GDC | March 20-24, 2023
San Francisco, CA

Mobile Math: From ATan2 to Gyro Calibration

Patrick Martin
Developer Relations Engineer, Google

Who am I?

- Mobile games at Venan Entertainment
- Games for toys at Sphero
- DevRel in games at Firebase, Now Android

Why this talk?

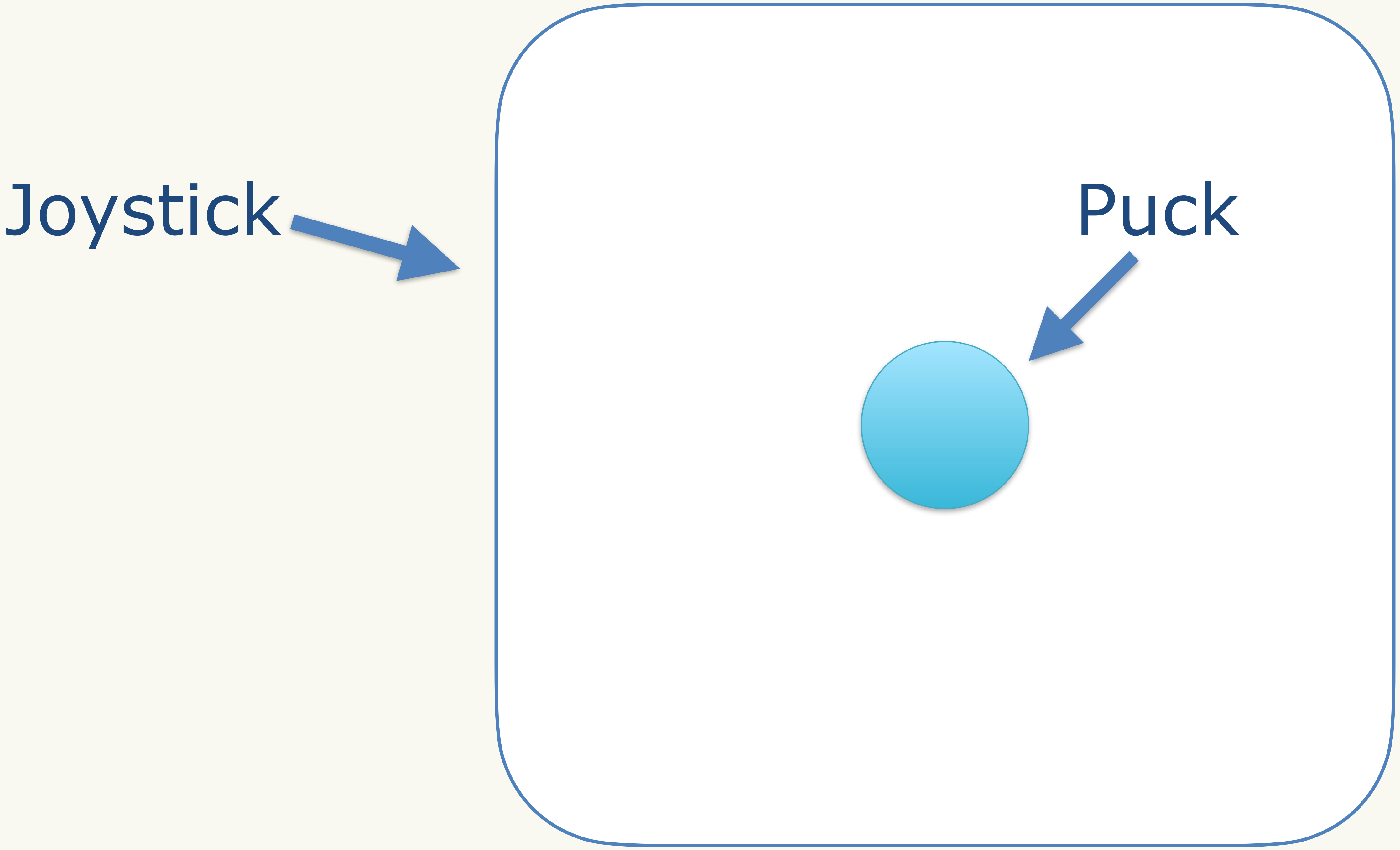
- I like mobile gaming and want to help make it better
- Pre-production - great for rapid prototyping
- This works on any engine
  - Mostly use Godot because it’s new to me

Disclaimer

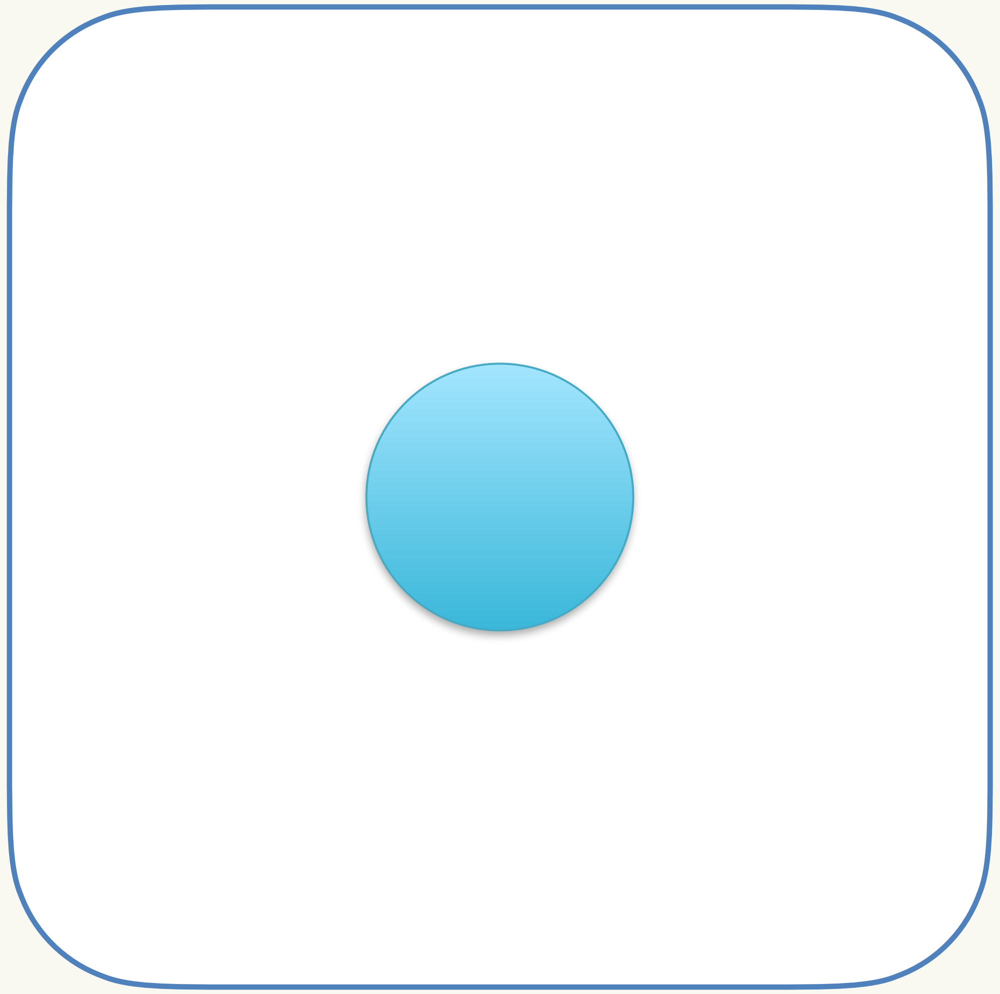
I’ve been asked to remind you that I’m here today on my time and not as a representative of Google. The time for cool Google related announcements was yesterday, today I got special permission to just nerd out about math.

Mobile Joysticks

Unofficial anatomy of a virtual joystick

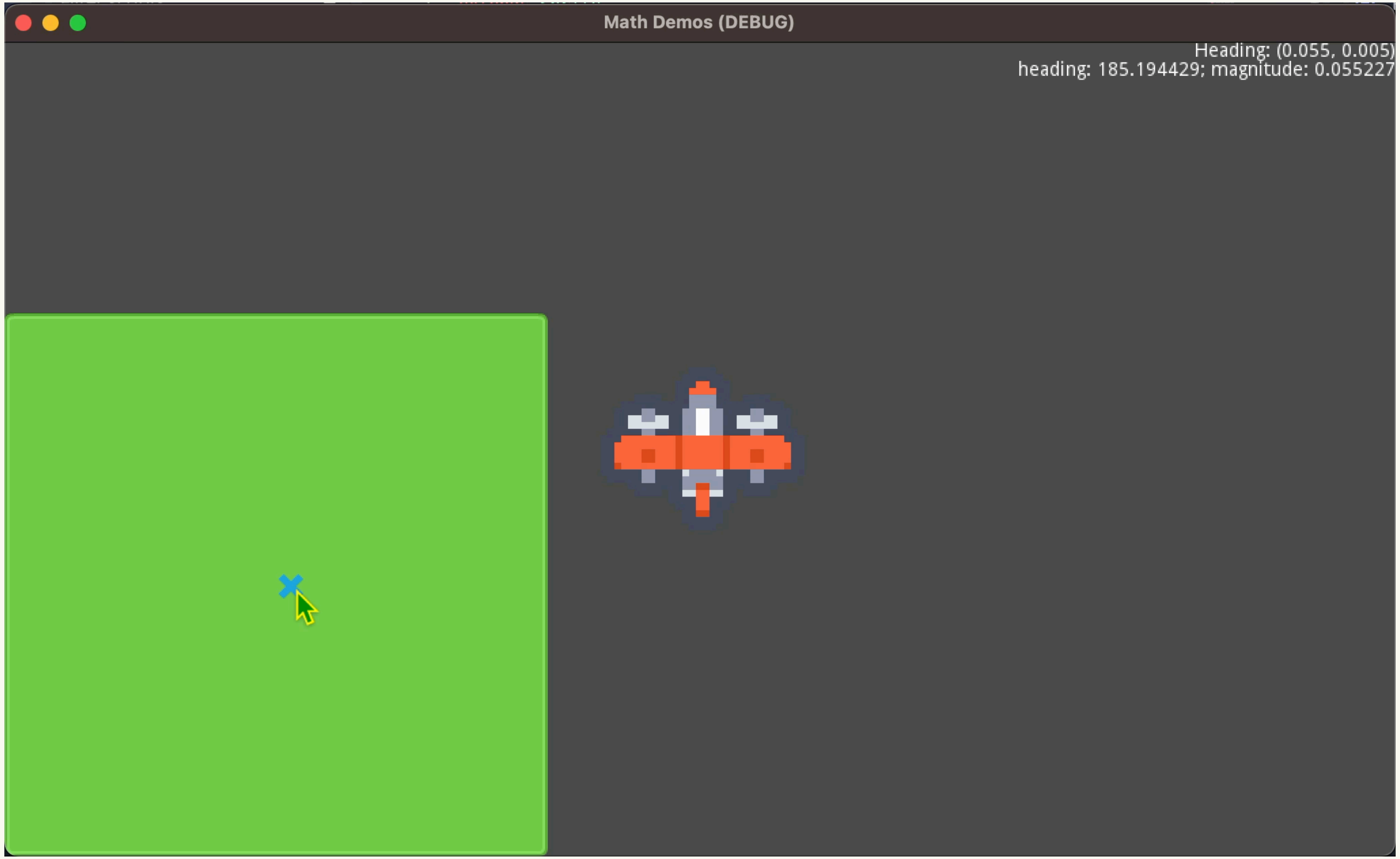


Typical Joystick



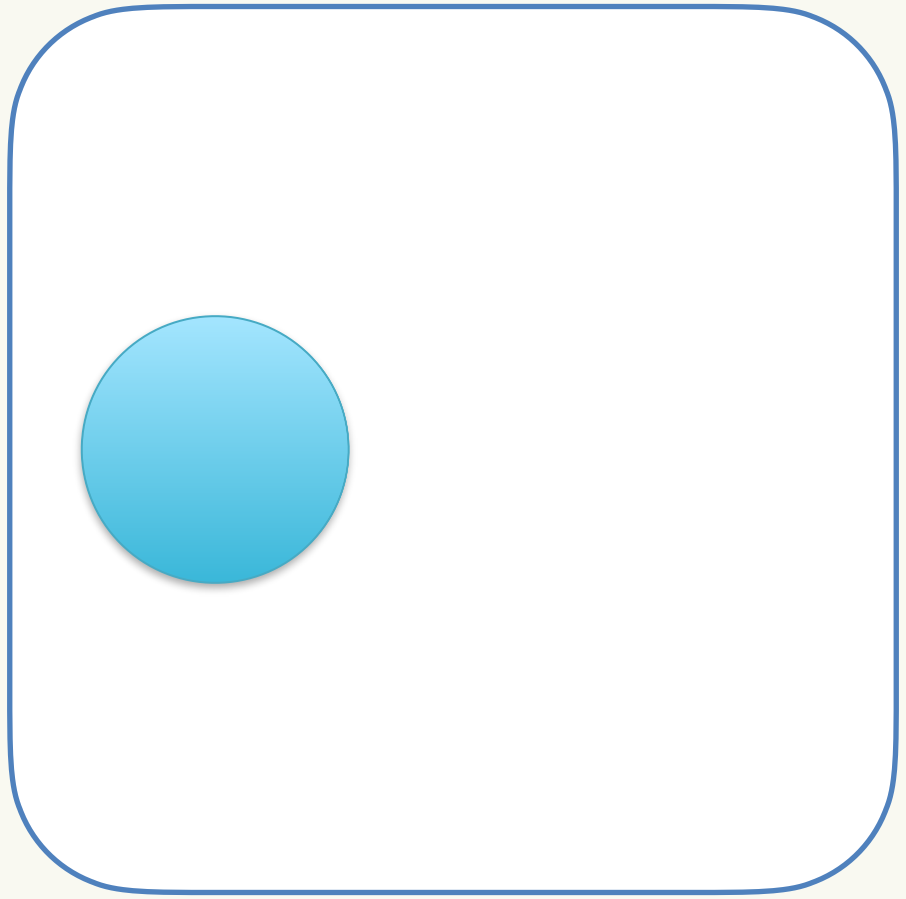
```
func _process_joystick(position: Vector2):
	var half_size = rect_size / 2
	var normalized = (position - half_size) / half_size
	if normalized.length_squared() > 1:
		normalized = normalized.normalized()
	emit_signal("vector_changed", normalized)
```

