Sign in here!

fsu.devlup.org/signin

3D Math

DevLUp FSU

GBM #11

November 21th, 2024

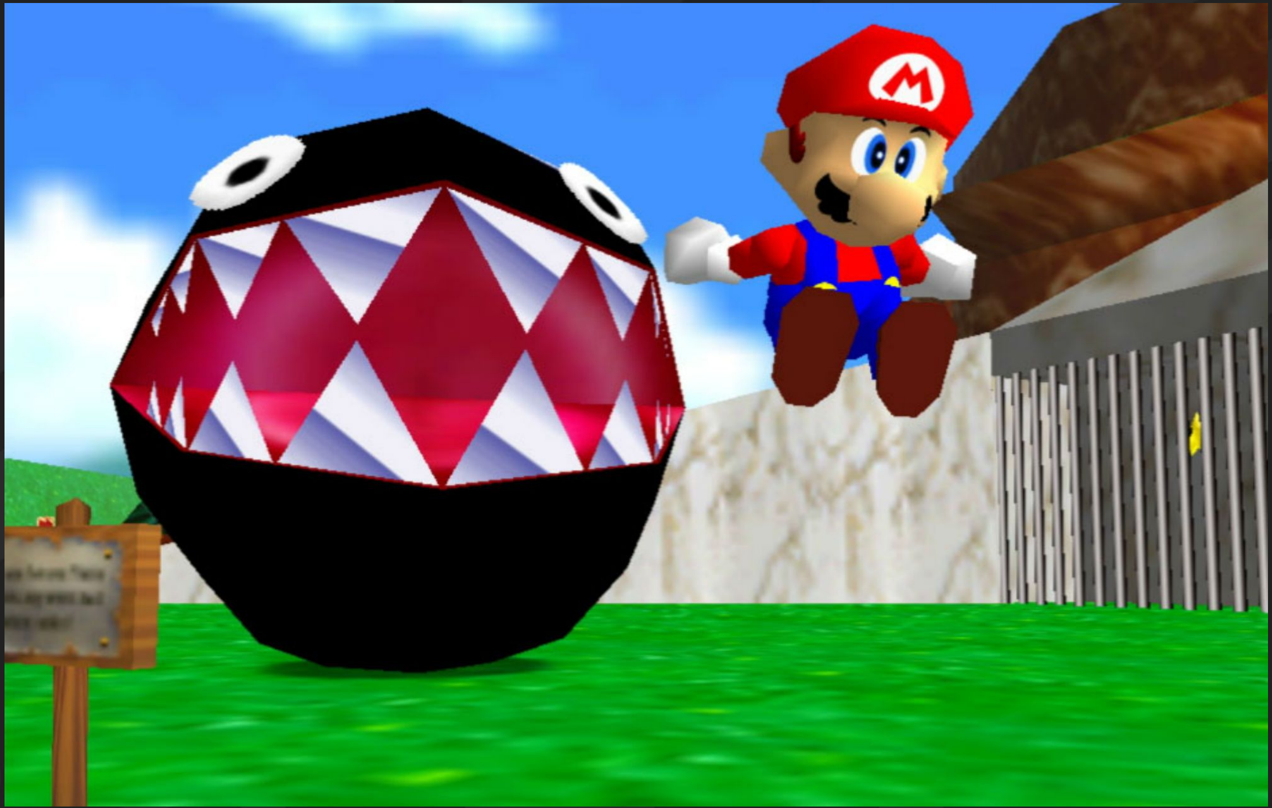
Welcome!

Next Few Weeks

Date	Week #	GBM Title	Secondary Event	Presenter
29 Aug	1	(No Meeting)	Involvement Fair	
5 Sep	2	Intro to Club and New Club Project		Club
12 Sep	3	Intro to Game Design		Chris
19 Sep	4	Intro to 3D Game Dev in Godot		Dion
26 Sep	5	Intro to 3D Modelling in Blender		Jake, Parker, Emma
3 Oct	6	Blender Animations		Ares
10 Oct	7	Blender Materials		Parker, Jake
17 Oct	8	Pixel Art		Ares, Emma
24 Oct	9	Tile Maps		Jake, Ares
31 Oct	10	Spooky Game Night Social	CANDY FOR ALL (No Candy)	Jack Skellington
7 Nov	11	Writing for Games		Emma, Chris
14 Nov	12	UI Design		Emma, Jake
21 Nov	13	3D Math		Dion
28 Nov	14	Thanksgiving Break		
5 Dec	15	Goodbye Chris Social		Chris
12 Dec	16	Finals		

#👀showoff **recap**

So… *YOU* want to make a 3D Game?



What is a vector?

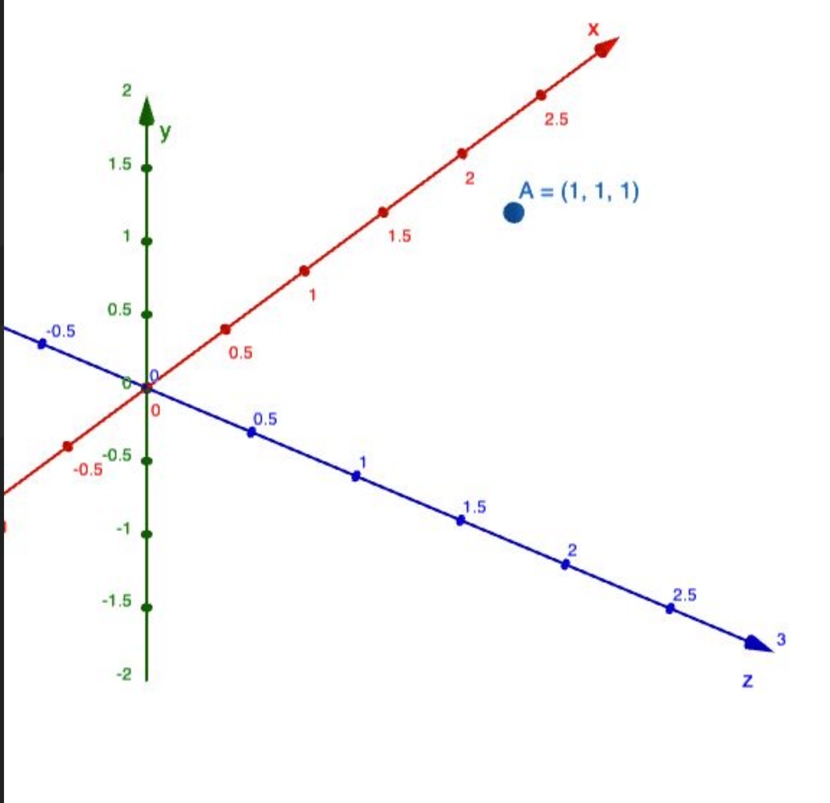
- Sometimes we need more than just one number (a scalar) to express a quantity.

