Volume 1

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

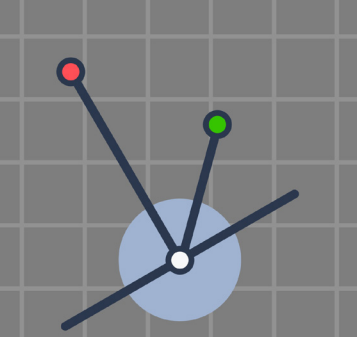
Designed by
Pablo Yeber

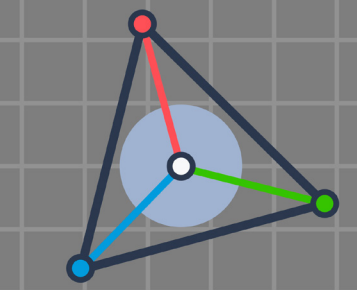

Jettelly

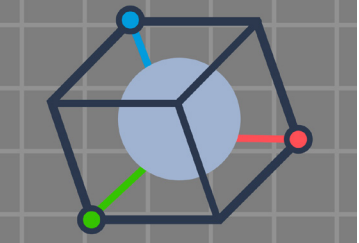
Visualizing Equations
, Essential Math for Game Devs.

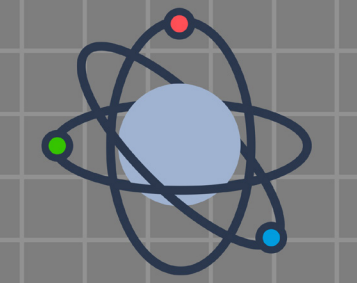
A visual explanation book that will help you understand essential mathematical concepts applicable to game development.

Visualizing Equations, - vol.1, version 0.0.6.
 www.jettelly.com
@jettelly.

