CS 470 Game Development — Week 3, Lecture 8

Micro-Game 5: Flappy Bird

> **Goal:** Build Flappy Bird by adding gravity, flapping, pipe scrolling, collision, scoring, and a state machine to a pre-built starter project — using only familiar Godot nodes (Area2D, Sprite2D, Timer) with new *design patterns*.

---

1. Quick Recap: The Journey So Far

Over the last four lectures, you built four 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()`
Pong	`@export`, editor groups, custom signals (`signal scored`), `signal.emit()`, AudioStreamPlayer
Flappy Bird	Gravity (manual physics), state machines (`enum`, `if/elif`), infinite scrolling

How Flappy Bird Uses Everything

Flappy Bird introduces only three new patterns. Everything else is a direct reuse of concepts you already know:

Flappy Component	Concepts From	Origin
Bird movement	`position += velocity * delta`	Ricochet
Collision detection	Area2D + `area_entered` signal	Fruit Frenzy
Group-based identification	`is_in_group()`	Fruit Frenzy / Pong
Pipe spawning	`preload()` + `instantiate()` + Timer	Ricochet / Fruit Frenzy
Score display	Label + `str()`	Pong
Sound effects	AudioStreamPlayer + `.play()`	Pong
Scrolling background	`_process(delta)` movement	Wanderer

The three new patterns — gravity, state machines, and infinite scrolling — are *design patterns*, not new Godot features. You already know every node and function involved.

---

2. The Starter Project

Since most of Flappy Bird reuses familiar concepts, the starter project has those pre-implemented. You will add $ $25 lines of code across 3 files, focusing entirely on the new patterns.

What's Pre-Built

The starter includes four GDScript files:

- **`scrolling_bg.gd`** — Fully pre-built. Scrolls Sprite2D children left and wraps them. Used by both Background and Ground nodes.
- **`bird.gd`** — Has TODOs 1–2. Pre-built: constants, `die()`, `reset()`, visual tilt.
- **`pipe_pair.gd`** — Has TODO 3. Pre-built: speed variable.
- **`game.gd`** — Has TODOs 4–7. Pre-built: state enum, `start_countdown()`, `start_game()`, `game_over()`, `show_score_screen()`, `restart()`, `show_menu()`, signal connections in `_ready()`.

Scene Trees

The starter provides complete scenes. Here are their node hierarchies:

Bird Scene (`bird.tscn`)

```
Bird (Area2D)
+-- Sprite2D
+-- CollisionShape2D (CircleShape2D, ~8px radius)
```

The bird is an Area2D, consistent with Pong. No CharacterBody2D — we handle physics manually.

PipePair Scene (`pipe_pair.tscn`)

```
PipePair (Node2D)
+-- TopPipe (Area2D)          [group: "pipe"]
|   +-- Sprite2D              (flipped vertically)
|   +-- CollisionShape2D      (RectangleShape2D)
+-- BottomPipe (Area2D)       [group: "pipe"]
|   +-- Sprite2D
|   +-- CollisionShape2D      (RectangleShape2D)
+-- ScoreZone (Area2D)        [group: "score_zone"]
    +-- CollisionShape2D      (thin vertical strip in the gap)
```

TopPipe and BottomPipe are in the `"pipe"` group. ScoreZone is in the `"score_zone"` group. The entire PipePair is a single Node2D, so moving it moves all three Area2Ds together.

Game Scene (`game.tscn`)

```
Game (Node2D)
+-- Background (Node2D)         [scrolling_bg.gd, scroll_speed=20]
|   +-- BG1 (Sprite2D)          [background.png, centered=false]
|   +-- BG2 (Sprite2D)          [background.png, centered=false, x=512]
+-- Ground (Node2D)              [scrolling_bg.gd, scroll_speed=80, z_index=1]
```

```
|   +-- G1 (Sprite2D)            [ground.png, centered=false, y=280]
|   +-- G2 (Sprite2D)            [ground.png, centered=false, x=512, y=280]
|   +-- GroundCollider (Area2D)  [group: "pipe"]
|       +-- CollisionShape2D
+-- Bird (bird.tscn instance, position: 120, 144)
+-- PipeSpawnTimer (Timer)     [wait_time=2.0]
+-- ScoreLabel (Label)         [centered top, text="0"]
+-- ReadyLabel (Label)         [centered, text="Press Space to Start"]
+-- GameOverLabel (Label)      [centered, text="Game Over!", visible=false]
+-- Sound (Node2D)
      +-- HitSound (AudioStreamPlayer)
      +-- FlapSound (AudioStreamPlayer)
      +-- ScoreSound (AudioStreamPlayer)
      +-- Music (AudioStreamPlayer)
```

