# THE UNITY
# SHADERS
# BIBLE

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



By Jettelly

Fabrizio Espíndola
Pablo Yeber

# The Unity Shaders Bible.

(0.1.2. version)

**Fabrizio Espíndola.**

Game Developer & Technical Artist.

# Credits.

**Author.**

Fabrizio Espíndola.

**Graphic Design.**

Pablo Yeber.

**Technical Revision.**

Daniel Santalla.

**Translation.**

Martin Clarke
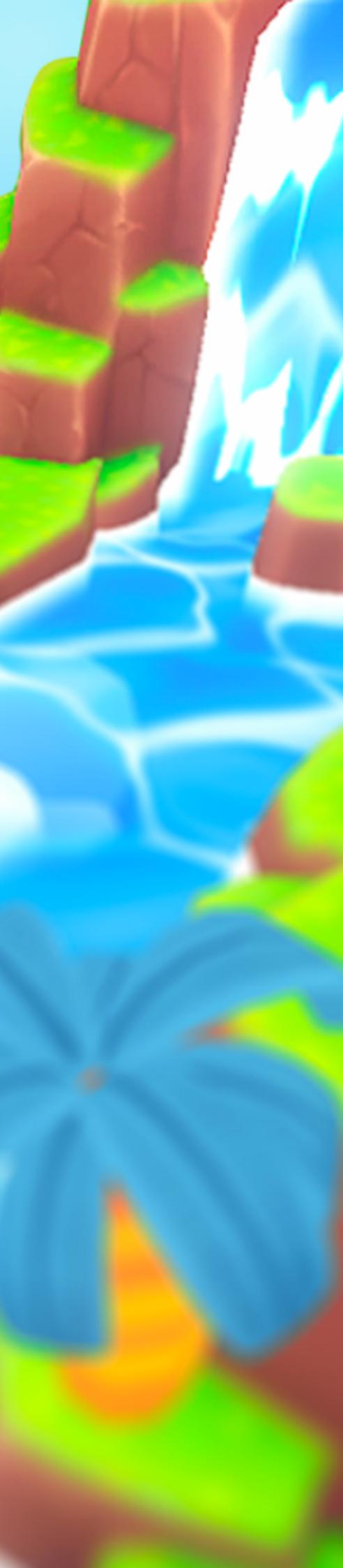
# About the author.

Fabrizio Espíndola is a Chilean video game developer, specialized in computer graphics. He has dedicated much of his career to developing visual effects and technical art for different projects, e.g., Star Wars - Galactic Defense, Dungeons and Dragons - Arena of War, Timenaus, 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 with the appearance of Donkey Kong Country, a game that aroused a deep interest in wanting to know how to develop that kind of technology. To date, he has more than nine years of experience in the industry, and this book contains a part of the knowledge that he has acquired during this time.

Years ago, while at university, he met a professor named Freddy Gacitúa, who once commented that "a person becomes professional when he contributes to others". These words were significant to him throughout his education and generated the need to bestow his knowledge on the international community of 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 would be one of the most important due to its intellectual nature.

# Content

## Chapter I | Introduction to the shader programming language

## Chapter II | Lighting, shadows and surfaces.

# Chapter III | Compute shader, ray tracing and sphere tracing.

One of the biggest problems that video game developers have when they start studying shaders in Unity is that there is little information for beginners on the web. Whether you are an independent developer or focused on AAA projects, it can be a little daunting due to the technical nature of the knowledge necessary to develop these types of programs.

Despite this challenge, Unity offers a significant advantage since it is multiplatform. It allows us to write our videogames only once and then export them to different devices, including consoles and smartphones. So, once we start our adventure into the world of shaders, we only need to write our code one time, and the software will take care of compiling it for the different platforms (OpenGL, Metal, Vulkan, Direct3D, GLES 20, GLES 3x).

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

Did you know that there are three types of Rendering Pipeline in Unity, and each of them has its own qualities? Throughout the book, we will specify each of them, verifying how Unity processes the graphics to project our video games on the computer screen.

## I. Topics we will see in this book.

The book is divided into three chapters to linearly address the topics to the extent that we need to; however, it is worth mentioning that this book will be subject to **structural changes** over time to improve the understanding of its content and achieve a good interaction with the reader.

All the code we will see 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 that needed before starting, such as the structure of a shader 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, the design of a function, Input vertex analysis, Output vertex 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, shadows, and surfaces.

Addresses highly relevant issues, such as normal maps and their implementation, reflection maps, lighting and shadow analysis, basic lighting model, surface analysis, mathematical functions, specularity and ambient light. We also review Shader Graph, its structure, functions in HLSL, nodes, properties and more. In this chapter, we will make our video game look professional with simple lighting concepts.

### Chapter III: Compute shader, ray tracing and Sphere tracing.

We will put into practice advanced concepts, such as the structure of a compute shader, buffer variables, kernels, sphere tracing implementation, implicit surfaces, constructive solid geometry, shapes and algorithms, introduction to ray tracing, configurations and high-quality rendering. Our studies will conclude in this chapter. We will investigate GPGPU programming (general-purpose GPU) using shaders of type .compute, reviewing the sphere tracing technique and using Direct Raytracing (DXT) in HDRP.

## II. Recommendations.

Working with a code editor with **IntelliSense** in graphics language programming is essential, 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 both C#, ShaderLab, and HLSL.

For those who will use Visual Studio Code editor, we recommend installing the following extensions: **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 knowledge or create professional-looking effects. It is assumed that the reader already knows, understands, and has access to the Unity interface, so we will not go into these details.

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 essential to have basic knowledge in arithmetic, algebra and trigonometry to understand more advanced concepts. Anyway, we will still review mathematical operations and functions to explain what we're developing fully.

## IV. Glossary.

Given their nature, in this book, we will find phrases and words distinguished from the rest, which we can quickly identify because we will highlight them to emphasise an explanation or concept. Likewise, there will be blocks of code in HLSL language to exemplify some functions.

The book will show in **bold** font (e.g. _Color) essential words or lines of code. Some relevant words or technical definitions will be shown in italic (e.g. normal), and constant values will be displayed in UPPER CASE (e.g. RGBA)

In the book, we can find blocks of code that include three points (…), which refer to variables and/or functions included by default within the code.

## V. Errata.

At the time of writing this book, we have taken every precaution to ensure the fidelity of the content. Even so, please remember that we are human beings, and it is very likely that some points may not have been well explained or may have incurred mistakes in the spelling/ grammar correction. If you find a conceptual error, code or other, we would appreciate

it if you could inform us by email at **contact@jettelly.com**, indicating "**USB Errata**" in the subject field; by this way, you will be helping other readers to reduce their level of frustration, and we can update the following releases.

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

## VI. Assets and donations.

Its members have developed all the work you see on Jettelly; this includes drawings, designs, videos, audio, tutorials, and everything you see with the brand. Jettelly is an independent game development studio, and your support is fundamental. If you want to support us financially, you can do it directly from our Patreon, (www.patreon.com/jettelly) or through our PayPal account (www.paypal.com/paypalme/jettelly).

This book has exclusive assets that are included in the download to reinforce the learning. These have been developed using Unity 2020.3.21f1 and tested in both Built-in Render Pipeline and Scriptable Render Pipeline.

## 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 you detect an illegal copy, please contact us at **contact@jettelly.com** (attaching the link if necessary). In this way, we can find a solution.

**Chapter I**

# Introduction to the shader programming language

## Initial observations.

Years ago, when I was just starting my studies about shaders in Unity, it was challenging to understand much of the content I found in the books for several factors. I still remember that day of studies, wishing to understand the operation of the semantics POSITION[n]; however, when I managed to find its definition, I found the following statement:
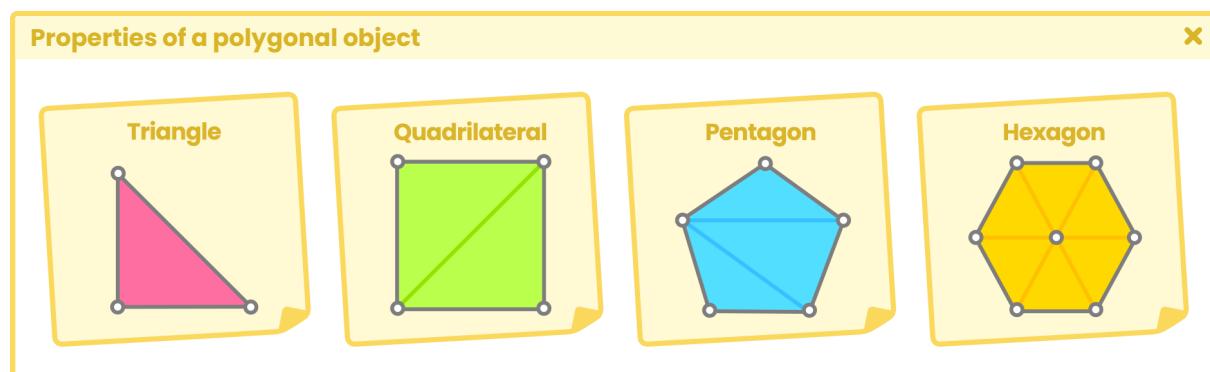
> *Vertex position in object-space.*

At that moment, I asked myself, what is the 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, the structure of a render pipeline, matrices, and coordinate systems.

## 1.0.1. | Properties of a polygonal object.

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



**Properties of a polygonal object**

Triangle | Quadrilateral | Pentagon | Hexagon
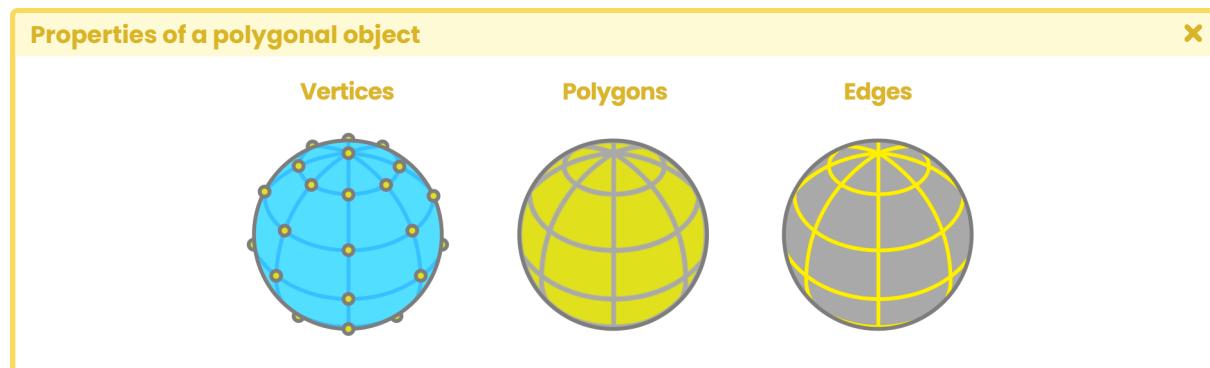
(Fig. 1.0.1a)

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, we can

find other primitives. The most common are: Spheres, Boxes, Quads, Cylinders and Capsules. These objects are different in shape but have similar properties; all have *vertices, tangents, normals, UV coordinates and color*, which are stored within a data type called *"mesh"*.

We can access all these properties independently within a shader and keep them in vectors (e.g. float4 pos: POSITION [n]). It is beneficial because we can modify their values and thus generate exciting effects. To understand this concept much better, we will give a small definition of the properties of a polygonal object.



*(Fig. 1.0.1b)*

## 1.0.2. | Vertices.
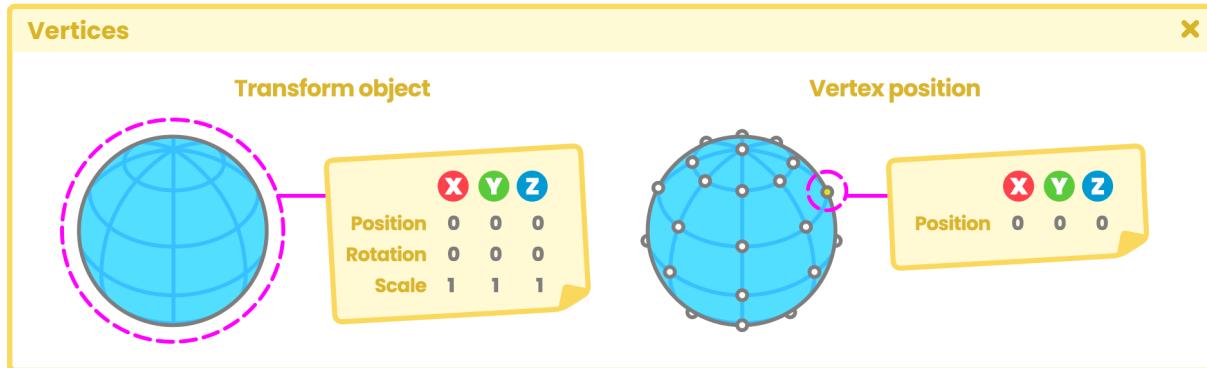
The vertices of an object, corresponding to the set of points that define the area of a surface in either a two-dimensional or three-dimensional space. In Maya and Blender, the vertices are represented as the intersection points of the mesh and an object.

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? Suppose, in Maya 3D, there are 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 about the object's pivot. Instead, the shape node, child of the transform node, contains the geometry attributes, that is, the position of the object's vertices concerning its volume. It means we could move, rotate or scale the set of vertices of an object, but at the same time, we could change the position of a specific point or vertices.

The POSITION[n] semantics exemplified in the previous paragraph is precisely the one that gives access to the position of the vertices concerning the volume of the same, that is, to the configuration exported by the shape node from Maya.
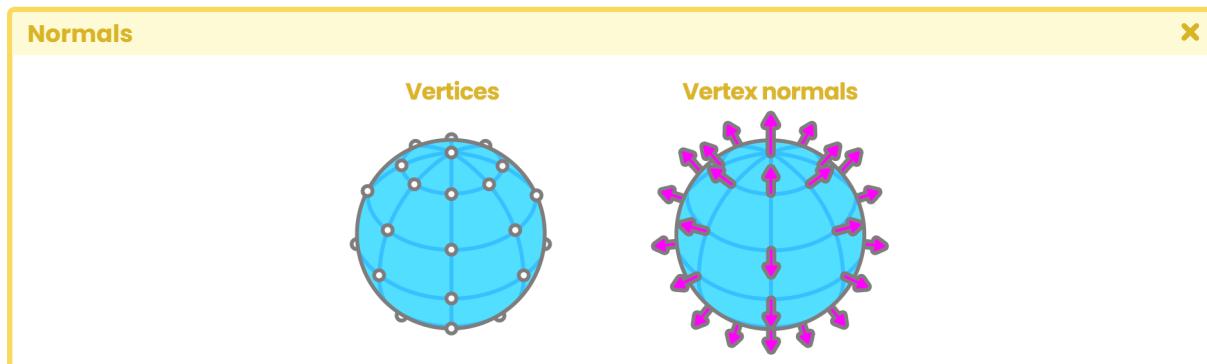


*(Fig. 1.0.2a. On the left, we can see the Transform node coming from Maya, and on the right, it's the Shape node.)*

## 1.0.3. | Normals.

Let's imagine that we have a blank sheet of paper, and we ask a friend to draw on the front face of the sheet. How could we 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, we can visualize the normals of an object by selecting the property **vertex normals**. It allows us to see where a vertex points in space and determines the hardness level between the different faces of an object.
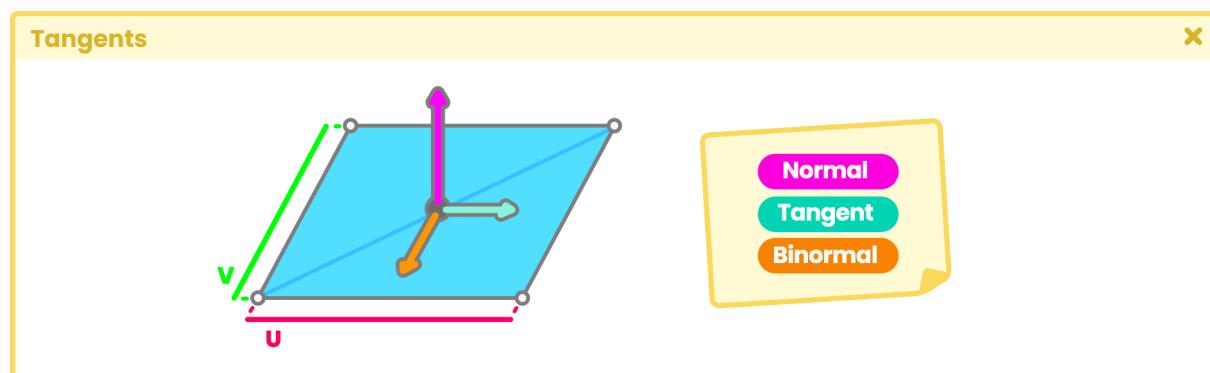


*(Fig. 1.0.3a. Graphical representation of the normals per each vertex)*

## 1.0.4. | Tangents.

According to Unity official documentation:

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

*W*hat does this mean? The tangents follow the U coordinate of the UV on each geometry face.
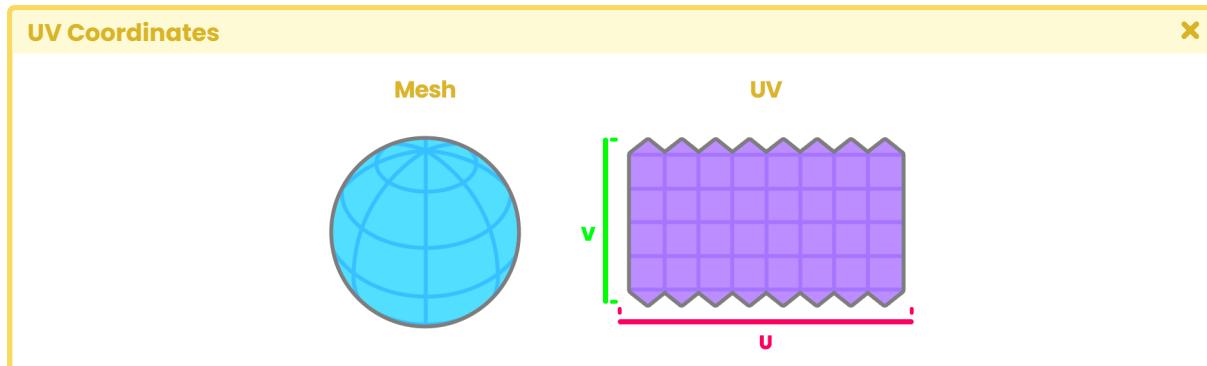


*(Fig. 1.0.4a.  By default, we cannot access the Binormals in a shader. Instead, we will have to calculate it concerning the normals and tangents.)*

Later in Chapter II, section 6.0.1, we will review this property in detail and include the binormals for the normal map implementation on an object.

## 1.0.5. | UV coordinates.
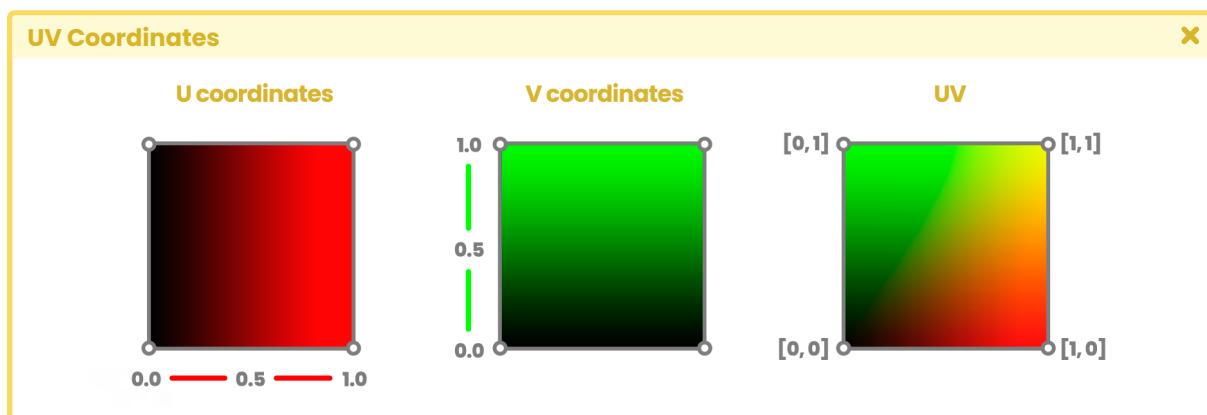
We have all changed the skin of our favorite character for a better one. UV coordinates are directly related to this concept, since they allow us to position a two-dimensional texture on the surface of a three-dimensional object. These coordinates act as reference points, which control which texels in the texture map correspond to each vertex in the mesh.

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

*(Fig. 1.0.5a, The 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 "zero" means the starting point and "one" is the endpoint.



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

## 1.0.6. | Vertex color.

When we export an object from 3D software, it assigns a color to the object to be affected, either by lighting or replicating another color. Such color is known as **vertex color** and corresponds to the color white by default, having the values "one" in RGBA channels. Later we will be able to see this concept in practice.

## 1.0.7. | Render pipeline architecture.
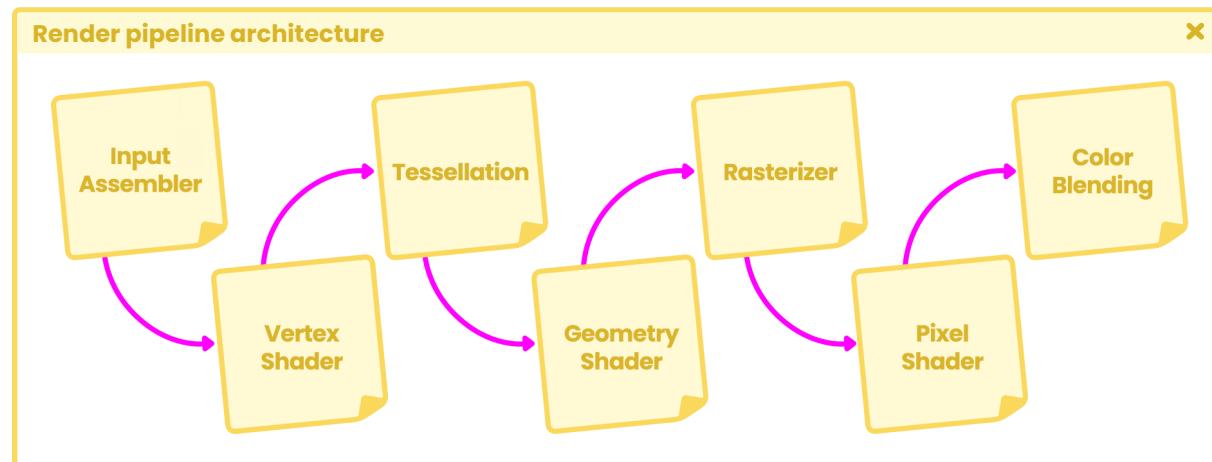
In current versions of Unity, there are three types of rendering pipeline: **Built-in RP**, **Universal RP** (called **Lightweight** in previous versions) and **High Definition RP**.

It is worth asking ourselves, what is a render pipeline, then? To answer this, the first thing we have to understand is the "pipeline" concept.

A pipeline is a series of stages that perform a more significant task operation. So what does rendering pipeline refer to? Let's think of this concept as the complete process that a polygon object must take (e.g. object with extension .fbx) to be rendered onto our computer screen; it is like an object travelling through the Super Mario pipes until it reaches its final destination. So, each rendering pipeline has its characteristics, and depending on the type we are using: material properties, light sources, textures, and all the functions that are occurring internally within the shader, will affect the appearance and optimization of objects on the screen.

Now, how does this process happen? For this, we must talk about basic rendering pipeline architecture. Unity divides this architecture into four stages: application, geometry processing, rasterization, and pixel processing.

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 we will define next.
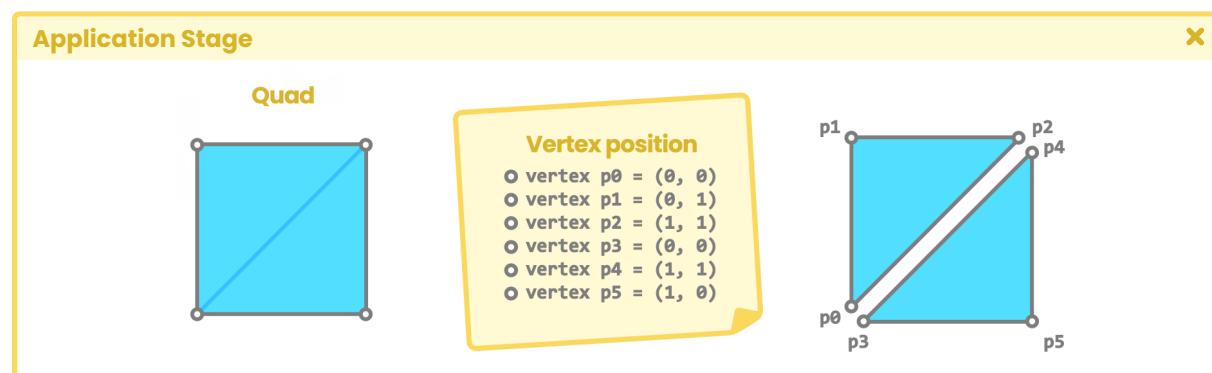


*(Fig. 1.0.7a. Logic Render Pipeline)*

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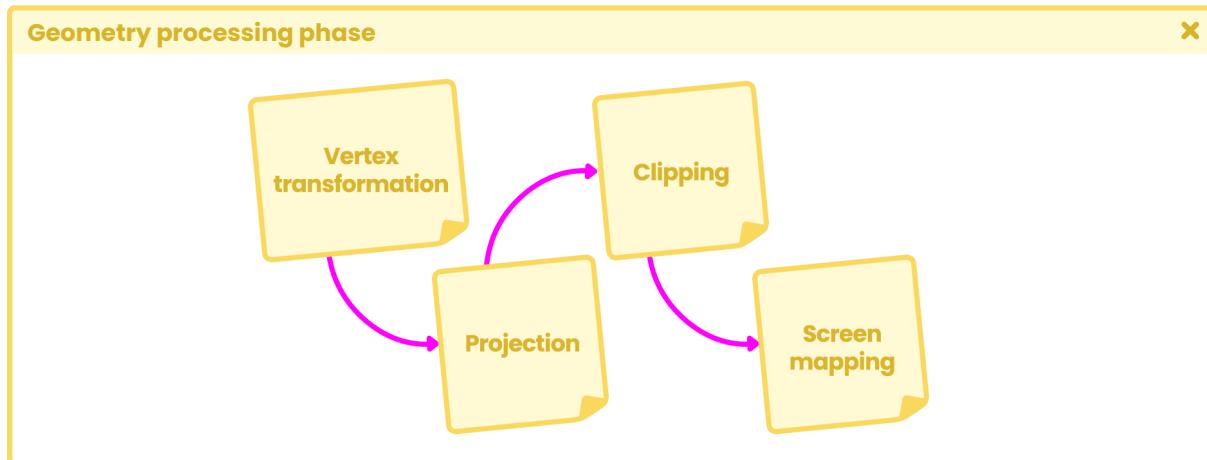


*(Fig. 1.0.8a)*

## 1.0.9. | Geometry processing phase.

The CPU requests the images that we see on our computer screen from the GPU. These requests are carried out in two main steps:

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

The geometry processing phase occurs on the GPU and is responsible for the vertex processing of our object. This phase is divided into four subprocesses which are: *vertex shading, projection, clipping* and *screen mapping*.

Geometry processing phase

Vertex transformation

Clipping

Projection

Screen mapping

*(Fig. 1.0.9a)*
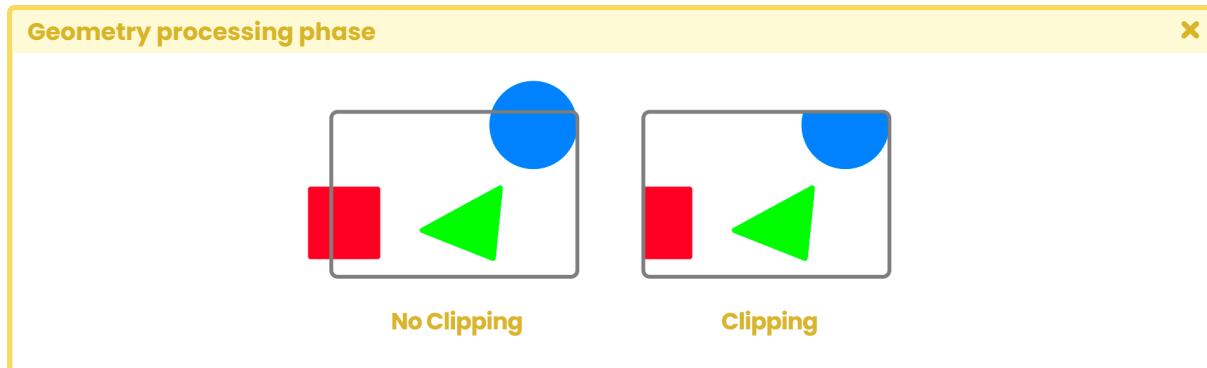
When the primitives have already been assembled in the application stage, the vertex shading, more commonly known as the **vertex shader stage**, handles two main tasks:

1.  It calculates the position of the vertices of the object.
2.  Transforms its position to different space coordinates so that they can be projected onto the computer screen.

Also, within this subprocess, we can select the properties that we want to pass on to the following stages. It means that within the **vertex shader stage,** we can include *normals, tangents, UV coordinates* etc.

*Projection* and *clipping* occur as part of the process, which varies according to the properties of our camera in the scene. It is worth mentioning that the whole rendering process occurs only for those elements that are within the camera frustum, also known as the *view-space*.

The projection and the clipping will depend on our camera, if it is set to perspective or orthographic (parallel). To understand this process, we are going to assume that we have a Sphere in our scene, where half of it is outside the frustum of the camera, so only the area of the Sphere that lies within the frustum will be projected and subsequently clipped on the screen, that is, the area of the Sphere that is out of sight will be discarded in the rendering process.

*(Fig. 1.0.9b)*

Once we have our clipped objects in the memory, they are subsequently sent to the screen map (screen mapping). At this stage, the three-dimensional objects that we have in our scene are transformed into screen coordinates, also known as window coordinates.

## 1.1.0. | Rasterization stage.

Our third stage corresponds to the rasterization. At this point, our objects have screen coordinates (2D coordinates), and now we must look for the pixels in the projection area. The process of finding all the pixels that are occupied by an on-screen object is called rasterization. This process can be seen as a step of synchronization between the objects in our scene and the pixels on the screen.
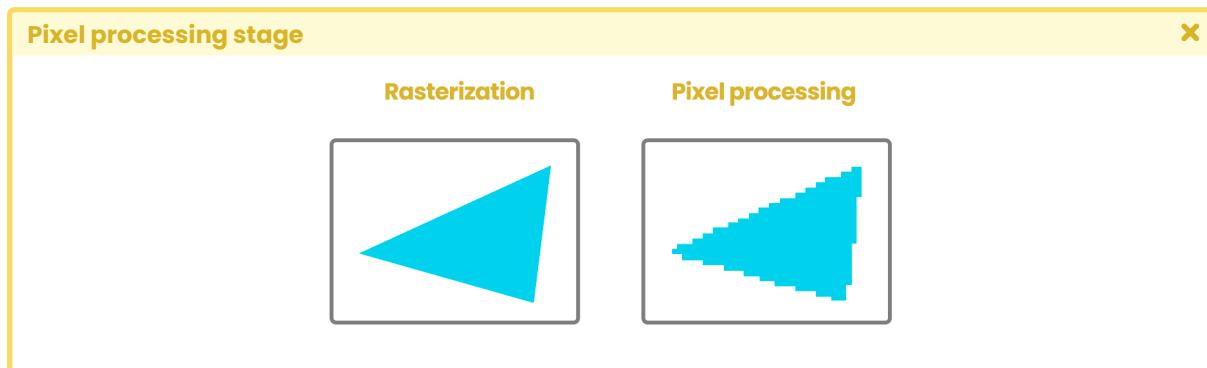
For each object, the **rasterizer** performs two processes:

1. Triangle setup.
2. Triangle traversal.

Triangle setup is in charge of generating the data that will be sent to the triangle traversal. It includes the equations for the edges of an object on the screen. The triangle traversal lists the pixels that are covered by the area of the polygon object. In this way, a group of pixels called "fragments" is generated. However, this word is used many times to refer to an individual pixel.

## 1.1.1. | Pixel processing stage.
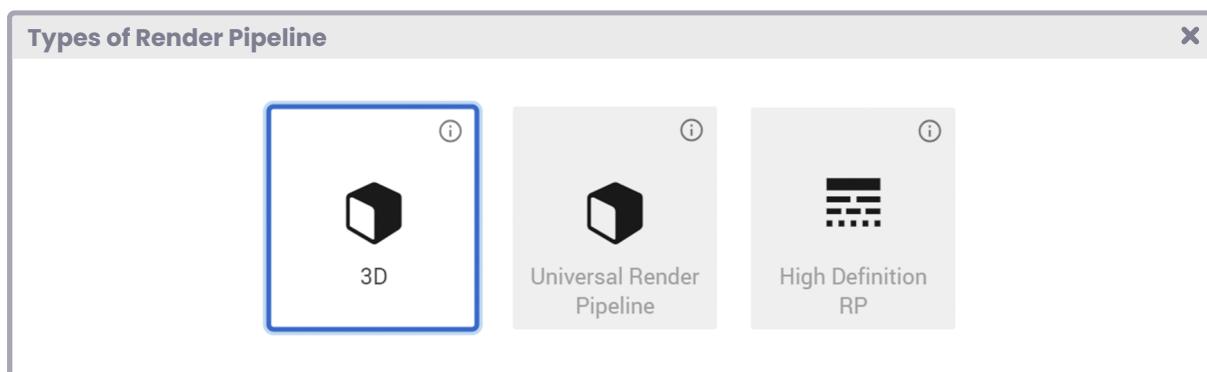
Using the interpolated values from the previous processes, this last stage starts when all the pixels are ready to be projected onto the screen. At this point, the **fragment shader stage**, also known as a **pixel shader stage,** begins and is responsible for the visibility of each pixel. Basically what it does is compute 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 we already know, there are three types of render pipelines in Unity. By default, we can find a **Built-in RP** that corresponds to the oldest engine belonging to the software, on the other hand, **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 we create a new project in Unity, we can choose between these three rendering engines. Our choice depends on the needs of the project at hand)*

Regardless of the rendering pipeline, if we want to generate an image on the screen, we have to travel through the "pipeline".
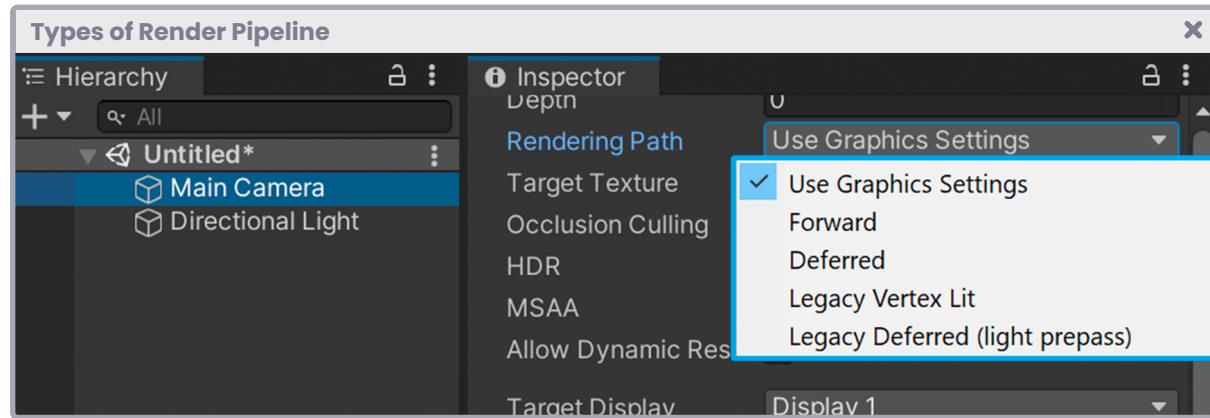
A pipeline can have different processing paths. These are known as **render paths**; as if the example pipe in section 1.0.7 had more than one way to reach its destination.

A rendering path corresponds to a series of operations related to lighting and shading objects. This allows us to graphically process an illuminated scene (e.g. a scene with directional light and a Sphere).

Examples of these paths are *forward rendering, deferred shading, legacy deferred* and *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 pipeline render that are included in Unity. This is because it has greater graphics card compatibility and a lighting calculation limit, making it a more optimized process.

Please note that in Universal RP, we can only use *forward* as a rendering path, whereas High Definition RP allows illuminated material rendering using either *forward* or *deferred shading*.



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

To understand this concept, we are going to suppose that we have an "object" and a "direct light" in a scene. The interaction between the light and the object is based on two points, they are.

1. Lighting characteristics.
2. Material characteristics.

The interaction between these two elements is called the **lighting model**.

The basic lighting model corresponds to the sum of three different properties, which are ambient color, diffuse reflection and specular reflection.

The lighting calculation is carried out within the shader, this can be carried out *per-vertex* or per *fragment*. When the illumination is calculated by vertex it is called *per-vertex lighting* and is performed in the **vertex shader stage**, likewise, when it is calculated per *fragment* it is called *per-fragment* or *per-pixel shader* 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 (e.g. normal maps, pixel lighting, shadows, etc.). This rendering path has two different code written passes that we can use in our shader, the first, **base pass** and the second **additional pass**.

In the base pass, we can define the **ForwardBase** *light mode* and in the additional pass, we can define the **ForwardAdd** *light mode*. Both are characteristic functions of a shader with lighting calculation. The base pass can process directional light *per-pixel* and will use 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 (skylight).
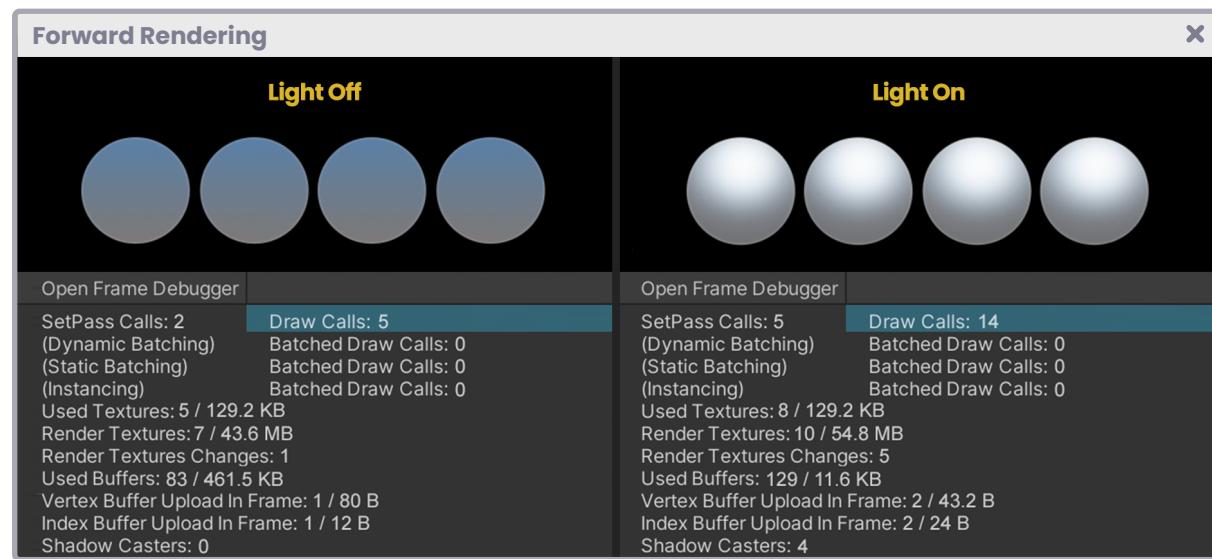
As its name says, the additional pass can process "additional lights" per-pixel or also shadows that affect the object, what does this mean? If we have two lights in the scene, our object will be influenced only by one of them, however, if we have defined an additional pass for this configuration, then it will be influenced by both.

One point that we must take into consideration is that each illuminated pass will generate a separate *draw call*. A **draw call** is a call graphic that is made in the GPU every time we want

to draw an element on the screen of our computer. These calls are processes that require a large amount of computation, so they need to be kept to a minimum possible, even more so if we are working on projects for mobile devices.

To understand this concept, we are going to suppose that we have four Spheres and one directional light in our 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 Sphere, 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 draw calls in total.



*(Fig. 1.1.3a. In the image above, we 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 we add another pass in our shader, then we are going to add a new draw call for each object, and consequently, the graphic load will increase significantly.

There are some ways to optimize this process, which we will talk about later in the book. For now, we will continue the rendering path concept.

## 1.1.4. | Deferred shading.

This rendering path ensures that there is only one lighting pass computing each light source in our 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 we could generate a significant amount of light 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 multi-light source computing, it brings with it some hardware compatibility restrictions and issues.

## 1.1.5. | What rendering engine should I use?

This is a very common and recurring question these days.

In the past, there was only Built-in RP, so it was very easy to start a project, be it in 2D or 3D. However, nowadays, we must start our video game according to its needs, so we may wonder, how could we know what our project needs? To answer this question, we must consider the following factors:

1. If we are going to develop a PC video game we can use any of the three Unity render pipelines available since generally, a PC has larger computing power than a mobile device or even a console. Then, if our video game is for PC, do we need it to be viewed graphically in a high or medium definition? In the case that we require our video game in high definition, we can create it in both High Definition RP and Built-in RP.
2. If we want our video game graphically in medium definition, we can use Universal RP or, as in the previous case; Built-in RP too. Now, why is Built-in RP an option in both cases?

Unlike the previous ones, this rendering 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 our render pipeline is the shaders. Generally, in both High Definition RP and Universal RP, the shaders are created in **Shader Graph**, which is a package that includes an interface that allows us to create a shader through nodes,

which has a positive and negative side. On the one hand, we can create shaders visually through nodes without the need to write code in HLSL, however, on the other hand, if we update our version of Unity to a newer version in production (e.g. from 2019 to 2020), the shaders will most likely stop compiling because Shader Graph has versions and updates independent of the version of Unity.

The best way to create shaders in Unity is through the HLSL language, since this way we can ensure that our program compiles in the different rendering pipelines and continues to work regardless of any Unity upgrade. Later in this chapter, we will review in detail the structure of a program in HLSL.

## 1.1.6. | Matrices and coordinates systems.

One of the concepts that we see frequently in the creation of shaders is that of **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, we can find:

- 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. Their conceptual representation is as follows:

```
UNITY_MATRIX
(
        Xx,        Yx,        Zx,        Tx,
        Xy,        Yy,        Zy,        Ty,
        Xz,        Yz,        Zz,        Tz,
        Xt,        Yt,        Zt,        Tw
);
```
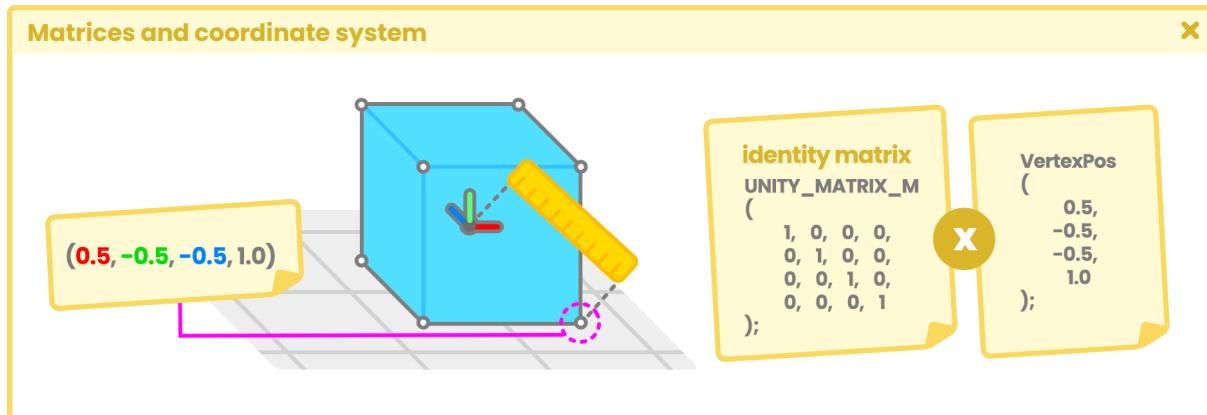
As we explained previously in section 1.0.2 where we talked about vertices, a polygon object has two nodes by default. In Maya, these nodes are known as *transform* and *shape*, and both are in charge of calculating the position of the vertices in a space called *object-space*, which defines the position of the vertices about the position of the object's center.

The final value of each vertex in the object space is multiplied by a matrix known as the *model matrix* (UNITY_MATRIX_M), which allows us to modify the transformation, rotation and scale values of the vertices of an object. Every time that we rotate, change position or scale our object then the *model matrix* is updated, but how does this process happen?

To understand this we are going to suppose that we have a Cube in our scene and we want to transform its values using a *model matrix*. We will start by taking a vertex of our Cube that is at position XYZW [0.5f, -0.5f, -0.5f, 1] relative to its center.

It should be mentioned that channel "W" in the previous example corresponds to a "homogeneous" system of coordinates that allow us to handle vectors and points uniformly. In matrix transformations, the W coordinate can have a value of "zero or one". When W equals one (e.g. X, Y, Z, 1), it refers to a point in space, whereas, when it equals zero (e.g. X, Y, Z, 0), it refers to a direction in space.

Later in this book, we will talk about this system when we multiply vectors by matrices and vice versa.
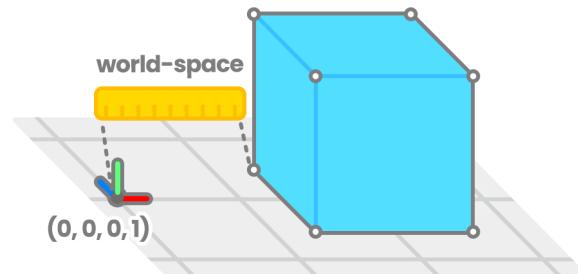
*(Fig. 1.1.6a. Identity matrix refers to the matrix default values)*

One of the elements to consider concerning matrices is that multiplication can only be carried out when the number of columns in the first matrix is equal to the number of rows in the second. As we already know, our *model matrix* has a dimension of four rows and four columns (4x4), and the vertex position has a dimension of four rows and one column (4x1). Since the number of columns in the *model matrix* is equal to the number of rows in the vertex position, they can be multiplied and the result will be equal to a new matrix of four rows and one column (4x1), which would define a new vertex position. This multiplication process occurs for all vertices in our object, and this process is carried out in the **vertex shader stage** in our shader.

We already know that *object-space* refers to the position of the vertices of an object concerning its center, so what do *world-space*, *view-space*, and *clip-space* mean? The concept is the same.

*World-space* refers to the position of the vertices according to the center of the world; the distance between the point XYZW [0, 0, 0, 1] of the grid in our scene and the position of a vertex in the object. If we want to transform a space coordinate from *object-space* to *world-space*, we can use the Built-in shader variable unity_ObjectToWorld.

**Matrices and coordinate system**

world-space

(0, 0, 0, 1)
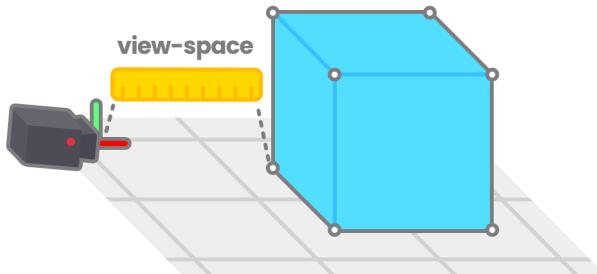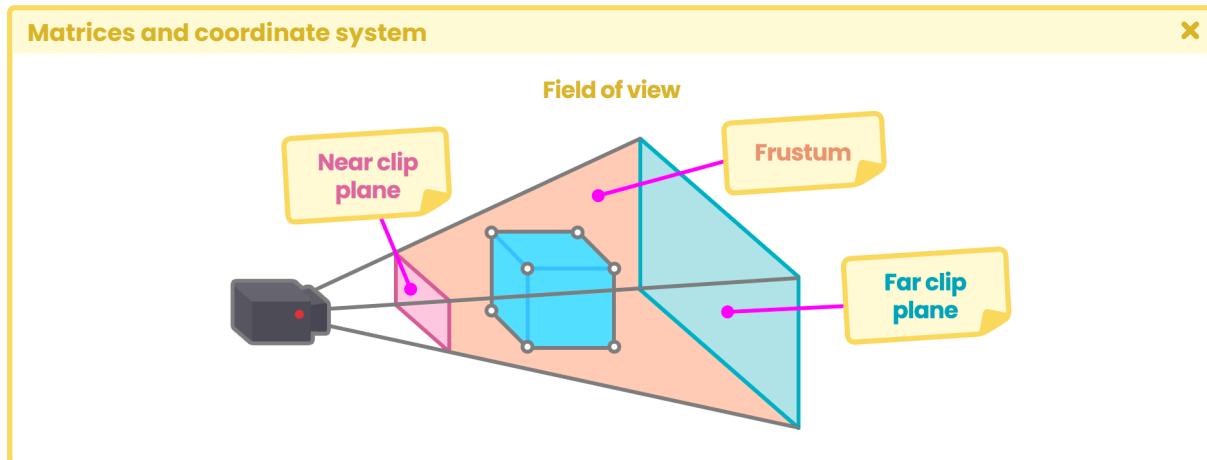
*(Fig. 1.1.6b)*

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



**Matrices and coordinate system**

view-space

*(Fig. 1.1.6c)*

Finally, *clip-space*, also known as *projection-space*, refers to the position of a vertex of our object about the camera's frustum, so, this factor will be affected by the camera's near *clipping plane, far clipping plane* and *field of view*. Again, if we want to transform a space coordinate from view-space to clip-space, we can do it using the UNITY_MATRIX_P matrix.

**Matrices and coordinate system**

Field of view

Near clip plane

Frustum

Far clip plane

*(Fig. 1.1.6d)*

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

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 vertices of an object from object-space to clip-space. Let's remember that our polygonal object has been created in a "three-dimensional" environment while the screen of our computer, where it will be projected, is "two-dimensional", therefore we will have to transform our object from one space to another.

Later in this book, we will review these concepts in detail when we use the UnityObjectToClipPos function included in our shader, in the vertex shader stage.

# Shaders in Unity.

## 2.0.1. | What is a shader?
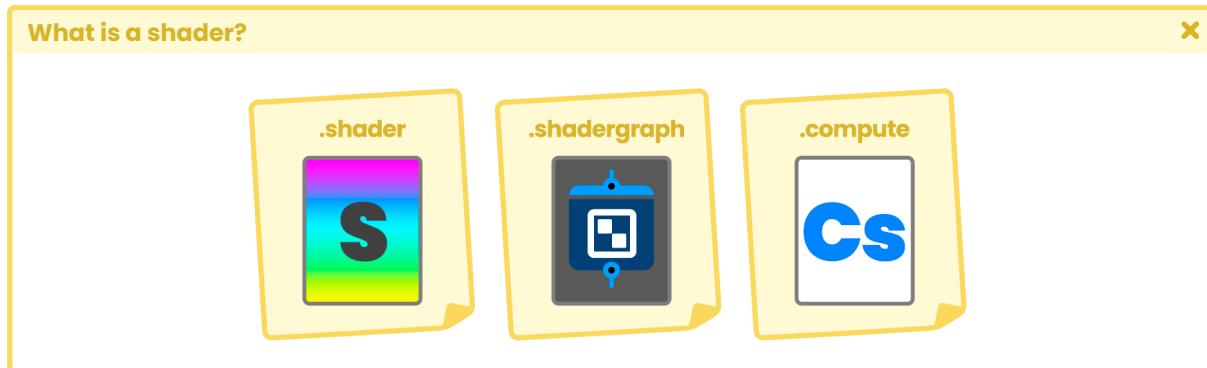
Considering some previous knowledge, we will go into the topic of shaders in Unity. A shader is a small program with a "**.shader**" extension (e.g. color.shader), which can be used to generate interesting effects in our projects. Inside, it has mathematical calculations and lists of instructions (commands) that allow color processing for each pixel within the area covering an object on our computer screen.

This program allows us 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, we have programs with the "**.shader**" extension that are capable of compiling in the different types of render pipelines.

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

There is also another type of program with the extension "**.cginc**" which we will review in detail later on. For now, we will limit ourselves to making the following association: "**.cginc**" is linked to "**.shader**" CGPROGRAM, and "**.hlsl**" is linked to "**.shadergraph**" HLSLPROGRAM. Knowing this analogy is fundamental because each extension fulfills a different function and is used in specific contexts.

**What is a shader?** ✕

.shader    .shadergraph    .compute

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

In Unity, there are at least four types of structures defined to generate shaders, among which we can find the combination of **vertex shader** and **fragment shader** then **surface shader** for automatic lighting calculation and **compute shader** for more advanced concepts. Each of these structures has previously described properties and functions that facilitate the compilation process; we can also easily define our operations, since the software adds these structures automatically.

## 2.0.2. | Introduction to the programming language.

Before starting in the definition of code, we must take into consideration that in Unity there are three programming languages associated with shader development, these are HLSL (High-Level Shader Language - Microsoft), Cg (C for Graphics - NVIDIA) which still compiles into the shader but is no longer used in current versions of the software, and ShaderLab (declarative language - Unity).

We will start our adventure working with the Cg and ShaderLab languages in Built-in RP and later give way to the introduction of HLSL in Universal RP.

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

When we create a shader, our 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 shortly these blocks will be replaced by HLSLPROGRAM and ENDHLSL since

34

HLSL is the official shader programming language in current versions of Unity (version 2019 onwards).

All shaders in Unity (except Shader Graph and Compute) are written within a declarative language called ShaderLab. The syntax of this language allows you to display the properties of a shader in the Unity inspector. This is very interesting since we can manipulate the values of variables and vectors in real-time, adjusting our shader to get the desired result.

In ShaderLab we can manually define several properties and commands, among which is the **Fallback** block, which is compatible with the different types of rendering pipelines that exist.
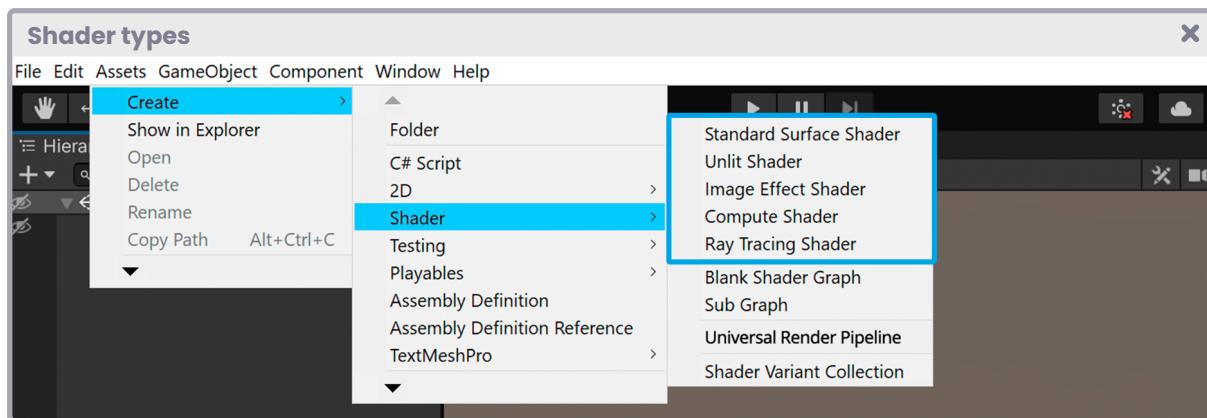
*Fallback* is a fundamental code block in multiplatform games. It allows us to compile a different shader to one that has generated an error. If the shader breaks in its compilation process, *Fallback* returns a different shader, and so the graphics hardware can continue its work.

**SubShader** is another block in ShaderLab which allows us to 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, we 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).

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

*(Fig. 2.0.3a. We 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 used to create the project. Another variable that could affect the number of shaders that appear in the list would be **Shader Graph**. If the project were created in Universal RP or High Definition RP, it may have the Shader Graph package included, which increases the number of shaders that can be created.

We will not go into details about this subject for now; since we must understand some concepts before starting with it, we will simply limit ourselves to work with the shaders that come by default in Built-in RP.

Before creating our first shader, we will do a little review of the different types that exist in the software.

### 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 we want to create a shader that interacts with light, we have 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 in some cases includes albedo, specular, and diffuse.

### 2.0.5. | Unlit shader.

The "Lit" word refers to a material affected by illumination, and "Unlit" is the opposite. The **unlit Shader** refers to the primary color model and will be the base structure that we will generally use to create our effects. This type of program, ideal for low-end hardware, has no optimization in its code; therefore, we can see its complete structure and modify it according to our needs. Its main feature is that it works both in Built-in and Scriptable RP.

### 2.0.6. | Image effect shader.

It is structurally very similar to an Unlit Shader. Image effects are used mainly in post-processing effects in Built-in RP and require the function "OnRenderImage" (C #).

### 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 some game processing.

Chapter three 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 allows Ray Tracing processing on the GPU. It works only in High Definition RP and has some technical limitations. If we want to work with DXR (DirectX Ray Tracing), we must have at least one GTX 1080 graphics card or equivalent with RTX support, Windows 10 version 1809+ and Unity 2019.3b1 onwards.

We can use this kind of program to replace the ".compute" type shader in processing algorithms for ray-casting, e.g., global illumination, reflections, refraction, or caustic.

# Properties, commands and functions.

### 3.0.1. | Structure of a vertex / fragment shader.

To analyze its structure, we will create an **Unlit Shader** and call it "**USB_simple_color**". As we already know, this type of shader is a basic color model and does not have great optimization in its code, this will allow us to analyze in-depth its various properties and functions.

When we create a shader for the first time, Unity adds default code to facilitate its compilation process. Within the program, we can find blocks of code structured in such a way that the GPU can interpret them. If we open our **USB_simple_color** 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

            #include "UnityCG.cginc"
```

*Continued on next page.*

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

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

sampler 2D _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;
}
ENDCG
        }
    }
}
```

Likely, we do not fully understand what is happening in the different code blocks from the shader we just created. However, to begin our study, we will 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
            // programa Cg – HLSL in this field
            ENDCG
        }
    }
    Fallback "ExampleOtherShader"
}
```

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

The previous example shows the main structure of a shader. The shader starts with a path in the **inspector** (InspectorPath) and a **name** (shaderName), then the **properties** (e.g. textures, vectors, colors, etc.) after that the **SubShader** and at the end of it all is the optional Fallback.

The "inspectorPath" refers to the place where we will select our shader to apply it to a material. This selection is made through the Unity Inspector.

We must remember that we cannot apply a shader directly to a polygonal object, instead, it will have to be done through a previously created material. Our USB_simple_color shader

**41**

has the path "Unlit" by default, this means that: from Unity, we must select our material, go to the inspector, search the path Unlit and apply the material called USB_simple_color.

A structural factor that we must take into consideration is that the GPU will read the program from top to bottom linearly, therefore, if shortly we create a function and position it below the code block where it will be used, the GPU will not be able to read it, generating an error in the shader processing, therefore Fallback will assign a different shader so that graphics hardware can continue its process.

Let's do the following exercise to understand this concept.

**Structure of a vertex / fragment shader**                                    ✕

```
// 1 . declare our function
float4 ourFunction()
{
    // your code here …
}
// 2. we use the function
fixed4 frag (v2f i) : SV_Target
{
    // we are using the function 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 for another.

In section 4.0.4 we will talk in detail about the structure of a function. For now, the only important thing is that in the previous example its 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**".

Let's look at a different case.

```
Structure of a vertex / fragment shader                              ✕

// 2. we use our function

fixed4 frag (v2f i) : SV_Target

{

    // we are using the function here

    float4 f = ourFunction();

    return f;

}

// 1 . declare the function

float4 ourFunction()

{

    // your code here …

}
```

On the contrary, this structure will generate an "error", because the function called "**OurFunction**" has been written below the code block which it is using.

## 3.0.2. | ShaderLab shader.

Most of our shaders written in code will start with the declaration of the **Shader**, then its path in the Unity **Inspector** and finally the **name** that we assign to it (e.g. Shader "shader inspector path/shader name").

Both the properties such as the SubShader and the Fallback are written inside the "Shader" field in ShaderLab declarative language.

```
ShaderLab shader                                                     ✕

Shader "InspectorPath/shaderName"

{

    // write ShaderLab code here

}
```

Since USB_simple_color has been declared as "Unlit/USB_simple_color" by default, if we want to assign it to a material, then we will have to go to the Unity Inspector, look for the Unlit path and then select "USB_simple_color".

Both the path and the name of the shader can be changed as needed 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
}
```

---
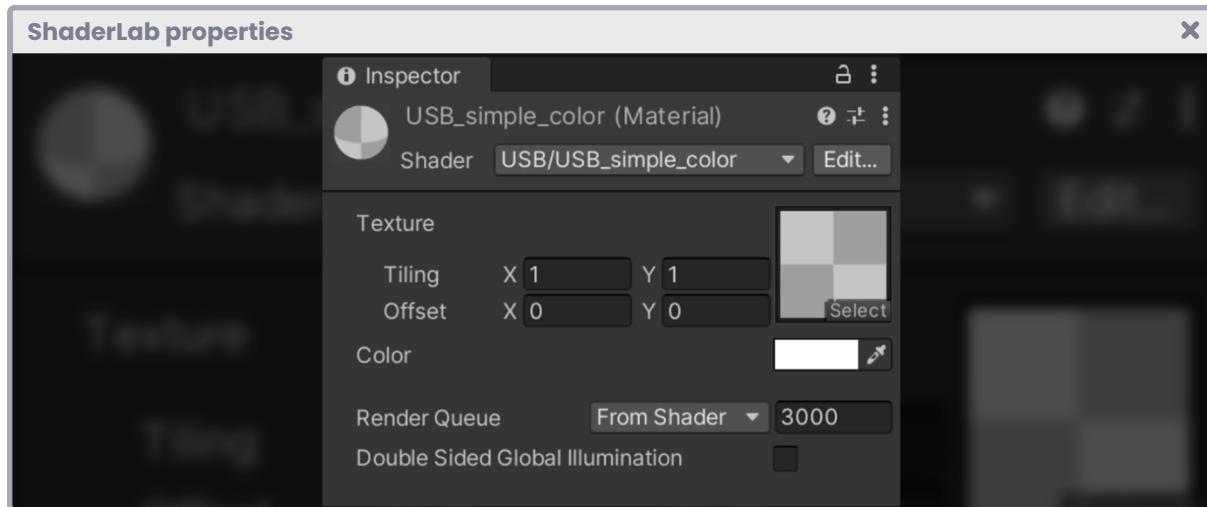
## 3.0.3. | ShaderLab properties.

The properties correspond to a list of parameters that can be manipulated from the Unity inspector. There are eight different properties both in value and usefulness. We use these properties concerning the shader that we 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 (e.g. _MainTex), "**display name**" corresponds to the name of the property that will be displayed in the Unity inspector (e.g. Texture), "**type**" indicates the property type (e.g. Color, Vector, 2D, etc.) and finally, as its name implies, "**defaultValue**" is the default value assigned to the property (e.g. if the property is a "Color" we can set it as white as follows (1, 1, 1, 1)).

---

44

*(Fig. 3.0.3a)*

If we look at the *properties* of our USB_simple_color shader we will notice that there is a texture property that has been declared inside of the field, we can corroborate this in the following line of code.

```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
}
```

One factor to consider is that when we declare a property, it remains "open" within the field of properties, therefore we must avoid the 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 allow us to add numerical values to our shader. Let's suppose that we want to create a shader with functions of illumination, where "zero" equals 0% illumination and "one" equals 100% illumination. We can create a range for this (e.g. Range (min, max)) and configure the minimum, maximum and default illumination values.

The following syntax declares numbers and sliders in our shader:

```
// 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
    }
}
```
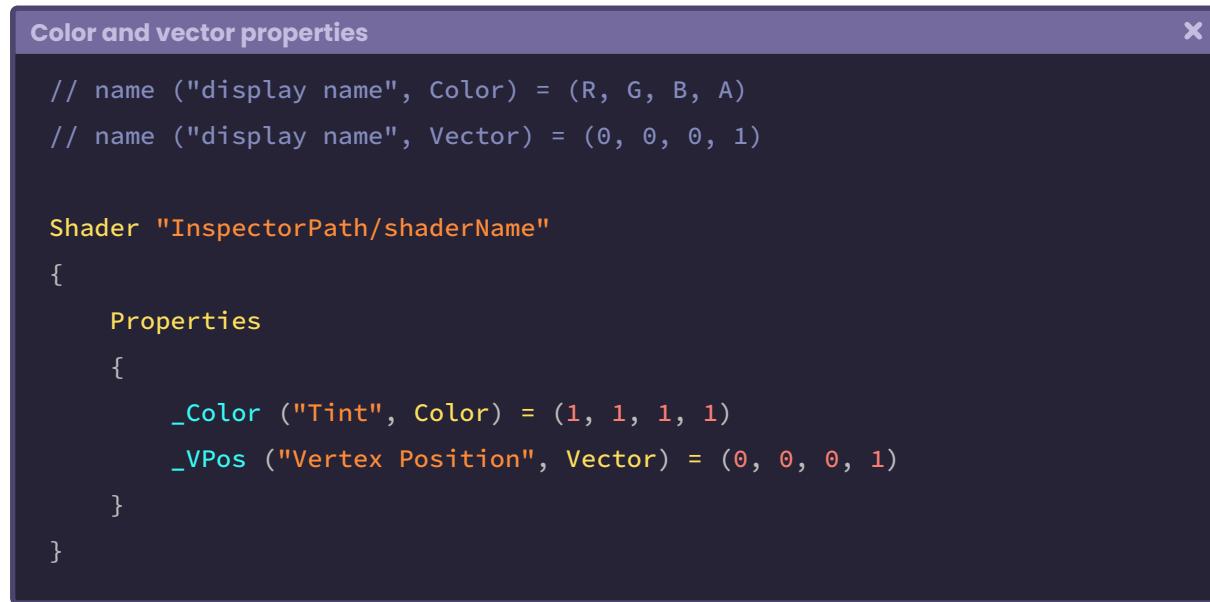
In the previous example we declared three properties, a "floating range" type called **_Specular**, another "floating scale" type called **_Factor**, and finally, an "integer" type called **_Cid**.

### 3.0.5. | Color and vector properties.

With this property, we can define colors and vectors in our shader.

We are going to suppose that we want to create a shader that can change color in execution time, for this we would have to add a color property where we can modify the RGBA values of the shader.

To declare colors and vectors in our shader, use the following syntax:

```
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 previous example, we declared two properties, a "color" called **_Color** and a "vector" called **_VPos**.

## 3.0.6. | Texture properties.

These properties allow us to implement textures within our shader.

If we want to place a texture on our object (e.g. a 3D character), then we would have to create a 2D property for its texture and then pass it through a function called "tex2D" which will ask us for two parameters: the texture and the UV coordinates of our object.

A property that we will use frequently in our video games is the "Cube" which itself refers to a "Cubemap". This type of texture is quite useful for generating reflection maps, e.g., reflections in our character's armor or to metallic elements in general.

Other types of textures that we can find are those of the 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 our 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 our program will be written in either Cg or HLSL language. As they are two different languages, we have to 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" we must first declare the property in ShaderLab, then the global variable using the same name in Cg or HLSL, and then we can finally use it.

48

```
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 previous example, we declared two properties: **_MainTex** and **_Color**. Then we created two connection variables within our CGPROGRAM, these correspond to "**sampler2D _MainTex**" and "**float4 _Color**". Both the properties and the connection variables must have the same name so that the program can recognize them.

In section 3.2.7 we will detail the operation of a 2D sampler when we talk about data types.

### 3.0.7. | Material property drawer.
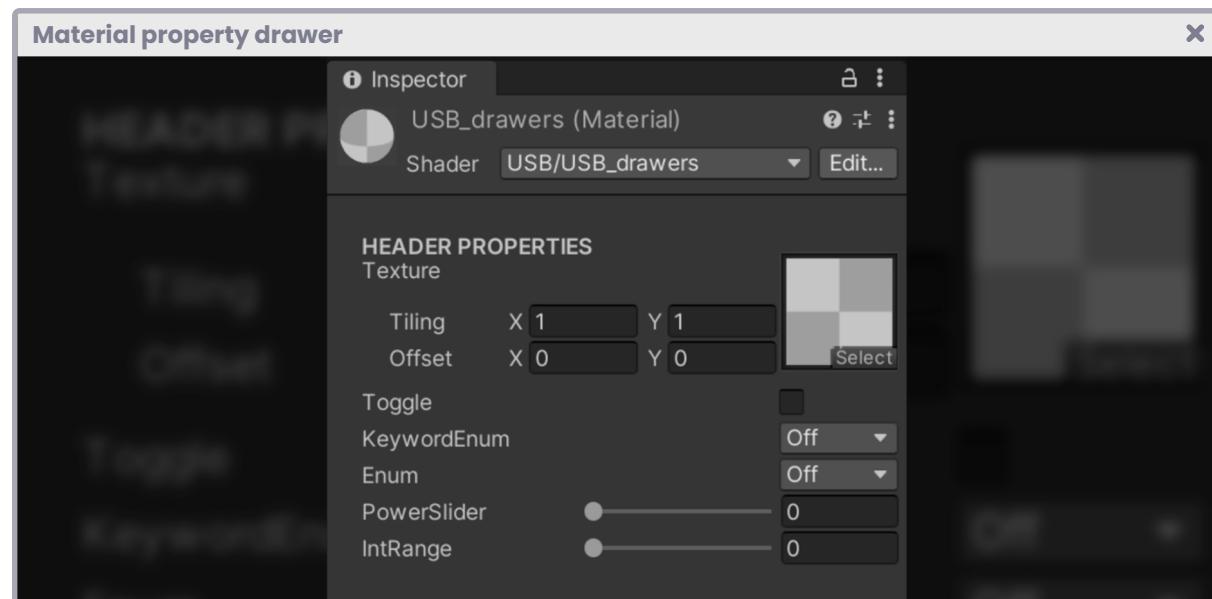
Another type of property that we can find in ShaderLab are "drawers". This class allows us to generate custom properties in the Unity Inspector, thus facilitating the programming of conditionals in the shader.

By default, this type of property is not included in our shader, instead, we have to declare them according to our 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 these properties, we can generate multiple states within our program, allowing the creation of dynamic effects without the need to change materials at execution time. We generally use these drawers together with two types of **shader variants**, these refer to **#pragma multi_compile** and **#pragma shader_feature**.



*(Fig. 3.0.7a)*

### 3.0.8. | MPD Toggle.

Within ShaderLab we cannot use boolean type properties, instead, we have the Toggle that fulfills the same function. This drawer is going to allow switching from one state to another using a condition within our shader. To run it, we must first add the word Toggle between brackets and then declare our property, taking into account that it must be a Float type. 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:

```
MDP Toggle                                                    ✕
[Toggle] _PropertyName ("Display Name", Float) = 0
```

As we see, we add the Toggle in brackets, then we declare the property, then the display name, followed by the Float data type, and finally we initialize the property to "Off" since we add a zero in its default value.

Something that we must consider when working with this drawer is that, if we want to implement it in our code, we will have to use the #pragma **shader_feature**. This belongs to the shader variants and its function is to generate different conditions depending on the state it is in (enabled or disabled). To understand its implementation, we will 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
    }
```
*Continued on next page.*

```
    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 conditions
            #if _ENABLE_ON
                return col;
            #else
                return col * _Color;
            #endif
            }
            ENDCG
        }
    }
}
```
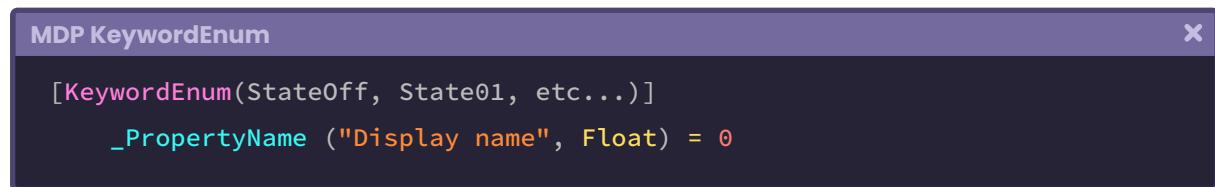
In this example, we declared a Toggle-type property called "**_Enable**". Then we added it to the **shader_feature** found in the CGPROGRAM, however, unlike the property in our program, the Toggle has been declared as "**_ENABLE_ON**", why is this? The variants added in shader_feature are "constants" therefore they are written in capitals, this means that if, for example, our property had been called _Change, then in the shader variant it should be added as "_CHANGE". The word **_ON** corresponds to the default state of the Toggle, so, if the _Enable property is active, we return the default texture color in the fragment shader stage, otherwise we multiply the _Color property by itself.

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

## 3.0.9. | MPD KeywordEnum.

This drawer generates a **pop-up** style menu in the material inspector. Unlike a Toggle, this drawer allows you to configure up to nine different states for the shader. To execute it we must add the word "KeywordEnum" in brackets and then list the set of states that we are going to use.

```
MDP KeywordEnum                                              ✕
  [KeywordEnum(StateOff, State01, etc...)]
      _PropertyName ("Display name", Float) = 0
```

In the previous example, we add the KeywordEnum drawer in brackets, and then we list its states, where the first corresponds to the default state (StateOff). We continue with the property declaration, display name in the material inspector, its Float data type and finally, we initialize with its default value.

To declare this drawer within our code, we can use both the shader variant **shader_feature** and **multi_compile**. The choice will depend on the number of variants that we want to include in the final build.

As we 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 great for exporting or compiling multiple states that will change at execution time (e.g. star status in Super Mario).