- In 2D, we can use a **Vector2**.
  In 3D, a **Vector3**.

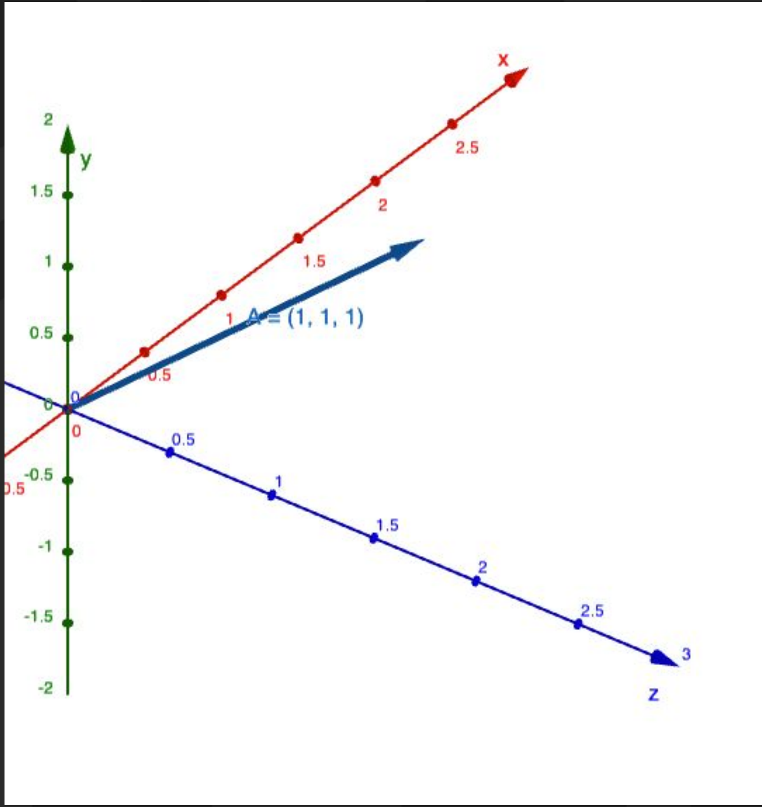
- Godot hides a lot of the complex math in these types to make your life easier!

This movie was 14 years ago…

Can represent a point in space,

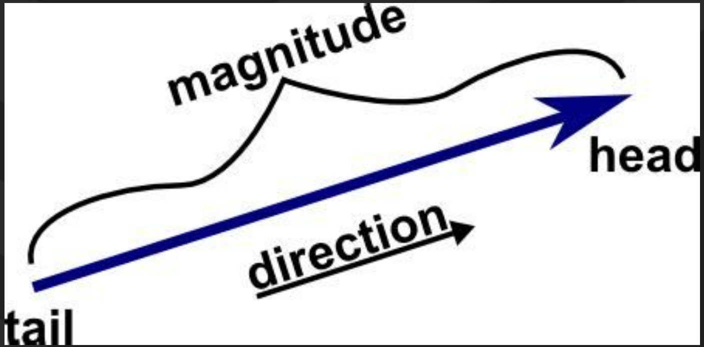


but also a direction.



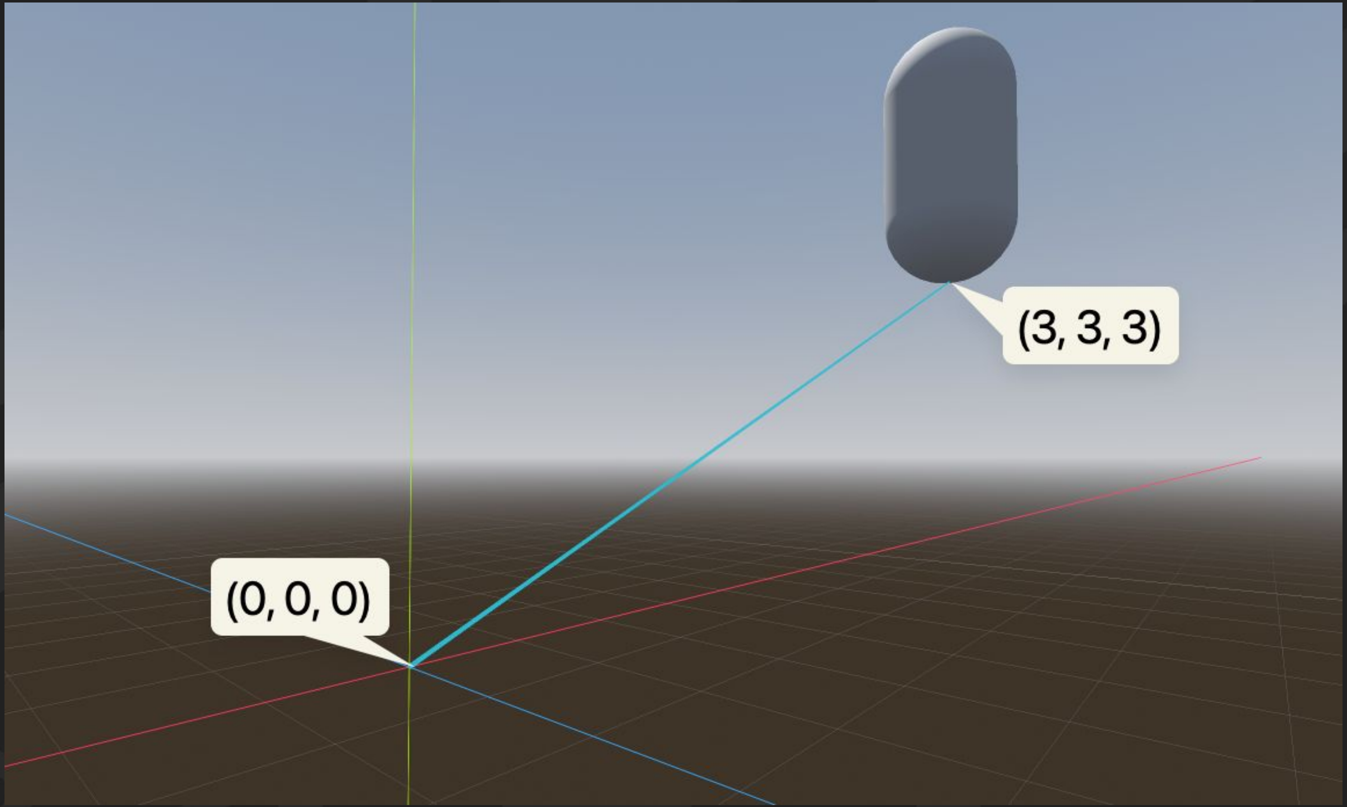
What is a vector?

