Contents

CS 470 Game Development — Week 2, Lecture 4

Micro-Game 1: Wanderer

Goal: Build a character that moves with arrow keys. Along the way, learn the fundamental building blocks of every Godot game.

---

1. Nodes, Scenes, and the Scene Tree

Everything is a Node

Godot's entire architecture is built on one idea: **everything in your game is a Node**. A player, a background image, a sound effect, a timer — all nodes. Each node type has a specific job:

Node Type	What It Does
`Node`	Base type. Does nothing visible, but can hold logic and children.
`Node2D`	Adds a 2D **position**, rotation, and scale. The foundation of all 2D objects.
`Sprite2D`	Displays an image (texture) on screen.
`Area2D`	Detects overlaps with other objects (we'll use this next lecture).
`AudioStreamPlayer`	Plays sound.

There are hundreds of node types. You'll learn them gradually.

Scenes = Saved Node Trees

A **scene** is a tree of nodes that you save as a `.tscn` file. A scene can be anything: a player character, a bullet, an entire level, or a full game. The key insight is that scenes are **reusable** — you can drag a saved scene into another scene as many times as you want.

For Wanderer, our entire game is one scene:

```
Game (Node2D)          <- root of the scene
  +-- Player (Node2D)   <- has a position we can move
       +-- Sprite2D     <- displays the icon image
```

Why nest Sprite2D under Player instead of just using a Sprite2D directly? Because `Player` is a container. Later we might add a collision shape, a health bar, or particles under it. The Sprite2D is just one part of the player. The `Player` node's `position` controls where the whole group appears.

The Scene Tree (Runtime)

When you press Play, Godot loads your main scene and builds a **scene tree** — a live hierarchy of every node currently in the game. The engine walks this tree every frame, calling functions on each node. This is the heartbeat of your game.

---

2. The Coordinate System

Godot 2D uses a coordinate system that may surprise you if you're coming from math class:

```
(0, 0) ----------→ +X (right)
  |
  |
  |
  ↓
 +Y (down)
```

- **Origin (0, 0)** is the **top-left** corner of the screen.
- **+X** goes **right**.
- **+Y** goes **down** (not up!).

This is standard in almost all 2D game engines and graphics programming. The reason is historical: screens draw pixels top-to-bottom, left-to-right — so row 0 is the top.

So if your window is 1152×648 pixels:

Position	Where on Screen
(0, 0)	Top-left corner
(1152, 0)	Top-right corner
(0, 648)	Bottom-left corner
(576, 324)	Dead center

Implication for movement: To move a character *up* on screen, you **subtract** from Y. To move *down*, you **add** to Y.

---

3. Quick Vector Refresher

A `Vector2` in Godot is simply a pair of numbers `(x, y)`. It can represent a position, a direction, a velocity — whatever you need.

Key Operations

```
Addition:        (2, 3) + (1, -1) = (3, 2)         → combining movement
Scalar multiply: (1, 0) * 5       = (5, 0)         → scaling (speed)
Length:          |(3, 4)|         = 5             → distance from origin
Normalized:      (3, 4).normalized() = (0.6, 0.8) → same direction, length 1
```

Why Normalize?

Suppose the player presses **right + down** simultaneously. Without normalizing:

```
direction = (1, 0) + (0, 1) = (1, 1)
length of (1, 1) = sqrt(2) ~= 1.414
```

The player moves ~41% faster diagonally! That's unfair. Normalizing squashes it back to length 1:

```
(1, 1).normalized() = (0.707, 0.707)
length = 1.0  << correct
```

Now all directions produce the same speed.

---

4. GDScript — Quick Comparison with Python

GDScript is Godot's built-in scripting language. If you know Python, you're 80% there.

Similarities

- Indentation-based (no curly braces)
- Dynamic-feeling syntax
- `for`, `if`, `while` work the same way
- `print()` works the same way

Key Differences

Python	GDScript	Notes
`class MyClass:`	`extends Node2D`	Every script *is* a class that extends a node type
`self.x`	`x` or `position`	No `self` keyword — properties of the node are accessed directly
`def __init__(self):`	`func _ready():`	Called once when the node enters the scene tree
`def update(self, dt):`	`func _process(delta):`	Called every frame by the engine
`x: int = 5`	`var x: int = 5`	Variables declared with `var`
`def foo(self):`	`func foo():`	Functions declared with `func`

Where's `self`?

