

Aprende. Crea. Domina.

Visualizando Ecuaciones – Vol. 2

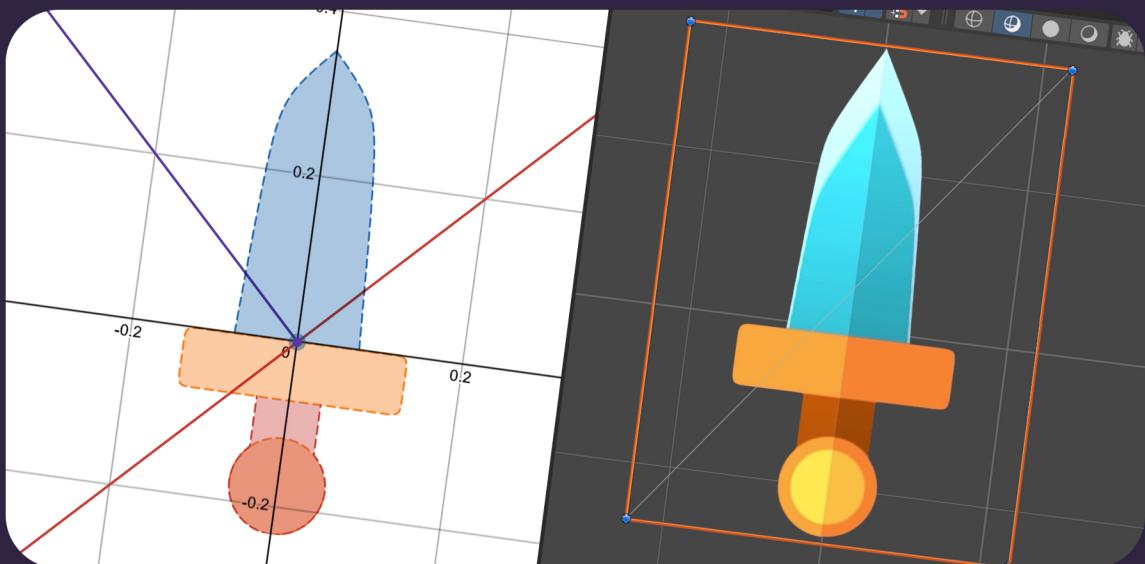
Shaders y Figuras Procedurales en Unity 6.

Una guía visual para transformar ecuaciones matemáticas en formas procedurales interesantes con Shader Graph.

Unity

GameDev

ProceduralArt



Fabrizio Espíndola.



Jettelly[®]

Visualizando Ecuaciones Vol. 2

Shaders y Figuras Procedurales en Unity 6.

Una guía visual para transformar ecuaciones matemáticas
en formas procedurales interesantes con Shader Graph.

Fabrizio Espíndola

#Unity

#GameDev

#ProceduralArt

Visualizando Ecuaciones – Vol. 2, versión 0.0.6d.

Publicado por Jettelly Inc. ® Todos los derechos reservados. jettelly.com

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

Créditos.

Autor.

Fabrizio Espíndola

Diseño.

Pablo Yeber

Revisión Técnica.

Martin Molina

Roberto Ortiz

Editor.

Ewan Lee.

Contenido.

Prefacio.	6
Para quién es este libro.....	7
Convenciones.....	8
Archivos descargables.....	9
Errata.....	9
Piratería.....	10
Capítulo 1: Funciones polinomiales.	11
1.1. Función lineal.....	12
1.2. Visualizando la función lineal en HLSL.....	15
1.3. Visualizando el punto de origen.....	29
1.4. Trazando una recta entre dos puntos.....	36
1.5. Función cuadrática.....	44
Resumen de capítulo.....	53
Capítulo 2: Funciones trigonométricas.	54
2.1. Funciones.....	55
2.2. Visualizando funciones trigonométricas en HLSL.....	65
2.3. Reflexión de un punto.....	77
2.4. Reflexión de un punto en HLSL.....	81
Resumen de capítulo.....	93
Capítulo 3: Figuras procedurales.	94
3.1. Analizando la forma de un Shuriken.....	95
3.2. Dibujando un Shuriken en HLSL.....	107
3.3. Implementando Anti-Aliasing.....	115
3.4. Definiendo una estética para el Shuriken.....	127
Resumen del capítulo.....	136
Capítulo 4: Función de distancia con signo.	137



4.1. Naturaleza de una SDF.	138
4.2. Dibujando un segmento SDF en la interfaz de usuario.	147
4.3. Análisis de la estructura de un pentágono SDF. . .	162
4.4. Reflexiones y transformaciones del pentágono SDF.	171
Resumen del capítulo.	177
Capítulo 5: Figuras tridimensionales y renderizado.	178
5.1. Añadiendo una tercera dimensión.	179
5.2. Normales en funciones de distancia con signo. . .	185
5.3. Renderizando una forma tridimensional.	193
5.4. Mezclando dos formas tridimensionales.	215
Resumen del capítulo.	221
Glosario.	222
Agradecimientos.	225

Durante mi carrera como artista técnico y programador, he observado que los aspectos técnicos del desarrollo de videojuegos son fundamentales para su éxito. Esto se debe a que la optimización (o parte de ella), las estructuras y la organización están directamente vinculadas a este tema. Por ejemplo, si deseas convertirte en un gran artista de UI, debes conocer qué es el espacio de pantalla (o screen-space en inglés), las coordenadas UV, los anchors y otros conceptos relevantes. Sin embargo, en la mayoría de los casos, artistas y diseñadores no se adentran por completo en estos temas, ya sea por falta de conocimiento matemático o por desconocimiento de gráficos por computadoras.

Este libro nace de la necesidad de comprender precisamente estos aspectos esenciales que hacen posible transformar fórmulas matemáticas en experiencias visuales y funcionales en el desarrollo de videojuegos, así como en procesos de investigación y desarrollo.

En **Shaders y Figuras Procedurales**, he reunido más de once años de experiencia en la industria para ofrecerte una guía visual y práctica que te ayude a dominar el arte de convertir ecuaciones en formas procedurales interesantes. Este libro está diseñado para llevarte, paso a paso y de manera lineal, desde los conceptos matemáticos básicos hasta su aplicación directa en Unity, uno de los motores más utilizados en la creación de videojuegos.

Cada capítulo ha sido estructurado para que puedas construir conocimientos de manera progresiva. Comenzando con el **Capítulo 1**, en el que descubrirás cómo las funciones polinomiales pueden ser la base para generar curvas y superficies, aplicables incluso a transiciones de pantalla. Continuamos con el **Capítulo 2**, donde profundizaremos en la importancia de las funciones trigonométricas, esenciales para modelar movimientos cílicos y patrones naturales. En el **Capítulo 3**, pondremos en práctica lo aprendido y desarrollaremos una figura procedural; un Shuriken, empleando únicamente funciones lineales y áreas geométricas. Durante el **Capítulo 4**, transformaremos funciones lineales en funciones de distancia con signo (SDF) en dos dimensiones, abriendo la puerta a efectos de renderizado más complejos y eficientes. Finalmente, en el **Capítulo 5**, extenderemos estos conceptos al espacio tridimensional, donde aprenderás a construir objetos y comprender la matemática detrás de cada uno.

Mi objetivo al compartir este conocimiento es que no solo adquieras una comprensión teórica, sino que también desarrolles habilidades prácticas que te permitan experimentar, crear y mejorar la calidad visual de tus proyectos. Estoy convencido de que la combinación de fundamentos matemáticos y shaders es la clave para innovar y dar un salto en tu carrera como profesional.

Para quién es este libro.

Este libro está diseñado tanto para artistas, como diseñadores y programadores con el afán de enriquecer su conocimiento técnico sobre la aplicación de funciones matemáticas en HLSL (High Level Shader Language), utilizando Unity 6000.0.29f1 LTS como motor gráfico. Nos enfocaremos en la versión 17.0.3 de Universal Render Pipeline (URP) dentro de un entorno 3D, aunque los conceptos y técnicas introducidas podrán aplicarse en cualquier plataforma que emplee un lenguaje de shader, incluyendo CG o GLSL.

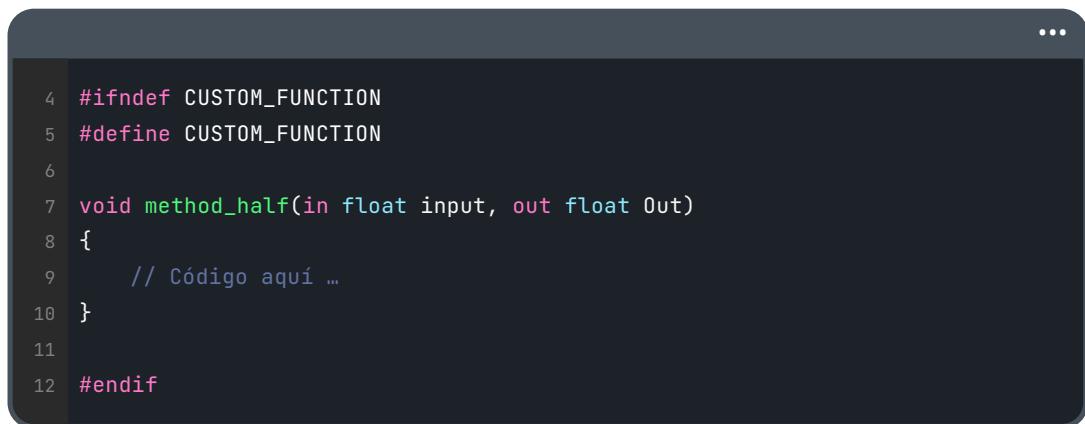
Independientemente de tu experiencia previa con funciones polinomiales, exponenciales o trigonométricas, este libro te guiará paso a paso en tu exploración de HLSL. Para quienes aborden estos conceptos por primera vez, se ofrecerán detalladas explicaciones visuales, claras y accesibles. Aspiramos a que cada lector, sin importar su nivel de experiencia, pueda sumergirse en el apasionante universo de la creación de figuras procedurales, descubriendo la potencia de las matemáticas como herramienta esencial en la generación de efectos visuales.

Convenciones.

En este libro, hemos definido una serie de convenciones para resaltar ciertos elementos y hacer que la información sea más accesible. Estas convenciones incluyen el uso de **negritas** para destacar términos técnicos, *itálicas*, corchetes y otros estilos. También utilizaremos letras mayúsculas para representar los componentes de un vector.

- **Resaltado de código:** Las funciones, métodos y variables se destacarán con una fuente específica que subraya su importancia y naturaleza técnica.
- **Funciones matemáticas y ecuaciones:** Se presentarán principalmente en *italica matemática*, lo que nos permitirá distinguir fácilmente un número genérico de aquellos que están directamente relacionados con el código o una operación específica.
- **Nombres y objetos:** Estos se presentarán en **negrita** a lo largo del texto.
- **Rangos:** Los rangos se presentarán entre corchetes (por ejemplo, [0 : 1]) en algunos casos para facilitar su lectura y comprensión.

Los bloques de código se mostrarán en un formato especial para mantener consistencia y legibilidad, como se ilustra a continuación:



```
4 #ifndef CUSTOM_FUNCTION
5 #define CUSTOM_FUNCTION
6
7 void method_half(in float input, out float Out)
8 {
9     // Código aquí ...
10 }
11
12 #endif
```

Estas convenciones se han diseñado para mejorar la claridad y facilitar la comprensión del contenido técnico, y se mantendrán consistentes a lo largo de toda la lectura.

Archivos descargables.

Con el afán de dar soporte y ayudarte a seguir los ejemplos expuestos en este libro, hemos preparado un conjunto de archivos descargables, los cuales incluyen archivos generales, shaders, texturas, código HLSL y otros, que se emplearán en cada capítulo.

Puedes acceder a este contenido visitando nuestro sitio web:

- Link de descarga: <https://jettelly.com>
- O escaneando el código QR a continuación:



Si tienes algún problema o pregunta sobre los recursos, no dude en comunicarse con nuestro equipo de soporte a través del correo electrónico contact@jettelly.com.

Errata.

Al momento de escribir este libro, hemos tomado todas las precauciones necesarias para asegurar la fidelidad de su contenido. Aun así, debemos recordar que somos seres humanos y es muy probable que algunos puntos no estén bien explicados o que se hayan cometido errores ortográficos o gramaticales.

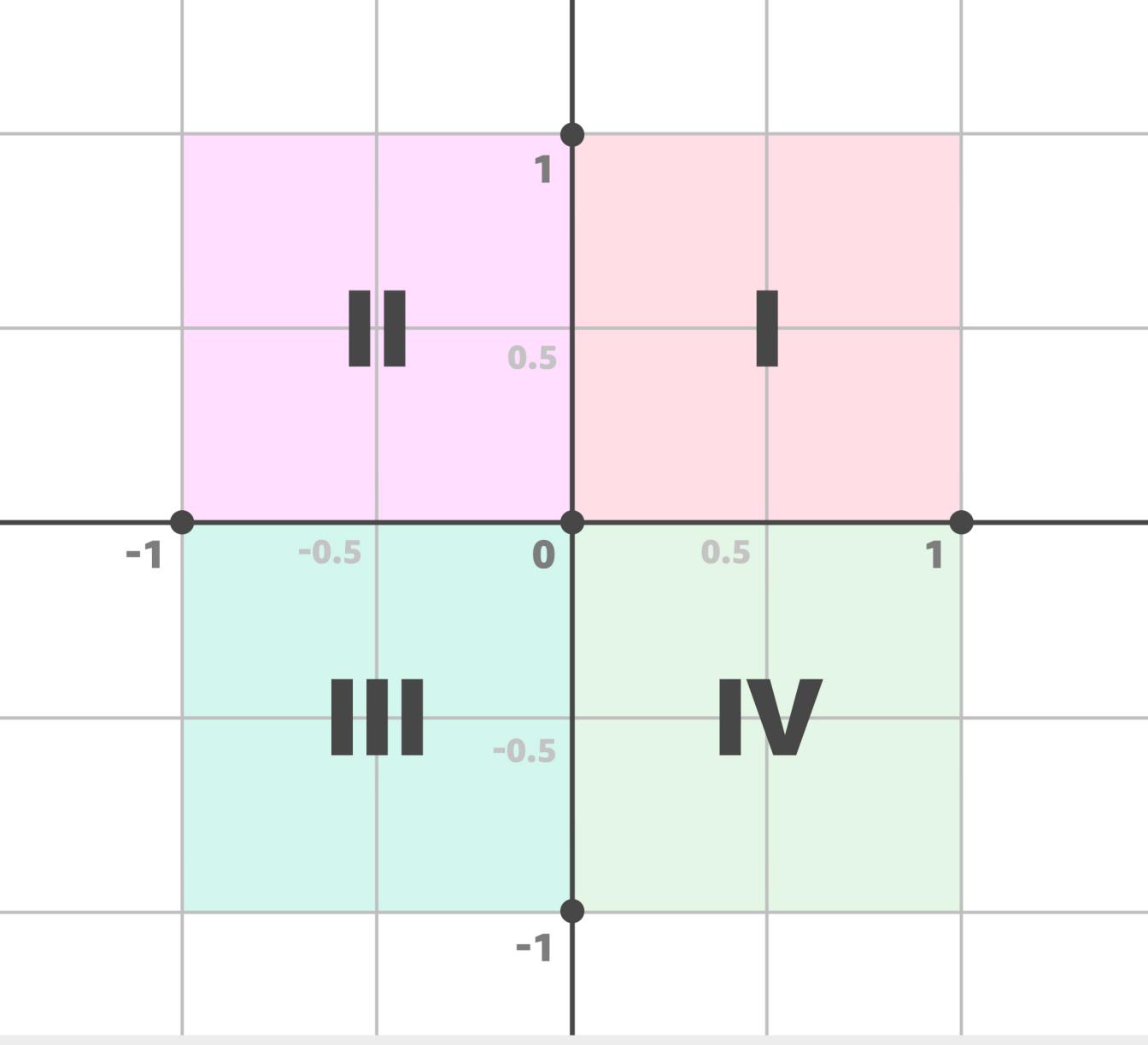
Si encuentras un error conceptual, de código u otro tipo, te agradeceríamos que envias un mensaje al correo contact@jettelly.com indicando en el asunto "VE2 Errata". De esta manera, estarás ayudando a otros lectores a reducir su nivel de frustración y contribuirás a mejorar cada versión del libro en futuras actualizaciones.

Asimismo, si consideras que sería útil agregar alguna sección de interés, puedes enviarnos un correo de todas formas y nosotros incluiremos esa información en próximas ediciones.

Piratería.

Antes de copiar, reproducir o facilitar este material sin nuestro consentimiento, recuerda que Jettelly Inc. es un estudio independiente y autofinanciado, por lo que cualquier práctica ilegal podría afectar nuestra integridad como equipo desarrollador.

Este libro se encuentra patentado bajo derechos de autor y tomaremos la protección de nuestras licencias de manera seria. Si encuentras este libro en una plataforma distinta de jettelly.com o detectas una copia ilegal, te solicitamos que te comuniques con nosotros al correo contact@jettelly.com (adjuntando el enlace si fuese necesario). De esta manera, podremos buscar una solución adecuada. Muchas gracias de antemano por tu colaboración. Todos los derechos reservados.



Capítulo 1
Funciones polinomiales.

En este capítulo te adentrarás en el fascinante mundo de las funciones polinomiales, que constituyen la base de muchas técnicas en la generación de gráficos y formas procedurales. Comenzarás analizando la función lineal, la forma más elemental de un polinomio, para que puedas comprender sus propiedades básicas y aplicarlas en la visualización mediante Shader Graph. Aprenderás no solo a dibujar la línea resultante, sino también a resaltar su punto de origen, un elemento crucial para entender su comportamiento y la transformación de coordenadas en el espacio gráfico.

A medida que avances, te introducirás en la función cuadrática, explorando cómo su curvatura y propiedades geométricas te permiten modelar comportamientos más complejos y naturales en tus figuras. Realizarás ejemplos concretos que te mostrarán la diferencia entre la simplicidad de la función lineal y la versatilidad de la cuadrática, para que sepas en qué contexto utilizar cada una y lograr formas específicas.

Con este enfoque teórico-práctico, tu objetivo es adquirir una comprensión sólida y aplicable que te permita experimentar y crear figuras procedurales dinámicas.

1.1 Función lineal.

Si bien es cierto que las funciones lineales tienen una amplia aplicación en campos como la estadística, la economía y las ciencias sociales, su utilidad se expande aún más, adentrándose incluso en el fascinante universo de los videojuegos. En el ámbito de la computación gráfica, estas desempeñan un papel crucial al trazar líneas rectas, convirtiéndose en una herramienta esencial para la creación de diversos efectos visuales, incluidas las transiciones entre escenas. ¿Pero cómo ocurre esto? Iniciaremos este proceso analizando brevemente el cuerpo de la función lineal.

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

(1.1.a)

Si prestamos atención a función anterior, notamos que la variable m representa la pendiente, la cual indica la inclinación de una recta, mientras que b señala el punto de intersección en el eje y ; es decir, el valor que toma $f(x)$ cuando $x = 0$. Sin embargo, para entender plenamente cómo estos parámetros influyen en la gráfica, es fundamental comprender qué son las coordenadas cartesianas.

Las coordenadas cartesianas conforman un sistema de coordenadas ortogonales en forma rectangular que abarca el espacio euclíadiano en dos dimensiones, x e y . Su definición puede extenderse a una tercera dimensión denominada z , convirtiéndose así en una herramienta esencial para la construcción de modelos tridimensionales. Al utilizar los ejes de las coordenadas cartesianas, podemos subdividir el espacio bidimensional en un conjunto de subregiones conocidas como cuadrantes, que resultan convenientes para definir funciones con precisión al trabajar con otros sistemas de coordenadas.



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

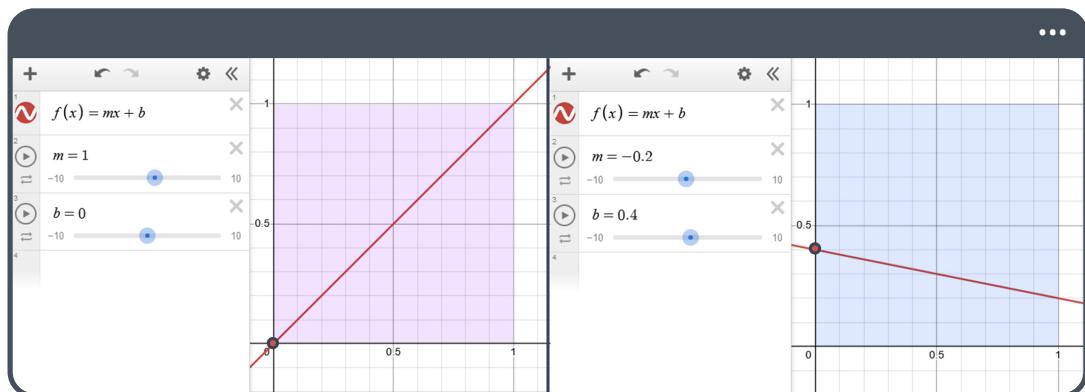
En esta primera etapa, nos centraremos en visualizaciones bidimensionales utilizando la calculadora gráfica de Desmos, que puedes encontrar en el enlace de la referencia 1.1.b. Como se observa en la herramienta, se han empleado colores para diferenciar claramente cada cuadrante, lo que resalta que, según su ubicación, las coordenadas x e y pueden tener valores positivos o negativos.

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

Un aspecto interesante a considerar cuando trabajamos en gráficos por computadora es que, por defecto, las coordenadas UV abarcan únicamente el primer cuadrante, comenzando en [0.0, 0.0] y terminando en [1.0, 1.0]; por lo tanto, ambas coordenadas son positivas.

Dado que aplicaremos estos conocimientos posteriormente en lenguaje HLSL, será esencial tener una comprensión sólida de estos conceptos. De lo contrario, resultará difícil entender cómo trazar una línea en una posición específica en nuestro shader.

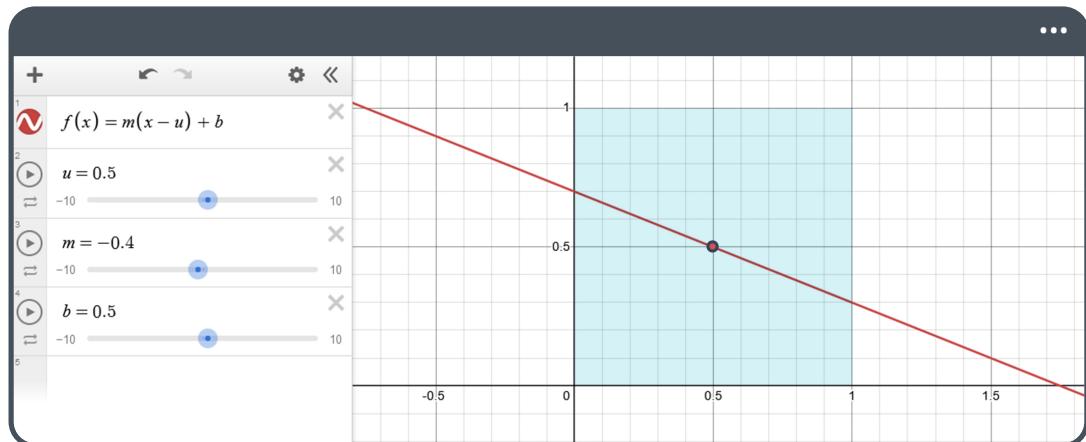
La función que vimos anteriormente en la figura 1.1.a se visualiza en el plano cartesiano de la siguiente manera.



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

Como se evidencia en el enlace de la referencia anterior, la pendiente m representa la cantidad de unidades por las cuales y aumenta o disminuye cuando x experimenta un cambio, mientras que la variable b indica la distancia desde el origen hasta la intersección de la recta con el eje y . Recordemos que el “origen” se refiere al punto de partida en el plano cartesiano, es decir, la posición [0.0, 0.0]

Ahora bien, ¿cómo podemos centrar la función alrededor de la posición [0.5, 0.5] dentro del plano? Para lograrlo, deberíamos ampliar nuestra función lineal e introducir una nueva variable en la ecuación, la cual se reste de x . Veámoslo a continuación.

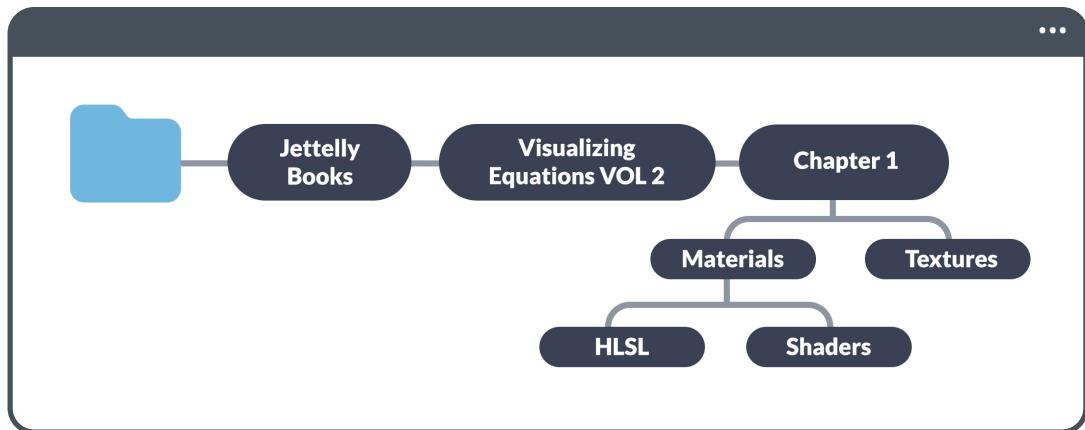


(1.1.d <https://www.desmos.com/calculator/amp8nacnqb>)

Como se observa en el enlace de la referencia 1.1.d, se ha introducido una nueva variable denominada como u , en la función. Esta variable está asociada a la coordenada U de los UV y tiene un valor de 0.5. Al restarla de x , u genera un desplazamiento horizontal que centra el punto de referencia en el plano cartesiano. Este procedimiento es crucial, ya que al trabajar con coordenadas UV, a menudo necesitaremos ajustar el punto de origen para trazar líneas desde ubicaciones distintas.

1.2 Visualizando la función lineal en HLSL.

Antes de sumergirnos en la representación gráfica de la función lineal, es fundamental establecer una estructura organizativa para el proyecto en desarrollo. Comenzaremos con el siguiente esquema, el cual iremos ampliando a medida que avancemos en la lectura de este libro.



(1.2.a)

Considerando la naturaleza de Shader Graph, para visualizar una función personalizada en el editor, necesitaremos crear al menos tres objetos en nuestro proyecto, los cuales corresponden a:

- Shader.
- Material.
- Script con extensión **.hsls**.

Cada uno de los elementos mencionados desempeña un rol fundamental en el proceso de visualización. Por una parte, el shader se encarga de ejecutar las operaciones matemáticas especificadas en el script HLSL, realizando los cálculos necesarios para determinar cómo se representarán gráficamente las imágenes, mientras que el material actúa como un medio a través del cual se visualizan estos cálculos, aplicando los efectos definidos por el shader en los objetos dentro de la escena, lo que resulta en la representación visual deseada.

Es importante destacar que, en Unity, los shaders se estructuran utilizando dos lenguajes de programación principales: ShaderLab y HLSL. El primero sirve como lenguaje declarativo que el software utiliza para definir propiedades, configuraciones y estructuras, estableciendo el marco dentro del cual operan. Actúa como el esqueleto que organiza cómo y cuándo se aplicarán los diferentes pasos de procesamiento gráfico, mientras que

HLSL se enfoca en el aspecto computacional, permitiendo describir precisamente las operaciones matemáticas y algoritmos gráficos que definen el comportamiento de color.

Procederemos a crear los objetos mencionados previamente, iniciando con un shader de tipo **Unlit Shader Graph** al cual llamaremos **Linear Function**. Podemos encontrar este objeto en la siguiente ruta:

- Assets > Create > Shader Graph > URP.

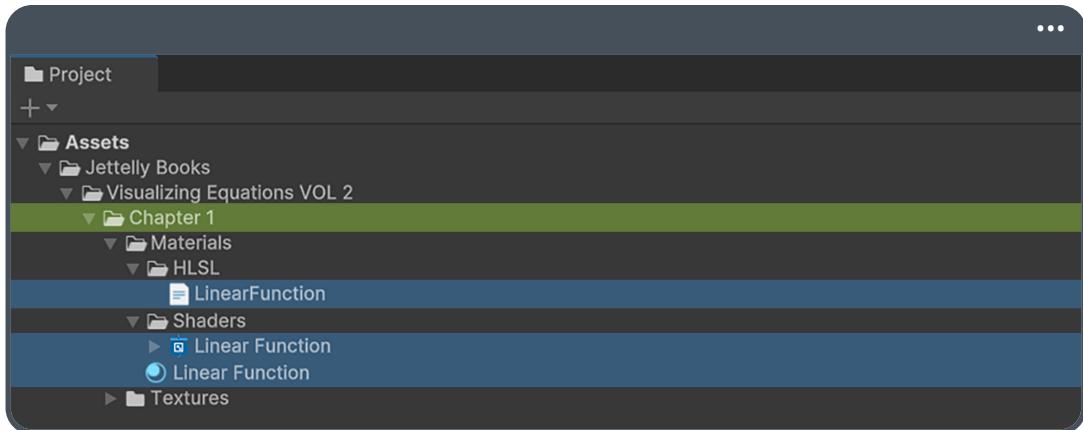
En general, optaremos por un shader de tipo **Unlit Shader Graph** a lo largo de este libro. La característica **Unlit** no responde a la iluminación del entorno, lo cual resulta ideal para la exemplificación de la función, evitando así cálculos adicionales. No obstante, en secciones posteriores, exploraremos un enfoque diferente (UI Default) que nos permitirá aplicar cálculos sobre imágenes de interfaz de usuario, expandiendo así el alcance de nuestras técnicas de visualización.

Continuaremos creando un material que, por razones prácticas, también denominaremos como **Linear Function**. Este podemos encontrarlo en:

- Assets > Create > Material.

Finalmente, generaremos un script HLSL. Sin embargo, Unity no incluye este tipo de scripts de forma predeterminada, por lo que tendremos que crearlo directamente desde el editor de código que estemos utilizando o manualmente desde una carpeta en nuestro sistema operativo. Esto implica que naveguemos hasta la carpeta HLSL en nuestro proyecto (ya sea en Windows, Mac o Linux), crearemos un nuevo documento de texto y cambiaremos su extensión de **.txt** a **.hlsl**. Una vez creado, nos aseguraremos de emplear la misma denominación utilizada en los dos objetos generados previamente, omitiendo los espacios en su definición, para que quede como **LinearFunction**. Este esquema lo aplicaremos en todos los archivos de este tipo a lo largo del libro.

Si todas las configuraciones se han realizado correctamente, nuestro proyecto deberá presentar los siguientes archivos:



(1.2.b)

Cabe destacar que, será necesario realizar algunos pasos complementarios para asegurar la correcta visualización de las funciones en la ventana Scene. Para ello:

- Debemos asignar el shader al material.
- Luego, incluir un objeto 3D de tipo Quad en la ventana Hierarchy.
- Por último, asignar el material al Quad en la ventana Scene.

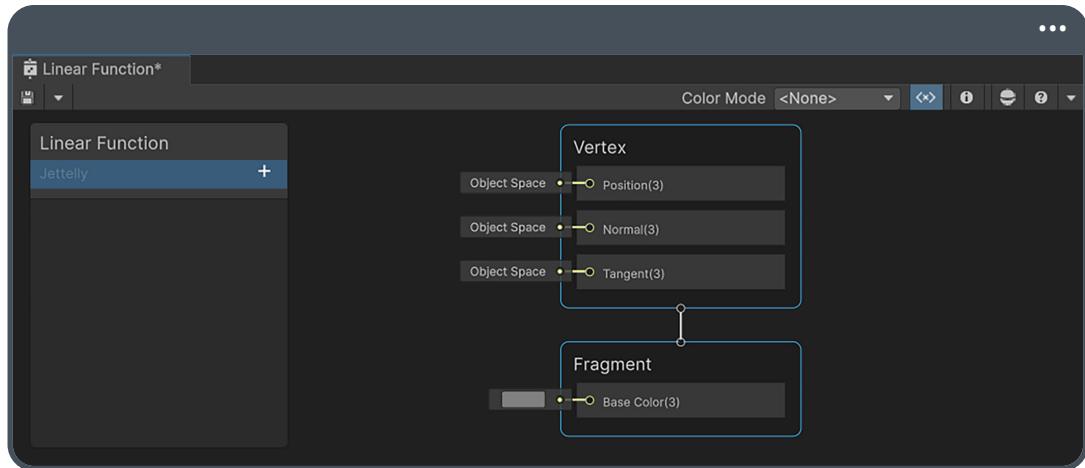
No obstante, ¿cómo podemos encontrar el shader creado previamente? Para esto, será necesario conocer su ubicación por defecto. Cada vez que creamos un shader de tipo **.shadergraph**, se guarda en la ruta:

- Shader > Shader Graph.

Puedes verificar esta información abriendo el shader **Linear Function** y seleccionando el **Blackboard**. La ruta debería aparecer debajo del nombre del archivo. Sin embargo, por motivos prácticos, configuraremos la ruta con el nombre **Jettelly**, quedando de la siguiente manera:

- Shader > Jettelly > Linear Function.

Cabe mencionar que, por consistencia, realizaremos el cambio de ruta sobre todos los shaders que desarrollemos a lo largo del libro, por aquella mencionada previamente.



(1.2.c)

Dado que la función lineal carece de un nodo independiente en Shader Graph, será necesario utilizar el nodo **Custom Function** para definirla. Dentro del **Master Stack** (área donde conectamos los nodos), pulsaremos la tecla ESPACIO y buscaremos el nodo por su nombre. Para comprender plenamente el funcionamiento de este nodo, resulta esencial activar el **Graph Inspector**, de esta manera podremos definir tanto sus entradas como salidas.

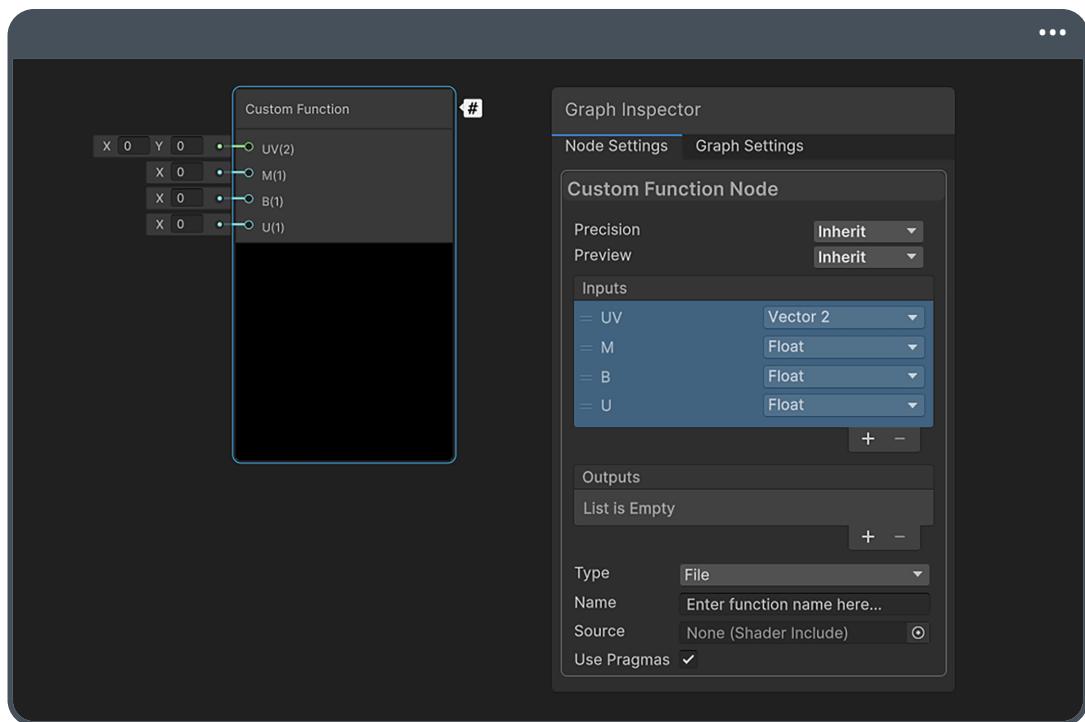
En cuanto a las entradas, debemos considerar las variables definidas en la función que se presentan en la referencia 1.1.a, es decir:

- La variable m .
- La variable b .
- Las coordenadas xy , que es lo mismo a UV.

Además, incorporaremos la variable u , utilizada para expandir la función, como se ilustra en la referencia 1.1.d para visualizar el desplazamiento horizontal que centra la función en el punto deseado.

Cuando nos referimos al plano cartesiano en gráficos por computadora, en realidad hablamos de diferentes espacios que podemos encontrar en nuestro entorno de desarrollo. Por ejemplo, las coordenadas espaciales globales; las que definen el mundo

euclídeo en la ventana Scene, pueden funcionar como nuestro plano cartesiano. Sin embargo, también existen las coordenadas de pantalla o las coordenadas UV, que representan espacios de manera similar. En este caso, optaremos por trabajar con coordenadas UV, ya que a partir de ellas podremos comprender de manera clara el concepto de uv-space.

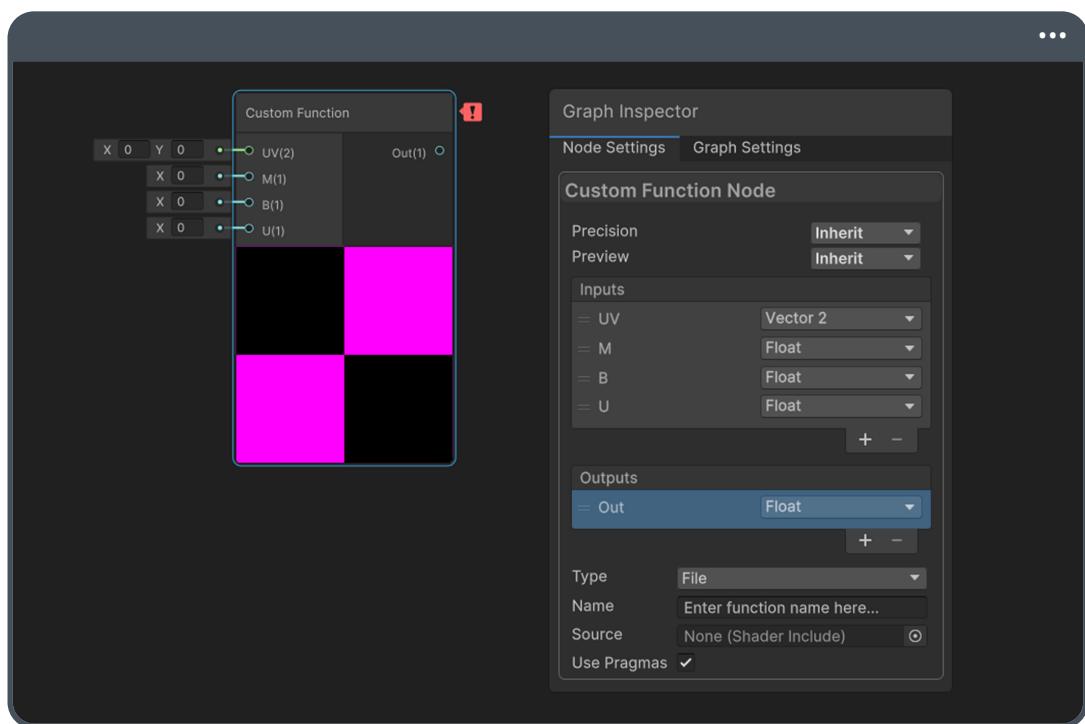


(1.2.d com.unity.shadergraph@17.0/manual/Custom-Function-Node.html)

Si prestamos atención a la figura 1.2.d, notamos que las variables de la función lineal se han integrado como entradas en el nodo **Custom Function**. Es importante señalar que el tipo de dato de las coordenadas UV corresponde a un vector bidimensional, ya que en él almacenaremos la ubicación de los píxeles en sus ejes *x* e *y*. En cuanto a la salida, esta varía según el ejercicio que estemos realizando. En este caso, retornaremos un valor flotante determinado por una condicional, que será igual a 1.0 o 0.0, dependiendo de un umbral que podremos modificar en tiempo real a través de las variables *m*, *b* y *u* desde el Inspector.

Es importante destacar que, en este contexto, los valores numéricos se traducen a una gama de colores en la pantalla del ordenador. Por lo tanto, 1.0 y 0.0 se pueden interpretar lumínicamente como blanco y negro respectivamente para cada píxel. En otras palabras, si el **Output** es igual a 1.0, los píxeles dentro del área de la función se mostrarán de color blanco, mientras que si es igual a 0.0, se mostrarán en negro. De manera similar, un valor de 0.5 devolverá un color gris.

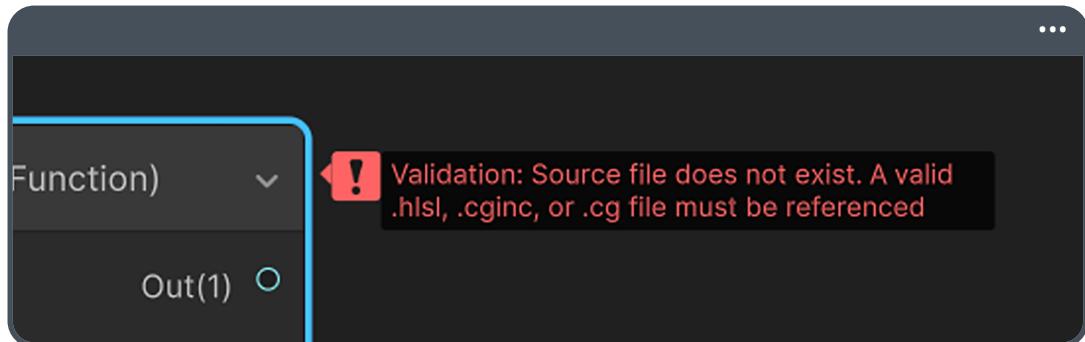
Por razones prácticas, utilizaremos **Out** como nombre de salida del nodo, tal como se presenta en el siguiente ejemplo:



(1.2.e Error en el nodo Custom Function)

Un aspecto crucial en Shader Graph es que, cada vez que efectuamos cambios en la configuración del nodo, como añadir o eliminar inputs, modificar la disposición de variables en las listas o actualizar funciones en su archivo HLSL, pueden surgir errores comunes, que van desde problemas de validación hasta la sobreescritura de variables. Al examinar la referencia 1.2.e, identificamos un error que, en este caso, se origina por

un conflicto de validación: no hemos asignado un script con extensión **.hlsl** como fuente (Source) y, además, no hemos definido un nombre para el nodo.



(1.2.f Error de validación)

Para resolver este error, llevaremos a cabo las siguientes acciones:

- Asignaremos el archivo **LinearFunction** en la propiedad **Source** del nodo.
- Utilizaremos **linear_function** como nombre para el nodo.

Según las convenciones de denominación de Microsoft para HLSL, los tipos de datos, métodos y funciones internas deben presentarse en CamelCase. Sin embargo, optaremos por palabras separadas por guiones bajos, como por ejemplo **linear_function_half()** para mejorar la legibilidad de las funciones y métodos que implementemos a lo largo del libro.

Es importante tener en cuenta que, dado que aún no hemos definido el método mencionado previamente en el archivo **LinearFunction.hlsl**, es posible que el error persista después de realizar los cambios en el nodo **Custom Function**. A diferencia de C#, Unity no incluye código por defecto para este caso, por lo que será necesario abrir nuestro script y declarar el método desde cero:

```
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 }
```

Al analizar el fragmento de código anterior, se aprecia que tanto el método definido como sus argumentos son de tipo **half**, el cual corresponde a un tipo de dato de media precisión, es decir, 16 bits. Lo interesante de este tipo de dato es que, según la documentación en Unity:

“

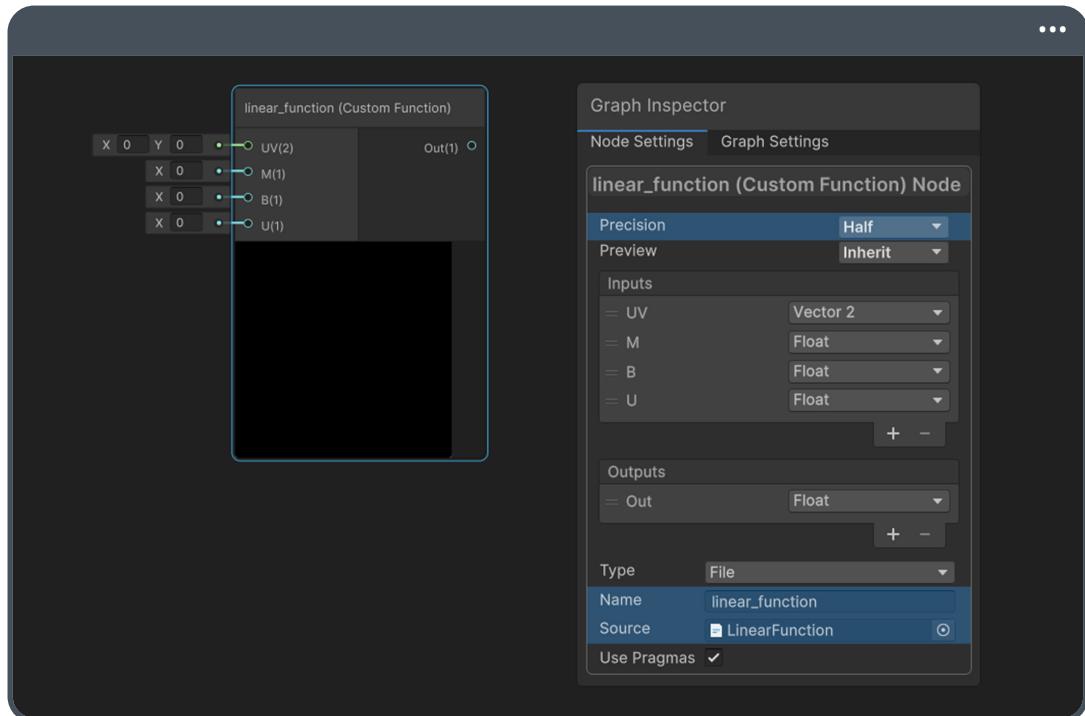
Aquellas plataformas que soporten **half**, serán 16 bits. En otras plataformas, este tipo de datos se convierte en **float** (32 bits).

”

Aunque la función que vamos a emplear no generará una carga significativa en la GPU, resulta innecesario el uso de este tipo de dato. No obstante, por razones educativas, optaremos por utilizarlo y posteriormente configuraremos nuestro nodo en media precisión.

La incorporación de **half** en nuestro programa resulta interesante, ya que potencialmente puede mejorar el rendimiento al reducir el ancho de banda de memoria, ser más eficiente al ahorrar espacio e incluso puede ser más rápido en operaciones aritméticas. Aunque presenta algunas desventajas, como la pérdida de precisión, especialmente en sistemas de coordenadas, no profundizaremos en esos detalles a lo largo del libro. No obstante, es importante tener en cuenta que la elección de utilizar este tipo de dato debe evaluarse en función del contexto y las necesidades específicas de cada proyecto, considerando siempre el balance entre la eficiencia y precisión.

Si todo ha transcurrido correctamente, nuestro nodo debería producir un color negro como resultado, tal como se ilustra en la siguiente imagen:



(1.2.g)

El color negro se genera a partir del valor de salida definido temporalmente como 0.0 en la línea número 3 de nuestro código. Dado que este valor corresponde a un tipo de dato escalar, solo podemos visualizar colores en escala de grises. Si deseamos obtener un color RGB como salida, será necesario cambiar el tipo de dato de **half** a **half3**.

En este ejercicio, nos limitaremos a trabajar únicamente con escala de grises. Por lo tanto, procederemos a implementar la función lineal, comenzando por la declaración e inicialización de sus coordenadas.

```

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 }
```

Si observamos la línea de código número 3, notaremos que hemos declarado e inicializado la variable **fx** con la coordenada **uv.y**. ¿Por qué razón? Porque ambas hacen referencia a lo mismo, a la coordenada *y* del plano cartesiano. Posteriormente, en la línea de código siguiente, hemos declarado e inicializado la variable **x** con la coordenada **uv.x**. Teniendo ambas coordenadas, solo falta definir la función lineal, de manera similar a como se detalla en la ecuación 1.1.a de la sección anterior.

```

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 }
```

Si nos enfocamos en la línea de código número 6, observamos que se ha aplicado el esquema de la función lineal a la variable **f**, la cual resta de la función de la abscisa **fx** en la línea siguiente. Como se ha mencionado previamente, en esta ocasión utilizaremos la función extendida, la cual incorpora la variable **u** en la operación.

```

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 }
```

Al examinar nuevamente la línea de código número 6, se aprecia que, **u** resta a **x**, lo cual permite un desplazamiento horizontal del punto central en la ecuación. Ahora solo falta definir el valor de salida de la función. Para ello, emplearemos una condicional: devolveremos 1.0 o 0.0 si **fx > 0**. De esta manera, lograremos proyectar tanto el área positiva como negativa de la función lineal. Esto será de bastante ayuda más adelante, ya que, para este ejercicio, proyectaremos dos texturas, cada una en las distintas áreas de la función para luego interpolarlas según el valor de **f**.

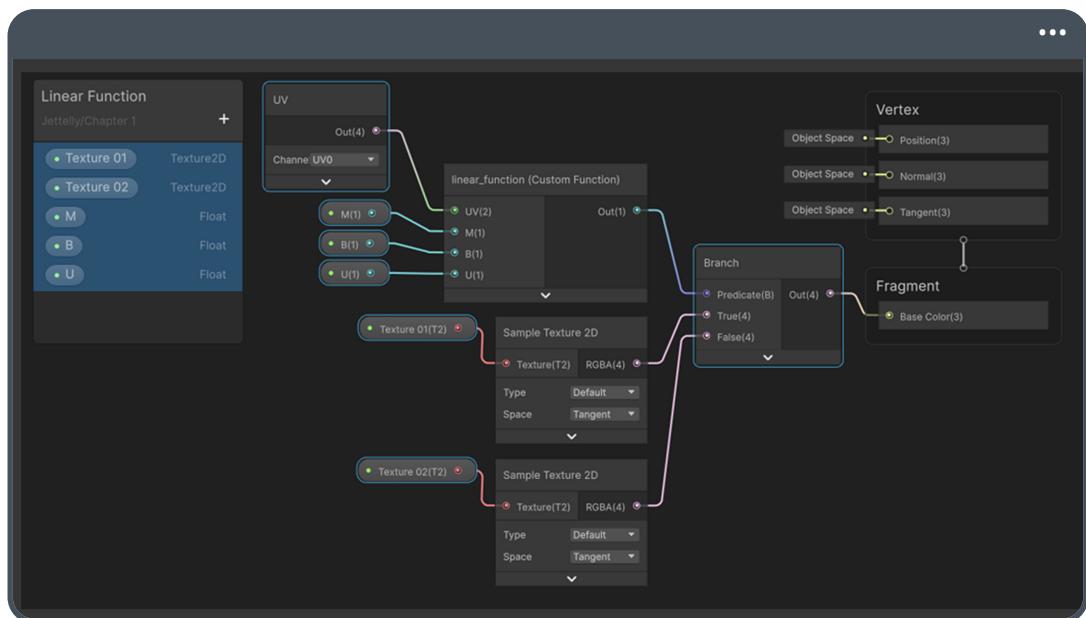
```

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 }
```

A continuación, nos adentraremos en el siguiente paso, donde nos enfocaremos en la manipulación de texturas y su interpolación utilizando el nodo **Branch** en Shader Graph.

Este nodo tiene la capacidad única de devolver un valor basándose en dos condiciones: verdadero o falso. La configuración de estas condiciones se realiza a través de la propiedad **Predicate**, que es de tipo **bool**.

En este ejemplo concreto, aprovecharemos esta función para proyectar dos texturas distintas, asignando cada una a las áreas positiva y negativa de la función lineal que definimos previamente en la línea de código número 9. No obstante, antes de iniciar este proceso, declararemos algunas propiedades en el **Blackboard** que emplearemos en conjunto con nuestro nodo **linear_function**. También incorporaremos el nodo **UV**, que sirve como coordenadas cartesianas, específicamente; las del primer cuadrante.



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

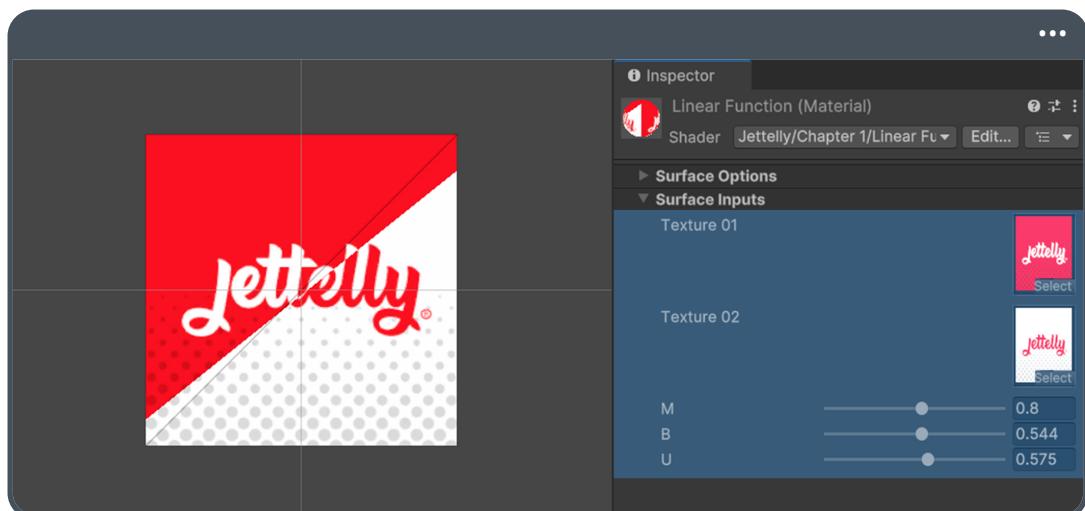
Como podemos apreciar en la referencia 1.2.h, se han definido cinco propiedades en el **Blackboard**: dos texturas; **Texture 01** y **Texture 02**, y tres valores flotantes: **m**, **b** y **u**. Cada una de estas propiedades tiene un rango específico, estudiado previamente, con el fin de evitar cálculos fuera del área del Quad que utilizaremos como referencia para proyectar la función lineal posteriormente. Estos límites aseguran que los datos

se mantengan dentro de intervalos óptimos, lo que nos permite ajustar la función sin generar errores de renderizado ni cálculos imprecisos.

- La propiedad **m** abarca un rango entre -10 y 10.
- Mientras que las propiedades **b** y **u** tienen un rango entre 0.0 y 1.0.

Dichas propiedades se han vinculado al nodo **linear_function** siguiendo sus correspondientes equivalencias. Del mismo modo, el nodo **UV** ha sido conectado a este último. Posteriormente, la salida del nodo **linear_function** se ha enlazado a la entrada Predicate del nodo **Branch**, ¿Por qué hemos realizado esta conexión? Principalmente, porque necesitamos que el nodo devuelva tanto **true** como **false** en una sola ejecución.

Finalmente, el resultado de la operación global se ha vinculado al color de entrada **Base Color** en la etapa de procesamiento de fragmento. Al guardar nuestro shader y regresar al material **Linear Function** en nuestro proyecto, podremos ajustar las variables de la función lineal desde la ventana Inspector. Es importante destacar que será necesario asignar dos texturas a este último para visualizar el funcionamiento de la operación mencionada previamente.



(1.2.i Visualización de la función lineal)

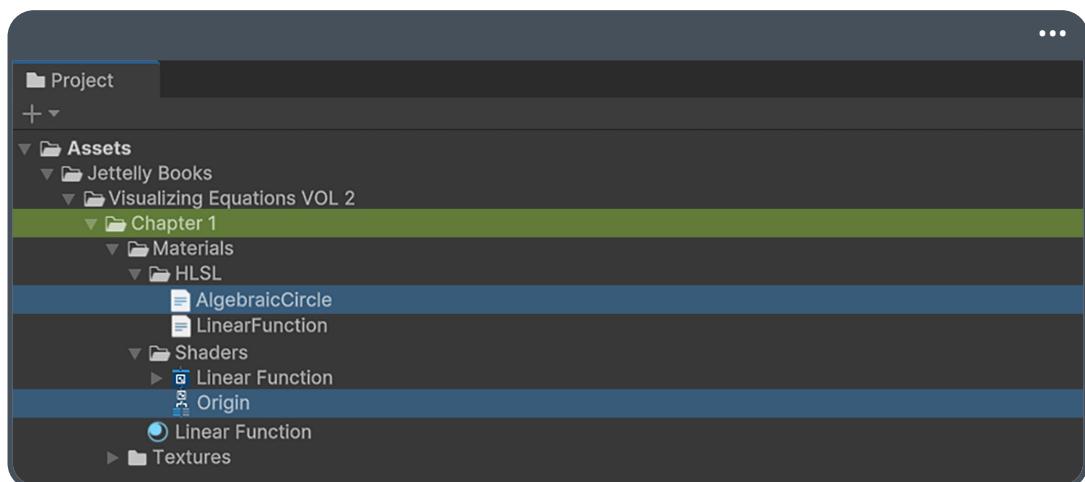
Al ajustar dinámicamente los valores de las propiedades, resulta evidente la carencia de una representación gráfica que facilite la identificación del punto de origen de la función lineal. Por lo tanto, en la próxima sección, dirigiremos parte de nuestros esfuerzos hacia la creación de un nuevo nodo con este propósito.

1.3 Visualizando el punto de origen.

Continuando con el proceso realizado hasta el momento, nos dirigiremos a nuestro proyecto y crearemos un objeto de tipo Sub Graph al cual daremos el nombre de **Origin**. Este elemento se encuentra en la siguiente ruta:

- Assets > Create > Shader Graph > Sub Graph.

Simultáneamente, incorporaremos un nuevo script de tipo HLSL, al cual nombraremos **AlgebraicCircle**.



(1.3.a)

¿Cuál es el propósito detrás de la creación de estos objetos? Sub Graph nos brinda la capacidad de construir un nodo que podremos emplear dentro de un shader, de manera análoga a cómo se utilizaría una función en un archivo **.cgincl**. Mientras que, como ya sabemos, el script de tipo HLSL nos permitirá generar una función personalizada.

