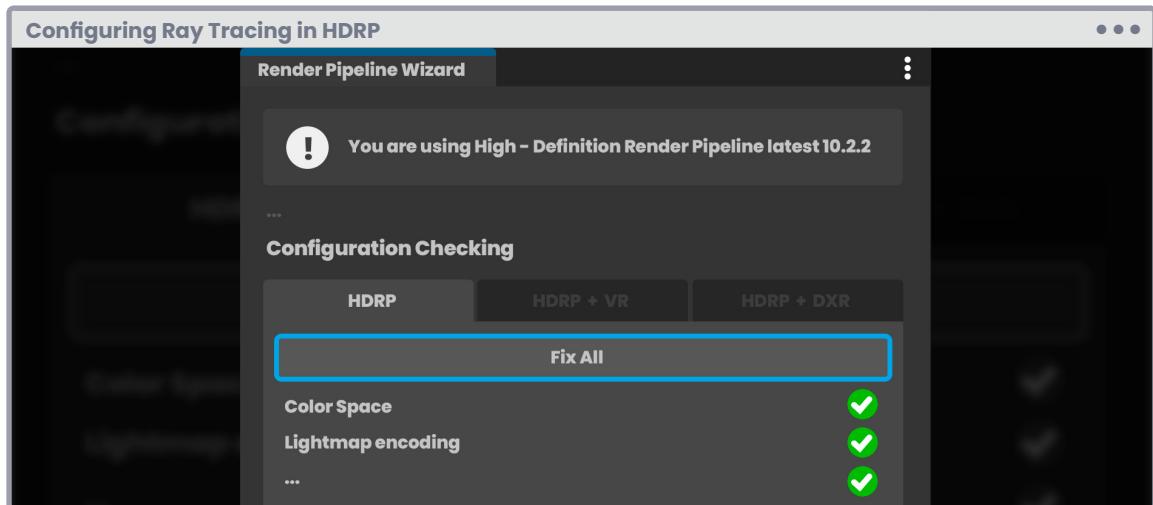
(Fig. 12.0.1i)

Now Ray Tracing is configured in your project. However, you must change your **DirectX** configuration since, as mentioned above, this technique only works in version 12, and your project, by default, has been configured in **DX11**. You can verify this in the Unity window on the toolbar, as shown in Figure 12.0.1j.



(Fig. 12.0.1j)

To do this, you must go to the Windows / Render Pipeline / HD Render Pipeline Wizard menu and press the **Fix All** button from the **DirectX Raytracing** tab (HDRP + DXT).



(Fig. 12.0.1k. DirectX Raytracing tab)

The software may ask to restart once the process has finished. When you reload the project, Unity will appear with the **<DX12>** tag at the top of the interface. If you go back to the Render Pipeline Wizard window, you will notice that all properties appear in green, which means that Ray Tracing is enabled for them.

12.0.2. Using Ray Tracing in your scene.

You will start by creating a new scene in your project. For this exercise, use a template included, called **Basic Outdoors (HDRP)**, which is characterized by having the following objects by default:

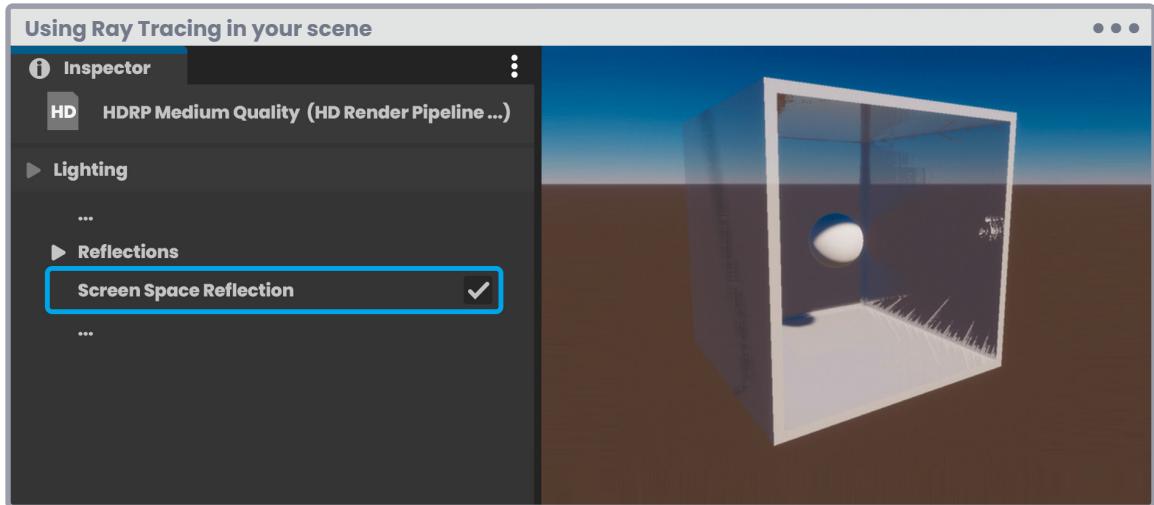
- A camera (Main Camera).
- A directional light (Sun).
- A sky (Sky and Fog Volume).

