Contents

CS 470 Game Development — Week 3, Lecture 7

Micro-Game 4: Pong

> **Goal:** Build a complete 2-player Pong game by combining everything from Wanderer, Ricochet, and Fruit Frenzy — then add `@export`, editor-based groups, custom signals, and sound.

---

1. Quick Recap: The Journey So Far

Over the last three lectures, you built three micro-games. Each one introduced new Godot concepts:

Micro-Game	Concepts Introduced
Wanderer	Nodes, scenes, scripts, `_process(delta)`, input polling, `clamp()`
Ricochet	Vector2, velocity, autonomous movement, bouncing, `_ready()`, `_input()`, `preload()`, `instantiate()`
Fruit Frenzy	Area2D, CollisionShape2D, signals (`area_entered`, `timeout`), groups (`add_to_group`, `is_in_group`), `queue_free()`, Timer, `.connect()`, `get_parent()`

How Pong Uses Everything

Pong Component	Concepts From	Origin
Paddle movement	Input polling + `clamp()`	Wanderer
Ball movement	`position += velocity * delta`	Ricochet
Wall bounce	`velocity.y = -velocity.y`	Ricochet
Collision detection	Area2D + CollisionShape2D	Fruit Frenzy
Signal handling	`area_entered` + `.connect()`	Fruit Frenzy

Pong is the culmination — it reuses everything you've learned and adds new concepts: `@export`, editor-based group assignment, custom signals, and sound.

---

2. The Starter Project

Since Pong is larger than our previous games, we provide a **starter project** with the familiar concepts pre-implemented. This lets you focus on the new material during the lecture.

What's Pre-Built

The starter includes three GDScript files with working code for previously-taught concepts and `TODO` comments marking where you'll add new code:

- **`paddle.gd`** — Movement with input polling and clamp (Wanderer pattern)

- `ball.gd` — Autonomous movement with wall bounce and reset (Ricochet pattern)
- `game.gd` — Score variables and empty handler functions

Opening the Starter

1. Copy the `Pong/` folder to your workspace
2. Open Godot → **Import** → navigate to the `Pong/` folder → select `project.godot`
3. Click **Import & Edit**

Creating the Scenes

The starter provides scripts but not scenes. Create them in the editor:

Paddle Scene (`paddle.tscn`)

1. **Scene → New Scene** → root node: **Area2D** → rename to `Paddle`
2. Add child: **Sprite2D** (or **ColorRect** set to 20×100px, white)
3. Add child: **CollisionShape2D** → Inspector: Shape → New **RectangleShape2D** → size 20×100
4. Attach `paddle.gd` to the Paddle root node
5. Save as `paddle.tscn`

```
Paddle (Area2D)
  +-- Sprite2D (or ColorRect, 20x100)
  +-- CollisionShape2D (RectangleShape2D, 20x100)
```

Ball Scene (`ball.tscn`)

1. **Scene → New Scene** → root node: **Area2D** → rename to `Ball`
2. Add child: **Sprite2D** (or **ColorRect** set to 20×20px, white)
3. Add child: **CollisionShape2D** → Inspector: Shape → New **RectangleShape2D** → size 20×20
4. Attach `ball.gd` to the Ball root node
5. **Connect signal**: Select Ball → Node panel → Signals → double-click `area_entered` → Connect to self → method `_on_area_entered`
6. Save as `ball.tscn`

```
Ball (Area2D)
  +-- Sprite2D (or ColorRect, 20x20)
  +-- CollisionShape2D (RectangleShape2D, 20x20)
```

Game Scene (`game.tscn`)

1. **Scene → New Scene** → root node: **Node2D** → rename to `Game`
2. Add child: **ColorRect** → size 800×600, dark color (background)
3. Drag `paddle.tscn` into scene → rename to `Player1` → position: `(50, 300)`
4. Drag `paddle.tscn` into scene again → rename to `Player2` → position: `(750, 300)`
5. Drag `ball.tscn` into scene → position: `(400, 300)`
6. Add child: **Label** → rename to `Score1` → position: `(350, 20)` → text: `0` → font size: 48
7. Add child: **Label** → rename to `Score2` → position: `(430, 20)` → text: `0` → font size: 48

