Contents

CS 470 Game Development — Week 2, Lecture 5

Micro-Game 2: Ricochet

Goal: Build a logo that bounces off screen edges. Click to spawn more! Along the way, learn about speed & direction, autonomous movement, and scene instantiation.

---

1. Quick Recap: Wanderer Foundations

Last lecture, we built Wanderer — a character the player moves with arrow keys. Here's what we used:

Concept	What You Learned
Nodes & Scenes	Game objects arranged in a tree, saved as `.tscn` files
`_process(delta)`	Runs every frame. `delta` = time since last frame.
Input Polling	`Input.is_action_pressed()` — non-blocking, checked every frame
Clamping	`clamp()` keeps objects inside the screen

What's Different Today?

	Wanderer	Ricochet
Who controls it?	**You** (keyboard)	**Itself** (speed + direction)
At boundaries?	Stays on screen (clamp)	Bounces off edges
Direction from?	Player input	`_ready()` — set once at start

The key shift: from **player-driven** movement to **autonomous** movement. This is exactly how the Pong **ball** will work.

---

2. Vector2 Deep Dive

What is Vector2?

A `Vector2` in Godot is simply a pair of numbers `(x, y)` bundled together. The same type can represent very different things depending on context:

```
var pos = Vector2(400, 300)    # a position -- WHERE something is
var dir = Vector2(1, 0)        # a direction -- WHICH WAY to go (right)
var spd = Vector2(200, 150)    # speed in each axis
```

The type is the same (`Vector2`), but the **meaning** comes from how you use it.

Vector2 Operations

Operation	Example	Used For
Addition	`(2, 3) + (1, -1) = (3, 2)`	Combining movements
Subtraction	`(5, 3) - (2, 1) = (3, 2)`	Direction between points
Scalar multiply	`(1, 0) * 200 = (200, 0)`	Scaling (speed)
Negation	`-(3, 4) = (-3, -4)`	Reverse direction

Every one of these operations appears in Ricochet: - **Addition** — `position += direction * SPEED * delta` moves the logo - **Negation** — `direction.x = -direction.x` reverses direction on bounce - **Scalar multiply** — `direction * SPEED` scales the direction by speed

Length and Normalization

```
Length:     |(3, 4)| = 5
Normalized: (3, 4).normalized() = (0.6, 0.8)
```

Normalized means: same direction, but **length 1**. Why is this useful? Because once you have a unit-length direction, you can multiply by any speed you want at movement time:

```
var direction = Vector2(3, 4).normalized()   # (0.6, 0.8), length 1
# At movement time: direction * SPEED * delta
```

No matter what random direction you start with, multiplying by `SPEED` always gives the same movement rate.

Random Direction Generation

This is a 2-step pattern we'll use in Ricochet's `_ready()`:

```
# Step 1: Random x and y between -1 and 1
var random_vec = Vector2(randf_range(-1, 1), randf_range(-1, 1))
# Could be anything: (0.3, -0.7), (-0.9, 0.1), (0.5, 0.5), etc.

# Step 2: Normalize to length 1
var direction = random_vec.normalized()
# Now it's a unit vector -- always length 1
# Speed (SPEED) is a separate constant, applied during movement
```

In the actual code, we compress this into one line:

```
direction = Vector2(randf_range(-1, 1), randf_range(-1, 1)).normalized()
```

---

3. Speed, Direction & Autonomous Movement

Speed + Direction (Separate Variables)

We keep speed and direction as **separate variables**: - **SPEED** — how fast (a number, e.g., 300 pixels/sec). Uppercase = constant convention. - **direction** — which way (a normalized Vector2, e.g., (0.6, 0.8), length 1)

```
var SPEED = 300                    # pixels per second
var direction = Vector2(0, 0)      # set in _ready()
```

They combine at movement time: `direction * SPEED * delta`.

Autonomous Movement

In Wanderer, the direction came from the keyboard every frame. In Ricochet, direction is stored in a variable and the object moves itself:

```
# Every frame: move by direction * SPEED * delta
position += direction * SPEED * delta
```

There is no `Input.is_action_pressed()`. The object moves **itself**. This is autonomous movement — the foundation of enemies, projectiles, and physics objects in games.

Notice that `direction * SPEED * delta` is frame-rate independent, just like Wanderer's movement.

The Bounce

When the logo hits a wall, we reverse the appropriate direction component:

- **Hitting left or right wall:** flip horizontal → `direction.x = -direction.x`
- **Hitting top or bottom wall:** flip vertical → `direction.y = -direction.y`

```
Before hitting right wall:  direction = (0.8, 0.6)
After bounce:               direction = (-0.8, 0.6)
                                         ^^^^
                                      only x flips!
```

That's all bounce physics is: negate the direction component perpendicular to the wall. Speed stays the same — only direction changes. This same principle applies in Pong, Breakout, and most 2D physics games.