- "A geometric object that has a magnitude (or length) and direction."
  —Wikipedia



- We usually write just the head.
  E.g., Vector3(3, 3, 3)
- The tail's position is always* (0, 0, 0).

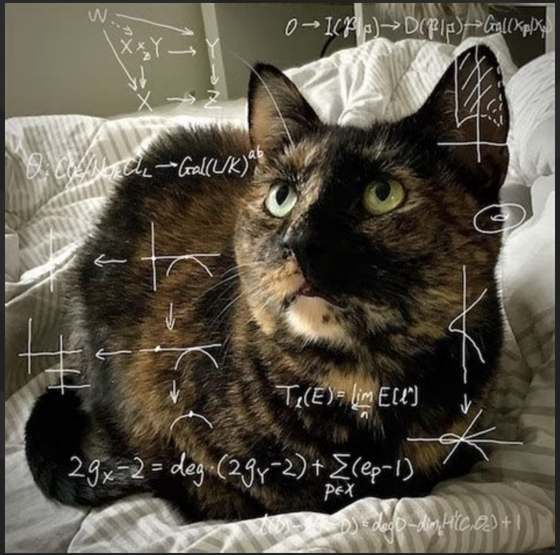
What is a vector?



What can we do with vectors?

- Add
- Negate
- Scale
- Find Magnitude (or Length)
- Normalize
- Angle between vectors
- Perpendicular vector between vectors

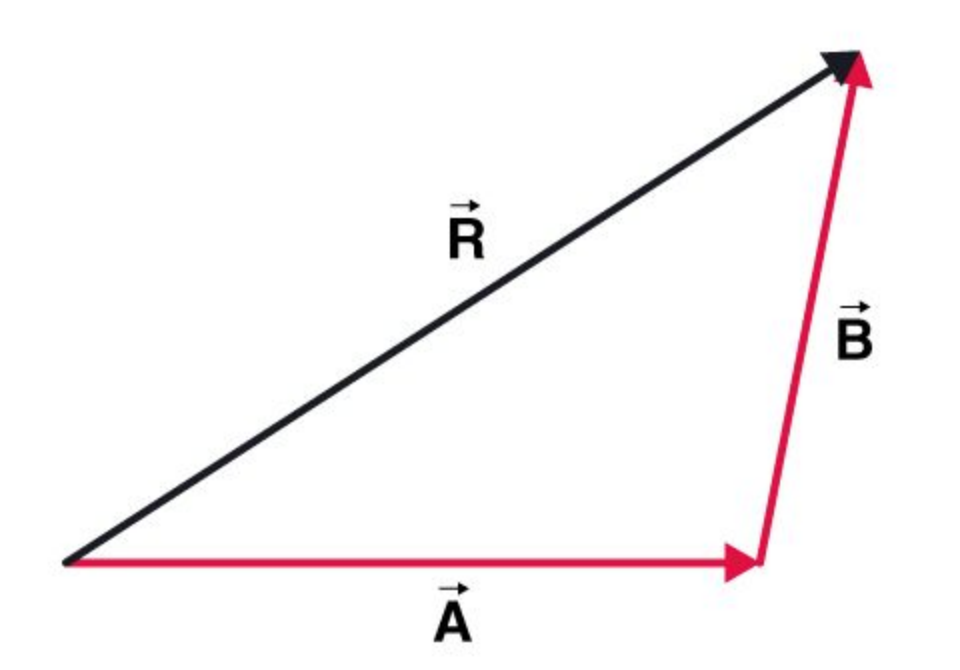
- Rotate…?



Adding Vectors

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \end{bmatrix}$$

Adding Vectors Geometrically



Moving an Object

```
func _physics_process(_delta: float) -> void:
	global_position += Vector3(0.1, 0, 0)
```

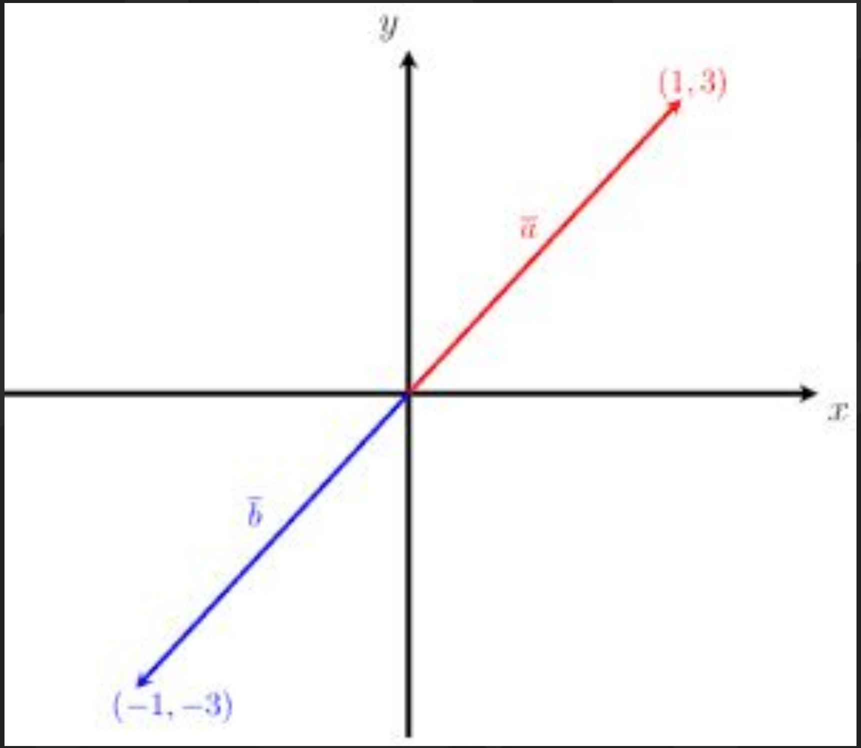
Live Demo


Test (DEBUG)
Camera Position: (0, 0, 4.96779)
Tracked Position: (7.199996, 0, 0)