The viewport is 512$×$288 — a retro resolution that gives the game its pixel-art feel.

Notice that `GroundCollider` is in the `"pipe"` group. This means hitting the ground triggers the same collision response as hitting a pipe — game over. One group handles both cases.

Pre-Built Code Walkthrough

Bird Script (starter portion) The starter provides the constants, state variable, and helper functions:

```
extends Area2D

const GRAVITY = 800
const FLAP_STRENGTH = -250
const MAX_FALL_SPEED = 400

var velocity = Vector2.ZERO
var alive = false

func _process(delta):
    if not alive:
        return

    # === TODO 1: Gravity ===
    # === TODO 2: Flap ===

    # Visual: tilt bird based on velocity
    rotation = clamp(velocity.y / 400.0, -0.5, 1.0)

func die():
    alive = false
    rotation = 1.2

func reset():
```

```
    position = Vector2(120, 144)
    velocity = Vector2.ZERO
    rotation = 0
    alive = true
```

The `alive` flag starts as `false` so the bird doesn't fall on the menu screen. `start_game()` sets it to `true` when gameplay begins. The `die()` and `reset()` functions are called by `game.gd`. The visual tilt line tilts the bird sprite up when rising (negative velocity) and down when falling (positive velocity).

---

3. Scrolling Background

The Two-Sprite Trick

The bird in Flappy Bird doesn't actually move forward — the world scrolls past it. To create this illusion, we use a simple pattern: place two identical sprites side by side, scroll both to the left each frame, and wrap each one back to the right when it goes off-screen.

The starter project includes `scrolling_bg.gd`, which handles this pattern:

```
extends Node2D

@export var scroll_speed = 60

func _process(delta):
    for child in get_children():
        if child is Sprite2D:
            var w = child.texture.get_width()
            child.position.x -= scroll_speed * delta
            if child.position.x <= -w:
                child.position.x += w * 2
```

How it works:

- **`@export var scroll_speed`** — Controls how fast the sprites scroll. Set differently per node for parallax.
- **`var w = child.texture.get_width()`** — Reads the sprite's texture width automatically. When a sprite's left edge reaches `-w` (fully off-screen), it jumps forward by `w * 2`, placing it behind the other sprite.
- The loop iterates only `Sprite2D` children, so it skips the `GroundCollider` Area2D under the Ground node.
- Sprites use `centered = false` so `position.x` represents the left edge.

Parallax

The same script is attached to both `Background` and `Ground`, but with different `scroll_speed` values:

Node	scroll_speed	Effect
Background	20	Slow — appears far away
Ground	80	Fast — appears close to the camera

This speed difference creates a **parallax** effect: layers at different "distances" move at different speeds, giving the illusion of depth. The background drifts slowly behind while the ground rushes past below.

Both nodes have two Sprite2D children (BG1/BG2 and G1/G2) placed at `x = 0` and `x = texture width`. The script reads each sprite's texture width automatically, so the wrap point is always correct.

This pattern is fully pre-built — no TODO. When you run the game, the background and ground will already be scrolling.

---

4. Gravity

The Pattern

Gravity is a constant downward acceleration applied every frame. In real physics, gravity accelerates objects at 9.8 m/s^2. In games, we use the same principle with tuned values:

1. **Acceleration**: `velocity.y += GRAVITY * delta` — increase downward speed each frame
2. **Terminal velocity**: `velocity.y = clamp(velocity.y, -MAX_FALL_SPEED, MAX_FALL_SPEED)` — cap the speed so the bird doesn't fall infinitely fast
3. **Movement**: `position.y += velocity.y * delta` — apply velocity to position

This is the same `position += velocity * delta` pattern from Ricochet, but now velocity itself changes each frame.