To understand its implementation, we will perform 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
            }
            ENDCG
        }
    }
}
```

In this example, we declare a KeywordEnum type property called "**_Options**" and configure three states for it (Off, Red and Blue). Later we 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, we define the three states for our shader that correspond to color changes for the main texture.

## 3.1.0. | MPD Enum.

This drawer is very similar to the KeywordEnum with the difference that it can define a "value/id" as an argument and pass this property to a command in our shader to change its functionality 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. To understand their implementation, we will perform the following operation:

MPD Enum                                                                           ✕

```
Shader "InspectorPath/shaderName"
{
    Properties
    {
        // declare drawer
        [Enum(Off, 0, Front, 1, Back, 2)]
        _Face ("Face Culling", Float) = 0
```
*Continued on next page.*

```
        }
    SubShader
    {
        // we use the property as a command
        Cull [_Face]
        Pass { … }
    }
}
```

As presented in this example, we declare a property type "**Enum**" called "**_Face**" and we pass as an argument the values: Off, 0, Front, 1, Back and 2. Then we add the property to the command "Cull" found in the SubShader; this way we can change the object face we want to render from the material inspector. In section 3.2.1 we will talk about the Cull command in detail.

### 3.1.1. | MPD PowerSlider and IntRange.

These drawers are quite useful when working with numerical ranges and precision. On the one hand, we have the PowerSlider, which allows us to generate a non-linear slider with curve control.

Its syntax is as follows:
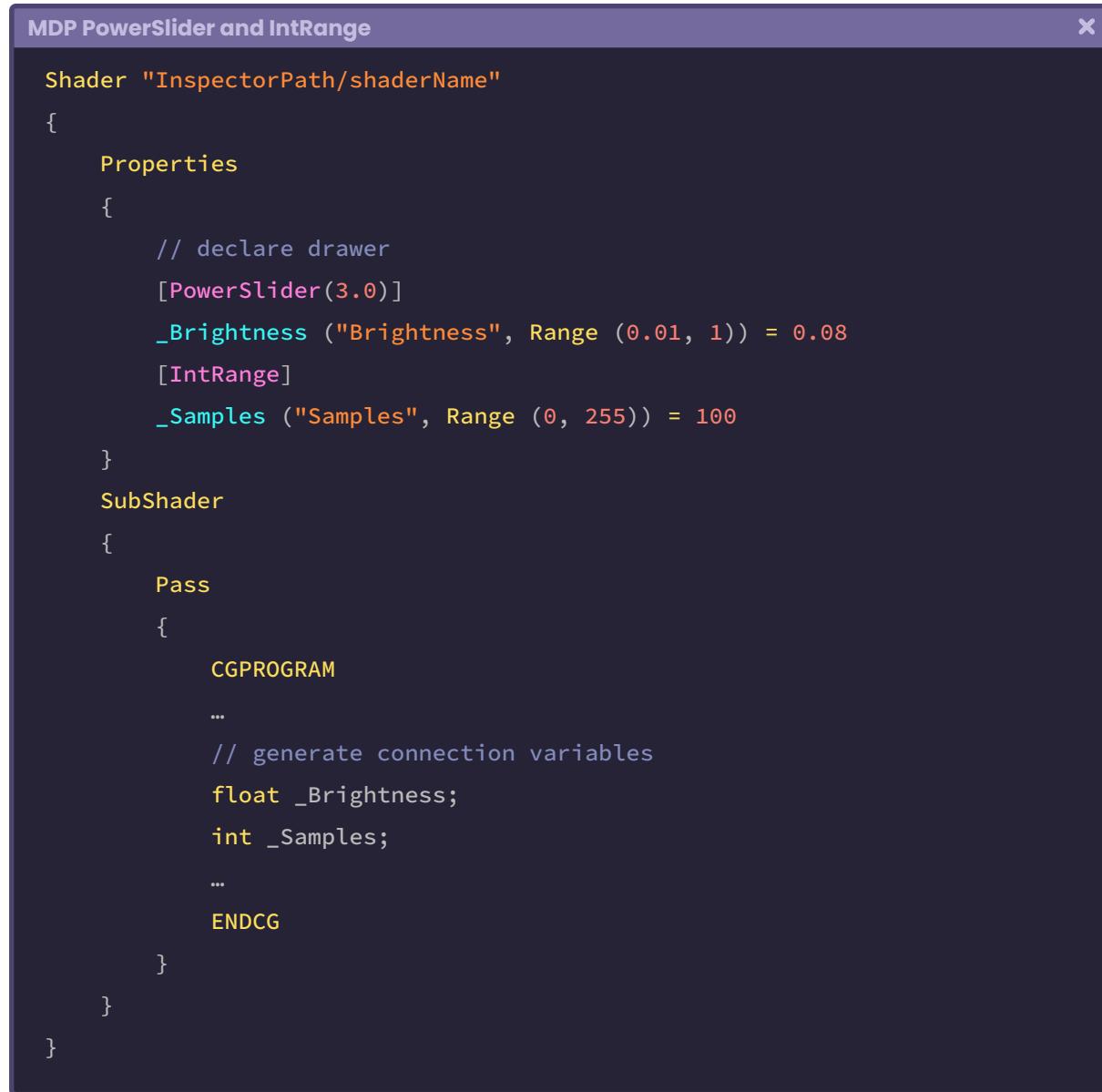
```
MDP PowerSlider and IntRange                                              ✕

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

On the other hand, we have the IntRange which, as its name says, adds a numerical range of integer values.

Its syntax is as follows:

```
MDP PowerSlider and IntRange                                              ✕

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

Note that, if we want to use these properties within our shader, they have to be declared within the CGPROGRAM in the same way as a conventional property. To understand how to use it, we will do the following operation:

**MDP 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
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM

            …
            // generate connection variables
            float _Brightness;
            int _Samples;

            …
            ENDCG
        }
    }
}
```
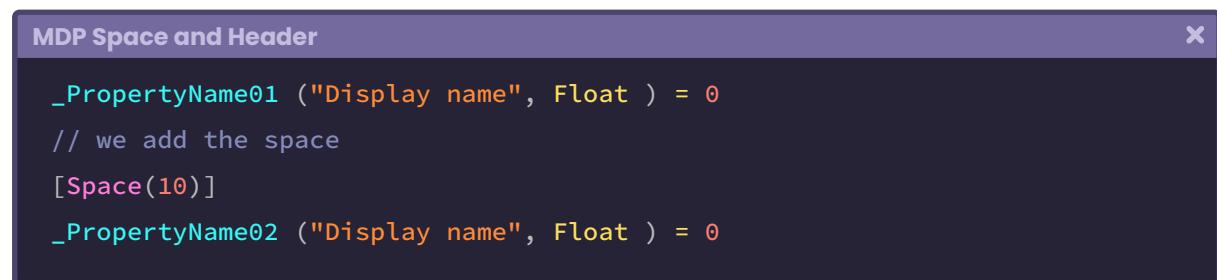
In the example above, we declare a PowerSlider called **_Brightness** and an IntRange called **_Samples**. Finally, using the same names, we generate our connection variables within the CGPROGRAM.

### 3.1.2. | MPD Space and Header.

Finally, these drawers help a lot in organization tasks.

"Space" allows us to add space between one property and another. If we wish that our properties appear separate in the material inspector, we can add space between them.

Its syntax is as follows:
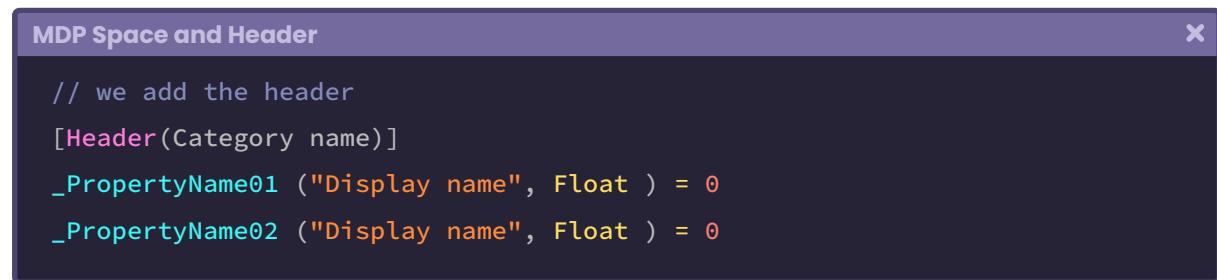
```
MDP Space and Header                                               ✖

_PropertyName01 ("Display name", Float ) = 0
// we add the space
[Space(10)]
_PropertyName02 ("Display name", Float ) = 0
```

In this case, we are adding ten points of space between the property **_PropertyName01** and the property **_PropertyName02**. Equally, as its name says, "Header" adds a header in the Unity Inspector. This is very useful when generating categories in our properties.
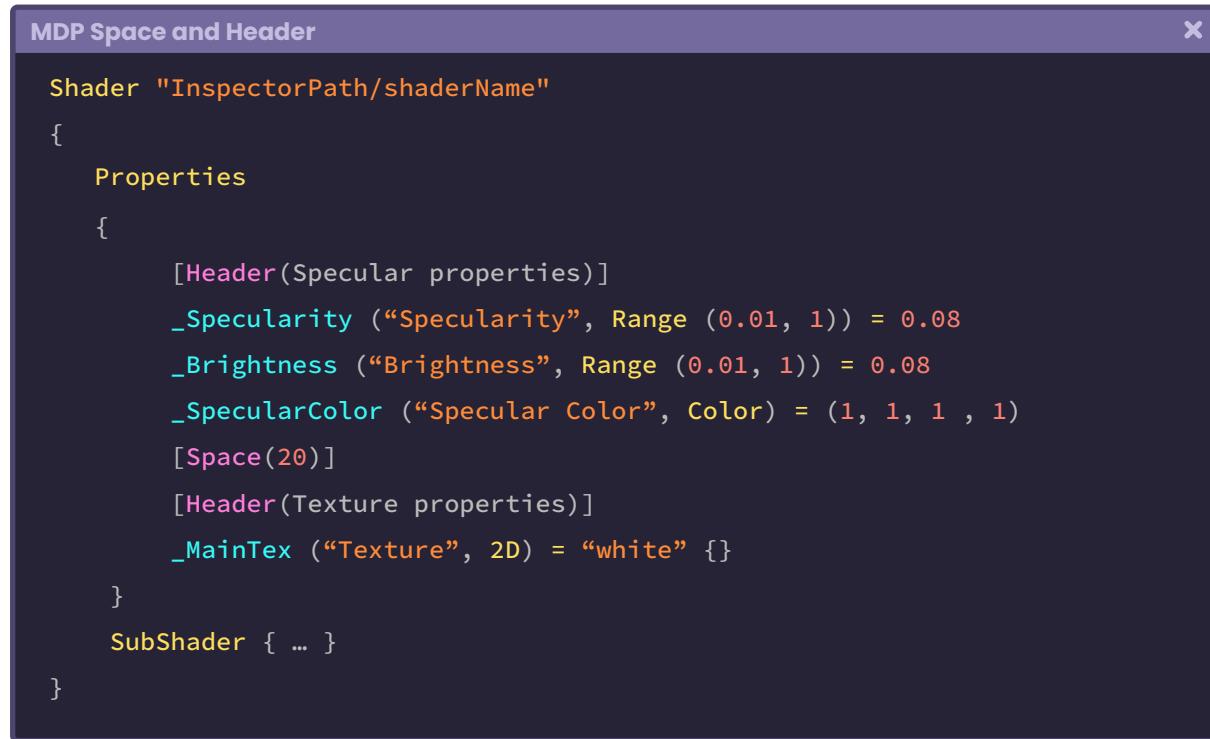
Its syntax is as follows:

```
MDP Space and Header                                               ✖

// we add the header
[Header(Category name)]
_PropertyName01 ("Display name", Float ) = 0
_PropertyName02 ("Display name", Float ) = 0
```

In this example, we have added a small header before starting our properties, which will be visible in the Inspector. To understand both properties, we will perform the following operation:

```
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, let's assume that the shader is going to run on hardware that supports **metal graph** API (iOS). For this, Unity will run the first SubShader that supports metal graphs and run it. When a SubShader is not supported, Unity will try to use the Fallback component that corresponds to a default shader, so the hardware can continue with its task without graphical errors.

```
ShaderLab SubShader                                              ✕

Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        // shader configuration here
    }
}
```

If we pay attention to our 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:

```
ShaderLab 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 we want to obtain. If we write a tag inside the SubShader it will affect all the passes that are included in the shader, but if we write it inside the Pass, it will only affect the selected pass.

The "Queue" is a Tag that we use more frequently, since it allows us to define how the surface of the object will look. By default, all surfaces are defined as "**opaque**", that is, they do not have transparency.

If we look at our USB_simple_color shader, we will find the following line of code inside the SubShader, which defines an opaque surface for our shader.

```
ShaderLab Tags                                                      ✖

 SubShader
 {
      Tags { "RenderType"="Opaque" }
      LOD 100

      Pass { … }
 }
```

### 3.1.5. | Queue Tags.

By default, this Tag does not appear graphically as a line of code. This is because it is compiled by default in the GPU, as it is directly related to the object processing order for each material.
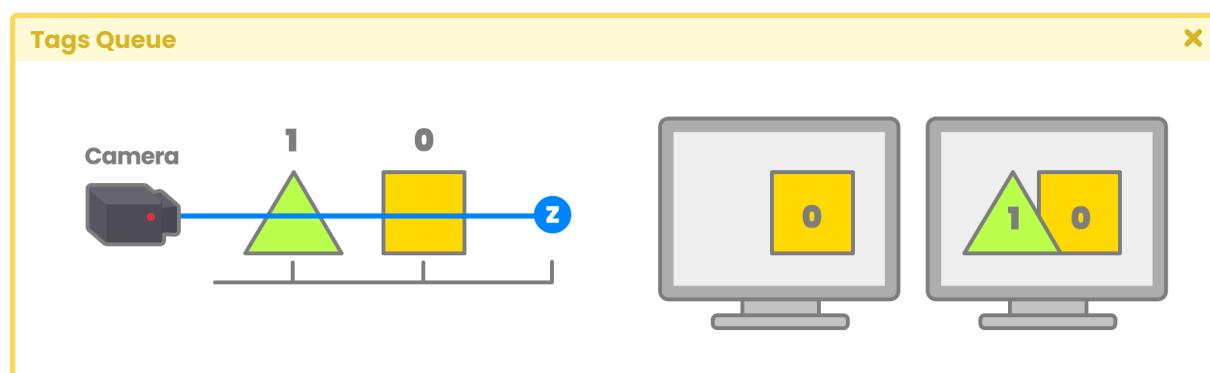
```
Tags { "Queue"="Geometry" }
```

This Tag has a close relationship between the camera and the GPU.

Every time we position an object in our scene, we pass its information to the GPU (e.g. position of vertices, normals, color, etc.). In the **Game View** case, it is the same, with the difference that the information that we send to the GPU corresponds to the object that is inside the frustum of the camera. Once the information is inside the GPU, we send this data to the VRAM, and we ask it to draw the object on our screen.

The process of drawing an object is called a "**draw call**". 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, if we have a shader with two passes inside it, then that material will only generate two *draw calls* on the GPU.

Now, how does the GPU draw these elements on the screen? In short, the GPU will first draw the objects found farthest from the camera and the elements closest to it will be drawn at the end. This calculation will be made concerning the distance between the object and the camera, following its "Z" axis.

**Tags Queue** ✕



*(Fig. 3.1.5a. As we can see from the picture, the triangle is drawn on the screen first due to it being farthest from the camera. In the end, the square is drawn, and together generate 2 draw calls)*

Unity has a processing queue called "**Render Queue**" which allows us to 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 we modify the **Queue** value in the shader, the default value of the *Render Queue* in the material will also be modified.

This property has order values ranging from 0 to 5000, where "0" corresponds to the farthest element and "5000" to the element closest to the camera. These order values have predefined groups, which 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 Tags** ✕

```
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, this instead introduces two control methods which are:

• Material order.
• And render order.

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

## 3.1.6. | Render Type Tags.

According to the official documentation in Unity,
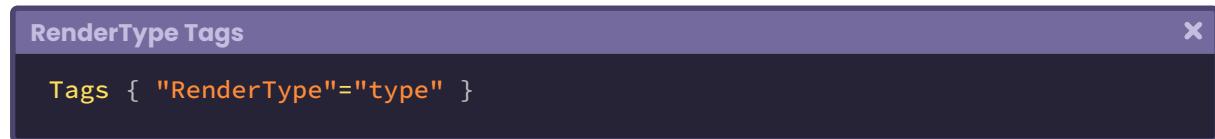
> *Use the RenderType tag to overwrite the behavior of a shader"*

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

To carry out its function, we need at least two shaders:

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

64

Its syntax is as follows:

```
RenderType Tags                                                    ✖

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

Like the Queue Tags, RenderType has different configurable values that vary depending on the task we want to perform. Among them, we can find.

- **Opaque.**                          Default.
- **Transparent.**
- **TransparentCutout.**
- **Background.**
- **Overlay.**
- **TreeOpaque.**
- **TreeTransparentCutout.**
- **TreeBillboard.**
- **Grass.**
- **GrassBillboard.**
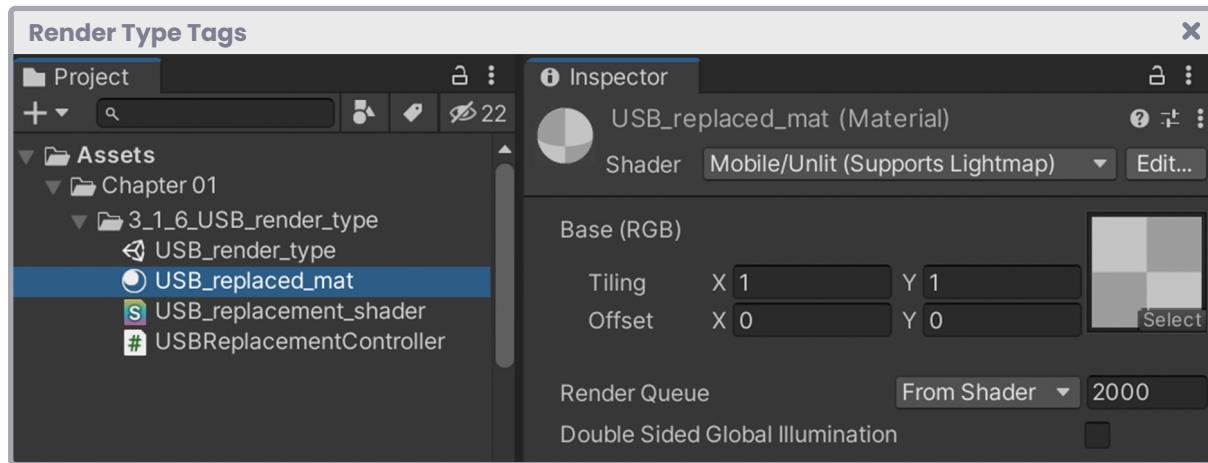
By default, the "Opaque" type is set every time we create a new shader. Likewise, most of the built-in shaders in Unity are assigned with this value, since they do not have a configuration for transparencies. However, we can freely change this category; everything will depend on the effect we apply to a match.

To understand the concept thoroughly, we will do the following. In our project,

1. We will make sure to create some 3D objects in the scene.
2. We will generate a C# script that we will call **USBReplacementController**.
3. Then we will create a shader that we will call **USB_replacement_shader**.
4. Finally, we will add a material that we will call **USB_replaced_mat**.

We will assign a shader over the material USB_replaced_mat dynamically using the Camera.SetReplacementShader. The material shader must have a Tag RenderType equal to the replacement shader to perform the function.

To exemplify, we will assign the Mobile/Unlit shader to USB_replaced_mat. This built-in shader has a Tag of type "RenderType" equal to "Opaque." Therefore, the shader USB_replacement_shader must match the same RenderType for the operation to be carried out.



*(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 a replacement one, as long as they have the same configuration in the RenderType.

```
RenderType Tags                                                ✕

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

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

    private void OnEnable()
    {
        if(m_replacementShader != null)
        {
```

*Continued on next page.*

```
                // the camera will replace all the shaders in the scene with
                the replacement one the "RenderType" configuration must match
                in both shader
                GetComponent<Camera>().SetReplacementShader(
                m_replacementShader, "RenderType");
            }
        }


        private void OnDisable()
        {
            // let's reset the default shader
            GetComponent<Camera>().ResetReplacementShader()
        }
    }
```

It is worth mentioning that we have defined the [**ExecuteInEditMode**] function over the class. This property will allow us to preview changes in edit mode.

We will use **USB_replacement_shader** as a replacement shader.

As we already know, every time we create a new shader, it configures its RenderType equal to "Opaque". Consequently, **USB_replacement_shader** may replace the Unlit shader we previously assigned to the material.

To preview the changes clearly, we will go to the fragment shader stage of **USB_ replacement_shader** and add a red color, which we will multiply by the output color.
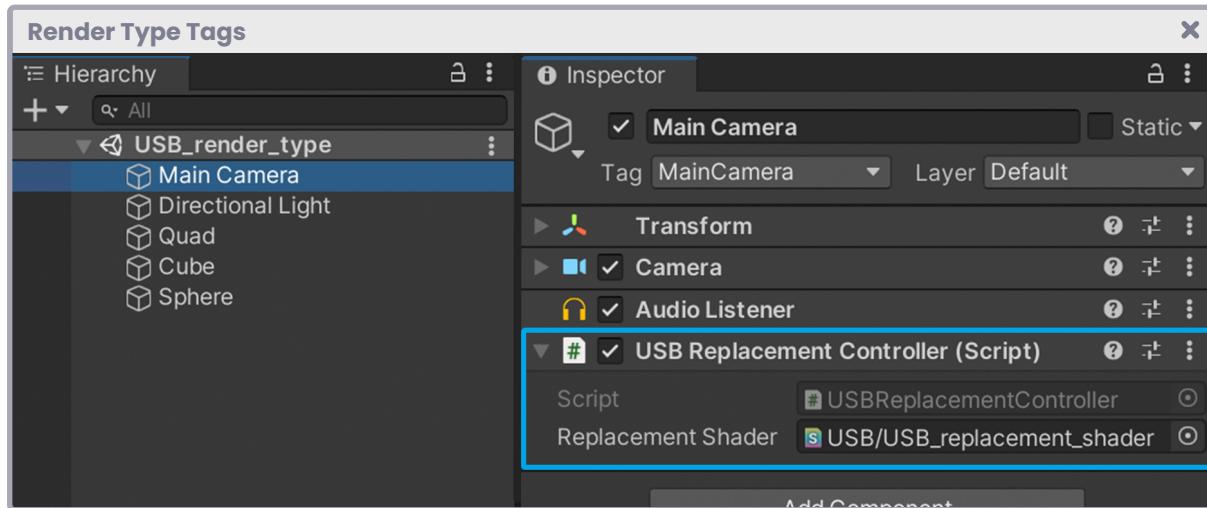
**RenderType Tags**                                                      ✕

```
  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;

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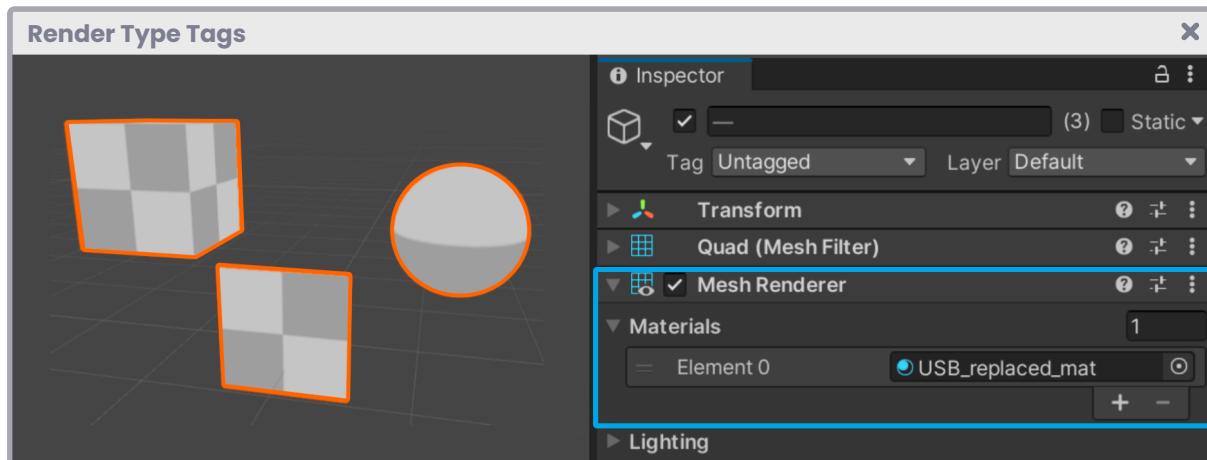
We must make sure to include **USB_replacement_shader** in the Shader type replacement variable found in the **USBReplacementController** script.
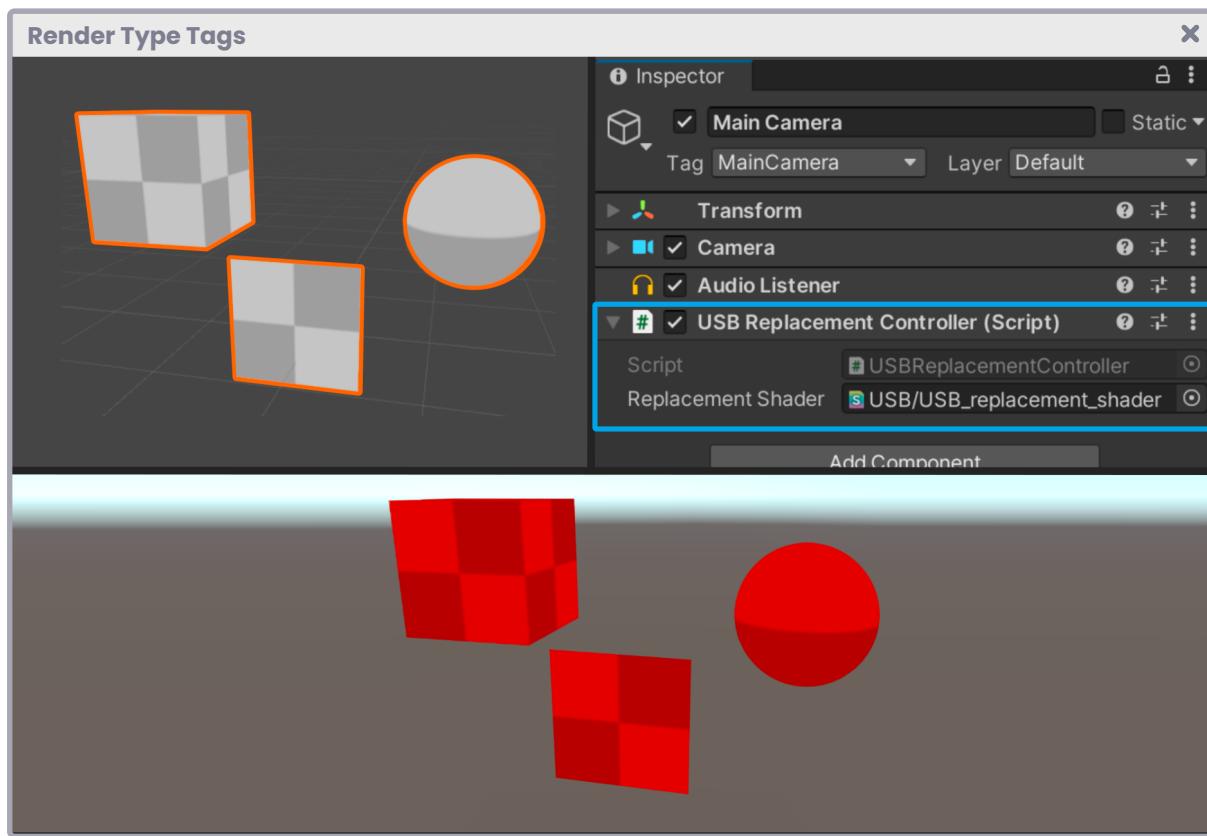


*(Fig. 3.1.6b. The USBReplacementController has been assigned to the camera)*

And in addition, those objects that we 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 included the OnEnable and OnDisable functions, if we activate or deactivate the script, we 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 (those pixels that have been processed in the fragment shader stage) with its depth. 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, this property is not written in our shader since it is an optional function and is used mainly when we work with transparent objects, e.g., when we must draw a pixel with a low opacity level in front of another.

Its default value is "Blend Off", but we can activate it to generate different types of *Blending*, like those that appear in photoshop.

Its syntax is as follows:

**SubShader Blending** ✕

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

"**Blend**" is a function that requires two values called "factors" for its operation and based on an equation it will be the final color that we will obtain on-screen. According to the official documentation in Unity, the equation that defines the value of the Blending is as follows:

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

To understand this operation, we 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 our shader, SrcValue overwrites DstValue. However, if we 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 vectors of three dimensions that vary depending on their configuration. Their main function is to modify the SrcValue and DstValue values to achieve interesting effects.

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

- **Off**, disables Blending options.
- **One**, (1, 1, 1).
- **Zero**, (0, 0, 0).
- **SrcColor** is equal to the RGB values of the *SrcValue*.
- **SrcAlpha** is equal to 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** is equal to the RGB values of the *DstValue*.
- **DstAlpha** is equal to 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 we process the RGB color of a pixel, but it is done in an independent process because it is not used frequently. Likewise, by not performing this process, the writing on the render target is optimized.

Let's exemplify the above explanation as follows.

Let's say we have an RGB color pixel with the values $[0.5_R, 0.45_G, 0.35_B]$. This color has been processed by the **fragment shader stage,** therefore it corresponds to the *DstValue*. Now, we multiply this value by the "*SrcFactor* **One**" which equals $[1, 1, 1]$. Every number multiplied by "1" results in the same value, therefore, the result between the *SrcFactor* and the *DstValue* is the same as its initial value.

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

"OP" refers to the operation that we are going to perform. By default, it is set to "Add".

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

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

DstFactor $[0.5_R, 0.45_G, 0.35_B]$ * 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.

$B = [0.5_R, 0.45_G, 0.35_B] + [0.25_R, 0.20_G, 0.12_B]$.
$B = [0.75_R, 0.65_G, 0.47_B]$

If we want to activate *Blending* in our shader, we 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 we want to use *Blending* in our shader, it will be necessary to add and modify the "*Render Queue*". As we already know, the default value of the "*Queue*" tag is "*Geometry*", which means that our object will appear opaque. If we want our object to look transparent, then we 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 our Blending is through the dependency "UnityEngine.Rendering. BlendMode". This line of code allows us to change, from the inspector, the Blending of an object in the material. To set it up, first we must add the "*Toggle Enum*" to our properties and then declare both the SrcFactor and the DstFactor.

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
    }
```

*Continued on next page.*

73

```
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
        Blend [_SrcBlend] [_DstBlend]
    }
}
```

Blending options can be written in different fields: within the **SubShader** field or the **Pass** field, the position will depend on the number of passes and the final result that we need.

## 3.1.8. | SubShader AlphaToMask.

There are some types of Blending that are very easy to control, e.g., the "SrcAlpha OneMinusSrcAlpha Blend", which adds a transparent effect with the Alpha channel included, but there are other cases where Blending is not capable of generating transparency for our shader. In this case, the "AlphaToMask" property is used, which applies a mask over the Alpha channel and is a technique compatible with both Built-in RP and Scriptable RP.

Unlike Blending, a mask can only assign the values "one or zero" to the Alpha channel, what does this mean? While the Blending can generate different levels of transparency; levels from "0.0f" to "1.0f", AlphaToMask can only generate integers. This translates to a harsher type of transparency, which works in specific cases, e.g., it is very useful for vegetation in general and to create space portal effects.

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

To activate this command, we can declare it in both the SubShader field and the pass field. It only has two values: "On and Off", and it 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. We just add *AlphaToMask* and automatically the fourth color channel "A" acquires the qualities of the mask in our program.

## 3.1.9. | SubShader ColorMask.

This command allows our GPU to limit itself to writing a selected color channel and is compatible with both Built-in RP and Scriptable RP.

When we create a shader, by default the GPU writes all channels corresponding to the color (RGBA), however, in some cases, we may want to show only some color channels (e.g. red channel or "R" of an effect).

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

As we already know, the acronym RGBA stands for Red, Green, Blue and Alpha, therefore, if we configure our mask with the value "G" we will only show the green channel as color output. This ShaderLab command is very simple to use, and we can utilize 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, we must first know how **Z-Buffer** (also known as **Depth Buffer**) and **Depth Testing** work.

Before starting, we must consider that the 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 we 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 permits 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.
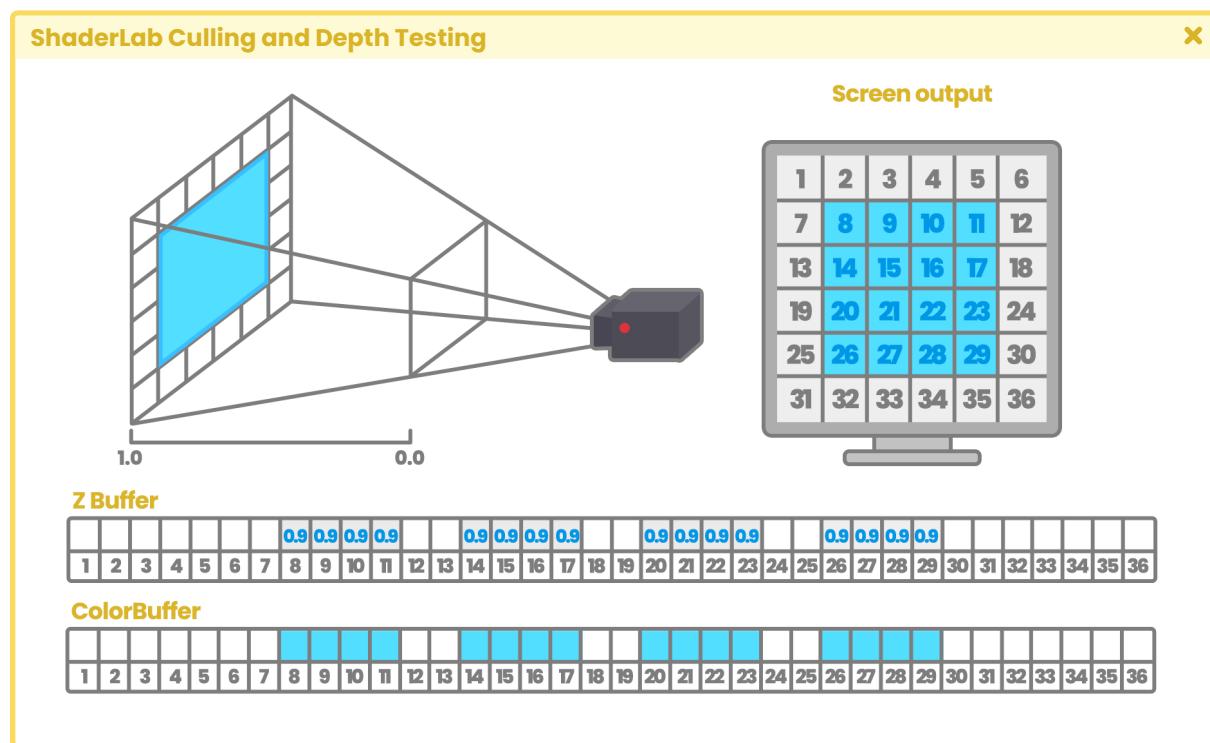
To understand the concept, let's assume that we have a camera and some primitives in our scene, all positioned on the "Z" space axis. Now, why on the z-axis? The "Z" in Z-Buffer comes from the fact that the Z value measures the distance between the camera and an

76

object on the "Z" axis of space, while the "X and Y" values measure horizontal and vertical displacement on the screen.

The word "Buffer" refers to a "memory space" in which data will be temporarily stored, therefore, Z-Buffer refers to the depth values between the objects in our scene and the camera, which are assigned to each pixel.

For example, we are going to draw a screen with a total of 36 pixels.
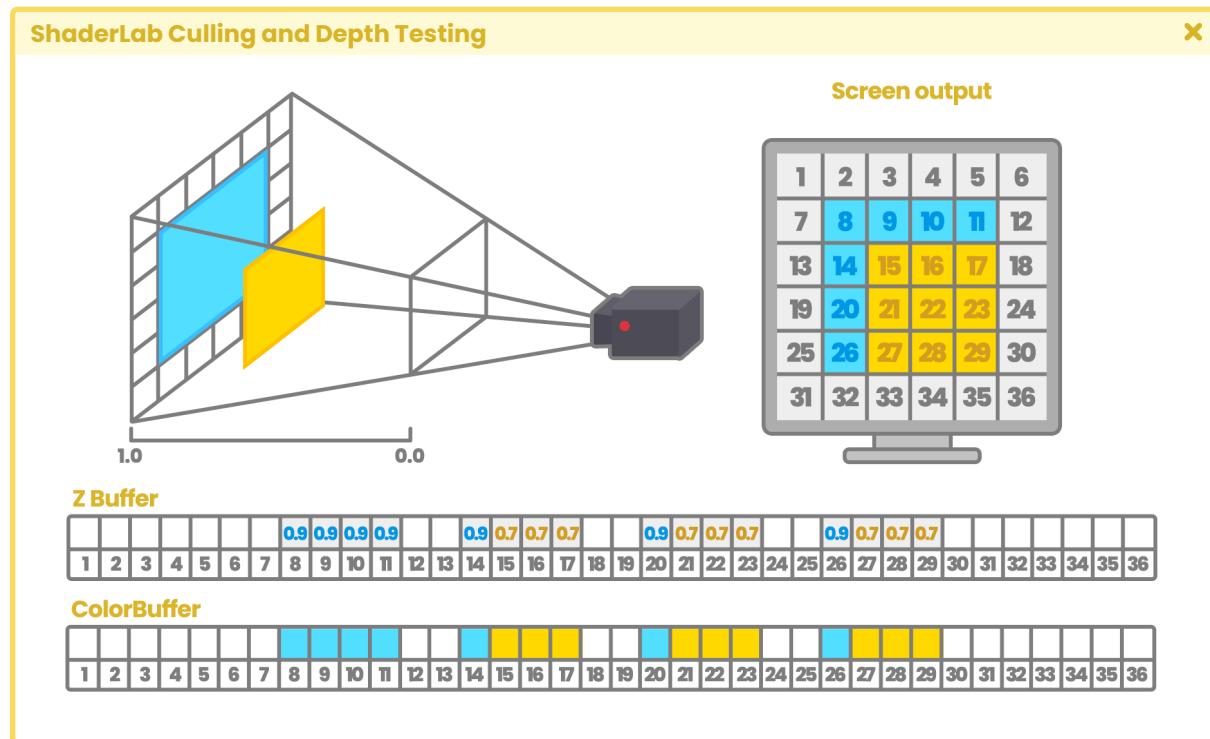
Every time we position an object in our scene, that object occupies a certain pixel area on the screen. So, let's assume that we want to position a green square in the scene. Given its nature, it will occupy from pixel 8 to 29, therefore, all the pixels inside this area are activated and painted green, likewise, this 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).

We position a new square in the scene, this time in red and closer to the camera. To differentiate it from the previous one, we will make this square smaller, occupying from pixel 15 to 29. As we can see, this area is already occupied by the information from the initial square, so what happens here? Since the red 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)

In the case of adding a new element to the scene that is even closer to the camera, this process will be repeated in the same way. In conclusion, the values of the Z-Buffer and Color Buffer will be overwritten by the object that is closest to the camera.

One way to generate attractive visual effects is by modifying the *Z-Buffer* values. For this, we will talk about three options that are included in Unity: **Cull, ZWrite,** and **ZTest**.

Like Tags, culling and depth testing options can be written in different fields: within the SubShader field or the Pass field. The position will depend on the result we want to achieve and the number of passes we want to work with.

To understand this concept, let's assume that we want to create a shader to represent the surface of a diamond. For this, we will need two passes: The first we will use for the background color of the diamond, and the second for the shine of its surface. In this hypothetical case, since we require two passes that fulfil different functions, 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 the 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, we can activate the inner faces 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, by default.
- **Cull Front**    The front faces of the object are rendered.

This command has three values which are: **Back**, **Front** and **Off**. By default, "Back" is active, however, generally, the line of code associated with culling is not visible in the shader for optimization purposes. If we want to modify the culling options, we must add the word "**Cull"** followed by the mode we want to use.

```
ShaderLab Cull                                              ✕

Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        // Cull Off
        // Cull Front
        Cull Back
    }
}
```

We can also dynamically configure the Culling options 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 helpful option occurs through the semantics **SV_IsFrontFace**, which allows us to project different colors and textures on both mesh faces. To do so, we 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;
 }
```
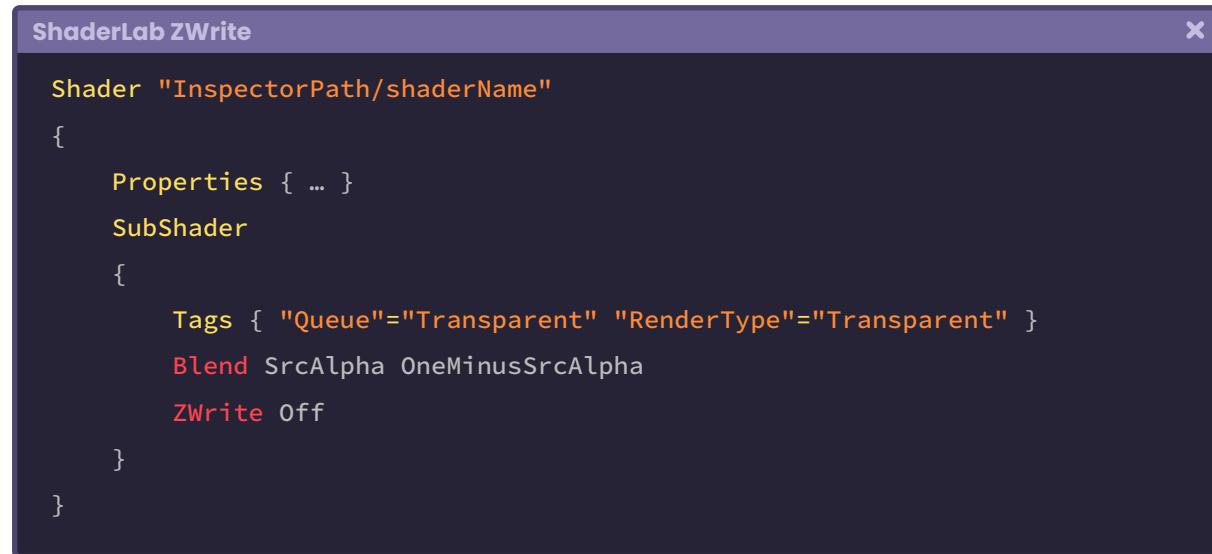
It is worth mentioning that this option only works when the "Cull" command has been previously set to "Off" in the shader.

### 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 us to ignore or respect the depth distance between the camera and an object. ZWrite has two values, which are: On and Off, where "On" corresponds to its default value. We generally use this command when working with transparencies, e.g., when we activate the Blending options.

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

Why should we disable the *Z-Buffer* when working with transparencies? Mainly because of the translucent pixel overlay (Z-fighting). When we work 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 we move the camera in the scene. To fix this problem, we must simply deactivate the *Z-Buffer* by turning the **ZWrite** command to "**Off**" as the following example shows:
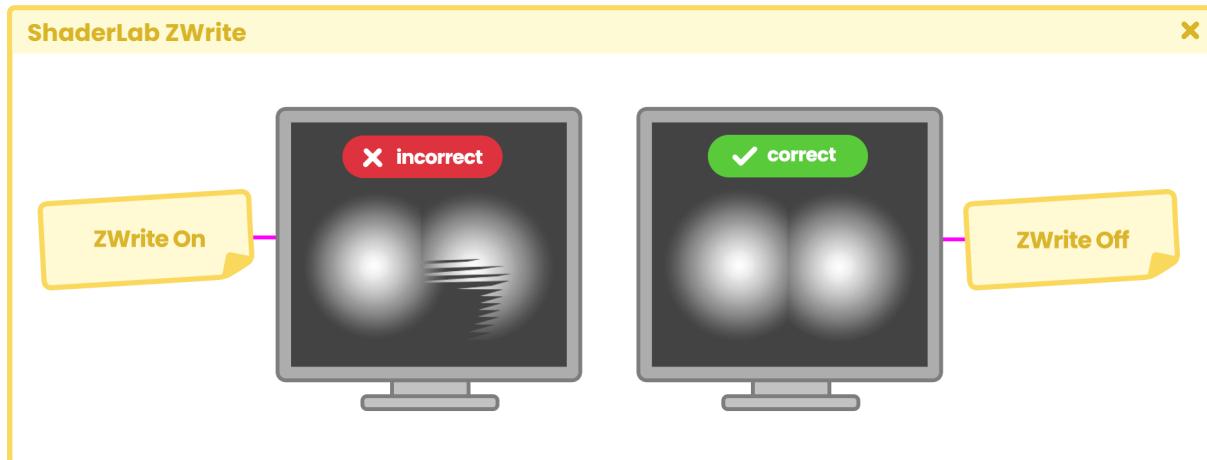
**ShaderLab ZWrite**    ✖

```
Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off
    }
}
```

The Z-fighting occurs when we have two or more objects at 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 rendering 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.

To correct this issue, we simply need to disable the Z-Buffer using the "ZWrite off" command.

## 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 are:

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

Which they correspond to a comparison operation.

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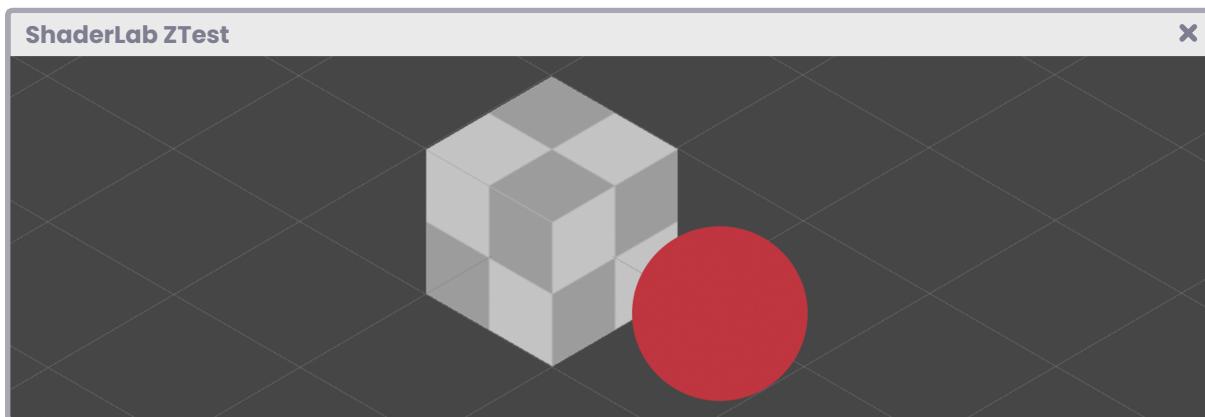
**ZTest GEqual:** ($\geq$) Draws the objects behind or at the same distance. Does not draw objects in front of the shader object.

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

**ZTest NotEqual:** ($!=$) Draws objects that are not at the same distance. Does not draw objects that are the same distance from the shader object.
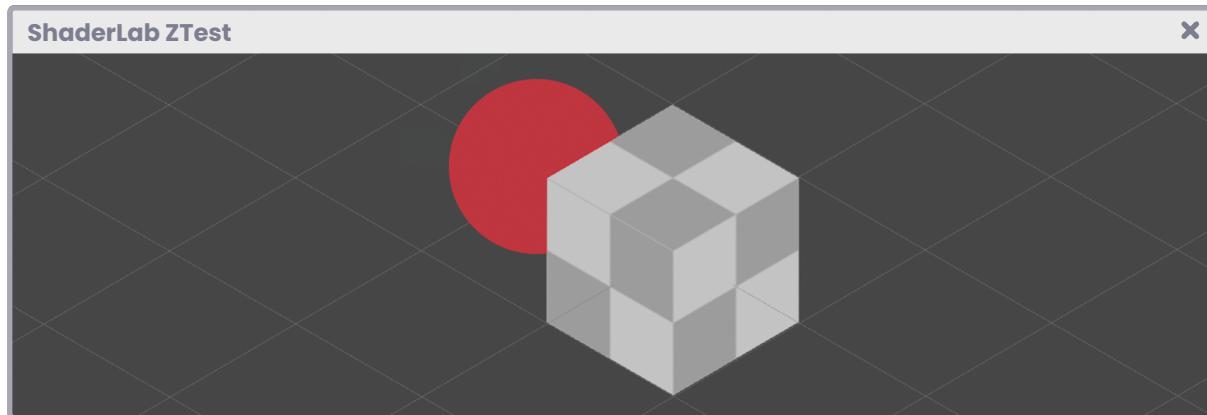
**ZTest Always:** Draws all pixels, regardless of the distance of the objects relative to the camera.

To understand this command, we will do the following exercise: Let's suppose we have two objects in our 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.
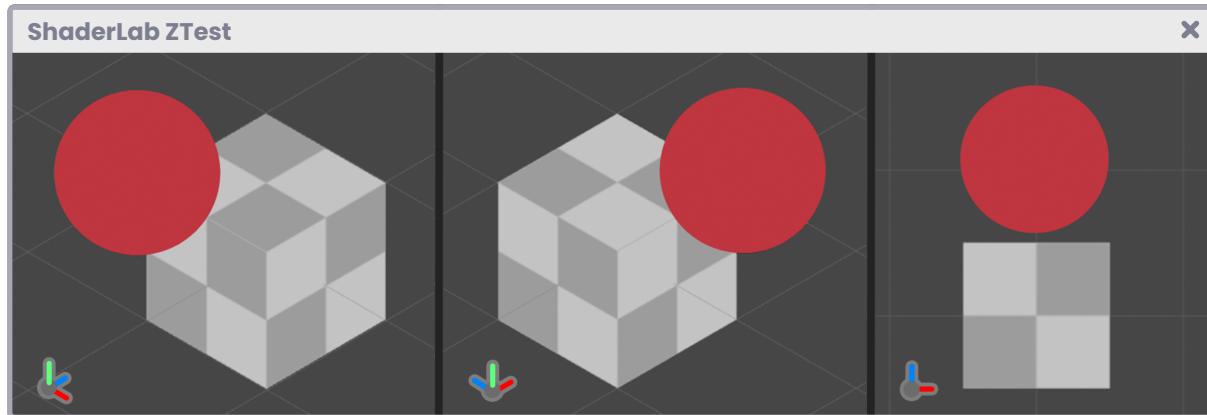


*(Fig. 3.2.3a)*

If we position the *Sphere* behind the *Cube* then again, 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 we activated ZTest Always? In this case, Depth Testing would not be done, therefore, all pixels would appear at the same depth on screen.



*(Fig. 3.2.3c)*

Its syntax is the following:
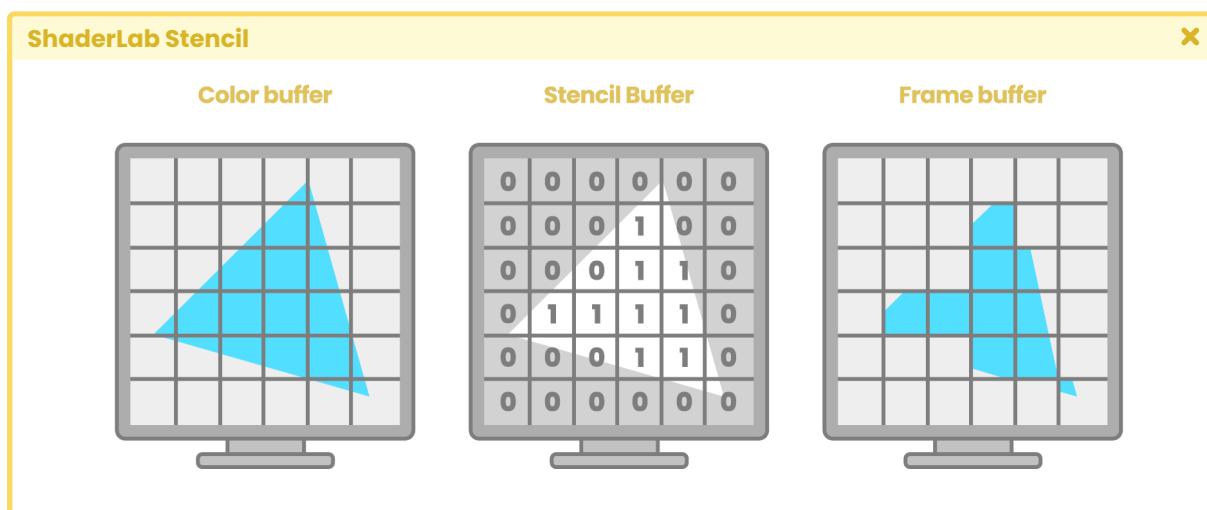
```
Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
         ZTest LEqual
    }
}
```

## 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.* "

To understand the above description, we must take into consideration that the Stencil Buffer itself is a "texture" that must be created. For this, it stores an integer value from 0 to 255 for each pixel in the Frame Buffer.
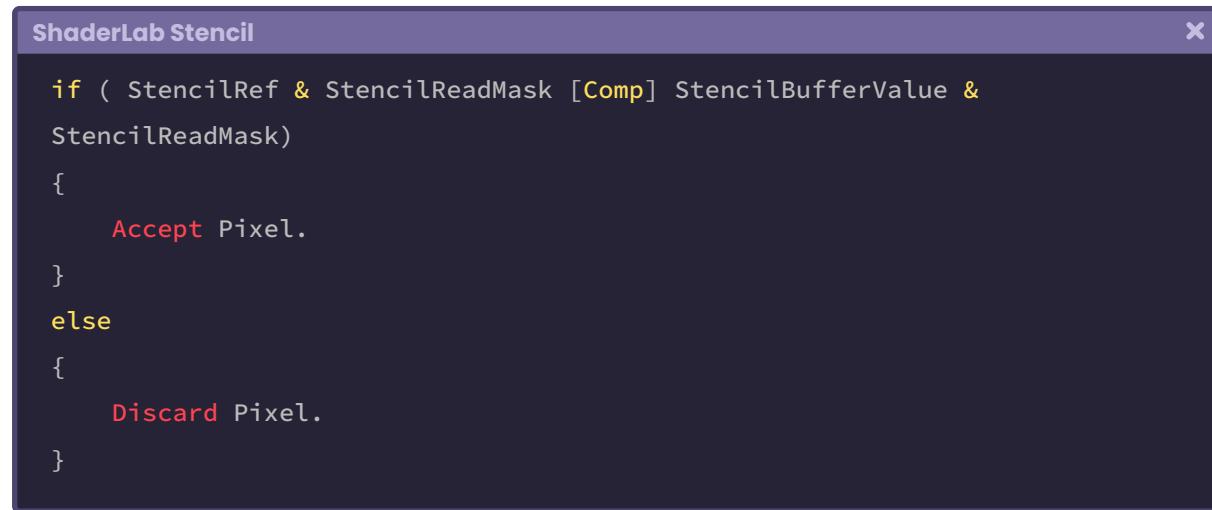


*(Fig. 3.2.4a)*

As we already know, when we position objects in our scene, their information is sent to the **vertex shader stage** (e.g. vertex position). Within 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 inside the frustum of the camera.

When this information has been processed correctly, it is sent to the **rasterizer** which allows us to project in pixels the coordinates of our objects in the scene, however, before reaching this point it goes through a previous stage of processing called *Culling* and *Depth testing*. In this stage various processes occur that we can manipulate within our shader, among them are **Cull, ZWrite, ZTest** and **Stencil**.

Basically what the *Stencil Buffer* does is activate the *Stencil Test*, which allows the discarding of "fragments" (pixels) so that they are not processed in the **fragment shader stage**, thus generating a mask effect in our shader. The function performed by the *Stencil Test* has the following syntax:
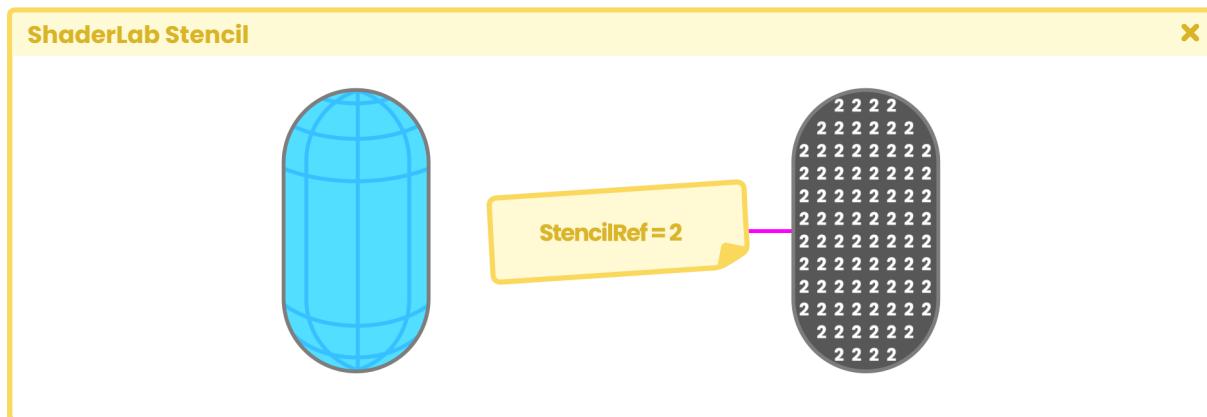
```
ShaderLab Stencil                                              ✕

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

"**StencilRef**" is the value that we are going to pass to the Stencil Buffer as a reference, 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*.

**86**
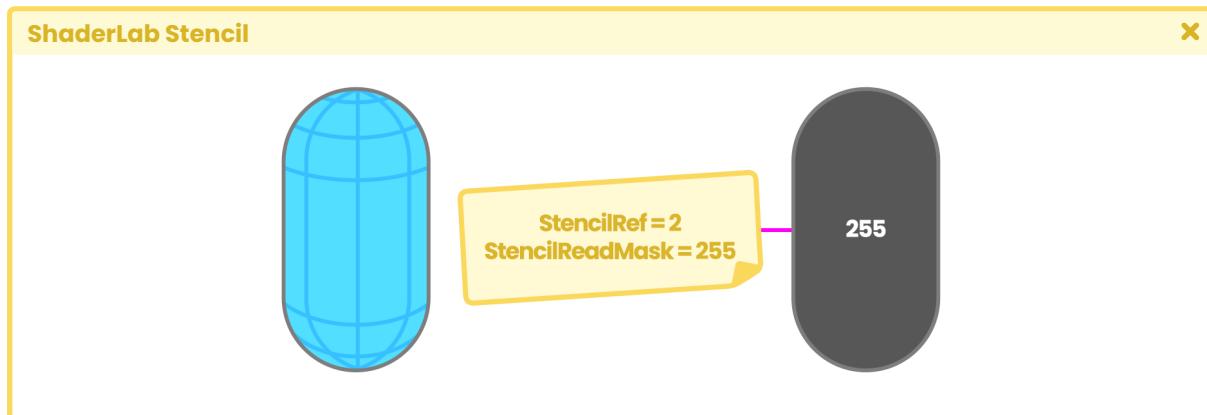
For example, we are going to make the StencilRef value 2.



*(Fig. 3.2.4b)*
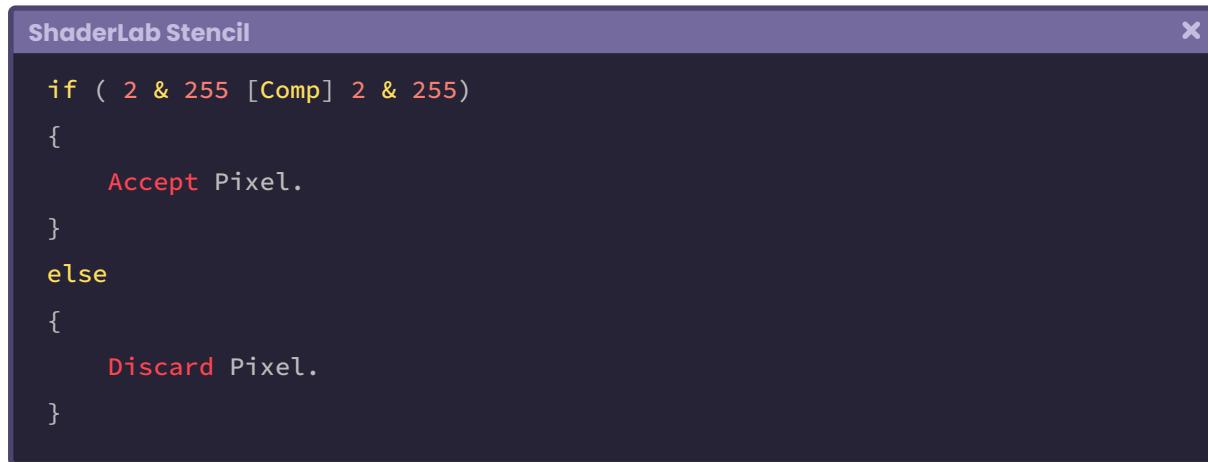
In the example above, all the pixels covering the capsule area have been marked with the value of the *StencilRef*, therefore now the Stencil Buffer equals 2.

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**" refers to 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, we can find the following predefined operators:
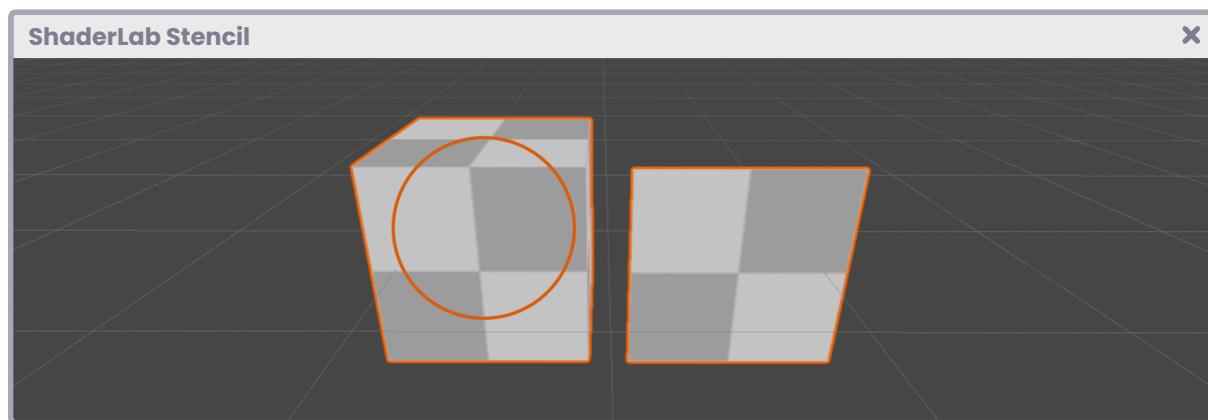
- **Comp Never:**       The operation will always deliver false.
- **Comp Less:**        <.
- **Comp Equal:**       ==.
- **Comp LEqual:**      ≤.
- **Comp Greater:**     >.
- **Comp NotEqual:**    !=.
- **Comp GEqual:**      ≥.
- **Comp Always:**      The operation will always deliver true.

We need at least two shaders to use the Stencil Buffer: One for the mask and one for the masked object.

Let's suppose we have three objects in our scene: A **Cube**, a **Sphere** and a **square**. The Sphere is inside the Cube, and we want to use the square as a "mask" to hide the Cube so that the Sphere inside can be seen. To create this example mask, we will create a shader called **USB_stencil_ref** and its syntax would be the following:

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

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



*(Fig. 3.2.4d)*

Let's analyze the above. The first thing we did was configure "Queue" equals "Geometry minus one". Geometry defaults to 2000, therefore, Geometry minus 1 equals 1999, this will process our square (mask), to which we will apply this shader, first in the Z-Buffer. However, as we know, Unity by default processes objects according to their position in the scene concerning the camera, therefore if we want to disable this function we must set the property "**ZWrite** to **Off**".

Then we set "**ColorMask** to zero" so that the mask pixels are discarded in the Frame Buffer and appear transparent.
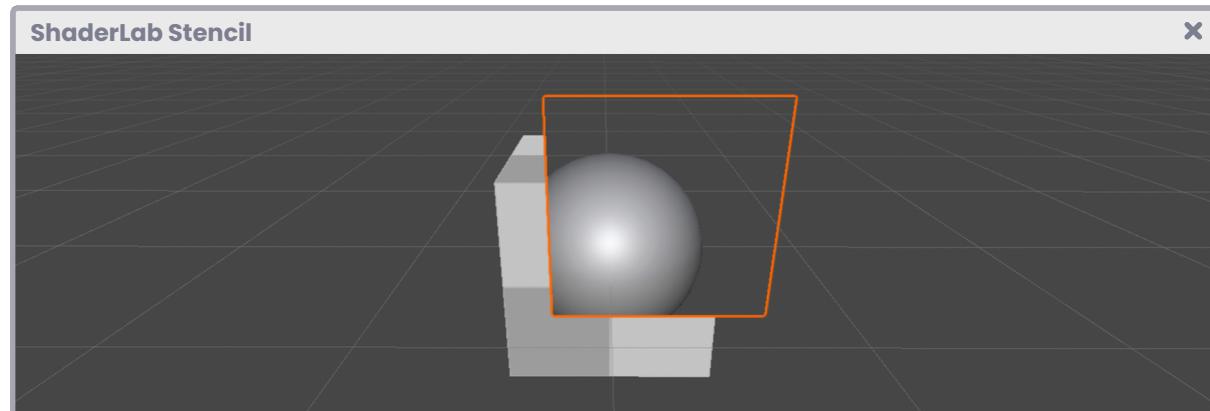
At this point, our square still does not work as a mask, therefore what we must do next is add the command "**Stencil**" so that it functions as such.

"**Ref 2**" (StencilRef), our reference value, is compared on the GPU with the current content of the Stencil Buffer using the operation defined in the "comparison operation".

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

Now we are going to analyze the object that we want to mask. We will create another shader called **USB_stencil_value** to illustrate this. Its syntax is as follows:

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

SubShader Stencil



ShaderLab Stencil

*(Fig. 3.2.4e)*

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

Then, in the comparison operation, we assign "Comp NotEqual". This states that the Cube area exposed around the square will be rendered because the Stencil Test passes (no comparison or true).

On the other hand, for the square area, being equal (Equal) the Stencil Test will not pass, and the pixels will be discarded. "Pass Keep" means that the Cube maintains the current content of the Stencil Buffer.

If we want to work with more than one mask, we can give a Ref property number different to "2".

### 3.2.5. | ShaderLab Pass.

The third component in our shader corresponds to the passes (Pass).

There can be multiple passes within a shader, however, by default Unity adds only one inside the SubShader field.

If we look at our **USB_simple_color** shader, we 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"

              Continued on next page.
```

```
    struct appdata { … };

    struct v2f { … };

    sampler2D _MainTex;
    float4 _MainTex;

    v2f vert (appdata v) { … }

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

A Pass refers to a Render Pass literally. For those who have worked with rendering in 3D software (e.g., Maya or Blender), this concept will be easier to understand since when an image is being processed, it can generate different layers or passes separately (e.g., color Pass, light Pass, occlusion Pass, etc.) and thus obtain a separate composition in different layers.

Each **Pass** renders one object at a time, that is, if we have two passes in our shader, the object will be rendered twice on the GPU and the equivalent of that would be two draw calls.

A Pass is equal to a draw call which is why we must use the least amount of passes possible, otherwise, we could generate a significant graphic load.

**ShaderLab Pass** ✕

```
Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        Pass
        {
            // first default pass
        }
```

*Continued on next page.*

```
        Pass                                    93
        {
            // second additional pass
        }
    }
}
```

## 3.2.6. | CGPROGRAM / ENDCG.

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

By default, we can see that Unity has included the word CG in our 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, our shader will appear with the declaration of HLSLPROGRAM instead of CGPROGRAM since HLSL is currently the official graphics programming language (in fact, Shader Graph is based on it).

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

If we want to update the program to HLSL, we will have to change some settings and add some routes to our 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 our shader to compile will be written inside the CGPROGRAM and ENDCG field. Those functions outside this fragment will be taken by Unity as ShaderLab properties, and if they do not correspond to this language, then our program will not compile.

**CGPROGRAM / ENDCG** ✕

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

## 3.2.7. | Data Types.

Before continuing to define the properties and functions of our shader, we must look at data types because there is a small difference between Cg and HLSL.

When we create a default shader in current versions of Unity, we 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 the "fixed" data type, so, if we work with this language, we will have to replace all the variables and vectors of this type either with half or float.

"**Float**" (Cg and HLSL) is a high-precision data type, it is 32 bit and generally used in calculating positions in *world-space*, texture coordinates (UV), or scalar calculations involving complex functions such as trigonometry or exponentiation.

"**Half**" (Cg and HLSL) is half-precision, is 16 bit and is mostly used in the calculation of low magnitude vectors, directions, *object-space* positions, and high dynamic range colors.

"**Fixed**" (Cg) is low precision, it is only 11 bits and is mainly used in the calculation of simple operations (e.g. basic color storage).

A question that commonly arises in the vector use is, what would happen if we only use a floating data type (Float) for all our variables? In practice this is possible, however, we 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 we can optimize our program, reducing the graphic load on the GPU.

Other widely used data types that we can find in both languages are "Int, sampler2D and samplerCube".

**95**

"**Sampler**" refers to the sampling state of a texture. Within this type of data, we can store a texture and its UV coordinates.

Generally, when we want to work with textures in our shader, we must use the "Texture2D" type to store the texture and create a "**SamplerState**" to sample. The data type "**sampler**" allows us to store both the texture and the sampling status in a single variable. To understand in detail the function of a sampler, we will 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 "**sampler**".

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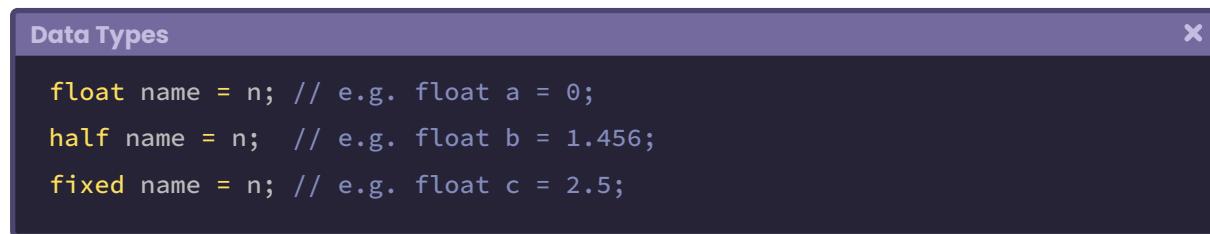
```
// declare the sampler for _MainTex
sampler2D _MainTex;
// go to the fragment shader stage
half4 frag(v2f i) : SV_Target
{
    // sampler the texture in UV coordinates using the function tex2D().
    half4 col = tex2D(_MainTex, i.uv)
    // return the color of the texture.
    retu rn col;
}
```

Both examples return the same value that corresponds to the texture that we generate in our properties and will be assigned from the Unity Inspector.

We can use scalar values, vectors and matrices in our program.

Scalar values are those that return a real number, either an integer or decimals (e.g. 1, 0.4, 4, etc.) and to declare them in our shader we must first add the "data type", then the "scalar value name" and finally initialize its default value.
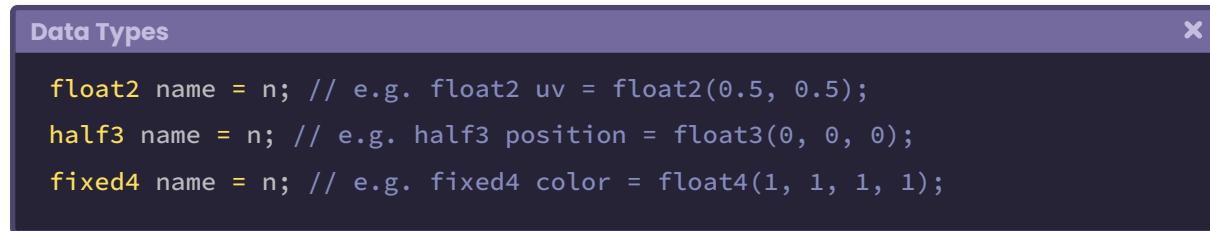
Its syntax is as follows:

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

Vectors are those that return a value with more than one dimension (e.g. XYZW) and to declare them in our shader we must first add the "data type", then "dimension number", then "vector name" and finally initialize its default value.

Its syntax is as follows:

```
Data Types                                              ✕
float2 name = n; // e.g. float2 uv = float2(0.5, 0.5);
half3 name = n; // e.g. half3 position = float3(0, 0, 0);
fixed4 name = n; // e.g. fixed4 color = float4(1, 1, 1, 1);
```

Matrices are those that have values stored in rows and columns, they have more than one dimension and are used mainly for shearing, rotation and change of vertex position.

To declare a matrix in our shader we must first add the "data type", then the "dimension quantity multiplied by itself", then the "matrix name" and finally "initialize its default values" taking into consideration that the coordinates Xx, Yy & Zz will be equal to 1.

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 Unity shaders, you may well not fully understand what has been explained. Don't worry, later in this chapter we will review all these parameters in detail and see how the fragment shader stage works.

## 3.2.8. | Cg / HLSL Pragmas.

In our shader, we can find at least three default pragmas. These are processor directives and are included in Cg or HLSL. Their function is to help our shader recognize and compile certain functions that could not otherwise be recognized as such.

The **#pragma vertex vert** allows the **vertex shader** stage called **vert** to be compiled to the GPU as a vertex shader. This is essential since without this line of code the GPU will not

be able to recognize the "**vert**" function as the *vertex shader stage*, therefore, we will not be able to use information from our objects and neither pass information to the *fragment shader stage* to be projected onto the screen.

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

```
Cg / HLSL Pragmas                                              ✕

// allow us 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 serves the same function as the pragma vertex, with the difference that it allows the fragment shader stage called "frag" to compile in the code as a fragment shader.

```
Cg / HLSL Pragmas                                              ✕

// allow us 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;
}
```

99

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

### 3.2.9. | Cg / HLSL Include.

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

If we look at our USB_simple_color shader, we will find the following directives that have been declared within the pass:

- UNITY_FOG_COORDS(texcoordindex).
- UnityObjectToClipPos(inputVertex) .
- TRANSFORM_TEX(tex, name).
- UNITY_TRANSFER_FOG(outputStruct, clipSpacePos).
- UNITY_APPLY_FOG(inputCoords, colorOutput).

All these functions belong to "**UnityCG.cginc**". If we remove this directive, our shader will not be able to compile because it will not have its reference.

Another defined function that we can find in UnityCG.cginc is UNITY_PI, which equals 3.14159265359f. The latter is not included in our default shader because it is used only in specific cases (e.g. when calculating a triangle or Sphere). In case we want to review the variables and functions that come in our .cginc files, we can 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, we can create our directives using the extension ".cginc", to do this we must simply create a new document, save it with that extension and then start defining our variables and functions. Later in this book, we will start using some custom directives to work with lighting, shadows, and volumes.

## 3.3.0. | Cg / HLSL vertex input & vertex output.

A data type that we will use frequently in the creation of our shaders is "**struct**". For those who know the C language, 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. We will use structs to define both inputs and outputs in our shader. Its syntax is as follows:

```
Cg / HLSL vertex input & vertex output                                    ✖

struct name
{
    vector[n] name : SEMANTIC[n];
};
```

First, we declare the **struct** and then its **name**. Then we store the vector semantics inside the *struct* field for later use. The struct "*name*" corresponds to the name of the structure, the "*vector*" corresponds to the type of vector that we will use (e.g. float2, half4) to assign a semantic. Finally, "SEMANTIC" corresponds to the semantics that we will pass as input or output.

By default, Unity adds two **struct** functions, which are: **appdata** and **v2f**. *Appdata* corresponds to the "**vertex input**" and *v2f* refers to the "**vertex output**".

The *vertex input* will be the place where we store our object properties (e.g. position of vertices, normals, etc.) as an "entrance" to take them to the "vertex shader stage". Whereas, *vertex output* will be where we store the rasterized properties to take them to the "fragment shader stage".

We can think of semantics as "access properties" of an object. According to the official Microsoft documentation:

> *A semantic is a chain connected to a shader input or output that transmits usage information of the intended use of a parameter.*

We will exemplify using the POSITION[n] semantic.

In previous pages, we have talked about the properties of a primitive. As we already know, a primitive has its vertex position, tangents, normals, UV coordinates and color in the vertices. A semantic allows individual access to these properties, that is, if we declare a four-dimensional vector, and we pass it the POSITION[n] semantic then that vector will contain the primitive vertices position. Suppose we declare the following vector:

**Cg / HLSL vertex input & vertex output**                                                ✕

```
float4 pos : POSITION;
```

It means that inside the four-dimensional vector called "pos," we store the object vertices position in object-space.

The most common semantics that we use are:

- **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;
```

*Continued on next page.*

102

```
    float2 texCoord : TEXCOORD0;
    float3 tangentWorld : TEXCOORD1;
    float3 binormalWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 vertColor: COLOR0;
};
```

**TEXCOORD[n]** allows access to the UV coordinates of our primitive and has up to four dimensions (x, y, z, w).

**TANGENT[n]** gives access to the tangents of our primitive. If we want to create normal maps, it will be necessary to work with a semantic that has up to four dimensions as well.