Negating Vectors

$$-\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -x_1 \\ -x_2 \\ -x_3 \end{bmatrix}$$

Negating Vectors Geometrically



Moving Backwards

```
3 func _physics_process(_delta: float) -> void:
4     global_position += -Vector3(0.1, 0, 0)
```

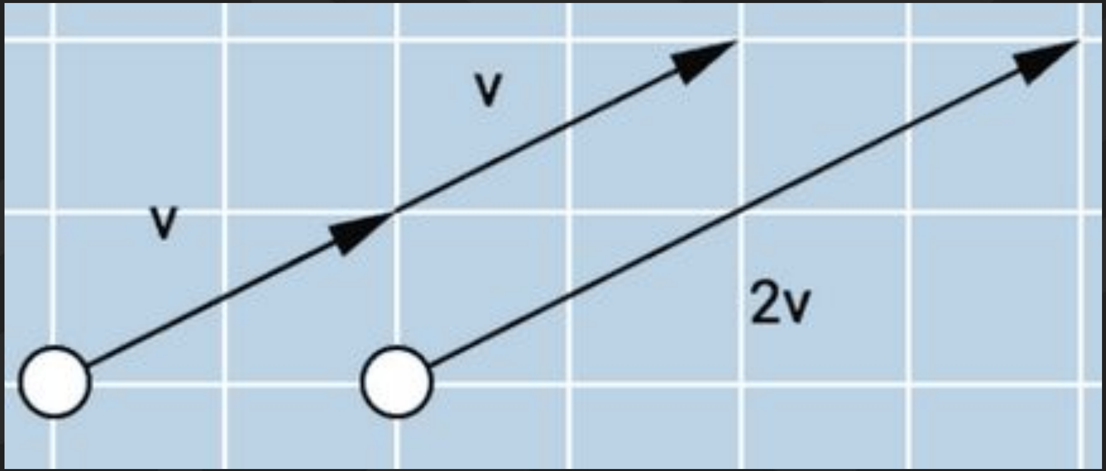
Live Demo

Demo (DEBUG)
Camera Position: (0, 0, 4.96779)
Tracked Position: (-12.80001, 0, 0)

Scaling Vectors

$$2\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \\ 2x_3 \end{bmatrix}$$

Scaling Vectors Geometrically



Moving with a Set Speed

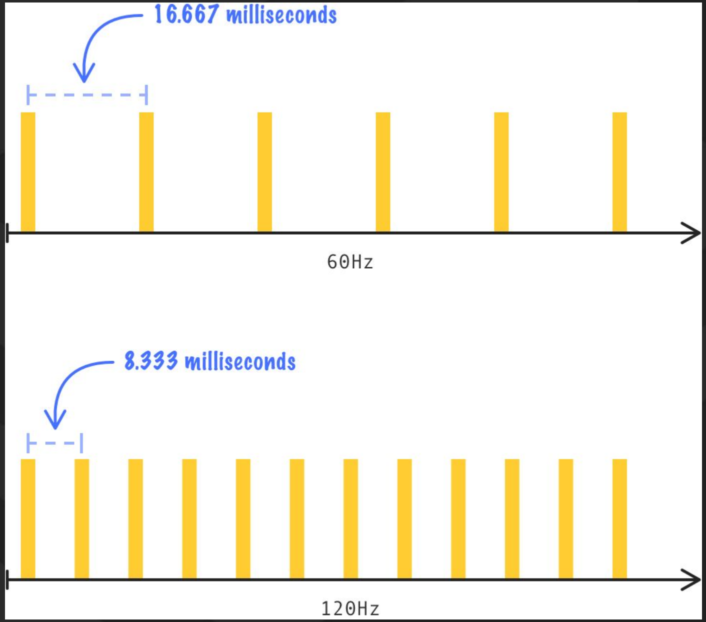
```
const SPEED = 3

func _physics_process(_delta: float) -> void:
	global_position += Vector3(1, 0, 0) * SPEED
```

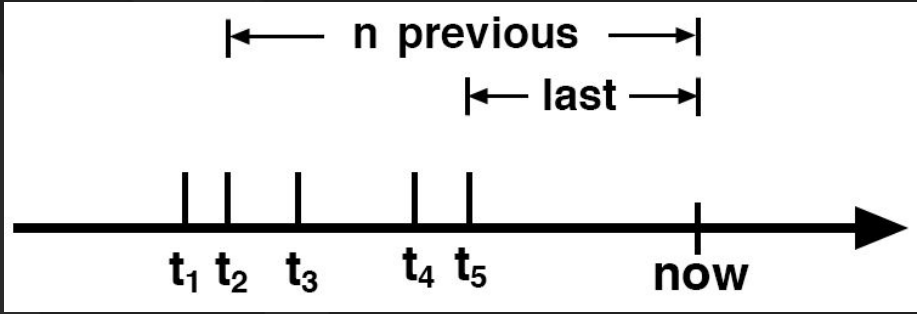
An Aside on Delta Time

- The amount of time, in seconds, since the last frame.

- 120 fps => 0.00833… s
- 60 fps => 0.0166… s
- 30 fps => 0.0333… s

- Everything happening every process() should use delta.