Boundary Checking

How do we know the logo hit a wall? We compare its position to the screen edges, accounting for sprite size:

```
var screen_size = get_viewport_rect().size
var sprite_width = 128
var sprite_height = 128

if position.x > screen_size.x - sprite_width/2 or position.x < sprite_width/2:
    direction.x = -direction.x    # hit left or right wall

if position.y > screen_size.y - sprite_height/2 or position.y < sprite_height/2:
    direction.y = -direction.y    # hit top or bottom wall
```

We use `get_viewport_rect().size` to get the screen size dynamically, and account for the sprite's half-width/height so the logo bounces at its edges, not its center.

> **Compare to Wanderer:** Wanderer used `clamp()` to **stop** at the boundary. Ricochet uses `if`/negation to **bounce** at the boundary. Both are valid boundary strategies.

---

4. `_ready()` vs `_process()`

Two Lifecycle Functions

Function	When	How Often	Ricochet Use
`_ready()`	Node enters the scene	**Once**	Set random starting direction
`_process(delta)`	Every frame	**~60 times/sec**	Move and bounce

Think of it this way: - `_ready()` is like Python's `__init__` — runs once at creation - `_process(delta)` is like the body of an infinite loop — runs forever

Both are called automatically by the engine. You just define the function and Godot handles the rest.

Why `_ready()` for Direction?

```
var SPEED = 300                    # pixels per second
var direction = Vector2(0, 0)      # overwritten in _ready()

func _ready():
```

```
    # Set random direction (normalized = length 1)
    direction = Vector2(randf_range(-1, 1), randf_range(-1, 1)).normalized()
```

Why not just set the random direction in the `var` declaration line? Because:

1. **Each logo needs a different random direction.** If we set it at declaration, `randf_range` would run once when the script loads — and ALL logos from that scene would get the same direction.
2. **`_ready()` runs once per instance.** When we spawn 10 logos, each one calls `_ready()` independently and gets its own unique random direction. `SPEED` stays the same for all of them.

This distinction — declaration (once per script load) vs. `_ready()` (once per instance) — matters whenever you create multiple instances of the same scene.

---

5. Building Ricochet — Step by Step

Setup

1. Create new project: `Ricochet`
2. Set window size: 800×600 in Project Settings → Display → Window
3. We'll use the Godot icon (`res://icon.svg`) as our bouncing logo

Step 1: Create the Player Scene

1. Create a **Node2D** root, rename to `Player`
2. Add a child **Sprite2D** under `Player`
3. Drag `icon.svg` into the Sprite2D's **Texture** property in Inspector
4. Save scene as `prefabs/player.tscn`

```
Player (Node2D)     <- script goes here, controls movement
  +-- Sprite2D      <- just displays image, follows parent automatically
```

This is simpler than Wanderer's scene tree. There's no separate container node — the `Player` node itself is the moving object. We save it in a `prefabs/` folder to keep things organized.

Step 2: Add the Player Script

1. Select `Player` (not Sprite2D!)
2. Click Attach Script → Create
3. Replace everything with:

```
extends Node2D

var SPEED = 300                    # pixels per second
var direction = Vector2(0, 0)      # random direction to begin with

func _ready():
    direction = Vector2(randf_range(-1, 1), randf_range(-1, 1)).normalized()

func _process(delta):
    var screen_size = get_viewport_rect().size
    var sprite_width = 128
    var sprite_height = 128

    position += direction * SPEED * delta

    # Bounce off walls
```

```
    var bounced = false
    if position.x > screen_size.x - sprite_width/2 or position.x < sprite_width/2:
        direction.x = -direction.x
        bounced = true
    if position.y > screen_size.y - sprite_height/2 or position.y < sprite_height/2:
        bounced = true

    if bounced:
        $Sprite2D.modulate = Color(randf(), randf(), randf())
        direction.y = -direction.y
```

Line-by-Line Walkthrough

`var SPEED = 300` — Declares the movement speed in pixels per second. Uppercase = constant convention.

`var direction = Vector2(0, 0)` — Declares direction as a Vector2. This default gets overwritten by `_ready()`.

`func _ready():` — Runs once when the Player enters the scene. Creates a random unit vector (`normalized()` makes length 1). Each instance gets a different random direction.

`position += direction * SPEED * delta` — Every frame, move the Player. `direction * SPEED * delta` ensures frame-rate independence. No keyboard input — the Player moves autonomously.