Note that you will use a room and a sphere to demonstrate this exercise. These **.fbx** objects can be found in the package attached to this book in their respective section.

Create two materials, one for each element. Call the material for the room **mat_room**, and the material for the sphere **mat_sphere**. Make sure to assign the **HDRP/Lit** shader to both materials.

Before you begin, assign each material to its respective object.

Previously, you enabled the **Screen Space Reflection** property from the **Render Pipeline Asset**; therefore, if you increase the value of the **Metallic** and **Smoothness** properties in any of the materials, you will be able to visualize reflections in real-time, as shown in Figure 12.0.2a. However, such reflections depend on the camera angle of view, consequently generating graphic artifacts.

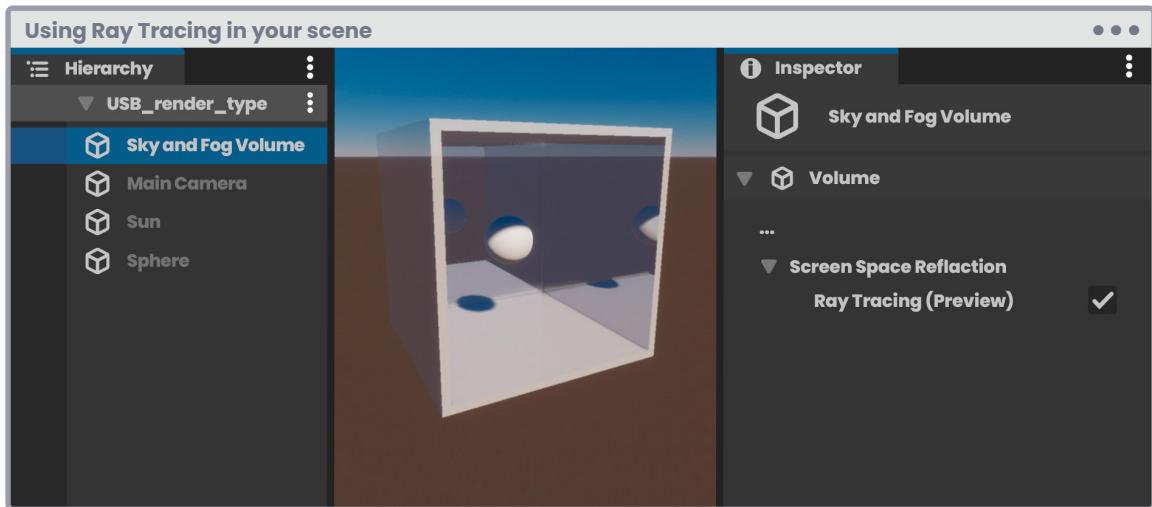


(Fig. 12.0.2a. mat_room material, Metallic equal to 0.5f, Smoothness equal to 1.0f)

If you want to activate reflections through **Ray Tracing**, you must perform the following steps:

- 1 Select the **Sky and Fog Volume** object.
- 2 Go to its **Volume** component.
- 3 Click on the **Add Override** button.
- 4 Select the menu **Lighting / Screen Space Reflection**.

For this exercise, activate all the **Override** properties. However, you will appreciate significant graphical changes once you enable **Ray Tracing (Preview)** because the reflections will be calculated in real-time.



(Fig. 12.0.2b)

By modifying the value of the **Bounce Count** parameter, you can increase or decrease the amount of bounce for the reflected rays.

You can use the same analogy to configure ambient occlusion and global illumination. To do this, press the **Add Override** button again and select the **Lighting / Screen Space Global Illumination** or **Ambient Occlusion** menu.

The process is similar in the case of shadows. To do this, go to the global light in your scene (Sun), select the **Shadows** menu, and enable **Screen Space Shadows**. You must make sure to activate the property **Ray Traced Shadows (Preview)** for it to take effect. Once this process is done, you can go to the **Shape** menu and modify the **angular diameter** according to your project needs.

A.

Analogy between a shader and a material. (116)
Adding transparency in Cg or HLSL. (118)
Adding URP compatibility. (126)
Analyzing Shader Graph. (284)
Ambient Color. (208)
Application stage. (22)
Abs function. (131)
Ambient reflection. (233)

B.

Built-in Render Pipeline. (25)

C.

Clip-Space Coordinates. (33)
CGPROGRAM / ENDCG. (96)
Cg / HLSL Pragmas. (103)
Cg / HLSL Include. (104)
Cg / HLSL Vertex Input & Output. (105)
Cg / HLSL Variables and Connection Vectors. (109)
Cg / HLSL Vertex Shader Stage. (110)
Cg / HLSL Fragment Shader Stage. (113)
Custom Functions. (298)
Compute Shader. (303)
Compute Shader, UV and texture. (321)
Compute Buffer. (325)
Constructive solid geometry. (353)
Ceil function. (136)
Clamp function. (141)
Cross product. (193)

D.

