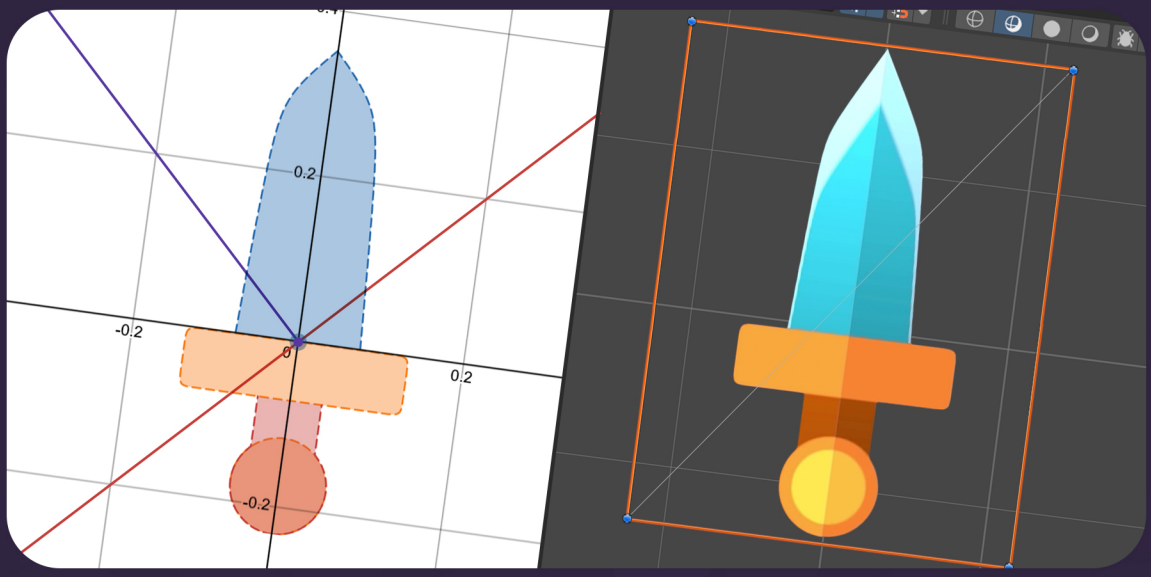
Learn. Create. Master.

Visualizing Equations – Vol. 2

Shaders & Procedural Shapes in Unity 6.

A visual guide to creating stunning procedural shapes from math equations with Shader Graph.

#Unity #GameDev #ProceduralArt



Fabrizio Espíndola.


Jettelly

Visualizing Equations Vol. 2

Shaders & Procedural Shapes in Unity 6.

A visual guide to creating stunning procedural shapes
from math equations with Shader Graph.

Fabrizio Espíndola

#Unity **#GameDev** **#ProceduralArt**

Visualizing Equations – Vol. 2, version 0.0.6d.
 jettelly.com
77 Bloor St. West, Suite #663, Toronto ON M5S 1M2, Canada.

Credits.

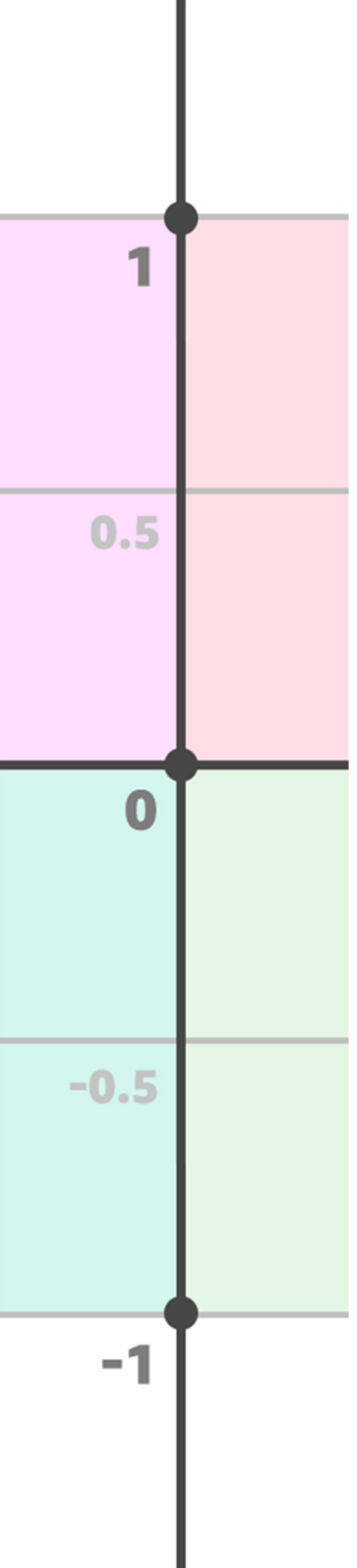
Author.
Fabrizio Espíndola

Design.
Pablo Yeber

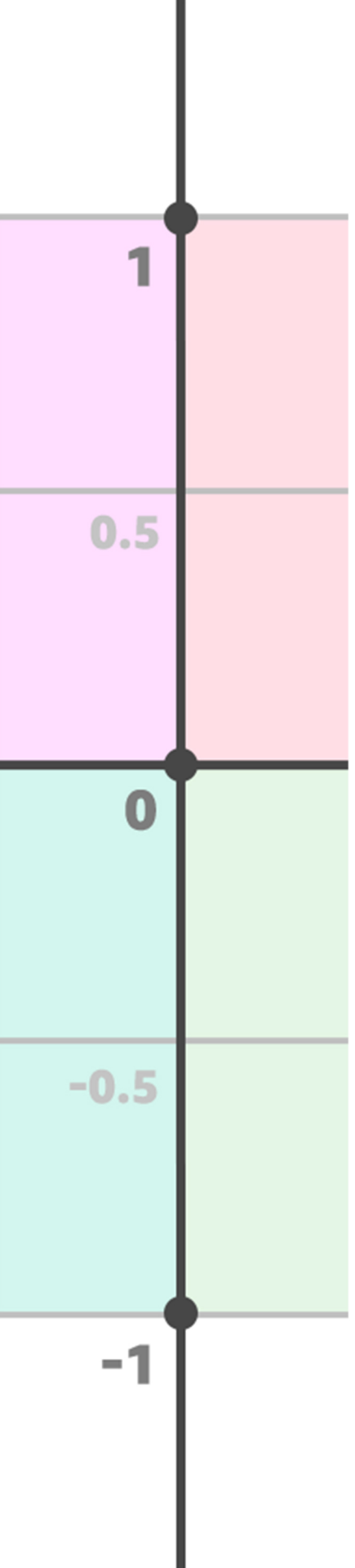
Technical Review.
Martin Molina
Roberto Ortiz

Translation, Grammar, and Spelling.
Martin Clarke

Head Editor.
Ewan Lee



Content.



Throughout my career as a Technical Artist and programmer, I have observed that the technical aspects of videogame development are fundamental to its success. This is because optimization (or part of it), structures, and organization are directly linked to this subject. For example, if you want to become a great UI artist, you must understand what screen-space is, how UV coordinates work, how anchors function, and other relevant concepts. However, in most cases, artists and designers do not fully delve into these topics, either due to lack of mathematical knowledge or unfamiliarity with computer graphics.

This book was born from the need to understand precisely these essential aspects that make it possible to transform mathematical formulas into visual and functional experiences in game development, as well as in research and development processes.

In **Shader and Procedural Shapes in Unity 6**, I have compiled over eleven years of industry experience to offer you a visual and practical guide that will help you master the art of converting equations into interesting procedural shapes. This book is designed to take you step-by-step and in a structured manner, from basic mathematical concepts to their direct application in Unity, one of the most widely used engines in game development.

Each chapter has been structured to allow you to build knowledge progressively. We begin with **Chapter 1**, where you will discover how polynomial functions can serve as the foundation for generating curves and surfaces, even applicable to screen transitions. We continue with **Chapter 2**, where we delve into the importance of trigonometric functions, essential for modeling cyclical movements and natural patterns. In **Chapter 3**, we put what we have learned into practice and develop a procedural shape—a Shuriken—using only linear functions and geometric areas. In **Chapter 4**, we transform linear functions into signed distance functions (SDF) in two dimensions, opening the door to more complex and efficient rendering effects. Finally, in **Chapter 5**, we extend these concepts to three-dimensional space, where you will learn to create objects and understand the mathematics behind each one.

My goal in sharing this knowledge is not only for you to acquire a theoretical understanding but also to develop practical skills that allow you to experiment, create, and enhance

the visual quality of your projects. I am convinced that the combination of mathematical fundamentals and shaders is the key to innovation and a significant step forward in your professional career.

Who this book is for.

This book has been designed for artists, designers, and programmers who want to deepen their understanding of mathematical functions applied to HLSL (High-Level Shader Language), using Unity 6000.0.29f1 LTS as the graphics engine. We will focus on version 17.0.3 of the Universal Render Pipeline (URP) within a 3D environment, although the concepts and techniques introduced can be applied to any platform that employs a shader language, including CG or GLSL.

Regardless of your prior experience with polynomial, exponential, or trigonometric functions, this book will guide you through HLSL step by step. For those approaching these concepts for the first time, detailed visual explanations that are clear and accessible will be provided. Our goal is for every reader, regardless of experience, to dive into the exciting universe of procedural shape creation, discovering the power of mathematics as an essential tool in generating visual effects.

Conventions.

In this book, we have defined a set of conventions to highlight specific elements and make the information more accessible. These conventions include the use of **bold** text to emphasize technical terms, *italic*, brackets, and other styles. We also use uppercase letters to represent the components of a vector.

- **Code Highlighting:** Functions, methods, and variables will be presented in a specific font to emphasize their importance and technical nature.
- **Mathematical Functions and Equations:** These will primarily be displayed in *mathematical italics*, allowing us to easily distinguish between a generic number and those directly related to code or a specific operation.

- **Names and Objects:** These will be presented in bold throughout the text.
- **Ranges:** Ranges will be shown in brackets (e.g., [0 : 1]) in some cases to make them easier to read and understand.

The code blocks will be **presented in a standardized format** to **maintain consistency and clarity**, as illustrated below:

```
#ifndef CUSTOM_FUNCTION
#define CUSTOM_FUNCTION

void method_half(in float input, out float Out)
{
    // Code here …
}

#endif
```

These conventions have been established to improve the clarity and understanding of the information presented and are maintained consistently throughout the book.

Downloadable Resources.

To help you follow along with the examples in this book, we have prepared a set of downloadable resources. These include project files, Shader Graphs, textures, HLSL scripts, and other assets used throughout the chapters.

You can access these resources by visiting our website:

- https://jettelly.com
- Or by scanning the QR code below.

If you encounter any issues or have questions about the resources, feel free to reach out to our support team at contact@jettelly.com.

Errata.

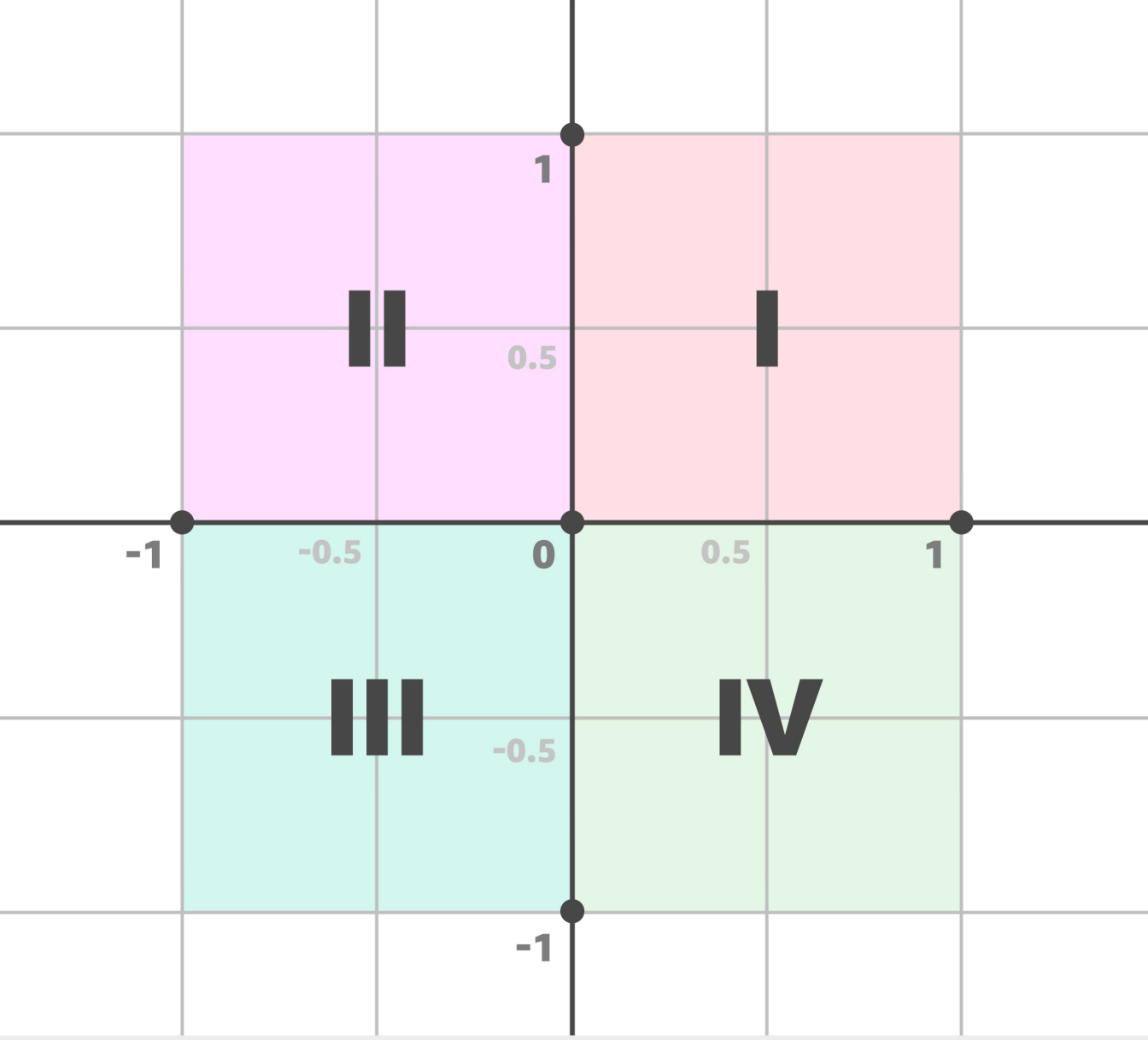
While writing this book, we have taken precautions to ensure the accuracy of its content. Nevertheless, you must remember that we are human beings, and it is highly possible that some points may not be well-explained or there may be errors in spelling or grammar.

If you come across a conceptual error, a code mistake, or any other issue, we appreciate you sending a message to contact@jettelly.com with the subject line "VE2 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.



Chapter 1

Polynomial Functions.

In this chapter, we'll dive into the fascinating world of polynomial functions, which serve as the foundation for many techniques in graphics generation and procedural shape creation. We'll start by analyzing the linear function, the most fundamental form of a polynomial, to understand its core properties and apply them in visualization using Shader Graph. This includes not only drawing the resulting line but also highlighting its origin point—an essential aspect for understanding its behavior and coordinate transformations in graphical space.

As we progress, we'll explore the quadratic function, examining how its curvature and geometric properties allow for more complex and natural shape modeling. Through hands-on examples, we'll compare the simplicity of the linear function with the versatility of the quadratic function, helping us to decide which one to use to achieve specific forms.

With this theoretical and practical approach, the goal is to build a strong, applicable understanding that empowers you to experiment and create dynamic procedural shapes.

1.1 Linear Function.

While it's true that linear functions are widely used in fields such as statistics, economics, and social sciences, their applications extend even further, reaching into the fascinating world of video games. In the field of computer graphics, they play a crucial role in drawing straight lines, making them an essential tool for creating various visual effects, including scene transitions. But how do they do this? To understand, we'll start by briefly examining the structure of a linear function.

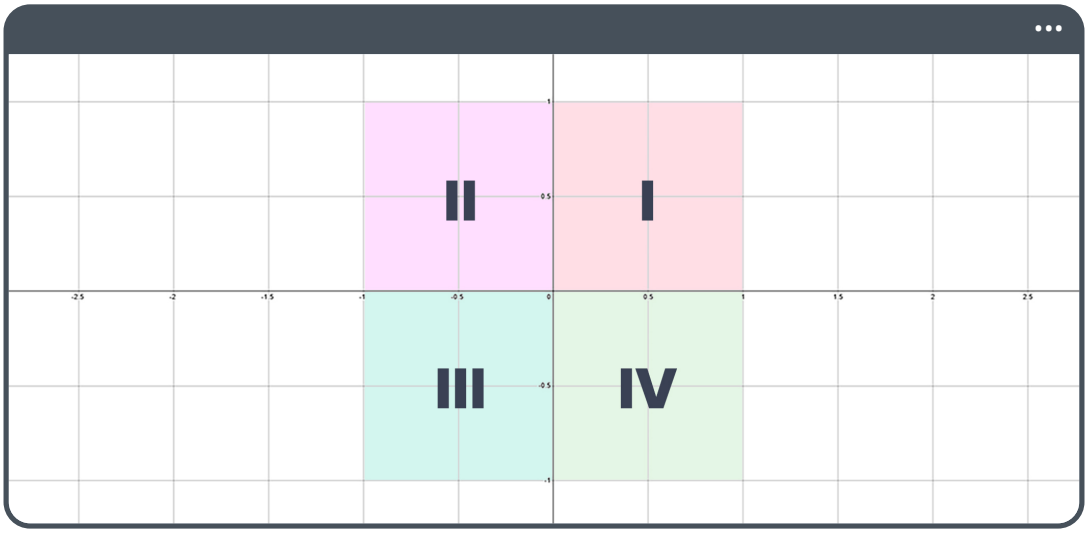
$$f(x) = mx + b$$

(1.1.a)

If you take a close look at the function above, you'll notice that the variable m represents the slope, which indicates the inclination of a line, while b marks the intersection point on

the y-axis—that is, the value of $f(x)$ when $x = 0$. However, to fully understand how these parameters affect the graph, it's essential to grasp the concept of Cartesian coordinates.

Cartesian coordinates form an orthogonal coordinate system in a rectangular grid that spans the Euclidean space in two dimensions, x and y. This definition can be extended to a third dimension, z, making it an essential tool for constructing three-dimensional models. By using the axes of the Cartesian coordinate system, we can subdivide two-dimensional space into a set of subregions known as quadrants, which are useful for precisely defining functions when working with other coordinate systems.



(1.1.b https://www.desmos.com/calculator)

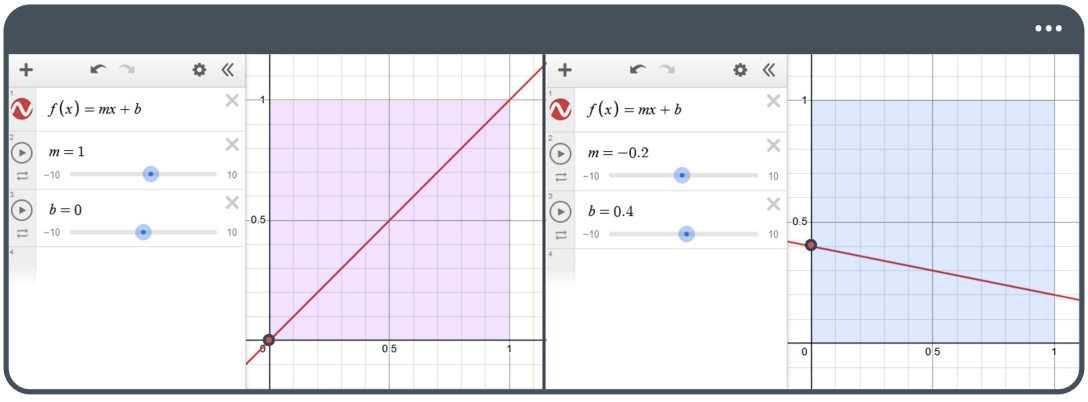
In this initial stage, we'll focus on two-dimensional visualizations using the Desmos graphing calculator, which you can find at the link in Reference 1.1.b. As seen in the tool, colors have been used to clearly distinguish each quadrant, highlighting the fact that, depending on their location, the x and y coordinates can have positive or negative values.

Quadrant 1	Quadrant 2	Quadrant 3	Quadrant 4
$x+$	$x-$	$x-$	$x+$
$y+$	$y+$	$y-$	$y-$

An interesting aspect to consider when working with computer graphics is that, by default, UV coordinates only cover the first quadrant, starting at [0.0, 0.0] and ending at [1.0, 1.0]. As a result, both coordinates are always positive.

Since we'll later apply this knowledge in HLSL, having a solid understanding of these concepts is essential. Otherwise, it will be challenging to grasp how to draw a line at a specific position in your shader.

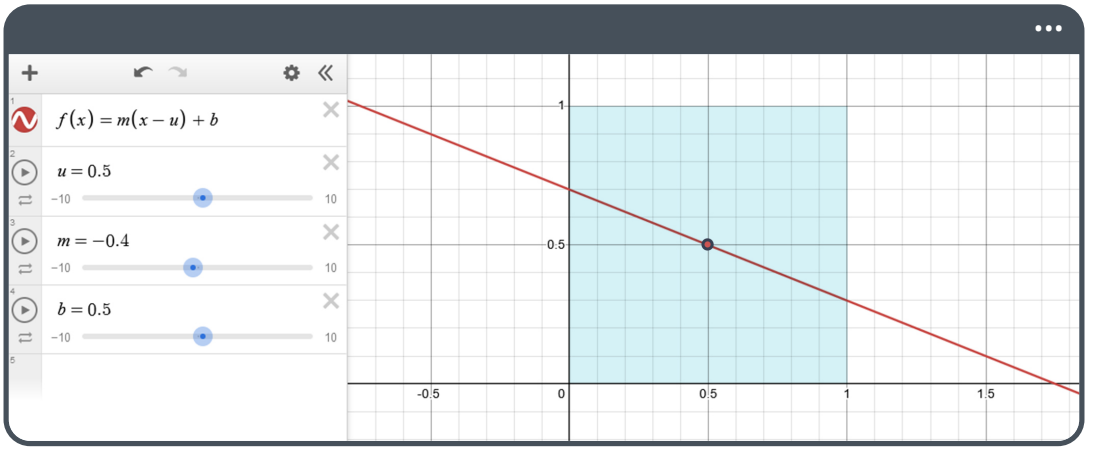
The function we saw earlier in Reference 1.1.a is visualized in the Cartesian plane as follows.



(1.1.c https://www.desmos.com/calculator/3ajykatgok)

As shown in previous reference, the slope m represents the number of units by which y increases or decreases when x changes, while variable b indicates the distance from the origin to the point where the line intersects the y-axis. Recall that the "origin" refers to the starting point on the Cartesian plane, which is located at [0.0, 0.0].

Now, how can we center the function around the position [0.5, 0.5] within the plane? To achieve this, we need to extend the linear function and introduce a new variable into the equation, which will be subtracted from x. Let's examine this further:

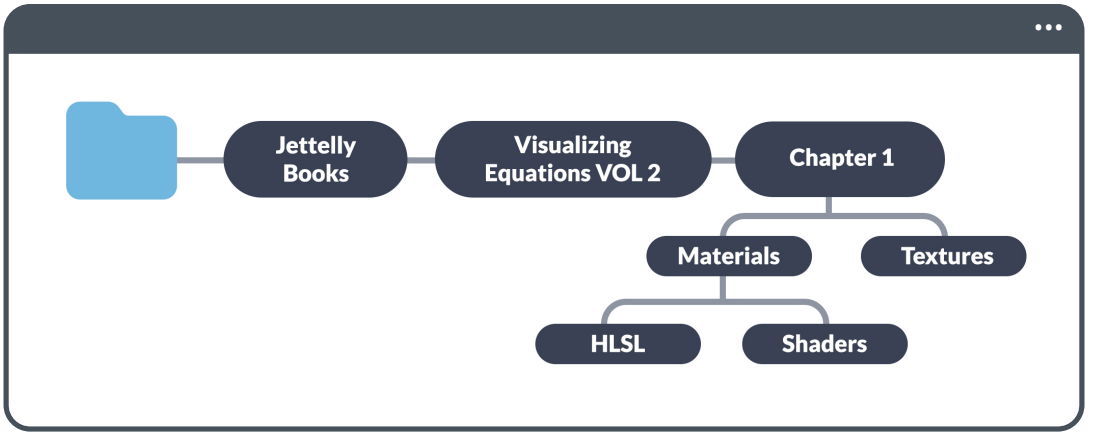


(1.1.d https://www.desmos.com/calculator/amp8nacnqb)

As shown in Reference Link 1.1.d, a new variable denoted as u has been introduced into the function. This variable is associated with the U coordinate of the UVs and has a value of 0.5. By subtracting it from x, u causes a horizontal shift that centers the reference point on the Cartesian plane. This adjustment is essential because, when working with UV coordinates, it's often necessary to adjust the origin point to draw lines from different locations.

1.2 Visualizing the Linear Function in HLSL.

Before exploring the graphical representation of the linear function, it's essential to establish an organizational structure for the project. You'll start with the following framework, which you'll expand as you progress through this book:



(1.2.a)

Given the nature of Shader Graph, to visualize a custom function in the editor, you'll need to create at least three objects in your project, which correspond to:

- Shader.
- Material.
- Script with **.hlsl** extension.

Each of these elements plays a fundamental role in the visualization process. For example, the shader is responsible for executing the mathematical operations defined in the HLSL script, performing the necessary calculations to determine how the images will be graphically represented. Meanwhile, the material serves as a medium for visualizing these calculations, applying the effects defined by the shader to objects within the scene, resulting in the desired visual representation.

It's important to highlight that shaders in Unity are structured using two main programming languages: ShaderLab and HLSL. The first serves as a declarative language used to define properties, configurations, and structures, establishing the framework within which shaders operate, organizing how and when different steps of the graphics processing pipeline are applied. In contrast, HLSL focuses on the computational aspect, allowing for the precise definition of mathematical operations and graphic algorithms that determine color behavior.

Let's proceed by creating the previously mentioned objects, starting with an **Unlit Shader Graph** shader, and name it **Linear Function**. You'll find this at the following path:

- Assets > Create > Shader Graph > URP.

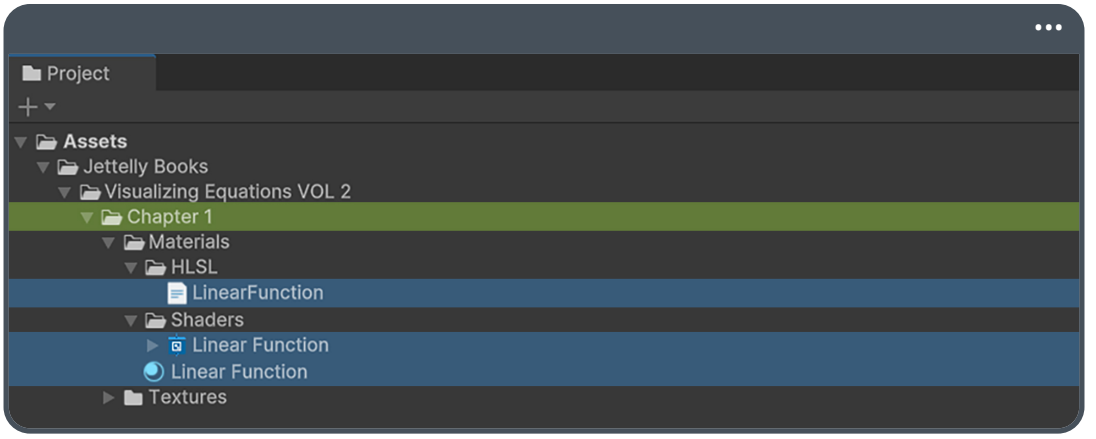
Generally, we opt for an **Unlit Shader Graph** type shader throughout this book. The choice **Unlit** refers to an element that isn't affected by environmental lighting, which is ideal for exemplifying the function, thus avoiding additional calculations. However, in later sections, a different approach (UI Default) will be explored that allows us to apply calculations to user interface images, thereby expanding the scope of our visualization techniques.

You'll continue by creating a material, which, for practical reasons, will also be named **Linear Function**. You'll find this at:

- Assets > Create > Material.

Finally, you'll generate an HLSL script. However, since Unity doesn't include this type of script by default, you'll need to create it either directly from the code editor you're using or manually from a folder in your operating system. This involves navigating to the HLSL folder in your project (whether on Windows, Mac, or Linux), creating a new text document, and changing its extension from **.txt** to **.hlsl**. Once created, the script should follow the same naming convention as the two previously generated objects, omitting spaces in its definition, resulting in **LinearFunction**. We'll apply this naming scheme to all similar files throughout the book.

If all settings have been correctly implemented, your project should be displayed as shown in the figure below:



(1.2.b)

It's important to note that some additional steps will be necessary to ensure the correct visualization of the function in the Scene view. To do this:

- You must assign the shader to the material.
- Then, include a Quad type 3D object in the Hierarchy window.
- Finally, assign the material to the Quad in the Scene window.

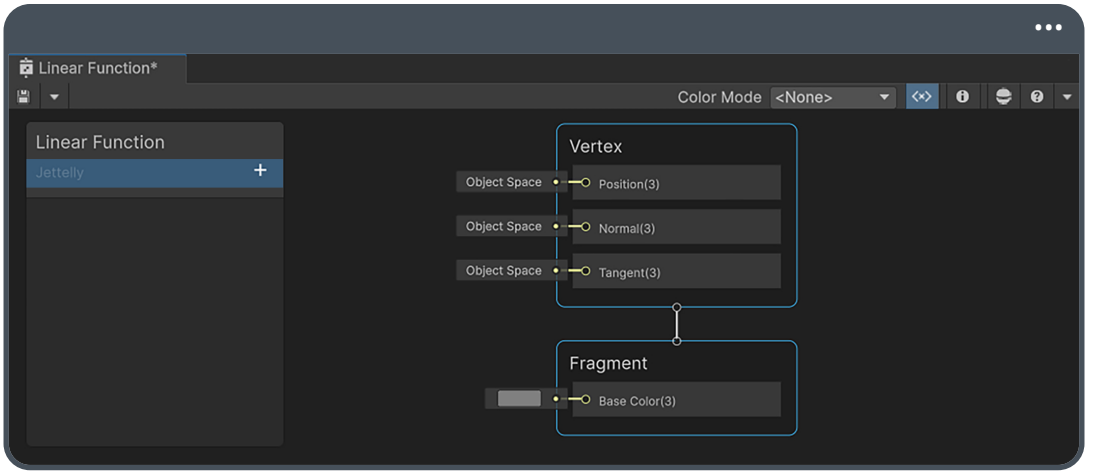
However, how can you find the previously generated shader? To do this, it's necessary to know its default location. Each time we create a **.shadergraph** shade, it's saved in the following path:

- Shader > Shader Graph.

This can be verified by opening the shader **Linear Function** and selecting the **Blackboard**. Its path should appear below the file name. However, for this exercise, you'll configure the path with the name **Jettelly**, as follows:

- Shader > Jettelly > Linear Function.

It should be noted that, for consistency, we'll change the path on all the shaders that we develop throughout the book to the one used above.



(1.2.c)

Since the linear Function lacks an independent node in Shader Graph, you'll use the **Custom Function** node to define it. Inside the **Master Stack** (the node connection area), press the SPACEBAR and search for the node by its name. To fully understand how this node works, activate the **Graph Inspector**; this allows you to define both its inputs and outputs.

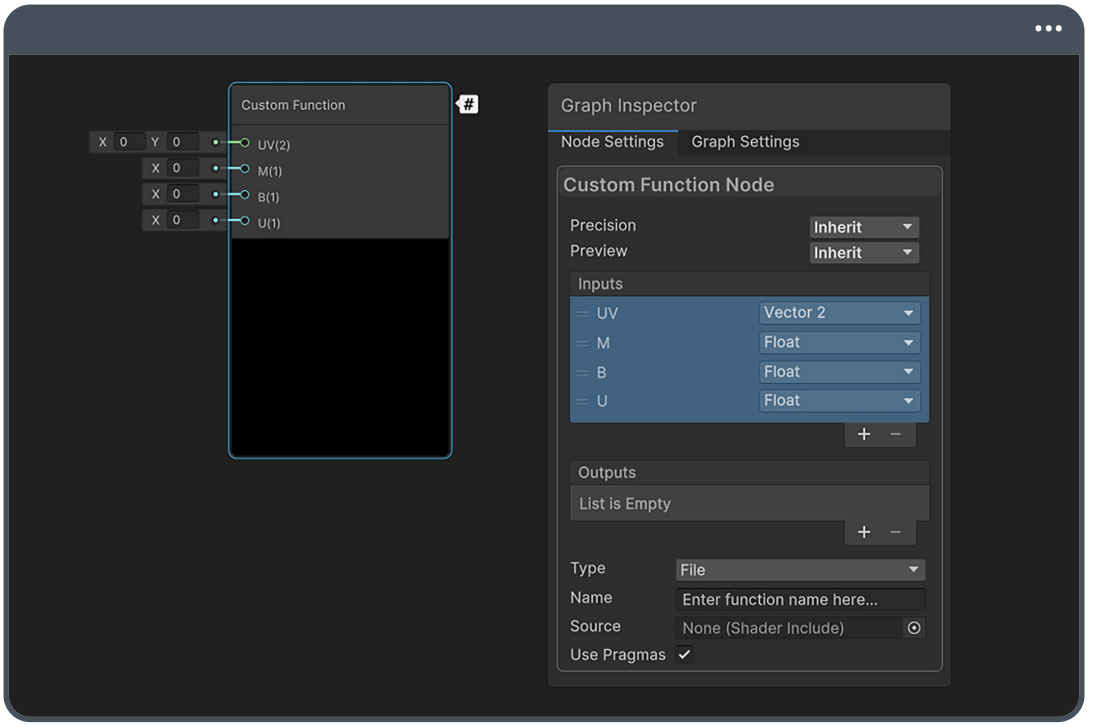
As for the inputs, you must consider the variables defined in the function presented in Reference 1.1.a in the previous section, i.e.:

- The variable m.
- The variable b.
- The xy coordinates, which are the same as UV.

Additionally, you'll incorporate the variable u, used to expand the function, as illustrated in Figure 1.1.d to visualize the horizontal shift that centers the function at the desired point.

When we refer to the Cartesian plane in computer graphics, we are actually talking about different spaces within the development environment. For example, global spatial coordinates, which define the Euclidean world in the Scene window, can function as a Cartesian plane. However, there are also screen coordinates or UV coordinates, which

represent spaces in a similar way. In this case, we'll choose to work with UV coordinates, because they provide a clear understanding of the concept of uv-space.



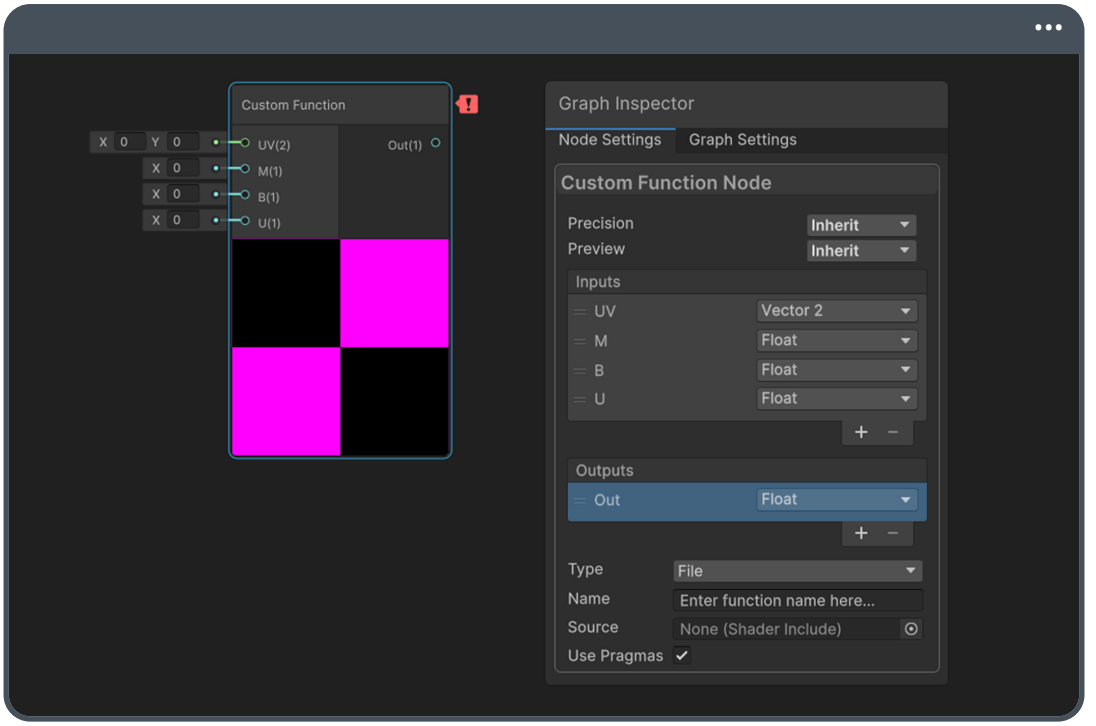
(1.2.d com.unity.shadergraph@17.0/manual/Custom-Function-Node.html)

If you look at Figure 1.2.d, you can see that the variables mentioned above have been integrated as inputs into the **Custom Function** node. It's important to highlight that the data type of the UV coordinates corresponds to a two-dimensional vector, as this will store the location of the pixels of the linear function on its x and y axes. As for the output, this varies according to the exercise being carried out. In this case, you'll return a floating value determined by a conditional. This value will be either 1.0 or 0.0, depending on a threshold that you can modify in real time through the variables m, b, and u of the linear function from the inspector.

It's important to highlight that, in this context, numerical values are translated into a range of colors. Therefore, 1.0 and 0.0 can be interpreted as white and black, respectively, for each pixel. In other words, if the **output** is equal to 1.0, the pixels within the function

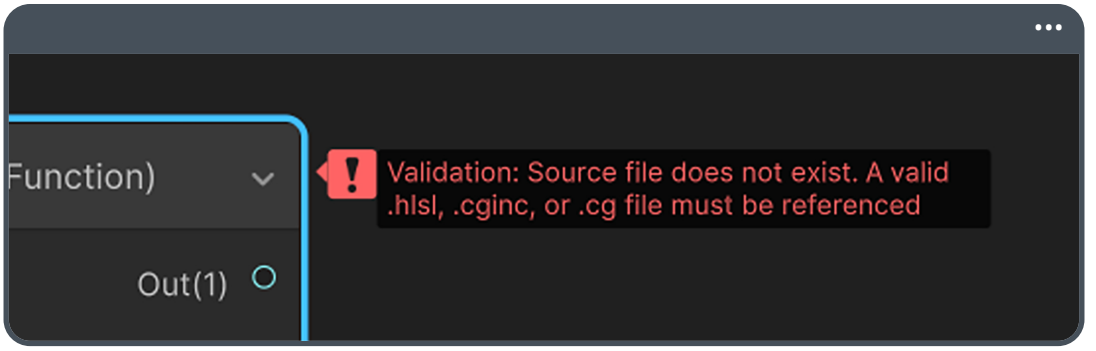
area will be displayed as white, whereas if the output is 0.0, they will be displayed as black. Similarly, a value of 0.5 will return in a gray color.

For practical reasons, you'll use **Out** as the exit name of the node, as presented in the following example:



(1.2.e Custom Function node error)

A crucial aspect of Shader Graph is that every time we make changes to the node's configuration—such as adding or removing inputs, altering the arrangement of variables in the lists, or updating functions in its HLSL file—common artifacts can arise. These artifacts can range from validations to overwriting variables. By looking closely at Figure 1.2.e, you can see an error identified that, in this case, is caused by a validation conflict. You haven't assigned a script with an **.hlsl** extension as Source and, furthermore, you haven't defined a name for the node.



(1.2.f Validation error)

To correct this error, you need to carry out the following steps:

- Assign the file **LinearFunction** to the **Source** property of the node.
- Use **linear_function** as the name for the node.

According to Microsoft's naming conventions for HLSL, data types, methods, and built-in functions should be presented in CamelCase. However, to improve readability, we'll use underscore-separated words, such as **`linear_function_half()`**, for the functions and methods implemented throughout this book.

It's important to note that since you haven't defined the previously mentioned method in the **LinearFunction.hlsl** file, it's possible that the error may persist even after making changes to the **Custom Function** node. Unlike C#, Unity doesn't include default code for this case, so you'll need to open your script and manually declare the following method from scratch:

```
1 void linear_function_half(in half2 uv, in half m, in half b, in half u, out
  ↪ half Out)
2 {
3     Out = 0.0;
4 }
```

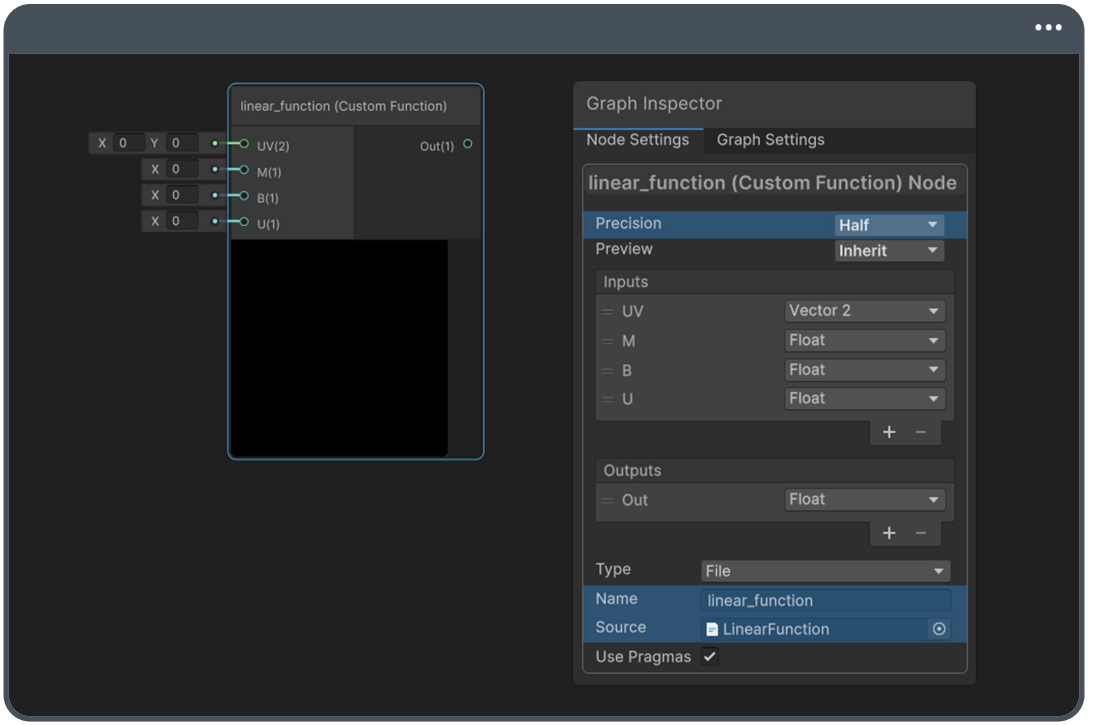
Examining the piece of code above, it's noticeable that the defined method and its arguments are of **`half`** type, corresponding to a half-precision data type, i.e., 16 bits. An interesting aspect of this data type, according to Unity documentation, is:

> **For platforms that support half, it will be 16 bits. On other platforms, this data type becomes float (32 bits).**

Although your function will not generate a significant load on the GPU, making the use of this type of data unnecessary, it'll be used for this exercise, so configure the node to **half** precision.

Incorporating the **`half`** data type into a program is an interesting choice, as it can potentially improve performance by reducing memory bandwidth, enhancing space efficiency, and even accelerating arithmetic operations. However, it also presents some disadvantages, such as loss of precision, particularly in coordinate systems. We'll explore those details in depth as we work through this book. Nonetheless, it's important to consider that the decision to use this data type should be evaluated based on the specific context and requirements of each project, always balancing efficiency and precision.

If everything has gone correctly, your node should produce a black color as a result, as illustrated in the following image.



(1.2.g)

The black color results from the output value temporarily defined as 0.0 in line 3 of your code. Since this value corresponds to a scalar data type, it can only display in grayscale. To obtain an RGB color as output, the data type must be changed from **`half`** to **`half3`**.

In this exercise you'll exclusively work with grayscale. Therefore, proceed to implement the linear function, starting with the declaration and initialization of its coordinates.

```
1  void linear_function_half(in half2 uv, in half m, in half b, in half u, out
   ↪  half Out)
2  {
3      half fx = uv.y;
4      half x = uv.x;
5
6      Out = 0.0;
7  }
```

Looking at line number 3, you can observe that the variable **fx** has been declared and initialized with the coordinate **uv.y**. Why is this the case? Because they both refer to the same concept: the y—coordinate on the plane. Subsequently, in the next line of code, you've declared and initialized the variable **x** and with the **uv.x** coordinate. Having defined both coordinates, the only remaining task is to define the linear function, similar to the one described in Figure 1.1.a in the previous section.

```
1  void linear_function_half(in half2 uv, in half m, in half b, in half u, out
   ↪  half Out)
2  {
3      half fx = uv.y;
4      half x = uv.x;
5
6      half f = m * x + b;
7      fx -= f;
8
9      Out = 0.0;
10 }
```

Looking at line 6, you can observe that the scheme for the linear function has been applied to the variable **f**, which subtracts from the abscissa function **fx** in the following line. As mentioned before, you'll use the extended function for this example, incorporating the variable **u** into the operation:

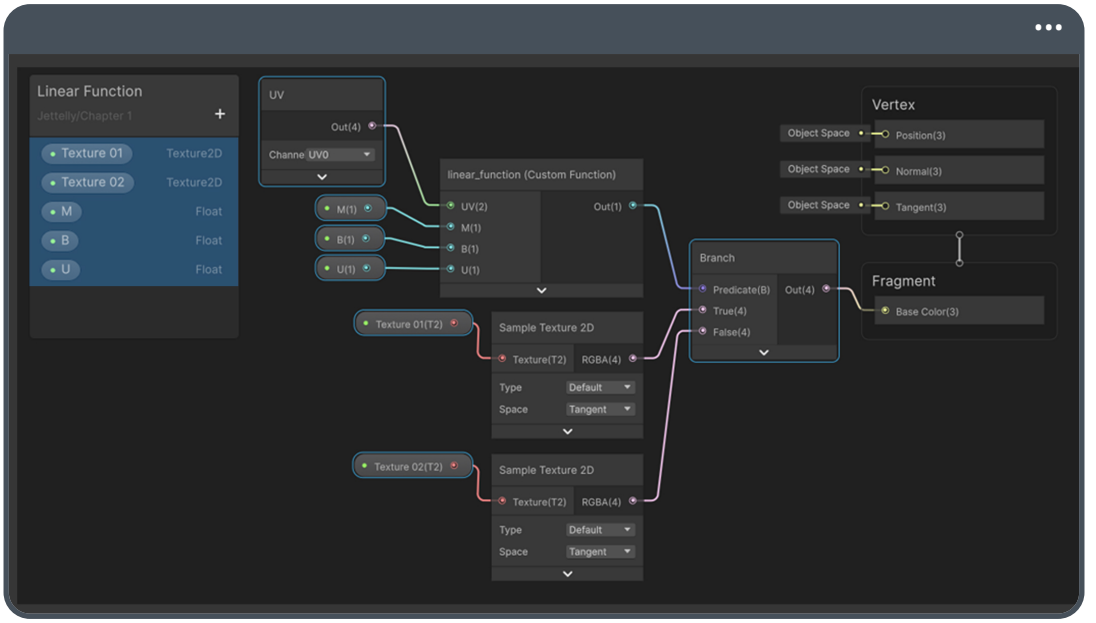
```
void linear_function_half(in half2 uv, in half m, in half b, in half u, out
    half Out)
{
    half fx = uv.y;
    half x = uv.x;

    half f = m * (x - u) + b;
    fx -= f;

    Out = 0.0;
}
```

A re-examination of the previous line of code reveals that **u** subtracts from **x**, which horizontally shifts the center point in the equation. Now all that remains is to define the output value of the function. To do this, you can use a conditional that will return 1.0 or 0.0, depending on whether **fx** is greater than 0.0. In this way, both the positive and negative areas of the linear function can be projected. This will be quite helpful later on since, for this exercise, you'll project two textures onto the different areas of the function, and then interpolate them according to the **f** value.

```
void linear_function_half(in half2 uv, in half m, in half b, in half u, out
    half Out)
{
    half fx = uv.y;
    half x = uv.x;

    half f = m * (x - u) + b;
    fx -= f;

    Out = (fx > 0.0) ? 1.0 : 0.0;
}
```

The next step focuses on manipulating textures and interpolating them using the **Branch** node in the Shader Graph. This node has the unique ability to return a value based on two conditions: true or false, and is configured through the **Boolean**-type **Predicate** property.

In this example, you will use the function to project two different textures, assigning each to the positive and negative regions of the linear function previously defined in line 9. However, before beginning this process, you'll need to declare some properties in **Blackboard** to be used in conjunction with the **linear_function** node. Additionally, you'll add the UV node, which represents Cartesian coordinates—specifically the coordinates of the first quadrant.



(1.2.h com.unity.shadergraph@6.9/manual/Branch-Node.html)

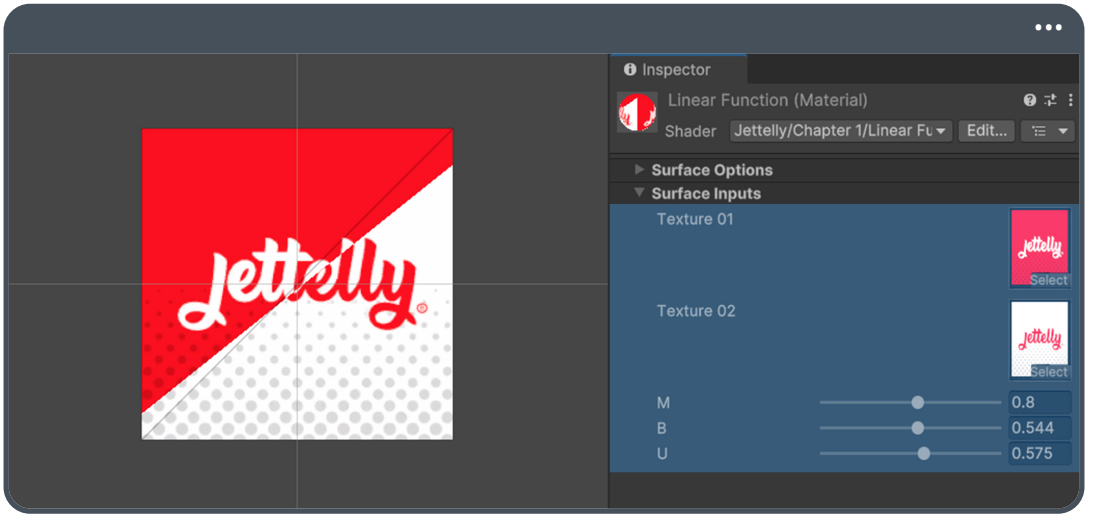
As you can see in Reference 1.2.h, five properties have been defined in the **Blackboard**: two textures—**Texture 01** and **Texture 02**—and three floating-point values: **m**, **b**, and **u**. Each of these properties adheres to specific ranges, previously studied, to prevent calculations from exceeding the area of the Quad, which will later be used as a reference to project the linear function. These limits ensure that the data remains within optimal

intervals, allowing adjustments to the function without causing rendering artifacts or inaccurate calculations.

- The **m** property spans a range between -10 and 10.
- While both the **b** and **u** properties have a range between 0.0 and 1.0.

These properties have been linked to the **linear_function** node, following their respective similarities. Likewise, the **UV** node has been linked to the same node. Then, the output of the **linear_function** node has been linked to the **Predicate** input of the **Branch** node. Why have we established this connection? Primarily, to ensure that the node returns both the value assigned to the **true** property and the **false** property in a single execution.

Finally, the result of the overall operation has been linked to the **Base Color** input in the **Fragment** processing stage. After saving the shader and returning to the **Linear Function** material, you can now adjust the variables of the linear function from the Inspector window. It's important to emphasize that two textures must be assigned to the material to display the operation of this function.



(1.2.i Visualizing the linear function)

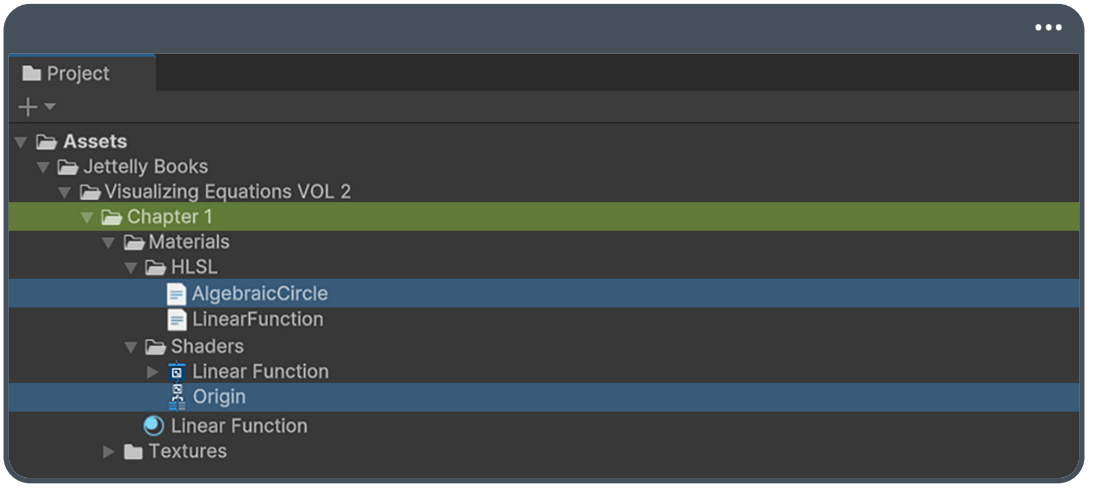
While dynamically adjusting property values, it becomes apparent that there is no graphical representation to help identify the origin of the linear function. Therefore, the next section will assist you in creating a new node for this purpose.

1.3 Visualizing the Origin.

Continuing with the process carried out so far, in the project create an object of type Sub Graph, and name it **Origin**. This element is located in the following path:

- Assets > Create > Shader Graph > Sub Graph.

At the same time, you can add a new HLSL script and call it **AlgebraicCircle**.



(1.3.a)

What is the purpose behind the creation of these objects? Sub Graph provides us with the ability to build a node that can be used inside a shader, analogous to how a function would be used in a **.cginc** file. Meanwhile, as you already know, the HLSL script allows us to generate a custom function.

Since the visualization of the origin point could be useful in multiple functions, it's essential to perform this action. With the new Sub Graph created, you'll proceed to customize the node to graphically represent the origin point of your linear function.

The equation you're going to use to draw it corresponds to:

$$x^2 + y^2 = r^2$$

(1.3.b)

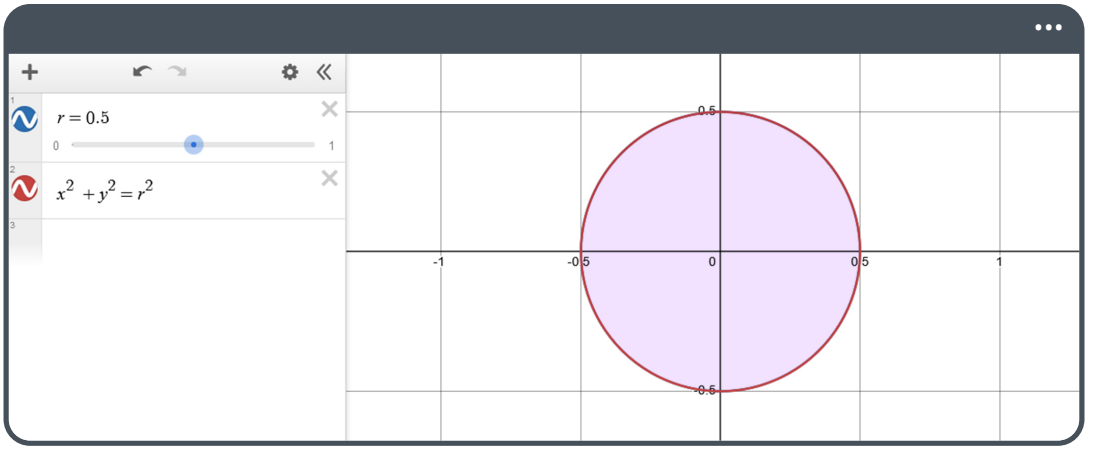
As can be seen in Figure 1.3.b, the equation describes a geometric area that takes the form of a circle. Why is this specific figure generated? The answer lies in the Pythagorean theorem, which establishes the fundamental relationship between the sides of a right-angled triangle. By making r constant, the solutions of this equation correspond to points that are all at a distance r from the origin, thus forming the area of the circle.

