Math for 3D/Games Programmers

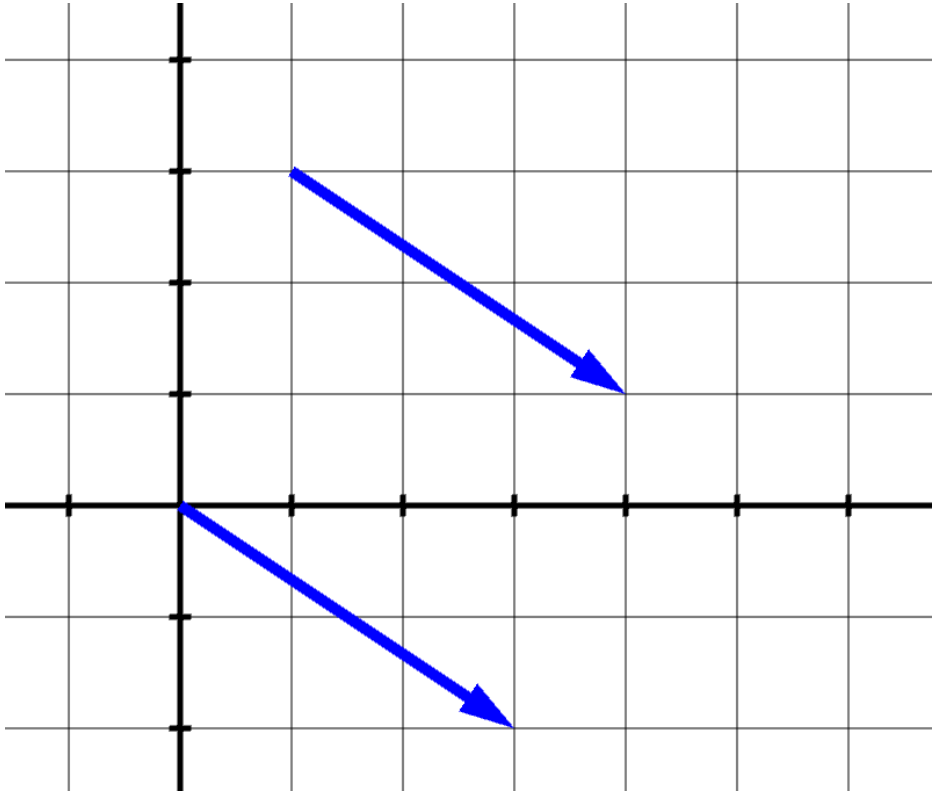
3. Vectors

Table of Contents

Introduction

- **Vector** is a pair (in 2D) of numbers, which tell a **direction** on a plane.
  In 3D it's a triplet
- An example vector (in 2D), pointing in a certain direction:

$$\vec{v} = [3, -2]$$



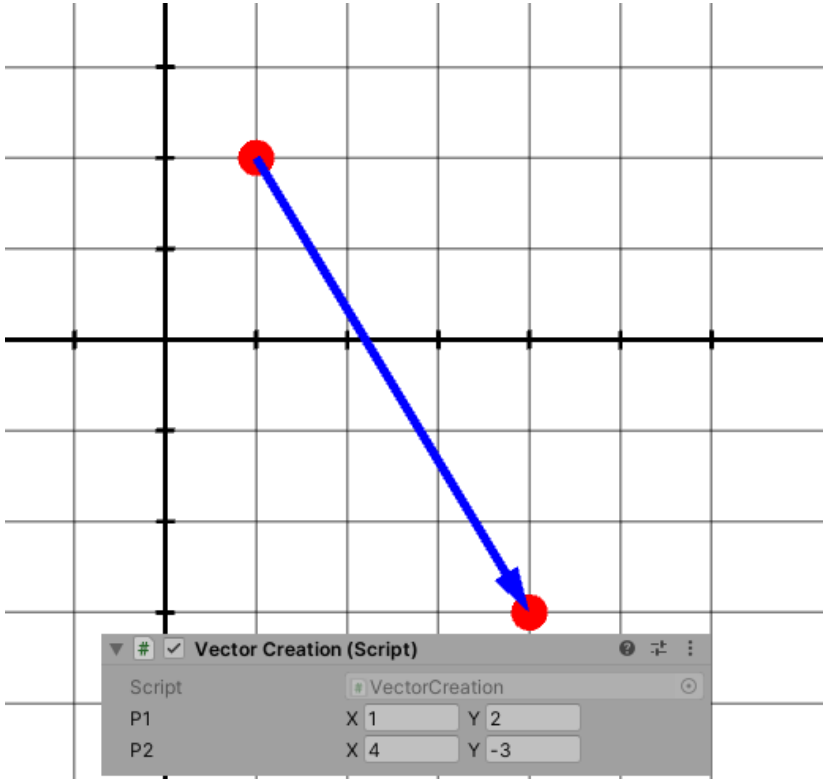
- Note that we can talk about direction at
  any point on the plane
- Point is like geographical coordinates;
  vector is like a geographical direction (east/west/...)
- Vector is also characterized by its **length**

Introduction – Vector Creation from Two Points

- Let's say we have two points $p_1 = (1, 2)$ and $p_2 = (4, -3)$
- Vector $\vec{v}$ can be created from two points by subtracting the beginning point from the end point:

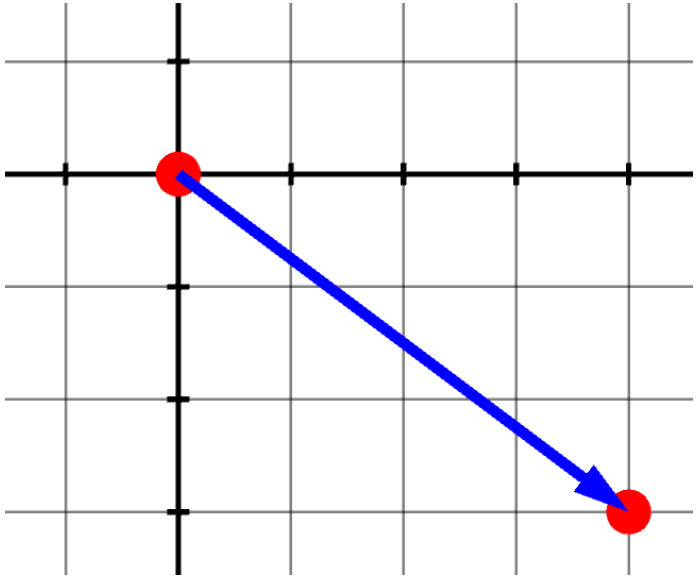
$$\vec{v} = p_2 - p_1$$

- $\vec{v} = (4, -3) - (1, 2) = [4 - 1, -3 - 2] = [3, -5]$
- $p_2 = p_1 + \vec{v}$
- In Unity to represent vectors we use the following types: `Vector2`, `Vector3` and `Vector4`



Introduction – Vector as a Point

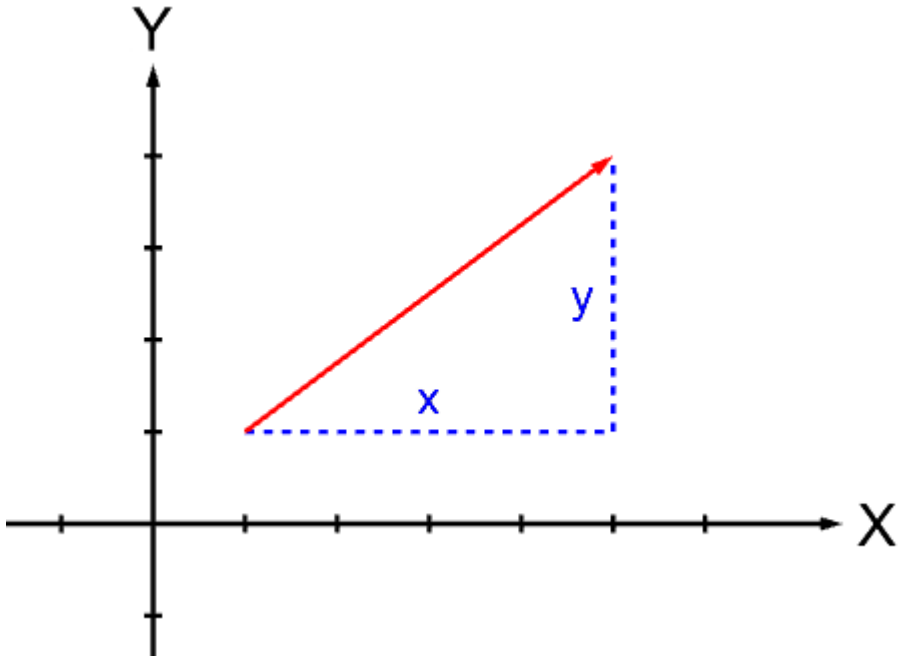
- Vector in itself represents only a direction. It has nothing to do with location/position
- However, if we decide that a vector has its beginning at the origin, then that vector's coordinates are the same as its end point coordinates
- In the figure we have a vector $\vec{v} = [4, -3]$, whose beginning and end have coordinates $p_1 = (0, 0)$ oraz $p_2 = (4, -3)$.
  As a result: $p_2 = \vec{v}$
- For this reason very often, in programming, the same types that are to represent vectors (`Vector2/3/4`) are used to represent points