Deferred Shading. (29)
Debugging. (123)
DTX compression. (201)

Dot Product. (189)

Diffuse reflection. (211)

Data types. (98)

E.

Exp, Exp2 and Pow functions. (155)

F.

Fresnel effect. (243)
Forward rendering. (27)
Fragment Shader Stage. (113)
Floor function. (157)
Frac function. (168)

G.

Graph Inspector. (294)
Geometry processing phase. (22)

H.

HLSL function structure. (119)
High Definition Render Pipeline. (25)
HLSL. (36)

I.

Inputs and Outputs configuration. (182)
Intrinsic functions. (131)
Introduction to programming language. (36)
Image Effect Shader. (39)

L.

Length function. (165)
Lerp function. (173)
Lighting model. (208)

M.

- Min and Max function. (176)
- Matrices and coordinate systems. (30)
- Material Property Drawer. (52)
- MPD Toggle. (53)
- MPD KeywordEnum. (55)
- MPD Enum. (57)
- MPD PowerSlider and IntRange. (59)
- MPD Space and Header. (60)

N.

- Normal maps. (195)
- Normals. (18)

O.

- Our first shader using Cg or HSLS. (116)
- Our first shader using Shader Graph. (287)
- Object-Space. (31)

P.

- Properties of a polygonal object. (15)
- Properties for numbers and sliders. (48)
- Properties for colors and vectors. (49)
- Properties for textures. (49)
- Projecting a texture with Sphere Tracing. (347)

Q.

- Queue Tag . (64)

R.

- Ray Tracing configuration in HDRP. (359)
- Rasterization stage. (24)
- Render Pipeline. (20)
- Render Type Tag. (67)
- Ray Tracing Shader. (40)
- Render Pipeline Asset. (284)
- Ray Tracing. (358)

S.

- Shader structure. (41)
- Standard Surface shader structure. (251)
- Sin and Cos function. (146)
- Step and Smoothstep function. (161)
- Shadow implementation. (266)
- Starting with Shader Graph. (283)
- Sphere Tracing implementation. (338)
- Smooth minimum between two surfaces. (353)
- Shadow map optimization. (270)
- Specular reflection. (221)
- Shader. (35)
- Shader types. (38)
- Standard Surface Shader. (39)
- ShaderLab Shader. (45)
- ShaderLab Properties. (46)
- ShaderLab SubShader. (61)
- SubShader Tags. (63)
- SubShader Blending. (72)
- SubShader AlphaToMask. (77)
- SubShader ColorMask. (78)
- SubShader Culling and Depth Testing. (79)
- ShaderLab Cull. (82)
- ShaderLab ZWrite. (84)
- ShaderLab ZTest. (85)
- ShaderLab Stencil. (88)
- ShaderLab Pass. (95)
- ShaderLab Fallback. (114)
- Standard Surface input and output (253)
- Shadows. (255)
- Shadow Mapping. (255)
- Shadow Caster. (256)
- Shadow Map texture. (261)
- Shadow Mapping, Universal RP. (274)
- Shader Graph. (281)
- Sphere Tracing. (336)
- Signed Distance Functions. (338)
- Screen-Space. (363)

Screen Space Global Illumination. (363)
Screen Space Reflections. (363)
Screen Space Shadows. (363)
Screen Space Ambient Occlusion. (363)

T.

Tan function. (151)
TBN matrix. (206)
Tangents. (18)
Types of Render Pipeline. (25)
Time and animation. (177)

U.

UV coordinates. (19)
Unlit Shader. (39)
Universal Render Pipeline. (25)

V.

Vertex Color. (20)
Vertices. (17)
Vertex Shader Stage. (110)
vectors. (187)
View-Space. (33)

W.

World-Space. (32)
Which Render Pipeline should I use? (29)
What is a shader? (35)

Special thanks.