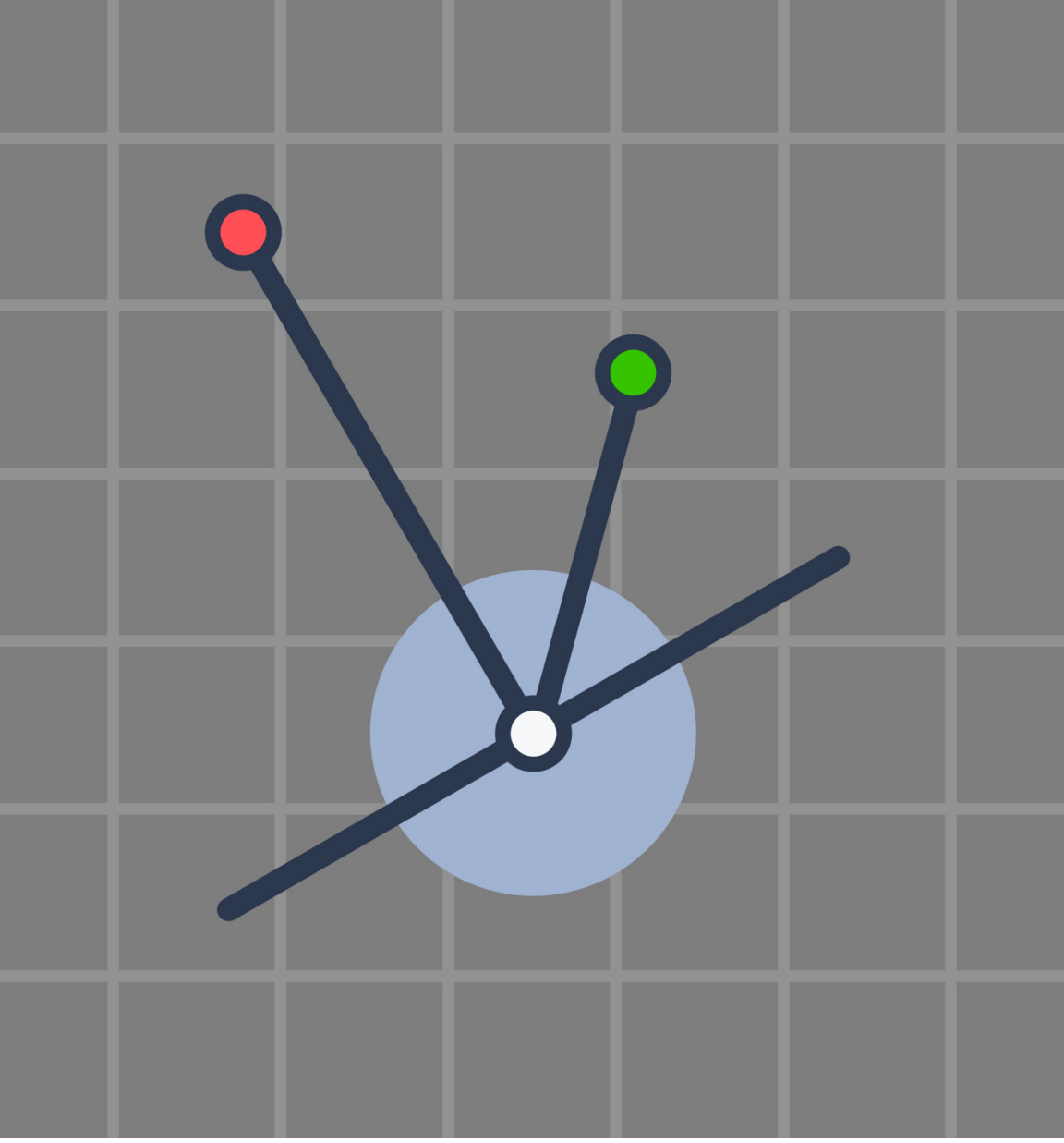








Content.



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)

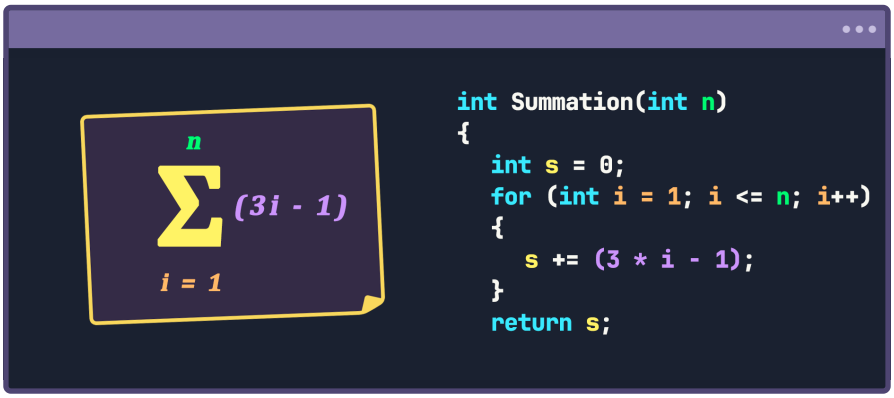
Can you identify which function or operation the above equation belongs to? It is quite common to find these types of equations in programming-oriented books. They are used to illustrate a function that yields a specific result.

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 « $\sum$ » (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 is worth remembering that a variable, as the name suggests, refers to a value that varies over time, such as a person's age. Age can be 1, 2, 3, and so on, but it will never be a constant value. Why is that? Because time marches on, unfortunate, doesn't it?

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



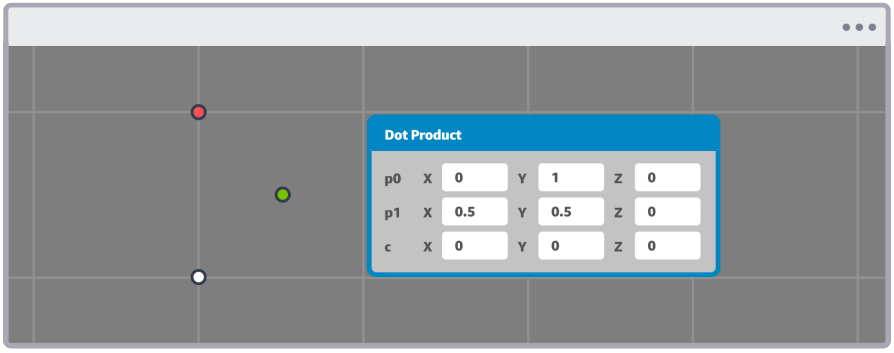
(1.1.i)

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 a vector projection « A » onto vector « B ». What does this mean? Imagine standing on a sunny day, with 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.