Introduction – Length

- **Length of vector** $\vec{v}$ is calculated using the formula (Pythagorean theorem):

$$|\vec{v}| = \sqrt{x^2 + y^2}$$

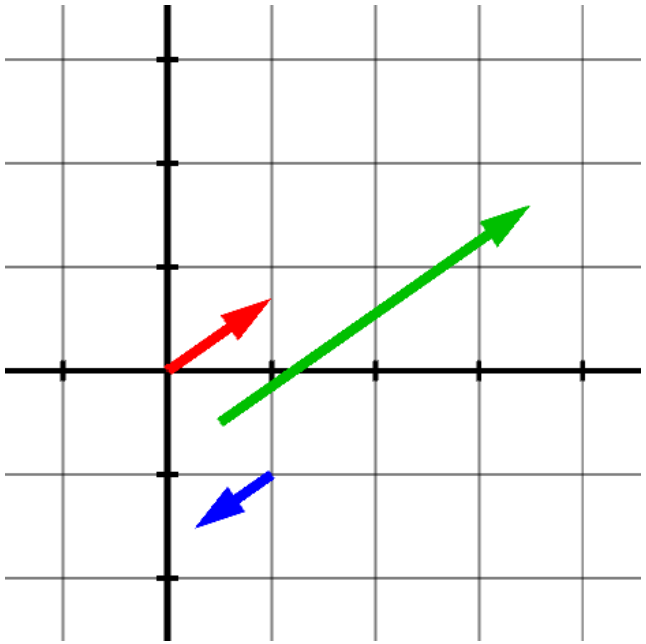


- For example, for $\vec{v} = [4, 3]$ we get:

$$|\vec{v}| = \sqrt{4 * 4 + 3 * 3} = \sqrt{16 + 9} = \sqrt{25} = 5$$

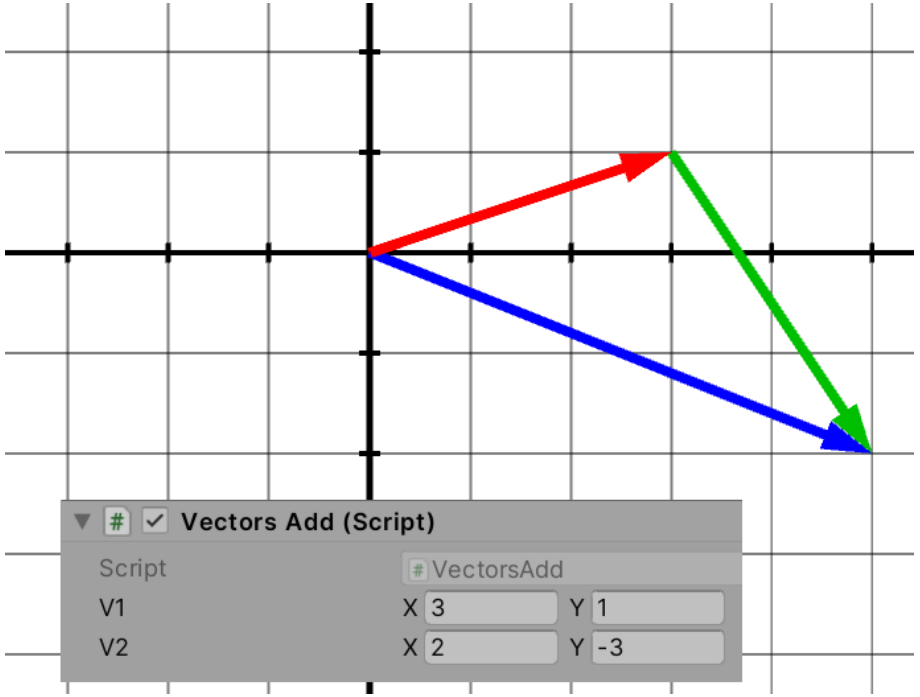
Introduction – Scalar Multiplication

- $\vec{v}_1 = [1, 0.7]$
- $\vec{v}_2 = 3 * \vec{v}_1 = 3 * [1, 0.7] = [3, 2.1]$
- $\vec{v}_3 = -0.75 * \vec{v}_1 = -0.75 * [1, 0.7] = [-0.75, -0.525]$



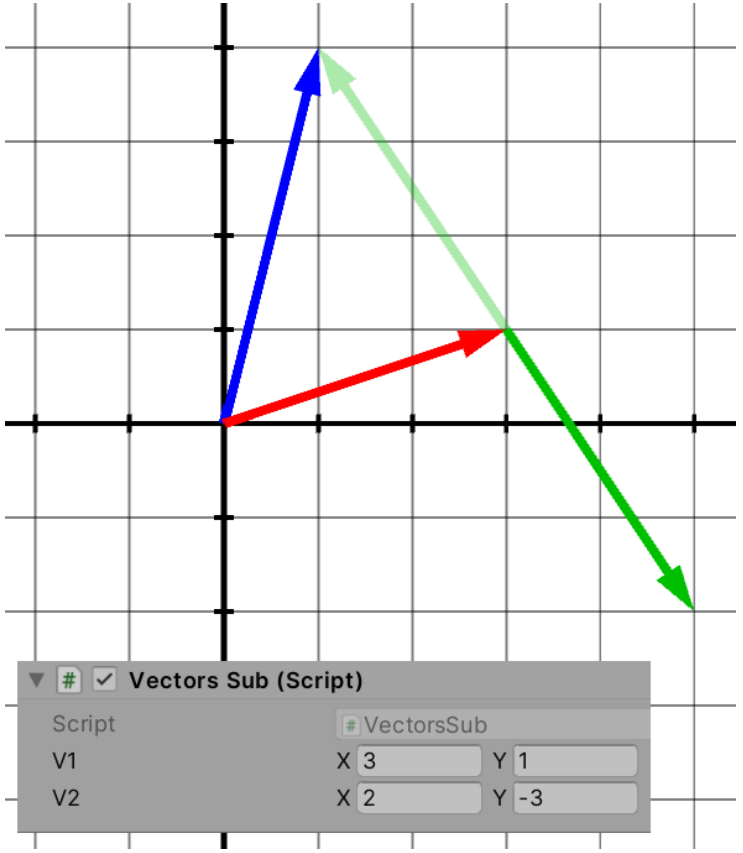
Introduction – Addition

- $\vec{v}_1 = [3,\ 1]$ $\vec{v}_2 = [2, -3]$
- $\vec{v}_3 = \vec{v}_1 + \vec{v}_2$
- $\vec{v}_3 = [3,\ 1] + [2, -3] = [5, -2]$



Introduction – Subtraction

- It's best to **negate** first, and then add
- $\vec{v}_1 = [3, 1] \qquad \vec{v}_2 = [2, -3]$
- $\vec{v}_3 = \vec{v}_1 - \vec{v}_2 = \vec{v}_1 + (-\vec{v}_2)$
- $\vec{v}_3 = [3, 1] + (-[2, -3]) = [3, 1] + [-2, 3] = [1, 4]$



Introduction – Multiplication

- Component-wise multiplication of two vectors does not have too many useful geometrical interpretations
- In practice we most often use it when mixing colors (when we represent them with vector types)
- $\vec{c}_1 = [0.4, 0.9, 0.3] \qquad \vec{c}_2 = [1, 0.2, 0.1]$
- $\vec{c}_3 = \vec{c}_1 * \vec{c}_2 = [0.4 * 1, 0.9 * 0.2, 0.3 * 0.1] = [0.4, 0.18, 0.03]$

Linear Interpolation

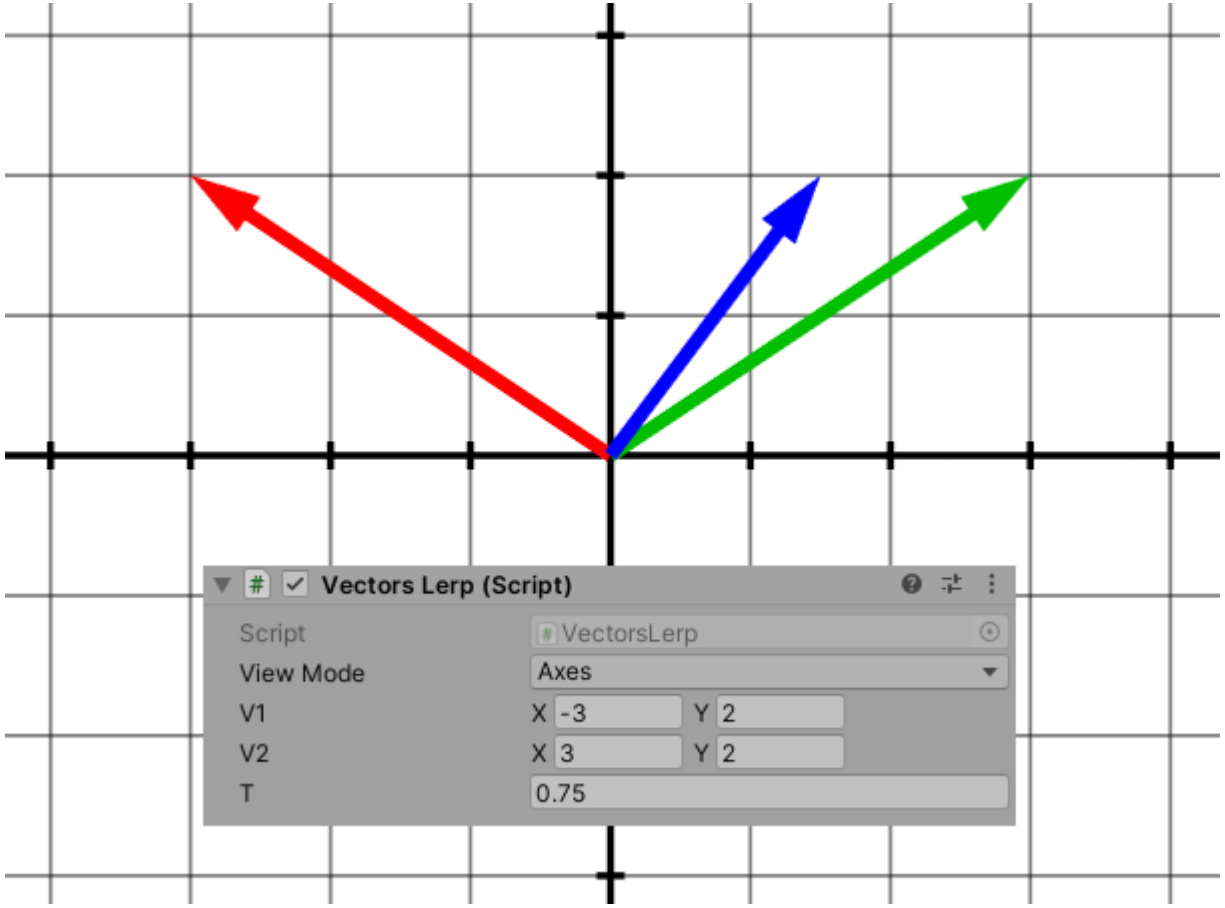
- **Linear interpolation** allows progressive change from one value into another
- The classical linear interpolation formula:

