

**Learn. Create. Master.**

**First Edition**

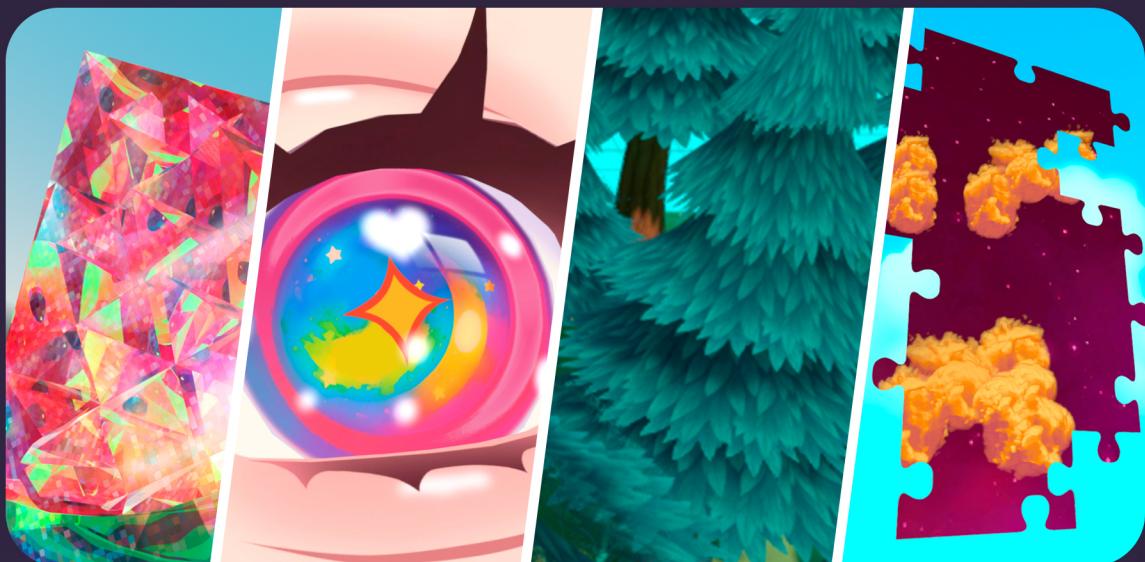
# The Godot Shaders Bible.

Everything you need to know about shaders to enhance your game's visuals.

# IndieDev

# GameDev

# 3DCG



**Fabrizio Espíndola.**



jettelly®

# **The Godot Shaders Bible.**

Everything you need to know about shaders to enhance your game's visuals.

**Fabrizio Espíndola**

#IndieDev

#GameDev

#3DCG

First Edition 2025

The Godot Shaders Bible, version 0.1.1.

Published by Jettelly Inc. ® All rights reserved. [jettelly.com](http://jettelly.com)

77 Bloor St. West, Suite #663, Toronto ON M5S 1M2, Canada.

# **Credits.**

## **Author.**

Fabrizio Espíndola

## **Design.**

Pablo Yeber

## **Illustration.**

Hanbit Park

## **Head Editor.**

Ewan Lee



# Content.

<b>Introduction.</b>	<b>6</b>
About the Author.....	6
Errata.....	6
Piracy.....	7
<b>1. Introduction to Mesh Composition.</b>	<b>8</b>
1.1 Properties .....	9
1.2 Mesh Data.....	12
1.3 Introduction to Spaces.....	21
1.4 Configuring a Unique Object.....	25
1.5 Rendering Stages and Pipeline.....	32
1.6 Working with World Space Coordinates.....	38
1.7 Introduction to Tangent Space.....	48
1.8 Implementing Tangent Space in Our Shader.....	51
1.9 Linear Interpolation Between Colors.....	64
1.10 Introduction to Matrices.....	71
1.11 Built-in Matrices.....	74
1.12 Implementing Custom Matrices.....	83
<b>2. Lighting and Rendering.</b>	<b>92</b>
2.1 Material Properties.....	93
2.2 Introduction to the Lambertian Model.....	96
2.3 Implementing the Lambertian Model.....	99
2.4 Vector: Points vs. Directions.....	111
2.5 Introduction to the Blinn-Phong Model.....	112
2.6 Implementing the Blinn-Phong Model.....	115
2.7 From Linear Lighting to sRGB.....	122
2.8 Introduction to the Rim (Fresnel) Effect.....	127
2.9 Implementing the Rim Effect.....	130
2.10 Introduction to the Anisotropic Effect.....	138
2.11 Implementing the Ashikhmin-Shirley Model.....	146
2.12 Hemispheric Shading.....	162
2.13 ShaderInclude Implementation.....	170



2.14	Normal Map Implementation.	175
<b>3.</b>	<b>Procedural Shapes and Vertex Animation.</b>	<b>186</b>
3.1	Drawing with Functions and Inequalities.	187
3.2	Animation and Procedural Modification.	200
3.3	Implementation of the Procedural Character in GDSL.	205
3.4	Vertex Transformation and Animation.	228
3.5	Adding a Specular Effect to the UI.	242
3.6	Introduction to Rotations.	254
3.7	Implementing a Rotation Matrix.	258
3.8	Quaternion Implementation.	267
<b>4.</b>	<b>Advanced VFX and Post-Processing.</b>	<b>282</b>
4.1	Post-Processing and the Game Boy Effect.	284
4.2	Porting an Effect from Shadertoy.	299
4.3	Transparencies and Distance.	303
4.4	Ray Marching and 3D Textures.	319
4.5	Introduction to Stencil Buffer.	337
<b>Special Thanks.</b>		<b>354</b>

## About the Author.

Fabrizio Espindola is a renowned Technical Artist and developer from Santiago, Chile, with extensive experience in Computer Graphics and a deep passion for creating shaders in environments like Unity and, more recently, Godot.

He is the Author of "The Unity Shaders Bible" and the books "Visualizing Equations VOL. 1 & 2," essential resources for developers and Technical Artists who want to master the art of procedural shapes. His practical and detailed approach has helped over 15,000 professionals worldwide enhance their skills and understanding in developing special effects for video games.

With more than 17,000 followers on LinkedIn, Fabrizio is an influential figure in the game development community. He regularly shares knowledge, tutorials, assets, and updates of the latest trends in real-time graphics. His ability to simplify complex concepts and his commitment to education have made him a reference for many developers.

Since 2022, Fabrizio has been working at Rovio, the creators of Angry Birds, as a Senior Technical Artist. Previously, he collaborated with renowned companies like DeNA and Osmo, where he gained extensive experience in creating and optimizing assets for various applications. His passion for teaching has led him to write books, tutorials, and manuals, sharing his knowledge with the independent developer community.

For more information about his work and projects, visit [jettelly.com](http://jettelly.com)

## Errata.

While writing this book, we have taken precautions to ensure the accuracy of its content. Nevertheless, you must remember that we are human beings, and it is highly possible that some points may not be well-explained or there may be errors in spelling or grammar.

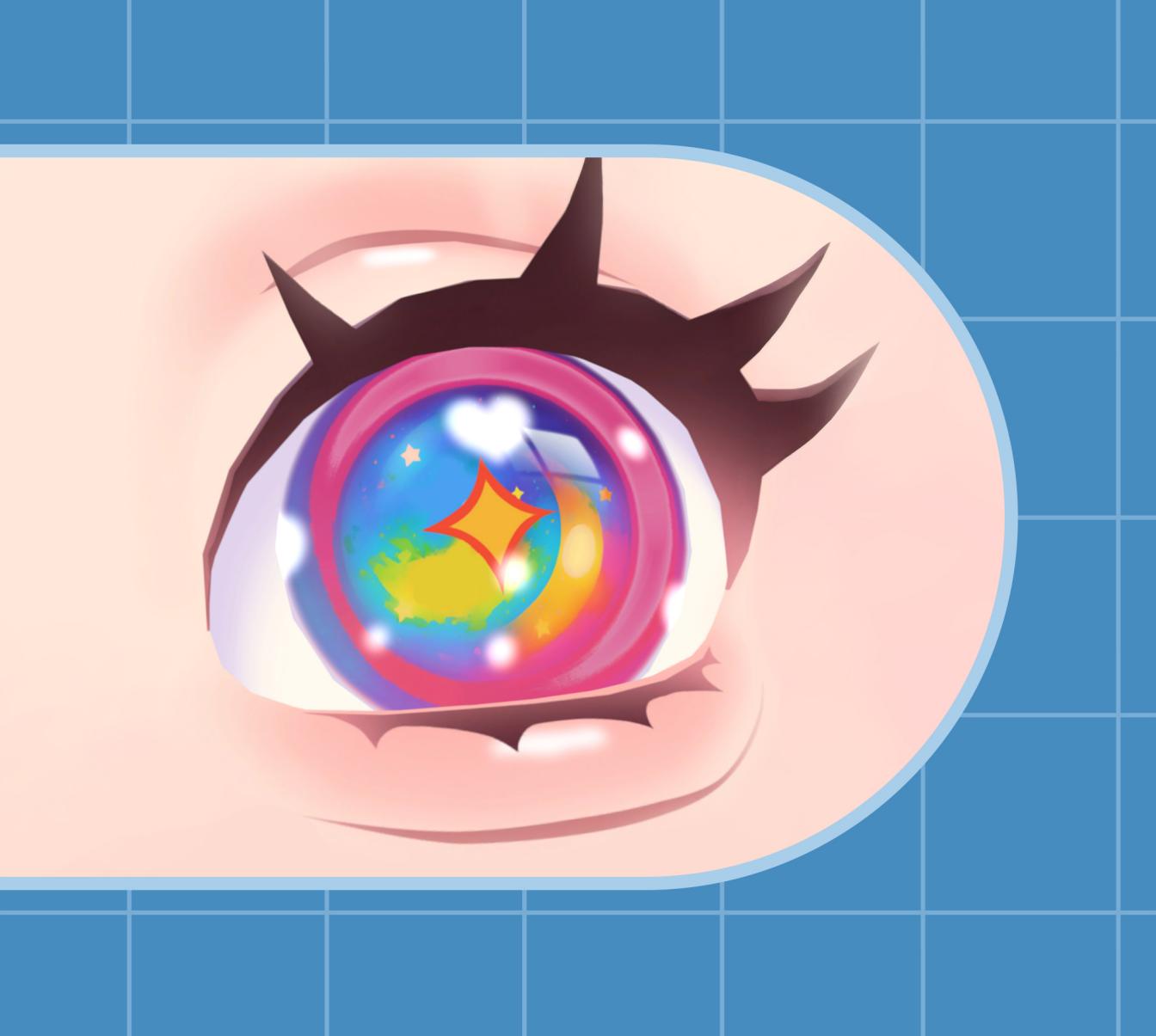
If you come across a conceptual error, a code mistake, or any other issue, we appreciate you sending a message to [contact@jettelly.com](mailto:contact@jettelly.com) with the subject line "GSB Errata." By doing so, you will be helping other readers reduce their frustration and improving each subsequent version of this book in future updates.

Furthermore, if you have any suggestions regarding sections that could be of interest to future readers, please do not hesitate to send us an email. We would be delighted to include that information in upcoming editions.

## Piracy.

Before copying, reproducing, or distributing this material without our consent, it is important to remember that Jettelly Inc. is an independent and self-funded studio. Any illegal practices could negatively impact the integrity of our work.

This book is protected by copyright, and we will take the protection of our licenses very seriously. If you come across this on a platform other than [jettelly.com](http://jettelly.com) or discover an illegal copy, we sincerely appreciate it if you contact us via email at [contact@jettelly.com](mailto:contact@jettelly.com) (and attach the link if possible), so that we can seek a solution. We greatly appreciate your cooperation and support. All rights reserved.



# Chapter 1

# Introduction to Mesh Composition.

In this chapter, you'll take your first steps into shader development — an essential skill for crafting everything you see on a computer screen. But what exactly is a shader? In Godot, a shader is a small program with the **.gdshader** extension that runs on the GPU. It processes every visible pixel in the scene, allowing you to draw and visually modify objects using coordinates, colors, textures, and other properties derived from the object's geometry.

Shaders take advantage of the GPU's parallel architecture. A modern GPU contains thousands of small cores that can execute operations simultaneously, making it ideal for performance-intensive tasks such as rendering, visual effects, and dynamic lighting.

Before you start writing your own shaders, it's important to understand the building blocks that make them work. You'll explore the internal and external properties you can control through code. This foundational knowledge will help you understand how shaders operate, how they interact with the scene, and how you can harness them to create unique and compelling visual effects.

## 1.1 Properties.

Before you dive into shaders and visual effects, it's important to understand what a mesh is in Godot. According to the official Godot documentation:

“

Mesh is a type of Resource that contains vertex arrays-based geometry, divided in surfaces. Each surface contains a completely separate array and a material used to draw it. Design wise, a mesh with multiple surfaces is preferred to a single surface, because objects created in 3D editing software commonly contain multiple materials.

[https://docs.godotengine.org/en/stable/classes/class\\_mesh.html](https://docs.godotengine.org/en/stable/classes/class_mesh.html)

”

This definition is consistent with those found in other 3D tools, such as **Autodesk Maya** or **Blender**, as they share fundamental principles in defining and manipulating 3D geometry. In the context of shader and visual effects development, **meshes** serve as the foundation upon which materials, textures, and other resources are applied — bringing life and personality to the models displayed on screen.

Analyzing the previous description, we can identify two key components that deserve closer attention: vertex matrices and materials, with the latter being essential for rendering a mesh. In computer graphics, both numerical data — such as geometry, coordinates, and attributes — and visual resources that enable objects to be displayed on screen must be considered. Understanding this duality between data (geometry, coordinates, attributes) and visual representation (materials, textures, shaders) is fundamental to your role as a technical artist.

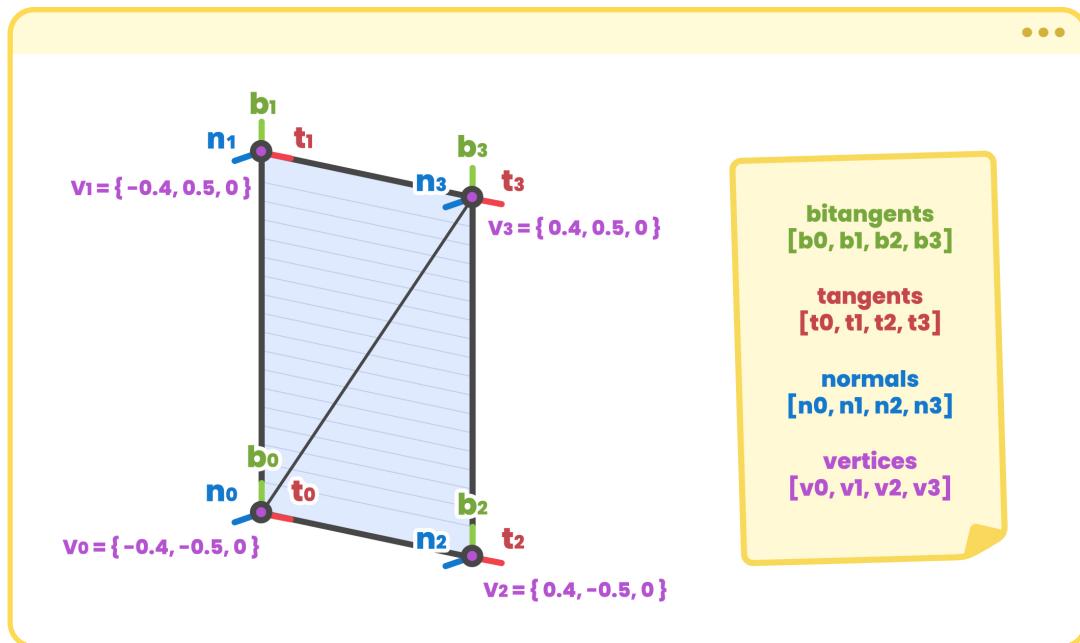
When we talk about vertex arrays, we refer to the ordered collection of points that define an object's shape. A mesh does more than just store these points — it also integrates various properties that, together, form the polygonal faces (triangles) that shape the model. These properties include:

- Vertices.
- Bitangents (or Binormals).
- Normals.
- Tangents.
- UV Coordinates.
- Vertex Color.

To understand the importance of these properties, imagine holding a sheet of paper. If someone asked you to draw on the “front” side, how would you determine which side that is if both look identical? This is where **normals** come into play — they indicate the direction a surface is facing, helping define which side will be visible when viewing the object. In 3D applications like Maya or Blender, the software typically renders only the faces whose normals point toward the observer (camera), ignoring or not rendering those that are flipped. This optimization technique improves performance by reducing the number of unnecessary calculations during rendering.

Tangents and bitangents are also important, though they play a slightly lesser role than normals in basic visual effects. Continuing with the sheet of paper analogy, if the sheet has four vertices and four normals, you can also define four tangents and four bitangents associated with them.

Together, normals, tangents, and bitangents form a local coordinate system for the model's surface, which is essential for applying normal maps and achieving advanced lighting effects. This foundational concept will be crucial when designing, optimizing, and refining your shaders and visual effects, as it helps you understand how geometry and graphical resources interact in the creation of 3D environments and characters.



(1.1.a Conceptualizing Mesh properties on a notebook page)

The tangent is typically aligned with the U-axis of the UV coordinates, while the bitangent is oriented along the V-axis. Together, they define a local reference system at each vertex, known as tangent space, which is essential for computing advanced effects such as normal mapping, reflection simulation, and other complex visual effects. This space allows you to interpret additional texture details relative to the surface of the mesh, giving you greater control over the final visual outcome.

Later in this book, we will dedicate an entire section to creating visual effects using tangent space, UV coordinates, and vertex color. For now, it's important to highlight that for a mesh's properties to be visible in the engine, it must be linked to a rendering process and one or more materials.

In Godot, the API responsible for processing a mesh's geometry and displaying it in the scene is the **RenderingServer**. Without this API, the mesh would exist solely as a data resource in memory, with no visual representation or interaction with the camera or lights. Its representation in the engine takes shape through a node called **VisualInstance3D**, which serves as the entry point for manipulating 3D objects. One of its most commonly used child nodes is **MeshInstance3D**, which internally manages the visual interactions between the mesh and

its material. This allows you to work more intuitively, without having to deal directly with the technical details of the rendering process.

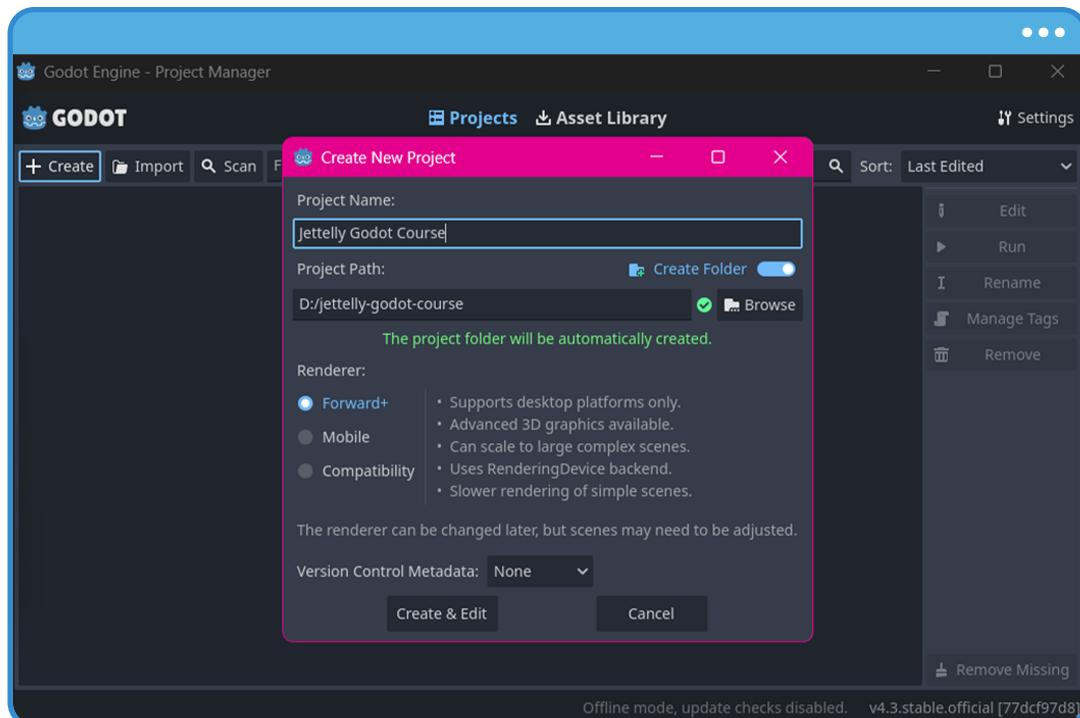
From the **MeshInstance3D** node, you can access both the mesh and its material directly from the **Inspector**. In Godot, a material is a resource that defines how an object's surface is rendered, determining its interaction with light, shadows, textures, and other visual effects. At its core, a material contains a shader – a program that runs on the GPU. Godot uses a simplified variant of **GLSL** (OpenGL Shading Language) to define these shaders, making them easier to edit and understand.

Through the shader, you can access the mesh's properties. For instance, the predefined variable **VERTEX** allows you to manipulate vertex positions, enabling direct modifications to the mesh within the shader. Similarly, variables such as **NORMAL**, **TANGENT**, and **BITANGENT** provide control over normals, tangents, and bitangents, respectively. Understanding how these variables work requires visualizing the numerical values stored in the Vertex Data. By doing so, you can better grasp how different properties interact to produce the final visual output on screen.

## 1.2 Mesh Data.

In this section, you'll carry out a hands-on exercise to understand how vertex lists, and their properties are grouped, and then read by the GPU through a shader to visually render geometry on-screen. To do this, you'll create a **QuadMesh** in **.obj** format using a simple text editor (such as Notepad).

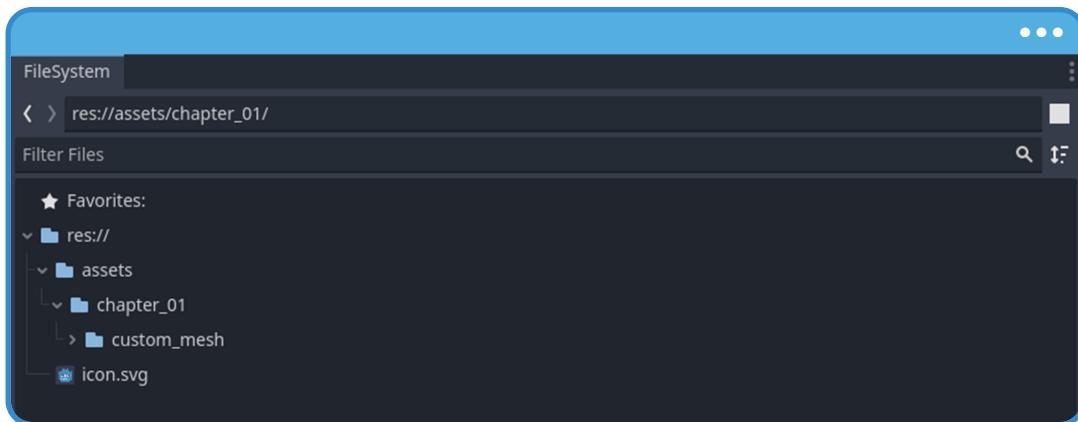
First, create a new Godot project and name it **Jettelly Godot Course**. Make sure to select **Forward+** as the **Renderer**. Later, we will explore the architecture of a Rendering Pipeline and the differences between various rendering techniques. For now, assume that you'll be using this project throughout the entire learning process on a desktop computer. This approach ensures a consistent reference setup as you progress.



(1.2.a Creating a new project in Godot 4.3)

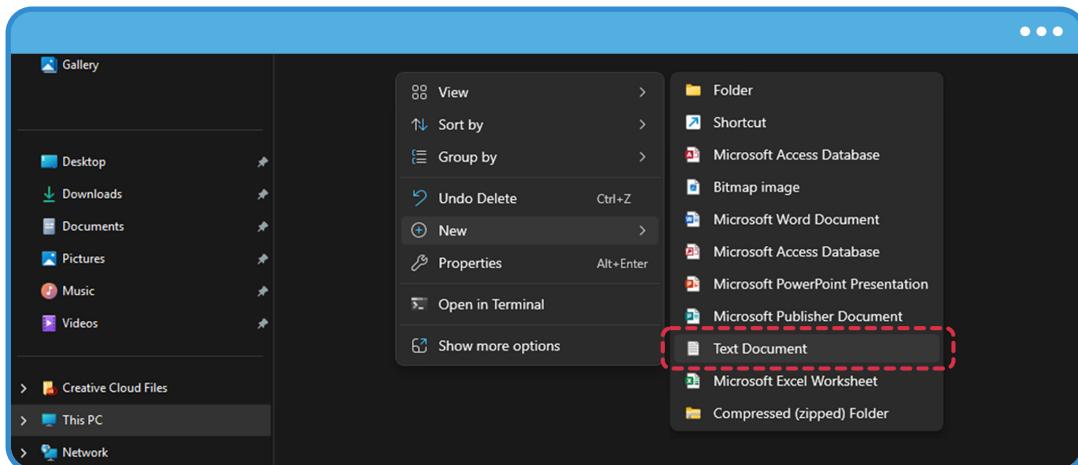
Once your project is open, start by organizing your resources. Navigate to the **FileSystem** panel and create your first folder, naming it **assets**. From this point on, all generated files will be stored in subfolders within **assets**, providing a cleaner workspace structure and making version control easier. For instance, if you are using Git, you can link this folder and commit changes exclusively to its contents, preventing accidental modifications to the project's main configuration.

Inside **assets**, create a new subfolder called **chapter\_01**, and then, within it, create another folder named **custom\_mesh**. This is where you'll store the files needed for the upcoming exercises



(1.2.b Initial project structure)

As mentioned, you'll create the **QuadMesh** for this exercise using a text editor. The process is straightforward: navigate to the **custom\_mesh** folder in your operating system and create a new text file. On Windows, you can do this by right-clicking the destination folder and selecting **New > Text Document**, as shown below:



(1.2.c Creating a text document in Windows)

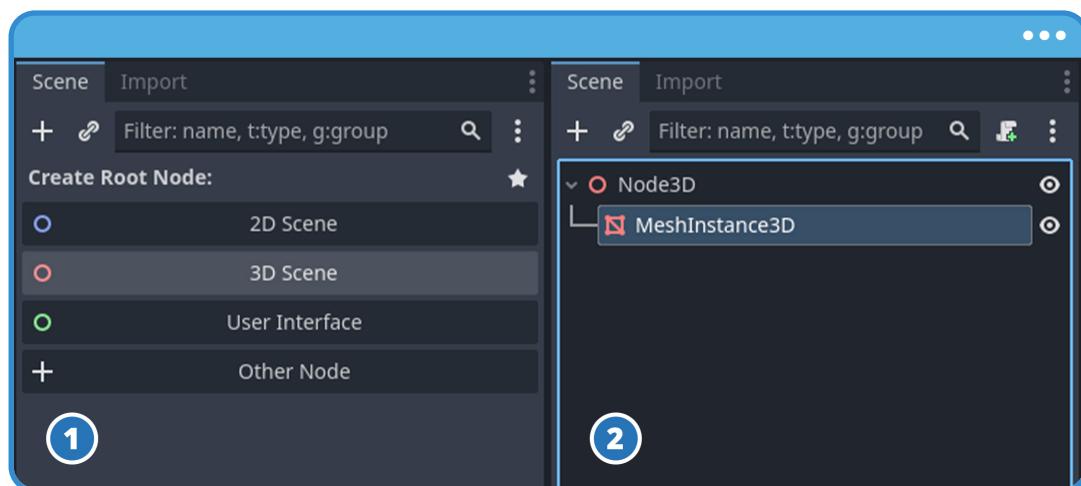
Name this document **custom\_quad**, and change its extension from **.txt** to **.obj**. This **.obj** format is a standard for representing 3D geometry. However, after making this change and returning to Godot, you might notice a warning icon (an "X") next to the file. This indicates that the file is "corrupt" or incomplete — which is expected, as we haven't yet added the necessary data to define the QuadMesh's geometry. In the next steps, you'll define all the required information so that Godot can correctly interpret and render the object.

When exporting 3D models in **.obj** format, an additional file with the **.mtl** (Material Template Library) extension is also generated. This file contains information about the object's lighting properties. For now, we won't focus on this file, as our main priority is the geometry itself. We'll let Godot assign its default lighting values. Later in this book, you'll learn how to work with materials and explore how this library affects the final appearance of your geometry.

A basic **.obj** file structure typically includes:

- Definitions of vertex positions in local space.
- Definitions of UV coordinates in uv space.
- Definitions of normals in local space.
- Smoothing angle parameters.
- Model face groups.
- Model face definitions.

To properly visualize the **QuadMesh** in Godot, you'll need to define this information. This ensures that the GPU can project the geometry onto the screen. Before you begin, create a 3D scene by choosing a **Node3D** as the parent node. Then, add a **MeshInstance3D** node as its child. This node will allow you to assign the custom mesh you just created and render it within the scene.



(1.2.d Selecting a 3D scene)

According to the official documentation:

“

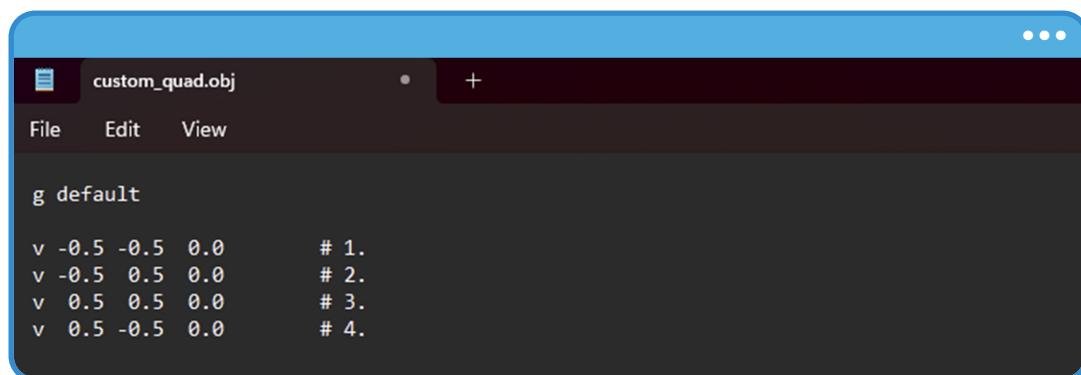
**MeshInstance3D** is a node that takes a **Mesh** resource and adds it to the current scene by creating an instance of it. This is the class most often used to render 3D geometry and can be used to instance a single Mesh in many places.

[https://docs.godotengine.org/en/stable/classes/class\\_meshinstance3d.html](https://docs.godotengine.org/en/stable/classes/class_meshinstance3d.html)

”

If you select the **MeshInstance3D** in the **Scene** window and navigate to the **Inspector**, you’ll notice that its first property is **Mesh**, which allows you to assign a mesh resource directly. Attempting to do this now with your **custom\_quad.obj** will not work, because you have not yet defined the object’s data, and it remains marked as corrupt. To fix this, open the **.obj** file in your preferred text editor and add the necessary properties.

Since Godot uses meters by default, you’ll use a unit system where a range of -0.5 to 0.5 represents a 1x1 meter **QuadMesh**. This approach makes it easier for you to visualize and adjust the geometry as needed.



```

custom_quad.obj
File Edit View

g default

v -0.5 -0.5 0.0      # 1.
v -0.5 0.5 0.0       # 2.
v 0.5 0.5 0.0        # 3.
v 0.5 -0.5 0.0       # 4.

```

(1.2.e Defining the vertices for a **QuadMesh**)

In the Figure 1.2.e, four vertices have been defined, one at each corner of the **QuadMesh**, following a clockwise order for convenience. However, when defining the faces later, we will use the reverse order to align with Godot’s conventions, where the **+z** axis faces the camera. In **.obj** files, vertices are defined using the letter **v**.

```

Vertices : VERTEX
{
    v -0.5  -0.5  0.0 #1.
    v -0.5   0.5  0.0 #2.
    v  0.5   0.5  0.0 #3.
    v  0.5  -0.5  0.0 #4.
}

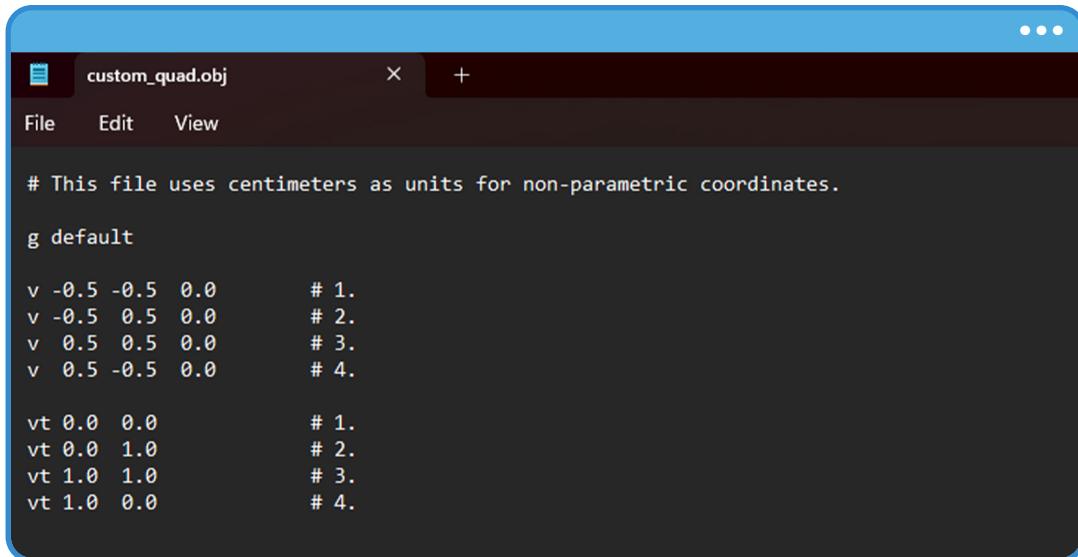
```

(1.2.f The QuadMesh's vertices defined in a clockwise direction)

The **VERTEX** variable in the shader will give you access to these values at runtime, allowing you to eventually modify their positions or manipulate the geometry in more complex ways. While this example only involves four vertices, a realistic model could contain thousands or even millions of them, organized into different surfaces and matrices. For instance, imagine you have a 3D warrior divided into two separate pieces — body and sword — the model would likely have one matrix for the entire body and another for the sword.

Next, let's define the UV coordinates, represented as **vt** in **.obj** files. These coordinates, ranging from 0.0 to 1.0, allow you to map textures onto a surface. In Section 1.3, we will explore this type of space in greater detail. For now, assign a UV coordinate to each vertex of the **QuadMesh** using the following values:

- Vertex #1 (bottom-left): [0.0, 0.0].
- Vertex #2 (top-left): [0.0, 1.0].
- Vertex #3 (top-right): [1.0, 1.0].
- Vertex #4 (bottom-right): [1.0, 0.0].



```
# This file uses centimeters as units for non-parametric coordinates.

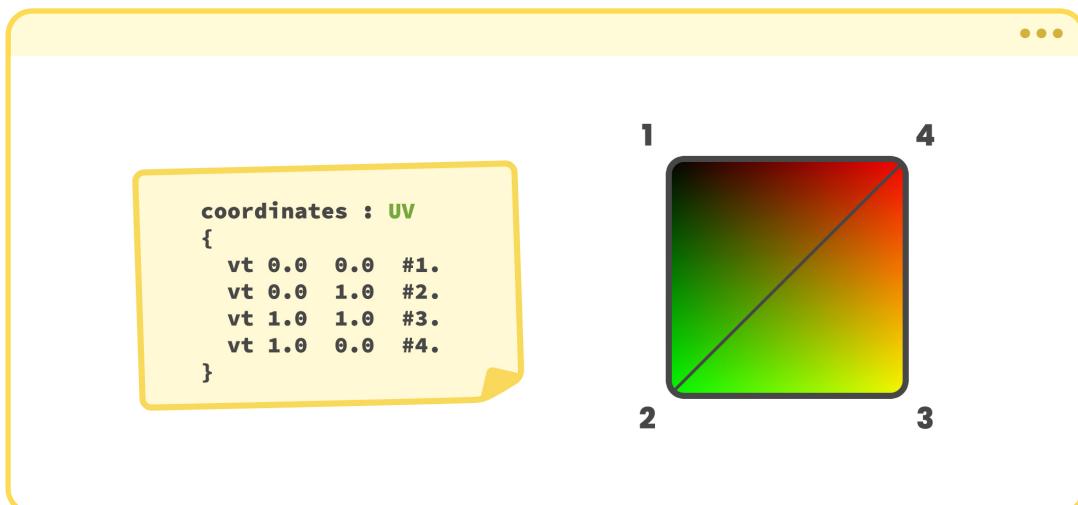
g default

v -0.5 -0.5 0.0      # 1.
v -0.5 0.5 0.0       # 2.
v 0.5 0.5 0.0        # 3.
v 0.5 -0.5 0.0       # 4.

vt 0.0 0.0            # 1.
vt 0.0 1.0            # 2.
vt 1.0 1.0            # 3.
vt 1.0 0.0            # 4.
```

(1.2.g Defining UV Coordinates on a **QuadMesh**)

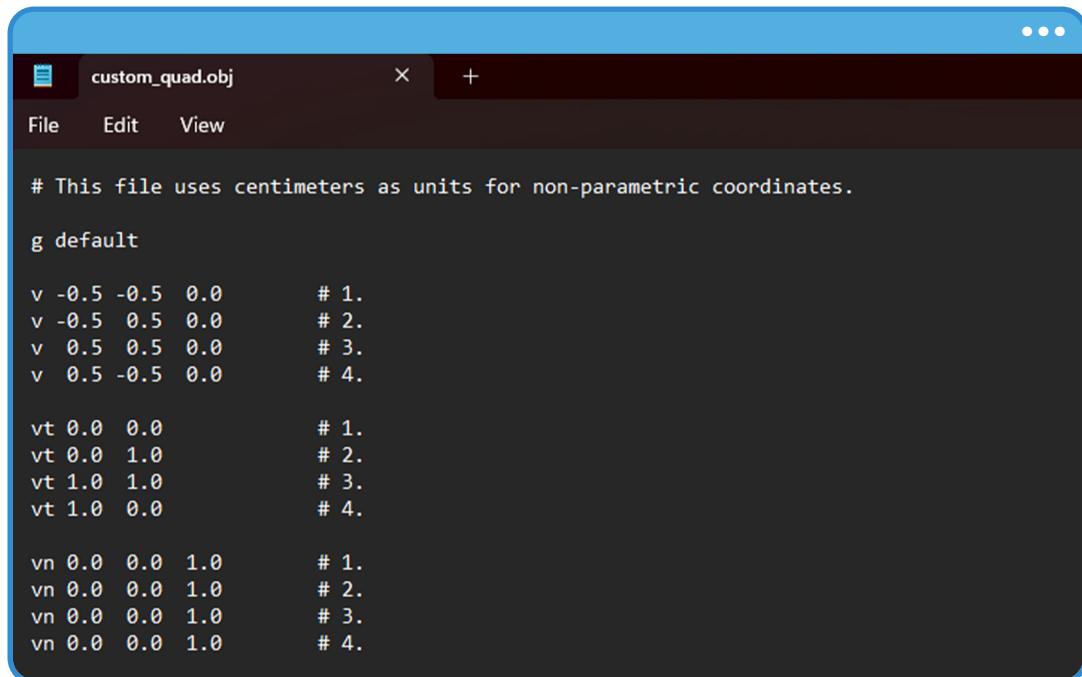
These coordinates are essential for projecting textures onto the object's surface. In the shader, the built-in variable **UV** lets you access and manipulate these values, making it particularly useful for creating visual effects, shifting textures across the surface, or even implementing more complex mapping techniques.



(1.2.h Visualizing the UV coordinates)

The next step is to define the normals, represented in **.obj** files by **vn**. Normals are vectors that indicate the orientation of the surface and are essential for calculating lighting. In this

case, we will define the normals so that they face toward the viewer. This means their XY components will be 0.0, while the Z component will be 1.0, pointing forward.



```
# This file uses centimeters as units for non-parametric coordinates.

g default

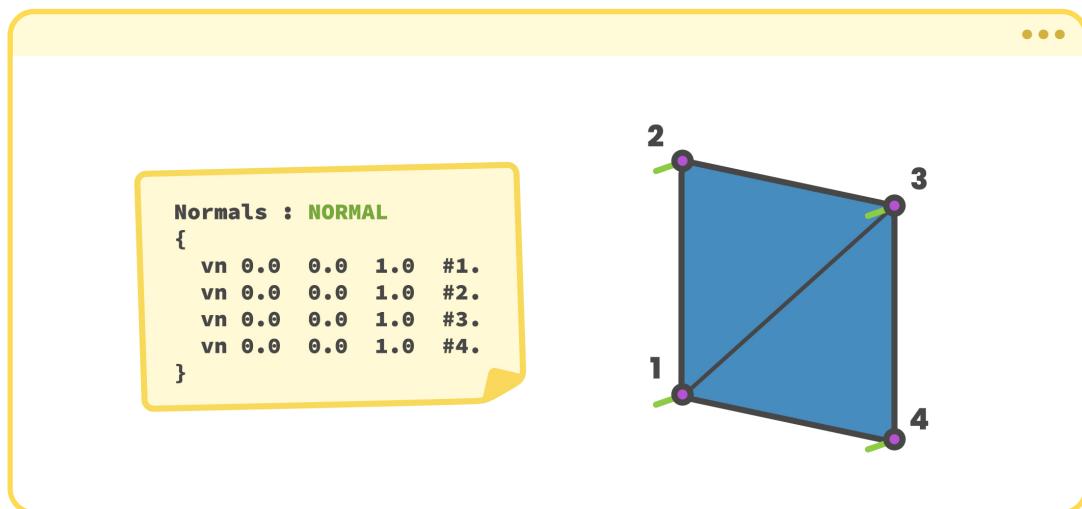
v -0.5 -0.5 0.0      # 1.
v -0.5 0.5 0.0       # 2.
v 0.5 0.5 0.0        # 3.
v 0.5 -0.5 0.0       # 4.

vt 0.0 0.0            # 1.
vt 0.0 1.0            # 2.
vt 1.0 1.0            # 3.
vt 1.0 0.0            # 4.

vn 0.0 0.0 1.0        # 1.
vn 0.0 0.0 1.0        # 2.
vn 0.0 0.0 1.0        # 3.
vn 0.0 0.0 1.0        # 4.
```

(1.2.i Defining the Normals on a **QuadMesh**)

It is worth noting that the Z component could just as easily be -1.0. However, a positive value has been chosen to align with Godot's spatial axes and its default directional lighting.



(1.2.j Visualizing Normalized Normals on a QuadMesh)

The **NORMAL** variable in the shader will give you access to this information, and by manipulating it, you can change how light interacts with the surface. This opens the door to a wide range of effects — from subtle changes in the appearance of lighting to creating complex visual distortions.

Up to this point, we've defined vertices, UV coordinates, and normals — the basic elements the GPU needs to understand your geometry's shape and orientation. However, we still need to specify how these vertices are grouped to form polygonal faces. In an **.obj** file, faces are defined with the letter **f**, followed by the vertex index, UV coordinate index, and normal index, in that order, separated by slashes (/).

```

custom_quad.obj
File Edit View

# This file uses centimeters as units for non-parametric coordinates.

g default

v -0.5 -0.5 0.0      # 1.
v -0.5 0.5 0.0       # 2.
v 0.5 0.5 0.0        # 3.
v 0.5 -0.5 0.0       # 4.

vt 0.0 0.0            # 1.
vt 0.0 1.0            # 2.
vt 1.0 1.0            # 3.
vt 1.0 0.0            # 4.

vn 0.0 0.0 1.0        # 1.
vn 0.0 0.0 1.0        # 2.
vn 0.0 0.0 1.0        # 3.
vn 0.0 0.0 1.0        # 4.

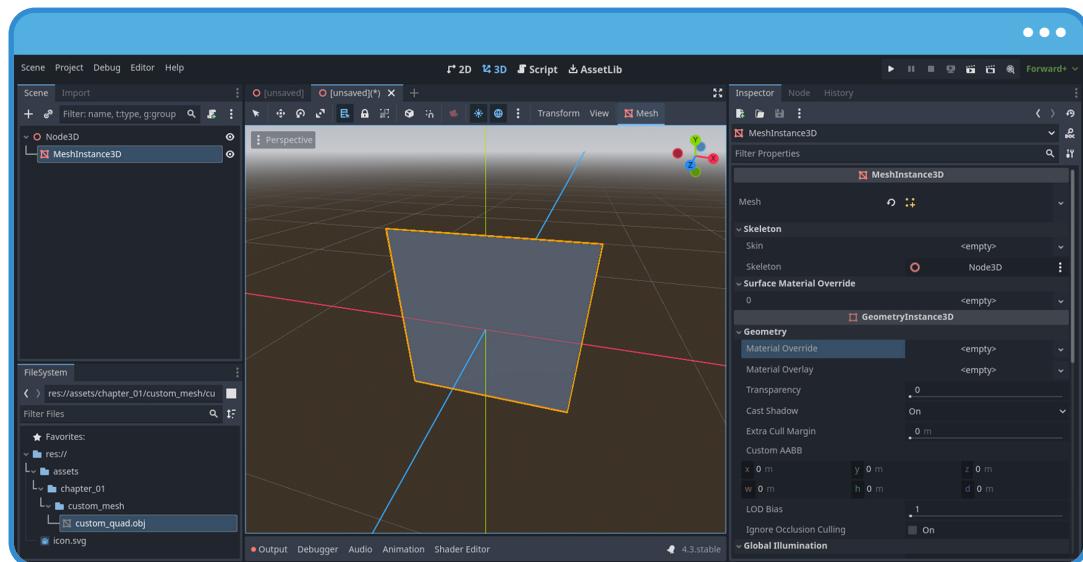
s off
g quad_mesh
f 4/4/4 3/3/3 2/2/2 1/1/1

```

(1.2.k Defining the Mesh faces **f**))

For example, a reference like [4/4/4] indicates that the face uses vertex number 4, UV coordinate number 4, and normal number 4. The **s** parameter sets normal smoothing; when it is **off**, no smoothing is applied.

Once you define the faces, save the changes and return to Godot. By right-clicking on **custom\_Quad.obj** in the FileSystem and selecting **Reimport**, the object will update. You can now assign the resulting Mesh to the **MeshInstance3D** node without any issues.



(1.2.I The **custom\_Quad.obj** has been assigned as a mesh in the node)

With this, you have completed the basic definition of a **QuadMesh** using an **.obj** file, setting up vertices, UV coordinates, normals, and the faces that form the geometry. In later chapters, you'll explore how to manipulate this data further, how it integrates with materials and different coordinate spaces, and how you can leverage it to create visually appealing effects in your 2D and 3D scenes.

### 1.3 Introduction to Spaces.

Which came first: time or space?

When working with shaders, one of the most challenging concepts for graphics developers is managing spaces. These define how fundamental rendering elements — such as position, direction, and movement — are represented and transformed within a virtual world. Each space serves a specific purpose, and understanding their differences is crucial for developing advanced visual effects and complex mechanics.

Some of the most common spaces in computer graphics include:

- uv space (or texture space).
- View space.
- Object space (or local space).
- Clip space.
- World space.
- Screen space.
- Tangent space.

Before defining each of these spaces, let's go through a practical exercise to help you understand their nature using a real-world analogy. Imagine your position in the world. Right now, you are on planet Earth, meaning your location is relative to it. If you remain completely still, your position relative to Earth does not change. Even though the planet moves through the Milky Way, your position remains the same in relation to Earth.

However, if we wanted to calculate our position relative to the galaxy, the situation would change. In this case, considering Earth's movement through space, our position within the Milky Way would vary over time. This introduces changes in position values, where the XYZ components would be in constant update, reflecting our movement on a larger scale.

This concept of relative reference is key to understanding how different spaces work in computer graphics. Depending on the reference system used, the same position can be expressed differently, directly affecting how we interpret and manipulate data within a shader.

This same principle applies to the different spaces mentioned earlier. For example, object space describes the position of a mesh's vertices relative to the object itself, meaning its pivot point or center of mass. If you move the object within the world, its vertices shift accordingly. However, their relative position to the object's center remains unchanged — just like when we previously defined vertices in a **.obj** file, where each position was expressed in relation to the object itself rather than its surroundings.

**Existential question:** The atoms in our body move with us, which might lead us to think they exist in object space. But relative to what? Is there an absolute center in our body? How do atoms "know" where they should be at all times? These questions reflect the relative nature of spaces in computer graphics: the position of a point only has meaning within a specific frame of reference.

Continuing with our analogies, world space describes positions relative to the world itself. In a software like Godot, this space is defined within the **viewport**, which represents the game world. It doesn't matter whether you're working in a 2D or 3D environment – the grid visible in the editor view corresponds to this global space.

On the other hand, UV space refers to the coordinates used to map textures onto a surface. However, before fully understanding this concept, we first need to define what a texture is. From a graphical perspective, a texture is a collection of pixels arranged in a two-dimensional structure, forming an image that can be applied to an object.

From a mathematical perspective, a texture is defined as:

$$T(u, v) \rightarrow R^n$$

(1.3.a)

Where  $T$  represents the texture function,  $uv$  are its coordinates, and  $R^n$  defines the output space, meaning the number of color components. For example, if  $n = 3$ , the texture could be a **.jpg** image, which contains three color channels RGB. In contrast, if  $n = 4$ , we could be referring to a **.png** image, which includes a fourth channel RGBA for transparency.

From a programmatic perspective, a texture is simply a data structure organized as a list of values. Each element in this list represents a pixel, containing both color information and a specific position within the image.

To better visualize this, let's consider a **4×4 pixel** texture:

```
image_data_4x4 = {

    (0, 0):(0 , 0 , 255),
    (0, 1):(255, 255, 0 ),
    (0, 2):(128, 128, 128),
    (0, 3):(0 , 255, 255),
    (1, 0):(255, 0 , 0 ),
    (1, 1):(255, 0 , 255),
    (1, 2):(0 , 255, 0 ),
    (1, 3):(0 , 0 , 0 ),
    (2, 0):(0 , 0 , 255),
    (2, 1):(255, 255, 0 ),
    (2, 2):(128, 128, 128),
    (2, 3):(0 , 255, 255),
    (3, 0):(255, 0 , 0 ),
    (3, 1):(255, 0 , 255),
    (3, 2):(0 , 255, 0 ),
    (3, 3):(0 , 0 , 0 ),

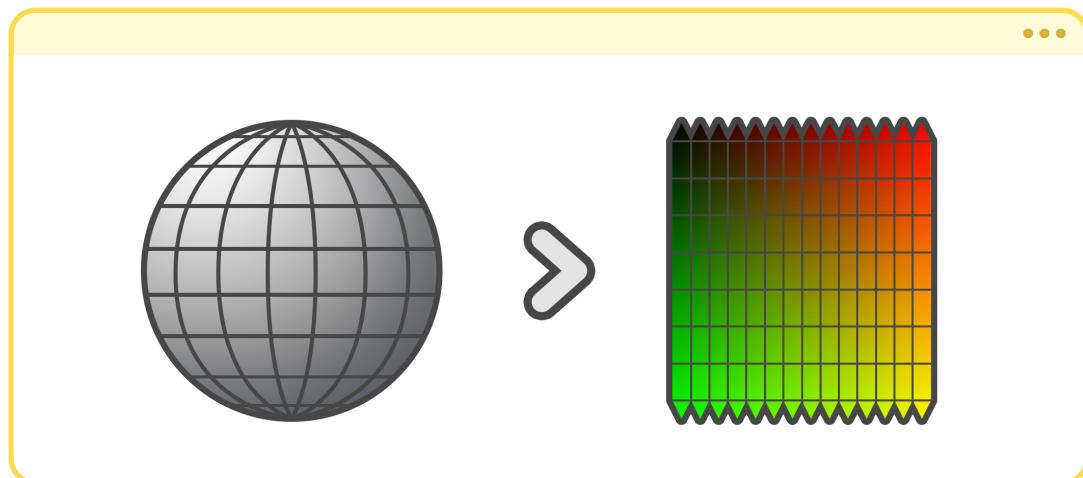
}
```

(1.3.b Texture data visualization)

If we examine the reference above, we can see that on the left, there is a dictionary-style representation of a list called **image\_data\_4x4**. In this structure, each entry stores both the pixel's position and its color value in RGB format. On the right, we find the visual representation of this data: a texture composed of 16 texels (or pixels in Photoshop), each assigned to a specific color.

It's important to note that, while we can conceptually represent a texture as a dictionary, this approach is neither memory-efficient nor performance-friendly. However, it serves as a useful model for understanding the nature of a texture and its relationship with UV coordinates. In practical applications, textures in computer graphics are typically stored as one-dimensional arrays of numerical values, optimized for GPU processing. Each texel within the texture is accessed using its UV coordinates, enabling operations such as sampling, interpolation, and filtering.

UV coordinates are defined in a normalized space, where the origin is located at  $(0.0_u, 0.0_v)$  and the opposite corner at  $(1.0_u, 1.0_v)$ . This coordinate system allows mapping the information stored in a texture, similar to the representation shown in Reference 1.3.b.



(1.3.c Vertices and UV coordinates)

Since UV space is relative to each vertex of the object, moving the model in world space does not affect its UV coordinates. This ensures that the texture remains correctly projected onto the surface. Additionally, UV coordinates are defined using floating-point values, providing high precision in texture mapping. This characteristic is crucial for achieving high texel density, allowing you to add sharp details to characters and objects without losing resolution.

## 1.4 Configuring a Unique Object.

During this chapter, you will complete a hands-on exercise in Godot to better understand how world space functions and its relationship with the internal variables `NODE_POSITION_WORLD` and `MODEL_MATRIX`.

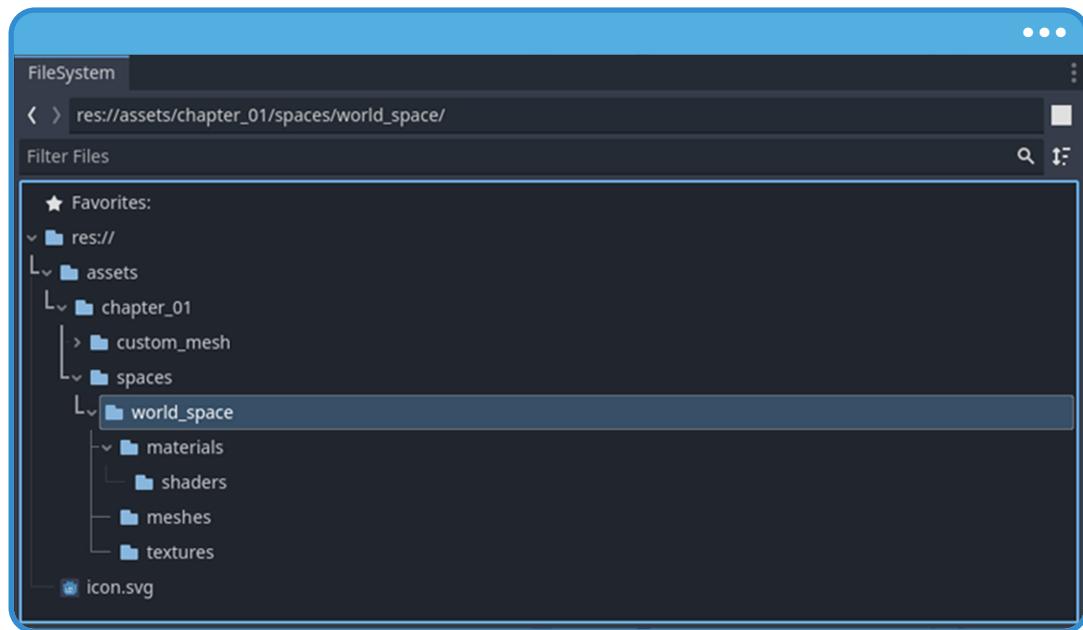
To streamline the implementation process, follow these steps to set up your project:

- ① Inside your project folder, navigate to **chapter\_01** and create a new folder named **spaces**.
- ② Inside **spaces**, add another folder named **world\_space**.

③ Finally, within **world\_space**, organize the content by creating the following subfolders:

- a materials.**
- b shaders.**
- c meshes.**
- d textures.**

If you have followed these steps correctly, your project structure should look like this:



(1.4.a The shader folder has been included inside materials)

To illustrate the usefulness of global coordinates, we will complete the following exercise: Using a **shader**, we will modify the scale and color of a 3D object's texture based on its position in world space. This will allow us to observe how the same object dynamically changes as it moves through the scene. As a result, if you duplicate the object and move it across the environment, each instance will display unique variations in its appearance without manually modifying its material.

Assuming you have already downloaded the supporting resources for this book, we will use the **tree\_mesh.fbx** model during this exercise. Import this model into the **meshes** folder that we created earlier. Additionally, make sure to include the **checker\_tex** and **tree\_tex**

textures inside the textures folder. These assets will be essential for visualizing the effects applied through the shader.

**Note**

Remember that this book includes a downloadable package containing all the necessary files to follow the exercises step by step. You can download it directly from your Jettelly account at: <https://jettelly.com/>

It's important to mention that, in addition to the model and textures, we will also need a **material** and a **shader**. In the shader, we will implement functions in both the vertex stage **vertex()** and the fragment stage **fragment()** to modify both the color and the scale of the object. To create these resources, right-click on the **materials** folder and select:

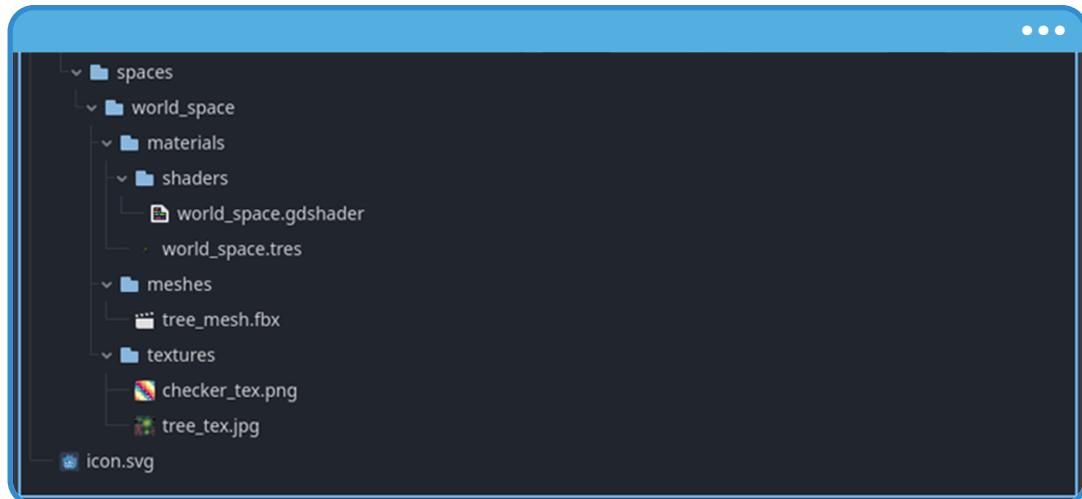
- Create New > Resource > ShaderMaterial.

For practical reasons, name this material **world\_space**.

Repeating part of the previous process, navigate to the **shaders** folder, right-click, and select:

- Create New > Resource > Shader.

For the **shader**, use the same name as the material (**world\_space**). This will make it easier to identify the connection between the two. If everything has been set up correctly, your project structure should look like this:



(1.4.b The downloadable files have been included in the project)

Next, we will begin the implementation process. To do this, we need a single instance of the **tree\_mesh** object.

Follow these steps to create it:

- ① Locate the file **tree\_mesh.fbx** in the **FileSystem** panel.
- ② Right-click on it and select **New Inherited Scene**.

This action allows you to create a scene based on the original file, maintaining an inheritance relationship. This makes it easier to apply future modifications without altering the base model.

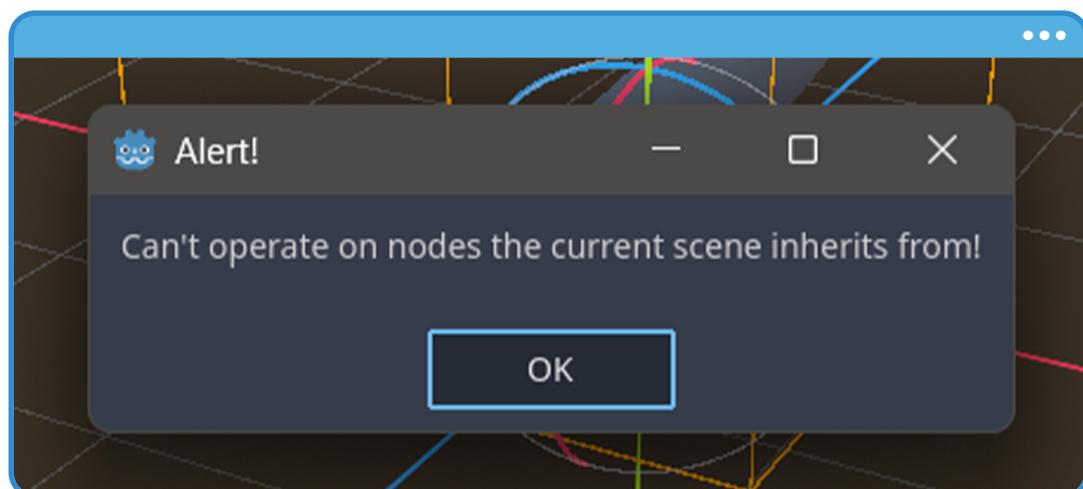
By doing this, a new scene will open in the **Scene** window, containing the following elements:

- A **Node3D** node named **tree\_mesh**.
- A **MeshInstance3D** node named **geo\_tree\_013**.

To demonstrate the use of global coordinates, we need to create multiple instances of **tree\_mesh** at different positions within the 3D scene.

However, when viewing the object in the **Scene** window, you may notice it appears highlighted in **yellow**. This indicates that the node is locked due to **inheritance**, preventing direct editing.

In fact, if you attempt to rename the node by pressing **F2 (Rename)**, you will see the following message on the screen:

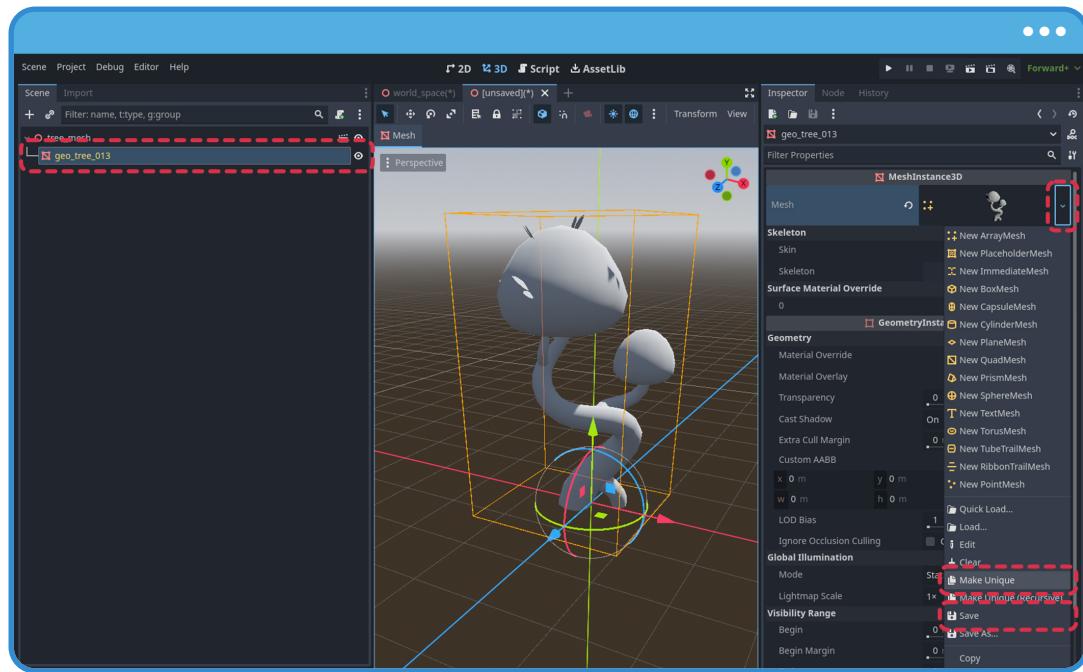


(1.4.c The node is not editable)

To use and customize the node according to your needs, follow these steps:

- 1 Select the node in the **Scene** window.
- 2 In the **Inspector** panel, locate the **mesh** property.
- 3 Click **Make Unique** to unlink the inherited resource and convert it into an independent instance.
- 4 **Save** the new resource in the **meshes** folder of your project.
- 5 To make it easier to identify, rename it **tree**.

By doing this, you will have an independent object that you can modify freely without affecting the original resource.

(1.4.d The object has been marked as **unique**)

If you have followed all the steps correctly, you should now find a new independent file inside the **meshes** folder. This file can be used flexibly throughout the exercise without affecting the original resource.



(1.4.e A unique object has been saved)

In this step, you need to create a new scene that contains multiple instances of the **tree** object. Therefore, navigate to the **Scene** window and start a new scene by selecting:

- Menu: Scene > New Scene.

Alternatively, you can use the keyboard shortcut **Ctrl + N**. Since we are working with a 3D object, select **3D Scene** as the scene type.

**Note**

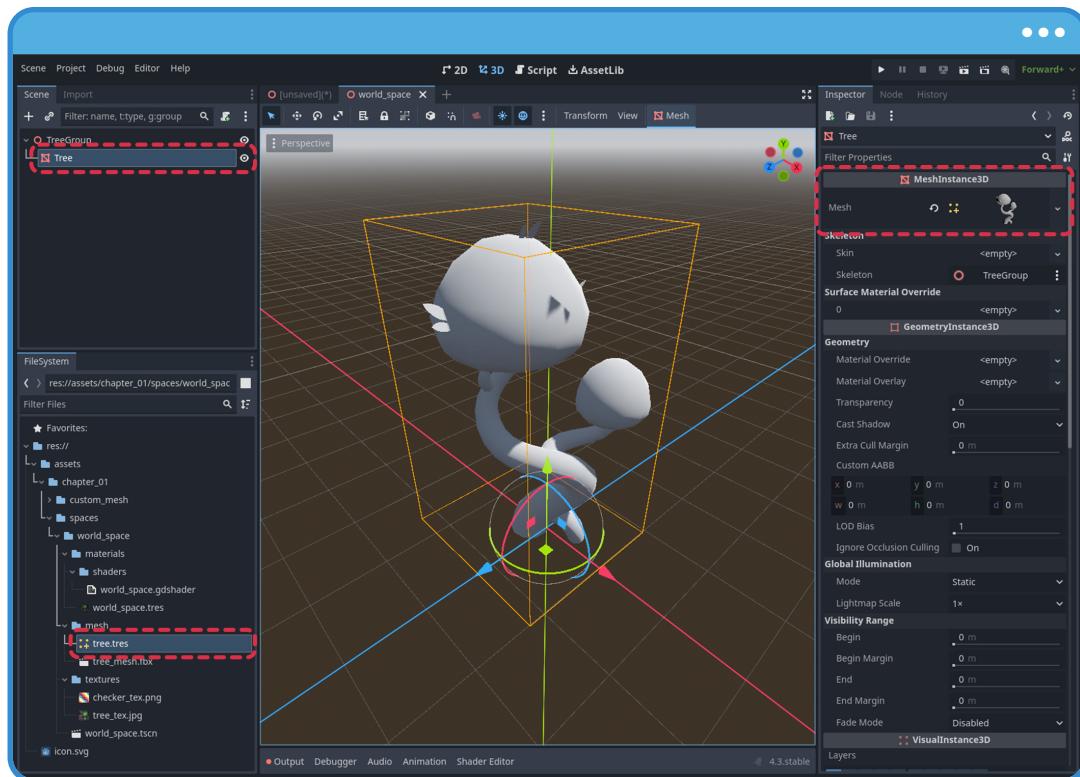
According to the Godot 4.3 documentation, Node3D is the base node for 3D objects, providing essential transformation and visibility properties. All other 3D nodes inherit from it. In a 3D project, you should use Node3D as the parent node whenever you need to move, scale, rotate, show, or hide its child elements in an organized and efficient manner. For more information, visit the following link: [https://docs.godotengine.org/en/stable/classes/class\\_node3d.html](https://docs.godotengine.org/en/stable/classes/class_node3d.html)

Next, add a **MeshInstance3D** as a child node by following these steps:

- 1 Select the previously created **Node3D** in the **Scene** window.
- 2 **Right-click** on it and choose **Add Child Node**.
- 3 In the node list, search for **MeshInstance3D** and add it to the scene.

While we could use a **MultiMeshInstance3D** node along with a **MultiMesh** resource to optimize the creation of multiple instances, it is not necessary in this case. Since we will only use a few copies of the **tree** object to demonstrate the utility of global coordinates, a **MeshInstance3D** node is sufficient and appropriate. This is because it allows you to directly assign a **Mesh** resource.

Now, simply select the **tree** object and assign it to the **Mesh** property of the **MeshInstance3D** node, as shown below:



(1.4.f The tree object has been assigned into the Mesh property)

With this last step, you have completed the initial setup needed to begin developing your **shader**. Now, follow these steps:

- 1 Assign the **world\_space** shader to the **material**.
- 2 Apply the material to the **tree** object.

If everything has been done correctly, the **tree** object should appear white in the scene. This indicates that the shader has been successfully applied and is now ready for programming.

## 1.5 Rendering Stages and Pipeline.

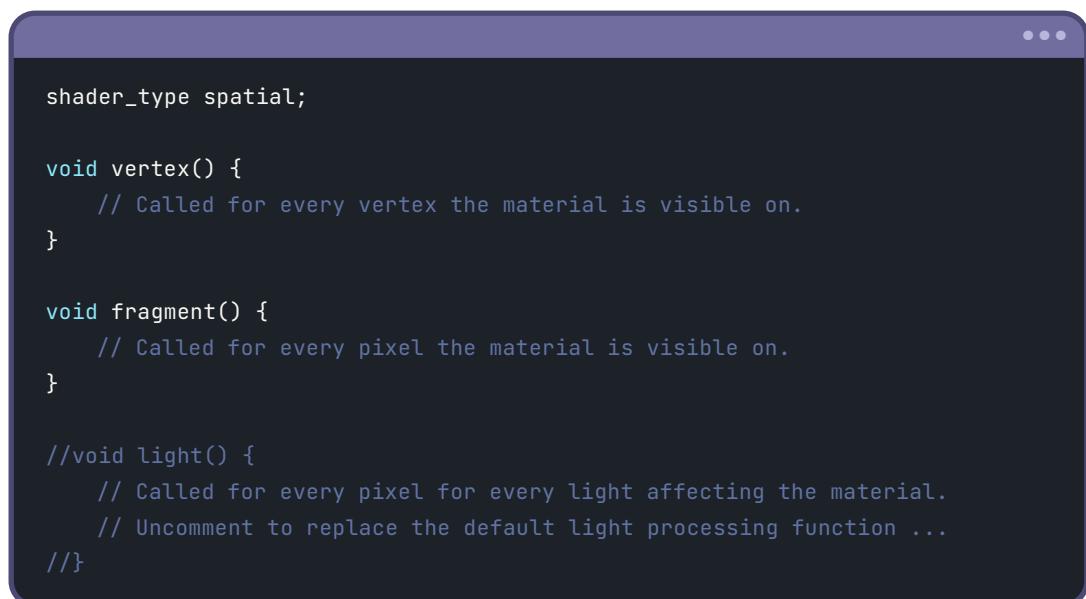
To begin this section, locate the **world\_space** shader and open it by double-clicking on it. In Godot, shaders are structured into processing stages, and two of the most commonly used methods are:

- **vertex()**: Responsible for manipulating vertices before rasterization.
- **fragment()**: Handles processing each fragment (pixel) in the final stage of rendering.

These methods control a series of GPU processes, including:

- Vertex transformation.
- Attribute processing.
- Projection.
- Clipping.
- Screen mapping.
- Color processing.
- Transparency calculations.

In fact, when you open a shader in Godot, its initial configuration typically follows this structure:



```

shader_type spatial;

void vertex() {
    // Called for every vertex the material is visible on.
}

void fragment() {
    // Called for every pixel the material is visible on.
}

//void light() {
//    // Called for every pixel for every light affecting the material.
//    // Uncomment to replace the default light processing function ...
//}

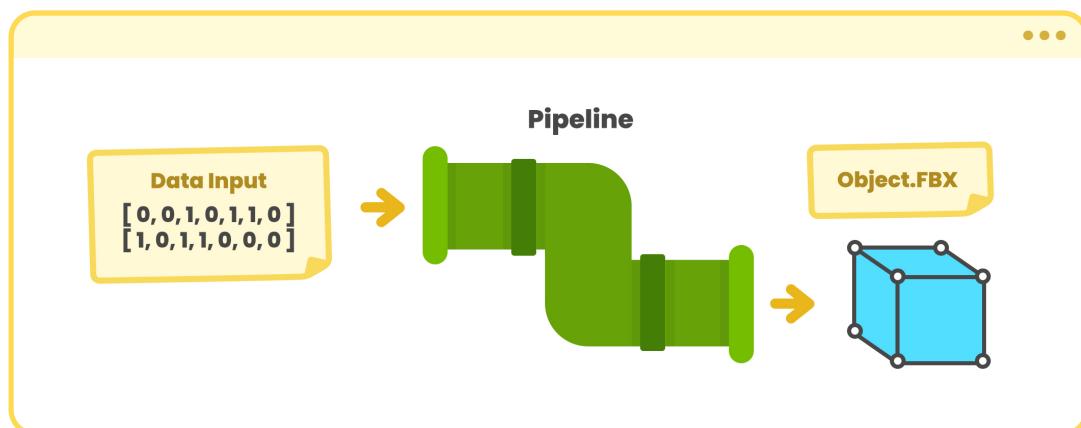
```

In the previous code, you may notice the presence of a third method called **light()**, which is commented out by default. This method executes once per light source that affects the object. However, in this section, we will focus only on the **vertex()** and **fragment()** methods, leaving the analysis of **light()** for a later discussion.

To understand how the `vertex()` and `fragment()` methods work, it's essential to grasp the nature of the **Render Pipeline**. An analogy I like to use to explain this concept is the pipes in Super Mario.

In Section 1.2 of this book, we created a **QuadMesh** using coordinates defined in a **.obj** file. As you may have noticed, all shapes – whether 2D, 3D, or even colors – initially exist only as data in memory. This leads us to a key question: How are these data transformed into visible, colorful objects on the screen? The answer lies in the Render Pipeline.

We can think of this concept as a chain process, where a polygonal object travels through different stages until it is finally rendered on screen. It's as if the object moves through a series of pipes, gradually transforming step by step until it reaches its final destination: your computer screen.



(1.5.a Data being transformed into a three-dimensional Cube)

Godot uses a unified Render Pipeline, unlike Unity, where each Render Pipeline (URP, HDRP, or Built-in RP) has its own characteristics and rendering modes. In Unity, the choice of Render Pipeline directly affects material properties, light sources, color processing, and overall graphic operations that determine the appearance and optimization of objects on screen.

In Godot, the rendering system is based on **OpenGL** and includes **different rendering modes**. These modes were introduced at the beginning of the book when we set up our project. The available rendering modes are:

- Forward+.
- Forward Mobile.
- Compatibility.

Starting with **Forward+**, if we refer to the **official Godot documentation**, we find the following information:

“

This is a **Forward** renderer that uses a clustered approach to lighting. Clustered lighting uses a compute shader to group lights into a 3D frustum aligned grid. Then, at render time, pixels can look up what lights affect the grid cell they are in and only run light calculations for lights that might affect that pixel. This approach can greatly speed up rendering performance on desktop hardware, but is substantially less efficient on mobile.

<https://docs.godotengine.org/en/stable/tutorials/rendering/renderer.html>

”

The term **clustered** approach refers to an optimization method used in lighting calculations. Instead of computing lighting individually for each pixel or object in the scene, this method divides the 3D space into clusters (small spatial regions). During rendering, each pixel only queries the lights affecting its specific **cluster**, avoiding unnecessary calculations and improving performance — especially in scenes with complex dynamic lighting.

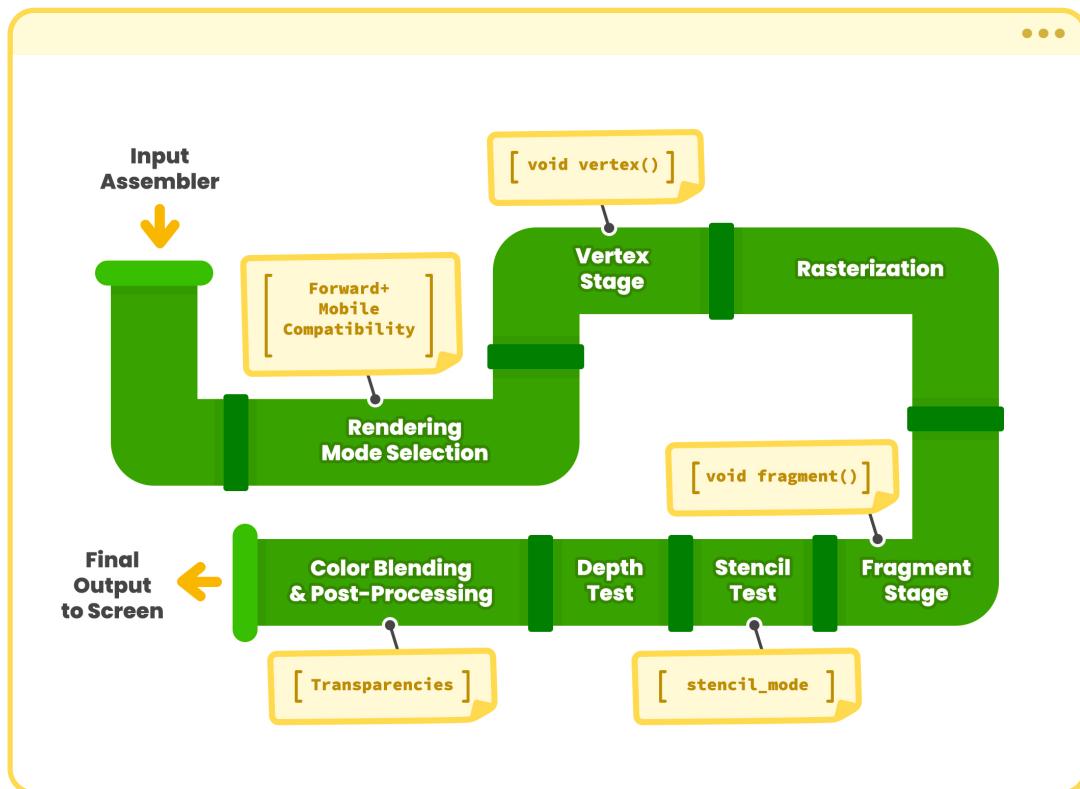
Thanks to this optimization, **Forward+** is primarily used in PC projects, where high graphical quality and multiple light sources are required. In contrast, other rendering modes in Godot are designed for different needs:

- **Forward Mobile:** Optimized for mobile devices, prioritizing performance over graphical quality.
- **Compatibility:** Also based on Forward Rendering, but intended for older GPUs that do not support Vulkan.

**Note**

Given the specifications and limitations of OpenGL, we will not cover all aspects of Godot's rendering system in detail. If you would like to learn more about the different rendering modes and their features, you can visit the following link: [https://docs.godotengine.org/en/4.4/contributing/development/core\\_and\\_modules/internal\\_rendering\\_architecture.html](https://docs.godotengine.org/en/4.4/contributing/development/core_and_modules/internal_rendering_architecture.html)

Now, what is the relationship between these **rendering modes** and the `vertex()` / `fragment()` methods? To understand this better, let's examine the following reference:



(1.5.b Some of the most important processes of the Render Pipeline)

The **Input Assembler** is the component responsible for receiving and organizing the data of objects (meshes) present in the scene. This process includes information such as:

- Vertices.
- Indices.
- Normals.
- Tangents.
- UV Coordinates.
- Other geometric attributes.

All this information is stored as numerical values and organized in a structured manner so that the Render Pipeline can correctly interpret it. This structured data is then processed to ultimately generate the 3D objects displayed on screen.

The choice of rendering mode determines which rendering technique will be used in the project (Forward+, Forward Mobile, or Compatibility). This selection has a direct impact on both performance and visual quality, as it defines how graphics are processed and which rendering API will be used (Vulkan, Direct3D, OpenGL ES, etc.). Depending on the selected mode, certain graphical effects and optimizations will be either available or restricted, influencing how shaders interact with the scene.

The **vertex processing stage**, represented by the `vertex()` method, is one of the fundamental phases in a shader. During this stage, the data retrieved from the **Input Assembler** is transformed through different coordinate spaces, following this sequence:

- 1 Object Space:** The local space of the model, where vertices are defined relative to their own pivot.
- 2 World Space:** Transforms vertices based on the object's position within the scene.
- 3 View Space:** Adjusts vertices according to the camera's perspective, determining how they appear to the observer.
- 4 Clip Space:** Projects vertices into a homogeneous coordinate system, preparing them for the next phase.
- 5 Normalized Device Coordinates:** Performs division by W, normalizing coordinates to a range between -1.0 and 1.0 on each axis.
- 6 Screen Space:** Assigns vertices to pixel coordinates on the screen, where the object is rasterized and rendered.

This process ensures that the 3D scene is correctly projected onto a 2D surface (screen), converting polygonal models into visible images on screen.

Once the vertices are transformed into screen space, the next step in the Render Pipeline is **rasterization**. During this stage, the geometric primitives – such as triangles, lines, or points – are converted into fragments (potential pixels) on the screen. This process determines which parts of the object will be visible and how they will be distributed in the final image.

Additionally, during this stage, **Depth Testing** is applied — a key mechanism that determines the visibility of fragments based on their relative distance from the camera. This test compares the depth of each fragment with the values stored in the **Z-buffer** (or **Depth Buffer**), discarding those that are occluded by other objects. This ensures that only the closest visible surfaces are rendered, preventing overlapping artifacts and improving scene realism.

After rasterization, each fragment enters the fragment stage **fragment()**, where its final color is computed before being displayed as a pixel on the screen. In this phase, key visual effects are determined, including lighting, textures, transparency, and reflections. Additionally, each fragment is processed independently in the GPU, enabling the efficient execution of complex calculations while maintaining high performance.

**Note**

It's important to highlight that the **light()** method does not replace the **fragment()** method; rather, it is executed afterward, but only in the presence of light source that affect the current fragment. We'll explore its behaviour in more detail throughout Chapter 2.

Finally, the **Blending and Post-Processing** stages are applied, incorporating visual effects before rendering the image on screen. At this stage, elements such as transparency, color blending, bloom, anti-aliasing, and color correction are processed. These effects enhance visual quality and optimize the final appearance of the scene, ensuring a more polished and immersive rendering.

Now that we understand the relationship between the different rendering modes and shader methods, we will proceed with the implementation of **world\_space**.

## 1.6 Working with World Space Coordinates.

The directive **shader\_type spatial** is a key attribute of our shader as it specifies that the shader will be applied to 3D objects.

**Note**

If we wanted to create shaders for UI elements or sprites, we would need to use the `shader_type canvas_item` directive. For particle effects, on the other hand, we would use `shader_type particles`. You can find more information about this at the following link: [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/index.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/index.html)

Since our `tree` object currently appears white by default, we will start by adding a **texture** to our shader. To do this, add the following line of code:

```
shader_type spatial;

uniform sampler2D _MainTex : source_color;

void vertex()
{
    // Called for every vertex the material is visible on.
}
```

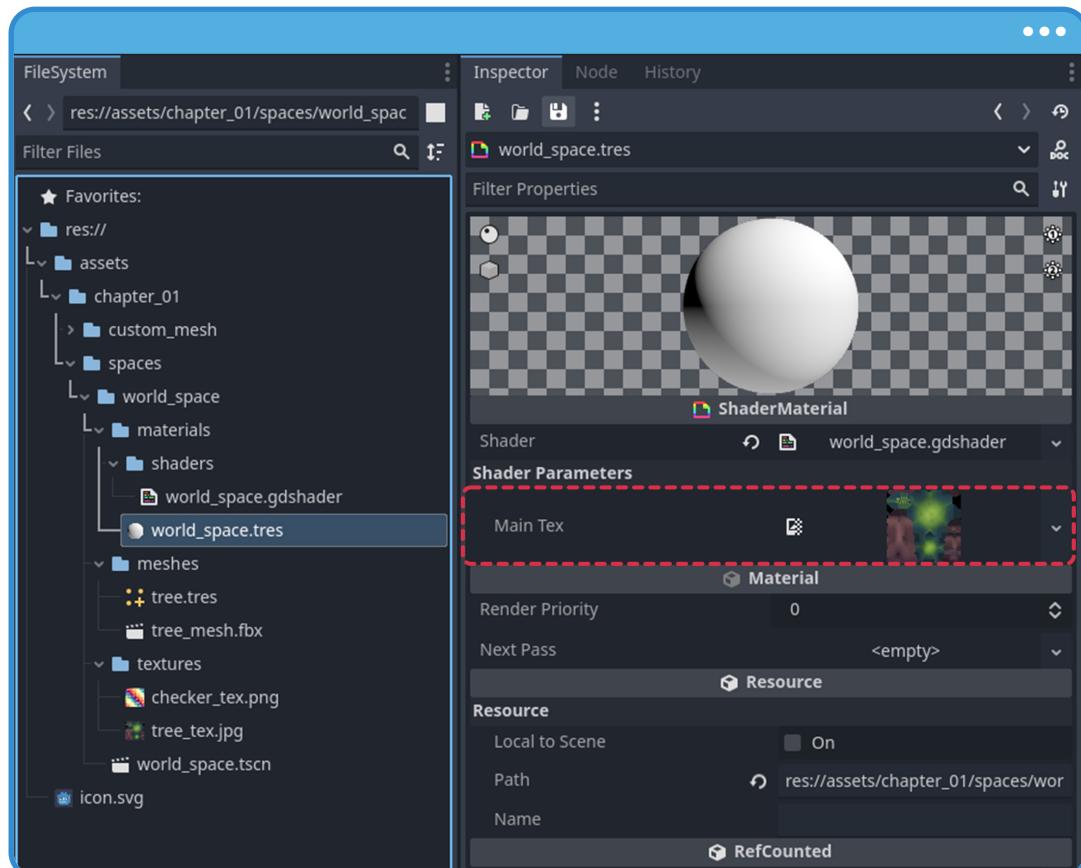
In the previous code, a new `sampler2D` variable named `_MainTex` was declared, which includes the **hint source\_color**. This variable is defined globally, meaning it can be used in any method within the shader.

A `sampler2D` refers to a two-dimensional texture, which can be used within the shader. The keyword `uniform` indicates that the texture will be assigned externally from the **material**. On the other hand, `source_color` specifies that the texture functions as an albedo map (base color), meaning it determines the main color of the material.

**Note**

`source_color` is just one of the many hints you can use in a shader. For example, you can use `hint_normal` to define a normal map. If you want to see the complete list of available hints, you can check the following link: [https://docs.godotengine.org/en/latest/tutorials/shaders/shader\\_reference/shading\\_language.html](https://docs.godotengine.org/en/latest/tutorials/shaders/shader_reference/shading_language.html)

In fact, if you save the changes and select the **material** from the **FileSystem** window, you will see a new property called "**Main Tex**" in the **Shader Parameters** section. This property directly references the `_MainTex` variable we just declared in the shader, allowing you to assign a texture from within the Godot editor.



(1.6.a A new texture property has been declared)

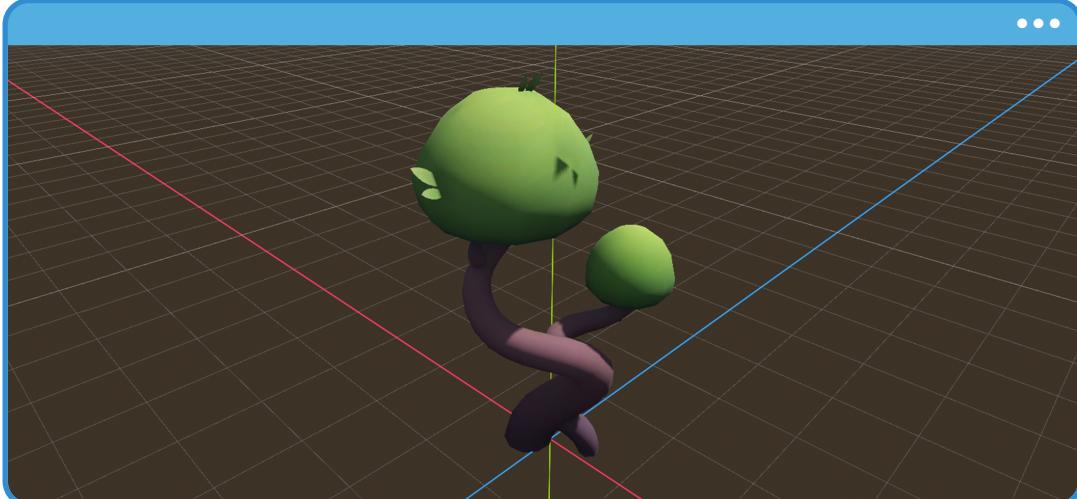
However, if you assign any texture to the property, you'll notice that the **tree** object in the scene does not change color. This happens because we haven't yet declared an RGB vector to process the texture color for each fragment.

Since we need to compute the texture color for each pixel, we must go to the **fragment stage** of our shader and add the following lines of code:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}
```

From the previous code, there are some key aspects we need to consider for interpreting the operation we are performing:

- The **texture()** function: This function returns an RGBA vector and takes two arguments: A **sampler2D** variable (the texture), and a 2D vector (the UV coordinates). Essentially, **texture()** samples the texture **\_MainTex** using the UV coordinates, retrieving the corresponding color, which is then stored in the RGB vector **albedo**.
- Assignment to **ALBEDO**: The retrieved texture color is assigned to the **ALBEDO** variable, a predefined Godot variable representing the base color of the fragment's surface.
- Texture Projection: Since UV coordinates are calculated based on vertex positions in object space, the texture should be correctly projected onto the object within the scene.



(1.6.b The tree object has color now)

**Note**

Remember that texels are the color points within a texture—the smallest color units that make up the image. In contrast, pixels are the color points displayed on your computer screen. Now, why don't we see clearly defined points in the texture applied to the tree object? This happens because Godot automatically applies linear interpolation, smoothing the transition between texels instead of displaying sharp edges.

Let's do a simple exercise now: we'll modify the texture color based on the **tree's** global position. To achieve this, we'll use the internal variable **NODE\_POSITION\_WORLD**, which refers to the object's **Transform Position** in the scene. Since this variable is directly linked to the object's position, we can assume that its return value is a three-dimensional vector, covering the usable area within the scene.

This approach presents a potential issue: What happens if the global position of the **tree** object is  $(100_x, 234_y, 134_z)$  or any other value greater than 1.0? In a shader, colors are defined within a range of 0.0 to 1.0. If position values exceed this range, the result could be an entirely **white** object, as color values beyond 1.0 are interpreted as maximum intensity.

To prevent this issue, we will create a function that limits the return values, ensuring the global position of the object is normalized within the appropriate range. However, we must make sure to declare this function before the **vertex()** method. Why? Because the GPU reads shaders sequentially, from top to bottom. If we need to use this function in both the vertex stage and the fragment stage, it must be declared at the beginning of the shader code so that both methods can access it.