Dado que la visualización del punto de origen podría ser útil en múltiples funciones, resulta esencial llevar a cabo esta acción. Con el nuevo Sub Graph creado, procederemos a personalizar el nodo para representar de manera gráfica el punto de origen de nuestra función lineal.

La ecuación que utilizaremos para su representación corresponde a:

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

(1.3.b)

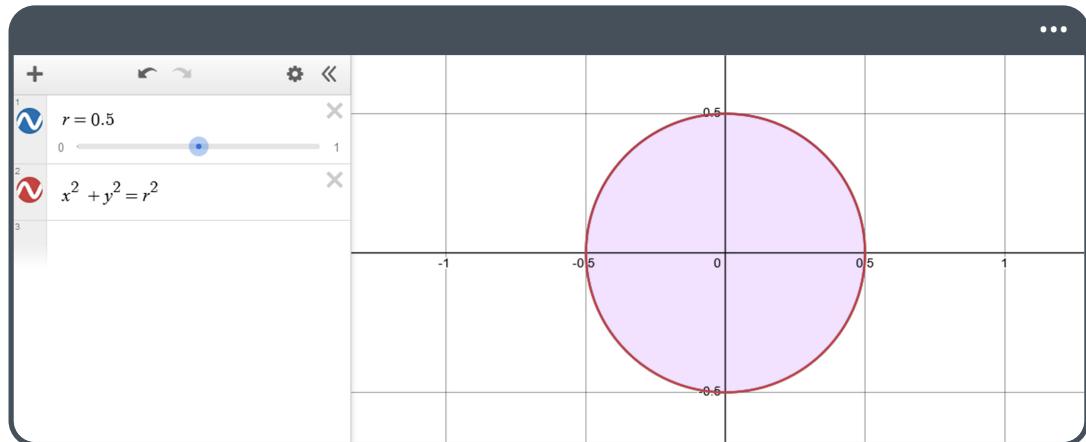
Como se puede observar en la ecuación 1.3.b, esta describe una región geométrica que toma la forma de un círculo. Surge entonces la interrogante: ¿por qué se genera esta figura específica? La respuesta se encuentra en el teorema de Pitágoras, el cual establece la relación fundamental entre los lados de un triángulo rectángulo. Al imponer que r sea constante, las soluciones de esta ecuación corresponden a todos los puntos que se encuentran a una distancia r del origen, formando así la región geométrica.

Es importante señalar que, debido a su naturaleza, esta ecuación no se ajusta a la definición precisa de una “función”. ¿Por qué razón? Una función implica una relación especial entre dos conjuntos, donde a cada elemento del primer conjunto le corresponde un único elemento en el segundo conjunto. En otras palabras, para cada valor de la variable independiente x , debe existir un único valor correspondiente en la variable dependiente y .

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

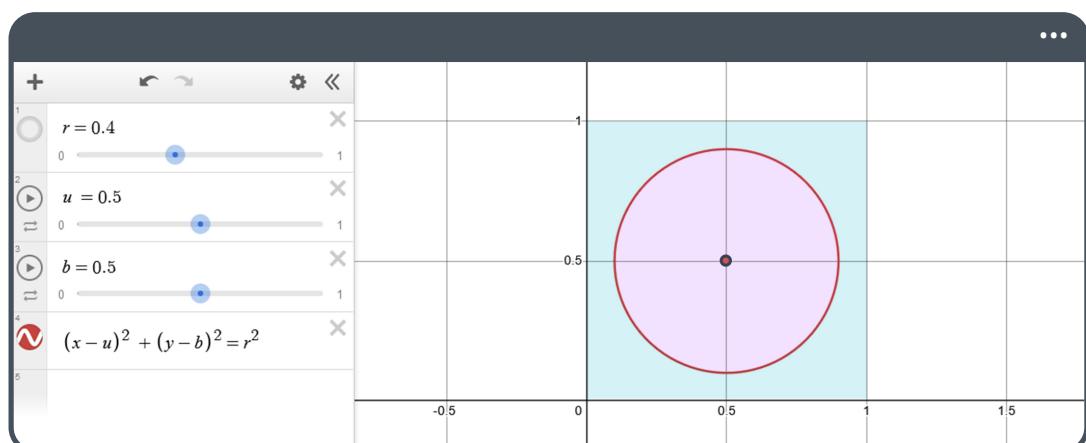
(1.3.c)

Como podemos observar en la función 1.3.c, en este caso, tal relación no se cumple, ya que, al invertir la ecuación 1.3.b, la variable y adopta tanto un valor positivo como negativo.



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

Si representamos la ecuación 1.3.b en un plano cartesiano, notaremos que su forma permanece estática en el punto de origen, lo cual resulta inconveniente para nuestro propósito, ya que necesitamos que se desplace conforme al origen de la función lineal. Por ende, será necesario expandir nuevamente nuestra ecuación, como se ilustra en la siguiente imagen:

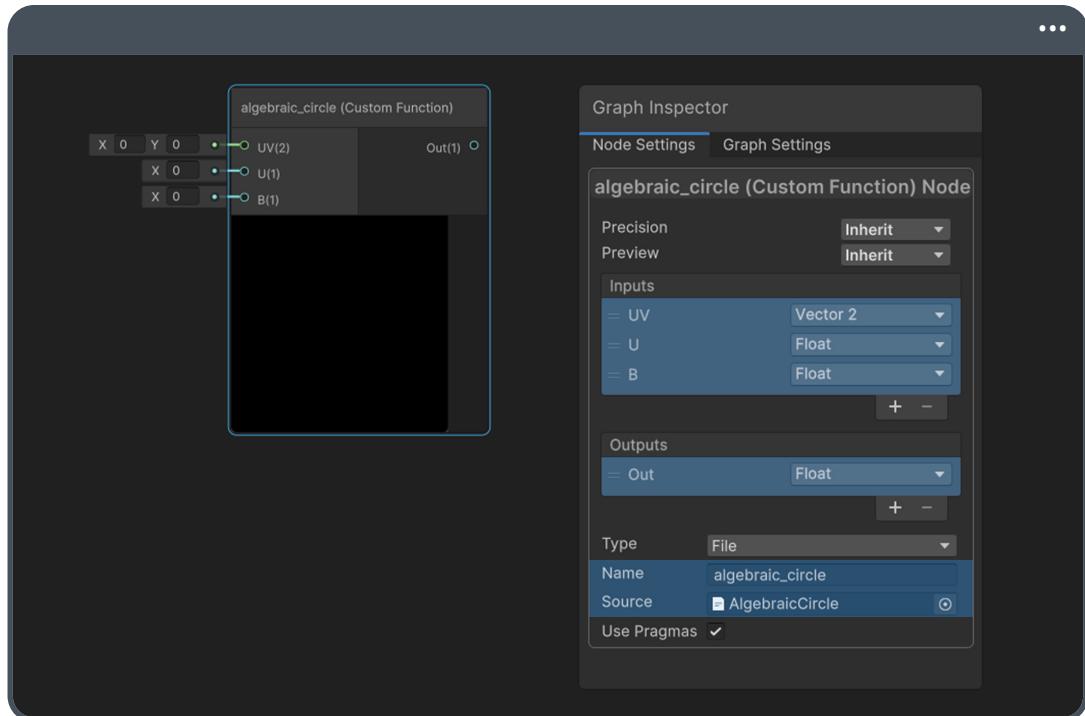


(1.3.e <https://www.desmos.com/calculator/twzngfv0qj>)

Como podemos observar, se han incorporado dos nuevas variables en la ecuación, u y b , que generan un desplazamiento tanto horizontal como vertical del círculo en el plano cartesiano. Con esta información, avanzaremos en el desarrollo de nuestro Sub Graph, utilizando un nodo **Custom Function** para crear la forma geométrica.

A diferencia de un gráfico de tipo **.shadergraph**, un Sub Graph no cuenta con un nodo **Master**, en su lugar, presenta únicamente una salida con un valor de entrada que podemos ajustar tanto en tipo de dato como en precisión. Esto nos lleva a deducir que el resultado de la operación realizada dentro del gráfico deberá conectarse posteriormente a este nodo.

Puesto que emplearemos el círculo exclusivamente para visualizar el punto de origen, no será necesario incorporar funcionalidades de color RGB a nuestra función por ahora. Por lo tanto, nos aseguraremos de incluir las variables u y b , así como las coordenadas x e y de la ecuación, como inputs en el nodo **Custom Function**. Para la salida **Out**, asignaremos un valor flotante. Además, seleccionaremos el archivo **AlgebraicCircle** como **Source** y utilizaremos **algebraic_circle** como **Name**. Es importante recordar que las coordenadas xy corresponden a las coordenadas UV en el contexto de los shaders.



(1.3.f)

Si examinamos detenidamente la referencia 1.3.f, podemos notar que no se ha incluido el radio r como entrada en el nodo. Esto se debe a que posteriormente definiremos un valor constante para el mismo. Antes de declarar las propiedades de nuestro círculo en el **Blackboard**, nos aseguraremos de integrar el método **algebraic_circle_half()** en el script HLSL. De esta manera, garantizamos una compilación sin errores.

```

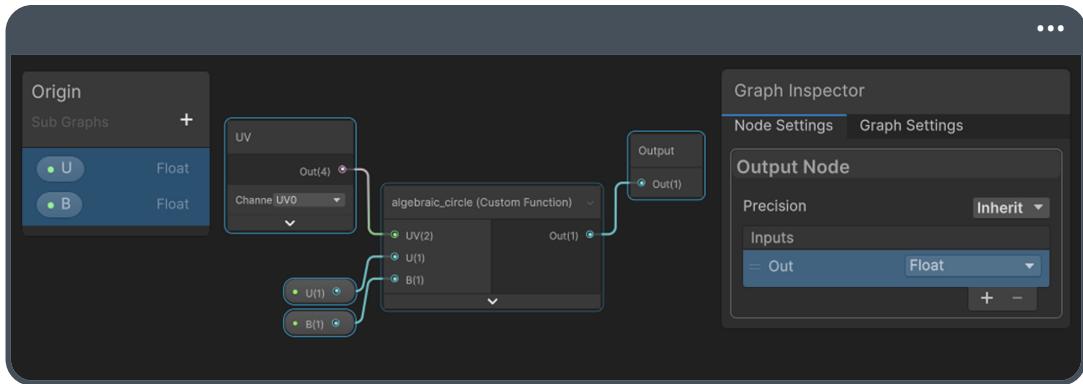
1 #define RADIUS 0.01
2
3 void algebraic_circle_half(in half2 uv, in half u, in half b, out half Out)
4 {
5     const half r = RADIUS * RADIUS;
6
7     half x = uv.x - u;
8     half y = uv.y - b;
9
10    half circle = x * x + y * y; // pow(x, 2) + pow(y, 2)
11
12    Out = (circle <= r) ? 0.0 : 1.0;
13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

```

Como observamos en la figura 1.3.b, la formación del círculo resulta cuando la suma de los cuadrados de las coordenadas *xy* es igual al cuadrado del radio *r*. Al centrar nuestra atención en la línea de código número 5 del ejemplo anterior, notaremos que la constante *r* se define como el doble de **RADIUS**. Este mismo factor se refleja en la línea de código número 10, donde la variable **circle** incluye dos veces las coordenadas **x** e **y**.

Un detalle interesante es la ausencia de la función **pow()** en la operación. ¿Por qué no la estamos utilizando en esta ocasión? Principalmente, debido a que involucra aritmética de punto flotante, lo cual podría conducir a problemas de rendimiento en algunas GPU. Es importante señalar que esta dificultad está en gran medida resuelta en GPU modernas; sin embargo, siempre es prudente minimizar la cantidad de cálculos cuando sea posible.

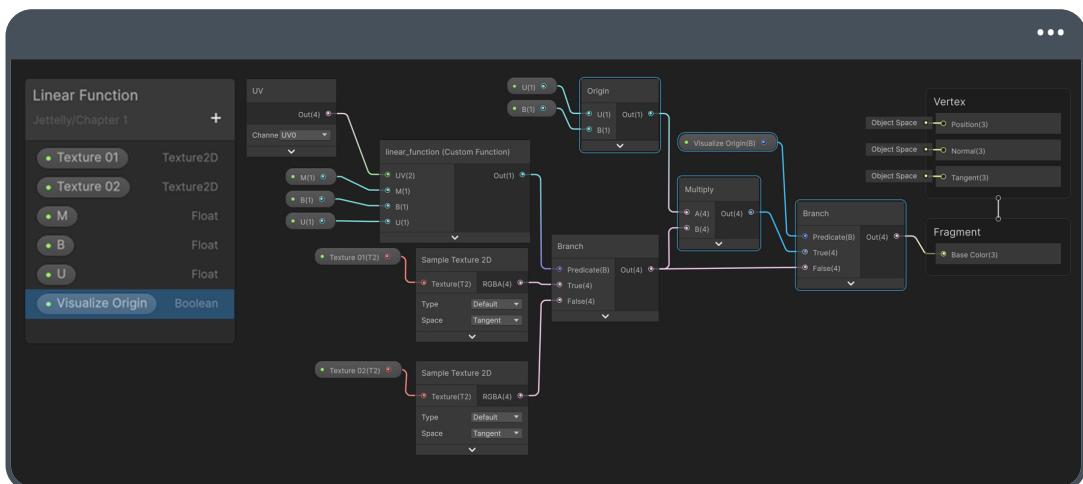
Finalmente, retornamos un color blanco o negro según el radio del círculo, como se evidencia en la línea de código número 12. Si todo ha sido realizado correctamente, nuestro nodo debería compilar sin inconvenientes. Solo resta agregar las propiedades **u** y **b** en el **Blackboard**, y conectarlas al nodo para asegurar su correcto funcionamiento.



(1.3.g)

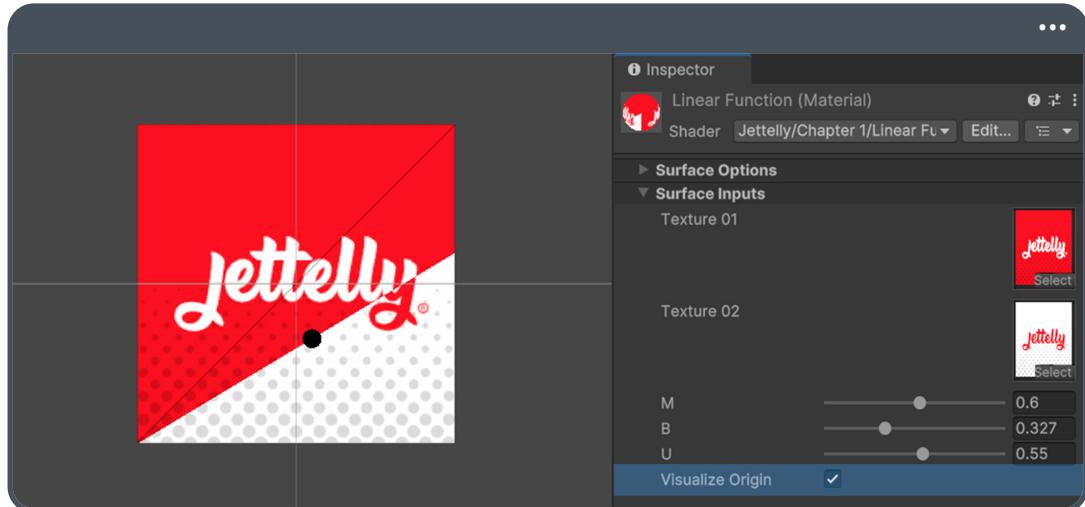
A continuación, incorporaremos una nueva funcionalidad a nuestro gráfico **Linear Function**: la visualización del punto de origen. Para lograr esto, llevaremos a cabo los siguientes pasos:

- 1 Integramos el nodo **Origin** y conectamos las propiedades **u** y **b** como entradas.
- 2 Añadimos una nueva propiedad de tipo **Boolean** al **Blackboard** y la nombramos **Visualize Origin**.
- 3 Multiplicamos el resultado de la operación lineal por **Origin**.
- 4 Incluimos un nodo **Branch** adicional para activar o desactivar la visualización del punto de origen.



(1.3.h)

Si todo ha transcurrido perfectamente, la visualización del punto de origen (el cual podremos apreciar a través de un punto de color negro por defecto) podrá ser habilitado directamente desde la propiedad **Visualize Origin** en la ventana del Inspector.



(1.3.i)

1.4 Trazando una recta entre dos puntos.

Cuando nos enfrentamos al desafío de generar imágenes de forma procedural, una de las ecuaciones que encontramos con mayor frecuencia es aquella que nos permite trazar una recta entre dos puntos. La denominada ecuación de la recta es una expresión matemática que proporciona una descripción geométrica precisa de una línea recta en un plano.

La forma más común de esta ecuación es conocida como la forma punto-pendiente, la cual se expresa como:

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

(1.4.a)

Donde xy son las coordenadas de un punto de la recta, m es la pendiente y $[a_x, a_y]$ hacen referencia a los componentes del primer punto por el cual queremos que pase la recta. Simplificando la operación presentada en la función 1.4.a, podemos expresar la ecuación de la siguiente manera,

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

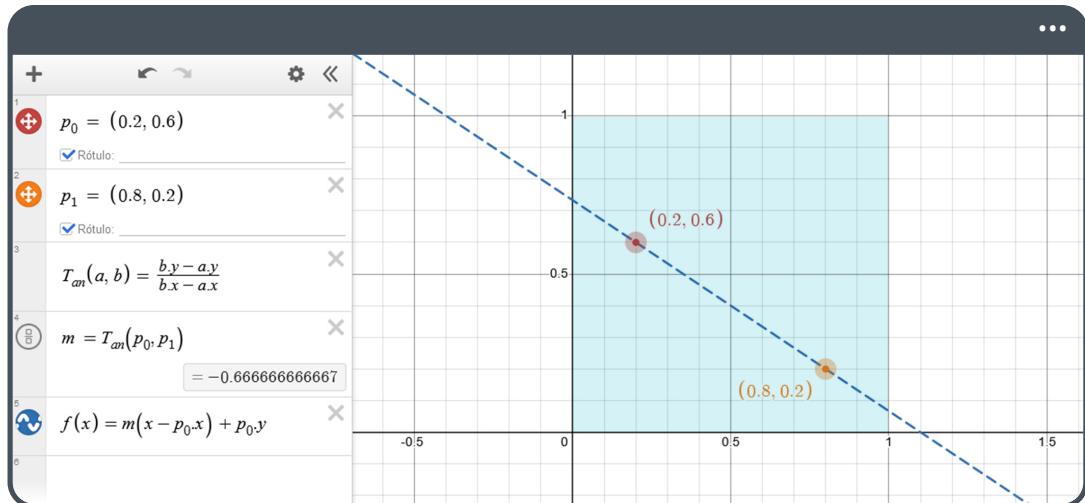
(1.4.b)

Al observar la función 1.4.b, notamos que tal ecuación corresponde con la definición de una función lineal en la forma $y = mx + b$, la cual pasa por el punto $[a_x, a_y]$. Si quisiéramos trazar una recta entre un punto $[a_x, a_y]$ y otro punto $[b_x, b_y]$, la definición de la pendiente está dada por:

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

(1.4.c)

Para trazar una recta entre dos puntos, el primer paso que deberíamos llevar a cabo es definir la posición de ambos puntos. Posteriormente, determinamos el valor de la pendiente y, finalmente, trazamos la recta, tal como se ilustra en la siguiente imagen.



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

Si dirigimos nuestra atención a la imagen 1.4.d, observaremos la definición de dos puntos p_0 y p_1 , en el primer cuadrante del plano cartesiano. Luego, se ha introducido una función denominada $T_{an}(a, b)$, que hace referencia a la ecuación presentada de la fórmula 1.4.c. Posteriormente, se ha declarado la variable m con el propósito de mejorar la legibilidad de la pendiente, para finalmente aplicar la ecuación de la recta descrita en la función 1.4.b.

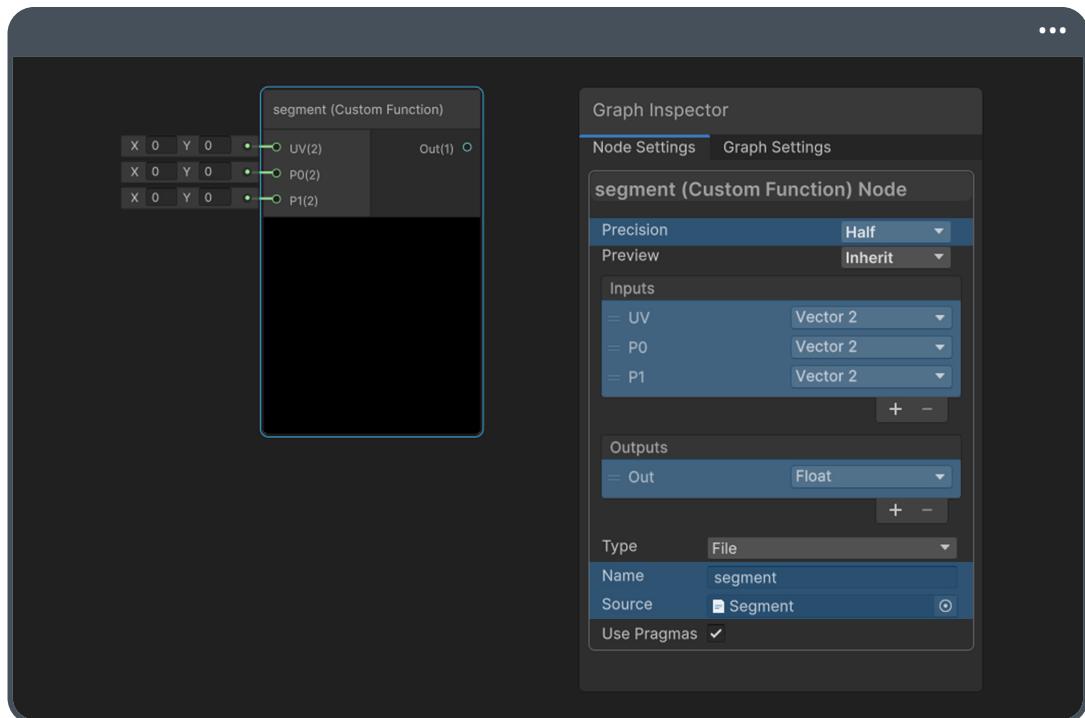
Es importante destacar que la recta trazada en el ejercicio anterior representa un segmento “infinito” en ambas direcciones. La comprensión de este comportamiento es crucial, ya que, en la mayoría de los casos, al dibujar formas mediante ecuaciones, será necesario limitar la longitud de la recta. Para ilustrar este concepto, crearemos un nuevo shader de tipo **Unlit Shader Graph** en nuestro proyecto. Con propósitos educativos, denominaremos a este objeto como **Segment**. Además, generaremos tanto un nuevo archivo HLSL como un material, ambos con el mismo nombre empleado previamente.



(1.4.e Estructura del proyecto)

Utilizaremos el archivo **Segment.hlsl** para construir una función personalizada. En este caso, utilizaremos los puntos definidos en la imagen 1.4.d como entradas en nuestro nodo, lo que nos permitirá modificarlos dinámicamente desde la ventana del Inspector. A continuación, procederemos con los siguientes pasos:

- Dentro de nuestro shader crearemos un nodo **Custom Function**.
- Como entradas, añadiremos tres vectores bidimensionales: Al primero lo llamaremos **UV**; al segundo, **P0**; y al tercero, **P1**, manteniendo el mismo orden.
- Como salida, retornaremos un valor de coma flotante.
- Finalmente, asignaremos **Segment.hlsl** como **Source**, y como **Name** escribiremos **segment**, el nombre del método que emplearemos a continuación.



(1.4.f)

En nuestro script, la primera línea de código que incluiremos corresponde a la declaración del método como tal, la cual considera las variables fundamentales **UV**, **P0**, **P1** como entradas, y una salida **Out**, como se muestra en el siguiente ejemplo:

```

1 void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
2 {
3     Out = 0.0;
4 }
```

Sin embargo, considerando la pendiente m de la fórmula 1.4.c, la cual utilizaremos para trazar una recta entre dos puntos, también será necesario implementar su función dentro de nuestro código. Para ello, implementaremos un nuevo método al cual denominaremos **tangent()**, como se muestra a continuación:

```

1 half tangent(half2 a, half2 b)
2 {
3     return (b.y - a.y) / (b.x - a.x);
4 }
5
6 void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
7 {
8     Out = 0.0;
9 }
```

Este método (línea 1) adquiere los puntos **a** y **b** como argumentos de la misma manera en que se presenta en la figura 1.4.c. Cabe destacar que, aunque estos puntos podrían denominarse **p0** y **p1** respectivamente, se ha optado por una nomenclatura diferente para representar de forma precisa la función descrita anteriormente.

Posteriormente, inicializaremos tanto las coordenadas **uv.x** e **uv.y** como la variable de la pendiente **m** siguiendo el mismo esquema de la ecuación de la recta:

```

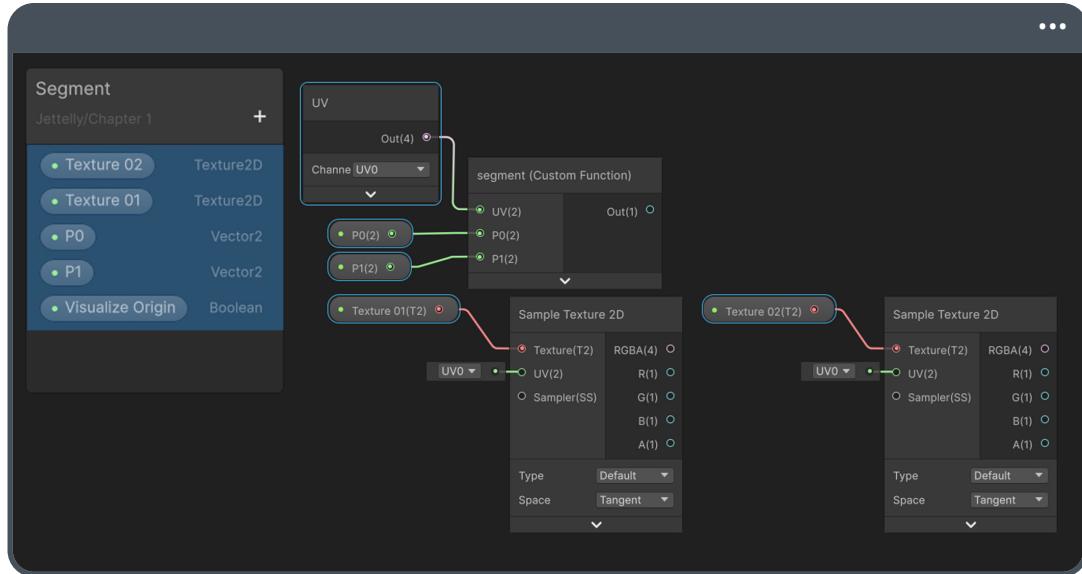
6 void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
7 {
8     half fx = uv.y;
9     half x = uv.x;
10    half m = tangent(p0, p1);
11
12    Out = 0.0;
13 }
```

Para concluir, incorporaremos la función que representa el segmento entre dos puntos (función 1.4.b) y devolvemos blanco o negro según el valor de la variable **fx**. Es importante mencionar que la elección de devolver 1.0 o 0.0 se realiza únicamente como una simplificación del ejercicio con el fin de visualizar la función. En capítulos posteriores de este libro, emplearemos un filtro para generar áreas de color y así construir formas procedurales más complejas.

```
6 void segment_half(in half2 uv, in half2 p0, in half2 p1, out half Out)
7 {
8     half fx = uv.y;
9     half x = uv.x;
10    half m = tangent(p0, p1);
11
12    half f = m * (x - p0.x) + p0.y;
13    fx -= f;
14
15    Out = (fx > 0.0) ? 1.0 : 0.0;
16 }
```

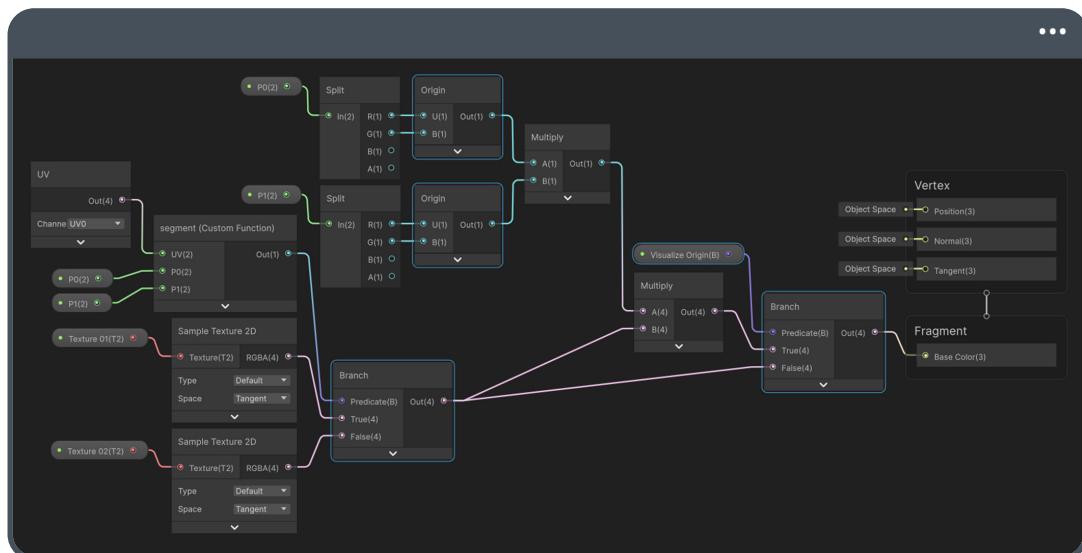
Hasta este momento, nuestro nodo debería compilar sin problemas. No obstante, para garantizar su funcionamiento integral, será fundamental declarar los vectores **p0** y **p1** en el **Blackboard**.

Es relevante señalar que, con el objetivo de optimizar el contenido, repetiremos parte del proceso abordado en secciones anteriores. En este sentido, añadiremos dos texturas de tipo **Texture2D**, nombradas como **Texture 01** y **Texture 02**, y también incluiremos una propiedad booleana para la visualización del punto de origen. Podemos anticipar que estas texturas serán empleadas para diferenciar las distintas áreas generadas por el segmento comprendido entre los puntos previamente definidos.



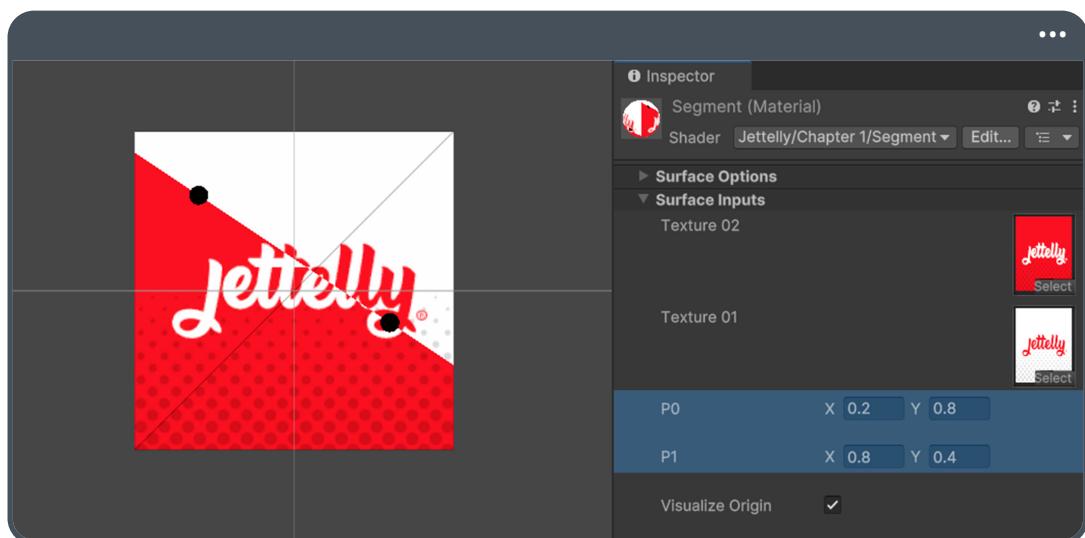
(1.4.g)

Mediante el nodo **Branch**, conectamos ambas texturas para lograr una proyección de la composición en el Quad presente en la ventana Scene. Además, será necesario incluir dos nodos **Origin** para visualizar con precisión la posición de los puntos, tal como se ilustra en el siguiente ejemplo:



(1.4.h)

De la imagen 1.4.h, podemos observar que se ha conectado cada punto **P0** y **P1**, con un nodo **Origin**, respectivamente. No obstante, como parte del proceso intermedio, se ha utilizado un nodo **Split**, el cual separa los canales de cada vector. Al configurar el valor por defecto de los puntos a un rango entre [0.0 : 0.5] para **P0**, y [0.5 : 0.0] para **P1**, podremos apreciar con facilidad el comportamiento del segmento generado entre ambos puntos. Cabe destacar que el valor de ambos vectores puede ser modificado dinámicamente desde la ventana Inspector.



(1.4.i Visualización de los puntos en el Quad)

1.5 Función cuadrática.

Existen diversas situaciones en las cuales podríamos potencialmente emplear una función cuadrática; por ejemplo, al lanzar una moneda al aire y modelar su movimiento. Sin embargo, al igual que con una función lineal, la cuadrática nos brinda la capacidad de generar patrones visualmente interesantes en el ámbito de la representación de gráficos por computadora. De hecho, podemos utilizar la parábola para trazar curvas que representen áreas de iluminación o incluso para definir bordes redondeados en una figura específica. Ahora bien, ¿cómo podemos lograr esto?

Prestemos atención a la siguiente ecuación para comprender el concepto:

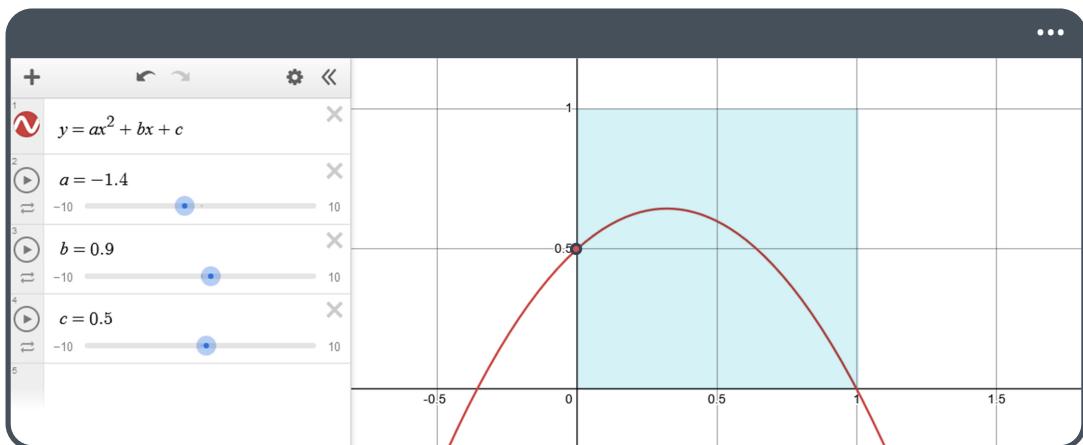
$$f(x) = ax^2 + bx + c$$

(1.5.a)

Los coeficientes constantes representados por las variables a , b y c intervienen en la función, mientras que x denota la variable independiente. En términos de salida, $f(x)$, equivalente a y , refleja el resultado de la función; una curva.

Al visualizar esta función en sistema de coordenadas, observamos que el coeficiente cuadrático a controla la concavidad y apertura de la parábola. Si es positivo, la parábola se abre hacia arriba (convexa); en cambio, si es negativo, se abre hacia abajo (cónvexo).

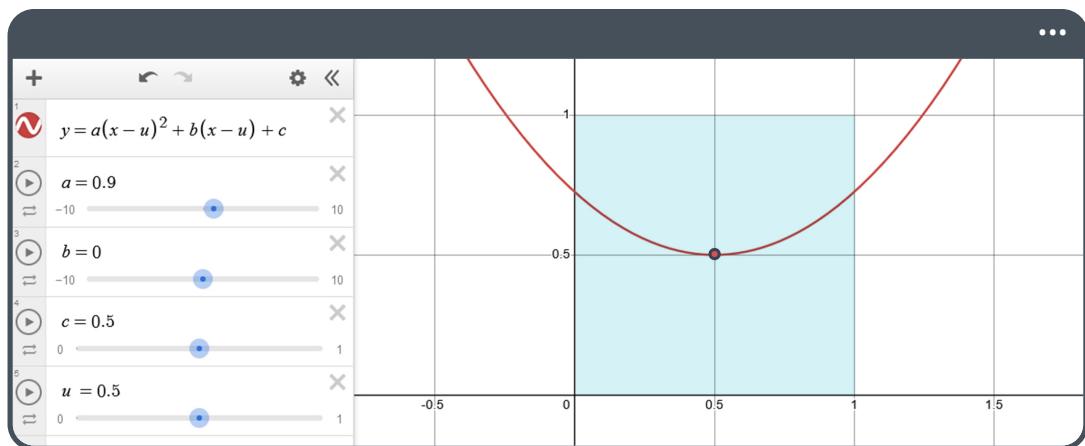
La variable b , o coeficiente lineal, influye en la posición horizontal de la parábola. Un valor positivo desplaza la curva hacia la izquierda, mientras que uno negativo la desplaza hacia la derecha. Finalmente, la variable c que corresponde al término constante, determina la posición vertical de la parábola en el eje y .



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

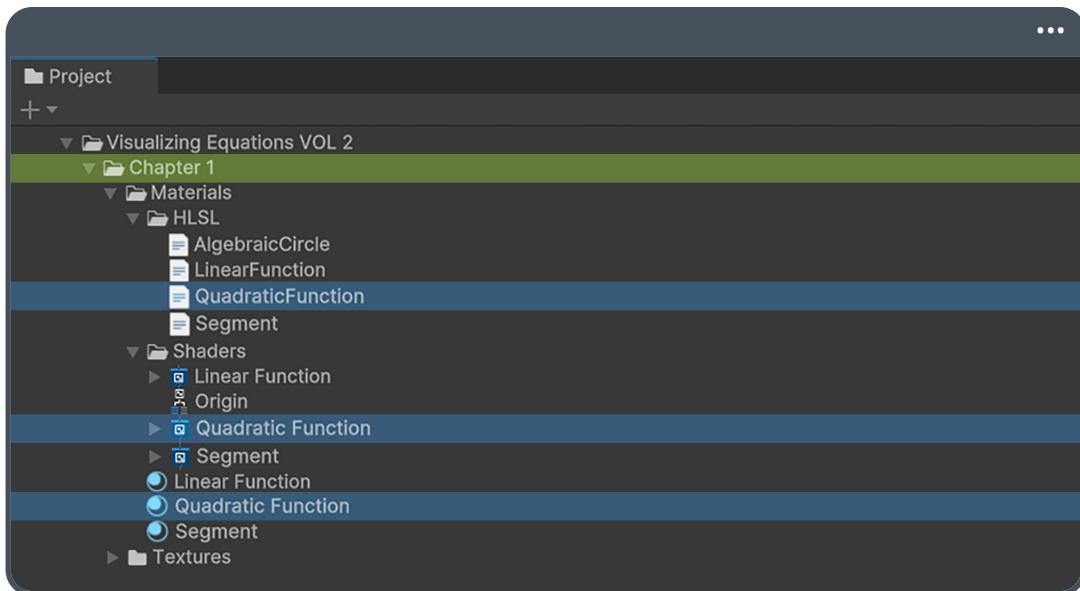
Cuando trabajamos con HLSL y planeamos utilizar la parábola de manera recurrente, al igual que en otras funciones, surge la necesidad de modificar su punto de origen

horizontalmente. Para lograr esto, la solución es bastante sencilla: incorporamos una variable que reste a la variable independiente x . Este ajuste nos permite desplazar la parábola horizontalmente en el plano cartesiano, brindándonos mayor flexibilidad en la manipulación de su posición y adaptándola de manera más precisa a nuestras necesidades.



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

En la imagen 1.5.c, introducimos una nueva variable u a nuestra función cuadrática. Al restar u de la variable independiente x , conseguimos un desplazamiento horizontal del punto de origen, el cual se representa gráficamente como un punto negro en la imagen. Este ajuste nos lleva a la pregunta clave: ¿cómo podemos implementar esta operación HLSL? Para abordar esta cuestión, emprenderemos la creación de un nuevo shader en nuestro proyecto. Adicionalmente, generaremos un archivo con extensión **.hlsl** y un material correspondiente. Esta configuración nos facilitará el desarrollo de un nodo personalizado, permitiéndonos además visualizar directamente la función en el Quad de la ventana Scene. Por simplicidad, denominaremos **Quadratic Function** a los tres elementos recientemente generados.

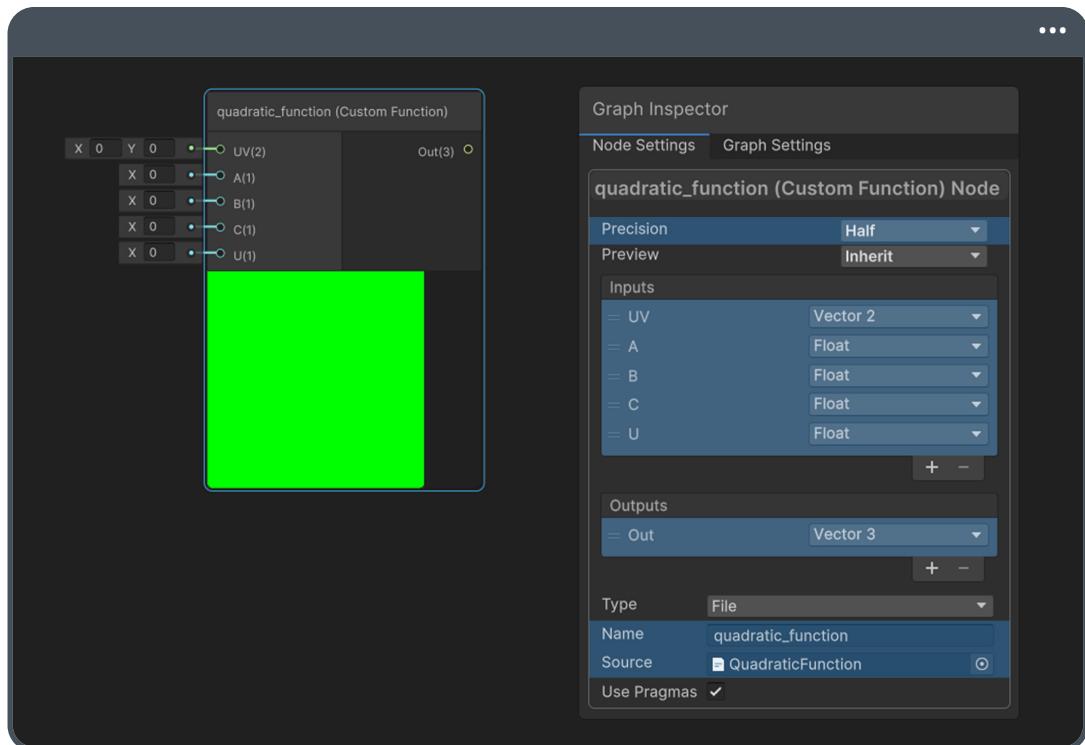


(1.5.d)

A continuación, procederemos a definir las propiedades de la función cuadrática dentro de un nodo **Custom Function**. Considerando la ecuación de la imagen 1.5.a, podemos deducir que las variables de entrada corresponden a los coeficientes a , b y c , los cuales son valores de coma flotante que determinan la forma y posición de la parábola. Sin embargo, para ampliar nuestra función, también será necesario incluir las coordenadas UV y la variable u , permitiendo así la interacción dinámica del punto de origen. Al igual que los coeficientes, u se define como una variable de coma flotante.

En este caso, también debemos definir el valor de salida. No obstante, en esta oportunidad no implementaremos texturas en el shader, en cambio nos centraremos únicamente en el uso de colores para explorar la creación de figuras procedurales. Por este motivo, configuraremos el **output** como un vector de tres dimensiones (RGB).

Para vincular estas propiedades con el código, seleccionaremos y arrastraremos el archivo **QuadraticFunction.hsl** al campo **Source** del nodo. Además, estableceremos **quadratic_function** como nombre de la función, asegurándonos de que coincida con su implementación HLSL.



(1.5.e)

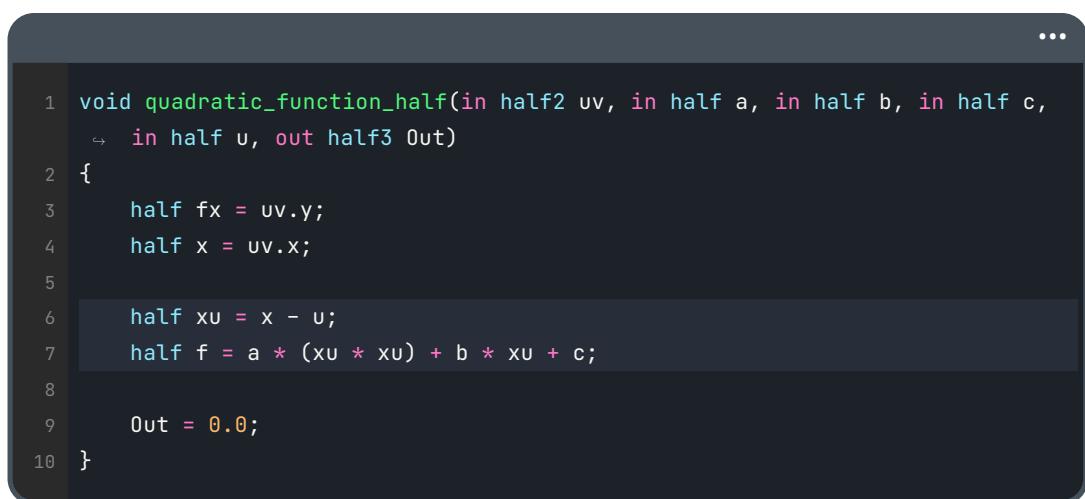
Con estas configuraciones en su lugar, estaremos listos para definir la función cuadrática dentro del archivo HLSL, incorporando la variable **u** como argumento para permitir la interacción dinámica del punto de origen.

```

1 void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
2                               in half u, out half3 Out)
3 {
4     half fx = uv.y;
5     half x = uv.x;
6
7     Out = 0.0;
}

```

Si observamos detenidamente la línea número 3 del código, notaremos que se ha definido la función de la abscisa, **fx**, e inicializado con la coordenada **uv.y**. De manera similar, hemos inicializado y definido la variable **uv.x** con su respectiva coordenada. Por lo tanto, solo falta especificar la función cuadrática, es decir, la relación entre estas variables que dará forma a la parábola en nuestro shader. Teniendo en cuenta la función cuadrática estándar, implementaremos la ecuación considerando sus variables de entrada, es decir: **a**, **b**, **c**, **x** e **y**. Además, incluiremos la variable **u**.

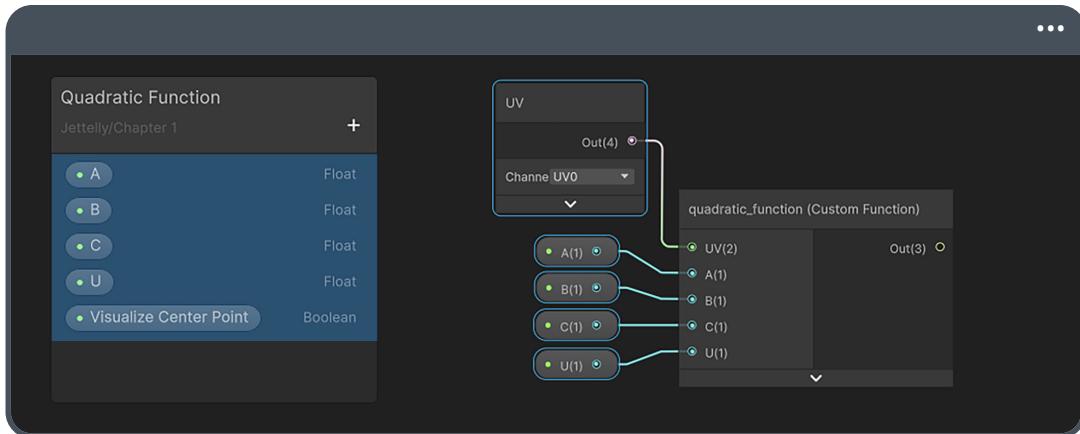


```
1 void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
2     in half u, out half3 Out)
3 {
4     half fx = uv.y;
5     half x = uv.x;
6
7     half xu = x - u;
8     half f = a * (xu * xu) + b * xu + c;
9
10    Out = 0.0;
}
```

Si prestamos atención a la línea de código número 7, observaremos que la variable **f** toma el coeficiente cuadrático, lineal y constante, así como las coordenadas respectivas, y retorna el valor de la parábola. Para concluir el ejercicio, en esta oportunidad, definiremos dos colores aleatorios constantes los cuales harán referencia tanto al área positiva como negativa de la función cuadrática. Cabe destacar que la elección de colores es únicamente a efectos de demostración y simplificación del ejercicio. En prácticas más avanzadas, estos podrían ser valores dinámicos o responder a lógicas más específicas.

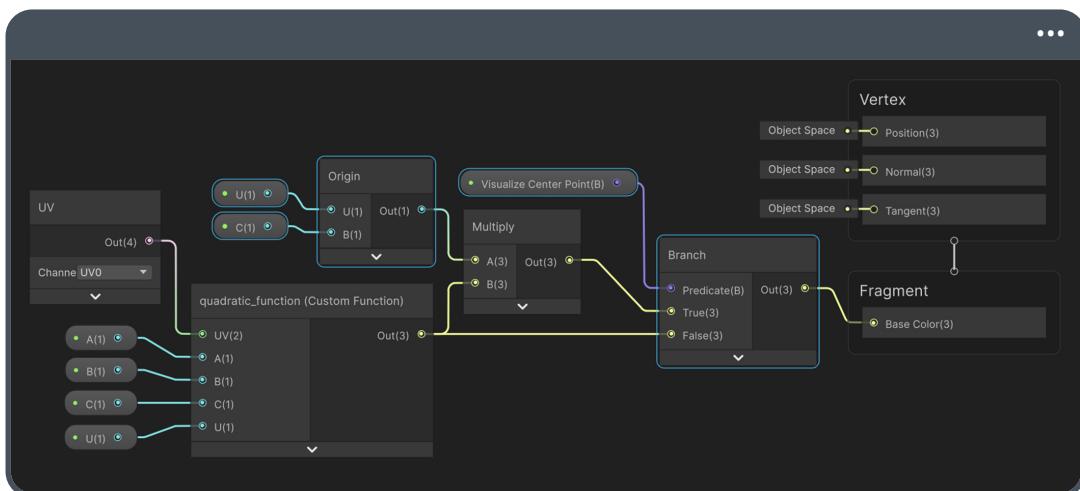
```
1 void quadratic_function_half(in half2 uv, in half a, in half b, in half c,
2     in half u, out half3 Out)
3 {
4     half fx = uv.y;
5     half x = uv.x;
6
7     half xu = x - u;
8     half f = a * (xu * xu) + b * xu + c;
9
10    const half3 red = half3(1, 0, 0);
11    const half3 green = half3(0, 1, 0);
12
13    fx -= f;
14    Out = (fx > 0.0) ? red : green;
}
```

En las líneas número 9 y 10 observamos la declaración e inicialización de dos vectores tridimensionales que definen los colores rojo y verde en nuestro código. Luego, se ha empleado una estructura condicional en la línea número 13 para determinar tanto el área positiva como negativa de la función cuadrática. En este punto, nuestro nodo **Custom Function** debería compilar sin problemas. No obstante, al igual que en ejercicios anteriores, Shader Graph podría solicitar ajustar la resolución del nodo en el Graph Inspector. En caso de que sea necesario, configuraremos la propiedad **Precision** como **Half**. Además, definiremos las propiedades en el **Blackboard** para modificar de manera dinámica la función desde el Inspector.



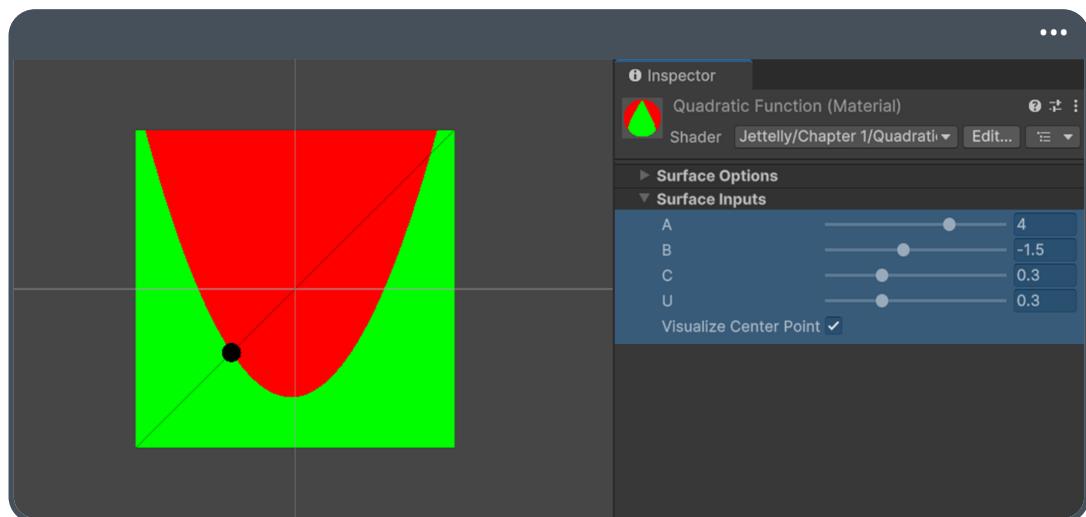
(1.5.f)

Si dirigimos nuestra atención al **Blackboard** de la imagen anterior, notaremos que se ha añadido una propiedad booleana llamada **Visualize Center Point**. Siguiendo la misma metodología de ejercicios anteriores, utilizaremos esta propiedad para activar o desactivar la visualización del punto de origen en nuestra función. En consecuencia, emplearemos nuevamente un nodo **Branch** en el shader, donde su valor positivo será igual a la multiplicación de la función cuadrática por la salida del nodo **Origin**.



(1.5.g)

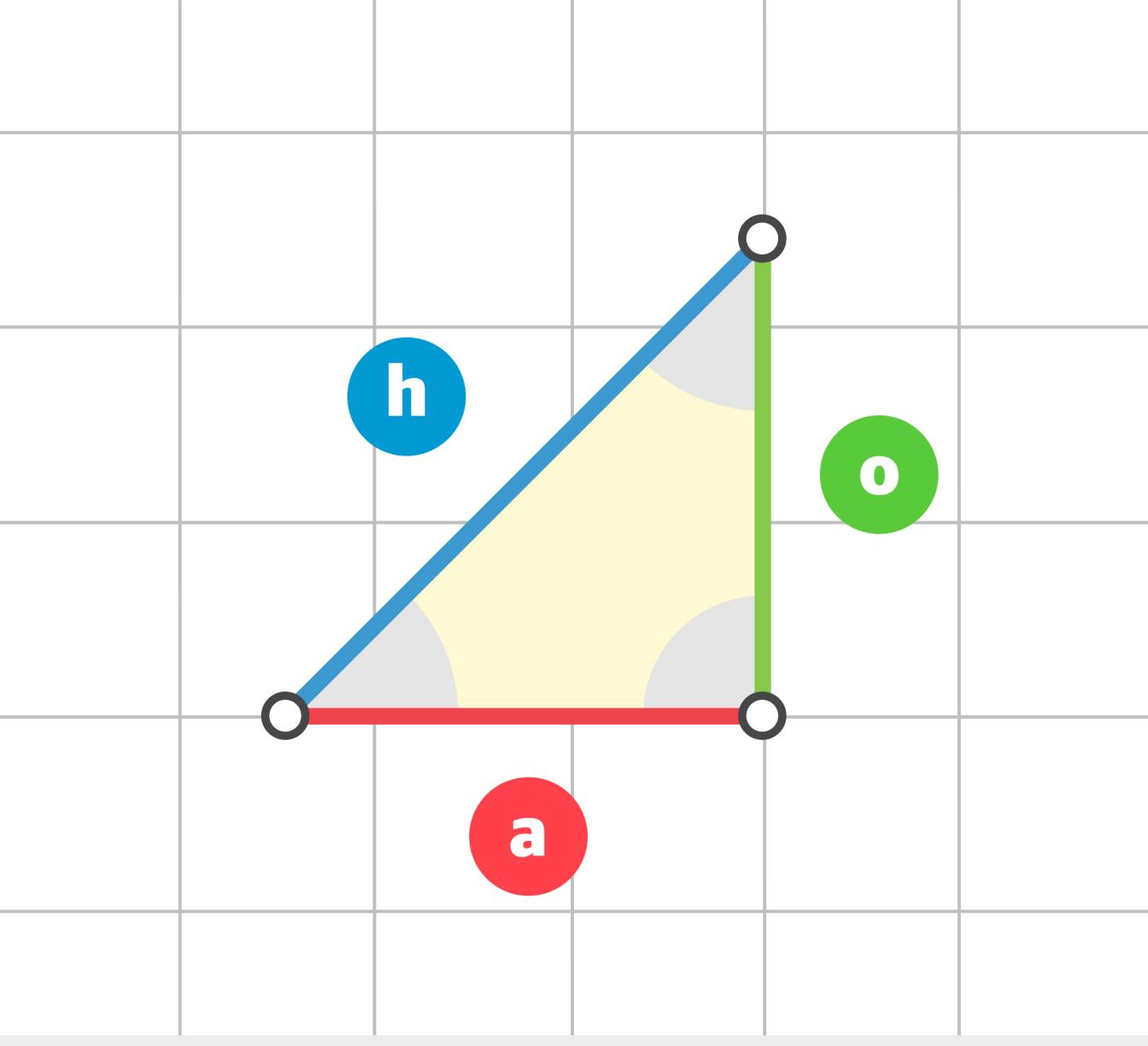
Si guardamos los cambios y volvemos a la escena, podremos experimentar de manera dinámica cómo las distintas variables afectan la forma y posición de la parábola, brindando una comprensión más profunda de la influencia de los coeficientes en la apariencia visual del shader.



(1.5.h)

Resumen de capítulo.