8. Attach `game.gd` to the Game root node
9. Save as `game.tscn`

```
Game (Node2D)
  +-- Background (ColorRect, 800x600)
  +-- Player1 (paddle.tscn, position: 50, 300)
  +-- Player2 (paddle.tscn, position: 750, 300)
  +-- Ball (ball.tscn, position: 400, 300)
  +-- Score1 (Label, position: 350, 20, text: "0")
  +-- Score2 (Label, position: 430, 20, text: "0")
```

Pre-Built Code Walkthrough

```
extends Area2D

var speed = 400
var move_up = "ui_up"      # <-- Will become @export
var move_down = "ui_down"  # <-- Will become @export

func _process(delta):
    if Input.is_action_pressed(move_up):
        position.y = position.y - speed * delta
    if Input.is_action_pressed(move_down):
        position.y = position.y + speed * delta
    position.y = clamp(position.y, 50, 550)
```

Paddle Script (starter) This is the Wanderer pattern: input polling, delta-time movement, and clamping. The problem: both paddles use the same keys (`ui_up`/`ui_down`). We'll fix this with `@export`.

```
extends Area2D

var velocity = Vector2.ZERO
var speed = 400

func _ready():
    reset()

func reset():
    position = Vector2(400, 300)
    var direction = Vector2([-1, 1].pick_random(), randf_range(-0.5, 0.5))
    velocity = direction.normalized() * speed

func _process(delta):
    position = position + velocity * delta
    if position.y < 10 or position.y > 590:
        velocity.y = -velocity.y
```

Ball Script (starter) This is the Ricochet pattern: autonomous movement with wall bouncing. `reset()` places the ball at center with a random direction — same technique as Ricochet's `_ready()` random velocity.

Missing: scoring detection, paddle collision, and sound.

---

3. @export — Inspector-Editable Variables

The Problem

Both paddles use the same script with the same keys. We need Player 2 to use different controls (W/S instead of arrow keys).

We *could* create two separate scripts, but that's wasteful — the only difference is two variable values.

The Solution: `@export`

Adding `@export` before a variable makes it appear in the **Inspector panel**, where you can set a different value for each instance:

```
# Before (hardcoded - same for all instances):
var move_up = "ui_up"
var move_down = "ui_down"

# After (configurable per instance):
@export var move_up = "ui_up"
@export var move_down = "ui_down"
```

Configuring Player 2

1. Add `@export` to `move_up` and `move_down` in `paddle.gd`
2. Open `game.tscn`
3. Select the **Player2** node
4. In the Inspector, you'll see `Move Up` and `Move Down` fields
5. Change `Move Up` to `w`
6. Change `Move Down` to `s`
7. Run the game — Player 1 uses arrow keys, Player 2 uses W/S

The `@export` keyword is powerful: one script serves many instances with different settings. This is a fundamental game development pattern — configurable, reusable components.

---

4. Groups in the Editor

Recap: Groups in Code

In Fruit Frenzy, we used `add_to_group()` in code to tag spawned fruits:

```
func _ready():
    add_to_group("fruits")    # tag this node as a fruit
```

Then we checked membership in the collision handler with `is_in_group("fruits")`. This code-based approach works well for nodes spawned at runtime.

The Editor Approach

For nodes placed in the editor (like our paddles), there's a simpler way — the **Groups tab** in the Node panel:

1. Open `paddle.tscn`
2. Select the **Paddle** root node
3. Go to **Node** panel → **Groups** tab
4. Type `paddle` → click **Add**

Now every instance of `paddle.tscn` is automatically in the `"paddle"` group — both Player1 and Player2. No code needed.

Code Groups vs Editor Groups

	Code (`add_to_group`)	Editor (Groups tab)
Used in	Fruit Frenzy	Pong
Best for	Nodes spawned at runtime	Nodes placed in the editor
Checked with	`is_in_group()`	`is_in_group()` — same!

The checking side is identical either way — only the assignment method differs. Use code groups for dynamically spawned nodes, editor groups for nodes you place manually in scenes.

