

Aprende. Crea. Domina.

Visualizando Ecuaciones - Vol. 1

# Matemática Esencial para Game Devs.

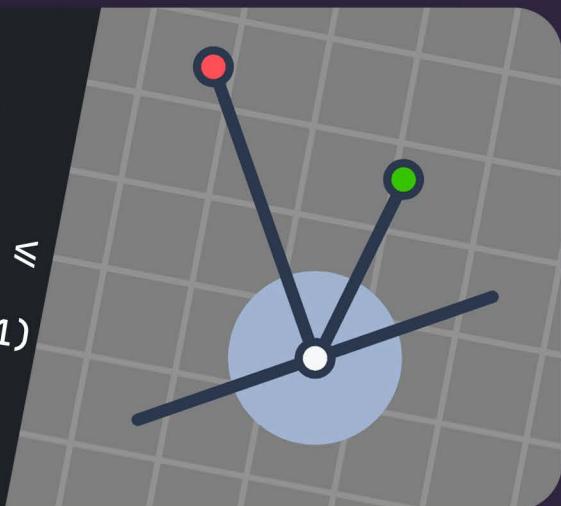
Explicaciones visuales para ayudarte a entender conceptos matemáticos clave para el desarrollo de videojuegos.

# TechnicalArtist

# GameDev

# MadeWithUnity

```
int Summation( int n )
{
    int s = 0;
    for ( int i = 1; i <= n; i++ )
        s += ( 3 * i - 1 );
    return s;
}
```



Fabrizio Espíndola.



*jettelly*

# **Visualizando ecuaciones, matemática esencial para game devs.**

Libro de explicaciones visuales que te ayudará a comprender conceptos matemáticos esenciales y aplicables al desarrollo de videojuegos.

**Autor.**

Fabrizio Espindola.

**Diseño.**

Pablo Yeber.

**Revisión gramática.**

Elena Miranda.

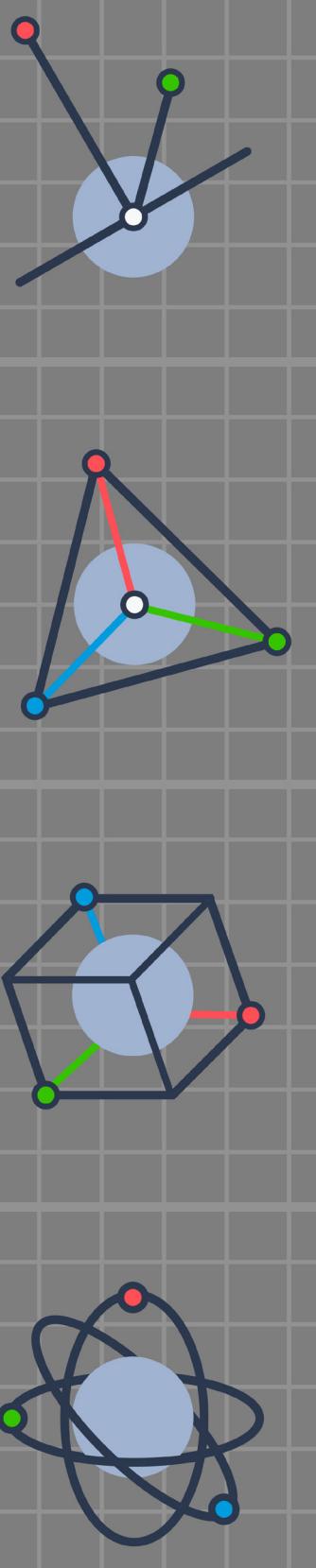
**Revisión técnica.**

Martin Molina.

## **Agradecimientos.**

Agradezco y dedico esta obra a mis padres Marcia Vivas y Victor Espíndola, por brindarme la vida; a mis hermanos Angela Espíndola, Franco Espíndola y Camilo Espíndola, por acompañarme en los momentos difíciles; a mi esposa Aida Cid e hijo Santino Espíndola, por ser mi luz y motivación; a mi amigo y colega Pablo Yeber, por acompañarme en esta gran aventura, y finalmente, a mis mentores David Sanhueza, Cristian Klett y Ewan Lee, por guiarme con sabiduría.

~ Fabrizio Espíndola.



## Contenido.

Prefacio. ....	7
<b>Capítulo 1   Producto Punto. ....</b>	<b>10</b>
1.1. Introducción a la función. ....	11
1.2. Desarrollando una herramienta en Unity. ..	17
Resumen. ....	41
<b>Capítulo 2   Producto Cruz. ....</b>	<b>42</b>
2.1. Introducción a la función. ....	43
2.2. Desarrollando una herramienta en Unity. 48	48
Resumen. ....	68
<b>Capítulo 3   Cuaterniones. ....</b>	<b>69</b>
3.1. Introducción a la función. ....	70
3.2. Desarrollando una herramienta en Unity. 79	79
Resumen. ....	98
<b>Capítulo 4   Rotación de matrices y ángulos Euler. ....</b>	<b>99</b>
4.1. Introducción a la función. ....	100
4.2. Desarrollando una herramienta en Unity. 109	109
Resumen. ....	128
Conclusión. ....	129
Glosario. ....	131
Agradecimientos especiales. ....	136

# Prefacio.

Ya sea porque deseamos generar una mecánica, un algoritmo, crear una herramienta u otros, una práctica común que nos caracteriza como desarrolladores es la acción de repasar ecuaciones matemáticas mientras intentamos resolver una dificultad en nuestro programa.

En más de una oportunidad nos hemos encontrado con el mismo desafío: ¿Cómo traduzco ciertas ecuaciones a código? Dada nuestra naturaleza, hay veces en que simplemente no recordamos los símbolos matemáticos o el orden de una operación, generando consigo desconcentración en nuestra labor o incluso frustración.

Este libro está diseñado para ayudar a los desarrolladores Unity a crear herramientas efectivas en el software para la visualización de ecuaciones en sus proyectos. A través de una combinación de teoría y ejemplos prácticos, aprenderás a aplicar los conceptos matemáticos y herramientas de programación para crear visualizaciones dinámicas y atractivas que ilustran conceptos matemáticos de una manera clara y accesible.

## Para quien es este libro.

Este libro está diseñado para desarrolladores de Unity en busca de perfeccionar su comprensión de funciones matemáticas, la visualización de ecuaciones, y la creación de herramientas profesionales. Se asume que los lectores ya poseen un conocimiento básico de Unity, por lo que no nos adentraremos en detalles sobre su interfaz.

Si bien es cierto que tener experiencia previa en lenguaje C# puede resultar útil para entender ciertas partes del contenido, no es un requisito excluyente para el lector.

Se recomienda tener una pequeña base de conocimientos en aritmética y álgebra para entender algunos de los conceptos que se abordarán a lo largo del libro.

De todas maneras, el libro incluirá revisiones de las operaciones y funciones matemáticas necesarias para comprender completamente el material que se desarrolla.

## Convenciones.

En este libro, hemos establecido algunas convenciones para resaltar ciertos elementos y hacer que la información sea más accesible. Estas convenciones incluyen el uso de comillas angulares para destacar funciones, métodos y variables, así como la forma de escribir variables numéricas y de código. Además, utilizaremos letras mayúsculas para representar ejes espaciales.

- **Resaltado de elementos:** Hemos elegido enmarcar funciones, métodos y variables entre comillas angulares (p. ej. « **Start** ») para resaltar su importancia y naturaleza técnica.
- **Variables numéricas y de código:** las variables numéricas y de código se escriben principalmente en minúsculas (p. ej. « **a** »). Esto ayuda a distinguir entre variables técnicas y valores numéricos.
- **Ejes espaciales:** Los ejes espaciales, como coordenadas, se escriben en mayúsculas (p. ej. « **XYZ** »). Esto facilita la identificación de elementos relacionados con el espacio y la geometría.

A lo largo del libro, un bloque de código se presentará en un formato especial, de la siguiente manera.

```
4  public class ExampleClass : MonoBehaviour
5  {
6      void Start ()
7      {
8          // Código aquí ...
9      }
10 }
```

Estas convenciones se han establecido para mejorar la claridad y la comprensión de la información presentada en el libro, y se mantendrán consistentes a lo largo de sus páginas.

### **Errata.**

Mientras escribimos este libro, hemos tomado precauciones para asegurar la precisión de su contenido. No obstante, es esencial recordar que somos seres humanos, y es posible que algunos puntos no estén completamente claros o que hayamos cometido errores de ortografía o gramática.

Si identificas algún error conceptual, de código u otro tipo, te agradecemos que nos envías un mensaje a la dirección de correo [contact@jettelly.com](mailto:contact@jettelly.com) con el asunto "**VE1 Errata**". De esta manera, estarás contribuyendo a otros lectores a reducir cualquier nivel de frustración y a mejorar cada edición futura de este libro en sus próximas actualizaciones.

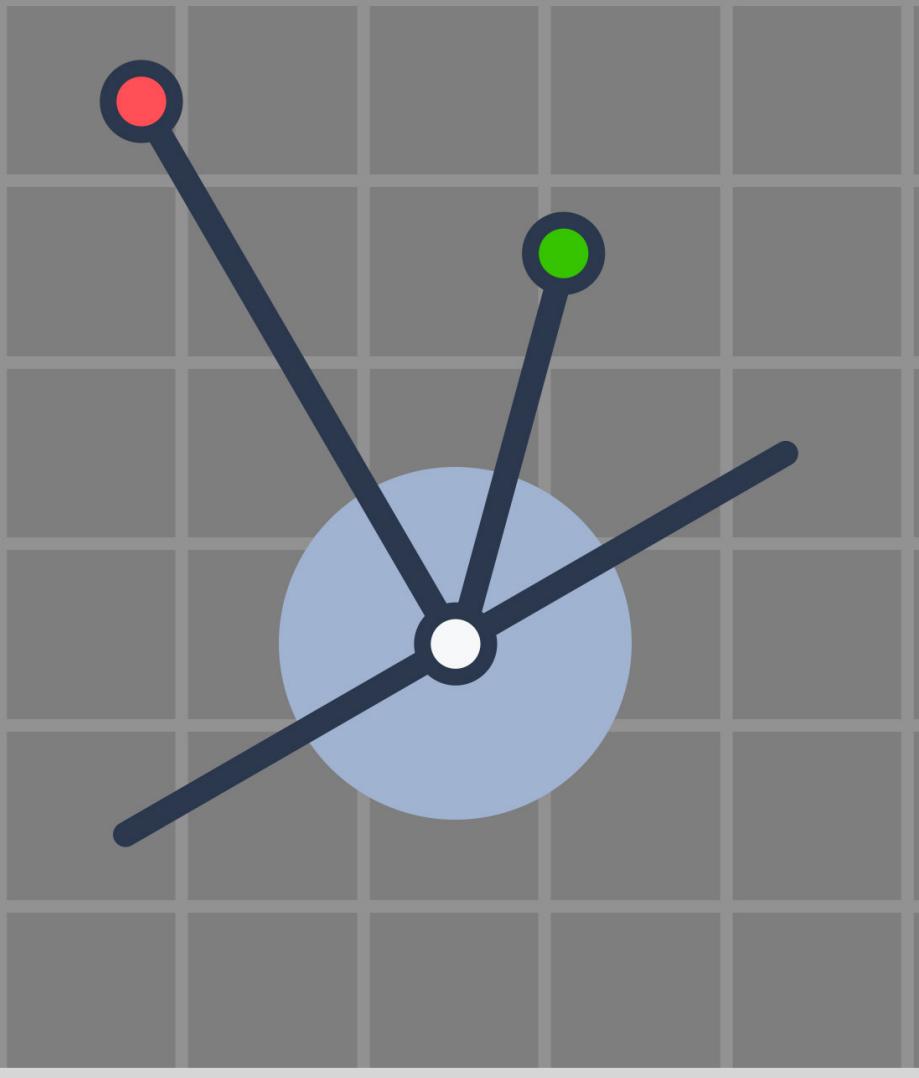
Además, si tienes sugerencias sobre secciones que consideras que podrían ser de interés para futuros lectores, no dudes en enviarnos un correo. Estaremos encantados de incluir esa información en las ediciones venideras.

### **Piratería.**

Te invitamos a apoyar a nuestro equipo. Antes de considerar la copia, reproducción o distribución de este material sin nuestro consentimiento, es importante tener en cuenta que Jettelly es un estudio independiente y autofinanciado. Cualquier práctica ilegal podría tener un impacto negativo en la integridad de nuestro equipo de desarrollo.

Este libro está protegido por derechos de autor, y nos tomamos muy en serio la defensa de nuestras licencias. Si encuentras este libro en una plataforma distinta de Jettelly o identificas una copia ilegal, te agradecemos de todo corazón que te pongas en contacto con nosotros por correo electrónico a [contact@jettelly.com](mailto:contact@jettelly.com)

(y si es posible, que adjuntes el enlace) para que podamos buscar una solución.  
Apreciamos tu cooperación y apoyo.



**Capítulo 1.**  
**Producto Punto.**

## 1.1. Introducción a la función.

Comenzaremos nuestra aventura prestando atención a la siguiente ecuación:

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

(1.1.a)

¿Puedes identificar a qué función u operación pertenece la igualdad anterior? Es bastante común encontrar este tipo de igualdades en libros orientados a la programación. Estas son utilizadas para exemplificar una función que otorga un resultado específico.

Dado el título de este capítulo, es posible que ya hayas descubierto a que operación pertenece la ecuación de la Figura 1.1.a. Tal igualdad se refiere a la definición algebraica del Producto Punto (también conocido como Producto Escalar) de dos vectores n-dimensionales «  $\mathbf{A}$  » y «  $\mathbf{B}$  » definidos en el espacio euclídeo.

Su símbolo «  $\Sigma$  » (sigma) representa la sumatoria de los términos escalares de una sucesión, es decir, una suma de valores constantes, complejos o que se encuentren dentro del conjunto de números reales, los cuales inician en «  $i$  » y finalizan en «  $n$  »; ambas variables.

Cabe recordar que una variable, como su nombre lo menciona, se refiere a un valor que varía en el tiempo, como por ejemplo la edad de una persona. La edad puede ser igual a 1, 2, 3, etcétera, pero en ningún caso será un valor constante, ¿Por qué razón? Porque el tiempo avanza, lamentable, ¿verdad?

Realizaremos el siguiente ejercicio para entender el concepto,

$$x = \sum_{i=1}^5 (3i - 1)$$

(1.1.b)

De la Figura 1.1.b, dado que « *i* » es igual a 1, la primera operación a realizar sería  $(3 * 1 - 1)$  la cual da 2 como resultado. Posteriormente, el valor de « *i* » es reemplazado por cada sucesión en el ejercicio hasta llegar a « *n* », la que en este caso es igual a 5.

La siguiente sumatoria sería  $(3 * 2 - 1)$ , y así continuamente.

$$(3 * 1 - 1) + (3 * 2 - 1) + (3 * 3 - 1) + (3 * 4 - 1) + (3 * 5 - 1)$$

(1.1.c)

Que es lo mismo decir,

$$2 + 5 + 8 + 11 + 14$$

(1.1.d)

Por tanto,

$$40 = \sum_{i=1}^5 (3i - 1)$$

(1.1.e)

Ahora, volviendo a la ecuación que se muestra en la Figura 1.1.a. ¿Cómo podríamos traducir tal igualdad? Esto es bastante sencillo: el valor de «  $n$  » se refiere a la cantidad de dimensiones que posee el vector; e «  $i$  » corresponde a una etiqueta para cada componente. Por ejemplo, para un vector de tres dimensiones tendríamos la siguiente operación:

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

(1.1.f)

Que es lo mismo decir,

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

(1.1.g)

Por tanto,

$$\mathbf{A} \cdot \mathbf{B} = (A_x * B_x) + (A_y * B_y) + (A_z * B_z)$$

(1.1.h)

Si prestamos atención a la operación de la Figura 1.1.f notaremos que la variable «  $n$  » ha sido reemplazada por un valor constante, el cual es igual a 3; número de dimensiones. Tal ecuación puede ser leída de la siguiente manera:

“El producto punto entre el vector  $\mathbf{A}$  y el vector  $\mathbf{B}$  es igual a la sumatoria de los productos de cada componente.”

La implementación en código de la sumatoria va a depender del contexto para el cual la estamos usando. Por ejemplo, podríamos utilizar un bucle « **for** » para obtener la igualdad de la Figura 1.1.b.



The screenshot shows a dark-themed code editor. On the left, there is a yellow-bordered callout box containing a mathematical summation formula:

$$\sum_{i=1}^n (3i - 1)$$

To the right of the callout box is the corresponding C++ code:

```
int Summation(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++)
    {
        s += (3 * i - 1);
    }
    return s;
}
```

(1.1.i)

Como podemos observar en la Figura anterior, el método « **Summation** » retorna 40 si « **n** » es igual a 5. Sin embargo, cuando hablamos de vectores, su implementación es distinta dado que, en este caso, estaríamos buscando un resultado dependiendo de:

- Si los vectores son puntos en el espacio.
- Si los vectores son direcciones en el espacio.
- Si deseamos obtener la proyección de un vector sobre el otro.

Geométricamente, el Producto Punto corresponde a la proyección de un vector « **A** » sobre el vector « **B** », ¿qué significa esto? Imagina que estás de pie en un día soleado y tienes una fuente de luz que proyecta tu sombra en el suelo. Ahora, piensa en dos vectores en el espacio tridimensional: vector « **A** » y vector « **B** ». El Producto Punto entre estos dos se puede interpretar como la proyección de un vector sobre el otro, similar a cómo la luz proyecta una sombra en el suelo.

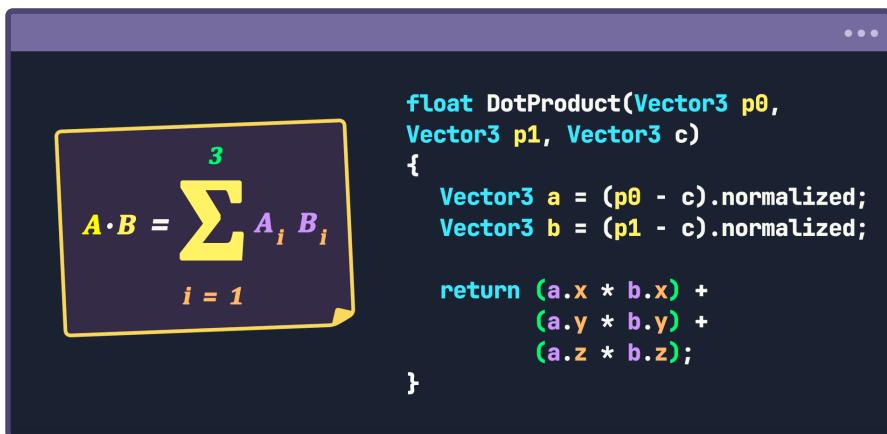
Cabe destacar que una sumatoria puede tener distintos usos según la función que esta deba cumplir. La interpretación geométrica del Producto Punto nos permite introducir otra igualdad de importancia, y es que el mismo está dado por,

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| |\mathbf{B}| \cos \theta$$

(1.1.j)

De la figura anterior, tal ecuación da el mismo resultado que la ecuación de la Figura 1.1.h, con la diferencia que, con este último podemos obtener el ángulo «  $\theta$  » entre ambos vectores.

Considerando que los vectores «  $\mathbf{A}$  » y «  $\mathbf{B}$  » de la Figura 1.1.a son direcciones en el espacio, su implementación podría ser igual a la siguiente función,



(1.1.k)

Es importante tener en cuenta que al trabajar con vectores siempre necesitaremos un punto de referencia que actúe como el origen de nuestro sistema de referencia. En la Figura 1.1.k se puede observar que en el método « **DotProduct** », además de los vectores « **a** » y « **b** », se utiliza un nuevo vector adicional llamado « **c** » como argumento, el cual actúa como punto de referencia entre « **p0** » y « **p1** ».

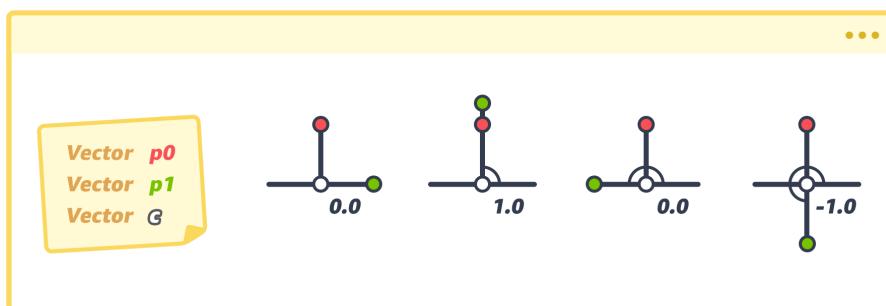
Al restar el vector « **c** » a los vectores « **p0** » y « **p1** », podemos interpretar que los vectores « **a** » y « **b** » son direcciones en el espacio. Es importante destacar que estos vectores han sido normalizados para que su magnitud sea igual a 1.

$$\begin{aligned} ||\mathbf{a}|| &= 1 \\ ||\mathbf{b}|| &= 1 \end{aligned}$$

(1.1.l)

Podríamos preguntarnos entonces, ¿Para qué nos podría ser útil el Producto Punto entre dos vectores? Esto va a depender de lo que estamos desarrollando. Por ejemplo: en una mecánica de videojuego, podríamos emplear tal función para determinar la dirección de un objeto respecto a otro.

El Producto Punto entre dos vectores unitarios retorna el valor del coseno del ángulo entre ambos, por lo que su resultado está dentro de un rango de -1f a 1f. Por lo tanto, tal resultado está determinado según la posición de cada vector respecto a un punto de referencia. Esto quiere decir que podríamos utilizar estos valores para establecer si un objeto se encuentra en frente o detrás del otro, ¿Cómo haríamos esto?

(1.1.m. <https://www.desmos.com/calculator/nmkdargld3>)

En la Figura anterior podemos observar la representación gráfica del Producto Punto entre dos vectores, donde el vector « **c** » marca el punto de referencia. Suponiendo que el vector « **p0** » es el personaje principal; y « **p1** », su enemigo, podemos señalar que:

- Si el Producto Punto retorna 1f, el enemigo se encuentra en frente o arriba de nuestro personaje.
- Si el mismo retorna -1f, entonces el enemigo está detrás o abajo.

## 1.2. Desarrollando una herramienta en Unity.

Considerando el carácter abstracto de la explicación anterior, a continuación, crearemos una pequeña herramienta, donde implementaremos la igualdad presentada en la Figura 1.1.a de la sección anterior. Tal instrumento nos ayudará a visualizar el comportamiento del Producto Punto entre dos vectores.

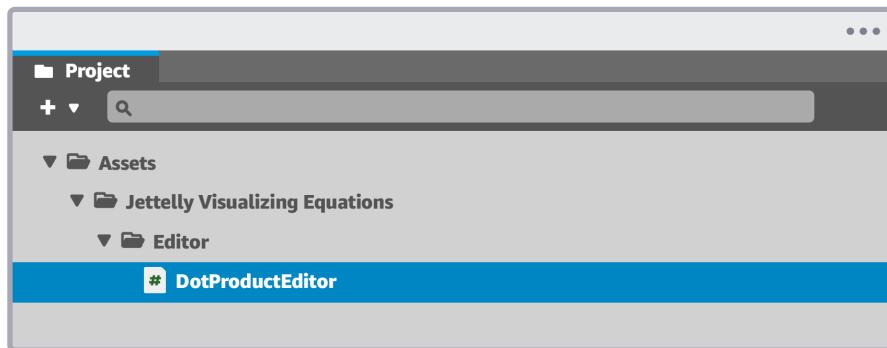
Iniciaremos el proceso yendo a nuestro proyecto (ventana Project en Unity), y crearemos un nuevo script al cual llamaremos « **DotProductEditor** ». Una vez abierto, nos aseguraremos de extender desde « **EditorWindow** » por dos razones en particular:

- Porque será una herramienta visual.
- Porque únicamente necesitaremos una instancia del mismo objeto en la ventana Scene.

En consecuencia, será importante tanto, utilizar la dependencia « **UnityEditor** » en el código, así como también, guardar el script dentro de una carpeta de tipo Editor en nuestro proyecto ¿Por qué razón? Según la documentación oficial del software:

Los scripts de tipo Editor agregan funcionalidades a Unity durante el desarrollo, pero no están disponibles en las construcciones en tiempo de ejecución.

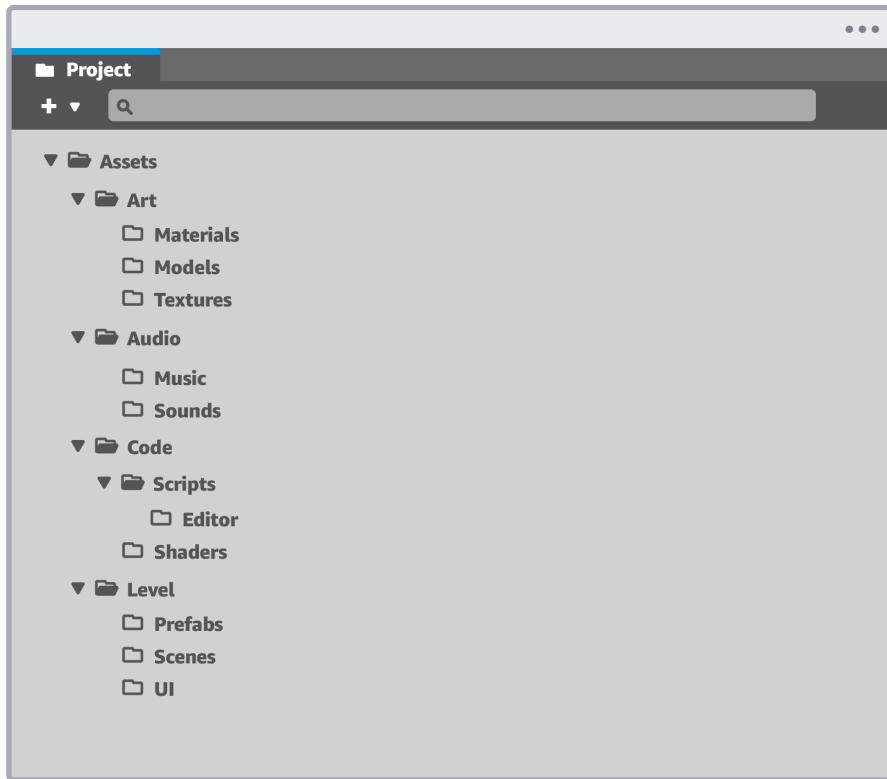
Es decir, que no podemos exportar un videojuego o aplicación con gráficos de Editor.