Note that, due to its nature, this equation doesn't fit the precise definition of a "function." Why? A function implies a special relationship between two sets, where each element in the first set corresponds to a single element in the second set. In other words, for each value of the independent variable x, there must be a single corresponding value in the dependent variable y.

$$y = \pm\sqrt{r^2 - x^2}$$

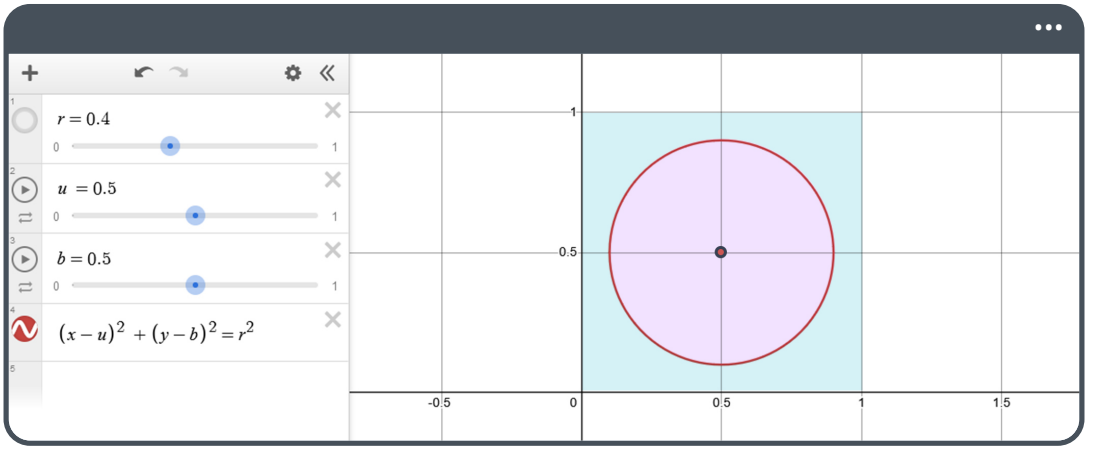
(1.3.c)

As you can see in Function 1.3.c, this relationship isn't true since, when Equation 1.3.b is reversed, the variable y can take both positive and negative values.



(1.3.d https://www.desmos.com/calculator/8i6kaohhtr)

If you represent the equation from Figure 1.3.b in a Cartesian plane, you'll observe that it remains static at the origin, which is inconvenient for your purpose, since the goal is for it to move in accordance with the origin of the linear function. Therefore, you'll need to modify your equation again, as illustrated in the next figure:

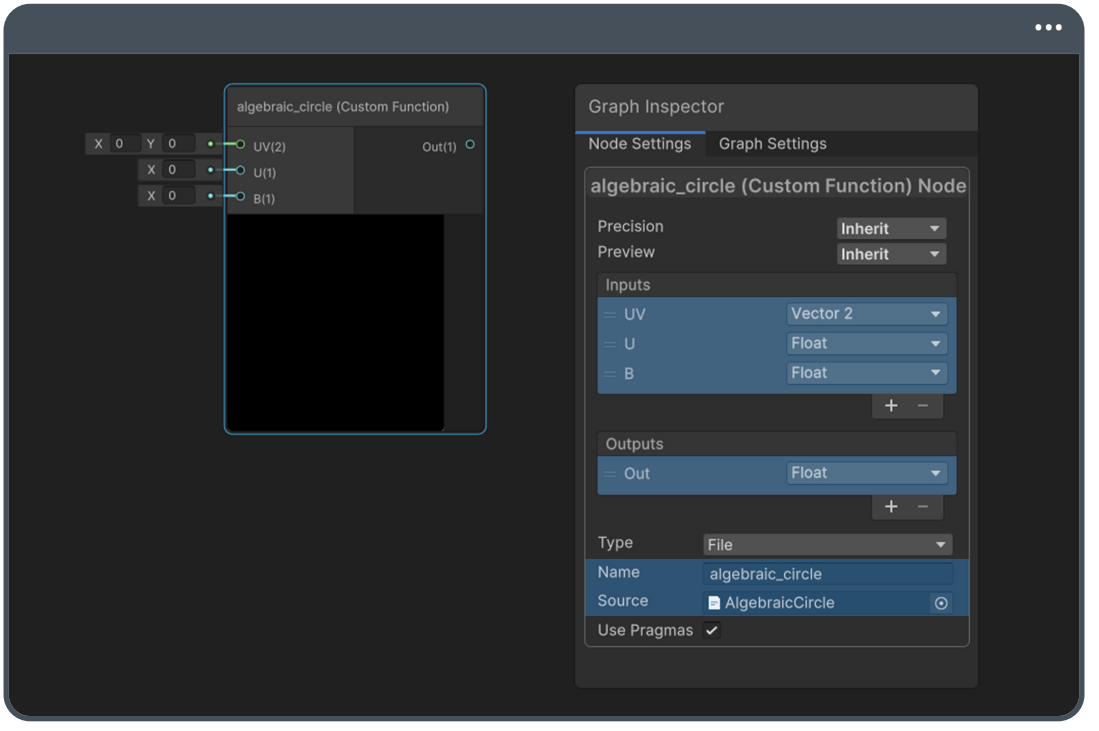


(1.3.e https://www.desmos.com/calculator/twzngfv0qj)

As observed, two new variables, u and b, have been incorporated into the equation, generating both a vertical and horizontal displacement of the circle on the Cartesian plane. Understanding this concept allows for further progress in the development of the Sub Graph, using a **Custom Function** node to create the circle.

Unlike a **.shadergraph** graph, a Sub Graph doesn't have a **Master** node; instead, it features only an **Output** with an input value that we can adjust in terms of both data type and precision. From this, you can deduce that the result of the operation performed within the graph must subsequently be connected to this node.

Since the circle is used exclusively to show the origin, you won't need to add RGB color functionalities to the function for now. Therefore, make sure to include the variables u and b, as well as the coordinates x and y of the equation, as **inputs** in the **Custom Function** node. For the output, assign a floating value. In addition, you can select the file **AlgebraicCircle.hlsl** as **Source** and use **algebraic_circle** as **Name**. It's important to remember that the xy coordinates correspond to the UV coordinates in the context of shaders.



(1.3.f)

Looking closely at Figure 1.3.f, you'll notice that the radius r hasn't been included as an input in the node. This is because you'll define a constant value for it later.

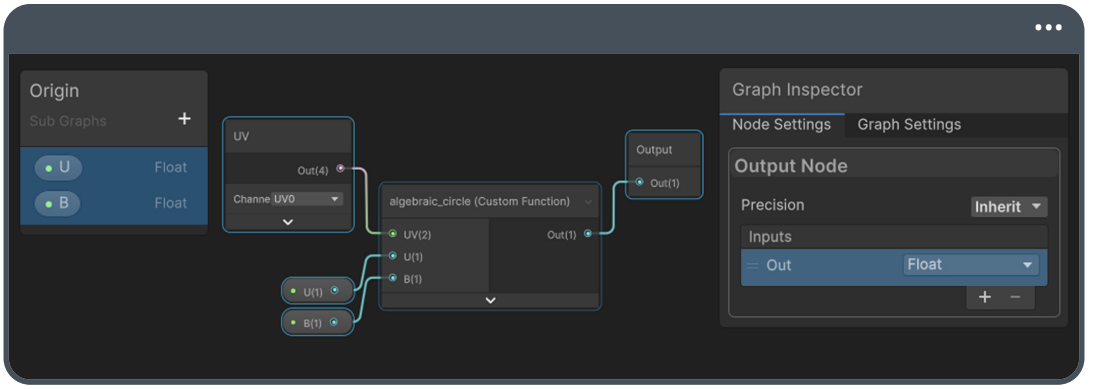
Before declaring the properties of the circle in the **Blackboard**, make sure to add the **`algebraic_circle_half()`** method to the HLSL script. Doing this guarantees an error-free build.

```
1  #define RADIUS 0.01
2
3  void algebraic_circle_half(in half2 uv, in half u, in half b, out half Out)
4  {
5      const half r = RADIUS * RADIUS;
6
7      half x = uv.x - u;
8      half y = uv.y - b;
9
10     half circle = x * x + y * y;  // pow(x, 2) + pow(y, 2)
11
12     Out = (circle <= r) ? 0.0 : 1.0;
13 }
```

As you can observe in Figure 1.3.b, the formation of the circle occurs when the sum of the squares of the coordinates xy equals the square of the radius r. Turning your attention to line number 5 from the code in the example above, note that the constant **`r`** is defined as twice the **`RADIUS`**. This same factor is reflected in code line 10, where the variable **`circle`** includes the coordinates **`x`** and **`y`** twice.

An interesting detail is the absence of the **`pow()`** function in the operation. Why aren't we using it this time? Mainly because it involves floating-point arithmetic, which could lead to performance issues on some GPUs. It's important to note that while this issue is largely resolved on modern GPUs, it's always prudent to minimize the number of calculations where possible.

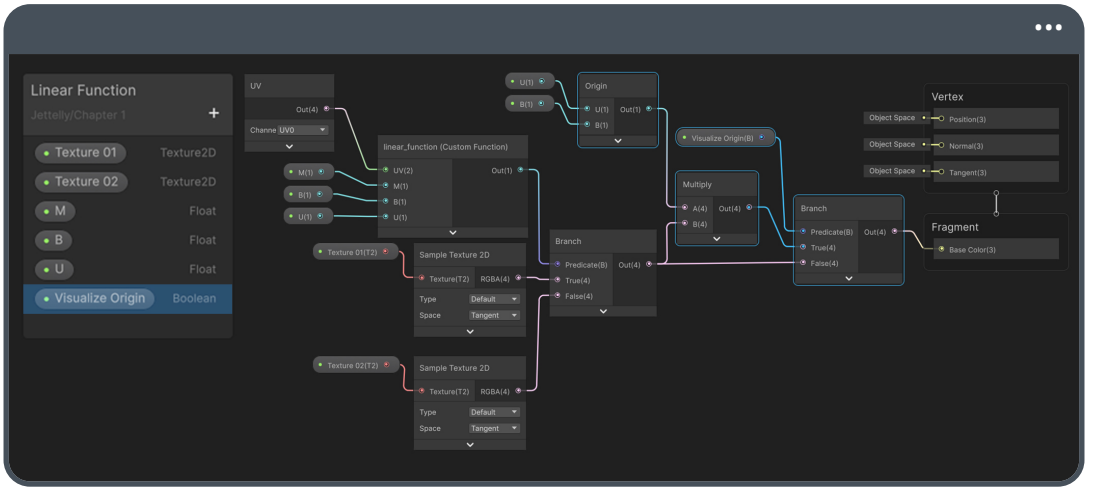
Finally, you'll return a black or white color depending on the circle's radius, as seen in code line 12. If everything has proceeded correctly, the node should compile without any problems. All that remains is to add the **`u`** and **`b`** properties to the **Blackboard** and connect them to the node to ensure it works correctly.



(1.3.g)

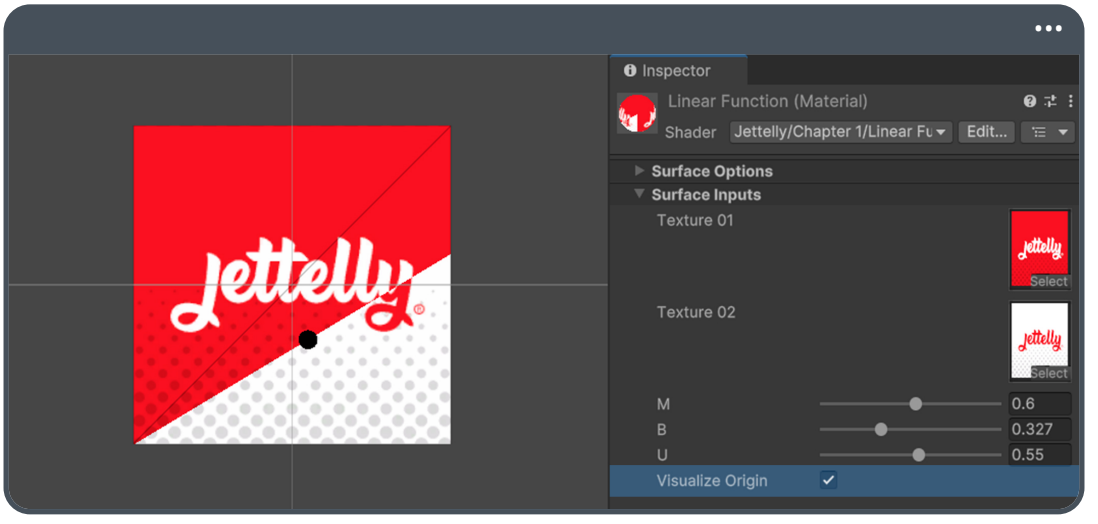
Next, you'll add a new feature to the **Linear Function** graph: The visualization of the origin. To achieve this, follow these steps:

1. Integrate the **Origin** node and connect the properties **u** and **b** as inputs.
2. Add a new **Boolean** property to Blackboard and call it **Visualize Origin**.
3. Multiply the result of the linear operation by **Origin**.
4. Include an additional **Branch** node to enable or disable displaying the origin.



(1.3.h)

If everything has gone perfectly, displaying the origin (which appears as a black dot by default) can be enabled directly from the **Visualize Origin** property in the Inspector window.



(1.3.i)

1.4 Drawing a Line Between Two Points.

When faced with the challenge of procedurally generating shapes, one of the equations most often encountered is one that draws a line between two points. The so-called line equation is a mathematical expression that provides an accurate geometric description of a straight line in a plane.

The most common form of this equation is known as the point-slope form, which is expressed as:

$$(y - a_y) = m(x - a_x)$$

(1.4.a)

Where xy are the coordinates of a point on the line, m is its slope, and $[a_x, a_y]$ correspond to the coordinates of the first intersection point. Simplifying the operation presented in the Figure 1.4.a, the equation can be expressed as follows:

$$y = m(x - a_x) + a_y$$

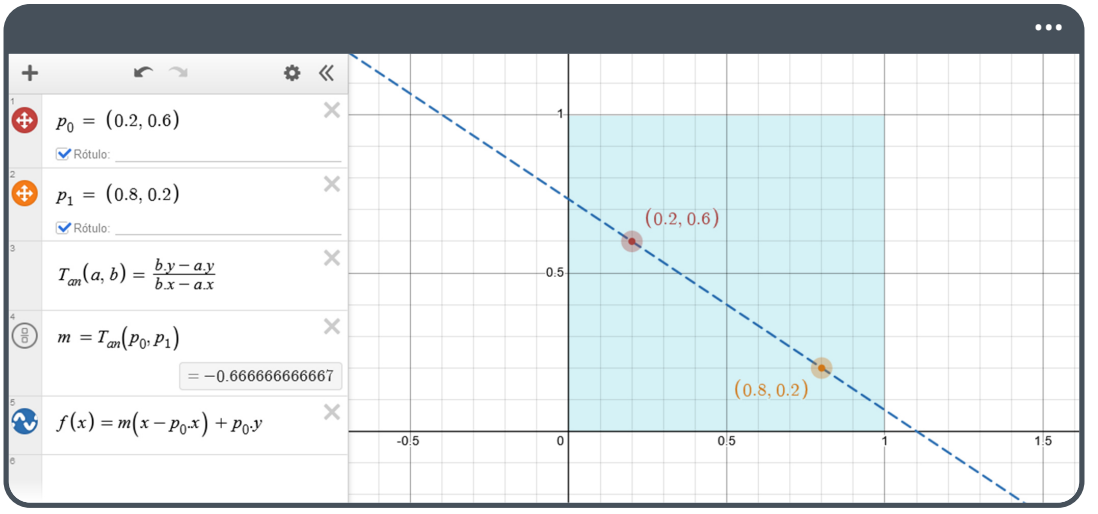
(1.4.b)

Looking closely at Figure 1.4.b, you'll notice that this equation represents the definition of a linear function in the form $y = mx + b$, which passes through the point $[a_x, a_y]$. If you wanted to draw a line between a point $[a_x, a_y]$ and another point $[b_x, b_y]$, the definition of the slope is given by:

$$m = \frac{b_y - a_y}{b_x - a_x}$$

(1.4.c)

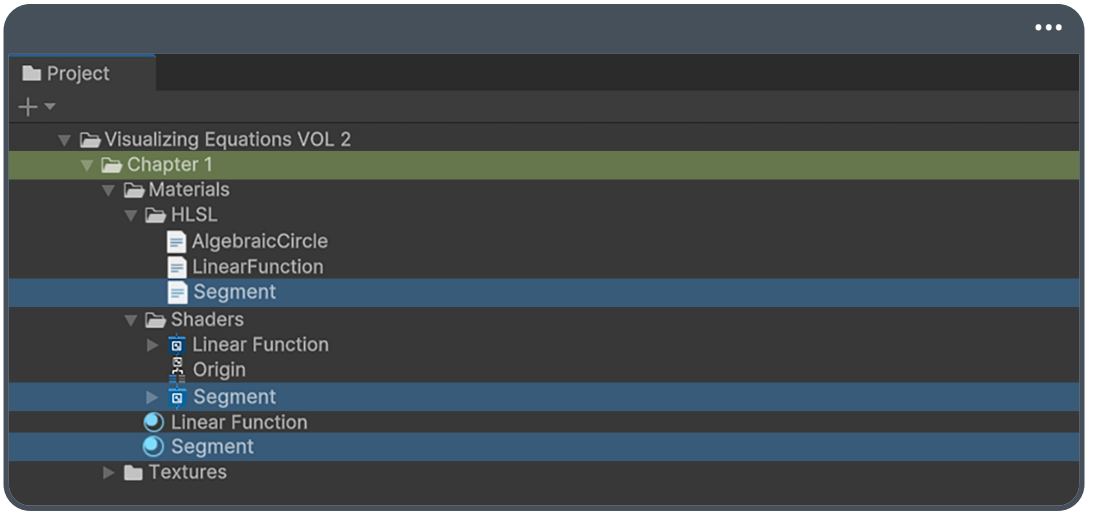
To draw a line between two points, the first step is to define the position of both points. Then, determine the value of the slope and, finally, draw the line, as illustrated in the following reference.



(1.4.d https://www.desmos.com/calculator/kk6nogiwlb)

Looking at Figure 1.4.d, two points p_0 and p_1 are defined in the first quadrant of the Cartesian plane. Next, a function called $T_{an}(a, b)$ has been introduced, which refers to the equation presented in Figure 1.4.c. Then, the variable m has been declared in order to improve the legibility of the slope before finally applying the equation of the line described in Figure 1.4.b.

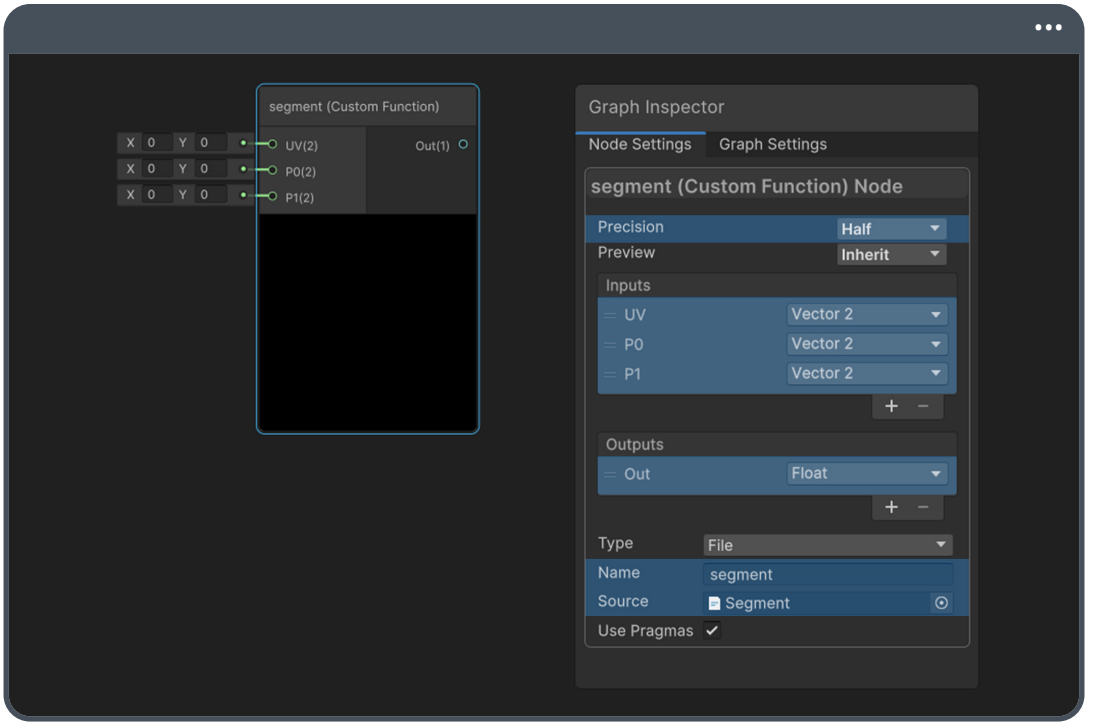
It's important to note that the line drawn in the previous exercise represents an infinite segment in both directions. Understanding this behavior is crucial, as in most cases, when drawing shapes using equations, you'll need to limit the length of the line. To illustrate this concept, create a new **Unlit Shader Graph** in the project and name it **Segment**. Additionally, generate both a new HLSL file and a material, ensuring they share the same name as before.



(1.4.e Project structure)

You'll use the **Segment.hlsl** file to build a custom function. In this case, the points defined in Figure 1.4.d will serve as inputs in the node, allowing for dynamic modification from the Inspector window. Next, you'll proceed with the following steps:

- Within the shader, create a **Custom Function** node.
- As inputs, add three two-dimensional vectors: The first one called **UV**; the second, **P0**; and the third, **P1**, maintaining the same order.
- As an output, return a floating-point value.
- Finally, designate **Segment.hlsl** as **Source**, and **segment** for its **Name**.



(1.4.f)

The first line of code you'll add to your script corresponds to the definition of the method itself, which includes the fundamental variables **UV**, **P0**, and **P1** as inputs, and **Out** as an output, as shown in the following example:

```
void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
{
    Out = 0.0;
}
```

However, considering the slope m in Figure 1.4.d, which will be used to draw a line between two points, you'll also need to implement its function within the code. To achieve this, a new method will be implemented, named **`tangent()`**, as shown below:

```
half tangent(half2 a, half2 b)
{
    return (b.y - a.y) / (b.x - a.x);
}

void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
{
    Out = 0.0;
}
```

This method (line 1) takes the points **a** and **b** as arguments in the same manner as presented in Reference 1.4.c. It's worth noting that, although these points could be named **p0** and **p1**, a different naming convention has been chosen to accurately represent the previously described function.

Subsequently, initialize both the **uv.x** and **uv.y** coordinates as well as the slope variable **m**, following the same scheme as the line equation:

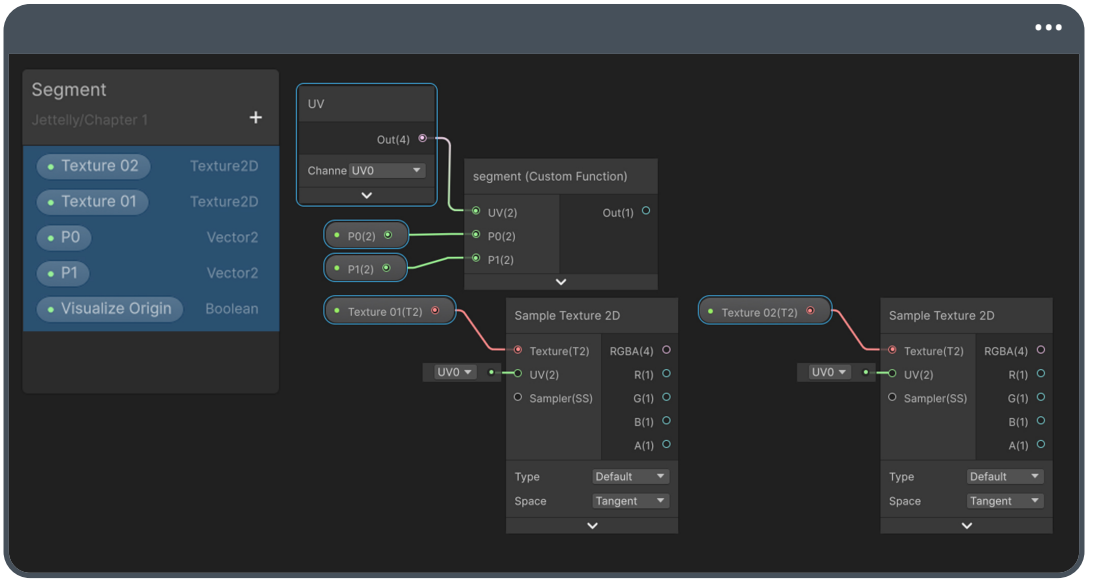
```
void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
{
    half fx = uv.y;
    half x = uv.x;
    half m = tangent(p0, p1);

    Out = 0.0;
}
```

To conclude, add the function that represents the segment between two points and return black or white according to the value of the variable **fx**. Note that the choice to return 1.0 or 0.0 is made solely as a simplification for this exercise to visualize the function more clearly. In later chapters of this book, we'll use a filter to generate colored areas to construct more complex procedural shapes.

```
void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
{
    half fx = uv.y;
    half x = uv.x;
    half m = tangent(p0, p1);

    half f = m * (x - p0.x) + p0.y;
    fx -= f;

    Out = (fx > 0.0) ? 1.0 : 0.0;
}
```

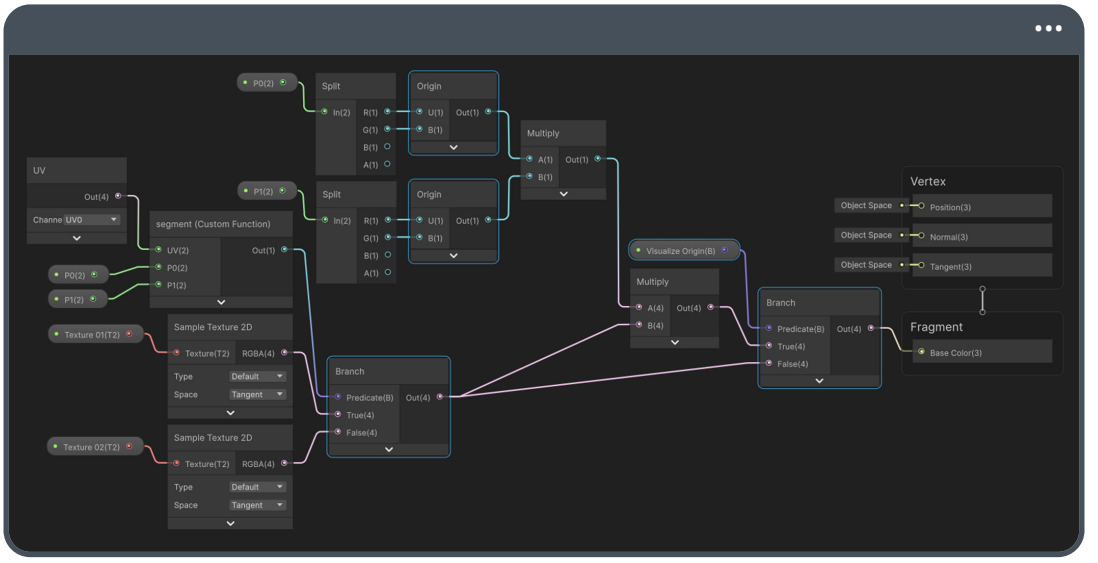
Up to this point, the node should compile without any issues. However, to ensure its full operation, you should declare the vectors **p0** and **p1** in the **Blackboard**.

It's important to point out that, with the aim of optimizing the content, you'll repeat part of the process discussed in previous sections. In this regard, add two **Texture2D** textures, named **Texture 01** and **Texture 02**, along with a **Boolean** property for visualizing the origin point. You can anticipate that these textures will be used to differentiate the various areas generated by the segment between the previously defined points.



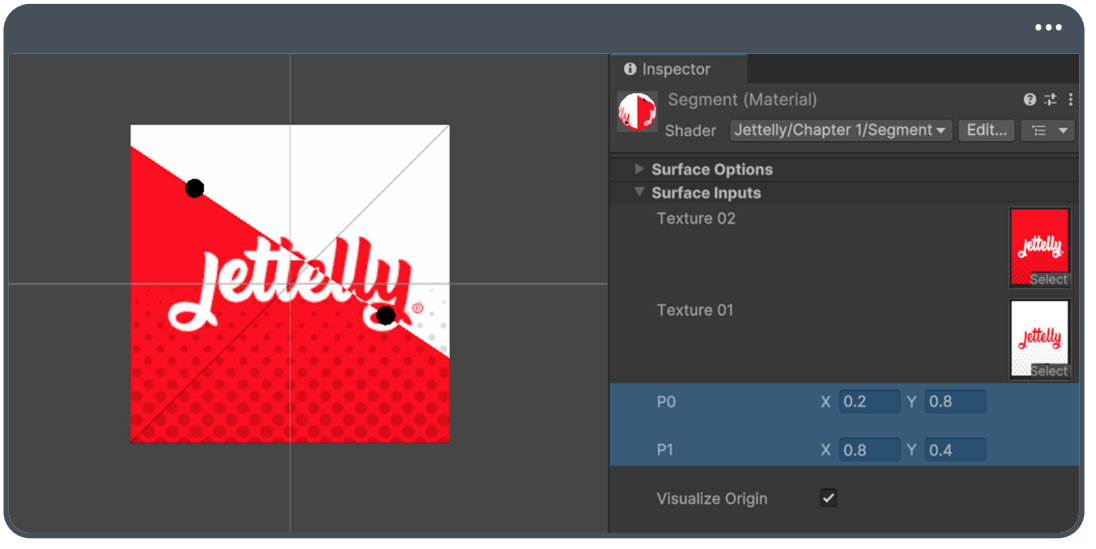
(1.4.g)

Using the **Branch** node, you can connect both textures to project the composition onto the Quad present in the Scene view. In addition, you need to include two **Origin** nodes to accurately display the position of the points, as illustrated in the following example:



(1.4.h)

In Figure 1.4.h, you can see that each two-dimensional vector, **P0** and **P1**, is respectively connected to an **Origin** node. However, as part of the intermediate process, a **Split** node has been used to separate the channels of each vector. By setting the default value of the vectors to a range between [0.0 : 0.5] for **P0**, and [0.5 : 0.0] for **P1**, you can easily observe the behavior of the segment generated between the two points. You should note that you can dynamically modify the value of both vectors from the Inspector window.



(1.4.i Visualizing Quad's two points)

1.5 Quadratic Function.

There are several situations in which we can use quadratic functions. For example, by flipping a coin in the air and modeling its movement. However, like linear functions, quadratics allow us to generate visually interesting patterns in representing graphics on a computer. In fact, we can use the parabola to draw curves that represent areas of lighting or even to define rounded edges on a specific figure. Now, how can we do this?

To understand this concept, pay attention to the following equation:

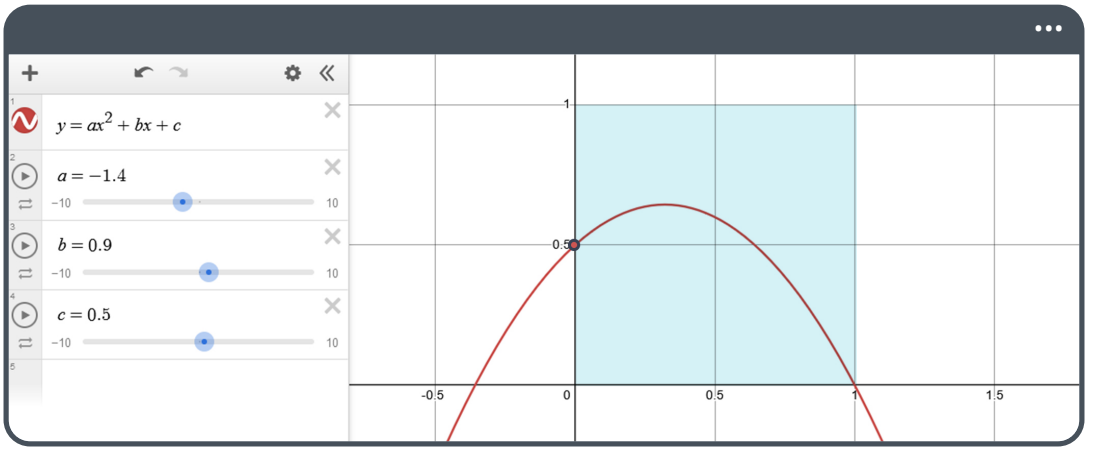
$$f(x) = ax^2 + bx + c$$

(1.5.a)

The constant coefficients, represented by the variables a, b, and c, influence the function, while x denotes the independent variable. In terms of output, $f(x)$, equivalent to y, represents the result of the function: a curve.

When displaying this function in the coordinate system, observe that the quadratic coefficient a controls the concavity and aperture of the parabola. If positive, the parabola opens upwards (convex), if negative, it opens downwards (concave).

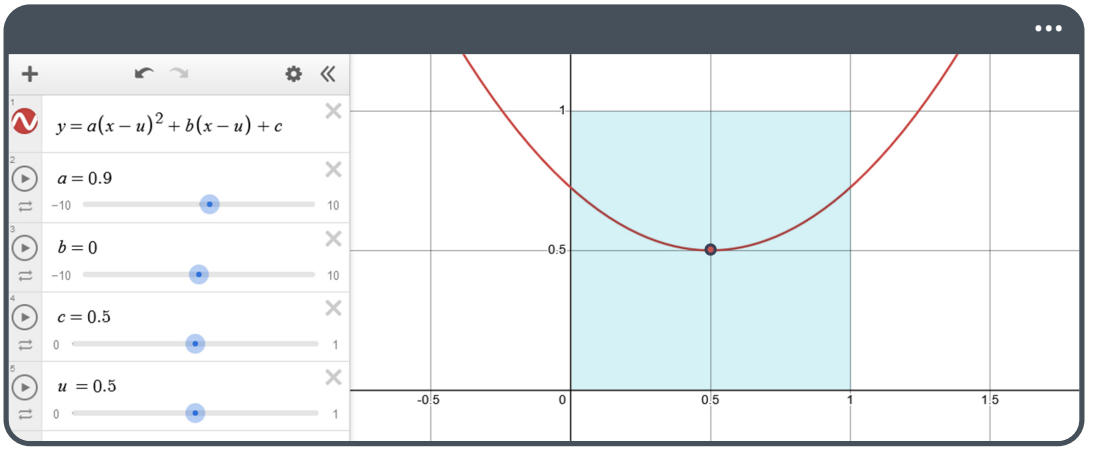
The variable b, or linear coefficient, affects the horizontal position of the parabola. A positive value moves it to the left, while a negative value moves it to the right. Finally, the variable c, which corresponds to the constant term, determines the vertical position of the parabola on the y-axis.



(1.5.b https://www.desmos.com/calculator/yjuz1wwet4)

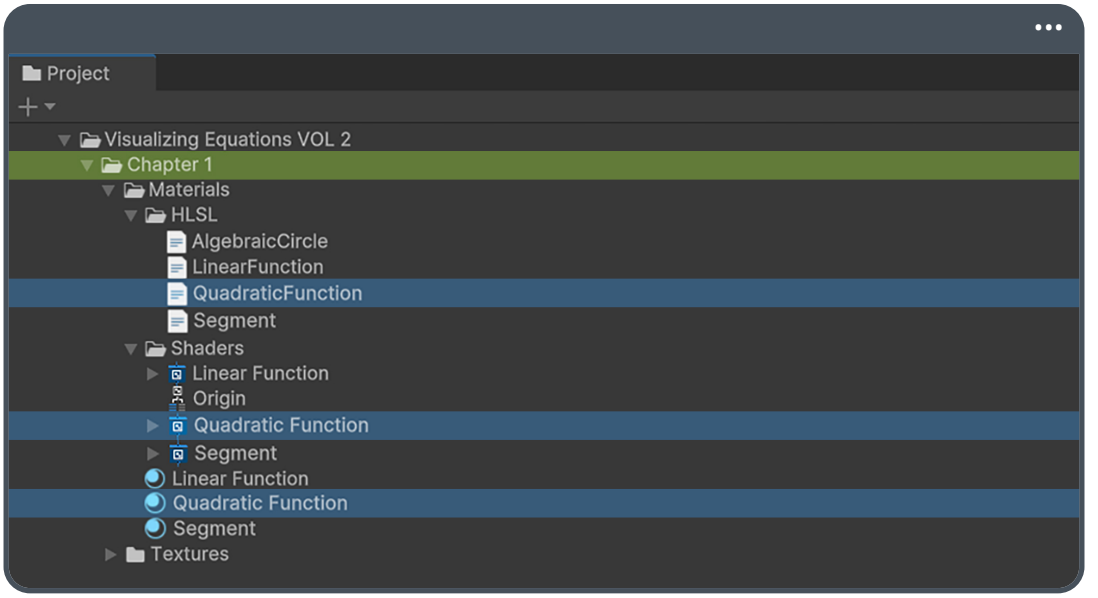
When working in HLSL and planning to use the parabola frequently—like other functions—we need to horizontally modify its origin. To achieve this, the solution is quite simple: we incorporate a variable that subtracts from the independent variable

x. This adjustment allows us to move the parabola horizontally in the Cartesian plane, thereby providing greater flexibility in manipulating its position and adapting it more precisely to our needs.



(1.5.c https://www.desmos.com/calculator/bex75gquqi)

In Figure 1.5.c, we've introduced a new variable, u, to the quadratic function. By subtracting u from the independent variable x, we've achieved a horizontal displacement of the origin, which is graphically represented as a black dot in the image. This adjustment leads to a key question: How can we implement this operation in HLSL? To answer this, you'll create a new shader in the project, and additionally you'll generate a **.hlsl** script, and a corresponding material. This setup will facilitate the development of a custom node, allowing for direct visualization of the function on the Quad in the Scene view. For simplicity, you'll name the three newly generated elements **Quadratic function**.

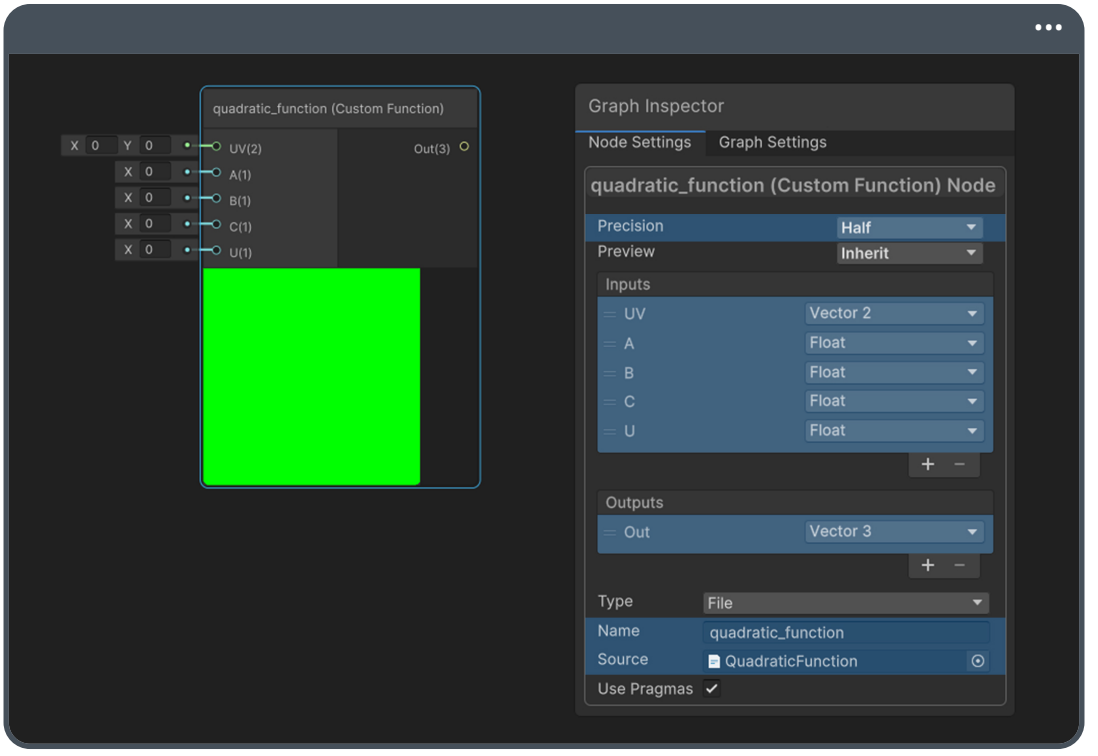


(1.5.d)

Then, you'll proceed to define the properties of the quadratic function within a **Custom Function** node. Considering the equation in Figure 1.5.a, it can be deduced that the input variables correspond to the coefficients a, b, and c, which are floating-point values that determine the shape and position of the parabola. However, to expand the function, it'll also be necessary to include the UV coordinates and the variable u, allowing for dynamic interaction with the origin point. Similar to the coefficients, u is also a floating-point variable.

In this case, it's also necessary to define the output value. However, this time, you won't implement textures in the shader; instead, you'll use only colors to explore creating procedural figures. Because of this, you'll configure the **output** as a three-dimensional (RGB) vector.

To link these properties to the code, you can select and drag the **QuadraticFunction.hlsl** file into the **Source** field in the Graph Inspector. In addition, you can set **quadratic_function** as the name of the function, making sure it matches its HLSL implementation.



(1.5.e)

With these configurations in place, you're now ready to define the quadratic function within the HLSL file, incorporating the variable **u** as an argument to enable dynamic interaction with the origin point.

```hlsl
void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
    in half u, out half3 Out)
{
    half fx = uv.y;
    half x = uv.x;

    Out = 0.0;
}
```

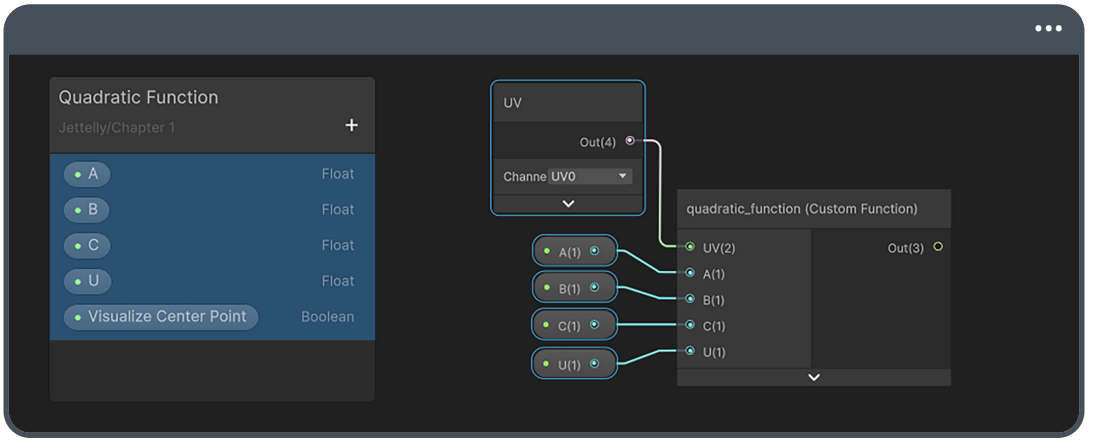
Looking closely at line 3, notice that the function of the abscissa, **fx**, has been defined and initialized with the coordinate **uv.y**. Similarly, the variable **uv.x** has been initialized and defined with its respective coordinate. Therefore, all that remains is to specify the quadratic function, that is, the relationship between these variables that will shape the parabola in your shader. Considering the standard quadratic function, you can implement the equation using its input variables: a, b, c, x, and y. Additionally, for greater flexibility, include the variable u.

```
1  void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
    ↪  in half u, out half3 Out)
2  {
3      half fx = uv.y;
4      half x = uv.x;
5
6      half xu = x - u;
7      half f = a * (xu * xu) + b * xu + c;
8
9      Out = 0.0;
10 }
```

Paying attention to line 7 of the code, you'll notice that the variable **f** takes the quadratic, linear, and constant coefficients, as well as the respective coordinates, and returns the value of the parabola. To conclude the exercise, this time, define two constant random colors, which will represent both the positive and negative areas of the quadratic function. It should be noted that the choice of colors is for demonstration and simplification purposes of the exercise only. In more advanced practices, these could be dynamic values or respond to more specific logic.

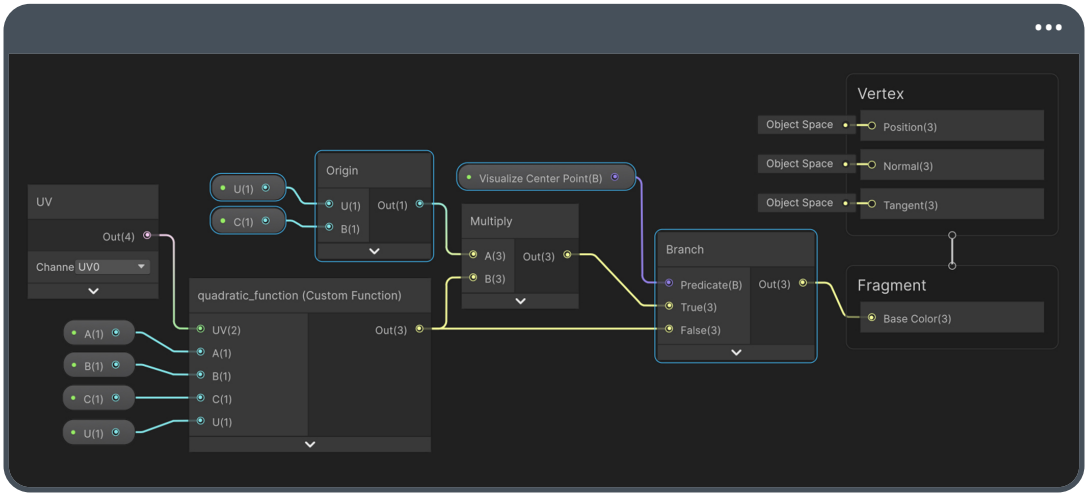
```hlsl
void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
    in half u, out half3 Out)
{
    half fx = uv.y;
    half x = uv.x;

    half xu = x - u;
    half f = a * (xu * xu) + b * xu + c;

    const half3 red = half3(1, 0, 0);
    const half3 green = half3(0, 1, 0);

    fx -= f;
    Out = (fx > 0.0) ? red : green;
}
```

In lines 9 and 10, you can see the declaration and initialization of two three-dimensional vectors, which define the colors red and green in your code. A conditional structure in line 13 has been employed to determine both the positive and negative areas of the quadratic function. At this point, your **Custom Function** node should compile without any issues. However, as in previous exercises, Shader Graph may require an adjustment to the node's resolution in the Graph Inspector. In this case, set the **Precision** property to **Half**, and set the properties in the **Blackboard** to dynamically modify the function from the Inspector.



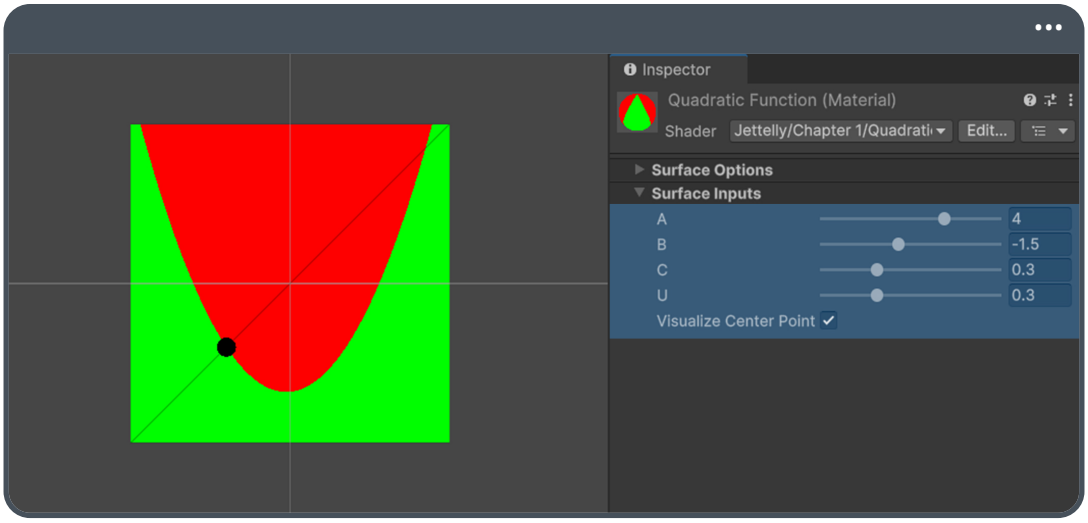
(1.5.f)

Turning your attention to the **Blackboard** in the figure above, notice that a Boolean property called **Visualize Center Point** has been added. Following the same methodology as previous exercises, you can use this property to enable or disable displaying the origin in your function. Consequently, you'll again use a **Branch** node in the shader, with a positive value equal to the multiplication of the quadratic function by the output of the **Origin** node.



(1.5.g)

On saving the changes and returning to the scene, you'll be able to dynamically experience how different values affect the shape and position of the parabola, thereby providing a deeper understanding of how coefficients influence the visual appearance of the shader.

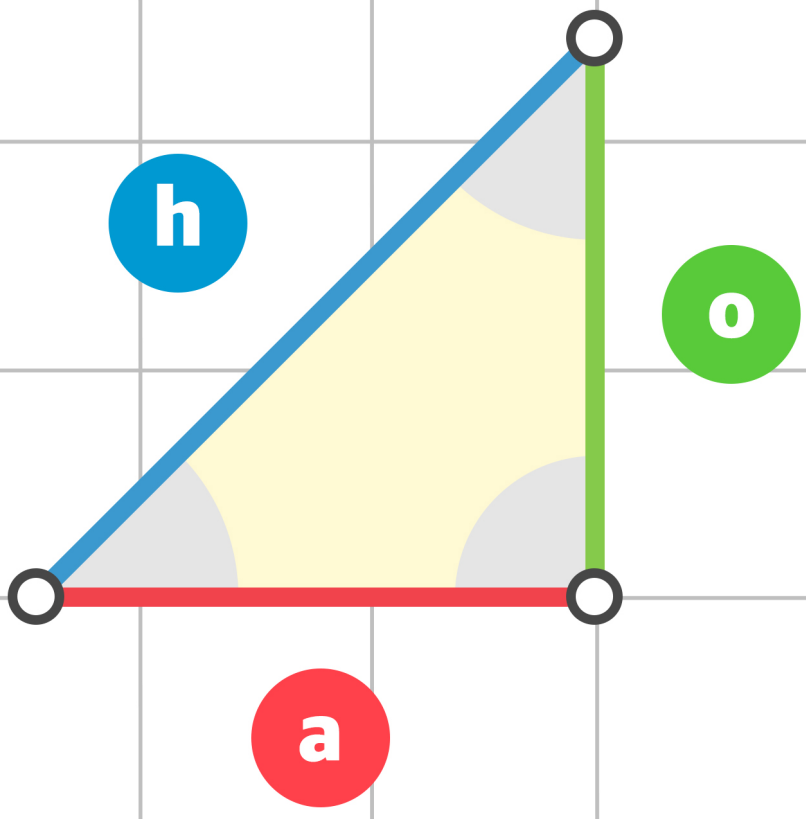

Inspector
Quadratic Function (Material)
Shader
Jettelly/Chapter 1/Quadrati
Edit...
Surface Options
Surface Inputs
A
4
B
-1.5
C
0.3
U
0.3
Visualize Center Point

(1.5.h)

Chapter Summary.

- In this chapter, we illustrated the importance of the linear function in computer graphics, beginning with an analysis of each variable in the equation. You used the linear function to draw a segment between two points in Shader Graph with HLSL, demonstrating how different variable values determine the line's behavior on a Cartesian plane.

- The chapter continued with an exploration of the first geometric area, focusing on a circle. This section covered fundamental concepts such as the importance of Sub Graphs and **.cginc** files in shader development. The discussion then progressed to the analysis of quadratic functions, exploring their application in generating parabolas. Finally, the equation was broken down to show how the coefficients influenced the parabola's shape and position.



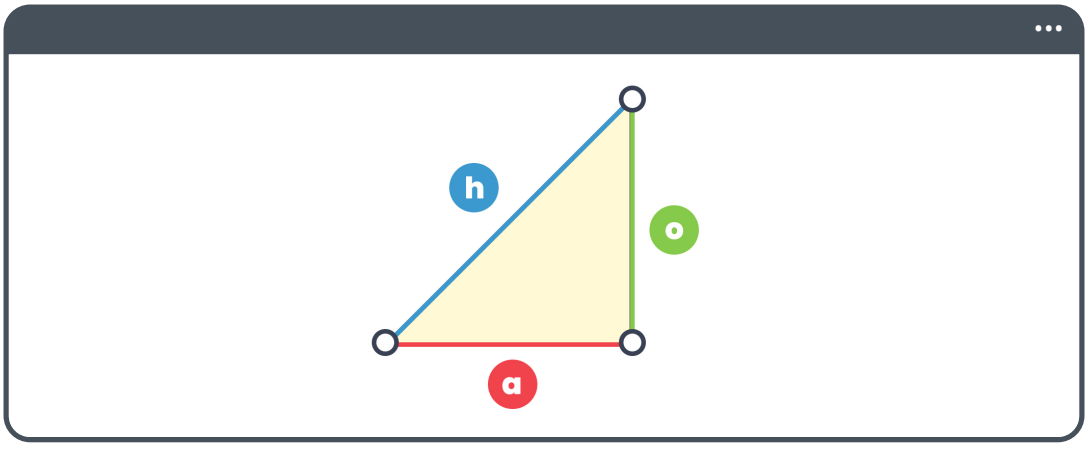
Chapter 2

Trigonometric Functions.

In this chapter, we'll dive into the world of trigonometric functions, an essential tool for programmers, designers, and technical artists alike. We'll explore how to visualize these functions and build the foundational knowledge needed to understand and apply these mathematical principles in HLSL. Later, we'll set aside the theoretical concepts and focus on two fundamental operations: reflection and rotation of points in two-dimensional space. Through practical examples, we'll learn how to apply these concepts to generate shapes and patterns within our Unity projects, unlocking a wide range of creative and technical possibilities for visual effects development.

2.1 Functions.

Trigonometric functions, fundamental in disciplines such as physics and engineering, also play a crucial role in digital simulations and animations. In computer graphics, functions such as sin (sine), cos (cosine), and tan (tangent) play indispensable roles in the creation of visual effects, color variations, spatial transformations, and even in designing specific patterns such as sinusoidal waves. But where do these functions originate? They derive from the study of the relationships between the angles and sides of a right triangle, so named because it includes a 90° angle. Let's examine the following figure to explore this concept further:



(2.1.a)

Considering that a triangle has three sides—labeled h, a, and o in Figure 2.1.a—we can explore various combinations to better understand their relationships and applications in trigonometry. In fact, by using these three sides, we can define a total of six unique combinations:

$$\frac{o}{h}; \frac{o}{a}; \frac{h}{o}; \frac{h}{a}; \frac{a}{h}; \frac{a}{o}$$

(2.1.b)

Each of these ratios forms the basis of a fundamental trigonometric function. For example, **sine** represents the ratio between the side opposite the angle and the hypotenuse of a right triangle:

$$sin(\theta) = \frac{o}{h}$$

(2.1.c)

Conversely, the cosecant csc of an angle is given by:

$$csc(\theta) = \frac{h}{o} = \frac{1}{sin(\theta)}$$

(2.1.d)

Next, the cosine of an angle is given by:

$$cos(\theta) = \frac{a}{h}$$

(2.1.e)

While its reciprocal, the sec is given by:

$$sec(\theta) = \frac{h}{a} = \frac{1}{cos(\theta)}$$

(2.1.f)

Finally, the tangent of an angle is given by:

$$tan(\theta) = \frac{o}{a}$$

(2.1.g)

And its reciprocal, the cotangent cot of an angle is given by:

$$cot(\theta) = \frac{a}{o} = \frac{1}{tan(\theta)}$$

(2.1.h)

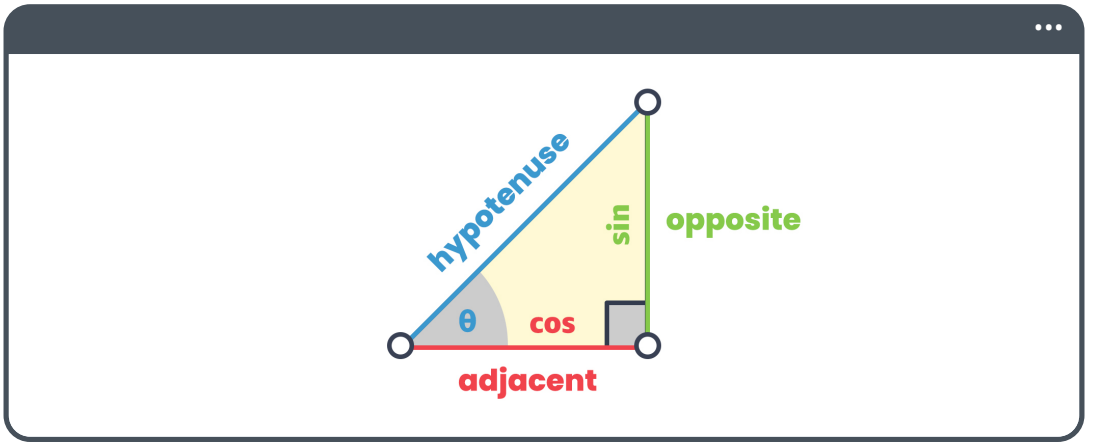
HLSL doesn't include direct functions for cosecant, secant, and cotangent, as they are simply the reciprocals of sine, cosine, and tangent, respectively. Instead of analyzing these reciprocal functions in detail, we'll focus on sine, cosine, and tangent, since HLSL provides them natively, making them more practical for shader development.

