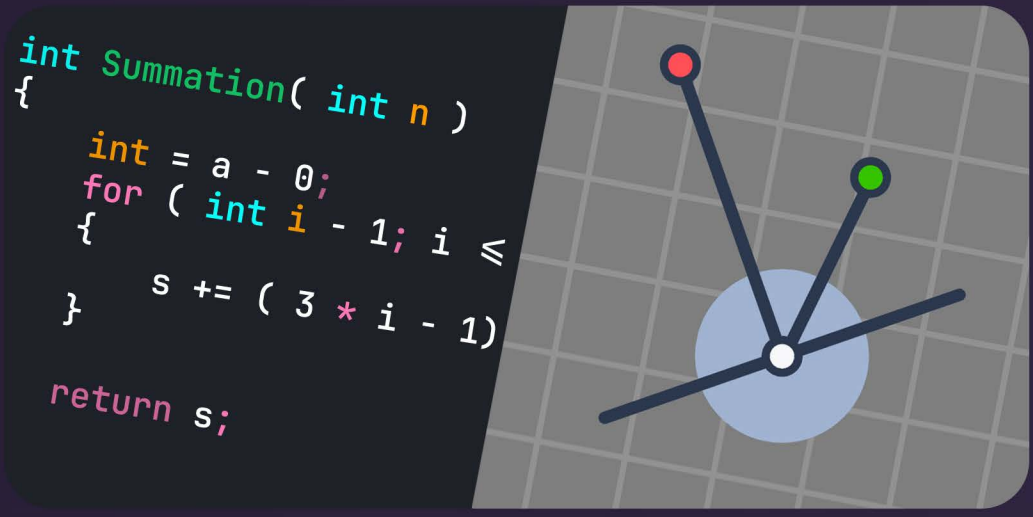
Learn. Create. Master.

Visualizing Equations - Vol. 1

Essential Math for Game Devs.

A visual book to help you understand key math concepts for game development.

#TechnicalArtist #GameDev #MadeWithUnity



Fabrizio Espíndola.


Jettelly

Visualizing Equations,
essential math for game devs.

A book of visual explanations that will help you understand essential mathematical concepts applicable to game development.

Visualizing Equations, - vol.1, version 0.0.8.

Author.
Fabrizio Espindola.

Design.
Pablo Yeber.

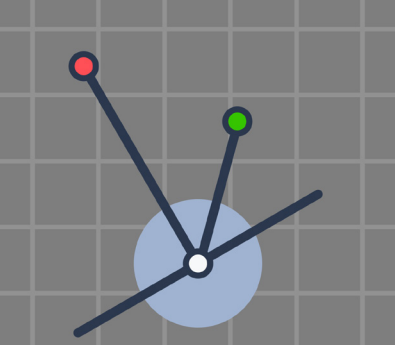
Technical review.
Martin Molina.

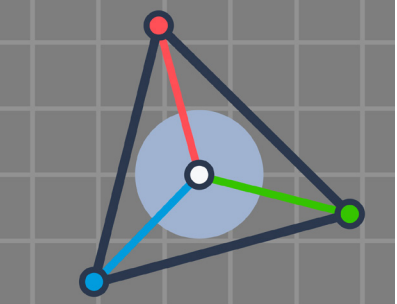
Translation, grammar, and spelling.
Martin Clarke.

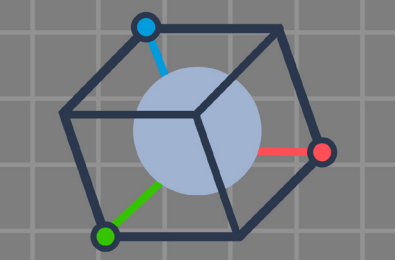
Acknowledgement.

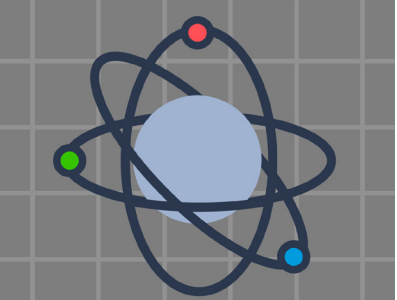
I would like to express my gratitude and dedicate this work to my parents, Marcia Vivas and Victor Espíndola, for giving me life; my siblings, Angela Espíndola, Franco Espíndola, and Camilo Espíndola, for being there for me during challenging times; to my wife, Aida Cid, and my son Santino Espíndola,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

~ Fabrizio Espíndola.









Content.

Preface.

Whether your goal is to develop a mechanic, design an algorithm, create a tool, or engage in other programming endeavors, a common practice that defines us as developers is the act of revisiting mathematical equations while attempting to solve programming challenges.

On more than one occasion, we have encountered the same challenge: How can we translate certain equations into code? Due to our nature, there are times when we simply cannot recall mathematical symbols or the order of operations, leading to distraction in our work or even frustration.

This book is designed to assist Unity developers in creating effective tools within the software for visualizing equations in their projects. Through a combination of theory and practical examples, readers will learn how to apply mathematical concepts and programming tools to create dynamic and engaging visualizations that illustrate mathematical concepts in a clear and accessible manner.

Who this book is for.

This book has been designed for Unity developers looking to enhance their understanding of mathematical functions, equation visualization, and the creation of professional tools. It is assumed that readers already have a basic knowledge of Unity, so we will not delve into details about its interface.

Although having prior experience in the C# programming language can help understand certain parts of the content, it is not a mandatory requirement for the reader.

It is recommended to have a basic foundation of knowledge in arithmetic and algebra to grasp some of the concepts that will be addressed throughout the book. Nevertheless, the book will include reviews of the mathematical operations and functions necessary to understand the material being developed fully.

Conventions.

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

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

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

```
4    public class ExampleClass : MonoBehaviour
5    {
6        void Start ()
7        {
8            // Code here …
9        }
10   }
```

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

Errata.

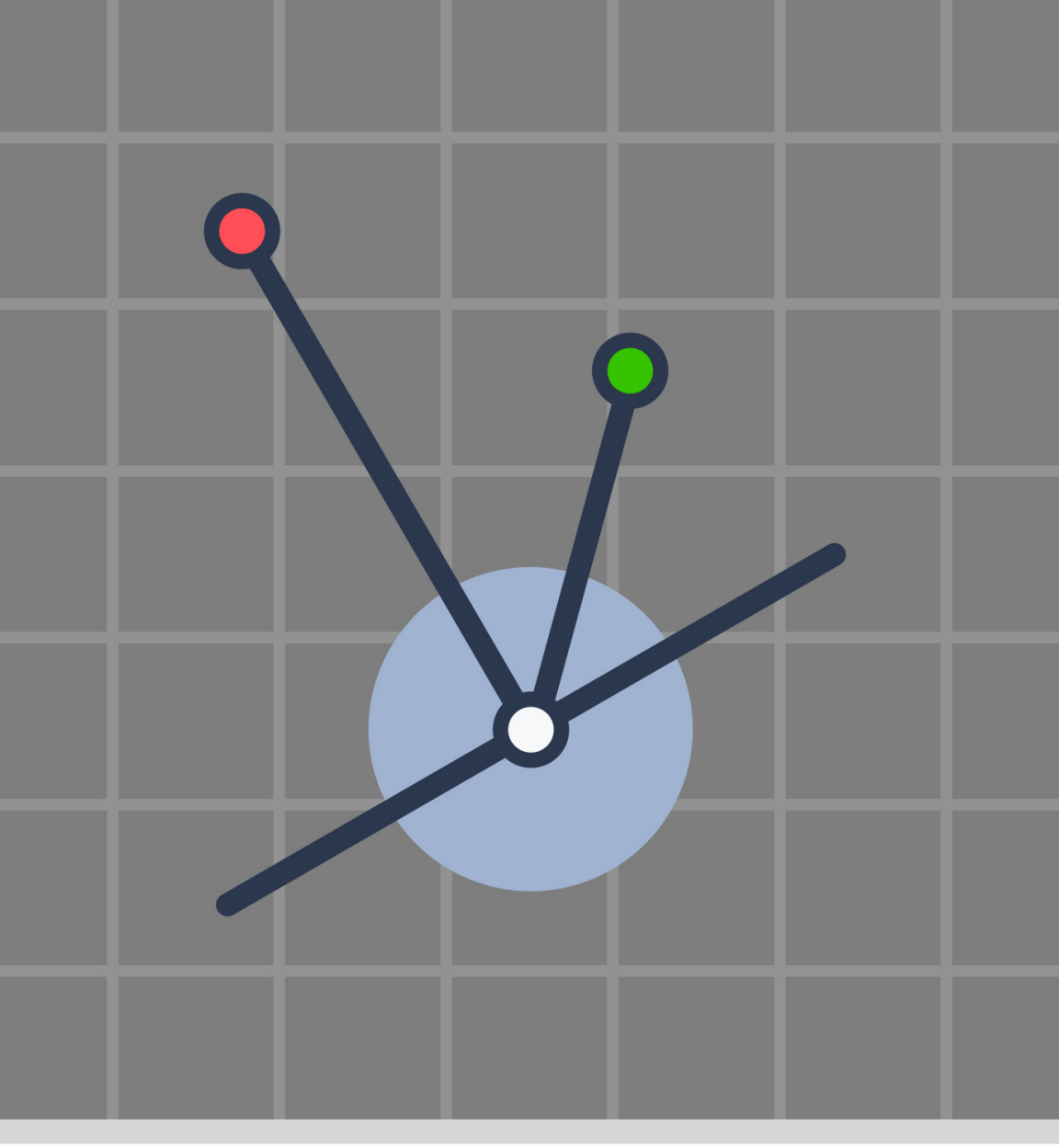
While writing this book, we have taken precautions to ensure the accuracy of its content. Nevertheless, you must remember that we are human beings, and it is highly possible that some points may not be well-explained or there may be errors in spelling or grammar.

If you come across a conceptual error, a code mistake, or any other issue, we appreciate you sending a message to **contact@jettelly.com** with the subject line "**VE1 Errata.**" By doing so, you will be helping other readers reduce their frustration and improving each subsequent version of this book in future updates. Furthermore, if you have any suggestions regarding sections that could be of interest to future readers, please do not hesitate to send us an email. We would be delighted to include that information in upcoming editions.

Piracy.

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

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



Chapter 1.

Dot Product.

1.1. Function introduction.

We begin the adventure by looking at the following equation:

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

(1.1.a)

Can you identify the function or operation to which the above equation belongs to? It is common to encounter such equations in programming-oriented books, where they are used to illustrate a function that produces a specific result.

Given the title of this chapter, you may have already deduced the operation to which the equation in Figure 1.1.a belongs. This equation represents the algebraic definition of the Dot Product (also known as the Scalar Product) of two n-dimensional vectors « A » and « B » defined in Euclidean space.

The symbol « Σ » (sigma) represents the summation of scalar terms in a sequence, i.e., a sum of constant values, complex numbers, or real numbers, starting at « i » and ending at « n », both being variables.

It is worth remembering that a variable, as the name suggests, refers to a value that varies over time, such as a person's age. Age can be 1, 2, 3, and so on, but it will never be a constant value. Why is that? Because, unfortunately, time marches on!

Now, proceed with the following exercise to understand the concept better,

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

(1.1.b)

In Figure 1.1.b, since « i » equals to 1, the first operation to perform would be (3 * 1 - 1), which results in 2. Subsequently, the value of « i » is replaced by each sequence in the exercise until it reaches « n », which in this case equals 5. The next calculation would be (3 * 2 - 1), and so on.

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

(1.1.c)

Which is the same as,

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

(1.1.d)

Therefore,

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

(1.1.e)

Now, returning to the equation shown in Figure 1.1.a, how could this be translated? It is relatively straightforward: the variable « n » refers to the vector's number of dimensions, and « i » corresponds to a label for each component. For instance, for a three-dimensional vector, the operation would look like this:

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

(1.1.f)

Which is the same as saying,

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

(1.1.g)

Therefore,

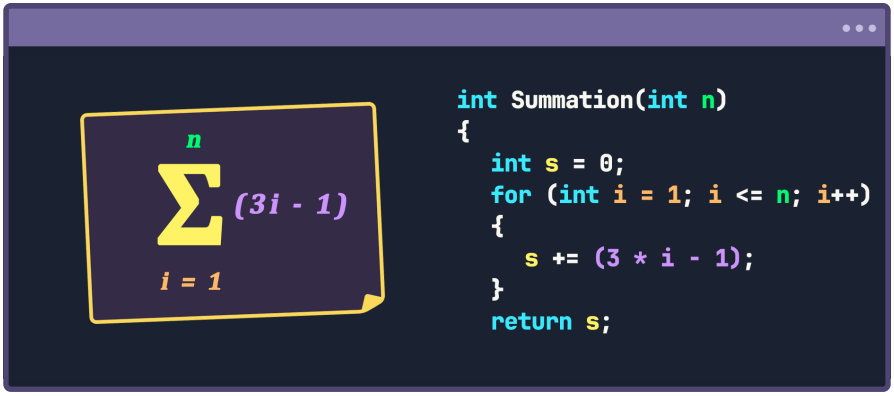
$$A \cdot B = (A_x * B_x) + (A_y * B_y) + (A_z * B_z)$$

(1.1.h)

Looking at Figure 1.1.f, notice that the variable « n » has been replaced by a constant value, which is equal to 3—the number of dimensions. This equation can be read as follows:

> **The Dot Product between vector A and vector B is equal to the sum of the products of each component.**

The summation implementation in code will depend on the context in which it is used. For example, you could use a « **for** » loop to obtain the equation shown in Figure 1.1.b.



(1.1.i)

As observed in the previous Figure, the « **Summation** » method returns 40 if « n » is equal to 5. However, when dealing with vectors, their implementation differs because, in this case, the result will depend on:

- Whether the vectors are points in space.
- Whether the vectors are directions in space.
- Whether you want to project one vector onto another.

Geometrically, the Dot Product corresponds to a projection of vector « A » over vector « B ». What does this mean? Imagine standing on a sunny day with a light source projecting your shadow onto the ground. Now, envision two vectors in three-dimensional space: vector « A » and vector « B ». The Dot Product between these two can be interpreted as the projection of one vector over the other, much like how light projects a shadow onto the ground.

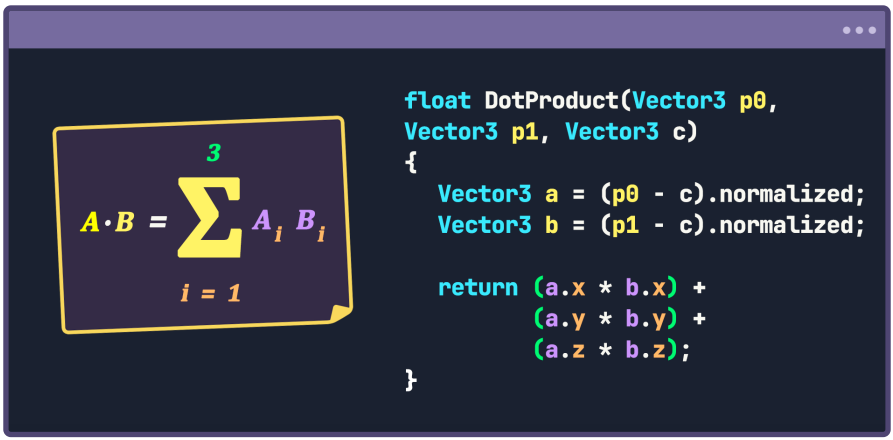
A summation can have different uses depending on the function it is meant to fulfill. The geometric interpretation of the Dot Product introduces another important equation, which is,

$$A \cdot B = |A||B|Cos\,\theta$$

(1.1.j)

From the previous Figure, this equation yields the same result as the equation in Figure 1.1.h, with the difference that, with the latter, you can obtain the angle « θ » between each vector.

Considering that the vectors « A » and « B » from Figure 1.1.a are directions in space, they can be implemented as in the following figure:



(1.1.k)

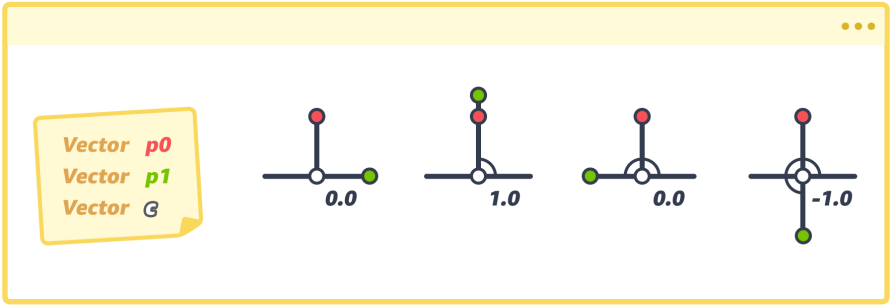
It is important to note that when working with vectors, you always need a reference point that acts as the origin of your reference system. In Figure 1.1.k, you can observe that in the « **DotProduct** » method, in addition to vectors « **a** » and « **b** », a new additional vector called « **c** » is used as an argument, which acts as a reference point between « **p0** » and « **p1** ».

