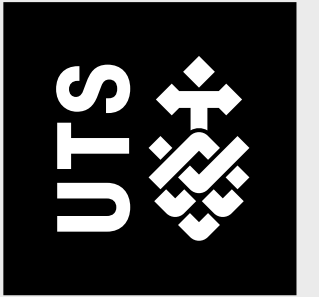


31263 / 32004

Intro to Games Programming

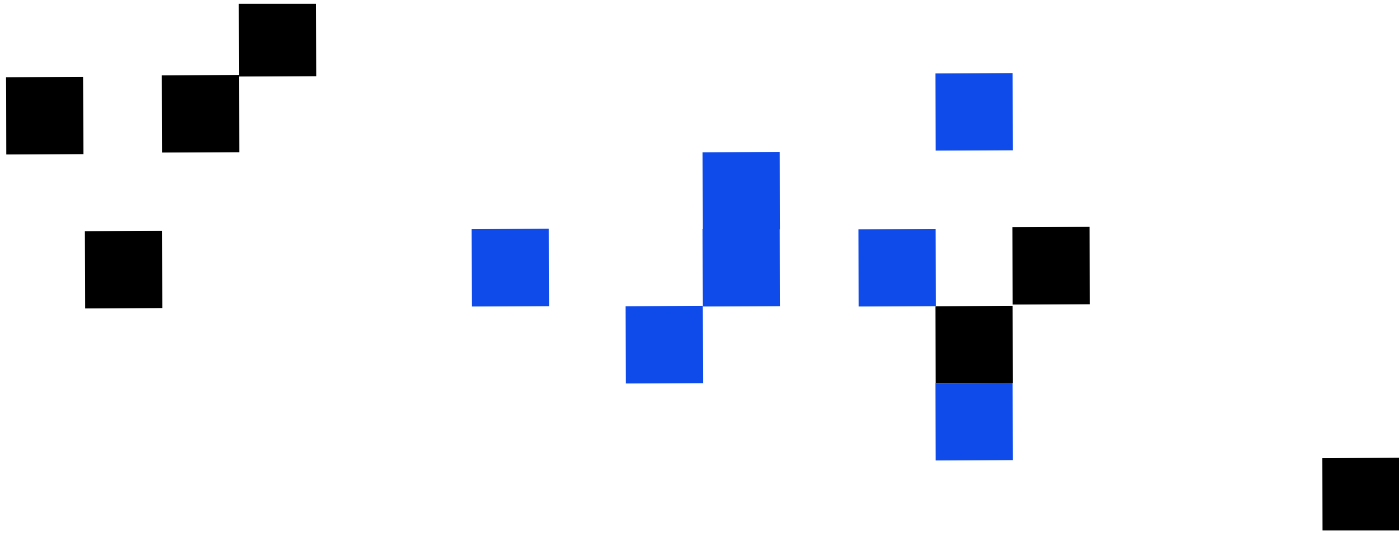
Week 8



Dr. William Raffe
william.raffe@uts.edu.au
Senior Lecturer
School of Software, FEIT



Overview

- **Moving with Input**
 - **Position**
 - **Rotation**
 - **Collisions**
 - **Collision Detection**
 - **Triggers**
 - **Layers**
 - **Collision Matrix**
 - **Ray Casting**
- 



■ Moving with Input - Position

- `transform.position += direction * maxSpeed * Time.deltaTime;`
 - For frame rate independent movement
- `maxSpeed` is a float variable
- `Time.deltaTime` is out of our control
- So direction is what our input needs to affect
- `transform.position += Vector3.up * 5.0f * Time.deltaTime`
- `transform.position += new Vector3(0.0f, 1.0f, 0.0f) * 5.0f * Time.deltaTime;`



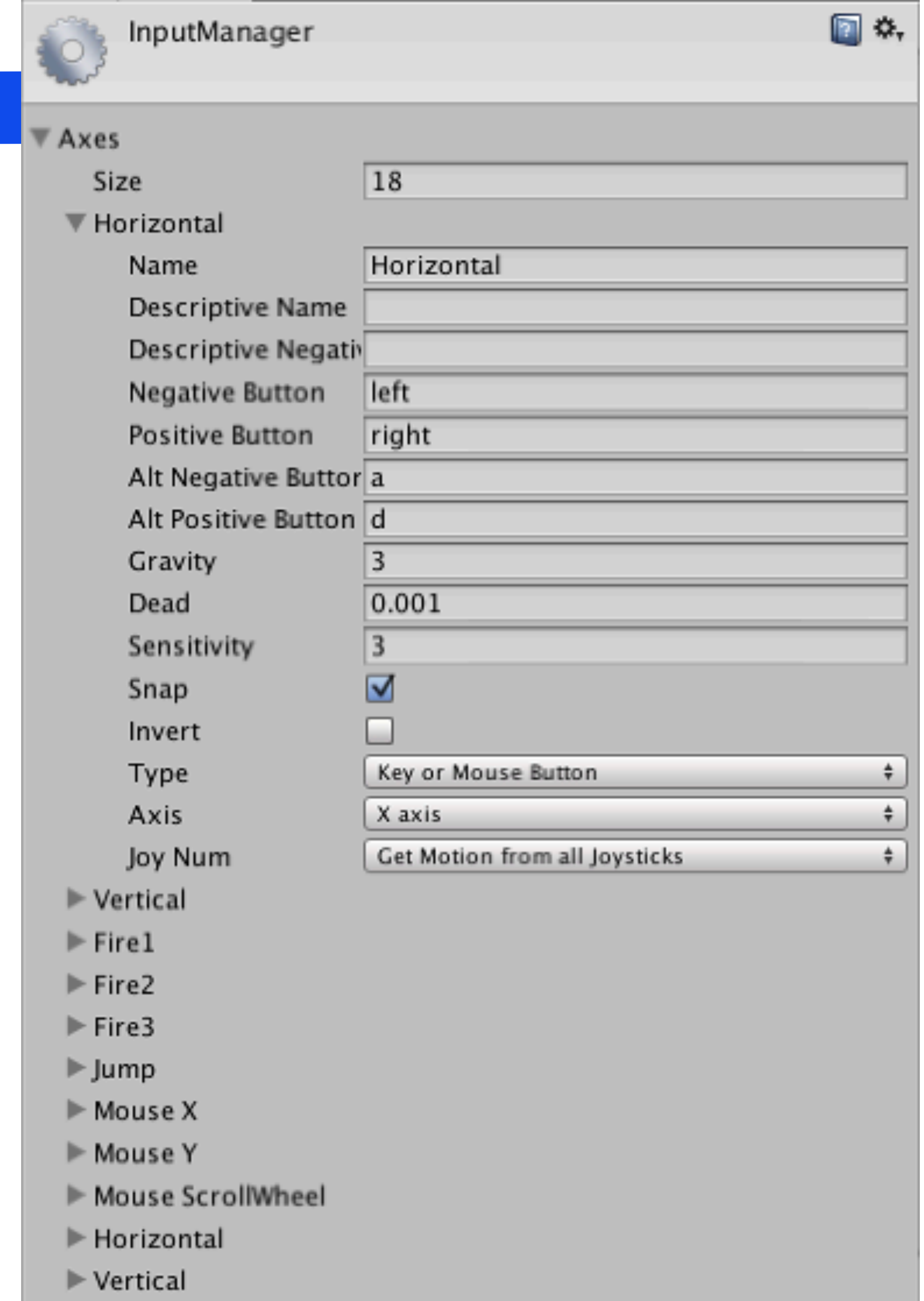
■ Moving with Input - Position

That is:

- We need a directional vector with....
- Values between $(-1.0, 1.0)$
 - A magnitude of 1.0 (important!)
 - Otherwise known as **“Unit Vector”**

