Version 0.0.4.

VISUALIZING EQUATIONS

ESSENTIAL MATH FOR GAME DEVS

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

Written by
Fabrizio Espíndola
Designed by
Pablo Yeber

Visualizing Equations

, Essential Math for Game Devs.

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

Visualizing Equations, version 0.0.4.

Jettelly ® All rights reserved www.jettelly.com

@jettelly.

Author.

Fabrizio Espindola.

Design.

Pablo Yeber.

Technical revision.

Martin Molina.

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.

Preface.

Whether we aim to generate a mechanic, an algorithm, create a tool, or engage in other programming endeavors, a common practice that define 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 do I translate certain equations into code? Due to our nature, there are times when we simply cannot recall mathematical symbols or the order of an operation, 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 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 written for Unity developers seeking to enhance their knowledge in solving and visualizing mathematical functions, as well as creating professional tools. It is assumed that the reader already knows and understands the Unity development engine; therefore, will not delve into the details of it.

Although previous knowledge of the C# programming language will be of great help in understanding some of the content presented in this book, it is not a strict requirement for the reader.

It will be essential 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, there will be a review of mathematical operations and functions necessary to fully understand what is being developed.

Conventions.

In this book, you will find a carefully curated selections of texts designed to stand out from the rest of the information. These texts, identified with the style « **`Function`** », focus on essential operations, variables, and methods for the effective implementation of code. Furthermore, a clear distinction has been made between the terms "method" and "function;" the latter is used to refer to functions included within the scope of a method.

The code block is presented consistently throughout the book:

```
4     public class ExampleClass : MonoBehaviour
5     {
6         void Start ()
7         {
8             // Code here …
9         }
10    }
11
```

Errata.

While writing this book, we have taken precautions to ensure the accuracy of its content. Nevertheless, we 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.

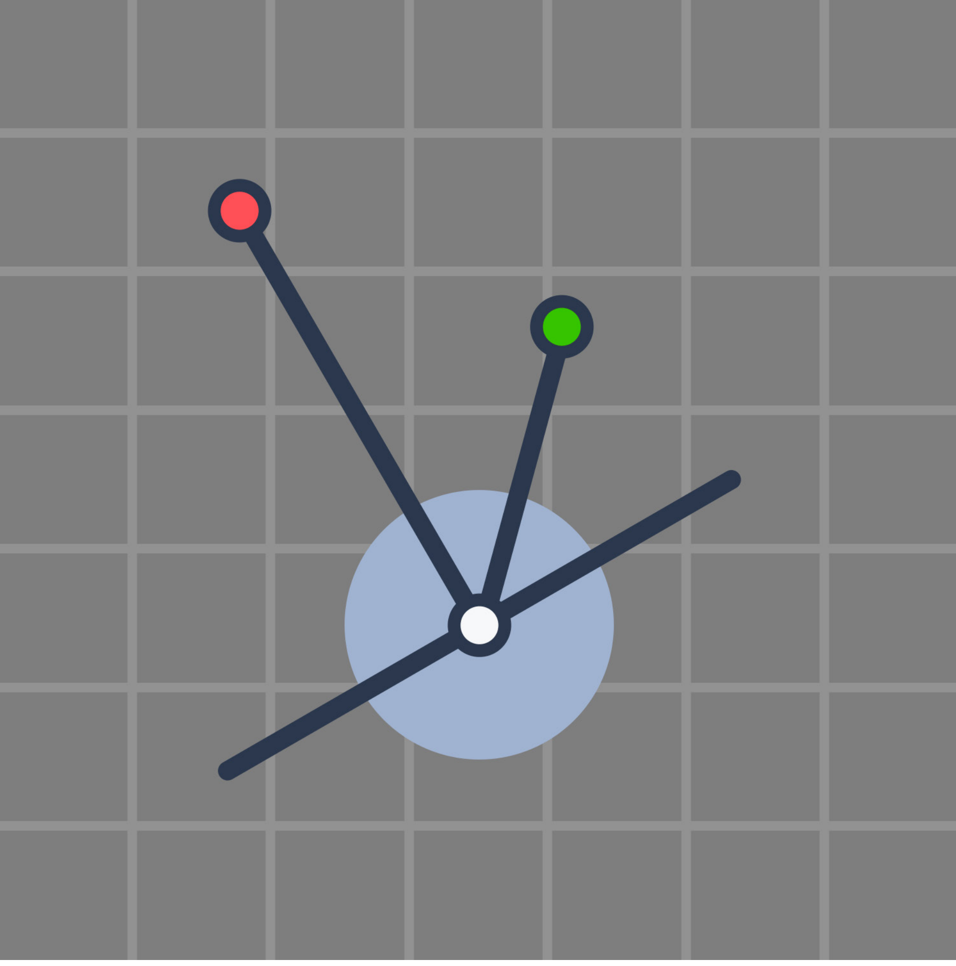
If you come across a conceptual error, a code mistake, or any other issue, we appreciate you sending a message to contact@jettelly.com with the subject line "VE Errata." By doing so, you will be helping other readers reduce their frustration and improving each subsequent version of this book in future updates.

Furthermore, if you have suggestions for adding sections of interest, please feel free to send us an email, and we will consider including such information in future 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 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. Introduction to the function.

We will begin our adventure by paying attention to the following equation:

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

(1.1.a)

Can you identify which function or operation the above equation belongs to? It is quite common to find this type of equations in programming-oriented books. They are used to illustrate a function that yields a specific result.

Given the title of this chapter, you may have already discovered to which operation the equation in Figure 1.1.a belongs. This equation refers to 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.

Its symbol « Σ » (sigma) represents the summation of scalar terms in a sequence, that is, a sum of constant values, complex numbers, or real numbers, which start at « i » and end at « n », both variables.

It's 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 time marches on, unfortunate, doesn't it?

We will now perform the following exercise to understand the concept,

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

(1.1.b)

In Figure 1.1.b, since « i » is equal 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 is equal to 5.

The next summation would be (3 * 2 - 1), and so on continuously.

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

(1.1.c)

Which is the same to say,

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

(1.1.d)

Therefore,

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

(1.1.e)

Now, returning to the equation shown in Figure 1.1.a, how could we translate such an equation? This is quite straightforward: the value of « $\boldsymbol{n}$ » refers to the number of dimensions the vector has, and « $\boldsymbol{i}$ » corresponds to a label for each component. For example, for a three-dimensional vector, we would have the following operation:

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

(1.1.f)

Which is the same to say,

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

(1.1.g)

Therefore

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

(1.1.h)

If we pay attention to the operation of Figure 1.1.f, we will notice that the variable « $\boldsymbol{n}$ » 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 implementation of the summation in code will depend on the context in which we are using it. For example, we could use a « **for** » loop to obtain the equation shown in Figure 1.1.b.

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

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

(1.1.i)

As we can observe in the previous figure, the « **Summation** » method returns 40 if « n » is equal to 5. However, when we talk about vectors, their implementation is different because, in this case, we would be seeking a result depending on:

- If the vectors are points in space.
- If the vectors are directions in space.
- If we want to obtain the projection of one vector onto the other.

Geometrically, the Dot Product corresponds to the projection of a vector « A » onto vector « B ». What does this mean? Imagine you are standing on a sunny day, and you have a light source projecting your shadow on the ground. Now, think about 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 onto the other, similar to how light projects a shadow on the ground.

It's worth noting that a summation can have different uses depending on the function it is meant to fulfill. The geometric interpretation of the Dot Product allows us to introduce another important equation, which is that it is given by,

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

(1.1.j)

From the previous Figure, this equation yields the same result as the equations in Figure 1.1.h, with the difference that, with the latter, we can obtain the angle « θ » between both vectors. Considering that the vectors « A » and « B » of the Figure 1.1.a are directions in space, their implementation could be the same as in the following figure:

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

```
float DotProduct(Vector3 p0, Vector3 p1, Vector3 c)
{
    Vector3 a = (p0 - c).normalized;
    Vector3 b = (p1 - c).normalized;

    return (a.x * b.x) +
           (a.y * b.y) +
           (a.z * b.z);
}
```

(1.1.k)

It's important to note that when working with vectors, we always need a reference point that acts as the origin of our 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`** ».

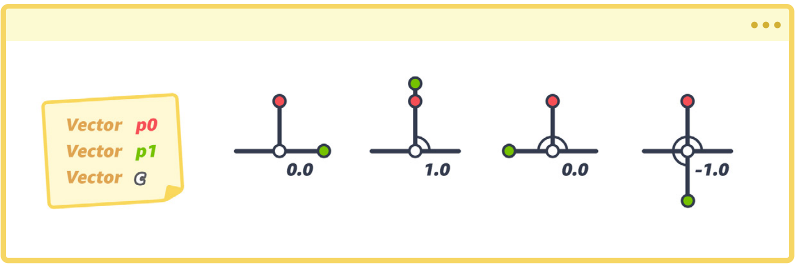
By subtracting vector « **c** » from vectors « **p0** » and « **p1** », we can interpret that vectors « **a** » and « **b** » are directions in space. It's important to highlight that these vectors have been normalized so that they magnitude is equal to 1.

$$||a|| = 1$$
$$||b|| = 1$$

(1.1.l)

So, we might ask, what can de Dot Product between two vectors be useful for? This will depend on what we are developing. For example, in a video game mechanics, we could use this function to determine the direction of one object relative to another.

The Dot Product between two-unit vectors returns the value of the cosine 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. This means that we could use these values to establish whether an object is in front or behind another. How would we do this?



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

In the above figure, we can observe 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, we can note that:

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

1.2. Developing a tool in Unity.

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

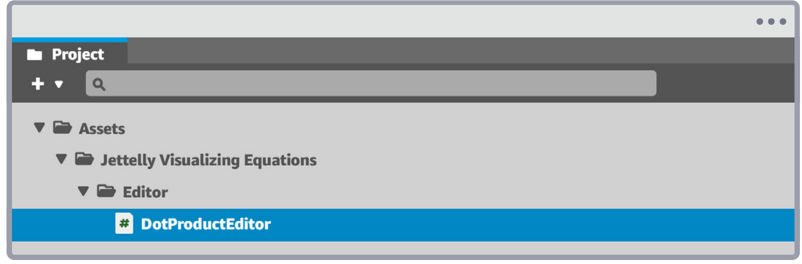
We will start the process by going to our project (the Project window in Unity) and create a new script called « **`DotProductEditor`** ». Once opened, we will make sure to extend it from « **`EditorWindow`** » for two particular reasons:

- Because it will be a visual tool.
- Because we will only need an instance of the same object in the Scene view.

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 our project. Why is that? According to the official software documentation:

> **Editor-type scripts add functionality to Unity during development but are not available in runtime builds.**

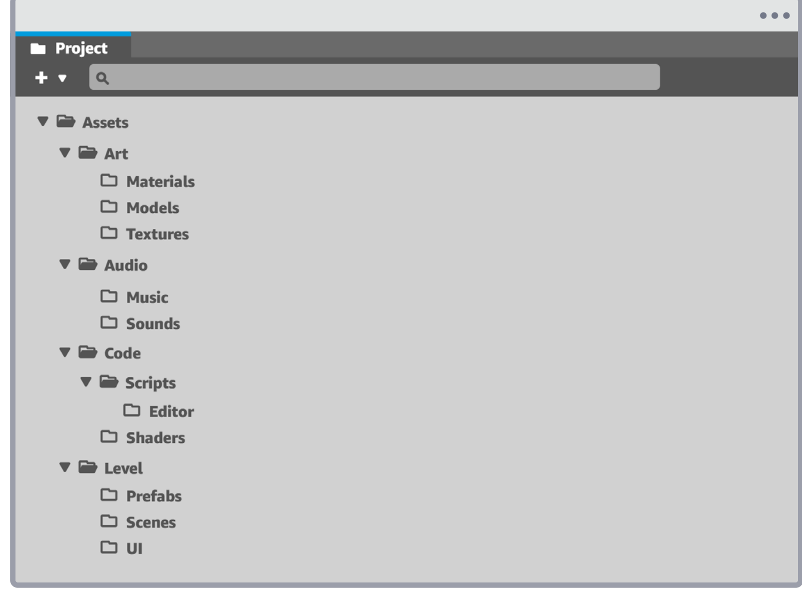
In other words, we cannot export a video game or application with Editor graphics.



