Volume 2

VISUALIZING EQUATIONS

SHADER FUNCTIONS

FOR TECHNICAL ARTIST

A visual explanation book that will help you transform polynomial, exponential, trigonometric and other functions into shader language.

Written by
Fabrizio Espíndola

Designed by
Pablo Yeber


Jettelly

Visualizing Equations,
shader functions for technical art.

A book of visual explanations that will help you transform polynomial, exponential, trigonometric functions, and more into shader language.

Visualizing Equations – vol. 2, version 0.0.1.

ISBN XXX-XXX-XXXX-XX-X

Author.
Fabrizio Espindola.

Design.
Pablo Yeber.

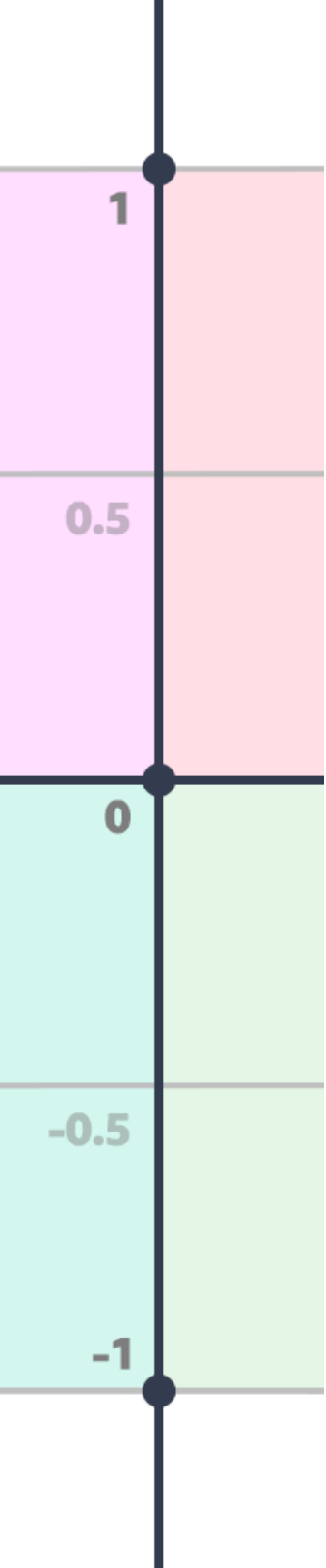
Technical review.
Martin Molina.

Grammar and spelling.
Martin Klark.

Acknowledgements.

I would like to express my gratitude and dedicate this work to my parents, Marcia Vivas and Victor Espíndola, for giving me life; my siblings, Angela Espíndola, Franco Espíndola, and Camilo Espíndola, for being there for me during challenging times; to my wife, Aida Cid, and my son Santino Espíndola, for being my light and motivation; to my friend and colleague Pablo Yeber; for accompanying me on this great adventure, and finally, to my mentors, David Sanhueza, Cristian Klett, and Ewan Lee, for guiding me with wisdom.

~ Fabrizio Espíndola.



Content.

Preface.

Visual effects are a constant presence in most video games worldwide. However, behind the magic of these effects lies a world of mathematical complexities ranging from simple linear operations to extensive logarithmic calculations.

Shaders are small programs that run on the Graphic Processing Unit (GPU) and contain color information for each pixel displayed on the screen. In this book, we will delve into the intriguing world of mathematical functions that allow us to bring shapes to life through shaders in this book, from representing points and lines to creating intricate figures.

Each chapter will take you deeper into the wide world of mathematics applied to technical art. We will approach concepts linearly and gradually, so there is no need to be a mathematical genius to proceed. Additionally, we will guide you through algorithms and programming techniques that will empower you to master creating visual effects, regardless of the graphic engine you are using.

Who this book is for.

This book has been designed for technical artists and programmers who want to deepen their understanding of mathematical functions applied to shader language. It is important to note that we use the Unity game engine and the Universal Render Pipeline to develop the content in this book. However, the information you will find here can be applied to any software that incorporates a graphic programming language, such as HLSL, GLSL, or similar.

Prior knowledge of polynomial, exponential, or trigonometric functions will significantly benefit your journey through these pages. Nevertheless, you do not have to worry if these concepts are new to you as no prior knowledge is not required. Each technical idea will be explained in detail throughout the book to ensure that any developer, regardless of their experience level, can delve into the exciting world of mathematics applied to shaders.

Conventions.

We have established some conventions to highlight certain elements and make the information in this book more accessible. These conventions include using angle brackets to emphasize functions, methods, and variables, as well as the formatting of numeric and code variables. Additionally, we use capital letters to represent spatial axes.