`get_viewport_rect().size` — Gets the screen size dynamically, so the code works at any resolution.

`if position.x > screen_size.x - sprite_width/2 or position.x < sprite_width/2:` — Checks if the Player has passed the right or left boundary, accounting for the sprite's half-width. If so, reverse the horizontal direction (`direction.x = -direction.x`).

`if bounced:` — If any bounce occurred, change the sprite's color to a random RGB value and flip the vertical direction.

Step 3: Create the Game Scene

1. Create a new scene: **Node2D** root → `Game`
2. Drag `prefabs/player.tscn` from the FileSystem panel into the scene
3. Position the Player at `(400, 300)` (center of the 800×600 window)
4. Save as `game.tscn`

```
Game (Node2D)                    <- will get spawning script later
  +-- Player (player.tscn)     <- instance, starts at (400, 300)
```

Step 4: Run It

1. Press **F5**, select `game.tscn` as main scene
2. The logo bounces off all four walls endlessly!

You just built ball physics. This is *exactly* how the Pong ball will move.

---

6. Mouse Input and Spawning

Two Ways to Handle Input

In Wanderer, we used **polling** — checking key states every frame inside `_process()`:

```
func _process(delta):
    if Input.is_action_pressed("ui_right"):
        # this key is held down RIGHT NOW
```

For Ricochet's click-to-spawn, we use **events** — Godot calls `_input()` only when something actually happens:

```
func _input(event):
    if event is InputEventMouseButton:
        # a mouse button was pressed or released
```

Approach	Where	Best For
Polling	`_process()`	Continuous actions (movement, held keys)
Events	`_input(event)`	One-shot actions (clicks, key taps)

Use polling for held keys, events for single clicks.

`_input()` and `InputEventMouseButton`

```
func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and event.pressed:
            # Left mouse button was just pressed!
            print("Clicked at: ", event.position)
```

Three nested checks: 1. **`event is InputEventMouseButton`** — Is this a mouse event? (not keyboard, not joystick) 2. **`event.button_index == MOUSE_BUTTON_LEFT`** — Was it the left button? (not right, not middle) 3. **`event.pressed`** — Was it a press? (not a release)

`event.position` gives the click coordinates as a `Vector2`.

`preload()` and `instantiate()`

To spawn copies of a scene at runtime, you need two operations:

```
# Load the scene file ONCE, at script load time
var logo_scene = preload("res://prefabs/player.tscn")
```

`preload()` is like Python's `import` — it loads the scene file once and keeps a reference. The path `"res://"` means "from the project root."

```
# Create a NEW node tree from the scene
var new_logo = logo_scene.instantiate()
```

`instantiate()` is like creating a new object from a class. Each call produces a fresh, independent copy with its own properties.

```
# Place it and add to the game
new_logo.position = event.position    # where the mouse clicked
add_child(new_logo)                   # add to the scene tree -- it starts living!
```

`add_child()` puts the new node into the scene tree as a child of the current node. The moment it enters the tree, the engine starts calling its `_ready()` and `_process()`.

Important: Each instantiated logo calls its OWN `_ready()`, getting its own random direction. That's why all the logos bounce in different directions.

The Full Game Script

Attach this script to the `Game` node:

```
extends Node2D

var logo_scene = preload("res://prefabs/player.tscn")

func _ready():
    pass

func _process(delta):
    pass

func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and event.pressed:
            var new_logo = logo_scene.instantiate()
            print(event.position)
            new_logo.position = event.position
            add_child(new_logo)
```

Run it again — click anywhere to spawn new logos. Each one bounces with its own random direction. Click 20 times and watch the beautiful chaos!

---

7. Bonus: Color Modulation on Bounce

The `$` Shorthand

GDScript provides a shorthand for accessing child nodes:

```
$Sprite2D                    # same as get_node("Sprite2D")
$Sprite2D.modulate           # the tint color of the sprite
```

The `$` prefix followed by a node name is equivalent to calling `get_node("NodeName")`. It only works for **direct children** (or paths like `$Child/GrandChild`).

Adding Color Change

In the player script, we use a `bounced` flag to consolidate color changes. When any bounce occurs, the sprite gets a random color:

```
var bounced = false
if position.x > screen_size.x - sprite_width/2 or position.x < sprite_width/2:
    direction.x = -direction.x
    bounced = true
if position.y > screen_size.y - sprite_height/2 or position.y < sprite_height/2:
    bounced = true

if bounced:
    $Sprite2D.modulate = Color(randf(), randf(), randf())
    direction.y = -direction.y
```

- **`$Sprite2D`** — access the child Sprite2D node
- **`.modulate`** — a `Color` property that tints the sprite
- **`Color(randf(), randf(), randf())`** — random red, green, blue values (0.0 to 1.0 each)

Now every wall hit produces a new random color — visual feedback that makes the bouncing more satisfying.