(1.2.a)

In Unity, it's common 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. This can result in multiple Editor folder nested within subfolders throughout the project, making it challenging to locate scripts as the project expands in size.

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



(1.2.b)

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

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

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

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

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

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

```
1    using UnityEngine;
2    using UnityEditor;
3
4    public class DotProductEditor : EditorWindow
5    {
6
7    }
8
```

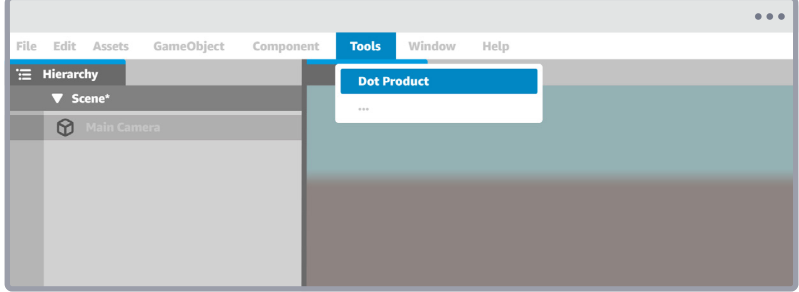
To officially begin the development process, we need to add a method that allows us to display a window for our tool. To do this, we can perform the following steps as indicated by Unity on their web platform. There steps are:

- Declare a public and static method to display a window in the Editor.
- Add the attribute « **MenuItem** » on the function, mentioned the menu path (where we want to enlist our 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 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 we have used three in the example. The first one refers to the window type, which is « **DotProductEditor** ». The second is a Boolean value that determines whether our window will be a pop-up or not, 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, which corresponds to the definition we made earlier using the « **`MenuItem`** » attribute in line number 6. From there, we can access the "Dot Product" window, which currently does not perform any action.



(1.2.c)

Considering that our tool will be used to represent the behavior of the Dot Product, it will be necessary to declare three vectors in our 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 properties of type « **`SerializedProperty`** ». This way, we can ensure that our tool is flexible and efficient enough to accurately represent the behavior of the Dot Product.

```
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 essential to include some functions in our code to see the vectors in action. When creating a tool, it's important to note that both the user interface (GUI) and the graphics added to the scene must be programmed separately. As a result, there will be some values that need to be initialized, such as the initial position of the vectors. To achieve this, we will add the following methods to our code.

- « **OnGUI** »: This will help us display information and handle events for our tool in the Dot Product window.
- « **SceneGUI »: This** corresponds to our own version of the « **Editor.OnSceneGUI** » method, which allows us to handle events in the Scene view.
- Another method we will include is « **OnEnable** », which will use to initialize values when the tool is active.
- Finally, « **OnDisable** » will be employ 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's worth noting that both the « **`OnEnable`** » and « **`OnDisable`** » methods, as well as « **`OnGUI »,`** belongs to « **`MonoBehaviour`** », meaning they are native to Unity. In the previous exercise, we can see them 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 its proper functioning. This is precisely why it has an argument of type « **`SceneView`** », 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.duringSceneGu**i » event (lines 24 and 29) is updated during the use of the Scene view. Consequently, if we active our Dot Product window and move the cursor within the Scene view area, we can debug the number of times the « **SceneGUI** » method is called.

The next step to be performed is to draw the vectors in the Scene view using the « **Handles.FreeMoveHandle** » method. However, we must remember that the vectors « **m_p0** », « **m_p1** », and « **m_c** » declared earlier have not yet been initialized. Therefore, if we perform the process at this point, they will all appear in the same position, which corresponds to "zero" in all their dimensions . To avoid this conflict, we will initialize the vectors in the « **OnEnable** » method:

```
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 we are developing.

Once the values are initialized, we can continue with the process of painting each vector as "Gizmos" in the Scene view. As mentioned earlier, it will be 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, we will implement a new method in our program that will handle the process.

```
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    }
```

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

If we return to Unity and try to move the points by clicking and dragging, we will notice that none of them change position. This mainly happens because we have not returned the return values of their global counterparts. Now, considering an "optimization" aspect, we will only carry out the process when there is a difference between the global vectors and those declared within the « **SceneGUI** » method. Therefore, it will be essential to use a conditional statement to determine if there is a difference between them, and we will 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
63
```

If we look at the code line 53, we 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.

Considering the « **SerializedProperty** » type properties, which are representations of serialized fields or properties in the Inspector window, in order for the current value of each vector « **p0** », « **p1** », and « **c** » to be displayed in the Dot Product window, we must use the « **FindProperty** » function, which takes a « **string** » path as an argument and returns a « **SerializedProperty** » object.

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

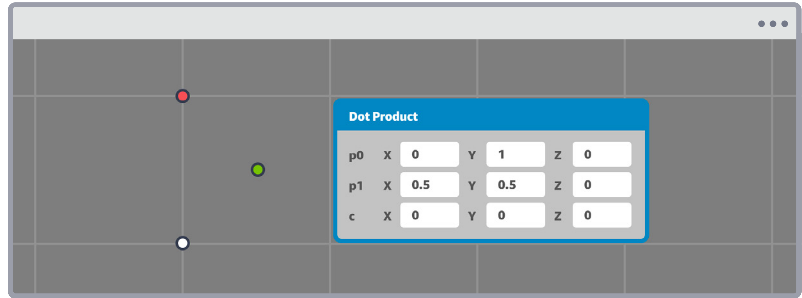
Now, we only need to display these properties in the Dot Product window. To do so, it will be necessary to go to the « **`OnGUI`** » method and consider at least three factors:

- Call the « **`SerializedObject.Update`** » method to update the representation of serialized objects.
- Create a field in the Dot Product window for each property. This can be achieved using « **`EditorGUILayout.PropertyField`** » function.
- Update the Scene view when there are value changes in the properties we are displaying. This can be achieved through « **`SerializedObject.ApplyModifiedProperties`** » function.

```
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
66
```

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

If we go back to Unity and select our tool, we will be able to modify the values of each vector directly from its respective property.



(1.2.d)

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

Up to this point, we have focused our efforts on creating graphics for our vectors. However, the purpose of all the aforementioned is to improve our understanding of the behavior of the Dot Product of two vectors. Therefore, we will continue to implement 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 our tool, it would be ideal to display a text in the Scene window that shows the real-time result of the Dot Product between « A » and « B ». For this reason, we will add a new global variable of type « **GUIStyle** » and modify its font size, font type, and color in the « **OnEnable** » method to obtain a text with a stronger graphical presence.

```
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
43          SceneView.duringSceneGui += SceneGUI;
        }
```

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

We will now continue by declaring and implementing a new method that we will call « **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    }
```

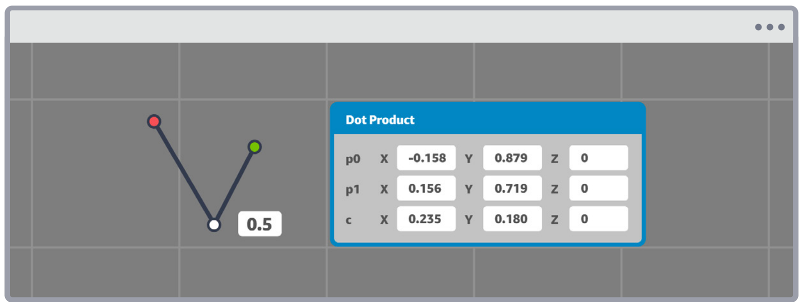
We can easily deduce that the arguments of the « **DrawLabel** » method (line 105) correspond to the points or vectors that have been previously declared in the « **SceneGUI** » method. In line 107, you will find the « **Handles.Label** » function, which creates a text considering:

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

Notice that the « **DotProduct** » method has been used to define the name of the text; it returns a floating-point number, meaning decimal places. For this reason, the « **ToString** » function has been used to change its format, and « F1 » to display only one decimal. The « **Handles.DrawAAPolyLine** » function is responsible for generating stylized graphical lines (anti-aliasing) at the points defined in its arguments.

Having completed the process, we should include the « **DrawLabel** » method at the end of « **SceneGUI** » and pass the vectors « **p0** », « **p1** », and « **c** » as arguments. If we return to Unity, we can observe how the value of the "Label" changes based to 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
```

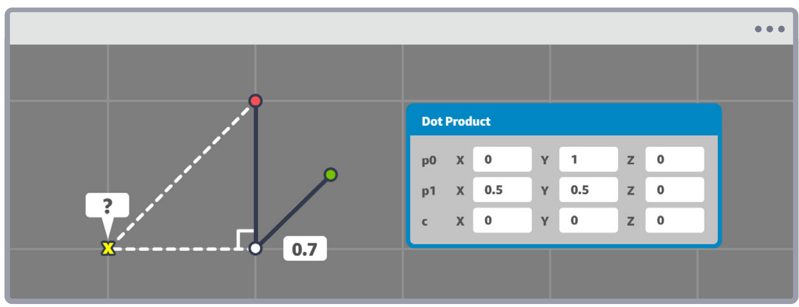


(1.2.e)

As mentioned earlier, the normalized Dot Product return a real number in a range from -1f to 1f with respect to a reference point. If we return to Unity and modify the position of the vectors in the Scene view, we 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.

The functionality of our tool is ready up to this point; however, we will add a surface, a line parallel to vector « **c** » as a "feature" that will help us to visualize the difference in position between « **p0** » and « **p1** ». To carry out this process, we will once again use the « **`Handles.DrawAAPolyLine`** » function, which, as we already know, requires two vectors to generate graphics.



(1.2.f)

As we can see in the previous Figure, a new vector has been referenced to the left side of vector « **c** » (white point), creating a right triangle. To determine a new position, we 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.

To achieve this, it will be necessary to apply the following equation:

$$R = C + HP$$

(1.2.g)

From the Figure 1.2.g, « C » refers to the vector we defined as the central point, « H » corresponds 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 », we could use the following trigonometric relationship to determine its angle:

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

(1.2.h)

But how would we do this? To achieve this, we will return to our 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, our method does not perform any operations. However, if we pay attention to its arguments, we will notice that it has three vectors; three input positions.

Since the arctangent requires the opposite and adjacent side, the first operation we need to perform within the method is 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    }
```

Next, we would need to calculate the arctangent as shown in Figure 1.2.h. To do this, we can use the static method « **Atan2** », which returns the angle in radians. It's worth noting that we will need to use the « **Mathf.Rad2Deg** » function 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, we need to convert the angle from degrees to rotation. To do this, it's necessary to understand the nature of the « **Quaternions** ».

The Quaternions were discovered by William Rowan Hamilton in 1843. These 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, we simply need to transform the angle obtained previously into a Quaternion. For this, we can use the « **AngleAxis** » function, which generates a rotation based on an axis. Considering that our 3D tool is designed within a two-dimensional plane, we will use the « **Z** » axis for its rotation.