By subtracting vector « **c** » from vectors « **p0** » and « **p1** », you can interpret that vectors « **a** » and « **b** » are directions in space. It is important to highlight that these vectors have been normalized so that their magnitude is equal to 1.

$$||\boldsymbol{a}|| = \mathbf{1}$$
$$||\boldsymbol{b}|| = \mathbf{1}$$

(1.1.l)

So, why can the Dot Product between two vectors be beneficial? It depends on what you are developing. For example, in video game mechanics, you can use this function to determine the direction of one object relative to another. The Dot Product between two-unit vectors returns the cosine value of the angle between them, so its result is within a range from -1f to 1f. Therefore, this result is determined based on the position of each vector relative to a reference point. It means you could use these values to establish whether an object is in front or behind another. How can you do this?



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

In the figure above, you can see the graphical representation of the Dot Product between two vectors, where the vector « **c** » marks the reference point. Assuming that the vector « **p0** » represents the main character and « **p1** » represents their enemy, you can notice that:

- If the Dot Product returns 1f, the enemy is in front of or above your character.
- If it returns -1f, the enemy is behind or below.

1.2. Developing a Unity tool.

Considering the abstract nature of the above explanation, a small tool will now be created to implement the equation presented in Figure 1.1.a from the previous section. This tool will help visualize the behavior of the Dot Product between two vectors.

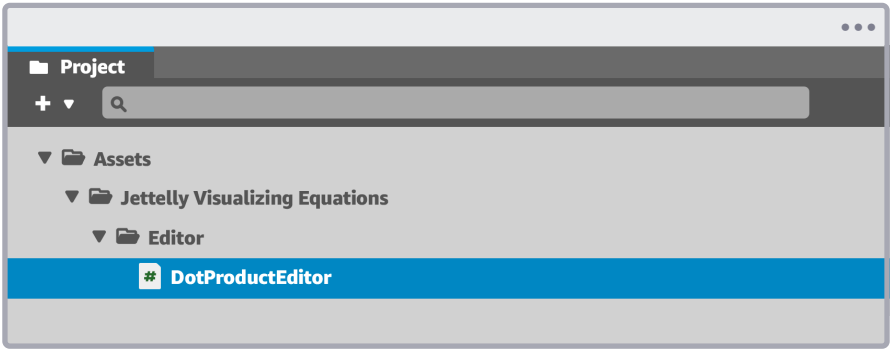
Start the process by going to your project (the Project window in Unity) and create a new script called « **DotProductEditor** ». Once open, make sure to extend it from the « **EditorWindow** » for two main reasons:

- Because it will be a visual tool.
- Because you will only need one instance of the same object in the Scene window.

As a result, it will be important to both use the « **UnityEditor** » dependency in the code and save the script within an « **Editor** » folder in the project. Why is that? According to the official software documentation:

> **Editor-type scripts add functionality to Unity during development but are unavailable in runtime builds.**

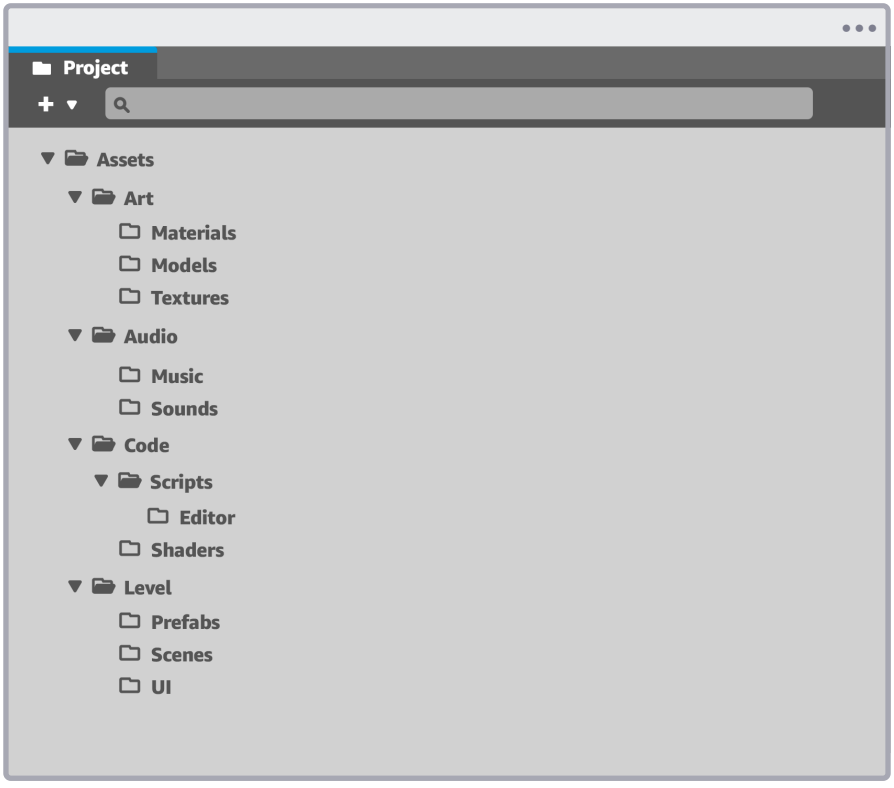
In other words, you cannot export a video game or application with Editor graphics.



(1.2.a)

In Unity, it is expected to encounter several Editor folders and subfolders within the Assets directory. While this feature provides some flexibility when defining the project's organization, it can also lead to issues if a coherent structure is not established. It can result in multiple Editor folders nested within subfolders throughout the project, making it challenging to locate scripts as the project expands.

To avoid this situation, Unity suggests following 'best practices' for organizing your project and provides the following structure as a reference.



(1.2.b)

Each time you create a new script, by default, it extends from « **`MonoBehaviour`** » and adds two methods:

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

However, these methods will not be used in this case since, as mentioned previously, your script will extend from « **EditorWindow** ». Therefore, it will be necessary to:

- Include the « **UnityEditor** » dependency.
- Extend the script from « **EditorWindow** ».
- Remove default functions.

The « **EditorWindow** » class is included in the « **UnityEditor** » dependency. The latter contains several classes that are useful in tool development, among which are:

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

```
using UnityEngine;
using UnityEditor;

public class DotProductEditor : EditorWindow
{
}
}

```

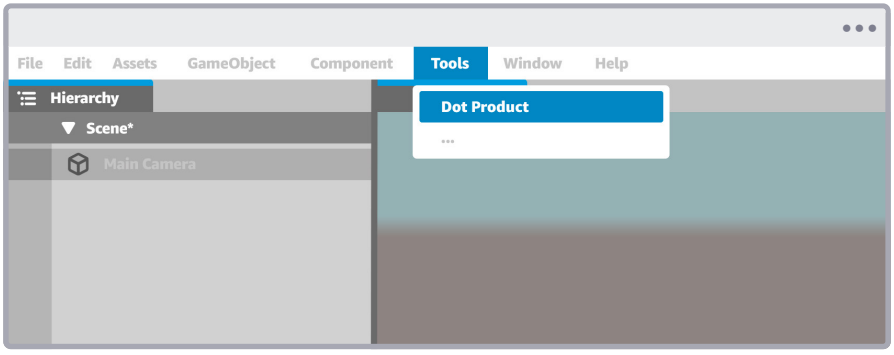
To officially begin the development process, add a method to display a window for your tool. To do this, perform the following steps as indicated by Unity on their web platform:

- Declare a public and static method to display a window in the Editor.
- Add the attribute « **`MenuItem`** » on the function, mentioning the menu path (where you want to list your tool).
- Create and display the new window in the main menu.

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

Following code line number 9, the static function « **`GetWindow`** », which returns the first « **`EditorWindow`** » of type « **`t`** » currently visible on the screen, has up to four arguments, of which you have used three in the example. The first refers to the window type « **`DotProductEditor`** ». The second is a Boolean value that determines whether your window will be a pop-up, and finally, the last argument corresponds to the title that the window will have at the top.

If everything has worked optimally, a new menu called 'Tools' will appear in Unity, corresponding to the definition made earlier using the « **`MenuItem`** » attribute in code line 6. From there, you can access the 'Dot Product' window, which currently does not perform any action.



(1.2.c)

Considering that your tool will be used to represent the behavior of the Dot Product, it will be necessary to declare three vectors in the code: « **`p0`** », « **`p1`** », and « **`c`** » as the reference point. Furthermore, to project the vectors in their respective window and dynamically modify their values, it will be necessary to declare some « **`SerializedProperty`** » type properties. This will ensure that your tool is flexible and efficient enough to represent the Dot Product's behavior accurately.

```
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
            (typeof (DotProductEditor), true, "Dot Product");
19          window.Show();
20      }
21  }
22
```

It will be necessary to include some functions in your code to see the vectors in action. When creating a tool, it is important to note that the user interface (GUI) and the graphics added to the scene must be programmed separately. As a result, some values need to be initialized, such as the starting position of the vectors. To achieve this, add the following methods to the code.

- « **`OnGUI`** »: This displays information and manages events for your tool in the Dot Product window.
- « **`SceneGUI`** »: This corresponds to your version of the « **`Editor.OnSceneGUI`** » method, which allows you to manage events in the Scene view.
- Another method to include is « **`OnEnable`** », which is used to initialize values when the tool is active.
- Finally, « **`OnDisable`** » is employed solely for unsubscribing from events.

```
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
```

It is worth noting that both the « **`OnEnable`** » and « **`OnDisable`** » methods, as well as « **`OnGUI`** », belong to « **`EditorWindow » which inherits from « ScriptableObject`** », meaning they are native to Unity. In the previous exercise, they were implemented in lines 22, 27, and 32.

One important consideration is that « **`SceneGUI`** » should be called every time the Scene view is updated. However, since it is not native, it will require an event to work correctly. That is why it has a « **`SceneView`** » type argument, which allows for various configurations, including subscribing to events.

```
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
```

As the name suggests, the « **`SceneView.duringSceneGui`** » event (lines 24 and 29) is updated during the use of the Scene view. Consequently, if you activate the Dot Product window and move the cursor within the Scene view area, you can debug the number of times the « **`SceneGUI`** » method is called.

The next step is to draw the vectors in the Scene view using the « **`Handles.FreeMoveHandle`** » method as the next step. However, remember that the vectors « **`m_p0`** », « **`m_p1`** », and « **`m_c`** » declared earlier have not yet been initialized. Therefore, if you perform the process at this point, they will all appear in the same position, corresponding to 'zero' in all their components. You will initialize the vectors in the « **`OnEnable`** » method to avoid this conflict:

```
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
```

The values assigned to the coordinates of « **`m_p0`** » and « **`m_p1`** » establish a reference point to carry out the exercise, so they can be adjusted as desired because they will not generate changes in the final result of the tool you are developing.

Once the values are initialized, continue painting each vector as 'Gizmos' in the Scene view. As mentioned earlier, is it necessary to use the « **`Handles.FreeMoveHandle`** » function because it returns a new position based on the user's interaction with the respective handler (the colored point that appears in the scene). However, to avoid repetitive code, implement a new method to carry out the process in the program.

```
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,
        size, Vector3.zero, Handles.SphereHandleCap);
58  }
59
```

The « **SetMovePoint** » method (code line 54) takes an initial position as an argument, and it returns to a new position based on the user's interaction with the colored point displayed in the Scene view. If you look at the code lines 47, 49 and 51, you will notice that three new vectors, « **p0** », « **p1** » and « **c** », have been declared and initialized using the method mentioned earlier.

If you return to Unity and try to click and draw the points, you will notice that they will not move. This mainly happens because you still have to return the return values of their global counterparts. Now, for 'optimization', only carry out the process when there is a difference between the global vectors and those declared within the « **SceneGUI** » method. It is essential to use a conditional statement to determine if there is a difference between them and assign the new values only when necessary.

```
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
```

Looking at code line 53, you will see that the vectors « **m_p0** », « **m_p1** », and « **m_c** » are updated when their values differ from their respective vectors. In the same process, the « **Repaint** » method is invoked (line 59), which updates the values of each vector in the Dot Product window after interacting with them. For « **SerializedProperty** » type properties, which are representations of serialized fields or properties in the Inspector window, you must use the « **FindProperty** » function to display the current value of each vector « **p0** », « **p1** », and « **c** » in the Dor Product window. This takes a « **string** » path as an argument and returns a « **SerializedProperty** » type object.

```
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      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
```

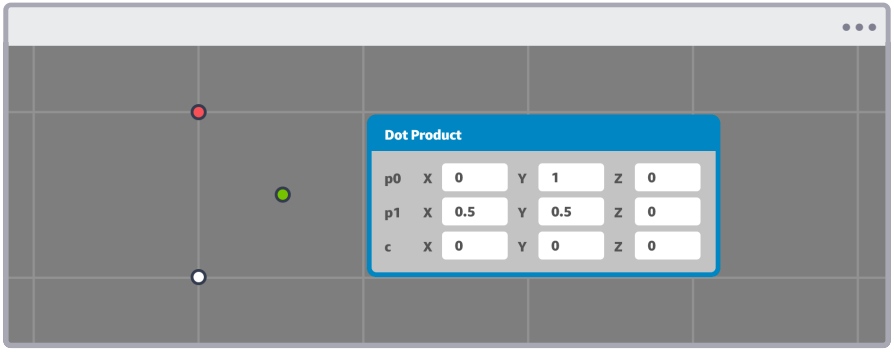
The last thing needed is to display these properties in the Dot Product window. To do so, go to the « **`OnGUI`** » method and consider at least these three factors:

- Call the « **`SerializedObject.Update`** » method to update the representation of serialized objects.
- Create a field in the Dot Product window for each property. Do this by using the « **`EditorGUILayout.PropertyField`** » function.
- Update the Scene view when there are value changes in the properties being displayed. Use the « **`ApplyModifiedProperties`** » function for this.

```
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
```

Note that between code lines 58 and 64, a new method called « **`DrawBlockGUI`** » has been added, which has two arguments: a « **`string`** » type text and a « **`SerializedProperty`** » type property. Its function implements a structure that helps organize the properties being displayed in the Dot Product window. This function is purely for 'optimization' once again, as otherwise, there would be repetitive code using the same internal structure for each property in the « **`OnGUI`** » method.

Going back to Unity, select your tool and modify the values of each vector directly from its respective property.



(1.2.d)

The vectors can be modified by interacting with them through the Handler or the Dot Product window.

Up to this point, the focus has been on creating graphics for your vectors. However, the purpose of all the aforementioned is to improve your understanding of the behavior of the Dot Product of two vectors. Therefore, continue implementing the method mentioned in Figure 1.1.k from the previous section.

```
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
```

Considering that the return value of the « **DotProduct** » method should be displayed somewhere in your tool, it would be ideal to show a text in the Scene window that reflects the real-time result of the Dot Product between « A » and « B ». To do this, add a new global variable of type « **GUIStyle** » and modify its font size, font type, and color in the « **OnEnable** » method to give the text some emphasis.

```
15    private GUIStyle guiStyle = new GUIStyle();
16
17    [MenuItem("Tools/Dot Product")]
18 >  public static void ShowWindow() …
23
24    private void OnEnable()
25    {
26        if (m_p0 == Vector3.zero && m_p1 == Vector3.zero)
27        {
28            m_p0 = new Vector3(0.0f, 1.0f, 0.0f);
29            m_p1 = new Vector3(0.5f, 0.5f, 0.0f);
30            m_c = Vector3.zero;
31        }
32
33        obj = new SerializedObject(this);
34        propP0 = obj.FindProperty("m_p0");
35        propP1 = obj.FindProperty("m_p1");
36        propC = obj.FindProperty("m_c");
37
38        guiStyle.fontSize = 25;
39        guiStyle.fontStyle = FontStyle.Bold;
40        guiStyle.normal.textColor = Color.white;
41
42        SceneView.duringSceneGui += SceneGUI;
43    }
44
```

From the previous example, in line 15, a new variable of type « **`GUIStyle`** » called « **`guiStyle`** » has been declared in the code. It serves the purpose of 'styling' the text you will display in the Scene view. Then, the values mentioned earlier in lines 38, 39, and 40 have been modified.

Continue by declaring and implementing a new method called « **`DrawLabel`** ». As the name suggests, it will be responsible for drawing both the resulting value of the Dot Product and the lines that connect each vector in the Scene view.

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

It can be easily deduced that the arguments of the « **`DrawLabel`** » method (line 105) correspond to the points or vectors that have been previously declared in the « **`SceneGUI`** » method.

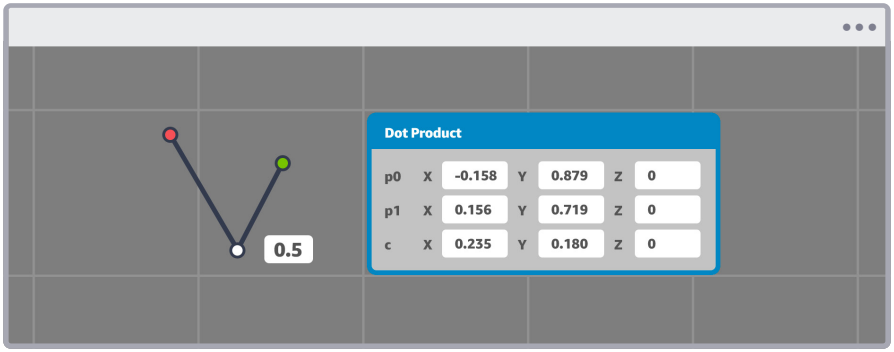
In line 107, you can find the « **`Handles.Label`** » function, which creates a text considering:

- A spatial position.
- A name for the text.
- A style.

