Contents

CS 470 Game Development — Week 2, Lecture 6

Micro-Game 3: Fruit Frenzy

Goal: Fruits fall from the sky. Move a basket to catch them. Score points! Along the way, learn about Area2D, collision detection, signals, and timers.

---

1. Quick Recap: Ricochet Foundations

Last lecture, we built Ricochet — a bouncing logo that you could spawn by clicking. Here's what we used:

Concept	What You Learned
Velocity	Speed + direction combined as a `Vector2`
Autonomous movement	`position += velocity * delta` — objects move themselves
Bouncing	Reverse velocity component at screen boundaries
`preload()` / `instantiate()`	Load a scene file, create copies at runtime
`_input(event)`	Event-driven input for one-shot actions (mouse clicks)
`add_child()`	Add a node to the scene tree at runtime

What's Different Today?

	Ricochet	Fruit Frenzy
Root node	Node2D	Area2D
Objects interact?	No (bounce off walls only)	Yes (basket catches fruit)
How to detect contact?	Manual boundary `if` checks	Signals from Area2D
Object lifetime	Live forever (bounce back)	Destroyed when caught or off-screen

The key shift: from **boundary checking** to **collision detection**. In Ricochet, we manually compared positions to screen edges. In Fruit Frenzy, Godot automatically detects when two objects overlap.

---

2. New Node: Area2D

What is Area2D?

An `Area2D` is a Godot node that detects when other Area2D nodes overlap with it. Think of it as an invisible sensor:

- It doesn't apply physics (no gravity, no bouncing)
- It just **detects** when something enters or exits its space
- When overlap happens, it emits a **signal**

Use cases: collectibles (coins, power-ups), damage zones, trigger areas — and our basket catching fruit.

CollisionShape2D

An Area2D by itself can't detect anything — it needs a **shape** to define its boundaries:

- **Area2D** = the sensor (detects overlaps)

- **CollisionShape2D** = the shape (defines the hitbox)

An Area2D without a CollisionShape2D is like an alarm system with no sensors. Always add a shape as a child node.

Common shape types: - `RectangleShape2D` — for boxes and rectangular sprites - `CircleShape2D` — for round objects - `CapsuleShape2D` — for tall/pill-shaped objects

We'll use `RectangleShape2D` for both the basket and fruit.

Area2D vs Node2D

	Node2D	Area2D
Used in	Wanderer, Ricochet	Fruit Frenzy
Can move?	Yes	Yes
Detects overlaps?	No	Yes
Needs collision shape?	No	Yes
Emits collision signals?	No	`area_entered`, `area_exited`

Rule of thumb: if a node needs to **detect contact** with another node, use `Area2D`. If it just needs to exist and move, `Node2D` is fine.

---

3. Building the Basket

Create Basket Scene

1. Create a new scene
2. Root node: **Area2D** → rename to `Basket`
3. Add child: **Sprite2D** → assign a basket texture (or any placeholder image)
4. Add child: **CollisionShape2D**
5. In Inspector: Shape → New **RectangleShape2D**
6. Resize the rectangle to match the sprite
7. Save as `basket.tscn`

Scene tree:

```
Basket (Area2D)
  +-- Sprite2D
  +-- CollisionShape2D
```

Compare to Wanderer's scene tree:

```
Player (Node2D)
  +-- Sprite2D
```

Same idea, but `Area2D` replaces `Node2D` and we add a `CollisionShape2D` for detection.

The `$` Shorthand

In GDScript, `$NodeName` is shorthand for `get_node("NodeName")` — it gets a child node by name:

```
Basket (Area2D)        <- our script lives here
  +-- Sprite2D         <- $Sprite2D
  +-- CollisionShape2D <- $CollisionShape2D
```

You'll see `$` used throughout our scripts to reference child nodes. It only works for direct children (or paths like `$Child/Grandchild`).