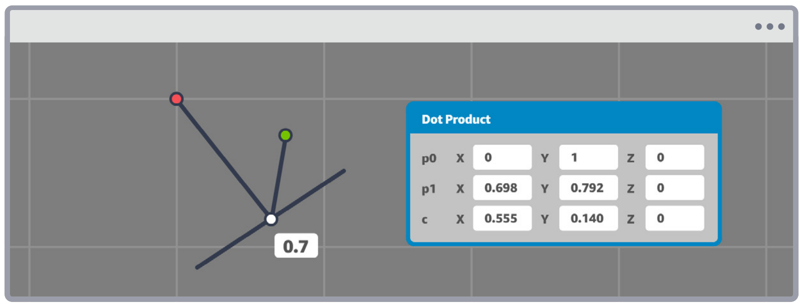
```
121   Vector3 WorldRotation (Vector3 p, Vector3  c, Vector3 pos)
122   {
123       Vector2 dir = (p - c).normalized;
124       float ang = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
125       Quaternion rot = Quaternion.AngleAxis(ang, Vector3.forward);
126
127       return c + rot * pos;
128   }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

As we can see in line 127 of the code, we have applied the equation detailed in Figure 1.2.g. The only thing left to do is to generate two new vectors, one to the right and the other to the left of « **C** » and add them to the « **DrawLabel** » method so that they can 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
```

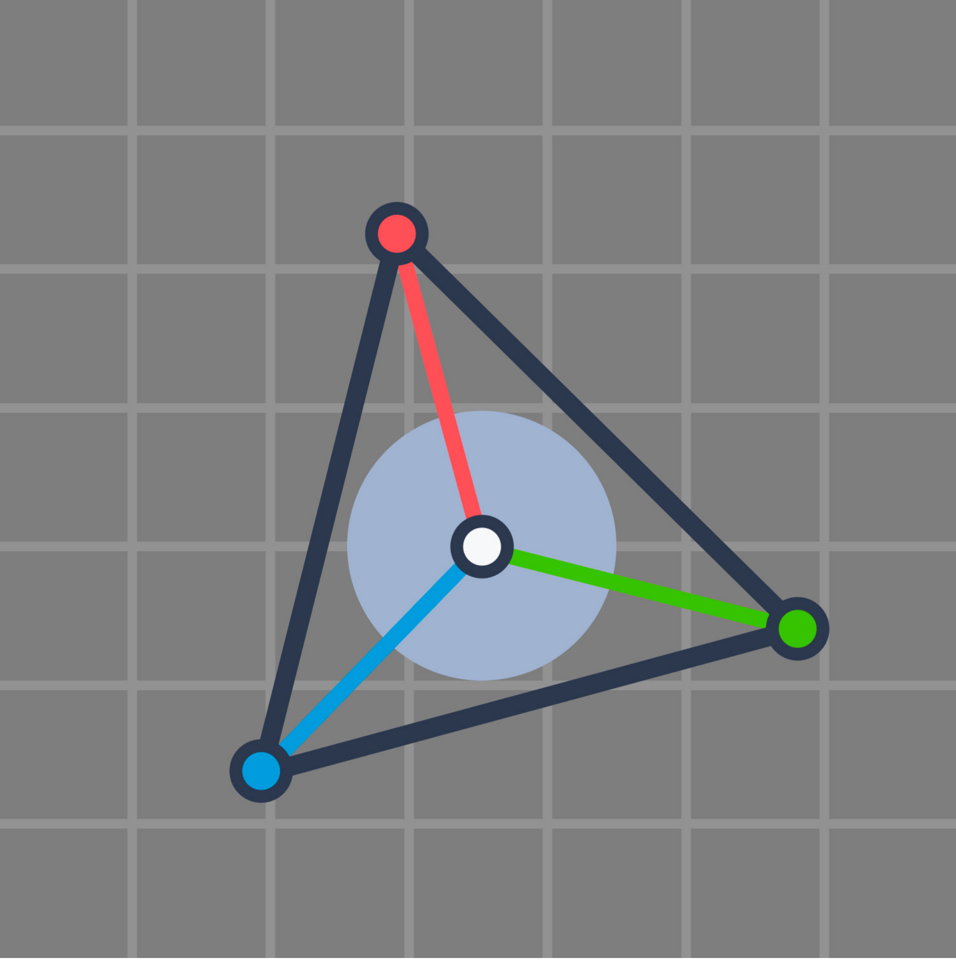
From the previous example, if we pay attention to lines 112 and 113, we can observe that two vectors « **cLef** » and « **cRig** » have been declared and initialized. These vectors create a displacement of one unit in the « **Y** » coordinate.

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



(1.2.i)

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



Chapter 2.
Cross Product.

2.1. Introduction to the function.

We will continue our adventure by discussing 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.

Let's 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 quite common to confuse the symbol « **×** » with the multiplication one when we are getting familiar with vector mathematics. However, it’s crucial to keep in mind that this symbol specifically refers to the Cross Product and not the multiplication operation.

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

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

(2.1.b)

To better understand the concept, let's proceed with the following exercise: We will define two vectors, « $\boldsymbol{A}$ » and « $\boldsymbol{B}$ », which will have the following components:

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

(2.1.c)

Using the formula presented in Figure 2.1.a, we can 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 we substitute the given components from Figure 2.1.c, we obtain:

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

(2.1.e)

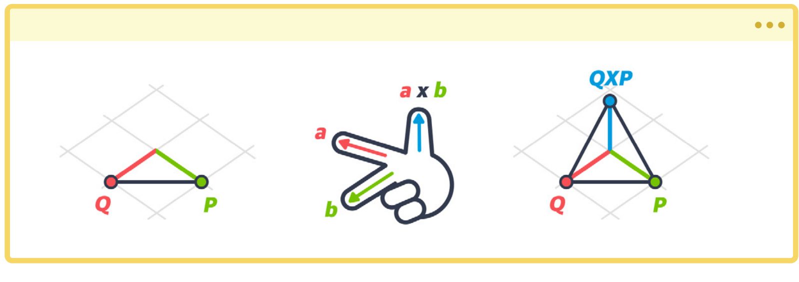
Therefore,

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

(2.1.f)

The resulting vector « $\boldsymbol{C}$ » 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 « $\boldsymbol{A}$ » and « $\boldsymbol{B}$ ». It's worth noting that 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 we use this operation for? In various application, 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 especially useful in triangulations and other geometry problems.



(2.1.g)

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

$$P \times Q = (P_yQ_z - P_zQ_y,\ P_zQ_x - P_xQ_z,\ P_xQ_y - P_yQ_x)$$

```
vector3 CrossProduct (Vector3 p, Vector3 q)
{
    float x = p.y * q.z - p.z * q.y;
    float y = p.z * q.x - p.x * q.z;
    float z = p.x * q.y - p.y * q.x;

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

(2.1.h)

If we observe the Figure 2.1.h, we will notice that within the « **CrossProduct** » method, three scalar variables « **x** », « **y** », and « **z** » have been declared, each containing a part of the equation. How could we express the same operation as a linear transformation?
Let's 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)

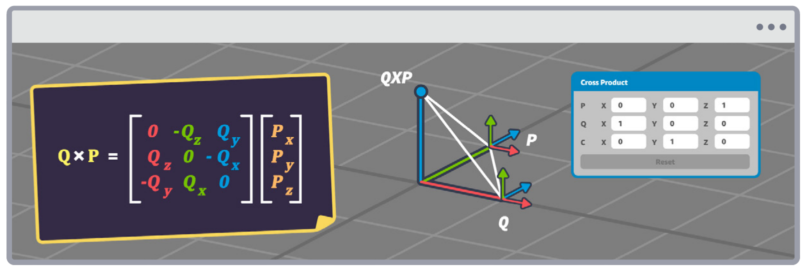
The definition presented in the previous Figure may seem complex, but its implementation in the C# programming language is actually quite straightforward. We just need to remember that the Cross Product is expressed using a tree-dimensional matrix of « Q » operating on « P ».

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

```
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;
}
```

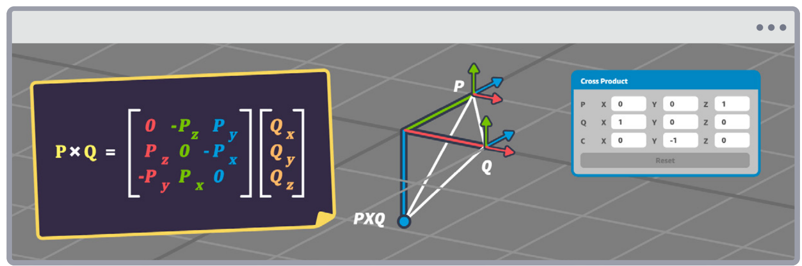
(2.1.j)

It is important to highlight that in the « **`CrossProduct`** » method presented in the previous Figure, a new four-by-four-dimensional matrix called « **`m`** » has been defined. This matrix stores each dimension of « **`Q`** » in the same order as it appears in the equation of Figure 2.1.j. One factor to consider is the direction of the Cross Product, which will be 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 we flip the matrix values, the Cross Product will be negated, pointing in the opposite direction.



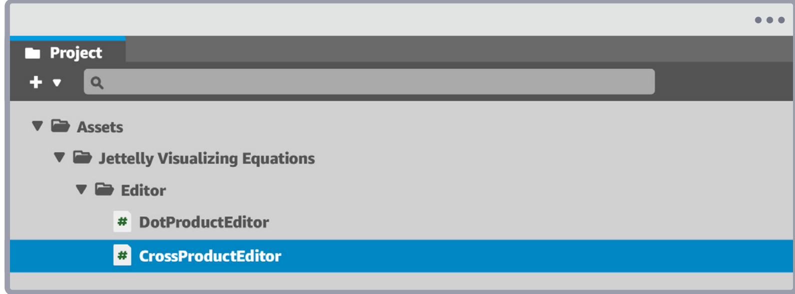
(2.1.l)

2.2. Developing a tool in Unity.

Continuing with the purpose of this book, we will create a tool in Unity that will help us better understand the nature of the Cross Product. To achieve this, we will repeat part of the process detailed in the previous chapter, where we used the « **`UnityEditor`** » dependency and extended our script from « **`EditorWindow`** ». Why are we doing this? Mainly,

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

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

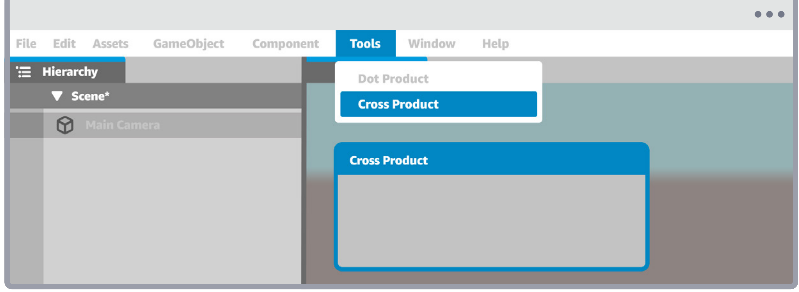


(2.2.a)

The first action we will take in out script is to add a static method that we will use to display a popup window in Unity.

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

As seen in the previous example, we have once again used the « **`GetWindow`** » function, which returns a tool window instance. For this instance, we have used "Cross Product" for illustration purposes. Upon returning to Unity, you can see that a new window has been added to the "Tools" menu. However, this window does not execute any function for now.



(2.2.b)

As we 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, we will 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
```

We can observe in lines of code 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 initializing the values of the vectors and properties, as well as updating the Scene view based on the new values of each vector, is necessary. To carry out these tasks, we will include the following methods in our script.

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

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

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

We will use at least two methods that have been previously employed during the development of "Dot Product" window tool. These methods correspond to:

- « **SceneGUI** », which we 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 leads us to face a problem: Repeated code! Since both the « **CrossProductEditor** » and « **DotProductEditor** » scripts are going to use the same methods, it would be ideal to implement them in separate classes for more efficient access. To achieve this, let's go to our Unity project and create two new scripts. The first on will be called « **CommonEditor** », and we will place it inside our Editor folder. It will be essential to ensure that « **UnityEditor** » is included in it and that it extends it from the « **EditorWindow** » class. Why? because our current tool will also extend from the latter, and we 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
15
```

By observing line 6 of the code, we can note that the « **DrawBlockGUI** » method has been declared and used in developing the Dot Product tool. It has been also marked as « **virtual** », which implies that it can be overwritten when 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, meaning this method does not require a specific algorithm within its scope. However, it will be necessary in several tools we develop of the « **EditorWindow** » type.

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

Next, we can simply extend our current tool from « **CommonEditor** » and include 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, we 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 the tool creation, it will be necessary to remove or comment out this line of code. Otherwise, Unity may throw a compilation exception.

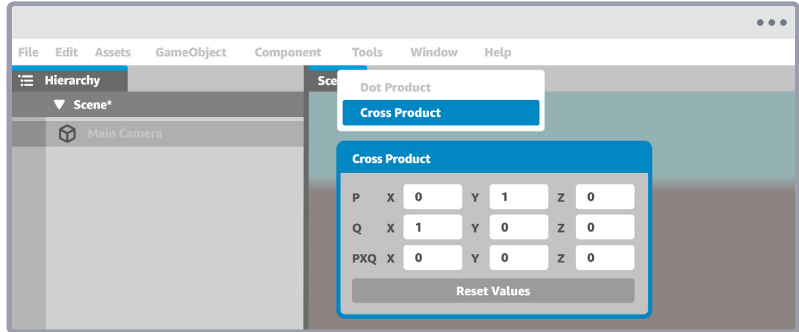
Then, we will initialize the serialized properties in the « **OnEnable** » method. Additionally, we will declare a new method called « **SetDefaultValues** », which we will use to assign default values to the « **m_p** » an « **m_q** » vectors.