- En este capítulo, ilustramos la importancia de la función lineal en gráficos por computadora, comenzando con un análisis de cada variable en la ecuación. Luego, utilizamos la función lineal para dibujar un segmento entre dos puntos en Shader Graph con HLSL, demostrando cómo los distintos valores en las variables determinan el comportamiento de la línea en un plano cartesiano.
- El capítulo continúa con una exploración y desarrollo de una primera área geométrica, enfocada en un círculo. Esta sección cubre conceptos fundamentales como la importancia de los gráficos de tipo Sub Graph y los archivos .cgincl en el desarrollo de shaders. Posteriormente, la discusión continua con el análisis de las funciones cuadráticas, explorando su aplicación en la generación de paráboles. Finalmente, concluimos con la descomposición de la ecuación para mostrar cómo los coeficientes influencian la forma y posición de la parábola.

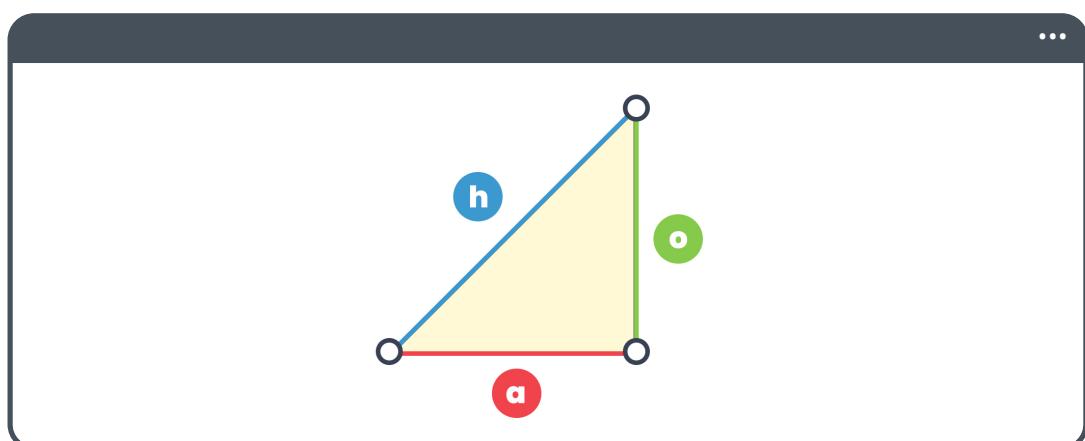


Capítulo 2
Funciones trigonométricas.

En este capítulo, te introducirás en el mundo de las funciones trigonométricas, una herramienta esencial en tu arsenal como programador, diseñador o artista técnico. Explorarás cómo visualizar funciones trigonométricas y examinarás las bases necesarias para comprender y aplicar estos principios matemáticos en HLSL. Posteriormente, dejarás las abstracciones a un lado para centrarte en dos operaciones esenciales: la reflexión y rotación de puntos en dos dimensiones. Utilizando ejemplos prácticos, verás cómo se aplican estos conceptos para generar figuras o formas dentro de tu proyecto en Unity, ampliando tu conocimiento a un sinfín de posibilidades creativas y técnicas en el desarrollo de efectos visuales.

2.1 Funciones.

Las funciones trigonométricas, fundamentales en disciplinas como la física y la ingeniería, también desempeñan un papel crucial en simulaciones y animaciones digitales. En el ámbito de los gráficos por computadora, funciones como el seno *sin*, coseno *cos*, y tangente *tan* juegan roles indispensables en la creación de efectos visuales, variaciones de color, transformaciones espaciales y hasta en el diseño de patrones específicos como las ondas sinusoidales. Pero ¿cuál es el origen de estas funciones? Ellas emergen del estudio de las relaciones (proporciones) entre los ángulos y lados en un triángulo rectángulo, llamado así por contar con un ángulo de 90° (ángulo recto). Observemos el siguiente gráfico para profundizar en este concepto:



(2.1.a)

Considerando que un triángulo posee tres lados, definidos como h , a y o en la imagen 2.1.a, podemos explorar diversas combinaciones para comprender mejor sus relaciones y aplicaciones en trigonometría. De hecho, utilizando cada uno de estos lados, podemos formular un total de seis combinaciones, las cuales podemos definir como:

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

(2.1.b)

Cada una de estas razones representa una función trigonométrica fundamental. Por ejemplo, el seno describe la relación entre el lado opuesto al ángulo y la hipotenusa de un triángulo rectángulo, es decir:

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

(2.1.c)

Recíprocamente, la cosecante csc de un ángulo está dada por:

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

(2.1.d)

Consecutivamente, el coseno de un ángulo está dado por:

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

(2.1.e)

Mientras que su recíproco, la secante *sec* está dada por:

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

(2.1.f)

Finalmente, la tangente de un ángulo está dada por:

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

(2.1.g)

Y su recíproco, la cotangente *cot* de un ángulo está dada por:

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

(2.1.h)

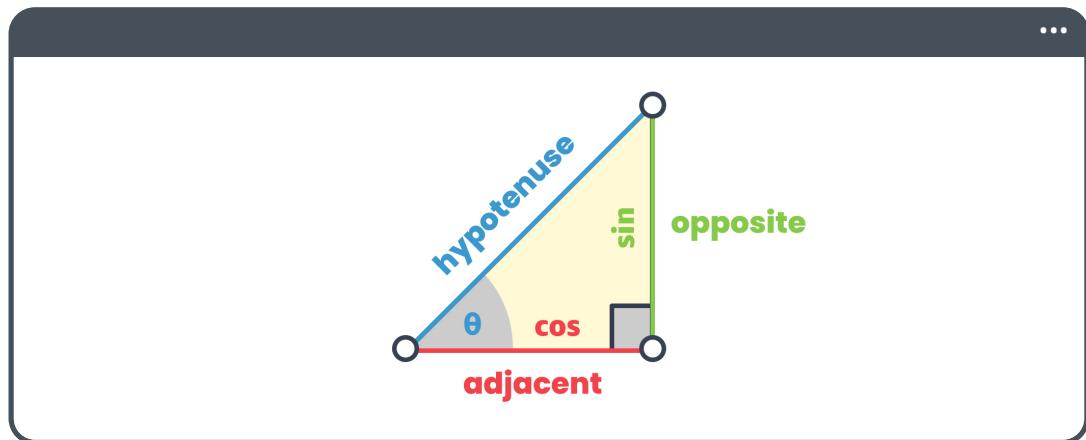
Dentro del lenguaje HLSL, se omite la inclusión directa de las funciones cosecante, secante y cotangente, ya que estas son recíprocas del seno, coseno y tangente, respectivamente. Esto implica que, en lugar de dedicar un análisis extenso a las primeras, profundizaremos en el seno, coseno y tangente por ser directamente aplicables en nuestro shader.

Para ilustrar, consideremos la función del seno, expresada como:

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

(2.1.i)

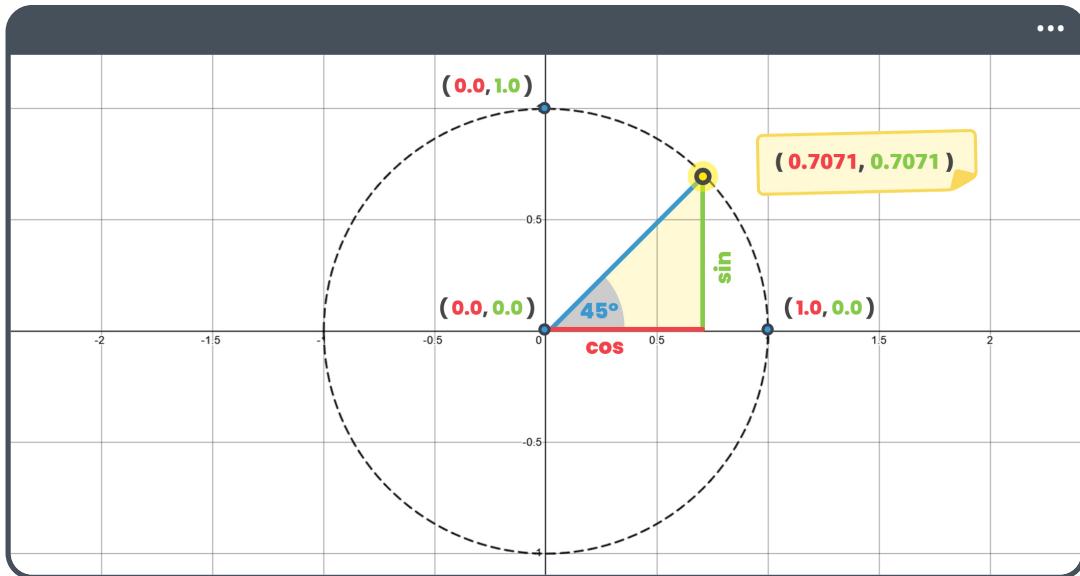
Antes de comenzar en los detalles de esta función, es crucial comprender qué representa y su conexión con la geometría de un triángulo rectángulo. El seno de un ángulo específico dentro de este tipo de triángulo se calcula tomando la longitud del lado opuesto al ángulo y dividiéndola por la longitud de la hipotenusa.



(2.1.j)

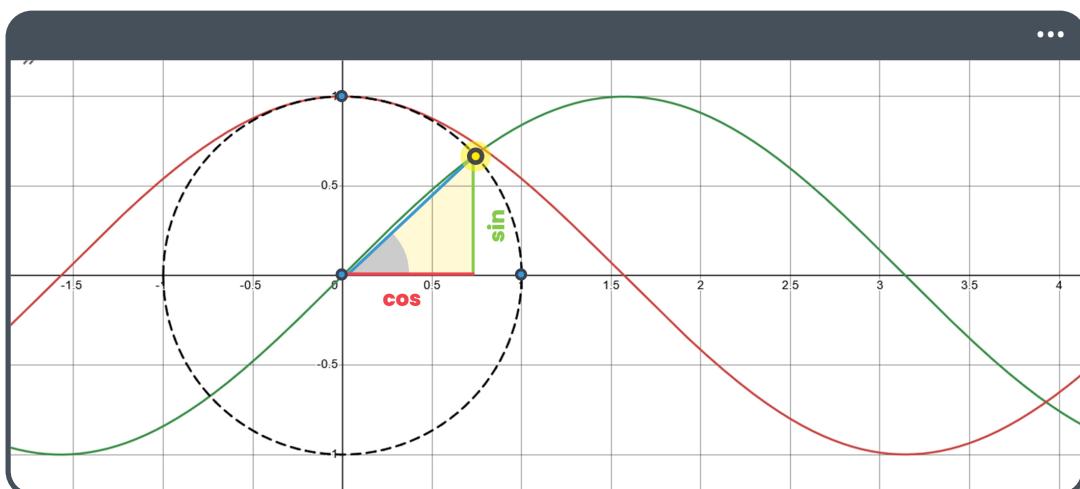
Al examinar con atención la imagen 2.1.j, notamos que se han empleado los colores rojo y verde para diferenciar las funciones seno y coseno en el contexto de un triángulo rectángulo. La elección de estos colores tiene un propósito en particular: facilitan la comprensión de su aplicación sobre las coordenadas UV.

Teniendo en cuenta una hipotenusa de longitud constante, denotada como h , y ubicando un punto p en su extremo, al rotar este punto alrededor del origen, se genera la región de un círculo completo. Esta observación resalta la relación entre las funciones trigonométricas y la geometría circular, un concepto que se presentó brevemente en la sección 1.3 del capítulo anterior.



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

Un aspecto interesante emerge al observar las curvas producidas en el plano cartesiano cuando calculamos el seno o el coseno de un número. Tomemos, por ejemplo, el eje x . Al evaluar las funciones seno y coseno sobre esta última, se revela un patrón distinto en el gráfico resultante, el cual, no tan sólo ilustra la naturaleza periódica de estas funciones, sino que también destaca su simetría y cómo se complementan entre sí a lo largo del espectro de valores en x .



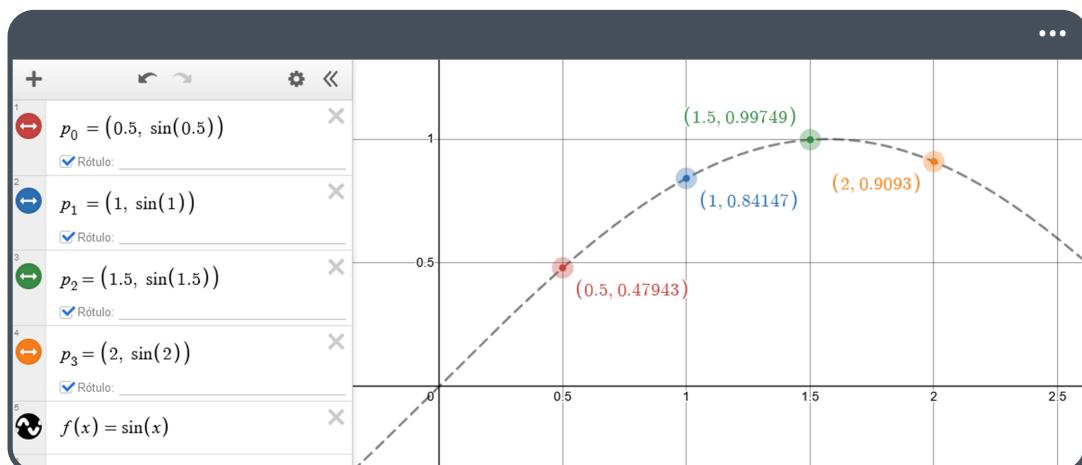
(2.1.l <https://www.desmos.com/calculator/y07lbi4req>)

La visualización de las curvas de seno y coseno se evidencian cuando se representan ambas funciones simultáneamente en un único sistema de coordenadas. Uno podría preguntarse, ¿cuál es la razón por la que se generan estas curvas? Es clave recordar que el sistema de coordenadas cartesianas se compone de una secuencia de valores numéricos que se extienden desde el infinito negativo hasta el infinito positivo, configurando así las posiciones en dos dimensiones. Para profundizar en esta dinámica, consideraremos la definición de cuatro puntos específicos a lo largo del eje x .

$$\begin{aligned} p_0 &= (0.5, \sin(0.5)) = (0.5, 0.479) \\ p_1 &= (1.0, \sin(1.0)) = (1.0, 0.841) \\ p_2 &= (1.5, \sin(1.5)) = (1.5, 0.997) \\ p_3 &= (2.0, \sin(2.0)) = (2.0, 0.909) \end{aligned}$$

(2.1.m)

Usaremos la función del seno para determinar el componente y de cada uno de estos puntos, lo que nos permitirá explorar más a fondo cómo las variaciones en x influyen directamente en las alturas (valores de y) generadas por la función del seno, trazando así la característica onda sinusoidal.

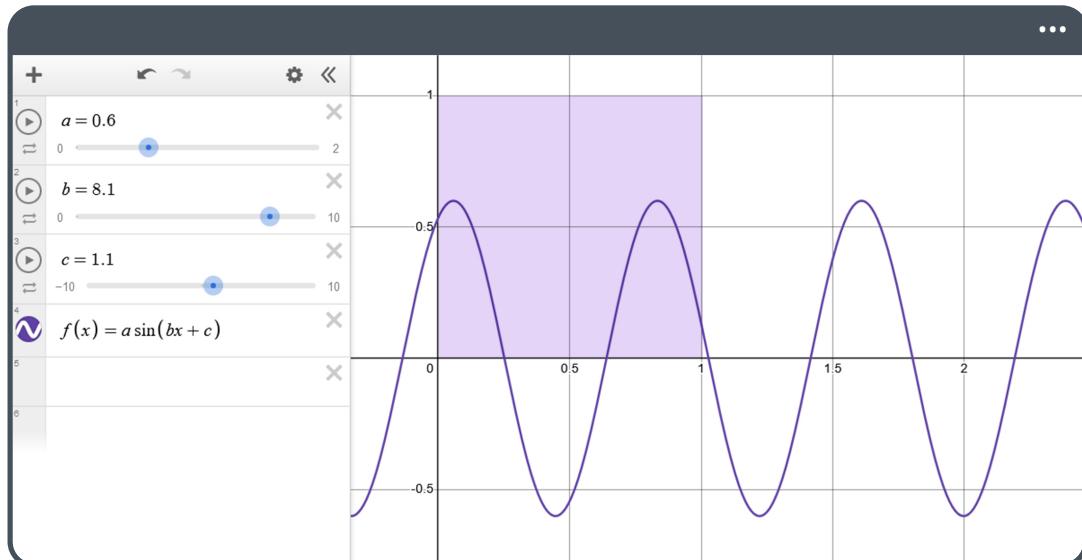
(2.1.n <https://www.desmos.com/calculator/gltqybvp8>)

Como se muestra en la referencia anterior, la curva de la función nace al calcular el seno de cada punto a lo largo de la coordenada x , y lo mismo ocurre con el coseno de x . Esta diferencia entre ambas funciones se explica porque el seno de un ángulo corresponde al coseno del mismo ángulo desplazado por 90° (o $\pi/2$ radianes). Es decir:

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

(2.1.ñ)

Las curvas admiten modificaciones en amplitud, frecuencia y fase. Si retomamos la imagen 2.1.a, observamos que la variable a modifica la amplitud de la onda, b afecta su frecuencia, y c ajusta su fase. ¿Cómo podríamos aprovechar esta función en el contexto de HLSL? Un ejemplo práctico sería la creación de simulaciones de fluidos bidimensionales, manipulando la coordenada x dentro de los parámetros UV.



(2.1.o <https://www.desmos.com/calculator/kns4ip5cd0>)

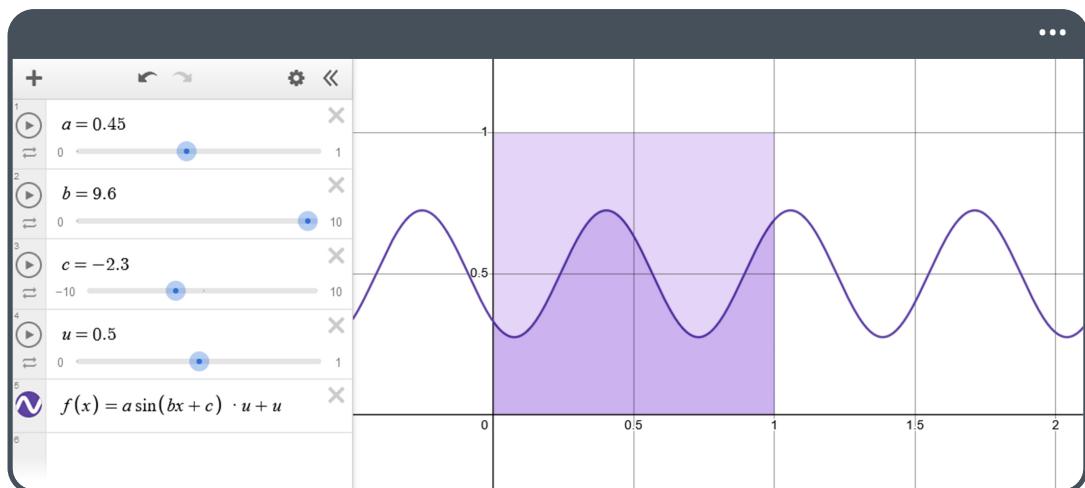
Al examinar la imagen 2.1.p, se puede apreciar la función previamente mencionada que, con un valor de $a = 0.6$, muestra una amplitud que oscila entre -0.6 y 0.6. No obstante, recordemos que el rango de las coordenadas UV difiere de este. Por ende, será

necesario ajustar la función para que su rango se encuentre entre 0.0 y 1.0. Para ello, introduciremos una nueva variable llamada u , asignándole el valor 0.5. Esta variable u nos facilitará recalibrar el punto central de la curva, de tal manera que su origen en la coordenada y quede a una distancia equivalente a su valor. Posteriormente, al aplicar la operación, multiplicaremos la función del seno por u y luego le sumamos u , quedando de la siguiente manera:

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

(2.1.p)

Es importante mencionar que también necesitaremos limitar la variable $a = 1$ para alinearla con el rango de las coordenadas UV.



(2.1.q <https://www.desmos.com/calculator/1scbrmw7zm>)

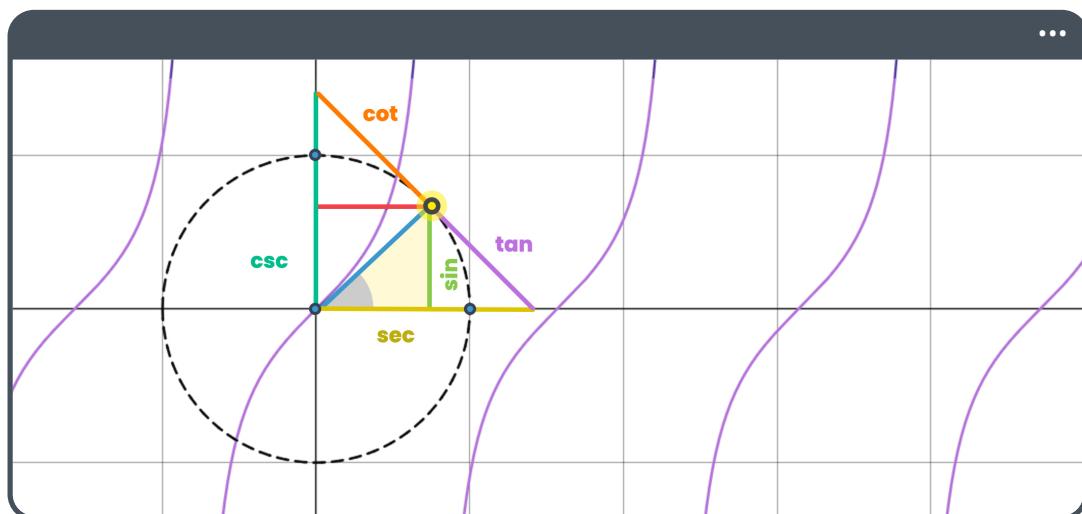
Hasta ahora, hemos explorado el comportamiento de los operadores seno y coseno en detalle. Pero ¿qué sucede con el operador tangente? La tangente se define como la relación entre la longitud del cateto opuesto y la del cateto adyacente, tal como observamos en la función 2.1.g. Así, cuando el ángulo de un punto con relación a su origen es mayor a cero y menor a 90° , el resultado de la tangente es positivo. ¿La razón? En este intervalo de

ángulos, tanto el cateto opuesto como el adyacente son positivos. Sin embargo, a medida que el ángulo se incrementa más allá de este rango, el signo del resultado de la tangente varía.

Para ángulos en los rangos de:

$$\begin{array}{lll} 0^\circ < \theta < 90^\circ & 90^\circ < \theta < 180^\circ & 180^\circ < \theta < 270^\circ & 270^\circ < \theta < 360^\circ \\ \tan(a)+ & \tan(a)- & \tan(a)+ & \tan(a)- \end{array}$$

Esta característica de cambio de signo refleja la periodicidad y el comportamiento único de la tangente en diferentes cuadrantes. Al graficar la curva de la función tangente, este patrón de alternancia de signos se traduce en una serie de discontinuidades que definen la naturaleza de esta función trigonométrica.



(2.1.r <https://www.desmos.com/calculator/czpygnurms>)

La tangente resulta ser una herramienta versátil en la creación de diversos efectos visuales, abarcando desde rotaciones hasta la aplicación de máscaras de textura. Su aplicación va más allá, facilitando la estimación de aspectos como la altura de un objeto o la orientación de un punto a lo largo de una curva. Esto se visualiza claramente cuando consideramos la pendiente m de una función lineal, representada por $y = mx + b$.

Tomemos, por ejemplo, un punto p , definido como:

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

(2.1.s)

Si consideramos m como la pendiente de la recta, definida como:

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

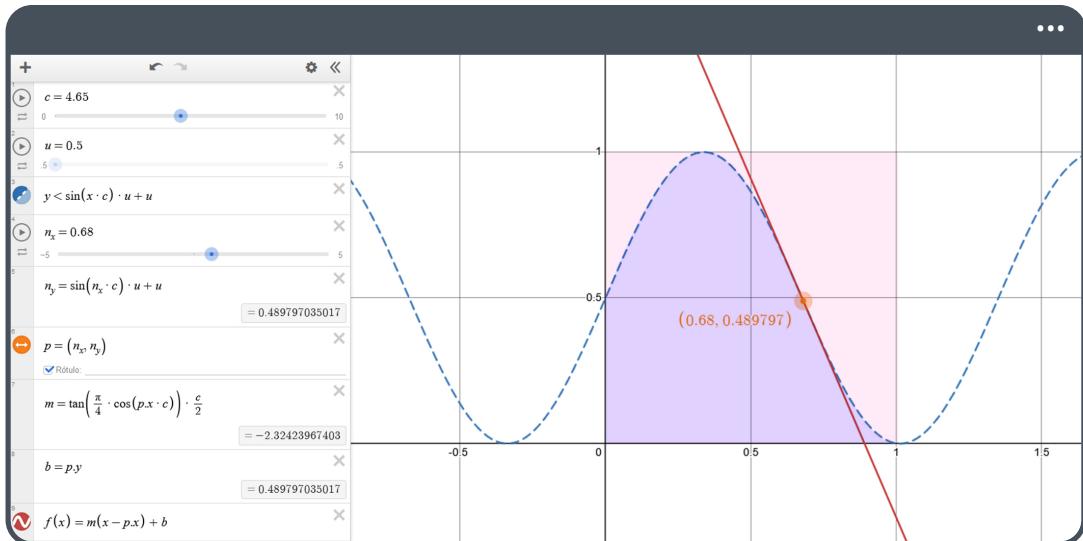
(2.1.t)

Y establecemos b , el término de intersección con el eje y , igual a la coordenada p_y del punto:

$$b = p_y$$

(2.1.u)

Así, es posible adaptar una línea recta para que se ajuste a una curva sinusoidal, logrando una aproximación lineal del punto p .



(2.1.v <https://www.desmos.com/calculator/f8hjcush5a>)

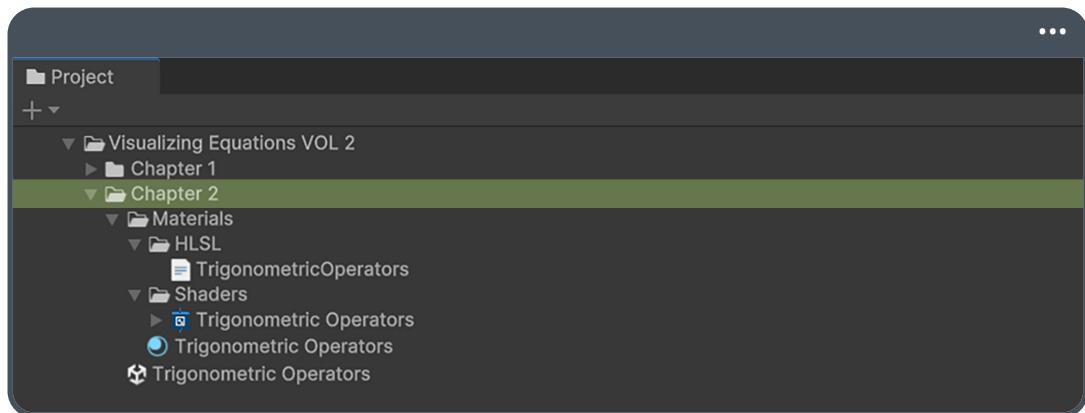
De la imagen 2.1.w, considerando el punto definido como $p = (n_x, n_y)$, si modificamos el valor de n_x , observaremos que este se desplaza a lo largo de la curva. Además, la tangente determina su orientación con relación a la posición actual en que se encuentra.

2.2 Visualizando funciones trigonométricas en HLSL.

Dado que los operadores trigonométricos son esenciales en programación, es común encontrarlos implementados en la mayoría de los lenguajes, y HLSL no es la excepción. Por ende, comprender su funcionamiento mientras trabajamos con ellos será una tarea fundamental.

Para comenzar esta sección, iremos a nuestro proyecto y crearemos un nuevo shader de tipo **Unlit Shader Graph**, al que nombraremos **Trigonometric Operators**. Continuando nuestra práctica habitual, generaremos un material y un archivo HLSL con el mismo nombre que el shader que acabamos de crear, y nos aseguraremos de asignar el shader al material desde el Inspector.

Manteniendo la estructura introducida en el primer capítulo, nuestro proyecto debería presentarse de la siguiente forma.



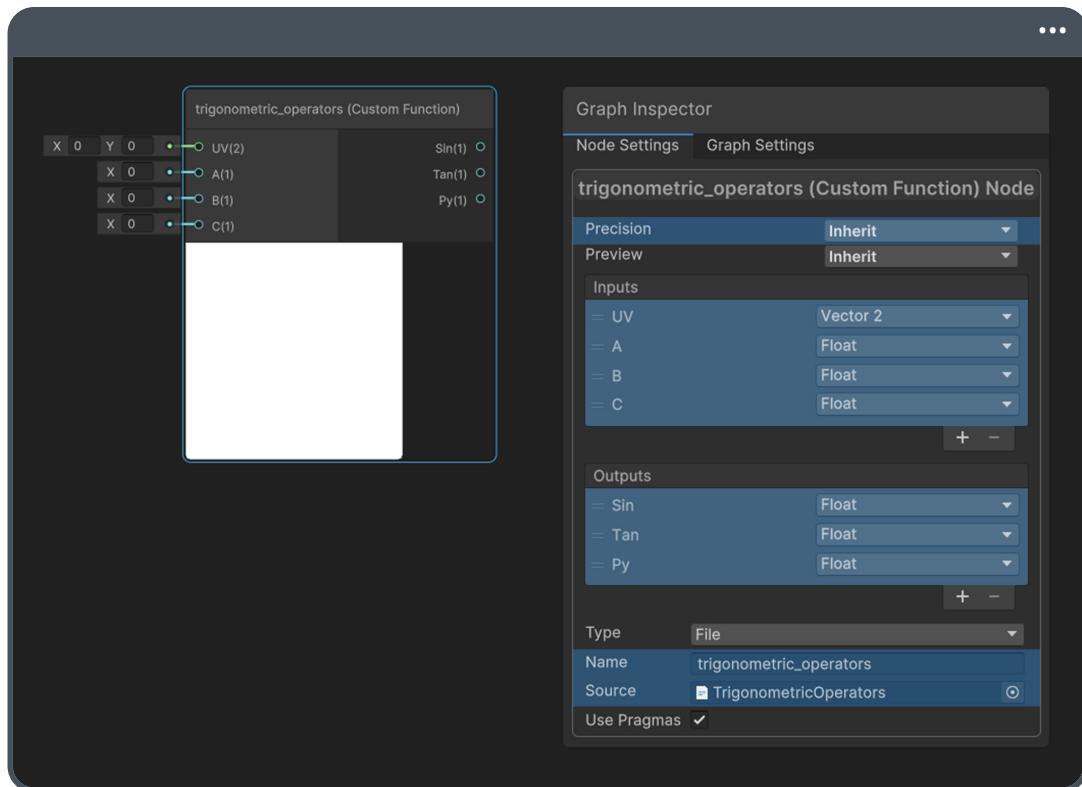
(2.2.a)

Es cierto que los nodos **Sine**, **Cosine** y **Tangent** ya están implementados en Shaders Graph. Sin embargo, optaremos nuevamente por usar un nodo **Custom Function** para comprender profundamente el proceso y lógica detrás de estos cálculos trigonométricos. Basándonos en algunas de las operaciones discutidas previamente, no solo visualizaremos las funciones del seno y tangente, sino también integraremos un método que nos permita ajustar la posición en V del círculo **Origin**, facilitando así la visualización del punto central.

Para comenzar, abriremos nuestro shader y crearemos un nodo **Custom Function**. Tomando como referencia la función ilustrada en la figura 2.1.i, el primer paso que realizaremos será incorporar **a**, **b** y **c** como variables de entrada, incluyendo también las coordenadas **UV**.

En este caso, buscamos visualizar tres resultados distintos: el seno, la tangente, y la posición del punto de origen. Por ende, como salida añadiremos tres valores flotantes, los cuales nombraremos **Sin**, **Tan** y **Py**.

Además, incluiremos el archivo **TrigonometricOperators.hlsl** como **Source**, y utilizaremos **trigonometric_operators** como denominación. Si el proceso se ha seguido correctamente, nuestro nodo se debería apreciar tal como se muestra a continuación.



(2.2.b)

Comenzaremos declarando el método `trigonometric_operators_float()` en nuestro shader. Para ello, incluiremos como argumentos las variables mencionadas anteriormente, así como los tres valores de salida: **Sin**, **Tan** y **Py**, los cuales inicializaremos en cero para garantizar que nuestro nodo compile sin errores.

```

1 void trigonometric_operators_float(in float2 uv, in float a, in float b, in
2                                     float c, out float Sin, out float Tan, out float Py)
3 {
4     Sin = 0.0;
5     Tan = 0.0;
6     Py = 0.0;
7 }
```

Antes de proceder a implementar las funciones específicas para cada salida, es imprescindible incluir las constantes matemáticas π y 2π . Estas jugarán un papel crucial en las operaciones trigonométricas que desarrollaremos a continuación, asegurando la precisión y eficacia de nuestros cálculos.

```

1 #define PI 3.14159265358
2 #define TWO_PI 6.28318530718
3
4 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) { ... }
```

La constante **PI** se refiere a medio giro de un círculo, mientras que **TWO_PI** equivale a un giro completo. Aunque estas constantes podrían declararse e inicializarse directamente en cada método que las requiera, optaremos por definirlas a través de macros para facilitar su reutilización. Por lo general, se acostumbra a declarar e inicializar directivas **#define** en archivos de inclusión CG **.cginc**, lo cual permite su invocación única. Por ejemplo, es posible crear un archivo **ShaderCommon.cginc**, incluir ahí tanto a **PI** como **TWO_PI**, y posteriormente incorporarlos en nuestro shader mediante la directiva **#include**.

En relación con el área total del Quad que utilizaremos para la proyección gráfica tanto del seno como de la tangente, y el punto central, nos aseguraremos de añadir dinamismo a las figuras generadas. Sin embargo, centraremos la acción en el medio del Quad. Para este fin, emplearemos la siguiente fórmula matemática:

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

(2.2.c)

Observando la función anterior, notamos que la variable ***u*** extiende la función para limitar la curva producida por el seno ***sin***, dentro de un intervalo entre [0.0 : 1.0].

Su implementación en HLSL luce de la siguiente manera:

```
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 }
```

En esta función, el parámetro **x** actúa como el valor de entrada sobre el cual se calculará la función. El parámetro **c** ajusta el desplazamiento o fase de la onda, **a** es un coeficiente que escala su amplitud, y **b** modifica la frecuencia.

A continuación, emplearemos el método recientemente implementado para modelar la forma de la onda mediante un nuevo método que denominaremos **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 }
```

En el ejemplo previo, observamos que la variable **fx** (línea 12) se asigna inicialmente al componente vertical de las coordenadas UV, es decir **uv.y**. En contraste, la variable **x** (línea 13) representa el componente horizontal, o la coordenada **uv.x**. La línea 15 muestra cómo **f** se calcula usando el método **sinusoidal()**, el cual incluye parámetros para amplitud, fase y frecuencia. Posteriormente, **fx** se ajusta restando el valor de **f**, lo que resulta en un desplazamiento vertical de la onda en concordancia con el valor de la onda

sinusoidal para una posición horizontal dada. Para completar, añadiremos un método que nos permita visualizar la orientación del punto sobre la onda.

```
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 }
```

En este proceso, volvemos a asignar el componente **uv.y** de las coordenadas UV a **fx** (línea 23) y el componente **uv.x** a la variable **x** (línea 24). Luego, establecemos la constante **px** igual a 0.5, representando el centro de la posición de la onda en el eje **x** (línea 26). Posteriormente, **py** (línea 27) calcula la posición vertical de la onda en el punto central utilizando el método **sinusoidal()**, mientras que la variable **m** (línea 28) define la pendiente en el punto **px**. Utilizando la ecuación lineal **mx + b**, calculamos **f**, el desplazamiento vertical para la posición **x** actual, basado en la pendiente **m** y la referencia vertical **py**. Ahora, únicamente faltaría asignar estos valores a cada output en el método **trigonometric_operators_float()**.

```

36 void trigonometric_operators_float(in float2 uv, in float a, in float b, in
37   float c, out float Sin, out float Tan, out float Py)
38 {
39     Sin = sine_wave(uv, a, b, c);
40     Tan = tan_wave(uv, a, b, c);
41     Py = sinusoidal(0.5, a, b, c);

```

Considerando que la variable **c**, que define la fase de la onda, se mantiene constante, no observaremos variaciones significativas en la forma final de la onda. Por ello, integraremos un nuevo método denominado **normalized_time()** a nuestro código. Esta operación, como indica su nombre, normalizará el tiempo mediante la parte fraccional de la división del tiempo **_Time.y** por la cantidad de segundos **s**.

```

36 float normalized_time(float s)
37 {
38     return frac(_Time.y / s);
39 }

```

Para comprender su funcionamiento, es importante enfocarse en la variable **_Time.y**, que devuelve el tiempo transcurrido desde que se cargó un nivel. Esto significa que, al iniciar una escena (al presionar el botón Play), el tiempo empieza a contar desde cero hacia adelante. Sin embargo, esto genera un problema: aunque los tipos de datos como **fixed**, **half** o **float** ofrecen una gran resolución, no son ilimitados. Por lo tanto, es esencial restringirlos en algún momento para prevenir errores gráficos o de procesamiento. Aquí es donde entra en juego la función **frac()**, que retorna la parte fraccional del tiempo, es decir, los valores que van desde 0.0 a 0.99. De esta manera, cuando el valor fraccionario de **_Time.y** alcanza 0.99, se reinicia a 0.0, creando un ciclo y optimizando el proceso a la vez. La variable **s** divide al tiempo, permitiéndonos realizar el cálculo en segundos, por ejemplo: si **s = 2**, la operación completará su ciclo en dos segundos.

```

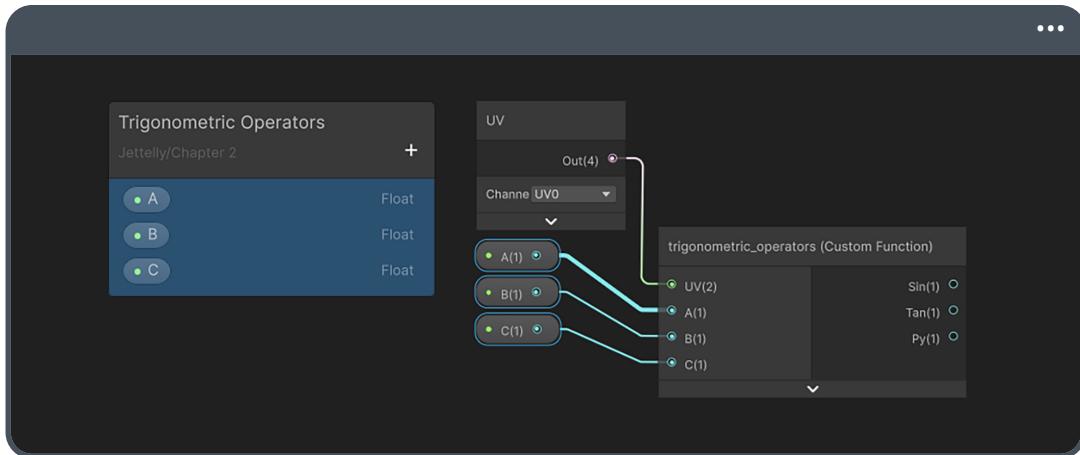
41 void trigonometric_operators_float(in float2 uv, in float a, in float b, in
42   float c, out float Sin, out float Tan, out float Py)
43 {
44     float t = normalized_time(c) * (TWO_PI / b);
45     Sin = sine_wave(uv, a, b, t);
46     Tan = tan_wave(uv, a, b, t);
47     Py = sinusoidal(0.5, a, b,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

Como podemos observar, en la línea de código 43, hemos declarado e inicializado una nueva variable **t**, que almacena el resultado del método **normalized_time()** utilizando **c** como parámetro. La operación **TWO_PI / b** se emplea para cubrir un ciclo completo de la onda, ajustándose a un cambio en la frecuencia **b**. Posteriormente, **t** se introduce como cuarto valor en cada método (líneas 45, 46 y 47), resultando en un número distinto para cada output en el nodo. De esta manera, la variable **c** determina el tiempo en segundos que el ciclo de la onda sinusoidal tarda en repetirse.

Hasta este punto, nuestro nodo debería compilar sin inconvenientes. En esta instancia, estamos optando por **float** como la resolución para el nodo, eliminando la necesidad de ajustar el valor de la propiedad **Precision** en el **Graph Inspector**.

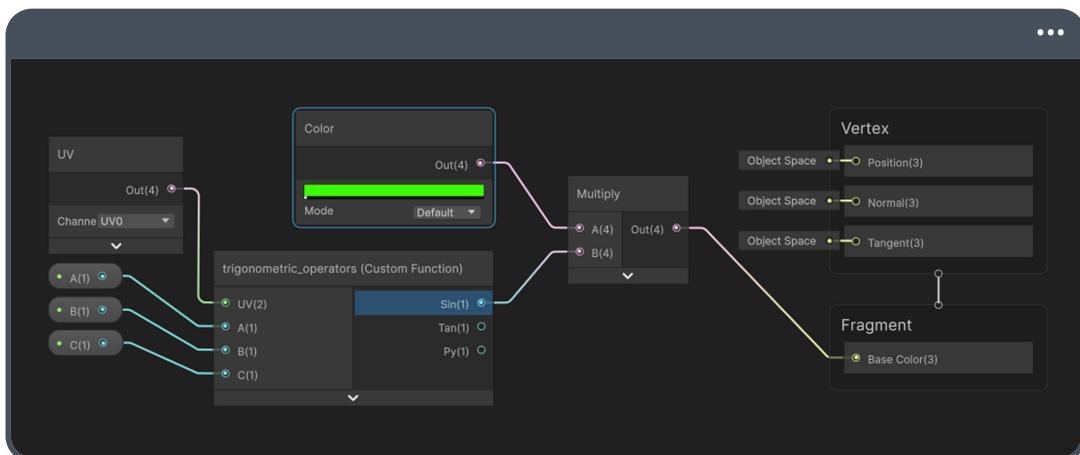
Como es habitual, procederemos a añadir las propiedades necesarias en el **Blackboard** para permitir la modificación dinámica de los resultados de nuestro nodo desde el Inspector, asegurándonos de restringir sus valores a los siguientes intervalos:

- Para la propiedad **A**, el rango será de [0.0 : 1.0], con 0.5 como valor por defecto.
- Respecto a **B**, estará entre [1 : 50], con 1.0 como valor por defecto.
- Y para **C**, el rango será de [1 : 5], con 1.0 como valor por defecto.



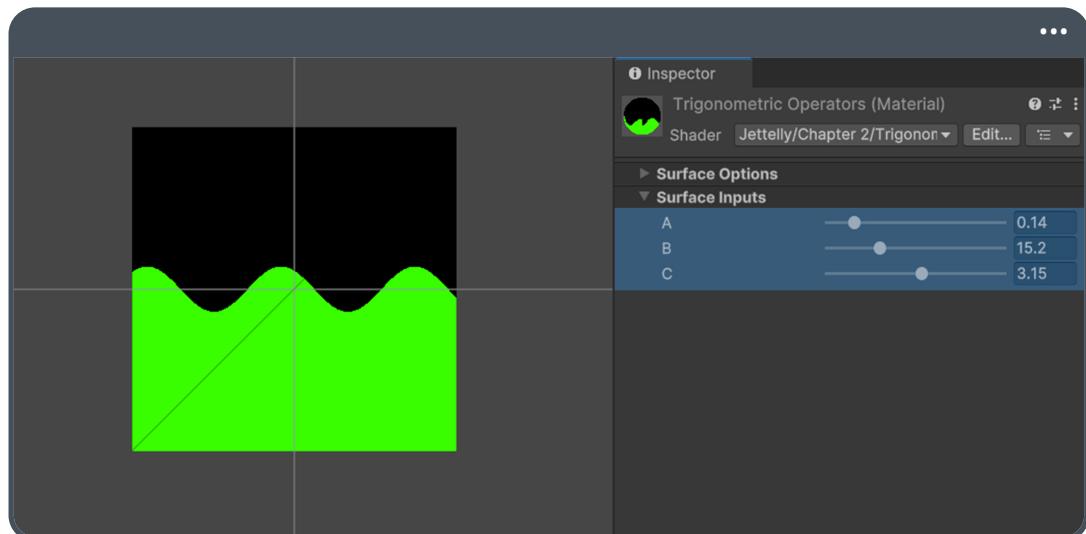
(2.2.d Se han vinculado las propiedades)

Si prestamos atención a la imagen anterior, notaremos que cada propiedad ha sido vinculada al nodo **trigonometric_operators**, incluidas las coordenadas UV. A continuación, procederemos a proyectar la onda sinusoidal sobre el Quad en nuestra escena, utilizando una tonalidad específica para su visualización. Para ello, integraremos un nodo **Color** en nuestro shader, al cual le asignaremos un color verde por defecto. Luego, multiplicaremos el valor de salida **Sin** por el **Color** obtenido, y el resultado lo utilizaremos como **Base Color** en nuestro shader.



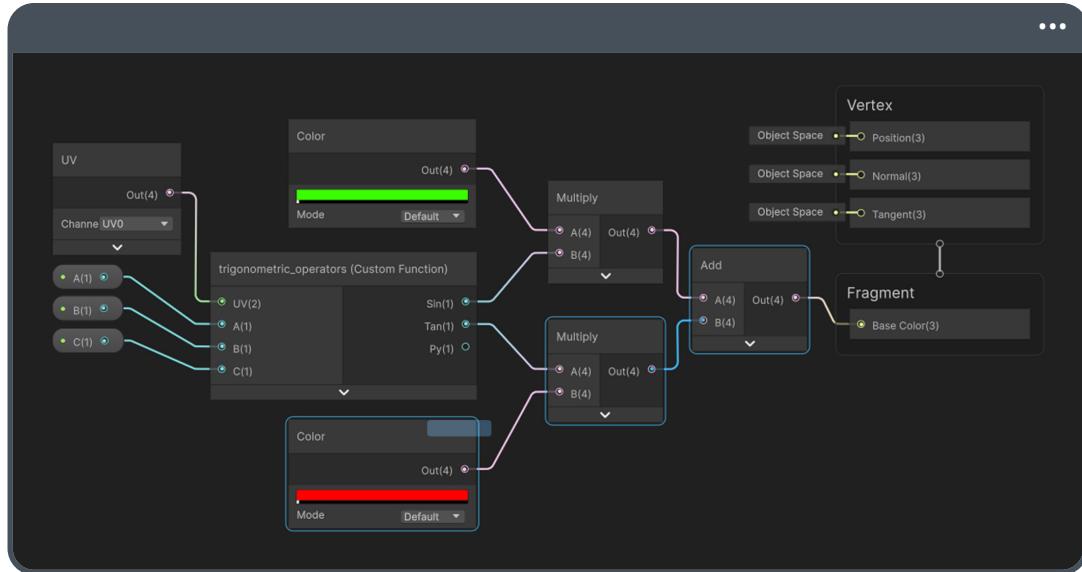
(2.2.e)

Al guardar nuestros cambios y regresar a la escena, tendremos la capacidad de visualizar y ajustar el comportamiento de la onda sinusoidal directamente a través de las propiedades del material **Trigonometric Operators**, el cual fue creado al inicio de esta sección.



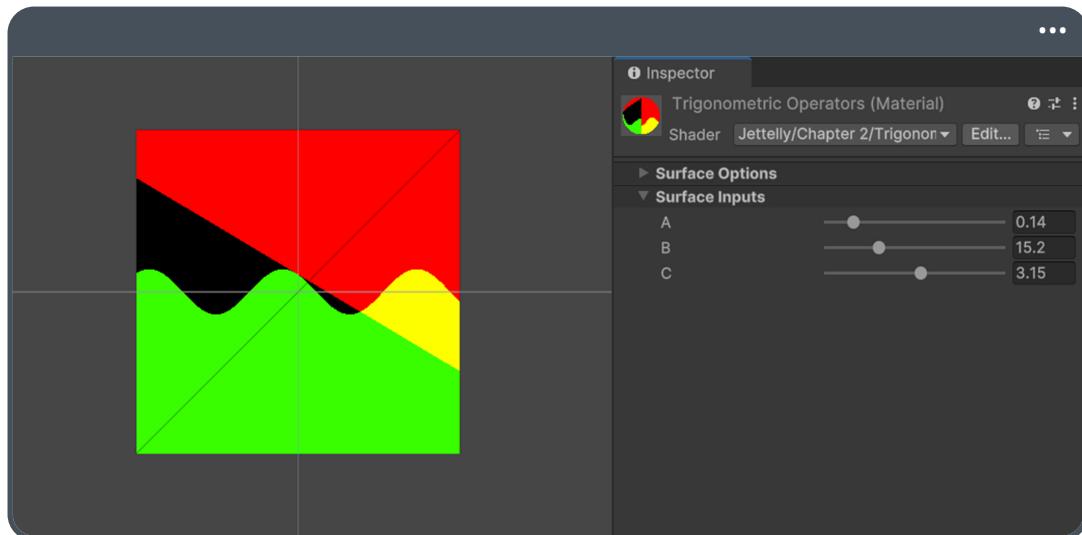
(2.2.f)

A continuación, integraremos la pendiente a la visualización. Para ello, regresamos a nuestro shader y añadimos el valor de **Tan** al resultado previamente obtenido de multiplicar **Sin** por el color verde. No obstante, antes de proceder, multiplicaremos el valor de **Tan** por un color rojo. Esta acción nos permitirá visualizar claramente ambas funciones.



(2.2.g)

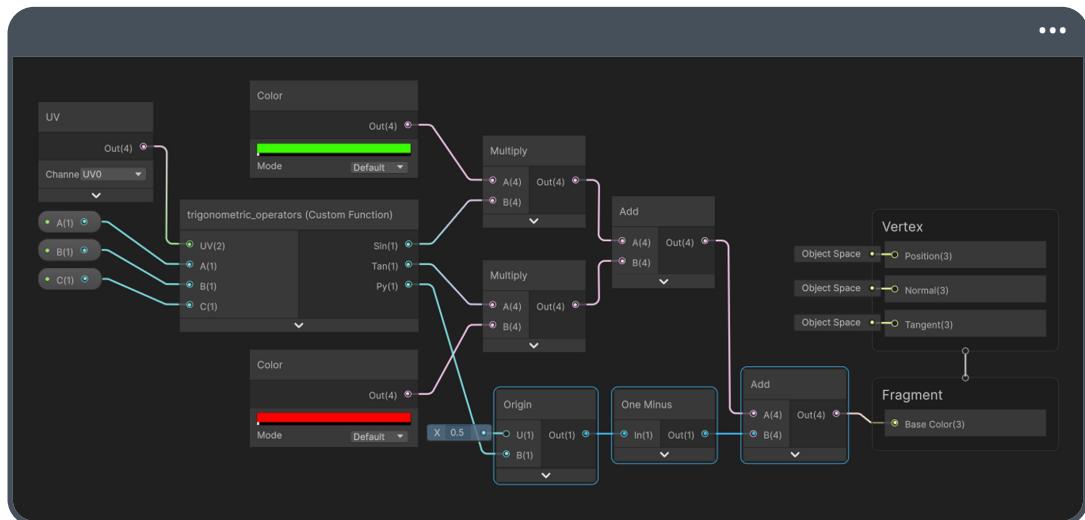
Al guardar nuestros cambios y regresar a la escena una vez más, podremos apreciar cómo se comportan ambas funciones cuando operan en conjunto.



(2.2.h)

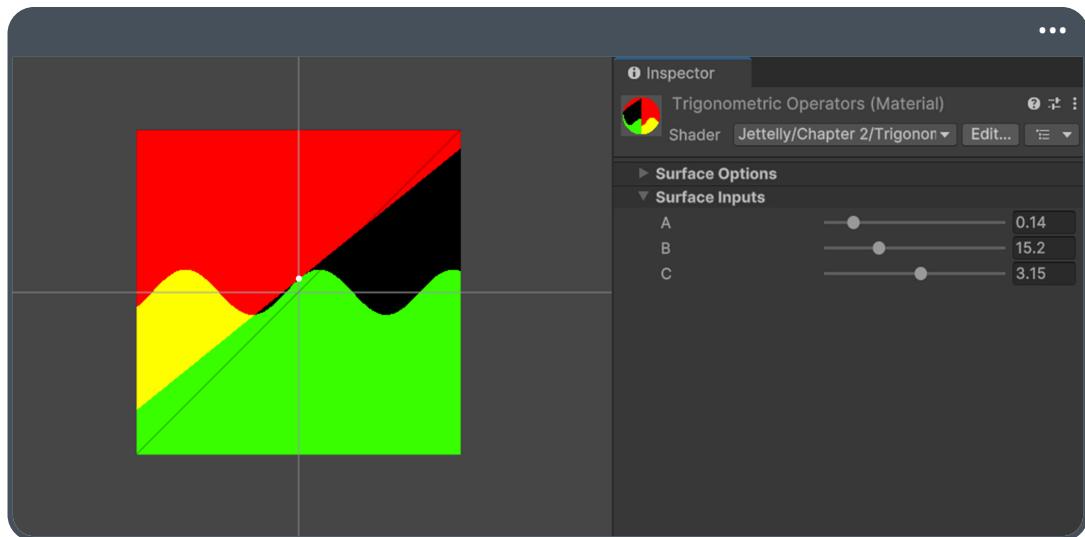
Un aspecto notable es la ausencia del punto central; aunque conocemos que $x = 0.5$ y que la coordenada y varía según la amplitud de la onda, carecemos de un indicador gráfico

que confirme la precisión de nuestros cálculos. Para abordar esta cuestión, regresaremos a nuestro shader e integraremos el nodo **Origin** en nuestra operación. Este nodo, como ya sabemos, facilita la visualización de un punto, representado por un círculo de color negro. Dado que el fondo de nuestra onda sinusoidal también es negro, cambiaremos el color del círculo para asegurar su visibilidad en el gráfico. Finalmente, incorporaremos este punto a la operación global de nuestro shader, enriqueciendo así nuestra visualización.



(2.2.i)

Si observamos la propiedad **u** del nodo **Origin** en la imagen 2.2.i, notaremos que se ha asignado un valor de 0.5 para alinearla con las posiciones tanto de la onda sinusoidal como de la pendiente. Además, hemos empleado el nodo **One Minus** para invertir el color del nodo a blanco. Este resultado se ha incluido como parte de la operación del nodo. Al volver a la escena, podremos apreciar cómo la pendiente se ajusta según la posición del punto central.



(2.2.j Proyección de la onda sinusoidal)

2.3 Reflexión de un punto.

Cuando discutimos el concepto de reflexión, nos referimos esencialmente a la inversión de la posición de un punto con respecto a un eje o plano específico, emulando el comportamiento de un espejo. Este fenómeno es de vital importancia en la creación de efectos visuales, ya que no solo enriquece la inmersión y el realismo de las escenas, sino que también aporta una dimensión adicional a la estética del entorno virtual. Más allá de simular superficies reflectantes, la reflexión desempeña un papel crucial en técnicas de iluminación avanzada, tales como el trazado de rayos (Ray Tracing), en el que la luz se refleja en los objetos del entorno, imitando las interacciones lumínicas del mundo real.

Para comprender mejor los principios subyacentes de la reflexión, consideremos un punto p denotado como $p = (3.0, 3.0)$. Al reflejar este punto con respecto al eje y , obtenemos un nuevo punto p' que resulta en $p' = (-3.0, 3.0)$, evidenciando que la coordenada x se invierte mientras que la coordenada y permanece inalterada. De manera similar, al reflejar p con respecto al eje x , el resultado es $p = (3.0, -3.0)$, donde la coordenada y cambia su signo, manteniendo constante la coordenada x . Estas operaciones nos introducen a las fórmulas generales para la reflexión:

Reflexión con respecto al eje y .

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

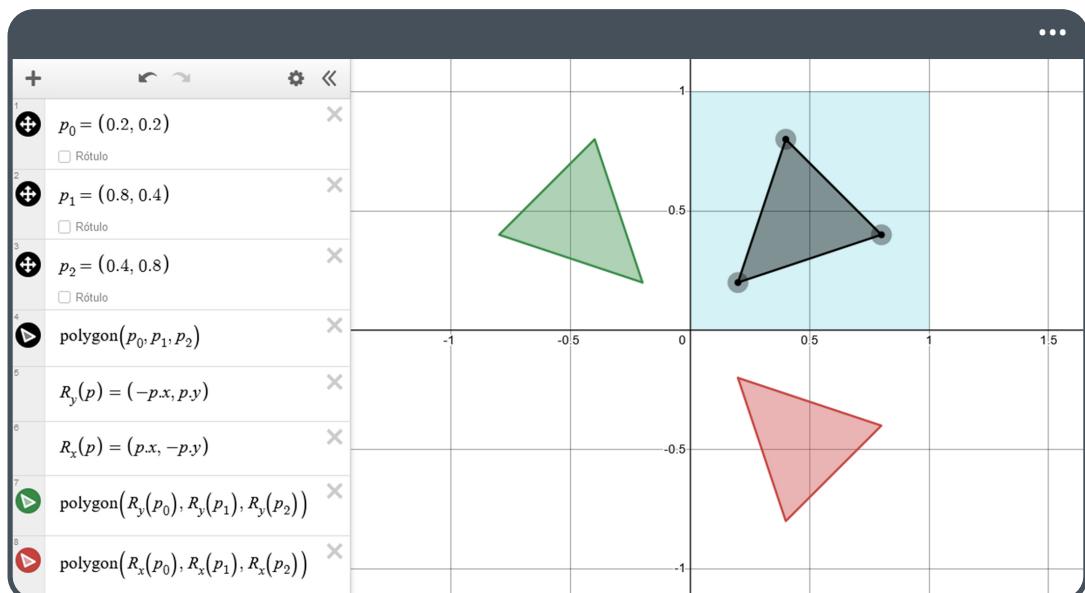
(2.3.a)

Reflexión con respecto al eje x .

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

(2.3.b)

Utilizando estas fórmulas, si trazamos un polígono empleando los puntos $p_0 = (0.2, 0.2)$, $p_1 = (0.8, 0.4)$ y $p_2 = (0.4, 0.8)$, y posteriormente lo proyectamos sobre un plano cartesiano bidimensional, la reflexión de este polígono en cada eje revelaría las siguientes transformaciones visuales.



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

Este proceso no solo demuestra la influencia directa de las reflexiones en la percepción visual de objetos y escenas, sino que también resalta la importancia de entender matemáticamente estas transformaciones para su aplicación práctica en gráficos por computadora.