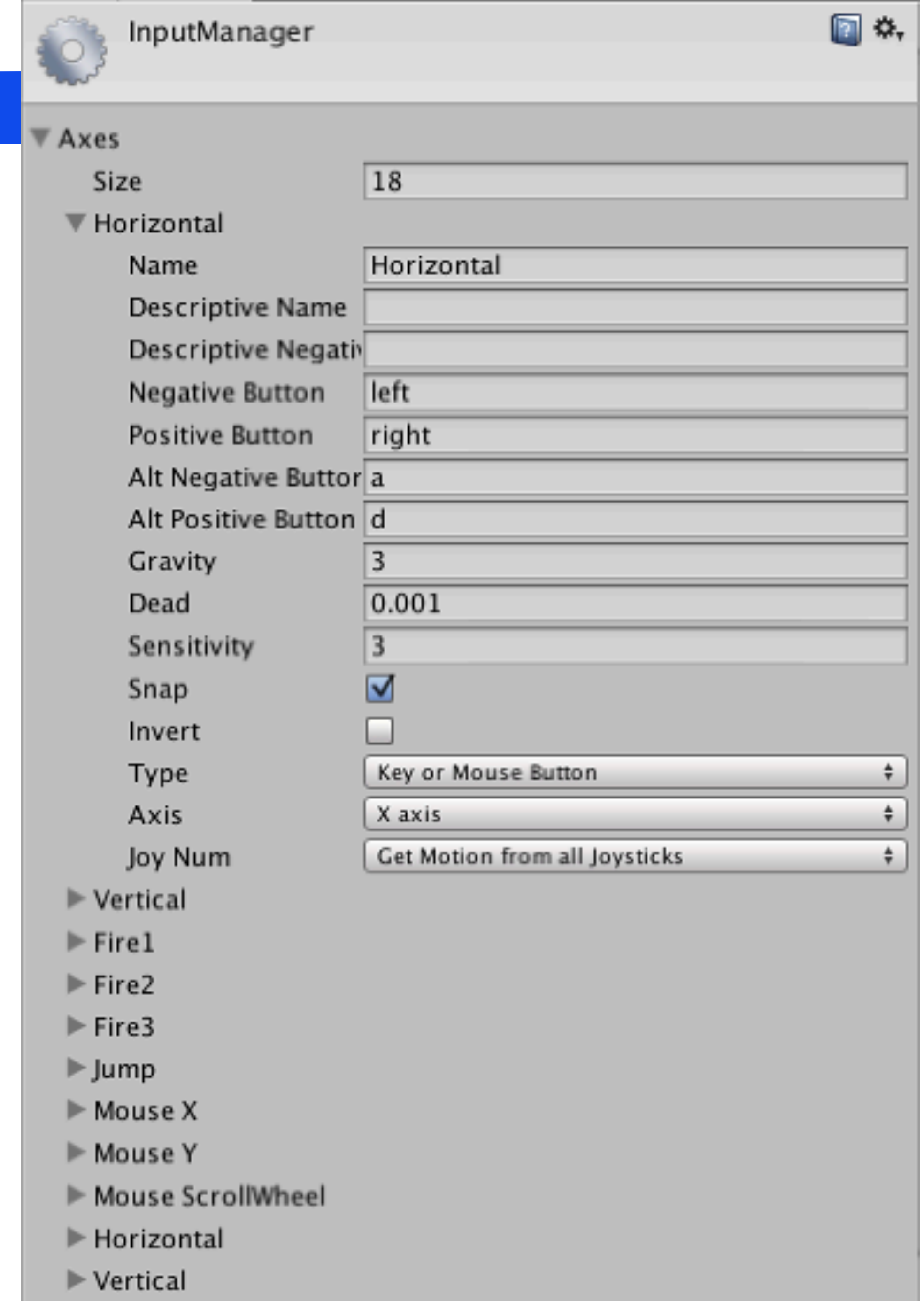
Moving with Input – Position Input Axes

- Buttons / keys are either down or up
- Axes return a value between -1.0 and 1.0
 - On a joystick, this is how far the joystick has been moved
 - On a keyboard/mouse, this is how long a button has been pressed down
 - “Sensitivity” and “Gravity” control Acceleration and Deceleration of axes



Moving with Input – Position Input Axes

- Buttons / keys are either down or up
- Axes return a value between -1.0 and 1.0
 - On a joystick, this is how far the joystick has been moved
 - On a keyboard/mouse, this is how long a button has been pressed down
 - Sensitivity and Gravity control Acceleration and Deceleration of axes





Moving with Input – Position

Input Axes

- `position.x += Input.GetAxis("Horizontal") * 5.0 * Time.deltaTime;`
- Ignoring `Time.deltaTime` for a moment:
 - If left key is held down then:
`position.x += -1.0 * 5.0;` ---- Moving -5 units on the x axis per frame
 - If right key is held down then:
`position.x += 1.0 * 5.0;` ---- Moving +5 units on the x-axis per frame
- Do this for all axis (x,y,z), for all values between (-1, 1), and you can move 5 units per frame in any direction.

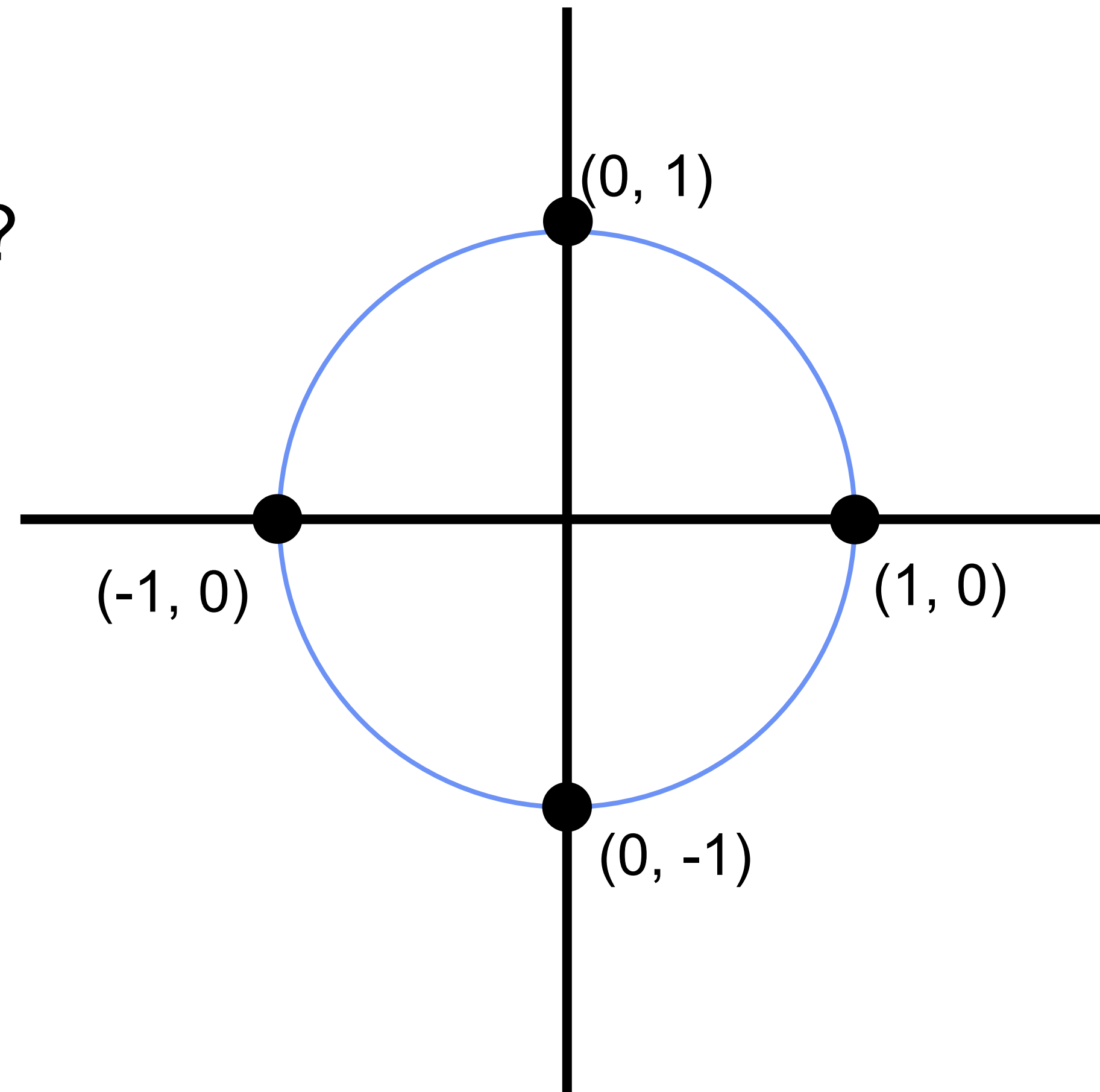


■ Magnitude of 1

- For the (x,y) vector:
 - (1.0, 0.0) has a magnitude of 1. 😊
 - (-1.0, 0.0) has a magnitude of 1. 😊
 - (0.0, 1.0) has a magnitude of 1. 😊
 - (0.5, 0.0) has a magnitude of 0.5. 😊
 - (1.0, 1.0) has a magnitude of (roughly)1.4. 🚫