```
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, this time we will include a button in the Cross Product window, which we will use to reset the vectors and their values to their default state. This process will be carried out solely for the purpose of enhancing our comprehension 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    }
```

From the previous example, it is important to remember that the « **DrawBlockGUI** » method (line of code 51 to 53) is currently implemented in the « **CommonEditor** » script. Therefore, we can make use of it through inheritance. As shown in line of code 60 to 63, a condition has been added 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 this point, we have configured the Cross Product window and its main functionality. Next, we will proceed to draw the tool in the Scene window. However, before continuing with the process, it will be necessary to declare a global variable of type « **`GUIStyle`** » which we will use 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. Just like in the previous chapter, its size, style, and color have been initialized in the « **OnEnable** » method, specifically in lines 41 to 43.

Before proceeding with the implementation of the « **SceneGUI** » method, we will 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
```

In order to facilitate the identification of the vectors in the Scene window, it is convenient to add text labels to them. If we analyze the internal structure of the « **DrawLineGUI** » method, we can see that it mainly focuses on two actions:

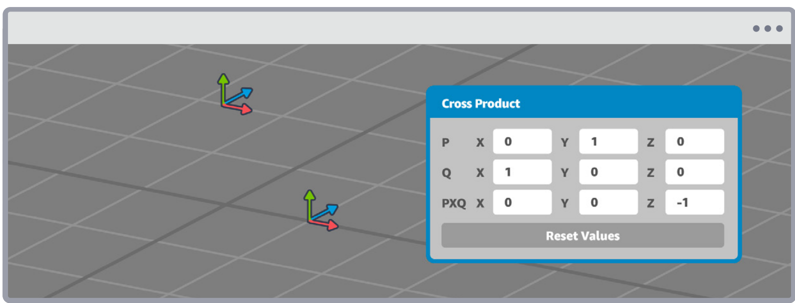
1. It draws a text using the « Handles.Label » function.
2. It draws a line from the current position of the vector to the world origin (point zero) using the « **Handles.DrawAAPolyLine** » function.

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

We will continue with the « **SceneGUI** » method by declaring two new three-dimensional vectors, « **p** » and « **q** », which we will use to return 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
```

If we return to Unity at this point, we will see that the vectors haven 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 important to note that at this stage, we can only modify the position of the "Handles" through the Cross Product window. Why? Because we have not yet assigned values of « **p** » and « **q** » to their global counterparts.



(2.2.d)

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

- We will declare and initialize a new three-dimensional vector using the « **CrossProduct** » method and represent it in the Scene window as a solid blue disc.
- We will verify if there have been any changes in the current vector position by using an « **if** » conditional that evaluates the user's interaction with the previously implemented « **Handles** ».
- In case of changes, we will 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 important 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) changes made to the tool can 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, we will notice two issues:

1. The Cross Product « **pxq** » appears on 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 CTRL + Z is pressed after changing the position of any of the Handles, the Cross Product window doesn't update correctly.

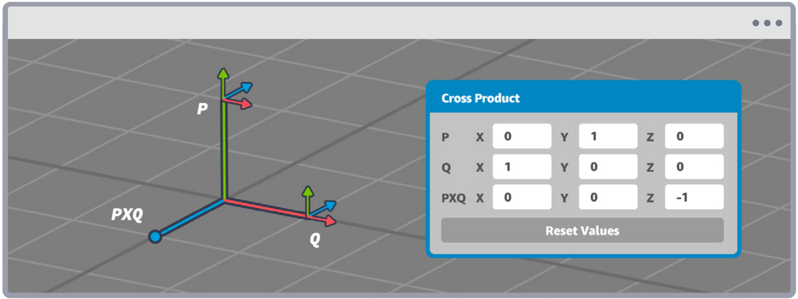
To address the first problem, we will include the « **DrawLineGUI** » method within the field as follows:

```
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    }
```

The code has been updated in lines 92 to 94. Essentially, we have added text and a line to each vector, which is reflected in the Unity Scene window. However, we still face an issue when undoing changes by pressing CTRL + Z. Fortunately, we 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, which is triggered after undoing a change in the Scene window. If everything has been done correctly, we should be able to observe the behavior of the Cross Product between vectors « **p** » and « **q** » in Unity.



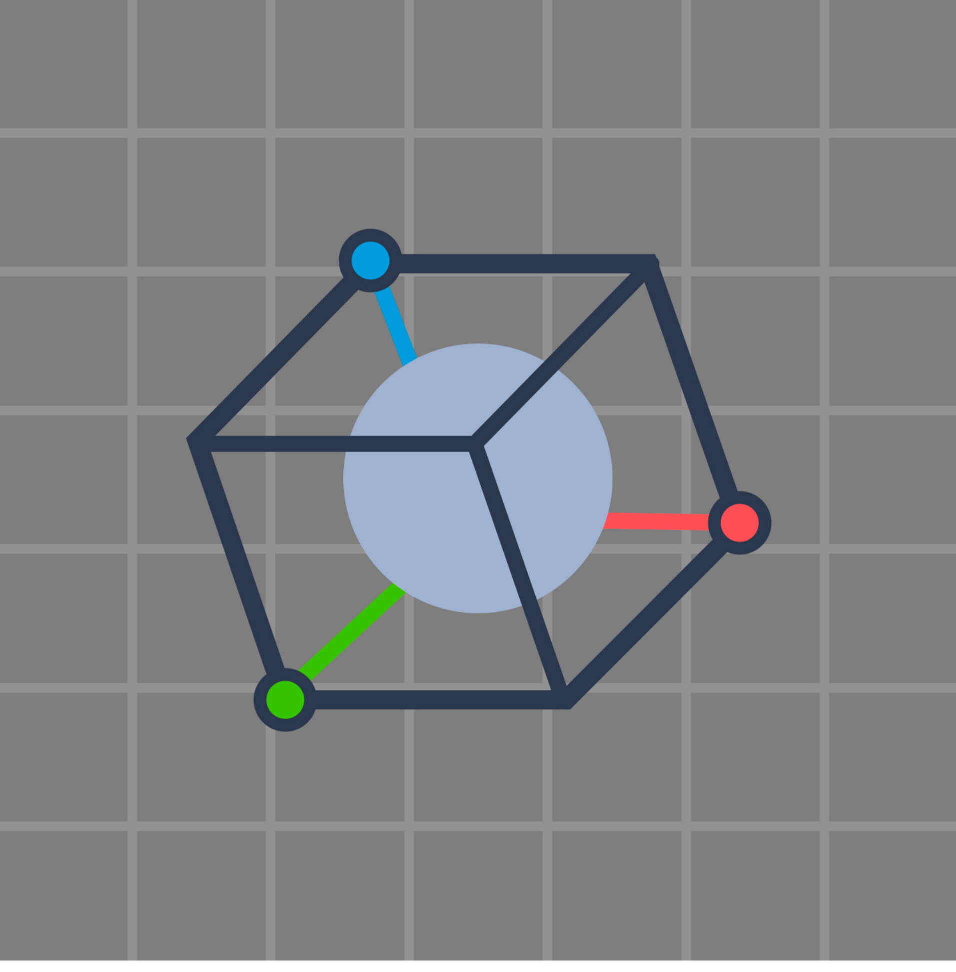
(2.2.e)

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

- Comment out the currently implemented « `CrossProduct` » method in the script.
- Define a new method based on Figure 2.1.j.

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

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



Chapter 3.
Quaternions.

3.1. Introduction to the function.

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

To understand their definition, let's pay attention to the following formula:

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

(3.1.a)

It is quite common to feel uncomfortable when attempting to interpret this type of equation for the first time. This misgiving is precisely due to our limited understanding of the properties of theses mathematical objects.

Since this type of mathematical object extends from the concept of complex numbers, we can deduce that the variables « i », « j » and « k » in equation 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 stablish a direct relationship with their corresponding axis:

$$i = x \ axis$$
$$j = y \ axis$$
$$k = z \ axis$$

(3.1.b)

The variables « $\boldsymbol{w}$ », « $\boldsymbol{x}$ », « $\boldsymbol{y}$ » and « $\boldsymbol{z}$ » correspond to real numbers, representing a point in space with components given by the triplet « $\boldsymbol{x}$ », « $\boldsymbol{y}$ », « $\boldsymbol{z}$ » and associated scalar « $\boldsymbol{w}$ ». Therefore, on one side, we have the coordinates representing the base of three-dimensional space, while on the other side, we have the 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,

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

(3.1.c)

Why are they imaginary? Basically, because they have the particularity that their square results in -1, meaning that they exist in a mathematical world distinct from real numbers. Now, we might ask ourselves, how can we perform the rotation of an object with multiple vertices considering the previous explanation? To do so, we must first consider at least three factors:

- The Quaternion product.
- Its conjugate.
- The rotation angle.

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

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

(3.1.d)

Where « s » represents the scalar value of the component « w », and « v » refers to the vector with components « x », « y » and « z », that is, the imaginary part of the Quaternion « 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 better understand the concept, we will perform the following exercise: We will 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)

Subsequently, we will apply the multiplication operation between 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, there are two important observations that we must consider 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 ». This property helps us avoid making mistakes in the future. Second, when examining the structure of the resulting Quaternion from the multiplication,

we can notice that the scalar and vector parts correspond to operations we have already performed with vectors. This allows for a relatively straightforward implementation in C#.

$$q_1q_2 = s_1s_2 - v_1 \cdot v_2 + s_1v_2 + s_2v_1 + v_1 \times v_2$$

```
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 we can see in the « ScalarVector » method in Figure 3.1.h, two float variables « **s1** » and « **s2** » has 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, which encapsulates each component « **x** », « **y** » « **z** » and « w » of a Quaternion.

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

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

(3.1.i)

Which is the same as,

$$\overline{\boldsymbol{q}} = \boldsymbol{w} - \boldsymbol{x}\boldsymbol{i} - \boldsymbol{y}\boldsymbol{j} - \boldsymbol{z}\boldsymbol{k}$$

(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's as if we placed a mirror perpendicular to the vector and observed its reflected image. This property makes conjugation an operation that allows us to naturally obtain the reflection of a particular vector. The conjugation of a Quaternion also provides us with a simple way to obtain its absolute value, which can be expressed as follows:

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

(3.1.k)

Which is the same to say,

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

(3.1.l)

There are several reasons why we need to conjugate a Quaternion. For instance, when performing interpolation, inverting a Quaternion, or carrying out a rotation. In our case, it will be necessary to perform conjugation 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)

If we pay attention to Figure 3.1.m, we will notice that the « **w** » component remains positive, while each vector component becomes negative.

Given the nature of the previous explanation, we will now represent quaternions as points in a three-dimensional space, and for that, we will once again focus on Figure 3.1.a. However, this time we will completely ignore the real part of the Quaternion.

$$\boldsymbol{p} = \boldsymbol{xi} + \boldsymbol{yj} + \boldsymbol{zk}$$

(3.1.n)

As we can see in Figure 3.1.n, we can express a point in three-dimensional space in a more concise and direct way by using only the imaginary components. Now, when we talk about rotations in space, it will be essential to define an angle and axis. It's worth noting that the rotation angle is measured in radians, while the rotation axis is defined by a unit vector in space.