To illustrate this, let's consider the sine function, expressed as:

$$f(x) = a\ sin(bx + c)$$

(2.1.i)

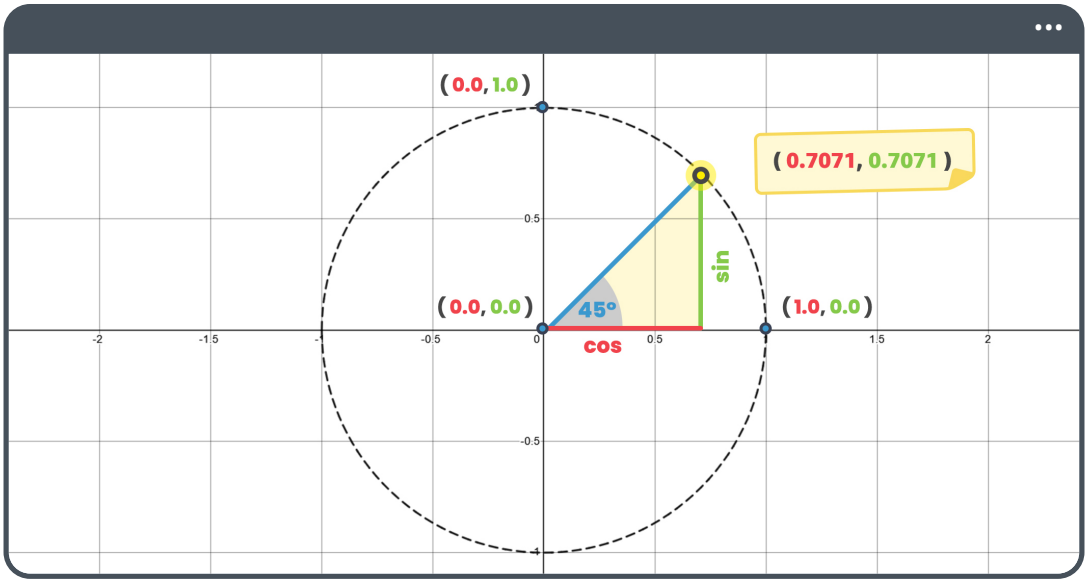
Before delving into the details of this function, it's crucial to understand what it represents and its connection to the geometry of a right triangle. The sine of a specific angle in this type of triangle is calculated by dividing the length of the opposite side of the length of the hypotenuse.



(2.1.j)

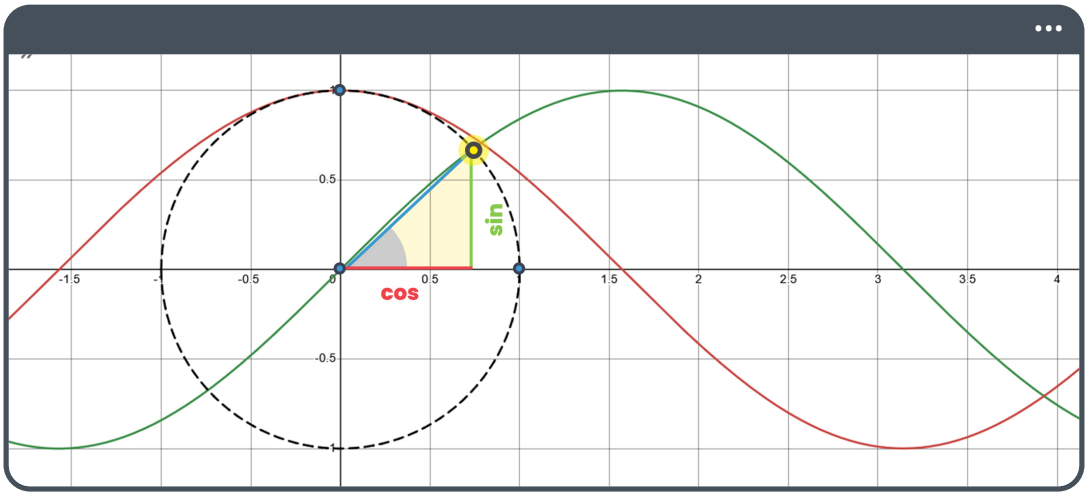
If we take a closer look at Figure 2.1.j, we can see that red and green have been used to differentiate the sine and cosine functions within the context of a right triangle. This color selection helps visually distinguish these functions, making it easier to understand their application in UV coordinates.

Considering a hypotenuse of constant length, denoted as h, and placing a point p at its end, we observe that, as this point rotates around the origin, it traces a complete circle. This observation highlights the close relationship between trigonometric functions and circular geometry, a concept briefly introduced in Section 1.3 of the previous chapter.



(2.1.k https://www.desmos.com/calculator/vcx4kxwivn)

An interesting aspect emerges when examining the curves formed in the Cartesian plane by calculating the sine or cosine of a number. Consider, for example, the x-axis as the input variable. When evaluating the sine and cosine functions along this axis, a distinct pattern emerges in the resulting graph. This pattern not only illustrates the periodic nature of these functions but also highlights their symmetry and how they complement each other across the range of x values.



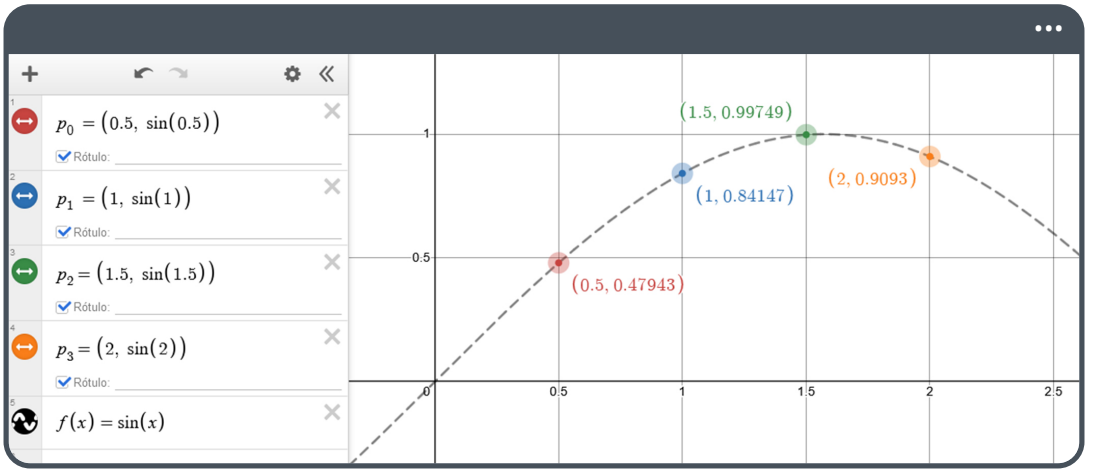
(2.1.l https://www.desmos.com/calculator/y07lbi4req)

The visualization of sine and cosine curves becomes evident when both functions are plotted simultaneously in a single coordinate system. One might ask, what causes these curves to form? It's important to remember that the Cartesian coordinate system consists of a continuous sequence of numerical values, extending from negative infinity to positive infinity, defining positions in two-dimensional space. To explore this concept further, let's consider the definition of four specific points along the x-axis.

$$p_0 = (0.5, sin(0.5) = (0.5, 0.479)$$
$$p_1 = (1.0, sin(1.0) = (1.0, 0.841)$$
$$p_2 = (1.5, sin(1.5) = (1.5, 0.997)$$
$$p_3 = (2.0, sin(2.0) = (2.0, 0.909)$$

(2.1.m)

Let's use the sine function to determine the y-component of each of these points, which will allow you to further explore how variations in x directly influence the heights (y-values) generated by the sine function, thus tracing the characteristic sine wave.



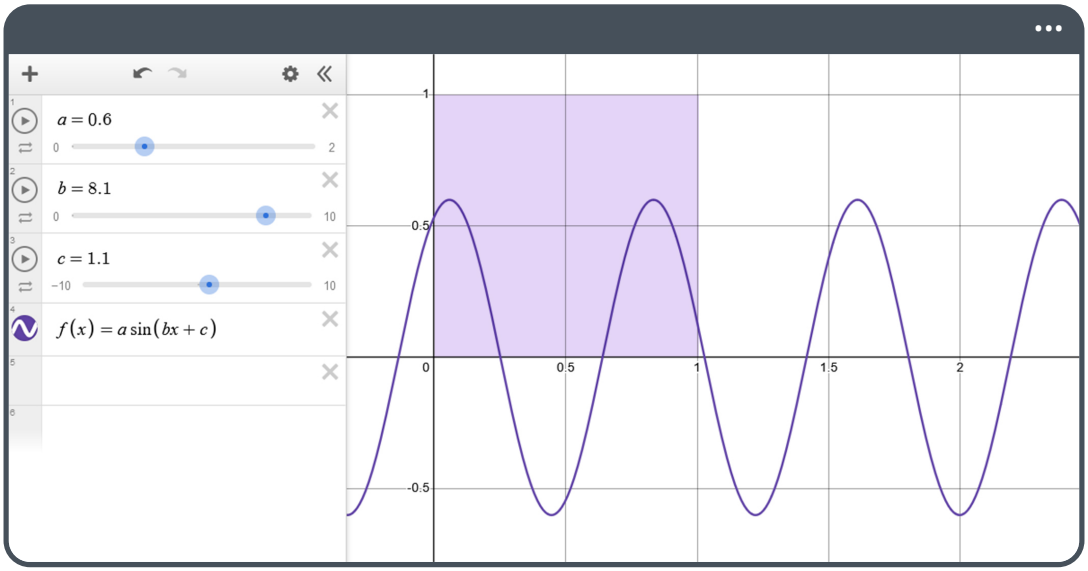
(2.1.n https://www.desmos.com/calculator/gltqybpvp8)

As shown in the previous reference, the sine curve is formed by calculating the sine of each point along the x-axis, with the same applying to the cosine function. This difference occurs because the sine of an angle is equivalent to the cosine of the same angle shifted by 90° (or $\pi/2$ radians), meaning:

$$sin(\theta) = cos(\theta \pm \frac{\pi}{2})$$

(2.1.o)

The curves can be modified in amplitude, frequency, and phase. Revisiting Figure 2.1.a, we can see that the variable a modifies the amplitude of the wave, b affects its frequency, and c adjusts its phase. How can we take advantage of this function in HLSL? A practical example is creating 2D fluid simulations by manipulating the x coordinate within the UV parameters.



(2.1.p https://www.desmos.com/calculator/kns4ip5cd0)

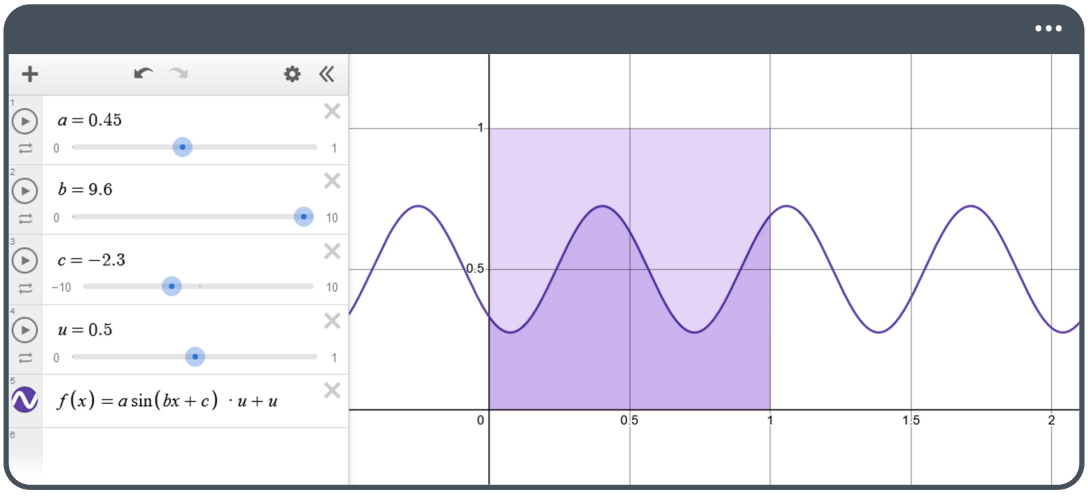
Examining Figure 2.1.p, we can see the aforementioned function which, with a value of $a = 0.6$, shows an amplitude ranging from -0.6 to 0.6. However, remember that the range of the UV coordinates differs from this. Therefore, it'll be necessary to adjust the

function so that its range is between 0.0 and 1.0. To do this, we'll introduce a new variable called u, assigning it the value 0.5. This variable u will allow us to recalibrate the central point of the curve, so that its origin in the y-coordinate is at a distance equivalent to its value. Subsequently, when applying the operation, we'll multiply the sine function by u and then add u to it, as follows:

$$f(x) = a\ sin(bx + c)\ u + u$$

(2.1.q)

It's important to mention that the variable a also needs to be limited to 1 to align it with the range of the UV coordinates.



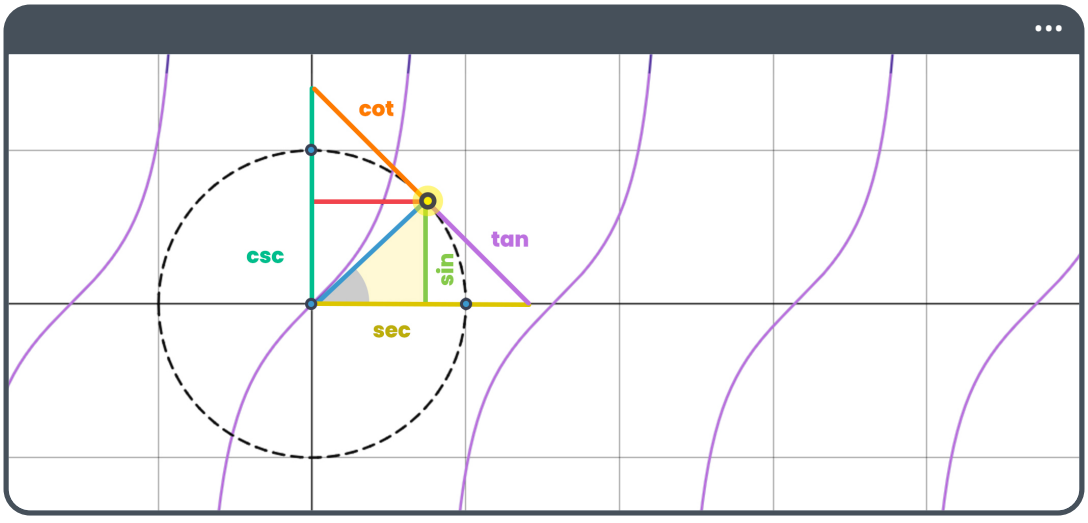
(2.1.r https://www.desmos.com/calculator/1scbrmw7zm)

So far, we've examined the behavior of the sine and cosine functions in detail. But what about the tangent function? The tangent is defined as the ratio of the opposite side to the adjacent side, as shown in Figure 2.1.g. Thus, when the angle of a point with respect to its origin is greater than zero and less than 90°, the result of the tangent is positive. Why? In this range of angles, both the opposite and adjacent sides are positive. However, as the angle increases beyond this range, the sign of the tangent changes.

For the angles in the ranges of:

$0° < \theta < 90°$	$90° < \theta < 180°$	$180° < \theta < 270°$	$270° < \theta < 360°$
$tan(a)+$	$tan(a)-$	$tan(a)+$	$tan(a)-$

This characteristic change in sign highlights the tangent's periodicity and unique behavior across different quadrants. When graphing its curve, this alternating sign pattern leads to a series of discontinuities that define its properties.



(2.1.s https://www.desmos.com/calculator/czpygnurms)

The tangent proves to be a versatile tool in the creation of various visual effects, ranging from rotations to the application of texture masks. Its utility extends further, facilitating the estimation of attributes such as the height of an object or the orientation of a point along a curve. This is particularly evident when considering the slope m of a linear function, represented by $y = mx + b$.

Take, for example, a point p, defined as:

$$p = (n, sin(n))$$

(2.1.t)

If m is considered as the slope of the line, defined as:

$$m = tan(\frac{\pi}{4} * cos(p_x))$$

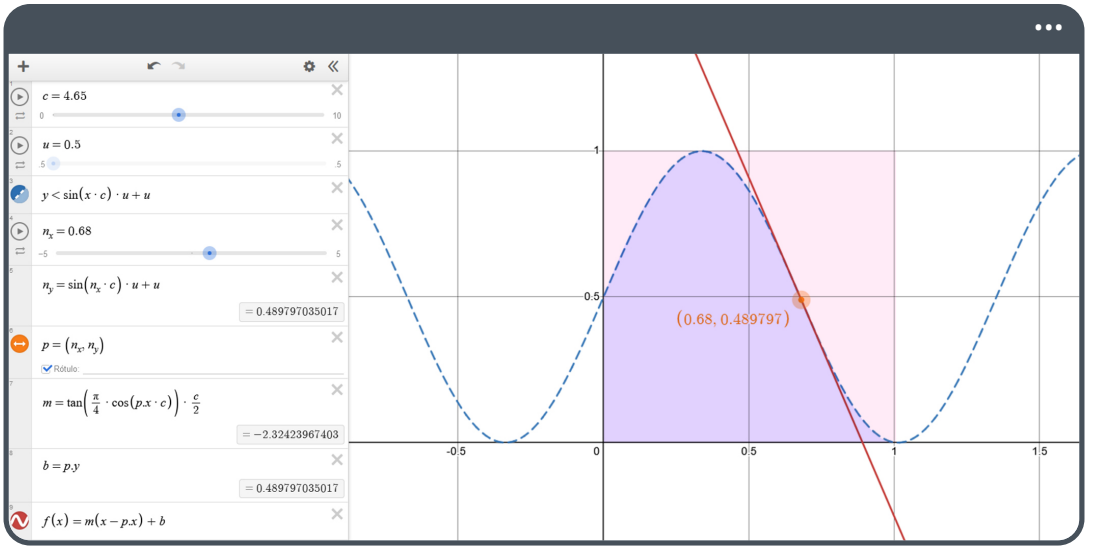
(2.1.u)

And set b, the term of intersection with the y-axis, equal to the coordinate p_y of the point:

$$b = p_y$$

(2.1.v)

Thus, it's possible to adjust a straight line to fit a sinusoidal curve, achieving a linear approximation in the vicinity of the point p.



(2.1.w https://www.desmos.com/calculator/f8hjcush5a)

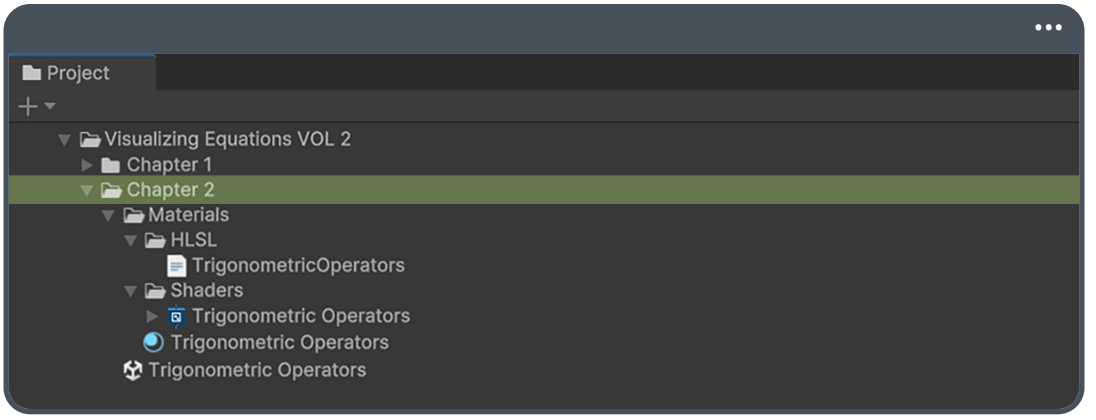
From Figure 2.1.w, considering the point defined as $p = (n_x, n_y)$, if we change the value of n_x, the point moving along the curve. Additionally, the tangent determines its orientation relative to its current position.

2.2 Visualizing Trigonometric Functions in HLSL.

Since trigonometric functions are essential in programming, it's common to find them implemented in most programming languages, and HLSL is no exception. Thus, understanding how they work as we work with them will therefore be a fundamental task.

To begin this section, you can go to your project and create a new **Unlit Shader Graph** shader type, calling it **Trigonometric Operators**. Following standard practice, you can generate a material and an HLSL file with the same name as the shader just created, ensuring that the shader is assigned to the material from the Inspector.

Maintaining the structure introduced in the first chapter, the project should be presented as follows.



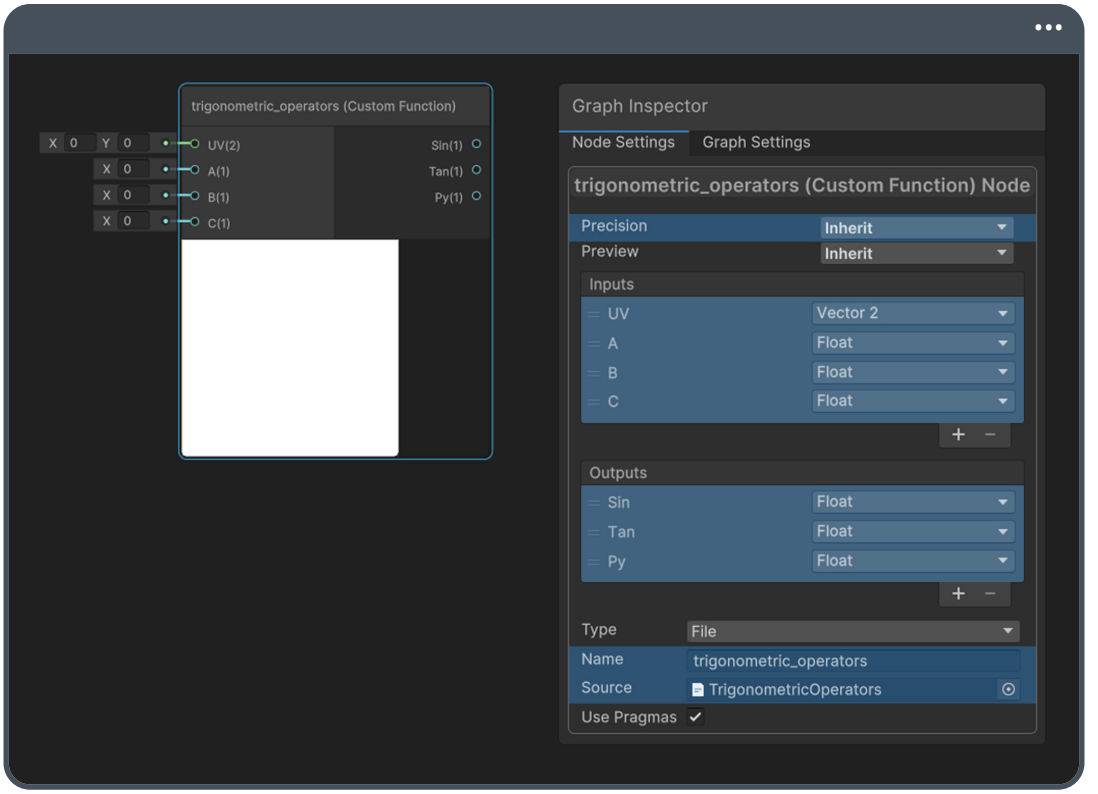
(2.2.a)

It is acknowledged that the nodes **Sine**, **Cosine**, and **Tangent** are already implemented in Shader Graph. However, opting for a **Custom Function** node is recommended to gain a deeper understanding of the process and logic behind these trigonometric calculations. Based on some of the operations discussed above, not only can the functions of the sine and tangent be visualized, but a method can also be integrated that allows for the adjustment of the position of the **Origin** circle along V, thus facilitating the visualization of the center point.

To begin, open your shader and create a **Custom Function** node. Using the function illustrated in Figure 2.1.i as a reference, the first step involves incorporating **a**, **b**, and **c** as input variables, and also including the **UV** coordinates as well.

In this scenario, the objective is to visualize three different results: the sine, the tangent, and the position of the point of origin. Therefore, as output, add three floating values and name them **Sin**, **Tan**, and **Py**.

You can also include the file **TrigonometricOperators.hlsl** as **Source**, and assign **trigonometric_operators** as its name. If the process has been executed successfully, your node should appear as described below:



(2.2.b)

Now, you need to declare the **`trigonometric_operators_float()`** method in your shader. To do this, include the previously mentioned variables along with the three output values: **`Sin`**, **`Tan`** and **`Py`**, as arguments, and initialize them to zero to ensure that your node compiles without errors.

```
void trigonometric_operators_float(in float2 uv, in float a, in float b, in
    float c, out float Sin, out float Tan, out float Py)
{
    Sin = 0.0;
    Tan = 0.0;
    Py = 0.0;
}
```

Before implementing the specific functions for each output, you must include the mathematical constants π and 2π. These constants will play a crucial role in the trigonometric operations to be developed below, ensuring the accuracy and efficiency of the calculations.

```
#define PI 3.14159265358
#define TWO_PI 6.28318530718

void trigonometric_operators_float(in float2 uv, in float a, in float b, in
    float c, out float Sin, out float Tan, out float Py) { … }
```

The constants **PI** and **TWO_PI** refer to half a rotation of a circle and a complete rotation, respectively. While these constants could be declared and initialized directly in each method that requires them, they are instead defined through macros to facilitate reuse. It's customary to declare and initialize **#define** directives in CG **.cginc** files, allowing them to be invoked only once. For instance, a **ShaderCommon.cginc** file may be created to include both **PI** and **TWO_PI**, which can then be incorporated into your shader using the **#include** command.

In relation to the total area of the Quad that you'll use for the graphic projection of both the sine and the tangent, as well as the central point, make sure to add dynamism to the generated figures. However, you'll center the action in the middle of the Quad. For this purpose, you can use the following mathematical formula:

$$f(x) = a\ sin((x + c)\ b)\ u + u$$

(2.2.c)

Looking at the previous reference, you'll notice that the variable u extends the function to confine the curve produced by the sin function within an interval of [0.0 : 1.0].

When implemented in HLSL, it appears as follows:

```
4  float sinusoidal(float x, float a, float b, float c)
5  {
6      const float u = 0.5;
7      return a * sin((x + c) * b) * u + u;
8  }
```

In this function, the parameter **x** acts as the input value on which the function is calculated. The parameter **c** adjusts the shift or phase of the wave, **a** acts as a coefficient that scales its amplitude, and **b** modifies its frequency.

Next, you can use the **`sinusoidal()`** method to model the shape of the wave through a new method, which you will name **`sine_wave()`**.

```
10 float sine_wave(float2 uv, float a, float b, float c)
11 {
12     float fx = uv.y;
13     float x = uv.x;
14
15     float f = sinusoidal(x, a, b, c);
16     fx -= f;
17
18     return (fx > 0) ? 0.0 : 1.0;
19 }
```

In the previous example, you can see that the variable **`fx`** (line 12) is initially assigned to the vertical component of the UV coordinates, i.e., **`uv.y`**. In contrast, the variable **x** (line 13) represents the horizontal component, or the **`uv.x`** coordinate. Line 15 illustrates how f is calculated using the **`sinusoidal()`** method, which includes parameters for amplitude, phase, and frequency. Next, you can adjust **`fx`** by subtracting the value of **f**, resulting in a vertical shift of the wave in accordance with the sinusoidal wave's value for a given

horizontal position. To complete the implementation, add a new method to visualize the orientation of the point on the wave.

```
21 float tan_wave(float2 uv, float a, float b, float c)
22 {
23     float fx = uv.y;
24     float x = uv.x;
25
26     const float px = 0.5;
27     float py = sinusoidal(px, a, b, c);
28     float m = a * tan(PI / 4.0 * cos((px + c) * b)) * (b / 2.0);
29
30     float f = m * (x - px) + py;
31     fx -= f;
32
33     return (fx > 0) ? 1.0 : 0.0;
34 }
```

In this process, you can reassign the **`uv.y`** component of the UV coordinates to **`fx`** (line 23), and the **`uv.x`** component to the variable **`x`** (line 24). Then, set the constant **`px`** to 0.5, representing the center of the wave's position on the x-axis (line 26). Subsequently, **`py`** (line 27) calculates the vertical position of the wave at the central point using the **`sinusoidal()`** method, while the variable **`m`** (line 28) defines the slope at **`px`**. Using the linear equation $mx + b$, compute **`f`**, the vertical displacement for the current **`x`** position, based on the slope **`m`** and the vertical reference **`py`**. The only thing left to do is assign these values to each output in the **`trigonometric_operators_float()`** method.

```
void trigonometric_operators_float(in float2 uv, in float a, in float b, in float c, out float Sin, out float Tan, out float Py)
{
    Sin = sine_wave(uv, a, b, c);
    Tan = tan_wave(uv, a, b, c);
    Py = sinusoidal(0.5, a, b, c);
}
```

Considering that the variable **c**, which defines the phase of the wave remains constant, no significant variations will occur in the final shape of the wave. For this reason, we will integrate a new method called **normalized_time** into the code. As the name suggests, this operation will normalize time by taking the fractional part of the division of **_Time.y** by the number of seconds **s**.

```
float normalized_time(float s)
{
    return frac(_Time.y / s);
}
```

To understand its functionality, it's important to look closely at the **_Time.y** variable, which returns the time elapsed since a level was loaded. This means that when a scene is started (Play button is pressed), time begins counting from zero forward. However, this situation presents a dilemma: although data types such as **fixed**, **half** or **float** offer high resolution, their capacity isn't unlimited. Therefore, you need to restrict them at some point to prevent graphical artifacts or processing errors. This is where the **frac()** function becomes useful, as it returns the fractional part of time, i.e., values ranging from 0.0 to 0.99. In this way, when the fractional value of **_Time.y** reaches 0.99, it resets to 0.0, creating a cycle and optimizing the process at the same time. The variable **s** divides the time, allowing the calculation to be performed in seconds. For example: if **s = 2**, the operation will complete its cycle in two seconds.

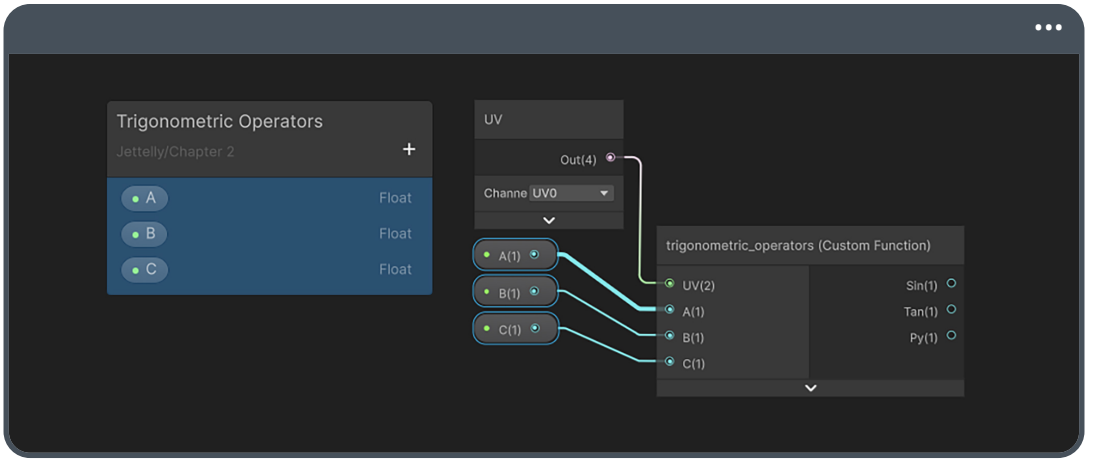
```
41 void trigonometric_operators_float(in float2 uv, in float a, in float b, in
   ↪ float c, out float Sin, out float Tan, out float Py)
42 {
43     float t = normalized_time(c) * (TWO_PI / b);
44
45     Sin = sine_wave(uv, a, b, t);
46     Tan = tan_wave(uv, a, b, t);
47     Py = sinusoidal(0.5, a, b, t);
48 }
```

In code line 43, a new variable, **t**, is declared and initialized, storing the result of the **normalized_time()** method using **c** as a parameter. You can use the operation **TWO_PI / b** to cover a complete cycle of the wave, adjusting for a change in the frequency **b**. After this, **t** is entered as the fourth value in each method (lines 45, 46, and 47), resulting in a unique number for each node output. In this way, the variable **c** determines the duration in seconds that the sinusoidal wave cycle takes to repeat.

Up to this point, your node should compile smoothly. In this instance, **float** has been chosen as the resolution for the node, eliminating the need to adjust the value of the **Precision** property in the **Graph Inspector**.

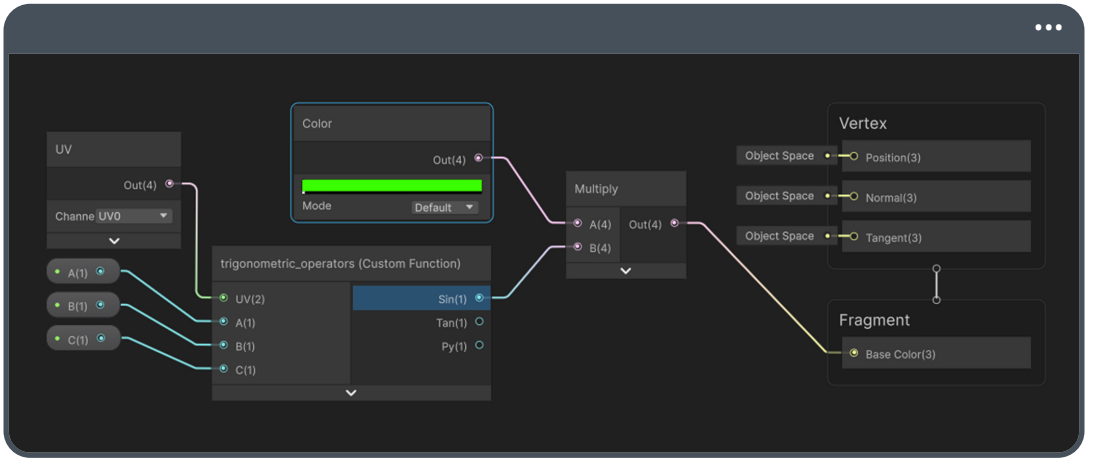
As usual, you should add the necessary properties to the **Blackboard** to allow the dynamic modification of the node's results from the Inspector, ensuring that their values are restricted to the following ranges:

- For the **A** property, the range will be [0.0 : 1.0], with 0.5 as the default.
- For **B**, it will be between [1 : 50], with 1.0 as the default.
- And for **C**, the range will be [1 : 5], with 1.0 as the default.



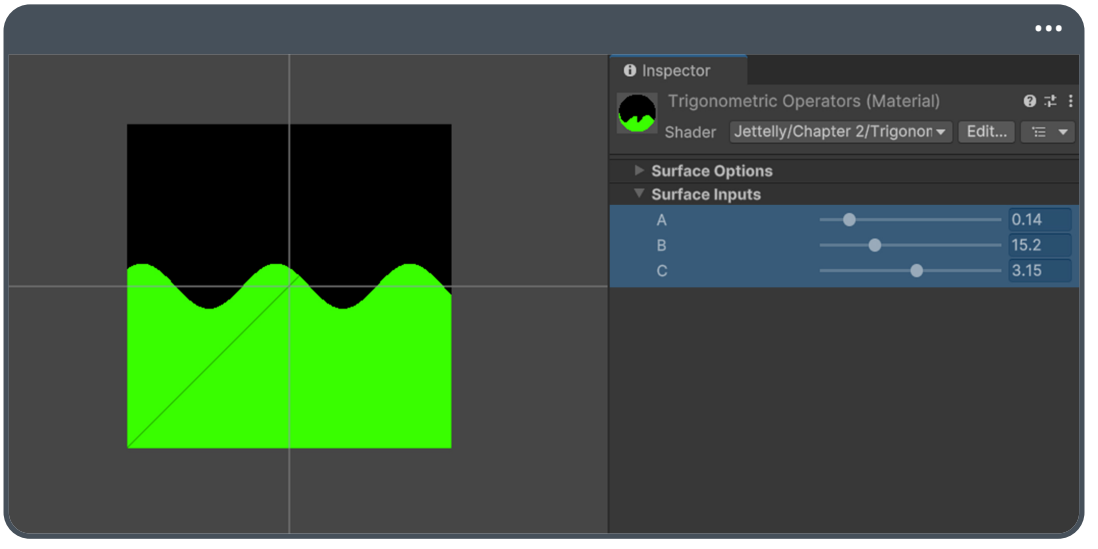
(2.2.d The properties have been linked)

If you closely examine the previous image, you'll notice that each property has been linked to the **trigonometric_operators** node, including the UV coordinates. From here, the sinusoidal wave will be projected onto the Quad in the scene using a specific hue for its visualization. To do this, you'll add a **Color** node into the shader, assigning it a default green color. Then, you'll multiply the value of **Sin** by the **Color** obtained, and use the result as **Base Color** in the shader.



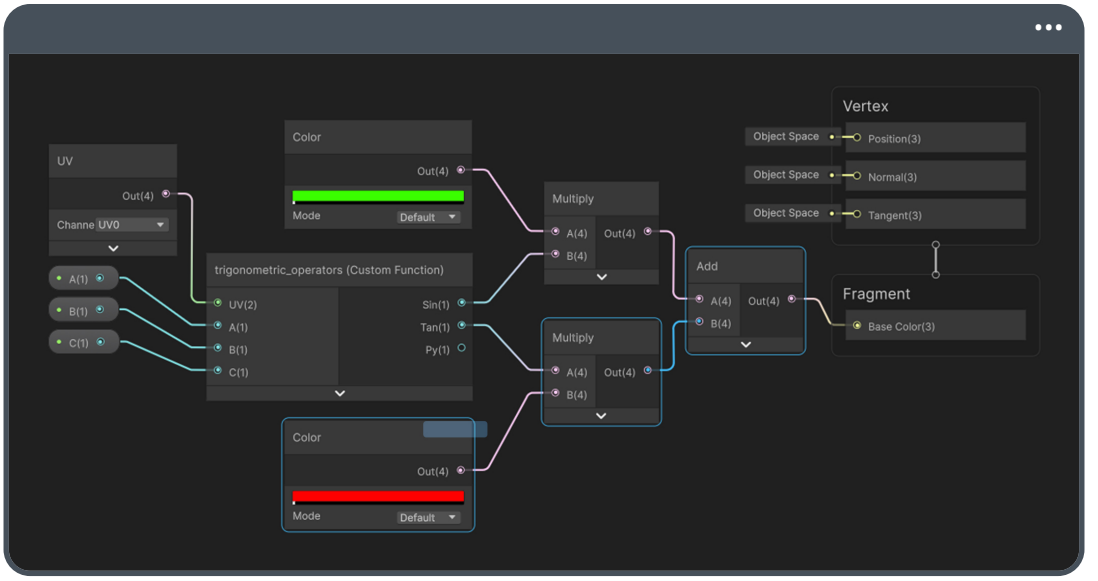
(2.2.e)

After saving these changes and returning to the scene, you can visualize and adjust the behavior of the sinusoidal wave directly through the properties of the **Trigonometric Operators** material, which was created at the beginning of this section.



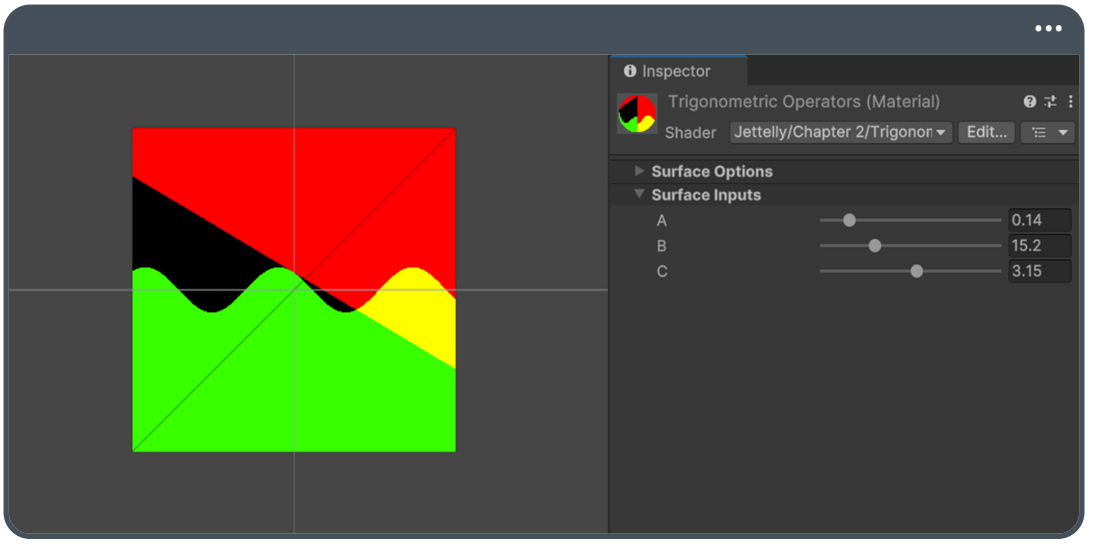
(2.2.f)

Next, you can integrate the slope into the visualization. To do this, return to the shader and add the value of **Tan** to the result previously obtained by multiplying **Sin** by the color green. However, before proceeding, you'll need to multiply the value of **Tan** by a red color. This action will allow you to clearly display both functions.



(2.2.g)

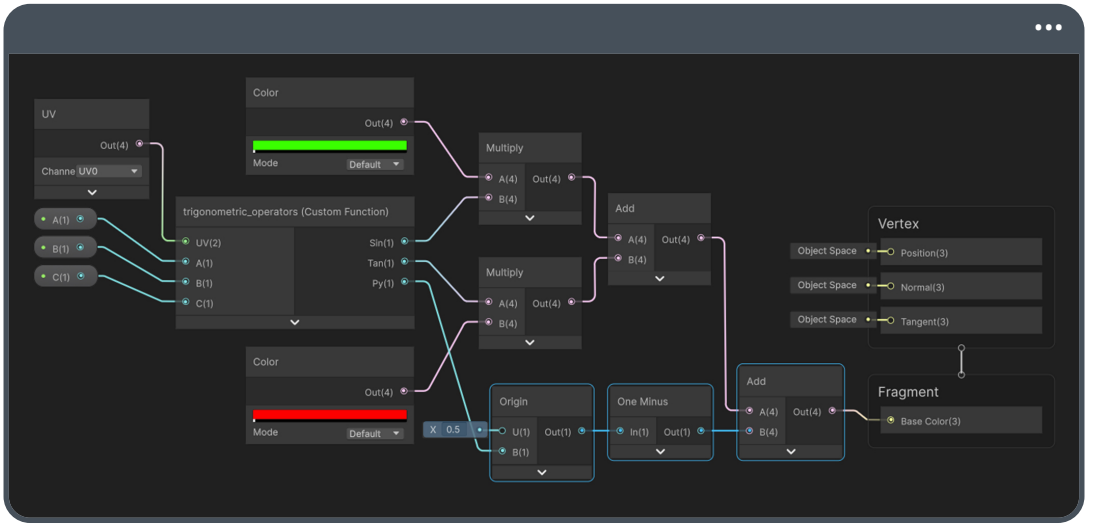
On saving the changes and returning to the scene once again, you can observe how both functions behave when operating together.



(2.2.h)

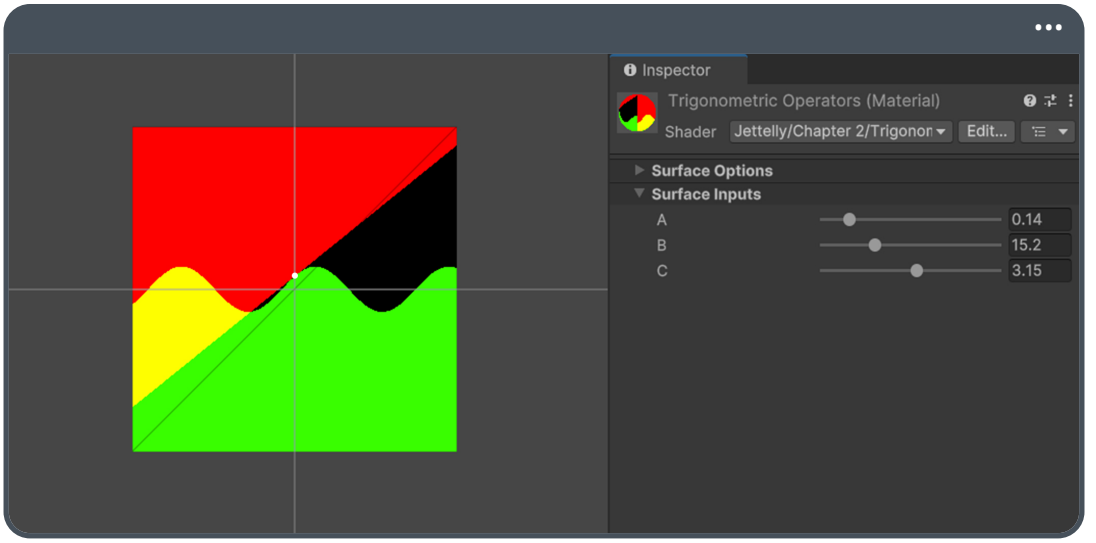
One notable aspect is the absence of the central point; although it is known that $x = 0.5$ and that y coordinate varies according to the amplitude of the wave, a graphical indicator

is needed to confirm the accuracy of your calculations. To address this issue, go back to the shader and integrate the **Origin** node into the operation. This node makes it easy to visualize a point, represented by a black circle. Since the background of the sinusoidal wave is also black, you can change the circle color to ensure it stands out on the graph. Finally, you can incorporate this point into the overall operation of the shader, thus improving the visualization.



(2.2.i)

If you look at the **u** property of the **Origin** node in figure 2.2.i, you can see that a value of 0.5 has been assigned to align it with the positions of both the sinusoidal wave and the slope. Additionally, the **One Minus** node has been used to invert the color of the node to white. This result has been included as part of the node operation. When returning to the scene, you can observe how the slope adjusts according to the position of the center point.



(2.2.j Wave projection)

2.3 One-point Reflection.

When we talk about the concept of reflection in the realm of computer graphics, the term essentially refers to the reversal of the position of a point with respect to a specific axis or plane, emulating the behavior of a mirror. This phenomenon is vitally important in creating visual effects, enriching the immersion and realism of the scene, and adding an additional dimension to the aesthetics of the virtual environment. Beyond simulating reflective surfaces, reflection plays a crucial role in advanced lighting techniques, such as Ray Tracing, where light reflects off objects in the environment, mimicking real-world light interactions.

To better understand the underlying principles of reflection, consider a point p denoted as $p = (3.0, 3.0)$. By reflecting this point with respect to the y-axis, a new point p' results in $p' = (-3.0, 3.0)$, indicating that the coordinate x is inverted while the y coordinate remains unchanged. Similarly, when reflecting p with respect to the x-axis, the result is $p = (3.0, -3.0)$, where the coordinate y changes its sign, keeping the x coordinate constant. These operations introduce the general formulas for reflection:

Reflection on the y-axis.

$$p' = (-x, y)$$

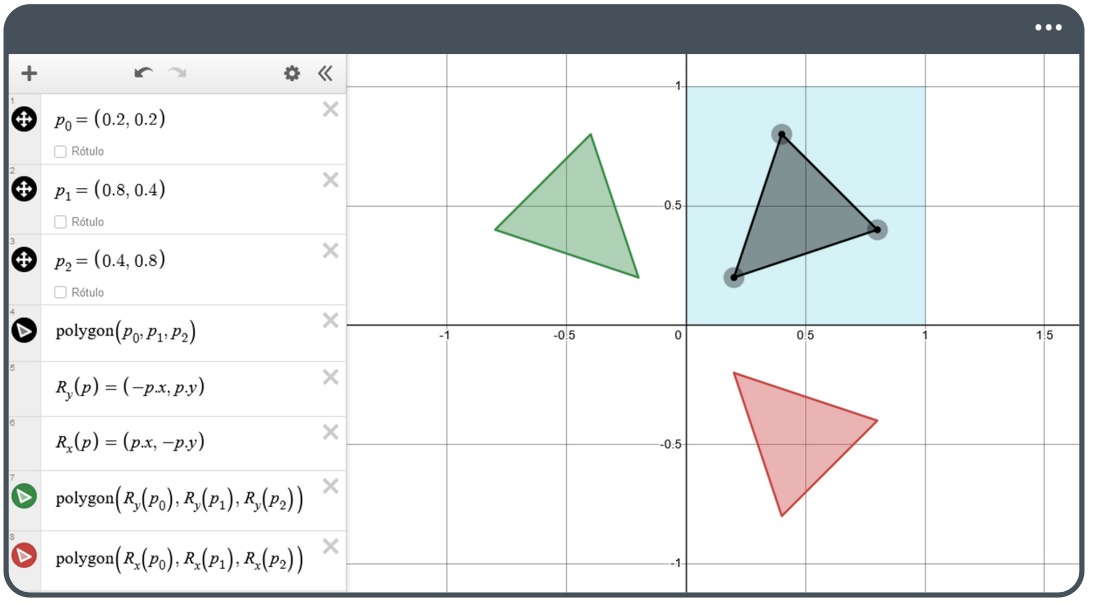
(2.3.a)

Reflection on the x-axis.

$$p' = (x, -y)$$

(2.3.b)

Applying these formulas, if we draw a polygon using the points $p_0 = (0.2, 0.2)$, $p_1 = (0.8, 0.4)$, and $p_2 = (0.4, 0.8)$, and then project it onto a two-dimensional Cartesian plane, the reflection of this polygon on each axis would reveal significant visual transformations.



(2.3.c https://www.desmos.com/calculator/9eiq5hkamn)

This process not only demonstrates the direct influence of reflections on the visual perception of objects and scenes but also highlights the importance of mathematically understanding these transformations for their practical application in computer graphics.

In Figure 2.3.c, the red polygon represents the reflection on the x-axis of the original polygon, which is colored black and formed by the points p_0, p_1, and p_2. Meanwhile, the green polygon illustrates the reflection of the same black polygon on the y-axis. To reflect a polygon in the third quadrant, a simple strategy is to make both components of a point p negative, resulting in:

$$p' = (-x, -y)$$

(2.3.d)

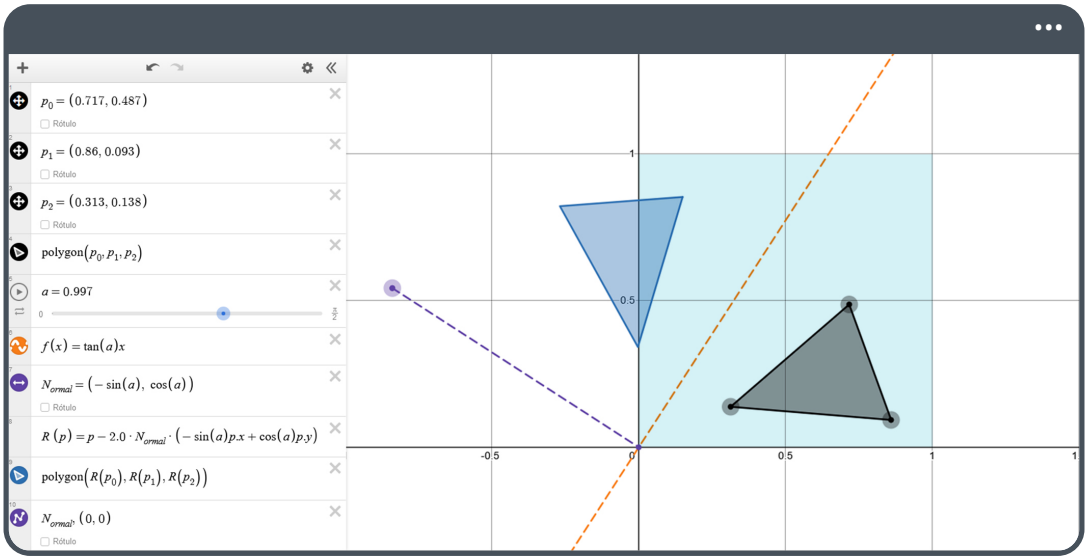
However, a more complex question arises when we want to apply a reflection on a line that doesn't correspond to the axes xy; specifically, what is the necessary formula to reflect a point on a line defined as $mx + b$? To address this, consider the following formula:

$$p' = p - 2(-sin(a), cos(a))(-sin(a)p_x + cos(a)p_y)$$

(2.3.e)

From this equation, there are several concepts that we already understand. For example, we know the sine and cosine operators derive from relationships in a right triangle and are linked to the properties of a complete circle. Additionally, we know that both functions are 90° out of phase. So, how should this formula be correctly interpreted? The term p denotes the original point, while p' indicates the position of the point after reflection, which is determined by the angle a. Therefore, a specifies the orientation of the line of reflection with respect to the coordinate axes.

To see this more intuitively, we can consider that the reflection line acts as an inclined mirror, where a represents the angle of inclination with respect to the x-axis. The reflection of a point on this line involves not only an inversion in its coordinates but also a rotation that depends directly on the angle of inclination of the line.



(2.3.f https://www.desmos.com/calculator/aouzncgbyj)

The line of reflection can be represented in various ways. However, its role in the operation mentioned above is crucial, as reflection is carried out through its normal vector n, defined as,

$$n = (-sin(a), cos(a))$$

(2.3.g)

As we may anticipate, this vector is perpendicular to the line of reflection.

Reflecting a point across a line involves finding a point p' such that the line acts as the perpendicular bisector of the segment joining p and p'. The function $-sin(a)p_x + cos(a)p_y$ calculates a scalar projection (Dot product) that determines how

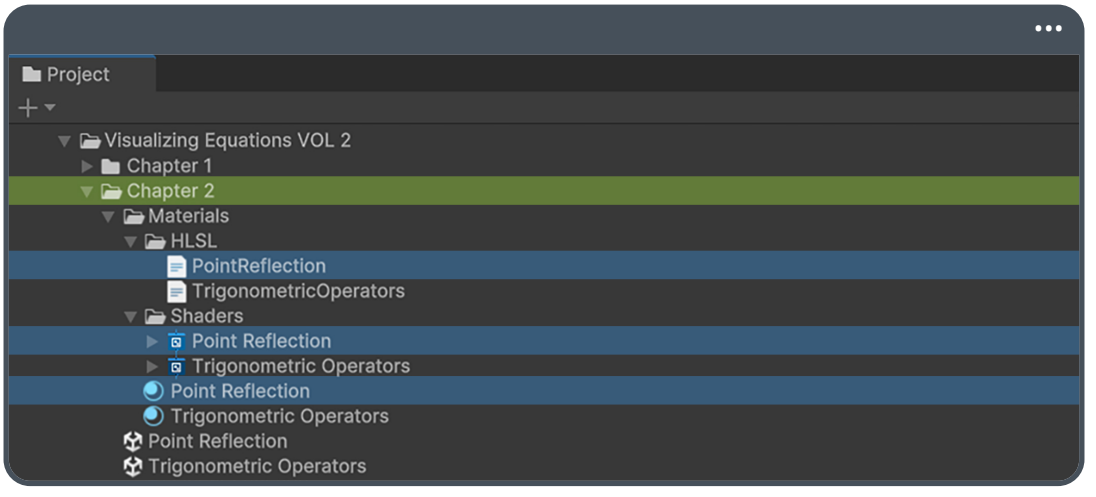
far away the original point p is in the direction of the normal vector n. Multiplying this scalar by 2 and then by the normal vector itself yields the displacement vector needed to transit from p to p' across the line of reflection.