---

5. Custom Signals

Built-in vs Custom Signals

In Fruit Frenzy, you used **built-in signals** — events that Godot defines and emits automatically:

Signal	Defined By	Emitted When
`area_entered`	Godot (Area2D)	Two areas overlap
`timeout`	Godot (Timer)	Timer completes a cycle

Now you'll create **custom signals** — events YOU define and emit:

Signal	Defined By	Emitted When
`scored`	You (Ball script)	Ball passes left/right edge

Defining a Custom Signal

Add this line at the top of `ball.gd`:

```
extends Area2D

signal scored(player)    # <-- NEW: define the signal

var velocity = Vector2.ZERO
var speed = 400
```

- `signal` — keyword to define a new signal
- `scored` — the name you chose
- `(player)` — parameter: which player scored (1 or 2)

Emitting the Signal

Add scoring detection to `_process()` in `ball.gd`:

```
func _process(delta):
    position = position + velocity * delta

    if position.y < 10 or position.y > 590:
        velocity.y = -velocity.y

    # Ball passes left edge -> Player 2 scores
    if position.x < 0:
        scored.emit(2)
        reset()

    # Ball passes right edge -> Player 1 scores
    if position.x > 800:
        scored.emit(1)
        reset()
```

`scored.emit(2)` fires the signal with the value `2` as the player parameter. The ball doesn't know about scores or labels — it just announces the event. The game script handles the response.

Connecting the Custom Signal

In `game.gd`, connect the signal in `_ready()`:

```
func _ready():
    $Ball.scored.connect(_on_ball_scored)
```

This is the same `.connect()` syntax from Fruit Frenzy's Timer. The only difference: `scored` is a signal YOU defined, not a built-in one.

Built-in vs Custom: Comparison

Built-in Signal	Custom Signal
Defined by	Godot	You (`signal name(params)`)
Emitted by	Godot automatically	You (`signal_name.emit(...)`)
Connected with	`.connect(handler)`	`.connect(handler)` — same!
Example	`area_entered`	`scored`

The connection and handling pattern is identical. Once you understand built-in signals, custom signals are a natural extension.

---

6. Ball-Paddle Collision

The Basic Bounce

Complete the `_on_area_entered()` function in `ball.gd`:

```
func _on_area_entered(area):
    if area.is_in_group("paddle"):
        velocity.x = -velocity.x
        position.x = position.x + sign(velocity.x) * 10
```

- **`is_in_group("paddle")`** — only react to paddles (using the group we added)
- **`velocity.x = -velocity.x`** — reverse horizontal direction (bounce)
- **`position.x += sign(velocity.x) * 10`** — nudge the ball away from the paddle. Without this, the ball can get stuck inside the paddle and trigger multiple collisions

`sign()` returns `+1` or `-1` based on direction, so the nudge always pushes the ball away from the paddle.

Adding Angle Control

A flat bounce (ball goes straight back) is boring. Real Pong varies the bounce angle based on WHERE the ball hits the paddle:

- **Top of paddle** → ball goes up
- **Center of paddle** → ball goes straight
- **Bottom of paddle** → ball goes down

```
# How far from paddle center did the ball hit?
var hit_offset = (position.y - area.position.y) / 50

# Convert to vertical velocity
velocity.y = hit_offset * speed * 0.75

# Keep total speed consistent
velocity = velocity.normalized() * speed
```

- `hit_offset` ranges from roughly -1 (top) to +1 (bottom)
- Multiplying by `speed * 0.75` converts this to a vertical velocity component
- `velocity.normalized() * speed` ensures the ball always moves at the same total speed, regardless of angle. Without this, steep angles would make the ball faster.

Complete Collision Code

```
func _on_area_entered(area):
    if area.is_in_group("paddle"):
        # 1. Horizontal bounce
        velocity.x = -velocity.x

        # 2. Prevent double-hit
        position.x = position.x + sign(velocity.x) * 10

        # 3. Angle based on hit position
        var hit_offset = (position.y - area.position.y) / 50
        velocity.y = hit_offset * speed * 0.75

        # 4. Maintain consistent speed
        velocity = velocity.normalized() * speed
```

Four steps: bounce → nudge → angle → normalize. This is what makes Pong feel like a skill game — players can aim their returns by hitting different parts of the paddle.

