Math for 3D/Games Programmers

3. Vectors

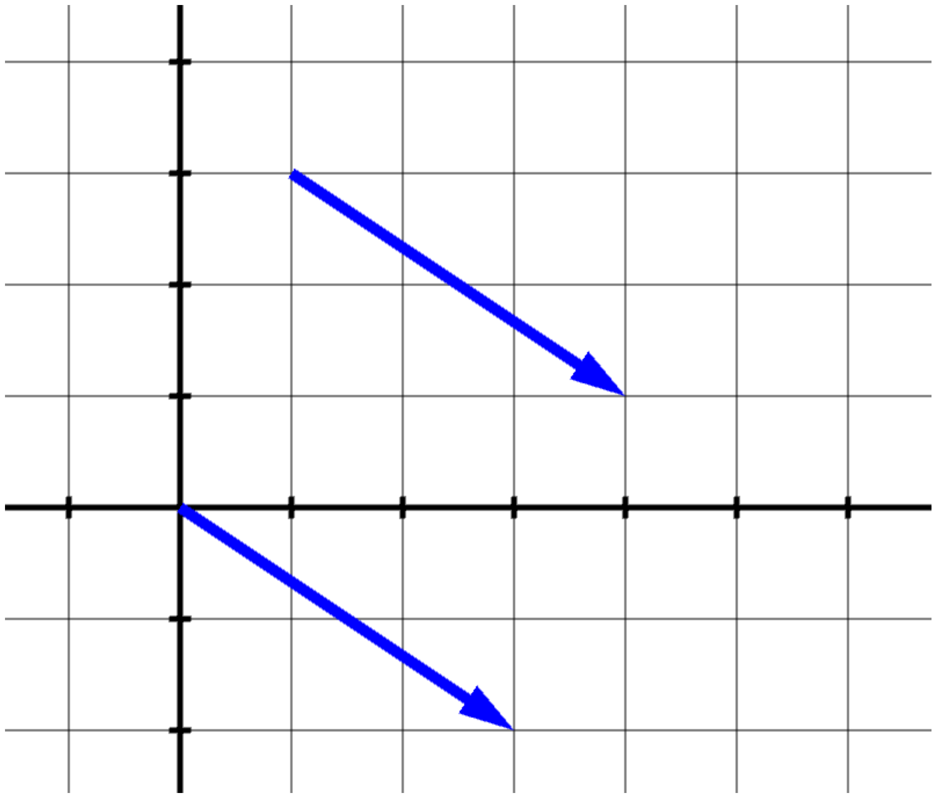
Table of Contents

Table of Contents

What is a Vector

- **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:

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



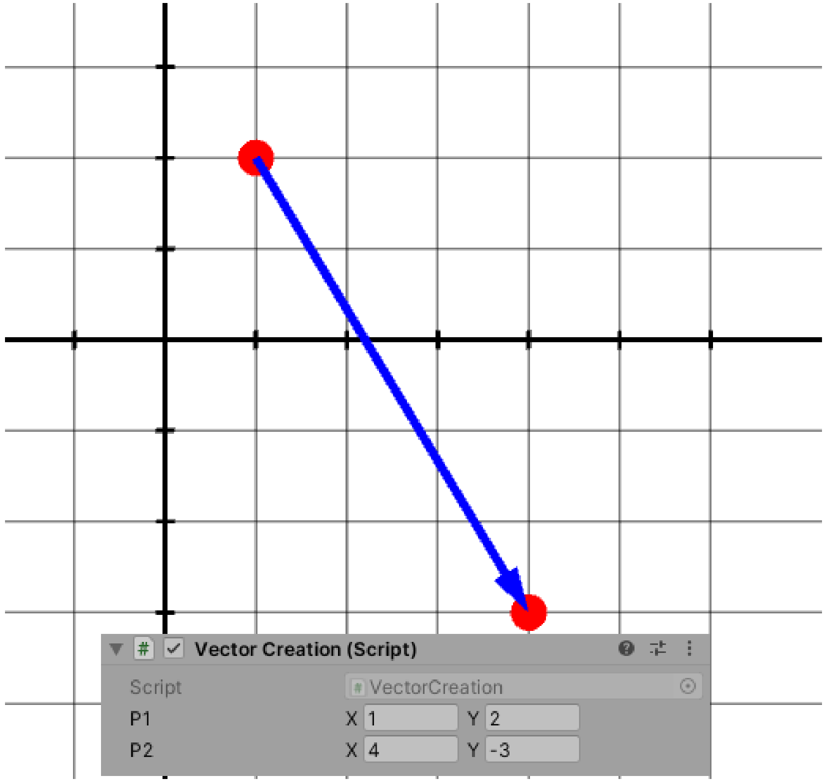
- Note that we can talk about direction at any point on the plane
- Point is like geographical coordinates; vector is like a geographical direction (east/west/...)
- Vector is also characterized by its **length**

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:

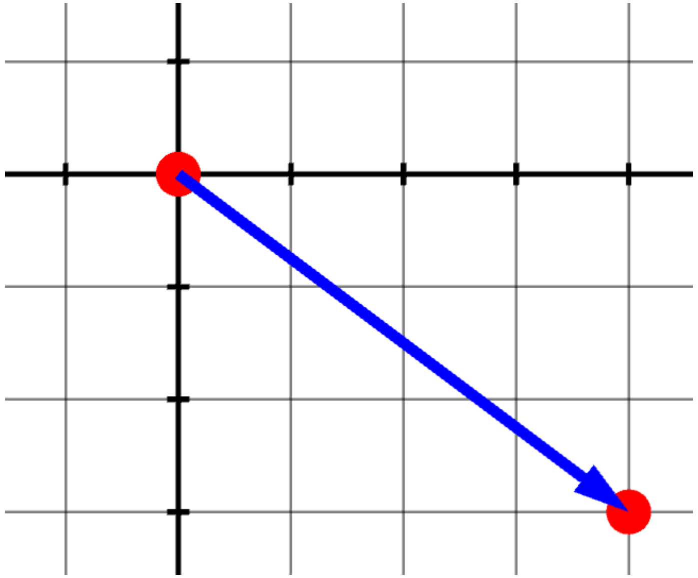
$$\vec{v} = p_2 - p_1$$

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



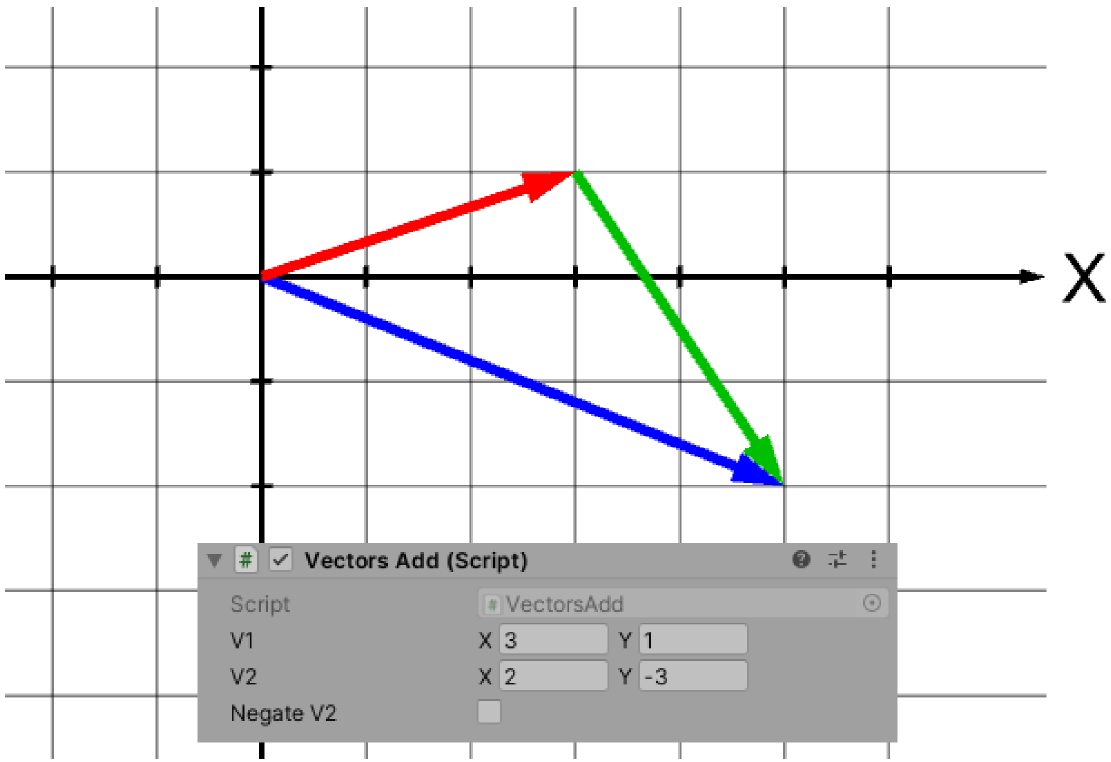
Vector as a Point

- 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



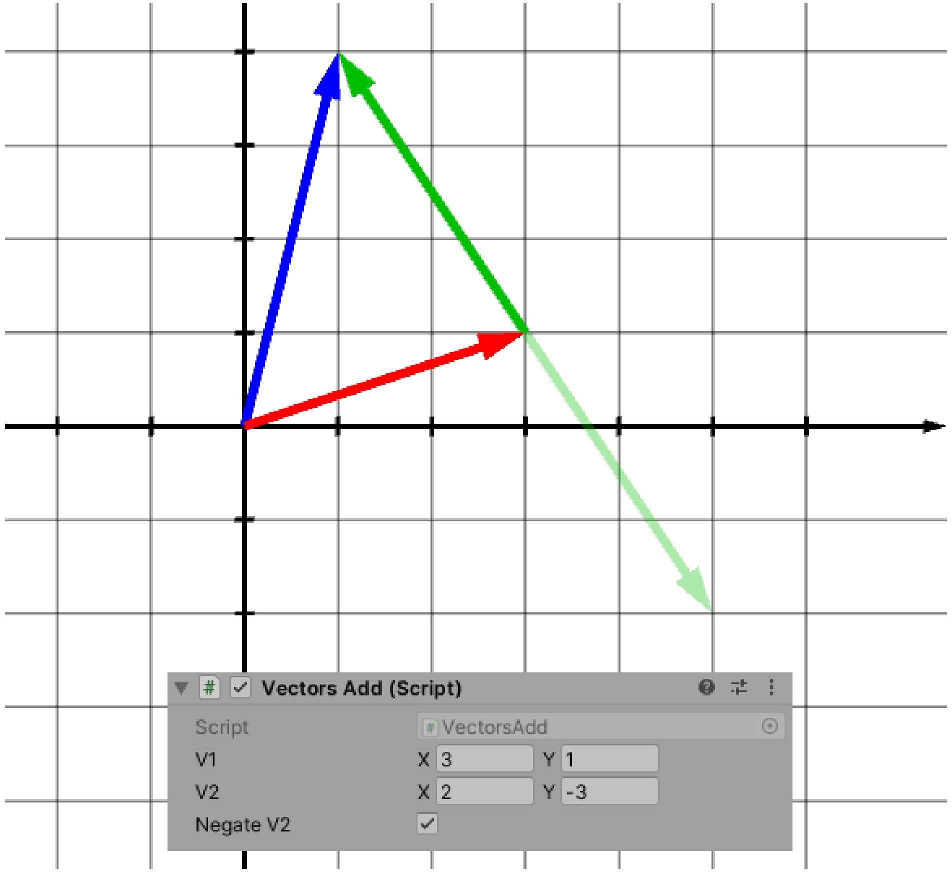
Addition

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



Subtraction

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



Linear Interpolation

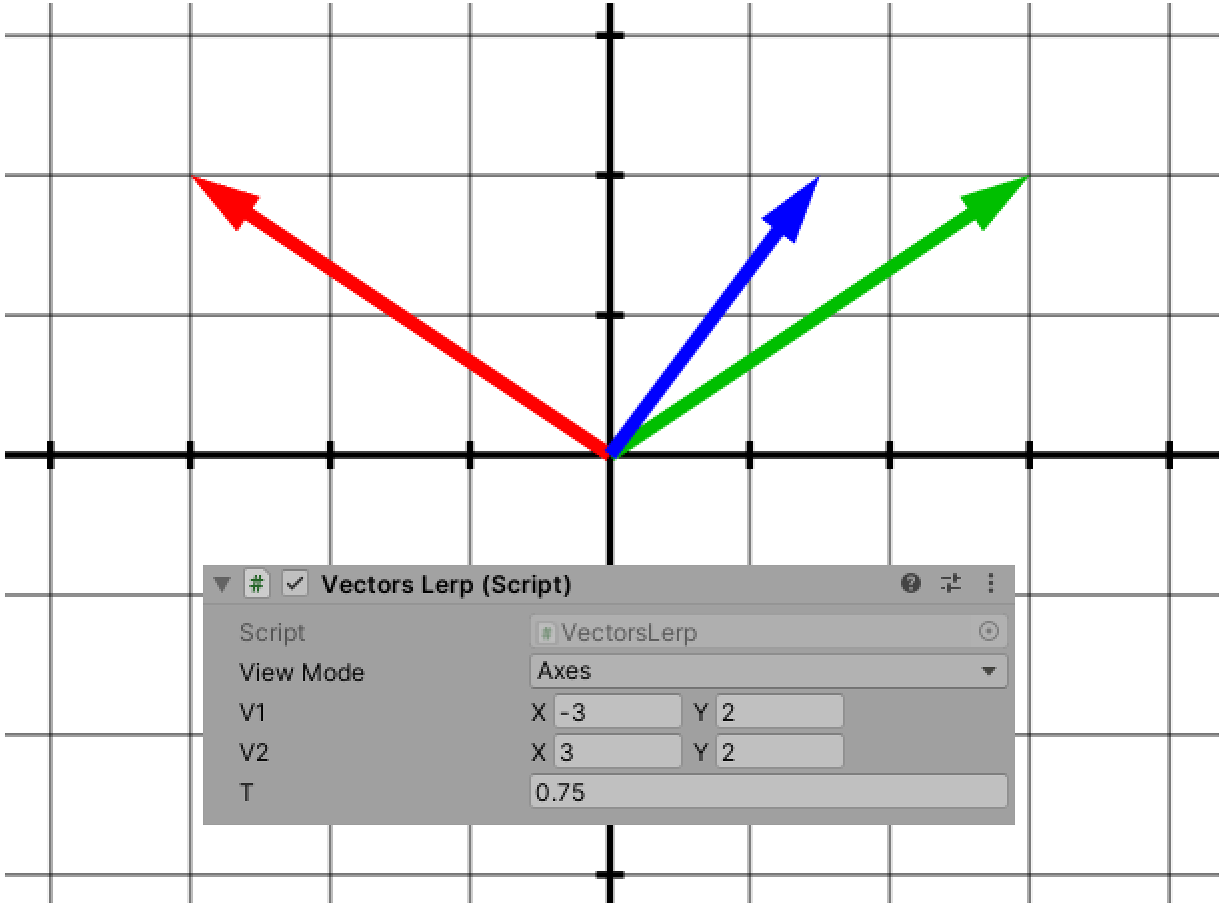
- **Linear interpolation** allows progressive change from one value into another
- The classical linear interpolation formula:

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

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

- 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

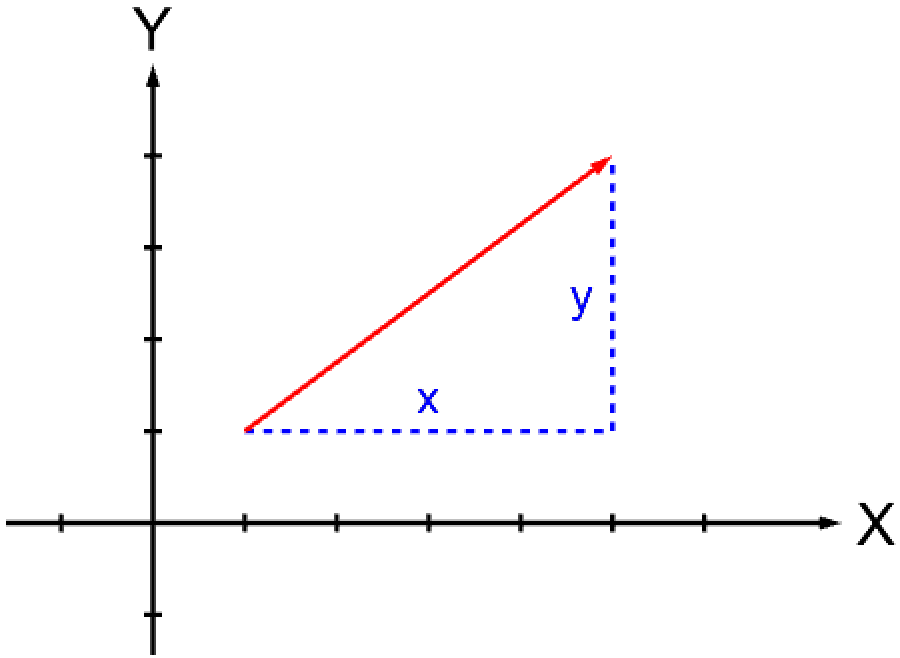
Linear Interpolation



Vector Length

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

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

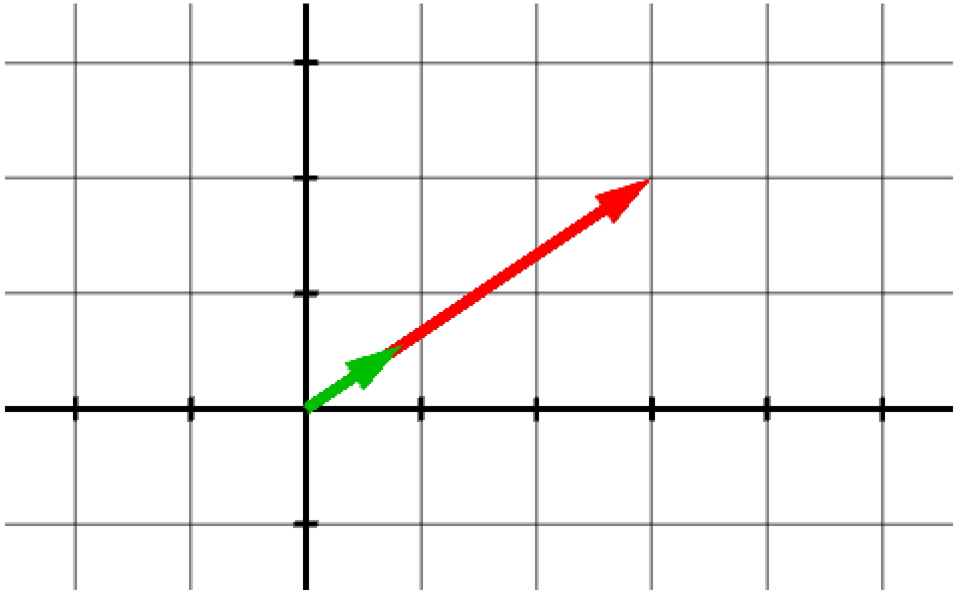


- For example, for $\vec{v} = [4, 3]$ we get:

$$|\vec{v}| \;=\; \sqrt{4 * 4 + 3 * 3} \;=\; \sqrt{16 + 19} \;=\; \sqrt{25} \;=\; 5$$

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

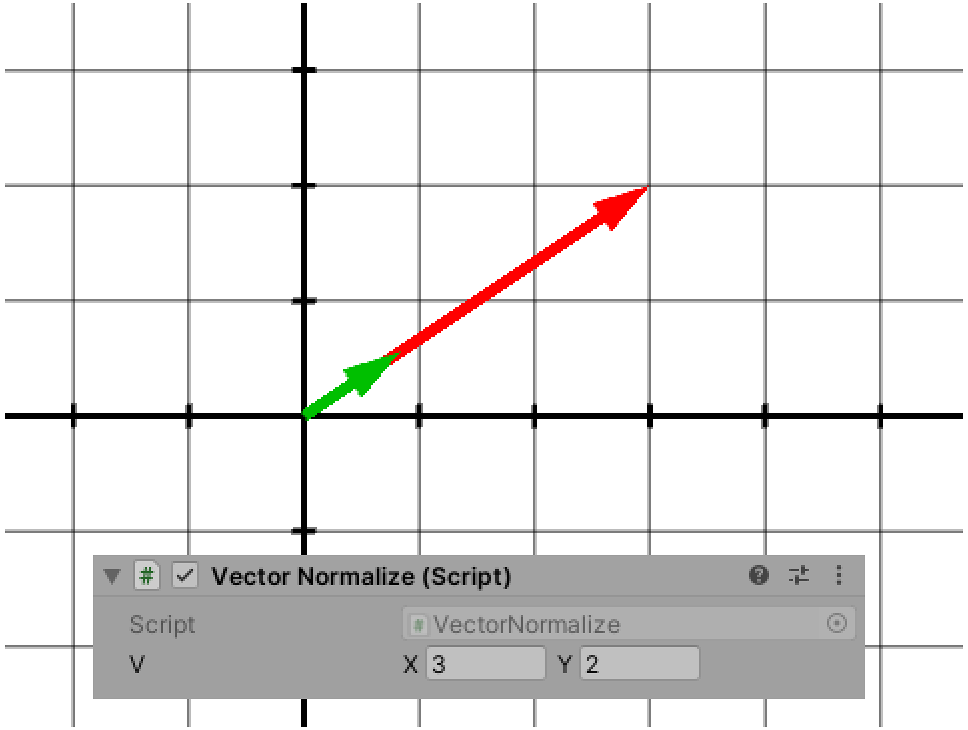
- To normalize the vector $\vec{v}$ all we need to do is to divide each vector's coordinate by the length of that vector. As a result we get a vector $\vec{u}$ with the same direction, but length of 1
- Formula for $\vec{u}$:

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

- 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



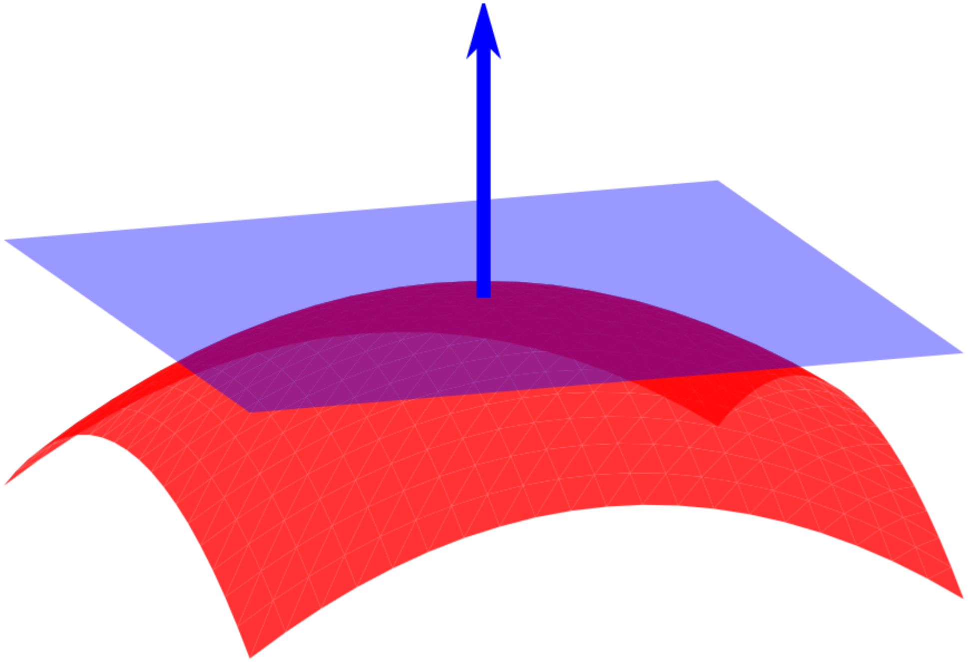
Normalization

- 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 5. We then need to compute:

$$5\frac{\vec{v}}{|\vec{v}|}$$

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



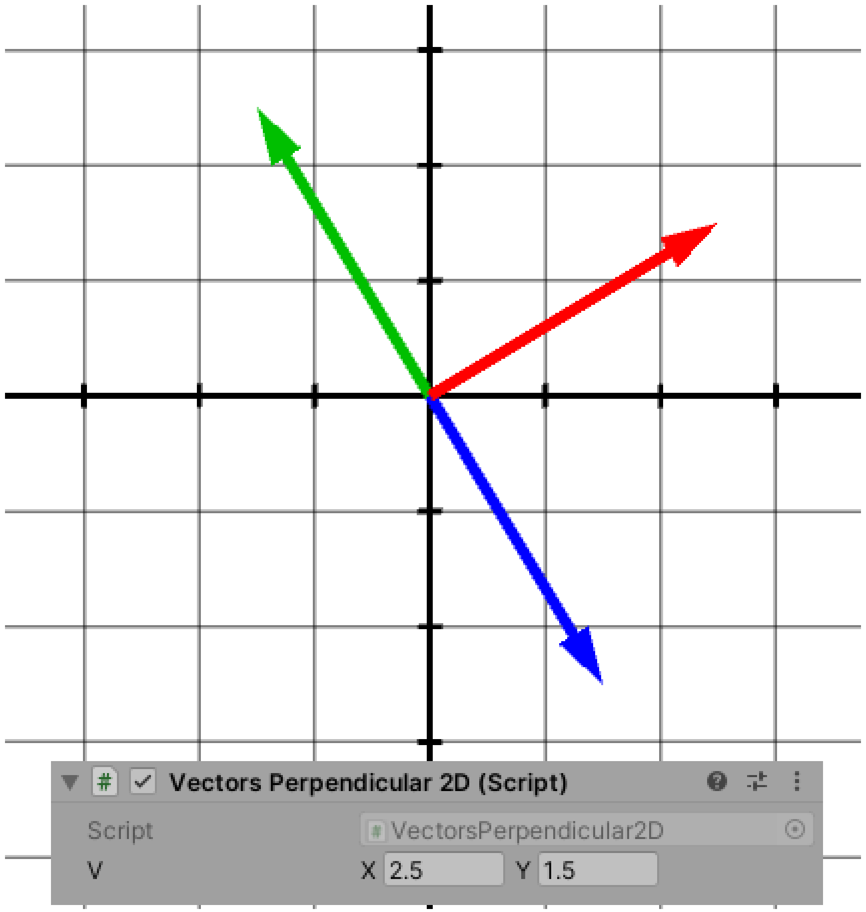
https://en.wikipedia.org/wiki/File:Surface_normal_illustration.svg **Author** Patrick87

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 $\vec{n}_2$:

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

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



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

- If $|\vec{a}| = 1$ and $|\vec{b}| = 1$, then:

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

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

$$\theta \;\; = \;\; \cos^{-1}(\vec{a} \circ \vec{b}) \;\; = \;\; \cos^{-1}(a_x b_x + a_y b_y)$$

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:

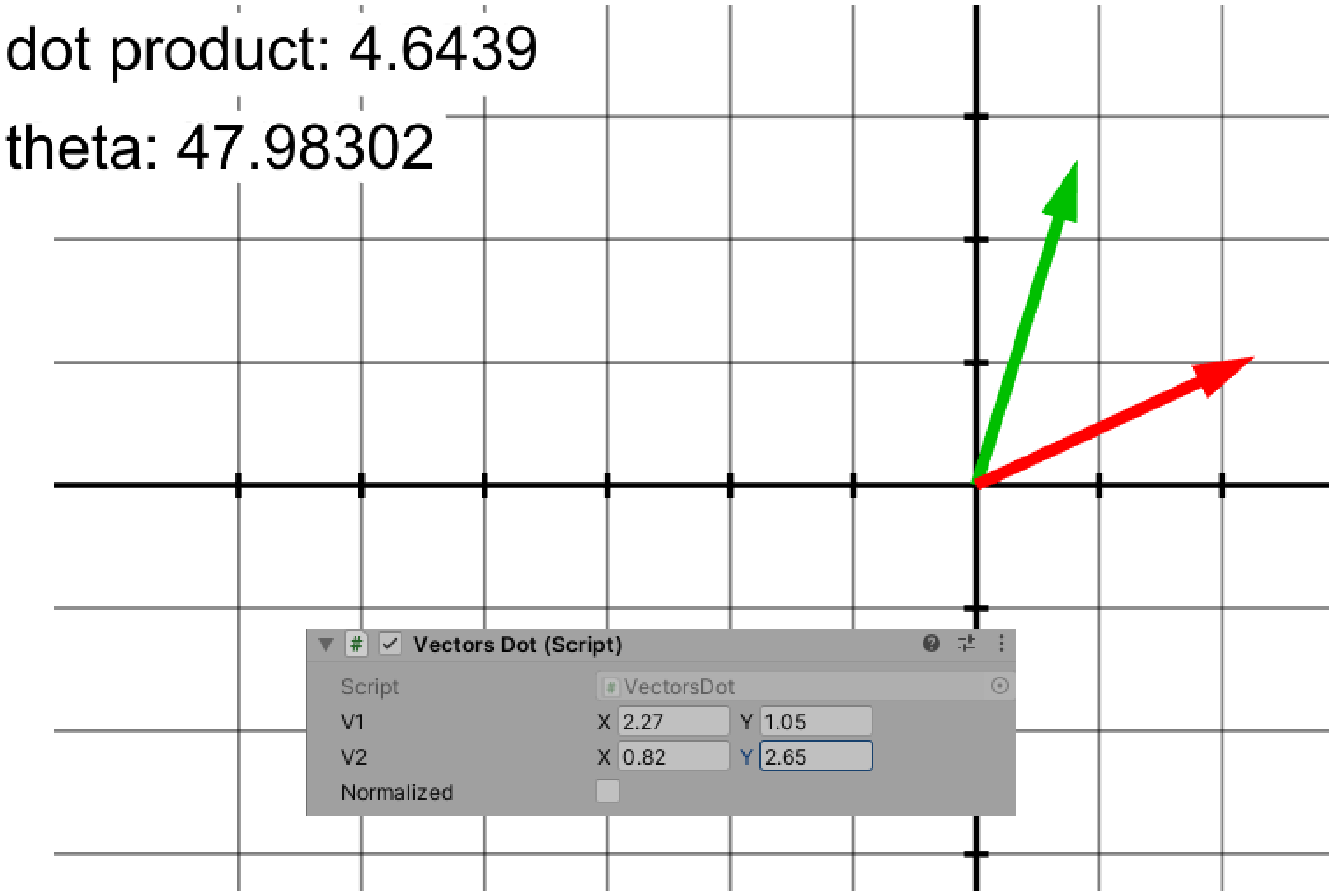
$$|\vec{a}| = \sqrt{a_x a_x + a_y a_y}$$

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

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

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

Dot Product


dot product: 4.6439
theta: 47.98302
Vectors Dot (Script)
Script
VectorsDot
V1
X 2.27
Y 1.05
V2
X 0.82
Y 2.65
Normalized

Dot Product

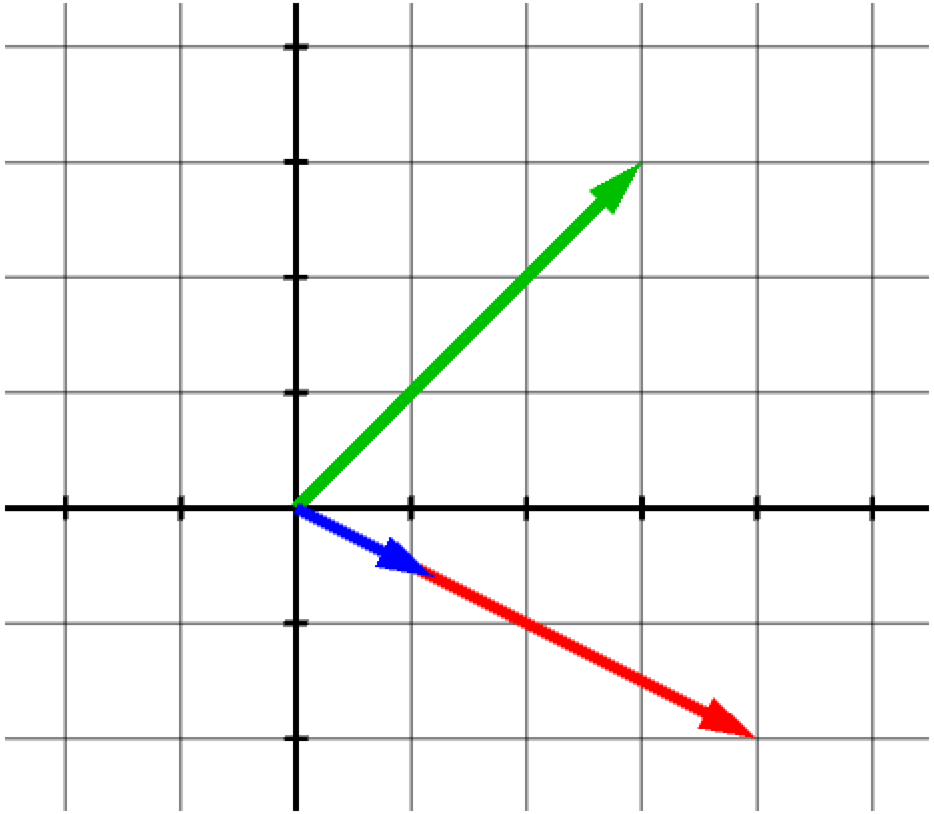
- The dot product has **very** wide applications in both 2D and 3D graphics
- When vectors are normalized their dot product lets us determine their „likeness”:

$$\begin{aligned} \vec{a} \circ \vec{b} &= -1 \quad &\rightarrow \quad &\text{opposite directon} \\ \vec{a} \circ \vec{b} &= 0 \quad &\rightarrow \quad &\text{perpendicular} \\ \vec{a} \circ \vec{b} &= 1 \quad &\rightarrow \quad &\text{the same} \end{aligned}$$

- 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
  - ...