(1.2.a)

En Unity, es común encontrar varias carpetas y subcarpetas de tipo Editor dentro de la carpeta Assets. Aunque esta característica brinda cierta libertad al definir la organización del proyecto, también puede causar problemas si no se establece una estructura coherente. Esto puede resultar en múltiples carpetas de tipo Editor incluidas en subcarpetas en todo el proyecto, lo que dificulta la búsqueda de scripts a medida que el proyecto crece en tamaño.

Para evitar esta situación, Unity sugiere seguir "las mejores prácticas" para organizar tu proyecto y utiliza la siguiente estructura como referencia.



(1.2.b)

Cada vez que creamos un nuevo script, por defecto, extiende desde « **MonoBehaviour** » y agrega dos métodos:

- « **void Start** ».
- « **void Update** ».

Sin embargo, para este caso, estos métodos no serán de utilidad dado que, como se ha mencionado previamente, nuestro script extenderá desde « **EditorWindow** », por lo tanto, será necesario:

- Incluir la dependencia « **UnityEditor** ».
- Extender nuestro script desde « **EditorWindow** ».
- Eliminar las funciones por defecto.

La clase « **EditorWindow** » viene incluida en la dependencia « **UnityEditor** ». Esta última contiene varias clases que son de utilidad en el desarrollo de herramientas, entre las cuales podemos destacar:

- « **EditorGUILayout** ».
- « **Handles** ».
- « **Undo** » entre otros.

```
1  using UnityEngine;
2  using UnityEditor;
3
4  public class DotProductEditor : EditorWindow
5  {
6
7  }
```

Para comenzar oficialmente el proceso de desarrollo, debemos agregar algún método que nos permita desplegar una ventana para nuestra herramienta. Para ello podemos realizar los siguientes pasos según indica Unity en su plataforma web. Estos son:

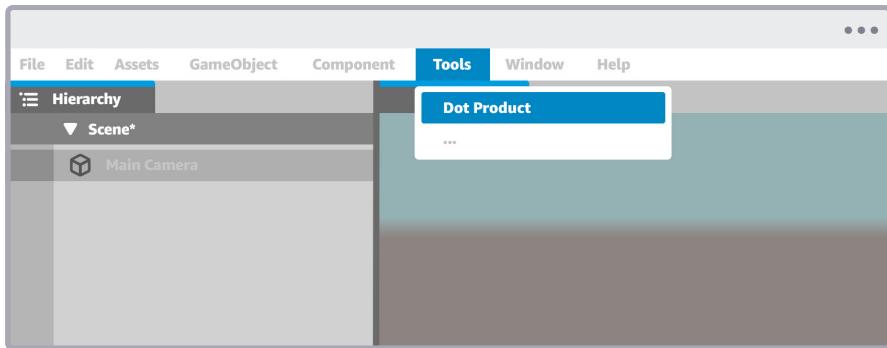
- Declarar un método público y estático para mostrar una ventana en el Editor.
- Agregar el atributo « **MenuItem** » sobre el método, mencionado la ruta de menú; donde deseamos listar nuestra herramienta.
- Crear y mostrar la nueva ventana en el menú principal.



```
4  public class DotProductEditor : EditorWindow
5  {
6      [MenuItem("Tools/Dot Product")]
7      public static void ShowWindow()
8      {
9          DotProductEditor window = (DotProductEditor) GetWindow
10             (typeof(DotProductEditor), true, "Dot Product");
11         window.Show();
12     }
13 }
```

Siguiendo la línea de código número 9, la función estática « **GetWindow** »; quien devuelve el primer « **EditorWindow** » de tipo « **t** » que se encuentra actualmente en la pantalla, posee hasta cuatro argumentos, de los cuales hemos utilizado tres de ellos en el ejemplo. El primero se refiere al tipo de ventana, el cual corresponde a « **DotProductEditor** ». El segundo es un valor booleano, el cual determina si nuestra ventana será emergente o no, y finalmente el último argumento corresponde al título que tendrá la ventana en la parte superior de la misma.

Si todo ha resultado de manera óptima, en Unity aparecerá un nuevo menú llamado "Tools" el cual corresponde a la definición que realizamos previamente desde el atributo « **MenuItem** », en la línea de código número 6. Desde ahí podremos acceder a la ventana Dot Product, la cual de momento no realiza acción alguna.



(1.2.c)

Teniendo en cuenta que nuestra herramienta se utilizará para representar el comportamiento del Producto Punto, será necesario declarar tres vectores en nuestro código « **p0** », « **p1** »; y « **c** » como punto de referencia. Además, para proyectar los vectores en su respectiva ventana y modificar sus valores de manera dinámica, será necesario declarar algunas propiedades del tipo « **SerializedProperty** ». De esta manera, podremos asegurarnos de que nuestra herramienta sea lo suficientemente flexible y eficiente para representar el comportamiento del Producto Punto correctamente.

Será esencial incluir algunas funciones en nuestro código para ver los vectores en acción. Al crear una herramienta, es importante tener en cuenta que tanto la interfaz de usuario (GUI) como los gráficos que se agreguen a la escena deben programarse por separado. Como resultado, habrá algunos valores que deberán inicializarse, como la posición inicial de los vectores. Para lograr esto, agregaremos los siguientes métodos en nuestro código:

```
4  public class DotProductEditor : EditorWindow
5  {
6      public Vector3 m_p0;
7      public Vector3 m_p1;
8      public Vector3 m_c;
9
10     private SerializedObject obj;
11     private SerializedProperty propP0;
12     private SerializedProperty propP1;
13     private SerializedProperty propC;
14
15     [MenuItem("Tools/Dot Product")]
16     public static void ShowWindow()
17     {
18         DotProductEditor window = (DotProductEditor) GetWindow
19             (typeof (DotProductEditor), true, "Dot Product");
20         window.Show();
21     }
22 }
```

- « **OnGUI** », nos ayudará a desplegar información y a manejar eventos sobre nuestra herramienta en la ventana Dot Product.
- « **SceneGUI** », corresponde a nuestra propia versión del método « **Editor.OnSceneGUI** », el cual permite manejar eventos en la ventana Scene.
- Otro método que incluiremos será « **OnEnable** », este lo utilizaremos para inicializar valores cuando la herramienta se encuentre activa.
- Finalmente, « **OnDisable** », lo emplearemos únicamente en la cancelación de suscripciones a eventos.

```
22     private void OnEnable()
23     {
24
25     }
26
27     private void OnDisable()
28     {
29
30     }
31
32     private void OnGUI()
33     {
34
35     }
36
37     private void SceneGUI(SceneView view)
38     {
39
40     }
41 }
42
```

Cabe destacar que tanto el método « **OnEnable** », como « **OnDisable** » y « **OnGUI** », pertenecen a « **EditorWindow** », el cual hereda desde « **ScriptableObject** », es decir, que son nativos de Unity. En el ejercicio anterior podemos verlas implementadas en las líneas de código 22, 27 y 32.

Un factor por considerar es que « **SceneGUI** » debe ser llamado cada vez que la ventana Scene es actualizada. Sin embargo, al no ser nativo, va a requerir de un evento para su correcto funcionamiento. Precisamente por este motivo, posee un argumento de tipo « **SceneView** » el cual admite varias configuraciones, entre ellas; suscribirse a eventos.

```
22     private void OnEnable()
23     {
24         SceneView.duringSceneGui += SceneGUI;
25     }
26
27     private void OnDisable()
28     {
29         SceneView.duringSceneGui -= SceneGUI;
30     }
31
32 >     private void OnGUI() ...
36
37     private void SceneGUI(SceneView view)
38     {
39         Debug.Log("Being updated!");
40     }
41 }
42
```

Tal como su nombre lo menciona, el evento « **SceneView.duringSceneGui** » (líneas 24 y 29) se actualiza durante la utilización de la ventana Scene. En consecuencia, si activamos nuestra ventana Dot Product y movemos el cursor dentro del área de la ventana Scene, podremos debuggear la cantidad de veces que es llamado el método « **SceneGUI** ».

El siguiente paso por realizar sería dibujar los vectores en la ventana Scene mediante el uso del método « **Handles.FreeMoveHandle** ». No obstante, debemos recordar que los vectores « **m\_p0** », « **m\_p1** » y « **m\_c** » declarados anteriormente, aún no han sido inicializados, por ende, si realizamos el proceso en este momento aparecerán todos en la misma posición, la cual corresponde a "cero" en todas sus coordenadas. Para evitar este conflicto, inicializaremos los vectores en el método « **OnEnable** »:

```

22  private void OnEnable()
23  {
24      if (m_p0 == Vector3.zero && m_p1 == Vector3.zero)
25      {
26          m_p0 = new Vector3(0.0f, 1.0f, 0.0f);
27          m_p1 = new Vector3(0.5f, 0.5f, 0.0f);
28          m_c = Vector3.zero;
29      }
30
31      SceneView.duringSceneGui += SceneGUI;
32  }
33

```

Los valores asignados a las coordenadas de « **m\_p0** » y « **m\_p1** » establecen un punto de referencia para llevar a cabo el ejercicio, de modo que, pueden ser ajustados a gusto debido a que no generarán cambios en el resultado final de la herramienta que estamos desarrollando.

Una vez inicializados los valores podemos continuar con el proceso de pintar cada vector a modo de « **Gizmos** » en la ventana Scene. Como se mencionó anteriormente, será necesario utilizar la función « **Handles.FreeMoveHandle** » debido a que este retorna a una nueva posición según la interacción del usuario con el "Handler" respectivo (punto de color que aparece en escena). Sin embargo, para evitar un código repetitivo, implementaremos un nuevo método en nuestro programa el cual se encargará de llevar a cabo el proceso:

```
44 void SceneGUI(SceneView view)
45 {
46     Handles.color = Color.red;
47     Vector3 p0 = SetMovePoint(m_p0);
48     Handles.color = Color.green;
49     Vector3 p1 = SetMovePoint(m_p1);
50     Handles.color = Color.white;
51     Vector3 c = SetMovePoint(m_c);
52 }
53
54 Vector3 SetMovePoint(Vector3 pos)
55 {
56     float size = HandleUtility.GetHandleSize(Vector3.zero) * 0.15f;
57     return Handles.FreeMoveHandle(pos, Quaternion.identity,
58                                     size, Vector3.zero, Handles.SphereHandleCap);
59 }
```

El método « **SetMovePoint** » (línea de código 54) toma una posición inicial como argumento y retorna a una nueva posición según la interacción del usuario con el punto de color que aparece en la ventana Scene. Si prestamos atención a las líneas de código 47, 49 y 51, notaremos que se han declarado e inicializado tres nuevos vectores « **p0** », « **p1** » y « **c** » utilizando el método mencionado previamente.

Si volvemos a Unity e intentamos mover los puntos cliqueando y arrastrando, notaremos que ninguno de ellos cambia de posición. Esto ocurre principalmente porque no hemos devuelto los valores de retorno a sus semejantes globales. Ahora, considerando una cuestión de "optimización", llevaremos a cabo el proceso únicamente cuando exista una diferencia entre los vectores globales y de aquellos declarados dentro del método « **SceneGUI** ». En consecuencia, será esencial utilizar una condicional para determinar si es que existe una diferencia entre semejantes, de esta manera, asignaremos los nuevos valores únicamente cuando sea necesario.

```
44 void SceneGUI(SceneView view)
45 {
46     Handles.color = Color.red;
47     Vector3 p0 = SetMovePoint(m_p0);
48     Handles.color = Color.green;
49     Vector3 p1 = SetMovePoint(m_p1);
50     Handles.color = Color.white;
51     Vector3 c = SetMovePoint(m_c);
52
53     if (m_p0 != p0 || m_p1 != p1 || m_c != c)
54     {
55         m_p0 = p0;
56         m_p1 = p1;
57         m_c = c;
58
59         Repaint();
60     }
61 }
62
```

Si prestamos atención a la línea de código 53, observaremos que los vectores « `m_p0` », « `m_p1` » y « `m_c` » son actualizados cuando sus valores presentan una diferencia respecto a su vector semejante. En el mismo proceso se invoca al método « `Repaint` » (línea de código 59) el cual se encarga de actualizar los valores de cada vector en la ventana Dot Product, después de interactuar con ellos.

Considerando las propiedades de tipo « `SerializedProperty` », las cuales son representaciones de campos serializados o propiedades en la ventana de Inspector; para que el valor actual de cada vector « `p0` », « `p1` » y « `c` » sea mostrado en la ventana Dot Product, debemos hacer uso de la función « `FindProperty` » el cual toma una ruta de tipo « `string` » como argumento, y retorna un objeto de tipo « `SerializedProperty` ».

```

22  private void OnEnable()
23  {
24      if (_p0 == Vector3.zero && _p1 == Vector3.zero)
25      {
26          _p0 = new Vector3(0.0f, 1.0f, 0.0f);
27          _p1 = new Vector3(0.5f, 0.5f, 0.0f);
28          _c = Vector3.zero;
29      }
30
31      obj = new SerializedObject(this);
32      propP0 = obj.FindProperty("m_p0");
33      propP1 = obj.FindProperty("m_p1");
34      propC = obj.FindProperty("m_c");
35
36      SceneView.duringSceneGui += SceneGUI;
37  }
38

```

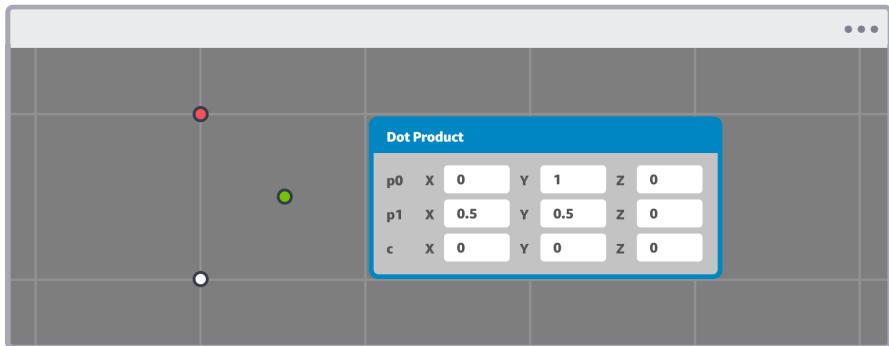
Ahora, únicamente faltaría desplegar estas propiedades en la ventana de Dot Product. Para ello será necesario ir al método « **OnGUI** », y considerar al menos tres factores:

- Llamar al método « **SerializedObject.Update** » para actualizar la representación de los objetos serializados.
- Crear un campo en la ventana Dot Product para cada propiedad. Esto podemos lograrlo utilizando la función « **EditorGUILayout.PropertyField** ».
- Actualizar la ventana Scene cuando existan cambios de valor en las propiedades que estamos desplegando. Esto podemos lograrlo mediante la función « **SerializedObject.ApplyModifiedProperties** ».

```
44  private void OnGUI()
45  {
46      obj.Update();
47
48      DrawBlockGUI("p0", propP0);
49      DrawBlockGUI("p1", propP1);
50      DrawBlockGUI("c", propC);
51
52      if (obj.ApplyModifiedProperties())
53      {
54          SceneView.RepaintAll();
55      }
56  }
57
58  void DrawBlockGUI(string lab, SerializedProperty prop)
59  {
60      EditorGUILayout.BeginHorizontal("box");
61      EditorGUILayout.LabelField(lab, GUILayout.Width(50));
62      EditorGUILayout.PropertyField(prop, GUIContent.none);
63      EditorGUILayout.EndHorizontal();
64  }
65
```

Nótese que entre las líneas de código 58 a 64 se ha agregado un nuevo método llamado « **DrawBlockGUI** » el cual posee dos argumentos: un texto de tipo « **string** » y una propiedad de tipo « **SerializedProperty** ». Su función implementa una estructura que ayuda en la organización de las propiedades que están siendo desplegadas en la ventana Dot Product. La razón de esta función es netamente de "optimización" nuevamente, dado que, de otra manera, obtendríamos un código repetitivo tras usar la misma estructura interna para cada propiedad en el método « **OnGUI** ».

Si volvemos a Unity y seleccionamos nuestra herramienta, seremos capaces de modificar los valores de cada vector directamente desde su respectiva propiedad.



(1.2.d)

Los vectores pueden ser modificados ya sea interactuando con ellos a través del Handler o desde la ventana Dot Product.

Hasta este punto, hemos concentrado nuestras energías en la creación de gráficos para nuestros vectores. Sin embargo, la razón de todo lo mencionado anteriormente tiene como finalidad mejorar nuestra comprensión sobre el comportamiento del Producto Punto sobre dos vectores. Por lo tanto, continuaremos implementando al método mencionado en la Figura 1.1.k de la sección anterior.

```

91 float DotProduct(Vector3 p0, Vector3 p1, Vector3 c)
92 {
93     Vector3 a = (p0 - c).normalized;
94     Vector3 b = (p1 - c).normalized;
95
96     return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);
97 }
98

```

Considerando que el valor de retorno del método « **DotProduct** » debe ser mostrado en alguna parte de nuestra herramienta, sería ideal desplegar un texto en la ventana Scene que muestre el resultado en tiempo real del Producto Punto entre « **A** » y « **B** ». Por esta razón agregaremos una nueva variable global de tipo « **GUIStyle** » y modificaremos: su tamaño de fuente, tipo de fuente y color en el método « **OnEnable** » para obtener un texto con mayor presencia gráfica.

```
15     private GUIStyle guiStyle = new GUIStyle();
16
17     [MenuItem("Tools/Dot Product")]
18 >    public static void ShowWindow() ...
19
20     private void OnEnable()
21     {
22         if (m_p0 == Vector3.zero && m_p1 == Vector3.zero)
23         {
24             m_p0 = new Vector3(0.0f, 1.0f, 0.0f);
25             m_p1 = new Vector3(0.5f, 0.5f, 0.0f);
26             m_c = Vector3.zero;
27         }
28
29         obj = new SerializedObject(this);
30         propP0 = obj.FindProperty("m_p0");
31         propP1 = obj.FindProperty("m_p1");
32         propC = obj.FindProperty("m_c");
33
34         guiStyle.fontSize = 25;
35         guiStyle.fontStyle = FontStyle.Bold;
36         guiStyle.normal.textColor = Color.white;
37
38         SceneView.duringSceneGui += SceneGUI;
39     }
40 }
```

Del ejemplo anterior, en la línea de código 15, se ha declarado una nueva variable de tipo « **GUIStyle** » llamada « **guiStyle** » en nuestro código. Esta cumple la función de "estilizar" el texto que desplegaremos en la ventana Scene. Luego, se han modificado los valores mencionados anteriormente en las líneas 38, 39 y 40. Continuaremos declarando e implementando un nuevo método al cual llamaremos « **DrawLabel** ». Como su nombre lo indica, estará a cargo de dibujar tanto el valor resultante del Producto Punto, como las líneas que conectan a cada vector respectivamente en la ventana Scene.

```
105 void DrawLabel(Vector3 p0, Vector3 p1, Vector3 c)
106 {
107     Handles.Label(c, DotProduct(p0, p1, c).ToString("F1"),
108     guiStyle);
109     Handles.color = Color.black;
110     Handles.DrawAAPolyLine(3f, p0, c);
111     Handles.DrawAAPolyLine(3f, p1, c);
112 }
```

Podemos deducir con facilidad que los argumentos del método « **DrawLabel** » (línea 105) corresponden a los puntos o vectores que han sido declarados previamente en el método « **SceneGUI** ».

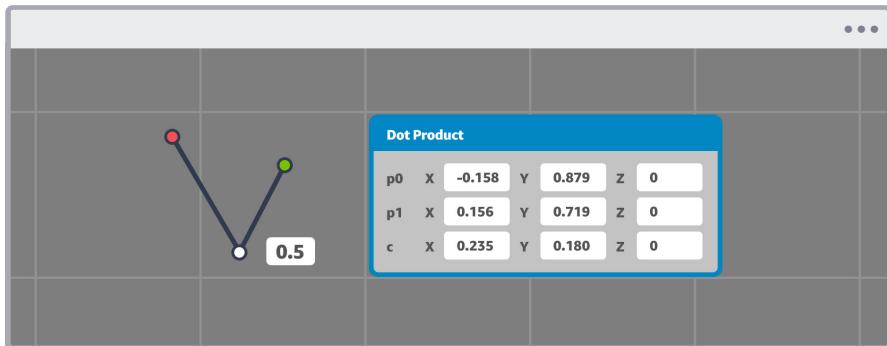
En la línea de código 107 se encuentra la función « **Handles.Label** », la cual crea un texto considerando:

- Una posición espacial.
- Un nombre para el texto.
- Un estilo.

Nótese que se ha utilizado al método « **DotProduct** » para definir el nombre del texto, este último retorna un número de coma flotante, es decir, con decimales. Por esta razón, se ha utilizado la función « **ToString** » para cambiar su formato; y « **F1** », para que muestre únicamente un decimal. La función « **Handles.DrawAAPolyLine** » está a cargo de generar líneas gráficas estilizadas (anti-aliasing) en los puntos definidos en sus argumentos.

Habiendo realizado el proceso, debemos incluir el método « **DrawLabel** » al final de « **SceneGUI** » y pasar los vectores « **p0** », « **p1** » y « **c** » como argumentos. Si volvemos a Unity, podremos apreciar cómo cambia el valor del « **Label** » según la posición de « **m\_p0** » y « **m\_p1** », respecto a « **m\_c** ».

```
72 void SceneGUI(SceneView view)
73 {
74     Handles.color = Color.red;
75     Vector3 p0 = SetMovePoint(m_p0);
76     Handles.color = Color.green;
77     Vector3 p1 = SetMovePoint(m_p1);
78     Handles.color = Color.white;
79     Vector3 c = SetMovePoint(m_c);
80
81 >     if (m_p0 != p0 || m_p1 != p1 || m_c != c) ...
82
83
84     DrawLabel(p0, p1, c);
85
86 }
```

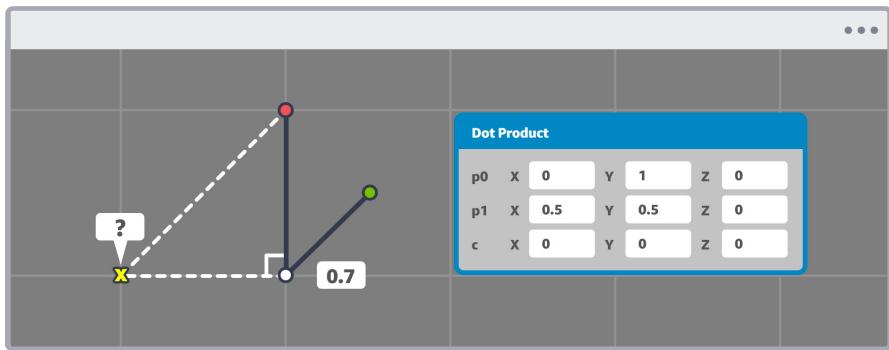


(1.2.e)

Como se ha mencionado anteriormente, el Producto Punto normalizado retorna un número real en un rango que va desde -1f a 1f respecto a un punto de referencia. Si volvemos a Unity y modificamos la posición de los vectores en la ventana Scene, notaremos que:

- Independientemente de la posición de « **c** », el valor de retorno será 1f si « **p0** » y « **p1** » se encuentran en la misma dirección.
- De la misma manera, el valor de retorno será -1f si « **p0** » y « **p1** » se encuentran en dirección opuesta.
- El valor de retorno será "cero" si « **p0** » y « **p1** » son perpendiculares.

La funcionalidad de nuestra herramienta está lista hasta este punto, sin embargo, agregaremos una superficie; una línea paralela al vector « **c** » a modo de "característica" la cual nos ayudará a visualizar de mejor manera la diferencia de posiciones entre « **p0** » y « **p1** ». Para llevar a cabo este proceso nuevamente haremos uso del método « **Handles.DrawAAPolyLine** » el cual, como ya sabemos, requiere de dos vectores para generar los gráficos.



(1.2.f)

Como podemos apreciar en la Figura anterior, se ha referenciado un nuevo vector al costado izquierdo del vector « **C** » (punto blanco), generando consigo un triángulo rectángulo. Para determinar una nueva posición debemos tomar en consideración:

- La dirección entre « **pθ** » y « **c** ».
- El arctangente del cateto opuesto, partido por el cateto adyacente.
- La rotación del ángulo resultante.

Para ello será necesario aplicar la siguiente ecuación:

$$\mathbf{R} = \mathbf{C} + \mathbf{HP}$$

(1.2.g)

De la Figura 1.2.g, « **C** » se refiere al vector que hemos definido como punto central, « **H** » corresponde a la rotación en Quaternion del ángulo resultante al calcular el arctangente de la dirección del vector « **pθ** - **c** ». Finalmente, « **P** » es un vector con módulo igual a la distancia que existe entre « **C** » y el nuevo vector « **R** ».

Considerando que un nuevo vector generaría un triángulo rectángulo si se encuentra a la derecha o izquierda de « **C** », podríamos utilizar la siguiente razón trigonométrica para determinar su ángulo:

$$\theta = \text{atan} \frac{oc_y}{ac_x}$$

(1.2.h)

Pero ¿Cómo haríamos esto? Para ello, volveremos a nuestro código y agregaremos un nuevo método al cual llamaremos « **WorldRotation** »,

```
121 Vector3 WorldRotation (Vector3 p, Vector3 c, Vector3 pos)
122 {
123     return Vector3.zero;
124 }
125
```

De momento nuestro método no realiza operación alguna. Sin embargo, si prestamos atención a sus argumentos notaremos que posee tres vectores; tres posiciones de entrada.