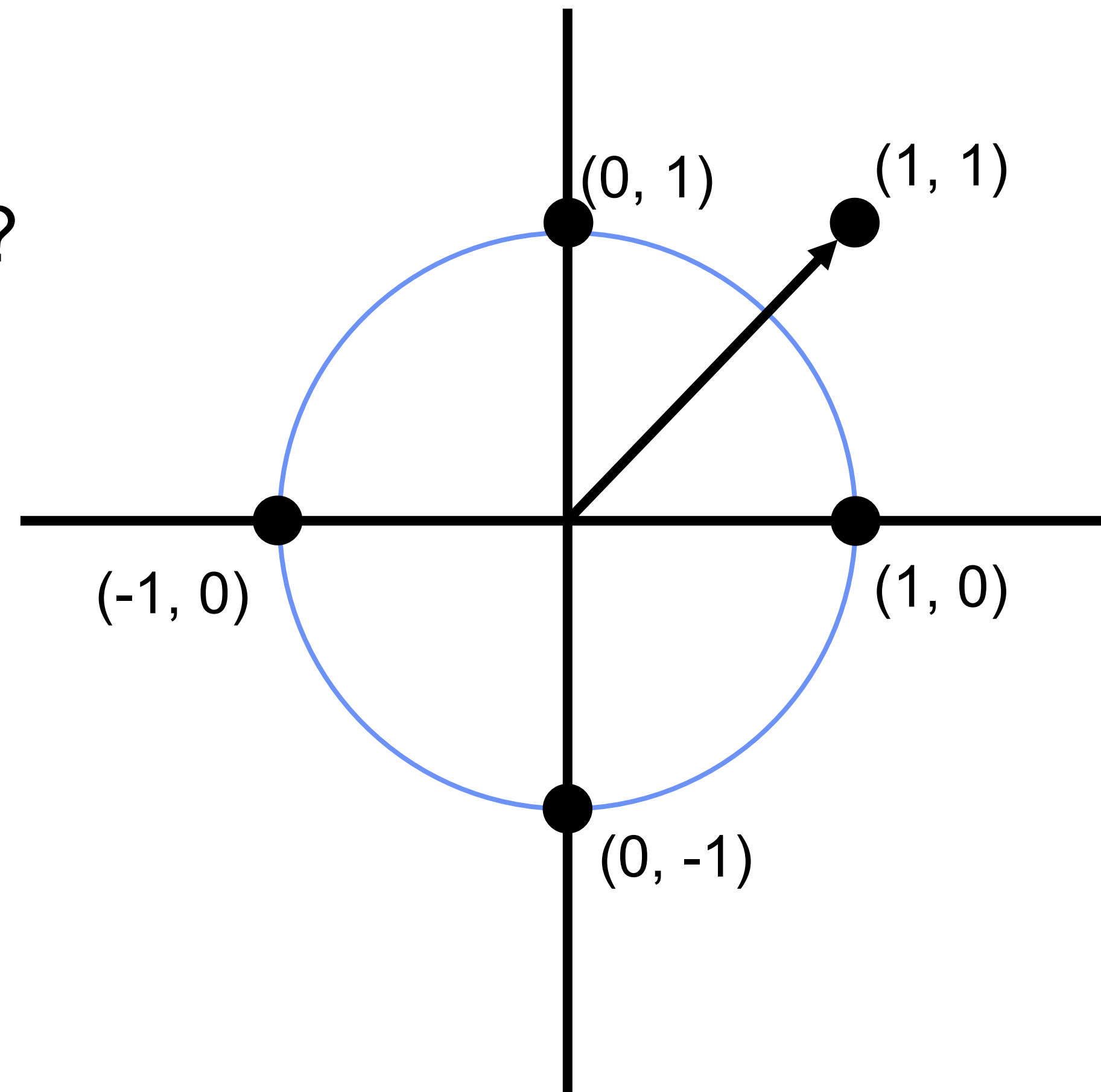
■ Vector Circle with Magnitude of 1

Where is point (1, 1)?



■ Vector Circle with Magnitude of 1

Where is point $(1, 1)$?