$$\mathrm{Lerp}(a, b, t) \;=\; a(1-t) + bt \;=\; a + (b-a)t$$

$$\mathrm{Lerp}(0.4, 1.3, 0.5) = 0.4 + (1.3 - 0.4) * 0.5 = 0.4 + 0.9 * 0.5 = 0.4 + 0.45 = 0.85$$

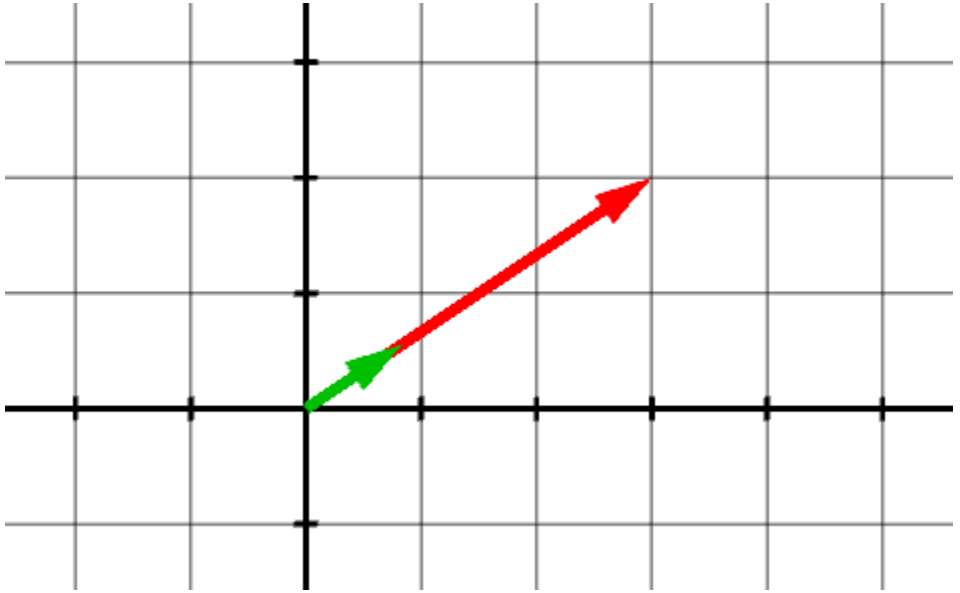
- The t argument denotes „progress”
- For $t = 0$ we get a, for $t = 1$ we get b
- Arguments a and b can be any values that „make sense”. Such as real numbers or vectors
- Watch [An In-Depth look at Lerp, Smoothstep, and Shaping Functions]

Linear Interpolation



Normalization

- A special case of a vector is the **unit vector**. It is a vector whose length is 1
- **Vector normalization** is a process in which a vector whose length is not 1, will end up with length of 1. This operation preserves the vector's direction



Normalization

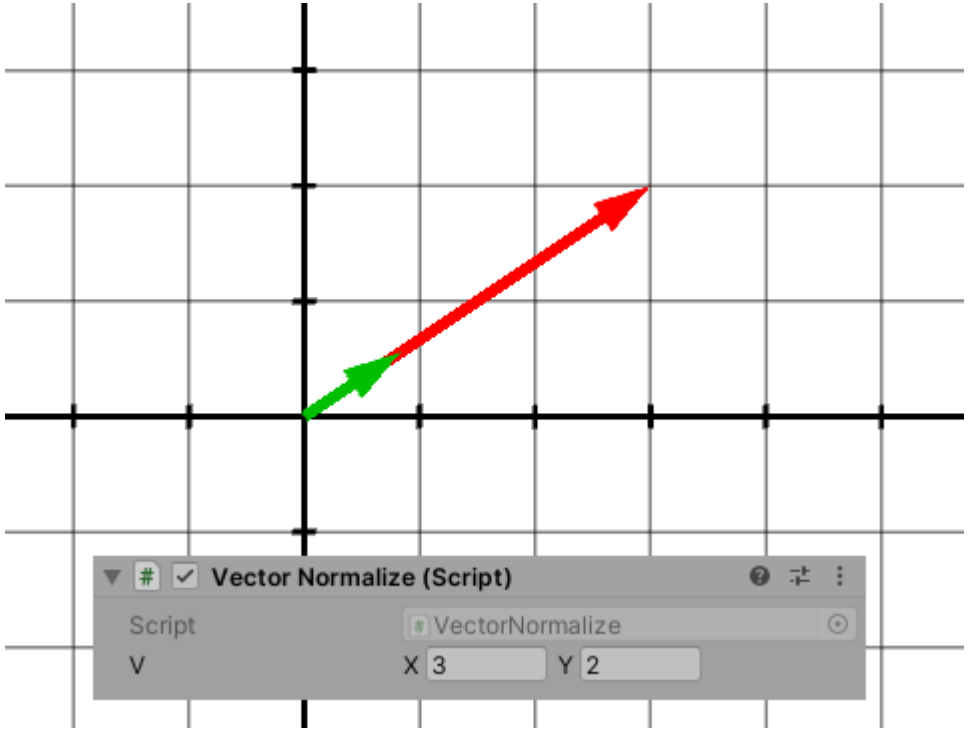
- To normalize the vector $\vec{v}$ all we need to do is to divide each vector's coordinate by the length of that vector. As a result we get a vector $\vec{u}$ with the same direction, but length of 1
- Formula for $\vec{u}$:

$$\vec{u} = \frac{\vec{v}}{|\vec{v}|}$$

- For example, for $\vec{v} = [4, 3]$ we had $|\vec{v}| = 5$, so:

$$\vec{u} = \frac{\vec{v}}{|\vec{v}|} = \frac{[4, 3]}{5} = [0.8, 0.6]$$

Normalization



Normalization

- If we want a vector to have a certain length we can first normalize it, and then multiply it by the desired length
- Suppose we want the vector $\vec{v}$ to have length 7. We then need to compute:

$$7\frac{\vec{v}}{|\vec{v}|}$$

Dot Product

- We have two vectors $\vec{a} = [a_x, a_y]$ and $\vec{b} = [b_x, b_y]$
- Their **dot product** is defined as:

$$\vec{a} \circ \vec{b} = a_x b_x + a_y b_y$$

- But also as:

$$\vec{a} \circ \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$$

where θ is the angle between the vectors

Dot Product

- Let's equate those two formulas:

$$a_x b_x + a_y b_y = |\vec{a}||\vec{b}| \cos(\theta)$$

$$\cos(\theta) = \frac{a_x b_x + a_y b_y}{|\vec{a}||\vec{b}|}$$

- By calculating $\cos^{-1}$ on both sides we can get θ:

$$\theta = \cos^{-1}\left(\frac{a_x b_x + a_y b_y}{|\vec{a}||\vec{b}|}\right)$$

Dot Product

- Note that when $\vec{b} = \vec{a}$:

$$\vec{a} \circ \vec{a} = a_x a_x + a_y a_y$$

then as a result we get the square length of the vector $\vec{a}$, because:

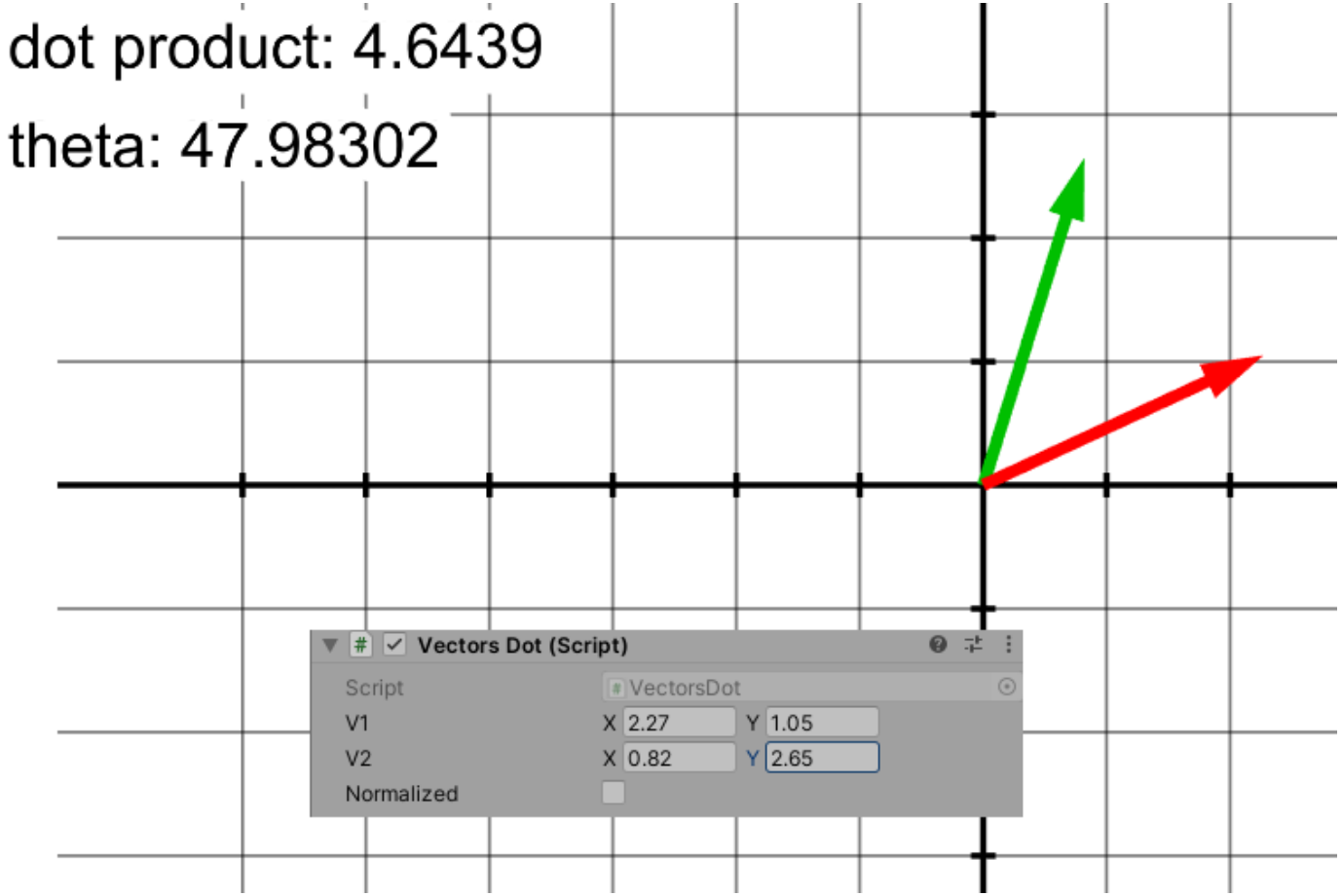
$$|\vec{a}| = \sqrt{a_x a_x + a_y a_y}$$

$$|\vec{a}|^2 = a_x a_x + a_y a_y$$

$$|\vec{a}|^2 = \vec{a} \circ \vec{a}$$

- This gives us the ability to compare lengths between vectors very efficiently

Dot Product



Dot Product

- Dot product has **very** wide applications in both 2D and 3D
- When vectors are normalized their dot product lets us determine their „likeness”:

$$\vec{a} \circ \vec{b} = -1 \quad \rightarrow \quad \text{opposite direction}$$
$$\vec{a} \circ \vec{b} = 0 \quad \rightarrow \quad \text{perpendicular}$$
$$\vec{a} \circ \vec{b} = 1 \quad \rightarrow \quad \text{the same}$$

- Example applications:
  - calculate amount of lighting that falls onto a surface, depending on the angle of incidence
  - fast (square) distance calculation between points
  - determining whether an AI can see the player
  - calculating velocity with which an object should slide along a wall
  - ...

Dot Product

- Commutative property:

$$\vec{a} \circ \vec{b} = \vec{b} \circ \vec{a}$$

- Distributive property (over vector addition):

$$\vec{a} \circ \left(\vec{b} + \vec{c}\right) = \vec{a} \circ \vec{b} + \vec{a} \circ \vec{c}$$

- Scalar Multiplication Property:

$$(s * \vec{a}) \circ \vec{b} = s\left(\vec{a} \circ \vec{b}\right)$$

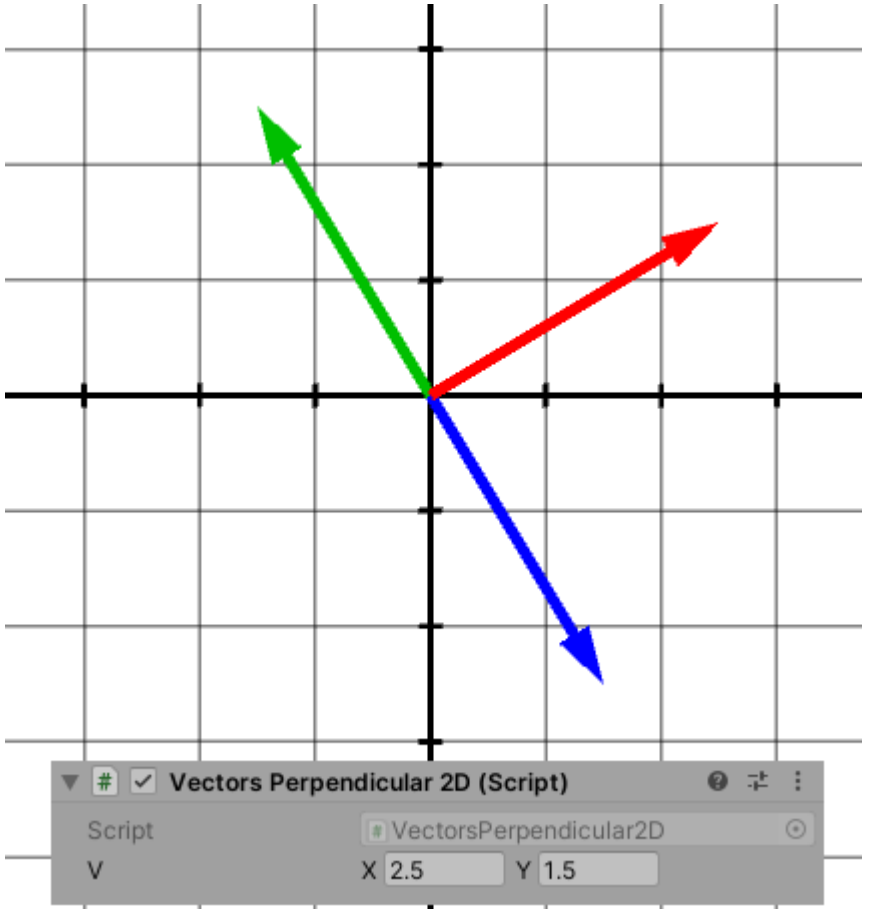
$$\vec{a} \circ \left(s * \vec{b}\right) = s\left(\vec{a} \circ \vec{b}\right)$$

Perpendicular Vectors in 2D

- Having a vector $\vec{v} = [x, y]$ in 2D we can calculate **perpendicular vectors** to it
- There are two such vectors $\vec{n}_1$ oraz $\vec{n}_2$:

$$\vec{n}_1 = [-y, x]$$

$$\vec{n}_2 = [y, -x]$$



Perpendicular Vectors in 2D

- Thanks to knowing the dot product we can easily prove that
- Let's say we have some vector $\vec{v} = [x, y]$
- Calculate the dot product with one of the perpendicular vectors:

$$\vec{v} \circ \vec{n}_1 = [x, y] \circ [-y, x]$$

$$x(-y) + yx = -xy + xy = 0$$

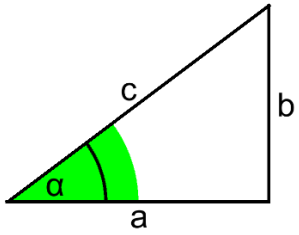
Perpendicular Vectors in 2D

- Let’s revisit program Functions from chapter 1:

Functions.cs

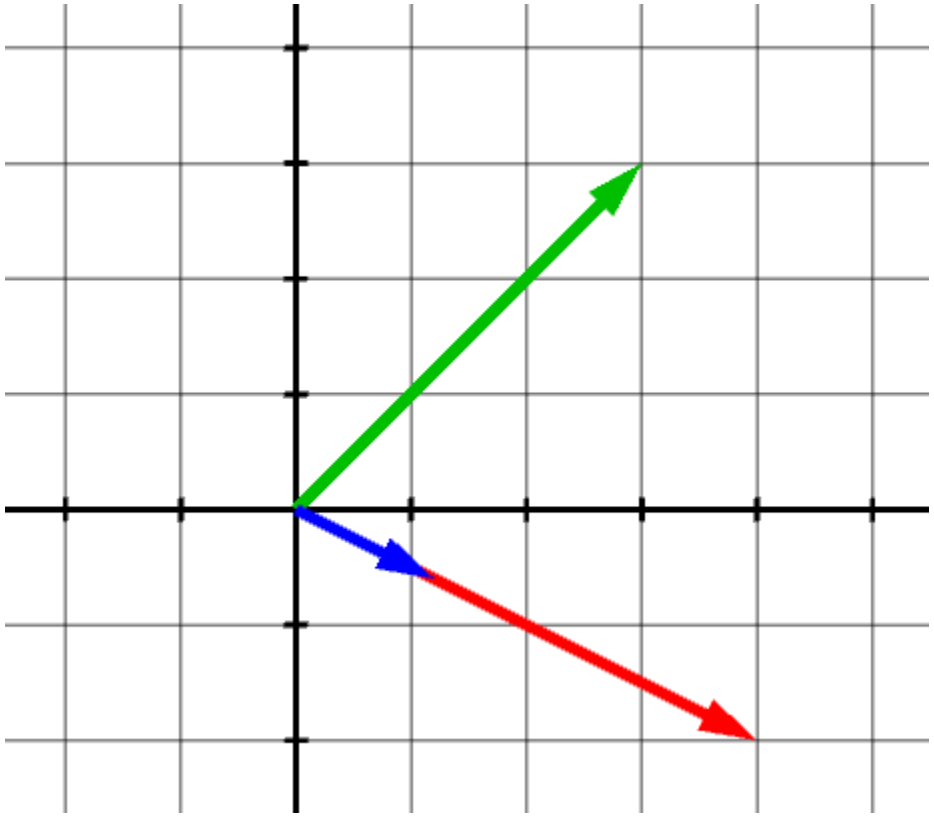
Assembly-CSharp | Functions

```
8    void Update()
9    {
10       float x_atRightAngle = x2;
11       float y_atRightAngle = y1;
12
13       Vector2 p1 = new Vector2(x1, y1);
14       Vector2 p2 = new Vector2(x2, y2);
15       Vector2 p3 = new Vector2(x_atRightAngle, y_atRightAngle);
16
17       Vector2 e1 = p2 - p1;
18       Vector2 e2 = p3 - p2;
19       Vector2 e3 = p1 - p3;
20       Vector2 e1_normal = new Vector2(-e1.y, e1.x).normalized;
21       Vector2 e2_normal = new Vector2(-e2.y, e2.x).normalized;
22       Vector2 e3_normal = new Vector2(-e3.y, e3.x).normalized;
23
24       Vector2 labelAPosition = p3 + 0.5f*e3 + 0.25f*e3_normal;
25       Vector2 labelBPosition = p2 + 0.5f*e2 + 0.25f*e2_normal;
26       Vector2 labelCPosition = p1 + 0.5f*e1 + 0.25f*e1_normal;
27
```



Vector Projection

- **Vector projection** is an operation where we take one vector and project it onto another one:

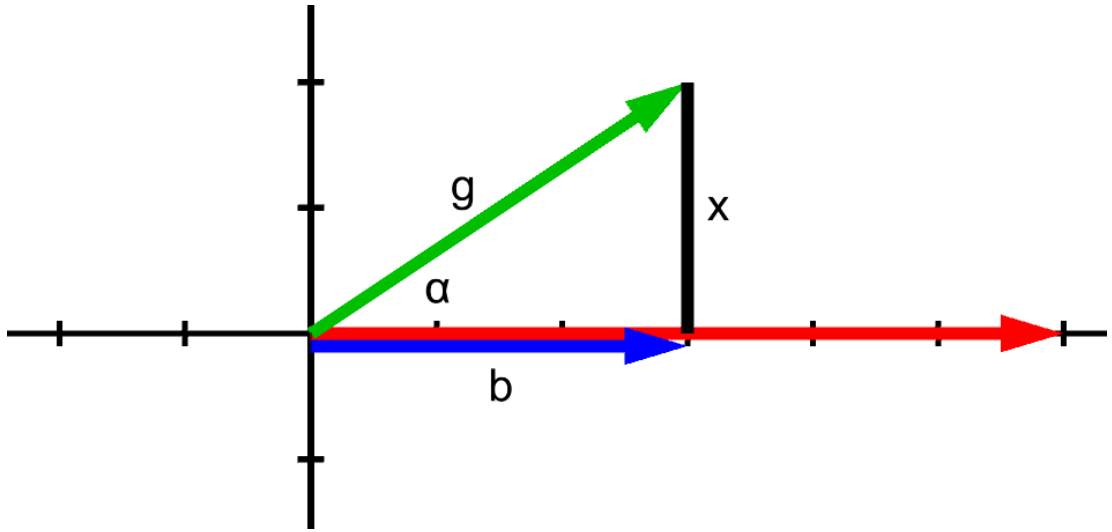


Vector Projection

- By projecting the green vector $\vec{G}$ onto the red $\vec{R}$ we get the blue $\vec{B}$
- The green and blue vectors form a right triangle, where:

$$\cos(\alpha) = \frac{b}{g}$$

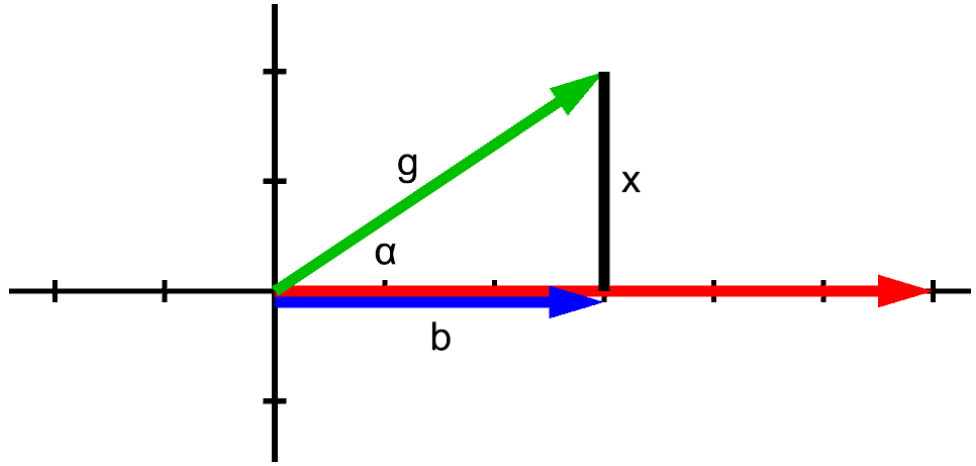
- Our goal is to find b (length of $\vec{B}$)



Vector Projection

- So:

$$\cos(\alpha) = \frac{b}{g} \qquad b = g\cos(\alpha)$$



$$b = |\vec{G}|\frac{\vec{R} \circ \vec{G}}{|\vec{R}||\vec{G}|} = \frac{\vec{R} \circ \vec{G}}{|\vec{R}|}$$

$$\vec{B} = b * \frac{\vec{R}}{|\vec{R}|} = \frac{\vec{R} \circ \vec{G}}{|\vec{R}|} * \frac{\vec{R}}{|\vec{R}|} = \frac{(\vec{R} \circ \vec{G})\vec{R}}{(\vec{R} \circ \vec{R})}$$

Vector Projection

- If $\vec{R}$ is already normalized, then:

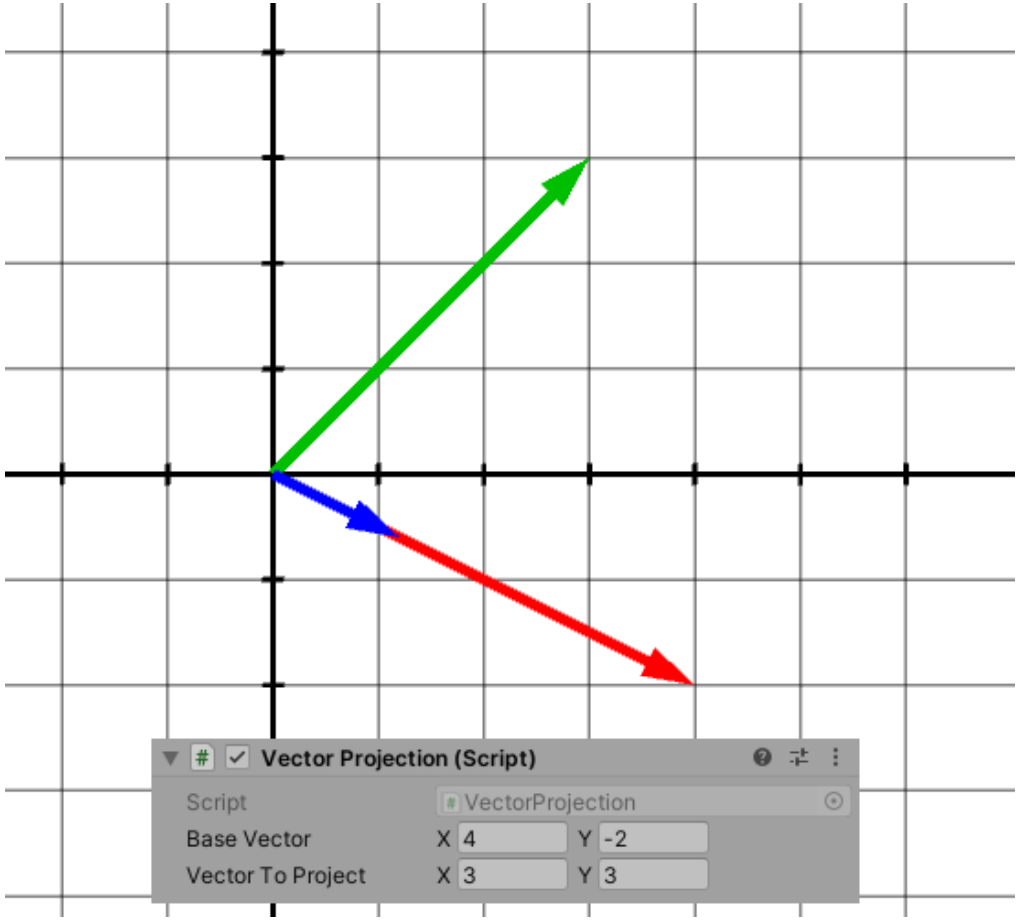
$$b = \vec{R} \circ \vec{G}$$

- With that assumption ($\vec{R}$ is normalized) we get $\vec{B}$:

$$\vec{B} = b\vec{R} = \left(\vec{R} \circ \vec{G}\right)\vec{R}$$

- The above formulas work in all dimensions

Vector Projection



Cross Product (Perpendicular Vectors in 3D)

