Free Sample

THE UNITY SHADERS BIBLE

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

Written by
Fabrizio Espíndola

Designed by
Pablo Yeber


Jettelly

The Unity Shaders Bible.

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

(First Edition)

Fabrizio Espíndola.
Game developer & Technical artist.

The Unity Shaders Bible, version 0.1.5b.

DDI 2021-A-11866
ISBN 979-883-3189-84-9

Credits.

Author.
Fabrizio Espíndola.

Design.
Pablo Yeber.

Technical Revision.
Daniel Santalla.

Translation, grammar, and spelling.
Martin Clarke.

About the author.

Fabrizio Espíndola is a Chilean video game developer, specialised in Computer Graphics. He has dedicated much of his career to developing visual effects and technical art, his projects include Star Wars - Galactic Defence, Dungeons and Dragons - Arena of War, 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 after Donkey Kong Country (Super Nintendo) appeared, a game that awoke a deep interest for him in the tools and techniques associated with this technology. To date, he has more than ten years of experience in the industry, and this book contains a part of the knowledge he has acquired during this time.

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

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

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

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

Content.

Preface. **10**

Chapter I | Introduction to the shader programming language.

1. Initial observations. **15**
1.0.1. | Properties of a polygonal object. 15
1.0.2. | Vertices. 17
1.0.3. | Normals. 18
1.0.4. | Tangents. 18
1.0.5. | UV Coordinates. 19
1.0.6. | Vertex Color. 20
1.0.7. | Render Pipeline Architecture. 20
1.0.8. | Application stage. 22
1.0.9. | Geometry processing phase. 22
1.1.0. | Rasterization stage. 24
1.1.1. | Pixel processing stage. 25
1.1.2. | Types of Render Pipeline. 25
1.1.3. | Forward Rendering. 27
1.1.4. | Deferred Shading. 29
1.1.5. | What Render Pipeline should I use? 29
1.1.6. | Matrices and coordinate systems. 30

2. Shaders in Unity. **35**
2.0.1. | What is a shader? 35
2.0.2. | Introduction to the programming language. 36
2.0.3. | Shader types. 38
2.0.4. | Standard Surface Shader. 39
2.0.5. | Unlit Shader. 39
2.0.6. | Image Effect Shader. 39
2.0.7. | Compute Shader. 40
2.0.8. | Ray tracing Shader. 40

3. Properties, commands and functions. **41**
3.0.1. | Structure of a Vertex-Fragment Shader. 41
3.0.2. | ShaderLab Shader. 45

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

4. Implementation and other concepts. 116
4.0.1. | Analogy between a shader and a material. 116
4.0.2. | Your first shader in Cg or HLSL. 116
4.0.3. | Adding transparency in Cg or HLSL. 118
4.0.4. | Structure of a function in HLSL. 119
4.0.5. | Debugging a shader. 123

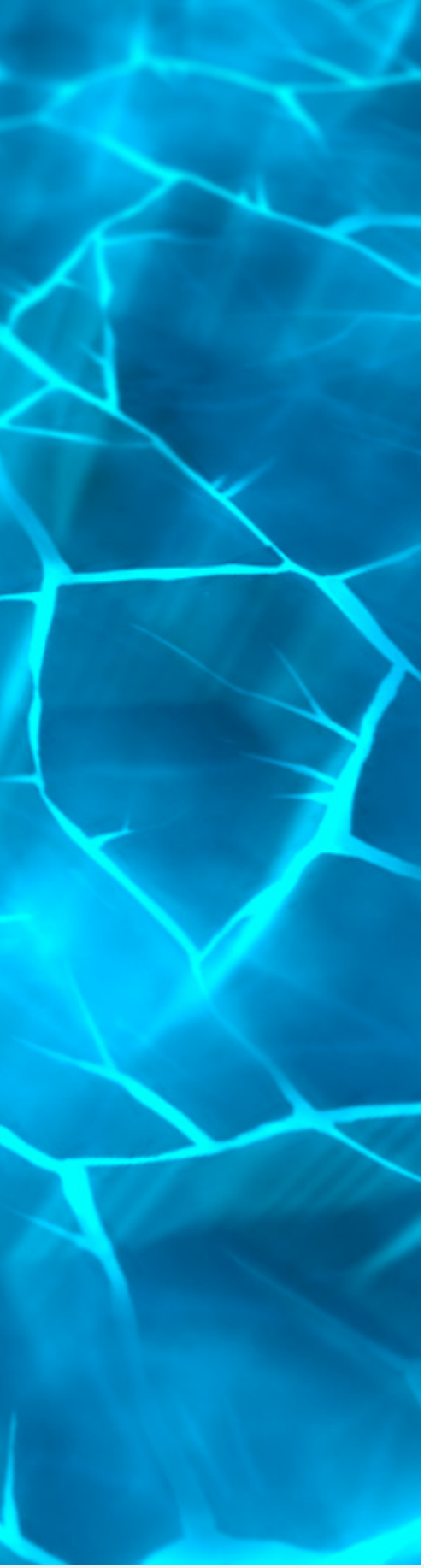
4.0.6. | Adding URP compatibility. 126
4.0.7. | Intrinsic functions. 131
4.0.8. | Abs function. 131
4.0.9. | Ceil function. 136
4.1.0. | Clamp function. 141
4.1.1. | Sin and Cos functions. 146
4.1.2. | Tan function. 151
4.1.3. | Exp, Exp2 y Pow functions. 155
4.1.4. | Floor function. 157
4.1.5. | Step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

Chapter II | Lighting, shadow and surfaces.

5. Introduction to the chapter. 182
5.0.1. | Configuring inputs and outputs. 182
5.0.2. | Vectors. 187
5.0.3. | Dot Product. 189
5.0.4. | Cross Product. 193

6. Surface. 195
6.0.1. | Normal Maps. 195
6.0.2. | DXT compression. 201
6.0.3. | TBN Matrix. 206

7. Lighting. 208
7.0.1. | Lighting model. 208
7.0.2. | Ambient Color. 208
7.0.3. | Diffuse Reflection. 211
7.0.4. | Specular Reflection. 221
7.0.5. | Environmental Reflection. 233
7.0.6. | Fresnel Effect. 243
7.0.7. | Structure of a Standard Surface. 251
7.0.8. | Standard Surface Input & Output. 253



8. Shadow. **255**
8.0.1. | Shadow Mapping. 255
8.0.2. | Shadow Caster. 256
8.0.3. | Shadow Map Texture. 261
8.0.4. | Shadow implementation. 266
8.0.5. | Built-in RP Shadow Map optimization. 270
8.0.6. | Universal RP Shadow Mapping. 274

9. Shader Graph. **281**
9.0.1. | Introduction to Shader Graph. 281
9.0.2. | Starting in Shader Graph. 283
9.0.3. | Analyzing its interface. 284
9.0.4. | Your first shader in Shader Graph. 287
9.0.5. | Graph Inspector. 294
9.0.6. | Nodes. 296
9.0.7. | Custom Functions. 298

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

10. Advanced concepts. **303**
10.0.1. | Compute Shader structure. 304
10.0.2. | Your first Compute Shader. 308
10.0.3. | UV coordinates and texture. 321
10.0.4. | Buffers. 325

11. Sphere Tracing. **336**
11.0.1. | Implementing functions with Sphere Tracing. 338
11.0.2. | Projecting a texture. 347
11.0.3. | Smooth minimum between two surfaces. 353

12. Ray Tracing. **358**
12.0.1. | Configuring Ray Tracing in HDRP. 359
12.0.2. | Using Ray Tracing in your scene. 366

Index. **369**
Special thanks. **372**

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

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

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

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

I. Topics we will see in this book.

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

Chapter I: Introduction to the shader programming language.

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

Chapter II: Lighting, shadow, and surfaces.