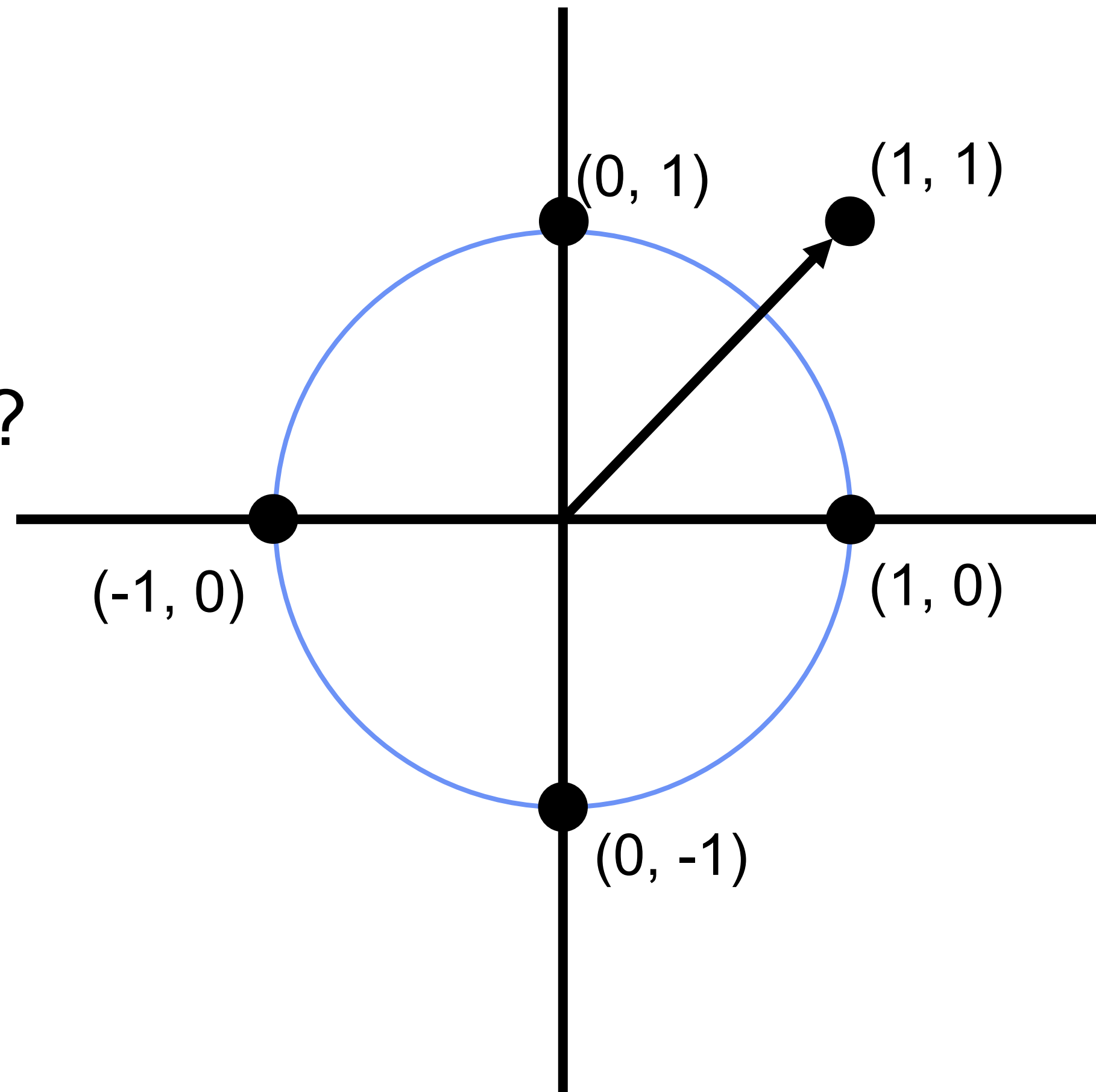


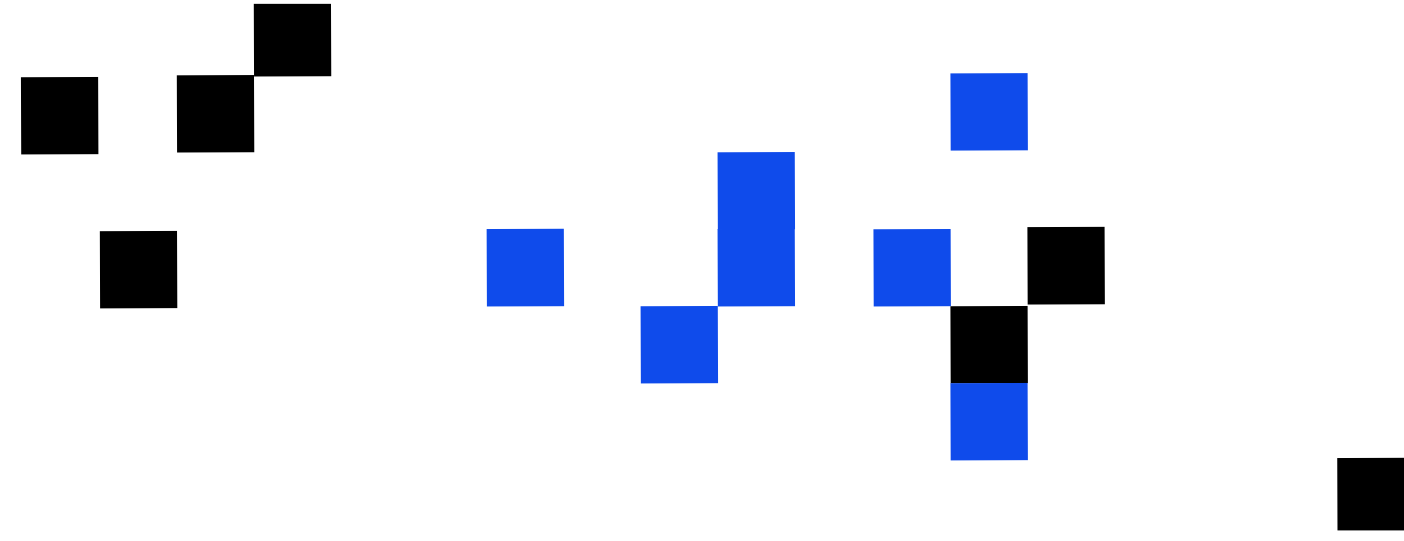
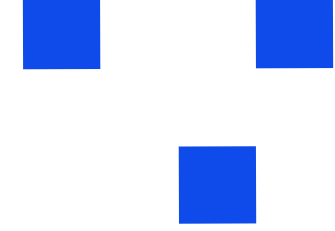
So....

- If we set maxSpeed to be 5 units per frame...
- ... and we don't use a unit vector magnitude...
- ... then it is possible to move 7 units per frame by holding down both a horizontal and vertical input key!

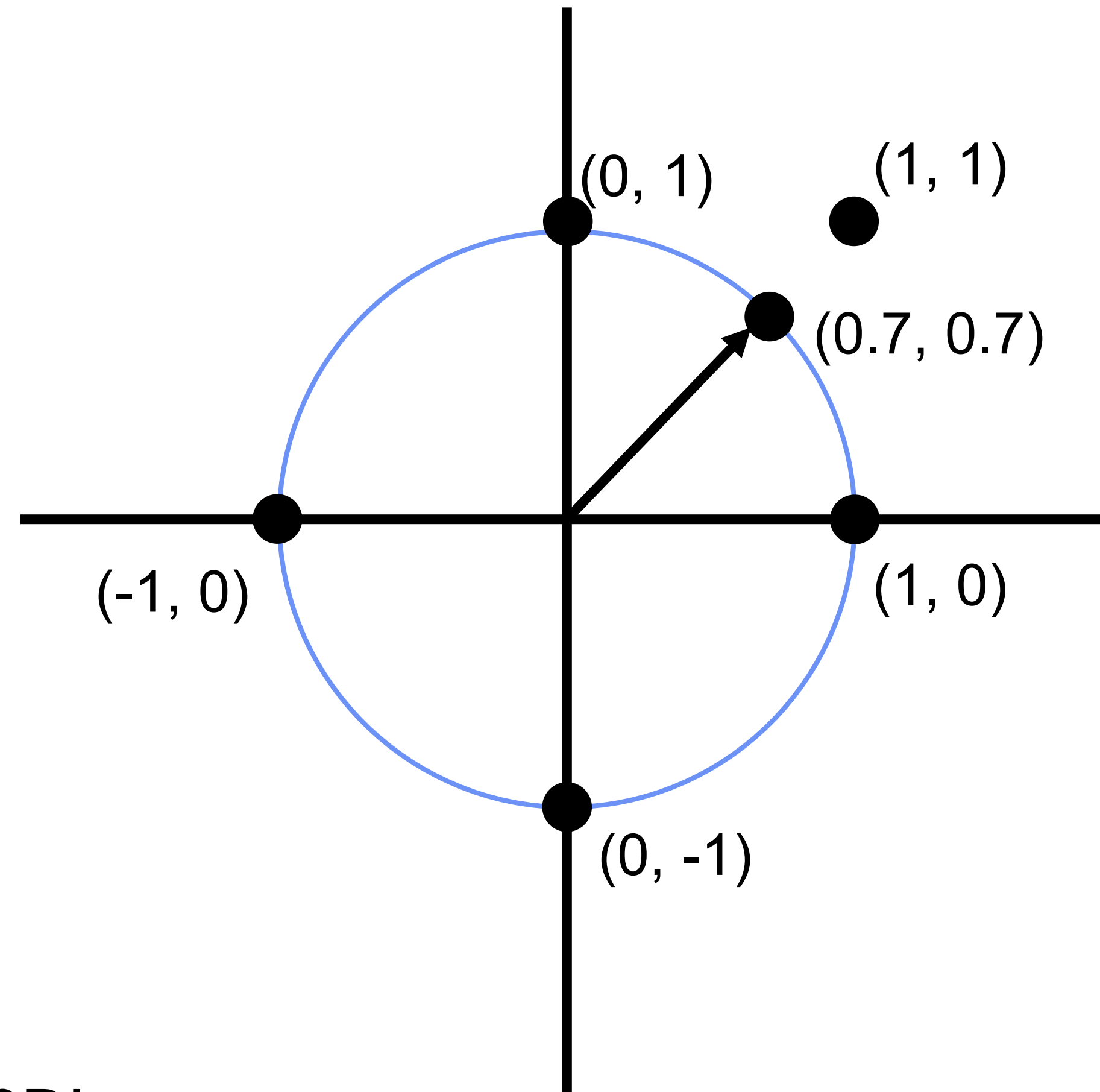
■ Vector Circle with Magnitude of 1

What is the directional vector, with magnitude 1, that points towards (1, 1)?





■ Unit Circle



Handy functions:

`Random.insideUnitCircle`

`Random.insideUnitSphere`

Random directions in 2D or 3D!



■ Moving with Input – Rotation

- Even easier!
- Combining all the values of `Input.GetAxis(...)` gives you a direction, right?



■ Moving with Input – Rotation

- Even easier!
- Combining all the values of `Input.GetAxis(...)` gives you a direction, right?
- Simply pass this direction to `Quaternion.LookRotation(...)` to rotate an object to face that direction!
- `Sensitivity and Gravity of Axis` will give smooth turning



■ 2D vs 3D Physics

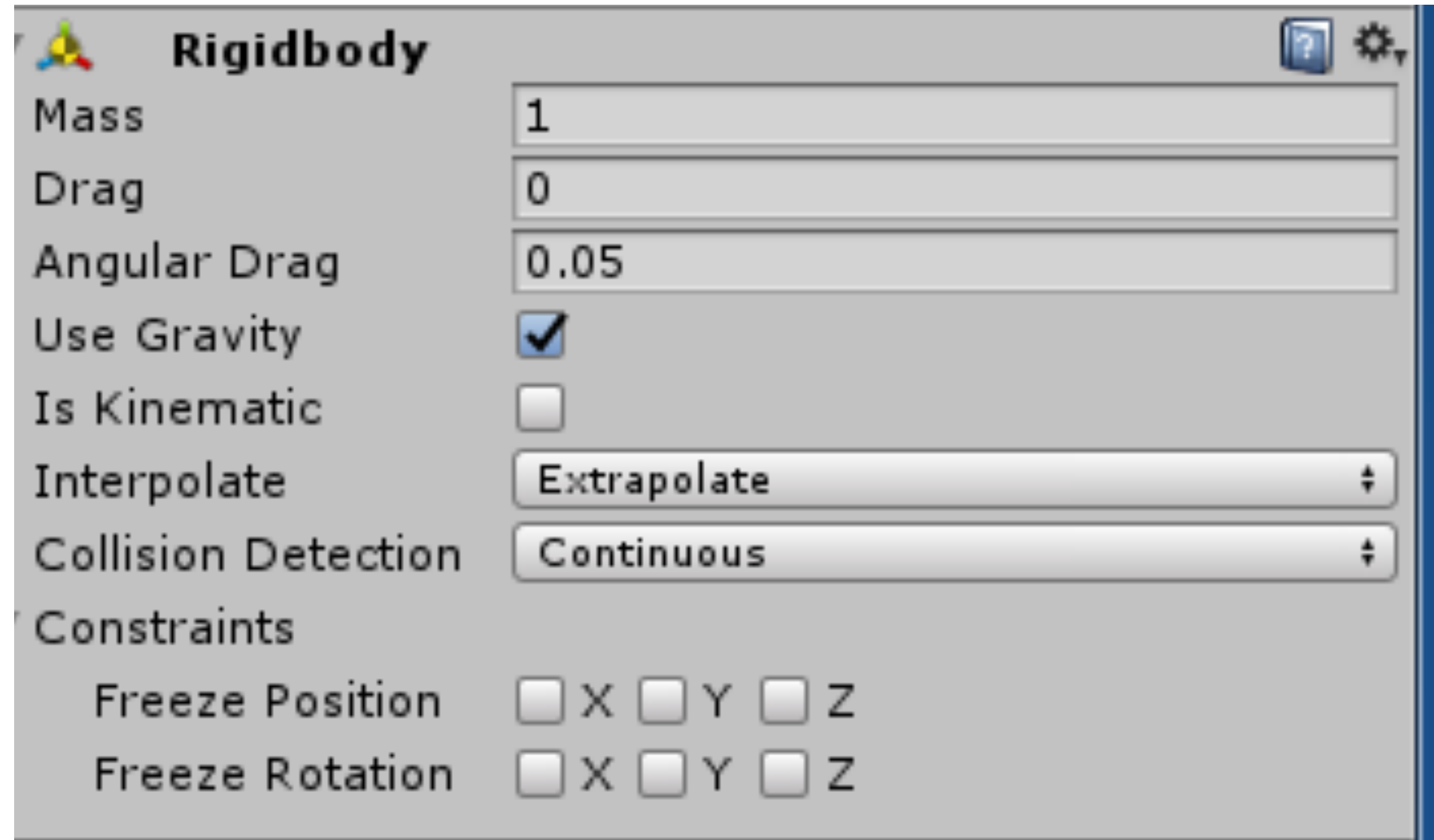
- **For every 3D Physics behaviour there is an equivalent in 2D**
- **2D Physics and 3D Physics are processed separately so they DO NOT INTERACT!**
 - **If you mix and match in a single scene, objects are not going to be colliding**
- **2D Physics is actually more versatile in Unity**
 - **Even though 3D physics is what made Unity famous when it started.**
 - **A lot more base functionality is provided to you in 2D.**