Example - Just Offset



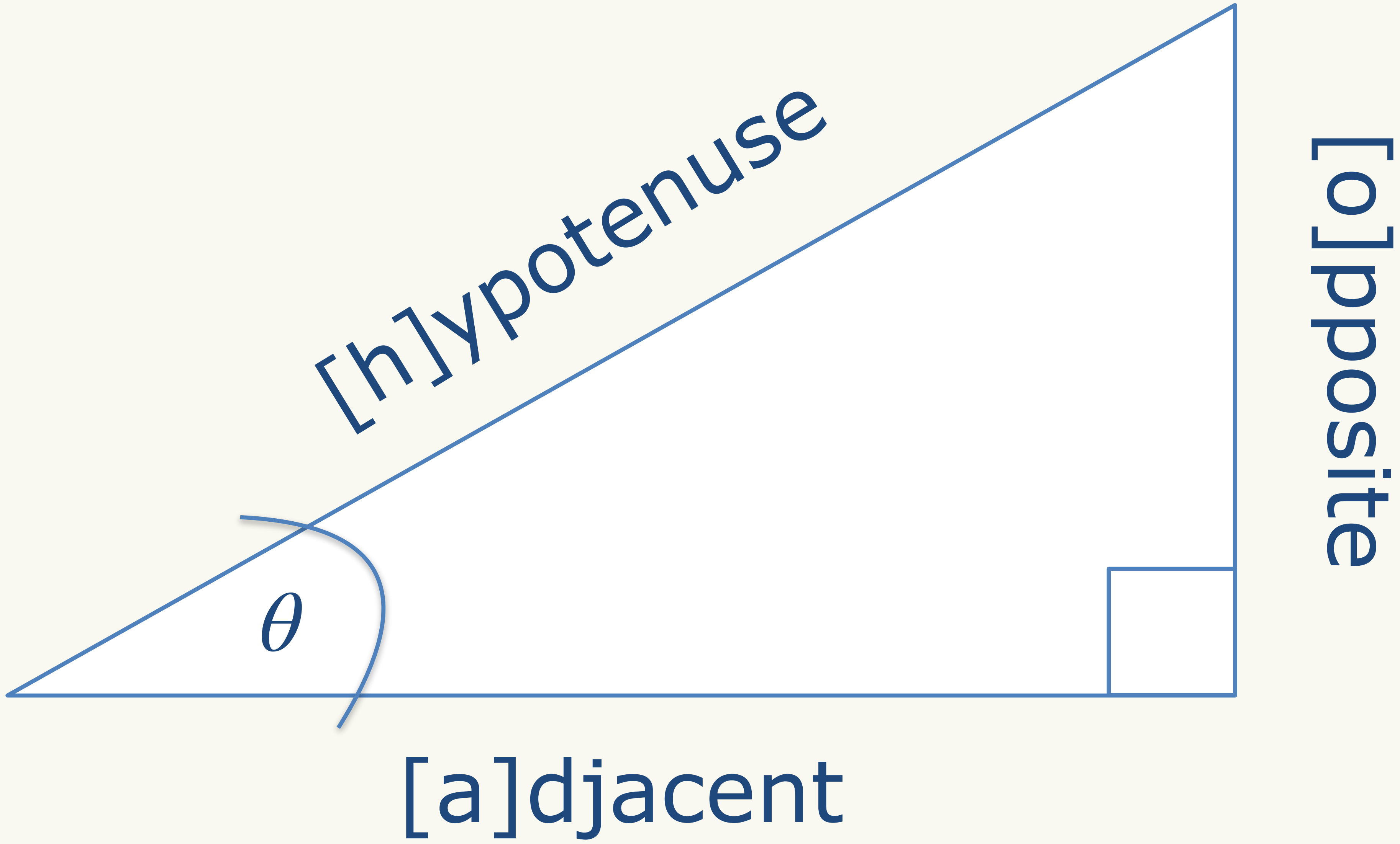
Sometimes you want heading

$$\begin{bmatrix} -1 \\ 0 \end{bmatrix}$$



$$(180°,1)$$

SohCahToa break

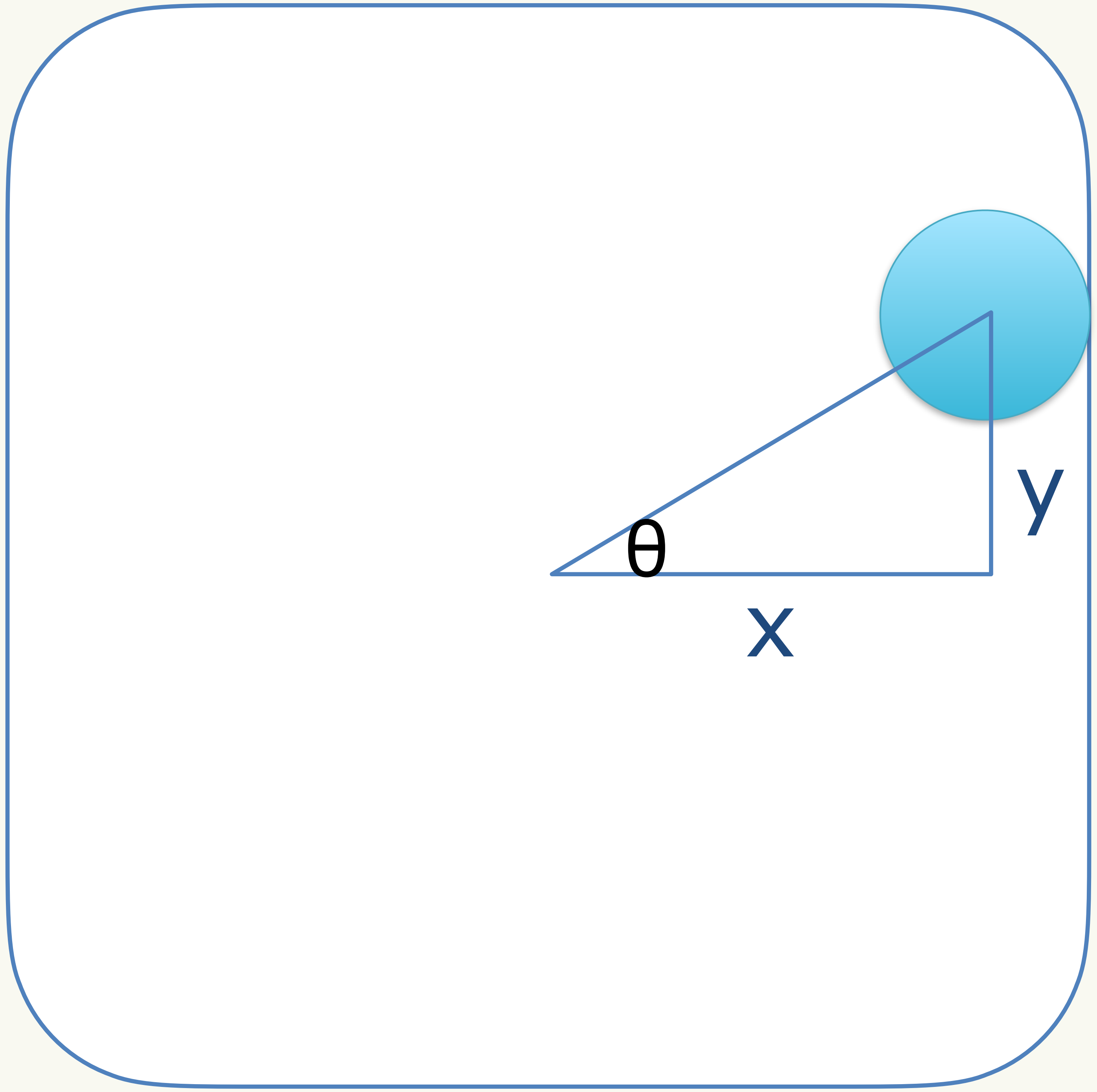


$\sin(\theta) = o / h$

$\cos(\theta) = a / h$

$\tan(\theta) = o / a$

Triangle to Joystick

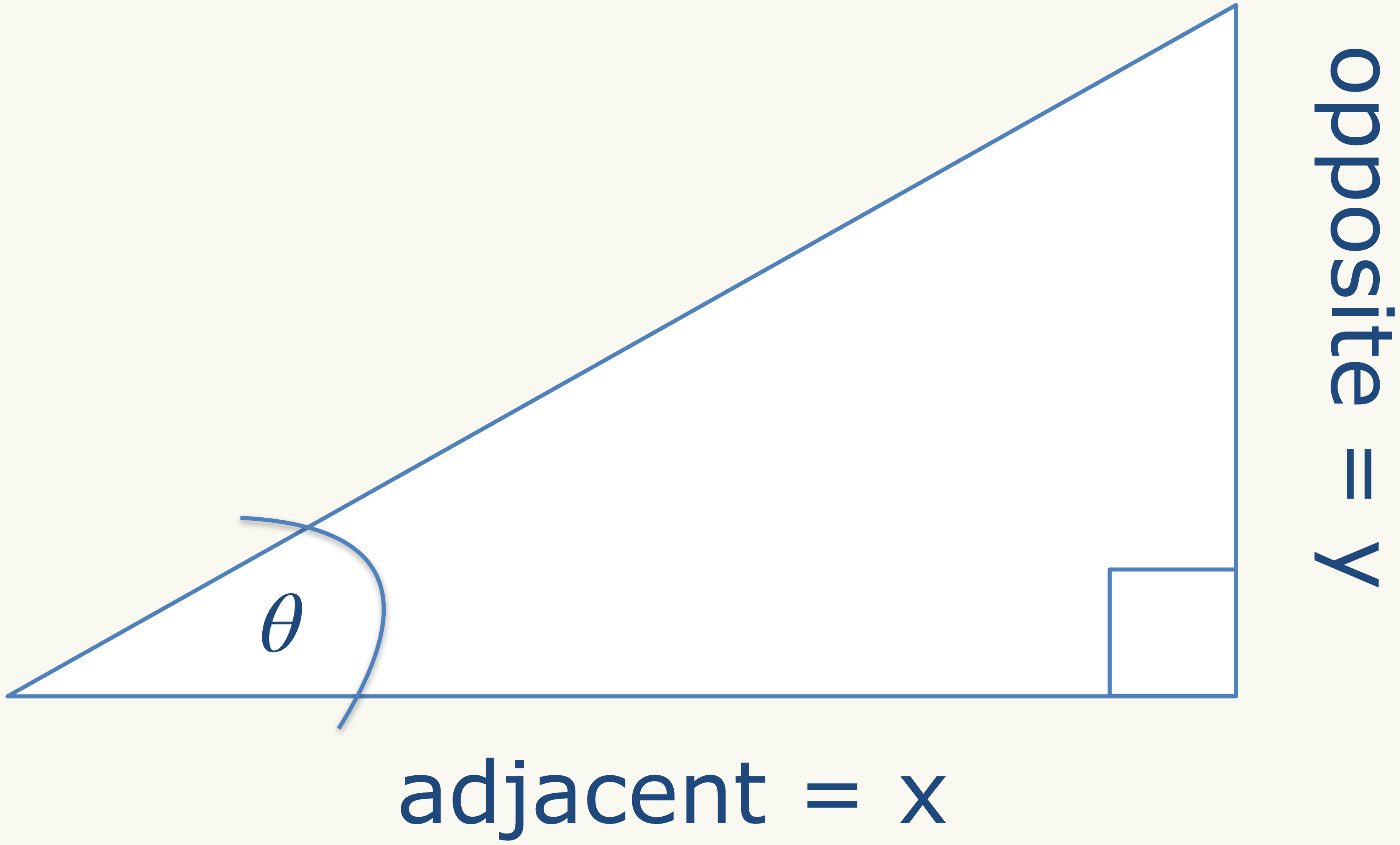


Opposite = y

Adjacent = x

$\tan(\theta) = y / x$

atan undoes tan



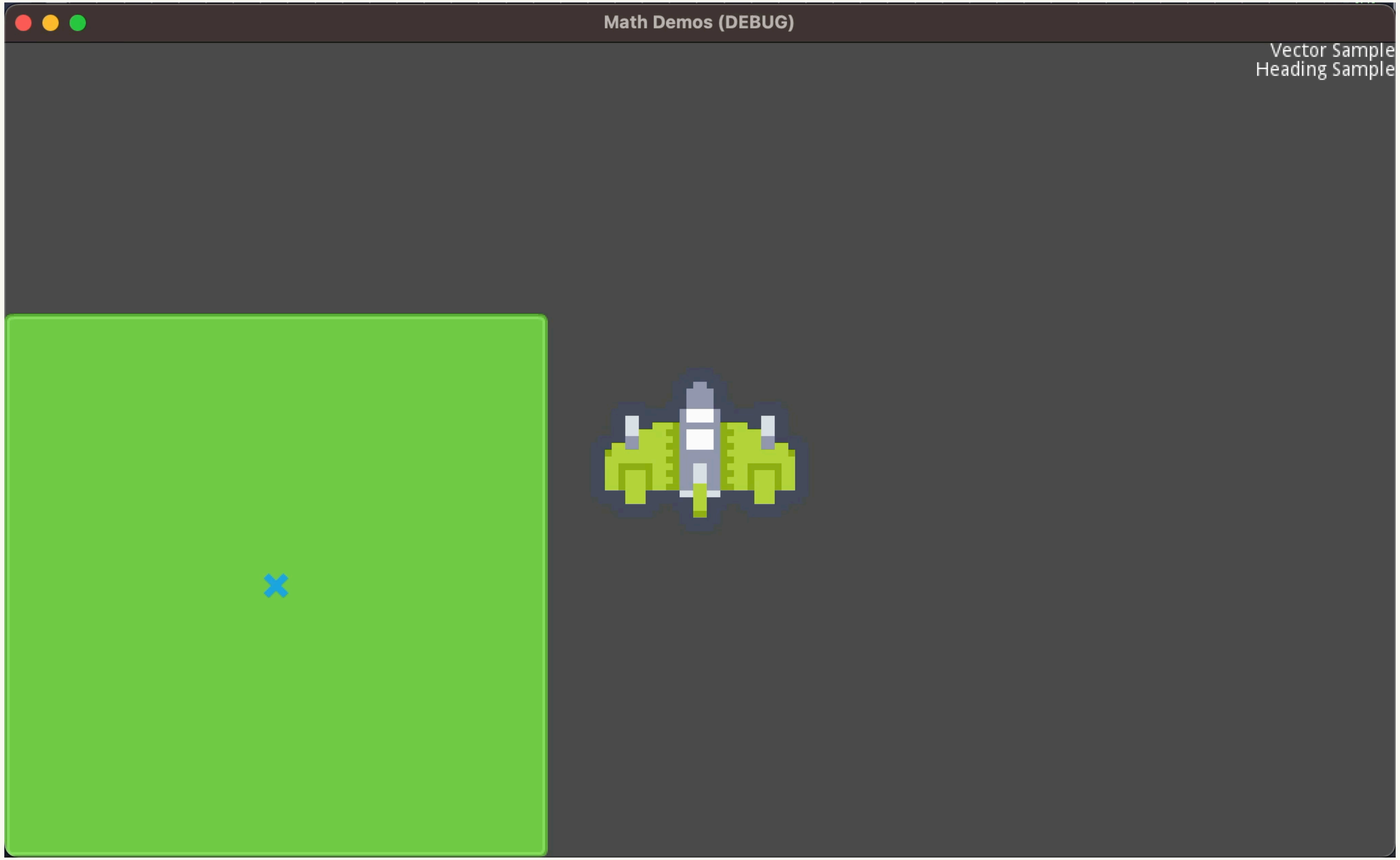
tan(θ) = y / x

θ = atan(y/x)

atan2(y, x) ≈ atan(y/x)

“arctangent with style!”

Example - Angle



"LERP"

LERP = Linear Interpolation

$f(t) = (1 - t)A + t(B)$

A and B are points (or anything that can multiply)

At $t = 0$ becomes $f(0) = (1 - 0)A + 0B = A$

At $t = 1$ becomes $f(1) = (1 - 1)A + 1B = B$

“LERP”

While resetting

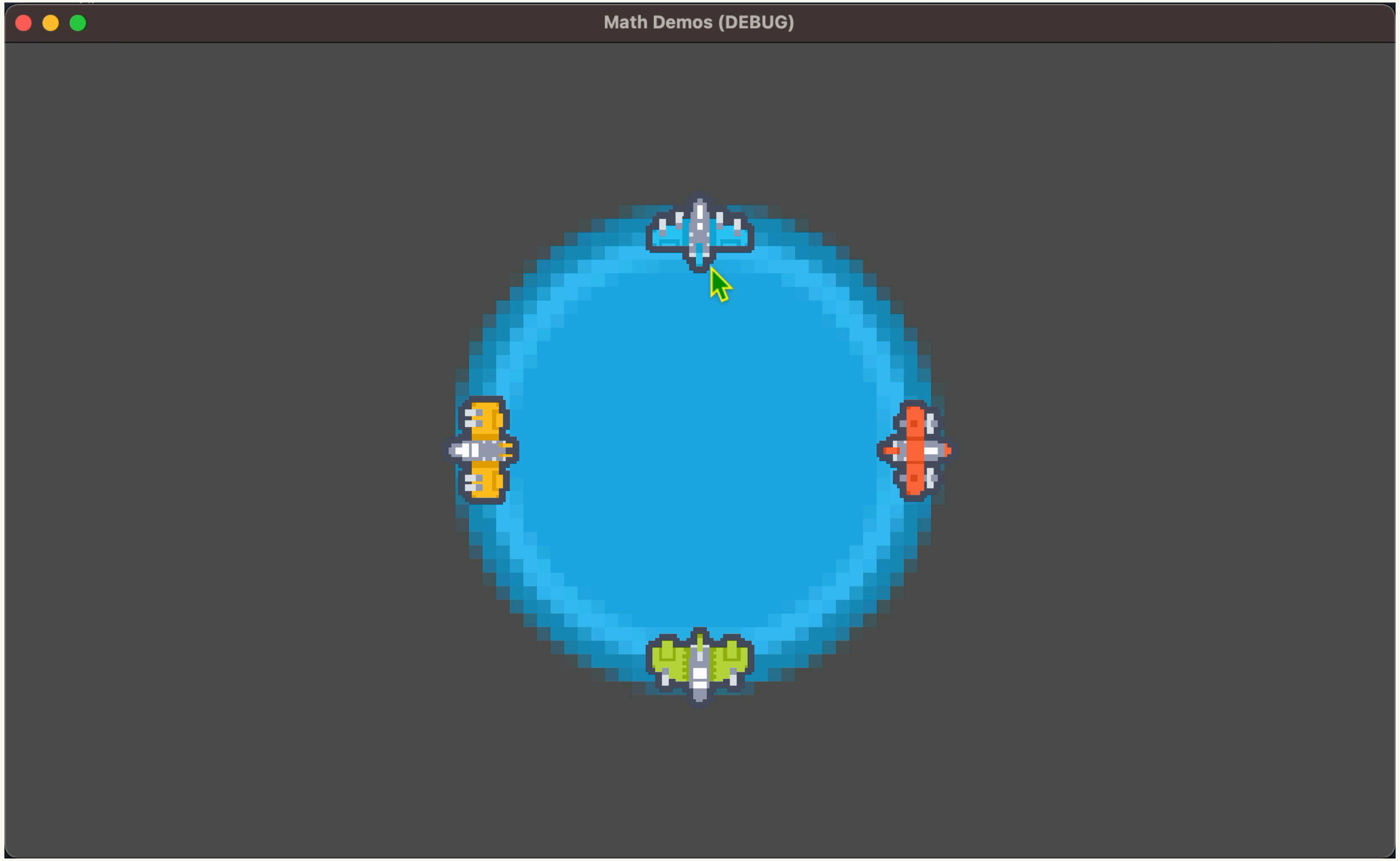
Count Down

Turn “seconds left” into 0 to 1

```
func _process(delta):
	if _current_reset_time > 0:
		_current_reset_time = _current_reset_time - delta
		if (_current_reset_time < 0):
			_current_reset_time = 0
		var rest_position = _joystick.rect_size / 2
		var t = 1 - _current_reset_time / reset_time
		t = t * t
		position = (1 -t) * _reset_start_position + t * rest_position
```

LERP

Wheels



Wheels

Basically a Joystick

```
var current_touch = event_to_world(event.position)
var current_rotation = atan2(current_touch.y, current_touch.x)
_rotational_offset = current_rotation - _touch_start_rotation
```

Wheels

Get the touch

```
var current_touch = event_to_world(event.position)
var current_rotation = atan2(current_touch.y, current_touch.x)
_rotational_offset = current_rotation - _touch_start_rotation
```

Wheels

Use ATan2 to find the angle

```
var current_touch = event_to_world(event.position)
var current_rotation = atan2(current_touch.y, current_touch.x)
_rotational_offset = current_rotation - _touch_start_rotation
```

Wheels

Rotating becomes simple addition/subtraction

```
var current_touch = event_to_world(event.position)
var current_rotation = atan2(current_touch.y, current_touch.x)
_rotational_offset = current_rotation - _touch_start_rotation
```

“How do I add angles?”

Add like normal to get θ

Then cap between 0 and 360 (similar math works for 0 to 2π):

$\theta = \theta - 180$: shift everything over 180

$\theta = \theta \,\%\, 180$: cap it between -180 and 180 (this is 360 degrees)

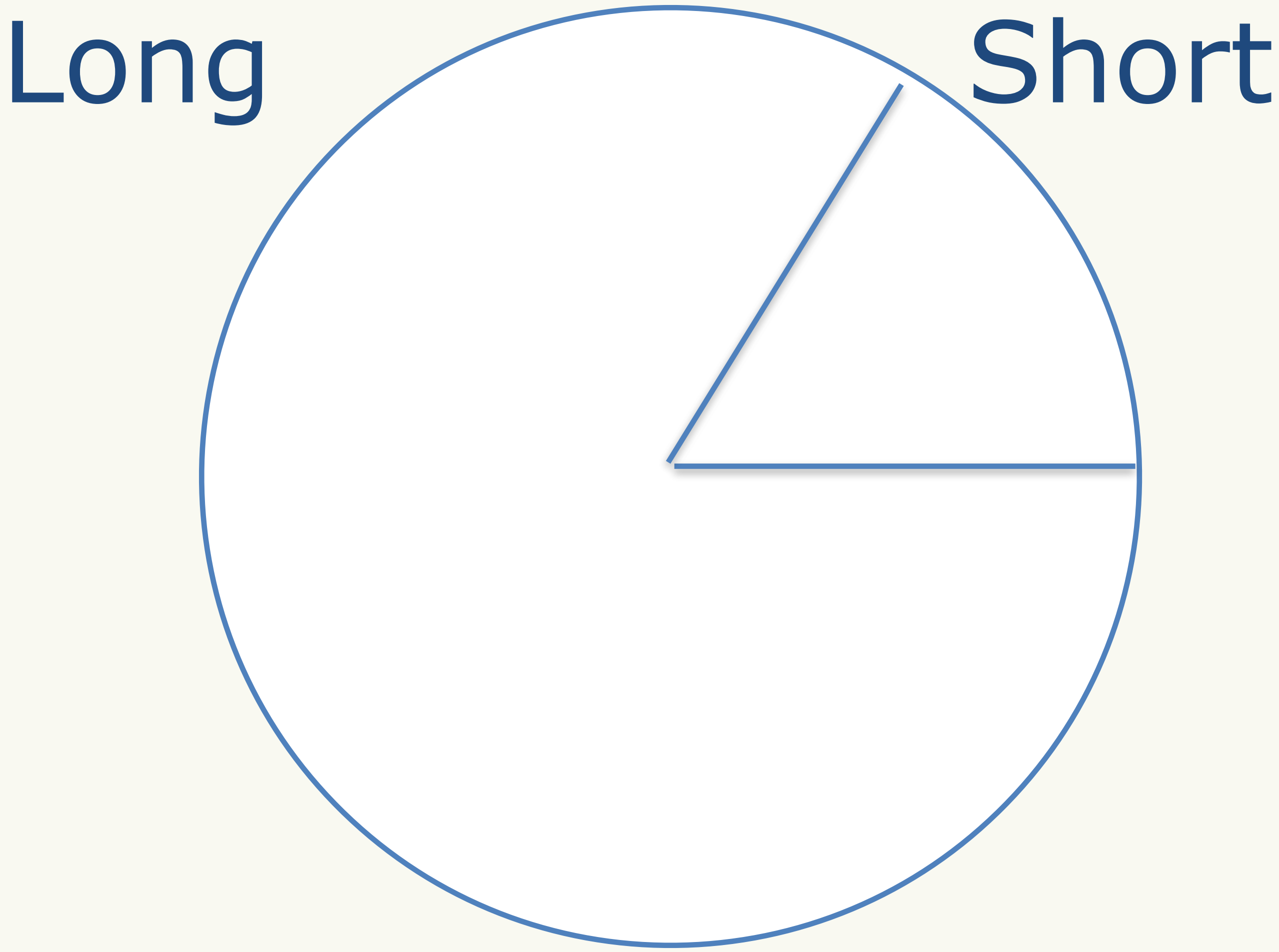
$\theta = \theta + 180$: put it back, now we have 0 to 360

This Becomes:

$$cap(\theta) = ((\theta - 180) \,\%\, 180) + 180$$

“How do I subtract angles?”

- Always two solutions
  - “Long” way and “Short” way
- Choose the smallest
  - If above 180 degrees, Subtract 360
  - If below -180 degrees, Add 360



“How do I subtract angles?”

```
var delta_angle = current_rotation - _last_rotation
if delta_angle > PI:
    delta_angle -= 2*PI
elif delta_angle < -PI:
    delta_angle += 2*PI
```

The 8-bit way

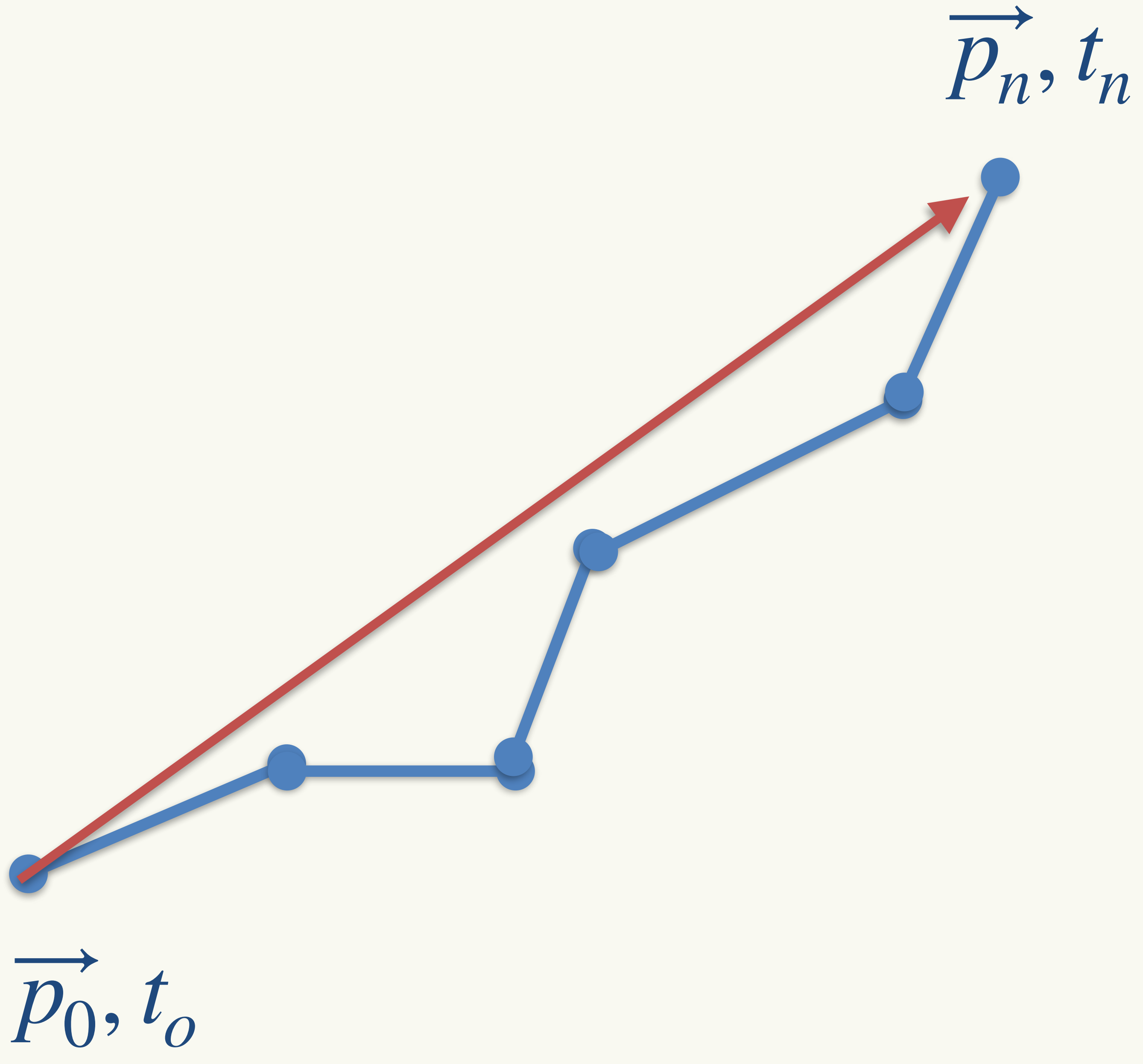
- “There are 255 degrees in an angle”
- When a uint_8 rolls over, it’s back to zero. No fancy math.
- You will need to scale it out for “real math” to move 255 to 360.

Flicking

Flicking

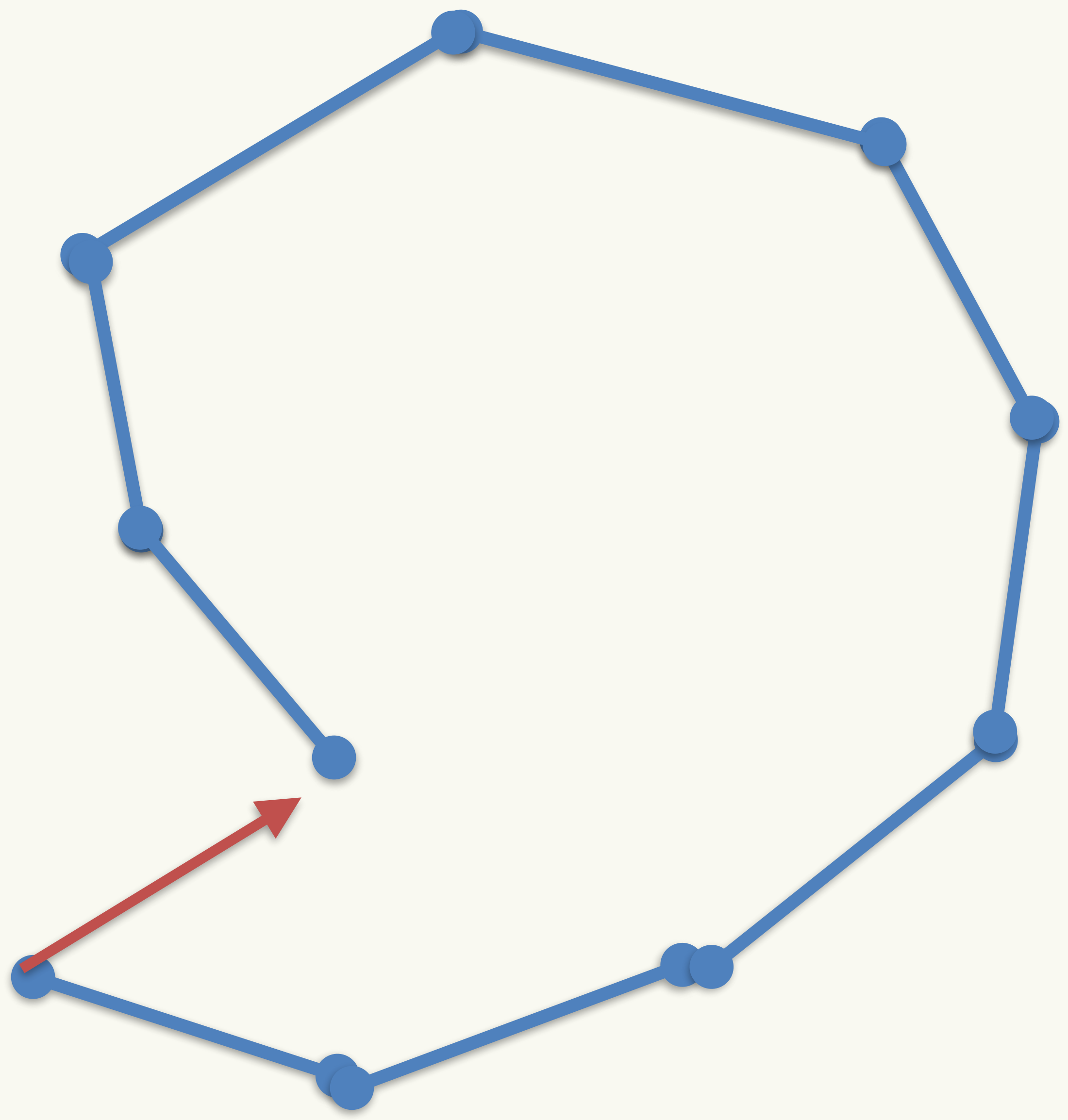
- Springs sound great, but not for this
- Will come back to this later