Dado que el arcotangente requiere del cateto opuesto y del cateto adyacente, la primera operación que deberíamos realizar dentro del método sería calcular la dirección entre « **p** » y « **c** ».

```
121 Vector3 WorldRotation (Vector3 p, Vector3 c, Vector3 pos)
122 {
123     Vector2 dir = (p - c).normalized;
124
125     return Vector3.zero;
126 }
127
```

Posteriormente, habría que calcular el arcotangente como se muestra en la Figura 1.2.h. Para ello podemos emplear el método estático « **Atan2** » el cual retorna el ángulo en radianes. Cabe mencionar que, será necesario utilizar la función « **Mathf.Rad2Deg** » para transformar el resultado a grados.

```
121 Vector3 WorldRotation (Vector3 p, Vector3 c, Vector3 pos)
122 {
123     Vector2 dir = (p - c).normalized;
124     float ang = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
125
126     return Vector3.zero;
127 }
128
```

Finalmente, debemos pasar el ángulo desde grados a rotación, para ello será necesario entender la naturaleza de los "Quaternions".

Los Quaternions fueron descubiertos por William Rowan Hamilton en el año 1843. Estos son utilizados para representar rotaciones, de hecho, la propiedad « **Rotation** » en Unity (aquella que se encuentra en el componente Transform) corresponde a este tipo de dato.

Continuando con la operación, simplemente debemos transformar el ángulo obtenido previamente a Quaternion. Para ello podemos utilizar la función « **AngleAxis** » la cual crea una rotación en base a un eje. Considerando que nuestra herramienta 3D la estamos diseñando dentro de un plano bidimensional, utilizaremos el eje « **Z** » para su rotación.

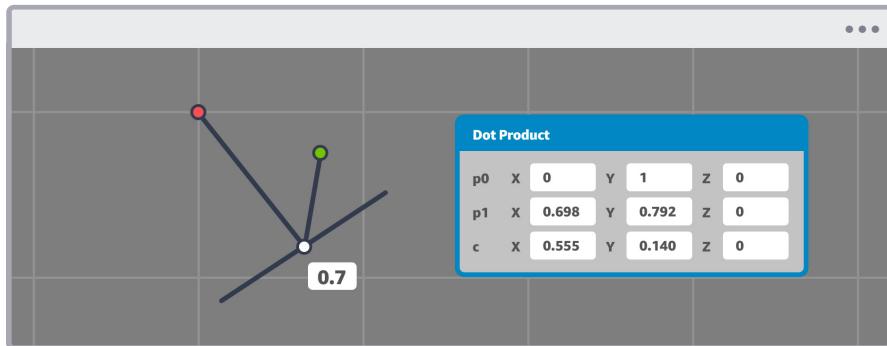
```
121 Vector3 WorldRotation (Vector3 p, Vector3 c, Vector3 pos)
122 {
123     Vector2 dir = (p - c).normalized;
124     float ang = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
125     Quaternion rot = Quaternion.AngleAxis(ang, Vector3.forward);
126
127     return c + rot * pos;
128 }
```

Como podemos ver en la línea de código 127, hemos aplicado la ecuación detallada en la Figura 1.2.g. Únicamente faltaría crear dos nuevos vectores, uno a la derecha y otro a la izquierda de « **c** », y agregarlos al método « **DrawLabel** » para que puedan ser apreciados gráficamente en la ventana Scene.

```
107 void DrawLabel(Vector3 p0, Vector3 p1, Vector3 c)
108 {
109     Handles.Label(c, DotProduct(p0, p1, c).ToString("F1"),
110                 guiStyle);
111     Handles.color = Color.black;
112
113     Vector3 cLef = WorldRotation(p0, c, new Vector3(0f, 1f, 0f));
114     Vector3 cRig = WorldRotation(p0, c, new Vector3(0f, -1f, 0f));
115
116     Handles.DrawAAPolyLine(3f, p0, c);
117     Handles.DrawAAPolyLine(3f, p0, c);
118     Handles.DrawAAPolyLine(3f, c, cLef);
119     Handles.DrawAAPolyLine(3f, c, cRig);
120 }
```

Del ejemplo anterior, si prestamos atención a las líneas de código 112 y 113, notaremos que se han declarado e inicializado dos vectores « **cLef** » y « **cRig** », los cuales generan un desplazamiento en una unidad de longitud en la coordenada « **Y** ».

Para finalizar el ejercicio, tales vectores han sido utilizados por la función « **Handles.DrawAAPolyLine** » en las líneas de código 117 y 118, generando consigo dos rectas gráficas que simulan una "superficie".



(1.2.i)

Si volvemos a Unity, podremos apreciar que « **cLef** » y « **cRig** » mantienen una posición de 90° respecto a la recta « **p0** » y « **c** ».

## Resumen.

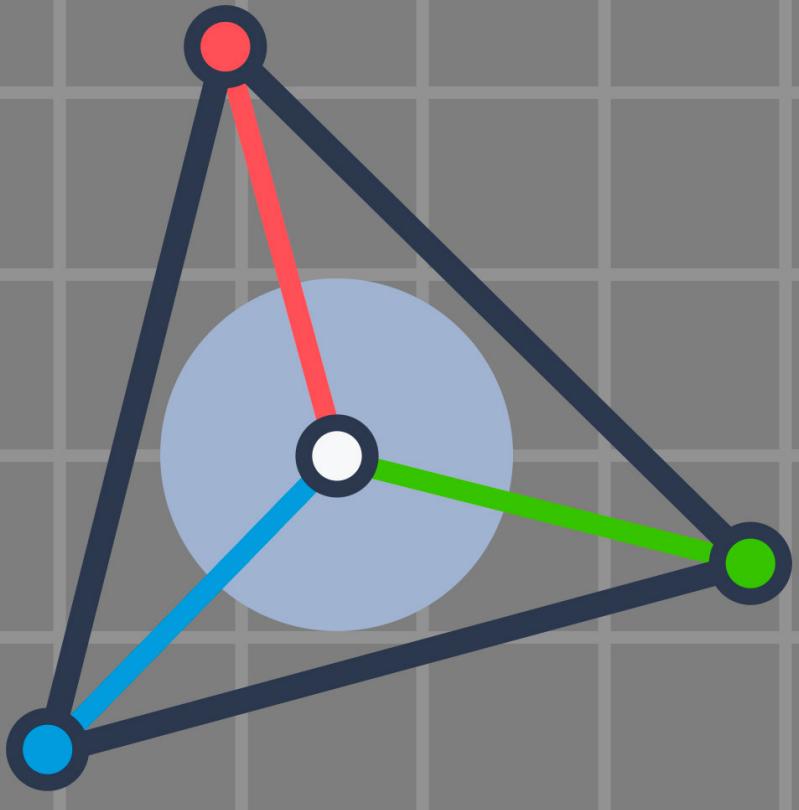
En este capítulo, hemos aprendido acerca de la naturaleza del Producto Punto y sus diversas aplicaciones, abordando desde símbolos matemáticos hasta el desarrollo de ejercicios simples.

Además, hemos comprendido la diferencia entre un punto y una dirección en el espacio mientras se detalla el funcionamiento del método « **Summation** » en la sección uno del primer capítulo.

Posteriormente, abordamos otros puntos relevantes, como la implementación del método « **DotProduct** » y la normalización de los vectores para obtener valores dentro de un rango entre -1f y 1f.

En la segunda sección, nos sumergimos por completo en C# para abordar clases como « **EditorWindow** », que nos permite crear ventanas personalizadas en el editor de Unity.

Finalmente, desarrollamos una herramienta gráfica basándonos en el conocimiento adquirido en la sección anterior, lo que nos permitió visualizar el comportamiento del Producto Punto entre dos vectores.



**Capítulo 2.**  
**Producto Cruz.**

## 2.1. Introducción a la función.

Continuaremos nuestra aventura hablando de una igualdad fundamental en programación: el Producto Cruz, también conocido como Producto Vectorial. A diferencia del Producto Punto, esta operación retorna un vector de tres dimensiones y es ampliamente utilizada en diversas aplicaciones.

Vamos a prestar atención a la siguiente fórmula para entender su definición:

$$\mathbf{P} \times \mathbf{Q} = (P_y Q_z - P_z Q_y, P_z Q_x - P_x Q_z, P_x Q_y - P_y Q_x)$$

(2.1.a)

Es importante destacar que es bastante común confundir el símbolo «  $\times$  » con el de multiplicación cuando nos estamos familiarizando con la matemática vectorial. Sin embargo, es importante tener en cuenta que este símbolo se refiere específicamente al Producto Cruz y no a la operación de multiplicación.

Como el Producto Cruz devuelve un vector tridimensional, podemos inferir que las operaciones dentro de los paréntesis en la Figura 2.1.a corresponden a los componentes del nuevo vector. En otras palabras,

$$(\mathbf{P} \times \mathbf{Q})_x = (P_y Q_z - P_z Q_y)$$

$$(\mathbf{P} \times \mathbf{Q})_y = (P_z Q_x - P_x Q_z)$$

$$(\mathbf{P} \times \mathbf{Q})_z = (P_x Q_y - P_y Q_x)$$

(2.1.b)

Para entender el concepto de mejor manera, realizaremos el siguiente ejercicio: Vamos a definir dos vectores, «  $\mathbf{A}$  » y «  $\mathbf{B}$  », los cuales tendrán las siguientes componentes,

$$\begin{aligned}\mathbf{A} &= (0, 1, 0) \\ \mathbf{B} &= (1, 0, 0)\end{aligned}$$

(2.1.c)

Con la fórmula presentada en la Figura 2.1.a, podemos definir un tercer vector «  $\mathbf{C}$  » que será el resultado de Producto Cruz entre los vectores mencionados previamente.

$$\mathbf{C} = (A_y B_z - A_z B_y, A_z B_x - A_x B_z, A_x B_y - A_y B_x)$$

(2.1.d)

Si reemplazamos por las componentes dadas en la Figura 2.1.c, obtenemos,

$$\mathbf{C} = (1 * 0 - 0 * 0, 0 * 1 - 0 * 0, 0 * 0 - 1 * 1)$$

(2.1.e)

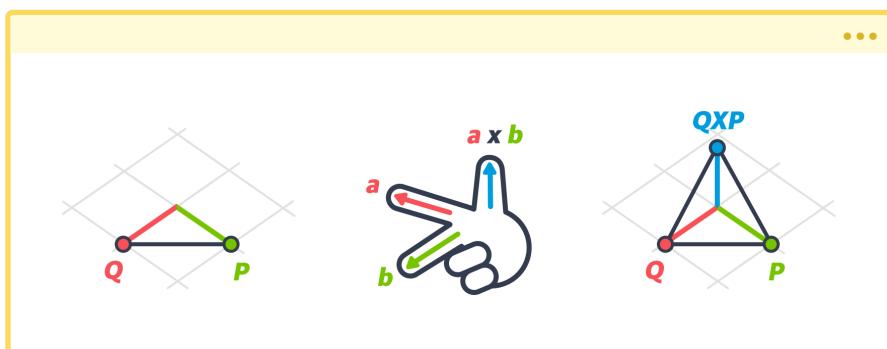
Por lo tanto,

$$\begin{aligned}\mathbf{C} &= (0 - 0, 0 - 0, 0 - 1) \\ \mathbf{C} &= (0, 0, -1)\end{aligned}$$

(2.1.f)

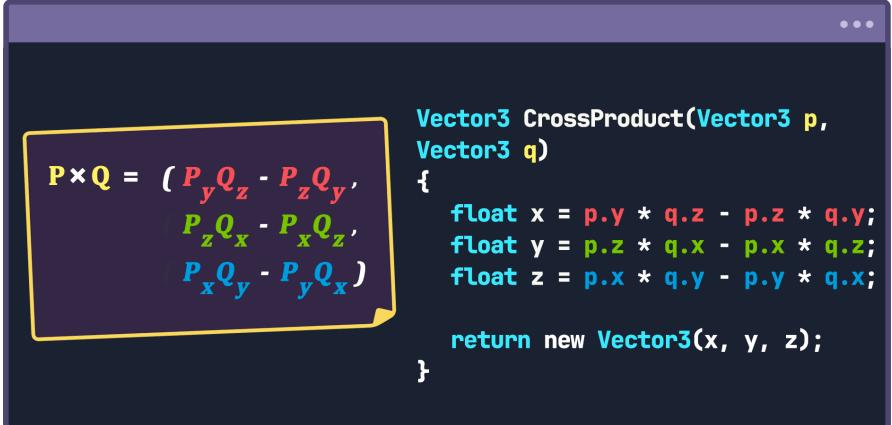
El vector resultante «  $C$  » tiene la particularidad de que es perpendicular a los vectores originales. Además, el módulo del mismo está relacionado con el área del paralelogramo que los dos vectores «  $A$  » y «  $B$  » generan. Cabe destacar que su dirección se puede determinar utilizando la regla de "la mano derecha", la cual, establece que, al colocar tu mano derecha extendida con los dedos en una posición específica, el pulgar señalará la dirección o eje de giro del vector resultante.

¿Para qué podríamos utilizar esta operación? En diversas aplicaciones, por ejemplo: calcular la normal de un vértice o superficie, o incluso determinar si un polígono es cóncavo o convexo según la dirección del vector resultante. Esto es especialmente útil en triangulaciones y otros problemas de geometría.



(2.1.g)

Cabe mencionar que el Producto Cruz puede ser expresado de distintas maneras, ya sea desde una cantidad vectorial, como vimos anteriormente en la Figura 2.1.a, o a través de una transformación lineal. Por ejemplo, podríamos declarar tres valores escalares para obtener las igualdades de la Figura 2.1.b.



```


$$\mathbf{P} \times \mathbf{Q} = (P_y Q_z - P_z Q_y, P_z Q_x - P_x Q_z, P_x Q_y - P_y Q_x)$$


Vector3 CrossProduct(Vector3 p,
Vector3 q)
{
    float x = p.y * q.z - p.z * q.y;
    float y = p.z * q.x - p.x * q.z;
    float z = p.x * q.y - p.y * q.x;

    return new Vector3(x, y, z);
}

```

(2.1.h)

Si prestamos atención a la Figura 2.1.h, notaremos que dentro del método « **CrossProduct** » se han declarado tres variables escalares « **x** », « **y** », « **z** » donde cada una posee un extracto de la ecuación. ¿Cómo podríamos expresar la misma operación desde una transformación lineal? Vamos a prestar atención a la siguiente fórmula,

$$\mathbf{Q} \times \mathbf{P} = \begin{bmatrix} 0 & -Q_z & Q_y \\ Q_z & 0 & -Q_x \\ -Q_y & Q_x & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

(2.1.i)

La definición presentada en la Figura anterior puede parecer compleja, pero su implementación en lenguaje C# es en realidad bastante sencilla. Solo necesitamos recordar que el Producto Cruz se expresa mediante una matriz tridimensional de « **Q** » operando sobre « **P** ».

```

Vector3 CrossProduct(Vector3 p,
Vector3 q)
{
    Matrix4x4 m = new Matrix4x4();

    m[0, 0] = 0;
    m[0, 1] = q.z;
    m[0, 2] = -q.y;

    m[1, 0] = -q.z;
    m[1, 1] = 0;
    m[1, 2] = q.x;

    m[2, 0] = q.y;
    m[2, 1] = -q.x;
    m[2, 2] = 0;

    return m * p;
}

```

(2.1.j)

Es importante destacar que en el método « **CrossProduct** » presentado en la Figura anterior, se ha definido una nueva matriz de cuatro por cuatro dimensiones llamada « **m** ». Esta matriz almacena cada componente de « **Q** » en el mismo orden que aparece en la igualdad de la Figura 2.1.i.

Un factor por considerar es la dirección del Producto Cruz, la cual será determinada en relación con la posición de sus vectores. Por ejemplo: la transformación lineal presentada en la Figura 2.1.i, retorna el siguiente vector,

$$Q \times P = \begin{bmatrix} 0 & -Q_z & Q_y \\ Q_z & 0 & -Q_x \\ -Q_y & Q_x & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

Cross Product			
P	X 0	Y 0	Z 1
Q	X 1	Y 0	Z 0
C	X 0	Y 1	Z 0

(2.1.k)

Sin embargo, si volteamos los valores de la matriz, el Producto Cruz es negado, apuntando en la dirección contraria.

$$P \times Q = \begin{bmatrix} 0 & -P_z & P_y \\ P_z & 0 & -P_x \\ -P_y & P_x & 0 \end{bmatrix} \begin{bmatrix} Q_x \\ Q_y \\ Q_z \end{bmatrix}$$

Cross Product			
P	X 0	Y 0	Z 1
Q	X 1	Y 0	Z 0
C	X 0	Y -1	Z 0

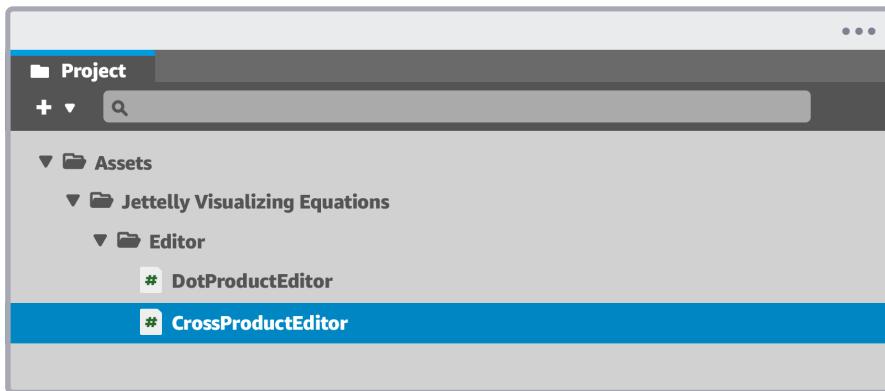
(2.1.l)

## 2.2. Desarrollando una herramienta en Unity.

Siguiendo el propósito de este libro, crearemos una herramienta en Unity que nos ayudará a comprender mejor la naturaleza del Producto Cruz. Para lograrlo, repetiremos parte del proceso detallado en el capítulo anterior, donde utilizamos la dependencia « **UnityEditor** » y extendimos nuestro script desde « **EditorWindow** ». ¿Por qué lo hacemos? Principalmente,

- Porque será una herramienta visual.
- Porque únicamente necesitaremos una instancia de esta.

Para comenzar, crearemos un nuevo script en nuestro proyecto y lo nombraremos « **CrossProductEditor** ». Es importante recordar que para que funcione correctamente, debemos almacenar nuestro controlador dentro de una carpeta de tipo Editor. Para mayor practicidad, se recomienda utilizar la misma carpeta creada en el capítulo anterior.

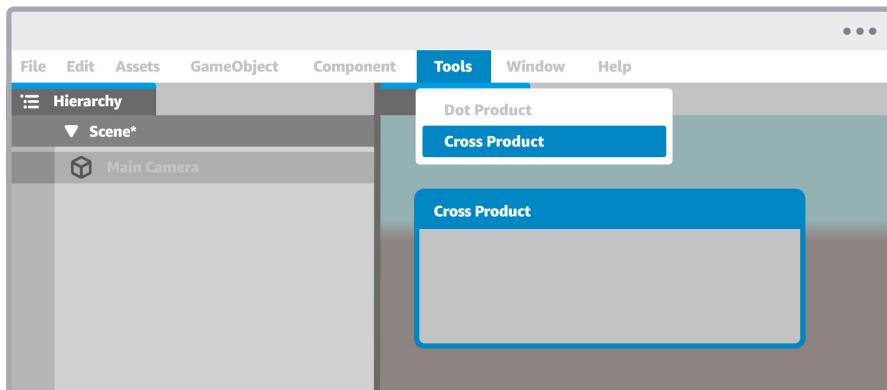


(2.2.a)

En nuestro script, la primera acción que realizaremos será agregar un método estático el cual utilizaremos para desplegar una ventana emergente en Unity.

```
1  using UnityEngine;
2  using UnityEditor;
3
4  public class CrossProductEditor : EditorWindow
5  {
6      [MenuItem("Tools/Cross Product")]
7      public static void ShowWindow()
8      {
9          GetWindow(typeof(CrossProductEditor), true,
10             "Cross Product");
11      }
12 }
```

Como se puede observar en el ejemplo anterior, hemos vuelto a utilizar la función « **GetWindow** », quien retorna una instancia de una ventana. En este caso, la hemos nombrado "Cross Product" a modo de ejemplo. Al regresar a Unity, se puede notar que se ha agregado una nueva ventana en el menú "Tools". Sin embargo, por el momento, esta ventana no realiza acción alguna.



(2.2.b)

Como ya se sabe, el Producto Cruz devuelve un nuevo vector tridimensional basado en la posición de dos vectores de entrada también tridimensionales. Por lo tanto, procederemos a declarar las variables que necesitaremos para el ejercicio de la siguiente manera:

```
4  public class CrossProductEditor : EditorWindow
5  {
6      public Vector3 m_p;
7      public Vector3 m_q;
8      public Vector3 m_pxq;
9
10     private SerializedObject obj;
11     private SerializedProperty propP;
12     private SerializedProperty propQ;
13     private SerializedProperty propPXQ;
14
15     [MenuItem("Tools/Cross Product")]
16 >     public static void ShowWindow() ...
20 }
21
```

Podemos observar en las líneas de código 6, 7, y 8 que se han definido tres vectores tridimensionales: « `m_p` », « `m_q` » y « `m_pxq` ». Además, se han creado tres propiedades de tipo « `SerializedProperty` », una para cada vector. Es importante destacar que es necesario inicializar los valores de los vectores y propiedades, así como también actualizar la ventana Scene en función de los nuevos valores de cada vector. Para llevar a cabo estas tareas, agregaremos los siguientes métodos a nuestro script:

- « `OnEnable` ».
- « `OnDisable` ».
- « `OnGUI` ».

Como pudimos ver en el capítulo anterior, tales funciones vienen incluidas dentro de la clase « **MonoBehaviour** » por consiguiente, dependiendo del editor de código que estemos utilizando, serán reconocidas fácilmente por el "IntelliSense".

```
15 [MenuItem("Tools/Cross Product")]
16 > public static void ShowWindow() ...
20
21 private void OnEnable()
22 {
23
24 }
25
26 private void OnDisable()
27 {
28
29 }
30
31 private void OnGUI()
32 {
33
34 }
35
```

Es importante destacar que utilizaremos al menos dos métodos que ya han sido empleados anteriormente, cuando desarrollamos la ventana "Dot Product". Estos métodos corresponden a:

- « **SceneGUI** », el cual utilizamos para actualizar/visualizar los cambios de herramienta en la ventana Scene.
- Y el método « **DrawBlockGUI** », quien está a cargo de dibujar los valores en la GUI.

Este planteamiento nos lleva a enfrentar un problema: ¡Código repetido! Dado que tanto el script « **CrossProductEditor** » como « **DotProductEditor** » van a utilizar los mismos métodos, sería ideal implementarlos en clases separadas para acceder a ellos de manera más eficiente. Para lograr esto, iremos a nuestro proyecto en Unity y crearemos dos nuevos scripts. El primero se llamará « **CommonEditor** » y lo ubicaremos dentro de nuestra carpeta Editor. Será fundamental asegurarnos de incluir « **UnityEditor** » en él y extender desde la clase « **EditorWindow** ». ¿Por qué? Porque nuestra herramienta actual va a extender de este último y necesitaremos acceso a las funciones integradas para el desarrollo de la misma.

```
1  using UnityEngine;
2  using UnityEditor;
3
4  public class CommonEditor : EditorWindow
5  {
6      public virtual void DrawBlockGUI(string lab,
7          SerializedProperty prop)
8      {
9          EditorGUILayout.BeginHorizontal("box");
10         EditorGUILayout.LabelField(lab, GUILayout.Width(50));
11         EditorGUILayout.PropertyField(prop, GUIContent.none);
12         EditorGUILayout.EndHorizontal();
13     }
14 }
```

Si prestamos atención a la línea de código 6, notaremos que se ha declarado el método « **DrawBlockGUI** », el cual es utilizado en el desarrollo de la herramienta Dot Product. Este método se ha marcado como « **virtual** », lo cual significa que podremos "sobrescribirlo" si es necesario.

Nuestro segundo script será una interfaz llamada « **IUpdateSceneGUI** », la cual se encargará de implementar el método « **SceneGUI** ». ¿Por qué una interfaz? Debido a su naturaleza abstracta, es decir, este método no requiere de un algoritmo específico en su ámbito. Sin embargo, será necesario en varias herramientas que desarrollemos del tipo « **EditorWindow** ».

```
1  using UnityEditor;
2
3  public interface IUpdateSceneGUI
4  {
5      void SceneGUI(SceneView view);
6  }
7
```

A continuación, simplemente podemos extender nuestra herramienta actual desde « **CommonEditor** » e incluir la interfaz « **IUpdateSceneGUI** » como se muestra en el siguiente ejemplo:

```
4  public class CrossProductEditor : CommonEditor, IUpdateSceneGUI
5  {
6      public Vector3 m_p;
7      public Vector3 m_q;
8      public Vector3 m_pxq;
9
10     private SerializedObject obj;
11     private SerializedProperty propP;
12     private SerializedProperty propQ;
13     private SerializedProperty propPXQ;
14
15     [MenuItem("Tools/ Cross Product")]
16 >    public static void ShowWindow() ...
17
18     private void OnEnable()
19     {
20         SceneView.duringSceneGui += SceneGUI;
21     }
22
23
24     private void OnDisable()
25     {
26         SceneView.duringSceneGui -= SceneGUI;
27     }
28
29
30     private void OnGUI() ...
31
32
33
34     public void SceneGUI(SceneView view)
35     {
36         throw new System.NotImplementedException();
37     }
38
39
40 }
41
```

Si prestamos atención a la línea de código número 38, notaremos que, por defecto, C# ha incluido la clase « **NotImplementedException** » perteneciente a « **System** », para comunicar que el método « **SceneGUI** » no ha sido desarrollado. Para evitar conflictos en la creación de nuestra herramienta, será necesario eliminar o comentar tal línea de código. De lo contrario, Unity podría arrojar una excepción de compilación.