In Python, every method carries an explicit `self` parameter. In GDScript, the node *is* the implicit context. When you write `position` inside a script attached to a Node2D, it means "this node's position" — as if Python had automatically resolved `self.position` for you.

```
# GDScript - these are equivalent:
position.x += 5
self.position.x += 5    # works, but nobody writes this
```

5. Scripts and How They Run

Attaching a Script

When you attach a script to a node, you're giving that node custom behavior. The script file (`.gd`) is a class definition that **extends** the node's type:

```
extends Node2D    # "I am a Node2D with extra behavior"
```

You can only attach one script per node. The script has access to all of that node's built-in properties (like `position`, `rotation`, `visible`, etc.).

Behind the Scenes — The Game Loop

Every game engine runs a **game loop** — an infinite loop that drives the entire game:

```
while game_is_running:
    process_input()
    update_all_objects()     ← your _process() functions run here
    render_frame()
    wait_for_vsync()         ← targets ~60 FPS
```

Each iteration of this loop = **one frame**. At ~60 FPS, that's roughly every 16.6 milliseconds.

During the "update all objects" phase, Godot walks the **entire scene tree** top-to-bottom and calls `_process(delta)` on every node that has a script with that function defined. No node is skipped — the engine doesn't care if there's "nothing to do." If `_process` exists, it runs.

```
Frame 1:  Game._process() → Player._process() → (Sprite2D has no script, skip)
Frame 2:  Game._process() → Player._process() → ...
Frame 3:  Game._process() → Player._process() → ...
...60 times per second
```

The Special Functions

Function	When It Runs	Use It For
`_ready()`	**Once**, when the node first enters the scene tree	Initialization: set starting values, cache references
`_process(delta)`	**Every frame** (~60 times/sec)	Game logic: movement, animation, checking conditions
`_input(event)`	**Every input event** (key press, mouse click, etc.)	Responding to discrete input events

There are others (`_physics_process`, `_enter_tree`, etc.) — we'll meet them later.

What is `position`?

Every `Node2D` has a built-in property called `position` — a `Vector2` representing where the node sits **relative to its parent**. When you write:

```
position.x += 5
```

…you're moving this node 5 pixels to the right of where its parent is. Since our `Player`'s parent is `Game` (which sits at the origin), `Player.position` is effectively the screen position.

---

6. Understanding `delta`

The Problem

Suppose you write:

```
func _process(delta):
    position.x += 5     # move 5 pixels per frame
```

At 60 FPS → 300 pixels/sec. At 30 FPS (maybe the computer is struggling) → 150 pixels/sec. **The game speed depends on the framerate.** That's bad.

The Solution: `delta`

`delta` is the **time in seconds since the last frame**. At 60 FPS, delta ≈ 0.0166. At 30 FPS, delta ≈ 0.0333.

```
func _process(delta):
    position.x += 300 * delta     # move 300 pixels per SECOND, regardless of FPS
```

At 60 FPS: 300 × 0.0166 = 5.0 pixels/frame → 300 px/sec ✓ At 30 FPS: 300 × 0.0333 = 10.0 pixels/frame → 300 px/sec ✓

Rule: Always multiply movement (and any time-based value) by `delta`.

Mini-Exercise: Print After 3 Seconds

Use delta to accumulate elapsed time:

```
extends Node2D

var timer = 0.0
var message_printed = false
```

```
func _process(delta):
    timer += delta
    if timer >= 3.0 and not message_printed:
        print("3 seconds have passed!")
        message_printed = true
```

This runs every frame, adding the tiny time slice to `timer`. After ~180 frames (3 seconds × 60 FPS), the total crosses 3.0 and the message prints once.

> **Key takeaway:** `delta` turns the discrete world of frames into smooth, continuous time.

---

7. How Input Works (Without Blocking)

The Wrong Mental Model (Blocking)

In a console Python program, you might write:

```
key = input("Press a key: ")    # BLOCKS – program freezes here
```

The entire program halts until the user types something. **Games can never do this** — if the game froze waiting for input, nothing would move, animate, or render.