```

uniform sampler2D _MainTex : source_color;

vec3 hash33(vec3 p)
{
    p = fract(p * 0.4567);
    p += dot(p, p.yzx + 1.3456);
    p.x *= p.x;
    p.y *= p.y;
    p.z *= p.z;
    return fract(p);
}

void vertex()

```

What happens in the `hash33()` method? Let's analyze its structure from a mathematical perspective. Given two three-dimensional vectors — one as input and one as output, if  $p = (4.23, 5.52, 3.74)$ :

For the first line of code,

$$\text{fract}(p * 0.4567) = (0.931841, 0.520984, 0.708058)$$

(1.6.c)

If we multiply the first component of the vector  $p$  by 0.4567, we get  $p_x * 0.4567 = 1.931841$ . However, the `fract()` function is applied in the operation, which returns only the fractional part of a number, discarding the integer part. This means  $\text{fract}(1.931841) = 0.931841$ . This applies to all components of the vector  $p$ .

Its definition for a floating-point value is:

```
float fract (float v)
{
    return v - floor(v);
}
```

Then,

$$p \cdot (p_{yzz} + 1.3456) = \sim 4.42184278$$

(1.6.d)

This value is added to each component of the vector  $p$ . Therefore,

$$p = (5.353684, 4.942827, 5.129901)$$

(1.6.e)

Squaring each component,

$$p_x * p_x = 28.661930$$

$$p_y * p_y = 24.431537$$

$$p_z * p_z = 26.315882$$

(1.6.f)

Finally,

$$\text{fract}(p) = (0.661930, 0.431537, 0.315882)$$

(1.6.g)

It is worth noting that the previous method could have any name we choose. However, it has been named **hash33** for the following reasons:

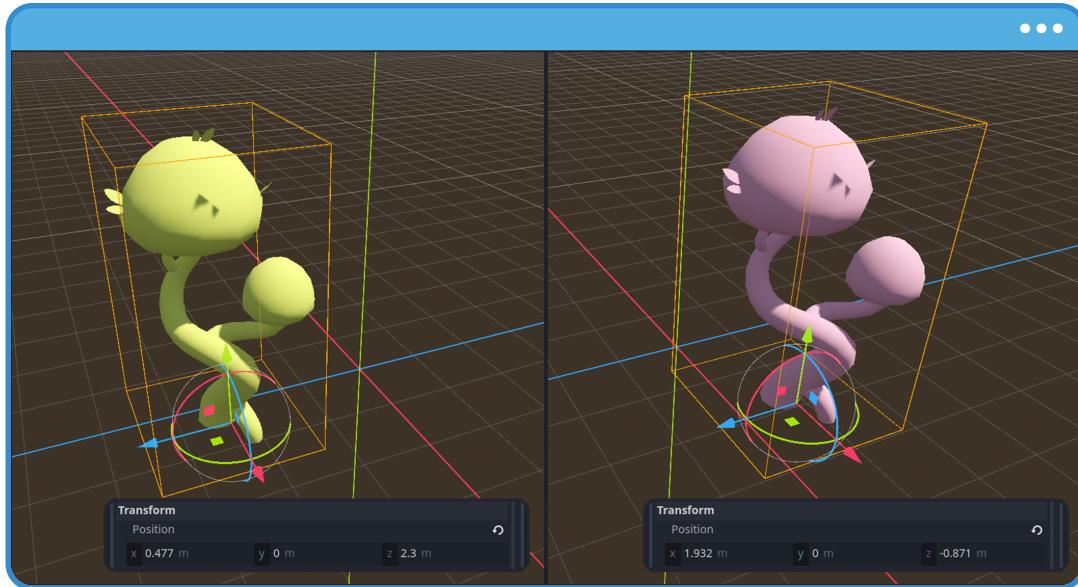
- It returns a three-dimensional vector as output (3).
- It generates pseudo-random values (hash).
- It receives a three-dimensional vector as an argument (3).

This type of method is commonly used to generate visual “noise.” However, when combined with other functions, it can produce more interesting effects, such as the one we will implement next.

```
void fragment()
{
    vec3 random_color_ws = hash33(NODE_POSITION_WORLD);
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = random_color_ws;
}
```

As we can see, a new three-dimensional vector named **random\_color\_ws** has been declared and initialized with the output of the **hash33()** method, using the global position of the **tree** object as input. Subsequently, this RGB vector is assigned to the **ALBEDO** variable, setting the base color of the shader based on the object’s position in world space.

If you save the changes and move the object across the scene, you will notice that its color changes dynamically depending on its position.



(1.6.h The object's color changes every time it changes its position in the world)

Due to the resolution of the object's position, the color may flicker multiple times as you move it. This happens because position values change continuously, causing multiple variations in color.

In some cases, this effect may be a problem. To fix this, we can limit the color change so that it only updates per unit of movement. Implementing this solution is simple: we extend the color calculation operation and include the `floor()` function, which, by definition:

“

It returns largest integer not greater than a scalar or each vector component.

<https://registry.khronos.org/OpenGL-Refpages/gl4/html/floor.xhtml>

”

Its definition for a three-dimensional vector is::

```
vec3 floor(vec3 v)
{
    vec3 rv;
    int i;

    for (i=0; i<3; i++) {
        rv[i] = v[i] - fract(v[i]);
    }
    return rv;
}
```

The **floor()** function removes the decimal part of a value, returning only its integer component. In practice, this means the object's position will be rounded down, ensuring that its color only changes when it crosses a full metric unit in the scene.

For example,  $p = (1.165, 0.0, 1.712)$

$$\text{floor}(p) = (1, 0, 1)$$

(1.6.i)

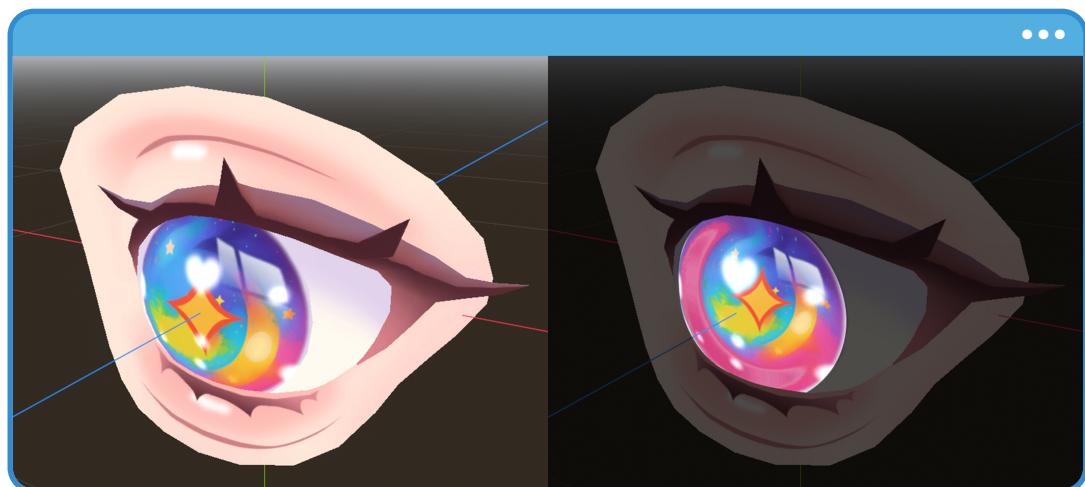
```
void fragment()
{
    vec3 random_color_ws = hash33(floor(NODE_POSITION_WORLD));
    vec3 albedo = texture(_MainTex, UV).rgb;
    albedo += random_color_ws * 0.15;
    ALBEDO = albedo;
}
```

If you save the changes and move the **tree** object within the scene again, you'll notice that its color only changes when it crosses from one metric unit to another on the grid. This prevents constant flickering and allows for better control over color variations based on the object's global position.

## 1.7 Introduction to Tangent Space.

Have you ever wondered how those eye-catching depth effects are created in the eyes of a 3D character? One common approach is to use multiple mesh layers — one for the cornea, another for the pupil, and so on — stacking them with transparency to simulate depth. While this method works, it's not always the most efficient. Instead, you can achieve a similar result by leveraging the interaction between the view direction (from the camera) and the vertex coordinates of the object. This technique simplifies the setup and can be more performance-friendly.

To better understand how this works, take a look at the following image:



(1.7.a The right eye displays depth)

At first glance, you might not notice it clearly; however, the eye on the right conveys a subtle sense of depth that responds to the direction from which you view the scene. As you move the camera, you'll notice that the character's pupil exhibits a certain level of three-dimensionality, even though it's rendered using only a two-dimensional texture.

To achieve this kind of effect, you need to rely on a specific coordinate system known as tangent space, which is defined by three orthogonal vectors: the tangent, bitangent, and normal.

As shown earlier in Figure 1.1.a (at the beginning of this chapter), tangent space can be visualized through these three vectors that define the orientation of a vertex, almost as if

each vertex had its own transformation gizmo. However, tangent space isn't limited to the vertex stage – you can also define it per fragment, allowing you to perform transformations at the pixel level.

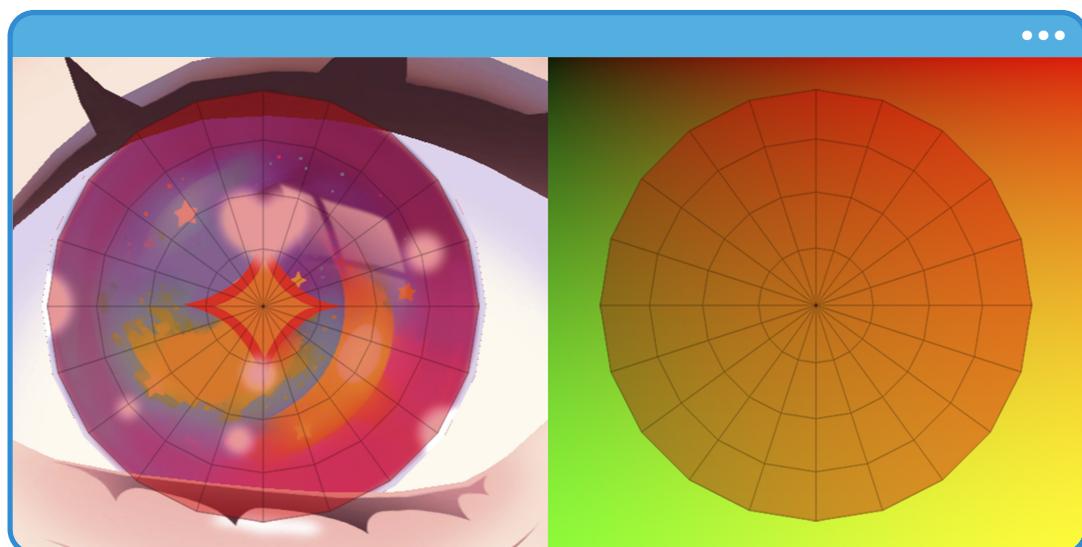
Before diving into the implementation, it's important to quickly analyze the topology of the 3D object you'll be working with. You'll focus on some of its most important properties, particularly its UV coordinates. For this purpose, you'll use the **anime\_eye.fbx** model included in the downloadable files for this book, located in the following directory:

- Assets > chapter\_01 > spaces > tangent\_space > meshes

If you examine the object closely, you'll see that it's made up of three distinct layers:

- Eye.
- Skin.
- Pupil.

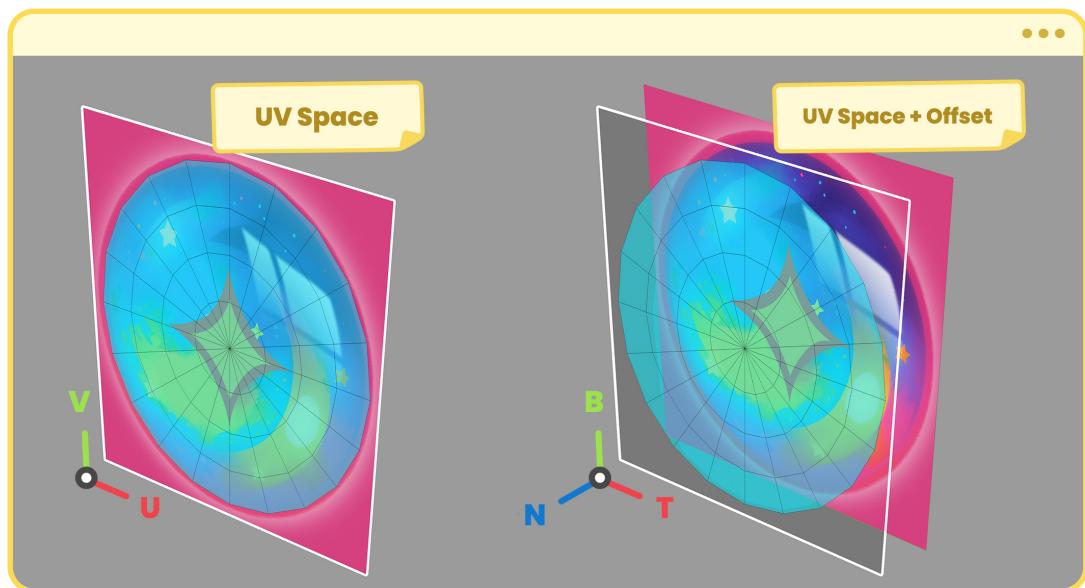
You'll apply the shader to the **Pupil** layer. Therefore, your first step will be to review its UV coordinates. Then, by making a comparison, you'll explore how tangent space can help you generate visual effects like displacement based on the view direction.



(1.7.b Pupil vertices within the UV space)

Although the pupil has a geometric shape similar to an oval, its UV coordinates, in terms of mapping, are contained within a square space. This detail is important because UV coordinates allow movement along only two axes: U and V. In contrast, tangent space introduces a third dimension, enabling you to simulate depth effects directly on the texture.

By applying displacement in tangent space, you can modify the appearance of the texels around the pupil area. As a result, the perceived visual field expands, creating the illusion of three-dimensionality without needing to alter the model's geometry.

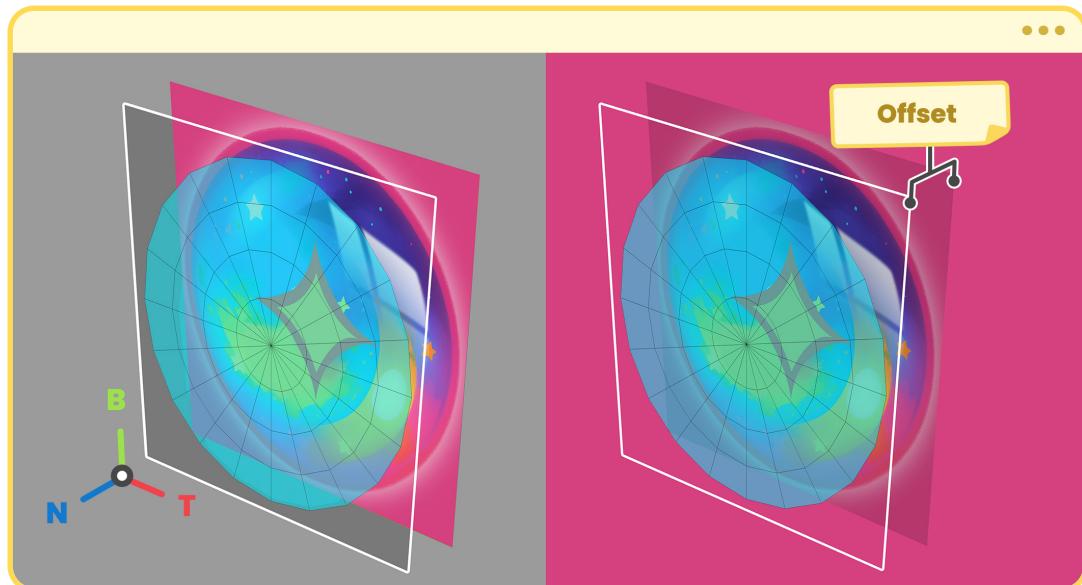


(1.7.c The image on the right shows a slight UV displacement)

In Godot, textures are set to **Repeat** mode by default. This means that if you apply a significant offset to simulate depth, the edges of the texture may begin to repeat across the 3D model – in this case, the pupil. This repetition can negatively impact the visual result, breaking the illusion of depth and natural movement you're aiming to achieve in the eye.

To avoid this unwanted effect, it's best to switch the texture's wrap mode to **Clamp** within the shader settings. With **Clamp**, when the UV coordinates exceed the texture's boundaries, the nearest edge color is extended instead of repeating the image. This results in smoother, more natural transitions in the displaced areas.

Additionally, you can limit the intensity of the displacement by using a mask that defines the area of influence. This mask can be a grayscale texture specifically designed to restrict where the effect is applied, leaving the rest of the surface untouched.



(1.7.d Edge pixels are repeated to ensure accurate results)

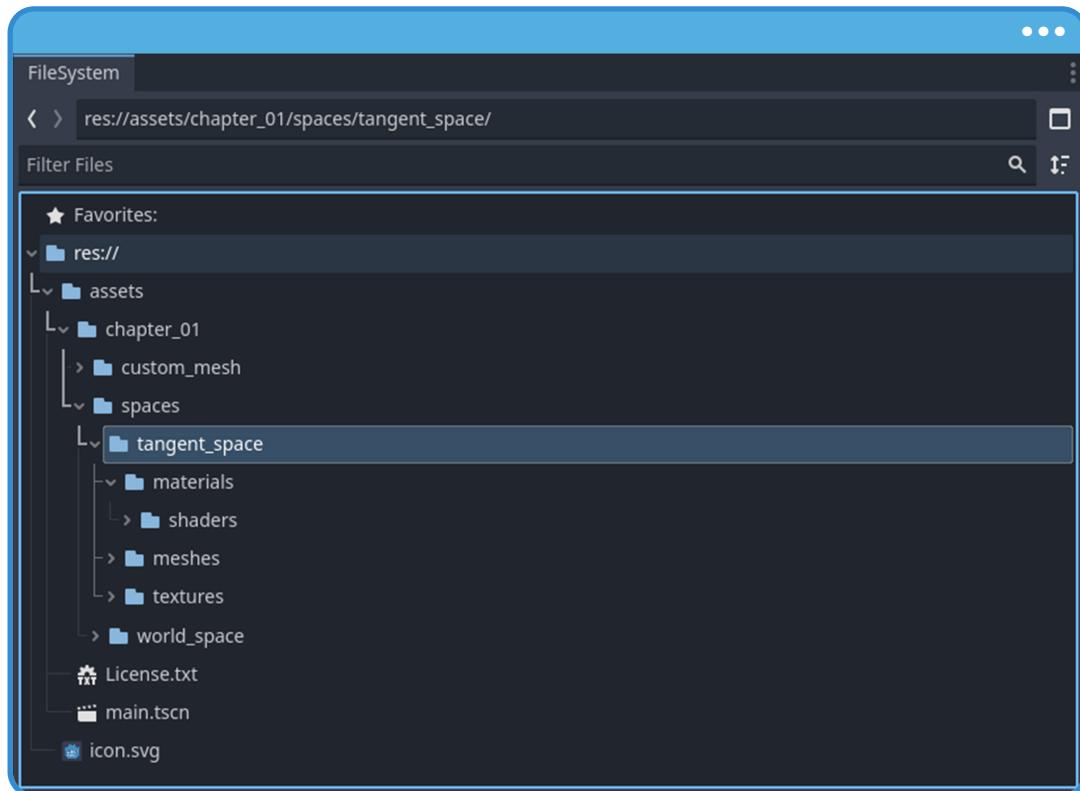
## 1.8 Implementing Tangent Space in Our Shader.

In this section, you'll complete a new hands-on exercise to deepen your understanding of how tangent space works and how it relates to the internal variables **TANGENT**, **NORMAL** and **BINORMAL**.

As a first step, you'll organize the project to make it easier to implement the various functions you'll be working with. Follow these steps:

- 1 Inside your project, navigate to **chapter\_01 > spaces** and create a new folder named **tangent\_space**.
- 2 Inside **tangent\_space**, organize the content by creating the following subfolders:
  - a **materials**.
  - b **shaders**.
  - c **meshes**.
  - d **textures**.

If you've followed the steps correctly, your project structure should look like this:



(1.8.a The **shaders** folder has been included inside **materials**)

To demonstrate the utility of tangent space, you'll work through the following exercise: using a shader, you'll transform the view vector from view space into tangent space. To accomplish this, you'll define a custom function called **view\_to\_tangent()**, which will take four vectors as arguments: the **tangent**, the **bitangent**, the **normal**, and the **view** vector.

This function will calculate the dot product between the view vector and each of the three vectors that form the tangent space basis. The result will be a new vector expressed in local tangent space coordinates. You can then use this vector to create camera-dependent visual effects, such as displacement, shading, or simulated depth.

#### Note

The files you'll need for this exercise are available in the downloadable package linked to your account at: <https://jettelly.com/store/the-godot-shaders-bible>

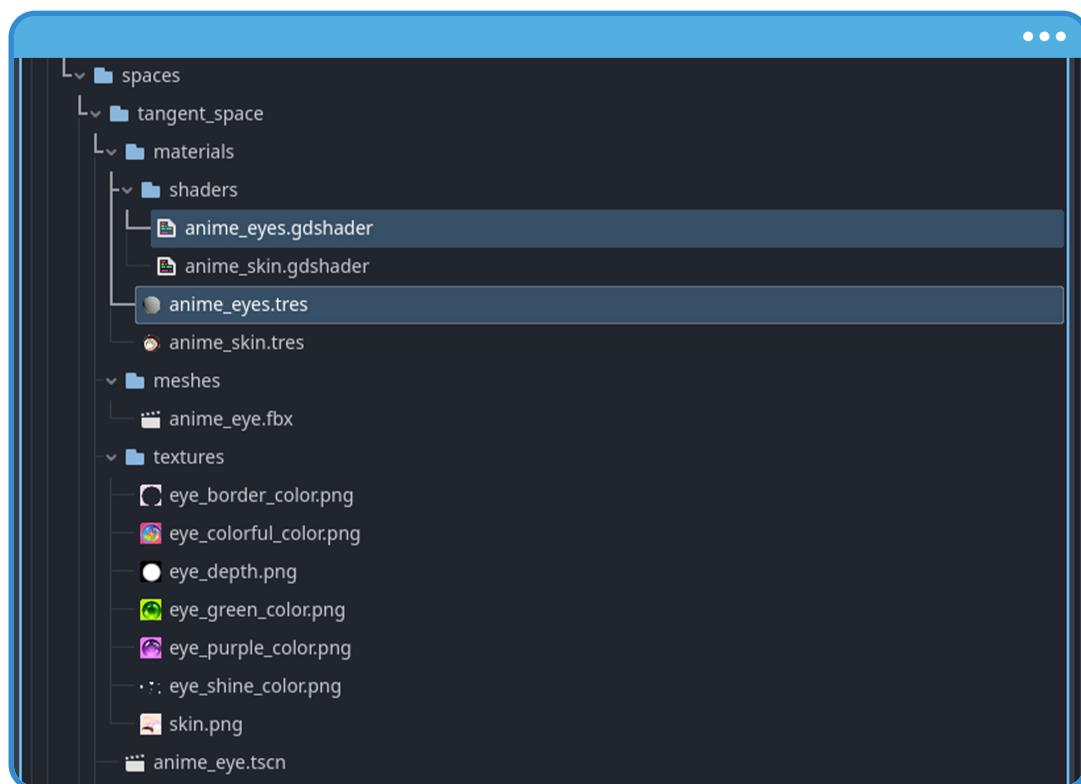
To start the development process, you'll first create a new **material** and a **shader**. Begin by right-clicking on the **materials** folder and selecting:

- Create New > Resource > ShaderMaterial

For practical reasons, name this material **anime\_eyes**. Then, repeat the same process in the **shaders** folder: right-click and select:

- Create New > Resource > Shader

Name the shader **anime\_eyes** as well to maintain a clear connection between the two files and make them easier to identify. If everything was done correctly, your project structure should now look like this:



(1.8.b The downloadable files have been added to the project)

Now that you have the necessary files, locate the **anime\_eye.fbx** object inside the **meshes** folder. Right-click on it and select **New Inherited Scene**. This action will allow you to create a new scene based on the original model, preserving its imported structure and properties.

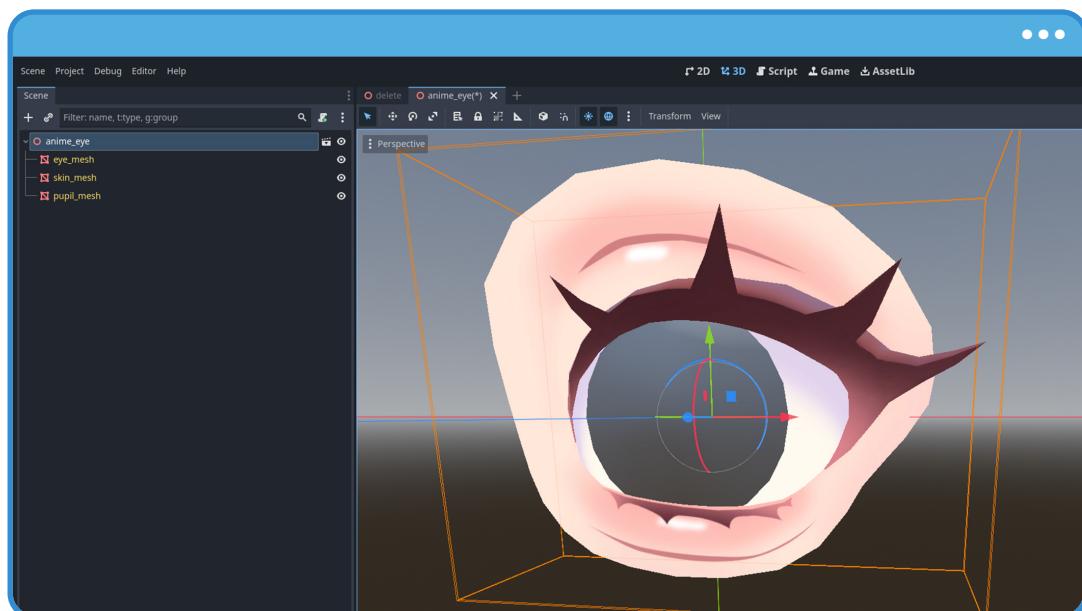
Inside the **Scene** panel, you'll find the three layers associated with the 3D model: **eye**, **skin**, and **pupil**. Assign the **anime\_skin** material to both the **eye** and **skin** layers. Then, apply the **anime\_eyes** material to the **pupil** layer, as this is where you'll develop the effect.

Make sure you have already linked the appropriate shader to the material by using the **Shader** property.

**Note**

To assign a shader to a material: 1) select the material, 2) double-click it to open its properties, 3) assign the shader through the Shader property.

If everything was set up correctly, your scene should now look like this:



(1.8.c The appropriate materials have been assigned to each 3D layer)

Before moving on, there are a couple of important details you should check regarding the **anime\_eye** object in your scene. Based on the current lighting setup:

- ① The material applied to the skin and eye is not being affected by light.
- ② Meanwhile, the material on the pupil is reacting to light, giving it a grayish tint – when in fact, it should appear completely white by default.

Additionally, you'll notice that the pupil doesn't display any visible texture yet. This is expected since you haven't declared a `sampler2D` in the shader to link an external texture.

Next, you'll implement these initial elements, as shown in the following code snippet:

```
shader_type spatial;
render_mode unshaded;

uniform sampler2D _MainTex : source_color;

void vertex()
{
    // Called for every vertex the material is visible on.
}

void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}
```

If you look closely at the previous code block, you'll notice that the second line sets the `render_mode` to `unshaded`. According to the official Godot documentation:

“

The result is only albedo. No lighting or shading is applied to the material, which speeds up rendering.

[https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/spatial\\_shader.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/spatial_shader.html)

”

In practical terms, this means that the unshaded render mode disables all lighting calculations. This can be especially beneficial when optimizing performance — for example, in mobile games targeting mid to low-end devices.

However, this setting also means that any lighting or depth effects must be manually simulated within the shader. In this particular case, that limitation isn't an issue, since your goal

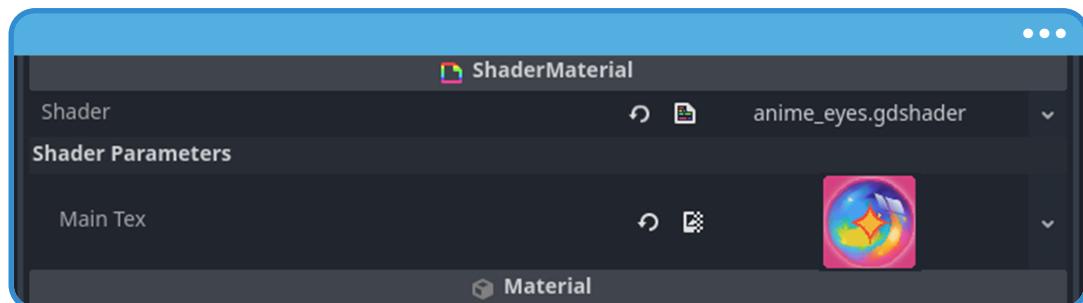
is to clearly visualize the displacement effect applied in tangent space. That's why using unshaded here is completely appropriate: it allows you to focus solely on the visual behavior you're trying to achieve without distractions from additional lighting computations. This not only simplifies implementation, but also helps you better understand the concept you're exploring.

**Note**

In Godot, **spatial** shaders allow you to configure multiple render modes. One of these is **cull\_disabled**, which enables rendering on both the inside and outside faces of a 3D object. For more details about the different available modes, you can refer to the official documentation here: [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/spatial\\_shader.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/spatial_shader.html)

With this small setup complete, you can now go to the **anime\_eyes** material and assign a texture to the **Main Tex** property. For this exercise, you can use any of the following textures included in the project:

- eye\_colorful\_color.
- eye\_green\_color.
- eye\_purple\_color.



(1.8.d **Main Tex** set to **eye\_colorful\_color**)

If everything has been set up correctly, the pupil object should now display the assigned texture. However, at this stage, no displacement effect is visible yet. If you observe the **albedo** RGB vector, you'll notice that the texture is simply being projected onto the surface using the basic UV coordinates.

To create a displacement effect, you'll need to complete a few key steps:

- Define the tangent space for the pupil's geometry.
- Transform the camera's **VIEW** direction into tangent space by calculating the dot product with each of the tangent space basis vectors.
- Distort the UV coordinates using the result of this transformation, creating a new dynamic projection that reacts to the viewer's perspective.

You'll begin by declaring a method that transforms the view vector into tangent space, as shown below:

```
vec3 view_to_tangent(vec3 tangent, vec3 bitangent, vec3 normal, vec3 view)
{
    float t = dot(tangent, view);
    float b = dot(bitangent, view);
    float n = dot(normal, view);
    return normalize(vec3(t, b, n));
}

void fragment() { ... }
```

What does this method do? To fully understand how this method works, you first need to analyze the **dot()** function. This function, widely used across programming languages, calculates the dot product between two vectors. In this specific case, it operates on three-dimensional vectors, and the result is a scalar value, which is then assigned to the variables **t**, **b**, and **n** based on the corresponding base vector — **tangent**, **bitangent**, and **normal**.

Let's begin by reviewing its algebraic definition for three-dimensional vectors:

$$A \cdot B = \sum_{i=1}^3 A_i B_i$$

(1.8.e)

Which is the same as,

$$A \cdot B = (A_1 * B_1) + (A_2 * B_2) + (A_3 * B_3)$$

(1.8.f)

The equivalent implementation in GLSL can be represented like this:

```
float dot(vec3 a, vec3 b)
{
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

In this context, the `view_to_tangent()` method projects the `view` vector (the direction from the fragment toward the camera) onto the orthonormal basis formed by the `tangent`, `bitangent`, and `normal` vectors. The result is a new vector expressed in tangent space, allowing you to work within this coordinate system to apply visual effects such as displacement, shading, or simulated depth.

Now, if you look at the first operation inside the method, both the `tangent` vector and the `view` vector must be expressed in **view space**. But how can you be sure that both vectors are indeed in that space?

According to the official Godot documentation, under the **Fragment Built-ins** section:

“

**VIEW:** A normalized vector from the fragment (or vertex) position to the camera (in view space). It is the same for both perspective and orthogonal cameras.

”

In simple terms, **VIEW** is a vector that points from the fragment being processed toward the camera, and it's expressed in the camera's reference space. This vector is automatically interpolated from the vertices during the rasterization process, and it's particularly useful for effects that depend on the viewpoint, such as the displacement effect you're implementing.

Additionally, the documentation states:

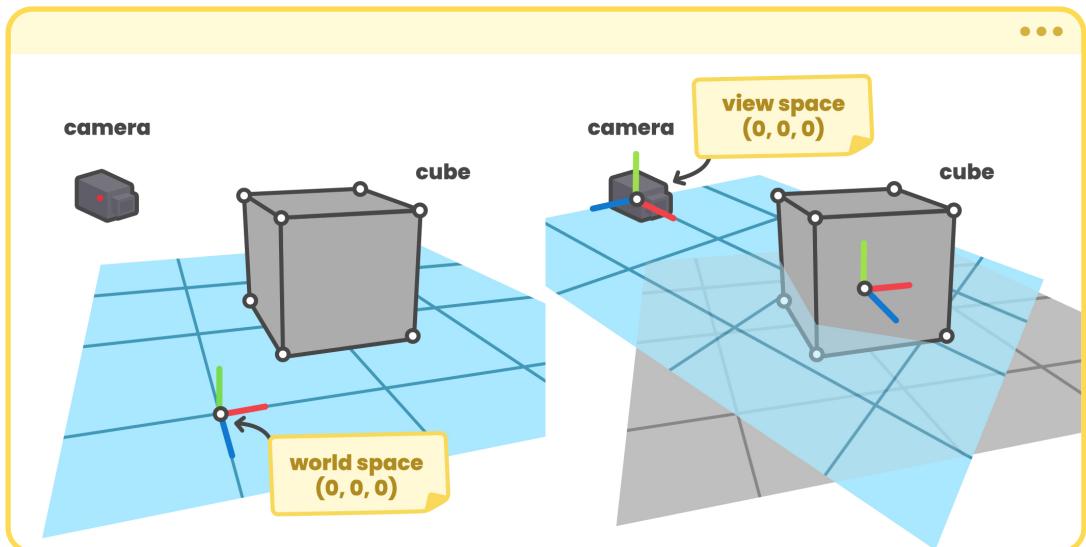
“

**TANGENT**: The tangent coming from the `vertex()` function, in view space. If `skip_vertex_transform` is enabled, it may not be in that space.

”

Considering that you haven't applied any custom transformations to the tangent inside the `vertex()` method, you can safely assume that the internal **TANGENT** variable is already in view space, just like **VIEW**.

This leads to a new question: how can you visualize this space? To answer that, let's take a closer look at the following visual reference:



(1.8.g Difference between world space (left) and view space (right))

Starting with the image on the left, you can observe that the cube is defined in world space, meaning its position and orientation are determined relative to the origin point in the **Viewport**. In contrast, in the image on the right, the cube is defined in view space. This means its position and orientation are no longer expressed relative to the world but from the camera's perspective. As a result, its rotation no longer matches exactly  $(0_x, 0_y, 0_z)$  because its frame of reference has changed.

The same principle applies to the **tangent**, **bitangent**, and **normal** vectors used in each fragment within the **view\_to\_tangent()** function. All of them must exist within the same reference space — in this case, view space — for vector operations like the dot product to be mathematically valid and produce correct results.

**Note**

From a Technical Artist's perspective, it's not strictly necessary to memorize every mathematical formula when implementing these effects. What matters most is understanding what each function does and making sure that all vectors involved are expressed in the same space before combining them in any operation.

Now that you understand how to transform the view direction into tangent space, you can apply this knowledge to distort the **UV** coordinates inside the **texture()** function. This will allow you to visualize the displacement effect on the pupil's texture.

To achieve this, you'll declare a new vector to store the transformation result. Then, you'll multiply this vector by a scalar displacement value and apply it directly to the **UV** coordinates, as shown below:

```
vec3 view_to_tangent(vec3 tangent, vec3 bitangent, vec3 normal, vec3 view)
{ ... }

void fragment()
{
    float offset = 0.5;
    vec3 view_tangent_space = view_to_tangent(-TANGENT, BINORMAL, NORMAL,
        ↵ VIEW);
    view_tangent_space *= offset;

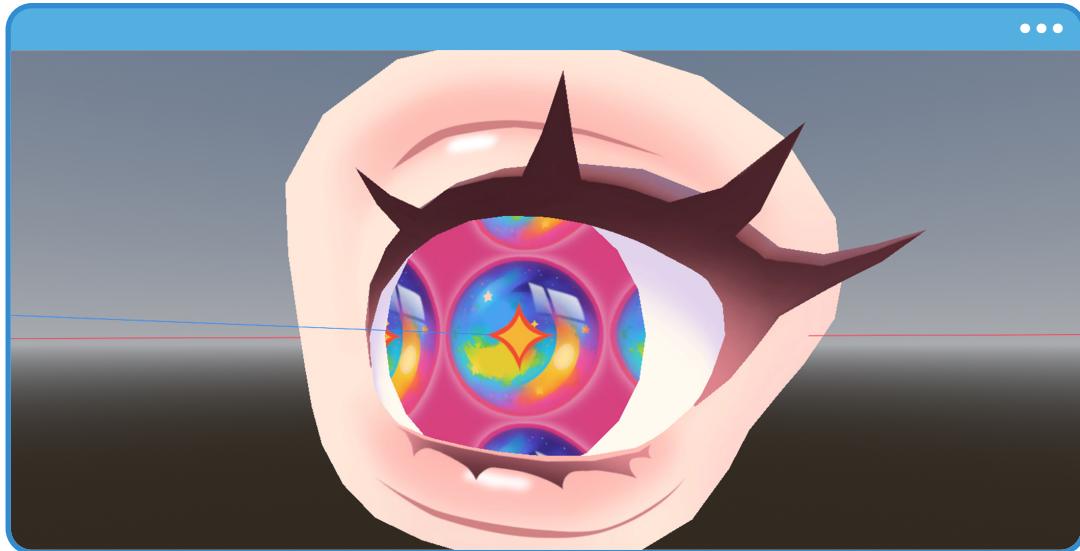
    vec3 albedo = texture(_MainTex, UV + view_tangent_space.xy).rgb;
    ALBEDO = albedo;
}
```

In the previous code, three key actions take place:

- 1 A scalar value called **offset** has been declared and set to 0.5. This value is used for demonstration purposes only, as you'll later adjust it to a maximum of 0.3 to achieve a more controlled result.
- 2 A new three-dimensional vector called **view\_tangent\_space** has been declared and initialized with the result of the **view\_to\_tangent()** method that transforms the view vector into tangent space.
- 3 The first two components XY of the **view\_tangent\_space** vector have been taken and added to the original UV coordinates inside the **texture()** function, generating a visual displacement effect on the texture.

If you pay close attention to the first argument passed to the **view\_to\_tangent()** function, you'll notice that the **TANGENT** variable is negated. Although this might seem counterintuitive at first, there's a logical reason behind it: Godot uses a right-handed spatial coordinate system, where the *z*-axis points forward. In this system, it's necessary to invert the tangent vector's direction so that the tangent-bitangent-normal (TBN) basis is constructed correctly. In contrast, engines like Unity use a left-handed system, where the **TANGENT** vector can be used without inversion.

By adding the first two components of the **view\_tangent\_space** vector to the UV coordinates, you are projecting the view direction onto the plane defined by the tangent and bitangent vectors, which are aligned with the UV space. Since this vector represents the direction toward the camera in tangent space, the visual result is a dynamic texture displacement that simulates depth on the object's surface. This technique enhances the visual perception of depth without altering the model's actual geometry, achieving a convincing effect at a low computational cost.

(1.8.h The **Main Tex** texture is set to repeat mode)

If you return to the **Viewport**, you'll notice two main things:

- ① The pupil now displays an effective depth effect.
- ② However, the texture appears repeated, creating an undesirable visual result.

Although one possible solution would be to manually disable the **Repeat** mode on the **eye\_colorful\_color** texture, in this case, you'll adopt a more robust approach suited for shader-controlled environments. Since you are implementing this behavior directly within the shader code, you'll use the **repeat\_disable** hint next to the **sampler2D** declaration for **\_MainTex**. This will prevent the UV coordinates from automatically wrapping when exceeding the standard range [0.0 : 1.0].

Additionally, to allow greater flexibility from the editor, you'll declare a new uniform variable called **\_Offset**, with an adjustable range between [0.0 : 0.3]. This variable will let you dynamically modify the displacement intensity directly from the **Inspector**.

The updated shader will look like this:

```
uniform sampler2D _MainTex : source_color, repeat_disable;
uniform float _Offset : hint_range(0.0, 0.3, 0.05);

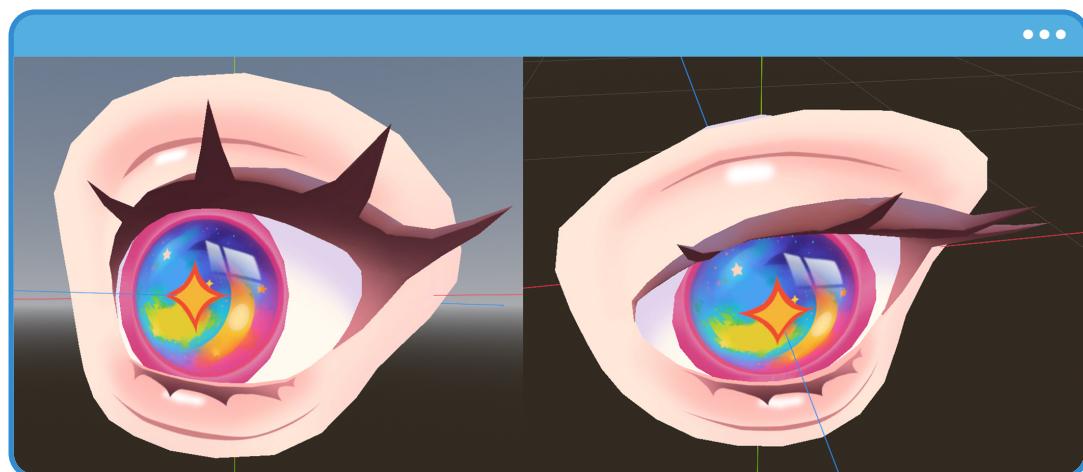
void vertex() { ... }

vec3 view_to_tangent(vec3 tangent, vec3 bitangent, vec3 normal, vec3 view)
{ ... }

void fragment()
{
    float offset = _Offset;
    vec3 view_tangent_space = view_to_tangent(-TANGENT, BINORMAL, NORMAL,
        ↵ VIEW);
    view_tangent_space *= offset;

    vec3 albedo = texture(_MainTex, UV + view_tangent_space.xy).rgb;
    ALBEDO = albedo;
}
```

With a displacement value set to 0.2, the pupil should now look like this:



(1.8.i Depth effect on the pupil)

At this point, you could consider the effect complete, as it successfully fulfills its main purpose. However, even though the displacement creates a sense of depth, the result still feels somewhat flat due to the lack of visual contrast or layering. For this reason, in the next section, you'll spend a few more minutes enhancing the effect by adding additional elements that will enrich the visual perception and increase the overall realism of the final result.

## 1.9 Linear Interpolation Between Colors.

Up to this point, the displacement effect is working correctly. However, it's being applied uniformly across the entire texture, when ideally it should only affect the pupil – the central circular area.

To fix this behavior, you can simply introduce a mask: a new texture where the area you want to displace is white, and the rest is black. Why use these colors? Because mathematically, white represents a value of 1.0, while black represents 0.0. Therefore, by multiplying this value by your displacement amount, you can achieve gradual control over the effect: it will be fully applied in the white areas (the pupil) and suppressed in the black areas (the rest of the eye).

To implement this improvement, you'll extend the shader by adding a new uniform texture called `_Depth`, as shown below:

```

uniform sampler2D _MainTex : source_color, repeat_disable;
uniform sampler2D _Depth : source_color, repeat_disable;
uniform float _Offset : hint_range(0.0, 0.3, 0.05);

void vertex() { ... }

vec3 view_to_tangent(vec3 tangent, vec3 bitangent, vec3 normal, vec3 view)
{ ... }

void fragment()
{
    float depth = texture(_Depth, UV).r;
    float offset = _Offset * depth;
    vec3 view_tangent_space = view_to_tangent(-TANGENT, BINORMAL, NORMAL,
        ↵ VIEW);
    view_tangent_space *= offset;

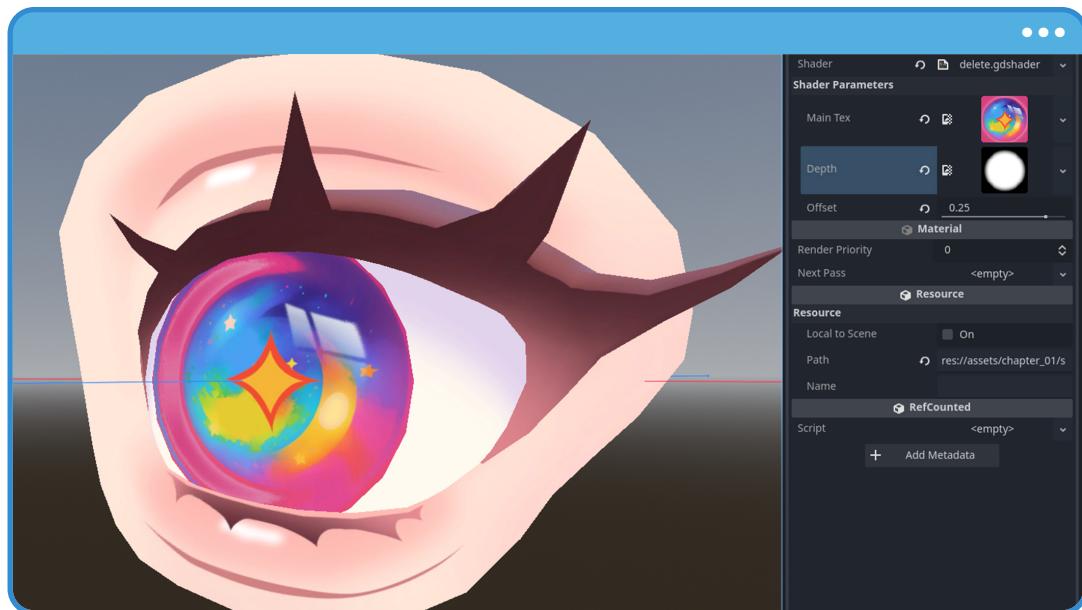
    vec3 albedo = texture(_MainTex, UV + view_tangent_space.xy).rgb;
    ALBEDO = albedo;
}

```

As you can see, a new texture called **\_Depth** has been introduced, acting as a mask to control the displacement effect. This texture should contain white values in the region you want to affect — the pupil — and black values in the surrounding areas. Since you’re working with a grayscale image, the red R channel alone is sufficient to represent the intensity at each point of the texture.

Inside the **fragment()** method, you access this value using **texture(\_Depth, UV).r** and store the result in the **depth** variable. This value determines how much displacement should be applied to that specific fragment. You then multiply this value by the global displacement defined by **\_Offset**, generating a fragment-controlled variation: full displacement over the pupil and no displacement over the surrounding areas.

In this way, the displacement is no longer applied uniformly across the entire surface but only in the areas you define through the mask. To see this in action, make sure you assign the **eye\_depth** texture to the **Depth** property in the **Inspector**.



(1.9.a The **eye\_depth** texture has been assigned to the **Depth** property)

Once you've correctly assigned the mask, you'll notice a colored border appearing around the pupil, visually emphasizing the displacement effect. This color corresponds to the regions of the texture outside the white area defined by the mask — in other words, outside the pupil — and it shows how the distortion affects the surrounding texels.

Now is a great time to experiment with additional color layers to complement and enhance the visual effect. For example, you could introduce extra highlights or decorative details to reinforce the sense of depth and direction. To achieve this, you'll add two new global textures to the shader, as shown below:

```
uniform sampler2D _MainTex : source_color, repeat_disable;
uniform sampler2D _Depth : source_color, repeat_disable;
uniform sampler2D _Border : source_color, repeat_disable;
uniform sampler2D _Shine : source_color, repeat_disable;
uniform float _Offset : hint_range(0.0, 0.3, 0.05);

void vertex() { ... }
```

For the `_Border` and `_Shine` textures, you'll use the `eye_border` and `eye_shine` textures respectively, which must be assigned from the **Inspector**. Once these textures are applied and the necessary adjustments are made in the shader, you'll be able to visualize the effect directly in the **Viewport**, allowing for quick iteration and fine-tuning.

With the textures now declared, the next step is to blend them with the base color of the pupil. To do this, you'll use linear interpolation through the `mix()` function. This function, commonly used in GLSL, allows you to combine two values (in this case, colors or vectors) based on a third scalar value that acts as the blending factor.

The basic definition of `mix()` for a three-dimensional vector is as follows:

```
vec3 mix(vec3 a, vec3 b, float w)
{
    return a + w * (b - a);
}
```

In this context:

- The first argument, `vec3 a`, is the original value (for example, the base color from the `albedo` texture).
- The second argument, `vec3 b`, is the value you want to introduce (for example, the border or shine).
- The third argument, `float w`, is a value between 0.0 and 1.0 that defines how much `b` influences `a`.

When `w = 0.0`, the result is completely `a`. When `w = 1.0`, the result is completely `b`.

For any value in between, you get a smooth transition between the two. This allows you to combine additional visual effects in a controlled and coherent way across the surface of the eye.

Now, let's continue by extending the shader, starting with the implementation of the `_Border` texture, as shown below:

```
void fragment()
{
    float depth = texture(_Depth, UV).r;
    float offset = _Offset * depth;
    vec3 view_tangent_space = view_to_tangent(-TANGENT, BINORMAL, NORMAL,
        → VIEW);
    view_tangent_space *= offset;

    vec3 albedo = texture(_MainTex, UV + view_tangent_space.xy).rgb;
    vec4 border = texture(_Border, UV);
    vec3 lerp_ab = mix(albedo, border.rgb, border.a);

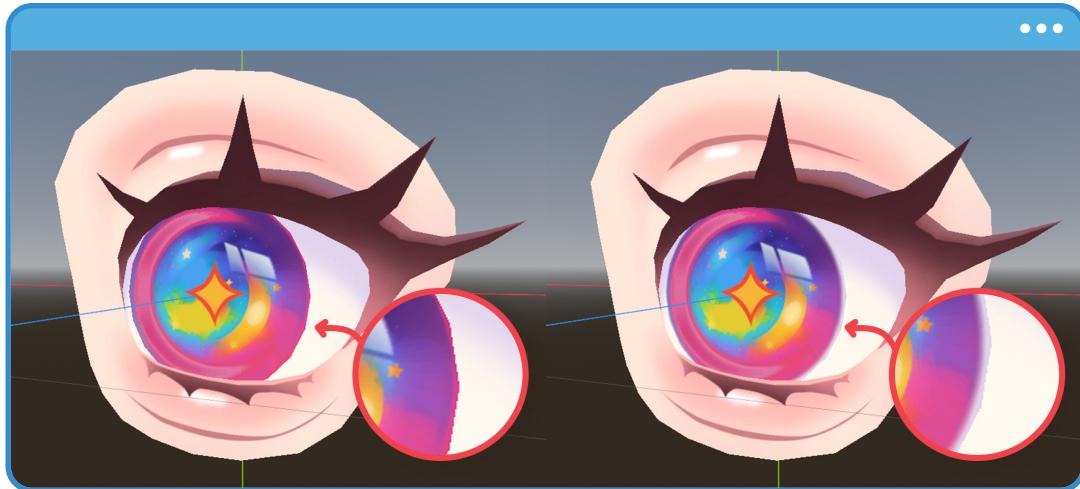
    ALBEDO = lerp_ab;
}
```

Before checking the result in the **Viewport**, let's take a moment to break down what's happening in the code. If you look at the `border` variable, you'll notice it's a four-component vector RGBA, which becomes important in the following operation.

In the next line, a vector called `lerp_ab` is declared, obtained through a linear interpolation between `albedo` (the base color from the texture) and `border.rgb` (the decorative color from the border texture). This interpolation is controlled using the **alpha channel** of the border texture — that is, `border.a`.

Graphically, this means:

- When `border.a = 0.0`, the resulting color is completely `albedo`.
- When `border.a = 1.0`, the resulting color is completely `border.rgb`.
- For intermediate values, a smooth blend between both colors is produced.



(1.9.b Comparison between base **albedo** and linear interpolation across two layers)

In the **Viewport**, you can now observe a subtle gradient between the pupil and the rest of the eye. This gradient, both in shape and color, comes from the **eye\_border** texture, which has been aesthetically adapted to blend into the overall composition. I recommend experimenting with different colors in this texture to see the principle of linear interpolation in action and observe how the final result changes.

The only step left is to apply the final visual layer: the **eye\_shine** texture. To do this, you'll repeat the same process using another **mix()** function, as shown below:

```

void fragment()
{
    float depth = texture(_Depth, UV).r;
    float offset = _Offset * depth;
    vec3 view_tangent_space = view_to_tangent(-TANGENT, BINORMAL, NORMAL,
        ↵ VIEW);
    view_tangent_space *= offset;

    vec3 albedo = texture(_MainTex, UV + view_tangent_space.xy).rgb;
    vec4 border = texture(_Border, UV);
    vec4 shine = texture(_Shine, UV);

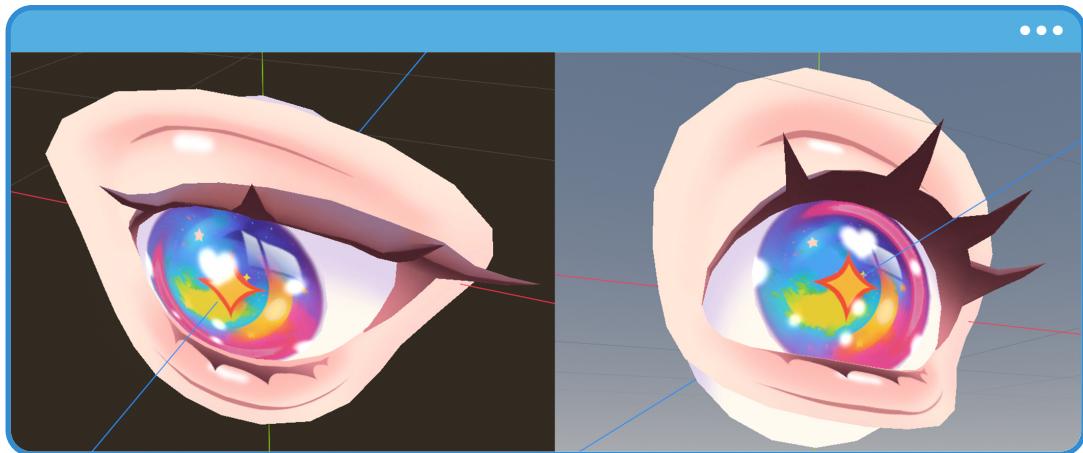
    vec3 lerp_ab = mix(albedo, border.rgb, border.a);
    vec3 lerp_abs = mix(lerp_ab, lerp_ab + shine.rgb, shine.a);

    ALBEDO = lerp_abs;
}

```

As you can see, at this stage a second interpolation has been applied. You start with the previous result (`lerp_ab`, which blends the base color and the border) and combine it with the sum of `lerp_ab + shine.rgb`. This addition strengthens the highlight effect, allowing it to overlay smoothly onto the original color based on the intensity defined by the alpha channel of the `eye_shine` texture.

Finally, if you return to the **Viewport**, you'll be able to see the complete effect: the displacement applied to the pupil along with the layered decorative borders and highlights – all controlled through masks and the principles of linear interpolation.



(1.9.c Textures working together)

## 1.10 Introduction to Matrices.

As you dive deeper into shader development, you'll frequently encounter the concept of **matrices**. A matrix is a numerical structure made up of elements arranged in rows and columns, following specific arithmetic and algebraic rules. In the context of graphics programming, matrices play a crucial role in performing spatial transformations such as **translation**, **rotation**, and **scaling**.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

```
mat3 matrix = mat3
(
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0),
);
```

(1.10.a On the left: a mathematical 3x3 matrix. On the right: a 3x3 matrix in .gdshader)

In shaders, matrices are primarily used to transform vertex positions from one space to another. For instance, the reason you're able to view a 3D object on a 2D screen is because

its position has been transformed through a series of matrices — most commonly referred to as **MODEL\_MATRIX**, **VIEW\_MATRIX**, and **PROJECTION\_MATRIX**.

This transformation process happens automatically during the rendering pipeline. However, you can also use these matrices directly in your shader code to apply custom transformations, dynamically modify vertex positions, or create camera-dependent visual effects like the **billboard** effect.

But how exactly does this process work? Let's take a simple primitive — a cube — as an example. At its core, the cube is just a collection of numerical data. There's no actual 3D object inside the computer. These numbers represent the cube's geometry in **local space**. So, the first logical step is to transform this data into **world space** using the **MODEL\_MATRIX**. According to Godot's official documentation:

“

Model/local space to world space transform.

”

```
Normals : VERTEX
{
    -0.5  0.5 -0.5,
    -0.5  0.5  0.5,
    -0.5 -0.5 -0.5,
    -0.5 -0.5  0.5,
    0.5  0.5 -0.5,
    0.5  0.5  0.5,
    0.5 -0.5 -0.5,
    0.5 -0.5  0.5,
}
```

MODEL\_MATRIX

(1.10.b The vertices in local space have been transformed to world space)

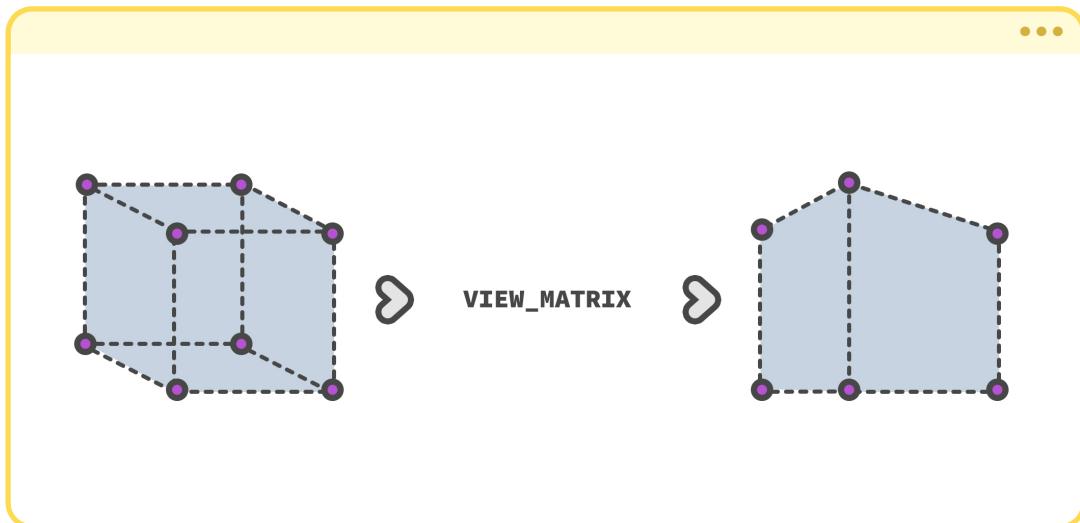
Once the data has been transformed into world space, the cube can be correctly positioned within the scene. However, it still isn't visible — there's no point of view from which to observe it. It's as if the cube exists, but your eyes are closed.

To make it visible, you need to transform its position from world space to view space using the **VIEW\_MATRIX**, which:

“

World space to view space transform.

”



(1.10.c The vertices in world space have been transformed to view space)

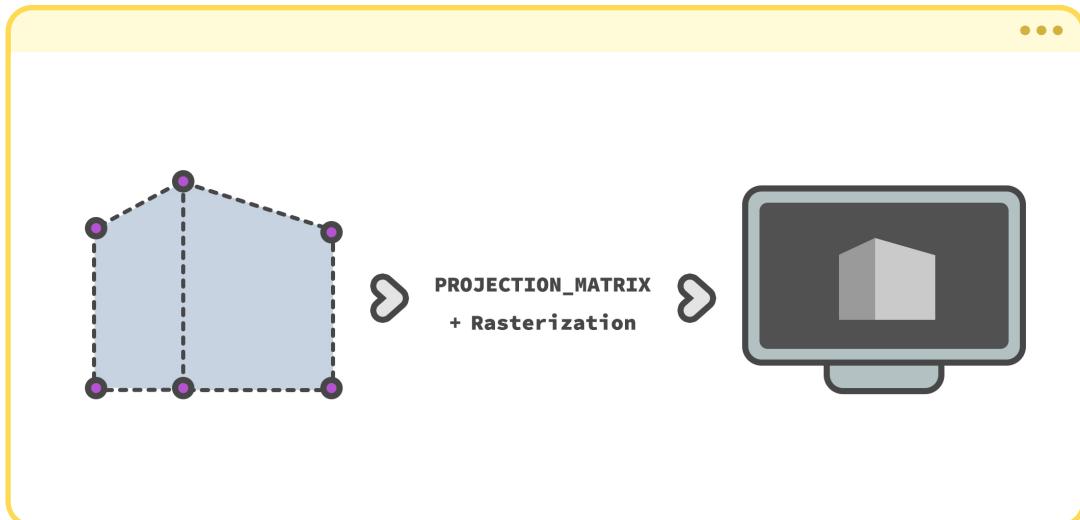
At this stage, the object can already be represented from the camera’s perspective. However, to actually display it on your computer screen, its coordinates must be converted from view space to projected space. This is achieved through the **PROJECTION\_MATRIX**, which, according to Godot’s official documentation:

“

View space to clip space transform

”

This projected space – also known as clip space – is the final stage before the rasterization process, which ultimately converts data into the pixels you see on the screen. Therefore, you can assume that this entire transformation process takes place within the **vertex()** function.



(1.10.d The cube has been projected onto a two-dimensional screen)

#### Note

In Godot, you'll find a matrix called **MODELVIEW\_MATRIX**. This matrix combines both the model (**MODEL\_MATRIX**) and view (**VIEW\_MATRIX**) transformations. As a result, it transforms a point directly from local space to view space. [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/spatial\\_shader.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/spatial_shader.html)

Understanding matrices can be challenging at first. That's why, in the following sections, you'll spend some time implementing each of these matrices to get a clearer picture of how they work. You'll also create custom matrices that allow you to dynamically transform your object's vertices directly from the Inspector.

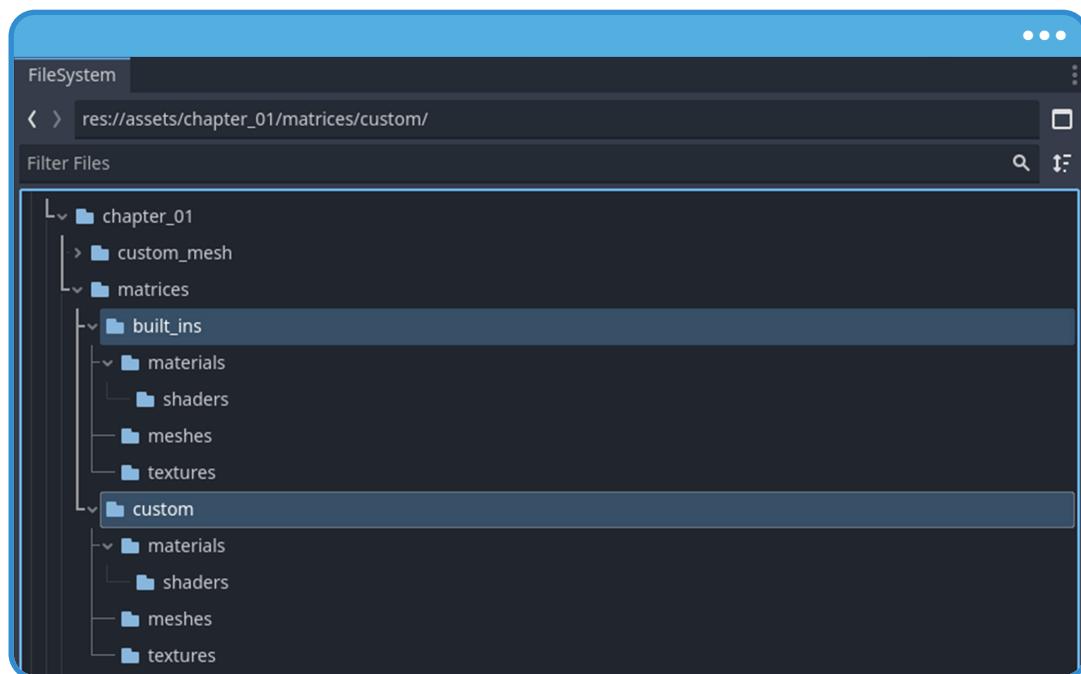
## 1.11 Built-in Matrices.

In this section, you'll work through a new hands-on exercise to deepen your understanding of the internal matrices: **MODEL\_MATRIX**, **VIEW\_MATRIX**, and **PROJECTION\_MATRIX**, and how they relate to the **VERTEX** variable. You'll also create a visual effect known as **Billboard**, using the **INV\_VIEW\_MATRIX**.

To keep implementation easier and keep things organized, structure your project by following these steps:

- 1 Inside your project, navigate to the **chapter\_01** folder and create a new folder called **matrices**.
- 2 Since this section will be divided into two parts, create two subfolders within **matrices**:
  - a **built\_ins**.
  - b **custom**.
- 3 Inside each of these folders, organize the content by creating the following subfolders:
  - a **materials**.
  - b **shaders**.
  - c **meshes**.
  - d **textures**.

If you've followed these steps correctly, your project structure should look like this:



(1.11.a The **built\_ins** and **custom** folders have been added to the project)

As mentioned earlier, you'll implement the **Billboard** effect using a shader. This effect automatically orients a 3D object (or QuadMesh) to always face the camera, no matter

where it is in the world. A great example of this is the question mark symbol on the Item Box in Mario Kart – no matter where you stand, the symbol always faces the camera.

In your case, you'll implement this behavior directly in the `vertex()` function by adjusting the object's orientation based on the camera's position. To start, let's create a few folders to keep your project organized.

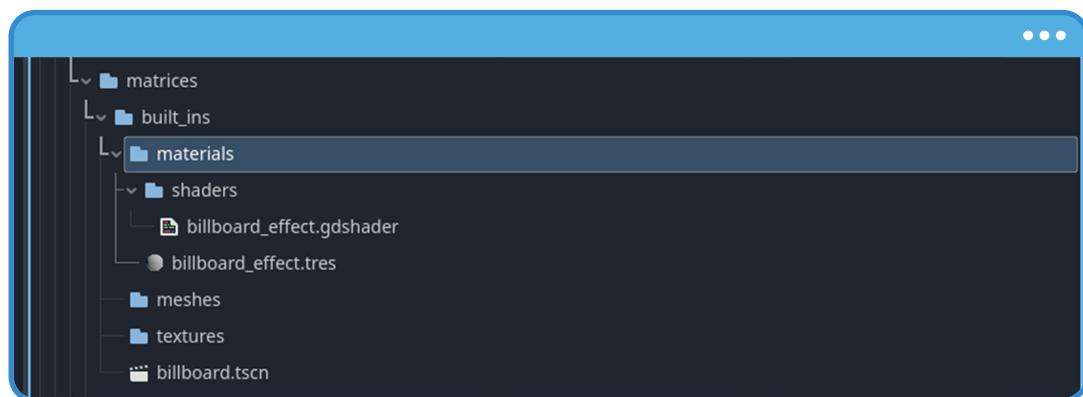
Inside the **built\_ins** > **materials** folder, right-click and select:

- Create New > Resource > ShaderMaterial.

For convenience, name this material **billboard\_effect**. Then, repeat the process inside the Shaders folder: right-click and select:

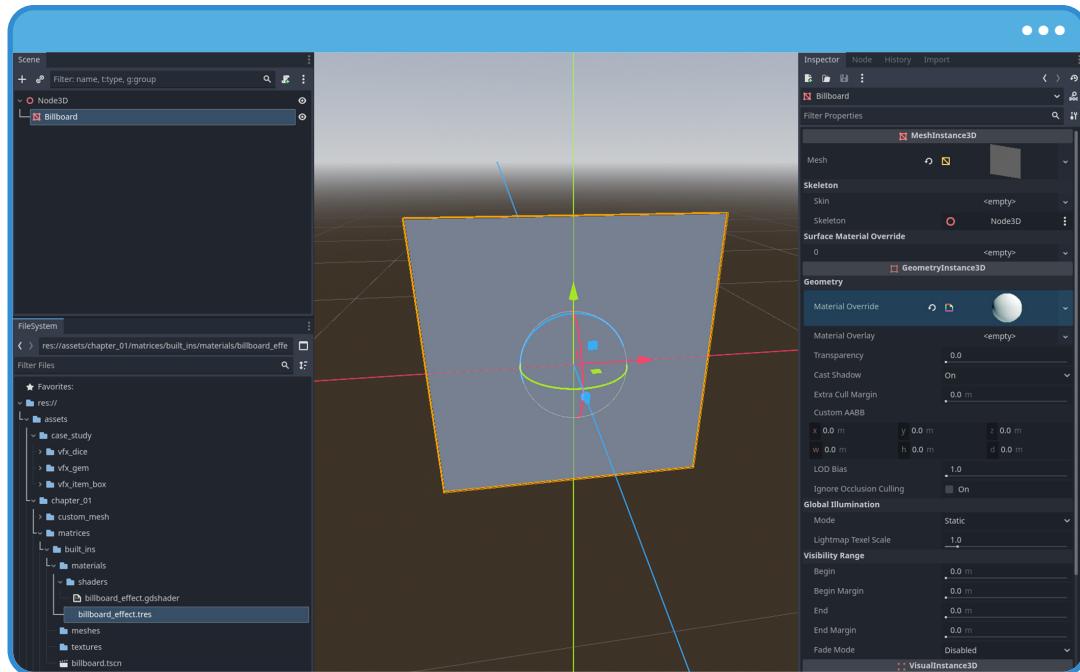
- Create New > Resource > Shader.

As usual, make sure you give the shader the same name as the material. This naming convention makes it easier to identify the connection between the two. If you've followed the steps correctly, your project structure should now look like this:



(1.11.b Both the materials and shader have been added to the project)

To begin the implementation process, start by creating a new 3D scene and add a **MeshInstance3D** node as its child. Name this node Billboard. Then, assign a **QuadMesh** to its **Mesh** property. After creating the **billboard\_effect** shader and its corresponding material, make sure to assign the material to the **Material Override** property of the **Billboard** object.

(1.11.c A **QuadMesh** has been added to the scene)

Now, turn your attention to the **billboard\_effect** shader – specifically, the **vertex()** method. At first glance, you'll notice that the function appears empty. However, as you've learned, several mathematical operations are performed automatically behind the scenes, such as matrix multiplications. If you check Godot's official documentation, you'll find the following explanation:

“

Users can override the modelview and projection transforms using the **POSITION** built-in. If **POSITION** is written to anywhere in the shader, it will always be used, so the user becomes responsible for ensuring that it always has an acceptable value. When **POSITION** is used, the value from **VERTEXPOSITION** is ignored and projection does not happen.

”

To see this in action, you'll use the **render\_mode skip\_vertex\_transform**. This mode disables Godot's automatic vertex transformations within the **vertex()** function, giving you full control over how vertices are positioned and projected. However, before applying this setting,

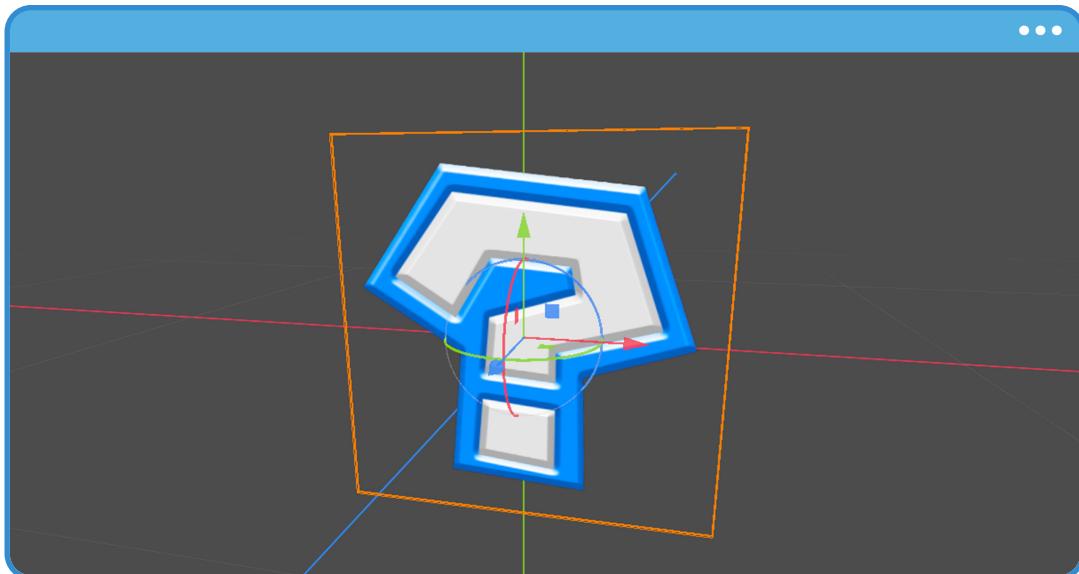
make sure to configure and assign a texture to the shader. This will prevent the QuadMesh from appearing completely white or empty during the exercise.

```
shader_type spatial;  
render_mode unshaded;  
  
uniform sampler2D _MainTex : source_color;  
  
void vertex() { ... }  
  
void fragment()  
{  
    vec4 albedo = texture(_MainTex, UV);  
    ALBEDO = albedo.rgb;  
    ALPHA = albedo.a;  
}
```

In the previous code snippet, a new variable named **albedo** was declared as a **vec4** (representing RGBA or XYZW). This variable stores the result of sampling the **\_MainTex** texture using the **UV** coordinates. As you already know, this texture will be assigned later through the Inspector. Unlike the examples in previous sections, this time you're not only using the texture's RGB color channels but also its alpha channel, which is assigned directly to the built-in **ALPHA** variable.

The **ALPHA** variable controls the transparency of the fragment. This allows pixels in the texture to appear fully opaque, partially transparent, or completely invisible depending on the alpha values defined in the image. For this exercise, you'll use the texture named **item\_box\_symbol\_tex**, which is included in the downloadable package attached to this book.

Once you've saved the shader and assigned the texture, the **Billboard** object in your scene should look like this:



(1.11.d The texture has been assigned to the Billboard object)

Next, you'll add the `skip_vertex_transform` property to your shader using the following code:

```
shader_type spatial;
render_mode unshaded;
render_mode skip_vertex_transform;

uniform sampler2D _MainTex : source_color;

void vertex() { ... }
```

Once you add this line, your QuadMesh will disappear from the **Viewport**. This happens because no vertex transformations are being performed – so nothing is being rendered. As a result, you'll need to manually handle the vertex transformations. To begin, transform the mesh's vertices from model space to view space using the **MODELVIEW\_MATRIX**, as shown below:

```

render_mode skip_vertex_transform;

uniform sampler2D _MainTex : source_color;

varying vec3 vertex_os;

void vertex()
{
    vertex_os = VERTEX;
    VERTEX = (MODELVIEW_MATRIX * vec4(vertex_os, 1.0)).xyz;
}

```

There are a few important concepts you need to understand in order to interpret the following code correctly. Let's start with the `varying` keyword. This is used to define a global variable that is shared between the `vertex()` and `fragment()` functions. In this case, the `vertex_os` vector can be accessed in both stages of the shader, allowing data to be passed from the `vertex` function to the `fragment` function when needed.

In the first line of the `vertex()` method, the `vertex_os` vector is assigned the value of `VERTEX`. This means the vertex data is still in **object space** — hence the `_os` suffix in the variable name.

#### Note

It's considered good practice not only to use descriptive variable names in your shaders but also to include suffixes that indicate the space the variable belongs to. For example: `variable_os` for object space, `variable_ws` for world space, `variable_vs` for view space, `variable_ss` for screen space.

Next, the `VERTEX` variable is transformed by multiplying the `MODELVIEW_MATRIX` with a four-dimensional vector. The first three components (XYZ) represent the original position of the vertex (`vertex_os`), while the fourth component (W) is set to 1.0.

Why this value? The 1.0 represents a position vector in homogeneous space. In computer graphics, 4D vectors (`vec4`) are used to allow for affine transformations through  $4 \times 4$  matrix multiplication. Setting `W = 1.0` tells the GPU that the vector is a position in space, meaning it will be affected by translation, rotation, and scaling. On the other hand, if `W = 0.0`, the vector is treated as a direction, which will be affected by rotation and scaling — but not by

translation. Since vertices represent actual positions in 3D space, it's essential to ensure that W is set to 1.0.

You can achieve the same result by separating the transformation matrices, as shown below:

```
void vertex()
{
    vertex_os = VERTEX;
    vec4 vertex_ws = MODEL_MATRIX * vec4(vertex_os, 1.0);
    vec4 vertex_vs = VIEW_MATRIX * vertex_ws;
    VERTEX = vertex_vs.xyz;
}
```

In this code snippet, as you can see, the transformation process involves only the **MODEL\_MATRIX** and **VIEW\_MATRIX**. However, projection is still handled automatically as part of the final transformation. To take full control — including the projection — you'll need to use the built-in **POSITION** variable, as shown below:

```
void vertex()
{
    vertex_os = VERTEX;
    vec4 vertex_ws = MODEL_MATRIX * vec4(vertex_os, 1.0);
    vec4 vertex_vs = VIEW_MATRIX * vertex_ws;
    vec4 vertex_proj = PROJECTION_MATRIX * vertex_vs;
    POSITION = vertex_proj;
}
```

If you choose this approach, you must ensure that **POSITION** always contains valid values. This adds an extra level of responsibility and increases your workload when writing shaders.

Now that you understand this, how can you apply the **Billboard** effect directly in the shader? The core idea of this effect is to make the object always face the camera, regardless of its original orientation in world space.

To achieve this, you can build a custom transformation matrix that combines the object's world position with the camera's orientation. You'll extract the first three rows of the **INV\_VIEW\_MATRIX**, which, according to Godot's official documentation:

“

View space to world space transform.

”

This approach keeps the vertices in world space, but oriented from the camera's point of view. By combining this custom orientation matrix with the last row of the **MODEL\_MATRIX**, which holds the object's position in world space, you can ensure the object always faces the camera.

Below is an example of how to apply this logic in the **vertex()** function, while preserving the previous implementation:

```
void vertex()
{
    vertex_os = VERTEX;

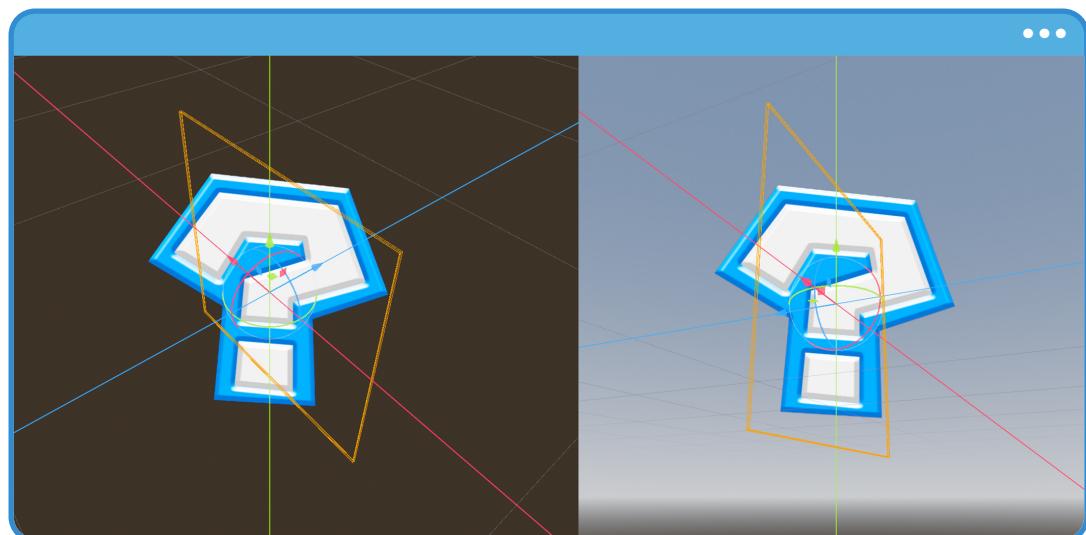
    mat4 BILLBOARD_MATRIX = mat4(
        INV_VIEW_MATRIX[0],
        INV_VIEW_MATRIX[1],
        INV_VIEW_MATRIX[2],
        MODEL_MATRIX[3]
    );

    vec4 vertex_ws = BILLBOARD_MATRIX * vec4(vertex_os, 1.0);
    vec4 vertex_vs = VIEW_MATRIX * vertex_ws;
    vec4 vertex_proj = PROJECTION_MATRIX * vertex_vs;
    POSITION = vertex_proj;
}
```

As shown in the example above, a new matrix called **BILLBOARD\_MATRIX** has been declared. This matrix reorients the object so that it always faces the camera. It is built by taking the first three rows of **INV\_VIEW\_MATRIX**, which represent the camera's orientation in world space.

This ensures that the vertices remain in their original space, but are reoriented from the camera's point of view. The fourth row is taken from **MODEL\_MATRIX**, preserving the object's world position.

If you've implemented everything correctly, your Billboard object should now face the camera in the **Viewport**, regardless of its position in the scene.



(1.11.e The Billboard object always faces the camera)

## 1.12 Implementing Custom Matrices.

Up to this point, you've focused on the **built-in matrices** that make it possible to render a 3D object in a scene. But what if you wanted to create your own matrices? According to Godot's official documentation, you can define matrices using the following data types:

- **mat2** for  $2 \times 2$  two-dimensional matrices.
- **mat3** for  $3 \times 3$  three-dimensional matrices.
- **mat4** for  $4 \times 4$  four-dimensional matrices.

Each type represents a different kind of transformation space and is used in different scenarios. For instance, you might define a matrix to scale, deform, or rotate the vertices of your models, whether in 2D or 3D. While a **mat3** might work in a 2D game, from a usability and

optimization standpoint, it's better to use a **mat2** — since you only need to transform the *x* and *y* axes in a 2D space.

So, how do you declare and initialize a matrix in shader code? According to Godot's documentation, you can do it as follows:

```
mat2 m2 = mat2(vec2(1.0, 0.0), vec2(0.0, 1.0));
mat3 m3 = mat3(vec3(1.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0), vec3(0.0, 0.0, 1.0));
mat4 identity = mat4(1.0);
```

These declarations create identity matrices of sizes  $2 \times 2$ ,  $3 \times 3$ , and  $4 \times 4$ , respectively. You can visualize them as follows:

For a two-dimensional matrix:

$$m_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

(1.12.a)

For a three-dimensional matrix:

$$m_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(1.12.b)

To define a four-dimensional matrix, simply extend the  $3 \times 3$  identity matrix by adding an extra row and column:

$$m_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(1.12.c)

**Note**

When you write `mat4 identity = mat4(1.0)`, Godot automatically fills the main diagonal with 1.0 and the remaining elements with zeros, creating an identity matrix. This is essential for performing linear transformations without altering the original vector.

As a hands-on exercise, you could create a rotation matrix to visualize the gimbal effect. However, we'll save that exercise for Chapter 3, where you'll explore and implement quaternions in `.gdshader`. That way, you'll have a clearer comparison between both methods. For now, let's focus on creating a small custom matrix that transforms only the vertices of your object.

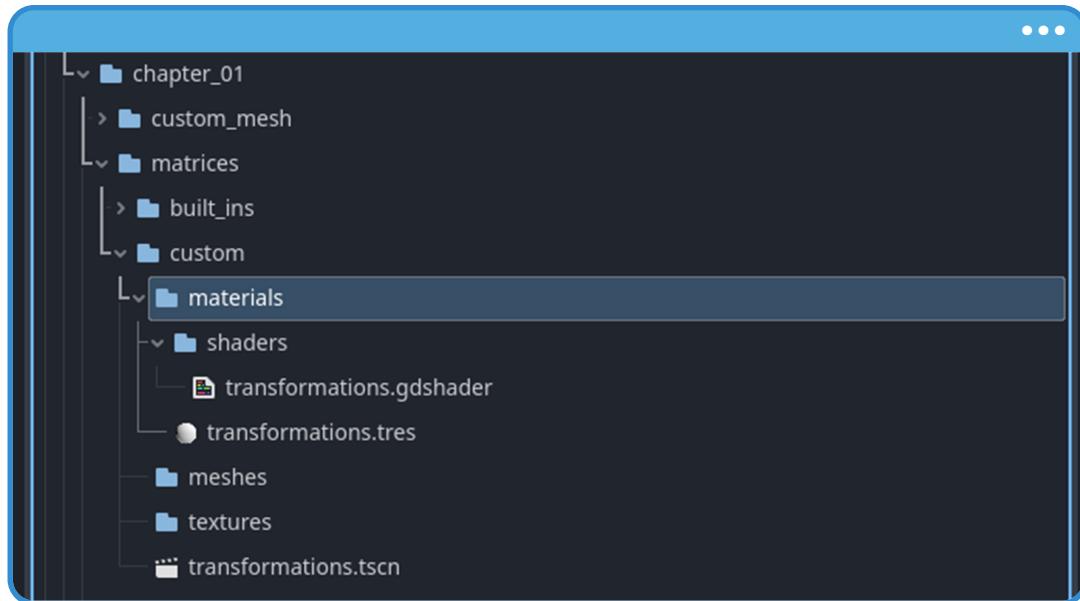
To begin, go to the **FileSystem**, and inside the folder **custom > materials > shaders**, create a new shader by selecting:

- Create New > Resource > Shader.

For practical purposes, name this shader **transformations**. Then, repeat the process in the **materials** folder: right-click and select:

- Create New > Resource > ShaderMaterial.

Make sure to give the material the same name as the shader to keep the connection between both resources clear. If you followed the steps correctly, your project should now look like this:



(1.12.d A material and a shader have been added to the project)

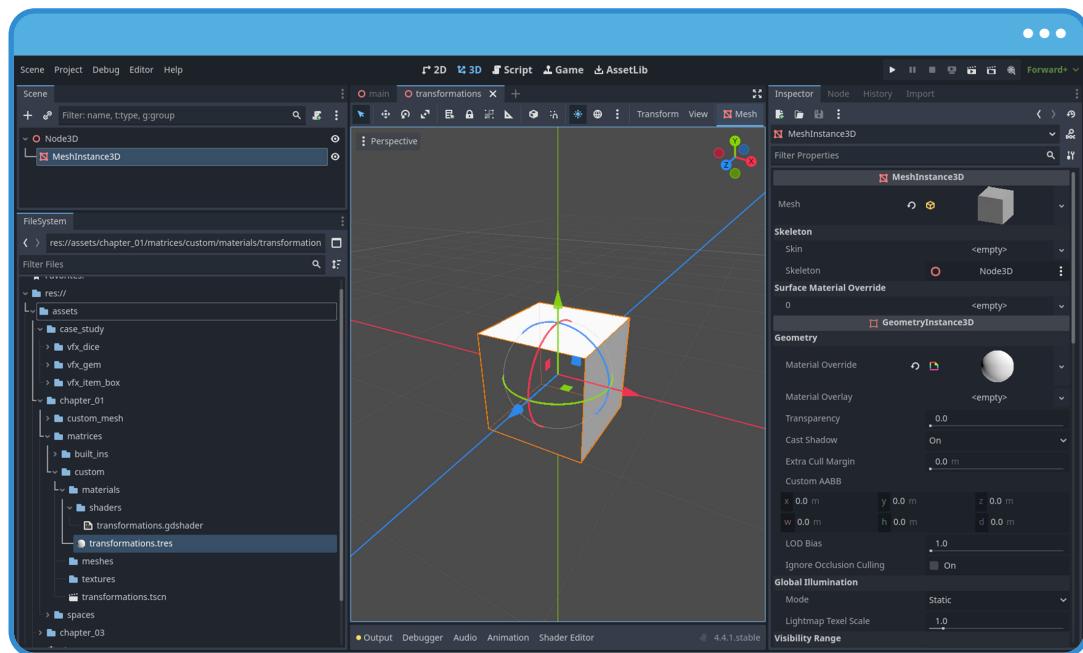
For the exercise you're about to complete, you can use any 3D model available on your computer. However, it's recommended to use a primitive model, such as a generic **BoxMesh**.

The transformation you'll implement will not only scale the object's vertices but will also apply a linear transformation known as **shearing**. Shearing tilts the shape of an object by moving its vertices along a fixed direction. The amount of this displacement depends on the vertices' positions along another axis. This transformation doesn't change the object's volume, but it does alter its shape by distorting right angles.

Start by preparing your scene as follows:

- Create a new **3D scene**.
- Add a `MeshInstance3D` as a child node.
- Assign a `BoxMesh` to the `MeshInstance3D`.
- Apply the `transformations` material to the mesh.

If you've followed these steps correctly, your scene should now look like this:



(1.12.e The **transformations** material has been assigned to the **BoxMesh**)

Since your object is three-dimensional, you'll begin the exercise by defining a  $3 \times 3$  matrix and initializing it as an identity matrix. This ensures that no visible transformation is applied to the 3D model at this stage. To do this, open your **transformations** shader and insert the following macro above the **vertex()** function:

```
shader_type spatial;

#define TRANSFORMATION_MATRIX mat3(vec3(1, 0, 0), vec3(0, 1, 0), vec3(0, 0,
    1))

void vertex()
{
    VERTEX = VERTEX * TRANSFORMATION_MATRIX;
}
```

As you can see in the example, a macro called `TRANSFORMATION_MATRIX` has been defined. It represents a  $3 \times 3$  identity matrix and will be expanded wherever it's used in the shader code. Keep in mind that you could also define a matrix using a function or by declaring a new variable directly inside the `vertex()` function. However, in this case, the `#define` directive is used to improve readability, consistency, and reusability throughout development.