Explained in simple terms, this equation moves the point p in a direction perpendicular to the line of reflection, defined by the angle a, exactly twice the shortest distance between p and the line. Thus, it ensures that p' is located on the opposite side of the line, and at the same distance from it.

2.4 One-Point Reflection in HLSL.

After having explored the graphical representation of various functions and becoming familiar with the structure of your project, we'll now focus on implementing one-point reflection in HLSL. To illustrate this concept, you'll add an **Unlit Shader Graph** and an **.hlsl** extension script, both specifically designed for this purpose. As with your previous work with trigonometric operators, you must name each object **Point Reflection**.

Once the three aforementioned objects have been incorporated, the project will appear as follows:



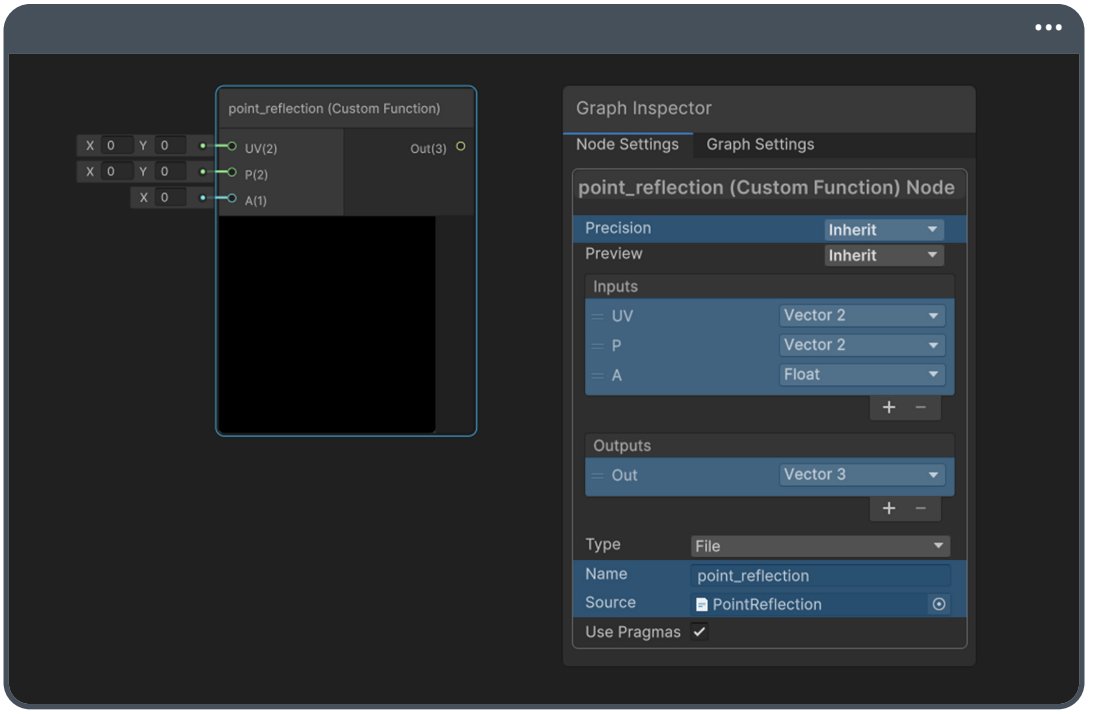
(2.4.a Project structure)

To start the process, first assign the shader to the material and then this material to a Quad in your scene. This allows you to visualize the adjustments made in the shader configuration.

Once the shader is open, you can include a **Custom Function** node and enter the necessary operations to visualize the reflection of a point. Based on the function illustrated in Figure 2.3.e in the previous section, the first step involves incorporating the UV coordinates, the point p to be reflected, and the angle a that determines the orientation of the reflection line.

Since both the coordinates and the point are two-dimensional vectors with components x and y, and the angle is a scalar value, a three-dimensional vector is required for the **Out** output. This is because each visual result is associated with a unique color, facilitating an easier understanding of the concept.

You can call the function **point_reflection** in the **Name** property, and attach the script **PointReflection** for the **Source**.



(2.4.b point_reflection node)

Continue by defining the **`point_reflection_float()`** method in your HLSL script, adding the variables **uv**, **p**, and **a** as inputs, and the output value **Out**, as seen below:

```
void point_reflection_float(in float2 uv, in float2 p, in float a, out
    float3 Out)
{
    Out = float3(0, 0, 0);
}
```

To visualize the reflection effectively, it's essential to consider two points: p (the original point) and p' (the reflected point), as well as the reflection angle itself. Before we delve into the development of the **`point_reflection_float()`** method, it's vital to include the functions that allow each of these elements to be clearly represented.

Begin with the implementation of a linear function in the form $y = mx + b$, where $m = tan(a)$ to facilitate a controlled rotation. Since the value of b is zero in this context, you can omit it from the function, thus simplifying the equation.

```
#define PI 3.14159265358

float linear_function(float2 uv, float a)
{
    float fx = uv.y;
    float x = uv.x;

    float m = tan(a);
    float f = m * x;      // -- mx + 0 --
    fx -= f;

    return (fx > 0.0) ? 1.0 : 0.0;
}

void point_reflection_float(in float2 uv, in float2 p, in float a, out
 ↪  float3 Out) { … }
```

From the code snippet, the first line starts by defining the constant **PI** in the first line, which represents half the circumference of a circle, i.e., 180°. This constant is essential for adjusting the x sign in situations where the line slope exceeds 90°. Moving forward in the code, specifically at line 8, the slope **m** of the line is calculated as the tangent of the angle **a**. In this context, **a**, expressed in radians, indicates the inclination of the line with respect to the positive x-axis and is measured counterclockwise. Therefore, if the angle **a** is less than 90°, then **tan(a)** results in a positive value. However, for angles between 90° and 180°, **tan(a)** assumes a negative value. This change is visually reflected as a transition in color from white to black.

To deepen your understanding, proceed to implement the **point_reflection_float()** method, incorporating the **linear_function()** into its definition. Additionally, you can

introduce a new function in the script to simplify the calculation of the angle in degrees instead of radians, thus improving your comprehension and handling of the concept.

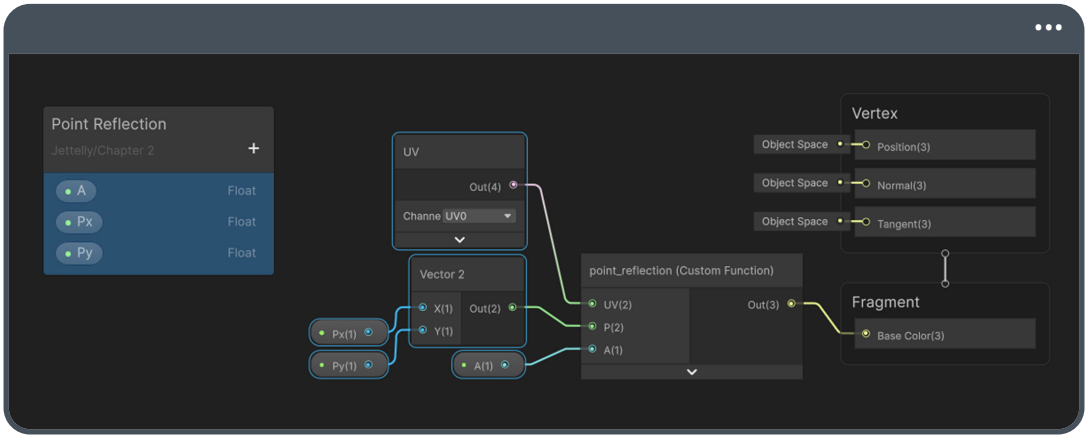
```
17 float to_degree(float a)
18 {
19     return a * (PI / 180.0);
20 }
21
22 void point_reflection_float(in float2 uv, in float2 p, in float a, out
    ↪ float3 Out)
23 {
24     uv -= 0.5;
25     a = to_degree(a);
26
27     const float3 color_g = float3(0, 1, 0);
28     float3 l = linear_function(uv, a) * color_g;
29
30     Out = l;
31 }
```

Line 17 introduces a method called **`to_degree()`**, designed to convert an angle from radians to degrees, using the formula $d = r\frac{\pi}{180}$. Then, in the **`point_reflection_float()`** method, specifically in line 24, the **`uv`** coordinates are adjusted by subtracting 0.5 from their components **`x`** and **`y`**. This modification is critical for centering the origin point in the middle of the Quad in your scene. In line 27, **`color_g`** is defined as a three-dimensional vector representing the color green in the RGB system. This color is then used to influence the result of the **`linear_function()`**, and the product is stored in **`l`**, a three-dimensional vector.

It's important to ensure that the properties are correctly linked to your node in the **Master Stack** to observe the rotation of the linear function on the Quad. It's worth mentioning that for point **p**, two floating values have been chosen and grouped into a two-dimensional vector due to the limitation of not being able to assign ranges directly to the components of

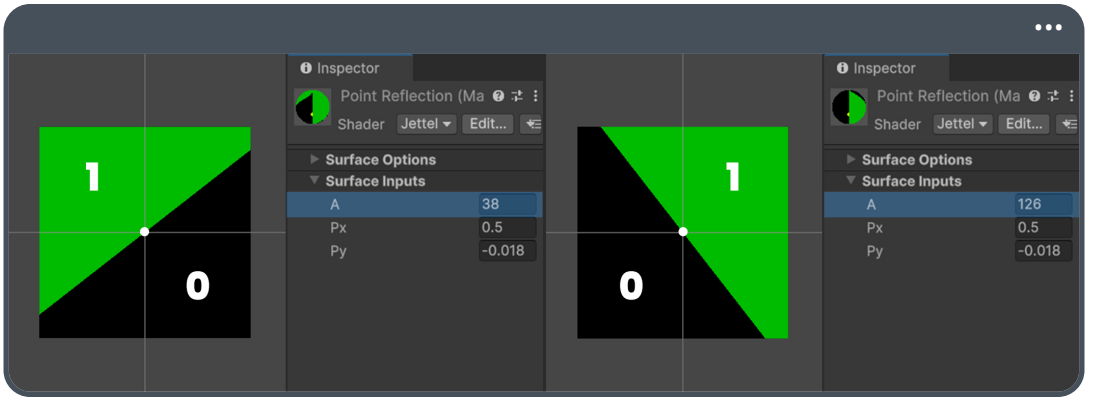
a vector. In contrast, a float value can be defined with a minimum, maximum, and default value. Therefore, you can proceed to the **Blackboard** to add the following properties.

- Add the **A** property and set it as a range, with a minimum of 0 and a maximum of 180.
- Then, include the first component of the point **p**, i.e., **Px**, also as a range. Therefore, its minimum will be -0.5 and its maximum 0.5.
- Repeat the previous procedure for the **Py** component, setting its minimum and maximum limits equivalent to those of **Px**.



(2.4.c)

As illustrated in Figure 2.4.c, the relevant properties have been linked to the **point_reflection** node. By saving the changes and returning to your scene, you can observe how the linear function behaves, specifically noting that it inverts its sign when the angle **A** exceeds 90°.



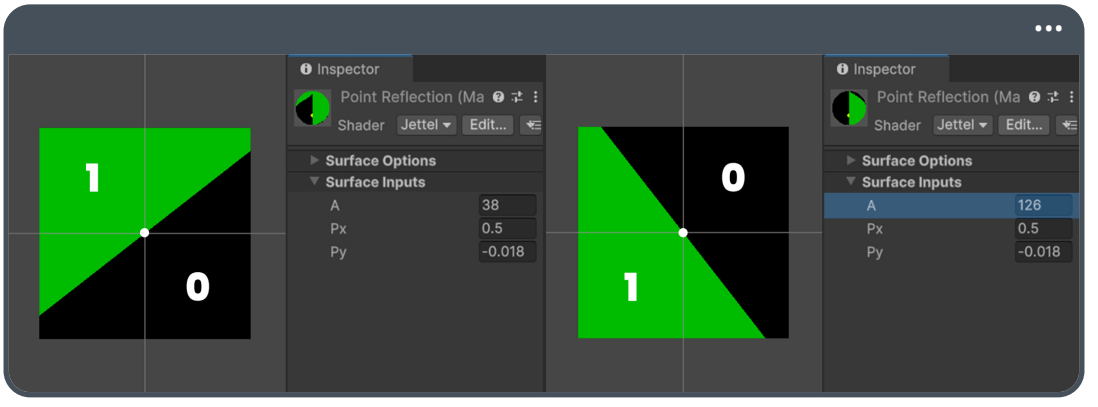
(2.4.d)

In Figure 2.4.d, the image on the left shows a positive slope, colored green, for an angle **A** of 38°, while the image on the right shows a negative slope for an angle **A** of 126°. This behavior can present a graphical comprehension challenge, especially when attempting to reflect a point, as it can be confusing to determine the actual position of the mirrored point. To address this problem, one solution is to adjust the linear function to take negative values when the angle is greater than 90°. To implement this fix, go back to your script and add the following piece of code.

```
3  float linear_function(float2 uv, float a)
4  {
5      float fx = uv.y;
6      float x = uv.x;
7
8      float m = tan(a);
9      float f = m * x;
10     fx -= f;
11
12     float h = (a > PI / 2) ? 1 : -1;
13     return (h * fx > 0.0) ? 1.0 : 0.0;
14 }
```

Looking at line 12, you can see that a new variable **h** has been introduced, which assumes a value of 1 or -1 depending on whether the angle **a** exceeds **PI / 2** in radians (equivalent

to 90°). This variable **h** is then multiplied by **fx**, ensuring that the slope displayed always remains positive. Upon saving the changes and returning to the scene, you can observe that the slope retains its coherence without reversing its sign.



(2.4.e)

Now, you can focus on graphically visualizing the point p and, subsequently, its reflection p'. To do this, you can use the algebraic circle previously defined in the **AlgebraicCircle**. To incorporate it into your project, use the **#include** directive, which instructs the compiler to insert the contents of the file into the source code at the exact location where the directive is placed.

Next, you have to specify the file path in quotation marks. Following the organizational structure proposed at the beginning of this book, the path to the **AlgebraicCircle** file would be established as:

- Assets > Jettelly Books > Visualizing Equations VOL 2 > Chapter 1 > Materials > HLSL > AlgebraicCircle.hlsl.

```
#define PI 3.14159265358

#include "Assets/Jettelly Books/ … /AlgebraicCircle.hlsl"

float linear_function(float2 uv, float a) { … }
```

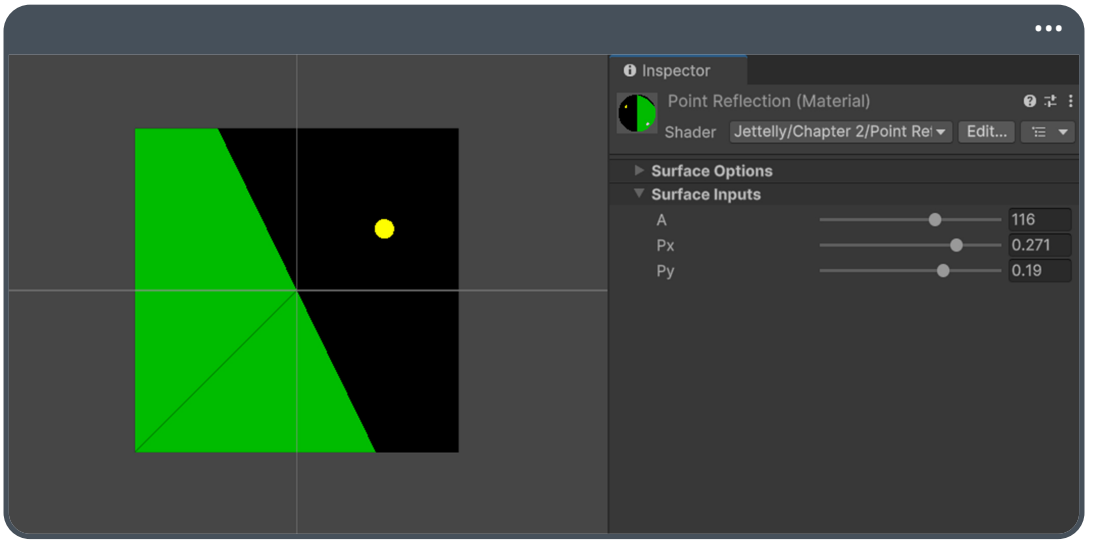
In the context of the **`point_reflection_float()`** method, given the void nature of the **`algebraic_circle_half()`**, first it'll be necessary to declare and initialize the variables that will function as output values in the method. You can observe the implementation starting from line 29 of the following code:

```
23 void point_reflection_float(in float2 uv, in float2 p, in float a, out
    ↪ float3 Out)
24 {
25     uv -= 0.5;
26     a = to_degree(a);
27
28     const float3 color_g = float3(0, 1, 0);
29     const float3 color_y = float3(1, 1, 0);
30
31     float3 l = linear_function(uv, a) * color_g;
32
33     float p0 = 0;
34     algebraic_circle_half(uv, p.x, p.y, p0);
35     float3 c = (1 - float3(p0.xxx)) * color_y;
36
37     float3 render = l + c;
38     Out = render;
39 }
```

Line 33 has introduced the variable **`p0`** as intended to receive the result of **`algebraic_circle_half()`** (line 34). Next, **`c`**, a three-dimensional vector, is calculated by inverting the value of **`p0`** using the operation **`1 - float3(p0.xxx)`**, where **`p0.xxx`**

implies replicating the value of **p0** in each component of the vector. This inversion, when combined with **color_y**, results in the display of a yellow dot.

By applying these changes to the script and returning to the scene, you can see the interaction between the linear function and the point p, thus demonstrating the effectiveness of the implementation of both the line and the original point in the visualization.



(2.4.f)

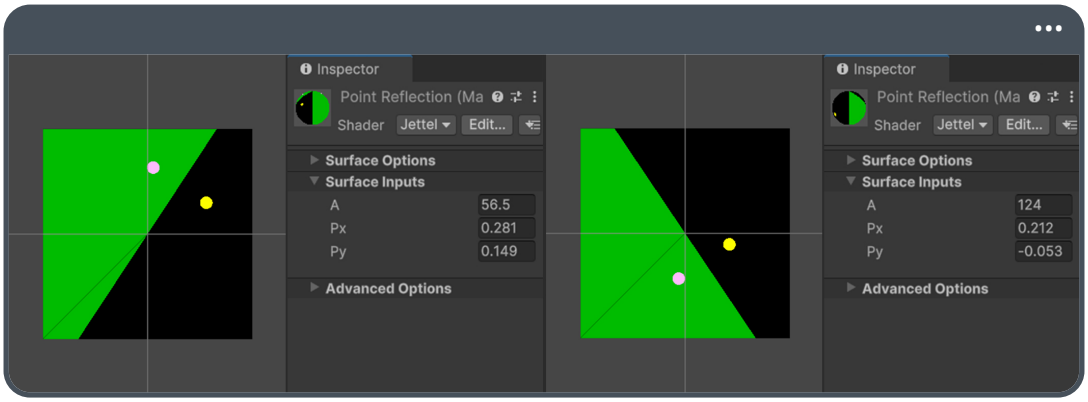
For the exercise illustrated in Figure 2.4.f, the value of the **RADIUS** property was set to 0.03 in the **AlgebraicCircle** script to obtain a graphical representation of a larger circle. The last step to complete this process is to add the reflection point p', which is derived from the original point p, and the normal n of the angle a.

```
23 void point_reflection_float(in float2 uv, in float2 p, in float a, out
   ↪ float3 Out)
24 {
25     uv -= 0.5;
26     a = to_degree(a);
27
28     const float3 color_g = float3(0, 1, 0);
29     const float3 color_y = float3(1, 1, 0);
30     const float3 color_m = float3(1, 0, 1);
31
32     float3 l = linear_function(uv, a) * color_g;
33
34     float2 n = float2(-sin(a), cos(a));
35     float2 pr = p - 2 * n * (-sin(a) * p.x + cos(a) * p.y);
36
37     float p0 = 0;
38     float p1 = 0;
39
40     algebraic_circle_half(uv, p.x, p.y, p0);
41     algebraic_circle_half(uv, pr.x, pr.y, p1);
42
43     float3 c = (1 - float3(p0.xxx)) * color_y;
44     float3 cr = (1 - float3(p1.xxx)) * color_m;
45
46     float3 render = l + c + cr;
47     Out = render;
48 }
```

Line 34 introduces **n** as the normal, based on the equation in Figure 2.3.g. The reflection point **pr** is calculated on line 35, applying the equation shown in Figure 2.3.e. Subsequently, line 41 implements a second algebraic circle, which essentially represents the point p'. For this, **pr** is used as an argument in the function, and its result is stored in the variable $p1$ (line 38).

Line 44 defines **cr**, a three-dimensional vector that adopts the value of **p1** in each of its components. Then, **cr** is multiplied by the magenta color **color_m**, set on line 30, to differentiate it from the previous point. By saving the changes and returning to the scene,

you'll be able to visualize both the original point p and its reflection p' on the Quad in your scene.



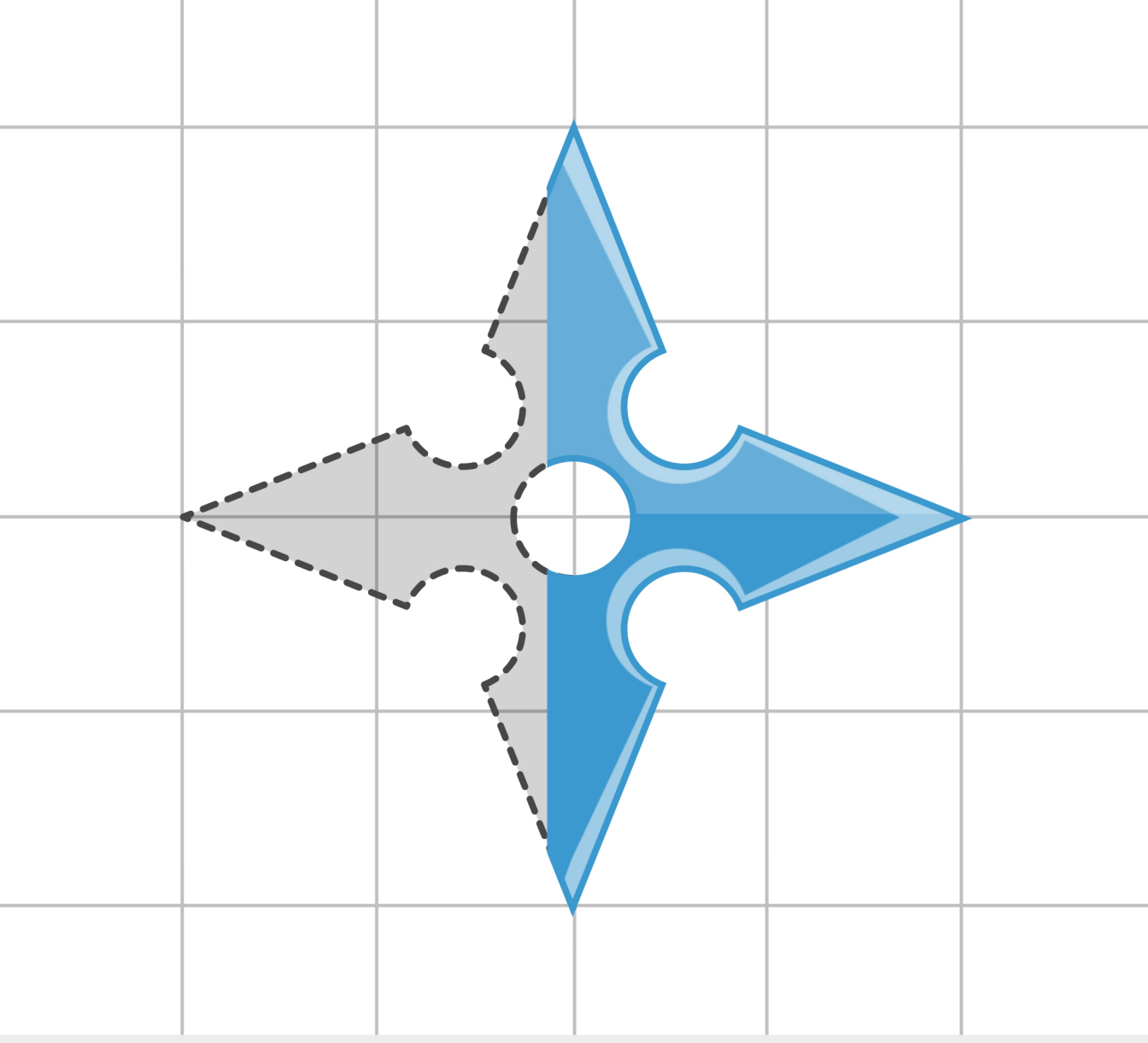
(2.4.g)

Chapter Summary.

- Throughout this chapter, we've explored trigonometric functions and their application in developing visualizations with HLSL in Unity. You began with an introduction to the fundamental concepts of trigonometric functions, establishing their relationship with the right triangle. Then we covered the basic definitions of sine, cosine, and tangent, along with their geometric interpretation.

- In the following section, we developed a shader to visualize trigonometric functions. Then we constructed a sine wave and implemented the tangent function to represent the orientation of a point traveling along the wave in a two-dimensional space.

- Next, we explained mathematical operations involved in reflecting a point across a line defined by a linear function, detailing the calculations step by step to ensure a clear understanding of the reflection process on a Cartesian plane. Finally, we developed a shader to visualize the real-time reflection of a point on a Quad in the scene.



Chapter 3

Procedural Shapes.

In this chapter we introduce the creation of two-dimensional procedural shapes in HLSL using previously explored mathematical functions. We'll begin with a detailed analysis of these shapes in the Cartesian plane, identifying and addressing the challenges associated with coordinate manipulation. Throughout the process, we'll implement specific strategies, such as filtering to refine resolutions, smoothing edges, and improving visual quality. This hands-on approach will demonstrate how to apply fundamental mathematical concepts in the development of visual effects in Unity, opening up a range of creative and technical opportunities for the generation of graphic content.

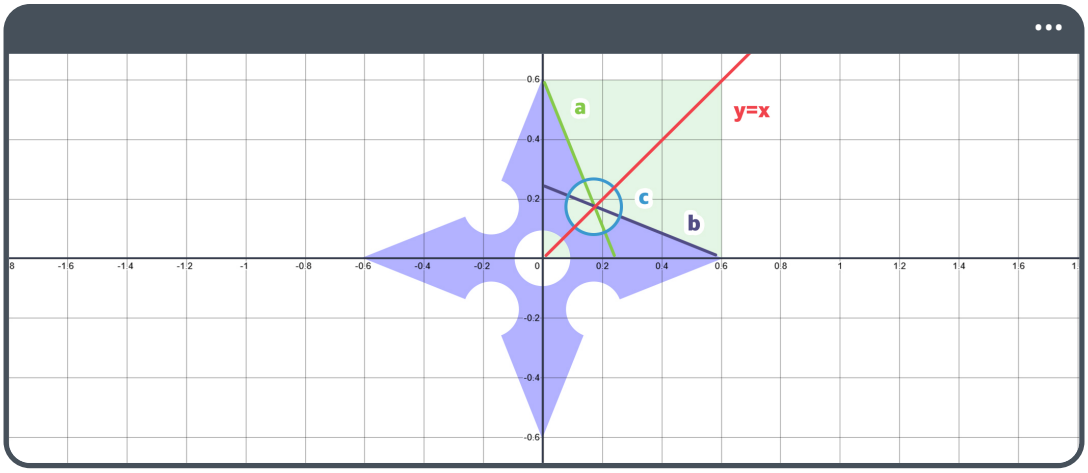
3.1 Analyzing the Shape of a Shuriken.

When discussing procedural images, it's important to understand the requirements of the shape you want to create. Based on these, specific functions are adopted to draw curves or lines, which together form graphical areas on the screen. Before beginning this process, it's helpful to consider a few key questions that can speed up the creation and development process:

For any procedural shape:

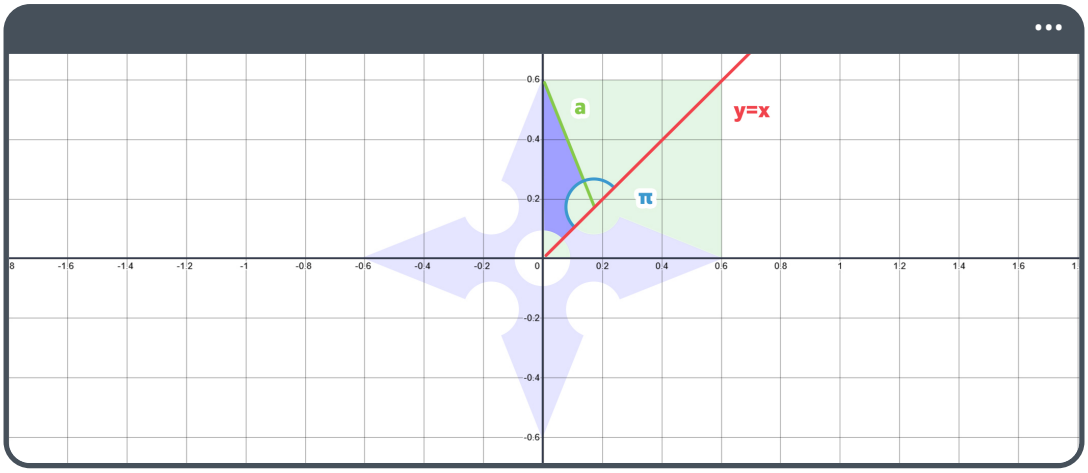
- Can it be recreated using polygons?
- Are its sides the same?
- Does it have similar angles?

Although there are other relevant questions, these are the most important because their analysis demonstrates how to optimize not only the shape itself, but also to develop simplified mathematical functions, thereby minimizing calculations on the GPU. For example, consider the shape of a Shuriken and project its coordinates onto a Cartesian plane.



(3.1.a Shuriken visualization on Desmos)

As shown in Figure 3.1.a, the Shuriken, centered on the origin, has a symmetrical shape. Therefore, it can be optimized by using only three points a, b, and c and a semicircle to recreate its complete shape. In fact, the point b could be equal to the reflection of a, i.e., $Ref_l(b) = a$, since both lines connect with c. This can be seen in the following reference:



(3.1.b Repetition pattern)

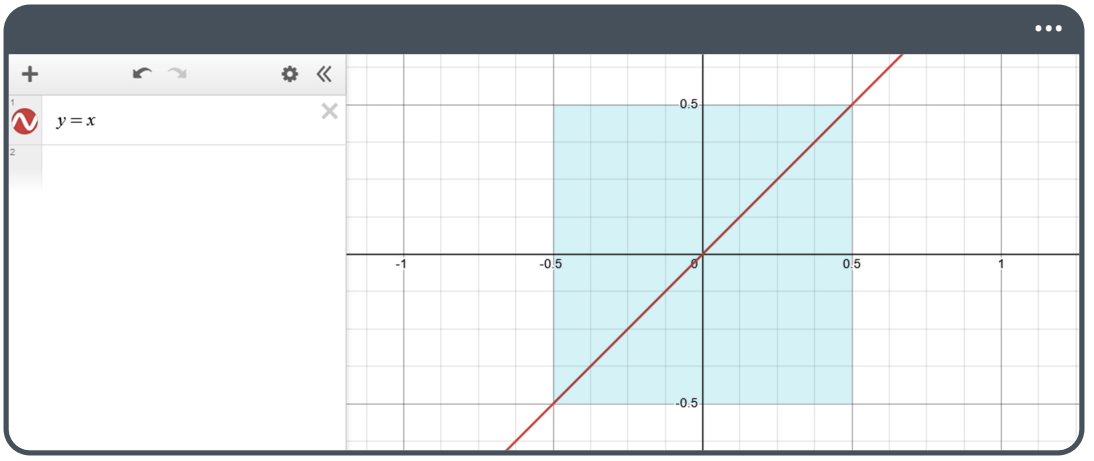
It's true that we aren't considering the circle located at the central point, but its analogy remains the same. If you want to draw the first point of a Shuriken on a Cartesian plane, you should consider exclusively:

- A triangle.
- A semicircle, for cutting.
- An eighth of a circle, for the grip of the center.

Next, you can apply absolute values and other functions to extend the calculations to the other quadrants, as well as limit the length of the infinite lines produced by these functions. However, the entire operation would be centered on the initial symmetry of the first quadrant.

To understand the process, you'll apply the concepts mentioned up to this point and draw the shape of a four-pointed Shuriken on the plane using the Desmos interface. Before beginning, you need to define certain values to make the procedural image an object that can be drawn in UV coordinates later. For example, the maximum diameter of the shape will be equal to 1.0. Therefore, the points and their positions will be kept within an area between [-0.5 : 0.5], considering the origin as the center point.

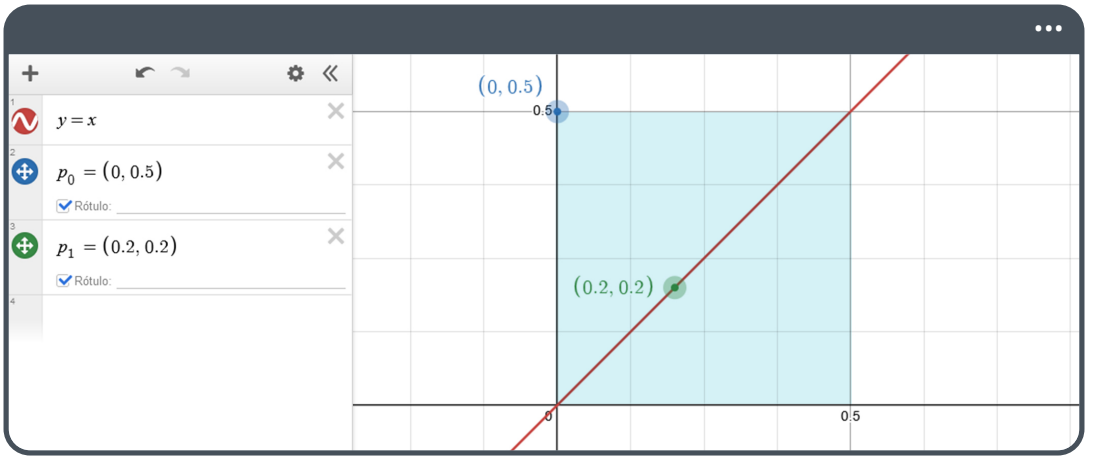
To visualize this, start by generating a line of the type $y = x$ as follows:



(3.1.c https://www.desmos.com/calculator/3oeo5tpox0)

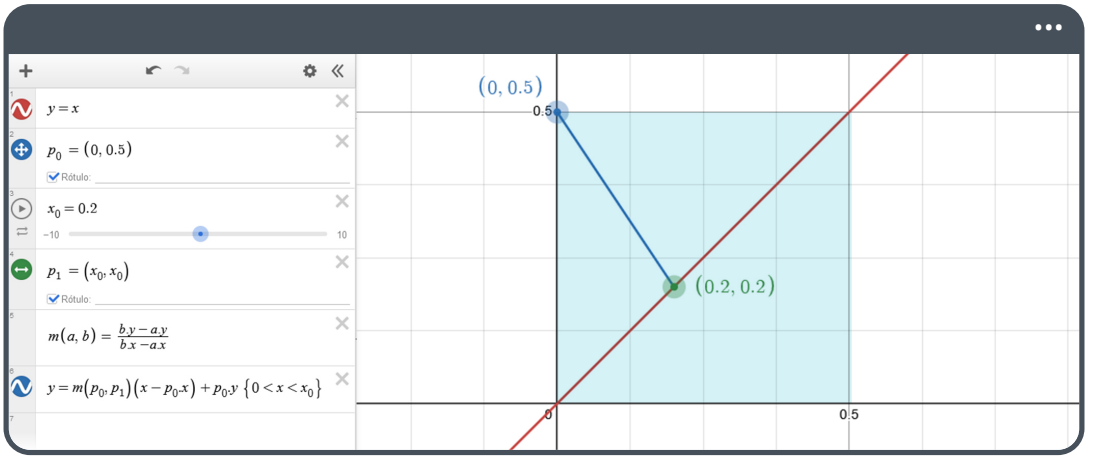
Considering that you can modify the shape of the Shuriken dynamically from the Unity Inspector, this line will allow you to visualize its opening, generating variations according

to the position of a point to be defined. Next, you can define an additional point to draw a line that determines the tip of the line.



(3.1.d https://https://www.desmos.com/calculator/7dp22ivjfk)

If you pay attention to the previous reference, you'll notice the points p_0 and p_1 have been defined and are located at the positions [0.0, 0.5] and [0.2, 0.2], respectively. These points exist only at the data level. To generate graphs with them, it will be essential to draw a line that joins them, i.e., $\overline{p_0\ p_1}$. To do this, you can use the equation of the line mentioned in section 1.4 of Chapter 1, which is defined as $y = (x - a.x) + a.y$, with the slope $m = \frac{b.y - a.y}{b.x - a.x}$. This process is illustrated in the following example:



(3.1.e https://www.desmos.com/calculator/78ubysryfq)

There are two important aspects in Figure 3.1.e. First, the x_0 variable. Since this has been defined for each component at point p_0, if its value changes, it'll move along the line $y = x$. As you've previously limited the diameter of the Shuriken, x_0 has also been restricted to a value between [0.0 : 0.5]. This process is critical, as otherwise, the shape of the figure will be affected when the values are outside the mentioned range.

Second, the limits of the line. The line generated between point p_0 and p_1 is infinite. Consequently, the operation $\{0 < x < x_0\}$ has been used to limit the line to the required area.

Your next step will be to add the point p_2, which connects to p_1 in the Cartesian plane. Since $Ref_l(p_2) = p_0$, you can apply Equation 2.3.e from Chapter 2. However, considering that $y = x$ has no change in its angle, you can simplify the reflection formula as follows:

$$p' = p - 2n(p \cdot n)$$

(3.1.f)

Where,

$$n = (-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$$

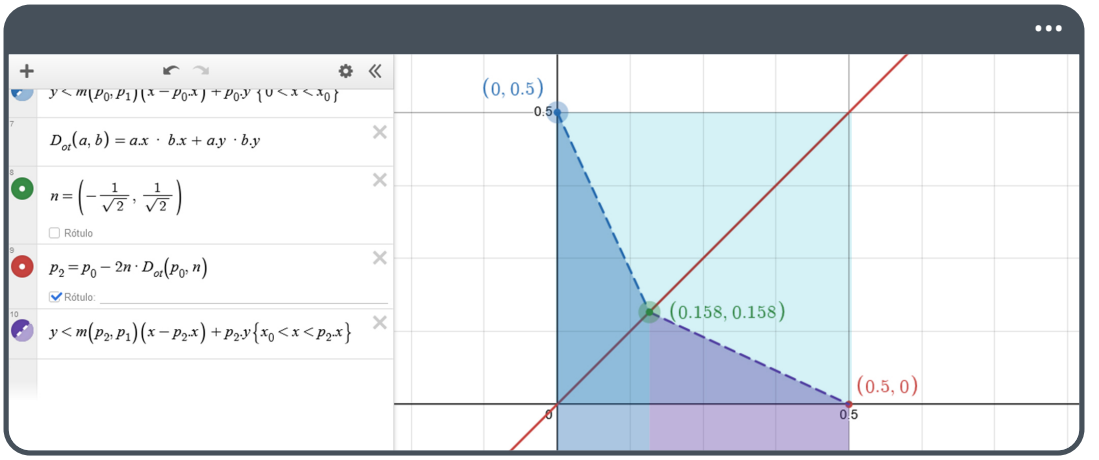
(3.1.g)

From Equation 3.1.g, you can deduce that the variable n refers to the normal of the line, and $p \cdot n$ corresponds to the dot product, i.e.,

$$p \cdot n = p.x * n.x + p.y * n.y$$

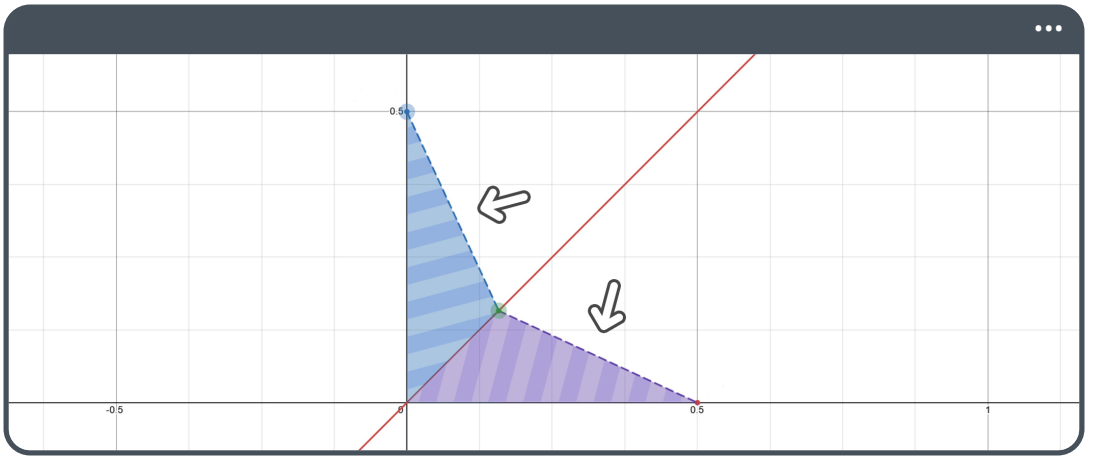
(3.1.h)

If a line is drawn from point p_2 to point p_1 using the functions mentioned above, you'll get the following result:



(3.1.i https://www.desmos.com/calculator/5vteybnzzm)

A factor to consider when working with line segments is the region of color and direction it forms. Paying attention to Figure 3.1.i, you can see that two regions of color have been generated: one light blue and the other purple, both pointing in a vertical direction. You could assume that each region should point in a direction according to the angle it has, as shown in the following reference:



(3.1.j Point reflection)

However, this isn't the case, as the regions presented in Figure 3.1.i indicate that all points within their respective areas meet the condition defined for each case, i.e.,

$$y < m(x - a.x) + a.y$$

(3.1.k)

To obtain a more optimal result, you need to use a different technique known as **Signed Distance Function (SDF)**, which allows the distance of a point to be determined from a line. However, you'll continue to experiment with the linear and trigonometric functions used up to this point. Later, this book devotes two chapters to implicit distance functions in both two and three dimensions.

As you can see in the previous reference, the tip of the procedural Shuriken has already been defined. All that remains is for you to apply the functions in the other quadrants to generate their complete form. To do this, you can use the absolute value over the xy coordinates in the functions that define the color regions, that is:

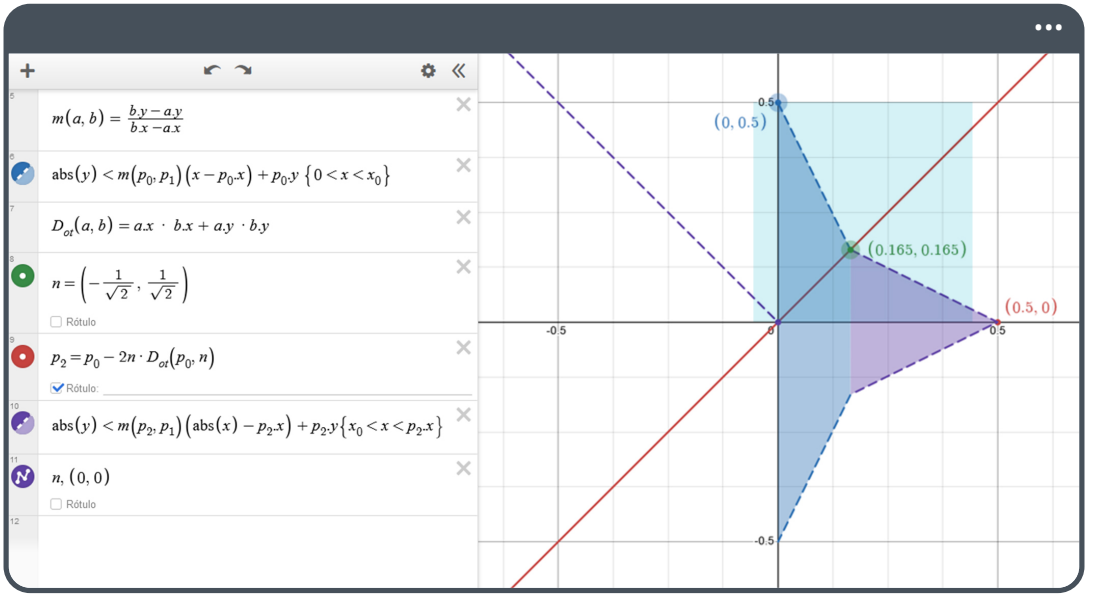
$$|y| < m(|x| - a.x) + a.y$$

(3.1.l)

The absolute value corresponds to the distance of any number from zero on the number line, regardless of direction. This factor always makes the number positive. For example, consider a negative number, such as -5. Since there are 5 spaces between 0 and -5 on the number line, its absolute value equals 5.

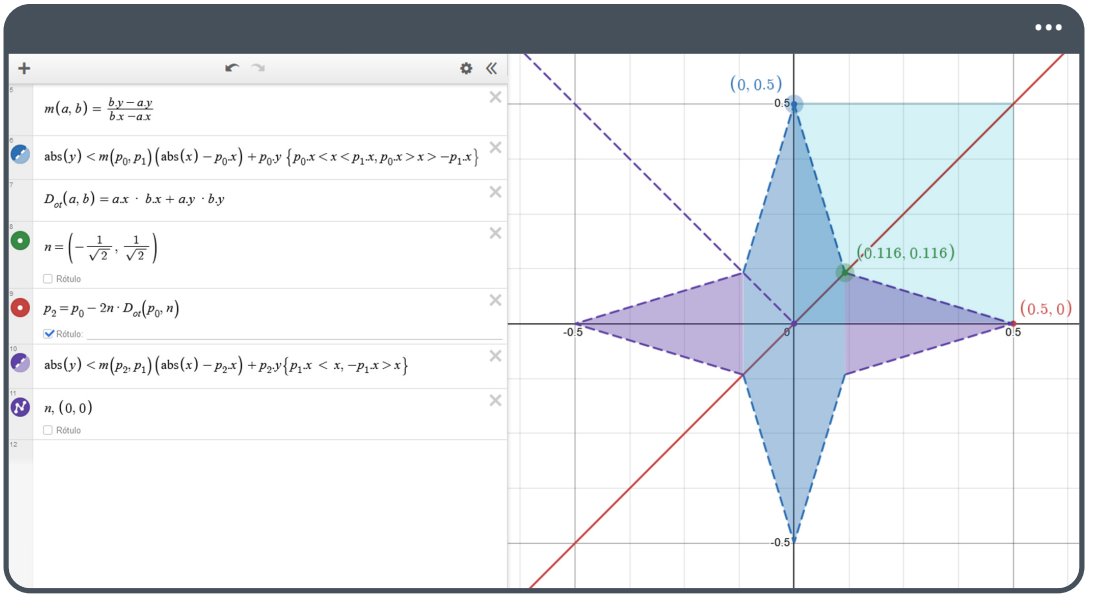
It's worth noting that this operation could be affected due to the limits defined for the x coordinate in both regions. The first segment has been bounded by $\{0 < x < x_0\}$, while the second is bounded by $\{x_0 < x < p_{2^x}\}$. Therefore, you'll need to adjust these limits before applying the absolute value to each coordinate.

Begin by applying the absolute value to y as follows:



(3.1.m https://www.desmos.com/calculator/0501ekurek)

As seen in previous reference, both segments $\overline{p_0\ p_1}$ and $\overline{p_2\ p_1}$ have been reflected with respect to the x-axis due to the use of the absolute value on the y-axis, which is expressed as $abs(y)$ in the example. In computer graphics, this represents an optimization, as no other points in the shape have been calculated. To conclude, you should apply the absolute value to the x coordinate, and update the limits to complete the Shuriken.



(3.1.n https://www.desmos.com/calculator/wsvatszf13)

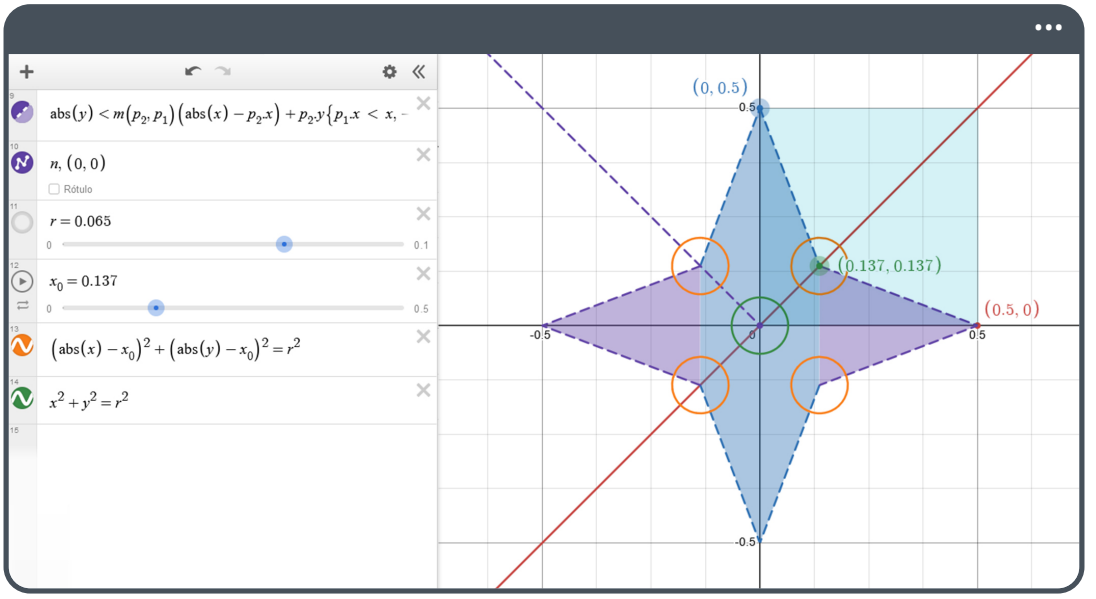
You can see in Figure 3.1.n that the first segment $\overline{p_0\ p_1}$ has been limited twice: when p_{0x} is less than x, and x is less than p_{1x}, that is $\{p_{0x} < x < p_{1x}\}$ and furthermore, when $\{p_{0x} > x > p_{1x}\}$. The same applies to the second segment $\overline{p_2\ p_1}$: first when $\{p_{1x} < x\}$, and then when $\{-p_{1x} > x\}$. In this context, these boundaries are separated by commas, but the result is the same.

At this point, the shape of the Shuriken is complete; all that remains is to add circles to further differentiate the body from the figure. As you've seen throughout this book, you can recreate circles using the formula $x^2 + y^2 = r^2$. However, you'll need to apply the absolute value to both xy coordinates to obtain reflections in the different quadrants. Therefore, you can extend the function, considering that:

$$(|x| - u)^2 + (|y| - u)^2 = r^2$$

(3.1.o)

The above equation includes a new variable u, has been included, which functions to move the absolute values of the xy coordinates before calculating their distance to the center of the circle. When you apply this to the Cartesian plane, you get the following result:



(3.1.p https://www.desmos.com/calculator/bma1c49gww)

As you can see, you've defined a circle for each quadrant where $u = x_0$. In this way, the position of the circles will be equal to the position of point p_1.

Since you already have the equations needed to implement the Shuriken in HLSL, the exercise could conclude at this point. However, you'll perform an additional task to better understand the boundaries when working with segments.

Geometric areas can not only be defined as shapes but also as boundaries. To do this, simply replace the equality symbol with an inequality symbol, that is:

$$(|x| - u)^2 + (|y| - u)^2 > r^2$$

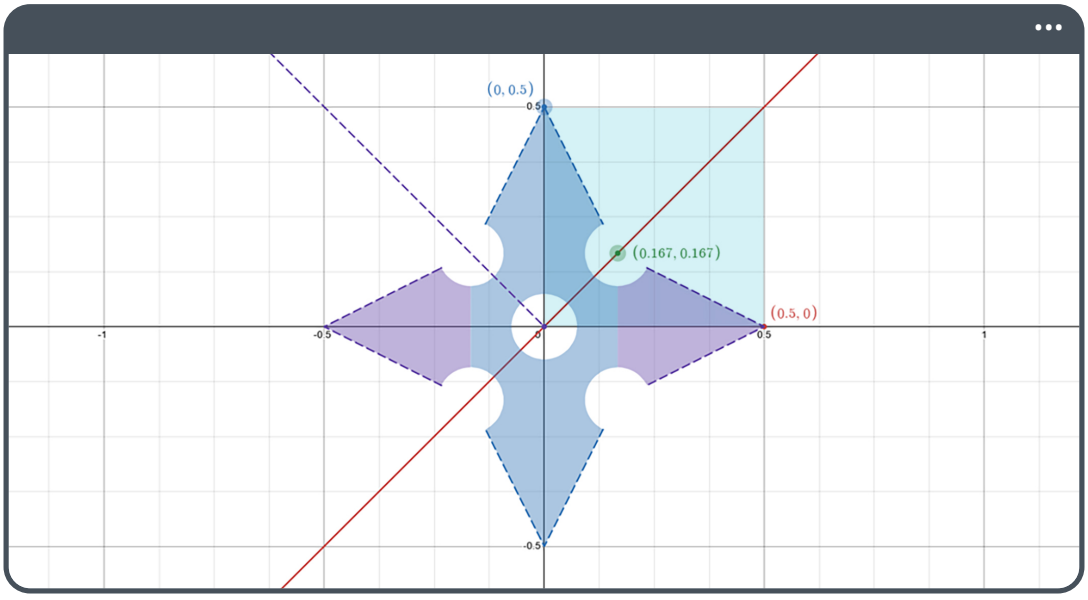
(3.1.q)

In Desmos, for example, you can define the previous equation as a limit in both segments $\overline{p_0\ p_1}$ and $\overline{p_2\ p_1}$, including the function in square brackets, as shown below.

$$|y| < m(|x| - a.x) + a.y \quad \{(|x| - u)^2 + (|y| - u)^2 > r^2\}$$

(3.1.r)

If you add the central circle as a boundary, the Shuriken will be displayed as follows:

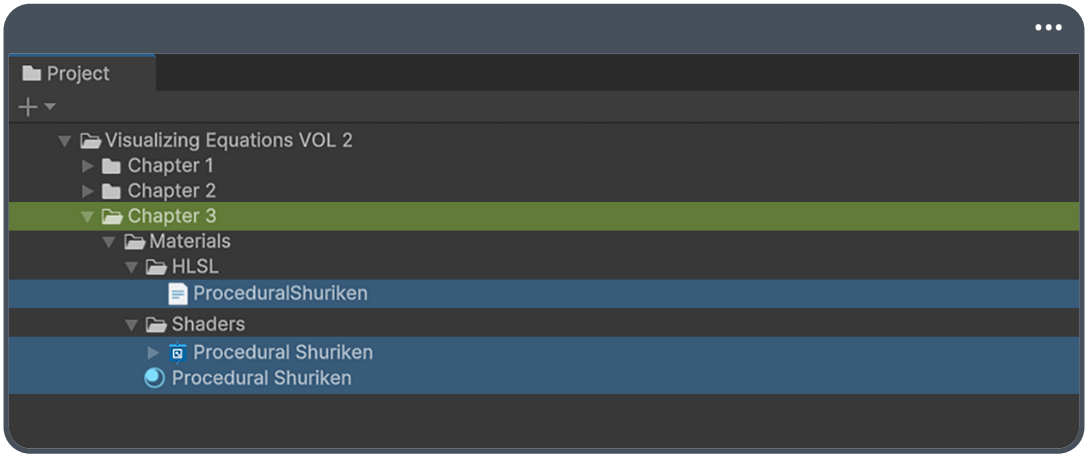


(3.1.s https://www.desmos.com/calculator/8cc82ghuwo)

3.2 Drawing a Shuriken in HLSL.

In this section, we'll review the features mentioned above and implement them in HLSL. To do this, you can use another **Custom Function** node in Shader Graph. Begin by creating a new **Unlit Shader Graph** shader in your project folder and name it **Procedural Shuriken**. Then create both a material and a **.hlsl** script with the same name.