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

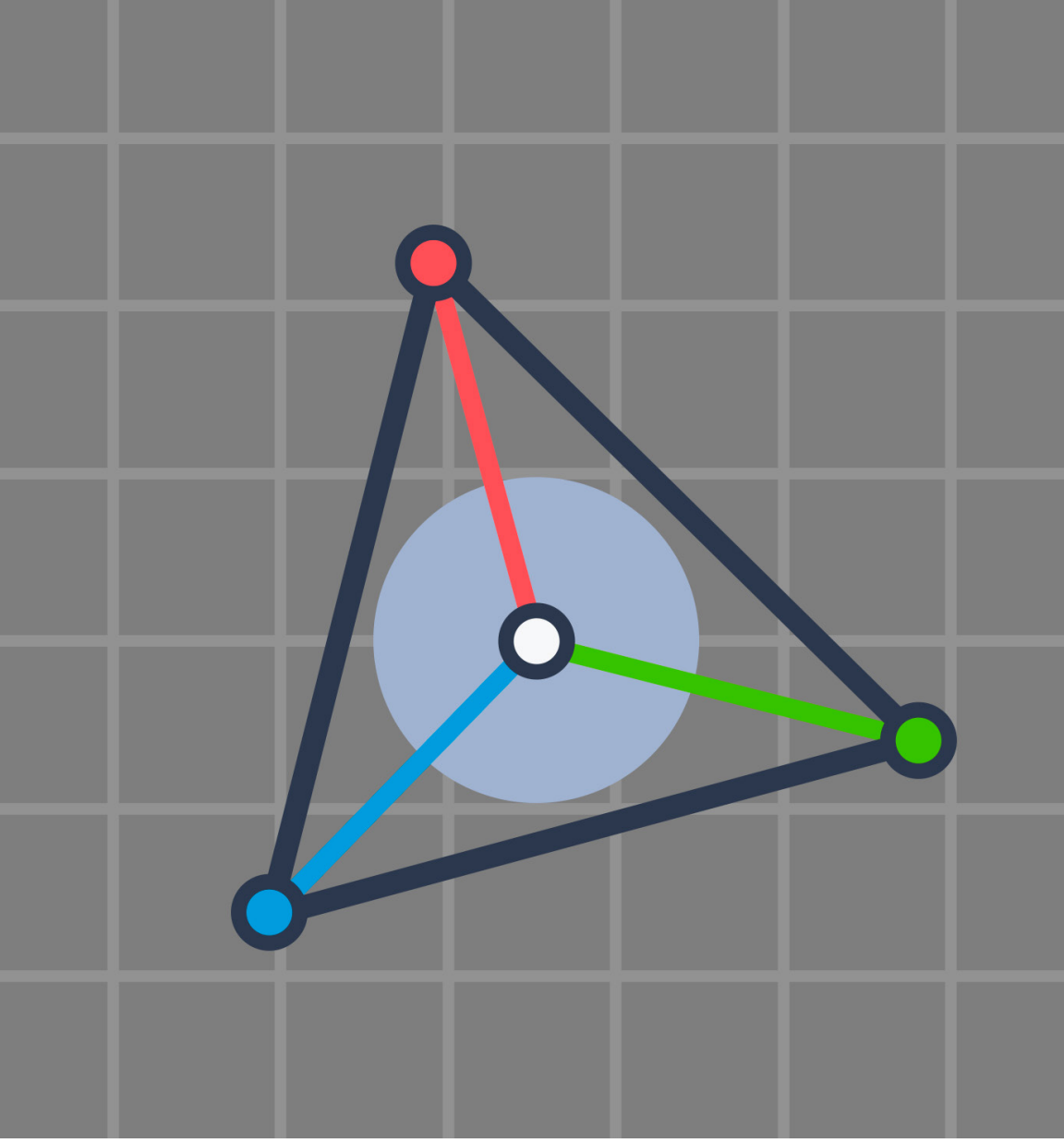


(1.2.d)

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

Up to this point, 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 behaviour of the Dot Product of two vectors. Therefore, we will continue implementing the method mentioned in Figure 1.1.k from the previous section.