**Note**

It's recommended to use the `#define` directive when: 1) You want to define reusable or constant blocks of code, 2) You're aiming for cleaner code organization and reduced duplication, 3) You need to change a value globally with a single edit. However, it's generally better to use `const` or `uniform` variables when substitution behavior is not required, as macros don't offer type safety. [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/shader\\_preprocessor.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/shader_preprocessor.html)

The matrix used earlier didn't take any parameters, but you could also define it like this:

```
shader_type spatial;

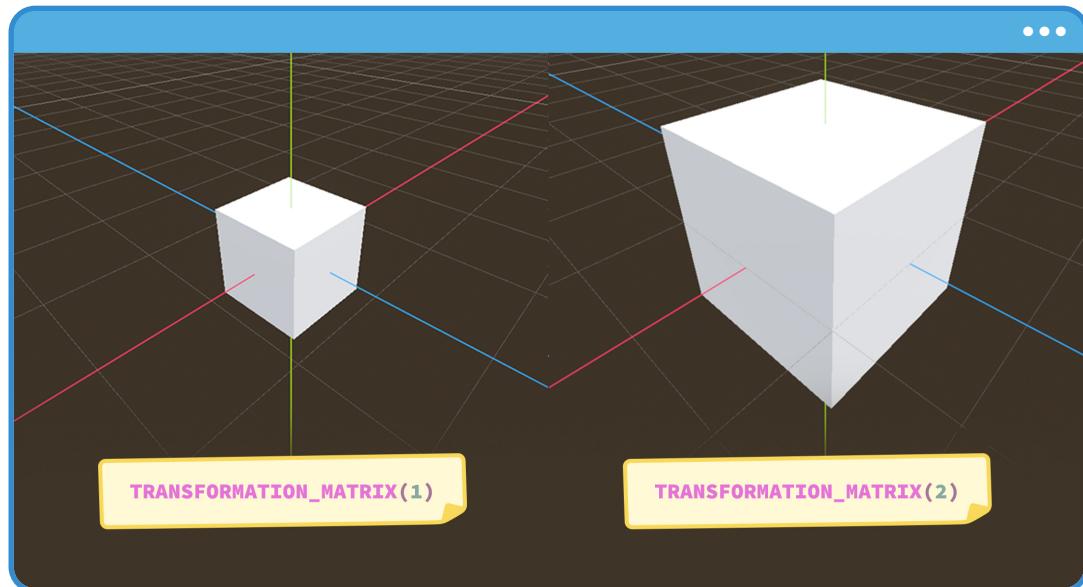
#define TRANSFORMATION_MATRIX(s) mat3(vec3(s, 0, 0), vec3(0, s, 0), vec3(0, 0,
    s))

void vertex()
{
    VERTEX = VERTEX * TRANSFORMATION_MATRIX(1);
}
```

In this version, you've defined a macro that takes an argument: `s`, which adjusts the values of the identity matrix. This configuration is ideal for applying scale transformations to a 3D object. You only need to change the argument to scale the object in real time:

- If `s = 1`, the BoxMesh keeps its original size.
- If `s = 2`, the BoxMesh doubles in size along all axes.
- If `s = 0.5`, the BoxMesh scales down to half its original size.

This approach allows for quick experimentation and gives you a clear visual understanding of the matrix's effect without modifying multiple lines of code. Later, you could replace the argument with a uniform variable to make the scale adjustable directly from the Inspector.



(1.12.f A scale transformation has been applied to the BoxMesh)

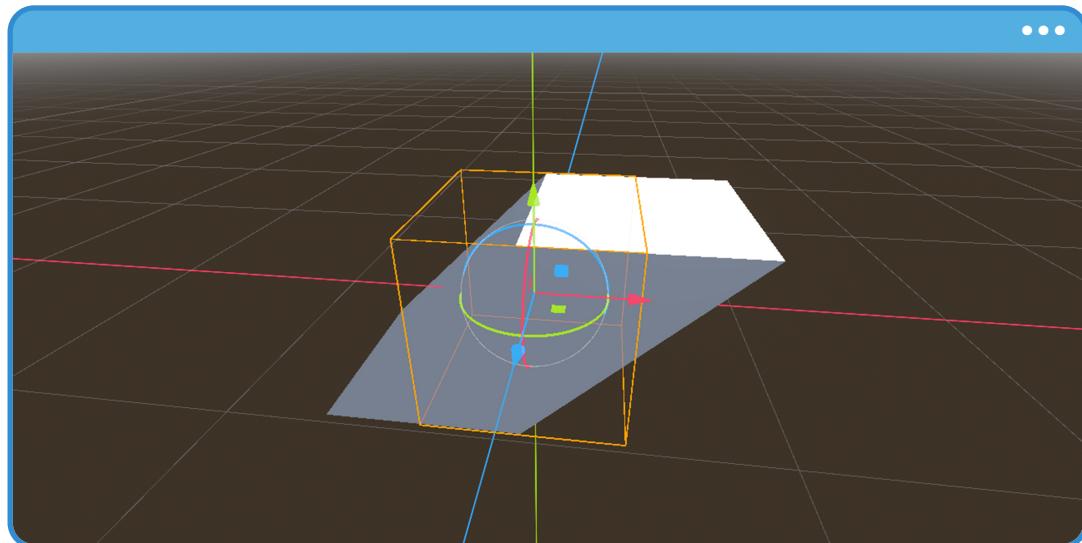
Now, you might ask – why does the **BoxMesh** scale when we change the matrix value? This happens because you're directly altering the original positions of each vertex in the 3D object – before those positions are transformed into world space by the **MODEL\_MATRIX**. In other words, the transformation is applied in object space, right at the point where each vertex still holds its local coordinates. This lets you modify the object's geometry directly, without affecting its global transformation or the scene hierarchy.

You can also extend this transformation matrix to apply **shearing**. To do so, simply introduce new values into the matrix, changing how one axis influences another.

For example, if you want to apply **shearing** along the *x*-axis based on the *y*-axis, you could modify the matrix as follows:

```
#define TRANSFORMATION_MATRIX(s) mat3(vec3(s, 1, 0), vec3(0, s, 0), vec3(0, 0, -s))
```

In this case, the value 1 has been added to the second component of the first **vec3**. This means the **x**-axis is now influenced by the **y**-axis, tilting the shape of the object as each vertex's vertical position increases.

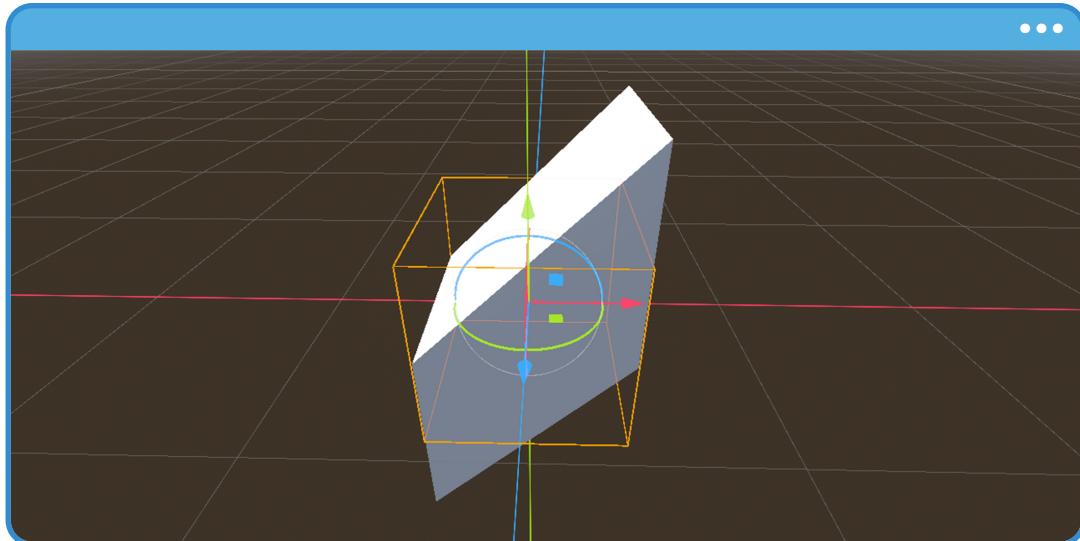


(1.12.g Shearing along the *x*-axis)

The same principle applies to other axis combinations, such as **x** and **z**, or **x** and **y**. For example, if you want to apply shearing on the **y**-axis, you simply need to add a numerical value to the first component of the second **vec3** in the matrix, as shown below:

```
#define TRANSFORMATION_MATRIX(s) mat3(vec3(s, 0, 0), vec3(1, s, 0), vec3(0, 0, -s))
```

This setup introduces an influence from the **x**-axis onto the **y**-axis. In other words, each vertex's horizontal displacement now affects its vertical position, causing the object to tilt diagonally to the side – proportional to its position along the **x**-axis.

(1.12.h Shearing along the  $y$ -axis)

This type of transformation is useful for creating visual effects such as animated deformations, weight simulation, dynamic leaning, or even controlled vibrations. The key lies in understanding how one axis influences another within the matrix structure. Now it's your turn – go ahead and experiment. Try modifying the matrix values and observe the different results you can achieve!



## Chapter 2

# Lighting and Rendering.

In this chapter, you'll continue your journey into shaders — this time with a focus on lighting and rendering. But how exactly is light calculated in a video game? What mathematical formulas allow us to simulate the interaction between light and objects?

Every time you apply a material to a surface — whether in 2D or 3D — you're activating a series of properties that go far beyond a simple texture. These properties include normal calculations, occlusion, specular reflection, light attenuation, and other physical phenomena that contribute to the final appearance of the object on screen.

Understanding how these processes work is essential if you want to modify, optimize, or even reinvent them using custom shaders. Throughout this chapter, you'll explore how the GPU interprets light, how different types of lighting behave in Godot, and what tools you have at your disposal to control the final look of a scene.

Before diving into practical examples, you'll first review the basic principles behind one of the most widely used lighting models in real-time graphics: the **Lambertian model**. You'll also learn about the built-in functions that give you access to information like the light direction, the object's normal, or the camera position. This foundational knowledge will prepare you to create advanced visual effects such as toon lighting, halftone shading, or other complex materials.

## 2.1 Material Properties.

Whenever you want to assign a new material to a 3D object in Godot, the engine prompts you to choose the type of material to create. For example, if you click the `<empty>` field in the **Material Override** property of a **MeshInstance3D**, you'll see three options:

- StandardMaterial3D.
- ORMMaterial3D.
- ShaderMaterial.

So far, you've primarily worked with the third option — **ShaderMaterial** — as you've been writing your own shaders. But what about the other two? What are they used for — and more importantly, how do they differ? Let's begin by comparing them. This will help you better understand the purpose of each configuration and when it's appropriate to use one over the other.

According to the official Godot documentation, **StandardMaterial3D**:

“

**StandardMaterial3D's** properties are inherited from **BaseMaterial3D**.

**StandardMaterial3D** uses separate textures for ambient occlusion, roughness and metallic maps. To use a single ORM map for all 3 textures, use an **ORMMaterial3D** instead.

[https://docs.godotengine.org/en/stable/classes/class\\_standardmaterial3d.html](https://docs.godotengine.org/en/stable/classes/class_standardmaterial3d.html)

”

The main difference between **StandardMaterial3D** and **ORMMaterial3D** lies in the use of an **ORM** map — a single RGB texture that packs three different grayscale textures into its individual channels:

- O – Occlusion: stored in the red channel (R).
- R – Roughness: stored in the green channel (G).
- M – Metallic: stored in the blue channel (B).

This technique can be a bit confusing when you're just starting out with shaders. For example, when you create a .png texture, it typically includes four channels by default: RGBA. So how is it possible to store different textures in individual channels of the same image?

The key is understanding the type of information these textures represent. Occlusion, roughness, and metallic values are all grayscale textures — they don't encode color, but rather intensity. That means all four channels in each of these images contain the same value. As a result, you only need to store one of them in a single channel (for example, storing occlusion only in the red channel) and ignore the rest.

#### Note

By taking advantage of this characteristic, you can combine three grayscale textures into a single image — each one stored in a different channel. This technique is known as **channel packing**, and it not only simplifies resource management but also improves performance by reducing the number of textures the GPU has to load and process during rendering.

Now then, how does this compare to a **ShaderMaterial**? Let's break down the differences between these three material types to better understand their purpose and usage:

**StandardMaterial3D,**

- A material based on **Physically Based Rendering (PBR)**, commonly used for 3D objects.
- Easy to configure, with built-in support for normal maps, roughness, metallic, occlusion, fresnel, and many other visual properties.
- Supports transparency, shadows, refraction, subsurface scattering, and multiple UV coordinates.
- The shader is predefined — so no coding is required to use it. However, it can be **GPU-intensive**, especially on low-end devices.

**ORMMaterial3D,**

- Like **StandardMaterial3D**, this is a PBR material used for 3D objects, with the key difference being that it uses a packed ORM texture.
- Retains all the features of **StandardMaterial3D**, but is slightly more optimized by reducing the number of texture samplers.
- It also uses a predefined shader, so no custom code is needed.

**ShaderMaterial,**

- Unlike the previous two, this material does not include any predefined properties — you have full control to define everything manually.
- Can be used for both 2D and 3D objects, depending on how you configure it.
- While it includes basic lighting by default, taking full advantage of its potential requires writing code in Godot's shader language (`.gdshader`);
- Its performance impact depends entirely on your shader code — the more complex the operations, the greater the load on the GPU.

In conclusion, **StandardMaterial3D** and **ORMMaterial3D** are ready-to-use materials built into Godot. Both are excellent for quickly getting started on a project without writing any code, as they come with built-in lighting calculations and basic visual effects.

On the other hand, **ShaderMaterial** gives you full control over what is rendered on screen. This makes it the perfect tool for creating custom visual effects, stylized materials, or even complex simulations. In fact, with the right approach, you can fully replicate the behavior of the built-in materials from scratch — just by writing your own shader code.

That's exactly what you'll do throughout this chapter. You'll recreate many of the visual features found in Godot's default materials, but with a focus on custom shader-based effects. This hands-on practice will give you a deep understanding of the lighting calculations involved in material creation — and prepare you to build shaders tailored to your project's artistic vision.

## 2.2 Introduction to the Lambertian Model.

One of the simplest — and most widely used — lighting models in computer graphics is the **Lambertian model**, named after an observation made by **Johann Heinrich Lambert** in the 18th century. But how does this model actually work? To understand it, you need to consider two key variables:

- The direction of the light source.
- The surface normal.

The amount of light hitting a surface depends on the angle between that surface and the direction of the light. If the surface faces the light directly, it receives the maximum possible illumination. If it faces tangentially — or away from the light — it receives little to no light.

Let's look at its mathematical definition to better understand how this works:

$$D = \max(0, n \cdot l) i k$$

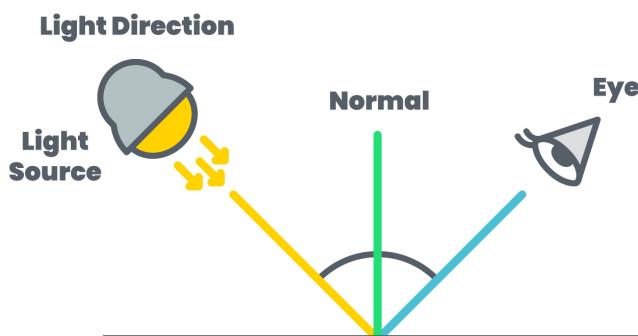
(2.2.a)

Where,

- $D$  represents the intensity of diffuse light received by a point on the surface.
- $k$  is the diffuse coefficient or the base color of the surface (typically assigned to the **ALBEDO** variable in the **fragment()** function).
- $i$  is the intensity of the light source.
- $n$  is the surface normal (a unit vector).
- $l$  is the light direction (also a unit vector).

The dot product between  $n$  and  $l$ , written as  $(n \cdot l)$ , is equivalent to the cosine of the angle  $\theta$  between the two vectors. Since both  $n$  and  $l$  are unit vectors (their lengths equal 1), their dot product can be directly interpreted as  $\cos(\theta)$ . This relationship is incredibly useful for both developing lighting formulas and implementing them in shaders.

While it's true that you could expand this equation to include additional elements – such as light color, distance attenuation, or other physical factors – this simplified version of the Lambertian model is enough to help you understand the core logic behind diffuse lighting calculations.



(2.2.b Diagram illustrating Lambertian shading)

If you look at the diagram in Figure 2.2.b, you'll notice that it includes the viewer (or camera) as part of the geometry. However, the camera is not present in Equation 2.2.a. Why is that?

The reason is simple: the **Lambertian** model describes purely diffuse reflection. This means light is scattered uniformly in all directions, regardless of the angle from which the surface is observed. In other words, the resulting illumination is independent of the viewer or camera position.

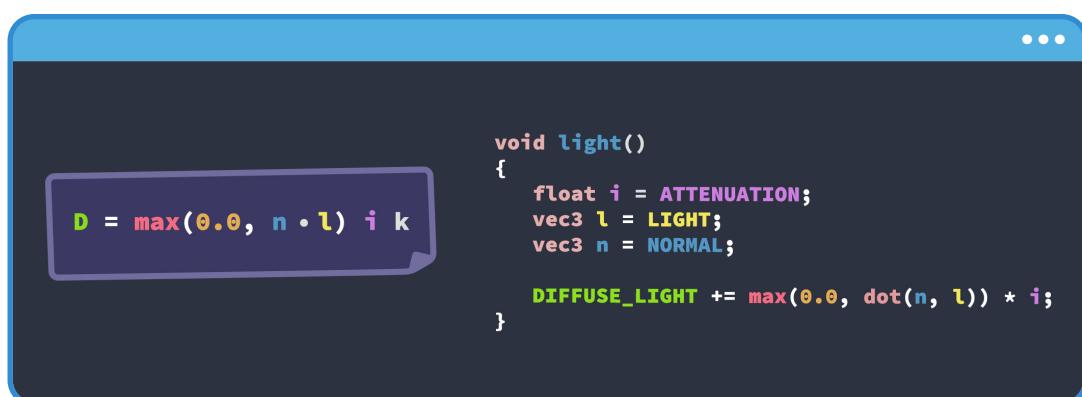
This behavior contrasts with more complex lighting models – like **Phong** or **Blinn-Phong** – which do account for the viewer’s direction when calculating specular reflection. In those cases, the view direction must be included as an additional variable in per-pixel lighting calculations.

Despite its simplicity, implementing the Lambertian model in Godot is straightforward thanks to a set of **built-in variables** provided by the engine. For example:

- $D$  corresponds to the built-in variable **DIFFUSE\_LIGHT**.
- $n$  is the surface normal, represented by **NORMAL**.
- $l$  is the light direction, represented by **LIGHT**.
- $i$  is the light intensity, which you can approximate using **ATTENUATION**.

Finally, the term  $k$  – the surface color – is represented by the **ALBEDO** variable. However, you don’t need to include it manually in the equation, because Godot automatically multiplies the diffuse light by **ALBEDO** internally during the rendering process.

It’s also important to note that lighting calculations in Godot are performed inside the **light()** function, which is commented out by default. Here’s what a basic Lambertian model implementation looks like in Godot’s **.gdshader** language:



```

void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;

    DIFFUSE_LIGHT += max(0.0, dot(n, l)) * i;
}

```

(2.2.c Colors have been defined to identify the variables)

The first step in implementing this model is to enable the `light()` function. In practice, this method works as an extension of `fragment()`, since it also runs once per pixel – but with a key difference: it's called once for every light that affects that pixel.

By default, this function is commented out because it's optional – you'll only use it if you want to customize how lighting interacts with the material. But why can this be costly for the GPU? You already know that `fragment()` runs once for every visible pixel on screen. For example, in a game running at 1920×1080 resolution, `fragment()` is called around 2,073,600 times per frame. The `light()` function, however, executes not just per pixel, but per light affecting that pixel. If a single pixel is affected by four different light sources, `light()` will be called four times for that same pixel – potentially resulting in over 8 million executions per frame, depending on the complexity of the scene. That's why it's important to use this function only when necessary, and with care – especially if your project needs to run efficiently on lower-end devices.

## 2.3 Implementing the Lambertian Model.

In this section, you'll complete a series of tasks to help you not only understand how the Lambertian model works but also how to use the `light()` function in Godot. You'll also explore the `ambient_light_disabled` render mode, which will let you clearly identify which light sources are affecting your character by default.

Here's what you'll accomplish:

- Work with the `light()` function.
- Implement the Lambertian lighting model.
- Transform the lighting result into a simple toon-style effect.

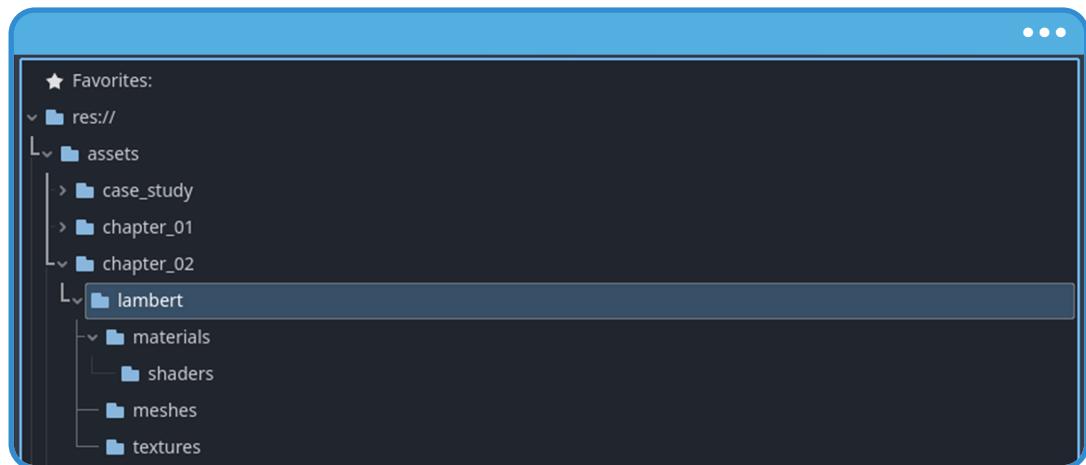
Before diving into the technical implementation, let's first organize your project, just like in previous chapters. Follow these steps:

- 1 Inside your project, under the `assets` folder, create a new subfolder named `chapter_02` to store all resources for this chapter.
- 2 Inside `chapter_02`, create another folder named `lambert`.

- 3 Within the **lambert** folder, organize your content by creating the following subfolders:

- a **materials**.
- b **shaders**.
- c **meshes**.
- d **textures**.

If you've followed the steps correctly, your project structure should now look like this:



(2.3.a The **chapter\_02** folder has been added under assets)

Next, you'll implement the example illustrated in Figure 2.2.c. To do so, you'll create a new **shader** and enable the **light()** function in order to put the various aspects of the Lambertian model into practice.

For this exercise, you'll use the **suzanne.fbx** model, which should be imported into the **meshes** folder you created earlier. This file is included in the downloadable package that comes with the book.

#### Note

Since the model uses the **.fbx** extension, you'll need to follow the process described in **Section 1.4** of the previous chapter. That involves opening the file using **New Inherited Scene** and then selecting **Make Unique** to save an editable copy of the model within your project.

Start by creating a new **material**. Right-click the **materials** folder and choose:

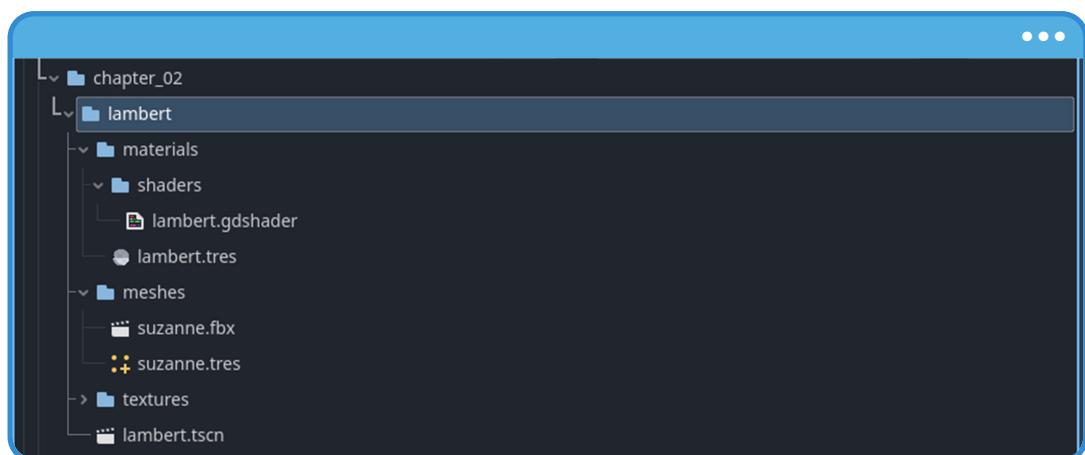
- Create New > Resource > ShaderMaterial.

For practical purposes, name this material **lambert**. Then, create the associated **shader**. In the **shaders** folder, right-click and select:

- Create New > Resource > Shader.

Use the same name — `lambert` — for the shader. This helps maintain a clear and organized relationship between the two resources.

If everything has been configured correctly, your project structure should now look like this:



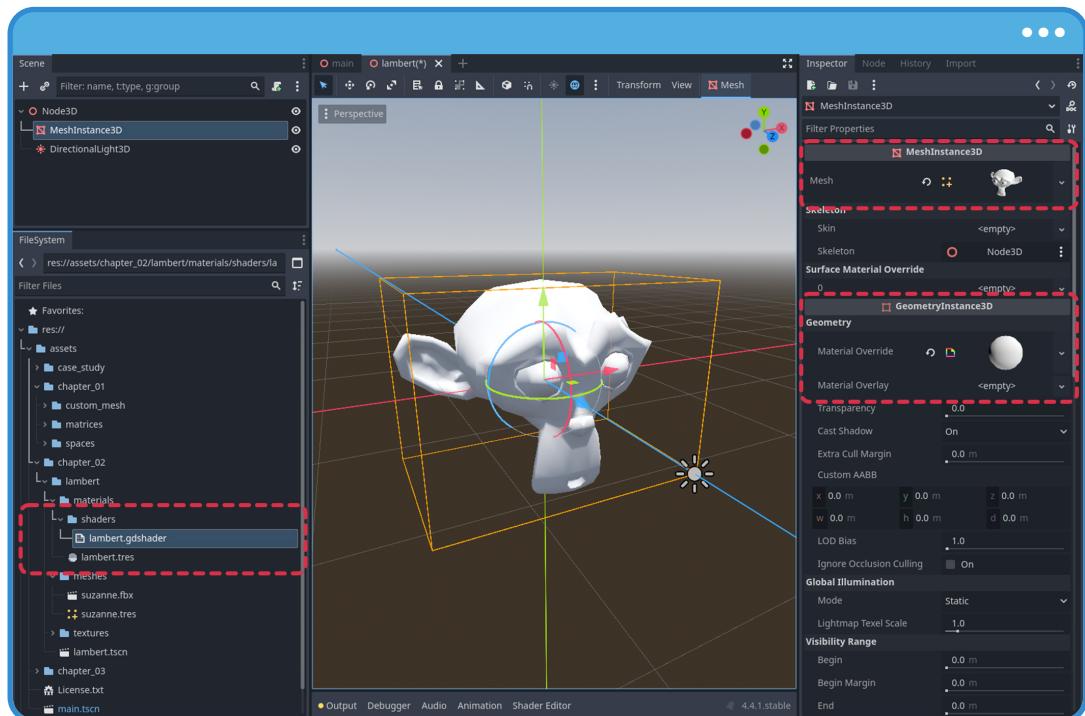
(2.3.b The resources have been added and **Suzanne** has been marked as unique)

At this point, you could begin editing the shader right away. However, before doing so, it's important to set up the scene properly. This is crucial because enabling the `light()` function will change how the object is lit — giving you a clearer view of how Godot's default lighting system behaves.

For the scene setup, use a main **Node3D** as the root, and add a **MeshInstance3D** as its child. Assign the previously saved **suzanne.tres** file to this mesh. As part of this process, keep the following in mind:

- The **lambert** shader must be assigned to its corresponding **ShaderMaterial**.
- The **ShaderMaterial** must be applied to the mesh – specifically in the **Material Override** property of the **MeshInstance3D**.
- Add a **DirectionalLight3D** node to the scene. This will allow you to observe how the lighting changes as you modify the light's direction.

If all steps have been followed correctly, your scene should now look like this:



(2.3.c The character has been configured in the scene)

As mentioned earlier, the first step is to enable the **light()** function in your shader. To do this, open the **lambert** shader and uncomment the **light()** function, allowing its content to run.

Here's the initial structure of the shader:

```
shader_type spatial;

void vertex() { ... }

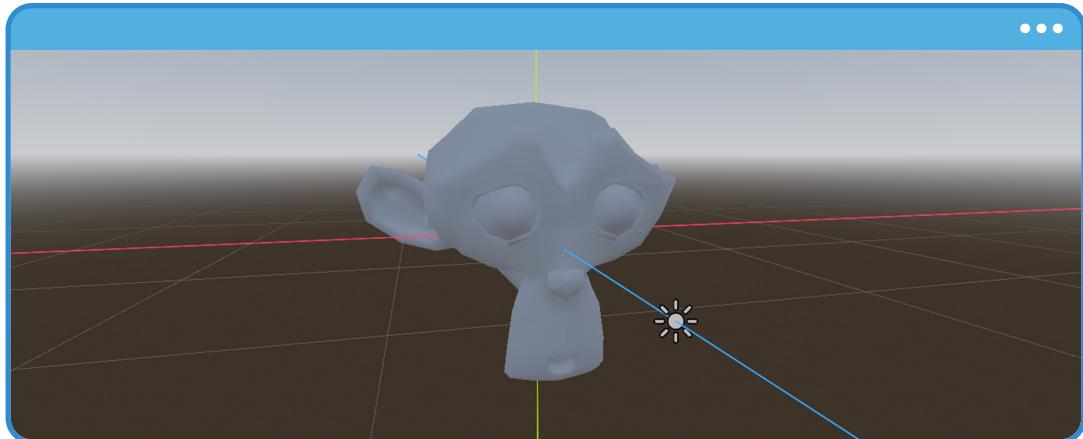
void fragment() { ... }

void light()
{
    // Called for every pixel for every light affecting the material.
    // Uncomment to replace the default light processing function with this...
}
```

Because the **light()** function replaces Godot's default light processing, you'll notice an immediate change in the object's appearance once it's activated. Specifically, the direct lighting that was affecting the 3D model will appear to disappear.

And we say "partially disappears" because, although direct lighting is overridden, the model is still influenced by ambient lighting, which Godot applies globally by default. This ambient light comes from the environment settings and is not tied to any specific lights in the scene, such as **DirectionalLight3D**.

This distinction is useful – it allows you to clearly separate the visual impact of ambient light from that of directional light as you begin implementing the Lambertian model.



(2.3.d The character is affected by ambient lighting)

If you want to disable ambient lighting, you need to use the `ambient_light_disabled` render mode. This directive tells the engine to completely ignore any lighting contributions from the environment — that is, the global illumination Godot applies by default when no explicit light sources are present.

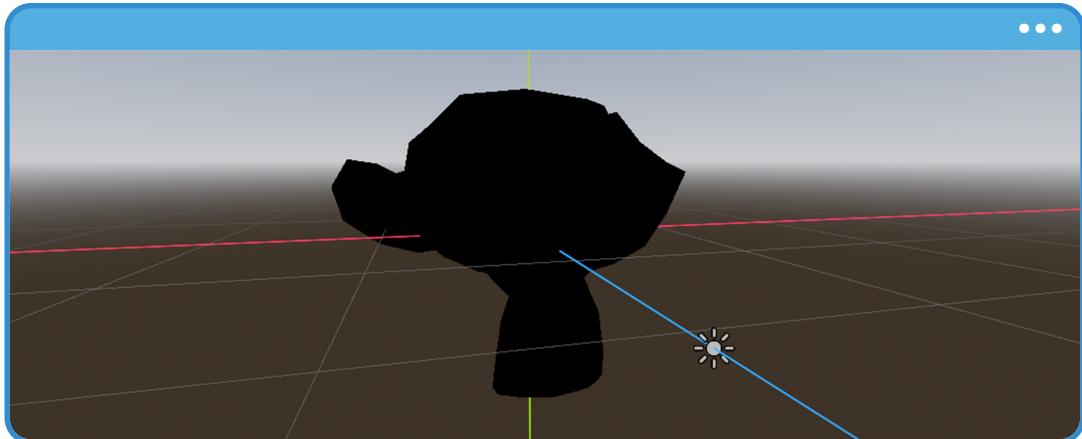
Once you add this line, your character will appear completely dark. This is expected: there are no active light sources affecting the model visually. However, this condition is quite useful for the lighting tests you're about to perform, as it allows you to observe only the results of your custom lighting calculations inside the `light()` function.

Your base shader code with the directive applied will look like this:

```
shader_type spatial;
render_mode ambient_light_disabled;

void vertex() { ... }
```

After applying this render mode — and with a directional light active in the scene — you should see the model appear entirely black.



(2.3.e No light source is affecting our 3D model)

Now that your character is properly configured, it's time to implement the lighting function shown in Figure 2.2.c, as demonstrated below:

```
void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;
    float D = max(0.0, dot(n, l)) * i;

    DIFFUSE_LIGHT += vec3(D);
}
```

**Note**

You can also incorporate the **light's color** into this operation by using the **LIGHT\_COLOR** variable, as described in Godot's official documentation on Light Built-ins: [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/spatial\\_shader.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/spatial_shader.html)

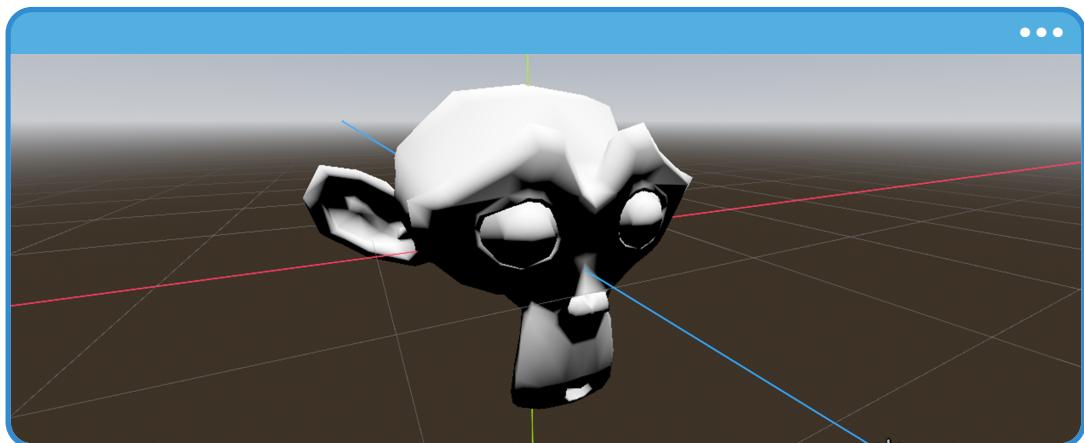
It's important to mention that the variable names in this function ( $i, l, n, D$ ) follow traditional conventions when implementing the Lambertian model. However, in a production setting, it would be preferable to use more descriptive names or operate directly with Godot's built-in

variables. Avoiding temporary variables that are only used once can improve both readability and performance.

A more concise and production-ready version of the same calculation might look like this:

```
DIFFUSE_LIGHT += vec3(max(0.0, dot(NORMAL, LIGHT)) * ATTENUATION);
```

That said, for educational purposes, this chapter will continue to use intermediate variables to help reinforce the connection to the original mathematical model. Once you save your changes and return to the scene, you should see an immediate change in Suzanne's lighting — she now responds only to the direction and intensity of the directional light, as dictated by the Lambertian model you've just implemented.



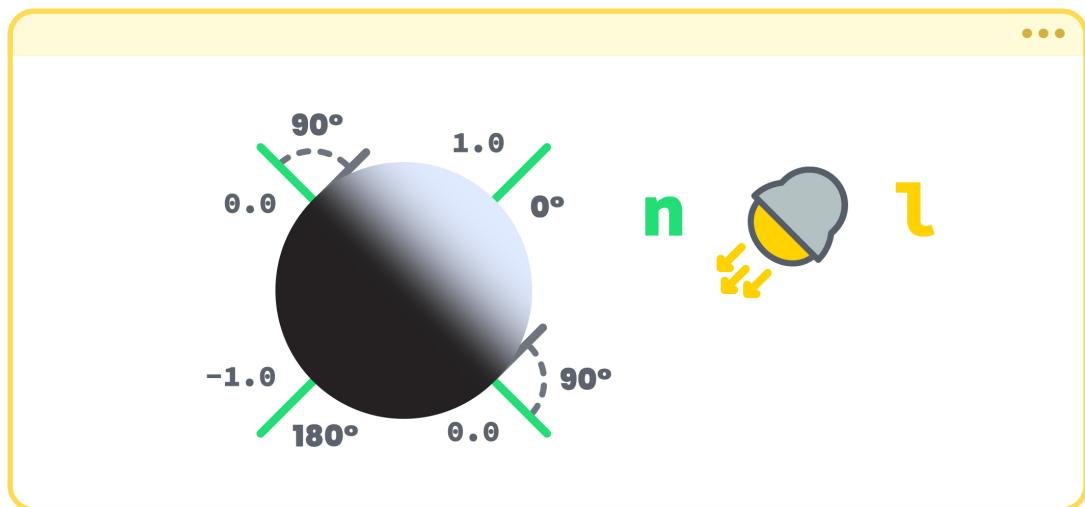
(2.3.f The Lambertian model has been applied to Suzanne)

As you can see, the 3D model now displays only two tones: light and shadow (white and black). This outcome is mathematically accurate — it faithfully reflects the behavior of the Lambertian model in its most basic form.

But why does this simple effect create a sense of volume in the character? The key lies in the angle between the surface normals and the direction of the light. As this angle changes across different points on the model, so does the amount of light that hits each point. This smooth variation in intensity is what creates gradual shading across curved surfaces — like

Suzanne's cheeks and forehead — allowing you to perceive depth and contour, even in the absence of color.

Let's take a closer look at this principle in the next figure, where the relationship between the angle  $\theta$ , the surface normal, and the light direction is illustrated graphically.



(2.3.g The Lambertian model has been conceptually applied to a sphere)

A sphere — like any 3D model — has a normal vector at each of its vertices. However, for illustration purposes, the previous figure highlights only four of them. Each normal points in a different direction, depending on the curvature and volume of the surface.

If you look closely, you'll notice that the angle between the normals and the light direction varies from  $0^\circ$  to  $180^\circ$ . This angular relationship is fundamental to the Lambertian model, as it determines how much light each part of the object receives.

Let's revisit how the `dot()` function behaves in this context:

- It returns 1.0 (white) when the angle between the normal and the light is  $0^\circ$  — meaning the light hits the surface perpendicularly.
- It returns 0.0 (black) when the angle is  $90^\circ$  — the light hits the surface tangentially.
- It returns -1.0 (completely dark, not useful for diffuse lighting) when the angle is  $180^\circ$  — the light is pointing in the opposite direction.

However, the Lambertian model uses the function `max(0.0, dot(n, l))`, meaning any negative value is discarded. That's because light coming from "behind" the surface should have no visual effect on it.

Since diffuse light can't be negative in the real world, allowing values below zero could introduce visual artifacts, especially in areas where lighting should have no effect at all. For that reason, `max()` is used to ensure that light contributions are always zero or greater.

In some cases, this function is replaced by `clamp()`, which restricts the result to both a minimum and a maximum value. This can be useful if you want to ensure the output stays within a specific range — for example, between 0.0 and 1.0.

The `max()` function for a 3D vector can be defined like this:

```
vec3 max(vec3 a, vec3 b)
{
    return vec3(
        a.x > b.x ? a.x : b.x,
        a.y > b.y ? a.y : b.y,
        a.z > b.z ? a.z : b.z
    );
}
```

Now that you understand the fundamentals of Lambertian shading, you can take a creative step further by limiting the gradient between lit (white) and shadowed (black) regions. This results in a more stylized visual effect. One way to achieve this is by using the `smoothstep()` function, which performs a smooth interpolation between two values.

Here's how the function is defined for floating-point values:

```
float smoothstep(float edge0, float edge1, float x)
{
    float t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
    return t * t * (3.0 - 2.0 * t);
}
```

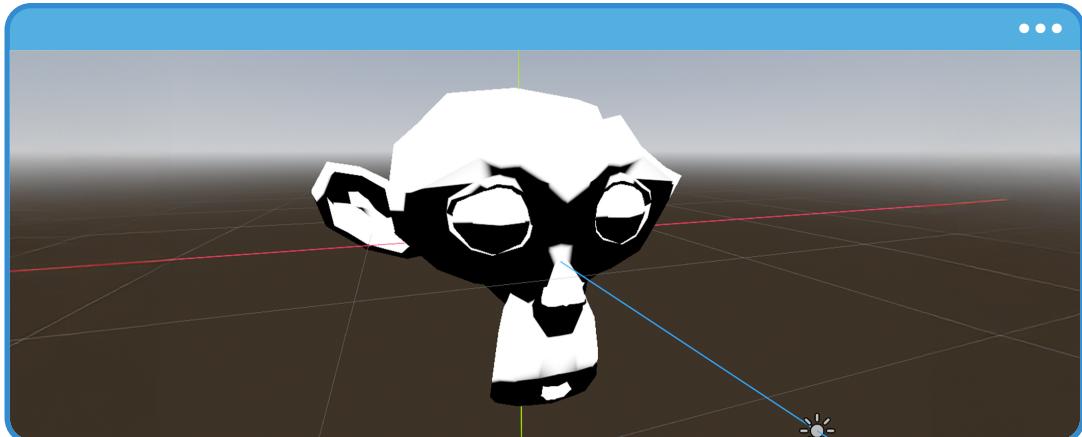
The function takes three arguments: an input value `x`, and two edges `edge0` and `edge1`. It returns a result between 0.0 and 1.0, forming a smooth transition. This is especially useful when you want more control over gradient blending or to simulate soft cutoff regions in lighting.

You can apply `smoothstep()` directly to the output of the Lambertian model to narrow the range of intermediate values. This sharpens the contrast between light and shadow, producing a more graphic, toon-like style.

```
void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;
    float D = max(0.0, dot(n, l)) * i;
    D = smoothstep(0.0, 0.05, D);

    DIFFUSE_LIGHT += vec3(D);
}
```

This operation creates a binary lighting effect, where the transition between bright and dark areas is more abrupt, yet still smooth enough to avoid harsh edges. It's a perfect technique for achieving a stylized look, particularly for art-driven shading models like toon rendering.



(2.3.h Smoothed gradient)

Another interesting effect you can implement is to adjust the lighting tone your character receives. You can achieve this using the `mix()` function, which smoothly interpolates between two colors and returns a new `vec3` based on a blend factor.

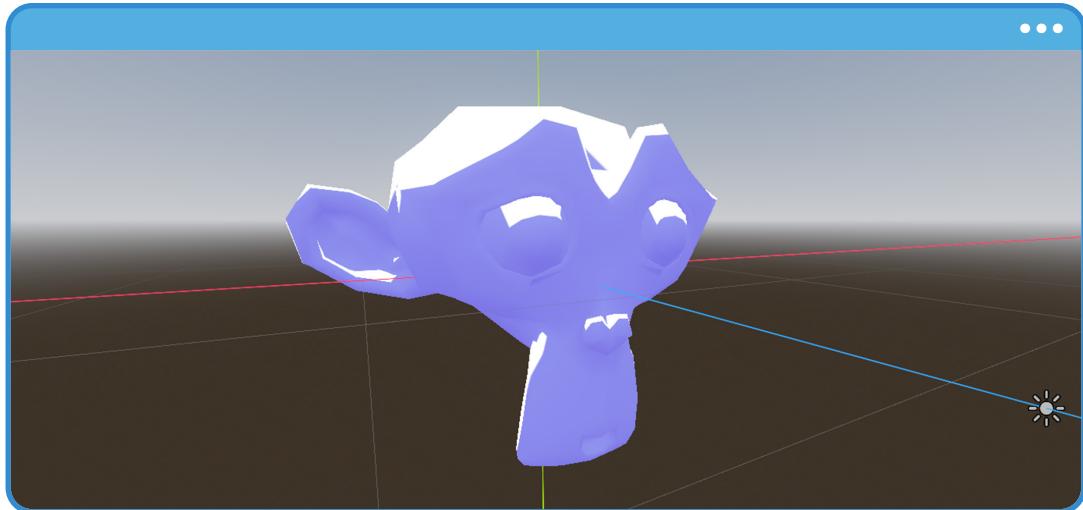
```
void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;
    float D = max(0.0, dot(n, l)) * i;

    D = smoothstep(0.0, 0.05, D);
    vec3 diffuse = mix(vec3(0.042, 0.023, 0.534), vec3(1.0), D);

    DIFFUSE_LIGHT += diffuse;
}
```

In this case, the variable `D` is used as a blend factor between two colors: a dark bluish tone `vec3(0.042, 0.023, 0.534)` and pure white `vec3(1.0)`. As `D` increases — meaning the surface receives more direct light — the resulting color gradually approaches white.

Conversely, in areas that receive little to no light (where `D` is close to 0), the color retains its blue tint. This behavior produces a stylized and custom shading effect, making it ideal for scenes with an artistic aesthetic or projects that require non-photorealistic rendering (NPR).



(2.3.i The resulting shadow has a bluish tint)

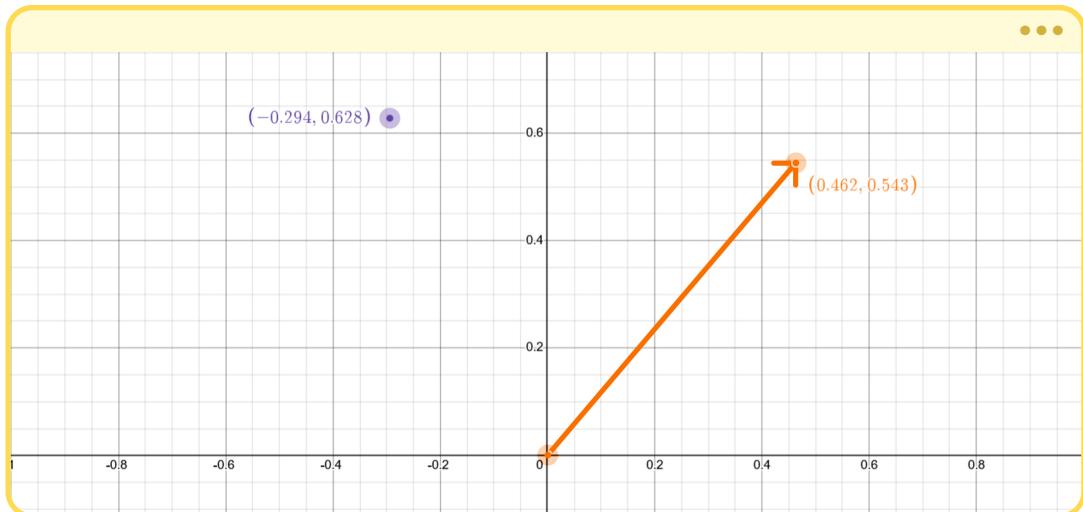
## 2.4 Vector: Points vs. Directions.

Up to this point, we've been working exclusively with the implementation of the Lambertian model. If you've followed the steps correctly, you're likely experimenting with different values to achieve various lighting tones in Suzanne's rendering. But this process raises an essential question: Why does it work? Why does changing certain vectors produce noticeable shifts in the character's lighting?

One key concept that wasn't explicitly mentioned in the previous section is this: in the context of lighting, the vectors you manipulate — such as the light direction or surface normal — represent directions, not points in space.

So, what's the difference between a point and a direction? A point defines a specific position in 3D space. For example, a vertex on a mesh represents a corner of the model in either local or global coordinates. It tells you where something is.

In contrast, a direction indicates an orientation without referring to a fixed position. It tells you where something is pointing, not where it's located. Vectors like **LIGHT**, **VIEW**, and **NORMAL** are unit directions — they describe the direction of the light source, the viewing angle of the camera, or the orientation of the surface at a given fragment.



(2.4.a A point (purple) vs. a direction (orange))

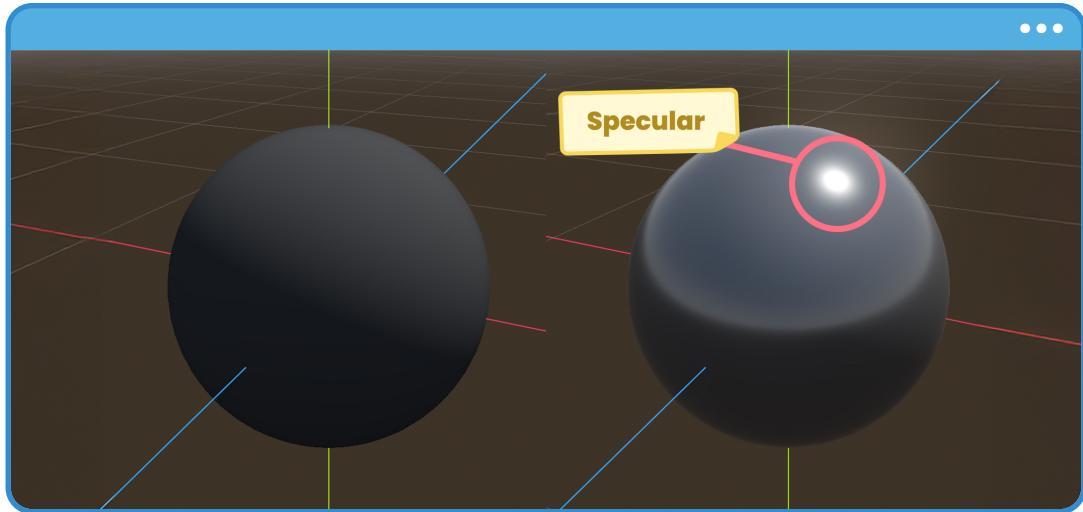
Understanding this distinction is critical, especially as you dive deeper into lighting calculations in this chapter. When you calculate the dot product between the normal and the light direction, you're not comparing spatial positions. Instead, you're evaluating the angular relationship between two orientations. This angular relationship determines how much light hits a surface – and therefore, how bright or shadowed it appears.

## 2.5 Introduction to the Blinn-Phong Model.

In this section, we'll talk about the specular effect, the one found in the **Metallic > Specular** property of a **StandardMaterial3D**. This effect lets you render surfaces with shiny reflections, simulating how light reflects off polished materials.

To understand how it works, you need to consider three main properties:

- The direction of the light source.
- The surface normal.
- The view direction (camera).



(2.5.a On the left, Lambert; on the right, Specular)

This model, widely used in computer graphics, originates from the work of Bui Tuong Phong, who proposed a method for adding specular highlights to a surface based on the orientation of its normals.

According to the original Phong model, if you want to simulate specular reflectance, you must perform the following mathematical operation:

$$S = \max(0, \mathbf{r} \cdot \mathbf{v})^m$$

(2.5.b)

Here,  $\mathbf{r}$  is the reflection vector of the light,  $\mathbf{v}$  is the view direction (camera),  $\mathbf{n}$  is the normal, and  $m$  is the shininess exponent, which controls the concentration of the reflection. A low value for  $m$  (for example,  $m = 8$ ) produces a wide, soft highlight, while a high value (such as  $m = 256$ ) results in a small, sharp reflection.

The Phong model implemented in **.gdshader** would look like this:

```
void light()
{
    vec3 n = NORMAL;
    vec3 v = VIEW;
    vec3 l = LIGHT;
    vec3 r = reflect(-l, n);
    float m = 64.0;

    float s = pow(max(0.0, dot(r, v)), m);
    SPECULAR_LIGHT = vec3(s);
}
```

(2.5.c Colors have been defined to identify each variable)

However, in real-time graphics like Godot, it's common to use an optimized variant known as the **Blinn-Phong** model. Instead of calculating the reflection vector, this model uses an intermediate vector called **halfway** ( $h$ ), which represents the midpoint between the light direction and the view direction. This leads to a new specular formula, which is visually similar to the Lambertian model (Figure 2.2.a), but with a different approach.

$$S = \max(0, n \cdot h)^m \quad (n \cdot l > 0)$$

(2.5.d)

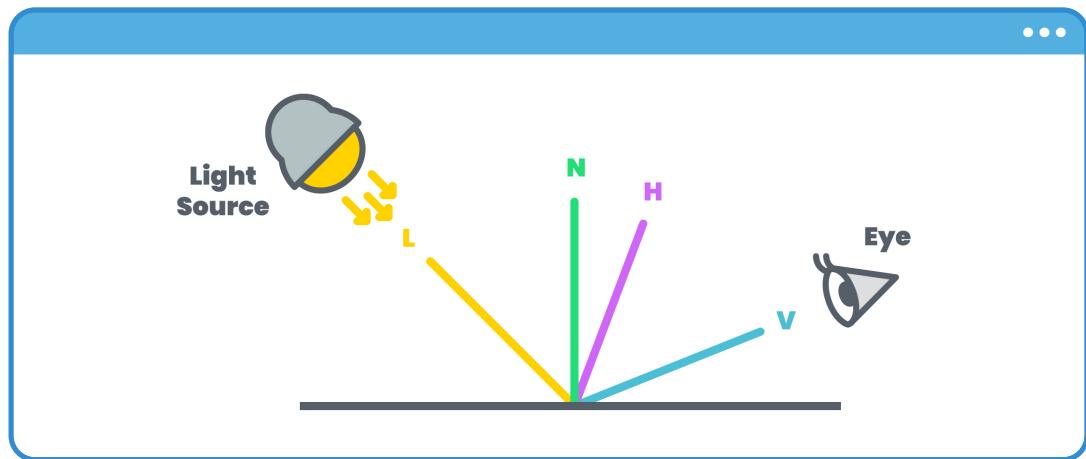
Its implementation in **.gdshader** looks like this:

```
void light()
{
    vec3 n = NORMAL;
    vec3 l = LIGHT;
    vec3 v = VIEW;
    vec3 h = normalize(l + v);
    float m = 64.0;

    float s = pow(max(0.0, dot(n, h)), m);
    s *= float(dot(n, l) > 0.0);
    SPECULAR_LIGHT = vec3(s);
}
```

(2.5.e Colors have been defined to identify each variable)

The more closely the object's normal aligns with the  $\mathbf{h}$  vector, the stronger the specular highlight will be. This produces smooth, realistic reflections that depend on the viewing angle – unlike the Lambertian model, which does not take the camera's position into account.



(2.5.f Diagram illustrating the Blinn-Phong model)

## 2.6 Implementing the Blinn-Phong Model.

In this section, you'll implement the **Blinn-Phong** lighting model inside the `light()` function using the two configurations previously introduced in Figures 2.5.b and 2.5.d. You'll also learn how to enhance the appearance of specular highlights by converting lighting from linear space to sRGB, which is essential for achieving more accurate visual results on screen.

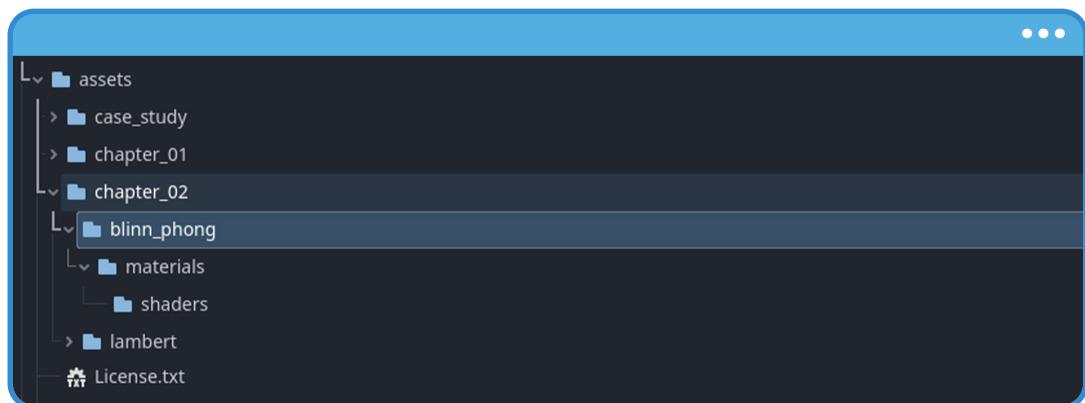
To do this, follow these steps:

- We'll start from the Lambertian shader we already implemented.
- Then, we'll add both variants of the specular equation.
- Finally, we'll apply a color conversion from linear to sRGB to enhance the highlights.

Before writing any code, let's organize the project to keep everything structured and consistent:

- 1 Inside the **chapter\_02** folder, create a new subfolder named **blinn\_phong**.
- 2 Inside this folder, add the following subfolders:
  - a **materials**.
  - b **shaders**.

You won't need to import new models or textures this time, as you'll reuse the resources from the previous section. If you followed the steps correctly, your project structure should now look like this:



(2.6.a The blinn\_phong folder has been added to the project)

As mentioned earlier, you'll continue working from the Lambert shader created in Section 2.3. However, to preserve the original shader, duplicate it before making any changes. To do this, right-click the shader file and select **Duplicate** (or press **Ctrl + D**).

Once duplicated, rename the file to blinn\_phong and move it to the following path:

- chapter\_02 > blinn\_phong > materials > shaders.

For the scene setup, you'll reuse the one based on the **Suzanne** model. Following the same logic, duplicate the file **lambert.tscn**, rename it to **blinn\_phong.tscn**, and save it in:

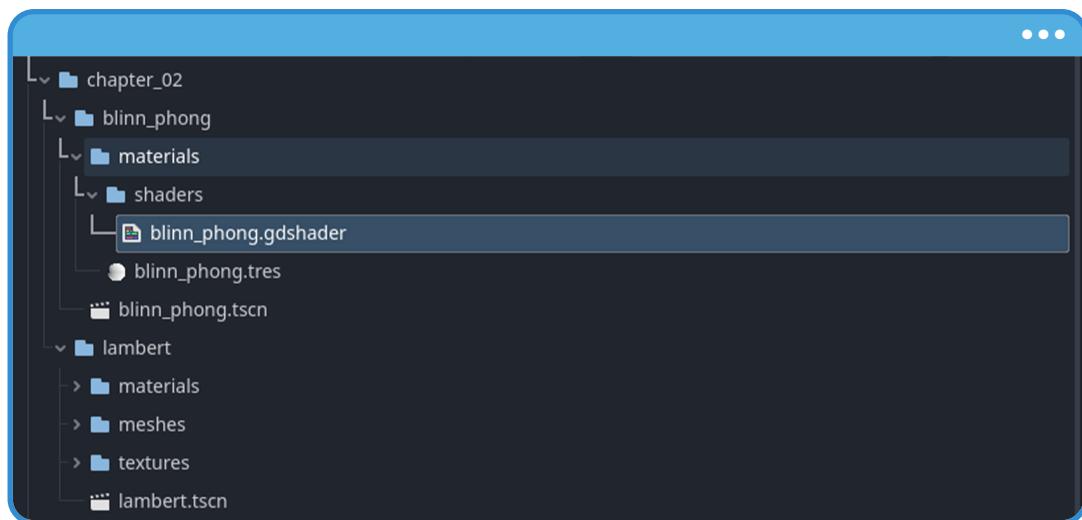
- chapter\_02 > blinn\_phong.

Next, create a new material. Right-click the materials folder and choose:

- Create New > Resources > ShaderMaterial.

Name this material **blinn\_phong** as well, so that all elements are properly linked and easy to identify in this section.

If you've followed all the steps correctly, your project structure should now look like this:



(2.6.b The material and scene from lambert have been duplicated)

**Note**

After creating the shader and material, make sure to assign the shader to the material, and then apply the material to the 3D model in the scene. This is necessary in order to visualize the effects of your code changes in the final render.

Since the Suzanne scene is already set up, you can now proceed to edit the **blinn\_phong** shader. Start by restructuring the **Lambertian** model logic into a separate function. This will improve readability and make your code easier to reuse.

```

float lambert(float i, vec3 l, vec3 n)
{
    return max(0.0, dot(n, l)) * i;
}

void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;

    float d = lambert(i, l, n);

    DIFFUSE_LIGHT += vec3(d);
}

```

As you can see, the Lambertian model logic remains unchanged. The only difference is that the main operation — the dot product between the normal and the light direction, scaled by intensity — is now encapsulated in a separate function called `lambert()`.

**Note**

In this updated shader, the properties `_Shadow`, `_Highlight`, and `_Smoothness` have been removed, since this section focuses exclusively on implementing the Blinn-Phong model.

This change improves code modularity, making it easier to reuse the function elsewhere or tweak it later if needed.

Now, implement the Phong model, as defined by the equation shown in Figure 2.5.b. To do this, create a new function named `phong()`, which calculates the specular component using the view vector, light direction, surface normal, and shininess exponent:

```
float phong(vec3 v, vec3 l, vec3 n, float m)
{
    vec3 r = reflect(-l, n);
    return pow(max(0.0, dot(r, v)), m);
}
```

**Note**

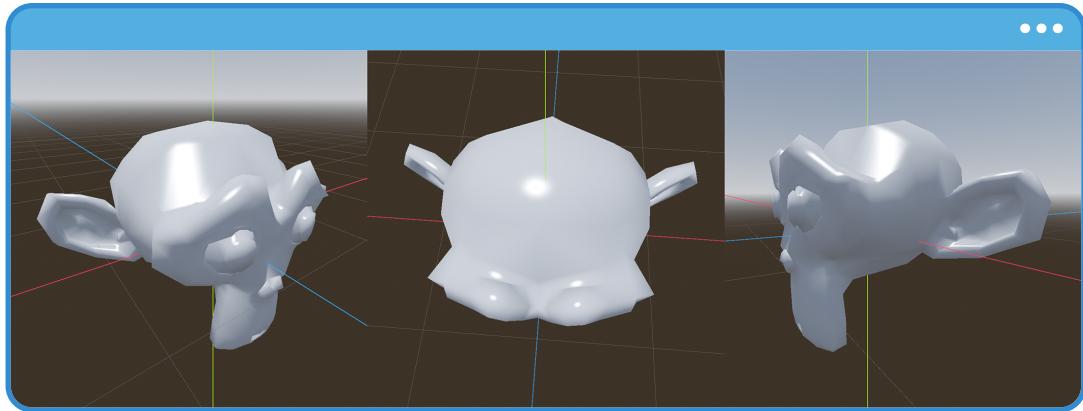
While Godot provides the built-in **SPECULAR** variable inside the **fragment()** function for simplified specular computation, implementing this function manually helps you understand how the model works in detail — and, more importantly, how to customize it to your needs.

With both functions ready, you can now update the **light()** method to include the specular calculation from the Phong model:

```
28 void light()
29 {
30     float i = ATTENUATION;
31     vec3 l = LIGHT;
32     vec3 n = NORMAL;
33     vec3 v = VIEW;
34
35     float d = lambert(i, l, n);
36     float s = phong(v, l, n, 64.0);
37
38     DIFFUSE_LIGHT += vec3(d) * 0.3;
39     SPECULAR_LIGHT += vec3(s);
40 }
```

In this code snippet, line 33 introduces a new variable **v** to store the view direction — necessary for computing the specular component with the **phong()** function. The result is stored in the scalar variable **s** (line 36) and added to **SPECULAR\_LIGHT** (line 39).

This modular approach will allow you to replace the Phong model with its Blinn-Phong variant in the next section, while keeping the rest of the logic intact.

(2.6.c Phong specular with  $m = 64.0$ )**Note**

In line 38, the diffuse light value is multiplied by 0.3 to reduce its intensity and ensure the specular component is clearly visible. Also, remember that Godot enables ambient lighting by default. If you'd like to disable it for more controlled testing, use the `ambient_light_disabled` render mode.

Now you'll implement the **Blinn-Phong** model. As mentioned earlier, this model changes the specular calculation by using an intermediate vector called **halfway** (or **half-vector**), which represents the average direction between the light and the view. This technique improves performance on certain platforms and creates smoother visual results, especially for soft highlights.

To implement it, define a new function named `blinn_phong()`, which corresponds to the equation in Figure 2.5.d:

```
float blinn_phong(vec3 v, vec3 l, vec3 n, float m)
{
    vec3 h = normalize(l + v);
    float s = pow(max(0.0, dot(n, h)), m);
    s *= float(dot(n, l) > 0.0);
    return s;
}
```

The line `vec3 h = normalize(l + v)` calculates the halfway vector by normalizing the sum of the light (`LIGHT`) and view (`VIEW`) directions. Then, the dot product between `n` (`NORMAL`) and `h` determines the specular intensity, raised to the exponent `m`, which controls the sharpness of the highlight. The result is multiplied by 1.0 or 0.0 depending on whether the angle between `n` and `l` is positive — preventing unwanted highlights in shadowed regions.

To apply this model, simply replace the `phong()` call with `blinn_phong()` inside the `light()` method. The updated code looks like this:

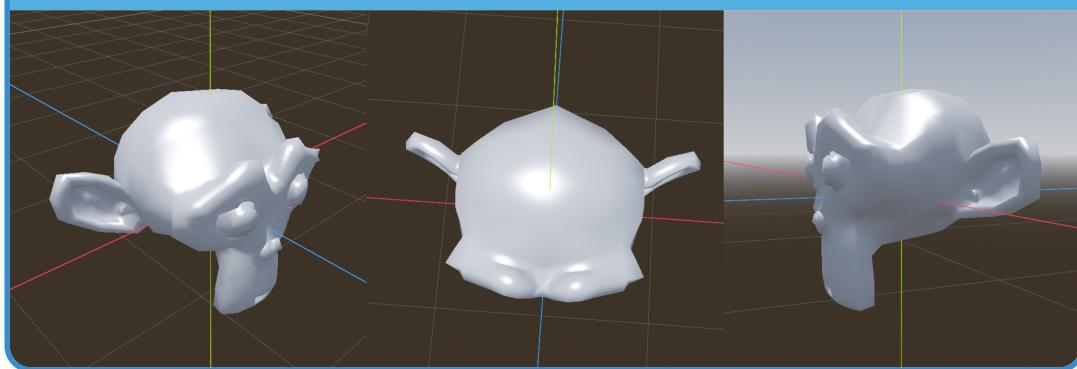
```

36 void light()
37 {
38     float i = ATTENUATION;
39     vec3 l = LIGHT;
40     vec3 n = NORMAL;
41     vec3 v = VIEW;
42
43     float d = lambert(i, l, n);
44     //float s = phong(v, l, n, 64.0);
45     float s = blinn_phong(v, l, n, 64.0);
46
47     DIFFUSE_LIGHT += vec3(d) * 0.3;
48     SPECULAR_LIGHT += vec3(s);
49 }
```

As shown in line 45, the `s` variable now uses `blinn_phong()` instead of `phong()`. Although the internal logic differs, both models share the same arguments:

- View direction `v`.
- Light direction `l`.
- Surface normal `n`.
- Shininess exponent `m`.

This change results in a visual output similar to Phong, but with subtle differences in the shape and distribution of the specular highlight.

(2.6.d Blinn-Phong specular with  $m = 64.0$ )

If you compare the results in Figures 2.6.c and 2.6.d, you'll notice that both models produce a similar effect. However, the Blinn-Phong highlight appears slightly broader and smoother. This difference stems from the use of the halfway vector instead of the reflection vector, which distributes the highlight's intensity differently across the surface.

Regardless of whether you use Phong or Blinn-Phong, keep in mind that both models operate in linear lighting space. Mathematically, this is correct. However, human vision does not perceive light linearly — we are more sensitive to changes in darkness than in brightness.

For this reason, when aiming for realistic lighting or a more visually striking result, it's recommended to convert your shader's output from **linear** space to **sRGB**, which better matches human perception. This conversion helps enhance highlights and improves the visual fidelity of your materials.

Since this section has already covered a lot, we'll explore color space and sRGB conversion in more detail in the next section.

## 2.7 From Linear Lighting to sRGB.

When we talk about **sRGB (Standard Red Green Blue)**, we're referring to the international standard IEC 61966-2-1, amendment 1 (2003), which precisely defines the behavior of the sRGB color space. This standard not only describes the color gamut that can be represented on a screen but also specifies how colors should be stored, interpreted, and displayed consistently across different devices.

Until this point in the chapter, you've explored various spaces and coordinate systems. But what exactly does "color space" mean? In simple terms, it refers to three key aspects:

- What colors can be represented.
- How those colors are stored in memory (encoding).
- How to interpret a color – for example, `vec3(0.143, 0.456, 0.673)` – consistently across different devices and displays.

Understanding this concept is crucial when developing shaders in Godot. By default, the images or textures you use (such as those exported from Photoshop in sRGB format) already include gamma correction. However, real-time lighting calculations must be done in **linear space** for physically correct results.

For this reason, it's common to find conversions between linear space and sRGB at different stages of the shader workflow. In practical terms, this means:

- When reading color values from albedo or diffuse textures (which are encoded in sRGB), you first need to convert them to linear space.
- When writing the final lighting result to the screen, you need to convert the colors back to sRGB so they appear correctly.

**Note**

sRGB color space should be used only for color images when exporting from editing software like Photoshop. Grayscale images (such as normal maps, roughness, etc.) must remain in linear space to preserve data accuracy in the [0.0 : 1.0] range.

To simplify these conversions in your shaders, you can define helper functions that handle the process automatically. Below are two functions that will let you convert between color spaces accurately:

To convert from sRGB to linear space:

```
vec3 to_linear(vec3 srgb)
{
    vec3 a = pow((srgb + 0.055) / 1.055, vec3(2.4));
    vec3 b = srgb / 12.92;
    bvec3 c = lessThan(srgb, vec3(0.04045));
    return mix(a, b, c);
}
```

To convert from linear space to sRGB:

```
vec3 to_sRGB(vec3 linearRGB)
{
    vec3 a = vec3(1.055) * pow(linearRGB.rgb, vec3(1.0/2.4)) - vec3(0.055);
    vec3 b = linearRGB.rgb * vec3(12.92);
    bvec3 c = lessThan(linearRGB, vec3(0.0031308));
    return vec3(mix(a, b, c));
}
```

These functions allow for accurate transformation between color spaces, ensuring that highlights and shadows in your materials are perceived correctly by the human eye.

Let's pay attention to the following reference to understand the concept:



(2.7.a Top: sRGB, Bottom: Linear)

To illustrate the conversion from **linear** to **sRGB** space, Figure 2.7.a uses the first channel of the UV coordinates (**uv.x**) as a reference. This example helps you visualize how luminance values are redistributed through gamma correction: mid-tones are brightened, while values close to black occupy a smaller portion of the dynamic range.

The same principle can be applied to any value calculated in linear space. In your case, you'll apply this conversion to the **specular component** to achieve a sharper and more visually appealing reflection.

However, keep in mind that the **to\_sRGB()** function returns a three-component vector (**vec3**), while the **blinn\_phong()** method returns a scalar value (**float**). To resolve this mismatch, you'll redefine the **s** variable as a three-dimensional vector and apply the color space conversion afterward.

The following code snippet shows this modification:

```

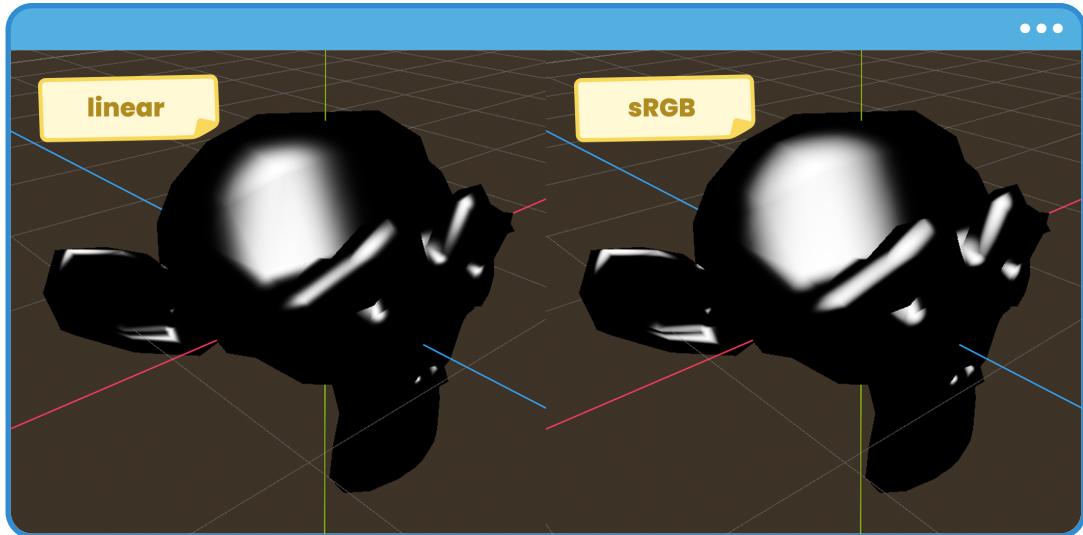
36 vec3 to_sRGB(vec3 linearRGB)
37 {
38     vec3 a = vec3(1.055) * pow(linearRGB.rgb, vec3(1.0/2.4)) - vec3(0.055);
39     vec3 b = linearRGB.rgb * vec3(12.92);
40     bvec3 c = lessThan(linearRGB, vec3(0.0031308));
41     return vec3(mix(a, b, c));
42 }
43
44 void light()
45 {
46     float i = ATTENUATION;
47     vec3 l = LIGHT;
48     vec3 n = NORMAL;
49     vec3 v = VIEW;
50
51     //vec3 d = vec3(lambert(i, l, n));
52     vec3 s = vec3(blinn_phong(v, l, n, 64.0));
53     s = to_sRGB(s);
54
55     DIFFUSE_LIGHT += vec3(0.0);
56     SPECULAR_LIGHT += s;
57 }
```

If you look at line 36, you'll see that the `to_sRGB()` function has been added to the shader. Then, in line 52, the variable `s` is defined as a `vec3` to store the result of the `blinn_phong()` function. In line 53, that value is converted from linear space to sRGB.

#### Note

In this example, diffuse light has been disabled (`DIFFUSE_LIGHT += vec3(0.0)`) to focus exclusively on the specular effect. It's also recommended to disable ambient lighting using the `ambient_light_disabled` render mode, allowing you to observe the impact of sRGB conversion on the highlight more clearly.

If everything has been implemented correctly, the visual result will show a more intense and noticeable specular highlight that better matches the behavior expected by the human eye.



(2.7.b Left: Linear specular; Right: sRGB specular)

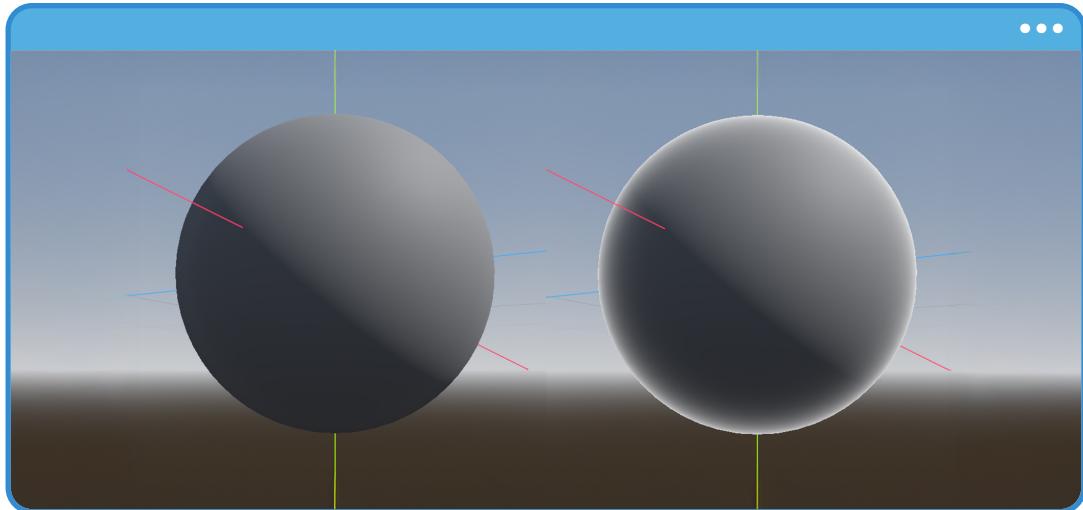
Although this example applies the linear-to-sRGB conversion only to the specular component, you can also apply it to the diffuse component, since it is also calculated in linear space.

Every game has its own visual style and unique rendering needs. So, feel free to experiment with these conversions and tweak the values to achieve a result that aligns with the artistic direction of your project.

## 2.8 Introduction to the Rim (Fresnel) Effect.

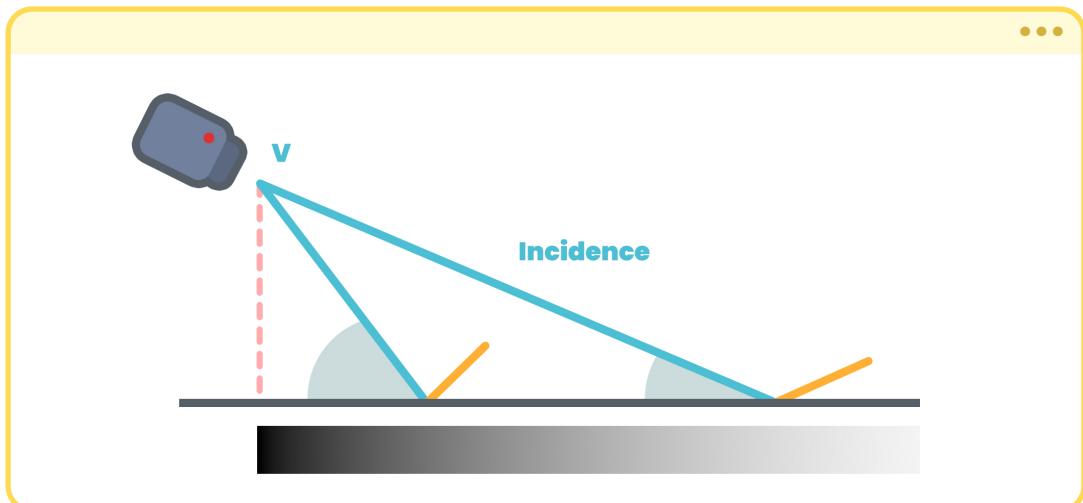
The **Rim** effect, also known as the **Fresnel effect** (named after French physicist Augustin-Jean Fresnel), is an optical phenomenon that describes how the amount of reflected light on a surface changes depending on the viewing angle. In practical terms, this effect appears as a glowing edge around the outer parts of an object when viewed from oblique angles.

If you were using a **StandardMaterial3D**, you could easily enable this effect through the **Rim** property, which is disabled by default. However, in this section, you'll learn how to implement the rim effect manually in a shader, giving you greater creative control and a deeper understanding of how it works internally.



(2.8.a Left: Lambert; Right: Rim)

How does the Fresnel effect work exactly? The principle behind the **Fresnel** effect is relatively simple: when viewing a surface head-on — that is, when the view vector and surface normal are parallel (with an angle close to  $0^\circ$ ) — the amount of reflected light is minimal or even nonexistent. In contrast, when you observe the surface at a **grazing angle** — when the view vector is perpendicular to the normal (with an angle close to  $90^\circ$ ) — reflection reaches its maximum value.



(2.8.b Diagram illustrating the Fresnel or Rim effect)

This behavior produces a noticeable highlight along the edges of objects. Visually, it appears as a luminous silhouette around the model, which can be used to enhance contours or to create stylized effects, similar to an outline or an artistic glow.

From a mathematical perspective, a common way to approximate the rim effect in shaders is with the following equation:

$$F = (1.0 - \max(0.0, n \cdot v))^m$$

(2.8.c)

Where  $n$  is the surface normal,  $v$  is the view vector normalized from the surface point to the camera, and  $m$  is the exponent that controls the sharpness of the effect. The lower the value of  $m$  (e.g.,  $m = 1.0$ ), the smoother and more spread out the rim effect will be. Conversely, higher values (e.g.,  $m > 5.0$ ) produce a thinner, sharper, and brighter edge.

You can implement this function directly in your .gdshader using the Godot Shader Language as shown below:

```
float rim(vec3 n, vec3 v, float m)
{
    float a = 1.0 - max(0.0, dot(n, v));
    float f = pow(a, m);
    return f;
}
```

(2.8.d Colors have been defined to identify each variable)

The image above shows both the mathematical equation for the **Rim (Fresnel)** effect and its direct implementation in shader code.

It's important to note that in Godot, this effect is already built into the material system. According to the official documentation, you can access the rim effect value from the `fragment()` function using the built-in **RIM** variable.

The behavior of **RIM** is influenced by the **ROUGHNESS** property, which affects how the illuminated edge is scattered. However, we'll explore this in more detail in the next section, where you'll learn how to manually integrate and customize the rim effect in a visual shader.

## 2.9 Implementing the Rim Effect.

In this section, you'll implement the **Rim** effect using two different approaches: first, by relying on Godot's built-in shader variables, and then by applying the custom function you defined in the previous section (Figure 2.8.d).

The goal is to understand both the simplified method provided by the engine and the underlying technical foundations of the effect, so you can apply it more flexibly in various contexts.

To achieve this, you'll follow these steps:

- We'll duplicate the **blinn\_phong** shader and rename it so you can work from that version.
- Also, we'll duplicate the **blinn\_phong.tscn** scene, keeping the Suzanne model as your reference.
- Finally, we'll implement both approaches to the Rim effect inside the shader to compare their behavior.

Before writing code, organize your folder structure to keep your project tidy. Proceed with the following:

- Inside **chapter\_02**, create a new folder called **rim\_fresnel**.
- Inside this folder, add the following subfolders:
  - **materials**.
  - **shaders**.

Since you'll continue using the Suzanne model, there's no need to add new meshes or textures.

Next, duplicate the `blinn_phong` shader and move it to the following path:

- `rim_fresnel > materials > shaders`.

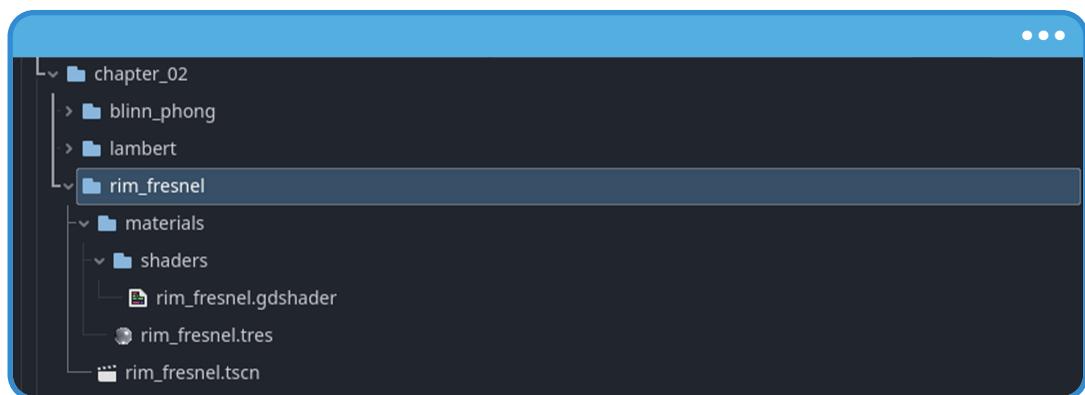
Once it's in place, rename the shader to **rim\_fresnel** to keep it consistent with the section name.

Now create a new **ShaderMaterial**. Right-click on the **materials** folder and select:

- ▶ Create New > Resource > ShaderMaterial.

Name the material **rim\_fresnel** as well, so its connection to the shader is clear and easy to identify.

If you've followed the steps correctly, your project structure should now look like this:



(2.9.a The shader and material have been added to the rim\_fresnel section)

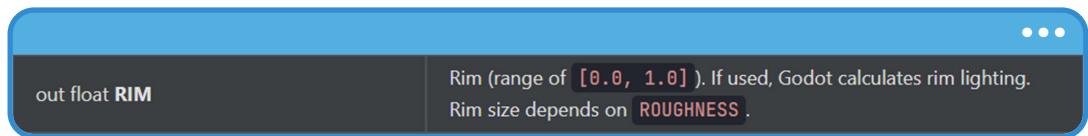
Before implementing the functions inside the shader, make sure everything is connected correctly. Assign the **rim\_fresnel** shader to its corresponding material, and then apply this material to the Suzanne model inside the **rim\_fresnel.tscn** scene. This step is essential to visualize the changes made during this section.

To understand how the Rim effect works in Godot, start by briefly reviewing the official documentation, where the built-in variables available in both the **fragment()** and **light()** functions are listed.

**Note**

You can find this information at the following link: [https://docs.godotengine.org/en/stable/tutorials/shaders/shader\\_reference/spatial\\_shader.html](https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/spatial_shader.html)

In the **Fragment built-ins** section, you'll find the internal **RIM** variable, which represents the intensity of the rim effect already calculated by the engine. You can use this variable directly in the **fragment()** function, as shown below:



(2.9.b RIM output variable)

However, you'll notice that **RIM** doesn't appear in the **Light built-ins** section of the documentation. Why is that?

In Godot, when working only with the **fragment()** function, you are responsible for assigning values directly to internal variables such as **ALBEDO**, **SPECULAR**, **EMISSION**, and others. In this mode, the engine automatically handles lighting calculations, and any modifications (such as applying the rim effect with **RIM**) are immediately reflected on screen.

In contrast, when you define the **light()** function, you activate **deferred lighting mode** for that material. This changes the shader's behavior: Godot disables part of the automatic logic in **fragment()** and expects you to manually calculate the lighting components. As a result, many internal variables like **RIM** no longer have a visual effect in **fragment()** when **light()** is present.

In other words:

- Without **light()**: the shader handles everything automatically. You only need to write code in **fragment()** and assign values to built-in variables.
- With **light()**: you must manually recalculate lighting for every light affecting the fragment. The **fragment()** function loses some of its visual impact.

**Note**

In gdshader documentation, many internal variables include qualifiers such as **in**, **out**, and **inout**, which indicate their behaviour in the pipeline:

- **in vec3 VIEW**: read-only in **fragment()**. Contains the direction from the fragment to the camera.
- **inout vec3 NORMAL**: comes interpolated from the **vertex()** function and can be read and modified in **fragment()** before lighting is calculated.

This means that if you use the **v** variable inside **fragment()** while **light()** is active, you won't see any effect, since the deferred lighting system ignores it.

To properly study how **RIM** works, it's best to begin by disabling the **light()** function. This way, you can observe how it behaves visually in **fragment()** without interference.

Here's how you'll proceed:

- Temporarily comment out the content of the **light()** function.
- Use the internal variables **RIM**, **ROUGHNESS**, and **RIM\_TINT** inside **fragment()** to observe the effect on the Suzanne model.

Comment out the **light()** function in the **rim\_fresnel** shader like this:

```

/*
void light()
{
    float i = ATTENUATION;
    vec3 l = LIGHT;
    vec3 n = NORMAL;
    vec3 v = VIEW;

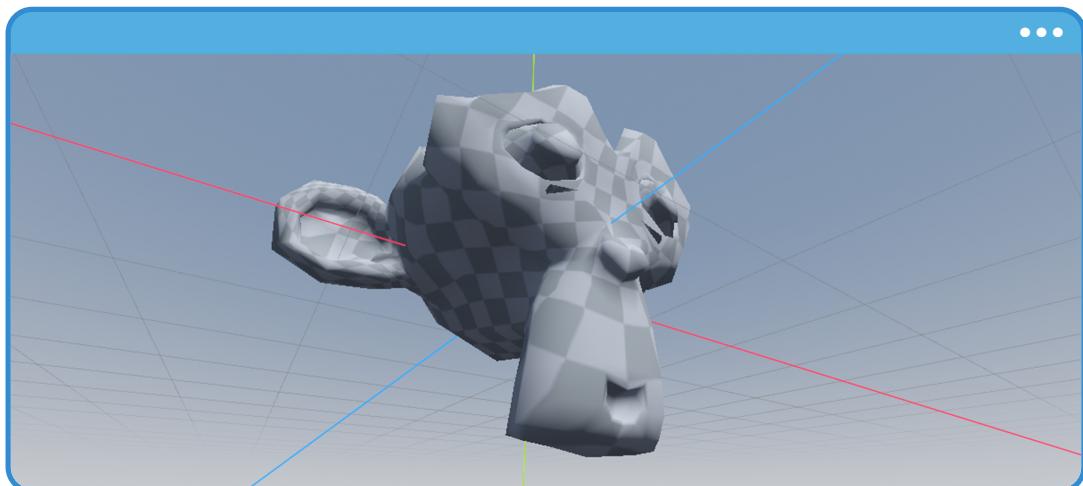
    vec3 d = vec3(lambert(i, l, n));
    vec3 s = vec3(blinn_phong(v, l, n, 128.0));
    s = to_sRGB(s);

    DIFFUSE_LIGHT += d;
    SPECULAR_LIGHT += s;
}

*/

```

As you can see, the entire **light()** function has been commented out, disabling its logic temporarily. Once you've made this change, your model should look like this:



(2.9.c Lighting is calculated from the fragment stage)

Even with the **light()** function commented out, your model still shows lighting — mainly from diffuse and ambient light. This happens because, without a custom **light()** function, Godot defaults to its internal lighting system within the **fragment()** stage.