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

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

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

II. Recommendations.

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

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

III. Who this book is for.

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

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

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

IV. Glossary.

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

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

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

V. Errata.

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

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

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

VI. Assets and donations.

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

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

Jettelly is an independent video game development studio, your support is critical to us. If you want to support us financially, you can do so directly through our PayPal account.
→ **paypal.com/paypalme/jettelly**

VII. Piracy.

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

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

Chapter I

Introduction to the shader programming language.

Initial Observations.

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

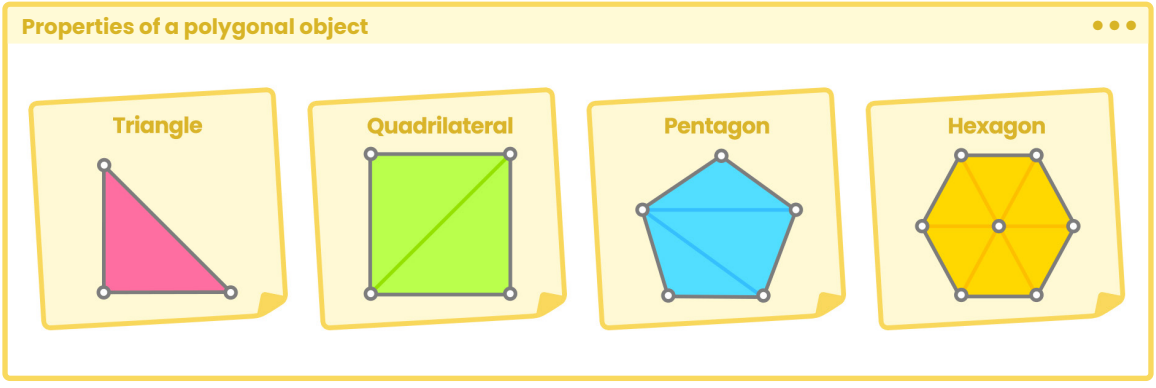
> *Vertex position in object-space.*

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

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

1.0.1. Properties of a polygonal object.

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



(Fig. 1.0.1a)

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

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

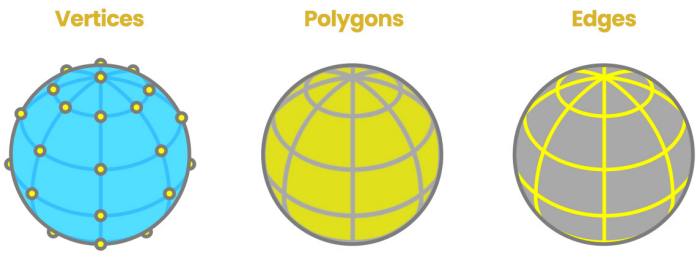
All these objects are different in shape but have similar properties; all have:

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

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

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

Properties of a polygonal object



(Fig. 1.0.1b)

1.0.2. Vertices.

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

Two main things characterize these points:

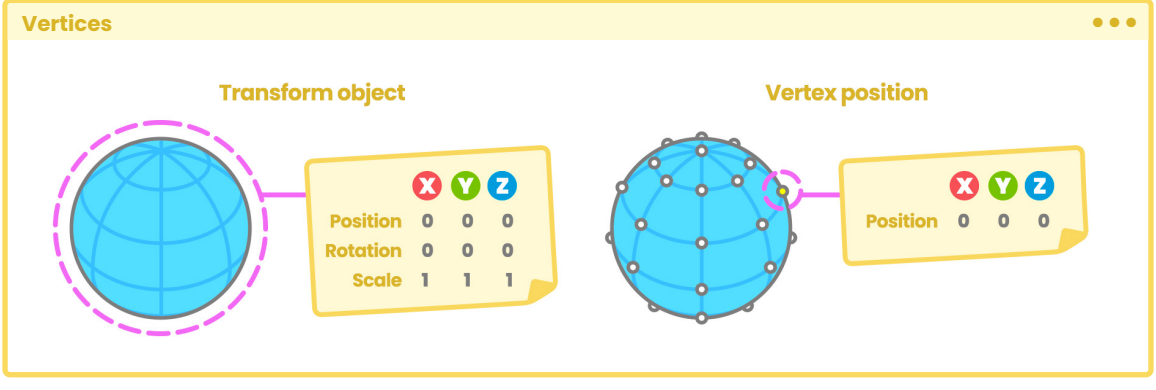
1. They are children of the Transform component.
2. They have a defined position according to the center of the total volume of the object.

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

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

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

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



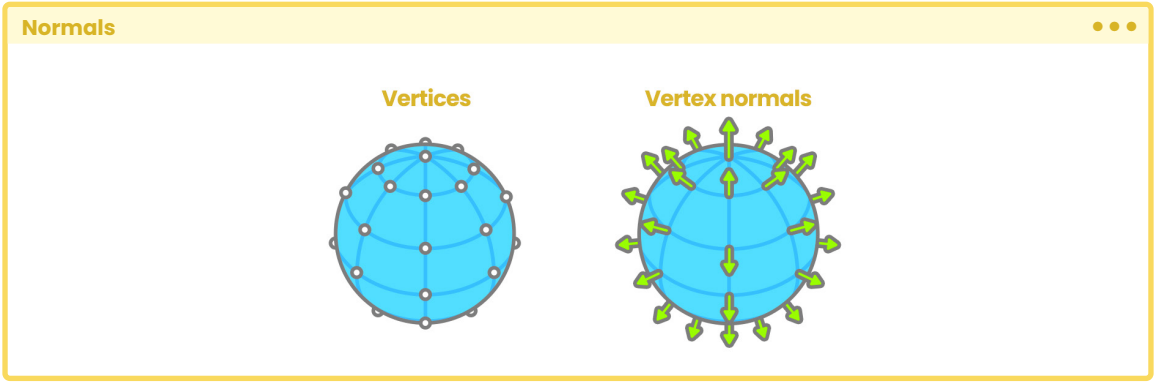
(Fig. 1.0.2a)

1.0.3. Normals.

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

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

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



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

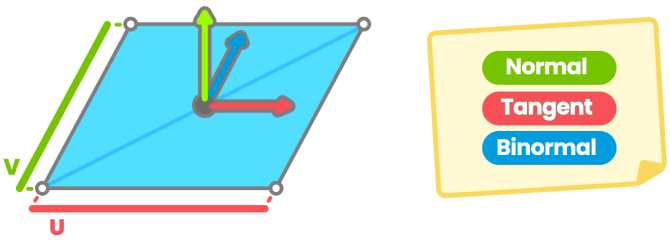
1.0.4. Tangents.

According to official Unity documentation:

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

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

Tangents



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

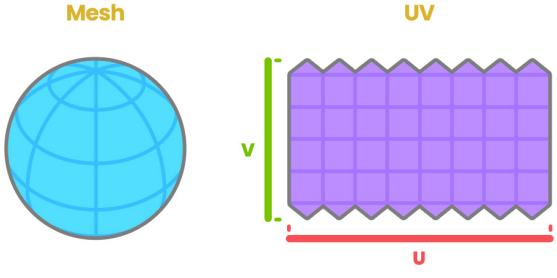
This property is reviewed in detail later, in Chapter II, section 6.0.1, as well as the **Binormals** for the implementation of a Normal Map over an object.