TODO 1: Gravity

Add these three lines inside `_process()` in `bird.gd`:

```
# === TODO 1: Gravity ===
velocity.y += GRAVITY * delta
velocity.y = clamp(velocity.y, -MAX_FALL_SPEED, MAX_FALL_SPEED)
position.y += velocity.y * delta
```

After adding this, the bird will fall when you run the game. It accelerates downward smoothly — not a constant speed, but an increasing speed, just like real gravity.

Why Area2D Instead of CharacterBody2D?

Godot provides `CharacterBody2D` with built-in gravity and collision response. We use `Area2D` with manual physics for three reasons:

1. **Consistency** — Pong used the same approach. One pattern for the whole course so far.
2. **Matches GD50** — The Harvard course this curriculum draws from teaches the underlying math.

3. **Understanding** — When you eventually use `CharacterBody2D`, you will know what it does internally.

---

5. Flap Mechanic

TODO 2: Flap

Add this code after the gravity block in `bird.gd`:

```
# === TODO 2: Flap ===
if Input.is_action_just_pressed("flap"):
    velocity.y = FLAP_STRENGTH
```

How It Works

- `FLAP_STRENGTH` is −250 — a negative value means upward in Godot's coordinate system (y increases downward).
- This line *replaces* the current `velocity.y`, not adds to it. Whether the bird is falling at 100 or 400 pixels/second, a flap instantly sets velocity to −250. This gives consistent, predictable jumps.
- `is_action_just_pressed` (not `is_action_pressed`) ensures the flap triggers once per press. Holding Space does not keep the bird flying — the player must tap repeatedly.

The Gravity-Flap Cycle

Every frame:

1. Gravity pulls `velocity.y` toward positive (downward)
2. If the player presses Space, `velocity.y` is reset to −250 (upward)
3. Gravity immediately starts pulling it back down

This creates the characteristic arc: a quick upward burst followed by a gradual fall. The entire mechanic is two concepts — gravity (TODO 1) and flap (TODO 2) — working together.

---

6. Pipe Pairs & Movement

The PipePair Design

Each PipePair is a Node2D with three Area2D children:

- **TopPipe** and **BottomPipe** (group: `"pipe"`) — the obstacles the bird must avoid
- **ScoreZone** (group: `"score_zone"`) — an invisible Area2D positioned in the gap between the pipes

When the bird enters a pipe, it is game over. When the bird enters the score zone, the player earns a point. This reuses the group-based identification pattern from Fruit Frenzy and Pong.

TODO 3: Pipe Movement

Add pipe movement to `pipe_pair.gd`:

```
func _process(delta):
    position.x -= speed * delta
    if position.x < -100:
        queue_free()
```

How It Works

- The entire PipePair node moves left. Because TopPipe, BottomPipe, and ScoreZone are children, they all move together — no need to move each one individually.
- `speed` is set to 60, giving a steady scroll that feels right for the game's retro resolution.
- When the PipePair passes x=−100 (safely off-screen), `queue_free()` removes it and all its children from memory. Without this, pipes would accumulate forever and eventually slow down the game.

---

7. Pipe Spawning

Keeping Pipes on Screen

Two constraints prevent unfair pipe placement:

Margin clamping. The gap center (in world coordinates) is `PipePair.position.y + 144`. To keep the gap at least `PIPE_MARGIN` pixels from the viewport top and the ground (y=280):

- Gap top $\geq$ `PIPE_MARGIN` $\Rightarrow$ `PipePair.y` $\geq$ `PIPE_MARGIN + PIPE_GAP/2 - 144`
- Gap bottom $\leq$ `280 - PIPE_MARGIN` $\Rightarrow$ `PipePair.y` $\leq$ `280 - PIPE_MARGIN - PIPE_GAP/2 - 144`

With `PIPE_GAP=80` and `PIPE_MARGIN=20`, this gives `PIPE_Y_MIN = -84` and `PIPE_Y_MAX = 76`.