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



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:

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

(2.1.a)

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

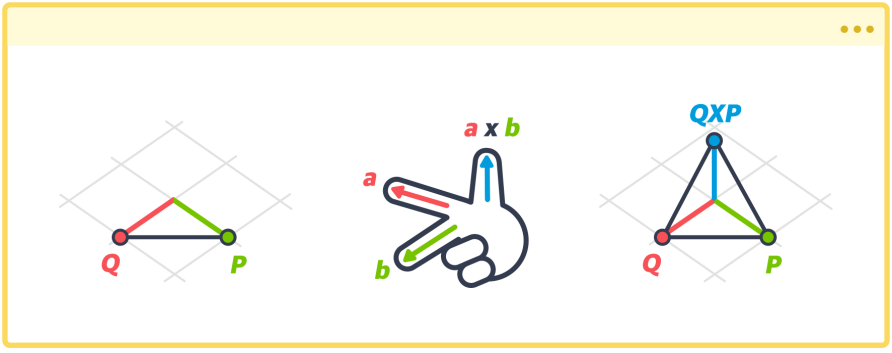
Since the Cross Product returns a three-dimensional vector, we can infer that the operations within the parentheses in Figure 2.1.a corresponds to the components of the new vector. In other words,

$$\begin{aligned}(P \times Q)_x &= \left(P_y Q_z - P_z Q_y\right) \\ (P \times Q)_y &= (P_z Q_x - P_x Q_z) \\ (P \times Q)_z &= (P_x Q_y - P_y Q_x)\end{aligned}$$

(2.1.b)

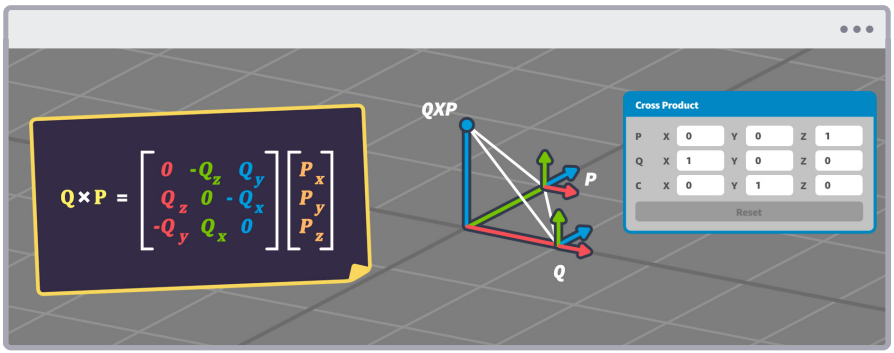
The resulting vector « C » has the particularity that it is perpendicular to the original vectors. Furthermore, its magnitude is related to the area of the parallelogram generated by the two vectors « A » and « B ». It is worth noting that its direction can be determined using the "right-hand rule," 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 applications, such as calculating the normal vector of a vertex or surface or even determining whether a polygon is concave or convex based on the direction of the resulting vector. This is beneficial in triangulations and other geometry problems.



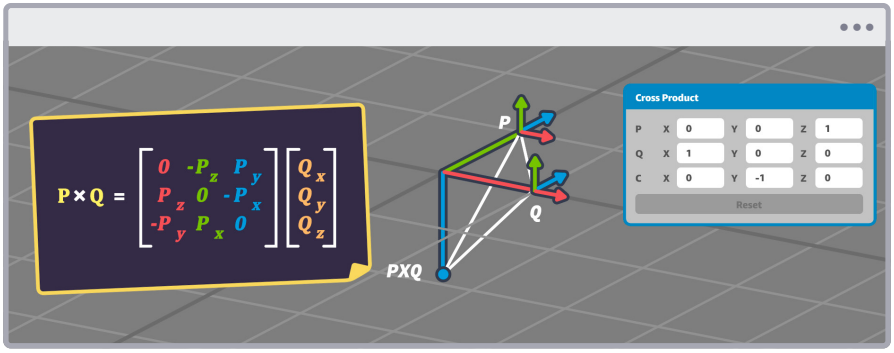
(2.1.g)

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.