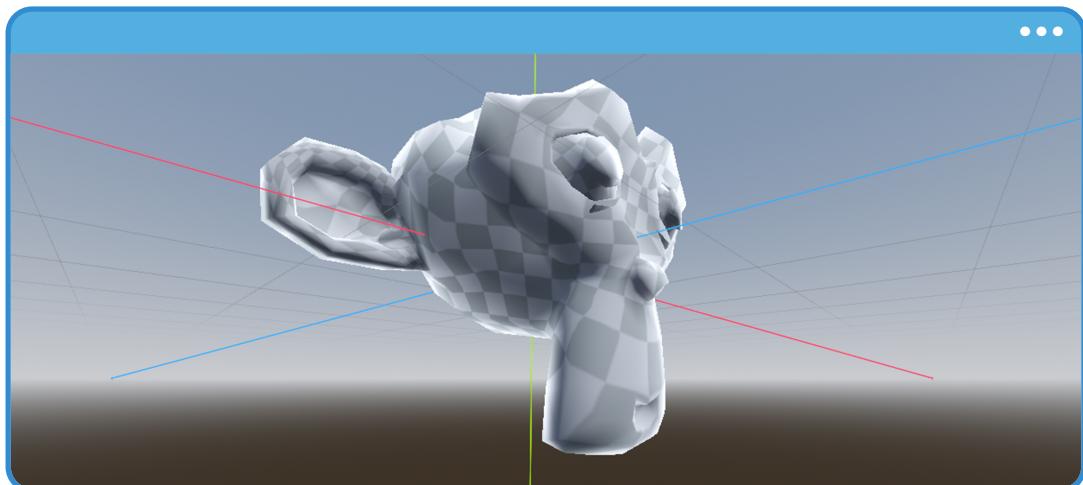
To enable the rim effect in this context, simply use the internal variables **RIM**, **ROUGHNESS**, and **RIM\_TINT** in the **fragment()** function. Here's a basic implementation:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;

    ROUGHNESS = 0.8;
    RIM = 1.0;
    RIM_TINT = 0.5;
    ALBEDO = albedo;
}
```

Let's break down the role of each variable:

- **RIM** controls the **intensity** of the Rim effect on the model: 0.0 disables it completely, while 1.0 applies it fully.
- **ROUGHNESS** affects the **thickness** of the illuminated edge. Values near 0.0 produce a sharp, narrow effect; values near 1.0 create a wider, softer effect.
- **RIM\_TINT** determines how the Rim effect **blends** with the material color. At 0.0, the rim is completely white. At 1.0, it blends with the material's color (overlay mode).



(2.9.d Rim effect applied to Suzanne in the fragment stage)

As shown above, applying the rim effect in the `fragment()` function is quite simple thanks to Godot's internal variables. However, if you now reactivate the `light()` function, you'll notice the effect disappears. This is because defining `light()` shifts responsibility for lighting to that function, and the values set in `fragment()` for properties like `RIM`, `ROUGHNESS`, or `RIM_TINT` no longer take effect.

To restore the rim effect while the `light()` function is active, you'll implement a custom version based on the equation shown in Figure 2.8.d. To do this:

- First, comment out the `RIM`, `ROUGHNESS`, and `RIM_TINT` lines in the `fragment()` function so they don't interfere with rendering.
- Then, inside the `light()` function, add the following code:

```
float fresnel(vec3 n, vec3 v, float m, float s)
{
    float f = 1.0 - max(0.0, dot(n, v));
    return s * pow(f, m);
}
```

Although this function was called `rim()` in Figure 2.8.d, the functionality remains the same. Here, a new argument `s` has been added to serve as an intensity multiplier, allowing for better visual control of the effect.

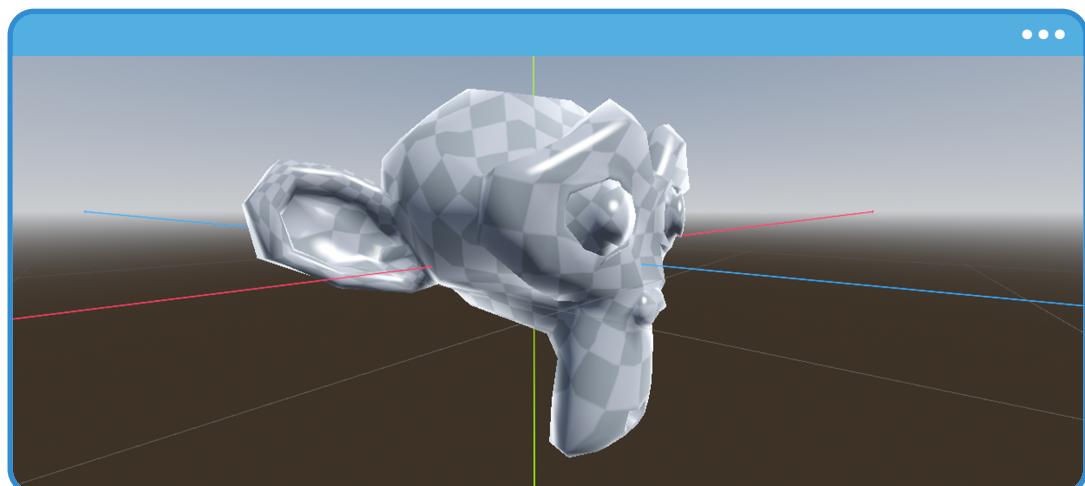
Now integrate the Fresnel effect directly into the `light()` function:

```

55 void light()
56 {
57     float i = ATTENUATION;
58     vec3 l = LIGHT;
59     vec3 n = NORMAL;
60     vec3 v = VIEW;
61
62     float f = fresnel(n, v, 5.0, 2.0);
63     vec3 d = vec3(lambert(i, l, n));
64     d += vec3(f);
65     vec3 s = vec3(blinn_phong(v, l, n, 128.0));
66     s = to_sRGB(s);
67
68     DIFFUSE_LIGHT += d;
69     SPECULAR_LIGHT += s;
70 }
```

Analyzing the code, in line 62 you define a new floating-point variable **f**, which stores the result of the **fresnel()** method. Then in line 64, this value is added to the diffuse component **d**, creating the visual rim highlight effect.

The values **m = 5.0** (exponent) and **s = 2.0** (multiplier) were chosen for demonstration purposes. Feel free to experiment with different values to achieve the style you want.



(2.9.e Rim effect implemented in the light stage)

## 2.10 Introduction to the Anisotropic Effect.

Up to this point, you've explored several specular reflection models to understand how to interpret and implement mathematical functions in the **.gdshader** language. However, the models used so far represent only a small subset of what's available.

When working with an anisotropic reflection model, the first question you should ask is: Which model will work best for my project? While it's true that several models can produce similar visual results under certain conditions, the differences can become significant when considering factors like visual finish, computational cost, or optimization.

Some common anisotropic specular models include:

- Ashikhmin-Shirley.
- Ward.
- Cook-Torrance.
- Anisotropic Phong.
- Anisotropic Blinn-Phong.

To keep the focus of this book accessible and practical, you'll examine two of these: first, the anisotropic **Phong** model, due to its simplicity and low computational cost; and then the **Ashikhmin-Shirley** model, which is more complex but offers a higher-quality visual result.

Start by analyzing the following mathematical expression:

$$A = \frac{(a_u + 1)(a_v + 1)}{8\pi} \cdot (\mathbf{h} \cdot \mathbf{n}) \frac{a_u(\mathbf{h} \cdot \mathbf{t})^2 + a_v(\mathbf{h} \cdot \mathbf{b})^2}{1 - (\mathbf{h} \cdot \mathbf{n})^2}$$

(2.10.a)

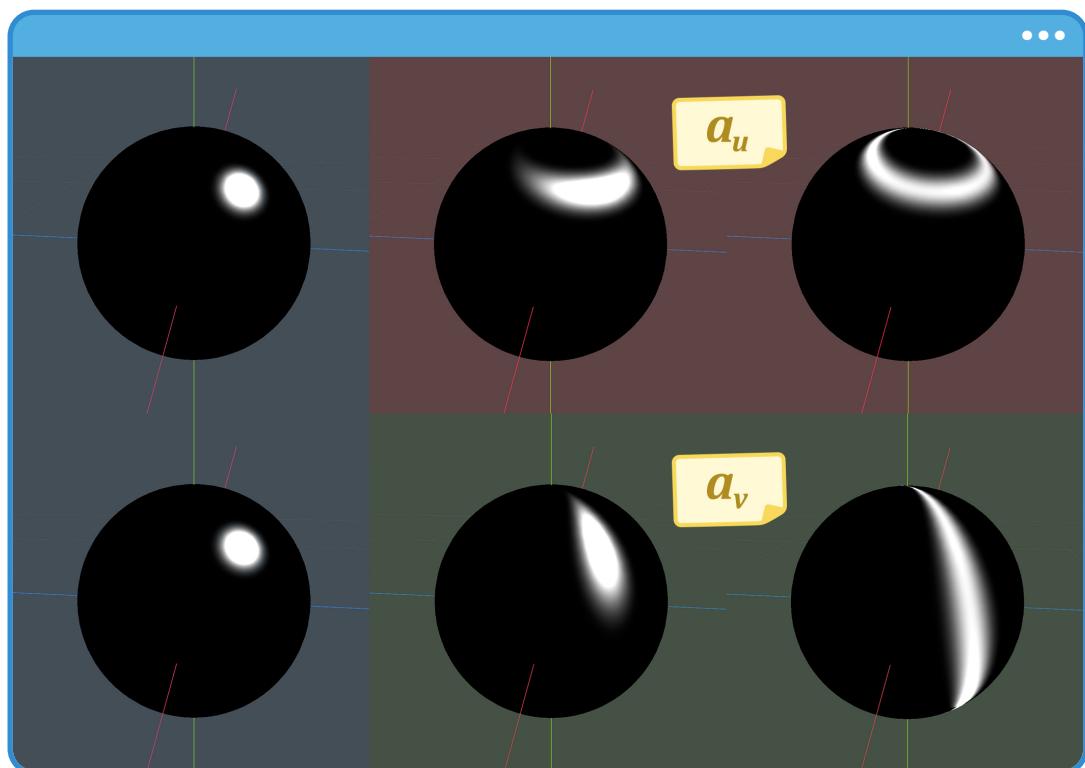
This equation is an extension of the classic Phong model, adapted to reflect anisotropy by using different exponents for the tangent and bitangent directions. In this formula:

- $a_u$  is the exponent along the tangent direction ( $t$ ).
- $a_v$  is the exponent along the bitangent direction ( $b$ ).
- $\mathbf{h}$  is the halfway vector (**LIGHT + VIEW**).

- $n$  is the normal (**NORMAL**).
- $t$  is the tangent (**TANGENT**).
- $b$  is the binormal (**BINORMAL**).

At first glance, this equation might seem complex, but the idea is straightforward: by adjusting the parameters  $a_u$  and  $a_v$ , you distort the circular shape of the specular highlight into an ellipse aligned to the surface's orientation. This distortion helps simulate materials such as brushed metal, hair, or glossy fabrics, where light reflects in a directional pattern.

Below is a visual illustration of how varying  $a_u$  and  $a_v$  affects the specular highlight:



(2.10.b Anisotropic specular deformation in both directions)

As shown in Figure 2.10.b, the  $a_u$  variable controls the **width** of the specular reflection, while  $a_v$  controls its **height**. In other words, increasing  $a_u$  extends the reflection horizontally along the tangent, and increasing  $a_v$  stretches it vertically along the bitangent.

If you want to implement Equation 2.10.a in your shader, you could do it like this:

$$A = \frac{(a_u + 1) (a_v + 1)}{8\pi} \cdot (\mathbf{h} \cdot \mathbf{n}) \frac{a_u (\mathbf{h} \cdot \mathbf{t})^2 + a_v (\mathbf{h} \cdot \mathbf{b})^2}{1 - (\mathbf{h} \cdot \mathbf{n})^2}$$

```
float phong_anisotropic(vec3 n, vec3 l, vec3 v, vec3 t,
vec3 b, float au, float av)
{
    vec3 h = normalize(l + v);
    float HdotN = dot(h, n);
    float HdotT = dot(h, t);
    float HdotB = dot(h, b);

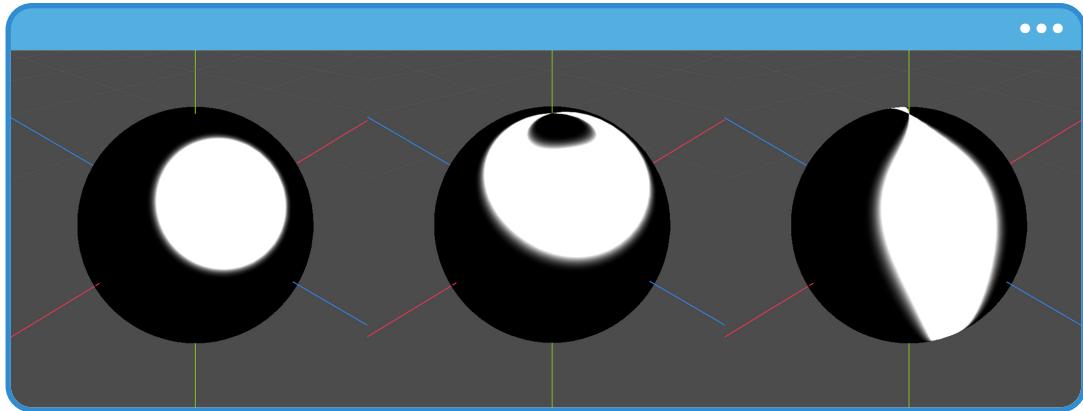
    float exponent = au * pow(HdotT, 2.0) + av * pow(HdotB, 2.0);
    exponent /= 1.0 - pow(HdotN, 2.0);
    float specular = ((au + 1.0) * (av + 1.0)) / 8.0 * PI;
    float a = dot(specular, pow(HdotN, exponent));

    return a;
}
```

(2.10.c Colors have been defined to identify each variable)

This model allows you to deform the shape of the specular reflection in a controlled way by adjusting  $a_u$  and  $a_v$  to achieve the desired effect. While it's not a physically accurate model and doesn't conserve energy precisely, its low computational cost makes it suitable for stylized rendering or non-photorealistic effects.

You can now visualize the effect on a sphere:



(2.10.d Anisotropic specular reflection on a sphere)

As seen in Figure 2.10.d, the anisotropic effect produces a highly directional specular reflection whose shape depends on the values assigned to  $a_u$  and  $a_v$ . This deformation of the specular lobe is useful for representing materials where reflection is distributed unevenly.

However, the full implementation above can be optimized using a simpler and more efficient approximation that still yields a visually acceptable result in performance-sensitive contexts.

This simplified version is defined by the following expression:

$$A = \max(n \cdot h, 0.001) a_u (h \cdot t)^2 + a_v (h \cdot b)^2$$

(2.10.e)

Which, when implemented in code, looks like this:

$$A = \max(n \cdot h, 0.001)$$

$a_u(h \cdot t)^2 + a_v(h \cdot b)^2$

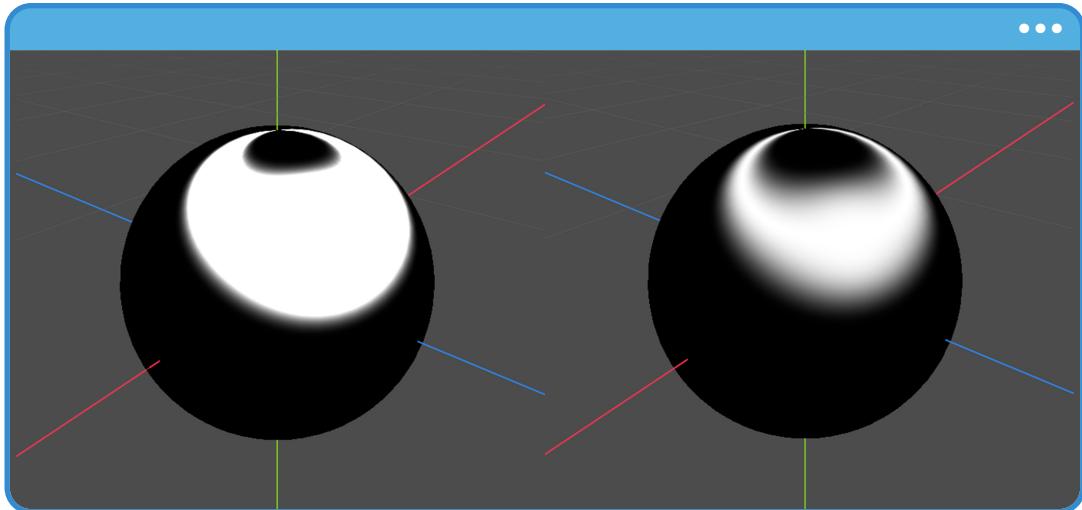
```
float phong_anisotropic_simplified(vec3 n, vec3 l, vec3 v,
vec3 t, vec3 b, float au, float av)
{
    vec3 h = normalize(l + v);
    float HdotT = dot(h, t);
    float HdotB = dot(h, b);
    float NdotH = max(dot(n, h), 0.001);

    float exponent = au * pow(HdotT, 2.0) + av * pow(HdotB, 2.0);
    float a = pow(NdotH, exponent);

    return a;
}
```

(2.10.f Colors have been defined to identify each variable)

Unlike the first version (based on Equation 2.10.a), this approximation produces a smoother and more efficient specular reflection without requiring additional divisions. In practice, this kind of simplification is ideal for stylized effects or low-end devices, where computational savings make a significant impact.



(2.10.g Left: Classic Phong; Right: Simplified Phong)

As shown in Figure 2.10.g, the classic Phong model (left) produces a sharper, more focused specular lobe, while the simplified version (right) softens the result by removing the normalization factor. Although both models are useful in stylized contexts, if you require a more accurate and physically grounded representation of anisotropy, you'll need to rely on more advanced models.

One of the most prominent models in this category is **Ashikhmin-Shirley**, whose mathematical formulation is defined as follows:

$$A = \frac{\frac{a_u(h \cdot t)^2 + a_v(h \cdot b)^2}{\sqrt{(a_u + 1)(a_v + 1)}(n \cdot h)}}{8\pi(v \cdot h) \max(n \cdot l, n \cdot v)}$$

(2.10.h)

This model evolves from the anisotropic Phong model but introduces several critical elements that bring it closer to real-world light behavior over rough surfaces:

- The numerator includes a square root normalization factor to ensure energy conservation.
- The exponent applied to  $n \cdot h$  takes into account the orientation of the halfway vector with respect to the tangent and bitangent, enabling precise deformation of the specular lobe.

- The denominator introduces view-angle dependence through  $v \cdot h$  and the maximum of  $n \cdot l$  and  $n \cdot v$ , improving angular reflectance response.

What makes this model especially interesting is that it strikes a balance between visual precision and artistic control, making it suitable for creating realistic materials with directional microstructure.

The original **Ashikhmin-Shirley** model also includes an approximation of the Fresnel effect, using **Schlick's** formula, defined as:

$$F = r_s + (1 - r_s)(1 - (v \cdot h))^5$$

(2.10.i)

This term improves the material's behavior at grazing angles by increasing the intensity of the specular reflection as the view direction moves away from the surface normal. However, its use is optional depending on the visual style you're aiming for. Both the **Ashikhmin-Shirley** model and the Fresnel term can be implemented in a single method in Godot as shown below:

$$A = \frac{\sqrt{(a_u + 1)(a_v + 1)}(n \cdot h) \frac{a_u(h \cdot t)^2 + a_v(h \cdot b)^2}{1 - (n \cdot h)^2}}{8\pi (v \cdot h) \max(n \cdot l, n \cdot v)} (r_s + (1 - r_s)(1 - (v \cdot h))^5)$$

```
float ashikhmin_shirley_anisotropic(float rs, float au, float av,
vec3 n, vec3 l, vec3 v, vec3 t, vec3 b)
{
    vec3 h = normalize(l + v);
    float NdotH = max(dot(n, h), 0.0001);
    float NdotL = max(dot(n, l), 0.0001);
    float NdotV = max(dot(n, v), 0.0001);
    float VdotH = max(dot(v, h), 0.0001);

    float HdotT = dot(h, t);
    float HdotB = dot(h, b);

    float exponent = au * pow(HdotT, 2.0) + av * pow(HdotB, 2.0);
    exponent /= 1.0 - pow(NdotH, 2.0);

    float a = sqrt((au + 1.0) * (av + 1.0)) * pow(NdotH, exponent);
    a /= (8.0 * PI) * VdotH * max(NdotL, NdotV);
    float f = rs + (1.0 - rs) * pow(1.0 - VdotH, 5.0);
    a *= f;

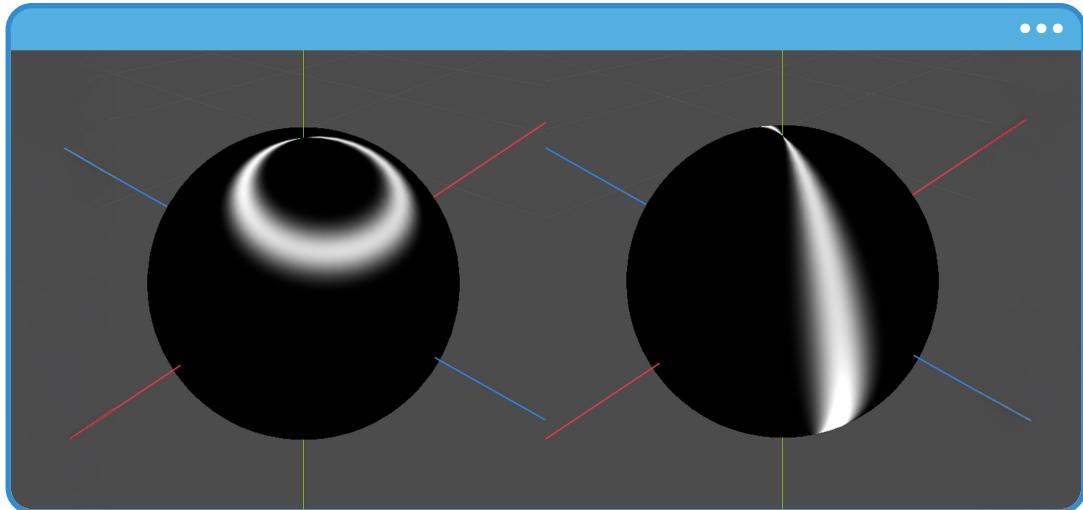
    return a;
}
```

(2.10.j Colors have been defined to identify each variable)

In this implementation:

- $r_s$  represents the **base reflectance**.
- $a_u$  and  $a_v$  control anisotropy along the **tangent** and **bitangent** directions, respectively.
- The formula includes a **correction term** to avoid division by zero and improve numerical stability.

The resulting visual effect is shown in the next figure:



(2.10.k Ashikhmin-Shirley model)

As you can see, this model produces a sharper and more directional specular lobe, whose shape depends on the orientation of the halfway vector with respect to the tangent and bitangent. This level of precision makes it especially useful for simulating materials like hair, satin fabrics, brushed metals, or any surface with a structured, realistic directional reflection.

## 2.11 Implementing the Ashikhmin-Shirley Model.

In this section, you'll put the **Ashikhmin-Shirley** model into practice by applying it to the hair of an anime-style character. The goal is to observe how this model behaves on more complex geometry and how its anisotropic component can enhance the visual style of the render — especially for materials like hair, which tend to reflect light directionally.

To begin, let's organize the project to maintain a clear and consistent structure. Follow these steps:

- Inside the **chapter\_02** folder, create a new folder called **anisotropy**.
- Inside this folder, organize the content as follows:
  - **materials**.
  - **shaders**.
  - **meshes**.
  - **textures**.

Next, create the material that will be applied to the character's hair. Right-click on the materials folder and select:

- Create New > Resources > ShaderMaterial.

Name this material **character\_hair**.

Then, create the corresponding shader by right-clicking on the **shaders** folder and selecting:

- Create New > Resources > Shader.

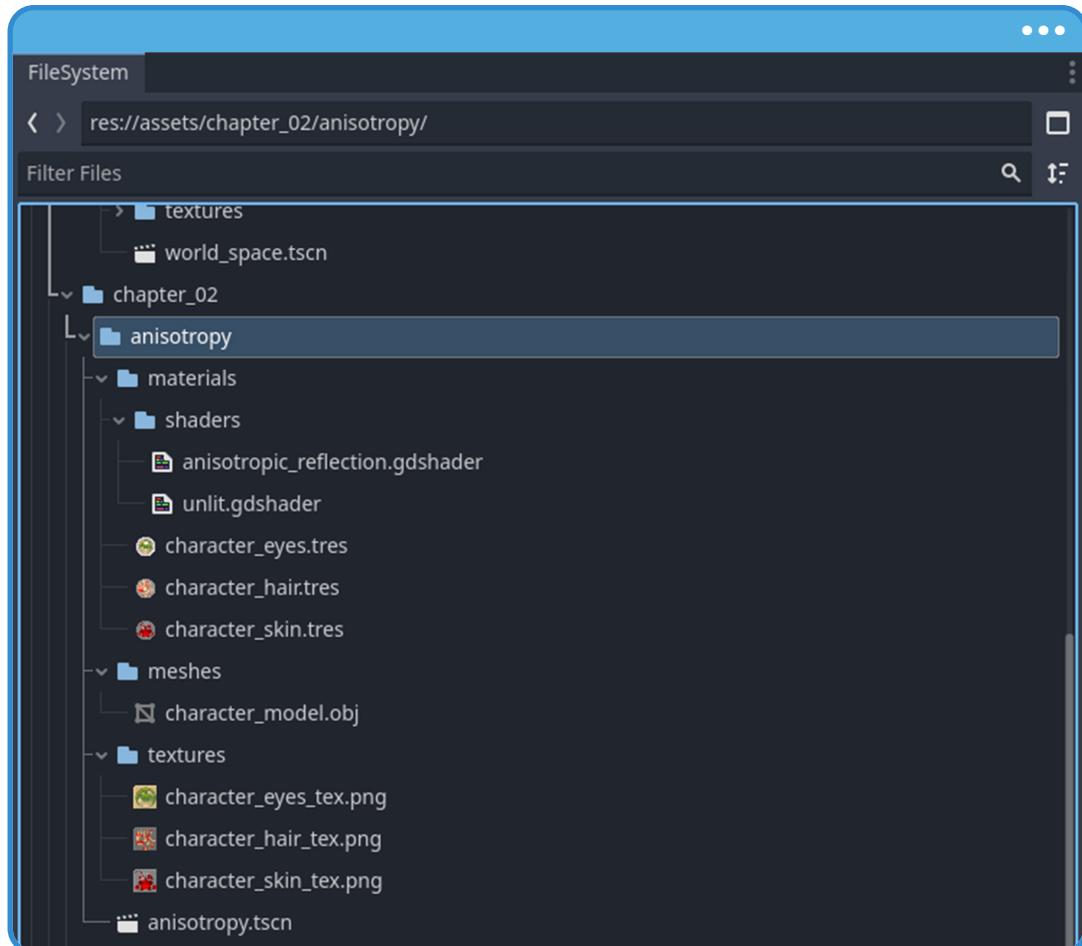
Name this file **anisotropic\_reflection**, keeping it consistent with the section title.

For this section, you'll use an anime-style character along with its corresponding textures and materials, all of which are available as part of the book's downloadable package.

**Note**

This book includes a downloadable package that contains all the necessary files to follow along with each exercise. You can download it directly from: <https://jettelly.com/store/the-godot-shaders-bible>

If everything has been configured correctly, your project structure should now look like this:



(2.11.a Project structure)

Begin the implementation by dragging the **character\_model** into the Viewport. Make sure to center its position by setting its transform values to  $(0.0_x, 0.0_y, 0.0_z)$ .

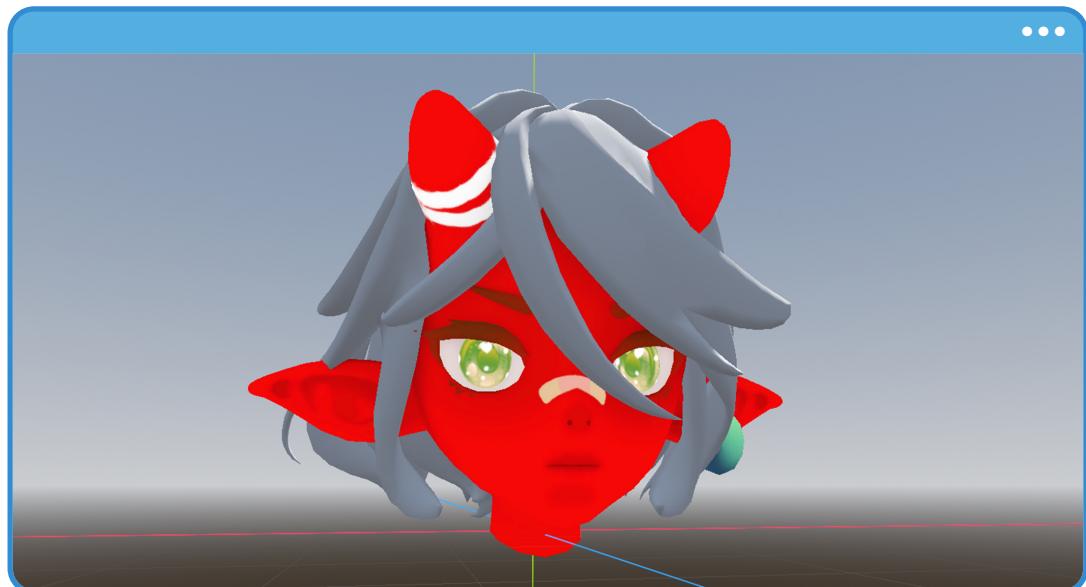
An interesting aspect of this model is how its materials are configured through the **Surface Material Override** property. If you select the MeshInstance3D node (named CharacterModel), you'll see that it includes **three material slots**, labeled 0, 1, and 2.

These slots correspond to the materials assigned in the 3D modeling software used to create the asset (in this case, **Maya**):

- Slot **0** refers to the character's eyes. Assign the **character\_eyes** material here.

- Slot **1** corresponds to the hair. Assign the **character\_hair** material to this slot. Make sure the **anisotropic\_reflection** shader is already assigned to the material before applying it to the model — just as you've done in previous sections.
- Slot **2** refers to the character's skin. Assign the **character\_skin** material in this slot.

Once the materials are assigned correctly — and since no code has been added to the **anisotropic\_reflection** shader yet — the character's hair will appear completely flat, without any anisotropic reflection. Its appearance in the **Viewport** should look like this:



(2.11.b The materials have been assigned to the character)

Obviously, this will change once you start implementing the necessary functions in your shader. To do that, open the file **anisotropic\_reflection.gdshader** and go to the **fragment()** function.

Next, add the following lines of code:

```
shader_type spatial;

uniform sampler2D _MainTex;

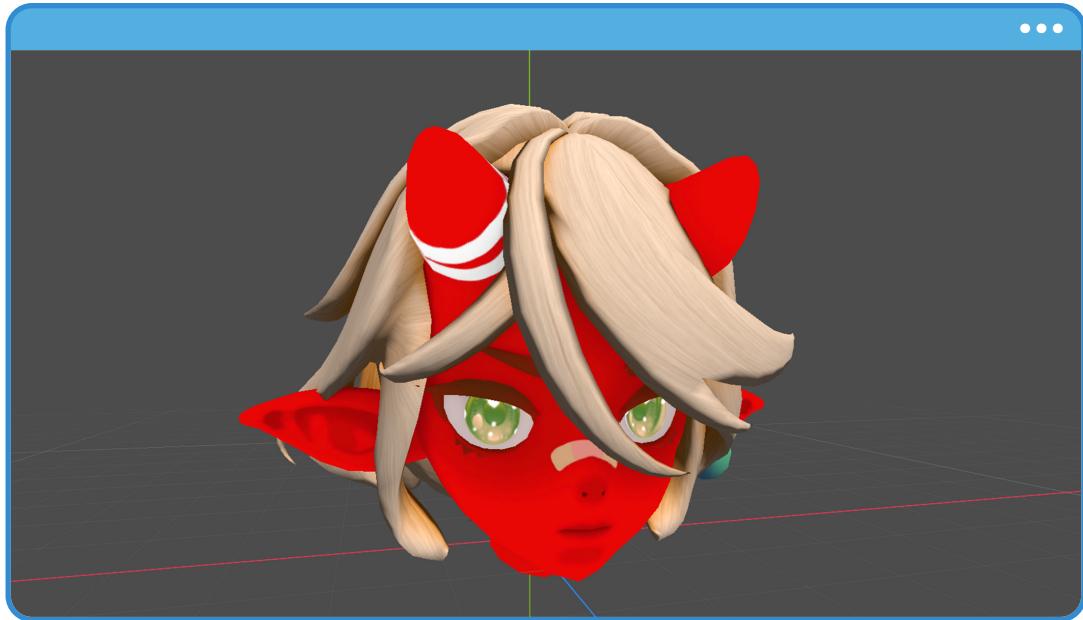
void vertex()
{
    // Called for every vertex the material is visible on.
}

void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}

//void light() {
//    // Called for every pixel for every light affecting the material.
//    // Uncomment to replace the default light processing function ...
//}
```

As you can see, you've declared a new three-dimensional vector named `albedo`, which stores the result of sampling the `_MainTex` texture using the model's `UV` coordinates. This value is then assigned to the built-in `ALBEDO` variable, allowing the texture color to be correctly applied to the geometry.

After making this change, assign the texture `character_hair_tex` to the **Main Tex** property in the **Inspector**, under the `character_hair` material. This texture contains the base color of the character's hair.



(2.11.c The texture has been assigned to the character's hair)

Although the current result does not yet reflect the effect we're aiming for, it serves as a starting point to understand the implementation we'll carry out in this section. The **Ashikhmin-Shirley** model needs to be implemented inside the `light()` function, but before going back to the shader, we need to add a light source to the scene. To do this, add a `DirectionalLight3D` by selecting the `Node3D` and choosing:

- ▶ Add Child Node > DirectionalLight3D

This will give us control over the direction and intensity of the light, which is essential to observe how the reflection behaves on the character's hair.

Returning to the `.gdshader` file, enable the `light()` function by uncommenting it. When doing this, you'll notice the character's hair becomes unlit. This happens because all lighting calculations are now delegated to this function, and we haven't defined any logic inside it yet. Despite this, the character will still be affected by ambient light, which is handled automatically.

To properly structure our implementation, we'll begin by separating the necessary functions into blocks, which will make the code easier to understand and maintain. We'll use the equation presented in **Figure 2.10.j** from the previous section as our base.

Declare the function just between the `fragment()` and `light()` methods. For practical purposes, we'll call it `ashikhmin_shirley()`:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}

float ashikhmin_shirley()
{
    return 0.0;
}

void light()
{
```

This model requires several variables to perform its calculations. Below is the list of arguments we'll pass to the function, which include reflection factors, orientation vectors, and geometric surface properties:

- $r_s$ : Reflection factor.
- $a_u$ : tangent direction.
- $a_v$ : binormal direction.
- $n$ : the **NORMAL** variable.
- $v$ : the **VIEW** variable.
- $l$ : the **LIGHT** variable.
- $t$ : the **TANGENT** variable.
- $b$ : the **BINORMAL** variable.

Next, we'll include these variables as arguments in the function:

```
float ashikhmin_shirley(float rs, float au, float av, vec3 n, vec3 l, vec3 v,
    vec3 t, vec3 b)
{
    return 0.0;
}
```

In the next step, we'll implement the body of this function by dividing it into three logical blocks, based on the mathematical formula presented in **Figure 2.10.j**. This breakdown will allow us to interpret and develop the model step by step in a clear and structured way.

$$A = \frac{\sqrt{(a_u+1)(a_v+1)}(n \cdot h) \frac{a_u(h \cdot t)^2 + a_v(h \cdot b)^2}{1 - (n \cdot h)^2}}{8\pi(v \cdot h)\max(n \cdot l, n \cdot v)} (r_s + (1 - r_s)(1 - (v \cdot h))^5)$$

**Specular =**

$$A = \frac{\sqrt{(a_u+1)(a_v+1)}(n \cdot h)}{8\pi(v \cdot h)\max(n \cdot l, n \cdot v)}$$

**Exponent =**

$$\frac{a_u(h \cdot t)^2 + a_v(h \cdot b)^2}{1 - (n \cdot h)^2}$$

**Fresnel =**

$$(r_s + (1 - r_s)(1 - (v \cdot h))^5)$$

(2.11.d The anisotropic model functions have been separated)

To implement the model, we'll use the general equation shown in **Figure 2.11.d** as our foundation. One strategy we can follow is to break it down into smaller blocks. For example, if we look at the specular equation, we'll notice several operations based on the dot product:

- $n \cdot l$  ( $N_{dotL}$ ).
- $n \cdot v$  ( $N_{dotV}$ ).
- $n \cdot h$  ( $N_{dotH}$ ).
- $v \cdot h$  ( $V_{dotH}$ ).

Since some of these variables are used in both the exponent and the Fresnel term, it's most efficient to define them at the beginning of our function:

```
float ashikhmin_shirley(float rs, float au, float av, vec3 n, vec3 l, vec3 v,
    ↵  vec3 t, vec3 b)
{
    vec3 h = normalize(l + v);
    float NdotL = max(dot(n, l), 0.0001);
    float NdotV = max(dot(n, v), 0.0001);
    float NdotH = max(dot(n, h), 0.0001);
    float VdotH = max(dot(n, h), 0.0001);

    return 0.0;
}
```

There are two important points in this initial block:

- The variable **h** is the **halfway vector**, calculated as a normalized sum of the light and view directions. Normalization ensures its length is exactly 1.0.
- The use of **max( ..., 0.0001)** avoids values close to or equal to 0.0, which could cause visual errors or rendering artifacts, especially at grazing angles.

Next, we can incorporate the numerator and denominator of the model, leaving the exponent undefined for now:

```
float ashikhmin_shirley(float rs, float au, float av, vec3 n, vec3 l, vec3 v,
    vec3 t, vec3 b)
{
    vec3 h = normalize(l + v);
    float NdotL = max(dot(n, l), 0.0001);
    float NdotV = max(dot(n, v), 0.0001);
    float NdotH = max(dot(n, h), 0.0001);
    float VdotH = max(dot(n, h), 0.0001);

    float exponent; // <-- not initialized

    float specular = sqrt((au + 1.0) * (av + 1.0)) * pow(NdotH, exponent);
    specular /= (8.0 * PI) * VdotH * max(NdotL, NdotV);

    return 0.0;
}
```

According to **Figure 2.11.d**, the exponent depends on three components:

- $h \cdot t$  (HdotT).
- $h \cdot b$  (HdotB).
- $n \cdot h$  (NdotH).

We've already defined  $(n \cdot h)$ , so we only need to add  $(h \cdot t)$  and  $(h \cdot b)$ , as shown below:

```

float ashikhmin_shirley(float rs, float au, float av, vec3 n, vec3 l, vec3 v,
    vec3 t, vec3 b)
{
    vec3 h = normalize(l + v);
    float NdotL = max(dot(n, l), 0.0001);
    float NdotV = max(dot(n, v), 0.0001);
    float NdotH = max(dot(n, h), 0.0001);
    float VdotH = max(dot(n, h), 0.0001);

    float HdotT = dot(h, t);
    float HdotB = dot(h, b);

    float exponent = au * pow(HdotT, 2.0) + av * pow(HdotB, 2.0);
    exponent /= 1.0 - pow(NdotH, 2.0);

    float specular = sqrt((au + 1.0) * (av + 1.0)) * pow(NdotH, exponent);
    specular /= (8.0 * PI) * VdotH * max(NdotL, NdotV);

    return 0.0;
}

```

While the Ashikhmin–Shirley model can function without the Fresnel term, we'll include it to complete the equation and improve the specular response on object edges:

```

float specular = sqrt((au + 1.0) * (av + 1.0)) * pow(NdotH, exponent);
specular /= (8.0 * PI) * VdotH * max(NdotL, NdotV);

float f = rs + (1.0 - rs) * pow(1.0 - VdotH, 5.0);
specular *= f;

return specular;

```

Now, to use this function inside the `light()` method, we must keep in mind the following:

- The `TANGENT` and `BINORMAL` variables are not directly available in `light()`. To solve this, we'll declare them as `varying` variables and assign their values inside the `fragment()` method.
- The `rs`, `au`, and `av` variables must be declared as uniform so they can be adjusted from the Inspector.

Let's proceed to declare and initialize the required global and uniform variables so our shader can properly use the anisotropic model inside `light()`:

```

1  shader_type spatial;
2
3  uniform sampler2D _MainTex;
4  uniform float _AU : hint_range(0.0, 256.0, 0.1);
5  uniform float _AV : hint_range(0.0, 256.0, 0.1);
6  uniform float _ReflectionFactor : hint_range(0.0, 1.0, 0.1);
7
8  varying vec3 _tangent;
9  varying vec3 _binormal;
10
11 void vertex() { ... }
15 //Line break (12 to 14)
16 void fragment()
17 {
18     _tangent = TANGENT;
19     _binormal = BINORMAL;
20
21     vec3 albedo = texture(_MainTex, UV).rgb;
22     ALBEDO = albedo;
23 }
```

If we examine the code, we'll notice the material properties are declared between lines 4 and 6. Then, in lines 8 and 9, the `varying` variables that will store the tangent and binormal vectors are declared. These are initialized in the `fragment()` method, as shown in lines 18 and 19. This ensures that the `TANGENT` and `BINORMAL` vectors, which are unavailable inside `light()`, can be accessed through `_tangent` and `_binormal`.

Now that everything is correctly set up, we can go to the `light()` method and use the `ashikhmin_shirley()` model. To do this, we declare a new float variable called `aniso`, and initialize it with the result of the method. The code block looks like this:

```
void light()
{
    float aniso = ashikhmin_shirley(_ReflectionFactor, _AU, _AV,
        NORMAL, LIGHT, VIEW, _tangent, _binormal);

    SPECULAR_LIGHT += aniso;
}
```

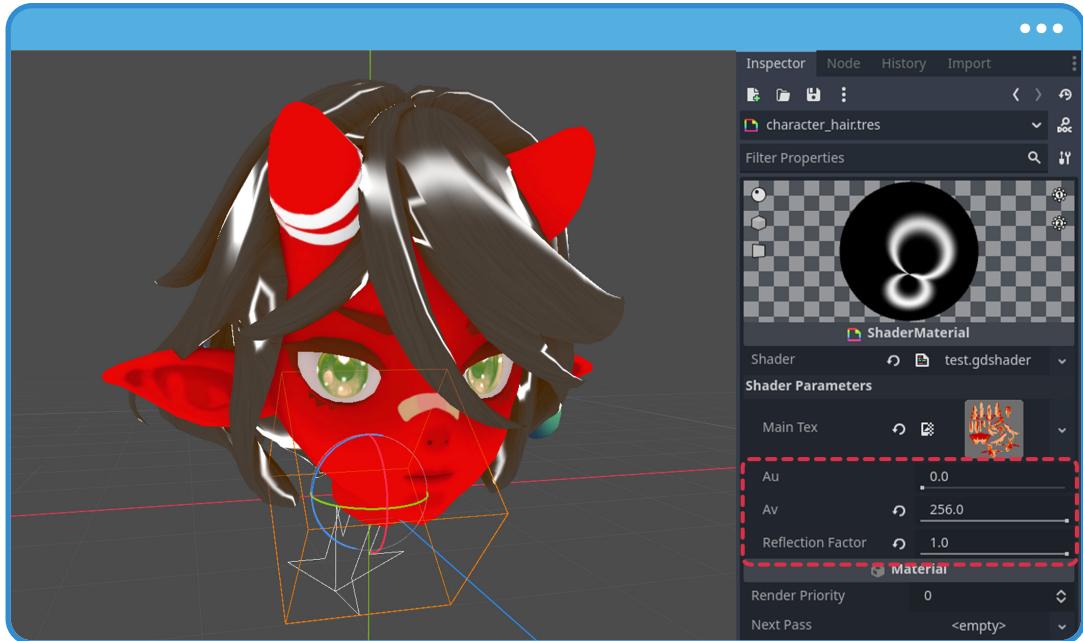
Since anisotropic reflection is an advanced form of specularity, the result is directly assigned to the `SPECULAR_LIGHT` output.

To check how this reflection affects our character in the Viewport, follow these steps:

- ▶ Select the `character_hair` material.
- ▶ Go to the Inspector.
- ▶ Set one of the properties, `Au` or `Av`, to its maximum value (e.g., 256.0).
- ▶ Keep the other property (`Au` or `Av`) at 0.0.
- ▶ Set the **Reflection Factor** to 1.0.

After applying these adjustments, the character's hair will begin to show anisotropic highlights whose shape depends on the orientation of the tangent vector assigned to the model.

The result in the Viewport should look as follows:



(2.11.e Material properties have been modified)

However, the anisotropic reflection presents a few rendering issues. For example:

- The reflection affects the hair uniformly, regardless of the light direction. This is an issue because if we look at the underside of the character's hair, we'll notice that reflections are still rendered in that area.
- The reflection remains consistent with the 3D model's shape. However, it should be distorted according to the hair texture.
- The hair is still only affected by ambient lighting, so it continues to look flat.

We'll address these problems in our shader, starting with the first one.

To prevent the reflection from affecting areas that should remain in shadow, we'll multiply the result of the anisotropic model by the **Lambert** diffuse term, which is useful because:

- It returns values close to 1.0 (white) when the normal points in the same direction as the light.
- It returns values close to 0.0 (black) when the normal opposes the direction of the light.

This allows us to automatically attenuate reflections in regions with no direct lighting.

```

float lambert(float i, vec3 l, vec3 n)
{
    return max(0.0, dot(n, l)) * i;
}

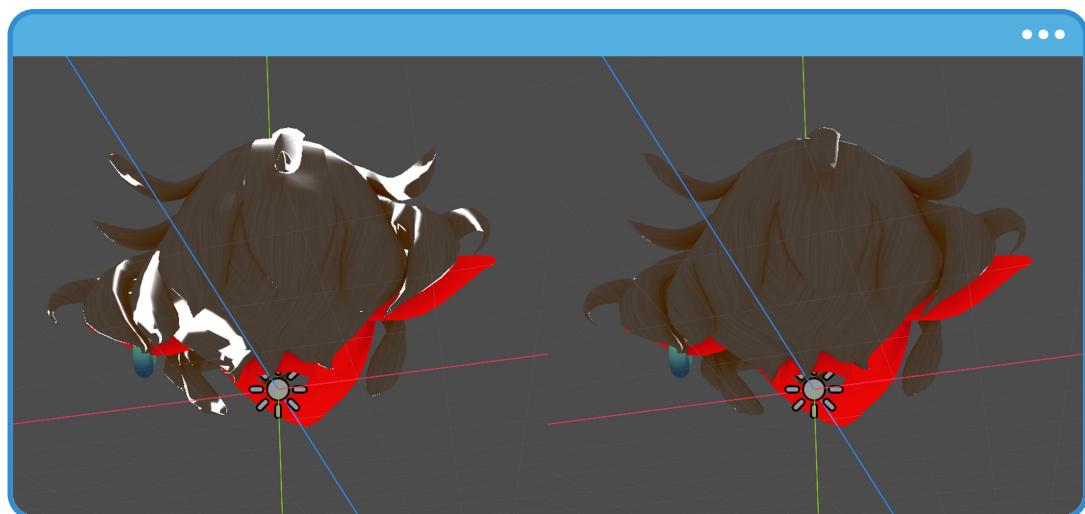
void light()
{
    float aniso = ashikhmin_shirley(_ReflectionFactor, _AU, _AV,
        NORMAL, LIGHT, VIEW, _tangent, _binormal);
    aniso *= lambert(ATTENUATION, LIGHT, NORMAL);

    SPECULAR_LIGHT += aniso;
}

```

By implementing the Lambert model in our code, the graphical result will better match the scene's lighting conditions. Specular highlights will now fade out in shadowed areas, producing a more realistic result.

The render should now look like this:



(2.11.f Left: `aniso` only; Right: `aniso` multiplied by Lambert)

As you can see in **Figure 2.11.f**, applying the Lambert model helps eliminate reflections in areas that should be shadowed.

To solve the second issue mentioned earlier, we need to modulate the anisotropic reflection using the albedo channels. This is done by declaring a new `vec3` inside the `light()` method and copying the current values of the `ALBEDO` variable:

```
void light()
{
    float aniso = ashikhmin_shirley(_ReflectionFactor, _AU, _AV,
NORMAL, LIGHT, VIEW, _tangent, _binormal);

    vec3 albedo = ALBEDO;
    albedo *= albedo.r * albedo.g * albedo.b;
    aniso = smoothstep(0.0, 0.1, aniso);
    aniso *= lambert(ATTENUATION, LIGHT, NORMAL);

    SPECULAR_LIGHT += aniso;
}
```

As you can see, the anisotropic effect is modulated through the operation `aniso *= albedo.r * albedo.g * albedo.b`. This multiplies the reflection value by the combined intensity of the RGB color channels, allowing us to attenuate the specular component based on the overall brightness of the texture.

To address the third issue, we simply assign the value of `albedo` directly to the `DIFFUSE_LIGHT` output variable, which allows us to see the texture's colors in their original tones:

```

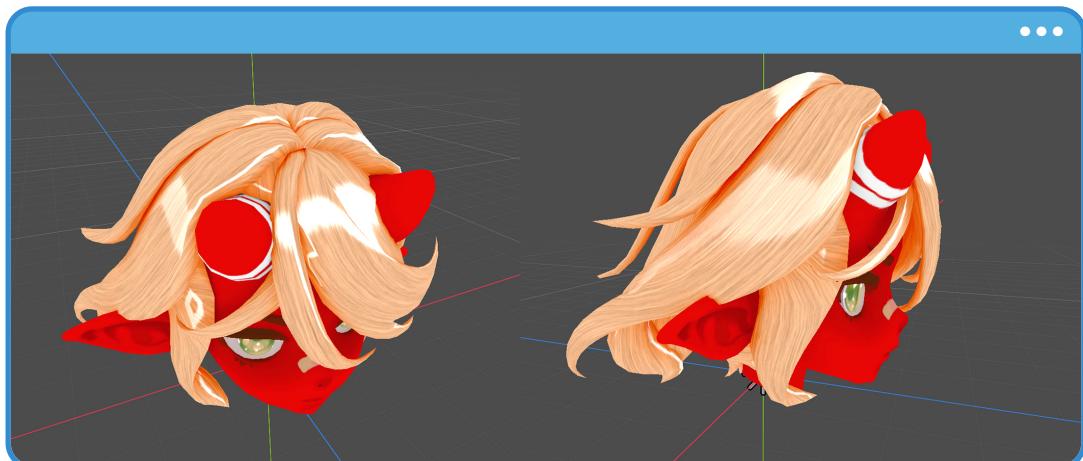
void light()
{
    float aniso = ashikhmin_shirley(_ReflectionFactor, _AU, _AV,
NORMAL, LIGHT, VIEW, _tangent, _binormal);

    vec3 albedo = ALBEDO;
    aniso *= albedo.r * albedo.g * albedo.b;
    aniso = smoothstep(0.0, 0.1, aniso);
    aniso *= lambert(ATTENUATION, LIGHT, NORMAL);

    DIFFUSE_LIGHT += albedo;
    SPECULAR_LIGHT += aniso;
}

```

This configuration produces the following visual result:

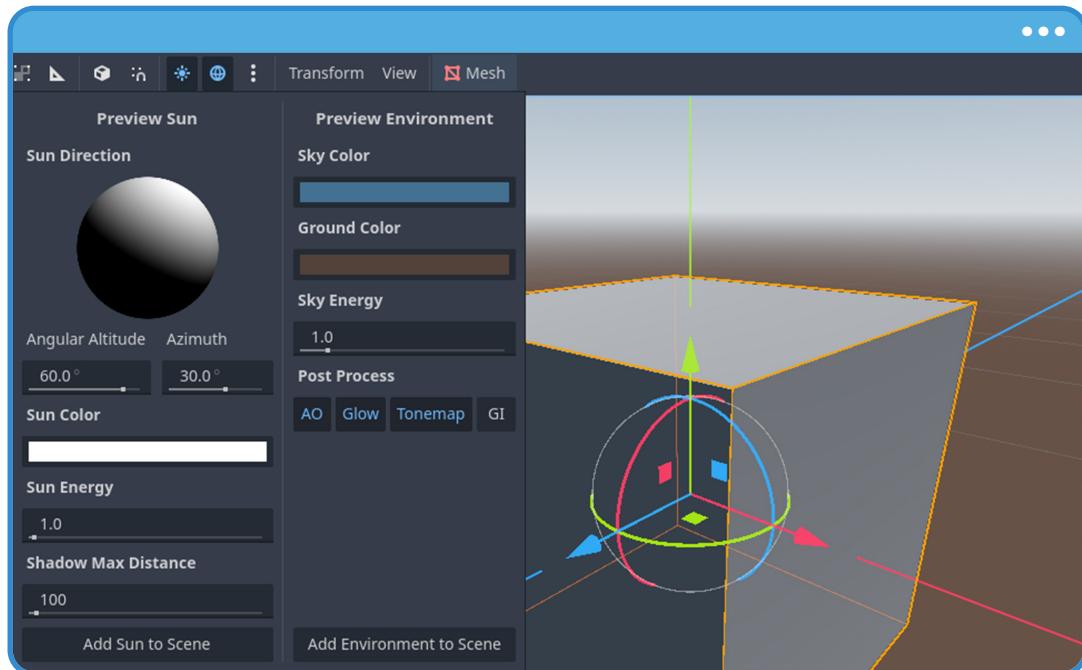


(2.11.g Albedo and anisotropic specularity working together)

## 2.12 Hemispheric Shading.

When we talk about hemispheric shading, we are referring to ambient light – light that simulates the direct and indirect bounces of rays across the scene in a global way. This type of illumination is essential because it helps integrate objects naturally into their environment, preventing them from looking flat or disconnected from the background.

In Godot, there is already a model that performs this operation called **World Environment**. Through it, you can define how much global illumination influences your characters and objects. Inside the Viewport, you can even toggle this influence on or off using the **Toggle Preview Environment** button. This feature allows you to quickly preview how the scene looks with or without ambient lighting.



(2.12.a World Environment)

So, why would you develop your own ambient lighting model if Godot already provides a global illumination system through **World Environment**? The main reason is control — when you are aiming for a stylized render, the default tools may not be enough.

The built-in features that game engines provide are excellent and well-optimized for most cases. However, if you want to achieve a specific visual style — something unique, like the look of DOGWALK — you need full control over how your render behaves. The more lighting techniques you master, the greater your chances of reaching the artistic style you want for your project.

The logic behind hemispheric shading is simple and can be broken down into three main steps:

- 1 Take the Y component of the normals in world space.
- 2 Remap its value from the range  $[-1 : 1]$  to  $[0 : 1]$ .
- 3 Use the remapped value as a gradient to perform a linear interpolation between two colors: one for the ground and one for the sky.

With this method, every surface of a model receives a tone based on the orientation of its normal. Surfaces facing upward blend toward the sky color, while those facing downward shift toward the ground color.

Why is the remapping step necessary? Normals are unit vectors that indicate the perpendicular direction of a surface. When you analyze their Y component in world space:

- A normal pointing upward has the value  $(0, 1, 0)$ .
- A normal pointing downward has the value  $(0, -1, 0)$ .

This means the Y component always varies between  $-1$  and  $1$ . However, the gradient used for linear color interpolation expects values that start at  $0$  and end at  $1$ .

$$G = n_y * 0.5 + 0.5$$

(2.12.b)

With this adjustment, a value of  $-1$  is remapped to  $0$ , while a value of  $1$  stays as  $1$ . This makes it possible to use the result as the interpolation factor between the ground color and the sky color.

To see this in practice, you'll start by organizing your project in the same way you did in previous exercises:

- Inside the **chapter\_02** folder, create a new folder called **hemisphere**.
- Within this folder, set up the following subfolders:
  - **materials**
  - **shaders**
  - **meshes**
  - **textures**

Next, you'll create the material to apply to the character. Go to the **materials** folder, right-click, and select:

- Create New > Resources > ShaderMaterial.

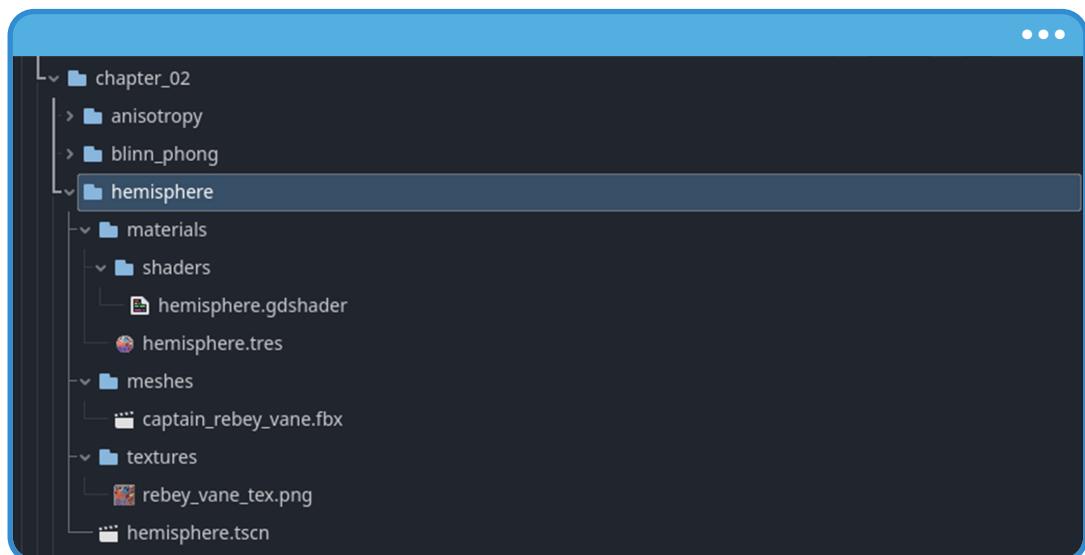
Name this material **hemisphere**. After that, create the corresponding shader. Inside the **shaders** folder, right-click and choose:

- Create New > Resources > Shader.

Also name this file **hemisphere** to keep everything consistent with the section.

In this part of the exercise, you'll use our iconic character, **Rebey Vane**, along with his textures and materials. All these resources are included in the downloadable package that comes with this book, so you can follow the steps without any issues.

If everything has been set up correctly, the folder structure of your project should now look like this:



(2.12.c The hemisphere folder has been added to the project)

The first step is to create a new 3D scene and drag the character file **captain\_rebey\_vane.fbx** into it. Back in section 1.4 of Chapter 1, you learned how to define a unique instance of an .fbx

model, but this time you'll take a different approach: you'll mark the character as Local in the scene. To do this:

- Go to the **Scene** panel.
- Select the node **captain\_rebey\_vane**.
- Right-click on it.
- Choose **Make Local**.

By making the character local, Godot gives you direct access to the **MeshInstance3D** of the model. This allows you to assign the previously created hemisphere material to the character through the **Material Override** property.

Once that's done, open the hemisphere shader. In this section, you'll be developing your own ambient lighting model. Start by declaring a few properties that you'll use later. You'll also disable Godot's default ambient light influence by enabling the **ambient\_light\_disabled** render mode, as shown below:



```

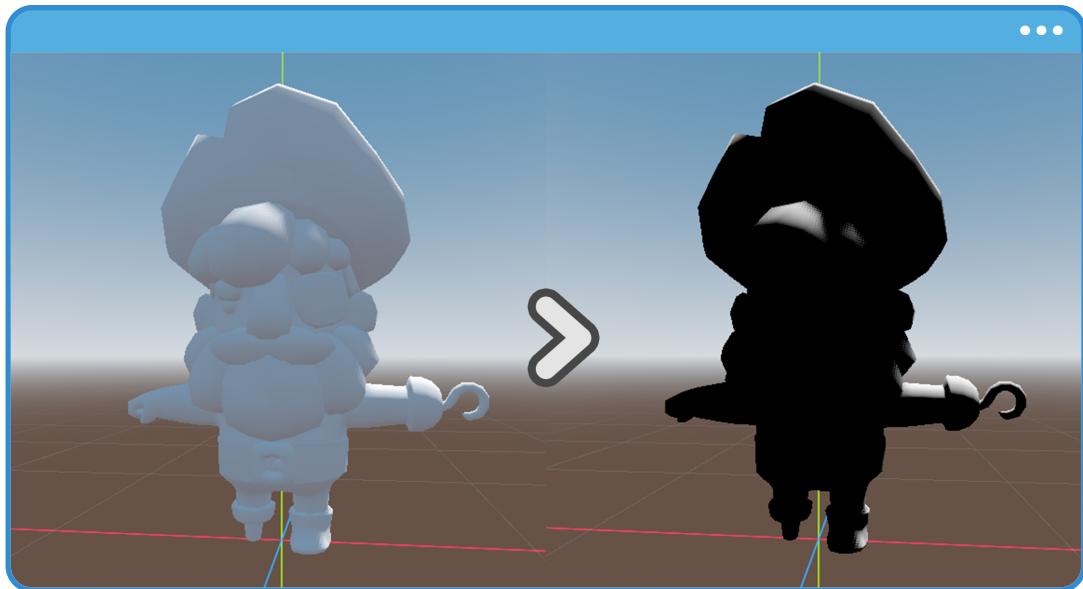
shader_type spatial;
render_mode ambient_light_disabled;

uniform sampler2D _MainTex : source_color;
uniform vec3 _GroundColor : source_color;
uniform vec3 _SkyColor : source_color;
uniform float _Smoothness: hint_range(0.0, 0.5, 0.01);

void vertex() { ... }

```

After saving the shader, you'll notice two main things: first, the new properties are now accessible from the material in the **Inspector**, and second, your character appears completely black, since it no longer receives any influence from the default ambient light.



(2.12.d On the right, ambient light has been disabled)

Now it's time to implement the texture. First, make sure you assign **rebey\_vane\_tex.png** to its corresponding property in the **Inspector**. Next, declare a three-dimensional vector in the **fragment** function and use it as the value for **ALBEDO**, like this:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}
```

With this operation, your character will display the colors defined in the texture, correctly mapped to each part of the model. However, the character still won't receive any ambient lighting, which makes the result appear flat.

To partially solve this, enable the **Light()** function and add a constant value of 0.5 to **DIFFUSE\_LIGHT**:

```
void light()
{
    DIFFUSE_LIGHT += vec3(0.5);
}
```

The value 0.5 was chosen because it works as a neutral midpoint: it doesn't make the model too bright or too dark, making it easier to clearly visualize the albedo colors at this stage.



(2.12.e)

Instead of using a constant value for diffuse lighting, you'll now replace it with the operation described below. First, include the function responsible for calculating hemispheric shading:

```
float hemisphere(vec3 n, float s)
{
    float t = n.y * 0.5 + 0.5; // remap [-1, 1] -> [0, 1]
    return smoothstep(0.0 + s, 1.0 - s, t);
}

void light() { ... }
```

In the first line of `hemisphere()`, you remap the Y component of the normal. As you already know, this process transforms the original range of -1 to 1 into a normalized range of 0 to 1. In the second line, the result is smoothed using the `smoothstep()` function, which lets you create a more stylized effect and control the transition between the ground and sky colors.

Since the first argument of the function corresponds to the model's normal in world space, and the second controls the level of smoothness, you can implement the calculation inside the `light()` function as follows:

```
void light()
{
    vec3 normal_ws = (INV_VIEW_MATRIX * vec4(NORMAL, 0.0)).xyz;

    float gradient = hemisphere(normal_ws, _Smoothness);
    vec3 custom_ambient = mix(_GroundColor, _SkyColor, gradient);

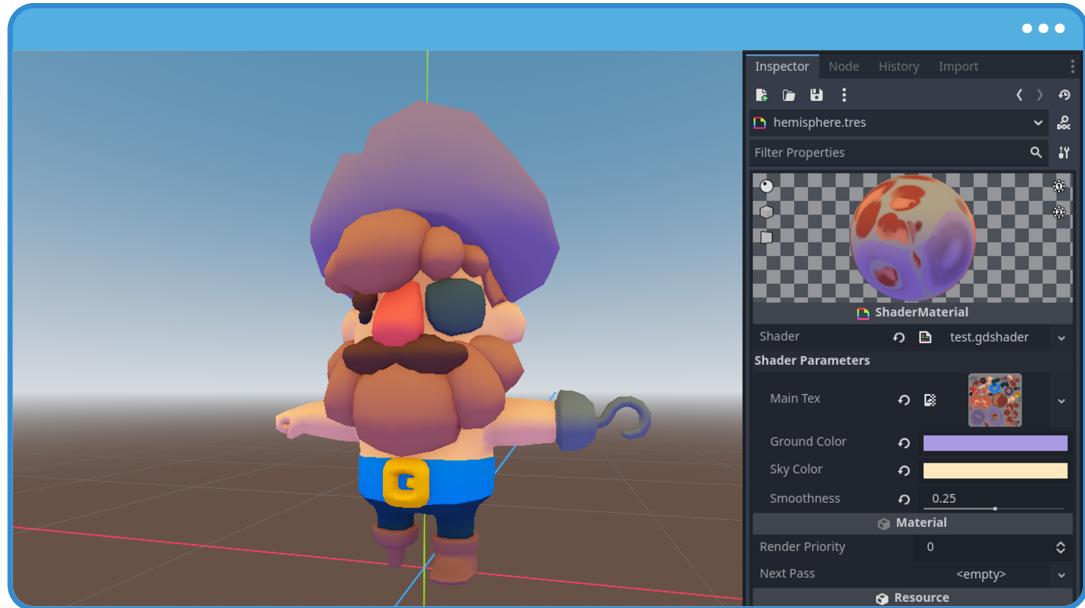
    DIFFUSE_LIGHT += custom_ambient;
}
```

In the first line of `light()`, you transform the normal from view space to world space, because by default normals in this stage are expressed in view space. The result is stored in the vector `normal_ws`, which you then use as input for the gradient calculation.

#### Note

From an optimization perspective, it's more efficient to perform the normal transformation inside the `fragment()` function rather than in `light()`. This way, you avoid repeating the calculation for every light in the scene.

The value returned from `hemisphere()` is stored in the scalar `gradient`. This gradient becomes the interpolation factor for the `mix()` function, which blends the ground and sky colors. The blended result is stored in the `custom_ambient` vector, which is then added to `DIFFUSE_LIGHT` to apply your custom hemispheric shading.



(2.12.f Two ambient colors applied to the pirate)

You could consider the exercise complete at this point, since hemispheric shading has now been implemented successfully. However, in the next section you'll work through a simple exercise that introduces **ShaderInclude**. This feature will help you organize your code more effectively while also improving the final result of the effect.

## 2.13 ShaderInclude Implementation.

Let's suppose you want to transform the custom ambient color from linear space to sRGB. This process is simple – you were already introduced to the concept in Section 2.7. But here comes an important question: what's the best way to implement this transformation in your shader? Should you copy and paste the method again, or should you reuse it? The correct answer is to reuse it, and for that you'll need to understand the concept of **ShaderInclude**.

According to Godot's official documentation:

“

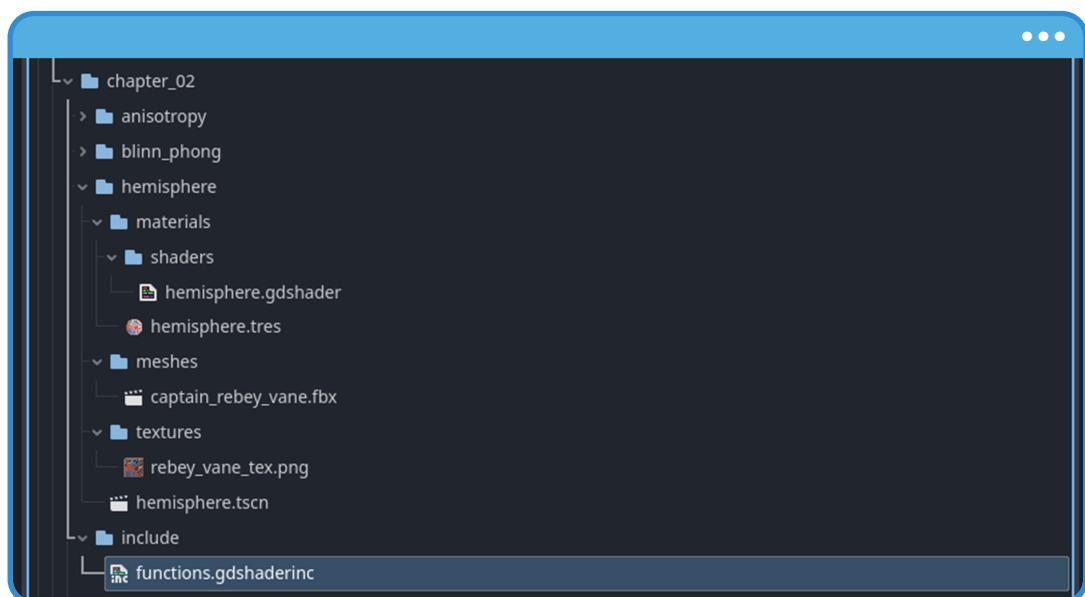
A shader include file, saved with the `.gdshaderinc` extension. This class allows you to define a custom shader snippet that can be included in a shader by using the preprocessor directive `#include`, followed by the file path.

”

To see this in practice, follow these steps:

- Inside your project folder **chapter\_02**, create a new folder called **include**.
- Right-click on this folder and select: Create New > Resource > ShaderInclude.
- Name this file **functions.gdshaderinc**.

If you've followed the steps correctly, the structure of your project should now look like this:



(2.13.a A ShaderInclude has been added to the project)

Next, go back to the **blinn\_phong** shader, where you previously defined the `to_sRGB()` method. Copy the entire method and paste it into the **functions** file, as shown below:

```

File Search Edit Go To Help
functions.gdshaderinc
1  vec3 to_sRGB(vec3 linearRGB)
2  {
3      vec3 a = vec3(1.055) * pow(linearRGB.rgb, vec3(1.0/2.4)) - vec3(0.055);
4      vec3 b = linearRGB.rgb * vec3(12.92);
5  }> bvec3 c = lessThan(linearRGB, vec3(0.0031308));
6      return vec3(mix(a, b, c));
7 }

```

(2.13.b The `to_sRGB()` method has been added to the functions)

Now, comment out the `to_sRGB()` method in the `bling_phong` shader. When you do this, the shader will fail to compile — this is expected, since the function has been removed. To fix the issue, simply include your **ShaderInclude** like this:

```

File Search Edit Go To Help
functions.gdshaderinc
blinn_phong.gdshader
1  shader_type spatial;
2
3  #include "res://assets/chapter_02/include/functions.gdshaderinc"
4
5  uniform sampler2D _MainTex : source_color;
6
7  void vertex()
8  {
9  }> // Called for every vertex the material is visible on.
10 }
11

```

(2.13.c The ShaderInclude path has been added)

As you can see in line 3 of the code, the **ShaderInclude** file is referenced inside the `bling_phong` shader. From that point on, you can reuse the `to_sRGB()` method through the preprocessor directive `#include`.

This approach is especially useful because if you need to reuse additional functions later, you only need to declare them once in `functions.gdshaderinc` and then include the file in any shader that requires them. This workflow not only saves time but also centralizes future modifications in a single shared file.

Returning to your **hemisphere** shader, add the include path as follows:

```
shader_type spatial;
render_mode ambient_light_disabled;

#include "res://assets/chapter_02/include/functions.gdshaderinc"

uniform sampler2D _MainTex : source_color;
```

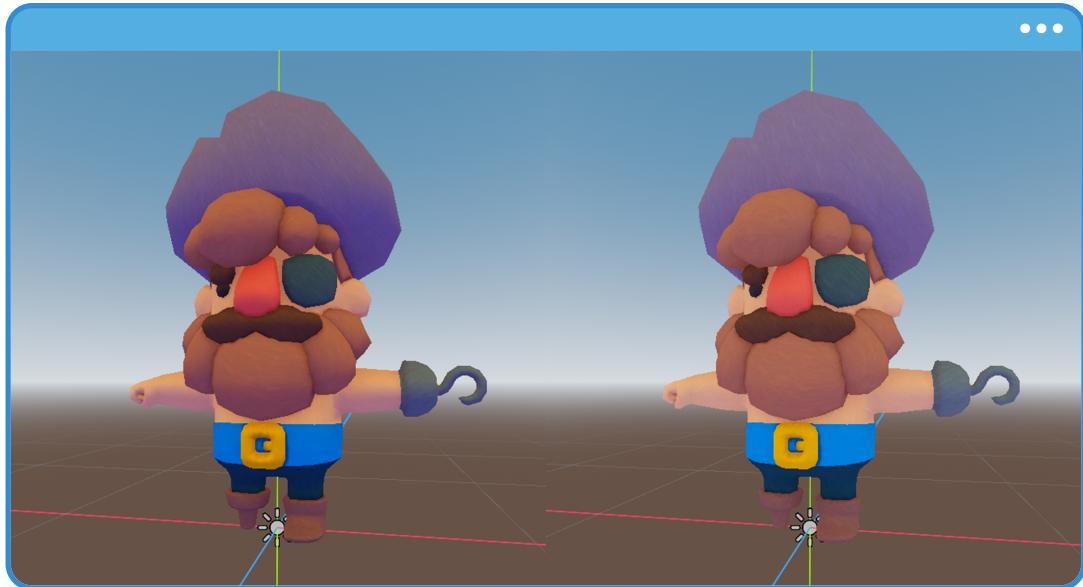
Next, update the **light()** method so that **custom\_ambient** is converted from linear space to sRGB:

```
void light()
{
    vec3 normal_ws = (INV_VIEW_MATRIX * vec4(NORMAL, 0.0)).xyz;

    float gradient = hemisphere(normal_ws, _Smoothness);
    vec3 custom_ambient = mix(_GroundColor, _SkyColor, gradient);
    custom_ambient = to_sRGB(custom_ambient);

    DIFFUSE_LIGHT += custom_ambient;
}
```

By applying this change, the visual result gets closer to the stylized rendering you're aiming for, while keeping the code clean, reusable, and easy to maintain.



(2.13.d To the left, linear; to the right, sRGB)

To finish, you'll implement the **Lambertian** model in the shader to emphasize the contrast between light and shadow.

```
void light()
{
    float NdotL = max(dot(NORMAL, LIGHT), 0.0);
    vec3 lambert = NdotL * ATTENUATION * LIGHT_COLOR;

    DIFFUSE_LIGHT += lambert;

    vec3 normal_ws = (INV_VIEW_MATRIX * vec4(NORMAL, 0.0)).xyz;

    float gradient = hemisphere(normal_ws, _Smoothness);
    vec3 custom_ambient = mix(_GroundColor, _SkyColor, gradient);
    custom_ambient = to_sRGB(custom_ambient);

    DIFFUSE_LIGHT += custom_ambient;
}
```

In this case, as you can see above, you've duplicated the model explained in Section 2.2. Now it's your turn: do the exercise by moving the Lambert calculation into your **ShaderInclude** and calling it from **light()**. Make sure the shader compiles and that the render matches the math exactly.



(2.13.e Stylized rendering with light and shadows)

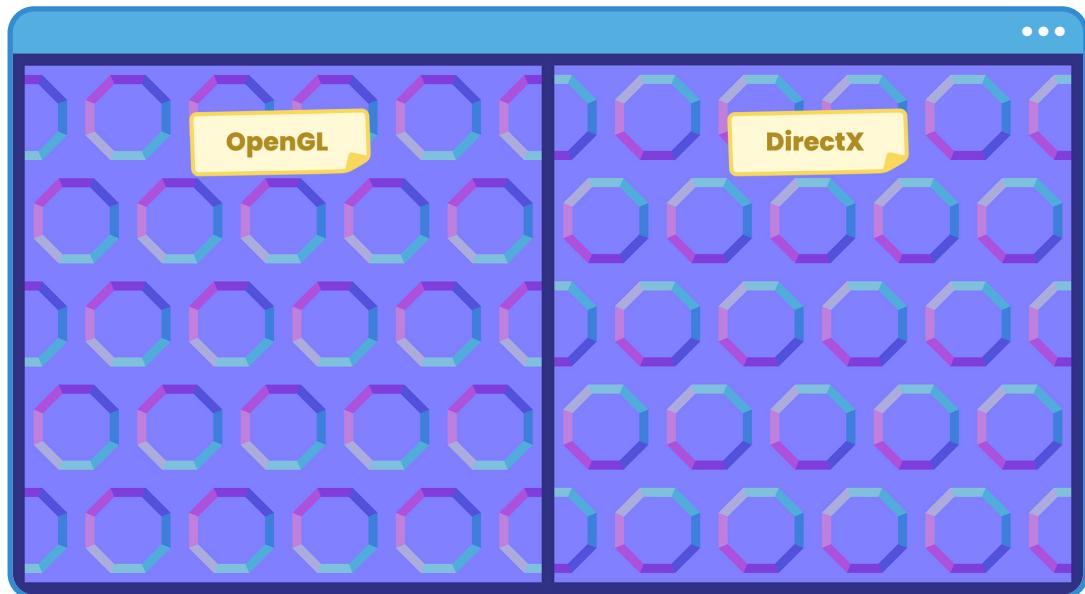
## 2.14 Normal Map Implementation.

In Godot, when you create a **StandardMaterial3D**, you'll find an option called **Normal Map** among its properties. It's disabled by default. Once you enable it, you gain access to two main parameters:

- **Scale:** controls the intensity of the effect.
- **Texture:** the normal map texture itself.

Using a normal map is a widely adopted technique for adding surface detail without increasing vertex count. In combination with lighting, a normal map simulates bumps and fine variations on the surface, producing a richer, more convincing result with minimal performance cost.

Here's an example of what a normal map looks like:



(2.14.a Normal map in OpenGL and DirectX)

A normal map encodes directions as colors:

- X channel: red when pointing in the positive direction, cyan when negative.
- Y channel: green when pointing in the positive direction, magenta (purple) when negative.
- Z channel: represents the component pointing outward, perpendicular to the surface

If you look at Figure 2.14.a, you'll notice that the Y component differs between OpenGL and DirectX normal maps. This comes from differences in each API's tangent-space conventions. Despite this difference, the final shaded result is equivalent once the normal is interpreted correctly.

To implement normal maps in a shader, you'll consider three fundamentals:

- Remap the coordinates.
- Reconstruct the Z component.
- Apply the TBN matrix (Tangent, Bitangent, and Normal).

Back in Section 1.8 of Chapter 1, you explored tangents, normals, and bitangents when creating an anime-style eye depth effect. You'll now take that a step further and apply a full TBN matrix to transform normals from the normal map into view space, adding detail to your object.

It's true that in the fragment stage you can use the built-in **NORMAL\_MAP** variable, which automatically derives the normal from a normal map, together with **NORMAL\_MAP\_DEPTH** to control intensity. However, to fully understand what's happening under the hood, you'll walk through a manual implementation step by step.

Do the following:

- 1 In your project, under **chapter\_02**, create a new folder named **normal\_map**.
- 2 Inside it, add these subfolders:
  - a **materials**
  - b **shaders**
  - c **meshes**
  - d **textures**

With that structure in place, include the downloadable assets provided for this section.

To demonstrate why normal maps are useful, you'll build a shader that implements each step required to apply them. You'll examine each function individually to understand the math, and you'll intentionally use both **NORMAL** and **NORMAL\_MAP** at different stages so you can appreciate their technical differences.

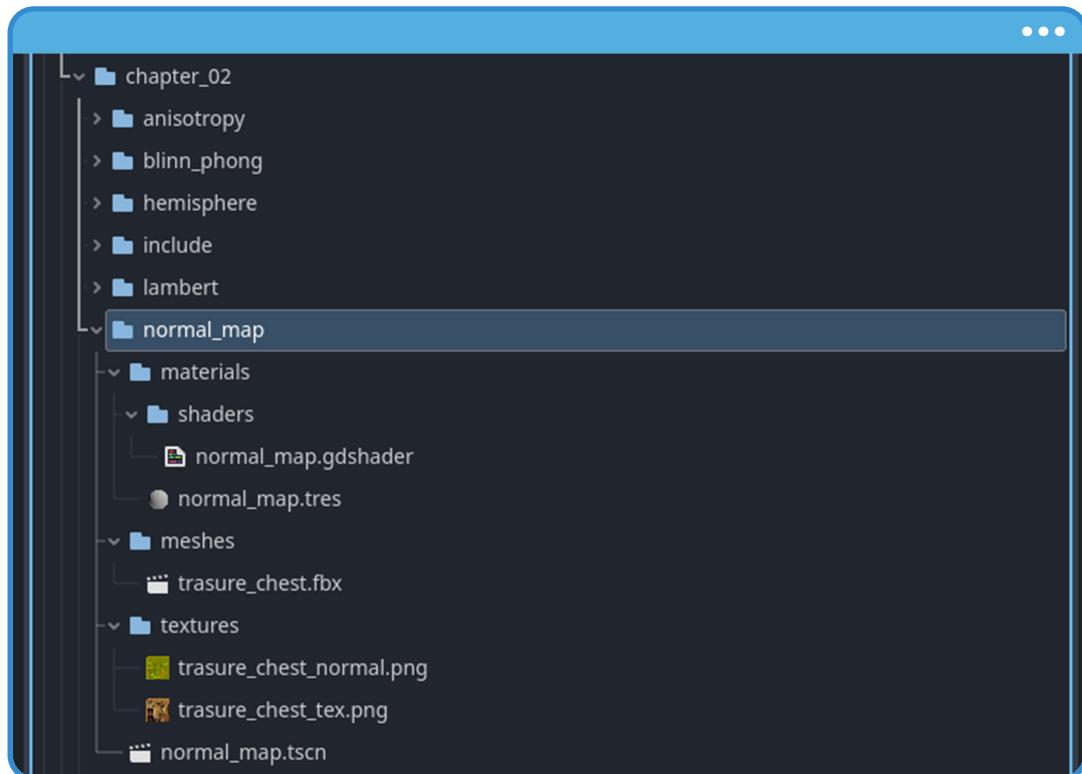
Next, create a material. In the **materials** folder, right-click and choose:

- Create New > Resource > ShaderMaterial.

Name it **normal\_map**. Then create a shader in the **shaders** folder, reusing the same name for consistency:

- Create New > Resource > Shader.

If you've followed the steps correctly, your project should now look like this.



(2.14.b A new material and shader have been added to the project)

Before writing the shader, create a new 3D scene and drag **treasure\_chest.fbx** into it. Since it's an .fbx file, make sure to mark it Make Local in the Scene panel so you can access the model's **MeshInstance3D**. Add a DirectionalLight3D as well — you'll need it to clearly see how the normal map affects the chest's surface details.

Next, assign the **normal\_map** shader to its corresponding material, then apply that material to the object in the **Viewport**. This will let you observe the changes as you implement each step.

The first step in your shader is to declare the properties you'll use throughout:

- An albedo map called **\_MainTex**.
- A normal map called **\_NormalMap**.
- A float to control intensity called **\_Scale**.

Your initial code should look like this:

```
shader_type spatial;

uniform sampler2D _MainTex : source_color;
uniform sampler2D _NormalMap : hint_normal;
uniform float _Scale : hint_range(-16.0, 16.0, 0.1);

void vertex() { ... }
```

If you look closely at the previous code, you'll notice that `_NormalMap` has the attribute `hint_normal`. This hint is important because it ensures that when you assign a texture in the **Inspector**, Godot reimports it as a normal map. For that reason, it's essential to include it in the shader.

As for the `_Scale` variable, we've defined a range from -16.0 to 16.0. Practically speaking, a range between 0.0 and 1.0 would be more than enough. However, in this case the wider range is used to stay consistent with the default value that **StandardMaterial3D** provides for normal maps.

The next step is to initialize and implement the `ALBEDO` variable so the chest's texture displays correctly in the scene. To see the proper result, make sure to assign both `treasure_chest_tex` and `treasure_chest_normal` to their corresponding properties.

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    ALBEDO = albedo;
}
```

At this stage, the chest is already affected by directional and ambient lighting. However, you'll adjust the lighting configuration to make the relief from the normal map stand out more clearly. To do this, add the `diffuse_toon` and `specular_toon` render modes:

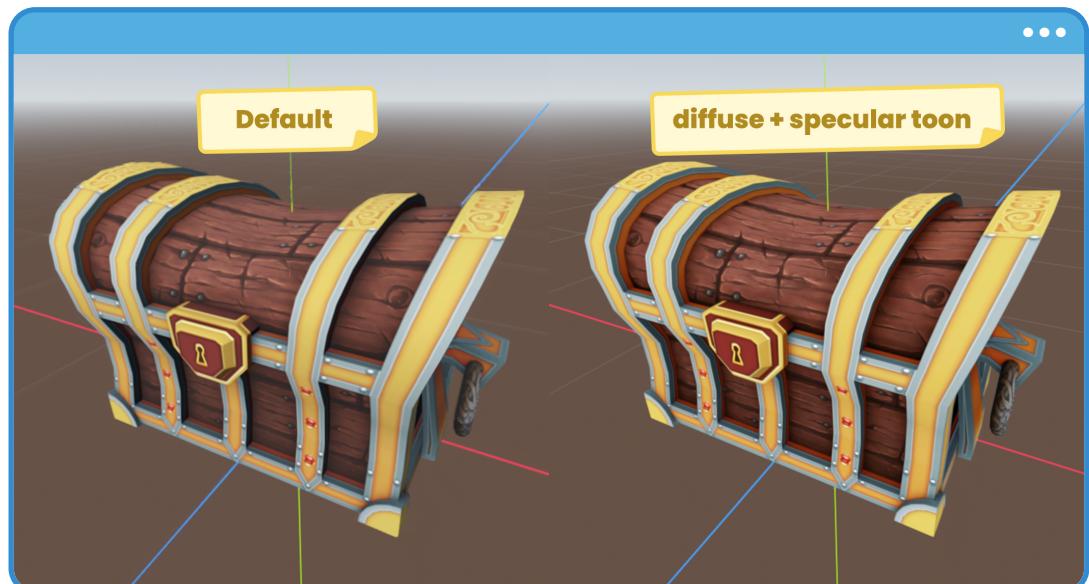
```

shader_type spatial;
render_mode diffuse_toon;
render_mode specular_toon;

uniform sampler2D _MainTex : source_color;
uniform sampler2D _NormalMap : hint_normal;
uniform float _Scale : hint_range(-16.0, 16.0, 0.1);

```

These render modes give you a stylized result where shading and highlights are simplified, making the relief created by the normal map much easier to perceive.



(2.14.c Contrast has been improved on the chest)

To start implementing normal mapping, the first step is to sample the texture and store the result in a three-dimensional vector. Unlike UV coordinates, normal maps work in a range from -1.0 to 1.0. This means your next step is to remap the texture values into this range. You can do this with a simple operation:

```

void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;

    vec3 normal_map = texture(_NormalMap, UV).rgb;
    normal_map = normal_map * 2.0 - 1.0;

    ALBEDO = albedo;
}

```

**Note**

If the Y channel of your normal map is inverted because of convention differences (OpenGL vs. DirectX), you can correct it with: `normal_map.y *= -1.0;` before reconstruction. Alternatively, you can fix this directly in the **Import** panel: select the normal map in your project and enable **Normal Map Invert Y**. This ensures that lighting reacts in the expected direction.

Why does the operation `normal_map * 2.0 - 1.0` perform the remapping we need? The explanation is straightforward:

If you take the minimum value of the original range (0.0):

$$0.0 * 2.0 - 1.0 = -1.0$$

(2.14.d)

If you take the maximum value (1.0):

$$1.0 * 2.0 - 1.0 = 1.0$$

(2.14.e)

This way, the range  $[0.0 : 1.0]$  used by a texture is correctly transformed into the range  $[-1.0 : 1.0]$ , which is required for working with normals.

The next step is to reconstruct the Z component of the normal map. This is necessary because the texture stores only the X (red channel) and Y (green channel) components, while the Z component — representing the direction perpendicular to the surface — is usually omitted to save memory.

Keep in mind that normal maps are expressed in tangent space, where normals generally point toward the positive Z axis. Tangent space is defined by three orthogonal vectors: the normal, tangent, and bitangent. This basis lets you transform the sampled normal into a common lighting space — view space in this case — ensuring consistent and accurate interaction with light.