Flicking



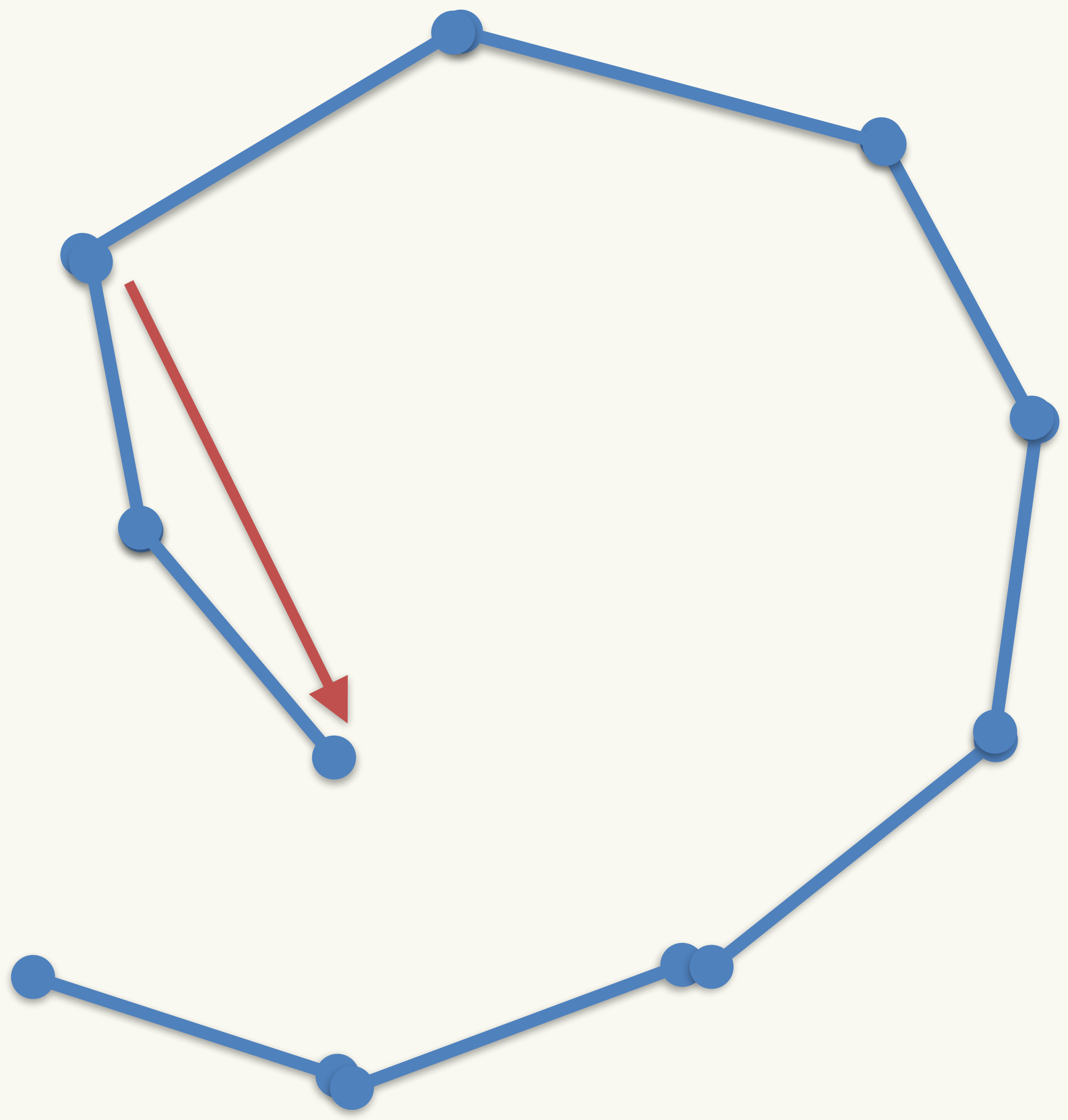
- Measure beginning and end
- $\overrightarrow{v'} = \dfrac{\overrightarrow{p_1} - \overrightarrow{p_0}}{t_1 - t_0}$
- Works for fast straight flicks but…

Flicking



- Spiral?
- Breaks immediately
- Just take the last sample?
- Still want to smooth jitters

Flicking

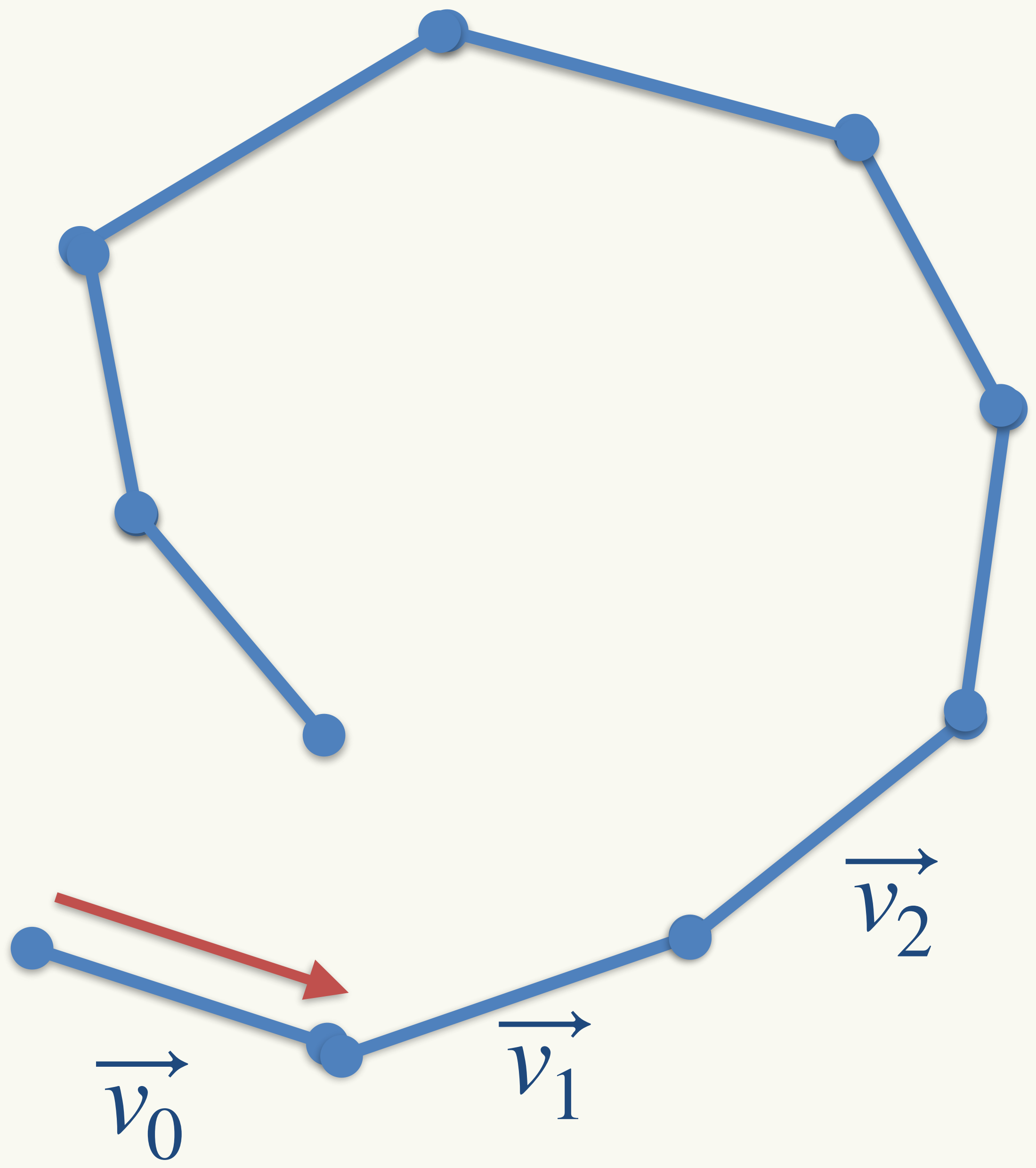


- Keep a rolling list!
- With n samples, $n = 3$ gives us a good result!
- What happens if samples don't come in at an even rate or your frame rate changes?

Flicking

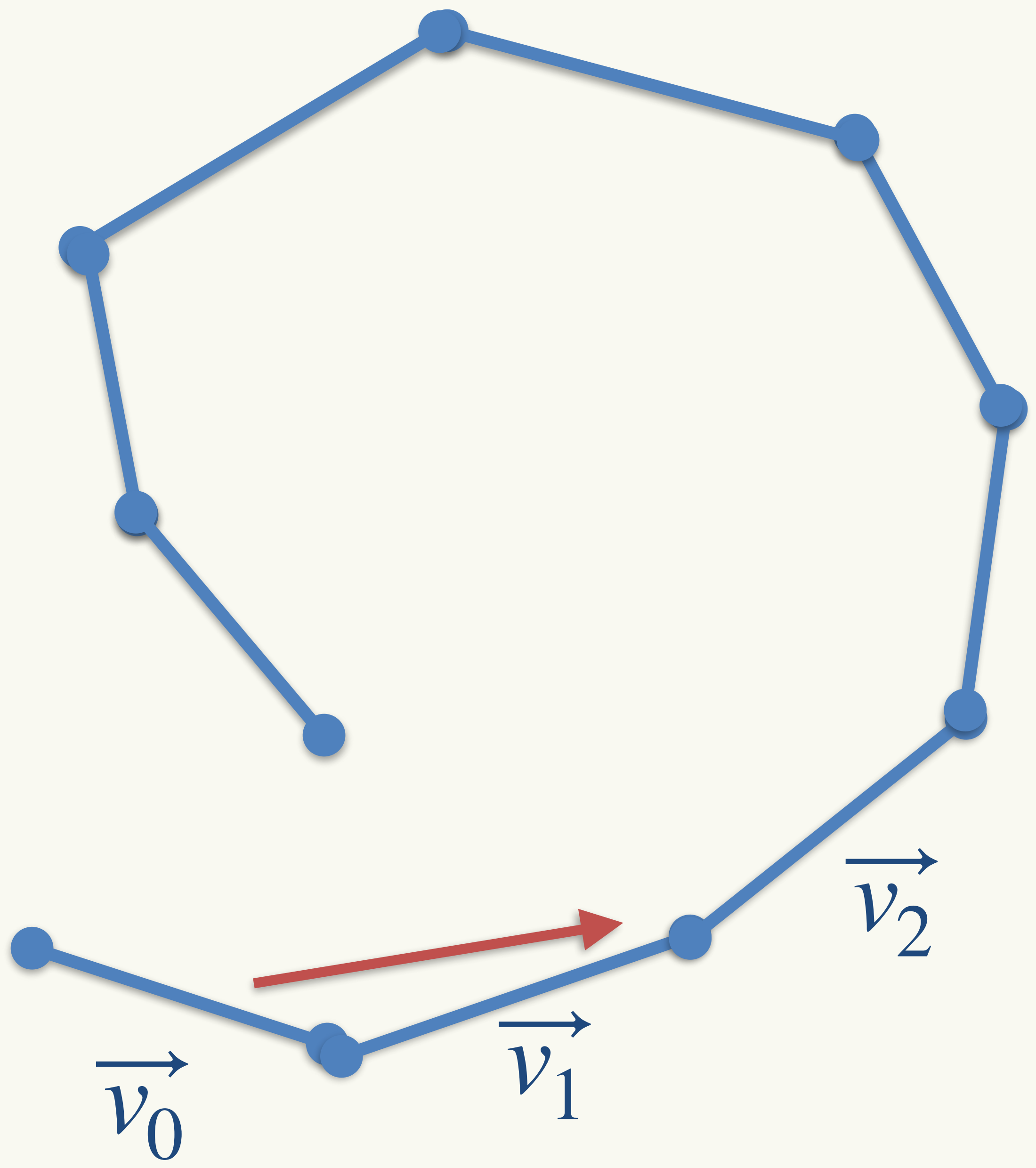
- Remember LERP?
- $f(t) = (1 - t)A + tB$
- What happens if B is the new sample, and A is the old one?
- t is fixed at .75

Flicking



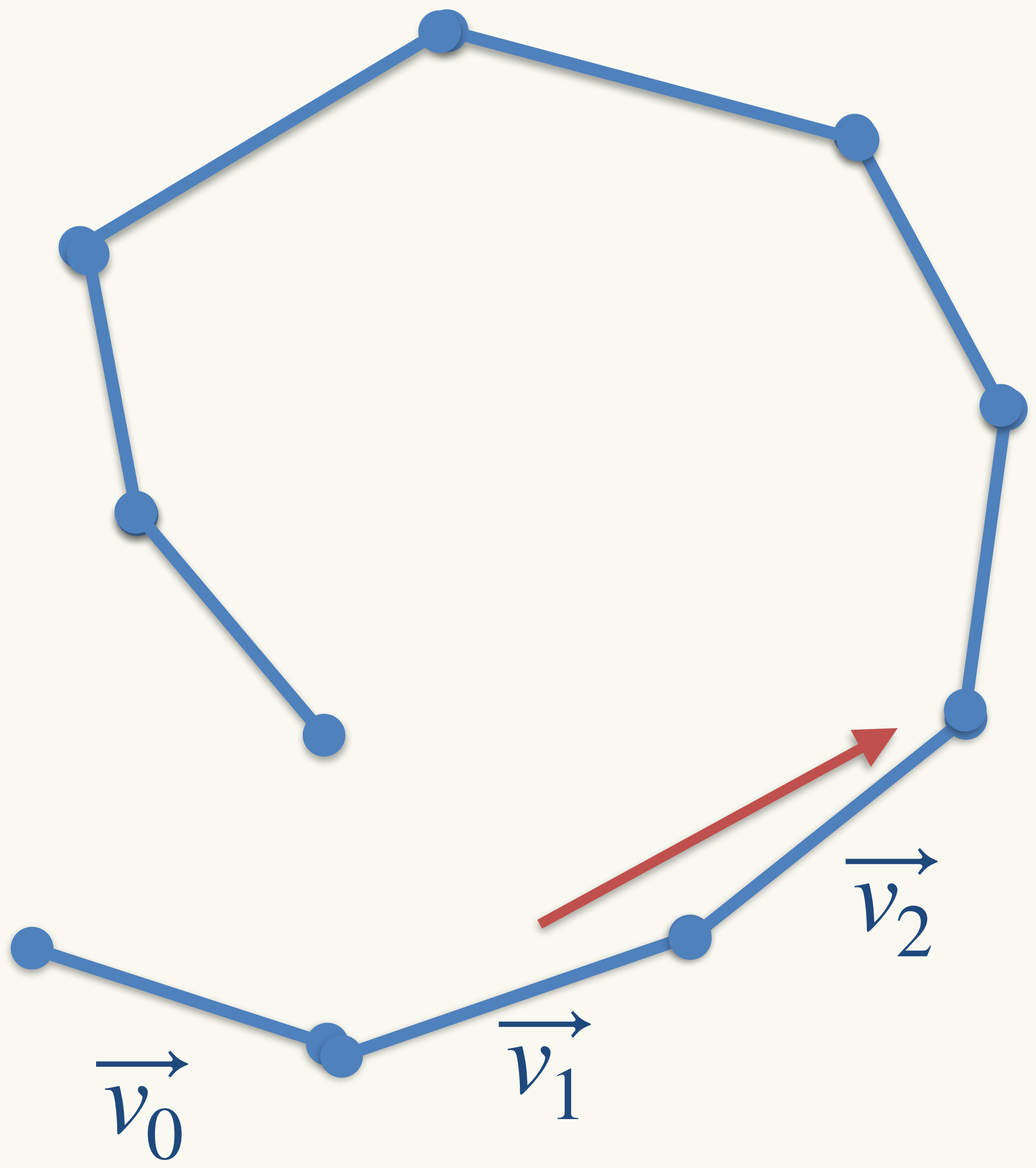
- Start with the flick speed being the first two points over the time between them.

Flicking



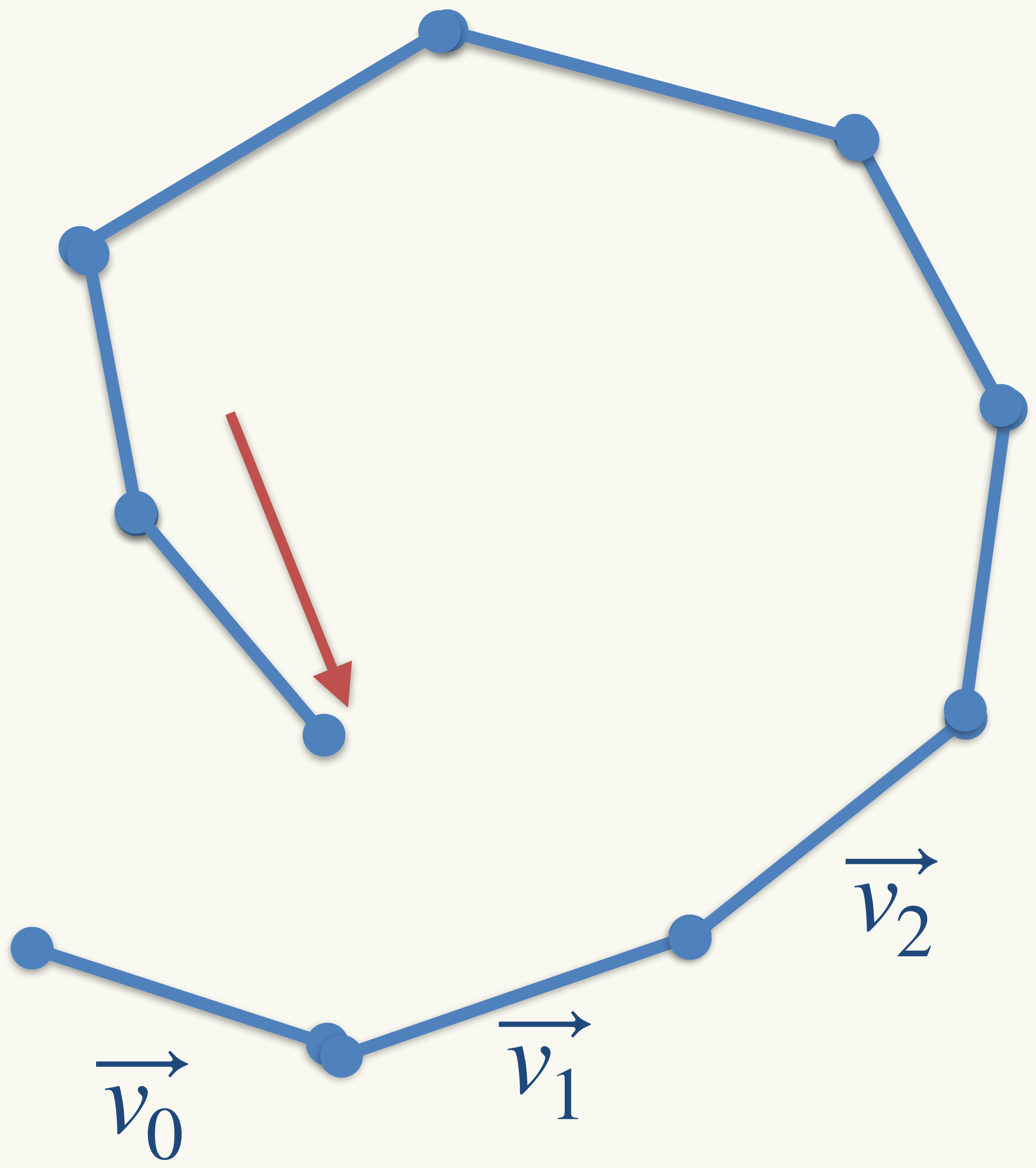
- The next vector mostly takes over (75%) but the old one still factors in (25%)

Flicking



- The next sample has the most weight (75%)
- The rest are in the last 25%

Flicking



- The end of the spiral is still in the right direction, but previous samples are still factored

Flick - Code

Duration of change

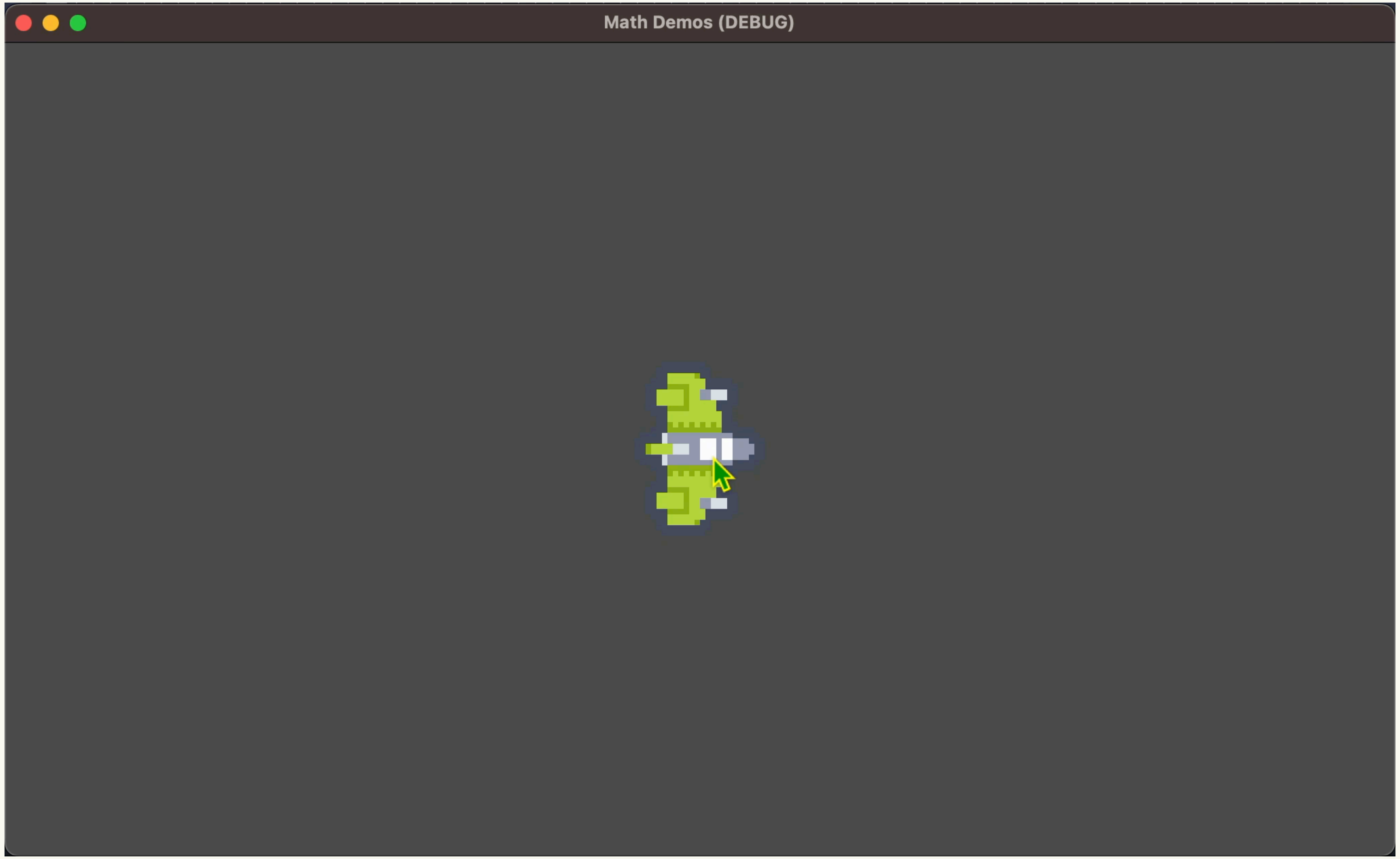
Change in touch position

```
var delta = world_click - _start_position
var time = OS.get_ticks_msec()
var time_delta = time - _last_time
if time_delta != 0:
	_last_time = time
	var current_flick = (world_click - _last_position) / _msec_to_sec(time_delta)
	_flick_vector = (1 - slew) * _flick_vector + slew * current_flick
	_last_position = world_click
	_sync_position = _start_local_position + delta
```