- **Highlighting elements:** We have chosen to enclose functions, methods, and variables between angle brackets (e.g., « **Frag** ») to emphasise their importance and technical nature.
- **Numeric and code variables:** Numeric and code variables are primarily written in lowercase (e.g., « **a** ») to help distinguish between technical variables and numeric variables.
- **Spatial axes:** Spatial axes, such as coordinates, are written in uppercase (e.g., « XYZ ») to facilitate the identification of elements related to space and geometry.

Throughout the book code blocks are presented in a unique format:

```
4    #ifndef CUSTOM_FUNCTION
5    #define CUSTOM_FUNCTION
6
7        Vvoid Method_half (in float Inp, out float Out)
8    {
9        // Code here …
10   }
11
12   #endif
```

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

Errata.

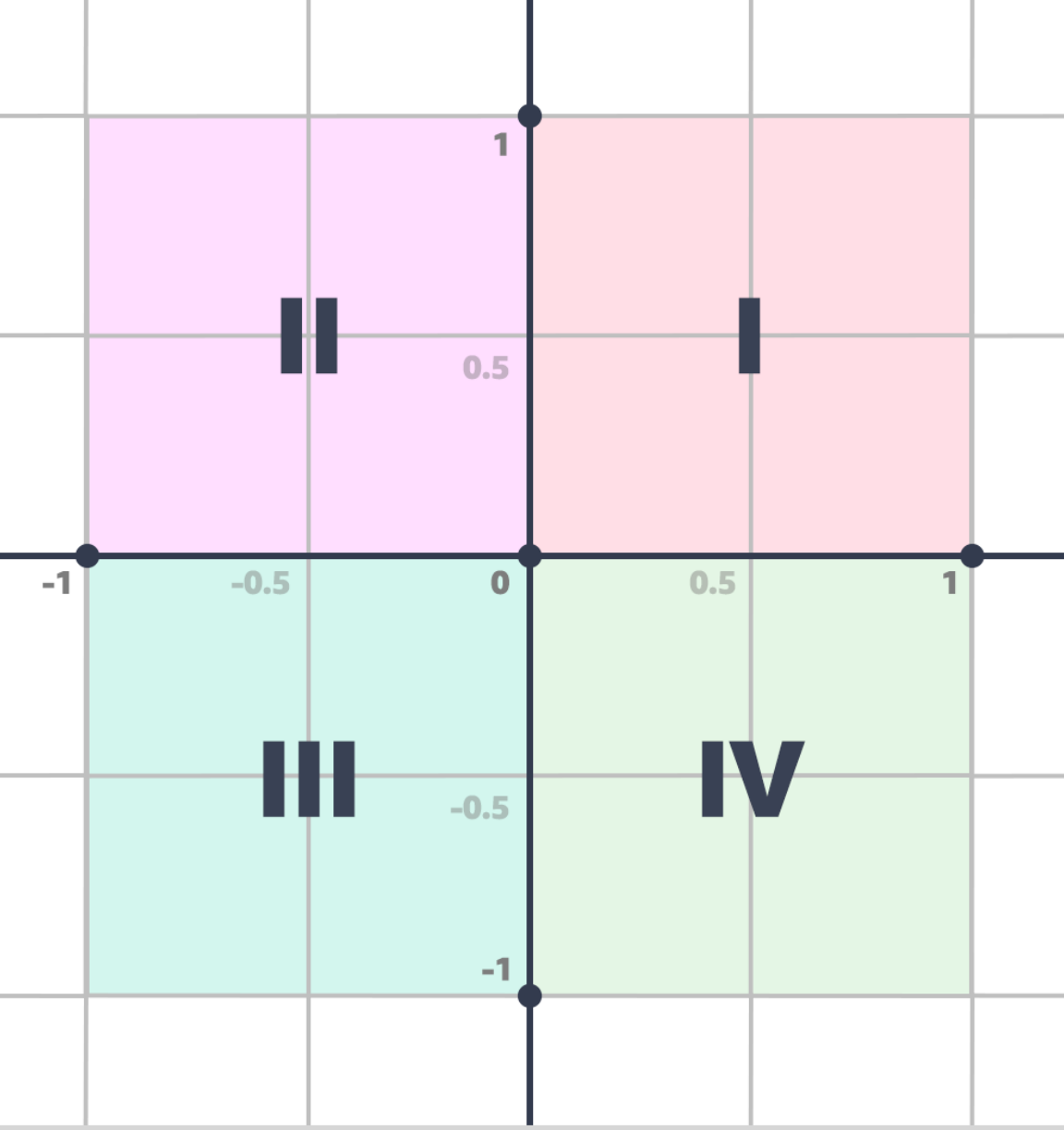
While writing this book, we have taken precautions to ensure the accuracy of its content. Nevertheless, you must remember that we are human beings, and it is highly possible that some points may not be well-explained or there may be errors in spelling or grammar.

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.

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

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



Chapter 1.

Polynomial functions.