Let's 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 « u_x », « u_y » and « u_z » components are the unit vectors that define 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 we have the rotation Quaternion, we can apply it to a point in space. The transformation rule is as follows:

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

(3.1.q)

Following the Figure 3.1.q, we can easily deduce that the variable « $\boldsymbol{q}$ » corresponds to the rotation Quaternion, while « $\overline{\boldsymbol{q}}$ » refers to its conjugate. On the other hand, « $\boldsymbol{p}$ » corresponds to the point we want to rotate. How can we visualize this? To do so, let's perform the following exercise: We will start by placing a point or vertex « $\boldsymbol{p}$ » in space.

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

(3.1.r)

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

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

(3.1.s)

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

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

(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's 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, we will see in detail its implementation in C#.

Now, we just need to rotate the initial point. To do that, we need to 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 to say,

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

(3.1.x)

Therefore,

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

(3.1.y)

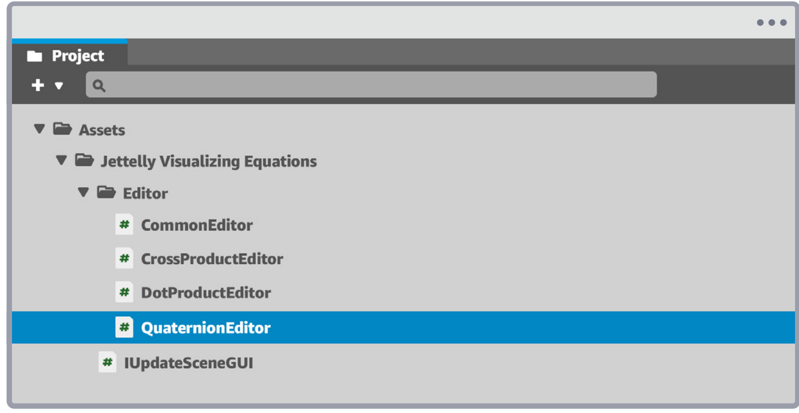
From the previous figure, « p' » refers to a new position of a point rotated 45 degrees around the « Y » axis in three-dimensional space.

3.2. Developing a tool in Unity.

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

The process described in this chapter does not differ from the one explained in previous chapters. Therefore, once again, we will make use of the « **`UnityEditor`** » dependency and extend our script from « **`CommonEditor`** » since the latter contains the fundamental « **`EditorWindow`** » class for creating this type of tool.

To start this process, we will create a new script in our project called « **`QuaternionEditor`** ». It is important to remember that for it to work correctly, we must place this script in our Editor folder, which we created earlier in our Unity project.



(3.2.a)

This time, it will not only be necessary to implement the functions and methods that enable the generation of the tool, but we will also have to create a « **`struct`** » object type to encapsulate the various properties that a Quaternion offers. To begin development, we will define the window that we will show from the Tools menu in Unity, which we will conveniently call "Quaternions."

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

In the previous code, if we look at the line of code number 4, we can see that the « **`IUpdateSceneGUI`** » interface has been implemented. This is mainly because we will use the « **`SceneGUI`** » method again to project a list of points that, together, will form a three-dimensional cube.

We will continue by declaring a global list of vectors where we will store each cube's point or vertex. Later, we will 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
```

It will be necessary to include the « **System.Collections.Generic** » dependency when working with list of type « **List** ». As seen in the previous code, this dependency has been included in the line code number 1. Later, the list of vertices is declared in the line number 7. It is important to mention that, for educational purposes, a list is being used in the current implementation. However, for consistency, considering that no more points will be added to the ones already defined, we could easily use a constant list of three-dimensional vectors.

The eight vertices included in the cube have been initialized in the code lines 17 to 19. However, it will be essential to generate gizmos for each of them in order to visually represent them in the Scene view. We will do this by iterating each vertex within a « **for** » loop and using the « **Handles.SphereHandleCap** » function for their visual representation.

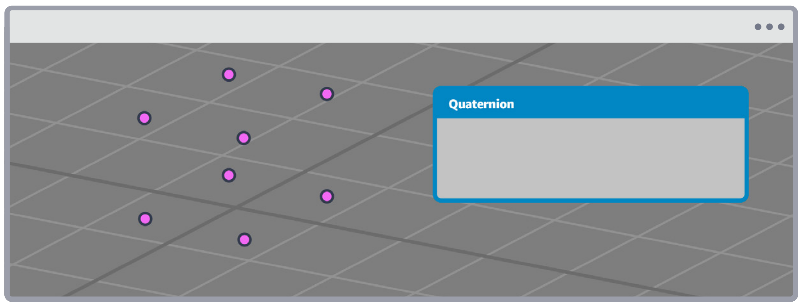
```
15    public void SceneGUI(SceneView view)
16    {
17        vertices = new List<Vector3>
18        {
19            new Vector3(-0.5f,  0.5f,  0.5f),
20            new Vector3( 0.5f,  0.5f,  0.5f),
21            new Vector3( 0.5f, -0.5f,  0.5f),
22            new Vector3(-0.5f, -0.5f,  0.5f),
23            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 we pay attention to the lines of code 29 to 32, we can see that a loop has been create within the « **SceneGUI** » method, which iterates over 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, we are returning the Quaternion identity, which means there is no rotation. This is because later in this book we will develop our own implementation of quaternions to rotate the cube's vertices according to a defined angle.

If we go back to Unity at this point, we won't see any graphical representation of the cube's points in the Scene view. This is because we are not running this process during the « **`OnGUI`** » method call. To fix this, we simply need to subscribe to the « **`SceneView.duringSceneGui`** » function to receive a callback. We will implement this subscription in both 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      }
```

From the previous code, we can see the implementation mentioned earlier in the code lines 17 and 22. If we go back to Unity, we will find our new "Quaternion" window within the Tool menu, which projects the eight vertices of the cube.

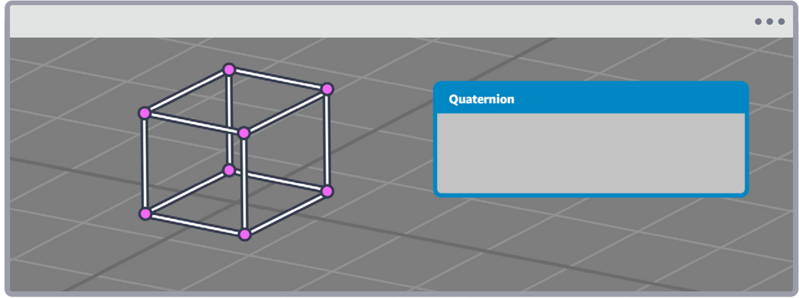


(3.2.b)

One action we can take in out tool is to draw a line between each vertex to enhance cube's projection in the Scene view. To achieve it, we use the « **`Handles.DrawAAPolyLine`** » function, which allows us to draw lines between points in three-dimensional space. However, its implementation in this case presents a challenge: if we want to connect all the vertices without repetition, it is necessary to group the different point combinations into a list of integer values. One way to approach this process is by 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    }
```

If we focus on the line number 44 in the « **SceneGUI** » method, we can see that 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 aim is to generate a graphical representation by drawing lines that connect pairs of cube points, thereby improving its visualization. Furthermore, in line 60, a « **for** » loop has been included that iterates through the index list and uses the « **DrawAAPolyLine** » function to draw the corresponding lines. Upon examining the Scene view once again, we will observe the connections between each vertex of the cube.



(3.1.c)

Up to this point, we have focused on projecting points in space, mainly to visualize the shape that we will later rotate. Now, we will put our logical thinking into practice by incorporating all the necessary properties of a Quaternion. To do so, we will start by declaring two new global variables:

- A floating-point value that will represent the rotation angle.
- A three-dimensional vector that will 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() …
```

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

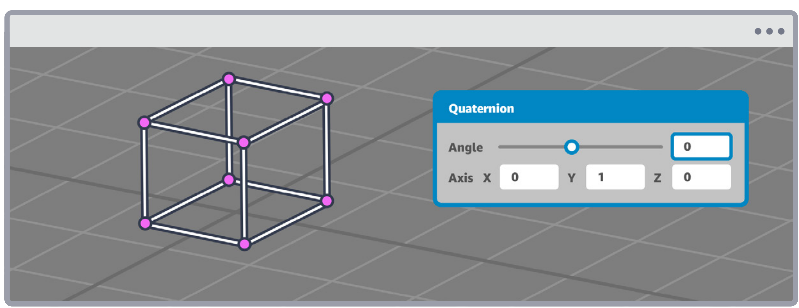
Next, we will proceed to initialize the variables in the « **OnEnable** » method and then expose them in the Quaternion window using the « **OnGUI** » method.

```
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    }
30
```

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

```
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 we go back to Unity, we will see that the mentioned properties have been correctly included in our tool.



(3.2.d)

Up to this point, we can consider that the initial process has concluded. The cube has been projected in the Scene window, and its vertices remain constant. Next, we will proceed to implement the necessary equations to rotate the vertices, following the definition of a Quaternion. For practical reasons, within the same script we are working on, we will declare a structure called « **HQuaternion** » to avoid conflicts with 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     }
```

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

As we saw 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  }
```

As we already know, to rotate a vertex using quaternions, we need to follow the equation mentioned in Figure 3.1.q from the previous section. Therefore, we will start by defining a rotation Quaternion following the operation detailed in Figure 3.1.o. To achieve this, we will 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   }
```

If we look at the line code 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.

Continuing with the process, it is necessary to perform the conjugation of the rotation Quaternion, i.e., inverting its vector component. To achieve this, we will 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   }
```

If we pay attention to line 121, we can see that the mentioned method returns a new « **HQuaternion** », following the equation presented in Figure 3.1.i from the previous section. Now, all that's left is to perform the necessary multiplications. To achieve it, we will 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, we can see that a new static method called « **Multiplication** » has been declared, which takes two arguments: two quaternions. Within its scope, the operation detailed in Figure 3.1.g from the previous section is implemented, which illustrates the multiplication of two quaternions.

Up to this point, we have defined several points 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, we will declare a new static method called « **Rotate** ». This one will take the position of the cube points, three rotation axes, and an angle as arguments. Why us is the method public? Because we will use it later within the « **SceneGUI** » method to rate 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   }
149
```

Following the equation presented in Figure 3.1.q, if we look at line number 140, we can see that 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, which obtains 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, which id obtained by multiplying « **q** » and « **p** » and then multiplying it by the conjugate of « **q** ».

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

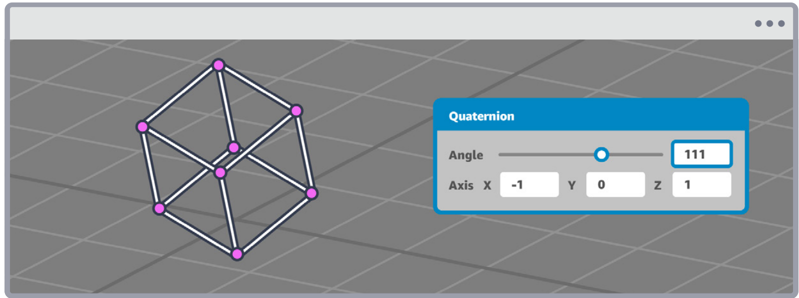
The only thing left is to apply the rotation to the vertices themselves. To do this, we need to go to « **SceneGUI** » method and call the « **Rotate** » method for each vertex defined in the cube.