Rigidbody

- If an object is going to **receive forces (including gravity)**, then it needs a **Rigidbody** component
- E.g. When a ball hits a static wall, **the wall exerts a force on the ball and the ball exerts a force on the wall**
 - The wall is not affected because it is heavier. From a game perspective, no point in simulating those physics, so no rigidbody component
 - The ball however is pushed back the other way. This force should be simulated on the ball and so it needs a rigidbody
- Called a “**Rigid-body**” because the objects do not bend/morph/deform/etc.
 - **Soft-body** physics used to simulate cloth, skin, blobs, etc (also in Unity)
 - **Fluid dynamics** use to simulate liquid and gases (in Unity Asset Store)

Rigidbody



The image shows a Unity Inspector panel for a Rigidbody component. The panel has a title bar with the Unity logo and the word "Rigidbody". Below the title bar, there are several properties and settings:

- Mass:** A text field containing the value "1".
- Drag:** A text field containing the value "0".
- Angular Drag:** A text field containing the value "0.05".
- Use Gravity:** A checkbox that is checked.
- Is Kinematic:** A checkbox that is unchecked.
- Interpolate:** A dropdown menu with "Extrapolate" selected.
- Collision Detection:** A dropdown menu with "Continuous" selected.
- Constraints:** A section with two rows of checkboxes:
 - Freeze Position:** Three checkboxes for X, Y, and Z, all of which are unchecked.
 - Freeze Rotation:** Three checkboxes for X, Y, and Z, all of which are unchecked.



Kinematic Rigidbody

- **Static colliders** (i.e. a collider without a rigidbody) are named such because they are assumed to stay stationary in the scene.
 - This allows Unity to optimize physics calculations
 - Moving a static collider essentially causes the scene to “re-bake”, similar to dynamic lighting – computationally expensive.
- A **Kinematic Rigidbody** is essentially a “moving static collider”
 - No forces are exerted on the Kinematic rigidbody, but it does exert forces on other objects.
 - Movement is reverted to Transform.Translate(..) calls and animations.
 - E.g. The player’s character moves through the world pushing objects out of the way but not being affected themselves – the character is kinematic

Rigidbody Constraints

Constraints

Freeze Position ☐ X ☐ Y ☐ Z

Freeze Rotation ☐ X ☐ Y ☐ Z

- Sometimes its useful to have an object only partially physically simulated.
 - Constraints allow for this by freezing certain physics calculation
- An object (e.g. an elevator) that only moves up and down should not be affected by forces that will move it forward, back, left, or right.
 - Freeze Position of X and Z
- An object may want to only rotate on a few axis (e.g. a carousel) or not at all (E.g. a bullet moving in a straight line)
 - Freeze Rotation on X and Z (for carousel) or all axis (for bullet)

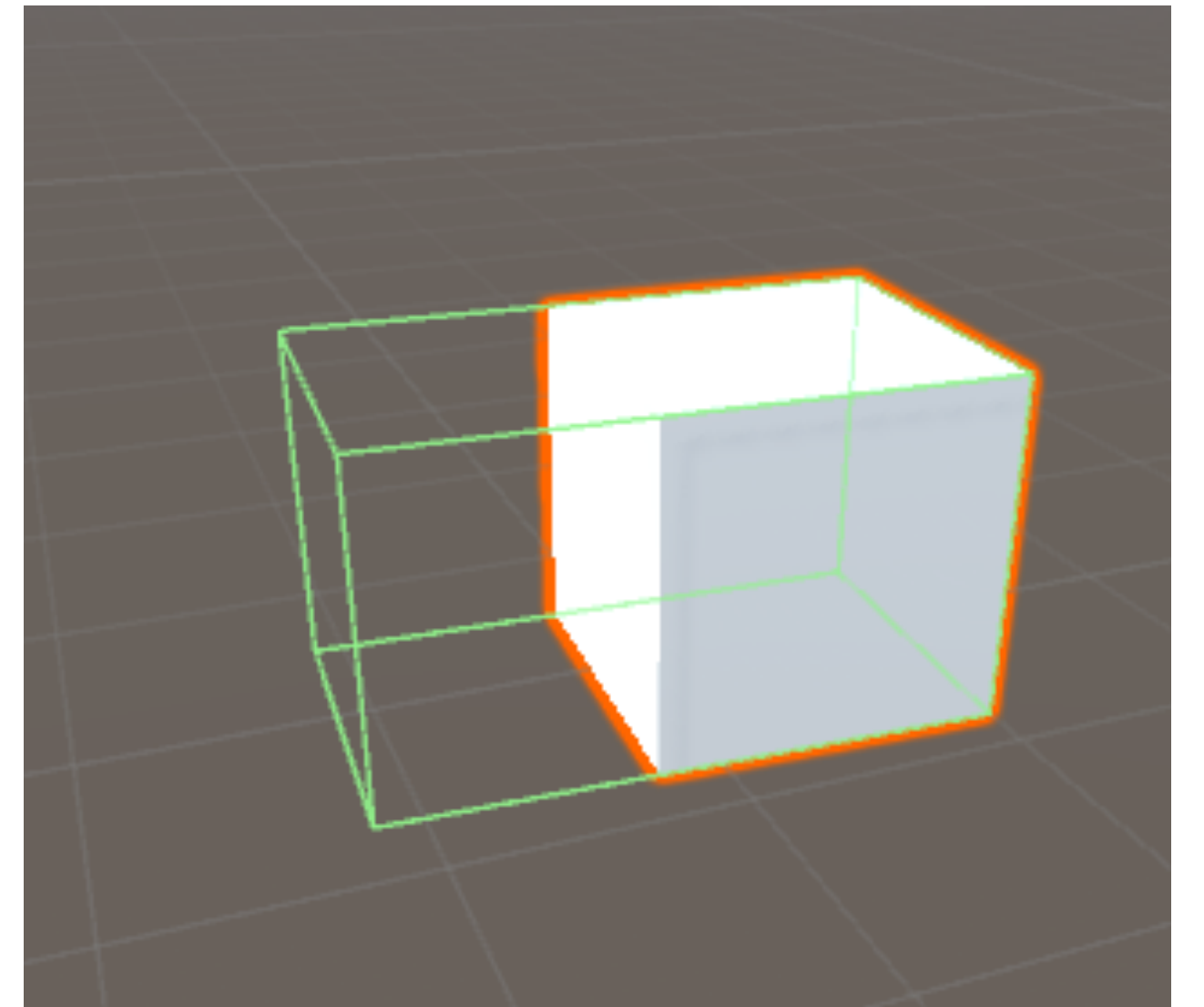


Colliders

- For any object to interact with any other object in any kind of physically simulated way it must have a **Collider**
- Colliders are simple shapes that surround an object
 - Represented as green outlines in Unity
 - Used to detect when that object is overlapping another object.
 - In Unity, both objects must have Colliders for this detection to occur
- Once a collision has occurred, many different things can happen:
 - Simulate the resulting forces of the collision (say hello to Newton's Laws!)
 - Register a bullet hitting a player, deduct health, destroy the bullet
 - Open a door when a player steps on a button
 - etc.