This chapter explores the transformation of various mathematical functions into shader code to graphically represent shapes on the screen. Polynomial, exponential, and trigonometric functions will be analyzed and applied to « **UV** » coordinates. We will use the HLSL language to do this in the Unity 2022.3.8f1 LTS environment within a 3D project and in conjunction with its Universal Render Pipeline (URP), version 14.0.8. URP will allow us to approach each aspect intuitively through nodes and visually observe the graphic behaviour of each function in action.

Linear Function.

Linear functions are valuable in fields such as statistics, economics, and social sciences. However, their utility extends even further, reaching into the exciting world of video games. In computer graphics, linear functions play a fundamental role in drawing straight lines, becoming essential for creating many visual effects, including transitions between scenes. Now, the question is: How can you do this?

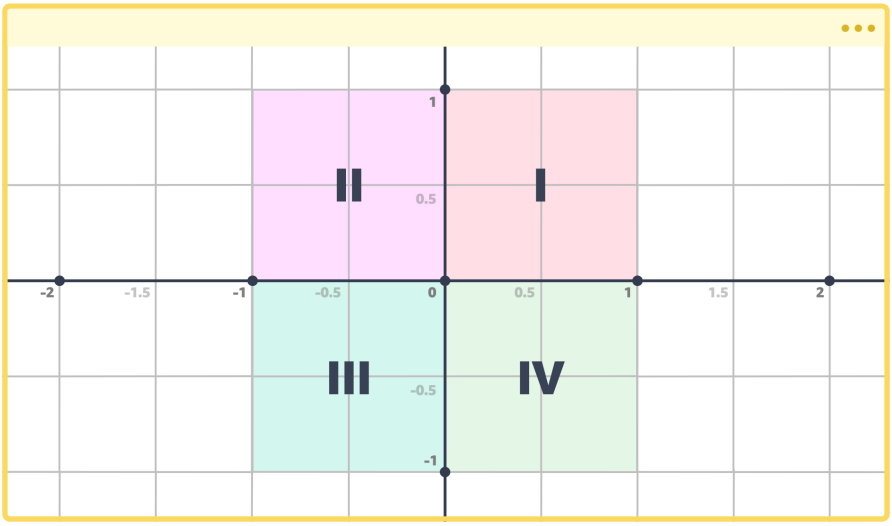
To understand the nature of this process, focus on the analysis of the following equation:

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

(1.1.a)

In Figure 1.1.a above, the variable « m » represents the slope, and « b » represents the point at which the line intersects the « Y » axis, that is, the value the function takes when « X » is equal to zero. However, to fully understand this function, it is crucial to delve into the world of Cartesian coordinates.

Cartesian coordinates form a system of orthogonal rectangular coordinates that span Euclidean space in two dimensions. The definition of these coordinates may include a third dimension called « Z », thus becoming a fundamental tool for constructing three dimensional models. Using the axes of Cartesian coordinates, you can subdivide 2D spaces into a set of subregions known as quadrants (four regions), which are convenient for precisely defining functions when working with other coordinate systems. Cartesian coordinates are represented by two real values, « X » and « Y », which determine the location of a point relative to defined axes.

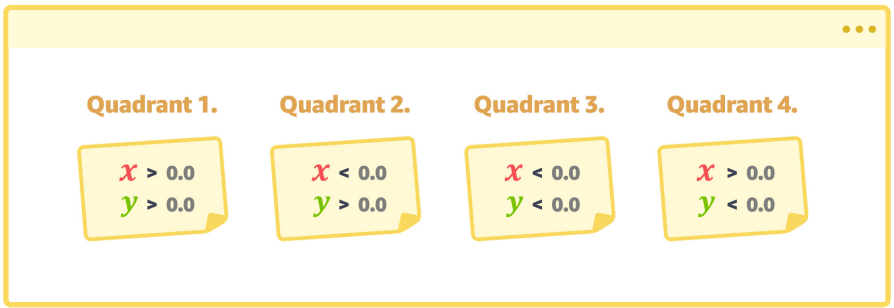


(1.1.b)

This initial stage focuses exclusively on two-dimensional visualizations. The Desmos graphic calculator is used as the primary tool for mathematical calculations and can be found at the following link:

- **https://www.desmos.com/calculator**

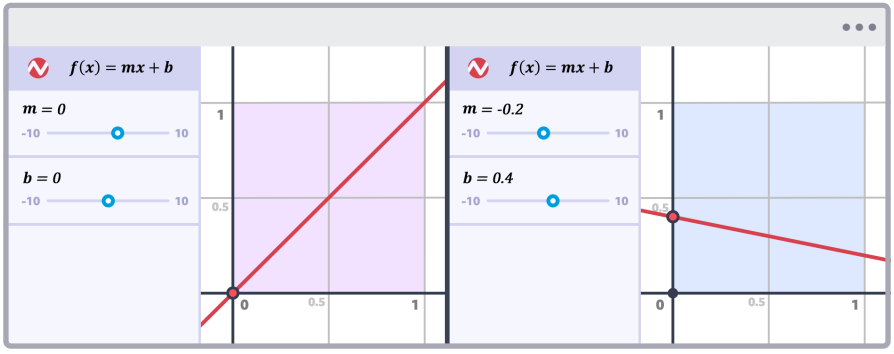
As illustrated in Figure 1.1.b, different colors have been used to clearly distinguish each quadrant. An important aspect of these quadrants is that depending on their location, the coordinates « X » and « Y » present changes in their sign, meaning they can be negative or positive.