Note that the « **Dot Product** » method has been used to define the name of the text; it returns a floating-point number, meaning it includes decimals. To accommodate this, the « **ToString** » function has been used to change its format, and « **F1** » is used to display only one decimal. Additionally, the « **Handles.DrawAAPolyLine** » function generates stylized graphical lines (anti-aliasing) at the points defined in its arguments.

Having completed the process, the « **DrawLabel** » should be included at the end of « **SceneGUI** » and pass the vectors « **p0** », « **p1** », and « **c** » as arguments. On returning to Unity, you can observe how the value of the 'Label' changes based on the position of « **m_p0** » and « **m_p1** » relative to « **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) …
89
90       DrawLabel(p0, p1, c);
91   }
92
```

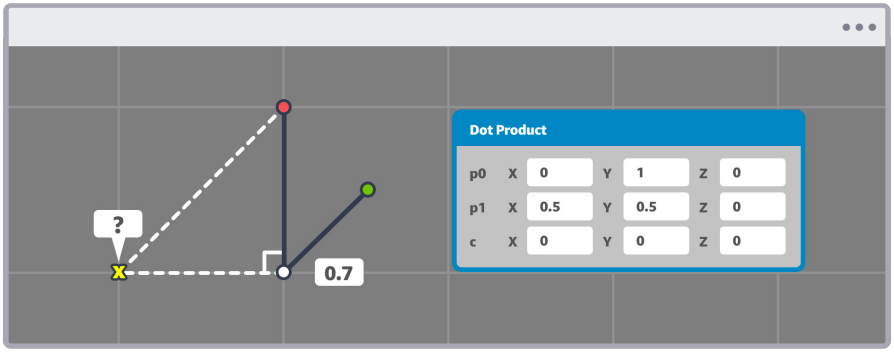


(1.2.e)

As mentioned earlier, the normalized Dot Product returns a real number ranging from -1f to 1f in respect to a reference point. If you return to Unity and modify the position of the vectors in the Scene view, you will notice that:

- Regardless of the position of « **`c`** », the return value will be 1f if « **`p0`** » and « **`p1`** » are in the same direction.
- Similarly, the return value will be -1f if « **`p0`** » and « **`p1`** » are in the opposite direction.
- The return value will be 'zero' if « **`p0`** » and « **`p1`** » are perpendicular.

Your tool is now ready to work; however, you will now add a surface, a line parallel to vector « **`c`** » as a 'characteristic' that will help visualize the difference in position between « **`p0`** » and « **`p1`** ». To carry out this process, once again use the « **`Handles.DrawAAPolyLine`** » function, which, as you already know, requires two vectors to generate graphics.



(1.2.f)

As you can see in the Figure above, a new vector has been referenced to the left side of vector « **c** » (white point), creating a right-angled triangle. To determine a new position, you must consider:

- The direction between « **p0** » and « **c** ».
- The arctangent of the opposite side divided by the adjacent side.
- The rotation of the resulting angle.

In order to get the expected result, apply the following equation:

$$R = C + HP$$

(1.2.g)

From Figure 1.2.g, « C » refers to the vector defined as the central point, « H » corresponding to the Quaternion rotation of the angle resulting from calculating the arctangent of the direction of the vector « **p0 - c** ». Finally, « P » is a vector with a magnitude equal to the distance between « C » and the new vector « R ».

Considering that a new vector would generate a right-angled triangle if it is to the right or left of « C », you can use the following trigonometric relationship to determine its angle:

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

(1.2.h)

To achieve this, return to your code and add a new method called « **WorldRotation** »,

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

At the moment, the method does not perform any operations. However, if you pay attention to its arguments, you will notice that it has three vectors, meaning three input positions.

Since the arctangent requires the opposite and adjacent side, you first need to calculate the direction between « **p** » and « **c** ».

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

Next, calculate the arctangent as shown in Figure 1.2.h. To do this, use the static method « **Atan2** », which returns the angle in radians. It is worth noting that the « **Mathf.Rad2Deg** » function will be needed to convert the result to degrees.

```
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
```

Finally, convert the angle from degrees to radians. In order to do this, it is necessary to understand the nature of the « **Quaternions** ».

The Quaternions were discovered by William Rowan Hamilton in 1843. They are used to represent rotations, and in fact, the « **Rotation** » property in Unity (found in the Transform component) corresponds to this data type.

Continuing with the operation, simply transform the angle obtained previously into a Quaternion. For this, use the « **AngleAxis** » function, which generates a rotation based on an axis. Considering that your 3D tool is designed within a two-dimensional plane, you need to use the « Z » axis for its rotation.

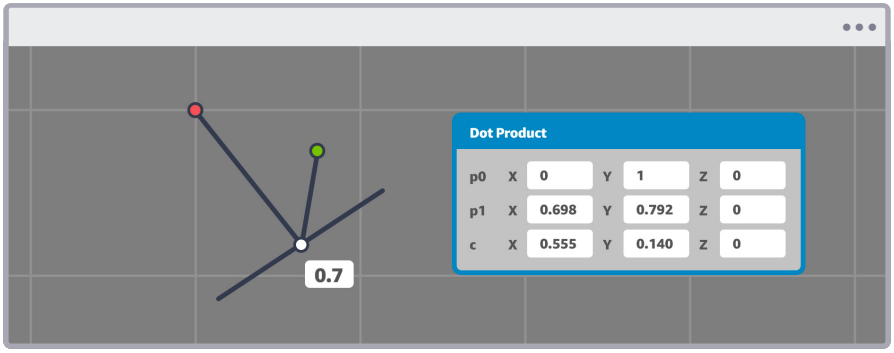
```
121  Vector3 WorldRotation (Vector3 p, Vector3  c, Vector3 pos)
122  {
123      Vector2 dir = (p - c).normalized;
124      float ang = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
125      Quaternion rot = Quaternion.AngleAxis(ang, Vector3.forward);
126
127      return c + rot * pos;
128  }
```

As can be seen in line 127 of the code, you have applied the equation detailed in Figure 1.2.g. The only thing left to do is generate two new vectors, one to the right and the other to the left of « **c** » and add them to the « **DrawLabel** » method to be visually appreciated in the Scene view.

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

From the previous example, paying attention to lines 112 and 113, you can observe that two vectors, « **cLef** » and « **cRig** » have been declared and initialized. These vectors create a displacement of one longitudinal unit in the « **Y** » coordinate.

To conclude the exercise, these vectors have been used by the « **`Handles.DrawAAPolyLine`** » function in lines 117 and 118, generating two graphic lines that simulate a 'surface.'



(1.2.i)

Returning to Unity, you can observe that « **`cLef`** » and « **`cRig`** » maintain a 90° position relative to the line formed by « **`p0`** » and « **`c`** ».

Summary.

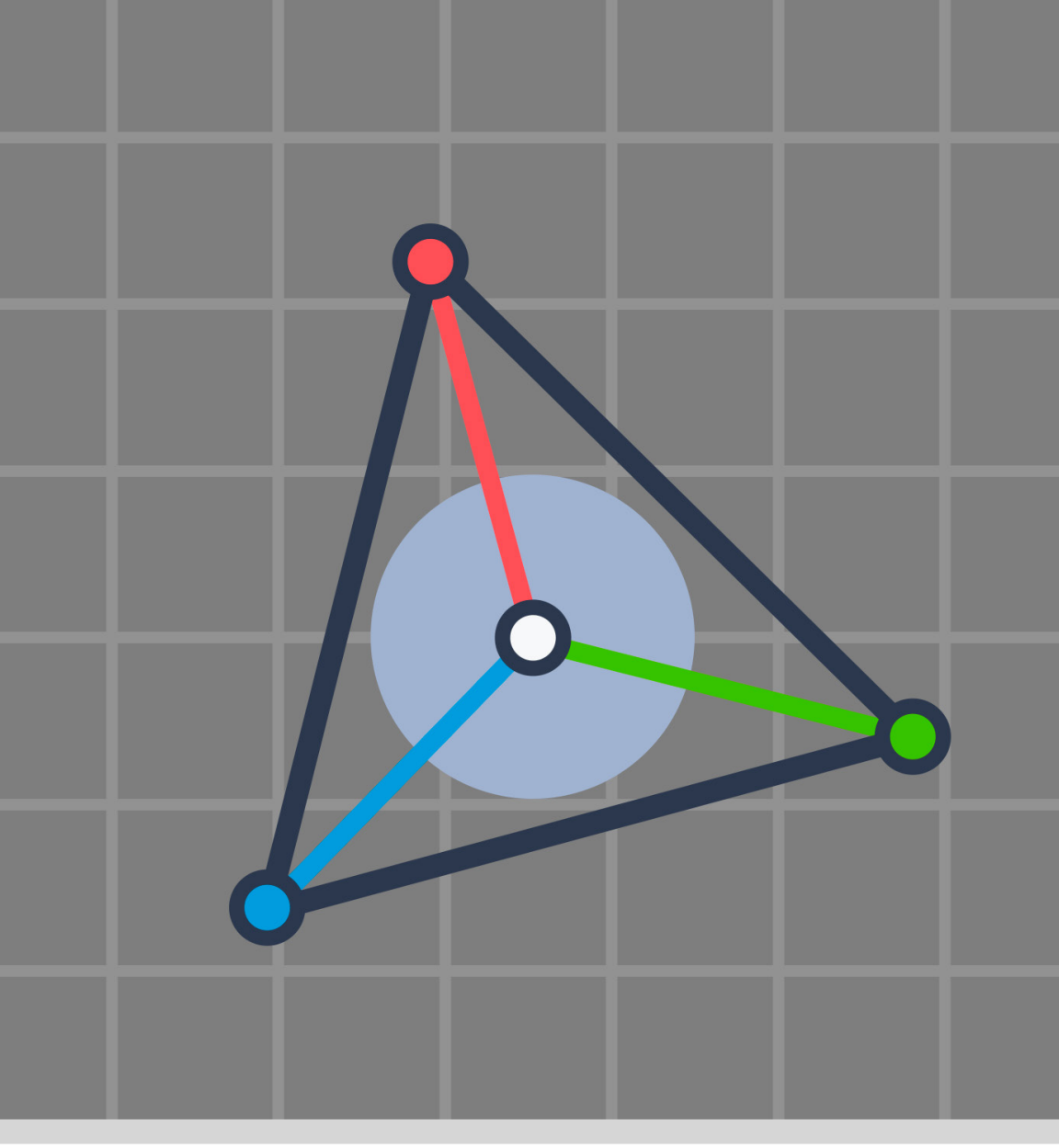
In this chapter, you have learned about the nature of the Dot Product and its various applications, covering everything from mathematical symbols to developing simple exercises.

Additionally, you have understood the difference between a point and a direction in space while detailing the operation of the « **`Summation`** » method in the first section of the first chapter.

Subsequently, other relevant topics were addressed, such as the implementation of the « **`DotProduct`** » method and the normalization of vectors to obtain values within a range between -1f to 1f.

In the second section, you will dive fully into C# to address classes like « **`EditorWindow`** », which allow you to create custom windows in the Unity editor.

Finally, you will develop a graphical tool based on the knowledge acquired in the previous section, allowing you to visualize the behavior of the Dot Product between two vectors.



Chapter 2.

Cross Product.

2.1. Introduction to the function.

Your adventure will continue by talking about a fundamental concept in programming: the Cross Product, also known as the Vector Product. Unlike the Dot Product, this operation yields a three-dimensional vector and is widely used in various applications.

Pay attention to the following formula to understand its definition:

$$\boldsymbol{P} \times \boldsymbol{Q} = (\boldsymbol{P}_y\boldsymbol{Q}_z - \boldsymbol{P}_z\boldsymbol{Q}_y, \boldsymbol{P}_z\boldsymbol{Q}_x - \boldsymbol{P}_x\boldsymbol{Q}_z, \boldsymbol{P}_x\boldsymbol{Q}_y - \boldsymbol{P}_y\boldsymbol{Q}_x)$$

(2.1.a)

It is important to note that it is pretty common to confuse the symbol « $\times$ » with the multiplication symbol when getting familiar with vector mathematics. However, it is crucial to remember that this symbol refers explicitly to the Cross Product and not the multiplication operation.

Since the Cross Product returns a three-dimensional vector, you can see that the operations within the parentheses in Figure 2.1.a correspond to the components of the new vector. In other words,

$$\begin{aligned}(\boldsymbol{P} \times \boldsymbol{Q})_x &= \left(\boldsymbol{P}_y\boldsymbol{Q}_z - \boldsymbol{P}_z\boldsymbol{Q}_y\right) \\ (\boldsymbol{P} \times \boldsymbol{Q})_y &= (\boldsymbol{P}_z\boldsymbol{Q}_x - \boldsymbol{P}_x\boldsymbol{Q}_z) \\ (\boldsymbol{P} \times \boldsymbol{Q})_z &= (\boldsymbol{P}_x\boldsymbol{Q}_y - \boldsymbol{P}_y\boldsymbol{Q}_x)\end{aligned}$$

(2.1.b)

To understand the concept better, proceed with the following exercise: Define two vectors, « $\boldsymbol{A}$ » and « $\boldsymbol{B}$ », with the following components:

$$\boldsymbol{A} = (\mathbf{0}, \mathbf{1}, \mathbf{0})$$
$$\boldsymbol{B} = (\mathbf{1}, \mathbf{0}, \mathbf{0})$$

(2.1.c)

Using the formula presented in Figure 2.1.a, define a third vector, « $\boldsymbol{C}$ », which will be the result of the Cross Product between the aforementioned vectors.

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

(2.1.d)

If you substitute the given components from Figure 2.1.c, you get:

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

(2.1.e)

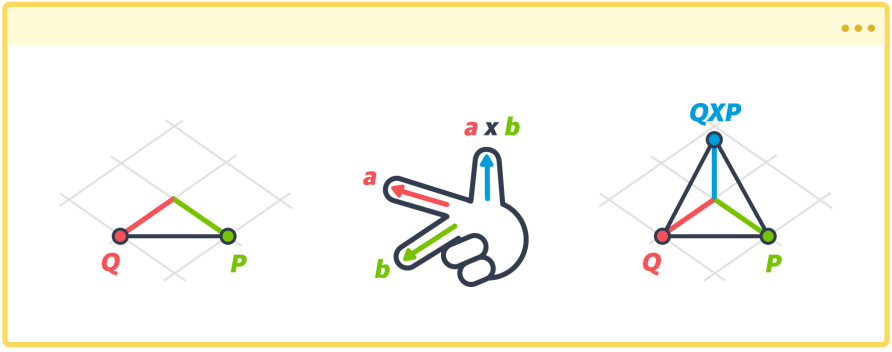
Therefore,

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

(2.1.f)

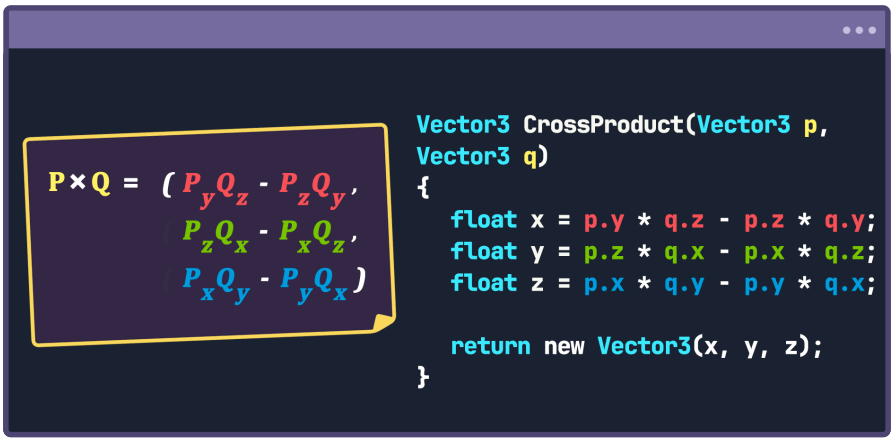
The resulting vector « C » has the particularity that it is perpendicular to the original vectors. Furthermore, its magnitude is related to the area of the parallelogram generated by the two vectors « A » and « B ». Its direction can be determined using the 'right-hand rule,' which states that by extending your right hand with your fingers in a specific position, your thumb will point in the direction or axis of the resulting vector's rotation.

What can you use this operation for? For various applications, such as calculating the normal vector of a vertex or surface or even determining whether a polygon is concave or convex based on the direction of the resulting vector, this is beneficial in triangulations and other geometry problems.



(2.1.g)

Note that the Cross Product can be expressed in different ways, either as a vector quantity, as seen earlier in Figure 2.1.a, or through a linear transformation. For instance, you could declare three scalar values to obtain the equations shown in Figure 2.1.b.



(2.1.h)

Looking at Figure 2.1.h, notice that within the « **CrossProduct** » method, three scalar variables « **x** », « **y** », and « **z** » have been declared, each containing a part of the equation. How can you express the same operation as a linear transformation? Focus on the following formula,

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

(2.1.i)

The definition presented in the previous Figure may seem complex, but its implementation in the C# programming language is relatively straightforward. We just need to remember that the Cross Product is expressed using a three-dimensional matrix of « Q » operating on « P ».

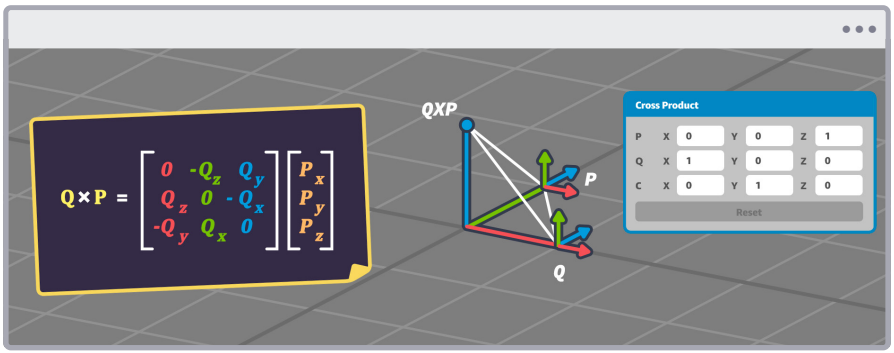
$$Q \times P = \begin{bmatrix} 0 & -Q_z & Q_y \\ Q_z & 0 & -Q_x \\ -Q_y & Q_x & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

```
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)

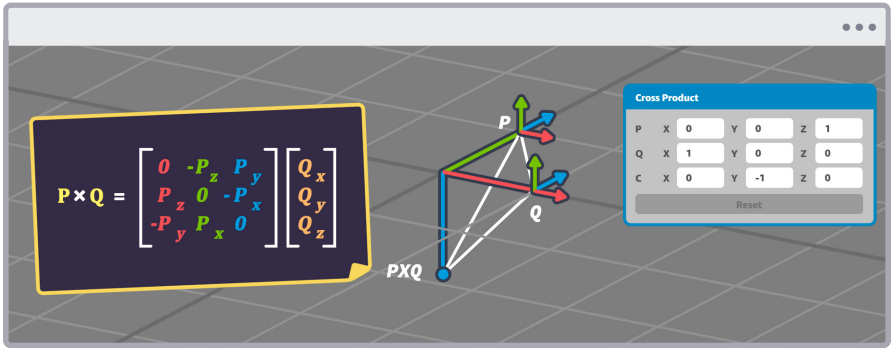
It must be pointed out that in the « **`CrossProduct`** » method presented in the previous Figure, a new four-by-four matrix called « **`m`** » has been defined. This matrix stores each dimension of « Q » in the same order as in the equation of Figure 2.1.j.