Getting Sprite Size

To get a sprite's **actual display size**, you need to account for scale:

```
# Raw texture size (pixels in the image file)
var tex_w = $Sprite2D.texture.get_width()
var tex_h = $Sprite2D.texture.get_height()

# Actual display size (accounts for scale!)
var display_w = tex_w * $Sprite2D.scale.x
var display_h = tex_h * $Sprite2D.scale.y
```

Why does this matter? If your sprite is 128px wide but scaled to `0.5`, it displays as 64px. Always multiply by `scale` when you need the real on-screen size. This is a common gotcha — students resize sprites in the editor by changing scale, then wonder why their collision math is off.

Basket Script

```
extends Area2D

var speed = 400

func _process(delta):
    if Input.is_action_pressed("ui_left"):
        position.x = position.x - speed * delta
    if Input.is_action_pressed("ui_right"):
        position.x = position.x + speed * delta

    var half_w = $Sprite2D.texture.get_width() * $Sprite2D.scale.x / 2   # display half-width
    var screen_w = get_viewport_rect().size.x                             # viewport width
    position.x = clamp(position.x, half_w, screen_w - half_w)            # keep edges on screen
```

This should look very familiar — it's the **Wanderer pattern**:

- `extends Area2D` instead of `extends Node2D` (the only structural change)
- Same `Input.is_action_pressed()` polling for left/right movement
- Same `speed * delta` for frame-rate-independent movement
- Same `clamp()` to keep the basket on screen

The clamp boundaries are now **dynamic** — we compute the sprite's half-width using its texture and scale, and get the screen width from `get_viewport_rect().size.x`. No hardcoded pixel values.

---

4. Building the Fruit

Create Fruit Scene

1. New scene → root node: **Area2D** → rename to `Fruit`
2. Add child: **Sprite2D** → assign a fruit texture (or colored shape)
3. Add child: **CollisionShape2D** → New **RectangleShape2D**, sized to match
4. Save as `fruit.tscn`

Scene tree:

```
Fruit (Area2D)
  +-- Sprite2D
  +-- CollisionShape2D
```

Same structure as the Basket. Both need `Area2D` because both need to detect overlap with each other.

Fruit Script

```
extends Area2D

var speed = 200                                                          # pixels per second

func _ready():
    add_to_group("fruits")                                               # tag for collision check

func _process(delta):
    position.y = position.y + speed * delta                              # fall downward

    var half_h = $Sprite2D.texture.get_height() * $Sprite2D.scale.y / 2  # display half-height
    var screen_h = get_viewport_rect().size.y                            # viewport height
    if position.y > screen_h + half_h:                                   # fully below screen?
        queue_free()                                                     # destroy this fruit
```

Three things happening here:

1. **Group membership** — `add_to_group("fruits")` tags this node so the basket can reliably identify it during collision (more on this below).
2. **Fall downward** — `position.y += speed * delta` moves the fruit down every frame. This is like Ricochet's autonomous movement, but only in one direction.
3. **Self-destruct** — when the fruit is fully below the screen, it calls `queue_free()` to remove itself. We use the sprite's display half-height plus the viewport height to ensure it's completely off screen before destroying it.

Understanding `queue_free()`

`queue_free()` removes a node from the scene tree and frees its memory:

- The node stops existing — no more `_process()` calls, no more rendering
- "Queue" means it happens safely at the end of the current frame, not mid-processing
- Without this, off-screen fruits would pile up forever — a **memory leak**

In Ricochet, logos bounced back and lived forever. In Fruit Frenzy, missed fruits are **destroyed**. This is an important pattern: any dynamically spawned object should eventually be freed.

Why Groups Instead of Name Checking?

You might think we could check `area.name.begins_with("Fruit")` to identify fruits during collision. **This doesn't work reliably.** When you `instantiate()` multiple copies of a scene, Godot auto-renames duplicates to avoid conflicts: the first is `"Fruit"`, but subsequent ones become `"@Fruit@2"`, `"@Fruit@3"`, etc. — they start with `@`, not `"Fruit"`.

Instead, we use **groups**: call `add_to_group("fruits")` in `_ready()`, then check `area.is_in_group("fruits")` during collision. Groups are stable regardless of how Godot renames nodes.

---

5. Signals and Collision

What Are Signals?

Signals are Godot's event system. They let one node **notify** another that something happened.