Vector Projection

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

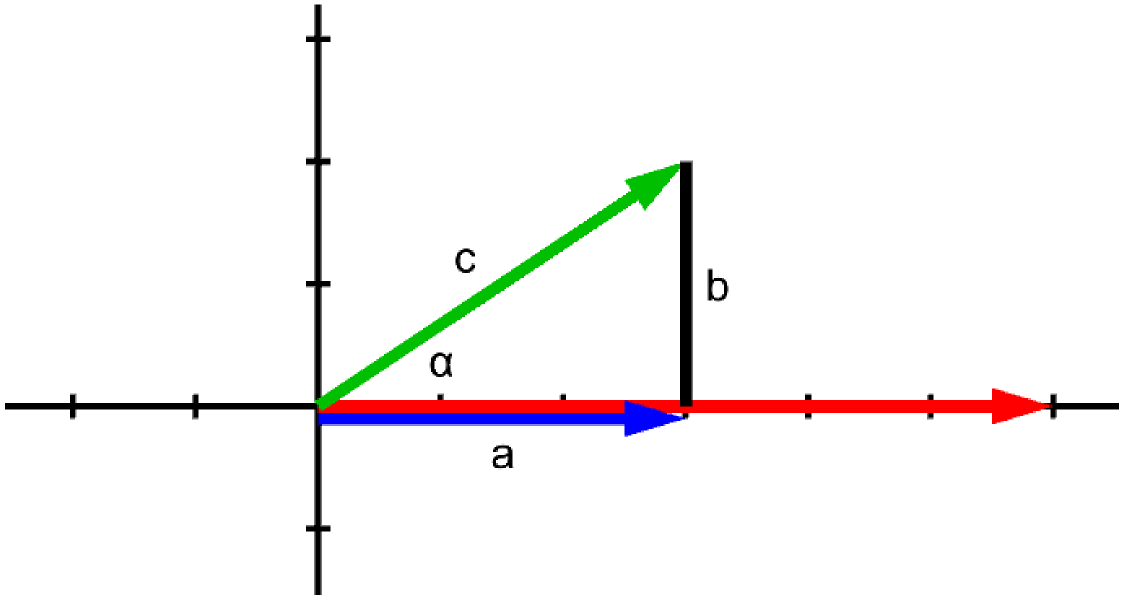


Vector Projection

- 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:

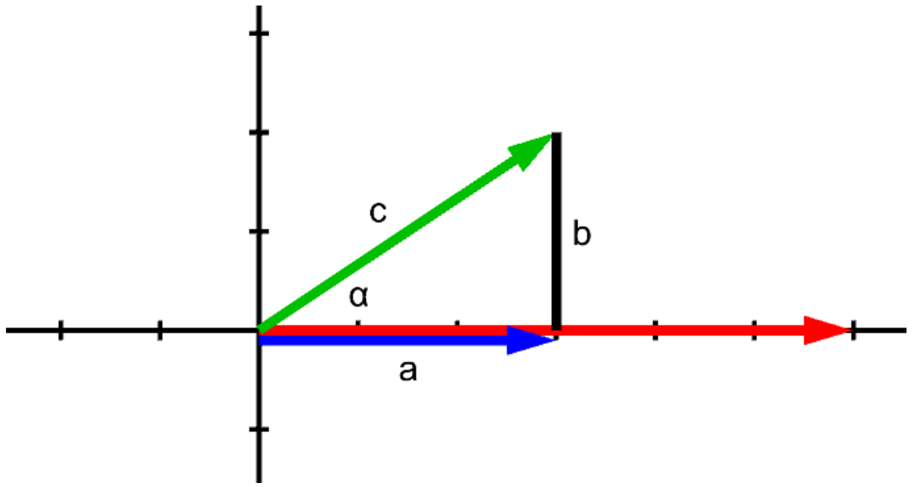
$$\cos(\alpha) = \frac{a}{c}$$

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



Vector Projection

- So:



$$\cos(\alpha) = \frac{a}{c} \qquad a = c\cos(\alpha)$$

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

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

Vector Projection

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

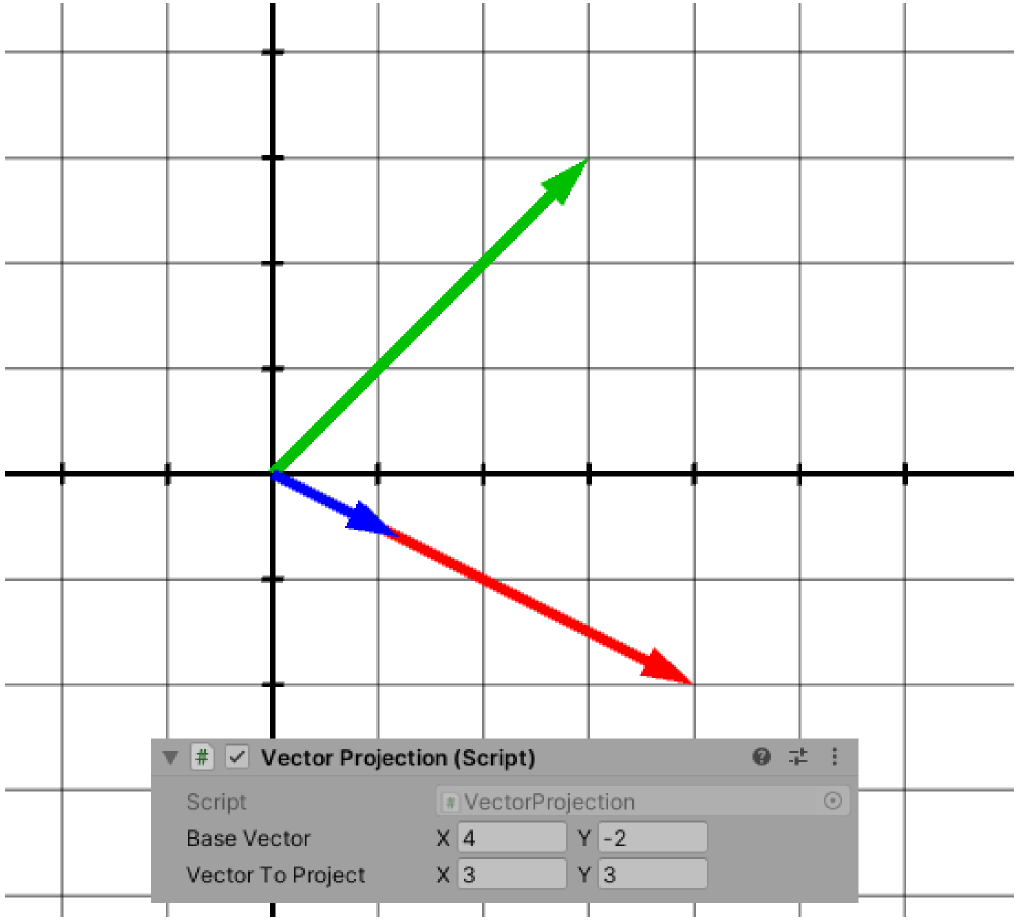
$$a = \vec{R} \circ \vec{G}$$

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

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

- The above formulas work in all dimensions

Vector Projection



Cross Product (Perpendicular Vector 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):

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

$$\begin{aligned} c_x &= a_y b_z - a_z b_y \\ c_y &= a_z b_x - a_x b_z \\ c_z &= a_x b_y - a_y b_x \end{aligned}$$