Taneli Nyssönen; David Jesús Ville Salazar; Carlos Aldair Roman Balbuena; Sergio Mireles Zamorano; Jhoseman Cesar Maraza Ytomacedo; Luis Fernando Salcido Infante; José Pizarro Rocco; Nicholas Hutchind; Luis Bazan Bravo; Elsie Ng; Lewis Hackett; Sarang Borude; Jonathan Sanchez; Иван Востриков; Alberto Pérez-Bermejo Galilea; Stephen Liu; Михаил Хаджинов; Zach Hilbert; Jake Manfre; Carlos Castro; Orlando Javier Orozco Guzmán; Furrholic; Alejandro Ruiz Ferrer; Éric Le Maître; Dimas Alcalde; Mika Juhani Makkonen; Giuseppe Graziano Softwareentwicklung; Xury Greer; Luca Palmili; Timothy Nedvyga; Coelet Swart; David Tamayo; Pedro Martins; Vaclav Vancura; Rene Melendez; Cesar Daniel Cerino Susano; Marcin Skupień; Ryan Bridge; Victor Celis Padrón; Josef Rogovsky; Jay Edry; Angel Daniel Vanches Segura; Mikail Miller; Insaneety Gaming; Ruilan Berg Pereira; PointNine; GameDevHQ Inc; David Muñoz López; Andrea Vollendorf; Djamschid Arefi; Srikanth Siddhu; Broken Glass LLC; Ben Vanhaelst; Anthony Davis; Asim Ullah; John Schulz; Rubén Luna de San Macario; Nathalie Barbosa Vásquez; Lukas Aue; Arnis Vaivars; Joel MacFadyen; Rebecca L Dilella; 志炜 马; Nhân Nguyễn; Eduardas Klenauskis; Stylianos Petrakis; 承晏 蔡; Ivan Paulo Guazzelli Machado; Daniel García Fernández; Victor Pan; Amit Netanel; Daniel Ponce; Thibaut Chergui; Tuanminh Vu; Craig Herndon; Jordan Huot-Roberge; Dragonhill LLC; Yuan Chiu; Sergey Ladychin; 庆 常; Patrick Pilmeyer; Ignacio María Muñoz Márquez; Tapani Heikkinen; Александр Максимович; Sebastian Cruz Dussan; Dustin Hoye; Matthias Rich; Cory Bujnowicz; Josue Ortigoza; Skydsgaard Translations; Marc Cacho; Geovane Pereira; Brainstorm Games LLC; César Iván Gallo Flores; Mark Mainardi; Hello Labs; Ricardo Costa Maginador; Yannick Vanhoutte; Peter Winston; Orlando Batista da Silva Lando; Brandon Brown.

MKS Soluciones; Jdui; Duca Stefan Stefan; Daniel Kavanaugh; Dominik Gygax; Erick Breto; Ángel Paredes Zambrano; Derek Westbrook; Gabriele Lange; Christopher Manasse; Timothy Fehr; Taylor Bazhaw; Nathaniel Shirey; AuKtagon; Matteo Lo Piccolo; Jonathan Smith; Rowan Goswell; De Blasio Corporation; Chaya Jagroep Jagroep; Nikhil Sinha; Dana Frenklach; Wilmer Lin; Roman Lembersky; Carlos Daniel Ahuactzin Parra; 江哉 漢; Tesseractaction Ltd; Nasrul Nasir; Alexandre Caila; Sime Tadic; Michael Dunkley; Blue Robot Creations; Juan Medina; Anne Postma; Adrian Higareda; Ignacio Gajardo; Edward Fernandez Silva; Juan Camilo Alcaraz Cartagena; Gabriela Mylonas; Juan Castillo; Sarah Sturm; Fernando Labarta; Alan Pereira; Tom Nemec; Angel Ordoñez Sanchez;

Sabina Kurgunayeva; Kayden Tang Tang; Kluge Strategic Inc; Joshua Byron; Afif Faris; Jonathan Morales-Rocha; Ángel David García Gómez; 용근 류; Shawn Sarwar; inMotion VR B.V; Joakim Lundkvist; Valerio Bellia; BetaJester Ltd; Bryan Pierce; Marco Tieghi; Dominique Maier; Carl Emil Hattestad; Eric Rico; Ossama Obeid; Liam Walsh; Quentin Chalivat; Carlos Melo; Michel Bartz; Lidia Arzenton; Abandon Ship LLC; Paola

 González Olea; Kasper Røgen; Jozef Bátrna; Luke Blaker; Noah Gude; Peter Britton;
 Timo Sikinger; Михаил Екимов; Kevin Toet; Patrick Geoghegan; Radosław Polasiak;
 Gerard Belenguer; Roberto Chiovenda; Justen Chong; Michael Stein; Brett Beers;
Abraham Armas Cordero; Jae-Hyeok Hong; Juan Sabater Sanjaume; Piotr Wardyński;
Kerry Leonard; Tsukada Takumi; Crisley Maihana; Ben Kahlert; Parag Ponkshe; Giyong
Park; Ryan Kann; Samuel Porter; Jacob Bind; Andrew Fitzpatrick.