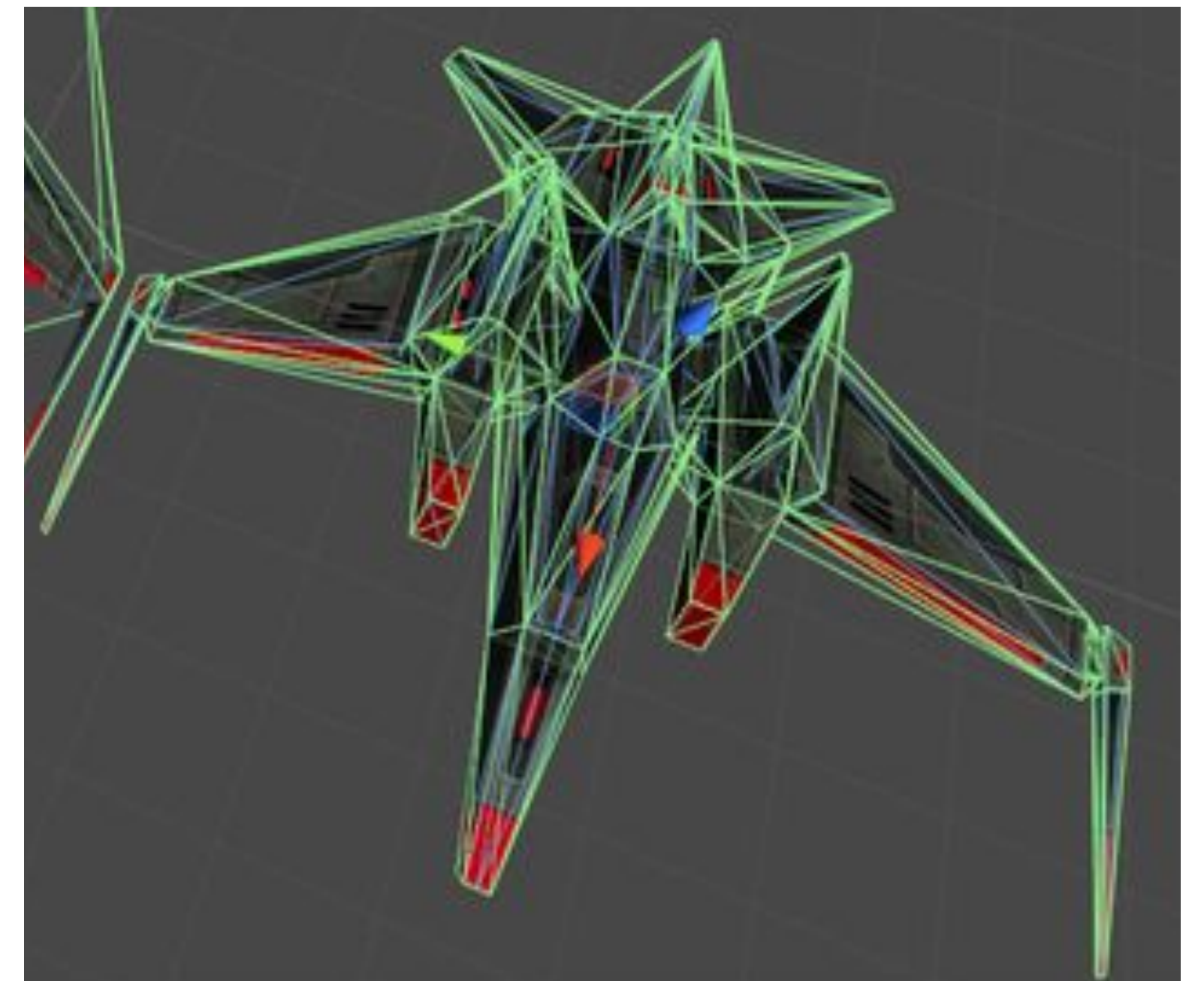
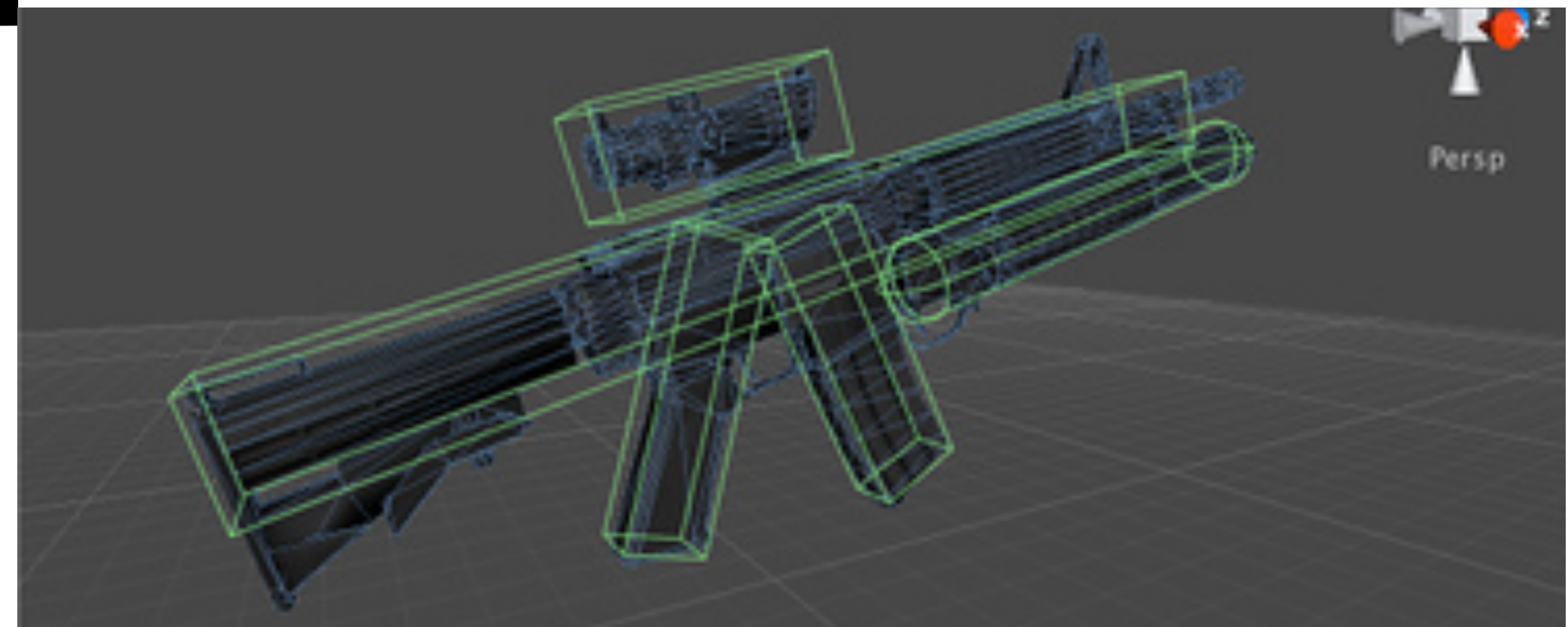
Primitive Collider

- Box, Sphere, and Capsule Colliders
 - Most common
 - Perfectly fit their counterpart GameObjects
- Scale the GameObject and the Collider scales as well
 - The collider is essentially a child of the gameobject (it just isn't shown in the hierarchy window)
 - Dimensions of the collider will match the primitive.