One factor to consider is the direction of the Cross Product, which is determined in relation to the orientation of its vectors. For example, the linear transformation presented in Figure 2.1.j returns the following vector,



(2.1.k)

However, if the matrix values are flipped, the Cross Product will be negated, pointing in the opposite direction.



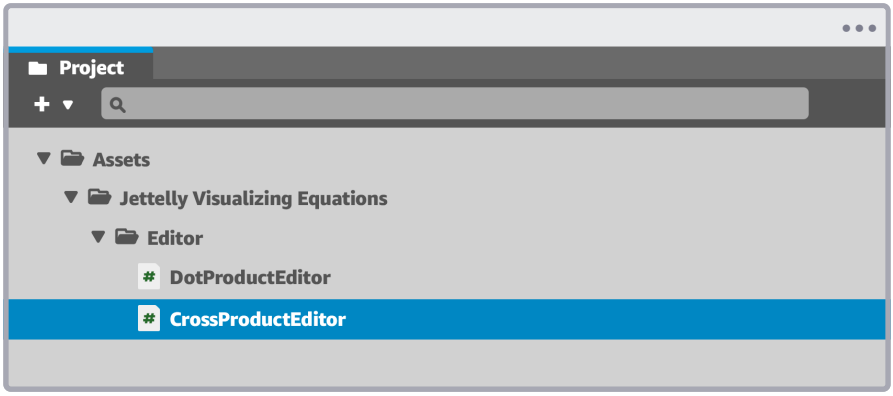
(2.1.l)

2.2. Developing a Unity tool.

Continuing with the purpose of this book, you will create a tool in Unity to help you understand the nature of the Cross Product better. To achieve this, you need to repeat part of the process detailed in the previous chapter, using the « **`UnityEditor`** » dependency and extending your script from « **`EditorWindow`** ». Why are you doing this? Mainly because,

- It will be a visual tool.
- You will only need one instance of the tool.

To get started, create a new script in your project called « **`CrossProductEditor`** » to get started. It is important to remember that for it to work correctly, you must store your controller within an Editor-type folder. For ease-of-use, it is suggested that you use the same folder created in the previous chapter.

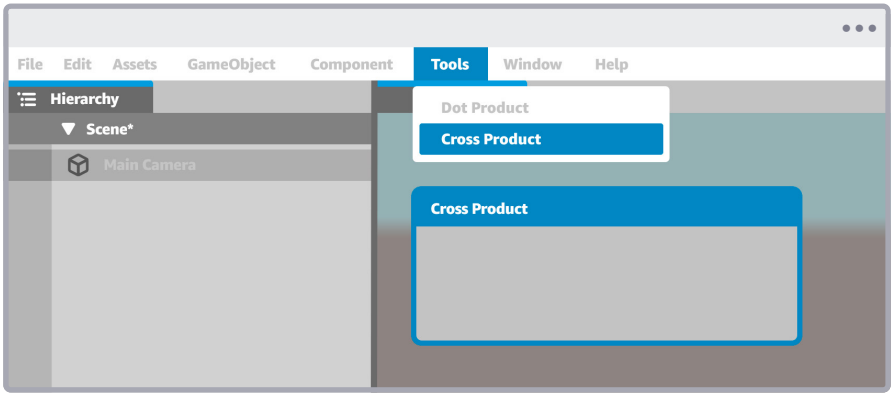


(2.2.a)

In your script, the first action is to add a static method, which will be used to display a popup window in Unity.

```
using UnityEngine;
using UnityEditor;

public class CrossProductEditor : EditorWindow
{
    [MenuItem("Tools/Cross Product")]
    public static void ShowWindow()
    {
        GetWindow(typeof(CrossProductEditor), true,
        "Cross Product");
    }
}

```

As seen in the previous example, you have once again used the « **`GetWindow`** » function, which returns one instance of the tool. For this case, it is called 'Cross Product' for illustration purposes. Upon returning to Unity, observe that a new window has been added to the 'Tools' menu. However, this window does not currently execute any function.



(2.2.b)

As you already know, the Cross Product yields a new three-dimensional vector based on the position of two input vectors, which are also three-dimensional. Therefore, proceed to declare the necessary variables for the exercise as follows:

```
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
```

You can observe in code lines 6, 7, and 8 that three three-dimensional vectors have been defined: « **m_p** », « **m_q** », and « **m_pxq** ». Additionally, three properties of type « **SerializedProperty** » have been declared, one for each vector. It is important to note that it is necessary to initialize the values of the vectors and properties and update the Scene view based on the new values of each vector. Include the following methods in your script to carry out these tasks.

- « **OnEnable** ».
- « **OnDisable** ».
- « **OnGUI** ».

As you saw in the previous chapter, these functions are included within the « **MonoBehaviour** » class; therefore, they will be easily identified by IntelliSense depending on the code editor you are using.

```
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
```

At least two methods that have been previously employed will be used during the development of the 'Dot Product' window tool. These methods correspond to:

- « **SceneGUI** », which is used to Update/display tool changes in the Scene view.
- And the « **DrawBlockGUI** » method, which is responsible for drawing values in the GUI.

This approach creates a problem: repeated code! Since both the scripts « **CrossProductEditor** » and « **DotProductEditor** » are going to use the same methods, it would be ideal to implement them in separate classes for more efficient access. To achieve this, go to your Unity project and create two new scripts. Call the first one « **CommonEditor** » and place it inside the Editor folder. Once opened, make sure to include « **UnityEditor** » and extend it from the « **EditorWindow** » class. Why? Because your current tool will also extend from the latter, and you will require access to the built-in functions for its development.

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

Line 6 of the code shows that the « **`DrawBlockGUI`** » method has been declared and used in developing the Dot Product tool. It has also been marked as « **`virtual`** », which implies that it can be overwritten if required.

The second script will be an interface called « **`IUpdateSceneGUI`** », which will be responsible for implementing the « **`SceneGUI`** » method. Why an interface? Because of its abstract nature, this method does not require a specific algorithm within its scope. However, it will be necessary for developing several « **`EditorWindow`** » type tools.

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

Next, simply extend your current tool from « **`CommonEditor`** » and implement the « **`IUpdateSceneGUI`** » interface, as shown in the following example:

```
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() …
20
21        private void OnEnable()
22        {
23            SceneView.duringSceneGui += SceneGUI;
24        }
25
26        private void OnDisable()
27        {
28            SceneView.duringSceneGui -= SceneGUI;
29        }
30
31 >      private void OnGUI() …
35
36        public void SceneGUI(SceneView view)
37        {
38            throw new System.NotImplementedException();
39        }
40    }
41
```

Observing line 38 of the code, you will notice that C# has included the « **NotImplementedException** » class from « **System** » to indicate that the « **SceneGUI** » method has not been developed. To avoid conflicts in creating the tool, you need to remove or comment on this line of code; otherwise, Unity may throw a compilation exception.

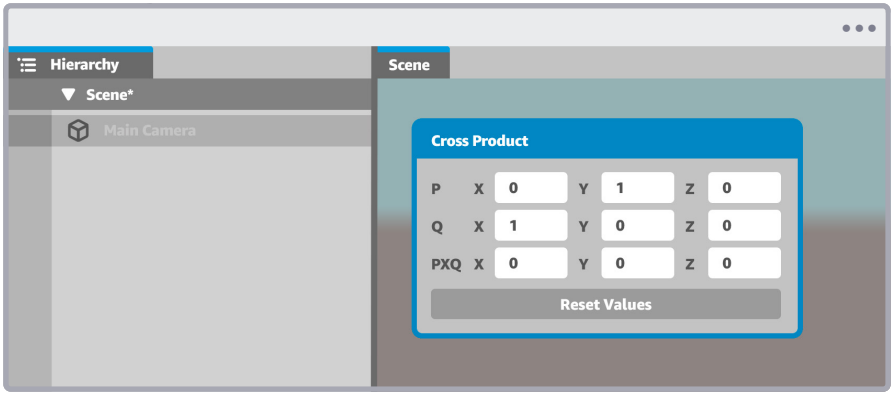
Then, initialize the serialized properties in the « **OnEnable** » method, and declare a new method called « **SetDefaultValues** », which will be used to assign default values to the « **m_p** » and « **m_q** » vectors.

```
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
```

Unlike the tool developed in the previous chapter, you will include a button in the Cross Product window. This button will be used to reset the vectors and their values to their default state. This process is aimed solely at enhancing your understanding of the Cross Product operation.

```
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
```

From the previous example, it is important to remember that the « **DrawBlockGUI** » method (code lines 51 to 53) is currently implemented in the « **CommonEditor** » script. Therefore, it can be inherited. As shown in code lines 60 to 63, there is an « **if** » statement that calls the « **GUILayout.Button** » function. This function returns true or false depending on whether the button has been pressed in the 'Cross Product' window.



(2.2.c)

Up to now, you have configured the Cross Product window and its main functionality. Next, you will proceed to draw the tool in the Scene window. However, before continuing with the process, it is necessary to declare a global variable of type « **`GUIStyle`** », which is used to style the text in the tool.

```
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
```

In line 15, a new variable called « **guiStyle** » has been declared. As in the previous chapter, its size, style, and colour have been initialized in the « **OnEnable** » method, specifically in lines 41 to 43.

Before proceeding with the implementation of the « **SceneGUI** » method, you need to add three new methods: « **DrawLineGUI** », « **RepaintOnGUI** », and « **CrossProduct** ». The latter corresponds to the method defined in the previous section, Figure 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
```

Add text labels to the vectors in the Scene window in order to make it easy to identify them. If you analyze the internal structure of the « **DrawLineGUI** » method, we can see that it mainly focuses on two actions:

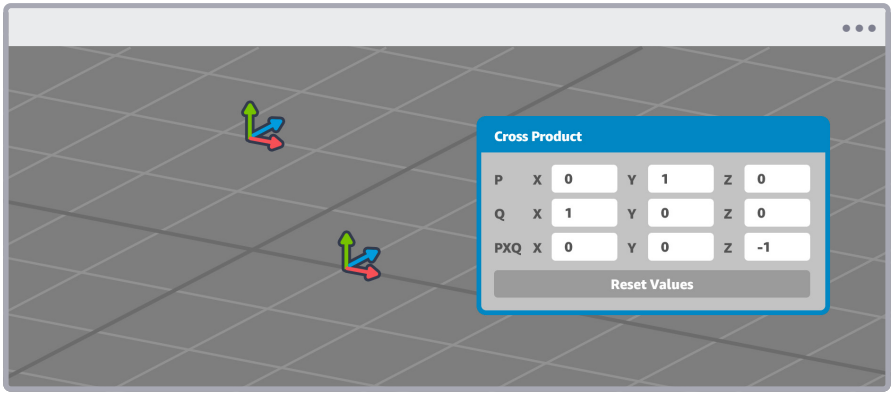
1. To draw text using the « **Handles.Label** » function.
2. To draw a line from the current position of the vector to the world origin (point zero) using the « **Handles.DrawAAPolyLine** » function.

On the other hand, the « **RepaintOnGUI** » method only contains the « **Repaint** » function of « **EditorWindow** » in its scope. This is because you will use « **RepaintOnGUI** » in conjunction with the « **UnityEditor.Undo** » function, which allows you to register 'undo' operations (CTRL + Z) in your code.

In this section, continue with the « **SceneGUI** » method, and declare two new three-dimensional vectors, « **p** » and « **q** », to be used to determine a new position for the « **m_p** » and « **m _q** » vectors later on.

```
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
```

Returning to Unity, you will see that the vectors have appeared in the Scene window. This is mainly due to the call of the function « **Handles.PositionHandle** », which returns a new position based on the user's interaction with the handler. However, it is essential to note that at this stage, the position of the 'Handles' can only be modified through the Cross Product window. Why? Because you have not yet assigned values to « **p** » and « **q** » for their global counterparts.



(2.2.d)

Next, carry out the following actions in the « **SceneGUI** » method:

- Declare and initialize a new three-dimensional vector using the « **CrossProduct** » method and represent it in the Scene window as a solid blue disc.
- Verify if there have been any changes in the current vector position by using an « **if** » condition that evaluates the user's interaction with the previously implemented « **Handles** ».
- In case of changes, update the values of the vectors « **m_p** », « **m_q** », and « **m_pxq** » with the new corresponding values.

```
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
```

In the example above, it is crucial to highlight line 78, where the « **pxq** » vector is initialized with the result of the « **CrossProduct** » method, which takes « **p** » and « **q** » as arguments. Additionally, thanks to the « **Undo.RecordObject** » function (line 83), all the changes made to the tool will be saved. If you move the handles and press CTRL + Z, the tool will return to its previous state in the Scene window.

However, after saving the changes and returning to Unity, you will notice two issues:

1. The Cross Product « pxq » appears in the Scene window, but it is difficult to understand since there is not a graphic line connecting it to the vectors « **p** » and « **q** ».
2. If you press CTRL + Z after changing the position of any of the handles, the Cross Product window does not update correctly.

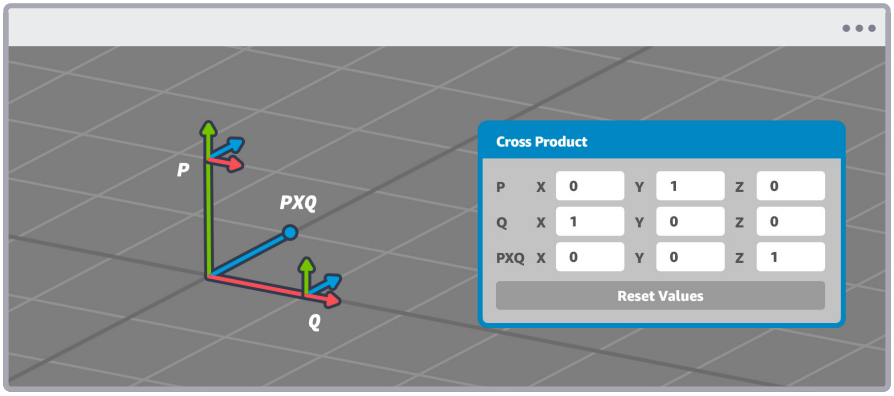
To address the first problem, you need to include the « **DrawLineGUI** » method within the field as follows:

```
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
```

The code has been updated in lines 92 to 94. Text and a line have been added to each vector, which is reflected in the Unity Scene window. However, there is still an issue when undoing changes by pressing CTRL + Z. Fortunately, you can fix this by calling the « **RepaintOnGUI** » method every time an 'Undo/Redo' event is triggered.