The Right Mental Model (Polling)

Godot checks the state of all keys/buttons **every frame** and stores that snapshot. Your code simply *asks* about the current state:

```
if Input.is_action_pressed("ui_right"):
    # Right arrow is currently held down - this frame
```

This is called **polling**. It doesn't wait. It doesn't block. It just checks: "Is this key down *right now*?" and returns `true` or `false` instantly.

`is_action_pressed` vs `is_action_just_pressed`

Function	Returns `true` when…
`is_action_pressed("ui_right")`	The key is **held down** (true every frame it's held)
`is_action_just_pressed("ui_right")`	The key was **just pressed this frame** (true for one frame only)

For movement, we want `is_action_pressed` — the character should keep moving as long as the key is held.

What are "Actions"?

Godot maps physical keys to named **actions**. Some are built-in:

Action Name	Default Key
`"ui_right"`	→ (Right Arrow)
`"ui_left"`	← (Left Arrow)
`"ui_up"`	↑ (Up Arrow)
`"ui_down"`	↓ (Down Arrow)
`"ui_accept"`	Enter / Space

You can define custom actions in **Project → Project Settings → Input Map** (we'll do this for Pong).

---

8. Building Wanderer — Step by Step

Step 1: Create the Project

1. Open Godot 4 → **New Project**
2. Name: `Wanderer` — pick a folder — click **Create & Edit**

Step 2: Build the Scene Tree

1. In the Scene panel (top-left), click **+ Other Node** → search `Node2D` → Create
2. Rename it to `Game` (double-click the name in the scene tree)
3. Select `Game` → click the **+** button (or right-click → Add Child Node) → `Node2D` → rename to `Player`
4. Select `Player` → Add Child Node → `Sprite2D`
5. Select the `Sprite2D` → in the Inspector (right panel), find **Texture** → drag `icon.svg` from the FileSystem panel (bottom-left) into the Texture slot

You should now see the Godot icon in the viewport, and your scene tree reads:

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

6. Select `Player` → in the Inspector, set **Transform → Position** to (576, 324) to center it
7. **Ctrl+S** → save as `game.tscn`

Step 3: Attach a Script to Player

1. Select the `Player` node
2. Click the scroll icon at the top of the Scene panel (or right-click → **Attach Script**)
3. Keep all defaults → click **Create**
4. The script editor opens with a nearly empty file. Replace everything with:

```
extends Node2D

var speed = 300.0

func _process(delta):
    var direction = Vector2.ZERO    # (0, 0) - no movement yet

    if Input.is_action_pressed("ui_right"):
        direction.x += 1
    if Input.is_action_pressed("ui_left"):
        direction.x -= 1
    if Input.is_action_pressed("ui_down"):
        direction.y += 1            # remember: +Y is DOWN
    if Input.is_action_pressed("ui_up"):
        direction.y -= 1            # -Y is UP

    if direction.length() > 0:
        direction = direction.normalized()  # fix diagonal speed

    position += direction * speed * delta
```

5. **Ctrl+S** to save the script.

Step 4: Run It

1. Press **F5** (or the Play ▶ button, top-right)

2. Godot asks for a main scene → select `game.tscn`
3. Use arrow keys — the Godot icon moves around!

Walkthrough of the Code

Let's trace through what happens **every single frame** (roughly 60 times per second):

1. `direction` starts as `(0, 0)` — no movement.
2. Each `if` check **adds** to the direction vector. Pressing Right sets it to `(1, 0)`. Pressing Right + Up sets it to `(1, -1)`.
3. If any key was pressed, `direction.length() > 0` is true, so we normalize it to length 1.
4. `direction * speed * delta` = a small movement vector. For example, pressing right at 60 FPS: `(1, 0) * 300 * 0.0166 = (4.98, 0)` — about 5 pixels this frame.
5. `position +=` moves the Player node by that amount.

Notice: We build a direction vector *first*, normalize it *once*, then apply speed and delta. This is cleaner than handling each key independently and is a pattern you'll use in almost every game.

---

9. Getting Screen Size and Clamping

The Problem

Run the game and hold Right — the character disappears off the edge. There are no walls. We need to **clamp** (restrict) the position to stay within the screen.