(2.1.k)

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

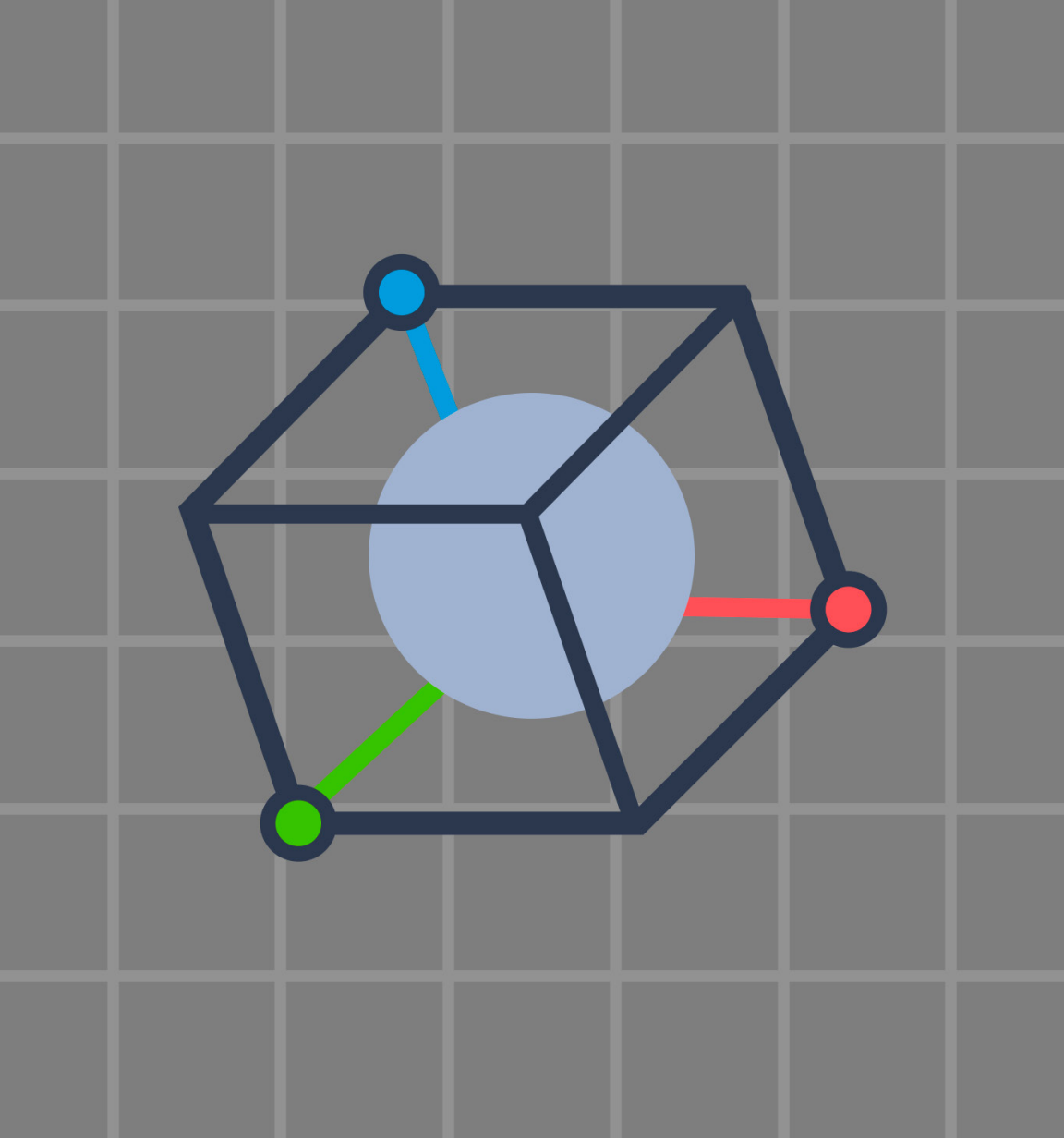


(2.1.l)

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, using the « **`UnityEditor`** » dependency and extending 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.