- The resulting vector $\vec{c}$ will be perpendicular to both $\vec{a}$ and $\vec{b}$
- For this reason the cross product is often used to calculate the normal vector of a surface

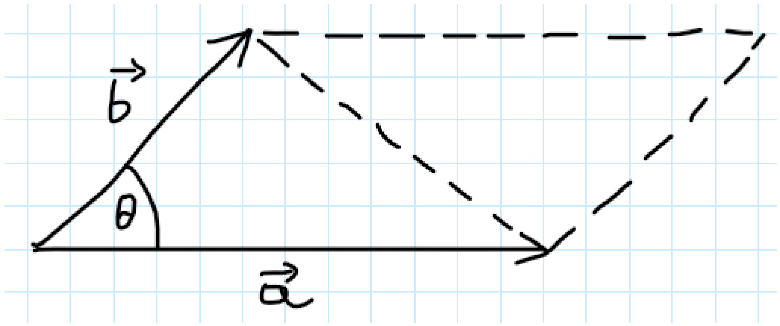
Cross Product (Perpendicular Vector in 3D)

- The length of the resulting vector is:

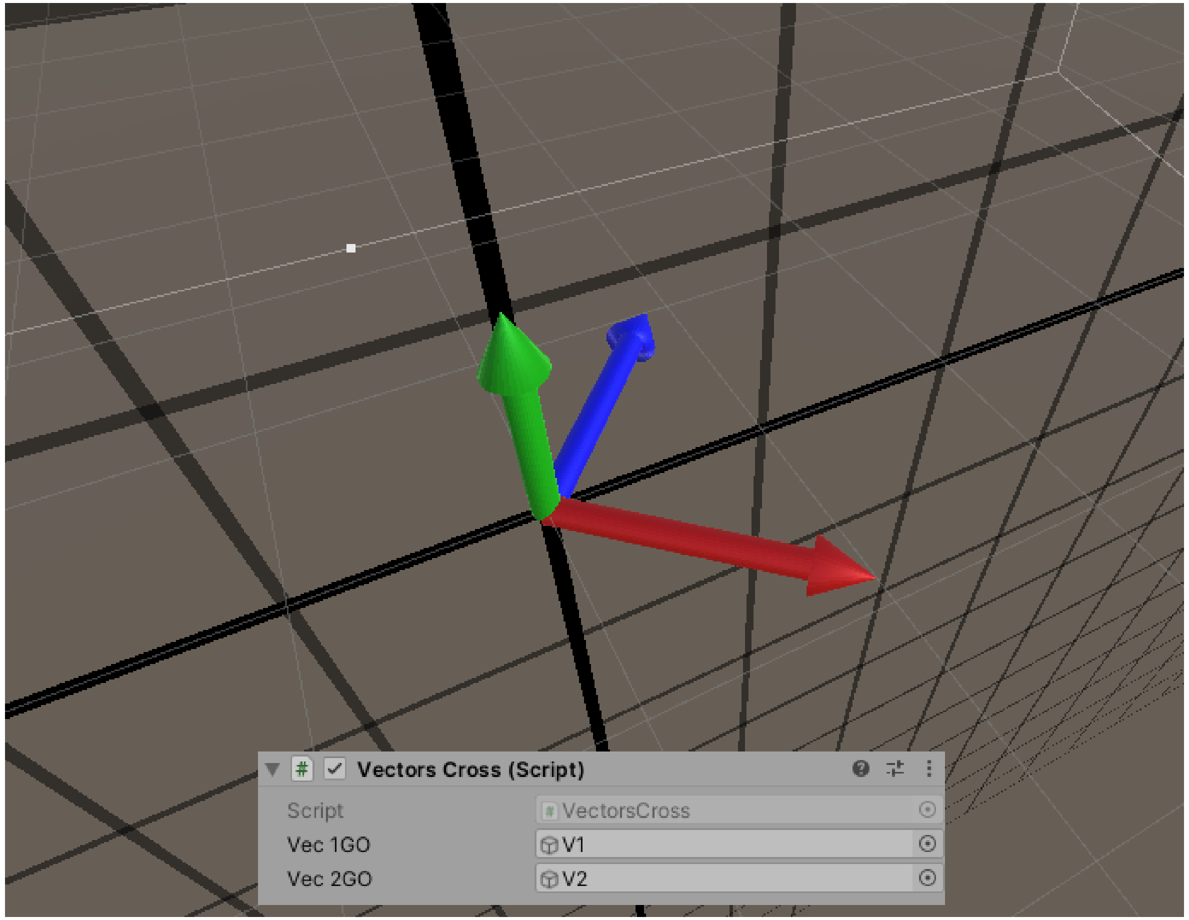
$$|\vec{c}| = |\vec{a}||\vec{b}|\sin(\theta)$$

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 $\vec{b}$:



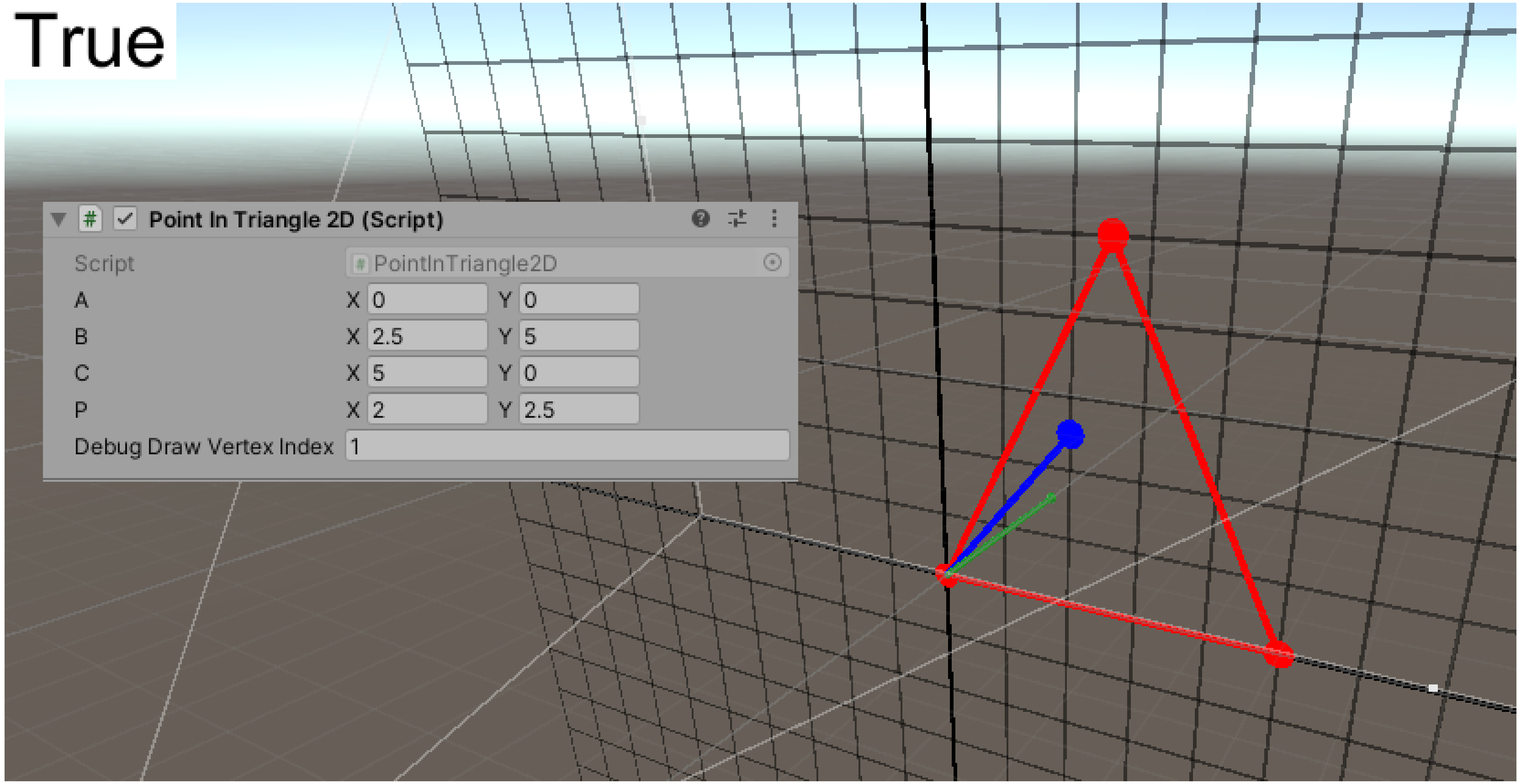
Cross Product (Perpendicular Vector in 3D)