Daniel Holmes; Anura Rajapaksa; Giorgi Tsaava; Pau Elias Soriano; Robert Hoole; SinisterUX;
Psypher Games LLP; Patricia Kelley; Алексей Черендацов; Esko Evtyukov; Fabricio Henrique; Zach
Jaquays; Jad Deeb; Michael Hein; Sourav Chatterjee; Djordje Ungar; House of How Games LLC;
Alexandre Rene; Daniel López; Thibaut Hunckler; Sunny Valley Studio; 裕貴 新井; Marcos Rebollo;
François Therasse; Jimmy Brown III; Rupert Morris; Zach Williamson; Gage Kilmer; Adam Lomax;
Ian Winter; Adrian Galeazzi; Tan Jen; Kayla Slifer; Simon Kandah; Alexei Tristan Menardo; Gorilla
Gonzales Studios; Fred Mastropasqua; Bradley White; Johnathan Pardue; Matan Poreh; Curtis
Weekusk-White; Wei Zhang; Davide Jones; Luis Reynaldo Alves; Diego Peña; Virtuos Vietnam;
Robert Krakower; Owen Duckett; Muhammad Azman; Metacious; Mike Curtis; Freelance; Tammy
Martin; Piotr Rudnicki; Adam Anh Doan Kim Caramés; Micael Brito de Jesus; 이 미주; Juan
Restrepo; Eran Mani; Mario Fatati; Lucas Keven Simoes dos Santos Sales; Jeremias
 Meister; Scott Allison; Christopher Goy; Le Canh; Leonardo Oropeza; Implosive
Games; Victor Mahecha; Deyanira Llanes; Omer Avci; Marcos Silva; Stanislav
Kirdey; Camila González; Cesar Mory Jorahua; Juan Diego Vázquez Moreno;
Unknown Worlds Entertainment Inc; Daniel Garcia; Michał Łęcznar; David Nieves Trujillo;
友太 本山; Carsten Flöth; Ángel Siendones Sillero; Mikko McMenamin; Richard Le; Erick Maciel;
Максим Хламов; Alessandro Salvati; Marc Jensen; Ariel Nuñez Bodden; Khang Nguyen; Pepijn van
der Linden; Alexander Horvat; Carlos Alberto Montiel Zavala; Christopher O'Shea; Useful Slug;
Chris Rosati; Sean Loughran; Suresh Venkataramana; Benjamin Radcliffe; José Manuel Vera
Menárguez.



 Charlie Darraud; 健斗 神田; Jonathan Rodriguez Ruiz; Jose Vicente Fernandez Pardo;
 Ishan Prakash; Ari Hoopes; Jarvis Hill; Ekipa2 d.o.o.; Marcin Iwanowski; Fredy Espinosa;
 Étienne Loignon; Datorien Anderson; Thomas Brown; Sebastián Procek; Shahar Butz;
Tomás Quiroz; Frayed Pixel Limited; Mayank Ghanshala; TwinRayj Studios; One Wheel
Studio; Luca Naselli; Pablo José de Andrés Martín; Максим Голубенко; Nikola Garabandić;
Ferdinando Spagnolo; Unreality3D; Péter Nagyidai; Winfried Schwan; VRFX Realtime Studio
GmbH; Dana Würzburg; 陳芸軒; Juan Manuel Ramon Vigo; Eric White; Yu Gao; David Bermudez
Lopez; Berlinger Gilles; Derek Yiok Teik Lau; Robin Hinderiks; Dennis Koch; Grant Nelson; Marcel
Marti; Daniel Corujeira Ortega; Charlotte Delannoy; Casey Mooney; Yorai Omer; Eric Manahan;
Anastasiya Tolkachyova; Wei Tang; NieddaWorks; Colin Mongabure; בוקי יירדן; Shuxing Li; Shiri

Blumenthal; Kevin Choo Fun Young; Slufter; Nikolai McNeely; Kevin Roberto Gomez Peralta; Diego Esedin; Andrew Bowen; Matthew Spencer; Susan Cho; Bright Future GmbH; Sander De Pauw; Alen Brkicic; Guillaume Cauvet; Michael Aviles; Fabio Schegg; Michael Center; Jean-Baptiste Sarrazin; Rolandas Cinevskis; 学贤 张; Brian Smith; Nathan Buckley; Julien Rochefort Delsalle; Coldharbour Media; Ali Taher; Wade Lewis; Wei Zeng; Vesselin Handjiev; Lukasz Lampka; Groove Jones; Stephen Eisenmann; Kaochoy Saetern; Christopher Kline; Darya Luchaninova; Akash Castelino; Daniel Radu; Théo Monnom; Javier Molla Garcia; Aykut Yildirim; Bernhard Esperester; Nathan Sheppard; Remi Kroll; Катерина Колесникова; Alexander Sachuk; Cesar Ramirez Cervantes; Christian Miller; James Rossiter; Moritz Großfurtner; Ludwig Broman.