Luego procederemos a inicializar las propiedades serializadas en el método « **OnEnable** ». Además, declararemos un nuevo método llamado « **SetDefaultValues** », el cual utilizaremos para configurar los valores predeterminados de los vectores « **m\_p** » y « **m\_q** ».

```
21  private void SetDefaultValues()
22  {
23      m_p = new Vector3(0.0f, 1.0f, 0.0f);
24      m_q = new Vector3(1.0f, 0.0f, 0.0f);
25  }
26
27  private void OnEnable()
28  {
29      if (m_p == Vector3.zero && m_q == Vector3.zero)
30      {
31          SetDefaultValues();
32      }
33
34      obj = new SerializedObject(this);
35      propP = obj.FindProperty("m_p");
36      propQ = obj.FindProperty("m_q");
37      propPXQ = obj.FindProperty("m_pxq");
38
39      SceneView.duringSceneGui += SceneGUI;
40  }
41
```

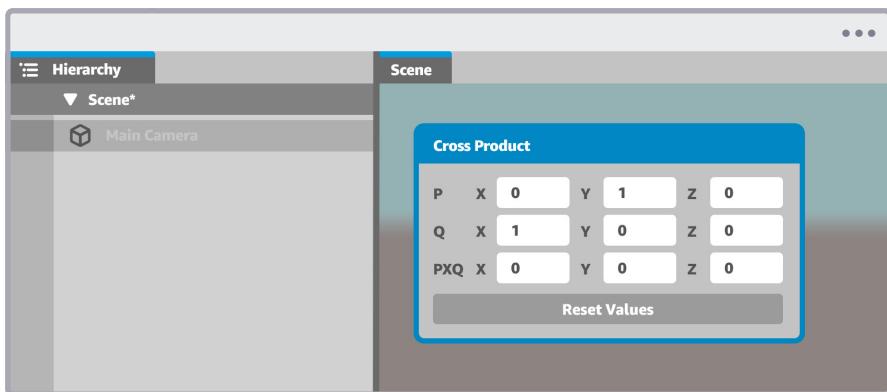
A diferencia de la herramienta desarrollada en el capítulo anterior, en esta oportunidad, agregaremos un botón en la ventana Cross Product, el cual utilizaremos para configurar los vectores y sus valores a su estado por defecto. Este proceso se llevará a cabo con el único propósito de mejorar nuestra comprensión sobre la operación Producto Cruz.

```

47  private void OnGUI()
48  {
49      obj.Update();
50
51      DrawBlockGUI("P", propP);
52      DrawBlockGUI("Q", propQ);
53      DrawBlockGUI("PXQ", propPXQ);
54
55      if (obj.ApplyModifiedProperties())
56      {
57          SceneView.RepaintAll();
58      }
59
60      if (GUILayout.Button("Reset Values"))
61      {
62          SetDefaultValues();
63      }
64  }
65

```

Del ejemplo anterior, cabe recordar que el método « **DrawBlockGUI** » (línea de código 51 a 53) está actualmente implementado en el script « **CommonEditor** ». Por lo tanto, podemos hacer uso de él a través de la herencia. Como se puede apreciar en las líneas de código 60 a 63, se ha agregado una condición que llama a la función « **GUILayout.Button** ». Esta última retorna verdadero o falso dependiendo si el botón ha sido presionado desde la ventana "Cross Product".



(2.2.c)

Hasta este punto, hemos configurado la ventana Cross Product y su funcionalidad principal. A continuación, procederemos a dibujar la herramienta en la ventana Scene. Sin embargo, antes de continuar con el proceso, es importante declarar una variable global de tipo « **GUIStyle** », la cual utilizaremos para dar estilo a los textos en la herramienta.

```
15  private GUIStyle guiStyle = new GUIStyle();
16
17  [MenuItem("Tools/ Cross Product")]
18 > public static void ShowWindow() ...
22
23 > private void SetDefaultValues() ...
28
29  private void OnEnable()
30  {
31      if (m_p == Vector3.zero && m_q == Vector3.zero)
32      {
33          SetDefaultValues();
34      }
35
36      obj = new SerializedObject(this);
37      propP = obj.FindProperty("m_p");
38      propQ = obj.FindProperty("m_q");
39      propPXQ = obj.FindProperty("m_pxq");
40
41      guiStyle.fontSize = 25;
42      guiStyle.fontStyle = FontStyle.Bold;
43      guiStyle.normal.textColor = Color.white;
44
45      SceneView.duringSceneGui += SceneGUI;
46  }
47
```

En la línea de código 15, se ha declarado una nueva variable llamada « **guiStyle** ». Al igual que en el capítulo anterior, su tamaño, estilo y color han sido inicializados en el método « **OnEnable** », líneas 41 a 43.

Antes de continuar con la implementación del método « **SceneGUI** », agregaremos tres nuevos métodos: « **DrawLineGUI** », « **RepaintOnGUI** » y « **CrossProduct** ». Este último corresponde al método definido en la sección anterior, Figura 2.1.h.

```
77  private void DrawLineGUI(Vector3 pos, string tex, Color col)
78  {
79      Handles.color = col;
80      Handles.Label(pos, tex, guiStyle);
81      Handles.DrawAAPolyLine(3f, pos, Vector3.zero);
82  }
83
84  private void RepaintOnGUI()
85  {
86      Repaint();
87  }
88
89  Vector3 CrossProduct(Vector3 p, Vector3 q)
90  {
91      float x = p.y * q.z - p.z * q.y;
92      float y = p.z * q.x - p.x * q.z;
93      float z = p.x * q.y - p.y * q.x;
94
95      return new Vector3(x, y, z);
96  }
97
```

Con el fin de facilitar la identificación de los vectores en la ventana Scene, resulta conveniente agregar un texto que los identifique. Si analizamos la estructura interna del método « `DrawLineGUI` », podemos observar que se enfoca principalmente en dos acciones:

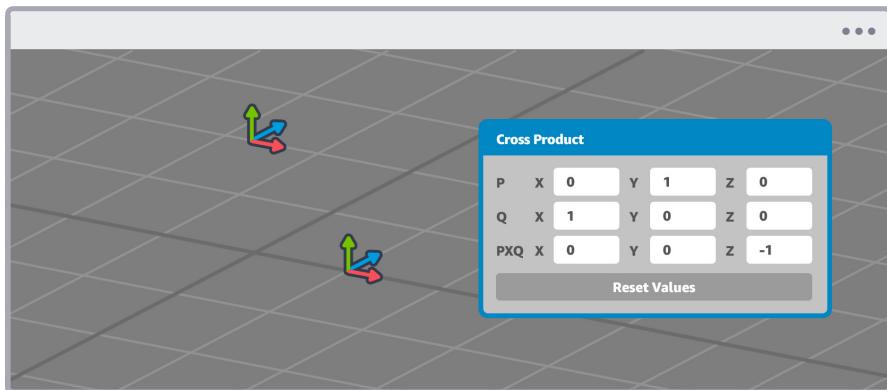
- ① Dibuja un texto utilizando la función « `Handles.Label` ».
- ② Dibuja una línea que va desde la posición actual del vector hasta el punto "cero" del mundo, utilizando la función « `Handles.DrawAAPolyLine` ».

Por otro lado, el método « **RepaintOnGUI** » únicamente tiene la función « **Repaint** » de « **EditorWindow** » en su campo, ya que lo utilizaremos en conjunto con la función « **UnityEditor.Undo** », la cual nos permite registrar operaciones de tipo "deshacer" (CTRL + Z) en nuestro código.

Continuaremos con el método « **SceneGUI** », declarando dos nuevos vectores tridimensionales, « **p** » y « **q** », los cuales utilizaremos para devolver una nueva posición para los vectores « **m\_p** » y « **m\_q** » posteriormente.

```
72  public void SceneGUI(SceneView view)
73  {
74      Vector3 p = Handles.PositionHandle(m_p, Quaternion.identity);
75      Vector3 q = Handles.PositionHandle(m_q, Quaternion.identity);
76  }
77
78 > private void DrawLineGUI(Vector3 pos, string tex, Color col) ...
84
```

Si regresamos a Unity en este momento observaremos que los vectores han aparecido en la ventana Scene. Esto se debe principalmente a la llamada de la función « **Handles.PositionHandle** » la cual devuelve una nueva posición según la interacción del usuario con el Handler. Sin embargo, debemos tomar en cuenta que hasta este punto solo podremos modificar la posición de los « **Handles** » únicamente a través de la ventana « **CrossProduct** », ¿Por qué razón? Porque aún no hemos asignado los valores de « **p** » y « **q** » a sus semejantes globales.



(2.2.d)

A continuación, llevaremos a cabo las siguientes acciones en el método « **SceneGUI** »:

- Declararemos e inicializaremos un nuevo vector de tres dimensiones utilizando el método « **CrossProduct** » y lo representaremos en la ventana Scene mediante un disco sólido de color azul.
- Verificaremos si ha habido algún cambio en la posición de los vectores mediante un condicional « **if** » que evalúe la interacción del usuario con los « **Handles** » previamente implementados.
- En caso de haber cambios, actualizaremos los valores de los vectores « **m\_p** », « **m\_q** » y « **m\_pxq** » con los nuevos valores correspondientes.

```
72  public void SceneGUI(SceneView view)
73  {
74      Vector3 p = Handles.PositionHandle(m_p, Quaternion.identity);
75      Vector3 q = Handles.PositionHandle(m_q, Quaternion.identity);
76
77      Handles.color = Color.blue;
78      Vector3 pxq = CrossProduct(p, q);
79      Handles.DrawSolidDisc(pxq, Vector3.forward, 0.05f);
80
81      if (m_p != p || m_q != q)
82      {
83          Undo.RecordObject(this, "Tool Move");
84
85          m_p = p;
86          m_q = q;
87          m_pxq = pxq;
88
89          RepaintOnGUI();
90      }
91  }
```

En el ejemplo anterior, es importante destacar la línea 78, en la que el vector « `pxq` » se inicializa con el resultado del método « `CrossProduct` », que recibe como argumentos a « `p` » y « `q` ». Además, gracias a la función « `Undo`.  
`RecordObject` » (línea 83), se pueden guardar los cambios realizados en la herramienta. Si se mueven los Handles y se presiona el comando CTRL + Z, la herramienta volverá al último estado que se encontraba en la ventana Scene.

Sin embargo, al guardar los cambios y volver a Unity, se puede notar dos problemas:

- ① El Producto Cruz « pxq » aparece en la ventana Scene, pero es difícil de entender ya que no hay una línea gráfica que lo une a los vectores « p » y « q ».
- ② Si se presiona CTRL + Z después de cambiar la posición de alguno de los Handles, la ventana Cross Product no se actualiza correctamente.

Para solucionar el primer problema, incluiremos al método « **DrawLineGUI** » dentro del campo de la siguiente manera:

```

72     public void SceneGUI(SceneView view)
73     {
74         Vector3 p = Handles.PositionHandle(m_p, Quaternion.identity);
75         Vector3 q = Handles.PositionHandle(m_q, Quaternion.identity);
76
77         Handles.color = Color.blue;
78         Vector3 pxq = CrossProduct(p, q);
79         Handles.DrawSolidDisc(pxq, Vector3.forward, 0.05f);
80
81         if (m_p != p || m_q != q)
82         {
83             Undo.RecordObject(this, "Tool Move");
84
85             m_p = p;
86             m_q = q;
87             m_pxq = pxq;
88
89             RepaintOnGUI();
90         }
91
92         DrawLineGUI(p, "P", Color.green);
93         DrawLineGUI(q, "Q", Color.red);
94         DrawLineGUI(pxq, "PXQ", Color.blue);
95     }
96

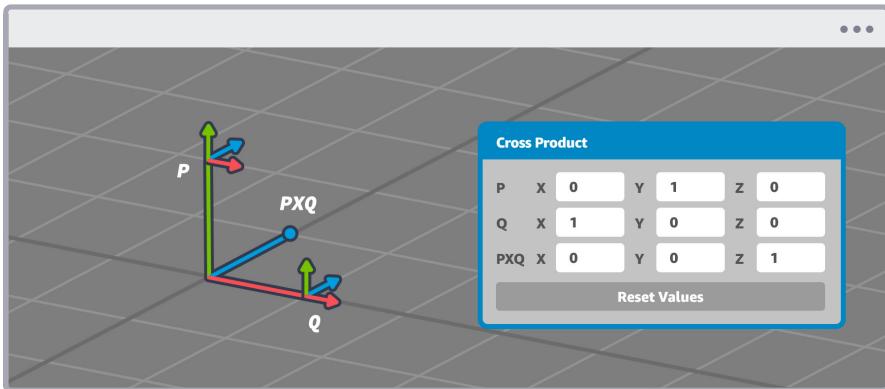
```

El código ha sido actualizado en las líneas 92 a 94. Básicamente hemos agregado un texto y una línea a cada vector, lo que se refleja en la ventana Scene de Unity. Sin embargo, aún enfrentamos un problema cuando deshacemos los cambios presionando CTRL + Z. Afortunadamente, podemos solucionar esto al llamar al método « **RepaintOnGUI** » cada vez que se active un evento del tipo "Undo/Redo".

```
29     private void OnEnable()
30     {
31         if (_p == Vector3.zero && _q == Vector3.zero)
32         {
33             SetDefaultValues();
34         }
35
36         obj = new SerializedObject(this);
37         propP = obj.FindProperty("m_p");
38         propQ = obj.FindProperty("m_q");
39         propPQ = obj.FindProperty("m_pxq");
40
41         guiStyle.fontSize = 25;
42         guiStyle.fontStyle = FontStyle.Bold;
43         guiStyle.normal.textColor = Color.white;
44
45         SceneView.duringSceneGui += SceneGUI;
46         Undo.undoRedoPerformed += RepaintOnGUI;
47     }
48
49     private void OnDisable()
50     {
51         SceneView.duringSceneGui -= SceneGUI;
52         Undo.undoRedoPerformed -= RepaintOnGUI;
53     }
54
```

Como podemos ver, se ha utilizado la función « **Undo.undoRedoPerformed** » en las líneas de código 46 y 52, la cual se activa después de deshacer un cambio

en la ventana Scene. Si todo se ha realizado correctamente, podremos apreciar el comportamiento del Producto Cruz entre los vectores «  $\mathbf{p}$  » y «  $\mathbf{q}$  » en Unity.



(2.2.e)

Como se ha mencionado previamente en la Figura 2.1.j de la sección anterior, podemos implementar el Producto Cruz en nuestro código a través de una transformación lineal. Para entender el proceso, haremos lo siguiente:

- Comentaremos el método « **CrossProduct** » actualmente implementado en el script.
- Definiremos un nuevo método para el mismo, basándonos en la Figura 2.1.j.

```
111  /*
112  Vector3 CrossProduct(Vector3 p, Vector3 q)
113  {
114      float x = p.y * q.z - p.z * q.y;
115      float y = p.z * q.x - p.x * q.z;
116      float z = p.x * q.y - p.y * q.x;
117
118      return new Vector3(x, y, z);
119  }
120 */
121
122  Vector3 CrossProduct(Vector3 p, Vector3 q)
123  {
124      Matrix4x4 m = new Matrix4x4();
125
126      m[0, 0] = 0;
127      m[0, 1] = q.z;
128      m[0, 2] = -q.y;
129
130      m[1, 0] = -q.z;
131      m[1, 1] = 0;
132      m[1, 2] = q.x;
133
134      m[2, 0] = q.y;
135      m[2, 1] = -q.x;
136      m[2, 2] = 0;
137
138      return m * p;
139  }
140
```

Si analizamos con atención la estructura interna de la nueva definición del método « **CrossProduct** », podremos observar que se ha creado e inicializado una matriz de cuatro por cuatro dimensiones, en la cual se ha asignado a cada valor su respectiva columna/fila correspondiente. Como resultado, se obtiene el mismo valor que en versiones anteriores del método.

## Resumen.

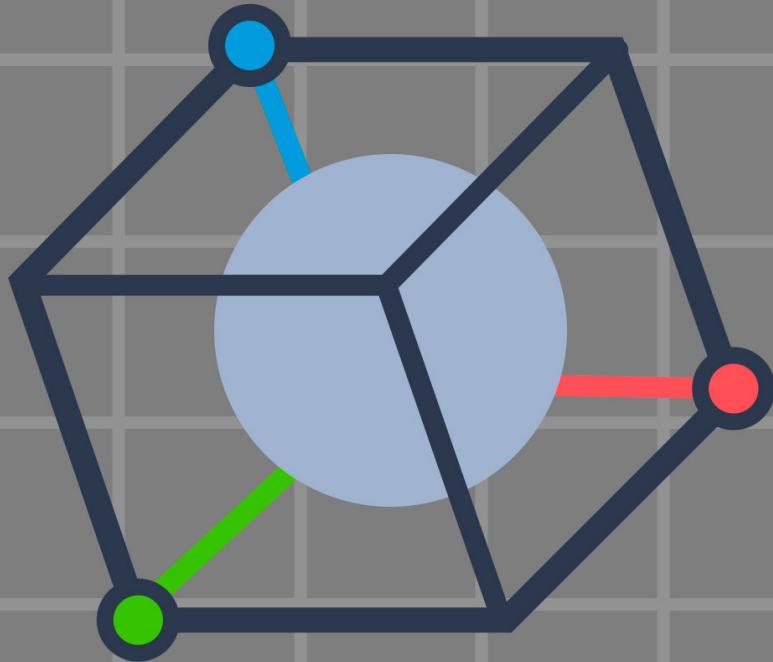
En este capítulo, comenzamos definiendo el Producto Cruz entre dos vectores utilizando la regla de la mano derecha, una técnica tradicional que nos permite comprender su dirección y magnitud en un espacio tridimensional.

También exploramos una perspectiva diferente al introducir el Producto Cruz como una aplicación de una matriz; una transformación lineal que relaciona dos vectores en un espacio vectorial.

A lo largo del capítulo, desglosamos las propiedades geométricas fundamentales del Producto Cruz, incluyendo su dependencia en la orientación de los vectores originales y su relación con el área de un paralelogramo.

Para una comprensión más práctica, presentamos un ejemplo sencillo que ilustra cómo calcular el Producto Cruz en la práctica, mostrando cómo obtener tanto el vector resultante como su magnitud.

Finalmente, concluimos el capítulo explorando la implementación del Producto Cruz en lenguaje C#, específicamente en el contexto de Unity. Para ello, desarrollamos una herramienta dentro del editor, la cual facilitó la visualización tridimensional de la función, brindando una comprensión más amplia de su aplicación en la práctica y su utilidad en la creación de entornos virtuales.



# Capítulo 3. Cuaterniones.

### 3.1. Introducción a la función.

Los cuaterniones son objetos fundamentales que podemos encontrar en cualquier software de desarrollo enfocado a la generación de objetos 3D, ¿por qué razón? Estos nos permiten generar rotaciones de vértices, evitando el fenómeno producido por el bloqueo del cardán (también conocido como efecto gimbal), el cual provoca la pérdida de un grado de libertad, causando un comportamiento inesperado en el sistema de rotación.

Para entender su definición, vamos a prestar atención a la siguiente fórmula:

$$q = w + xi + yj + zk$$

(3.1.a)

Es bastante común sentir un grado de temor cuando intentamos interpretar este tipo de ecuaciones por primera vez. Tal temor se debe precisamente al poco entendimiento que tenemos de las propiedades de estos objetos.

Dado que este tipo de objeto matemático extiende del concepto de los números complejos, podemos deducir que las variables «  $i$  », «  $j$  » y «  $k$  » de la ecuación 3.1.a, están relacionados con los vectores base canónicos del espacio euclídeo tridimensional, la notación para estos "entes" ha sido tomada adrede para obtener una relación directa con sus correspondientes ejes:

$$\begin{aligned} i &= x \text{ axis} \\ j &= y \text{ axis} \\ k &= z \text{ axis} \end{aligned}$$

(3.1.b)

Las variables «  $\mathbf{w}$  », «  $\mathbf{x}$  », «  $\mathbf{y}$  » y «  $\mathbf{z}$  », corresponden a números reales, que representan un punto en el espacio con componentes dados por el triplete «  $\mathbf{x}$  », «  $\mathbf{y}$  », «  $\mathbf{z}$  » y un escalar asociado «  $\mathbf{w}$  ». Por lo tanto, de un lado tenemos a las coordenadas que representan los vectores base del espacio tridimensional, mientras que, por el otro, tenemos las componentes reales que corresponden a las coordenadas de un vértice dentro de ese espacio. Los objetos «  $\mathbf{i}$  », «  $\mathbf{j}$  » y «  $\mathbf{k}$  » son diferentes a los vectores base ya que son números imaginarios que obedecen a la regla de multiplicación dada por,

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

(3.1.c)

¿Por qué son imaginarios? Básicamente, porque tienen la particularidad de que su cuadrado da como resultado -1, es decir, que viven en un mundo matemático distinto de los números reales.

Cabe preguntarnos entonces, ¿Cómo podemos llevar a cabo la rotación de un objeto con múltiples vértices considerando la explicación anterior? Para ello, primero debemos considerar al menos tres factores:

- ① El producto de un Cuaternión.
- ② El conjugado del mismo.
- ③ El ángulo de rotación.

A diferencia de la definición mencionada anteriormente en la Figura 3.1.a, a menudo, un Cuaternión se puede representar en la forma escalar-vector,

$$\mathbf{q} = \mathbf{s} + \mathbf{v}$$

(3.1.d)

Donde, «  $s$  » representa el valor escalar del componente «  $w$  », y «  $v$  » se refiere al vector con componentes «  $x$  », «  $y$  » y «  $z$  », es decir, la parte imaginaria del Cuaternión «  $q$  » en la ecuación 3.1.a,

$$\begin{aligned}s &= w \\ v &= (x, y, z) = xi + yj + zk\end{aligned}$$

(3.1.e)

Tal definición es más compacta y ayuda a simplificar ciertas operaciones, como lo es la multiplicación de dos cuaterniones. Para entender el concepto de mejor manera, realizaremos el siguiente ejercicio: Definiremos dos cuaterniones, «  $q_1$  » y «  $q_2$  » como muestra la siguiente figura,

$$\begin{aligned}q_1 &= s_1 + v_1 \\ q_2 &= s_2 + v_2\end{aligned}$$

(3.1.f)

Posteriormente, aplicaremos la operación de multiplicación entre ambos,

$$q_1 q_2 = s_1 s_2 - v_1 \cdot v_2 + s_1 v_2 + s_2 v_1 + v_1 \times v_2$$

(3.1.g)

La ecuación presentada en la Figura 3.1.g puede parecer compleja a primera vista, sin embargo, hay dos observaciones importantes que debemos tener en cuenta de inmediato. En primer lugar, la multiplicación entre cuaterniones no es conmutativa, es decir, que «  $q_1$  » por «  $q_2$  » no es lo mismo que «  $q_2$  » por «  $q_1$  ». Esta propiedad nos ayuda a evitar cometer errores en el futuro. En segundo lugar, al examinar la estructura del Cuaternión resultante de la multiplicación, podemos notar que su parte escalar y vectorial corresponden a

operaciones que ya hemos realizado previamente con vectores. Esto nos permite una implementación bastante sencilla en C#.

$$\mathbf{q}_1 \mathbf{q}_2 = \mathbf{s}_1 \mathbf{s}_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + \mathbf{s}_1 \mathbf{v}_2 + \mathbf{s}_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2$$

```
public static Q ScalarVector(Q q1, Q q2)
{
    float s1 = q1.w;
    float s2 = q2.w;

    Vector3 v1 = new Vector3 (q1.x, q1.y, q1.z);
    Vector3 v2 = new Vector3 (q2.x, q2.y, q2.z);

    float s = s1 * s2 - Vector3.Dot(v1, v2);
    Vector3 v = s1 * v2 + s2 * v1 + Vector3.Cross(v1, v2);

    return new Q(v.x, v.y, v.z, s);
}
```

(3.1.h)

Como podemos apreciar en el método « **ScalarVector** » de la Figura 3.1.h, se han declarado dos variables de tipo flotante « **s1** » y « **s2** » para almacenar los valores del componente « **w** » para cada Cuaternión. Posteriormente, se han declarado dos vectores tridimensionales para guardar los valores « **x** », « **y** » y « **z** », siguiendo la misma lógica presentada en la Figura 3.1.g.

Finalmente, se ha declarado una variable escalar denominada « **s** » y un vector « **v** » siguiendo la definición presentada en la Figura 3.1.d. Estos reciben los resultados de la operación que se lleva a cabo en la multiplicación de los cuaterniones.

Cabe destacar que el tipo de valor « **Q** » presentado en la Figura 3.1.h, permite exemplificar una estructura de tipo « **struct** », la cual encapsula cada componente « **x** », « **y** », « **z** » y « **w** » de un Cuaternión.

Considerando los números imaginarios en la parte vectorial de un Cuaternión, podemos realizar su conjugación, lo cual implica cambiar el signo de todos los términos imaginarios. A modo de ejemplo, conjugaremos la ecuación planteada en la Figura 3.1.d de la siguiente forma,

$$\bar{q} = s - v$$

(3.1.i)

Que es lo mismo decir,

$$\bar{q} = w - xi - yj - zk$$

(3.1.j)

Desde una perspectiva geométrica, la conjugación se puede entender como una operación que preserva la parte escalar (o real) de un Cuaternión, mientras genera una reflexión en su parte vectorial. Es como si colocásemos un espejo perpendicular al vector y observáramos su imagen reflejada. Esta propiedad convierte a la conjugación en una operación que nos permite obtener de manera natural la reflexión de un vector en particular. La conjugación de un Cuaternión también nos proporciona una forma sencilla de obtener su valor absoluto, el cual podemos expresar de la siguiente manera,

$$|q| = \sqrt{q * \bar{q}}$$

(3.1.k)

Que es lo mismo decir,

$$|q| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

(3.1.k)

Hay algunas razones por la que debemos conjugar un Cuaternión, por ejemplo: Cuando realizamos una interpolación, invertimos un Cuaternión o efectuamos una rotación. Para nuestro caso, será necesario realizar la conjugación cuando implementemos la rotación de un Cuaternión que represente un punto en el espacio.