---

7. Game Assembly & Scoring

Wiring the Signal

Complete the `game.gd` script:

```
extends Node2D

var score1 = 0
var score2 = 0

func _ready():
    $Ball.scored.connect(_on_ball_scored)

func _on_ball_scored(player):
    if player == 1:
        score1 = score1 + 1
        $Score1.text = str(score1)
    else:
        score2 = score2 + 1
        $Score2.text = str(score2)
```

Signal Flow

Here's the complete data flow when Player 2 scores:

1. Ball moves past the left edge (`position.x < 0`)
2. Ball calls `scored.emit(2)` — fires the custom signal
3. Game receives the signal in `_on_ball_scored(2)`
4. Game increments `score2` and updates `$Score2.text`
5. Ball calls `reset()` — returns to center with a new random direction

The ball announces the event. The game handles the response. Neither needs to know the other's internal details — signals keep them decoupled.

---

8. Sound Effects

Where to Get Sound Effects

Source	Type	Best For
jsfxr (sfxr.me)	Generator (browser)	Retro blips, hits, jumps, pickups
bfxr (bfxr.net)	Generator (desktop)	Same style, more controls
freesound.org	Library (free, CC)	Realistic sounds, ambient, music
kenney.nl/assets	Library (CC0)	Game-ready packs, no attribution needed

For Pong: open **jsfxr**, click the **Hit/Hurt** preset, randomize until it sounds good, and export as `.wav`. Godot supports `.wav` and `.ogg` formats.

AudioStreamPlayer Setup

1. Open `game.tscn`
2. Select the `Game` node → add child: **AudioStreamPlayer** → rename to `HitSound`
3. In Inspector: drag your `.wav` or `.ogg` sound file into the **Stream** property

Playing Sound on Paddle Hit

Add this line at the end of the collision handler in `ball.gd`:

```
func _on_area_entered(area):
    if area.is_in_group("paddle"):
        velocity.x = -velocity.x
        position.x = position.x + sign(velocity.x) * 10

        var hit_offset = (position.y - area.position.y) / 50
        velocity.y = hit_offset * speed * 0.75
        velocity = velocity.normalized() * speed

        # Play hit sound
        get_parent().get_node("HitSound").play()
```

This uses the same `get_parent()` pattern from Fruit Frenzy: - `get_parent()` — go up to the Game node - `.get_node("HitSound")` — find the AudioStreamPlayer - `.play()` — play the sound

Every paddle hit now produces audio feedback — a small addition that makes the game feel much more polished.

---

9. Complete Final Scripts

Paddle Script (`paddle.gd`)

```
extends Area2D

var speed = 400

@export var move_up = "ui_up"
@export var move_down = "ui_down"

func _process(delta):
    if Input.is_action_pressed(move_up):
        position.y = position.y - speed * delta
    if Input.is_action_pressed(move_down):
        position.y = position.y + speed * delta
    position.y = clamp(position.y, 50, 550)
```

Ball Script (`ball.gd`)

```
extends Area2D

signal scored(player)

var velocity = Vector2.ZERO
var speed = 400

func _ready():
    reset()

func reset():
    position = Vector2(400, 300)
    var direction = Vector2([-1, 1].pick_random(), randf_range(-0.5, 0.5))
    velocity = direction.normalized() * speed

func _process(delta):
    position = position + velocity * delta

    # Top and bottom wall bounce
    if position.y < 10 or position.y > 590:
        velocity.y = -velocity.y
```

```
    # Scoring
    if position.x < 0:
        scored.emit(2)
        reset()
    if position.x > 800:
        scored.emit(1)
        reset()

func _on_area_entered(area):
    if area.is_in_group("paddle"):
        velocity.x = -velocity.x
        position.x = position.x + sign(velocity.x) * 10

        var hit_offset = (position.y - area.position.y) / 50
        velocity.y = hit_offset * speed * 0.75
        velocity = velocity.normalized() * speed

        get_parent().get_node("HitSound").play()
```

Game Script (`game.gd`)

```
extends Node2D

var score1 = 0
var score2 = 0

func _ready():
    $Ball.scored.connect(_on_ball_scored)

func _on_ball_scored(player):
    if player == 1:
        score1 = score1 + 1
        $Score1.text = str(score1)
    else:
        score2 = score2 + 1
        $Score2.text = str(score2)
```

---

10. Summary of Concepts

New Concepts in Pong

Concept	What You Learned
`@export`	Makes a variable editable in the Inspector. One script can serve many instances with different settings.