Analogy: a doorbell - Someone presses the button → the bell rings - You hear the ring → you open the door - The person doesn't need to know HOW you'll respond

Signals work the same way: - An event occurs (two areas overlap) → the signal fires - A connected function runs in response - The emitter doesn't know or care what the handler does

Signal Flow: Fruit Meets Basket

Here's what happens when a fruit overlaps with the basket:

1. Fruit moves into Basket's collision area
2. Basket automatically emits the `area_entered` signal
3. The signal calls the connected function `_on_area_entered(area)`
4. The handler destroys the fruit and updates the score

The `area` parameter is the **other** Area2D that entered — in our case, the Fruit node.

Connecting the `area_entered` Signal

To make this work, we need to connect the Basket's signal to a handler function:

1. Open `game.tscn` in the editor
2. Select the **Basket** node
3. Go to the **Node** panel (right side, next to Inspector)
4. Click the **Signals** tab
5. Find and double-click `area_entered(area: Area2D)`
6. In the dialog, connect to **Basket** with method name: `_on_area_entered`
7. Click **Connect**

A green connection icon will appear next to the function in the script editor, confirming the signal is connected.

The `_on_area_entered` Handler

Add this function to the **Basket** script:

```
func _on_area_entered(area):
    if area.is_in_group("fruits"):                                    # is it a fruit?
        area.queue_free()                                             # destroy it
        get_parent().add_score()                                      # tell Game we scored
```

Line by line: - `area` — the other Area2D that overlapped with the basket (the Fruit) - `area.is_in_group("fruits")` — check if the area belongs to the "fruits" group (reliable, unlike name checking) - `area.queue_free()` — destroy the caught fruit (remove it from the game) - `get_parent().add_score()` — get the Game node (Basket's parent) and call its scoring function

Understanding `get_parent()`

```
Game (Node2D)          <- get_parent() returns this
  +-- Basket (Area2D)  <- we're here
  +-- ScoreLabel
  +-- SpawnTimer
```

`get_parent()` returns the node directly above in the scene tree. Since Basket is a child of Game, `get_parent()` gives us the Game node. We can then call any function on it — in this case, `add_score()`.

This is a simple way for a child to communicate upward. In more advanced projects, you'd use custom signals for more decoupled communication, but `get_parent()` works well for simple cases.

---

6. Game Scene: Spawning and Score

Create Main Scene

1. New scene → root: **Node2D** → rename to `Game`
2. Drag `basket.tscn` into the scene → position at `(400, 550)`
3. Add child: **Label** → rename to `ScoreLabel` → position at `(10, 10)` → text: `Score: 0`
4. Add child: **Timer** → rename to `SpawnTimer`
5. Save as `game.tscn`

Scene tree:

```
Game (Node2D)
  +-- Basket (basket.tscn instance, position: 400, 550)
  +-- ScoreLabel (Label, position: 10, 10, text: "Score: 0")
  +-- SpawnTimer (Timer)
```

The Timer Node

A **Timer** node emits a `timeout` signal at regular intervals:

- `wait_time` — how many seconds between each tick
- `start()` — begins the countdown
- `timeout` signal — fires every time the timer completes a cycle

With `wait_time = 1.0`: every second, the timer emits `timeout`. We connect that to a function that spawns a fruit.

Game Script

Select the `Game` node and attach a script:

```
extends Node2D

var fruit_scene = preload("res://fruit.tscn")                                  # load once at startup
var score = 0

func _ready():
    $SpawnTimer.wait_time = 1.0                                                 # spawn every 1 second
    $SpawnTimer.start()                                                         # begin the countdown
    $SpawnTimer.timeout.connect(_on_spawn_timer_timeout)                        # signal -> function

func _on_spawn_timer_timeout():
    var fruit = fruit_scene.instantiate()                                       # create a new fruit
    var screen_w = get_viewport_rect().size.x                                   # viewport width
    var margin = 50                                                             # edge padding
    fruit.position = Vector2(randf_range(margin, screen_w - margin), -20)  # random x, above screen
    add_child(fruit)                                                            # add to scene, starts falli

func add_score():                                                               # called by Basket on catch
    score = score + 1
    $ScoreLabel.text = "Score: " + str(score)                                   # update the UI
```

Code Walkthrough

`preload("res://fruit.tscn")` — Load the fruit scene once at script load time. You already know this pattern from Ricochet's `preload("res://logo.tscn")`.