(1.1.c)

Another interesting aspect is that « **UV** » coordinates span the first quadrant by default, meaning they start at « $\mathbf{0.0}_x, \mathbf{0.0}_y$ » and end at « $\mathbf{1.0}_x, \mathbf{1.0}_y$ ». Since we will late apply this knowledge to the HLSL language, it is essential to have a solid understanding of these concepts. Why is it so important? Otherwise, we will be unable to comprehend how to draw a line in « **UV** » coordinates in a specific position.

The function in Figure 1.1.a is shown on the Cartesian plane as follows:



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

As you can observe in Figure 1.1.d, the slope « m » represents the number of units by which « Y » increases or decreases when « X » changes, while the variable « b » indicates the distance from the origin. The 'origin' refers to the starting point on the Cartesian plane corresponding to « $0.0_x, 0.0_y$ ».

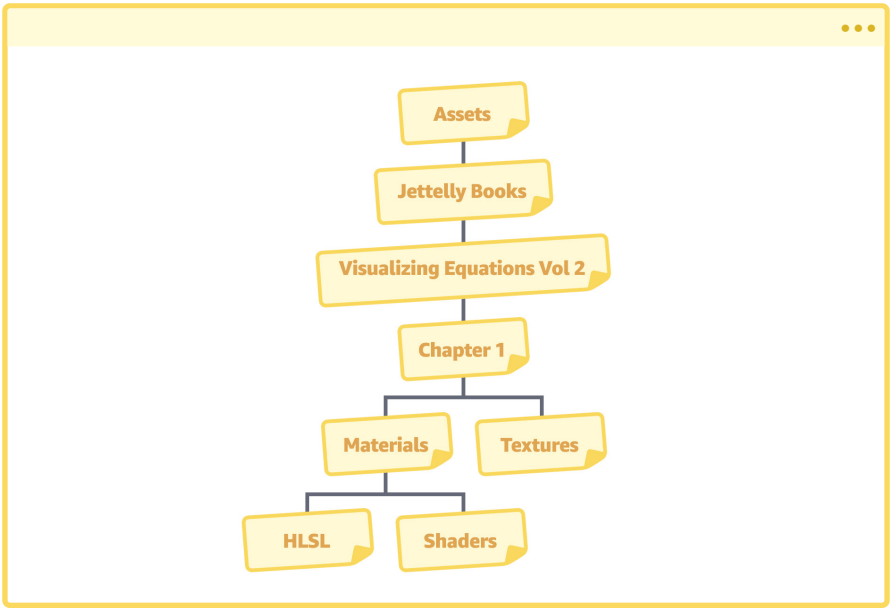
How can the function be centered around the position « $0.5_x, 0.5_y$ » within the Cartesian plane? To achieve this, extend the linear function and introduce a new variable into the equation, which must be subtracted from « X ».

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

As you can see in the previous Figure, a new variable, « u » with a value of 0.5 has been introduced into the function. When subtracted from « X », this variable causes a horizontal shift of the central point on the Cartesian plane. Performing this procedure is essential since, when working with shader, you often need to adjust the origin to « $0.5_x, 0.5_y$ » or draw lines from points different to the origin.

Visualizing the function through a shader.

Before embarking on the development of your shader, it is essential to establish an organizational structure for your project. Start with the following outline, which will expand as you progress through the book:



(1.2.a)