Christian Hertwig; Ben Boniface; Jans Margevics; William Barteck; Nils Hammerich; Oskar Kogut; Darko Nikolic; 043 Imagine; Jose Torres; Digitalpro; Юрий Ануфриев; Ludibyte Games; Hamish Dickson; Timothy Neville; William Beard; Евгений Шишkin; 広希 奈良; Alessio Landi; Niklas Weber; James Macgill; Jessamyn Dahmen; Justin Herrick; Raymond Micheau; Troy Patterson; Sandor Fejer; Anthony Massingham; Ryousuke Nakai; Kazumi Mitarai; Samuel Furr jr; Katsuya Taniguchi; Arthur Del; Matthew Burgess; Anil Ayaz; Neomorph Studio SRL; David Moscoso; KoriinArt; Rodrigo Abreu; Antonio Ripa; Victor Lalo; Gold Gnome; Vitai Tamás Egyéni Vállalkozó; Low Zhe Ming Walter; Jordan Dubreuil; Jun Kyung Kim; Adrian Orcik; Edwin Morizet; Wojciech Sajdak; Roberto Bernous; Wilson Ortiz Morales; Mario Tudon; Marc Segura Molina; Malik Abu Aune; Miguel Grunfeldt; Pitchayah Chiothian; Iván Godoy Rojas; Juan Mauricio Ochoa Castillo; Alina Sommer; Omar El Halabi; 재영 최; Ryan Salam; Ben Guiden; Kevin Hagen; Ninquiet; Kevin Willis; Carlos Gerardo Enríquez Valerio; Valerie Nunez; Peter Cowen; Chang Hoon Oh; Pavel Shutau; Mattias Gyllerup; T K; Yuri Kovtunovych; Jason Peterson; Nicole Wade; Starloop SL; Raul Torres Gonzalez; Adrián Rangel Suárez; Zaibatsu Interactive Inc; Arnaud Jopart; Valdeir Antonio Nascimento Santos; Juan Carlos Romero; Joss Gitlin; Christian Santoni; Du Yoon; Adrian Koretski; Juan Antonio López Rodríguez; Dongyeon Kim; Flashosophy; Colter Wehmeier; Wendeline Aerts; Joan Sierra Patiño; Michael Ha; Test Out Web Design; Eddy May; Josh Lowes; Do Minh Triet; Matthew Anderson; Durmus Ali Collu; Emma Barnes; Christer Bjoerk.

Angie Rojas Mendez; Adriel Almirol; Jacob Fletcher; Krzysztof Bziuk; Simon Borg; Diego Paniagua Morales; Alan Berfield; Adam Warkentin; Jonathon Stone; Chèze Chèze; Senem Gokce Ogultekin; Louise Crouch; Koi koi; Victor Carvalho Estrella; Bruno Fernando Pita Sassioto Silveira de Figueiredo de Figueiredo; Oleksandr Kokoshyn; Danny Darwiche; The Life Forge; Natus; Patrick Schnorbus; Raúl Vera Ortega; Daniel Izacar Memije Fábrego; Kevin Babin; Diego Millan; Jean-Bernard Géron; David Alonso Alapont; Andrea Osorio; Manuel Obertlik; Juan Alvarez Mesa; Brad Johnson; Ricardo Díaz; Alejandro Azpitarte; Wayne Moodie; Brock Williams; Ernesto Salvador Solares Guerrero; Abdullah

Al Zeer; Mathis Schmidtke; Juan Carlos Horta; Ilham Effendi; Aaron Prideaux; Syed Salahuddin; Damian Griffin; David Roume; Mal Duffin; Armonte Williams; Daiya Shinobu; Michael Joyce; Danik Tomyn; Syama Mishra; Marine Le Bornge; Daniel Ilett; Omar Espinosa; Pollywog Games; Paulo Sergio Fernandes; Ahmed Gado; Gilberto Alexandre dos Santos; Ato Ishimoto; Alexander Grinkevich; Victor Cavagnac; Andoni Torres; Kailun Cai; Wilko Willame; Christopher Johnson; Jefferson Perez; 1998; Iván Gallego Muñoz; Sergio Labbe Grandon; Arnold Wittenberg; Dominic Butler; Steve Barr; Misumi Yuki; Dawid Ochryniak; Yingdi Fu; Doge Corp; Sofía Lozano Valdés; 健斗 中島; Thomas Tassi Joergensen; Christopher Munguia; Lim Siang; Rolf Vidstd; Desarius Games di Dario Visaggio; David Vivas Estevao; Ciria Quispe; Nathan Clark; Marco Di Timoteo; Jan Neuber; Angel Aristides Zuniga; James Meade; Iniciativas Digitales; Marvin Gewiss; Yulia Trukhan; Paulo Lara Carreño; Daniel Sierra; Andrés Cortés Dávalos; Raul Bustamante Morales; Aldo Eduardo Fuentes Millan; Raimundo Gallino; Martin Perez Villabrille; José Francisco Torreblanca Nava; Julio Ortiz Acosta.