If you follow the structure introduced in the first chapter, your project should be presented as follows:



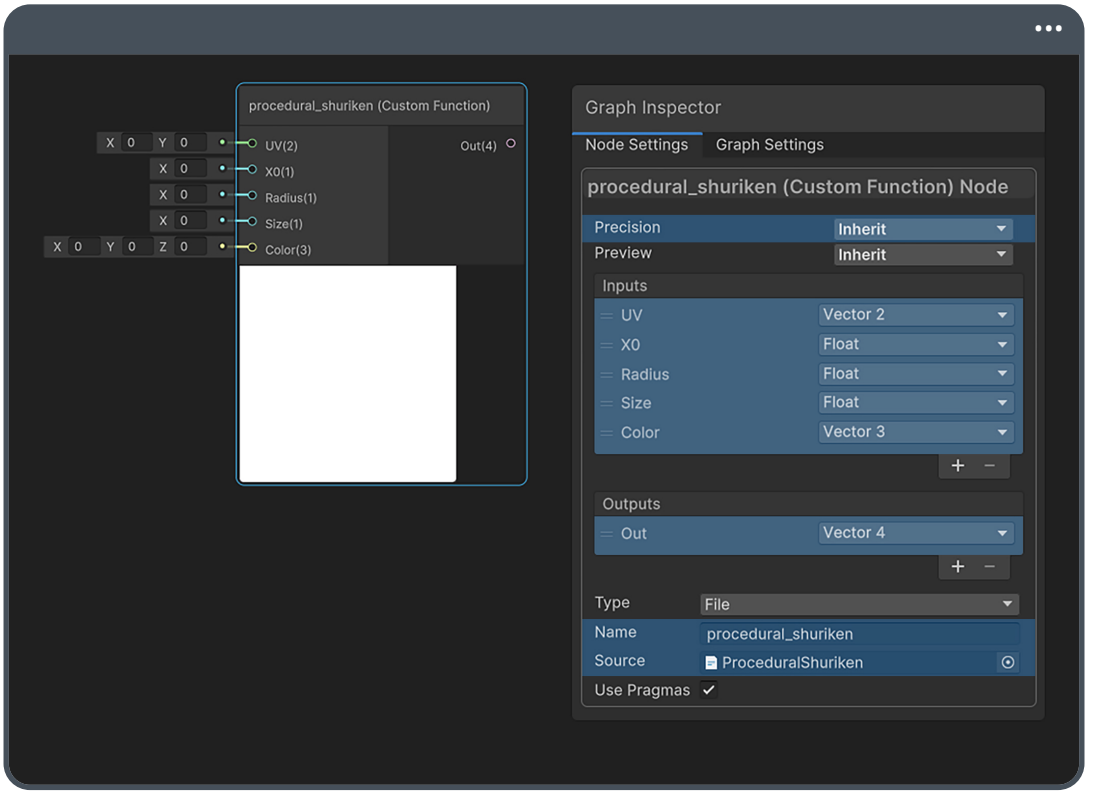
(3.2.a Project structure)

It should be noted that, as has been done in previous chapters, you need to assign the material to the shader and then apply this material to a Quad in the scene to visualize the changes made to the shader itself.

Once in the **Master Stack**, you can proceed to create a **Custom Function** node and call it **procedural_shuriken**. Make sure to include the **ProceduralShuriken** script as a **Source** on the node, keeping its default configuration.

As seen in the previous section, the variables needed to create your procedural shape correspond to x_0 and r. The first modifies both the position of p_0 and that of the circle and its reflections, while the second refers to the radius of the circles. You also need the

UV coordinates to project the shape. Therefore, you can add these variables as inputs to the node, including a float value to modify the total size of the Shuriken in the Quad, and a three-dimensional vector for its color.



(3.2.b procedural_shuriken node)

Generally, four-dimensional vectors are used for procedurally created shapes due to the alpha (A) channel, which means that your shader will contain a transparency channel. Therefore, you need to modify the shader's **Surface Type** property value by going to the **Graph Settings** window and setting it as **Transparent**.

Next, you can start defining the function to draw the Shuriken. To do this, open the **ProceduralShuriken** script and add the following lines of code:

```
void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    in float size, in float3 color, out float4 Out)
{
    Out = 0;
}
```

Looking at the arguments of the function, you can see that the previously defined inputs have been included in the **Custom Function** node. Likewise, both **`x0`** and **`radius`** correspond to the variables entered into the Shuriken function in Desmos. Preliminarily, the **Output** of the function is set to 0.0, only to avoid GPU compilation errors and prevent visual artifacts from appearing while developing your shape.

At this point, you might ask: what do I do first? Defining points p_0 and p_1 might be a starting point. However, you must define the Cartesian coordinates before any equation. Therefore, start by declaring a new method and call it **`shuriken()`**, which will include the **`uv`** coordinates, the **`x0`** variable, and the **`radius`** as arguments.

```
float shuriken(float2 uv, float x0, float radius)
{
    float y = abs(uv.y);
    float x = abs(uv.x);

    return 0;
}

void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    in float size, in float3 color, out float4 Out) { … }
```

Following the explanation in the previous section, two new variables, **x** and **y**, have been declared and initialized, corresponding to the absolute values of the UV coordinates, respectively. Now, you can define the points mentioned above, translating them into HLSL as follows:

```
float shuriken(float2 uv, float x0, float radius)
{
    float y = abs(uv.y);
    float x = abs(uv.x);

    float2 p0 = float2(0.0, 0.5);
    float2 p1 = float2(x0, x0);

    return 0;
}
```

If you observe the previous example, you've declared and initialized the respective points. Note that you can declare point **p0** as constant since its position in time won't change.

You'll need to apply the equation of the line to visualize the different segments that make up the procedural shape. However, the **tangent()** slope function was previously used in section 1.4 of Chapter 1, therefore you won't need to declare and reinitialize it in the current script. You can simply add the Segment.hlsl script as an **#include** in the code, following the path where the file is located:

- Assets > Jettelly Books > Visualizing Equations VOL 2 > Chapter 1 > Materials > HLSL > Segment.hlsl.

In this way, your code looks like this:

```
#include "Assets/Jettelly Books/…/Segment.hlsl"

float shuriken(float2 uv, float x0, float radius)
{
    float y = abs(uv.y);
    float x = abs(uv.x);

    float2 p0 = float2(0.0, 0.5);
    float2 p1 = float2(x0, x0);

    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);

    return segment_p0_p1;
}
```

As you know, the **`#include`** directive is used to include content from one file in another, which is quite useful when organizing code and making it modular.

You can notice that in line 13 the function returns the first segment of the Shuriken. However, it contains some mathematical complexities that could affect the result and consequently present visual artifacts. First, the slope or **`tangent()`** gradient contains a division in its definition, if $x_0 = 0$, the slope becomes infinite. Second, the UV coordinates haven't been normalized, meaning that the output value can exceed the range between [0.0 : 1.0].

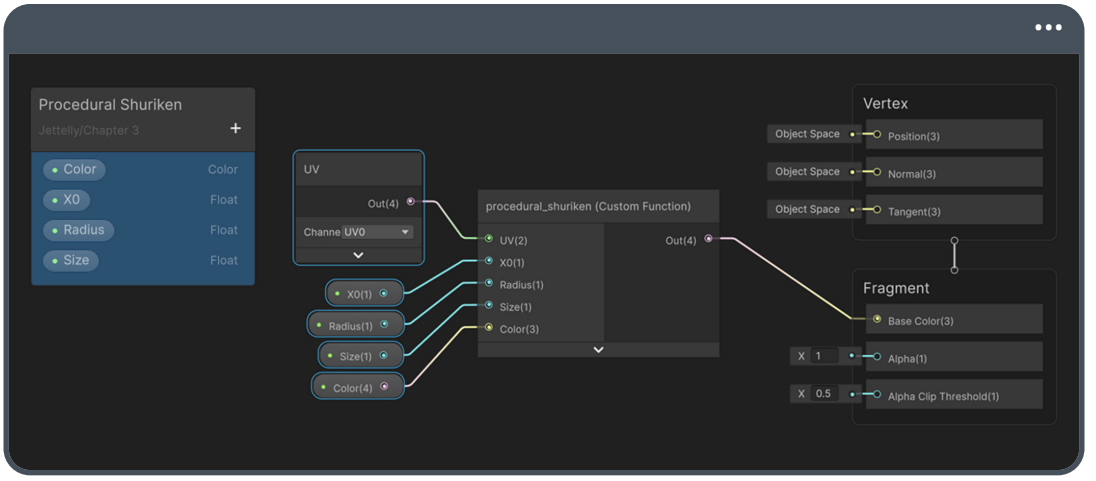
You can easily solve this by using the **`saturate()`** function, which restricts the value of the arguments to your required range. However, to visualize the problem, you'll need to carry out the following operations: include in the **Blackboard** the variables defined as input in the node, i.e., **UV**, **X0**, **Radius**, **Size** and **Color**. Additionally, you can declare a new four-dimensional RGBA vector within the scope of the **`procedural_shuriken_float()`** method to pass the shuriken calculation as output.

Start by declaring and initializing a new four-dimensional vector within the **`procedural_shuriken_float()`** method , as shown in the following example:

```
17 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    ↪ in float size, in float3 color, out float4 Out)
18 {
19     uv -= 0.5;
20
21     float4 shape = shuriken(uv, x0, radius);
22     shape.rgb *= sin(float3(0.123, 0.453, 0.432) * 10); // color
23
24     Out = shape;
25 }
```

One of the most important operations occurs in code line 19, where 0.5 is subtracted from the UV coordinates. This is done to center the procedural image in the Quad. Subsequently, in line 21, the shape vector has been initialized using the result of the **`shuriken()`** method. As a result, its four RGBA channels have the same value, which facilitates the implementation of color over the RGB channels and transparency over the A channel. Finally, in line 22, the RGB channels of the **shape** vector are temporarily multiplied by the **`sin()`** function, which in turn is multiplied by 10, resulting in **`sin(1.23, 4.53, 4.32)`**, which produces a chromatic effect in the final render.

All that remains is for you to declare the properties in **Blackboard** and connect them to the **procedural_shuriken** node, as shown in the following image.

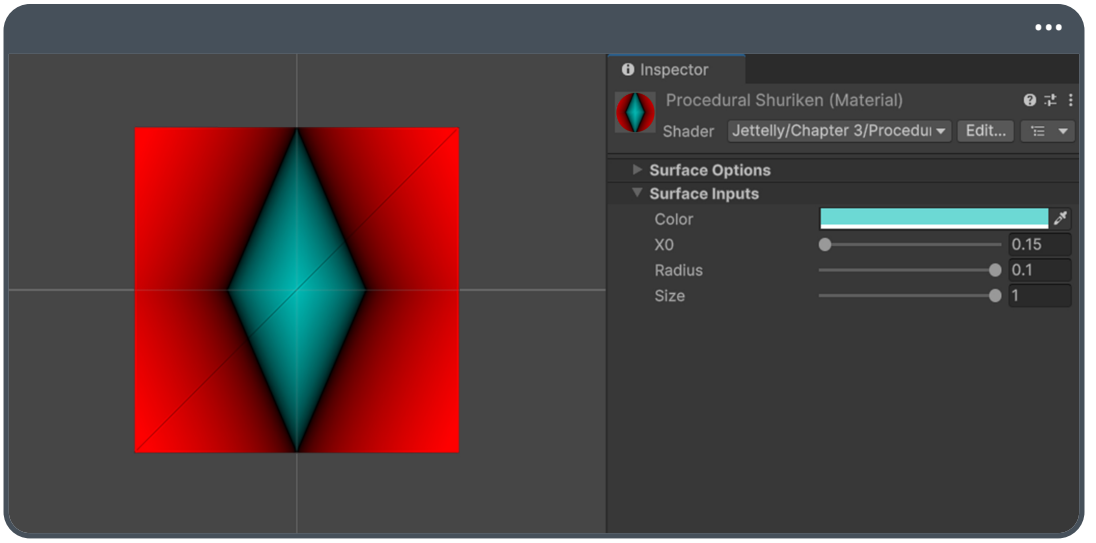


(3.2.c Properties of the procedural_shuriken node)

If you look closely, you'll notice that only the **Out** of your custom function has been connected as **Base Color** in the shader. If you want transparencies you'll also need to connect the alpha channel. However, you can leave this process for the end, as the focus will first be on correctly defining and adjusting the parameters and functions to determine the visual aspect of the Shuriken.

It should be noted that some property values have been limited in order to obtain a result that corresponds to the form of the Shuriken. For example, **X0** has a range between [0.15 : 0.25], while **Radius** has a range between [0.0 : 0.1]. Finally, **Size** has a range between [0.0 : 1.0].

If all the steps have been done correctly, the current shape should look like this:

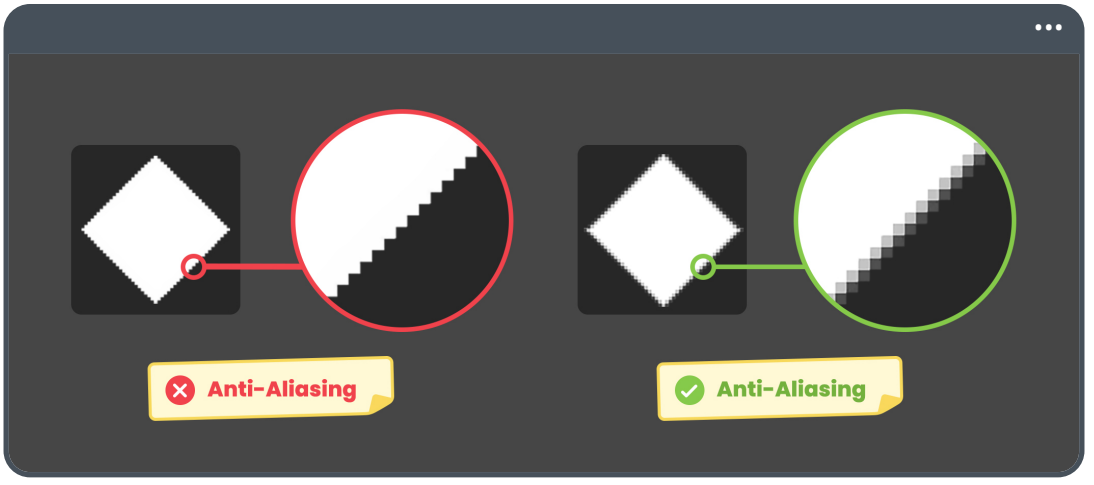


(3.2.d)

Looking at the previous reference, you'll notice that the center of the shape has been colored with a bluish hue, indicating that the values within the **shape** variable are different from 0.0. As mentioned earlier, you can easily fix this by applying the **`saturate()`** function over the first segment. However, before you continue with its implementation, we'll digress and talk about a common problem that occurs when drawing procedural shapes, which is directly related to its resolution.

There are two predefined functions that are frequently used when defining edges or shapes: we're referring to **`step()`** and **`smoothstep()`**. Both are important tools for manipulating edges and transitions in procedural figures. The **`step()`** function takes two arguments: a threshold and an input value. If the input value is greater than or equal to the threshold, the function returns 1; otherwise, it returns 0. This feature is useful for creating abrupt transitions or sharply defined edges.

For example, if you apply the **`step()`** function directly to the segment, with a threshold of 0, and then rotate the Quad in your scene you'll get the following result.



(3.2.e You get hard edges when using the **`step()`** function)

On the other hand, the **`smoothstep()`** function allows for smoother transitions between two boundaries. It accepts three arguments: a lower limit, an upper limit, and an input value. The transition between [0 : 1] is done gradually. However, it also generates visual artifacts at the edges when rotating the Quad in space. This occurs mainly because a smooth transition is being applied over the shape itself, rather than over the area of pixels on the screen.

To solve this issue, you'll need to create a function that allows for a certain level of anti-aliasing on the figure under development.

3.3 Implementing Anti-Aliasing.

One way to generate a smooth transition between pixels is through the **`fwidth()`** function. According to its official documentation, this function returns the sum of the absolute values of each approximate partial derivative of an input **a** with respect to both xy coordinates in screen-space. What does this mean? Consider its definition in CG language to gain a better understanding:

```
float3 fwidth(float3 a)
{
    return abs(ddx(a)) + abs(ddy(a));
}
```

The concept of a "partial derivative" refers to the derivative of a multivariate function with respect to one of its variables, while keeping the others constant. You can use it to analyze how the function changes when only one of its independent variables varies, while the others remain fixed. For example, so far this book has reviewed $f(x)$ functions of a single variable, but what if there is a function $f(x, y)$ with two variables?

The partial derivative of f with respect to x is denoted as:

$$\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}$$

(3.3.a)

While the partial derivative of f with respect to y denoted as:

$$\frac{\partial f}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y}$$

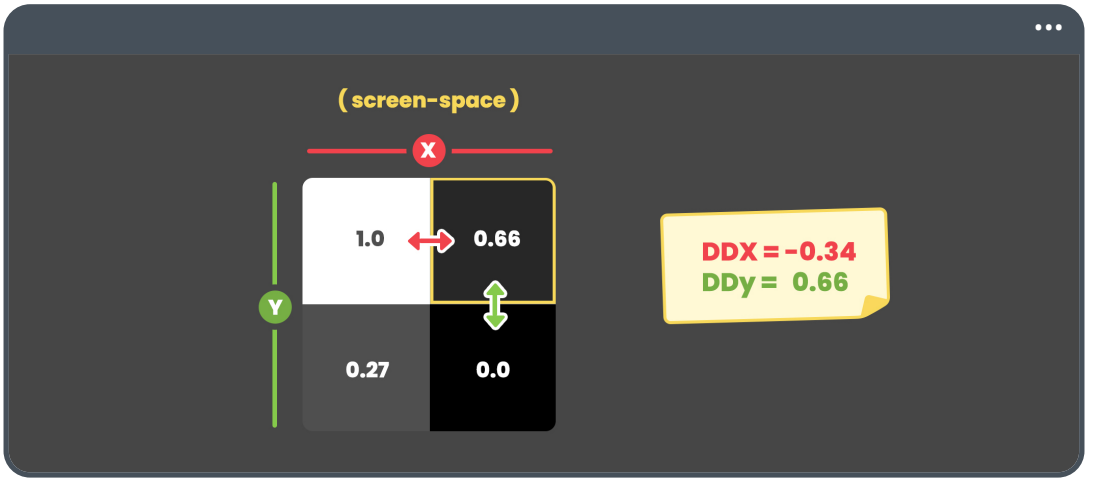
(3.3.b)

The **`ddx()`** and **`ddy()`** functions, both included in **`fwidth()`**, are approximations of the equations shown in Figure 3.3.a and 3.3.b, respectively. This means:

$$ddx = \frac{\partial f}{\partial x}$$
$$ddy = \frac{\partial f}{\partial y}$$

(3.3.c)

Therefore, **ddx()** calculates the rate of change of an expression with respect to the x coordinate of the screen, while **ddy()** performs the same operation with respect to the y coordinate. To understand the process, consider four pixels on the screen of different colors and analyze their rate of change using these functions.



(3.3.d)

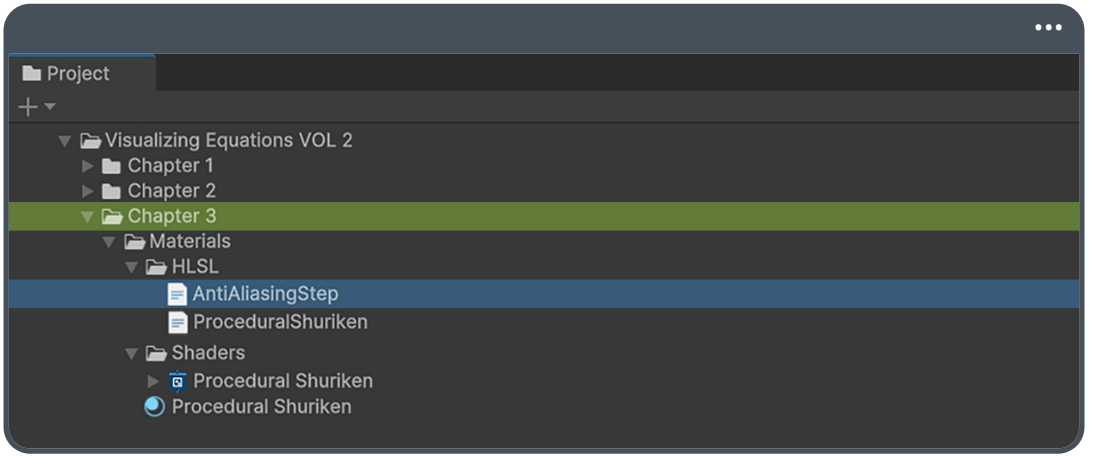
From the previous reference, the pixel in the upper corner marked in yellow has a rate of change equal to -0.34 with respect to the preceding pixel in the x coordinate, while in the y coordinate, the rate of change is equal to 0.66. Considering that **fwidth()** returns the absolute values of both **ddx()** and **ddy()**, the rate of change of the x coordinate in the example above would be positive.

Now, how can you generate an anti-aliasing filter using these functions? Just follow these steps:

- Calculate the rate of change of the pixels that your on-screen Shuriken covers.
- Create a border where the anti-aliasing effect will be applied.
- Perform inverse linear interpolation to smooth the pixels at the edges of the shape.

Begin by adding a new HLSL script to your project and name it **AntiAliasingStep**. This file will be used from this point onward in all the procedural figures generated in this book.

If everything has gone well, your project should look like this:



(3.3.e Project structure)

In the new script, you'll begin by declaring a method of type **`float3`** and name it **`anti_aliasing_step()`**. This method will have two arguments: a gradient, to be passed as an argument in the **`fwidth()`** function, and a threshold.

```
1  float3 anti_aliasing_step(float3 gradient, float edge)
2  {
3      float3 rate_of_change = fwidth(gradient);
4
5      return 0;
6  }
```

As you can see, line 3 of the previous example has declared and initialized a new variable called **`rate_of_change`** which determines the partial derivative with respect to a pixel in the screen space. Next, you can define the edge or range to determine the region where the anti-aliasing effect will be applied. To do this, add the following code:

```
float3 anti_aliasing_step(float3 gradient, float edge)
{
    float3 rate_of_change = fwidth(gradient);

    float3 lower_edge = edge - rate_of_change;
    float3 upper_edge = edge + rate_of_change;

    return 0;
}
```

Finally, using an inverse linear interpolation, you can scale the gradient value to a range between [0 : 1] based on its position relative to the lower edge and the top edge. This is presented in the following example:

```
1  float3 anti_aliasing_step(float3 gradient, float edge)
2  {
3      float3 rate_of_change = fwidth(gradient);
4
5      float3 lower_edge = edge - rate_of_change;
6      float3 upper_edge = edge + rate_of_change;
7
8      float3 stepped = (gradient - lower_edge) / (upper_edge - lower_edge);
9      stepped = saturate(stepped);
10
11     return stepped;
12 }
```

From the example above, you can see that line 11 has returned the **`stepped`** value, which is now guaranteed to be in the range [0 : 1]. This value represents the smoothed transition of the gradient in the specific area of the edge, providing an anti-aliasing effect.

To implement this effect in your procedural Shuriken, you can integrate the **`anti_aliasing_step()`** method into the **procedural_shuriken** node of the shader.

This can be done by using the **#include** directive, including the file path, which corresponds to:

- Assets > Jettelly Books > Visualizing Equations VOL 2 > Chapter 1 > Materials > HLSL > AntiAliasingStep.hlsl.

As a result, the code looks like this:

```
#include "Assets/Jettelly Books/…/Segment.hlsl"
#include "Assets/Jettelly Books/…/AntiAliasingStep.hlsl"

float shuriken(float2 uv, float x0, float radius)
{
    float y = abs(uv.y);
    float x = abs(uv.x);

    float2 p0 = float2(0.0, 0.5);
    float2 p1 = float2(x0, x0);

    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
    segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);

    return segment_p0_p1;
}
```

To better understand this, save the changes and return to the scene. You'll notice that the first segment is drawn perfectly on the screen. Even if the Quad is rotated approximately eighty degrees, the figure's resolution remains intact.

Following Equations 3.1.f and 3.1.g from the previous section, you'll need to determine the normal of the slope to establish the position of point **p2**, which will be the reflection of point **p0**. To do this, perform the following exercise:

```
    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
    segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);

    float2 n = float2(-1 / sqrt(2), 1 / sqrt(2));
    float2 p2 = p0 - 2.0 * n * dot(p0, n);
```

You can see that the previous code has declared a new variable **n**, which refers to the normal of the slope. Subsequently, this variable has been used in the calculation of point **p2**, which you will use to determine the second segment—the connecting the points $\overline{p_2\ p_1}$.

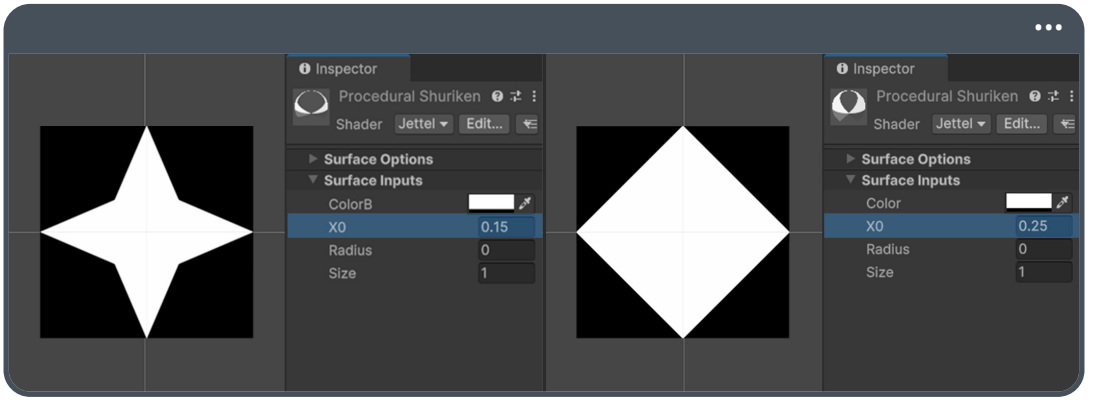
Now, you can declare and initialize a new variable called **`segment_p2_p1`**. Since this new segment has the same structure as the first one in mathematical terms, the same functions can be used, with the difference that they'll apply the corresponding points for each case. This process allows you to clearly define the second segment and ensure continuity and symmetry in the form of the Shuriken.

```
    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
    segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);

    float2 n = float2(-1 / sqrt(2), 1 / sqrt(2));
    float2 p2 = p0 - 2.0 * n * dot(p0, n);

    float segment_p2_p1 = y - (tangent(p2, p1) * (x - p2.x) + p2.y);
    segment_p2_p1 = 1.0 - anti_aliasing_step(segment_p2_p1, 0.0);
    segment_p0_p1 += segment_p2_p1;

    return saturate(segment_p0_p1);
}
```

You can see that line 18 has declared and initialized the second segment, which joins points $\overline{p_2\ p_1}$, and then, in the next line, applied anti-aliasing to the same segment to obtain a smoothed line. This second segment is then added to the first in line 20. Since

this operation generates values outside the desired range for the render, line 22 has applied the **`saturate()`** function, which sets a maximum range between [0.0 : 1.0] for the output value.

After saving the changes, if you modify the **x0** property value from the Inspector, you can see the shape of the Shuriken expand and contract dynamically.



(3.3.f)

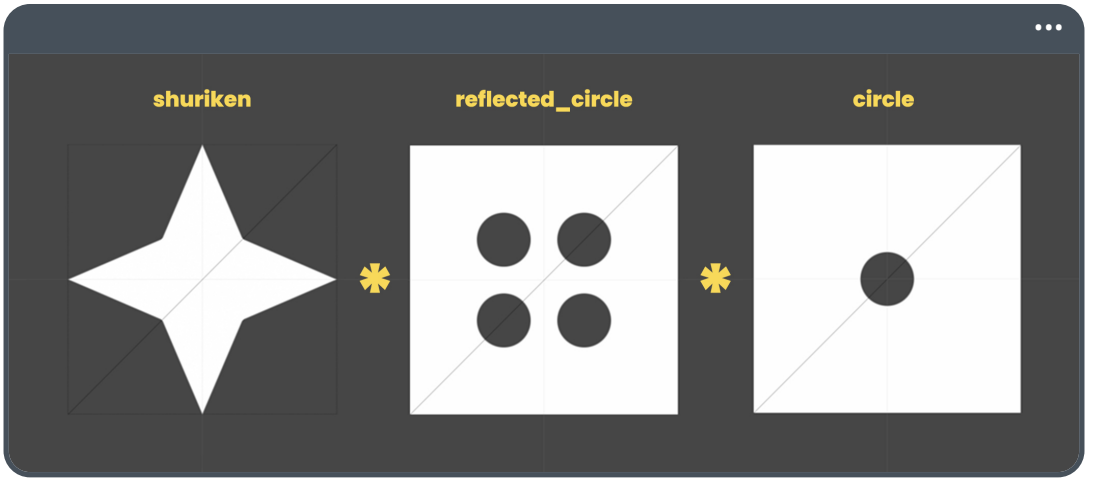
To conclude this procedural shape, you can apply the formulas presented in equations from Figure 1.3.b and 3.1.o to generate the circles of the shape being developed. Add the following lines of code to accomplish this:

```
18      float segment_p2_p1 = y - (tangent(p2, p1) * (x - p2.x) + p2.y);
19      segment_p2_p1 = 1.0 - anti_aliasing_step(segment_p2_p1, 0.0);
20
21      float r = radius * radius;
22      float reflected_circle = (x - x0) * (x - x0) + (y - x0) * (y - x0);
23      reflected_circle = anti_aliasing_step(reflected_circle, r);
24
25      float circle = (x * x) + (y * y);
26      circle = anti_aliasing_step(circle, r);
27
28      segment_p0_p1 += segment_p2_p1;
29      segment_p0_p1 *= reflected_circle;
30      segment_p0_p1 *= circle;
31
32      return saturate(segment_p0_p1);
33  }
```

The previous example has declared and initialized a new variable called **`reflected_circle`** (line 22), corresponding to the holes of each Shuriken tip. The process is then repeated in line 25, with the declaration and initialization of the **`circle`** variable. However, since the latter has no subtraction in its coordinates, it remains constant in the center of the plane.

It's worth noting that the process of declaring and initializing circles could have been optimized by declaring a new method containing the relevant variables. However, for this exercise it has been carried out as shown to maintain fidelity to the mathematical explanation presented in the previous section.

Finally, lines 29 and 30 have multiplied the first segment by each defined circle. Since the area of the circles is black (equals 0.0), when multiplying, these areas are subtracted from the original Shuriken, generating the desired effect for each case. This means:



(3.3.g Composition of a procedural Shuriken)

Up to this point, your procedural shape is conceptually complete. Therefore, you can use it to generate color masks or specific effects for the UI.

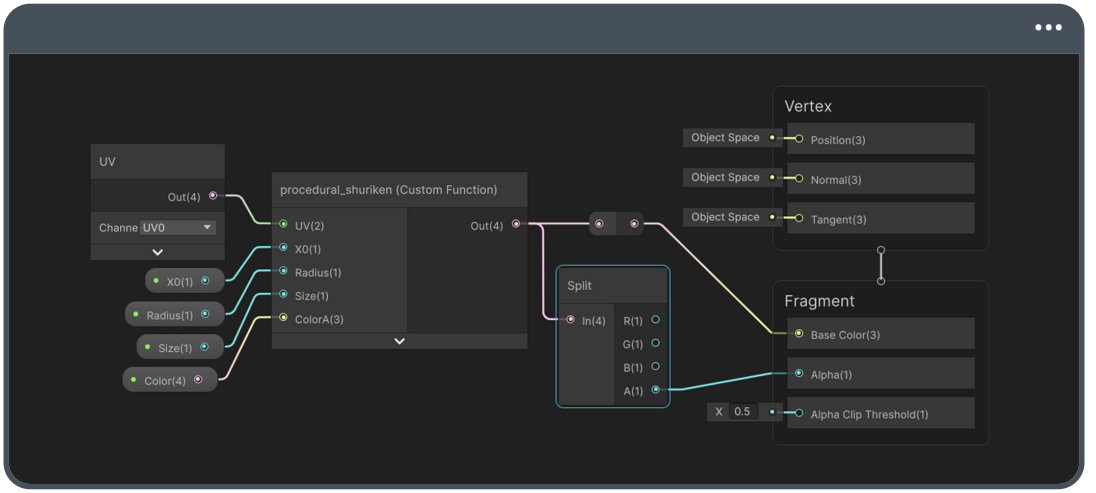
Before returning to the scene, ensure that you implement a function to increase or decrease the size of the Shuriken. To do this, you can use the following operation within the **`procedural_shuriken_float()`** field.

```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    ↪ in float size, in float3 color, out float4 Out)
36 {
37     uv -= 0.5;
38     uv *= lerp(2.0, 1.0, size);
39
40     float4 shape = shuriken(uv, x0, radius);
41     shape.rgb *= color;
42
43     Out = shape;
44 }
```

Looking at the previous example, you can see that the UV coordinates have been multiplied by a linear interpolation using the **`lerp()`** function (line 38). Considering that the **`size`**

property has a range set between [0.0 : 1.0], when its value equals 0.0, the UV coordinates are multiplied by 2.0, which decreases the size of the Shuriken within the Quad. You can easily see this effect by modifying the value of the property directly from the Inspector in Unity.

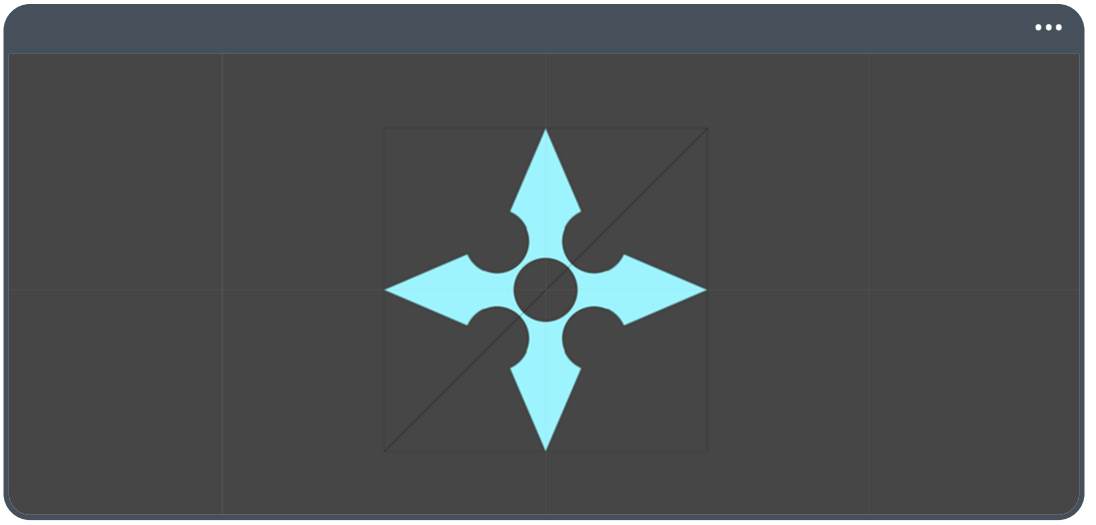
As a final step, ensure to connect the alpha channel as the output in the shader. You can do this by using a **Split** node, which separates the channels of any vector.



(3.3.h)

Admittedly, you could have implemented a dedicated output for the alpha channel directly on the **procedural_shuriken** node, instead of using a **Split** node. However, opting for the latter provides greater flexibility and allows you to modify the channels individually, if necessary, without altering the rest of the vector.

If all the steps have been done correctly, the black area on the Shuriken should appear transparent, as shown in the reference below.



(3.3.i Shuriken's body on UV coordinates)

3.4 Defining an Aesthetic for the Shuriken.

In this section, we'll dedicate our time to extending the shader you're developing, improving its aesthetics through color layers and custom light effects, which we'll implement linearly to understand the process.

It's evident that a real Shuriken is characterized by having sharp edges, which are essentially conveyed as a different color from their general shape. Therefore, your first step in the shader would be to add support for a second color channel to generate a difference between the body and its edges. You can start this by extending the **`procedural_shuriken_float()`** method to include one of the color inputs. Additionally, ensure to rename the current color entry to avoid any confusion.

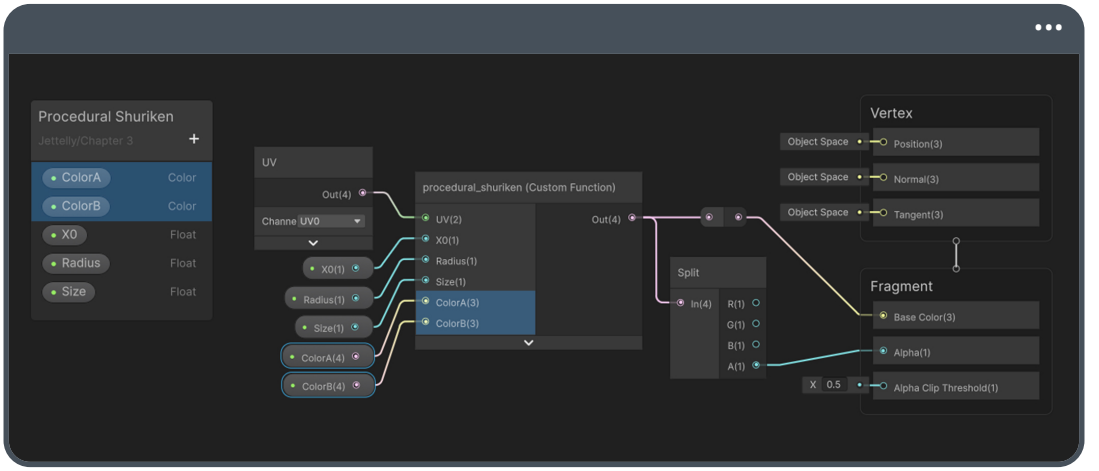
```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    ↪ in float size, in float3 colorA, in float3 colorB, out float4 Out) { … }
```

As seen in the example above, you've included a new input as an argument in the function, corresponding to a three-dimensional RGB vector called **`colorB`**. In addition, the first input color has been renamed from **`color`** to **`colorA`**. Given this change, the program will inevitably stop compiling, resulting in a magenta color lying over the Quad in the scene. Therefore, it's essential that you update both the inputs of the **Custom Function** node in the **Shader Graph**, as well as the properties in the **Blackboard**.

You can take the following steps to fix this issue:

- Update the **Custom Function** node: Include the new three-dimensional vector named **colorB** as input and rename the current color vector to **colorA**.
- Update the **Blackboard**: Start by renaming the current color vector as **ColorA**, and then include a new color called **ColorB**. Finally, connect the properties to the node under development, and save the changes.

If everything has been done correctly, the shader should look like the following:



(3.4.a Color properties of procedural_shuriken node)

Having made these changes, you can return to the code to implement both the Shuriken's blade and body, differentiating them in color and shape.

```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
   ↪ in float size, in float3 colorA, in float3 colorB, out float4 Out)
36 {
37     uv -= 0.5;
38     uv *= lerp(2.0, 1.0, size);
39
40     const float d0 = 0.9;
41     float sharp = shuriken(uv, x0, radius * d0);
42     float body = shuriken(uv, x0 * d0, radius);
43
44     float4 shape = 0;
45
46     Out = shape;
47 }
```

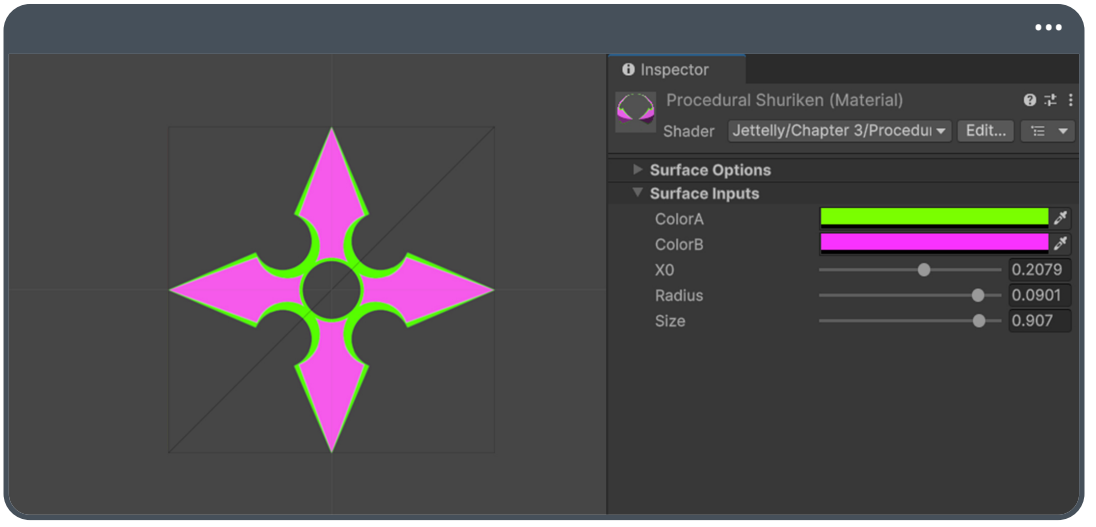
If you pay attention to lines 41 and 42 in the previous example, they have declared and initialized two new variables called **sharp** and **body**, which refer directly to the components of the Shuriken. In addition, line 40 has declared and initialized a new constant variable called **d0**. Its purpose is to generate a difference between the cutting edge and its body by multiplying the different arguments according to each case.

If you want to see both shapes working together, you can employ a color technique to pass two colors in a linear interpolation and then return the first or second depending on a grayscale input. You could deduce that the two input colors correspond to **colorA** and **colorB**, while grayscale could be the multiplication between **sharp** and **body**. However, in this case, one of the two forms needs to have a different range or brightness from the other; otherwise, it cannot be used as an argument in linear interpolation.

```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    ↪ in float size, in float3 colorA, in float3 colorB, out float4 Out)
36 {
37     uv -= 0.5;
38     uv *= lerp(2.0, 1.0, size);
39
40     const float d0 = 0.9;
41     float sharp = shuriken(uv, x0, radius * d0);
42     float body = shuriken(uv, x0 * d0, radius);
43
44     float v0 = (sharp * d0) * body;
45
46     float4 shape = 0;
47     shape.rgb = lerp(colorA, colorB, v0);
48     shape.a = sharp;
49
50     Out = shape;
51 }
```

In the example above, line 44 has declared and initialized a new variable called **`v0`**, which corresponds to the multiplication between **`sharp`** and **`body`**, with **`d0`** multiplying the former to generate a difference in brightness, translating to grayscale. Subsequently, line 47 has introduced a linear interpolation to each RGB component of the vector **`shape`**, where the minimum value is **`colorA`**, the maximum value is **`colorB`**, and the grayscale is **`v0`**. Additionally, the fourth A component of **`shape`** acquires the value of the **`sharp`** variable, which is essential for establishing the transparency channel of the entire shape.

If you save the changes and return to the scene, you can observe the effect of both colors on the Shuriken's body.

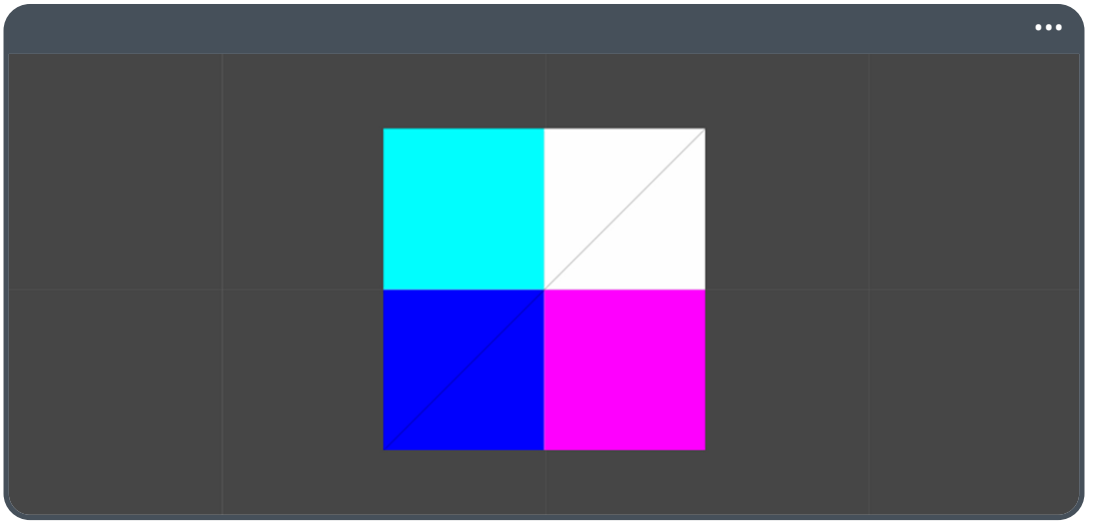


(3.4.b)

At this point, you could say your procedural figure is finished. However, since the form itself lacks volume, you can explore your artistic vision to incorporate a layer of light into it. To do this, considering the UV coordinates, add a different color block to each quadrant, so that you obtain a balance between dark and light areas in the shape. You can start by declaring a new three-dimensional vector in the function, as shown below:

```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
    ↪ in float size, in float3 colorA, in float3 colorB, out float4 Out)
36 {
37     uv -= 0.5;
38     uv *= lerp(2.0, 1.0, size);
39
40     const float d0 = 0.9;
41     float sharp = shuriken(uv, x0, radius * d0);
42     float body = shuriken(uv, x0 * d0, radius);
43
44     float v0 = (sharp * d0) * body;
45     floa3 l0 = anti_aliasing_step(float3(uv, 1.0), 0.0);
46
47     float4 shape = 0;
48     shape.rgb = lerp(colorA, colorB, v0);
49     shape.a = sharp;
50
51     Out = shape;
52 }
```

As you already know, the **`anti_aliasing_step()`** method returns a smoothed transition of any gradient. Observing line 45 of the above code, you can see that a three-dimensional RGB vector has been used as an argument in the function, where the first two elements correspond to the UV coordinates, and the third is a constant number, which equals 1. The output of this vector produces the following graphical result:



(3.4.c)

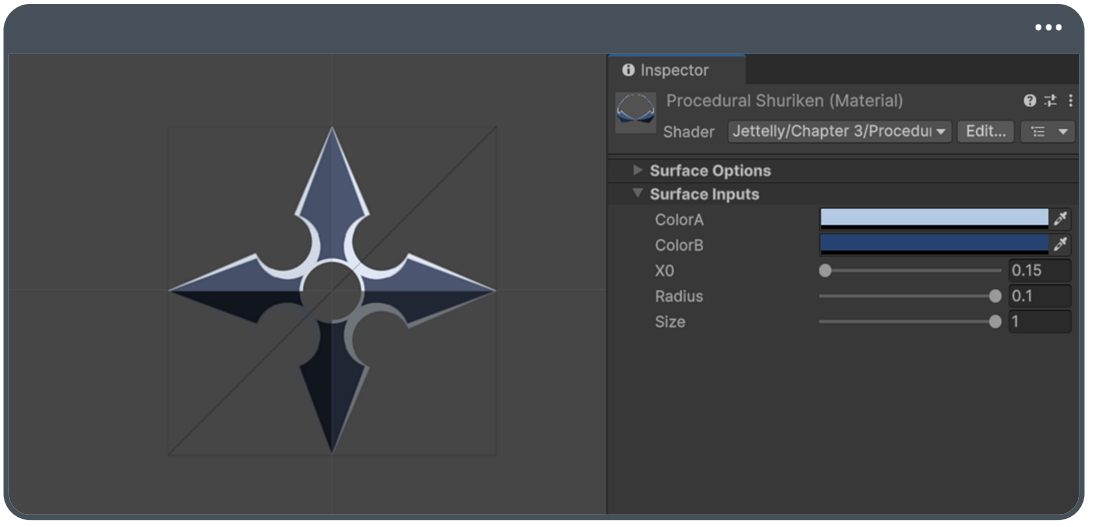
Why are these colors generated? The answer lies in the UV coordinates. While the third component B is a constant value, in this case the UV coordinates correspond to a range between [-0.5 : 0.5] in this context, which means that the **`anti_aliasing_step()`** method will create regions where interpolation results in different values for each quadrant, i.e.

Quadrant 1	Quadrant 2	Quadrant 3	Quadrant 4
R1G1B1	**R0G1B1**	**R0G0B1**	**R1G0B1**
$x > 0$	$x < 0$	$x < 0$	$x > 0$
$y > 0$	$y > 0$	$y < 0$	$y < 0$

However, since you only need highlights and shadows, the equivalent of each hue will be transformed into grayscale. To achieve this, use the following function.

```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
   ↪ in float size, in float3 colorA, in float3 colorB, out float4 Out)
36 {
37     uv -= 0.5;
38     uv *= lerp(2.0, 1.0, size);
39
40     const float d0 = 0.9;
41     float sharp = shuriken(uv, x0, radius * d0);
42     float body = shuriken(uv, x0 * d0, radius);
43
44     float v0 = (sharp * d0) * body;
45     floa3 l0 = anti_aliasing_step(float3(uv, 1.0), 0.0);
46     float l0_gray = saturate(dot(l0, float3(0.126, 0.7152, 0.0722)));
47
48     float4 shape = 0;
49     shape.rgb = lerp(colorA, colorB, v0);
50     shape.rgb *= l0_gray;
51     shape.a = sharp;
52
53     Out = shape;
54 }
```

You can observe that line 46 has declared a new variable called **`l0_gray`**, which acquires the result of the dot product between **`l0`** and a three-dimensional vector corresponding to the values used to convert a color to its grayscale equivalent. These values correspond to the relative luminescence coefficients for the RGB components in linear space. Subsequently, line 50 multiplies the RGB components of **`shape`**, your final shape by **`l0_gray`**, generating light and dark areas in the Shuriken, as seen in the following image:

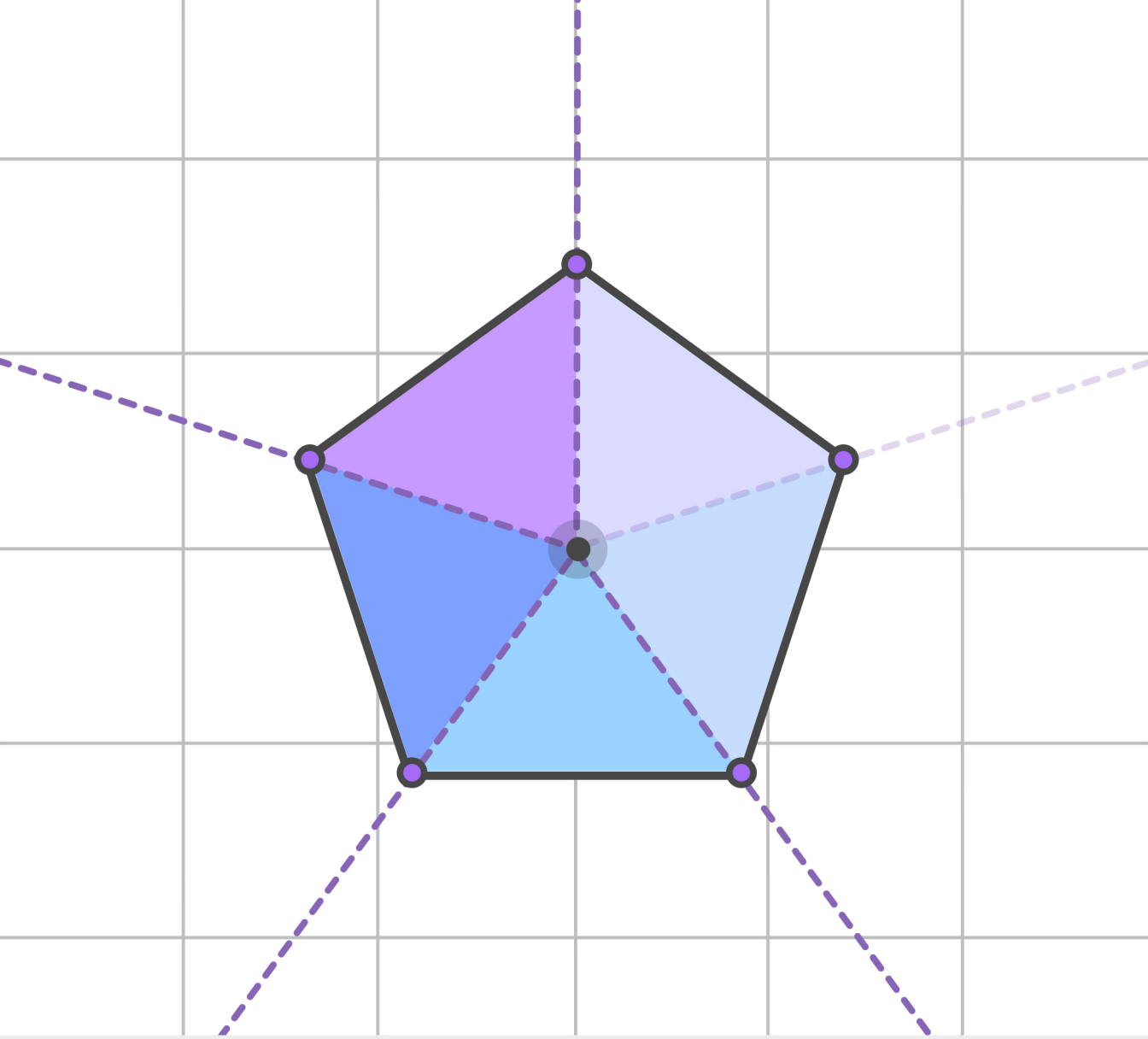

Inspector
Procedural Shuriken (Material)
Shader
Jettelly/Chapter 3/Procedu
Edit...
Surface Options
Surface Inputs
ColorA
ColorB
X0
0.15
Radius
0.1
Size
1

(3.4.d)

Chapter Summary.

- In this chapter, we've explored techniques for creating procedural shapes using mathematical functions, focusing on the development of a Shuriken as a case study. We began with an introduction to the structure of a Shuriken from a mathematical perspective, demonstrating how to break down a complex shape into simple geometric components. Later, in Section Two, we put the mathematical concepts covered so far into practice to develop the Shuriken in Shader Graph using HLSL. This section explained how to translate mathematical formulas into graphical operations to obtain a precise and optimized shape.

- Next, we addressed the issue of pixelated edges common in procedural shapes. We introduced the **`fwidth()`** function as a solution for implementing anti-aliasing, improving the smoothness of the Shuriken's contours. Finally, with the Shuriken's shape completed, we applied color and lighting techniques to enhance the visual aesthetics of the render.



Chapter 4

Signed Distance Function.

In the first few chapters we focused on creating shapes using mostly algebraic and trigonometric functions. These functions establish a solid foundation for the generation of basic shapes. However, to create more complex and detailed geometries, we need a broader approach.

In this chapter we'll delve into the fascinating world of signed distance functions (SDF), with a special focus on their **two-dimensional** applications. SDFs can represent geometry efficiently and accurately, providing a powerful tool for defining and manipulating procedural shapes. Through examples and practical applications, we'll explore how these functions can simplify the creation of complex shapes and unlock new possibilities in technical art development in Unity.

We'll examine the theory behind SDFs, learn how to implement them in projects, and discover how you can integrate them with other techniques to achieve stunning visual effects. This approach won't only enhance your understanding of procedural geometry, but also equip you with valuable innovative tools in game design and development.

4.1 Nature of an SDF.

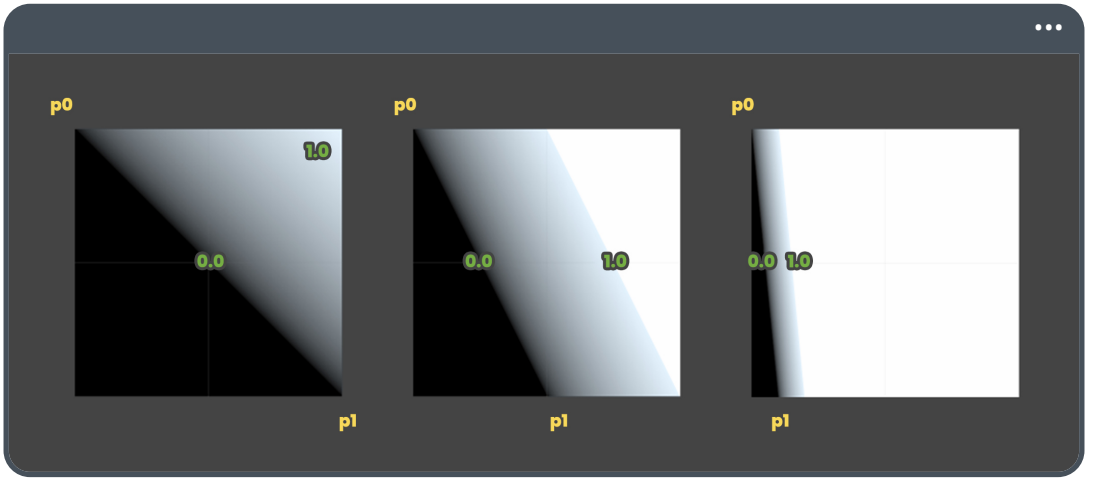
Up to this point, we've put into practice the implementation of polynomial and trigonometric functions in HLSL, which are used to draw procedural shapes from geometric areas and line segments. However, when it comes to complex shapes, explicit functions, such as those defined in the form $y = mx + b$ they are often insufficient when you need to represent irregular curves and contours.