Through **NORMAL[n]** we can access the *normals* of our primitive, and it has up to four dimensions. We must use this semantic if we want to work with lighting within our shader.

Finally, **COLOR[n]** allows us to access the color of the vertices of our primitive and has up to four dimensions like the rest. Generally, the vertex color corresponds to a white color (1, 1, 1, 1).

To understand this concept, we are going to look at the structures that have been declared automatically within our **USB_simple_color** shader. We will start with **appdata**.

**Cg / HLSL vertex input & vertex output**                                    ✕

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

As we can see, there are two vectors within the structure: **vertex** and **uv**. "Vertex" has the POSITION semantic; this means that inside the vector, we are storing the position of the vertices of the object in *object-space*. These vertices are later transformed to *clip-space* in the **vertex shader stage** through the **UnityObjectToClipPos**(v.vertex) function.

The vector uv has the semantic TEXCOORD0, which gives access to the UV coordinates of the texture.

Why does the *vertex* vector have four dimensions (float4)? Because within the vector we are storing the values XYZW, where W equals "one" and vertices correspond to a position in space.

Within the **v2f** structure we can find the same vectors as in *appdata*, with a small difference in the SV_POSITION semantic, which fulfills the same function as POSITION[n], but has the prefix "**SV_**" (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);
    …
}
```

"**o.vertex"** is equal to the **vertex output**, that is, the vertex vector that has been declared in the **v2f** structure, while "**v.vertex**" is equal to the **vertex input**, that is, the vector vertex that has been declared in the **appdata** structure. This same logic applies to **uv** vectors.

## 3.3.1. | Cg / HLSL variables and connection vectors.

Continuing with the **USB_simple_color** shader, notice that within our program there is a variable of **sampler2D** type and a vector of four dimensions for the definition of the _MainTex texture.

**Cg / HLSL variables and connection vectors** ✕

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

These connection variables have been declared "**global**" within the Cg program and correspond to a_MainTex property reference, previously included in the shader **Properties**.

The connection variables are used to connect the values or parameters of the properties with our program variables or internal vectors. In the case of _MainTex, we can assign a texture from the Unity Inspector and use it as a texture in the shader.

To understand this concept better, let's assume that we want to create a shader that can change color. To do this, we would have to go to our **properties**, create a "**Color**" type and then generate a connection variable within the CGPROGRAM field, in this way we can generate a link between them.

**Cg / HLSL variables and connection vectors** ✕

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

    …

    CGPROGRAM

    …

    // then the connection variable or vector
    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 ignores this step since this declaration is included within the shader.

### 3.3.2. | Cg / HLSL vertex shader stage.

The vertex shader corresponds to a rendering pipeline's programmable stage, where the vertices are transformed from a 3D space to a two-dimensional projection on 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 our **vertex shader stage**. The reason why we know that it is our vertex shader is that it was declared as such in the #pragma vertex.

```
Cg / HLSL vertex shader stage                                          ✕

#pragma vertex vert

…

v2f vert (appdata v)
{
    …
}
```

Before continuing with the explanation of this stage, we must remember that our shader **USB_simple_color** is an **Unlit** type (it does not have light) that's why it includes a function for the **vertex shader** and another for the **fragment shader** (#pragma fragment frag). It is essential to mention this since Unity provides a quick way to write shaders in the form of "Surface Shader" (surf) that generates Cg code automatically, exclusively for materials that are affected by lighting. This allows optimization of development time but does not help us understand it because many functions and calculations occur internally in the program. This is why we created an Unlit shader at the beginning of this book; to understand its operation in detail.

We are going to analyze the structure of the vertex shader stage. Our function begins with the word "v2f" which means "vertex to fragment". This name makes a lot of sense when we understand the internal process that is happening inside the program. V2f will be used later as an argument in the **fragment shader stage**, hence its name. So, as our function

starts with v2f that means it is a vertex output type, therefore, we will have to return a value associated with this data type.

Continue with "**vert**" which is the **name** of our **vertex shader stage** and then the arguments in parentheses where appdata fulfills the function of vertex input.

```
Cg / HLSL vertex shader stage                                    ✕
v2f vert (appdata v) { … }
```

If we continue our analysis, we will notice that the struct v2f has been initialized with the letter "o" inside the function. Therefore, inside this variable, we will find all the previously declared properties in v2f.

```
Cg / HLSL vertex shader stage                                    ✕
v2f o;
o.vertex …
o.uv …
return o;
```

So the first operation that occurs within the vertex shader stage is the transformation of the object vertices from object-space to clip-space through the "UnityObjectToClipPos" method. Let's remember that our objects are within a three-dimensional space in the scene, and we must transform those coordinates into a two-dimensional projection of pixels on the screen. That transformation occurs precisely within the "UnityObjectToClipPos" function. What this function does is multiply the matrix of the current model (unity_ObjectToWorld) by the factor of the multiplication between the current 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 is declared in the "**UnityShaderUtilities.cginc**" file which has been included as a dependency in UnityCG.cginc and that is why we can use it inside our shader. So, we take the vertex input from our object (v.vertex), transform the matrix from object-space to clip-space (UnityObjectToClipPos) and save the result in the vertices output (o.vertex).

```
Cg / HLSL vertex shader stage                                          ✖

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

}
```

A factor that we must consider when working with inputs or outputs is that both properties must have the same number of dimensions, e.g., if we go to the vertex input appdata we will see that the vector "**float4** vertex" has the same number of dimensions as "**float4** vertex" in the vertex output (v2f).

If one property is of type float4 and the other float3, then Unity may return an error because in most cases you cannot transform from a four-dimensional vector to a three-dimensional vector or less.

Then we can find the TRANSFORM_TEX function. This function asks for two arguments, which are:

1. The input UV coordinates of the object (v.uv).
2. And the texture that we are going to position over those coordinates (_MainTex). It fulfills the function of controlling the "tiling and offset" in the UV coordinates of the texture.

Finally, we pass these values to the UV output (o.uv) since later they will be used in the *fragment shader stage*.

### 3.3.3. | Cg / HLSL fragment shader stage.

Our next and last function in the Pass corresponds to the **fragment shader stage** that appears in our shader with the name "**frag**". The reason we can tell that *frag* is the function

of the *fragment shader stage* is because it has been declared as such in the #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. This means that the fragment shader stage will process every pixel on the computer screen concerning the object we are viewing.

We are going to analyze the structure of the frag function. It starts with the data type "**fixed4**" which means that we will have to return a vector of four dimensions.

On the other hand, let's remember that this shader is currently written in Cg, so it will only compile in Built-in RP. If we want our shader to compile both in Universal RP and High Definition RP, we will have to change this data type for "**half4** or **float4**", otherwise our program could generate an error.

```
Cg / HLSL fragment shader stage                                          ✕

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


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

After the data type, the name continues with "frag" and as an argument, it has a "v2f" type variable called "i".

Unlike the vertex shader stage, this function has an output called "SV_Target", which allows us to render our scene in an intermediate buffer (render target) instead of sending the data to the Frame Buffer. In previous versions of Direct3D (version 9 and lower), the color output

**109**

in the fragment shader appeared with the COLOR semantic. However, in modern GPUs (version 10 onwards), this semantics is updated by  , which means "system value target". It can apply additional effects to the image before projecting them on the computer screen. Inside the **fragment shader stage** we can find a **fixed4** vector type called "**col**" which is the same as the **tex2D** function, where, as an argument, it receives the _MainTex texture and UV coordinate input. Basically what this operation does is store a texture within the **col** vector.

The reason why this vector has four dimensions is due to two conditions: The first because the frag function is a four-dimensional vector, so we will have to return a vector with an equal quantity of dimensions, and the second is because within the col vector we are going to store the colors of the texture in its RGBA channels, therefore, if the texture we use has an alpha channel (transparency), then it will be reflected in the object.

**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.

As mentioned above, this property allows us to assign a shader type object to the object in case our SubShader generates an error or is not compatible with the target hardware.

Its syntax is as follows:

**ShaderLab Fallback** ✕

```
Fallback "shaderPath"
```

"Shader Path" corresponds to the name and path of the shader that we want to use in case the SubShader fails.

The fallback can also be undeclared, leaving the space blank or using "Fallback Off", all the same, it is recommended to use a path and names of shaders that have been included in Unity such as "Mobile/Diffuse". This way, we can be sure that our shader will keep compiling 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 category "**Mobile**" in case the SubShader generates an error.

If we are developing a multiplatform game, it is advisable to declare a specific path for the Fallback, this way we can ensure that our 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 we translate the above definition to a practical level? Let's think of a material as "*the container of a shader*", this means that we have the program that is performing the surface calculations (shader) and the container (material) that is capable of reading these calculations.

A material alone cannot perform any operations. 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. | Our first shader in Cg or HLSL.

We will continue working with our "USB_simple_color" shader that we created at the beginning of this chapter.

As we already know, our default shader has a texture called _MainTex which is configured in the properties. What we will do next is add color to change the tint of the texture.

Before we begin, we must remember that we cannot directly apply a shader to an object in our scene. For this, we need to create a material and assign the shader to it so that it can be rendered graphically. To create a material we must go to our project, right-click on the folder in which we are working, go to *Create* and select *Material*. The default configuration of the material depends on the rendering pipeline we have just created. Generally, when we

create a material in *Built-in RP* it comes with a *Standard Surface* shader type which allows us to visualize the direction of lighting, shadows and other calculations associated with it.

So, we go to the hierarchy and create a 3D object. We assign the shader USB_simple_color to the material we have recently created and then apply the material to the object in our scene. At this point, we can assign a texture to our material to be projected onto the object.

Go back to our USB_simple_color shader and create a color property, as we will see in the following example:

<div>

**Our 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 { … }
}
```

</div>

What we just did was declare a color property for our shader. If we save and return to Unity we can see that this property appears in the Material Inspector, however, it is not working yet because we have not declared it within our CGPROGRAM. What we need to do next is add the **_Color** connection variable so that we can use it within the program. For this we go to the vertex shader stage; where _MainTex has been declared as sampler2D and create the connection variable in the following way:

<div>

**Our first shader in CG or HLSL**   ✕

```
uniform sampler2D _Maintex;
uniform float4 _MainTex_ST;
uniform float4 _Color;         // connection variable
```

</div>

As we can see, we have declared a global variable for **_Color** inside our CGPROGRAM or HLSLPROGRAM, however, we are still not using it yet. To change the tint of our texture, we

will go to the **fragment shader stage** and multiply **col** (texture color) by **_Color** using the multiplication sign. The operation should look like this:

```
// 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;
}
```

If we save our shader and go back to Unity, we can now change the tint texture from the material inspector.

### 4.0.3. | Adding transparency in Cg or HLSL.

In this section, we will add Blending so that our shader has a defined Alpha channel. In the previous configuration of USB_simple_color, we added color to change the tint of the texture. Now, it is worth mentioning that the color property has four channels (RGBA), however, if we modify the Alpha channel of the color from the Inspector, we will notice that this does not generate any change, why is this? Mainly because it does not have Blending properties. For our shader to be affected by transparency, we must: configure the **RenderType** as transparent, add the **Queue** with transparency and add the **Blending** type that we want to use.

**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 we save our shader and go back to Unity, we can now change the Alpha channel color, consequently, this will influence the transparency of the material's texture.

## 4.0.4. | Structure of a function in HLSL.

As in C#, in HLSL we can declare empty functions (void) or functions that return a value (return).

We have to use "declarations" that depend on the function type; these determine if a value corresponds to an **input** (in), **output** (out), **global variable** (uniform) or a **constant** value (const).

We will start by analyzing an empty function using the following syntax:

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

To declare an empty function, we start with the **void** nomenclature, then the **name** of the function accompanied by **precision** and final **arguments**. As we can see in the previous example, inside the function field we write the algorithm or operation to be performed. Generally, in the arguments, we must define whether these will be inputs or outputs.

How can we know if they have a declaration? Everything will depend on the functions that we want to pass as arguments. To understand this concept, let's suppose we want to create a function to calculate the illumination in an object, for this, one of the properties that we need are the normals, with this we can identify from which direction the light source will be illuminating our object. Therefore, the normals would be an "input to calculate" inside our empty function.

**Structure of a function in HLSL**

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

This function does not fulfil any specific functionality, but we will use it to understand the concepts mentioned before it.

The function is called "FakeLight" and "_float" corresponds to its precision, this can be of float or half type since, as we know, these are HLSL compatible formats.

We must always add precision to an empty function, otherwise, it cannot be compiled within our program. Then in the arguments, we can see that the object's normals (float3 Normal) have been declared as input through the "**in**" declaration, likewise, there is an output called "**out**" which will be the operation's final value.

To use this type of function within another, we must declare both inputs and outputs in our code before the function itself.

Let's simulate our FakeLight function inside the fragment shader stage as if it will actually operate

**Structure of a function in HLSL**　✕

```
// create our 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);
}
```

In the example above, there are several situations that are occurring. First, the "FakeLight" function has been declared before the "frag" function because the GPU reads our code from top to bottom. Then, in the fragment shader stage, we have created a three-dimensional vector called "**n**" and another three-dimensional vector called "**col**". In this case, both are three-dimensional vector types, this is because we will use both vectors as arguments in the FakeLight_float function, which asks for three-dimensional input and output vectors. Then, the first argument corresponds to the normal input of the object and the second, to the result of the operation that is being carried out within the FakeLight_float function.

**117**

The **col** vector has been started at "zero", this means that it has "0" for the red, green and blue (RGB) color, which corresponds to black by default, however, since it has been declared as output, it is now taking place inside the FakeLight_float function.

Finally, we return to a four-dimensional vector, where the first three values correspond to the **col** vector in RGB and "one" to the Alpha.

Why are we returning a four-dimensional vector? This is because the function frag is a **half4** type, that is, a four-dimensional vector.

Now we will analyze the structure of a function that returns a value. In essence, they are very similar, with the difference that in this case, it will not be necessary to add the precision function. To illustrate this, we are going to use the same FakeLight function, but this time it 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);
}
```
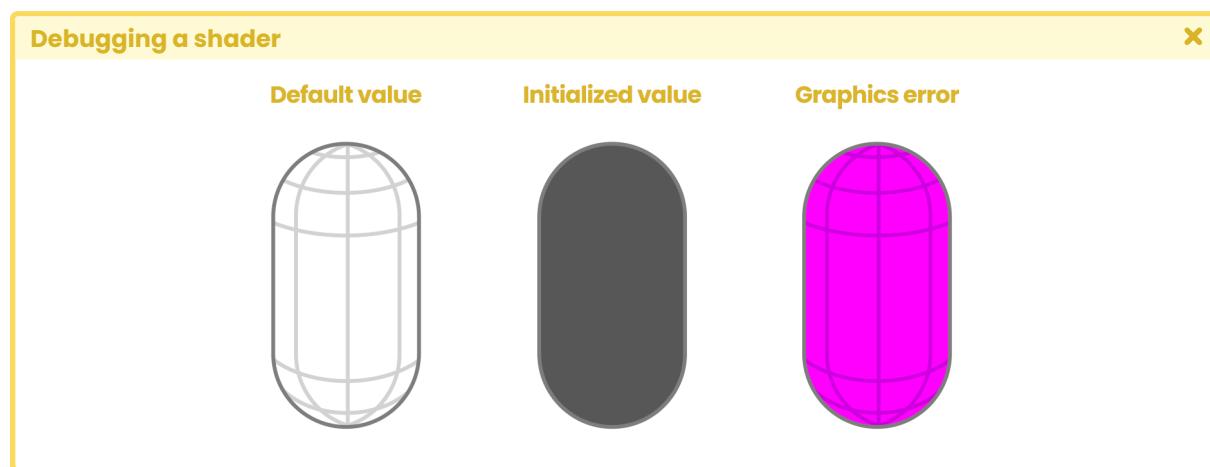
Unlike the empty function, we just add the Normal argument given that it does not require output, likewise, in the fragment shader stage we made the vector "col" equal to this function because it returns the same number of dimensions that this vector possesses.

## 4.0.5. | Debugging a shader.
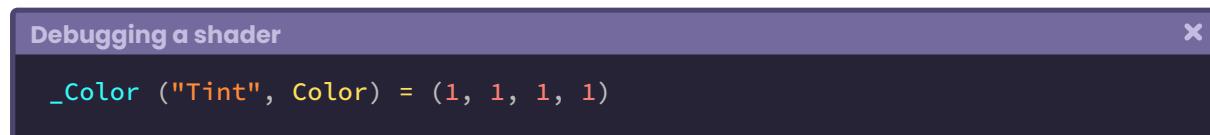
When we write a script in C#, in Unity we can debug our program using the Debug.Log (object message) function. This function allows us to print an operation of our code on the console, however, this function is not available in Cg or HLSL.

So how can we debug a shader? For this, we must consider several factors, including: "the colors". In a shader, there are three important colors that we see constantly, these are **white** (1, 1, 1, 1), **black** (0, 0, 0, 1) and **magenta** (1, 0, 1, 1). **White** represents a default value, while black represents an initialized value. Magenta represents a graphics error, in fact, it is very common to import assets to Unity that are magenta into our scene.



**Debugging a shader**                                                    ✕

Default value          Initialized value          Graphics error

*(Fig. 4.0.5a)*

To address this concept, let's recall the Properties' declaration in ShaderLab, for this, we will recreate a color property.

```
Debugging a shader                                                        ✕

_Color ("Tint", Color) = (1, 1, 1, 1)
```

In the example above, the property has been declared with a color "**white**" by default, we can corroborate this in the four-dimensional vector; at the end of the operation (1, 1, 1, 1). This property as such will generate a "color selector" in the Unity Inspector, and its color and/or initialization value will be white.

Let's analyze another property.

```
_MainTex ("Texture", 2D) = "white"{}
```

Again, this property as such will generate a "texture selector" in the Unity Inspector, but what will its default color be if we don't use any texture? Its color will be white, we can confirm this in the declaration "white" found on the property.

We see the color black frequently in declarations of internal vectors and variables in our code, in fact, it is very common to initialize vectors to "zero" and then add some operation, e.g., the four-dimensional vector "**float4 col = 0**" has been initialized to "0", this means that all its channels (RGBA) will have a default value of zero.

Note that white represents the maximum illumination value of one pixel and black represents its minimum illumination. It is important to remember this since, as we move forward, we will have some operations that will exceed the maximum (x > 1) and minimum (x < 0) values. In these cases, color saturation occurs and for this, we will use some functions like "clamp" that will allow us to limit a value between two numbers (minimum and maximum).

Why do our objects sometimes appear magenta? As we already know, this color represents a "graphical error". Basically, when the GPU could not carry out the operations found within the shader, it returns a default magenta color. In Unity, the reasons for this color are mainly due to two factors:
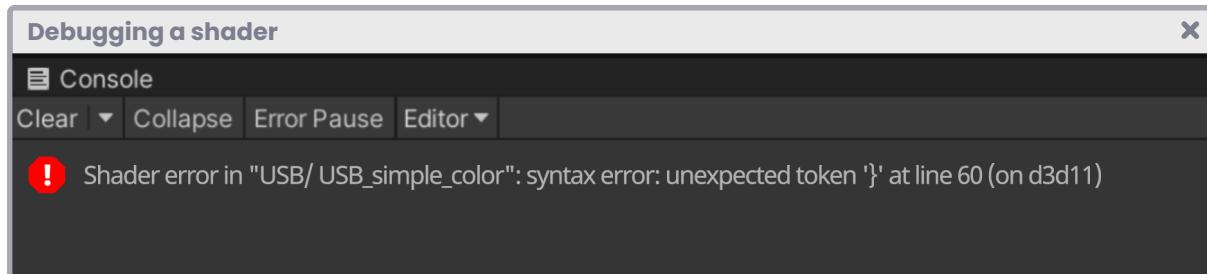
1.  That the render pipeline has not been configured.
2.  Or that the shader has an error in its code.

When we import an asset into the software, the first thing we need to know is, which render pipeline are we working in? Remember that in Unity there are three types of render pipeline and each one has a different configuration.

Suppose we import an asset that includes a material with a Standard Surface shader in Universal RP, then our asset will appear in magenta because this type of shader is supported only in Built-in RP. To solve this graphical error, we have to select the material, go to the inspector and change the type of shader to one found in "Shader Graphs".

Once we have identified the type of rendering pipeline that we are using, we must proceed to shader evaluation.

The errors in a shader are graphically evident since the objects acquire a magenta color. If we pay attention to the console, we can find its definition. To solve them, we must consider some factors: completing an instruction or function in HLSL and ShaderLab. Most errors are generated by misspelling or syntax. Below, we will review a reasonably common one:

**Debugging a shader**                                                        ✕

☰ Console

Clear ▾ Collapse | Error Pause | Editor ▾

🛑 Shader error in "USB/ USB_simple_color": syntax error: unexpected token '}' at line 60 (on d3d11)
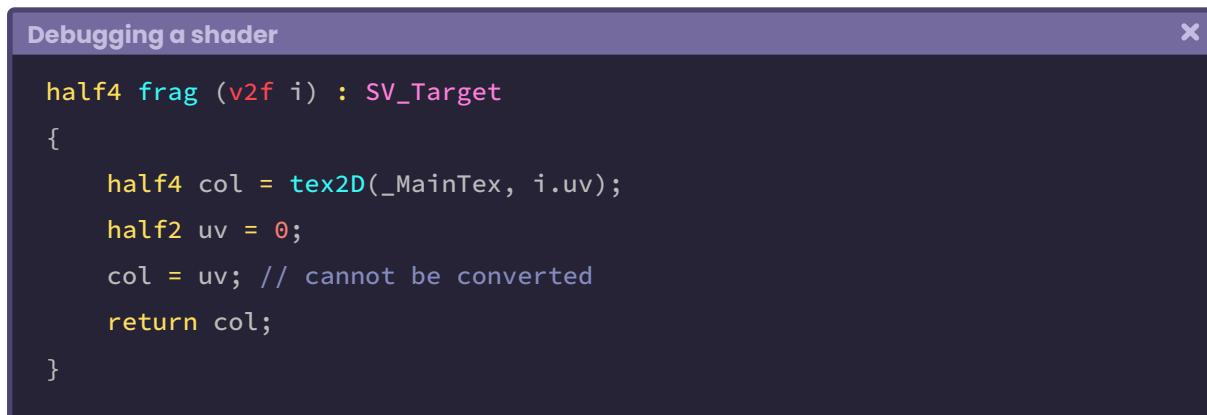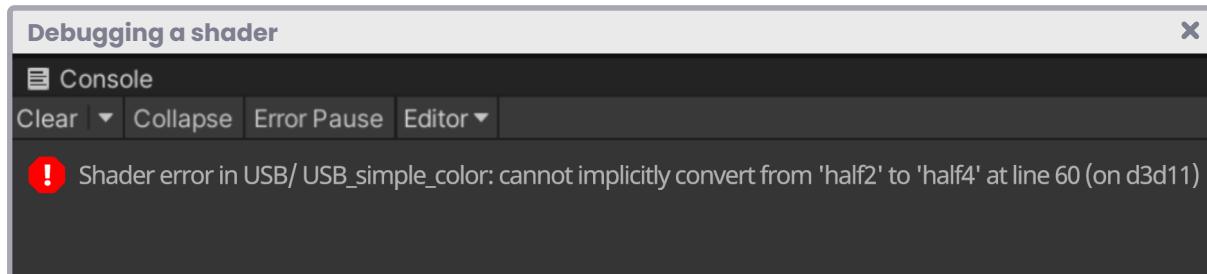
How can we read the above statement? In this case, the shader found in the "Unlit" menu, called "USB_simple_color", has an error in code line 60 and cannot be compiled in Direct3D 11.

Now, is that a shader problem? To do this, we are going to analyze the line of code that is generating the error.

**Debugging a shader**                                                        ✕

```
58  fixed4 frag (v2f i) : SV_Target
59  {
60      fixed4 col = tex2D(_MainTex, i.uv);
61      return col // (;)
62  }
```

According to the console, the error is being generated in code line number 60, but as we can see, there is no issue there. We have to ask ourselves then, what is happening? The error is because we forgot to close the operation in code line number 61. If we look at the **return** we will realize that the **col** vector is missing a semicolon (;) so the GPU thinks that the operation continues, so it can't compile it.

The following is another error that we see frequently:

**Debugging a shader**                                                                    ✕

📄 Console

Clear ▾ | Collapse | Error Pause | Editor ▾

❗ Shader error in USB/ USB_simple_color: cannot implicitly convert from 'half2' to 'half4' at line 60 (on d3d11)

**Debugging a shader**                                                                    ✕

```
half4 frag (v2f i) : SV_Target
{
    half4 col = tex2D(_MainTex, i.uv);
    half2 uv = 0;
    col = uv; // cannot be converted
    return col;
}
```

In this case, the error has been generated because we 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 the combination of two channels (RG, RB, GB, etc.), therefore it cannot be converted from a two-dimensional vector to one of four.

### 4.0.6. | Adding URP compatibility.

Up to this point, much of the variables, functions and vectors we have implemented, works for both Cg and HLSL, however, there will be some cases where we will have to add URP support so that our shader can compile.

In the case of Shader Graph, if we want to create a shader via code using Universal RP, we will have to add some dependencies so that the GPU can read this rendering pipeline. Such dependencies can be found in different routes, among which we can mention:

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

**122**

The "**Core RP Library**" is included automatically when we install Shader Graph in our project, likewise, the "**Universal RP**" package is included when we select this rendering pipeline as our rendering engine.

Both packages have files with ".hlsl" extension, which are required by our program to compile shaders in HLSL.

To understand this concept, let's duplicate the shader USB_simple_color and rename it as USB_simple_color_URP. It is worth mentioning that the shader USB_simple_color (Cg), being a basic color model, already has compatibility in Universal RP, however, we will generate a copy of this and modify it to review in detail its implementation in HLSL.

We will start by adding the Tags "**RenderPipeline**" and make it equal to "**UniversalRenderPipeline**", in this way the GPU will know that this shader will have compatibility in this render engine.

```
Adding URP compatibility                                    ✕

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

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

```
Adding URP compatibility                                    ✕

Pass
{
    HLSLPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    …
    ENDHLSL
}
```

For our shader to compile perfectly and its process to be more efficient we will have to include the dependency "Core.hlsl", which contains functions, structures and others, which allows the proper functioning of the program in Universal RP. In itself, Core.hlsl replaces "UnityCg.cginc", which means that we will have to eliminate this dependency of our 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.
hlsl"

…

ENDHLSL
```

Unlike UnityCg.cginc which comes included in the Unity installation as such, Core.hlsl is added as a package once we install Universal RP in our project, therefore, we will have to write the full path of its location in the project. Once we replace these dependencies, 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 function for the same reason.

Instead, we will have to use the function TransformObjectToHClip(v.vertex) which is included in the dependency "SpaceTransforms.hlsl", which in turn, has been included in "Core.hlsl".

The function TransformObjectToHClip performs the same operation as UnityObjectToClipPos, that is, it transforms the position of the vertices from object-space to clip-space, however, the former is more efficient in its execution process.

**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 we save our shader and go back to Unity, we can see that an error has been generated in the console of type "unrecognized identifier", what is the reason for this? Remember that both the fragment shader stage and the col vector, by default, are of type **fixed4.** So we 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;
}
```

**Debugging a shader**                                                          ✕

```
☰ Console
Clear ▼  Collapse  Error Pause  Editor ▼

⛔  Shader error in "USB/USB_simple_color_URP": unrecognized identifier "fixed4" at line 62 (on d3d11)
```

As we already know, Universal RP can not compile variables or vectors of the type "fixed", instead we will have to use "half or float". Therefore, at this point, we can do two things.

1. Manually replace variables and vectors of type fixed 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, we can work with variables and vectors of the fixed type, since now our 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, we can find intrinsic functions that will help us program effects. Such functions correspond to general mathematical operations, and we use them in specific cases depending on the result we wish to obtain. Among the most common we can find:

- **Abs.**
- **Ceil.**
- **Clamp.**
- **Cos.**
- **Sin.**
- **Tan.**

- **Exp.**
- **Exp2.**
- **Floor.**
- **Step.**
- **Smoothstep.**
- **Frac.**

- **Length.**
- **Lerp.**
- **Min.**
- **Max.**
- **Pow.**

Which we will define 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:
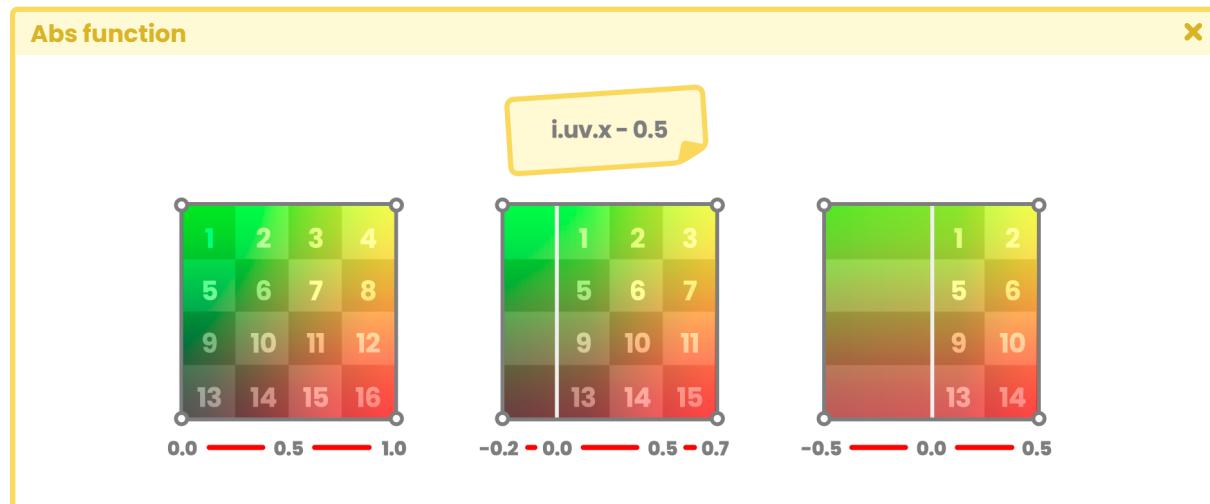
```
Abs function                                                    ✕
// return the absolute value of n
float abs(float n)
{
    return max(-n, n);
}

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 is equal to itself
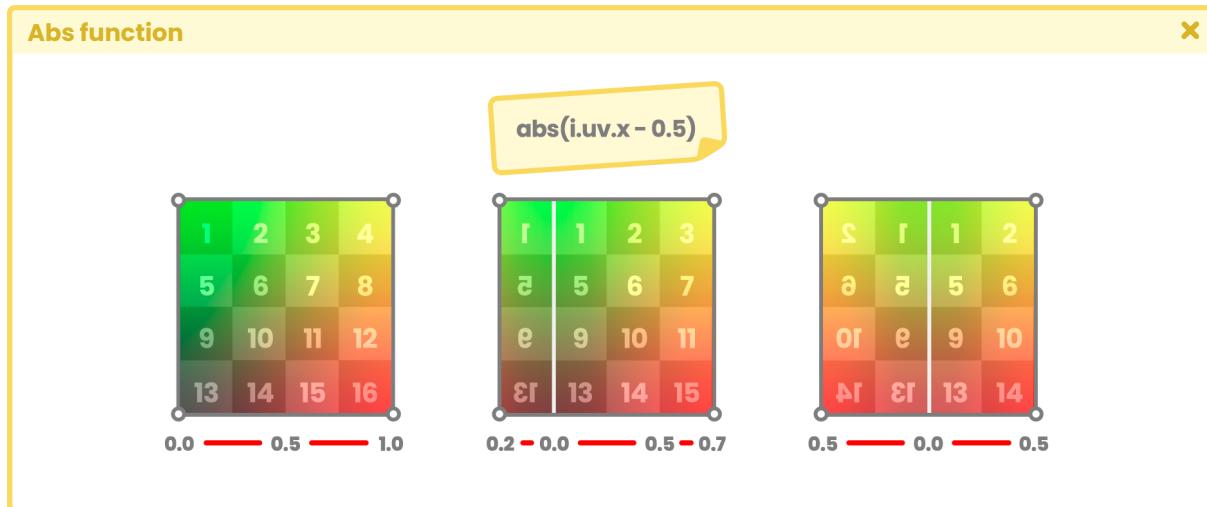
Our program could use the **abs(n)** function for multiple effects, including recreating a kaleidoscope or generating a triplanar mapping. In fact, for the first case, we could perform such an effect by calculating the absolute value in the UV coordinates. At the same time, in the triplanar mapping, we could determine the absolute value of the mesh's normals to generate projections on both positive and negative axes.



(Fig. 4.0.8a. As we can see in section 1.0.5, UV coordinates start at 0.0f and end at 1.0f. In the previous figure, we can see the behavior of the U coordinate when we subtract 0.5f. The texture has been set to clamp in the Inspector)
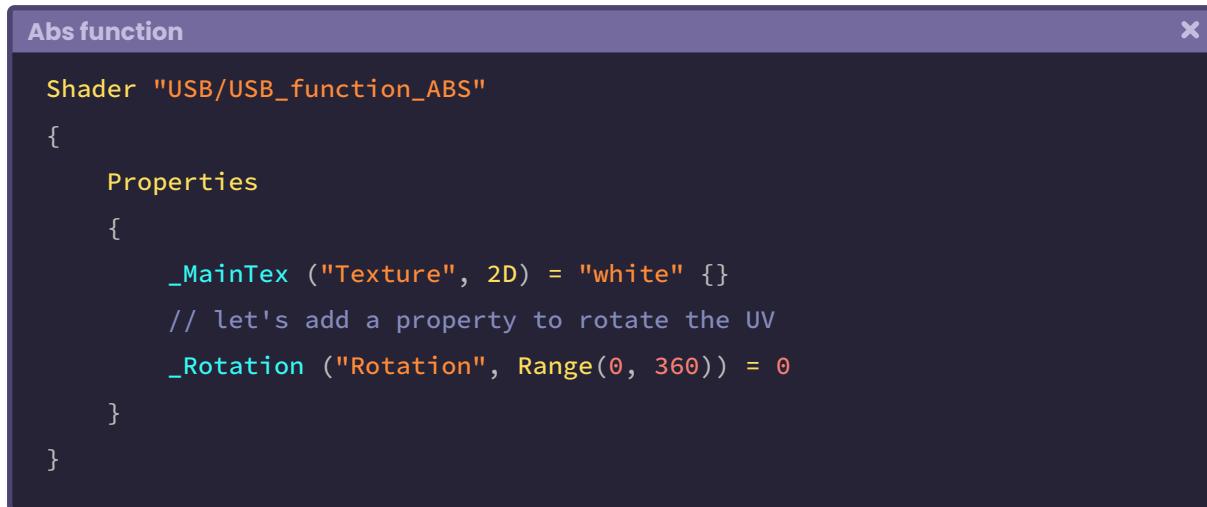
If we pay attention to the starting point in the UV coordinates of the previous figure, we will notice that the U coordinate is centered on the Quad by subtracting 0.5f, and the minimum value becomes a negative number [-0.5f].

In the example, the texels get stretched because the texture has been set to **clamp** in the **wrap mode**. In this context, we could apply the absolute value to generate a "mirror" effect.

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

Next, we will develop the effect of a kaleidoscope to understand the concept fully. We will start by creating a new shader type, "Unlit Shader", which we will call "USB_function_ABS". Then, we declare a new property in the shader properties that we will use later to rotate the UV coordinates.

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

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

```
Abs function                                                    ✕

 Pass

 {

     CGPROGRAM

     …

     sampler2D _MainTex;

     float4 _MainTex_ST;

     float _Rotation;

     …

     ENDCG

 }
```

We can calculate both the U and V coordinates' absolute values 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

 {

     // let's calculate the absolute value of U

     float u = abs(i.uv.x - 0.5);

     // let's 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;

 }
```
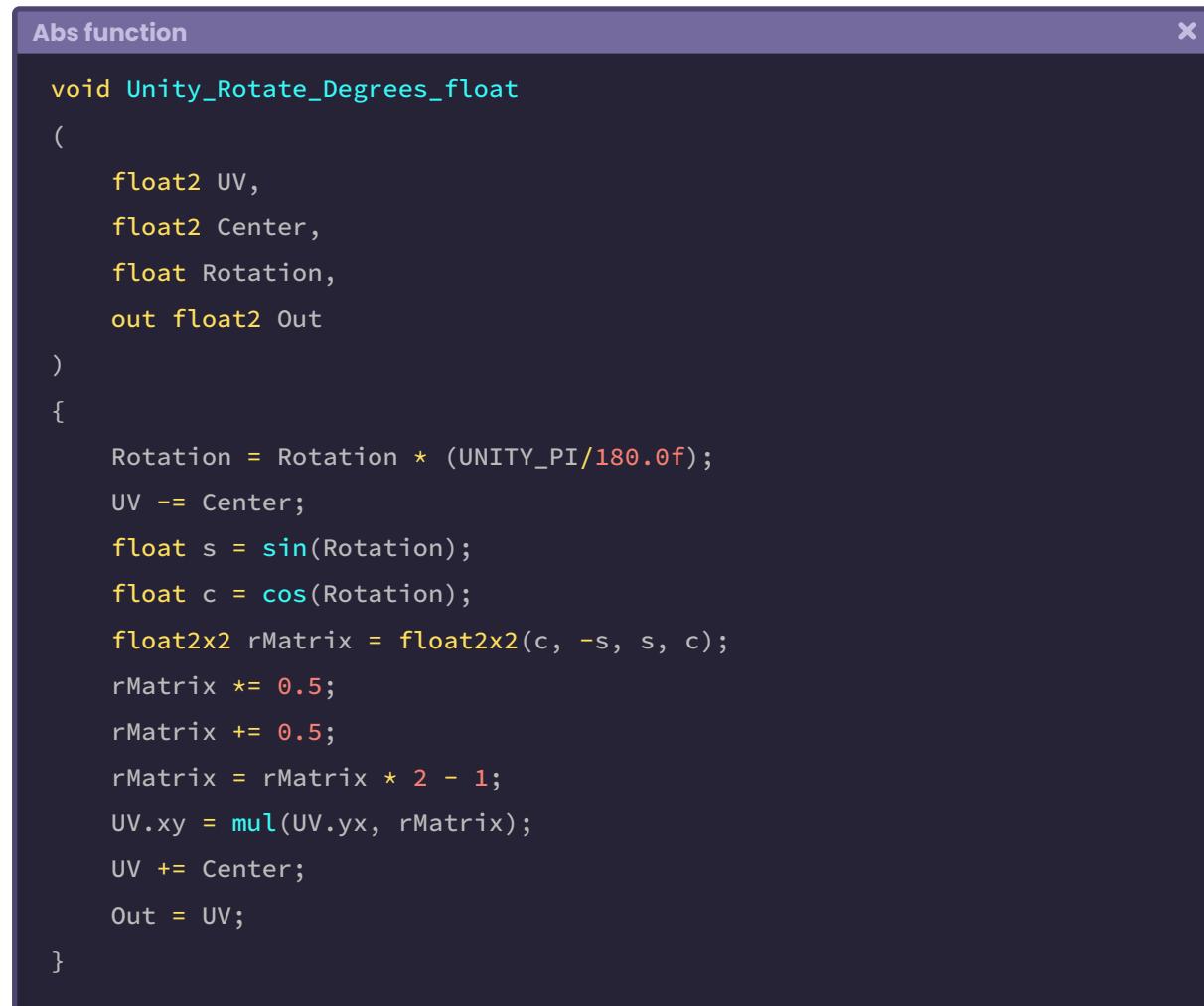
Two main things can happen depending on the texture configuration we are assigning as _MainTex,

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

Regardless of the configuration, the mirror effect will be evident since we use each coordinate's abs(n) function.

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

```
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 the function is of type "**void**", we 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);
    // we link the rotation property
    float rotation = _Rotation;
    // we center the rotation pivot
    float center = 0.5;
    // let's 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 we want to rotate, continues the center of rotation or pivot, then the number of degrees, and finally the output with the new coordinate values that we will use for the texture.

## 4.0.9. | Ceil function.

According to NVIDIA's official documentation,

> *Ceil returns the smallest integer not less than a scalar or each vector component.*

What does this mean? The function ceil(n) will return an integer, that is, without decimals, close to its argument, e.g., if the number is equal to 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                                                        ✕

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

float2 ceil (float2 n);
float3 ceil (float3 n);
float4 ceil (float4 n);
```
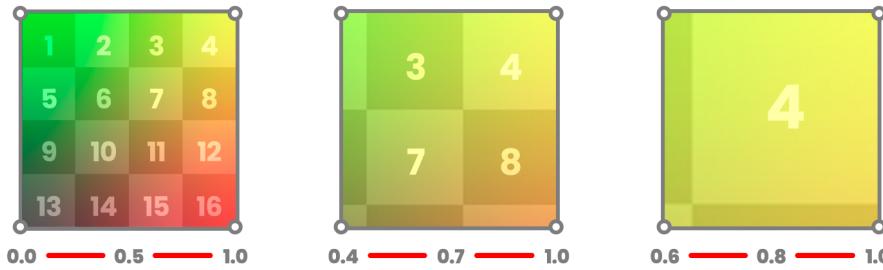
This function is quite helpful for generating "zoom or magnifying glass" effects in a video game. To do this, we simply have to calculate the value of ceil(n) for both the U and V coordinates, multiply the final value by 0.5 and then generate a linear interpolation between the UV's default values and those resulting from the ceil(n) function.

To understand the concept in-depth, we will do the following: We will create a new shader type, "Unlit Shader", which we will call **USB_function_CEIL**, and we will start by declaring the new UV coordinates for the texture _MainTex, in the fragment shader stage.

**Ceil function** ✕

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

}
```

The operation we performed above will return the texel position [1, 1], for what reason? Because both the U and V coordinates start at 0.0f and end at 1.0f, ceil(n) will only return the last texel in the texture. Consequently, it will generate a zoom effect that will go from the upper right point of the texture to the lower-left end.

**Ceil function** ✕



*(Fig. 4.0.9a. Ceil is going to return the last texel found in the texture; one who is in position 1, 1)*

For this effect, we will need a property that allows us to increase or decrease the size of the texture. To do this, we will go to the properties and declare a floating range that we will call _Zoom.

```
Ceil function                                                    ✕

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

In the range, "0" symbolizes zero per cent zoom, and "1" is equivalent to one hundred per cent. Then we declare the connection variable within the program.

```
Ceil function                                                    ✕

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

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

```
Ceil function                                                    ✕

fixed4 frag (v2f i) : SV_Target
{
    // let's ceil the U coordinate
    float u = ceil(i.uv.x) * 0.5;
    // Let's do the same with the V coordinate
    float v = ceil(i.uv.y) * 0.5;
```

*Continued on next page.*

```
    // we assign the new values for 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, we still can not appreciate the zoom effect since the ceil(n) function continues returning one; therefore, we will continue to see a single color filling the Quad area. What we can do is generate a **linear interpolation** between the UV's default values and those resulting from the ceil(n) function,

**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);

    // we assign the new values for the texture
    fixed4 col = tex2D(_MainTex, float2(uLerp , vLerp));

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

In the example above, we created two variables called **uLerp** and **vLerp** for the different coordinates. We are doing a linear interpolation through the lerp(x, y, s) function belonging to the Cg/HLSL language. In addition, the property _Zoom has been included, which has a range between 0.0f and 1.0f. If we modify the values of the property _Zoom from the Unity Inspector, we can see how the texture increases or decreases its size, taking as a reference its central point.

**Ceil function** ✕

ceil(i.uv) * 0.5

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

## 4.1.0. | Clamp function.

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

As we develop functions, we 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 the mesh normals and the illumination direction, we can obtain a range between -1.0f and 1.0f. Since a negative value would generate color artifacts in the final effect, with clamp, we can limit and redefine the range between 0.0f and 1.0f.

Its syntax is the following:
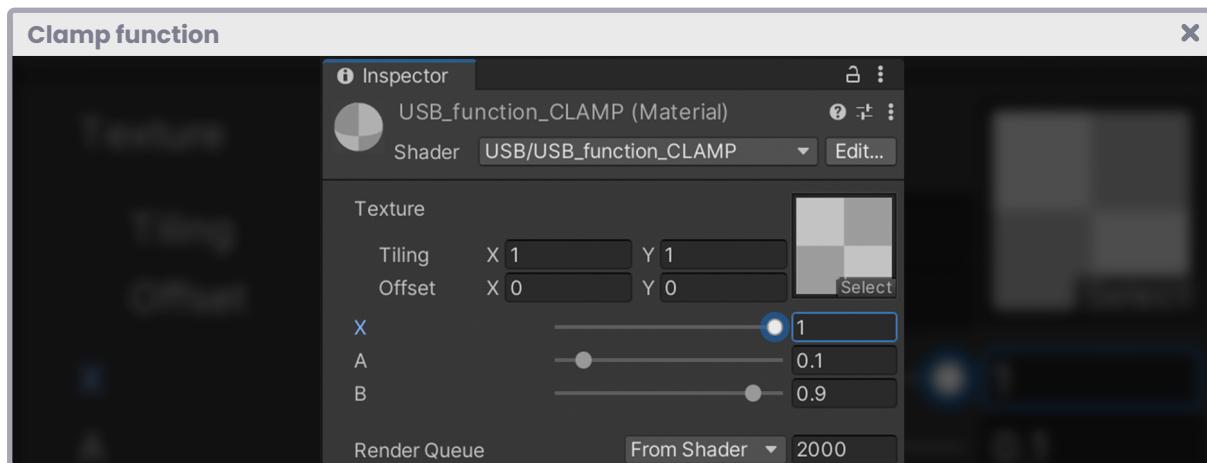
```
Clamp function                                               ✕
float clamp (float a, float x, float b)
{
    return max(a, min(x, b));
}

float2 clamp (float2 a, float2 x, float2 b);
float3 clamp (float3 a, float3 x, float3 b);
float4 clamp (float4 a, float4 x, float4 b);
```
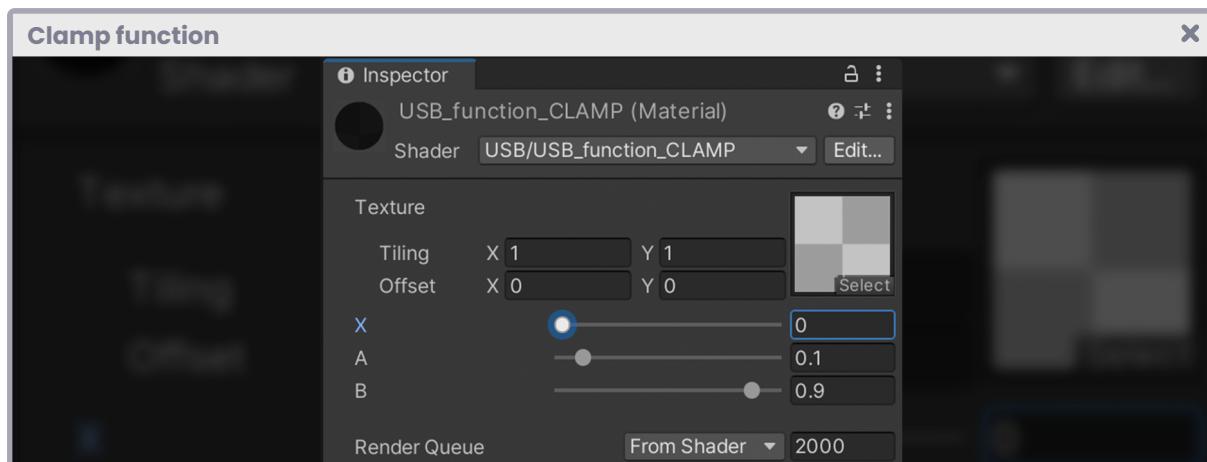
In the above function, the argument "a" refers to the minimum return value, while the argument "b" refers to the maximum return value within the range. As for the argument, "x" corresponds to the value we want to limit according to a and b. What does this mean? Let's assume a set range for "a" and "b" and a variable number for x.

When "x" is equal to 1.0f, if argument "a" is equal to 0.1f and argument "b" is equal to 0.9f, the maximum return value for "x" will be 0.9f; why? Because "b" defines the highest limit value for the operation.



*(Fig. 4.1.0a)*

The same happens in the opposite case. When *"x"* is equal to 0.0f, the return value will be 0.1f, because *"a"* defines the minimum return value for *"x"*.



*(Fig. 4.1.0b)*

To understand this concept in-depth, we will do the following: First, we will create a new shader type, "Unlit Shader", which we will call **USB_function_CLAMP**, and in its properties, we will declare three floating ranges; one for each argument.
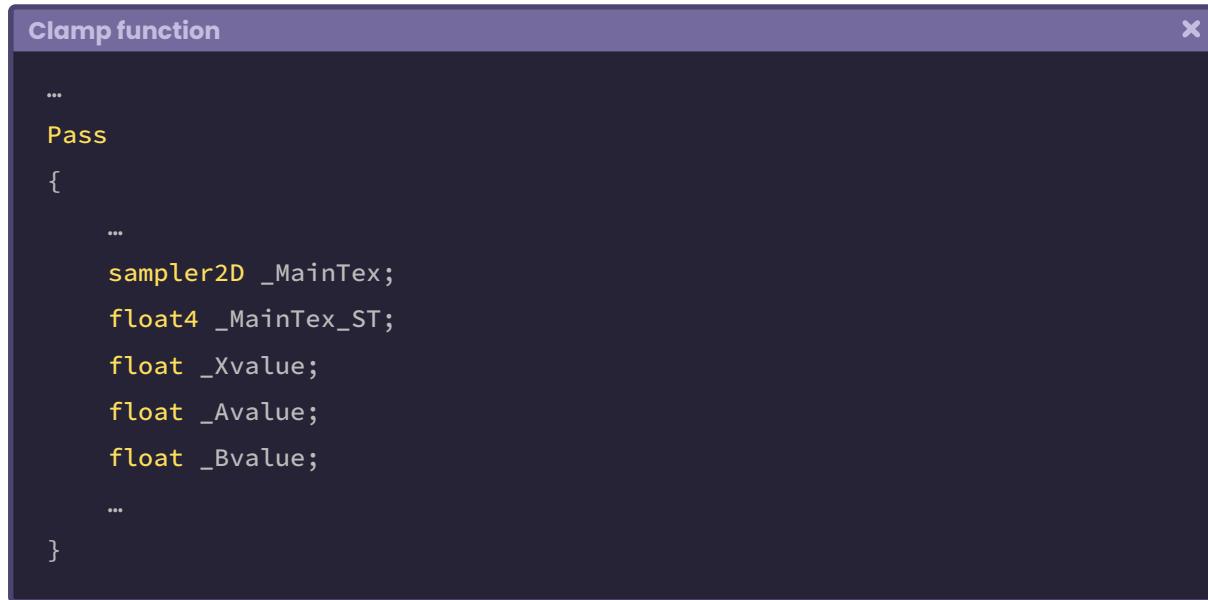
**Clamp function** ✖

```
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 we declare the global or connection variables to connect them to the program.

**Clamp function** ✖

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

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

1. Generate a simple function within our program that limits a value within a range.
2. Or use the included function "clamp" that is in the Cg/HLSL language.

We will start by declaring a new function. To do this, we position ourselves between the vertex shader and the fragment shader stage and write a new method that we will call "**ourClamp**".

```
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 { … }
```

As we can see in the implementation of the previous function, ourClamp will limit a value (float x) between two established numbers (float a and float b).

Then, in the fragment shader stage, we create a floating variable called "**darkness**" and make it equal to our new function. As arguments, we will pass the properties that we declared above, following the mentioned 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, multiplying the col vector by the mentioned variable will affect the four channels of the vector col (RGBA).

```
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;

  }
```

We can simplify the above operation by including the "clamp" function in the language we are applying.

```
Clamp function                                               ✖

  // float ourClamp(float a, float x, float b) { … }


  fixed4 frag(v2f i) : SV_Target
  {
      // Cg and HLSL include the clamp function.
      float darkness = clamp(_Avalue, _Xvalue, _Bvalue);
      fixed4 col = tex2D(_MainTex, i.uv) * darkness;


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

  }
```

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

## 4.1.1. | Sin and Cos function.

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:
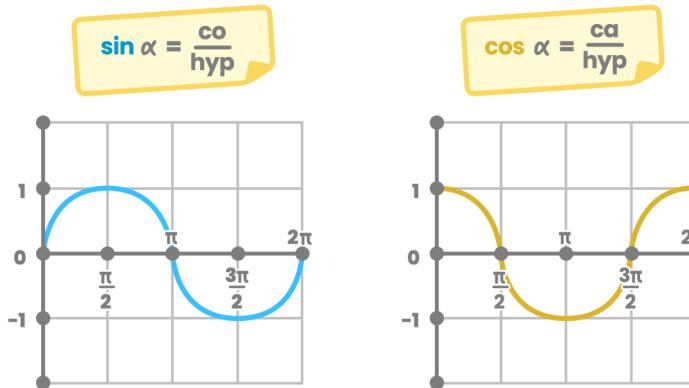
**Sin and Cos function**                                                                    ✕

```
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);
```

We can use both "cos and sin" on scalar values and vectors.

**Sin and Cos function**                                                                    ✕
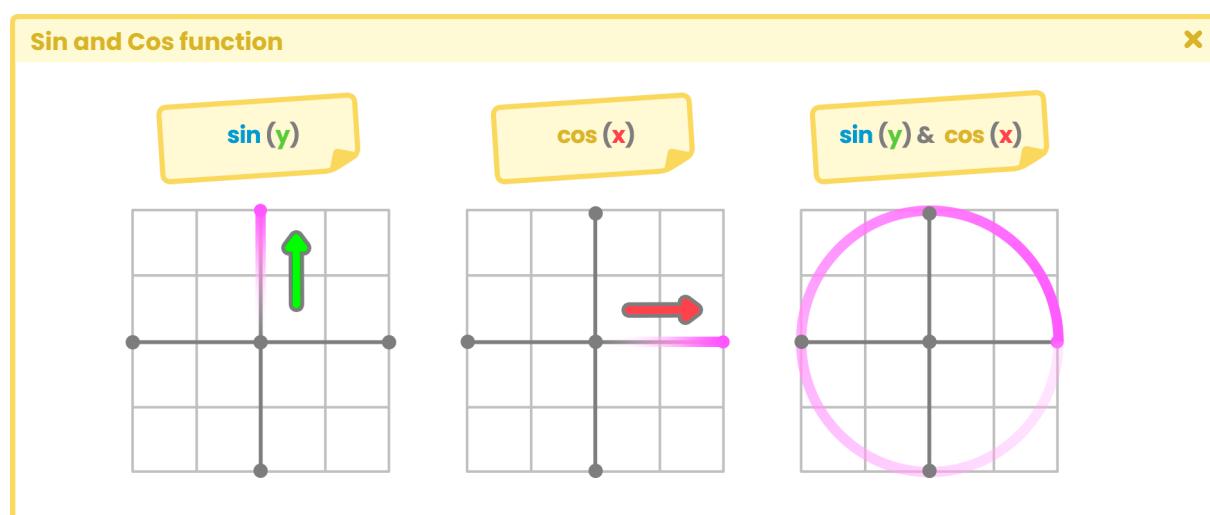


*(Fig. 4.1.1a. Graphical representations of the functions on a Cartesian plane.*

*On the left, we can see the sin of x, and on the right, cos of x.)*

These functions also included in the Cg/HLSL language are very useful in Computer Graphics. With them, we can generate multiple geometric figures and even matrix transformations. A practical example of implementation is the rotation of vertices in an object.

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

Let's imagine that we want to rotate a vertex in a two-dimensional space. Applying the function "sin" on its "Y" axis will obtain a wave motion going from top to bottom. Using the function "cos" on its "X" axis will reproduce a circular motion.

**Sin and Cos function** ✕



sin (y)      cos (x)      sin (y) & cos (x)

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

To understand the concept, we will do the following: we will create a new shader type, "Unlit Shader," which we will call **USB_function_SINCOS**. We will use this shader to generate a small rotation animation on the Cube's vertices. We will start by declaring a floating range, which we will use later in the function to determine the rotation speed.

**Sin and Cos function** ✕

```
Shader "USB/USB_function_SINCOS"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
```
*Continued on next page.*

```
        _Speed ("Rotation Speed", Range(0, 3)) = 1
    }
    SubShader { … }
}
```

We need the help of a matrix to rotate coordinates; in section 3.2.7, we were able to review its structure using matrices of different dimensions. This time, we will use a three times three-dimensional matrix (float3x3) on our object.

**Sin and Cos function**                                                    ✕
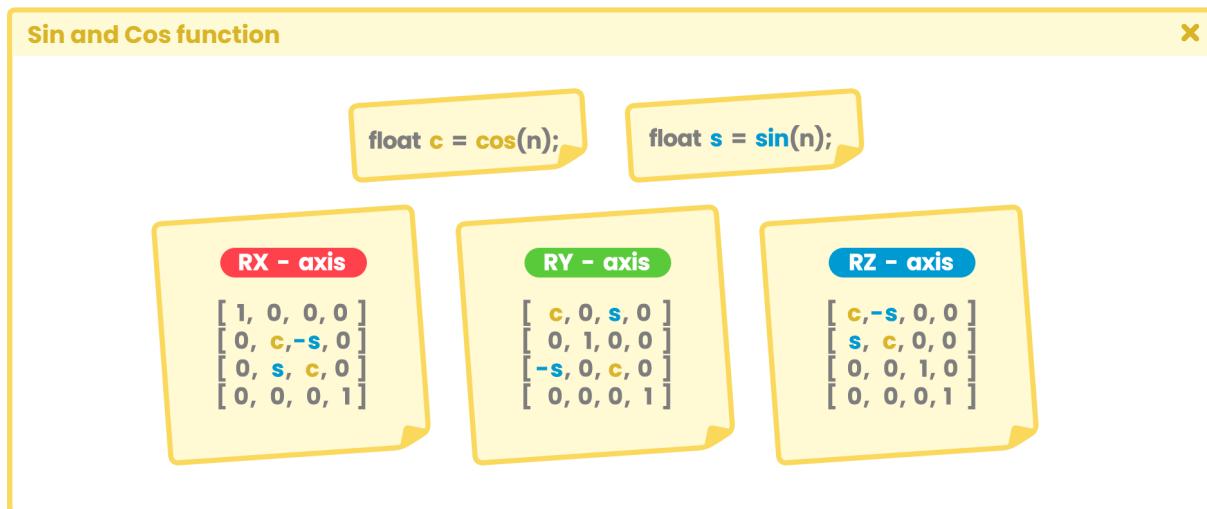
```
Pass
{
    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Speed;

    // let's 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
        );
        // let's multiply the matrix times the vertex input
        return mul(m, vertex);
    }
    …
}
```

The "rotation" function returns a three-dimensional vector, and it does not perform any specific action because the matrix "m" has only its identity values. However, we can use it later to transform the vertices of an object in the vertex shader stage.

We will start by selecting a rotation axis; we will pay attention to the following diagram.

**Sin and Cos function** ✕

float c = cos(n);   float s = sin(n);

**RX - axis**
```
[ 1, 0, 0, 0 ]
[ 0, c,-s, 0 ]
[ 0, s, c, 0 ]
[ 0, 0, 0, 1 ]
```

**RY - axis**
```
[ c, 0, s, 0 ]
[ 0, 1, 0, 0 ]
[-s, 0, c, 0 ]
[ 0, 0, 0, 1 ]
```

**RZ - axis**
```
[ c,-s, 0, 0 ]
[ s, c, 0, 0 ]
[ 0, 0, 1, 0 ]
[ 0, 0, 0, 1 ]
```

*(Fig. 4.1.1c. The rotation axis in a four-dimensional matrix)*

In the figure above, each of the matrices represents a rotation transformation axis. To exemplify, in this opportunity, we will perform the exercise using the "Y" axis (RY axis). Therefore, we will have to add two floating variables in the rotation method, using sin and cos trigonometric functions.

**Sin and Cos function** ✕
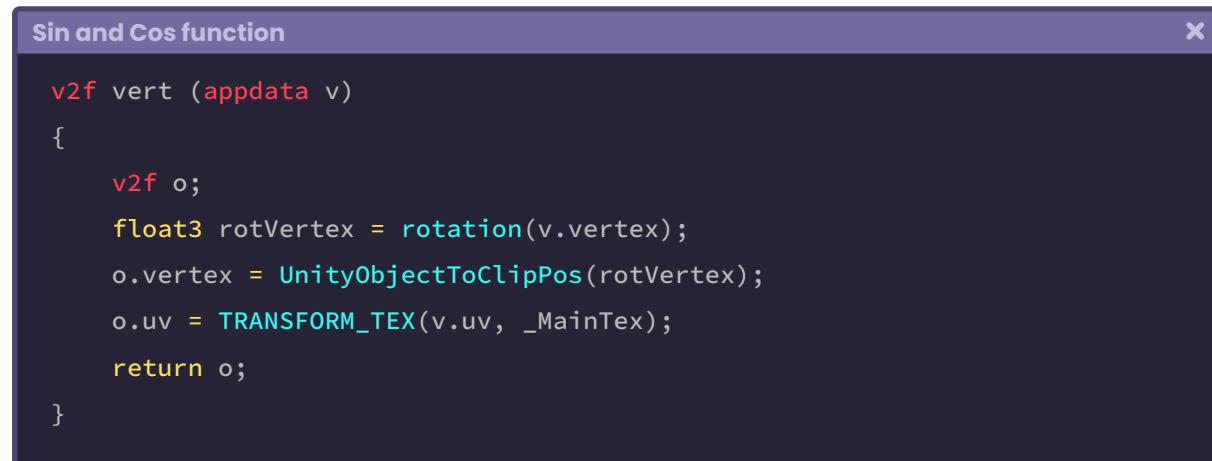
```
float3 rotation(float3 vertex)
{
    // let's add the rotation variables
    float c = cos(_Time.y * _Speed);
    float s = sin(_Time.y * _Speed);

    // create a three-dimensional matrix
    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, we will talk about the "**_Time**" property in detail. For now, we will 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 our scene, so they will start to move according to their rotation axis.

The **rotation** function can operate perfectly; now, we must implement it on the vertex shader stage. For this purpose, we must consider the **UnityObjectToClipPos** method since, as explained in section 3.3.2, it allows transforming the vertices of an object from object-space to clip-space. Therefore, we will have to implement the rotation of the vertices before transforming their coordinates into screen position.

```
Sin and Cos function                                              ✕

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 exercise, a new three-dimensional vector called "rotVertex" has been declared. In it has been stored the input of mesh vertices and their rotation in object-space. This vector was then used as an argument in the UnityObjectToClipPos method.

If we go back to Unity and press the Play button, we will see the rotation of the Cube vertices as a whole.

## 4.1.2. | Tan function.

This trigonometric function refers to the tangent of an angle, that is:
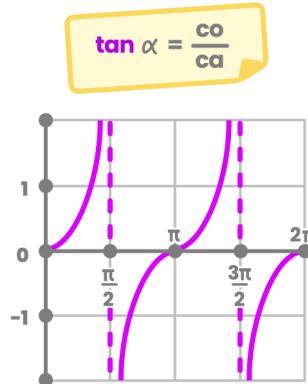
- The ratio of the opposite side to the adjacent side.

Its syntax is as follows:

**Tan function** ✕
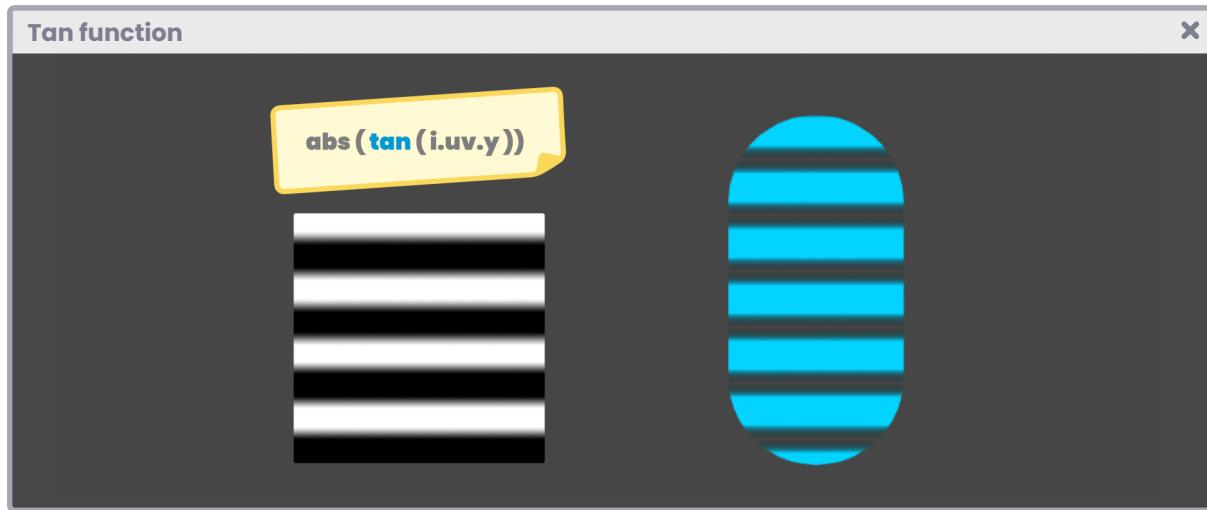
```
float tan (float n);
float2 tan (float2 n);
float3 tan (float3 n);
float4 tan (float4 n);
```

**Tan function** ✕



$$\tan \alpha = \frac{co}{ca}$$

*(Fig. 4.1.2a. Graphical representation of the **tan** function on a Cartesian plane)*

Like **sin** and **cos**, **tan** 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 we can use to generate the holographic projection effect on an object. For this purpose, we can simply calculate the absolute value of the tangent at one of the UV coordinates within the fragment shader stage.

*(Fig. 4.1.2b)*

We will exemplify by generating a new shader of type "Unlit Shader," which we will call USB_function_TAN. We will start by declaring a color property and a range to increase or decrease the number of lines we want to project.

```
Tan function                                               ✕

  Shader "USB/USB_function_TAN"
  {
      Properties
      {
          _MainTex ("Texture", 2D) = "white" {}
          _Color ("Color", Color) = (1, 1, 1, 1)
          _Sections ("Sections", Range(2, 10)) = 10
      }
      SubShader { … }
  }
```

It's worth mentioning that we will apply this shader to 3D objects included in the software. The reason for saying this point is due to the operation we will perform on the V-coordinate of the UV.

As we can see in Figure 4.1.2b, such a visual effect has transparency; therefore, we will have to add blending options in the SubShader so that the black color will be recognized as an alpha channel.

**Tan function** ✕

```
SubShader
{
    Tags {"RenderType"="Transparent" "Queue"="Transparent"}
    Blend SrcAlpha OneMinusSrcAlpha


    Pass { … }

}
```

We will make sure to declare the global or connection variables, and then we will go to the fragment shader stage to add the functionality that will allow us to 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, we have declared a four-dimensional vector called **tanCol**. We have stored the result of the absolute value for the calculation of the tangent; of the product between the V coordinate and the **_Sections** property in it. Subsequently, the factor between the **texture** and the vector **tanCol** has been stored in the four-dimensional vector named col. Therefore, the texture that we assign to the **_MainTex** property will be interlined.

A small detail that we can find in the previous operation is that the tangent of V returns a numerical range less than "zero" and greater than "one." Therefore, the final color will be saturated on the computer screen. To solve this problem, we can use the **clamp** function, limiting the values between 0.0f and 1.0f.

**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;

}
```

We can use the _Time variable again to generate movement in the spacing. To do this, we simply subtract or add this property to 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 function.

These functions are characterized by using exponents in their operations, e.g., the function "**exp**" returns the exponential of **base-e** in scalar and vector values, that is to say, "e" (2.7182828182846f) raised to a number.
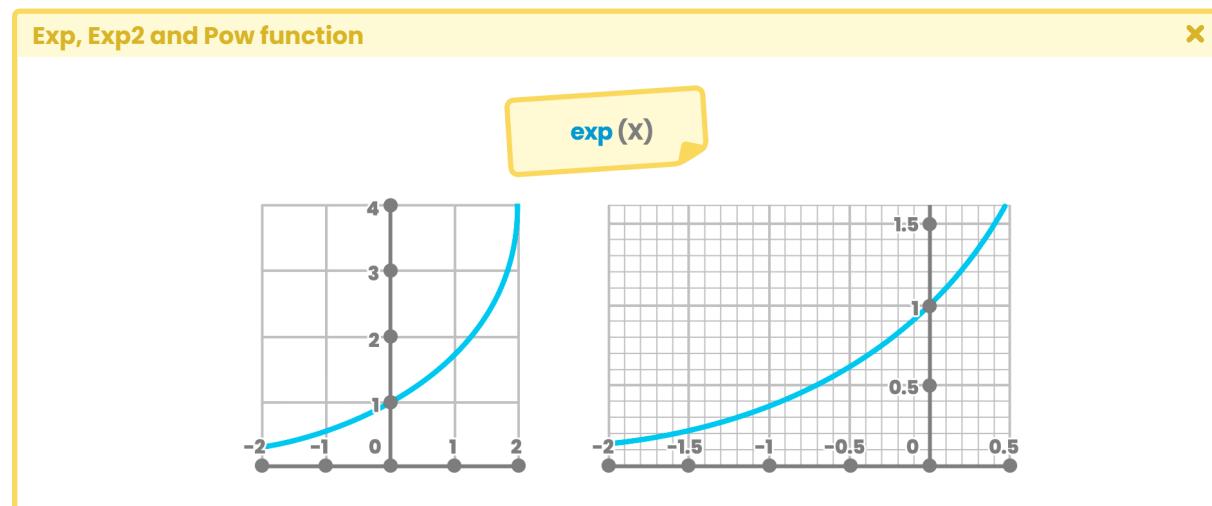
exp (2) = 7.3890560986f     It's the same as $2.71828182846f^2$

Its syntax is as follows:

```
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 function**  ✕

**Exp, Exp2 and Pow function**  ✕



exp (x)

*(Fig. 4.1.3a. Graphical representation of exp(x) on a Cartesian plane)*

Moreover, "exp2" returns the base-2 exponent in values of different dimensions, that is, two raised to a number.

exp2 (3) = 8           It's the same as $2^3$
exp2 (4) = 16
exp2 (5) = 32

**Exp, Exp2 and Pow function**                                          ✕

```
float exp2 (float n);
float2 exp2 (float2 n);
float3 exp2 (float3 n);
float4 exp2 (float4 n);
```

On the other hand, "pow" has two arguments: The base number (x) and its exponent (n).

pow (3, 2) = 9                    It's the same as $3^2$
pow (2, 2) = 4
pow (4, 2) = 16

**Exp, Exp2 and Pow function**                                          ✕

```
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 1; why? Because this function subtracts the decimals of a number from its total.
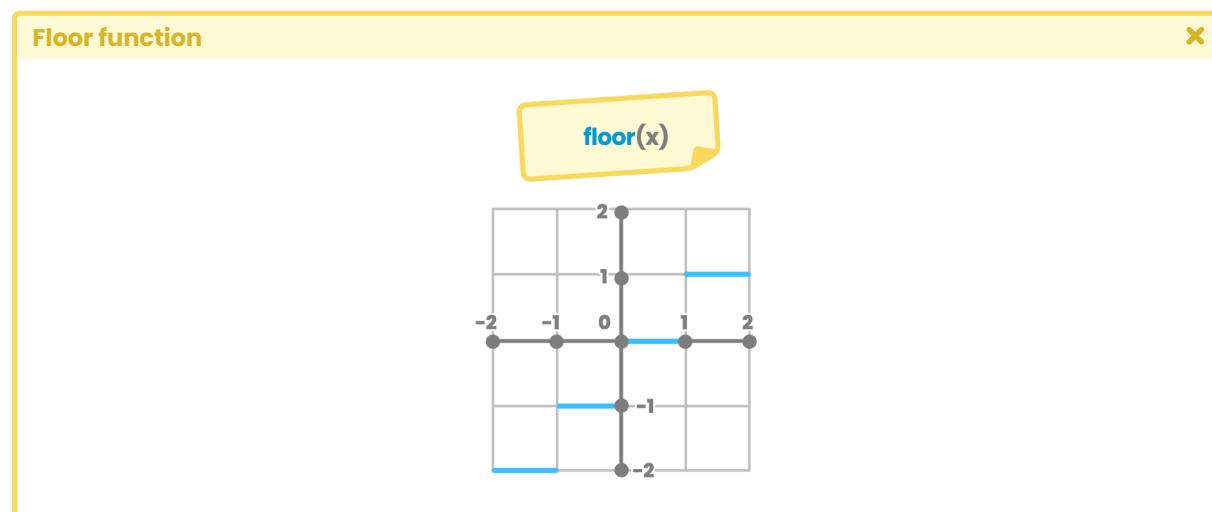
floor (1.56) = 1                 it's the same as 1.56f – 0.56f.
floor (0.34) = 0
floor (2.99) = 2

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);
```

**Floor function** ✕



*(Fig. 4.1.4a. Graphical representation of floor(x) on a Cartesian plane)*

Contrary to the ceil function, the **floor** is quite useful when creating visual effects with solid blocks of color, e.g., toon shader or repeating patterns in general.

To exemplify, let's understand the principle of implementing a toon shader. In this sense, we are going to generate a new shader type, "Unlit Shader," which we will call **USB_functions_ FLOOR**.

We will start by adding two properties in our shader: we will use the first to generate multiple splits 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 aggregated uniformly, **[IntRange]** has been defined for the _ Sections variable. As in previous processes, we must include the global variables inside the Pass so that we will achieve direct communication between ShaderLab and our program.