Point in Triangle (in 2D)

- Cross product can be used to determine on which side of one vector the other vector is located
- Thanks to this property of cross product we can check if a point is inside a triangle (or any convex polygon)

Point in Triangle (in 2D)



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:

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

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

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

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

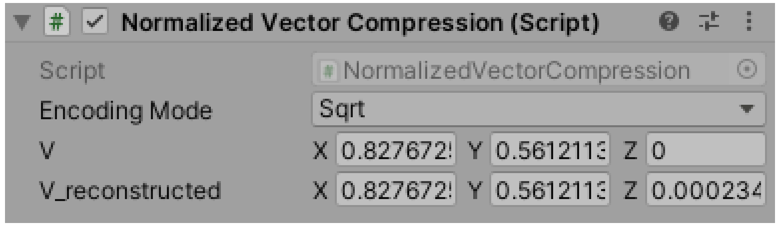
Normalized Vector Compression

- 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

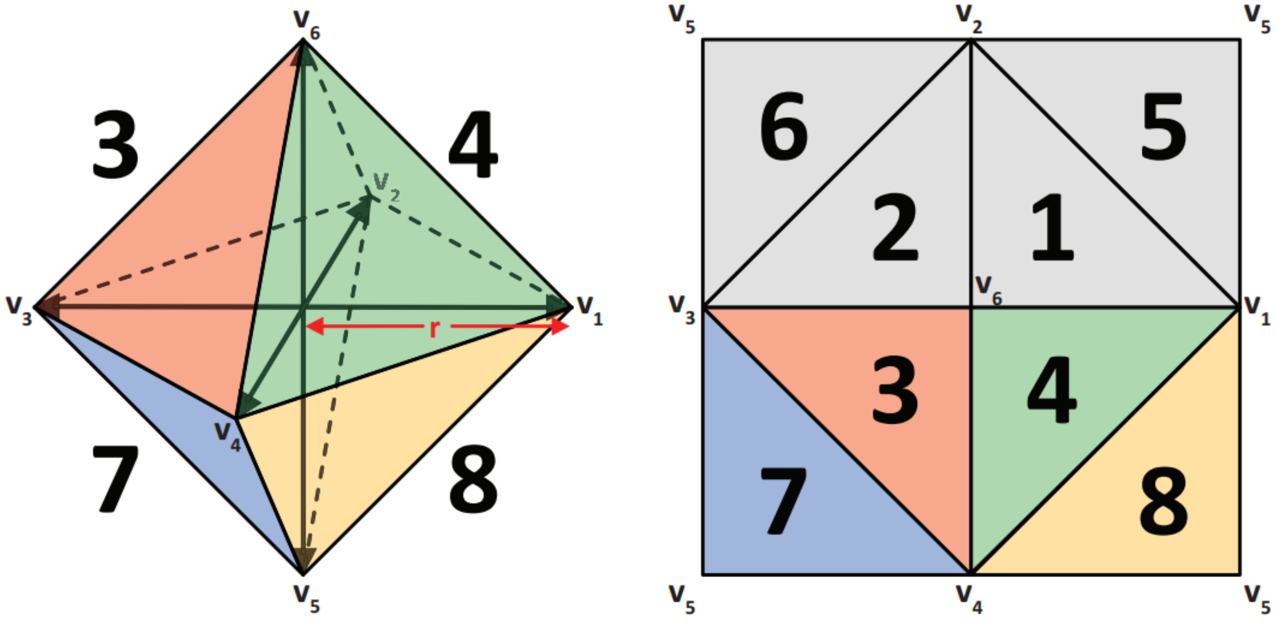


Normalized Vector Compression

- In the algorithm that we have just used, the need to store the sign separately is cumbersome
- There are in fact many alternatives for compressing/encoding a normalized vector
- One of them are **octahedron vectors**

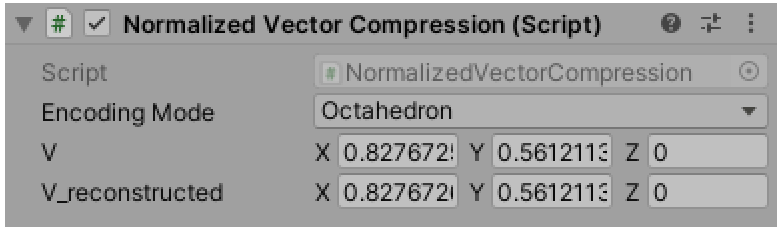
Normalized Vector Compression

- Illustration:



Octahedron Environment Maps, 2008
Engelhardt and Dachsbacher

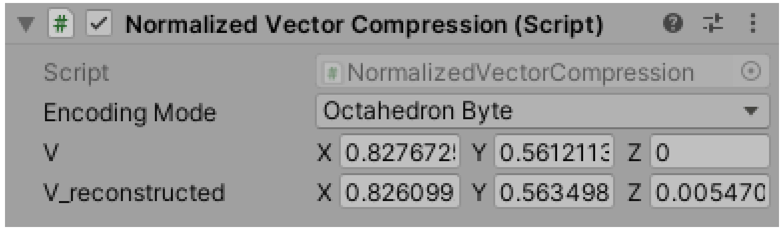
Normalized Vector Compression



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

- 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. In many cases this will be good enough

Exercises

1. Write a program that calculates the area of a parallelogram (slide 29)
2. Implement normalized vector compression using the algorithm with the square root and `byte` type (slide 36)