1.0.5. UV coordinates.

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

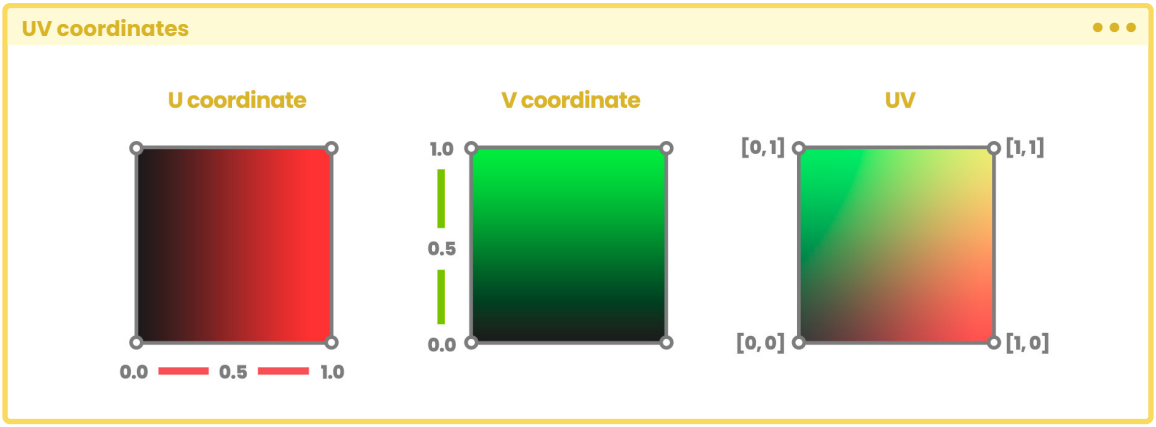
The process of positioning vertices over UV coordinates is called **UV mapping** and is a process by which UV, that appears as a flattened, two-dimensional representation of the object's mesh, is created, edited, and organized. You can access this property within your shader, either to position a texture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

UV coordinates



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

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



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

1.0.6. Vertex Color.

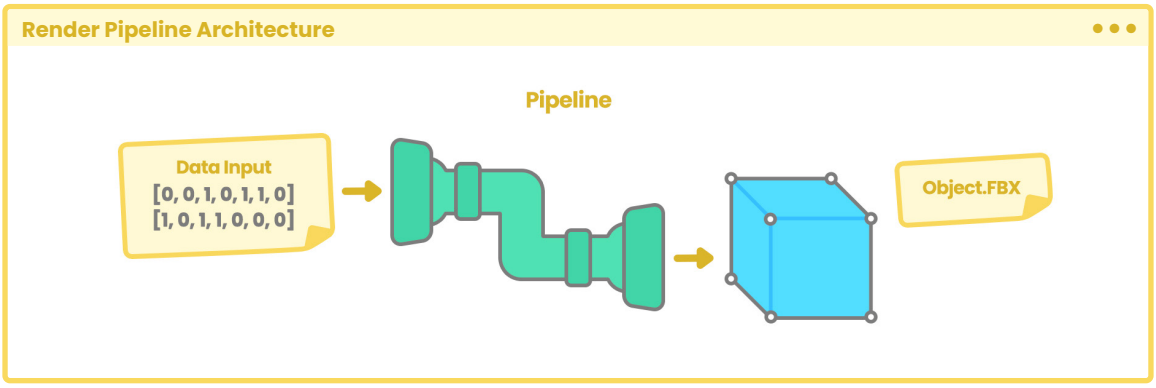
When you export an object from 3D software, it assigns a color to the object to be affected, either by lighting or multiplication of another color. Such color is known as **Vertex Color** and is white by default, with the values 1.0f in RGBA channels.

1.0.7. Render Pipeline Architecture.

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

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

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



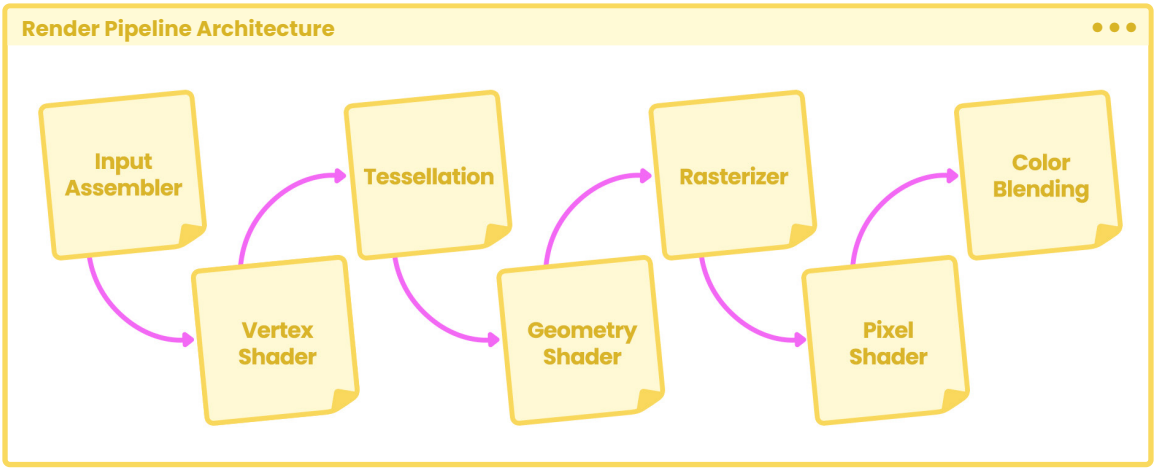
(Fig. 1.0.7a)

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

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

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

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



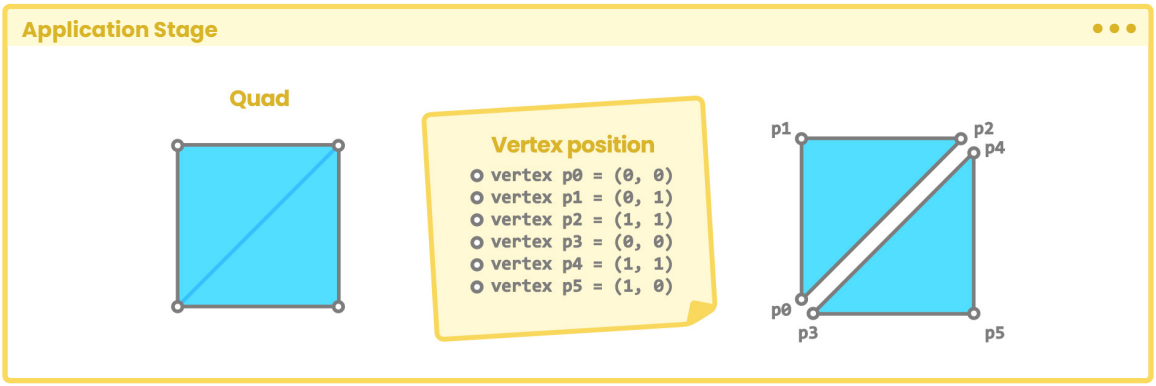
(Fig. 1.0.7b. Logic Render Pipeline)

1.0.8. Application Stage.

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

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

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



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

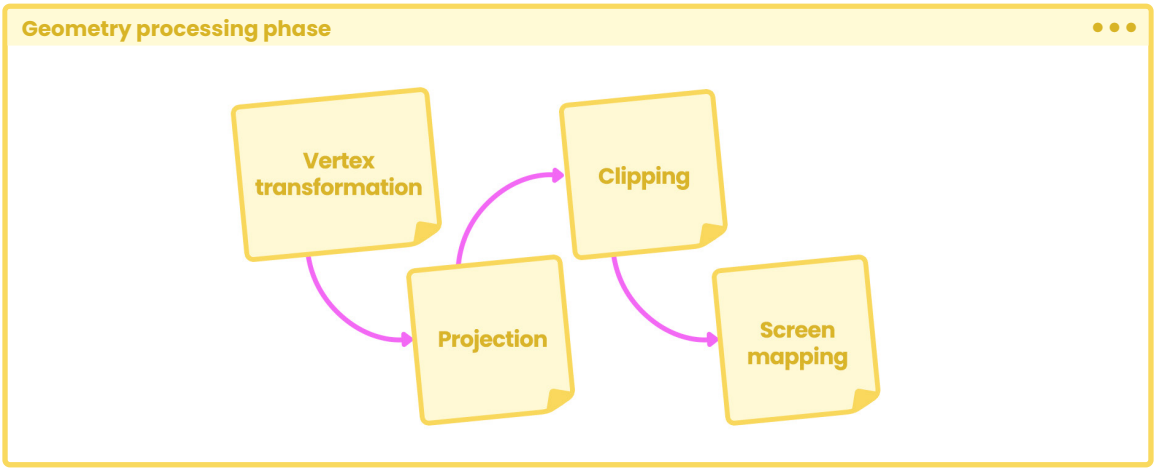
1.0.9. Geometry processing phase.