Smooth path. Without a limit, consecutive pipes can jump wildly in Y, creating an unplayable sequence. `MAX_PIPE_SHIFT` caps how much the gap can move between consecutive pipes. We track `last_pipe_y` and use `max()`/`min()` to narrow the random range. When difficulty increases later, raising `MAX_PIPE_SHIFT` makes wider swings (harder).

TODO 4: Pipe Spawning

The constants at the top of `game.gd`:

```
const PIPE_GAP = 80
const PIPE_MARGIN = 20
const PIPE_Y_MIN = PIPE_MARGIN + PIPE_GAP / 2 - 144
const PIPE_Y_MAX = 280 - PIPE_MARGIN - PIPE_GAP / 2 - 144
const MAX_PIPE_SHIFT = 60

var last_pipe_y = 0.0
```

Complete `_on_pipe_spawn_timer_timeout()` in `game.gd`:

```
func _on_pipe_spawn_timer_timeout():
    var pipe = pipe_scene.instantiate()
    pipe.gap = PIPE_GAP
    pipe.position.x = 550
    var min_y = max(PIPE_Y_MIN, last_pipe_y - MAX_PIPE_SHIFT)
    var max_y = min(PIPE_Y_MAX, last_pipe_y + MAX_PIPE_SHIFT)
    pipe.position.y = randf_range(min_y, max_y)
    last_pipe_y = pipe.position.y
    pipe.add_to_group("pipe_pairs")
    add_child(pipe)
```

How It Works

This is the same `preload()` + `instantiate()` + `add_child()` pattern from Ricochet and Fruit Frenzy, with two extra clamping steps:

1. `pipe_scene` was loaded at the top of the script with `preload("res://prefabs/pipe_pair.tscn")`
2. `instantiate()` creates a new copy of the PipePair scene
3. `position.x = 550` places it just off the right edge of the 512px viewport
4. `max(PIPE_Y_MIN, last_pipe_y - MAX_PIPE_SHIFT)` ensures the pipe stays within the viewport margin *and* within `MAX_PIPE_SHIFT` of the previous pipe
5. `min(PIPE_Y_MAX, last_pipe_y + MAX_PIPE_SHIFT)` applies the same logic to the lower bound
6. `randf_range(min_y, max_y)` randomizes within the clamped window
7. `last_pipe_y` is updated so the next pipe is anchored to this one
8. `add_to_group("pipe_pairs")` adds the pipe to a group so we can freeze all pipes later (in the DYING state)
9. `add_child()` adds it to the scene tree, where `_process()` starts running immediately

The `restart()` function resets `last_pipe_y = 0.0` so the first pipe after restart isn't anchored to the last death position.

Draw Order: Why Ground Has `z_index = 1`

`add_child(pipe)` appends each new pipe as the last child of the Game node. In Godot 2D, children are drawn top-to-bottom — later children draw on top of earlier ones. This means pipes would draw *over* the ground.

The fix is simple: the Ground node has `z_index = 1` set in the Inspector. Pipes default to `z_index = 0`. A higher `z_index` always draws on top, regardless of tree order. This ensures the ground covers pipe bottoms, just as it should.

Spawning Wiring

The `PipeSpawnTimer` fires every 2 seconds, creating a steady stream of obstacles. Combined with pipe movement (TODO 3) and cleanup (`queue_free()`), this creates endless obstacle generation with no memory buildup.

The wiring is pre-built in `_ready()`:

```
$PipeSpawnTimer.timeout.connect(_on_pipe_spawn_timer_timeout)
```

To test right away, temporarily add `$PipeSpawnTimer.start()` to `_ready()` so pipes begin spawning when the game launches. We will remove this line in Section 9 (State Machine) once `start_game()` takes over timer control.