```
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        Undo.undoRedoPerformed += RepaintOnGUI;
47    }
48
49    private void OnDisable()
50    {
51        SceneView.duringSceneGui -= SceneGUI;
52        Undo.undoRedoPerformed -= RepaintOnGUI;
53    }
54
```

As you can see, the « **`Undo.undoRedoPerformed`** » function has been used in lines 46 and 52, and it is triggered after undoing a change in the Scene window. If everything has been done correctly, you should be able to observe the behavior of the Cross Product between vectors « **p** » and « **q** » in Unity.



(2.2.e)

As mentioned in Figure 2.1.j from the previous section, you can implement the Cross Product in the code through a linear transformation. To understand the process:

- We will discuss the currently implemented « **`CrossProduct`** » method in the script.
- Define a new method based on Figure 2.1.j.

```
/*
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);
}
*/

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;
}

```

If you carefully analyze the internal structure of the new definition of the « **CrossProduct** » method. You can observe that a four-by-four matrix has been created and initialized, with each value assigned to its respective corresponding column/row. As a result, we obtain the same value as in the previous versions of the method.

Summary.

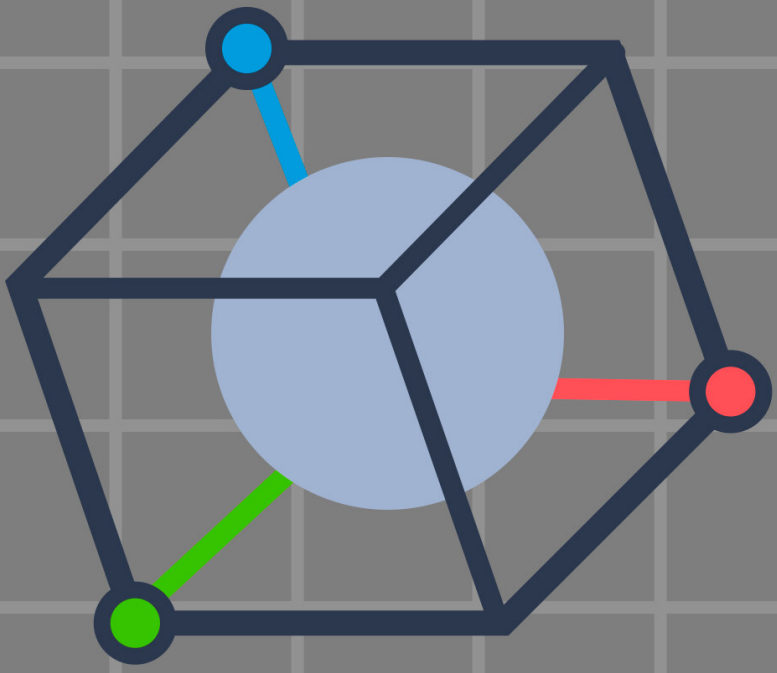
This chapter began by defining the Cross Product between two vectors using the right-hand rule, a traditional technique that helps understand both its direction and magnitude in three-dimensional space.

A different perspective was also explored by introducing the Cross Product as an application of a matrix; a linear transformation that relates two vectors in a three-dimensional vector space.

Throughout the chapter, the fundamental geometry properties of the function were explained, including its dependence on the orientation of the original vectors and its relationship with the area of a parallelogram.

For a more practical understanding, we presented a simple example illustrating how to calculate the Cross Product in practice. We demonstrated how to obtain both the resulting vector and its magnitude.

Finally, the chapter concluded by exploring the implementation of the Cross Product in C# language, specifically within the context of Unity. To do this, a tool within the editor was developed, facilitating the three-dimensional visualization of the function, and providing a broader understanding of its practical application and its utility in the creation of virtual environments.



Chapter 3.

Quaternions.

3.1. Introduction to the function.

Quaternions are fundamental objects in any development software focused on 3D objects. Why is that? They allow you to generate vertex rotations, avoiding the phenomenon caused by gimbal lock (also known as the gimbal effect), which results in the loss of a degree of freedom and causes unexpected behavior in the rotation system.

To understand their definition, look carefully at the following formula:

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

(3.1.a)

It is common to feel a little fear when attempting to interpret this type of equation for the first time. This misgiving is precisely due to the limited understanding of the properties of these mathematical objects.

Since this type of mathematical object extends from the concept of complex numbers, you can deduce that the variables « i », « j » and « k » in Figure 3.1.a are related to the canonical base vectors of three-dimensional Euclidean space. The notation for these 'entities' has been deliberately chosen to establish a direct relationship with their corresponding axes:

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

(3.1.b)

The variables « w », « x », « y » and « z » correspond to real numbers, representing a point in space with components given by the triplet « x », « y », « z » and associated scalar « w ». Therefore, on the other hand, there are coordinates representing the basis of three-dimensional space, while on the

other, there are real components corresponding to the coordinates of a vertex within that space. The objects « $\boldsymbol{i}$ », « $\boldsymbol{j}$ » and « $\boldsymbol{k}$ » are different from the base vectors as they are imaginary numbers that obey the multiplication rule given by,

$$i^2 = j^2 = k^2 = ijk = -1$$

(3.1.c)

Why are they imaginary? Because they have the particularity that their square equals -1, meaning that they exist in a mathematical world distinct from real numbers.

Now, how can you rotate an object with multiple vertices considering the previous explanation? To do so, first consider at least three factors:

1. The Quaternion product.
2. Its conjugate.
3. The rotation angle.

Unlike the definition mentioned earlier in Figure 3.1.a, a Quaternion is often represented in the scalar-vector form.

$$q = s + v$$

(3.1.d)

Where, « $\boldsymbol{s}$ » represents the scalar value of the component « $\boldsymbol{w}$ », and « $\boldsymbol{v}$ » refers to the vector with components « $\boldsymbol{x}$ », « $\boldsymbol{y}$ » and « $\boldsymbol{z}$ » —in other words, the imaginary part of the Quaternion « $\boldsymbol{q}$ » in equation 3.1.a.

$$s = w$$
$$v = (x, y, z) = xi + yj + zk$$

(3.1.e)

This definition is more concise and helps simplify certain operations, such as Quaternion multiplication. To understand the concept better, let us perform the following exercise: Define two quaternions, « q_1 » and « q_2 » as follows.

$$q_1 = s_1 + v_1$$
$$q_2 = s_2 + v_2$$

(3.1.f)

Then multiply them,

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

(3.1.g)

The equation presented in Figure 3.1.g may seem complex at first glance; however, two important points must be considered immediately. First, quaternion multiplication is not commutative, meaning that « q_1 » multiplied by « q_2 » is not the same as « q_2 » multiplied by « q_1 ». Understanding this helps us avoid making mistakes in the future. Second, upon examining the structure of the resulting Quaternion from the multiplication, you can notice that the scalar and vector parts correspond to operations that have already been performed with vectors, enabling a straightforward implementation in C#.

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

```
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)

As seen in the « **ScalarVector** » method in Figure 3.1.h, two float variables, « **s1** » and « **s2** » have been declared to store the value of the « **w** » component for each Quaternion. Subsequently, two three-dimensional vectors have been declared to store the values of « **x** », « **y** » and « **z** » following the same logic presented in Figure 3.1.g.

Finally, a scalar variable named « **s** » and vector « **v** » have been declared, following the definition presented in Figure 3.1.d. They receive the result of the operation carried out in the Quaternion multiplication.

It is worth noting that the « **Q** » value type presented in Figure 3.1.h exemplifies a « **struct** » type structure, encapsulating each component « **x** », « **y** » « **z** » and « **w** » of a Quaternion.

Considering the imaginary numbers in the vector part of a Quaternion, you can perform its conjugation, which implies changing the sign of all imaginary terms. As an example, conjugate the equation presented in Figure 3.1.d as follows:

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

(3.1.i)

Which is the same as,

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

(3.1.j)

From a geometric perspective, conjugation can be understood as an operation that preserves the scalar (or real) part of a Quaternion while generating a reflection in its vector part. It is as if a mirror has been placed perpendicular to the vector and you can see its reflection. This property turns conjugation into an operation that allows you to obtain the reflection of a particular vector naturally. The conjugation of a Quaternion also provides a simple way to obtain its absolute value, which can be expressed as follows:

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

(3.1.k)

Which is the same as saying,

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

(3.1.l)

There are several reasons why you need to conjugate a Quaternion. For instance, when performing interpolation, inverting a Quaternion, or carrying out a rotation. In this case, conjugation becomes necessary when implementing the rotation of a Quaternion representing a point in space.

Its implementation in C# looks as follows:

$$\bar{Q} = S - V$$

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

(3.1.m)

Looking at the Figure 3.1.m, notice that the « **w** » component remains positive, while each vector component becomes negative.

Given the nature of the previous explanation, you now need to represent quaternions as points in a three-dimensional space, and for that, again turn your attention to Figure 3.1.a. However, this time, completely ignore the real part of the Quaternion.

$$p = xi + yj + zk$$

(3.1.n)

As shown in Figure 3.1.n, a point in three-dimensional space can be expressed more concisely and directly by using only the imaginary components. Now, when discussing rotations in space, it is essential to define an angle and an axis. It is worth noting that the rotation angle is measured in radians, while a unit vector in space defines the rotation axis.

Pay attention to the following formula to understand the concept.

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

(3.1.o)

From the previous figure,

- The « θ » symbol refers to the rotation angle.
- The variables « u_x », « u_y » and « u_z » are the components of the unit vector that defines the rotation axis.
- Finally, « i », « j » and « k » are the standard imaginary numbers.

Its implementation in C# can be represented as follows,

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

```
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)

Once you have the rotation Quaternion, you can apply it to a point in space using the following transformation rule:

$$\boldsymbol{p' = (q * p) * \overline{q}}$$

(3.1.q)

Referring to Figure 3.1.q, it can easily be deduced that the variable « $\boldsymbol{q}$ » corresponds to the rotation Quaternion, while « $\boldsymbol{\overline{q}}$ » refers to its conjugate. On the other hand, « $\boldsymbol{p}$ » corresponds to the point you want to rotate. How can this be visualized? Perform the following exercise: Start by placing a point or vertex « $\boldsymbol{p}$ » in space.

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

(3.1.r)

Then, define an angle and a rotation axis. For this example, use a 45-degree rotation around the « $\boldsymbol{Y}$ » axis.

$$\boldsymbol{\theta = 45°}$$
$$\boldsymbol{j = (0_i, 1_j, 0_k)}$$

(3.1.s)

Next, define the rotation Quaternion following the equation from Figure 3.1.o as follows,

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

(3.1.t)

Which is the same as,

$$q = 0.923 + 0.382j$$

(3.1.u)

Therefore,

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

(3.1.v)

It is worth noting that the first operation in the above equation, « $\cos\left(\frac{45}{2}\right)$ » refers to the real part of the Quaternion, i.e., the « s » component in the form « $q = s + v$ », while the remaining terms define the vector values « v ». Later in this book, you will see its implementation in C#.

Now, you just need to rotate the initial point. To do that, apply the equation from Figure 3.1.q as follows:

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

(3.1.w)

Which is the same as saying,

$$p' = 0.707_i - 0.707_k$$

(3.1.x)

Therefore,

$$p' = (0.707_x, 0_y, -0.707_z)$$

(3.1.y)

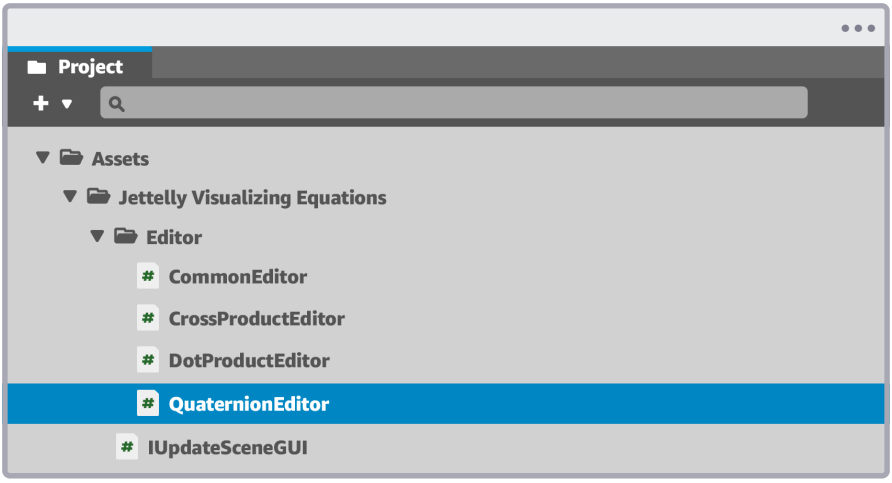
From the previous Figure, « p' » refers to a new position of a point rotated 45 degrees around the « Y » axis in three-dimensional space.

3.2. Developing a Unity tool.

Up to this point, we have reviewed some of the equations involved in the process of rotating a point in space using quaternions. Now, let us proceed to develop a tool, focusing on the different functions or operations associated with it to gain a deeper understanding of the concept mentioned earlier.

The process described in this chapter is similar to the one explained in previous chapters. Therefore, once again, you will make use of the « **`UnityEditor`** » dependency to extend the script from « **`CommonEditor`** », as the latter contains the fundamental « **`EditorWindow`** » class for creating these types of tools.

To initiate this process, create a new script in your project called « **`QuaternionEditor`** ». Remember that for it to work correctly, you must place this script in the Editor folder, which was created earlier in your Unity project.



(3.2.a)

This time, you need to not only implement the functions and methods that enable the generation of the tool, but also to create a « **`struct`** » object type to encapsulate the various properties that a Quaternion offers. To begin development, define the window that will appear in the Tools menu in Unity, which will conveniently be called 'Quaternions.'

```
using UnityEngine;
using UnityEditor;

public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
{
    [MenuItem("Tools/Quaternion")]
    public static void ShowWindow()
    {
        GetWindow(typeof(QuaternionEditor), true, "Quaternion");
    }

    public void SceneGUI(SceneView view)
    {
        // throw new System.NotImplementedException();
    }
}
```

In the previous code, looking at line number 4, you can see that the « **IUpdateSceneGUI** » interface has been implemented. This implementation is necessary because you need to use the « **SceneGUI** » method once again to project a list of points that, together, will form a three-dimensional cube.

Continuing, declare a global list of vectors to store each cube's point or vertex, and then initialize each vertex inside the « **SceneGUI** » method.

```
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() …
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
```

To work with a list of type « **List** », you need to include the « **System.Collections.Generic** » dependency, as seen in line 1 of the previous code. The list of vertices is then declared in line 7. It is essential to mention that, for this exercise, the list « **List** » is being used in the current implementation. However, for consistency, considering that no more points will be added to the ones already defined, you could easily use a constant array of three-dimensional vectors.

The eight vertices included in the cube have been initialized in code lines 17 to 19. However, to represent them visually in the Scene view, it is essential to generate gizmos for each of them. Do this by repeating each vertex within a « **for** » loop and using the « **Handles.SphereHandleCap** » function to visualize them.

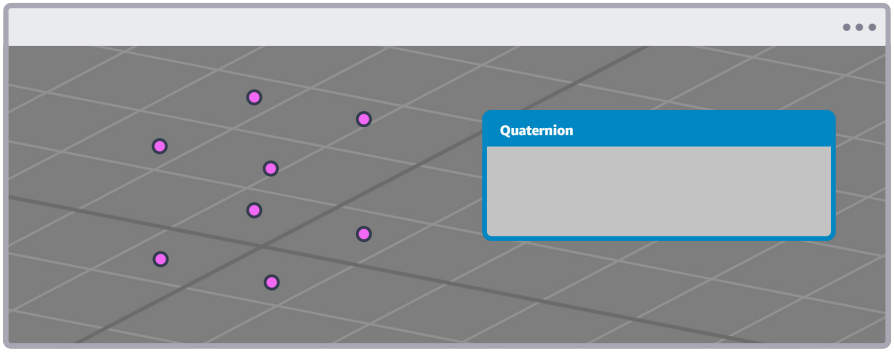
```
15   public void SceneGUI(SceneView view)
16   {
17       vertices = new List<Vector3>
18       {
19           new Vector3(-0.5f,  0.5f,  0.5f),
20           new Vector3( 0.5f,  0.5f,  0.5f),
21           new Vector3( 0.5f, -0.5f,  0.5f),
22           new Vector3(-0.5f, -0.5f,  0.5f),
23           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],
             Quaternion.identity, 0.1f, EventType.Repaint);
32       }
33   }
34
```

If you look at code lines 29 to 32, you can see that a loop has been created within the « **SceneGUI** » method, repeating each vertex defined in the list. It is important to note that the second argument of the « **SphereHandleCap** » function, called « **Quaternion.identity** », corresponds to the rotation of the vertices using quaternions. However, in this specific case, the Quaternion identity is being returned, which means there is no rotation. This is because, later in this book, you will develop your own implementation of quaternions to rotate the cube's vertices according to a defined angle.

Returning to Unity at this point, there will not be any graphical representation of the cube's points in the Scene view. This is because you are not running this process during the « **OnGUI** » method call. To fix this, simply subscribe to the « **SceneView.duringSceneGui** » function to receive a callback. This is implemented in the « **OnEnable** » and « **OnDisable** »methods.