**Floor function** ✕

```
Pass
{
    CGPROGRAM

    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _Sections;
    float _Gamma;

    …
    ENDCG
}
```

Next, we will 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" has been declared and initialized, with only one dimension. Its value is equal to the floor result of  V; hence, it is equal to "zero." it's because the UV coordinates start at 0.0f and end at 1.0f, and as we already know, this function returns an integer not greater than its argument.

We can corroborate the operation 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);

  }
```

```
Floor function                                                              ✕
```
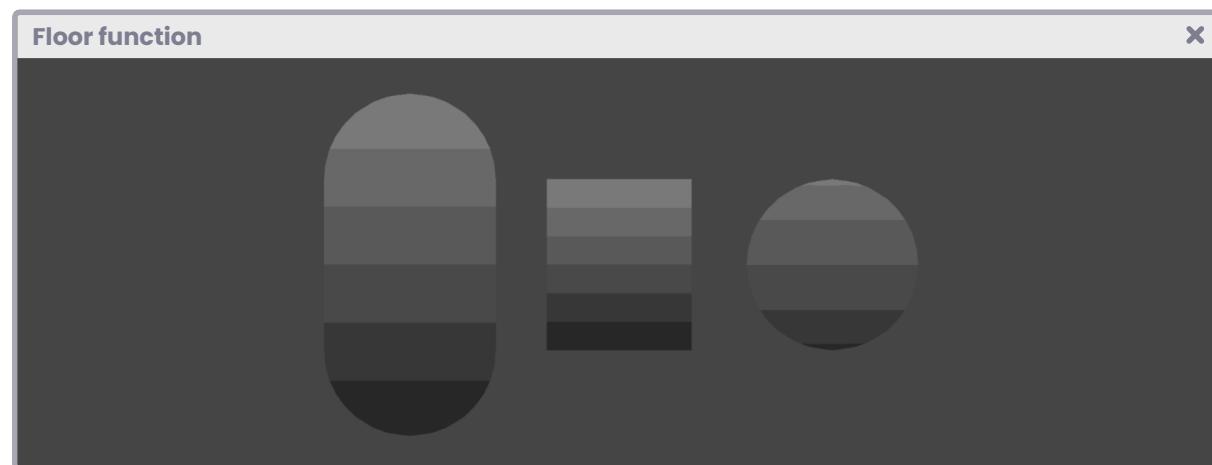


(Fig. 4.1.4b)

To add solid color blocks, we can simply multiply the operation by a certain number of sections and then divide by a decimal value.

As for gamma, we 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);

    // 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 0.17f of gamma)*

The implementation principle is the same for a toon shader, with the difference that we use global illumination in the calculation instead of the V-coordinate.

In the second chapter of this book, we will review the custom lighting implementation in detail.

## 4.1.5. | Step and Smoothstep function.

**Step** and **smoothstep** are quite similar functions, in fact, both have an argument called "edge" in charge of differentiating the return between two values.

We will begin our study in the compression of **step** to then approach the operation of **smoothstep**, which has a more elaborate structure than the previous one.

According to the official documentation at NVIDIA;

> *Step can return 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 function                                    ✕

float step (float x, float edge)
{
    return edge >= x;
}

float2 step (float2 x, float2 edge);
float3 step (float3 x, float3 edge);
float4 step (float4 x, float4 edge);
```

Exemplifying the previous statement, we can perform a simple operation on the fragment shader stage to understand how it works.
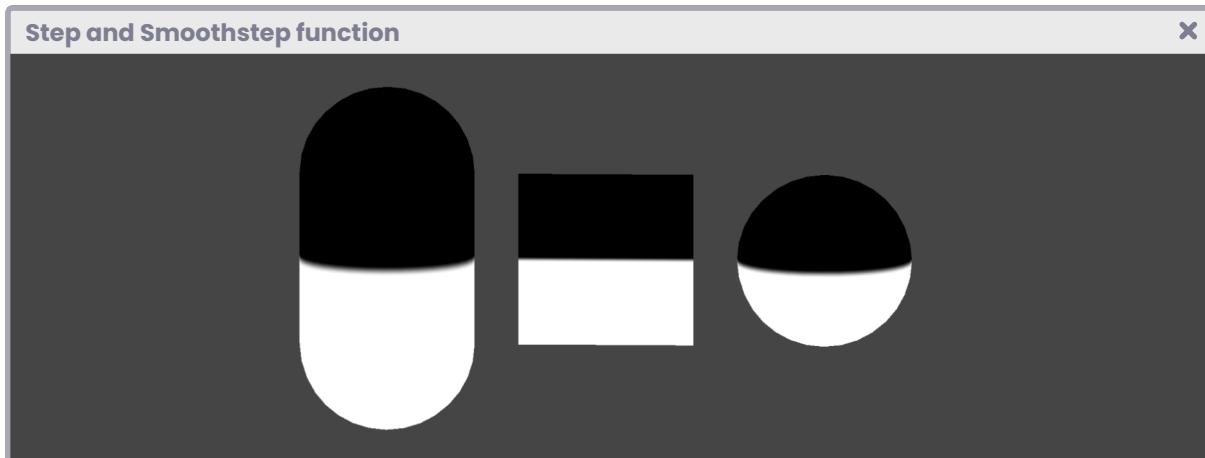
```
Step and Smoothstep function                                    ✕

 fixed4 frag (v2f i) : SV_Target
 {
     // add the color edge
     float edge = 0.5;
     // let's return to RGB color
     fixed3 sstep = 0;
     sstep = step (i.uv.y, edge);


     // fixed4 col = tex2D (_MainTex, i.uv);


     return fixed4(sstep, 1);

 }
```

We started by declaring a three-dimensional vector called "**sstep**" in the above operation. You can see its graphical representation in Figure 4.1.5a. As an argument, we have used the V coordinate of the UV and 0.5 as "edge." Finally, we 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 the edge will return "one" (white color) and "zero" in the opposite case (black color). If we modify the argument "edge" between these values, it could be balanced to one side or the other.

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 function** ✕

```
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, we could add a new variable to determine the amount of interpolation between the return values.

**Step and Smoothstep function** ✕

```
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 (i.uv.y, edge);
    sstep = smoothstep((i.uv.y - smooth), (i.uv.y + smooth), edge);

    // fixed4 col = tex2D (_MainTex, i.uv);

    return fixed4(sstep, 1);

}
```

In the previous exercise, we could modify the value of "smooth" between 0.0f and 0.5f to obtain different levels of interpolation.



**Step and Smoothstep function** ✕
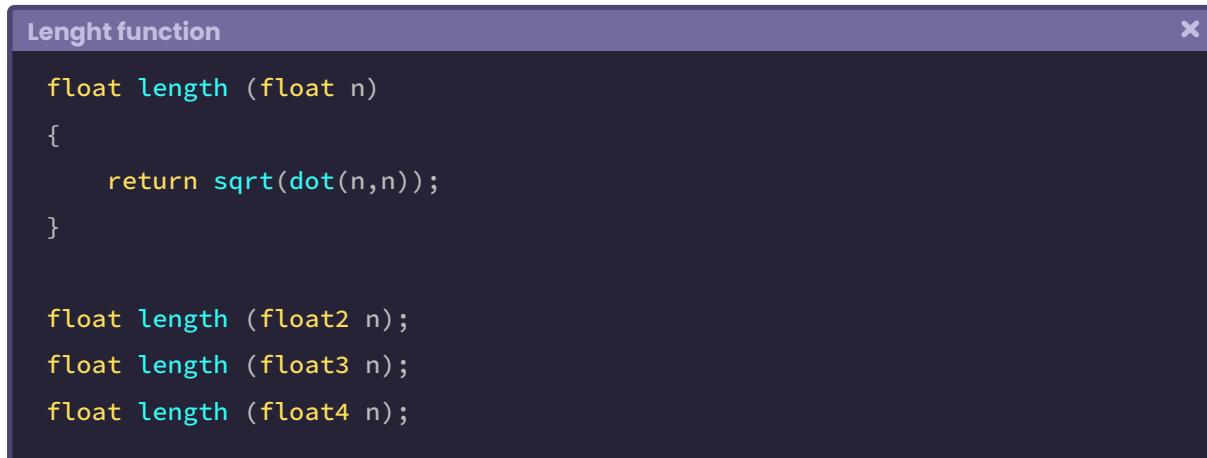
*(Fig. 4.1.5b. Smooth is equal to 0,1f)*

## 4.1.6. | Length function.

As its title mentions, length refers to the magnitude that expresses the distance between two points. This function is handy when creating geometric shapes, e.g., we can generate circles or polygonal shapes with rounded edges.

Its syntax is as follows:

**Lenght function** ✕

```
float length (float n)
{
    return sqrt(dot(n,n));
}

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

As usual, we will create a new shader type, "Unlit Shader," which we will call **USB_function_LENGTH**. We will use some functions to represent a circle in our program this time. We will start by adding properties that we will use later to enlarge, center, and smooth the shape.
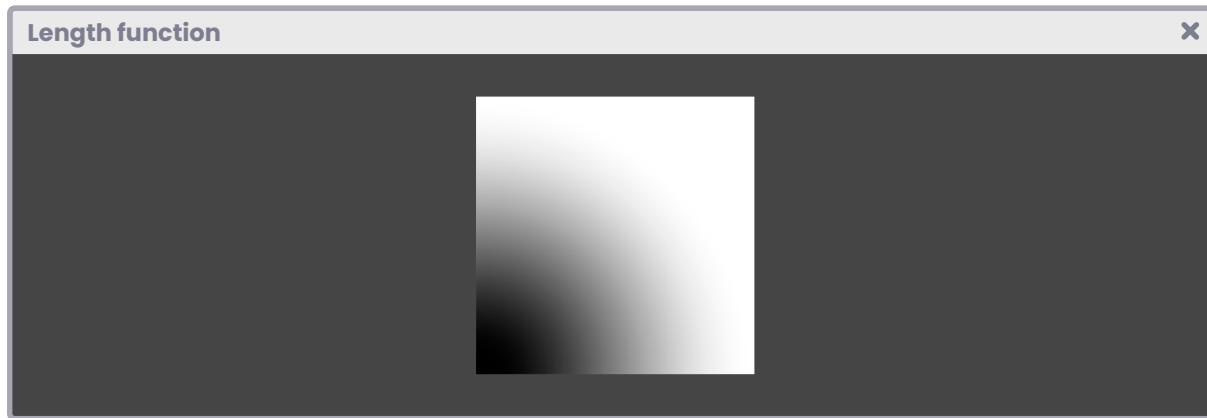
```
Lenght function                                                              ✕

 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

     {

         …

         Pass

         {

             …

             float _Smooth;

             float _Radius;

             float _Center;

             …

         }

     }

 }
```

To create a circle, we simply calculate the magnitude of the UV coordinates and subtract the radius. Its representation would look like the following:
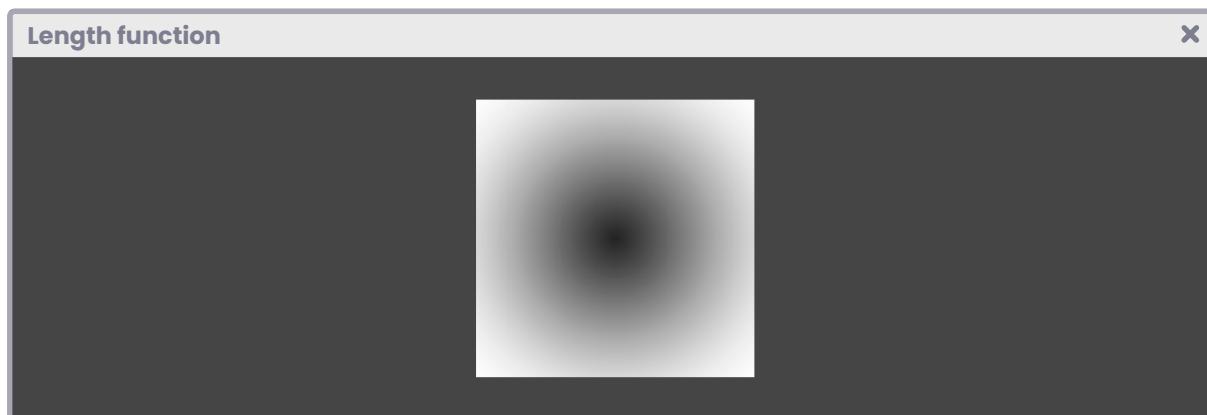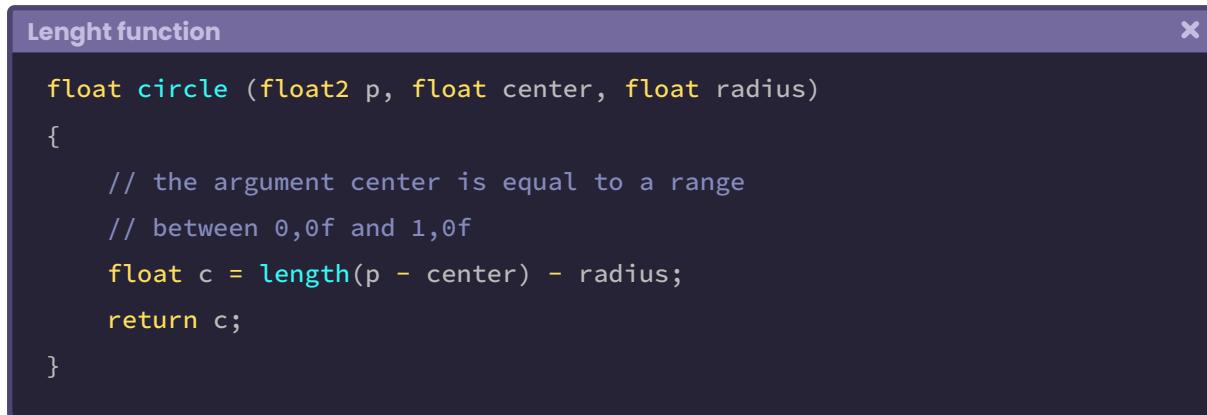
```
Lenght function                                                              ✕

 float circle (float2 p, float radius)

 {

     // let's create the circle

     float c = length(p) - radius;

     return c;

 }
```

However, as we can see in Figure 4.1.6a, the previous operation returns a blurred circle starting at the point 0,0f and ending at 1,0f, and the result we are looking for corresponds to a centered and more compact shape.

**Length function**



*(Fig. 4.1.6a. The circle fades as the value of the UV coordinates increases)*

For this purpose, we can add a new argument in the function that allows us to center the circle on the object to which we are applying the material.

**Lenght 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 to 0,5f)*

However, the circle will still be blurred. If we want to control the amount of blurring, we can use the smoothstep function, which, as we already know, generates a linear interpolation between two values.

```
Lenght function                                              ✕

  float circle (float2 p, float center, float radius, float smooth)
  {
      float c = length(p - center);
      return smoothstep(c - smooth, c + smooth, radius);
  }
```
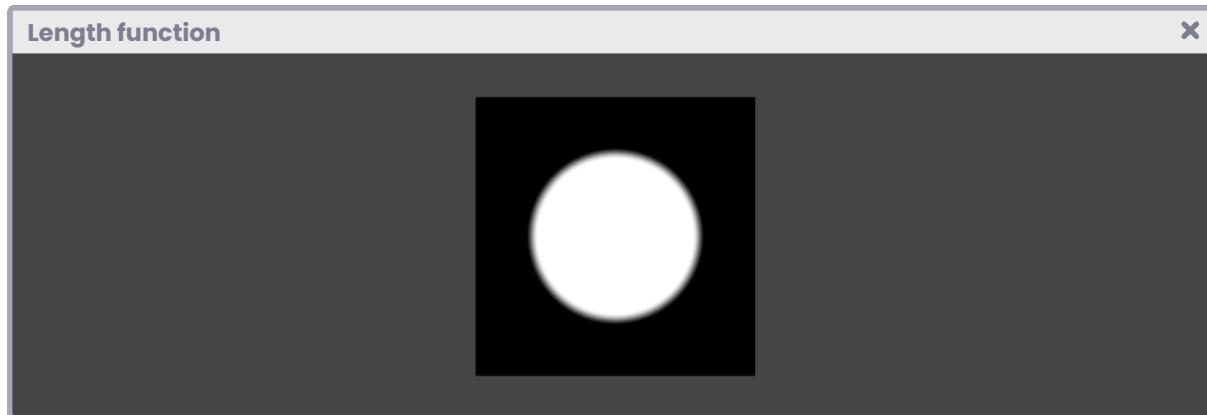
In the previous operation, we have added a new argument called "smooth," which will allow us to control the amount of blurring. Then we can apply this function in the fragment shader stage as follows.

```
Lenght 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 we can see, we 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).
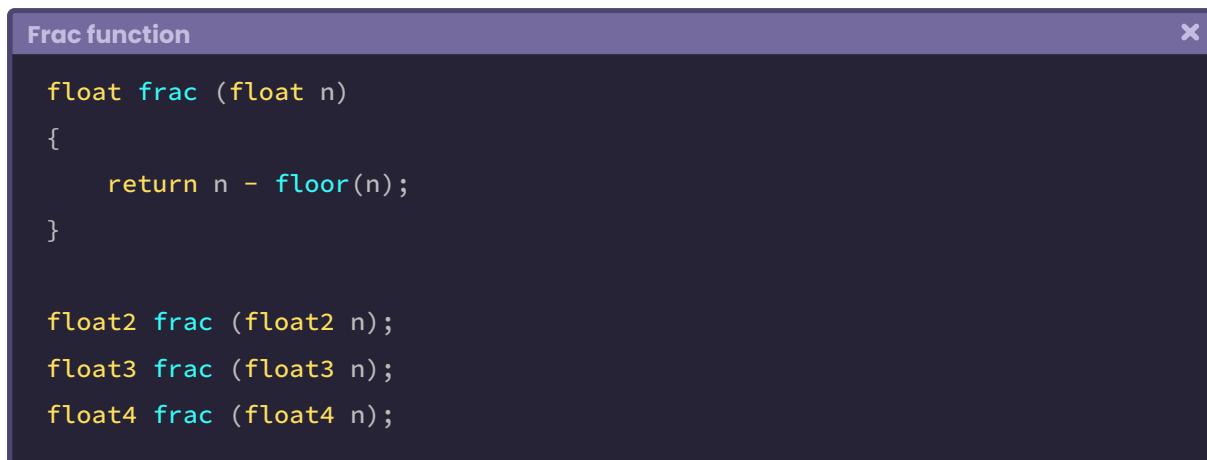
*(Fig. 4.1.6c. Radius 0,3f, Center 0,5f and Smooth 0,023f)*

## 4.1.7. | Frac function.

This function returns the fraction of a value, that is to say, its decimal values, e.g., frac of 1.534f returns 0.534f; why? It is due to the operation performed in the function.
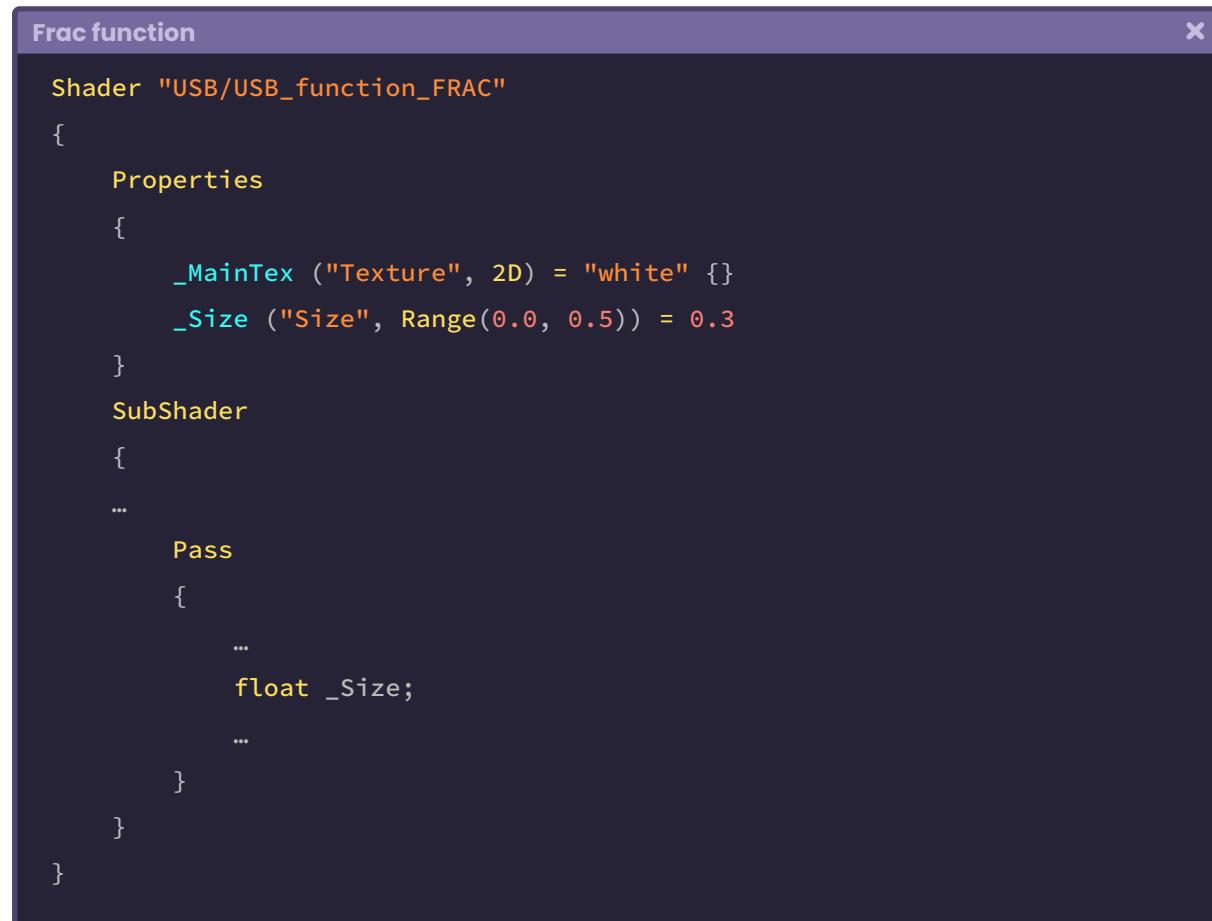
frac (3,27) = 0,27f          It's the same as 3,27f - 3.
frac (0,47) = 0,47f
frac (1,0) = 0,0f

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);
```

We could use Frac in multiple operations like noise calculation, random repeating patterns, and much more.

To understand the concept, we will do the following: We will create a new shader of type "Unlit Shader," which we will call **USB_function_FRAC**. We will start adding a property called "size" that we will use later to increase or decrease the size of a circle.
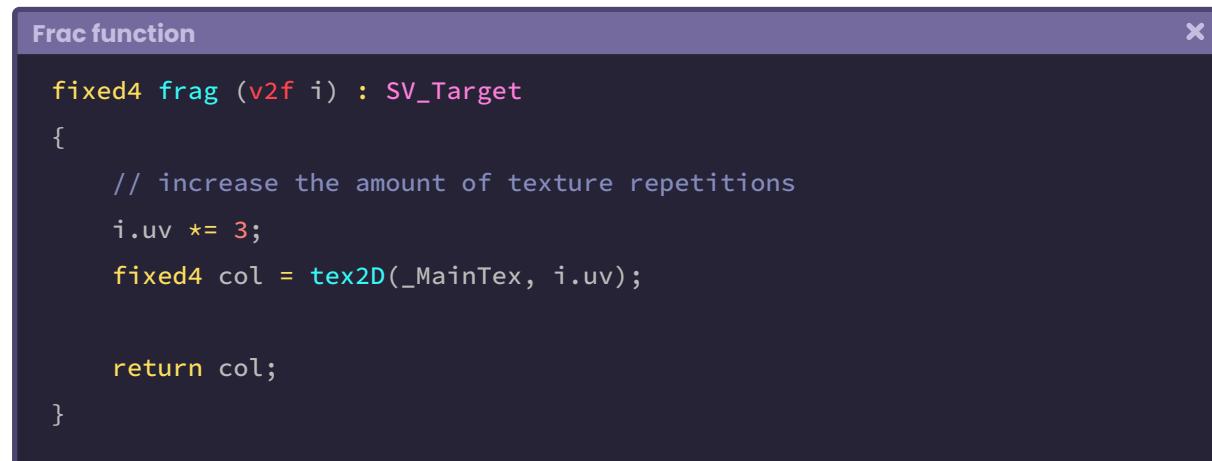
**Frac function** ✕

```
Shader "USB/USB_function_FRAC"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Size ("Size", Range(0.0, 0.5)) = 0.3
    }
    SubShader
    {
    …
        Pass
        {
            …
            float _Size;
            …
        }
    }
}
```

The first operation we will perform is to multiply the UV coordinates to obtain a repeating pattern in the fragment shader stage.
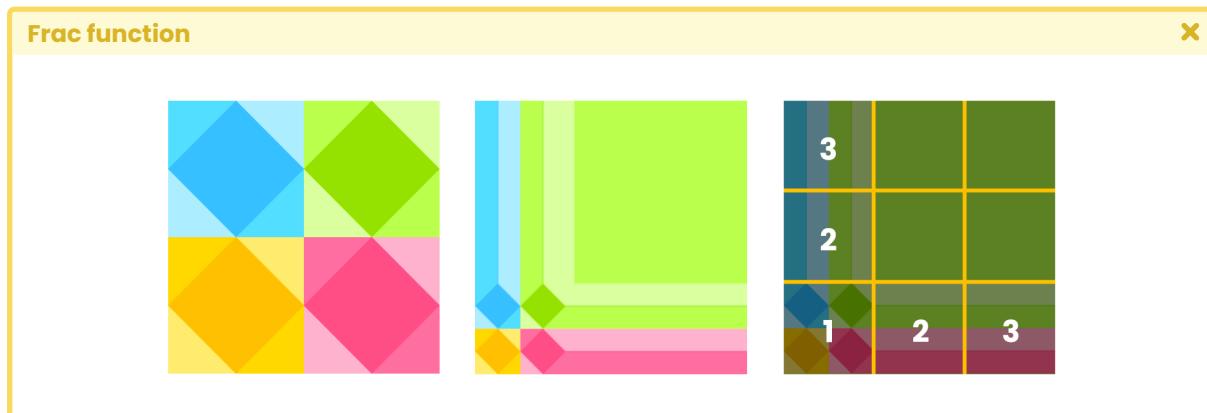
**Frac function** ✕

```
fixed4 frag (v2f i) : SV_Target
{
    // increase the amount of texture repetitions
    i.uv *= 3;
    fixed4 col = tex2D(_MainTex, i.uv);

    return col;
}
```
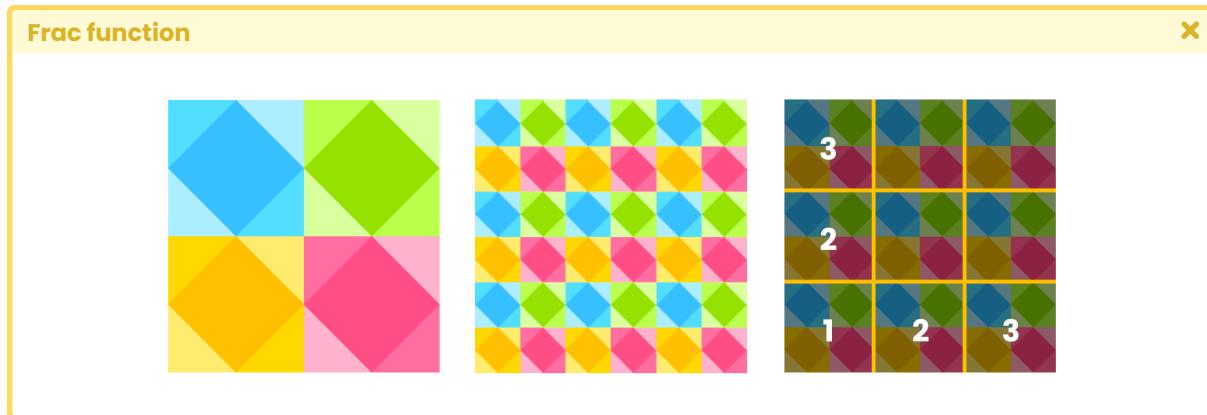
If we go back to Unity and assign a texture set to "clamp," we will obtain a similar result in figure 4.1.7a. We can notice that the edge texels in the image are stretched along with itself.



(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, we could use the "**frac**" function to return the fractional value of the UV coordinates to generate a defined repeating pattern.

```
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**                                                       ✕
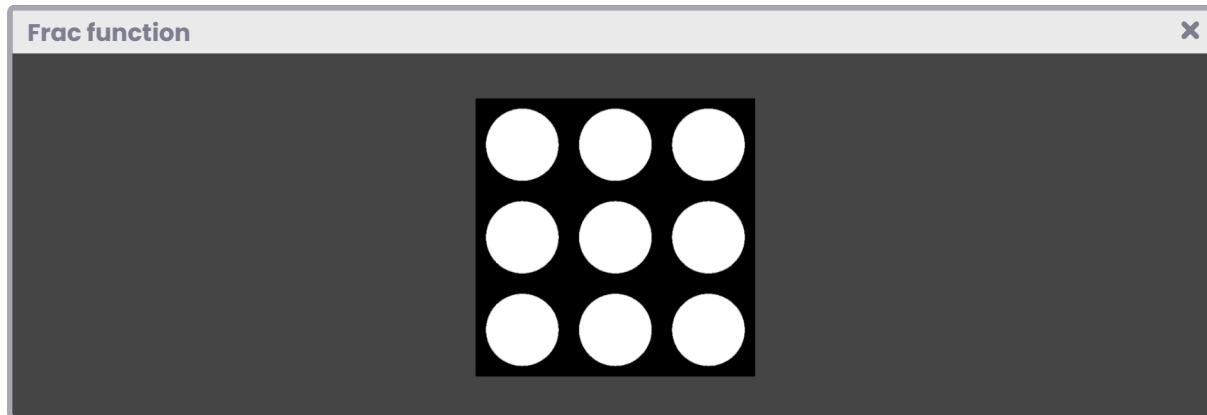


(Fig. 4.1.7b)

It is worth mentioning that performing this operation on a texture is not very useful, since we can easily set it to "Repeat" mode from the Inspector.

For a more practical example, let's 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 we pay attention to the return value, we will notice that the same channel is being used for the output colors in RGB (wCircle.xxx). If we modify the value of the **_Size** property, we can increase or decrease the size of the circles.

**Frac function**
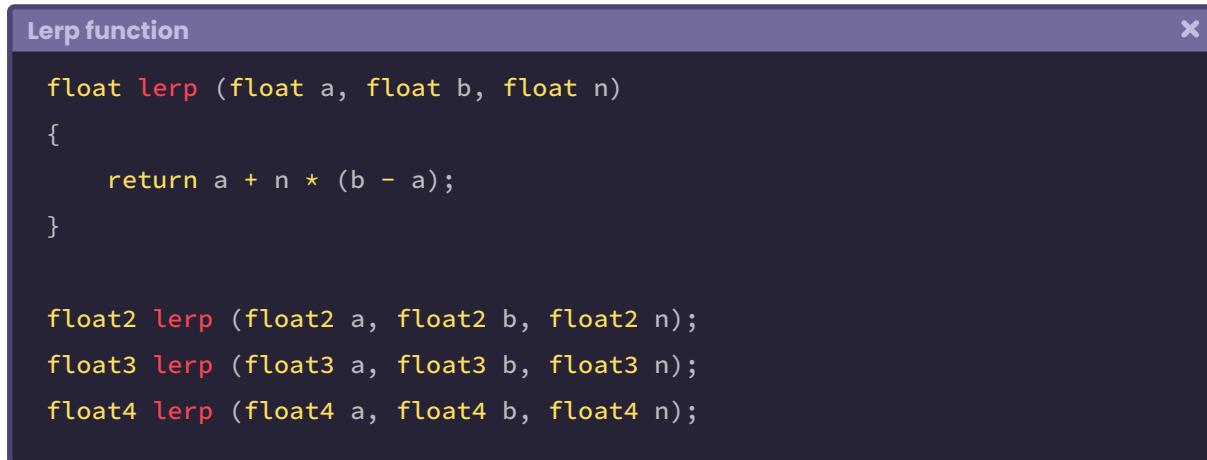
*(Fig. 4.1.7c. The property _Size has been configured in 0,3f)*

## 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., we could use lerp on one of our 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);
```

We will create a small shader of type "Unlit Shader" to exemplify the function, which we will call **USB_function_LERP**. We will start by declaring two textures that we will use later as "skins" in effect, plus a numeric range to perform the cross-fading.

**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
    {
        …
        Pass
        {
            …
            sampler2D _Skin01;
            float4 _Skin01_ST;
            sampler2D _Skin02;
            float4 _Skin02_ST;
            float _Lerp;

            …
        }
    }
}
```

Since we will use two textures, it will be necessary to use UV coordinates in each case. These must be declared in both the vertex input and output for two main reasons:

1. Because the textures will be affected by the "tiling and offset" through the TRANSFORM_TEX function.
2. Because we 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;
    // we will use the UV coordinates in the fragment shader stage
    float2 uv_s01 : TEXCOORD0;
    float2 uv_s02 : TEXCOORD1;
};

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;
}
```
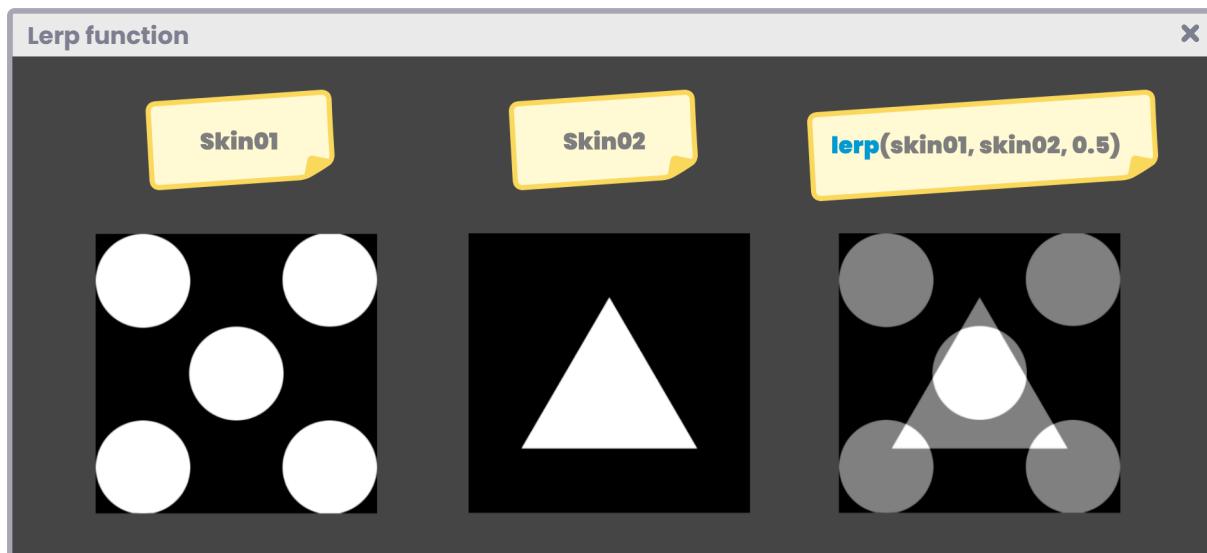
In the fragment shader stage, we can declare two four-dimensional vectors, use the tex2D function in each case, and then perform a linear interpolation between them using the lerp function.

**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 we pay attention to the _Lerp property, we notice that its value has been limited between 0.0f and 1.0f. By default, it has the value 0.5f; therefore, if we assign two different textures to each property of _Skin[n], the result will be equal to transparency or fading of 0.5f in each case.



*(Fig. 4.1.8a. Lerp between two textures)*

## 4.1.9. | Min and Max function.

On the one hand, "**min**" refers to the minimum value between two vectors or scalars, while the "**max**" is the opposite. We will use these functions frequently in different operations, e.g., we can use **max** to calculate the diffusion on an object, returning the maximum between "zero" and the dot product between the normals of the mesh and the direction of the light.

Its syntax is as follows:

**Min and Max function** ✖

```
// if "a" is less than "b", it returns "a", otherwise returns "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 function** ✖

```
// if "a" is greater than "b", it returns "a", otherwise returns "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);
```

We 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 we will frequently use in animating properties for our 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 function of the **Time.timeSinceLevelLoad**. 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 equal to the operation "**cos(_Time.y)**"; 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);
```
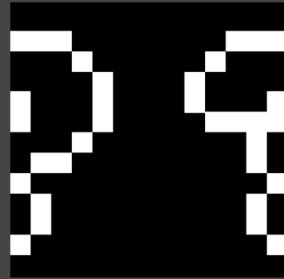
We could use the _Time function to generate the Super Mario icons' offset animation to deepen the concept. For this purpose, we would have to add time to the U coordinate of the UV in the fragment shader stage.

```
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)*
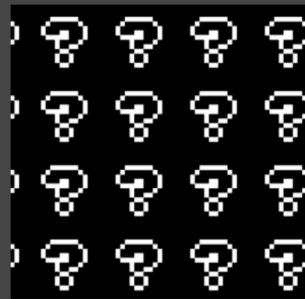
Or we could also generate a rotation animation using _SinTime and _CosTime, adding both U and V motion.

**Timing and animation** 175 ✖

```
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;
}
```

**Timing and animation** ✖



*(Fig. 4.2.0b. Offset rotation in both U and V on a Quad. The tiling is equal to 4)*

# Lighting, shadows 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 our functions for lighting, shadow, and surface calculations, we begin by detailing the theory behind each concept and then move on to their implementation in the HLSL language.

It is worth mentioning that in Unity we have some predefined programs that facilitate lighting calculations for a surface (e.g. Standard Surface, Lit Shader Graph) however, it is essential to continue our study using **Unlit Shader** type Shaders**,** since being basic color models, they allow us to implement our own functions, and thus, obtain the necessary understanding generating 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, we learned the analogy between the property of a polygonal object and a semantic. Likewise, we could see how the latter is initialized within a **struct.**

In this section, we will detail the steps necessary to configure the *normals* of our object and transform its coordinates from *object-space* to *world-space.*

As we already know, if we want to work with our object *normals* then we 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, in the *normal* case, has a "zero" default value since it is a direction in space.

Once we have declared our object *normals* as input, we must ask the following question: are we going to need to pass these values to the *fragment shader stage?* If the answer is yes, then we will have to declare the normals once again, but this time as output.

This analogy applies equally to all the inputs and outputs that we use in our program.

```
Configuring inputs and outputs.                                        ✕

struct v2f
{
    ...
    float3 normal : TEXCOORD1;
};
```

Why have we declared normals in both **vertex input** and **vertex output?** This is because we 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, we must use a semantic that can store at least three coordinates of space. That is why we have used TEXCOORD1 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 *vertex output,* we can go to the *vertex shader stage* to connect them.

```
v2f vert (appdata v)
{
    v2f o;

    …

    // we connect the output with the input
    o.normal = v.normal;

    …

    return o;
}
```

What did we just do? Basically, we connected the *normals* output of **struct v2f** with the *normals* input of **struct appdata.** Remember that *v2f* is used as an argument in the *fragment shader stage,* this means that we can use the object *normals* as a property in this stage.

To illustrate this concept we will create an example function, which we will call **unity_light** and will be responsible for calculating the lighting on a surface in the *fragment shader stage.*

In itself, this function is not going to perform any actual operation; however, it will help us to understand some factors that we should know.

```
void unity_light (in float3 normals, out float3 Out)
{
    Out = [Op] (normals);
}
```

According to its declaration, we can observe that *unity_light* is an empty function; that has two arguments: The object normals and the output value.

Note that all lighting calculation operations require the normal as one of their variables, why? Because without this, the program will not know how light should interact with the object surface.

We will 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 our light in black
    half3 light = 0;
    // initialize our function and pass the vectors
    unity_light(normals, light);

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

Since *unity_light* corresponds to an empty function, two three-dimensional vectors have been created which will be used as arguments in the function.

The first one corresponds to the normals, and the second, the output value, to the lighting output.

If we look closely, we will notice that the *normals* output that was declared in the previous stage (i.normal) is in *object-space,* how do we know this? We 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, we will have to transform the space coordinates of the normals in the *fragment shader stage.*

To do this, we do the following:

**Configuring inputs and outputs.** ✕

```
// let's create a new function
half3 normalWorld (half3 normal)
{
    return normalize(mul(unity_ObjectToWorld, float4(normal, 0))).xyz;
```

*Continued on next page.*

180
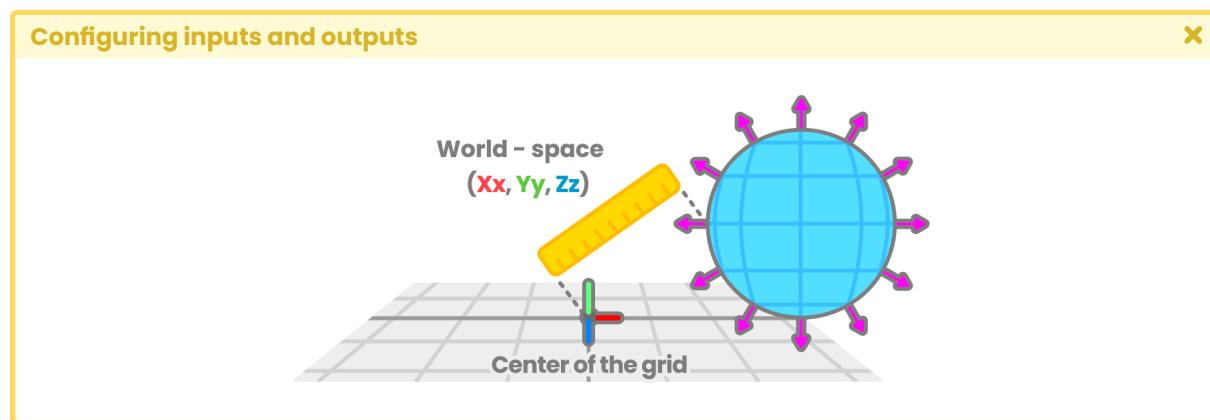
```
    }

    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 coordinates transformation to the normals. In its process, it uses the unity_ObjectToWorld *matrix,* which allows us to go from *object-space* to *world-space.*



*(Fig. 5.0.1a)*

A question that frequently arises in the transformation of spaces is, why are the *normals* within a four-dimensional vector in the **normalWorld** multiplication function?

As we already know the *normals* 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, we will obtain a new direction for the normals in their four XYZW channels.

Therefore, we must make sure to assign the value "zero" to the W component (float4(normal. xyz, 0)), because, by arithmetic rules, any number multiplied by zero equals zero, and thus, the *normals* can be calculated without errors.

181

The process mentioned above can also be carried out in the *vertex shader stage*. The operation is basically the same, except that if we 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.
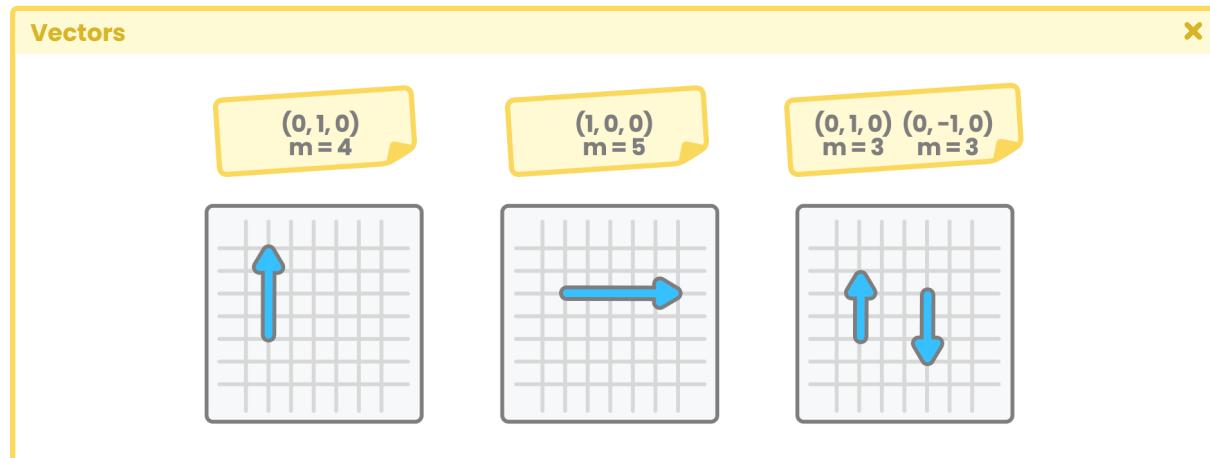
**Configuring inputs and outputs.** ✕

```
v2f vert (appdata v)
{
    …
    o.normal = normalize(mul(unity_ObjectToWorld, float4(v.normal, 0))).xyz;
    …
}
```

## 5.0.2. | Vectors.

Before we start implementing lighting in our programs, we 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.
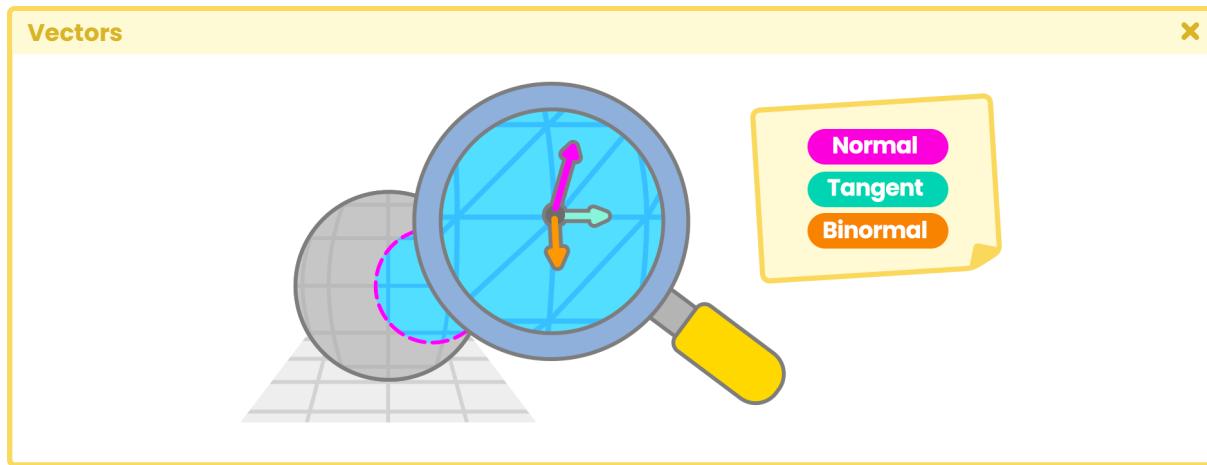
**Vectors** ✕



(Fig. 5.0.2a)

In the example 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 given

scalar magnitude containing 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.
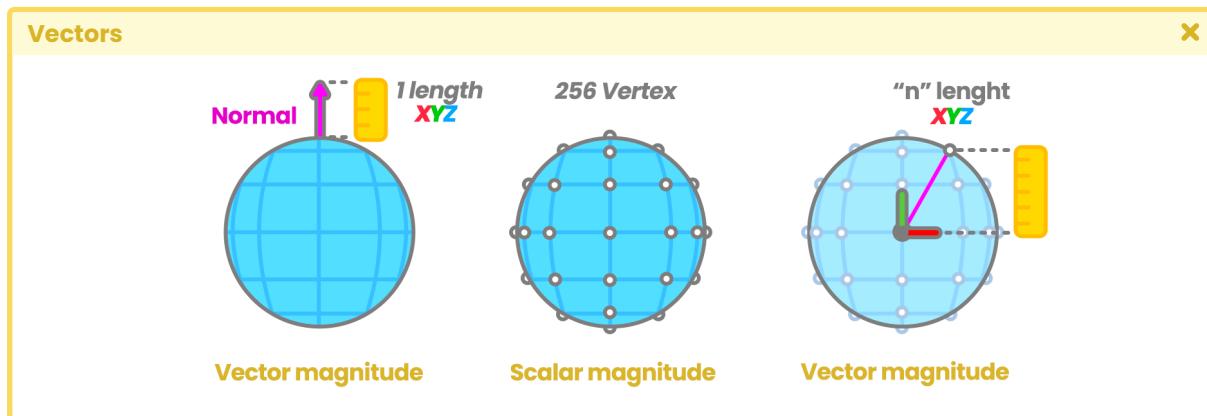
Because vector magnitudes have direction, we can conclude that they have a starting point and an endpoint, and are also included within a three-dimensional coordinate system.

The *normals* of an object are vector magnitudes, why? Because they have a length (which equals *one* in most cases) and a direction in space that is measured in three dimensions. Likewise, *tangents* and *binormals* 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 we try to calculate the distance between the center of the object and the position of a vertex, we will obtain a vector magnitude, since, by definition, our 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, etc.).

### 5.0.3. | Dot product.

The dot product is an operation that we will use frequently in the calculation of illumination and reflection since it allows us 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 one and minus one [1, -1].

$$a \cdot b = \| a \| \| b \| \cos \theta$$

*(The "point" (·) between a and b refers to the dot product. The "|| x ||" symbol refers to the vector magnitude.)*
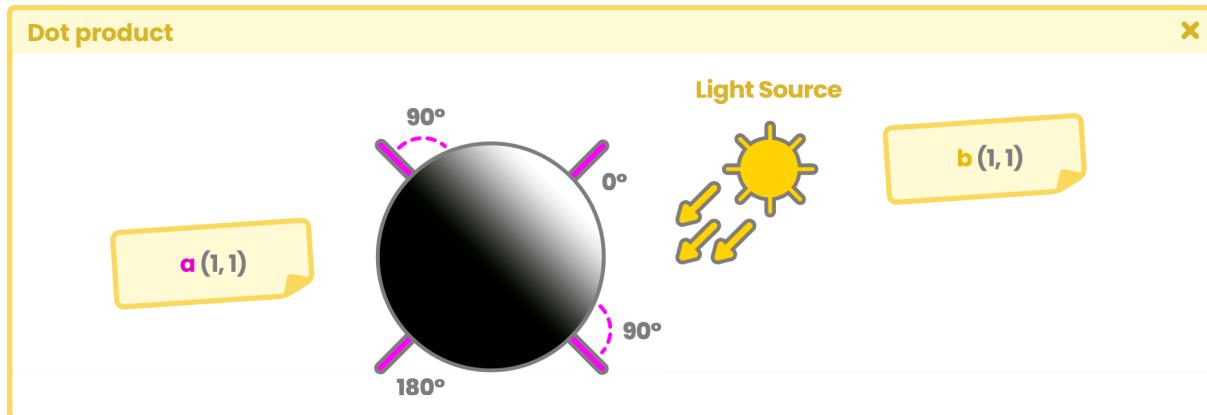
In the previous function: when the angle between vectors *a* and *b* equals 0°, the dot product will return "one". In turn, when the angle equals 90°, the dot product will return "zero". Finally, when the angle equals 180°, the dot product will return "minus one".

How does this operation work? To understand it, we must calculate the cosine of the angle between these two vectors.

$$\cos \theta = \frac{a \cdot b}{\| a \| \| b \|}$$

To understand the above function, let's assume that we have two two-dimensional vectors with the following values.

- vector a (1, 1)
- vector b (1, 1)

Light Source

90°

0°

90°

180°

a (1, 1)

b (1, 1)

*(Fig. 5.0.3a)*

Both vectors are equal in both direction and magnitude. To calculate the dot product between $a$ and $b$ we have to multiply $a_x$ times $b_x$ then $a_y$ times $b_y$ and finally add the values.

$$a_x * b_x = 1$$
$$a_y * b_y = 1$$
$$1 + 1 = 2$$

Consequently, the product point between $a$ and $b$ equals 2. To show this, let's replace these values in the function mentioned above.

$$Cos\ \theta\ =\ \frac{2}{\|\,a\,\|\,\|\,b\,\|}$$

Then we must calculate the magnitude of both $a$ and $b$. To do this, we must 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*" we can deduce that its value is the same.

$$\| b \| = \sqrt{1x^2 + 1y^2}$$
$$\| b \| = \sqrt{2}$$

If we multiply the magnitude of *a* by the magnitude of *b*, we get the following value.

$$\| a \| * \| b \| = \sqrt{2} * \sqrt{2} = \sqrt{2 * 2} = \sqrt{2^2} = 2$$
$$\| a \| * \| b \| = 2$$

As we can see, the factor between the magnitude of a and b equals two.
Again, we are going to replace this value in the previous function to understand it better.

$$Cos\ \theta\ =\ \frac{2}{2}$$

$$Cos\ \theta\ =\ 1$$
$$\theta\ =\ Cos^{-1}(1) = 0^0$$
$$Cos\ (0^0)\ =\ 1$$

So, the cosine of zero degrees equals one, therefore, if we carry out the initial operation it would be as follows.

$$// a \cdot b = \| a \| \| b \| Cos\ \theta$$
$$a \cdot b = 2 * 1 = 2$$

The result of the product point between *a* and *b* equals "two". Now, if we normalize this value, we will get a magnitude of one.

$$normalize(a \cdot b) = 1$$

What is the use of all this explanation? When we implement lighting in our shader, we have to use two vectors: one for the light calculation and the other for the calculation of the normals. 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 illumination, the dot product will return one, and for those that are on the opposite side will be minus one.

### 5.0.4. | Cross product.

The cross product (also known as the vector product) is an operation that, unlike the dot product, returns a three-dimensional vector that is perpendicular to its arguments.

To understand this concept, we will take two vectors: *a* and *b,* and position them in space as follows.

- vector *a* (1, 0, 0)
- vector *b* (0, 1, 0)



*(Fig. 5.0.4a)*

In the example above, the cross product generates a third vector named "*c*" with new space coordinates.

To understand the operation of the cross product, we must pay attention to the next operation.

$$|| \, a \times b \, || = || \, a \, || \, || \, b \, || \, Sin \, \theta$$

The magnitude of the resulting vector will be related to the function of *sin.*

Taking into consideration the vectors *a* and *b* mentioned above, we can calculate the cross product from a determinant matrix between both vectors.

- vector *a* (1, 0, 0)
- vector *b* (0, 1, 0)

We calculate the third vector from the *a* and *b* values.

$$\text{vector c} = x \, [(a_y * b_z) - (a_z * b_y)] \, y \, [(a_z * b_x) - (a_x * b_z)] \, z \, [(a_x * b_y) - (a_y * b_x)]$$

By replacing the values, we get.

$$\text{vector c} = x \, [(0 * 0) - (0 * 1)] \, y \, [(0 * 1) - (1 * 0)] \, z \, [(1 * 1) - (0 * 0)]$$
$$\text{vector c} = x \, [(0 - 0)] \, y \, [(0 - 0)] \, z \, [(1 - 0)]$$

- vector c = (0, 0, 1)
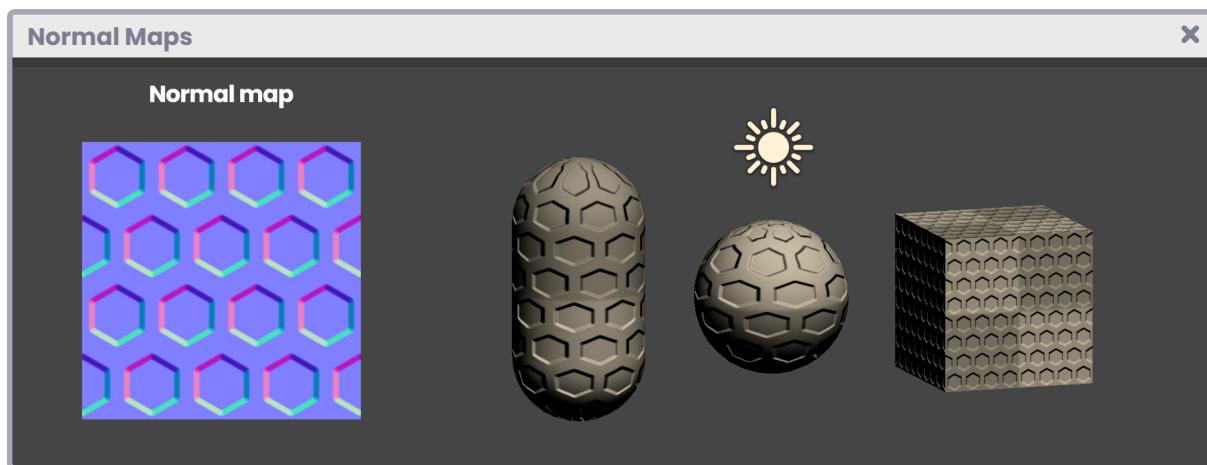
In conclusion, the resulting vector is perpendicular to its arguments.

Later, we will use the *cross product* 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 some frame of reference. To do this, we can store each vertex within a space coordinate called *tangent-space.* This type of space is used for the calculation of illumination on the object surface.



*(Fig. 6.0.1a)*

To generate our *normal map* we have to use three normalized vectors, which correspond to **tangents, binormals** and **normals.** These three, together, form a matrix called TBN (T for tangent, B for binormal and N for normal).

In section 5.0.1 we talked about how to transform *normals* into *world-space* using the matrix *unity_ObjectToWorld*.

Similarly, we will use the TBN matrix to pass from one space to another, and thus, be able to transform both the lighting and our normal map, from world-space to tangent-space. The graphical representation of the TBN array 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, we will create a new **Unlit Shader**, which we will call **USB_ normal_map.** The application of the normal map will start by adding a texture property in our program.

**Normal Maps**                                                                  ✕

```
Shader "USB/USB_normal_map"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _NormalMap ("Normal Map", 2D) = "white" {}
    }
    SubShader { … }
}
```

In the properties' field we have declared a *2D type texture* property called **_NormalMap,** we will use this to apply our *normal map* dynamically, from the Inspector window.

Next, we must add the variables and/or connection vectors corresponding to the declared property.

```
Normal Maps                                                        ✕

 Pass

 {

     CGPROGRAM

     …

     sampler2D _MainTex;

     float4 _MainTex_ST;

     sampler2D _NormalMap;

     float4 _NormalMap_ST;

     …

     ENDCG

 }
```

Now, our normal map can be used inside the shader as a texture. Therefore, we must create the TBN matrix, for this, we will extract the *normals* and *tangents* from our object to the *vertex input,* called **appdata,** and then implement the NORMAL and TANGENT semantics.
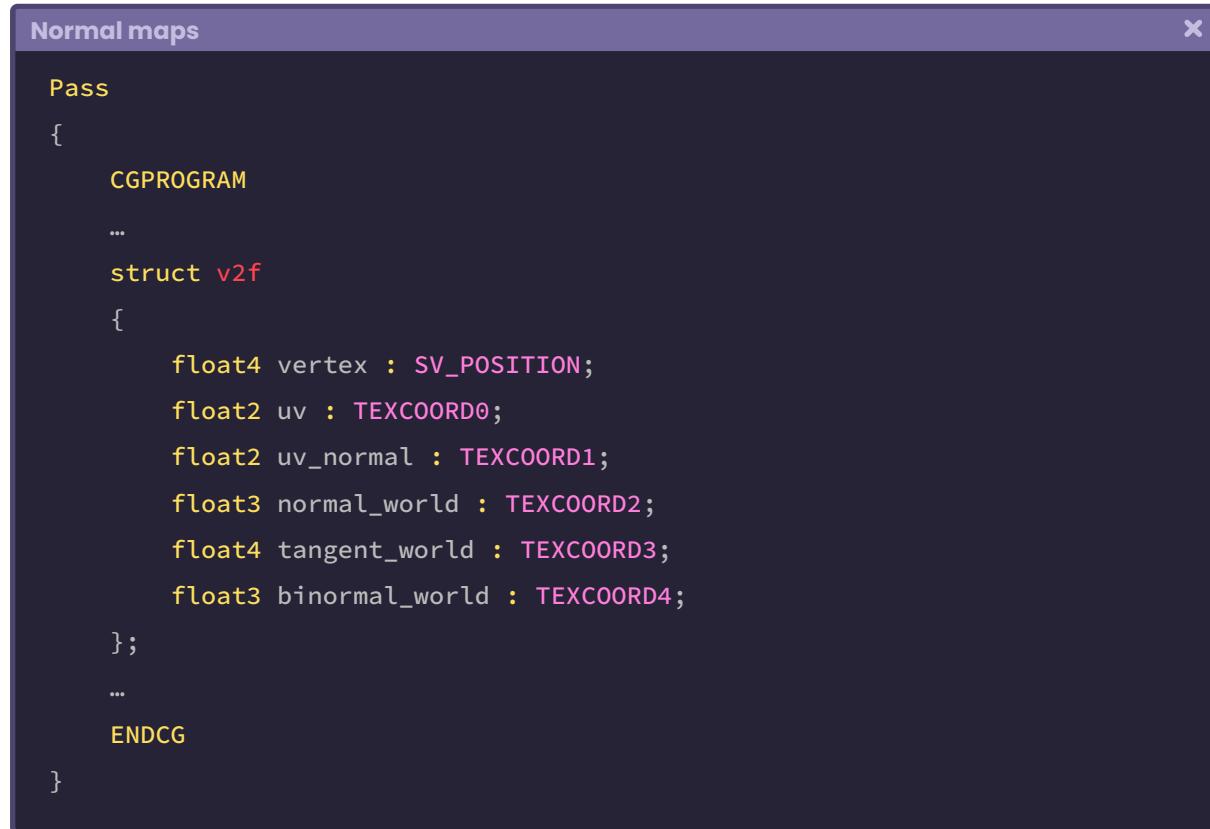
```
Normal maps                                                        ✕

 Pass

 {

     CGPROGRAM

     …

     struct appdata

     {

         float4 vertex : POSITION;

         float2 uv : TEXCOORD0;

         float3 normal : NORMAL;

         float4 tangent : TANGENT;

     };

     …

     ENDCG

 }
```

A factor to consider is that, up to this point, both *normals* and *tangents* are in *object-space*, and we need them to be transformed into *world-space* before being converted into *tangent-space.* To do this, we have to connect them in the *vertex shader stage,* and

**191**

then pass the *vertex output* to the *fragment shader stage.* Then, we must add the *normal, tangent,* and *binormal* coordinates in the **struct v2f** for their calculation in *world-space* in the following manner:

```
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, you cannot use the semantics NORMAL or TANGENT within the *vertex output*, this is because they do not exist for this process. In this case, we must use semantics that can store up to four dimensions in each of their coordinates. This is why we used the semantics TEXCOORD[n] in the example above.

Now, if we pay attention, we will notice that each of the properties has a coordinate with a different ID, e.g., **uv_normal** is assigned to TEXCOORD with its index in [1] while **binormal_world** has the index [4]. It is essential that the IDs have different values because, otherwise, we would perform operations on a duplicate coordinate system.

To transform the properties from *object-space* to *world-space,* we go to the *vertex shader stage* and perform the following operation:

**Normal maps** ✕
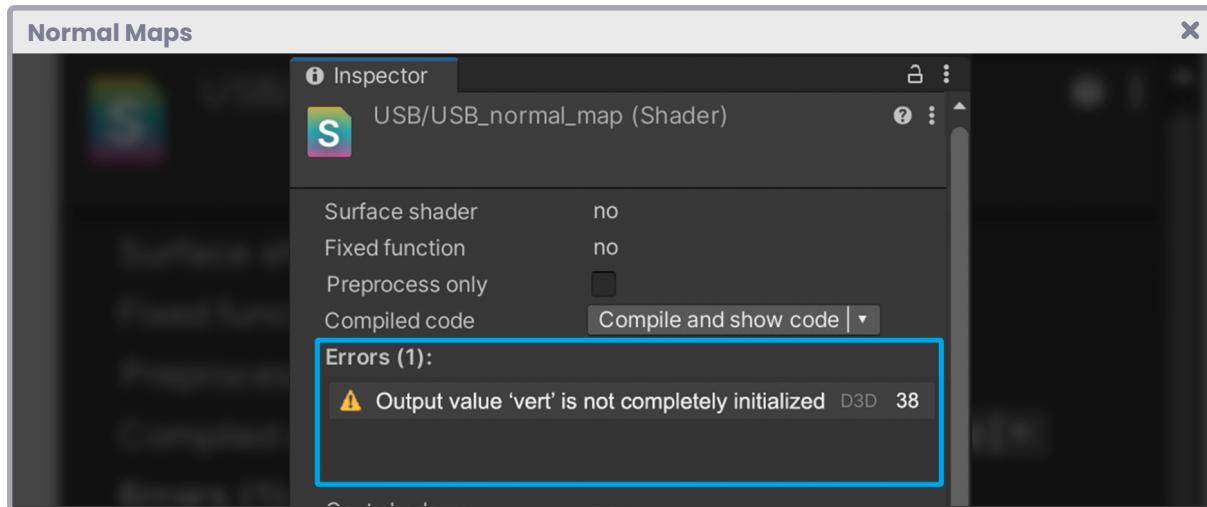
```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);

    // 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, we have started the operation by adding *tiling* and *offset* to the *normal map* UV coordinates through the TRANSFORM_TEX function, which is included in "UnityCg.cginc". Then, we multiply the four-dimensional matrix called *unity_ObjectToWorld* by the *normals* input to transform their space coordinates from *object-space* to *world-space.* The multiplication result is stored within the *normals* output called **normal_world,** which we will use later for the *per-pixel* calculation in the *fragment shader stage.*

Next, we multiply the *tangents* by the matrix *unity_WorldToObject* to inversely transform their coordinates from *world-space* to *object-space*, and finally calculate a perpendicular vector between the *normals* and *tangents,* using the **cross** function, which, refers to **cross product.** Its result is known as *binormal,* which is why we store it within 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 calculation of the normals in our shader. To confirm this, a warning will appear in the Unity console, and in addition, the shader will show a small error related to this point.

*(Fig. 6.0.1b)*

In this case, we have to use the macro UNITY_INITIALIZE_OUTPUT within the vertex shader as follows:

```
Normal maps                                                         ✕

 v2f vert (appdata v)
 {
     v2f o;
     UNITY_INITIALIZE_OUTPUT (v2f, o);
     o.vertex = UnityObjectToClipPos(v.vertex);

     …

 }
```

Now we have our inputs and outputs connected in the *vertex shader stage*. What we need to do next is generate the TBN array to transform the coordinates of the *normal map* from *world-space* to *tangent-space*. This process will be carried out in the *fragment shader stage.*

One factor we must consider when reading our *normal map* is that the XYZW coordinates are embedded within the RGBA channels. The RGBA colors or channels in Unity, have a range between zero and one $(0, 1)$, this means that, for example, the minimum value of the color red is "zero" and the maximum is "one" $[0, 1]$.

It is essential to understand this concept since, if we want to extract the normals from the *normal map,* these will have a color range between minus one and one $[-1, 1]$ so, the first thing we must do is change the scale of the range using the following function:

```
normal_map.rgb * 2 - 1;
```

To illustrate the above operation, we will do the following exercise: If we multiply a range between zero and one, by two, then the new value of the range will be between zero and two, why? Because zero times two, is zero, and one times two, is two.

```
0 * 2 = 0        Minimum color
1 * 2 = 2        Maximum color
```

As we can see, we have changed the range, now it goes from zero to two $[0, 2]$, however, if we subtract one, then our range will go from minus one to one $[-1, 1]$.

```
0 - 1 = -1       Minimum color
2 - 1 = 1        Maximum color
```

Performing this operation is essential for our *normal map* to work correctly. Another factor that we must consider is that *normal maps* are much heavier than a normal texture, which is why we have to compress them to reduce their GPU graphic load.
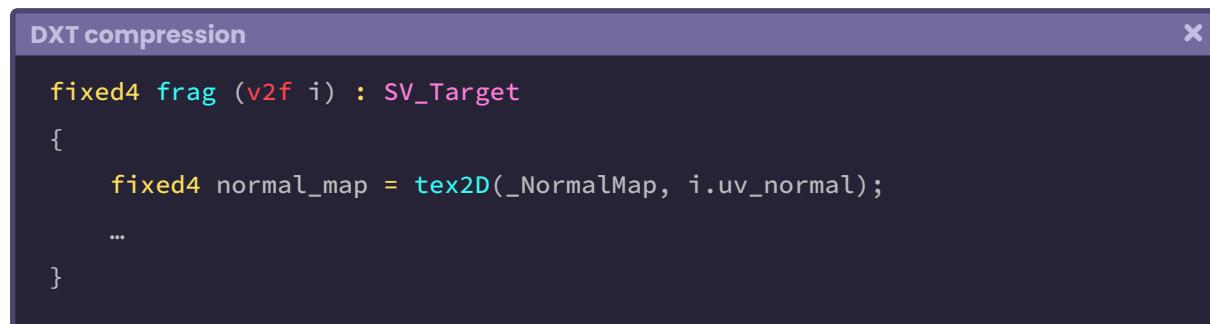
## 6.0.2. | DXT compression.

*Normal maps* are very useful when generating detail in our objects, however, they are very heavy and produce a significant graphic load on the *GPU*. Likewise, if we 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 our 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, then compressed using only two of their channels (AG), allowing the normal map to be optimized to ¼ resolution.

To understand this concept, we first have to calculate the *normal map* and its UV coordinates in the *fragment shader stage.* For this, we will use the **tex2D** function.

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);

    …
}
```

Like the default *col* vector, we have generated a four-dimensional vector (RGBA) called **normal_map**. The *normal map* and its UV coordinates have been stored in it. Its RGBA channels currently have a range between zero and one [0, 1]. This will have to be modified because the *normal map* has a range between minus one and one [-1, 1]. To do this, we will create a new function which we will call **"DXTCompression"** and position it over 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
}
```
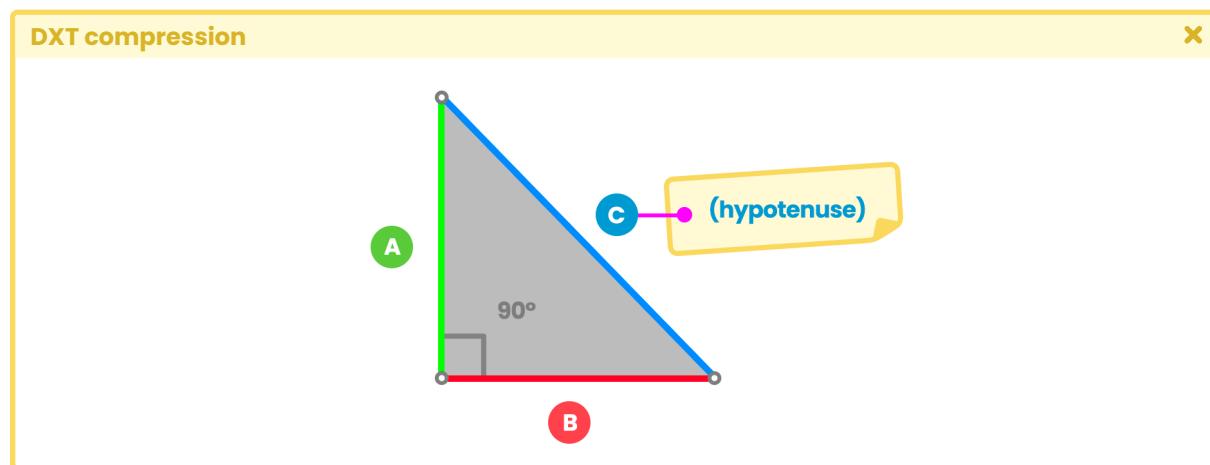
Four main things are happening in the previous function:

1.  The function returns a three-dimensional vector according to a condition.
2.  It has been defined 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 map* will be encoded in RGB instead.
3.  In the #else condition, we have generated DXT compression using only two channels. If we pay attention, we will notice that we have replaced the "red" channel with the "alpha" channel (normalMap.a), and then we have used the "green" channel for the second channel..
4.  The third channel in the vector (normalCol.b) has been discarded, but then calculated independently using the function:

sqrt(1 - (pow(normalCol.r, 2) + pow(normalCol.g, 2)))

Why are we using this feature? The answer lies in the Pythagoras theorem. As we already know:

> *In every right triangle, the square of the hypotenuse equals the sum of the squares of the other two sides.*



**DXT compression**

A

90°

C ── (hypotenuse)

B

*(Fig. 6.0.2a)*

Consequently.

$$C^2 = A^2 + B^2$$

A vector in space generates a right triangle, so likewise, if we want to calculate the magnitude of a three-dimensional vector, we will have to do it through the Pythagoras theorem.

$$\| V \|^2 = A^2 + B^2 + C^2$$

When we transform the space coordinates for the normal, tangent and binormal, we use the "*normalize*" function which returns a vector with a magnitude of one, likewise, the magnitude of the vector that we use for the coordinate B or Z; which is the same, equals one, then the operation would be as follows:

$$1 = X^2 + Y^2 + Z^2 \qquad\qquad \text{Or in its variation } R^2 + G^2 + B^2 \text{ which is the same.}$$

For the operation we must calculate the coordinate B or Z, then we must make the sum of X and Y, 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 language would be translated as:

```
normalCol.b = sqrt(1 - (pow(normalCol.r, 2) + pow(normalCol.g, 2)))
```

Why are we doing this? Remember that our normal map has up to four channels (RGBA or XYZW) and in DXT compression we are only using two of them (AG). The third channel has been discarded in the *normalCol* vector; this means that we will not use the B coordinate values that are included in its normal map. Now, we must calculate this coordinate if not our map will not work correctly, that is why we carried out the previous operation, where we calculated a new normalized vector based on the AG coordinates.

Now we must apply the compression to the *normal map*, so we will go back to the *fragment shader stage* and pass the texture as an argument to 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);

     …

 }
```
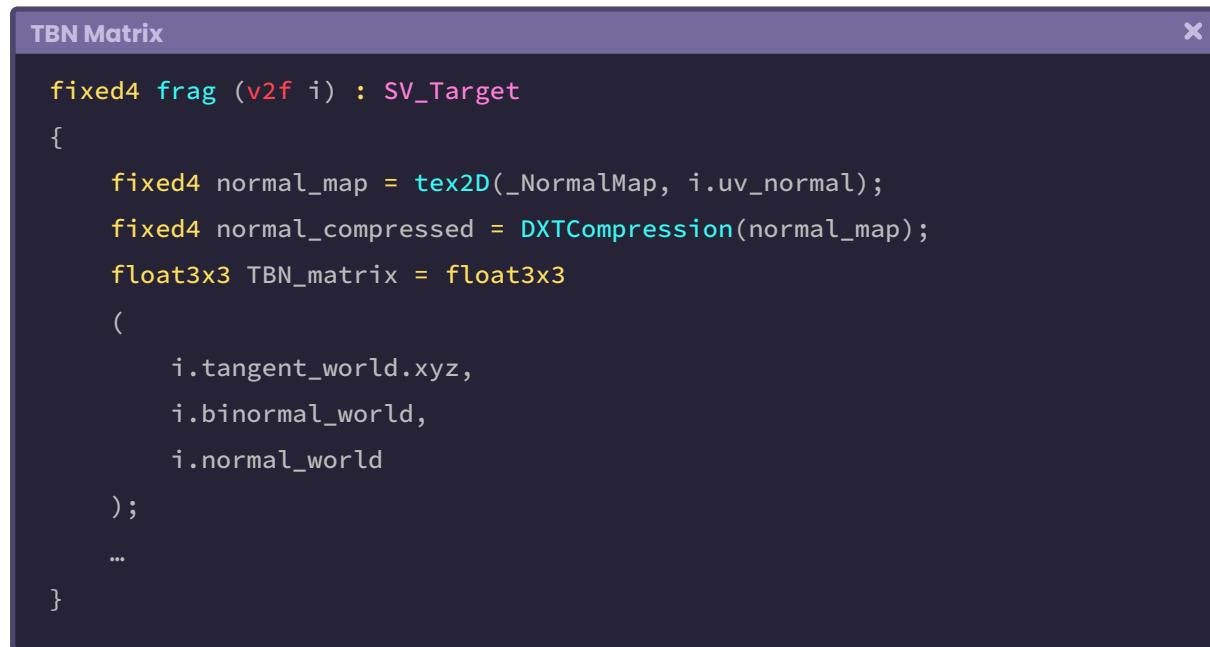
The exercise we have just performed is equivalent to the **UnpackNormal** function which is included in **UnityCg.cginc,** this means that we could replace the *DXTCompression* function with the latter and get the same result.