En la imagen 2.3.c, el polígono de color rojo representa a la reflexión en el eje x del polígono original, formado por los puntos p_0 , p_1 y p_2 , el cual se presenta en color negro. Por otro lado, el polígono de color verde ilustra la reflexión del mismo polígono negro, sobre el eje y . Para reflejar un polígono en el tercer cuadrante, una estrategia simple consiste en hacer negativas ambas componentes de un punto p , resultando en:

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

(2.3.d)

No obstante, surge una interrogante más compleja cuando queremos aplicar una reflexión sobre una línea que no corresponde a los ejes xy , es decir, ¿cuál sería la fórmula necesaria para reflejar un punto sobre una recta definida como $mx + b$?

Para ello, prestaremos atención a la siguiente fórmula:

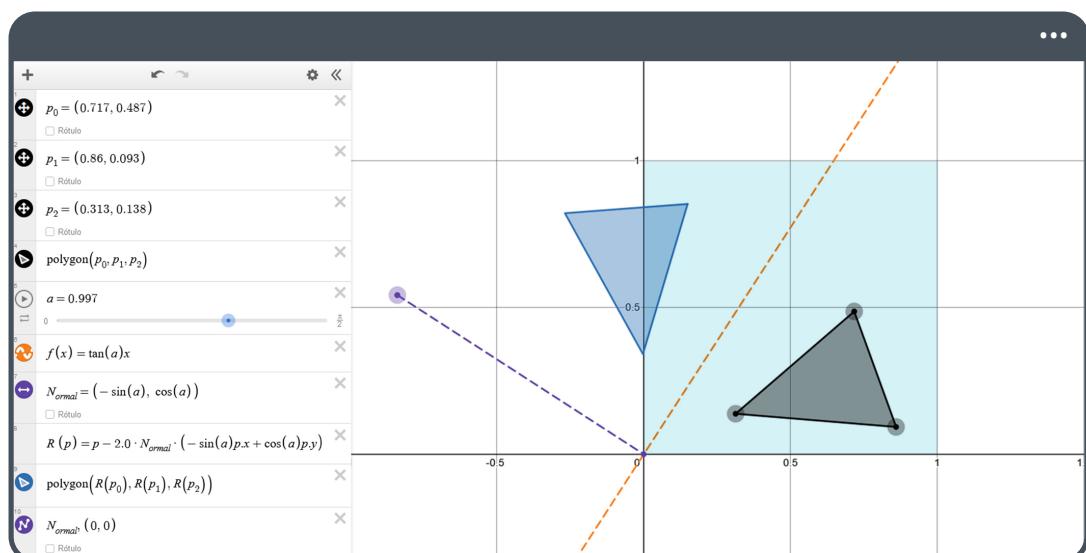
$$p' = p - 2(-\sin(a), \cos(a))(-\sin(a)p_x + \cos(a)p_y)$$

(2.3.e)

De esta ecuación, hay varios conceptos que ya comprendemos. Por ejemplo, reconocemos que los operadores seno y coseno se derivan de las relaciones en un triángulo rectángulo y están vinculados con las propiedades de un círculo completo. Además, sabemos que ambas funciones presentan un desfase de 90° entre sí. Entonces, ¿cómo interpretamos correctamente esta fórmula? El término p denota al punto original, mientras que p' indica

la posición del punto después de la reflexión, la cual se determina según el ángulo a . Por lo tanto, a especifica la orientación de la línea de reflexión respecto a los ejes coordenados.

Para visualizar esto de manera más intuitiva, podemos considerar que la línea de reflexión actúa como un espejo inclinado, donde a representa el ángulo de inclinación con respecto al eje x . La reflexión de un punto sobre esta línea no sólo implica una inversión en sus coordenadas, sino también una rotación que depende directamente del ángulo de inclinación de la línea.



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

La línea de reflexión puede representarse de diversas maneras. No obstante, su papel en la operación mencionada previamente es crucial, ya que la reflexión se lleva a cabo a través de su vector normal n , definido como,

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

(2.3.g)

Como se podría anticipar, este vector es perpendicular a la línea de reflexión.

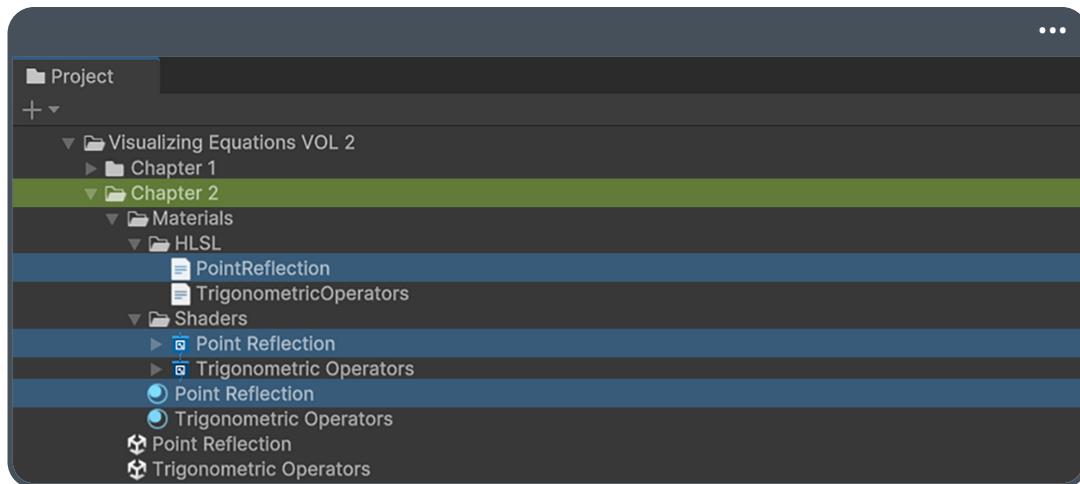
Reflejar un punto a través de una línea implica encontrar un punto p' de tal manera que la línea actúe como la bisectriz perpendicular del segmento que une p y p' . La operación $-\sin(a)p_x + \cos(a)p_y$ calcula una proyección escalar (producto punto) que determina qué tan lejos, en la dirección del vector normal n , se encuentra el punto original p . Al multiplicar este escalar por 2 y, a continuación, por el propio vector normal, se obtiene el vector de desplazamiento necesario para transitar desde p a p' a través de la línea de reflexión.

Explicado de manera sencilla, esta ecuación traslada el punto p en una dirección perpendicular a la línea de reflexión, definida por el ángulo a , exactamente el doble de la distancia más corta entre p y la recta. Así, garantizamos que p' se ubique en el lado opuesto de la línea, manteniendo la misma distancia que separa a p de esta.

2.4 Reflexión de un punto en HLSL.

Tras haber explorado la representación gráfica de diversas funciones y familiarizarnos con la estructura de nuestro proyecto, dedicaremos esta sección a la implementación de la reflexión de un punto en HLSL. Para ilustrar este concepto, agregaremos un shader de tipo **Unlit Shader Graph**, un material, y un script de extensión **.hlsł**, todos específicamente orientados a este fin. De manera similar a nuestra exploración anterior con los operadores trigonométricos, en esta oportunidad nombraremos a cada objeto como **Point Reflection**.

Una vez incorporados los tres objetos mencionados, nuestro proyecto se presentará de la siguiente manera:



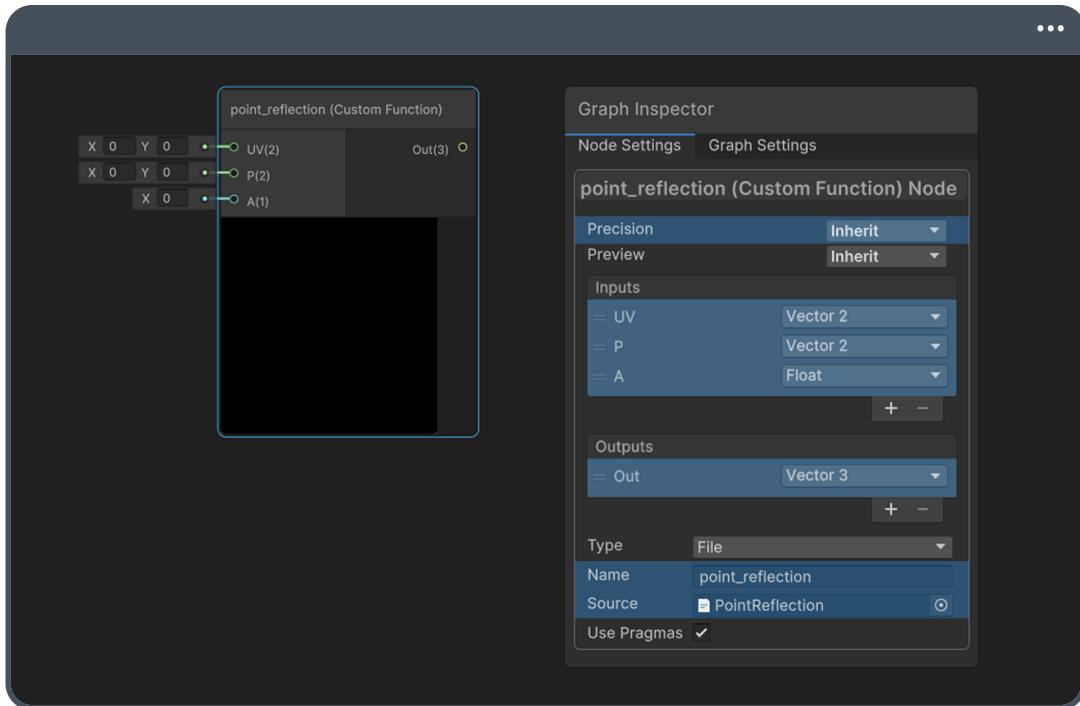
(2.4.a Estructura del proyecto)

Iniciando el proceso, primero asignaremos el shader al material y luego este material a un Quad en nuestra escena, lo cual nos permitirá visualizar los ajustes realizados en la configuración del shader.

Una vez abierto el shader, incluiremos un nodo **Custom Function** donde introduciremos las operaciones necesarias para visualizar la reflexión de un punto. Basándonos en la función ilustrada en la fórmula 2.3.e de la sección previa, el primer paso consistirá en incorporar las coordenadas UV, el punto p que reflejaremos, y el ángulo a que determina la orientación de la línea de reflexión.

Dado que tanto las coordenadas como el punto son vectores bidimensionales, por tener componentes x e y , mientras que el ángulo es un valor escalar, optaremos por utilizar un vector tridimensional para el valor de salida **Out**. Esta decisión se debe a que cada resultado visual se asociará con un color único, facilitando así la comprensión del concepto.

En cuanto a la propiedad **Name**, usaremos **point_reflection** para nombrar su función, y para **Source**, adjuntaremos el script **PointReflection**.



(2.4.b Nodo point_reflection)

Continuaremos definiendo el método **point_reflection_float()** en nuestro script HLSL, incorporando como argumentos las variables **uv**, **p** y **a**, además del valor de salida **Out**, como se muestra a continuación:

```

1 void point_reflection_float(in float2 uv, in float2 p, in float a, out
2   float3 Out)
3 {
4     Out = float3(0, 0, 0);
5 }
```

Para visualizar la reflexión de manera efectiva, es imprescindible considerar dos puntos: **p** (el punto original) y **p'** (el punto reflejado), así como el propio ángulo de reflexión. Antes de adentrarnos en el desarrollo del método **point_reflection_float()**, es vital incluir las funciones que permitan representar cada uno de estos elementos de forma clara.

Comenzaremos con la implementación de una función lineal en la forma $y = mx + b$, donde $m = \tan(a)$ para facilitar una rotación controlada. Dado que el valor de b será cero en este contexto, lo excluiremos de nuestra función, simplificando así nuestra ecuación.

```

1 #define PI 3.14159265358
2
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;      // -- mx + 0 --
10    fx -= f;
11
12    return (fx > 0.0) ? 1.0 : 0.0;
13 }
14
15 void point_reflection_float(in float2 uv, in float2 p, in float a, out
   float3 Out) { ... }
```

Del fragmento de código, comenzamos definiendo la constante **PI** en la primera línea, que representa la semicircunferencia de un círculo, es decir 180° . Esta constante será esencial para ajustar el signo de x en situaciones donde la inclinación de línea exceda los 90° . Avanzando en el código, específicamente en la línea 8, calculamos la pendiente **m** de nuestra línea como la tangente del ángulo **a**. En este contexto, **a** quien está expresada en radianes, indica la inclinación de la recta con respecto al eje **x** positivo, y se mide en sentido antihorario. Por lo tanto, si el ángulo es menor que 90° , entonces **tan(a)** resulta en un valor positivo; sin embargo, para ángulos entre 90° y 180° , **tan(a)** adopta un valor negativo. Este cambio se refleja visualmente como una transición en el color de blanco a negro.

Para profundizar en el entendimiento, procederemos a implementar el método **point_reflection_float()**, incorporando la función **linear_function()** dentro

de su definición. Adicionalmente, introduciremos una nueva función en nuestro script que facilitará el cálculo del ángulo en grados en lugar de radianes, mejorando así nuestra comprensión y manejo del concepto.

```

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
23     float3 Out)
24 {
25     uv -= 0.5;
26     a = to_degree(a);
27
28     const float3 color_g = float3(0, 1, 0);
29     float3 l = linear_function(uv, a) * color_g;
30
31     Out = l;
}

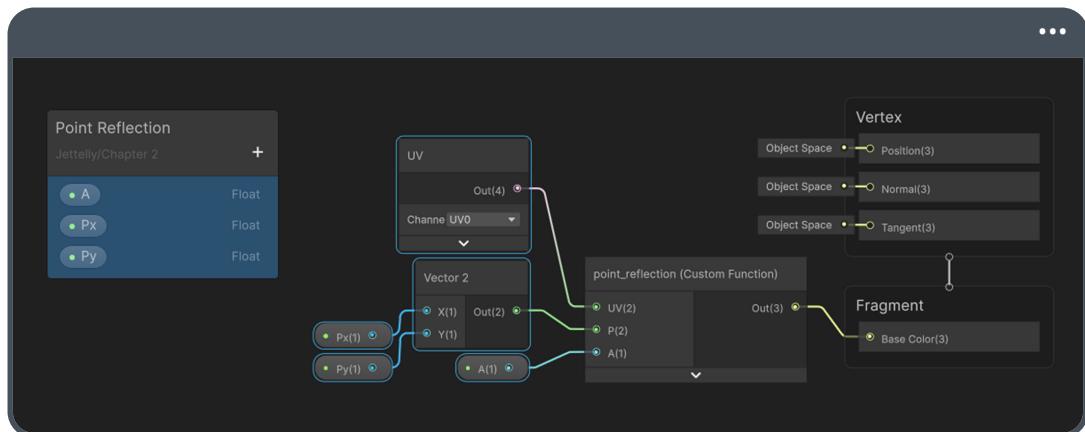
```

En la línea 17, introducimos una función denominada **to_degree()**, diseñada para convertir un ángulo de radianes a grados, utilizando la fórmula $d = r \frac{\pi}{180}$. Posteriormente, en el método **point_reflection_float()**; específicamente en la línea 24, ajustamos las coordenadas **uv** restando 0.5 de sus componentes **x** e **y**. Esta modificación es fundamental para centrar el punto de origen en el medio del Quad en nuestra escena. En la línea 27, definimos **color_g** como un vector tridimensional que representa el color verde en el sistema RGB. Finalmente, este color se utiliza para ponderar el resultado de la función lineal **linear_function()**, y el producto se almacena en **l**, un vector tridimensional.

Es importante asegurar que las propiedades estén correctamente vinculadas a nuestro nodo en el **Master Stack** para poder observar la rotación de la función lineal en el Quad. Es relevante mencionar que, para el punto **p**, optaremos por usar dos valores flotantes

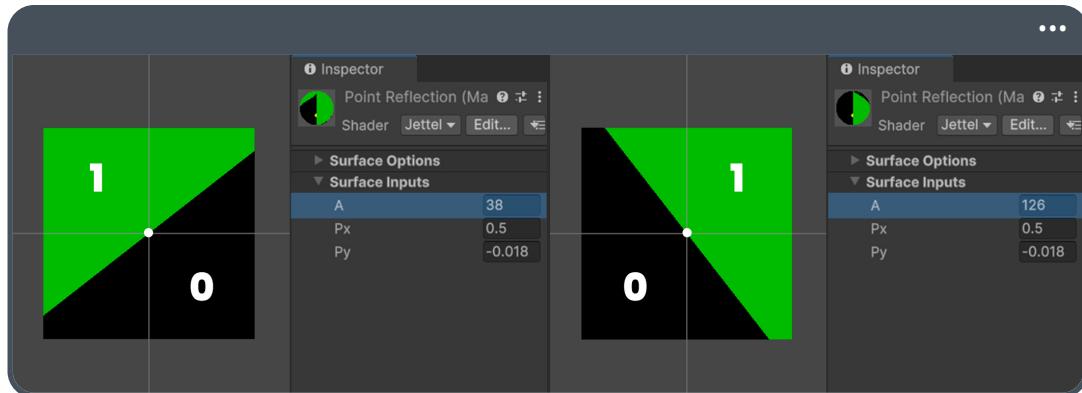
agrupados en un vector bidimensional debido a la limitación de no poder asignar rangos directamente a los componentes de un vector. En cambio, un valor flotante puede definirse con un mínimo, un máximo y un valor predeterminado. Por lo tanto, procederemos al **Blackboard** para añadir las siguientes propiedades.

- Agregamos la propiedad **A** y la configuramos como un rango, con un mínimo de 0 y un máximo de 180.
- Luego, incluimos el primer componente del punto **p**, es decir **Px**, también como un rango. Por consiguiente, su mínimo será -0.5 y su máximo 0.5.
- En cuanto al componente **Py**, replicamos el procedimiento anterior, estableciendo sus límites mínimos y máximos equivalentes a los de **Px**.



(2.4.c)

Como se ilustra en la imagen 2.4.c, se han vinculado las propiedades relevantes al nodo **point_reflection**. Al guardar los cambios y regresar a nuestra escena, tendremos la oportunidad de observar cómo se comporta la función lineal, notando específicamente que invierte su signo cuando el ángulo **A** supera los 90°.



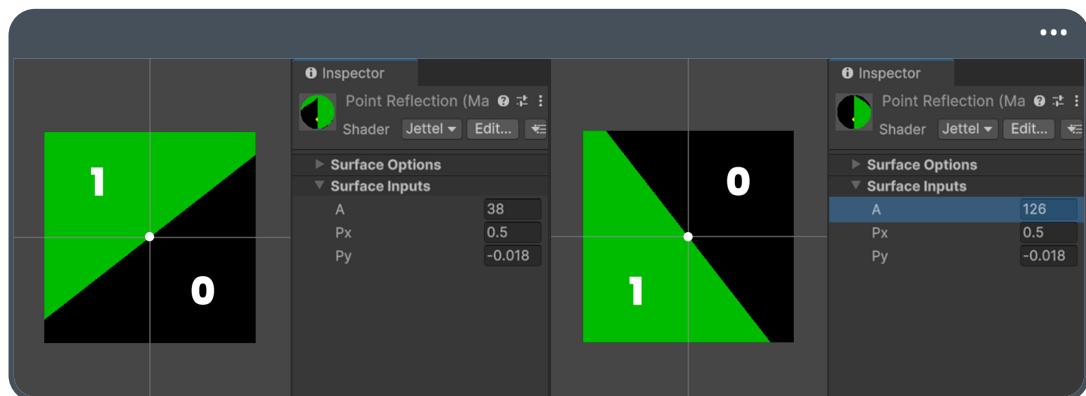
(2.4.d)

De la imagen 2.4.d, la referencia de la izquierda exhibe una pendiente positiva, coloreada en verde, para un ángulo **A** de 38° . Por otro lado, la imagen de la derecha muestra una pendiente negativa para un ángulo **A** de 126° . Este comportamiento puede presentar un desafío para la comprensión gráfica, especialmente cuando se busca reflejar un punto, ya que puede resultar confuso determinar la posición real del punto reflejado. Para abordar este problema, una solución es ajustar la función lineal para que adopte valores negativos cuando el ángulo sea mayor a 90° . Para implementar esta corrección, regresaremos a nuestro script y añadiremos el siguiente fragmento de código.

```

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 }
```

Al revisar la línea 12 del código notaremos que se ha definido una nueva variable **h** la cual adopta un valor de 1 o -1 dependiendo de si el ángulo **a** supera **PI / 2** en radianes (equivalente a 90°). Luego, esta variable **h** se multiplica por **fx**, asegurando que la visualización de la pendiente se mantenga positiva de manera constante. Guardando los cambios y regresando a nuestra escena, observaremos que la pendiente conserva su coherencia sin invertir su signo.



(2.4.e)

Ahora, enfocaremos nuestros esfuerzos en visualizar gráficamente el punto *p* y, posteriormente, su reflexión *p'*. Para lograr esta representación, recurriremos al uso del círculo algebraico que hemos definido anteriormente en el archivo **AlgebraicCircle**. Para incorporarlo a nuestro proyecto, emplearemos la directiva **#include**, la cual le indica al compilador que inserte el contenido del archivo en el código fuente en el lugar exacto donde se encuentra la directiva.

A continuación, detallaremos su ubicación empleando la ruta entre comillas. Siguiendo la estructura organizativa propuesta al comienzo de este libro, **AlgebraicCircle** quedaría establecida como:

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

```

1 #define PI 3.14159265358
2
3 #include "Assets/Jettelly Books/ ... /AlgebraicCircle.hlsl"
4
5 float linear_function(float2 uv, float a) { ... }

```

En el contexto del método `point_reflection_float()`, dada la naturaleza void de `algebraic_circle_half()`, será necesario declarar e inicializar previamente las variables que funcionarán como valores de salida en el método. Observemos la implementación a partir de la línea 29 en adelante del siguiente esquema:

```

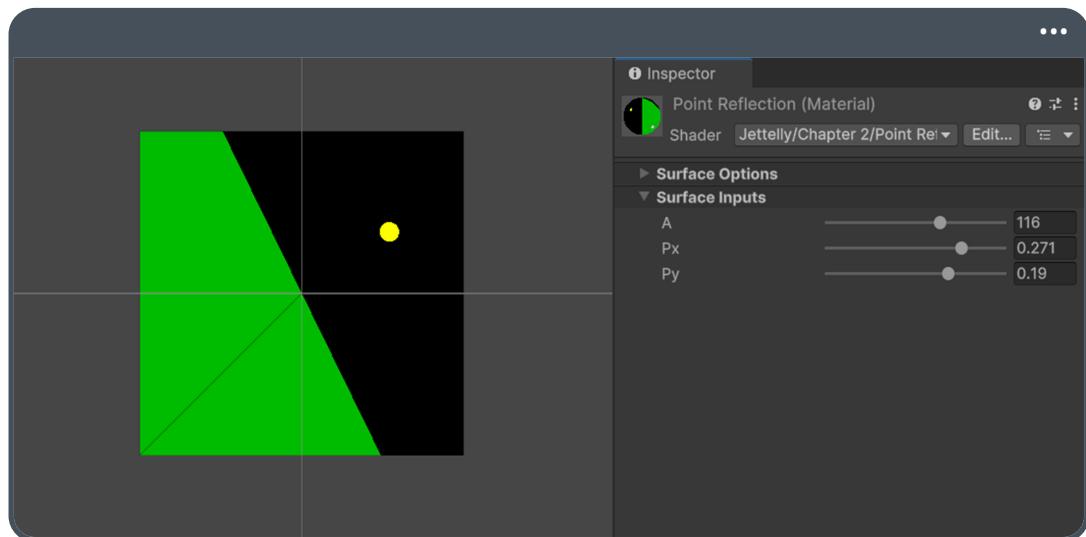
23 void point_reflection_float(in float2 uv, in float2 p, in float a, out
24     float3 Out)
25 {
26     uv -= 0.5;
27     a = to_degree(a);
28
29     const float3 color_g = float3(0, 1, 0);
30     const float3 color_y = float3(1, 1, 0);
31
32     float3 l = linear_function(uv, a) * color_g;
33
34  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

```

Enfocándonos en la línea 33, `p0` se ha introducido como una nueva variable destinada a recibir el resultado de `algebraic_circle_half()` (línea 34). Posteriormente, `c`, un vector tridimensional, se calcula invirtiendo el valor de `p0` mediante la operación `1 - float3(p0.xxx)`, donde `p0.xxx` implica replicar el valor de `p0` en cada

componente del vector. Esta inversión permite que, al combinarse con **color_y**, resulte en la visualización de un punto de color amarillo.

Al aplicar estos cambios en nuestro script y regresar a la escena, podremos observar la interacción entre la función lineal y el punto p , evidenciando así la efectividad de nuestra implementación en la visualización tanto de la línea como del punto original.



(2.4.f)

Para el ejercicio ilustrado en la imagen 2.4.f, se ajustó el valor de la propiedad **RADIUS** en 0.03 en el script **AlgebraicCircle** con el fin de obtener una representación gráfica de un círculo de mayor tamaño. El último paso para completar este proceso es agregar el punto de reflexión p' , que se deriva del punto original p y la normal n del ángulo a .

...

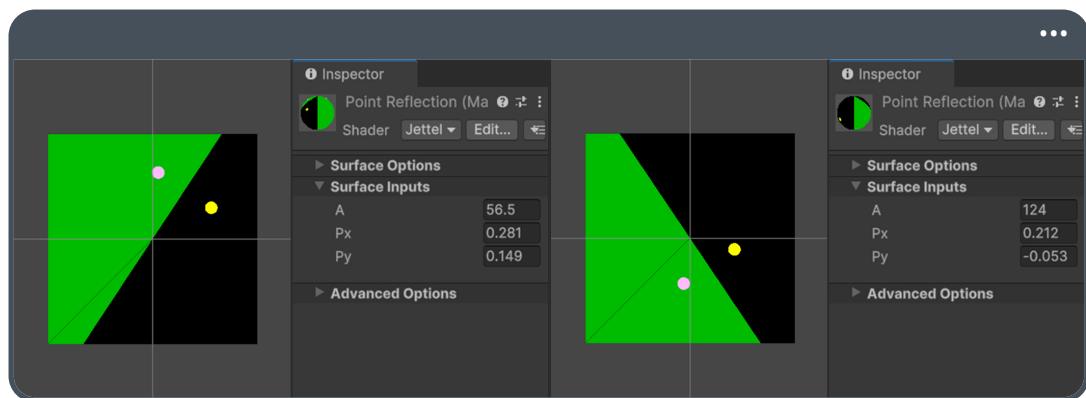
```

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

En la línea 34, se introduce **n** como la normal basada en la descripción de la ecuación 2.3.g. En la línea 35, calculamos el punto de reflexión **pr**, aplicando la fórmula vista en la referencia 2.3.e. Posteriormente, se implementa un segundo círculo algebraico (línea 41) el cual esencialmente representa al punto p' . Para este último, se utiliza **pr** como argumento en la función, y su resultado se guarda en la variable **p1** (línea 38).

En la línea 44, se define **cr**, un vector tridimensional que adopta el valor de **p1** en cada componente. Luego, **cr** se multiplica por el color magenta **color_m**, establecido en la línea 30, para diferenciarlo del punto anterior. Guardando los cambios y regresando a la

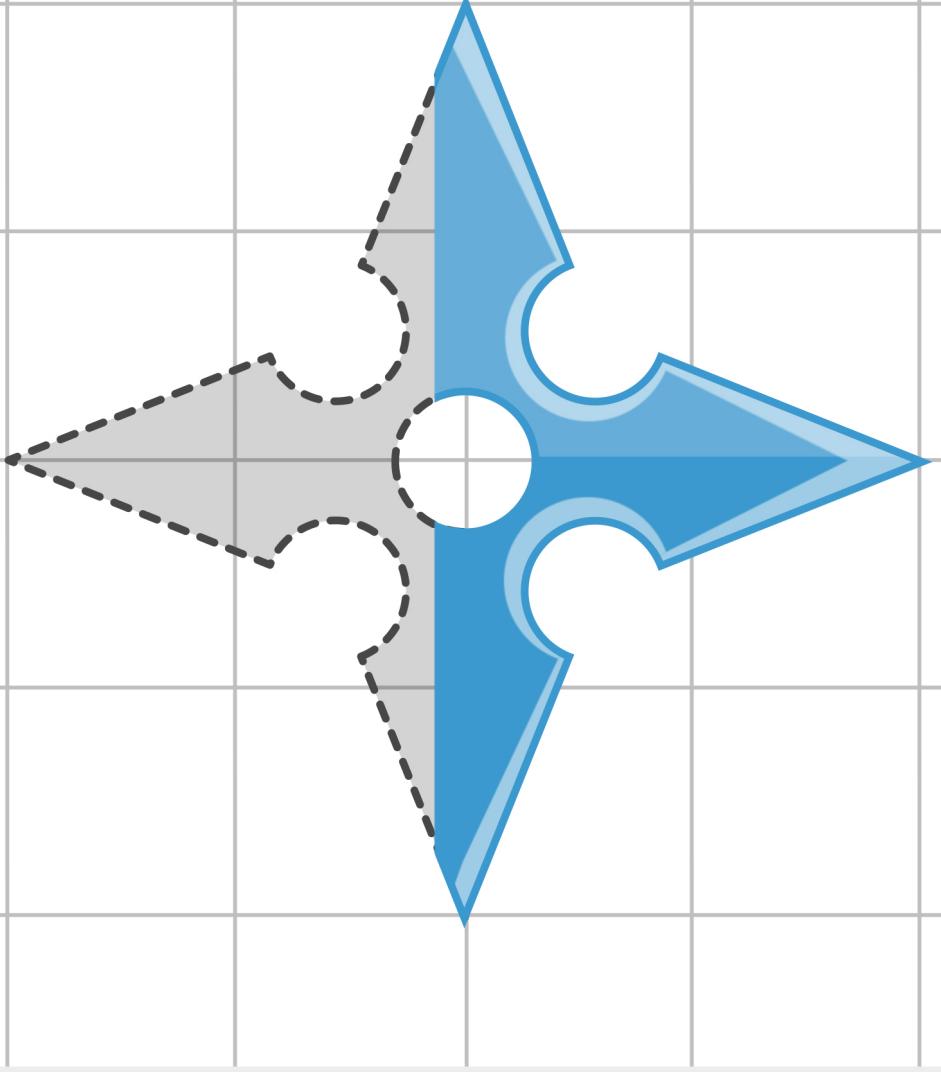
escena, lograremos visualizar tanto el punto original p como su reflejo p' en el Quad de nuestra escena.



(2.4.g)

Resumen de capítulo.

- Durante este capítulo, hemos explorado las funciones trigonométricas y su aplicación en el desarrollo de visualizaciones con HLSL en Unity. Iniciamos con una introducción a los conceptos fundamentales de las funciones trigonométricas, estableciendo su relación con el triángulo rectángulo. Se abordan las definiciones básicas del seno, coseno y tangente, justo con su interpretación geométrica.
- En la sección siguiente, se desarrolla un shader para visualizar las funciones trigonométricas. Se construye una onda sinusoidal y se implementa la función tangente para representar la orientación de un punto que viaja sobre la onda en un espacio bidimensional.
- Posteriormente, se explican las operaciones matemáticas para reflejar un punto respecto a una línea, definida por la función lineal. Se detallan los cálculos paso a paso para comprender el proceso de reflexión en un plano cartesiano. Finalmente, se desarrolla un shader que permite visualizar la reflexión de un punto en tiempo real sobre un Quad en la escena.



Capítulo 3
Figuras procedurales.

En este capítulo, te iniciarás en la creación de figuras procedurales bidimensionales en Shader Graph utilizando las funciones matemáticas que exploraste anteriormente. Empezarás con un análisis detallado de estas figuras en el plano cartesiano, identificando y abordando los retos asociados con la manipulación de coordenadas. Durante este proceso, implementarás estrategias específicas que te ayudarán, por ejemplo, a refinar la resolución de tus creaciones, suavizando los bordes y mejorando la calidad visual de los mismos. Este enfoque práctico te permitirá aplicar conceptos matemáticos fundamentales en el desarrollo de figuras procedurales en Unity, abriendo un abanico de oportunidades creativas y técnicas para la generación de gráficos.

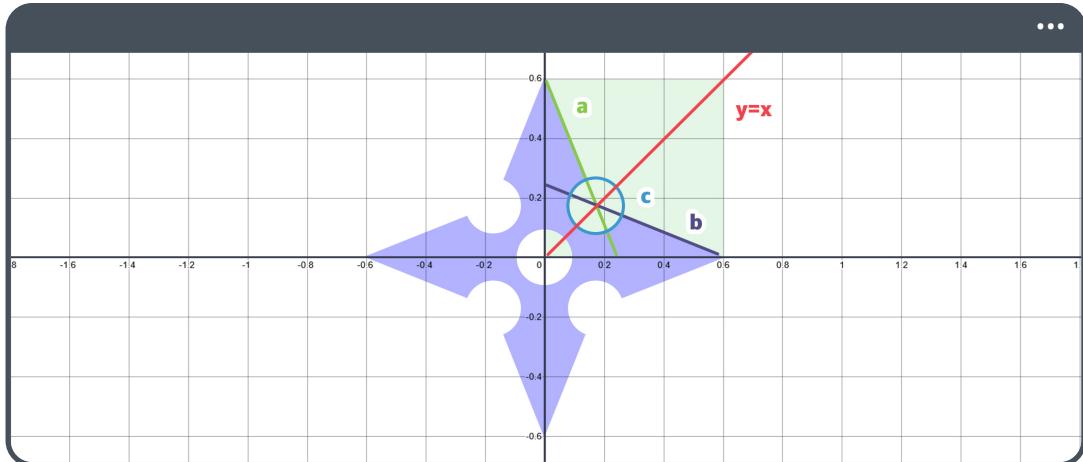
3.1 Analizando la forma de un Shuriken.

Cuando hablamos de imágenes procedurales, es importante conocer los requerimientos de la figura que deseamos crear. Dependiendo de esta, se adoptan funciones específicas para trazar curvas o líneas que, en su conjunto, se traducen en áreas gráficas en la pantalla. Antes de comenzar con nuestro análisis, es útil considerar algunas preguntas clave que pueden acelerar el proceso de creación y desarrollo:

Para una figura procedural cualquiera:

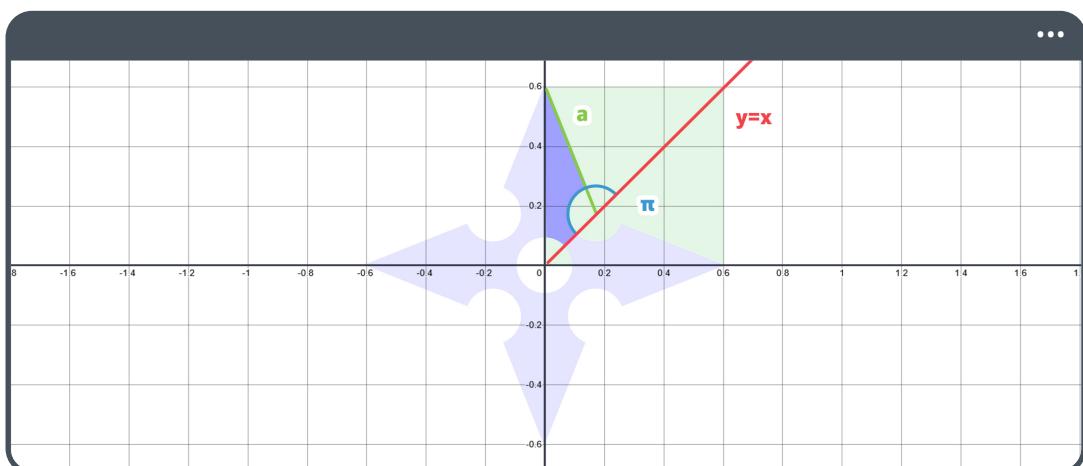
- ¿Se puede recrear utilizando polígonos?
- ¿Son sus lados iguales?
- ¿Tiene ángulos semejantes?

Aunque podríamos incluir otras preguntas relevantes, estas son las más importantes porque su análisis nos permite optimizar no solo la figura en sí, sino también desarrollar funciones matemáticas simplificadas, lo que minimiza el cálculo en la GPU de manera directa. Por ejemplo, tomemos la forma de un Shuriken y proyectemos sus coordenadas sobre un plano cartesiano.



(3.1.a Visualización de un Shuriken en Desmos)

Como podemos observar en la imagen 3.1.a, el Shuriken, centrado en el origen, tiene una forma simétrica. Por ende, puede ser optimizado utilizando solo tres puntos a , b y c , y un semicírculo para recrear su forma completa. De hecho, el punto b podría ser igual al reflejo de a , es decir, $Ref_l(b) = a$, dado que ambas rectas conectan con c . Esto se puede ver en la siguiente Imagen:



(3.1.b Patrón de repetición)

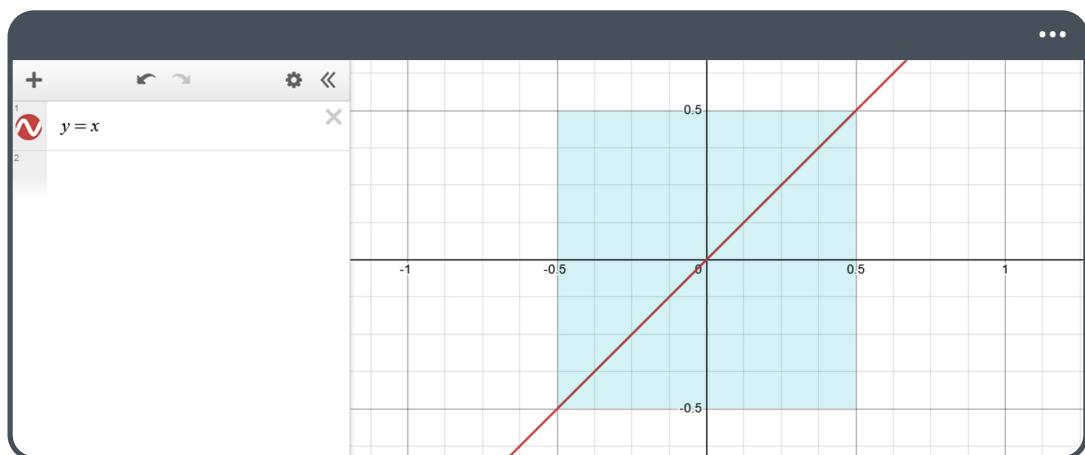
Es cierto que no estamos considerando el círculo ubicado en el punto central, pero su analogía es la misma. Si quisiéramos dibujar la primera punta de un Shuriken en un plano cartesiano, deberíamos considerar exclusivamente:

- Un triángulo.
- Un semicírculo, para el corte.
- Un octavo de círculo, para el agarre del centro.

Luego, sería necesario aplicar valores absolutos y otras funciones para llevar el cálculo a los otros cuadrantes, además de limitar la longitud de las rectas infinitas producidas por estas funciones. Sin embargo, toda la operación estaría centrada en la simetría inicial del primer cuadrante.

Para entender el proceso, aplicaremos los conceptos mencionados hasta este punto y dibujaremos la forma de un Shuriken de cuatro puntas sobre el plano utilizando la interfaz de Desmos. Antes de comenzar, definiremos ciertos valores que harán de nuestra imagen procedural un objeto dibujable en coordenadas UV posteriormente. Por ejemplo, el diámetro máximo de la forma será igual a 1.0, por lo tanto, los puntos y sus posiciones se mantendrán dentro de un área entre [-0.5 : 0.5], considerando el origen como punto central.

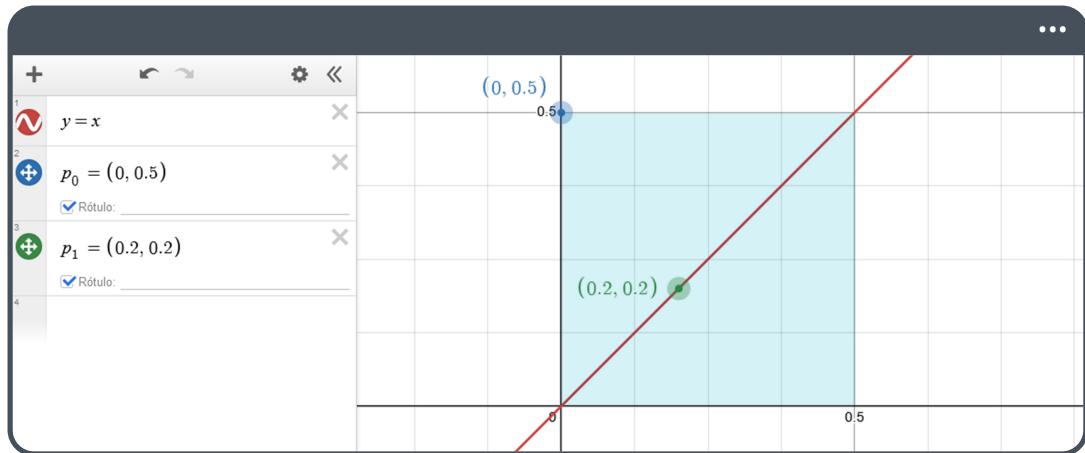
Para visualizarlo, comenzemos generando una recta del tipo $y = x$ de la siguiente manera:



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

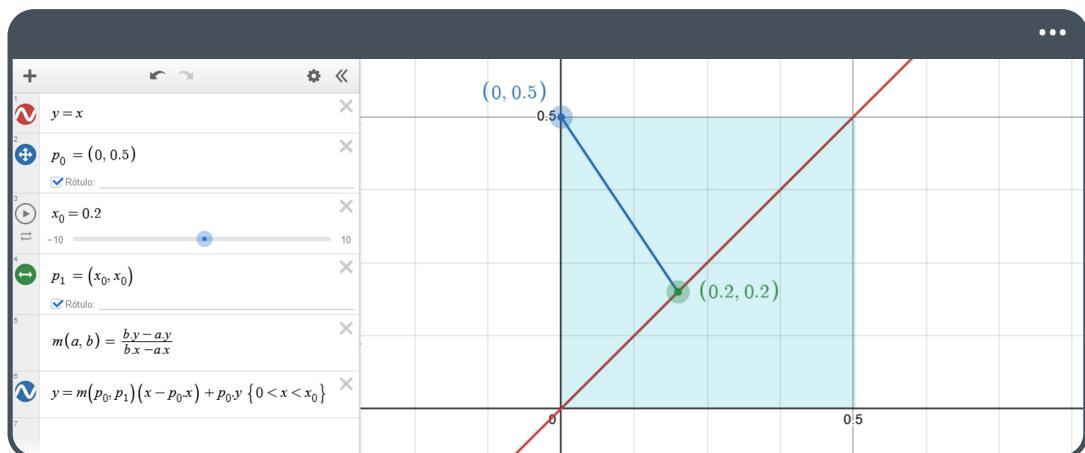
Considerando que modificaremos la forma del Shuriken de manera dinámica desde el Inspector de Unity, esta recta nos permitirá visualizar su apertura, generando variaciones

según la posición de un punto por definir. Posteriormente, será necesario definir un punto adicional para trazar una recta que determine la punta de la misma.



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

Si prestamos atención a la imagen anterior, notaremos que se han definido dos puntos: p_0 y p_1 , ubicados en las posiciones [0.0, 0.5] y [0.2, 0.2] respectivamente. Estos puntos existen solo a nivel de datos. Para generar gráficos con ellos, será esencial trazar una recta que los une, es decir $\overline{p_0 p_1}$. Para ello, podemos utilizar la ecuación de la recta mencionada en la sección 1.4 del capítulo 1, que se define como $y = (x - a.x) + a.y$, donde la pendiente $m = \frac{b.y - a.y}{b.x - a.x}$. Este proceso se ilustra en el siguiente ejemplo:



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

Hay dos aspectos importantes en la imagen 3.1.e. Primero, la variable x_0 . Dado que esta se ha definido para cada componente en el punto p_0 , si modificamos su valor, el punto se moverá siguiendo la recta $y = x$. Considerando que previamente hemos limitado el diámetro del Shuriken, x_0 también se ha restringido a un valor entre $[0.0 : 0.5]$. Este proceso es fundamental, ya que, de otra manera, la forma de la figura se verá afectada cuando los valores estén fuera del rango mencionado.

Segundo, los límites de la recta. La recta generada entre el punto p_0 y p_1 es infinita. En consecuencia, la operación $\{0 < x < x_0\}$ se ha empleado para limitar la recta al espacio que necesitamos.

El siguiente paso consistiría en agregar el punto p_2 , que conecta con p_1 en el plano cartesiano. Dado que $Ref_l(p_2) = p_0$, podemos aplicar la ecuación 2.3.e del capítulo 2. Sin embargo, considerando que $y = x$ no presenta cambios en su ángulo, podemos simplificar la fórmula de reflexión de la siguiente manera:

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

(3.1.f)

Donde,

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

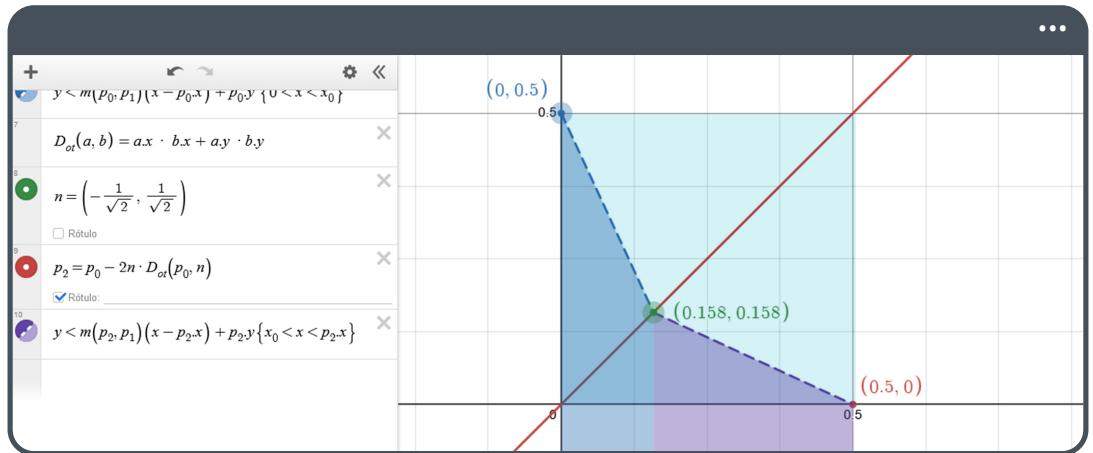
(3.1.g)

De la ecuación 3.1.g, podemos deducir que la variable n se refiere a la normal de la recta, y $p \cdot n$ corresponde al producto punto, es decir,

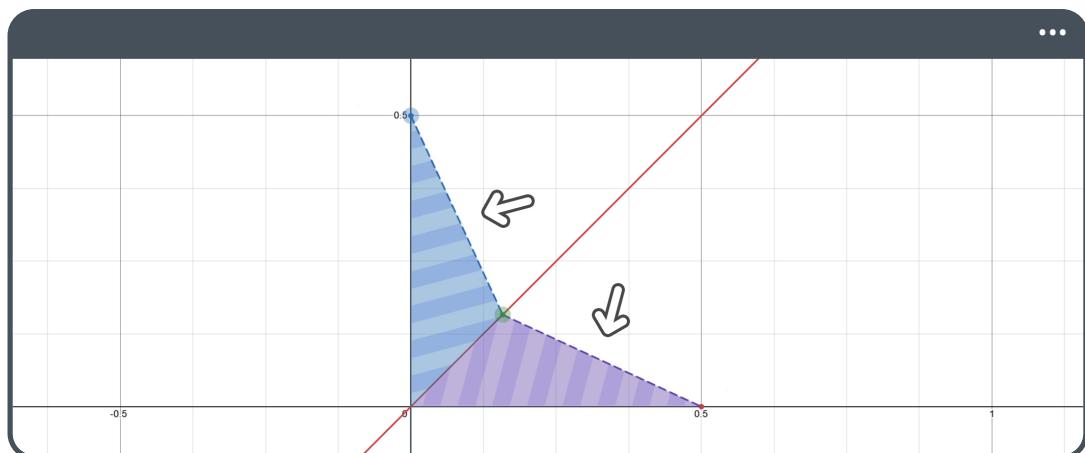
$$p \cdot n = p.x * n.x + p.y * n.y$$

(3.1.h)

Si trazamos una recta desde el punto p_2 hacia el punto p_1 utilizando las funciones mencionadas previamente, obtendremos el siguiente resultado:

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

Un factor por considerar cuando trabajamos con segmentos de recta es la región de color y dirección que se forma. Si prestamos atención a la imagen 3.1.i, veremos que se han generado dos regiones de color: una celeste y otra purpura, ambas apuntando en dirección vertical. Podría suponerse que cada región debería apuntar según el ángulo que se encuentra, como se muestra en la siguiente imagen:



(3.1.j Reflexión de un punto)

Sin embargo, este no es el caso, ya que las regiones presentadas en la imagen 3.1.i indican que todos los puntos dentro de sus respectivas áreas cumplen con la condición definida para cada caso, es decir:

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

(3.1.k)

Para obtener un resultado óptimo, sería necesario emplear una técnica distinta, conocida como **Signed Distance Function (SDF)**, la cual permite determinar la distancia de un punto respecto a una recta. No obstante, continuaremos experimentando con las funciones lineales y trigonométricas empleadas hasta este punto. Más adelante en este libro, dedicaremos dos capítulos a las funciones de distancia implícita tanto en dos como en tres dimensiones.

Como podemos observar, la punta de nuestra figura procedural ya ha sido definida. Solo falta aplicar las funciones en los otros cuadrantes para generar su forma completa. Para ello, podemos emplear el valor absoluto sobre las coordenadas xy en las funciones que definen las regiones de color, es decir:

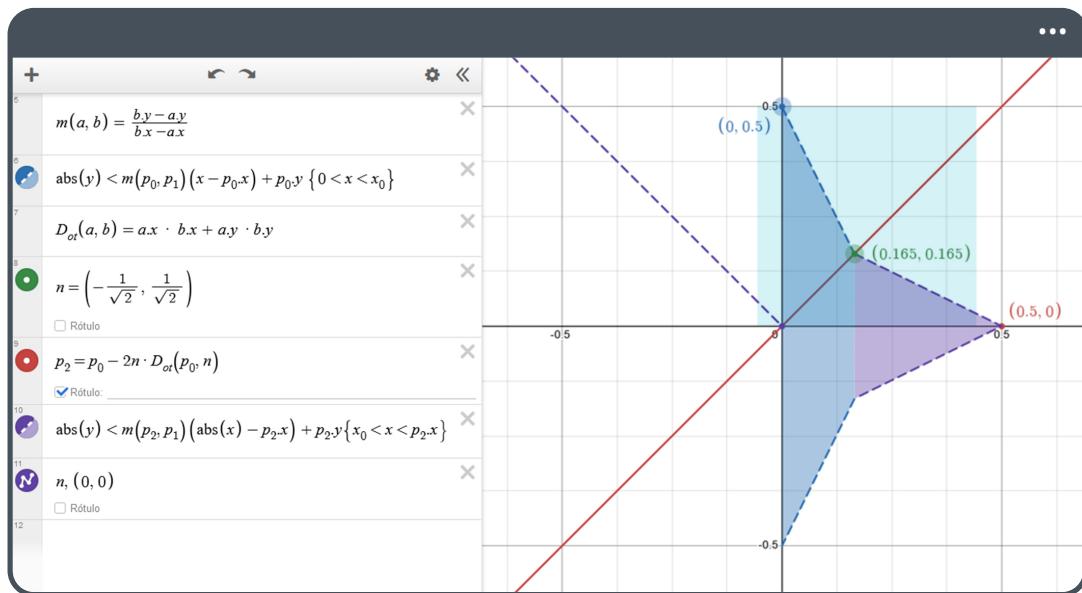
$$|y| < m(|x| - a.x) + a.y$$

(3.1.l)

El valor absoluto corresponde a la distancia de un número cualquiera, hasta el cero en la recta numérica, sin considerar la dirección. Este factor hace que tal número sea siempre positivo. Tomemos por ejemplo un número negativo; el -5. Dado que entre el 0 y este último en la recta numérica hay 5 espacios, el valor absoluto de -5 es igual a 5.

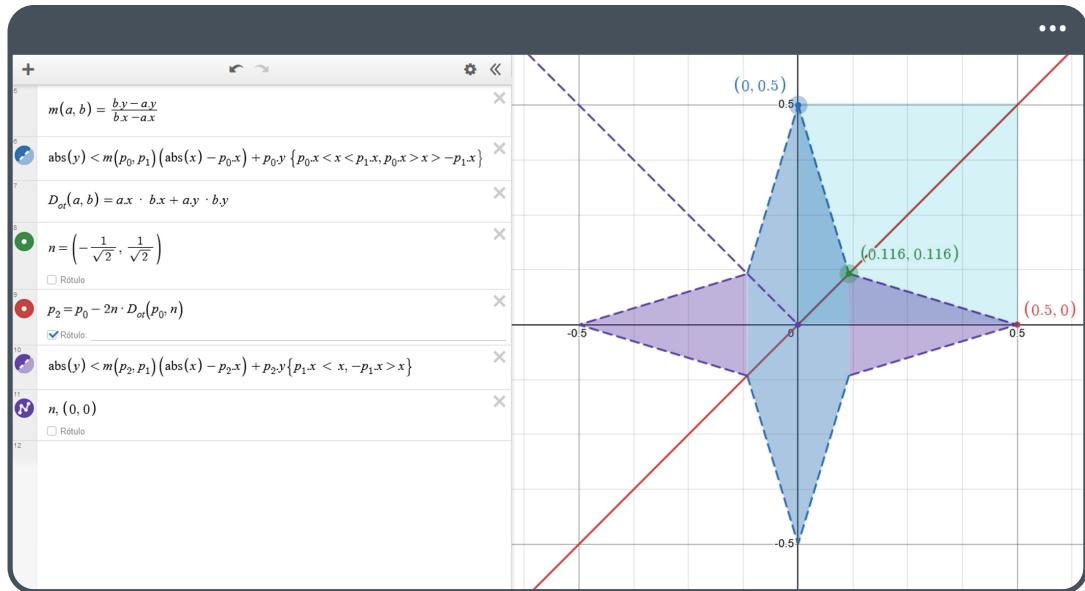
Ahora, cabe destacar que esta operación se podría ver afectada debido a los límites que hemos definido para la coordenada x en ambas regiones. El primer segmento ha sido limitado mediante $\{0 < x < x_0\}$, mientras que el segundo se limita mediante $\{x_0 < x < p_{2x}\}$. Por lo tanto, será necesario ajustar estos límites antes de aplicar el valor absoluto a cada coordenada.

Comenzaremos aplicando en valor absoluto sobre la coordenada y de la siguiente manera:



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

Como podemos observar en la imagen anterior, ambos segmentos $\overline{p_0 p_1}$ y $\overline{p_2 p_1}$ han sido reflejados con respecto al eje x debido a la utilización del valor absoluto sobre el eje y , el cual se expresa como $abs(y)$ en el ejemplo. En computación gráfica, esto representa una optimización, ya que evitamos calcular los demás puntos de la figura. Para concluir, solo faltaría aplicar el valor absoluto sobre la coordenada x y actualizar los límites para completar nuestro Shuriken.



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

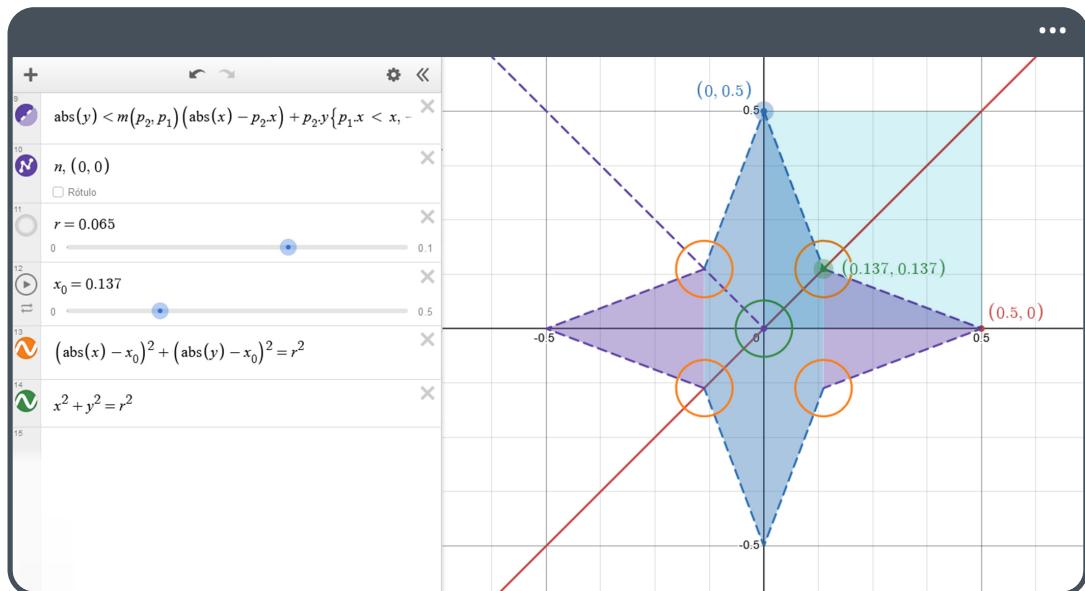
En la imagen 3.1.n, podemos observar que el primer segmento $\overline{p_0 p_1}$ ha sido limitado en dos ocasiones: cuando p_{0x} es menor a x , y x es menor a p_{1x} , es decir $\{p_{0x} < x < p_{1x}\}$ y, además, cuando $\{p_{0x} > x > p_{1x}\}$. De la misma manera, el segundo segmento $\overline{p_2 p_1}$ ha sido limitado en dos ocasiones: primero cuando $\{p_{1x} < x\}$, y luego cuando $\{-p_{1x} > x\}$. En este contexto, estos límites aparecen separados por comas, pero el resultado es el mismo.

Hasta este punto, la forma de nuestro Shuriken está completa; solo falta agregar círculos que nos permitan diferenciar aún más el cuerpo de la figura. Como hemos visto a lo largo del libro, podemos recrear círculos utilizando la fórmula $x^2 + y^2 = r^2$. No obstante, será necesario aplicar el valor absoluto sobre ambas coordenadas xy para obtener reflexiones en los distintos cuadrantes. Por lo tanto, procederemos a extender la función, considerando que:

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

(3.1.ñ)

De la ecuación anterior, observamos que se ha incluido una nueva variable u , la cual tiene la función de desplazar los valores absolutos de las coordenadas xy antes de calcular su distancia al centro del círculo. Si llevamos esto al plano cartesiano, obtendremos el siguiente resultado:



(3.1.o <https://www.desmos.com/calculator/bma1c49gww>)

Como podemos observar, se ha definido un círculo para cada cuadrante donde $u = x_0$. De esta manera, la posición de los círculos será igual a la posición del punto p_1 .

Dado que ya tenemos las ecuaciones necesarias para implementar el Shuriken en HLSL, podríamos terminar nuestro ejercicio en este punto. Sin embargo, realizaremos una tarea adicional que nos ayudará a entender de mejor manera los límites cuando trabajamos con segmentos.

Las áreas geométricas no tan solo pueden ser definidas como formas, sino también como límites. Para ello, simplemente debemos reemplazar el símbolo de igualdad por uno de desigualdad, es decir:

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

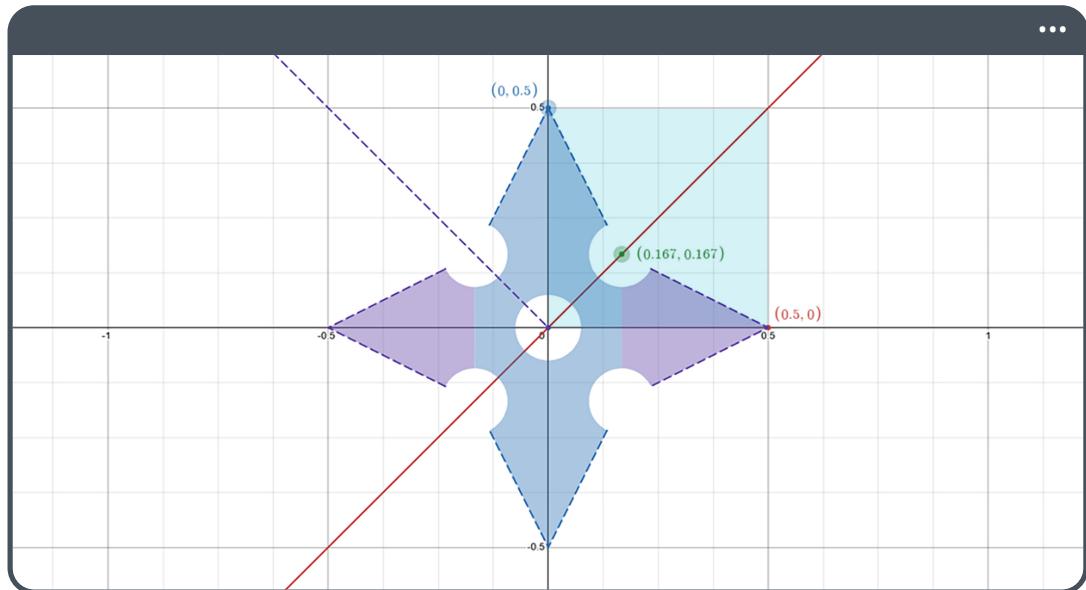
(3.1.p)

En Desmos, por ejemplo, podríamos definir la ecuación anterior como límite en ambos segmentos $\overline{p_0 p_1}$ y $\overline{p_2 p_1}$, incluyendo la función entre corchetes, como se muestra a continuación.

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

(3.1.q)

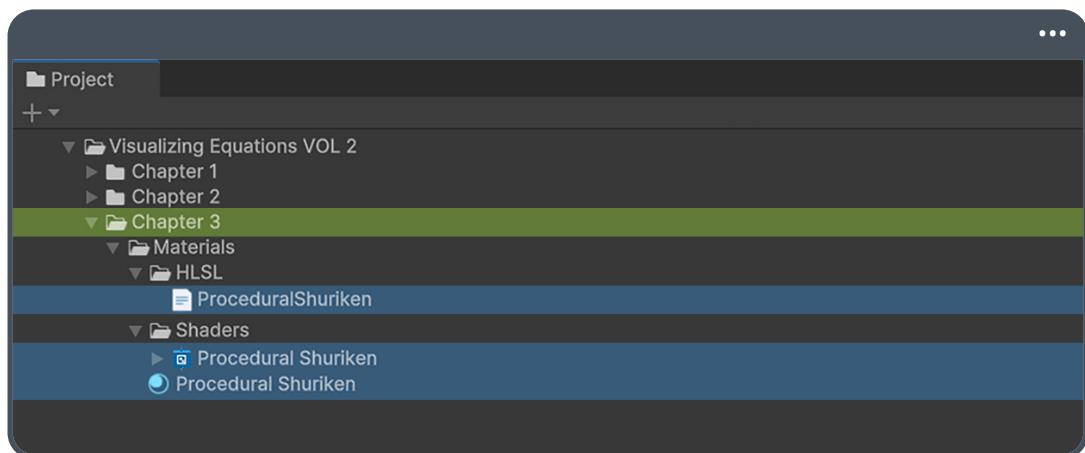
Si además incluimos el círculo central como límite, nuestro Shuriken se visualizará de la siguiente manera:

(3.1.r <https://www.desmos.com/calculator/8cc82ghuwo>)

3.2 Dibujando un Shuriken en HLSL.

En esta sección, revisaremos las funciones mencionadas anteriormente y las implementaremos en HLSL. Para ello, nuevamente haremos uso de un nodo **Custom Function** en Shader Graph, por ende, comenzaremos creando un nuevo shader de tipo **Unlit Shader Graph** en nuestra carpeta de proyecto, al cual nombraremos **Procedural Shuriken**. Luego, crearemos tanto un material como un script **.hls** utilizando el mismo nombre.

Siguiendo la estructura introducida en el primer capítulo, nuestro proyecto debería presentarse de la siguiente manera:

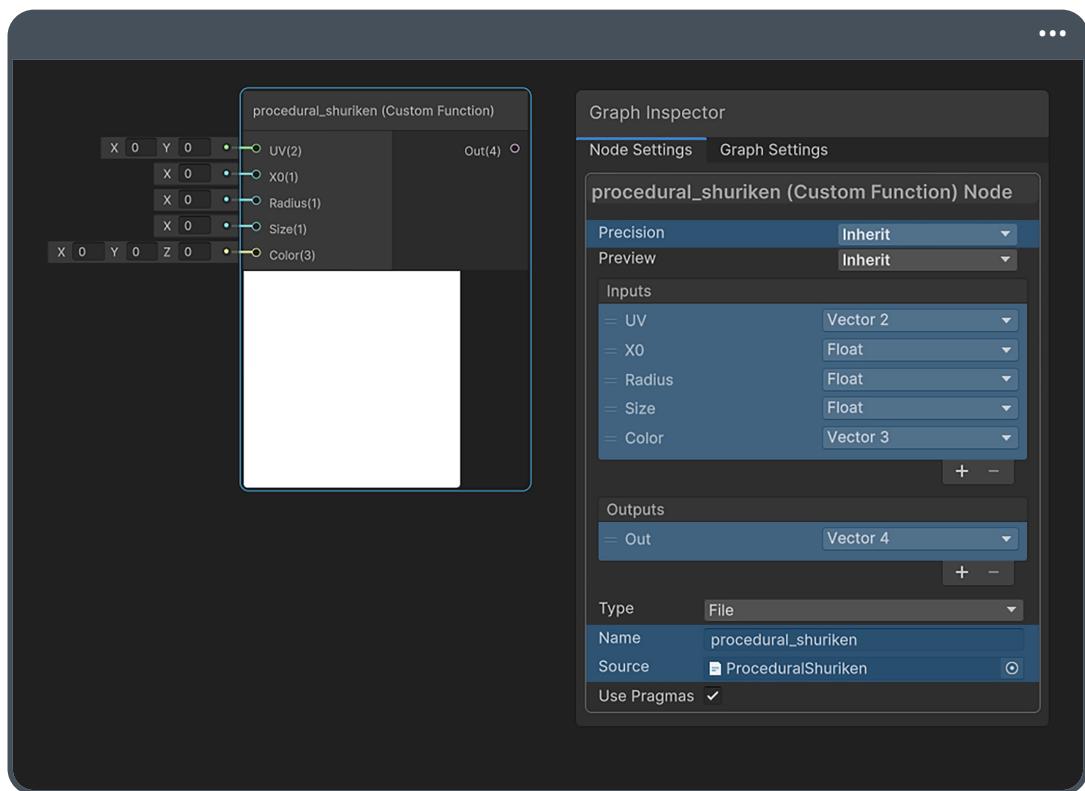


(3.2.a Estructura del proyecto)

Cabe destacar que, de la misma manera que hemos hecho en capítulos anteriores, será necesario asignar el material al shader y luego aplicar este material a un Quad en nuestra escena para visualizar los cambios que realizaremos en el shader como tal.

Una vez en el **Master Stack**, crearemos un nodo **Custom Function** al cual nombraremos **procedural_shuriken**. Nos aseguraremos de incluir el script **ProceduralShuriken** como **Source** en el nodo, manteniendo su configuración por defecto.

Como vimos en la sección anterior, las variables necesarias para crear nuestra figura procedural corresponden a x_0 y r . La primera modifica tanto la posición de p_0 como la del círculo y sus reflexiones, mientras que la segunda se refiere al radio de los círculos. También necesitaremos las coordenadas UV para proyectar la forma. Por lo tanto, agregaremos estas variables como entradas al nodo, incluyendo un valor flotante para modificar el tamaño total del Shuriken en el Quad, y un vector de tres dimensiones, para el color.



(3.2.b Nodo procedural_shuriken)

Generalmente, se utilizan vectores de cuatro dimensiones para las formas creadas de manera procedural debido al canal alpha A, lo cual significa que nuestro shader contendrá una canal de transparencia. Por lo tanto, nos aseguraremos de modificar el valor de la propiedad **Surface Type** de nuestro shader, yendo a la ventana **Graph Settings** y configurándola como **Transparent**.

Luego, comenzaremos con la definición de la función que nos permitirá dibujar nuestro Shuriken. Para ello, abriremos el script **ProceduralShuriken** e incluiremos las siguientes líneas de código:

```
1 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
2     in float size, in float3 color, out float4 Out)
3 {
4     Out = 0;
```

Si prestamos atención a los argumentos de la función, observaremos que se han incluido las entradas definidas previamente en el nodo **Custom Function**. Asimismo, tanto **x0** como **radius** corresponden a las variables introducidas para la función del Shuriken en Desmos. De manera preliminar, el **Output** de la función se ha establecido en 0.0 únicamente para evitar errores de compilación de la GPU, evitando que nos aparezcan artefactos visuales mientras desarrollamos nuestra figura.

En este momento, podríamos preguntarnos: ¿cuál es la primera tarea que debemos llevar a cabo? Podríamos comenzar definiendo los puntos p_0 y p_1 . Sin embargo, las coordenadas cartesianas deben ser definidas antes que cualquier ecuación. Por lo tanto, comenzaremos declarando un nuevo método al cual nombraremos **shuriken()**, el cual incluirá las coordenadas **uv**, la variable **x0**, y el **radius** como argumentos.

```

1 float shuriken(float2 uv, float x0, float radius)
2 {
3     float y = abs(uv.y);
4     float x = abs(uv.x);
5
6     return 0;
7 }
8
9 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
10    in float size, in float3 color, out float4 Out) { ... }
```

Siguiendo la explicación de la sección anterior, hemos declarado e inicializado dos nuevas variables, **x** e **y**, que corresponden al valor absoluto de las coordenadas UV respectivamente. Ahora, podemos definir los puntos mencionados anteriormente, considerando que su traducción a lenguaje HLSL se realiza de la siguiente manera:

```

1 float shuriken(float2 uv, float x0, float radius)
2 {
3     float y = abs(uv.y);
4     float x = abs(uv.x);
5
6     float2 p0 = float2(0.0, 0.5);
7     float2 p1 = float2(x0, x0);
8
9     return 0;
10 }
```

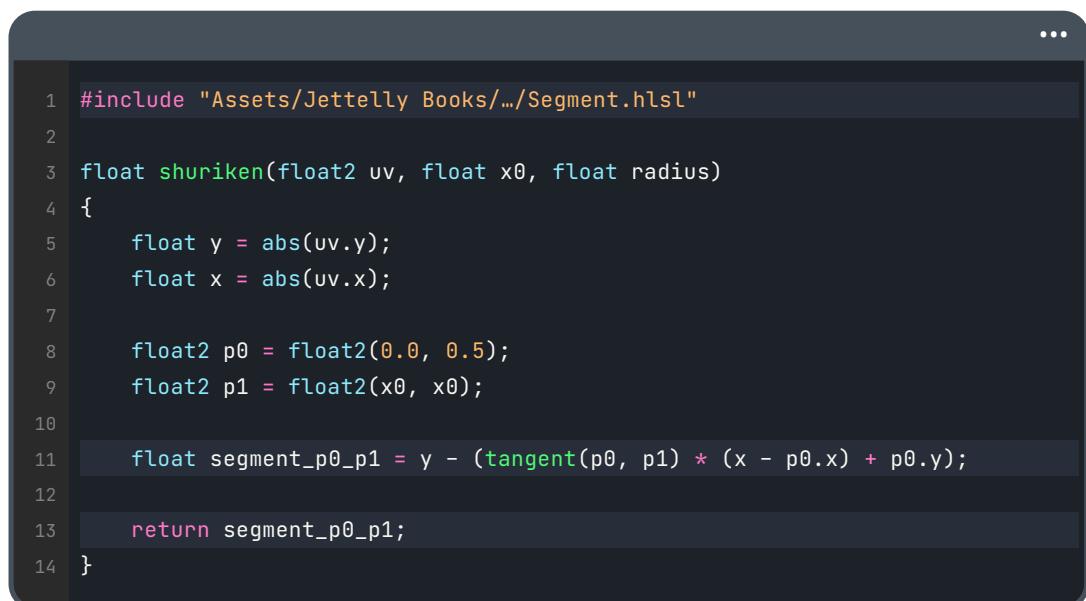
Si prestamos atención al ejemplo anterior, observaremos que se han declarado e inicializado los puntos respectivos. Cabe destacar que el punto **p0** podría ser declarado como constante, ya que su posición en el tiempo no cambiará.

Continuaremos aplicando la ecuación de la recta para visualizar los distintos segmentos que componen nuestra figura procedural. Sin embargo, la función de la pendiente,

tangent(), ya fue empleada anteriormente en la sección 1.4 del capítulo 1. Por lo tanto, no será necesario declararla e inicializarla nuevamente en nuestro script actual. Simplemente agregaremos el script **Segment.hlsl** como **#include** en nuestro código, siguiendo la ruta donde se encuentra el archivo:

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

De esta manera, nuestro código queda como se muestra a continuación:



```

1 #include "Assets/Jettelly Books/.../Segment.hlsl"
2
3 float shuriken(float2 uv, float x0, float radius)
4 {
5     float y = abs(uv.y);
6     float x = abs(uv.x);
7
8     float2 p0 = float2(0.0, 0.5);
9     float2 p1 = float2(x0, x0);
10
11    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
12
13    return segment_p0_p1;
14 }

```

Como ya sabemos, la directiva **#include** se utiliza para incluir contenido desde un archivo a otro, lo cual es bastante útil al momento de organizar nuestro código y hacerlo modular.

Si prestamos atención a la línea de código número 13, observaremos que la función retorna el primer segmento del Shuriken. Sin embargo, la misma contiene algunas complejidades matemáticas que podrían afectar el resultado y, por consecuencia, presentar artefactos visuales. Primero, la pendiente o gradiente **tangent()** contiene una división en su definición, por lo tanto, si $x_0 = 0$, la pendiente se vuelve infinita. Segundo,

las coordenadas UV no han sido normalizadas, lo que significa que el valor de salida puede exceder el rango entre [0.0 : 1.0].

Esto podemos solucionarlo fácilmente empleando la función **saturate()**, que restringe el valor de sus argumentos al rango que necesitamos. No obstante, llevaremos a cabo el siguiente ejercicio para visualizar el problema. Para ello, será necesario realizar las siguientes operaciones: incluir en el **Blackboard** las variables definidas como entrada en nodo, es decir, **UV**, **X0**, **Radius**, **Size** y **Color**, y, además, declarar un nuevo vector de cuatro dimensiones RGBA dentro del campo del método **procedural_shuriken_float()** para pasar el cálculo del shuriken como output.

Iniciaremos declarando e inicializando un nuevo vector de cuatro dimensiones dentro del método **procedural_shuriken_float()**, como se muestra en el siguiente ejemplo:

```

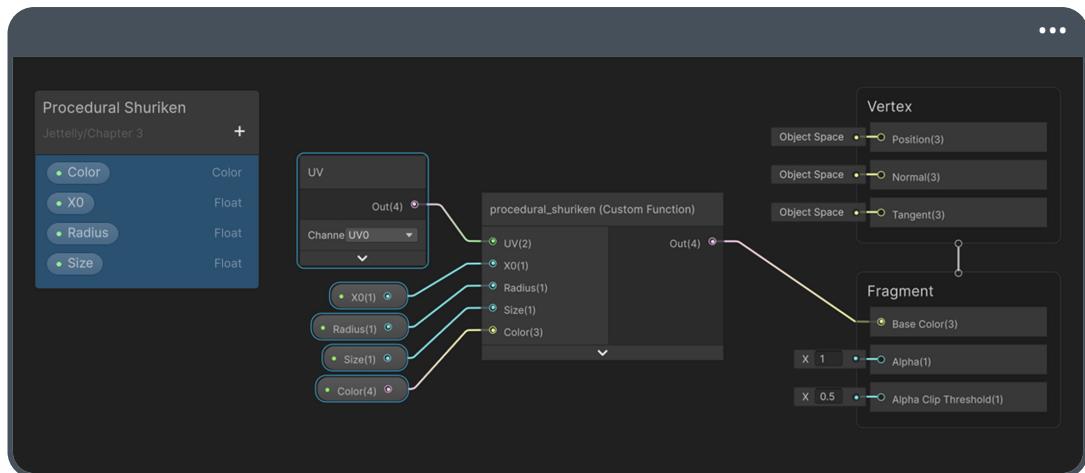
17 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,
18   in float size, in float3 color, out float4 Out)
19 {
20     uv -= 0.5;
21
22     float4 shape = shuriken(uv, x0, radius);
23     shape.rgb *= sin(float3(0.123, 0.453, 0.432) * 10); // color
24
25     Out = shape;
}

```

Una de las operaciones más importantes ocurre en la línea de código 19, donde se resta 0.5 a las coordenadas UV. Esto se lleva a cabo para centrar nuestra imagen procedural sobre el Quad. Posteriormente, en la línea de código número 21, el vector **shape** se inicializa utilizando el resultado de la función **shuriken()**. Como resultado, sus cuatro componentes RGBA poseen en mismo valor, lo cual facilita la implementación de color sobre los canales RGB y de transparencia sobre el canal A. Finalmente, en la línea 22, los canales RGB del vector **shape** son multiplicados temporalmente por la función **sin()**,

que a su vez es multiplicada por 10, resultando en **sin(1.23, 4.53, 4.32)**, lo cual produce un efecto cromático en el render final.

Únicamente falta declarar las propiedades en el **Blackboard** y conectarlas con nuestro nodo **procedural_shuriken**, como se presenta en la siguiente imagen.

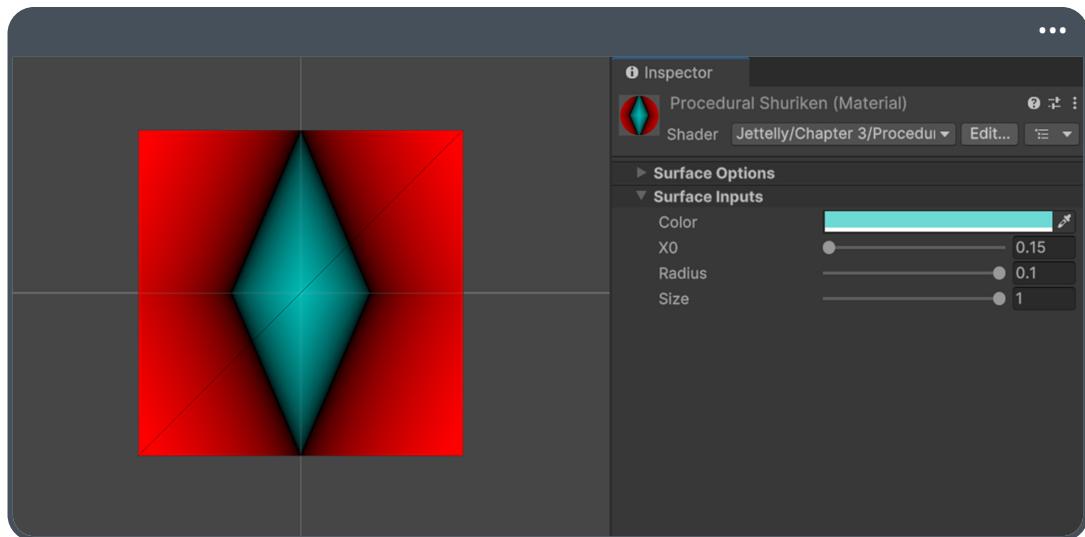


(3.2.c Propiedades del nodo procedural_shuriken)

Si prestamos atención, observaremos que solamente se ha conectado el **Out** de nuestra función personalizada como **Base Color** en el shader. Será necesario también conectar el canal alpha si deseamos ver transparencias. Sin embargo, dejaremos este proceso para última instancia, ya que primero nos enfocaremos en definir y ajustar correctamente los parámetros y funciones que determinan el aspecto visual del Shuriken.

Cabe destacar que algunos valores de las propiedades se han limitado para obtener un resultado que responda a la forma del Shuriken. Por ejemplo, **X0** posee un rango entre [0.15 : 0.25], mientras que **Radius** posee un rango entre [0.0 : 0.1]. Por último, **Size** posee un rango entre [0.0 : 1.0].

Si todos los pasos han sido realizados correctamente, nuestra figura debería lucir de la siguiente manera:

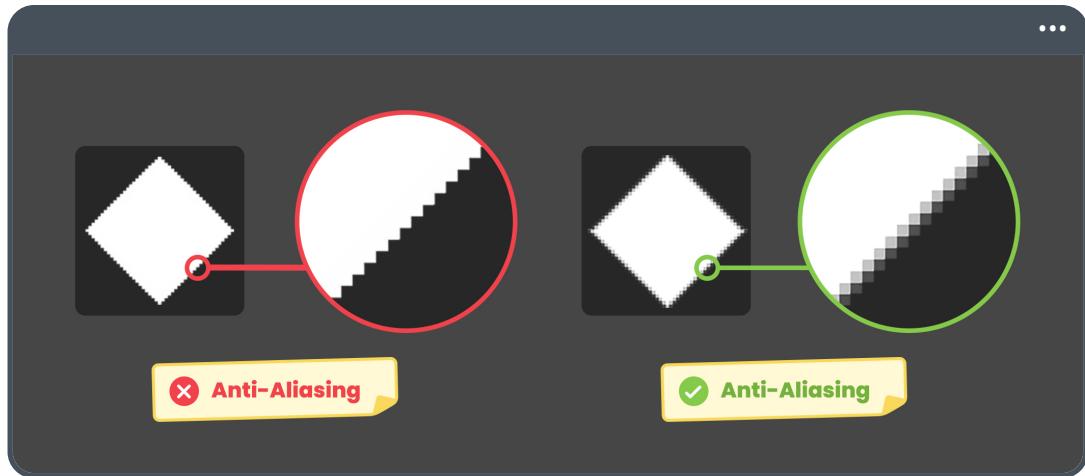


(3.2.d)

Observando la imagen anterior, notaremos que el centro de la forma se ha coloreado con un tono azulado, lo cual demuestra que los valores dentro de la variable **shape** son distintos de cero. Como hemos mencionado anteriormente, esto se puede solucionar fácilmente aplicando la función **saturate()** sobre el primer segmento. Sin embargo, antes de continuar con su implementación, haremos un paréntesis sobre un problema común que ocurre cuando estamos dibujando figuras procedurales, y que está directamente relacionado con su resolución.

Existen dos funciones predefinidas que se utilizan con frecuencia al definir bordes o formas: nos referimos a **step()** y **smoothstep()**. Ambas son herramientas importantes para manipular bordes y transiciones en gráficos procedurales. La función **step()** toma dos argumentos, un umbral y un valor de entrada. Si el valor de entradas es mayor o igual al umbral, la función devuelve 1; de lo contrario, devuelve 0. Esta función es útil para crear transiciones abruptas o bordes muy definidos.

Por ejemplo, si aplicamos la función **step()** directamente sobre el segmento, con un umbral de 0, obtendremos el siguiente resultado cuando rotamos el Quad en nuestra escena.



(3.2.e Se obtienen bordes duros al emplear la función `step()`)

Por otro lado, la función `smoothstep()` permite crear transiciones más suaves entre dos límites. Acepta tres argumentos: un límite inferior, un límite superior y un valor de entrada. La transición entre $[0 : 1]$ se realiza de forma gradual. Sin embargo, también genera artefactos visuales en los bordes cuando rotamos el Quad en el espacio. Esto se debe principalmente a que estamos aplicando una transición suavizada sobre la forma en sí, y no sobre el área en píxeles en la pantalla.

Para solucionar este problema, será necesario crear una función que nos permita generar cierto nivel de anti-aliasing sobre la figura en desarrollo.

3.3 Implementando Anti-Aliasing.

Una manera de generar una transición suavizada entre píxeles es a través de la función `fwidth()`. Según su documentación oficial, esta función devuelve la suma de los valores absolutos de cada derivada parcial aproximada de un input `a` con respecto a ambas coordenadas `xy` en screen-space. ¿Qué significa esto? Prestemos atención a su definición en lenguaje CG para comprenderlo:

```

1 float3 fwidth(float3 a)
2 {
3     return abs(ddx(a)) + abs(ddy(a));
4 }
```

El concepto de “derivada parcial” se refiere a la derivada de una función multivariable con respecto a una de sus variables, manteniendo las otras como constantes. Se utiliza para analizar cómo cambia la función cuando una sola de sus variables independientes varía, mientras las otras permanecen fijas. Por ejemplo, a lo largo de este libro hemos revisado funciones $f(x)$ de una sola variable, pero ¿qué pasaría si tuviéramos una función $f(x, y)$ con dos variables?

La derivada parcial de f con respecto a x se denota como:

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

(3.3.a)

Mientras que la derivada parcial de f con respecto a y se denota como:

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

(3.3.b)

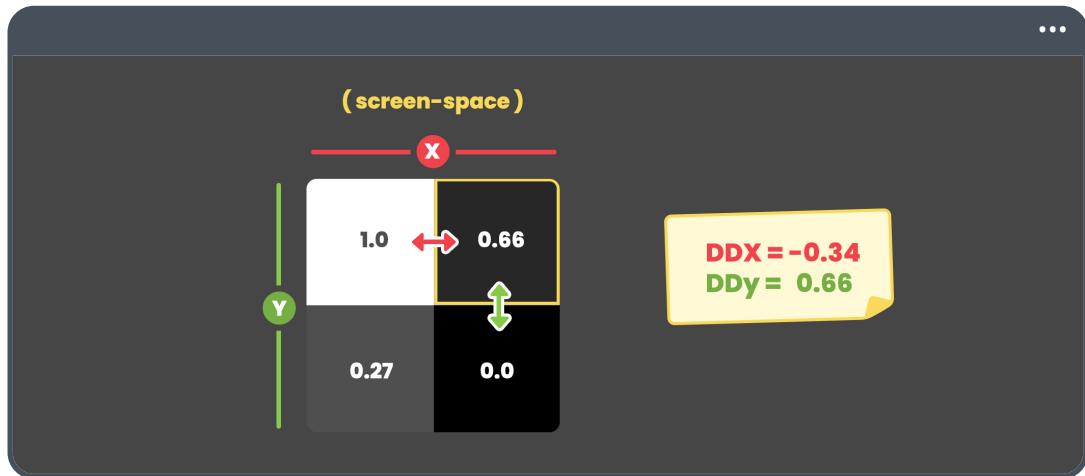
Las funciones `ddx()` y `ddy()`, ambas incluidas en `fwidth()`, son una aproximación de las funciones 3.3.a y 3.3.b respectivamente. Es decir:

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

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

(3.3.c)

Por lo tanto, **ddx()** calcula la tasa de cambio de una expresión con respecto a la coordenada *x* de la pantalla, mientras que **ddy()** realiza la misma operación respecto a la coordenada *y*. Para comprender el proceso, tomemos por ejemplo cuatro píxeles de la pantalla de distinto color y analicemos su tasa de cambio utilizando estas funciones.



(3.3.d)

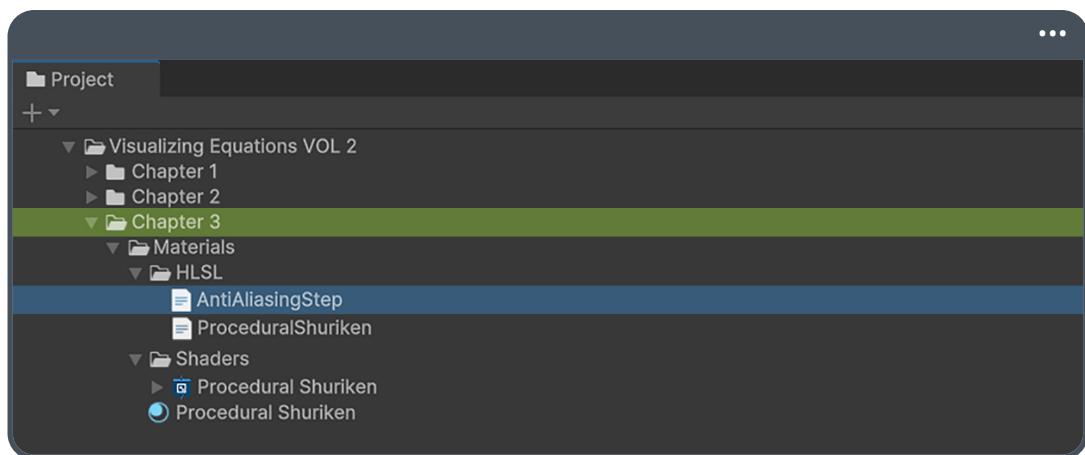
De la referencia anterior, el píxel de la esquina superior derecho marcado en amarillo posee una tasa de cambio igual a -0.34 respecto al píxel que lo precede en la coordenada *x*, mientras que en la coordenada *y*, la tasa de cambio es igual a 0.66. Considerando que **fwidth()** devuelve los valores absolutos tanto de **ddx()** como de **ddy()**, la tasa de cambio de la coordenada *x* del ejemplo anterior sería positiva.

Ahora, ¿cómo podemos generar un efecto de anti-aliasing con estas funciones? Para ello, podríamos llevar a cabo los siguientes pasos:

- Calcular la tasa de cambio de los píxeles que cubre nuestro Shuriken en pantalla.
- Crear un borde donde deseamos aplicar el efecto de anti-aliasing.
- Llevar a cabo una interpolación lineal inversa para suavizar los píxeles de los bordes de la figura.

Iniciaremos incluyendo un nuevo script HLSL en nuestro proyecto, al cual nombraremos **AntiAliasingStep**. Este archivo lo utilizaremos posteriormente sobre todas las formas procedurales que generemos desde este punto en adelante en este libro.

Si todo ha marchado correctamente, nuestro proyecto debería lucir de la siguiente manera:



(3.3.e Estructura del proyecto)

En nuestro nuevo script, declararemos un método de tipo **float3** al cual nombraremos **anti_aliasing_step()**. Este método tendrá dos argumentos: un gradiente, el cual pasaremos como argumento en la función **fwidth()**, y un umbral.

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

Como podemos observar en la línea número 3, se ha declarado e inicializado una nueva variable llamada **rate_of_change**, la cual determina la derivada parcial con respecto a un píxel en el espacio de la pantalla (screen-space). A continuación, debemos definir el borde o rango que ayudará a determinar la región donde deseamos aplicar el efecto de anti-aliasing. Para ello, agregaremos las siguientes líneas de código:

```
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     return 0;
9 }
```

Finalmente, empleando una interpolación lineal inversa, podemos escalar el valor del gradiente a un rango entre [0 : 1] en función de su posición relativa al borde inferior (**lower_edge**) y al borde superior (**upper_edge**). Esto se presenta en el siguiente ejemplo:

```
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(steped);
10
11    return stepped;
12 }
```

Del ejemplo anterior, se puede observar en la línea número 11 que se ha retorna el valor **stepped**, que ahora está garantizado que se encuentra en el rango [0 : 1]. Este valor representa la transición suavizada del gradiente en el área específica del borde, proporcionando un efecto de anti-aliasing.

Para implementar este efecto en nuestro Shuriken procedural, integraremos el método **anti_aliasing_step()** dentro del nodo **procedural_shuriken** de nuestro shader. Esto se logra mediante la directiva **#include**, declarando la ruta del archivo, el cual corresponde a:

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

En consecuencia, nuestro código queda de la siguiente manera:

```

1 #include "Assets/Jettelly Books/.../Segment.hlsl"
2 #include "Assets/Jettelly Books/.../AntiAliasingStep.hlsl"
3
4 float shuriken(float2 uv, float x0, float radius)
5 {
6     float y = abs(uv.y);
7     float x = abs(uv.x);
8
9     float2 p0 = float2(0.0, 0.5);
10    float2 p1 = float2(x0, x0);
11
12    float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
13    segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);
14
15    return segment_p0_p1;
16 }
```

Para una mejor comprensión, si guardamos los cambios y regresamos a la escena, observaremos que el primer segmento se ha dibujado perfectamente en la pantalla. Incluso, si realizamos el ejercicio de rotar el Quad aproximadamente ochenta grados, notaremos que la resolución de la figura se mantiene en perfectas condiciones.

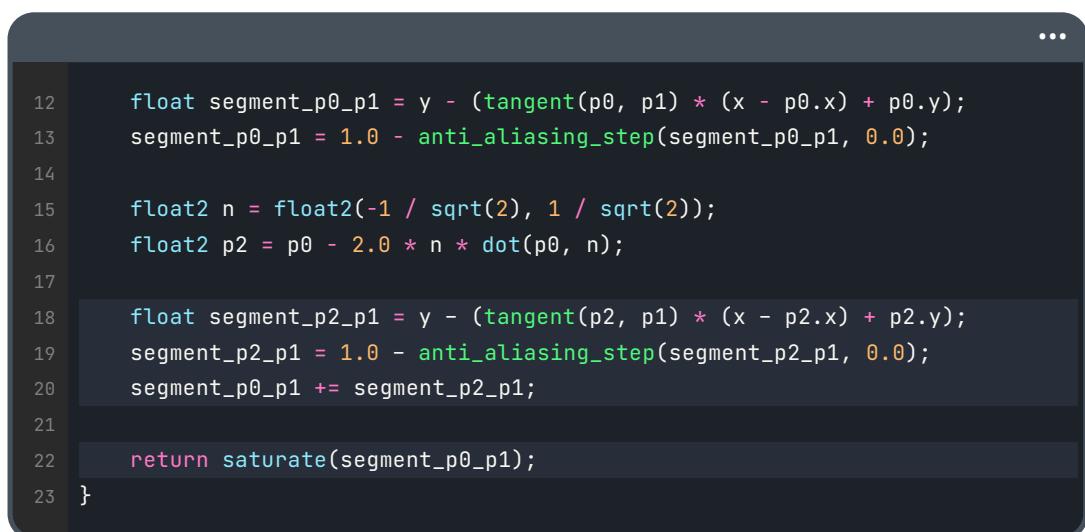
Siguiendo las ecuaciones 3.1.f y 3.1.g, presentadas en la sección anterior, será necesario determinar la normal de la pendiente para establecer la posición del punto **p2**, que será el reflejo del punto **p0**. Para ello, podemos llevar a cabo el siguiente ejercicio:

```

12     float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
13     segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);
14
15     float2 n = float2(-1 / sqrt(2), 1 / sqrt(2));
16     float2 p2 = p0 - 2.0 * n * dot(p0, n);
```

Del código anterior, observamos que se ha declarado una nueva variable **n**, la cual hace referencia a la normal de la pendiente. Posteriormente, se ha empleado esta variable en el cálculo del punto **p2**, el cual utilizaremos para determinar el segundo segmento, aquel que une los puntos $\overline{p_2 p_1}$.

Por consiguiente, vamos a declarar e inicializar una nueva variable a la que denominaremos **segment_p2_p1**. Dado que este nuevo segmento posee la misma estructura del primero en términos matemáticos, emplearemos las mismas funciones, con la diferencia de que aplicaremos los puntos correspondientes para cada caso. Este proceso nos permitirá definir claramente el segundo segmento y asegurar la continuidad y simetría en la forma del Shuriken.

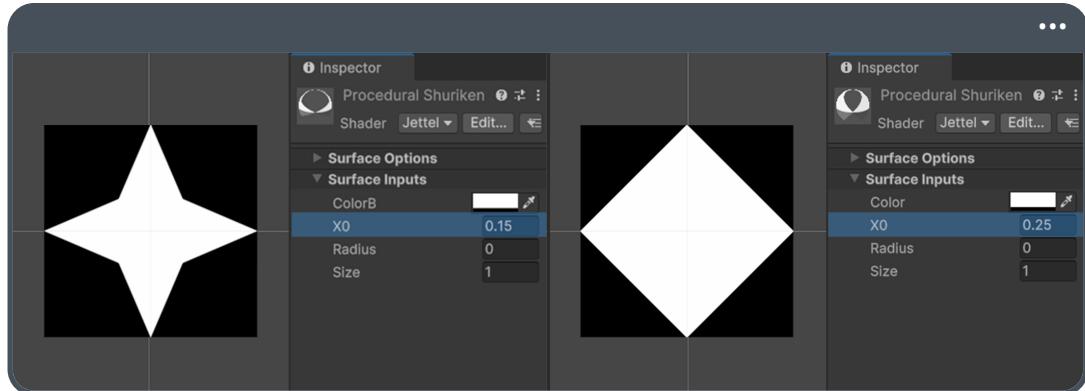


```

12     float segment_p0_p1 = y - (tangent(p0, p1) * (x - p0.x) + p0.y);
13     segment_p0_p1 = 1.0 - anti_aliasing_step(segment_p0_p1, 0.0);
14
15     float2 n = float2(-1 / sqrt(2), 1 / sqrt(2));
16     float2 p2 = p0 - 2.0 * n * dot(p0, n);
17
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     segment_p0_p1 += segment_p2_p1;
21
22     return saturate(segment_p0_p1);
23 }
```

En la línea número 18, se ha declarado e inicializado el segundo segmento, que une los puntos $\overline{p_2 p_1}$. Posteriormente, en la línea siguiente, se ha aplicado anti-aliasing sobre el mismo segmento para obtener una línea suavizada. En la línea número 20, el segundo segmento se ha sumado al primero. Dado que esta operación genera valores fuera del rango que necesitamos en el render, se ha aplicado la función **saturate()** en la línea 22, la cual establece un rango máximo entre [0.0 : 1.0] para el valor de salida.

Al guardar nuestros cambios, si modificamos el valor de la propiedad **x0** desde el Inspector, observaremos cómo se expande y contrae la forma del Shuriken de manera dinámica.



(3.3.f)

Para concluir nuestra figura procedural, solo falta aplicar las fórmulas presentadas en las ecuaciones 1.3.b y 3.1.ñ para generar los círculos que definen la forma que estamos desarrollando. Para ello, agregaremos las siguientes líneas de código:

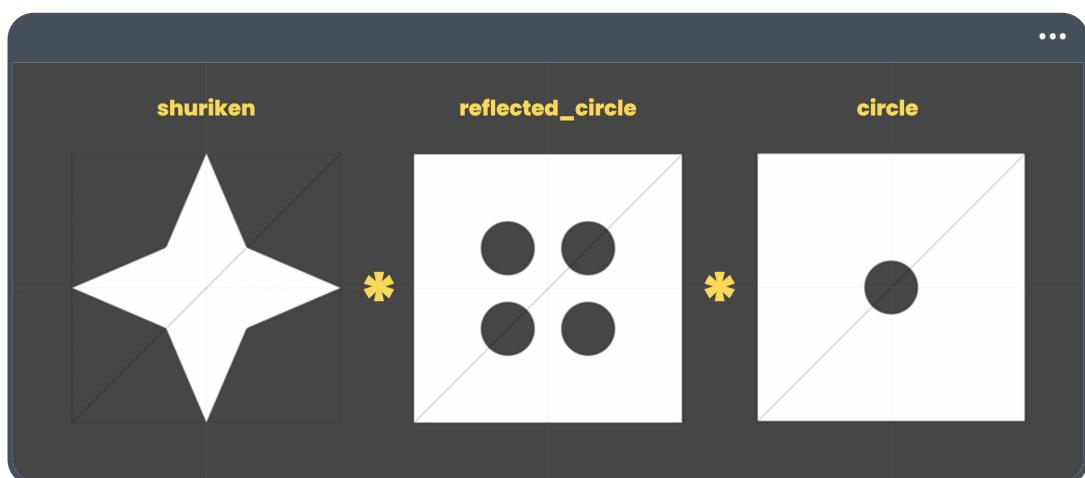
```

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 }
```

En ejemplo anterior, se ha declarado e inicializado una nueva variable llamada **reflected_circle** (línea 22) la cual corresponde a los orificios comunes de cada esquina del Shuriken. El proceso se repite posteriormente en la línea 25, con la declaración

e inicialización de la variable **circle**. No obstante, dado que este último no posee sustracción en sus coordenadas, se mantiene constante en el centro del plano. Cabe destacar que el proceso de declaración e inicialización de los círculos podría haberse optimizado, declarando un nuevo método que contenga las variables pertinentes. Sin embargo, por razones educativas y para mantener la fidelidad con la explicación matemática expuesta en la sección anterior, el ejercicio se ha realizado de la manera en que se presenta.

Finalmente, en las líneas 29 y 30, se ha multiplicado el primer segmento por cada círculo definido. Dado que el área de los círculos es de color negro (igual a 0.0), al realizar la multiplicación, estas áreas son sustraídas del Shuriken original, generando el efecto deseado para cada caso. Es decir:



(3.3.g Composición de un Shuriken 2D procedural)

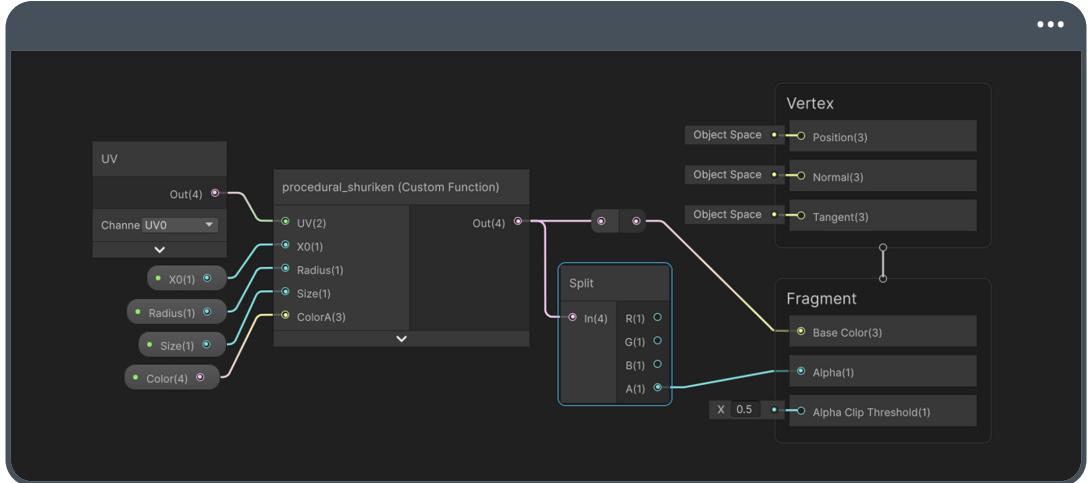
Hasta este punto, nuestra figura procedural está completa a nivel conceptual, por lo tanto, podríamos generar máscaras de color con ella, o algún efecto en específico para UI.

Antes de volver a la escena, nos aseguraremos de implementar una función que nos permita aumentar o disminuir el tamaño del Shuriken. Para ello, realizaremos la siguiente operación dentro del campo de **procedural_shuriken_float()**.

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

Si prestamos atención al ejemplo anterior, observaremos que se han multiplicado las coordenadas UV por una interpolación lineal mediante la función **lerp()** (línea 38). Considerando que la propiedad **size** posee un rango establecido entre [0.0 : 1.0], cuando su valor sea igual a 0.0, las coordenadas UV se multiplicarán por 2.0, lo cual, disminuye el tamaño del Shuriken dentro del Quad. Este efecto podremos apreciarlo con facilidad modificando el valor de la propiedad directamente desde el Inspector en Unity.

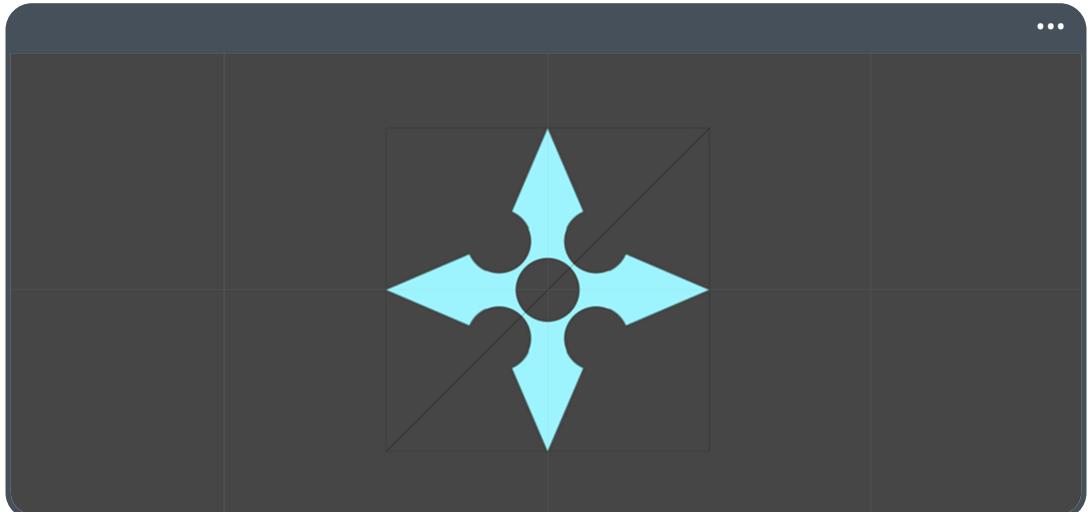
Como último paso, nos aseguraremos de conectar el canal alpha como salida en nuestro shader. Para ello, simplemente podemos utilizar un nodo **Split**, el cual separa los canales de un vector cualquiera.



(3.3.h)

Es cierto que podríamos haber implementado una salida exclusiva para el canal alpha directamente en el nodo **procedural_shuriken**, en lugar de utilizar un nodo **Split**. Sin embargo, optar por este último nos brinda mayor flexibilidad y permite modificar los canales individualmente si es necesario, sin alterar el resto del vector.

Si hemos realizado todos los pasos correctamente, el área de color negro en el Shuriken debería lucir transparente, tal como se muestra en la siguiente imagen.



(3.3.i Cuerpo de un Shuriken en coordenadas UV)

3.4 Definiendo una estética para el Shuriken.

Durante esta sección dedicaremos nuestro tiempo a extender el shader que estamos desarrollando, mejoraremos la estética del mismo mediante capas de color y efectos de luz customizada, los cuales iremos implementando de manera lineal para entender el proceso.

Si tomamos la imagen de un Shuriken real, observaremos que estos se caracterizan por poseer una zona afilada, la cual esencialmente se transmite como un color distinto de su forma general. Por lo tanto, el primer paso en nuestro shader consistiría en agregar soporte a un segundo canal de color para generar una diferencia entre el filo y el cuerpo del Shuriken. Para ello, extenderemos el método **procedural_shuriken_float()**, agregando una nueva entrada de color. Además, nos aseguraremos de renombrar la entrada de color actual para evitar confusiones.

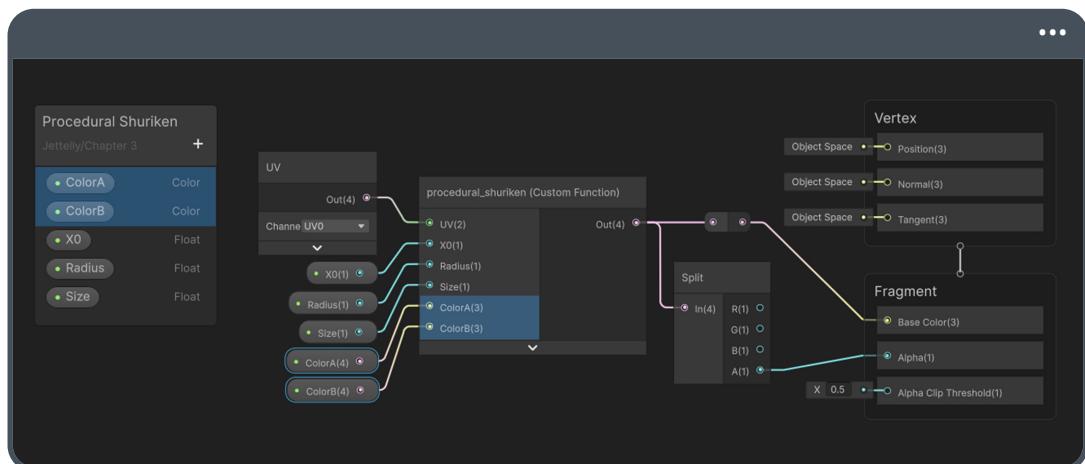
```
35 void procedural_shuriken_float(in float2 uv, in float x0, in float radius,  
    in float size, in float3 colorA, in float3 colorB, out float4 Out) { ... }
```

Como se observa en el ejemplo anterior, se ha incluido una nueva entrada como argumento en la función, que corresponde a un vector tridimensional RGB denominado **colorB**. Además, se ha renombrado el primer vector de **color** como **colorA**. Dado este cambio, inevitablemente nuestro programa dejará de compilar, arrojando un color magenta sobre el Quad en la escena. Por ende, será fundamental actualizar tanto las entradas del nodo **Custom Function** en **Shader Graph**, como las propiedades en el **Blackboard**.

Para solucionar este problema, realizaremos los siguientes pasos:

- Actualizar el nodo **Custom Function**: Nos aseguraremos de incluir el nuevo vector tridimensional denominado **colorB** como entrada y, además, renombrar el actual vector de color como **colorA**.
- Actualizar el **Blackboard**: Iniciaremos renombrando el actual vector de color como **ColorA** y, posteriormente, incluiremos un nuevo color al cual denominaremos **ColorB**. Finalmente, conectaremos las propiedades con el nodo en desarrollo, y guardaremos los cambios.

Si todo ha marchado correctamente, nuestro shader debería lucir de la siguiente manera:



(3.4.a Propiedades de color en el nodo procedural_shuriken)

Habiendo realizado estos cambios, volveremos al código para implementar tanto el filo como el cuerpo del Shuriken, diferenciándolos en color y forma.