- We have two vectors $\vec{a} = [a_x, a_y, a_z]$ and $\vec{b} = [b_x, b_y, b_z]$
- **Cross product** is defined as (3D only):

$$\vec{c} = \vec{a} \times \vec{b}$$

$$c_x = a_y b_z - a_z b_y$$
$$c_y = a_z b_x - a_x b_z$$
$$c_z = a_x b_y - a_y b_x$$

- The resulting vector $\vec{c}$ will be perpendicular to both $\vec{a}$ and $\vec{b}$ (this will become very important shortly)

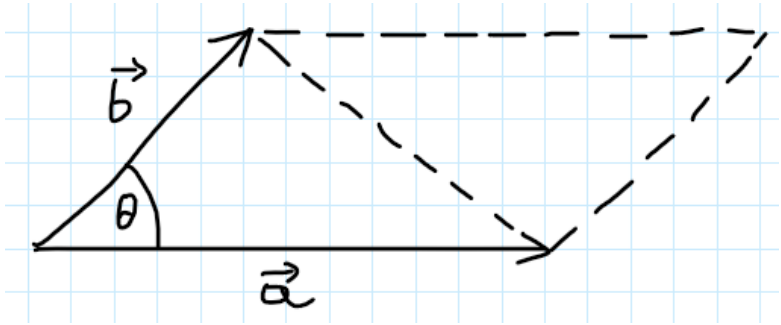
Cross Product (Perpendicular Vectors in 3D)

- The length of the resulting vector is:

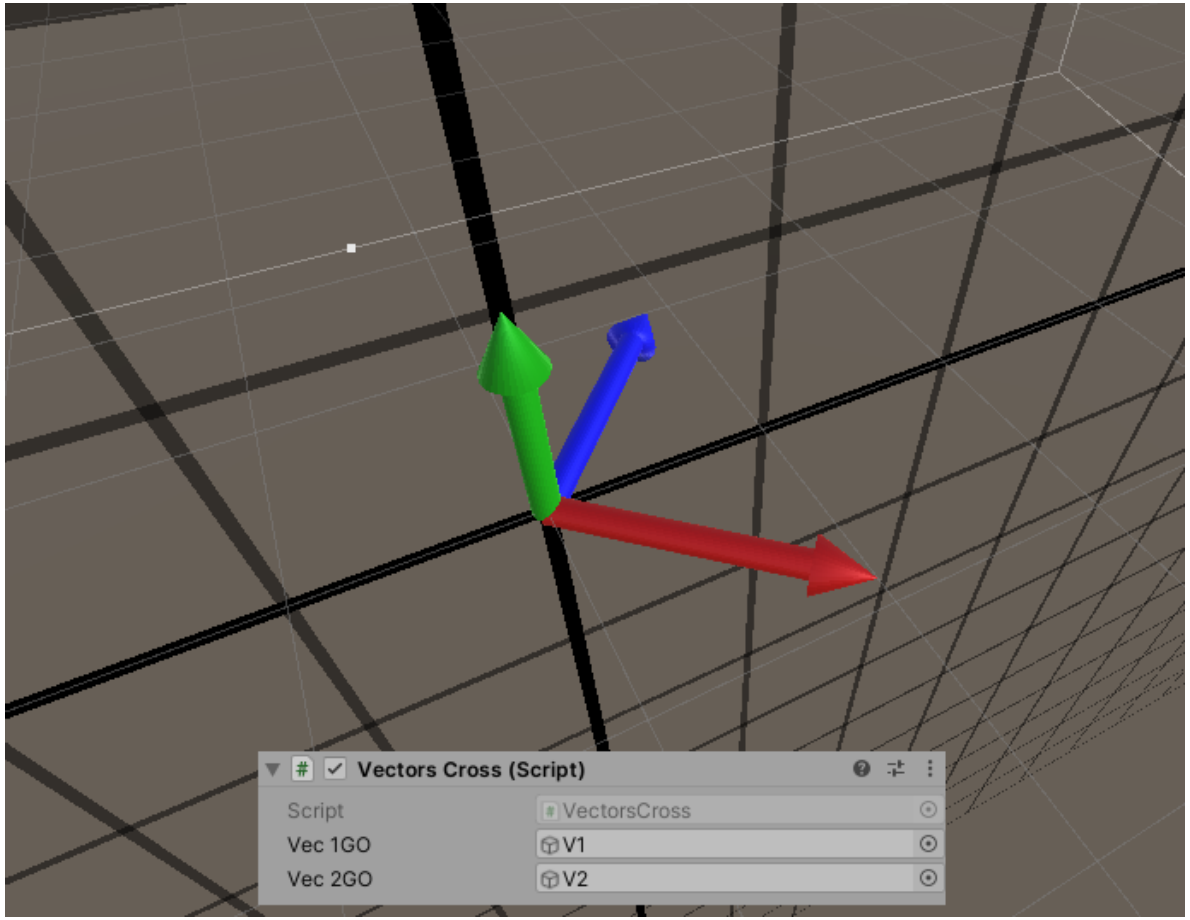
$$|\vec{c}| = |\vec{a}||\vec{b}|\sin(\theta)$$

where θ is the angle between the vectors

- The length of vector $\vec{c}$ is equal to the area of the parallelogram that spans $\vec{a}$ and $\vec{b}$:



Cross Product (Perpendicular Vectors in 3D)



Cross Product (Perpendicular Vectors in 3D)

- Cross product is **not** commutative:

$$\vec{a} \times \vec{b} \neq \vec{b} \times \vec{a}$$

$$\vec{a} \times \vec{b} = -\left(\vec{b} \times \vec{a}\right)$$

- Worth to remember:

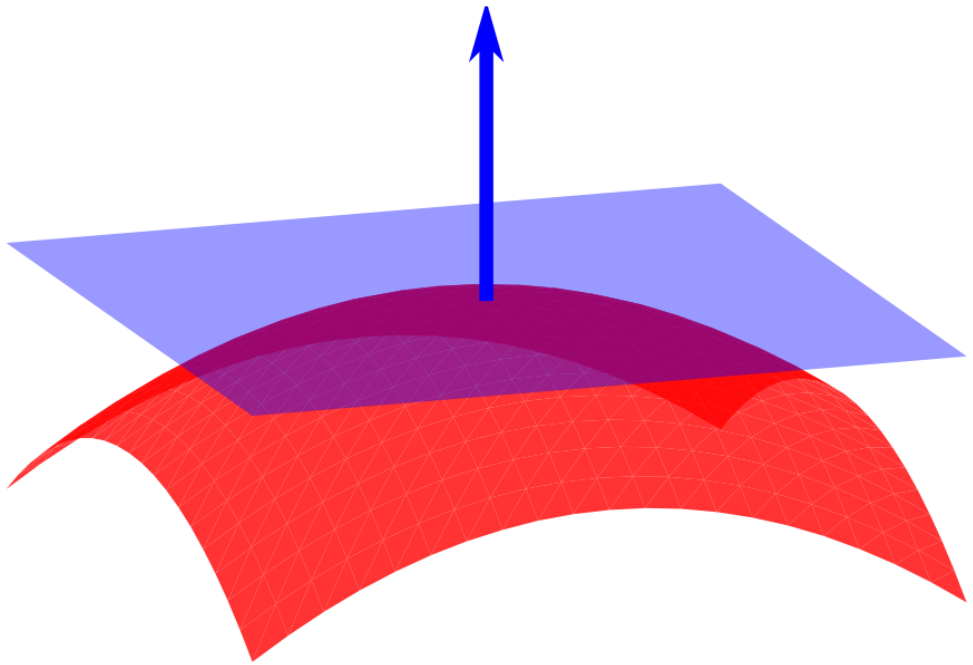
$$[1,0,0] \times [0,1,0] = [0,0,1]$$

$$[0,1,0] \times [0,0,1] = [1,0,0]$$

$$[0,0,1] \times [1,0,0] = [0,1,0]$$

Normal Vector

- **Normal vector** is a vector that is **perpendicular** to a surface, line, etc.
- A normal vector does not have to be normalized, although it usually is



https://en.wikipedia.org/wiki/File:Surface_normal_illustration.svg **Author** Patrick87