---

8. Collision & Scoring

TODO 5: Collision Handling

Complete `_on_bird_area_entered()` in `game.gd`:

```
func _on_bird_area_entered(area):
    if area.is_in_group("pipe"):
        game_over()
    elif area.is_in_group("score_zone"):
        score += 1
        $ScoreLabel.text = str(score)
        $Sound/ScoreSound.play()
        area.queue_free()
```

How It Works

The bird's `area_entered` signal fires whenever it overlaps any Area2D. We use `is_in_group()` to determine what was hit — the same pattern from Fruit Frenzy:

- **Hit a pipe** (or the ground, which is also in the `"pipe"` group) → call `game_over()`, which stops the timer, kills the bird, freezes all movement, and after a 1-second pause shows the score screen.
- **Hit a score zone** → increment score, update the label, play a sound, and `queue_free()` the score zone so passing through the same gap twice does not award double points.

The signal connection was pre-built in `_ready()`:

```
$Bird.area_entered.connect(_on_bird_area_entered)
```

This is the same `.connect()` pattern from Fruit Frenzy and Pong — the only difference is that we connect in the game script rather than in the bird script, because scoring is a game-level concern.

---

9. State Machine

The Problem

Without a state machine, the game has a subtle bug: pressing Space during Game Over would restart the game AND make the bird flap simultaneously. Input handling depends on *context* — what should happen when Space is pressed depends on whether the game is waiting to start, actively playing, or showing the Game Over screen.

The Solution: `enum` + `if/elif`

A **state machine** defines a set of states and routes behavior based on the current state. In GDScript:

```
enum State { MENU, COUNTDOWN, PLAYING, DYING, SCORE }
var current_state = State.MENU
```

`enum` creates named constants: `State.MENU = 0`, `State.COUNTDOWN = 1`, `State.PLAYING = 2`, `State.DYING = 3`, `State.SCORE = 4`. Using names instead of raw numbers makes the code self-documenting.

Cleanup: Remove Temporary Timer Start

Now that `start_game()` will handle starting the pipe timer, remove the temporary `$PipeSpawnTimer.start()` line you added to `_ready()` in Section 7.

TODO 6: State Machine

Complete `_process()` in `game.gd`:

```
func _process(_delta):
    if current_state == State.MENU:
        if Input.is_action_just_pressed("flap"):
            start_countdown()
    elif current_state == State.COUNTDOWN:
        pass  # Countdown runs itself via await
    elif current_state == State.PLAYING:
        pass  # Bird and pipes update themselves
    elif current_state == State.DYING:
        pass  # Freeze runs itself via await
    elif current_state == State.SCORE:
        if Input.is_action_just_pressed("flap"):
            restart()
```

State Transitions

The game flows through five states:

```
MENU --[Space]--> COUNTDOWN --[3s]--> PLAYING --[Hit]--> DYING --[1s]--> SCORE --[Space]--> MEN
```

- **MENU**: The "Press Space to Start" label is shown. Space → `start_countdown()`.
- **COUNTDOWN**: Shows 3, 2, 1 on screen using `$ReadyLabel`. After 3 seconds → `start_game()`, which sets state to PLAYING and starts the pipe timer.
- **PLAYING**: The bird falls and flaps (handled by `bird.gd`). Pipes scroll (handled by `pipe_pair.gd`). The game script does nothing extra — `pass`. A pipe hit → `game_over()`.
- **DYING**: All movement freezes (bird, pipes, backgrounds) for 1 second. After the delay → `show_score_screen()`.
- **SCORE**: Shows "Score: X / Press Space to Continue" on the `$GameOverLabel`. Space → `restart()`, which cleans up, unfreezes, and returns to MENU.

Why State Machines Matter

State machines solve the input conflict cleanly: each state handles Space independently. In MENU, Space starts the countdown. In PLAYING, bird.gd handles Space for flapping. In SCORE, Space restarts.

State machines also make the game easy to extend. Want to add a pause feature? Add `State.PAUSED` to the enum and a new case to the `if/elif` block. The existing states remain untouched.

---

10. Menus

Section 9 introduced the state machine pattern with 5 states. This section walks through the *implementation* of each new state's behavior — countdown, freeze, and score screen. This code is pre-built in the starter.

Countdown

The `start_countdown()` function uses GDScript's `await` keyword to pause execution until a timer fires:

```
func start_countdown():
    current_state = State.COUNTDOWN
    $ReadyLabel.text = "3"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.text = "2"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.text = "1"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.visible = false
    start_game()
```

`await get_tree().create_timer(1.0).timeout` creates a one-shot timer that fires after 1 second, then pauses the function until that timer fires. The function resumes on the next line. This is a clean way to write sequential delays without nesting callbacks.