In these cases, we need to resort to implicit functions or parametric representations, which provide greater flexibility and precision. This is where the concept of the signed distance function comes in, which allows us to accurately describe the distance from a point to a surface, indicating whether that point is inside, outside or on the surface itself.

To help understand this concept, we'll compare two segments: one explicitly defined, following Equation 1.1.a in Chapter 1, and the other implicitly defined using the SDF technique. For the latter, we need to consider the following:

- All the points of the first quadrant.
- The distance between the points and the segment to be defined.

These concepts may initially seem challenging to grasp. However, as we progress we'll delve deeper into the logic behind this technique, with a step-by-step explanation of its implementation for various forms. Now, let's examine the gradient of a segment in HLSL defined in the linear form.



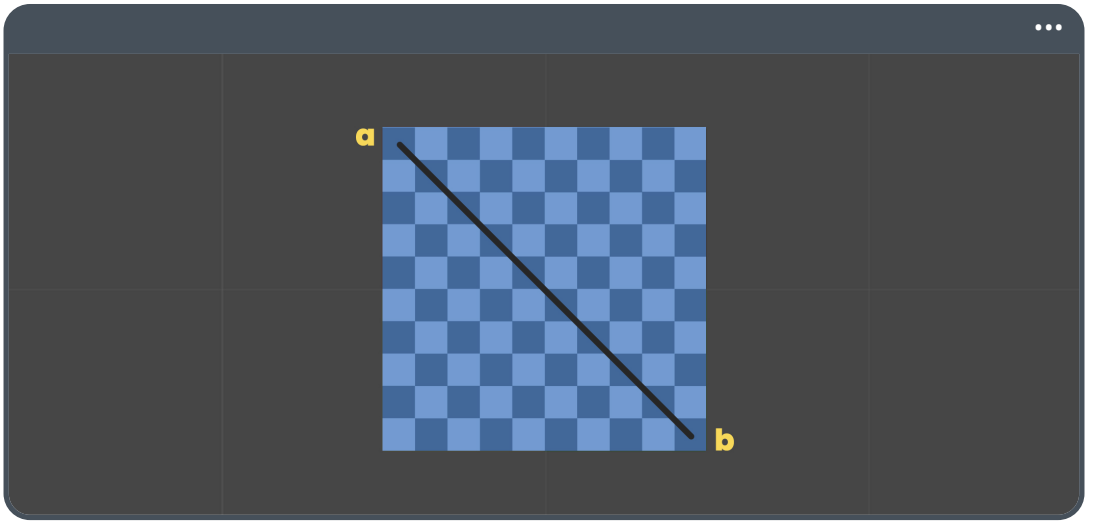
(4.1.a The gradient from a linear function)

As illustrated in Figure 4.1.a, you can see that the gradient expands or contracts according to the positions of points p_0 and p_1. While this behavior is mathematically correct, it presents a challenge when defining complex shapes within your program. For instance, when you apply the **`smoothstep()`** function to the resulting operation it produces an uneven gradient influenced by the position of both points. To address this issue, a different approach is required: calculating the distance between the points outside the segment (represented by the white area in the image) and the segment itself.

To better understand this concept, visualize the first quadrant of the Cartesian plane as a grid. Each space on the grid represents a specific position, which can be determined as a two-dimensional vector.

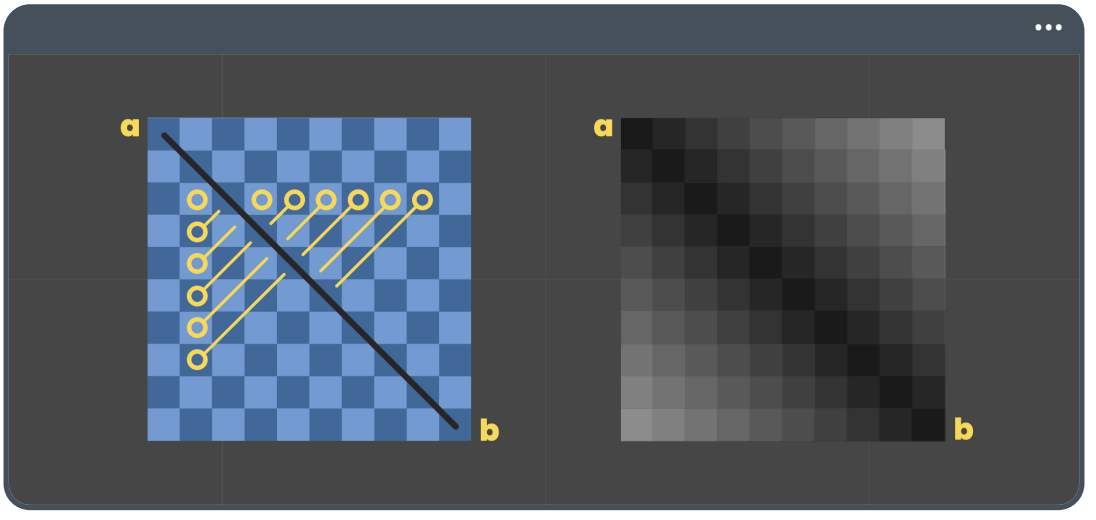
(4.1.b)

You can see that in the example shown in Figure 4.1.b, 100 unique spaces have been used in the grid to illustrate the operation of the signed distance function. As you can imagine, each space represents a point that could eventually be used to perform an operation. For example, a diagonal segment can be drawn passing between the points $a = (0.1, 1.0)$ and $b = (1.0, 0.0)$, as shown below:



(4.1.c Two points on a Cartesian Plane)

We could easily represent the segment shown in Figure 4.1.c in linear form. However, as previously discussed, this approach can generate graphical artifacts as the points approach zero. It's preferable to use the signed distance function to achieve the desired result instead.



(4.1.d)

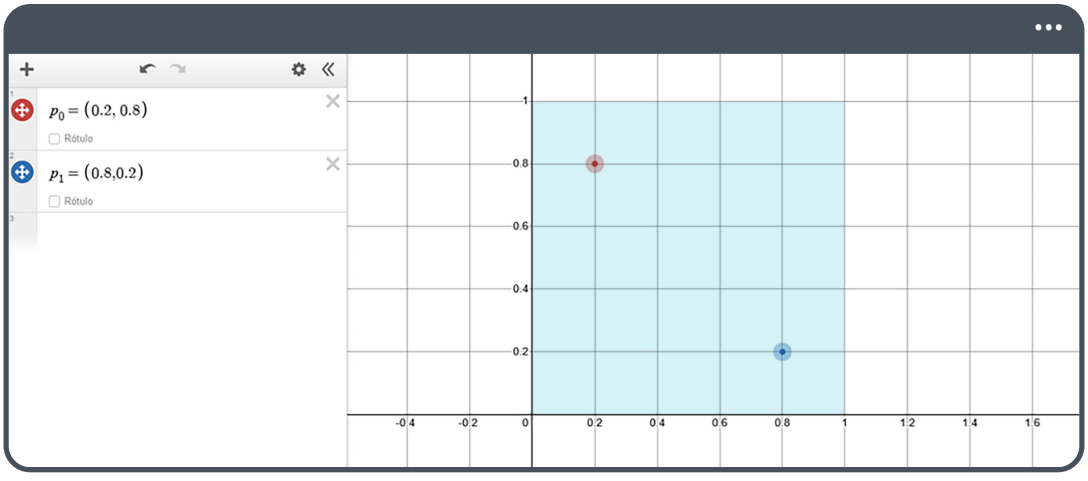
You can see from Figure 4.1.d, that the reference on the left represents the distance between random points and the segment, while the reference on the right shows the

resulting color when calculating that distance in HLSL. The closer the point is to the segment, the darker the corresponding pixel will appear. Conversely, the farther the point is from the segment, the lighter the pixel will be. Points outside the segment area return positive values, while points within the segment area return negative values. A value of zero indicates that the point lies exactly in the segment.

This technique avoids the issues that may arise when working with explicit functions, enabling the generation of complex shapes in both two and three dimensions. But how can we calculate a segment using distance? Similar to the explicit form, we must consider the following:

- The position of the first point.
- The position of the second point.
- The segment between the two points.

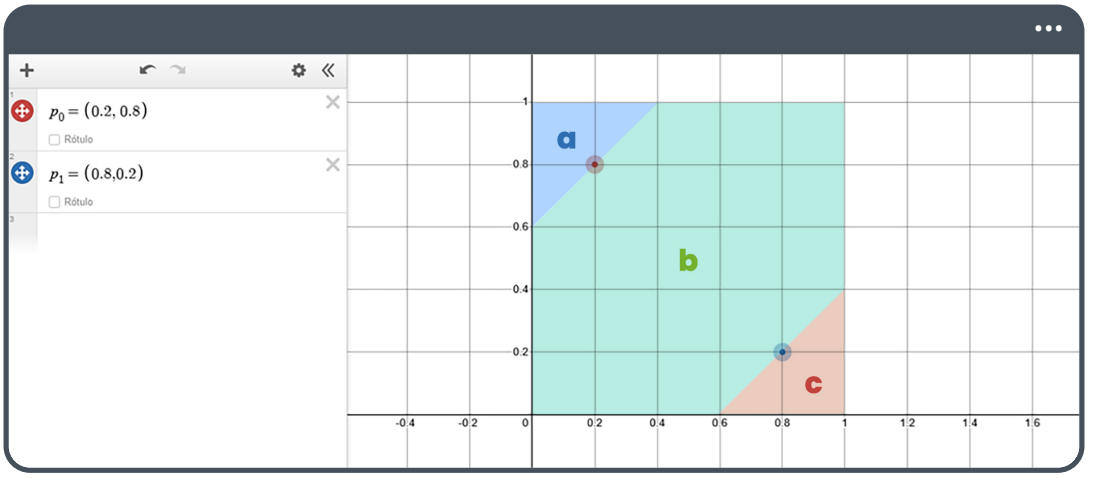
To better understand this exercise, you'll define two points, p_0 and p_1, on the Cartesian plane.



(4.1.e https://www.desmos.com/calculator/v0ihcwldiz)

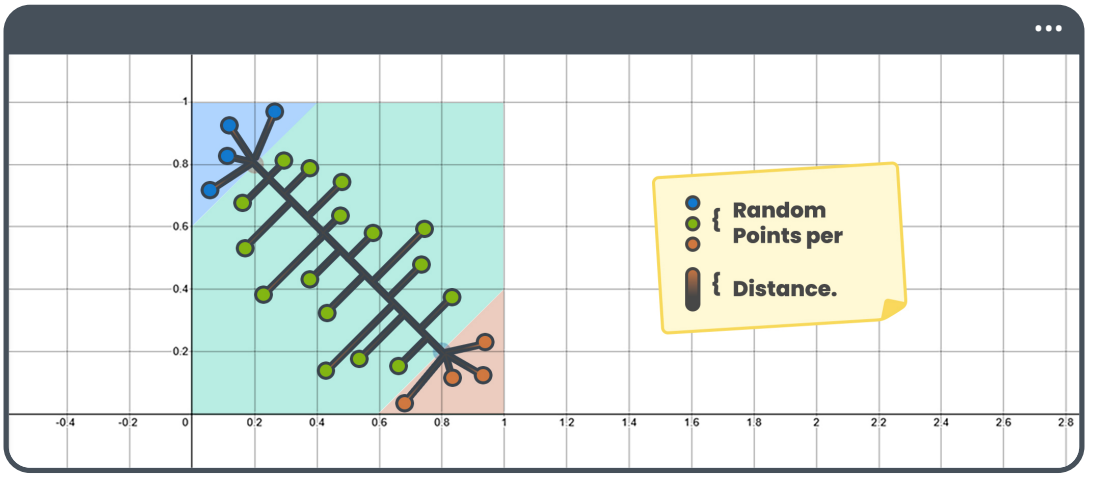
A key factor you need to consider when positioning the two points is that drawing a segment between them will result in an infinite line. Therefore, it's necessary to limit the segment at certain points to create a more compact and suitable shape for your project.

For example, by limiting the segment to the points defined above, three distinct areas will naturally emerge.



(4.1.f https://www.desmos.com/calculator/noz23qjhs5)

As you can see in Figure 4.1.f, we've labeled the three areas a, b, and c to differentiate them. Now, consider what would happen when calculating the distance between points in each area relative to the nearest point or segment. For instance, both area a and area c would generate semicircles, as all the points in these areas converge on a single point. Conversely, in area b, a segment would be formed that connects points p_0 and p_1.



(4.1.g)

In Figure 4.1.g, the colored circles represent "random" points within each area, while the gray lines indicate the distance of each point from p_0, p_1, and the segment. This raises the question: how can we determine these distances for each area? For instance, if we select any point $p_n = (0.1, 1.0)$ from area a relative to point $p_0 = (0.2, 0.8)$, we can calculate its distance d using the following function:

$$d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

(4.1.h)

Where the vector a of Equation 4.1.h represents a random point within the UV coordinates, vector b represents a defined point. Using the points p_n and p_0 in the scheme, we'll get the following result:

$$d = \sqrt{(0.1 - 0.2)^2 + (1.0 - 0.8)^2}$$

(4.1.i)

By performing the exercise, we can observe that:

$$d = \sqrt{0.01 + 0.04}$$

(4.1.j)

Which is equal to,

$$d = \sqrt{0.05}$$

(4.1.k)

Therefore,

$$d \approx 0.2236$$

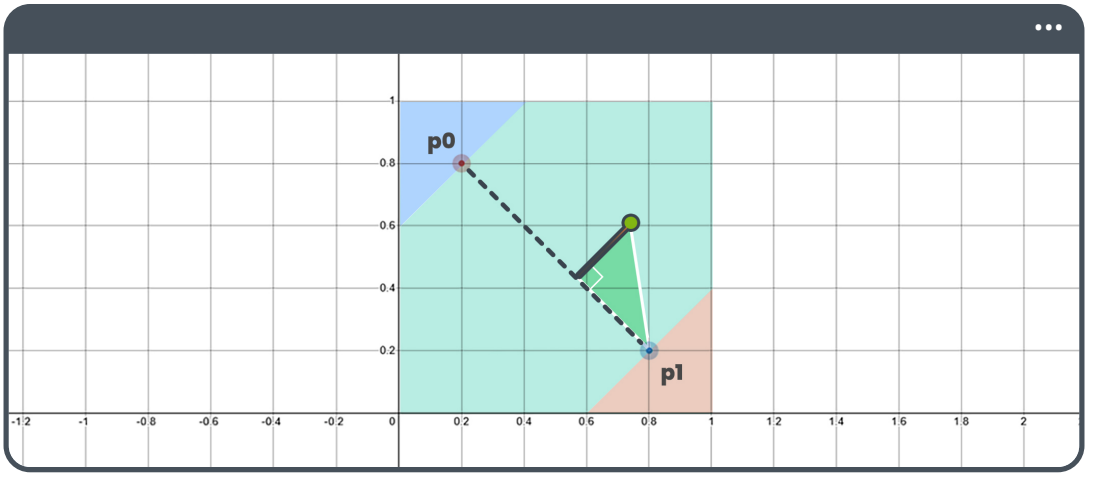
(4.1.l)

The result we got in the previous exercise (4.1.l) not only reflects the distance of one point from the other, but also the color of the resulting pixel on the screen in a grayscale. You should note that the different areas presented above correspond to sets of points. Therefore, implementing the distance function using UV coordinates would yield the following result:

$$d = \sqrt{(uv_x - p_x)^2 + (uv_y - p_y)^2}$$

(4.1.m)

Given that the segment has a second point, p_1, we can perform the same exercise to calculate the distance between the points in area c and this point. However, how would we go about calculating the segment itself? To achieve this, we must first define the segment in mathematical terms.



(4.1.n https://www.desmos.com/calculator/4jhmzqeflq)

The segment is the line that passes between points p_0 and p_1. If we examine the previous image closely, we can notice that calculating the distance between a random point and the segment naturally forms a right triangle. Consequently, we can use a projection to calculate all the points in group b that lie between the defined points.

We can use the following function to calculate this projection:

$$Proj = \frac{(a-b)\cdot(c-b)}{(\sqrt{c-b\cdot c-b})^2}$$

(4.1.o)

Where the variable a corresponds to the group of points by areas, that is, the UV coordinates, b corresponds to the first point p_0, and c is equal to p_1. Once we obtain the projection of the points located between the two mentioned points, we only need to calculate the distance between the points of each area with respect to a new point p_2, which can be determined as follows:

$$p_2 = b + Proj * (c - b)$$

(4.1.p)

Where again, b corresponds to the first point; and c, to the second.

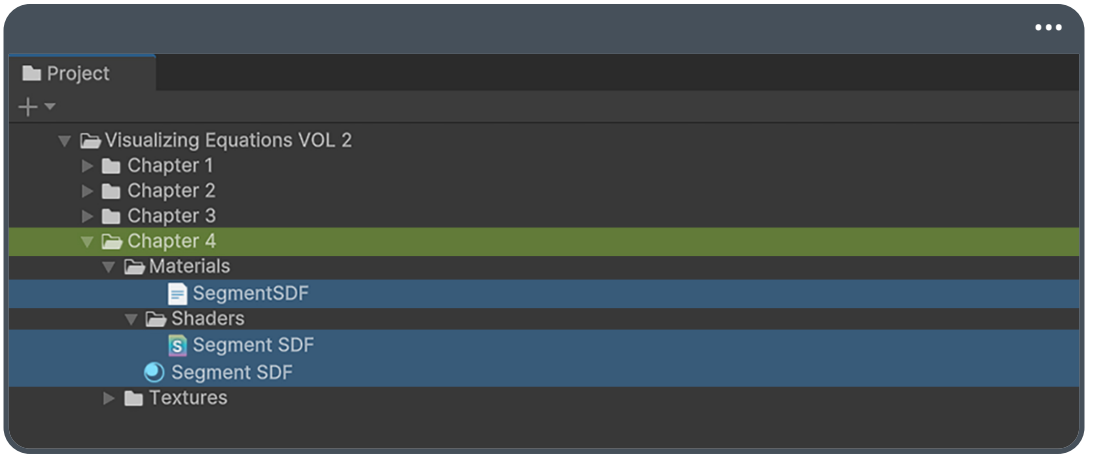
4.2 Drawing an SDF segment in the User Interface.

Up to this point, we've learned that creating a signed distance function requires calculating the distance between points in a region and a specific point or segment. In this section, we'll apply that knowledge to recreate the SDF segment explained earlier, but this time, we'll adapt it to a shader with UI support. This will allow us to integrate our effect into user interface elements, expanding its practical applications.

It's worth noting that the Unity version 6000.0.29f1, which we're currently using, includes support for UI shaders created with Shader Graph. However, for this exercise, we'll work with a general shader to explore the full workflow of shader creation and customization. This approach won't only help you understand fundamental concepts but also allow you to apply these techniques in a variety of contexts beyond UI elements. To get started, create a new shader of type **Unlit** in your project folder, and name it **Segment SDF**.

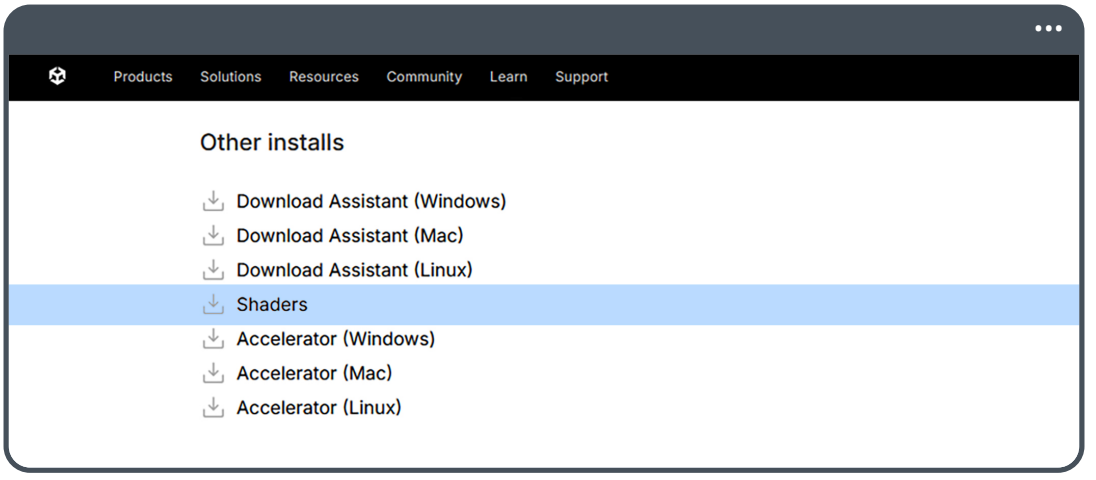
- Assets > Create > Shader > Unlit Shader.

Additionally, you need to create a material and a **.hlsl** file with the same name.



(4.2.a The project structure)

The shader you've just created has a **.shader** extension, which differs from the **.shadergraph** files used earlier in the project. By default, this type of shader doesn't include UI support, so you'll need to add a few extra lines of code to ensure it works correctly with the version of Unity you are using. To simplify the learning process and keep the focus on the essentials, download the **UI-Default** shader from the **.zip** file included with Unity. This shader will serve as a foundation for understanding and efficiently building the desired effect.



(4.2.b https://unity.com/es/releases/editor/archive)

Once you've downloaded this file you can find the shader following the path:

- builtin_shaders-[version] > build > BuiltinShaders > builtin_shaders > DefaultResourcesExtra > UI > UI-Default.shader.

The only step left is for you to copy all the code that is inside the shader field itself, specifically the **Properties** and the **SubShader**, and paste it into the newly created shader. If everything has gone well, the properties of the shader should look like this:

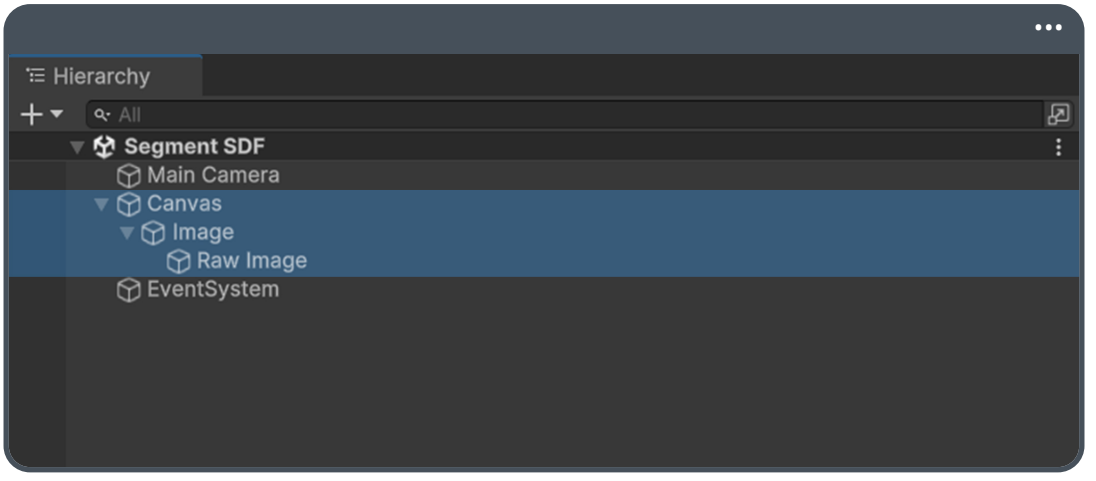
```
Shader "Jettelly/Chapter 4/UI/Segment SDF"
{
    Properties
    {
        [PerRendererData] _MainTex ("Sprite Texture", 2D) = "white" {}
        _Color ("Tint", Color) = (1, 1, 1, 1)

        _StencilComp ("Stencil Comparison", Float) = 8
        _Stencil ("Stencil ID", Float) = 0
        _StencilOp ("Stencil Operation", Float) = 0
        _StencilWriteMask ("Stencil Write Mask", Float) = 255
        _StencilReadMask ("Stencil Read Mask", Float) = 255

        _ColorMask ("Color Mask", Float) = 15
        [Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip ("Use Alpha Clip",
        ↪ Float) = 0
    }

    SubShader { … }
}
```

The code also includes other useful functions that define the behavior of the shader when applied to UI images. However, these won't be covered, as the focus of this chapter is on explaining signed distance functions.

Before you start to implement functions in your script, make a small configuration in the **Hierarchy** window, to include the following elements:

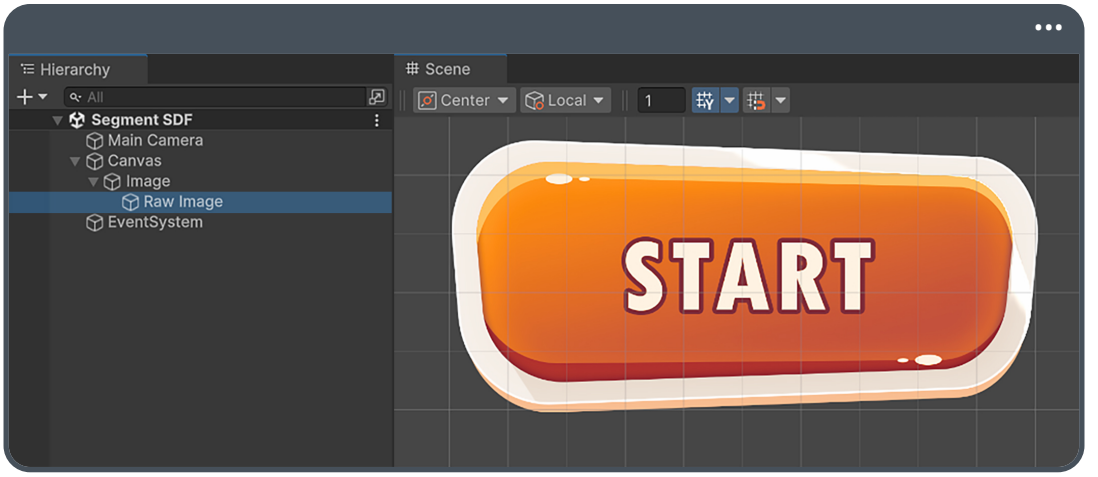
- A Canvas.
- An Image.
- A Raw Image.

You need to group these elements as follows:



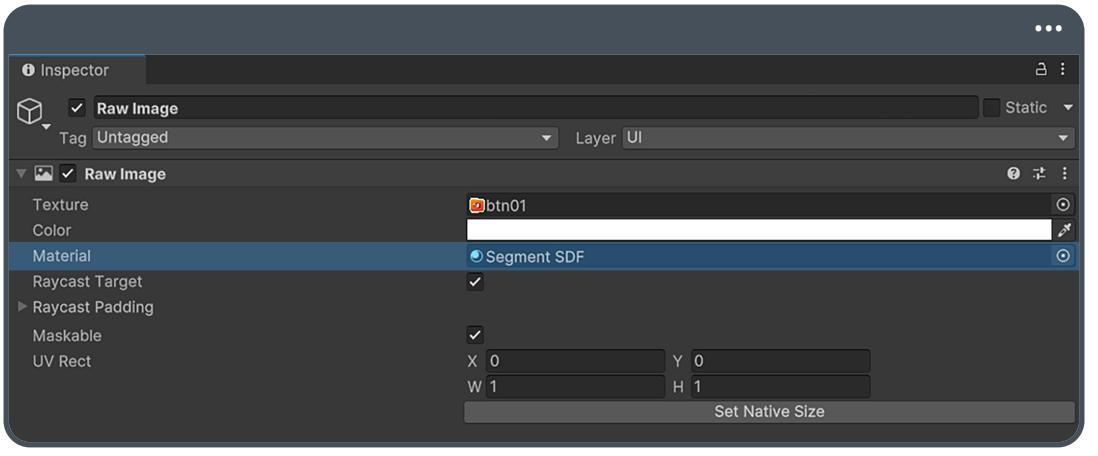
(4.2.c A Canvas on the Hierarchy window)

It's important to note that you will apply the **Segment SDF** material directly to the **Raw Image** component, leaving the **Image** component unused. Why is this necessary? It comes down to the UV coordinate settings. The **Image** component is compatible with **Atlas** (or Spritesheet), meaning it can be positioned within a set of images, with its coordinates adjusting to the new position within the set. In contrast, **Raw Image** doesn't have this compatibility, as it's designed for general textures, making it ideal for visual effects in UI. In this scenario, **Raw Image** will function as a second layer, which can be additive, multiplicative, or use any **Blending Mode** desired for your effect, while preserving the original margins of the image.



(4.2.d)

Therefore, you must make sure to include the **Segment SDF** shader in its respective material. Then, assign this material to the **Material** property of the Raw Image component, as shown below:



(4.2.e Custom material in the Material property)

In the book package you'll find a section containing license-free images created by the Jettelly team for the texture or UI image that we use for this exercise and others throughout this book. Regardless of which image you choose to work with, it's necessary to include the selected image in both the **Image** and **Raw Image** components to preserve the edges for applying the effect later.

Once you've configured the user interface, ensure that the **.hlsl** script is included in the **.shader** you're working on. To do this, add the following line of code:

```
#include "UnityCG.cginc"
#include "UnityUI.cginc"
#include "Assets/Jettelly Books/… /SegmentSDF.hlsl"
```

Although you haven't added any code to the HLSL script yet, the **`#include`** directive allows you to use functions directly from it. Therefore, the next step is to add the functions needed to represent the signed distance segment. Begin by implementing the projection detailed in the previous section, Equation 4.1.o.

While it might seem more linear to start by defining the distance function between two points, HLSL already includes a function that performs this operation, which looks like this:

```
float distance(float2 pt1, float2 pt2)
{
    float2 v = pt2 - pt1;
    return sqrt(dot(v, v));
}
```

Which is the same as saying,

```
float distance(float2 p, float2 uv)
{
    return sqrt(pow(uv.x - p.x, 2.0) + pow(uv.y - p.y, 2.0));
}
```

The latter is much more similar to the definition of the function presented in the previous section, Equation 4.1.m. Begin by implementing the projection function in HLSL, considering the following variables:

- The variable **a** (UV coordinates).
- The variable **b** (first point).
- The variable **c** (second point).

Then you must declare the following method in **SegmentSDF.hlsl**:

```
float projection(float2 a, float2 b, float2 c)
{
    return 0;
}
```

Following Equation 4.1.o, you can observe that there are two subtractions present: $a - b$ and $c - b$. Therefore, the first step is to declare and initialize these two-dimensional vectors within the **`projection()`** method.

```
float projection(float2 a, float2 b, float2 c)
{
    float2 cb = c - b;
    float2 ab = a - b;

    return 0;
}
```

Next, return the body of the function, which calculates the dot product between the vectors **cb** and **ab**, divided by the squared magnitude of the first vector. That is to say:

```
1  float projection(float2 a, float2 b, float2 c)
2  {
3      float2 cb = c - b;
4      float2 ab = a - b;
5
6      return dot(ab, cb) / pow(length(cb), 2.0);
7  }
```

If you refer to the previous exercise in line 6, you can observe that the second operation in the function corresponds to the square of the length of the vector **cb**. However, it's important to note that,

$$length(cb)^2 = dot(cb, cb)$$

(4.2.f)

This is because the **`length()`** function involves a square root, and as you already know, squaring a square root eliminates it, leaving the result isolated in the operation.

```
1  float length(float3 v)
2  {
3      return sqrt(dot(v, v));
4  }
```

So, you can simplify the return value of the **`projection()`** method as follows:

```
float projection(float2 a, float2 b, float2 c)
{
    float2 cb = c - b;
    float2 ab = a - b;

    return dot(ab, cb) / dot(cb, cb);
}
```

As you can see from the previous section, the projection itself only generates an infinite segment that will pass between the two points mentioned above. Therefore, you'll need to limit the segment and define a second point p_2, to determine the distance between the UV coordinates and the segment itself.

You'll continue by declaring a new empty method which you'll call **`segment_sd_float()`**. This method will limit the segment and determine the position of all the points between p_0 and p_1.

```
void segment_sd_float(in float2 uv, in float2 p0, in float2 p1, out float4
    Out)
{
    float h = projection(uv, p0, p1);
    h = clamp(h, 0.0, 1.0);

    Out = 0;
}
```

As you can see from the previous example, line 11 has declared a new variable named **h**, which includes the projection between the UV coordinates and points **p0** and **p1**. Subsequently, the value of **h** has been limited to a range between [0.0 : 1.0], corresponding to the distance between the two mentioned points, to avoid infinite values.

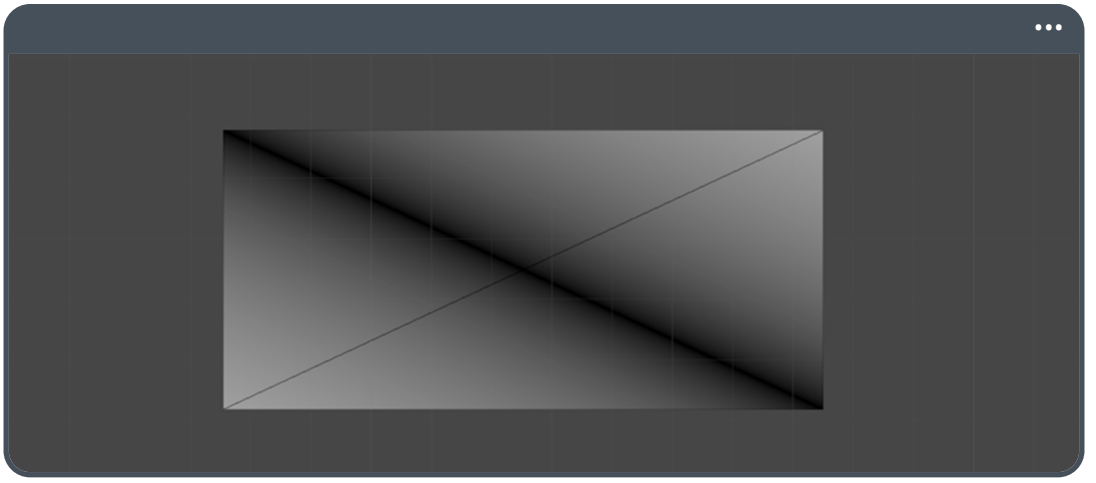
The final action you need to do is calculate the distance between the UV coordinates and all the points generated between **p0** and **p1**. For this, you can use Equation 4.1.p, described in the previous section, as follows:

```
9  void segment_sd_float(in float2 uv, in float2 p0, in float2 p1, out float4
   ↪ Out)
10 {
11     float h = projection(uv, p0, p1);
12     h = clamp(h, 0.0, 1.0);
13     float2 p2 = p0 + h * (p1 - p0);
14
15     Out = distance(p2, uv);
16 }
```

You could conclude at this point that the signed distance segment is ready. Now, you simply need to visualize it. To do this, go to your **Segment SDF** shader and add the following line of code in the fragment stage **frag()**, as shown below:

```
126 half4 color = IN.color * (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd);
127
128 float4 segment = 0;
129 segment_sd_float(IN.texcoord, float2(0.0, 1.0), float2(1.0, 0.0), segment);
130
131 #ifdef UNITY_UI_CLIP_RECT
```

As you can see, line 128 has declared a new four-dimensional vector called **segment** and used it as the output in the **segment_sd_float()** function. The following line includes two constant points with values [0.0, 1.0] for the first one, and [1.0, 0.0] for the second. If you use the **segment** vector as the return value for each pixel it'll yield the following graphical result:



(4.2.g)

Note that in the previous image, the shader's **Blending Mode** was removed to better visualize the segment. As you can see, the segment is correctly projected from a mathematical perspective. However, you need to add some properties to the shader to adjust its behavior and appearance. To do this, follow these steps:

Within the properties field:

- Add a four-dimensional vector, where the first two **xy** components correspond to the first point, and the remaining components **zw** correspond to the second point.
- Next, include a variable to define the radius of the segment.
- Finally, declare a variable to control the smoothness of the segment edges.

```
6  _Color ("Tint", Color) = (1,1,1,1)
7  _Points ("Points", Vector) = (0.0, 1.0, 1.0, 0.0)
8  _Radius ("Radius", Range(0.01, 0.5)) = 0.1
9  _Smooth ("Smooth", Range(0.01, 0.5)) = 0.01
```

The **_Color** property is included in the shader by default; however, **_Points**, **_Radius**, and **_Smooth** have been declared to define dynamic values within the shader. It's important to note that, similarly, you must declare these properties within the **Pass**, along with their respective data type, as follows:

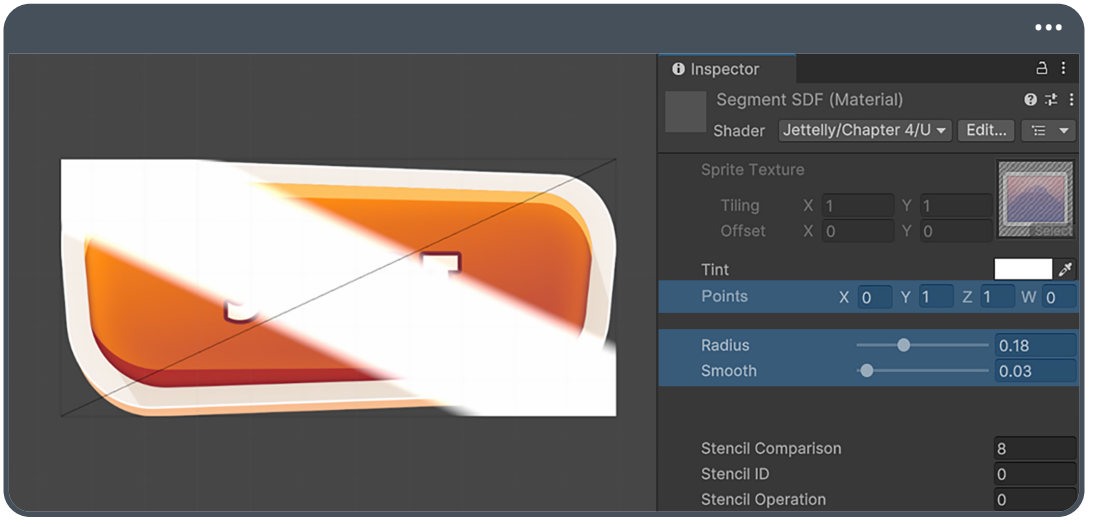
```
89  int _UIVertexColorAlwaysGammaSpace;
90  float4 _Points;
91  float _Radius;
92  float _Smooth;
```

Once declared, you can use these properties to define the segment.

```
132 float4 segment = 0;
133 float2 p0 = _Points.xy;
134 float2 p1 = _Points.zw;
135 segment_sd_float(IN.texcoord, p0, p1 segment);
136 float s = _Smooth;
137 float r = _Radius;
138 segment = smoothstep(segment - s, segment + s, r);
```

As you can see in the previous example, two new 2D vectors, **p0** and **p1**, have been declared and initialized using the components of the vector **_Points** (lines 133 and 134). Subsequently, these two points have replaced the constant points in the **segment_sd_float()** method, which was previously included as an example. This process allows you to modify the segment's orientation by adjusting the points' positions directly in the Inspector. Then, two new variables, **s** and **r** (lines 136 and 137) have been declared, where **s** controls the smoothing of the segment's edges and **r** determines its thickness.

After saving the changes, you can return to Unity to modify the segment body.



(4.2.h)

Something you may notice from the current implementation is that the segment's edges extend beyond the boundaries of the UI image. Therefore, you need to limit its graphic representation. You can do this easily by multiplying the alpha channel of the color input by the segment, as shown below:

```
149 // color.rgb *= color.a;
150 segment.rgb *= IN.color;
151 segment.rgb *= color.a;
152
153 return segment;
```

It's important to remember that the alpha channel represents an image's transparency or opacity. Therefore, when multiplying the A component by the segment, you get a completely grayscale texture, where the range of [1.0 : 0.0] represents the intensity of each pixel. In this context, black corresponds to 100% transparency, while white corresponds to 0% transparency.

(4.2.i)

After saving the changes and returning to Unity, you can see that the segment remains within the image area regardless of the points' positions.

4.3 Analysis of an SDF Pentagon Structure.

Continuing with our study of signed distance functions, this section examines a method for drawing a pentagon in UV coordinates, developed by Inigo Quilez, who, at the time of this book's publication, is a Principal Research Engineer at Adobe, specializing in computer graphics.

To begin, let's look at the following code snippet, written in OpenGL Shading Language (GLSL):

```
float sdPentagon( in vec2 p, in float r )
{
    const vec3 k = vec3(0.809016994, 0.587785252, 0.726542528);
    p.y = -p.y;
    p.x = abs(p.x);
    p -= 2.0 * min(dot(vec2(-k.x ,k.y), p), 0.0) * vec2(-k.x, k.y);
    p -= 2.0 * min(dot(vec2( k.x, k.y), p), 0.0) * vec2( k.x, k.y);
    p -= vec2(clamp(p.x, -r * k.z, r * k.z),r);
    return length(p) * sign(p.y);
}
```

The complete code for this example can be found at the following link:

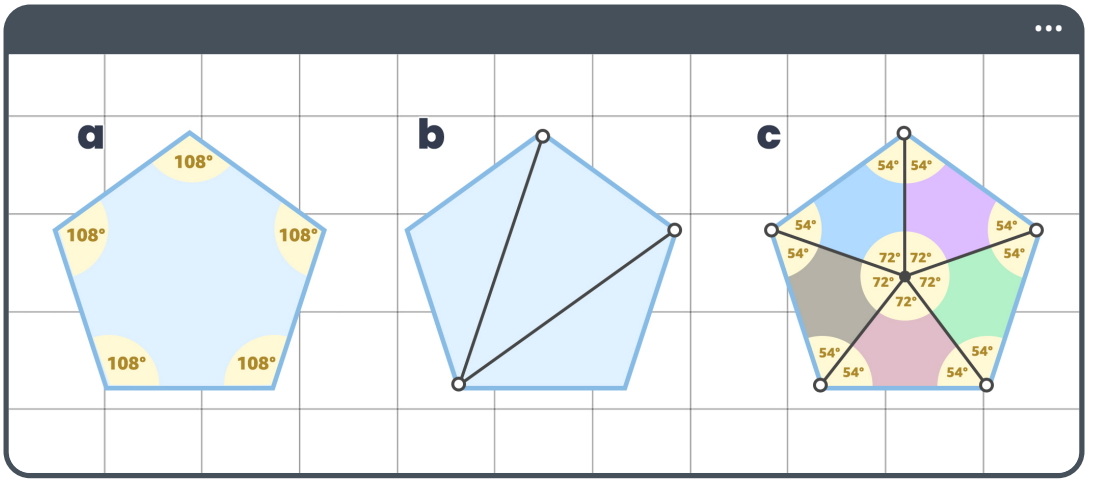
https://www.shadertoy.com/view/llVyWW

It's important to highlight that there's no significant difference between GLSL and HLSL in terms of data types. For example, in line 3, the GLSL three-dimensional vector **vec3** corresponds to **float3** in HLSL. The same applies to the two-dimensional vector **vec2** in lines 6, 7, and 8. Therefore, the equivalent code in Unity would look like this:

```
void pentagon_sd_float(in float2 uv, in float r, out float4 Out)
{
    const float3 k = float3(0.809016994, 0.587785252, 0.726542528);
    uv -= 0.5;
    uv.y = -uv.y;
    uv.x = abs(uv.x);
    uv -= 2.0 * min(dot(float2(-k.x, k.y), uv), 0.0) * float2(-k.x, k.y);
    uv -= 2.0 * min(dot(float2( k.x, k.y), uv), 0.0) * float2( k.x, k.y);
    uv -= float2(clamp(uv.x,-r * k.z, r * k.z),r);
    Out = length(uv) * sign(uv.y);
}
```

In this example, some naming conventions have been adjusted to improve code legibility. Additionally, the method has been set to **`void`** to ensure compatibility with the **Custom Function** node. In line 4, 0.5 is subtracted from the UV coordinates to center the shape on a Quad. You may ask: what exactly happens within this method? To answer this, it's first essential to understand how a pentagon is formed from a trigonometric perspective.

A regular pentagon is a geometric figure with five sides of equal length and equal interior angles. In other words,



(4.3.a)

The internal angles of a regular pentagon each measure 108°. This can be easily determined using triangles. According to a fundamental principle of Euclidean geometry, the internal angles of a triangle always sum to 180°. If you examine Reference **B** in Figure 4.3.a, you'll notice that a pentagon can be divided into three internal triangles. Therefore:

$$180° * 3 = 540°$$

(4.3.b)

If you divide the result of this multiplication by the five sides of the pentagon, you'll get the angle corresponding to each side.

$$\frac{540°}{5} = 108°$$

(4.3.c)

From another perspective, if you refer to Reference **C** in Figure 4.3.a, and divide a complete circle into five equal parts, you get the following result:

$$\frac{360°}{5} = 72°$$

(4.3.d)

Additionally, you can divide the internal angle to get:

$$\frac{108°}{2} = 54°$$

(4.3.e)

These angles are crucial for constructing a pentagon, as they precisely determine the measurements needed to draw its sides and define its vertices. Knowing that each internal angle measures 108° ensures that all the intersections are correctly aligned, preserving the pentagon's regular shape.

Calculating the 72° angle in Reference **C** is essential to understand the relationship between the internal and external angles of the pentagon. Each external angle, formed by extending the pentagon's sides, measures 72°. When added to the internal angle of 108°, the total is 180°.

For instance, line 3 of the **`sdPentagon()`** function declares a three-dimensional vector named **k**, containing the constants [0.809016994, 0.587785252, 0.726542528]. These values, expressed in radians, are derived from the following mathematical calculation:

$$\frac{\pi}{5} = 0.628318530 = 36°$$

(4.3.f)

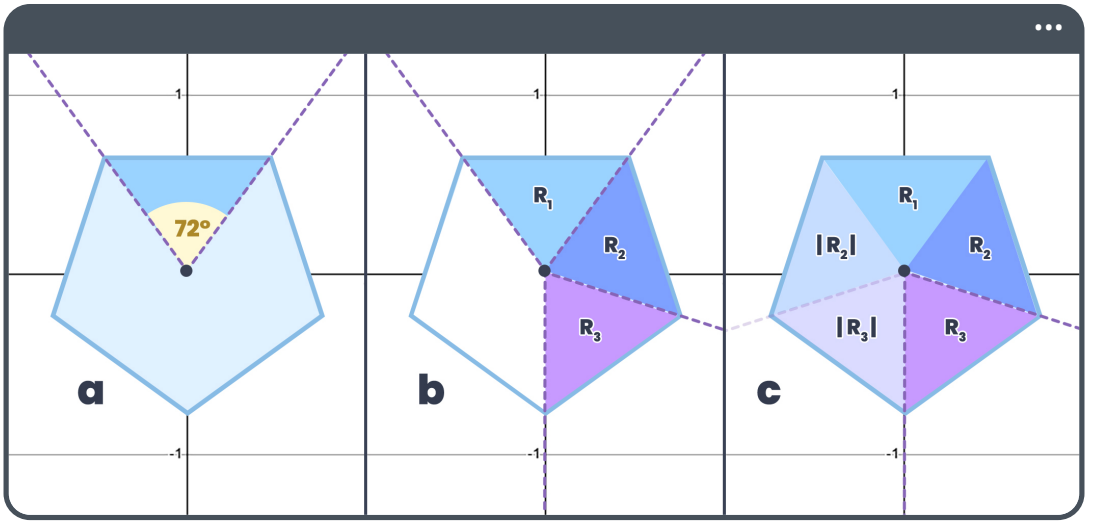
Where,

$$k.x = cos(0.628318530) = 0.809016994$$
$$k.y = sin(0.628318530) = 0.587785282$$
$$k.z = tan(0.628318530) = 0.726654252$$

(4.3.g)

You'll notice in Figure 4.3.f that 36° corresponds to half of 72°. But, how can you determine the relevance of $\frac{\pi}{5}$ within the context of an SDF pentagon? To answer this, recall the reflections discussed in Section 2.3 of Chapter 2.

Since the pentagon has five equal sides, you don't need to calculate its full shape. Instead, you can define a single region and then apply transformations. Considering only one region of the full shape, how can you calculate the distance between all points outside the pentagon and its edge? To answer this, refer to the following reference:



(4.3.h https://www.desmos.com/calculator/1jvqip75ke)

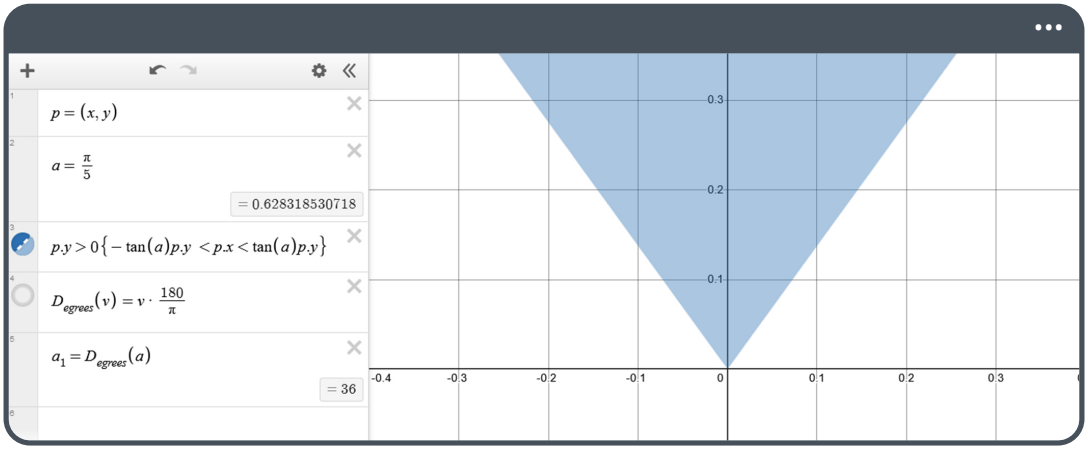
Each figure in the image above represents a necessary step in constructing the **`sdPentagon()`** method, which is divided into three regions R_1, R_2, and R_3. The first region, R_1, serves as a reference area for calculating the remaining regions through reflection and transformation. Therefore, we'll focus on directly calculating the distance to the pentagon in this first region.

If you consider $a = \frac{\pi}{5}$ and based on the method's code, you can determine that all points p (or UV coordinates) in the first region lie within a range defined by the following condition:

$$p.y > 0 \{-tan(a)p.y \leq p.x \leq tan(a)p.y\}$$

(4.3.i)

You can visualize Equation 4.3.k in the Cartesian plane as follows:



(4.3.j https://www.desmos.com/calculator/ypkrhldiwh)

From the previous exercise, you can begin to identify the variables included in the **`sdPentagon()`** method. For example:

$$p = in\ vec2\ p$$

(4.3.k)

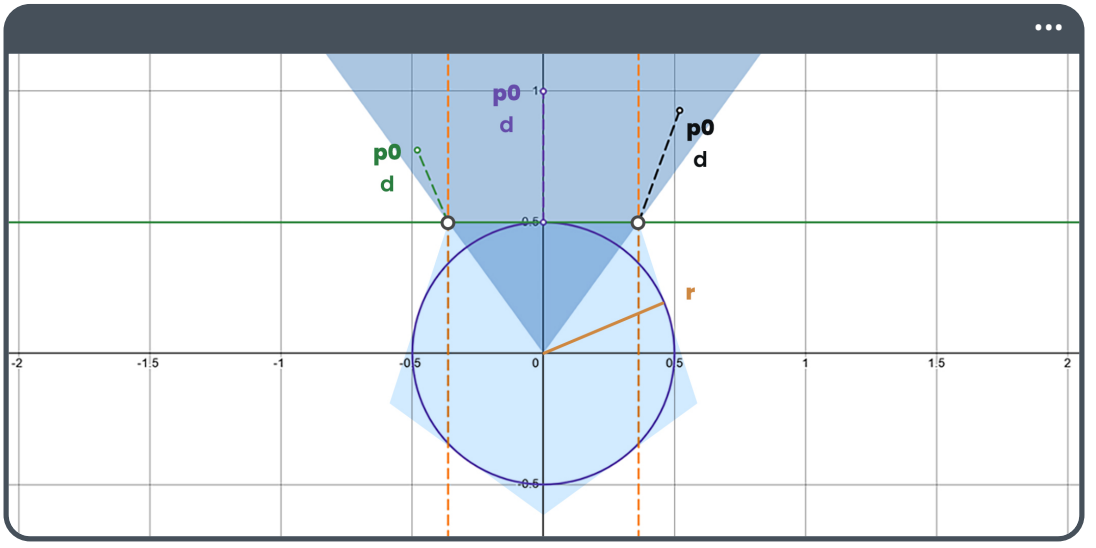
Or also:

$$tan(a) = k.z$$

(4.3.l)

Equation 4.3.k defines the boundaries of the first region, R_1. However, the distance between all the points in this region and the pentagon's edge hasn't been calculated yet. To determine this distance, you must first establish how to perform this calculation. Therefore, you first need to define:

- A radius r to determine the size of the pentagon.
- The distance d between all points in region R_1 and the vertices closest to each point.



(4.3.m https://www.desmos.com/calculator/vkf65m4iz5)

When determining the distance between a vertex (as shown in Figure 4.3.m, white dots) and its closest point, three distinct zones naturally emerge, each requiring different parameters or arguments. For instance, in the case where $p_x > r\ tan(a)$, you can calculate the distance by using the following values: Considering a random point $p_n = (0.6,\ 1.0)$ and its nearest vertex $v_0 = (r\ tan(a), r)$, you get:

$$d = \sqrt{(0.6 - r\ tan(a))^2 + (1.0 - r)^2}$$

(4.3.n)

This is equivalent to saying:

$$d = ||p_n - v_0||$$

(4.3.o)

This is a more concise way to express the Euclidean distance between two points.

For the case where $p_x < r\ tan(a)$, considering a random point $p_n = (-0.5,\ 0.7)$ and its nearest vertex $v_1 = (-r\ tan(a), r)$ the distance is calculated as:

$$d = \sqrt{(-0.5 + r\ tan(a))^2 + (0.7 - r)^2}$$

(4.3.p)

You must also consider cases where $p_x < 0$, since points p_n closer to the center will eventually be nearer to the right vertex v_0 or the left vertex v_1, especially when the radius approaches zero. If you examine line 8 of the **`sdPentagon()`** method, you'll notice that it uses the **`clamp()`** function, which limits the range of a variable (in this case, p_x) to a specific value, resulting in the following expression:

$$d = ||p - (max(-r\,tan(a), min(r\,tan(a), p.x)), r)||$$

(4.3.q)

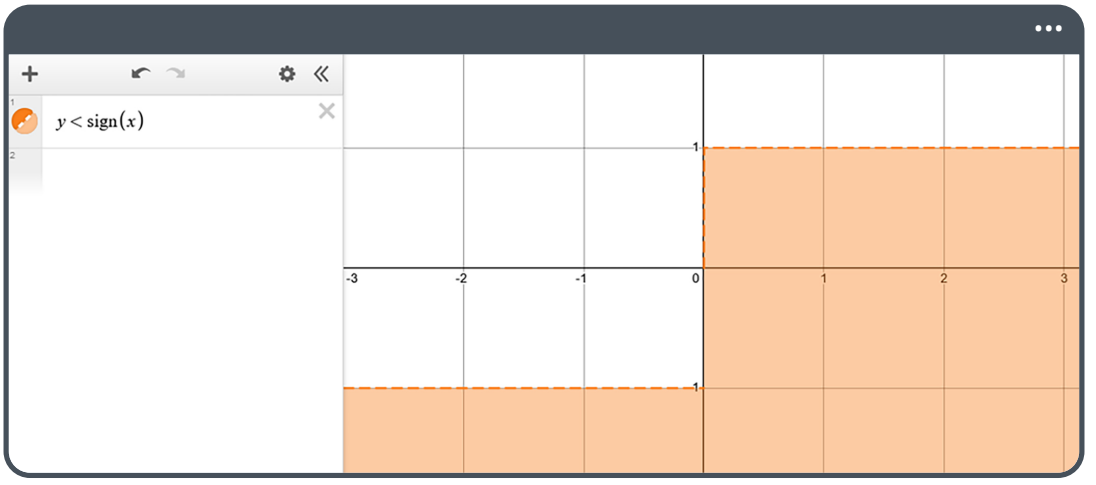
This translates to the following code:

```
p -= vec2( clamp( p.x, -r * k.z, r * k.z), r);
return length(p) * sign(p.y);
```

Given that,

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

Subsequently, the **`sign()`** function is added to the method to return 1, -1 or 0, depending on whether the value of the y coordinate is greater than, less than, or equal to zero, respectively. In the **`sdPentagon()`** method, this function is used to determine whether the points are inside or outside the pentagon's body.