Velocity of sample

The magic LERP

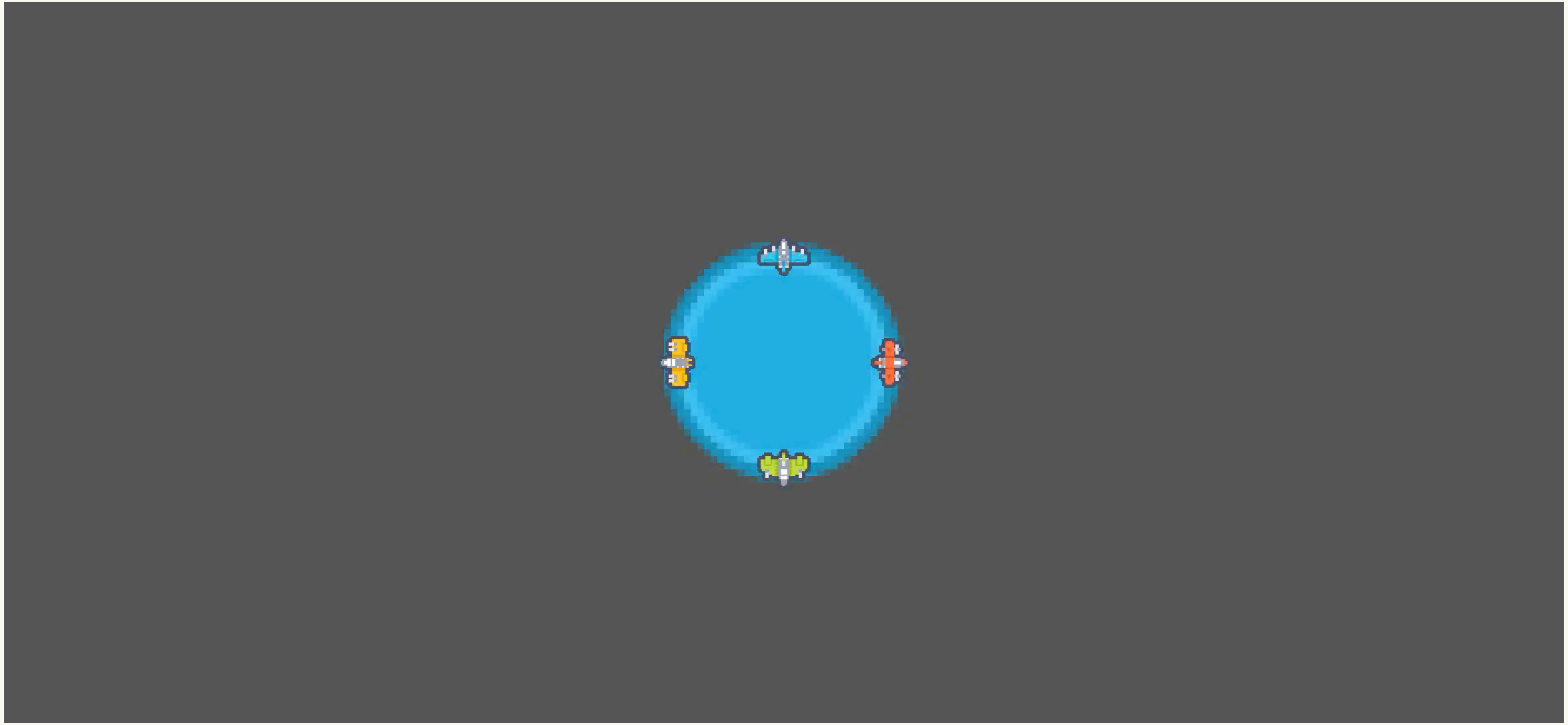
Flicking Demo



Spinning Wheel

- Works with wheels too!
- Use angle deltas over time rather than points
- End with rotational velocity
- Requires subtracting angles

Spinning Wheel



Flicking

- What about 3d?
- Project it to 2D
  - But we need picking…

3D Picking

Picking (the easy way)

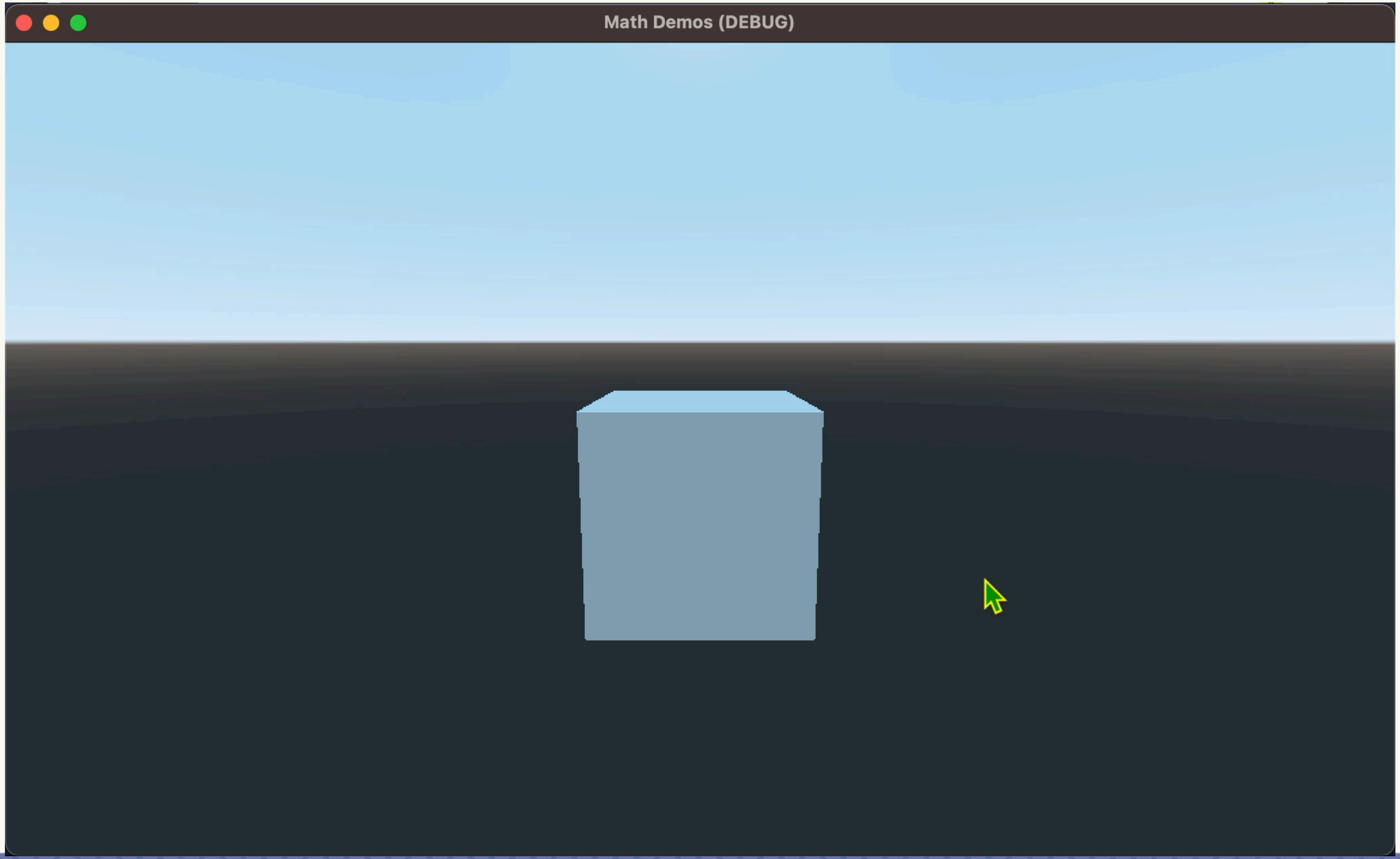
Screen point to ray

Pick an object

```
var origin = project_ray_origin(event.position)
var direction = project_ray_normal(event.position)

var space_state = get_world().direct_space_state
var result = space_state.intersect_ray(origin, origin + direction * 100)
if result:
	print("Hit at point: ", result.position, " with object: ", result.collider)
```

Picking Demo



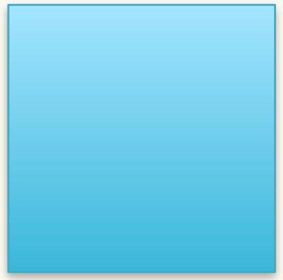
Projection (Math Break!)

What if your engine doesn't do picking?

- The math is well known, but complicated
- Let's talk about how we see a triangle in game

Projection

Cool cube!

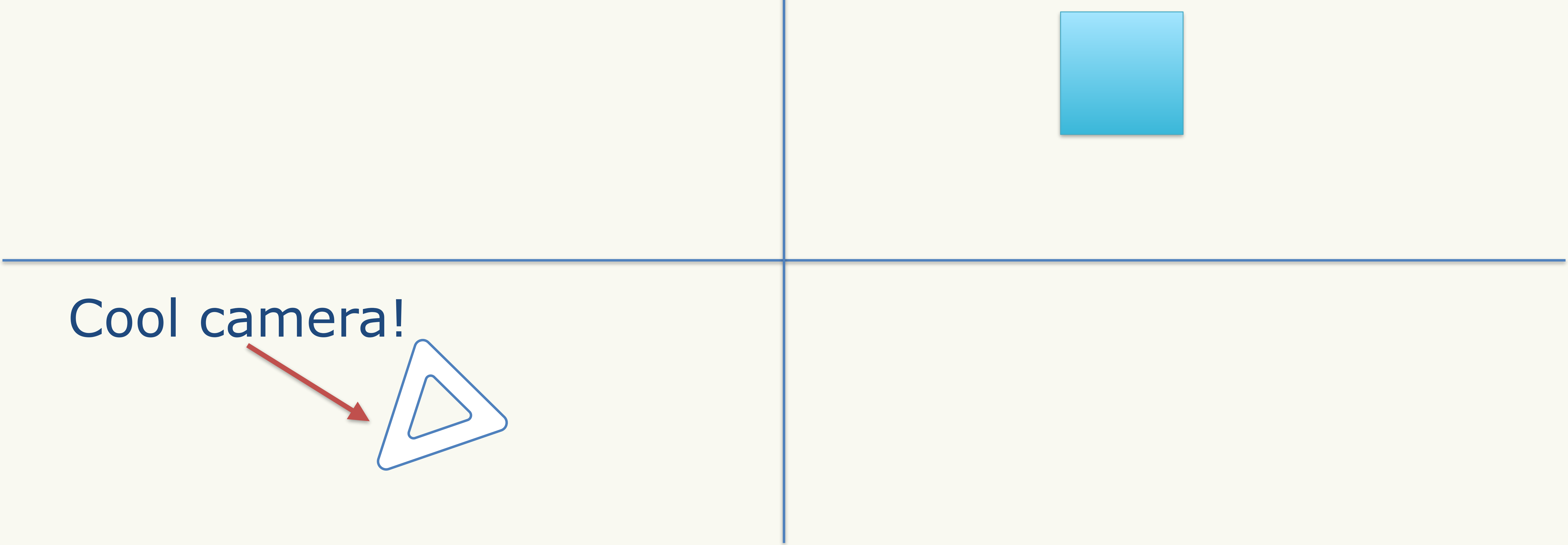


Model transform

Whatever we're transforming
(A vertex)

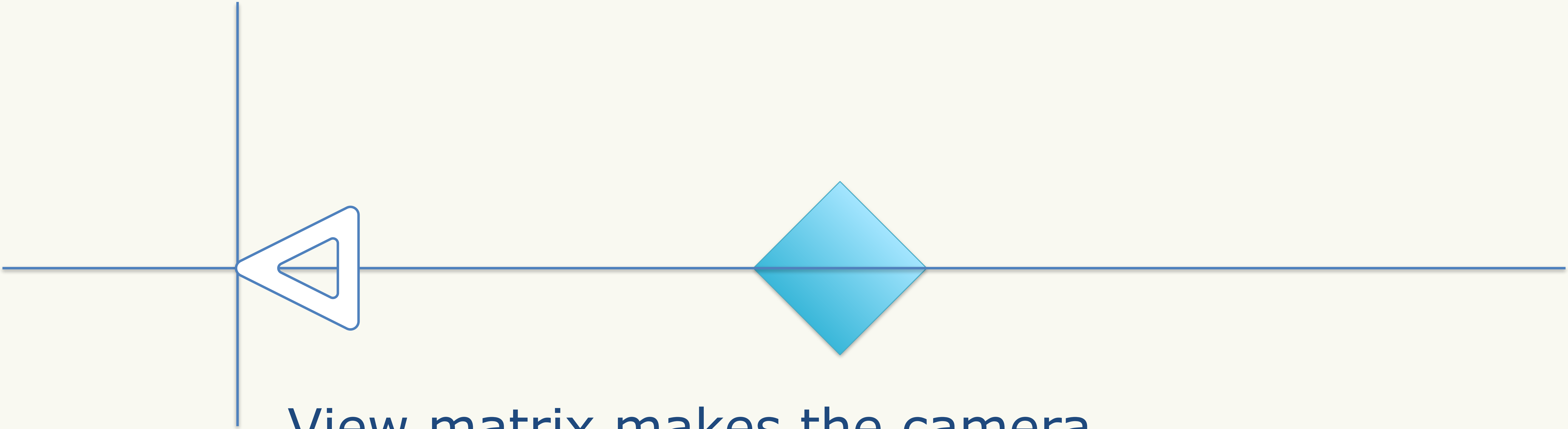
$M\vec{v}$

Projection



$M\vec{v}$

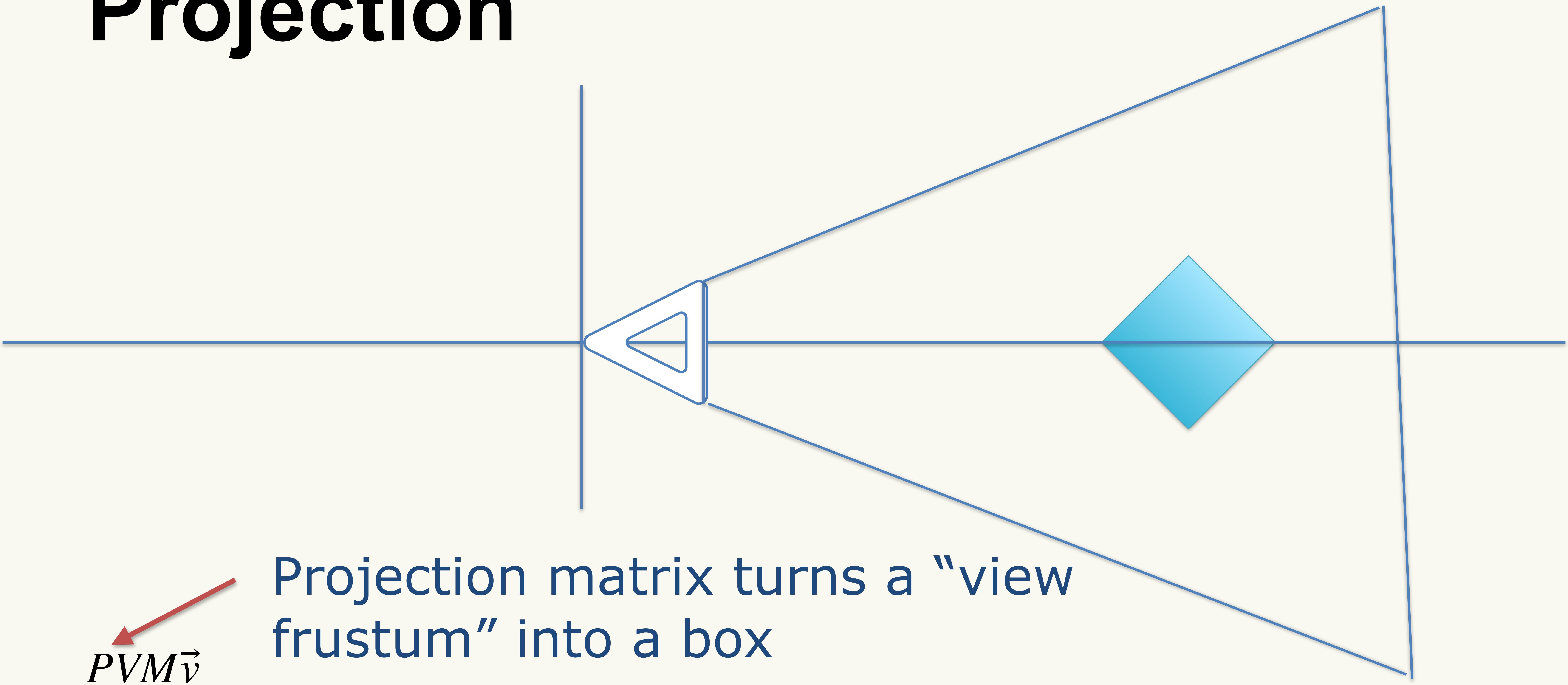
Projection



View matrix makes the camera the center of the world

$VM\vec{v}$

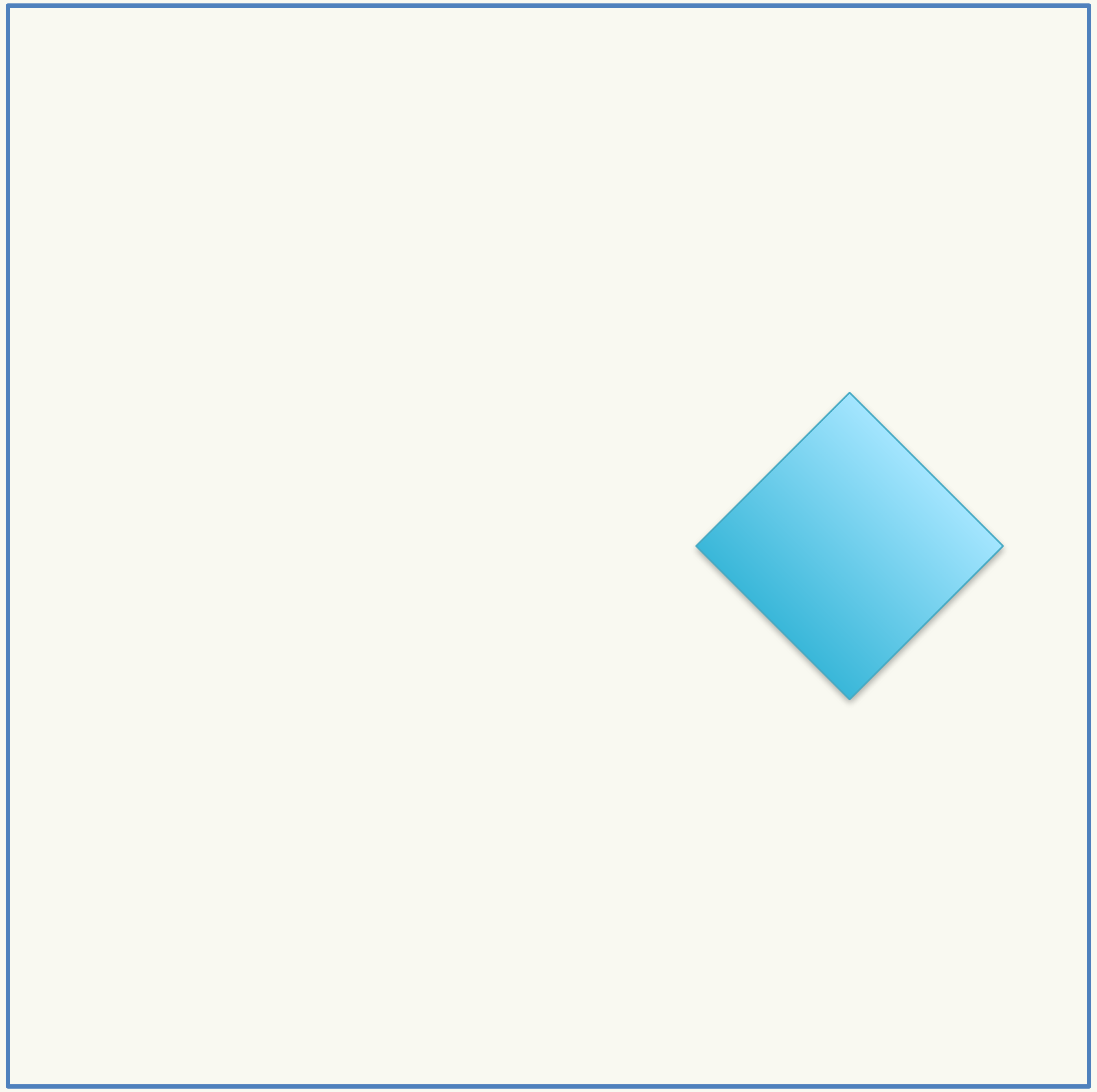
Projection



Projection matrix turns a “view frustum” into a box

$PVM\vec{v}$

Projection



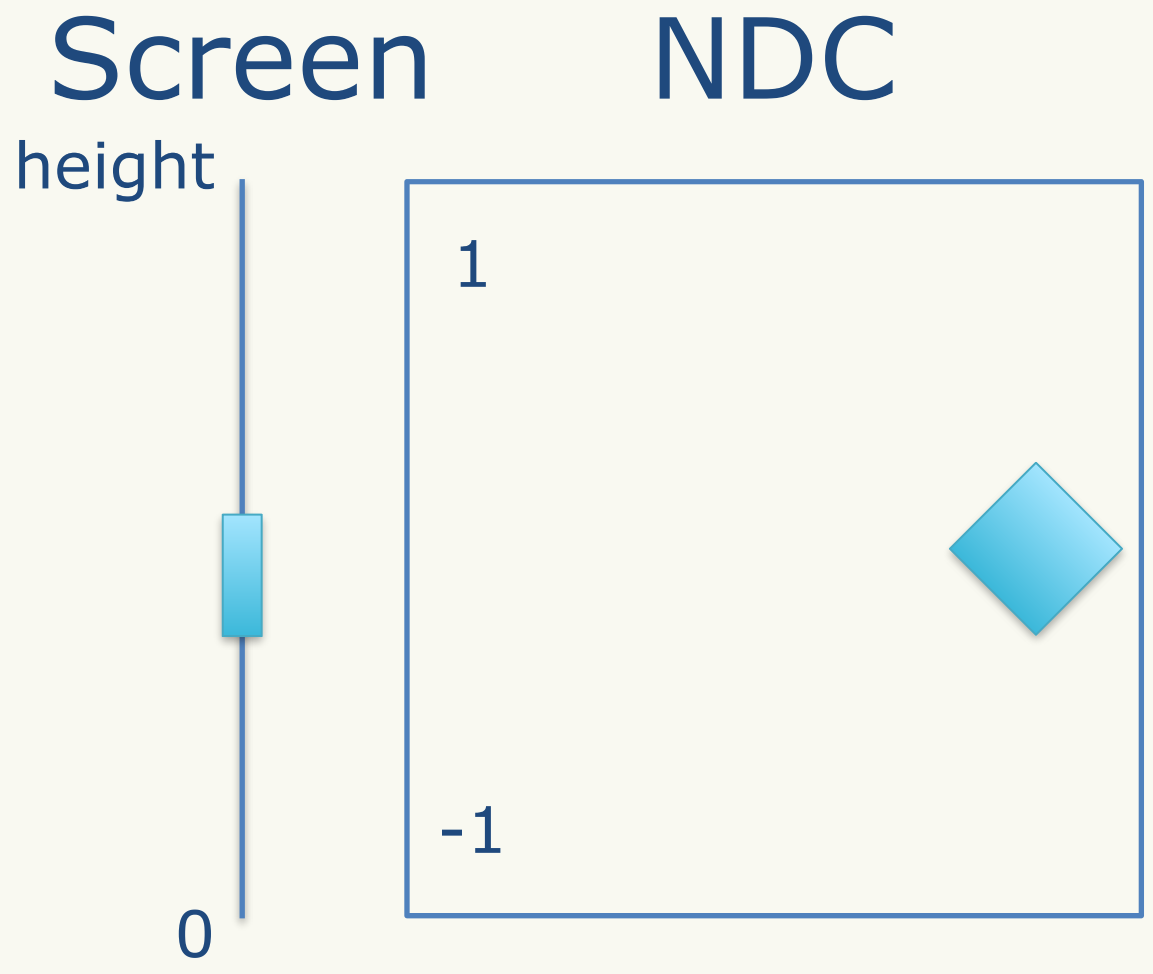
$PVM\vec{v}$

Projection

- Something weird called "homogeneous division"
- Your vertex right now is 4D: $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$
- Divide by w to get to "normalized device coordinates" $\begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{bmatrix}$
- There's a bunch here I'm skipping, the math works