```
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 we already know, the TBN matrix is composed of the *tangents, binormals* and *normals* of our object. In section 6.0.1 we looked at how to transform these properties from *object-space* to *world-space.* What we will do next is create a new matrix to transform our *normal map* to *tangent-space.* To do this, we will 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, we created a matrix called **TBN_matrix,** which has three by three dimensions [3x3]. In it, we included the *tangent* and *binormal* outputs and the *normals;* all in *world-space.* Please note that in the case of *tangents,* we included their XYZ coordinates explicitly. This is because they were declared as a four-dimensional vector in the **struct v2f,** otherwise, if we do not specify the number of dimensions we want to use, the program will assume that it must use all of them (XYZW), and this could, consequently, generate an error with our shader unable to compile them.

To conclude, all we have to do is multiply our *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);

}
```
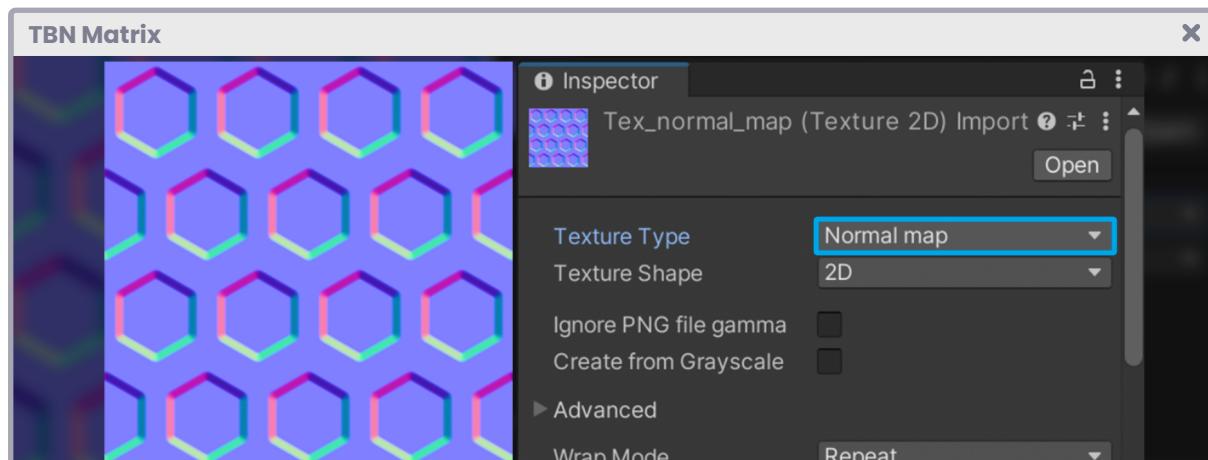
The example has created a three-dimensional vector called **normal_color**. This vector has the result or factor between the *normal map* and the TBN matrix. Finally, we have returned the color of the normals in RGB, and assigned the value "one" to the A (alpha) channel.

It is important to mention that when we import a *normal map* to Unity, by default, it is configured in "**Texture Type Default**" within our project.

Before assigning the *normal map* to our material, we must select the texture, go to the inspector, and set it as **Texture Type Normal Map**, otherwise, it may create a program error.



*(Fig. 6.0.3a)*

# Lighting.

## 7.0.1. | Lighting model.
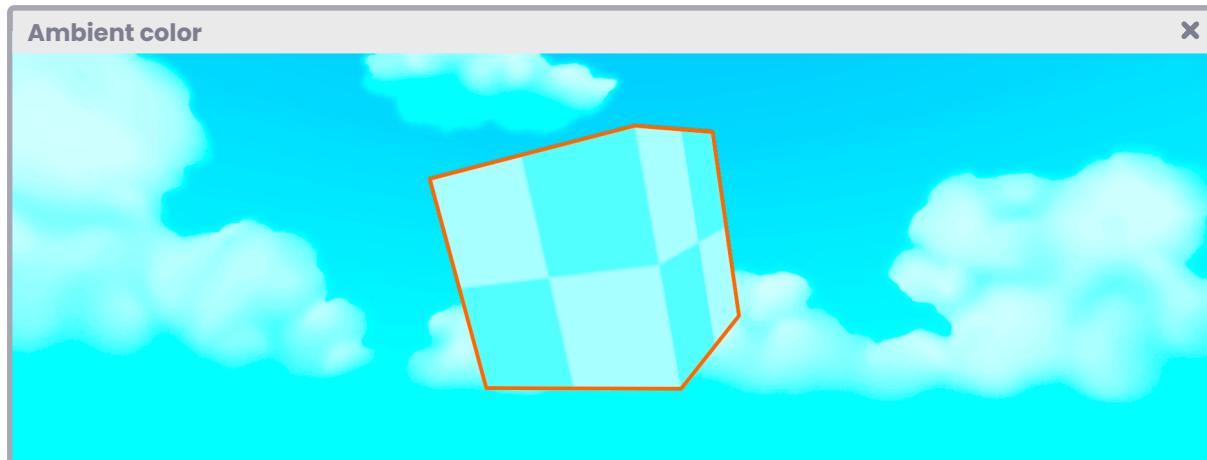
A lighting model refers to the result of the interaction between an object surface and the light source. By definition, it includes the light source properties (e.g. color, intensity, etc.), and the assigned material properties.

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 lighting and is performed within the **fragment shader stage.**

In the previous chapter, section 1.1.2, we conceptually defined the function of a *rendering path,* which corresponds to a series of operations related to the lighting and shading of objects. We also illustrated two types of rendering, *forward rendering* and *deferred shading.* In this section we will recreate a shader using a basic lighting model, for this we will talk about ambient color, diffuse reflection, and specular reflection.

## 7.0.2. | Ambient color.

A default value that we can find in real life is darkness. Lighting is the reason why we can perceive volume. This is an interesting point because all objects in their nature are dark and the reason we can differentiate between one surface and another, is due to how lighting interacts with the properties of such a surface. This is due to the initialization of "zero" of a light property because zero equals "black", that is, its default value.

*(Fig. 7.0.2a)*

Ambient color refers to the hue of lighting that is generated by the bouncing 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 illumination to simulate the natural phenomenon of light reflection.

In Unity, we can easily access the environment color, for this we must follow the menu *Windows / Rendering / Lighting*. This displays the **Lighting** window and in it, we can find the global lighting properties for our project. In the "Environment" tab there are two properties that we will use in our shader directly, these refer to:

- 1. Source and,
- 2. Ambient color.

To illustrate this, we will create a new **Unlit Shader** and name it **USB_ambient_color** and focus on an internal variable called UNITY_LIGHTMODEL_AMBIENT.

We will start by creating a property to increase or decrease the amount of ambient color in our shader. To do this, we can use a range between zero and one [0, 1], this is because "zero" refers to 0% illumination and "one" to 100%, therefore, if we want to dynamically modify our property, we will have to add a numerical range.

```
Ambient color                                               ✖

  Shader "USB/USB_ambient_color"
  {
      Properties
      {
          _MainTex ("Texture", 2D) = "white" {}
          _Ambient ("Ambient Color", Range(0, 1)) = 1
      }
      SubShader { … }
  }
```

As we can see, we declared a floating type property called **_Ambient,** which has a range between zero and one [0, 1]. Its default value equals one [1], which means that the ambient color will be initialized at 100% illumination.

Next, we must define its connection variable so that this property can communicate with the program.

```
Ambient color                                               ✖

  Pass
  {
      CGPROGRAM

      …
      sampler2D _MainTex;
      float4 _MainTex_ST;
      float _Ambient;

      …
      ENDCG
  }
```
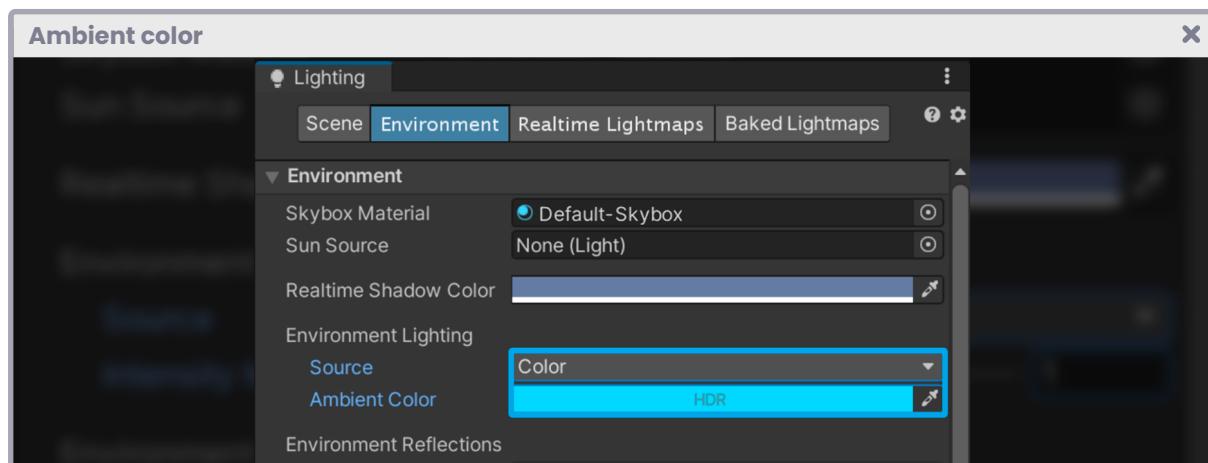
Now we simply go to the **fragment shader stage** and use the internal variable UNITY_LIGHTMODEL_AMBIENT as follows:

**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;

}
```

The factor between the ambient color and the property **_Ambient** has been saved in the vector **ambient_color**. Finally, the RGB ambient color has been added to the *col* texture. Up to this point, our ambient color is already working. Now we simply need to 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.
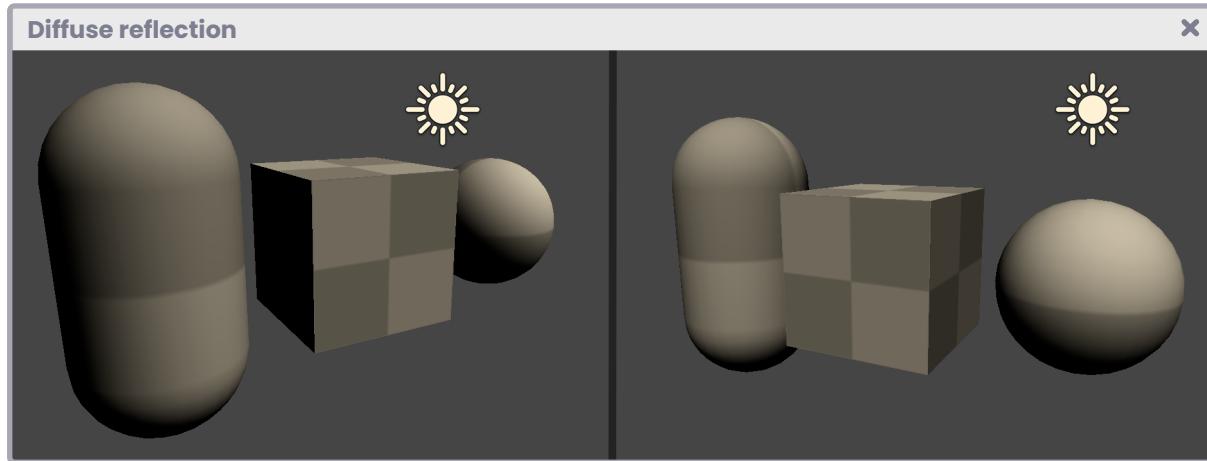
**Ambient color** ✕



*(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. For that reason, 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 Johann Heinrich Lambert's *Lambert's cosine law*, he makes an analogy between illumination and the surface of an object, considering the light source direction and the surface *normal*.



*(Fig. 7.0.3a)*
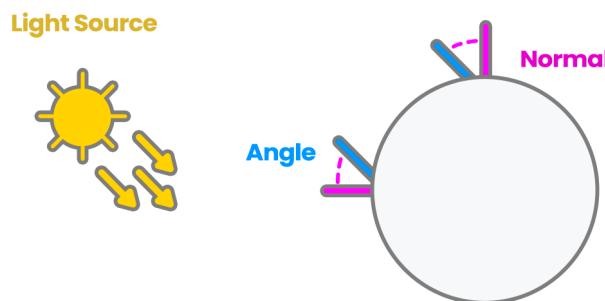
In Maya 3D we can find a material called *Lambert* which adds diffuse reflection by default. This same concept is applied in Blender, with the difference that the material is called *Diffuse,* however, the same operation is performed in both cases and the result is quite similar, with render pipeline architecture variations according to each software.

According to Johann Heinrich Lambert, we must carry out the following operation to obtain a perfect diffusion:

$$D = D_r \; D_l \; \max(0, n \cdot l)$$

How does this equation translate into code? To answer this question, we must first understand the analogy between the illumination and the *normal* direction of a surface. Let's assume that we have a Sphere and a directional light pointing towards it as follows:

**Diffuse reflection**

*(Fig. 7.0.3b)*

In the example above, the diffusion is calculated according to the angle between the normal $[n]$ of the surface 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 $[D_r]$ and its intensity $[D_l]$, and the maximum (max) between zero $[0]$ and the result of the product point between the surface normal and the lighting direction $[n \cdot l]$.*

To understand this definition better, we will create a new **Unlit Shader,** which we will call **USB_diffuse_shading.** We 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
```

*Continued on next page.*

```
    {
        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));
}
```
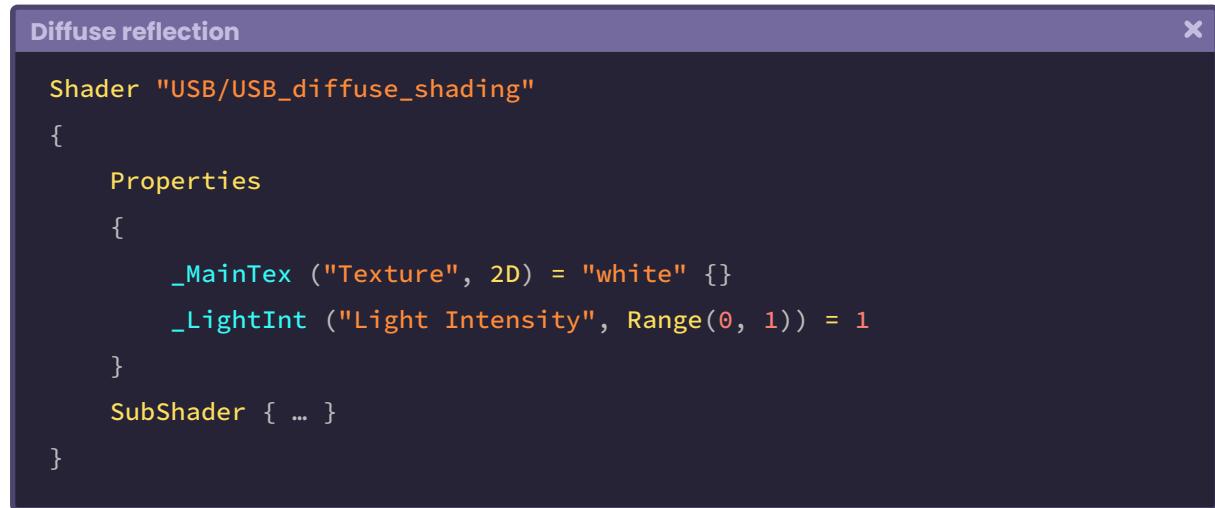
The **LambertShading** function returns a three-dimensional vector for its RGB colors. As an argument we have used the light reflection color (colorRefl RGB), the light intensity (lightInt [0, 1]), the surface normals (normal XYZ), and the lighting direction (lightDir XYZ).

It is worth pointing out that, both the normals and the lighting direction will be calculated in *world-space,* therefore, we will have to transform the normals in the **vertex shader stage.**

For lighting, it is not necessary to generate a transformation because we 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 [0.0f, 1.0f], we can start its implementation by going to the properties to declare a floating variable. In the next exercise, we will generate

**209**

a floating type property, which we will call **_LightInt.** This will be responsible for controlling the light intensity between the aforementioned range.

**Diffuse reflection**
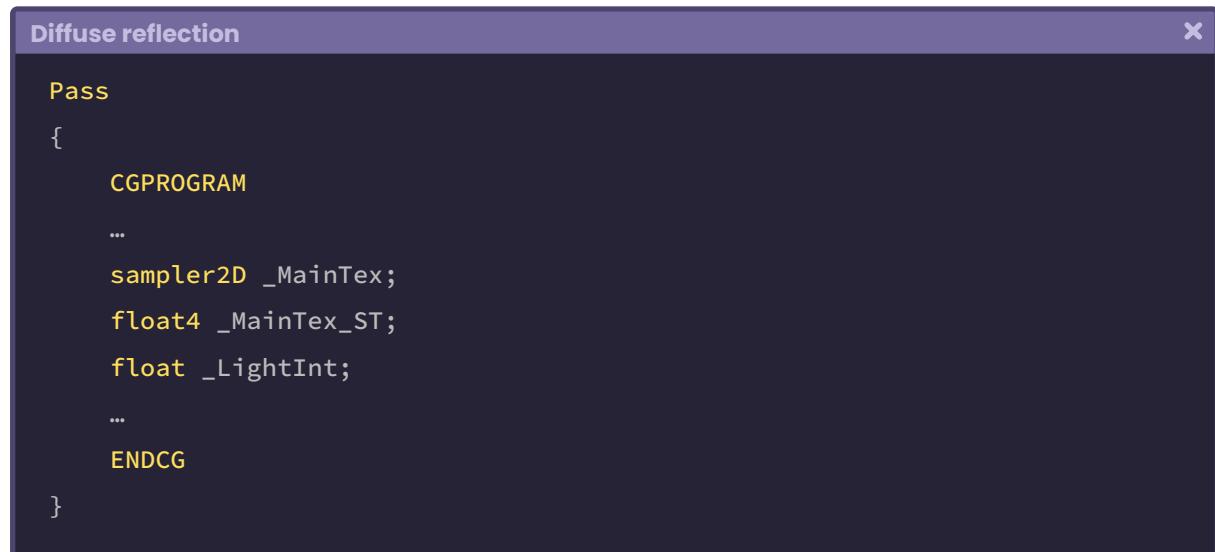
```
Shader "USB/USB_diffuse_shading"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _LightInt ("Light Intensity", Range(0, 1)) = 1
    }
    SubShader { … }
}
```

We use **_LightInt** to increase or decrease light intensity in our **LambertShading** function. As part of the process, we must then declare an internal variable to generate the connection between the property and our program.

**Diffuse reflection**

```
Pass
{
    CGPROGRAM

    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _LightInt;

    …
    ENDCG
}
```

Now our property is set up, we must implement the **LambertShading** function in the *fragment shader stage.* To do this, we will go to the stage and declare a new three-dimensional vector, we will 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 we already know, the **LambertShading** function has four arguments, which are:

- **colorRefl**.
- **lightInt**.
- **normal**.
- and **lightDir**.

In the example above, we have initialized these arguments to "zero" except for *"lightInt"* which, given its nature, must be replaced by the property **_LightInt** in the second box.

We will continue with the reflection color, for this, we can use the internal variable **_LightColor[n],** which refers to the lighting color in our scene. To use it, we must first declare it as a uniform variable within the CGPROGRAM as follows:

**Diffuse reflection**  ✖

```
Pass
{
    CGPROGRAM

    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    float _LightInt;
```
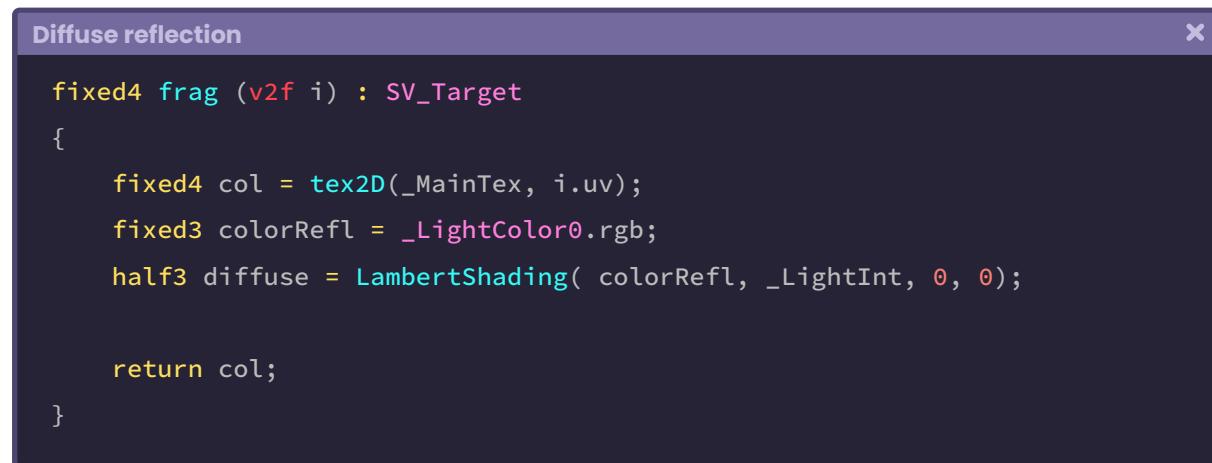
*Continued on next page.*

```
    float4 _LightColor0;

    …

    ENDCG
}
```
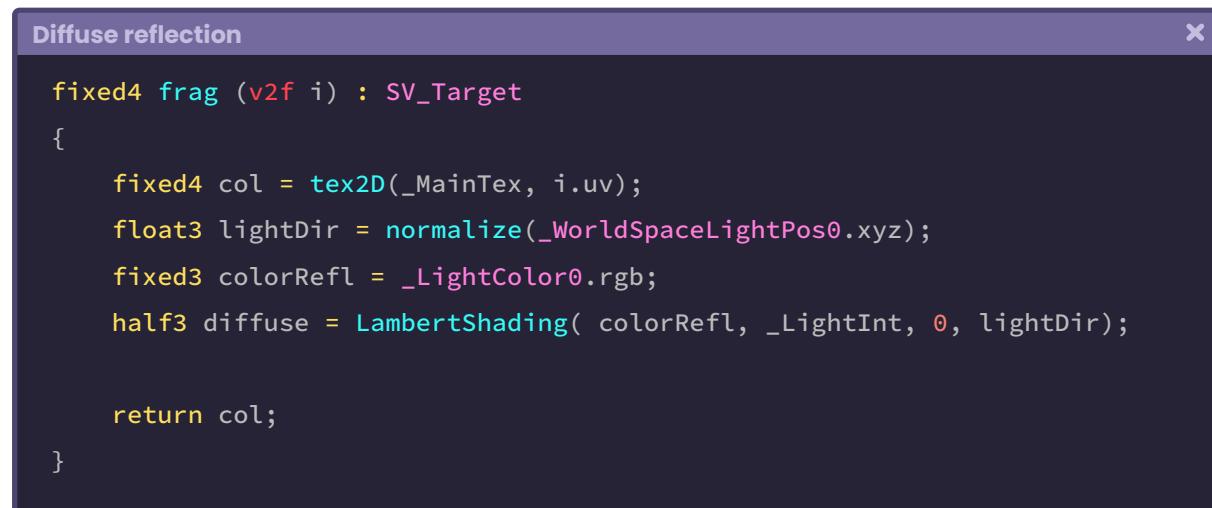
Now we can use it as the first argument in the **LambertShading** function and declare a new three-dimensional vector using only the light's RGB colors.

**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;
}
```

For the lighting direction, we can use the internal variable **_WorldSpaceLightPos[n],** which, as we already mentioned, refers to the direction of directional light in *world-space.*

Unlike **_LightColor[n],** it is not necessary to declare this variable as a uniform vector because it has already been initialized in the #include "UnityCG.cginc", so we can use it directly as an argument in our function.

**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 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" [1]. Finally, the lighting direction has been positioned as the fourth argument in the function.

Only the third argument that corresponds to the *world-space* object normals is missing, for this we have to go to both the **vertex input** and the **vertex output** and include this semantic.

**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 we are configuring the *normals* in both the *vertex input* and *output* is precisely because we will use them in the **fragment shader stage,** where our 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**. This means that the *normals* have been configured in *world-space* and have a magnitude of "one" [1].

Now we 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.
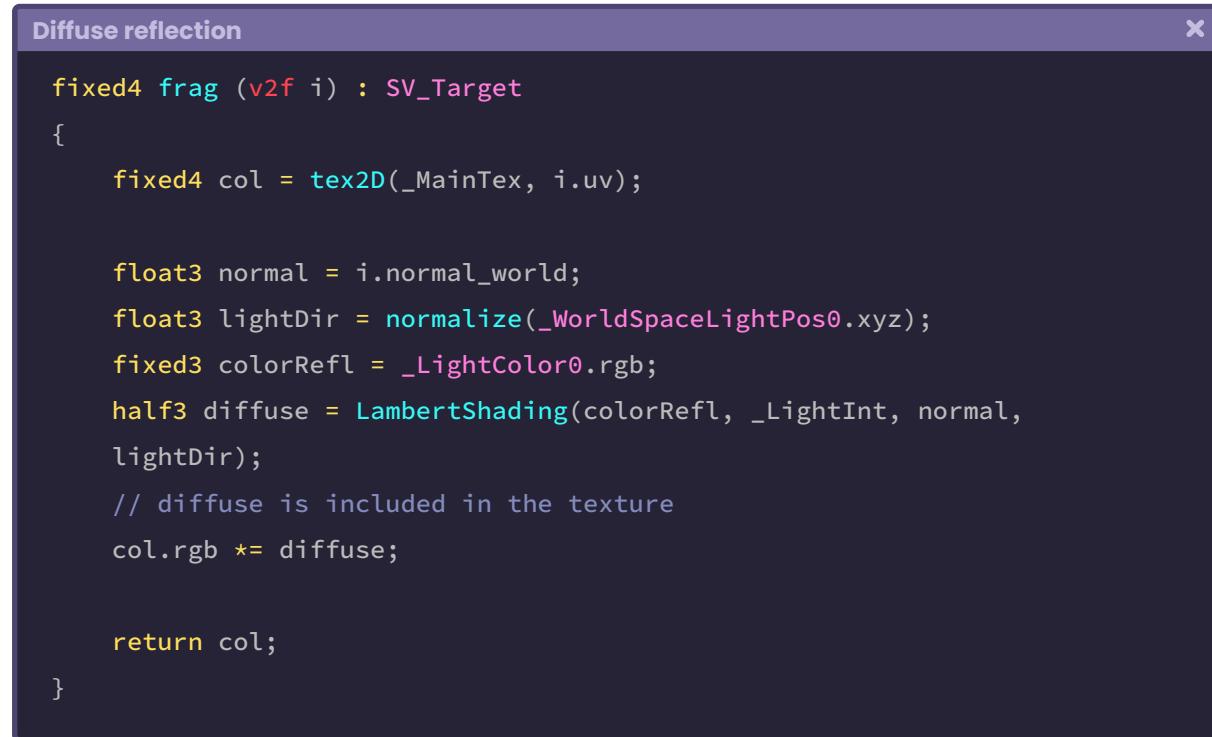
```
Diffuse reflection                                              ✕

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;

}
```

In the above example, we declare a three-dimensional vector called **normal,** to which we pass the object *normals*. Then, we use this vector as the third argument in the **LambertShading** function.

Next, multiply the diffusion by the texture's RGB color and, since our shader is interacting with a light source, we will need to include the **LightMode ForwardBase,** thus the *render path* will be configured.

**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.*
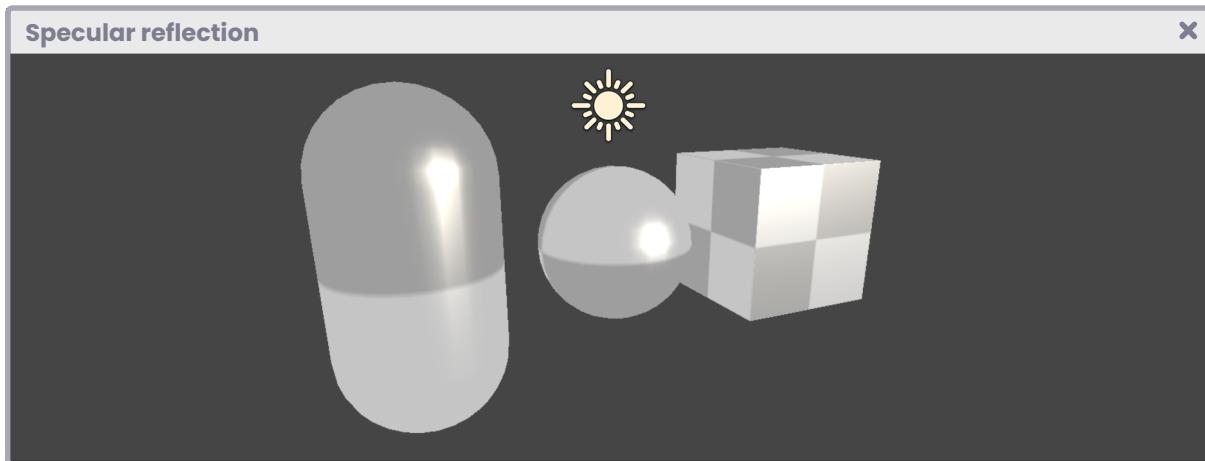
**Diffuse reflection** ✕

```
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 its *normal*. In fact, in Maya 3D there is a material with this name and its precise 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:
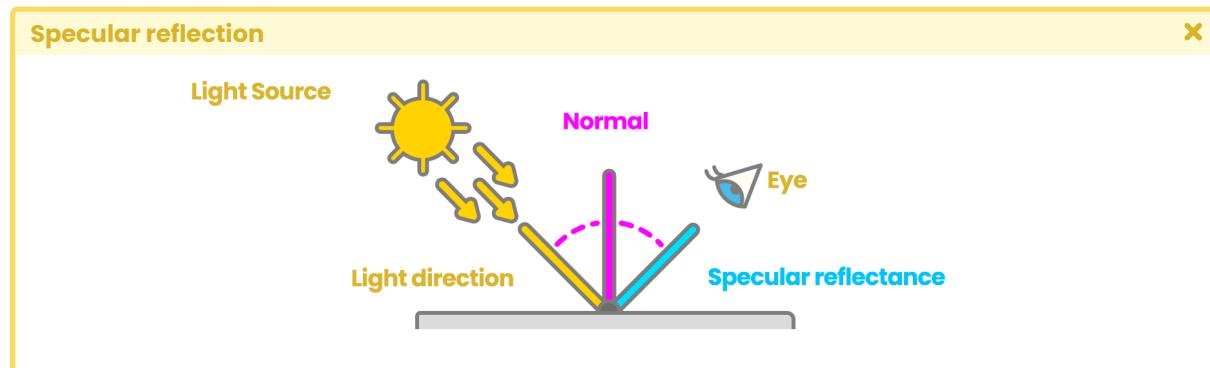
$$S = S_a \ S_p \ \max(0, h \cdot n)^2$$

Note that this equation is very similar to the function that allows us to calculate diffuse reflection.

$$D = D_r \ D_l \ \max(0, n \cdot l) \qquad \text{Diffuse reflection.}$$
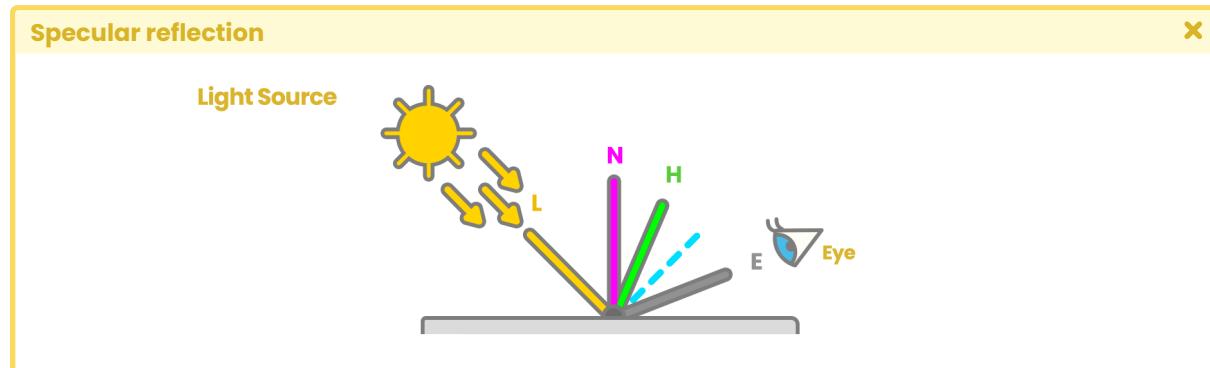
The big difference lies in the calculation of the vector $[h]$ which corresponds to a half vector called "*halfway*". This allows us to appreciate the brightness of the reflection when it is close to $[n]$; where the latter corresponds to the surface *normals.*

To understand the concept, we will begin our study by demonstrating specular reflection, assuming we have a flat surface and directional light pointing towards it as follows:



*(Fig. 7.0.4b)*

From the image above, we can deduce that specular reflection has the same angle as the light direction. This represents a problem given that if our eye/camera is not in the same direction of reflection, then we will not be able to see it. To solve this, we can calculate an intermediate vector between the normals and the light direction, following the view direction.



*(Fig. 7.0.4c)*

The vector [e] corresponds to the "view direction", while the vector [h] is the *halfway* value that we have calculated between the light direction and the surface normal. To determine the value of the vector [h] we can perform the following function.

$$h = \frac{l + e}{\| l + e \|}$$

It is worth mentioning that for our program we 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:

$h$ = normalize( $l + e$ ).

In the reflection calculation, there will be at least three variables that we will have to add in our code, these correspond to

- lighting direction,
- surface normals and
- the *halfway* value that includes the view direction.

Additionally, if we want to add specular maps, we will have to calculate the reflection color.

We will start a new program to review these concepts, for this, we will create an **Unlit Shader** which we will call **USB_specular_reflection.** It is worth mentioning that many of the operations that we will perform in this section are the same as those carried out in **USB_diffuse_shading,** so we can start from scratch or continue from the shader that we developed in the previous section.

Within our program, we will create a function called "SpecularShading." Within it, we will include the properties mentioned above as follows:

**Specular reflection** ✖

```
Shader "USB/USB_specular_reflection"
{
    Properties { … }
    SubShader
    {
        Pass
        {
            CGPROGRAM

            …
```
*Continued on 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);
}
```

On this occasion, we have declared a function called **SpecularShading** that returns a three-dimensional vector for its RGB colors. Among its arguments we can find the reflection color (colorRefl RGB), specular intensity (specularInt [0, 1]), surface normals (normal XYZ), light direction (lightDir XYZ), view direction (viewDir XYZ) and the specular exponent of (specularPow [1, 128]).

In the same way that we did in the diffuse reflection calculation, the normals and the view and lighting directions will be calculated in *world-space,* therefore, some transformations will have to be made in the **vertex shader stage.**

219
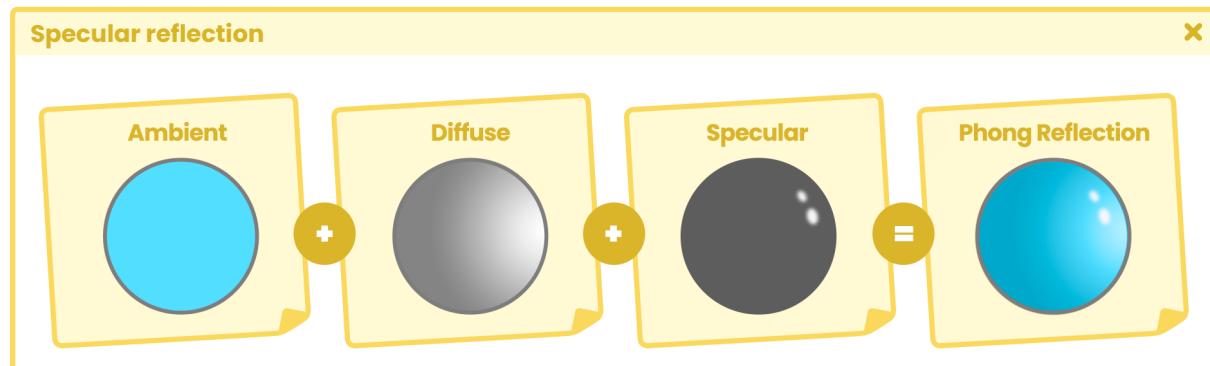
We will start by configuring three properties for our shader: a texture property for the specular map, a reflection intensity range between zero and one, and a new specular exponent range between 1 and 128.

```
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
      }
  }
```
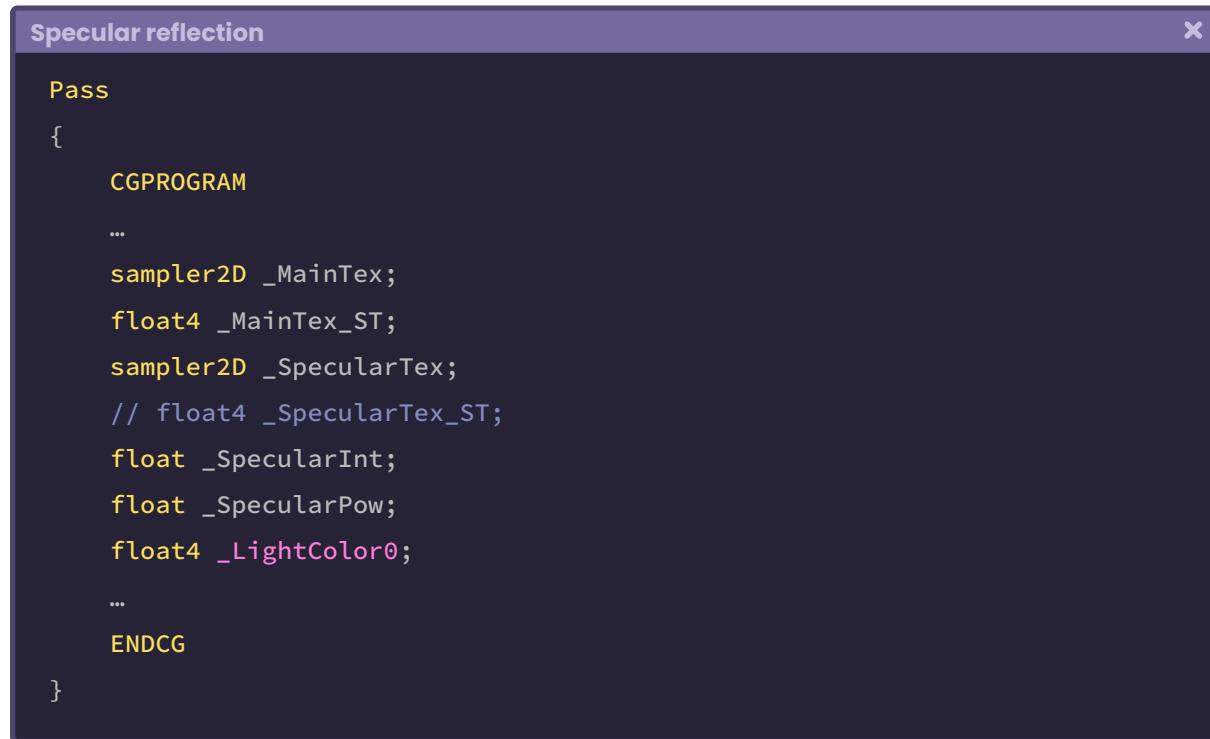
Unlike the _MainTex property, _SpecularTex has a black color as a default. This can be corroborated in the definition "*black*" found at the end of the statement. Its operation is quite simple: If we do not assign a texture from the Unity Inspector, then the object will look black.

Note that specular is going to be added to the main texture, therefore, for this case black will not be visible graphically because, as we already know, black equals "zero", and zero plus one equals one.



(Fig. 7.0.4d)

Next, we must declare the connection variables for the three properties that we have added.

```
Pass
{
    CGPROGRAM

    …
    sampler2D _MainTex;
    float4 _MainTex_ST;
    sampler2D _SpecularTex;
    // float4 _SpecularTex_ST;
    float _SpecularInt;
    float _SpecularPow;
    float4 _LightColor0;

    …
    ENDCG
}
```

Why have we discarded the variable **_SpecularTex_ST** in the above example? As we 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 specular maps do not need them due to their consistent nature.

Another connection variable that we 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 color of the light source that we have in our scene.

The properties are now functional, so we can start implementing the **SpecularShading** function in the *fragment shader stage*.

We will 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, we multiply the texture *_SpecularTex* by the lighting color *(colorRefl),* and the factor is stored within a three-dimensional vector called **specCol,** which is assigned as the first argument.

The second argument corresponds to specular intensity, for this, we can simply assign the property **_SpecularInt,** which is a range between zero and one [0, 1].

```
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, we have to add the normals in both the **vertex input** and **output** and then transform their space into the **vertex shader stage,** in the same way we did in the diffuse reflection calculation.

We will start by configuring the normals in the *vertex input.*

**Specular reflection**　　　　　　　　　　　　　　　　　　　　　　　✕

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Then we must assign the normals in the *vertex output,* however, we must remember that the NORMAL semantic does not exist for this process, therefore, we 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,* we create one last property in the *vertex output*. This property will be used later in the calculation of the view direction, as a reference point.

```
Specular reflection                                              ✕

struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float3 normal_world : TEXCOORD1;
    float3 vertex_world : TEXCOORD2;
};
```

As we can see, a new property called **vertex_world** has been included, which refers to the position of the object vertices in *world-space.*

We then go to the **vertex shader stage** to transform the coordinates' space, however, unlike the previous processes, we 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 *normal* input. Next we 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.*

We will continue with the SpecularShading function.

```
Specular reflection                                                    ✖

  fixed4 frag (v2f i) : SV_Target

  {

      fixed4 col = tex2D(_MainTex, i.uv);

      // we 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 we can see, a new three-dimensional vector called **normal** has been created. This vector has its normal 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 we 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 we only need to calculate the *view direction* since the last argument in the **SpecularShading** function corresponds to the exponential value (specularPow) which increases or decreases reflection.

To calculate the view direction, we must subtract the object vertices in *world-space* from the camera also in *world-space*. Unity has an internal variable called **_WorldSpaceCameraPos,** which gives us precise access to the scene's camera position.

**Specular reflection**                                                                    ✕

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // let's calculate the light direction
    float3 viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
    float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
```

*Continued on next page.*
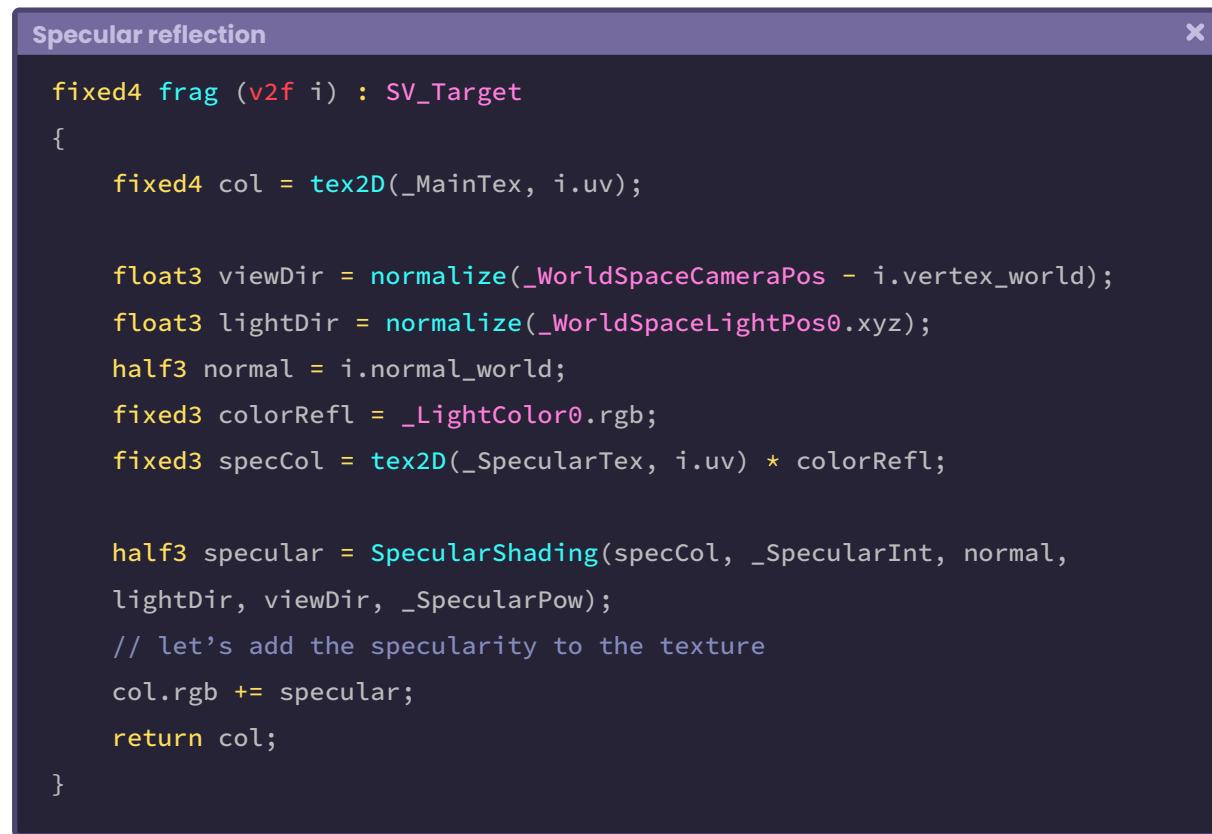
```
        float3 normal = i.normal_world;

        fixed3 colorRefl = _LightColor0.rgb;

        fixed3 specCol = tex2D(_SpecularTex, i.uv) * colorRefl;

        // we 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 we 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 we cannot add a four-dimensional vector to a three-dimensional vector, so we must make sure that specular reflection is only added to the main texture RGB channels.
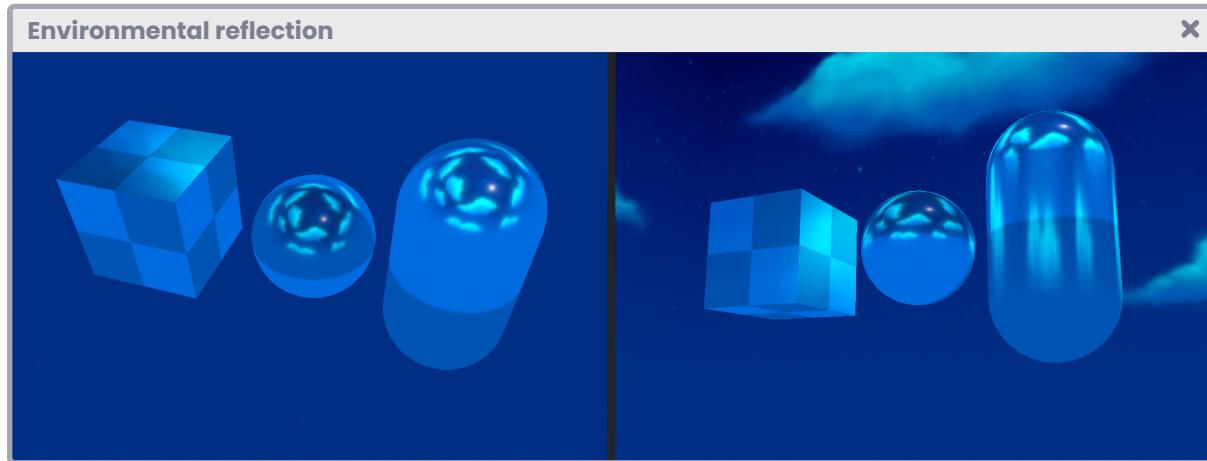
By the fact that specular reflection is a lighting pass, we must once again go to the **Tags** and configure the *render path* in the same way as we did for diffuse reflection.

```
Specular reflection                                              ✖

  Shader "USB/USB_specular_reflection"
  {
      Properties { … }
      SubShader
      {
          Tags
          {
              "RenderType"="Opaque"
              "LightMode"="ForwardBase"
          }
      }
  }
```

## 7.0.5. | Environmental reflection.
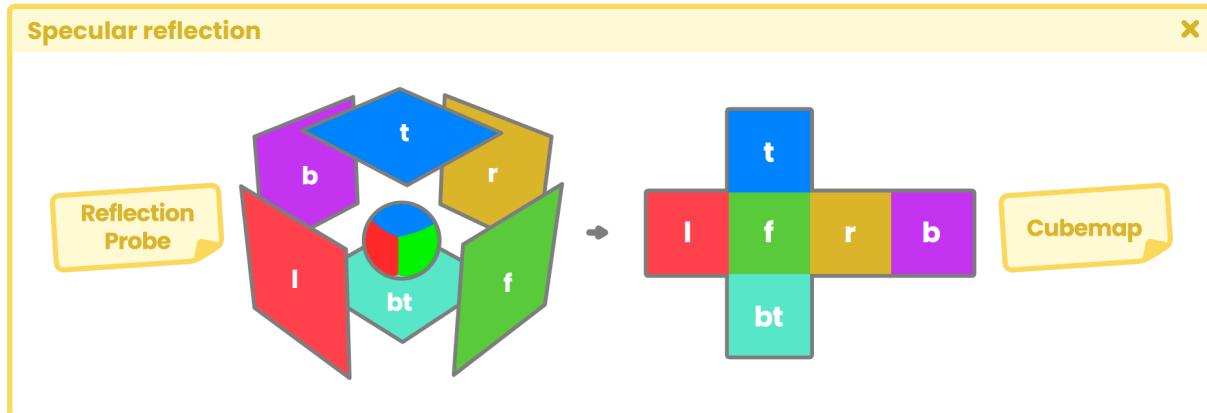
Environmental reflection occurs similarly to specular reflection. The difference is that the latter is affected only by the main light source, while ambient reflection is affected by every light ray that impacts the surface.



Environmental reflection                                         ✖

*(Fig. 7.0.5a)*

Given its nature, calculating this type of reflection in real time uses a lot of GPU power, instead, we can use a **Cubemap** type texture. In chapter one, section 3.0.6, we mentioned the **Cube** property, which refers precisely to this type of texture.
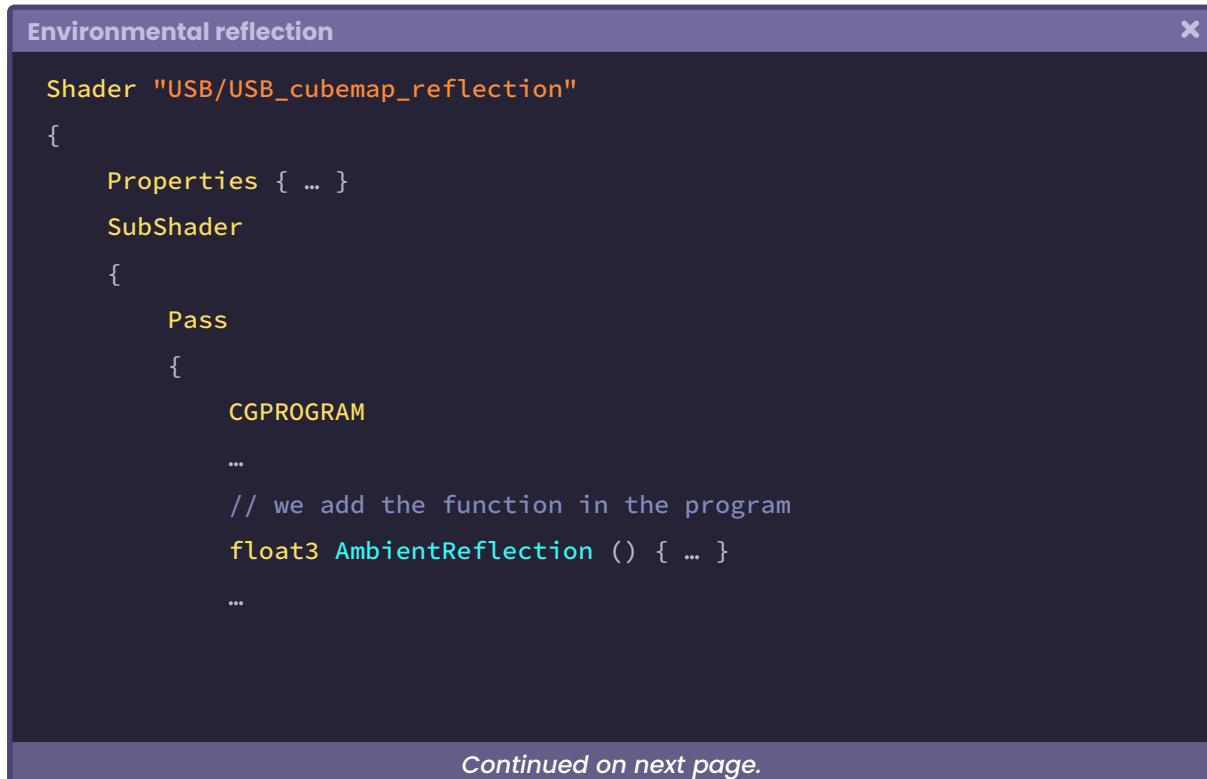
In Unity, we can generate *Cubemaps* using the Reflection Probe component. This object works similar to a camera, which captures a spherical view of its surroundings in all directions and then generates a Cube-type texture that we can use as a reflection map.



*(Fig. 7.0.5b)*

To see the implementation of this type of reflection in detail, we will create a new **Unlit Shader** called **USB_cubemap_reflection.**

Let's start by creating a function called **AmbientReflection** and include it in our program as follows:

```
Shader "USB/USB_cubemap_reflection"
{
    Properties { … }
    SubShader
    {
        Pass
        {
            CGPROGRAM

            …

            // we add the function in the program
            float3 AmbientReflection () { … }

            …
```

*Continued on next page.*

```
        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 = texCUBElod(colorRefl, float4(reflection_world,
    reflectionDet));

    return reflectionInt * cubemap.rgb * (cubemap.a * reflectionExp);
}
```

The first argument declared in the previous function corresponds to a *sampler* for a *Cube* type texture (colorRefl RGBA); this involves the creation of a property of this type. Then we can find a variable that will modify the reflection intensity (reflectionInt) where the maximum value is "one" and the minimum "zero". The third argument corresponds to a medium-precision variable called "reflectionDet" ([1, 9]). This variable increases or decreases the **texel** density of the *samplerCUBE*.

Note that *reflectionDet* has been included in the **texCUBElod** method as follows:

**Environmental reflection** ✕

```
texCUBElod(colorRefl, float4(reflection_world, reflectionDet));


// float4 texCUBElod(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 we 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 values of the vector 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).

As in specular reflection, we must calculate both the *normals* and the view direction in *world-space,* for this reason, we have included these vectors as arguments in the function. Finally, as a final argument, we can find a variable called "reflectionExp". This refers to the reflection map color exposure.

A function that we will frequently use in the reflection calculation is "**reflect**". 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);
```

Argument [$i$] refers to the *incidence* value, that is, the view direction, while [$n$] refers to 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 graphically turn the reflection map as if we were seeing the reflection through a concave lens. To solve this, we have to make the incidence value negative.

Now that we understand a great deal of the operation, we will start implementing it in the *fragment shader stage.* To do this we will 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;
}
```
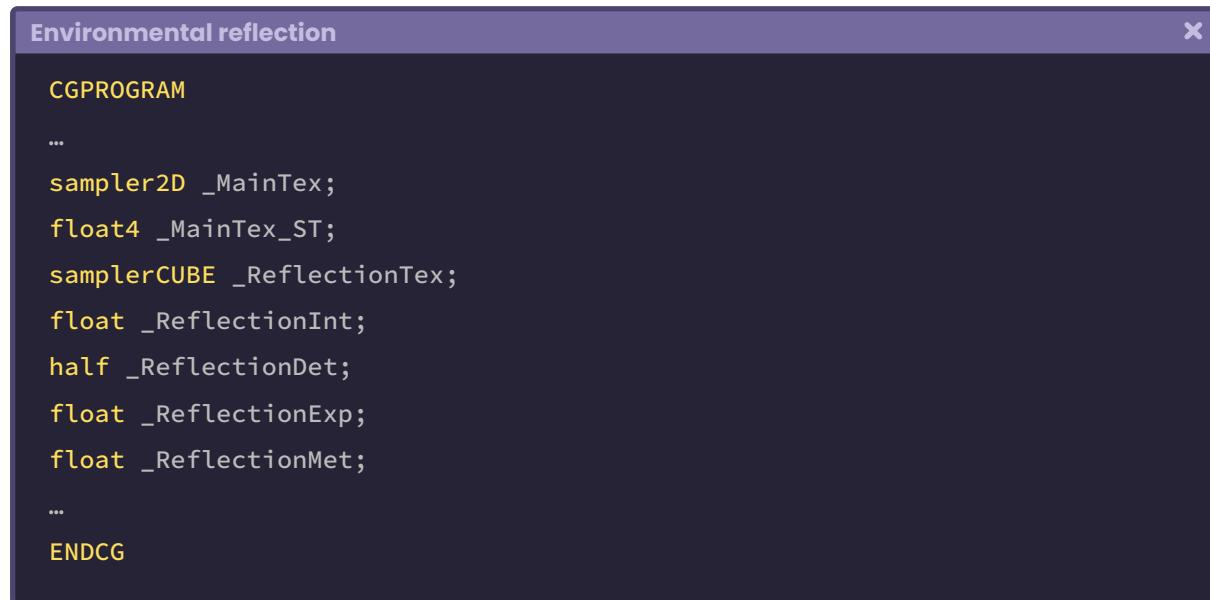
As we already know, the first argument in the function corresponds to the Cube type reflection color, so we will go to our shader properties and declare the texture along with the intensity, detail, and exposure variables.

**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, we can find **_ReflectionMet** which has not been included in the **AmbientReflection** function. As the name says, we use this property to control the reflection shininess and thus emulate a metal surface.

We will continue to generate 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, we can pass the *_ReflectionTex* property directly as the first argument in the function statement.

**Environmental reflection** ✕

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // let's add the cubemap
    half3 reflection = AmbientReflection(_ReflectionTex, 0, 0, 0, 0, 0);

    return col;
}
```

We can also add the second and third arguments to the function, since these correspond to *reflectionInt* and *reflectionDet*.

**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, we carry out exactly the same operation that we performed in the specular reflection, that is, we have to include the normals in the *vertex input* and then declare both the normals and 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 we can 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,** we create a new vector in which we assign the *normals* and then carry out the view direction calculation, however, this time we will use the **UnityWorldSpaceViewDir** function included in UnityCg.cginc, which is equivalent to the camera position and object vertices calculation.
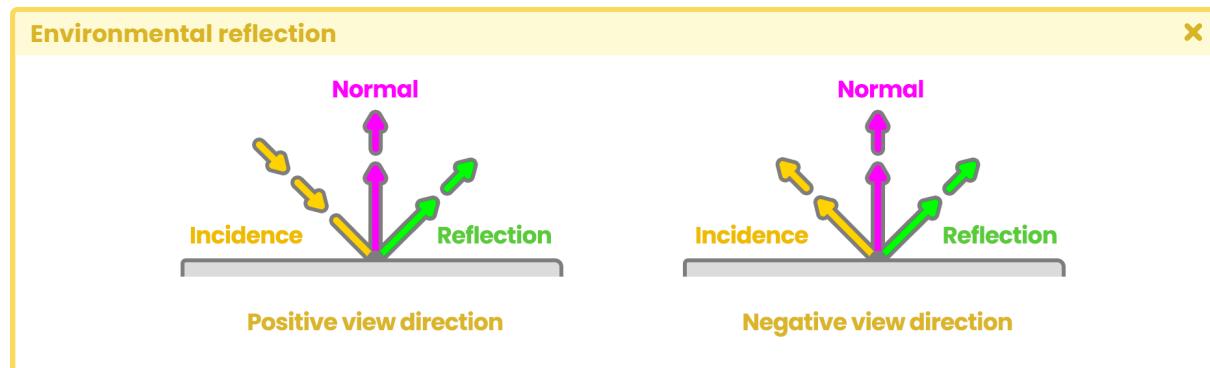
**Environmental reflection**    ✕

```
// 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;

}
```

Looking closely, we notice that the fifth argument; corresponding to the view direction (-*viewDir),* has been included in a negative form, 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 (_MainTex).

```
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 exposure
    half3 reflection = AmbientReflection(_ReflectionTex, _ReflectionInt,
    _ReflectionDet, normal, -viewDir, _ReflectionExp);

    col.rgb *= reflection + _ReflectionMet;
    return col;

}
```
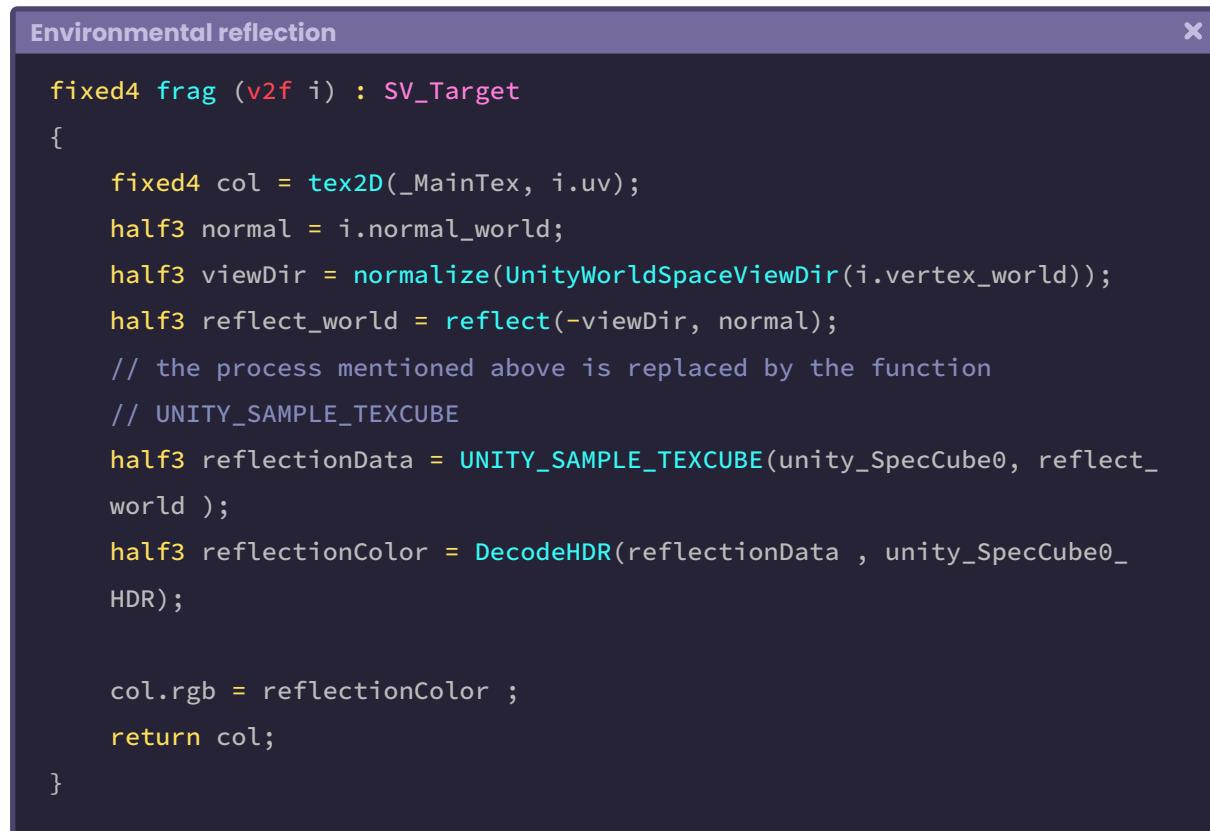
In the example above, we multiply the texture color in RGB by the reflection and then add the **property _ReflectionMet** which corresponds to a range between zero and one [0, 1]. To

understand this operation, we must pay attention to the property **_ReflectionInt** which is a range as well.

Since the *reflection* vector is being multiplied to *col.rgb,* the resulting color will be that of a metallic surface. Now, we 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 function UNITY_SAMPLE_TEXCUBE. This automatically assigns the environmental reflection that is configured in our scene, this means that, if we have configured a *skybox* from the lighting window then the reflection will be saved as a texture within our shader, and we 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 macro UNITY_SAMPLE_TEXCUBE samples this data using the reflection coordinates (reflect_world) and then decodes the colors in HDR through the *DecodeHDR* function, which is 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.
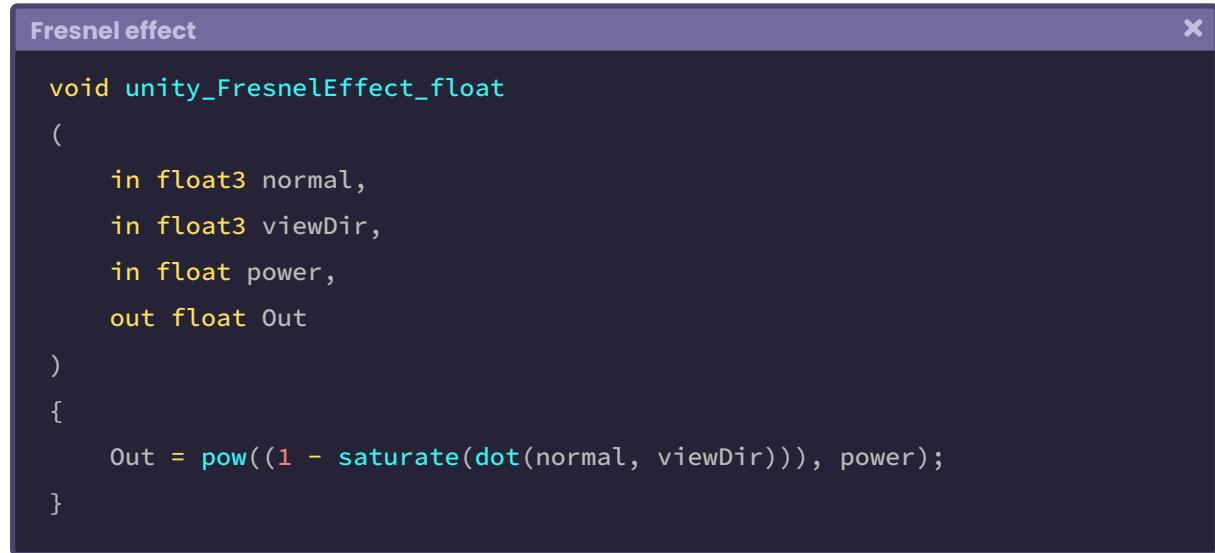


*(Fig. 7.0.6b)*

When the angle between the incidence value and the normals equals "zero" degrees there is no reflection, because both vectors are parallel, on the other hand, when the angle equals "ninety" degrees, the reflection is full, and the vectors are perpendicular. This is quite

interesting because, when the reflection is null, our program must return black. On the contrary, when it is full, it returns white, why? Because these correspond to the maximum and minimum illumination values of a pixel.

To understand this concept, we must analyze the following function coming 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);
 }
```

In the previous function several things are happening that we will detail throughout this section, for now, we will only focus on the output's internal operation. This operation can be divided into three processes:

> saturare(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 "zero and one" [0.0f, 1.0f].

As we already know, the "dot" function will return "one, zero or minus one" depending on the angle between its arguments [-1.0f, 1.0f]. Since the reflection operation requires only a value between "zero and one," the intrinsic function **saturate** has been added, limiting 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));
}
```
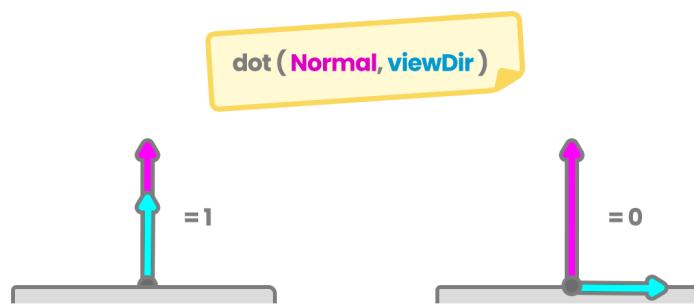
**Saturate** fulfills the same function as **clamp,** with the difference that with the latter we can modify the minimum and maximum value to generate the limit.

Let's continue with the operation "1 - x".
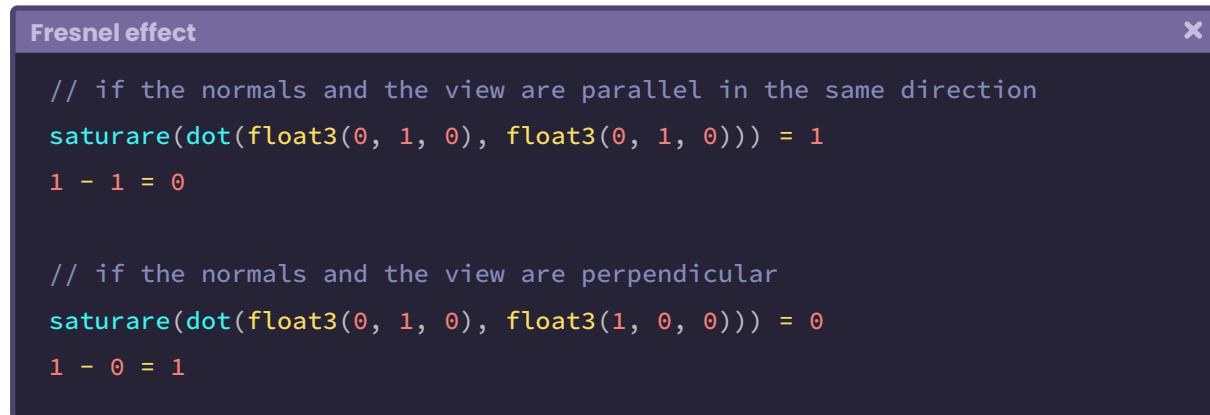
$$( 1 - x )$$

To understand its nature, we must return to the previous exercise. Dot product will return "*one*" [1] when the view direction vector and the normals are parallel and point in the same direction. This is a problem for us, because we need the operation to return "*zero*" [0] which is equivalent to black.

**Fresnel effect** ✖



*(Fig. 7.0.6c)*

The operation "1 - x" has the function of flipping the result as follows.

**Fresnel effect**                                                          ✕
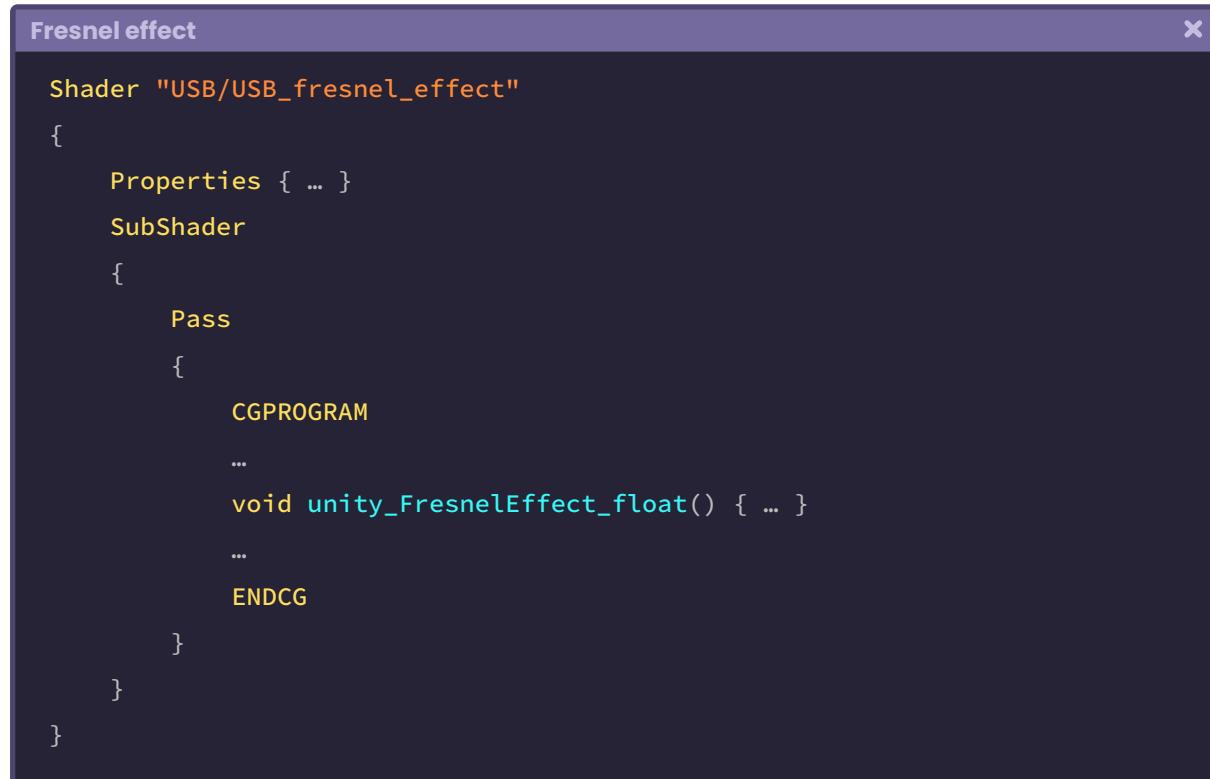
```
// if the normals and the view are parallel in the same direction
saturare(dot(float3(0, 1, 0), float3(0, 1, 0))) = 1
1 - 1 = 0


// if the normals and the view are perpendicular
saturare(dot(float3(0, 1, 0), float3(1, 0, 0))) = 0
1 - 0 = 1
```

Finally, in the function, we can find the operation **"pow( *x*,power )"** which allows us to increase or decrease the range of reflection.