You can reconstruct the Z component with the following equation:

$$z = \sqrt{1.0 - (x^2 + y^2)}$$

(2.14.f)

This formula comes directly from the Pythagorean theorem in 3D:

$$x^2 + y^2 + z^2 = 1$$

(2.14.g)

Since normals must be unit vectors, and the map only gives you X and Y, reconstructing Z ensures the vector's length remains 1. In code, it's best to protect against numerical errors using `clamp` (or `max`) before applying the square root:

```
void fragment()
{
    ...

    float n_z = clamp(dot(normal_map.xy, normal_map.xy), 0.0, 1.0);
    normal_map.z = sqrt(1.0 - n_z);

    ALBEDO = albedo;
}
```

Next, transform the normal from tangent space to view space using the TBN matrix (Tangent, Bitangent, Normal). In Godot, you already have access to **TANGENT**, **BINORMAL**, and **NORMAL**. When constructing the **mat3**, each vector is used as a column of the matrix:

```
void fragment()
{
    ...

    mat3 TBN = mat3(TANGENT.xyz, BINORMAL.xyz, NORMAL.xyz);
    vec3 normal_vs = normalize(TBN * normal_map);

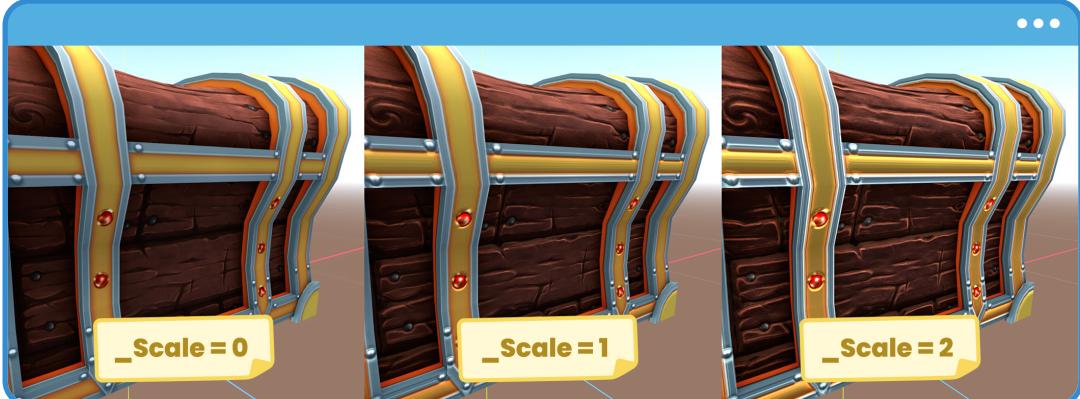
    ALBEDO = albedo;
    NORMAL = normal_vs;
}
```

As soon as you assign the value of **normal\_vs** to the built-in **NORMAL** variable, you'll notice the surface details of the chest become more pronounced in the **Viewport**. If you want to adjust the strength of this effect, you can multiply the **normal\_map** vector by **\_Scale**, like so:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;

    vec3 normal_map = texture(_NormalMap, UV).rgb;
    normal_map = normal_map * 2.0 - 1.0;
    normal_map *= _Scale;

    ...
}
```

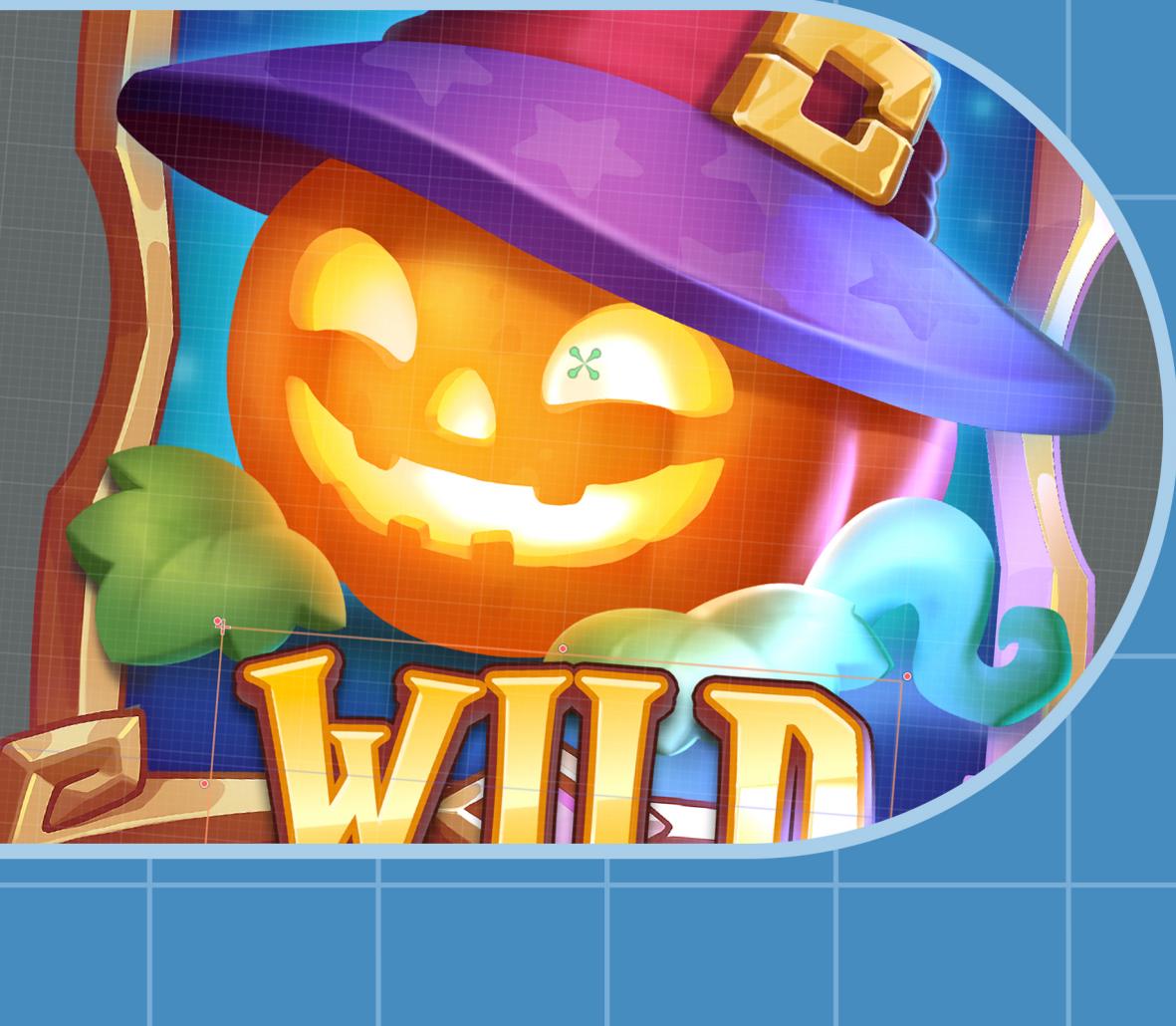


(2.14.h Different strength for the normal map)

In Godot, you don't need to manually perform every step to calculate a normal map. The engine provides a built-in variable called **NORMAL\_MAP**, which handles the process automatically. With it, you can achieve the same result using a much simpler implementation:

```
void fragment()
{
    vec3 albedo = texture(_MainTex, UV).rgb;
    vec3 normal_map = texture(_NormalMap, UV).rgb;

    NORMAL_MAP = normal_map;
    NORMAL_MAP_DEPTH = 1.0;
    ALBEDO = albedo;
}
```



# Chapter 3

## Procedural Shapes and Vertex Animation.

In this chapter, you'll dive into several essential topics that will elevate your shader skills to a new level. You'll begin by exploring how to draw two-dimensional procedural shapes using simple mathematical functions, inequalities, and UV coordinates. Because you're working within a Cartesian space, you can define areas, outlines, and transitions directly on the texture surface — laying the foundation for constructing complex forms without relying on external assets.

You'll then move on to procedural animation and deformation, learning how to transform these shapes into dynamic elements through time-based operations, interpolations, and parameter-driven adjustments. These concepts will allow you to develop a fully procedural character implemented in GDScript, where every visual component is generated mathematically and responds to configurable variables.

In addition, you'll study vertex-deformation techniques tailored for user interface elements, enabling you to animate UI components directly from the shader. Along the same lines, you'll learn how to integrate a specular highlight effect for UI surfaces — useful for adding glossy finishes, enhancing details, or giving any panel or figure a more stylized look.

In the second part of the chapter, you'll dive into rotations. You'll review their geometric interpretation and learn how to implement custom rotation matrices — an essential tool for manipulating points in space and animating geometry with precision. From there, you'll study quaternions, understanding their structure and the advantages they offer over traditional matrices by avoiding issues such as gimbal lock. You'll implement these tools in GDScript to perform stable, efficient rotations directly inside your shader.

Each topic will be explored through practical examples that show you, step by step, how these techniques integrate into both artistic and technical workflows. The ultimate goal is to expand your toolkit, giving you the mathematical and shader-based techniques needed to solve complex problems with clarity and control.

### 3.1 Drawing with Functions and Inequalities.

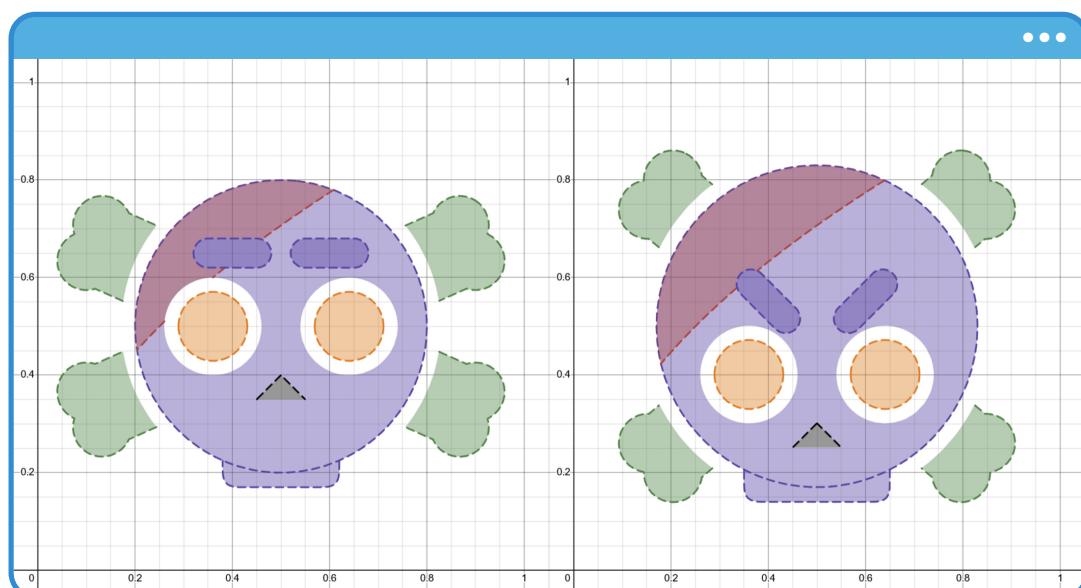
When we talk about functions and inequalities, we are referring to mathematical expressions that can be applied directly to Cartesian coordinates. Some of the most common include:

- Linear functions.

- Quadratic functions.
- Trigonometric functions.
- Absolute value functions.
- Exponential functions.
- And more.

Each of these functions has unique properties that, when combined, allow you to construct complex shapes procedurally. While a deep analysis of their behavior and a detailed breakdown of their implementation in Godot's shader language could easily fill an entire book, in this chapter we will take a more hands-on approach. You will focus on translating these functions into the shader context while designing a procedural figure step by step.

As a concrete example, you will build a pirate-style skull using only mathematical functions and conditional statements, as shown in the following visual reference:



(3.1.a Pirate skull on Desmos)

**Note**

Some time ago, I wrote a book titled *Shaders and Procedural Shapes in Unity 6*, which explores this topic in depth. While the focus was on Unity, the functions and concepts described there are fully transferable to Godot. If you are interested in learning more about that project, you can check it out at the following link: <https://jettelly.com/store/visualizing-equations-vol-2>

If you look closely at Figure 3.1.a, you will notice that the skull is built almost entirely from mathematical equations applied to two-dimensional coordinates. These equations – defined later in this chapter – allow you to delimit visible regions on the screen through comparisons and operations on the  $x$  and  $y$  coordinates. This approach greatly simplifies procedural design, as each part of the figure is tied to a precise mathematical condition.

Now, where should you start if you wanted to create this figure from scratch? While there is no single “correct” entry point, you can follow a logical workflow that will help you better understand UV coordinates and, over time, master two-dimensional design in shaders.

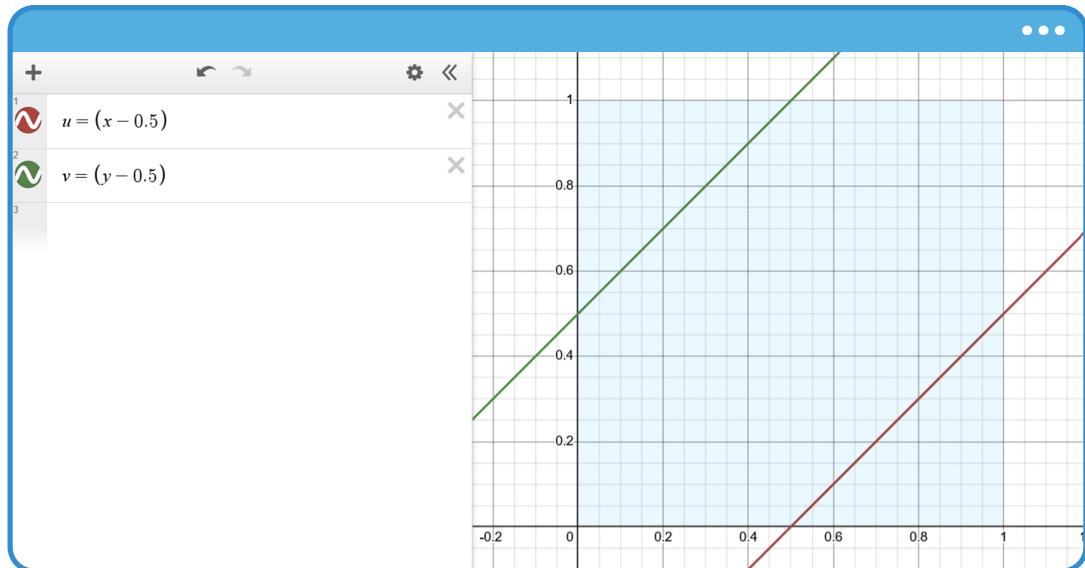
As a first step, you will explore the shape in Desmos, a powerful graphing tool that lets you experiment with equations and instantly visualize their effects. It is recommended that you have the following page open as you begin your analysis:

➤ <https://www.desmos.com/calculator>

**Note**

This book includes a downloadable project containing a folder named **case\_study**, where you will find more examples of procedural shapes created using Desmos. These additional cases will allow you to further explore shape construction through equations and experiment with different strategies for visual modeling based on coordinates.

Assuming you are already inside the Desmos calculator interface, you will now begin developing the procedural shape step by step. The first step is to declare and initialize the UV coordinates on the Cartesian plane, which will allow you to work with functions applied directly to these variables, as shown below:



(3.1.b <https://www.desmos.com/calculator/i5ad2cvmqh>)

This first step is particularly valuable because it allows you to understand how UV coordinates behave in a normalized space. From the visualization in Desmos, you can draw the following observations:

- 1 The  $u$  coordinate corresponds directly to the  $x$ -axis.
- 2 The  $v$  coordinate corresponds to the  $y$ -axis.
- 3 Both coordinates are constrained between 0.0 and 1.0, which represents the typical UV space in a fragment shader.
- 4 To center any shape on the plane, you must subtract 0.5 from each coordinate, effectively shifting the origin to the center of the domain.

With this foundation in place, you are ready to begin designing the general shape of the skull. As shown in Figure 3.1.a, the head consists of two regions defined by implicit equations:

The first describes the upper part of the skull, represented by a circular equation:

$$u^2 + v^2 < r^2$$

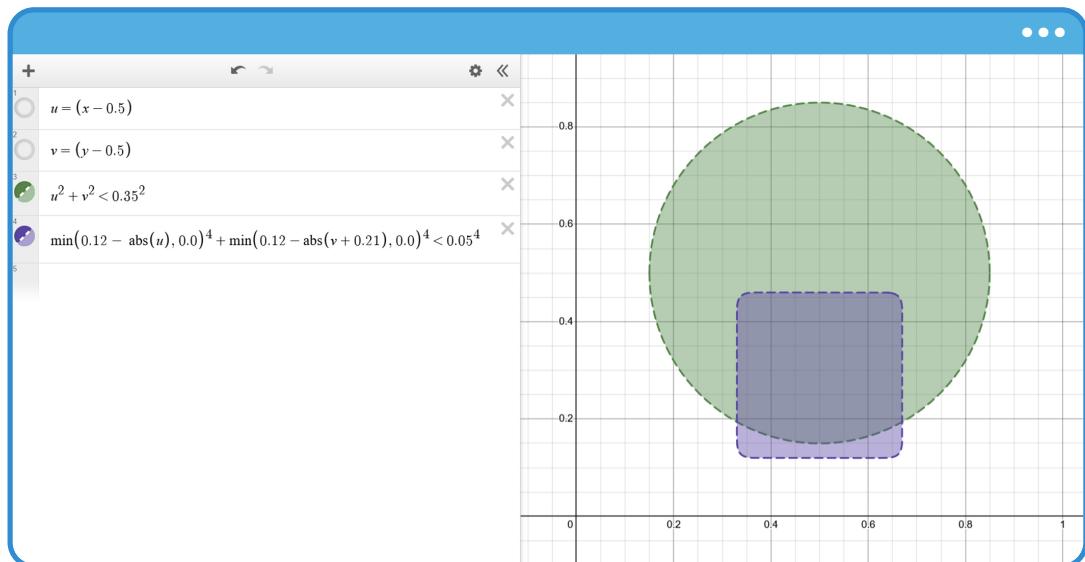
(3.1.c)

The second corresponds to the chin, modeled as a rectangular deformation with smooth edges:

$$\min(n_1 - |u|, k_1)^4 + \min(n_2 - |v|, k_2)^4 < r^4$$

(3.1.d)

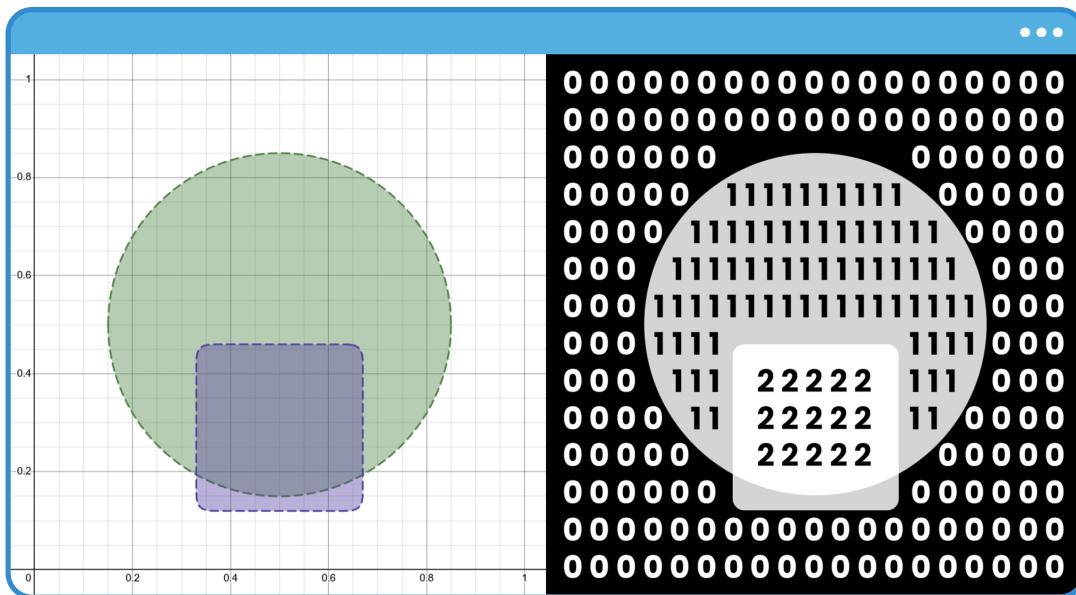
You will start by using constant values to set the position and size of both elements on the plane. This will make it easier to fine-tune them later when combining them into a single cohesive shape.



(3.1.e <https://www.desmos.com/calculator/9kgoilxgz2>)

An important detail visible in Figure 3.1.e is the intersection between the two shapes. This phenomenon is particularly interesting from a computer graphics perspective. Later, when you translate these mathematical operations into GDSL inside your shader, the visible regions of each shape will evaluate to 1.0, while the areas outside those regions will evaluate to 0.0.

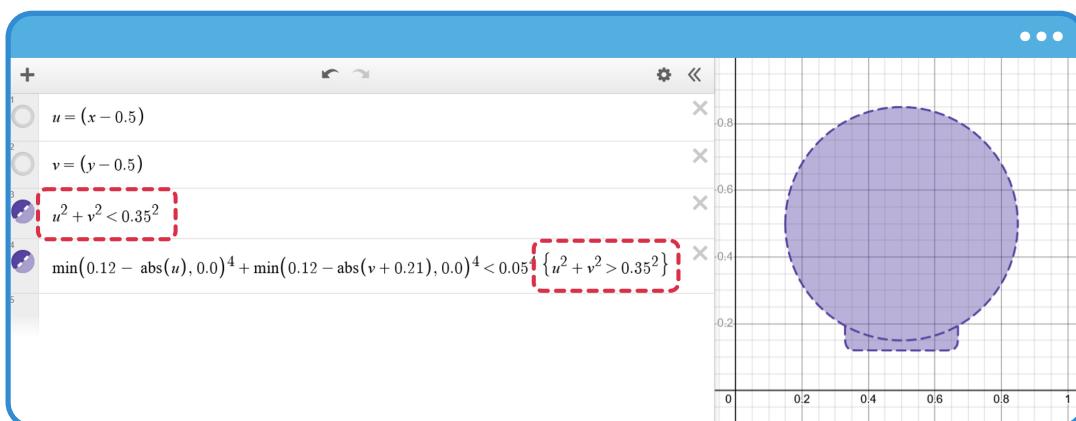
This means that if you combine both shapes through direct addition, the intersection area will result in a value of 2.0. This excess can lead to unwanted visual artifacts, such as color or luminance saturation — especially if these values are being used to control color, opacity, or light intensity.



(3.1.f Contrast values have been adjusted to represent the difference in values)

Therefore, when combining shapes, you should consider techniques such as clamping (`clamp()`), averaging, or even the `max()` operation, depending on the visual behavior you want to achieve. We will explore these methods in more detail later.

For now, you will avoid having the two shapes overlap in Desmos by restricting the rectangle's area so it does not interfere with the circle. To achieve this, you can limit the domain of the function using brackets, ensuring that the equation is only applied within a specific range. In this case, you will constrain the rectangle's function so that it only operates outside the area defined by the circle, as shown below:

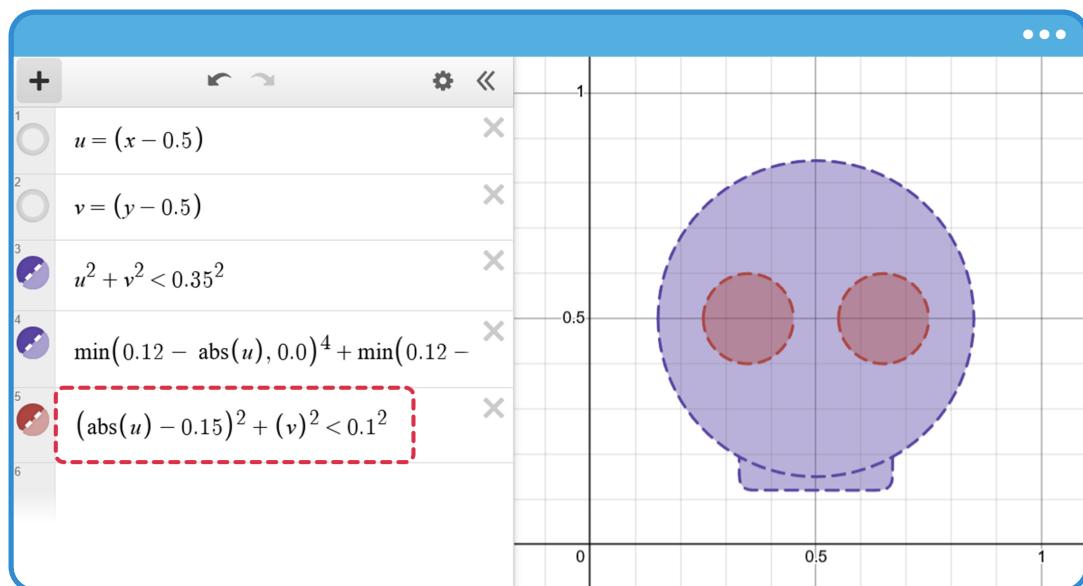
(3.1.g <https://www.desmos.com/calculator/2hytxw2ee6>)

**Note**

Procedural shapes can be demanding on the GPU, so it's recommended to use them with caution — especially in production environments for video games. However, understanding these techniques deepens your control over UV coordinates and equips you with tools to optimize visual performance by combining textures with mathematical functions.

It's worth noting that you can also implement these kinds of conditional limits directly in GDSL. Whether you do so will depend on the desired result, since explicit clipping is not always necessary if the visual effect does not require it.

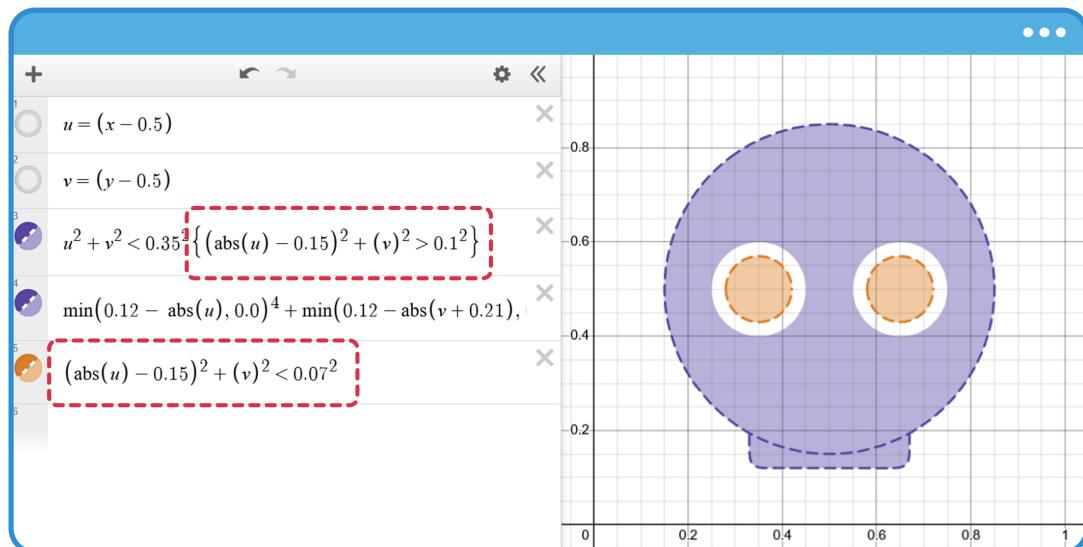
Now, let's move on to the next step in the design: the skull's eyes. In this case, you will also use a circular equation, but there's no need to duplicate it. Thanks to horizontal symmetry, you can simply apply the function  $\text{abs}(u)$  to automatically mirror the shape, giving you both eyes with a single mathematical expression.



(3.1.h <https://www.desmos.com/calculator/6dbg8pf7vw>)

As you can see, the equation you just added corresponds to the same shape defined earlier in Figure 3.1.c. The key difference is that this time, you applied the absolute value to the  $u$  coordinate. This small adjustment takes advantage of the object's horizontal symmetry, allowing you to represent both eyes with a single operation and thereby optimize the process.

Additionally, if you wish, you can use this new region as a clipping condition for the overall head shape. This would allow you to create a precise cutout in the skull area, respecting the construction hierarchy and maintaining control over shape overlaps.



(3.1.i <https://www.desmos.com/calculator/qqv6aqwmjp>)

With this adjustment, you reduce the size of the eyes to achieve the desired visual result. This technique allows you to fine-tune visual details with precision, without the need to define multiple independent shapes.

Next, you can define the nose. One option would be to use a linear function in the form:

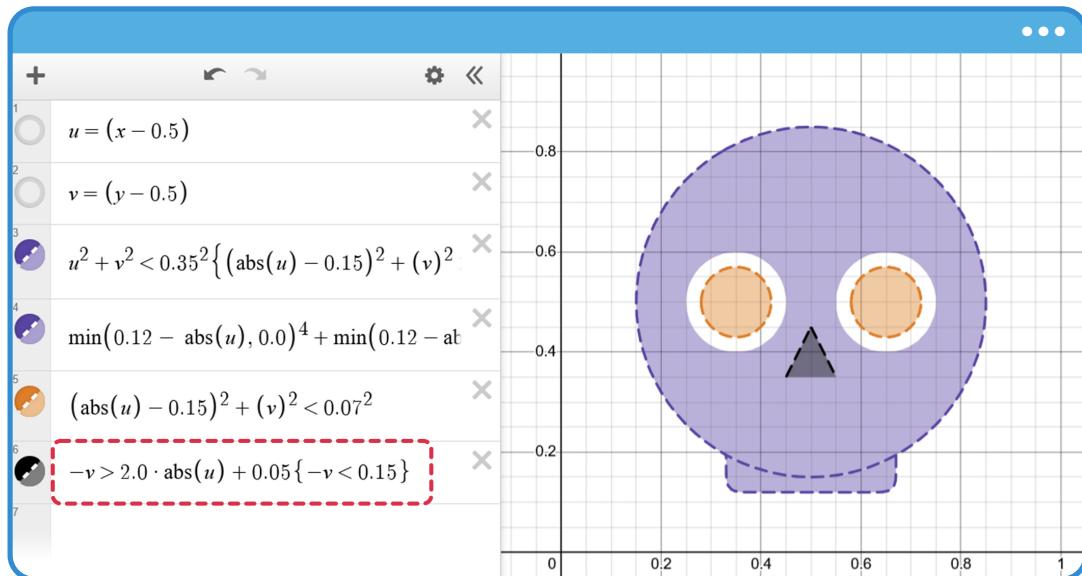
$$v > mu + b$$

(3.1.j)

However, this expression describes only a straight line. To model a triangular nose, you need to define a region bounded by two symmetrical lines that converge at a single point. You can achieve this in Desmos in two steps:

- ① Apply the absolute value to the u coordinate to create horizontal symmetry.
- ② Restrict the expression to a specific vertical range to close the shape.

For example:



(3.1.k <https://www.desmos.com/calculator/tkw4cieh3u>)

This condition generates an inverted triangular region positioned at the lower center of the head – perfect for representing the nose. By constraining the  $v$  range, you prevent the shape from extending beyond what is necessary, keeping it compact and well-controlled.

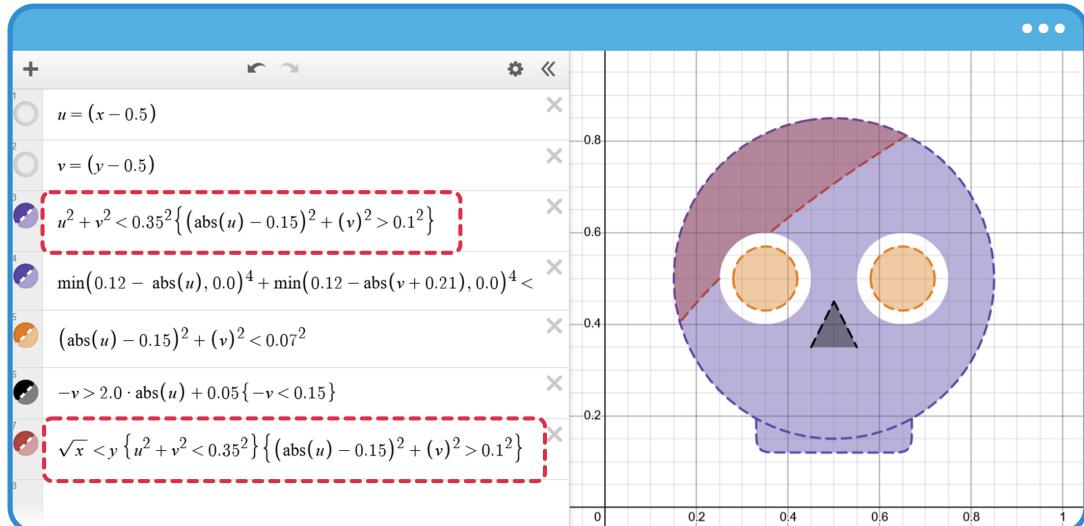
As shown in Figure 3.1.k, the function's parameters take the values  $m = 2.0$  and  $b = 0.05$ . Additionally, the  $v$  coordinate has been inverted to tilt the slope downward, and the absolute value of  $u$  has been applied to create the horizontal symmetry characteristic of a triangular skull nose. By adjusting these values, you can control both the width and the vertical position of the shape.

With the nose defined, you can move on to the pirate bandana. There are multiple ways to approach this element, but a simple and effective option for this character is to use a condition based on a square root. Specifically, you can visualize all points where the square root of  $x$  is less than  $y$ , which can be expressed as:

$$\sqrt{x} < y$$

(3.1.l)

This expression produces an upward curve across the face, simulating part of the fabric's contour. You can then trim this region using the previously defined areas for the head and eyes, ensuring that the bandana does not overlap or interfere with other parts of the design.



(3.1.m <https://www.desmos.com/calculator/nvkem2ifdl>)

As shown in Figure 3.1.m, after applying the square root-based condition, you limited the result to two regions: the head area and the eye region. This ensures that the pirate bandana does not overlap important facial features or extend beyond the skull's edges.

It's important to note that in this case you are using  $xy$  coordinates instead of  $uv$ . This is because Desmos starts  $x$  and  $y$  from 0.0, which makes it ideal for directly representing expressions like  $\sqrt{x} < y$  without requiring recentering.

At this point, only a few elements remain to complete the character: the eyebrows and the crossed bones behind the head. For these, you will introduce a fundamental resource in computer graphics – two-dimensional rotation using matrices.

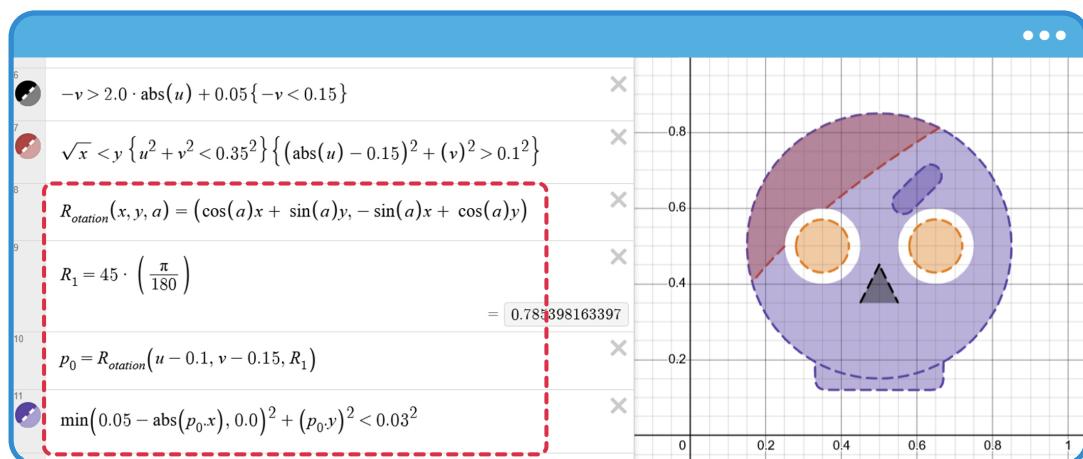
You will create a function in Desmos called `Rotation`, which will take three arguments: the  $x$  and  $y$  coordinates of the point to transform, and an angle  $a$ . The rotation is defined as:

$$R_{\text{rotation}}(x, y, a) = (\cos(a)x + \sin(a)y, -\sin(a)x + \cos(a)y)$$

(3.1.n)

This transformation allows you to rotate any point on the plane, which is particularly useful for giving expression to the eyebrows or adjusting the orientation of the bones.

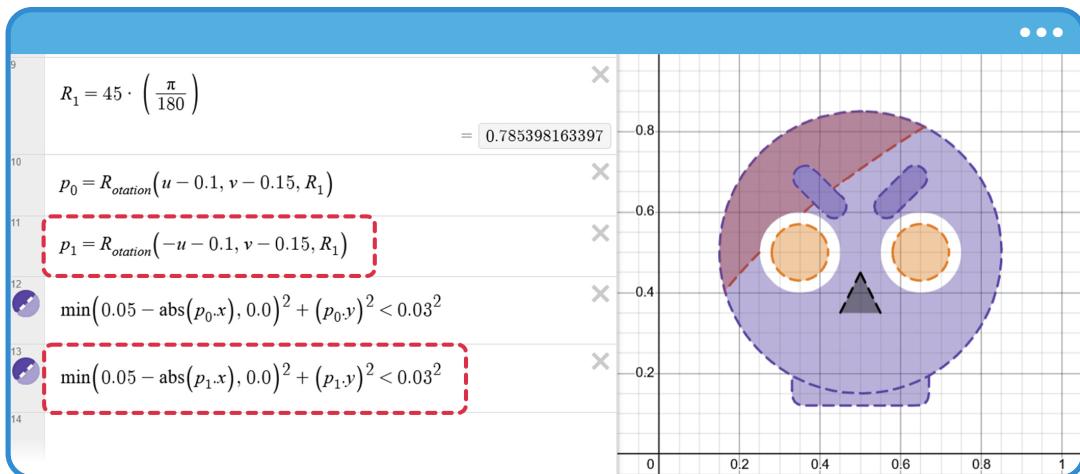
For the eyebrows, you will use soft cylinders, defined with the same equation you saw in Figure 3.1.d, but with lower exponents to create smoother edges. Since you'll be applying rotation, you will first initialize a new point to store the rotated  $uv$  coordinates, and then define a specific rotation angle for each eyebrow. Separating these variables will let you reuse the same base shape in different positions and orientations.



(3.1.o <https://www.desmos.com/calculator/juc94owkuo>)

In Figure 3.1.o, you can see that a constant rotation angle of  $45^\circ$ —represented as  $R_1$ —has been defined. A new point,  $p_0$ , is then declared and initialized by applying the rotation to the previously offset  $uv$  coordinates. This new point replaces the original coordinates, allowing you to precisely determine the position and orientation of the character's right eyebrow.

To create the left eyebrow, you can optimize the process using a horizontal reflection. Instead of rotating a second set of coordinates, you simply declare a new point,  $p_1$ , as the mirror of  $p_0$  across the vertical axis. This lets you reuse the same base shape and shading logic, achieving perfect symmetry with minimal computational overhead.



(3.1.p <https://www.desmos.com/calculator/oillqby4fv>)

With this expression, you define the left eyebrow by reflecting the  $u$  coordinates to achieve symmetry without duplicating complex calculations. The point  $p_1$  is obtained by applying the same rotation used for  $p_0$ , but with the coordinates shifted in the opposite direction. This way, you generate both eyebrows from a single rotated base shape, improving efficiency while maintaining visual consistency.

To complete the reference figure, the last step is to add the crossed bones positioned behind the head. You will follow the same procedure used earlier: first, define a new rotated point, and then apply it to an equation that produces the desired shape. In this case, you will use a variation of the cylindrical form, adjusting the minimum values to sculpt the ends and create a more organic silhouette that closely resembles a bone. The resulting equation is as follows:

```

10  $R_2 = 45 \cdot \left( \frac{\pi}{180} \right)$ 
11  $p_0 = R_{rotation}(u - 0.1, v - 0.15, R_1)$ 
12  $p_1 = R_{rotation}(-u - 0.1, v - 0.15, R_1)$ 
13  $p_2 = R_{rotation}(u, v, R_2)$ 
14  $\min(0.05 - \text{abs}(p_0.x), 0.0)^2 + (p_0.y)^2 < 0.03^2$ 
15  $\min(0.05 - \text{abs}(p_1.x), 0.0)^2 + (p_1.y)^2 < 0.03^2$ 
16  $\min(0.42 - \text{abs}(p_2.x), 0.04)^2 + (\text{abs}(p_2.y) - 0.04)^2 < 0.06^2$ 

```

(3.1.q <https://www.desmos.com/calculator/g9u8f63rdi>)

This expression produces an elongated shape with rounded edges and wider ends, resulting in the characteristic form of a bone. By using 0.04 as the minimum value in the `min()` function, you smooth out the lateral cut, avoiding the straight edge of a traditional cylinder and giving the shape a more refined, stylized appearance.

It's important to limit the bone's region using the head's definition. This ensures the shape does not visually overlap the skull, maintaining a clean and well-structured composition.

Following the same approach as before, you can duplicate the bone by applying a horizontal reflection. To do this, declare a new point  $p_3$  that represents the reflection of  $p_2$  along the  $x$  axis, and then apply the same rotation transformation:

```

14  $p_3 = R_{rotation}(-u, v, R_2)$ 
15  $\min(0.05 - \text{abs}(p_0.x), 0.0)^2 + (p_0.y)^2 < 0.03^2$ 
16  $\min(0.05 - \text{abs}(p_1.x), 0.0)^2 + (p_1.y)^2 < 0.03^2$ 
17  $\min(0.42 - \text{abs}(p_2.x), 0.04)^2 + (\text{abs}(p_2.y) - 0.04)^2 < 0.06^2 \{ u^2 + v^2 > 0.38^2 \}$ 
18  $\min(0.42 - \text{abs}(p_3.x), 0.04)^2 + (\text{abs}(p_3.y) - 0.04)^2 < 0.06^2 \{ u^2 + v^2 > 0.38^2 \}$ 

```

(3.1.r <https://www.desmos.com/calculator/9m7xkdz2zf>)

Additionally, you set an extra condition on the radius ( $u^2 + v^2 > 0.38^2$ ) to ensure that the bones do not intrude into the head's space. This restriction keeps the elements separated and prevents the final result from appearing visually cluttered or overwhelming.

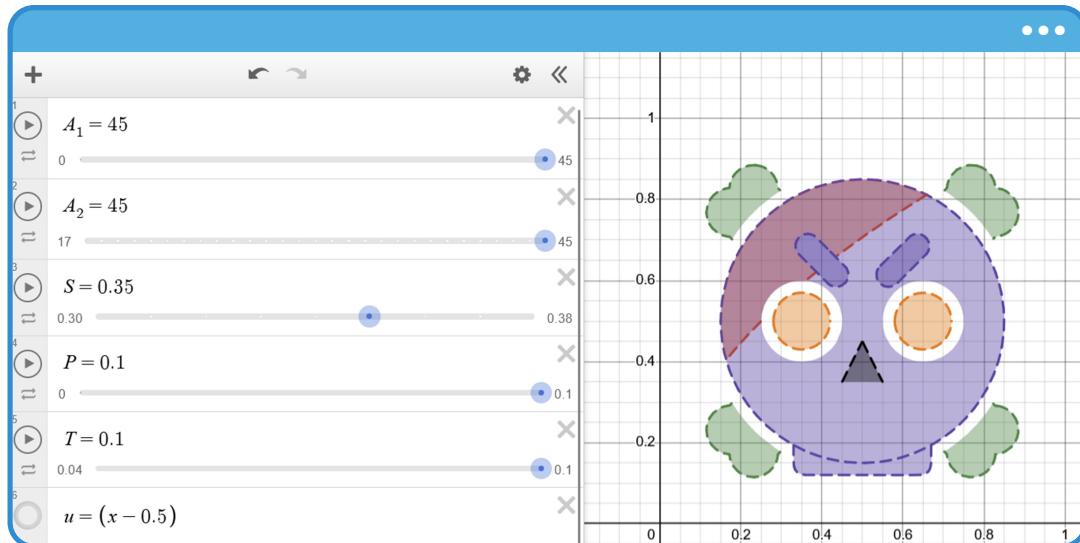
### 3.2 Animation and Procedural Modification.

Up to this point, your pirate skull has been built entirely using fixed coordinates and constant values. This approach has allowed you to interpret and construct the figure in a controlled manner. However, if you want to introduce movement or visual variations, you will need to incorporate variable values that let you manipulate key elements of the design.

To achieve this, you will define a set of properties in the Desmos interface. These variables will act as control parameters, enabling you to modify the character's expression in real time. Below is a description of each one:

- $A_1$ : Controls the eyebrow rotation angle. The range is from 0 to 45 degrees.
- $A_2$ : Adjusts the rotation angle of the rear bones, with a range from 17 to 45 degrees.
- $S$ : Sets the skull's face size, ranging from 0.30 to 0.38.
- $P$ : Moves the face (eyes, eyebrows, and nose) vertically, with a range from 0.0 to 0.1.
- $T$ : Controls the character's eye size, ranging from 0.04 to 0.1.

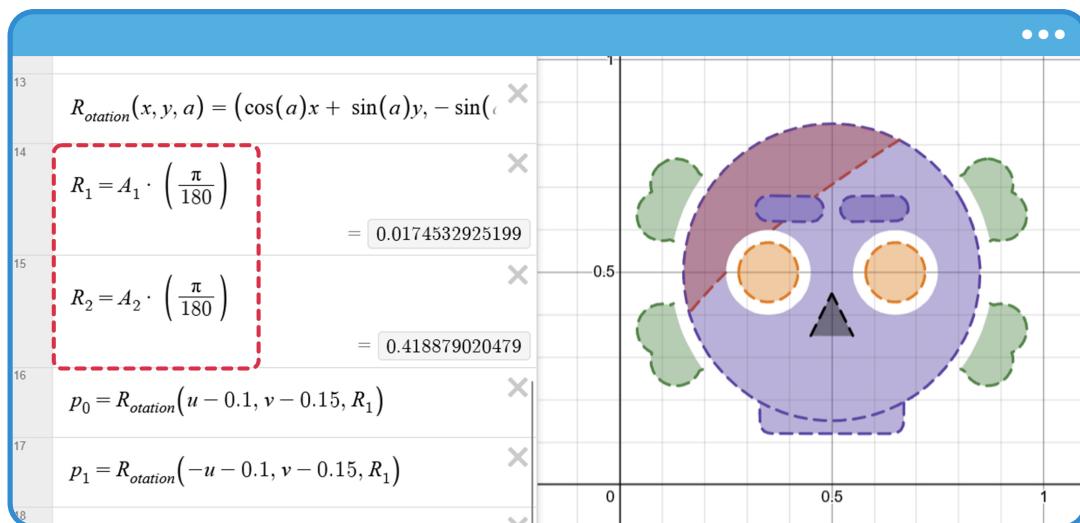
With these properties defined, you can place them at the top of the left panel in the Desmos interface, as shown below:



(3.2.a <https://www.desmos.com/calculator/yjhn9vpe3j>)

Implementing these properties in your equations is a straightforward process. You simply need to remember the purpose of each variable and replace the constant values with them in the corresponding expressions.

For example, if you replace the fixed value 45 with the variable  $A_1$  in the constant  $R_1$ , you will be able to dynamically adjust the eyebrow rotation angle. The same applies to the constant  $R_2$  – by replacing its value 45 with  $A_2$ , you can control the rotation angle of the rear bones in real time.



(3.2.b <https://www.desmos.com/calculator/v9krssdlh2>)

Let's move on to the  $S$  variable, which you will use to scale the head up or down. If you applied this variable only to the head's radius, other elements – such as the bandana, jaw, and rear bones – would remain unchanged, causing visual misalignment. To prevent this, you need to propagate the  $S$  variable to other parts of the design. Here's how to do it:

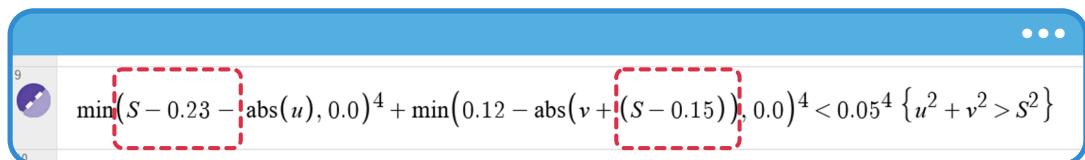
- **Head:** Replace the fixed value 0.35 with  $S$  so the head's radius becomes dynamic.
- **Bandana:** Apply  $S$  to the first limit used in constructing the bandana, ensuring its shape stays within the new scale.
- **Jaw:** Replace the value 0.35 in its corresponding limit with  $v$ , and adjust its dimensions so it scales consistently with the head.
- **Back bones:** Replace 0.38 with  $S$  in both limits, then add 0.03 to maintain their relative position to the outer edge of the skull.

If all these steps are applied correctly, the resulting expressions should look like this:

(3.2.c <https://www.desmos.com/calculator/qcdrrd2jav>)

To ensure that the jaw expands along with the head's radius, you must also adjust its dimensions based on the  $S$  variable. Specifically, you will modify both the  $u$  and  $v$  coordinates within the original equation so that its size and position adapt dynamically to the head's growth.

The adjusted expression is as follows:



(3.2.d <https://www.desmos.com/calculator/zay3r7ynov>)

Here, the amount  $S - 0.23$  defines the new maximum width allowed for the jaw, while  $S - 0.15$  adjusts its relative height within the face. By applying these values, the jaw grows proportionally with the head, preserving the original shape and avoiding visual misalignment.

The  $P$  variable will let you modify the vertical position of the facial elements, which is useful for creating expressive variations or simple animations. To keep the displacement consistent, you must apply it not only to the eyes, nose, and eyebrows, but also to the limits that define the face's contour and the bandana. The necessary changes are as follows:

- **Head:** Add  $P$  to the  $v$  coordinate to adjust its surrounding visual limits.
- **Eyes:** Apply  $P$  directly to the  $v$  variable inside the equation.
- **Nose:** Subtract  $P$  from  $v$  (that is, use  $-P - v$ ) in both the main condition and the limits to invert the displacement direction, since the nose points downward.
- **Eyebrows:** Add  $P$  to the  $y$  coordinate of the rotated points  $p_0$  and  $p_1$  to keep them aligned with the rest of the face.
- **Bandana:** Add  $P$  to the  $v$  variable inside the second limit used to trim it correctly according to the eyebrow rotation.

If all these adjustments are applied correctly, your expressions should look as follows:

The screenshot shows a sequence of procedural expressions for eye shapes. The expressions are numbered 8 through 17. Expressions 8, 10, 11, 12, and 17 have red dashed boxes around them, likely indicating they are part of a specific set or step in the process.

- 8:  $u^2 + v^2 < s^2 \{ (\text{abs}(u) - 0.15)^2 + (v + P)^2 > 0.1^2 \}$
- 10:  $(\text{abs}(u) - 0.15)^2 + (v + P)^2 < 0.07^2$
- 11:  $-P - v > 2.0 \cdot \text{abs}(u) + 0.05 \{ -P - v < 0.15 \}$
- 12:  $\sqrt{x} < y \{ u^2 + v^2 < s^2 \} \{ (\text{abs}(u) - 0.15)^2 + (v + P)^2 > 0.1^2 \} \{ \min(0.05 - \text{abs}(p_{1,x}), 0.0)^2 + (p_{1,y})^2 > 0.03^2 \}$
- 16:  $p_0 = R_{\text{rotation}}(u - 0.1, v - 0.15 + P, R_1)$
- 17:  $p_1 = R_{\text{rotation}}(-u - 0.1, v - 0.15 + P, R_1)$

(3.2.e <https://www.desmos.com/calculator/hbet5kdjgp>)

Finally, you will use the  $T$  variable to increase or decrease the character's eye size. Its implementation is straightforward: simply integrate it into the radius of the equation that defines the eyes, multiplying its value by a constant that preserves the original design's proportions and aesthetics.

In this case, the chosen base value is 0.5, as shown below:

The screenshot shows a single procedural expression for eye size control. It is numbered 10 and has a red dashed box around it.

$$(\text{abs}(u) - 0.15)^2 + (v + P)^2 < 0.5T^2$$

(3.2.f <https://www.desmos.com/calculator/bvccatrkgb>)

This approach allows you to dynamically control the eyes' opening, which can be useful both for adjusting the character's expression and for experimenting with more cartoon-like styles.

In the next section, you will implement all these expressions directly in GDSL. To do so, you will return to Godot, create a shader, assign it to a material, and set up the project structure once again.

### 3.3 Implementation of the Procedural Character in GDScript.

In this section, you will implement the mathematical equations developed in the previous sections to draw your pirate using GDScript. Before starting the coding process, you will organize the project's folder structure to maintain an orderly and easily scalable workflow. To do this:

- Inside the assets folder, create a new subfolder named **chapter\_03**, where you will store all the resources for this chapter.
- Inside **chapter\_03**, add another folder named **procedural\_shape**.
- Finally, inside **procedural\_shape**, create two subfolders:
  - **materials**.
  - **shaders**.

For this exercise, you will use only a **QuadMesh**, which you will set up in the scene using a **Node3D** along with a **MeshInstance3D**. This object will be enough to accurately visualize the pirate's shape, making it ideal for this case study.

You will also need a **shader** and its corresponding **material** to project the figure onto the Quad. In the **materials** folder, right-click and select:

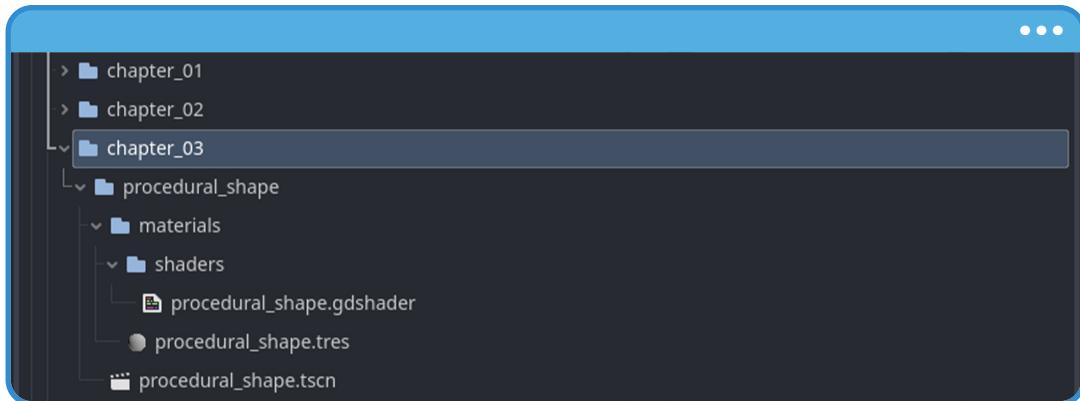
- Create New > Resource > ShaderMaterial.

For practicality, name it **procedural\_shape**. Then, create the associated shader. In the **shaders** folder, right-click and select:

- Create New > Resource > Shader.

Give it the same name (**procedural\_shape**) to keep a clear and consistent relationship between both resources.

If you have followed all the steps correctly, your project structure should look like this:



(3.3.a **procedural\_shape** has been included in the project)

**Note**

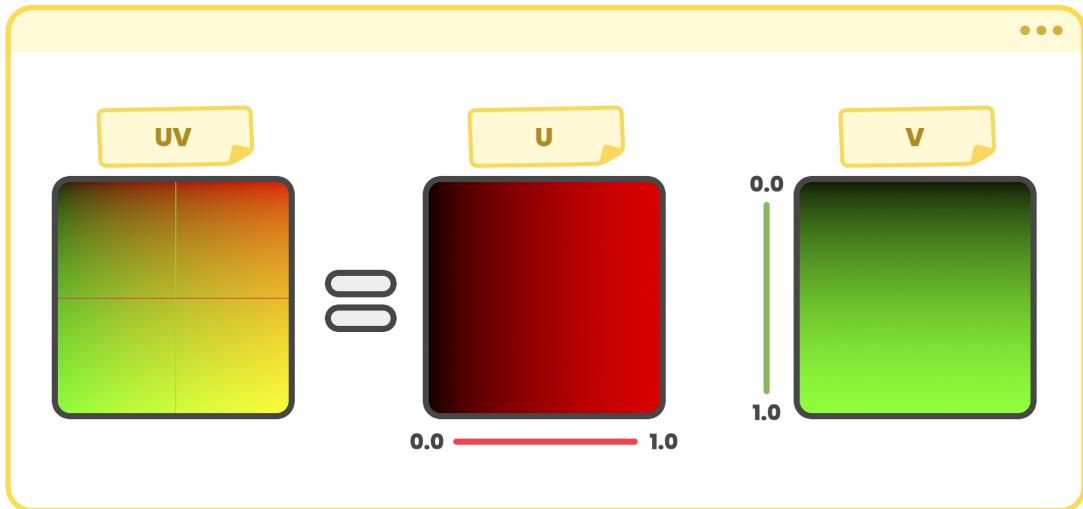
Before starting, make sure to assign the **procedural\_shape** shader to its corresponding material and then apply this material to the QuadMesh in your scene. This will allow you to visualize changes in real time as you work through the exercise.

As a first step, you will perform a quick practical test to analyze how UV coordinates behave in Godot, since their orientation may differ from other languages or environments.

If you go to the fragment stage of your shader and write the following code:

```
void fragment()
{
    vec2 uv = vec2(UV.x, UV.y);
    ALBEDO = vec3(uv, 0.0);
}
```

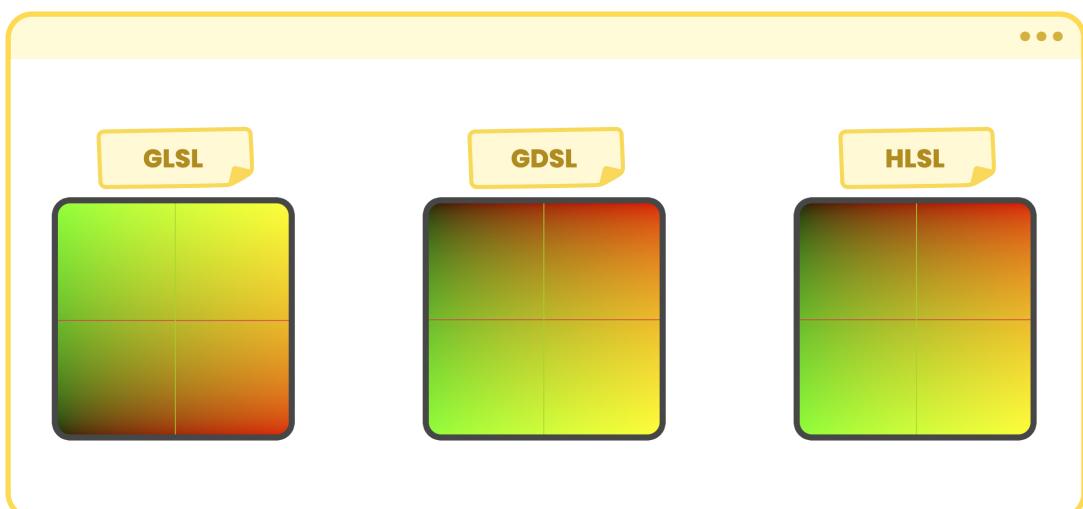
You will see, directly on the QuadMesh, that the V coordinate (equivalent to the *y*-axis in the Cartesian plane) is inverted, with the origin (0.0) located in the upper-left corner, as shown below:



(3.3.b The V coordinate points downward)

This behavior, common in languages such as HLSL, poses a problem when working with procedural shapes. If you use the same equations along with their respective variables and constants, the character will appear vertically flipped for obvious reasons.

Therefore, the first step is to adjust the V coordinate so that it follows the same orientation as the *y*-axis in the Cartesian plane. This will allow you to maintain visual consistency between what you design mathematically and what you render on screen.



(3.3.c UV coordinate comparison across languages)

**Note**

If you have previously worked with Unity, you may have noticed that UV coordinates start in the lower-left corner, not in the upper-left corner as shown in Figure 3.3.c (based on HLSL). This is because Unity supports multiple rendering APIs such as Direct3D, OpenGL, Metal, and Vulkan. To maintain visual consistency across platforms, Unity adopts a unified coordinate system and automatically flips the V coordinate, making its behavior closer to GLSL rather than HLSL.

To replicate this behavior in Godot, you simply need to invert the vertical coordinate with the following operation:

```
void fragment()
{
    vec2 uv = vec2(UV.x, 1.0 - UV.y);
    ALBEDO = vec3(uv, 0.0);
}
```

With this correction applied, you can now start drawing the different parts of the pirate skull directly in your shader. The first step is to declare the properties that will let you animate the character. While you could reuse the same names defined in Desmos, to improve readability and code clarity you will adopt a more structured naming scheme, as shown below:

```

1 shader_type spatial;
2 render_mode unshaded;
3
4 // source : https://www.desmos.com/calculator/bvccatrgb
5
6 uniform float _EyebrowRotation : hint_range(0.0, 45.0, 0.1); // A1
7 uniform float _BackBoneRotation : hint_range(17.0, 45.0, 0.1); // A2
8 uniform float _HeadSize : hint_range(0.30, 0.38, 0.01); // S
9 uniform float _FacePosition : hint_range(0.0, 0.1, 0.01); // P
10 uniform float _EyeSize : hint_range(0.04, 0.1, 0.01); // T
11
12 uniform vec3 _HeadColor : source_color;
13 uniform vec3 _EyeColor : source_color;
14 uniform vec3 _BackBoneColor : source_color;
15 uniform vec3 _BandanaColor : source_color;
16 uniform vec3 _EyebrowColor : source_color;

```

What is happening in our code? Let's break it down:

- Line 2: You set the shader to **unshaded**, since the QuadMesh doesn't need to receive scene lighting. This simplifies the display of flat shapes and colors.
- Lines 6 - 10: You declared the same properties used in Desmos ( $A_1, A_2, S, P, T$ ). However, here they are renamed using a more descriptive style consistent with common shader conventions, which makes the code easier to read and maintain.
- Lines 12 - 16: You added several **vec3** properties to define custom colors for each part of the character — such as the head, eyes, bandana, and eyebrows. These variables act as input parameters that you can modify from the Inspector.

Now that all properties are declared, you're ready to start drawing the pirate's head. You'll begin with the first of your equations: a centered circle.

Before that, you must make sure to center the UV coordinates. As you've seen, Godot's UVs go from 0.0 to 1.0, with the origin at the lower-left corner (after manually inverting V). To draw the figure in the center of the QuadMesh, subtract 0.5 from both components:

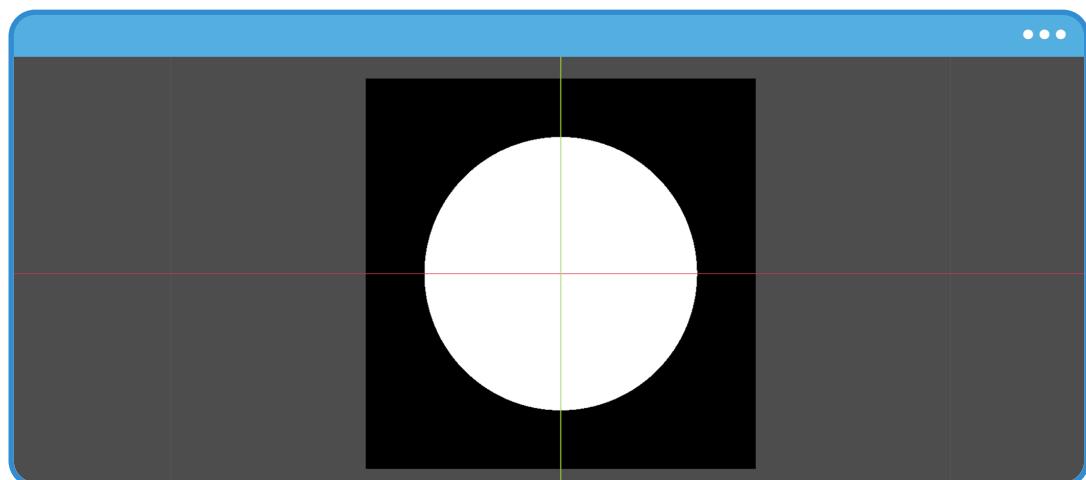
```

22 void fragment()
23 {
24     float u = UV.x - 0.5;
25     float v = (1.0 - UV.y) - 0.5;
26     vec2 uv = vec2(u, v);
27
28     float head = float(uv.x * uv.x + uv.y * uv.y < _HeadSize * _HeadSize);
29
30     ALBEDO = vec3(head);
31 }
```

As you can see:

- Lines 24 – 26: You declared a new 2D vector called **uv**, which contains the centered coordinates.
- Line 28: You declared a scalar named **head**, which represents the result of a centered-circle equation. This operation checks whether the current point (**uv**) lies within the radius defined by **\_HeadSize**. Since the comparison returns a boolean value (true or false), you explicitly cast it to float so it can be used as a visual output.
- Line 30: Finally, you assign **head** to the base color (**ALBEDO**), allowing you to visualize the shape on the Quad.

This procedure generates the visual representation of the **pirate's skull**, as shown below:



(3.3.d The pirate's head has been drawn)

It's worth noting that the initialization of **head** can be optimized by using the dot product **dot()** instead of manually summing the squared components, since both operations produce the same result:

```
float head = float(dot(uv, uv) < pow(_HeadSize, 2.0))
```

With this optimization applied, you can continue with the **jaw** declaration, which we implement using an equation composed of two **min()** functions raised to the fourth power. This shape creates a rectangular figure with softened edges. The implementation looks as follows:

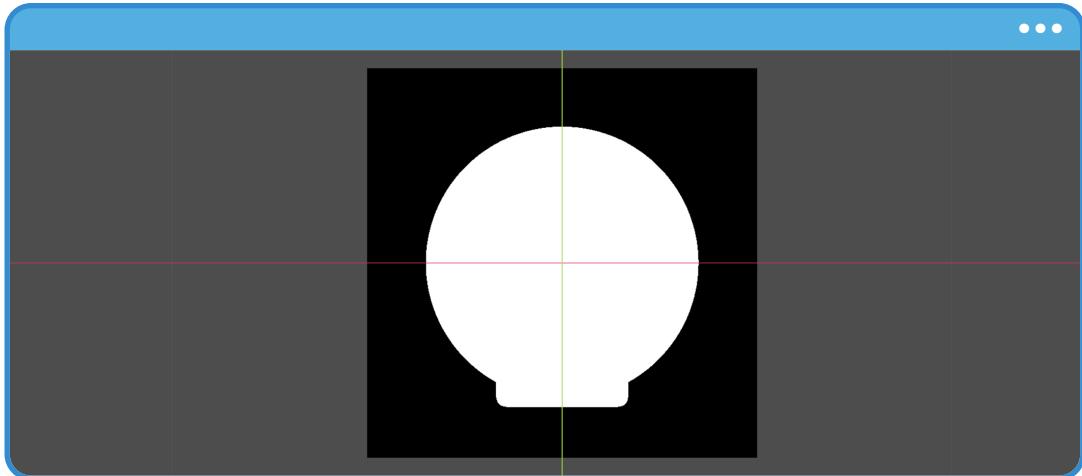
```
void fragment()
{
    ...

    float head = float(dot(uv, uv) < pow(_HeadSize, 2.0));
    float jaw = float(pow(min(_HeadSize - 0.23 - abs(u), 0.0), 4.0) +
        pow(min(0.12 - abs(v + (_HeadSize - 0.15)), 0.0), 4.0) < pow(0.05,
        4.0));

    head = clamp(head + jaw, 0.0, 1.0);

    ALBEDO = vec3(head);
}
```

Finally, you combine both shapes by adding **jaw** to **head** and using **clamp()** to limit the resulting value between 0.0 and 1.0, thus preventing potential visual artifacts caused by saturation. This operation allows you to display both the upper skull and the lower jaw as a single composite figure, preserving the aesthetic defined in Desmos.



(3.3.e The pirate's jaw has been drawn)

You can now continue with the implementation of the character's eyes. To achieve the desired effect, you will need two circles: one for the outer edge of the eye and another for its interior. To simplify the code and keep it modular, you can encapsulate this logic inside a custom function, as shown below:

```
float eyes_shape (vec2 uv, float p, float r, float s)
{
    float u_coord = abs(uv.x) - 0.15;
    float v_coord = uv.y + p;
    return float(u_coord * u_coord + v_coord * v_coord > (r * r) * s);
}
```

This function represents a shifted circle equation, based on the form described in Figure 3.1.i, with the difference that here you are using the *P* property (defined earlier in Section 3.2) to control the eyes' vertical position relative to the head. Two additional parameters are included: **r**, which represents the eye's base radius, and **s**, which acts as a scale factor to control its size.

```
void fragment()
{
    ...

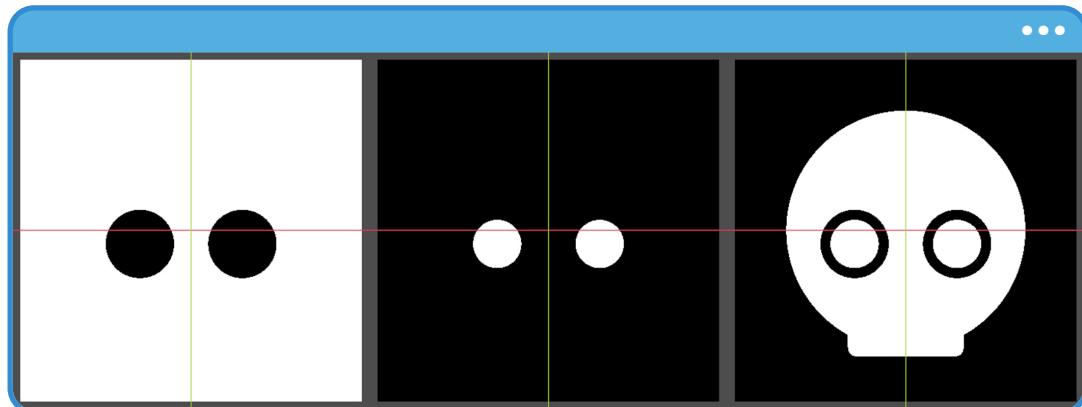
    head = clamp(head + jaw, 0.0, 1.0);
    head *= eyes_shape(uv, _FacePosition, 0.1, 1.0);
    head += 1.0 - eyes_shape(uv, _FacePosition, _EyeSize, 0.5);

    ALBEDO = vec3(head);
}
```

First, you multiply the output of `eyes_shape()` by the head, using a base radius of 0.1 and a scale factor of 1.0. This creates the eye socket, ensuring it is clipped within the head area.

Then, using the function again – but with the `_EyeSize` variable and a scale factor of 0.5 – you add the eye's interior. You invert the result so that the final shape appears as a circular white mass inside the previously defined socket.

The values used here match the parameters previously defined in Desmos, so the visual result should be practically identical to the reference.



(3.3.f Different eyes layers)

Additionally, since you've already integrated the `_FacePosition` and `_EyeSize` properties, you can return to the Inspector in Godot to adjust the eyes' position and size dynamically, giving you greater visual flexibility when experimenting with different expressions.

Before continuing with other shapes that make up the character, you need to modify your code to incorporate the color properties you declared earlier. Until now, the rendered shapes have used only black or white values (1.0 or 0.0), which is useful for masks but insufficient for the character's final render.

If you want each visual component to have its own custom RGB color, do the following:

```

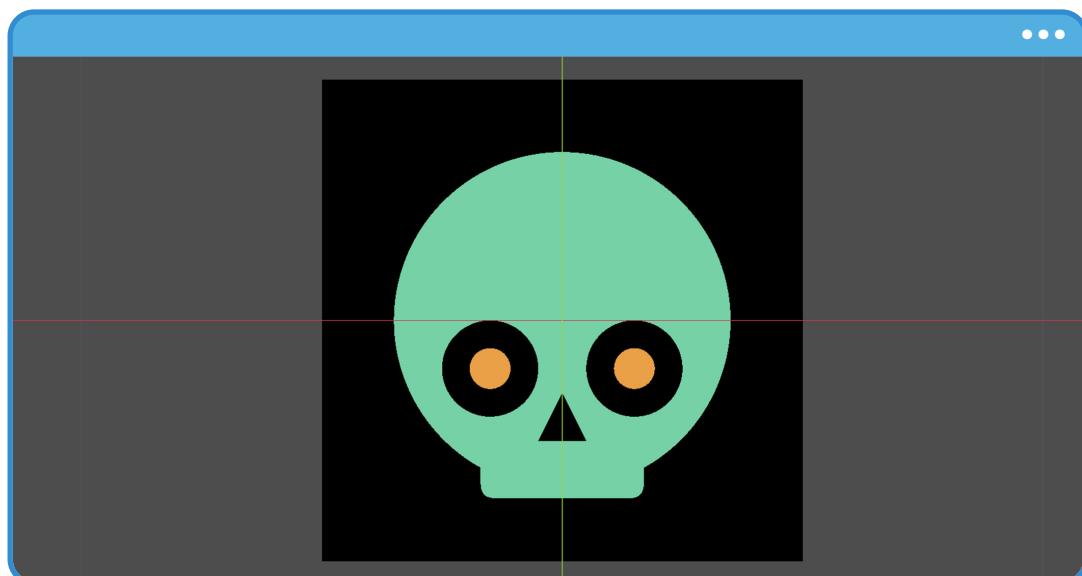
29 void fragment()
30 {
31     ...
34
35     float head = float(dot(uv, uv) < pow(_HeadSize, 2.0));
36     float jaw = float(pow(min(_HeadSize - 0.23 - abs(u), 0.0), 4.0)
37         + pow(min(0.12 - abs(v + (_HeadSize - 0.15)), 0.0), 4.0) < pow(0.05,
38             4.0));
39     float eyes = 1.0 - eyes_shape(uv, _FacePosition, _EyeSize, 0.5);
40     float nose = 1.0 - float(_FacePosition - v > 2.0 * abs(u) + 0.05)
41         * float(_FacePosition - v < 0.15);
42
43     head = clamp(head + jaw, 0.0, 1.0);
44     head *= eyes_shape(uv, _FacePosition, 0.1, 1.0);
45     head *= nose;
46
47     vec3 render_rgb = vec3(0.0);
48     float alpha = 0.0;
49
50     vec3 head_color = _HeadColor * head;
51     vec3 eyes_color = _EyeColor * eyes;
52
53     render_rgb += head_color;
54     render_rgb += eyes_color;
55
56     ALBEDO = render_rgb;
}

```

What's happening in your code? Let's break down each part:

- Line 38: You declare a new scalar named **eyes**, which stores the eye value generated earlier (previously added directly to **head**). You now separate it so you can mask it with its own color.
- Lines 39 – 40: You define the nose variable, which represents the triangular **nose** shape based on the equation described in Figure 3.1.j. It includes the properties needed to control its position dynamically.
- Line 44: You multiply **head** by **nose** to visually subtract the nose from the head, creating a hole in its place.
- Lines 46 – 47: You declare **render\_rgb**, an RGB vector that will accumulate the fragment's final color, and **alpha**, a scalar to control the shader's transparency.
- Line 49: You create **head\_color**, an RGB vector containing the defined color for the head, and later multiply it by the **head** mask.
- Line 50: You create **eyes\_color**, which contains the eye color, and mask it with the **eyes** variable.
- Lines 52 – 53: You add **head\_color** and **eyes\_color** to the **render\_rgb** accumulator.
- Line 55: You assign the final result to the shader output via **ALBEDO**.

With this implementation, if you assign custom colors from the **Inspector** to the **\_HeadColor** and **\_EyeColor** properties, you will obtain the following visual result:



(3.3.g Skull color: 00d2a6; eyes color: ffa13e)

Next, you will implement the pirate's bandana. As developed earlier in Desmos, this shape is composed of a curve that is trimmed in three specific regions:

- Around the head,
- below the eyebrows,
- and along the upper edges of the eyes.

To simplify the process in this first pass, you will focus on the head's general mask, since it already contains the necessary constraints to avoid overlapping with the eyes and nose. The code is as follows:

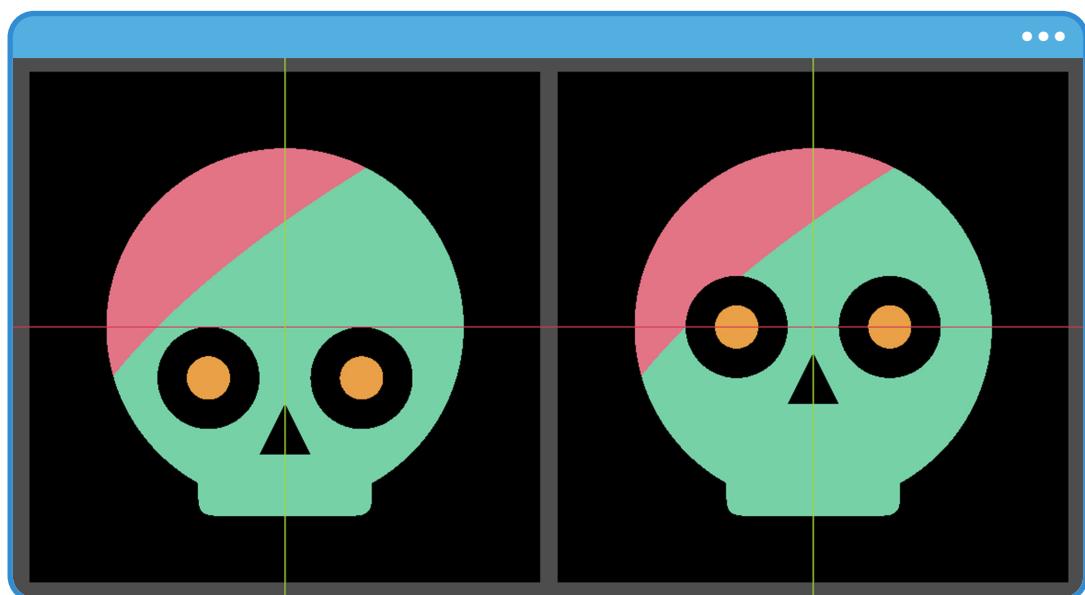
```

29 void fragment()
30 {
31     ...
32
33     float bandana = float(sqrt(UV.x) < 1.0 - UV.y);
34
35     head = clamp(head + jaw, 0.0, 1.0);
36     head *= eyes_shape(uv, _FacePosition, 0.1, 1.0);
37     head *= nose;
38     bandana *= head;
39
40     vec3 render_rgb = vec3(0.0);
41     float alpha = 0.0;
42
43     vec3 head_color = _HeadColor * head;
44     vec3 eyes_color = _EyeColor * eyes;
45     vec3 bandana_color = _BandanaColor * bandana;
46
47     render_rgb += head_color;
48     render_rgb += eyes_color;
49     render_rgb = mix(render_rgb, bandana_color, bandana);
50
51     ALBEDO = render_rgb;
52 }
```

Let's break down what's happening:

- Line 41: You declare the `bandana` variable, which defines its shape using the comparison `sqrt(UV.x) < 1.0 - UV.y`. This curve crosses the face, matching the design shown in Desmos.
- Line 46: You mask `bandana` with `head`, ensuring it only renders within the skull's boundaries. Since the head already excludes the eyes and nose, no additional masking is needed.
- Line 53: You define `bandana_color`, an RGB vector that contains the bandana's color multiplied by its mask.
- Line 57: You perform a linear interpolation between the accumulated `render_rgb` and `bandana_color`, using `bandana` as the weight. This ensures a smooth visual blend when shapes partially overlap.

If you go back to the **Inspector** and select a color for the bandana, you will obtain the following result:



(3.3.h Bandana color: ff7587)

Only the eyebrows and the back bones remain to be implemented. You already know these elements need to be rotated, so you will define a function that lets you apply rotations to specific points in space:

```
vec2 rotation(float x, float y, float a)
{
    float ru = cos(a) * x + sin(a) * y;
    float rv = -sin(a) * x + cos(a) * y;
    return vec2(ru, rv);
}
```

This function, first introduced in Section 3.1, lets you rotate a two-dimensional point by an angle  $a$  expressed in radians. Although you could optimize the  $x$  and  $y$  parameters by using a single `vec2`, they are kept separate to maintain a more direct correspondence with the original function shown in Figure 3.1.n, making its mathematical analysis easier.

Once implemented, you can use this function inside the fragment to rotate key points of the character. In this case, you will start with the right eyebrow. However, since you need to represent both eyebrows — left and right — you will define a new method that lets you determine their shape in a reusable way. Proceed as follows:

```
float eyebrow_shape(vec2 p)
{
    float r = 0.03;
    float u = min(0.05 - abs(p.x), 0.0);
    float v = p.y;
    vec2 uv = vec2(u, v);
    return float(dot(uv, uv) < r * r);
}
```

The `eyebrow_shape()` function is a direct translation into GDSL of the form shown in Figure 3.1.o. In this case, you use the expression `dot(uv, uv)` to compute the squared magnitude of the `uv` vector, which is an optimized way to check whether a point lies within a radius `r`.

**Note**

There are several ways to express this operation: (1)  $x * x + y * y$ , (2)  $\text{pow}(x, 2.0) + \text{pow}(y, 2.0)$ , and (3)  $\text{dot}(\text{uv}, \text{uv})$ . Of the three, `dot()` is the most efficient and readable, which is why it is used here to define the eyebrow's curved shape.

At this point, you will start with the right eyebrow, which will be evaluated using the `eyebrow_shape()` method, as shown below:

```

45 void fragment()
46 {
47     ...
48
49     float r1 = _EyebrowRotation * (PI / 180.0);
50     vec2 p0 = rotation(u - 0.1, v - 0.15 + _FacePosition, r1);
51     float eyebrow_r = eyebrow_shape(p0);
52
53     head = clamp(head + jaw, 0.0, 1.0);
54     head *= eyes_shape(uv, _FacePosition, 0.1, 1.0);
55     head *= nose;
56     bandana *= head;
57
58     vec3 render_rgb = vec3(0.0);
59     float alpha = 0.0;
60
61     vec3 head_color = _HeadColor * head;
62     vec3 eyes_color = _EyeColor * eyes;
63     vec3 bandana_color = _BandanaColor * bandana;
64     vec3 eyebrows_color = _EyebrowColor * eyebrow_r;
65
66     render_rgb += head_color;
67     render_rgb += eyes_color;
68     render_rgb = mix(render_rgb, bandana_color, bandana);
69     render_rgb = mix(render_rgb, eyebrows_color, eyebrow_r);
70
71     ALBEDO = render_rgb;
72 }

```

Let's break down what's happening:

- Line 59: The value of `_EyebrowRotation` is converted from degrees to radians, and the result is stored in the `r1` variable.
- Line 60: A prior translation is applied to place the eyebrow's local origin with an offset in `u` and `v`, plus the dynamic vertical adjustment via `_FacePosition`. Then the `rotation()` function is applied using the `r1` angle. The result is stored in the `p0` vector.
- Line 61: A new scalar named `eyebrow_r` is declared and initialized, corresponding to the pirate's right eyebrow.
- Line 74: A new RGB vector named `eyebrows_color` is declared and initialized, multiplying the eyebrow color by the `eyebrow_r` mask so that color is applied only to the pixels where the eyebrow shape is present.
- Line 79: A linear interpolation is performed between the current color accumulated in `render_rgb` and the eyebrow color `eyebrows_color`.

Once the changes are saved, by selecting a color for the `_EyebrowColor` property in the **Inspector**, you can clearly see the right eyebrow rendered on the QuadMesh.

Next, you will add the left eyebrow using a horizontal reflection, following the same principle previously applied in Desmos and revisited in Figure 3.1.p. To do this, return to the code and perform a symmetric operation to the one used for the right eyebrow.