Su implementación en C# luce de la siguiente manera:



```
public static Q Conjugate(Q q)
{
    return new Q (-q.x, -q.y, -q.z, q.w);
}
```

(3.1.m)

Si prestamos atención a la Figura 3.1.m, notaremos que el componente «  $w$  » se mantiene positivo, mientras que cada componente vectorial pasa a ser negativo. Dada la naturaleza de la explicación anterior, a continuación, vamos a representar cuaterniones como puntos en un espacio tridimensional, y para ello volveremos a prestar atención a la Figura 3.1.a. Sin embargo, en esta ocasión vamos a ignorar completamente la parte real de la ecuación.

$$p = xi + yj + zk$$

(3.1.n)

Como podemos apreciar en la Figura 3.1.n, podemos expresar un punto en un espacio tridimensional de manera más concisa y directa, utilizando únicamente las componentes imaginarias. Ahora, cuando hablamos sobre rotaciones en el espacio, será fundamental definir un ángulo y un eje. Cabe destacar que el ángulo de rotación se mide en radianes, mientras que el eje de rotación se define mediante un vector unitario en el espacio.

Vamos a prestar atención a la siguiente fórmula para entender el concepto.

$$q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) * (u_x * i + u_y * j + u_z * k)$$

(3.1.o)

De la figura anterior,

- El símbolo «  $\theta$  » se refiere al ángulo de rotación.
- Las variables «  $u_x$  », «  $u_y$  » y «  $u_z$  » son los componentes de un vector unitario que definen el eje de rotación.
- Finalmente, «  $i$  », «  $j$  » y «  $k$  » son los números imaginarios estándar.

Su implementación en C# puede ser representada de la siguiente manera,

$$q = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) * (\mathbf{u}_x * \mathbf{i} + \mathbf{u}_y * \mathbf{j} + \mathbf{u}_z * \mathbf{k})$$

```

Q Create(Float angle, Vector3 axis)
{
    float r = Mathf.Sin(angle / 2f);
    float s = Mathf.Cos(angle / 2f);
    Vector3 v = Vector3.Normalize(axis) * r;

    return new Q(v.x, v.y, v.z, s);
}

```

(3.1.p)

Una vez que tenemos el Cuaternión de rotación, podemos aplicarlo a un punto en el espacio. La regla de transformación es la siguiente:

$$\mathbf{p}' = (\mathbf{q} * \mathbf{p}) * \bar{\mathbf{q}}$$

(3.1.q)

De la Figura 3.1.q, podemos deducir con facilidad que la variable «  $\mathbf{q}$  » corresponde al Cuaternión de rotación, mientras que «  $\bar{\mathbf{q}}$  » se refiere a su conjugado. Por otra parte, «  $\mathbf{p}$  » corresponde al punto que deseamos rotar. ¿Cómo podemos visualizar esto? Para ello, realizaremos el siguiente ejercicio: Iniciaremos posicionando un punto o vértice «  $\mathbf{p}$  » en el espacio.

$$\mathbf{p} = (1_x, 0_y, 0_z)$$

(3.1.r)

Luego, definimos un ángulo y un eje de rotación. Para este ejercicio, utilizaremos  $45^\circ$  sobre el eje « ***Y*** ».

$$\theta = 45^\circ$$

$$j = (0_i, 1_j, 0_k)$$

(3.1.s)

A continuación, definimos el Cuaternión de rotación siguiendo la ecuación de la Figura 3.1.o de la siguiente manera,

$$q = \cos\left(\frac{45}{2}\right) + \sin\left(\frac{45}{2}\right) * j$$

(3.1.t)

Que es lo mismo decir,

$$q = 0.923 + 0.382j$$

(3.1.u)

Por tanto,

$$q = (0.923_w, 0_x, 0.382_y, 0_z)$$

(3.1.v)

Cabe destacar que la primera operación de la ecuación anterior «  $\cos\left(\frac{45}{2}\right)$  » se refiere a la parte real del Cuaternión, es decir, al componente « ***S*** » de la forma «  $q = s + v$  », mientras que el resto define los valores vectoriales « ***v*** ». Más adelante en este libro podremos ver en detalle su implementación en C#.

Únicamente faltaría rotar el punto inicial. Para ello, debemos aplicar la ecuación de la Figura 3.1.q, de la siguiente manera:

$$\mathbf{p}' = [(\mathbf{0.923}_w + \mathbf{0.382}_j) * (\mathbf{1}_x, \mathbf{0}_y, \mathbf{0}_z)] * (\mathbf{0.923}_w - \mathbf{0.382}_j)$$

(3.1.w)

Que es lo mismo decir,

$$\mathbf{p}' = \mathbf{0.707}_i - \mathbf{0.707}_k$$

(3.1.x)

Por lo tanto,

$$\mathbf{p}' = (\mathbf{0.707}_x, \mathbf{0}_y, -\mathbf{0.707}_z)$$

(3.1.y)

De la Figura anterior, «  $\mathbf{p}'$  » corresponde a la nueva posición de un punto rotado 45° grados sobre el eje «  $\mathbf{Y}$  » en un espacio tridimensional.

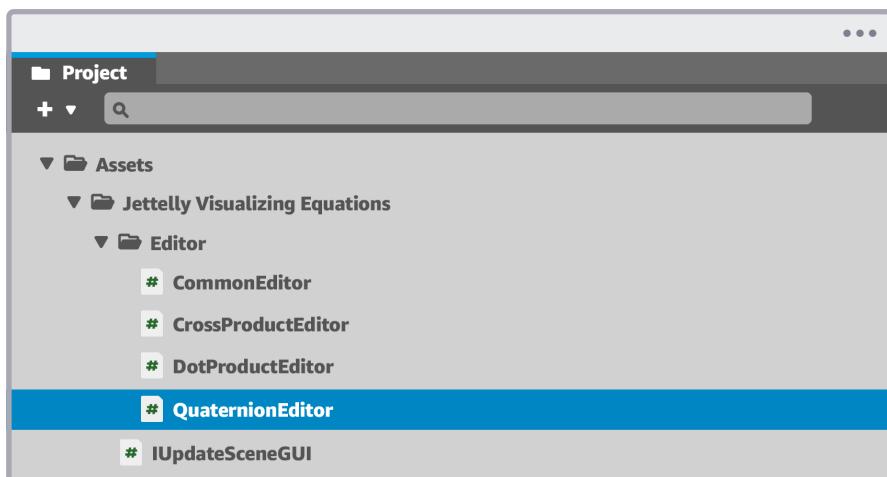
### 3.2. Desarrollando una herramienta en Unity.

Hasta este punto, hemos revisado algunas de las ecuaciones involucradas en el proceso de rotación de un punto en el espacio mediante cuaterniones. A continuación, procederemos a desarrollar una herramienta, centrándonos en las distintas funciones u operaciones asociadas, con el fin de comprender en mayor detalle el concepto mencionado.

El proceso descrito en este capítulo no difiere del explicado en capítulos anteriores, por lo tanto, una vez más, haremos uso de la dependencia « **UnityEditor** », y extenderemos nuestro script desde « **CommonEditor** », ya que, éste último

posee la clase « **EditorWindow** » fundamental para la creación de este tipo de herramientas.

Para iniciar este proceso, crearemos un nuevo script en nuestro proyecto llamado « **QuaternionEditor** ». Es importante recordar que, para que funcione correctamente, debemos ubicar este script en nuestra carpeta de tipo Editor, que creamos anteriormente en nuestro proyecto en Unity.



(3.2.a)

En esta oportunidad, no solo será necesario implementar las funciones y métodos que permiten la generación de la herramienta, sino que también tendremos que crear un objeto de tipo « **struct** » para encapsular las distintas propiedades que ofrece un Cuaternión. Para iniciar el desarrollo, definiremos la ventana que mostraremos desde el menú "Tools" en Unity, la cual llamaremos "Quaternion" por practicidad.

```
1  using UnityEngine;
2  using UnityEditor;
3
4  public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
5  {
6      [MenuItem("Tools/Quaternion")]
7      public static void ShowWindow()
8      {
9          GetWindow(typeof(QuaternionEditor), true, "Quaternion");
10     }
11
12     public void SceneGUI(SceneView view)
13     {
14         // throw new System.NotImplementedException();
15     }
16 }
```

Del código anterior, si observamos la línea número 4, notaremos que se ha implementado la interfaz « **IUpdateSceneGUI** ». Esto se debe principalmente a que volveremos a utilizar el método « **SceneGUI** » para proyectar una lista de puntos que, en su conjunto, formarán un cubo tridimensional.

Continuaremos declarando una lista global de vectores en la que almacenaremos cada punto o vértice del cubo. Posteriormente, inicializaremos cada vértice dentro del método « **SceneGUI** ».

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using UnityEditor;
4
5  public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
6  {
7      private List<Vector3> vertices;
8
9      [MenuItem("Tools/Quaternion")]
10 >     public static void ShowWindow() ...
11
12
13
14
15     public void SceneGUI(SceneView view)
16     {
17         vertices = new List<Vector3>
18         {
19             new Vector3(-0.5f, 0.5f, 0.5f),
20             new Vector3( 0.5f, 0.5f, 0.5f),
21             new Vector3( 0.5f, -0.5f, 0.5f),
22             new Vector3(-0.5f, -0.5f, 0.5f),
23             new Vector3(-0.5f, 0.5f, -0.5f),
24             new Vector3( 0.5f, 0.5f, -0.5f),
25             new Vector3( 0.5f, -0.5f, -0.5f),
26             new Vector3(-0.5f, -0.5f, -0.5f)
27         };
28     }
29 }
30
```

Será necesario incluir la dependencia « **System.Collections.Generic** » al trabajar con listas del tipo « **List** ». Como se observa en el código anterior, dicha dependencia se ha incluido en la línea de código número 1. Posteriormente, la lista de vértices es declarada en la línea número 7. Es importante mencionar que, por motivos educativos, se está utilizando una lista en la implementación actual. Sin embargo, para mayor coherencia, considerando que no se agregarán más puntos a los ya definidos, podríamos utilizar fácilmente una lista constante de vectores tridimensionales.

Los ocho vértices que posee el cubo han sido inicializados en las líneas de código 17 a 19. No obstante, será fundamental generar gizmos para cada uno de ellos, a fin de representarlos gráficamente en la ventana Scene. Realizaremos este proceso iterando cada vértice dentro de un bucle « **for** » y utilizaremos la función « **Handles.SphereHandleCap** » para su representación visual.

```

15  public void SceneGUI(SceneView view)
16  {
17      vertices = new List<Vector3>
18      {
19          new Vector3(-0.5f,  0.5f,  0.5f),
20          new Vector3( 0.5f,  0.5f,  0.5f),
21          new Vector3( 0.5f, -0.5f,  0.5f),
22          new Vector3(-0.5f, -0.5f,  0.5f),
23          new Vector3(-0.5f,  0.5f, -0.5f),
24          new Vector3( 0.5f,  0.5f, -0.5f),
25          new Vector3( 0.5f, -0.5f, -0.5f),
26          new Vector3(-0.5f, -0.5f, -0.5f)
27      };
28
29      for (int i = 0; i < vertices.Count; i++)
30      {
31          Handles.SphereHandleCap(0, vertices[i],
32                                  Quaternion.identity, 0.1f, EventType.Repaint);
33      }
34

```

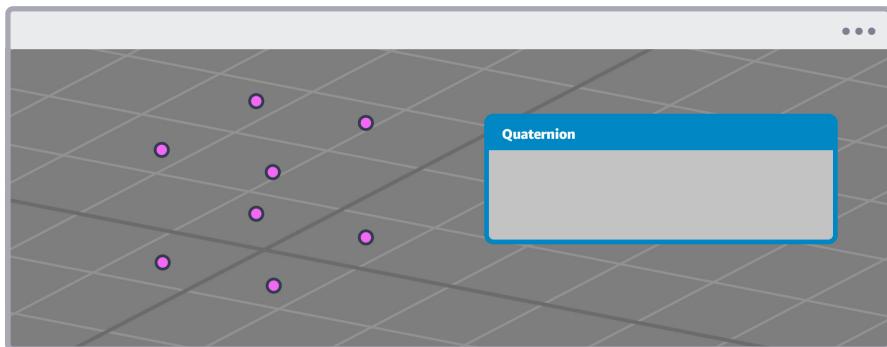
Si prestamos atención a las líneas de código del 29 al 32, podemos observar que se ha creado un bucle dentro del método « **SceneGUI** », el cual itera sobre cada vértice definido en la lista. Es importante destacar que el segundo argumento de la función « **SphereHandleCap** », llamado « **Quaternion.identity** », corresponde a la rotación de los vértices mediante cuaterniones. Sin embargo, en este caso específico, estamos devolviendo la identidad del Cuaternión, lo que significa que no hay rotación. Esto se debe a que más adelante en este libro

desarrollaremos nuestra propia implementación de cuaterniones para rotar los vértices del cubo según un ángulo definido.

Si volvemos a Unity en este momento, no veremos ninguna representación gráfica de los puntos del cubo en la venta Scene. Esto se debe a que no estamos ejecutando este proceso durante la llamada al método « **OnGUI** ». Para solucionarlo, simplemente debemos suscribirnos a la función « **SceneView.duringSceneGui** » para recibir una llamada de retorno. Implementaremos esta llamada tanto en el método « **OnEnable** » como en « **OnDisable** ».

```
 5  public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
 6  {
 7      private List<Vector3> vertices;
 8
 9      [MenuItem("Tools/Quaternion")]
10 >     public static void ShowWindow() ...
11
12      private void OnEnable()
13      {
14          SceneView.duringSceneGui += SceneGUI;
15      }
16
17      private void OnDisable()
18      {
19          SceneView.duringSceneGui -= SceneGUI;
20      }
21
22 >     public void SceneGUI(SceneView view) ...
23
24      }
25
26  }
```

Del código anterior, podemos ver la implementación mencionada anteriormente en las líneas número 17 y 22. Si volvemos a Unity, podremos encontrar nuestra nueva ventana "Quaternion" dentro del menú Tool, la cual proyecta los ocho vértices de un cubo.

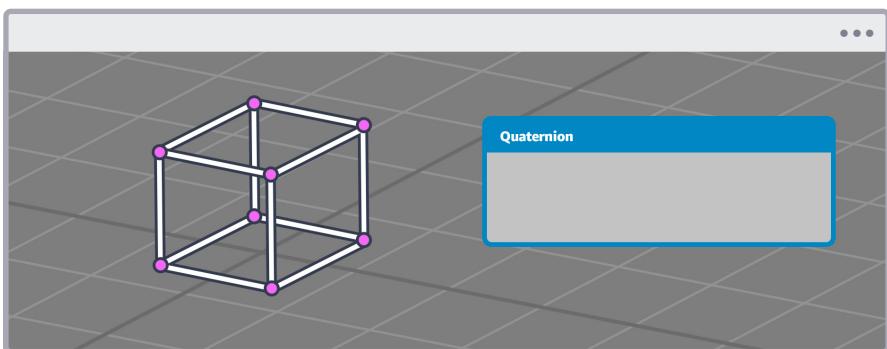


(3.2.b)

Una acción que podemos llevar a cabo en nuestra herramienta consiste en trazar una línea entre cada vértice para mejorar la proyección del cubo en la ventana Scene. Para lograrlo, utilizamos la función « `Handles.DrawAAPolyLine` », la cual nos permite dibujar líneas entre puntos en el espacio tridimensional. Sin embargo, su implementación en este caso presenta un desafío: si deseamos conectar todos los vértices sin repeticiones, es necesario agrupar las diferentes combinaciones de puntos en una lista de valores enteros. Una manera de abordar este proceso es mediante una lista doble, donde el primer grupo contendría las combinaciones de puntos, mientras que el segundo grupo albergaría los puntos espaciales correspondientes.

```
25 public void SceneGUI(SceneView view)
26 {
27     vertices = new List<Vector3>
28     {
29         new Vector3(-0.5f,  0.5f,  0.5f),
30         new Vector3( 0.5f,  0.5f,  0.5f),
31         new Vector3( 0.5f, -0.5f,  0.5f),
32         new Vector3(-0.5f, -0.5f,  0.5f),
33         new Vector3(-0.5f,  0.5f, -0.5f),
34         new Vector3( 0.5f,  0.5f, -0.5f),
35         new Vector3( 0.5f, -0.5f, -0.5f),
36         new Vector3(-0.5f, -0.5f, -0.5f)
37     };
38
39     for (int i = 0; i < vertices.Count; i++)
40     {
41         Handles.SphereHandleCap(0, vertices[i],
42             Quaternion.identity, 0.1f, EventType.Repaint);
43     }
44     int[][] index =
45     {
46         new int[] {0, 1},
47         new int[] {1, 2},
48         new int[] {2, 3},
49         new int[] {3, 0},
50         new int[] {4, 5},
51         new int[] {5, 6},
52         new int[] {6, 7},
53         new int[] {7, 4},
54         new int[] {4, 0},
55         new int[] {5, 1},
56         new int[] {6, 2},
57         new int[] {7, 3},
58     };
59
60     for (int i = 0; i < index.Length; i++)
61     {
62         Handles.DrawAAPolyLine(vertices[index[i][0]],
63             vertices[index[i][1]]);
64     }
}
```

Si dirigimos nuestra atención a la línea de código número 44 en el método « **SceneGUI** », podemos notar que se ha declarado e inicializado una lista constante de valores enteros. Estos valores representan diversos índices que establecen conexiones entre los vértices. Como se mencionó previamente, esto tiene como objetivo generar una representación gráfica mediante líneas que conectan pares de puntos del cubo, mejorando así su visualización. Más adelante, en la línea de código número 60, se ha incluido un bucle « **for** » que itera a través de la lista de índices y utiliza a la función « **DrawAAPolyLine** » para dibujar las líneas correspondientes. Al examinar la ventana Scene una vez más, podremos apreciar las conexiones entre cada vértice del cubo.



(3.2.c)

Hasta este momento, hemos dedicado nuestra atención a la proyección de puntos en el espacio, principalmente con el fin de visualizar la figura que posteriormente rotaremos. Ahora, pondremos en práctica nuestro pensamiento lógico para incorporar todas las propiedades necesarias de un Cuaternión. Para ello, empezaremos declarando dos nuevas variables globales:

- Un valor de tipo flotante que representará el ángulo de rotación.
- Un vector tridimensional que indicará el eje de rotación.

```

5   public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
6   {
7       [Range(-360, 360)] public float m_angle = 0f;
8       public Vector3 m_axis = new Vector3(0, 1, 0);
9
10      private SerializedObject obj;
11      private SerializedProperty propAngle;
12      private SerializedProperty propAxis;
13
14      private List<Vector3> vertices;
15
16      [MenuItem("Tools/Quaternion")]
17      public static void ShowWindow() ...
18

```

Siguiendo las prácticas establecidas en capítulos anteriores, es necesario declarar ciertas propiedades del tipo « **SerializedProperty** » para proyectar las variables en la ventana de nuestra herramienta en desarrollo. Este tipo de dato nos permitirá gestionar comandos de « **Undo** », editar múltiples objetos y controlar la sobreescritura de los Prefabs. la declaración de estas propiedades se puede observar en las líneas de código 11 y 12.

A continuación, procederemos a inicializar las variables en el método « **OnEnable** » para, posteriormente, proyectarlas en la ventana "Quaternion" mediante el uso del método « **OnGUI** ».

```

22  private void OnEnable()
23  {
24      obj = new SerializedObject(this);
25      propAngle = obj.FindProperty("m_angle");
26      propAxis = obj.FindProperty("m_axis");
27
28      SceneView.duringSceneGui += SceneGUI;
29  }

```

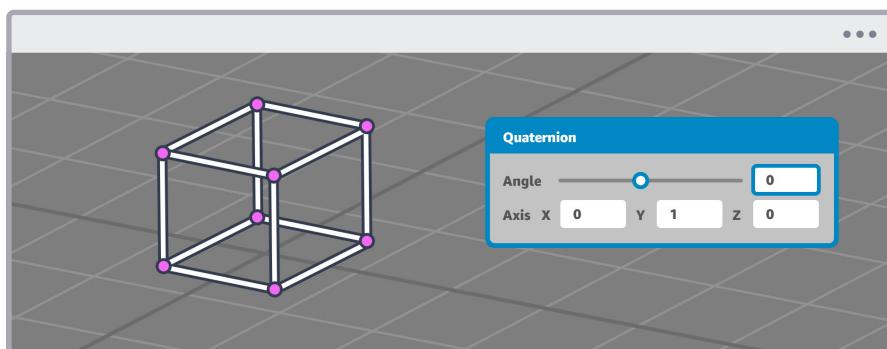
En el ejemplo previo, las propiedades « `propAngle` » y « `propAxis` » se han inicializado con los valores de cada variable en las líneas de código 25 y 26, respectivamente. Ahora, solo falta proyectar estas variables en nuestra herramienta. Dado que nuestro script es una extensión de « `CommonEditor` », utilizaremos nuevamente en el método « `DrawBlockGUI` », el cual nos permite dibujar una etiqueta y una propiedad dentro de un marco horizontal, como ya sabemos.

```

22     private void OnGUI()
23     {
24         obj.Update();
25
26         DrawBlockGUI("Angle", propAngle);
27         DrawBlockGUI("Axis", propAxis);
28
29         if (obj.ApplyModifiedProperties())
30         {
31             SceneView.RepaintAll();
32         }
33     }

```

Si volvemos a Unity podremos observar que las propiedades mencionadas anteriormente se han incluido correctamente en nuestra herramienta.



(3.2.d)

Hasta este punto, podemos considerar que el proceso inicial ha concluido. El cubo ha sido proyectado en la ventana Scene y sus vértices se mantienen constantes. A continuación, procederemos a implementar las ecuaciones necesarias para rotar los vértices, siguiendo la definición de un Cuaternión. Por razones prácticas, dentro del mismo script en el que estamos trabajando, declararemos una estructura llamada « **HQuaternion** » para evitar conflictos con el tipo de datos « **Quaternion** » proporcionado por las dependencias de Unity. Es importante destacar que este proceso también puede llevarse a cabo desde un script dedicado exclusivamente a esta tarea.

```
5 > public class QuaternionEditor ...
91
92 public struct HQuaternion
93 {
94
95 }
96
```

En el ejemplo anterior, si nos fijamos en la línea de código 92, veremos que se ha declarado una estructura « **struct** » fuera del ámbito de la clase « **QuaternionEditor** ». Entonces, surge la pregunta: ¿Por qué usar una estructura? Principalmente, se debe a sus características. Las estructuras se utilizan comúnmente para definir tipos de datos que se almacenan en la pila (Stack). Esto permite manipular instancias de este tipo de manera más rápida y eficiente cuando se trabaja con objetos.

Como vimos en la sección anterior de este capítulo, los cuaterniones constan de cuatro dimensiones, representadas por los componentes « **x** », « **y** », « **z** » y « **w** ». Por lo tanto, es necesario declarar estas variables dentro del ámbito de la estructura y también definir explícitamente un constructor para inicializar sus valores.

```
92  public struct HQuaternion
93  {
94      private float x;
95      private float y;
96      private float z;
97      private float w;
98
99      public HQuaternion(float x, float y, float z, float w)
100     {
101         this.x = x;
102         this.y = y;
103         this.z = z;
104         this.w = w;
105     }
106 }
107
```

Como ya sabemos, para rotar un vértice utilizando cuaterniones, debemos seguir la ecuación mencionada en la Figura 3.1.q de la sección anterior. Por lo tanto, comenzaremos definiendo un Cuaternion de rotación siguiendo la operación detallada en la Figura 3.1.o. Para lograrlo, implementaremos un método estático al que llamaremos « **Create** », el cual tendrá como argumento un valor escalar para el ángulo y un vector tridimensional para los ejes.

```
92  public struct HQuaternion
93  {
94      private float x;
95      private float y;
96      private float z;
97      private float w;
98
99 >     public HQuaternion(float x, float y, float z, float w) ...
106
107     private static HQuaternion Create(float angle, Vector3 axis)
108     {
109         float sin = Mathf.Sin(angle / 2f);
110         float cos = Mathf.Cos(angle / 2f);
111         Vector3 v = Vector3.Normalize(axis) * sin;
112
113         return new HQuaternion(v.x, v.y, v.z, cos);
114     }
115 }
116
```

Si nos fijamos en la línea de código 113, el método « **Create** » devuelve un nuevo Cuaternión de rotación en la forma escalar-vector. Es decir, los componentes « **v.x** », « **v.y** » y « **v.z** » corresponden a la parte vectorial, mientras que « **cos** » representa la parte escalar.

Siguiendo con el proceso, es necesario realizar la conjugación del Cuaternión de rotación, es decir, invertir su componente vectorial. Para lograrlo, declararemos un nuevo método estático llamado « **Conjugate** ». Este método se encargará de llevar a cabo la operación mencionada.