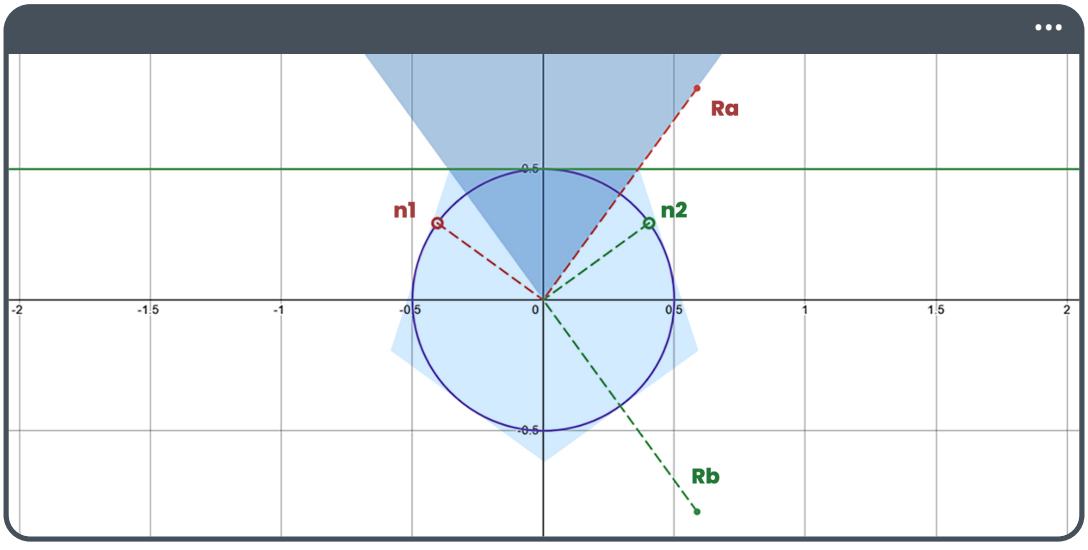


(4.3.r https://www.desmos.com/calculator/hiur1ldkgd)

This allows you to precisely define the interior and exterior areas of the pentagon, facilitating the creation of the shape by basing it on the proximity of the edges and the relative orientation of the points with respect to the figure's center.

4.4 Reflections and Transformations of the SDF Pentagon.

Up to this point, we've explored the mathematical functions involved in the implementation of the first region R_1 of the pentagon in **`sdPentagon()`**. Next, you need to define the regions R_2 and R_3 to complete the figure. To do this, you must establish the axes of reflection.



(4.4.a https://www.desmos.com/calculator/rutjfowcsl)

As shown in Figure 4.4.a, two reflection axes, R_a and R_b, have been defined, along with two normals, n_1 and n_2, which are perpendicular to their respective reflection axes. The axis R_a has been determined by calculating the sine and cosine of the angle a for each component, as follows:

$$R_a = (sin(a), cos(a))$$

(4.4.b)

While its normal is defined as:

$$n_1 = (-cos(a), sin(a)) = (-k.x, k.y)$$

(4.4.c)

Similarly, the axis R_b is defined as:

$$R_b = (sin(a), -cos(a))$$

(4.4.d)

And its normal is:

$$n_2 = (cos(a), sin(a)) = (k.x, k.y)$$

(4.4.e)

It's important to note that these reflections only allow us to calculate the regions on the positive side of the **`p.x`** coordinate. Therefore, when the values of **`p.x`** are negative, we must apply the absolute value to this coordinate to obtain the complete body of the pentagon, as shown in line 5 of the **`sdPentagon()`** method.

Next, you can transform the points, starting with the second region, R_2. This region requires only a reflection on R_a. Thus, for points belonging to R_2, the transformation formula is:

$$p' = p - 2(n_1 \cdot p)n_1$$

(4.4.f)

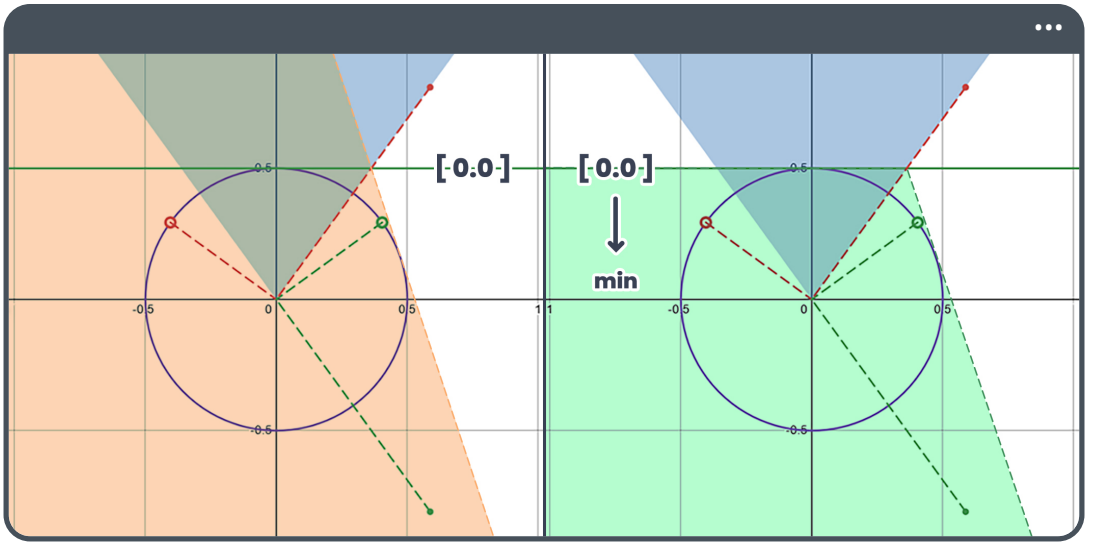
Which translates into the following code:

```
p -= 2.0 * dot(vec2(-k.x, k.y), p) * vec2(-k.x, k.y);
```

However, when you look at the original method, note that the **`min()`** function has been included. This is because, reflection is infinite, and when you reflect in R_a, some points fall outside the pentagon's body. Therefore, you must limit this reflection according to your desired values.

Since the edge of the first region is defined explicitly by $y = r$, the y coordinate serves as the boundary value for reflection, which is 0.0.

```
p -= 2.0 * min(dot(vec2(-k.x, k.y), p), 0.0) * vec2(-k.x, k.y);
```



(4.4.g https://www.desmos.com/calculator/syycpni4gc)

The same problem arises when calculating the third region R_3 around the second reflection.

$$p'' = p' - 2(n_2 \cdot p')n_2$$

(4.4.h)

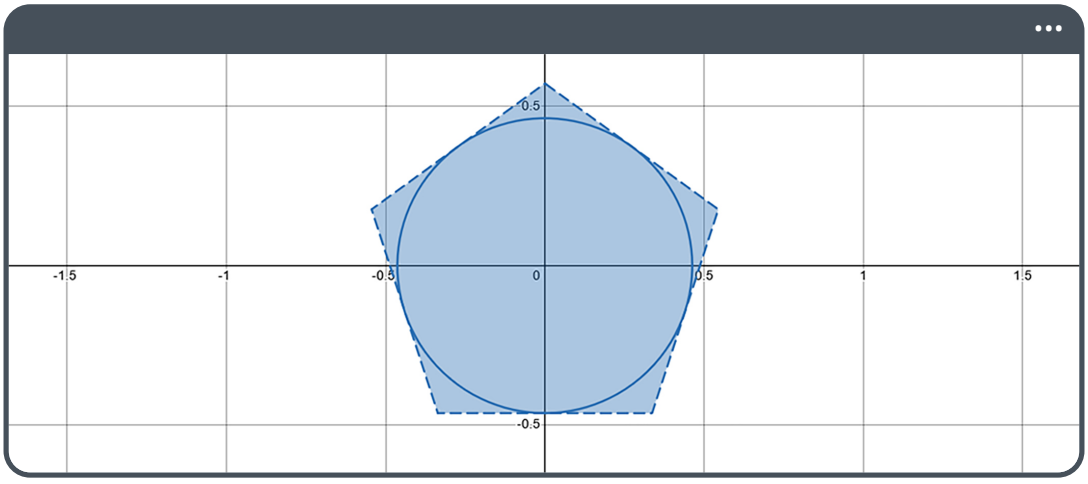
Which translates to the following code:

```
p -= 2.0 * dot(vec2(k.x, k.y), p) * vec2(k.x, k.y);
```

If the transformation is infinite, you'll also need to limit it at certain points; otherwise, you would be calculating the distance of points outside the pentagon's body. To achieve this, you can simply apply the **`min()`** function to the expression, resulting in the following:

```
7  p -= 2.0 * min(dot(vec2(k.x, k.y), p), 0.0) * vec2(k.x, k.y);
```

At this point, you might consider the shape complete. However, the previous calculations only apply to points where $p_x > 0$. Therefore, as indicated in line 5 of the **`sdPentagon()`** method, you must ensure that the absolute value is applied to **`p.x`**. Additionally, you can flip the figure by making the **`p.y`** coordinate negative.



(4.4.i https://www.desmos.com/calculator/mve8t0iuxn)

In conclusion, after analyzing each mathematical function in the **`sdPentagon()`** method, you can determine the following:

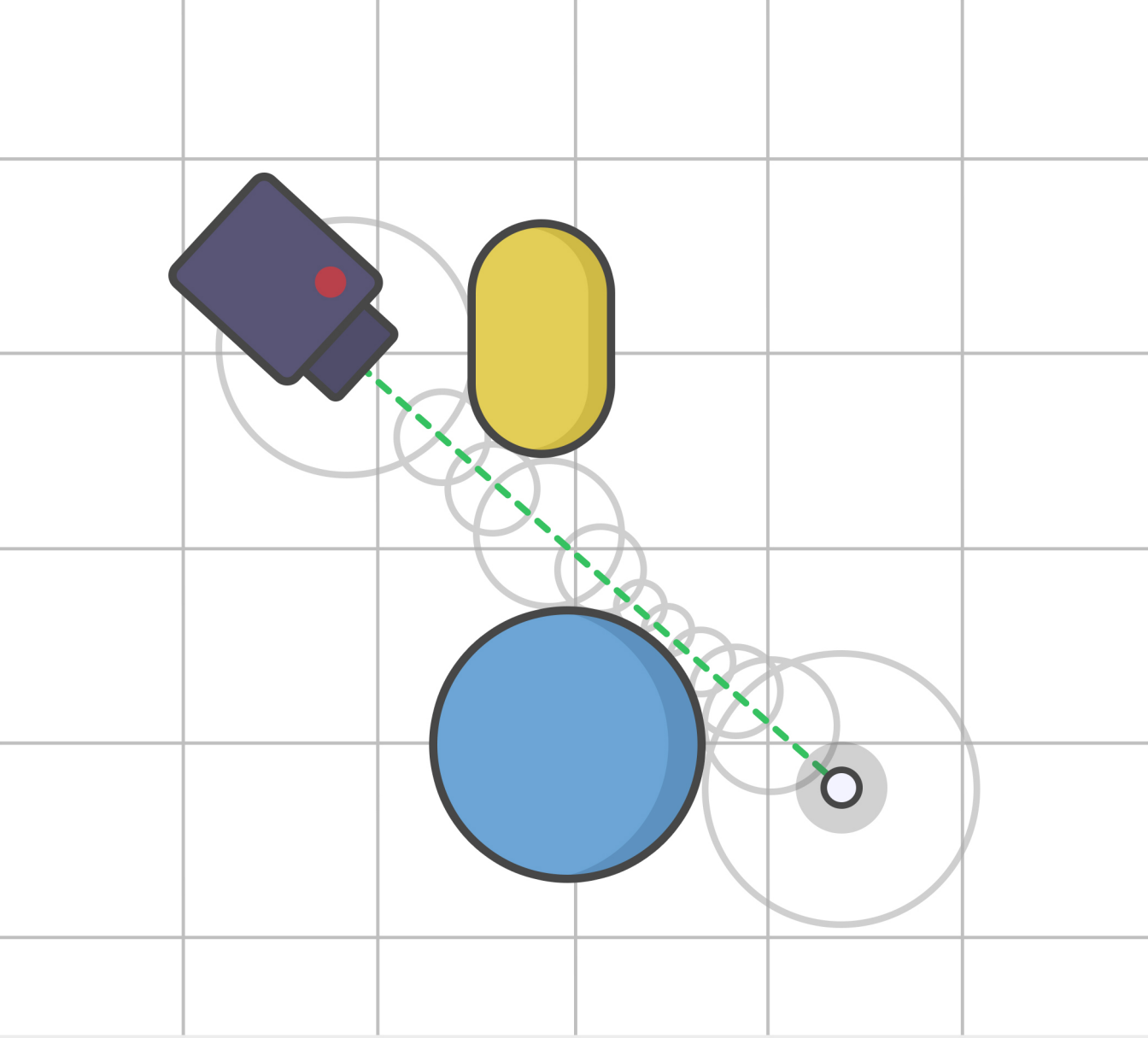
- Line 1: The arguments **`p`** and **`r`** refer to the UV coordinates and the radius respectively.
- Line 3: The constant **`k`**, a three-dimensional vector, contains the trigonometric calculations of cos, sin and tan for each component of the angle.

- Line 4: Inverts the **`p.y`** component around point **`p`** so that the pentagon faces upward.
- Line 5: Uses the absolute value of **`p.x`** to reflect the points when $p_x < 0$.
- Line 6: Defines implicitly the second region R_2 through reflection.
- Line 7: Defines implicitly the third region R_3 through a second reflection.
- Line 8: Explicitly delimits the first region R_1 using the **`clamp()`** function.
- Line 9: Calculates the distance between all points outside the pentagon and its edge, determining whether they are inside or outside the figure.

Chapter Summary.

- In this chapter we explored the construction of a segment and a pentagon using signed distance functions (SDF) in the context of computational graphics. We began by analyzing the mathematical structure of a segment, comparing it to a linear function of the type $mx + b$, and applying this knowledge to generate a UI image effect in Unity, which is useful for production.

- In the second section, we reviewed the functions associated with the pentagon's body. We analyzed the mathematical structure of the first region R_1, calculating the distance from internal points to the figure's edges. Then, we defined the R_2 and R_3 regions through reflections along the R_a and R_b axes, using their normals to adjust the coordinates of the reflected points. To ensure accuracy, we constrained these reflections using the **`min()`** function to prevent distance calculations outside the pentagon. Finally, we applied the absolute value to the **`p.x`** coordinate, and adjusted the **`p.y`** coordinate to obtain the complete shape and orientation of the pentagon.



Chapter 5

3D Shapes and Rendering.

In this chapter, we'll explore the use of signed distance functions to generate more complex three-dimensional geometries, focusing on creating shapes such as spheres and capsules. These shapes, known as primitives, form the foundation of most 3D models and will help you better understand the mathematical operations required to design your own figures. Additionally, we'll examine methods for combining and modifying these primitives, enabling you to sculpt more intricate and customized shapes.

You'll also learn how to calculate normals in geometries defined by signed distance functions, a fundamental aspect of lighting. Obtaining accurate normals is essential for achieving realistic lighting effects and enhancing the credibility of your projects. Additionally, we'll introduce the Ray Marching method, a crucial technique for efficiently rendering these structures.

Throughout this chapter, we'll work with practical examples and implementations in Unity, emphasizing the importance of mathematics at each step. Our goal isn't only to enhance your projects within this engine but also to help you understand how to apply these concepts across different platforms and graphical programming contexts. By the end of this chapter, you'll have the skills needed to design and develop highly complex procedural shapes, leveraging the synergy between mathematics and graphical programming to take your visual effects to a new creative level.

5.1 Adding a Third Dimension.

Up to this point, we've primarily worked with two-dimensional shapes to understand the fundamentals of various mathematical functions, including linear, trigonometric, and signed distance functions. However, the next natural step is to extend this approach to three-dimensional space, where we can generate more complex and realistic geometries.

To begin our transition from two-dimensional to three-dimensional space, we'll first review the mathematical formula that defines a circle in 2D and then demonstrate how it can be extended to represent a sphere in 3D.

$$r > \sqrt{x^2 + y^2}$$

(5.1.a)

The most common signed distance function for a circle is defined as the distance from any point in the xy-plane to the center of the circle, minus the radius r. This function returns negative values for points inside the circle, zero on the circumference, and positive values for points outside it. This distinction of signs is essential for determining the relative position of points with respect to the circle.

We can simplify Formula 5.1.a by using the scalar product:

$$r > \sqrt{x * x + y * y}$$

(5.1.b)

Or express it in terms of a two-dimensional vector v:

$$r > \sqrt{v \cdot v}$$

(5.1.c)

In HLSL, we can implement this signed distance function for a circle as:

```
float circle = length(uv) - r;
```

Here, **uv** is a two-dimensional vector representing the coordinates on the plane, and **r** corresponds to the radius of the circle. The **length()** function simplifies the calculation of the distance from a point on the plane to the center.

To transform this circle into a sphere, we need to extend the concept to three-dimensional space. Since both the circle and the sphere share the property that their points are equidistant from a center, we simply add a third z coordinate.

$$r > \sqrt{x^2 + y^2 + z^2}$$

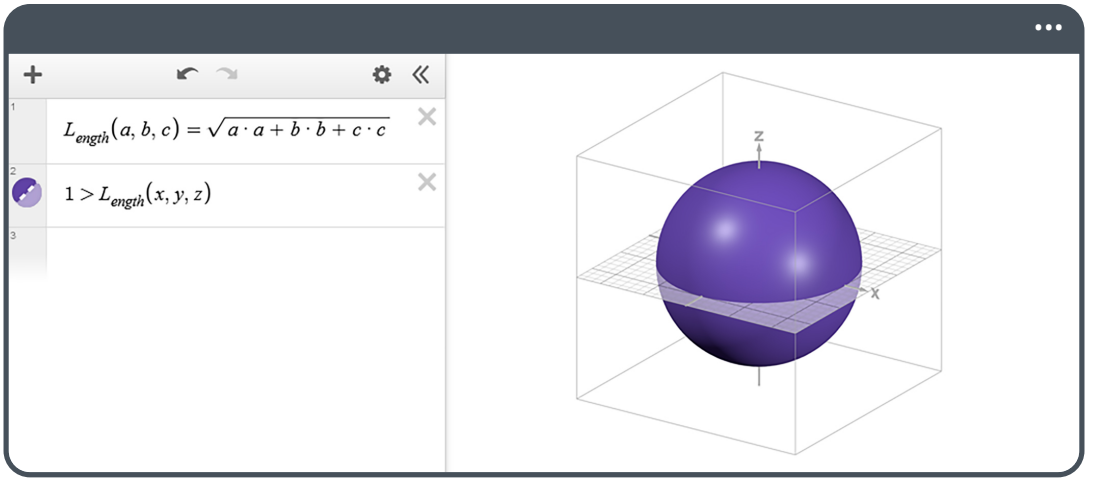
(5.1.d)

In HLSL, this expression translates as:

```
float sphere = length(p) - r;
```

Where **p** is a three-dimensional vector representing the position of a point in 3D space. Just like with the circle, the **length()** function calculates the distance from **p** to the origin.

When comparing the functions for the circle and the sphere, we can observe that both share the same structural form; the only difference is the dimensionality of the vector used. This pattern allows us to easily adapt distance functions to higher-dimensional spaces, making our implementations more versatile and powerful.



(5.1.e https://www.desmos.com/3d/bitqqlfpd2)

We can apply a similar analogy when converting a circle into a sphere and extend it to the case of a capsule, which represents the three-dimensional extension of a line segment. In the previous chapter, we worked with segment implementations while developing effects for UI images. To transform this geometric shape into a capsule, we can simply add a third dimension to our equations.

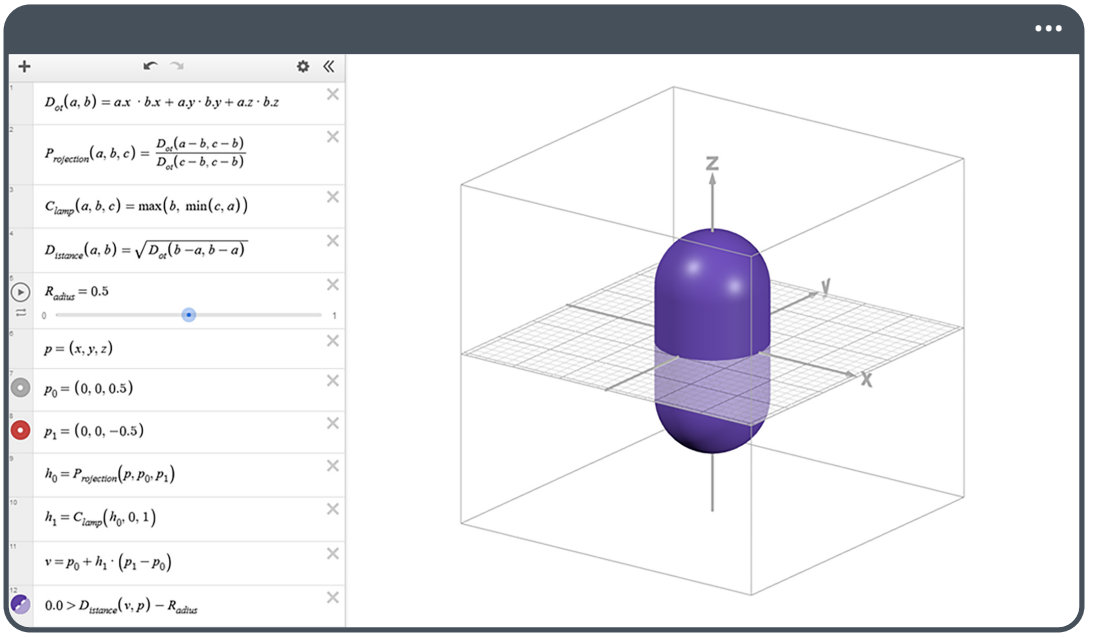
By taking the **`segment_sd_float()`** function defined in Section 4.2 and incorporating a third component, the definition becomes:

```
void capsule_sd_float(in float3 p, in float3 p0, in float3 p1, out float
 ↪ out)
{
    float h = projection(p, p0, p1);
    h = clamp(h, 0.0, 1.0);

    float3 p2 = p0 + h * (p1 - p0);
    Out = distance(p2, p);
}
```

In this code, **p** represents the point in three-dimensional space where we evaluate the distance function, while **p0** and **p1** define the endpoints of the 3D segment that forms the central axis of the capsule. The **projection()** method (extended to three dimensions) calculates the projection of **p** onto the segment defined by **p0** and **p1**, while **clamp()** ensures that this projection remains within the segment's range.

The variable **p2** determines the closest point on the segment relative to **p**, and finally, **distance()** calculates the distance between these points—a fundamental value for defining the "volume" of the capsule around the segment

By incorporating a third dimension into our equations, we not only add depth to our shapes but also unlock a wide range of possibilities for creating procedural geometry. With this foundation, we can flexibly combine and modify basic shapes, driving the creation of more advanced three-dimensional effects and structures.



(5.1.f https://www.desmos.com/3d/8xggngtnan)

Looking at Reference 5.1.f, we can observe that a capsule has been defined using the same mathematical operations as a signed distance function, implemented in Desmos

for visualization. This leads us to consider how to render these shapes using shaders in Unity and how to integrate them seamlessly into our graphical environment.

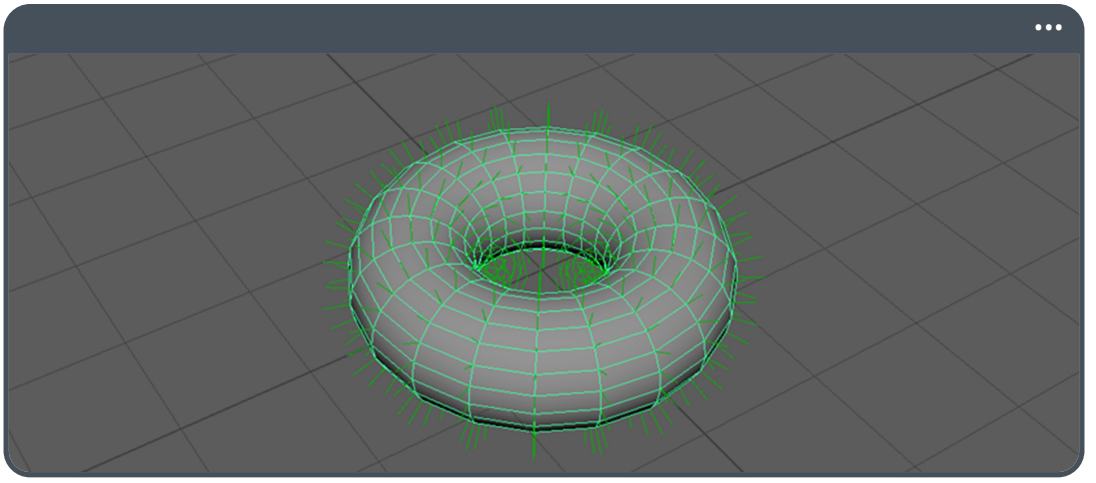
To properly render these three-dimensional shapes, we must take into account at least three essential factors:

- Surface normals: Determine how light interacts with the surface and consequently affect the final appearance of the shape.
- Lighting: Defines the light distribution throughout the shape, highlighting its contours and volumes.
- Rendering: Covers the process of drawing the shape on the screen. On this occasion, using the **Ray Marching** technique.

Normals play a crucial role in accurately computing lighting. These vectors, which are perpendicular to the surface, define the orientation of each point, enabling precise simulation of lighting effects such as reflections and shadows. If normals aren't calculated correctly, shapes may appear flat or display inaccurate reflections, negatively affecting their realism and overall appearance.

Lighting is essential for emphasizing details and enhancing the three-dimensionality of our shapes. By applying lighting models, we can improve the perception of volume and materiality in procedural forms. Both diffuse and specular lighting contribute to simulating how light interacts with different surfaces, adding greater realism to the scene.

With these aspects—normals, lighting, and rendering—in mind, we'll be ready to take the next step: implementing everything we've learned about generating and rendering three-dimensional shapes based on signed distance functions in Unity.



(5.1.g Normal vector on a torus)

As we can see in Figure 5.1.g, the torus displays its normals (represented by green lines) perfectly aligned with the orientation of each vertex, resulting in a smooth and harmonious distribution of lighting across the surface. However, there are lighting techniques that require precise control over normal orientation to achieve specific effects, such as toon-style lighting or Cel shading.

The reference image comes from Maya, a widely used 3D modeling software in the industry. However, in our case, we don't have vertices from which to extract or modify normals, as our shapes are generated using signed distance functions. Therefore, we must calculate and orient the normals through mathematical procedures, ensuring they accurately reflect the implicit geometry of each shape.

5.2 Normals in Signed Distance Functions.

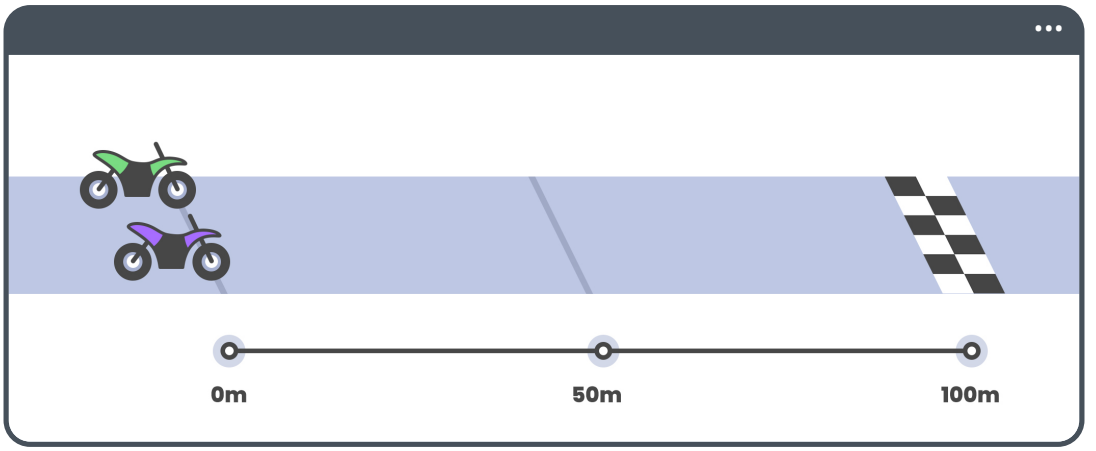
To calculate normals in procedural shapes, it's essential to understand the concept of the gradient. In the case of a signed distance function, the gradient is a vector that indicates the direction of the greatest increase of the function. For each point on the surface, this vector is perpendicular, defining the normal at that specific location. This allows us to precisely determine the surface's orientation, ensuring accurate lighting effects in the shader.

Before implementing these concepts in our project, we'll first introduce the fundamental principles of gradient calculation. We'll begin with the nabla symbol ∇, a vector operator whose components are partial derivatives:

$$\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$$

(5.2.a)

The derivative is a mathematical tool that describes how a function changes at specific points when its independent variable varies. To grasp this concept, it's essential to be familiar with the notions of average velocity and instantaneous velocity, as the derivative is fundamentally based on these ideas. We'll begin by explaining the concept of average speed through an example involving a motorcycle race.



(5.2.b)

Imagine two motorcycles racing over a distance of 100 meters. Both start from rest, and we want to determine their speed over time. Suppose the pink motorcycle reaches the finish line in 5 seconds, while the green motorcycle covers only 80 meters in the same time span.

To calculate the average speed v of each motorcycle, we'll use the following formula, where the delta symbol Δ represents change:

$$v = \frac{\Delta x}{\Delta t}$$

(5.2.c)

Where:

- Δx equals distance traveled.
- Δt equals elapsed time.

If we now apply this equation to the example, we get:

For the pink motorcycle:

$$V_1 = \frac{100m}{5s} = 20m/s$$

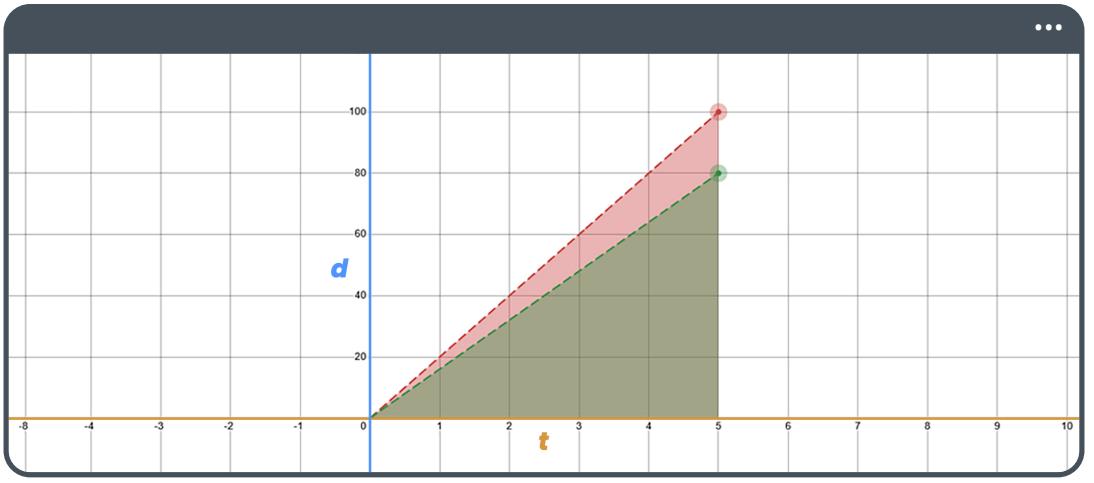
(5.2.d)

And for the green motorcycle:

$$V_2 = \frac{80m}{5s} = 16m/s$$

(5.2.e)

We can represent this data in a position vs. time graph to better visualize the progression of each motorcycle:



(5.2.f https://www.desmos.com/calculator/7yn0wlzdpq)

In the graph, the vertical axis represents the distance d traveled, while the horizontal axis indicates the time t. The dashed lines illustrate how each motorcycle progresses over time, and the slope of each corresponds to the average speed over a specific interval.

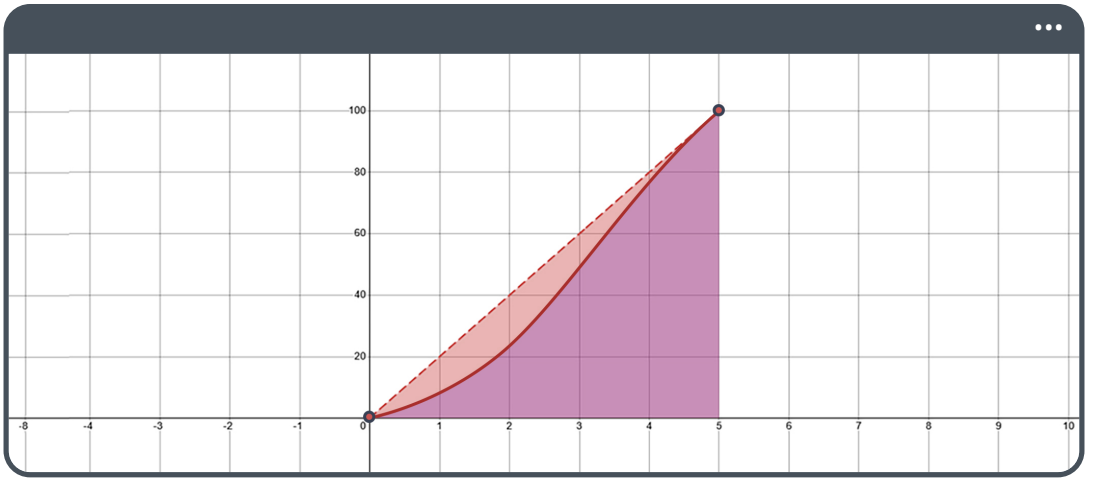
Continuing with the motorcycle example, we can determine the average speed over different segments. For instance, in the first 2 seconds, the pink motorcycle would have traveled 40 meters.

$$v_1 * 2s = 40m$$

(5.2.g)

However, in practice, motorcycles don't reach their maximum speed instantaneously. As they accelerate, their speed gradually increases, meaning that the actual distance traveled in the first few seconds is usually less than what is predicted by the average speed.

While average speed provides a global view of movement over a given interval, it doesn't describe how it varies second to second. To precisely determine speed variation at each instant, we have to introduce the concept of instantaneous speed, which corresponds to the rate of change of position at a specific point along the trajectory.



(5.2.h)

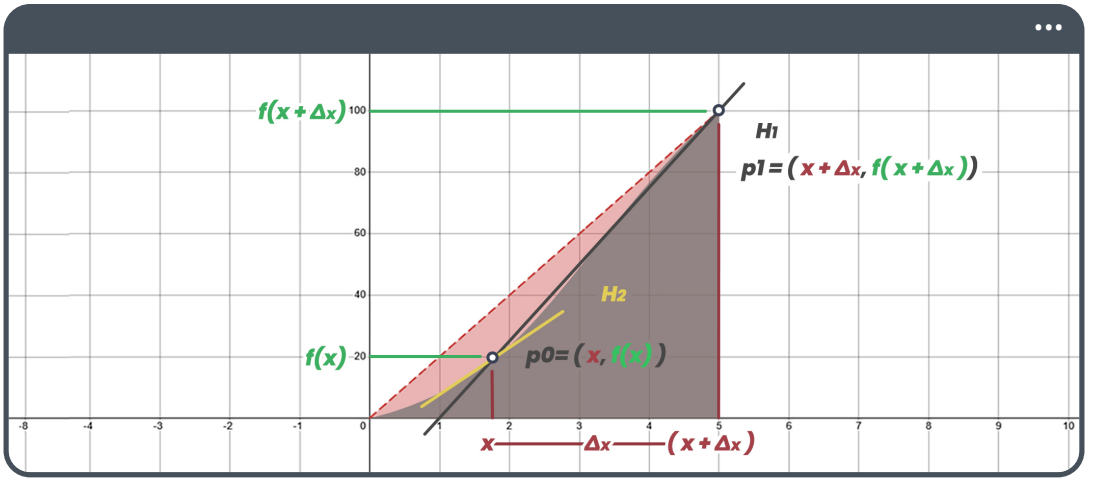
In the more realistic graph above, we observe that the pink motorcycle starts by accelerating gradually, which is reflected as a curve in the distance vs. time graph. Instantaneous velocity describes the rate of change of position at a specific moment and is mathematically calculated as the derivative of position with respect to time. To determine the speed at a given time t, we can evaluate the derivative of the position function at that instant, obtaining the exact velocity at that moment.

By definition, if $f(x)$ is a function, its derivative $\frac{df}{dx}$ is defined like this:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

(5.2.i)

To illustrate this concept, we'll consider two points on the Cartesian plane: p_0 and p_1. We'll place p_0 at the location where we want to calculate the derivative; following the previous example, this corresponds to determining the instantaneous speed approximately 2 seconds after the race begins, assuming that the x coordinate represents time on the graph. Meanwhile, p_1 is positioned at a point corresponding to, for example, 5 seconds.



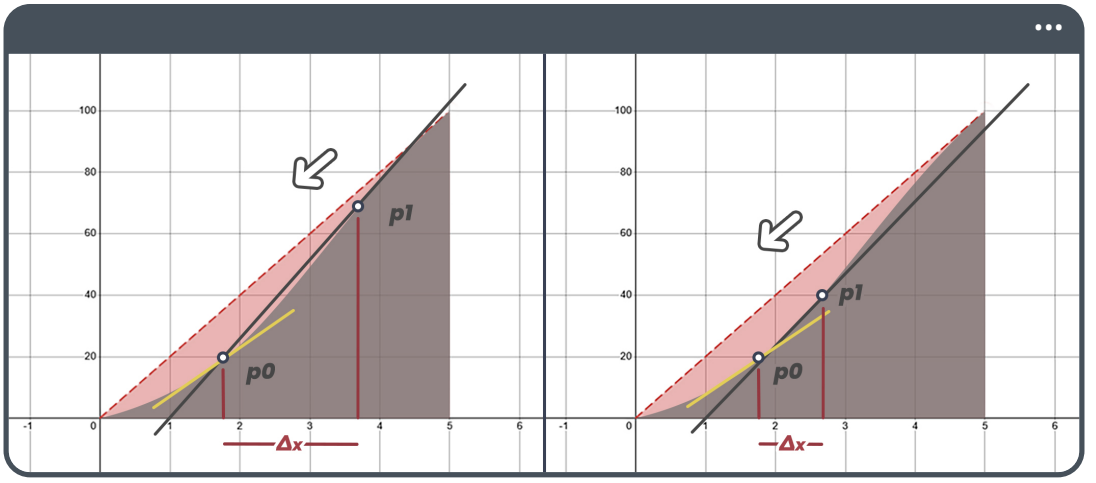
(5.2.j)

The interval Δx between p_0 and p_1 represents the increase in the x-coordinate. Thus, we can express the coordinates of p_1 as:

$$p_1 = (x + \Delta x, f(x + \Delta x))$$

(5.2.k)

If we make Δx increasingly smaller (tending to zero), the line connecting p_0 and p_1 approximates the tangent line to the curve at p_0. Although Δx is never strictly zero in practice, if we use a very small value (e.g., $\Delta x = 0.001$) we can get a good approximation of the instantaneous speed in the first 20 meters.



(5.2.l)

Up to this point, we've worked with the total derivative, which we apply to functions of a single variable x. However, when we deal with multivariable functions, such as $f(x, y, z)$, we must use partial derivatives, each corresponding to one of the function's dimensions.

In the one-dimensional case, where the function is represented as a curve, there's only one direction in which to calculate the derivative. However, for multivariable functions, such as $f(x, y, z)$, we can calculate derivatives along each of the directions corresponding to the variables x, y, and z. These multivariable functions describe surfaces (or volumes) in higher-dimensional spaces.

Returning to the equation in Figure 5.2.i, recall that partial derivatives can be approximated using finite differences, just as we saw with the general definition of the derivative. For example:

Partial derivative of f, with respect to x:

$$\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y, z) - f(x, y, z)}{\Delta x}$$

(5.2.m)

Partial derivative of f, with respect to y:

$$\frac{\partial f}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y, z) - f(x, y, z)}{\Delta y}$$

(5.2.n)

Partial derivative of f, with respect to z:

$$\frac{\partial f}{\partial z} = \lim_{\Delta z \to 0} \frac{f(x, y, z + \Delta z) - f(x, y, z)}{\Delta z}$$

(5.2.o)

When the surface in space is defined by $f(x, y, z) = k$, where k is any constant, a natural question arises: How can we determine a direction that is perpendicular to the surface at a given point? This perpendicular direction is known as the normal to the surface, and we can find it using the gradient, denoted as ∇f. This vector unites the partial derivatives, expressed as follows:

$$\nabla f(x, y, z) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$$

(5.2.p)

The Gradient ∇f describes the direction in which the function f increases most rapidly and the rate at which it does so. When considering a surface defined by $f(x, y, z) = 0$, for any movement along the surface f stays constant (equal to 0). This means that tangent vectors at the surface don't change the value of f.

To clarify this, suppose you move infinitesimally in the direction of a vector v that is tangent to the surface. The infinitesimal change df on the surface $f(x, y, z) = k$ as you move in that direction is described as the dot product:

$$df = \nabla f \cdot v$$

(5.2.q)

However, since $df = 0$ while remaining on the surface as the value of f changes, it holds that:

$$\nabla f \cdot v = 0$$

(5.2.r)

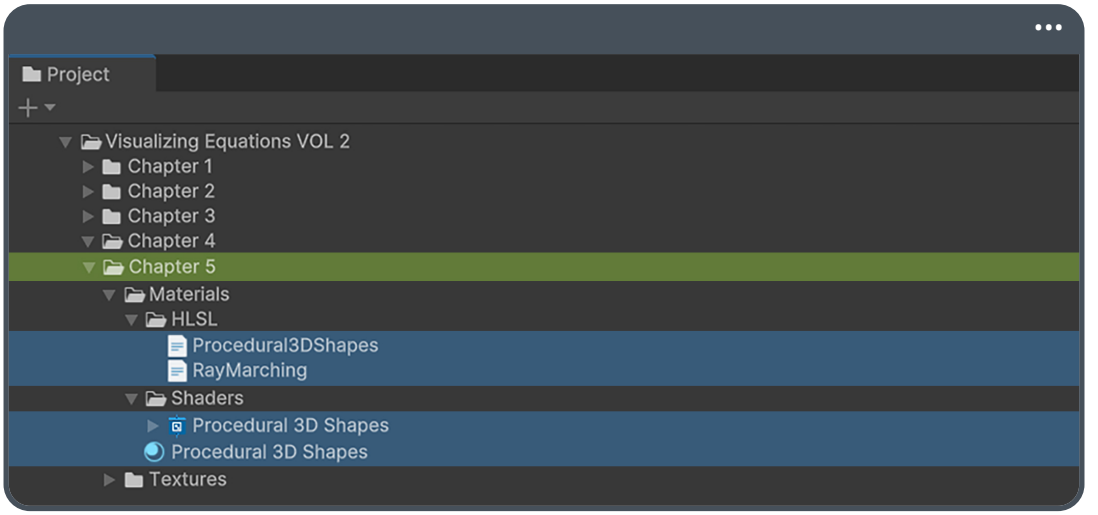
In other words, ∇f is perpendicular to any tangent vector v, and therefore, it coincides with the normal direction of the surface. This makes the gradient ∇f much more than just a collection of partial derivatives—it serves as a geometric tool that directly connects to the surface's structure through the concept of perpendicularity.

Its nature, by pointing to the maximum change in f naturally makes it the surface normal vector, demonstrating how mathematics and geometry combine to elegantly describe figures and spaces in rendering projects with signed distance functions.

5.3 Rendering a three-dimensional Shape.

Having explored the functions required to create a three-dimensional procedural shape and compute its normals, this section will focus on its rendering in Unity. To begin, create an **Unlit Shader Graph** and name it **Procedural 3D Shapes**. Next, you need to generate a material for this shader and create a separate **.hlsl** file with the same name to maintain a structured workflow.

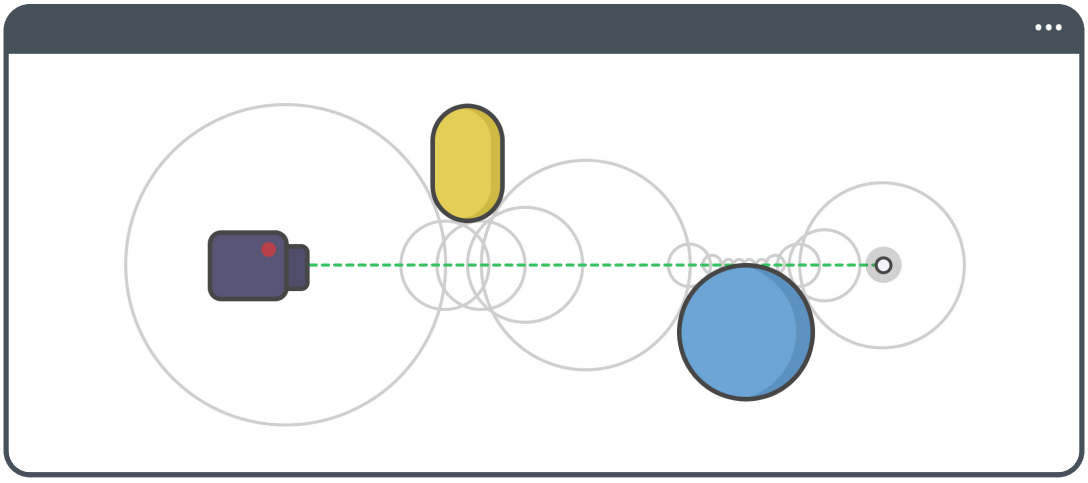
Additionally, create a new **.hlsl** file named **RayMarching** which you'll use to generate a custom node within Shader Graph and later to render the three-dimensional shapes. This file separation helps maintain a modular code structure, making it easier to edit or extend distance and normal calculation functions in the future.



(5.3.a)

The Ray Marching technique is essential for rendering shapes defined by signed distance functions. In this iterative process, we march along a ray, which is defined by its direction and an initial distance. At each step, we advance in small increments, measuring the distance to nearby objects. By following this approach, we can accurately determine whether the ray collides with a shape, enabling us to draw it realistically.

Below, there's a visual reference to help better understand this concept.

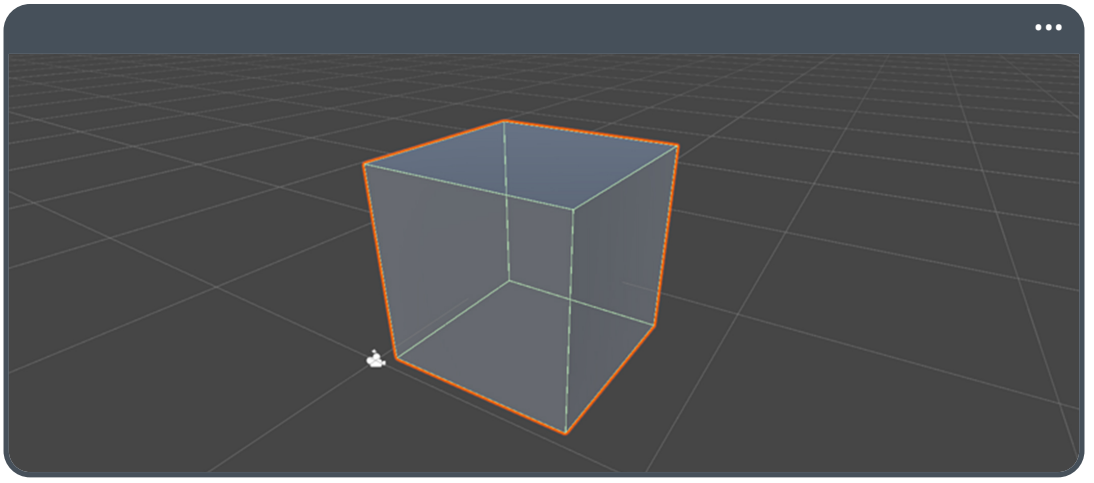


(5.3.b Two-dimensional conceptualization of a ray)

Looking at Figure 5.3.b, we can see a ray traveling through a scene. This ray is defined by a point of origin (the camera position), a direction, and a distance. Along its path there are two geometric shapes: a capsule and a sphere, both three-dimensional. At each iteration or "jump" of the ray the distance to the nearest surface is evaluated, allowing us to detect collisions and thus define the shape and location of objects in the scene.

Now, how do we apply this technique in Unity? First, we need to define properties in Shader Graph and create a custom node responsible for performing distance calculations and collision detection. This node will use signed distance functions and progressively advance the ray in small iterations until it either detects the surface of an object or not in the scene.

In this process, it's essential to have a "scene" in which to draw the procedural shapes. While any default primitive in the engine can be used, for this exercise, we'll use a Cube as the base for our calculations. This Cube will serve as the surface onto which we apply our Ray Marching shader, so that the procedural shapes (the capsule and the sphere) are correctly drawn inside it.



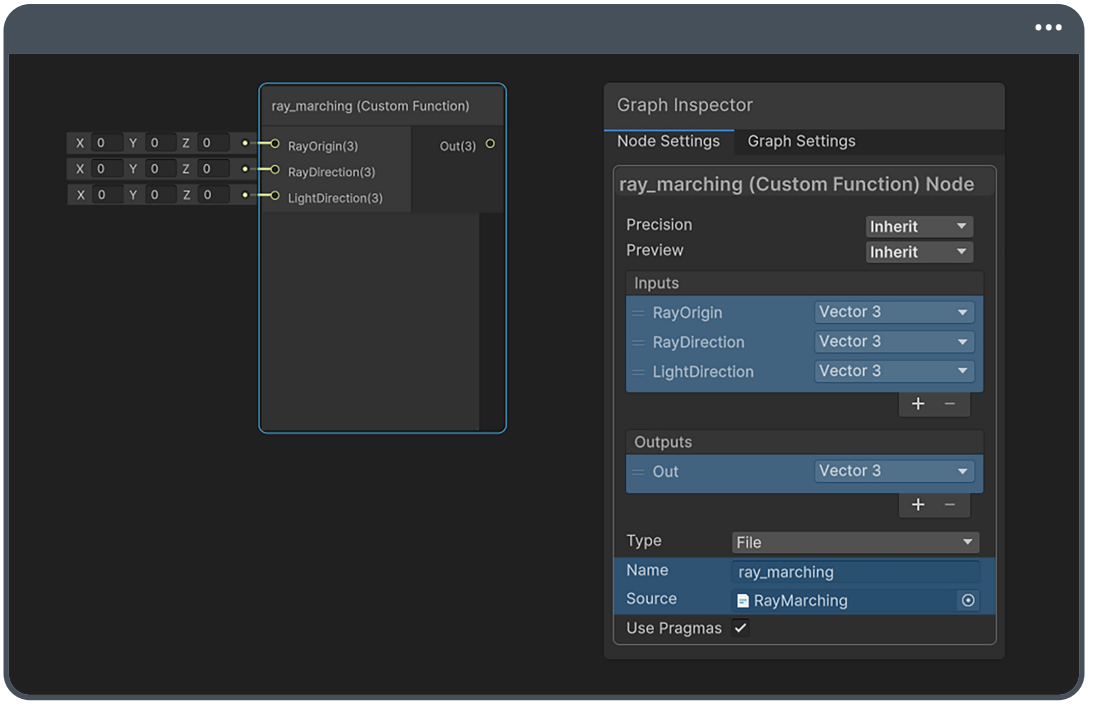
(5.3.c Hierarchy > 3D Object > Cube)

To visualize the desired effect, you need to assign the **Procedural 3D Shapes** shader to its respective material, and then apply that material to the Cube in the scene. Next, begin the development process by adding a **Custom Function** node within the previously created Shader Graph.

Before configuring the inputs and outputs of this node, let's briefly review the properties needed to render the 3D shapes:

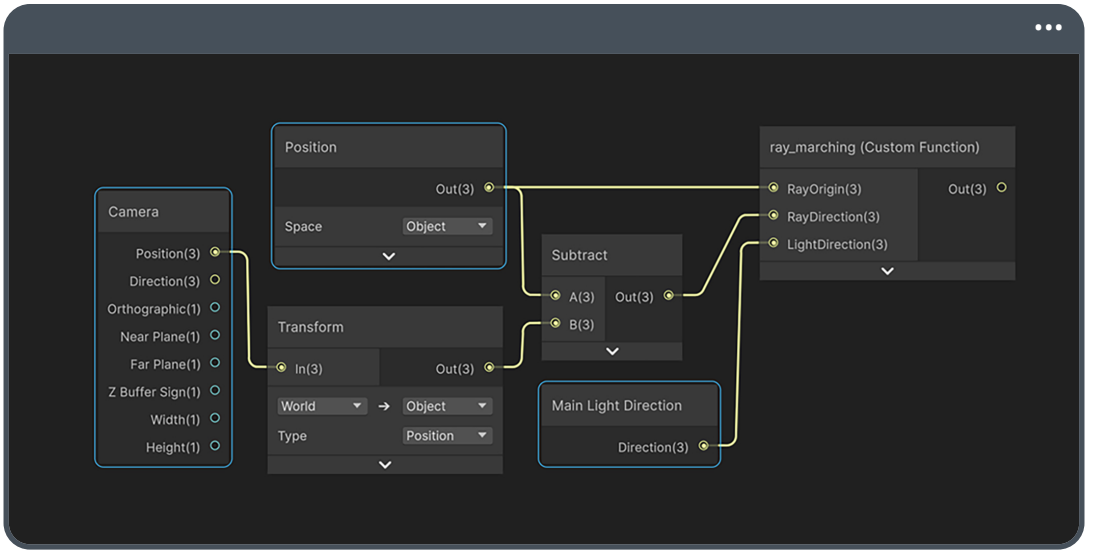
- **Vertex Position:** Refers to the space where we want to draw the procedural shape. We'll use the object-space vertex position as the origin point for calculations within the Cube's area. While we could also use the camera position, relying on vertex position in this case helps us optimize calculations and simplify the logic.
- **Camera:** We use this property to determine the direction of the ray. Essentially, the ray starts from the camera and travels toward the procedural shape in the scene, enabling us to calculate intersections with the generated surfaces.
- **Lightning:** To calculate lighting in our procedural shapes, we need both the normals and the light direction. For this, we'll use the **Main Light Direction** node, which provides access to the primary directional light in the scene.

As described in the previous section, we'll need to calculate procedural normals manually, since Shader Graph doesn't provide a dedicated node for this task. Therefore, we'll define only the essential properties for our node: **RayOrigin**, **RayDirection**, and **LightDirection**. These inputs allow us to perform the Ray Marching technique and apply the appropriate lighting models to render the three-dimensional shapes accurately.



(5.3.d Custom Function nodes with inputs and output)

You can notice in Figure 5.3.d that the three previously described properties are presented as three-dimensional vectors. Before proceeding with the implementation of the rendering functions, you must connect each of these properties to their corresponding nodes in the shader. This ensures that the vertex position, camera position, and lighting direction in the scene are correctly passed to your custom node.

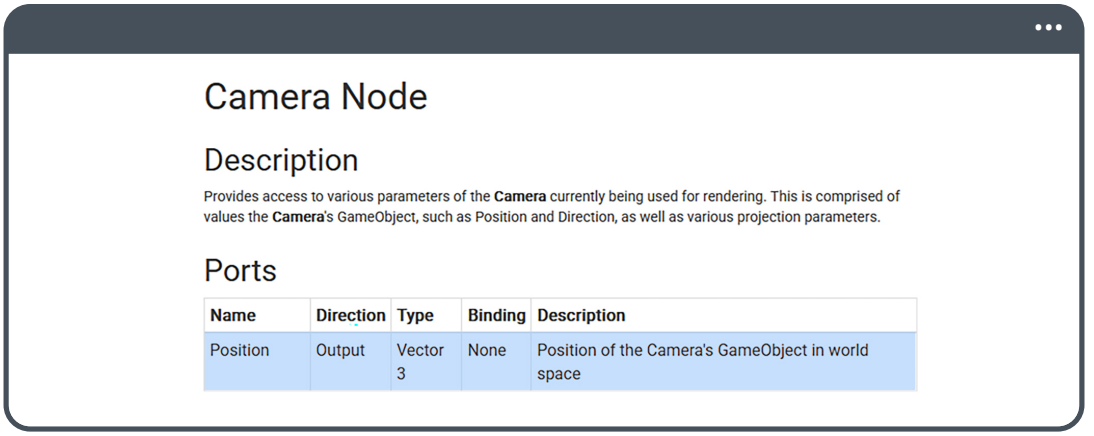


(5.3.e Visualization of nodes in Shader Graph)

In Figure 5.3.e, the vertex position (in object-space) has been connected as the ray origin in the **Custom Function** node. To better understand this choice, we need to consider the influence of the Cube on the procedural shapes inside it. For instance, if we rotate the Cube, which serves as the rendering stage for the procedural shapes, should the shapes inside it rotate along with it, or should they remain static, independent of the Cube's position, rotation, and scale?