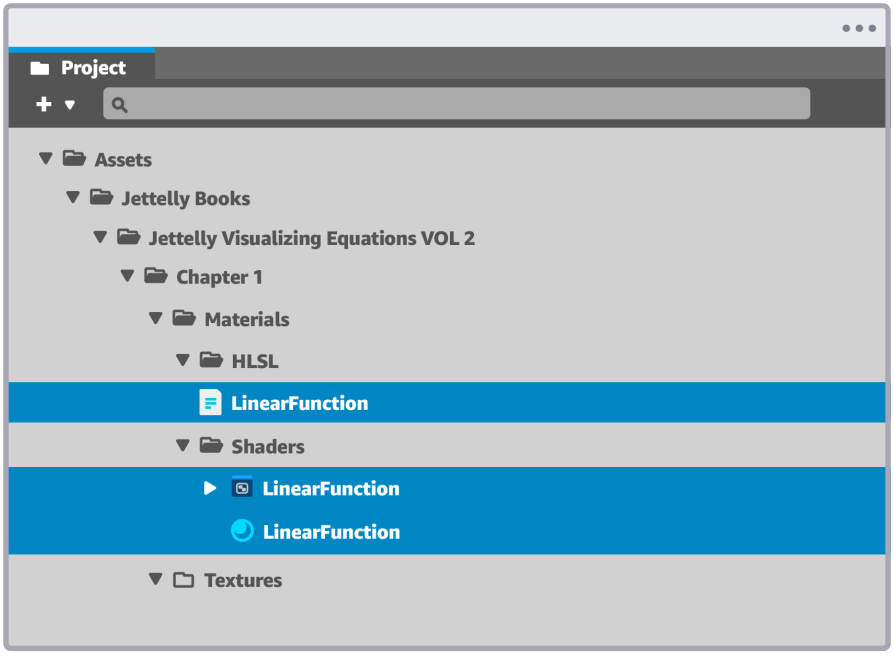
Next, create the following objects and place them in their respective folders according to their nature.

A shader named « **`Linear Function`** »: To create it, go to the main menu and select **Assets** > **Create** > **Shader Graph** > **URP** > **Unlit Shader Graph**.

A material with the same name as the shader: You can create this material following the same procedure as above, going to **Assets** > **Create** > **Material**.

A file with the extension « **`HLSL`** », also with the same name as the previous objects: Since Unity does not include objects of this type by default, you will have to create them directly from the code editor you are using or manually from the project folder. This involves navigating to the folder on your computer, creating a new text document, and changing its extension from « **`.txt`** » to « **`.hlsl`** ».

These three objects are created because you need to incorporate the graphical representation of the linear function into your project. These elements consist of a shader designed for the representation, a material that integrates the shader, and an HLSL file containing the necessary code for the function. If everything has been configured correctly, our project should look like the following Figure.

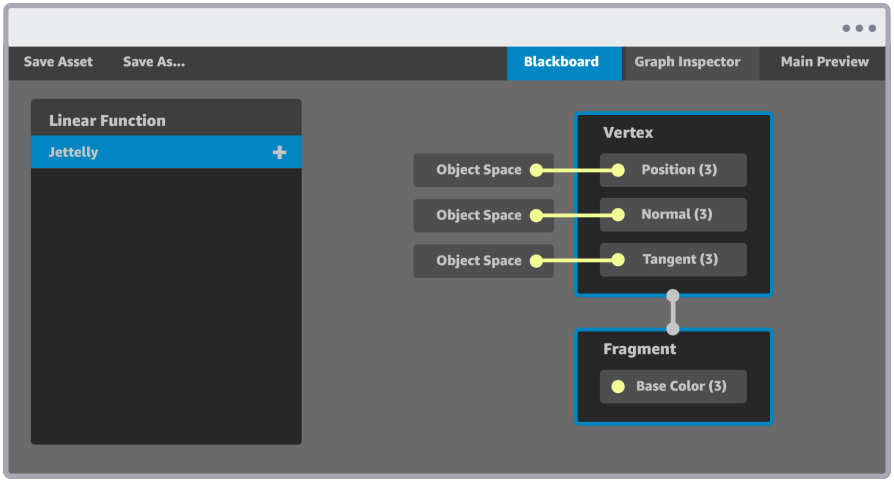


(1.2.b)

To do the exercise you need to perform at least three actions:

- Assign the shader to the material.
- Create a Quad object in the Hierarchy window.
- Finally, assign the material to the Quad in the Scene window.

This process involves several additional steps. For instance, to assign the shader to the material, first select the material, then look in the Inspector Window, and in the shader section, choose the newly created shader. However, the question arises: how can you find your recently created shader? For this, it is necessary to know its location. By default, it is located on the path: **Shader** > **Shader Graph** > **Linear Function**. You can verify this by opening the shader and selecting the Blackboard. The path should appear below the shader name, as shown in Figure 1.2.c. It is worth noting that for this exercise the path is configured with the name « **`Jettelly`** ».