The countdown reuses `$ReadyLabel` — no new scene nodes needed. It simply changes the label's text from "Press Space to Start" to "3", "2", "1", then hides it.

Dying and Freeze

When the bird hits a pipe or the ground, `game_over()` freezes *everything* in place:

```
func game_over():
    current_state = State.DYING
    $Bird.die()
    $PipeSpawnTimer.stop()
    $Sound/HitSound.play()
    $Background.set_process(false)
    $Ground.set_process(false)
```

```
    get_tree().call_group("pipe_pairs", "set_process", false)
    await get_tree().create_timer(1.0).timeout
    show_score_screen()
```

`set_process(false)` stops `_process()` from being called on a node, effectively freezing its movement. The bird is already frozen by `die()` setting `alive = false`. After a 1-second pause, the score screen appears.

Note: spawned pipes are added to the `"pipe_pairs"` group (in TODO 4's spawn function: `pipe.add_to_group("pipe_pairs")`), so `call_group("pipe_pairs", "set_process", false)` freezes all pipes at once.

Score Screen

```
func show_score_screen():
    current_state = State.SCORE
    $GameOverLabel.text = "Score: " + str(score) + "\nPress Space to Continue"
    $GameOverLabel.visible = true
```

This reuses `$GameOverLabel` to show the final score. The `\n` creates a line break.

Restart and Unfreeze

When the player presses Space on the score screen, `restart()` cleans up and returns to the menu:

```
func restart():
    $GameOverLabel.visible = false
    get_tree().call_group("pipe_pairs", "queue_free")
    $Bird.reset()
    last_pipe_y = 0.0
    $Background.set_process(true)
    $Ground.set_process(true)
    show_menu()
```

`set_process(true)` unfreezes Background and Ground, restoring their scrolling. We free by the `"pipe_pairs"` group (not `"pipe"`) because `GroundCollider` is also in the `"pipe"` group — freeing that group would destroy the ground collider permanently. Freeing the parent `PipePair` Node2D also frees its children (TopPipe, BottomPipe, ScoreZone).

Menu

```
func show_menu():
    current_state = State.MENU
    $Bird.alive = false
    $ReadyLabel.text = "Press Space to Start"
    $ReadyLabel.visible = true
    $ScoreLabel.text = "0"
    score = 0
```

`$Bird.alive = false` freezes the bird on the menu screen. After `restart()` calls `$Bird.reset()` (which sets `alive = true`), `show_menu()` immediately overrides it back to `false`. The bird stays still until `start_game()` sets `alive = true` again.

---

11. Sound

TODO 7: Hit Sound

The hit sound is already included in the pre-built `game_over()` function:

```
$Sound/HitSound.play()
```

The starter already has three `AudioStreamPlayer` nodes organized under a `Sound` container in the scene tree: `HitSound`, `FlapSound`, and `ScoreSound`. The score sound is already wired up in TODO 5's collision handler (`$Sound/ScoreSound.play()`). The hit sound plays in `game_over()`.

For the flap sound, you could add `get_parent().get_node("Sound/FlapSound").play()` in `bird.gd` inside the flap block (see Exercise 2). This uses the same `get_parent()` pattern from Pong — navigate up to the Game node, then find the AudioStreamPlayer by name.

---

12. Complete Final Scripts

Here are all scripts with TODOs filled in. Lines marked with `# <--` are the lines you added.