```

107 > private static HQuaternion Create(float angle, Vector3 axis) ...
115
116     private static HQuaternion Conjugate(HQuaternion q)
117     {
118         float s = q.w;
119         Vector3 v = new Vector3(-q.x, -q.y, -q.z);
120
121         return new HQuaternion(v.x, v.y, v.z, s);
122     }

```

Si prestamos atención a la línea de código 121, observaremos que el método mencionado anteriormente devuelve un nuevo Cuaternión « `HQuaternion` », siguiendo la ecuación presentada en la Figura 3.1.i de la sección anterior. Ahora, solo falta realizar las multiplicaciones necesarias. Para lograrlo, aplicaremos la ecuación presentada en la Figura 3.1.q de la siguiente manera,

```

116 > private static HQuaternion Conjugate(HQuaternion q) ...
123
124     private static HQuaternion Multiplication(HQuaternion q1,
125         HQuaternion q2)
126     {
127         float s1 = q1.w;
128         float s2 = q2.w;
129
130         Vector3 v1 = new Vector3(q1.x, q1.y, q1.z);
131         Vector3 v2 = new Vector3(q2.x, q2.y, q2.z);
132
133         float s = s1 * s2 - Vector3.Dot(v1, v2);
134         Vector3 v = s1 * v2 + s2 * v1 + Vector3.Cross(v1, v2);
135
136         return new HQuaternion(v.x, v.y, v.z, s);
137     }

```

En el ejemplo anterior; línea de código 124, notaremos que se ha declarado un nuevo método estático llamado « **Multiplication** », el cual recibe dos argumentos: dos cuaterniones. Dentro de su ámbito, se implementa la operación detallada en la Figura 3.1.g de la sección anterior, la cual ilustra la multiplicación de dos cuaterniones.

Hasta esta instancia, hemos definido varios puntos en la ventana Scene que forman en conjunto un cubo geométrico. Dado que cada vértice corresponde a un vector tridimensional, será necesario declarar un método que nos permita asignar una nueva posición a cada punto y, de esta manera, lograr el efecto de rotación. Para ello, declararemos un nuevo método estático llamado « **Rotate** ». Este método tomará como argumentos la posición de los puntos del cubo, tres ejes de rotación y un ángulo. ¿Por qué el método es público? Porque lo utilizaremos más adelante dentro del método « **SceneGUI** » para rotar los vértices definidos para el cubo.

```
124 > private static HQuaternion Multiplication (HQuaternion q1,  
    HQuaternion q2) ...  
137  
138     public static Vector3 Rotate (Vector3 point, Vector3 axis,  
        float angle)  
139     {  
140         HQuaternion q = Create(angle, axis);  
141         HQuaternion _q = Conjugate(q);  
142         HQuaternion p = new HQuaternion(point.x, point.y,  
            point.z, 0f);  
143  
144         HQuaternion rotatedPoint = Multiplication(q, p);  
145         rotatedPoint = Multiplication(rotatedPoint, _q);  
146  
147         return new Vector3(rotatedPoint.x, rotatedPoint.y,  
            rotatedPoint.z);  
148     }
```

Siguiendo la ecuación presentada en la Figura 3.1.q, si nos fijamos en la línea de código número 140, veremos que se ha declarado un nuevo Cuaternión « **HQuaternion** » llamado « **q** », el cual se inicializa con el resultado del método « **Create** ». A continuación, en la línea de código 141, se declara otro Cuaternión llamado « **\_q** », el cual obtiene el conjugado del Cuaternión de rotación declarado previamente. Luego, en la línea de código 142, se declara un nuevo Cuaternión que representa el punto a rotar.

Finalmente, en la línea de código 144, se declara un nuevo Cuaternión llamado « **rotatedPoint** », que se obtiene multiplicando « **q** » y « **p** », y luego multiplicándolo por el conjugado de « **q** ».

Es importante destacar que la nueva posición de un punto corresponde a un vector tridimensional. Por esta razón, si observamos nuevamente la línea de código número 142, veremos que el componente « **w** » del Cuaternión « **p** » se inicializa en 0f. Del mismo modo, el método « **Rotate** » devuelve un vector tridimensional.

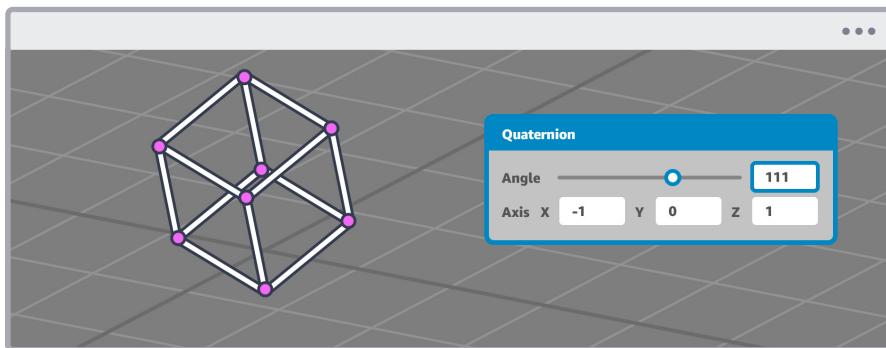
Lo único que falta es aplicar la rotación a los vértices. Para hacerlo, debemos ir al método « **SceneGUI** » y llamar al método « **Rotate** » para cada vértice definido en el cubo.

```
52     public void SceneGUI(SceneView view)
53     {
54 >         vertices = new List<Vector3> ... ;
55
56         float angle = m_angle * Mathf.PI / 180;
57
58         for (int i = 0; i < vertices.Count; i++)
59         {
60             vertices[i] = HQuaternion.Rotate(vertices[i],
61                 m_axis, angle);
62             Handles.SphereHandleCap(0, vertices[i],
63                 Quaternion.identity, 0.1f, EventType.Repaint);
64         }
65
66         int[][] index = ... ;
67
68         for (int i = 0; i < index.Length; i++) ...
69     }
70
71 }
```

Si nos fijamos en la línea de código número 70, veremos que hemos utilizado el método « **Rotate** » para cambiar la posición de cada vértice definido en la lista de vectores. Por razones lógicas, este proceso se lleva a cabo dentro del bucle « **for** », antes de dibujar las esferas que definen la posición gráfica de cada vértice.

Para el ángulo de rotación, se ha definido una nueva variable llamada « **angle** », que se multiplica por PI (3.14159265f) y se divide por 180, ¿Por qué se realiza esta operación? Como sabemos, por defecto, los cuaterniones calculan los ángulos en radianes. Por lo tanto, es necesario convertir el ángulo en grados a radianes para aplicar la rotación del Cuaternión.

Si regresamos a Unity y modificamos el valor del ángulo, observaremos que los vértices rotan siguiendo la orientación definida en los ejes.



(3.2.e)

## Resumen.

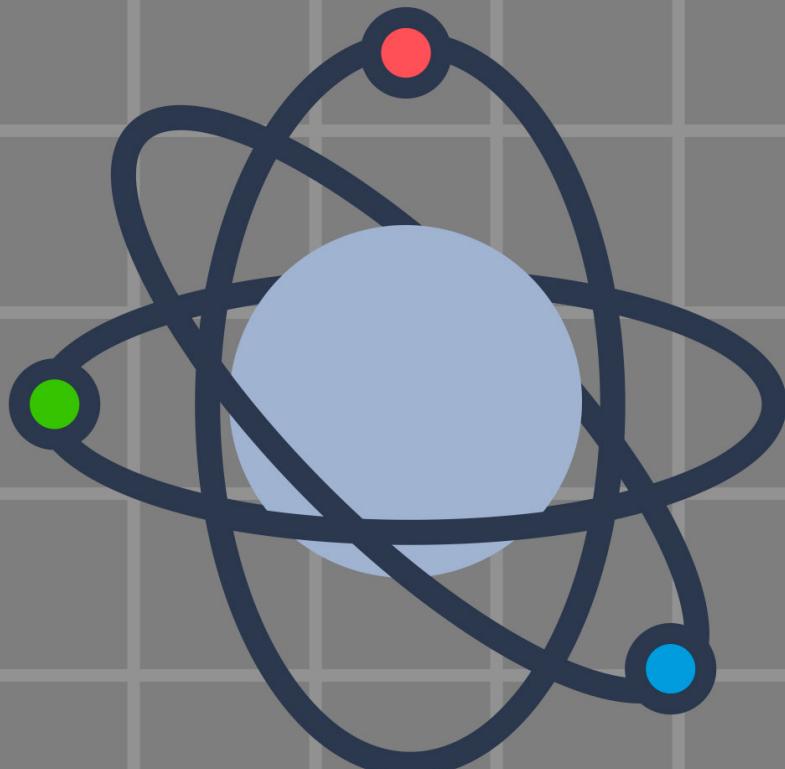
En este capítulo, exploramos el mundo de los cuaterniones, comenzando con su definición y estructura. Expusimos cómo los cuaterniones son una extensión de cuatro dimensiones de los números complejos y cuáles son sus propiedades únicas que los hacen adecuados para representar rotaciones en el espacio tridimensional.

Graficamos diversas operaciones con cuaterniones, centrándonos especialmente en su multiplicación y proporcionando su implementación en C# para mostrar cómo funcionan estas operaciones en la práctica.

Además, se introdujo el concepto del conjugado de un Cuaternion y cómo se relaciona con la inversión de rotaciones. Asimismo, discutimos cómo los cuaterniones pueden representar puntos en el espacio tridimensional y cómo son especialmente útiles para transformaciones espaciales.

Explicamos cómo realizar rotaciones utilizando cuaterniones de rotación, acompañados de un ejemplo sencillo que demuestra los cálculos matemáticos involucrados. Destacamos las ventajas de utilizar cuaterniones en lugar de matrices de rotación, evitando problemas como el bloqueo del cardán.

Finalmente, concluimos el capítulo creando una herramienta dentro de Unity que muestra la aplicación práctica de los cuaterniones. Utilizamos cuaterniones para rotar los vértices de un cubo en el espacio tridimensional, destacando su eficacia en el manejo de transformaciones espaciales complejas en la programación gráfica.



**Capítulo 4.**

# **Matrices de Rotación y Ángulos Euler.**

## 4.1. Introducción a la función.

En el capítulo anterior, revisamos los cuaterniones al aplicar una rotación a un cubo formado por ocho vértices previamente definidos. Cabe preguntarnos entonces: ¿hay otras formas de rotar vértices o puntos en el espacio? Dependiendo de las necesidades del proyecto que estemos llevando a cabo, las rotaciones mediante matrices pueden resultar de gran utilidad, especialmente si se aplican en gráficos por computadoras o en conversiones de ángulos de Euler a cuaterniones.

Las matrices de rotación constituyen herramientas matemáticas que posibilitan realizar rotaciones en el espacio tridimensional de una manera precisa y eficiente. Estas matrices se presentan como representaciones numéricas de las transformaciones que giran un objeto alrededor de un punto o eje en el espacio. Ahora, concentrémonos en la siguiente matriz para entender su definición:

$$r(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

(4.1.a)

A primera vista, la Figura 4.1.a podría parecer complicada de comprender. No obstante, al tener en cuenta su definición, podemos inferir que se trata de una matriz de rotación bidimensional. Esta matriz permite transformar la posición un punto mediante un ángulo medido en radianes. ¿Cómo es posible lograr esto? Para responder, necesitamos enfocarnos en las funciones trigonométricas « ***sin*** » y « ***cos*** », que, como sabemos, se emplean en matemáticas para relacionar los ángulos con las razones de los lados de un triángulo rectángulo.

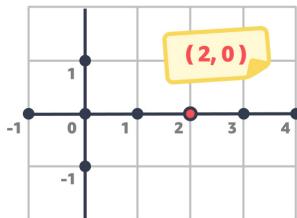
Considerando que « ***θ*** » (theta) representa un ángulo definido, podemos asumir que al multiplicar la matriz de la Figura 4.1.a por un punto bidimensional « ***p*** » ubicado en el plano, dicho punto cambiará de posición. El resultado de esta transformación será « ***p'*** », que es la denominación del punto después de haber sido rotado.

Realizaremos el siguiente ejercicio para entender el concepto: Consideremos un punto «  $p$  » en el espacio bidimensional, con las siguientes coordenadas,

$$p = (2, 0)$$

(4.1.b)

En un plano cartesiano, tal punto estaría ubicado de la siguiente manera,



(4.1.c)

Suponiendo que deseamos rotar  $45^\circ$  en sentido antihorario el punto «  $p$  » de la Figura anterior, será necesario tomar en consideración tanto el eje de rotación del punto, como el sistema de medida que utilicemos para llevar a cabo la rotación como tal. Por defecto, toda rotación de matrices se lleva a cabo en radianes, por lo tanto, será necesario transformar el ángulo a radianes antes de aplicar la transformación de matriz. Para ello, prestaremos atención a la siguiente regla de transformación:

$$\theta = \Omega * \frac{\pi}{180}$$

(4.1.d)

Donde «  $\Omega$  » (omega) corresponde al ángulo dado en grados. Para nuestro caso en particular, realizaremos la transformación mostrada en la Figura 4.1.d, considerando que «  $\Omega$  » es igual a  $45^\circ$ .

$$\theta = 45 * \frac{3.14159265f}{180}$$

(4.1.e)

Por lo tanto,

$$\theta = 0.78539816$$

(4.1.f)

Una vez que hemos obtenido el valor de rotación en grados, simplemente debemos aplicar el valor de la Figura 4.1.f directamente en la matriz presentada en la Figura 4.1.a. Por practicidad, únicamente agregaremos los primeros tres decimales a la ecuación, quedando de la siguiente manera:

$$r(\theta) = \begin{pmatrix} \cos(0.785) & -\sin(0.785) \\ \sin(0.785) & \cos(0.785) \end{pmatrix}$$

(4.1.g)

Una vez resuelto el ángulo, tendríamos que multiplicar la matriz de rotación por el punto previamente definido en la Figura 4.1.b, ¿cómo haríamos esto? Para ello, será fundamental considerar que la matriz que estamos utilizando como ejemplo, posee dos filas y dos columnas. Dado que las matrices pueden ser multiplicadas siempre y cuando la cantidad de columnas de la primera matriz sea igual a la cantidad de filas de la segunda, será necesario visualizar al vector «  $p$  » como una matriz de dos filas y una columna, es decir,

$$\mathbf{p} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

(4.1.h)

Teniendo en cuenta lo mencionado anteriormente, podemos llevar a cabo la multiplicación de la siguiente manera:

$$\mathbf{p}' = \begin{pmatrix} \cos(0.785) & -\sin(0.785) \\ \sin(0.785) & \cos(0.785) \end{pmatrix} * \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

(4.1.i)

Donde «  $\mathbf{p}'$  » corresponde al nuevo punto; a aquel que se obtiene luego de haber multiplicado la matriz de rotación «  $\mathbf{r}$  » por el punto «  $\mathbf{p}$  ». Cabe preguntarnos entonces: ¿cuál sería el orden de multiplicación de estos dos objetos? Considerando que estamos calculando la posición de un nuevo punto en un plano cartesiano, vamos a necesitar el valor de las coordenadas «  $x$  » e «  $y$  » del punto «  $\mathbf{p}$  ». En consecuencia, necesitaremos visualizar mediante variable e índices, los valores de la matriz definida anteriormente. De esta manera podremos entender de mejor manera el orden de la multiplicación en la operación que llevaremos a cabo.

$$\mathbf{p}' = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

(4.1.j)

Si desarrollamos el producto matricial mostrado anteriormente, obtendremos como resultado las siguientes ecuaciones para cada componente,

$$\begin{aligned}x' &= (a_{11} * x) + (a_{12} * y) \\y' &= (a_{21} * x) + (a_{22} * y)\end{aligned}$$

(4.1.k)

Que es lo mismo decir,

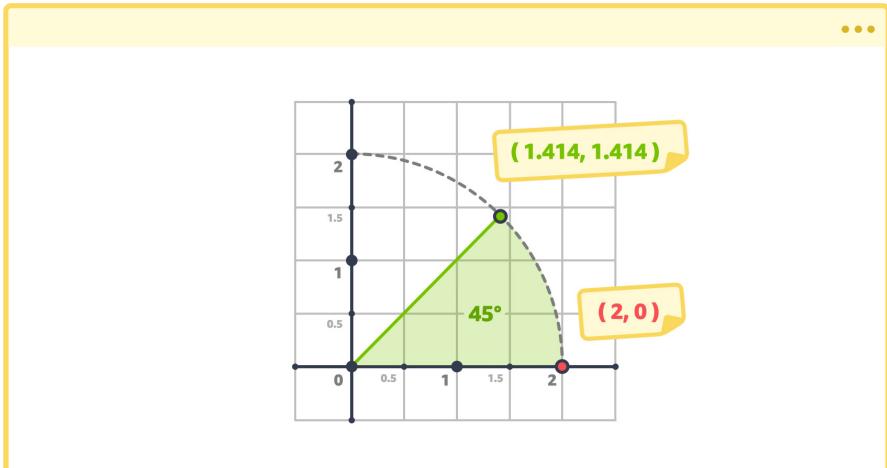
$$\begin{aligned}x' &= (\cos(0.785) * 2) + (-\sin(0.785) * 0) \\y' &= (\sin(0.785) * 2) + (\cos(0.785) * 0)\end{aligned}$$

(4.1.l)

Al realizar las operaciones aritméticas correspondientes, nos encontramos que las nuevas coordenadas para el punto «  $p'$  » son:

$$\begin{aligned}x' &= \mathbf{1.414213562} \\y' &= \mathbf{1.414213562}\end{aligned}$$

(4.1.m)



(4.1.n)

La implementación en código de una matriz de rotación va a depender de las necesidades de la operación o resultado que deseemos obtener. Sin embargo, la podríamos desarrollar de la siguiente manera para obtener la igualdad de la Figura 4.1.j.

```
Vector2 RotationMatrix2D(Vector2 p,
float angle)
{
    float a = angle * (Mathf.PI / 180);
    Matrix4x4 m = new Matrix4x4();

    m[0, 0] = Mathf.Cos(a);
    m[0, 1] = -Mathf.Sin(a);
    m[1, 0] = Mathf.Sin(a);
    m[1, 1] = Mathf.Cos(a);

    return m * p;
}
```

$$\mathbf{p}' = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

(4.1.o)

Como se puede observar en la Figura anterior, el método « **RotationMatrix2D** » devuelve la multiplicación de una matriz de cuatro por cuatro dimensiones, denotada como « **m** », por un vector bidimensional « **p** ». Este último hace referencia a la posición inicial del punto que se desea rotar. Cabe destacar que en C# no existen matrices de dos por dos dimensiones. Por esta razón, se ha implementado una matriz de cuatro por cuatro dimensiones y se han empleado únicamente cuatro índices para almacenar los valores en la lista.

Este tipo de matrices son generalmente utilizadas en gráficos por computadoras para crear efectos visuales interactivos mediante Shaders. Con ellas, es posible rotar las coordenadas UV de una figura geométrica. Por ejemplo, el siguiente método representa la implementación del nodo « **Rotate** » en grados, incluido en Shader Graph.

```
1 void Unity_Rotate_Degrees_float(float2 UV, float2 Center, float
2 Rotation, out float2 Out)
3 {
4     Rotation = Rotation * (3.1415926f/180.0f);
5     UV -= Center;
6     float s = sin(Rotation);
7     float c = cos(Rotation);
8     float2x2 rMatrix = float2x2(c, -s, s, c);
9     rMatrix *= 0.5;
10    rMatrix += 0.5;
11    rMatrix = rMatrix * 2 - 1;
12    UV.xy = mul(UV.xy, rMatrix);
13    UV += Center;
14    Out = UV;
15 }
```

Considerando que una matriz bidimensional contiene solamente un eje de rotación, en este caso el eje « **Z** », surge la pregunta sobre qué sucedería en el contexto de una matriz tridimensional. Cuando nos referimos a matrices de rotación en 3D, hablamos de matrices de tres filas y tres columnas. Cada elemento en esta matriz representa cómo cada coordenada del objeto será influenciada por la rotación alrededor de un eje que previamente ha sido definido.

La creación de una matriz tridimensional dependerá del eje alrededor del cual se desea realizar la rotación y del ángulo de rotación que se pretende aplicar. Por ejemplo, en una rotación en torno al eje « **X** », la matriz de rotación adoptará una estructura específica que impactará en las coordenadas « **Y** » y « **Z** » del objeto. La misma lógica se extiende a las rotaciones en los ejes « **Y** » y « **Z** ».

Concentraremos nuestra atención a las siguientes matrices para entender este concepto, especialmente en relación con la configuración de rotación para el eje « **Z** ».

$$r_z(\psi) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(4.1.p)

Configuración de rotación para el eje « **X** ».

$$r_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

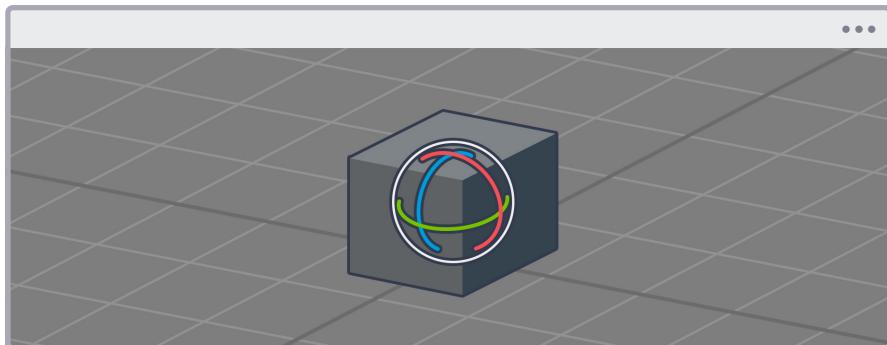
(4.1.q)

Configuración de rotación para el eje « ***Y*** ».

$$r_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

(4.1.r)

Como podemos observar en las Figuras anteriores, las matrices presentan una variación en su configuración según el eje espacial que deseemos utilizar, lo cual es bastante práctico, dado que, podríamos orientar a nuestro objeto en una dirección específica, definida por tres ángulos; tres valores distintos.



(4.1.s)

De tal comportamiento surge la pregunta: ¿cómo lograr la rotación de un objeto en tres ángulos distintos? Para responder a esto, debemos adentrarnos en el mundo de los ángulos de Euler, los cuales constituyen una forma común de representar orientaciones en el espacio tridimensional. Estos ángulos consisten en conjunto de tres valores angulares que se utilizan para describir una secuencia de rotaciones. Estos ángulos se denotan de la siguiente manera:

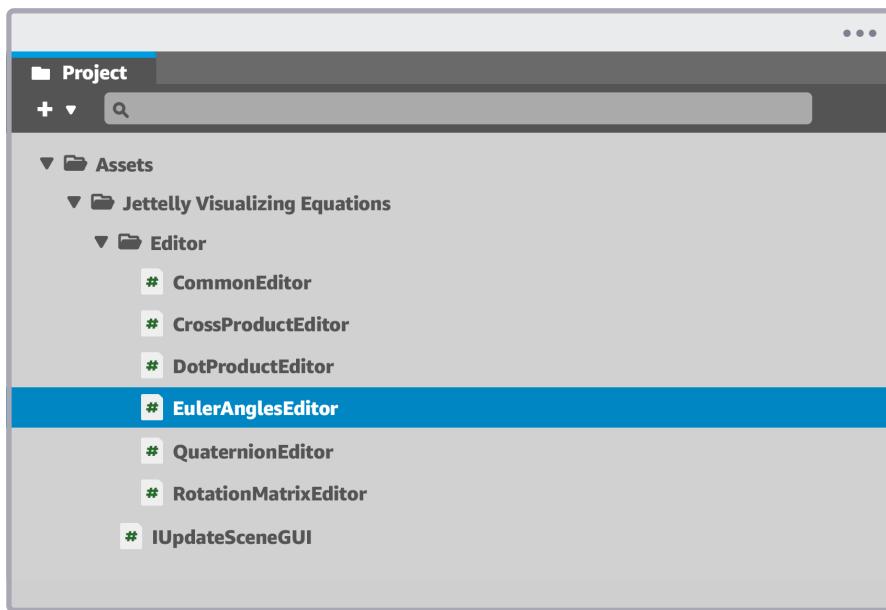
- « ***ψ*** » (Psi) para el eje « ***Z*** ».
- « ***θ*** » (Theta) para el eje « ***Y*** ».
- « ***ϕ*** » (Phi) para el eje « ***X*** ».

Un aspecto notable es el orden en el que se multiplican estos ángulos, ya que esto incide directamente a la orientación final del objeto. En otras palabras, según la convención que escojamos, obtendremos diferentes secuencias de rotación, por ejemplo: « **ZYX** », « **XYZ** », « **YZX** », entre otras. Es fundamental tener en mente que los ángulos de Euler pueden dar lugar a problemas de singularidad, ocasionando lo que se conoce como "el bloqueo del cardán" o "Gimbal Lock". Por esta razón, en aplicaciones más avanzadas, es aconsejable trabajar con cuaterniones, ya que ofrecen una mayor flexibilidad en la manipulación de rotaciones.

## 4.2. Desarrollando una herramienta en Unity.

Una vez más, nos sumergiremos en la elaboración de una herramienta que facilite la visualización de los ángulos Euler. De esta manera, lograremos una comprensión más profunda del fenómeno conocido como "bloqueo del cardán". Con este fin, procederemos a implementar las matrices que fueron mencionadas en la sección anterior.

Para iniciar esta unidad, vamos a generar un nuevo script en nuestro proyecto al que daremos el nombre de « **EulerAnglesEditor** », con el propósito de agilizar nuestro proceso. Dado que, una vez más, se trata de un script de tipo « **EditorWindow** », resulta esencial situarlo en la carpeta Editor, previamente creada en nuestro proyecto.



(4.2.a)

Una vez abierto el archivo, será necesario asegurarnos de extender nuestro script desde « **CommonEditor** », con el fin de integrar las funcionalidades de « **EditorWindow** ». Además, incorporaremos la clase abstracta « **IUpdateSceneGUI** », dado que nuevamente haremos uso del método « **SceneGUI** ».

Como ya sabemos, resultará fundamental declarar un método público y estático para exhibir la ventana de nuestra herramienta en el Editor. Con objetivos prácticos, repetiremos parte del proceso que previamente implementamos cuando creamos la función « **ShowWindow** » en nuestro script.

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using UnityEditor;
4
5  public class EulerAnglesEditor : CommonEditor, IUpdateSceneGUI
6  {
7      [MenuItem("Tools/Euler Angles")]
8      public static void ShowWindow()
9      {
10          GetWindow(typeof(EulerAnglesEditor), true,
11                      "Euler Angles");
12      }
13
14      public void SceneGUI(SceneView sceneView)
15      {
16      }
17  }
18
```

De manera similar, incluiremos los métodos « **OnEnable** », « **OnDisable** » y « **OnGUI** ». En este último, inicializaremos nuestras variables para que puedan ser mostradas en la ventana que previamente hemos declarado. Por otro lado, en los dos primeros métodos, llamaremos al método « **SceneGUI** » utilizando la función « **SceneView.duringSceneGui** ».

```

13  private void OnGUI()
14  {
15
16  }
17
18  private void OnEnable()
19  {
20      SceneView.duringSceneGui += SceneGUI;
21  }
22
23  private void OnDisable()
24  {
25      SceneView.duringSceneGui -= SceneGUI;
26  }
27

```

A continuación, procederemos a definir tres nuevos métodos privados que serán empleados en la definición de las matrices «  $\psi$  », «  $\theta$  » y «  $\phi$  ». Siguiendo convenciones establecidas, utilizaremos los nombres « **GetYaw** », « **GetPitch** » y « **GetRoll** », donde cada uno de ellos determinará un ángulo de rotación para los vértices de un objeto en el espacio tridimensional.