```

void fragment()
{
    ...

    float r1 = _EyebrowRotation * (PI / 180.0);
    vec2 p0 = rotation(u - 0.1, v - 0.15 + _FacePosition, r1);
    vec2 p1 = rotation(-u - 0.1, v - 0.15 + _FacePosition, r1);
    float eyebrow_r = eyebrow_shape(p0);
    float eyebrow_l = eyebrow_shape(p1);
    float eyebrows = eyebrow_l + eyebrow_r;

    ...
    vec3 eyebrows_color = _EyebrowColor * eyebrows;

    ...
    render_rgb = mix(render_rgb, eyebrows_color, eyebrows);

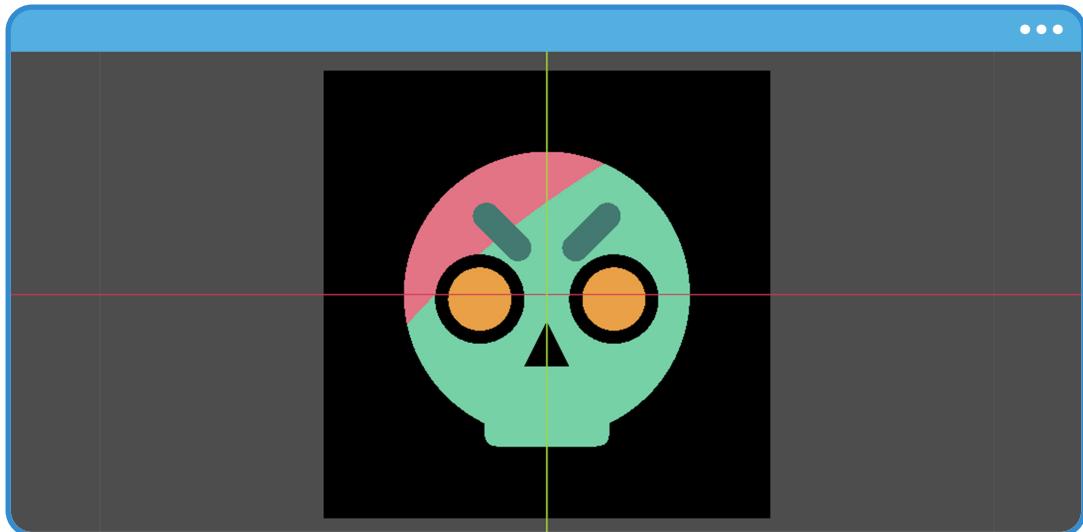
    ALBEDO = render_rgb;
}

```

As you can see, a new point named **p1** has been declared, which corresponds to the horizontal reflection of **p0** to generate the left eyebrow. The shape of each eyebrow is then evaluated once again using the **eyebrow\_shape()** function, and the results are stored in **eyebrow\_r** (right) and **eyebrow\_l** (left). Both values are added and stored in the **eyebrows** variable, which represents the combined mask of both eyebrows.

Finally, this mask is used both to generate the corresponding color (**eyebrows\_color**) and to perform a linear interpolation over the accumulated color (**render\_rgb**), visually integrating the eyebrows with the rest of the character.

These operations produce the following visual result:



(3.3.i Dark turquoise color: 007a71)

Finally, you will add the back bones. To do this, you will start from the equation shown in Figure 3.1.q, which defined this shape in Desmos. Its direct translation into GDSL is as follows:

```
float back_bone_shape(vec2 p)
{
    float r = 0.06;
    float u = min(0.42 - abs(p.x), 0.04);
    float v = abs(p.y) - 0.04;
    vec2 uv = vec2(u, v);
    return float(dot(uv, uv) < r * r);
}
```

Next, you will need to rotate both back bones. To do this, apply the `rotation()` function to two new points:  $p_2$  and  $p_3$ , which correspond to the right and left bones, respectively. This procedure is integrated into the fragment stage as follows:

```

void fragment()
{
    ...

    float to_radians = PI / 180.0;
    float r1 = _EyebrowRotation * to_radians;
    float r2 = _BackBoneRotation * to_radians;

    vec2 p0 = rotation(u - 0.1, v - 0.15 + _FacePosition, r1);
    vec2 p1 = rotation(-u - 0.1, v - 0.15 + _FacePosition, r1);
    vec2 p2 = rotation(u, v, r2);
    vec2 p3 = rotation(-u, v, r2);

    ...
}

```

If you look closely at the previous code snippet, you'll notice a new variable named **to\_radians**, which converts degree values to radians. This conversion is essential for correctly using trigonometric functions in GDSL, since they expect angles in radians. Thanks to this constant, you can enter values in degrees from the Inspector, keeping a more intuitive interface for adjusting rotations.

The rotated values of **\_EyebrowRotation** and **\_BackBoneRotation** are stored in **r1** and **r2**, respectively. Then, two new two-dimensional vectors are declared: **p2** and **p3**. The first represents the base point for one of the back bones, while the second is its reflection across the vertical axis, creating the symmetry needed on both sides of the character.

Once these points are defined, you can use the **back\_bone\_shape()** function to draw the bones in the shader. However, before doing so, remember that the back bones partially overlap with the head and that their area is slightly larger, as observed in Figure 3.1.r.

For this reason, it's a good idea to reuse the head mask as a way to visually limit the bones. To do this, you will encapsulate the operation that computes the head into a new function, as shown below:

```
float head_shape(vec2 uv, float s)
{
    return float(dot(uv, uv) < pow(s, 2.0));
}
```

The `head_shape()` method returns the same circular mask used to represent the head. Thanks to this reusable function, you can now directly replace the previous operation with a clearer, more concise call:

```
float head = head_shape(uv, _HeadSize);
```

This line improves code readability and lets you reuse the same logic in other elements that depend on the head's shape, such as the back bones.

To generate these bones, you start by evaluating their shape using the rotated points `p2` and `p3` defined earlier. These points represent the symmetric ends of the back, and they are evaluated with the `back_bone_shape()` function as shown below:

```

void fragment()
{
    ...
    float eyebrows = eyebrow_l + eyebrow_r;

    float back_bone_l = back_bone_shape(p2);
    float back_bone_r = back_bone_shape(p3);
    float back_bones = clamp(back_bone_l + back_bone_r, 0.0, 1.0);

    ...
    bandana *= head;
    back_bones *= 1.0 - head_shape(uv, _HeadSize + 0.03);

    ...
    vec3 eyebrows_color = _EyebrowColor * eyebrows;
    vec3 back_bone_color = _BackBoneColor * back_bones;

    ...
    render_rgb += back_bone_color;

    ALBEDO = render_rgb;
}

```

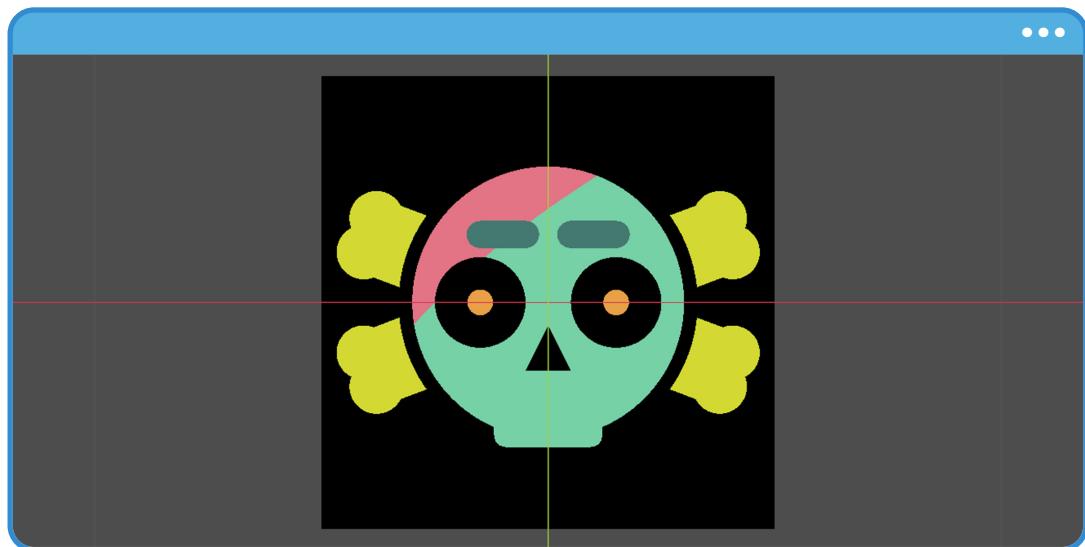
The sum of both evaluations generates the `back_bones` mask, which represents the two bones. You apply a `clamp()` function to ensure the final value stays between 0.0 and 1.0, preventing interpolation or saturation issues.

Next, you use the head shape as an inverse mask over the bones so they don't visually interfere with the character's face. To do this, you slightly increase the radius and subtract the mask, as shown in this line of code:

```
back_bones *= 1.0 - head_shape(uv, _HeadSize + 0.03);
```

Finally, you compute the corresponding bone color and integrate it into the overall color composition. The color is applied only where the `back_bones` mask has value, ensuring the

bones blend harmoniously into the visual composition. When you view the final result on screen, you should see the complete pirate with all elements correctly masked, rotated, and colored.



(3.3.j Back bones color: d2da00)

Up to this point, you could consider the pirate complete in terms of shape and color. However, one essential aspect is still missing to integrate it correctly into any scene: transparency. Adding transparency in GDScript is a simple, straightforward process.

To achieve this, you will reuse the alpha variable, previously declared as the opacity accumulator. Then, you will add the masks corresponding to the character's visible elements: the head (**head**), the eyes (**eyes**), and the back bones (**back\_bones**). This composite value will be used as the alpha channel in your fragment.

The resulting code looks as follows:

```
void fragment()
{
    ...
    render_rgb += back_bone_color;

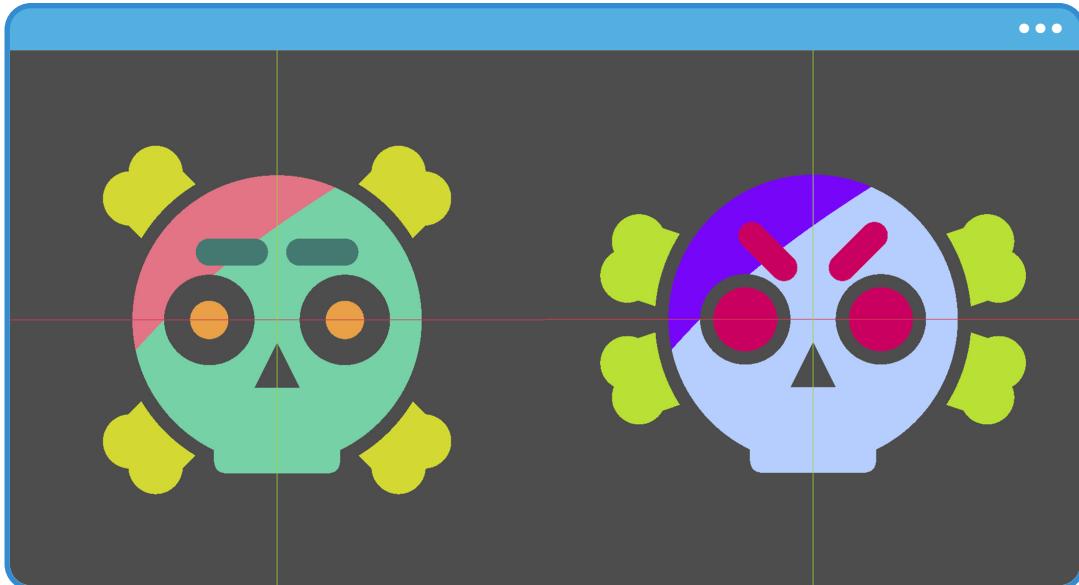
    alpha += head;
    alpha += eyes;
    alpha += back_bones;

    float back = back_bone_shape(p2);

    ALBEDO = render_rgb;
    ALPHA = alpha;
}
```

Thanks to this cumulative sum, transparency is applied only in the areas where the masks are present. This allows the rest of the quad to remain completely transparent, removing any unwanted background around the character.

When you view the result, you will notice that the pirate is now fully integrated with an alpha channel that outlines only its silhouette. This is particularly useful if you want to project it onto different scenes or combine it with other visual effects.



(3.3.k Different color configuration for the procedural pirate skull)

### 3.4 Vertex Transformation and Animation.

When you think of vertices, you probably picture the small magenta cubes (or black ones in Blender) that mark the edges of a 3D mesh. But beyond their visual representation, vertices are simply numerical values (vectors) that store essential information, such as their position in space relative to the model.

Understanding this is crucial, especially when you want to create effects through vertex transformation and animation. In this context, transformation and animation refer to any operation that modifies the position of a vertex in space. To achieve this, you'll rely on basic mathematical operations — addition, multiplication, division — and, most importantly, one fundamental variable: time. Everything happens as a function of time!

Godot provides a built-in variable called **TIME**, which, according to the official documentation:

“

Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's affected by `time_scale` but not by pausing. If you need a `TIME` variable that is not affected by time scale, add your own global shader uniform and update it each frame.

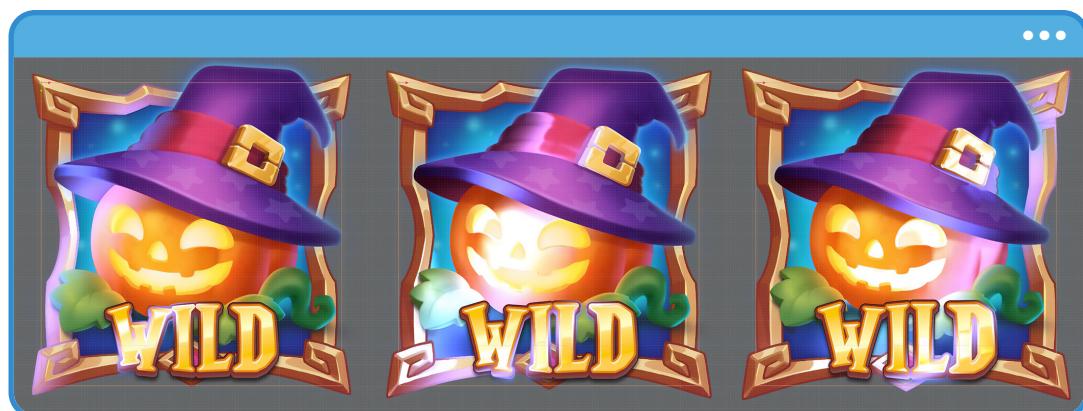
”

In this section, you'll experiment with vertex deformation and animation applied to a composition of images. And what better way to do it than directly on UI elements? Why UI images? Because, from a technical standpoint, they present a bigger challenge: in both the vertex and fragment stages, you have fewer built-in variables available. In practice, this means you'll need to design most of the logic yourself, which pushes you to develop more creative solutions to achieve the effect.

To put these concepts into practice, you'll build a shader that performs two main tasks:

- ① Animate the scale of the UI vertices.
- ② Add an animated specular effect, which automatically adapts to any image you use in the composition.

The second task is especially interesting because you'll be working with a **single material** applied to the entire composition, and yet the results will dynamically adapt to each individual image — as shown in the following reference:



(3.4.a Vertex animation and specularity)

Start by organizing your project as follows:

- Inside the **chapter\_03** folder, create a new folder named **vertex\_animation**.
- Within this folder, create three subfolders:
  - **materials**
  - **shaders**
  - **textures**

For this exercise, you'll need both a shader and a material. In the **materials** folder, right-click and select:

- Create New > Resources > ShaderMaterial.

Name this material **ui\_animation**.

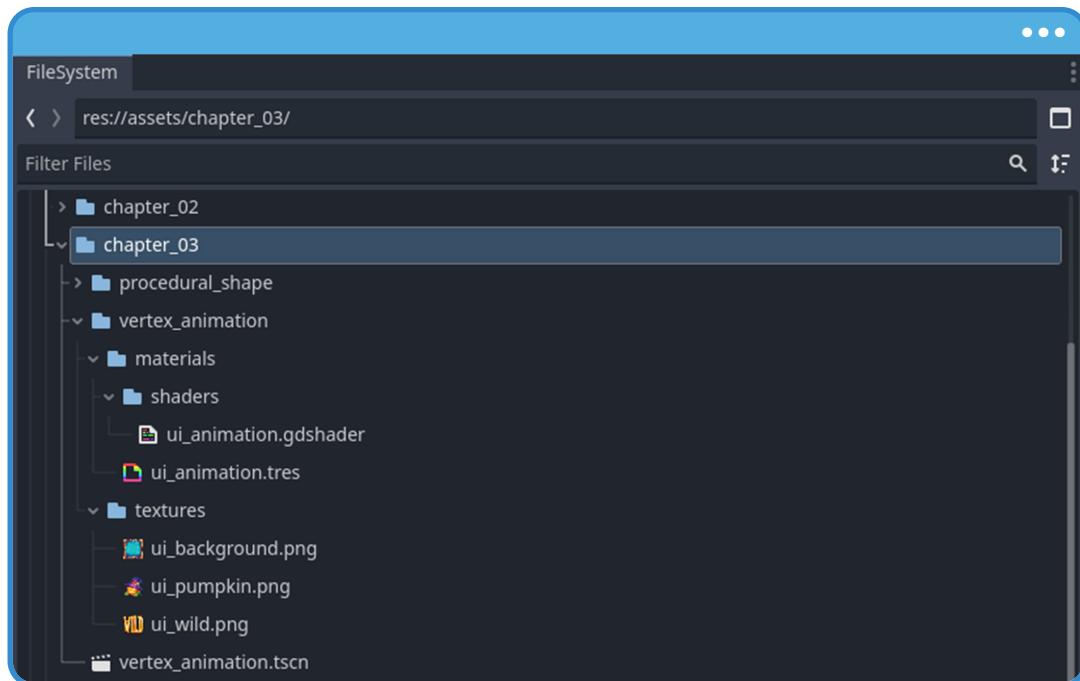
Next, create the corresponding shader in the **shaders** folder by right-clicking and choosing:

- Create New > Resources > Shader.

As usual, name it **ui\_animation** to keep everything consistent with the section.

Make sure you also download the images that will be used throughout this section. These are included in the downloadable package that comes with the book and include: **ui\_background**, **ui\_pumpkin**, and **ui\_wild**.

If everything has been set up correctly, your project folder structure should look like this:

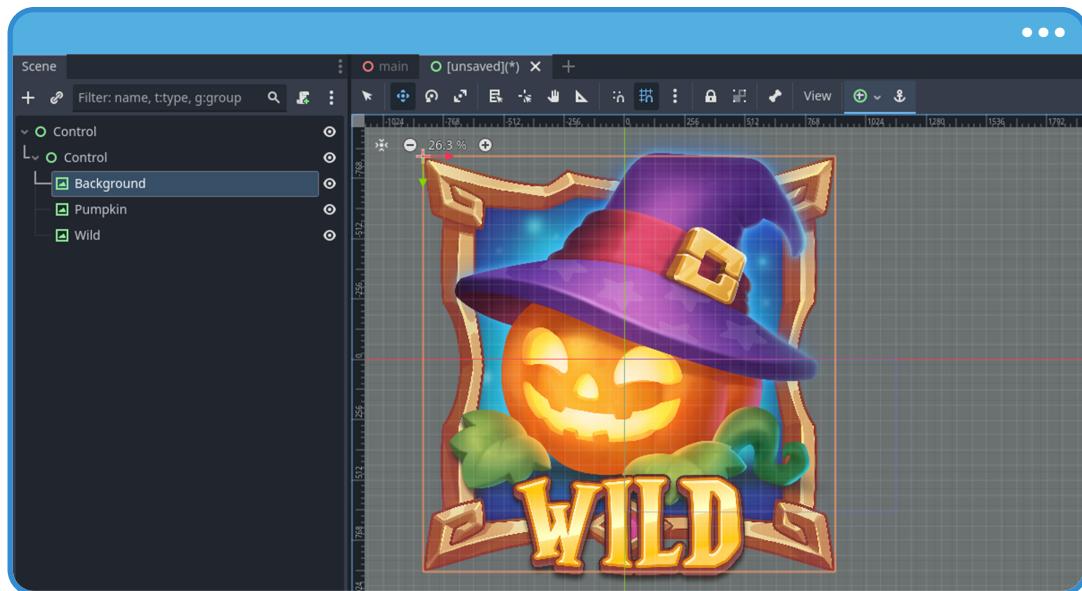


(3.4.b Project structure)

Before writing the shader, you'll set up the scene and the elements needed for this section. You'll also need to assign the **ui\_animation** material to each UI image through its **Material** property.

Start by creating a new empty scene with a **User Interface** node as the root. Add a **Control** node as its child, and under it, create three **TextureRect** nodes named: **Background**, **Pumpkin**, and **Wild**. Assign each texture to its respective node, then make sure all elements are centered on the screen. To do this, select **Anchor Preset** and set it to **Center**.

If everything has been positioned correctly, your scene should look like this:

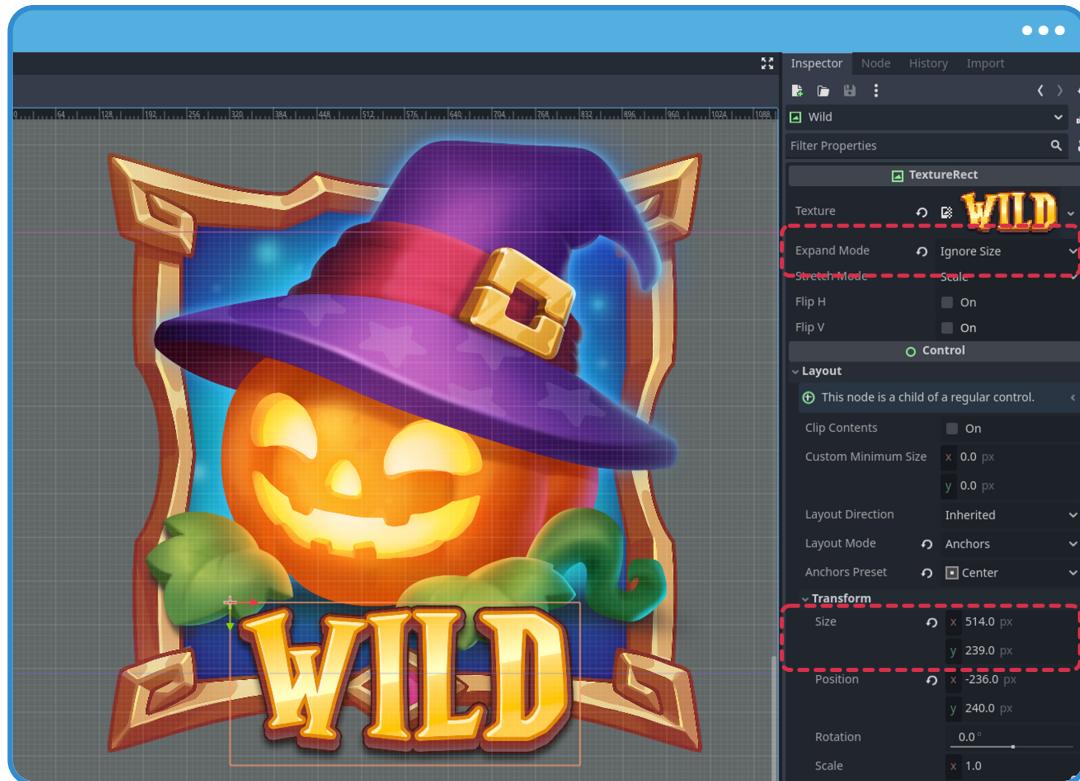


(3.4.c UI elements)

The images you're working with are quite large – in fact, they were created on a 2048x2048 canvas. To prevent the composition from exceeding the visible area in the final render when the game is compiled, you'll reduce their size by half. To do this, follow these steps:

- Select one of the images.
- In the **Inspector**, under **TextureRect**, change the **Expand Mode** property to **Ignore Size**. This will allow you to manually resize the image.
- Then go to **Layout > Transform** and multiply both the **X** and **Y** scale components by 0.5. You'll use this value later in the shader to ensure that proportions remain correct during vertex animation and transformation.

Repeat the same process for each image in the UI.



(3.4.d All images have been reduced to half their original size)

In Godot, you can work with different types of shaders. Throughout this book, you've mostly used the `spatial` type, which is ideal for 3D elements. However, in this case, you'll be working with UI images, so you'll need a different type of shader. Replace the shader type `spatial` with `canvas_item`, and add a few properties that you'll use to recreate the effects in this section:

```
shader_type canvas_item;

uniform bool _VertexAnimation;
uniform float _VertexScale : hint_range(0.0, 0.5, 0.01);
uniform float _VertexSpeed : hint_range(1.0, 3.0, 0.1);
uniform float _SpecularSpeed : hint_range(0.5, 2.0, 0.1);
uniform vec3 _SpecularColor : source_color;

void vertex() { ... }
```

Whenever you work with UI images in Godot, you must use `canvas_item` shaders because this type is designed specifically for drawing and manipulating 2D elements.

In the code, you can see several properties, each with a specific purpose:

- `_VertexAnimation` (bool): toggles the vertex animation effect on or off.
- `_VertexScale` (float): defines the maximum scale size used in the UI image animation.
- `_VertexSpeed` (float): controls the speed of the animation in seconds.
- `_SpecularSpeed` (float): determines the speed of the specular effect over the UI image.
- `_SpecularColor` (vec3): adjusts the color of the specular effect.

Because you'll be applying a custom transformation to the vertices during animation, you need to include the render mode `skip_vertex_transform`. As explained back in Section 1.11 of Chapter 1, this property disables Godot's automatic vertex transformation inside the `vertex()` function. Once you add it, you'll notice that the images shift position in the Viewport. That happens because you now have complete control over vertex transformations.

```
shader_type canvas_item;
render_mode skip_vertex_transform;
```

To fix the position, multiply the `MODEL_MATRIX` by the local-space vertices and store the result in `VERTEX`, like this:

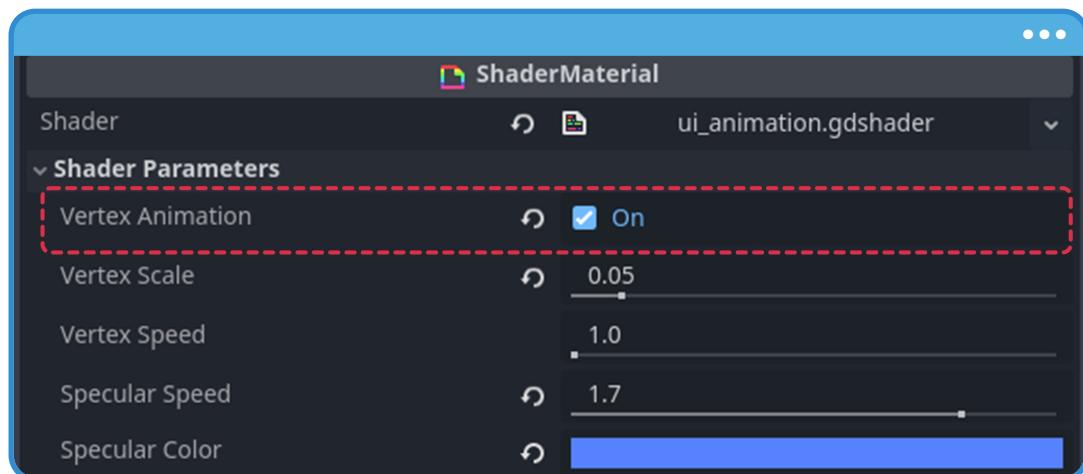
```
void vertex()
{
    if (!_VertexAnimation)
    {
        VERTEX = (MODEL_MATRIX * vec4(VERTEX, 0.0, 1.0)).xy;
    }
}
```

In this example, you've added a condition: when `_VertexAnimation` is set to `false` (its default value), the vertices remain unchanged. When it's set to `true`, you'll apply an animation by

scaling the UI vertices over time. To start visualizing this process, try the following simple example:

```
void vertex()
{
    if (!_VertexAnimation)
    {
        VERTEX = (MODEL_MATRIX * vec4(VERTEX, 0.0, 1.0)).xy;
    }
    else
    {
        vec2 scaled_vertex = VERTEX * vec2(1.0);
        VERTEX = (MODEL_MATRIX * vec4(scaled_vertex, 0.0, 1.0)).xy;
    }
}
```

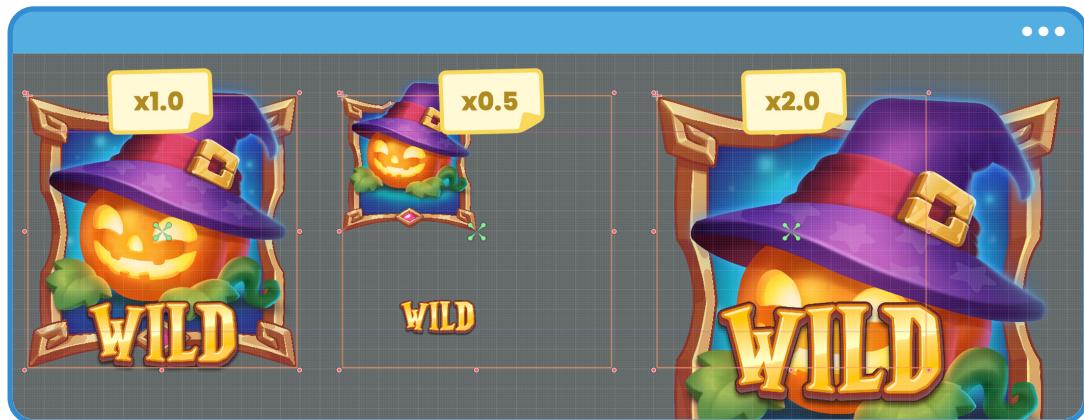
Both transformations currently produce the same result, but in the second case (the `else` block), you've explicitly introduced a scaling operation on the UI vertices. To see the effect, enable the **Vertex Animation** property from the **Inspector** in your project's material.



(3.4.e The **Vertex Animation** property has been enabled)

If you go back to your code and change the vertex multiplication value to something greater or less than 1.0, you'll see that the UI vertices do scale. However, the reference point for this

scaling is located at the top-left corner of the UI element. This isn't ideal, since what you really need is for the transformation to occur relative to the center of the object.



(3.4.f Different vertex scales)

There's a quick solution to the scaling problem: subtract half of the image's current size. To understand this, imagine you're working with a  $512 \times 512$  image. If the scale is 1, no correction is necessary. But if the scale is 2, you need to subtract half the size of the image ( $512 / 2$ ) from the vertices.

What happens if you need to scale by an arbitrary value  $n$ ? Let's define a few variables:

- $a$  → the scale value.
- $b$  → the value that needs to be subtracted.
- $x$  → the current resolution of the UI image.

From these, you can deduce the following cases:

- If  $a = 1$ , then  $b = 0$ .
- If  $a = 2$ , then  $b = x/2$ .
- If  $a = 3$ , then  $b = x$ .
- If  $a = 4$ , then  $b = x + (x/2)$ .
- If  $a = 5$ , then  $b = x + x$ .
- If  $a = 6$ , then  $b = x + x + (x/2)$ .
- If  $a = 7$ , then  $b = x + x + x$ .

Looking at the pattern, you can generalize with the following formula:

$$b = x * \frac{(a - 1)}{2}$$

(3.4.g)

This formula can be implemented in your shader through a helper function:

```
float scale_from_center(float a, float x)
{
    float b = x * (a - 1.0) * 0.5;
    return b;
}

void vertex() { ... }
```

To access the current resolution of the image, you can use Godot's built-in variable `TEXTURE_PIXEL_SIZE`, which the documentation defines as:

“

Normalized pixel size of default 2D texture. For a `Sprite2D` with a texture of size 64x32px, `TEXTURE_PIXEL_SIZE = vec2(1/64, 1/32)`.

”

Using this, you can implement centered scaling with the following code block:

```

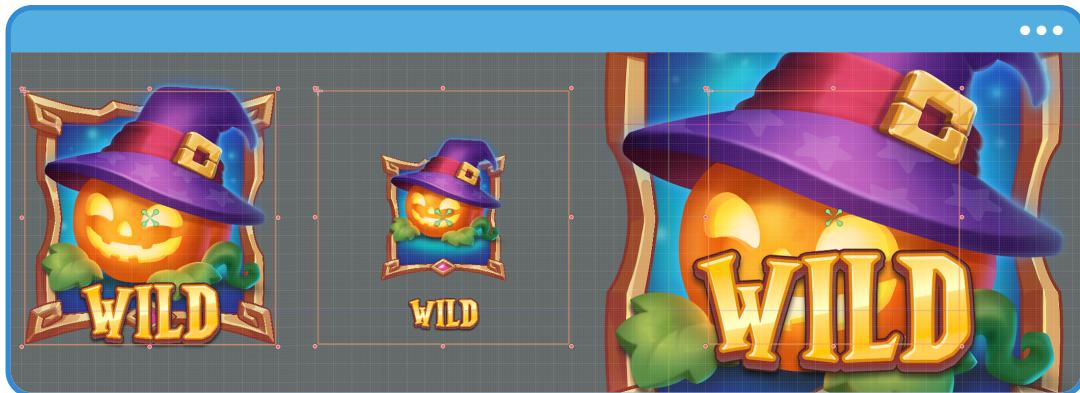
22 else
23 {
24     float scale_ratio = 1.0;
25     const float pixel_scaler = 0.5;
26
27     vec2 resolution = pixel_scaler / TEXTURE_PIXEL_SIZE;
28     float px = scale_from_center(scale_ratio, resolution.x);
29     float py = scale_from_center(scale_ratio, resolution.y);
30
31     vec2 scaled_vertex = VERTEX * vec2(scale_ratio);
32     scaled_vertex -= vec2(px, py);
33
34     VERTEX = (MODEL_MATRIX * vec4(scaled_vertex, 0.0, 1.0)).xy;
35 }

```

What happens line by line?

- Line 24: Define **scale\_ratio**, the factor applied to the UI vertices.
- Line 25: **pixel\_scaler = 0.5** reflects that you reduced the images by 50% in the Inspector. If you hadn't reduced them, this value would be 1.0.
- Line 27: Compute resolution in effective pixels. Since **TEXTURE\_PIXEL\_SIZE = (1/width, 1/height)**, dividing **pixel\_scaler** by it gives **(pixel\_scaler × width, pixel\_scaler × height)**, i.e., the resolution after reducing to 50%.
- Lines 28 – 29: Use **scale\_from\_center()** (Fig. 3.4.g) to calculate the necessary offset for each axis.
- Line 31: Scale the vertices by **scale\_ratio**.
- Line 32: Subtract **(px, py)** to re-center the transformation, ensuring scaling happens from the image's geometric center.

When you modify **scale\_ratio**, you'll now see that the UI images scale correctly from their center instead of the top-left corner.



(3.4.h The UI images scaled from the center)

At this point, you can experiment with different mathematical operations to animate scaling. For example, using `sin()` allows you to automatically increase and decrease the size of the images:

```
else
{
    float time = (TIME * _VertexSpeed) * PI;
    float scale_ratio = sin(time) * _VertexScale + (1.0 +
        _VertexScale);
    const float pixel_scaler = 0.5;

    ...
}
```

Here, time multiplies the global `TIME` constant by `_VertexSpeed` and by `PI`. The `sin()` function outputs a value in the range  $[-1:1]$ . You then scale its amplitude with `_VertexScale`, and add a base term so that the minimum scale is always 1.0. The result is stored in `scale_ratio`, which controls how the image size changes on screen. By adjusting `_VertexSpeed` and `_VertexScale` from the Inspector, you'll see the UI being dynamically animated.

The issue is that all images share the same material, which means they animate in sync. To introduce variation per image, you can take advantage of the built-in variable `COLOR`, a four-component vector. According to the documentation, this corresponds to:

“

Color from vertex primitive multiplied by CanvasItem's **modulate** multiplied by  
CanvasItem's **self\_modulate**.

”

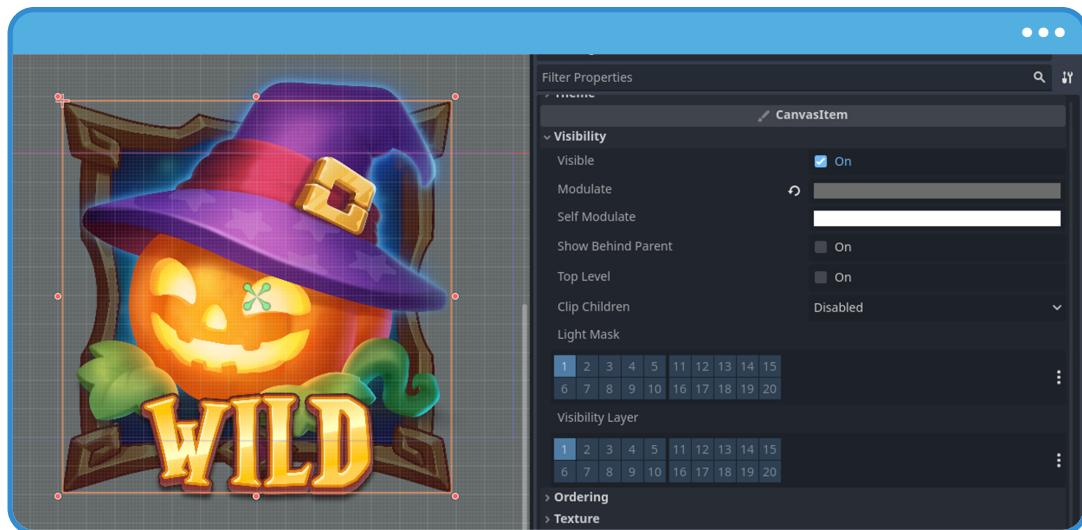
In simpler terms, it's the default vertex color (white), multiplied by the color you set in the material, and then by the **Modulate** and **Self Modulate** properties in the **Visibility** panel.

You can use this property to create a **color ID**, like so:

```
else
{
    float color_id = dot(COLOR.rgb, COLOR.rgb);
    float time = (color_id + TIME * _VertexSpeed) * PI;
    ...
}
```

Here, you declare a new variable called **color\_id**, which is the dot product of **COLOR.rgb** with itself. This produces a grayscale value that works perfectly as an ID.

If you go to the **Visibility** panel of the material and adjust the grayscale value of the **Modulate** property on any of your UI images, you'll see a clear phase shift in the animation. However, this introduces a new problem: the image now looks darker or lighter depending on the color value of the property.

(3.4.i The vertex color is being affected by the **Modulate** property)

You can solve this problem easily by passing the texture color into the **COLOR** variable in the fragment stage:

```
void fragment()
{
    vec4 color = texture(TEXTURE, UV);
    COLOR = color;
}
```

As expected, the **TEXTURE** variable refers to the default 2D texture assigned to the UI image, which will vary depending on the element.



(3.4.j The color in the UI images has been fixed)

In the next section, you'll add a custom specular effect to further enhance the animation of your UI images.

### 3.5 Adding a Specular Effect to the UI.

It's true that a specular effect on an image — whether it's a Sprite or a UI element — could easily be created using a simple texture, such as a vertical gradient. Doing so would save time and resources, since GPUs are highly optimized for these types of operations. However, from a technical perspective, it's much more interesting to generate this texture procedurally. This approach allows you to explore more advanced areas, such as implicit distance functions (SDFs).

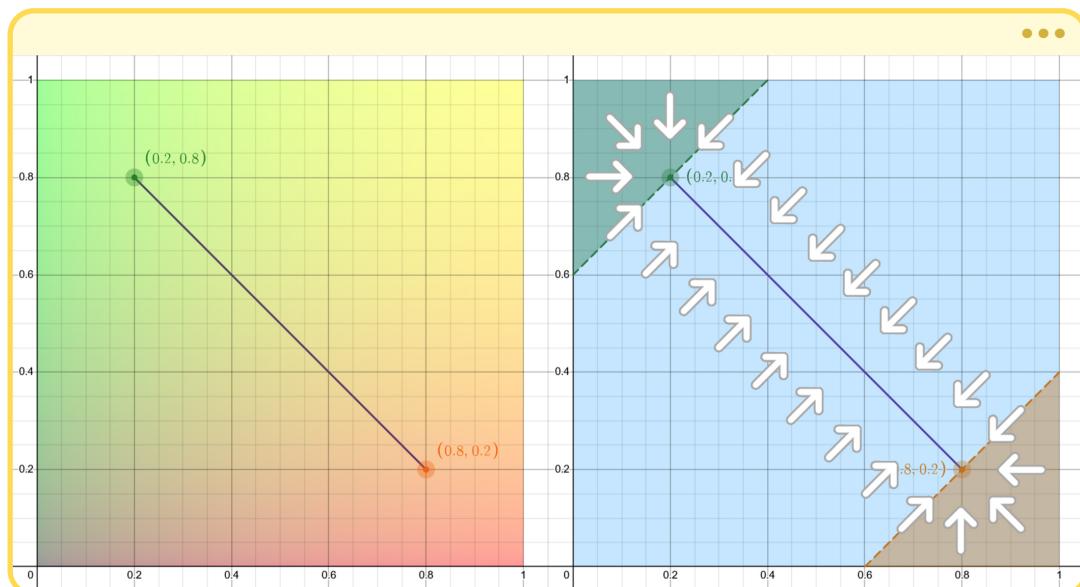
For this reason, in this section, you'll develop a segment using the implicit distance technique, which will require applying your knowledge of trigonometry.

An SDF segment looks as follows:

```
float segment_sd(vec2 uv, vec2 p0, vec2 p1)
{
    float h = projection(uv, p0, p1);
    h = clamp(h, 0.0, 1.0);
    vec2 p2 = p0 + h * (p1 - p0);
    return distance(p2, uv);
}
```

If you're just getting started with GDSL, this function might look complex at first. However, it becomes much easier to understand once you recall the definition of a segment in the Cartesian plane.

Let's start by plotting two points on the Cartesian plane:



From Figure 3.5.a, there are a few concepts you should pay close attention to:

- ① The points were placed within the range  $[0.0 : 1,1]$ , matching the default range of UV coordinates.
- ② When these points are positioned on the Cartesian plane, three distinct regions are naturally formed, shown in green, light blue, and orange.

The arrows on the right side of the image represent the distance between a pixel (or position) and the segment formed by connecting the two points. Now, how do you connect these points to form that segment? The answer lies in **projection**, which mathematically looks like this:

$$Proj = \frac{(a - b) \cdot (c - b)}{(\sqrt{c - b} \cdot c - b)^2}$$

(3.5.b)

You can translate this function into GDScript as follows:

```
float projection(vec2 a, vec2 b, vec2 c)
{
    vec2 cb = c - b;
    vec2 ab = a - b;
    return dot(ab, cb) / dot(cb, cb);
}
```

Now you just need to calculate the segment itself. To do this, you'll determine the distance between the pixels bordering the segment and the segment itself:

```
float segment_sd(vec2 uv, vec2 p0, vec2 p1)
{
    // 1
    float h = projection(uv, p0, p1);
    // 2
    h = clamp(h, 0.0, 1.0);
    // 3
    vec2 p2 = p0 + h * (p1 - p0);
    // 4
    return distance(p2, uv);
}
```

Let's analyze what happens in each line:

- 1 We define the projection of each pixel between points **p0** and **p1**.
- 2 We limit the projection so it only spans the segment between **p0** and **p1** — otherwise, the segment would extend infinitely.
- 3 We compute the position along the segment using a linear combination (similar to the form  $mx + b$ ).
- 4 We calculate the distance between each point (**uv**) and the closest point **p2** on the segment.

Since you'll use these functions inside the fragment stage, you can include them between the **vertex()** and **fragment()** methods, as shown below:

```
...
float projection(vec2 a, vec2 b, vec2 c)
{
    vec2 cb = c - b;
    vec2 ab = a - b;
    return dot(ab, cb) / dot(cb, cb);
}

float segment_sd(vec2 uv, vec2 p0, vec2 p1)
{
    float h = projection(uv, p0, p1);
    h = clamp(h, 0.0, 1.0);
    vec2 p2 = p0 + h * (p1 - p0);
    return distance(p2, uv);
}

void fragment()
{
    vec4 color = texture(TEXTURE, UV);
    COLOR = color;
}
```

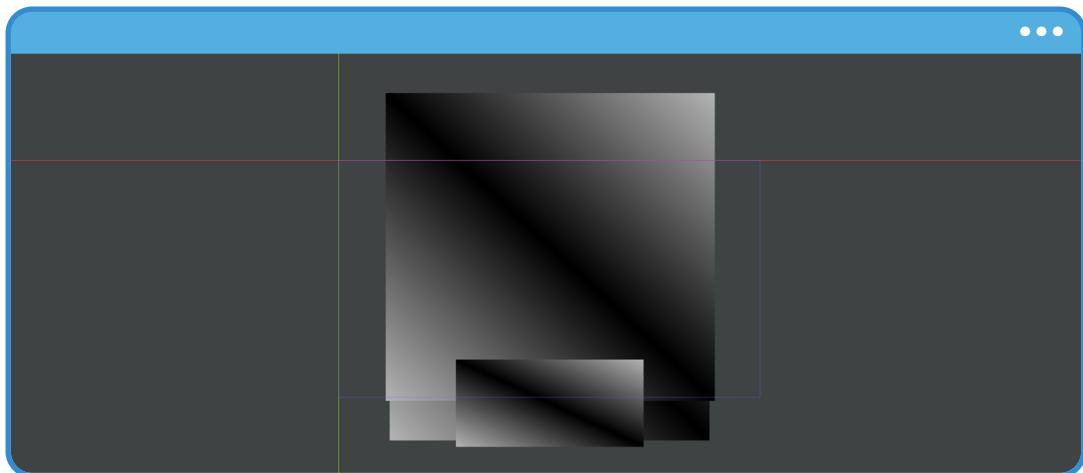
If you look closely at the arguments of `segment_sd()`, you'll notice it takes the UV coordinates along with points `p0` and `p1`. Therefore, you can start by declaring and initializing these values inside the `fragment()` method. However, keep in mind that the segment shown in Figure 3.5.a was calculated using Cartesian coordinates, where the origin is located at the bottom-left. In GDScript, the origin is at the top-left, so you'll need to invert the V coordinate of the UV when initializing the segment.

```
void fragment()
{
    vec4 color = texture(TEXTURE, UV);
    // 1
    vec2 p0 = vec2(1.0, 0.0);
    vec2 p1 = vec2(0.0, 1.0);
    // 2
    vec2 uv = vec2(UV.x, 1.0 - UV.y);
    // 3
    float specular = segment_sd(uv, p0, p1);
    // 4
    COLOR = vec4(vec3(specular), 1.0);
}
```

As you can see:

- ① We declare points `p0` and `p1` using constant values for visualization.
- ② We invert the V coordinate of the UV before using them in the segment.
- ③ We compute the specular effect from the segment.
- ④ We assign the result to the main color, making it visible in the Viewport.

This block of code produces the following visual result:



(3.5.c A gradient generated from the segment)

The gradient appears darkest at the center because of the distance between the segment and the surrounding pixels. Pixels closest to the segment approach a value near zero, so the resulting color is darker. As the distance increases, the value rises and the color becomes lighter.

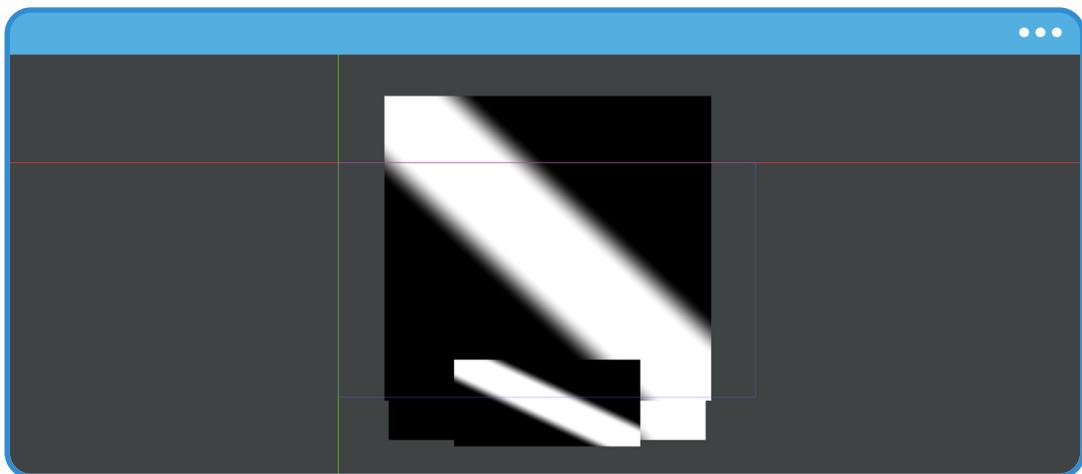
You now need the segment to be white at the center with a well-defined shape. To achieve this, use `smoothstep()`, which produces a smooth transition between two thresholds within a specified range. It evaluates an input value and returns a result interpolated between 0 and 1 – where 0 corresponds to the start of the range and 1 to the end. This behavior lets you precisely control where the color transition begins and ends, clearly defining the segment's edges and concentrating white at the center.

```
void fragment()
{
    ...
    float specular = segment_sd(uv, p0, p1);
    specular = smoothstep(0.2, 0.1, specular);

    COLOR = vec4(vec3(specular), 1.0);
}
```

**Note**

The constant values used above were tested to produce the result shown in Figure 3.5.d. You're encouraged to experiment with the thresholds to better understand the effect's behavior and to achieve different looks.



(3.5.d The specular body has been defined)

You could move the points manually and apply the effect to your UI images. However, to represent a proper specular highlight, the bright band should move automatically – that is, it should be driven by **TIME**.

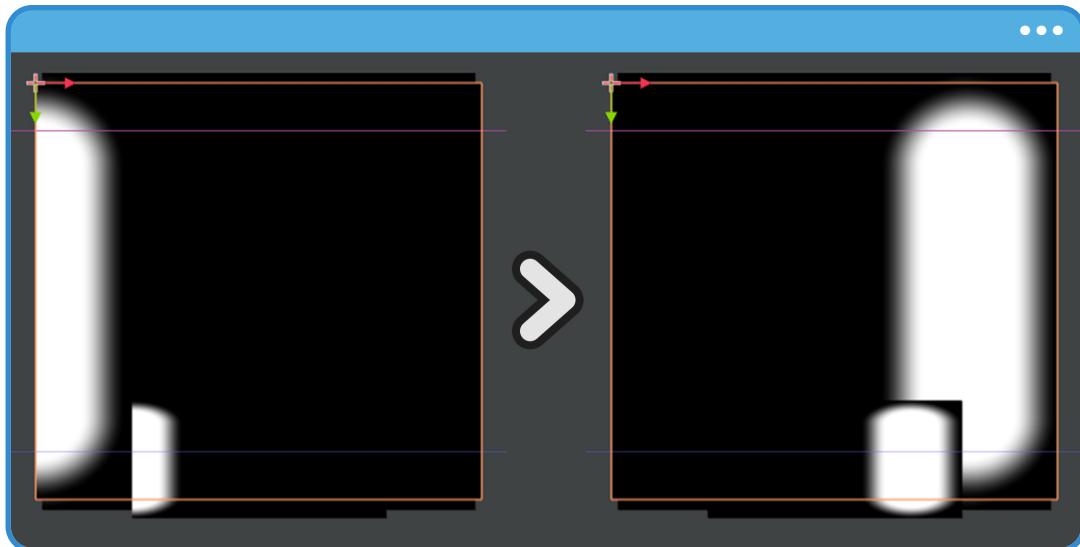
One simple way to animate the highlight is to repeat the effect on a loop: the band sweeps across the UI image, fades out, then restarts. In this example, the motion runs horizontally from left to right.

```
void fragment()
{
    ...
    // 1
    float time = TIME / 2.0;
    // 2
    float offset = fract(time) * 3.0 - 1.5;
    // 3
    vec2 p0 = vec2(0.5 + offset, 0.8);
    vec2 p1 = vec2(0.5 + offset, 0.2);

    ...
}
```

Here's what each step does:

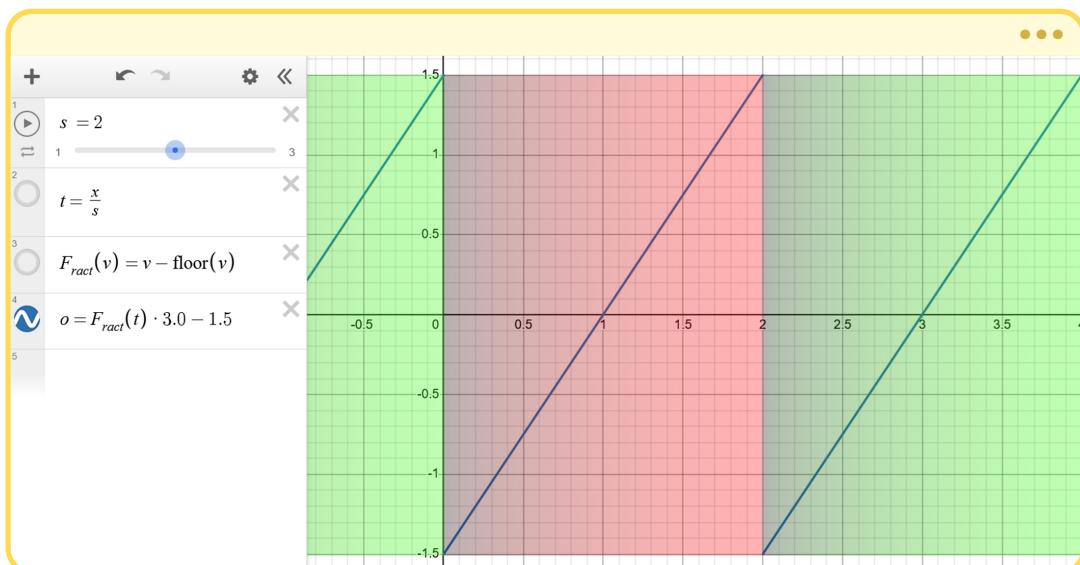
- ➊ The **time** variable results from dividing **TIME** by 2.0, which implies that the effect will repeat every two seconds. (Later, you can replace 2.0 with a material property like **\_SpecularSpeed** to make it adjustable.)
- ➋ You take the fractional part of time using **fract(time)**. When time reaches 0.99, it wraps back to 0.0. Multiplying by 3.0 and subtracting 1.5 maps the result to [-1.5 : 1.5], producing a cyclic horizontal **offset**.
- ➌ You add **offset** to the X component of both points to move the segment horizontally, and you set their Y components to frame the band vertically. The segment now enters and exits the UI image in a continuous loop.



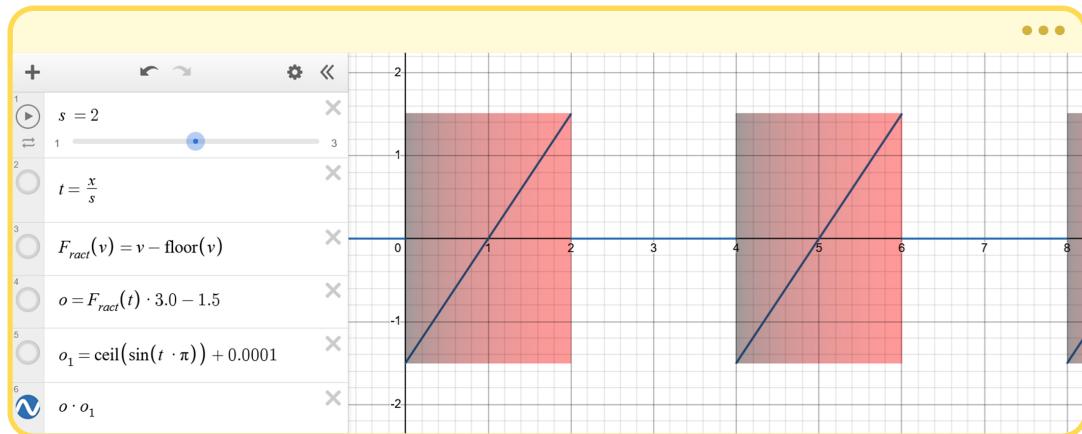
(3.5.e The segment moves from left to right)

If you play the effect for an extended period, you may notice the highlight repeating too frequently. To make the motion feel more natural, you can limit its frequency. Before doing so, it helps to visualize the function you're using. Using Desmos or a similar graphing tool, plot the following expression on the Cartesian plane to understand how it behaves:

**fract(time) \* 3.0 - 1.5.**

(3.5.f <https://www.desmos.com/calculator/4o5dww2dxf>)

As shown in Figure 3.5.f, two colors have been used to analyze the behaviour of the segment's animation, which remains constant over time. The cycles repeat continuously with no pause. To limit how often the highlight appears, multiply the horizontal offset by the function `ceil(sin(time * PI)) + 0.0001`, as shown next.



(3.5.g <https://www.desmos.com/calculator/a3konqs6nt>)

This makes every other cycle equal to zero, effectively throttling the effect's frequency.

```
void fragment()
{
    ...
    float time = TIME / _SpecularSpeed;
    float offset = fract(time) * 3.0 - 1.5;
    // 1
    float offset_mask = ceil(sin(time * PI)) + 0.00001;
    // 2
    offset *= offset_mask;

    vec2 p0 = vec2(0.5 + offset, 0.8);
    vec2 p1 = vec2(0.5 + offset, 0.2);

    ...
    specular = smoothstep(0.2, 0.1, specular);

    // The code continues on the next page.
}
```

```
// 3
specular *= offset_mask;

...
}
```

What's happening here:

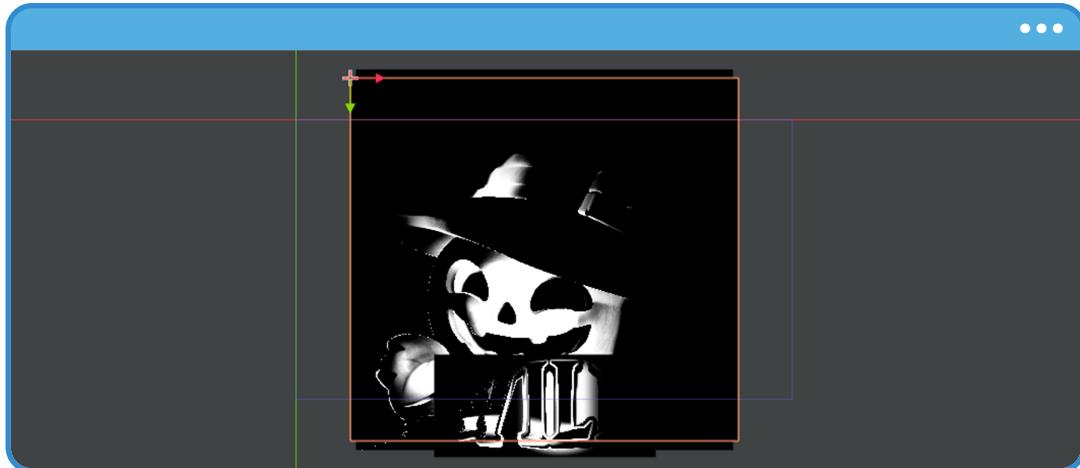
- ➊ We declare `offset_mask = ceil(sin(time * PI)) + 0.00001`, which alternates between 1.0 and 0.0.
- ➋ We multiply offset by `offset_mask` so the segment stops moving every other cycle.
- ➌ We also multiply specular by `offset_mask` to ensure the band isn't visible when it isn't moving.

To make the highlight feel like a true specular reflection and to blend it with the UI image, you'll now distort the segment with the base color and then tint and add it to the final RGB:

- Distort the segment's UVs using one component of the base color vector.
- Convert the scalar specular value to RGB so you can tint it.
- Add the tinted highlight to the base color.

If, for example, you use the B component to distort the UVs, you get:

```
void fragment()
{
    ...
    vec2 uv = vec2(UV.x, 1.0 - UV.y);
    uv += color.b;
    ...
}
```



(3.5.h The specular highlight adopts the UI image's shape)

Remember that UV coordinates are linear. When you offset them with a color channel, that linearity is perturbed and the highlight's shape adapts to the values you inject – in this case, the B component.

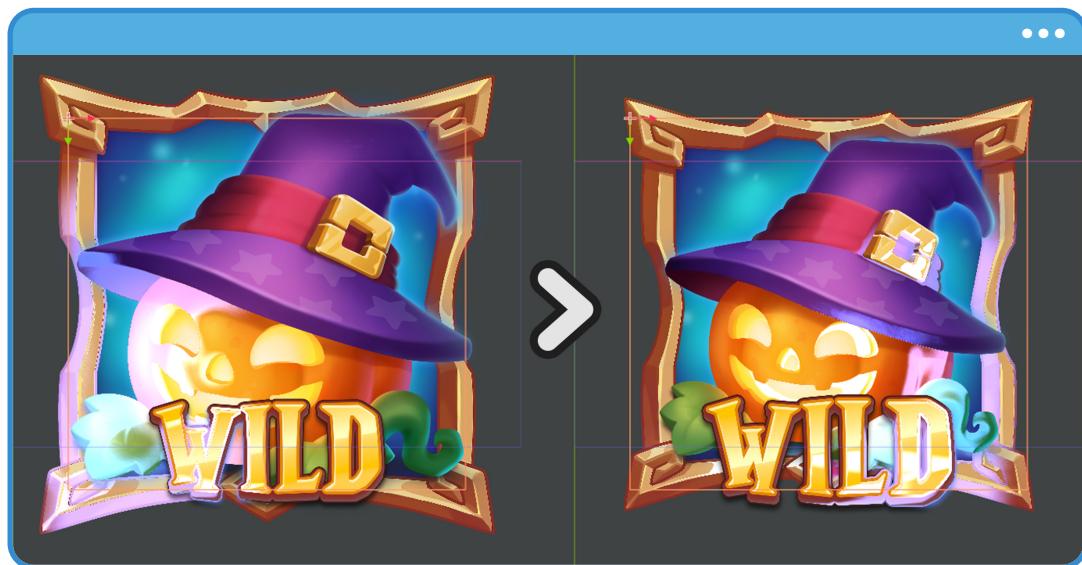
The specular term is a scalar. To colorize it, convert it to a 3D vector (RGB) and multiply by your chosen color:

```
void fragment()
{
    ...
    float specular = segment_sd(uv, p0, p1);
    specular = smoothstep(0.2, 0.1, specular);
    specular *= offset_mask;
    // 1
    vec3 specular_color = vec3(specular) * _SpecularColor;
    // 2
    color.rgb += specular_color;
    // 3
    color.rgb = clamp(color.rgb, 0.0, 1.0);
    // 4
    COLOR = color;
}
```

From this code:

- 1 We create `specular_color` by multiplying the scalar highlight by `_SpecularColor`.
- 2 We add the result to the base color's RGB.
- 3 We clamp the color to prevent values from exceeding 1.0.
- 4 We write the final color to `COLOR`.

This produces the visual result shown in Figure 3.5.i, where the specular effect is now applied to the UI image.



(3.5.i The specular effect has been applied to the image)

### 3.6 Introduction to Rotations.

When discussing rotations in computer graphics, we usually refer to two fundamental mathematical representations: matrices and quaternions. In Section 1.10, you were introduced to the concept of matrices and used them to implement a billboard effect in a shader. However, we haven't yet explored how these structures can be applied directly to produce rotations in three-dimensional space.

Remember that matrices are simply organized arrays of values that transform vectors when multiplied by them. In the case of rotations, these values involve sine and cosine

functions positioned in specific parts of the matrix. The placement of these trigonometric terms determines the axis of rotation.

For example, a rotation around the  $x$ -axis modifies the  $yz$  coordinates, while a rotation around the  $y$ -axis affects  $xz$ , and a rotation around the  $z$ -axis alters  $xy$ . This structure allows you to build rotation matrices for each individual axis or combine them to achieve more complex rotations.

Mathematically, rotation matrices in three-dimensional Euclidean space are defined as follows:

For the  $x$ -axis (Phi):

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

(3.6.a)

For the  $y$ -axis (Theta):

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

(3.6.b)

For the  $z$ -axis (Psi):

$$R_z(\psi) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(3.6.c)

These matrices produce counterclockwise rotations when observed following the right-hand rule, where the thumb points in the positive direction of the rotation axis.

Even though these expressions may appear complex at first, their implementation in GDSL is surprisingly straightforward:

```
// x-axis rotation
#define RX(a) mat3(
    vec3(1, 0, 0), \
    vec3(0, cos(a), -sin(a)), \
    vec3(0, sin(a), cos(a)))

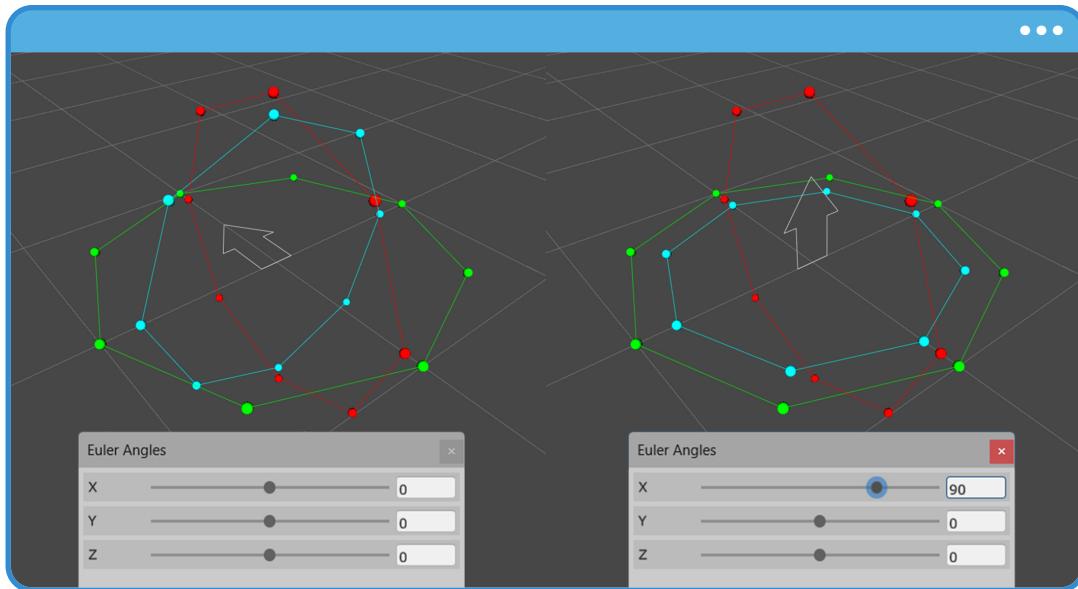
// y-axis rotation
#define RY(a) mat3(
    vec3(cos(a), 0, sin(a)), \
    vec3(0, 1, 0), \
    vec3(-sin(a), 0, cos(a)))

// z-axis rotation
#define RZ(a) mat3(
    vec3(cos(a), -sin(a), 0), \
    vec3(sin(a), cos(a), 0), \
    vec3(0, 0, 1))
```

Now, an important question arises: why are there two different ways to represent rotations – matrices and quaternions – if both serve the same purpose? The answer lies in a phenomenon known as Gimbal Lock.

This effect occurs when you use **Euler** angles for rotation – that is, when you apply sequential rotations around the *xyz* axes using matrices. The problem is that rotations are not commutative: the order in which you multiply the matrices matters. Under certain configurations, two of the three rotation axes can become aligned, reducing the system's original three degrees of freedom to only two.

This phenomenon, called a singularity, causes the object to lose the ability to rotate freely in all directions, resulting in unpredictable or erratic behavior in the animation or transformation.



(3.6.d On the left: three concentric rings; on the right: the Gimbal effect)

This is where quaternions come into play. Unlike traditional rotation matrices, quaternions represent rotations in a four-dimensional space, allowing them to completely avoid the Gimbal Lock singularity and maintain stable behavior regardless of orientation.

A quaternion can be expressed as:

$$Q = w + xi + yj + zk$$

(3.6.e)

Where  $wxyz$  are real numbers, and  $ijk$  are imaginary units that satisfy the relationships:

$$i^2 = j^2 = k^2 = ijk = -1$$

(3.6.f)

In the context of rotations, a quaternion corresponding to a unit rotation axis  $\mathbf{u} = (u_x, u_y, u_z)$  can be written as:

$$Q = \left( \cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right) \mathbf{u} \right)$$

(3.6.g)

Here,  $\mathbf{u}$  represents a unit vector that defines the axis of rotation, and  $\theta$  is the rotation angle in radians. This formulation allows you to describe any three-dimensional rotation without relying on Euler angles, thus eliminating the problem of axis alignment.

In the following sections, you'll implement rotations using both matrices and quaternions, applying them to geometric primitives to create different visual effects and gain a deeper understanding of how they behave in practice.

### 3.7 Implementing a Rotation Matrix.

In this section, you'll tackle two main exercises. First, you'll work with Euler angles to implement a three-dimensional rotation on a cube. Then, using the three rotation matrices from the previous section, you'll extend the **world\_space** shader introduced in Section 1.6 so the tree not only changes color based on position but also rotates around its axes.

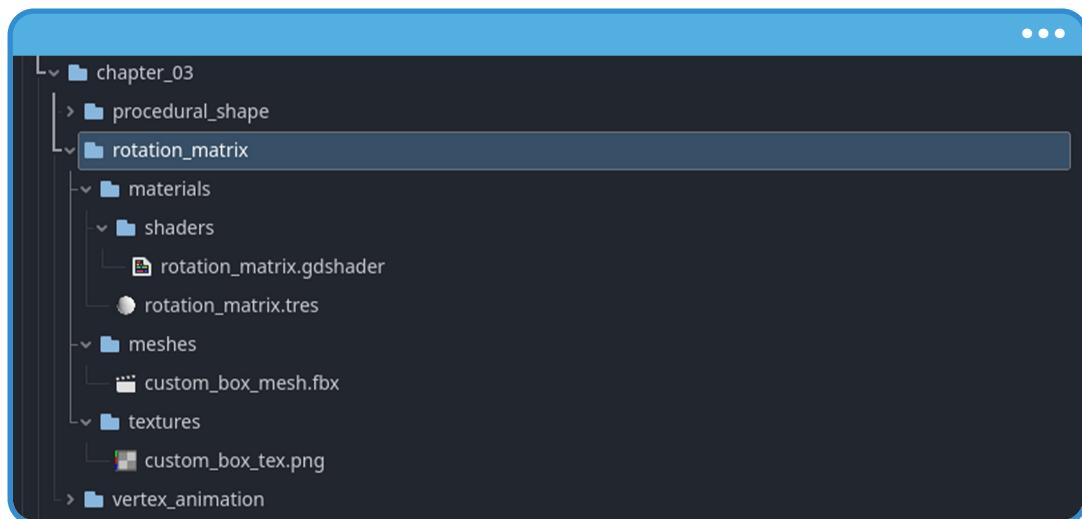
To begin, go to your project and, inside **Chapter 3**, create a new folder named **rotation\_matrix**.

At this point in the book, you're already familiar with the project structure — creating materials, organizing folders, and authoring shaders — so we'll skip the granular setup steps and focus on implementation.

For the first exercise, create a new **ShaderMaterial** and a corresponding **Shader**, both named **rotation\_matrix**. Following the same organization you've used in earlier chapters, add any necessary subfolders before you start building the effect.

For this chapter, you'll use a custom 3D model: a **BoxMesh** named **custom\_box\_mesh**. The **.fbx** file and its texture are provided in the downloadable project files that accompany this book, under the **rotation\_matrix** folder.

If you've followed the steps so far, your project should now include the following assets:



(3.7.a A new folder has been added to the project)

**Note**

To run this exercise, you need to assign the shader to a material and then apply that material to **custom\_box\_mesh** via the **Material Override** property.

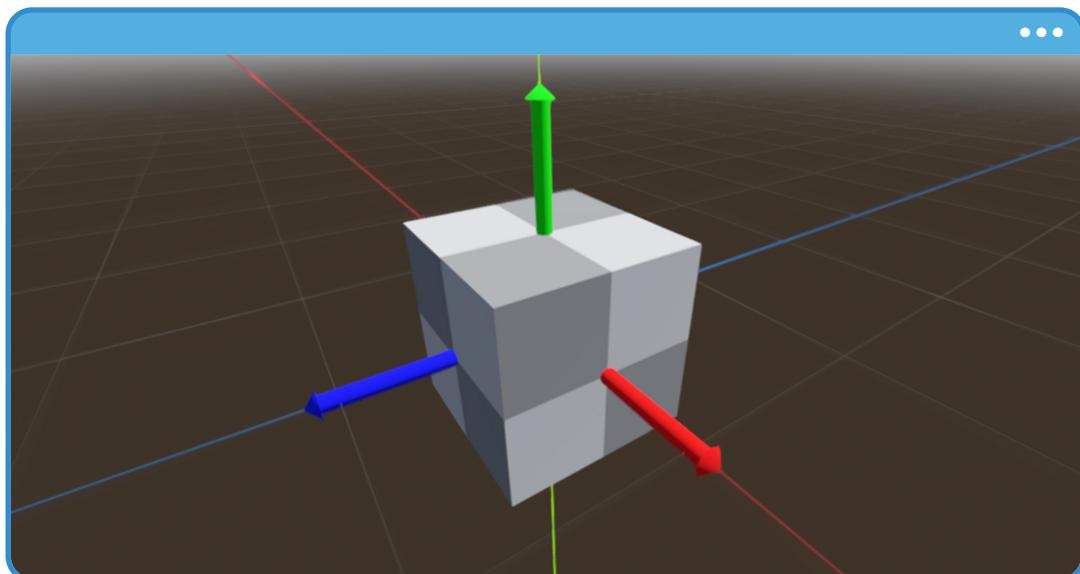
Your first step inside the **rotation\_matrix** shader is to set up a **base map** so you can visualize the albedo color on the custom cube in the Scene window. This helps you confirm the material is wired correctly before you add rotations, and it also makes the model's 3D axes easier to read.

```
shader_type spatial;

uniform sampler2D _BaseMap : source_color;

void vertex() { ... }

void fragment()
{
    vec3 albedo = texture(_BaseMap, UV).rgb;
    ALBEDO = albedo;
}
```



(3.7.b The texture has been assigned to the custom cube)

Next, you'll implement the rotation matrices for each axis. Use the layouts from Figures 3.6.a, 3.6.b, and 3.6.c as your guide.



```

uniform sampler2D _BaseMap : source_color;
// 1
uniform vec3 _Angles;

// 2
#define RX(a) mat3(           \
    vec3(1, 0, 0),          \
    vec3(0, cos(a), -sin(a)), \
    vec3(0, sin(a), cos(a)))

// 3
#define RY(a) mat3(           \
    vec3(cos(a), 0, sin(a)), \
    vec3(0, 1, 0),          \
    vec3(-sin(a), 0, cos(a)))

// 4
#define RZ(a) mat3(           \
    vec3(cos(a), -sin(a), 0), \
    vec3(sin(a), cos(a), 0), \
    vec3(0, 0, 1))

void vertex() { ... }

```

What this does:

- ➊ You declare a `vec3 _Angles` to define the rotation angle for each axis. You'll tweak these live from the **Inspector**.
- ➋ `RX(a)` rotates around *x*-axis.
- ➌ `RY(a)` rotates around *y*-axis.
- ➍ `RZ(a)` rotates around *z*-axis.

With the three matrices defined, the next step is to apply them to the mesh in the `vertex()` function using an Euler composition:

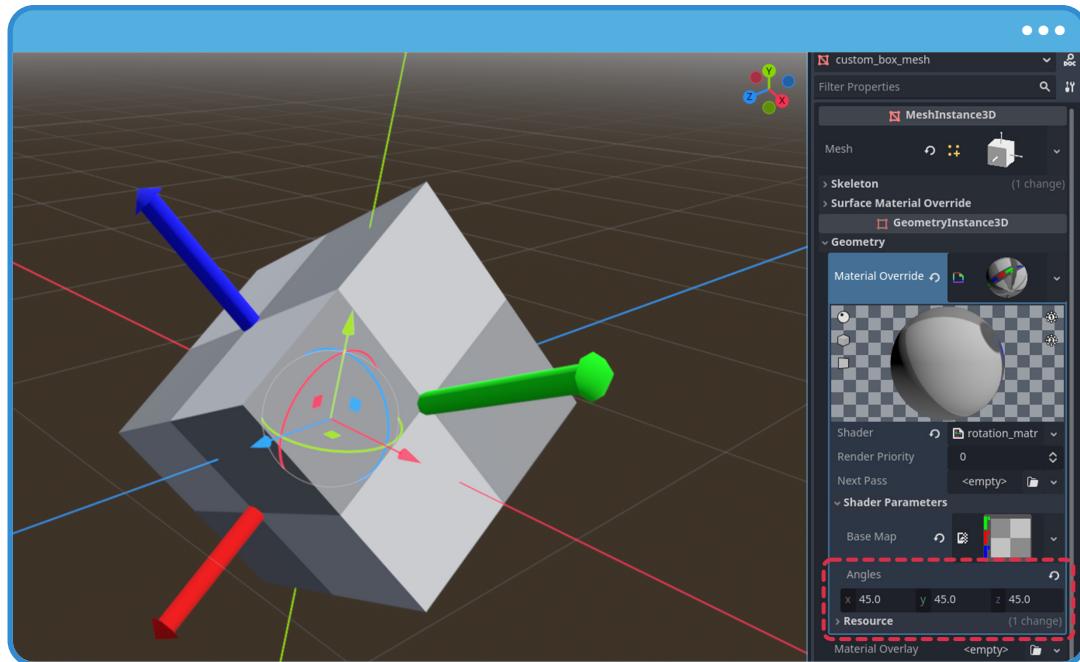


```
void vertex()
{
    // 1
    vec3 angles = _Angles * (PI / 180.0);
    // 2
    mat3 EULER_ZXY = RY(angles.y) * RX(angles.x) * RZ(angles.z);
    // 3
    VERTEX = EULER_ZXY * VERTEX;
    // 4
    NORMAL = normalize(EULER_ZXY * NORMAL);
}
```

Let's see what's happening here:

- ➊ We create a new vector called **angles**, which converts the **\_Angles** values from degrees to radians (the trigonometric functions expect radians).
- ➋ We declare the **EULER\_ZXY** matrix, which applies the rotations in the order **Z → X → Y**. This order is not arbitrary — when using Euler, the result depends directly on the order of the matrix multiplication. Here, we start rotating around the *z*-axis, then the *x*-axis, and finally on the *y*-axis.
- ➌ We apply the composed matrix to the mesh vertices in object space.
- ➍ We rotate and normalize the normals so the lighting matches the new orientation. This step is essential — if you skip it, the lighting will no longer align with the object's new orientation in space.

If you return to the Inspector and modify the **\_Angles** values, you'll see the custom cube begin to rotate in real time, following the transformations defined by the Euler matrices.



(3.7.c The cube has been rotated 45° on each axis)

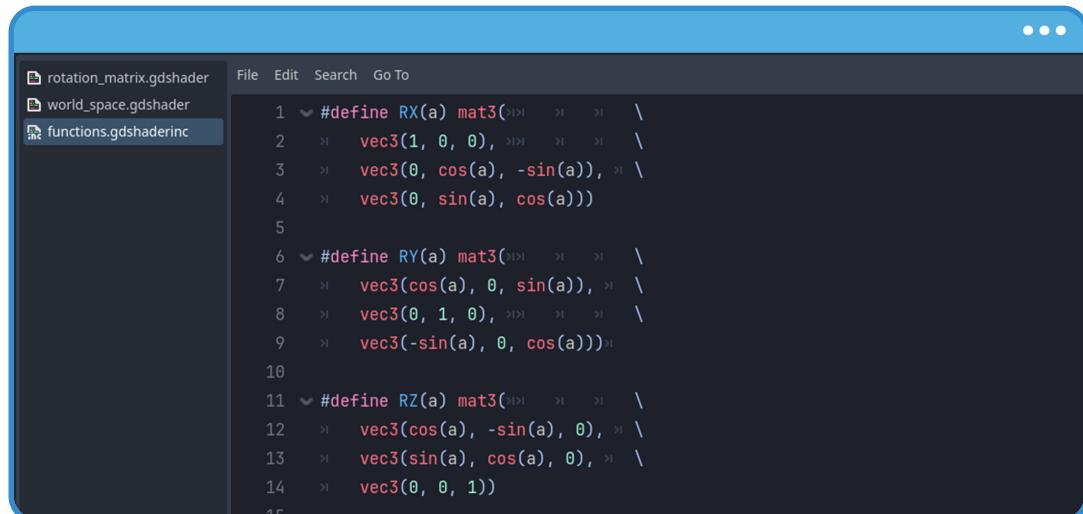
If you rotate the cube 45° around each axis, you'll notice that the rotation system works correctly in most cases. However, this implementation has one important limitation. When you rotate the cube 90° around the x-axis and then try to modify the y and z angles, the object loses a degree of freedom – both axes seem to rotate the cube in the same direction. This behavior is known as the Gimbal Lock effect.

**Note**

In Godot, you can configure the rotation mode directly from a node's **Transform > Rotation Edit Mode** property. By default, the editor uses **Euler** mode, but you can switch it to **Quaternion**, which prevents Gimbal Lock by representing cumulative rotations through a different mathematical structure.

Even with this limitation, the rotation matrices you implemented remain extremely useful. You can apply them individually or in combination to create dynamic effects inside the shader. In fact, you'll now reuse these same matrices to extend the **world\_space** shader developed earlier. Since we'll be using the same matrix definitions, the first step is to centralize them within the project.

To do this, open the **functions.gdshaderinc** file created in Chapter 2 and copy the three matrices (**RX**, **RY**, and **RZ**) there. This will allow you to access them from any shader in the project without redefining them each time.



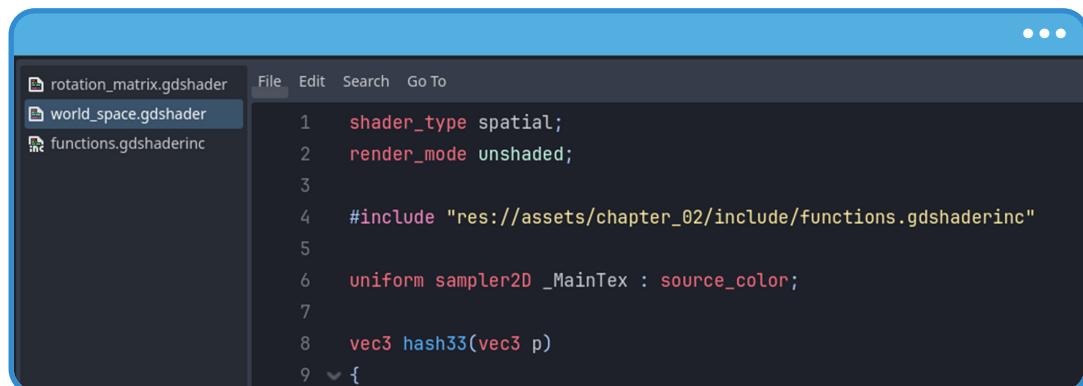
```

1 #define RX(a) mat3( \
2     vec3(1, 0, 0), \
3     vec3(0, cos(a), -sin(a)), \
4     vec3(0, sin(a), cos(a))) \
5 
6 #define RY(a) mat3( \
7     vec3(cos(a), 0, sin(a)), \
8     vec3(0, 1, 0), \
9     vec3(-sin(a), 0, cos(a))) \
10 
11 #define RZ(a) mat3( \
12     vec3(cos(a), -sin(a), 0), \
13     vec3(sin(a), cos(a), 0), \
14     vec3(0, 0, 1)) \
15

```

(3.7.d The matrices have been added to **functions.gdshaderinc**)

Next, go to your **world\_space** shader and include a reference to this shared file, as shown in the following figure:



```

1 shader_type spatial;
2 render_mode unshaded;
3 
4 #include "res://assets/chapter_02/include/functions.gdshaderinc"
5 
6 uniform sampler2D _MainTex : source_color;
7 
8 vec3 hash33(vec3 p)
9 {

```

(3.7.e The matrices has been included in **world\_space** via functions)

The final step is to make each tree in the scene rotate dynamically, so that every instance has its own unique orientation. To achieve this, you can use the **hash33()** function together with the **NODE\_POSITION\_WORLD** variable. This combination generates a pseudo-random value based on the object's world position and converts it into a rotation angle in radians. As

a result, each tree instance will adopt a slightly different rotation depending on its location in the Viewport, eliminating repetitive patterns and giving the environment a more natural and organic look.

```

void vertex()
{
    // 1
    vec3 vertex_os = VERTEX;
    // 2
    vec3 vertex_ws = (MODEL_MATRIX * vec4(vertex_os, 1.0)).xyz;
    // 3
    float gradient_ws = clamp(0.0, 1.0, vertex_ws.y - NODE_POSITION_WORLD.y);
    // 4
    float angle_map = hash33(floor(NODE_POSITION_WORLD)).y;
    angle_map *= gradient_ws;
    // 5
    float st = (sin(TIME + angle_map * 2.0) * 0.05 + 0.05);
    // 6
    float ct = (cos(TIME + angle_map * 2.0) * 0.05 + 0.05);
    // 7
    mat3 EULER_ZXY =
        RY(angle_map * TAU) *
        RX((angle_map * 0.1345) - st) *
        RZ((angle_map * 0.1567) - ct);
    // 8
    VERTEX = EULER_ZXY * vertex_os * clamp(angle_map, 0.4, 1.0);
}

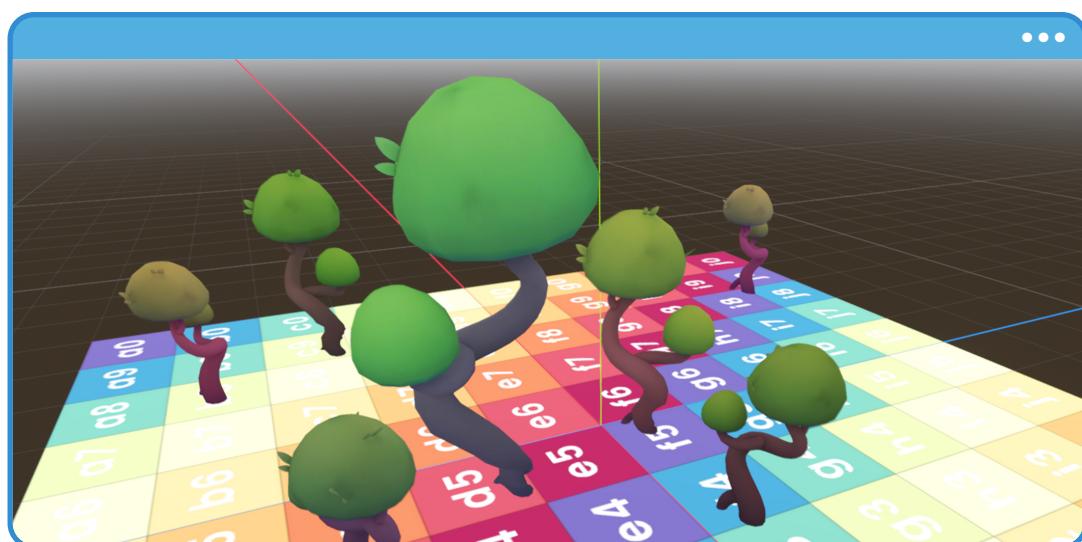
```

Let's analyze what's happening step-by-step:

- 1 We initialize **vertex\_os** with the vertices in object space. This is your starting point before applying any transformations.
- 2 We transform the vertices to world space using **MODEL\_MATRIX**, which lets you relate their position to the world's absolute coordinates.
- 3 We build a vertical gradient by subtracting the node's **NODE\_POSITION\_WORLD.y** from the vertex **vertex\_ws.y**, then clamping the result to [0 : 1]. Values near the ground approach 0, while higher points approach 1.

- 4 We generate a pseudo-random value called `angle_map` from the object's world position with `hash33()`. You multiply it by `gradient_ws` so the base of each tree stays stable, avoiding root deformation.
- 5 We add a sine oscillation with `sin(TIME)` to introduce a smooth, periodic sway.
- 6 We add a complementary cosine oscillation to create a temporal offset and more organic motion.
- 7 We construct the `EULER_ZXY` rotation matrix, combining rotations about all three axes using the random and oscillatory terms. The result is a gentle, continuous variation in each tree's orientation – mimicking wind.
- 8 We apply the rotation matrix to the local vertices and scale them with `clamp(angle_map, 0.4, 1.0)`, adding subtle size differences between trees to reinforce a natural look.

Back in the Viewport, you'll see that trees now change color, rotate, and scale dynamically based on their world position, producing a more organic landscape.



(3.7.f Trees rotate and scale based on their world position)

**Note**

While this exercise is a practical way to combine coordinates, matrices, and math functions to create rotations and scaling, it's not recommended to run these operations directly in a production shader.

These transformations execute per-vertex, every frame, which can impact performance on high-polygon models. In real projects, you should implement the same behavior in a script that runs once at game start: compute the rotations and scales on the CPU and store them in the object's transforms. This preserves the visual result while keeping your frame time stable.

### 3.8 Quaternion Implementation.

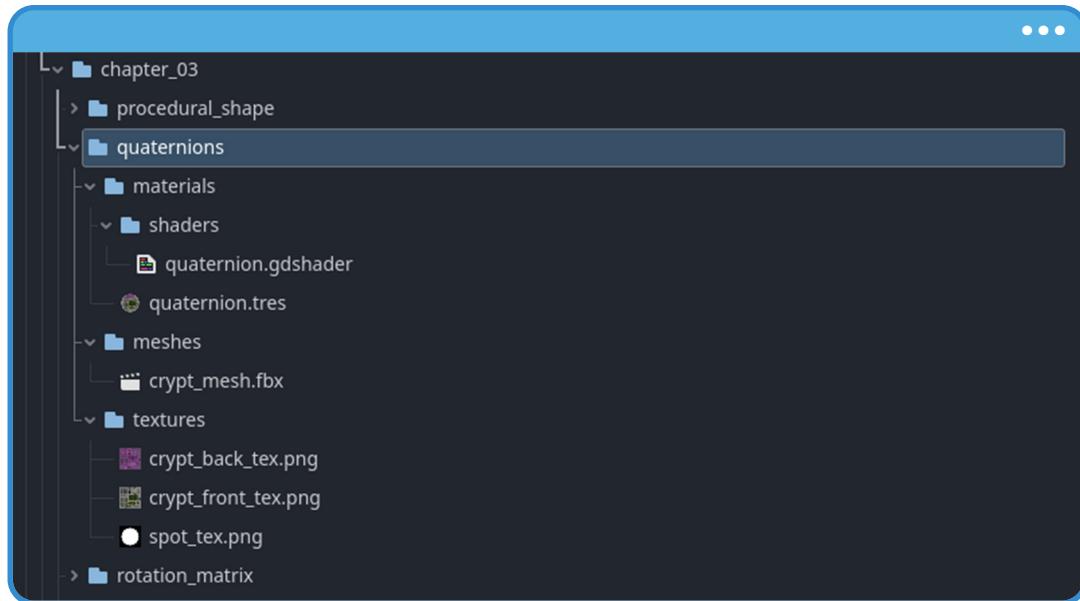
Continuing the rotation analysis, in this section you'll implement quaternions in a shader to project a texture onto a 3D model that aligns with the scene's light direction.

To begin, go to your project and, inside the **chapter\_03** folder, create a new folder named **quaternions**. To keep things organized, add the following subfolders inside it:

- **materials**
- **shaders**
- **textures**
- **meshes**

For this exercise, download the accompanying files for the book and place them in their respective folders. Then create a shader and a material dedicated to this section.

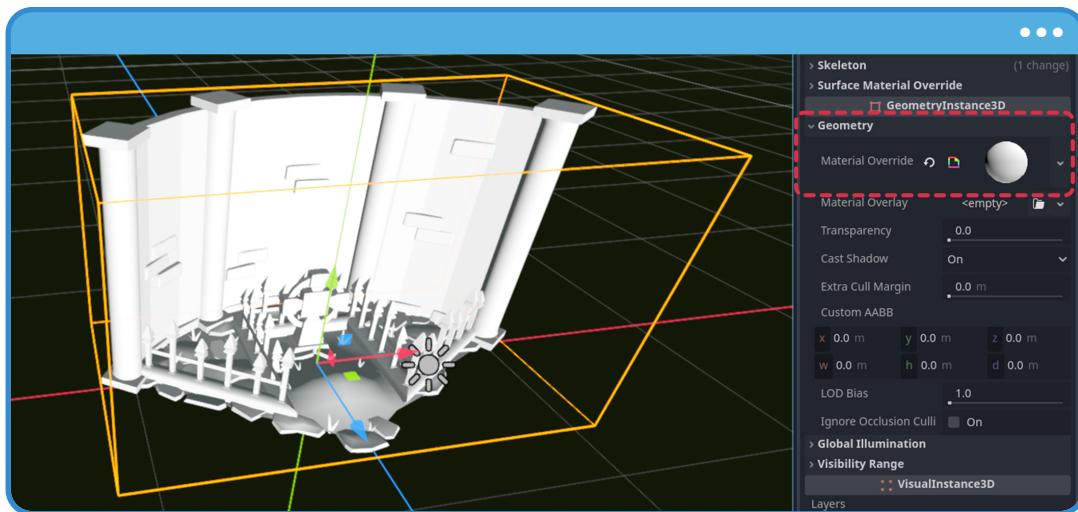
If everything is set up correctly, your project should now include the quaternions folder with its materials, shaders, textures, and meshes subfolders, plus the newly created shader and material ready for implementation.



(3.8.a The files have been added to the project)

Now set up the new scene. Start by dragging **crypt\_mesh.fbx** into the Viewport, repeating the process from Section 1.4: mark the model as **Make Unique**. Save the resulting resource inside this section's meshes folder and name it **Crypt.tres** for easy identification. This ensures the **MeshInstance3D** remains a child of the scene's **Node3D**, avoiding fragile references to child nodes from the script you'll create later.

With the **Crypt** object in the scene, assign the **quaternions** shader to its material and then set that material on the object via **Material Override**, as shown below.



(3.8.b The quaternions material has been assigned to the 3D model)

For the scene configuration, add a **DirectionalLight3D** and set both its position and rotation to [0, 0, 0] so the effect can be linked to the light accurately.

Begin by declaring the following global properties in the shader:

```
shader_type spatial;
// 1
render_mode unshaded;
// 2
uniform sampler2D _FrontMap : source_color, repeat_disable;
// 3
uniform sampler2D _BackMap : source_color, repeat_disable;
// 4
uniform sampler2D _SpotMap : source_color, repeat_disable;
// 5
uniform vec3 _LightRotation;
// 6
uniform vec3 _LightPosition;
```

As we can see:

- 1 We disable built-in lighting with unshaded so you can drive the look manually.
- 2 We will use two textures: **\_FrontMap** is the primary texture; it will cover most of the model.
- 3 **\_BackMap** is the secondary texture; you'll project it dynamically "behind" the primary one.
- 4 **\_SpotMap** acts as a mask to blend between Front and Back textures.
- 5 **\_LightRotation** will carry the current rotation of the directional light (you'll pass it from a script).
- 6 **\_LightPosition** will carry the current world-space position of the directional light (also passed from a script).

For a quick visual check, sample the textures in **fragment()** and, for now, assign **\_FrontMap** to **ALBEDO**, as shown below:

```
void fragment()
{
    vec3 front_map = texture(_FrontMap, UV).rgb;
    vec3 back_map = texture(_BackMap, UV).rgb;
    float spot_map = texture(_SpotMap, UV).r;

    ALBEDO = front_map;
}
```

Our next goal is to project **\_SpotMap** in world space using the vertex XY components so that:

- 1 The texture is projected in the world, and
- 2 it follows the default light direction.

Now, we want a world-space projection for two reasons:

- 1 The texture must follow the position and rotation of the directional light.
- 2 The texture must preserve its proportions (using mesh UVs directly could deform the circle due to the UV layout).

Here's the setup:

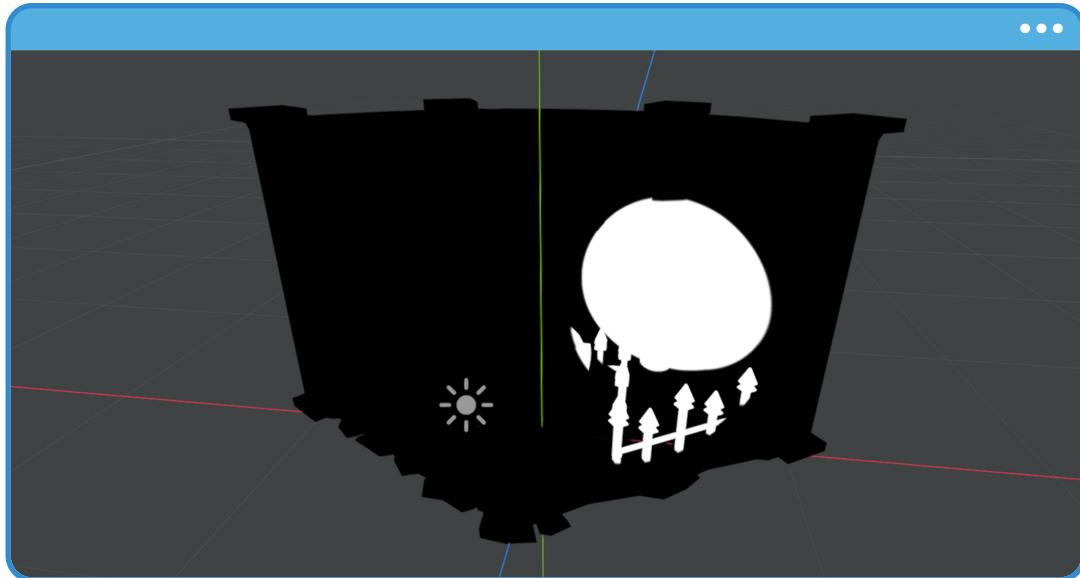
```
void fragment()
{
    // 1
    vec3 vertex_ws = (INV_VIEW_MATRIX * vec4(VERTEX, 1.0)).rgb;
    // 2
    vec3 position_ws = vertex_ws - _LightPosition;

    vec3 front_map = texture(_FrontMap, UV).rgb;
    vec3 back_map = texture(_BackMap, UV).rgb;
    // 3
    float spot_map = texture(_SpotMap, position_ws.xy).r;
    // 4
    ALBEDO = vec3(spot_map);
}
```

What's happening here?

- ➊ We transform the vertex to world space.
- ➋ We subtract the directional light's world position to get a light-relative coordinate (**position\_ws**).
- ➌ We sample **\_SpotMap** with **position\_ws.xy** instead of UV, producing a planar world-space projection.
- ➍ We output the mask as **ALBEDO** to visualize it.

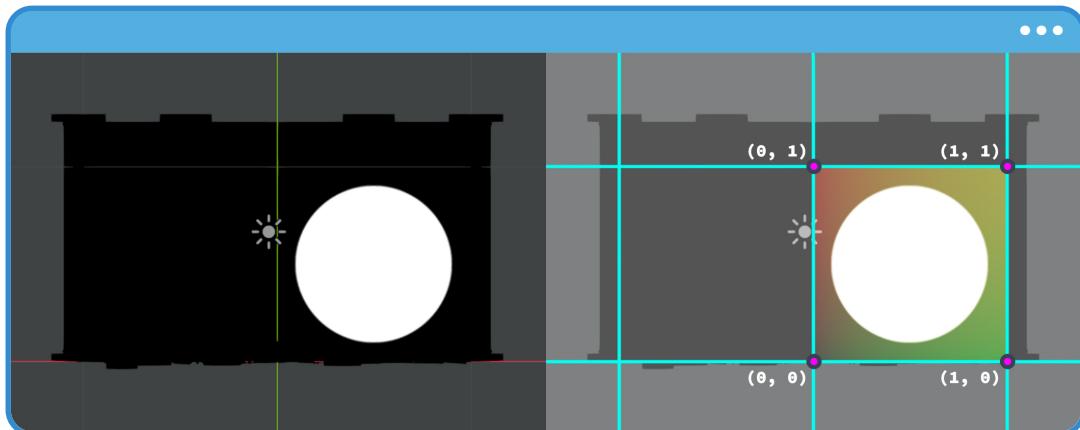
Assuming **spot\_tex.png** is already assigned to the material, you'll see the circular mask projected onto the model in the Viewport, aligned with the light's direction.



(3.8.c The circle is projected in world space)

However, the circle's position still doesn't match the light, which is currently at  $[0, 0, 0]$ . This happens because `spot_map` is sampling over the world range  $[0,0 : 1,1]$ , analogous to UV coordinates, rather than being centered on the light's origin.

Let's look at the following reference to clarify the idea.



(3.8.d Texture projection in world space)

To center the projection, you'll eventually subtract 0.5 from the XY components of **position\_ws**. Before doing that, let's first link the directional light to the shader. In the **shaders** folder, right-click and choose:

- ▶ Create New > Script.

Name the script **light\_to\_shader.gd** and add the following code:

```
// 1
@tool
extends Node
// 2
@onready var light = $DirectionalLight3D
@onready var mesh = $Crypt
@onready var shader_mat := mesh.material_override as ShaderMaterial

func _process(_delta):
    // 3
    if Engine.is_editor_hint():
        // 4
        shader_mat.set_shader_parameter("_LightPosition", light.position)
        shader_mat.set_shader_parameter("_LightRotation", light.rotation)
```

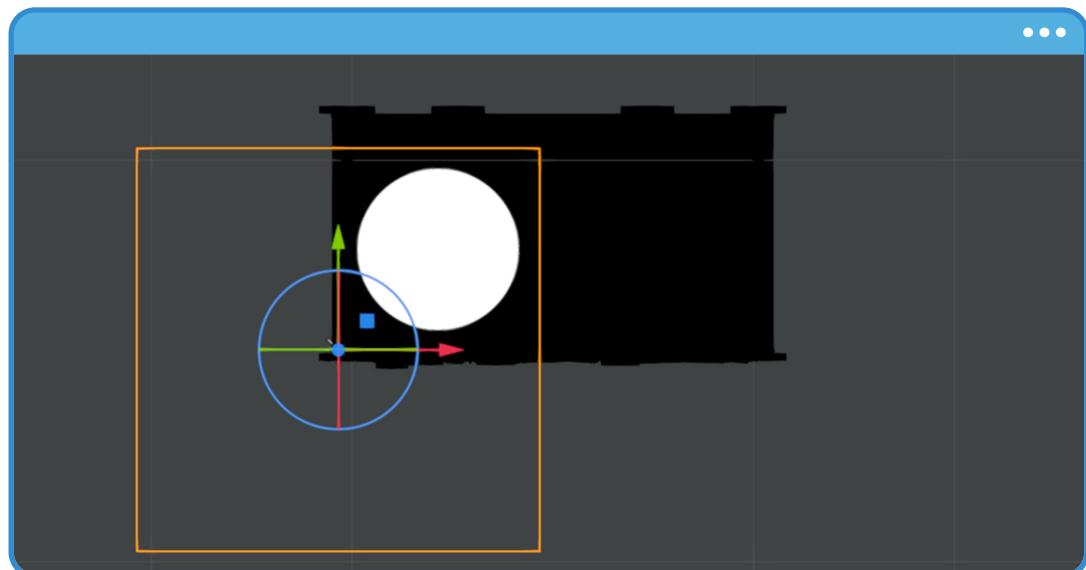
What this does:

- ➊ We run the script in the **Editor** so you can see changes in real time.
- ➋ We reference the **Scene** nodes we need: the light and the mesh, and the mesh's **ShaderMaterial**.
- ➌ We execute the update only in the editor, keeping runtime clean for this exercise.
- ➍ We push the light's position and rotation into the shader's **\_LightPosition** and **\_LightRotation** uniforms.

**Note**

This sample code isn't optimized – it runs every frame even if the light doesn't move. It's perfectly fine for the exercise. In production, you'd update shader parameters only when the light's transform changes.

Head back to the Viewport. When you move the light, the mask follows it. It's still offset (we haven't centered it in the shader yet), but it's working. If you rotate the light, you'll notice the mask doesn't rotate — that's where the quaternion implementation comes in next.



(3.8.e The mask follows the light's position)

**Note**

The script may not respond immediately after you change the light's transform. If that happens, restart Godot.

When working with **quaternions**, it helps to follow a clear rotation workflow:

- Declare them.
- Construct them.
- Conjugate them.
- Multiply them.
- Apply the rotation (transform).

With that in mind, declare a quaternion using a simple struct:

```
struct quaternion
{
    float x;
    float y;
    float z;
    float w;
};
```

These members — `x`, `y`, `z`, `w` — are the intrinsic components of a **quaternion**, as described in Figure 3.6.e. Next, you'll create quaternions using a helper function to build them from an axis-angle pair or from Euler angles.