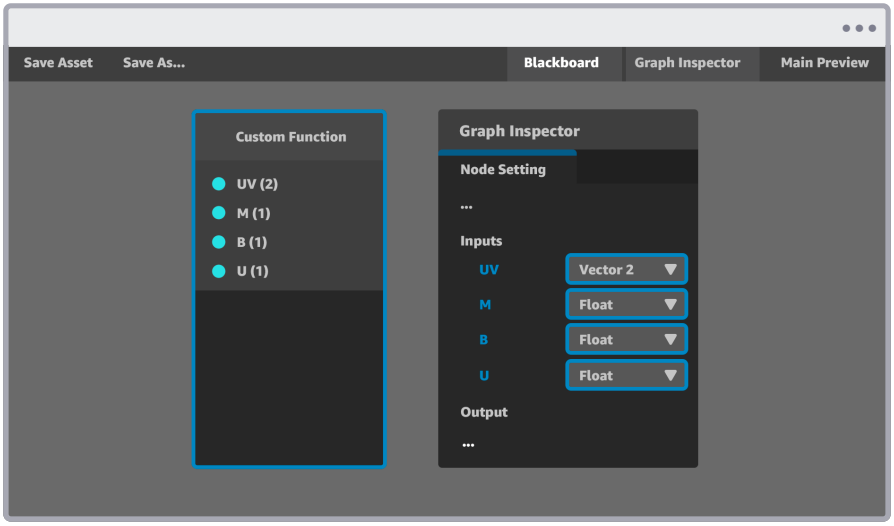


(1.2.c)

Since the linear Function does not have an independent node in Shader Graph, it will be necessary to use the « **`Custom Function`** » node to define it. To do this, press the 'space bar' on the keyboard and search for the « **`Custom Function`** » option. To fully understand how this node works, it is crucial to enable the Graph Inspector to define the node's inputs and outputs.

Regarding the inputs, you need to consider the variables defined in the function presented in Figure 1.1.a from the previous section, namely the variables « **`m`** », « **`b`** » and the « **`UV`** » coordinates. Also incorporate the variable « **`u`** », which is used to expand the function, as illustrated in Figure 1.1.d also from previous section.

When the Cartesian plane is referred to in Computer Graphics, it is either referring to global spatial coordinates (in the Euclidean world in the Scene window) or « **`UV`** » coordinates. In this case, opt for the latter: this way, it is easier to grasp the concept of « **`local space`** ».



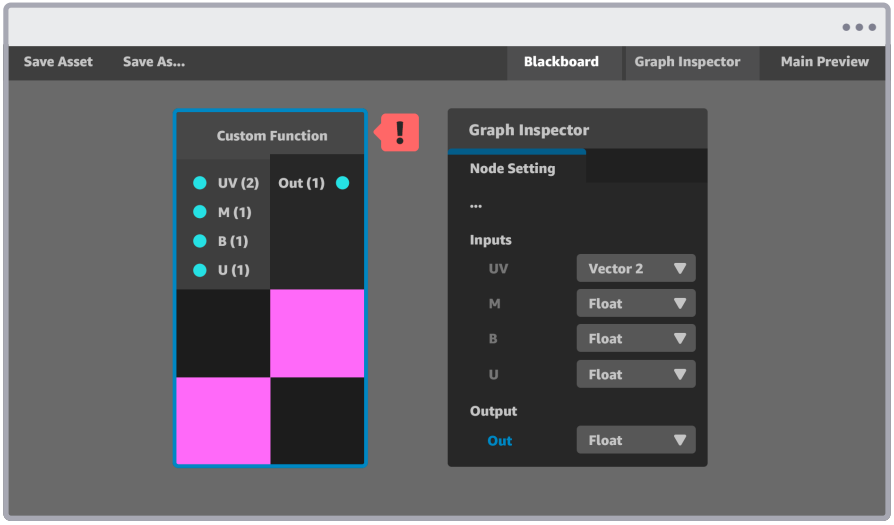
(1.2.d. com.unity.shadergraph@17.0/manual/Custom-Function-Node.html)

As shown in the previous Figure, the mentioned variables have been incorporated as « **`inputs`** » in the « **`Custom Function`** » node. It is important to note that the data type of the « **`UV`** » coordinates corresponds to a two-dimensional

vector because you will store the location of the pixels of your linear function in its « X » and « Y » axes. As for the « **`Output`** », it varies depending on the exercise we are performing. In this case, you will return a floating-point value determined by a conditional. This value will be equal to 1.0f or 0.0f, depending on a threshold that you can modify in real-time through the function variables « **`m`** », « **`b`** » and « **`u`** ».

It is worth noting that in Computer Graphics, values are used to create colors. Therefore, 1.0f and 0.0f can be interpreted luminance-wise as white and black, respectively, for each pixel. In other words, if the « **`output`** » equals 1.0f, pixels within the function's area will be displayed as white. In contrast, if it equals 0.0f, they will be displayed in black. Similarly, a value of 0.5f will result in a gray colour.