- « **GetYaw** » efectuará la rotación de los vértices en torno al eje « **Z** ».
- « **GetRoll** » realizará la rotación de los vértices en el eje « **X** ».
- « **GetPitch** » llevará a cabo la rotación de los vértices en el eje « **Y** ».

La tarea de definir las matrices podría ser abordada de diversas formas, como emplear múltiples listas de valores. No obstante, en esta ocasión optaremos por el uso del tipo de objeto « **Matrix4x4** », el cual está disponible en « **UnityEngine** ». Como ya conocemos, este tipo de objeto corresponde a una matriz estándar de cuatro filas por cuatro columnas.

```
33     Matrix4x4 GetYaw(float angle)
34     {
35         float cosTheta = Mathf.Cos(angle);
36         float sinTheta = Mathf.Sin(angle);
37
38         Matrix4x4 m = new Matrix4x4();
39
40         m[0, 0] = cosTheta;
41         m[0, 1] = -sinTheta;
42         m[0, 2] = 0;
43
44         m[1, 0] = sinTheta;
45         m[1, 1] = cosTheta;
46         m[1, 2] = 0;
47
48         m[2, 0] = 0;
49         m[2, 1] = 0;
50         m[2, 2] = 1;
51
52         return m;
53     }
54
55     Matrix4x4 GetPitch(float angle)
56     {
57         float cosTheta = Mathf.Cos(angle);
58         float sinTheta = Mathf.Sin(angle);
59
60         Matrix4x4 m = new Matrix4x4();
61
62         m[0, 0] = cosTheta;
63         m[0, 1] = 0;
64         m[0, 2] = -sinTheta;
65
66         m[1, 0] = 0;
67         m[1, 1] = 1;
68         m[1, 2] = 0;
69
70         m[2, 0] = sinTheta;
71         m[2, 1] = 0;
72         m[2, 2] = cosTheta;
```

Continúa en la siguiente página

```
73     return m;
74 }
75
76 Matrix4x4 GetRoll(float angle)
77 {
78     float cosTheta = Mathf.Cos(angle);
79     float sinTheta = Mathf.Sin(angle);
80
81     Matrix4x4 m = new Matrix4x4();
82
83     m[0, 0] = 1;
84     m[0, 1] = 0;
85     m[0, 2] = 0;
86
87     m[1, 0] = 0;
88     m[1, 1] = cosTheta;
89     m[1, 2] = -sinTheta;
90
91     m[2, 0] = 0;
92     m[2, 1] = sinTheta;
93     m[2, 2] = cosTheta;
94
95     return m;
96 }
97
```

Como podemos apreciar en el ejemplo anterior, cada valor ha sido incorporado en las respectivas filas y columnas, siguiendo la estructura inherente a cada matriz. Es relevante señalar que, dado que no se ha efectuado conversión alguna en el ángulo (argumento) éste último se calculará por defecto en radianes. Por lo tanto, más adelante, será necesario aplicar la función que se presenta en la Figura 4.1.d de la sección previa, con fin de llevar a cabo las rotaciones en radianes.

Proseguiremos con el desarrollo de nuestra herramienta al declarar e inicializar tres valores flotantes. Estos valores serán empleados en la definición de los ángulos.

```
5     public class EulerAnglesEditor : CommonEditor, IUpdateSceneGUI
6     {
7         [Range(-180, 180)] public float m_angleX = 0;
8         [Range(-180, 180)] public float m_angleY = 0;
9         [Range(-180, 180)] public float m_angleZ = 0;
10
11     private SerializedObject obj;
12     private SerializedProperty propAngleX;
13     private SerializedProperty propAngleY;
14     private SerializedProperty propAngleZ;
15
16     [MenuItem("Tools/Euler Angles")]
17 >     public static void ShowWindow() ...
18
```

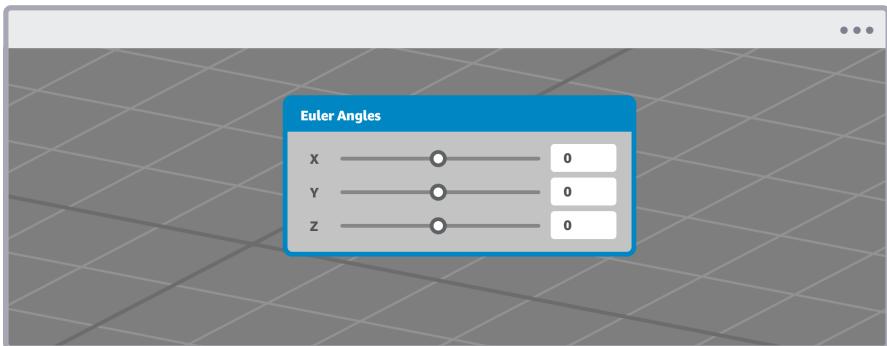
Si dirigimos nuestra atención a las líneas de código 7, 8 y 9, notaremos que nuestras variables flotantes están limitadas a un rango que abarca desde -180 a 180 grados. ¿Cuál es la razón detrás de esta elección? Siendo más específico, ¿por qué este rango?

Aunque este ejercicio podría ser abordado empleando un vector de tres dimensiones « **Vector3** », por razones de practicidad, optaremos por simplificarlo al rotar los ángulos en una totalidad de 360 grados.

Siguiendo adelante, continuaremos con la fase de inicialización y proyección de nuestras variables en la ventana de la herramienta en sí. Para lograrlo, intervendremos tanto en el método « **OnGUI** » como en « **OnEnable** », ejecutando la siguiente operación.

```
22  private void OnGUI()
23  {
24      obj.Update();
25
26      DrawBlockGUI("X", propAngleX);
27      DrawBlockGUI("Y", propAngleY);
28      DrawBlockGUI("Z", propAngleZ);
29
30      if (obj.ApplyModifiedProperties())
31      {
32          SceneView.RepaintAll();
33      }
34  }
35
36  private void OnEnable()
37  {
38      obj = new SerializedObject(this);
39
40      propAngleX = obj.FindProperty("m_angleX");
41      propAngleY = obj.FindProperty("m_angleY");
42      propAngleZ = obj.FindProperty("m_angleZ");
43
44      SceneView.duringSceneGui += SceneGUI;
45  }
46
```

Tal como ya tenemos conocimiento, el método « **DrawBlockGUI** » fue previamente definido al interior de la clase « **CommonEditor** ». Su principal propósito radica en dibujar bloques de información en la interfaz gráfica (GUI) de nuestra herramienta. Al regresar a Unity, observaremos que los ángulos de rotación se encuentran presentes en nuestra GUI.



(4.2.b)

Hasta este punto, únicamente nos estaría faltando dibujar algunos vértices en la ventana Scene a modo de visualizar las distintas rotaciones y ángulos de Euler. Para ello, declararemos tres listas « **List** », cada una para los distintos ejes de rotación, es decir: « **XYZ** ».

```

7      [Range(-180, 180)] public float m_angleX = 0;
8      [Range(-180, 180)] public float m_angleY = 0;
9      [Range(-180, 180)] public float m_angleZ = 0;
10
11     private SerializedObject obj;
12     private SerializedProperty propAngleX;
13     private SerializedProperty propAngleY;
14     private SerializedProperty propAngleZ;
15
16     private List<Vector3> circleX;
17     private List<Vector3> circleY;
18     private List<Vector3> circleZ;
19     private List<Vector3> arrow;
20

```

Si dirigimos nuestra atención al ejemplo anterior (líneas de código 16, 17 y 18), podremos observar la declaración independiente de listas para cada eje. Para ilustrar este concepto, incorporaremos puntos que, al unirse, conformarán polígonos destinados a visualizar un círculo. El propósito es presentar de manera

tangible la dinámica de la rotación de vértices en un espacio bidimensional en tiempo real. Cada una de estas representaciones gráficas encarnará un ángulo de rotación, manteniendo la misma orientación predeterminada de Unity: « **XYZ** »

Además, en la línea de código 19, se ha añadido una lista adicional denominada « **arrow** ». Esta lista será empleada para trazar una flecha en la escena, lo que permitirá visualizar la orientación actual de los ángulos de manera más precisa.

Continuaremos inicializando nuestras listas dentro del método « **SceneGUI** ». Para ello, incluiremos ocho puntos (vértices) dentro de cada una de nuestras listas, considerando sus coordenadas espaciales.

```

58     public void SceneGUI(SceneView sceneView)
59     {
60         circleY = new List<Vector3>
61         {
62             new Vector3( 0.00f, 0f,-1.00f),
63             new Vector3( 0.71f, 0f,-0.71f),
64             new Vector3( 1.00f, 0f, 0.00f),
65             new Vector3( 0.71f, 0f, 0.71f),
66             new Vector3( 0.00f, 0f, 1.00f),
67             new Vector3(-0.71f, 0f, 0.71f),
68             new Vector3(-1.00f, 0f, 0.00f),
69             new Vector3(-0.71f, 0f,-0.71f),
70         };
71
72         float degreeY = -m_angleY * MathF.PI / 180f;
73
74         for (int i = 0; i < 8; i++)
75         {
76             circleY[i] = GetPitch(degreeY) * circleY[i];
77             Handles.color = Color.green;
78             Handles.SphereHandleCap(0, circleY[i],
79             Quaternion.identity, 0.05f, EventType.Repaint);
80         }

```

Continúa en la siguiente página

```

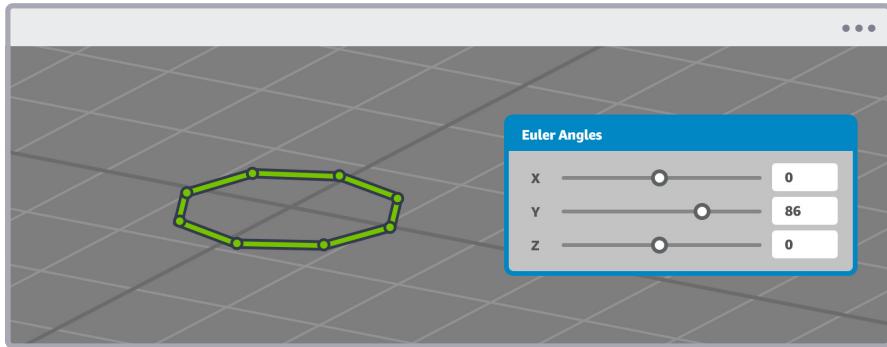
81      for (int i = 0; i < 8; i++)
82      {
83          Handles.color = Color.green;
84          Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
85                                     circleY.Count]);
86      }
87

```

Como se puede apreciar en la Figura anterior, específicamente en las líneas de código 62 a 69, hemos incorporado un total de ocho puntos en la lista llamada « **circleY** ». Esta lista está diseñada para representar visualmente la rotación en el eje « **Y** » de nuestra herramienta. Luego, en el primer bucle « **for** » (líneas 74 a 79) dibujamos una esfera mediante la función « **SphereHandleCap** », por cada punto presente en la lista. Estas esferas tienen la capacidad de rotar alrededor de un punto de referencia. Esto se logra debido a que, como se puede observar en la línea de código número 76, « **circleY** » se define como la multiplicación de la matriz « **GetPitch** » por cada punto en la lista.

Es importante mencionar que la rotación que aplicamos se realiza en "radianes". Esto se debe a que « **GetPitch** » toma como argumento a « **degreeY** », quien a su vez es igual al ángulo multiplicado por el resultado de dividir « **PI** » entre 180f (línea 72).

Dentro del segundo bucle « **for** » (líneas 81 a 85), se observa la creación de líneas conectando cada punto con el siguiente en la función de las intersecciones presentes en la lista que se ha inicializado previamente. Este proceso tiene como objetivo visualizar gráficamente los lados del polígono que presenta una aproximación visual del círculo. Este procedimiento nos ayudará a tener una mejor comprensión de la herramienta. Si guardamos los cambios y volvemos a Unity, notaremos que el eje « **Y** » ha sido agregado en nuestra escena. Además, podremos ajustar su orientación al modificar el valor de la propiedad « **Y** » de nuestra herramienta.



(4.2.c)

A continuación, volveremos a nuestro script y agregaremos el eje « **X** » siguiendo la misma lógica planteada anteriormente, la cual consiste en:

- Inicializar los puntos que definen un polígono en la lista.
- Iterar cada uno de ellos para dibujar una esfera en la posición de cada punto.
- Iterar nuevamente cada punto para trazar líneas entre ellos.

```

58     public void SceneGUI(SceneView sceneView)
59     {
60 >         circleY = new List<Vector3> ... ;
71
72         circleX = new List<Vector3>
73         {
74             new Vector3(0f, 1.00f, 0.00f) * 0.9f,
75             new Vector3(0f, 0.71f, -0.71f) * 0.9f,
76             new Vector3(0f, 0.00f, -1.00f) * 0.9f,
77             new Vector3(0f, -0.71f, -0.71f) * 0.9f,
78             new Vector3(0f, -1.00f, 0.00f) * 0.9f,
79             new Vector3(0f, -0.71f, 0.71f) * 0.9f,
80             new Vector3(0f, 0.00f, 1.00f) * 0.9f,
81             new Vector3(0f, 0.71f, 0.71f) * 0.9f,
82         };
83

```

Continúa en la siguiente página

```
84     float degreeY = -m_angleY * MathF.PI / 180f;
85     float degreeX = m_angleX * MathF.PI / 180f;
86
87     for (int i = 0; i < 8; i++)
88     {
89         circleY[i] = GetPitch(degreeY) * circleY[i];
90         Handles.color = Color.green;
91         Handles.SphereHandleCap(0, circleY[i],
92             Quaternion.identity, 0.05f, EventType.Repaint);
93
94         circleX[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
95             circleX[i]);
96         Handles.color = Color.red;
97         Handles.SphereHandleCap(0, circleX[i],
98             Quaternion.identity, 0.05f, EventType.Repaint);
99     }
100
101    for (int i = 0; i < 8; i++)
102    {
103        Handles.color = Color.green;
104        Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
105            circleY.Count]);
106    }
107 }
```

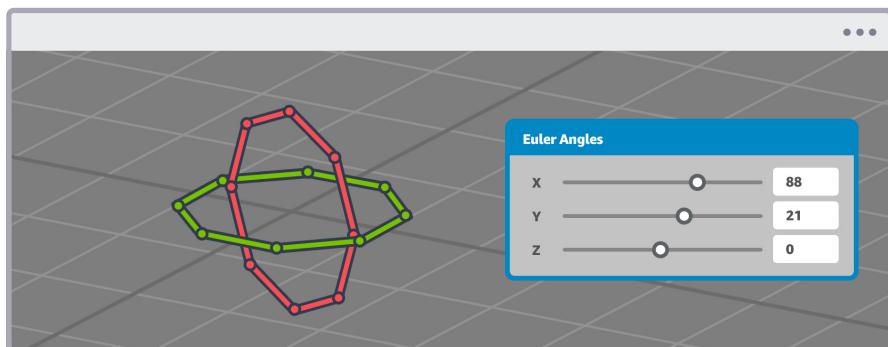
Del ejemplo anterior, si prestamos atención a las líneas de código 74 a 81, nuevamente hemos incluido puntos previamente definidos dentro de la lista « `circleX` » para generar un polígono que, en este caso, nos ayudará a visualizar el ángulo de rotación alrededor del eje «  $X$  ». Cabe destacar que cada punto ha sido multiplicado por 0.9f principalmente para disminuir el radio del polígono que forman los puntos en su conjunto. Es cierto que podríamos haber declarado e inicializado una variable con tal valor para evitar código repetido, sin embargo, en esta ocasión se ha realizado el ejercicio siguiendo fines educativos.

Un factor por considerar corresponde al cálculo del ángulo Euler para el eje « **X** ». Como podemos ver en la línea de código 93, la operación se realiza en dos multiplicaciones:

- Primero, se multiplica la matriz « **GetRoll** » por cada punto de la lista.
- Luego, se multiplica el resultado de la operación por la matriz « **GetPitch** ».

Esto ocurre debido a la peculiar interacción de los ángulos de Euler. Cuando rotamos en torno al eje « **Y** », el eje « **X** » también rota, como si estuvieran vinculados entre sí. Este mismo tipo de interacción se presenta en relación con el eje « **Z** ». Estas conexiones particulares generan complicaciones al trabajar con Euler, ya que un eje de giro puede perder relevancia en ciertas situaciones, lo que conduce a lo que conocemos como bloqueo de cardán. Este efecto puede dificultar la correcta representación de la orientación en el espacio.

Al regresar a Unity, notaremos que el eje « **X** » ahora está presente en la ventana Scene. Además, tenemos la posibilidad de ajustar su orientación mediante la propiedad « **x** » en nuestra herramienta.



(4.2.d)

Continuaremos la inicialización de nuestra última matriz, la cual corresponde al eje « **Z** ». Para lograrlo, incorporaremos una vez más ocho puntos que han sido previamente definidos, los cuales serán añadidos a la lista denominada « **circleZ** ». Estos puntos generarán un nuevo polígono en la ventana Scene.

```

58     public void SceneGUI(SceneView sceneView)
59     {
60 >         circleY = new List<Vector3> ... ;
61
62 >         circleX = new List<Vector3> ... ;
63
64         circleZ = new List<Vector3>
65         {
66             new Vector3( 0.00f, 1.00f, 0f) * 0.8f,
67             new Vector3( 0.71f, 0.71f, 0f) * 0.8f,
68             new Vector3( 1.00f, 0.00f, 0f) * 0.8f,
69             new Vector3( 0.71f,-0.71f, 0f) * 0.8f,
70             new Vector3( 0.00f,-1.00f, 0f) * 0.8f,
71             new Vector3(-0.71f,-0.71f, 0f) * 0.8f,
72             new Vector3(-1.00f, 0.00f, 0f) * 0.8f,
73             new Vector3(-0.71f, 0.71f, 0f) * 0.8f,
74         };
75
76         float degreeY = -m_angleY * MathF.PI / 180f;
77         float degreeX = m_angleX * MathF.PI / 180f;
78         float degreeZ = m_angleZ * Mathf.PI / 180f;
79
80         for (int i = 0; i < 8; i++)
81         {
82             circleY[i] = GetPitch(degreeY) * circleY[i];
83             Handles.color = Color.green;
84             Handles.SphereHandleCap(0, circleY[i],
85             Quaternion.identity, 0.05f, EventType.Repaint);
86
87             circleX[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
88             circleX[i]);
89             Handles.color = Color.red;
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107

```

Continúa en la siguiente página

```

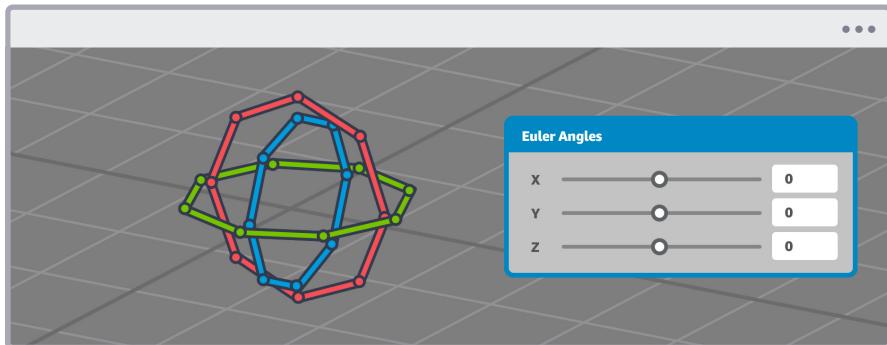
108         Handles.SphereHandleCap(0, circleX[i],
109                         Quaternion.identity, 0.05f, EventType.Repaint);
110         circleZ[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
111                         (GetYaw(degreeZ) * circleZ[i]));
112         Handles.color = Color.cyan;
113         Handles.SphereHandleCap(0, circleZ[i],
114                         Quaternion.identity, 0.05f, EventType.Repaint);
115     }
116     for (int i = 0; i < 8; i++)
117     {
118         Handles.color = Color.green;
119         Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
120             circleY.Count]);
121         Handles.color = Color.red;
122         Handles.DrawAAPolyLine(circleX[i], circleX[(i + 1) %
123             circleX.Count]);
124         Handles.color = Color.blue;
125     }
126 }
```

Como podemos observar en la operación previa (líneas 86 a 93), hemos repetido el mismo procedimiento una vez más, pero en esta ocasión, se aplica al eje « **Z** ». Es decir, estamos inicializando cada punto en la lista « **circleZ** » y luego iterando a través de estos puntos en los dos bucles « **for** » correspondientes.

Si nos enfocamos en la línea de código 110, podemos notar que para calcular el tercer eje se realizan al menos tres multiplicaciones:

- Primero, multiplicamos la matriz « **GetYaw** » por cada punto en la lista.
- Luego, multiplicamos el resultado anterior por la matriz « **GetRoll** ».
- Finalmente, multiplicamos toda la operación anterior por la matriz « **GetPitch** ».

De esta manera, las rotaciones se ven influidas por las rotaciones que las preceden, lo que genera el comportamiento característico de los ángulos de Euler. Al regresar a Unity podremos observar los distintos ejes de rotación con sus respectivos ángulos y propiedades. De hecho, si configuramos los ángulos numéricamente como « **90°, 0°, 0°** » perderemos un ángulo de rotación, lo que dará lugar al bloqueo del cardán.



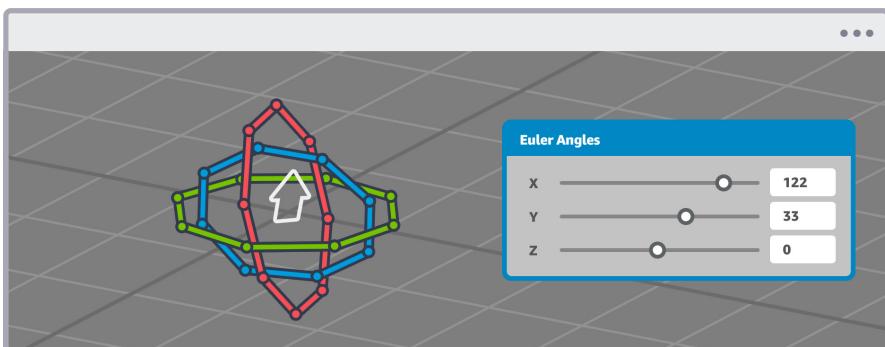
(4.2.e)

Procederemos a la inicialización de la lista « **arrow** », que fue declarada previamente. Esta lista puntos que ya han sido definidos y que, en su conjunto, forman la representación visual de una "flecha". Utilizaremos una flecha para visualizar la orientación de los ángulos de Euler. Para llevar a cabo este proceso, nos ubicaremos al final del bloque de código en el método « **SceneGUI** » y añadiremos las siguientes líneas de código.

```
115     for (int i = 0; i < 8; i++)
116     {
117         Handles.color = Color.green;
118         Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
119             circleY.Count]);
120
121         Handles.color = Color.red;
122         Handles.DrawAAPolyLine(circleX[i], circleX[(i + 1) %
123             circleX.Count]);
124
125     }
126
127     arrow = new List<Vector3>
128     {
129         new Vector3( 0.20f, 0f, 0.00f) * 0.5f,
130         new Vector3( 0.20f, 0f,-0.50f) * 0.5f,
131         new Vector3( 0.35f, 0f,-0.50f) * 0.5f,
132         new Vector3( 0.00f, 0f,-1.00f) * 0.5f,
133         new Vector3(-0.35f, 0f,-0.50f) * 0.5f,
134         new Vector3(-0.20f, 0f,-0.50f) * 0.5f,
135         new Vector3(-0.20f, 0f,-0.00f) * 0.5f,
136     };
137
138     for (int i = 0; i < arrow.Count; i++)
139     {
140         arrow[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
141             (GetYaw(degreeZ) * arrow[i]));
142     }
143
144     for (int i = 0; i < arrow.Count; i++)
145     {
146         Handles.color = Color.white;
147         Handles.DrawAAPolyLine(arrow[i], arrow[(i + 1) %
148             arrow.Count]);
```

En el código previo, existen tres bloques en los que es importante concentrarse. La lista « `arrow` » ha sido inicializada entre las líneas 127 y 136. Cada vector contenido en la misma, ha sido multiplicado por 0.5f con el fin de reducir el tamaño final de la representación de la flecha. Luego, en el bucle « `for` » (líneas 138 a 141), iteramos a través de cada punto y llevamos a cabo el mismo proceso que empleamos para el eje « `Z` » de nuestra herramienta. En este caso, multiplicamos cada punto por « `GetYaw` », después por el resultado obtenido de esa operación multiplicado por « `GetRoll` », y finalmente por « `GetPitch` ». Esto nos proporciona la orientación de los ángulos de Euler. Para finalizar, declaramos un nuevo bucle (líneas 143 a 147) para trazar una línea que conecta cada punto de la lista.

Al regresar a Unity podremos observar la funcionalidad completa de la herramienta que hemos estado desarrollando en este capítulo.



(4.2.f)

## Resumen.

En este capítulo, exploramos y definimos las matrices de rotación, comenzando con la matriz de rotación bidimensional aplicada a un punto en el plano. Esto nos sirvió como ejemplo para ilustrar la representación matricial de un punto en el espacio y comprender la multiplicación de matrices para obtener el punto transformado.