Chapter 3.

Quaternions.

3.1. Introduction to the function.

Quaternions are fundamental objects in any development software focused on 3D objects. Why is that? They allow us to generate vertex rotations, avoiding the phenomenon caused by the gimbal lock (also known as the 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 us pay attention to the following formula:

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

(3.1.a)

It is 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 these 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 establish a direct relationship with their corresponding axis:

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

(3.1.b)

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

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:

(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.

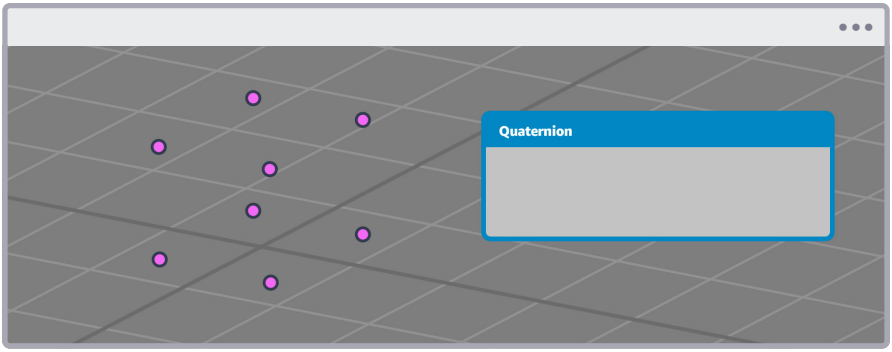
Given the nature of the previous explanation, we will now represent quaternions as points in a three-dimensional space, and for that, we will again focus on Figure 3.1.a. However, this time, we will completely ignore the real part of the Quaternion.

$$p = xi + yj + zk$$

(3.1.n)

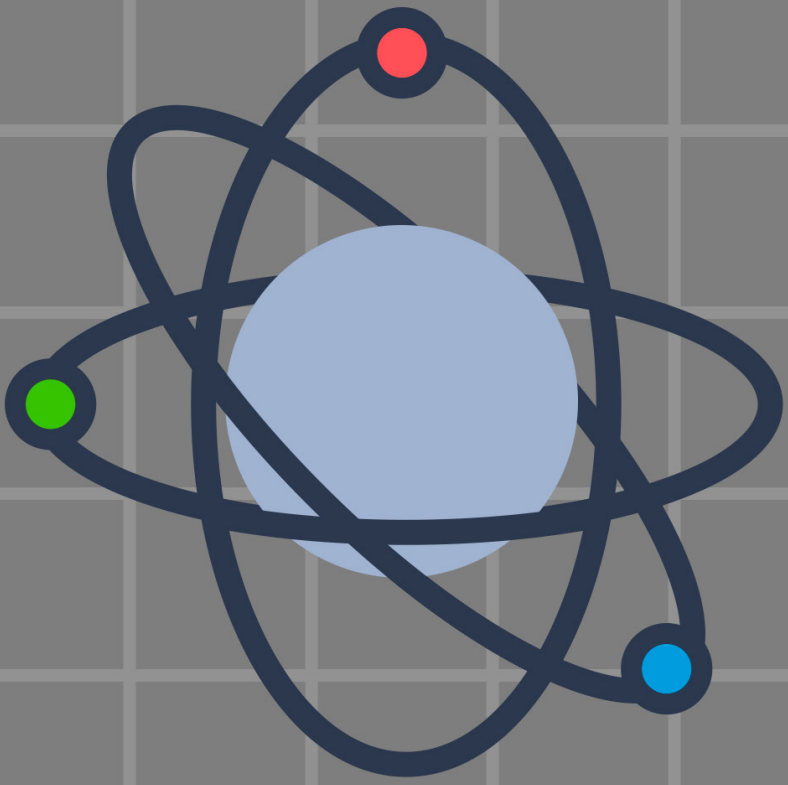
As shown in Figure 3.1.n, we can express a point in three-dimensional space more concise and direct way by using only the imaginary components. Now, when we talk about rotations in space, it will be essential to define an angle and axis. It is worth noting that the rotation angle is measured in radians,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

From the previous code, we can see the implementation mentioned earlier in 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 can carry out in our tool is to draw a line between each vertex to enhance the cube’s projection in the Scene view. In order 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, we must 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.