```
5     public class QuaternionEditor : CommonEditor, IUpdateSceneGUI
6     {
7         private List<Vector3> vertices;
8
9         [MenuItem("Tools/Quaternion")]
10 >      public static void ShowWindow() …
14
15        private void OnEnable()
16        {
17            SceneView.duringSceneGui += SceneGUI;
18        }
19
20        private void OnDisable()
21        {
22            SceneView.duringSceneGui -= SceneGUI;
23        }
24
25 >      public void SceneGUI(SceneView view) …
44    }
45
```

From the preceding code, observe the implementation mentioned earlier in code lines 17 and 22. Upon returning to Unity, the new 'Quaternion' window can be found within the Tool menu, projecting the eight vertices of the cube.

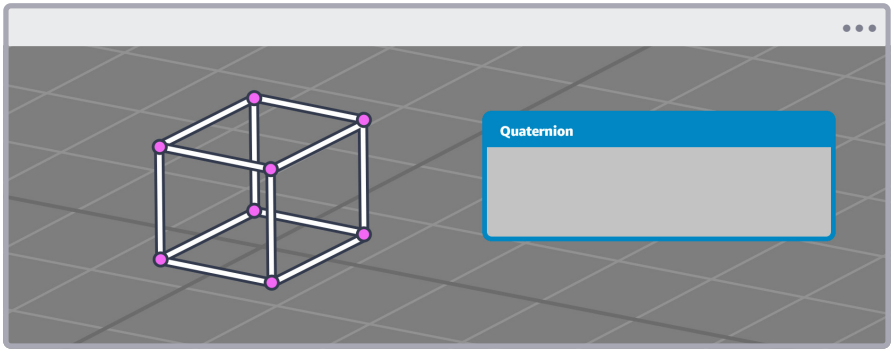


(3.2.b)

An action you can perform in your tool is to draw a line between each vertex to enhance the cube's projection in the Scene view. To do this, use the « **`Handles.DrawAAPolyLine`** » function, which enables you to draw lines between points in three-dimensional space. However, its implementation in this case presents a challenge: if you want to connect all the vertices without repetition, you must group the different point combinations into a list of integer values. One approach to this process is using a double list, where the first group would contain the point combinations, while the second group would hold the corresponding spatial points.

```
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],
            Quaternion.identity, 0.1f, EventType.Repaint);
42      }
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]],
            vertices[index[i][1]]);
63      }
64  }
```

Focusing on the « **`SceneGUI`** » method, in line number 44, a constant list of integer values has been declared and initialized. These values represent different indices that form connections between the vertices. As mentioned earlier, the goal is to generate a graphical representation by drawing lines that connect pairs of cube points, thereby enhancing its visualization. Additionally, in line 60, a « **`for`** » loop has been included that repeats through the index list and uses the « **`DrawAAPolyLine`** » function to draw the corresponding lines. Upon revisiting the Scene view, you can observe the connections between each vertex of the cube.



(3.1.c)

Up to this point, the focus has been on projecting points in space, primarily to visualize the shape that you will later rotate. Now, you will put your logical thinking into practice by incorporating all the necessary properties of a Quaternion. To do so, begin by declaring two new global variables:

- A floating-point value to represent the rotation angle.
- A three-dimensional vector to indicate the rotation axis.

```
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
```

Continuing with the practices established in previous chapters, it is necessary to declare specific « **SerializedProperty** » type properties to expose the variables in the tool's window. This data type allows you to manage « **Undo** » commands, edit multiple objects, and control prefab overrides. The declaration of these properties can be seen in lines 11 and 12.

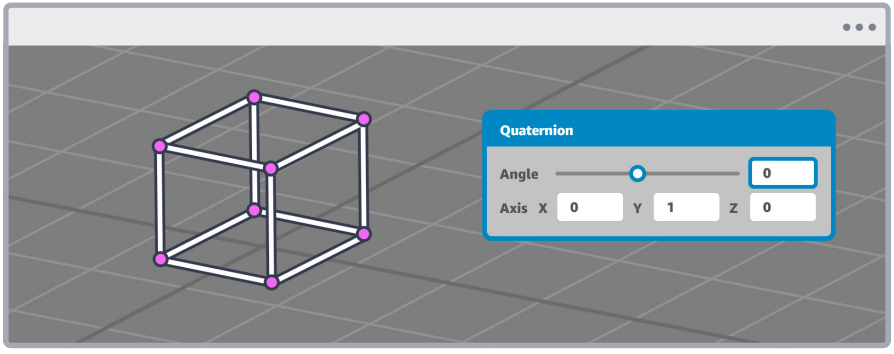
Moving forward, you need initialize the variables in the « **OnEnable** » method and expose them in the Quaternion window using the « **OnGUI** » method.

```
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   }
```

In the previous example, the « **propAngle** » and « **propAxis** » properties were initialized with the values of each variable in lines 25 and 26, respectively. Now, you just need to project these variables in your tool. Since your script is an extension of « **CommonEditor** », you need to use the « **DrawBlockGUI** » method again, which, as you already know, allows you to draw a label and property within a horizontal layout block.

```
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   }
```

If you go back to Unity, you will see that the mentioned properties have been correctly included in your tool.



(3.2.d)

At this point, you can consider that the initial process has concluded. The cube has been projected in the Scene window, and its vertices remain constant. Next, you need to implement the necessary equations to rotate the vertices, following the definition of a Quaternion. For practical reasons, within the same script being worked on, you need to declare a structure called « **HQuaternion** » to avoid conflicts with the data type « **Quaternion** » provided by Unity dependencies. It is worth noting that this process can also be carried out from a dedicated script exclusively for this task.

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

In the previous example, if you examine line 92, you can see that a « **struct** » structure has been declared outside the scope of the « **QuaternionEditor** » class. So, the question arises: Why do you use a structure? Primarily, it is because of its characteristics. Structures are commonly used to define the types of data stored on the stack, allowing for faster and more efficient manipulation of instances of this type when working with objects.

As seen in the previous section of this chapter, quaternions consist of four dimensions represented by the « **x** », « **y** », « **z** » and « **w** » components. Therefore, it is necessary to declare these variables within the scope of the structure and also explicitly define a constructor to initialize their values.

```
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
```

As you already know, to rotate a vertex using quaternions, you need to follow the equation mentioned in Figure 3.1.q from the previous section. Therefore, you will begin by defining a rotation Quaternion following the operation detailed in Figure 3.1.o. To do this, implement a static method called « **`Create`** » that takes a scalar value for the angle and a three-dimensional vector for the axis as arguments.

```
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
```

Examining line 113, the « **`Create`** » method returns a new rotation Quaternion in scalar-vector form. In other words, the « **`v.x`** », « **`v.y`** » and « **`v.z`** » components correspond to the vector part, while « **`cos`** » represent the scalar part.

To continue with the process, perform the conjugation of the rotation Quaternion, inverting its vector component. For this purpose, declare a new static method called « **`Conjugate`** ». This method will be responsible for carrying out the mentioned operation.

```
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    }
```

Examining line 121, you can see that the mentioned method returns a new « **`HQuaternion`** », following the equation presented in Figure 3.1.i from the previous section. Now, you only need to perform the necessary multiplications. To do this, apply the equation presented in Figure 3.1.q as follows,

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

In the previous example, in line 124, you can see the declaration of a new static method called « **`Multiplication`** », which takes two arguments: two quaternions.

Within its scope, the operation detailed in Figure 3.1.g from the previous section is implemented, illustrating the multiplication of two quaternions.

Up to this point, several points have been defined in the Scene view that together form a geometric cube. Since each point corresponds to a three-dimensional vector, it will be necessary to declare a method that allows us to assign a new position to each point and thus achieve the rotation effect. For this, you need to declare a new static method called « **`Rotate`** ». This method will take the position of the cube points, three rotation axes, and an angle as arguments. Why is the method public? You will use it later within the « **`SceneGUI`** » method to rotate the vertices defined for the cube.

```
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    }
```

Following the equation presented in Figure 3.1.q, if you examine line 140, a new « **`HQuaternion`** » called « **`q`** » has been declared, which is initialized with the result of the « **`Create`** » method. Then, in line 141, another « **`HQuaternion`** » called « **`_q`** » is declared, obtaining the conjugate of the previously declared

rotation Quaternion. Next, in line 142, a new Quaternion representing the point to be rotated is declared.

Finally, in line 144, a new « **HQuaternion** » called « **rotatedPoint** » is declared, obtained by multiplying « **q** » and « **p** » and then multiplying it by the conjugate of « **q** ».

It is important to note that the new position of a point corresponds to a three-dimensional value. For this reason, if you revisit line 142, you can see that the « **w** » component of the « **p** » Quaternion is initialized to 0f. Similarly, the « **Rotate** » method returns a three-dimensional vector.

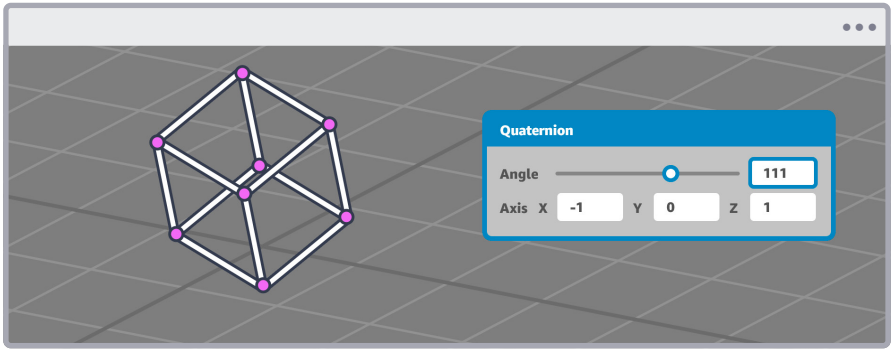
The only thing left is to apply the rotation to the vertices themselves. To do this, go to the « **SceneGUI** » method and call the « **Rotate** » method for each vertex defined in the cube.

```
52      public void SceneGUI(SceneView view)
53      {
54 >        vertices = new List<Vector3> … ;
65
66          float angle = m_angle * Mathf.PI / 180;
67
68          for (int i = 0; i < vertices.Count; i++)
69          {
70              vertices[i] = HQuaternion.Rotate(vertices[i],
                m_axis, angle);
71              Handles.SphereHandleCap(0, vertices[i],
                Quaternion.identity, 0.1f, EventType.Repaint);
72          }
73
74 >        int[][] index = … ;
89
90 >        for (int i = 0; i < index.Length; i++) …
94      }
95
```

Examining line 70, observe the use of the « **`Rotate`** » method to change the position of each vertex defined in the vector list. For logical reasons, this process is carried out within the « **`for`** » loop before drawing the spheres that define the graphical position of each vertex.

For the Rotation angle, a new variable called « **`angle`** » has been defined, which is multiplied by PI (3.14159265f) and divided by 180. Why? As you know, by default, quaternions calculate angles in radians. Therefore, it is necessary to convert the angle from degrees to radians to apply the Quaternion rotation.

Upon returning to Unity and modifying the angle value, you can see that the vertices rotate according to the orientation defined in the axes.



(3.2.e)

Summary.

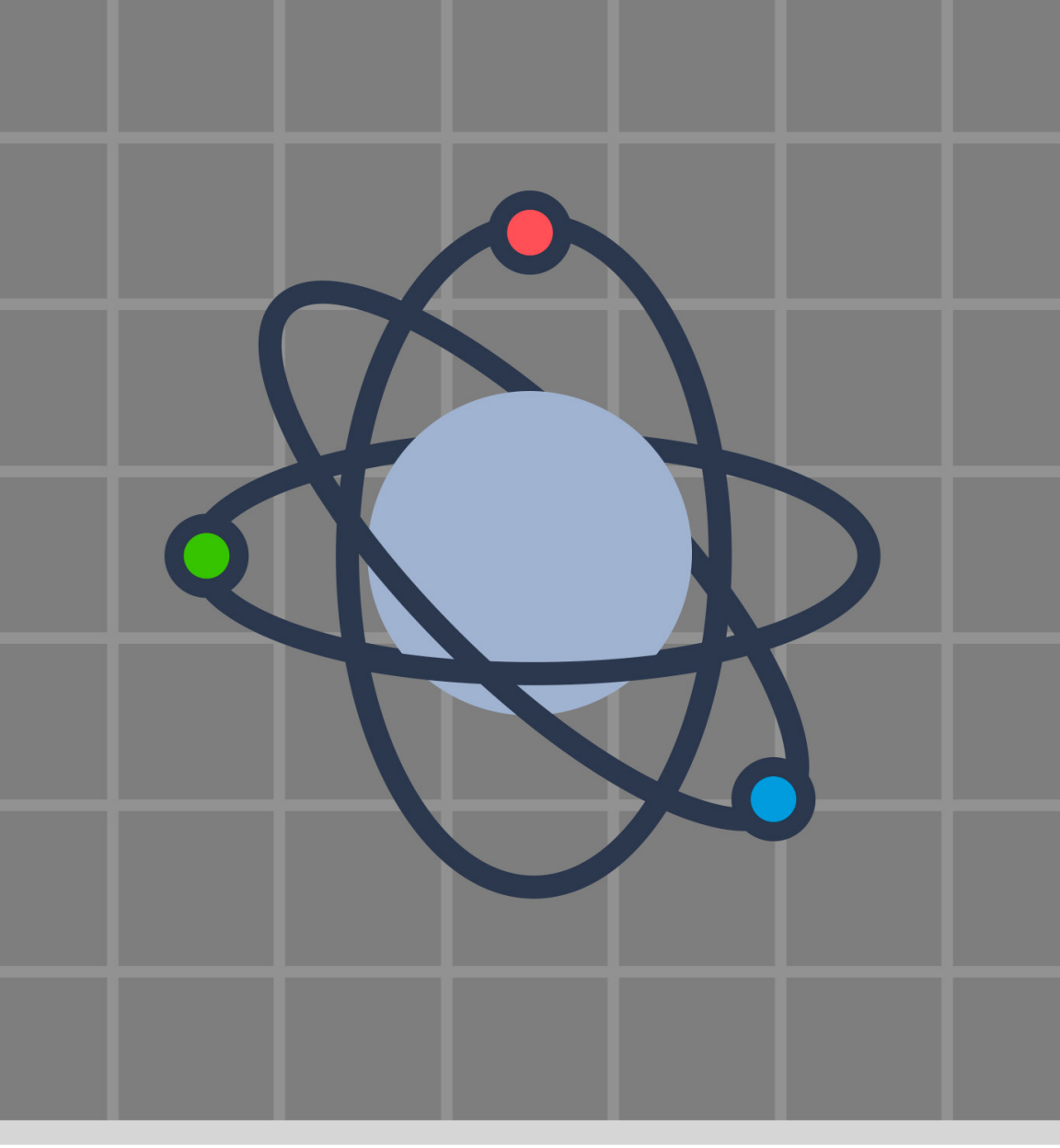
This chapter delved into the realm of quaternions, exploring their definition and structure. It also investigated how quaternions serve as a four-dimensional extension of complex numbers and discussed their unique properties, making them well-suited for representing rotations in three-dimensional space.

Various Quaternion operations were covered, with a focus on Quaternion multiplication, providing its implementation in C# to illustrate how these operations work in practice.

Additionally, the concept of the Quaternion conjugate and its relation to rotational inversion was introduced. Exploring how quaternions can represent points in three-dimensional space, highlighting their benefits in spatial transformation.

A simple example was offered to explain the process of performing rotations using rotation quaternion to demonstrate the mathematical calculations involved. This example emphasized the advantages of quaternion over rotation matrices and showcased their ability to avoid issues such as gimbal lock.

To wrap things up a Unity tool was created to demonstrate the practical applications of quaternions. This involved rotating the vertices of a cube in three-dimensional space, underlining their effectiveness in handling complex spatial transformations in graphics programming.



Chapter 4.

Rotation matrices and Euler Angles.

4.1. Introduction to the function.

The previous chapter explored the application of quaternions to rotate a cube, which is composed of eight pre-defined vertices. This naturally leads to the question: are there alternative methods for rotating vertices or points in space? Depending on the specific requirements of the project being developed, rotations using matrices can be beneficial, particularly in fields like computer graphics or when converting from Euler angles to quaternions.

Rotation matrices are mathematical tools that facilitate precise and efficient rotations in three-dimensional space. These matrices serve as numerical representations of transformations that can rotate an object around a point or axis in space. Now, look at the definition of the following matrix to understand better.

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

(4.1.a)

At first glance, Figure 4.1.a might appear complicated to grasp. However, upon considering its definition, you can deduce it is a two-dimensional rotation matrix. This matrix transforms a point's position by an angle measured in radians. How is this achieved? To answer that, focus on the trigonometric functions « sin » and « cos », which, as you know, are used in mathematics to relate angles to the ratios of the sides of a right-angled triangle.

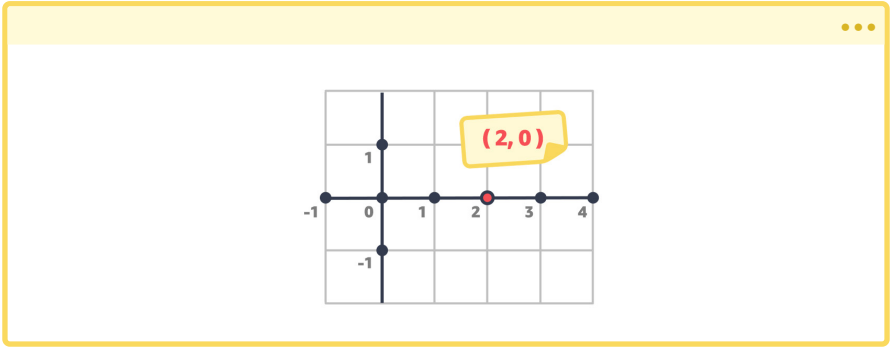
Assuming that « θ » (theta) represents a defined angle, you can assume that by multiplying the matrix from Figure 4.1.a by a two-dimensional point « p » located in the plane, the point will change its position. The result of this transformation will be « p' », which is the designation for the point after it has been rotated.

To understand the concept better, consider a point « p » in two-dimensional space with the following coordinates,

$$p = (2, 0)$$

(4.1.b)

On a Cartesian plane, such a point would be located as follows,



(4.1.c)

Suppose you want to rotate point « p » 45 degrees counterclockwise from the previous Figure. In this case, it becomes essential to consider both the rotation axis of the point and the measurement system used for the actual rotation. By default, all matrix rotations are carried out in radians, so before applying the matrix transformation, you need to convert the angle to radians. To do this, focus on the following transformation rule:

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

(4.1.d)

Here, « $\boldsymbol{\Omega}$ » (omega) represents the angle in degrees. In this case, follow the transformation illustrated in Figure 4.1.d, considering « $\boldsymbol{\Omega}$ » to be 45°.

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

(4.1.e)

Therefore,

$$\boldsymbol{\theta = 0.78539816}$$

(4.1.f)

After transforming the rotation value into degrees, directly apply the value from Figure 4.1.f to the matrix presented in Figure 4.1.a. For simplicity, round-up the values to three decimal places, resulting in the following:

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

(4.1.g)

Next, multiply the rotation matrix by the point defined earlier in Figure 4.1.b. How do you do this? It is crucial to note that the matrix used in your example has two rows and two columns. As matrices can be multiplied when the number of columns in the first matrix is equal to the number of rows in the second matrix, you need to consider the vector « $\boldsymbol{p}$ » as a matrix with two rows and one column, like this:

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

(4.1.h)

Building on the earlier discussion, proceed with the multiplication as follows:

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

(4.1.i)

Here, « p' » corresponds to the new point obtained after multiplying the rotation matrix « r » by the point « p ». Now, look at the multiplication order for these two objects. Since you are calculating the position of a new point on a Cartesian plane, it is crucial to have the values of the « x » and « y » coordinates of point « p ». To achieve this, use the variables and indices to visualize the matrix values defined earlier. This approach will help you understand the order of multiplication in the upcoming operations.

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

(4.1.j)

If you develop the matrix product shown, you obtain the following equations for each component:

$$x' = (a_{11} * x) + (a_{12} * y)$$
$$y' = (a_{21} * x) + (a_{22} * y)$$

(4.1.k)

Which is equivalent to,

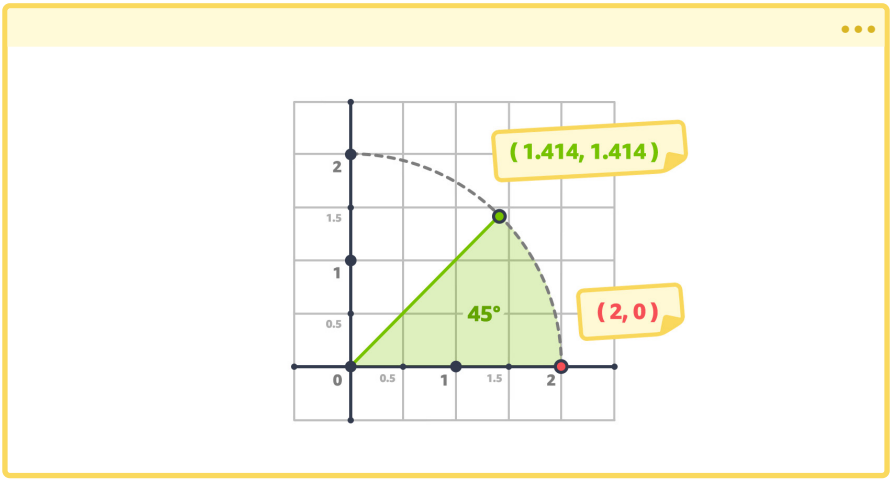
$$x' = (\cos(0.785) * 2) + (-\sin(0.785) * 0)$$
$$y' = (\sin(0.785) * 2) + (\cos(0.785) * 0)$$

(4.1.l)

Upon performing the corresponding arithmetic operations, the new coordinates for point « p' » will be:

$$x' = 1.414213562$$
$$y' = 1.414213562$$

(4.1.m)



(4.1.n. https://www.desmos.com/calculator/3uy3um4cox)

The code implementation of a rotation matrix will vary based on the specific needs or desired results of the operation. You can develop it as demonstrated in the following Figure to achieve the equality depicted in Figure 4.1.j.

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