Jesus Popocatl Lara; Juan Manuel Hernandez Hernandez; Pookzz3d Ninja; Данила Поляков; Juan Antonio Pascual Albarranch; Tahiche Maria Mena; Silvia Acevedo; Orlando Almario; Alfonso Varela Giménez; Ekaitz Segurola Elosua; Andrea Polanco; Murughavell Allagarsamy; Marcos Antonio Vilca; Javier Maldonado Díaz; Massimo Di Cesare; Paul Pinto Camacho; Hector Ortiz Muniz; Roberto Delgado Sánchez; Ahmed Elwardy; Omar Akkari; Guillermo Meléndez Morales; Daniel Garcia Daniel Garcia; Jorge Vecino Labajo; Juan Francisco Matheu García; Bastian Oñate; Thiago Carneiro; Eduardo Noé; John Estrada; Luis Garvi Zarco; Adalberto Perdomo Abreu; Miguel Cano Santana; Casey Hallis; Ignacio Alcaino; Juan Díaz de Jesús; Miguel Muñoz Ortega Terrazas; Vita Skruibyte; Daniel Sørensen Sørensen; Sebastian Manriquez; Patil Aslanian; Ersagun Kuruca; Ismael Salvado Fernandez; 永恒 朱; Oscar Yair Núñez Hernández; Nguyễn Đại; Fraser Hutchison; Mario Pinto Hermosell; Pedro Afonso de Aviz; Lee Wayne; 有成 胡; Matthew Berenty; Yimeng Chen; A Brunton; George Katsaros; Mark Kieran; Ryan Collins; Igor Dantsev; Ayoub Khourbach; Damian Osikovsky; Ross Furmidge; David Lozano Sánchez; Nguyen Cuong; Kyle Harrison; Trixtaro - Desarrollo de Software; Jordan Totten; Paul Moore; 현근 곽; Peter Law; Francisco Javier Lucas Martínez; Павел Плеханов; Andrea Zilio; Juan Mozo Osorio; Quentin Julien; Alexandre Calabuig Langa; Keith Mottram; Reinier Goijvaerts; Dilpreet Singh Natt; Patricia Sipes; Renan Dresch Martins; Sungjin Bae; Stephen Selwood; Unije Apps; Rosario Ranieri; Scott McCulloch; Jordan Fye; Wenpu Ng; Alexander Grunert; Jason Tu; Nikita Kotter; Nicole Cox; 一樹 西脇; Angéline Guignard; Bartosz Bielecki; Baptiste Valle; 順一 馬場; Lleïr Valerià Diego Gutierrez; Damian Turnbull; Ian Brenneman; Nicolas Acevedo Suzarte; 亮人 西尾; Anastasija Grigorjeva.

Manoj Jeyaram; 尼玛 胡; Jelle Husson; Karsten Westra; Judie Thai; Emmanuel Castro Flores; Eduardo Roa M; Daniel Fairgrieve; Steven Hurst; Michal Pawlowski; Robert Southgate; Jeffrey Scheidelaar; Daniel Turner; Денис Смольников; Shayam Thomas; Raúl Pla Ruiz; John Bulseco; Hongbum Kim; Roberto Margotta; Eran Eshkol; Maciej Miarecki; Alexandre Abreu; Kelvin Put; Alexander Mutuc; Valdream; Yifat Shaik; Ramon Ausio Mateu; Jose Ignacio Ferrer Vera; Burak Soylu; Thierry Berger; Alan Sorio; Jamie Niman; Adrian Impedovo; Jaeyoung Choi; Chara Sottou; Alessia Marra; Arno Poppe; Jose Aristizabal; Juan Lucas Arruda Maciel; Shounak Mandal; Daniel Gomez Atienza; Eddy Margueron; José Javier Serrano Solís; Mitchell Theriault; Alexander Isom; YuChen Ou; Venkatesh Muskam; Enmar Ortega; Katie McCarthy; Juan Martinez Lopez; Samuel Moreno Luque; Cody Scott; Elliot Padfield; Wilson Rivera; Ian Butterfield; Juan Casal; Jason Brock; Santiago Viso Cervera; Camilo Angel Grimaldo Arreguin; Samuel Swift-Glasman; John Rantala; Adrian Ślusarek; Phuoc Vu; Faisal; Jamie Hyland; José Ignacio Alonso Kuri; Eric Kalpin; Vasile Sebastian Mihali; Jonatas Santos; Daniel De Oliveira; Damian Smyth; Jouni Sarvanko; Giuseppe Modarelli; Juliusz Wojnicz; Ellitsa Ilieva; Angel German Pavon Cabrera; Guilherme Schüler; Simon Säaf malm; Logan Lewis; Mikel Gonzalez Alabau; Bethany Dixon; Daniel Fischer; Lewis Nicholson; Armando Soto; Lloyd Vincent; Michael Jonathan Magaña Dominguez; 修逸 谷; Julio Alejandro Quiroz Astorga; Michael Donnelly; Brendan Polley; Steafan Collins; Harry Emmanuel; Ryan Trowbridge; Alex Pritchard; Zhouming Tang; Manuel Galindez; Here's Joe, LLC; Jeremy Bonnaud; Christopher Medina; Daum Park.