```

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

```

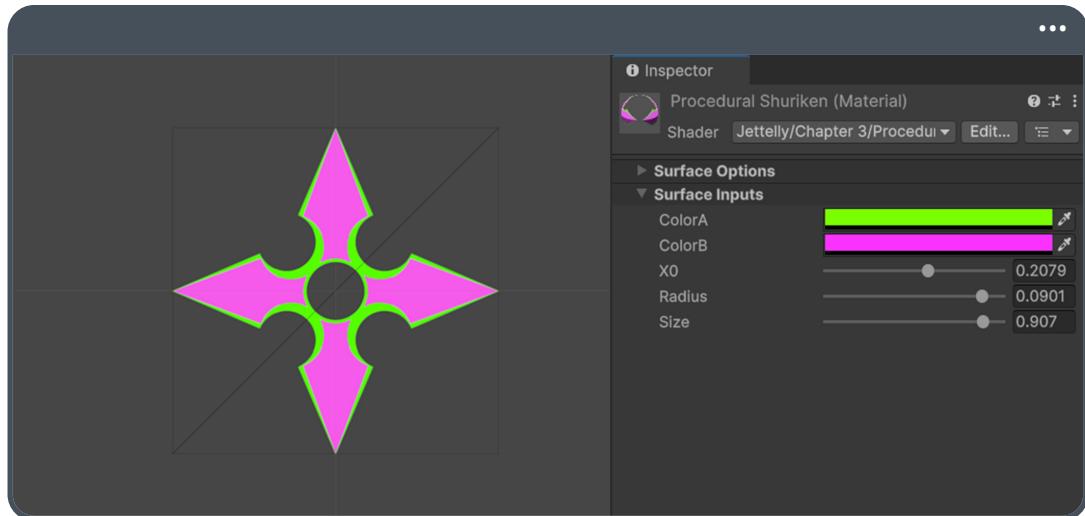
Si prestamos atención a las líneas de código número 41 y 42 del ejemplo anterior, observaremos que se han declarado e inicializado dos nuevas variables denominadas **sharp** y **body**, las cuales hacen referencia directamente a los componentes del Shuriken. Además, en la línea número 40, se ha declarado e inicializado una nueva variable constante denominada **d0**. Su objetivo es generar una diferencia entre el filo y el cuerpo del mismo, y para ello, multiplica los distintos argumentos según cada caso.

Si deseamos ver ambas formas trabajando en conjunto, una técnica de color que podemos emplear consiste en pasar dos colores en una interpolación lineal, y luego, retornar el primero o el segundo dependiendo de una entrada en escala de grises. Podemos deducir que los dos colores de entrada corresponden a **colorA** y **colorB**, mientras que la escala de grises podría ser la multiplicación entre **sharp** y **body**. Sin embargo, para este caso será necesario que una de las dos formas tenga una gama o brillo distinto del otro, de otra manera, no podremos utilizarlo como argumento en la interpolación lineal.

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

En el ejemplo anterior, se ha declarado e inicializado una nueva variable denominada **v0** (línea 44), la cual corresponde a la multiplicación entre **sharp** y **body**, con la característica que **d0** multiplica al primero para generar una diferencia de brillo, lo cual se traduce como una escala de grises. Posteriormente, en la línea 47, se introduce una interpolación lineal a cada componente RGB del vector **shape**, donde el valor mínimo es igual a **colorA**; el máximo, **colorB**; y la escala de grises, **v0**. Además, el cuarto componente A de **shape** adquiere el valor de la variable **sharp**, lo cual es fundamental para establecer el canal de transparencia de la forma total.

Si guardamos los cambios y regresamos a la escena, podremos observar cómo ambos colores afectan el cuerpo del Shuriken.

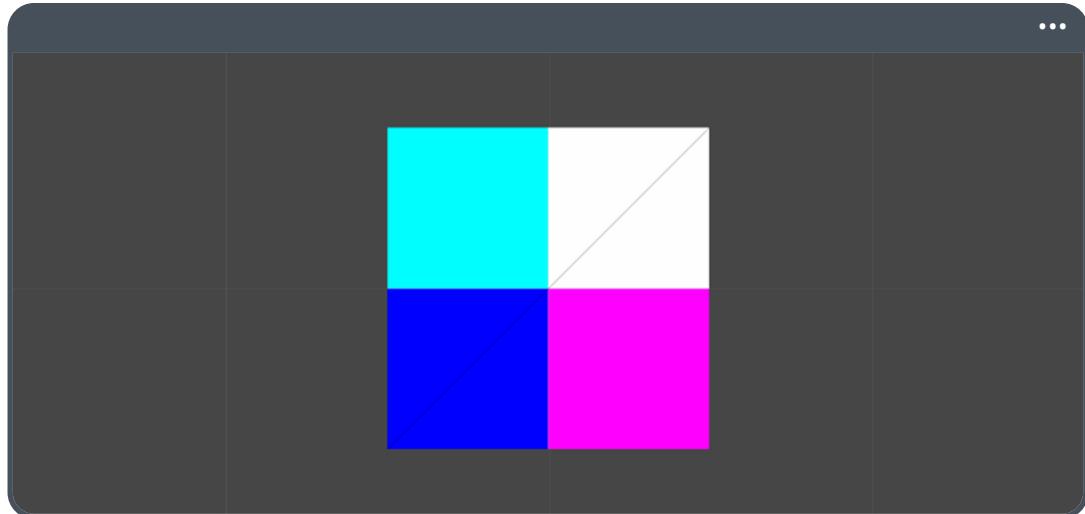


(3.4.b)

En este punto, podríamos dar por terminada nuestra figura procedural. Sin embargo, dado que la forma como tal carece de volumen, exploraremos nuestra visión artística para incorporar una capa de luz a la misma. Para ello, considerando las coordenadas UV, agregaremos un bloque de color distinto a cada cuadrante, de manera que obtengamos un equilibrio entre zonas oscuras y claras en la figura. Iniciaremos declarando un nuevo vector tridimensional en la función, como se muestra a continuación:

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

Como ya sabemos, el método `anti_aliasing_step()` retorna una transición suavizada de un gradiente cualquiera. Si prestamos atención a la línea 45 del código anterior, observaremos que, se ha empleado un vector tridimensional RGB como argumento en la función, donde los dos primeros corresponden a las coordenadas UV, y el tercero es un número constante igual a 1. La salida de este vector produce el siguiente resultado gráfico:



(3.4.c)

¿Por qué razón se generan estos colores? La respuesta está en las coordenadas UV. Mientras que el tercer componente B es un valor constante, las coordenadas UV corresponden a un rango entre [-0.5 : 0.5] lo cual significa que el método `anti_aliasing_step()` va a crear regiones donde la interpolación resulta en diferentes valores para cada cuadrante, es decir:

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

Sin embargo, dado que únicamente necesitamos aplicar estos matices a modo de luces y sombras, transformaremos el equivalente de cada uno en escala de grises. Para ello, emplearemos la siguiente función:

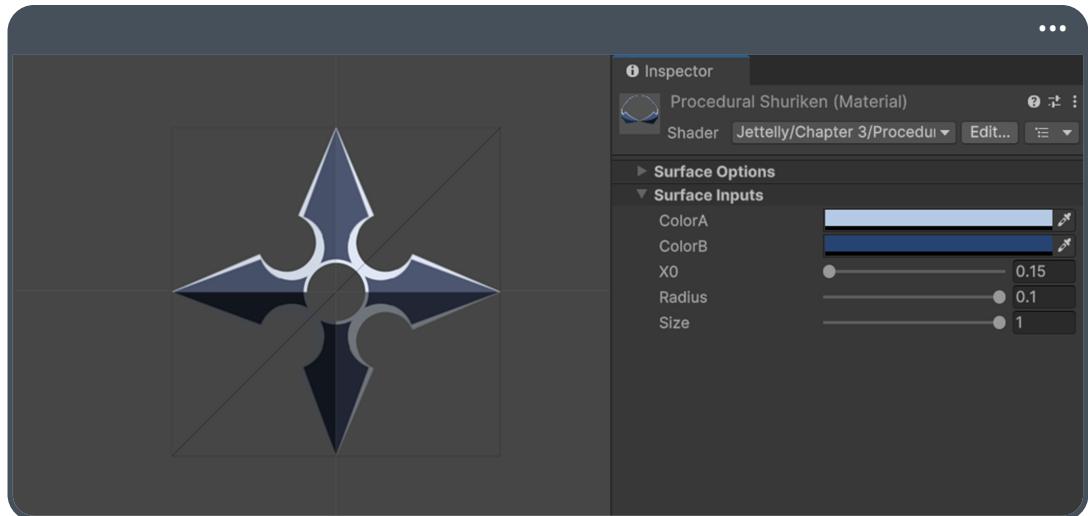
```

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

```

Si prestamos atención a la línea número 46, observaremos que se ha declarado una nueva variable denominada **l0_gray**, la cual adquiere el resultado del producto punto entre **l0** y un vector tridimensional que corresponde a los pesos utilizados para convertir un color a su equivalente en escala de grises. Estos pesos definen a los coeficientes de luminiscencia relativa para los componentes RGB en lineal-space. Posteriormente, en la línea 50, los componentes RGB de **shape**; nuestra forma final, son multiplicados por **l0_gray**, generando consigo áreas claras y oscuras en el Shuriken, como se observa en la siguiente imagen:

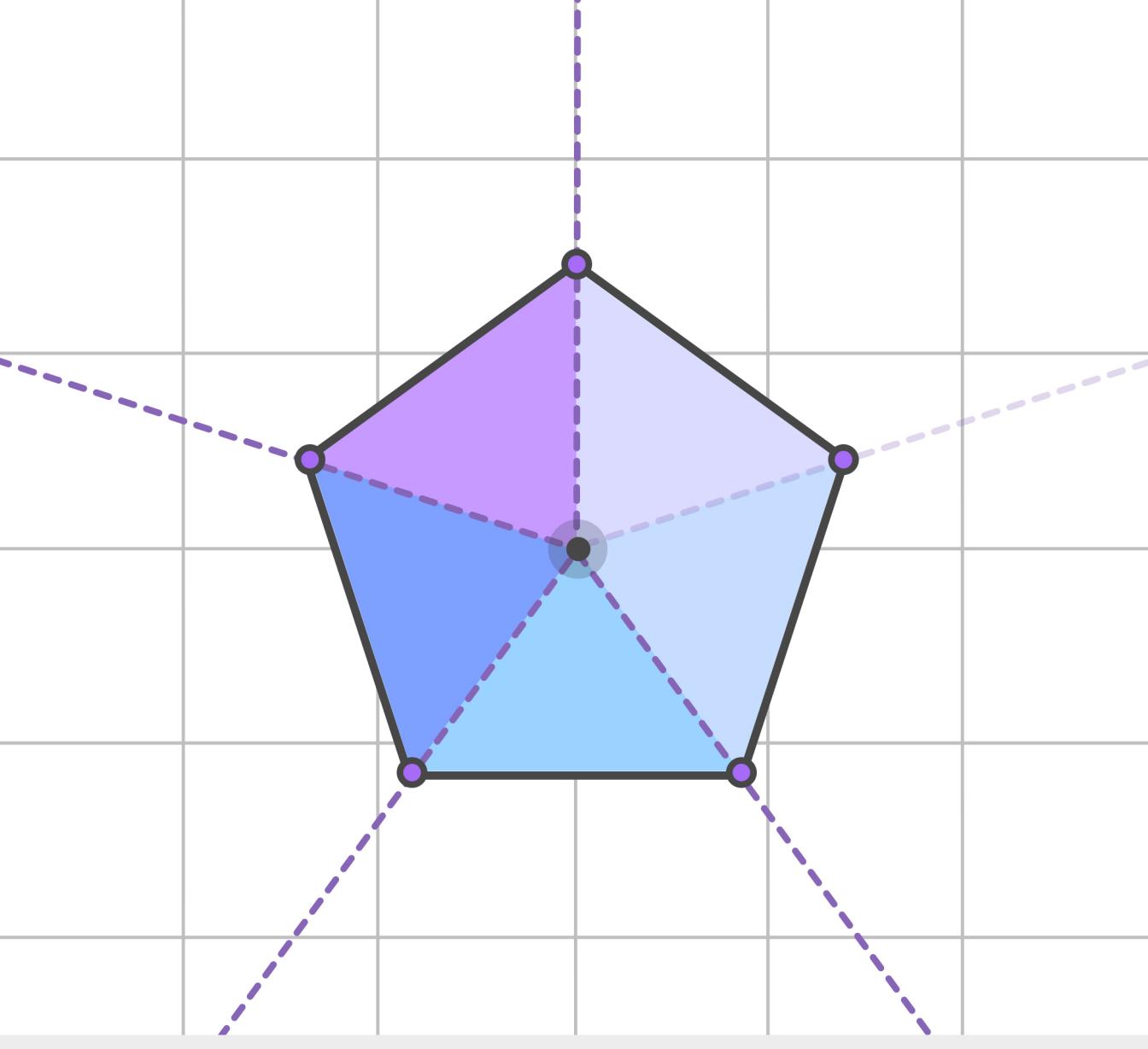
Figuras procedurales.



(3.4.d)

Resumen del capítulo.

- En este capítulo, se exploraron técnicas para la creación de figuras procedurales utilizando funciones matemáticas, enfocándonos en el desarrollo de un Shuriken como caso de estudio. Iniciamos con una introducción al cuerpo de un Shuriken desde un punto de vista matemático, donde se demuestra cómo descomponer una figura compleja en componentes geométricos simples. Luego, en la sección dos, ponemos en práctica los conceptos matemáticos vistos hasta entonces para desarrollar el Shuriken en Shader Graph utilizando HLSL. Durante esta sección se explicó cómo traducir las fórmulas en operaciones gráficas para obtener una figura precisa y optimizada.
- Posteriormente, se aborda el problema de los bordes pixelados comunes en figuras procedurales. Se introduce la función `fwidth()` como una solución para implementar anti-aliasing, mejorando la suavidad de los contornos del Shuriken. Finalmente, con la forma del Shuriken finalizada, se aplican técnicas de color e iluminación para darle una estética visual atractiva al render.



Capítulo 4

Función de distancia con signo.

En los primeros capítulos de este libro, te enfocaste en la creación de figuras utilizando principalmente funciones algebraicas y trigonométricas, las cuales establecieron una base sólida para la generación de formas básicas. Sin embargo, para crear geometrías más complejas y detalladas, necesitarás ampliar tu enfoque.

En este capítulo, explorarás el fascinante mundo de las funciones de distancia con signo (SDF por sus siglas en inglés), con un énfasis especial en aquellas de carácter bidimensionales. Las SDF te permitirán representar la geometría de manera eficiente y precisa, proporcionando una herramienta poderosa para definir y manipular formas procedurales. A través de ejemplos y aplicaciones prácticas, verás cómo estas funciones pueden simplificar la creación de formas complejas y abrir nuevas posibilidades en el desarrollo de arte técnico en Unity.

Te sumergirás en la teoría detrás de las SDF, aprenderás a implementarlas en tus proyectos y descubrirás cómo integrarlas con otras técnicas para lograr efectos visuales sorprendentes. Este enfoque no sólo mejorará tu comprensión de la geometría procedural, sino que también te brindará herramientas valiosas para innovar en el diseño y desarrollo de videojuegos.

4.1 Naturaleza de una SDF.

Hasta este punto, hemos puesto en práctica la implementación de funciones polinomiales y trigonométricas en HLSL, las cuales utilizamos para dibujar formas procedurales a partir de áreas geométricas y segmentos de rectas. Sin embargo, cuando se trata de formas complejas, las funciones explícitas, como aquellas definidas en la forma $y = mx + b$, a menudo resultan insuficientes cuando necesitamos representar curvas y contornos irregulares.

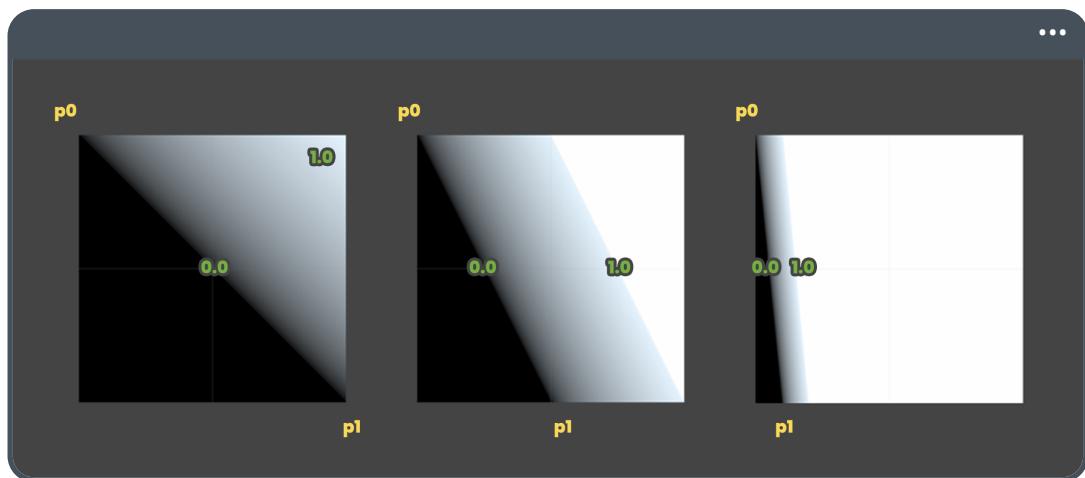
En estos casos, es necesario recurrir a las funciones implícitas o representaciones paramétricas que nos brinden mayor flexibilidad y precisión. Aquí es donde entra el concepto de función de distancia con signo, que nos permite describir con exactitud la

distancia de un punto a una superficie, indicando además si dicho punto se encuentra dentro, fuera o sobre la superficie misma.

Para entender este concepto, compararemos dos segmentos: uno definido de manera explícita, siguiendo la ecuación 1.1.a del capítulo 1, y otro de manera implícita mediante la técnica SDF. Para este último, consideraremos:

- Todos los puntos del primer cuadrante.
- La distancia entre los puntos y el segmento que vamos a definir.

Es posible que estos conceptos parezcan difíciles de comprender al principio. Sin embargo, a medida que avancemos, profundizaremos en la lógica detrás de esta técnica, detallando paso a paso su implementación para distintas formas. Prestemos atención a un gradiente de un segmento en HLSL definido en la forma lineal.

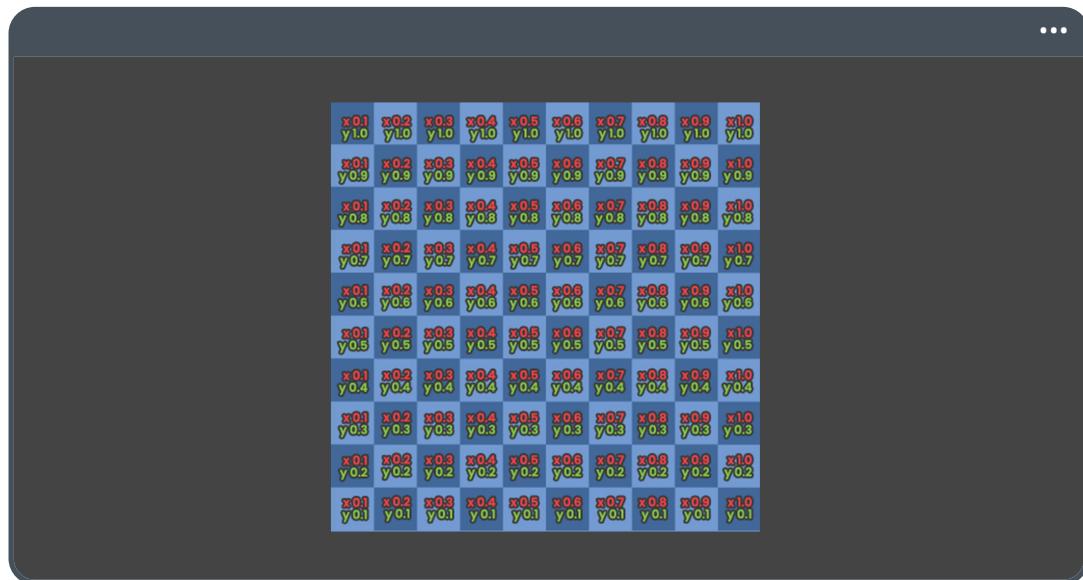


(4.1.a Gradiente de una función lineal)

Como podemos observar en la imagen 4.1.a, el gradiente se expande o contrae según la posición de los puntos p_0 y p_1 . Este comportamiento, aunque matemáticamente correcto, presenta un problema al definir formas complejas en nuestro programa. Por ejemplo, si aplicamos la función `smoothstep()` (interpolación suavizada entre dos valores de entrada) sobre la operación resultante, obtendremos un degradado desigual, influenciado por la posición de ambos puntos. Para corregir este problema, podríamos emplear un

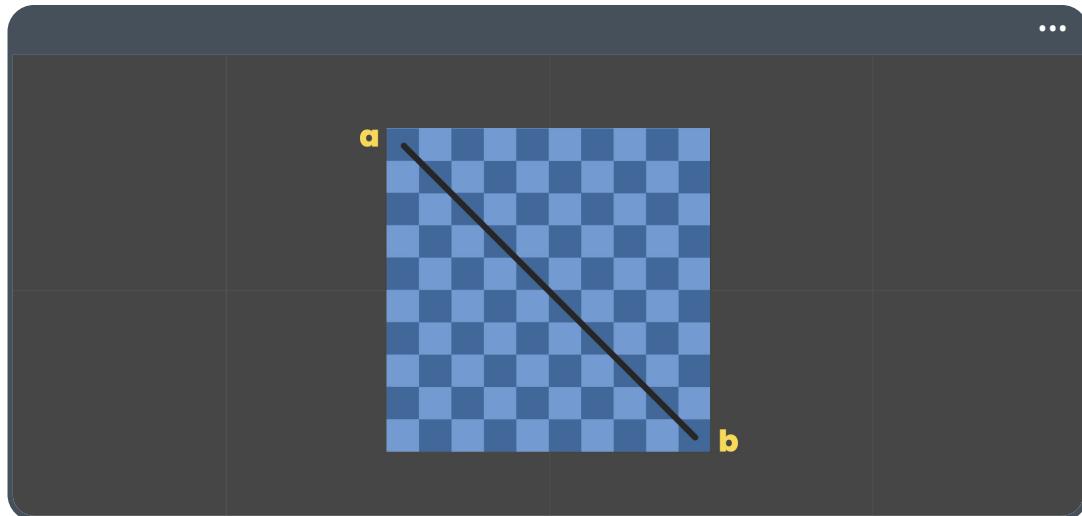
enfoque distinto, que consiste en calcular la distancia entre los puntos fuera del segmento (el área en color blanco en la imagen) y el segmento en sí mismo.

Para comprender de mejor manera este concepto, imaginemos el primer cuadrante de las coordenadas cartesianas como una rejilla. Cada espacio en la rejilla representa una posición específica, que podemos determinar como un vector bidimensional.



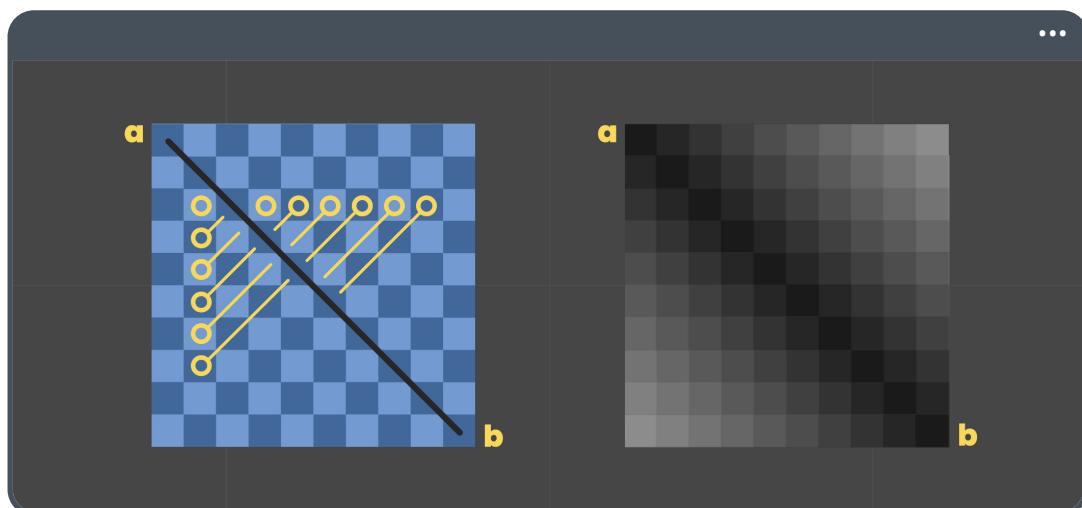
(4.1.b)

A modo de ejemplo, hemos empleado únicamente cien espacios en la rejilla de la imagen 4.1.b, que utilizaremos para exemplificar el funcionamiento de la función de distancia con signo. Como es de suponer, cada espacio representa un punto que eventualmente podríamos utilizar para realizar una operación. Por ejemplo, podríamos dibujar un segmento diagonal que pase entre los puntos $a = (0.1, 1.0)$ y $b = (1.0, 0.0)$, como se muestra a continuación:



(4.1.c Dos puntos en un plano cartesiano)

El segmento que observamos en la imagen 4.1.c podría fácilmente representarse en la forma lineal. No obstante, ya sabemos que este enfoque genera detalles gráficos cuando los puntos tienden a cero. Por lo tanto, podemos emplear la función de distancia con signo para obtener el resultado esperado.



(4.1.d)

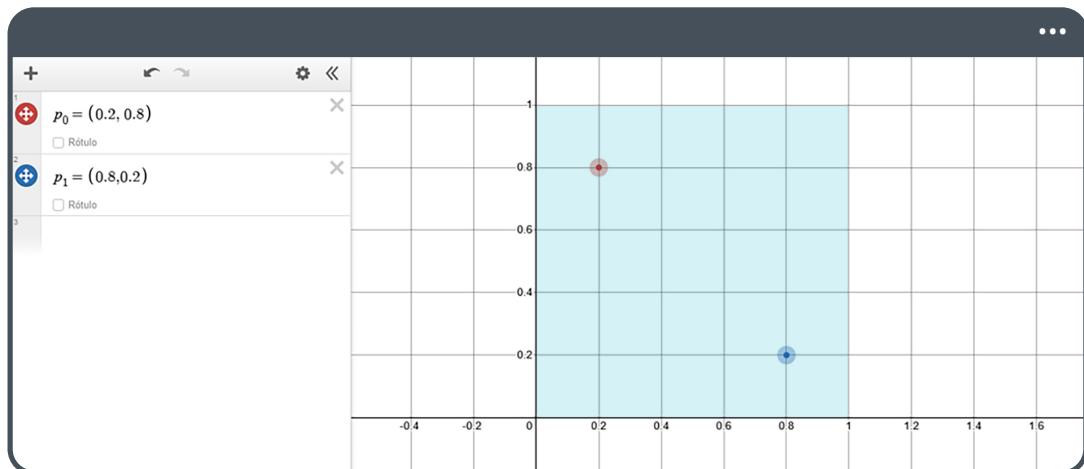
De la imagen 4.1.d, la referencia de la izquierda representa la distancia entre puntos aleatorios y el segmento, mientras que la referencia de la derecha muestra el resultado

en color que obtendremos al calcular esa distancia en HLSL. Cuanto más cercano sea el punto al segmento, más oscuro es el color del píxel. De manera similar, cuanto más lejos esté el punto, más claro será. Por consiguiente, los puntos fuera del área del segmento retornan números positivos, mientras que los puntos dentro del área del segmento retornan número negativos. Un valor igual a cero indica que los puntos se encuentran exactamente en el segmento.

Esta técnica evita los problemas que podríamos obtener al trabajar con funciones explícitas, permitiéndonos generar formas complejas tanto en dos como en tres dimensiones. Ahora, ¿cómo podríamos calcular un segmento utilizando distancia? Al igual que en la forma explícita, debemos considerar:

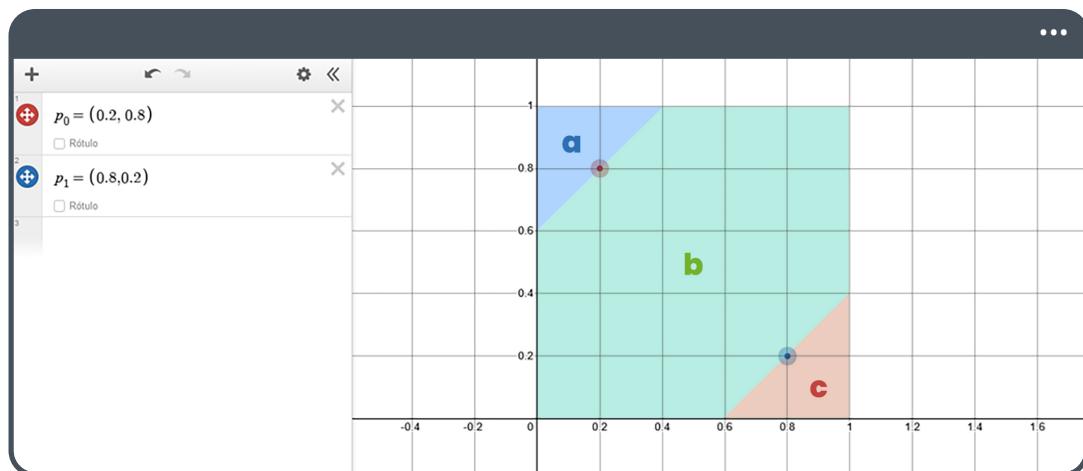
- La posición del primer punto.
- La posición del segundo punto.
- El segmento entre ambos puntos.

Para entender mejor este ejercicio, vamos a definir dos puntos: p_0 y p_1 en el plano cartesiano.



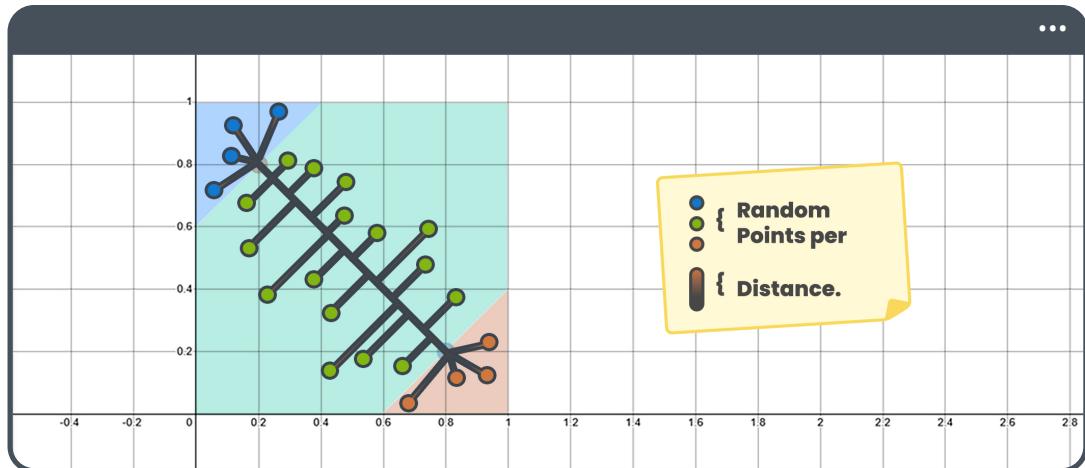
(4.1.e <https://www.desmos.com/calculator/v0ihcwldiz>)

Un factor importante por considerar al posicionar dos puntos es que, si trazamos un segmento entre ellos, la línea resultante será infinita. Por lo tanto, será necesario limitar el segmento en algún momento para obtener una forma más compacta y adecuada para nuestros proyectos. Por ejemplo, si limitamos el segmento en los puntos definidos anteriormente, obtendremos tres áreas que aparecen de manera natural.



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

Si prestamos atención a la imagen 4.1.f, notaremos que las tres áreas han sido denotadas con las letras *a*, *b* y *c* para diferenciarlas. Ahora, ¿qué ocurriría si calculamos la distancia entre los puntos de cada área respecto al punto o segmento más cercano? Por ejemplo, tanto el área *a* como el área *c* generarían un semicírculo, ya que en ambos casos todos los puntos culminan en un solo punto. Por el contrario, en el área *b* se generaría un segmento que uniría ambos puntos p_0 y p_1 .



(4.1.g)

En la imagen 4.1.g, los círculos de color representan puntos “aleatorios” dentro de cada área, mientras que las líneas grisáceas indican la distancia de cada punto respecto a los puntos p_0 , p_1 , y el segmento. Cabe preguntarnos entonces, ¿cómo podríamos determinar estas distancias para cada área? Por ejemplo, si tomamos un punto cualquiera $p_n = (0.1, 1.0)$ del área a respecto al punto $p_0 = (0.2, 0.8)$, podemos calcular su distancia d empleando la siguiente función:

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

(4.1.h)

Donde el vector a de la ecuación 4.1.h representa a un punto aleatorio dentro de las coordenadas UV, mientras que b representa un punto definido. Utilizando los puntos p_n y p_0 en el esquema, obtendremos el siguiente resultado:

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

(4.1.i)

Si llevamos a cabo el ejercicio, notaremos que:

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

(4.1.j)

Lo que es igual a,

$$d = \sqrt{0.05}$$

(4.1.k)

Por lo tanto,

$$d \approx 0.2236$$

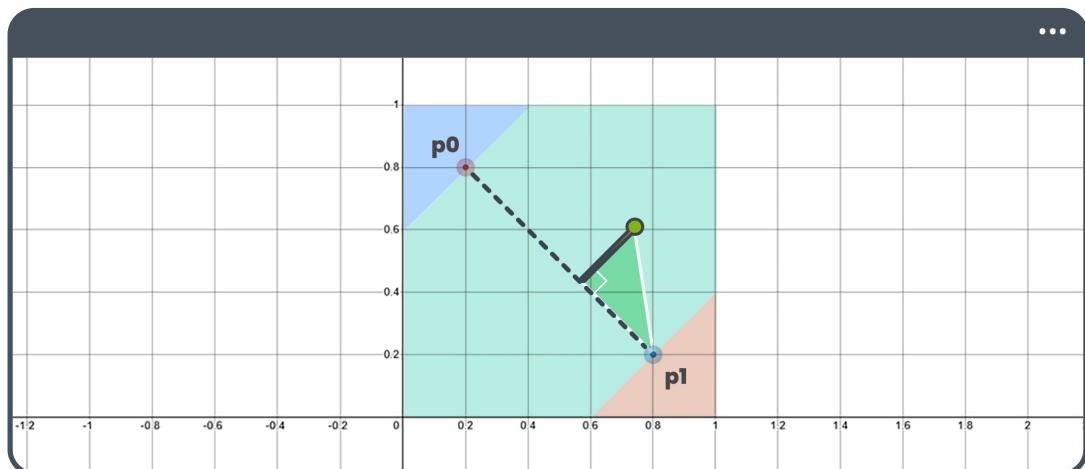
(4.1.l)

El resultado que hemos obtenido en el ejercicio anterior (4.1.l) no tan solo refleja la distancia de un punto respecto al otro, sino también el color del píxel resultante en pantalla en una escala de grises. Cabe destacar que las distintas áreas presentadas anteriormente corresponden a conjuntos de puntos. Por lo tanto, si tuviéramos que implementar la función de la distancia considerando las coordenadas UV, obtendríamos el siguiente resultado:

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

(4.1.m)

Dado que el segmento posee un segundo punto denominado p_1 , podríamos llevar a cabo el mismo ejercicio para calcular la distancia entre los puntos del área c y este último. Sin embargo, ¿qué tendríamos que hacer para calcular el segmento como tal? Para ello, debemos determinar qué es el segmento en términos matemáticos.



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

El segmento es aquella línea que pasa entre los puntos p_0 y p_1 . Si prestamos atención a la imagen anterior, observaremos que, al calcular la distancia entre un punto aleatorio y el segmento, se forma un triángulo rectángulo de manera natural. Por lo tanto, podríamos emplear una proyección para calcular todos los puntos del grupo b que pasan entre los puntos definidos.

Podemos utilizar la siguiente función para calcular dicha proyección:

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

(4.1.ñ)

Donde la variable a corresponde al grupo de puntos por áreas, es decir, las coordenadas UV, b corresponde al primer punto p_0 , y c es igual a p_1 . Una vez que obtenemos la proyección de aquellos puntos ubicados entre los dos puntos mencionados, únicamente

necesitaríamos calcular la distancia entre los puntos de cada área respecto a un nuevo punto p_2 , el cual podemos determinar de la siguiente manera:

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

(4.1.o)

Donde nuevamente, b corresponde al primer punto; y c , al segundo.

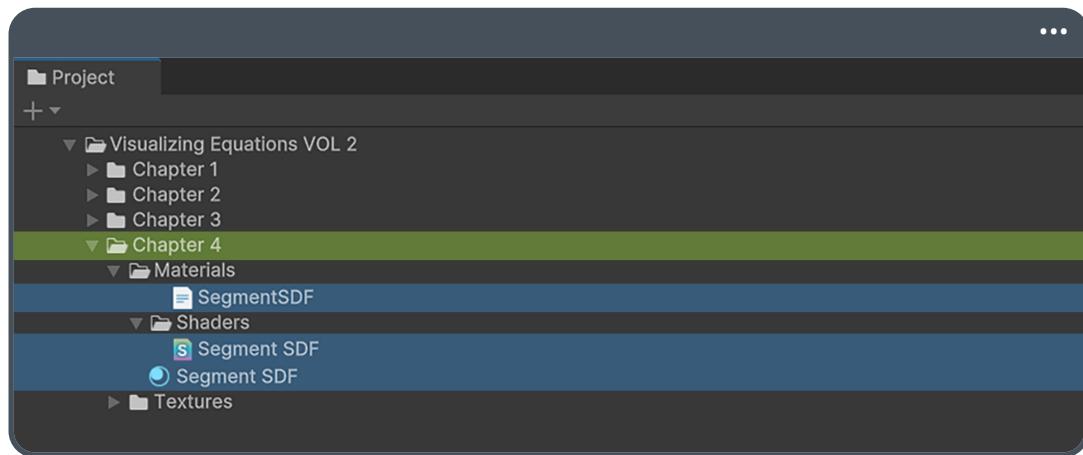
4.2 Dibujando un segmento SDF en la interfaz de usuario.

Hasta este punto, hemos aprendido que, para crear una función de distancia con signo, es necesario calcular la distancia entre los puntos de una región y un punto o segmento específico. En esta sección, aplicaremos este conocimiento para recrear el segmento SDF explicado anteriormente, pero adaptándolo a un shader con soporte para UI. Esto nos permitirá integrar nuestro efecto en elementos gráficos de interfaz de usuario, ampliando su aplicabilidad.

Es importante mencionar que la versión 6.0.0.29f1 de Unity, que estamos utilizando actualmente, incluye soporte para shaders de UI creados con Shader Graph. No obstante, en esta ocasión trabajaremos con un shader general para explotar el flujo completo de creación y personalización de shaders. Este enfoque no solo facilitará la comprensión de conceptos fundamentales, sino que también nos permitirá aplicar estas técnicas en una variedad de contextos más allá de la interfaz de usuario. Para comenzar, crearemos un shader de tipo **Unlit** en nuestra carpeta de proyecto, al cual denominaremos **Segment SDF**.

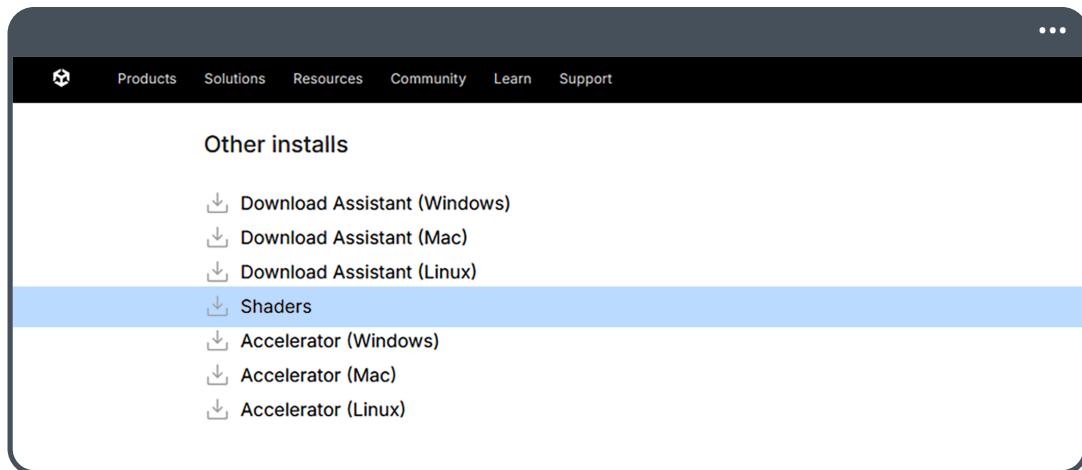
- Assets > Create > Shader > Unlit Shader.

Adicionalmente, generaremos un material y un archivo **.hlsl** empleando el mismo nombre.



(4.2.a Estructura del proyecto)

El shader creado tendrá la extensión **.shader**, lo que lo diferencia de los archivos **.shadergraph** que hemos utilizado anteriormente en este proyecto. Cabe resaltar que, por defecto, este tipo de shader no incluye soporte para UI, por lo que será necesario agregar ciertas líneas de código adicionales para garantizar su compatibilidad con la versión de Unity que estamos empleando. Sin embargo, para simplificar el proceso de aprendizaje y centrarnos en los aspectos esenciales, descargaremos el shader **UI-Default**, disponible dentro del archivo **.zip** proporcionado por Unity. Este shader servirá como base para entender y construir el efecto deseado de manera eficiente.



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

Una vez descargado este archivo, podremos encontrar el shader siguiendo la ruta:

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

El único paso que debemos llevar a cabo a continuación es copiar todo el código que se encuentra dentro del campo del shader en sí mismo, es decir, las propiedades (**Properties**) y el **SubShader**, y pegarlo en el shader recientemente creado. Si todo ha marchado bien, las propiedades de nuestro shader deberían lucir de la siguiente manera:

```

1 Shader "Jettelly/Chapter 4/UI/Segment SDF"
2 {
3     Properties
4     {
5         [PerRendererData] _MainTex ("Sprite Texture", 2D) = "white" {}
6         _Color ("Tint", Color) = (1, 1, 1, 1)
7
8         _StencilComp ("Stencil Comparison", Float) = 8
9         _Stencil ("Stencil ID", Float) = 0
10        _StencilOp ("Stencil Operation", Float) = 0
11        _StencilWriteMask ("Stencil Write Mask", Float) = 255
12        _StencilReadMask ("Stencil Read Mask", Float) = 255
13
14        _ColorMask ("Color Mask", Float) = 15
15        [Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip ("Use Alpha Clip",
16            →   Float) = 0
17    }
18
19    SubShader { ... }
}

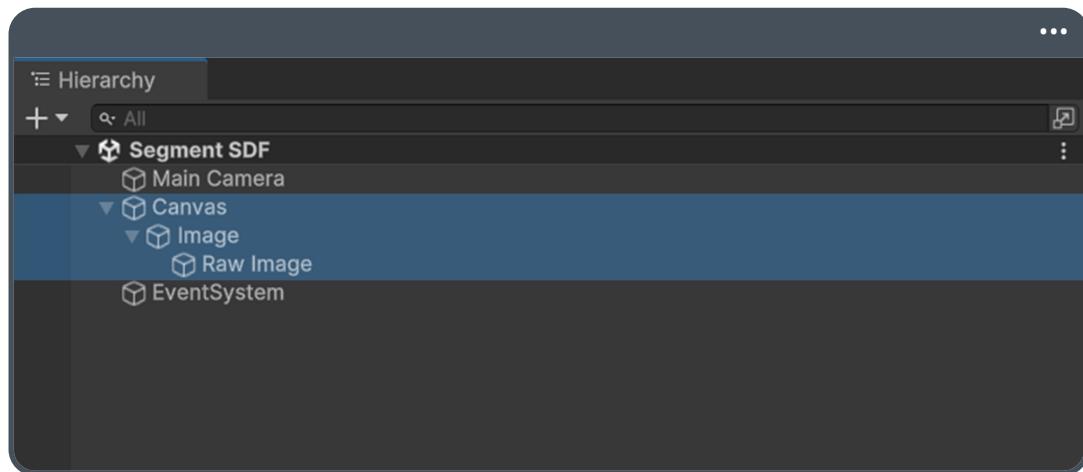
```

Hay otras funciones interesantes incluidas en nuestro código que definen el comportamiento del shader sobre imágenes para UI. Sin embargo, no enfatizaremos en ellas, ya que el enfoque de este capítulo estará en la explicación de las funciones de distancia con signo.

Antes de comenzar con la implementación de funciones en nuestro script, realizaremos una pequeña configuración en la ventana **Hierarchy**, que incluirá los siguientes elementos:

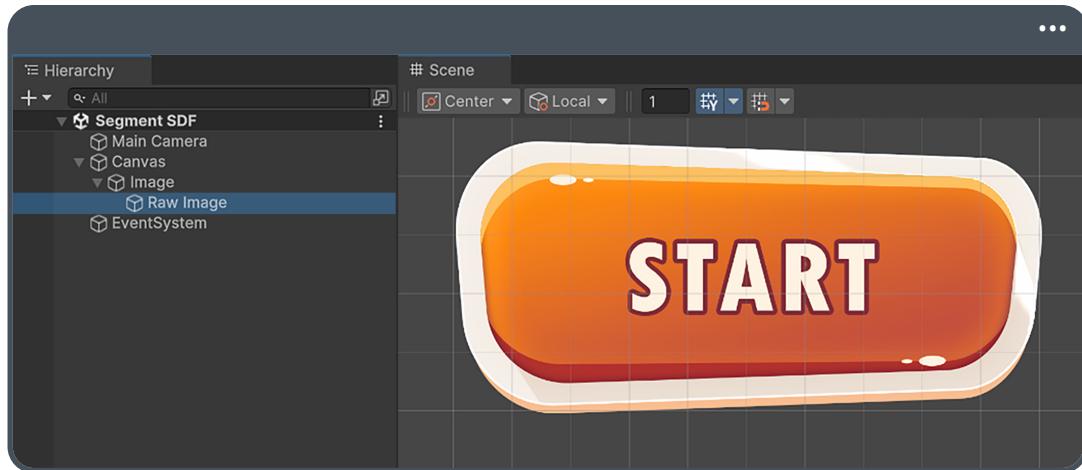
- Un Canvas.
- Una imagen.
- Una imagen de tipo “Raw”.

Agruparemos estos elementos de la siguiente manera:



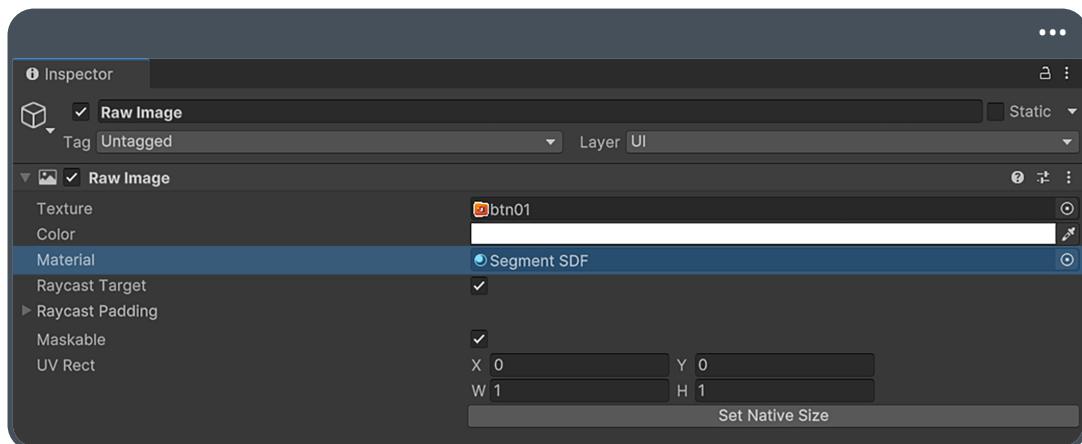
(4.2.c Canvas en la ventana Hierarchy)

Cabe destacar que aplicaremos el material **Segment SDF** directamente sobre el componente **Raw Image**, dejando libre al componente **Image**. ¿Por qué razón? Debido a su configuración de coordenadas UV. El componente **Image** posee compatibilidad con **Atlas** (o Spritesheet), lo cual significa que puede ser posicionado dentro de un conjunto de imágenes y sus coordenadas se ajustarán a la nueva posición dentro del mismo. En cambio, **Raw Image** no posee esta compatibilidad debido a que está diseñado para texturas en general, lo cual lo hace perfecto para efectos visuales en UI. En este caso, **Raw Image** funcionará como una segunda capa, que puede ser aditiva, multiplicativa o tener cualquier **Blending Mode** que deseemos para nuestro efecto, preservando los márgenes de la imagen como tal.



(4.2.d)

Por lo tanto, nos aseguraremos de incluir el shader **Segment SDF** en su respectivo material. Luego, asignaremos este material en la propiedad **Material** del componente **Raw Image**, como se muestra a continuación:



(4.2.e Material customizado en la propiedad Material)

Con respecto a la textura o imagen de UI que utilizaremos para el ejercicio, dentro del paquete de este libro, en su respectiva sección, podrás encontrar imágenes libres de licencia, creadas por el equipo de Jettelly. Independientemente de aquella que escojas para trabajar, será necesario incluir la imagen seleccionada tanto en el componente

Image como en **Raw Image** para preservar los bordes donde aplicaremos el efecto posteriormente.

Habiendo configurado la interfaz de usuario, nos aseguraremos de incluir el script **.hlsl** en el **.shader** con el que estamos trabajando. Para ello, agregaremos la siguiente línea de código:

```
53 #include "UnityCG.cginc"
54 #include "UnityUI.cginc"
55 #include "Assets/Jettelly Books/... /SegmentSDF.hlsl"
```

Aunque de momento no hemos agregado ninguna línea de código en el script HLSL, la directiva **#include** nos permitirá utilizar funciones directamente desde este último. Por lo tanto, el siguiente paso consiste en agregar las funciones que utilizaremos para la representación del segmento de distancia con signo. Para ello, comenzaremos implementando la proyección detallada en la sección anterior, ecuación 4.1.ñ.

Es cierto que, para una mejor explicación lineal, podríamos empezar definiendo la función de distancia entre dos puntos. Sin embargo, HLSL ya posee una función que realiza esta operación, la cual luce de la siguiente manera:

```
1 float distance(float2 pt1, float2 pt2)
2 {
3     float2 v = pt2 - pt1;
4     return sqrt(dot(v, v));
5 }
```

Lo cual es equivalente a,

```
1 float distance(float2 p, float2 uv)
2 {
3     return sqrt(pow(uv.x - p.x, 2.0) + pow(uv.y - p.y, 2.0));
4 }
```

Esta última se asemeja mucho más a la definición de la función presentada en la sección anterior, ecuación 4.1.m. Por consiguiente, iniciaremos implementando la función de proyección en HLSL, considerando las siguientes variables:

- La variable **a** (coordenadas UV).
- La variable **b** (primer punto).
- La variable **c** (segundo punto).

Por ende, declararemos el siguiente método en **SegmentSDF.hsls**:

```
1 float projection(float2 a, float2 b, float2 c)
2 {
3     return 0;
4 }
```

Siguiendo la ecuación 4.1.ñ, observaremos que existen dos sustracciones presentes: $a - b$ y $c - b$. Por lo tanto, el primer paso será declarar e inicializar estos vectores bidimensionales dentro del campo del método **projection()**.

```

1 float projection(float2 a, float2 b, float2 c)
2 {
3     float2 cb = c - b;
4     float2 ab = a - b;
5
6     return 0;
7 }
```

A continuación, podemos retornar el resultado de la operación, el cual corresponde al producto punto entre los vectores **cb** y **ab**, dividido por la magnitud del primero al cuadrado. Es decir:

```

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 }
```

Si prestamos atención al ejercicio anterior, línea 6, observaremos que la segunda operación de la función corresponde a la longitud del vector **cb** al cuadrado. Sin embargo, cabe destacar que,

$$\text{length}(cb)^2 = \text{dot}(cb, cb)$$

(4.2.f)

Esto se debe a que la función **length()** incluye una raíz cuadrada, y ya sabemos que cuando una raíz cuadrada es elevada al cuadrado, matemáticamente esta se elimina, quedando el resultado aislado en la operación.

```

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

Por lo tanto, podemos simplificar el valor de retorno en el método **projection()** de la siguiente manera:

```

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) / dot(cb, cb);
7 }
```

Como vimos en la sección anterior, la proyección por sí sola generará un segmento infinito que pasará entre los dos puntos mencionados anteriormente. Por lo tanto, será necesario tanto limitar el segmento como definir un segundo punto p_2 para determinar la distancia entre las coordenadas UV y el segmento en sí.

Continuaremos declarando un nuevo método vacío al cual denominaremos **segment_sd_float()**. Este método se encargará de limitar el segmento y determinar la posición de todos los puntos que se encuentran entre los puntos p_0 y p_1 .

```

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
14     Out = 0;
15 }
```

Del ejemplo anterior, como se puede observar en la línea número 11, se ha declarado una nueva variable denominada **h**, la cual incluye la proyección entre las coordenadas UV y los puntos **p0** y **p1**. Posteriormente, el valor de **h** se ha limitado a un rango entre [0.0 : 1.0], es decir, el rango entre los puntos mencionados, para evitar valores infinitos.

Por último, faltaría calcular la distancia entre las coordenadas UV y todos los puntos que se generan entre los puntos **p0** y **p1**. Para ello, podemos emplear la ecuación 4.1.0, descrita en la sección anterior, es decir:

```

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 }
```

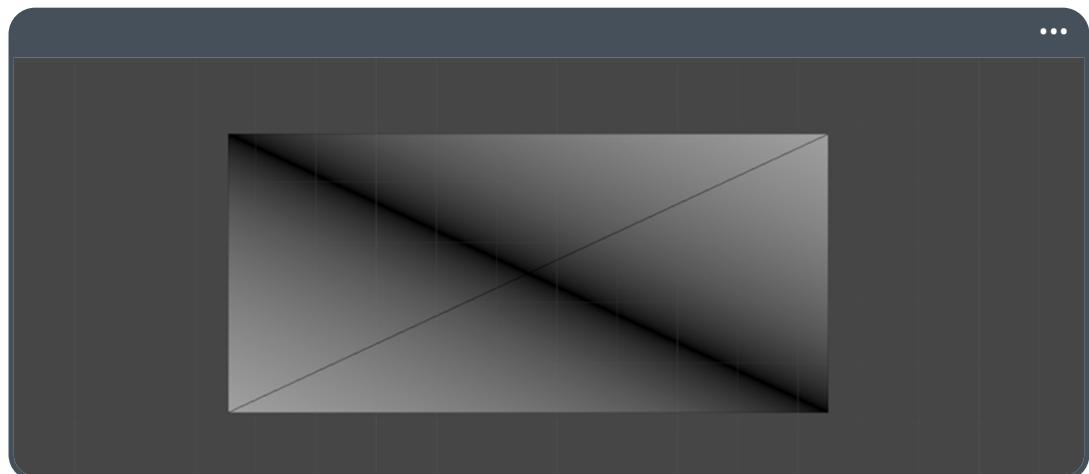
Podríamos concluir que el segmento de distancia con signo ya está listo hasta este punto. Ahora simplemente debemos visualizarlo. Para ello, iremos a nuestro shader **Segment SDF** y agregaremos la siguiente línea de código en la etapa de fragmento **frag()**, como se muestra a continuación:

```

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

```

Como podemos observar en la línea de código 128, se ha declarado e inicializado un nuevo vector de cuatro dimensiones denominado **segment**, el cual se utiliza como salida en la función **segment_sd_float()**. Posteriormente, en la siguiente línea, se han incluido dos puntos constantes en la función que poseen los valores [0.0, 1.0] para el primer punto, y [1.0, 0.0] para el segundo. Si utilizamos el vector **segment** como valor de retorno para cada píxel, obtendremos un resultado gráfico como el siguiente:



(4.2.g)

Cabe destacar que, en la imagen anterior, se ha removido el **Blending Mode** del shader para visualizar el segmento de mejor manera. Como podemos observar, el segmento se está proyectado correctamente desde un punto de vista matemático. No obstante, será necesario agregar algunas propiedades en nuestro shader para editar su comportamiento y visualización. Para ello, haremos lo siguiente:

Dentro del campo de las propiedades:

- Agregaremos un vector de cuatro dimensiones, donde los dos primeros componentes **xy** corresponderán al primer punto, y los restantes **zw** serán el segundo punto.
- Luego, incluiremos una variable que nos ayudará a determinar el radio del segmento.
- Finalmente, declararemos una última variable para definir qué tan suavizados serán los bordes del segmento.

```

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

```

Por defecto, la propiedad **_Color** viene incluida en el shader. No obstante, **_Points**, **_Radius** y **_Smooth** han sido declaradas para definir valores dinámicos dentro del shader. Cabe recordar que, de igual manera, debemos declarar estas propiedades dentro del **Pass**, siguiendo su tipo de dato, es decir:

```

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

```

Una vez declaradas, podemos emplear estas propiedades para la definición del segmento.

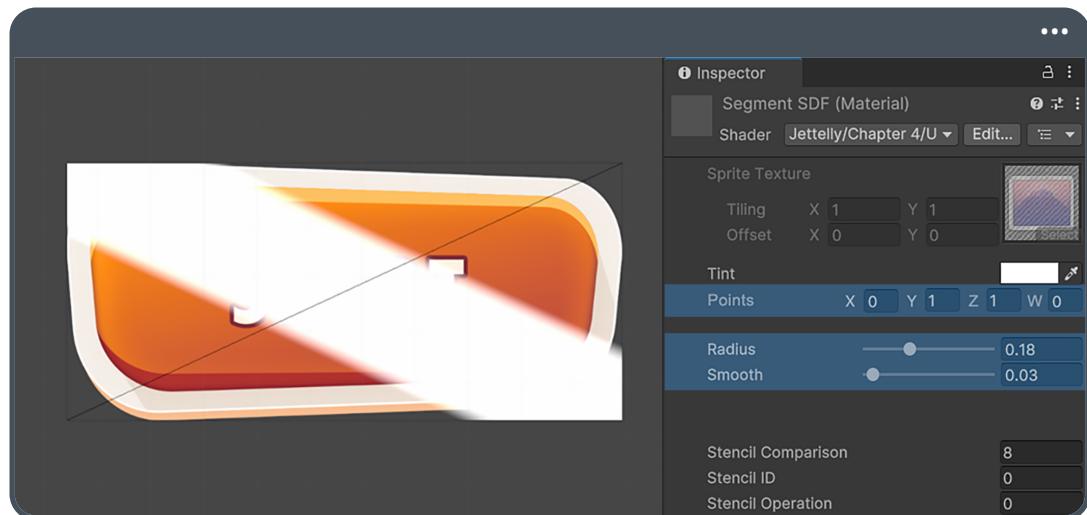
```

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);

```

Como se puede observar en el ejemplo anterior, se han declarado dos nuevos vectores bidimensionales, **p0** y **p1**, los cuales se han inicializado utilizando los componentes del vector **_Points** (líneas 133 y 134). Posteriormente, estos dos puntos han reemplazado a los puntos constantes en el método **segment_sd_float()**, incluidos con anterioridad a modo de exemplificación. Este proceso nos va a permitir modificar la orientación del segmento al cambiar la posición de los puntos directamente desde el Inspector. Luego, se han declarado e inicializado dos nuevas variables denominadas **s** y **r** (líneas 136 y 137). Por su parte, **s** nos permitirá modificar el suavizado de los bordes del segmento, mientras que **r** va a permitir modificar el grosor de este.

Si guardamos los cambios y volvemos a Unity, podremos modificar el cuerpo del segmento.

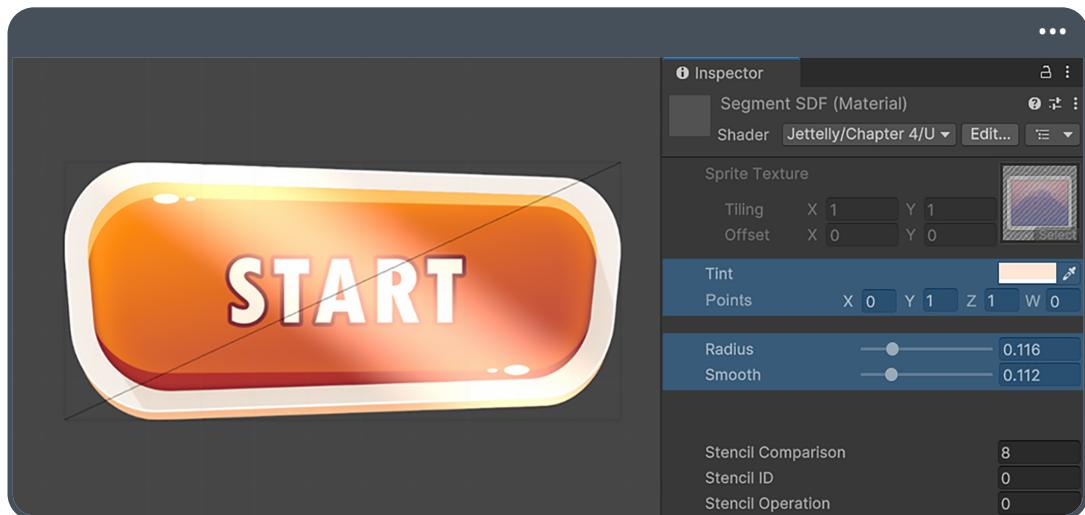


(4.2.h)

Un detalle que podemos identificar de la implementación actual es que los bordes del segmento exceden los bordes de la imagen de UI. Por ende, será necesario limitar su representación gráfica. Esto podemos lograrlo fácilmente multiplicando el canal alpha de la entrada de color por el segmento, como se muestra a continuación:

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

Cabe recordar que el canal alpha representa la transparencia u opacidad de una imagen. Por lo tanto, al multiplicar el componente A por el segmento, el resultado que obtenemos es una textura completamente en escala de grises, donde el rango de [1.0 : 0.0] representa el valor lumínico de cada píxel. Es decir, el color negro es igual a cien por ciento transparencias, mientras que el blanco es cero por ciento.



(4.2.i)

Si guardamos los cambios y regresamos a Unity, observaremos que el segmento se mantendrá dentro del área de la imagen independiente de la posición de los puntos.

4.3 Análisis de la estructura de un pentágono SDF.

Siguiendo con el estudio de las funciones de distancia con signo, en esta sección analizaremos un método que nos permitirá dibujar un pentágono en coordenadas UV, desarrollado por Iñigo Quilez, quien, al momento de la publicación de este libro, es ingeniero principal de investigación en Adobe, especializado en gráficos por computadora.

Para comenzar, examinemos el siguiente fragmento de código, escrito en OpenGL Shading Language (o GLSL en sus siglas en inglés):

```

1 float sdPentagon( in vec2 p, in float r )
2 {
3     const vec3 k = vec3(0.809016994, 0.587785252, 0.726542528);
4     p.y = -p.y;
5     p.x = abs(p.x);
6     p -= 2.0 * min(dot(vec2(-k.x, k.y), p), 0.0) * vec2(-k.x, k.y);
7     p -= 2.0 * min(dot(vec2(k.x, k.y), p), 0.0) * vec2(k.x, k.y);
8     p -= vec2(clamp(p.x, -r * k.z, r * k.z), r);
9     return length(p) * sign(p.y);
10 }
```

El código completo de este ejemplo puede consultarse en el siguiente enlace:

➤ <https://www.shadertoy.com/view/lIVyWW>

Es importante destacar que no existe una diferencia significativa entre GLSL y HLSL en lo que respecta a sus tipos de datos. Por ejemplo, si observamos la línea de código número 3, el vector tridimensional **vec3** en GLSL se traduce como **float3** en HLSL. Lo mismo ocurre con el vector bidimensional **vec2** de las líneas 6, 7 y 8. Por lo tanto, el mismo código en Unity se vería de la siguiente manera:

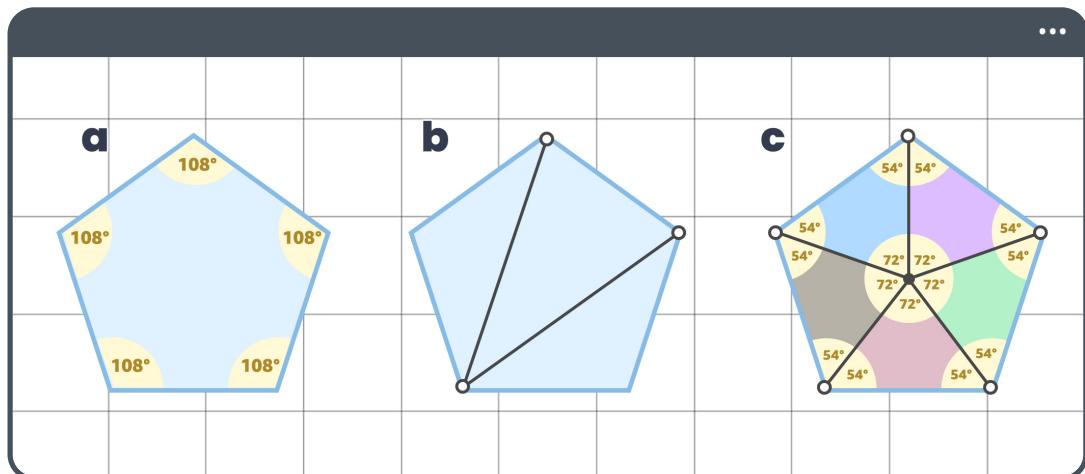
```

1 void pentagon_sd_float(in float2 uv, in float r, out float4 Out)
2 {
3     const float3 k = float3(0.809016994, 0.587785252, 0.726542528);
4     uv -= 0.5;
5     uv.y = -uv.y;
6     uv.x = abs(uv.x);
7     uv -= 2.0 * min(dot(float2(-k.x, k.y), uv), 0.0) * float2(-k.x, k.y);
8     uv -= 2.0 * min(dot(float2(k.x, k.y), uv), 0.0) * float2(k.x, k.y);
9     uv -= float2(clamp(uv.x,-r * k.z, r * k.z),r);
10    Out = length(uv) * sign(uv.y);
11 }

```

En este ejemplo, se han ajustado algunas nomenclaturas para mejorar la legibilidad del código. Además, se ha configurado el método como **void** para hacerlo compatible con el nodo **Custom Function**. En la línea 4, se ha restado 0.5 de las coordenadas UV para centrar la figura en un Quad. Ahora bien, surge la pregunta: ¿qué ocurre dentro de este método en general? Para responder a esta interrogante, primero debemos comprender cómo se forma un pentágono desde un punto de vista trigonométrico.