To understand in detail the Fresnel operation of Shader Graph, we will start a new **Unlit Shader** that we will call **USB_fresnel_effect.** The first thing we must do is to include this function within our program.
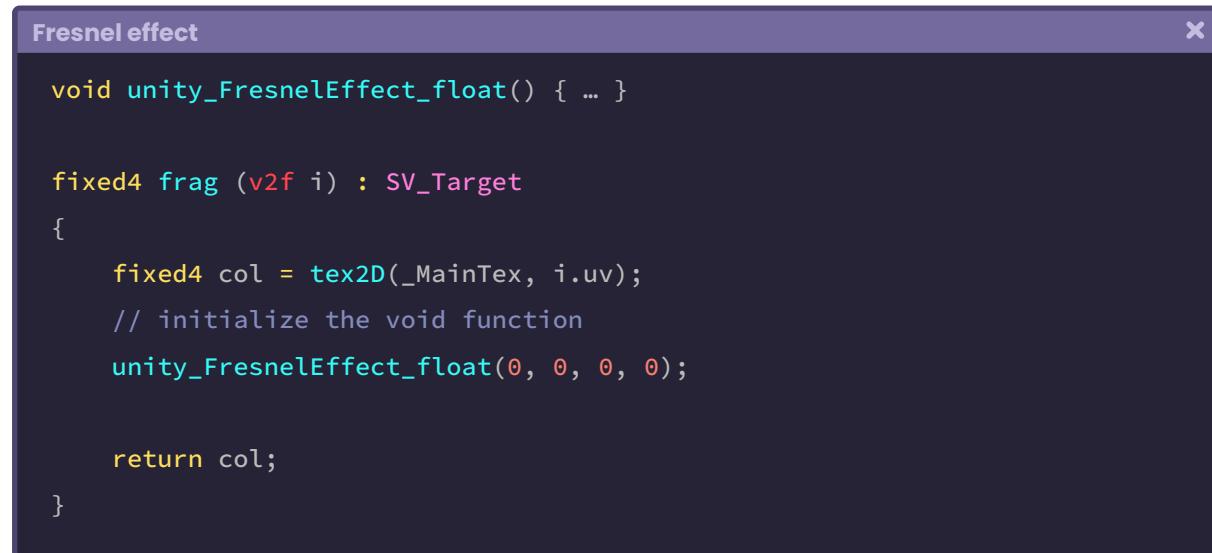
**Fresnel effect**                                                          ✕

```
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. In section 4.0.4 of Chapter I, we reviewed the difference between implementing an empty function and one that returns a value. In this case, we have to declare some variables and pass them as arguments, as appropriate.

We will 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 we have to go to **vertex input** and use NORMAL semantics, and then transform its space coordinates in the **vertex shader stage.**

Due to the fact that we use the normals in the **fragment shader stage,** we have to declare a vector in the **vertex output,** this way we can store the result of the transformation.

**Fresnel effect** ✕

```
// vertex input
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    float3 normal : NORMAL;
};
```

*Continued on next page.*

**242**
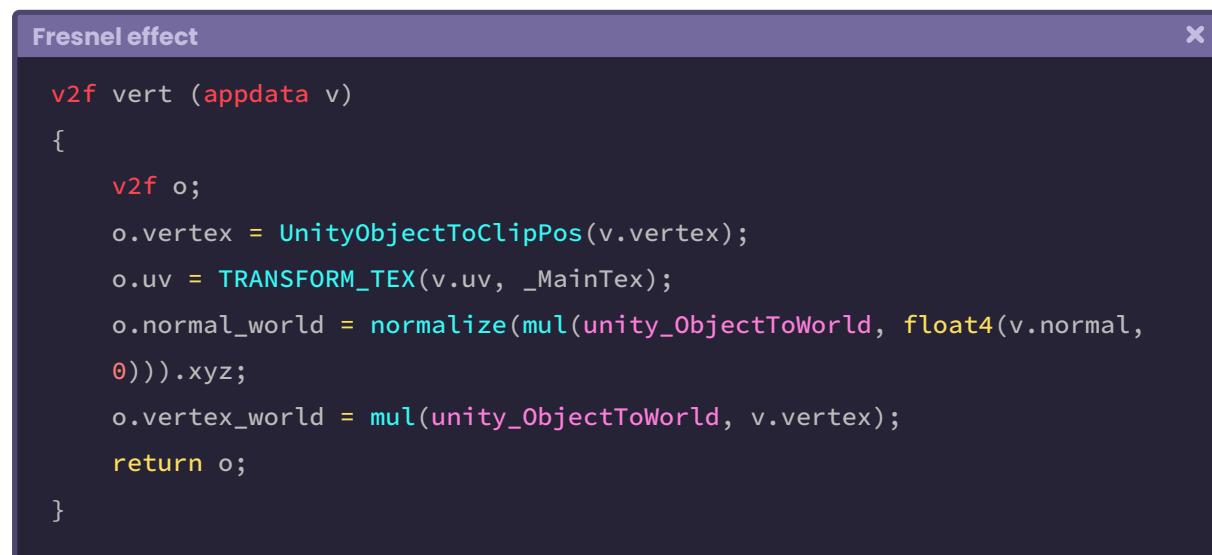
```
// vertex output
struct v2f
{
    float4 vertex : SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 normal_world : TEXCOORD1;
    float3 vertex_world : TEXCOORD2;
};
```

The vector **vertex_world** has been added to the *vertex output* because we need this variable to calculate the view direction.

If we pay attention, we will notice that the process is exactly the same as we 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;
}
```

To continue with the implementation of the Fresnel function, we will return to the fragment shader stage and declare two vectors: one for the normals calculation 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** has been declared to store the *normals* output value in *world-space.* Then a new vector called **viewDir** has been declared which contains the view direction calculation. Both vectors have been 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 (fresnel power) we have to declare a property with a numerical range. This 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 { … }
}
```

We will use the property **_FresnelPow** as a third argument in the function, as an exponential value; while we will use **_FresnelInt** to increase or decrease the amount of Fresnel effect in

the object. As we already know, we must declare connection variables for both properties within our program.

**Fresnel effect** ✖

```
CGPROGRAM
…
sampler2D _MainTex;
float4 _MainTex_ST;
float _FresnelPow;
float _FresnelInt;
…
ENDCG
```

Once this process is done, the property will be connected to our program, this means that we can dynamically modify the reflection range from the Unity Inspector. Now we 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 corresponds to the function output value, where we will save the color output. To do this, we simply create and add a floating variable to 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);
     // 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" (black). It was then included in the function as the fourth argument, as output. This means that within this variable is the result of the final operation that occurs in the *unity_FresnelEffect_float* function.

At the end of the operation, we can see that the *fresnel* variable result, due to its intensity **(_FresnelInt),** was added to the base texture color called "*col*". This adds reflection to the object in our scene and also allows us to modify its intensity value.

## 7.0.7. | Structure of a Standard Surface shader.

Before continuing defining some functions, we will take a look into the structure of a Standard Surface shader. This shader, different from the **Unlit Shader** type, is characterized by having a simplified structure, which is configured to interact with lighting only in *Built-in RP.*

The reflection functions that we saw in previous sections are included internally in this program, this means that this shader by default has global lighting, diffusion, reflection and fresnel.

**246**

If we create a **Standard Surface,** we 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 its operation, we must pay attention to the **surf** function that would be equivalent to the color output of the object surface. Within this function, we can find two arguments,

1. **Input IN**.
2. **And input SurfaceOutputStandard o**.

These refer to our shader inputs and outputs, and their semantics have been predefined internally in the code.

---

**Structure of a Standard Surface shader** ✕

```
Shader "Custom/SurfaceShader"
{
    Properties { … }
    SubShader
    {
        CGPROGRAM

        …

        #pragma surface surf Standard fullforwardshadows

        …

        void surf (Input IN, input SurfaceOutputStandard o)
        {
            …
        }
        ENDCG
    }
}
```

---

The reason why we 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,* with the difference in this case that we can declare other properties, e.g., the lighting model (Standard) and other optional parameters (fullforwardshadows).

By default, a *surface shader* comes configured with a **Standard** type lighting model, which contains some predefined properties that we can use as a color output.

Note that the defined properties in our code *surf* function are of **SurfaceOutputStandard** type, this means that the *color output* will be determined by the *Standard* lighting model.

**Structure of a Standard Surface shader** ✖

```
#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, in a *standard surface* by default we can find two *struct* type functions, these are:

- Input.
- *SurfaceOutputStandard*.

The *struct Input* is different from the *struct appdata* (vertex input) which we reviewed in the previous chapter, why?

In *appdata* we can define our object's semantics as an input, on the other hand, in *Input* we can determine our shader's predefined functions for the lighting calculation, what does this mean?

In *appdata* we can use the semantics POSITION[n] to use the position of the mesh vertices in *object-space,* on the other hand, in **Input** we can define the input *viewDir* which, as we already know, corresponds to the view direction in *world-space* and is used for the calculation of different lighting functions.

**Structure of a Standard Surface shader** ✕

```
struct Input
{
    float2 uv_MainTex;      // TEXCOORD0
    float3 viewDir;         // world-space view direction
    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 that we use to determine the view direction in *world-space.*

**Structure of a Standard Surface shader** ✕

```
viewDir = normalize(_WorldSpaceCameraPos - i.vertex_world);
```

249

The same goes for *worldPos* and *worldNormal,* which refer to the position of the vertices and normals in *world-space*.

<div>

**Structure of a Standard Surface shader**                                            ✖

```
worldPos = mul(unity_ObjectToWorld, v.vertex);

worldNormal = normalize(mul(unity_ObjectToWorld, float4(v.normal, 0)))xyz;
```

</div>
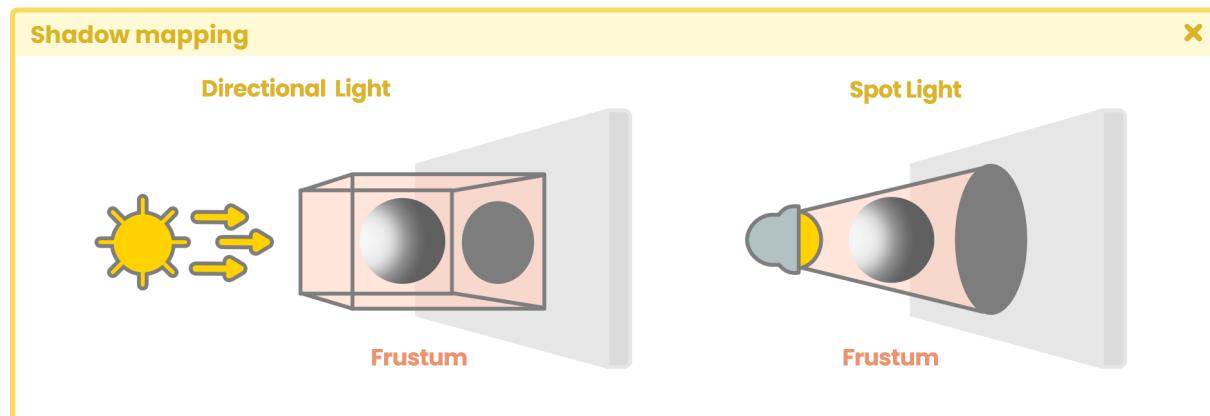
As we can see, in a surface shader we can use the same properties as in a vertex/fragment shader, with the difference that in this case, they are predefined internally.

# Shadow.

## 8.0.1. | Shadow mapping.

*Shadow mapping* is a technique that allows us to generate *shadow maps* in a scene. Its concept is quite simple: the area of light and shadow is generated concerning the frustum of illumination that we are using, that is, if the light source corresponds to a directional light, the projection of the shadow 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 or as if the source were the projection point.
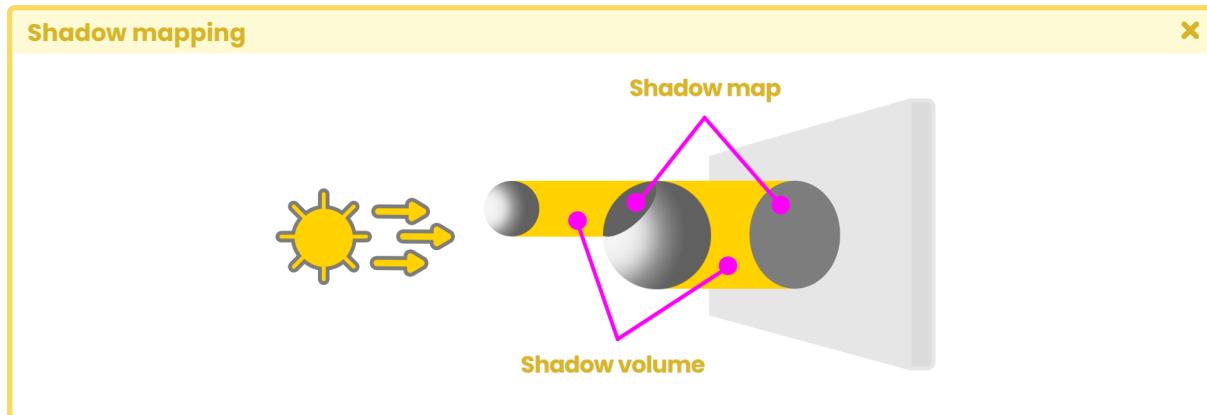


*(Fig. 8.0.1a)*

When we create a light source in our scene, the entire visible area, according to the light source viewpoint, will be the illuminated area, and the entire area outside it will be the shadow zone. According to this logic, we can determine that this corresponds to a comparison operation, but how does this process work?

To do this, we 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.

**Shadow mapping**

Shadow map

Shadow volume

*(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. In this stage, we must calculate the UV coordinates of the texture saved in memory to generate and apply the *shadow map* to the object.

## 8.0.2. | Shadow caster.

We will start with generating shadows. For this, we will create a new **Unlit Shader,** which we will call **USB_shadow_map.** In the process, we need two passes: one to cast shadows (shadow caster) and another to receive them (shadow map), therefore, the first thing we must do is include a second pass, which will be responsible for the projection of the shadows.

**Shadow caster**

```
Shader "USB/USB_shadow_map"
{
    Properties { … }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100
```

*Continued on next page.*

```
        // shadow caster Pass
        Pass { … }
        // default color pass
        Pass { … }
    }
}
```

The color pass corresponds to the default **Pass** that is included every time we create a shader. The new **Pass** will be responsible for generating the shadow projection (shadow caster), therefore, the first thing we have to do is 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** *Tags,* which in this case, must equal **ShadowCaster** in order for Unity to recognize its nature.

Naming a *Pass* is a great help when we want to use its functionality dynamically in a shader. Later we will review in detail the **UsePass** command, 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 can be omitted; however, we will use it this time to differentiate the two passes.

Because the *ShadowCaster* only corresponds to a shadow projection, we have to pass the vertex position to the *vertex shader stage* and return "zero" [0] in the fragment *shader stage.*

**Shadow caster**                                                              ✖

```
// shadow caster Pass
Pass
{
    Name "Shadow Caster"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ShadowCaster"
    }
    ZWrite On

    #pragma vertex vert
    #pragma fragment frag

    CGPROGRAM

    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;
    };

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

*Continued on next page.*

```
        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 us to adjust the projection values since they have not been defined yet

When we work with shadows we 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 our shader could take a long time, to avoid this we can work with the following macros that are included in UnityCG.cginc:

1. V2F_SHADOW_CASTER.
2. TRANSFER_SHADOW_CASTER_NORMALOFFSET(o).
3. SHADOW_CASTER_FRAGMENT(i).

**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 vertices position output (vertex : SV_POSITION), a normals output (normal_world : TEXCOORD1), a tangents output (tangent_world : TEXCOORD2) and a binormals output (binormal_world : TEXCOORD3).

**TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)** is responsible for transforming the coordinates of the vertices position to *clip-space* and also allows us to calculate the offset normal, so we can include shadows in the normal maps.

Finally, **SHADOW_CASTER_FRAGMENT** is in charge of color output for shadow projection. For Unity to compile these macros, we must make sure that we include the UnityCG.cginc directive and also use the #pragma multi_compile_shadowcaster to calculate multiple variants.

**Shadow caster** ✖

```
// shadow caster Pass
Pass
{
    Name "Shadow Caster"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ShadowCaster"
    }

    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_shadowcaster
    #include "UnityCg.cginc"

    CGPROGRAM
    struct v2f
    {
        V2F_SHADOW_CASTER;
    };
```

*Continued on next page.*

256

```
    v2f vert (appdata v)
    {
        v2f o;
        TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
        return o;
    }


    fixed4 frag (v2f i) : SV_Target
    {
        SHADOW_CASTER_FRAGMENT (i)
    }
    ENDCG
}
```

Now the s*hadow caster pass* is ready, and we can work on the included *pass* to generate a *shadow map.* To do this, we will take the default color pass included in the shader and begin the implementation.

## 8.0.3. | Shadow map texture.

Continuing with the shader **USB_shadow_map;** in this section, we will define a texture to be able to receive shadows on our object. To do this we have to include the **LightMode** in the color pass and make it equal to **ForwardBase,** this way Unity will know that this pass is affected by lighting.

**Shadow caster** ✖

```
// default color Pass
Pass
{
    Name "Shadow Map Texture"
    Tags
    {
        "RenderType"="Opaque"
```

*Continued on next page.*

```
        "LightMode"="ForwardBase"
    }
    …
}
```

Since these types of shadows correspond to a texture that is projected on UV coordinates, we have to declare a sampler2D variable. Likewise, we have to 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 caster** ✕

```
// default color Pass
Pass
{
    Name "Shadow Map Texture"
    Tags
    {
        "RenderType"="Opaque"
        "LightMode"="ForwardBase"
    }

    CGPROGRAM

    …
    struct v2f
    {
        float2 uv : TEXCOORD0;
        UNITY_FOG_COORDS(1)
        float4 vertex : SV_POSITION;
        // declare the UV coordinates for the shadow map
        float4 shadowCoord : TEXCOORD1;
    };
    sampler2D _MainTex;
    float4 _MainTex_ST;
```

*Continued on next page.*

```
    // 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 our shader *properties*, nor will we pass any texture dynamically from the Inspector, instead, we will generate a projection which will work as a texture.

So, how do we generate a texture projection in our shader?

To do this, we must understand how the UNITY_MATRIX_P projection matrix works. This matrix allows us to go from *view-space* to *clip-space,* which generates a clipping of the objects on the screen. Given its nature, it has a fourth coordinate that we have already talked about above, this refers to W, which; in this case, defines the homogeneous coordinates that allow such projection.

*Orthographic projection space*
$$W = - ((Z_{far} + Z_{near}) / (Z_{far} - Z_{near}))$$
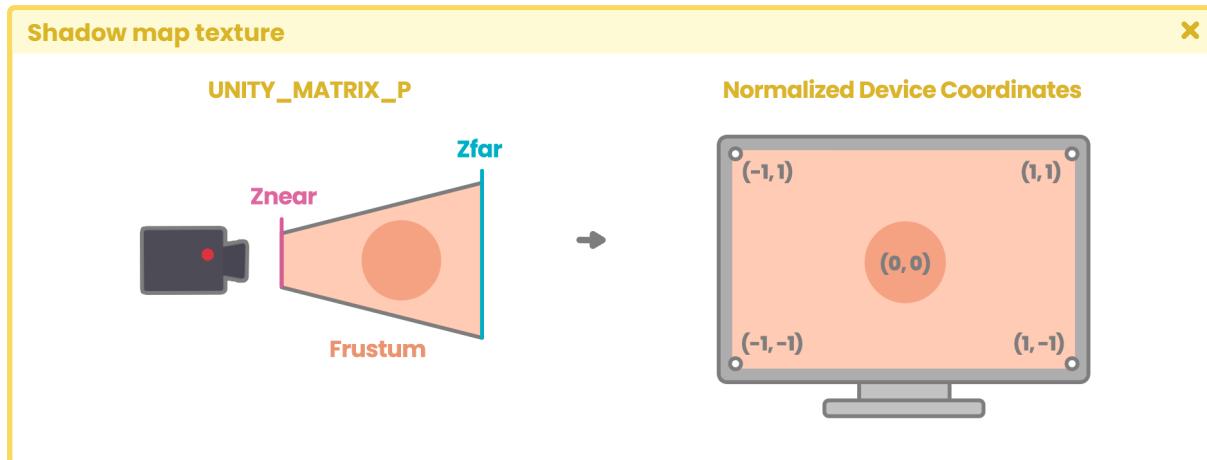
*Perspective projection space*
$$W = - ( 2 (Z_{far} \ Z_{near}) / (Z_{far} - Z_{near}))$$

As we mentioned in section 1.1.6, UNITY_MATRIX_P defines the vertex position of our object in relation to the frustum of the camera. The result of this operation, which is carried out within the UnityObjectToClipPos function, generates space coordinates called **Normalized Device Coordinates** (NDC). These types of coordinates have a range between minus one and one [-1, 1], and are generated by dividing the XYZ axes by the W component as follows:

$$\frac{projection.X}{projection.W} \quad \frac{projection.Y}{projection.W} \quad \frac{projection.Z}{projection.W}$$

*(Fig. 8.0.3a)*

It is essential to understand this process because we have to use the **tex2D** function to position the shadow texture in our shader. This function, as we already know, asks us for two arguments: The first refers to the texture itself; to the texture **_ShadowMapTexture** that we declared above, and the second refers to the texture's UV coordinates, however, in the case of a projection, we have to transform *normalized device coordinates* to *UV coordinates,* how do we do this? For this we must remember that UV coordinates have a range between zero and one [0.0f, 1.0f] and NDC; as we just mentioned, between minus one and one [-1.0f, 1.0f]. So, to transform from NDC to UV, we will have to perform the following operation:

$$NDC = [-1, 1] + 1$$
$$NDC = [0, 2] \, 2$$
$$NDC = [0, 1]$$

This operation is summarized in the following equation.

$$\frac{(NDC + 1)}{2}$$

Now, what is the NDC value? As we mentioned earlier, its coordinates are generated by dividing the XYZ axes by their component W.

Consequently, we can deduce that the X coordinate equals the X projection, divided by its W component, and the Y coordinate equals the Y projection, also divided by the W component.

$$NDC.x \; = \; \frac{projection.X}{projection.W}$$

$$NDC.y \; = \; \frac{projection.Y}{projection.W}$$

The reason we mention this concept is because, in Cg or HLSL, the UV coordinate values are accessed from the XY components, what does this mean?

Suppose, in the **float2 uv : TEXCOORD0** input, we can access the U coordinate from the **uv.x** component, likewise for the V coordinate which would be **uv.y,** however, in this case, we will 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:

$$U \; = \; \left(\frac{NDC.x}{NDX.w} + 1\right) * 0.5$$

$$V \; = \; \left(\frac{NDC.y}{NDX.w} + 1\right) * 0.5$$

## 8.0.4. | Shadow Implementation.

Now that we understand the coordinate transformation process, we can go back to our **shader USB_shadow_map** to generate a function that we will call **NDCToUV.** This will be responsible for transforming from *Normalized Device Coordinates* to UV coordinates. This function will be used in the *vertex shader stage,* so it will have to be declared above that stage.

```
// declare NDCToUV above the vertex shader stage
float4 NDCToUV(float4 clipPos)
{
    float4 uv = clipPos;
    uv.xy = float2(uv.x, uv.y) + uv.w;
    uv.xy = float2(uv.x / uv.w, uv.y / uv.w) * 0.5;
    return uv;
}
// vertex shader stage
v2f vert(appdata v) { … }
```
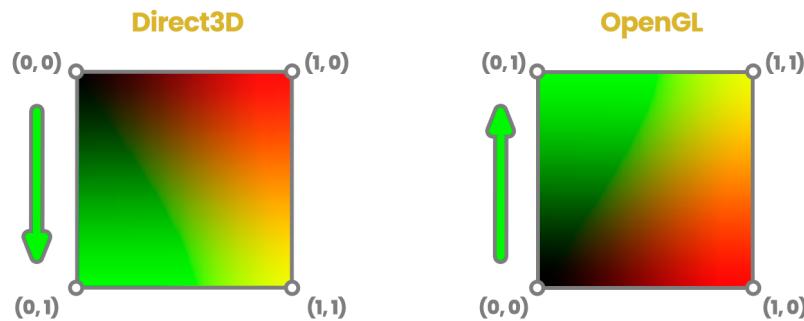
The previous example has declared the **NDCToUV** function to be type **float4,** that is, a four-dimensional vector (XYZW).

A new four-dimensional vector called **clipPos** has been used as an argument referring to the vertices output position; the result of the UnityObjectToClipPos(v.vertex) function.

Inside the function, the mathematical operation mentioned in the previous section is then translated into code. Despite this, the operation is not complete because there are some factors that we are not considering in the implementation of the function. These factors refer to the platform where we compile our code and the half-texel offset.

It is worth mentioning that there is a coordinate difference between OpenGL and Direct3D. In the latter, the UV coordinates start at the top right, while in OpenGL they start at the bottom right.



*(Fig. 8.0.4a)*

This factor generates a problem when implementing a function in our 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 depending on its coordinates, 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 $1/Z_{far}$ operation.

In this case, we will use **_ProjectionParams.x** since we only need to turn the V coordinate depending on its starting point. If we look at the previous image, we will notice that the U coordinate maintains its position regardless of the platform on which we are compiling.

Using the concept above, _ProjectionParams.x_ will equal "one" [1] when the shader is compiled in OpenGL, or "minus one" [-1] if compiled in Direct3D. Taking this factor into consideration, our operation would be as follows:

**Shadow Implementation** ✕

```
float4 NDCToUV(float4 clipPos)
{
    float4 uv = clipPos;
    uv.xy = float2(uv.x, uv.y * _ProjectionParams.x) + uv.w;
    uv.xy = float2(uv.x / uv.w, uv.y / uv.w) * 0.5;
    return uv;
}
```

As we can see, the V coordinate of the UV has been multiplied by _ProjectionParams.x. In this way we can flip the matrix in case our shader is compiled in Direct3D, now we 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, we will use the internal variable **_ScreenParams**, which, like _ProjectionParams_, is a four-dimensional vector that has a different value in each coordinate.

The value that we are going to use is _*ScreenParams.zw* since, Z equals one plus one, divided by the width of the screen (1.0f + 1.0f / width) and W equals one plus one, divided by the height of the screen (1.0f + 1.0f / height).

Taking into consideration the *half-texel offset,* our function will be as follows:

**Shadow Implementation** ✕

```
float4 NDCToUV(float4 clipPos)
{
    float4 uv = clipPos;
    #if defined(UNITY_HALF_TEXEL_OFFSET )
        uv.xy = float2(uv.x, uv.y * _ProjectionParams.x) + uv.w *
        _ScreenParams.zw;
    #else
        uv.xy = float2(uv.x, uv.y * _ProjectionParams.x) + uv.w;
    #endif
        uv.xy = float2(uv.x / uv.w, uv.y / uv.w) * 0.5;
    return uv;
}
```

Now the shadow UV coordinates are working perfectly. So we must declare them in the *vertex shader* stage, for this, we will make the shadowCoord output equal to the **NDCToUV** function, and pass the output vertices as an argument; those which were clipped.

**Shadow Implementation** ✕
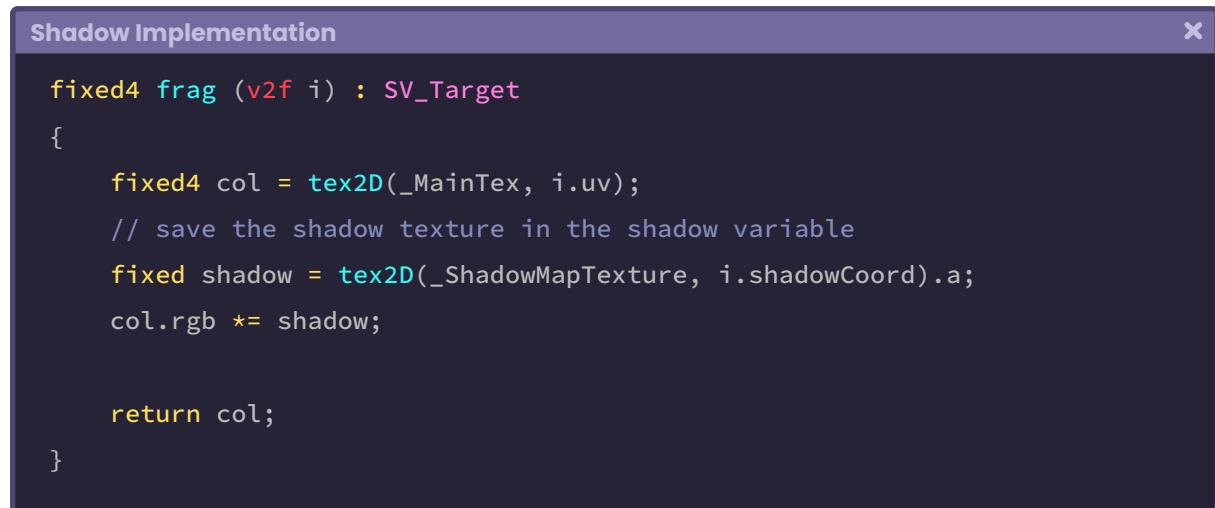
```
float4 NDCToUV() { … }

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

    return 0;
}
```

At this point, *shadowCoords* has the projection coordinates, and now we can use them as UV coordinates for the _ShadowMapTexture.

As we already know, the tex2D function asks for two arguments: the texture and its UV coordinates. We 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);
    // save the shadow texture in the shadow variable
    fixed shadow = tex2D(_ShadowMapTexture, i.shadowCoord).a;
    col.rgb *= shadow;

    return col;
}
```

In the previous example, we created a dimension variable called **shadow,** which saves the sampling values for the **_ShadowMapTexture**.

We can notice that in difference to the vector *col,* the variable *shadow* has only one dimension, why?

We must remember that a shadow texture only has the colors black and white, therefore, we only store the Alpha channel within the *shadow* variable since it gives us a range from zero to one [0, 1].

### 8.0.5. | Built-in RP shadow map optimization.

The process of implementing shadows that we saw earlier for color passing, can be optimized through the use of macros included in Unity, these macros refer to:

- SHADOW_COORDS(n)
- TRANSFER_SHADOW(output)
- SHADOW_ATTENUATION(output)

We can replace some functions and/or variables to optimize its process, in the same way, we did in the *shadow caster* pass.

If we want to use these macros, we will need the file #include "AutoLight.cginc" in our program, in addition to #pragma multi_compile_fwdbase, which 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_fwdbase nolightmap nodirlightmap nodynlightmap
    novertexlight

    #include "UnityCG.cginc"
    #include "AutoLight.cginc"
```

The variables defined after the #pragma (nolightmap, nodirlightmap, nodynlightmap and novertexlight) correspond to optional parameters that we can define to add or remove functionality in the shadow behavior.

Note that, if we use the macros for the implementation of the *shadow map,* we must use some predefined names in the vectors that we use as input/output, otherwise our code will generate an error.

**Built-in RP shadow map optimization** ✕

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

By default, our code includes the vector **uv**, with its semantics **TEXCOORD[n]** for the *vertex input.* On the other hand, if we use the macros, we must replace the word **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 for the *shadow map.* Furthermore, we have to include the macro SHADOW_COORDS(n), which includes the UV coordinates (o.shadowCoord) that we 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, we can synchronize the values in the *vertex shader stage.*

```
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(output) macro is the same as the **NDCToUV** operation we performed in the previous section. Basically, it calculates the UV coordinates for the shadow texture. Now we can transfer the coordinates to the *fragment shader stage.*

```
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;
}
```
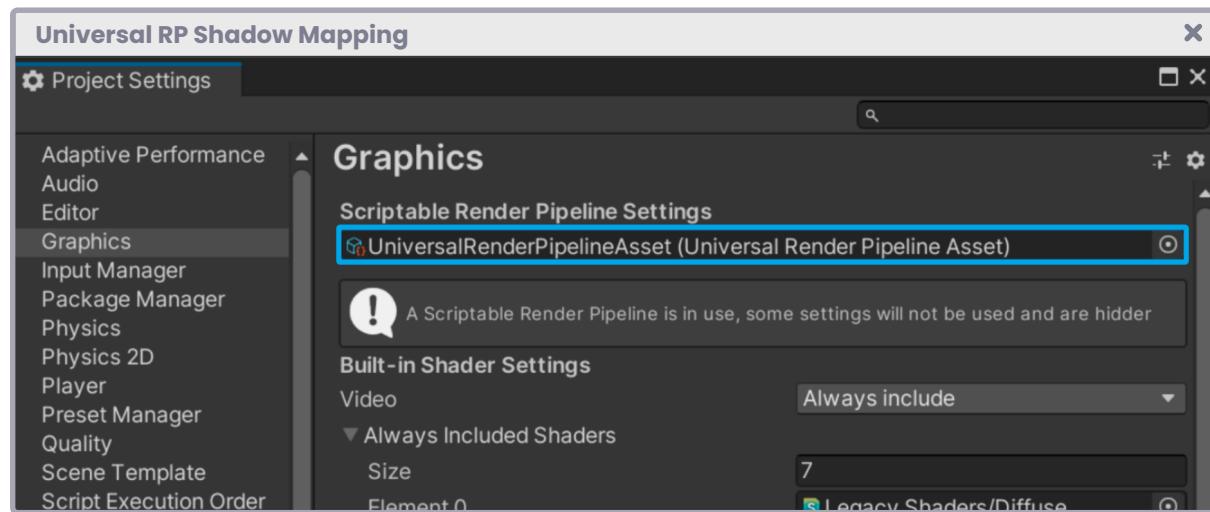
Finally, SHADOW_ATTENUATION(output) contains the texture and its projection. This function can be saved as a one-dimensional vector, since only one channel (Alpha) will be used in the texture projection.

## 8.0.6. | Universal RP Shadow Mapping.

The technique we use for the shadow calculation can vary depending on the type we want to implement. The process we use is the optimal for Unity, however, its implementation may vary depending on the rendering 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 we want to generate shadows in Universal RP, we have to use the dependency "Lighting.hlsl", as this package includes definitions for the coordinate calculations.

We will start by configuring the rendering pipeline in Universal RP. To do this, we must make sure that we have assigned a **Render Pipeline Asset** in the **Scriptable Render Pipeline Settings** box, in the **Graphics** menu.



*(Fig. 8.0.6a)*

Once configured, we will create a new **Unlit Shader,** which we will call **USB_shadow_map_ URP**. This shader will be used to implement both a shadow map *and* the *shadow caster* function in the rendering engine. Likewise, we will review the necessary dependencies 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 we already know, shadows are produced in two passes: one for shadow caster and one for texture (shadow map). By default, Unity adds only one pass which; again, we could use for the *shadow map.* This way, we have to add an extra pass in our shader for the *shadow caster* definition*.*

In Universal RP, there is a shader called "**Lit**", which has been included in the category "**Universal Render Pipeline**".

Within this shader, we can find a pass called "**ShadowCaster**" (Name "ShadowCaster") that is responsible for the calculation of coordinates for the projection of shadows on other objects.

In Unity, we can dynamically include passes using the **UsePass** command, what does this mean?

On the one hand, we could go to the *Lit* shader, copy the pass, and paste it into our shader, or we 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 we want to include functionalities of a pass that is in a different shader to the one we are programming. This means that, if we take 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.

```
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 we have included this path, we must define the default pass as the one that will process the *shadow map,* that is, we 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, we must add some dependencies that will help us to implement the *shadow map.*

**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.hlsl" and "Lighting.hlsl" have several functions that we will use in the calculation, among them: the **GetVertexPositionInputs** function, which belongs to a sub-dependence called "ShaderVariablesFunctions.hlsl" and **GetShadowCoord** that is included in a sub-dependence called "Shadows.hlsl".

Because we are going to implement a *shadow map,* we will need a vector that can store its UV coordinates, for this we can create a four-dimensional vector in the *vertex output.*

<div>

**Universal RP Shadow Mapping**                                             ✕

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    float4 shadowCoord : TEXCOORD1;
};
```

</div>

As detailed in section 8.0.4, the **shadowCoord** vector has four dimensions because it will store the result of the vertices' transformation from NDC to UV coordinates.

For the coordinates, we will have to use the **GetShadowCoord** function that asks us for a **VertexPositionInputs** type object as an argument.

<div>

**Universal RP Shadow Mapping**                                             ✕

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

    // you can find VertexPositionInputs at Core.hlsl
    // you can find GetVertexPositionInputs at
    // ShaderVariablesFunctions.hlsl
    VertexPositionInputs vertexInput =
    GetVertexPositionInputs(v.vertex.xyz);
```

*Continued on next page.*

</div>

```
    // you can 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 we simply have to pass it as an argument to the **GetMainLight** (float4shadowCoord) function that has the calculations for light direction, attenuation, shadow attenuation and light color. This function is 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* function has up to three variations, we can include the *shadowMask* in the third, in case we need it. Now we simply have to create a vector to store the shadow attenuation and multiply it by the texture RGB color.

**Universal RP Shadow Mapping**                                    275   ✖

```
fixed4 frag (v2f i) : SV_Target
{
    // you can 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;
}
```
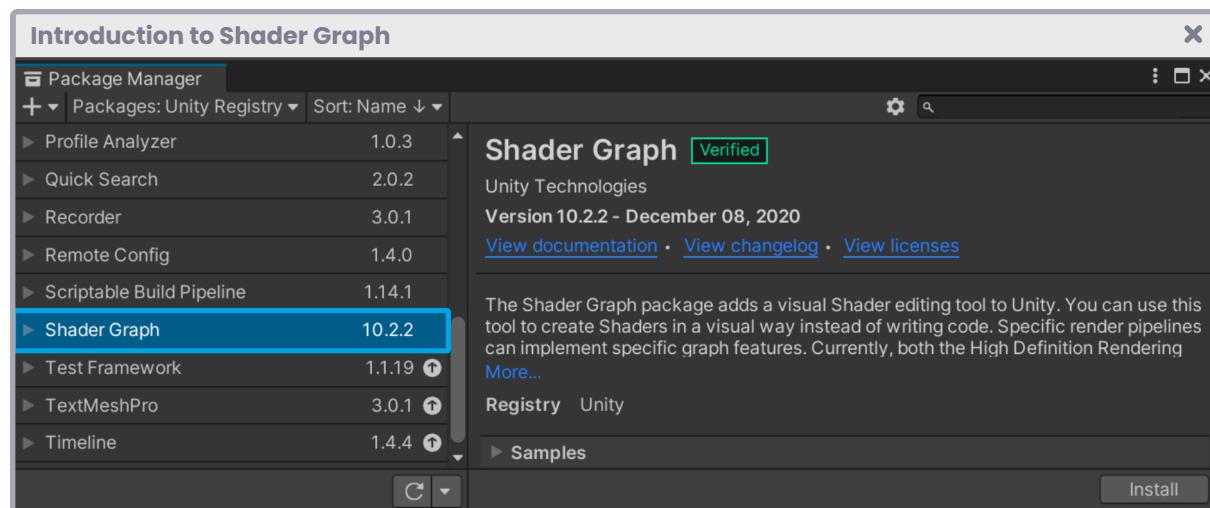
# Shader Graph.

## 9.0.1. | Introduction to Shader Graph.

Up to this point, we've reviewed much of the rendering pipeline structure, as well as understood how a Unity shader works. Now we will introduce **Shader Graph,** since its structure and analogy are based on all the previous knowledge we have acquired.

Shader Graph is a **package** that adds support for a visual node editing tool. Based on HLSL, its interface can be 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 us to generate specific functions within the program.

Currently, Shader Graph is available for two rendering modalities, these are High Definition RP and Universal RP.
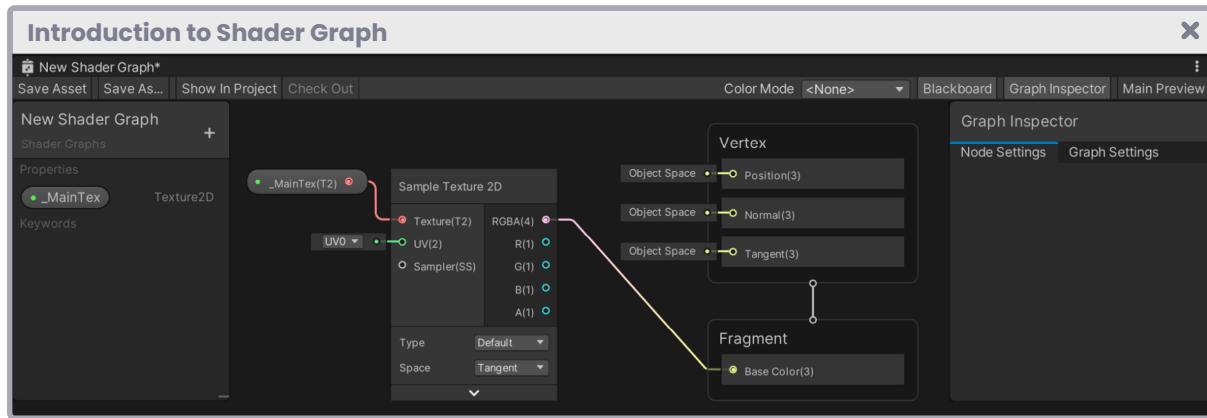


*(Fig. 9.0.1a)*

We must consider that when working with Shader Graph, the versions developed for Unity 2018 are BETA versions and do not receive support, whereas the versions developed for Unity 2019.1 and onwards are actively compatible and do receive support.

Another consideration is that it is very likely that our shaders created with this interface are not compiled correctly in their different versions. This is because new features are added in each update, which makes our node set stop compiling in most cases, even more so if we 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 the shader creation to be more intuitive.



*(Fig. 9.0.1b)*

Before introducing this matter, 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 corresponds to **12.0.0.**

If we create a node within this version we can see that the *vertex and fragment shader stage* appear separate and work independently, however, if we go to version **8.3.1,** both stages are merged within a node called "**Master**", which refers to the final shader output color.
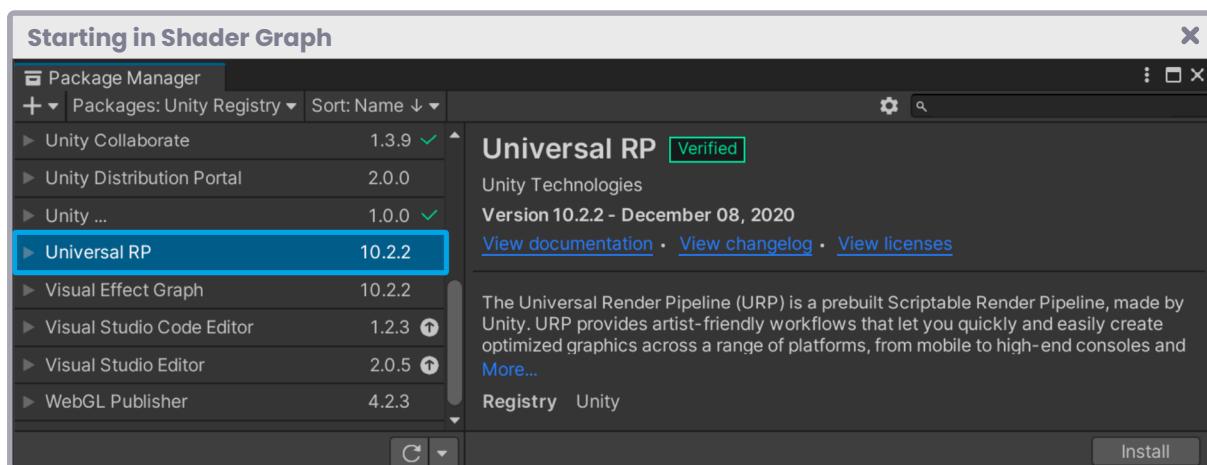
As we mentioned before, it is very possible that the shaders created in this interface do not compile in all its versions, in fact, if we create a shader in version 8.3.1 and update to version 12.0.0, there are probably functional changes that prevent its compilation process.

## 9.0.2. | Starting in Shader Graph.

There are two ways to include Shader Graph in our project:

1. From the default settings, when we create 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 we start a project in Universal RP or High Definition RP, this package is included by default, and we can use it immediately. This means that in the menu: Assets / Create / Shaders, we can find a new section of shaders that work exclusively for these types of rendering.



*(Fig. 9.0.2a)*

What if we 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, we can upgrade to Universal RP. To do this, we must follow the second step that was described in the inclusion of Shader Graph in the project. Likewise, it will be necessary to include the Universal RP package, which adds support to this type of rendering 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 rendering pipeline for high-end video games and can only be started as a project from the Unity Hub.
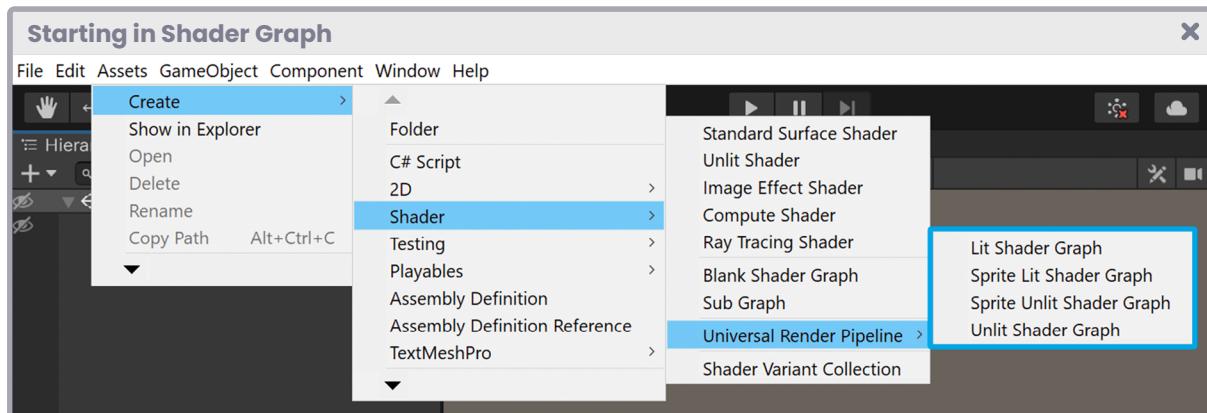
Once we have installed both packages, we will have to create a type of object called **Pipeline Asset** and another called **Forward Renderer Data** from the menu: Assets / Create / Rendering.

Generally, when we create a *Pipeline Asset,* Unity creates a *Forward Renderer Data* file by default, which works as *Render Path* for Universal RP.

Having these files, we must go to the **Project Settings** window and in the **"Graphics"** option 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, we must 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 with the 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.
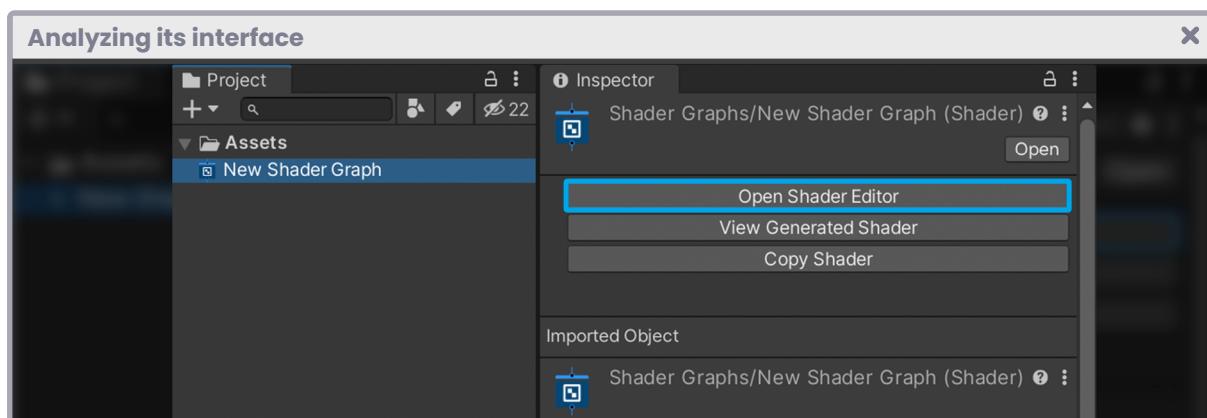


*(Fig. 9.0.2b)*

Please consider that the process and steps mentioned above may change depending on the version of Unity, however, its creation and installation analogy is the same.

## 9.0.3. | Analyzing its interface.
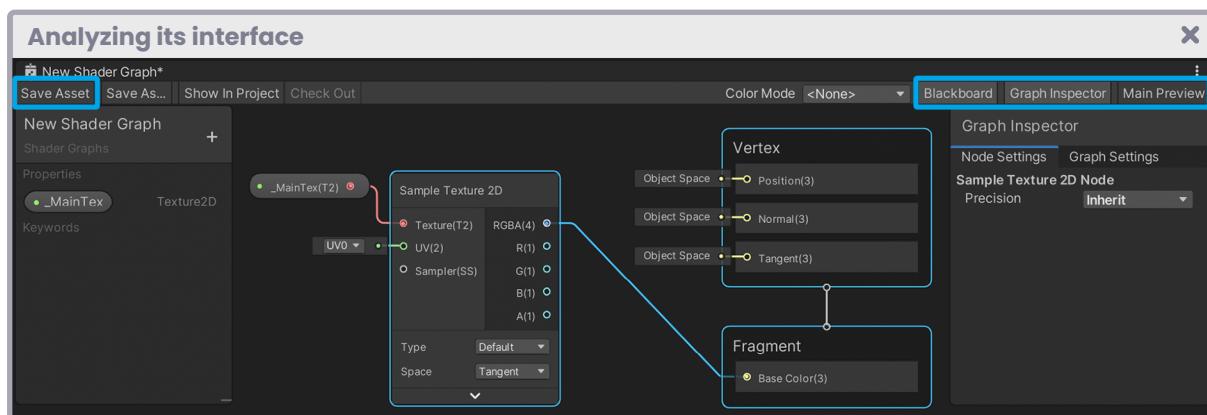
As we already know, the Shader Graph interface has structural changes depending on the version we are using. In this section, we will detail update number 10.4.0 that corresponds to the package included in Unity 2020.3.1f1. Now, regardless of its reference, the analogy between the different versions 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, we must have previously created a shader belonging to Universal RP or High Definition RP, as appropriate. Once we have a shader in our Project, we 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 we 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 the example above.

*Save Asset* allows us 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.
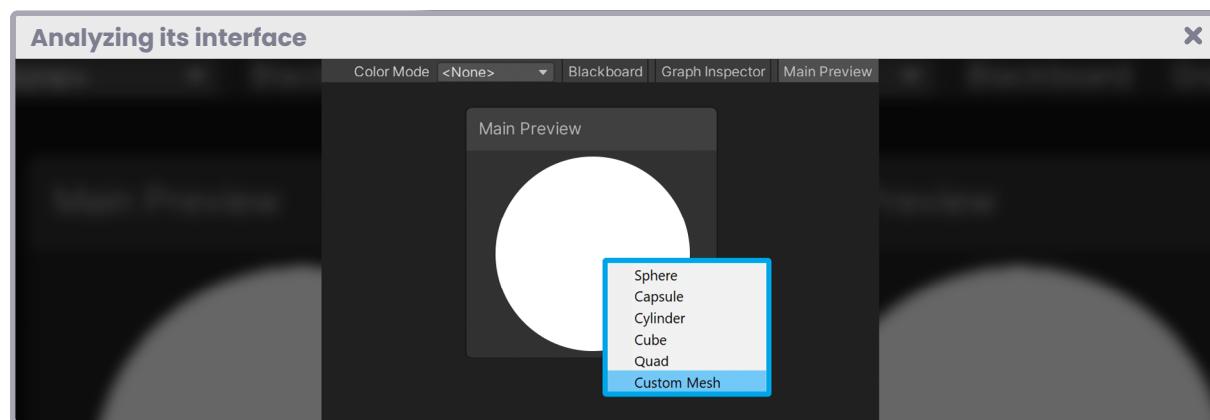
*Blackboard* is the direct analogy of *properties* in *ShaderLab.*

As we already know, if we want to create a property in a shader via code, we must add it within the Properties in declarative language. In Shader Graph it is practically the same with the difference that they are added in the *Blackboard* by pressing the "**plus**" button located at the top right of the menu.

*Graph Inspector* is a small panel that allows you to modify the node and general configuration of the shader. This panel has two tabs which are: **Node Settings** and **Graph Setting;** both with different functionalities.

*Node Settings* allows you to name properties, references, default values, mode, precision, and others, while *Graph Settings* allows you to define the general shader configuration, e.g. if our shader will be transparent, additive or opaque.

Finally, *Main Preview* activates or deactivates the node configuration **preview**, ideal for previewing the effect (set of nodes) that are being developed. An interesting feature of the preview pane is that it allows you to import custom objects to see the actual representation of the effect on an element. To import a custom object, simply right-click on the preview area and select "Custom Mesh".



*(Fig. 9.0.3c)*

### 9.0.4. | Our first shader in Shader Graph.

To test our shaders, we will work with Universal RP.
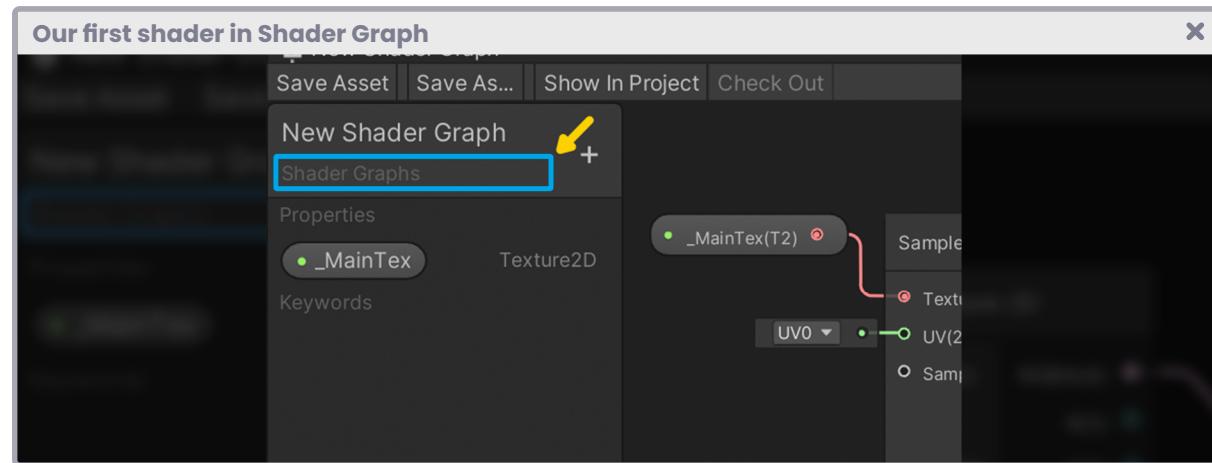
We will start our adventure by going to the Unity *Project* window and creating an **Unlit Shader Graph**, located by the following route: Create / Shader / Universal Render Pipeline. Please note: this route will only be visible if we have the Shader Graph package included in our project.

Next, we will recreate in Shader Graph the **USB_simple_color** shader that we started in section 3.0.1. To do this, we will name this new shader **USB_simple_color_SG** since it will have the same characteristics, although now being created through nodes instead of code.

At first glance, the first difference we 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 its nature.

It should be noted that, as in previous programs, in Shader Graph we can also modify the path in the Inspector.

By default, all the shaders that we create with this interface will be saved in the **Shader Graphs** path, however, if we want to modify their destination, we can do it directly from the access to the path which is found in the Blackboard, under the name of the shader.
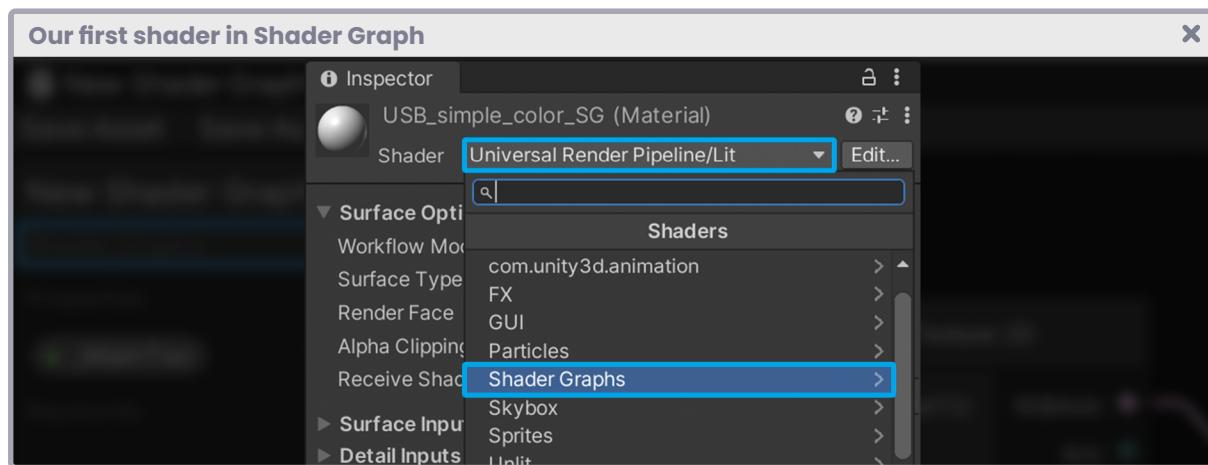


*(Fig. 9.0.4a)*

The above operation is the equivalent to changing the path of a program with the ".shader" extension.

**Our first shader in Shader Graph**                                    ✕

```
Shader "Shader Graphs/USB_simple_color_SG"

{

    Properties { … }
    SubShader { … }

}
```

**Our first shader in Shader Graph**                                    ✕
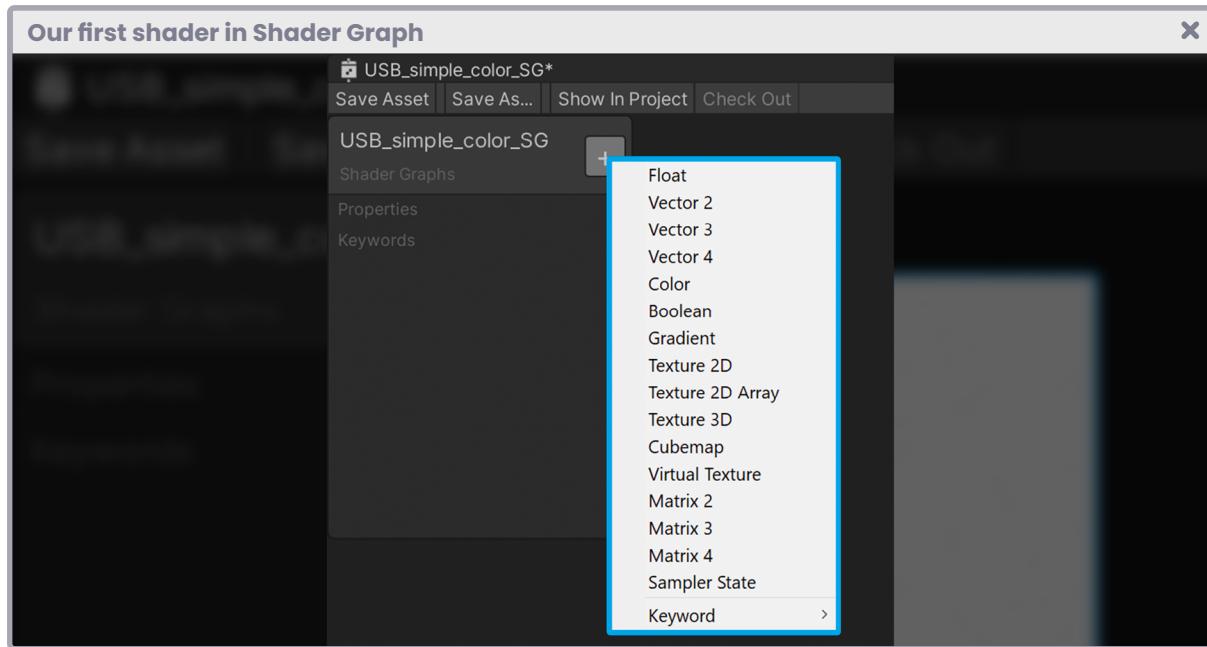


*(Fig. 9.0.4b)*

As we could see in previous sections, every time we created an **Unlit Shader,** Unity added a default property corresponding to a texture called **_MainTex.** Shader Graph cannot do this. When we start the interface, the Blackboard, where properties are added, is empty by default. This means that we will have to add each property to the shader independently.

Open the shader **USB_simple_color_SG** by double-clicking on it.
As we already know, in ShaderLab, the properties are added within the *Properties'* field via code, however, in Shader Graph they are added from the *Blackboard* by pressing the **plus** button on the panel, which is on the left side of the interface.
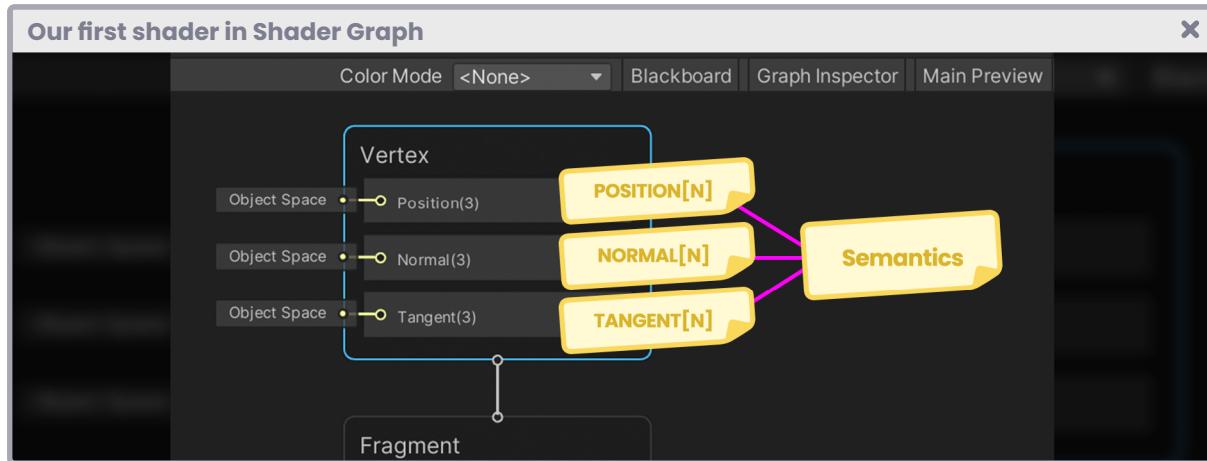
*(Fig. 9.0.4c)*

Before starting, we will make a small introduction to the vertex/fragment shader stage to understand how it works with this interface.

As we 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 we 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.*
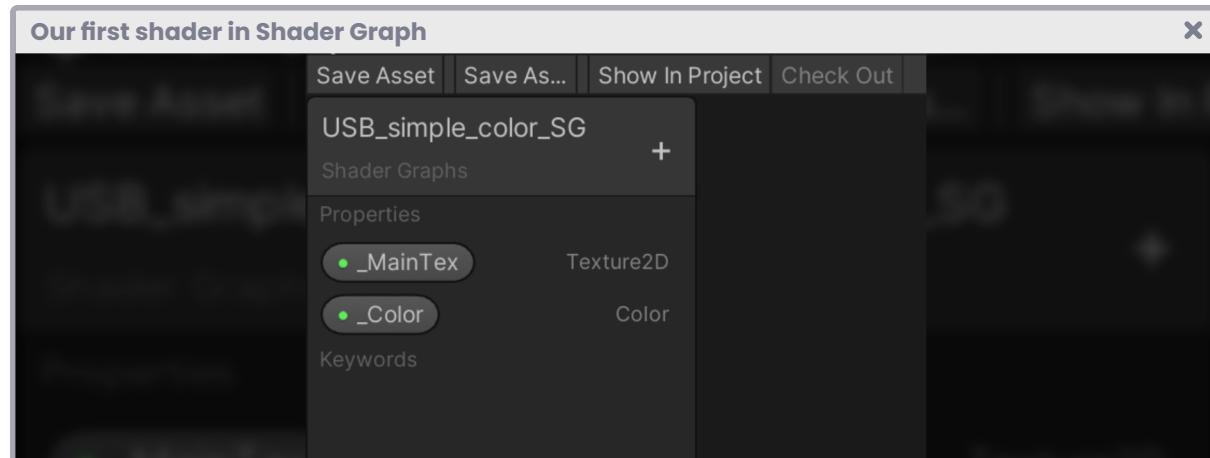
Why has Shader Graph got three dimensions but Cg or HLSL up to four?

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 W equals "zero" the vector corresponds to a direction in space.

Considering the above explanation, we can conclude that, if the input Position had four coordinates, then these would be, e.g., float4(X, Y, Z, 1) and for the Normals float4(X, Y, Z, 0), 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.
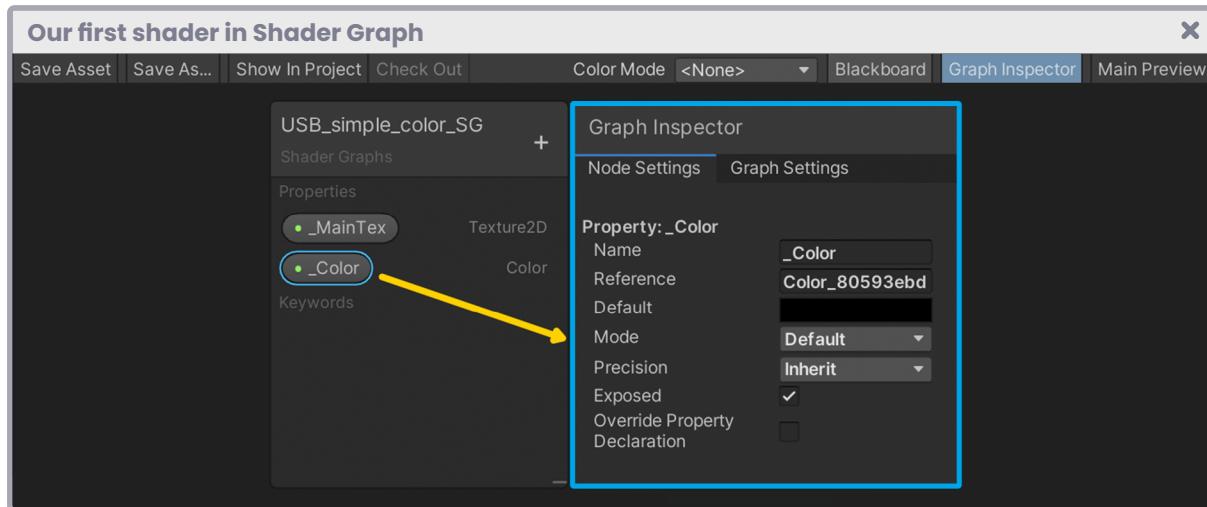
To start our program, the first thing we will do is go to **the Blackboard** and create two properties: a color type, which we will call **_Color**, and a Texture2D type; which we will call **_MainTex.**



*(Fig. 9.0.4e)*

A question that frequently arises is, what is the default color of our *property _Color* or even *_MainTex?*

When we declare a property in ShaderLab we can define its tonality by its value (e.g. (1, 1, 1, 1), "white"), however, in Shader Graph it is different. By default, our *property _Color* is black (0, 0, 0, 1), this can be corroborated by going to the **Graph Inspector; Node Settings** tab. If we look at the "Default" property, we will see that it is set to black.
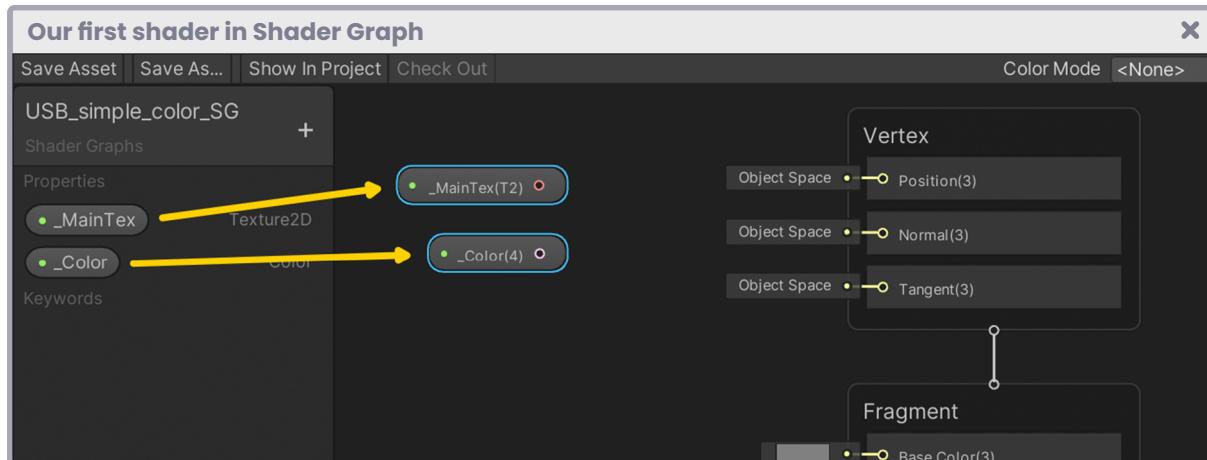
(Fig. 9.0.4f)

We 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, we can select its "Mode", which equals "white" by default.

To generate communication between the ShaderLab properties and our program, we must create connection variables within the CGPROGRAM field.

This process is different in Shader Graph. What we must do is drag the properties that we have in the *Blackboard* to the node area.
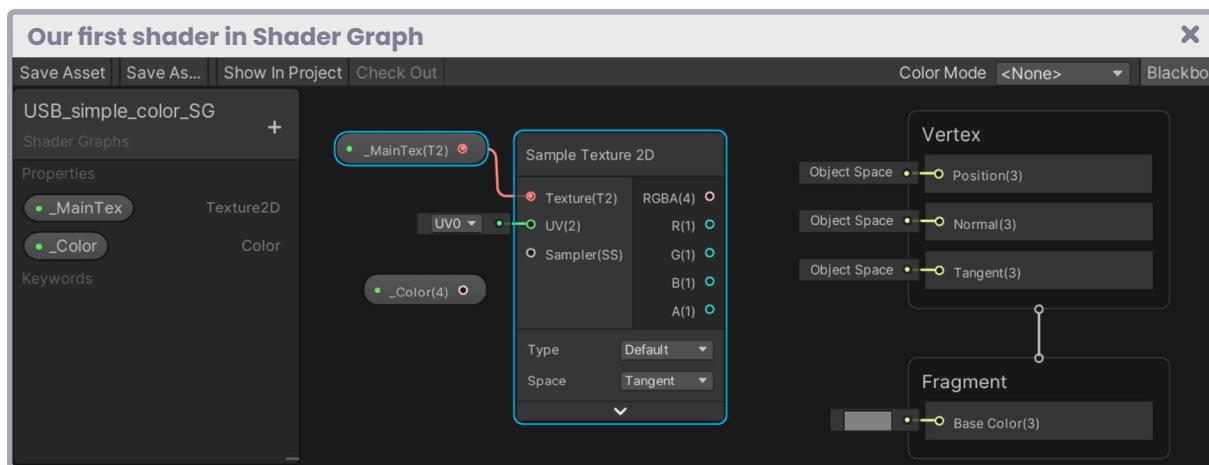


(Fig. 9.0.4g)

As we saw in chapter I, section 3.2.7, we 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 **tex2D.** If we want to work with this node, we can do two actions:

1. Press the "space bar" on the node area and write the name of the node we 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 we want to work with.
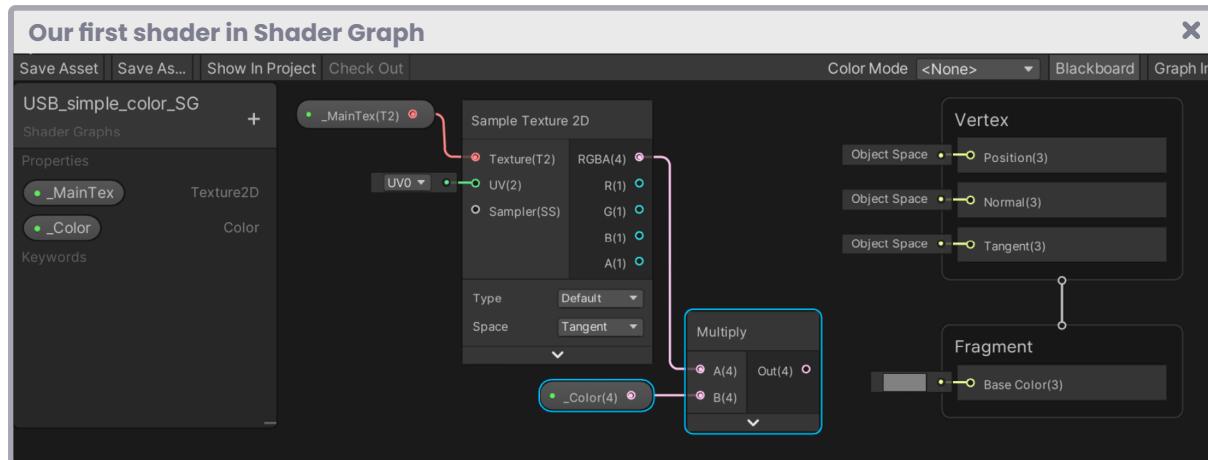
All we have to do for the Texture2D type texture to work in conjunction with the Sample Texture 2D node, is to connect the output of the _MainTex property with the Texture(T2) type input, which is included in the Sample Texture 2D.



*(Fig. 9.0.4h)*
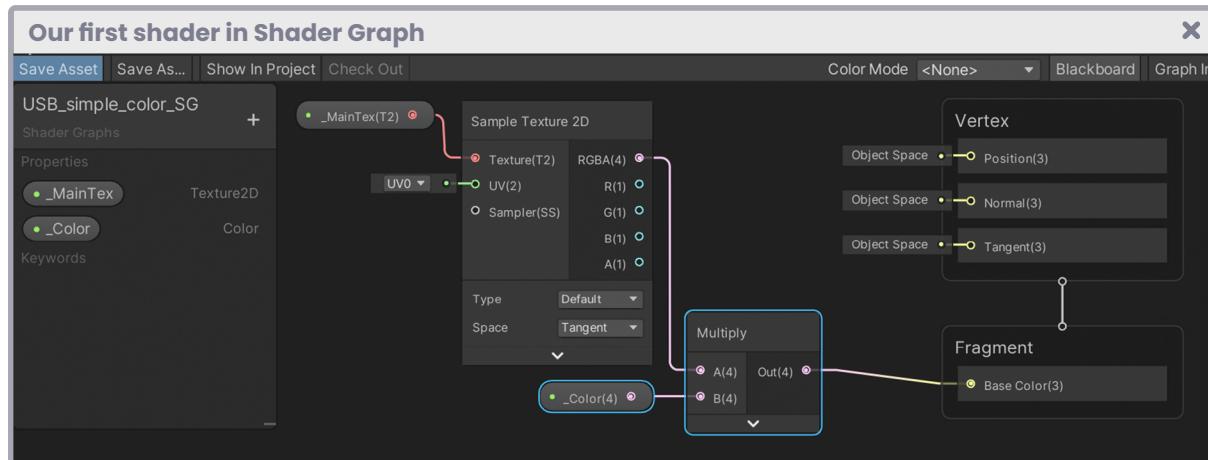
This operation is equivalent to creating a four-dimensional vector and passing both the texture and the uv coordinates input to it (e.g. float4 col = tex2D(_MainTex, i.uv)).

If we want to change the tint of the texture, all we have to do is 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, we 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, we must save our shader by pressing the **Save Asset** button that is located at the top left of the Shader Graph interface.



*(Fig. 9.0.4j)*

Something we must consider is that the *Base Color* is an input that corresponds to the final RGB color of our shader and is part of the operations that are occurring within the *fragment shader stage.* This means that the final color output (e.g. color + alpha + blending) would equal the SV_Target semantics that we can find in a ".shader".

Later we will see in detail how to add more operations in Shader Graph.
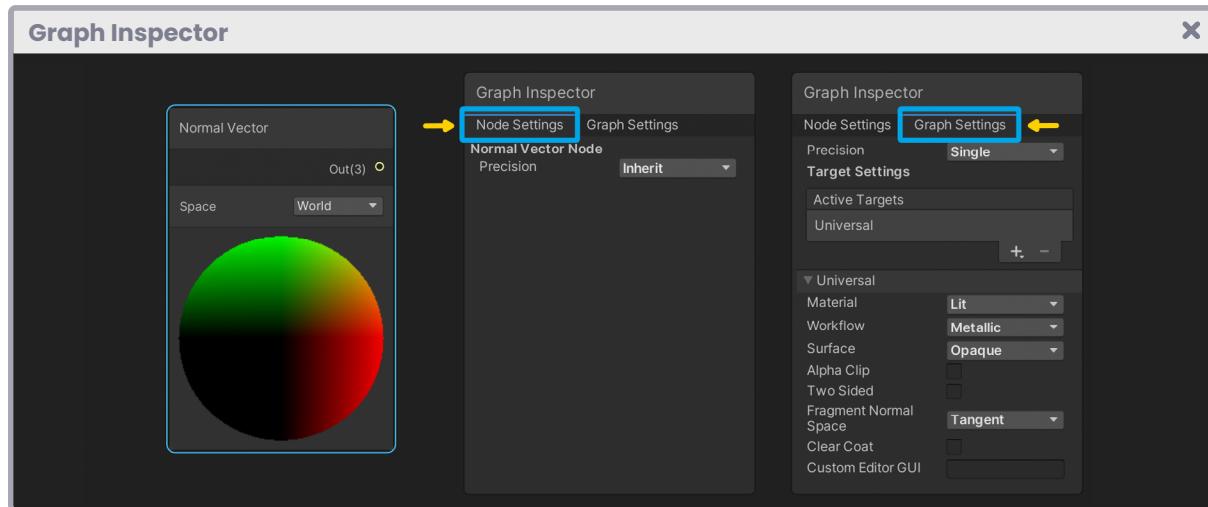
## 9.0.5. | Graph Inspector.

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.

According to the official documentation in Unity;

> *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.*

To summarize, this panel, divided into two sections called **Node** and **Graph**, has those configurable properties that allow modifying the color output. We can find blending, cull, and alpha clip options among them. Also, we can adjust the properties of the nodes in our configuration.
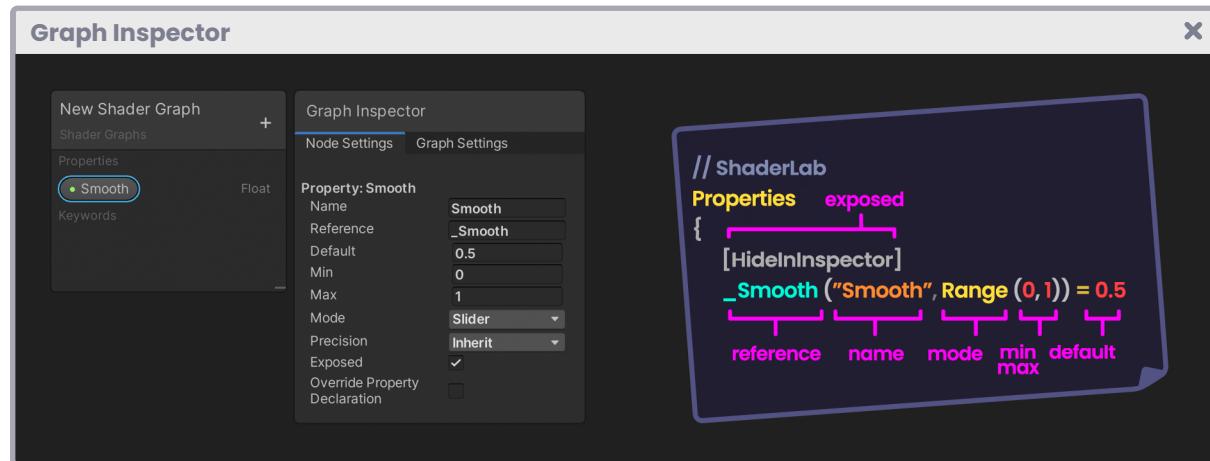


*(Fig. 9.0.5a)*

On the one hand, Graph Settings has the general properties of the node, in other words, those commands that we use previously to define the behavior of the screen pixels, e.g., the Blend [SourceFactor] [DestinationFactor] command is enabled once we determine the type of "surface" in the shader.  Likewise, we can enable or disable the Cull command through the "Two Sides" checkbox.

On the other hand, **Node Settings** contains the attributes of those selected elements, e.g., in the Custom Function node, the precision, and preview options appear in it, as well as those inputs and outputs that can be configured according to the operation being carried out. For the remaining ones, only the precision and preview options appear.

This same behavior is reflected in the properties we have added to the Blackboard. As we can see in Figure 9.0.5b, in the tab appear those attributes that we added manually in the writing of a shader.



*(Fig. 9.0.5b)*

The "precision" attribute refers to the calculation in decimal places of a node on the GPU. By default, it is set to "inherit." However, we can change it to "single" (float), which contains six decimal places of precision, or to "half," which has only three decimal places.

## 9.0.6. | Nodes.

In the first chapter, we reviewed the operations performed by some intrinsic functions, among them: clamp, abs, min, max, and many others. The nodes in the Shader Graph are the graphical representation of these functions; therefore, they fulfill the same functionalities, e.g., according to the official Unity documentation, the Clamp node is articulated as follows:
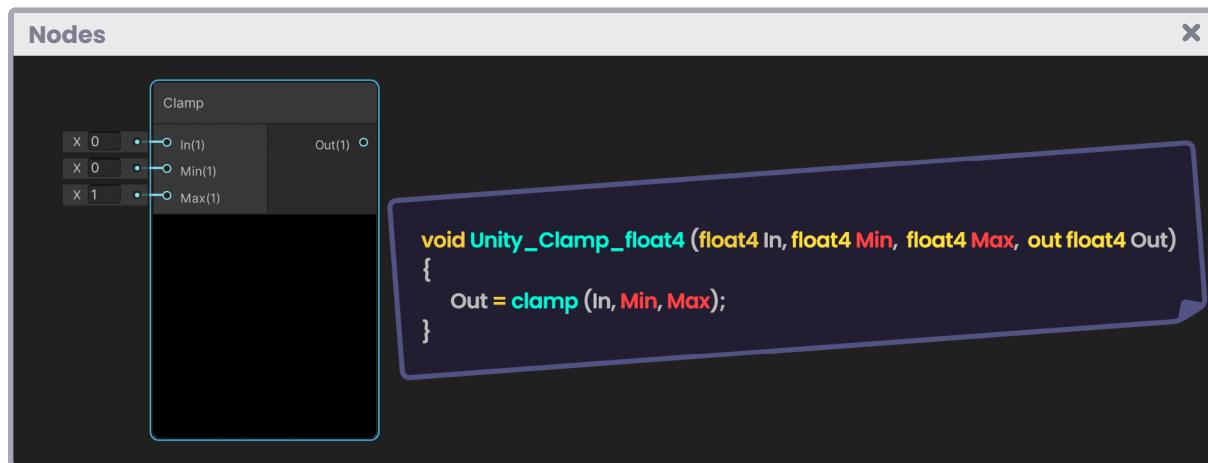
> *The following example code represents*
> *a possible outcome of this node.*