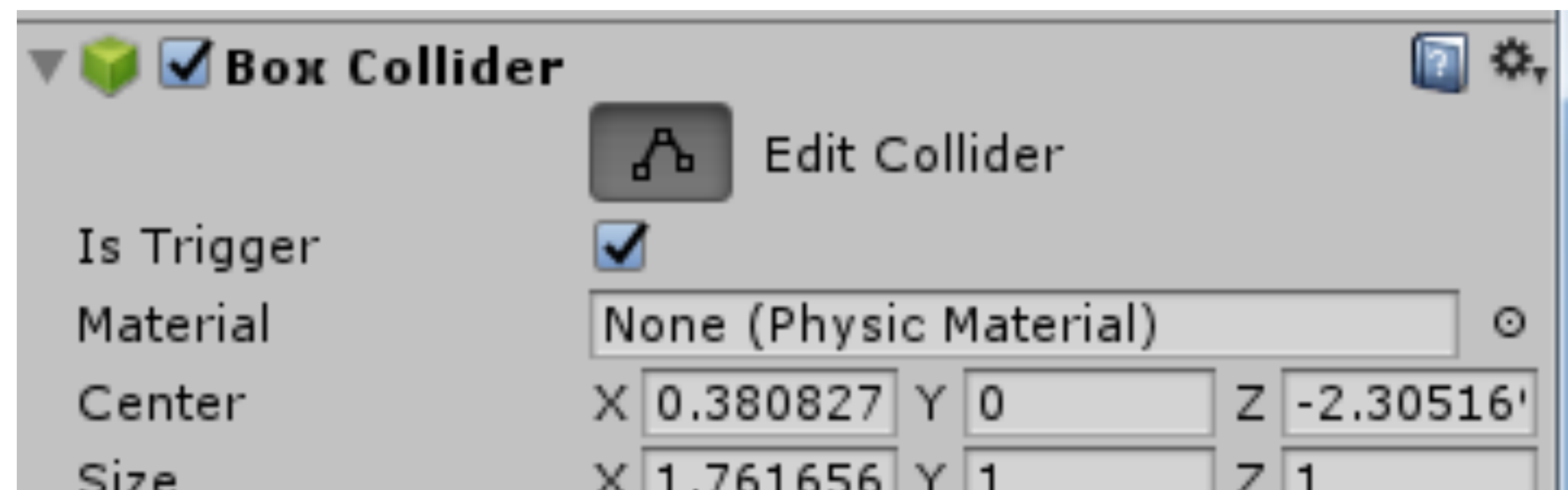
Complex Colliders

- Mesh, Terrain, Cloth, Wheel Collider
 - Use to perfectly fit the shape of a gameobject where precision collision detection is required
 - Offers additional physics controls to properly simulate interactions after the collision.
- **These can be hugely computationally expensive**
 - Many primitive colliders are always faster than one complex collider.
 - Use complex colliders only when really needed.



Triggers

- A Trigger is a simplified type of collider that is only used to call functions in Code.
- If a collider is marked as a Trigger, it will not simulate collision physics when it interacts with another collider.





■ Collision Action Matrix – For Colliders

- **The #1 reason for “MY PHYSICS ISN’T WORKING!!!” complaints**
- Colliders are mainly used for objects that want physics simulated on them
 - Only detect when a physically simulated object (a rigidbody) hits another collider and is going to be affected by the collision.
 - Static colliders are those without a rigidbody (e.g. walls – two stationary walls never need to collide with each other)

Collision detection occurs and messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						



■ Collision Action Matrix – For Triggers

- A static collider does not interact with other static colliders/triggers.
 - Do you want a door to tell you it has hit its own door frame?
- Otherwise, they interact with nearly every kind of collider
 - E.g. Animated character don't want physical forces simulated on them, but they should still register triggers for interacting with other game elements

Trigger messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y



Colliders in Code

- **OnCollisionEnter(Collision col)**
- **OnCollisionExit(...)**
- **OnCollision(...)** – Called every frame that two colliders are overlapping
- All are called once per **Physics cycle** for each one-to-one collision between objects.
- **Collision** class
 - Gives access to a lot of useful information
 - The other object involved in the collision, the point(s) of contact, the angle of contact, the normal of the contact, the force of the collision, etc.