```
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;
}
```

(4.1.o)

As observed in the preceding Figure, the « **RotationMatrix2D** » method returns the product of a four-by-four-dimensional matrix, denoted as « **m** », and a two-dimensional vector « **p** », representing the initial position of the point to be rotated. It is important to note that in C#, there are no two-by-two-dimensional matrices. Therefore, a four-by-four-dimensional matrix has been implemented, with only four indices used to store the values in the list.

This type of matrix is commonly employed in computer graphics to create interactive visual effects through shaders. With them, it is possible to rotate the UV coordinates of a geometric figure. For example, following method illustrates the implementation of the « **Rotate** » node in degrees, included in Shader Graph.

```
void Unity_Rotate_Degrees_float(float2 UV, float2 Center, float
Rotation, out float2 Out)
{
    Rotation = Rotation * (3.1415926f/180.0f);
    UV -= Center;
    float s = sin(Rotation);
    float c = cos(Rotation);
    float2x2 rMatrix = float2x2(c, -s, s, c);
    rMatrix *= 0.5;
    rMatrix += 0.5;
    rMatrix = rMatrix * 2 - 1;
    UV.xy = mul(UV.xy, rMatrix);
    UV += Center;
    Out = UV;
}

```

Considering that a two-dimensional matrix contains only one rotation axis, in this case, the « Z » axis, a question naturally arises regarding what happens in the context of a three-dimensional matrix. When referring to 3D rotation matrices, we are talking about matrices with three rows and three columns. Each element in this matrix signifies how the rotation around a predefined axis influences each object coordinate.

Creating a 3D rotation matrix depends on the axis chosen for the rotation and the angle you want to use. For instance, when rotating around the « X » axis, the rotation matrix adopts a specific structure influencing the « Y » and « Z » coordinates of the object. The same logic applies to rotations around the « Y » and « Z » axes.

Examine the following matrices to understand this concept, especially focusing on the rotation setup for the « Z » axis.

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

(4.1.p)

Rotation setup for the « X » axis:

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

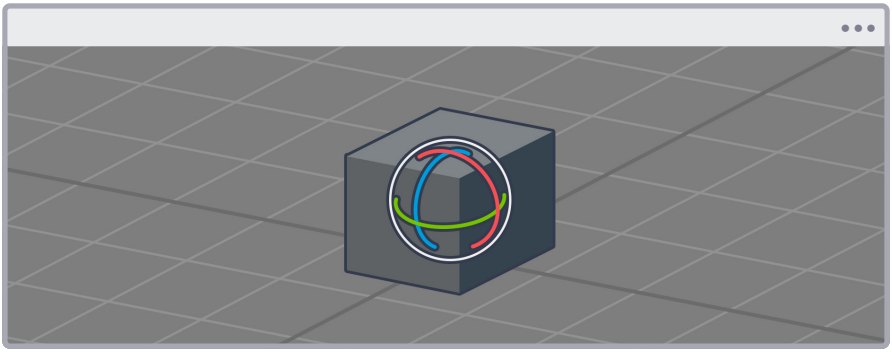
(4.1.q)

Rotation setup for the « Y » axis:

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

(4.1.r)

As depicted in the previous Figures, the matrices exhibit variations in configuration based on the chosen spatial axis. This is really practical as it enables the orientation of an object in a specific direction defined by three angles, or three distinct values.



(4.1.s)

Such behavior prompts the question: How can you achieve the rotation of an object using three different angles? To address this, you must enter the world of Euler angles, a common way to represent orientations in three-dimensional space. Euler angles consist of three angular values used to describe a sequence of rotations and are denoted as follows:

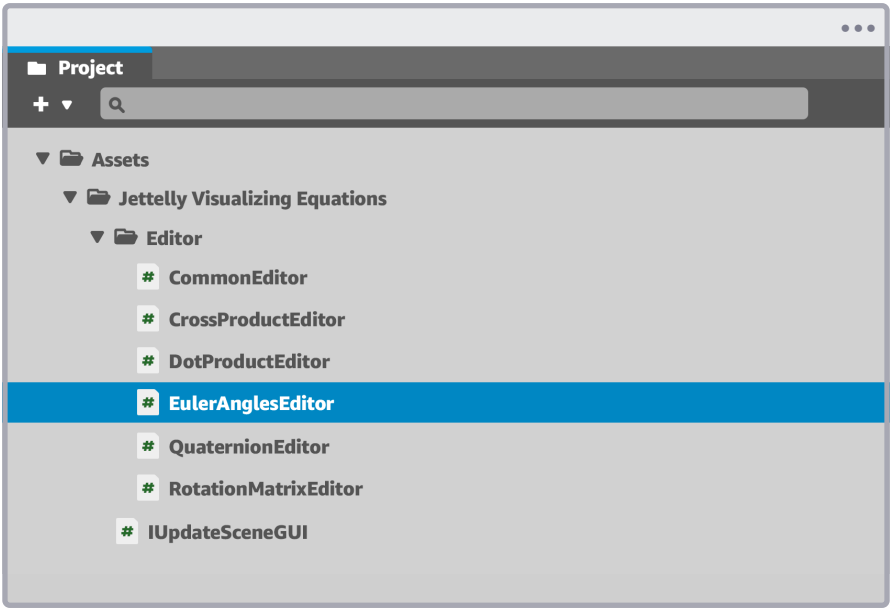
- « ψ » (Psi) for the « Z » axis.
- « θ » (Theta) for the « Y » axis.
- « ϕ » (Phi) for the « X » axis.

The order in which these angles are multiplied is very important, as it directly influences the final orientation of the object. In other words, depending on the chosen convention, you can obtain different rotation sequences like « ZYX », « XYZ », « YZX ». It is important to note that Euler angles may lead to singularity problems, resulting in what is known as a 'gimbal lock.' Hence, for more advanced applications, it is recommended to work with quaternions, offering greater flexibility in handling rotations.

4.2. Developing a Unity tool.

It is now a good idea to create a tool designed to facilitate the visualization of Euler angles, providing a deeper understanding of the gimbal lock phenomenon. To do this, implement the matrices discussed in the previous section.

To start, create a new script in your project called « **`EulerAnglesEditor`** ». As this script falls under the « **`EditorWindow`** » type script, it is essential to place it in the « **`Editor`** » folder, previously created.



(4.2.a)

After opening the file, be sure to extend your script from « **`CommonEditor`** » to integrate the functionalities of « **`EditorWindow`** ». Additionally, incorporate the abstract class « **`IUpdateSceneGUI`** » since you will be using the « **`SceneGUI`** » method once again.

As you know, it is crucial to declare a public static method to display the window of our tool in the Editor. Then repeat part of the process implemented earlier when creating the « **ShowWindow** » function in your script.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;

public class EulerAnglesEditor : CommonEditor, IUpdateSceneGUI
{
    [MenuItem("Tools/Euler Angles")]
    public static void ShowWindow()
    {
        GetWindow(typeof(EulerAnglesEditor), true,
        "Euler Angles");
    }

    public void SceneGUI(SceneView sceneView)
    {

    }
}

```

Similarly, include the methods « **OnEnable** », « **OnDisable** », and « **OnGUI** ». In the « **OnGUI** » method, initialize the variables so that they can be displayed in the previously declared window. In the first two methods, call the « **SceneGUI** » method using the « **SceneView.duringSceneGui** » function.

```
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
```

Next, define three new private methods that will be used in the definition of the matrices « ψ », « θ » and « ϕ ». Following established conventions, call them « **GetYaw** », « **GetPitch** », and « **GetRoll** », where each one can calculate rotation angles for vertices of an object in 3D space.

- The « **GetYaw** » method rotates vertices around the « Z » axis.
- The « **GetRoll** » method rotates vertices around the « X » axis.
- Finally, the « **GetPitch** » method rotates vertices around the « Y » axis.

Defining the matrices can be approached in various ways, such as using multiple lists of values. However, this time, you will use the UnityEngine « **Matrix4x4** » object type. As you already know, this object type corresponds to a standard four-by-four matrix.

```
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;
```

Continued on the next page

```
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
```

As shown in the example above, each value has been incorporated into the respective rows and columns, following the inherent structure of each matrix. It is relevant to note that, since no angle conversion has been performed (the argument), it will be calculated in radians by default. Consequently, you will later apply the function presented in Figure 4.1.d from the previous section to perform rotations in radians.

To continue, declare and initialize three floating-point values that will play a pivotal role in defining the rotation angles.

```
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
```

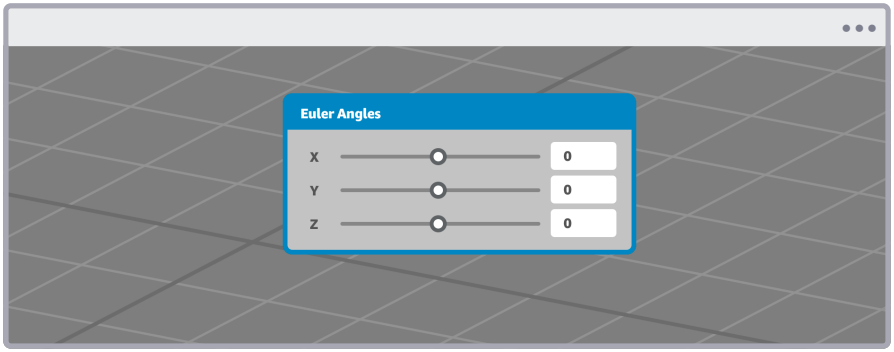
Looking at lines 7, 8, and 9, note that the floating-point variables are constrained to a range from -180 to 180 degrees. What is this? More specifically, why this range?

Even though this exercise could be done using a three-dimensional vector « **`Vector3`** », for practical reasons, it is better to simplify it by rotating the angles a total of 360 degrees.

Next, proceed with the initialization and projection phase of the variables into the tool's window. To do this, you need to intervene in both the « **`OnGUI`** » and « **`OnEnable`** » methods by executing the following operation.

```
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
```

As you know, the « **`DrawBlockGUI`** » method was previously defined within the « **`CommonEditor`** » class. Its main function is to draw information blocks on your tool's graphical user interface (GUI). Upon returning to Unity, observe that the rotation angles are now presented in your GUI.



(4.2.b)

Now, the only thing left to do is to draw some vertices in the Scene view to visualize the different rotations and Euler angles. For this, declare three lists, each corresponding to a rotation axis, namely « 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
```

In the example above (lines 16, 17, and 18), notice the separate declaration of lists for each axis. To illustrate this concept, incorporate points that, when connected, will form the shape of a circle. The purpose is to tangibly present the dynamics of vertex rotation in a two-dimensional space in real-time. Each of these graphical representations embodies a rotation angle, maintaining Unity's default orientation: « XYZ ».

Furthermore, a list named «**arrow**» has been declared on line 19. This list will be used to draw an arrow in the scene, giving a more precise visualization of the current orientation of the angles.

Continue by initializing your lists within the « **SceneGUI** » method. To do this, add eight points (vertices) within each list, considering their spatial coordinates.

```
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
```

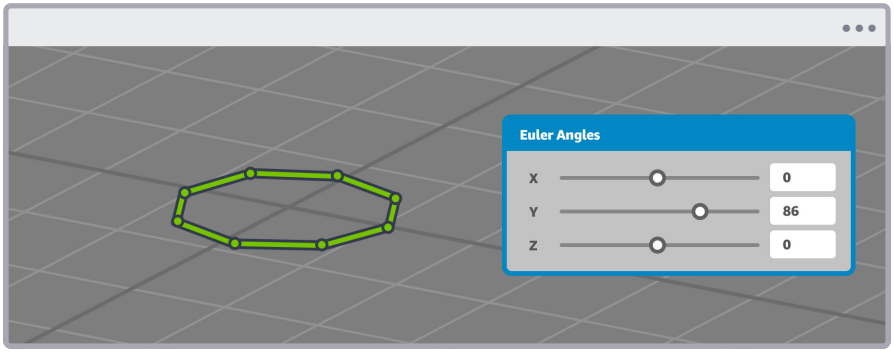
Continued on the next page

```
78            Handles.SphereHandleCap(0, circleY[i],
              Quaternion.identity, 0.05f, EventType.Repaint);
79        }
80
81        for (int i = 0; i < 8; i++)
82        {
83            Handles.color = Color.green;
84            Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
              circleY.Count]);
85        }
86    }
87
```

As shown in the Figure above, specifically in lines 62 to 69, a total of eight points have been incorporated in the « **circleY** » list. This list is designed to visually represent rotation around your tool's « Y » axis. Subsequently, in the first « **for** » loop (lines 74 to 79), use the « **SphereHandleCap** » function to draw a rotating sphere for each point on the list, with each sphere being able to rotate around its reference point. This rotation is achieved by defining « **circleY** » as the result of multiplying each point in the list by the « **GetPitch** » matrix, as shown in line 76.

It is worth noting that the rotation is executed in radians. This is because « **GetPitch** » takes « **degreeY** » as an argument, which in turn is equal to the angle multiplied by the result of dividing « **PI** » by 180 (line 72).

In the second « **for** » loop (lines 81 to 85), you can observe the creation of lines connecting each point to the next in the function of the intersections present in the list that has been previously initialized. The purpose of this process is to graphically visualize the sides of the polygon presenting a visual approximation of the circle. This procedure helps you understand the tool better. If you save the changes and return to Unity, you will notice that the « Y » axis has been added to your scene. In addition, you will be able to adjust its orientation by modifying the value of the « **Y** » property of your tool.



(4.2.c)

Next, return to your script and add the « X » axis following the same logic as outlined earlier, which consists of:

- Initializing the points that define a polygon in the list.
- Repeat each of them to draw a sphere at the position of each point.
- Repeat each point again to draw lines between them.