```
Nodes                                                          ✕

 void Unity_Clamp_float4(float4 In, float4 Min, float4 Max, out float4 Out)
 {
      Out = clamp(In, Min, Max);
 }
```
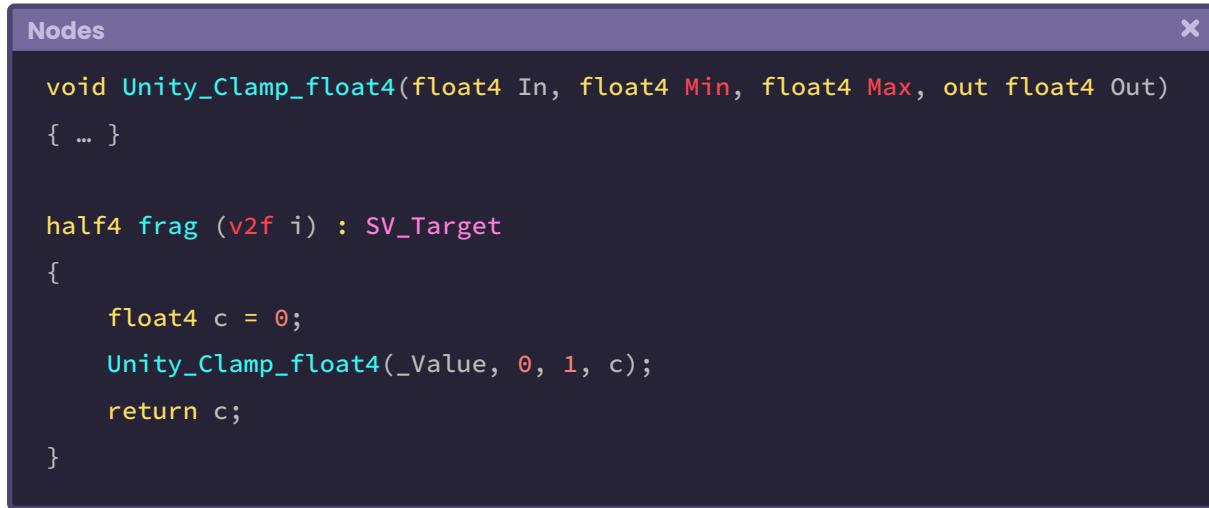
we can see in the previous example, the **Unity_Clamp_float4** function of type "*void*" has three inputs corresponding to four-dimensional vectors. Among them, we can find "**In**, **Min** and **Max**." These same values can be seen graphically in the construction of the node, as shown in Figure 9.0.6a.

As for the output called "**Out**," it simply has the operation to be carried out.



*(Fig. 9.0.6a)*

Since Shader Graph is based on HLSL language, we can use these same functions inside a shader with "*.shader*" extension, following the points mentioned in section 4.0.4.
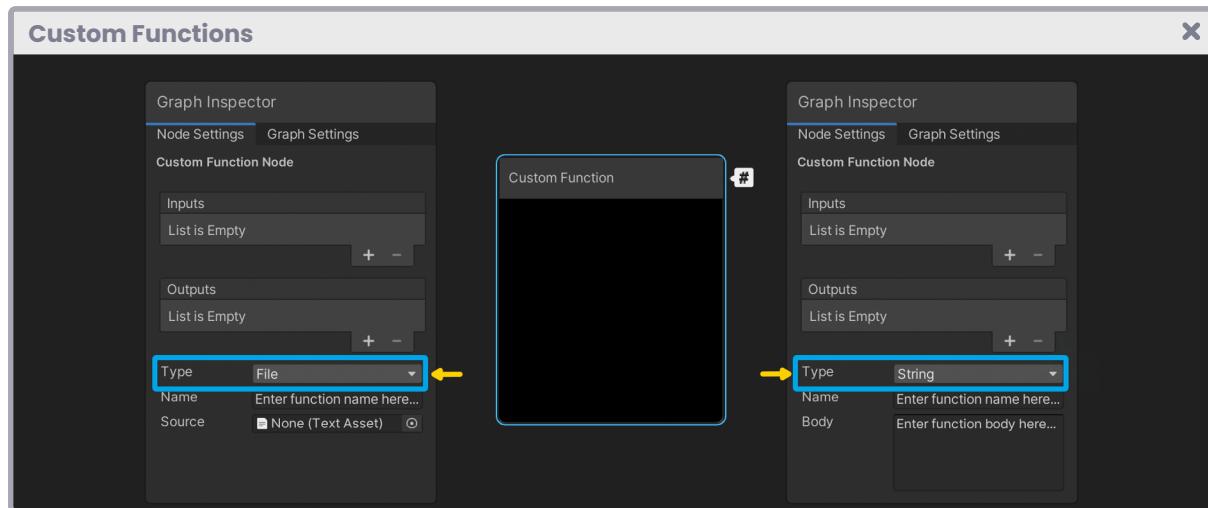
```
Nodes                                                                    ✕

void Unity_Clamp_float4(float4 In, float4 Min, float4 Max, out float4 Out)

{ … }


half4 frag (v2f i) : SV_Target

{

    float4 c = 0;

    Unity_Clamp_float4(_Value, 0, 1, c);

    return c;

}
```

## 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 allows us to create our own functions and work with them in Shader Graph. We can use it to create custom lighting, complex operations, or optimize long processes.
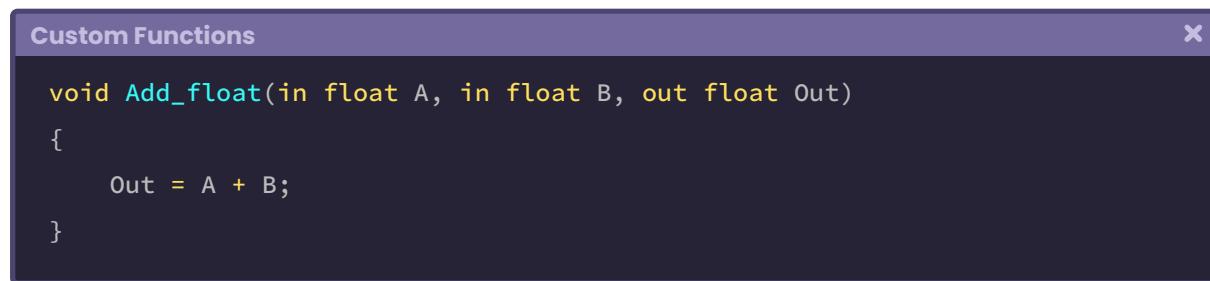
Given its structure, we can work with this node in two ways: from files with ".hlsl" extension (type file) or by writing functions directly in its body (type string), however, it is advisable to use the first option because we can reuse the files later in other projects.
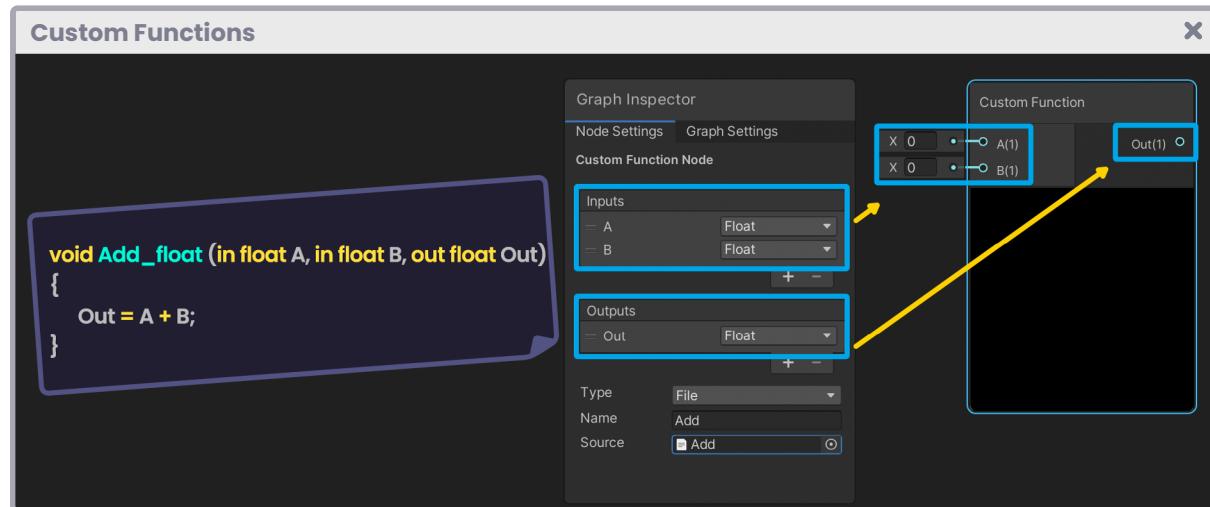


*(Fig. 9.0.7a)*

By default, Custom Functions has no input or output. Initially, we must go to the Graph Inspector and add the necessary properties for the function. 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, we must press the configuration button (gear button) at the top right of the node to perform the same action.

We will analyze the implementation of a simple function to understand the concept:

```
void Add_float(in float A, in float B, out float Out)
{
    Out = A + B;
}
```

In the inner exercise, the "Add" function simply returns a scalar value (Out) equal to the sum of two numbers. The first input we can find corresponds to the floating type variable "A," and the second is "B." Therefore, in our Custom Function node, we 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" were a three-dimensional vector, then such a vector would have to be added as input in the node configuration.

Next, we will recreate the behavior of the **_WorldSpaceLightPos** variable that we initially mentioned in section 7.0.3 when we talked about diffuse reflection, which we will use for the same purpose. Let's start by creating a new shader of type *"Unlit Shader Graph,"* which we will call **USB_custom_function**.

The first task we will perform is to bring a Custom Function node to the node area. As mentioned above, we will need a script with the *".hlsl"* extension. To generate it, we can simply create a file with the *".txt"* extension and replace it with *".hlsl,"* or create a script directly from the editor we are working with Unity.

We will call the ".hlsl" file "**CustomLight**" and start adding our function as follows:

**Custom Functions**                                                       ✖
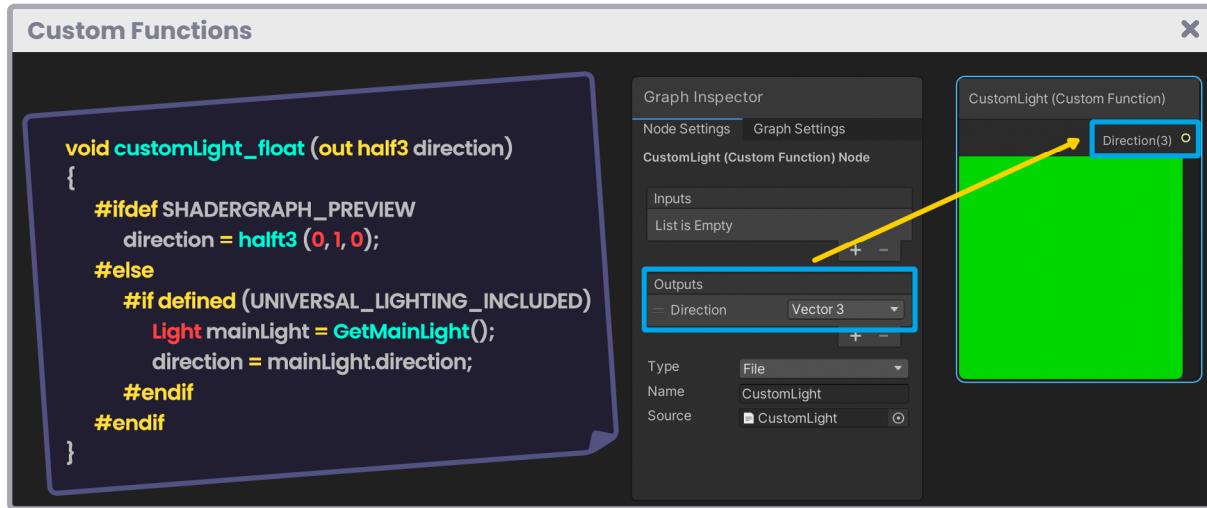
```
// CustomLight.hlsl

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, we can deduce that if the preview in Shader Graph is enabled (SHADERGRAPH_PREVIEW), we will project lighting in ninety degrees on the Y-axis of the space. Otherwise, if **Universal RP** has been defined, then the output "**direction**" will be equal to the direction of the main light, that is, to the directional light we have in the scene.
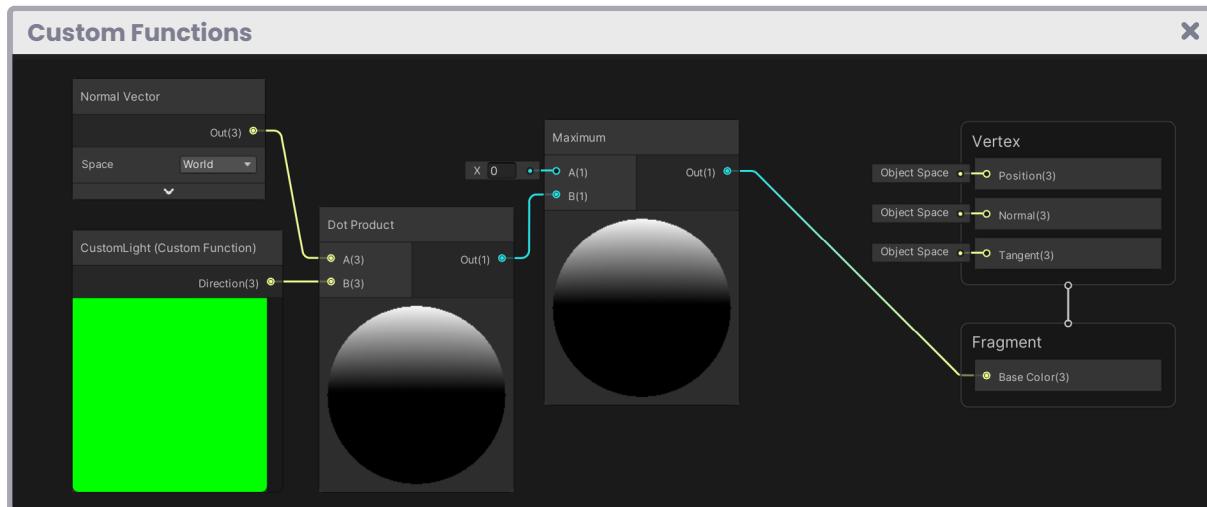
Unlike the previous case, the **CustomLight** function has no inputs; therefore, only a three-dimensional vector will have to be added as output later in the node configuration. On the other hand, it's worth noting that such output is of type "half3." Consequently, the accuracy of the node will be equal to a 16-bit value because, by default, it is configured as "inherit."

Next, we must make sure to drag or select the ".hlsl" file in the "**Source**" box located in the Graph Inspector **Node Settings** window. In the "**Name**" box, we must use the exact name of the function; in other words, *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** owns the light position, we can use our node to replicate the same behavior; in fact, the operation "mainLight.direction" is equal to the enclosed variable _MainLightPosition. We can check this by going to our project's "Lighting.hlsl" file.



*(Fig. 9.0.7d. D = max(0, n · l))*

As shown in Figure 9.0.7d, following the diffusion scheme we saw in section 7.0.3, we can generate the same behavior through nodes in the Shader Graph.

**Chapter III**

# Compute shader, Ray tracing and Sphere tracing.

# Advanced concepts.

In the previous chapters, we concentrated our study on the understanding of three types of programmable shaders: vertex, fragment, and surface shader, of which we reviewed their properties, data types, structure, functions, among other topics.

In this chapter, we will deepen more advanced concepts, and for this, we will investigate those shaders of type .compute. We 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 allows you to 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). With it, you can use functions to process applications that are not necessarily of a graphic nature, e.g., the vertex's calculation per million or multiple object's instances.

A fundamental feature of the Compute shader is its rendering pipeline.  It has its graphical pipeline, which is not part of those mentioned in previous chapters. However, because this shader is part of the Direct3D API, it links the output directly over the logical rendering pipeline.

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 our scene, determining the object's distance according to the camera's position. Three main parts perform this task:

- **Ray generation** (starting point).
- **Ray intersection** (the moment where it hits the object).
- **Shading** (Illumination, shadow, and surface).

Such algorithms increase the graphic quality of our project, generating a very polished and realistic aesthetic; 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 functions and algorithms implementation in "*.raytrace*" type shaders. It is worth stressing 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 and Pascal (GTX 1060+) generation cards. However, its performance is limited compared to those mentioned above.

## 10.0.1 | Compute shader structure.

Up to this point, we have focused our study on the understanding of Unlit and Surface shaders, which have a very similar structure; both are executed within the ShaderLab field, which, as we already know, is a declarative language that allows the communication between the program and Unity. However, another type of shader called Compute exists, which has a similar structure to the above-mentioned but does not include built-in shader variables that facilitate its programming.

We will begin this section creating a Compute shader, for it:

1. We will go to our project folder in Unity.
2. Then, we right-click.
3. And select Create / Shaders / Compute shader.
4. We will call it USB_simple_color_CS.

Later we will work with this shader to understand the basic structure in color, UV coordinates, and texture implementation. Therefore, it will not be beautiful at a functional level. Still, it will serve to illustrate the syntax behind the program.

Once opened, we 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, we can see a basic color structure that Unity adds by default in the program. In it, we can find the following components:

- The **kernel** of our CSMain function.
- A 2D texture for writing and reading, called **Result**.
- The number of threads we will use 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 language. Therefore, to understand their basic setup, we will make an analogy using the CSMain kernel.

The vertex shader is configured as a stage when we determine the pragma associated with its function (As mentioned in chapter 1, section 3.3.2); this means, e.g., the vert function must be declared as "vertex" in the pragma so that the GPU can recognize its nature within the rendering pipeline.

**Compute shader structure** ✕

```
// declare the vert function as vertex shader stage
#pragma vertex vert

// initialize the vert function
v2f vert(appdata v) { … }
```

We can find the same behavior in the fragment shader stage. If we want the default function called "frag" to compile as a fragment shader, we 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 we want to send the "CSMain" function to the "compute units" (physical unit where it performs the computation), then it must be defined as a "**kernel**" in the pragma.

**Compute shader structure** ✕

```
// declare the CSMain function as kernel
#pragma Kernel CSMain

// initialize the CSMain function
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID) { … }
```

The smallest unit that a Compute shader can process corresponds to an independent thread, and the attribute [numthreads(x, y, z)] is directly related to this

The threads perform the computation of the operation that we want to carry out, e.g., in the case of a texture, they are in charge of processing each texel that the image has.

**300**

By default, our program has a group of 64 threads. How can we determine this? Basically, by multiplying the values in X, Y and Z 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 warps. 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 and Y precisely to ensure that the program runs on both; NVIDIA and ATI cards. Later in this chapter, we will review this and other attributes in detail, since some semantics are associated with the threads we will operate with. For now, we will continue defining the internal structure of our program.

The RWTexture2D variable called "Result" refers to a 2D RGBA texture with reading/write capability (RW). This feature allows data to be sent from the CPU to the GPU, processed in parallel, and then returned.

If we want to add a variable that only has the *"write"* capability, we would have to add it without the RW prefix, e.g., Texture2D. Now, how could we determine the need for a variable in our program? For this, we will have to implement some functions in the Compute shader.
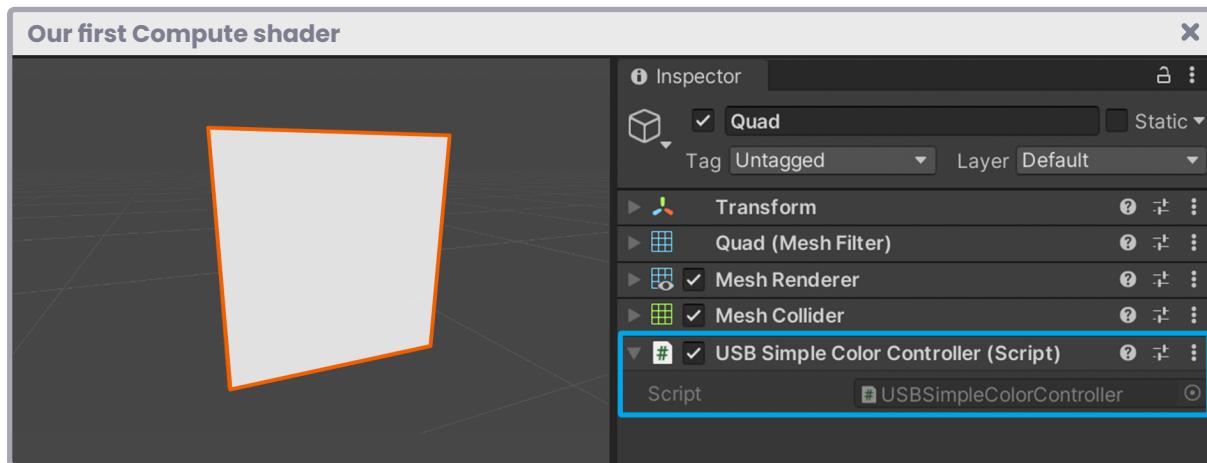
A vertex/fragment shader requires the declarative language ShaderLab for communication between Unity and the CGPROGRAM or HLSLPROGRAM. Analogously, a ".compute" shader requires a C# script for the same function. Every time we work with Compute shaders, we will have to create and associate a C# script in our scene. This last one declares the global variables and buffers that we will connect later with the HLSL program.

A function that we 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 we will discuss later in this book.

## 10.0.2 | Our first Compute shader.

Continuing with USB_simple_color_CS, we will need a 3D object for color, texture, and UV coordinate. We will add a Quad to our scene for this exercise and ensure it is centered, with its position and rotation set to "zero".

In our project, we will create a C# script called **USBSimpleColorController**.cs. We will use it as a controller for the Compute shader. Since we will write a texture on the object's material, we will have to assign the script directly to the 3D object.



(Fig. 10.0.2a.  The **USBSimpleColorController** script has been
assigned to the Quad we have in the scene).

Once the program is opened, we will obtain 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 next page.*

```
        }
        // Update is called once per frame
        void Update()
        {


        }
    }
```

We can see that it corresponds to the default structure of a C# script. It includes start and updates functions frame by frame to facilitate understanding.

We will start by adding a global public variable to our program to connect the Compute shader. We will call it **m_shader**.

**Our first Compute shader** ✕

```
    public class USBSimpleColorController : MonoBehaviour
    {
        public ComputeShader m_shader;

        …

    }
```

Assuming that we will write a **texture** on the Quad's material, we 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, we declare a public variable of type **RenderTexture** for the texture and an integer value for its dimensions. We will use 256 for both width and height.
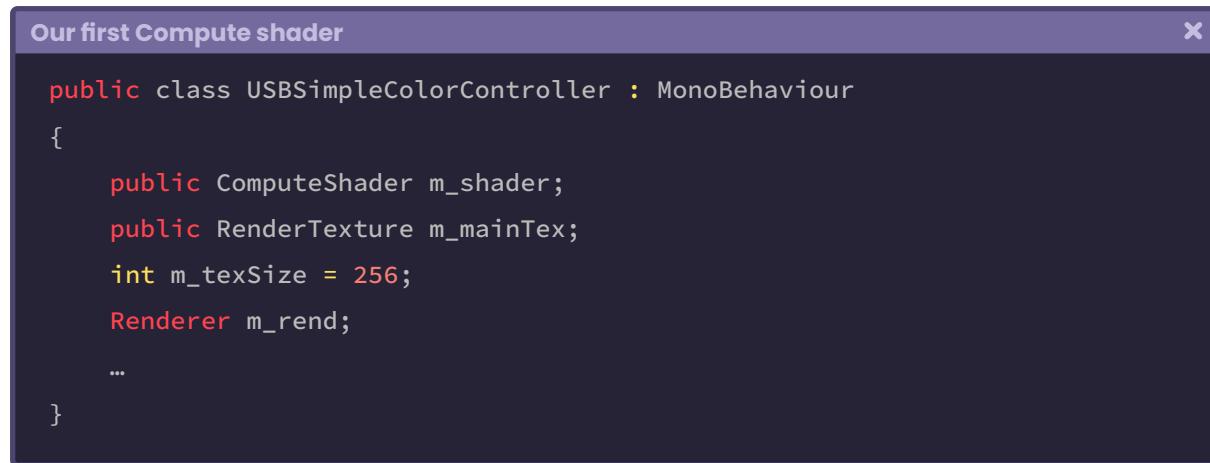
**Our first Compute shader** ✕

```
    public class USBSimpleColorController : MonoBehaviour
    {
        public ComputeShader m_shader;
        public RenderTexture m_mainTex;
        int m_texSize = 256;

        …

    }
```
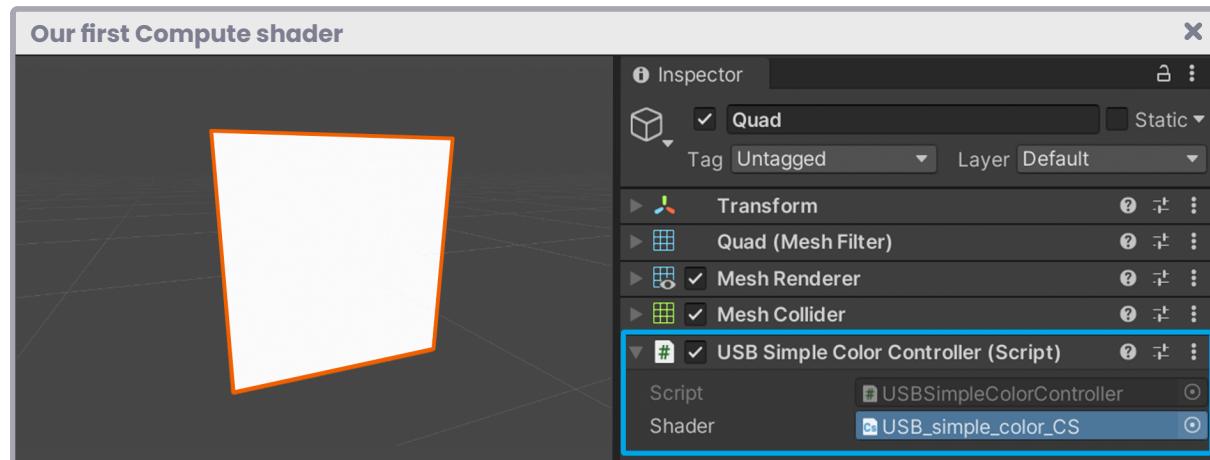
In the previous example, we declared a texture named **m_mainTex** and defined its dimension (m_texSize). Its nature is associated with the _MainTex property that we frequently use to determine properties for shader type ".**shader**". It suggests writing the variable m_mainTex on **_MainTex** later for color display. We only need to define a variable that allows us to write the Quad's material's texture.

**Our first Compute shader** ✕

```
public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;

    …

}
```

We will use the variable m_rend later to store the "Renderer" component of the material associated with the object. In contrast, the variable m_mainTex will be used to write the colors that we will generate in the Compute shader.

Before continuing with the explanation, we will save the code we 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 in the m_shader variable from the Unity Inspector).*

We return to our script and initialize the texture in the Start method. For this, we will use the variable m_texSize for both the texture's height and width.

**Our 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 constructor of the RenderTexture class has up to seven arguments; however, we only need four of them for the texture to work correctly. The first two arguments correspond to the texture's width and height, the depth buffer, and finally, the texture configuration (32-bit RGBA). Now we simply require enabling the random writing options and create the texture as such.

**Our first Compute shader** ✖

```
public class USBSimpleColorController : MonoBehaviour
{
    public ComputeShader m_shader;
    public RenderTexture m_mainTex;
    int m_texSize = 256;
    Renderer m_rend;
```

*Continued on next page.*

```
void Start ()
{
    m_mainTex= new RenderTexture(m_texSize , m_texSize, 0,
    RenderTextureFormat.ARGB32);
    // enable random writing
    m_mainTex.enableRandomWrite = true;
    // let's create the texture
    m_mainTex.Create();
}
…
}
```
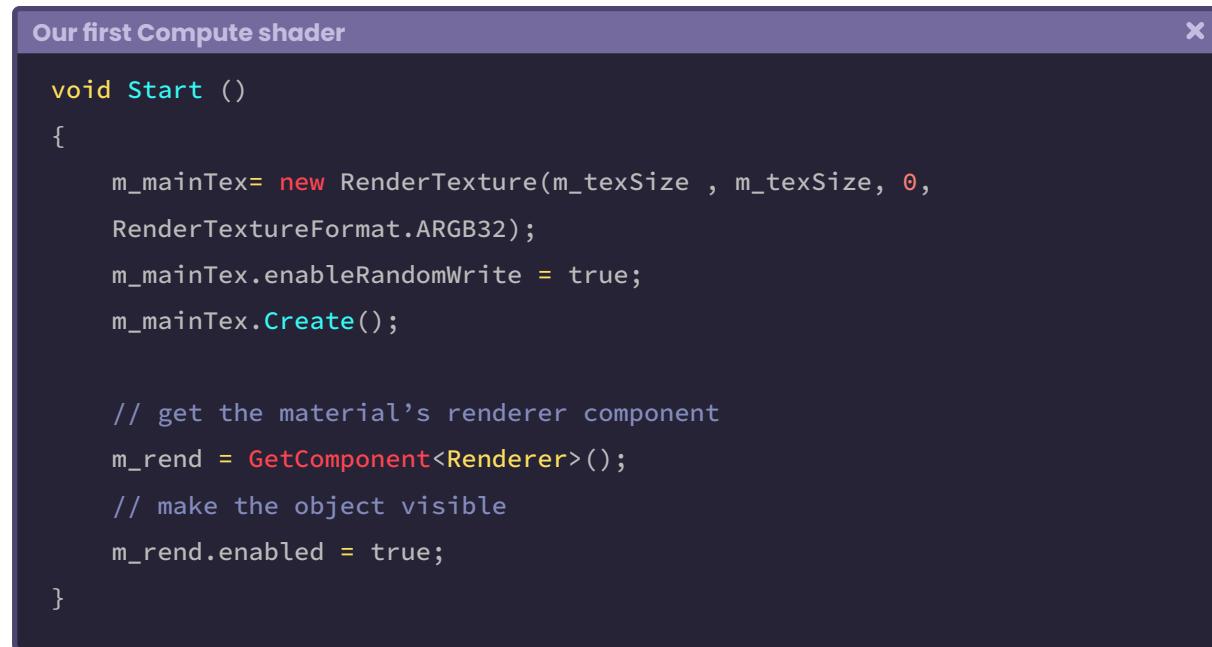
Since thread groups cannot synchronize with each other, we cannot determine which texel will be written to the texture first. For that reason, we must use the enableRandomWrite function before creating it. Finally, the "Create" function creates the texture; in fact, we can find the following text according to the official Unity documentation.

> *The RenderTexture constructor 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.*

**Our first Compute shader** ✕



(Fig. 10.0.2c Compute shaders can write texels arbitrarily on a texture).

The following process will allow the communication between the C# script and the Compute shader. We will start by saving the **Renderer** component (specific to the Quad material) in the "*rend*" variable that we created previously.
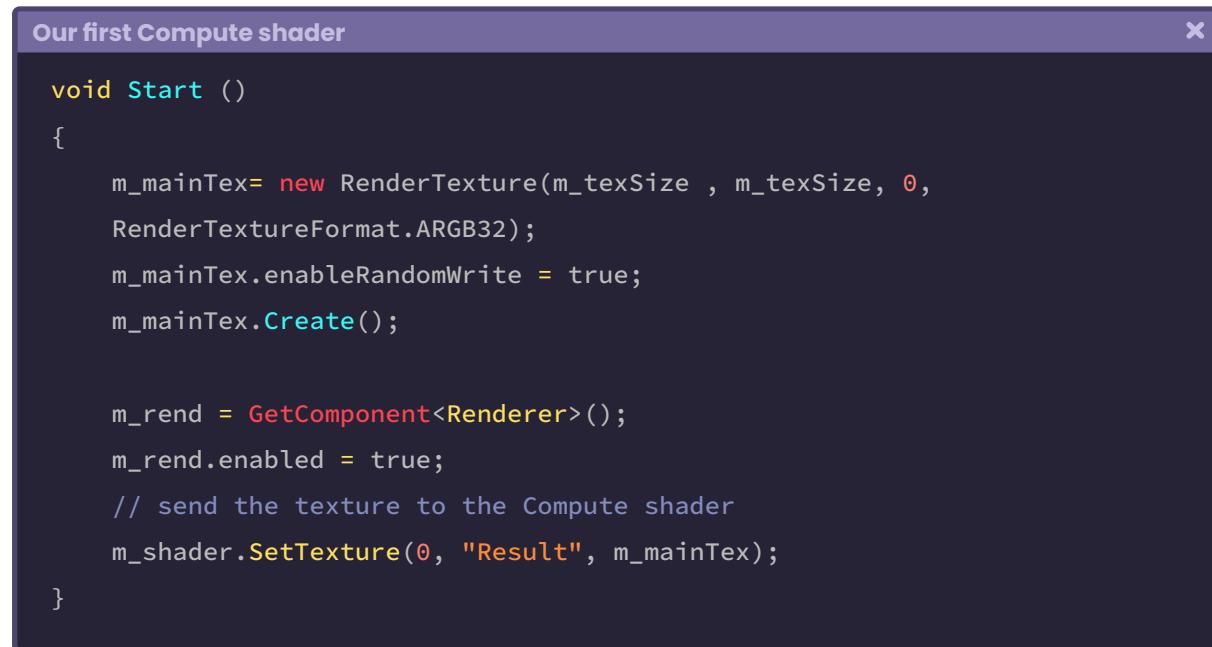
**Our first Compute shader**                                                                          ✕

```
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;
}
```

We have the texture created up to this point, but it does not have any specific color. Therefore, what we 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 is using the Quad so that it is visible from the scene. To do this, we can use the "**ComputeShader.SetTexture**" function.

**Our 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 (0) corresponds to the first kernel index we use in the Compute shader.

**Our 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 kernel called **CSMain**, which has been added by default. This kernel occupies index "zero" since it is the only one we have in the program. A Compute shader can have multiple kernels, and each one has an automatically assigned id.

**Our first Compute shader**                                           ✕

```
#pragma kernel CSMain           // id 0
#pragma kernel CSFunction01     // id 1
#pragma kernel CSFunction02     // id 2
```

The second argument in the "**SetTexture**" 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 (float4) with reading/write capability.

**Our first Compute shader**                                           ✕

```
RWTexture2D <float4> Result;
```

Finally, the third argument in the function corresponds to the texture that we will write on the buffer variable; in our case, it is called **m_mainTex**. We will process this variable in the

Compute shader within the CSMain function, we can corroborate this by the operation being performed into the method.

**Our 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, we will not go into details about the operation in the CSMain method. For now, we will continue implementing the texture on the material that currently has the Quad in our scene. To do this, we return to the **USBSimpleColorController** script and pass the texture **m_mainTex** on the **_MainTex** property using the **SetTexture** function of the material.

**v** ✕

```
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 shader in Unity has the _MainTex property, so we can assume that the Quad material has it as well.

Up to this point, the process is almost ready; we only need to generate the groups of threads that will process each texel of the texture we are creating. For this, we must call the **Dispatch** function.

**Our first Compute shader** ✖
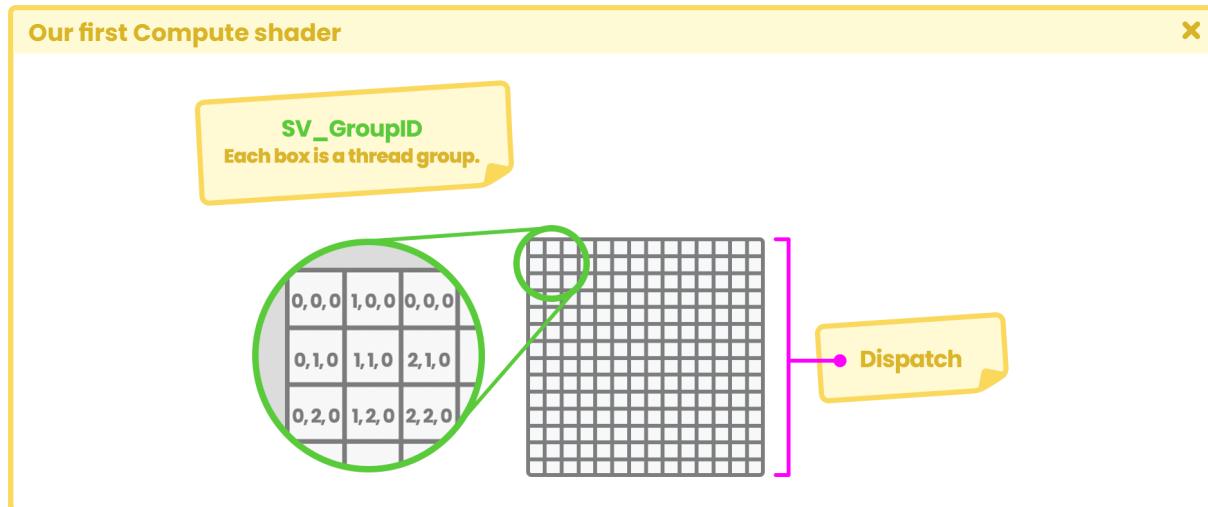
```
void Start ()
{
    m_mainTex= new RenderTexture(m_texSize , m_texSize, 0,
    RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    m_rend = GetComponent<Renderer>();
    m_rend.enabled = true;

    m_shader.SetTexture(0, "Result", m_mainTex);
    m_rend.material.SetTexture("_MainTex", m_mainTex);

    // generate the threads 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; again, since we are only using CSMain, we will have to place the value "zero" in it. The following three values correspond to the grid (groups of threads) that we will generate to process the texture texels. The first value corresponds to the number of columns that the grid will have, then the rows, and finally the number of dimensions. As we already know, **m_texSize** equals 256; therefore, if we divide that value into 8, we 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 A thread group on each independent compute unit.
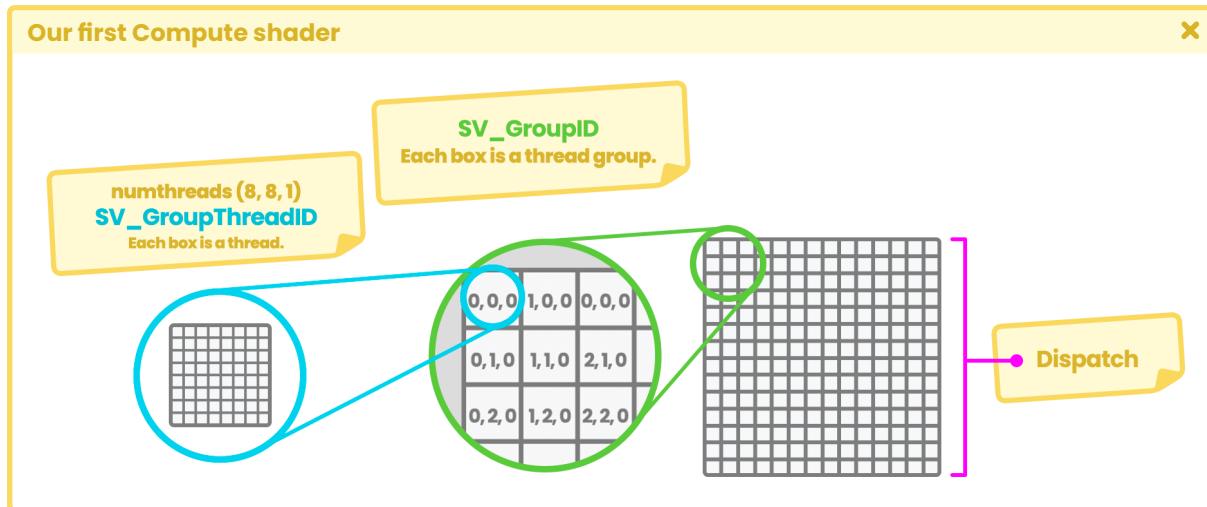
The operation of thread synchronization can occur only for those within the same group, generating more efficient parallel programming methods. The GPU cannot synchronize different groups of threads; we have no control over the order in which they will be processed. For this reason, it can send such groups 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? Its value is determined by the "numthreads" attribute at the top of the CSMain function.

**Our first Compute shader** ✕

```
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID) { … }
```

As we already know, to calculate the number of threads within a group, we simply multiply the number of columns, by the number of rows, by the dimensions (8 * 8 * 1). For our configuration, we obtain the number of 64 threads for each group.
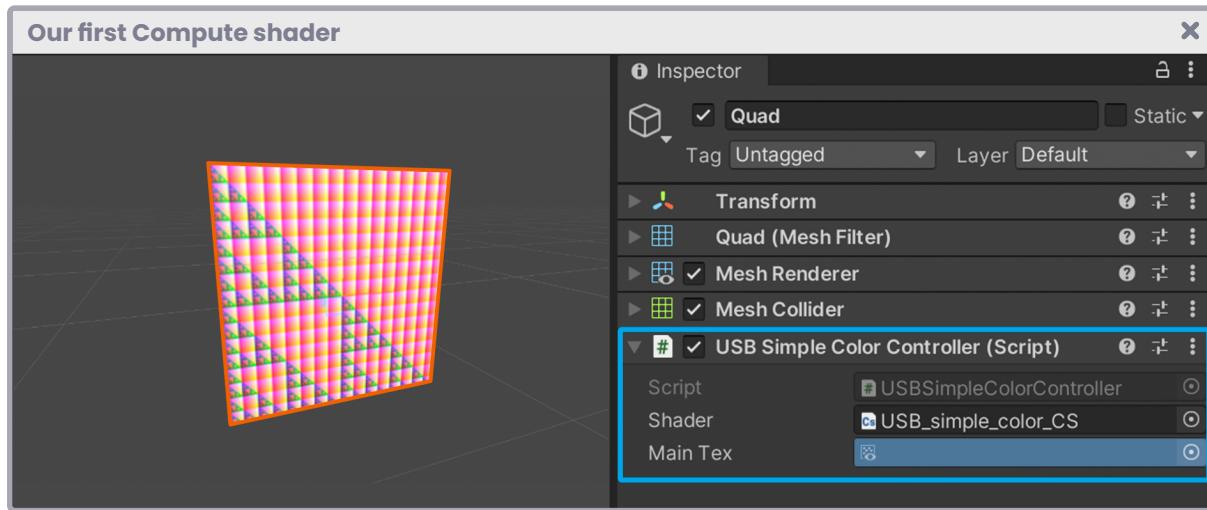
(Fig. 10.0.2e. The semantics **SV_GroupID** corresponds to the index of a group to be executed in the Compute shader, **numthreads** refers to the total number of threads we will have for each group, and **SV_GroupThreadID** refers to the identifier (index) of each thread separately).

It is essential to delve into this process to understand the semantics 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 we are using, plus the index of each thread.

SV_DispatchThreadID = ([(SV_GroupID) * (numthreads)] + (SV_GroupThreadID))

Going back to our **USBSimpleColorController**.cs program, if we save, go back to Unity and press the "play" button, we can see that the program has generated a texture and is dynamically assigned to the Quad.

**Our first Compute shader**



*(Fig. 10.0.2f. The texture corresponds to the graphical representation of the Sierpinski fractal).*

The texture we see in the figure above is being generated within the CSMain function, and its creation process is quite simple.

**Our 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, we should pay attention to the arguments of the CSMain function.
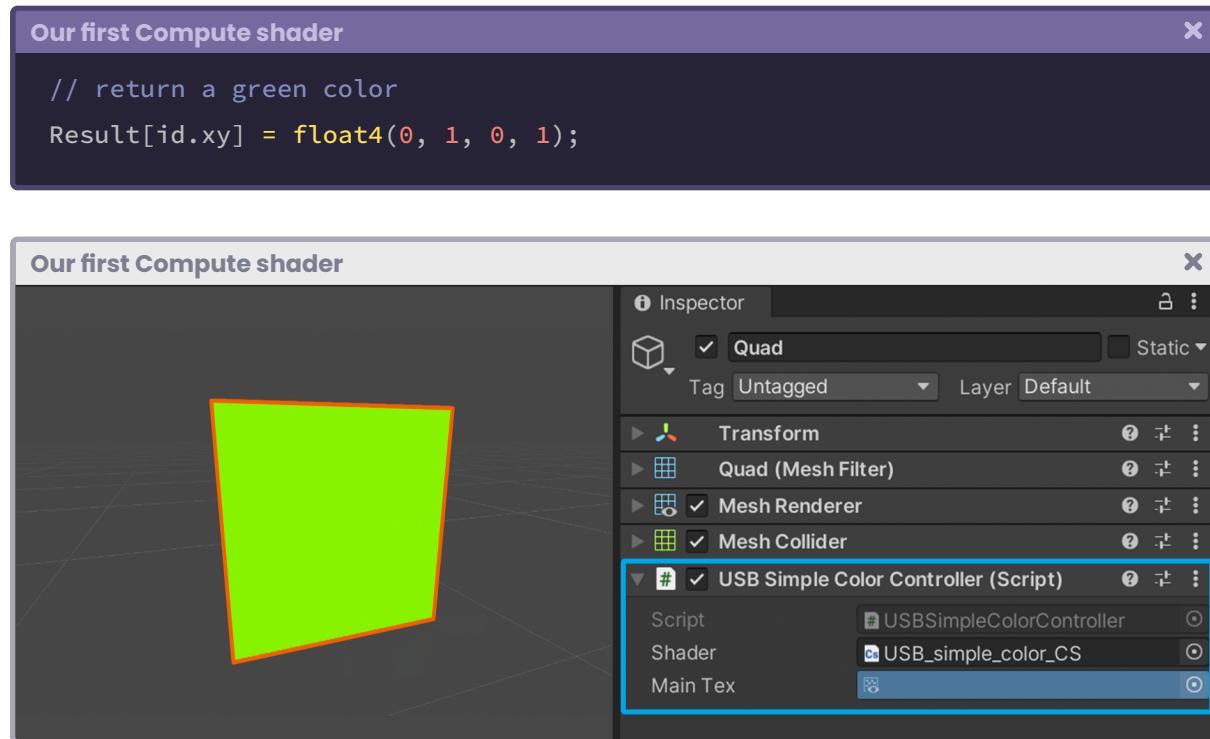
The semantics SV_DispatchThreadID represents the indices of those combined threads and thread groups executed in the Compute shader; this means that the identification of each thread is being stored in the id variable of type uint3 (unsigned integer).

Unlike an **int** variable, **uint** variables have only positive numbers, starting at "zero". It makes sense since the indexes of each thread group begin at [0, 0, 0] hence the **uint3** data type.

**Our 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, we can return a solid color, e.g., green.

**Our first Compute shader** ✕

```
// return a green color
Result[id.xy] = float4(0, 1, 0, 1);
```



**Our first Compute shader** ✕
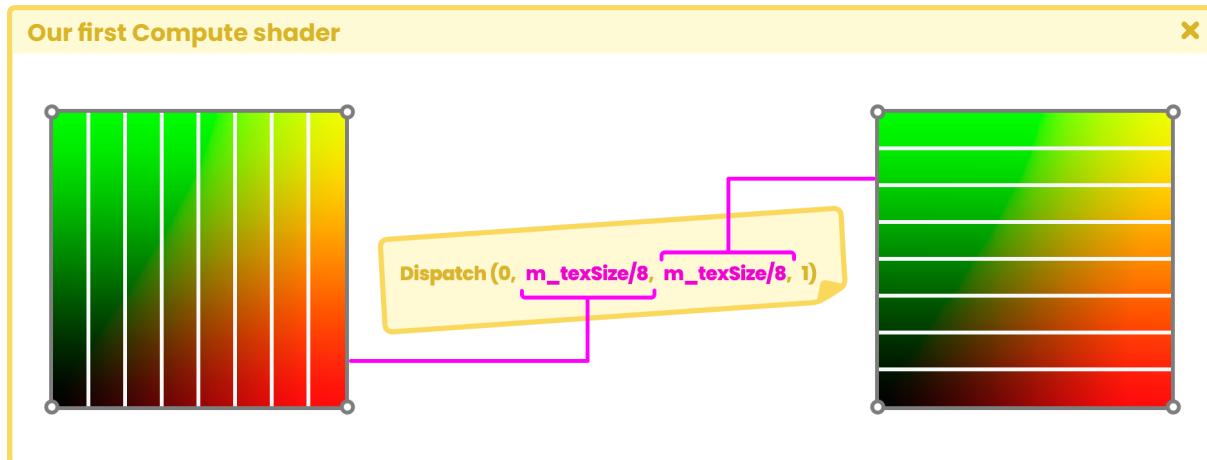
*(Fig. 10.0.2g)*

One factor to consider is the number of threads we will use in our Compute Shader, since it is directly related to the final result we will obtain.
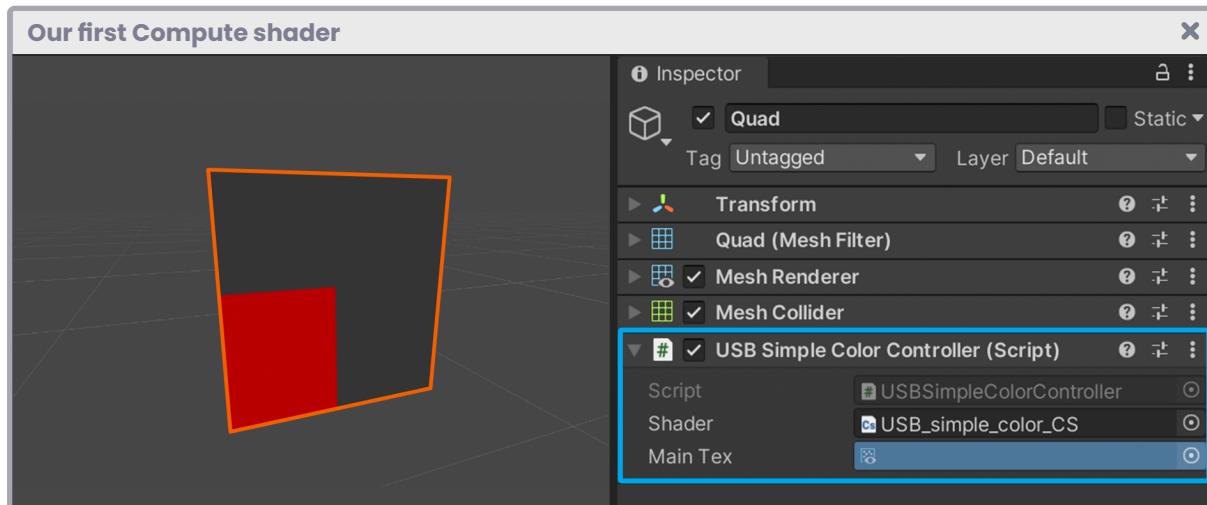
Earlier in the Dispatch function, we configured the width and height of the texture divided into eight to generate a grid of 32 x 32 x 1 groups of threads. Why do we use the number eight as a divisor in operation? It is related to the configured number of threads in the numthreads component.

Let's perform the mathematical operation to understand the concept: The texture we create has a width and height equal to 256 if we divide it into eight, results in 32. Its graphic representation would equal eight rows or columns of thirty-two texels each, one above the texture.

*(Fig. 10.0.2h)*

For the result to be equal to 265 again, we must multiply by eight; precisely, this number is set as the total number of threads in both X and Y in the numthreads attribute. If we change the number of threads to 4 [numthreads(4, 4, 1)], we will notice that only ¼ of the texture is rendered in the Quad.



*(Fig. 10.0.2i. Color (1, 0, 0, 1))*
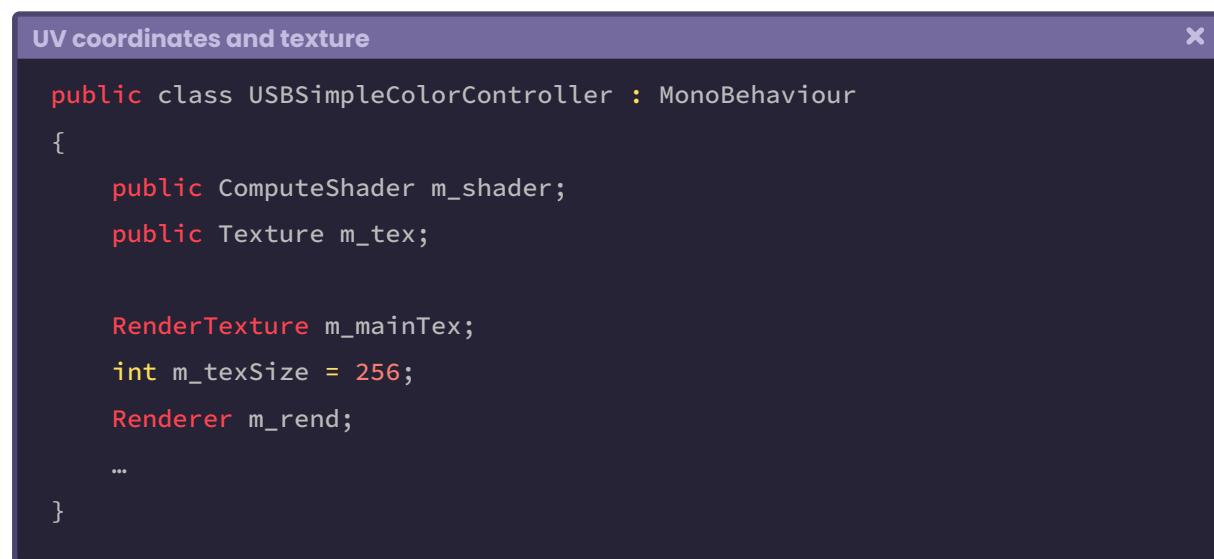
This factor occurs when multiplying 32 by 4, resulting in 128, precisely ¼ of the texture we are generating.

## 10.0.3 | UV coordinates and texture.

In the previous section, we defined a texture and its dimensions through the variables *m_mainTex* and *m_texSize*; however, the final result corresponds to the graphical representation of the Sierpinski triangle.

Considering the script we have been writing up to this point, in this section, we will assign an imported texture to the effect, and for this, we will have to define UV coordinates.

We will start by declaring a new public texture of type "**Texture**" called "m_tex" in our C# script.
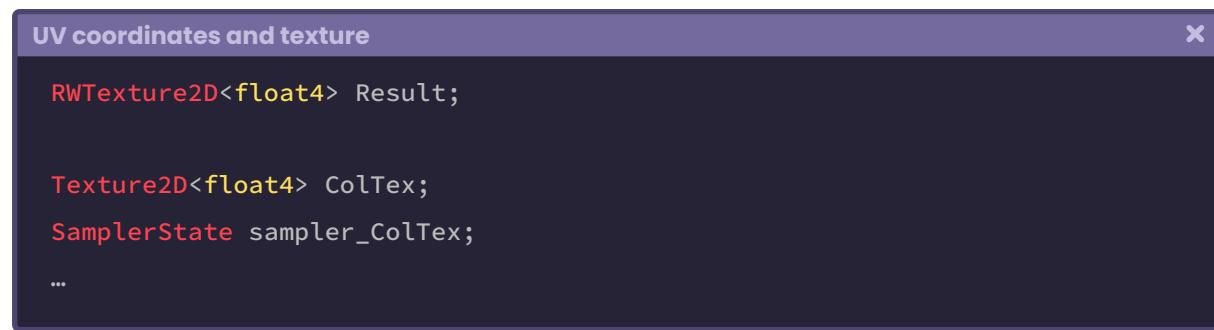
```
UV coordinates and texture                                    ✕

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, we will have to declare a type **Texture2D** and another type **SamplerState** in the Compute Shader.

```
UV coordinates and texture                                    ✕

RWTexture2D<float4> Result;


Texture2D<float4> ColTex;

SamplerState sampler_ColTex;

…
```

As shown in the example above, the declaration for a texture has the same analogy as those seen in previous sections. In the same way, we will need UV coordinates to position

the texture on the Quad we are using. We can use the **GetDimensions** function within the **CSMain** Kernel to do this.

**UV coordinates and texture**                                                      ✕

```
RWTexture2D<float4> Result;


Texture2D<float4> ColTex;
SamplerState sampler_ColTex;


[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width;
    uint height;
    Result.GetDimensions(width, height);

    …

}
```

The **GetDimensions** function is of type "void," which means we are saving the dimensions of the Result variable in **width** and **height**. The dimensions correspond to the value we assigned to the variable "m_texSize" from **USBSimpleColorController**.

**Compute shader structure**                                                        ✕

```
void GetDimensions(out uint width, out uint height);
```

We can then 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; for that reason, 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 we must evaluate according to the "**texture wrap mode**" that we assign to the Quad.

The above operation works perfectly for those textures declared in "**clamp**" mode. If the texture has been configured as "**repeat**," we will need to add 0.5f to the **id** variable; otherwise, it will reflect 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, we can then use the SampleLevel 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."

**Compute shader structure** ✕
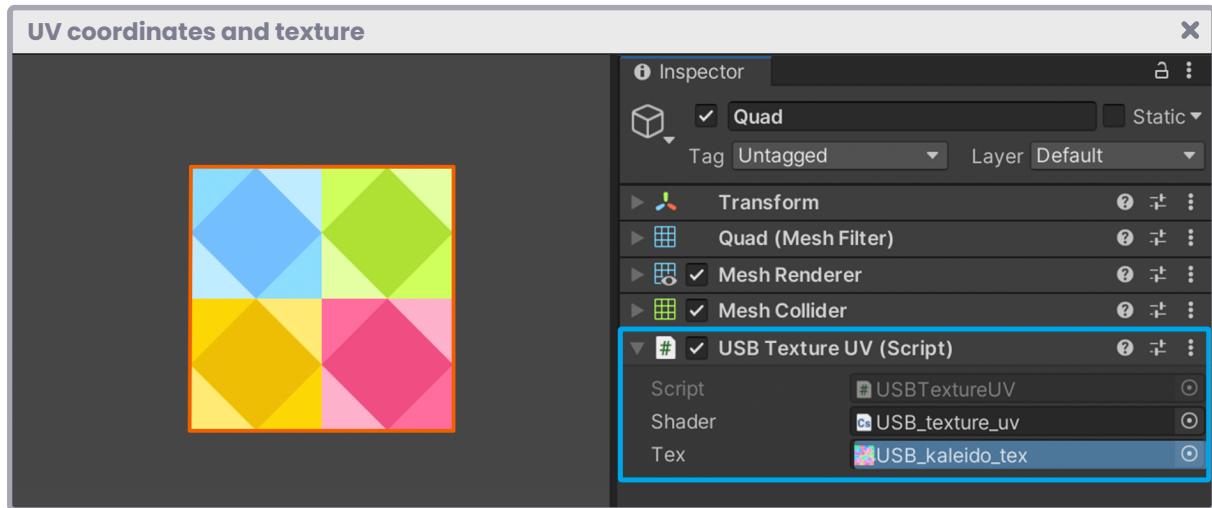
```
Object.SampleLevel(in SamplerState s, in float2 uv, in int LOD);
```

To conclude, we must return to the **USBSimpleColorController** script and send the texture to the Compute Shader through the **SetTexture** function, in the same way we 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, we will see the texture assigned in the "*m_tex*" field projected on our 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 our benefit, there are two associated data types that we can use in our program to speed up the reading and writing of values to the memory buffer: **ComputeBuffer** and **StructuredBuffer**.

As its name mentions, **ComputeBuffer** corresponds to a buffer, which we can create and fill with a list of values from our C# script.

A StructuredBuffer is essentially the same, except that we 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;
```

We will create a new Compute Shader in our project to understand its implementation, which we will call **USB_compute_buffer**. In the same way, we will create a new C# script which we will call **USBComputeBuffer**.

Previously, in section 4.1.6, we created a simple method called "circle," which we used to reproduce a circle in a Quad. In this section, we will perform the same exercise with the difference that we will use the scripts previously created for this purpose.

Having studied part of the Compute Shader integration in Unity, we might deduce that **USBComputeBuffer** will handle the data configuration through the **SetFloat** and **SetTexture** functions.

In the same way, we will configure data that we will send later to the predefined list of values in the Compute Shader through the **ComputeBuffer.SetBuffer** function.

We will start by declaring the public variables associated with the effect in section 4.1.6.

**Buffers** ✕

```csharp
using System.Collections;
using System.Collections.Generics;
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 we pay attention to the example above, we will notice that the same properties (*m_radius*, *m_center*, and *m_smooth*) have been defined that we used previously for the generation of a circle. In addition, a color property has been created for it.

Such variables can be sent individually to the Compute Shader using the **ComputeShader. SetFloat** function or create a buffer containing the complete list of values we want to assign.

For the exercise, we declare a structure and a buffer associated with the list of values we 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;

    ...

}
```

Inside the "struct Circle," we have declared the variables we will use later in the ComputeShader; through "m_circle," we will access each instance.

Given the exercise nature, we can deduce that the values of the global variables will be assigned to those defined in the struct. At this point, the **ComputeBuffer** becomes relevant since, once the program has filled the list with values, we must copy the data to the buffer and finally pass it to the shader.

We will start by creating the texture before performing such a process. To do so, we declare a new method which we will call "**CreateShaderTex**." The method will contain the algorithm described in section 10.0.2 for the texture definition.

**Buffers** ✖

```
void Start()
{
    CreateShaderTex();
}


void CreateShaderTex()
{
    // first, we create the texture
    m_mainTex = new RenderTexture(m_texSize, m_texSize, 0,
    RenderTextureFormat.ARGB32);
    m_mainTex.enableRandomWrite = true;
    m_mainTex.Create();

    // then we access to the mesh renderer
    m_rend = GetComponent<Renderer>();
    m_rend.enabled = true;
}
```

Next, we declare a new function, which we will use in the "Update" method for study purposes only.

**Buffers** ✖

```
void Update()
{
    SetShaderTex();
}


void SetShaderTex()
{
    // write the code here …
}
```

Before continuing, we will go to **USB_compute_buffer**, since we must configure its structure before bringing the data from **USBComputeBuffer**. We will add the "circle" function and then define its values in the CSMain Kernel.

```
Buffers                                                                    ✕

  #pragma kernel CSMain


  RWTexture2D<float4> Result;


  // declare de method
  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, we defined the "CircleShape" method, which is equal to the "circle" function detailed in section 4.1.6. Within the CSMain Kernel, such a function has been initialized with its values set to "zero." Consequently, the output corresponds to a black color by default.

Noteworthy that the number of threads for the operation is equal to 128 in "x," this is mainly due to two factors:

1. The texture size of the variable "m_texSize" is equal to 128.
2. We only need one dimension to traverse the previously defined "m_circle" list.

Next, we will 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)
{
```

*Continued on next page.*

```
    uint width;
    uint height;
    Result.GetDimensions(width, height);
    float2 uv = float2((id,xy + 0.5) / float2(width, height));


    // we access to the array values
    float center = CircleBuffer[id.x].center;
    float radius = CircleBuffer[id.x].radius;
    float smooth = CircleBuffer[id.x].smooth;


    // we use the value as arguments in the function
    float c = CircleShape(uv, center, radius, smooth);


    Result[id.xy] = float4(c, c, c, 1);
}
```
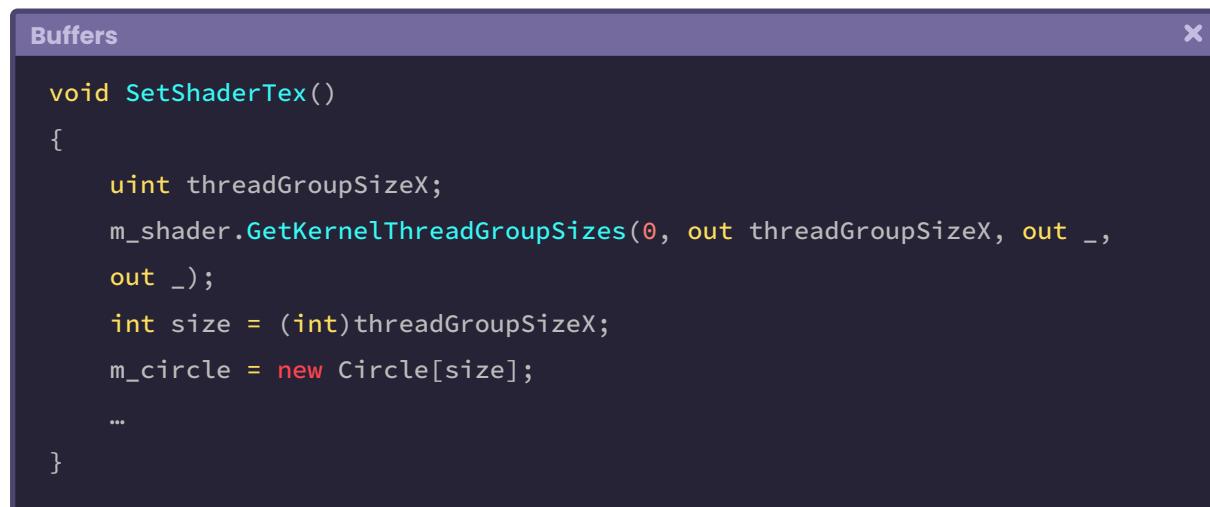
As in **USBComputeBuffer,** it has specified a list of scalars inside a struct called Circle. These variables match quantity and data type with those declared in the C# script.

Subsequently, we have declared a **StructuredBuffer** called *CircleBuffer*. The buffer will handle storing the values we send from *USBComputeBuffer*.