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 project's needs we are developing, rotations using matrices can prove to be helpful, mainly when applied in computer graphics or conversions from Euler angles to quaternions.

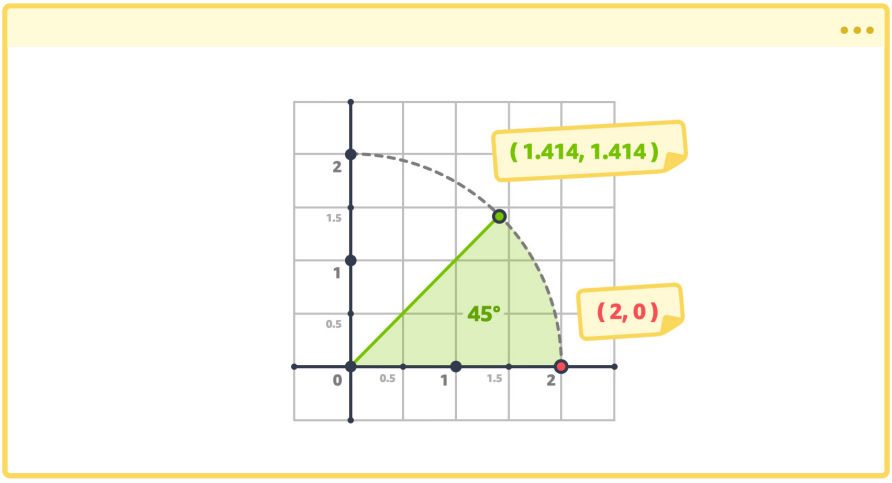
Rotation matrices are mathematical tools that enable precise and efficient rotations in three-dimensional space. These matrices serve as numerical representations of transformations that rotate an object around a point or axis in space. Now, let us focus on the following matrix to understand its definition.

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

(4.1.a)

At first glance, Figure 4.1.a might seem complicated to grasp. However, considering its definition, we can infer 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 functions « $\sin$ » 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.



(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 shown in the following Figure to achieve the equality in Figure 4.1.j.

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

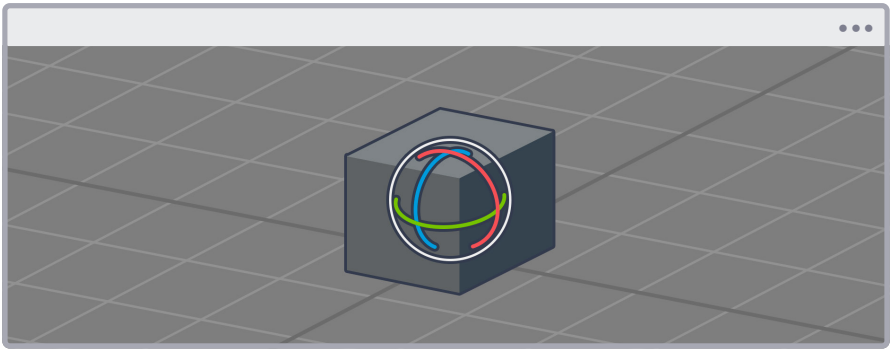
```
Vector2 RotationMatrix2D(Vector2 p,
float angle)
{
   float a = angle * (Mathf.PI / 180);
   Matrix4x4 m = new Matrix4x4();

   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 figures, the matrices vary in their configuration depending on the spatial axis you want to use. It is practical because it allows you to orient your object in a specific direction defined by three angles, meaning three different values.



(4.1.s)

From such behaviour 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 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 is essential to remember that Euler angles can lead to singularity problems, causing what is known as a “gimbal lock.” For this reason, it is advisable to work with quaternions in more advanced applications as they offer greater flexibility in handling rotations.

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.

Why not get the full book?

Visualizing Equations Vol. 1 book has a **5 star** rating
It has over **1K** readers so far!
Will you join us to learn about mathematical concepts?
With **+120 pages of information applicable to game development.**
Updates are free forever!

Buy on Gumroad

Jettelly Books!

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

Go to Books!