`_ready()` — Runs once when the Game enters the scene: - Set the spawn timer to fire every 1 second - Start the timer - Connect the `timeout` signal to our handler function

`_on_spawn_timer_timeout()` — Called every second by the timer: - `fruit_scene.instantiate()` — create a new fruit (same as Ricochet's spawning) - `get_viewport_rect().size.x` — get the viewport width dynamically (no hardcoded screen size) - `Vector2(randf_range(margin, screen_w - margin), -20)` — random x position with padding, just above the screen - `add_child(fruit)` — add to the scene tree; the fruit's `_ready()` and `_process()` start immediately

`add_score()` — Called by the Basket when it catches a fruit: - Increment the score variable - Update the label text with `str(score)` to convert the number to a string

Two Ways to Connect Signals

We used two different methods to connect signals in this project:

Method	How	We Used It For
Editor	Node panel → Signals tab → double-click	Basket's `area_entered` signal
Code	`.connect(function_name)`	Timer's `timeout` signal

Both approaches do the same thing: connect a signal to a handler function. The editor approach is more visual; the code approach is more explicit. Use whichever you prefer.

---

7. Bonus: Fruit Variety

Random Colors and Speeds

Update the `_ready()` function in the **Fruit** script:

```
func _ready():
    add_to_group("fruits")                                          # tag for collision check
    $Sprite2D.modulate = Color(randf(), randf(), randf())           # random color tint
    speed = randf_range(150, 350)                                   # random fall speed
```

- `add_to_group("fruits")` — still needed for reliable collision detection
- `$Sprite2D.modulate` — the same color tinting trick from Ricochet's bonus section
- `randf_range(150, 350)` — each fruit gets a unique fall speed
- Since `_ready()` runs once per instance, each spawned fruit is unique

This creates visual variety and gameplay variation — some fruits fall faster and are harder to catch.

---

8. Complete Final Scripts

Basket Script (`basket.gd`)

```
extends Area2D

var speed = 400

func _process(delta):
    if Input.is_action_pressed("ui_left"):
```

```
        position.x = position.x - speed * delta
    if Input.is_action_pressed("ui_right"):
        position.x = position.x + speed * delta

    var half_w = $Sprite2D.texture.get_width() * $Sprite2D.scale.x / 2   # display half-width
    var screen_w = get_viewport_rect().size.x                            # viewport width
    position.x = clamp(position.x, half_w, screen_w - half_w)           # keep edges on screen

func _on_area_entered(area):
    if area.is_in_group("fruits"):                                       # is it a fruit?
        area.queue_free()                                                # destroy it
        get_parent().add_score()                                         # tell Game we scored
```

Fruit Script (fruit.gd)

```
extends Area2D

var speed = 200                                                          # pixels per second

func _ready():
    add_to_group("fruits")                                               # tag for collision check
    $Sprite2D.modulate = Color(randf(), randf(), randf())                # random color tint
    speed = randf_range(150, 350)                                        # random fall speed

func _process(delta):
    position.y = position.y + speed * delta                              # fall downward

    var half_h = $Sprite2D.texture.get_height() * $Sprite2D.scale.y / 2  # display half-height
    var screen_h = get_viewport_rect().size.y                            # viewport height
    if position.y > screen_h + half_h:                                   # fully below screen?
        queue_free()                                                     # destroy this fruit
```