Moving with a Set Speed Per Second

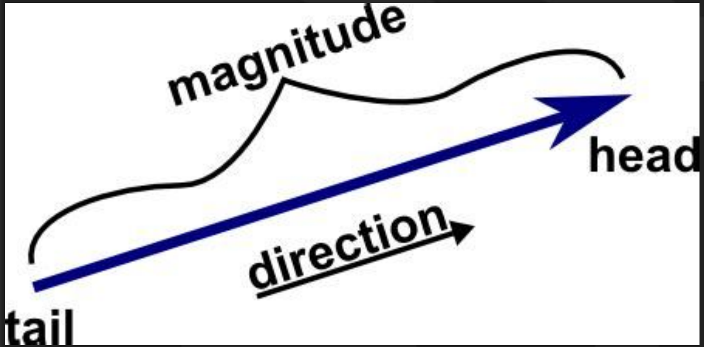
```
const SPEED = 3

func _physics_process(delta: float) -> void:
	global_position += Vector3(1, 0, 0) * SPEED * delta
```

Live Demo


Test (DEBUG)
Camera Position: (0, 0, 4.96779)
Tracked Position: (7.199996, 0, 0)

Magnitude (or Length) of a Vector

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



Moving Diagonally

```
const SPEED = 3

func _physics_process(delta: float) -> void:
	global_position += Vector3(1, 1, 0) * SPEED * delta


func _process(delta: float) -> void:
	print(global_position.length())
```

Using length_squared()

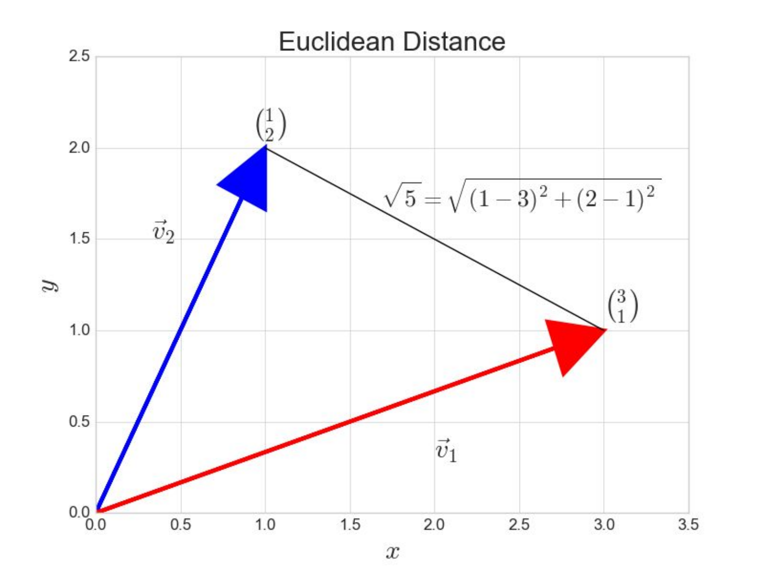
- Avoids costly sqrt() function.

- Useable for comparisons.

```
const RANGE = 5
if v.length_squared() <= RANGE * RANGE:
	print("In Range")
```

Distance Between Positions

$$\text{dist}(u, v) = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2 + (v_z - u_z)^2}$$



Godot Makes This Easy

```
var v = Vector3(1, 2, 3)
var u = Vector3(4, 5, 6)

v.distance_to(u)
v.distance_squared_to(u)
```

Normalize a Vector

- Sometimes we want a vector to represent just a direction.
- We can *normalize* it, meaning make its magnitude = 1.
- Resulting vector is called a *unit vector*.

$$v_{\text{normalized}} = \frac{1}{||v||} v$$

Moving Diagonally FIXED!

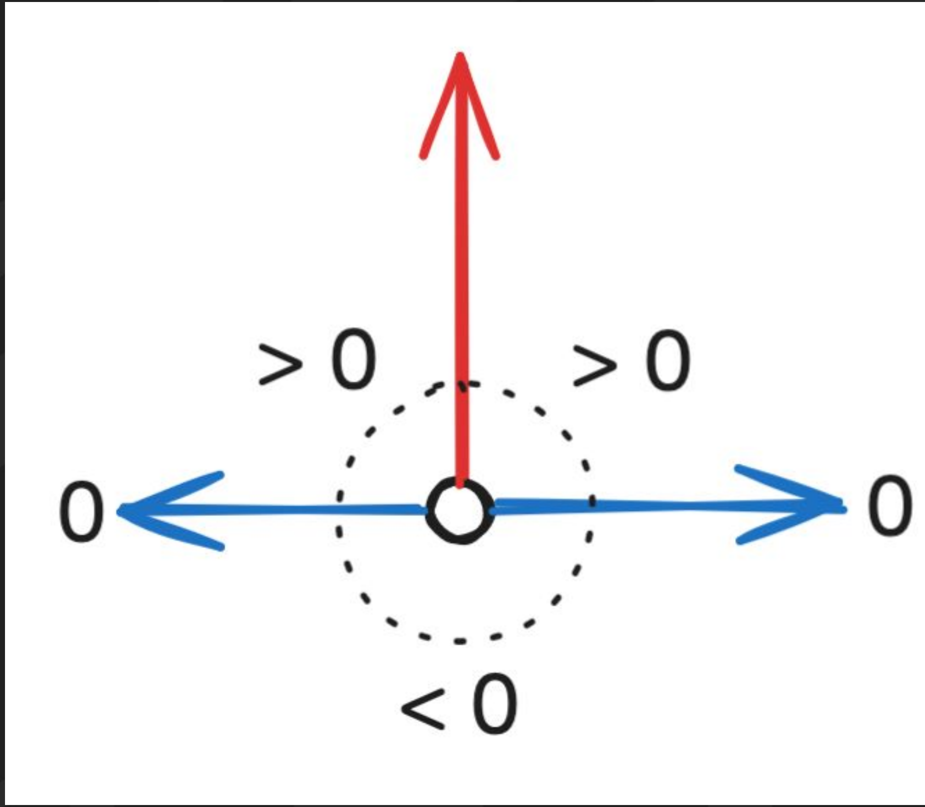
```
const SPEED = 3

func _physics_process(delta: float) -> void:
	global_position += Vector3(1, 1, 0).normalized() * SPEED * delta
```

Angle Between Vectors (Dot Product)

- Also called the Inner Product.
- 0 if perpendicular, positive if angle < 90, negative if angle > 90.