```
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
```

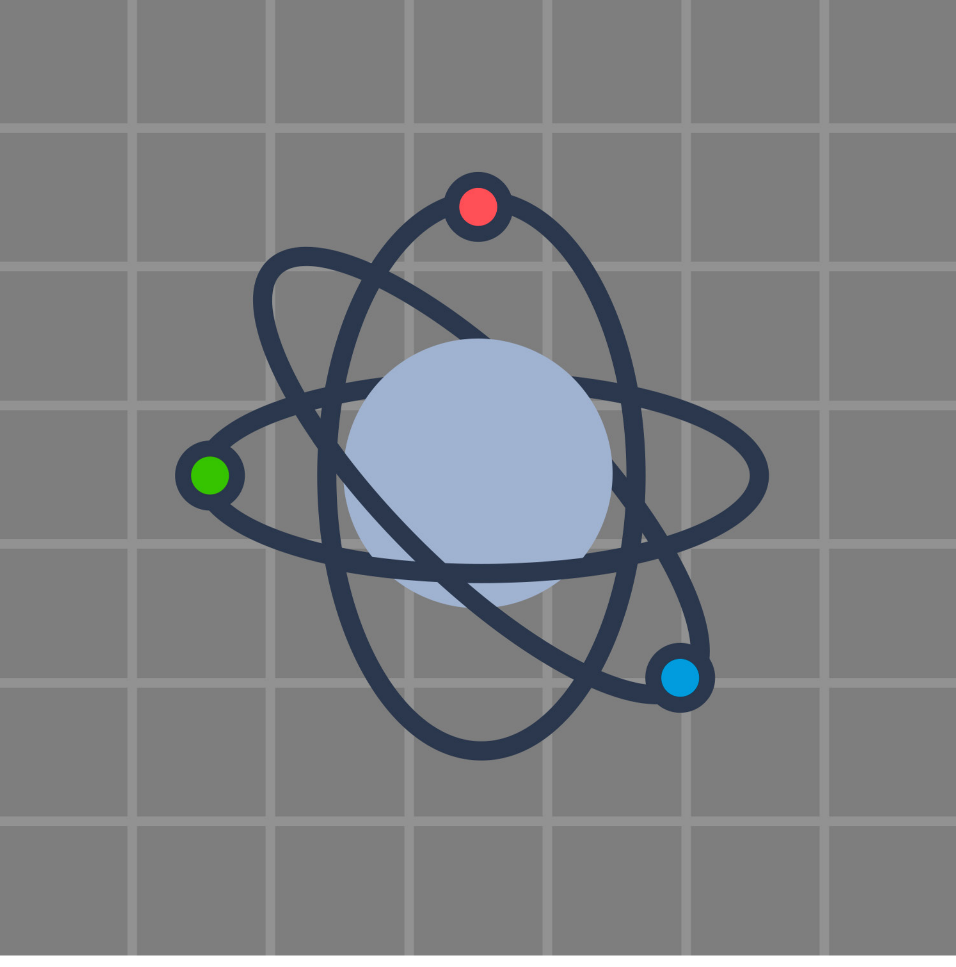
If we look at line number 70, we can see that we have used the « **Rotate** » method to change the position of each vertex defined in the vector list. For logical reasons, this process is carried out within the « **for** » loop, before drawing the spheres that define the graphical position or each vertex.

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

If we go back to Unity and modify the angle value, we will se that the vertices rotate according to the orientation defined in the axes.



(3.2.e)



Chapter 4.
Rotation matrices and Euler angles.

4.1. Introduction to the function.

In the previous chapter, we reviewed quaternions when applying a rotation to a cube composed of eight previously defined vertices. It begs the question: are there other ways to rate vertices or points in space? Depending on the projects needs we are developing, rotations using matrices can prove to be quite useful, specially when applied in computer graphics or in conversions from Euler angles to quaternions.

Rotation matrices are mathematical tools that enable precise and efficient rotations in tree-dimensional space. These matrices serve as numerical representations of transformations that rotate an object around a point or axis in space. Now, let's focus on the following matrix to understand its definition.

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

(4.1.a)

At first glance, Figure 4.1.a might seem complicated to grasp. However, when considering its definition, we can infer that it is a two-dimensional rotation matrix. This matrix allows for transforming the position of a point by an angle measured in radians. How is this achieved? To answer that, we need to focus on the trigonometric function « sin » and « cos », which, as we know, are used in mathematics to relate angles to the ratios of the sides of a right triangle.

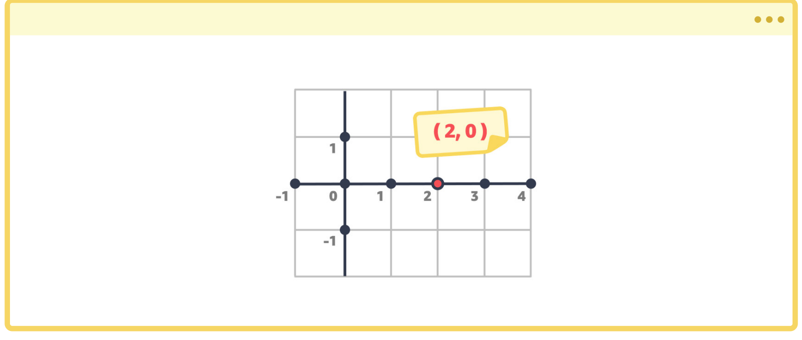
Considering that « θ » (theta) represents a defined angle, we can assume that by multiplying the matrix from Figure 4.1.a by a two-dimensional point « p » located in the plane, that point will change its position. The result of this transformation will be « p' », which is the designation for the point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

We will perform the following exercise to understand the concept: Let's consider a point « $\boldsymbol{p}$ » in two-dimensional space with the following coordinates,

$$\boldsymbol{p} = (\mathbf{2}, \mathbf{0})$$

(4.1.b)

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



(4.1.c)

Assuming we want to rotate point « $\boldsymbol{p}$ » 45° degrees counterclockwise from the previous figure, it will be necessary to take into consideration both the rotation axis of the point and the measurement system we use for the actual rotation. By default, all matrix rotations are carried out in radians, so it will be necessary to convert the angle to radians before applying the matrix transformation. To do this, we will pay attention to the following transformation rule:

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

(4.1.d)

Where « Ω » (omega) corresponds to the angle given in degrees. For our particular case, we will perform the transformation shown in Figure 4.1.d, considering that « Ω » is equal to 45°.

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

(4.1.e)

Therefore,

$$\theta = 0.78539816$$

(4.1.f)

Once we have obtained the rotation value in degrees, we simply need to apply the value from Figure 4.1.f directly to the matrix presented in Figure 4.1.a. For convenience, we will only include the first three decimal places in the equation, resulting in the following:

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

(4.1.g)

Once we've solved for the angle, we would need to multiply the rotation matrix by the point previously defined in Figure 4.1.b. How would we do this? It's essential to consider that the matrix we are using as an example has two rows and two columns. Since matrices can be multiplied when the number of columns in the first matrix is equal to the number of rows in the second matrix, we'll need to view the vector « $\boldsymbol{p}$ » as a matrix with two rows and one column, like this:

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

(4.1.h)

Considering what was mentioned earlier, we can perform the multiplication as follows:

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

(4.1.i)

Where « $\boldsymbol{p'}$ » corresponds to the new point; the one obtained after multiplying the rotation matrix « $\boldsymbol{r}$ » by the point « $\boldsymbol{p}$ ». So, the question arises: what would be the order of multiplication for these two objects? Considering that we are calculating the position of a new point on a Cartesian plane, we will need the values of the « $\boldsymbol{x}$ » and « $\boldsymbol{y}$ » coordinates of point « $\boldsymbol{p}$ ». Therefore, we will need to visualize, through variables and indices, the values of the matrix defined earlier. This way, we can better understand the order of multiplication in the operation we are going to perform.

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

(4.1.j)

If we expand the matrix product shown earlier, we will 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,

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

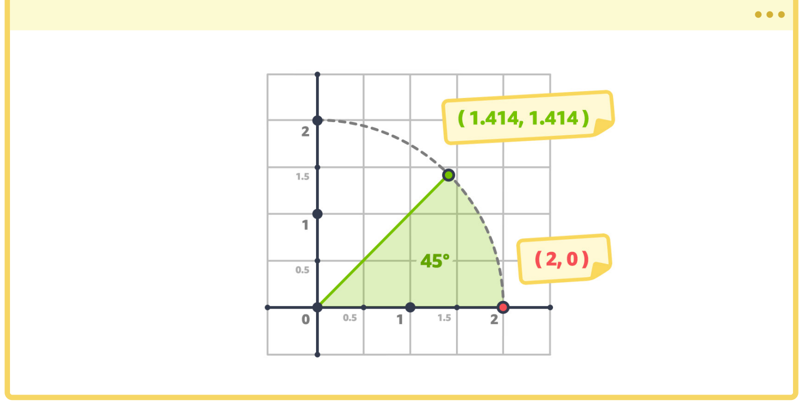
(4.1.l)

When we perform the corresponding arithmetic operations, we find that the new coordinates for point « p' » are:

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

(4.1.m)

If we pay attention to Figure 4.1.k, we will notice that we are performing the same operation that we did in the first chapter of this book when we calculated the Dot Product between two vectors, with the difference that in this case, we are applying it to different coordinates.



(4.1.n)

The code implementation of a rotation matrix will depend on the operation's specific needs or results you want to achieve. However, you could develop it as follows to achieve the equality 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 you can see in the previous figure, the « RotationMatrix2D » method returns the multiplication of a four-by-four-dimensional matrix denoted as « **m** » by a two-dimensional vector « **p** ». The latter refers to the initial position of the point you want to rotate. It's worth noting that in C#, there are no two-by-two-dimensional matrices. For this reason, a four-by-four-dimensional matrix has been implemented, and only four indices have been used to store the values in the list.

This type of matrices is commonly used in computer graphics to create interactive visual effects through shaders. With them, you can rotate the UV coordinates of a geometric shape. For example, the following method represents the implementation of the « **Rotate** » node in degrees, included in Shader Graph.

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

Considering that a two-dimensional matrix contains only one rotation axis, in this case, the « Z » axis, the question arises as to what would happen in the context of a three-dimensional matrix. When we refer to 3D rotation matrices, we speak to matrices with three rows and three columns. Each element in this matrix represents how the rotation around a previously defined axis will influence each object coordinate.

Creating a 3D rotation matrix depends on the axis around which you want to perform the rotation and the angle of rotation you intend to apply. For example, in a rotation around the « X » axis, the rotation matrix will adopt a specific structure that will affect the « Y » and « Z » coordinates of the object. The same logic extends to rotations around the « Y » and « Z » axes.

Let's focus our attention on the following matrices to understand this concept, especially in relation to the rotation setup for the « $\boldsymbol{Z}$ » axis.

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

(4.1.p)

Rotation setup for the « $\boldsymbol{X}$ » axis:

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

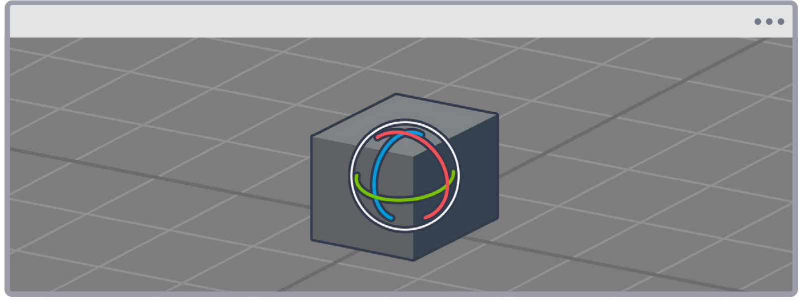
(4.1.q)

Rotation setup for the « $\boldsymbol{Y}$ » axis:

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

(4.1.r)

As you can see in the previous figures, the matrices have a variation in their configuration depending on the spatial axis you want to use, which is quite practical because it allows you to orient your object in a specific direction defined by three angles, three different values.



(4.1.s)

From such behavior arises the question: how to achieve the rotation of an object with three different angles? To answer this, we must delve into Euler angles, a common way to represent orientations in three-dimensional space. Euler angles consist of a set of three angular values used to describe a sequence of rotations. These angles are denoted as follows:

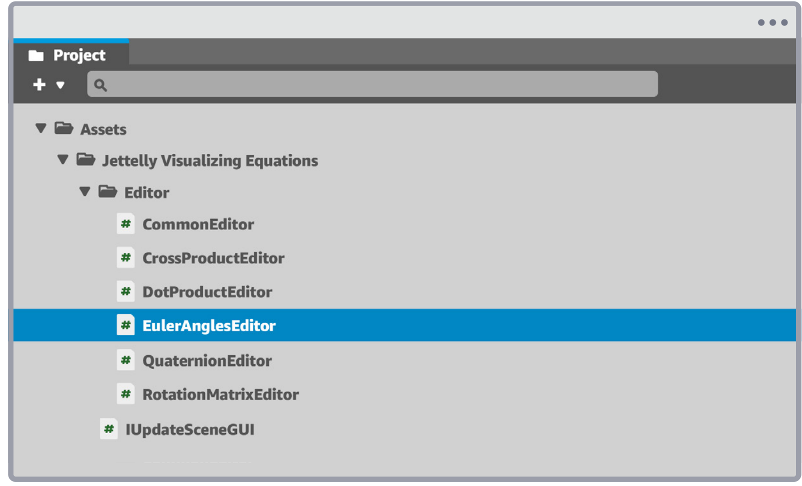
- « ψ » (Psi) for the « Z » axis.
- « θ » (Theta) for the « Y » axis.
- « ϕ » (Phi) for the « X » axis.

One notable aspect is the order in which these angles are multiplied, as it directly affects the final orientation of the object. In other words, depending on the convention we choose, we will obtain different rotation sequences, such as « ZYX », « XYZ », « YZX », among others. It's essential to keep in mind that Euler angles can lead to singularity problems, causing what is known as "gimbal lock". For this reason, it is advisable to work with quaternions in more advanced applications as they offer greater flexibility in handling rotations.

4.2. Developing a tool in Unity.

Once again, we will delve into the creation of a tool that facilitates the visualization of Euler angles. This way, we will better understand the phenomenon known as "gimbal lock". To accomplish this, we will proceed to implement the matrices that were mentioned in the previous section.

To start this unit, let's create a new script in our project named « **`EulerAnglesEditor`** » to streamline our process. Since, once again, it's an « **`EditorWindow`** » type script, it's essential to place it in the « **`Editor`** » folder that has been previously created in our project.



(4.2.a)

Once the file is open, it will be necessary to ensure that we inherit our script from « **`CommonEditor`** » to integrate the functionalities of « **`EditorWindow`** ». We will also incorporate the abstract class « **`IUpdateSceneGUI`** » since we will be using the « **`SceneGUI`** » method again.

As we already know, it will be crucial to declare a public static method to display the window of our tool in the Editor. For practical purposes, we will repeat part of the process we previously implemented when creating the « **ShowWindow** » function in our script.

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

Similarly, we will include the methods « **OnEnable** », « **OnDisable** », and « **OnGUI** ». In the « **OnGUI** » method, we will initialize our variables so that they can be displayed in the window we previously declared. On the other hand, in the first two methods, we will 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, we will define three new private methods that will be used in the definition of the matrices « ψ », « θ » and « ϕ ». Following established conventions, we will use the names « **`GetYaw`** », « **`GetPitch`** », and « **`GetRoll`** », where each of them will determine a rotation angle for the vertices of an object in three-dimensional space.

- « **`GetYaw`** » (ψ) will perform the rotation of vertices around the « Z » axis.
- « **`GetRoll`** » (θ) will perform the rotation of vertices around the « X » axis.
- « **`GetPitch`** » (ϕ) will carry out the rotation of vertices around the « Y » axis.

The task of defining the matrices could be approached in various ways, such as using multiple lists of values. However, this time, we will opt for using the « **`Matrix4x4`** » object type, which is available in UnityEngine. As we 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 we can see in the example above, each value has been incorporated into the respective rows and columns, following the inherent structure of each matrix. It's relevant to note that since no angle conversion has been performed (the argument), it will be calculated in radians by default. Therefore, later on, it will be necessary to apply the function presented in Figure 4.1.d from the previous section to perform the rotations in radians.

We will continue developing our tool by declaring and initializing three floating-point values. These values will be used in the definition of the 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
```