Game Script (game.gd)

```
extends Node2D

var fruit_scene = preload("res://fruit.tscn")                            # load once at startup
var score = 0

func _ready():
    $SpawnTimer.wait_time = 1.0                                          # spawn every 1 second
    $SpawnTimer.start()                                                  # begin the countdown
    $SpawnTimer.timeout.connect(_on_spawn_timer_timeout)                 # signal -> function

func _on_spawn_timer_timeout():
    var fruit = fruit_scene.instantiate()                                # create a new fruit
    var screen_w = get_viewport_rect().size.x                            # viewport width
    var margin = 50                                                      # edge padding
    fruit.position = Vector2(randf_range(margin, screen_w - margin), -20)  # random x, above screen
    add_child(fruit)                                                     # add to scene, starts falli

func add_score():                                                        # called by Basket on catch
    score = score + 1
    $ScoreLabel.text = "Score: " + str(score)                            # update the UI
```

9. Summary of Concepts

Concept	What You Learned
Area2D	A node that detects when other areas overlap with it. Use for collectibles, triggers, damage zones.
CollisionShape2D	Defines the hitbox shape for an Area2D. Without it, no detection occurs.
$ shorthand	`$NodeName` is shorthand for `get_node("NodeName")` — access child nodes by name.
Sprite size	Use `texture.get_width() * scale.x` for actual display size. Raw texture size display size when scaled.
Groups	Tag nodes with `add_to_group()` and check with `is_in_group()`. Reliable alternative to name checking.
Signals	Godot's event system. Nodes emit signals; connected functions respond. Like a doorbell.
`area_entered`	Signal emitted by Area2D when another Area2D overlaps. Receives the other area as a parameter.
`queue_free()`	Removes a node from the scene tree and frees its memory. Used for cleanup.
Timer	A node that emits `timeout` at regular intervals. Perfect for spawning on a schedule.
`.connect()`	Connects a signal to a handler function in code. Alternative to connecting in the editor.
`get_parent()`	Returns the parent node. Used for upward communication in the scene tree.

10. Connection to Pong

This lecture laid the foundation for Pong's collision system:

Fruit Frenzy Concept	Pong Equivalent
Area2D on basket and fruit	Area2D on paddles and ball
`area_entered` detects catch	`area_entered` detects paddle hit
`queue_free()` destroys fruit	Ball resets to center after scoring
`get_parent().add_score()`	Score updates when ball passes paddle
Timer spawns fruits	Ball resets on a timer after each point

The next lecture combines **everything**: Wanderer's input (paddles), Ricochet's bounce physics (ball), and Fruit Frenzy's collision detection (paddle-ball interaction).

Exercises

1. **Speed Experiment:** Change the fruit's fall speed from 200 to 400, then to 100. What speed makes the game challenging but fair? Try different basket speeds too.

2. **Miss Counter:** Add a `misses` variable to the Game script. When a fruit is destroyed because it went off screen (not caught), have it tell the game to increment the miss count. Display `"Missed: X"` in a second label. *Hint: the fruit needs a reference to the game, similar to how the basket uses `get_parent()`.*

3. **Spawn Rate Increase:** Make fruits spawn faster over time. Every 10 seconds, reduce `$SpawnTimer.wait_time` by 0.1 (minimum 0.3 seconds). *Hint: add a second Timer with `wait_time = 10.0` and connect its `timeout` to a function that adjusts the spawn timer.*

4. **Fruit Types:** Create 3 different fruit scenes with different colors and point values. Red = 1 point, gold = 3 points, rainbow = 5 points. In the spawn function, randomly pick which type to instantiate. *Hint: store multiple preloaded scenes in an array and use `randi() % array.size()` to pick one.*

5. **Power-Up:** Every 15 seconds, spawn a special "growth" item. When caught, make the basket's CollisionShape2D 50% wider for 5 seconds, then restore it. *Hint: access the shape size with `$CollisionShape2D.shape.size` and use a Timer to revert.*

6. **Falling Bombs:** Alongside fruits, occasionally spawn red "bomb" Area2Ds. Catching a bomb subtracts 3 points from the score. *Hint: add bombs to a `"bombs"` group and check `area.is_in_group("bombs")` in `_on_area_entered`.*

7. **(Advanced) Game Over:** Implement a lives system. Start with 3 lives. Missing a fruit (it goes off screen) costs 1 life. At 0 lives, stop the spawn timer, display "Game Over" text, and wait for a key press to restart. *Hint: use `get_tree().reload_current_scene()` to restart the entire scene.*