```
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
```

Continued on the next page

```
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],
              Quaternion.identity, 0.05f, EventType.Repaint);
92
93            circleX[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
              circleX[i]);
94            Handles.color = Color.red;
95            Handles.SphereHandleCap(0, circleX[i],
              Quaternion.identity, 0.05f, EventType.Repaint);
96        }
97
98        for (int i = 0; i < 8; i++)
99        {
100           Handles.color = Color.green;
101           Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
              circleY.Count]);
102
103           Handles.color = Color.red;
104           Handles.DrawAAPolyLine(circleX[i], circleX[(i + 1) %
              circleX.Count]);
105       }
106   }
107
```

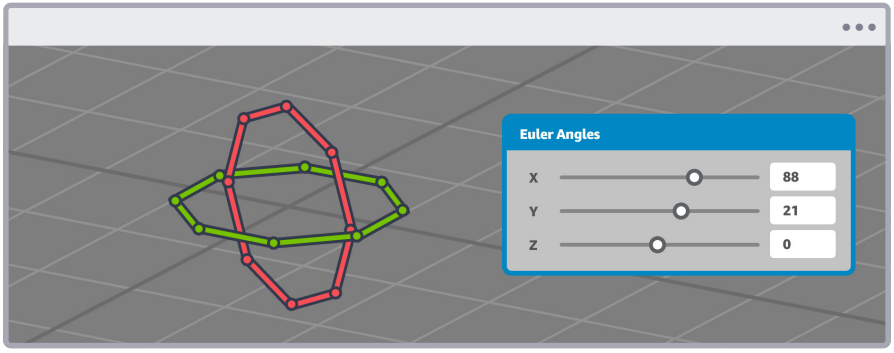
In the piece code above, if you pay attention to lines 74 to 81, previously defined points in the « **`circleX`** » list have been included to generate a polygon that, in this case, helps visualize the rotation angle around the « X » axis. It is worth noting that each point has been multiplied by 0.9f mainly to reduce the radius of the polygon formed by the points as a whole. A variable with this value could have been declared and initialized to avoid repeated code, but this exercise has been done for illustrational purposes.

One factor to consider is calculating the Euler angle for the « X » axis. As you can see in code line 93, the operation is done in two multiplications:

- First, the « **`GetRoll`** » matrix is multiplied by each point in the list.
- Then, the result of the operation is multiplied by the « **`GetPitch`** » matrix.

It happens due to the peculiar interaction of Euler angles. When rotated around the « Y » axis, the « X » axis also rotates as if they were linked. The same type of interaction occurs about the « Z » axis. These particular connections create complications when working with Euler angles because a rotation axis can lose relevance in certain situations, leading to what is known as gimbal lock. This effect can make it challenging to represent orientation in space correctly.

On returning to Unity, notice that the « X » axis is now present in the Scene view. Furthermore, you can adjust its orientation using the « **`x`** » property in your tool.



(4.2.d)

Continue by initializing the last matrix, corresponding to the « Z » axis. To do this, once again include the eight points that have been previously defined, which will be added to the list named « **`circleZ`** ». These points will generate a new polygon in the Scene view.

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

Continued on the next page

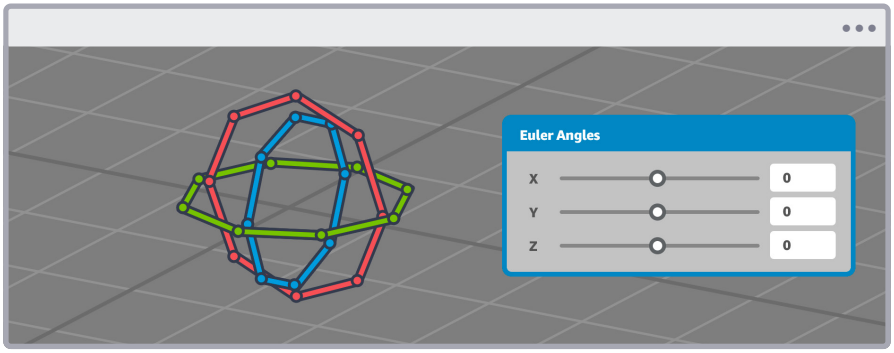
```
108             Handles.SphereHandleCap(0, circleX[i],
                Quaternion.identity, 0.05f, EventType.Repaint);
109
110             circleZ[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
                (GetYaw(degreeZ) * circleZ[i]));
111             Handles.color = Color.cyan;
112        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

As you can see in the previous operation (lines 86 to 93), you have repeated the same procedure again, but this time, it has been applied to the « Z » axis. In other words, you initialize each point in the « **`circleZ`** » list and then repeat it through these points in the two corresponding « **`for`** » loops.

If look at line 110 of the code, notice that to calculate the third axis, at least three multiplications are performed:

- First, the « **`GetYaw`** » matrix by each point in the list.
- Then, the previous result by the « **`GetRoll`** » matrix.
- Finally, the previous operation by the « **`GetPitch`** » matrix.

In this way, rotations are influenced by the rotations that precede them, creating the characteristic behavior of Euler angles. Returning to Unity, you can observe the different rotation axes with their respective angles and properties. In fact, if you configure the angles numerically as « **`90°, 0°, 0°,`** » you will lose one rotation angle, leading to a gimbal lock.



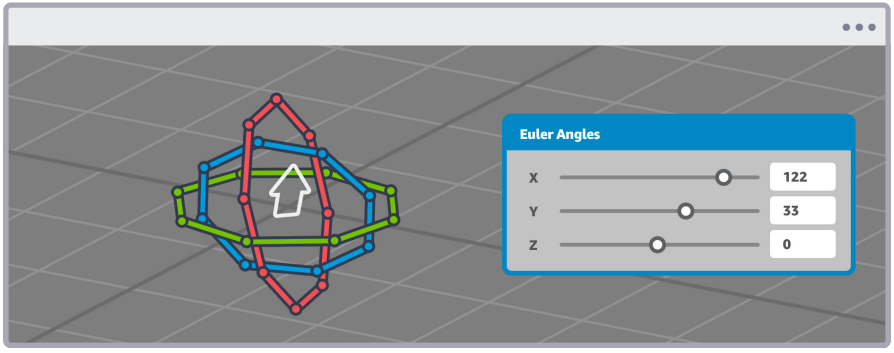
(4.2.e)

Now proceed with initializing the « **`arrow`** » list, which was declared earlier. This list consists of points that have already been defined and form the visual representation of an « **`arrow`** ». You will use an arrow to visualize the orientation of the Euler angles. To carry out this process, go to the end of the code block in the « **`SceneGUI`** » method and add the following lines of code.

```
115      for (int i = 0; i < 8; i++)
116      {
117          Handles.color = Color.green;
118          Handles.DrawAAPolyLine(circleY[i], circleY[(i + 1) %
             circleY.Count]);
119
120          Handles.color = Color.red;
121          Handles.DrawAAPolyLine(circleX[i], circleX[(i + 1) %
             circleX.Count]);
122
123          Handles.color = Color.blue;
124          Handles.DrawAAPolyLine(circleZ[i], circleZ[(i + 1) %
             circleZ.Count]);
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) *
             (GetYaw(degreeZ) * arrow[i]));
141      }
142
143      for (int i = 0; i < arrow.Count; i++)
144      {
145          Handles.color = Color.white;
146          Handles.DrawAAPolyLine(arrow[i], arrow[(i + 1) %
             arrow.Count]);
147      }
148
```

In the code above, there are three important blocks to focus on. The « **`arrow`** » list has been initialized between lines 127 and 136. Each vector contained in it has been multiplied by 0.5f to reduce the final size of the arrow representation. Then, in the « **`for`** » loop (lines 138 to 141), repeat for each point and perform the same process used for the « Z » axis of your tool. In this case, multiply each point by « **`GetYaw`** », then multiply the result by « **`GetRoll`** », and finally by « **`GetPitch`** ». This gives the orientation of the Euler angles. To finish, declare a new loop (lines 143 to 147) to draw a line connecting each point in the list.

Back in Unity, you can observe the complete functionality of the tool you have developed in this chapter.



(4.2.f)

Summary.

This chapter explored and defined rotation matrices, starting with a two-dimensional rotation matrix applied to a point in the plane. It served as an example to illustrate the matrix representation of a point in space and to understand matrix multiplication to obtain the transformed point.

Building on the foundation of two-dimensional rotational matrices, the study was broadened to three-dimensional space, defining rotations around each corresponding axis. In this way, Euler angles were used to represent and transform the orientation of an object.

Following the Unity convention, sequences of rotation were performed, ensuring consistency in your approach to applying Euler angles.

You concluded by creating a Unity tool to construct three geometric rings as a representation of a gimbal. Additionally, the gimbal lock problem was explained, which occurs when one rotation angle is lost.

Finally, the advantages of rotation matrices were highlighted, specially when dealing with angle loss. Understanding these concepts is essential for efficient 3D graphic programming.

Conclusion.

Throughout this book, we have explored a fundamental set of mathematical concepts that are essential in the world of programming and technical art. Through detailed analysis, we have broken down the equations of Dot Product, Cross Product, Quaternions, and Euler angles. Furthermore, we have demonstrated how these mathematical tools can be applied to solve various problems in developing interactive applications and video games.

We began with **Chapter 1**, where we delved into the Dot Product. We learned about its nature and applications, we illustrated concepts with simple examples, programmed in C# to perform summations, and created a tool in the Unity editor to visualize its behavior.

In **Chapter 2**, we delved into the Cross Product. We started by defining its operation through the right-hand rule and gave examples of its implementation using linear transformations with matrices. Later, we explored its geometric properties and concluded with a three-dimensional visualization tool to help you understand the concept better.

Then, in **Chapter 3**, we introduced the fascinating world of Quaternions, a versatile mathematical tool with applications in three-dimensional rotations. We explored its structure, operations, and ability to represent spatial transformations.

Finally, in **Chapter 4**, we tackled rotation matrices and Euler angles. We began with two-dimensional rotation matrices and then Euler angles for three-dimensional. We discussed Unity's convention for rotation sequences and the challenges of gimbal lock. Similar to previous chapters, we concluded with the creation of a tool that allowed you to visualize the behavior of its function and experiment with the loss of a rotation angle.

It is important to note that the world of programming and development is constantly evolving. However, regardless of new technologies and techniques, the understanding of these mathematical fundamentals will remain essential

for developers and technical artists. By mastering the mathematical equations presented in this book, readers will be better equipped to adapt to changing challenges and innovate in their own projects.

Glossary.

Function: A function is a mathematical relation that assigns each element of a set; a domain, to an element in another set of codomain. If the relationship is one-to-one, that is, for each element in the domain there is only one element int eh codomain, the function is bijective. A function can be described as a rule that associates an input with an output.

Method: In programming, a method is a set of instructions or procedures applied to an object or a class to perform specific tasks.

Operation: In mathematics, an operation is an action or procedure applied to one or more values to obtain a result. These operations can be arithmetic, logical, algebraic, among others.

Commutativity: In mathematics, commutativity refers to when the order of an operation does not matter. For example, saying 1 + 2 is equivalent to say 2 + 1.

Summation: In mathematics, summation is an operation that represents sums of many addends, even infinite ones. It is expressed by the Greek letter « Σ » sigma.

Variable: A variable is a symbol representing a value that can change within an equation, formula, or program, taking on different values during code execution.

Reference System: Corresponds to a set of spatial coordinates used to determine the position of a point in space.

Dot Product: Also known as the 'Scalar Product,' it is a mathematical operation between two vectors that gives a scalar value.

Cross Product: Also known as the 'Vector Product,' it is a mathematical operation between two vectors that produces a new vector perpendicular to the original vectors.

Anticommutativity: A mathematical property where the result of an operation changes sign when the order of the elements involved is altered. In an anticommutative operation, performing the operation in the reverse order produces the same result but with a negative sign. For example, given the operation a * b, can be the same as - (b * a).

Polygon: A polygon is a flat, closed geometric figure composed of straight-line segments called sides.

Quaternions: A quaternion is a type of complex number consisting of a real part and three imaginary numbers. It is used especially in 3D graphics and mechanics.

Rotation Matrices: Used to represent rotational transformations in three-dimensional space, these matrices change the orientation of objects around a reference point, such as the origin.

Euler: Euler angles are a set of three angles that describe the three-dimensional orientation of an object. They are used to specify the rotation of an object in terms of changes in its « X », « Y » and « Z » axes.

Equation: An equation is a mathematical equality that contains one or more unknowns and expresses a relationship between them.

Sigma: The Greek letter « Σ » sigma is used to represent a summation in mathematics. It is placed in front of a series of terms to be added.

Vector: A vector is a mathematical entity that has magnitude, direction, and orientation, usually represented as an arrow in space.

Component of a Vector or Matrix: These are the individual values that make up a vector or matrix, representing specific information in the corresponding mathematical structure.

Script: In programming, a script refers to a set of instructions or commands that are executed sequentially to perform a specific task.

GUI (Graphical User Interface): It is a visual and intuitive way to interact with a program or application through windows, buttons, menus, etc.

Gizmo: In computer graphics, a gizmo is a graphical tool or icon used to manipulate objects in a three-dimensional scene.

Handler: That corresponds to a function or routine used to handle specific events or actions in a program.

Anti-Aliasing: Anti-aliasing is a technique used in graphics and rendering to reduce the jagged effect on diagonal edges. Its implementation enhances the visual quality of the final image.

Arctangent: The arctangent is an inverse trigonometric function that returns the angle to a given tangent value.

Trigonometry: Trigonometry is a branch of mathematics that studies the relationships between angles and the sides of a triangle.

Radian: A radian is a unit of angle measurement used in mathematics and trigonometry, based on the length of the arc of a circle.

Stack: In programming, a stack is a data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added is the first to be removed.

Scalar-Vector: It is a mathematical operation where a scalar (number) is multiplied by a vector, resulting in a new vector with a different magnitude but the same direction and orientation.

Axis: In geometry and mathematics, an axis is a reference line around which rotation or reflection occurs.

Coordinate: Coordinates are a set of values that determine the position of a point in two-dimensional or three-dimensional space.

Scalar Value: Corresponds to a number that has magnitude but no direction, i.e., it is not associated with a coordinate system.

Matrix: An ordered table of elements arranged in rows and columns, matrices are used in mathematics and programming to represent data and perform linear operations.

IntelliSense: A feature found in most development environments that offers automatic code suggestions, completing keywords, functions, and variable names as you type.

Euclidean Space: A geometric space where concepts of Euclidean geometry, such as distance, angle, and properties of figures, can be applied.

Cardan: Cardan angles are three angles that describe the orientation of an object in three-dimensional space. They are also commonly known as Euler angles.

Conjugation: In mathematics, the conjugation of a complex number involves changing the sign of its imaginary part. In abstract algebra, the term "conjugation" can refer to different operations depending on the context.

Absolute Value: The absolute value of a number is its magnitude, always represented as a positive number regardless of its sign.

Interpolation: A technique used to estimate an unknown or unsampled value based on known data, using methods such as Lagrange polynomial or cubic spline.

Gimbal Lock: A problem in three-dimensional representation systems that arises when two of the three rotation axes align, limiting the freedom of movement to two dimensions.

Special thanks.

A Al Kooheji | Abraham Armas Cordero | Adam A D K Carames | Adam Bennett | Adam Gibson | Adam Myhre | Adrian Cuneo | Adrian Devlinn | Adriano Valle | Adrien Kissenpfennig | 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 Şekerdil | 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 Saidor | 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 Thomspon | Daghan Demirci | Damian Longman | Damian Turnbull | Đăng Trần

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

| 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 Tijsma | 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

| 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 Brząkalik | 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

| Rt Neal | Rt3D | Ruchi Hendre | Ryosuke Motrgi | S Julien Gauthier | Saiel | 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 | Unpyside | 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 | השנמ הסובורום ורטוביצ | 김현우 | 엄재천

Jettelly wishes you success in your professional career.