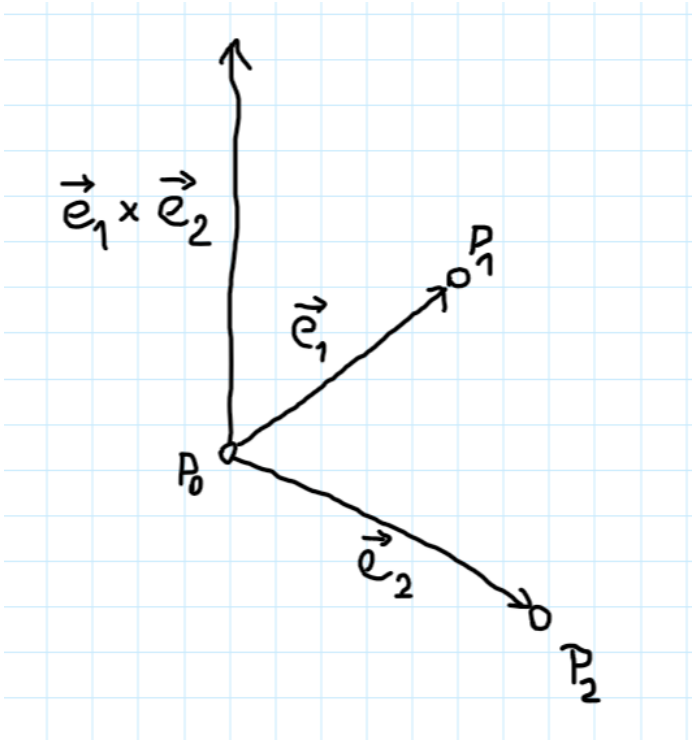
Triangle

- Triangles are the basic building blocks of geometry in 3D scenes
- One of the most important pieces of information that we usually need about geometry is knowing the normal vector(s)
- Usually we can calculate it using the cross product:

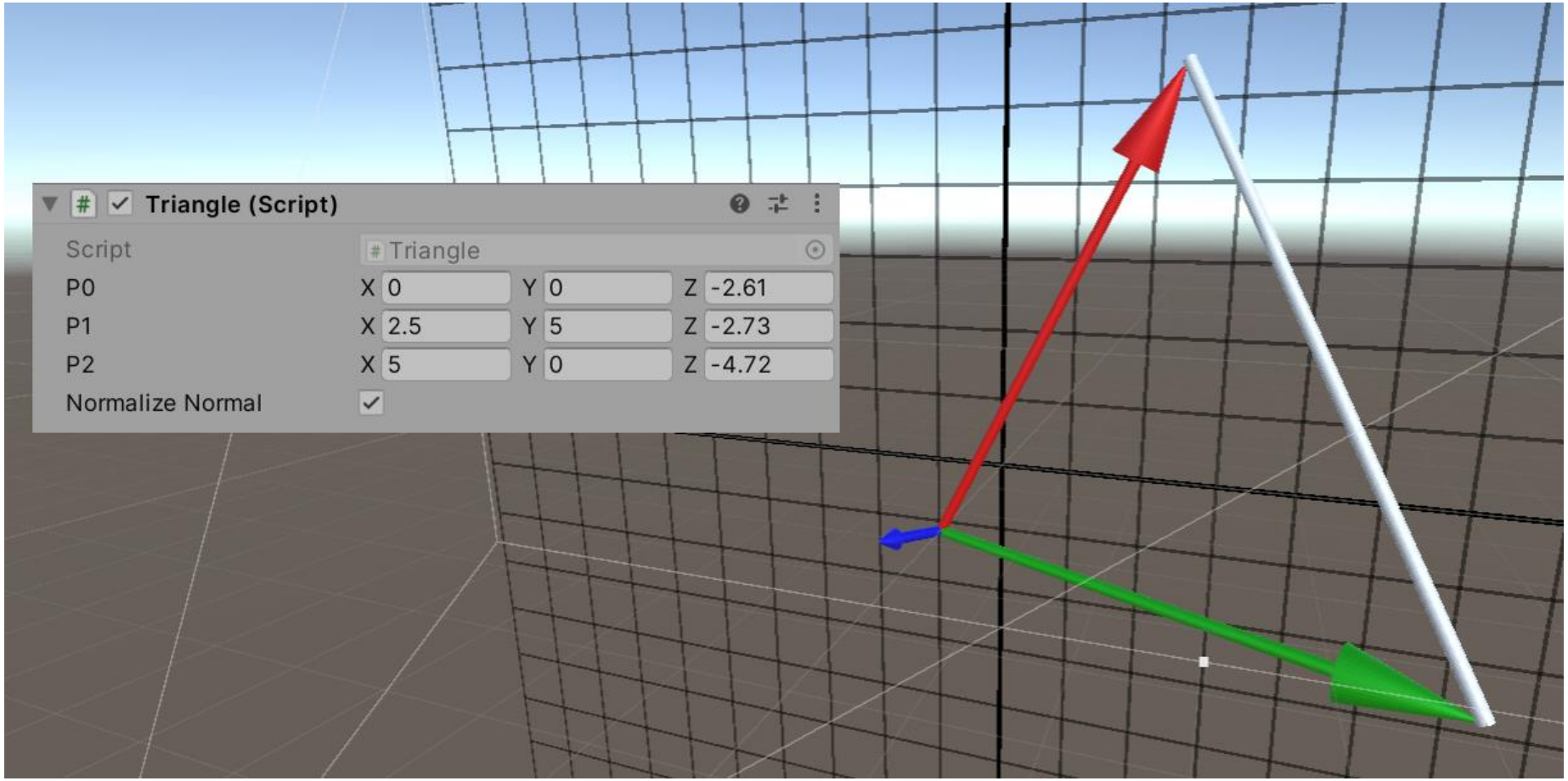
$$\vec{e}_1 = p_1 - p_0$$
$$\vec{e}_2 = p_2 - p_0$$

$$\vec{n} = \vec{e}_1 \times \vec{e}_2$$

- Vectors $\vec{e}_1$ and $\vec{e}_2$ are called **edge vectors**



Triangle



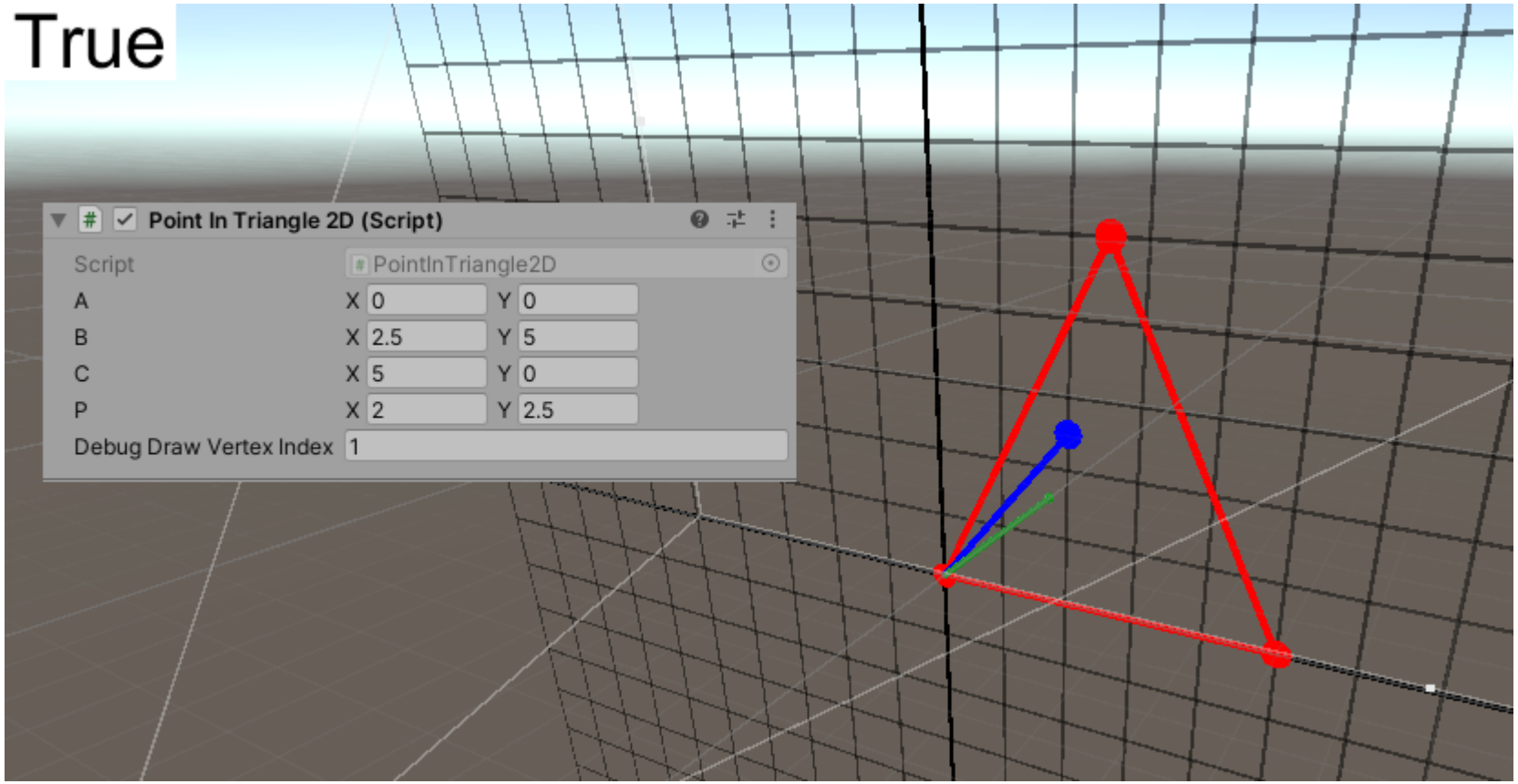
Triangle

- It is very important to realize that relative ordering of vertices will affect the direction of the normal vector
- **Clockwise (CW)** ordering of vertices and **counter-clockwise (CCW)** ordering will give us the opposite normal vectors
- In this course overall we employ CW ordering