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z$$

Angle Between Vectors (Dot Product)



Can the Enemy See the Player?

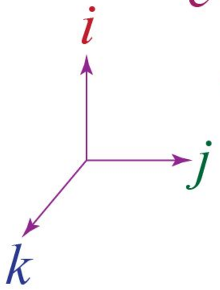
```
var enemy_forward = -tracked_node.transform.basis.z
var direction_to_camera = tracked_node.global_position.direction_to(global_position)

if enemy_forward.dot(direction_to_camera) > 0:
	%TrackedPosition.text += " Sees Player"
```

Perpendicular Vector Between Vectors (Cross Product)

$$\vec{a} = a_1\vec{i} + a_2\vec{j} + a_3\vec{k}$$

$$\vec{b} = b_1\vec{i} + b_2\vec{j} + b_3\vec{k}$$

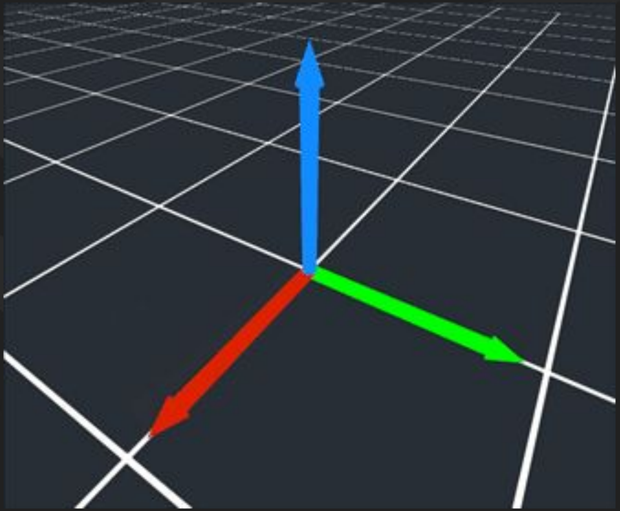
$$\vec{c} = \vec{a} \times \vec{b}$$



$$= \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

$$= i\begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} - j\begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + k\begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

$$= i\left|a_2 b_3 - a_3 b_2\right| - j\left|a_1 b_3 - a_3 b_1\right| + k\left|a_1 b_2 - a_2 b_1\right|$$



Godot makes this *much* easier!

```
Vector3 cross(with: Vector3) const
```

Returns the cross product of this vector and `with`.

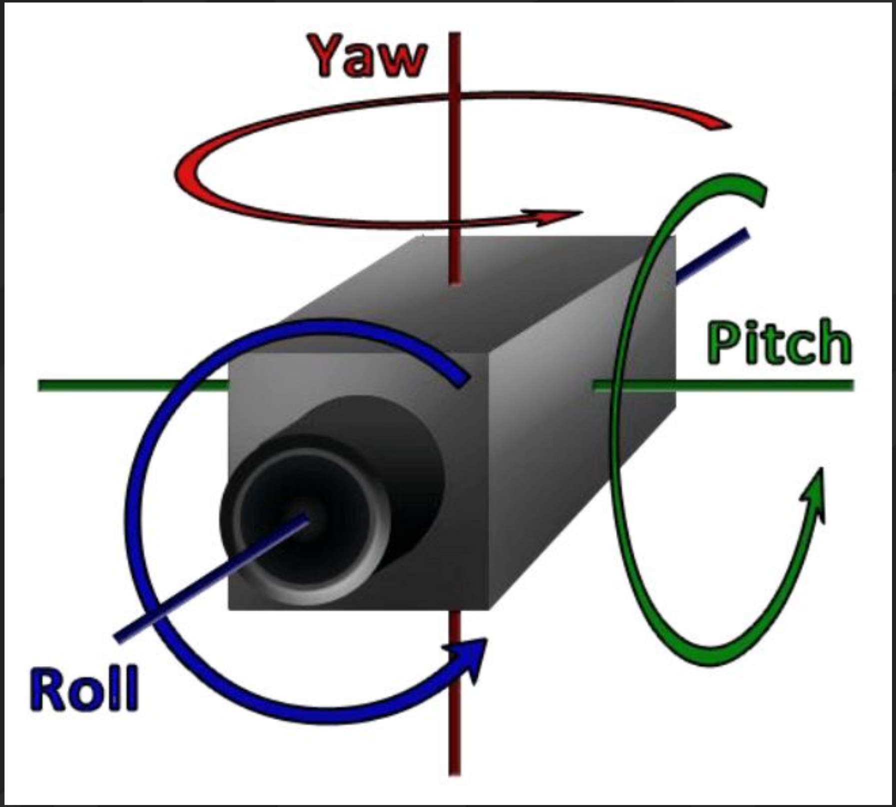
This returns a vector perpendicular to both this and `with`, which would be the normal vector of the plane defined by the two vectors. As there are two such vectors, in opposite directions, this method returns the vector defined by a right-handed coordinate system. If the two vectors are parallel this returns an empty vector, making it useful for testing if two vectors are parallel.

Use Cases?

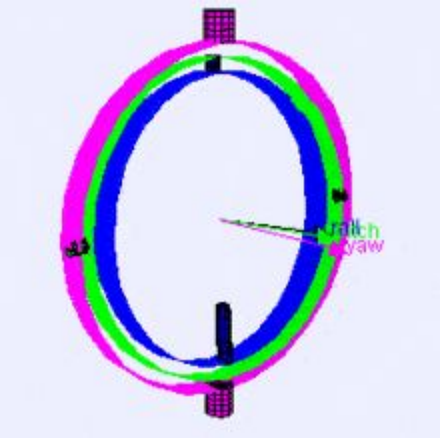
- Calculating the normal vector of a plane
- Lighting calculations
- Physics engine calculations
- Inverse kinematics calculations