Basándonos en la base de las matrices de rotación bidimensionales, ampliamos nuestro estudio al espacio tridimensional, definiendo rotaciones alrededor de cada uno de los ejes correspondientes. De esta manera, pudimos representar y transformar la orientación de un objeto utilizando ángulos de Euler.

Siguiendo la convención de Unity, pudimos realizar secuencias de rotación, garantizando coherencia en nuestro enfoque para aplicar ángulos de Euler.

Concluimos con la creación de una herramienta en Unity en la que construimos tres anillos geométricos como representación de un gimbal. Además, explicamos el problema del bloqueo de cardán, que ocurre cuando se pierde un ángulo de rotación.

Finalmente, destacamos las ventajas de utilizar matrices de rotación, especialmente al tratar con pérdida de ángulos. Comprender estos conceptos es fundamental para la programación eficiente en gráficos 3D.

## Conclusión.

A lo largo de este libro, hemos explorado un conjunto fundamental de conceptos matemáticos que son esenciales en el mundo de la programación y arte técnico. A través de un análisis detallado, hemos desglosado las ecuaciones del Producto Punto, Producto Cruz, Cuaterniones y Ángulos de Euler. Además, hemos demostrado cómo estas herramientas matemáticas pueden ser aplicadas para resolver una amplia gama de problemas en el desarrollo de aplicaciones interactivas y desarrollo de videojuegos.

Comenzamos con el **Capítulo 1**, donde profundizamos en el Producto Punto. Aprendimos acerca de su naturaleza y aplicaciones, ilustramos conceptos con ejemplos sencillos, programamos en C# para realizar sumatorias y creamos una herramienta en el editor para visualizar su comportamiento.

En el **Capítulo 2**, nos adentramos en el Producto Cruz. Iniciamos definiendo su operación a través de la regla de la mano derecha y, además, ejemplificamos su implementación mediante transformaciones lineales utilizando matrices. Posteriormente, exploramos sus propiedades geométricas y finalizamos con una herramienta de visualización tridimensional para mejorar la comprensión de este concepto.

Luego, en el **Capítulo 3**, nos introdujimos en el fascinante mundo de los Cuaterniones, una herramienta matemática versátil con aplicaciones en rotaciones tridimensionales. Exploramos su estructura, operaciones y su capacidad para representar transformaciones espaciales.

Finalmente, en el **Capítulo 4**, abordamos las matrices de rotación y los ángulos de Euler. Comenzamos con matrices de rotación bidimensionales y ampliamos nuestro conocimiento a rotaciones de tres dimensiones utilizando los ángulos de Euler. Discutimos la convención de Unity para secuencias de rotación y los desafíos del bloqueo del cardán (efecto Gimbal). De la misma manera que en capítulos anteriores, concluimos con la creación de una herramienta en Unity

que nos permitió visualizar el comportamiento de su función y experimentar con la pérdida de un ángulo de rotación.

Es importante destacar que el mundo de la programación y desarrollo está en constante evolución. Sin embargo, independientemente de las nuevas tecnologías, la comprensión de estos fundamentos matemáticos seguirá siendo esencial para los desarrolladores y artistas técnicos. Al dominar las ecuaciones matemáticas presentadas en este libro, los lectores estarán mejor equipados para adaptarse a los desafíos cambiantes y para innovar en sus propios proyectos.

## Glosario.

**Función:** Una función es una relación matemática que asigna cada elemento de un conjunto; un dominio, a un elemento en otro conjunto o codominio. Si la relación es uno a uno, es decir, que para cada elemento del dominio existe un único elemento en el codominio, la función es biyectiva. Puede describirse mediante una regla que asocia una entrada con una salida.

**Método:** En programación, un método es un conjunto de instrucciones o procedimiento que se aplican a un objeto o una case para realizar una tarea específica.

**Operación:** En matemáticas, una operación es una acción o procedimiento que se aplica a uno o más valores para obtener un resultado. Pueden ser aritméticas, lógicas, algebraicas, entre otras.

**Commutatividad:** En matemáticas, significa que el orden en una operación no importa. Por ejemplo: Es lo mismo decir  $1 + 2$ , que  $2 + 1$ .

**Sumatoria:** En matemáticas, es un operador que permite representar sumas de muchos sumandos, incluso infinitos. Se expresa con la letra griega  $\Sigma$  sigma.

**Variable:** Una variable es un símbolo que representa un valor que puede cambiar en una ecuación, fórmula o programa. Puede tomar diferentes valores durante la ejecución del código.

**Sistema de referencia:** Corresponde a un conjunto de coordenadas espaciales que se requieren para poder determinar la posición de un punto en el espacio.  
**Producto Punto:** También conocido como "Producto Escalar", es una operación matemática entre dos vectores que resulta en un valor escalar.

**Producto Cruz:** También conocido como "Producto Vectorial", es una operación matemática entre dos vectores que resulta en un nuevo vector perpendicular a los vectores originales.

**Anticonmutatividad:** Una propiedad matemática en la que el resultado de una operación cambia de signo cuando se altera el orden de los elementos involucrados. En una operación anticonmutativa, realizar la operación en el orden inverso produce el mismo resultado, pero con un signo negativo. Por ejemplo, dada la siguiente operación,  $a * b$ , puede ser igual a  $-(b * a)$ .

**Polígono:** Es una figura geométrica plana y cerrada, compuesta por segmentos de línea recta llamados lados.

**Cuaterniones:** Un Cuaternion es un tipo de número complejo que consta de una parte real y tres números imaginarios. Es utilizado especialmente en gráficos 3D y mecánica.

**Matrices de rotación:** Son matrices utilizadas para representar transformaciones de rotación en el espacio tridimensional. Estas matrices se utilizan para cambiar la orientación de objetos tridimensionales alrededor de un punto de referencia, como el origen.

**Euler:** Los ángulos de Euler son un conjunto de tres ángulos que describen la orientación tridimensional de un objeto. Estos ángulos se utilizan para especificar la rotación de un objeto en términos de cambios en sus ejes «**X**», «**Y**» y «**Z**».

**Ecuación:** Una ecuación es una igualdad matemática que contiene una o más incógnitas y expresa una relación entre ellas.

**Sigma:** La letra griega  $\Sigma$  sigma se utiliza para representar una suma en matemáticas. Se coloca delante de una serie de términos que se suman.

**Vector:** Un vector es una entidad matemática que tiene magnitud, dirección y sentido, generalmente representada como una flecha en el espacio.

**Componente de un vector o matriz:** Son los valores individuales que componen un vector o una matriz, representando información específica en la estructura matemática correspondiente.

**Script:** En programación, un script se refiere a un conjunto de instrucciones o comando que se ejecutan secuencialmente para realizar una tarea específica.

**GUI (interfaz gráfica de usuario):** Es una forma visual e intuitiva de interactuar con un programa o aplicación mediante ventanas, botones, menús, etc.

**Gizmo:** En gráficos por computadoras, un gizmo es una herramienta gráfica o ícono que se utiliza para manipular objetos en una escena tridimensional.

**Handler:** Corresponde a una función o rutina que se utiliza para gestionar eventos o acciones específicas en un programa.

**Anti-Aliasing:** Es una técnica utilizada en gráficos y renderizado para reducir el efecto de escalones en bordes diagonales. Su implementación mejora la calidad visual de la imagen final.

**Arcotangente:** Es una función trigonométrica inversa que devuelve el ángulo cuya tangente es un valor dado.

**Trigonometría:** Es una rama de las matemáticas que estudia las relaciones entre los ángulos y los lados de un triángulo.

**Radián:** Es una medida de ángulo utilizada en matemáticas y trigonometría, basada en la longitud del arco de una circunferencia.

**Stack:** En programación, un Stack (pila) es una estructura de datos que sigue el principio LIFO (last in, first out), donde el último elemento añadido es el primero en ser eliminado.

**Escalar-Vector:** Es una operación matemática donde se multiplica un escalar (número) por un vector, dando como resultado un nuevo vector con magnitud distinta, pero con la misma dirección y sentido.

**Eje:** En geometría y matemáticas, un eje es una línea de referencia alrededor de la cual se produce una rotación o reflexión.

**Coordenada:** Una coordenada es un conjunto de valores que determina la posición de un punto en el espacio, ya sea en el plano bidimensional o tridimensional.

**Valor escalar:** Corresponde a número que tiene magnitud, pero no dirección, es decir, no está asociado a un sistema de coordenadas.

**Matriz:** Es una tabla ordenada de elementos, dispuestos en filas y columnas, utilizada en matemáticas y programación para representar datos, y realizar operaciones lineales.

**IntelliSense:** Es una característica presente en algunos entornos de desarrollo que proporciona sugerencias automáticas de código, completando palabras clave, funciones y nombres de variables mientras se escribe.

**Espacio euclídeo:** Es un espacio geométrico donde se pueden aplicar conceptos de la geometría euclidiana, como la distancia, el ángulo y las propiedades de las figuras.

**Cardán:** Los ángulos del cardán son tres ángulos que describen la orientación de un objeto en el espacio tridimensional. También se les conoce como ángulos de Euler.

**Conjugación:** En matemáticas, la conjugación de un número complejo implica cambiar el signo de su parte imaginaria. En álgebra abstracta, la conjugación puede referirse a diferentes operaciones según el contexto.

**Valor absoluto:** El valor absoluto de un número es su magnitud sin considerar el signo, es decir, siempre es un número positivo.

**Interpolación:** La interpolación es una técnica utilizada para estimar un valor desconocido o no muestrado basándose en datos conocidos, utilizando métodos como el polinomio de Lagrange o el spline cúbico.

**Bloqueo del cardán:** Problema en sistemas de representación tridimensional que ocurre cuando dos de los tres ejes de rotación se alinean, reduciendo la libertad de movimiento a dos dimensiones.

# Agradecimientos Especiales.

A Al Kooheji | Abraham Armas Cordero | Adam A D K Carames | Adam Bennett | Adam Gibson | Adam Myhre | Adrian Cuneo | Adrian Devlinn | Adriano Valle | Adrien Kissennfennig | Afif Faris | Ahmet Usta | Ahren Foreman | Ajin Russel Raj | Alavuotunki Pekka Juhani | Albin Lundahl | Aleada Dzalia | Alejandro Allende | Alejandro García Guillot | Alejandro Hidalgo Acuna | Alexander Ewetumo | Alexander Froelich | Alexander Galloway | Alexander 'Gruni' Grunert | Alexandre Lagallarde | Alexandru Geana | Alexis Gonzalez | Alice Bottino | Alisia Martinez | Alvaro G. Lorenzo | Alyne Kelly Gois | Amit Netanel | Ana Perez Sierra | Andreas Zimmer | Andres | Andres Felipe Garcia | Andres Mendez Del Rio | Andrew Fellows | Andrey Valkov | Angel Ortiz | Anna Kocer | Anne Postma | Antao Almada | Anton Bandarenka | Anton Budnychuk | Anton Lazarets | Anton Sasinovich | Antonio Manzari | Arda Ozupek | Arkadiusz Kotarski | Army Vang | Aron Thompson | Artemii Ramzevich | Arthur Lopes | Arthur Monteiro | Arthur P V Salvador | Artiszin | Artur Shapiro | Arturo Zahar Alcibia | Ash Curkpatrick | Ashton Cross | Atakan Talay | Athanasios Zagkliveris | Austin Lothman | Aviad Biton | Avinash B | Aya Magdy Fawzy | Aziz Şekerdi | Baesungjin | Baglan Tolebay | Batın Seyrek | Bedrican Çalışkan | Ben Finkelstein | Ben Morgan | Benjamin A Brown | Benjamin Bouffier | Benjamin Koder | Benjamin Russell | Benny Franco | Benoît Valdes | Blas A Castañeda A | Boehrer Magali Claire | Braden Currah | Brandon Friend | Brent Carlin | Bret Bays | Brian Heinrich | Briceida Carillo | Brigitte Zheng | Bruno Costa | Bryce Paule | Caio Cesar Iglesias | Caio Hutter Cipó | Can Delibaş | Caner Coşkun | Caner Özdemir | Carlos A. Jaimes Vergara | Carolyn Stone | Catch Me Toys | César Augusto Fernández Cano | Cesar De Macedo | César Héctor | Chaosblare X | Charanjeet Singh Jaswani | Charlene Whittington | Chen Jingzhou | Chen Yimeng | Cheng Bowen | Chepe | Cheuk Hinyi | Chongyi | Chrisperrella | Christian Koch | Christian Sidor | Christopher Suffern | Christopher W Cannon | Christos Tzastas | Chusak Tan | Claudiu Barsan-Pipu | Clemens Pfauser | Cloud-Yo! | Coby Jennings | Cody Wilson | Cole Andress | Cole Q Azevedo | Connor Checkley | Cosmin Bararu | Cristian Thompson | Daghan Demirci | Damian Longman | Damian Turnbull | Đặng Trần

## Agradecimientos Especiales

Hải | Daniel | Daniel A Fisher | Daniel Garcia | Daniel Puentes |  
Daniel Ruiz Leyva | Daniel Tozer | Danimo | Darina Koycheva |  
David A Holland | David Alejandro Celis Pino | David Clabaugh |  
David Jumeau | David Nieves | David Treharne | David Vessup | Davit  
Badalyan | Davon Allen | Deehoi | Defrog | Degeneratewaste | Deinol |  
Delano Igbinoba | Dennis | Denis Smolnikov | Derek Brouwer | Dhaval Prajapati  
| Dina Khalil | Do Minh Triet | Douglas Kerr | Dragan Ignjatovic | Dragan Stamenkovic  
| Drew Fitzpatrick | Dylan Hunter | Ebru Sena Çatana | Ed Siomacco | Ediber J  
Reyes | Eduardo R - Iocusrise | Edward Ro | Edward Whitehead | Edwin J V  
Naranjo | Emery Sadler | Emett Speer | Emiliano Guzman M. | Emir Furkan Tokkan  
| Endri Kastrati | Enrico | Eren Göç | Eric Pattmon | Eric Rico | Eric W Nersesian  
| Eric Young | Erik Niese-Petersen | Esko Evtyukov | Esra Soylu | Esteban Jimenez  
| Etienne Virtualis | Etonix Pyro | Evan Hill | Evgeniy Novikov | Eyal Assaf | Fahrul  
Gamemaker | Felipe González | Felipe Papa Gimenes | Felipe R Silva | Feras  
| Fethi Isfarca | Florence Noe | Francisco Chagolla Angulo | Francisco J  
Estrada Salinas | Francisco Javier Tinoco Pérez | Francisco Ortega |  
Francois Dinh Quang | Franks Gonzalez | Fredrik Persson | Gabriel  
Gomez | Gabriela Bohorquez | Gabriele Boni | Gago Xachatryan  
| Garet Thomas | Gareth Bourn | Garry T Mcgee | George Castillo |  
George Katsaros | Germán Augusto | Gezihao | Gi Yong Park | Gilberto  
B P Junior | Gilberto B P Junior | Ginderbird | Giselle N O Silva | Greg Hendrix |  
Grzegorz Regliński | Guilherme Nunes | H Kotze | H. Carrasco Pagnossi | Halil  
Zeybek | Halo Field™ | Hamza Mohammed Rangoonia | Hatice Nur Efe | Hector  
Moscoso | Hello World | Henry L Quinones E | Herleen Dualan | Herman Garling  
Coll | Herschel Darko | Hien Nguyen | Hiroki Miyada | Hleb Shatrauka | Ho  
Cheng-Yi | Hugo Barrandon | Hugo Delgado | Hugo Tourneur | Humberto Gamboa  
| Huynh Dong | I N Cooper | Iain Pentelow | Ibrahim Yamaam | Ilina Bokareva |  
Ilyas Sadyrov | Invex 0 | Iram Maximiliano Lopez Guerrero | Ismael Bernard |  
Ismanart3D | Ivan Cardenas | Ivan Imerovic | J Campagna | J F  
Echeverría Pinto | Jack Haehl | Jacob James Dockter | Jakob  
Rendon | Jakub Slaby | James Please | Jan Schaumlöffel | Janesit  
Wongvorlachan | Jari | Jason Fotso-Puepi | Jason Peterson | Javier  
Eduardo Salcedo Rondón | Jayasurya Aasaithambi | Jean Ducellier

## Agradecimientos Especiales

| Jefferson Ferreira | Jengy Matantsev | Jesse Maccabe | Jessica Ambron | Jessica Canales | Jesus David Angarita | Jesus Hernandez Ortiz | Jesus Hernandez Ortiz | Jhoshua Ampo | Jing Yi Chong | Jing Yi Chong | Joan Toh | Joao L F Amorim | Jody Ruben | Joe Nickolls | Joel Freeman | John Fredy Espinosa | John Jones | John Reitze | Johnny M Roodt | Jonas Carvalho De Araujo | Jonatan Saari | Jonathan Jesus Cantor Gonzalez | Jonathan Keuchkarian | Jonathan Morales | Jonathan Richardson | Jonathan Smith | Jonathan Valderrama | Jordan Cox | Jordan Han | Jordan Shepherd | Jordi Moreno Lopez | Jorge Diaz Saez | Jorge L Chavez Herrera | Jose Aaron Meza Perez | José Antonio Victoria | Jose Domingo Ramirez Gutiérrez | Jose Jimenez | Jose Lopez | Jose M Dieck | Jose Miguel Casas Pagan | Jose Ramon Arias Gonzalez | Jose Rodrigo Martinez | Joseph Deluca | Joseph P Denike | Joshua Adrian Miles | Joshua Baillargeon | Joshua Baillargeon | Joshua D Horner | Joshua Hjelle | Joshua Petta | Jp Lee | Juan Camilo Cortes Esparza | Juan Carlos Barraza Mendo | Juan Carlos Horta | Juan Hernandez | Juan Luis Moreno | Juan Manuel Gonzalez Garcia | Juan Muniain Otero | Julia Caputa | Julius Fondem | Juris Savostijanovs | Jyoti Kamlakar Bhoir | Karan Mistry | Karim Castagnini | Karim Lachaize | Katerina Gramova | Keeevin Roussel | Kengo Ozawa | Kevin Willis | Kevin Zambrano | Kim Young Min | Kimkunbyo | Korintic | Kristopher Cigic | Kuntae Park | Kyrylo Samoilenko | L Tijmsma | Laise M Nogueira | Laith Hasan | Larry Fuhrmann | Lau Choi Sang | Lawrence Yip | Leathen | Lee Seungmin | Leevi Rantala | Lewis Nicholson | Li Jingtian | Li Kun Yi | Lim Donguk | Lin Li-Chin | Living Room Filmmaker | Loonatic | Lorenzo Bianciardi | Lucas Ferreira | Lucas Mcdermott | Lucia Gambardella | Luis Francisco Roa Castro | Luis Montero | Luis Urueta | Luiz Otavio Vaz | Luong Huu Tinh | Maciej Nabialczyk | Madeline McDougall | Mahmud Syakiran | Majed Mahmoud Ramadan Elwardy | Malik Aune | Malik Aune | Malin Anker | Manuel Uriel Olvera Castañeda | Manvendra Deora | Marc Hewitt | Marcin Kasica | Marcin Łuczak | Marcin Sadomski | Marco Arjona | Marco Secchi | Marcos Sanvitale | Marcos Wicket | Marcos Wicket | Maria Gonzalez | Mariia Chebotok | Mariya Zinchenko | Mark Steele | Markus Sebastian Bakken Storeide | Mateus S Pereira | Matheus Schon | Matt McMahon | Mattedickson

## Agradecimientos Especiales

| Matthew David Lee | Matthew J Strangio | Matthew Spencer | Max Krueger | Max Otto | Mb Net | Meera Sanghani | Melanie Tidler | Mher Vincent Viguilla | Micah Benson | Michael Aviles | Michael Clavan | Michael Eichenseer | Michael Fewkes | Michael Guerrero | Michael I Randrup | Michael J Marian | Michael Ross | Michael Woo | Michelle Moreno Arveras | Miguel Angel Bulnes Echave | Mikalai Danilenka | Mike Monroe | Miles J Barksdale | Minh Dia | Miquel Campos | Miquel Ferrer Mas | Miss Oona R Tukia | Miss V Johnson | Miyakou | Mo Yang | Mohit Sethi | Mohsen Tabasi | Monica Yaneth Loeb Willes | Moyu Nie | Mr James A Clark | Mr L P Boyd | Mr M T Georgiev | Mr R Naude | Mr Thomas Graveline | Mr W Scaife | Muhammad Azman | Muhammet Fatih Yılmaz | Munesadafumiki | Mustafa Erhan Serpek | Mustafa Memişoğlu | Mykyta Andropov | Namballa Durga Sandeep Varma | Nathanael De Jager | Nathaniel Biddle | Necati Akpinar | Nguyen Tuan Dat | Nhan Nguyen | Nicholas Chambers | Nicholas Jonathan Boyd | Nicholas M Stringer | Nicholas Routhier | Nicolas | Nicolás Alonso Acevedo Suzarte | Nicolas Dezubiria C | Niki Grunoski | Nikita Zhelezkov | Nikolai Matusevich | Ning Cai | Noelia Fernandez | Norbert Oleksy | O Grama | Ogulcan Topsakal | Olcay Daşer | Oleksii Dubrovskyi | Olivier Baron | Olivier P Beierlein | Omar Guendeli | Omar Rodríguez Pérez | Ömer Firat | Omgelsie | Ondřej Holan | Online Kort | Onur Can Erdil | Orkun Manap | Osakpemwokan Alonge | Osman Karaduman | Özkan Melen | Pablo José De Andrés | Pablo Trascasa | Palita Panyadee | Pamisetty Ranganath | Paritta Kijmahanon | Pariwat Phisittaphong | Parsa Jamshidi | Parsue Choi | Patryk Brzakalik | Patryk Cisek | Paul Killman | Paul Pop | Pavel Efimov | Pete Law | Peter Chen | Pherawat Puttabucha | Phoen Leo | Pia Guehne | Piotr Grechun | Pratna Meng | Privat | Przemysław George | Przemysław Przyłęcki | Radoslaw Polasik | Rafael Valentim | Ramiro Alurralde | Ranjit Menon | Raul Miguel Ruiz Esquer | Raveen Rajadorai | Raza Butt | Razvan Luta | Reira Ota | Rene Melendez | Richard Wallace | Rihards Valters | Rinat Enikeev | Rizal Ardianto Saleh | Robbiehowe | Robert Baffy | Robert May | Roberto Andres Estupinan Cuadrado | Roberto Macken | Robin Hoole | Robinson Enrique Rojas Rojas | Rodion Tabares | Rodrigo Moreira Abreu | Roman Hajdu | Rosenio Pinto | Roy Rodenhaeuser

## Agradecimientos Especiales

| Rt Neal | Rt3D | Ruchi Hendre | Ryosuke Motrgi | S Julien Gauthier  
| Sael | Saikirthi Velukumar | Samuel Luce | Samuel Trudgian |  
Samuel Wilton | Sandro Ponticelli | Sankar B | Sara Alzahrani | Sarah  
Young | Saurabh E Kachhap | Scott Benson | Sean Mcallister | Seán  
Walsh | Sebastian Emanuelsson | Semion Bojedai | Senliu | Seth Crawford  
| Simone Bembi | Siren | Sirrena Holmes | Skodje | Skodje | Skyler Fines | Soup  
Scythe | Sourav Chatterjee | Stefan Dieckmann | Stefan Groenewoud | Steffen  
| Stranger On The Road | Sujith R | Sunlight Technologies | Suthamon Hengrasmee  
| Swapnil Revankar | T. Yongprayoon | Takashi Nakamura | Takeru Kunimoto |  
Takumi Motoike | Takumi Tsukada | Tatiana Isupova Pr Novi Sad | Taylor L Ekena  
| Thach Quoc Khang | The Beast Makers | Theodoros Doukoulos | Thibaut Hunckler  
| Thomas Foster | Thomas Guyamier | Thomas Surin | Thomas Wormann | Timo  
Fettweiß | Timothy Lim | Timur Ariman | Timur Ariman | Tinnaput | Tiranice |  
Todd Akita | Tofulemon | Tom Keen | Tomas Gayo Perin | Tomas Jasinas |  
Tore Waldow | Trần Thiết Duy | Tristan Del Giudice | Tristan Lasfargue  
| Troy Aaron Richardson | Tyler Molz | Tyler Parker | Tyler Smith | Ufuk  
Apaydın | Uladzislau Daroshka | Umut Çetin Sağdıçoğlu | Unpsyside  
| Upendra Attarde | Vaclav Vancura | Valentin Boissel | Valentin  
Pantiukh | Varga Lorand Eugen | Vasilii Seleznev | Vasily Povalyaev  
| Victor H Cardona G | Victor Manuel Celis Padron | Victor Soler | Vincent  
Allen | Vinicius Monteiro | Vinicius Ramires Leite | Virtuos Vietnam | Vitalii  
Shaposhnikov | Vivek Goyal | Vladmir C. Souza | Vong Pha | Wajeeh Ul Hassan  
| Wang Qiang | Wataru Iizuka | Wei Zeng | Wesley Schneider | Wildeax | Will  
Andersen | Will Beard | William Ab | William Beltran | William C. Taylor | Wilmer  
Cedeno | Wojciech Kowalec | Woodys Fx | Yasmin Shitrit | Ybrayym Dathudayev  
| Yinon Ezra | Yolanda Afan | Yordan Kalbanov | Yoshiharu Sato | Youngmin Kim  
| Youssef Khaled | Yousung Kim | Yuichi Ishii | Yuichi Matsuoka |  
Yulia Trukhan | Zachariah Abueg | Zachary Alstadt | Zachary Chung  
| Zeptolab Barcelona Tamara Cecilia Ferreiro | Zeroonebit |  
Zhangbao | Виктор Григорьев | ネギ坊 | השם הסוביירום ורטובייצ' | 김현우  
| 엄재천



**“Jettelly te desea éxito en tu  
carrera profesional”.**