El pentágono regular es una figura geométrica de cinco lados de igual longitud y ángulos internos iguales. Es decir,



(4.3.a)

Los ángulos internos de un pentágono regular miden 108° cada uno. Podemos determinar esto fácilmente utilizando triángulos. Según un principio fundamental de la geometría euclíadiana, los ángulos internos de un triángulo siempre suman 180° . Si observamos la figura **b** en la imagen 4.3.a, notaremos que un pentágono posee tres triángulos internos. Por lo tanto:

$$180^\circ * 3 = 540^\circ$$

(4.3.b)

Si dividimos el resultado de esta multiplicación entre los cinco lados del pentágono, obtenemos el ángulo correspondiente a cada lado.

$$\frac{540^\circ}{5} = 108^\circ$$

(4.3.c)

Desde otra perspectiva, si consideramos la figura **c** de la imagen 4.3.a, al dividir un círculo completo en cinco partes iguales, obtenemos:

$$\frac{360^\circ}{5} = 72^\circ$$

(4.3.d)

Además, podemos dividir el ángulo interno para obtener:

$$\frac{108^\circ}{2} = 54^\circ$$

(4.3.e)

Estos ángulos son cruciales para construir nuestro pentágono, ya que nos permiten determinar con precisión las medidas necesarias para trazar sus lados y definir sus vértices. Al saber que cada ángulo interno mide 108° , podemos asegurarnos de que todas las intersecciones de los lados se realicen correctamente, manteniendo la forma regular del pentágono.

El cálculo del ángulo de 72° en la figura **c** es esencial para entender la relación entre los ángulos internos y los ángulos externos del pentágono. Cada uno de los ángulos externos, formados al extender los lados del pentágono, mide 72° , completando los ángulos internos de 108° para sumar 180° en total.

Por ejemplo, si observamos la línea número 3 de la función **sdPentagon()**, notaremos que se ha declarado e inicializado un vector tridimensional denominado **k**, que contiene las constantes [0.809016994, 0.587785252, 0.726542528]. Estos valores, expresados en radianes, se obtienen a partir de la siguiente operación matemática:

$$\frac{\pi}{5} = 0.628318530 = 36^\circ$$

(4.3.f)

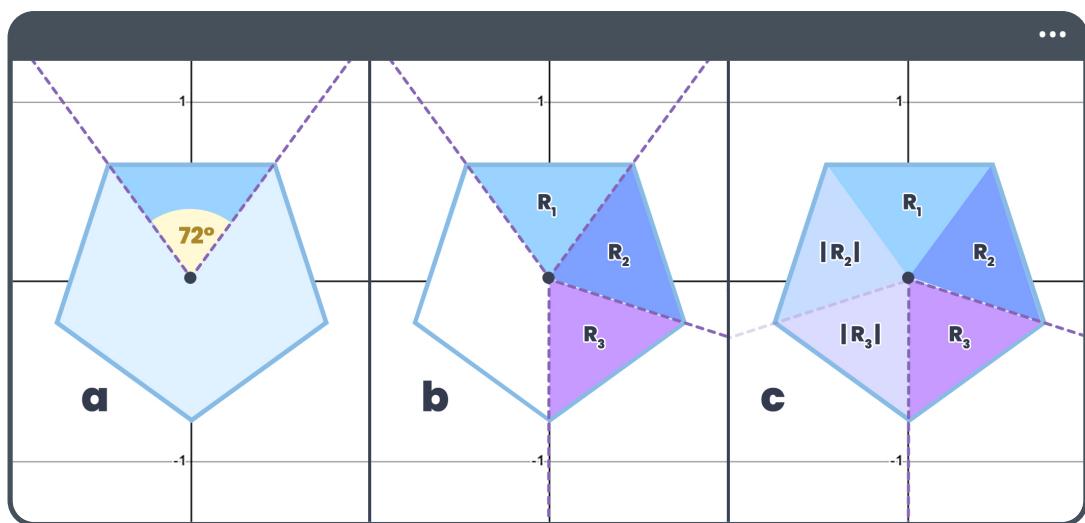
Donde,

$$\begin{aligned} k.x &= \cos(0.628318530) = 0.809016994 \\ k.y &= \sin(0.628318530) = 0.587785282 \\ k.z &= \tan(0.628318530) = 0.726654252 \end{aligned}$$

(4.3.g)

Si observamos la relación en la referencia 4.3.f, notaremos que 36° corresponde a la mitad de 72° . Sin embargo, ¿cómo determinamos la relevancia de $\frac{\pi}{5}$ dentro del contexto de un pentágono SDF? Para ello, debemos recordar las reflexiones que analizamos en la sección 2.3 del capítulo 2.

Dado que el pentágono tiene cinco lados iguales, no es necesario calcular su forma completa; en su lugar, podemos definir sólo una región y luego aplicar transformaciones. Considerando sólo una región de la forma completa, ¿cómo podríamos calcular la distancia entre todos los puntos fuera del pentágono y su borde? Para responder a esta pregunta, prestemos atención a la siguiente referencia:



(4.3.h <https://www.desmos.com/calculator/1jqqip75ke>)

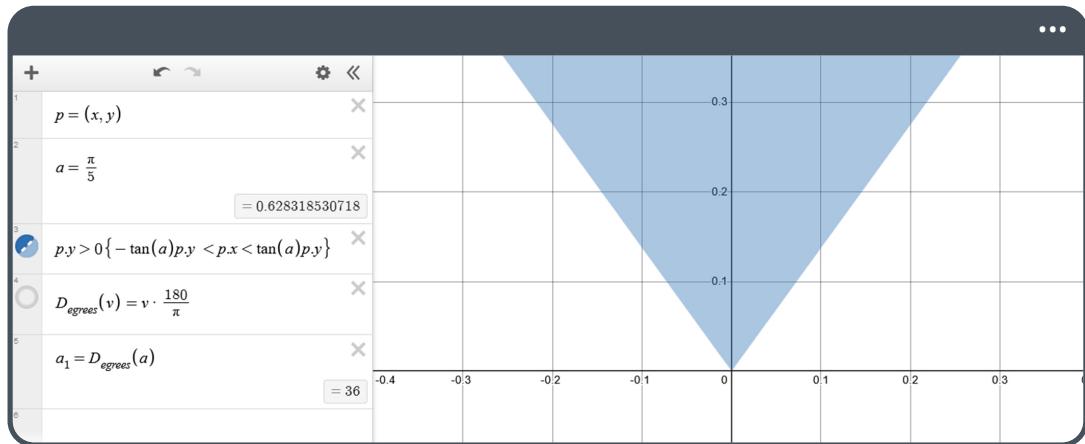
Cada figura de la imagen anterior representa un paso necesario para construir el método `sdPentagon()`, el cual se divide en tres regiones R_1 , R_2 y R_3 . La primera región R_1 sirve como área de referencia para calcular las siguientes regiones mediante reflexión y transformación. Por lo tanto, nos enfocaremos en calcular directamente la distancia al pentágono en esta primera región.

Considerando $a = \frac{\pi}{5}$ y basándonos en el código del método, podemos determinar que todos los puntos p (o coordenadas UV) de la primera región se encuentran dentro de un margen definido por la siguiente condición:

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

(4.3.i)

La ecuación 4.3.k se puede visualizar de la siguiente manera en el plano cartesiano:



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

A partir del ejercicio anterior, podemos comenzar a identificar las variables incluidas en el método **sdPentagon()**. Por ejemplo:

$$p = \text{in vec2 } p$$

(4.3.k)

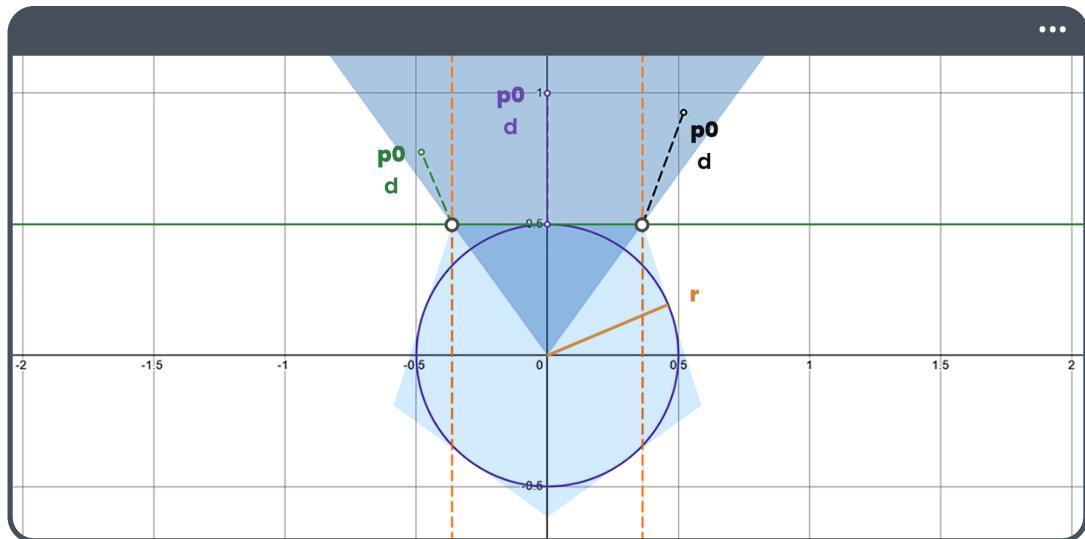
O también:

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

(4.3.l)

La ecuación 4.3.k define los márgenes dentro de los cuales se encuentra la primera región R_1 . Sin embargo, aún no hemos calculado la distancia entre todos los puntos de la primera región y el borde del pentágono. Para determinar esta distancia, primero debemos establecer cómo realizaremos este cálculo. Por lo tanto, es necesario definir en primer lugar:

- Un radio r para determinar el tamaño del pentágono.
- La distancia d entre todos los puntos de la región R_1 y los vértices más cercanos a cada punto.



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

Al intentar determinar la distancia entre un vértice y su punto más cercano, se generan naturalmente tres zonas distintas, lo que da lugar a parámetros o argumentos para cada caso. Por ejemplo, para el caso en que $p_x > r \tan(a)$ podemos calcular la distancia utilizando los siguientes valores: considerando un punto aleatorio $p_n = (0.6, 1.0)$ y su vértice más cercano $v_0 = (r \tan(a), r)$, obtenemos:

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

(4.3.n)

Esto equivale a decir:

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

(4.3.n)

Esta última es una manera más concisa de expresar la distancia euclíadiana entre dos puntos.

Para el caso en que $p_x < r \tan(a)$, considerando un punto aleatorio $p_n = (-0.5, 0.7)$ y su vértice más cercano $v_1 = (-r \tan(a), r)$. En este caso, la distancia se calcula como:

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

(4.3.o)

También debemos considerar los casos en que $p_x < 0$, ya que los puntos p_n , cercanos al centro eventualmente estarán más próximos a los vértices de la derecha v_0 o izquierda v_1 , especialmente cuando el radio tienda a cero. Si observamos la línea número 8 del método **sdPentagon()**, notaremos que se utiliza la función **clamp()**, que limita el rango de una variable (en este caso p_x) a un valor específico, dando lugar a la siguiente expresión:

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

(4.3.p)

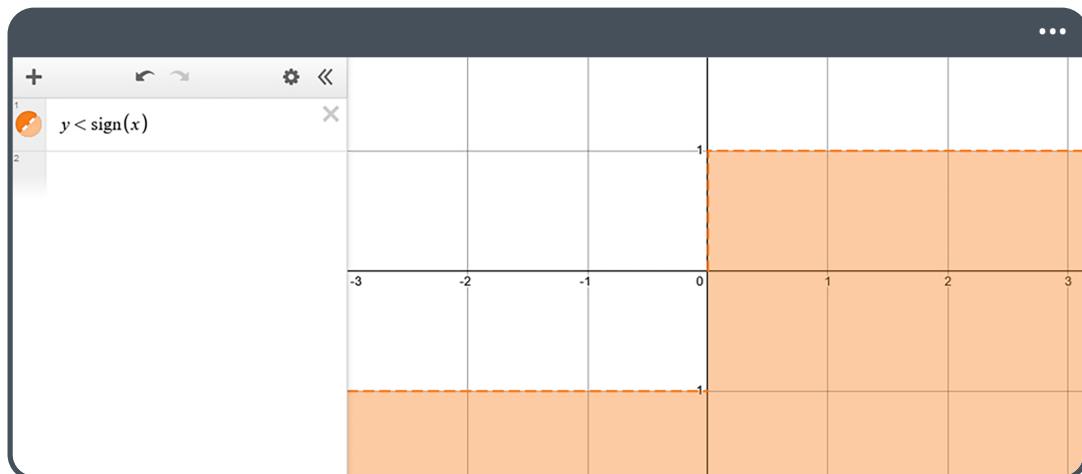
Esto se traduce en el siguiente código:

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

Dado que,

```
1 float clamp(float x, float a, float b)
2 {
3     return max(a, min(b, x));
4 }
```

Posteriormente, la función **sign()** se añade en el método para devolver 1, -1 o 0 dependiendo de si el valor de la coordenada *y* es mayor, menor o igual a cero respectivamente. En el método **sdPentagon()**, esta función se emplea para determinar si los puntos se encuentran dentro o fuera del cuerpo del pentágono.

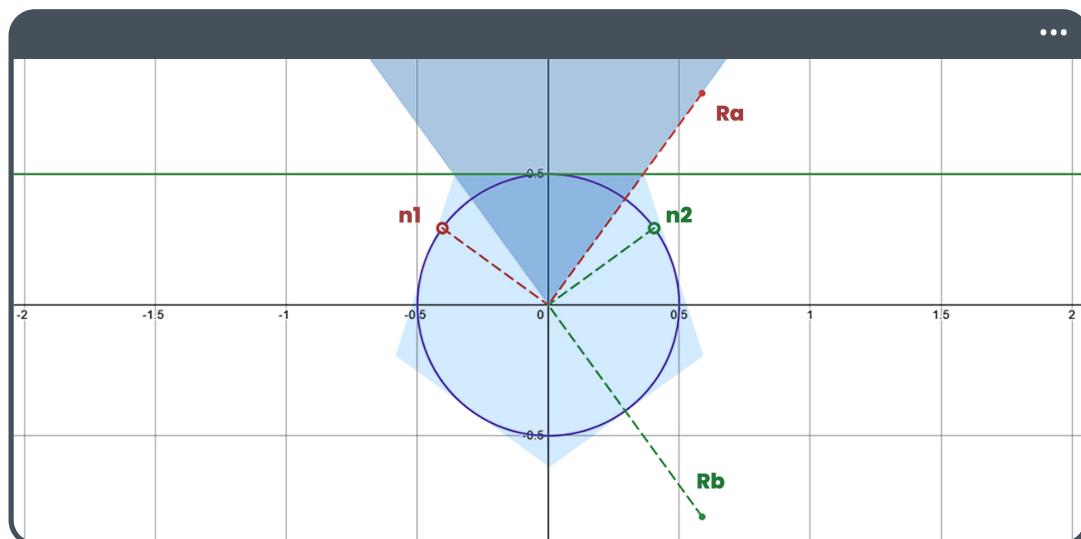


(4.3.q <https://www.desmos.com/calculator/hiur1ldkgd>)

Esto nos permite definir con precisión las áreas interiores y exteriores del pentágono, facilitando la creación de la forma al basarnos en la proximidad de los bordes y en la orientación relativa de los puntos con respecto al centro de la figura.

4.4 Reflexiones y transformaciones del pentágono SDF.

Hasta este punto, hemos explorado las funciones matemáticas involucradas en la implementación de la primera región R_1 del pentágono en `sdPentagon()`. A continuación, debemos definir las regiones R_2 y R_3 para completar la figura. Para ello, será necesario establecer los ejes de reflexión.



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

Como se puede observar en la imagen 4.4.a, se han definido dos ejes de reflexión R_a y R_b , así como dos normales n_1 y n_2 , que son perpendiculares a sus respectivas reflexiones. El eje R_a se ha determinado calculando el seno y coseno del ángulo a para cada componente, es decir:

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

(4.4.b)

Mientras que su normal se define como:

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

(4.4.c)

De manera similar, el eje R_b se define como:

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

(4.4.d)

Y su normal es:

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

(4.4.e)

Es importante destacar que estas reflexiones nos permitirán calcular únicamente las regiones que se encuentran en el lado positivo de la coordenada **p.x**. Por lo tanto, cuando los valores de **p.x** sean negativos, será necesario aplicar el valor absoluto a esta coordenada para obtener el cuerpo completo del pentágono, como se muestra en la línea número 5 del método **sdPentagon()**.

A continuación, transformaremos los puntos comenzando con la segunda región R_2 . Esta región requiere solo una reflexión en torno a R_a . Esto significa que, para los puntos pertenecientes a R_2 , la fórmula de transformación es:

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

(4.4.f)

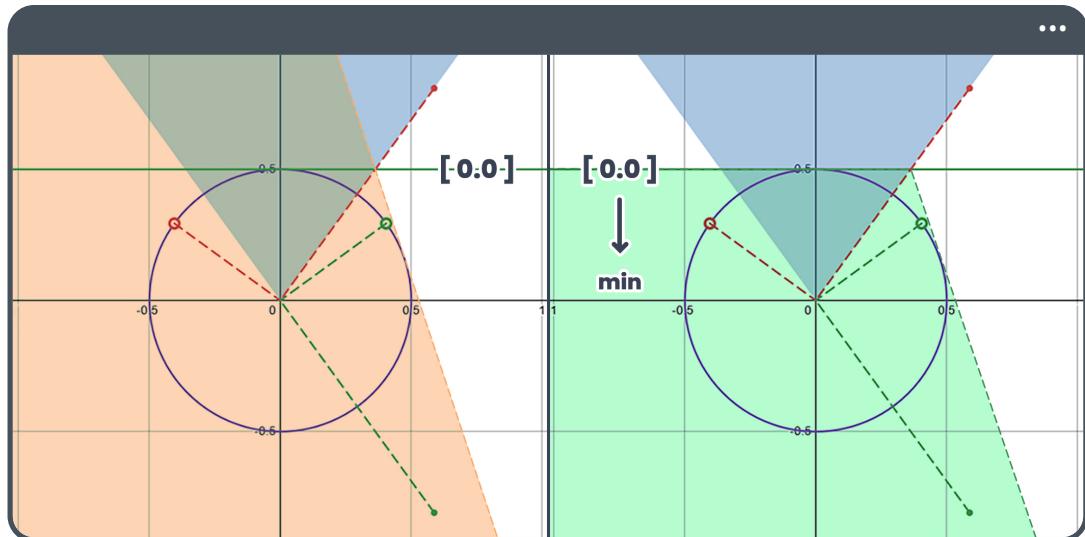
Lo que se traduce en el siguiente código:

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

Sin embargo, al observar el método original, notamos que se ha incluido la función `min()`. Esto se debe a que, dado que la reflexión es infinita, al reflejar en R_a , algunos puntos quedan fuera del cuerpo del pentágono. Por lo tanto, es necesario limitar esta reflexión según los valores deseados.

Dado que el borde de la primera región se define de manera explícita mediante $y = r$, la coordenada y se utiliza como el valor límite para la reflexión, es decir, el punto 0.0.

```
6 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>)

El mismo problema surge al calcular la tercera región R_3 en torno a la segunda reflexión.

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

(4.4.h)

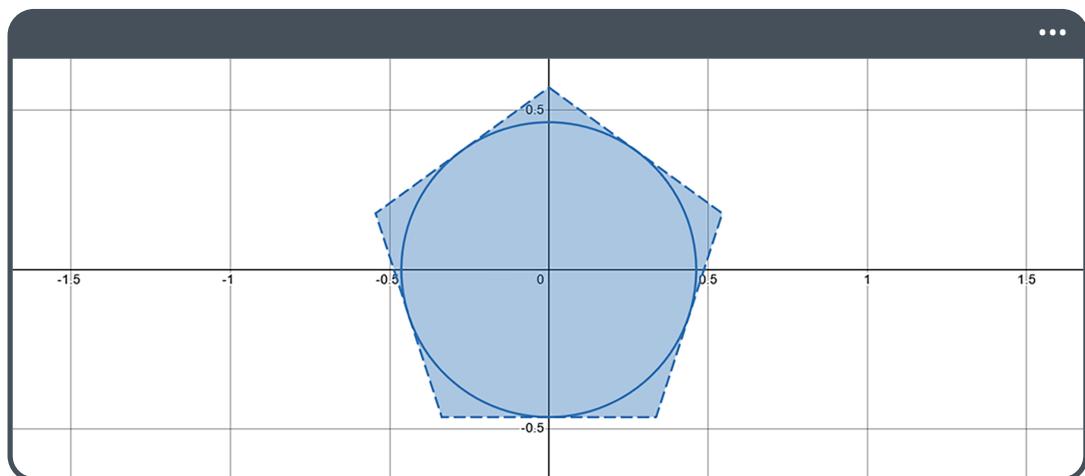
Lo que se traduce al siguiente código:

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

Si la transformación es infinita, también será necesario limitarla en ciertos puntos, ya que, de lo contrario, estaríamos calculando la distancia de puntos que están fuera del cuerpo del pentágono. Para ello, simplemente agregamos la función **min()** sobre la expresión, quedando de la siguiente manera:

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

Podríamos decir que, hasta este punto nuestra figura está lista. Sin embargo, los cálculos realizados previamente, únicamente aplican para aquellos puntos donde $p_x > 0$. Por lo tanto, siguiendo la línea 5 del método **sdPentagon()**, nos aseguraremos de aplicar el valor absoluto sobre **p.x**. Además, voltearemos la figura haciendo negativa la coordenada **p.y**.



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

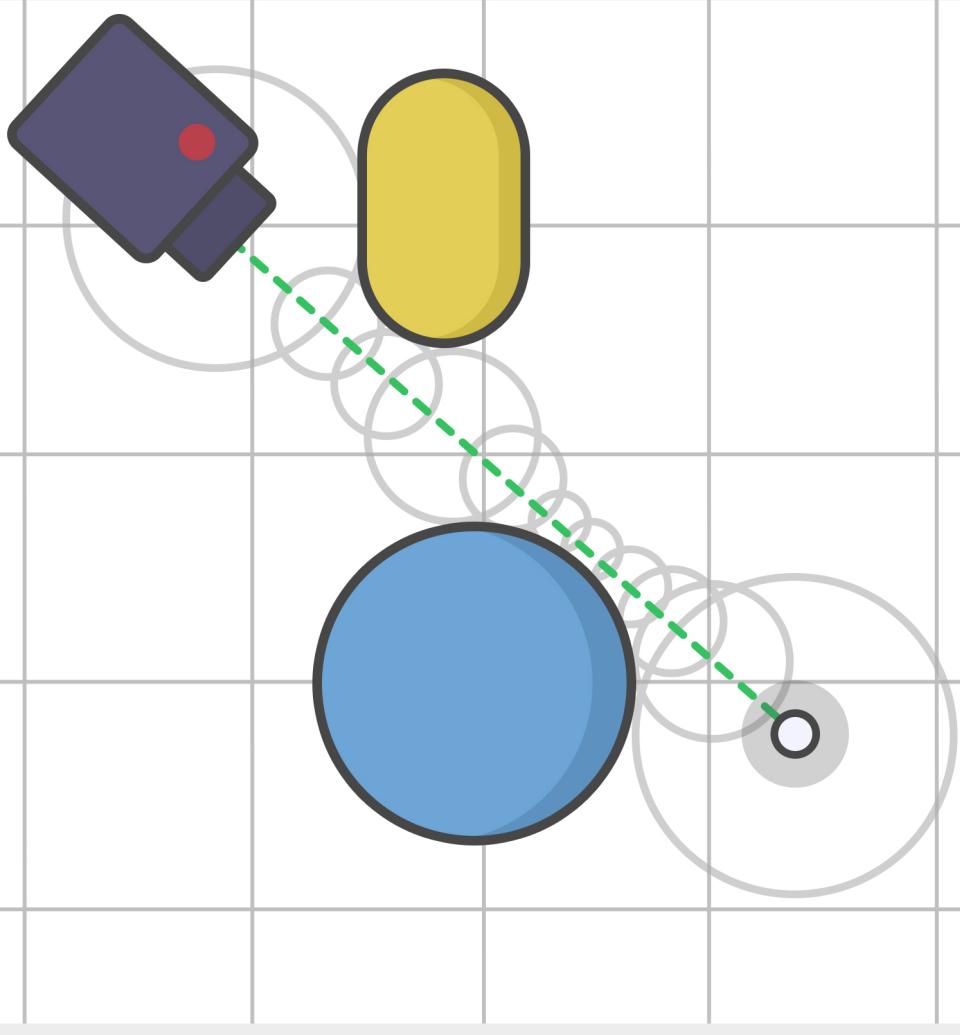
En conclusión, tras haber analizado cada función matemática en el método **sdPentagon()**, podemos determinar lo siguiente:

- Línea 1: Los argumentos **p** y **r** se refieren a las coordenadas UV y al radio respectivamente.
- Línea 3: La constante **k**, un vector tridimensional, contiene los cálculos trigonométricos de **cos**, **sin** y **tan** para cada componente del ángulo.
- Línea 4: Se invierte el componente **p.y** del punto **p** para que el pentágono se oriente hacia arriba.
- Línea 5: Se toma el valor absoluto de **p.x** para reflejar los puntos cuando $p_x < 0$.

- Línea 6: Se define implícitamente la segunda región R_2 mediante una reflexión.
- Línea 7: Se define implícitamente la tercera región R_3 mediante una segunda reflexión.
- Línea 8: Se delimita explícitamente la primera región R_1 utilizando la función `clamp()`.
- Línea 9: Se calcula la distancia entre todos los puntos fuera del pentágono y su borde determinando si están dentro o fuera de la figura.

Resumen del capítulo.

- En este capítulo, exploramos la construcción de un segmento y un pentágono utilizando funciones de distancia con signo (SDF) en el contexto de gráficos computacionales. Comenzamos analizando la estructura matemática de un segmento, comparándolo con una función lineal del tipo $mx + b$, y aplicamos este conocimiento para generar un efecto de una imagen de UI en Unity, útil para producción.
- En la segunda sección, revisamos las funciones asociadas al cuerpo del pentágono. Analizamos la estructura matemática de la primera región R_1 , calculando la distancia desde puntos internos hasta los bordes de la figura. Luego, definimos las regiones R_2 y R_3 mediante reflexiones en los ejes R_a y R_b , utilizando sus normales para ajustar las coordenadas de los puntos reflejados. Para asegurar la precisión, limitamos estas reflexiones con la función `min()` para evitar calcular distancias fuera del cuerpo del pentágono. Finalmente, aplicamos el valor absoluto sobre la coordenada `p.x` y ajustamos la coordenada `p.y` para obtener la forma completa y orientación del pentágono.



Capítulo 5

Figuras tridimensionales y renderizado.

En este capítulo, profundizarás en el uso de las funciones de distancia con signo para generar geometrías tridimensionales más complejas, centrándote en la creación de figuras como esferas y cápsulas. Estas formas, o “primitivas”, constituyen la base de la mayoría de los modelos 3D y te ayudarán a comprender mejor las operaciones matemáticas necesarias para diseñar tus propias figuras. Asimismo, explorarás métodos para combinar y modificar estas primitivas, lo que te permitirá comenzar a esculpir formas mucho más elaboradas y personalizadas.

También abordarás el cálculo de las normales en geometrías definidas por funciones de distancia con signo, un elemento fundamental para la iluminación. Obtener normales precisas es esencial para lograr efectos de luz realistas y dar mayor credibilidad a tus proyectos. Posteriormente, te introducirás al método de Ray Marching, imprescindible para renderizar estas estructuras de forma eficiente.

A lo largo de este capítulo, trabajarás con ejemplos prácticos e implementaciones en Unity, resaltando la importancia de las matemáticas en cada paso. Nuestro objetivo no es tan solo enriquecer tus proyectos dentro de este motor, sino también que comprendas cómo aplicar estos conceptos en otras plataformas y contextos de programación gráfica. Al finalizar, contarás con las habilidades necesarias para diseñar y desarrollar formas procedurales de alta complejidad, aprovechando la fusión entre las matemáticas y la programación gráfica para llevar tus efectos visuales a un nuevo nivel creativo.

5.1 Añadiendo una tercera dimensión.

Hasta este punto, hemos trabajado principalmente con figuras bidimensionales para comprender los fundamentos de distintas funciones matemáticas, incluidas las lineales, trigonométricas y de distancia con signo. Sin embargo, el siguiente paso natural es expandir este enfoque al espacio tridimensional, donde podemos generar geometrías más complejas y realistas.

Para iniciar nuestra transición del espacio bidimensional al tridimensional, revisaremos la fórmula matemática que define un círculo en 2D y luego mostraremos cómo se extiende para representar una esfera en 3D:

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

(5.1.a)

La función de distancia con signo más común para un círculo se define como la distancia de cualquier punto en el plano xy al centro del círculo, menos el radio r . Esta función arroja valores negativos para puntos dentro del círculo, cero en la circunferencia y valores positivos fuera de ella. Esta distinción de signos es fundamental para identificar la posición relativa de los puntos respecto al círculo.

Podemos simplificar la fórmula 5.1.a empleando el producto escalar:

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

(5.1.b)

O describirla en términos de un vector bidimensional v :

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

(5.1.c)

En HLSL, esta función de distancia para un círculo se puede implementar como:

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

...

Aquí, **uv** es un vector bidimensional que representa las coordenadas en el plano, y **r** corresponde al radio del círculo. La función **length()** simplifica el cálculo de distancia desde un punto en el plano al centro.

Para transformar este círculo en una esfera, extendemos el concepto al espacio tridimensional. Puesto que tanto el círculo como la esfera mantienen la propiedad de que sus puntos se encuentran a la misma distancia de un centro, simplemente añadimos una tercera coordenada **z**.

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

(5.1.d)

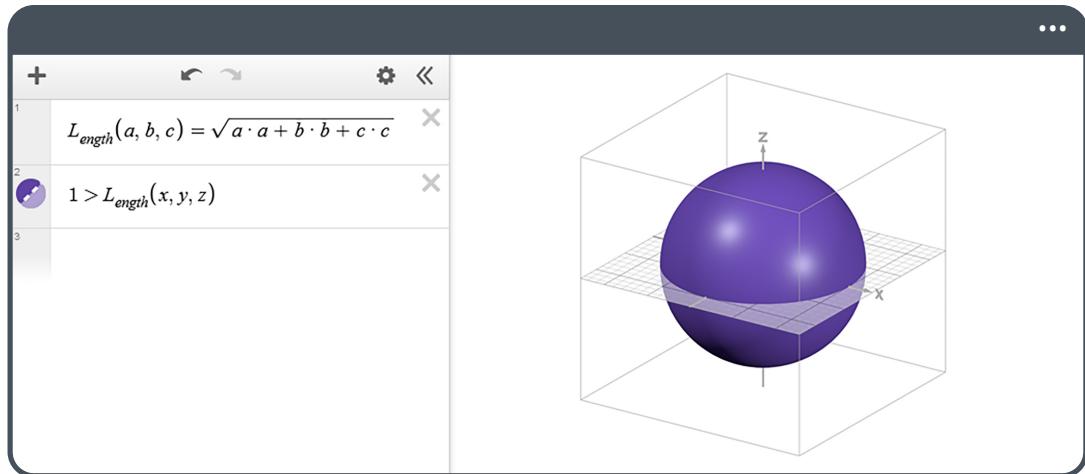
En HLSL, esta expresión se traduce como:

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

...

Donde **p** es un vector tridimensional que representa la posición de un punto en el espacio 3D. Al igual que con el círculo, **length()** calcula la distancia de **p** al origen.

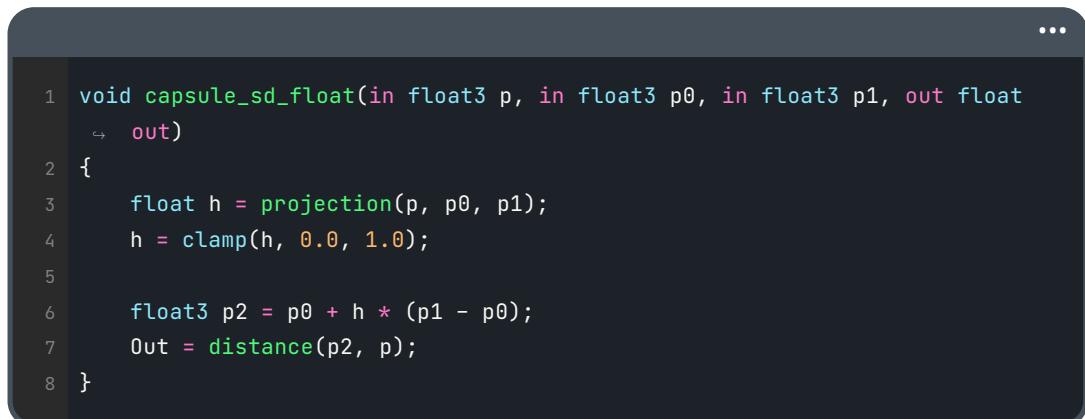
Al comparar las funciones para el círculo y la esfera, observamos que ambas son estructuralmente idénticas; solo varía la dimensionalidad del vector que utilizamos. Este patrón nos permite adaptar fácilmente nuestras funciones de distancia a espacios de mayor dimensión, haciendo que nuestras implementaciones resulten más versátiles y potentes.



(5.1.e <https://www.desmos.com/3d/bitqlfpd2>)

Podemos aplicar una analogía similar al convertir un círculo en una esfera para extenderla al de una cápsula, la cual representa la extensión tridimensional de un segmento. En el capítulo anterior, trabajamos con la implementación de un segmento mientras desarrollamos efectos para nuestras imágenes de UI. Para transformar dicha forma geométrica en una cápsula, basta con añadir una tercera dimensión a nuestras ecuaciones.

Si tomamos la función **segment_sd_float()** definida en la sección 4.2 y añadimos un tercer componente, la definición quedaría de la siguiente manera:

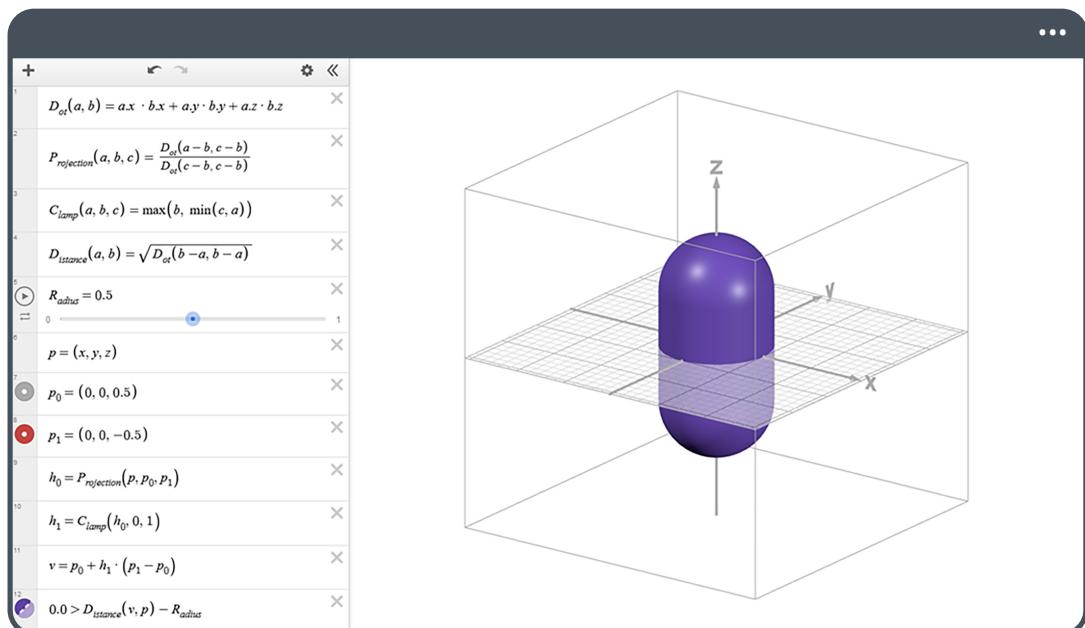


En este código, **p** representa el punto en el espacio tridimensional donde evaluamos la función de distancia, mientras que **p0** y **p1** son los extremos del segmento en 3D que

define el eje central de la cápsula. El método **projection()** (ampliado a una tercera dimensión) calcula la proyección del punto **p** sobre el segmento definido por **p0** y **p1**, y **clamp()** garantiza que esta proyección se mantenga dentro del rango del segmento.

La variable **p2** determina el punto más cercano al segmento desde la perspectiva de **p**, y finalmente, **distance()** calcula la distancia entre dichos puntos, valor fundamental para definir el “volumen” de la cápsula alrededor del segmento.

Al incorporar una tercera dimensión en nuestras ecuaciones, no solo agregamos profundidad a nuestras formas, sino que también abrimos la puerta a una amplia gama de posibilidades en la generación de geometrías procedurales. Con esta base, podremos combinar y modificar formas básicas de manera flexible, impulsando la creación de efectos y estructuras más avanzadas.



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

Al observar la referencia 5.1.f, notamos que se ha definido una cápsula utilizando las mismas operaciones matemáticas de una función de distancia con signo, implementadas en Desmos para su visualización. Esto nos lleva a reflexionar sobre cómo renderizar estas

formas mediante shaders en Unity y la manera de integrarlas de manera coherente en nuestro entorno gráfico.

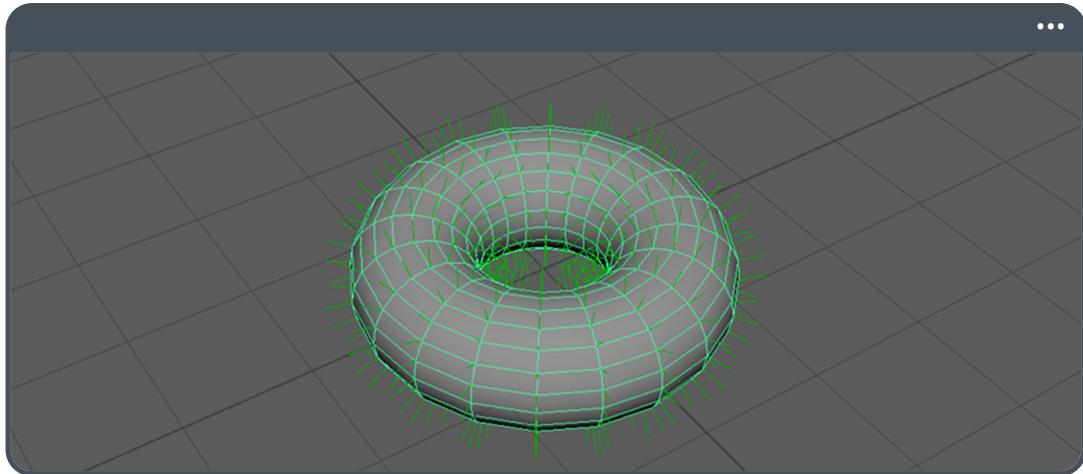
Para renderizar adecuadamente estas figuras tridimensionales, debemos tener en cuenta al menos tres factores esenciales:

- Normales de la superficie: Determinan cómo la luz interactúa con la superficie y, en consecuencia, afectan la apariencia final de la figura.
- Iluminación: Define la distribución de la luz a lo largo de la figura, resaltando sus contornos y volúmenes.
- Renderización: Abarca el proceso de dibujar la figura en la pantalla. En esta oportunidad, emplearemos la técnica de **Ray Marching**.

Las normales desempeñan un papel crítico en el cálculo preciso de la iluminación. Estos vectores perpendiculares a la superficie indican la orientación de cada punto, lo cual permite simular con precisión efectos de luz como reflejos y sombras. Si las normales no se calculan correctamente, las figuras pueden lucir planas o mostrar reflejos imprecisos, impactando de manera negativa su realismo y apariencia.

Respecto a la iluminación, esta resulta esencial para acentuar los detalles y realzar la tridimensionalidad de nuestras figuras. Al emplear modelos de iluminación, podemos mejorar la percepción de volúmenes y materialidad de las formas procedurales. Tanto la iluminación difusa como la especular nos ayudan a simular cómo la luz interactúa con las distintas superficies, añadiendo mayor verosimilitud a la escena.

Con estos aspectos en mente: normales, iluminación y renderizado, estaremos listos para dar el siguiente paso: implementar en Unity todo lo aprendido sobre la generación y representación de figuras tridimensionales basadas en funciones de distancia con signo.



(5.1.g Vector normal en un toroide)

Como podemos observar en la imagen 5.1.g, el toroide exhibe sus líneas normales (representadas en color verde) perfectamente alineadas con la orientación de cada vértice, lo que se traduce en una distribución armoniosa de la iluminación a lo largo de la superficie. Sin embargo, existen técnicas de iluminación que requieren un control específico de la orientación de las normales para lograr efectos determinados, como sucede con la iluminación estilo “toon” o Cel shading.

La imagen de referencia proviene de Maya, un software de modelado 3D ampliamente utilizado en la industria. En nuestro caso, no contamos con vértices para extraer o modificar las normales, pues nuestras figuras se generan a partir de funciones de distancia con signo. Por ello, debemos calcular y orientar las normales mediante procedimientos matemáticos, asegurando que reflejen con precisión la geometría implícita de cada figura.

5.2 Normales en funciones de distancia con signo.

El cálculo de las normales en nuestras figuras procedurales requiere entender el concepto de gradiente. En el caso de una función de distancia con signo, el gradiente es un vector que indica la dirección de mayor crecimiento de dicha función. Para cada punto de la superficie, este vector resulta perpendicular, definiendo la normal en ese punto específico

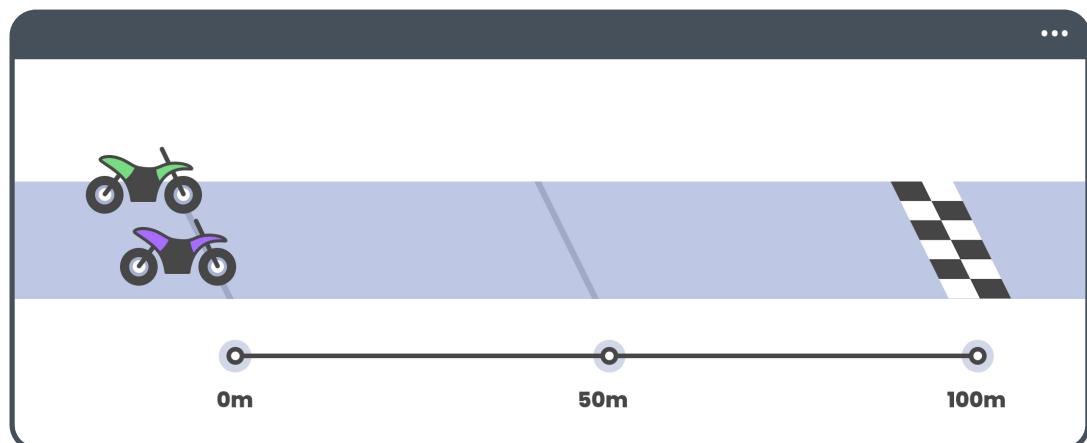
y permitiéndonos determinar con precisión la orientación de la superficie para efectos de iluminación en el shader.

Antes de implementar estos conceptos en nuestro proyecto, realizaremos una breve introducción a los fundamentos necesarios en el cálculo del gradiente. Comenzaremos con el símbolo nabla ∇ , un operador vectorial cuyas componentes son derivadas parciales:

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

(5.2.a)

La derivada es una herramienta matemática que describe cómo cambia una función en puntos específicos cuando se modifica su variable independiente. Para comprender este concepto, es fundamental familiarizarnos con las nociones de velocidad promedio y velocidad instantánea, ya que la derivada se centra en estas ideas. Iniciaremos explicando el concepto de velocidad promedio mediante un ejemplo de una carrera de motocicletas.



(5.2.b)

Imaginemos que dos motocicletas compiten en una distancia de 100 metros. Ambas parten desde reposo, y deseamos conocer su velocidad a lo largo del tiempo. Supongamos

que la motocicleta rosa llega a la meta en 5 segundos, mientras que la motocicleta verde, en ese mismo lapso, alcanza solo 80 metros.

Para calcular la velocidad promedio v de cada motocicleta, emplearemos la siguiente fórmula donde se hace presente el símbolo delta Δ :

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

(5.2.c)

Donde:

- Δx es la distancia recorrida,
- Δt es el tiempo transcurrido.

Aplicando esta ecuación a nuestro ejemplo, obtenemos:

Para la motocicleta rosa:

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

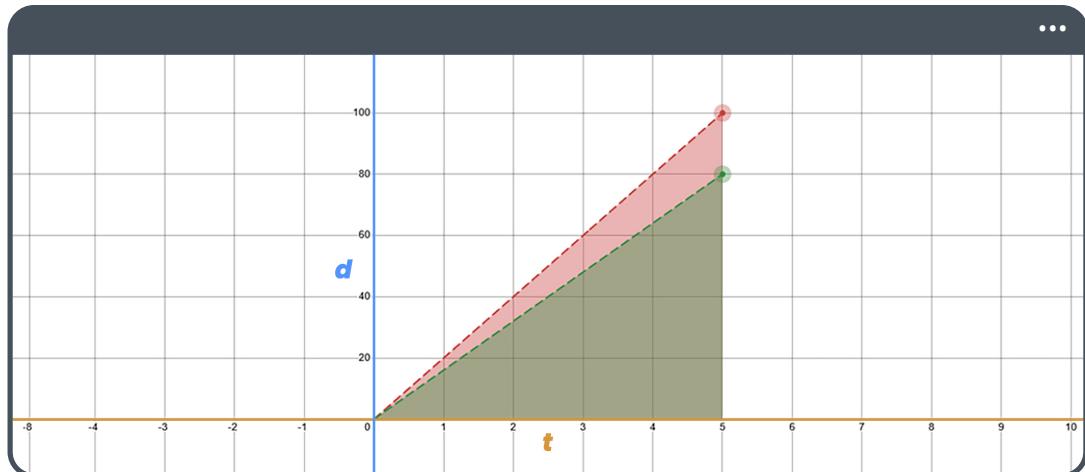
(5.2.d)

Para la motocicleta verde:

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

(5.2.e)

Podemos representar estos datos en un gráfico de posición contra tiempo para visualizar mejor cómo avanza cada motocicleta:



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

En el gráfico, el eje vertical representa la distancia d recorrida, mientras que el eje horizontal indica el tiempo t . Las líneas muestran cómo cada motocicleta avanza a lo largo del tiempo, y la pendiente de cada una corresponde a la velocidad promedio en un intervalo específico.

Siguiendo el ejemplo de las motocicletas, podemos determinar la velocidad promedio en diversos tramos. Por ejemplo, en los primeros 2 segundos, la motocicleta rosa habría recorrido 40 metros.

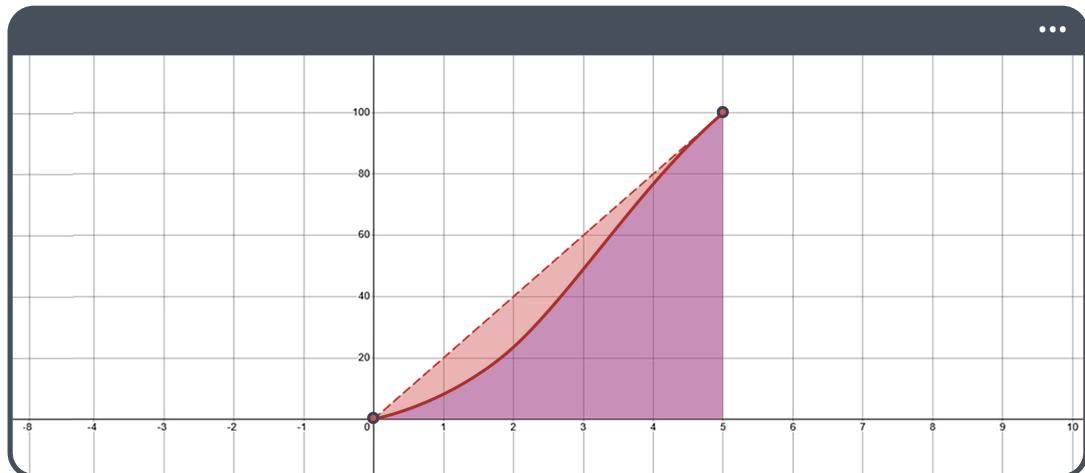
$$v_1 * 2s = 40m$$

(5.2.g)

No obstante, en la práctica, las motocicletas no alcanzan su velocidad máxima de manera instantánea. A medida que aceleran, su velocidad aumenta gradualmente, por lo que la distancia real recorrida en los primeros segundos suele ser menor que la predicha por la velocidad promedio.

Mientras que la velocidad promedio ofrece una visión global del movimiento en un intervalo, no describe cómo varía segundo a segundo. Para conocer con precisión

la variación de velocidad en cada instante, introducimos el concepto de velocidad instantánea, que corresponde a la tasa de cambio de la posición en un punto específico de la trayectoria.



(5.2.h)

En el gráfico anterior (más realista), vemos que la motocicleta rosa comienza acelerando lentamente, reflejándose en una curva en la gráfica de distancia contra tiempo. La velocidad instantánea describe el cambio de posición en un instante dado, calculado matemáticamente como la derivada de la posición con respecto al tiempo. Si queremos conocer la velocidad cuando han transcurrido t segundos, evaluamos la derivada de la función de posición en ese punto para obtener la velocidad exacta en ese instante.

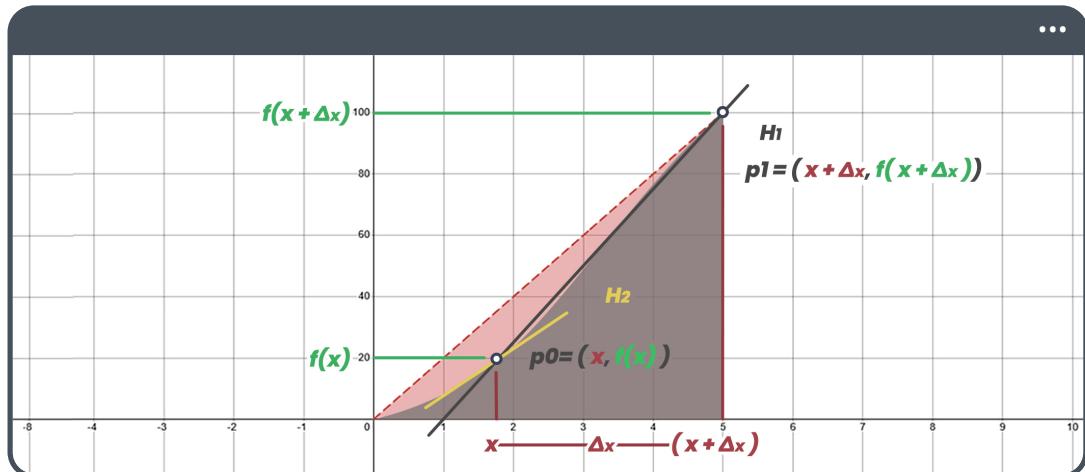
Por definición, si $f(x)$ es una función, su derivada $\frac{df}{dx}$ se define así:

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

(5.2.i)

Para ilustrar este concepto, consideraremos dos puntos en el plano cartesiano: p_0 y p_1 . Ubicamos a p_0 en el lugar donde deseamos calcular la derivada; siguiendo el ejemplo anterior, esto corresponde a determinar la velocidad instantánea aproximadamente a los

2 segundos desde el inicio de la carrera, asumiendo que la coordenada x representa el tiempo en el gráfico. Por otro lado, p_1 se sitúa, por ejemplo, en el punto correspondiente a los 5 segundos.



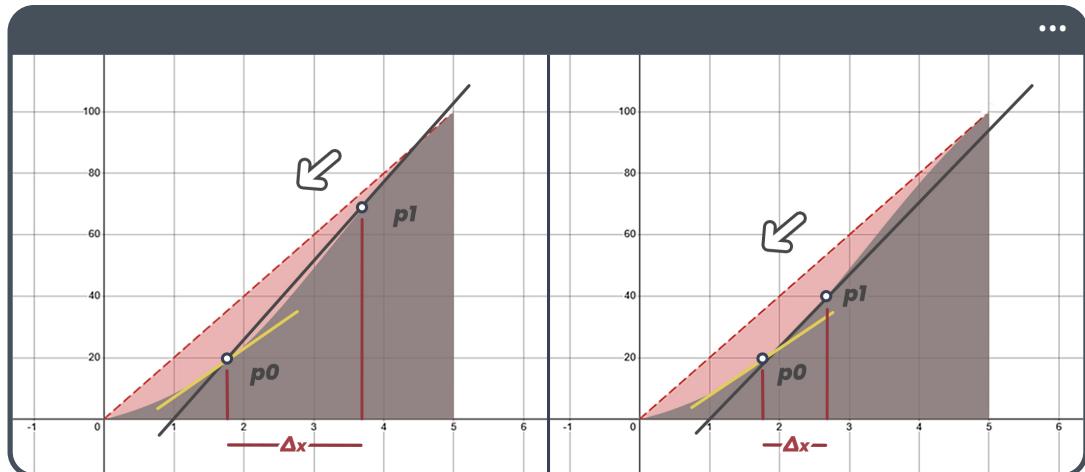
(5.2.j)

El intervalo Δx entre p_0 y p_1 describe el incremento en la coordenada x . De este modo, las coordenadas del punto p_1 se pueden expresar como:

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

(5.2.k)

Si hacemos Δx cada vez más pequeño (que tienda a cero), la línea que conecta p_0 y p_1 se aproxima a la recta tangente a la curva en el punto p_0 . Aunque Δx nunca es estrictamente cero en la práctica, un valor muy pequeño (por ejemplo, $\Delta x = 0.001$) nos permite obtener una buena aproximación de la velocidad instantánea en los primeros 20 metros.



(5.2.i)

Hasta este momento, hemos trabajado con la derivada total, que se aplica a funciones de una sola variable x . Sin embargo, cuando nos enfrentamos a funciones de varias variables, por ejemplo $f(x, y, z)$, debemos recurrir a las derivadas parciales, cada una asociada a una de las dimensiones de la función.

En el caso unidimensional; donde la función se presenta como una curva, solo existe una dirección para calcular la derivada. En cambio, para funciones de varias variables, como $f(x, y, z)$, es posible calcular derivadas en cada una de las direcciones correspondientes a las variables x , y y z . Estas funciones multivariadas describen superficies (o volúmenes) en espacios de mayor dimensión.

Volviendo a la ecuación de la figura 5.2.i, recordemos que las derivadas parciales pueden estimarse utilizando diferencias finitas, tal como vimos con la definición general de la derivada. Por ejemplo:

Derivada parcial de f , con respecto a x :

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

(5.2.m)

Derivada parcial de f , con respecto a y :

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

(5.2.n)

Derivada parcial de f , con respecto a z :

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

(5.2.ñ)

Cuando la superficie en el espacio está definida por $f(x, y, z) = k$, siendo k un número cualquiera constante, surge una pregunta natural: ¿Cómo podemos encontrar una dirección que sea perpendicular a la superficie en un punto dado? Esa dirección perpendicular es conocida como la normal a la superficie, y para encontrarla usamos el gradiente, denotado como ∇f . Este vector reúne las derivadas parciales de la siguiente manera:

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

(5.2.o)

El gradiente ∇f describe la dirección en la que la función f crece más rápidamente y con qué intensidad lo hace. Si consideramos la superficie definida por $f(x, y, z) = 0$, entonces para cualquier movimiento sobre la superficie, f permanece constante (igual a 0). Esto implica que los vectores tangentes a la superficie no modifican el valor de f .

Para aclararlo, supongamos que nos desplazamos infinitesimalmente en dirección de un vector v tangente a la superficie. El cambio infinitesimal df en la superficie $f(x, y, z) = k$ al movernos en esa dirección se describe como el producto punto:

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

(5.2.p)

Sin embargo, como $df = 0$ mientras nos mantenemos en la superficie ya que el valor de f cambia, se cumple que:

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

(5.2.q)

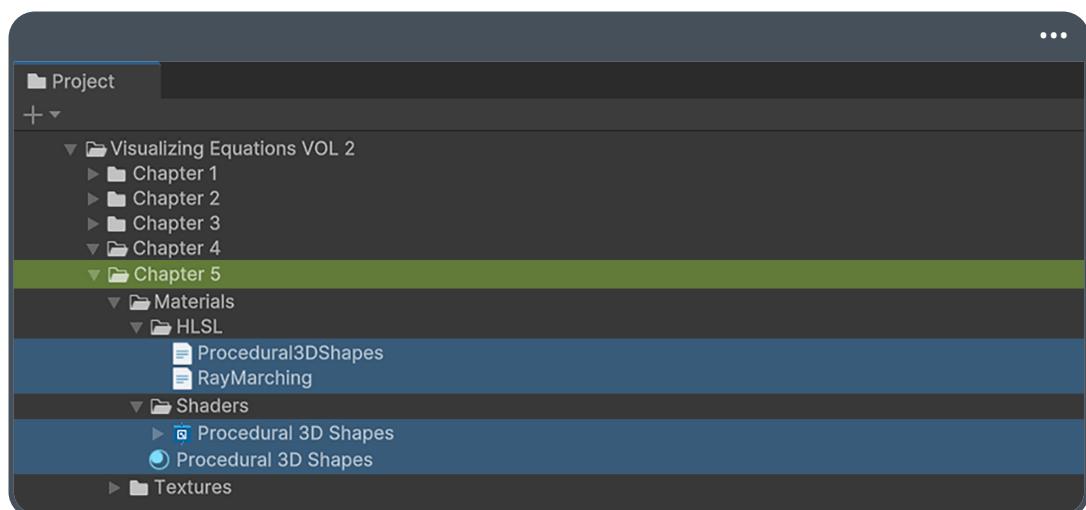
En otras palabras, ∇f es perpendicular a cualquier vector tangente v , y por lo tanto coincide con la dirección normal a la superficie. De esta forma, el gradiente ∇f resulta ser mucho más que una colección de derivadas parciales: es una herramienta geométrica que nos conecta directamente con la estructura de la superficie con la noción de perpendicularidad. Su naturaleza, al señalar el máximo cambio de f , lo convierte de manera natural en el vector normal a la superficie, mostrándonos cómo las matemáticas y la geometría se unen para describir de manera elegante figuras y espacios en nuestros proyectos de renderizado con funciones de distancia con signo.

5.3 Renderizando una forma tridimensional.

Habiendo explorado las funciones necesarias para crear una figura procedural tridimensional y el cálculo de sus normales, en esta sección nos enfocaremos en su renderización dentro de Unity. Para ello, comenzaremos creando un **Unlit Shader Graph** al que llamaremos **Procedural 3D Shapes**. A continuación, crearemos un material para

este shader, y otro archivo de tipo **.hlsl** con el mismo nombre, para mantener un flujo de trabajo ordenado.

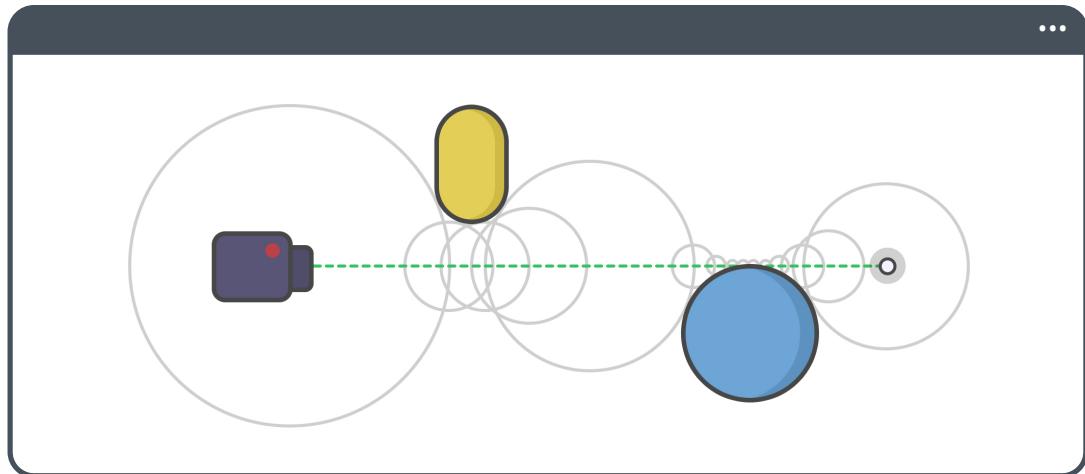
Además, añadiremos un nuevo archivo **.hlsl** al cual llamaremos **RayMarching**, que emplearemos para generar nuestro nodo personalizado dentro de Shader Graph y, posteriormente, para renderizar las figuras tridimensionales. Esta separación de archivos nos ayudará a mantener el código modular y facilitar la edición o ampliación de nuestras funciones de distancia y cálculo de normales en el futuro.



(5.3.a)

La técnica de Ray Marching resulta esencial al renderizar formas definidas por funciones de distancia con signo. En este proceso iterativo “marchamos” a lo largo de un rayo definido por su dirección y una distancia inicial. Avanzamos en segmentos pequeños, midiendo la distancia hasta los objetos cercanos de cada paso. De este modo, podemos determinar de manera precisa si el rayo colisionará con una figura, lo cual nos permitirá dibujarla de forma realista.

A continuación, podremos observar una referencia visual que nos ayudará a comprender mejor este concepto:

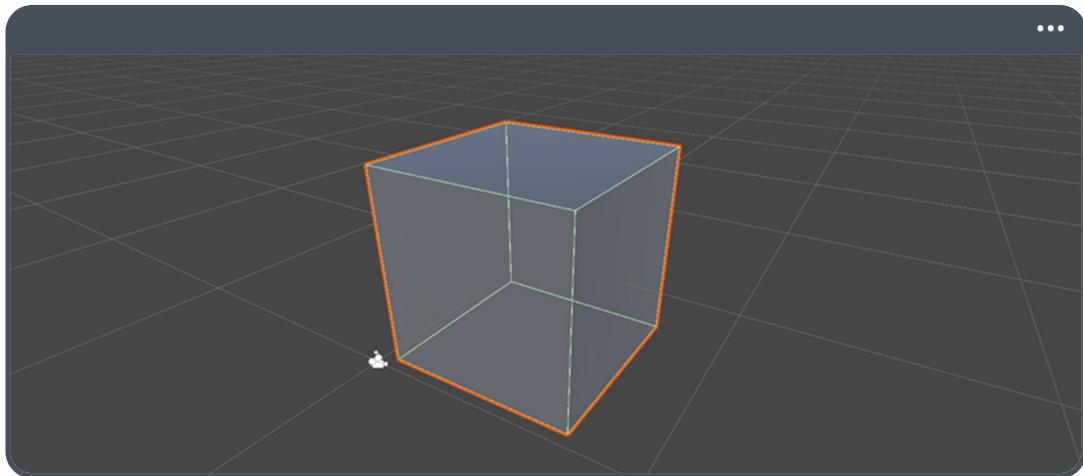


(5.3.b Conceptualización bidimensional de un rayo)

Al observar la imagen 5.3.b, vemos un rayo que viaja a través de la escena. Este rayo se compone de un punto de origen (posición de la cámara), una dirección y una distancia. A lo largo de su recorrido, se distinguen dos figuras geométricas; una cápsula y una esfera, ambas tridimensionales. En cada iteración o “salto” del rayo se evalúa la distancia a la superficie más cercana, lo que nos permite detectar colisiones y, así, definir la forma y ubicación de los objetos en la escena.

Ahora bien, ¿cómo aplicamos esta técnica en Unity? Para lograrlo, necesitamos definir propiedades en Shader Graph y crear un nodo personalizado que se encargue de realizar estos cálculos de distancia y detección de colisiones. Dicho nodo empleará nuestras funciones de distancia con signo y avanzará el rayo en pequeñas iteraciones hasta encontrar (o no) la superficie de un objeto.

En este proceso, será fundamental contar con un “escenario” en el cual dibujar las figuras procedurales. Podemos usar cualquier primitiva incluida por defecto en el motor, pero, con fines educativos, utilizaremos un Cubo como base para nuestros cálculos. Este Cubo actuará como la superficie sobre la que aplicaremos nuestro shader de Ray Marching, de modo que nuestras figuras procedurales (la cápsula y la esfera) se dibujen correctamente en su interior.



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

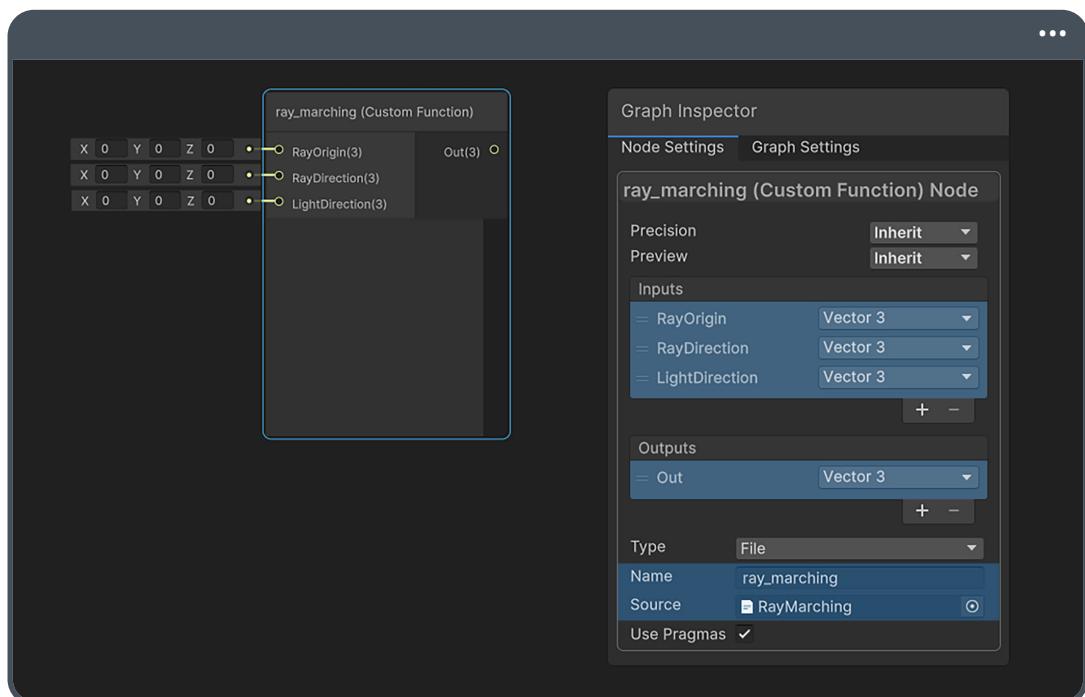
Para visualizar el efecto que queremos lograr, asignaremos el shader **Procedural 3D Shapes** a su respectivo material y, a continuación, aplicaremos dicho material al Cubo en la escena. Seguidamente, iniciaremos el proceso de desarrollo agregando un nodo **Custom Function** dentro del Shader Graph que hemos creado previamente.

Antes de configurar las entradas y salidas de este nodo, haremos un breve repaso de las propiedades que necesitaremos para renderizar las formas tridimensionales:

- **Posición de los vértices:** Hace referencia al espacio donde deseamos dibujar la figura procedural. Emplearemos la posición de los vértices en object-space como punto de origen para realizar los cálculos dentro del área del Cubo. Si quisieramos, también podríamos usar la posición de la cámara, pero, en este caso, preferimos apoyarnos en la posición de los vértices para optimizar los cálculos y simplificar la lógica.
- **Cámara:** Utilizaremos esta propiedad para determinar la dirección del rayo. En esencia, el rayo partirá de la cámara hacia la forma procedural que se encuentra en nuestra escena, lo cual nos permitirá calcular la intersección con las superficies generadas.

➤ **Iluminación:** Con el fin de calcular la iluminación en nuestras figuras procedurales, necesitaremos conocer tanto las normales como la dirección de la luz. En este sentido, emplearemos el nodo **Main Light Direction** que nos brinda acceso a la luz direccional principal de la escena.

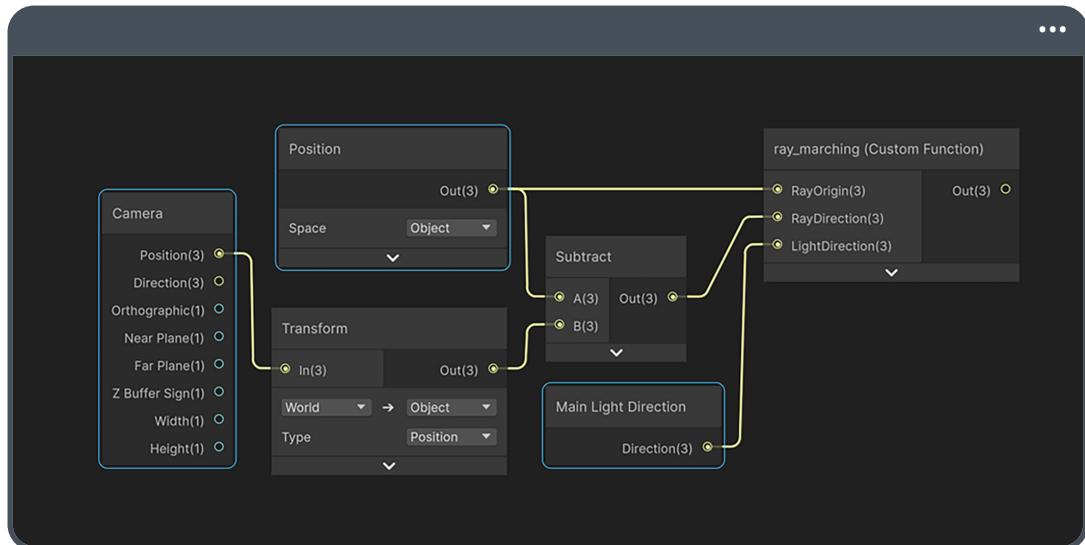
Tal como describimos en la sección anterior, será necesario calcular las normales de la forma procedural manualmente, ya que Shader Graph no proporciona un nodo específico para esta tarea. Por lo tanto, definiremos únicamente las propiedades esenciales para nuestro nodo: **RayOrigin**, **RayDirection** y **LightDirection**. Estas entradas nos permitirán realizar la técnica de Ray Marching, así como aplicar los modelos de iluminación adecuados para renderizar las figuras tridimensionales.



(5.3.d Nodos Custom Function con entradas y salida)

Si examinamos la imagen 5.3.d, notaremos que las tres propiedades descritas previamente se presentan como vectores tridimensionales. Antes de avanzar con la implementación de las funciones para la renderización, debemos conectar cada una de estas propiedades con los nodos correspondientes en el shader. De este modo, aseguraremos que la posición

de los vértices, la posición de la cámara y la dirección de la iluminación en nuestra escena lleguen correctamente a nuestro nuevo nodo personalizado.



(5.3.e Visualización de los nodos en Shader Graph)

En la imagen 5.3.e, podemos apreciar que la posición de los vértices (en object-space) se ha conectado como origen del rayo en nuestro nodo **Custom Function**. Para comprender mejor esta elección, debemos preguntarnos cuál debería ser la influencia del Cubo sobre las figuras que se dibujen en su interior. Por ejemplo, si rotamos el Cubo; nuestro escenario para renderizar las figuras procedurales, ¿queremos que las figuras en su interior roten junto a él o que permanezcan estáticas, sin importar la posición, rotación y escala de este?

Si buscamos que las figuras procedurales se vean influenciadas por la transformación del Cubo, debemos ejecutar nuestros cálculos en object-space. En consecuencia, la posición de la cámara se transforma de world-space a object-space mediante el nodo Transform, atendiendo a dos aspectos fundamentales:

- La documentación oficial de Shader Graph indica que la posición del nodo **Camera** se define en world-space de manera predeterminada.

- Posteriormente, calculamos la dirección del rayo restando la posición de la cámara a la de los vértices; por lo tanto, ambas coordenadas deben hallarse en el mismo espacio para que los resultados de la resta sean válidos.

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>)

En nuestro nodo Custom Function, también hemos conectado el nodo **Main Light Direction** al parámetro correspondiente en la función. Con estas referencias listas (origen del rayo, dirección del rayo y dirección de la luz), podemos implementar las funciones necesarias para la renderización.

Como primer paso, crearemos el método **ray_marching_float()**, el cual recibe como entrada las propiedades que hemos mencionado previamente. Para asegurarnos de que nuestro nodo compile correctamente, asignaremos un valor cualquiera a los componentes RGB de la salida. Esto nos permitirá confirmar que la estructura básica del nodo está funcionando antes de profundizar en la lógica de Ray Marching.

Por ejemplo, podríamos declarar el método de la siguiente manera en nuestro archivo **.hlsl**:

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

En el fragmento de código anterior, observamos que los argumentos **RayOrigin**, **RayDirection** y **LightDirection** pertenecen al nodo **Custom Function**, donde calcularemos la distancia a las superficies y determinaremos el color final según la iluminación y las propiedades de la figura. Este paso inicial nos permite comprobar que el shader se compila y ejecuta correctamente, estableciendo la base para la implementación completa de la técnica de Ray Marching.

Para concluir la configuración de nuestro shader, debemos llevar a cabo tres tareas fundamentales:

- Definir la forma que deseamos renderizar (en este caso, una esfera).
- Calcular las normales de dicha forma.
- Implementar un método que lleve a cabo la operación de renderizado. Este utilizará la función de distancia para recorrer el espacio y detectar la superficie de la figura.

Comenzaremos declarando la función que describe la esfera. Para ello, abriremos el script **Procedural3DShape** y añadiremos las siguientes líneas de código:

```
1 float sphere_sd (float3 p, float r)
2 {
3     return length(p) - r;
4 }
```

Esta función, denominada **sphere_sd()**, devuelve la distancia entre un punto arbitrario **p** (colisión del rayo) del espacio tridimensional y la superficie de una esfera de radio **r**. Con esta definición, un valor negativo indica que el punto se encuentra dentro de la esfera, un valor igual a cero señala que el punto está exactamente sobre la superficie y un valor positivo indica que el punto está fuera de la esfera. El punto **p** se describe como:

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

(5.3.g)

Donde **O** es igual al origen del rayo, **d_o** representa la distancia escalar recorrida en cada iteración y **D** es la dirección. Dado que la esfera se define en un archivo distinto de **RayMarching**, será necesario incluir este archivo para poder utilizar la función. Para ello, empleamos la directiva **#include**, como se muestra a continuación:

```
1 #include "Assets/Jettelly Books/.../Procedural3DShapes.hlsl"
2
3 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
  ↵ float3 lightDirection, out float3 RGB) { ... }
```

En el ejemplo anterior, hemos optimizado la ruta empleada en la directiva para simplificar su longitud. Sin embargo, podemos acceder al archivo utilizando su ruta completa:

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

A continuación, incluiremos la esfera que declaramos recientemente dentro de un nuevo método en **RayMarching**, al cual llamaremos **map()**. Este método nos permitirá posicionar todas las formas procedurales que deseemos colocar en el Cubo de la escena.

```

1 #include "Assets/Jettelly Books/.../Procedural3DShapes.hlsl"
2
3 inline float map (float3 p)
4 {
5     const float sphere_radius = 0.3;
6     float sphere = sphere_sd(p, sphere_radius);
7     return sphere;
8 }
9
10 void ray_marching_float (in float3 rayOrigin, in float3 rayDirection, in
    ← float3 lightDirection, out float3 RGB) { ... }

```

Si prestamos atención a la línea de código número 5, notaremos la declaración de la variable **sphere_radius**, cuyo valor es 0.3. Recordemos que el sistema métrico en Unity se define en metros, de modo que una unidad equivale a 100 cm. Por lo tanto, en este caso, nuestra esfera tiene un radio de 30 cm.

Otro aspecto relevante aparece en la línea de código número 3, donde vemos el uso del parámetro **inline** antes del nombre del método. Dicho parámetro (conocido como **StorageClass**) se relaciona con la forma en que el compilador gestiona los métodos durante el proceso de compilación. Cuando un método se invoca, el sistema se desplaza hasta la posición en memoria donde se encuentra el método, ejecuta el código y luego regresa al punto en el que se realizó la llamada. Este mecanismo genera un cierto costo adicional derivado de la propia llamada a la función. Sin embargo, al marcar un método como **inline**, el compilador puede insertar el código del método directamente en el lugar donde se llama, evitando sobrecargas y optimizando eventualmente el rendimiento.

Es importante señalar que, según la documentación oficial de Microsoft, en HLSL todos los métodos se tratan como **inline** de forma predeterminada, incluso si no se especifica este parámetro. Aun así, conviene tener presente que, al crear métodos muy extensos, el tamaño del shader podría aumentar durante la compilación y causar fallas de caché.

Con esta explicación en mente, continuaremos implementando el método encargado de realizar el algoritmo de Ray Marching, necesario para renderizar nuestra esfera. Por motivos prácticos, lo llamaremos `render()` le pasaremos tanto el punto de origen como la dirección del rayo como argumentos.

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

Considerando que el proceso de renderizado se lleva a cabo de manera iterativa, la primera variable que debemos incluir dentro del método es la distancia a lo largo del rayo d_0 . Esta comienza en 0, coincidiendo con la posición de la cámara en la escena:

```
10 inline float render(float3 rayOrigin, float3 rayDirection)
11 {
12     float ray_distance = 0;
13     return 0;
14 }
```

A continuación, definiremos la cantidad máxima de iteraciones y crearemos un bucle `for` que avance a lo largo del rayo en la escena. Para ello, realizaremos dos acciones:

- Declarar la directiva `#define MAX_ITERATIONS`.
- Añadir un bucle `for` que ejecute una iteración en cada paso.

```
3 #define MAX_ITERATIONS 256
```

```

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     }
18 }
19     return 0;
20 }
```

Para determinar la superficie de la esfera, debemos calcular la posición actual del rayo a medida que avanza. Con este fin, podemos usar la ecuación 5.3.g dentro del bucle, sumando al origen el desplazamiento por la dirección en cada iteración.

Posteriormente, pasamos el resultado (un nuevo vector tridimensional) como argumento al método `map()`, tal como se muestra a continuación:

```

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 }
```

En un principio, la posición del rayo se encuentra en el origen; no obstante, después de cada paso, avanzará en función de la expresión `ray_distance` multiplicado por `rayDirection`, como se aprecia en la línea 17 del ejemplo anterior. Más adelante, al utilizar el método `render()` en `ray_marching_float()`, deberemos normalizar la dirección del rayo para asegurar una escala adecuada.

Hasta este momento, la distancia del rayo permanece en su valor inicial. Por ello, necesitaremos incrementarla progresivamente en la escena hasta que colisione con la esfera que deseamos dibujar. Sin embargo, esto presenta dos interrogantes: ¿qué ocurre si el rayo viaja más allá de la superficie de la esfera procedural?, y ¿con qué precisión deberíamos determinar el contacto con dicha superficie? Para resolverlo, definiremos la distancia máxima que puede recorrer el rayo y la tolerancia con la cual mediremos la colisión.

```
3 #define MAX_ITERATIONS 256  
4 #define MAX_DISTANCE 100.0  
5 #define SURFACE_DISTANCE 0.001
```

En el fragmento anterior, utilizamos la directiva **#define** para especificar dos constantes adicionales.

- **MAX_DISTANCE** (100m) limita el alcance del rayo, de modo que solo se rendericen las figuras ubicadas dentro de ese rango.
- **SURFACE_DISTANCE** define la resolución o mínima distancia para considerar que el rayo ha tocado la superficie de la esfera.

Por último, en el método **render()**, incrementamos la distancia del rayo en cada iteración y verificamos si colisiona con la esfera o si supera la distancia máxima permitida:

```

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 }
```

De esta manera, cada vez que **scene_distance** sea menor a la precisión establecida, asumimos que el rayo impactó en la superficie de la esfera. Por el contrario, si **ray_distance** supera el máximo de distancia, detendremos el bucle, descartando la posibilidad de intersección dentro del rango definido.

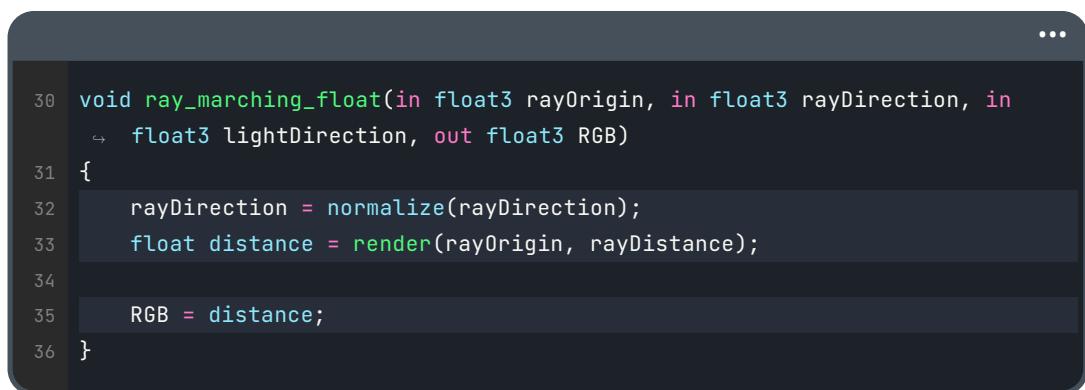
Si prestamos atención a las líneas 21 a 25, notaremos que el código implementa la lógica para determinar si el rayo ha alcanzado una superficie o ha superado el límite máximo en la escena. En la línea 21, la variable **ray_distance** se incrementa con el valor de **scene_distance**, que corresponde a la distancia calculada entre la posición actual del rayo y la superficie más cercana, a través del método **map()**. Posteriormente, en las líneas 22 a 25, aparece una condición crucial para el algoritmo de Ray Marching, la instrucción **if** verifica dos casos:

- Colisión con la superficie: Si **scene_distance** es menor que **SURFACE_DISTANCE**, el rayo está suficientemente cerca de la esfera (dentro de la tolerancia definida). En ese instante, consideraremos que ha ocurrido una intersección (tal como se muestra en la referencia 5.3.b).

- Exceso de distancia máxima: Si **ray_distance** supera **MAX_DISTANCE**, significa que el rayo ha viajado más allá del límite establecido sin toparse con una superficie. Continuar las iteraciones resultaría innecesario.

En cualquiera de estos escenarios, el bucle **for** se detiene (con la instrucción **break**), optimizando el rendimiento al evitar iteraciones adicionales que no aportarían cambios significativos al resultado. Finalmente, el método **render()** devuelve **ray_distance**, que representa la distancia total recorrida por el rayo hasta el punto de colisión o hasta alcanzar la distancia máxima.

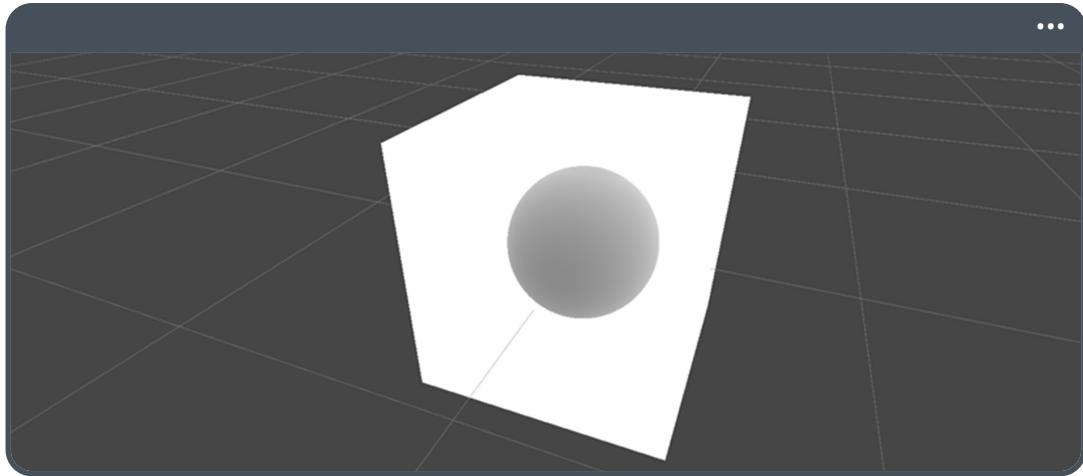
Para visualizar la esfera, podemos emplear el valor de distancia obtenido en el método **render()**, según se muestra a continuación:



```

30 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
31   float3 lightDirection, out float3 RGB)
32 {
33     rayDirection = normalize(rayDirection);
34     float distance = render(rayOrigin, rayDistance);
35     RGB = distance;
36 }
```

Si prestamos atención al código anterior, observamos que en la línea 32 normalizamos **rayDirection** para asegurar que el vector tiene longitud uno, garantizando así una dirección precisa del rayo. Posteriormente, en la línea 33, llamamos al método **render()**, el cual devuelve la distancia total que el rayo ha recorrido hasta encontrar una superficie (en este caso, la esfera) o alcanzar la distancia máxima permitida. Este resultado se almacena en la variable **distance**, la cual finalmente asignamos a **RGB**. De esta manera, podemos visualizar la escena en función de la distancia calculada.



(5.3.h Visualización de la distancia respecto a la cámara)

En el Cubo de nuestra escena, el color blanco alrededor de la esfera corresponde a los rayos que no lograron ninguna colisión, ya que retornan un valor mayor a uno. Para comprender mejor este comportamiento, es necesario implementar un método que nos permita aproximar las normales de la esfera. Para ello, utilizaremos la función 5.2.o de la sección anterior, la cual determina la derivada (o diferencias finitas) de una función respecto a sus 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 }
```

Cabe destacar que la implementación anterior no es la manera más optimizada ni precisa para aproximar las normales de una superficie implícita, ya que implica varias

sustracciones, divisiones y sumas. No obstante, resulta una forma intuitiva y directa de ilustrar cómo calcular las normales utilizando diferencias finitas.

Podemos considerar, además, la siguiente implementación como una alternativa más optimizada para el cálculo de las normales. Esta versión evita operaciones de división costosas y proporciona mayor estabilidad a la función, entregando el mismo resultado.

```
1 inline float3 get_normal(float3 p)
2 {
3     float2 e = float2(0.00001, 0);
4     float3 n = map(p) - float3(map(p - e.xyy), map(p - e.yxy), map(p -
5         e.yyx));
6     return normalize(n);
}
```

La función **get_normal()** calcula la normal de un punto **p** aproximando las derivadas parciales de la función de distancia **map()** respecto a cada componente del vector. Para ello, utilizamos un pequeño incremento **h** (línea 32) y evaluamos cómo varía la función de distancia al desplazarse ligeramente en cada dirección. Al normalizar el vector resultante (línea 37), obtenemos la dirección de la normal en el punto **p**, lo cual será esencial para calcular efectos de iluminación en nuestra forma procedural.

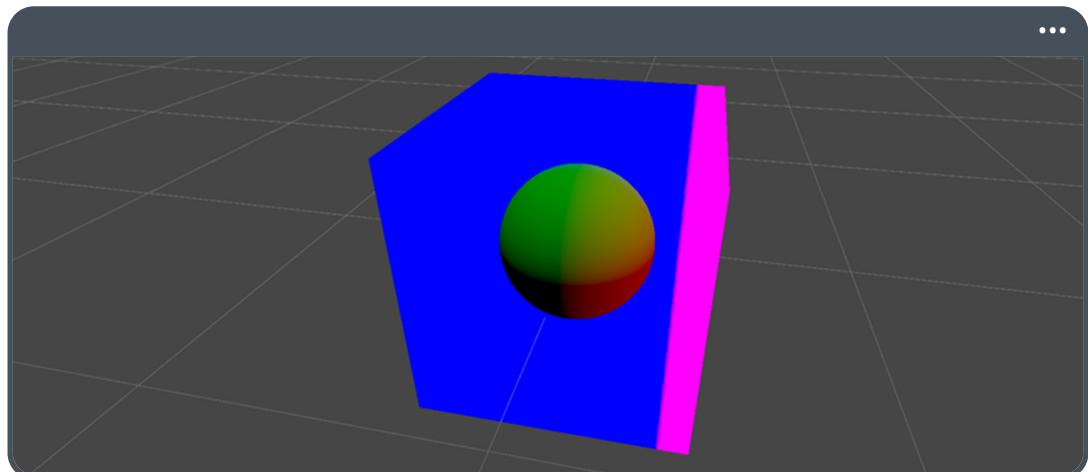
De manera similar al método **render()**, el punto **p** se determina mediante la fórmula 5.3.g, es decir:

```

40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
41   float3 lightDirection, out float3 RGB)
42 {
43     rayDirection = normalize(rayDirection);
44     float distance = render(rayOrigin, rayDistance);
45     half3 color;
46
47     float3 p = rayOrigin + distance * rayDirection;
48     float3 normal = get_normal(p);
49     color = normal;
50
51     RGB = color;
}

```

El vector **color** hace referencia al color final de la imagen. Como podemos observar en la línea 48, este se obtiene al calcular las normales mediante el método **get_normal()** en la línea 47. Al guardar y volver a la escena, podremos visualizar la dirección de las normales sobre la esfera en nuestra escena.



(5.3.i Visualización de las normales de la esfera)

A continuación, implementaremos un modelo de iluminación simple para mejorar la visualización de nuestra forma procedural. Utilizaremos el modelo de iluminación difusa de Lambert, que calcula la intensidad de la luz en función del ángulo entre las normales

de la superficie n y la dirección de la luz incidente l . La fórmula para la iluminación difusa es la siguiente:

$$L = \max(0, n \cdot l)$$

(5.3.j)

Podemos implementar esta fórmula de la siguiente manera:

```

40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
41   float3 lightDirection, out float3 RGB)
42 {
43     rayDirection = normalize(rayDirection);
44     float distance = render(rayOrigin, rayDistance);
45     half3 color;
46
47     float3 p = rayOrigin + distance * rayDirection;
48     float3 normal = get_normal(p);
49     float l = max(0.0, dot(normal, -lightDirection)) + 0.1;
50     color = l;
51
52     RGB = color;
}

```

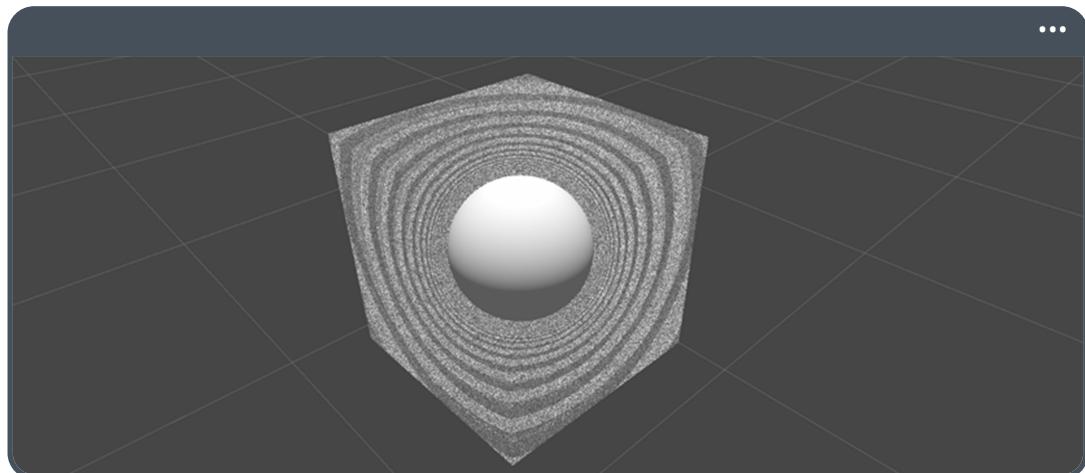
La función es bastante simple: toma el máximo entre 0.0 y el producto punto entre las normales en el punto p y la dirección de la luz que tenemos en la escena. Al establecer el valor mínimo en 0.0, evitaremos números negativos y, con ello, artefactos visuales asociados a la luz.

Si prestamos atención a la línea de código 48, veremos que al final de la operación se agrega 0.1 a la función. Este valor actúa como iluminación global, definiendo un nivel mínimo de iluminación en toda la escena, de modo que las áreas que no reciben luz directa no queden completamente oscuras.

Podríamos utilizar una implementación más elaborada para la iluminación global, por ejemplo: emplear el nodo **BakedGI** para generar un lightmap que cumpla la misma función. Sin embargo, en esta ocasión, este pequeño valor adicional resulta suficiente para mostrar la figura con un contraste suavizado. No obstante, la inclusión de este número genera un problema de sobrepasar el rango, ya que el valor máximo ahora puede ser mayor a 1.0, lo cual puede generar artefactos visuales. Para solucionarlo, simplemente debemos limitar el rango de la variable **l** a un valor entre 0.0 y 1.0, como se muestra a continuación:

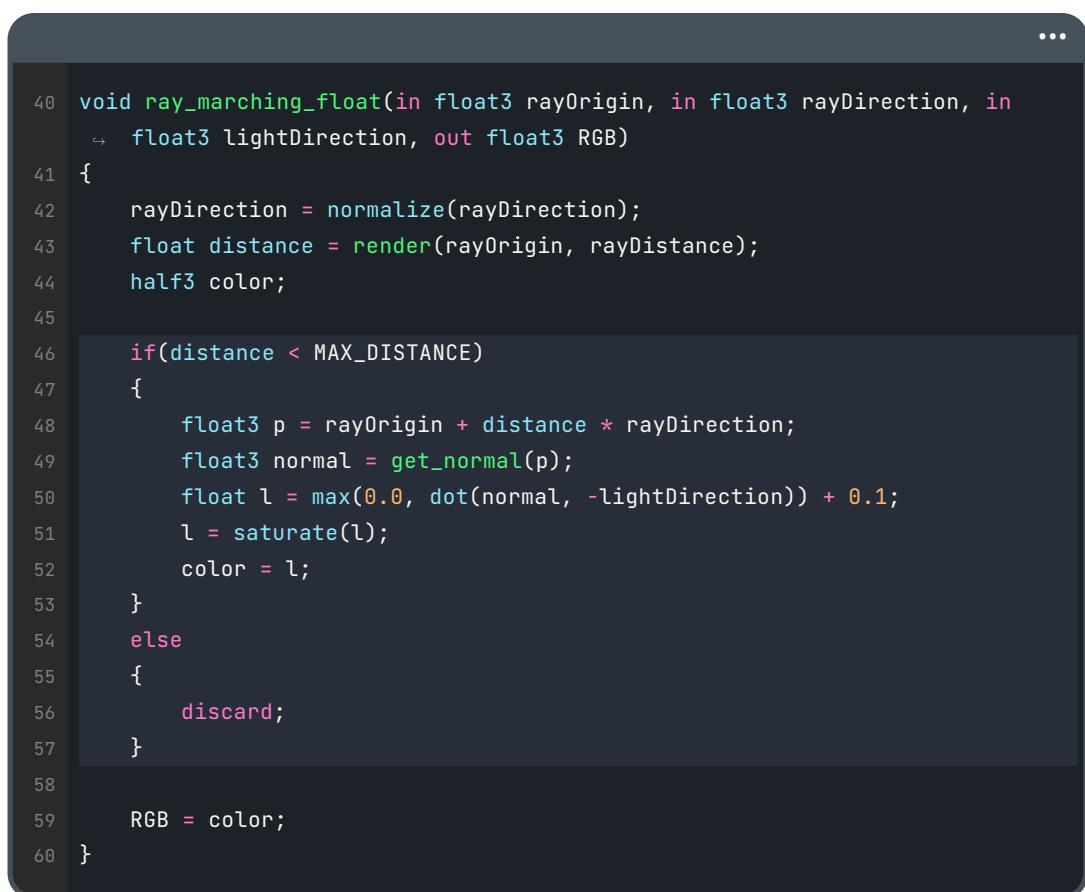
```
48 float l = max(0.0, dot(normal, -lightDirection)) + 0.1;  
49 l = saturate(l);  
50 color = l;
```

La función **saturate()** asegura que el valor de **l** se mantenga dentro del rango [0.0 : 1.0], evitando así posibles artefactos visuales causados por valores superiores a 1.0. Con esta corrección, logramos una iluminación más controlada y visualmente coherente en nuestra escena.



(5.3.k Visualización de ruido alrededor de la esfera)

Como se puede observar en la referencia anterior, el volumen de la esfera se percibe correctamente. Si orbitamos alrededor del Cubo o rotamos la luz direccional, notaremos que la proyección de iluminación y sombra sobre la esfera sigue la dirección de la luz. Sin embargo, al mismo tiempo todos los rayos provenientes del método `render()` también están siendo calculados, lo cual se percibe como “ruido” al interior del Cubo. Para deshacernos de este ruido, simplemente lo podemos descartar utilizando la declaración `discard`, la cual es equivalente a “no mostrar el resultado del píxel actual”.



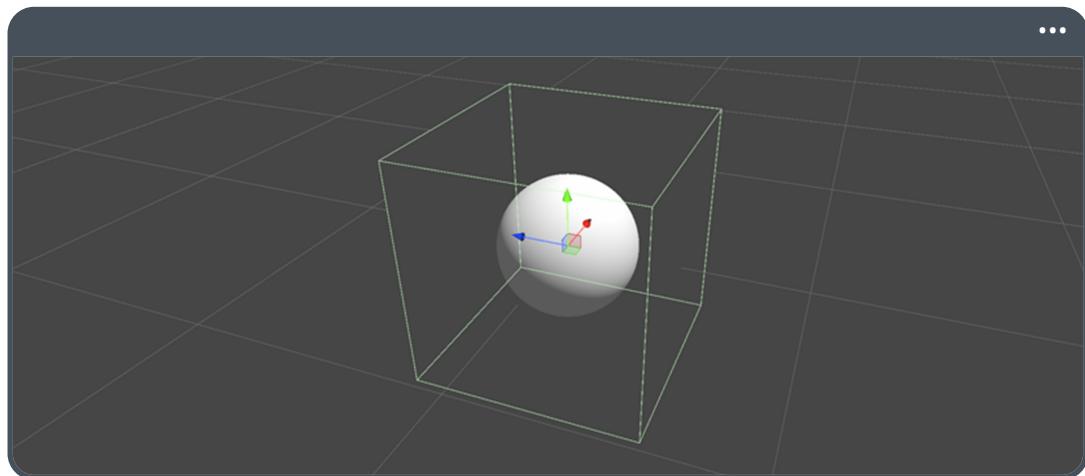
```

40 void ray_marching_float(in float3 rayOrigin, in float3 rayDirection, in
41   float3 lightDirection, out float3 RGB)
42 {
43     rayDirection = normalize(rayDirection);
44     float distance = render(rayOrigin, rayDistance);
45     half3 color;
46
47     if(distance < MAX_DISTANCE)
48     {
49         float3 p = rayOrigin + distance * rayDirection;
50         float3 normal = get_normal(p);
51         float l = max(0.0, dot(normal, -lightDirection)) + 0.1;
52         l = saturate(l);
53         color = l;
54     }
55     else
56     {
57         discard;
58     }
59     RGB = color;
60 }
```

Si prestamos atención a la condicional, entre las líneas 46 y 57, observaremos que todos aquellos píxeles fuera del área de la esfera han sido descartados mediante la palabra reservada `discard`. Esta instrucción tiene un efecto inmediato: el píxel correspondiente no se escribe en el `framebuffer`, por lo que cualquier cálculo realizado para ese píxel se descarta y no es visible en la imagen final. Este comportamiento es útil para evitar

artefactos como el ruido mencionado, que aparece porque el rayo penetra en áreas donde la geometría o la iluminación no se han definido de manera precisa; por ende, los rayos continúan su camino sin colisionar con ningún objeto.

Esta metodología de descartar píxeles y saturar la iluminación puede resultar muy útil cuando el objetivo es mostrar únicamente la geometría o la forma procedural que realmente nos interesa visualizar. No obstante, es importante tener en cuenta que **discard** no siempre es la mejor opción en términos de rendimiento, especialmente si se combina con técnicas de sombreado más complejas o efectos de post-processing. En estos casos, se podría considerar otras estrategias, como el uso de **Depth Buffers**. Este último también solucionaría los detalles de profundidad que ocurren cuando intentamos incluir otro objeto tridimensional (**Mesh Renderer**) a la escena.



(5.3.I Esfera SDF siendo afectada por la luz)

Con esta implementación, hemos logrado optimizar la visualización de nuestra esfera procedural, eliminando el ruido no deseado y asegurando que la iluminación refleje adecuadamente la interacción entre la luz y la superficie. En la siguiente sección, combinaremos la esfera con una cápsula empleando la función **smin()**, lo que nos permitirá crear formas más complejas y enriquecidas.

5.4 Mezclando dos formas tridimensionales.

La capacidad de combinar formas tridimensionales nos brinda una gran flexibilidad para crear objetos complejos y visualmente interesantes en un contexto procedural. En esta sección, exploraremos la incorporación de una cápsula siguiendo la definición de la sección 5.1.f, y extenderemos el método `map()` utilizando la función `smin()` para lograr una transición suave entre ambas formas.

Para comenzar, nos dirigiremos al script **Procedural3DShapes** y agregaremos un nuevo método al cual llamaremos `capsule_sd()`, como se muestra a continuación:

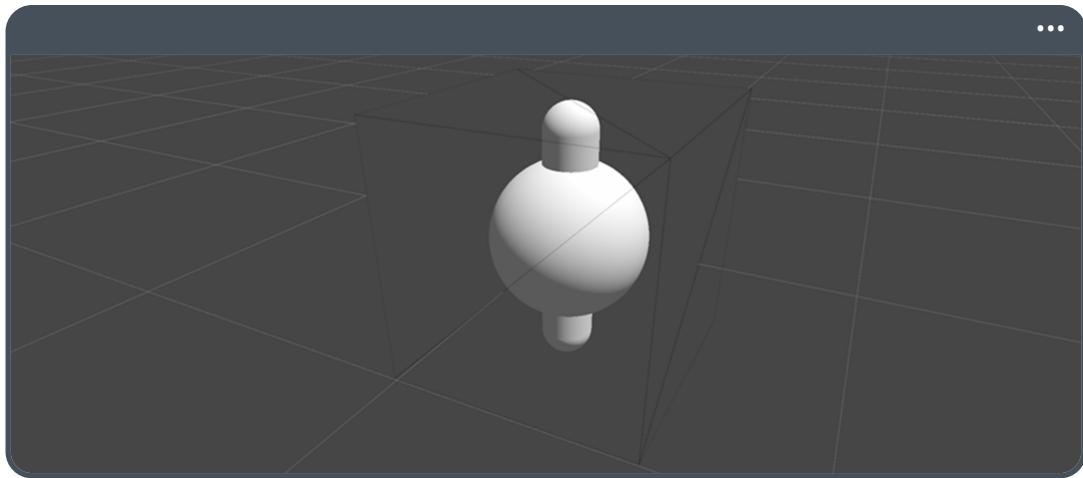
```
6 float capsule_sd(float3 p, float3 p0, float3 p1, float r)
7 {
8     float h0 = dot(p - p0, p1 - p0) / dot(p1 - p0, p1 - p0);
9     float3 h1 = clamp(h0, 0.0, 1.0);
10    float3 v = p0 + h1 * (p1 - p0);
11    return sqrt(dot(p - v, p - v)) - r;
12 }
```

Como podemos observar en el código anterior, `capsule_sd()` calcula la distancia mínima desde un punto `p` en el espacio tridimensional hasta la superficie de una cápsula definida por los argumentos `p0` y `p1`, los cuales corresponden a sus extremos. En la línea 8, la variable `h0` representa la proyección escalar del vector `p - p0` sobre el eje de la cápsula `p1 - p0`. Posteriormente, en la línea 9, el valor de `h0` se limita entre 0.0 y 1.0 utilizando la función `clamp()`. Esto asegura que la proyección permanezca dentro del segmento definido por `p0` y `p1`, evitando que se extienda más allá de sus extremos. Luego, en la línea 10, declaramos un nuevo vector `v`, que corresponde al punto más cercano en el eje de la cápsula en el punto `p`. Finalmente, retornamos el cálculo de la distancia euclíadiana entre `p` y `v`, restando el radio `r`, lo que nos da la distancia desde `p` hasta la superficie de la cápsula.

Para visualizar este objeto en la escena, será necesario regresar al script **RayMarching** y posicionarnos sobre el método **map()**, encargado de contener las figuras procedurales en nuestro código. Para su renderización, emplearemos algunos valores estudiados previamente para asegurar su correcta proyección en la escena, como se muestra a continuación:

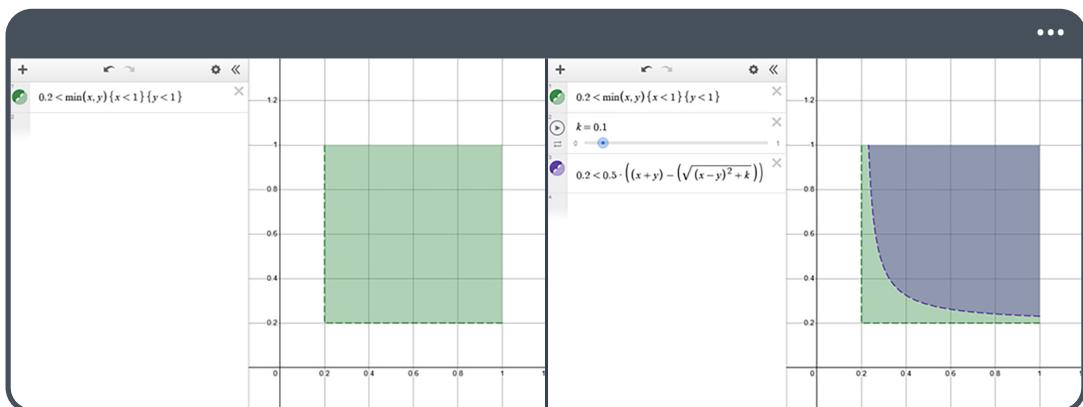
```
8 inline float map(float3 p)
9 {
10    const float sphere_radius = 0.3;
11    float sphere = sphere_sd(p, sphere_radius);
12
13    const float capsule_radius = 0.1;
14    const float3 capsule_p0 = float3(0.0, 0.4, 0.0);
15    const float3 capsule_p1 = float3(0.0, -0.4, 0.0);
16    float capsule = capsule_sd(p, capsule_p0, capsule_p1, capsule_radius);
17
18    return min(sphere, capsule);
19 }
```

Como podemos observar en el ejemplo anterior, se han declarado tres nuevas variables constantes dentro del método **map()**, las cuales hacen referencia al radio de la cápsula (**capsule_radius**), a su primer punto (**capsule_p0**) y a su segundo punto (**capsule_p1**). Estas variables se incluyen como argumentos en el método **capsule_sd**, el cual captura la forma geométrica tridimensional de la cápsula. Finalmente, devolvemos el mínimo entre la esfera y la cápsula utilizando la función **min()**, lo cual da el siguiente resultado gráfico:



(5.4.a Mínimo entre la esfera y la cápsula)

Las intersecciones entre ambas figuras son duras. Esto se debe a que la función `min()` retorna un valor dependiendo de si el primer argumento es menor que el segundo. En consecuencia, no existe un intervalo de valores que genere una interpolación suavizada entre ambas figuras. Para solucionar este problema, podemos emplear una función mínima suavizada, la cual corresponde a una interpolación entre dos valores mediante un operador de mínimo modificado. Esta función no realiza una transición abrupta, sino que introduce una curva de suavidad controlada por un parámetro k , que determina la amplitud de la zona de transición.

(5.4.b <https://www.desmos.com/calculator/zonkb4pky5>)

Como se puede observar en la imagen 5.4.b, la referencia de la izquierda contiene una desigualdad la que calcula si 0.2 es menor que el menor de los valores entre x e y . Para aquellos valores que son mayores a xy , se define una región en el plano cartesiano que podemos apreciar color verde. Este mismo comportamiento se observa en la referencia de la derecha, con la diferencia que la operación de la función **smin()** devuelve un valor suavizado que podemos controlar mediante la variable **k**.

Para comprender mejor esta diferencia, comparemos la definición por defecto de la función **min()** con su contraparte suavizada, **smin()**.

Función **min()**.

```
1 float3 min(float3 a, float3 b)
2 {
3     return float3(a.x < b.x ? a.x : b.x,
4                     a.y < b.y ? a.y : b.y,
5                     a.z < b.z ? a.z : b.z);
6 }
```

La función **min()** toma dos vectores tridimensionales **a** y **b**, y devuelve un nuevo vector donde cada componente es el mínimo correspondiente entre sus argumentos. Esta operación es útil para combinar dos funciones de distancia con signo de manera que la forma resultante representa la unión de ambas figuras.

Función **smin()**.

```
1 float smooth_min(float a, float b, float k)
2 {
3     return 0.5 * ((a + b) - sqrt(pow(a - b, 2.0) + k));
```

Por otro lado, la función `smin()` (o `smooth_min()` como aparece en el ejemplo) introduce una suavización en la operación de mínimo. Como se mencionó anteriormente, la variable `k` controla el grado de suavidad: valores más pequeños de `k` resultan en una transición más abrupta entre las dos formas, mientras que valores mayores generan una fusión más suave.

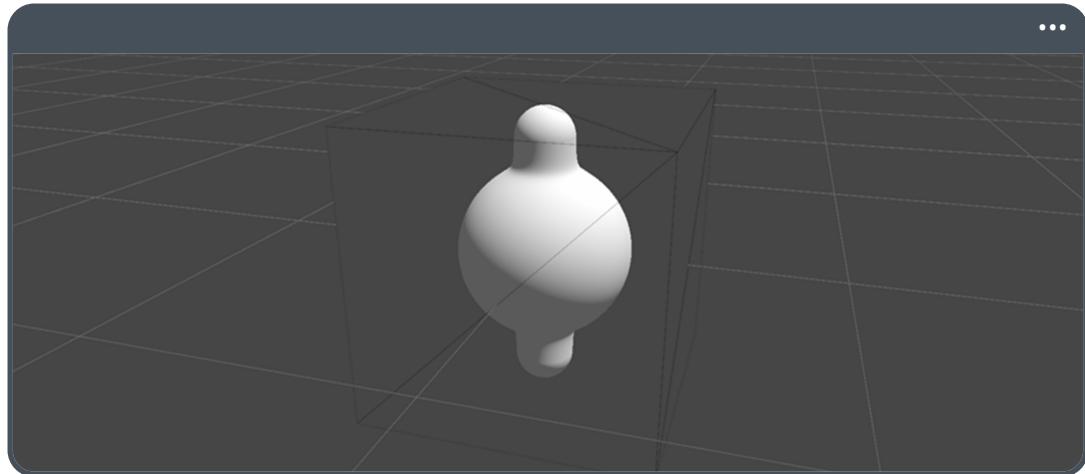
La principal ventaja de emplear `smin()` en lugar de `min()` radica en la capacidad de crear transiciones suaves entre dos SDFs. Mientras que `min()` produce una unión clara, pero con bordes angulares, `smin()` permite que las figuras se fusionen de manera más orgánica, eliminando las líneas duras y proporcionando una apariencia más natural y estéticamente agradable.

Para ver esto en práctica, nos aseguraremos de incluir la función `smooth_min()` en el script **Procedural3DShapes**, y luego nos dirigiremos al script **RayMarching**. Dentro del método `map()`, reemplazaremos la función `min()` por `smooth_min()`, como se muestra a continuación:

```
8 inline float map(float3 p)
9 {
10     const float sphere_radius = 0.3;
11     float sphere = sphere_sd(p, sphere_radius);
12
13     const float capsule_radius = 0.1;
14     const float k = 0.001;
15     const float3 capsule_p0 = float3(0.0, 0.4, 0.0);
16     const float3 capsule_p1 = float3(0.0, -0.4, 0.0);
17     float capsule = capsule_sd(p, capsule_p0, capsule_p1, capsule_radius);
18
19     return smooth_min(sphere, capsule, k);
20 }
```

Figuras tridimensionales y renderizado.

Al volver a nuestra escena, observaremos ambas figuras, la esfera y la cápsula, aparecen mezcladas, dando la sensación de que se trata de una sola forma más elaborada.



(5.4.c Mínimo suavizado entre la esfera y la cápsula)

Resumen del capítulo.

- En este capítulo, profundizamos en el uso de funciones de distancia con signo para crear geometrías tridimensionales en Unity. Comenzamos generando formas básicas como esferas y cápsulas, estableciendo las bases matemáticas necesarias para modelar figuras procedurales. Abordamos el cálculo de las normales mediante el gradiente de las SDF, esencial para dar volumen a nuestras formas mediante iluminación.
- Implementamos el método Ray Marching dentro de Shader Graph, creando nodos personalizados que permiten renderizar las figuras tridimensionales de manera eficiente. Introdujimos el modelo de iluminación de Lambert, mejorando la visualización de las formas al calcular la intensidad de la luz según el ángulo entre las normales y la dirección de la luz incidente.
- Además, exploramos la combinación de dos geometrías utilizando la función `smin` para lograr transiciones suaves entre ambas, lo que nos permitió modelar una figura más elaborada desde un punto de vista matemático. Otro aspecto interesante, fue la utilización de la instrucción `discard`, la cual nos permitió eliminar el ruido no deseado y asegurar una representación limpia de las geometrías.

Glosario.

Anti-Aliasing: Técnica utilizada para reducir el efecto de bordes irregulares en gráficos digitales.

Blackboard: Panel en Shader Graph donde se pueden definir y gestionar variables globales para los nodos.

Coordenadas cartesianas: Sistema de referencia basado en dos o tres ejes perpendiculares que permite ubicar puntos en un espacio.

Coordenadas UV: Sistema de coordenadas utilizado en gráficos por computadora para mapear texturas en superficies.

Coseno: Función trigonométrica que representa la relación entre el cateto adyacente y la hipotenusa en un triángulo rectángulo.

Curva Sinusoidal: Representación gráfica de la función seno, caracterizada por su oscilación periódica.

Custom Function: Nodo en Shader Graph que permite definir funciones personalizadas mediante código HLSL.

Desmos: Aplicación en línea utilizada para graficar ecuaciones matemáticas.

Ecuación: Expresión matemática que establece una igualdad entre dos expresiones.

Función: Relación matemática entre un conjunto de entrada y un conjunto de salida, donde cada entrada tiene una única salida.

Graph Inspector: Panel en Shader Graph donde se pueden modificar propiedades y configuraciones de los nodos.

Hipotenusa: Lado opuesto al ángulo recto en un triángulo rectángulo.

HLSL: Lenguaje de sombreado de alto nivel utilizado en gráficos por computadora para escribir shaders.

Master Stack: Conjunto final de nodos en Shader Graph que define la apariencia del material.

Material: Recurso en motores gráficos que define cómo se renderiza una superficie en función de shaders y texturas.

Método: Procedimiento o función utilizada en programación para realizar una tarea específica.

Nodo: Elemento en Shader Graph que representa una operación matemática o de procesamiento gráfico.

Pentágono: Figura geométrica de cinco lados y cinco vértices.

Procedural: Técnica en gráficos por computadora donde los datos se generan dinámicamente en lugar de almacenarse previamente.

Radianes: Unidad de medida angular utilizada en trigonometría y gráficos computacionales.

Ray Marching: Técnica de renderizado basada en la evaluación progresiva de un rayo a través de un campo de distancia.

Screen-Space: Técnica de renderizado que opera en el espacio de la pantalla, después de la transformación de la vista.

SDF: Signed Distance Field, técnica utilizada para representar formas mediante funciones de distancia.

Seno: Función trigonométrica que representa la relación entre el cateto opuesto y la hipotenusa en un triángulo rectángulo.

Shader: Programa utilizado en gráficos por computadora para determinar el color y apariencia de los píxeles.

Shader Graph: Herramienta visual en Unity que permite crear shaders mediante nodos sin necesidad de escribir código.

ShaderLab: Lenguaje utilizado en Unity para definir la estructura de los shaders.

Sub Graph: Gráfico dentro de Shader Graph que permite reutilizar un conjunto de nodos como un solo nodo.

SubShader: Sección dentro de un ShaderLab que define variantes del shader para distintas configuraciones de hardware.

Tangente: Función trigonométrica que representa la relación entre el cateto opuesto y el cateto adyacente en un triángulo rectángulo.

Trigonometría: Rama de las matemáticas que estudia las relaciones entre los ángulos y lados de los triángulos.

Unlit Shader Graph: Tipo de shader en Unity que no responde a la iluminación, ideal para efectos gráficos personalizados.

URP: Universal Render Pipeline, un sistema de renderizado en Unity optimizado para rendimiento y flexibilidad.

Variable: Espacio de almacenamiento en programación que contiene un valor que puede cambiar durante la ejecución.

World-Space: Sistema de coordenadas en gráficos computacionales donde las posiciones se expresan en relación al mundo global.

Agradecimientos.

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 Suriyashekhar | 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 Buadom | 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 Aspis | 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 | Marlion 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 | Joeper 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 | 유태양 | 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 | YoONo | Adrian | Luginis | 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 te desea éxito
en tu carrera profesional.**