Triangle

- In 2D the cross product can be used to determine on which side of one vector another vector is located
- Thanks to this property of cross product we can check if a point is inside a 2D triangle (or any 2D convex polygon).
  A similar test but in 3D we will cover in chapter 4

Triangle



Triangle

- This program doesn’t handle the case where ordering of the vertices becomes CCW
- This code works only for CW:

```
bool isInside = false;
if (ap_ab.z > 0.0f &&
    bp_bc.z > 0.0f &&
    cp_ca.z > 0.0f)
{
    isInside = true;
}
```

- This will work for CCW:

```
bool isInside = false;
if (ap_ab.z < 0.0f &&
    bp_bc.z < 0.0f &&
    cp_ca.z < 0.0f)
{
    isInside = true;
}
```

Normalized Vector Compression

- Normalized vectors are extremely common in 3D graphics
- A typical 3D vector requires three `float` variables (12 bytes total in memory)
- By making use of certain properties of normalized vectors we can easily reduce these requirements to two floats (8 bytes)

Normalized Vector Compression

- We know that the length of a normalized vector is 1:

$$|\vec{v}| = 1$$

$$|\vec{v}| = \sqrt{\vec{v}_x{}^2 + \vec{v}_y{}^2 + \vec{v}_z{}^2}$$

$$\sqrt{\vec{v}_x{}^2 + \vec{v}_y{}^2 + \vec{v}_z{}^2} = 1$$

Normalized Vector Compression

- From this equation/constraint we can calculate the value of one of the vector's coordinates:

$$\sqrt{\vec{v}_x{}^2 + \vec{v}_y{}^2 + \vec{v}_z{}^2} = 1$$

$$\vec{v}_x{}^2 + \vec{v}_y{}^2 + \vec{v}_z{}^2 = 1$$

$$\vec{v}_z{}^2 = 1 - \vec{v}_x{}^2 - \vec{v}_y{}^2$$

$$\vec{v}_z = \pm\sqrt{1 - \vec{v}_x{}^2 - \vec{v}_y{}^2}$$

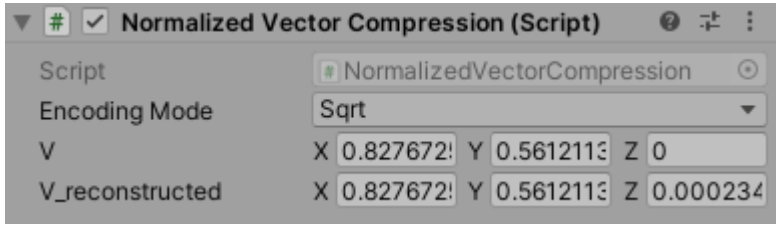
Normalized Vector Compression

- We got:

$$\vec{v}_z = \pm\sqrt{1 - \vec{v}_x{}^2 - \vec{v}_y{}^2}$$

- This means that to store a normalized 3D vector in memory we need only two coordinates instead of three; plus information about the sign

Normalized Vector Compression

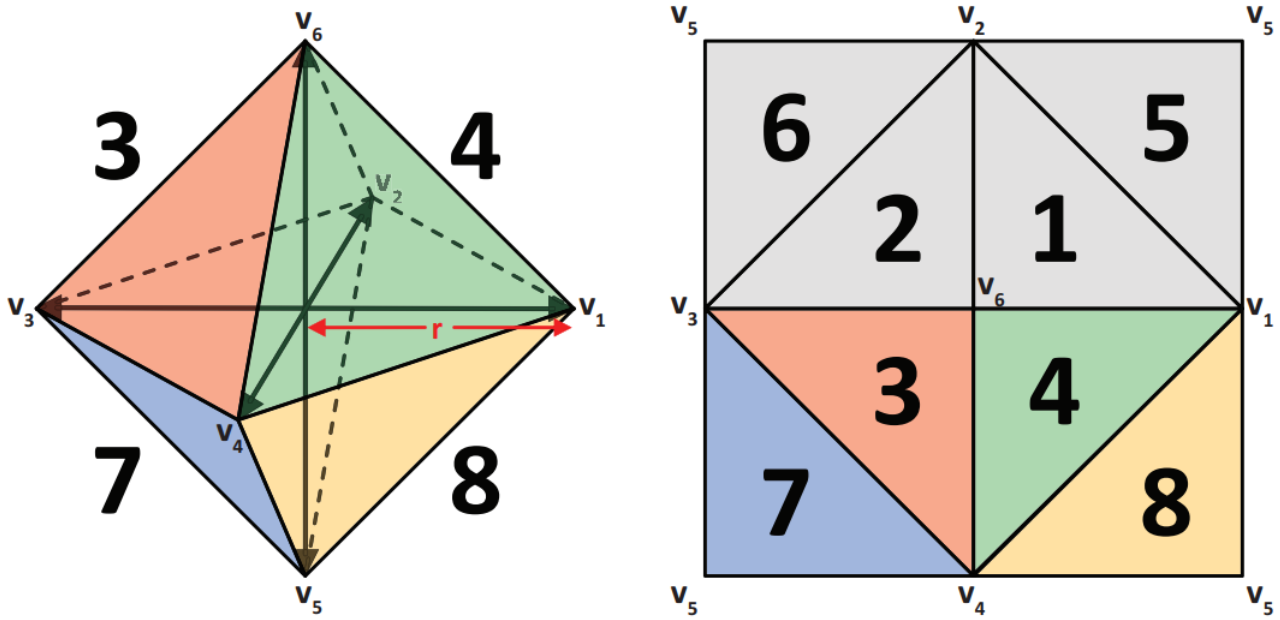


Normalized Vector Compression

- In the algorithm that we have just used, the need to store the sign is cumbersome
- There are in fact many alternatives for compressing/encoding a normalized vector, like spherical coordinates
- Another very good one are **octahedron vectors**

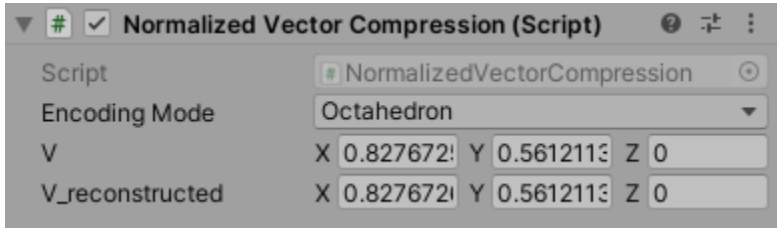
Normalized Vector Compression

- Illustration:



Octahedron Environment Maps, 2008
Engelhardt and Dachsbacher

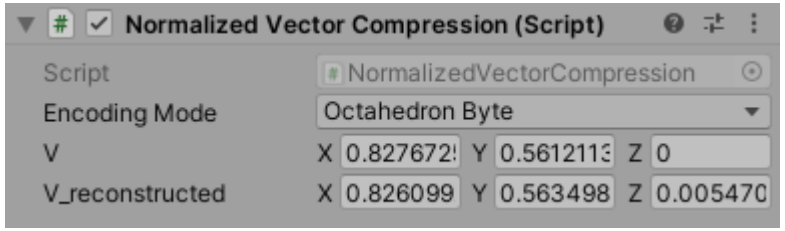
Normalized Vector Compression



Normalized Vector Compression

- Octahedron vectors allow us to get rid off of the troublesome sign storage
- To store an octahedron vector we need 8 bytes
- Due to the fact that the coordinates of normalized vectors are always in the $[-1, 1]$ range, we can try to use a data type with less precision

Normalized Vector Compression



Normalized Vector Compression

- Original vector was 12 bytes of memory
- The first algorithm that we used required 8 bytes of memory + packing the sign somewhere
- The second algorithm, octahedron vectors, required exactly 8 bytes of memory
- By using `byte` instead of `float` we can reduce the amount of memory needed to as little as 2 bytes with octahedrons. In many cases this will be good enough
- We can also use `byte` without any extra encodings; this will amount to 3 bytes

Exercises

1. Prove distributive property of dot product (slide 22)
2. Show that vectors $\vec{a}$ and $\vec{c}$ are perpendicular (slide 31)
3. Write a program that calculates the area of a parallelogram (slide 32)
4. Modify program PointInTriangle2D to work simultaneously for CW and CCW ordering of vertices (slide 40)
5. Write a program that checks if a point is inside a convex polygon (slide 40).
   You can assume that the input points form a polygon which is already convex and that the points are in CW order
6. Implement normalized vector compression using the algorithm with the square root and `byte` type (slide 45)