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

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

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

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



(Fig. 1.0.9a)

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

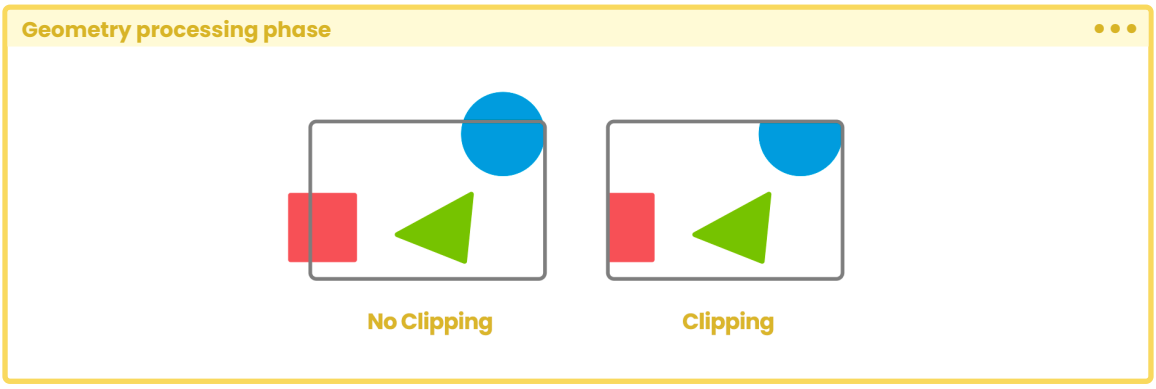
1. It calculates the object vertices position.
2. Then it transforms its position to different space coordinates so that they can be projected onto the computer screen.

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

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

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

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



(Fig. 1.0.9b)

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

1.1.0. Rasterization stage.

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

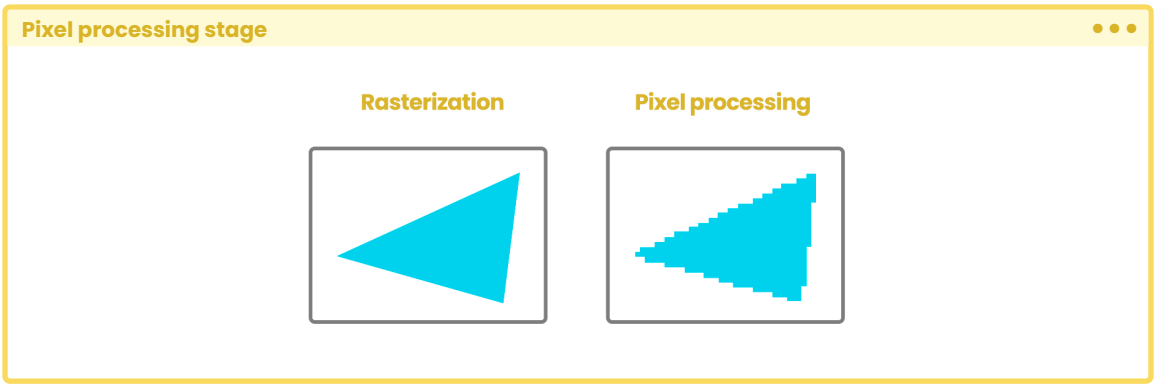
For each object, the **rasterizer** performs two processes:

1. Triangle Setup.
2. Triangle Traversal.

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

1.1.1. Pixel processing stage.

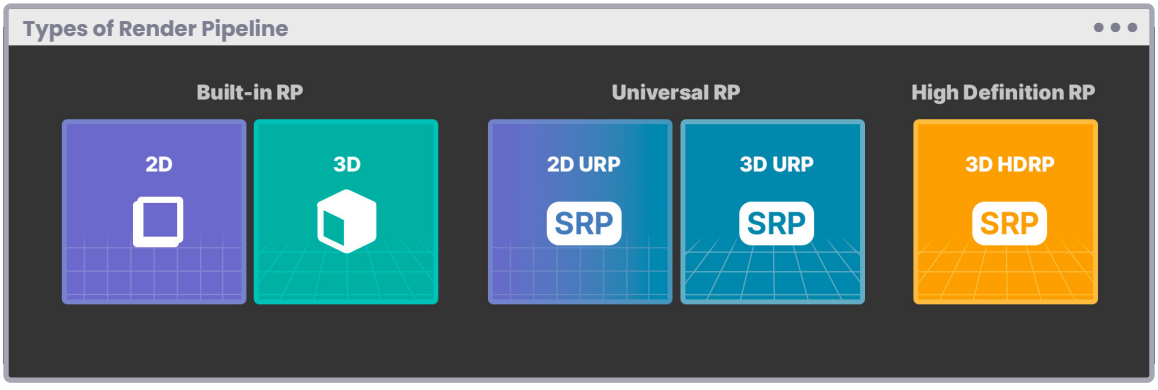
Using the interpolated values from the previous processes, this last stage starts when all the pixels are ready to be projected onto the screen. At this point, the **Fragment Shader Stage**, also known as a **Pixel Shader Stage,** begins and is responsible for the visibility of each pixel. What it does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



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

1.1.2. Types of Render Pipeline.

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



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

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

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

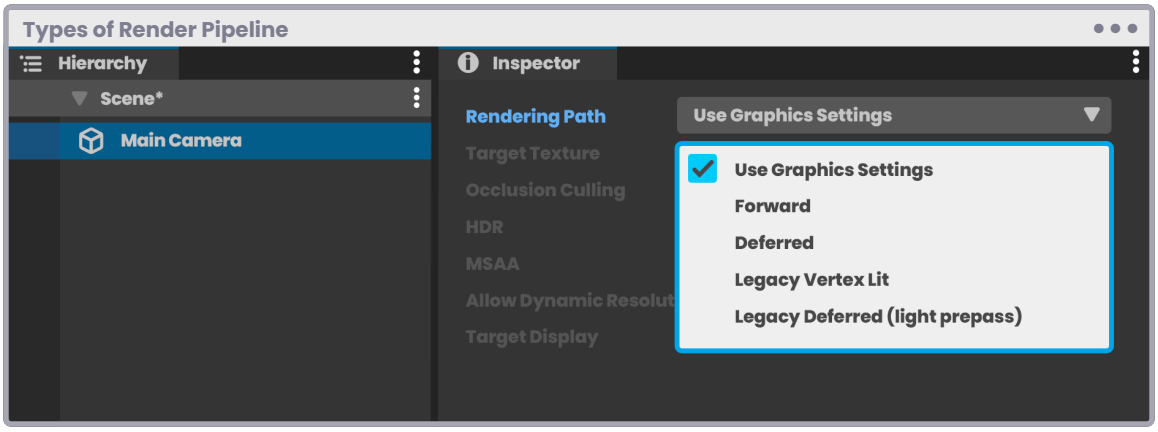
A Render Path corresponds to a series of operations related to lighting and shading objects. This allows you to graphically process an illuminated scene (e.g., a scene with directional light and a sphere).

Among them you can find:

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

Each of these has different capabilities and performance characteristics.