Projection

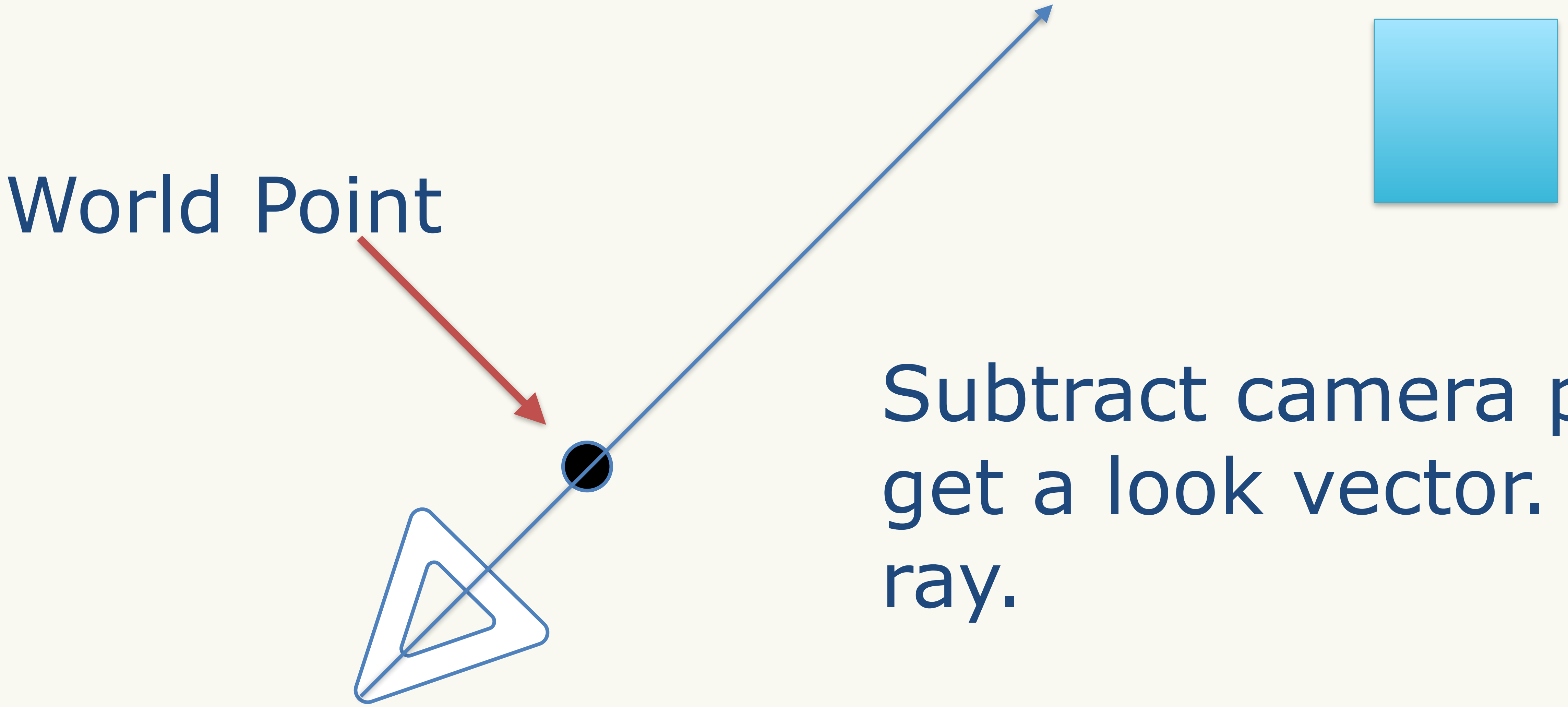
- Values are now -1 to 1 on any axis
- Stretch from -1, 1 to 0, screen size
- Squish on z to get a screen point



Unproject

- $\begin{bmatrix} x \\ y \end{bmatrix}$ screen point to NDC (change to -1 to 1)
- Remember $PVM\vec{v}$? Take PV and invert it
- You will have a 4D vector: $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$
- Make it 3D by dividing out w: $\begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{bmatrix}$

Unproject



Subtract camera position to get a look vector. Makes a ray.

Unproject

Raycast like before!

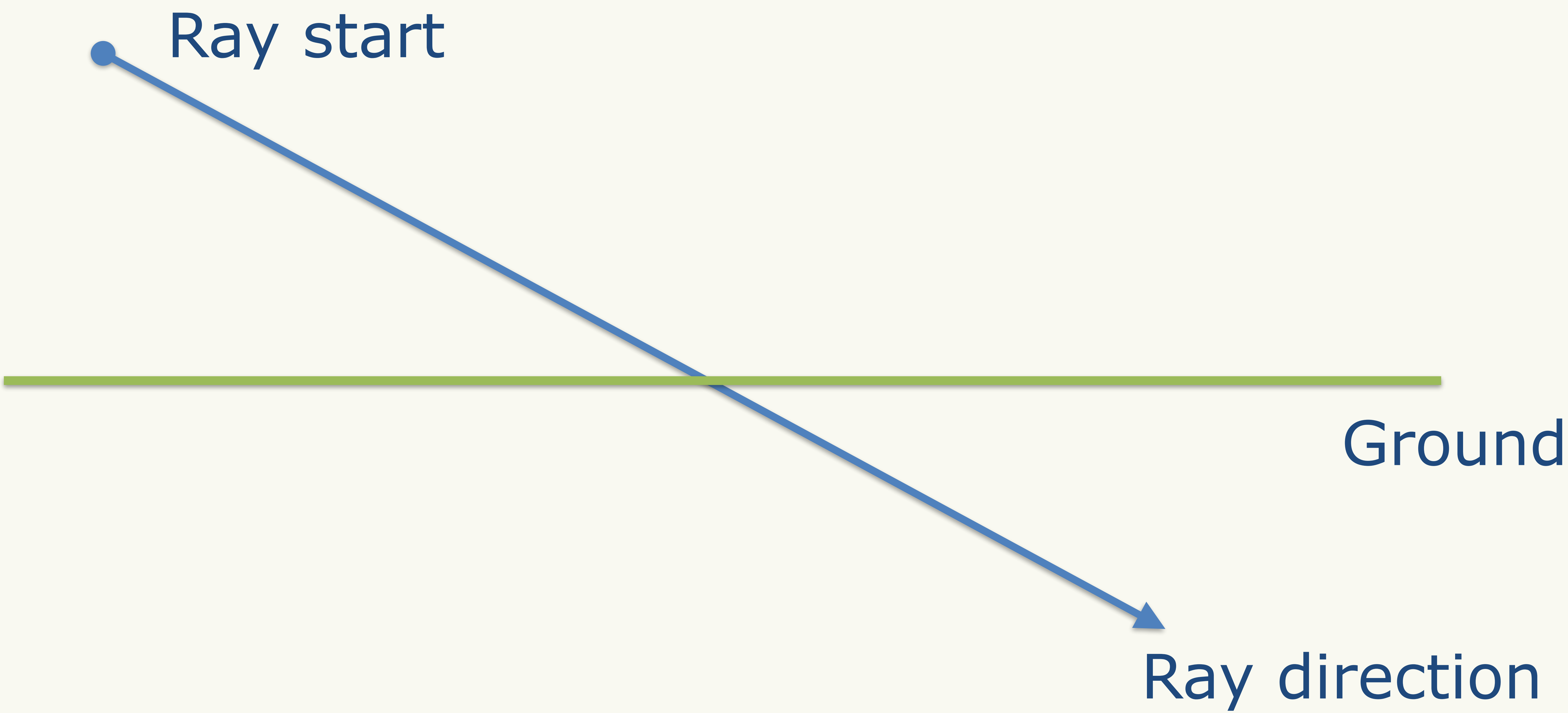
```
var origin = project_ray_origin(event.position)
var direction = project_ray_normal(event.position)

var space_state = get_world().direct_space_state
var result = space_state.intersect_ray(origin, origin + direction * 100)
if result:
    print("Hit at point: ", result.position, " with object: ", result.collider)
```

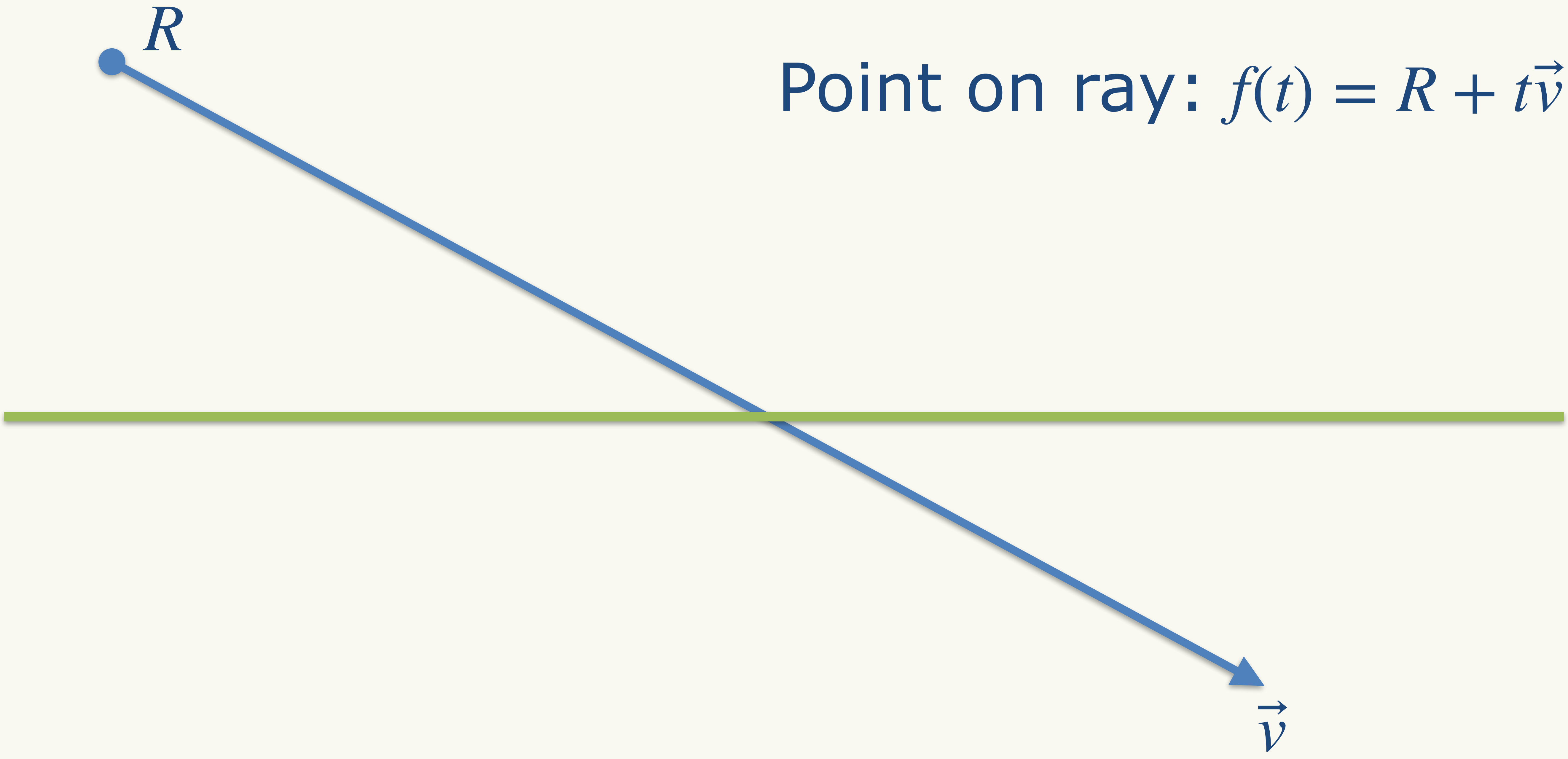
Flicking (3D)

This is why we needed to learn about picking…

Flicking (3D)

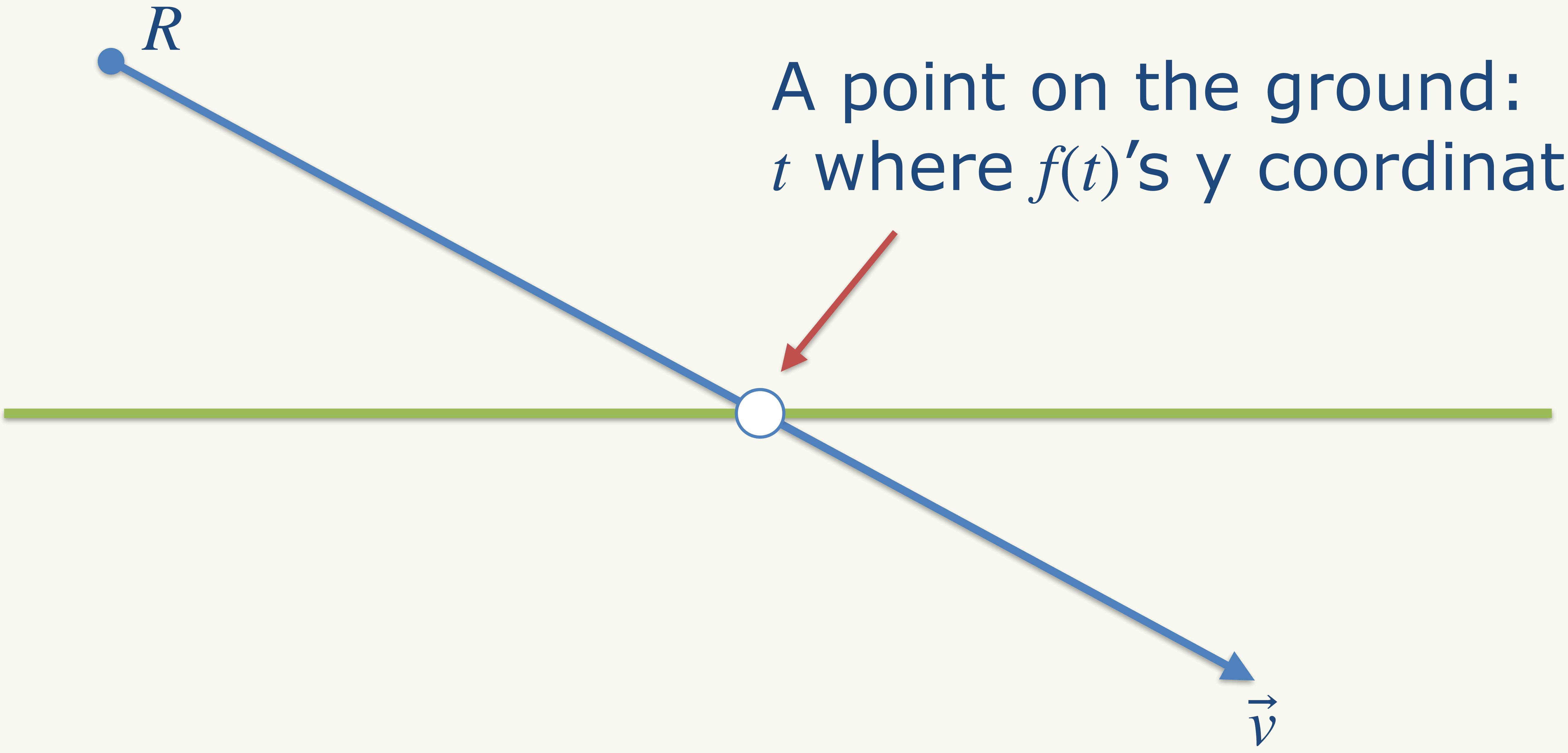


Flicking (3D)



Point on ray: $f(t) = R + t\vec{v}$

Flicking (3D)



A point on the ground:
t where $f(t)$'s y coordinate is 0

Flicking (3D)

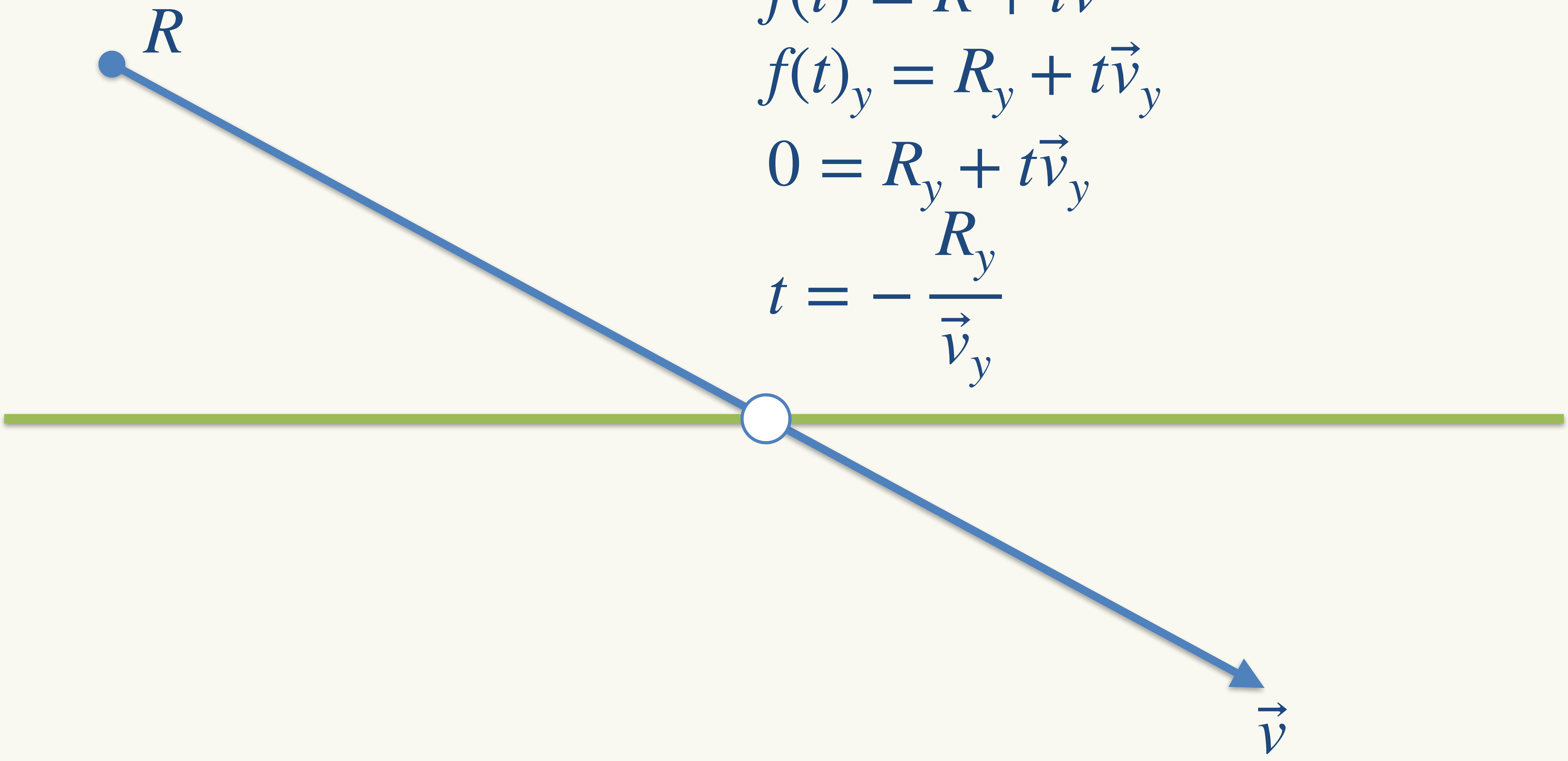
A point on the ground:

$$f(t) = R + t\vec{v}$$

$$f(t)_y = R_y + t\vec{v}_y$$

$$0 = R_y + t\vec{v}_y$$

$$t = -\frac{R_y}{\vec{v}_y}$$



Flicking 3D

Ray from touch

```
func _unhandled_input(event):
    if event is InputEventMouseButton:
        # get the ray from the touch
        var touch_origin = project_ray_origin(event.position)
        var touch_ray = project_ray_normal(event.position)

        # find where the ray touches the ground
        var t = - touch_origin.y / touch_ray.y
        var point = touch_origin + t * touch_ray

        # put an object there
        _marker_obj.translation = point
```