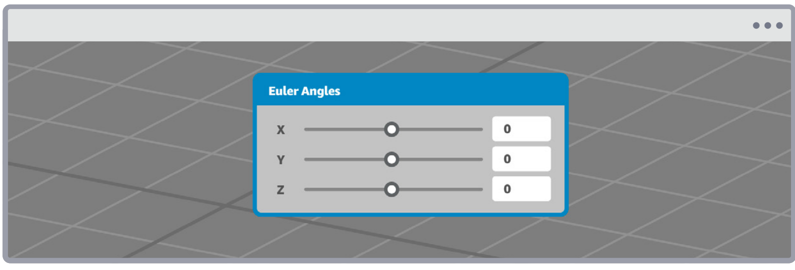
If we focus our attention on lines of code 7, 8, and 9, we'll notice that our floating-point variables are limited to a range from -180 to 180 degrees. What's the reason behind this choice? More specifically, why this range?

While this exercise could be tackled using a three-dimensional vector « **Vector3** », for practical reasons, we chose to simplify it by rotating angles within a total range of 360 degrees.

Moving forward, we will continue with the initialization and projection phase of our variables into the tool's window itself. We will intervene in both the « **OnGUI** » and « **OnEnable** » methods by executing the following operation to achieve this.

```
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    }
```

As we are already aware, the « **DrawBlockGUI** » method was previously defined within the « **CommonEditor** » class. Its main purpose is to draw information blocks on our tool's graphical user interface (GUI). When we return to Unity, we will see that our GUI's rotation angles are now present.



(4.2.b)

Up to this point, the only thing left is to draw some vertices in the Scene view to visualize the different rotations and Euler angles. We will declare three lists « **`List`** », one for each 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
```

If we focus our attention on the example above (lines of code 16, 17, and 18), we can see the independent declaration of lists for each axis. To illustrate this concept, we will incorporate points that, when connected, will form polygons to visualize 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 will embody a rotation angle, maintaining Unity's default orientation: « XYZ ».

Additionally, on line 19, an additional list named «arrow» has been added. This list will be used to draw an arrow in the scene, allowing for a more precise visualization of the current orientation of the angles.

We will continue by initializing our lists within the « **SceneGUI** » method. To do this, we will include eight points (vertices) within each of our lists, 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            Handles.color = Color.green;
78            Handles.SphereHandleCap(0, circleY[i], Quaternion.identity,
              0.05f, EventType.Repaint);
79        }
80
```

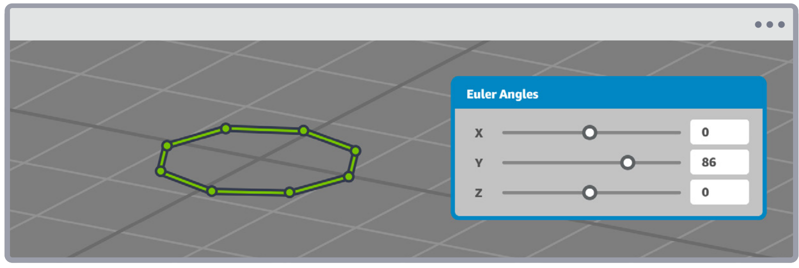
Continued on the next page

```
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 can be seen in the figure above, specifically in lines of code 62 to 69, we have incorporated a total of eight points in the « **circleY** » list. This list is designed to visually represent rotation around our tool's « Y » axis. Then, in the first « **for** » loop (lines 74 to 79), we draw a sphere using the « **SphereHandleCap** » function for each point in the list. These spheres have the ability to rotate around a reference point. This is achieved because, as seen in line of code 76, « **circleY** » is defined as the multiplication of the « **GetPitch** » matrix by each point in the list.

It's important to mention that the rotation we apply is done in radians. This is because « **GetPitch** » takes « **degreeY** » as an argument, which is equal to the angle multiplied by the result of dividing « **PI** » by 180f (line 72).

Within the second « **for** » loop (lines 81 to 85), you can see the creation of lines connecting each point with the next based on the intersections in the previously initialized list. This process aims to visually display the sides of the polygon that provides a visual approximation of the circle. This procedure will help us better understand the tool. If we save the changes and return to Unity, we will notice that the « Y » axis has been added to our scene. Additionally, we can adjust its orientation by modifying the « **y** » property value of our tool.



(4.2.c)

Next, we will return to our 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.
- Iterating through each of them to draw a sphere at the position of each point.
- Iterating through 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        };
```

Continued on the next page

```
83
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
```

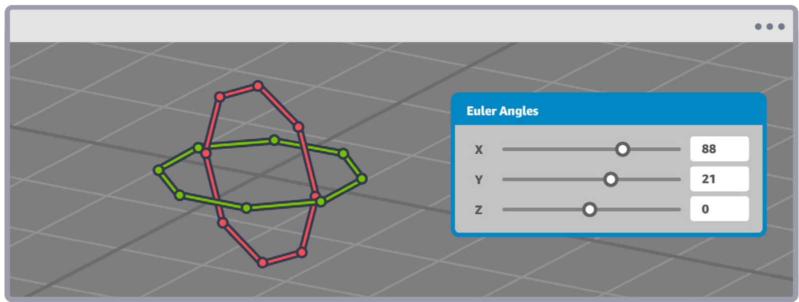
In the example above, if we pay attention to lines of code 74 to 81, we have again included previously defined points in the « **circleX** » list to generate a polygon that, in this case, helps us visualize the rotation angle around the « X » axis. It's worth noting that each point has been multiplied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

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

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

This happens due to the peculiar interaction of Euler angles. When we rotate around the « Y » axis, the « X » axis also rotates as if they are linked together. The same type of interaction occurs in relation to the « Z » axis. These particular connections create complications when working with Euler angles because one rotation axis can lose relevance in certain situations, leading to what we know as "gimbal lock." This effect can make it challenging to accurately represent orientation in space.

When we return to Unity, we will notice that the « X » axis is now present in the Scene view. Additionally, we have the ability to adjust its orientation using the **« `x` »** property in our tool.



(4.2.d)