Chanael Godefroid; Tushar Purang; Donnovan Feuillastre; Christopher Coyle; Aakash Shah; Aboud Malki Malki; Matthew Kinahan; Александр Джанашвили; 崇億 張; Adeline Kinsama; Chi Hung Wu; Clara Rodríguez Palacios; Cristian Peñas Laplaceta; Neel Rajeshkumar Mevada; Brian Heinrich; Glaswyll Entertainment LLC; Anil Mohan; Zdravko Nikolovski; Arda Hamamcioglu; Nick Ward; Designer Hacks; เจนสีทธิ์ วงศ์วรรธน์; Tu Ho Le Thanh; Erik Niese-Petersen; Emil Bachvarov; Christian Van Houten; Sweet Cheeks; Michał Szynal; Michael Hayter; Genevieve St-Michel; Fouad AlSabeh; Alejandro Ruiz; Jacob Rouse; Jeremedia; Under Galaxie; Elodie Marine Solange Saito; Shawn Beck; Ming Hau Loh; Marvin Saignat; Albert Marcelus; Kayra Kupcu; Long Nguyễn; Madeleine Kay; Zbigniew Zelga; Morgane Paulmier; Grant Blair; 승범 0; Ciro Continisio; Diego Gutiérrez Rondán; Antonella Vannucci; Jose Argenis Jimenez

Gonzalez; Jason Holland Holland; Alejandro Endo; 岩田 和己; Jordi Porras Estupiñá; Giulio Piana; Mishel Delgado; Kaliba Games & Technologies Inc; Ryan Miller; 国云 刘; Aesthezel; Sharatbabu Achary; Gigantic Teknoloji A.S; Andreas Tsimpanogiannis; Diwakar Singh; Steven Burgess; A Kim Arrate; Austin Eathorne; Caleb Greer; Antonio Rafael Ruano Rodriguez; Florian Geslin; Benjamin Thomas Harbakk; Xu Guo; Jonathan Kelly; Ronnie Denney; Aaron Stewart; Steven Cardenas; Александр Беспалов; Thomas Dik; Aristoteles Dominguez Gonzalez; Florent Lagrede; Kathy Huynh; Bryan Link; Alex Murphy; Johannes Peter; Yulia Yudintseva; Simon Gemmel; Christoph Weinreich; Cybernate PTY LTD;

Leonardo Marques; Jonathan Ludwig; Shin Tsukada; Eric Farmer; Виктор Григорьев;
Kushal Timsina; Jalexa Hernandez Baena; Trung Nguyen Tran; Hibbert IT Solutions
Ltd; Konrad Jastrzebski; Alexandre Bianchi.

Caio Marchi Gomes do Amaral; Matt Mccormack; Yuichi Matsuoka; Andriy
Matviychuk; Lieven Van den Audenaeren; Ediber Reyes; ຈິກາ້ຫຣ ພູມມາກ; Renáta Ivony;
Dmytro Derybas; Joshua Villarreal; Clement Brard; Sofia Caponnetto; Maksim
Ambrazhei; Seongho Lee; Jeff Minnear; Heath Sargent; Diego Sánchez Ramírez;
Christopher Page; Jean de Oliveira; Adriano Romano; Gerson Cardenas; Catgames;
Reder Joubert; Trevor Ings; Jonathan Jenkins; Hugo Bombail; Ivan Sazanov; Michael Daubert;
Jérôme Cremoux; Yefri Avella Molano; Luis Angel Meza Salinas; Mauricio Vargas; Olie Swinton;
Ayoub Ourahma; Yaroslau Sidarkevich; Kristófer Knutsen; Garrett McMichael; Ayoub Ourahma;
James Smith; Marc Alloza Ayxendri; Miquel Postigo Llabres; Илья Загайнов; Cameron Millar; Hamza
Rangoonia; Zach Holt; Holly Wolf Newlands; 信裕 石黒; Dai Zhen; Matthew Insley; Henrique Sousa;
Yusue Chen; Antti Hietaniemi; Anton Sasinovich; Mischa Wasmuth; Wang Yong-Gang; Carl
Hughes; Liyuan Qi; Jorge Mula Ferrer; Joshua Prosser; Aaron Scott; Pierre Tane; Ramlil Limarest
Roosileht; Patrick Yeung; Juan Manuel Altamirano Argudo; Hugo Alvarado; Jonas Sulzer; Nicole
Folliott; Donghwan Kim; Raees Rahim; Cole Andress; Orlando Si-Kae Fang; Pablo Guinot Gironda;
Matthew Tan; Peetsj; Nikita Pohutsa; Erik Minarini; Alexander Eisenhart; Sergi Herrero Collada;
Guido Meo; Showy Lee; Fabian Schweizer; Shane Nilsson; Hernan; Hugues Vincéy;
Abdul Brown; Faris Alshehri; Ji Hyun Ahn; Daniel Rondán; Jeff Registre; Jose Javier
Delgado Cuder; Rocio Campo; Arthur Gros Coumantaros Aulicino; Michael Hoen;
Michael Trainor; David Hooper; Takefumi I Kaido; Александр Кравцов; Александр Басюк;
Carlos Ivan Cordoba Quintana; Shadow Storm Limited.



"Jettelly wishes you success in your professional career."