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



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

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

1. Lighting characteristics.
2. Object material characteristics.

Such interaction is called the **lighting model**.
The basic lighting model corresponds to the sum of three different properties:

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

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

1.1.3. Forward Rendering.

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

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

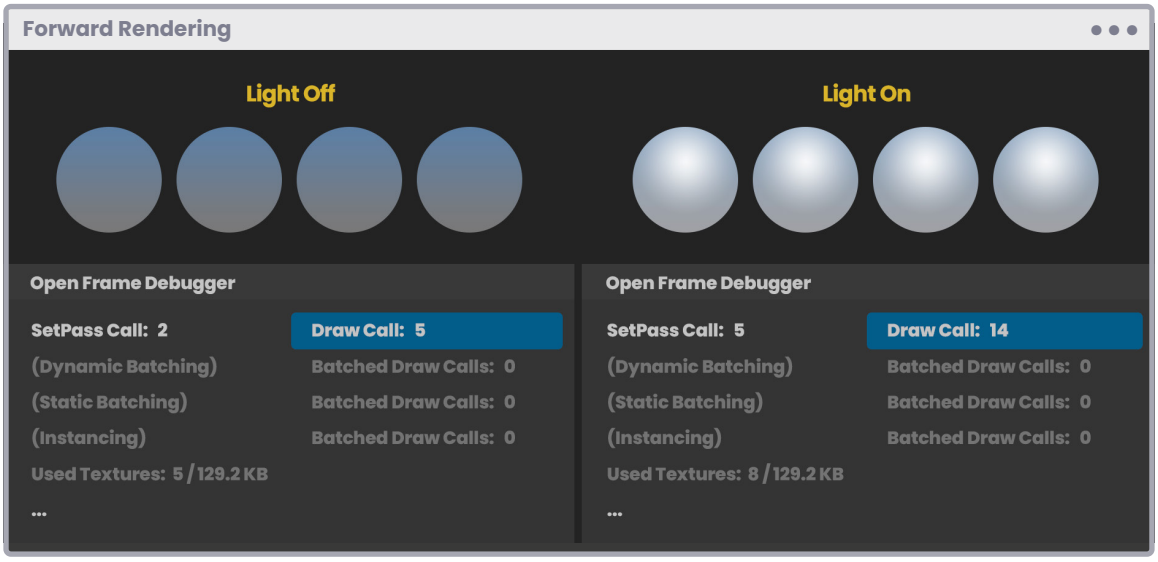
As its name says, the additional pass can process additional lights (Point Light, Spotlight, Area Light) or also shadows that affect the object. What does this mean? If there are two lights in the scene, your object will be influenced by only one of them. However, if you have defined an additional pass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

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

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

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

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



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

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

1.1.4. Deferred Shading.

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

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

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

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

1.1.5. What Render Pipeline should I use?

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

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

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

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

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

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

1.1.6. Matrices and coordinate systems.

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

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

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

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

They are conceptually represented is as follows:

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

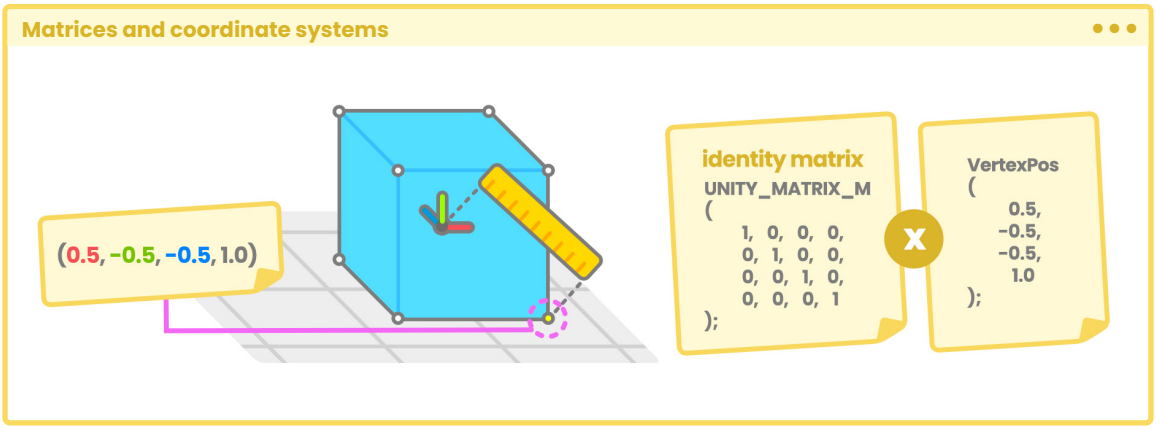
As previously explained in section 1.0.2 talking about vertices, a polygon object has two nodes by default. In Maya, these nodes are known as Transform and Shape, and both are in charge of calculating the position of the vertices in a space called Object-Space, which defines the position of the vertices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

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

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

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

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



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

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

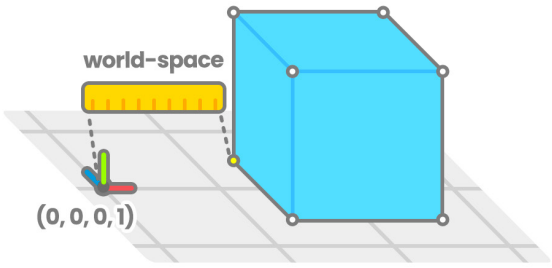
Since the number of columns in the Model Matrix is equal to the number of rows in the position of the vertices, they can be multiplied and the result will be equal to a new matrix of four rows and one column, which would define a new position for the vertices. This multiplication process occurs for all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

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

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

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

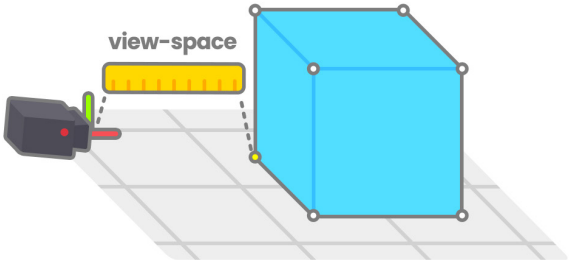
Matrices and coordinate systems



(Fig. 1.1.6b)

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

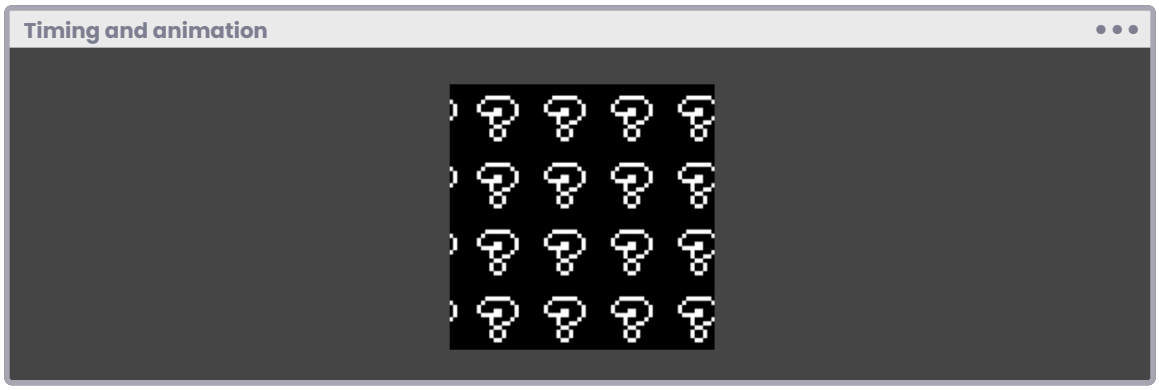
Matrices and coordinate systems



(Fig. 1.1.6c)

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

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



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

Chapter II

Lighting, shadow and surfaces.

The magnitude of the resulting vector will be related to the function of sin .
Taking into consideration vectors A and B mentioned above, you can calculate the Cross Product from a determinant matrix between both vectors.

$$\text{Vector C} = X\,[(A_y * B_z) - (A_z * B_y)]\;Y\,[(A_z * B_x) - (A_x * B_z)]\;Z\,[(A_x * B_y) - (A_y * B_x)]$$

By replacing the values, you get.

$$\text{Vector C} = X\,[(0 * 0) - (0 * 1)]\;Y\,[(0 * 1) - (1 * 0)]\;Z\,[(1 * 1) - (0 * 0)]$$
$$\text{Vector C} = X\,[(0 - 0)]\;Y\,[(0 - 0)]\;Z\,[(1 - 0)]$$
$$\text{Vector C} = (0, 0, 1)$$

In conclusion, the resulting vector is perpendicular to its arguments.
Later, you will use the cross(A_{RG}, B_{RG}) function to calculate the value of a Binormal in a Normals map.