Scrolling Background Script (`scrolling_bg.gd`)

Fully pre-built — no TODOs:

```
extends Node2D

@export var scroll_speed = 60

func _process(delta):
    for child in get_children():
        if child is Sprite2D:
            var w = child.texture.get_width()
            child.position.x -= scroll_speed * delta
            if child.position.x <= -w:
                child.position.x += w * 2
```

Bird Script (`bird.gd`)

TODOs 1 and 2 filled in:

```
extends Area2D

const GRAVITY = 800
const FLAP_STRENGTH = -250
const MAX_FALL_SPEED = 400
```

```
var velocity = Vector2.ZERO
var alive = false

func _process(delta):
    if not alive:
        return

    # Gravity
    velocity.y += GRAVITY * delta                                    # <--
    velocity.y = clamp(velocity.y, -MAX_FALL_SPEED, MAX_FALL_SPEED) # <--
    position.y += velocity.y * delta                                  # <--

    # Flap
    if Input.is_action_just_pressed("flap"):                         # <--
        velocity.y = FLAP_STRENGTH                                   # <--

    # Visual tilt
    rotation = clamp(velocity.y / 400.0, -0.5, 1.0)

func die():
    alive = false
    rotation = 1.2

func reset():
    position = Vector2(120, 144)
    velocity = Vector2.ZERO
    rotation = 0
    alive = true
```

Pipe Pair Script (`pipe_pair.gd`)

TODO 3 filled in:

```
extends Node2D

var speed = 60

func _process(delta):
    position.x -= speed * delta       # <--
    if position.x < -100:             # <--
        queue_free()                   # <--
```

Game Script (`game.gd`)

TODOs 4–7 filled in:

```
extends Node2D
```

```
enum State { MENU, COUNTDOWN, PLAYING, DYING, SCORE }
var current_state = State.MENU

var score = 0
var pipe_scene = preload("res://prefabs/pipe_pair.tscn")

const PIPE_GAP = 80
const PIPE_MARGIN = 20
const PIPE_Y_MIN = PIPE_MARGIN + PIPE_GAP / 2 - 144
const PIPE_Y_MAX = 280 - PIPE_MARGIN - PIPE_GAP / 2 - 144
const MAX_PIPE_SHIFT = 60

var last_pipe_y = 0.0

func _ready():
    $Bird.area_entered.connect(_on_bird_area_entered)
    $PipeSpawnTimer.timeout.connect(_on_pipe_spawn_timer_timeout)
    $Sound/Music.play()
    show_menu()

func _process(_delta):
    if current_state == State.MENU:                                  # <--
        if Input.is_action_just_pressed("flap"):                     # <--
            start_countdown()                                        # <--
    elif current_state == State.COUNTDOWN:                            # <--
        pass                                                         # <--
    elif current_state == State.PLAYING:                              # <--
        pass                                                         # <--
    elif current_state == State.DYING:                                # <--
        pass                                                         # <--
    elif current_state == State.SCORE:                                # <--
        if Input.is_action_just_pressed("flap"):                     # <--
            restart()                                                # <--

func _on_pipe_spawn_timer_timeout():
    var pipe = pipe_scene.instantiate()                              # <--
    pipe.gap = PIPE_GAP                                              # <--
    pipe.position.x = 550                                            # <--
    var min_y = max(PIPE_Y_MIN, last_pipe_y - MAX_PIPE_SHIFT)       # <--
    var max_y = min(PIPE_Y_MAX, last_pipe_y + MAX_PIPE_SHIFT)       # <--
    pipe.position.y = randf_range(min_y, max_y)                       # <--
    last_pipe_y = pipe.position.y                                    # <--
    pipe.add_to_group("pipe_pairs")                                  # <--
    add_child(pipe)                                                  # <--

func _on_bird_area_entered(area):
    if area.is_in_group("pipe"):                                     # <--
```

```
        game_over()                                                    # <--
    elif area.is_in_group("score_zone"):                               # <--
        score += 1                                                     # <--
        $ScoreLabel.text = str(score)                                  # <--
        $Sound/ScoreSound.play()                                       # <--
        area.queue_free()                                              # <--

func start_countdown():
    current_state = State.COUNTDOWN
    $ReadyLabel.text = "3"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.text = "2"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.text = "1"
    await get_tree().create_timer(1.0).timeout
    $ReadyLabel.visible = false
    start_game()

func start_game():
    current_state = State.PLAYING
    score = 0
    $ScoreLabel.text = "0"
    $Bird.alive = true
    $PipeSpawnTimer.start()

func game_over():
    current_state = State.DYING
    $Bird.die()
    $PipeSpawnTimer.stop()
    $Sound/HitSound.play()                                             # <--
    $Background.set_process(false)
    $Ground.set_process(false)
    get_tree().call_group("pipe_pairs", "set_process", false)
    await get_tree().create_timer(1.0).timeout
    show_score_screen()

func show_score_screen():
    current_state = State.SCORE
    $GameOverLabel.text = "Score: " + str(score) + "\nPress Space to Continue"
    $GameOverLabel.visible = true

func restart():
    $GameOverLabel.visible = false
    get_tree().call_group("pipe_pairs", "queue_free")
    $Bird.reset()
    last_pipe_y = 0.0
    $Background.set_process(true)
```

```
    $Ground.set_process(true)
    show_menu()

func show_menu():
    current_state = State.MENU
    $Bird.alive = false
    $ReadyLabel.text = "Press Space to Start"
    $ReadyLabel.visible = true
    $ScoreLabel.text = "0"
    score = 0
```