```
struct quaternion { ... }

quaternion create(float angle, vec3 axis)
{
    float s = sin(angle/2.0);
    float c = cos(angle/2.0);
    vec3 v = normalize(axis) * s;

    quaternion q = quaternion(v.x, v.y, v.z, c);
    return q;
}
```

From the previous code, there are some key point:

- ① The `axis` argument is the rotation vector — the axis corresponding to the *ijk* directions.
- ② The vector `v` provides the three vector components of the quaternion from a unit axis.
- ③ The quaternion `q` is stored in the standard *xyzw* layout.

The conjugate of a quaternion  $q = (v, s)$  (vector part  $v$ , scalar part  $s$ ) is,

$$\bar{q} = s - v$$

(3.8.f)

```
quaternion create() { ... }

quaternion conjugate(quaternion q)
{
    float s = q.w;
    vec3 v = vec3(-q.x, -q.y, -q.z);

    quaternion cq = quaternion(v.x, v.y, v.z, s);
    return cq;
}
```

Then, the quaternion multiplication is defined as,

$$q_1 q_2 = (s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2)$$

(3.8.g)

```
quaternion conjugate() { ... }

quaternion multiplication(quaternion q1, quaternion q2)
{
    float s1 = q1.w;
    float s2 = q2.w;

    // The code continues on the next page.
```

```

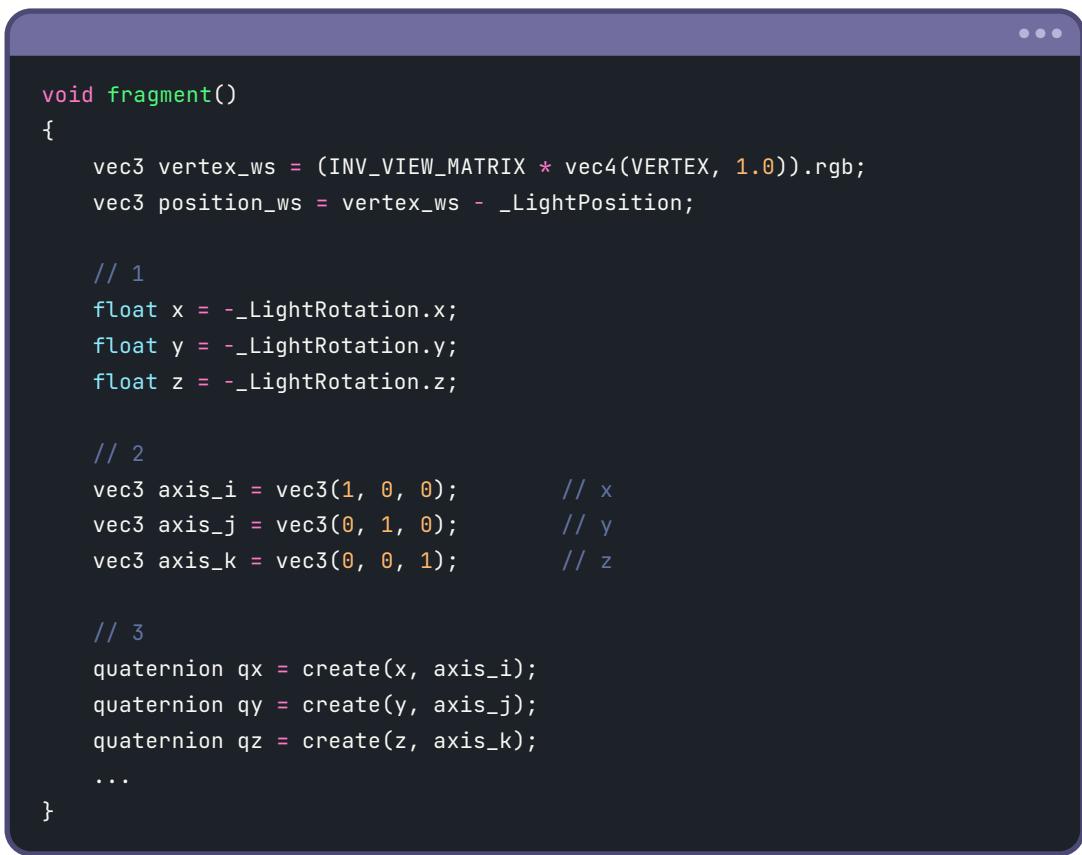
    vec3 v1 = vec3(q1.x, q1.y, q1.z);
    vec3 v2 = vec3(q2.x, q2.y, q2.z);

    float s = s1 * s2 - dot(v1, v2);
    vec3 v = s1 * v2 + s2 * v1 + cross(v1, v2);

    quaternion q = quaternion(v.x, v.y, v.z, s);
    return q;
}

```

Finally, we need to apply the rotation. Go to the **fragment()** method, initialize the XYZ angles from the light's rotation, and use the world unit axes IJK:



```

void fragment()
{
    vec3 vertex_ws = (INV_VIEW_MATRIX * vec4(VERTEX, 1.0)).rgb;
    vec3 position_ws = vertex_ws - _LightPosition;

    // 1
    float x = -_LightRotation.x;
    float y = -_LightRotation.y;
    float z = -_LightRotation.z;

    // 2
    vec3 axis_i = vec3(1, 0, 0);           // x
    vec3 axis_j = vec3(0, 1, 0);           // y
    vec3 axis_k = vec3(0, 0, 1);           // z

    // 3
    quaternion qx = create(x, axis_i);
    quaternion qy = create(y, axis_j);
    quaternion qz = create(z, axis_k);
    ...
}

```

In this case, we separate the variables to better understand how each quaternion is created per axis. Specifically:

- 1 You initialize rotation angles that follow the directional light's orientation.
- 2 You define the world unit axes.
- 3 You create one quaternion per axis.

Just like rotation matrices, you must respect a specific multiplication order when composing quaternions – the result depends directly on the sequence of rotations. In this case, use the YXZ order so the final orientation matches what you see when manipulating the directional light gizmo in the scene.

It's also essential to conjugate the resulting quaternion at the right step. Conjugation inverts the rotation direction and is crucial when you transform a vector between reference spaces (e.g., from object space to world space or the reverse), ensuring the rotation's orientation remains consistent.

```
void fragment()
{
    ...

    quaternion qx = create(x, axis_i);
    quaternion qy = create(y, axis_j);
    quaternion qz = create(z, axis_k);

    quaternion q_YXZ = multiplication(qz, multiplication(qx, qy));
    quaternion _q = conjugate(q_YXZ);

    ...
}
```

With the orientation quaternion ready, multiply it by the points you want to rotate — in this case, the vertices' world-space positions:

```
void fragment()
{
    ...
    quaternion _q = conjugate(q_YXZ);
    // 1
    quaternion p = quaternion(position_ws.x, position_ws.y, position_ws.z,
        → 0.0);
    quaternion pr = multiplication(q_YXZ, p);
    // 2
    pr = multiplication(pr, _q);

    ...
}
```

As we can see,

- 1 We create a new quaternion **p** that stores the world-space vertex position with  $W = 0$ .
- 2 We apply the rotation using the rule  $p' = q \cdot p \cdot \bar{q}$  and its conjugate **\_q**, to get the final world-space orientation.

At this point, **pr** holds the rotated XYZ components. Extract a **vec3** to represent the rotated position and use it to drive the projection:

```
void fragment()
{
    ...
    pr = multiplication(pr, _q);

    // 1
    vec3 position_ws_rot = vec3(pr.x, pr.y, pr.z);

    // The code continues on the next page.
```

```
// 2
vec2 uv_ws = position_ws_rot.xy + vec2(0.5);

vec3 front_map = texture(_FrontMap, UV).rgb;
vec3 back_map = texture(_BackMap, UV).rgb;
// 3
float spot_map = texture(_SpotMap, uv_ws.xy).r;

ALBEDO = vec3(spot_map);
}
```

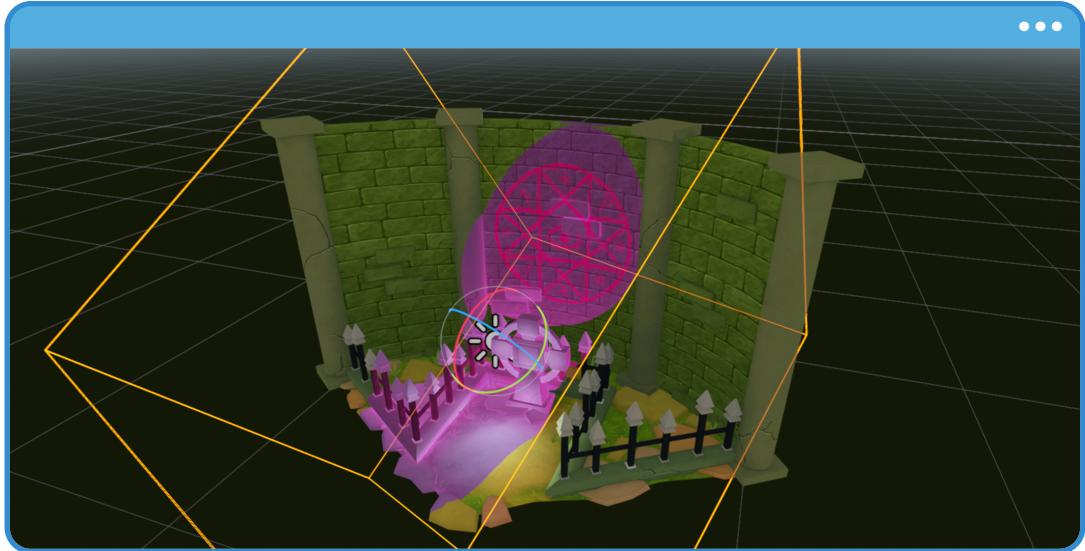
From this code:

- ① We define **position\_ws\_rot** as the rotated world-space position.
- ② We construct **uv\_ws** from its XY components and add 0.5 to center the projection in [0 : 1], aligning the circle with the scene origin.
- ③ We sample **\_SpotMap** with **uv\_ws** while keeping **front\_map** and **back\_map** on the model's regular **UV** – preserving each material's native mapping and avoiding base-texture distortion.

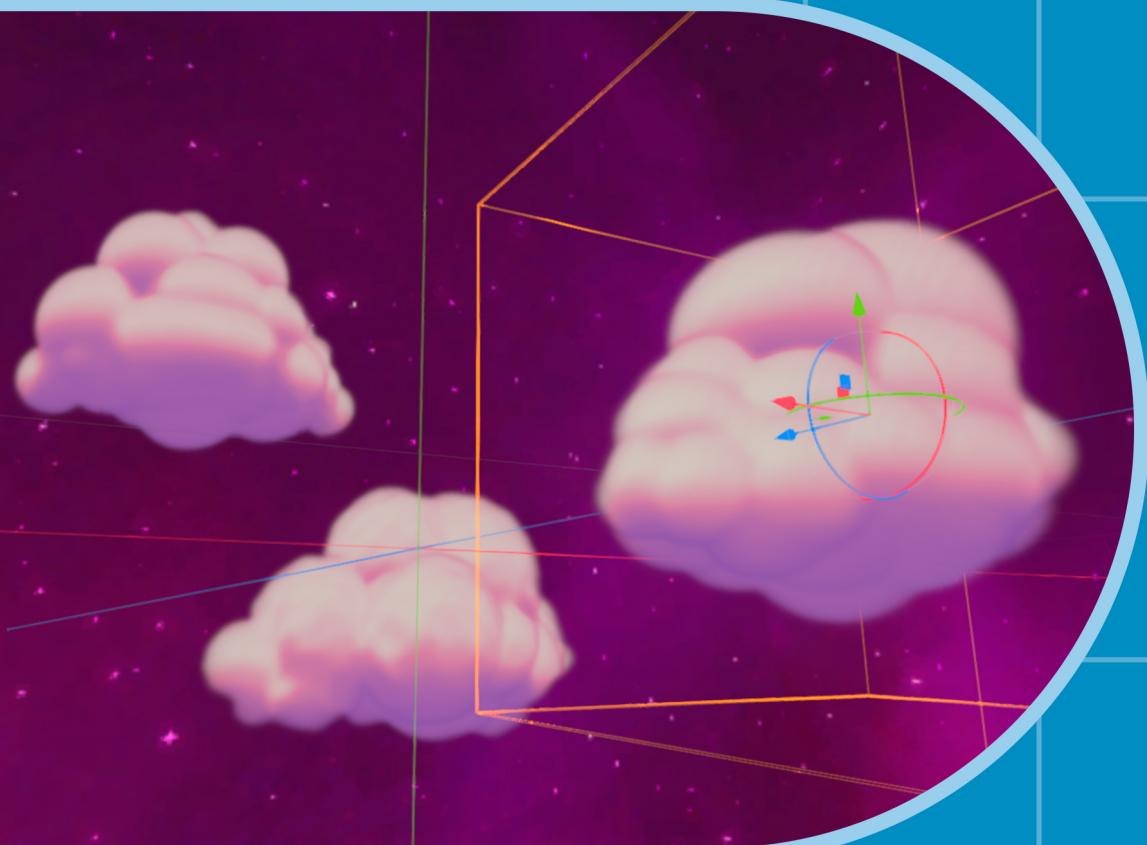
Back in the Viewport, when you rotate the light, the projection now rotates correctly and matches the gizmo's orientation. To finish the composition, linearly interpolate between **front\_map** and **back\_map** using **spot\_map** as the factor:

```
void fragment()
{
    ...
    vec3 color = mix(front_map, back_map, spot_map);

    ALBEDO = color;
}
```



(3.8.h Both textures applied to the model)



# Chapter 4

# Advanced VFX and

# Post-Processing.

Up to this point, you've learned several key concepts related to shader development in Godot. However, there comes a moment when you need to put all that knowledge into practice to truly internalize it. In this chapter, you will reinforce those ideas by creating effects commonly found in modern video games, exploring both their visual behavior and their technical implementation.

We'll begin by reviewing how to integrate shaders within the `canvas_item` pipeline to perform post-processing. You will use this approach to apply full-screen effects, including a practical exercise where you transform the final render into a Game Boy-inspired style using color gradients and predefined textures. This will help you understand how to manipulate the final output of a scene without altering the materials of individual objects.

From there, you'll move on to distance-based and camera-aware effects, implementing a progressive fade that activates when the camera gets too close to an object based on a predefined threshold. This type of translucency is especially useful in exploration games or isometric adventures, where maintaining clear visibility of the character is essential.

Next, you'll dive into the world of ray marching. You'll study its structure, walk through the algorithm step by step, and examine the geometric analogy that drives it. With this foundation, you will build a fully procedural three-dimensional shape from scratch, learning how volumes can be defined without meshes by using Signed Distance Functions, and how these ideas translate into GDSL.

After that, you'll be introduced to the Stencil Buffer, a powerful tool for controlling which parts of the scene are drawn or discarded. You'll review its properties, configuration options, and common use cases, and then create a custom shader that lets you observe its behavior directly from code.

To wrap up the chapter, you'll work on a porting exercise from Shadertoy. You will take a simple shader and adapt it to Godot's structure, learning how to reinterpret functions, coordinates, and conventions in the process. This workflow will deepen your understanding of graphics-oriented code and open the door to a much broader range of artistic and technical possibilities.

## 4.1 Post-Processing and the Game Boy Effect.

If you've ever developed a video game, you probably know the Game Boy — or maybe you even played on one. This handheld console appeared in the late eighties and reached peak popularity throughout the nineties. Despite its technical limitations, it had a unique charm, and one of its most recognizable traits was its signature color palette: a greenish display with an extremely limited range of tones.

Re-creating that palette with modern tools is simple, though not always practical for a real production environment. Even so, studying how to build post-processing filters that mimic classic visual styles is both valuable and fun. In this section, you'll develop a shader that imitates the Game Boy's color style. While the outcome won't be a perfect one-to-one reproduction, it will serve as an excellent exercise to help you deepen your understanding of post-processing techniques in Godot.

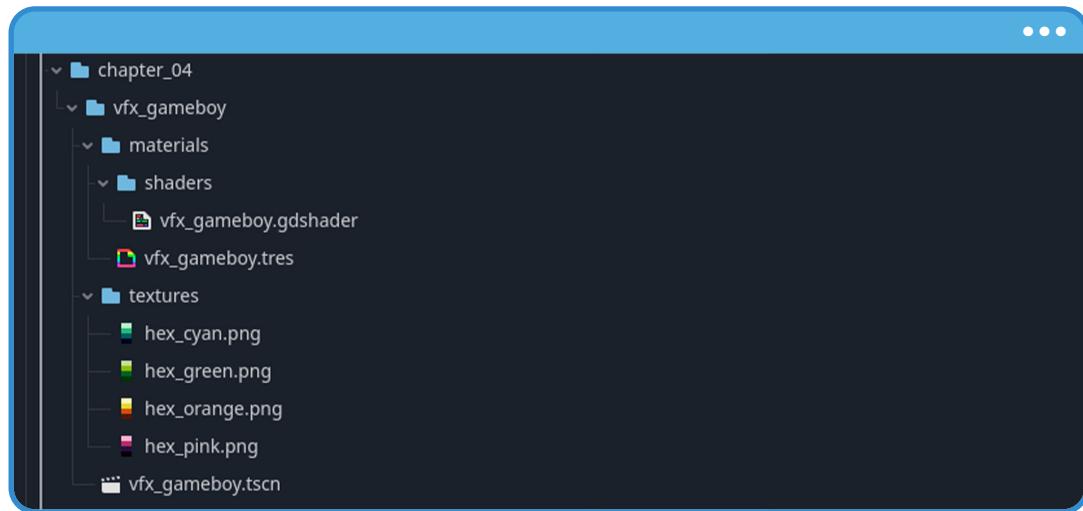
We'll begin by organizing the project. A clear structure will make it easier for you to follow the workflow in this chapter and reuse the assets later. To do that:

- 1 Inside the **assets** folder, create a new subfolder named **chapter\_04**, where you'll store all the resources used in this section.
- 2 Inside **chapter\_04**, add another folder named **vfx\_gameboy**.
- 3 Finally, inside **vfx\_gameboy**, create three subfolders:
  - a **materials**
  - b **shaders**
  - c **textures**

For this exercise, you'll reuse some of the assets from previous sections to build the base scene. However, you'll also need to download an additional set of textures specific to this chapter, since they'll be used to define the palette and color behavior of the final effect.

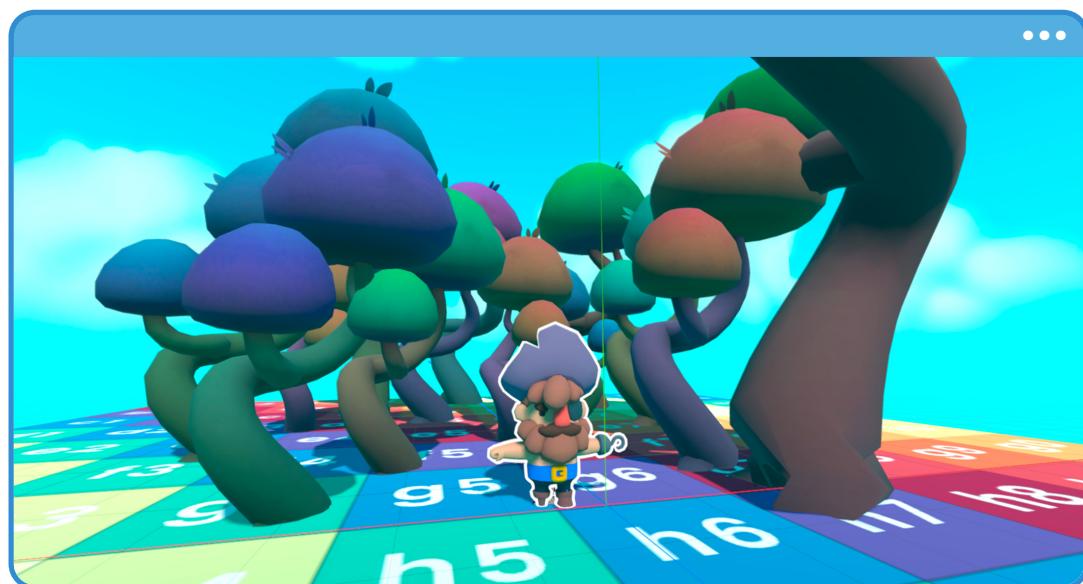
As usual, let's begin by creating both **Shader** and its corresponding **ShaderMaterial**. These two resources are essential for integrating the effect into the interface and applying it as part of the post-processing pipeline. To keep naming consistent and easy to track, you should name both resources after this section: **vfx\_gameboy**.

If you followed all the steps correctly, your project structure should look like this:



(4.1.a **vfx\_gameboy** folder has been added to the project)

For this exercise, there's a prebuilt scene. This scene contains a few trees, our character **Rebey Vane**, a plane that serves as the ground, and a **WorldEnvironment** node responsible for the sky and overall global lighting.



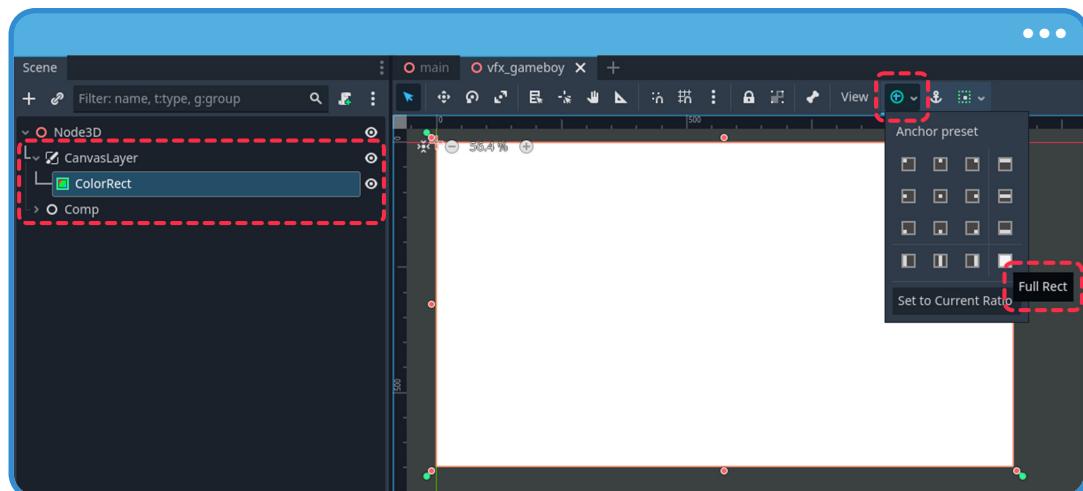
(4.1.b A scene has been configured)

Even though the scene already has its own style, the colors at this stage aren't particularly important. You're about to replace the tint, gamma, and most of the chromatic information through a post-processing effect. In fact, even if the entire scene were in grayscale, the result would be nearly identical because the filter will overwrite the composition's palette.

To begin configuring the effect, follow these steps:

- 1 Open the **Scene** panel.
- 2 Add a new **CanvasLayer** node.
- 3 As a child of the CanvasLayer, add a **ColorRect** node.

Make sure the ColorRect's **Anchor** is set to **Full Rect**. This ensures the rectangle covers the entire screen, which is essential when applying the shader as a global filter over the final rendered image.



(4.1.c The ColorRect has been configured as Full Rect)

#### Note

To properly preview the changes you'll make in this section, you must run the project. This means setting the current scene as the main scene and then pressing **Play**. Post-processing effects do not appear inside the editor; they are only applied during runtime.

At this stage, assign your **vx\_gameboy** material to the **Material** property of the **ColorRect** node. However, if you run the project now, you won't see any changes yet. By default, new shaders are created with the **spatial** type, and for the effect to work as a full-screen filter, you must explicitly switch the shader type to **canvas\_item** (as covered in Section 3.4 of Chapter 3).

When working with post-processing in Godot, there's an important technical detail: to apply an effect to the final image, the shader needs access to both the screen coordinates (**SCREEN\_UV**) and the rendered frame as a texture. In practice, this means that at the end of the Render Pipeline – right after the fragment stage (as discussed in Section 1.5 of Chapter 1) – the engine must take a “capture” of the framebuffer and feed it into your shader, pixel by pixel.

Godot provides an easy way to access the rendered frame through **hint\_screen\_texture**, which allows your shader to read the screen texture directly. Its basic usage looks like this:

```
shader_type canvas_item;

uniform sampler2D _ScreenTexture : hint_screen_texture, repeat_disable,
    filter_nearest;

void vertex()
{
    // Called for every vertex the material is visible on.
}
```

Here, you can infer that **repeat\_disable** prevents the texture from tiling outside the UV range, while **filter\_nearest** forces Godot to sample each pixel using nearest-neighbor instead of linear interpolation. In practice, this means every pixel is read exactly as it appears, without smoothing – perfect for retro-style effects like the Game Boy look.

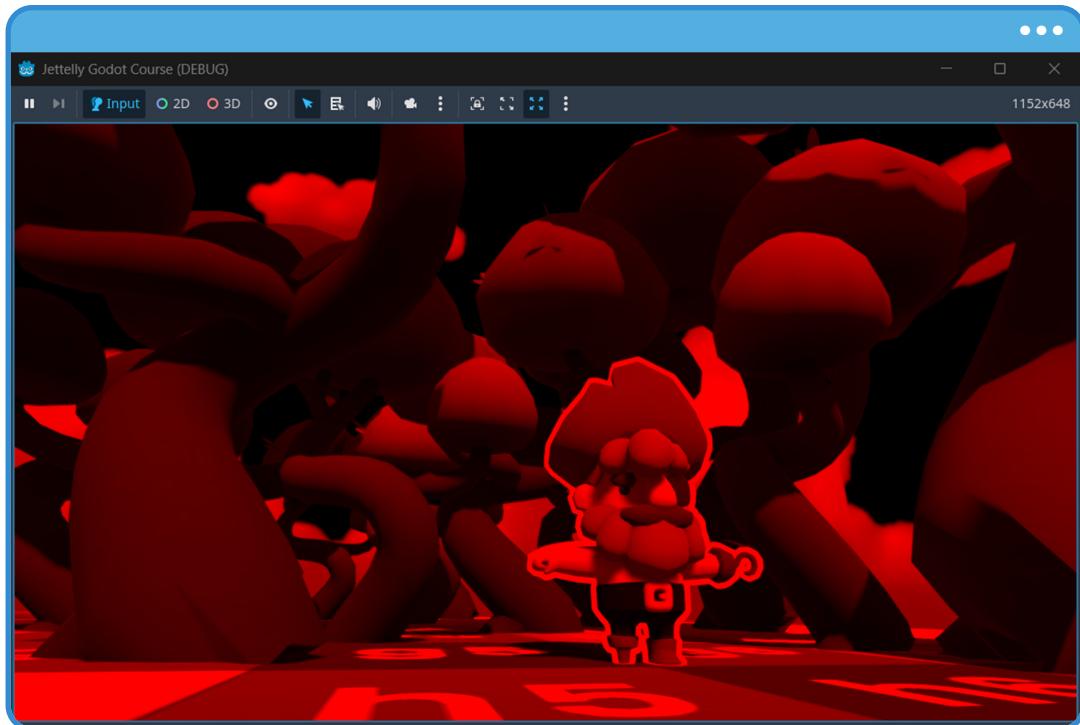
With the sampler declared, you can move on to the **fragment()** method and perform the following operations:

```
void fragment()
{
    // 1
    vec2 screen_uv = SCREEN_UV;
    // 2
    vec4 screen_texture = texture(_ScreenTexture, SCREEN_UV);
    // 3
    screen_texture.rgb *= vec3(1.0, 0.0, 0.0);
    // 4
    COLOR = screen_texture;
}
```

You can interpret the flow of this code as follows:

- 1 Access the screen UV coordinates using **SCREEN\_UV**.
- 2 Sample the current framebuffer texture using those coordinates and store the result in **screen\_texture**.
- 3 Apply a temporary color modification by multiplying the RGB values by a red vector. This step helps you confirm that the post-processing pipeline is working correctly.
- 4 Assign the result to **COLOR**, allowing the **ColorRect** to draw the processed texture across the entire screen.

Inside the editor, you won't see any visible change because post-processing effects only apply during runtime. But once you run the main scene by pressing **Play**, your entire render should appear tinted red.



(4.1.d A red tint has been applied to the final render)

With the pipeline working and the UI properly configured, you're ready to start building the Game Boy effect inside your shader. The first step is to convert the composition to grayscale. As mentioned earlier, the original color of the objects isn't relevant to the final effect because you'll be applying an entirely new palette based on luminance levels.

To perform this conversion, add the following function right before the **fragment()** method:

```
float luma(vec3 c)
{
    return max(0.0, dot(c, vec3(0.2126, 0.7152, 0.0722)));
}

void fragment() { ... }
```

What does the luma function do? In essence:

- It calculates the dot product between the input RGB color and a set of standard sRGB coefficients. These coefficients represent the relative contribution of each channel to the luminance perceived by the human eye: R contributes 21.26%, G contributes 71.52%, and B contributes 7.22%.

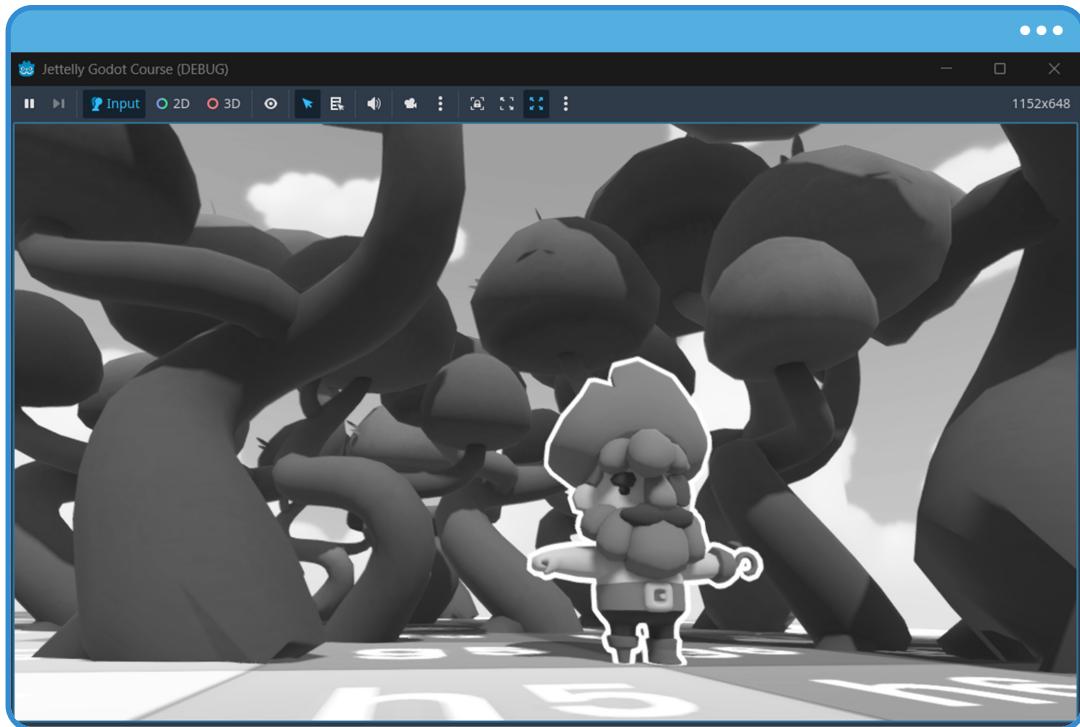
The function returns a scalar value describing how bright an RGB combination appears. With that value, you can replace the three RGB channels to generate a uniform tone, achieving a grayscale conversion.

If you update your **fragment()** method as shown below:

```
void fragment()
{
    vec2 screen_uv = SCREEN_UV;
    vec4 screen_texture = texture(_ScreenTexture, SCREEN_UV);
    screen_texture.rgb = vec3(luma(screen_texture.rgb));

    COLOR = screen_texture;
}
```

And then run the project by pressing **Play**, you should see the entire scene rendered in grayscale — an ideal starting point for applying the Game Boy's characteristic green palette.



(4.1.e The composition has been converted to grayscale)

Once you've converted the composition to grayscale, you can use that luminance to remap each pixel's color to a predefined range. In other words, you'll take the grayscale value — which represents how much light each pixel contributes — and replace it with one of the colors from your Game Boy palette. For example, you might assign one color to bright areas (near white), another to midtones, and a different one to the darkest regions.

If you explore the **textures** folder for this section, you'll find four textures that represent different variations of the Game Boy tone. These palettes emulate the console's classic color style, especially its most retro look. Each texture contains four colors ordered from highest to lowest luminance: one bright tone, two midtones, and one dark tone. This structure will let you classify each pixel based on its luminance and assign the most appropriate color from the palette.

For example, the **hex\_green** texture defines the following RGB values (from top to bottom):

- 202, 220, 160 (bright tone)
- 155, 187, 14 (high midtone)
- 48, 98, 47 (low midtone)
- 5, 56, 14 (dark tone)

You can treat the palette colors as indexed entries, starting at 0 for the brightest tone and ending at 3 for the darkest. The question is: how do you access each color individually inside the shader? To do that, you'll first declare a new **sampler2D** property that represents the Game Boy palette:

```
shader_type canvas_item;

uniform sampler2D _Palette : source_color;
uniform sampler2D _ScreenTexture : hint_screen_texture, repeat_disable,
    filter_nearest;

void vertex() { ... }
```

With this texture declared, you can create a helper function that retrieves a specific color based on an integer index:

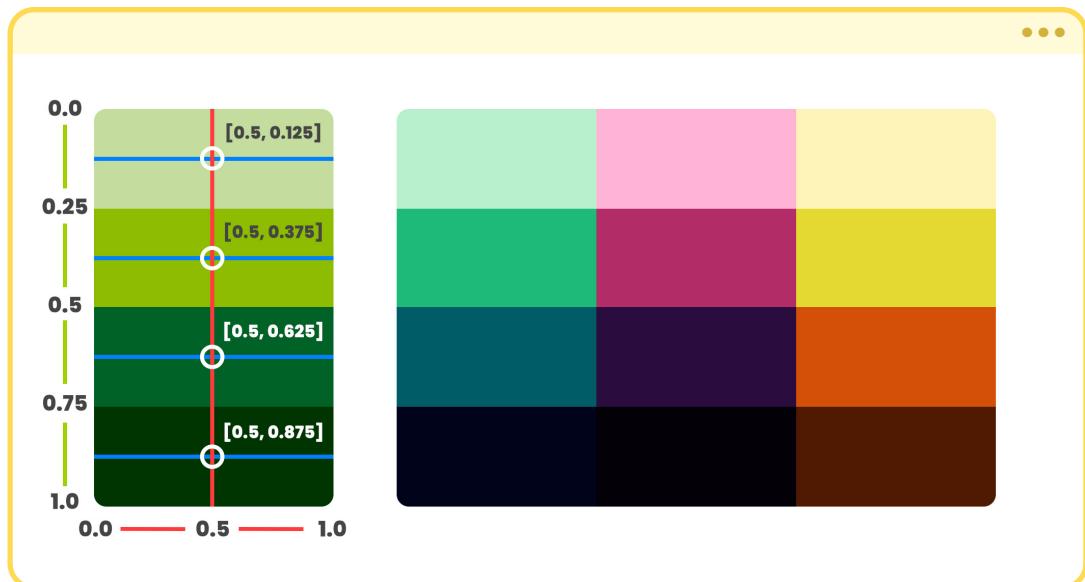
```
// 1
vec3 palette_fetch(int i)
{
    // 2
    i = clamp(i, 0, 3);
    // 3
    float v = 0.125 + 0.25 * float(i);
    // 4
    return texture(_Palette, vec2(0.5, 1.0 - v)).rgb;
}
```

Let's break down this method step by step:

- 1 Declare the `palette_fetch()` function, which receives an integer index. This index represents the luminance level you want to map (bright, midtones, or dark).
- 2 Clamp the index between 0 and 3, matching the number of colors available in the palette.
- 3 Compute the vertical coordinate (`v`) inside the palette texture. Each color occupies a band of 0.25 along the V axis because there are four colors. You then add 0.125 to target the center of each band (avoiding the edges). This produces the following UV positions:
  - a When `i = 0`:  $v = 0.125$
  - b When `i = 1`:  $v = 0.375$
  - c When `i = 2`:  $v = 0.625$
  - d When `i = 3`:  $v = 0.875$
- 4 Finally, you return the sampled color using the `vec2(0.5, 1.0 - v)` coordinates. You use 0.5 for U to stay at the horizontal center of the texture, and you sample at `1.0 - v` because Godot stores textures starting from the top-left corner. This inversion lets you read the colors in the correct order from top to bottom.

In other words, this function takes an integer index, computes the exact position of the corresponding color inside the palette texture, and returns it as an RGB value. With it, you can remap your grayscale composition to a Game Boy-style palette by assigning the appropriate color to each pixel according to its luminance.

Next, you can examine the visual reference to better understand how these texels are laid out and why you sample from the center of each band.



(4.1.f Texel index coordinates)

As you can see in Figure 4.1.f, the texels used for each index are located exactly at the center of each vertical block in the palette. This happens thanks to the coordinates passed to `vec2(0.5, 1.0 - v)`, where you sample at the horizontal midpoint (0.5) and at the precise vertical position computed with `v`.

In theory, you could reduce the size of the palette texture even further, since you're only sampling four specific texels. This would make the texture lighter and slightly optimize the sampling process. However, since this detail isn't the focus of the section, you'll continue using the textures as they were originally designed.

At this point, you already have a function that retrieves palette colors by index. What you need now is a function that determines which color to assign to each pixel based on the luminance of the composition. To accomplish this, implement the following:

```
// 1
vec3 gameboy_color(vec3 color)
{
    // 2
    float t = luma(color);
    // 3
    t = smoothstep(0.1, 0.9, t);
    // 4
    int idx = int(floor(t * 4.0));
    // 5
    if(idx == 4) idx = 3;
    // 6
    return palette_fetch(idx);
}
```

You can interpret this function as follows:

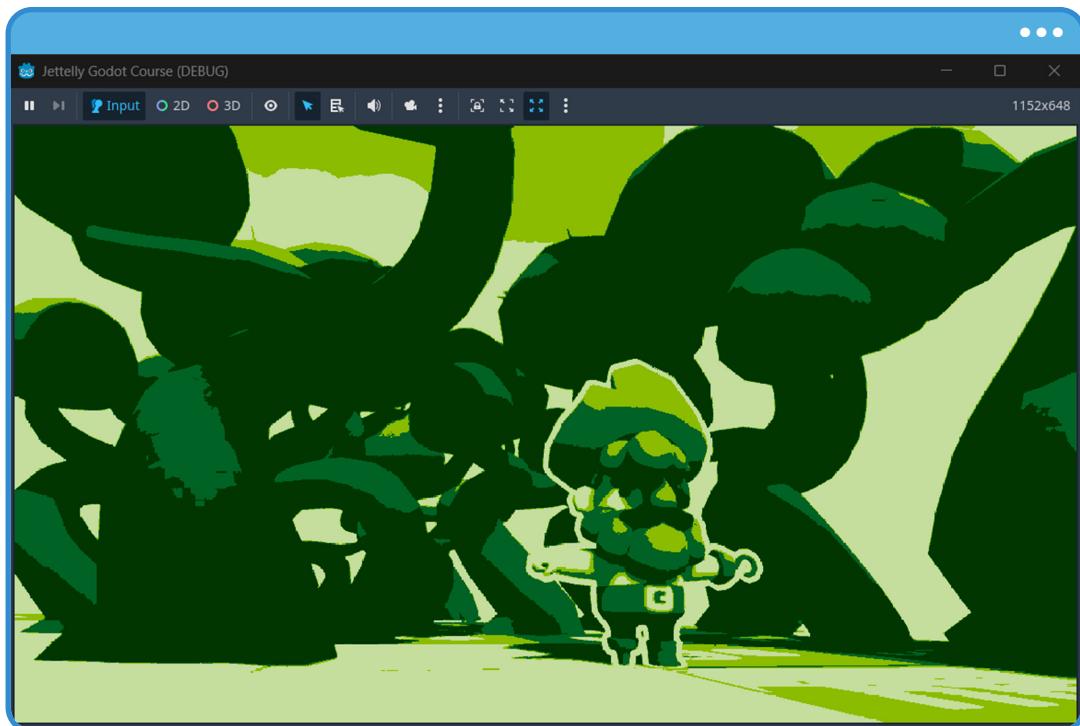
- ➊ Declare **gameboy\_color()**, which receives an RGB value as input. This represents the pixel's original color before applying the effect.
- ➋ Convert that color to luminance using **luma()**, obtaining a value between 0.0 and 1.0 that describes how bright the pixel is.
- ➌ Apply **smoothstep(0.1, 0.9, t)** to remap the luminance. This slightly compresses the darkest and brightest ranges, producing a more pronounced contrast — similar to the visual behavior of retro screens.
- ➍ Compute an index between 0 and 4 by multiplying the luminance by the number of possible intervals and applying **floor()**.
- ➎ Correct the edge case: if the index evaluates to 4, you clamp it down to 3, since the palette contains only four colors.
- ➏ Finally, you return the mapped palette color by calling **palette\_fetch(idx)**.

In this way, each luminance range is mapped to one of the four colors in the palette, emulating the Game Boy style. To finish the effect, go back to the `fragment()` method and update the RGB channels of the screen texture using the result of `gameboy_color()`:

```
void fragment()
{
    vec2 screen_uv = SCREEN_UV;
    vec4 screen_texture = texture(_ScreenTexture, SCREEN_UV);
    screen_texture.rgb = gameboy_color(screen_texture.rgb);

    COLOR = screen_texture;
}
```

When you run the project, you should see the selected palette applied across the full range of luminance values in the composition.



(4.1.g The **hex\_green** palette has been applied to the composition)

To push the effect even closer to the aesthetic of the original Game Boy, you can add a slight pixelation pass on top of the render. Implementing this effect is straightforward. Start by defining the following helper function:

```
// 1
vec2 to_pixel(vec2 screen_uv, vec2 screen_pixel, float pixel_size)
{
    // 2
    vec2 px = screen_uv * screen_pixel;
    // 3
    px = floor(px / pixel_size) * pixel_size + pixel_size;
    // 4
    return px / screen_pixel;
}
```

Here's what happens step by step:

- ➊ Declare the `to_pixel()` method, which receives the screen UV coordinates, the screen size in pixels, and the target pixel size you want to “round” to.
- ➋ Convert UV coordinates into pixel coordinates by multiplying them by the screen resolution. This lets you operate in pixel units rather than normalized values between 0 and 1.
- ➌ Divide those pixel coordinates by `pixel_size`, apply `floor()`, and multiply again by `pixel_size`. This groups several pixels into a single block, causing them to sample the same color and producing the pixelation effect. The additional `+ pixel_size` offsets sampling toward the center of each block instead of the edge.
- ➍ Finally, you normalize again by dividing by the screen resolution, converting the result back into valid UV coordinates between 0 and 1.

Before returning to the `fragment()` method, you'll need a property that allows you to control the pixel size dynamically. Go back to the global variables and add the following:

```
shader_type canvas_item;

uniform sampler2D _Palette : source_color;
uniform int _PixelSize : hint_range(1, 10, 1);
uniform sampler2D _ScreenTexture : hint_screen_texture, repeat_disable,
    filter_nearest;
```

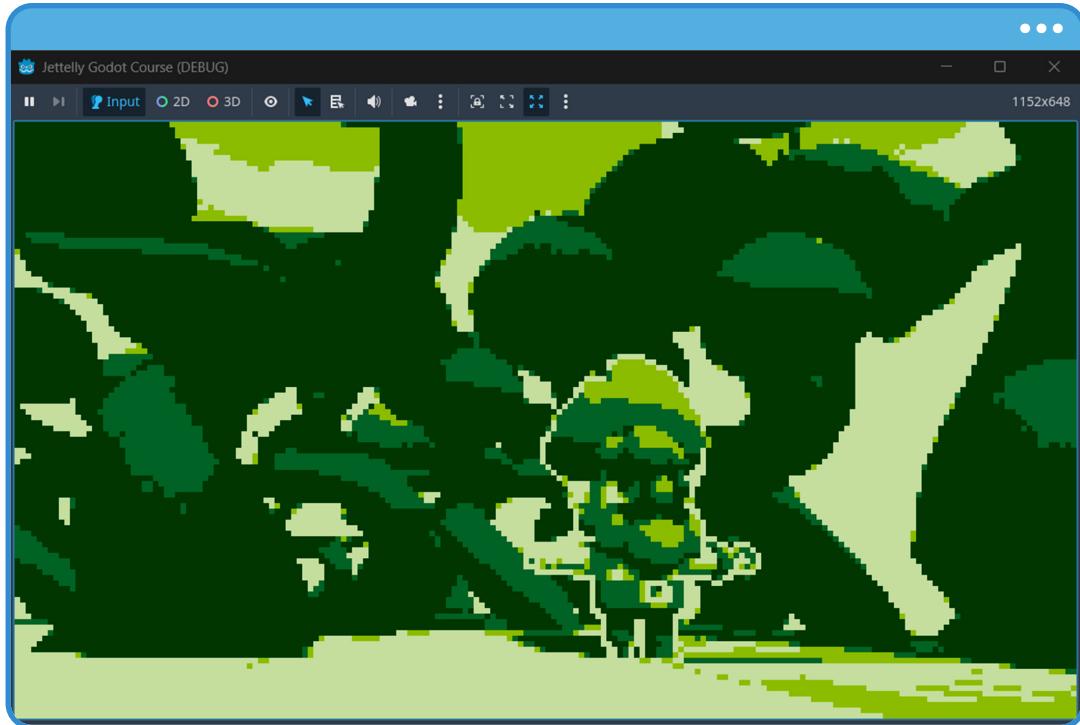
The `_PixelSize` property lets you adjust the size of the pixel blocks: smaller values produce a subtle pixelation effect, while larger values create a much more pronounced low-resolution look.

Now return to the `fragment()` method and update the `screen_uv` vector to use your new helper function:

```
void fragment()
{
    vec2 screen_uv = to_pixel(SCREEN_UV, 1.0 / SCREEN_PIXEL_SIZE,
        float(_PixelSize));
    vec4 screen_texture = texture(_ScreenTexture, screen_uv);
    screen_texture.rgb = gameboy_color(screen_texture.rgb);

    COLOR = screen_texture;
}
```

To see the effect in action, adjust **Pixel Size** in the Inspector to any value greater than 1. In the reference example, a value of **6** produces a pixelation similar to the low-resolution displays of the classic Game Boy.



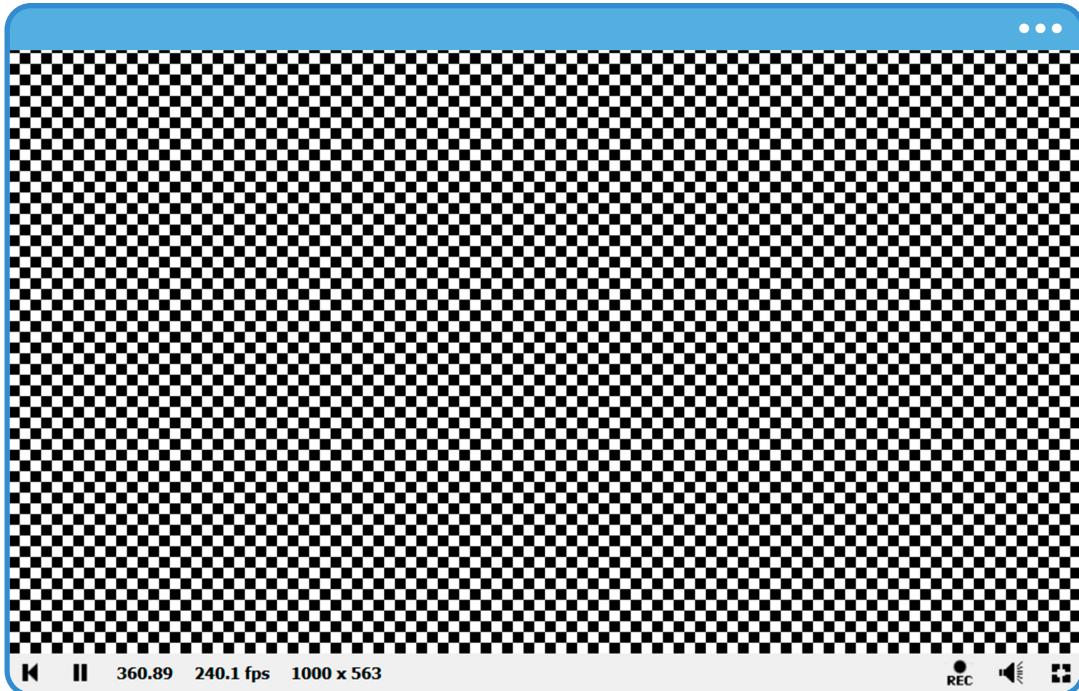
(4.1.h The composition has been pixelated)

You could consider this section complete at this point, since the result is already quite close to the classic Game Boy effect: you have a limited palette, a grayscale composition remapped to that palette, and a pixelation pass that reinforces the retro look. However, you can still take things one step further by porting an effect from **Shadertoy** to enrich the composition and explore how to adapt external shaders to Godot. That process will be the focus of the next section.

## 4.2 Porting an Effect from Shadertoy.

Shadertoy is a widely used platform among developers and artists who enjoy working with coordinates, mathematical functions, and real-time visual effects. If you've read more than one of our books, you already know that we frequently mention **Indigo Quilez**, one of the most influential figures in computer graphics and the founder of Shadertoy.

The effect you'll port in this section is a **checkerboard pattern** created by **Alundra**, which looks like this:



(4.2.a Checkerboard pattern)

The original GLSL implementation is the following:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    float Size = 10.0;
    vec2 Pos = floor(fragCoord / Size);
    float PatternMask = mod(Pos.x + mod(Pos.y, 2.0), 2.0);
    fragColor = PatternMask * vec4(1.0, 1.0, 1.0, 1.0);
}
```

**Note**

You can view the effect at the following link: <https://www.shadertoy.com/view/lt2XWK>

Let's briefly analyze this implementation:

- **mainImage()** is the entry point responsible for generating the output image. In Godot, this role is taken by the **fragment()** method.
- **fragColor** represents the final color of the fragment. In a post-processing shader, you can treat this as equivalent to the **COLOR** variable. In other types of materials, it could correspond to **ALBEDO**.
- **fragCoord** represents the absolute pixel coordinates of the screen. You might initially assume it maps directly to **SCREEN\_UV**, and although they are related, they're not the same. **fragCoord** uses pixel-space coordinates, while **SCREEN\_UV** is normalized between 0 and 1.

$$fragCoord \approx \frac{SCREEN\_UV}{SCREEN\_PIXEL\_SIZE}$$

(4.2.b)

This keeps the pattern stable even when the Viewport scale changes. The remaining operations are fairly straightforward. For example, the **mod()** function returns the remainder of dividing X by Y, which allows the shader to alternate values and generate the checkerboard pattern.

Now, how can you integrate this function into your current effect? The process is simple. Below is the adapted version for Godot, keeping the same variable names used in Shadertoy to make the comparison easier:

```
void fragment()
{
    // 1
    float size = float(_PixelSize);
    // 2
    vec2 screen_uv = to_pixel(SCREEN_UV, 1.0 / SCREEN_PIXEL_SIZE, size);
    vec4 screen_texture = texture(_ScreenTexture, screen_uv);
    screen_texture.rgb = gameboy_color(screen_texture.rgb);

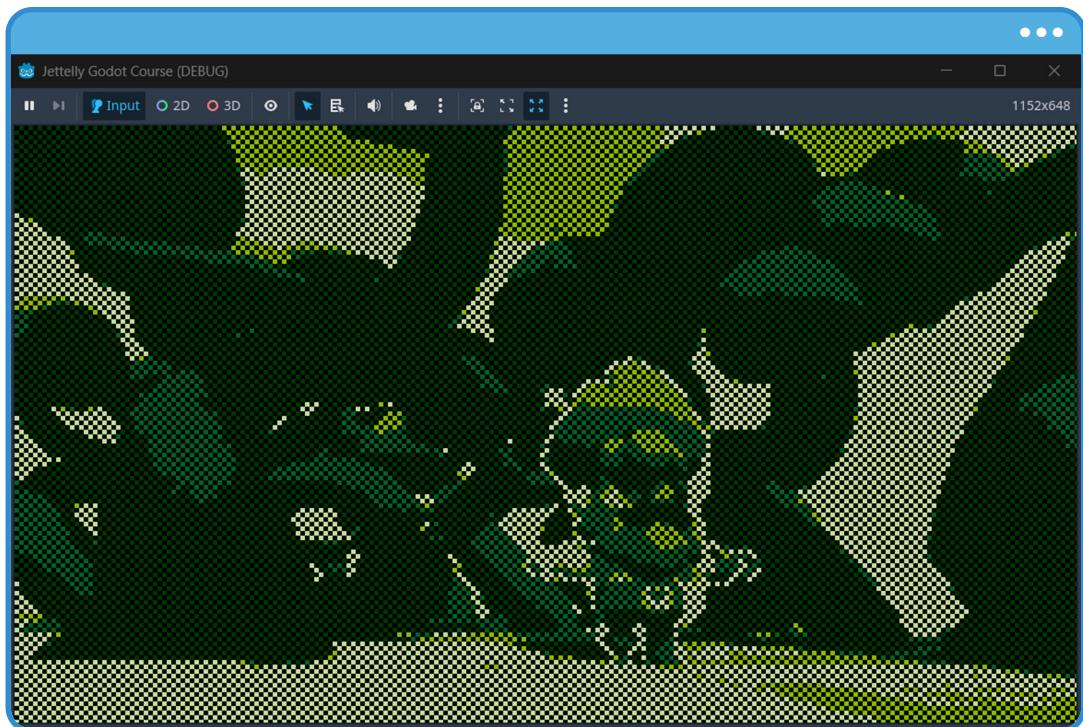
    // The code continues on the next page.
}
```

```
//3
vec2 pos = floor((SCREEN_UV / SCREEN_PIXEL_SIZE) / size);
float pattern_mask = mod(pos.x + mod(pos.y, 2.0), 2.0);
//4
screen_texture.rgb *= pattern_mask;
COLOR = screen_texture;
}
```

Here's what's happening:

- ➊ Declare a new variable called `size`, which stores `_PixelSize` converted to a float. This lets you reuse it both in `to_pixel()` and in the checkerboard calculation.
- ➋ Pass `size` as the third argument to `to_pixel()`, ensuring that both the pixelation and the pattern operate at the same scale.
- ➌ Use `size` to divide the normalized screen coordinates `(SCREEN_UV / SCREEN_PIXEL_SIZE)` before applying `floor()`. This determines the size of each checkerboard cell.
- ➍ Multiply the RGB channels of `screen_texture` by `pattern_mask`, applying the checkerboard over the Game Boy–processed composition.

The final result is a combination of pixelation, the Game Boy palette, and the checkerboard pattern, producing an interesting retro-style finish.



(4.2.c The pattern has been applied on the final render)

### 4.3 Transparencies and Distance.

When we talk about transparency, we're essentially referring to the **ALPHA** channel, which determines how opaque each fragment is during rendering. In a shader, this channel plays a special role: it not only controls the visibility of the final color, but it can also be used as a tool to create effects based on distance, depth, or light intensity within the composition.

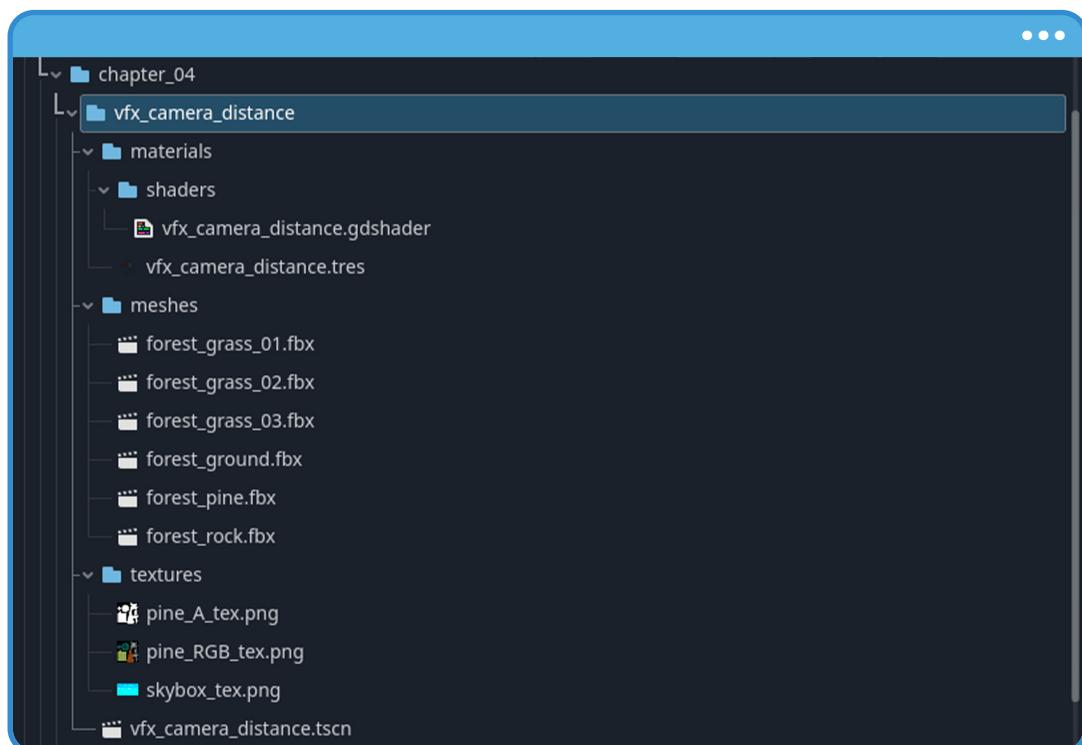
In this section, you'll explore how to manipulate the alpha channel using the distance between a 3D object and the camera. The goal is to create a transparency effect that activates when certain elements obstruct the player's view. This technique is widely used in third-person games: when a tree, rock, or wall blocks the camera's line of sight, the obstructing object becomes partially transparent, ensuring that important gameplay information remains visible.

To get started, you'll organize the project using the same modular structure you've been following throughout the previous chapters:

- 1 Inside the **chapter\_04** folder, create a subfolder named **vx\_camera\_distance**, where you'll store all the resources for this section.
- 2 Inside **vx\_camera\_distance**, create four additional subfolders:
  - a **materials**
  - b **shaders**
  - c **meshes**
  - d **textures**

Before moving on, make sure you place all the files associated with this section into these folders, including rock and pine models, textures, and any other provided assets. Then, create a **Shader** and its corresponding **ShaderMaterial**, naming them according to your project's existing conventions.

If you've completed each step correctly, your project structure should look like this:

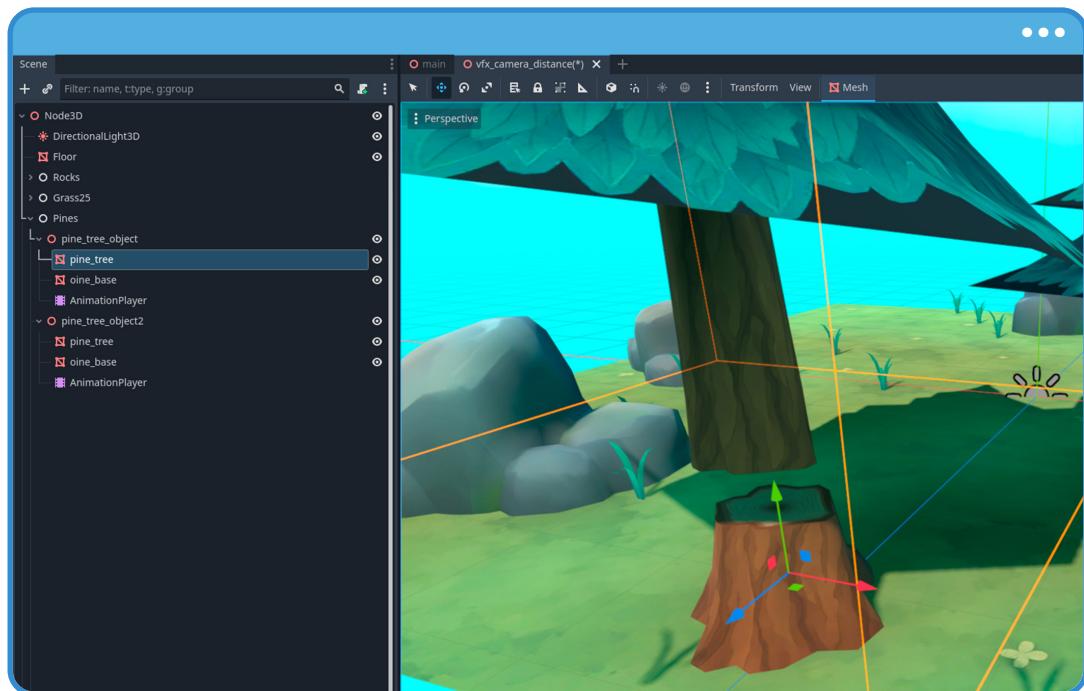


(4.3.a Project structure)

If you look closely at Figure 4.3.a, you'll notice that the **meshes** folder contains several 3D objects you can use to build your composition. However, the most important asset for this

exercise is **forest\_pine**. As its name suggests, this resource represents a pine tree – but what makes it particularly useful is that it's divided into two separate meshes: **pine\_tree** and **pine\_base**.

You'll apply the shader you just created exclusively to **pine\_tree**, while **pine\_base** will keep its original material. Thanks to this separation, you can reproduce a progressive transparency effect similar to the one used for the pine trees in **DOGWALK**, a project developed by the Blender team. This technique is especially effective because it allows you to fade only the upper part of the pine when the character is obscured by vegetation, while keeping the base fully visible to maintain visual coherence in the scene.



(4.3.b The forest\_pine object is divided in two parts)

Before moving on to the shader logic, you'll set up a small composition in your scene. This will help you clearly evaluate the changes you'll apply to the alpha channel in the next steps. Make sure to assign the material correctly:

- Assign the **Shader** to your **ShaderMaterial**, confirming that both resources are stored in their designated folders.

- Apply the **ShaderMaterial** only to the **pine\_tree** node of each **forest\_pine** model in the scene.

With this setup complete, you'll be ready to start implementing the distance-based transparency effect.



(4.3.c The ShaderMaterial has been applied to every pine in the scene)

The first step in your shader will be to incorporate the texture properties so you can better visualize the silhouette of the pine. If you explore the **textures** folder in your project, you'll notice that the alpha channel of the main texture is stored in a separate image. In a production environment, you would normally pack this information into a single texture, but here it's been separated for educational purposes so you can clearly see what each map contributes.

```
shader_type spatial;
render_mode diffuse_toon;
render_mode cull_disabled;

uniform sampler2D _MainTexRGB : source_color;
uniform sampler2D _MainTexA : source_color;

void vertex() { ... }
```

Most of this code should look familiar, but two render modes appear here that you haven't used before: **diffuse\_toon** and **cull\_disabled**. **diffuse\_toon** applies a stylized diffuse shading closer to an illustrated or cartoon look. **cull\_disabled** disables backface culling, allowing you to see both the inside and outside of the model. Both settings are optional, but they add a nice stylistic touch to the final result.

Next, go to the **fragment()** method and initialize the textures as shown:

```
void fragment()
{
    vec3 albedo = texture(_MainTexRGB, UV).rgb;
    float alpha = texture(_MainTexA, UV).r;

    ALBEDO = albedo;
    ALPHA = alpha;
}
```

When you return to the Viewport, the first thing you'll notice is an issue in how the render pipeline interprets transparency and depth between objects. This is known as a **depth sorting issue**. It occurs because transparent materials can't write directly to the **depth buffer** — they need to blend their color with everything already drawn behind them.

To handle this, the engine sorts transparent objects by their distance to the camera and draws them from back to front. When that ordering fails due to complex geometry, intersecting objects, or fast camera changes, you get incorrect visual results: objects that should appear

behind show up in front, certain portions of the mesh “jump” layers when moving the camera, or parts of the transparency appear inconsistent or flicker.



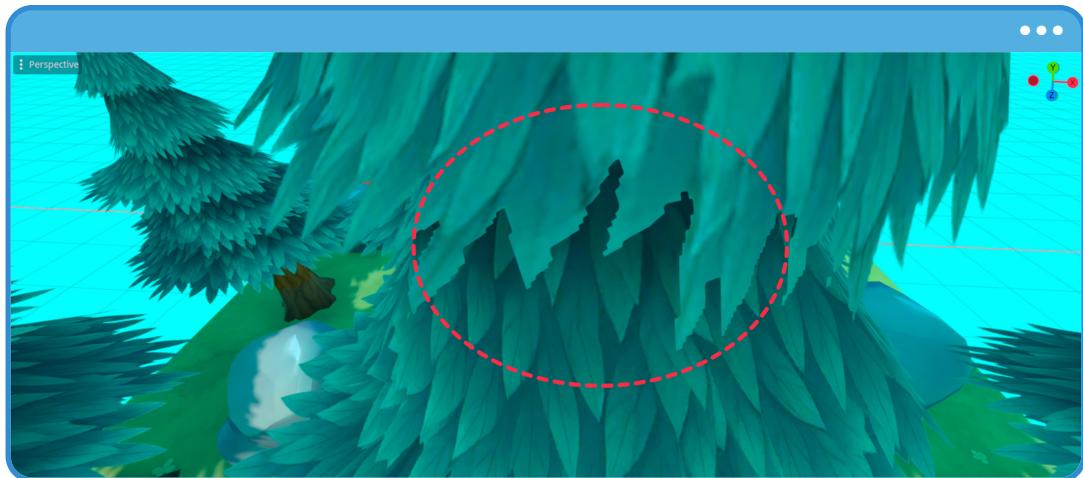
(4.3.d Depth issue in transparency)

To fix this problem, you first need a **depth pre-pass** over the opaque portions of the objects and then reuse that depth information when drawing the transparent fragments. In Godot, you can enable this behavior using the **depth\_prepass\_alpha** render mode, which makes the engine write depth before blending transparency:

```
shader_type spatial;
render_mode diffuse_toon;
render_mode cull_disabled;
render_mode depth_prepass_alpha;
```

By enabling **depth\_prepass\_alpha**, you tell Godot to render the object in two stages: It records the object's depth as if it were fully opaque, then applies transparency while respecting which fragments should appear in front or behind. This significantly reduces the most noticeable depth-sorting issues in your pines.

However, there is still another common transparency artifact: the translucency border or “halo” that appears around the leaves. In this region, the transition between the opaque parts of the foliage and the fully transparent areas of the texture is poorly defined, creating a visible fringe around the silhouette, as you can see in the following reference.



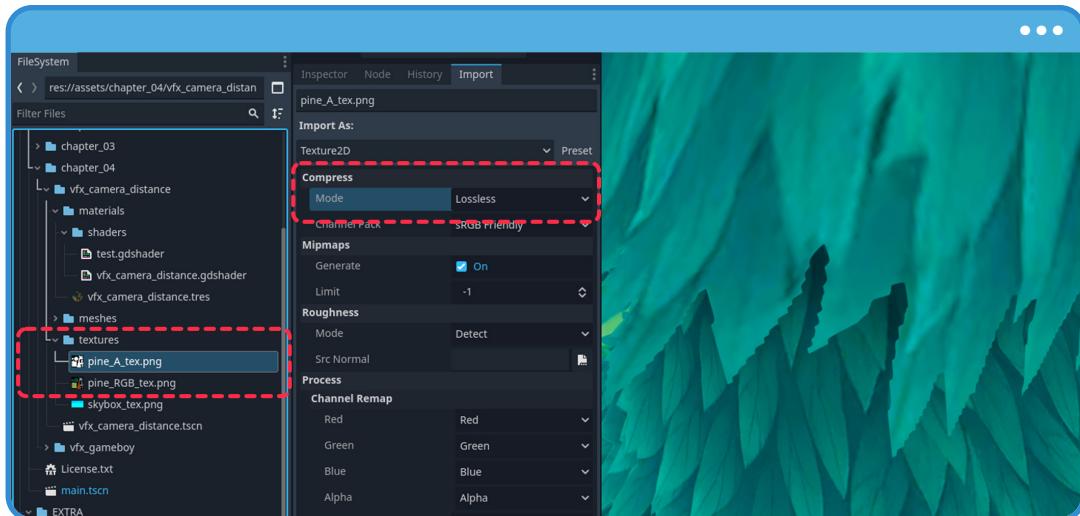
(4.3.e Hard transparency edges have appeared)

This issue arises mainly for two reasons:

- 1 Because of the texture's size and the compression applied to it.
- 2 Because no antialiasing has been applied yet to the cutout edges.

If you inspect the texture in your project and check the **Import** tab, you'll see that its compression mode is set to **VRAM Compressed**. This format is useful for saving memory, but it introduces visible artifacts in areas where the alpha channel contains sharp transitions — such as the edges of tree leaves.

If you switch this option to **Lossless**, you prevent the quality loss caused by compression. As a result, the cutout edges gain definition and the contour of the model looks cleaner and smoother. This improvement is especially helpful when working with textures that rely heavily on the alpha channel to define their shape, such as vegetation, hair, or hand-drawn elements.



(4.3.f The texture compression settings have been updated)

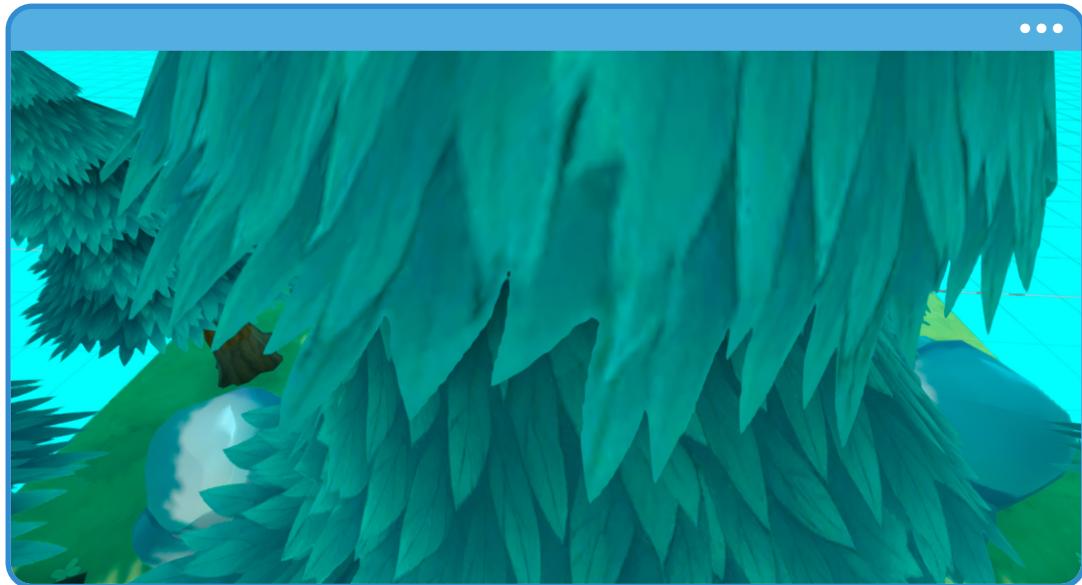
As shown in Figure 4.3.f, switching the compression mode noticeably improves the clarity of the transparent edges — by roughly 40%. However, you can enhance the result even further without increasing the texture size or changing its compression method. Godot provides an internal variable called **ALPHA\_ANTIALIASING\_EDGE**, which softens cutout edges by generating a transitional boundary between opaque and transparent regions. This variable affects the blending process directly, reducing the harsh cutouts and minimizing the jagged artifacts typically associated with alpha-masked textures.

To use it, simply assign a value to the variable inside the **fragment()** method, right after defining **ALPHA**:

```
void fragment()
{
    vec3 albedo = texture(_MainTexRGB, UV).rgb;
    float alpha = texture(_MainTexA, UV).r;

    ALBEDO = albedo;
    ALPHA = alpha;
    ALPHA_ANTIALIASING_EDGE = 0.5;
}
```

If you return to the scene and observe the foliage again, you'll notice that the edges now exhibit a smoother and more natural transition. This greatly improves the visual integration of vegetation within the composition.



(4.3.g The transparency edge resolution has been improved)

With the transparency issues under control, you're ready to start implementing a distance-based gradient. To do this, you need access to both the camera position and each vertex position in world space. With that information, you can calculate how far the tree's fragments are from the viewer and adjust their visibility or color accordingly.

Begin by adding the following properties and global variables to your shader:

```
uniform sampler2D _MainTexRGB : source_color;
uniform sampler2D _MainTexA : source_color;
// 1
uniform float _Distance : hint_range(0.0, 10.0, 0.1);
// 2
varying vec3 vertex_ws;

// The code continues on the next page.
```

```
void vertex()
{
    // 3
    vertex_ws = (MODEL_MATRIX * vec4(VERTEX, 1.0)).rgb;
}
```

Here's what this code does:

- ➊ You declare a new property named `_Distance`, which you'll use to control the strength of the gradient based on the distance between the camera and the object's vertices.
- ➋ You create a global varying variable, `vertex_ws`, to store the position of each vertex in world space. This will let you evaluate, fragment by fragment, how far the tree's geometry is from the camera.
- ➌ You transform the vertices from local space to world space using `MODEL_MATRIX`, and store the result in `vertex_ws`.

You can now use `vertex_ws` inside the distance calculation to create the effect we're aiming for. Update the `fragment()` method as follows:

```
void fragment()
{
    vec3 albedo = texture(_MainTexRGB, UV).rgb;
    float alpha = texture(_MainTexA, UV).r;
    // 1
    float distance_color = clamp(distance(vertex_ws, CAMERA_POSITION_WORLD) /
        _Distance, 0.0, 1.0);
    // 2
    ALBEDO = vec3(1.0 - distance_color);
    ALPHA = alpha;
    ALPHA_ANTIALIASING_EDGE = 0.5;
}
```

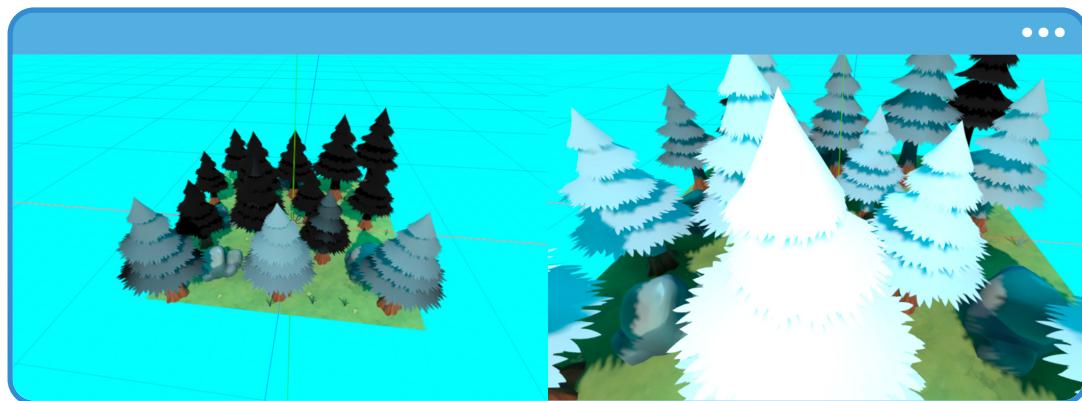
Here's what's happening:

- ➊ Declare a new variable, `distance_color`, which stores the normalized distance between each vertex of the object and the camera. We use the `distance()` function in

world space, and clamp the result to the [0:1] range to produce a value suitable for blending or for driving a gradient.

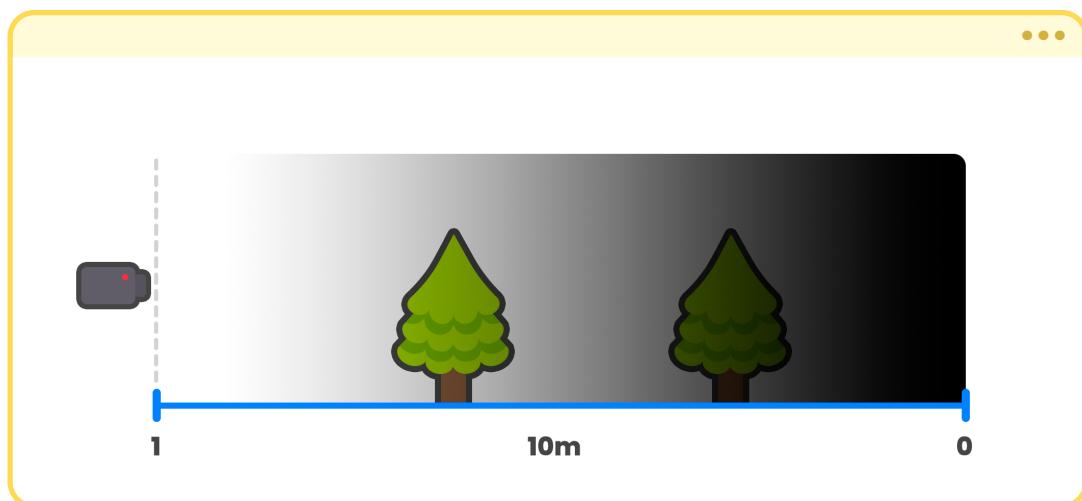
- ② Temporarily, we assign **distance\_color** to **ALBEDO** so you can visualize the gradient directly on the object. We invert the value ( $1.0 - \text{distance\_color}$ ) to better highlight the regions closest to the camera while testing.

When you return to the Viewport, you'll see that the color of the pine changes dynamically based on its proximity to the camera. You can adjust the strength and range of the gradient by modifying the **Distance** property in the **Inspector**.



(4.3.h Configuration with **Distance** set to 10)

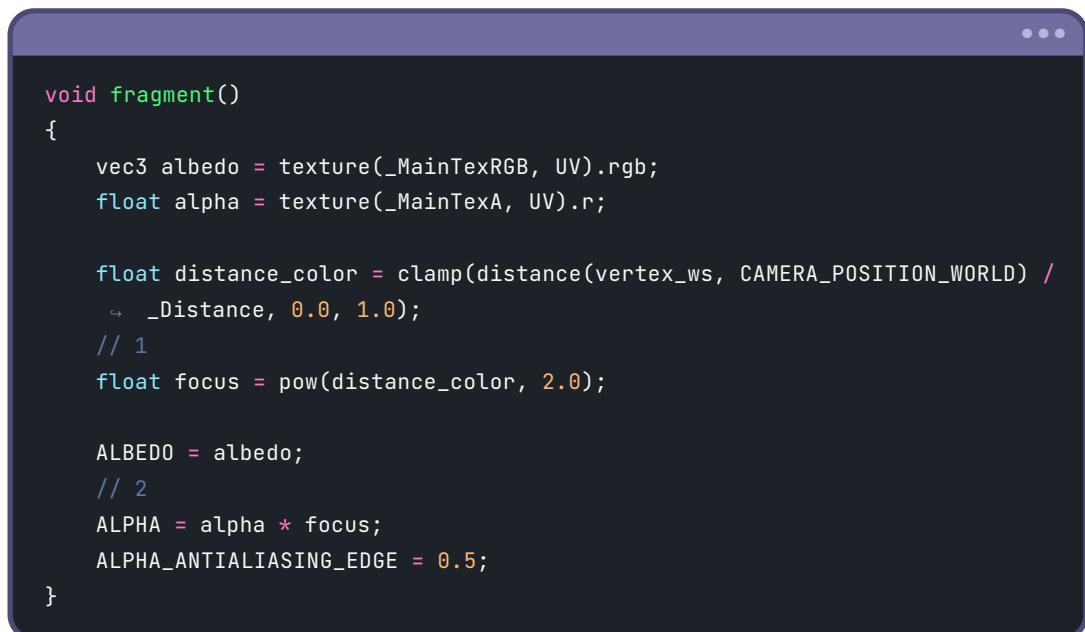
Now, what exactly is happening inside the distance calculation from a visual and geometric perspective? To better understand the concept, take a look at the following reference:



(4.3.i The color produced by the distance has been inverted)

Since the result of the distance calculation is a normalized value between 0 and 1, you can use it directly to drive the object's transparency. However, if you apply this transition as-is, the gradient will be too broad and will affect areas that shouldn't become translucent. To achieve more controlled behavior, you need to “tighten” the gradient curve so that the transparency change happens only within a specific band of the object in front of the camera.

To create this more localized effect, you can apply a smoothing function or transition curve to the distance value before assigning it to the alpha channel. This lets you define more precisely which areas fade out and over what distance range the effect occurs.



```

void fragment()
{
    vec3 albedo = texture(_MainTexRGB, UV).rgb;
    float alpha = texture(_MainTexA, UV).r;

    float distance_color = clamp(distance(vertex_ws, CAMERA_POSITION_WORLD) /
        _Distance, 0.0, 1.0);
    // 1
    float focus = pow(distance_color, 2.0);

    ALBEDO = albedo;
    // 2
    ALPHA = alpha * focus;
    ALPHA_ANTIALIASING_EDGE = 0.5;
}

```

Here's what's happening:

- ➊ The value **distance\_color** represents the normalized distance between the object's vertices and the camera. This value is naturally linear, so you square it to create a smoother, more progressive transition near the camera and a stronger falloff farther away.
- ➋ Multiply the original alpha by **focus** to control visibility based on distance. As a result, the regions closest to the camera gradually fade out, while the farther parts remain opaque.

This implementation produces the following visual result:



(4.3.j Transparency based on distance between vertices and camera)

As you can see in Figure 4.3.j, when you move the camera closer to the objects, the transparency behaves correctly. However, from a visual standpoint, the transition edge is still too abrupt, creating hard cuts that make the effect feel less natural.

A common way to soften this behavior is to introduce “noise” into the composition so that the transition between opacity and transparency occurs in a more organic way. This technique is frequently used in dithering effects, atmospheric fade-outs, and stylized transitions, because it breaks the strict linearity of the gradient and prevents the viewer from noticing overly rigid patterns.

To implement this behavior, add the following function above the `fragment()` method:

```
float hash21(vec2 uv)
{
    return fract(sin(dot(uv, vec2(12.9898, 78.233))) * 43758.5453);
}
```

This function generates a pseudo-random value between 0 and 1 from a pair of UV coordinates. Internally, it computes a dot product, applies a sine function, and then extracts the fractional part. Even though this isn’t a physically accurate noise generator, it’s more than sufficient for producing high-frequency variations — perfect for softening edges or generating dithering-style patterns on transparency effects.

Next, you can use the screen coordinates to produce the noise:

```
void fragment()
{
    vec3 albedo = texture(_MainTexRGB, UV).rgb;
    float alpha = texture(_MainTexA, UV).r;

    float distance_color = clamp(distance(vertex_ws, CAMERA_POSITION_WORLD) /
        _Distance, 0.0, 1.0);
    float focus = pow(distance_color, 2.0);
    // 1
    float noise = hash21(SCREEN_UV);
    // 2
    noise = mix(noise, 1.0, focus) * focus;

    ALBEDO = albedo;
    // 3
    ALPHA = alpha * noise;
    ALPHA_ANTIALIASING_EDGE = 0.5;
}
```

Here's what's happening:

- ➊ Declare a new variable named **noise**, whose value comes from **hash21()** applied to **SCREEN\_UV**. This generates static noise uniformly across the screen, letting you break the hard edge of the gradient.
- ➋ Adjust this noise with **mix(noise, 1.0, focus)** and then multiply it by **focus**. This makes the noise more noticeable where the gradient is strongest, while naturally fading it out in areas where the object should remain fully opaque.
- ➌ Finally, you multiply the original **alpha** by the resulting **noise**, creating an irregular pattern that softens the transition between opaque and transparent regions. This eliminates the harsh cutoff seen earlier and produces a more natural fade-out.

When you return to the Viewport, you'll notice that the previously rigid edges now display a dispersed noise pattern, significantly improving the smoothness and visual appeal of the transparency effect.



(4.3.k The transparency edges now contain noise)

At this point, you could consider the effect complete. However, if you want to enhance the visual behavior even further, you can introduce color into the regions affected by the transparency. To do this, declare a new color property as follows:

```
uniform sampler2D _MainTexRGB : source_color;  
uniform sampler2D _MainTexA : source_color;  
uniform float _Distance : hint_range(0.0, 10.0, 0.1);  
uniform vec3 _EdgeColor : source_color;
```

Next, use this property inside the **fragment()** method to create a linear interpolation that blends the object's original color with a custom edge color:

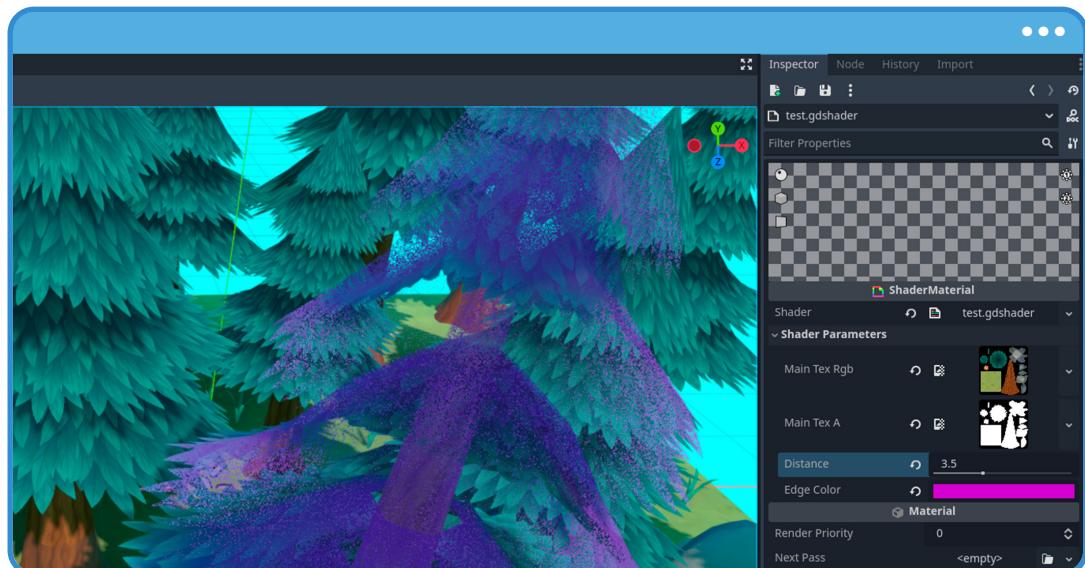
```

void fragment()
{
    ...
    albedo = mix(_EdgeColor, albedo, focus);

    ALBEDO = albedo;
    ALPHA = alpha * noise;
    ALPHA_ANTIALIASING_EDGE = 0.5;
}

```

With this interpolation, fragments that are close to the camera gradually shift toward the color defined in `_EdgeColor`, while more distant areas retain the original texture color. When you return to the Inspector and adjust the value of `_EdgeColor`, you'll see that the chosen color emerges progressively as the camera approaches the 3D object – reinforcing the fade effect with a more expressive visual tint.



(4.3.1 A violet color has been applied to the transparency edge)

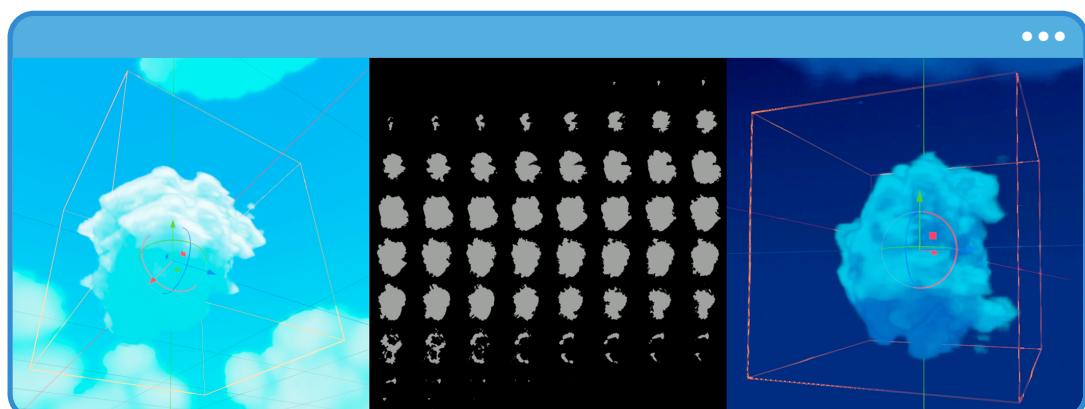
## 4.4 Ray Marching and 3D Textures.

Ray marching is often associated with implicit distance functions, commonly known as SDFs. However, this technique is not limited to that approach. Ray marching can also be combined with **3D textures**, which makes it especially suitable for representing volumetric phenomena such as clouds, smoke, or fog. By using 3D textures, you gain greater artistic control over the final visual result while maintaining a physically inspired rendering model.

At its core, ray marching is a method for generating three-dimensional volumes by casting a ray from each pixel on the screen and advancing it step by step along a given direction. At every step, a function describing the volume is evaluated. Depending on the chosen approach, this function may represent a distance value — as in the case of SDFs — or a density value sampled from a 3D texture.

In this section, the focus is on the use of three-dimensional textures. Here, each ray sample returns a scalar density value. These values are accumulated along the ray's path to simulate physical effects such as light absorption, transmission, and volumetric self-shadowing. These properties are essential for convincingly rendering clouds and other participating media.

Before implementing the algorithm, it is important to understand the fundamental properties of a 3D texture. To do so, take a closer look at the following figure.