If we want the procedural shapes to be influenced by the Cube's transformation, we must perform our calculations in object-space. As a result, the camera position must be transformed from world-space to object-space using the Transform node, considering two fundamental aspects:

- The official Shader Graph documentation states that **Camera** node position is set to world-space by default.
- Next, we must calculate the ray direction by subtracting the camera position from the vertex position. Therefore, both coordinates must be in the same space to ensure the subtraction produces valid results.



Camera Node

Description

Provides access to various parameters of the **Camera** currently being used for rendering. This is comprised of values the **Camera**'s GameObject, such as Position and Direction, as well as various projection parameters.

Ports

Name	Direction	Type	Binding	Description
Position	Output	Vector 3	None	Position of the Camera's GameObject in world space

(5.3.f https://docs.unity3d.com/.../...shadergraph@6.9/manual/Camera-Node.html)

In our Custom Function node, we've also connected the **Main Light Direction** node to the corresponding parameter in the function. With these references in place (ray origin, ray direction and light direction), we are now ready to implement the necessary rendering functions.

As a first step, you'll create the **`ray_marching_float()`** method, which takes the previously mentioned properties as input. To ensure that the node compiles correctly, you'll need to assign arbitrary values to the RGB components of the output. This step will help you verify that the basic node structure is working before delving into the Ray Marching logic.

For example, you can declare the method as follows in your **.hlsl** file:

```
4 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
  ↪ float3 lightDirection, out float3 RGB)
5 {
6     RGB = 0;
7 }
```

In the code snippet above, you can observe that the arguments **RayOrigin**, **RayDirection**, and **LightDirection** belong to the **Custom Function** node. This is where you'll calculate

the distance to the surfaces and determine the final color based on the lighting and the properties of the shape. This initial step allows you to confirm that the shader compiles and runs correctly, laying the foundation for the full implementation of the Ray Marching technique.

To complete the shader configuration, you must carry out three fundamental tasks:

- Define the shape to be rendered (in this case, a sphere).
- Calculate the normals for the shape.
- Implement a rendering method that uses the distance function to traverse the space and detect the surface of the figure.

You can start by declaring the function to describe the sphere. To do this, open the script **Procedural3DShape** and add the following lines of code:

```
float sphere_sd (float3 p, float r)
{
    return length(p) - r;
}
```

The **`sphere_sd()`** function returns the distance between an arbitrary point **`p`** (the ray collision) in three-dimensional space and the surface of a sphere with radius **`r`**. Here, a negative value indicates that the point is inside the sphere, a zero value means that the point is exactly on the sphere's surface, and a positive value signifies that the point is outside the sphere. Point p is described as:

$$p(d_o) = O + d_o * D$$

(5.3.g)

Where O is equal to the origin of the ray, d_0 represents the scalar distance traveled in each iteration, and D is the direction. Since the sphere is defined in a different file to **RayMarching**, you'll need to include this file to be able to use the function. You can use the directive **`#include`** to do this as shown below:

```
#include "Assets/Jettelly Books/…/Procedural3DShapes.hlsl"

void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
 ↪  float3 lightDirection, out float3 RGB) { … }
```

In the previous example, the path used in the directive has been optimized to make it shorter. However, you can access the file using its full path:

- Assets > Jettelly Books > Visualizing Equations VOL 2 > Chapter 5 > Materials > HLSL > Procedural3DShapes.hlsl.

Next, you need to include your recently declared sphere inside a new method in **RayMarching**, and call it **`map()`**. This method allows you to position all the procedural shapes that you want to place in the Cube in the scene.

```
#include "Assets/Jettelly Books/…/Procedural3DShapes.hlsl"

inline float map (float3 p)
{
    const float sphere_radius = 0.3;
    float sphere = sphere_sd(p, sphere_radius);
    return sphere;
}

void ray_marching_float (in float3 rayOrigin, in float3 rayDirection, in
 ↪  float3 lightDirection, out float3 RGB) { … }
```

If you pay attention to line 5, you'll notice the declaration of the variable **`sphere_radius`**, whose value is 0.3. Remember that Unity's metric system is defined in meters, where one unit is equivalent to 100 cm. Therefore, in this case, the sphere has a radius of 30 cm.

Another relevant aspect appears in line 3, where you can see the use of the parameter **`inline`** before the method name. This parameter, known as **`StorageClass`**, determines how the compiler handles methods during the compilation process. When a method is invoked, the system navigates to the position in memory where the method is located, executes the code, and then returns to the original call point. This process introduces an additional computational cost due to the same call to the function. However, by marking a method as **`inline`**, the compiler may insert the method's code directly at the call point, eliminating overloading and potentially optimizing performance.

It's important to note that according to the official Microsoft documentation, in HLSL all methods are treated as **`inline`** by default, even if this parameter isn't specified. However, it's worth bearing in mind that when creating very large methods, the shader size could increase during compilation and cause cache failures.

With this explanation in mind, you'll now proceed with implementing the method responsible for executing the Ray Marching algorithm, which is necessary to render the sphere. For practical reasons, you'll name this method **`render()`** and pass both the ray origin and the ray direction as arguments.

```
10 inline float render(float3 rayOrigin, float3 rayDirection)
11 {
12     return 0;
13 }
```

Since the rendering process is performed iteratively, the first variable we need to define in the **`render()`** method is the distance along the ray d_0. This starts at 0.0, coinciding with the camera position in the scene:

```
inline float render(float3 rayOrigin, float3 rayDirection)
{
    float ray_distance = 0;
    return 0;
}
```

Next, you need to define the maximum number of iterations and create a **`for`** loop that advances along the ray in the scene. To do this, follow these two steps:

- Declare the directive **`#define MAX_ITERATIONS`**.
- Add a **`for`** loop that runs an iteration on each step.

```
#define MAX_ITERATIONS 256
```

```
inline float render(float3 rayOrigin, float3 rayDirection)
{
    float ray_distance = 0;
    for ( int i = 0; i < MAX_ITERATIONS; i++)
    {

    }
    return 0;
}
```

To determine the surface of the sphere, you need to calculate the current position of the ray as it moves forward. To do this, use Equation 5.3.g inside the loop, adding the displacement in the direction in each iteration.

Next, you can pass the result (a new three-dimensional vector) as an argument to the **`map()`** method, as shown below:

```
12 inline float render(float3 rayOrigin, float3 rayDirection)
13 {
14     float ray_distance = 0;
15     for ( int i = 0; i < MAX_ITERATIONS; i++)
16     {
17         float3 p = rayOrigin + ray_distance * rayDirection;
18         float surface = map(p);
19     }
20     return 0;
21 }
```

Initially, the ray position starts at the origin. However, after each step, it advances according to the expression: **`ray_distance`** multiplied by **`rayDirection`**, as you can see in line 17 of the previous example. Later, when using the **`render()`** method in **`ray_marching_float()`**, you must normalize the ray direction to ensure an adequate scale.

Until now, the ray distance remains at its initial value. Therefore, you need to progressively increase it within the scene until it collides with the sphere you want to render. However, this raises two questions: what happens if the ray travels beyond the surface of the procedural sphere? And how precisely should I determine contact with the surface? To solve this, you need to define the maximum distance that the ray can travel and the tolerance with which you'll measure the collision.

```
3 #define MAX_ITERATIONS 256
4 #define MAX_DISTANCE 100.0
5 #define SURFACE_DISTANCE 0.001
```

The previous code fragment used the directive **#define** to specify two additional constants.

- **MAX_DISTANCE** (100 m)) limits the ray's range, ensuring that only figures within this distance are rendered.
- **SURFACE_DISTANCE** defines the resolution or minimum distance at which the ray is considered to have touched the surface of the sphere.

Finally, in the **render()** method, you need to increase the ray distance at each iteration, verifying whether it collides with the sphere or exceeds the maximum allowed distance.

```
14 inline float render(float3 rayOrigin, float3 rayDirection)
15 {
16     float ray_distance = 0;
17     for ( int i = 0; i < MAX_ITERATIONS; i++)
18     {
19         float3 p = rayOrigin + ray_distance * rayDirection;
20         float scene_distance = map(p);
21         ray_distance += scene_distance;
22         if(scene_distance < SURFACE_DISTANCE || ray_distance > MAX_DISTANCE)
23         {
24             break;
25         }
26     }
27     return ray_distance;
28 }
```

In this way, whenever **scene_distance** is less than the established precision, you can assume that the ray has hit the surface of the sphere. On the contrary, if **ray_distance** exceeds the maximum distance, the loop is stopped, ruling out the possibility of an intersection within the defined range.

If you pay attention to lines 21 to 25, you can see that the code implements logic to determine whether the ray has hit a surface or exceeded the maximum limit in the scene.

In line 21, the variable **ray_distance** is incremented with the value of **scene_distance**, which represents the calculated distance between the ray's current position and the nearest surface, determined through the **map()** method. Later, in lines 22 to 25, there's a crucial condition for the Ray Marching algorithm: the instruction **if**, this verifies two cases:

- Collision with surface: If **scene_distance** is less than **SURFACE_DISTANCE**, the ray is close enough to the sphere (within the defined tolerance). In this case, we consider an intersection to have occurred (as shown in Reference 5.3.b).
- Maximum distance exceeded: If **ray_distance** surpasses **MAX_DISTANCE**, it means that the ray has traveled beyond the established limit without encountering a surface. There's no need to continue the iterations.

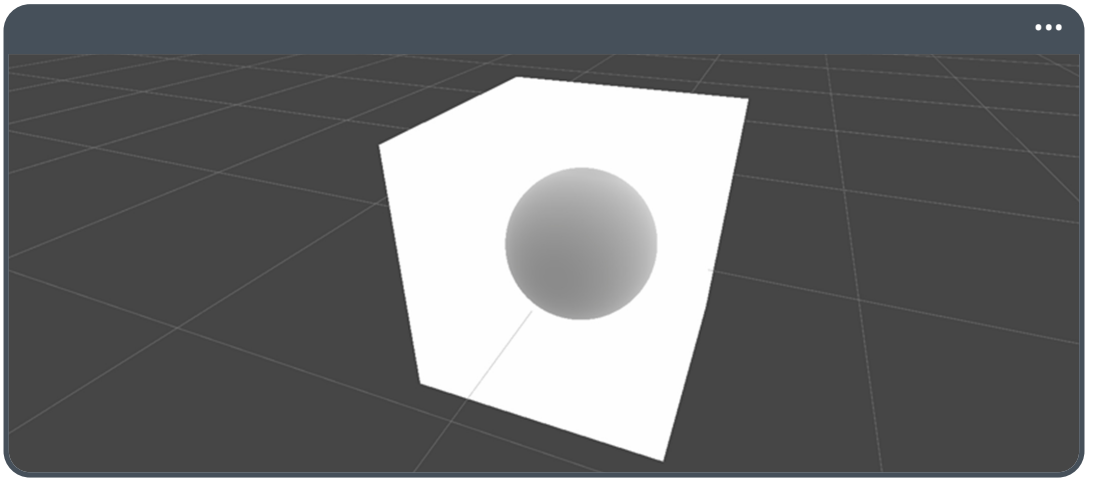
In either of these scenarios, the **for** loop stops (using the **break** instruction), optimizing performance by avoiding unnecessary iterations that would not significantly alter the result. Finally, the **render()** method returns **ray_distance**, which represents the total distance traveled by the ray until it either collides with a surface or reaches the maximum allowed distance.

You can use the distance value obtained in the **render()** method to visualize the sphere as shown below:

```
30 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
   ↪ float3 lightDirection, out float3 RGB)
31 {
32     rayDirection = normalize(rayDirection);
33     float distance = render(rayOrigin, rayDistance);
34
35     RGB = distance;
36 }
```

In the previous code, notice that the line 32 normalizes **rayDirection** to ensure that the vector has a unit length, guaranteeing an accurate ray direction. Then, line 33 calls

the **`render()`** method, which returns the total distance traveled by the ray until it either collides with a surface or reaches the maximum allowed distance. This result is stored in the **`distance`** variable, which is finally assigned to **`RGB`**. This allows you to visualize the scene based on the calculated distance.



(5.3.h Display of the distance from the camera)

In the Cube in your scene, the white color around the sphere corresponds to rays that didn't collide with any surface, as they return a value greater than one. To better understand this behavior, you need to implement a method that allows you to approximate the normals of the sphere. For this, you can use Function 5.2.p from the previous section, which calculates the derivative (or finite difference) of a function with respect to its variables.

```
30 inline float3 get_normal(float3 p)
31 {
32     float h = SURFACE_DISTANCE;
33     float x = (map(p + float3(h, 0, 0)) - map(p)) / h;
34     float y = (map(p + float3(0, h, 0)) - map(p)) / h;
35     float z = (map(p + float3(0, 0, h)) - map(p)) / h;
36
37     return normalize(float3(x, y, z));
38 }
```

It's important to note that the previous implementation is neither the most optimized nor the most precise way to approximate the normals of an implicit surface, as it involves multiple subtractions, divisions, and additions. However, it serves as an intuitive and direct approach to illustrating how normals can be calculated using finite differences.

You could consider the following implementation a more optimized alternative for computing normals. This version avoids costly division operations, provides greater stability to the function, and produces the same result.

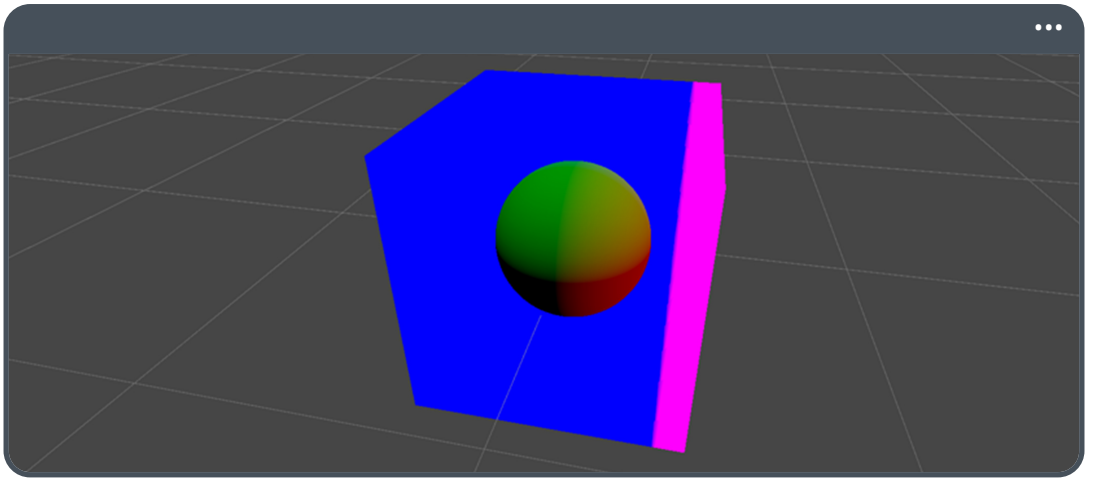
```
inline float3 get_normal(float3 p)
{
    float2 e = float2(0.00001, 0);
    float3 n = map(p) - float3(map(p - e.xyy), map(p - e.yxy), map(p -
     ↪  e.yyx));
    return normalize(n);
}
```

The **`get_normal()`** method calculates the normal of a point **`p`** by approximating the partial derivatives of the **`map()`** distance function with respect to each vector component. To do this, you can use a small increment **`h`** (line 32) and evaluate how the distance function varies when moving slightly in each direction. By normalizing the resulting vector (line 37), you get the direction of the normal at point **`p`**, which is essential for computing lighting effects in your procedural shape.

Similarly to the **`render()`** method, formula 5.3.g determines point **`p`**, that is:

```
40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
   ↪ float3 lightDirection, out float3 RGB)
41 {
42     rayDirection = normalize(rayDirection);
43     float distance = render(rayOrigin, rayDistance);
44     half3 color;
45
46     float3 p = rayOrigin + distance * rayDirection;
47     float3 normal = get_normal(p);
48     color = normal;
49
50     RGB = color;
51 }
```

The **`color`** vector represents the final color of the image. As you can see in line 48, it's obtained by calculating the normals using the **`get_normal()`** method on line 47. After saving and returning to the scene, you'll be able to visualize the direction of the normals on the sphere.



(5.3.i Visualization of the sphere normals)

Next, we'll implement a simple lighting model to improve the visualization of our procedural shape. We'll use Lambert's diffuse lighting model, which calculates the light

intensity as a function of the angle between the surface normals n and the direction of incident light l. The formula for diffuse lighting is as below:

$$L = \max(0, n \cdot l)$$

(5.3.j)

You can implement this formula as follows:

```
40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
   ↪ float3 lightDirection, out float3 RGB)
41 {
42     rayDirection = normalize(rayDirection);
43     float distance = render(rayOrigin, rayDistance);
44     half3 color;
45
46     float3 p = rayOrigin + distance * rayDirection;
47     float3 normal = get_normal(p);
48     float l = max(0.0, dot(normal, -lightDirection)) + 0.1;
49     color = l;
50
51     RGB = color;
52 }
```

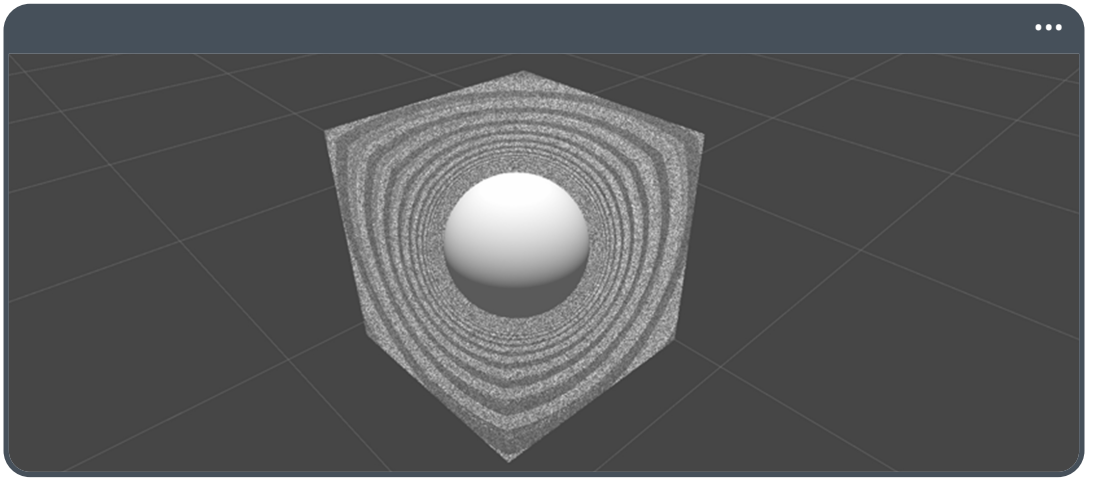
The function is fairly simple: it takes the maximum value between 0.0 and the dot product of the surface normals at point **p** and the light direction in the scene. By setting the minimum value to 0.0, we prevent negative values, which helps avoid visual artifacts related to lighting issues.

If you look at line 48, you'll notice that the function adds 0.1 at the end of the operation. This value acts as global lighting, defining a minimum illumination level across the entire scene, ensuring that areas not receiving direct light aren't completely dark.

You could use a more advanced implementation for global lighting, for example, the **BakedGI** node, to generate a lightmap that serves the same function. However, in this case, this small additional value is enough to add a softened contrast to the shape. That said, adding this number introduces an out-of-range issue, as the maximum value could now be greater than 1.0, which could lead to visual artifacts. To fix this, you can simply limit the range of **`l`** to a value between 0.0 and 1.0, as shown below:

```
48 float l = max(0.0, dot(normal, -lightDirection)) + 0.1;
49 l = saturate(l);
50 color = l;
```

The **`saturate()`** function ensures that the value of **`l`** stays within the range [0.0 : 1.0], preventing possible visual artifacts caused by values greater than 1.0. With this fix, the lighting in the scene becomes more controlled and visually coherent.



(5.3.k Noise visualization around the sphere)

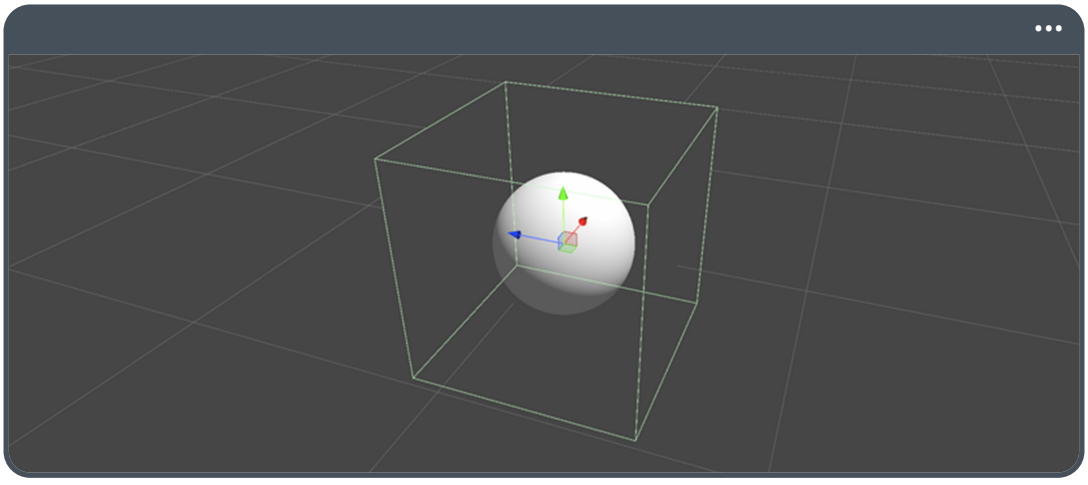
As you can see from previous reference, the volume of the sphere is correctly perceived. If you orbit around the Cube or rotate the directional light, you'll notice that the projection of lighting and shadows on the sphere follows the direction of the light. However, at the same time, all the rays computed from the **`render()`** method are also being calculated,

which appears as "noise" inside the Cube. To eliminate this noise, simply use the **`discard`** statement, which is equivalent to "don't show the result of the current pixel."

```
40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
   ↪ float3 lightDirection, out float3 RGB)
41 {
42     rayDirection = normalize(rayDirection);
43     float distance = render(rayOrigin, rayDistance);
44     half3 color;
45
46     if(distance < MAX_DISTANCE)
47     {
48         float3 p = rayOrigin + distance * rayDirection;
49         float3 normal = get_normal(p);
50         float l = max(0.0, dot(normal, -lightDirection)) + 0.1;
51         l = saturate(l);
52         color = l;
53     }
54     else
55     {
56         discard;
57     }
58
59     RGB = color;
60 }
```

If you look at the conditional statement between lines 46 and 57, you can observe that all pixels outside the sphere's area have been discarded using the **`discard`** command. This instruction has an immediate effect: the corresponding pixel isn't written to the **framebuffer**, meaning any calculations performed for that pixel are discarded and don't appear in the final render. This behavior is useful for preventing artifacts, such as the previously mentioned noise, which occurs because the ray penetrates areas where the geometry or lighting hasn't been precisely defined, and the rays continue traveling without colliding with any objects.

This methodology of discarding pixels and saturating the lighting can be very useful when your goal is to display only the geometry or procedural shape that is intended to be visualized. However, it's important to keep in mind that **`discard`** isn't always the best option in terms of performance, especially when combined with more complex shading techniques or post-processing effects. In such cases, you should consider alternative strategies, such as using **Depth Buffers**. This approach would also fix depth-related issues that occur when trying to include another three-dimensional object (**Mesh Renderer**) in the scene.



(5.3.l SDF sphere being affected by light)

With this implementation, we've successfully optimized the visualization of our procedural sphere, eliminating unwanted noise and ensuring that the lighting adequately reflects the interaction between light and surface. In the next section, we'll combine the sphere with a capsule using the **`smin()`** function, enabling the creation of more complex and detailed shapes.

5.4 Uniting two three-dimensional Shapes.

The ability to combine three-dimensional shapes provides great flexibility for creating complex and visually engaging objects in a procedural context. In this section, we'll explore the integration of a capsule, following the definition in Section 5.1.f, and extend the **`map()`** method using the **`smin()`** function to achieve a smooth transition between both shapes.

To begin, you can go to the **Procedural3DShapes** script and add a new method named **`capsule_sd()`**, as shown below:

```
6  float capsule_sd(float3 p, float3 p0, float3 p1, float r)
7  {
8      float h0 = dot(p - p0, p1 - p0) / dot(p1 - p0, p1 - p0);
9      float3 h1 = clamp(h0, 0.0, 1.0);
10     float3 v = p0 + h1 * (p1 - p0);
11     return sqrt(dot(p - v, p - v)) - r;
12 }
```

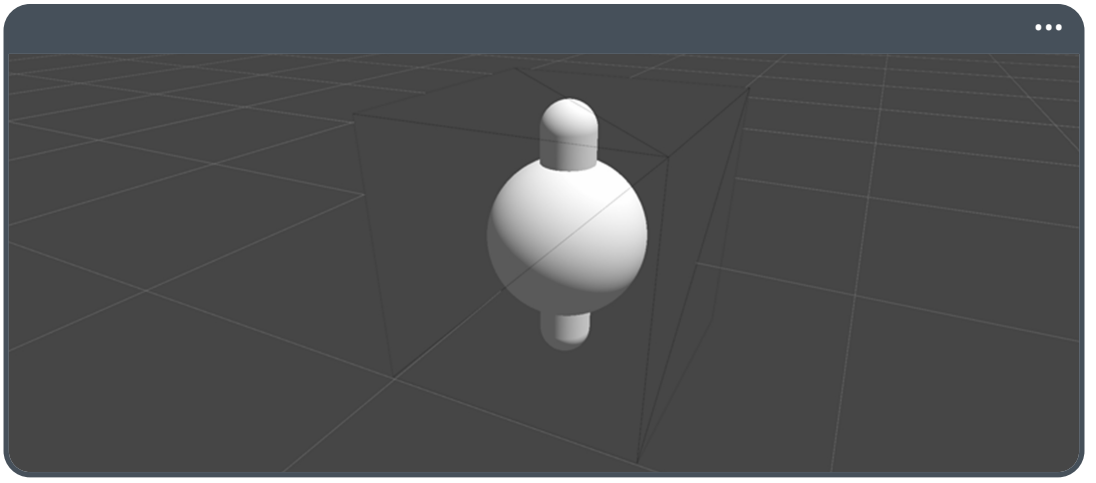
As you can see in the code above, **`capsule_sd()`** calculates the minimum distance from a point **`p`** in three-dimensional space to the surface of a capsule, which is defined by the arguments **`p0`** and **`p1`**—the capsule's endpoints. On line 8, the variable **`h0`** represents the scalar projection of the vector **`p - p0`** onto the capsule's axis **`p1 - p0`**. Then, on line 9, the value of **`h0`** is clamped between 0.0 and 1.0 using the **`clamp()`** function. This ensures that the projection remains within the segment defined by **`p0`** and **`p1`**, preventing it from extending beyond its endpoints. Line 10 declares a new vector **`v`**, which corresponds to the closest point on the capsule's axis to **`p`**. The final line computes and returns the Euclidean distance between **`p`** and **`v`**, subtracting the capsule's radius **`r`**, which gives the distance from **`p`** to the surface of the capsule.

To visualize this object in the scene, you need to return to the **RayMarching** script and navigate to the **`map()`** method, which is responsible for storing the procedural shapes in

the code. For rendering, you'll use some previously studied values to ensure its correct projection in the scene, as shown below:

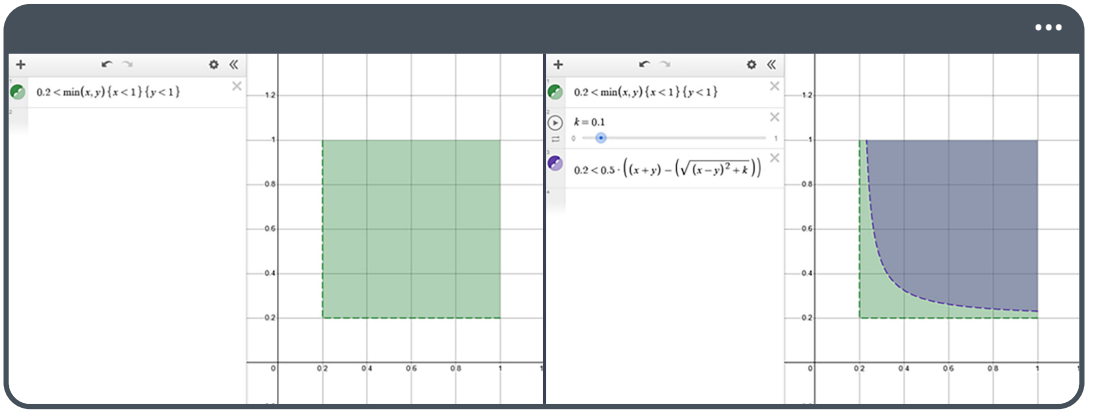
```
inline float map(float3 p)
{
    const float sphere_radius = 0.3;
    float sphere = sphere_sd(p, sphere_radius);

    const float capsule_radius = 0.1;
    const float3 capsule_p0 = float3(0.0, 0.4, 0.0);
    const float3 capsule_p1 = float3(0.0, -0.4, 0.0);
    float capsule = capsule_sd(p, capsule_p0, capsule_p1, capsule_radius);

    return min(sphere, capsule);
}
```

As you can see in the previous example, you've declared three new constant variables have been declared within the **`map()`** method. These refer to the capsule's radius (**`capsule_radius`**), its first point (**`capsule_p0`**) and its second point (**`capsule_p1`**). These variables are passed as arguments in the **`capsule_sd()`** method, which defines the three-dimensional geometric shape of the capsule. Finally, the **`min()`** function has returned the minimum between the sphere and the capsule, producing the following graphical result:



(5.4.a Minimum between the sphere and the capsule)

You can observe that the intersections between both figures appear sharp. This occurs because the **`min()`** function returns a value based on whether the first argument is less than the second. As a result, there is no range of values that produces a smooth interpolation between the two figures. To solve this issue, you can use a smoothed minimum function, which performs an interpolation between two values using a modified minimum operator. Unlike a sudden transition, this function introduces a smoothness curve, controlled by a parameter **k**, which determines the width of the transition zone.



(5.4.b https://www.desmos.com/calculator/zonkb4pky5)

As you can see in Figure 5.4.b, the reference on the left contains an inequality that checks whether 0.2 is less than the smallest values between x and y. The green region in the

Cartesian plane represents the values that are greater than xy. You can observe the same behavior in the reference on the right, with the key difference being that the **`smin()`** function returns a smoothed value which you can control using the variable **`k`**.

To understand this difference better, you can compare the default definition of the **`min()`** function with its smoothed counterpart, **`smin()`**.

The **`min()`** function.

```
float3 min(float3 a, float3 b)
{
    return float3(a.x < b.x ? a.x : b.x,
                  a.y < b.y ? a.y : b.y,
                  a.z < b.z ? a.z : b.z);
}
```

The **`min()`** function takes two three-dimensional vectors **`a`** and **`b`**, and returns a new vector where each component is the corresponding minimum between its arguments. This operation is useful for combining two signed distance functions, ensuring that the resulting shape represents the union of both figures.

The **`smin()`** function.

```
float smooth_min(float a, float b, float k)
{
    return 0.5 * ((a + b) - sqrt(pow(a - b, 2.0) + k));
}
```

On the other hand, the **`smin()`** function (or **`smooth_min()`** as shown in the example) smooths the minimum operation. As mentioned earlier, the variable **`k`** controls the degree

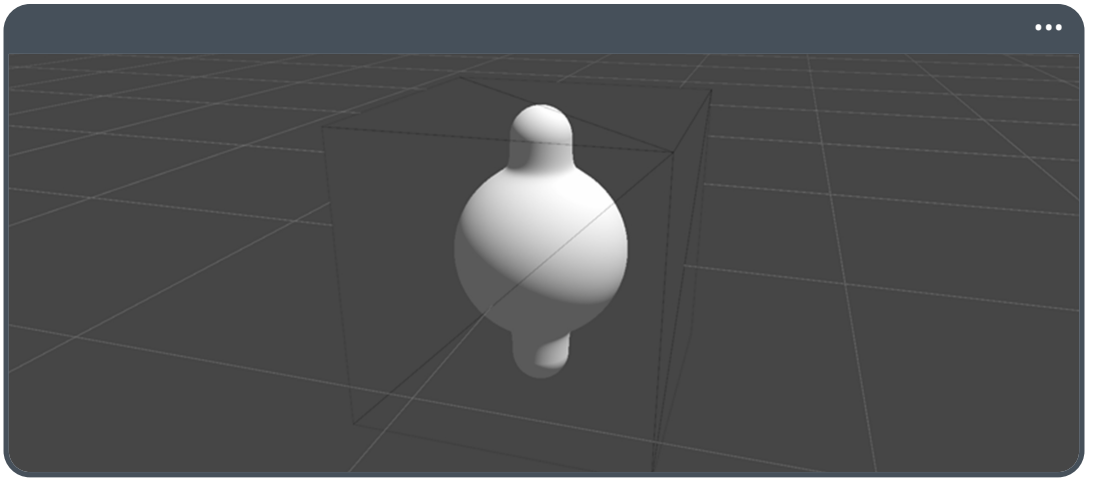
of smoothness: smaller values of **k** result in a sharper transition between the two shapes, while higher values create a smoother blend between them.

The main advantage of using **`smin()`** instead of **`min()`** lies in its ability to create smooth transitions between two signed distance functions. While **`min()`** produces a clear union but with sharp edges, **`smin()`** allows shapes to blend more organically, eliminating harsh lines and providing a more natural and visually appealing appearance.

To see this in practice, make sure you include the **`smooth_min()`** function in the **Procedural3DShapes** script, and then go to the **RayMarching** script. Inside the **`map()`** method, you must replace the **`min()`** function with **`smooth_min()`**, as shown below:

```
inline float map(float3 p)
{
    const float sphere_radius = 0.3;
    float sphere = sphere_sd(p, sphere_radius);

    const float capsule_radius = 0.1;
    const float k = 0.001;
    const float3 capsule_p0 = float3(0.0, 0.4, 0.0);
    const float3 capsule_p1 = float3(0.0, -0.4, 0.0);
    float capsule = capsule_sd(p, capsule_p0, capsule_p1, capsule_radius);

    return smooth_min(sphere, capsule, k);
}
```

On returning to the scene, you can observe that both figures—the sphere and the capsule—appear as one, giving the impression of a single, more elaborate shape.



(5.4.c Minimum smoothing between the sphere and the capsule)

Chapter Summary.

- In this chapter we delved into using signed distance functions to create three-dimensional geometries in Unity. We began by generating basic shapes such as spheres and capsules, establishing the mathematical bases necessary to model procedural shapes. We then addressed the calculation of normals using the signed distance function gradient, essential to give volume to shapes through lighting.

- Next we implemented the **Ray Marching** method within Shader Graph, creating custom nodes that allow efficient rendering of three-dimensional figures. After this we introduced the Lambert lighting model, for improving the visualization of shapes by calculating light intensity based on the angle between the normals and the direction of the incident light.

- Finally, we showed how to use the **`smin()`** function to achieve smooth transitions between two geometric shapes, allowing us to model a more mathematically elaborate figure. An interesting aspect was the use of the **`discard`** instruction to eliminate unwanted noise and ensure a clean representation of the shapes.

Glossary.

Anti-Aliasing: A technique used to reduce the appearance of jagged edges in digital graphics.

Blackboard: A panel in Shader Graph where global variables for nodes can be defined and managed.

Cartesian Coordinates: A reference system based on two or three perpendicular axes that allows locating points in space.

UV Coordinates: A coordinate system used in computer graphics to map textures onto surfaces.

Cosine: A trigonometric function that represents the ratio of the adjacent side to the hypotenuse in a right triangle.

Sine Wave: A graphical representation of the sine function, characterized by its periodic oscillation.

Custom Function: A node in Shader Graph that allows defining custom functions using HLSL code.

Desmos: An online application used for graphing mathematical equations.

Equation: A mathematical expression that establishes an equality between two expressions.

Function: A mathematical relationship between an input set and an output set, where each input has a unique output.

Graph Inspector: A panel in Shader Graph where node properties and configurations can be modified.

Hypotenuse: The side opposite the right angle in a right triangle.

HLSL: A high-level shading language used in computer graphics to write shaders.

Master Stack: The final set of nodes in Shader Graph that defines the material's appearance.

Material: A resource in graphics engines that determines how a surface is rendered based on shaders and textures.

Method: A procedure or function used in programming to perform a specific task.

Node: An element in Shader Graph that represents a mathematical or graphical processing operation.

Pentagon: A geometric figure with five sides and five vertices.

Procedural: A technique in computer graphics where data is generated dynamically instead of being pre-stored.

Radians: An angular measurement unit used in trigonometry and computer graphics.

Ray Marching: A rendering technique based on the progressive evaluation of a ray through a distance field.

Screen-Space: A rendering technique that operates in screen space, after the view transformation.

SDF: Signed Distance Field, a technique used to represent shapes through distance functions.

Sine: A trigonometric function that represents the ratio of the opposite side to the hypotenuse in a right triangle.

Shader: A program used in computer graphics to determine the color and appearance of pixels.

Shader Graph: A visual tool in Unity that allows creating shaders using nodes without the need to write code.

ShaderLab: A language used in Unity to define the structure of shaders.

Sub Graph: A graph within Shader Graph that allows reusing a set of nodes as a single node.

SubShader: A section within a ShaderLab shader that defines variations for different hardware configurations.

Tangent: A trigonometric function that represents the ratio of the opposite side to the adjacent side in a right triangle.

Trigonometry: A branch of mathematics that studies the relationships between angles and sides of triangles.

Unlit Shader Graph: A type of shader in Unity that does not respond to lighting, ideal for custom graphic effects.

URP: Universal Render Pipeline, a rendering system in Unity optimized for performance and flexibility.

Variable: A storage space in programming that contains a value that can change during execution.

World-Space: A coordinate system in computer graphics where positions are expressed relative to the global world space.

Special Thanks.

Seth Kohler | Margareta Winny | Sean R. | Bryce Lenhart | Joshua De Riggs | Brandon Fogerty | Aleksandar | Tori Holmes-Kirk | Sourav Chatterjee | Giacomo | Thomas | Joan Dalmau Alacreu | Marcus Wainwright | Eliza | Lucia Gambardella | Aleksandr Khokhlov | Pherawat Puttabucha | Lucia | Ash Curkpatrick | Aleh | Christopher Gough | David Clabaugh | Denis Smolnikov | Anne Postma | Andres Felipe Gonzalez | Minhson138 | Jasmin Daniel | Jonas Carvalho De Araujo | Jose Miguel | Robert | Sascha Henn-John | Raff | Andreas | Kieran Belkus | Housei Yoshida | Saeid Gholizade | Eyal Assaf | Gimli Damian Orawiec | Raghav Suriyashekar | Kylian | Carlos Meymar | Jona | Rafael Santos | Vincent Brunet-Dupont | Thuan Ta | Marc Destefano | Joey Green | Frederick Fowles | Kyle Avery | Adam Myhre | Kieun Mun | Alberto García González Javier C.M. | Josue Mariscal | Ngô Việt Luân | Michelle Moreno | Benjamin | Simon Kandah | Nico | Jaša Mihelčič | Jargueta | Kevin | Daniel Ocean | Yves | Etana | Max | Yosuke Ito | Tucker Cool | Robyn Wadey | Pistiwique | Noboru | Dgmpixy | Genna Zerzan | Ilya Kobzev | Aaron Fernandes | Randi Reynolds | Dougie Kerr | Kevin Roussel | Antonio Alonso | Ethan Smith | Yeove | Karl Fee | Jude Alonge | Alexandre | Sandro Ponticelli | Luis Raúl Castillo | Kidon Kim | Kevin Hernandez | Geonhan Lee | Michel Guenette | Fastzhuzhu | Dongsub Woo | Minh Triet Triết | Samuel | Jordan Bartlett | Parsue Choi | Wayne Kenneth | Joseph | Stefan Stefanov | Allen | Popper.

Heidi Borge | Deear | Marvin | Daniel Kaczkowski | Stuart Pentelow | Whm | Basile Lecouturier | Charlie | Take5Studios | Siarhei Ladychyn | Dave | Mugunth | Dinesh | Beomhee Lee | Marco | Paul Pinto Camacho | Alex Maximovich | Kyryl | Fang Ye | Marcin | Simone Ippoliti | Albin Lundahl | Ehb | Lauren Dunn | Steve Barr | Martin Frykler | Huỳnh Đông | Alex | Yuichi Matsuoka | Quim | 李谢谢 | Gnoqui | James Price | :3 | Benjamin Russell | Nick Boyd | Roberto | Anis | Adam | Nate | Benjamin Bouffier | German Rios | Luis | Marc Schaffer | Nikita | Bozhidar | Mimi Chio | Eduardo Roa | Gomorrah | Lars Devold | Darko | Matheus | Michael Antonio Velasco | Max | Will | Jake | Hardik Mistry | Juan Ramon Ramon | Ray | Billy Zhu | Cole | Sara | Aymeric | Matheus Araujo De Carvalho | A | Roshan | Pablo José De Andrés Martín | Willy Campos | Stefan | Bjarne Jørgensen | Jose Contreras | Irving

| Vincent | Rayn Olsen | Joe | Neruky | Robert | Michael Ha | Xun | Ke | Jon Rahoi | Alejandro Ruiz Ferrer | Vy | Cazchir | Adina Klein | Tim | Diego Xr | Max | Pedro | Dimas Alcalde | Andrew | Annabelle | Francois Bertrand | Mark Van Der Wal | Daniel Corzo Gonzalez | Ilya | Nicholas Guichon | Davidlopezdev | Kristoffer | Dayman.

Atte Vuorinen | Davon Allen | Marc | Christos | Daniel Oliveira | Nikos Kontis | Reyes | Scott Rays | Claudio Grassi | Theo | Liyi | Elsie | Ludovic Mantovani | Land Patricio | Avi | Phoebe | Ossama Obeid | Claudiu Barsan Pipu | Denis | Casilda De Zulueta | Paul Drummond | Alex K | Morgan Murphy | Afonso Cunha | Marina Henzenn | David | Denis | Sean Duggan | Davit Badalyan | Manno Bult | Amit Netane | Chance | Shiri Blumenthal | Jason | Hugo Delgado | Chiepomme | Clément | Lee Gramling | Joel Freeman | Sergei | Alexandre9 | Ricardo Díaz | Natalia | Ben Kurtin | Rúben Dias | Chen Si-Yu | Eduardo Rocha | Ariel Andrade | Matthieu Cherubini | Carsten | Juan Van Litsenborgh | Wonkee | Roberto Bianchini | Dionysus Acroreites | Siiri | Yurii | Aleksandr Kostochkin | Wai Teng Kwan | Joseph Leybovich | Miou Ng | Greenish | Ivan Shtuka | Jon | Gurtha | A. | Byeong Jun Kang | Romain Paget | Yu Jen Lu | Jimmy Lotare | Laurens Peeters | Dream Games | Khoa Mai | Luka Stefanovic | Thaunter | Flavien | Giovanni | André Gonçalves | Michele Raneri | Vittorio Durin | Dustin | Sev Wojtuś | Sergei | Anis Hadjari | Fernando Labarta | Paweł Ostrzołek | Matt Strangio | Damien | Alexandre Barré | Christopher Von Bronsart | Burak Soylu | Christopher Jackson | Kim Jin Sung | Eric Nersesian | Mikhail | Glen | Dana Frenklach | Kritsana | Landon Fowles | Andres Rueda | Diego Ferreira.

Martin Martiez | Kristaps | Paulo | Bence Hári | Anthony Davis | Diane Aveillan | Mike | Jack | Adam | Sungkuk Park | Andrew | David Moscoso | Bossa | Li | George Offley | Raveen Rajadorai | Shay Krainer | Adam | Chad Allen Josewski | Gerald Kelley | Damien | Ivan | Dominique Sandoz | Owen Magelssen | Marta Taszmowicz | Eliezer | Maxeee | Kane | Peter Hanshaw | Michael Parrish | Casey Dahlgren | Pedro | Guilherme Froes | Leslie Solorzano | Lucas Gustavo Schermak Alves | Cristian | Jen Huffa | Adalberto | Sean | Foosone | Bryce Dixon | Sandra | Bahaa | Simon | Justin Khan | Amir | Vincen Nguyen | Robin | Viviana | Bill Kastanakis | David | Marko | Jackson | Brittany | Razvan Luta | Slava Burlo | Anouk Van Uffelen | Mille |

Remi Rémi | Mille | Jacob | Pavetra Ltd | Julie | F | Alper | Goran Aleksic | Merwyn Lim | Syama Mishra | Alex Nechifor | Khaled Ahmed Younes | Benedict Chew | Johan Ouvrard | Hideo Daikoku | Ohmdob | Ryan Murdoch | Alessandro | Doxan | Thomas Surin | Thomas Schienagel | Justin Castonguay | Roman Boryslavskyy | Rodrigo Ramirez | Keith Guerrette | Madbrox | Francisco | Erdei Benjámin | Paulus | Pedro Sarraf | Tj Marbois | Cheng Yen Hsieh | Srslytrash | Thai Binh Nguyen | Marc | Claudia | Trevor | John | Nicolas Drew | Colter Haycock | Juan Camilo Hernández Ramírez | Jm.

Andrew | Francotzar | Francisco Javier Tinoco Pérez | Andriy | John Mckenna | Nigel | Billy Kwok | Paolo Orabona | Jean-Noel | Dimitrios Evgenidis | Pierre | Suresh U V | Osvaldo | Álvaro Colom Vidal | Stephanie Anderson | Domingo | Gaetan | Nicolás | Matan Poreh | Adalberto | Aleksei Mishchuk | Michael Woo | Ahn Jinuk | Grace Hsu | Geoffrey Legenty | Anil N | Serena | Willian Jefferson Freitas Da Silva | Alexis Emmanuel Lozano Angulo | Theo | Cong Obato | Lee Parks | Alcordev | Shanjing | Jonathan Westfall | Benjamin | Khemathat Buadam | Matthew Moldowan | Michelletai | Brandon | Level Ex | Chaos | Anfrollex | Rovane | Kimjaeyong | Kj | Mps | Mateusz | Dennis Schau Andersen | Novulus | Dillon Broadus | Simo Ahola | Supakorn Prasertsomboon | Craig | Corey Smedstad | Danny | Valla Folkhögskola | Julien De Marchi | Eliane Raymond | Román Marín | Diego André | Nikos Asfis | Pamela Melgar | Nick Fuhrberg | Enya | Mathilde Thauvette | Scott | Nate | Lazy | Arnaud | Jully Kado Mercado Elias | Matt Clinton | Ryan Dziurgot | Maciej | David Riewald | Heng Woon | Erynsen | Lachlan Dodds | Yan | Vlad Kononkov | Isaac Koerbin | Gabor Tornyos | Zeyad Kurdi | Andrew Hunt | Samuel Diaz Reyes | Daniel | Lauren | Serhii | Bill Kladis | Matheus Costa De Oliveira | Ser | Luis Martell | Dr Hogue | Sniperfolk | Paxlen | Daniele | Francesco Di Ruscio | Bertrand | Horaci Cuevas | Sergei Pavlovich.

Quintus | Vineet | Felix | Jaron | Hussien | Hennequin Angélique | Rozalia | Hakujin | Josh Kerekes | Cristian | Wolfgang | Caleb Brown | Jet | Michael Fernandez | Matthew Burgess | Andy | Thomas Jackson | Joshy M Raj | Minus1Player | Hayden | Emilian | Ulas Tosun | Yauming | Jesus Angarita | Juan Martinez | Mario | Stephan Maier | Tommy | Daniel Fairgrieve | Thom | Emiliano Guzman | Attila Sztankovics | Silvano Junior | Resistance Studio | Francisco Castillo | Carl

Boisvert | Oboro36 | Andrii | Michael Joseph Oakes | Laura | Peter Celko | Ruby Loudin | Miguel Nogales López | Lincoln Jones | Chris Walch | Jacob | Breno Aguirres | Michel | Seth | Chu Seyoung | Kaitlyn | Fidel | Giver | Nakayama Yasukazu | Jurriaan | Dakshesh Mamtora | Skylar | Chema | Nicholas Falcone | Mario Fatati | Sapha | Alina Sommer | Antonio Mata Marin | Fabian | Jahtani | Christos | Ice Code Games S.A. | Yu-Ruei Wan | Alvaro Luque | Saito Adrien | Yujen Chung | Luca Palmili | Aleksandra | Addison | Nick Schulz | Hortensia Costa Barcelos | Chris Janes | Szymon | Wenhsin Chang | Ramu | Tyler Drinkard | Dmitry | Zhang Siqi | Kimyh | Juan | Daniel Bruna Triviño | Snowdrama | Troy Donavin Patterson | Anthony Froissant | Benny | Zoe | Esther Felicies | Jonathan Martinez | Edgar Ulises Sánchez Izquierdo | Piotr | David Sparrow | Kaihatu | Harvey | Wei-An Chen | Jielin Sui.

Joseph Jolton | Aubrey Adin Mason-Park | Jake Evans | Pavel | Gabriel Pereira | Andrew | Oleg Fischer | Javir | Marcin | Declan | Todd Akita | Jim Rosson | Diego Moreira | Alireza | Thiago Laranjeira | Eric Rico | Ivan | Suman Kumar Singh | Johan Herrström | Dung Nguyen | José C. Montero Dávila | Codey Huntting | Danila Kiriukhin | Diego | Cyrille De Monneron | Gustavo Vitarelli | Vuvy | Biaaa | John Warner | Hotaru Kunstmann | Mario Ampov | Evgenii Nikolskii | Tom | Claudio | Jaehyeok Hong | Bluepained | Ilianette | Andrés | Sabrina Mini | Alvin | Anthonye | Trevor Harron | Ryan Van Cleave | Parashivamurthy B N | Shaun | Vincent | Graeme Fotheringham | Brent Elliott | Vitaliy | Milad Nazeri | Leonardo Amarillo Cantor | Josué Rivas Diaz | Andrey Skobelev | Ian | Ian | Peterchen | Joseph Lu | Yaroslav | Mike Marrone | Andre Castel | Marllon Vilano | Kiryu Yamashita | Kirthi | Rafa Mota | Mateusz | Tomasz Borowiecki | Ivan | Daniel Sarmiento | Abraham | Gigahood | Ewetumo Alexander | João Pereira Lemos Costa | Hwang Tae Gyu | Nguyen Huu Tin | Mohamed Al Hosani | Emmet | Ethan | Ron Gilmore | Emilie | Victor | Brooklyn Chen | Angel Camacho | Scott | Amy Ho | Moran | Atle | Sling | Tauxr | Carlos Melero | Cliff Cawley | Costin Vladimir | Chergui | Jonathan Le Faucheur | Rodrigo | Mark Fernandez | Cannaan John | Lidia Martínez | Ten Square Games | Jarukit Chanchiaw | François Giraud.

Nick | Akashcastelino Castelino | Towazumi | Mario Flores | Minh-Tri Nguyen | Ryan | Adrian Vides | Matt | Wabbite | Nicholas Young | Jaiwanth | Mathieu

Dufresne | Valentyn | Jasmin Skamo | Oneleven | Tanner | Ricardo Duaik | Filip | Alex | Akshit Pandey | Dale Newcomb | Shaun Jennings | Daniel | Michael Mcardle | Amelys | Ida Fontaine | Paulo Henrique Braganca | Maximilian Neubert | Jie Guan | Claudio Grassi | Alan Thorn | Ines | Charli | Ruchir Raj | Avishai Menashe | Nikolai | Yonatan Tepperberg | Oleg | Jordi | Kate | Markus Hamburger | Beck | Alexander | Victor | Angelo | Uros | Matt Dilallo | Luca | Eduardo Albino Gonelli | Phoenix | Fadii | Gerson Cardenas | Marchel | James Romero | Kylelle Wesley Chua | 盧朗正 | Fredrik Walldoff | Hadi Danial | Aran Ahmed | Andrii | Saku | Pakpoor | Brian | Ben Brown | Tyler Molz | Isuru Vithanaarchchige | Sungkoon Park | Egor Gostyshev | Biel Serrano Sanchez | Nicolas | Nacho Molina | Robert Northrup | Camilo Correa Rojas | Florian Machill | Lucimar Losi | Kim Dongjin | Daniel Carswell | Dominik | Bence Bordas | Mana | Chase Holton | Simon Swartout | Stephanie | Kode | Garrett | Hao Ye | Gosia | Mariusz Staszewski | Mark | Paulo Vilela | Esteban Meneses | Morgane | Malika | Romain | Alexander2010 | Jessie De Jong | Darius Reginis | Charlie Baldwin | Kelvin Put | Rafael De França.

Nibanez | Yusuf Ulutas | Jasper Lockwood | Philipp Forstner | Jin | Timothee Engel | Oliver Giddings | Graham | Dusan | Alex | Mainee | Cedric | Eduard Dobermann | Joshua Davies | Ji Eun Jeong | Henrique Fickert | Leo | Alex | Astler | Robingrotenfelt | Dominic | Luke Hastrup | Pablo | Jerome Lim | Paola Neyra | Kuuo | Sam | Barbora Kubišová | Alina | Matheus Freitas | Nicolas | Simone Odoardi | Zer0 | Ferdinand3D | Mark Schnoebelen | Jemma | Francisco | Md Zayed Al Sajed | Anton | Paritta | Neurony Solutions | Romans | Sula | Ryuichi Kawano | Axel Fransson | Michael | Semyoung Ahn | Russ Mcmackin | Dima | Arron Townshend | Enzo Hasegawa | Zouzal Khalid | Kristijonas Jalnionis | Tymoteusz Kaluzienski | Mane | Joepher Hung | Juan Alzate | Joe | Thirtysix Softworks | Orlando | S Parker | Ayman Horani | Tralyn Le | Olivier Beierlein | Tucker Lein | David Crooks | Igor Gustavo | Ryan Yassin | Jason Stark | Troy Richardson | Mikhail | Vinicius Pereira | Simona Sanin | Alexander | Pavlo | Chow Yik Khang | Jonathan Thorpe | Adrián | Pihla | Silver Wolf | Marc | Vitalii | Max | Théo | Alex | Ed Whitehead | Noah Williams | Jinzhongyin | Sterling J Jamaal Stokes | Carlos Aldair Roman Balbuena | Omri Yaloz | Caroline Hernandez | Anh Minh |

Lee Rosenbaum | Skyron | Keith | Alexander Konovalov | Jonathan Rivas | Janie Larson | 최재영.

Colton Pierson | Jan-Niklas Burose | Devon21 | Pery Manuel Silvestre Miguez | Ogmasoul3D | Olatz | Michael Hovland | Roger Ridley | Ahmed Shweiki | Ryan Pang | Terkel | Oleksandr Krotnyi | Santi | Walter Low | Matthew | Aleksandrs Grigorjevs | Guillaume Bernard | Ankit Rathore | Shounak Mandal | Mikheil Lomidze | Lucas Malek | 유태앙 | Milton Gustavo | Andrés Sánchez | Aniol | Thomas | Martina | Vladimir | Giovani Ramirez | Michael Fewkes | Nick | Tanguy | Ken Ichi | Dman22 | Eiyaphat Suntiwong | Hyeon Jun Jeong | Quentin | Edgars Skrabins | Ian Sedgebeer | Silverer | Chirag Morab | Alexis | Hannah | Sandra | Carlo Harvey | Stephen | Alejandra Antequera | Devon Chiu | Athip | Chris Butler | Luke Hannigan | Jayesh Makwana | Houssem | Sepehr Arya Yari | Dongjin Jung | Gozin | Marko Rankovic | Mayur Rathod | Rei | Geppy Parziale | Kyle Li | Amir Ahmadi | Yo0N0 | Adrian | Laginis | Evan Williams | Sirmazius | Ivan | Warren | Will | Arthur Trio | Danielle Briskin | Bjorn | Sourav Chatterjee | Miguel Torija Montejano | Andor | Ken | Alfonmc | Alejandro Campbell Legarreta | Diego Hernandez | Eduardo | Vuk Mihailovic Takimoto | Hwanhee Kim | Nicolas Pointet | Ana | Viktor Van Hulle | Ben Puckett | Tristan | Giuseppe Modarelli | Ece Ozmen | Shuhei Iwamoto | Richard | John Nyquist | Fahrul | Ducvu Fx | Stephen | Arvind | Vladislav | Jun | Yc.

Jettelly wishes you success in your professional career.