Concept	What You Learned
Groups (editor)	Assign groups via the Node panel's Groups tab — no code needed for editor-placed nodes. Check with `is_in_group()`, same as the code approach from Fruit Frenzy.
Custom signals	Define your own events with `signal name(params)`. Emit with `.emit()`. Connect with `.connect()`.
`signal.emit()`	Fire a custom signal with data. Receivers get the data as function parameters.
AudioStreamPlayer	Node that plays sound. Assign a stream file, call `.play()` to play it.

Full Concept Map: 4 Games, 17 Concepts

```
Wanderer                Ricochet                Fruit Frenzy
    |                       |                       |
    +-- Nodes & Scenes      +-- Vector2             +-- Area2D
    +-- Sprite2D            +-- Velocity            +-- CollisionShape2D
    +-- Scripts             +-- Bouncing            +-- Signals (built-in)
    +-- _process(delta)     +-- _ready()            +-- Groups (code)
    +-- Input polling       +-- _input(event)       +-- queue_free()
    +-- clamp()             +-- preload()           +-- Timer
                            +-- instantiate()       +-- .connect()
                                                        |
                                                        v
                                                    Pong
                                                      |
                                                      +-- @export
                                                      +-- Groups (editor)
                                                      +-- Custom signals
                                                      +-- AudioStreamPlayer
```

---

11. Connection to Future Projects

Pong establishes patterns that carry forward throughout the course:

Pong Pattern	Future Use
`@export` for configuration	Every project — tweaking speed, health, damage in Inspector
Groups (code + editor) for identification	Enemies, collectibles, hazards, projectiles
Custom signals for events	Health changes, game over, level complete, UI updates
Scene composition (multiple `.tscn`)	Every project — reusable, modular components
Starter projects	Larger games will always build on provided foundations

You now have a solid Godot foundation: input, movement, collision, signals, sound, and project structure. Everything from here builds on these patterns.

Exercises

1. **Speed Tuning:** Experiment with ball speed (300, 400, 500) and paddle speed (300, 400, 500). What combination makes the game challenging but fair? Does it change with different paddle sizes?

2. **Center Line:** Add a visual center line to the game. Use a thin, tall **ColorRect** (2×500px, semi-transparent white) centered at x=400. This is pure visual polish — no code needed, just scene editing.

3. **Win Condition:** First player to 5 points wins. In `_on_ball_scored`, check if either score reached 5. If so, stop the ball with `$Ball.set_process(false)` and display "Player X Wins!" by updating a new label. *Hint: add a `WinLabel` to the scene, initially hidden, and show it when someone wins.*

4. **Speed Increase:** Make the ball 5% faster on each paddle hit. Add `speed *= 1.05` inside the collision handler. Cap at 800 with `speed = min(speed, 800)`. How does this affect gameplay pacing? *Hint: you may want to reset speed in `reset()` too.*

5. **AI Paddle:** Replace Player 2 with a simple AI. In `_process()`, instead of checking input, move the paddle toward the ball's y position. A simple approach: `position.y += (ball_y - position.y) * 3.0 * delta`. *Hint: use `@export var ball_path: NodePath` and `get_node(ball_path)` to access the ball node.*

6. **Screen Shake:** On paddle hit, briefly shake the camera. Add a **Camera2D** to the scene. In the collision handler, set a random offset (`camera.offset = Vector2(randf_range(-5, 5), randf_range(-5, 5))`). Use a Timer or Tween to reset the offset to `Vector2.ZERO` after 0.1 seconds.

7. **(Advanced) Multi-Ball:** Every 20 seconds, spawn a second ball using `preload()` and `instantiate()` (Ricochet pattern). Connect each new ball's `scored` signal to `_on_ball_scored`. Both balls score independently. *Hint: use a Timer with `wait_time = 20.0` to trigger spawning. Don't forget to connect the new ball's signal!*