---

8. Complete Final Scripts

Player Script (`player.gd`)

```
extends Node2D

var SPEED = 300                    # pixels per second
var direction = Vector2(0, 0)      # random direction to begin with

func _ready():
    direction = Vector2(randf_range(-1, 1), randf_range(-1, 1)).normalized()

func _process(delta):
    var screen_size = get_viewport_rect().size
    var sprite_width = 128
    var sprite_height = 128

    position += direction * SPEED * delta

    # Bounce off walls
    var bounced = false
    if position.x > screen_size.x - sprite_width/2 or position.x < sprite_width/2:
        direction.x = -direction.x
        bounced = true
    if position.y > screen_size.y - sprite_height/2 or position.y < sprite_height/2:
        bounced = true

    if bounced:
        $Sprite2D.modulate = Color(randf(), randf(), randf())
        direction.y = -direction.y
```

Game Script (`game.gd`)

```
extends Node2D

var logo_scene = preload("res://prefabs/player.tscn")

func _ready():
    pass

func _process(delta):
    pass

func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and event.pressed:
            var new_logo = logo_scene.instantiate()
            print(event.position)
            new_logo.position = event.position
            add_child(new_logo)
```

---

9. Summary of Concepts

Concept	What You Learned
Vector2	Two numbers `(x, y)` stored together. Can represent position, direction, or movement.
SPEED + direction	Separate speed (number) and direction (normalized Vector2).
Autonomous movement	Objects move themselves — no keyboard input needed. `position += direction * SPEED * delta`.
Boundary checking	Compare position to screen edges to detect wall hits.
Bouncing	Reverse the direction component perpendicular to the wall: `direction.x = -direction.x`.
`_ready()`	Runs once when a node enters the scene. Used for initialization.
`_input(event)`	Called on input events (mouse clicks, key taps). Unlike polling, it responds to discrete events.
`preload()`	Load a scene file once at script load time. Returns a reference you can instantiate from.
`instantiate()`	Create a new independent copy of a preloaded scene.
`add_child()`	Add a node to the scene tree at runtime. The node "starts living" — `_ready()` and `_process()` begin.
`$NodeName`	Shorthand for `get_node("NodeName")`. Access child nodes easily.

---

10. Connection to Pong

This lecture laid the foundation for the Pong ball:

Ricochet Concept	Pong Ball Equivalent
Random direction in `_ready()`	Ball starts with random direction after each point
`position += direction * SPEED * delta`	Ball moves autonomously
Boundary bounce (top/bottom)	Ball bounces off top and bottom walls
Boundary check (left/right)	Ball passing left/right edge = a point scored

The only thing Pong adds is **collision with paddles** — and for that, we need `Area2D` (next lecture).

Exercises

1. **Speed Experiment:** Change `SPEED` from 300 to 500, then to 100. What speed feels right for a bouncing logo? What about a Pong ball?
2. **Gravity Pull:** Add a small constant to `direction.y` each frame inside `_process`: `direction.y += 0.5 * delta`. What happens to the bounce pattern? Try different values. *This is how real gravity works in games — a constant downward acceleration.*
3. **Spawn Limit:** Only allow a maximum of 10 logos on screen at once. Before spawning, check how many children the Game node has. *Hint: use `get_child_count()` in the `_input` function and only spawn if the count is below your limit.*
4. **Speed Increase:** Make the logo 10% faster on each bounce. Add `SPEED *= 1.1` inside the bounce block. How long before it becomes uncontrollable? *Hint: you might want to cap the speed with something like `SPEED = min(SPEED, 800)`.*
5. **Trail Effect:** Every 0.1 seconds, spawn a fading copy of the logo at its current position. The copy should not move — just fade out and disappear. *Hint: accumulate delta in a timer variable. When*

it reaches 0.1, spawn a copy, set its `modulate.a` *(alpha) to 0.5, and use a Tween or manual fade in* `_process` *to reduce alpha to 0 over time. Call* `queue_free()` *when alpha reaches 0.*

6. **Screen Wrap:** Instead of bouncing, make the logo wrap around to the opposite edge. When it goes past the right side, it reappears on the left. *Hint: use modulo or an if-else to reset position.*

7. **(Advanced) Logo Collision:** Make logos bounce off *each other*, not just walls. This requires detecting when two logos overlap — which needs `Area2D` and `CollisionShape2D` (next lecture's topics). Even if you can't implement it yet, plan the approach: What nodes would you add to the Player scene? What signal would you connect? How would you calculate the new directions?