It would only be necessary to complete the **SetShaderTex** function's operations and send them to the Compute Shader. To do this, we 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];

    …

 }
```

The example has started the exercise by declaring a variable of type "unsigned integer" called threadGroupSizeX. It is due to the Void type function called GetKernelThreadGroupSizes, which takes the group of threads that have been configured in the Kernel, i.e., the variable mentioned above will receive the value 128.

**Buffers**                                                                    ✕

```
// compute shader
[numthreads(128, 1, 1)]


// C#
GetKernelThreadGroupSizes(kernel, 128, 1, 1);
```
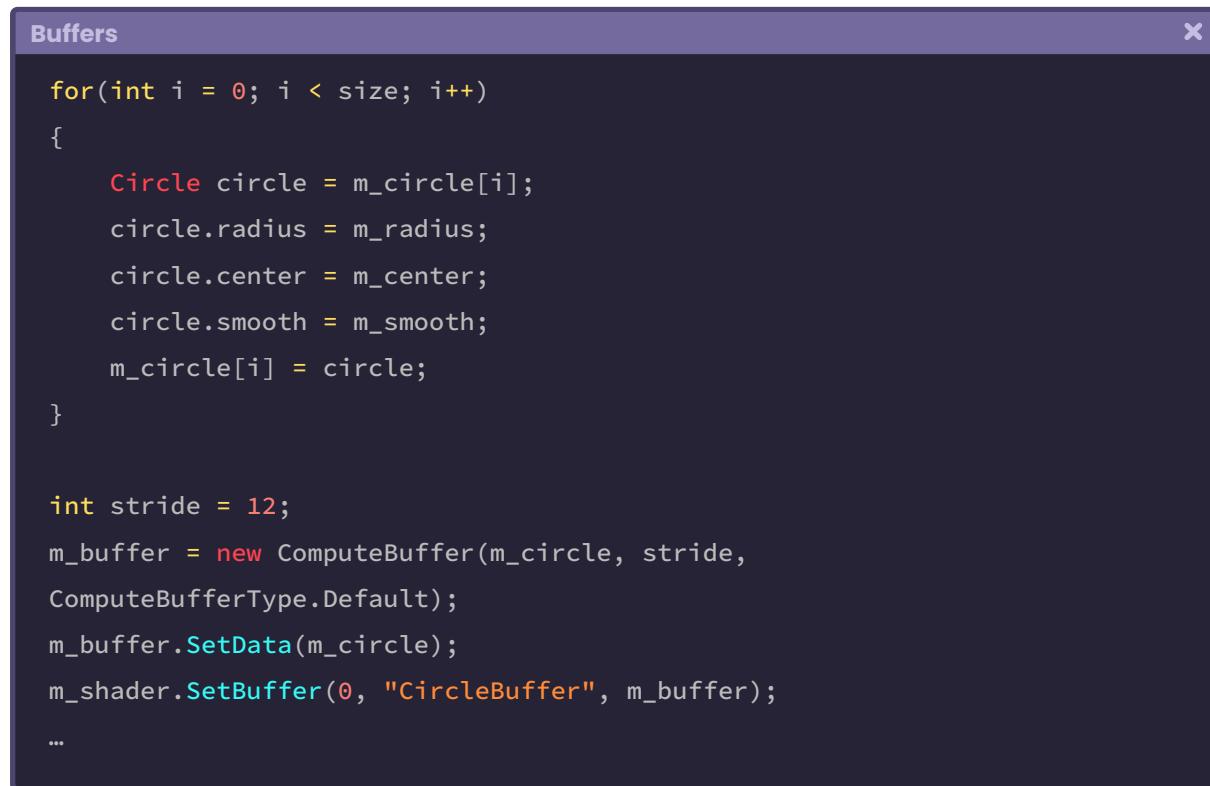
Finally, such a value has been added to the list "m_circle," which we will use as "data" for the buffer. Next, we will assign the public variables to those declared within the list. We can simply initialize a loop and pass the values to each variable separately to do this.

**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 we store the information in the list, we can declare a new ComputeBuffer, configure the information, and then send the data to the Compute Shader.

**Buffers** ✕
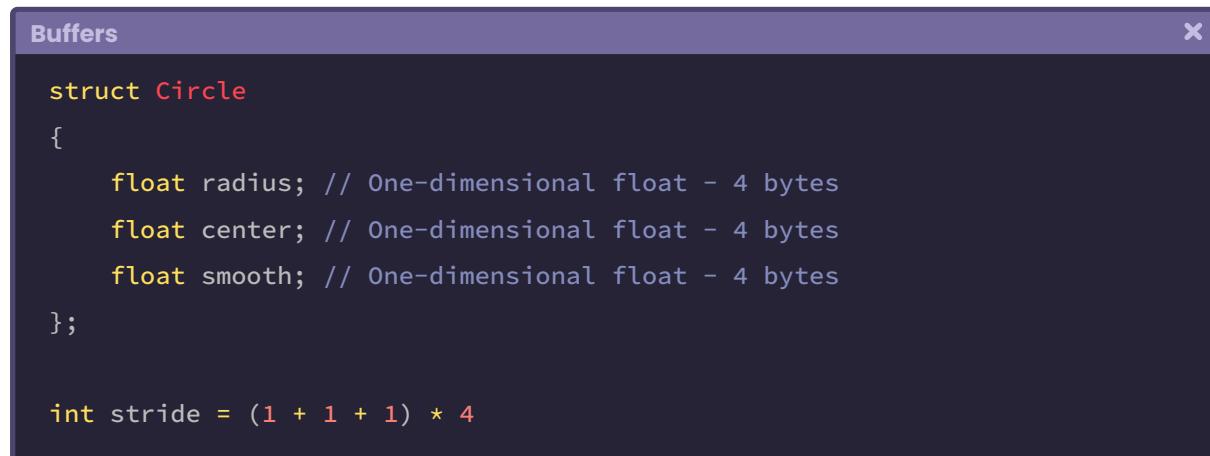
```
for(int i = 0; i < size; i++)
{
    Circle circle = m_circle[i];
    circle.radius = m_radius;
    circle.center = m_center;
    circle.smooth = m_smooth;
    m_circle[i] = circle;
}


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 constructor of the ComputeBuffer contains three arguments: the number of elements in the buffer, the size of the elements, and the type of buffer we are creating.

As we can see in the example above, we use the data stored in the variable "m_circle" as the first argument. The second one (stride) is equal to the number of dimensions of those scalars we are passing times the floating variable's bytes.
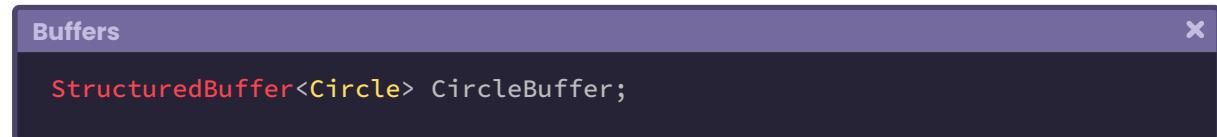
**Buffers** ✕

```
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 we are using in the third argument corresponds to the *StructuredBuffer* of type *Circle* that we declared earlier in the Compute Shader.

**Buffers** ✕

```
StructuredBuffer<Circle> CircleBuffer;
```

After passing the data to the buffer through the **SetData** function, we send the information to the StructuredBuffer "CircleBuffer" through the **SetBuffer** function.

Finally, we 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, we can call the "Release or Dispose" function, which manually releases 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 uses geometric distance.**

What does the above statement refer to? Before going into details, we 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. 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, we will have to find the intersection between the ray and a surface defined by an "implicit distance" equation.

**Sphere tracing** ✕
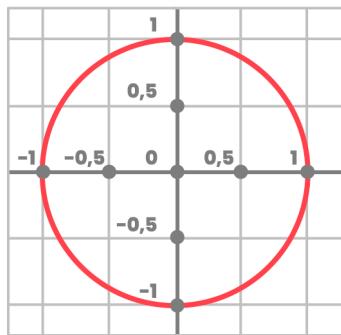


*(Fig. 11.0.1a. Sphere Casting three red spheres)*

In differential calculus, we can find algebraic and transcendental equations, explicit and implicit, e.g., the following implicit equation allows us to generate a sphere in three dimensions:

```
// all the variables are part of the equation
x² + y² + z² – 1 = 0
```

$$x^2 + y^2 + z^2 - 1 = 0$$

*(Fig. 11.0.1b. With "z" equal to 0.0f)*

Which is the same as saying,

$$\| x \| - 1 = 0$$

Therefore,

**Sphere tracing** ✕
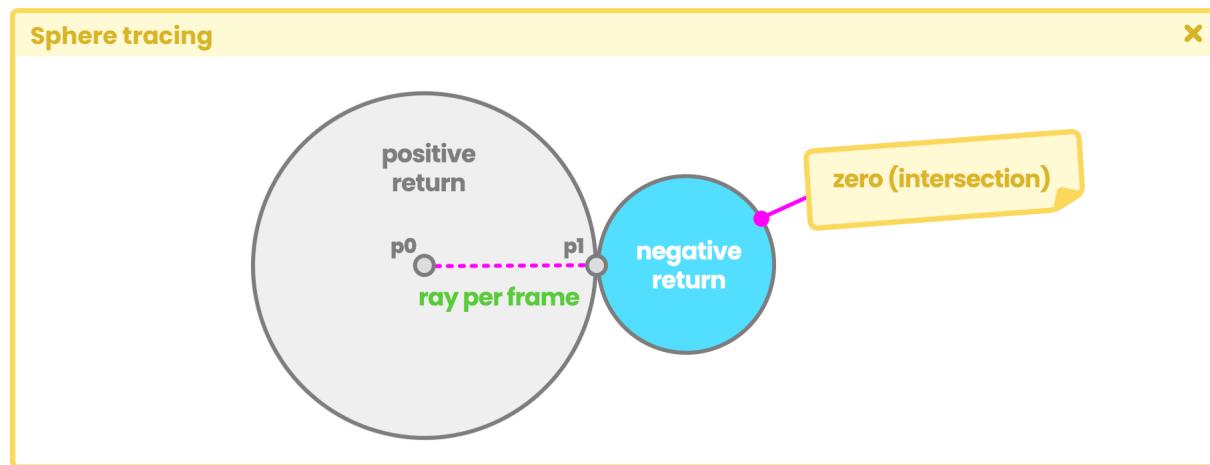
```
float sphereSDF(float3 p, float radius)
{
    float sphere = length(p) - radius;
    return sphere;
}
```

To define an implicit surface, we need to consider the position of a point in space.

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 we 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 equal to the surface's intersection we are hitting.

We have to consider the surface's shape we will generate to determine the ray's endpoint. In this context, those functions of type SDF (Signed Distance 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. 11.0.1c 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 functions in our shader for the correct operation of this technique. For this, we will have to consider:

- An SDF function to determine the type of surface.
- Another function to calculate the sphere casting.

In Unity, we will create a new shader of type "*Unlit Shader,*" which we will call **USB_SDF_fruit**. To understand the concept, we will develop an effect that we will call "sliced-fruit." We will divide a sphere into two parts to show its inside.

Noteworthy that we will apply this effect on 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.

We will start by declaring a new property in our shader, which will define the effect's border or division.
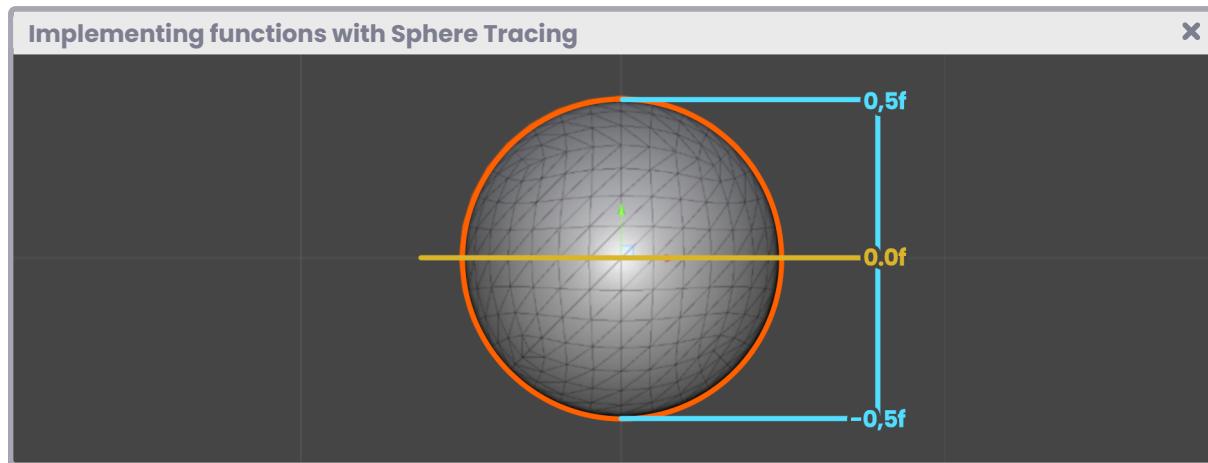
**Implementing functions with Sphere Tracing** ✖

```
Shader "USB/USB_SDF_fruit"
{
    Properties
    {
        _Maintex ("Texture", 2D) = "white" {}
        _Edge ("Edge", Range(-0.5, 0.5)) = 0.0
    }
    SubShader
    {
        Pass
        {
            …
            float _Edge;
            …
        }
    }
}
```

If we pay attention to the _Edge property defined previously, we will notice that its range corresponds to a value between -0.5f and 0.5f. It is due to the volume or scale of the sphere we are working with.

**Implementing functions with Sphere Tracing** ✖



(Fig. 11.0.2a)

We will use two primary operations to divide the sphere: discard the pixels on the **_Edge** and project a plane in the sphere's center. Discarding the pixels will optimize the effect and visualize the plane that we will generate later.

We continue by declaring a function of type SDF 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 the **_Edge** is subtracting from Ray's y-position, our three-dimensional plane will graphically move up or down in space according to the property's value. Its implementation can be seen later in this section. For now, we will define some constants that we will use in the sphere casting calculation, determining the plane's surface.

**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 allows us to declare an identifier we 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. At the same time, **MAX_MARCHING_STEPS** refers to the number of steps we will need to find the plane's intersection.

We continue by declaring a 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;
}
```
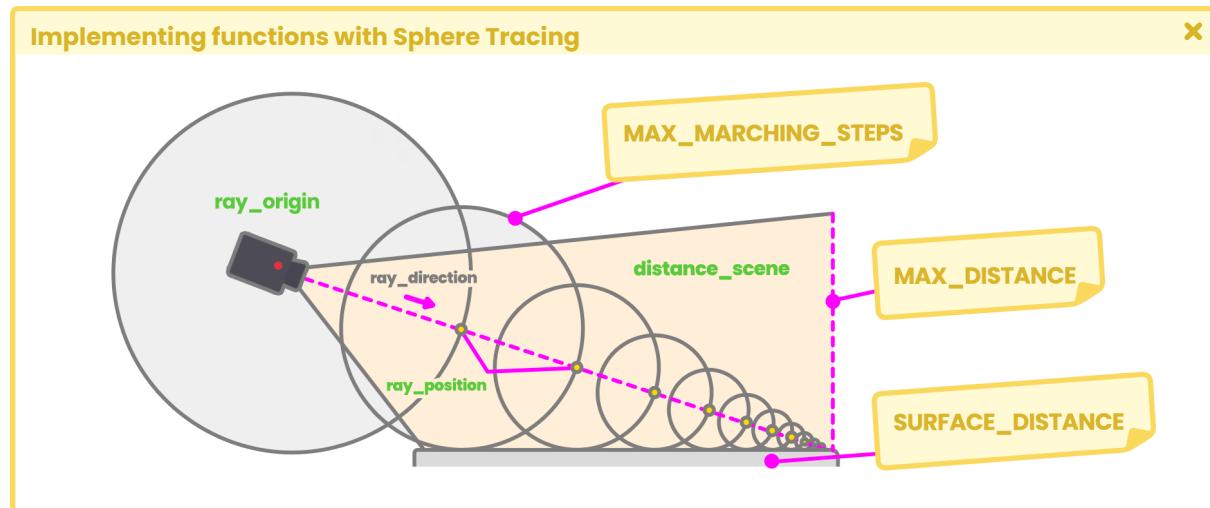
At first glance, the operation in the above example looks challenging to understand. However, it is not entirely complex. Initially, we should pay attention to its arguments.

The "*ray_origin*" vector corresponds to "the ray's starting point," e. i., the camera's position in *local-space*, while "*ray_direction*" is equal to "mesh vertices' position," in other words, to the position of the sphere we are working with, why? Since we will generate a division in the primitive according to an edge, we will need the position of the SDF plane to be equal to the position of the 3D object.



*(Fig. 11.0.2b)*

As mentioned above, it will be necessary to discard the pixels of the sphere that lie on the plane. To perform such an evaluation, we will need the *object-space* position of the vertices on the y-axis of the sphere. Using the "discard" statement, we 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 the 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, we will have to take the values from the *vertex shader* to the *fragment shader stage*.

We will create a new vector which we will call "hitPos."

337

**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 our effect. On the one hand, we will use it to define the position of the mesh vertices; and, on the other hand, to calculate the spatial plane's position 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);
    // we assign the vertex position in object-space
    o.hitPos = v.vertex;

    return o;
}
```

Finally, we can discard pixels that lie on the _Edge property.

**Implementing functions with Sphere Tracing** ✕
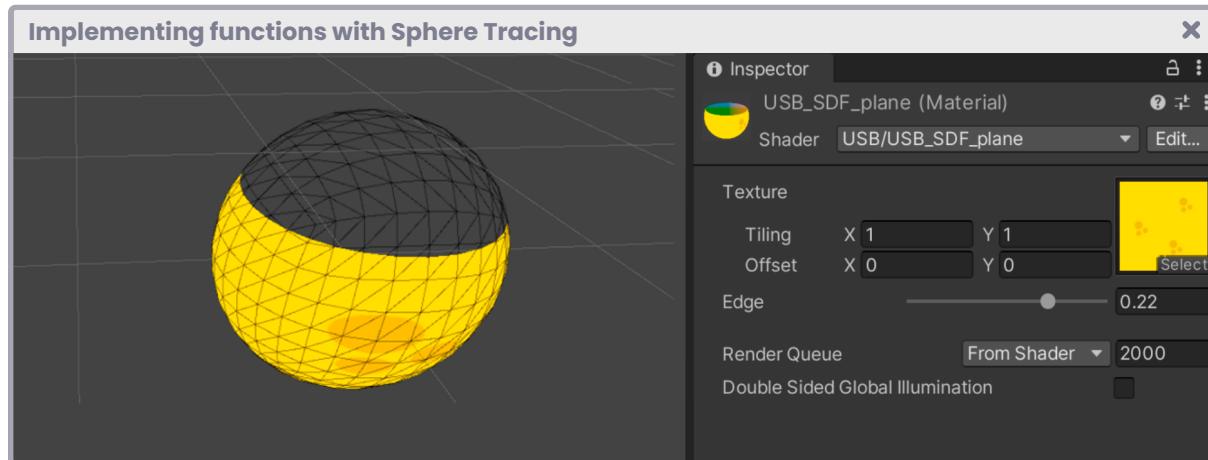
```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);


    // we discard the pixels that lie on the _Edge
    if (i.hitPos > _Edge)
        discard;


    return col;

}
```

If everything has gone well in Unity, we will appreciate a behavior similar to that of figure 11.0.2c. The _Edge property will dynamically modify the discarded pixel border.



*(Fig. 11.0.2c. _Edge equal to 0.251)*

Noteworthy that our effect is "opaque," that is, it has no transparency.
Those discarded pixels will not be executed; therefore, the program will not send them to the output color.

As we can see in Figure 11.0.2b, the ray's origin is given by the camera's position, while the ray's direction can be calculated by following the position of the vertices in the same space.

The declaration of these variables can be quickly 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);

    // use the values in the ray casting function
    float t = sphereCasting(ray_origin, ray_direction);

    // calculate the point position in space
    float3 p = ray_origin + ray_direction * t;

    if (i.hitPos > _Edge)
        discard;

    return col;
}
```

Looking at the previous example, we have stored the plane's distance concerning the camera in the "t" variable and then stored each plane's point in the variable "p."

It will be necessary to project our SDF plane onto the front face of the sphere. Consequently, we will have to disable *Culling* from the *SubShader*.

**Implementing functions with Sphere Tracing** ✕

```
SubShader
{
    …
    // show both faces of the sphere
    Cull Off
    …
    Pass
    {
        …
    }
}
```

We can project the pixels of the sphere on its back face, and for the plane, it can be projected on the front face using the SV_isFrontFace semantic.

**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);
}
```
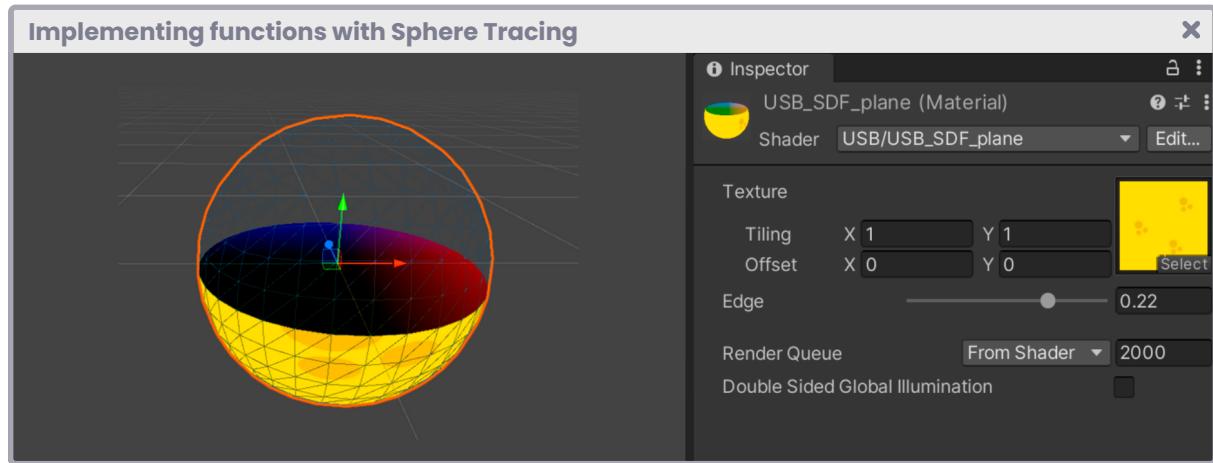
If we return to the scene, we can determine the position on the y-axis of the SDF plane using the _Edge property, as shown in Figure 11.0.2d.



*(Fig. 11.0.2d)*

## 11.0.2 | Projecting a texture.

Continuing with our **USB_SDF_fruit** shader, this time, we will project a texture over the SDF plane that we have previously generated. We will start by adding some properties that we 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
    }
```
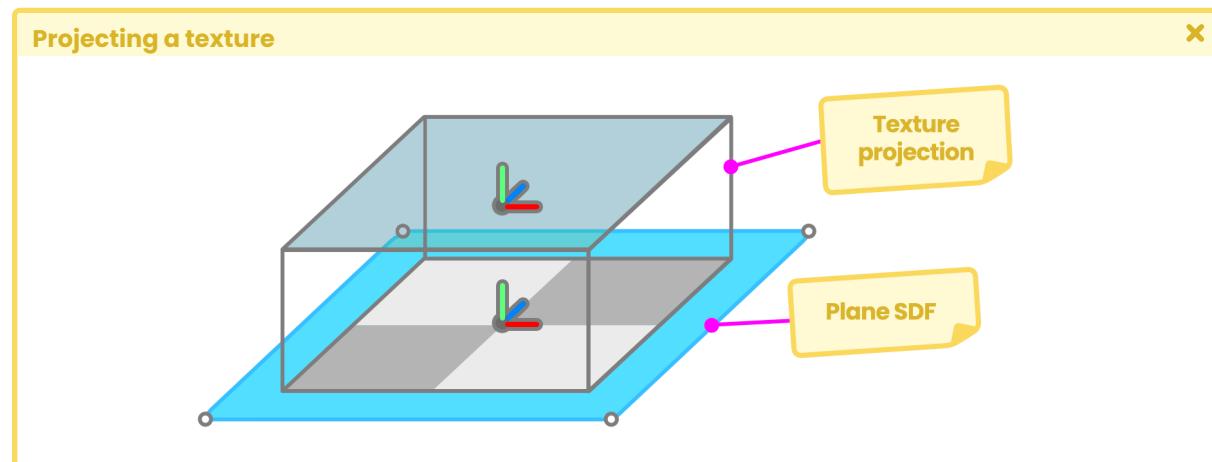
*Continued on next page.*

```
    SubShader
    {
        Pass
        {
            ...
            sampler2D _MainTex;
            sampler2D _PlaneTex;
            float4 _MainTex_ST;
            float4 _CircleCol;
            float _CircleRad;
            float _Edge;

            ...
        }
    }
}
```

For this case, if we want to project a texture on the SDF plane, we 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-axis. Therefore, the texture should be oriented in the opposite direction.



*(Fig. 11.0.3a)*

To do this, we can calculate UV coordinates within 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 we can use these coordinates in the tex2D 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;
```

*Continued on next page.*
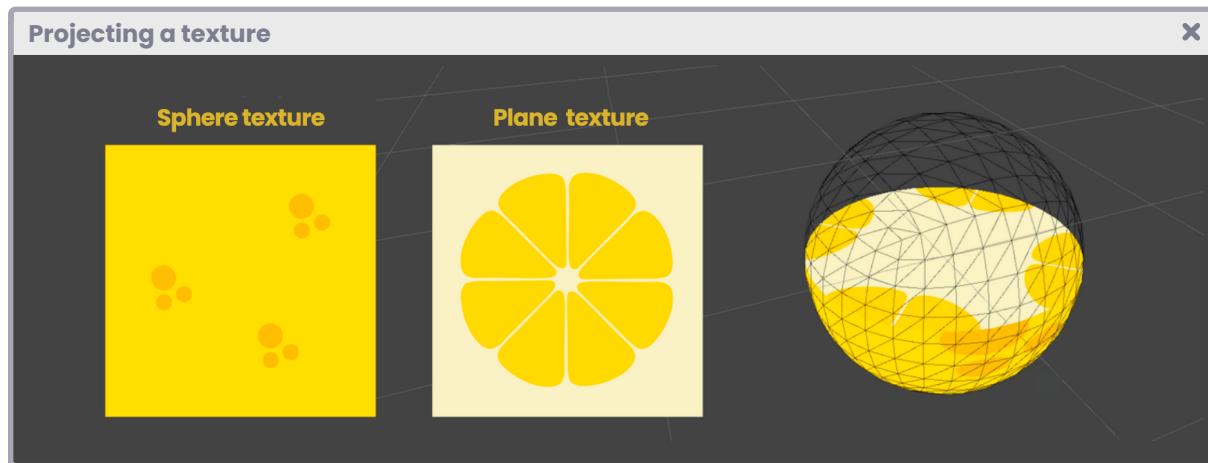
```
        planeCol = tex2D(_PlaneTex, uv_p);
    }


    if (i.hitPos > _Edge)
        discard;


    return face ? col : planeCol);
}
```
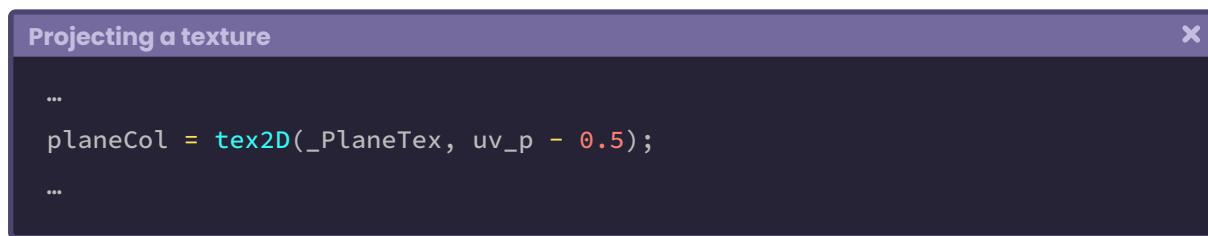
A new four-dimensional vector called "planeCol" has been declared in the previous exercise. In this vector, we have stored the projection of the texture for the SDF plane, oriented on the y-axis. If everything went well, we would see both the sphere and the plane with textures in our scene.



**Projecting a texture** ✕

**Sphere texture**　　**Plane texture**

*(Fig. 11.0.3b)*

Noteworthy that the starting point of the coordinate "uv_p" is at the center of the grid (0x, 0y, 0z); therefore, it will be necessary to subtract 0.5f to center the plane's projection.

**Projecting a texture** ✕

```
…
planeCol = tex2D(_PlaneTex, uv_p - 0.5);
…
```

If we modify the value of the _Edge property from the inspector material, we can notice that the effect is working. However, it will be necessary to find an operation that allows us to

**345**

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."

Then, the following operation has been performed, which simply solves the exercise,

$$(-abs(x))^2 + (-abs(x) - 1)^2$$
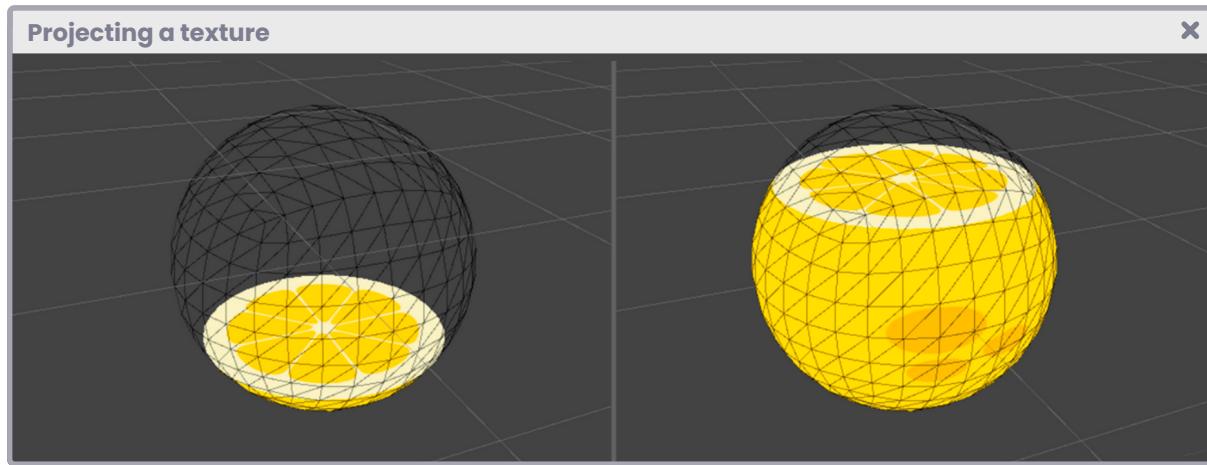
Thus,

**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);
}
...
```

We will see the plane's texture projection decreasing according to the sphere's volume once we modify the _Edge property's value.

**Projecting a texture**                                                                                    ✕



*(Fig. 11.0.3c. On the left, _Edge is equal to -0.246, while on the right, it equals 0.282.)*

We can stylize the effect by projecting a circle in the plane, which follows the sphere's circumference. To do this, we can 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, we modify the 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);
```

*Continued on next page.*

347

```
    // delete the texture borders
    planeCol *= circleCol;


    // add the circle and apply color
    planeCol += (1 - circleCol) * _CircleCol;

}

…
```
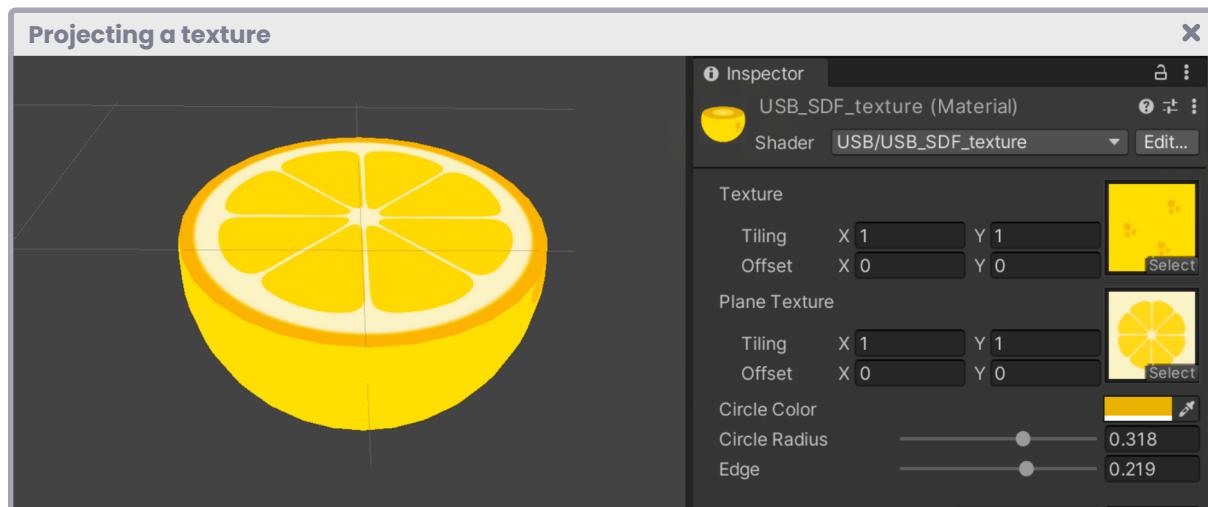
In the previous exercise, we generated a circle following the plane's UV coordinates projection and saved it in the "c" variable.

Then, the same mathematical scheme was used to increase or decrease the circle's radius. Finally, considering that the circle has only black and white color, the edges of the texture projection have been removed; why? It is 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 Inspector material.
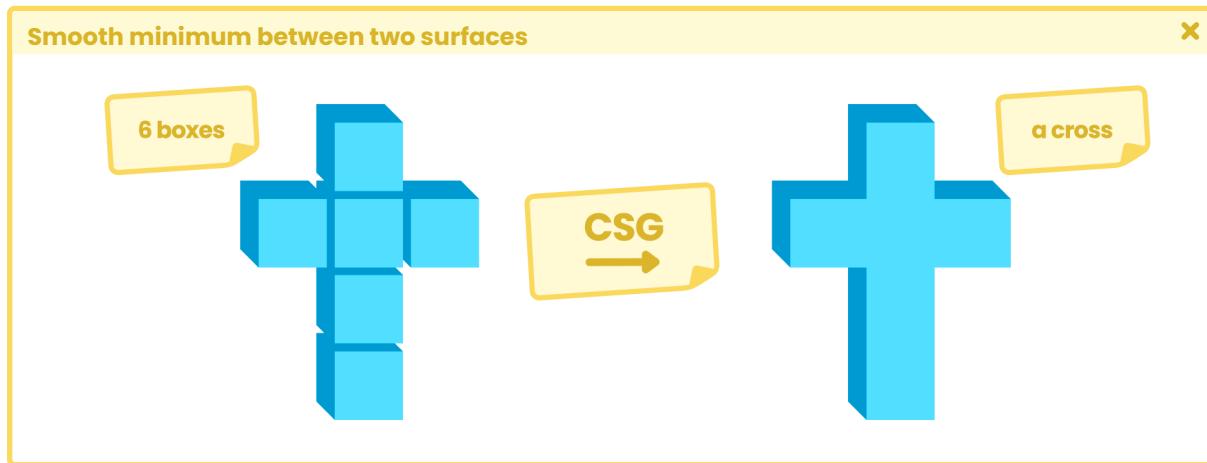


*(Fig. 11.0.3d)*

## 11.0.3 | Smooth minimum between two surfaces.

When working with Sphere Tracing, it is common to use operators for the elaborated objects' generation, e.g., if we want to create a cross in our shader, we could use six cubes,

join them and simulate the shape (as shown in the 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, and more.



*(Fig. 11.0.3a. Union of six boxes)*

One of the most used operators in this technique corresponds to the **union between two surfaces**. To calculate it, we 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 we want to mix both surfaces, we can use the solution of Íñigo Quilez, who proposes a function called "polynomial smooth minimum," which uses linear interpolation to approximate the minimum between "*a*" and "*b*," where each one refers to a specific shape.

Its syntax is as follows:

**Smooth minimum between two surfaces** ✖

```
float smin (float a, float b, float k)
{
    float h = clamp(0.5 + 0.5 * (b - a) / k, 0.0, 1.0);
    return lerp(b, a, k) - k * h * (1.0 - h);
}
```

To understand its implementation, we will start by creating a new shader of type *"Unlit Shader,"* which we will call **USB_function_SMIN**. We will make a smoothed union between two circles, which we will project onto a Quad in our scene.

We will start by defining some properties. These we will use 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;
            …
        }
    }
}
```

We will create two circles (*a* and *b*), as mentioned above. One of them will remain static, while the second one will move from side to side to appreciate the "smin" function operation. The "**_Position**" property is directly related to the circle's movement.

On the other hand, "**_Smooth**" will be used to smooth the edges of the set itself, while "**_K**" will be used to calculate the interpolation between the two circles.

We declare a function that allows us 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 we can see, the previous function has the same structure that we have seen previously. Using the "smin" function, we can calculate the minimum smoothing between two circles in the *fragment shader stage*.

**Smooth minimum between two surfaces**  ✕
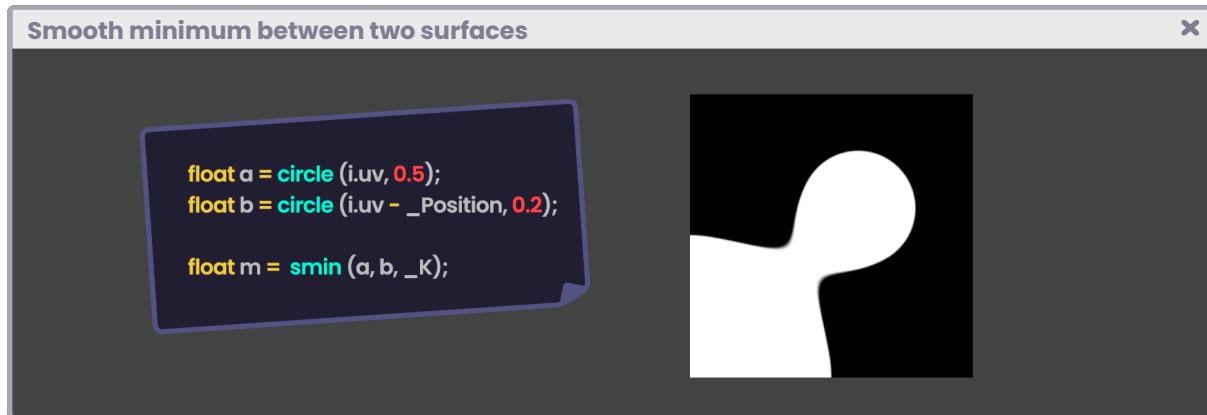
```
float circle (float2 p, float r) { … }

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);
}
```
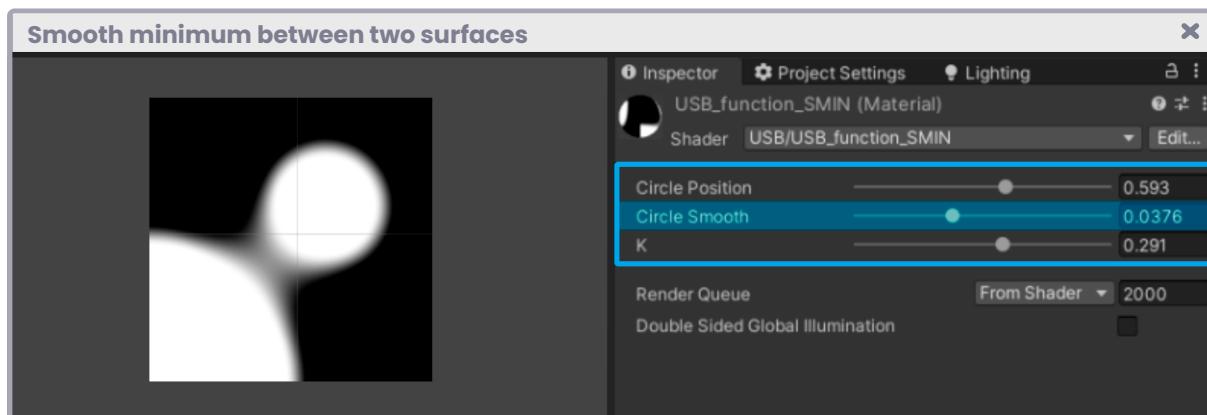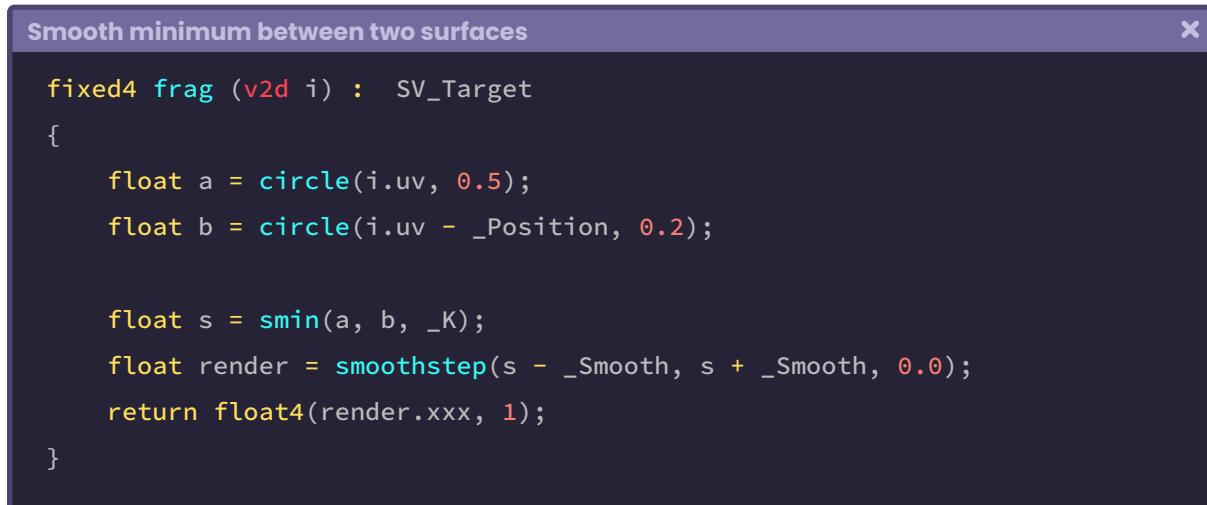
In the previous exercise, we created two scalar variables called "*a*" and "*b*." These we use in the function "smin," which returns the smoothed union between the two shapes.

*(Fig. 11.0.3c. Smooth union according to "k")*

By incorporating the "smoothstep" function into the operation, we can generate smoothed edges for the overall composition.

```
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);
    float render = smoothstep(s - _Smooth, s + _Smooth, 0.0);
    return float4(render.xxx, 1);

}
```
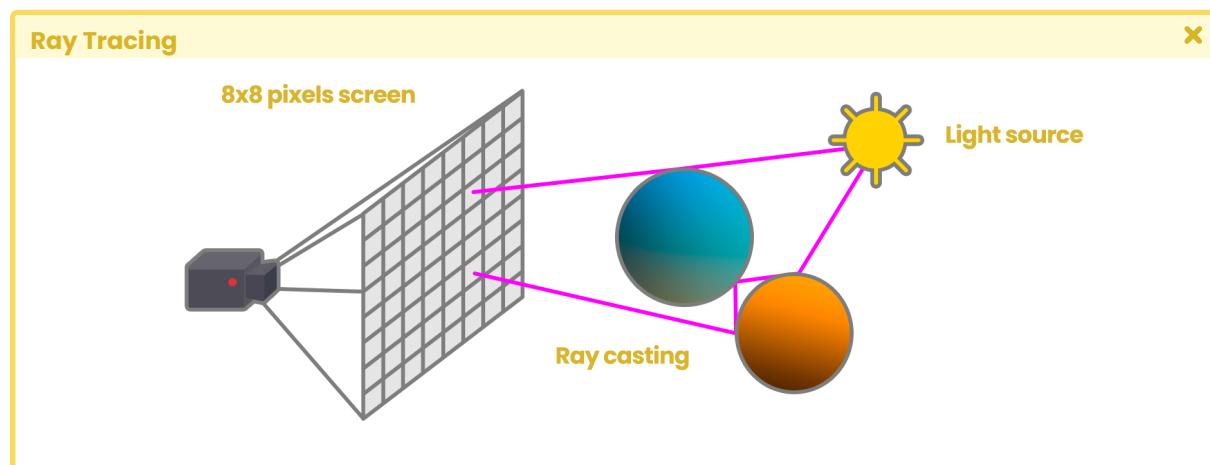


*(Fig. 11.0.3d)*

# Ray Tracing.

At this point, we have reviewed an essential part of the necessary knowledge for shaders development in Unity, both in Built-In and Universal RP. However, in this last section, we will concentrate our studies on High Definition RP understanding and its Ray Tracing configuration.

Let us start by asking the following question, what is Ray Tracing? To understand the concept, we must pay attention to the behavior of illumination 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 we can achieve a more realistic composition by sending rays through each pixel on our screen, which collide and bounce off each object in the scene.



*(Fig. 12.0.0a)*

A realistic composition can be accomplished by understanding the following characteristics.
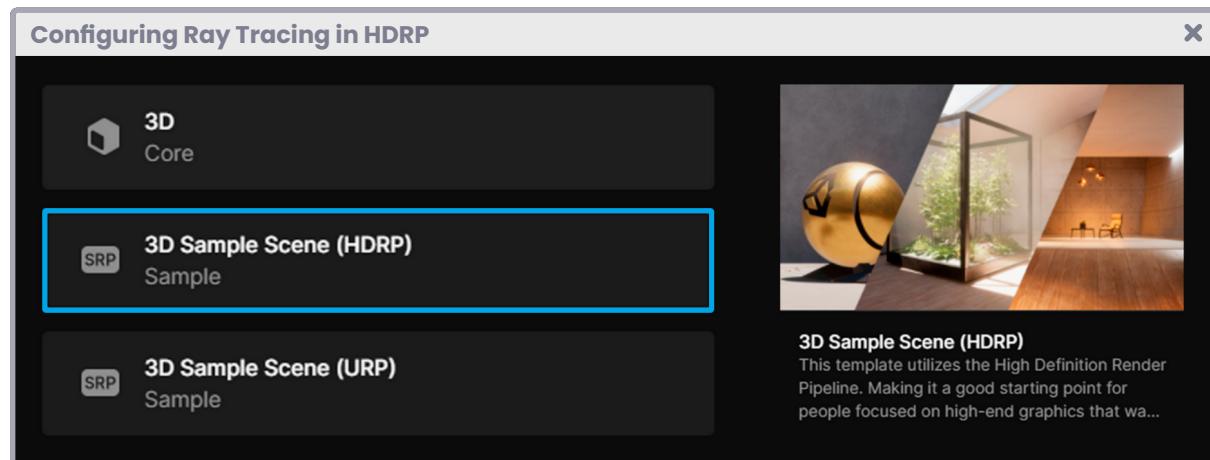
- Global illumination.
- Reflections.
- Refractions.
- Ambient occlusion.
- Shadows.

Before Ray Tracing, such properties were calculated in the software through "lightmaps," which we can get from the *Windows* | *Rendering* | *Lighting* panel. However, given the nature

of this technique, the elements in the scene had to remain static, disabling the possibility of real-time calculations.
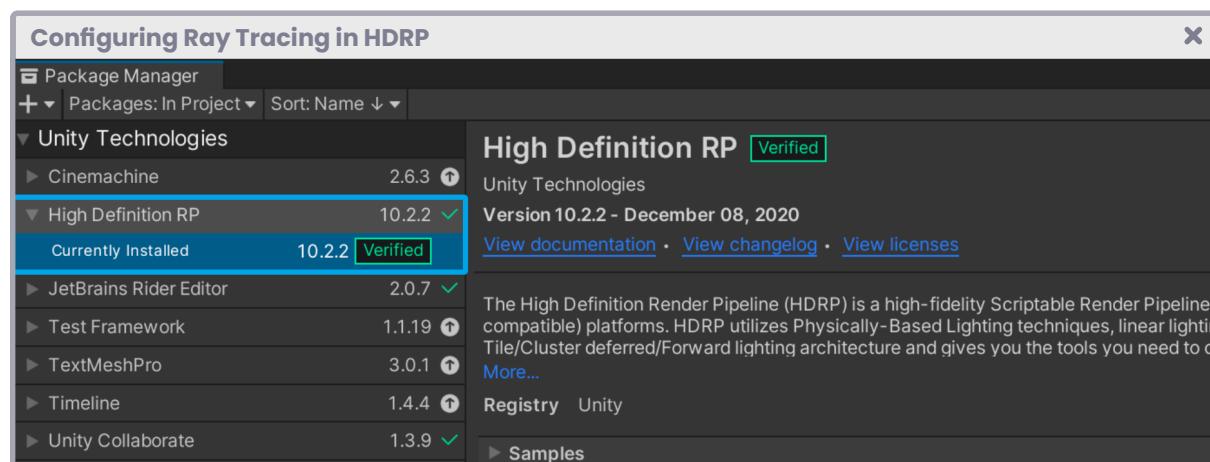
## 12.0.1 | Configuring Ray Tracing in HDRP.

We 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, using High Definition RP to perform the exercises will be necessary. We can check its configuration by going to the *Windows / Package Manager* menu and ensuring that the High Definition RP package is installed in our project (as shown in image 12.0.1b).
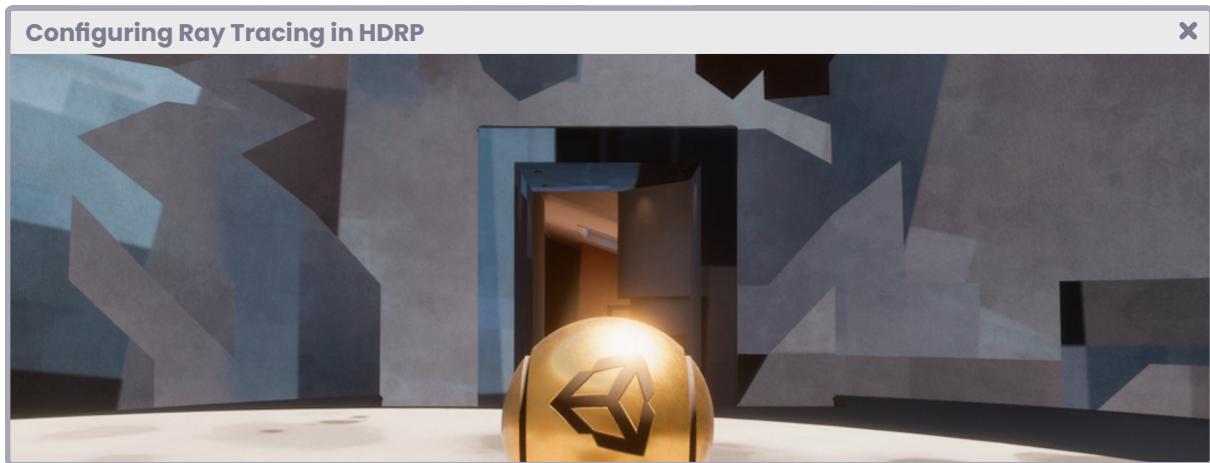


(Fig. 12.0.1b)

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.

In the opening of our scene, it is widespread that some textures look like in figure 12.0.1c. It is due to a configuration error in the generated lightmaps when creating the project.
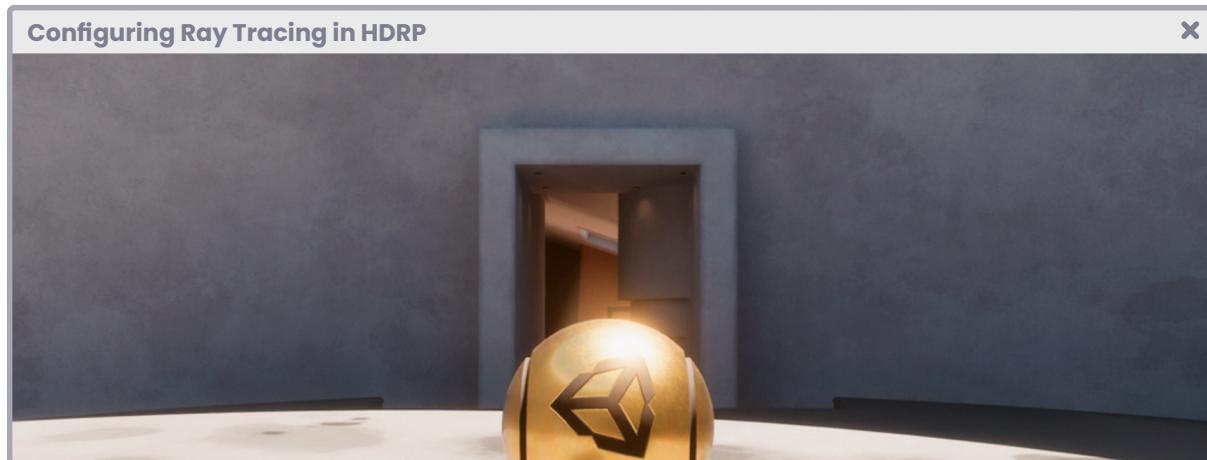


*(Fig. 12.0.1c. Walls have errors in their lightmaps)*

To solve this problem, we must pay attention to the objects' configuration. If we select any object, e.g., **FR_SectionA_01_LOD0** (wall), we will notice that it has been marked as "*static*" from the Unity Inspector.

Then, we must go to the menu *Windows / Rendering / Lighting* and perform the following operation:

1. We press the dropdown belonging to the Generate Lighting button and select Clear Baked Data. By doing this action, all the lightmaps will be eliminated, which will allow us to visualize the default lighting.
2. Next, we must press the Generate Lighting button.

The process may take a few minutes depending on our computer's capacity. However, the textures and lighting will be displayed correctly at the end of the process.

Configuring Ray Tracing in HDRP ✕

*(Fig. 12.0.1d. The lightmaps have been corrected)*

Noteworthy that the global illumination and other properties, such as ambient occlusion, are being "baked" on each texture. Therefore, if we 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 in our project. For this, we will have to consider several configurations for it, including DirectX 12 (DX12). The whole process can be summarized in three main steps;

1. Render Pipeline Asset.
2. Project Settings.
3. DirectX 12.
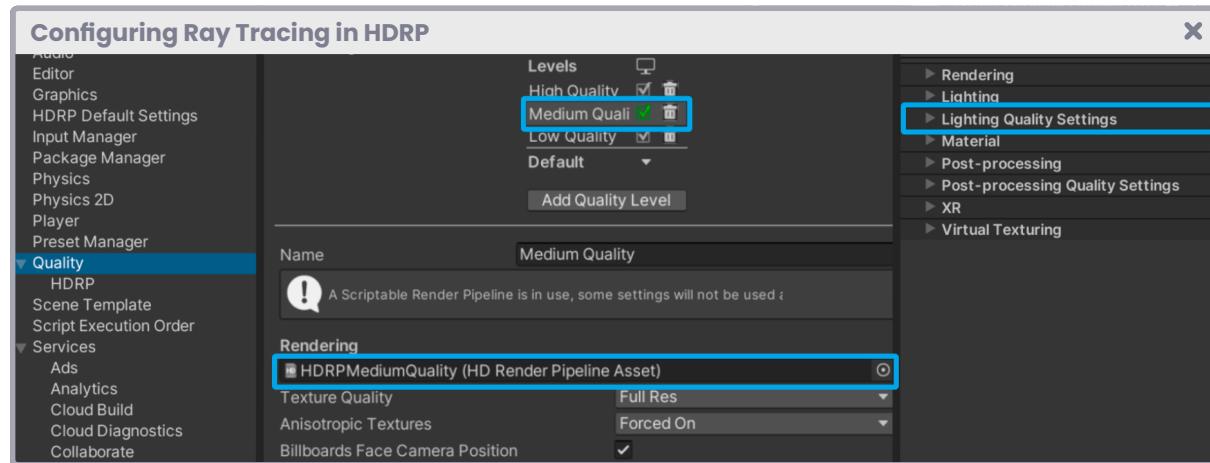
We will start by going to the menu *Windows / Panels / Project Settings*, and we will pay attention to the following categories:

• Quality.
• Graphics.
• HDRP Default Settings.

Note that Unity creates a different rendering configuration for each quality level in our 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 we want to work with Ray Tracing, we will have to enable its options from the previously configured Asset; in this particular case, Medium Quality.
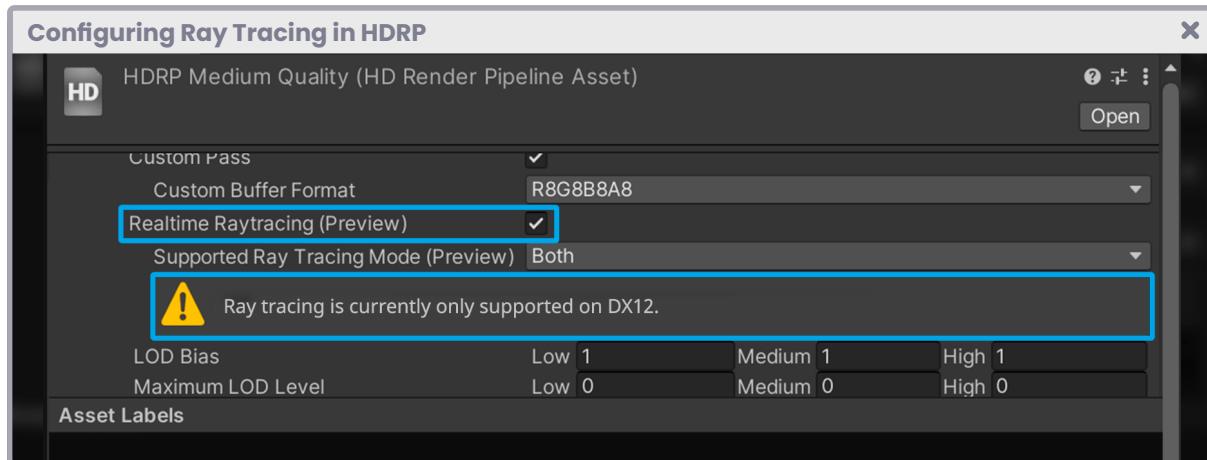


*(Fig. 12.0.1e. Medium Quality configuration)*

After selecting the HD Render Pipeline Asset from our project, we must pay attention to the "*Rendering*" and "*Lighting*" menu found in the Unity Inspector. Starting in the dropdown of the first one, we will have to activate the "Realtime Ray Tracing (Preview)" option, as shown in Figure 12.0.1f.

If our 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, we will have to change its configuration in our project later. For now, we will continue to enable some options that will allow us to perform global illumination calculations and other features.

*(Fig. 12.0.1f)*

Next, we will go to the "Lighting" menu and enable the following options:

- Screen Space Ambient Occlusion.
- Screen Space Global Illumination.
- Screen Space Reflection.
- Screen Space Shadows.
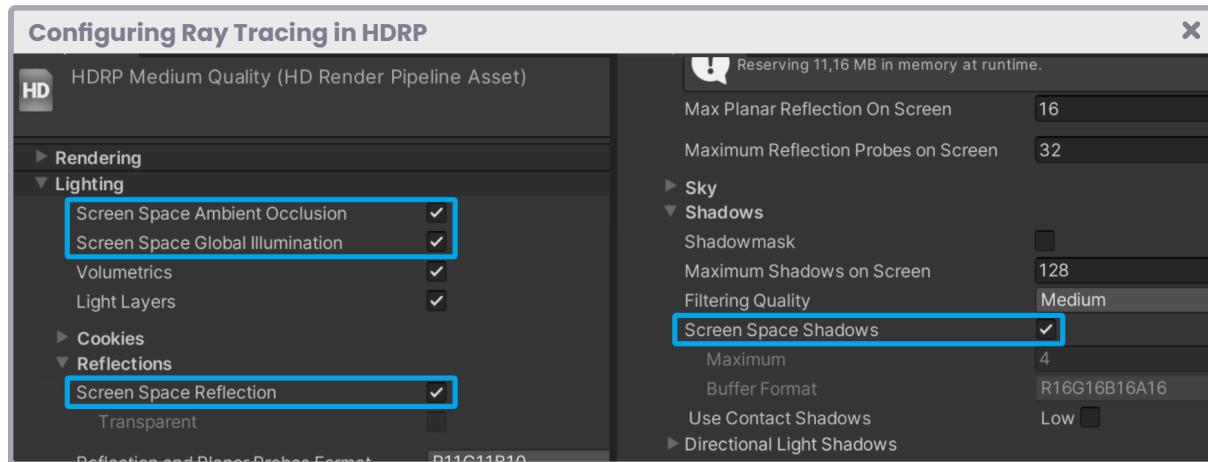
After the "*clip-space*" coordinate continues "*screen-space*," which refers to the transformation of coordinates between -1.0f to 1.0f to screen coordinates. Therefore, we can deduce that the higher the resolution, the higher the Ray Casting calculation. Therefore, the more power we will need in the 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)** will allow us to calculate the bounce of illumination in real-time, generating a more accurate light representation of the composition in our scene.

**Screen Space Reflection** will allow us to calculate reflections in real-time.
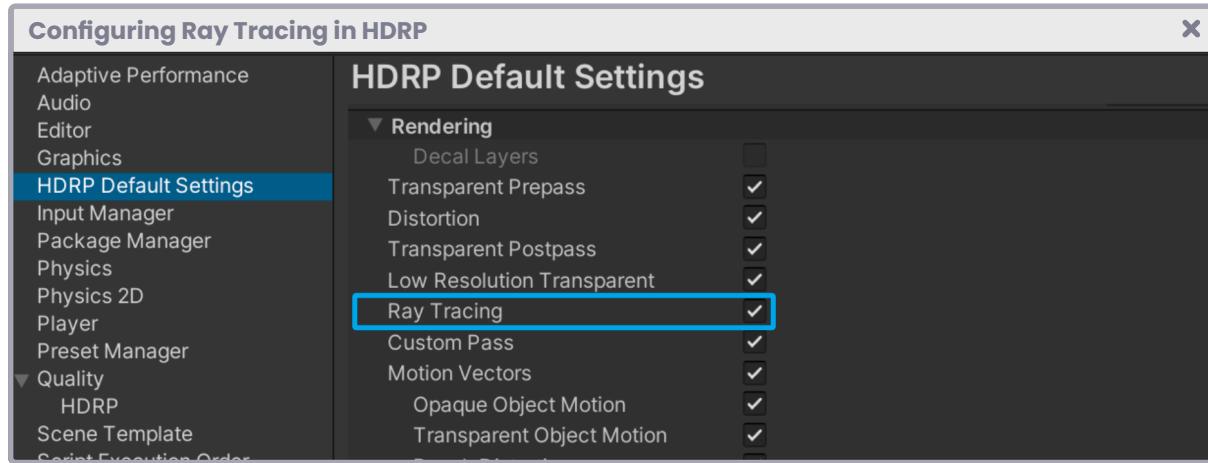
The same analogy is valid for **Screen Space Shadows**, which improves the projection of shadows in our project.

*(Fig. 12.0.1g. The different options have been enabled from the Lighting dropdown)*
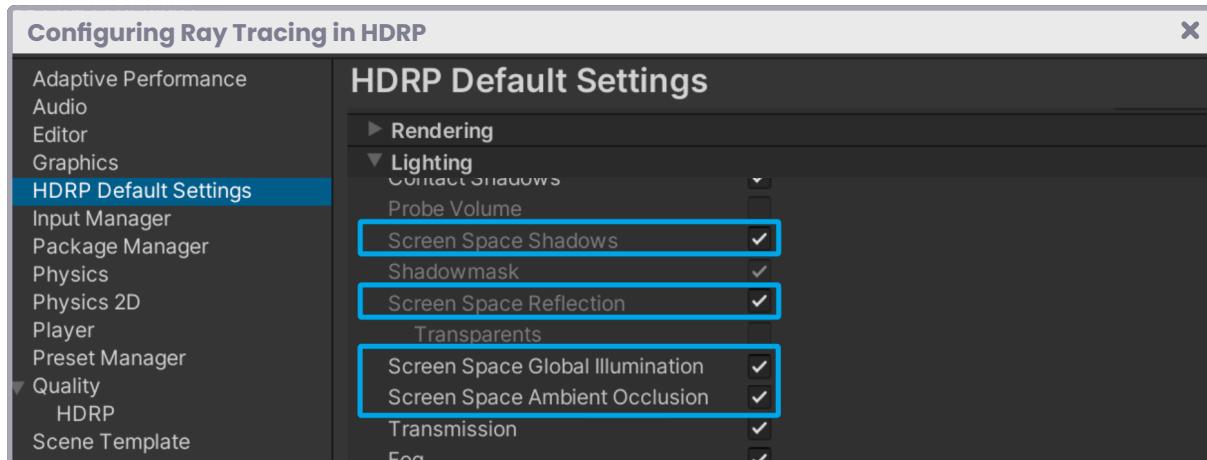
Up to this point, Ray Tracing and its properties are already enabled in the Render Pipeline Asset. Then, we must configure our project so that these properties can perform their operations.

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*.
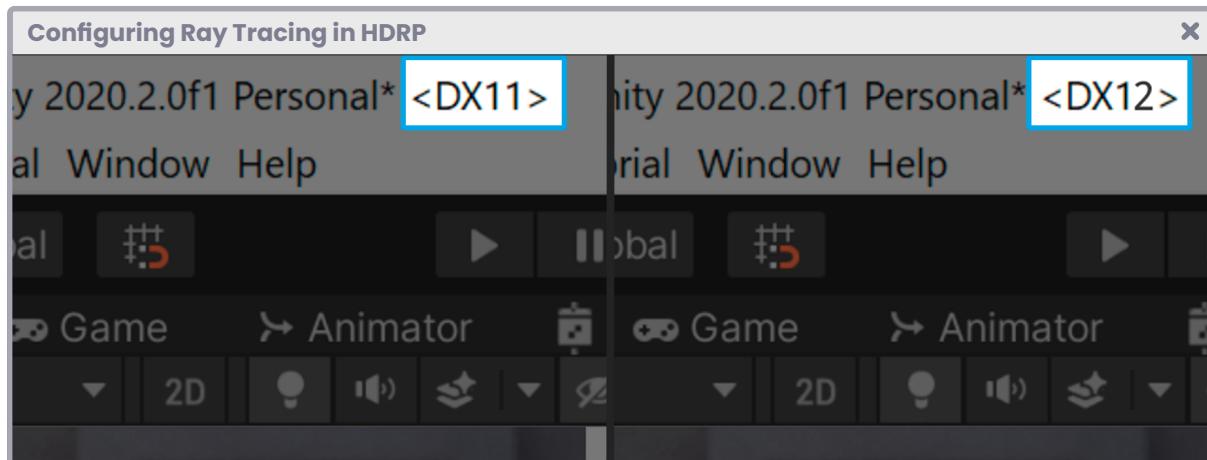


*(Fig. 12.0.1h. Enabling Ray Tracing for scene's camera)*

Then, from the *Lighting* menu, we must enable the same options that we activated in the Render Pipeline Asset: **Screen Space Shadows**, **Screen Space Reflection**, **Screen Space Global Illumination**, and **Screen Space Ambient Occlusion**.
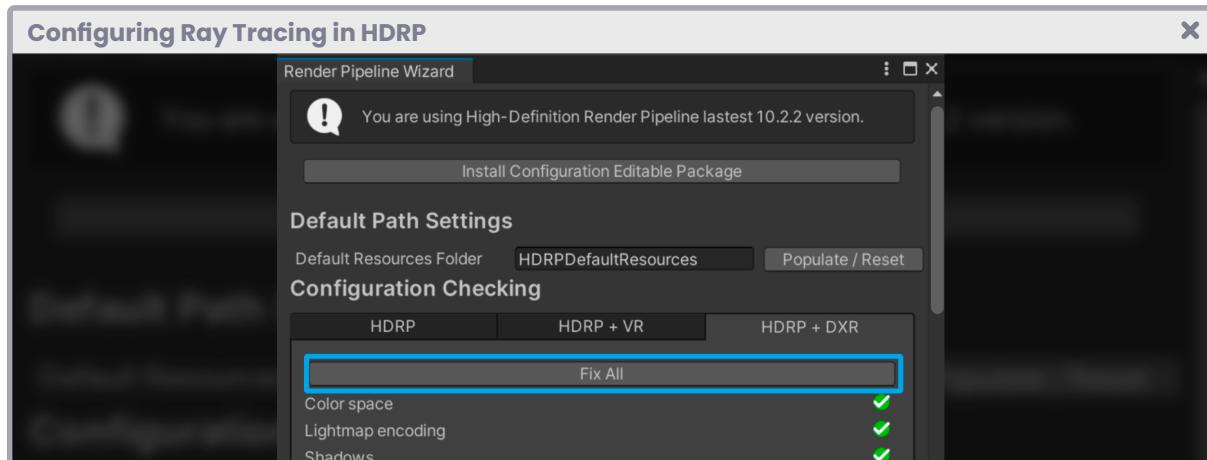
*(Fig. 12.0.1i)*

At the moment, *Ray Tracing* is already configured in our project. However, we must change our **DirectX** configuration since, as mentioned above, this technique only works in version 12, and our project, by default, has been configured in **DX11**. We can verify this in the Unity window on the toolbar, as shown in figure 12.0.1j.



*(Fig. 12.0.1j)*

To do this, we must go to the menu *Windows / Render Pipeline / HD Render Pipeline Wizard*. We must 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 we load our project again, Unity will appear with the **<DX12>** tag at the top of the interface. If we go back to the *Render Pipeline Wizard* window, we will notice that all properties appear in green, which means that Ray Tracing is enabled for them.

## 12.0.2 | Using Ray Tracing in our scene.

We will start by creating a new scene in our project. For the exercise, we will use a template included in our project, called "Basic Outdoors (HDRP)," which is characterized by having the following objects by default:
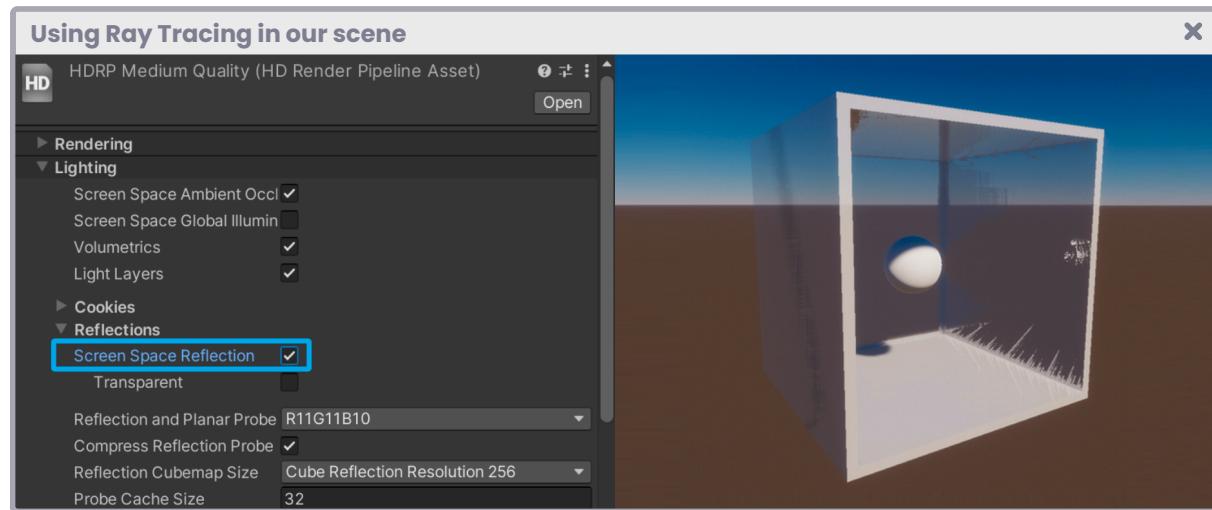
- A camera (Main Camera).
- A directional light (Sun).
- A sky (Sky and Fog Volume).

Noteworthy that we will use a room and a sphere to exemplify the exercise. Such ".fbx" objects can be found in the package attached to this book in their respective section.

We will create two materials in our project, one for each element. We will call the material for the room "**mat_room**," while the material for the sphere we will name "**mat_sphere**." We will make sure to assign the **HDRP/Lit** shader to both.

Before we begin, we will assign each material to its respective object.

**361**

Previously, we enabled the Screen Space Reflection property from the Render Pipeline Asset; therefore, if we increase the value of the "Metallic and Smoothness" properties in any of the materials, we will be able to visualize reflections in real-time, as shown in figure 12.0.2a. However, such reflection depends on the camera's angle of view, generating graphic artifacts.
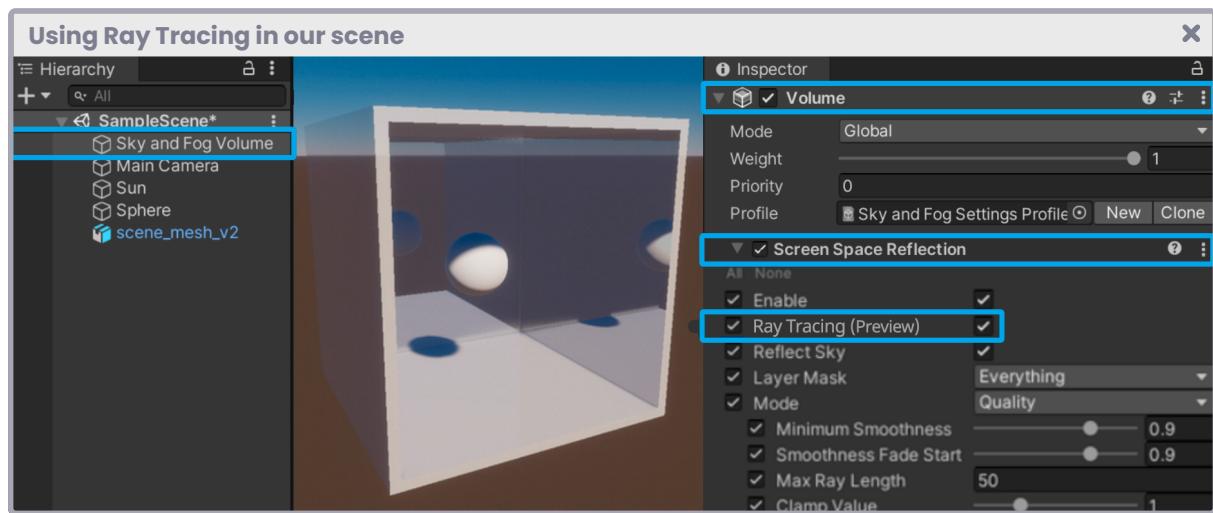


*(Fig. 12.0.2a. mat_room material, Metallic equal to 0.5f, Smoothness equal to 1.0f)*

If we want to activate reflections through Ray Tracing, we 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 the exercise, we will activate all the "**Override**" properties. However, we will appreciate significant graphical changes once we 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, we can increase or decrease the amount of bounce for the reflected rays.

We can use the same analogy to configure the ambient occlusion and global illumination. To do this, we 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, we must go to the global light in our scene (Sun), select the Shadows menu, and enable **Screen Space Shadows**.

We must make sure to activate the property "Ray Traced Shadows (Preview)" for it to take effect. Once this process is done, we can go to the **Shape** menu and modify the angular diameter according to the needs of our project.

# Index.

Screen Space Shadows. (0)

Screen Space Ambient Occlusion. (0)

**T.**

Tan function. (0)

TBN matrix. (0)

Tangents. (0)

Types of Render Pipeline. (0)

Type of shaders. (0)

Tags Queue. (0)

Tags Render Type. (0)

Time and animation. (0)

**U.**

UV coordinates. (0)

Unlit Shader. (0)

Universal Render Pipeline (0).

**V.**

Vertex Color. (0)

Vertices. (0)

Vertex shader stage. (0)

Vectors. (0)

View-Space. (0)

**W.**

World-Space. (0)

Which rendering engine should I use? (0)

What is a shader? (0)

# Special Thanks.

Taneli Nyyssönen; 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 湊; Tesseraction 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 Polasik; 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 Черендаков; 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 Lęcznar; David Nieves Trujillo; 友太 本山; Carsten Flöth; Ángel Siendones Sillero; Mikko McMenamin; Richard Le; Ericke Maciel; Максим Хламов; Alessandro Salvati; Marc Jensen; Ariel Nuńez 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; Berlingeri 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; בוקלו יירדנא; 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 Шишкин; 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 Vidstid; 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; Pookzzz3d 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

**"Jettelly wishes you success in your professional career".**