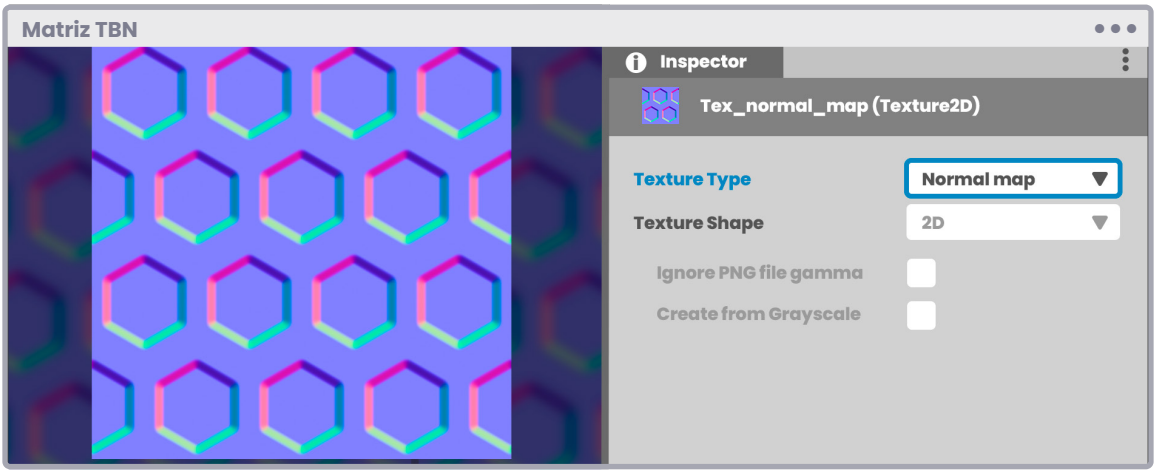
TBN Matrix

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 normal_map = tex2D(_NormalMap, i.uv_normal);
    fixed3 normal_compressed = DXTCompression(normal_map);
    float3x3 TBN_matrix = float3x3
    (
        i.tangent_world.xyz,
        i.binormal_world,
        i.normal_world
    );
    fixed4 normal_color = normalize(mul(normal_compressed, TBN_matrix));
    return fixed4 (normal_color, 1);
}
```

The example above created a three-dimensional vector called **normal_color**. This vector has the result of the Normal Map and the TBN matrix. Finally, you returned the color of the Normals in RGB, and assigned the value "one" to the A channel.

It is important to mention that when you import a Normal Map to Unity, by default, it is configured in **Texture Type Default** in your project.

Before assigning the Normal Map to your material, you must select the texture, go to the inspector, and set it as **Texture Type Normal Map**, otherwise the program might not work correctly.



(Fig. 6.0.3a)

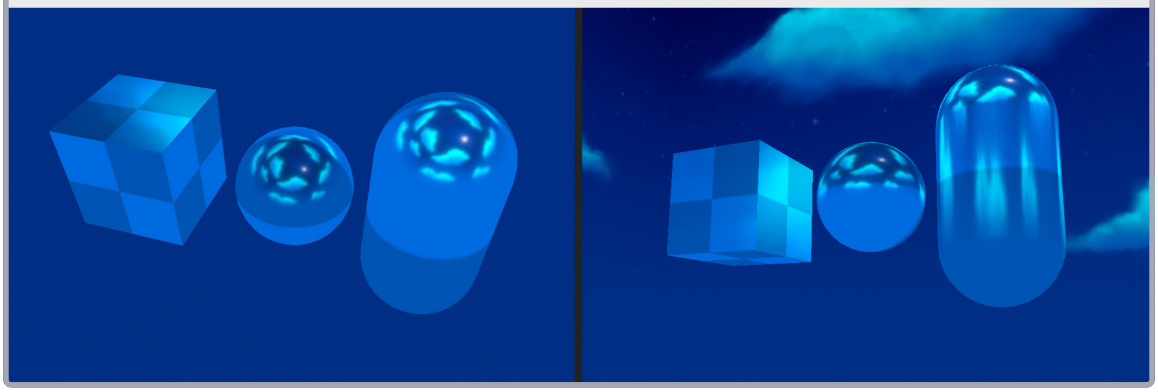
Specular Reflection

```
Shader "USB/USB_specular_reflection"
{
    Properties { … }
    SubShader
    {
        Tags
        {
            "RenderType"="Opaque"
            "LightMode"="ForwardBase"
        }
    }
}
```

7.0.5. Environmental Reflection.

Ambient Reflection occurs similarly to Specular Reflection. Its difference lies in the number of light rays that affect a surface, e.g., in a generic scene, Specular Reflection is only generated by the main light source, while Ambient Reflection is generated by each light ray hitting a surface, including bouncing from all angles.

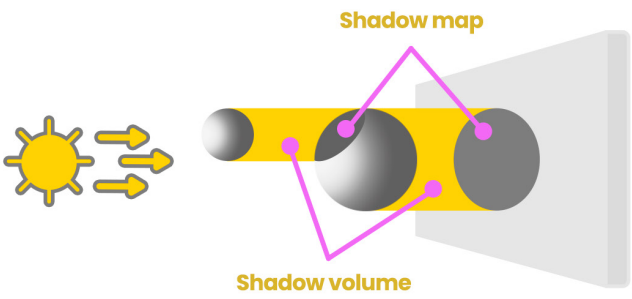
Environmental Reflection



(Fig. 7.0.5a)

Given its nature, calculating this type of reflection in real time uses a lot of GPU power, instead, you can use a **Cubemap** type texture. In Chapter I, section 3.0.6, the **Cube** property was mentioned, which refers precisely to this type of texture.

Shadow Mapping



(Fig. 8.0.1b)

The Shadow Map is a texture; therefore it has UV coordinates and is calculated in two stages: First, the scene is rendered according to the light source viewpoint. During the process, the depth information is extracted from the Z-Buffer and then saved as a texture in the internal memory. In the second, the scene is drawn on the GPU in the usual way according to the camera viewpoint. This is where you must calculate the UV coordinates of the texture saved in memory to generate and apply shadows onto the object you are working with.

8.0.2. Shadow Caster.

Start with generating shadows. For this, create a new Unlit shader and call it **USB_shadow_map.** In the process, you need two passes:

- One to cast shadows (Shadow Caster).
- And another to receive them (Shadow Map).

Therefore, the first thing you must do is include a second pass, which will be responsible for the shadow projection.

Shadow Caster

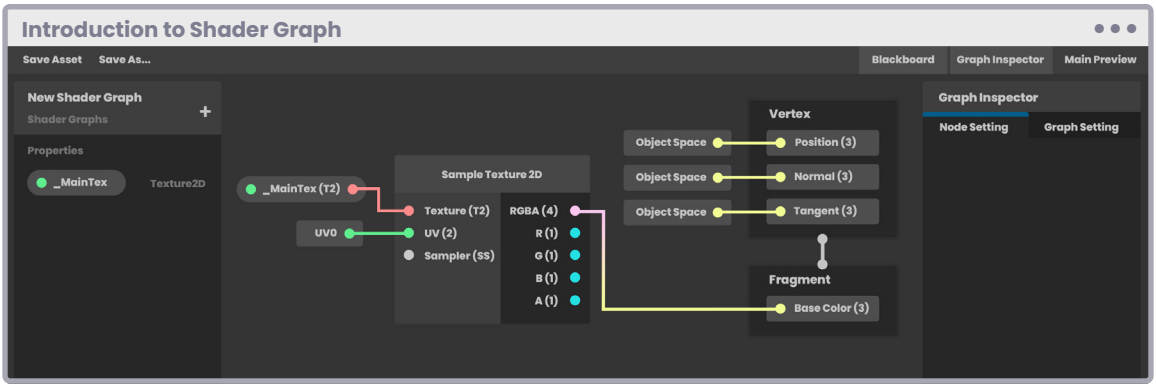
```
Shader "USB/USB_shadow_map"
{
    Properties { … }
    SubShader
```

Continued on the next page.

Another thing is that it is very likely that shaders created with this interface do not compile correctly in other versions. This is because new features are added in each update, sometimes affecting the node set, even more so if you are using custom functions.

So, is Shader Graph a good tool for developing shaders? The answer is yes, even more so for artists.

For those who have worked with 3D software such as Maya or Blender; Shader Graph will be very useful since it uses a system of nodes very similar to **Hypershader** and **Shader Editor** which allows for more intuitive shader creation.



(Fig. 9.0.1b)

Before introducing this topic, be aware that the Shader Graph interface has functional variations according to its version, e.g., at the time of writing this book, its most up-to-date version is **12.0.0.**

If you create a node with this version, you can see that the Vertex and Fragment Shader Stage appear separate and work independently. However, if you go to version **8.3.1,** both stages are merged within a node called **Master**, which refers to the final shader output color.

As you mentioned before, it is very possible that the shaders created in this interface do not compile in all its versions, in fact, if you create a shader in version 8.3.1 and update to version 12.0.0, there are probably functional changes that prevent its compilation.

Call the .hlsl file **CustomLight** and start adding your function as follows:

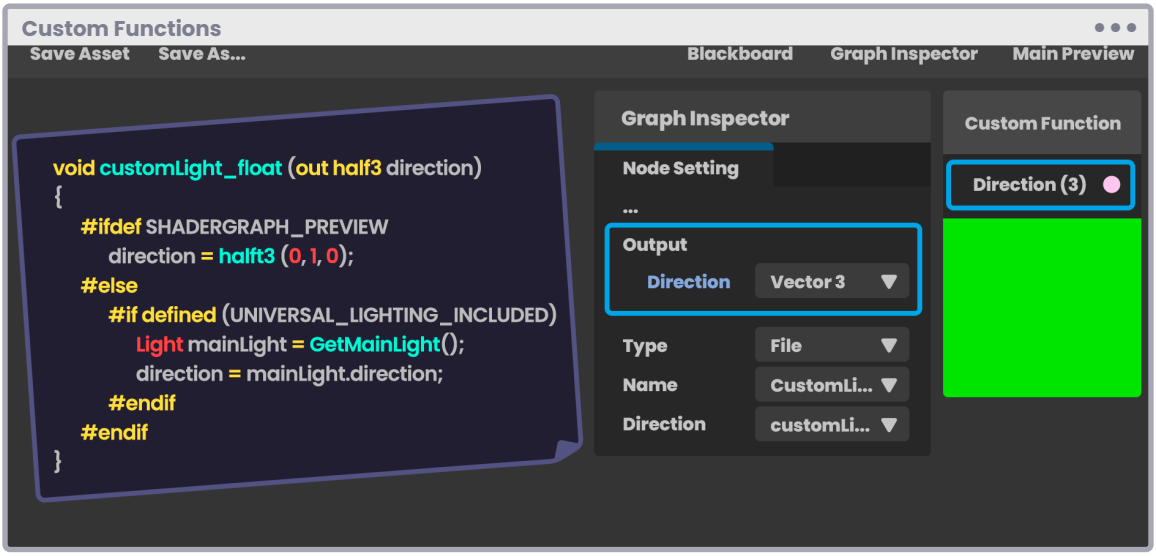
Custom Functions