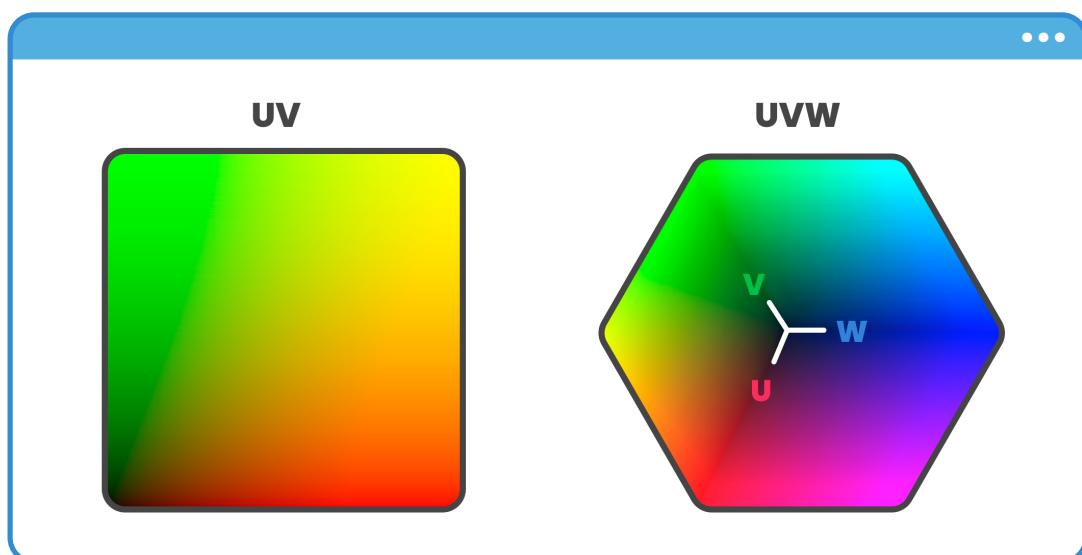


(4.4.a Three-dimensional cloud texture)

This volume can be represented using a 3D texture, which is essentially a block of data with width, height, and depth. In practice, this data is often generated by stacking multiple 2D

“slices” along the depth axis. Alternatively, it can be stored as a single resource that internally organizes these layers to construct the volumetric structure shown in Figure 4.4.a.

To fully understand this concept, you should be comfortable working with UV coordinates. When dealing with 3D textures, a third component is introduced, and you begin working with UVW coordinates (or equivalently XYZ). In this context, the W component represents the depth position inside the volume, allowing the shader to sample values throughout the interior of the texture rather than only on a surface.



(4.4.b Comparison between UV and UVW coordinates in GLSL)

As you already know, when working with a **sampler2D**, each texture sample is defined by two UV coordinates that describe a position on a two-dimensional plane. In contrast, a **sampler3D** requires three coordinates: U, V, and W. This additional component specifies which internal layer of the volume is being sampled — in other words, the depth at which the queried value resides.

You should also keep in mind that adding an extra dimension increases both computational cost and memory usage. Unlike a 2D texture, which can be directly displayed as an image, a 3D texture represents a volume of data. This volume cannot be visualized as a single image without first selecting a slice, projecting it, or traversing it using a sampling technique such as ray marching.

**Note**

In this section, the focus is on using a 3D texture inside the shader, so the texture creation process itself will not be covered in depth. If you want to understand how these textures are built from stacked 2D slices, you can refer to a tutorial that demonstrates the workflow and layer organization: [https://www.youtube.com/shorts/lGx\\_l4nzfxQ](https://www.youtube.com/shorts/lGx_l4nzfxQ)

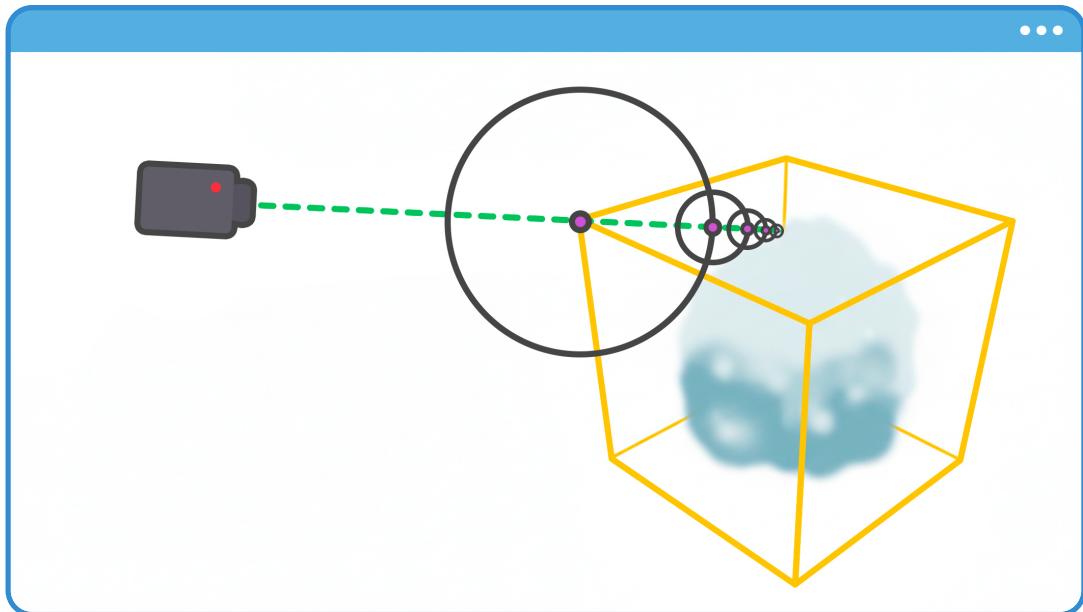
3D textures are typically stored in formats that preserve volumetric information accurately. For example, **.exr** is a common choice when high precision or HDR values are required. This is especially useful when the texture stores density, noise, or intermediate data that must be sampled with high fidelity.

In Godot, it is also possible to use **.png** images. However, you should keep in mind that **.png** is usually an LDR format (often 8 bits per channel), which can introduce quantization and banding when representing smooth density transitions. For this reason, if you are aiming for stable results with minimal artifacts, higher-precision formats are generally preferable.

Now, how do you render a volume defined by a 3D texture inside a shader? This is where ray marching comes into play.

Ray marching allows the shader to traverse the volume step by step along a ray cast from each pixel. At every step, the 3D texture is sampled to retrieve a density value. Using this information, the shader integrates the medium's behavior – such as absorption, transmission, and self-shadowing. Through this process, the visual appearance of the desired volumetric object is progressively reconstructed.

To better understand this workflow, take a look at the following figure.



(4.4.c Ray marching in action)

To implement this technique in your shader, you need to account for several key factors. These include the ray direction, the distance the ray is allowed to travel, and how the ray interacts with the volume to compute accumulated density as a function of distance from the camera. In this example, the cloud is contained inside a generic cube, so the shader also needs access to the world-space position of the object acting as the volume container.

Before jumping into the shader implementation, you will organize the project so you can work with the required resources in a clear and consistent way.

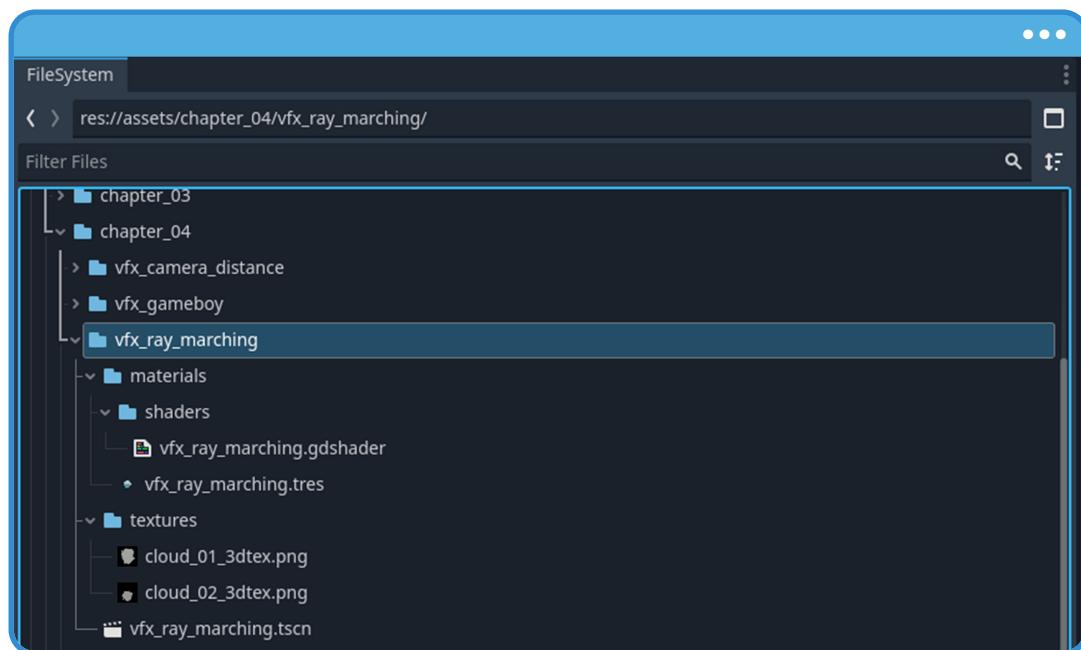
Follow these steps:

- 1 Inside the **chapter\_04** folder, create a subfolder named **vfx\_ray\_marching**. This folder will store all resources used in this section.
- 2 Inside **vfx\_ray\_marching**, create three additional subfolders:
  - a **materials**.
  - b **shaders**.
  - c **textures**.

Following the book's usual structure, you will create both a **Shader** and a **ShaderMaterial** for this section. For consistency and practical reasons, name both of them **vfx\_ray\_marching**.

You will also include the 3D textures required for the effect, which can be downloaded from the project files that accompany this book.

If everything has been set up correctly, your project structure should match the one shown in the next figure.



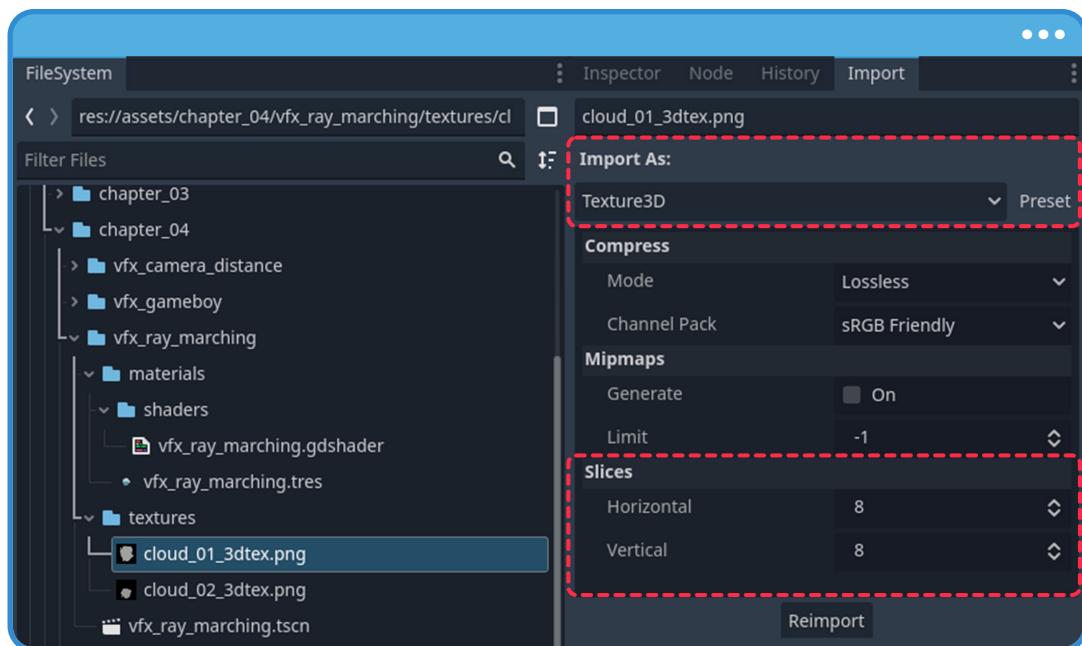
(4.4.d Project structure)

The first step is to set up the scene where the volume will be rendered. To do this, you will add a **BoxMesh** to the scene, which will act as the container for the volume. As mentioned earlier, the cloud will be rendered inside this cube, so its geometry defines the spatial boundaries of the volumetric effect.

The next step involves configuring the 3D textures. When you import images into Godot, they are configured as **Texture2D** by default. As a result, if the texture is composed of multiple images, they are displayed as a grid. However, this is not the configuration required when working with a three-dimensional volume.

To convert these images into a 3D texture, follow this procedure:

- 1 Select the image that represents the volumetric texture.
- 2 Go to the **Import** window.
- 3 From **Import As**, select **Texture3D**.
- 4 Set the number of **Slices** to **8 horizontally** and **8 vertically**, matching the layout of the images.
- 5 Finally, click Reimport to apply the changes.



(4.4.e The texture has been configured as a **Texture3D**)

Once the texture is correctly imported as a **Texture3D**, you can start building the shader. Begin by declaring a small set of global uniforms. You will use them later to compute the final color and to evaluate the volume during the ray-marching process.

```

shader_type spatial;
render_mode unshaded;

uniform sampler3D _BaseTex : source_color, repeat_disable;
uniform vec3 _ColorLight : source_color;
uniform vec3 _ColorShadow : source_color;

void vertex() { ... }

```

The `sampler3D _BaseTex` represents the volumetric texture that stores the medium's density, and it will be sampled repeatedly along the ray. The `_ColorLight` and `_ColorShadow` uniforms define a two-tone color range that the shader will blend based on density and lighting cues. This is a practical way to make the volume's shape read clearly and to enhance the perception of depth.

Notice that the shader is declared as `unshaded`. This means Godot's standard lighting pipeline will not be applied automatically. That said, the volume is not "unlit." Instead, the lighting will be computed explicitly inside the shader using a simplified model. This manual approach keeps the effect predictable and controllable, while still reinforcing the volume's three-dimensional structure.

**Note**

The ray-marching implementation presented in this section is based on a solution originally created by **DMeville**, which has been adapted and ported to Godot for the sake of simplicity and pedagogical clarity. You can find the original implementation in the following repository: <https://github.com/DMeville/DMVolumeRendering?tab=readme-ov-file>

Next, you will define a set of global constants that will be used later during the rendering of the 3D cloud. These constants directly control both the visual quality and the computational cost of the ray-marching algorithm, so configuring them correctly is essential.

```

render_mode unshaded;
// 1
#define NUM_STEPS 128
// 2
#define STEP_SIZE 0.01
// 3
#define NUM_LIGHT_STEPS 6
// 4
#define LIGHT_STEP_SIZE 0.05

```

What is the role of these constants?

- 1 **NUM\_STEPS:** This value defines how many samples the view ray takes as it travels through the volume. Each step corresponds to one density evaluation of the 3D texture. Higher values produce smoother and more detailed volumes, but they also increase the processing cost.
- 2 **STEP\_SIZE:** This constant determines how far the view ray advances at each iteration. Together with **NUM\_STEPS**, it defines the total distance the ray can travel inside the volume. Smaller step sizes allow for finer sampling, while larger steps reduce the number of effective evaluations.
- 3 **NUM\_LIGHT\_STEPS:** This value specifies how many steps are used to evaluate volumetric lighting. In this case, a secondary ray is cast toward the light source to estimate how much density blocks its path. This process allows the shader to simulate volumetric self-shadowing.
- 4 **LIGHT\_STEP\_SIZE:** This constant defines how far the light ray advances at each step. It controls the reach and softness of volumetric shadows. Larger step sizes result in softer but less accurate shadows, while smaller values produce sharper results at the cost of additional computation.

Begin by defining the method that will render the volumetric cloud. To do this, place the function between the **vertex()** and **fragment()** methods, and declare a new function called **ray\_marching()**, as shown below:

```
vec3 ray_marching()
{
    return vec3(0.0);
}

void fragment() { ... }
```

As you can see, this new function does not perform any calculations yet. For now, it simply returns black. This helps you establish the structure of the algorithm before introducing the actual ray marching logic.

Next, add the core variables you will use to describe how light interacts with the cloud volume:

```
vec3 ray_marching()
{
    float density = 0.0;
    float transmission = 0.0;
    float light_accumulation = 0.0;
    float final_light = 0.0;

    return vec3(0.0);
}
```

These variables represent key quantities in volumetric rendering. The **density** variable stores the accumulated density sampled along the view ray as it travels through the volume. The **transmission** variable represents how much light is able to pass through the medium after interacting with that density. The **light\_accumulation** variable estimates how much density exists between a point inside the volume and the light source, which is what enables volumetric self-shadowing. Finally, **final\_light** represents the final integrated lighting contribution that will drive the fragment's color.

Together, these quantities are fundamental for modeling how light is absorbed, transmitted, and attenuated inside a participating medium such as a cloud.

**Note**

These variables are part of the physical foundation of volumetric rendering. If you want to dive deeper into the principles that govern how light interacts with volumes, we recommend watching Sebastián Lague's video, which explains the process in a clear and visual way: <https://youtu.be/4QOcCGI6xOU?si=WdTzlr6LEcw8SgKV>

As mentioned at the beginning of this section, defining the cloud's volume requires advancing along a given direction using successive steps while evaluating the density stored in the 3D texture. When referring to "successive steps," this directly maps to the use of a **for()** loop, which allows the shader to incrementally traverse the volume along the view ray.

At this stage, the main goal is to sample and accumulate the volume's density. You can achieve this as follows:

```
vec3 ray_marching(vec3 ray_origin, vec3 ray_direction, vec3 offset)
{
    float density = 0.0;
    float transmission = 0.0;
    float light_accumulation = 0.0;
    float final_light = 0.0;
    // 1
    for (int i = 0; i < NUM_STEPS; i++)
    {
        // 2
        ray_origin += (ray_direction * STEP_SIZE);
        // 3
        vec3 sampled_position = ray_origin + offset;
        // 4
        float sample_density = texture(_BaseTex, sampled_position).r;
        // 5
        density += sample_density;
    }

    return vec3(0.0);
}
```

What is happening in this code?

- 1 The `for()` loop controls how many samples the ray takes as it travels through the volume. Each iteration represents one ray-marching step.
- 2 On each iteration, the ray origin is advanced in the direction given by `ray_direction`, moving a constant distance defined by `STEP_SIZE`. This value directly affects the spatial resolution of the sampling.
- 3 The position used to sample the 3D texture is computed next. The `offset` parameter is used to correctly align the volume in world space, ensuring that the cloud remains centered inside the `BoxMesh`.
- 4 The 3D texture `_BaseTex` is sampled using `sampled_position` as UVW coordinates. The returned value represents the density of the volume at that specific point in space.
- 5 The sampled density is added to the accumulated density value. This discrete sum approximates the total amount of matter the ray has passed through so far and will later be used to compute transmission, absorption, and volumetric self-shadowing.

Ultimately, this code segment performs an incremental traversal of the view ray through the volume. At each step, the shader samples the density stored in the 3D texture and accumulates its contribution. You also introduced additional parameters that will later let you control the effect more precisely from `fragment()`.

The next step is to compute volumetric lighting, along with transmission and internal shadowing. To do this, you will run a second sampling pass – this time marching in the direction of the light – as shown below:

```
vec3 ray_marching(vec3 ray_origin, vec3 ray_direction, vec3 offset, vec3
    ↵  light_direction)
{
    ...
    for (int i = 0; i < NUM_STEPS; i++)
    {
        ...
    }
}
```

// The code continues on the next page.

```

// 1
vec3 light_ray_origin = sampled_position;
// 2
for(int j = 0; j < NUM_LIGHT_STEPS; j++)
{
    // 3
    light_ray_origin += (light_direction * LIGHT_STEP_SIZE);
    // 4
    float light_density = texture(_BaseTex, light_ray_origin).r;
    // 5
    light_accumulation += light_density;
}
}

return vec3(0.0);
}

```

Here is what is happening:

- ➊ Initialize the light ray: You start the ray toward the light at the current view-ray sample position `sampled_position`. This creates a secondary ray that probes the volume between the current point and the light source.
- ➋ This inner loop defines how many samples are taken in the light direction. Increasing `NUM_LIGHT_STEPS` generally improves shadow stability and detail, but it also increases the cost.
- ➌ You move the light-ray sampling point along `light_direction` using a fixed step size defined by `LIGHT_STEP_SIZE`.
- ➍ The shader samples `_BaseTex` at the current light-ray position to retrieve the local density along the light path.
- ➎ Finally, each sampled density value is added to `light_accumulation`. The more density accumulated along this light ray, the less light reaches the original point, which results in stronger volumetric shadowing.

Because `light_accumulation` can keep growing across multiple steps of the main ray, you can apply an optional optimization by resetting it inside the outer `for()` loop. This makes the light occlusion computation local to each sampled point, which stabilizes the shading and usually produces a cleaner result.

Next, you use the density accumulated along the light ray to estimate how much illumination reaches the current point inside the volume. With that value, you build a shadow factor, clamp it with a minimum darkness level, add it into the accumulated lighting, and simultaneously update the view-ray transmittance — that is, how much light energy still “survives” after traveling through the medium.

```
vec3 ray_marching(vec3 ray_origin, vec3 ray_direction, vec3 offset, vec3
    ↵  light_direction, float darkness, float transmittance, float light_absorb)
{
    ...

    for (int i = 0; i < NUM_STEPS; i++)
    {
        ...
        // 1
        float light_accumulation = 0.0;

        for(int j = 0; j < NUM_LIGHT_STEPS; j++)
        {
            ...
        }

        // 2
        float light_transmition = exp(-light_accumulation);
        // 3
        float shadow = darkness + light_transmition * (1.0 - darkness);
        // 4
        final_light += density * transmittance * shadow;
        // 5
        transmittance *= exp(-density * light_absorb);
    }

    // 6
    transmission = exp(-density); // exp(-density * LIGHT_STEP_SIZE)
    // 7
    return vec3(final_light, transmission, transmittance);
}
```

What is happening here?

- 1 The `light_accumulation` is restarted for every sample along the view ray. This ensures the computed shadow represents only the local occlusion between the current point and the light source, instead of being polluted by values carried over from earlier view-ray steps.
- 2 We transform the accumulated density along the light ray into a transmission value using an exponential falloff. When `light_accumulation` is large,  $\exp(-x)$  becomes small, meaning less light reaches the point and the shading becomes stronger.
- 3 We compute `shadow` by blending the transmission with the `darkness` parameter. Here, `darkness` acts as a minimum shadow brightness: even if transmission is near zero, the volume retains a baseline amount of light. This prevents completely black regions and improves readability.
- 4 We add the lighting contribution of the current sample to `final_light`. This contribution is modulated by the sampled `density`, the current `transmittance`, and the computed `shadow`.
- 5 We reduce the view-ray transmittance using an absorption term. The `light_absorb` parameter controls how quickly the medium “consumes” light: higher density and higher absorption produce a faster drop in transmittance.
- 6 We compute an overall transmission estimate from the total accumulated density. If you multiply density by a step length (such as `LIGHT_STEP_SIZE` or the appropriate step size for your integration), you can better control how opacity scales with sampling distance.
- 7 The function returns three values: the accumulated lighting `final_light`, the global transmission estimate, and the progressive view-ray `transmittance`, which will be used later to build the final fragment color and alpha.

With this, we complete the `ray_marching()` implementation, and you now have a function that can sample density and estimate lighting inside the volume.

Next, you will call this function from the fragment stage to evaluate the volume per pixel on the screen. To do this, go to `fragment()` and initialize the following variables:

```

void fragment()
{
    // 1
    vec3 ray_origin_ws = (INV_VIEW_MATRIX * vec4(VERTEX.xyz, 1.0)).xyz;
    // 2
    vec3 ray_direction = normalize(ray_origin_ws - CAMERA_POSITION_WORLD);
    // 3
    vec4 transform = (MODEL_MATRIX * vec4(0.0, 0.0, 0.0, 1.0));
    // 4
    vec3 offset = (vec4(0.5, 0.5, 0.5, 0.0) - transform).xyz;

    ALBEDO = vec3(0.0);
    ALPHA = 1.0;
}

```

If you look closely, here is what each step is doing:

- ➊ We compute `ray_origin_ws` in world space using `INV_VIEW_MATRIX` to obtain a world-space position from the data available inside the shader.
- ➋ We compute the ray direction from the camera to the current world-space point and normalize it. Normalization ensures the vector has length 1, so `STEP_SIZE` corresponds to a constant distance per iteration during ray marching.
- ➌ Because the cloud is rendered inside a `BoxMesh`, you need the world-space position of the object's origin (its pivot). You get it by multiplying the local origin  $(0, 0, 0, 1)$  by `MODEL_MATRIX`. This anchors the volume to the `BoxMesh`, so moving the cube in the scene moves the cloud with it.
- ➍ We compute offset to keep the volume sampling centered within the container. The vector  $(0.5, 0.5, 0.5)$  compensates for the volume's center, and subtracting the object's world position (`transform`) aligns the sampling coordinate system with the container.

In addition, as part of the setup, we temporarily assign values to `ALBEDO` and `ALPHA`. Think of this as a starting point: in the next steps, you will replace both values with the final result produced by the ray-marching evaluation.

To continue, declare and initialize the remaining arguments required by `ray_marching()`. Specifically, you will define the light direction, a minimum lighting level for shadowed regions,

the ray's initial transmittance, and the medium's absorption coefficient. Finally, you will run the method for the current pixel and use its output to drive the fragment color.

```
void fragment()
{
    ...

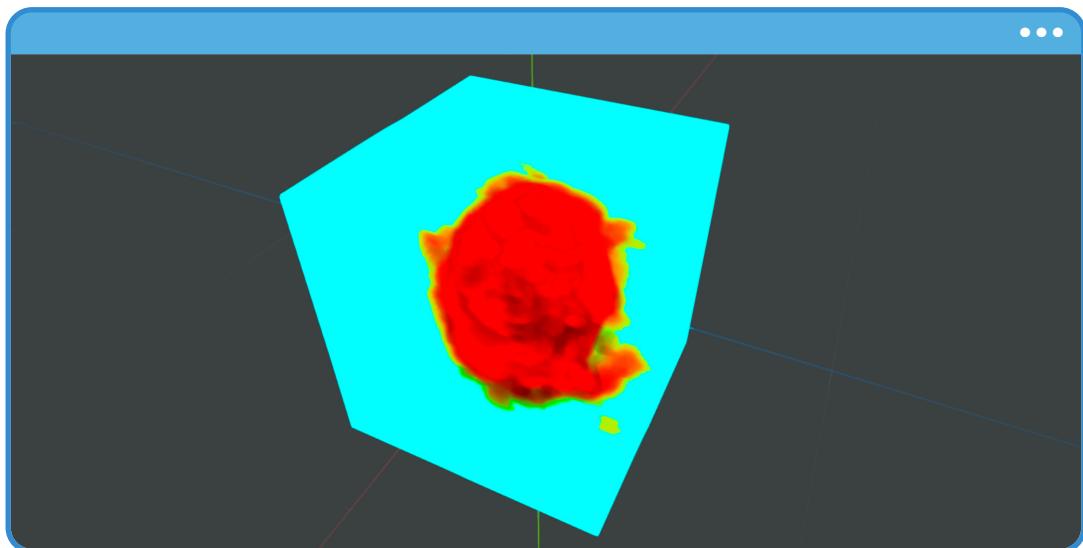
    // 1
    const vec3 light_direction = vec3(0.0, 1.0, 0.0);
    // 2
    const float darkness = 0.19;
    // 3
    const float transmittance = 1.0;
    // 4
    const float light_absorb = 1.5;
    // 5
    vec3 render = ray_marching(
        ray_origin_ws,
        ray_direction,
        offset,
        light_direction,
        darkness,
        transmittance,
        light_absorb
    );

    // 6
    ALBEDO = render;
    ALPHA = 1.0;
}
```

Here is what each part is doing:

- 1 You define the direction from which light hits the volume. In this example, you use the  $+y$ -axis, so the light comes from above. In a more complete implementation, this direction could be derived from Godot's lighting data (for example, via the **LIGHT** direction in the lighting stage, depending on your setup).

- 2 This value sets the minimum amount of light in shadowed regions. With **darkness = 0.0**, the volume can produce much deeper shadows. With **darkness = 1.0**, shadowing is effectively removed because the shadow term becomes constant.
- 3 This defines how much light “survives” before the ray starts traveling through the volume. A value of 0.0 makes the cloud completely dark from the start.
- 4 This parameter controls how quickly the medium absorbs light. Higher values cause the volume to absorb more light over a shorter distance, making the interior darken more aggressively.
- 5 We call **ray\_marching()** using the per-pixel ray origin and direction, along with the volume alignment offset and lighting parameters.
- 6 We assign the result to **ALBEDO**, which determines the fragment’s final color. At this stage, you keep **ALPHA** fixed at 1.0 so you can inspect the effect directly on the BoxMesh, as shown in Figure 4.4.f.



(4.4.f Rendering using **transmission = exp(-density)**)

To complete the rendering process, you only need to compute the fragment’s final color using the two global color properties, and then determine the volume’s transparency. To do this, you will use the values returned by **ray\_marching()**, which let the shader control both the color variation and the opacity of each pixel based on the integrated density.

```

void fragment()
{
    ...

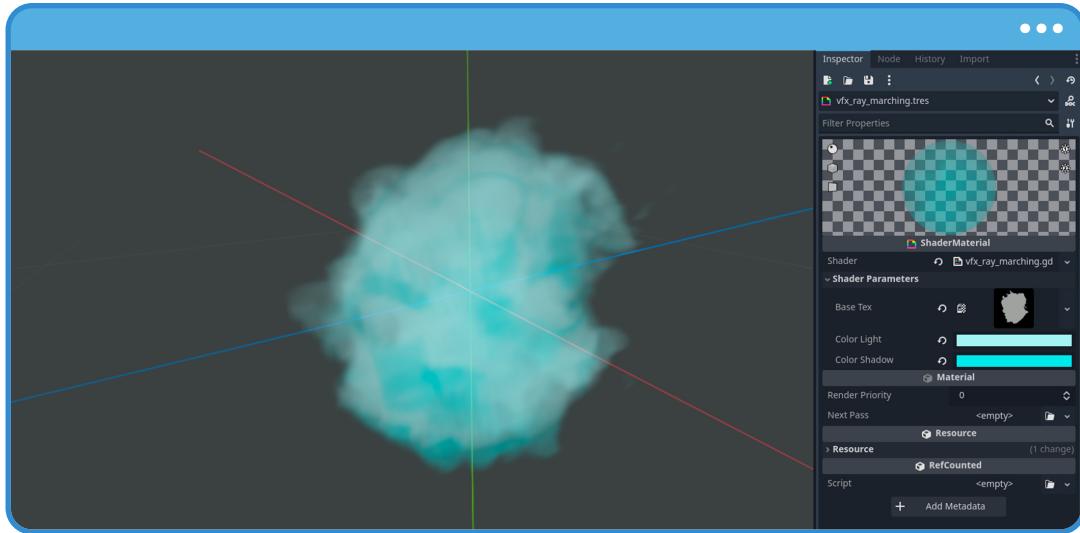
    // 1
    float gradient = clamp(render.x, 0.0, 1.0);
    // 2
    vec3 albedo = mix(_ColorShadow, _ColorLight, gradient);
    // 3
    float alpha = (1.0 - render.y);
    // 4
    ALBEDO = albedo;
    ALPHA = alpha;
}

```

What is happening here?

- ➊ We take the X component of **render**, which represents the accumulated lighting (or a density-driven light term) along the ray. You then clamp it to the [0:1] range so it can safely be used as an interpolation factor.
- ➋ We linearly interpolate between **\_ColorShadow** and **\_ColorLight** using **gradient**. This makes brighter regions of the cloud lean toward **\_ColorLight**, while darker regions are influenced by **\_ColorShadow**.
- ➌ We interpret the Y component of **render** as the accumulated transmission through the volume. By inverting it, we obtain an opacity value that matches the expected physical behavior.
- ➍ We assign the computed color to **ALBEDO** and the computed opacity to **ALPHA**, completing the volumetric rendering of the fragment in the viewport.

At this point, the effect is fully parameterized and functional. From the Inspector, you can easily adjust the colors associated with lower and higher density regions using **Color Light** and **Color Shadow**, allowing you to iterate visually without making further changes to the shader code.



(4.4.g Final Render,  $\text{transmission} = \exp(-\text{density} * \text{LIGHT\_STEP\_SIZE})$ )

## 4.5 Introduction to Stencil Buffer.

When working with special effects, the stencil buffer is one of the most versatile and powerful mechanisms you can use in your projects. Its primary purpose is to mask specific regions of the screen, enabling a wide range of visual effects such as portals, dimensional transitions, selective visibility, outlines, custom masks, and more.

Conceptually, the stencil buffer operates as an auxiliary per-pixel buffer, independent of both the color buffer and the depth buffer. Each pixel on the screen stores an unsigned integer (**uint**) value in the stencil buffer, which can be read from or written to during the rendering process.

It uses 8 bits per pixel, allowing values in the range of 0 to 255. By default, all pixels start with a value of 0, and this value remains unchanged until a shader or a specific stage of the rendering pipeline explicitly modifies it.

To use the stencil buffer correctly, you must consider several key aspects:

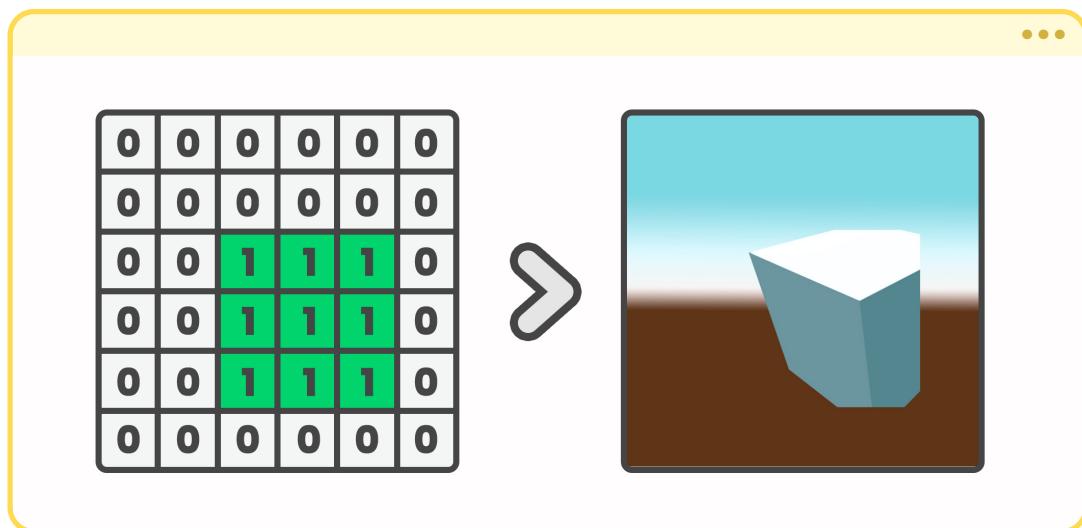
- The type of comparison performed between the value stored in the stencil buffer and the reference value defined in the shader.
- Whether fragments are discarded or kept, which directly affects transparency and clipping behavior.

- The state of depth testing and how it interacts with the stencil buffer, especially in scenes composed of multiple layers of geometry.
- The blending mode, when the result is combined with previously rendered objects.
- The rendering order (sorting), since stencil operations depend heavily on what is drawn first and what is evaluated afterward.

In Godot, the `stencil_mode` property in a shader allows you to either write to or read from the stencil buffer. A common workflow is to split the process into two distinct stages:

- **Write stage:** An object or piece of geometry is rendered as a mask, assigning a specific value to the stencil buffer, such as 1 (or any value other than 0).
- **Read stage:** A second object is then rendered only if the stencil buffer value at that pixel matches the expected reference value (in this case, 1).

For example, when using an **equal** comparison, the result is that the second object becomes visible only within the previously masked area. This produces a precise clipping effect, fully controlled by the geometry used as the stencil mask.



(4.5.a Applying a mask to a BoxMesh)

As shown in Figure 1.5.b from Chapter 1, the stencil test takes place after fragment processing in the fragment shader stage and before the depth test. Understanding this order is essential, because it allows you to later analyze and resolve visual issues related to object depth relative to the camera.

From a conceptual standpoint, both the stencil test and the depth test belong to the stage known as **per-fragment tests**, which are evaluated after the fragment shader has executed. In this logical model, the rendering flow always follows the same sequence: the fragment color is computed first, and only then is it determined whether that fragment is valid or must be discarded.

In practice, however, you should be aware that modern GPUs may reorder these operations through optimizations such as early depth testing or early stencil testing. These techniques allow the GPU to evaluate stencil or depth conditions before running the fragment shader, provided that the shader state and pipeline configuration allow it. This behavior is not guaranteed and depends on several factors, including the use of **discard**, depth writes, and specific blending settings.

For this reason, from a developer's perspective, you should always assume the following execution model: the fragment shader runs first, then the stencil test is evaluated, and finally the depth test is performed. If a fragment fails any of these tests — or is explicitly discarded — no color value is written to the **framebuffer**, causing that fragment to be absent from the final rendered image.

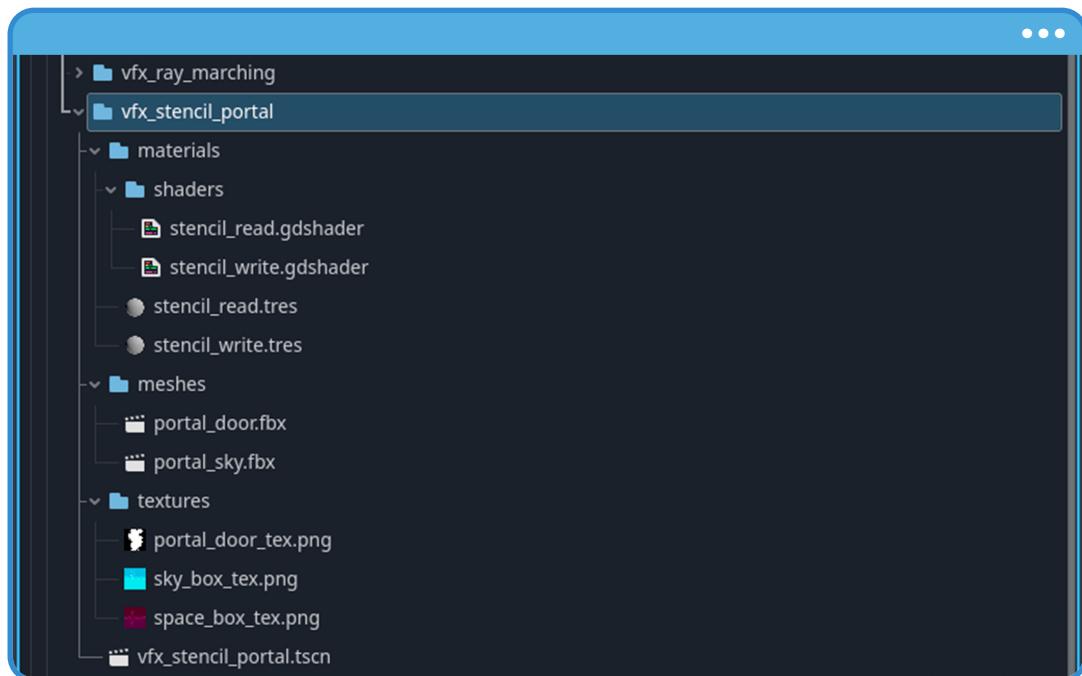
**Note**

The framebuffer refers to the memory region where the GPU stores the final result of the rendering process before the image is presented on screen. This buffer contains, among other data, the color buffer, the depth buffer, and the stencil buffer.

To see these concepts in practice, you will proceed as follows:

- Inside the **chapter\_04** folder, create a subfolder named **vfx\_stencil\_portal**. This folder will store all the resources used in this section.
- Inside this folder, create the following subfolders:
  - **material**
  - **shaders**
  - **meshes**
  - **textures**

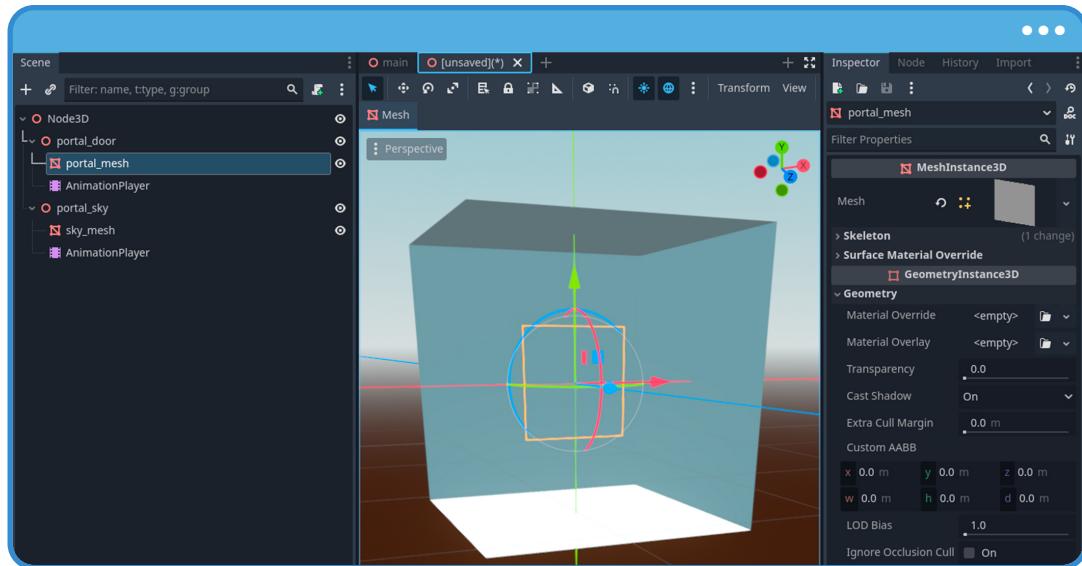
For this section, you will include two shaders and two **ShaderMaterial** resources, named **stencil\_write** and **stencil\_read**, respectively. You will also add the textures and 3D objects required for the effect developed throughout this section. All of these resources can be downloaded from the project files provided with this book. If all steps have been completed correctly, the project structure should match the layout shown in the following figure.



(4.5.b Project structure)

The first step is to organize the scene. To do this, create a new 3D scene and drag the **portal\_door** and **portal\_sky** objects into it, making sure that both their position and rotation are set to  $(0, 0, 0)$ . You should also mark both objects as Make Local, which allows you to directly access and modify the Mesh property of each one.

In this workflow, the **portal\_door** object is used as the mask, while **portal\_sky** is rendered only within the area defined by the portal's geometry.



(4.5.c Both objects centered in the Scene)

Next, assign the **stencil\_write** material and its corresponding shader to the **Material Override** property of the **portal\_mesh**, as shown in Figure 4.5.c. Then, repeat the process by assigning the **stencil\_read** material and its associated shader to the **sky\_mesh** object, which is a child of **portal\_sky**.

The goal of this workflow is to clearly separate the process into two distinct stages: first, defining the masked area by writing to the stencil buffer, and then restricting the rendering of another object to that predefined region.

You will begin by working on the **stencil\_write** shader, which is responsible for writing a specific value into the stencil buffer. This value defines the region that will later act as a mask for the **portal\_sky** object. To achieve this, add the following lines of code:

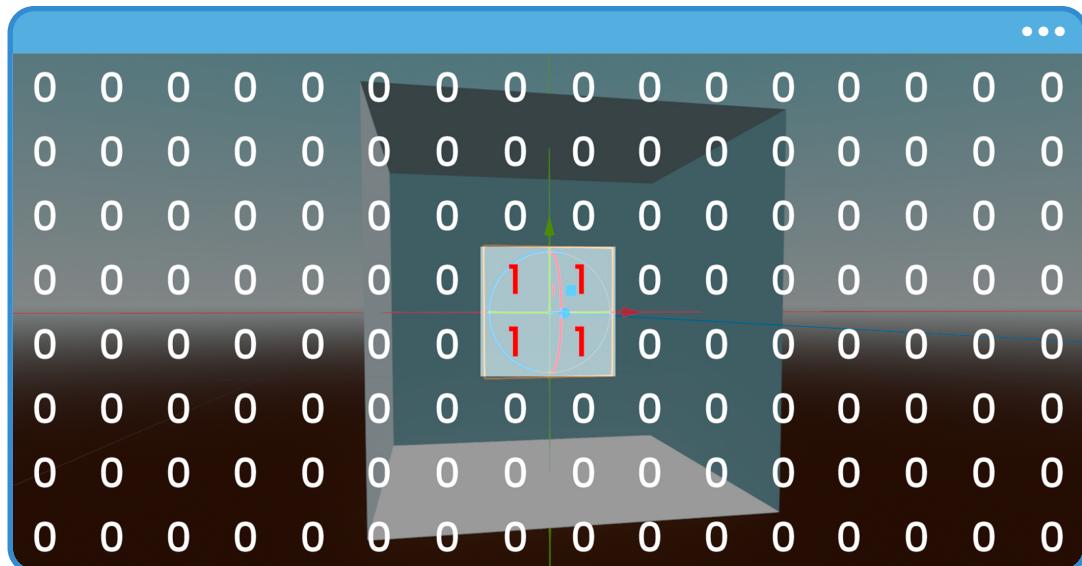
```
1 shader_type spatial;
2 render_mode unshaded;
3 stencil_mode write, 1;
```

Focusing on the third line, `stencil_mode write, 1`, this instruction defines how the shader interacts with the stencil buffer through two clearly defined parameters:

- 1 The first parameter, `write`, indicates that this shader writes a value to the stencil buffer rather than reading from it.
- 2 The second parameter, `1`, specifies the unsigned integer (`uint`) value that will be stored in the stencil buffer for each fragment rendered by this material.

In practical terms, every fragment generated by `portal_mesh` writes the value 1 into the stencil buffer, precisely defining the area that will later be used as a mask. In the next stage, this value is evaluated by the `stencil_read` shader, allowing the `sky_mesh` object to render only in fragments where the stencil buffer contains that value.

From a visual standpoint, this process can be understood conceptually as illustrated below:



(4.5.d The value 1 assigned to each fragment of `portal_mesh`)

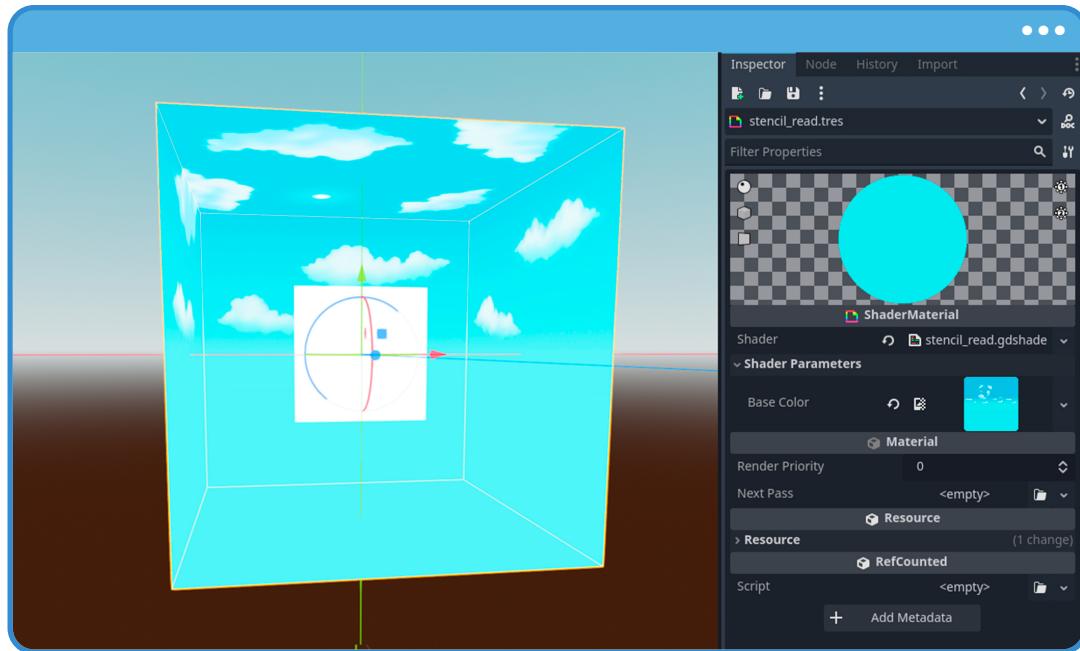
You can infer that the final image contains far more pixels (or fragments) than those shown in Figure 4.5.d. However, this simplified representation helps you clearly conceptualize the result of the process. By default, the stencil buffer assigns a value of 0 to every pixel on the screen, while the value 1 has been written only to the fragments corresponding to the area covered by `portal_mesh`.

At this stage, if you configure a shader to compare the stencil buffer value in the pixels covered by the `sky_mesh` object, that object will be visible only within the boundaries defined by **portal\_mesh**. To make this behavior easier to observe, proceed as follows: open the **stencil\_read** shader and add the code below. Initially, a texture is introduced to help visualize the result.

```
1 shader_type spatial;
2 render_mode unshaded;
3
4 uniform sampler2D _BaseColor : source_color, repeat_disable;
5
6 void fragment()
7 {
8     vec3 albedo = texture(_BaseColor, UV).rgb;
9     ALBEDO = albedo;
10 }
```

As shown above, a texture named `_BaseColor` is defined and used to assign the **ALBEDO** color in the fragment stage. This type of implementation has already been covered in earlier sections, so here you only need to make sure that the `sky_box_tex` texture is assigned to the corresponding property in the **stencil\_read** material.

If all steps have been completed correctly, you should obtain the following visual result:

(4.5.e Texture assigned to the **stencil\_read** Material)

At this point, you already know that the fragments covering the **portal\_mesh** object store the value 1 in the stencil buffer. Therefore, if you configure the shader to compare against that value, the sky rendered by the **sky\_mesh** object will be visible only within the area defined by the portal. To verify this behavior in practice, apply the following configuration to the **stencil\_read** shader:

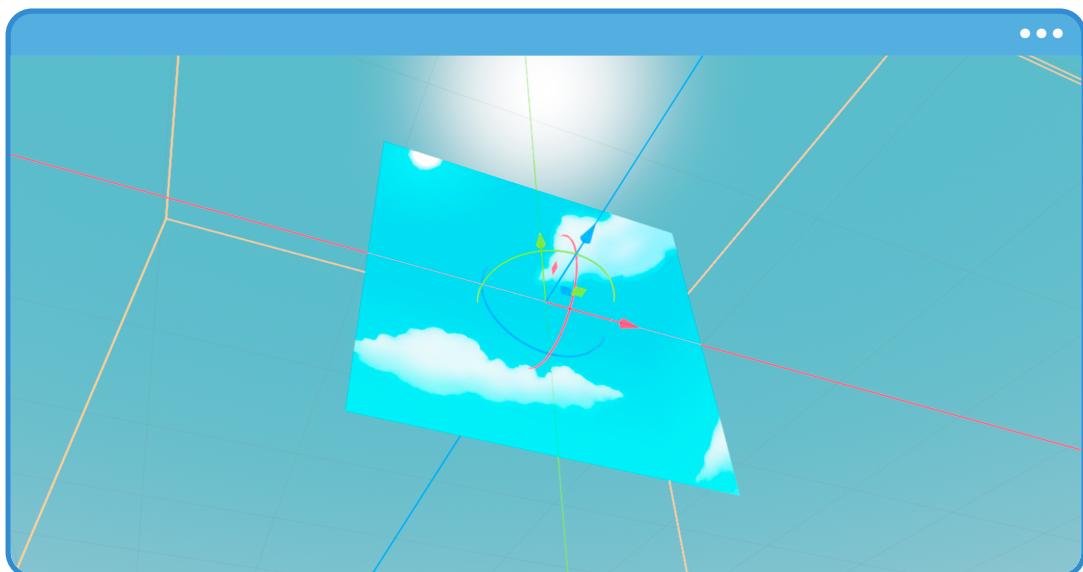
```

1  shader_type spatial;
2  render_mode unshaded, depth_draw_never, depth_test_disabled;
3  stencil_mode read, compare_equal, 1;
4
5  uniform sampler2D _BaseColor : source_color, repeat_disable;
6
7  void fragment()
8  {
9      vec3 albedo = texture(_BaseColor, UV).rgb;
10     ALBEDO = albedo;
11 }
```

What is happening here? Let's break down the key lines of code:

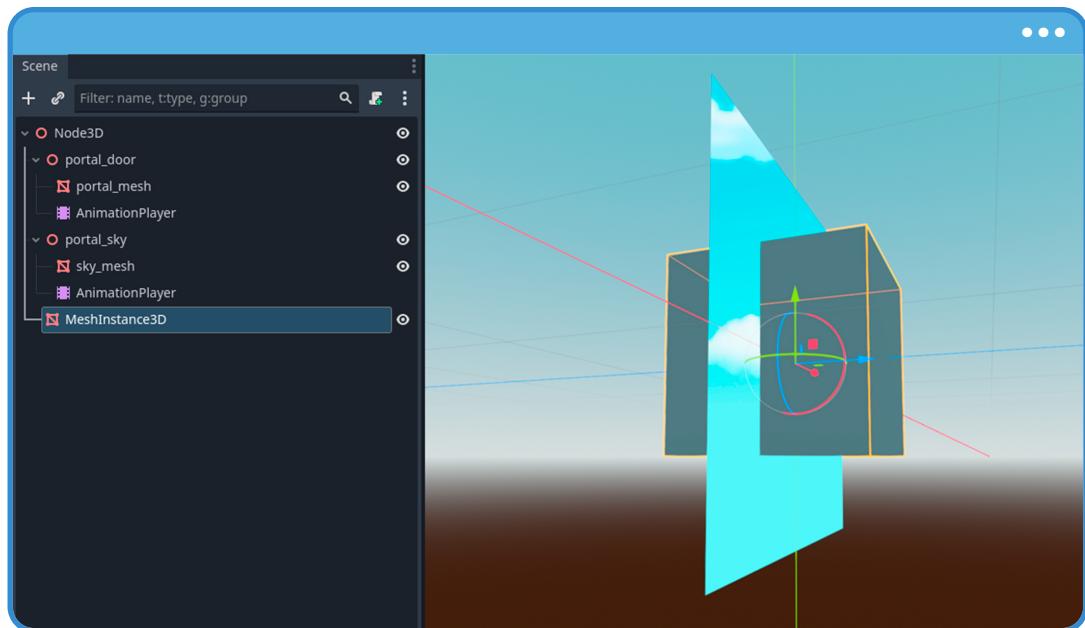
- Line 2: Two additional `render_mode` properties are introduced. First, `depth_draw_never` prevents this object from writing values to the depth buffer, which is evaluated later in the render pipeline. This ensures that the sky does not affect the depth of other objects rendered afterward. Second, `depth_test_disabled` disables depth comparison entirely, allowing the sky to be drawn regardless of its distance from the camera.
- Line 3: This instruction defines how the shader interacts with the stencil buffer. The keyword `read` indicates that the shader reads from the stencil buffer without modifying it. The `compare_equal` operation specifies that a fragment is accepted only if the value read from the stencil buffer matches the reference value — in this case, 1.

As a result, only fragments whose stencil value is equal to 1 pass the test. Since this value was previously written by `portal_mesh`, the sky is rendered exclusively within the portal's area, producing the intended visual effect.



(4.5.f The `sky_mesh` object is visible inside `portal_mesh`)

To better understand how depth-related properties interact with comparison mechanisms, you can run a simple practical exercise. Add a new `MeshInstance3D` to the scene and place it at the center, alongside the objects you have already configured.

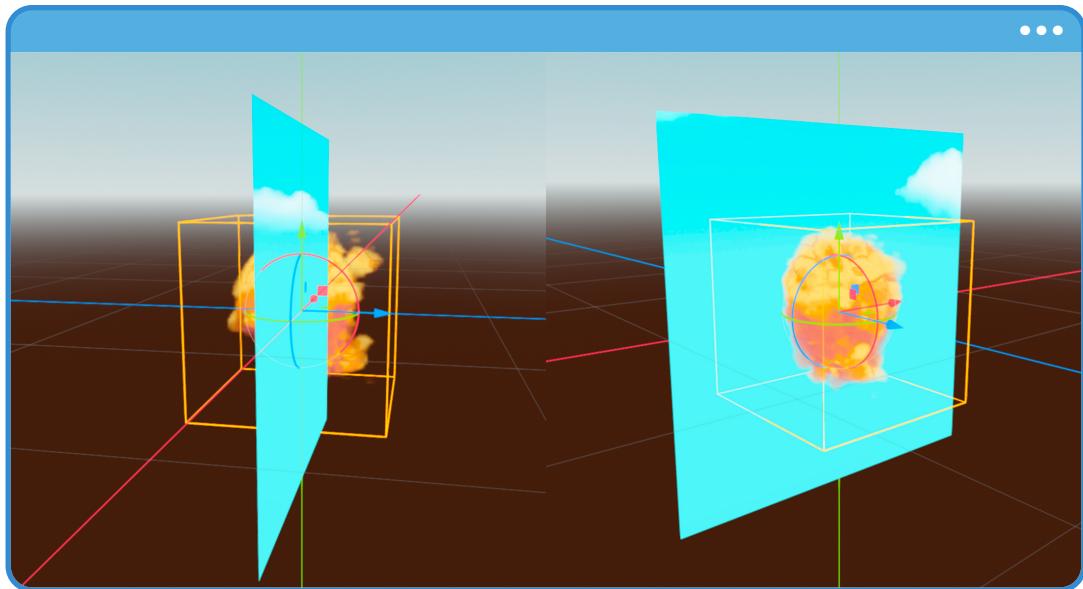


(4.5.g A new BoxMesh has been added to the Scene)

As you can see, the new BoxMesh is visible outside the area defined by **portal\_mesh**. This happens because the depth buffer continues to store depth values for every rendered fragment, and the depth test determines visibility relative to the camera and the rest of the scene geometry.

If you disable depth testing, prevent depth writes, or introduce an additional constraint through the stencil buffer, the visual behavior of this BoxMesh will change.

To examine this effect in more detail, perform the following exercise: assign the **vx\_ray\_marching** material used in the previous section to the newly created BoxMesh.

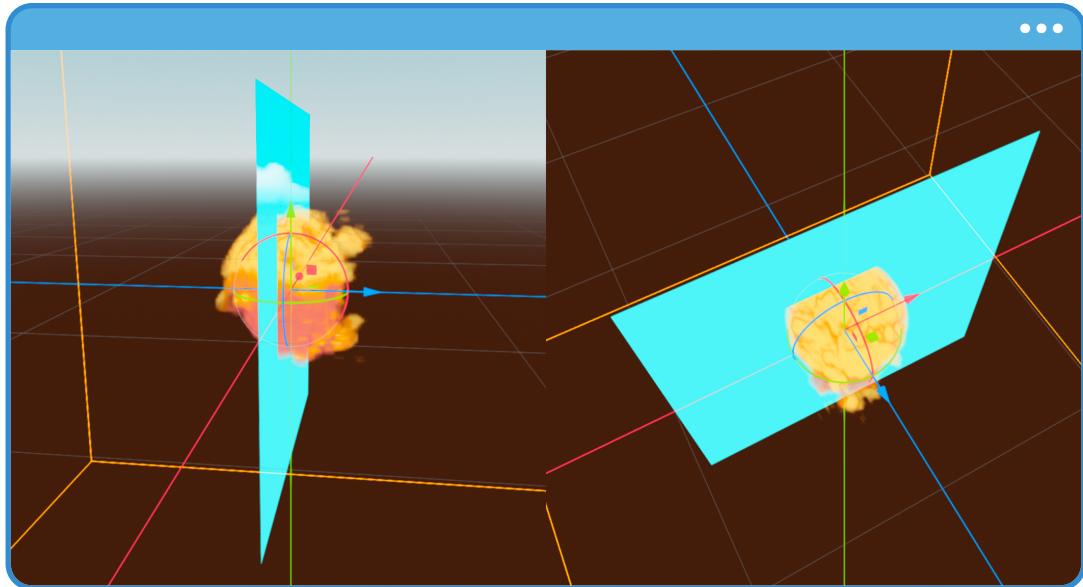


(4.5.h Depth is not perceived in the clouds)

If you move the camera around the cloud, you will notice a visual artifact caused by depth testing. This happens because the BoxMesh, as traditional geometry, performs depth tests for each fragment it generates, while the ray-marched cloud does not write depth values into the depth buffer (you would have to compute and output depth manually). As a result, you may see flickering as the camera rotates, because the GPU cannot reliably resolve which element should appear in front.

You can partially mitigate this behavior by adjusting the render order, a concept known as **sorting**. At the moment, both materials share the same GPU processing priority. However, if you select **sky\_mesh** and set the **Sorting > Offset** value to -10.0 or lower, the cloud will render more consistently in terms of visual ordering.

That said, this tweak introduces an important limitation: even though the draw order improves, the GPU still cannot determine whether the cloud is inside or outside the region defined by **portal\_mesh**. In other words, you improved the visual order, but you have not fully solved the spatial relationship between the cloud, the portal, and the rest of the scene.



(4.5.i textbf{sky\_mesh} renders first, then the clouds)

To fix this behavior, you can apply the same principle you used earlier with **sky\_mesh**. In this case, you will add the stencil configuration directly to the **vx\_ray\_marching** shader, so the cloud is also restricted to the area defined by the portal.

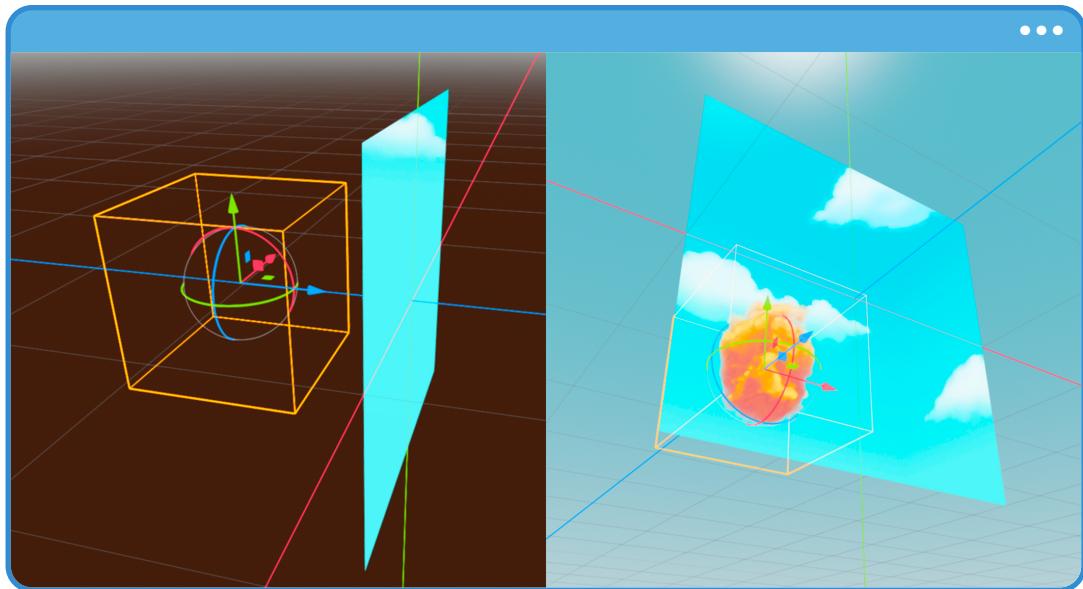
Add the following lines to the shader:

```

1 shader_type spatial;
2 // add stencil if needed
3 render_mode unshaded, depth_draw_never, depth_test_disabled;
4 render_mode stencil_mode read, compare_equal, 1;
```

As you can see, this is the same stencil comparison you previously applied to the **stencil\_read** material. This means that fragments generated by the ray marching shader are accepted only if the value stored in the stencil buffer is equal to 1.

As a result, the cloud is rendered exclusively inside the area delimited by **portal\_mesh**, resolving both the rendering-order issue and the spatial ambiguity between the inside and outside of the portal.

(4.5.j The cloud is rendered inside **portal\_mesh**)

Up to this point, you have worked with only one type of stencil comparison — specifically, `compare_equal`. However, the stencil buffer supports multiple comparison modes that allow you to control more precisely which fragments are accepted or rejected during rendering. Some of the most commonly used modes are:

- `compare_equal`
- `compare_always`
- `compare_not_equal`
- `compare_greater`
- `compare_greater_or_equal`
- `compare_less`
- `compare_less_or_equal`

When discussing stencil comparisons, you are referring to a per-fragment logical evaluation in which the value stored in the stencil buffer is compared against a reference value defined in the shader. Conceptually, this process can be represented as follows:

```

if ((StencilReference & StencilReadMask) [compare] (StencilBufferValue &
    ↵ StencilReadMask))
{
    // accept fragment
}
else
{
    // reject fragment
}

```

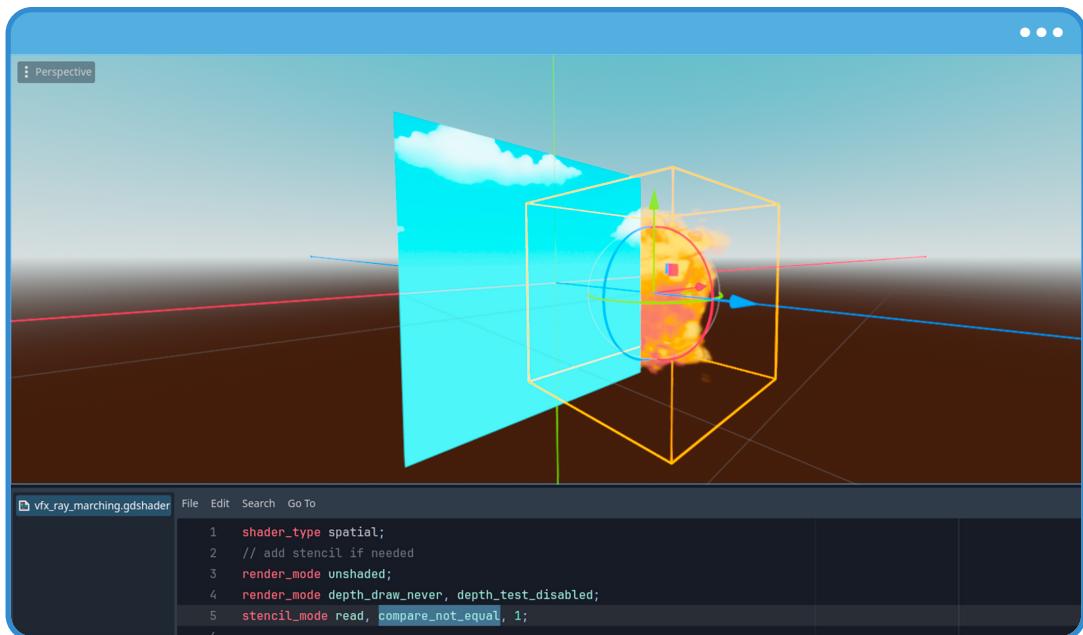
Where:

- **StencilReference** is the reference value used for the comparison.
- **StencilReadMask** limits which bits of the stencil value participate in the comparison.
- **represencodeBold[compare]**nts the selected comparison operator, such as **compare\_equal** in this example.
- **StencilBufferValue** is the value already stored in the stencil buffer, written earlier by another object — in this exercise, the value is 1.

#### Note

The stencil test in Godot follows the same conceptual model described in LearnOpenGL for modern graphics APIs. A reference value is compared against the value stored in the stencil buffer using a comparison function and a bit mask, and the fragment is either accepted or rejected based on the result of that evaluation. If you want to explore this topic in more depth, you can refer to the stencil testing section on LearnOpenGL: <https://learnopengl.com/Advanced-OpenGL/Stencil-testing>

For example, if you return to the **vx-ray\_marching** shader and change its comparison mode to **compare\_not\_equal**, the cloud will be rendered only when it lies **outside** the area defined by **portal\_mesh**.



(4.5.k The comparison type has been temporarily modified)

Another way to reject fragments is by using the `discard` instruction directly inside the fragment shader. Unlike the stencil test — which rejects fragments by comparing values against the stencil buffer — `discard` allows you to explicitly remove fragments based on any condition you define, such as a texture value.

When working with stencil buffers, you must keep in mind that materials relying on transparency (for example, those that write to **ALPHA**) may be rendered in a different render pass, altering the overall rendering order. For this reason, if your goal is to create masks with precise and predictable cutouts, it is often more stable to rely on `discard` (or a binary mask) rather than depending solely on the alpha channel in the shader that writes to the stencil buffer.

To demonstrate this approach, you will implement a texture in the **stencil\_write** shader, as shown below:

```

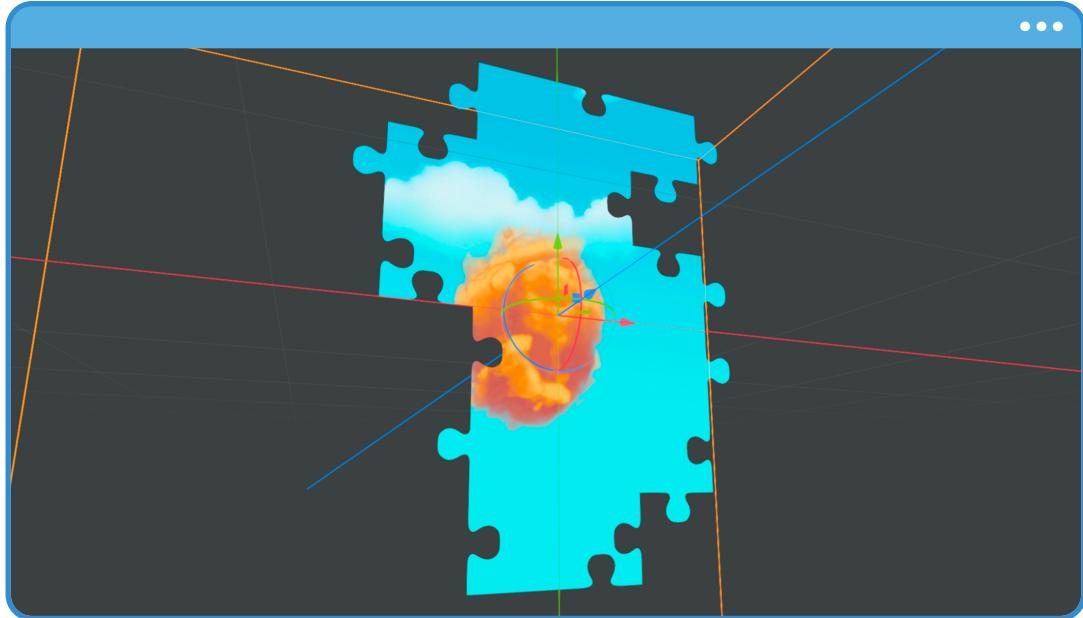
1 shader_type spatial;
2 render_mode unshaded;
3 stencil_mode write, 1;
4
5 uniform sampler2D _BaseTex : source_color;
6
7 void fragment()
8 {
9     vec4 base_tex = texture(_BaseTex, UV);
10 }
```

As shown, a texture named **\_BaseTex** has been added. If you use a grayscale texture as a mask (black and white), you can rely on the R channel of **base\_tex** to discard fragments based on the texel value. For example, if you want to keep only fully white texels, you can discard everything else as follows:

```

void fragment()
{
    vec4 base_tex = texture(_BaseTex, UV);
    if(base_tex.r < 1.0) discard;
}
```

Finally, if you return to the **stencil\_write** material and assign the **portal\_door\_tex** texture to the **Base Tex** property, you will obtain the following visual result:



(4.5.l Some fragments of **portal\_mesh** have been discarded)

# Special Thanks.

 Austin Hackett | Hang Yeh | Sergiox | Stephen Rice | Herman Garling Coll | Matthew | Ninanico | Neil Palmere | Shiena Jp | Bohdan "teo" Popov | Numa Schilling | Tm | Grzegorz Drabikowski | Sean Laplante | Max | Carlos Eduardo Boaro | Sergey Yaroshuk | Vojtěch Lacina | João Marinheiro | Jacek Adamus | Francis Jasmin | Antoine Luciani | Sander Vanhove | Tahldon | Raveen Rajadorai | Richard Cobbett | Adrian Jenkins | Vlad Scp | Hilton Perantunes | Pokemonbrycedixon | John Mandish | Alex | Daniel Stoica | Ricardo Pitta | Patientlion | Claudio Cth | Swagman | Christian Koch | Francisco Pereira Alvarado | Kyleharrison | Xxraxdex | Algio | Daniel Castellanos | Carlos Montes | Thomas Detoy | Combat Lobster | Adrian Rivero | Ewald Schulte | Punk Flag | Miri N | Gripfastgameworks | Laurenz | Gabriel Malinowski | Cameron Fernandez | Jes | Adamnakoniczny | James | Ethan Lucasawesome | Shahid Akhter | Benjamin Russell | John Pitchers | Jonne | Yun Wang | Mazhiwei | Timurariman | M Gewiss | Chunck Trafagander | Dmytro Hryppa | Patrick Joos | Joseph Torbett | Jason Peterson | Mako | Norbert Haacks | Mika | Sean McIntyre | Collin Myhre | Dottorreid | Morpheus Tsai | Aiastesla | Fabian Ruch | Jan Nothacker | Rémy Tauziac | Martijn Van Halderen | Yon Montoto | Walpurgis Time | Keating | Andrey Valkov | Thsage | Eric Wong | Nils Nordmark | Nikita Blizniuk | Joseph Deluca | Francesco Michelini | Christopher Hall | Jaycoleman | Madlion | Graham Reid | D-hill | Marcanthonycacho | Thomas

 Jahtani | Tskarthik | Mutrz | Vasil Genov | Manraaj Nijjar | Kovidomi | Robles | Jesse Virtanen | Domingo Mesa-maliniak | Frederic Batardy | Matoo | Jiří "venty" Michel | Goldennick | Ginton | 박용민 | Hi | María Luisa Carrión | Andrew Meyer | M | Trent L | Mark H | Crapule | Dorian Nicou | Sam | Suchaaver Chahal | Victor | Andrew Herch | Boti Beto | Raymond Gibbons | Phil | Thomas Hunt | Constantin Balan | Brandon Grenier | Abyssallure | Katia McCarthy | Joakim Karlsson | Dmitrii | Markus Michel | Robert Allen | Ross | Mr Thomas Graveline | Stephen James | Tiago Ferreira | Anderson Borba | Andreas Gillberg | Morgan Wesemann | Janusz Tarasewicz | Ben Thedragon | ガンズターン公式 | Saul | Flyn | Hellno | Karthik Nallan | Rosciuc Bogdan | Big Quail | Matt | Justo Delgado Baudí | Roose Alexandre | Mangirdas Balciunas | Jettelly | Asylbekz | Roman Kozolek | Alvaro Larre Borges | Creta Park | V | Nikola Stanojević | This Is Bennyk | Santiago | Alvaro G Lorenzo | Zachnorthcott | Orion Ironheart | Brandon Farrell | Nicolas | Douglas Ávila | Mª Dolores Lebrecht | Bernard Cloutier | Isaac Iverson | Matheus V | Christophe Brenon | Dave | Vtalyh | Jacopo Galati | K G H | Sam Van Berlo | Joshua | Šarūnas Ramonas | Tamego | Emerson Santos | Ashely | Zoe Firi | Liyizhanguk | Aleksander G Solheim | Ian | Michael Belknap | Mattias | Fu Ruru | Karim Matrah | Zhou Kaiwen | Quentin Delvallet | Mmdrewind



Emmanuel | Someone | Maxie | Patrick Exner | Ivan Ho | Jesse Fletcher | Chris Morgan | Carlos Eduardo Pérez Villanueva | William Brendaw | Swampnrd | Shader Snake | Sân Nguyễn Minh | Gerwyn Jones | Seth Foznot | Harry Cassell | Deear Art | Cillian Clifford | Sabre | M Fatati | Ardscoff | Guilherme Cattani | Albert Miro Montalban | Georges | Kev Zettler | Carsten Bloecker | Clément Vayer | Jose Ignacio Palacios Ortega | Mcspidey | François De La Taste | Paul | Ross Solomon | Gabo Salcedo | Caigan | Brad Svercl | Jing Yi Chong | Mykola Morozov | Kalsarin | Zewy | Phoen Leo | Sebastiankmilo | Hipton | Bawlshy | Tim Arlow | Ffdd | Maksim Loboda | Colin | Lisandro Lorea | Björn Wilke | Robert Mahon | Dave Lecompte | Pawel Antoniuk | Carlos Aguirre Vozmediano | Elliott Chillag | Kyle Hessel | Jose Angel Canabal Delgado | Benjamin Oesterle | Ngo Viet Luan | Meleg Zoltán | Michael Jones | Kevin Krause | Daniel Cavazos | Travis Womack | Ryan Hinds | Dewey Mowris | No Yes | Nigel Che | Bass | Juanye | Britt Selvitelle | Tor | Noah | Stuart Carnie | Kamwade | Smedstad | Anton | Js | Mads Sønderstrup | Steffenschmidt | Tor-arne Nordmo | Алексей Акулов | Bioplant | Lucas Felix | Natevc | Nathan | Kangweon Jeong | Mei Yamasaki | Meownoi A | Nathanraughn | Monica | Lighthalzenxii | Nike | Mapinher | Xavier Shaver | Jack Lueyo | James Clancey | Thomas Guyamier | Alfred Reinold Baudisch | Max | Default | Ben Mckenna



Ondřej Kadlec | Tate Sliwa | Dt | Marino | Calvin M Evans | Alen Lapidis | Thoughton | Mariano | Daquan Johnson | Eli Greenwald | Adam Gulacsi | Nea Pocem | Liming Liu | Michael Birtwistle | Tobias Brandner | Richard Pearson | Dimitri Bauer | Paulo Poiati | Edwin | Chauncey Hoover | João Luís Reis | Amy Haber | Corentin Lallier | Alvinanda Purnomo | Kerry K | Ana Greengrass | Juno Nupta | Israel | Aaron B | Kris | Gabe Volpe | Keita Fukuchi | Clémentine Ducournau | Gustavo León | Haydenvanearden | Jdon Leininger | Bohdan | Develop | Caitlin Cooke | Alexandre Schenk-lessard | Grāodopāo | Terencep Dixon | Anatol Bogun | Zammecas | Benji Lising | Kevin Abellan | Ryan T | William | John Daffue | Simone Sorrentino | Craig Spivack | Pavol Gocik | Clpereira | Alejandro Vera | Yb Park | Pyrat | Acorzomate | Oniric View SI | Tulasi Rao | Patricoland | Xanroticon | Nicolas Delbaer | Jordan | Thom | Findemor | Amasten Ameziani | Patrick | Kacey Walsh | Nathakorn Throngtrairat | Christian Snodgrass | Emmanuel Bouis | Pokkefe | Mike Klingbiel | Drew Herbert | Ross Rothenstine | Rhenn | Talal Almardoud | Vinicius Hoyer | Matti Pohjanvirta | Tuomas Ilomäki | Elias Manzaneda Santamaria | Daniel Canham | Liam Cornbill | Daniël Michael Kracht | Eliud De Leon | Sean Welgemoed | Pangake | Jules Fijolek | Krystof Klestil | Luc Magitem | Thijs Tak | Adam Spivey | David Kol | Glenn Barker | Adrián | Gianluca Pumoni | Mal | João Récio | Jacob Lerche | Daniel R Martinez



John Pimentel | Kyle Young | Lysandre Marti | Matthew Makary | Craig Kerwin | Sam Winfield | Alec White | Corentin Lecroq | Richard | Alex | Taylor Segura Vindas | Erik | Ashley Moon | Simke Nys | Spencer Hoban | Nicholas Brennan | Efrian Vega Jr | Sam | Gato | Thomas D | April Clark | Jakub | Colin | Lockyer Road | William Hoobler | Kenneth Holston | Brian | Carter Layton | Martin Voth | Brett Buckner | Joba López | Lynx | Panupat Chongstitwattana | Bryan English | William N Kentris | Penny!! | Minoqi | Anton Petersson | Aaron J Schneider | Nate Moore | Colin Miller | Edgar | Sam Blackman | Frank Gusche | Gillian Monte | Eric Alvarez | Viktor Sladek | Ian | Paul Cloete | Ben | Ppsi | William Beatty | Photophighter | Arman Frasier | Farhad Yusufi | Lee | Andrew Lee | Codebycandle | Łukasz K | Alfie | Jeff Zimmer | Game Log | Aaron Imrie | James Karavakis | Dylan Glow | Miles Harris | Kris | Reavenheart | Benjamin Schmitt | Frodo | Tom Kertels | Connor R | Nathan | Gary Johnson | Kein Hilario | Blake Rain | Bryant Flores | Joseph Sugabo | Eyal Assaf | Lewis Simpson | Christopher Holt | Bluen | Jayden Jackstet | Alan Jimenez | Noah Blake | Icoza | Elizabeth | Alexandra Bradford | Agustin Adducci | Tim | Xuxing | Marcial Santallory | Austin Grimm | Rob Brown | Odin Ugelstad | Henry Schwenk | James G Pearson | Marco | Danthecardboardman | Aaron Wu



Gerben Van Der Heijden | Nil Vilas | King Artorias Arthur | Marcos | Andrew Moffat | Jun Yeo | Matthew Swee Chuan Tan | Bruce Leidl | Mailjib Jb | Ryan Shetley | J Chris Druse | Mārtiņš Prelgauskis | Umut Ulutas | Niki | Hyou Sasaki | William Sattanuparp | Darrin Henein | Sean Pace | David Betancourt | Gabriel Martinez | Miroslav Edler | Angelina | Zane | James McVitty | Eric Roberts | James Jones | Robert Georges | Terry Wu | Eldaville | Coryo Moore | Joshua Tucker | William Kruspe | Dizzy Moth | Peter Chantler | Nural | Gustavo Lopes Domaradzki | Quock | Alec Gall | Steven Lenaerts | Mateus S Pereira | Alex | Andrew Brizendine | Daniel Holmen | Garrett Bare | Kevin Crowell | Spaghetmenot | Michael Simon | Yuval Dekel | Alan K | Aidan Davey | King Artorias Arthur | Dominic Perno | Anthony Beninati | Sammi Tu | James Steele | Pepijn Stuurwold Van Walsum | Holger Schmidt | Oliver Gutiérrez | Nova Obrien | Fredrik Hamstad | Jonathan | Thomas M Kidder | Kai | Gk | Ben Revai | Daniel Leiva | Miles Welsh | Cannon Pharaoh McClelland | Emanuel Lamela | Jānis Alunāns | Michael Radke | Ryan Woods | Torgabor | Reid Hannaford | Mike Barbee | Aria Ramshaw | Benjamin Paschke | Mathias | Charles Engel | Benjamin Flanagan | Jettelly Publishing | Antonio Colon | Sac Alanpereira | Patrick Grant | Marquis Jackson | Miguel Angel Mendez Cruz | Scottie Doria | Ordinary Fox | Daniel Ridao | Jorn Van Denbussche | Shelly Barnes | Celia Tran | Rem | George | Aleksander | Dmitry Maizik | Dave | Ryanimates | Annabella Rosen | Colton Wolkins



Daniel Rodriguez Aires | Matteo Kramer-tamburrino | Albin Lundahl | Luis Puentes | Andrew Cotter | Dylan Goncalves Martins | Arroyojavier Sanandres | Xury Greer | Jjolton | Adam | Daniel Carswell | Deear Art | Shawn Dhawan | Daniel Gerep | Steve Versace | David Dam | Bdilmore | Dima | Josuprasi | Yaramirez | Michael Frederic | Ruslan Horiuchkin | Nicholas Vosburgh | Chen | Mateusz | Flapp | Komatsuki | Auro | The Zefan | Michail Maridakis | Sebastian Rangger | Henry Liu | Brandonrconyers | Andreas Becker | André B Luz | Logan Croley | Daniel Mutis | Ulric Boel | Vitalik | Tom Brewer | Darmaz | Keagen Bouska | Chance Mcdonald | Bidda | Pedro Perez | Antonio Contreras Arzate | Dan Crowe | Jeong Hoon Shin | Benjamin F Hajas | Javis Jones | Kateryna | Seongho-son | Alessandro Pilati | Brian Huqueriza | Hugo Toti | Ronald Casili | Sdc | Joe | Jordan Faas-bush | Eli Makaiwi | Henry Audubon | Zoey Lome | Jaakko Ojalehto | Matt Odette | James Asumendi | Trevor Harron | Alexander Abshagen | Maxim Oblogin | Tresceneti | Asher | Aaron | David Tag | Jake Elliott | Tim | Tobias Cunningham | Ryan | Robert Fraser | Kristian Gibson | Konstantin Kudinov | Lizzy Moyes | Charlotte Jones | Sethcrawford | Carlosgolfer | Ryan Renna | Roman | Jean-francois Segretain | Joao Filho | Flowulf | Ben Boyter | Pearshaped | Jacob Yero | Felipe Fausto | Joaquin Muñiz | Mel Irizarry | George Thompson | Hakan | Gerard Parareda | Roger Bujan | Eric | Jully



Jubimage | Luis Alfredo Figueroa Bracamontes | Chris | Alessandro Talloru | Adam Gibson | Revo Wu | Aziz Abidi | Lennysioll | Maxim Jourenko | Tobias Holewa | Rxwp | Darren Larose | Sebastian Karlsen | Pendant Fills | Damian Stähr | Niklas | Xuwaters | Christian Ahlers | James Coleman | Ryan Moos | Tomáš Adámek | Adriangtuazon | Cagri Benli | Darren | Guy | Mateusz Wolos | Kristofer | Jonathanrivasmencia | Joelfreemand | Eric Persson | Saul | Lxyhaqs | Ericpdss | Tấn Phát Huỳnh | Guma | János Harsányi | Mr Connor S Adams | Florian Stumpf | Wyatt | Adam Helešic | Jakob Sorenson | Nick | Hannah Petherick | Erik Loide | Bradley | Artbarте | Christophe Delahaye | Cheng Bowen | Jeremy Kamrath | Amanda Koh | Philip Ewert | Michael Heywood | Tralexium | Maxfield Hewitt | J G | Chong Ren Li | Max Gratzl | Aaron | Ronald Gibson | Niall Brady | Sam Morgan | David Stewart | Marcel Gentner | Martin Prazak | Josh Bousfield | Pietro Maggi | Adrian Hernik | Sebastian | Jonathan Scheer | Alexander Velchev | Kai-fabian Schönauer | Mattia Belletti | Izaac Jordan | Kyle Donaldson | Daniel Maiorano | Jesse Schramm | Jonas Lund | Peter Massarello | David Carr | David Dunham | Mario Schilling | Jw | Danni | Antonio Sobrinho | Eric | Kaptain Radish | Z | Andrew Robyn | Andicus | Jacob | Alek D Flener-satre | Joseph Turnquist | Patrick D Rupp | Davide | Bandit | Matt | Wataru Ikeda | Andi Doty | Ryan | Joshua Rehling



Nathan Van Fleet | James Walter | Chris Mcpherson | Pixel Pilgrim Studios | Marcell Kovacs | Roro | Dominykas Djačenko | Alex Robbins | Ng Game Junk | Arturs Cernuha | Iramis Valentin | Niko Van Wonterghem | Michael Andrew Revit | Garrett Drinkard | Andrew Pachuilo | Drak | Kerry Shawcross | Vincent Michael Mooney | Brandon Wolff | John Laschober | Trevan Haskell | Andrew Greenwood | Dani Fey | Alec Burmeister | Kieran Lane | Seonhee Lim | Javier Teodoro De Molina Mejías | Keith Crabtree | Damu | Raffaele Picca | Anthony | Luis Ernesto Torres Santibañez | Adam Webber | Daniel | Mfernandez | Ryan | Miranda Schweitzer | Taylor Bray | Mark Hubczenko | Fathony Teguh Irawan | Renju Mathew | Alexander Cruz | Tyler Smith | Brad | Austin Baldwin | Jacobarkadie | Kevin Bergstrom | Alex | Daniel | Matthew Xiong | Quinn Collins | Abbey Howard | Michael Hrabánek | Vegard | Harrison Grant | Adrien Baud | Sa Sa | Raf Van Baelen | Santitham | Marianne Stalter | Terrence Coy | Jodie | Nicholas Jainschigg | Aerton | Bee | Trevor Starick | Michael | Matej Šulek | River Vondi | Jacob | Sam Shockey | Florencia Sanchez | Ezequiel Selva | Cody Keller | David Yaeger | Pewbop | Gabriel Benjamin Valdez De León | Ask Skivdal | Gautam Dey | Vinicius Nakamura | Semyon | Bas | Felix Bytow | Adam Reid | Clement | Kirill | Bawm | Hai Bo Wang | Anthony Leblanc | Duncan | Étienne Guerette | Wesley "Sky" Nworisa | Luca Vazzano | R Wijdenes | Tyler Hamilton | Ben Witte | Morchaint | Elliott | Alex Wang | Ethan



Jochen Weidner | George Dimopoulos | Christopher Margroff | Milan Koyš | Alexandra Lee | Sterling | Samir Alajmovic | David Devcic | Shane Gadsby | Murilo Gama | Max Daniels | Dylimm | James Minshull | Cory Kennedy-darby | Jeff T Klouda | Amit Feldman | Florian Frankenberger | Stefan Maier | Barbara Zacharewicz | Pablo Mansanet | Francis D Cleary | Joseph Krueger | Morgan Hezon | Benjamin Reber | Jonathan Tippett | Danny Eisenga | Michael Laplante | Matthew Arabo | Ryan Reed | Ludger Meyer-wuelfing | Raji | Thefre | Amani A | Iván D | Adrian | Max Olin | Ruben Rodriguez Torres | Eidel Odiseo Giménez | Kay | Daniel Steven Sarmiento Santos | Kevin Do | Anthony Tarr | Adam Slavik | Alberto Flor | Jared Moore | Hendrik Poettker | Eric Lugh | Zack | Warp Vessel | Tommi Shelton | Earl Braxton Mckenna | Adrien Farfals | Jonathan Cater | Luke | Matteo Stefanini | Oschijns | Keo Daniel Bun | Kevin Young | Hardtrip | Luc-frederic Langis | Pj Palomaki | Laurent Tourte | Geowarin | Barmy | Khairi Harris | Nick Strasky | Brent Lovatt | Alexander Graham | M Koray Duman | Houman Jafarnia | Cameron Oehler | Cesar | Justin Doornbos | Tj Morse | Jeavh | Street Nw | Pete | Ross | Sophie Freeman | Gerrit Schimpf | Brody Wilton | Deniz | Matthew Laurenson | Nicholas Venditti | Lucas Long | Jardian Halliday | Ahmad Takhimi Bin Ahmad Mahayuddin | Joe Bechtel | Delbert Vick | Stefan Larsen | Jollymeatball | Kelita Nolan | Lily Jin | Terence Steabner | Daniel Hall | Elia Théo | Matthew Brennan | Razvan | Ditherdream | Sjoerd Van Kampen



**Jettelly wishes you success  
in your professional career.**