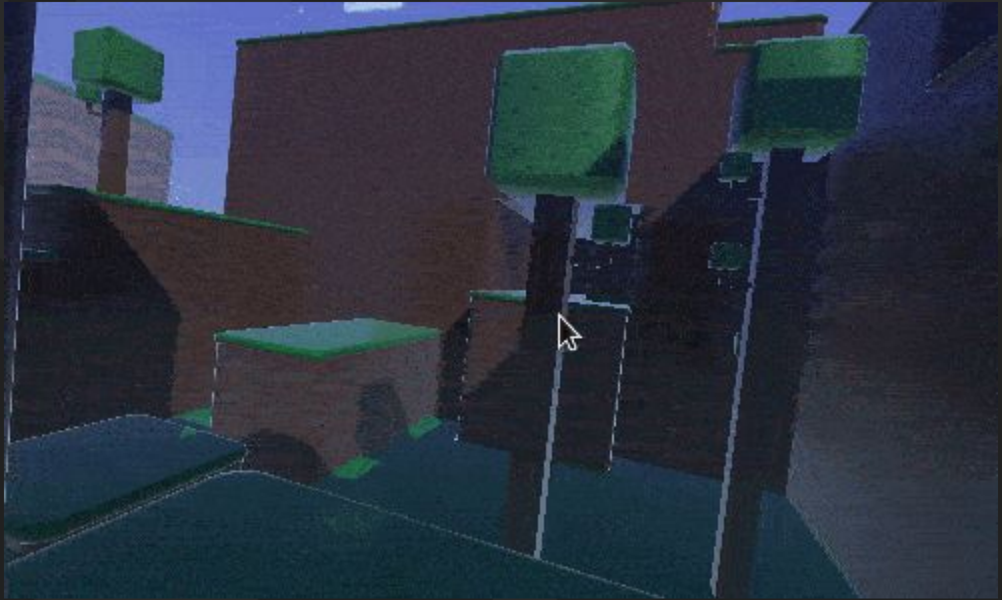
- Mostly stuff that game engines do for you

Representing Rotation in a Vector?

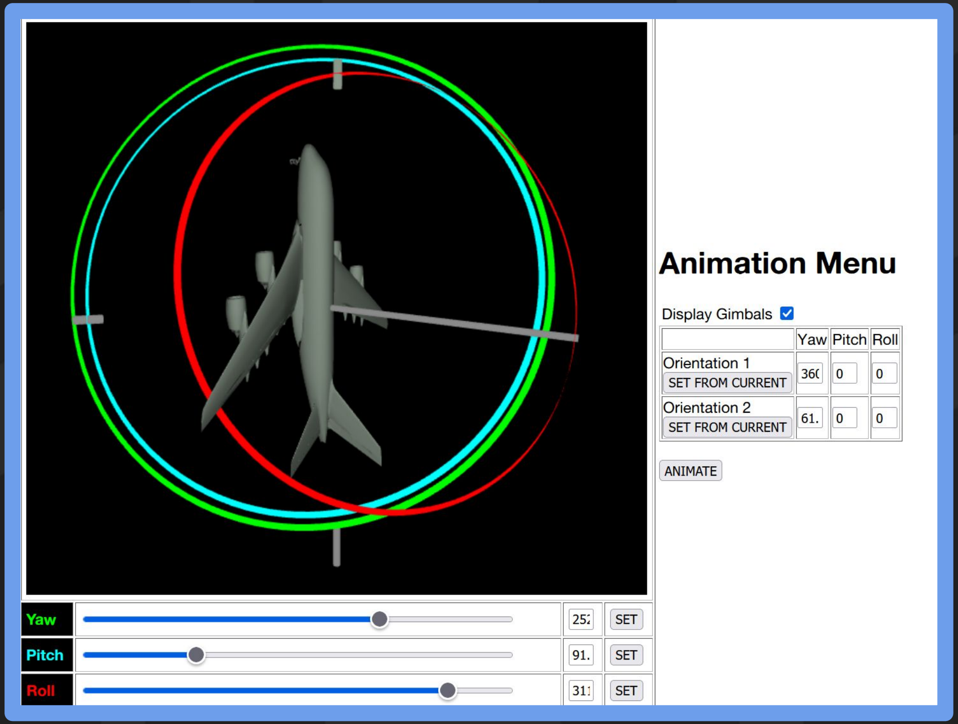


Axis Order Matters





Gimbal Lock


Animation Menu
Display Gimbals
Yaw
Pitch
Roll
Orientation 1
SET FROM CURRENT
36(
0
0
Orientation 2
SET FROM CURRENT
61.
0
0
ANIMATE
Yaw
25:
SET
Pitch
91.
SET
Roll
31:
SET