```
// 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, you can deduce that if the preview in Shader Graph is enabled (SHADERGRAPH_PREVIEW), you will project lighting for ninety degrees on the $\mathbf{Y_{AX}}$. Otherwise, if **Universal RP** has been defined, then the output **direction** will be the same as the direction of the main light, that is, to the directional light you have in the scene.

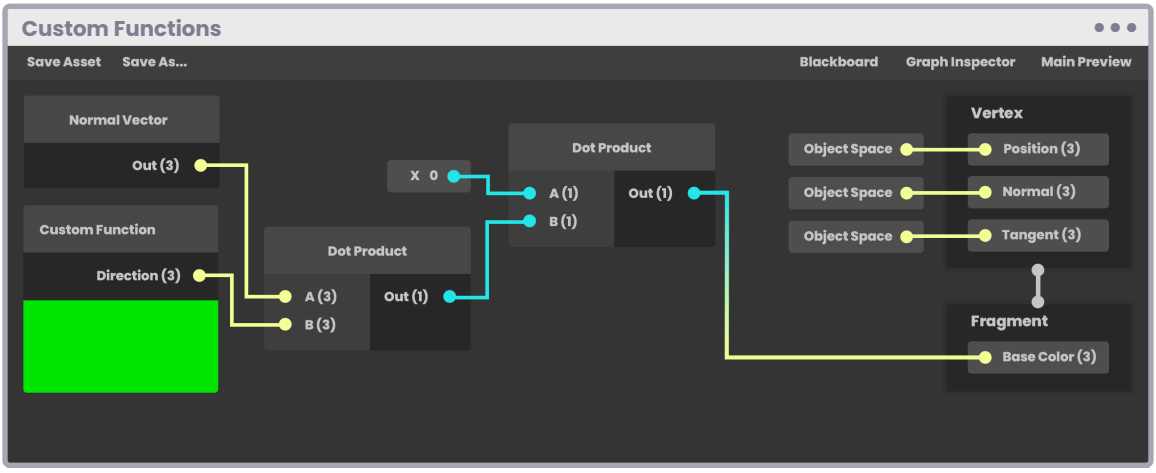
Unlike the previous case, the **CustomLight** function has no inputs; therefore, a three-dimensional vector will have to be added as output later in the node configuration. On the other hand, it's worth noting that such output is of **half3** type. Consequently, the accuracy of the node will be equal to a 16-bit value because, by default, it is configured as **inherit**.

Next, you must make sure to drag or select the **.hlsl** file in the **Source** box located in the Graph Inspector **Node Settings** window. You must be sure to use the same name of the function in the **Name** box; that is, CustomLight, otherwise it could generate compilation errors.



(Fig. 9.0.7c. The Custom Function node is green due to the values of the direction vector)

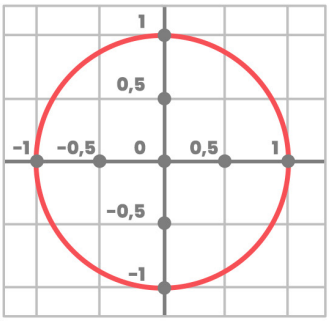
Since the enclosed variable **_WorldSpaceLightPos** has the light position, you can use the node to replicate the same behavior; in fact, the operation **mainLight.direction** is the same as the variable **_MainLightPosition** included. You can check this by going to the project's **Lighting.hlsl** file.



(Fig. 9.0.7d. $D = max(0, N \cdot L)$)

You can generate the same behavior through nodes in the Shader Graph as shown in Figure 9.0.7d, following the diffusion scheme seen in section 7.0.3.

Sphere Tracing



(Fig. 11.0.0b. With "Z" equal to 0.0f)

Which is the same as saying,

$$|| X || - 1 = 0$$

Therefore,

Sphere Tracing

```
float sphereSDF(float3 p, float radius)
{
    float sphere = length(p) - radius;
    return sphere;
}
```

An implicit surface is defined by a function that, given a point in space, indicates that said point is inside or outside the surface.

A ray travels from the camera through a pixel until it hits a surface to achieve the objective. This concept is called Ray Casting, which is the process of finding the closest object along the ray, hence the name "sphere casting."

```
    // pass the values to the function
    float t = sphereCasting(ray_origin, ray_direction);

    // calculate the spacial point of the plane
    float3 p = ray_origin + ray_direction * t;

    if (i.hitPos > _Edge)
        discard;

    return col;
}
```

Looking at the previous example, you have stored the plane's distance in respect to the camera in the "t" variable and then stored each point in the variable "p." It will be necessary to project the SDF plane onto the front face of the sphere. Consequently, you will have to disable Culling from the SubShader.