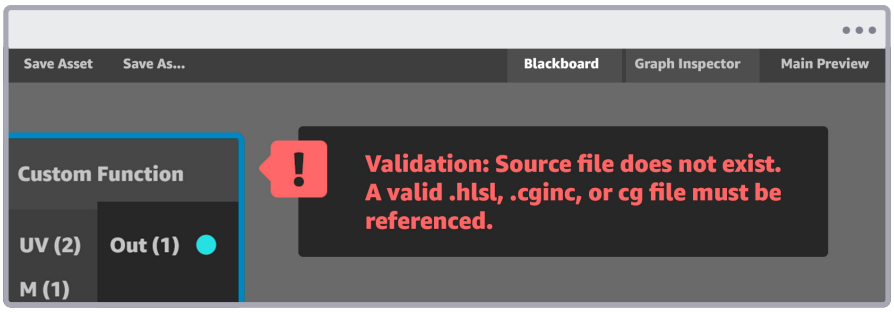
For this exercise, « **`Out`** » is used as the output name of the node, as shown in the following example:



(1.2.e)

One factor to consider in Shader Graph is that every time you change node configurations, such as adding or removing inputs, modifying the position of a variable in the list, or updating some functions in its HLSL file, common errors

can arise, ranging from validations to overwriting variables. Carefully examine Figure 1.2.e, you will notice an error that, in this context, is a validation error, this is because the script has not been assigned the « **`.hlsl`** » extension.



(1.2.f)

Next, follow these steps to ensure the node functions correctly:

1. Assign the file « **`LinearFunction`** » to the « **`Source`** » property of the node.
2. For the « **`Name`** » property, use « **`linear_function`** » to ensure consistency in your development.

The error may persist after making these changes because, unlike C#, Unity, in this case, does not add default code, and you have not yet defined a method for the node's operation. Consequently, declare the following method for the « **`LinearFunction`** » file, using the same name previously assigned to the « **`Name`** » property.

```
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   }
5
```

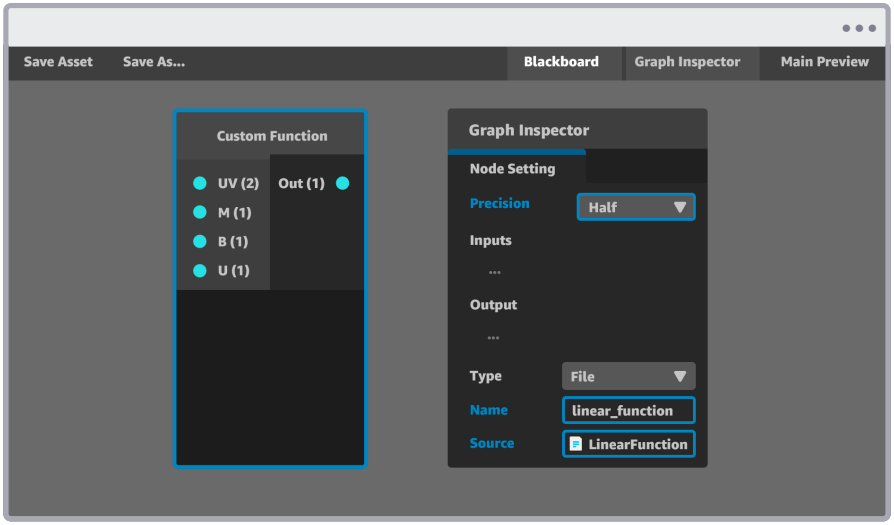
Examining the above code snippet, you will notice that the defined method and its arguments are of type « **`half`** », corresponding to a half-precision data type, i.e., 16 bits. The interesting aspect of this data type is that according to Unity's documentation:

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

Indeed, this function will generate only a few calculations on the GPU, making this data type unnecessary. However, for educational purposes, choose it and then configure your node to operate in half-precision.

Incorporating « **`half`** » into your program can potentially lead to performance improvements by reducing memory bandwidth, being more efficient in saving space, and even faster in arithmetic operations. However, it also has some disadvantages, such as precision loss, but this book will not delve into these details.

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



(1.2.g)

The black colour comes from the code's temporary output value « **`Out`** », which is set to 0.0f. The next step involves implementing the linear function. To begin, declare and initialize 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   }
8
```

Next, declare its function and initialize it using the scheme presented in Figure 1.1.a from 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  }
11
```

Looking at the code line 6 you can see that the linear function scheme has been implemented on the variable « **f** », involving a subtraction from the abscissa function « **fx** ». As mentioned before, now use the extended function for this example. Then, introduce the variable « **u** » into the operation in the following manner:

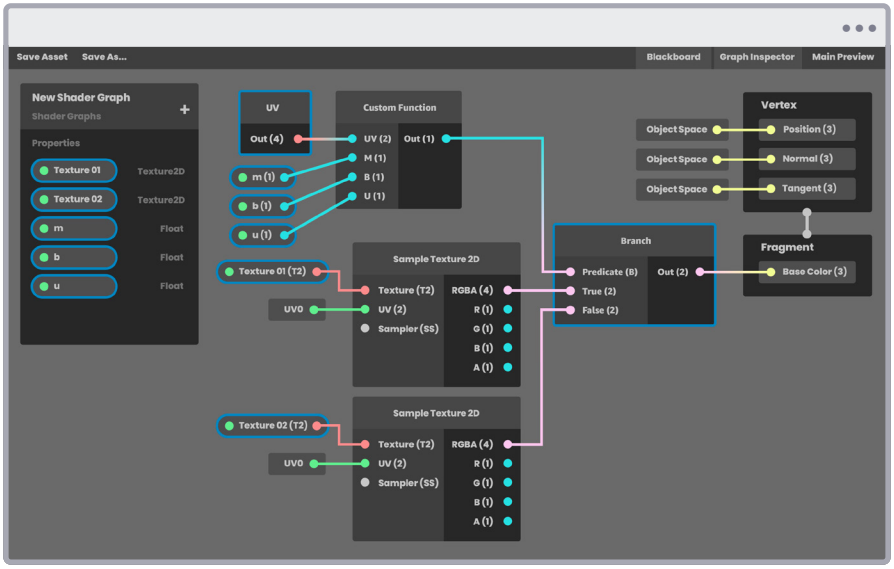
```
1   void linear_function_half(in half2 uv, in half m, in half b,
    in half u, out half Out)
2   {
3       half fx = uv.y;
4       half x = uv.x;
5
6       half f = m * (x - u) + b;
7       fx -= f;
8
9       Out = 0.0;
10  }
11
```

Upon examining line 6 of the previous example, it can be observed that « **u** » now subtracts from « **x** », enabling a horizontal shift of the central point in the equation.

Now, the only thing left is to define the output value « **Out** » of the function. To achieve this, do the following: return 1.0f or 0.0f if « **fx** » is greater than 0.0f. This way, you can project two textures and interpolate them based on the value of « **f** ».

```
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 - u) + b;
7       fx -= f;
8
9       Out = (fx > 0.0) ? 1.0 : 0.0;
10  }
11
```

If the preceding steps has been executed correctly, your node should compile without issues. Next, integrate the « **Branch** » node into your shader to visually represent the linear function visually. This node possesses the unique characteristic of returning two values, true or false, based on the « **Predicate** » property of type « **bool** ». In this scenario, use this function to project two textures that intersect according to the output value « **Out** », defined earlier in line 9 of the code. Before executing this process, you need to define certain properties in the « **Blackboard** » that will be used in conjunction with your « **linear_function** » node. Also, introduce the « **UV** » node, which serves as Cartesian coordinates, specifically, the coordinates of the first quadrant.



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

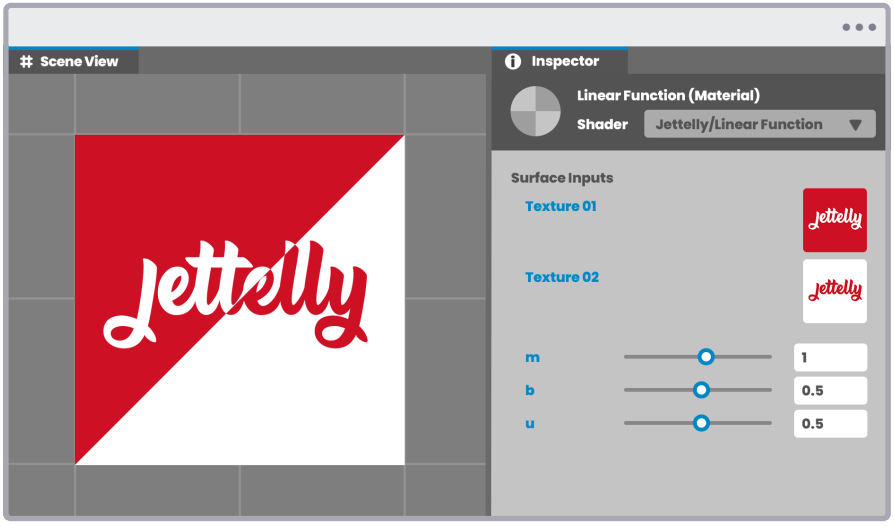
As seen in Figure 1.2.h, five properties have been defined in the « **BlackBoard** »: two textures, namely « **Texture 01** » and « **Texture 02** », and three float values, « **m** », « **b** » and « **u** ». Each of these values adhere to specific ranges:

- The « **m** » property spans a range between -10.0 y 10.0.
- 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 equivalences. Likewise, the « **UV** » node has been linked to the latter. Then, the output of the « **linear_function** » node has been connected to the « **Predicate** » input of the « **Branch** » node. Why has this connection been established? Primarily, it ensures 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 « **Fragment** ». By saving your shader and returning to the « **Linear Function** » material, you can adjust the variables of the linear function from the Inspector window. It is important to emphasize

that assigning two textures to the material is imperative for visualizing the operation of the previously mentioned function.



(1.2.i)

While adjusting property values, it becomes apparent that there is no image to help identify the origin point of the linear function. Therefore, you need to invest some energy in the creation of a new node for this purpose.

Jettelly Books!

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

Go to Books!