Say NO to Euler Angles

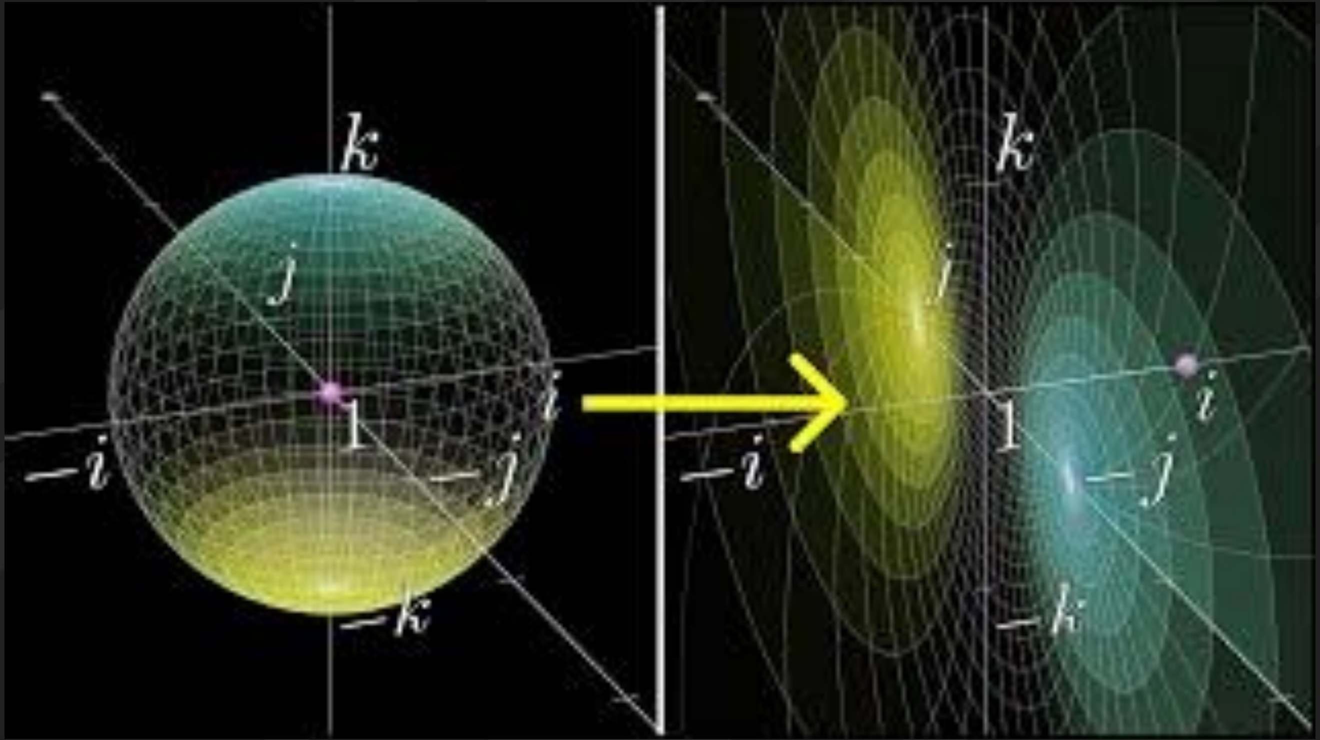
Say no to Euler angles

The result of all this is that you should **not use** the `rotation` property of Node3D nodes in Godot for games. It's there to be used mainly in the editor, for coherence with the 2D engine, and for simple rotations (generally just one axis, or even two in limited cases). As much as you may be tempted, don't use it.

Quaternions

"Quaternions… though beautifully ingenious, have been an **unmixed evil** to those who have touched them in any way, including Clerk Maxwell."
—Lord Kelvin

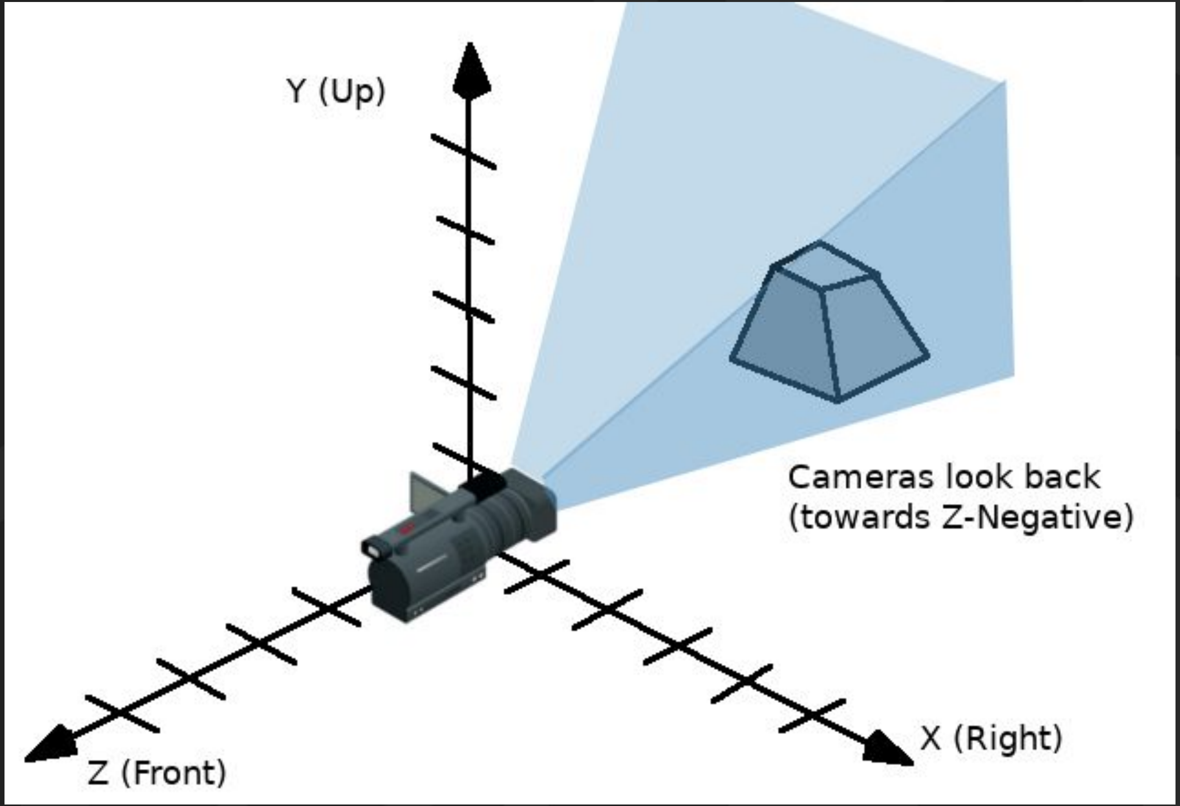
Quaternions Video by 3blue1brown



Use Transform3D Instead

The **Transform3D** built-in Variant type is a 3×4 matrix representing a transformation in 3D space. It contains a Basis, which on its own can represent rotation, scale, and shear. Additionally, combined with its own origin, the transform can also represent a translation.

What are Basis Vectors?




```
transform.basis.x
transform.basis.y
transform.basis.z
```

Rotating a Transform3D

```
var axis = Vector3(1, 0, 0) # Or Vector3.RIGHT
var rotation_amount = 0.1
# Rotate the transform around the X axis by 0.1 radians.
transform.basis = Basis(axis, rotation_amount) * transform.basis
# shortened
transform.basis = transform.basis.rotated(axis, rotation_amount)
```

Rotating a Transform3D

```
# Rotate the transform around the X axis by 0.1 radians.
rotate(Vector3(1, 0, 0), 0.1)
# shortened
rotate_x(0.1)
```

Use Quaternions for Interpolation

```
# Convert basis to quaternion, keep in mind scale is lost
var a = Quaternion(transform.basis)
var b = Quaternion(transform2.basis)
# Interpolate using spherical-linear interpolation (SLERP).
var c = a.slerp(b,0.5) # find halfway point between a and b
# Apply back
transform.basis = Basis(c)
```

Putting it together: How does the free camera work?

Exit Survey:

Fig. 1: *Homer dislikes exit surveys.*