Getting the Screen Size

```
var screen_size = get_viewport_rect().size
# Returns a Vector2, e.g. (1152, 648)
```

`get_viewport_rect()` returns the rectangle of the current viewport. `.size` gives its width and height as a Vector2.

The `clamp()` Function

`clamp(value, min, max)` forces a number to stay within a range:

```
clamp(500, 0, 1152)  → 500   (already in range)
clamp(-20, 0, 1152)  → 0     (was below min, clamped up)
clamp(1200, 0, 1152) → 1152  (was above max, clamped down)
```

Adding Clamping to Wanderer

Add these two lines at the **end** of `_process`, after the `position +=` line:

```
    var screen_size = get_viewport_rect().size
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)
```

The full final script:

```
extends Node2D

var speed = 300.0

func _process(delta):
    var direction = Vector2.ZERO
```

```
    if Input.is_action_pressed("ui_right"):
        direction.x += 1
    if Input.is_action_pressed("ui_left"):
        direction.x -= 1
    if Input.is_action_pressed("ui_down"):
        direction.y += 1
    if Input.is_action_pressed("ui_up"):
        direction.y -= 1

    if direction.length() > 0:
        direction = direction.normalized()

    position += direction * speed * delta

    # Keep player inside the screen
    var screen_size = get_viewport_rect().size
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)
```

Run it again — the icon now stops at the edges.

---

10. Summary of Concepts

Concept	What You Learned
Nodes	Building blocks of everything in Godot. Each type has a specific role.
Scenes	A saved tree of nodes (`.tscn` file). Reusable and nestable.
Scene Tree	The live hierarchy at runtime. Engine walks it every frame.
Coordinate System	Origin = top-left. +X = right. **+Y = down.**
extends	GDScript's version of class inheritance. Replaces Python's `class Foo(Bar):`
No self	Node properties like `position` are accessed directly.
_ready()	Runs once when the node enters the tree.
_process(delta)	Runs every frame. The heartbeat of your game logic.
delta	Seconds since last frame. Multiply by it for frame-rate independence.
position	A `Vector2` — where this node is relative to its parent.
Input polling	`Input.is_action_pressed()` — non-blocking, checked every frame.
Vector2	A pair `(x, y)`. Can represent position, direction, or velocity.
normalized()	Scales a vector to length 1. Prevents faster diagonal movement.
clamp()	Restricts a value to a min/max range. Keeps objects on screen.
get_viewport_rect().size	Returns the screen dimensions as a `Vector2`.

Exercises

1. **Speed Experiment:** Change `speed` to 600, then 100. Feel the difference. What speed feels "right"?

2. **Delta Timer:** Add a variable that accumulates `delta` each frame. Print `"Hello!"` every 2 seconds (not just once — reset the timer after each print).

3. **Boundary Padding:** The icon's *center* stops at the edge, but half the image still goes off-screen. Modify the clamp to account for the icon's size (the default icon is 128×128, so the "radius" is 64 pixels). *Hint: clamp between `64` and `screen_size.x - 64`.*

4. **Wrap Instead of Clamp:** Instead of stopping at edges, make the player **wrap around** — exit the right side, appear on the left. *Hint: use `if` statements instead of `clamp`, and set position to the opposite side.*

5. **Easy: Sprint Key:** Add a sprint feature that doubles the speed when holding Shift. *Hint: check `Input.is_key_pressed(KEY_SHIFT)` and adjust the speed multiplier accordingly before applying movement.*

6. **Mouse Follower:** Make the icon move toward the mouse cursor position instead of using keyboard input. The icon should move at constant speed toward wherever the mouse is. *Hint: use `get_viewport().get_mouse_position()` to get the cursor location, calculate the direction vector (`mouse_pos - position`), normalize it, and apply speed.*

7. **(Bonus): Smooth Rotation:** Make the icon smoothly rotate to face the direction it's moving. *Hint: calculate the target angle using `direction.angle()` (or `velocity.angle()` if you store velocity), then use `rotation = lerp_angle(rotation, target_angle, 5.0 * delta)` to smoothly interpolate each frame. Only rotate when actually moving (`direction.length() > 0`).*