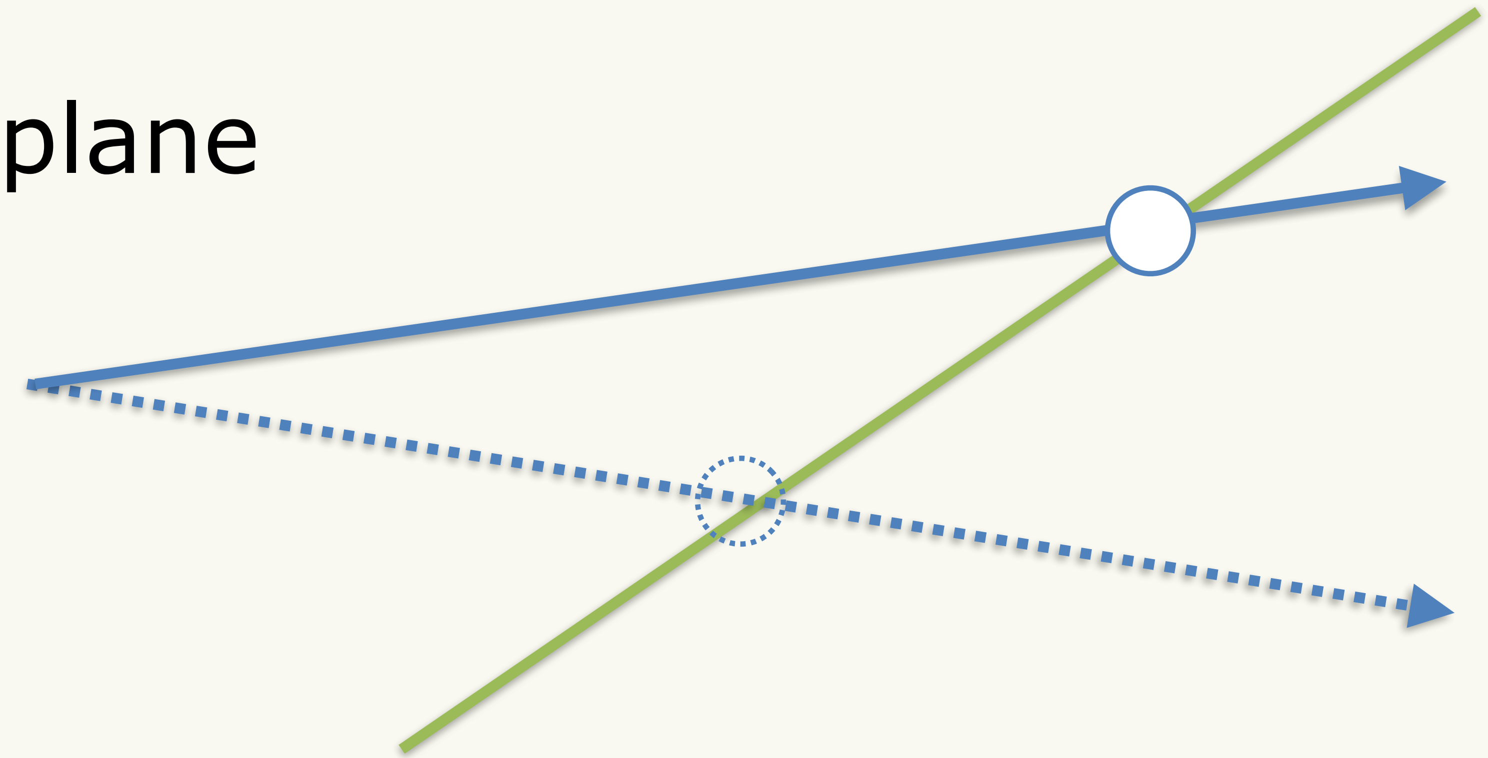
Where it hits the ground

Put an object there

Flicking 3D Demo

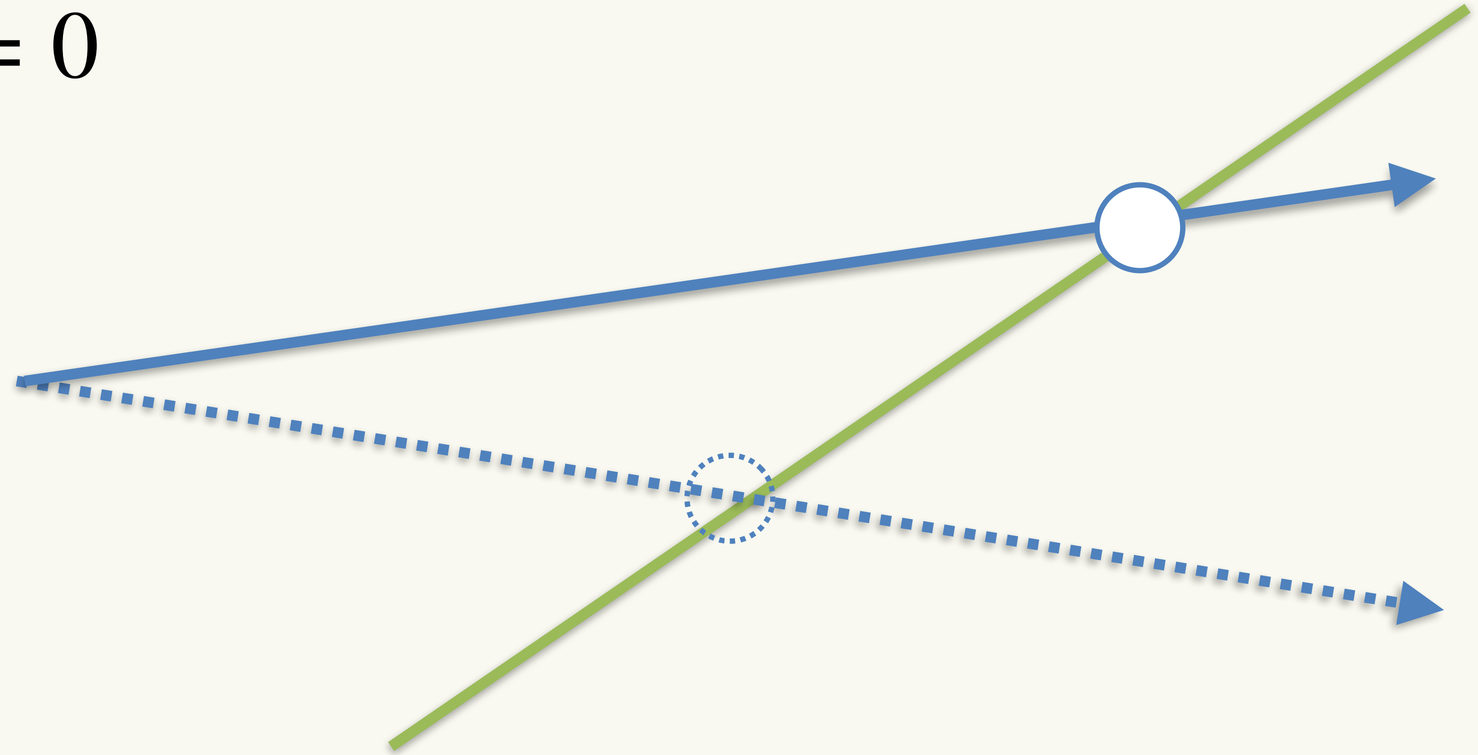
Flicking 3D

- Same as 2D
- Work in the plane
- Works best with an angled plane



Flicking 3D

- Math just like ground plane, but with more dot product
- Ray is $R, \vec{v}$; Plane is $P, \hat{n}$
- A point Q is on a plane if $(Q - P) \cdot \hat{n} = 0$
- Find t such that $((R + t\vec{v}) - P) \cdot \hat{n} = 0$
- $R \cdot \hat{n} + t\vec{v} \cdot \hat{n} - P \cdot \hat{n} = 0$
- $t\vec{v} \cdot \hat{n} = -R \cdot \hat{n} + P \cdot \hat{n}$
- $t = \dfrac{-R \cdot \hat{n} + P \cdot \hat{n}}{\vec{v} \cdot \hat{n}}$

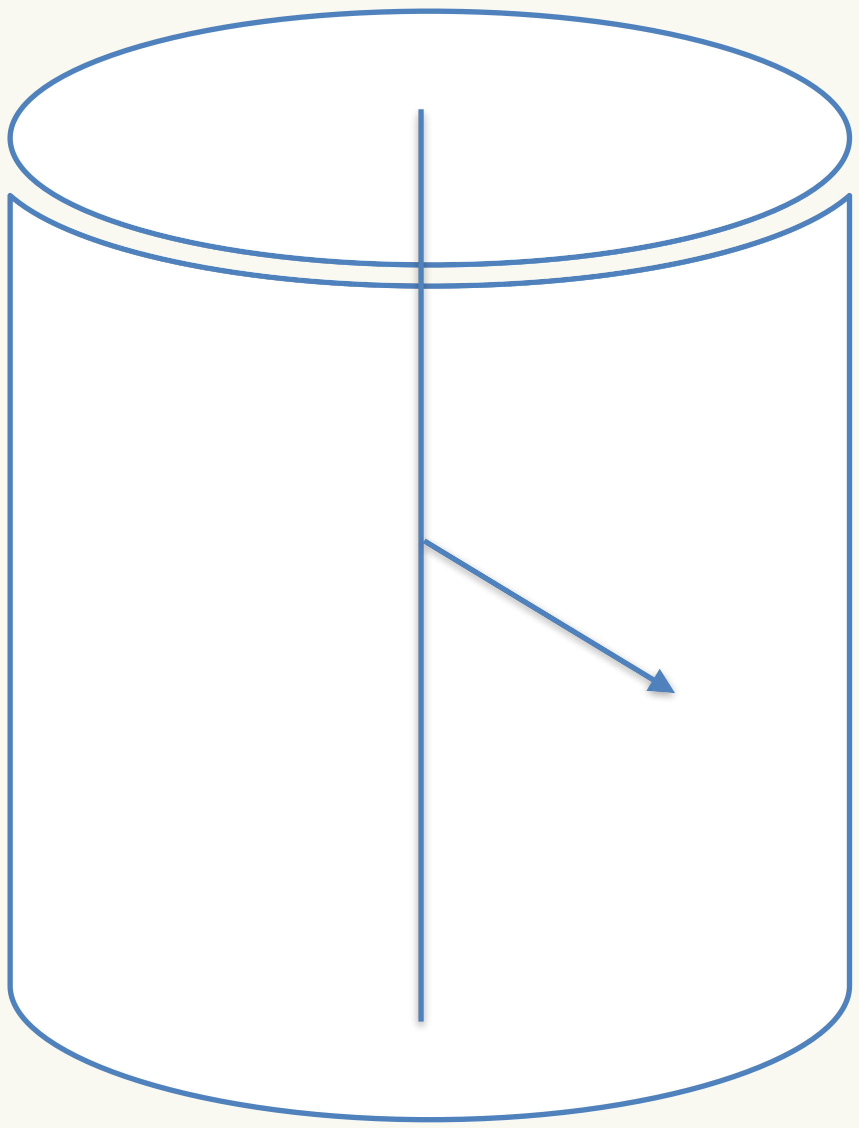


Flicking 3D

- Align the plane to the camera
- Especially in AR

Flicking (Carousels)

- Carousels are like flicking
- Project to cylinder
- Use Atan2 again to get an angle
- Measure angular velocity



Latency Reduction

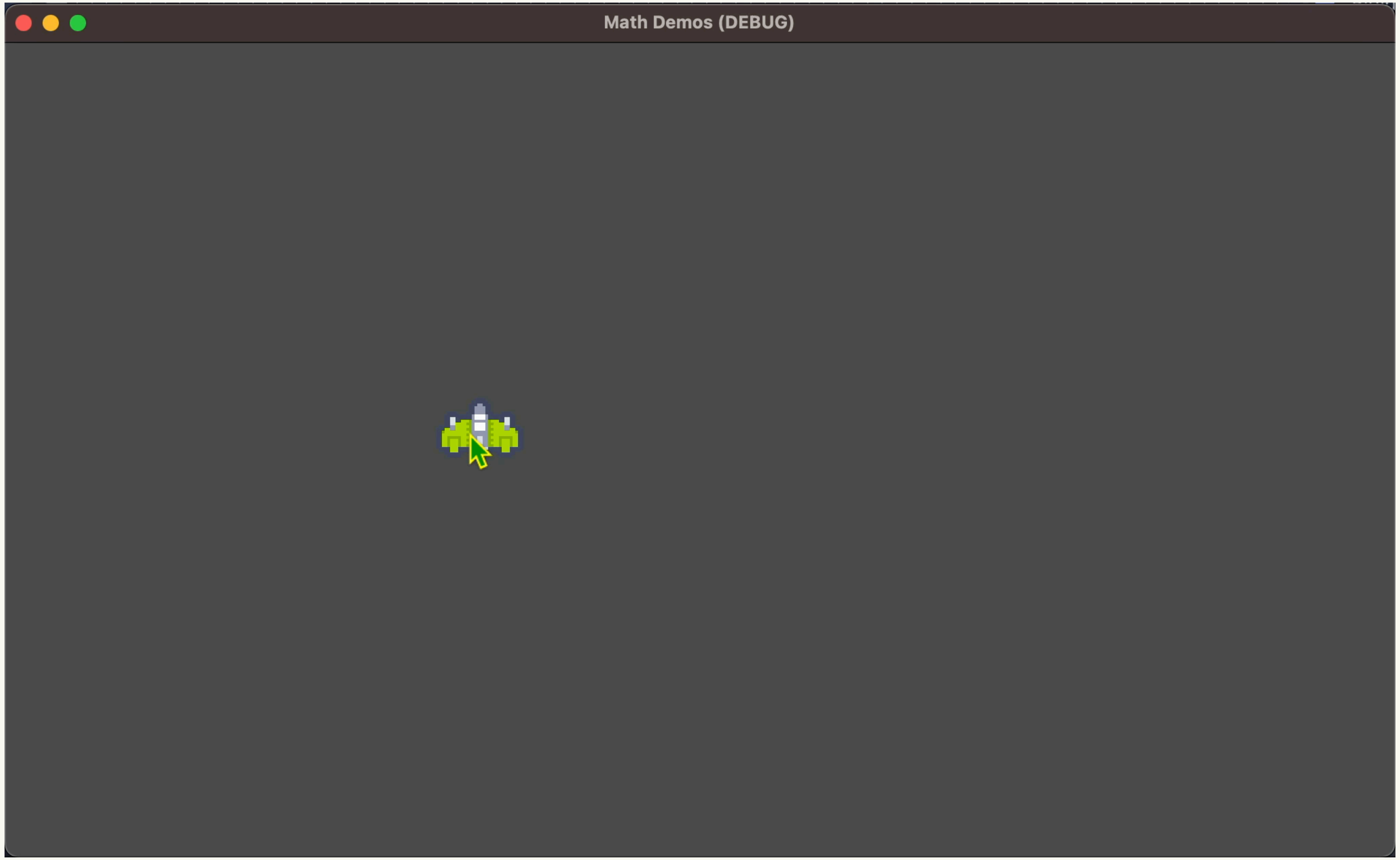
(Perceived)

Latency Reduction

Latency Reduction

- You can’t get rid of lag
- So lean into it (springs)
- Adds weight to the interaction

Latency Reduction



Latency Reduction

- Springs already in your physics engine

Latency Reduction

- But if not, $F_s = -kx$

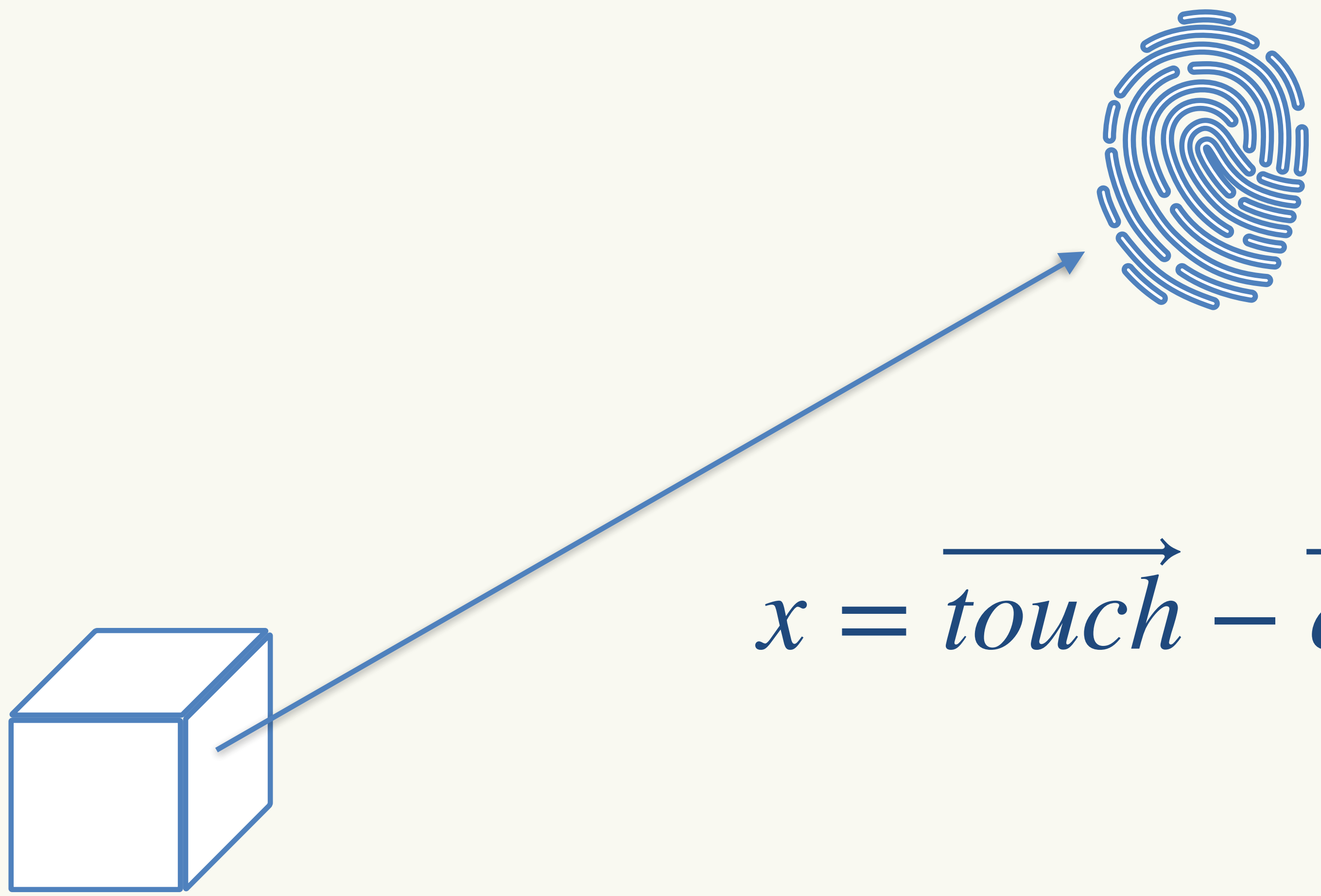
Distance from "rest" length
(Distance from touch)

Spring force

Spring strength

Latency Reduction

- But if not, $F_s = -kx$



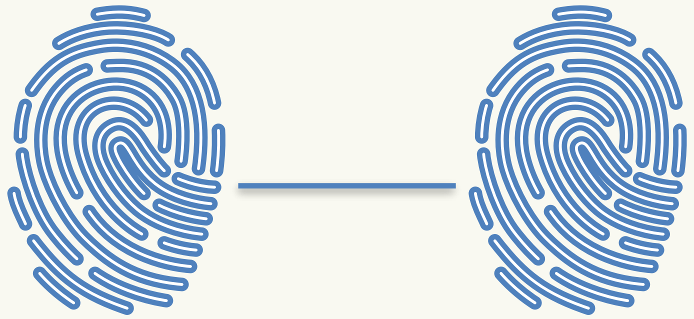
$$x = \overrightarrow{touch} - \overrightarrow{obj}$$

Latency Reduction

```
func _physics_process(_delta):
	if _pressed:
		var distance = _spring_to - transform.origin
		apply_central_impulse(distance * spring_force)
```

Pinch to Zoom

Pinch to Zoom



Just measure the distance?

Pinch to Zoom

- Works for 3D
- Especially if the camera doesn't just linearly zoom
- But for 2D, we want to glue the camera to the fingers

Pinch to Zoom

- 3 Components
- Scale - most important
- Pan - useful
- Rotate - you don’t always want this

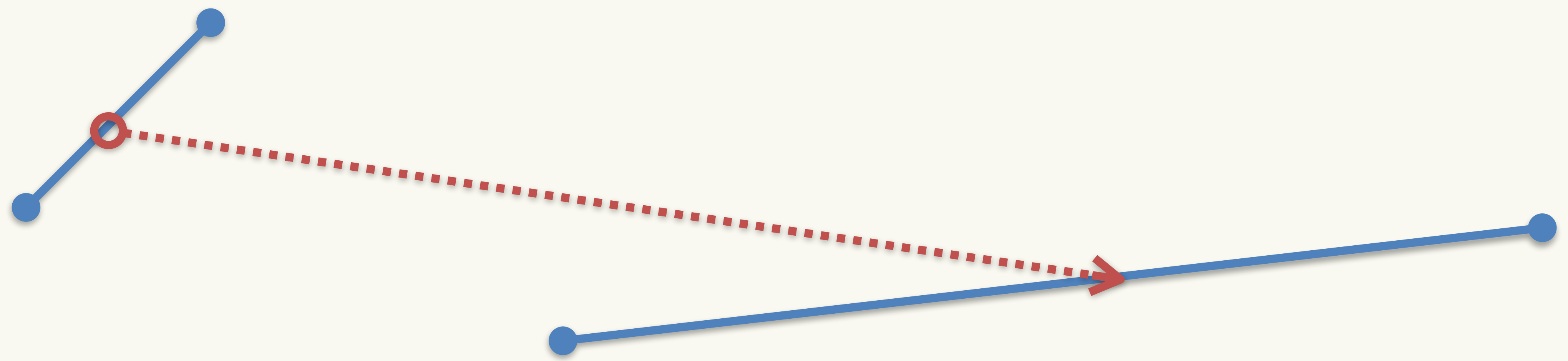
Pinch to Zoom

Naïve solution:

- Break into parts
- Apply each

Translation

Track how the middle moves over time



```
func _process_translation():
	# figure out translation
	var start_center = (_touches[0].first_touch + _touches[1].first_touch) * .5
	var end_center = (_touches[0].last_touch + _touches[1].last_touch) * .5
	var touch_delta = end_center - start_center

	# subtract the delta since we're moving opposite the fingers
	offset = _start_offset - touch_delta
```

Scale

Track how the length changes over time

```
func _process_scale():
	# figure out scale
	var start_length = (_touches[0].first_touch - _touches[1].first_touch).length()
	var end_length = (_touches[0].last_touch - _touches[1].last_touch).length()

	# inverse what we'd expect because we're in the view transform
	var scale = start_length / end_length
	zoom = scale * _start_zoom
```

Rotation

Track the rotation over time (Atan2)

```
func _process_rotation():
    # figure out rotation
    var start_vector = _touches[0].first_touch - _touches[1].first_touch
    var end_vector = _touches[0].last_touch - _touches[1].last_touch
    var start_rotation = atan2(start_vector.y, start_vector.x)
    var end_rotation = atan2(end_vector.y, end_vector.x)
    var delta = end_rotation - start_rotation

    # inverse because we're in view space
    rotation = _start_rotation - delta
```

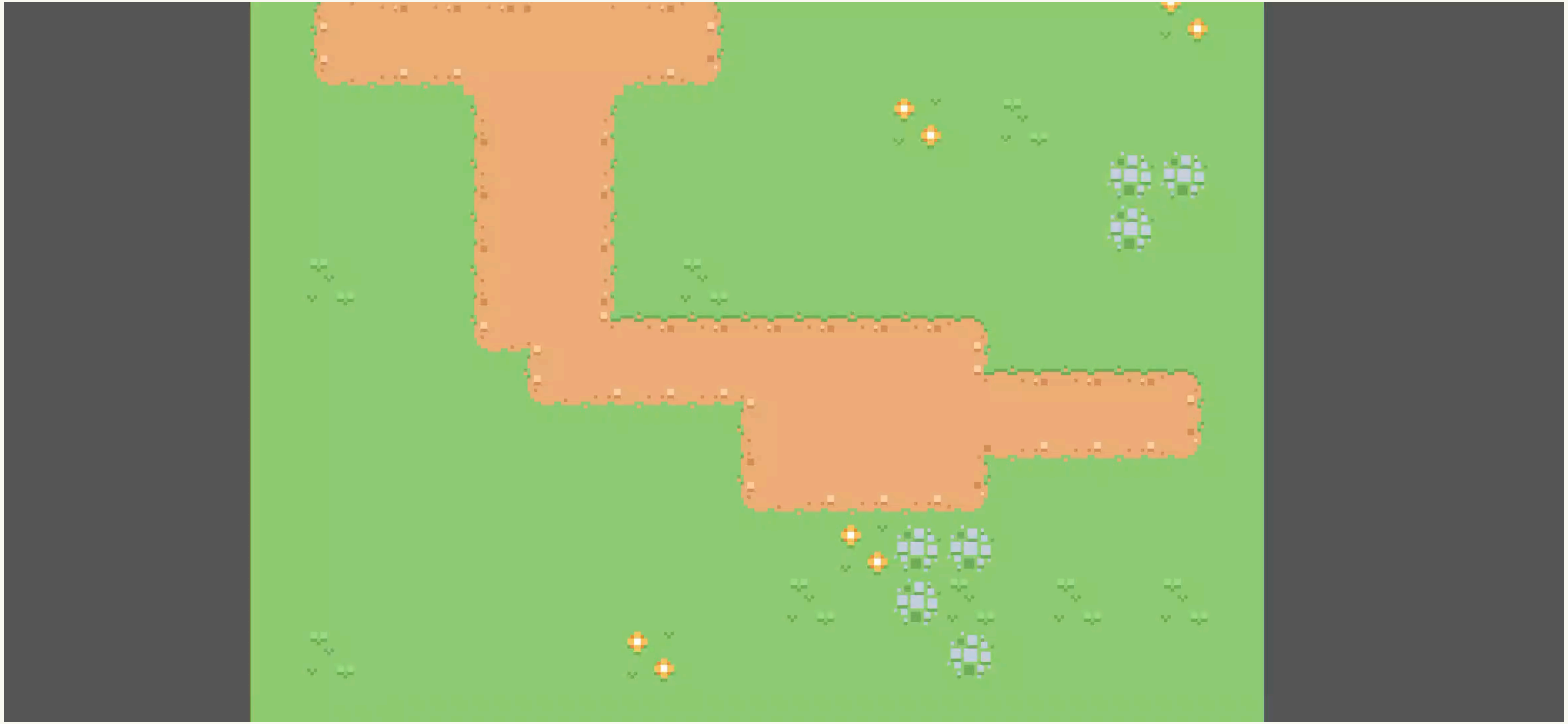
Pinch to Zoom

Put it all together and it works right?