Implementing functions with Sphere Tracing

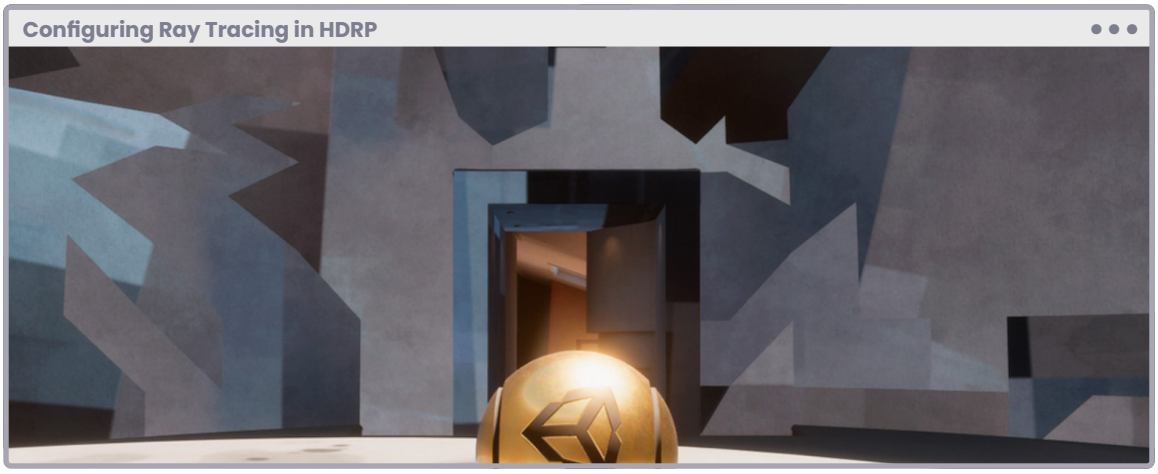
```
SubShader
{
    …
    // project both faces of the sphere
    Cull Off
    …
    Pass
    {
        …
    }
}
```

Using the **SV_isFrontFace** semantic, you can project the pixels of the sphere on its back face, and the plane on the front face.

High Definition RP is characterized by its quality rendering and compatibility with high-end platforms, i.e., **PC**, **PlayStation 4**, or **Xbox One** (onwards).

It also supports **DirectX 11** and later versions and **Shader Model 5.0**, which introduces Compute Shaders for graphics acceleration.

When opening a scene, it is very common for some textures to look like those in Figure 12.0.1c. This is due to a configuration error in the generated **Lightmaps** when creating the project.



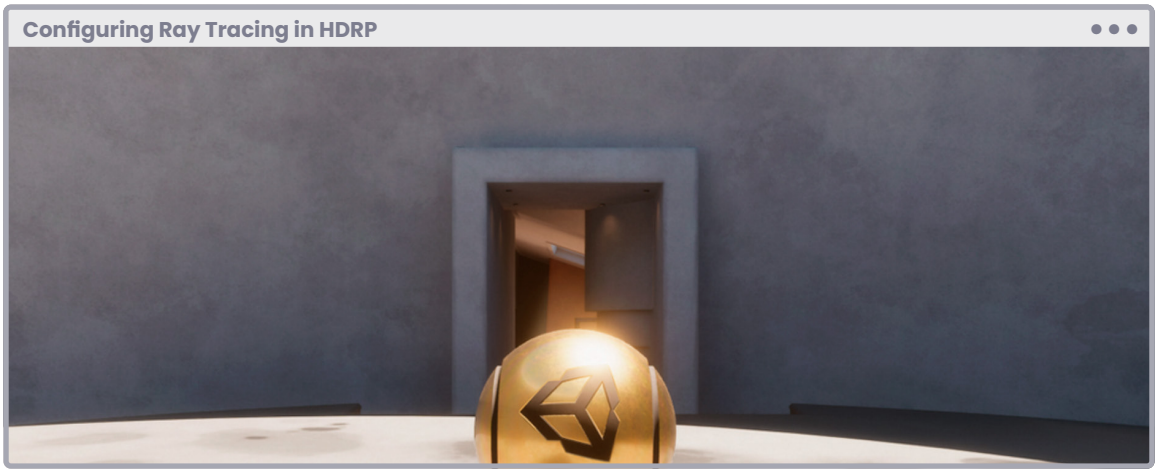
(Fig. 12.0.1c. The walls have errors in their lightmaps)

To solve this problem, you must pay attention to the configuration of the objects. If you select any object, e.g., **FR_SectionA_01_LOD0** (wall), you will notice that it has been marked as "static" from the Unity Inspector.

So you must go to the menu Windows / Rendering / Lighting and perform the following operation:

1. Press the dropdown belonging to the Generate Lighting button and select Clear Baked Data. By doing this action, all the Lightmaps will be eliminated and you will see the default lighting.
2. Next, you must press the Generate Lighting button.

The process may take a few minutes depending on your computer's capacity. However, the textures and lighting will finally be displayed correctly.



(Fig. 12.0.1d. The lightmaps have been corrected)

Note that the global illumination and other properties, such as ambient occlusion, are being **burned** onto each texture. Therefore, if you change the position of an object, its lighting properties will keep their shape and will not be recalculated.

The only way to perform this process in real-time is by activating Ray Tracing. For this, you will have to consider several configurations, including DirectX 12 (DX12). The whole process can be summarized in three main steps;

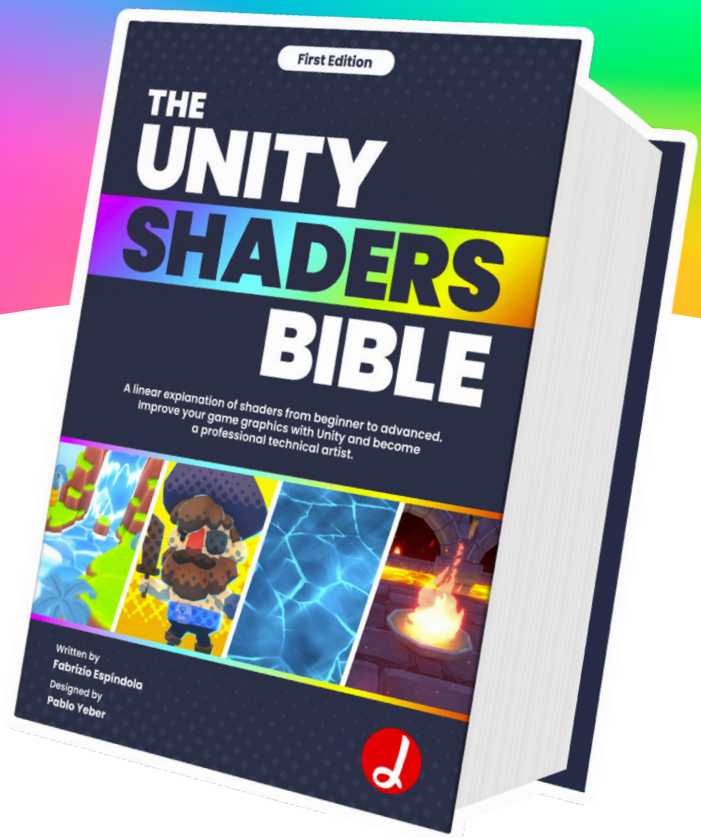
1. Render Pipeline Asset.
2. Project Settings.
3. DirectX 12.

Start by going to the menu Windows / Panels / Project Settings, paying attention to the following categories:

- Quality.
- Graphics.
- HDRP Default Settings.

Note that Unity creates a different rendering configuration for each quality level in the project, e.g., our project has three default quality levels, which can be found in the **Quality** tab.

- High Quality.
- Medium Quality.
- Low Quality.



Why not get the full book?

The Unity Shaders Bible book has a **5 star** rating

It has over **8K** readers so far!

Will you join us to learn about Shaders?

With +380 pages of information applicable to game development.

Updates are free forever!

Buy on Gumroad

Jettelly Books!

The place where we share our knowledge about development and video games with you.

Go to Books!