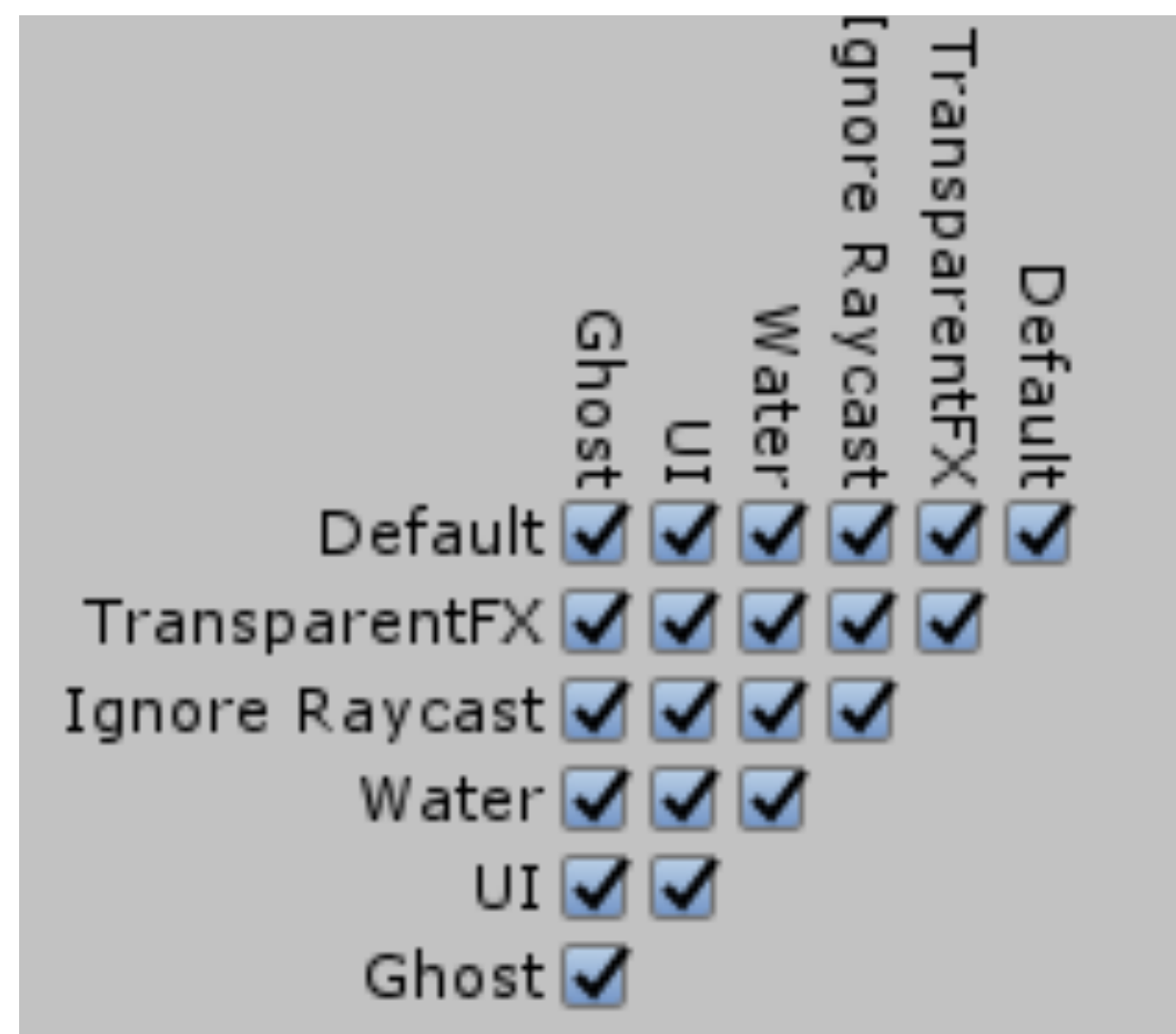


Triggers in Code

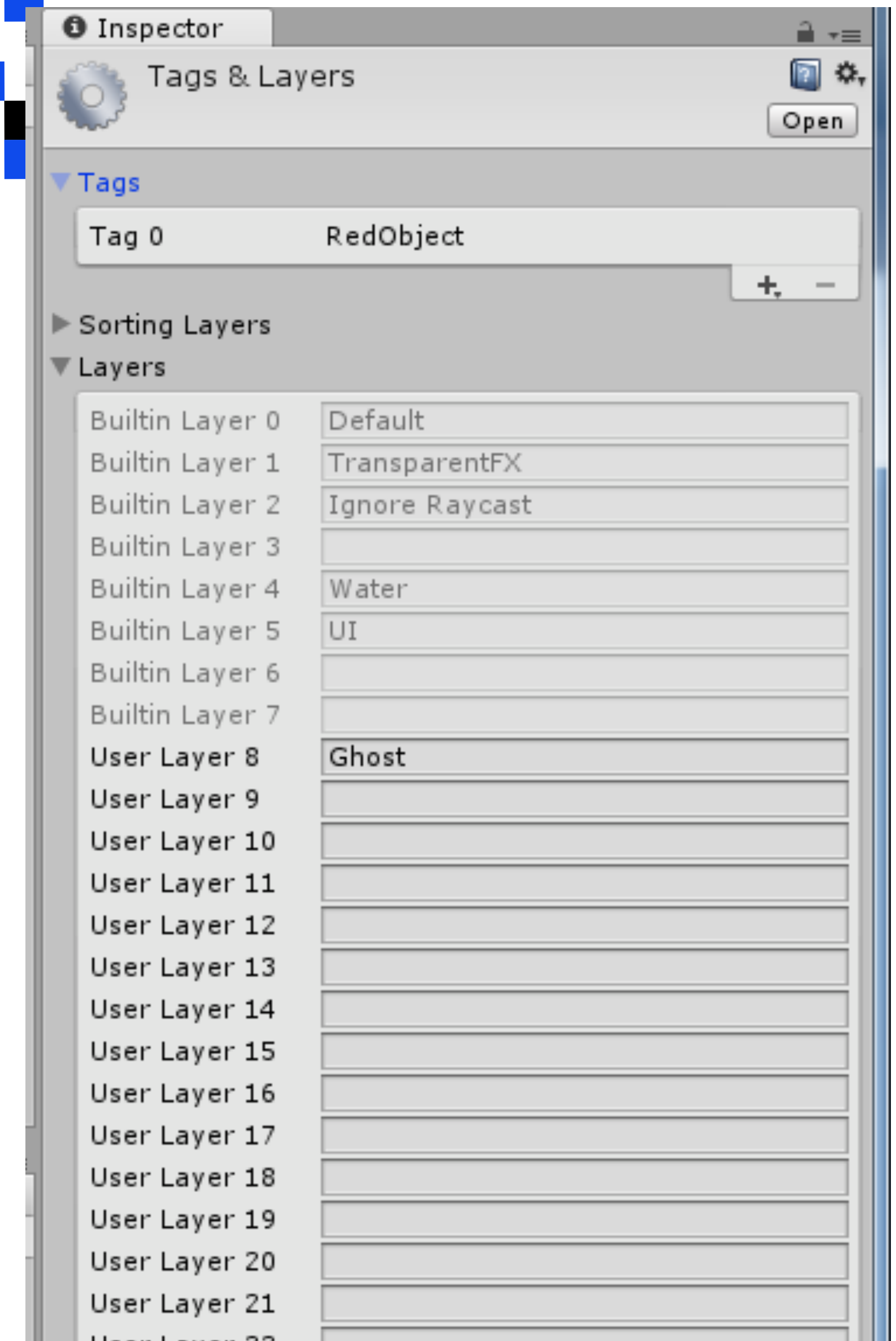
- **OnTriggerEnter(Collider other)**
- **OnTriggerExit(...)**
- **OnTrigger(...)** – Called every frame that two colliders are
- Note “**Collider** other” vs “**Collision** col”
- Simplified compared to normal Colliders because the main question is “Are the colliders overlapping?”
 - I.e. no need for details of the collision.
 - “Collider other” typically used just to test a tag of the other object or something similar
 - E.g. Did a player from Red Team touch the button? Yes, then capture the flag

Layer-based Collision Detection

- We have mostly been using Tags up until now but **Layers** offer a lot of functionality for physics.
- The Layer Collision Matrix – Any object with one layer ID will interact with objects with another layer ID if there is a tick in the corresponding box.
- Great for **reducing physics computation** (i.e. if it doesn't need to interact, it shouldn't) and **controlling complex interactions** (e.g. block enemy passage but not player passage)

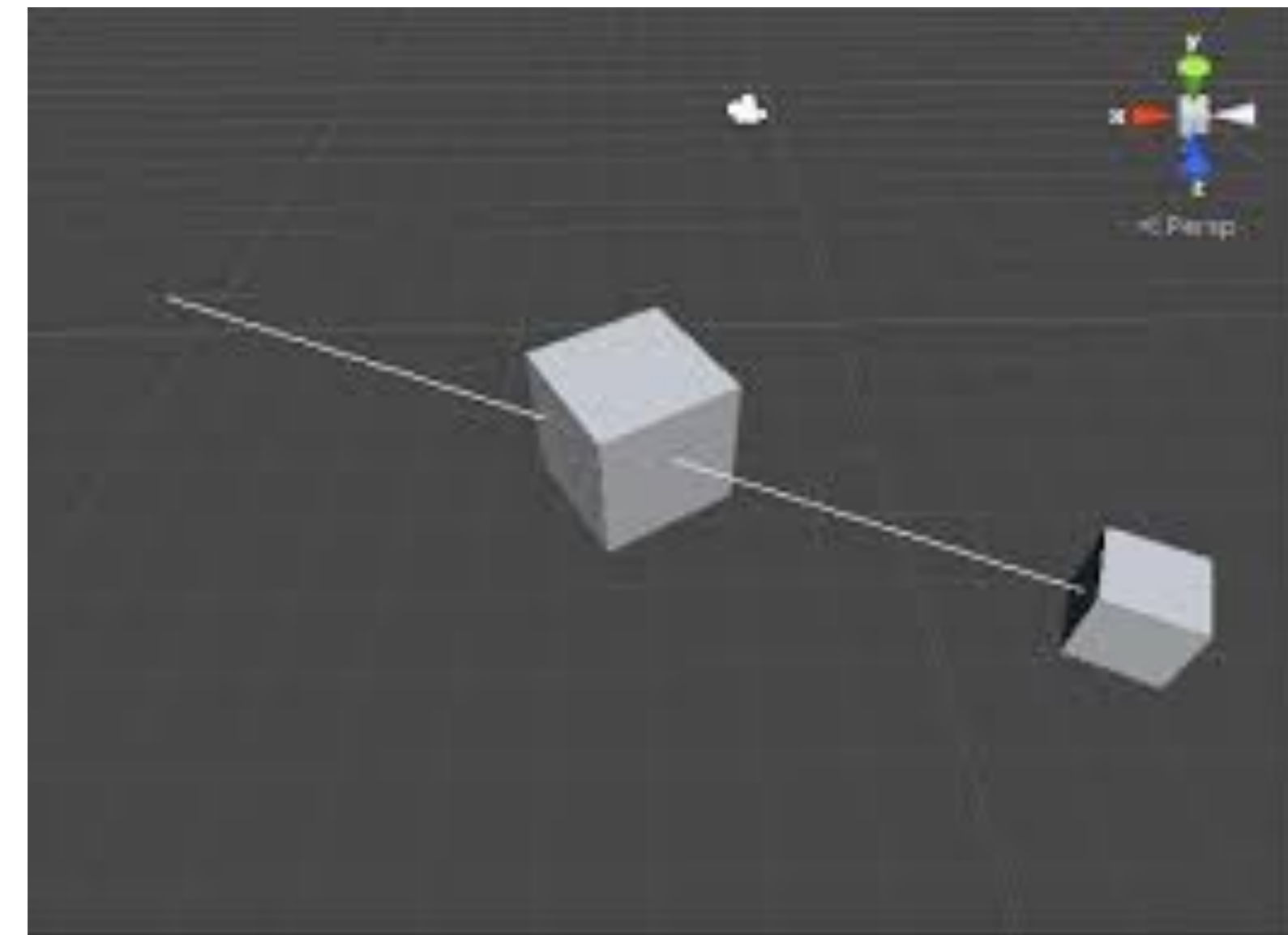


	Default	TransparentFX	Ignore Raycast	Water	UI	Ghost
Default	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
TransparentFX	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Ignore Raycast	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Water	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
UI	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Ghost	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>



Raycast

- **Raycast** – an invisible **ray** (line) that is **cast** (drawn) from one point to another in 3D space.
 - Used to detect what collides with the ray
 - Use to be expensive in other engines, but Unity makes them VERY efficient!
- Extremely useful for:
 - Detecting **line of sight** of enemies
 - Raycast from enemy to player and see if anything is hit before the player
 - Detecting **what the player is clicking on** with the mouse cursor
 - Raycast from the camera's screen space to world space
 - **Hitscan bullets**
 - Instant bullet shots. Raycast and instantly do damage to whatever the ray hits first.





Physics.Raycast

*public static bool **Raycast**(Vector3 origin, Vector3 direction, out RaycastHit hitInfo, float **maxDistance**, int **layerMask** = DefaultRaycastLayers)*

- **Origin** – Starting point of the ray
- **Direction** – The direction the Ray will travel in
- **maxDistance** – The distance the ray will travel
- **layerMask** – A series of ints that specify which Physics layers to hit with this ray

*public static bool **Linecast**(Vector3 start, Vector3 end, out RaycastHit hitInfo, int **layerMask** = DefaultRaycastLayers)*

- **Start** and **End** points – Instead of **origin** and **direction**



RaycastHit

- "out RaycastHit hitInfo" – hitInfo is passed in as a reference so that the method will store the result in the hitInfo parameter and we will have access to it.
- RaycastHit is like Collision in OnCollisionEnter(Collision col)
 - Provides information on what was hit, how far it is, at what angle was it hit, etc.

```
RaycastHit hitInfo;  
bool hit = Physics.Raycast(Vector3.zero, Vector3.forward, out hitInfo, 5.0f);  
if (hit)  
    if (hitInfo.transform.gameObject.name == "Enemy")  
        HurtEnemy();
```