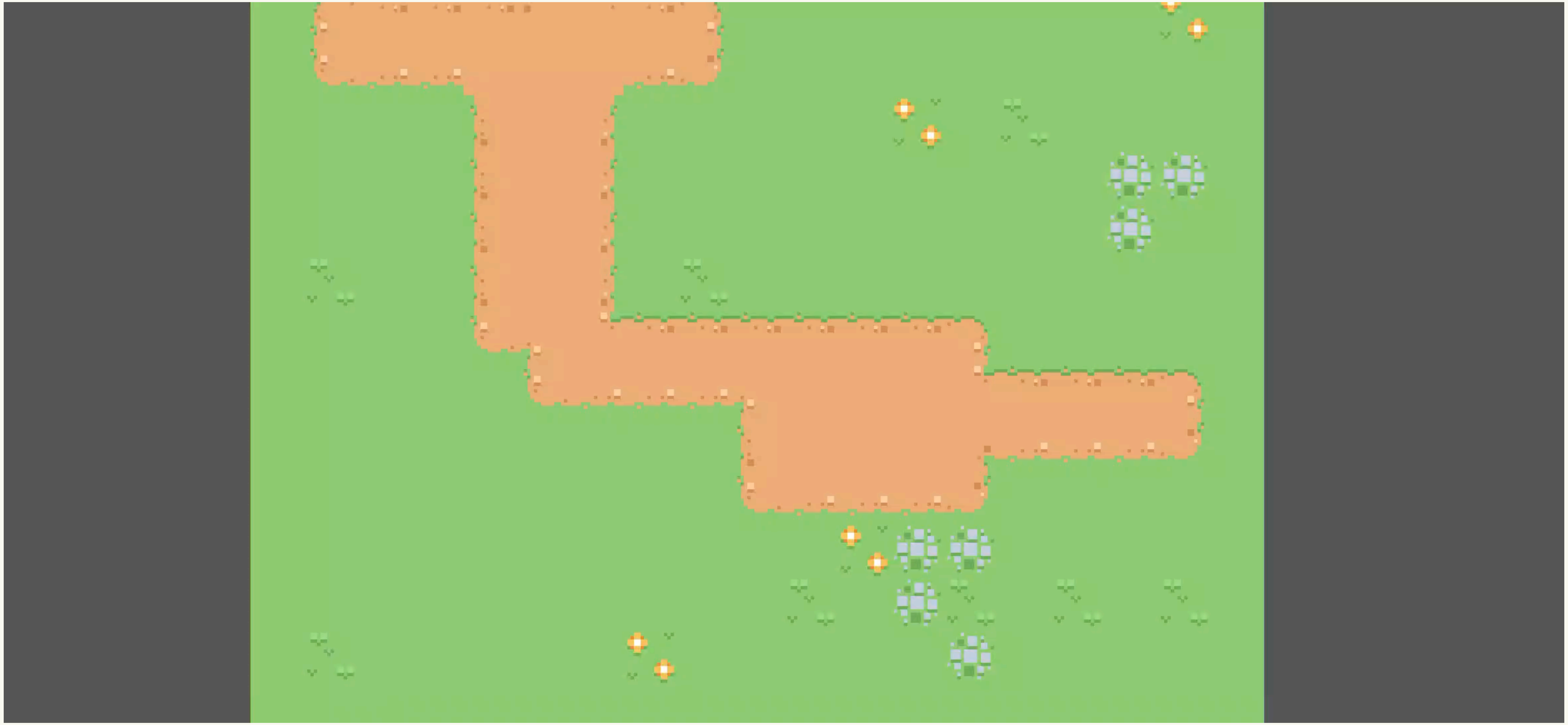
(No)

```
func _process(_delta):
    if len(_touches) == 2:
        _process_translation()
        _process_scale()
        _process_rotation()
```

Pinch to Zoom (bad)



Pinch to Zoom (good)



Pinch to Zoom (good)

- Moved all math into world space (remember picking?)

```
var viewport = get_viewport()
var view_transform = viewport.canvas_transform

var start_0 = view_transform.xform_inv(_touches[0].first_touch)
var end_0 = view_transform.xform_inv(_touches[0].last_touch)
var start_1 = view_transform.xform_inv(_touches[1].first_touch)
var end_1 = view_transform.xform_inv(_touches[1].last_touch)
```

Pinch to Zoom (good)

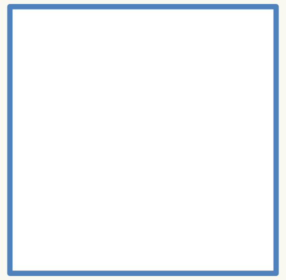
- I repositioned everything around the center of the touch…

```
var center_start = (start_0 + start_1) * .5
var center_end = (end_0 + end_1) * .5

var working = Transform2D(0, -center_start) * _start_transform
```

Pinch to Zoom (good)

- Scale and Rotation always apply to the “origin”



Pinch to Zoom (good)

- But we want it from the middle of our pinch

Pinch to Zoom (good)

- Three part process:
- Move the center to the origin T^{-1}
- Scale (or rotate) S
- Restore the center to where it belongs T
- $\vec{v}' = TST^{-1}\vec{v}$

Pinch to Zoom (finish up)

- Rotate

```
# rotate
var start_vector = start_1 - start_0
var end_vector = end_1 - end_0
var start_rotation = atan2(start_vector.y, start_vector.x)
var end_rotation = atan2(end_vector.y, end_vector.x)
var delta = end_rotation - start_rotation
working = working.rotated(delta)
```

Pinch to Zoom (finish up)

- Scale

```
# scale
var start_length = (start_1 - start_0).length()
var end_length = (end_1 - end_0).length()
var scale = end_length / start_length
working = working.scaled(Vector2(scale, scale))
```

Pinch to Zoom (finish up)

- Undo the translation then apply our touch offset

```
working = Transform2D(0, center_start) * working
working = Transform2D(0, center_end - center_start) * working
transform = working
```

Sensor Stuff

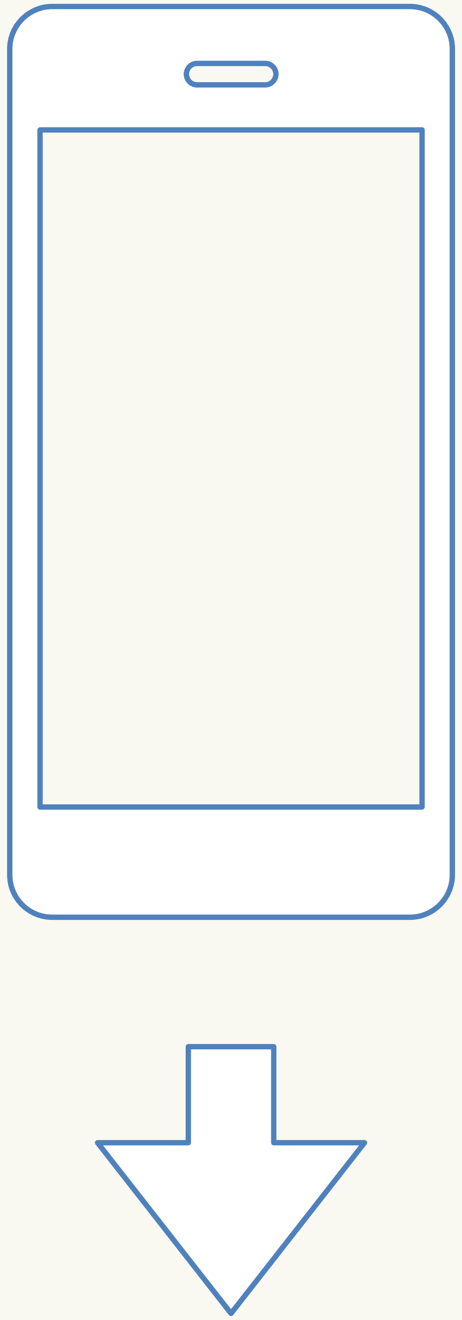
Everything but the touchscreen

Sensor stuff

- **IMU = Inertial Measurement Unit**
- All phones have this
  - How auto-rotate works!

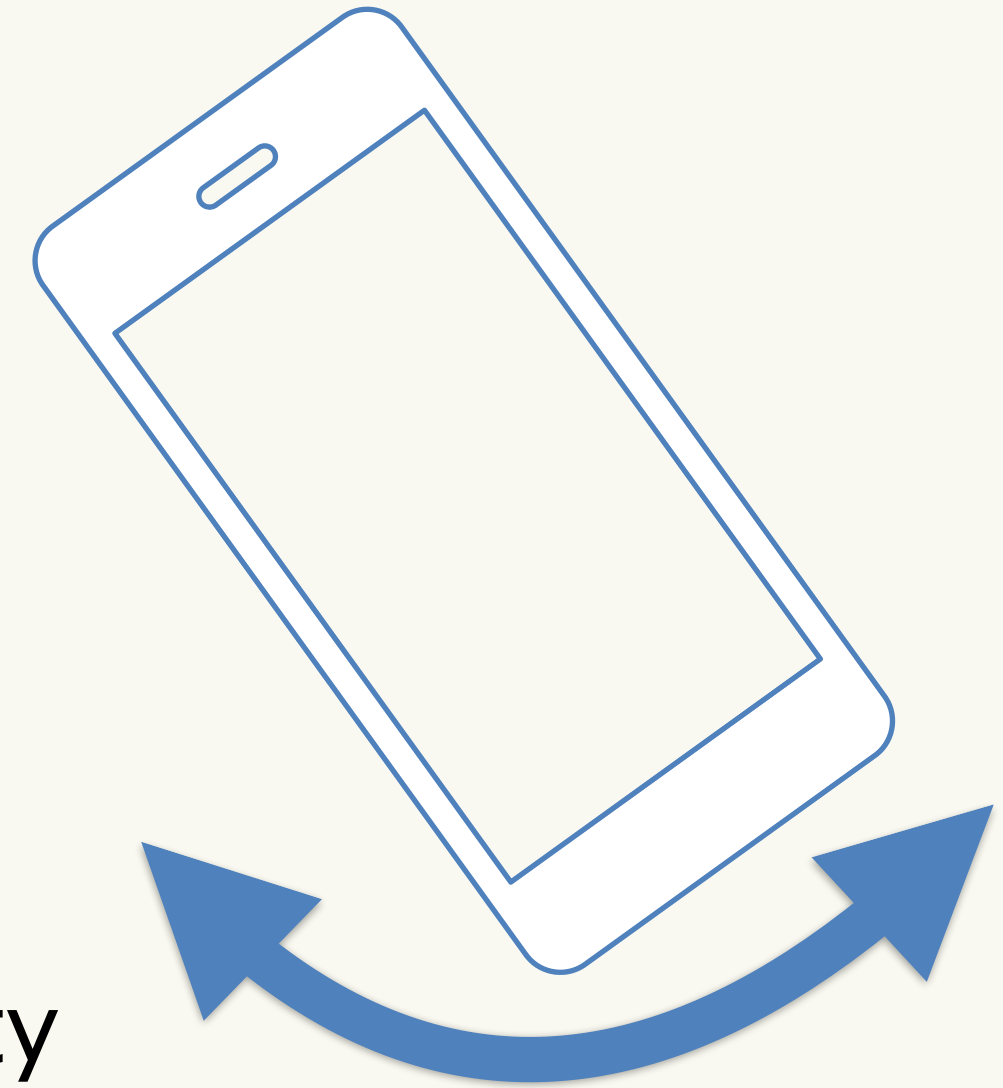
IMU

- **Accelerometer** most common
- Most reliable
- Measures acceleration
  - As a vector
  - Usually gravity
- Can’t “twist”
- Don’t fall for double integral



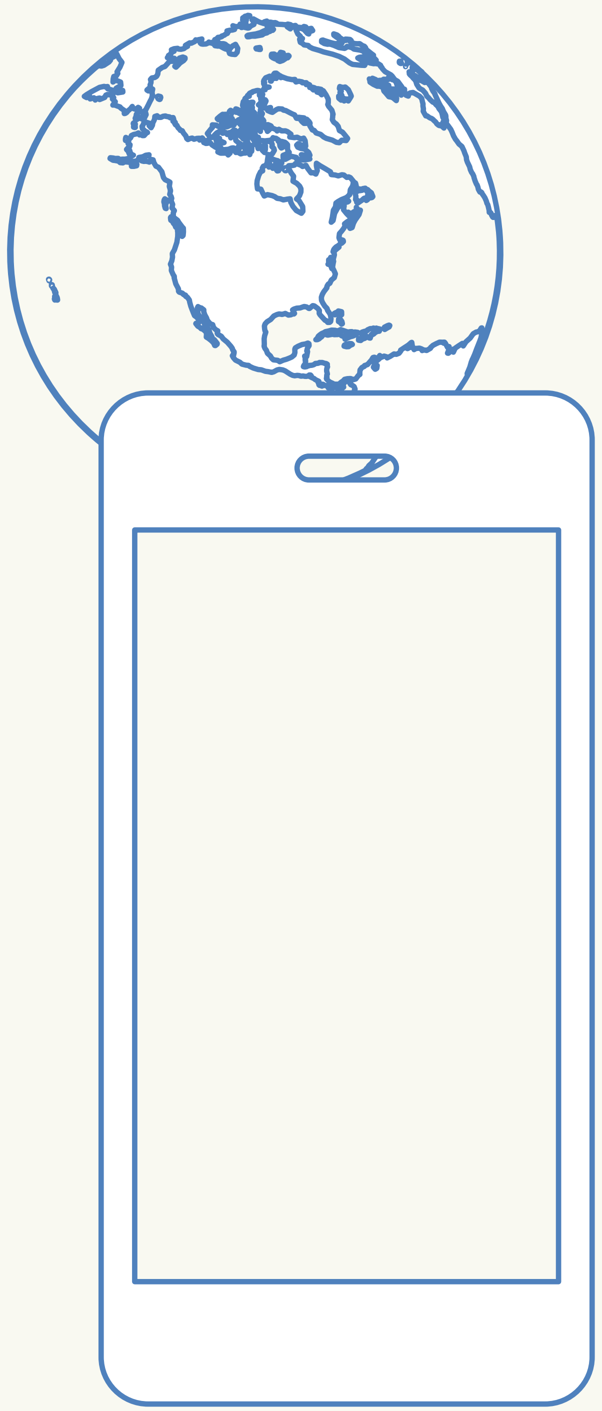
IMU

- **Gyroscope** usually included
- Measures change in orientation
- Often read as absolute orientation
  - As a quaternion
  - Thanks to accelerometer finding gravity
- Drifts on Yaw
  - Interesting for VR
- Phones don’t always calibrate well = Drift



IMU

- **Magnetometer** is fairly common
- Reads magnetic fields
  - As a Vector
  - Usually Earth's
- Would give you perfect orientation… if it worked
  - Everything messes with it



GDC | March 20-24, 2023
San Francisco, CA

Tilt Controls

#GDC23

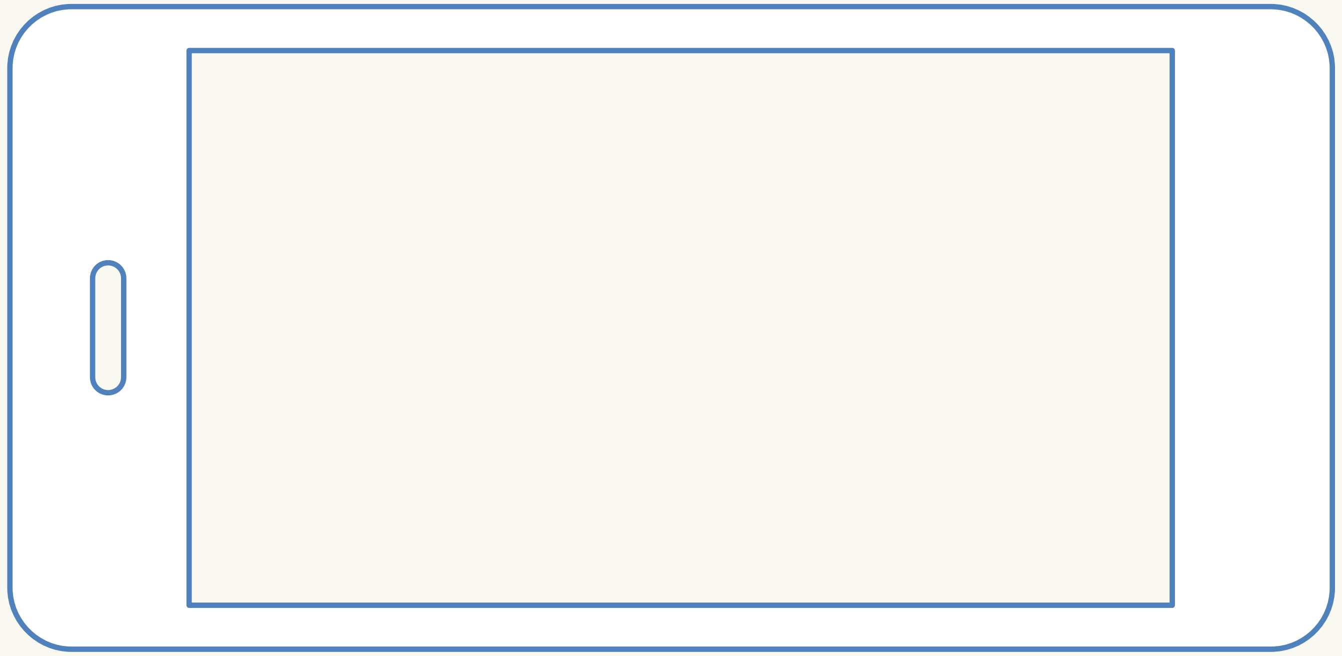
Tilt Controls

Making a tilt “marble maze” kind of game, what do you use?

- Accelerometer
- Gyroscope
- Magnetometer

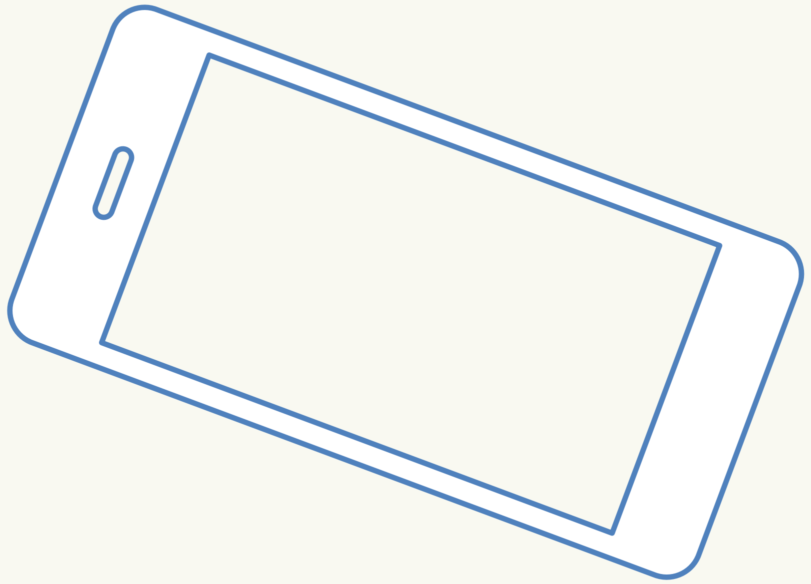
Tilt Controls

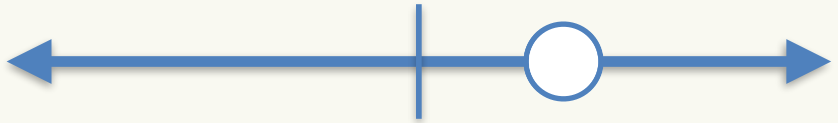
- Want offset in a 2D plane
- X (left/right) is usually fixed
- Y/Z needs to be calibrated



Tilt Controls (x)

- If you normalize acceleration
  - x = 0 not tilted
  - x = 1 or -1 full tilt
- Looks a lot like $sin(\theta)$
- So $asin(accel_x) = \theta$ or how much we're tilting





Tilt Controls (y/z)

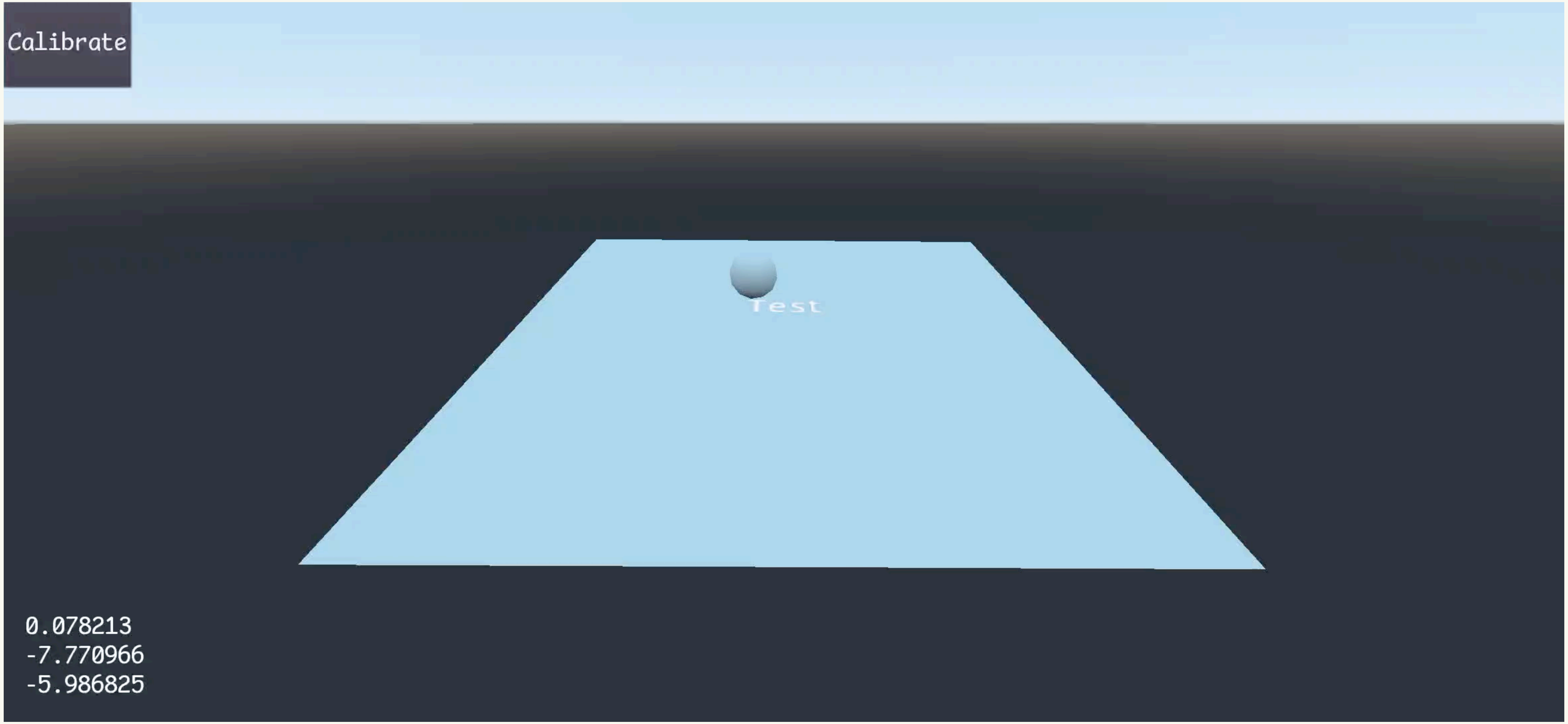
- Need to calibrate
- Have two axis that feed into it
- Sounds like $atan2()$ time!
- “Calibration” is saving an angle and restoring