We will continue initializing our last matrix, which corresponds to the « Z » axis. To do this, we will once again include 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
                circleX[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
106             circleX[i]);
107             Handles.color = Color.red;
```

Continued on the next page

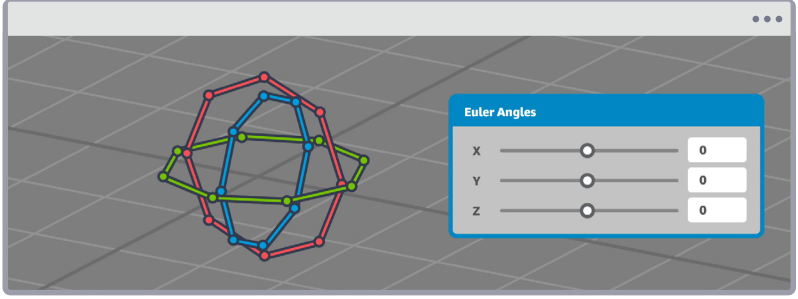
```
108             Handles.SphereHandleCap(0, circleX[i], Quaternion.identity,
                0.05f, EventType.Repaint);
109
110             circleZ[i] = GetPitch(degreeY) * (GetRoll(degreeX) *
                (GetYaw(degreeZ) * circleZ[i]));
111             Handles.color = Color.cyan;
112             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         }
127     }
128
```

As we can see in the previous operation (lines 86 to 93), we have repeated the same procedure once more, but this time, it is applied to the « Z » axis. In other words, we are initializing each point in the « **`circleZ`** » list and then iterating through these points in the two corresponding « **`for`** » loops.

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

- First, we multiplied the « **`GetYaw`** » matrix by each point in the list.
- Then, we multiplied the previous result by the « **`GetRoll`** » matrix.
- Finally, we multiplied the entire 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. When we return to Unity, we can observe the different rotation axes with their respective angles and properties. In fact, if we configure the angles numerically as « **`90°, 0°, 0°`** » we will lose one rotation angle, leading to gimbal lock.



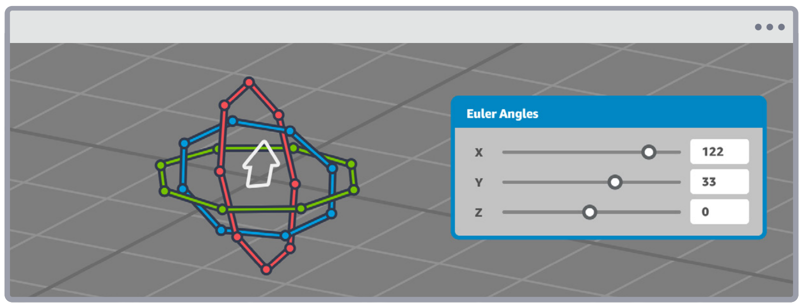
(4.2.e)

We will proceed with the initialization of 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`** ». We will use an arrow to visualize the orientation of the Euler angles. To carry out this process, we will 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 blocks that are important 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), we iterate through each point and perform the same process we used for the « Z » axis of our tool. In this case, we multiply each point by « **`GetYaw`** », then by the result obtained from that operation multiplied by « **`GetRoll`** », and finally by « **`GetPitch`** ». This provides us with the orientation of the Euler angles. To finish, we declare a new loop (lines 143 to 147) to draw a line connecting each point in the list.

When we return to Unity, we will be able to observe the complete functionality of the tool we have been developing in this chapter.



(4.2.f)

Glossary.

Function: A function is a mathematical relationship that assigns each element of a set, a domain, to an element in another set or codomain. If the relationship is one-to-one, meaning that for each element in the domain, there exists a unique element in the codomain, the function is bijective. It can be described by a rule that associates an input with an output.

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

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

Commutativity: In mathematics, it means that the order of an operation does not matter. For example, it is the same to say 1 + 2 as it is to say 2 + 1.

Summation: In mathematics, it is an operator that allows the representation of sums of many addends, including infinites. It is expressed with the Greek letter Σ sigma.

Variable: A variable is a symbol that represents a value that can change in an equation, formula, or program. It can take different values during the code execution.

Reference System: It corresponds to a set of spatial coordinates required to determine the position of a point in space.

Dot Product: Also known as "Scalar Product," it is a mathematical operation between two vectors that results in a scalar value.

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

Anticommutativity: A mathematical property in which 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, it can be equal to - (b * a).

Polygon: It 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: These are matrices used to represent rotational transformations in three-dimensional space. These matrices are used to change the orientation of three-dimensional 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. These angles 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 sense, 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: It 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: It is an inverse trigonometric function that returns the angle whose tangent is a given value.

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

Radian: It 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 sense.

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

Coordinate: A coordinate is a set of values that determines the position of a point in space, either 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: It is an ordered table of elements arranged in rows and columns, used in mathematics and programming to represent data and perform linear operations.

IntelliSense: It is a feature present in some development environments that provides automatic code suggestions, completing keywords, functions, and variable names as you type.

Euclidean Space: It is 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 known as Euler angles.

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

Absolute Value: The absolute value of a number is its magnitude without considering the sign, i.e., it is always a positive number.

Interpolation: Interpolation is 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 occurs when two of the three rotation axes align, reducing the freedom of movement to two dimensions.

Special thanks

Özkan melen | Đăng Trăn Hăi | Abraham Armas Cordero | Adam Carames | Adam Bennett | Adam Gibson | Adam Myhre | Adrian Cuneo | Adrian Devlinn | Adriano Valle | Adrien Kissenpfennig | Ahmet Usta | Alavuotunki Pekka Juhani | Alejandro Allende | Alejandro García Guillot | Alejandro Hidalgo Acuna | Alexander Ewetumo | Alexander Froelich | Alexander 'Gruni' Grunert | Alexandru Geana | Alexis Gonzalez | Alisia Martinez | Alyne Kelly Gois | Ana Perez Sierra | Andreas Zimmer | Andres | Andrew Fellows | Anna Kocer | Anton Sasinovich | Antonio Manzari | Arda Ozupek | Arkadiusz Kotarski | Aron Thompson | Arthur P V Salvador | artisZin | Artur Shapiro | Arturo Zahar Alcibia | Ash Curkpatrick | Ashton Cross | Atakan Talay | Athanasios Zagkliveris | Austin Lothman | Aviad Biton | Avinash B | Aziz Şekerdil | Baesungjin | Ben Finkelstein | Benjamin Bouffier | Benjamin Koder | Benjamin Russell | Benny Franco | Boehrer Magali Claire | Braden Currah | Brandon Friend | Bret bays | Brian Heinrich | Briceida Carillo | Bruno Costa | Bryce Paule | Cesar Augusto Fernandez Cano | Cesar Hector | Can Delibaş | Caner Özdemir | Chen Jingzhou | Cheng Bowen | Chepe | ChongYi | Christopher Suffern | Christopher W Cannon | Christos Tzastas | Claudiu Barsan-Pipu | Clemens Pfauser | Cody Wilson | Cole Andress | Connor Checkley | Damian Longman | Damian Turnbull | Daniel A Fisher | Daniel Garcia | Daniel Ruiz Leyva | Darina Koycheva | David A Holland | David Clabaugh | David Jumeau | david nieves | David Vessup | Davit Badalyan | Defrog | DegenerateWaste | Delano Igbinoba | Derek Brouwer | Dhaval Prajapati | Dina Khalil | Do Minh Triet | Douglas Kerr | Dragan Ignjatovic | Ebru Sena | Eduardo R iOcusRise | Edward Ro | Edwin J V Naranjo | Emiliano Guzman M. | Emir Furkan Tokkan | Endri Kastrati | EonTools | Eric Pattmon | Eric Rico | Eric W Nersesian | Erik Niese-Petersen | Esko Evtyukov | Etonix Pyro | Fahrul Gamemaker | Felipe R Silva | Florence Noe | Francisco Chagolla Angulo | Francisco J Estrada Salinas | Francisco Ortega | Franks Gonzalez | Frost Corp | gabriel gomez | Garet Thomas | Gareth Bourn | Gilberto B P Junior | GinderBird | Giselle Silva | Guilherme Nunes | H Kotze | H. Carrasco Pagnossi | Hamza

Mohammed Rangoonia | hatice nur efe | Hector Moscoso | Herleen Dualan | Herman Garling Coll | Hiroki Miyada | Ho Cheng-Yi | Hugo Barrandon | Hugo Delgado | Humberto Gamboa | Iain Pentelow | Ibrahim Yamaam | Ilyas Sadyrov | Invex | Iram Maximiliano Lopez Guerrero | ismael bernard | ivan cardenas | Ivan Imerovic | J. F. Echeverria Pinto | Jack Haehl | Jakob Rendon | Jakub Slaby | James Please | Jari | Jason Fotso-Puepi | Jason Peterson | Jean Ducellier | Jengy Matantsev | Jesse Maccabe | Jesus David Angarita | Jesus Hernandez Ortiz | Jhoshua Ampo | Jing Yi Chong | Joan Toh | Joao L F Amorim | Joel Freeman | John Fredy Espinosa | John Jones | John Reitze | Johnny M Roodt | Jonas Carvalho de Araujo | Jonathan Jesus Cantor Gonzalez | Jonathan Keuchkarian | Jonathan Morales | Jonathan Valderrama | Jordan Cox | Jordan Han | Jordi Moreno Lopez | Jose jimenez | Jose Lopez | Jose Miguel Casas Pagan | Jose Ramon Arias Gonzalez | Joseph P Denike | Joshua Baillargeon | Joshua D Horner | Juan Carlos Horta | Juan Hernandez | Juan Luis Moreno | Juan Manuel Gonzalez Garcia | Juan Muniain Otero | | Julia Caputa | Julius Fondem | Juris Savostijanovs | Jyoti Kamlakar Bhoir | Karim Lachaize | Kevin Willis | Kevin Zambrano | Kim Young Min | Kimkunbyo | Kristopher Cigic | Kyrylo Samoilenko | L Tijsma | Laise M Nogueira | Laith Hasan | Larry Fuhrmann | Lawrence Yip | Li Jingtian | Lim Donguk | Living Room Filmmaker | Loonatic | Lucas Ferreira | Lucas McDermott | Lucia Gambardella | Luis Francisco Roa Castro | Luis Urueta | Luong Huu Tinh | Maciej Nabialczyk | Madeline McDougall | Malik aune | Malik Aune | Manuel Uriel Olvera Castañeda | Manvendra Deora | Marc Hewitt | Marco Arjona | Marcos Wicket | Mark Steele | Markus Sebastian Bakken Storeide | Mateus S Pereira | Matt McMahon | Mattedickson | Matthew David Lee | Max Otto | Mb Net | Meera Sanghani | Melanie Tidler | Michael Clavan | Michael Guerrero | Michael I Randrup | Michael Ross | Michael Woo | Michelle Moreno Arveras | Miguel Angel Bulnes Echave | Mikalai Danilenka | Miles J Barksdale | Minh Dia | Miquel Campos | Miquel Ferrer Mas | Miss Oona R Tukia | Miss V Johnson | 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 |

Muhammad Azman | Munesadafumiki | Nathaniel Biddle | Nguyen Tuan Dat | Nicholas Jonathan Boyd | Nicholas M Stringer | Nicholas Routhier | Nicolas Alonso Acevedo Suzarte | Nicolas | Nicolas DeZubiria C | Ning Cai | Noelia Fernandez | Ogulcan Topsakal | Olivier Baron | Omgelsie | Online Kort | Osakpemwokan Alonge | Pablo Jose de Andres | Palita Panyadee | Pamisetty Ranganath | Paritta Kijmahan | Parsa Jamshidi | Parsue Choi | Patryk Brząkalik | Paul Killman | Peter Chen| | Pherawat Puttabucha | Pia Guehne | Przemysław George | Radoslaw Polasik | Ramiro Alurralde | Ranjit Menon | Raveen Rajadorai | Raza Butt | Razvan Luta | Reira Ota | Rene Melendez | Richard Wallace | Rihards Valters | Rinat Enikeev | Rizal Ardianto Saleh | Roberto Andres Estupinan Cuadrado| Robin Hoole | Robinson Enrique Rojas Rojas | Rodion Tabares | Rodrigo Moreira Abreu | Roy Rodenhaeuser | Rt3d | Ryosuke Motrgi | S Julien Gauthier | Saiel | Samuel Luce | Samuel Trudgian | Samuel Wilton | Sandro Ponticelli | Sankar | Sarah Young | Saurabh E Kachhap | Scott Benson | Sean McAllister | Sebastian Emanuelsson | Seth Crawford| Siren | Sirrena Holmes | Skodje | Sourav Chatterjee | Stefan Groenewoud | Steffen | Sujith R | Sunlight Technologies | Suthamon Hengrasmee | Takashi Nakamura | Takumi Motoike | Takumi Tsukada | Taylor L Ekena | Theodoros Doukoulos| | Thomas Foster | Thomas Guyamier | Thomas Surin | Thomas Wormann | Timo Fettweiß | Timur Ariman | Timur Ariman | Tinnaput | TiraniceD W | Todd Akita | TofuLemon | Tom Keen | Tomas Gayo Perin | Tore Waldow | Tyler Molz | Tyler Parker | Uladzislau Daroshka | Umut Çetin Sağdıçoğlu | UNIVERSAL | UnPySide | Valentin Boissel | Valentin Pantiukh | Victor H Cardona G | Victor Manuel Celis Padron | victor soler | Vinicius Ramires Leite | Virtuos Vietnam | Vitalii Shaposhnikov | Vong Pha | Wajeeh ul Hassan | Wang Qiang | Will Andersen | William Beltran | William C. Taylor | Wilmer Cedeno | Yasmin Shitrit | Ybrayym Dathudayev | Yordan Kalbanov | Youngmin Kim | Yousung Kim | Yuichi Matsuoka | Zachariah Abueg | Zachary Chung | Zhangbao

“Jettelly wishes you success in your professional career”.