---

13. Summary of Concepts

New Concepts in Flappy Bird

Concept	What You Learned
Gravity	Constant acceleration applied each frame: `velocity.y += GRAVITY * delta`. Combined with position update creates realistic falling.
Flap mechanic	Instantly set velocity to a negative value to jump upward. Uses `is_action_just_pressed` for single-press input.
Infinite scrolling	Two-sprite cycling for background/ground (wrap when off-screen); spawn-move-free for pipes. Different speeds create parallax depth.
State machine	`enum` defines states, `if/elif` routes input. Prevents input conflicts between game phases.
Pipe spawning	Timer + `preload()` + `instantiate()` + random Y position = endless obstacle generation.

Full Concept Map: 5 Games, 20 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
      |
      v
Flappy Bird
      |
      +-- Gravity (manual physics)
      +-- State machines (enum + i
      +-- Infinite scrolling
```

---

14. Exercises

Beginner

1. **Difficulty Tuning.** Experiment with `GRAVITY` (try 600, 800, 1000), `FLAP_STRENGTH` (try −200, −250, −300), `PIPE_GAP`, `PIPE_MARGIN`, and `MAX_PIPE_SHIFT`. Find a combination that feels fair but challenging. Write down which values you chose and why.

2. **Flap Sound.** Add one line in `bird.gd` inside the flap block to play the flap sound: `gdscript get_parent().get_node("Sound/FlapSound").play()` This uses the same `get_parent()` pattern from Pong. One line of code, big impact on game feel.

Intermediate

3. **Score-Based Speed.** Increase difficulty as the player scores higher. In `_on_bird_area_entered`, after incrementing the score, check if `score % 5 == 0` (every 5 points). If so, decrease `$PipeSpawnTimer.wait_time` by 0.1 (minimum 0.8 seconds). For newly spawned pipes, you can pass a higher speed by setting `pipe.speed` after `instantiate()`. Start the speed increase at 10 pixels/second per tier.

4. **Animated Bird.** Replace the bird's static `Sprite2D` with an `AnimatedSprite2D`. Create a `SpriteFrames` resource with a 3-frame wing-flap animation. Play the animation when the bird is alive, stop it when the bird dies. *Hint: call `$AnimatedSprite2D.play("flap")` in the flap block and `$AnimatedSprite2D.stop()` in `die()`.*

5. **High Score.** Track the highest score achieved across games (resets when the application closes). Add a `var high_score = 0` variable. In `game_over()`, update it with `high_score = max(high_score, score)`. On the Game Over screen, display both scores: update `$GameOverLabel.text` to `"Game Over!\nScore: " + str(score) + " | Best: " + str(high_score)`.

Advanced

7. **Pause State.** Add a `PAUSED` state to the state machine. Press Escape to toggle between `PLAYING` and `PAUSED`. When paused: stop the bird's physics (`$Bird.alive = false`), stop the pipe spawn timer, and pause all pipe movement (use `get_tree().call_group("pipe_pairs", "set_process", false)`). Show a "Paused" label. When unpaused, reverse all of these.

This demonstrates the extensibility of the state machine pattern — adding a new state requires no changes to the existing states.