Tilt Controls

```
func calibrate():
	_calibration = read_accelerometer()


func _process(_delta):
	if not _calibration:
		calibrate()
	var down = read_accelerometer()
	var roll = -asin(down.x)
	var pitch = atan2(down.y, down.z) - atan2(_calibration.y, _calibration.z)
	transform = Transform(Quat(Vector3(pitch, 0, roll)))
```

Tilt Controls



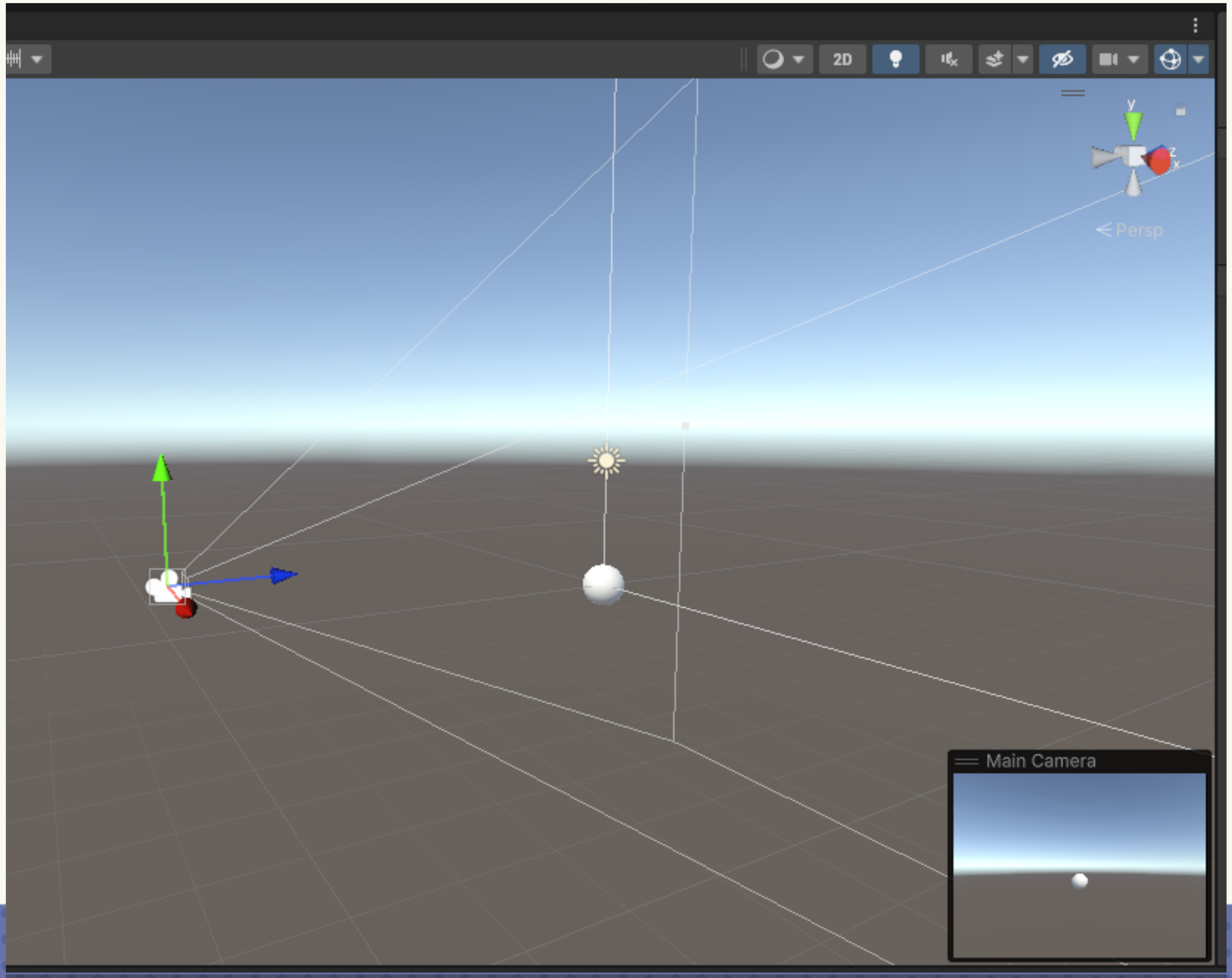
Gyroscope

Gyroscope

- Great for when you need to rotate around gravity
  - Like a 1st person camera
- There will be drift
  - Reads angular change
  - Often can get “attitude” (device orientation)
    - As a quaternion

Gyroscope

- Test scene - 1st person camera/look
  - Aim at sphere



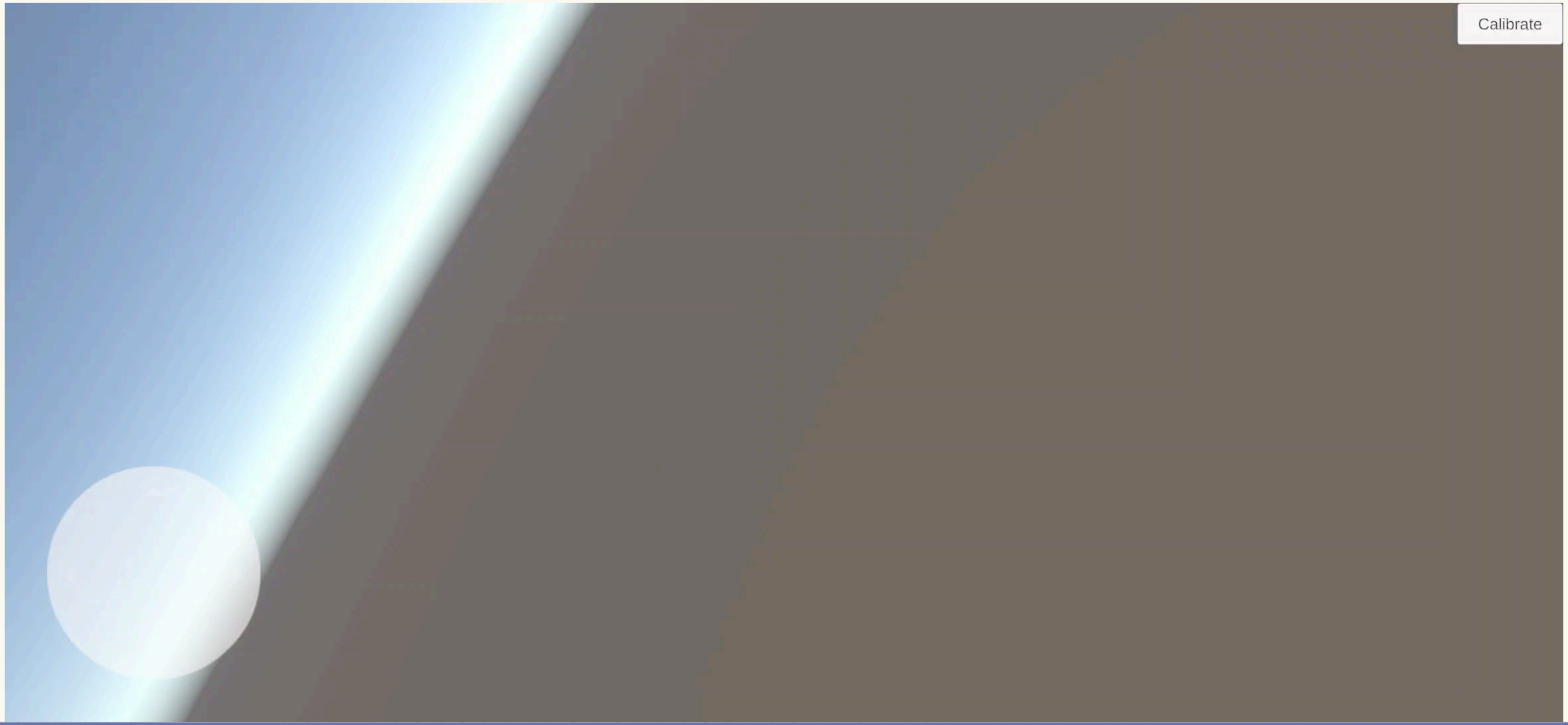
Gyroscope

- What could go wrong?

```
var orientation = ReadGyroscope();
transform.localRotation = orientation;
```

Gyroscope

- WOT?



Gyroscope

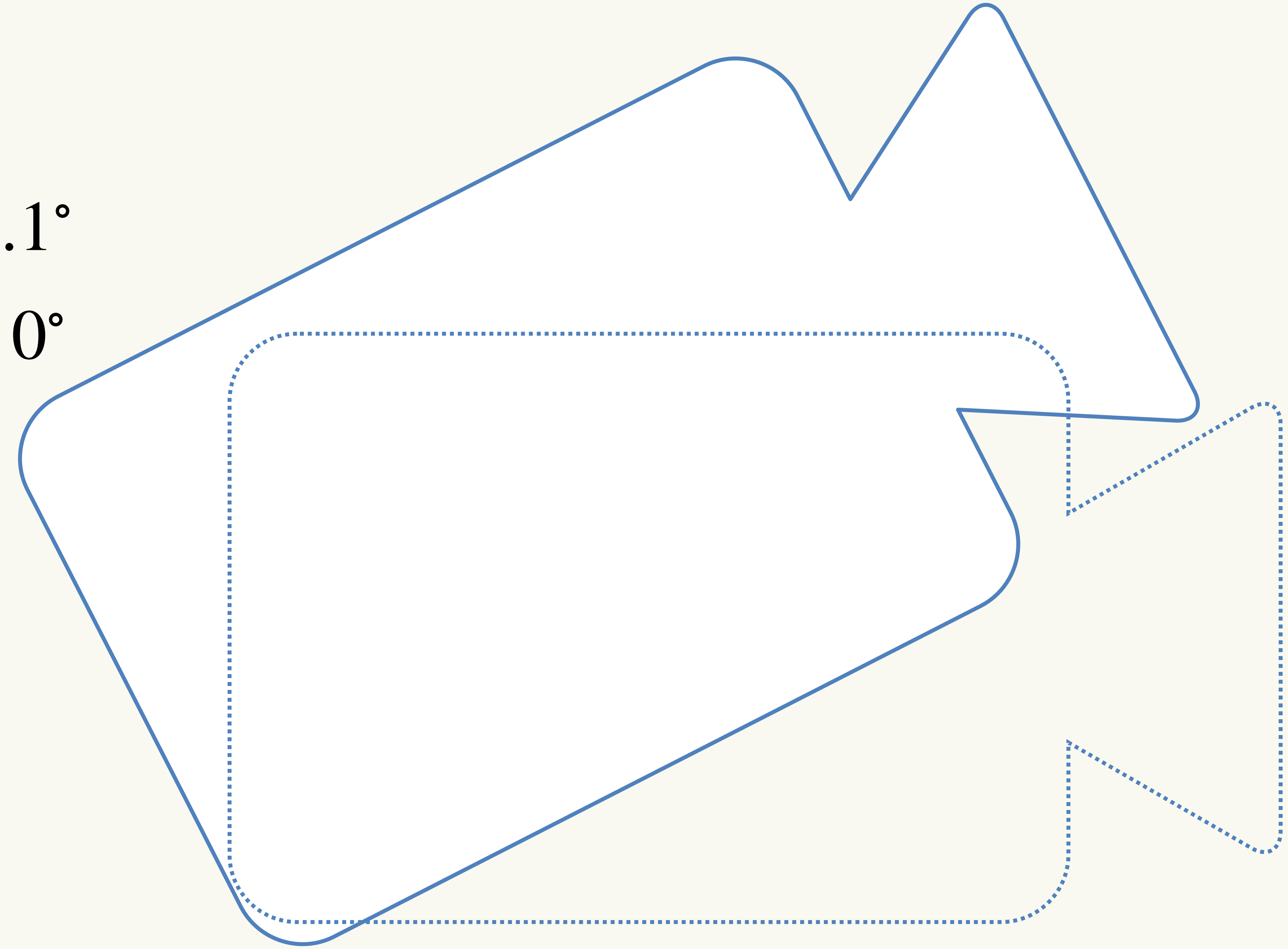
- Need to “calibrate”
- What is “calibration?”

Gyroscope

- Gyroscope “attitude” is an orientation in 3D space
- We need to “undo” this before applying a new one

Gyroscope

- 2D example
- Camera starts at $\theta = 27.1°$
- We want this to be $\theta' = 0°$



Gyroscope

- Simply subtract 27.1° from every new reading
- $f(reading) = reading - 27.1°$
- How do we do this in quaternions?

Gyroscope

$$f(reading) = reading - 27.1°$$

Gyroscope

$$f(reading, calibration) = reading - calibration$$

Gyroscope

Quaternion Multiply

$$f(reading, calibration) = reading + (-calibration)$$

Quaternion Inverse

Gyroscope

$$f(Q_{reading}, Q_{calibration}) = Q_{calibration}^{-1} Q_{reading}$$

Gyroscope

```
public void Calibrate() {
    _calibration = ReadGyroscope();
}
```

```
var orientation = Quaternion.Inverse(_calibration) * ReadGyroscope();
transform.localRotation = orientation;
```

Gyroscope

Quaternion

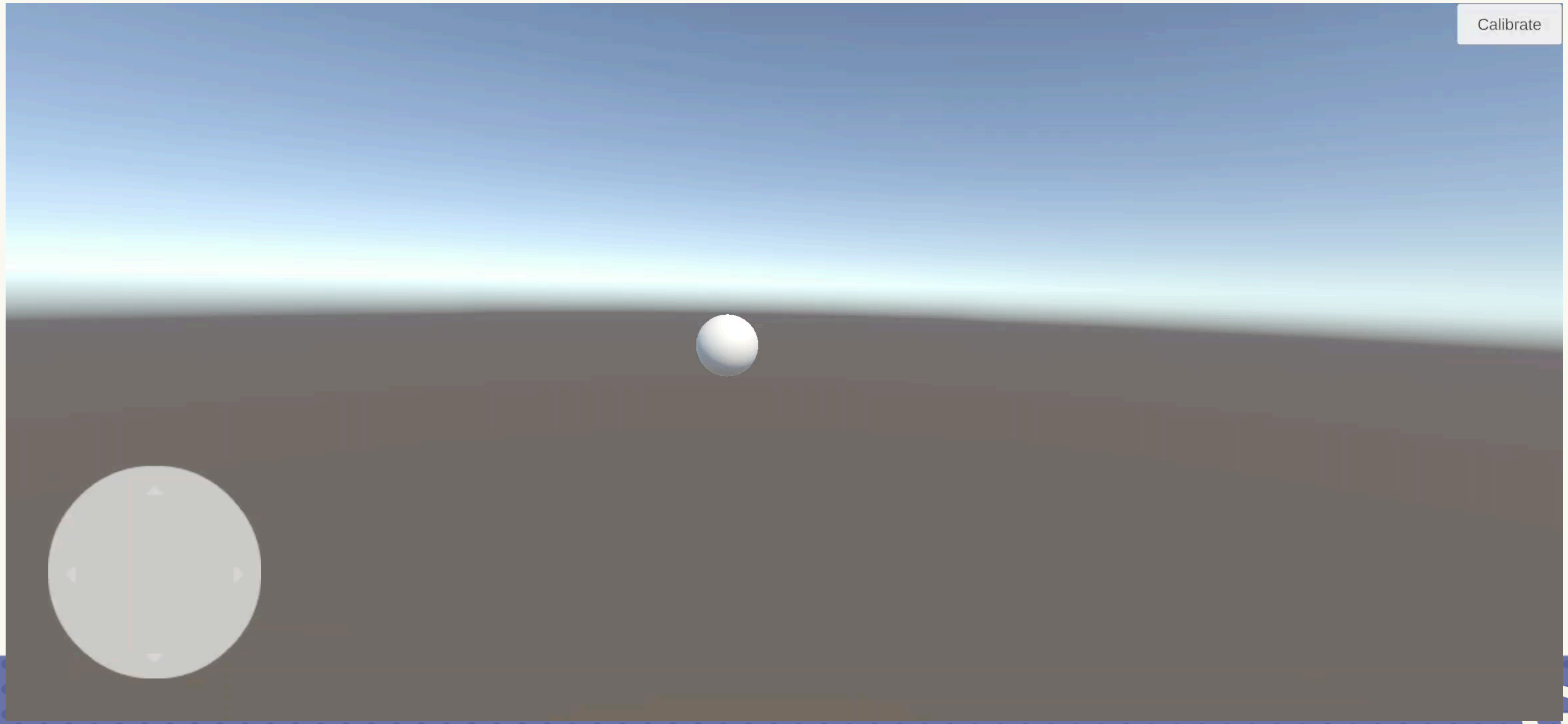
- Need to convert to “game engine” space
- See my Quaternion talk at a previous summit
- Unity tells us do this in their docs, so I copied it

```
private Quaternion ReadGyroscope(){
    var reading = AttitudeSensor.current.attitude.ReadValue();
    return new Quaternion(reading.x, reading.y, -reading.z, -reading.w);
}
```

Quaternion

Quaternion

- Note that Unity doesn't use TYPE_GAME_ROTATION_VECTOR
  - This means the magnetometer factors in
  - You'll do this near metal:



Quaternion

- Correct “up” vector
- Avoid tilting sideways (good outside VR)
- Really quick (avoid too much quaternion math)
  - There is a quaternion talk later today!

Quaternion - correct “up”

```
var refRight = Vector3.Cross(transform.forward, Vector3.up);
var targetUp = Vector3.Cross(refRight, transform.forward);
var angle = Mathf.Atan2(
    Vector3.Dot(transform.up, targetUp),
    Vector3.Dot(transform.up, refRight)) - Mathf.PI / 2f;
transform.localRotation = Quaternion.AngleAxis(angle * Mathf.Rad2Deg, transform.forward)
    * transform.localRotation;
```

Quaternion - correct “up”

Use “forward” to find “right” with no tilt
Use “right” to find “up” without rolling

```
var refRight = Vector3.Cross(transform.forward, Vector3.up);
var targetUp = Vector3.Cross(refRight, transform.forward);
var angle = Mathf.Atan2(
    Vector3.Dot(transform.up, targetUp),
    Vector3.Dot(transform.up, refRight)) - Mathf.PI / 2f;
transform.localRotation = Quaternion.AngleAxis(angle * Mathf.Rad2Deg, transform.forward)
    * transform.localRotation;
```

Quaternion - correct "up"

Atan2: angle between current "up" and desired "up"

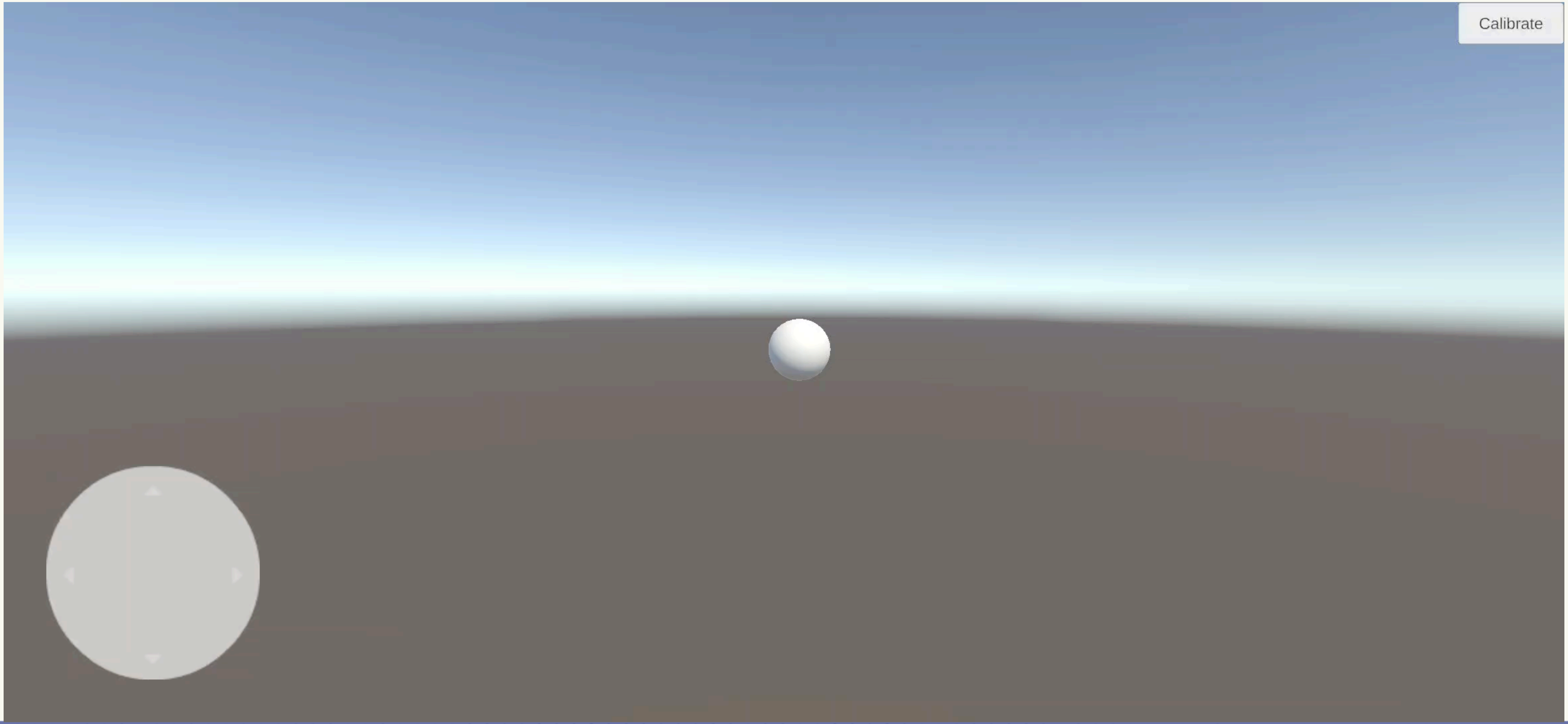
```
var refRight = Vector3.Cross(transform.forward, Vector3.up);
var targetUp = Vector3.Cross(refRight, transform.forward);
var angle = Mathf.Atan2(
    Vector3.Dot(transform.up, targetUp),
    Vector3.Dot(transform.up, refRight)) - Mathf.PI / 2f;
transform.localRotation = Quaternion.AngleAxis(angle * Mathf.Rad2Deg, transform.forward)
    * transform.localRotation;
```

Quaternion - correct "up"

Multiply it all in!

```
var refRight = Vector3.Cross(transform.forward, Vector3.up);
var targetUp = Vector3.Cross(refRight, transform.forward);
var angle = Mathf.Atan2(
    Vector3.Dot(transform.up, targetUp),
    Vector3.Dot(transform.up, refRight)) - Mathf.PI / 2f;
transform.localRotation = Quaternion.AngleAxis(angle * Mathf.Rad2Deg, transform.forward)
    * transform.localRotation;
```

Quaternion



Thanks!

Follow me:
@pux0r3 on Twitter and mastodon.gamedev.place
That's a zero and a silent 3 at the end