Collect the Coins!

A step-by-step 2D game development tutorial in Godot



What you will build: A small 2D game where the player moves around, collects coins, earns points, completes a timed round, and keeps a high score between sessions.

This tutorial uses Godot 4.7 and GDScript. Menu labels can vary slightly by Godot version. Follow the node names and scene paths consistently; the code examples assume the structure described here.

From Game Design to Development

Before building the game, consider how its design determines the features and systems we need to develop.

Game Concept: *Collect the Coins!* is a 2D collection game in which the player controls a character moving around a grassy environment to collect as many coins as possible before time runs out.

Game Objective: Collect as many coins as possible within a 60-second round to earn the highest score.

Core Gameplay Loop

- The **player moves around the gameplay area**.
- **Coins appear at different locations.**
- The **player collects coins to increase their score.**
- **Uncollected coins disappear after 8 seconds.**
- **New coins continue to appear every 3 seconds.**
- When the **60-second timer reaches zero**, the **round ends** and the **final score is displayed.**
- The player can start **another round** and try to beat their high score.

From Design to Development

Game Design Element	Development Implementation
Player movement	**CharacterBody2D and player movement script**
Collectible objects	**Reusable Coin scene with collision detection**
Scoring	**GameManager and score display**
Time pressure	**Countdown timer**
Coin availability	**Coin spawning and expiration timers**
Game feedback	**Score display and GameOver screen**
Replayability	**Play Again button and persistent high score**
Navigation	**Main Menu, Instructions, and Scores screens**

As you work through the tutorial, you will turn each design element into a working feature using Godot scenes, nodes, scripts, signals, and timers.

The goal is not just to make the game run, but to understand how the game's design is translated into its technical implementation.

Before You Begin

In the Unity labs, you practiced player input, movement, collision detection, scoring, and feedback. This lab applies those same game development ideas in Godot while introducing reusable scenes, signals, scene transitions, and persistent data.

Provided assets

Asset preview	Asset	Purpose
	Grass background	Fills the gameplay area behind the player and coins.
	Collectible coin	The player collects these to increase the score.
	Bear sprite sheet	Contains front, back, and side-facing frames for directional movement animations.
	Title Background	Provides the background artwork for the Main Menu and creates the visual setting for the game's title screen.

Prepare the project

- Create a **new Godot project** named **CollectTheCoins**.
- In the New Project window, set the **Renderer** to **Compatibility**. We will use this renderer because the assignment is a 2D game and Compatibility supports exporting games for the web (including itch.io).
- Click **Create** to create and open your project.
- In the **FileSystem** dock, create three folders:
    - **Scenes**
    - **Scripts**
    - **Assets**
- Download the **four provided PNG assets** and place them inside the **Assets** folder:
    - **Grass background**
    - **Bear Sprite Sheet**
    - **Collectible coin**
    - **Title background**
- As you work, save your scenes inside the **Scenes** folder and your scripts inside the **Scripts** folder. Keep spelling and capitalization consistent with the examples throughout this tutorial.

Project paths res:// refers to the root of your Godot project.

For example, res://Scenes/Game.tscn points to the Game.tscn file inside the Scenes folder.

1. Understand Nodes and Scenes

Godot games are built from nodes arranged in a scene tree.

A scene is a saved collection of nodes. It can represent a whole screen or a reusable object.

We will create the following scenes:

- **MainMenu.tscn**: title screen, start button, instructions button, and scores button
- **Game.tscn**: player, coins, and score display
- **Player.tscn**: reusable player character
- **Coin.tscn**: reusable collectible
- **GameOver.tscn**: final score and restart options
- **Instructions.tscn**: screen explaining the game objective and controls
- **Scores.tscn**: screen displaying the most recent round score and saved high score

Our scenes will be comprised of the following nodes:

Godot node	Typical purpose
CharacterBody2D	Player body controlled by movement code
AnimatedSprite2D	Displays animation frames from a SpriteFrames resource
Sprite2D	Displays a single image or texture
Area2D	Detects when another body enters an area
CollisionShape2D	Defines the collision area/shape
Control, Label, Button	Build menus and user-interface elements

2. Create the Gameplay Background

- Open **Game.tscn** or create a **new 2D scene** and save it as **Scenes/Game.tscn**.
- Add a **Node2D root** named **Game**.
- Add a **Sprite2D child** named **Background**.

- Select **Background** and assign **Assets/grass_background.png** to its **Texture** property.
- Position the background **behind gameplay objects** in the scene tree. If needed, **adjust its scale or project window size to cover the playable area**.

Check your work: The grass image should be visible and the player and coins should appear in front of it.

3. Create the Player and Add Movement

Step A — Build Player.tscn

- Create a **new 2D scene** with **CharacterBody2D** as the root. Name it **Player**.
- Add an **AnimatedSprite2D child** named **AnimatedSprite2D**.
- Add a **CollisionShape2D child** and choose a suitable shape in the Inspector.
- Save the scene as **Scenes/Player.tscn**.
- In the Node dock's **Groups tab**, add the **root Player node** to a group named **player**.

Step B — What is a sprite sheet?

A sprite sheet is a single image containing multiple frames or poses for a character. Godot can slice the image into individual cells so you can place frames into named animations.

A 4 × 4 sheet has four columns and four rows, for 16 cells total. The grid does not automatically decide which cells belong to which animation; you select the frames yourself.

Asset reference: The supplied bear sheet has 16 cells arranged in four columns and four rows. Use the image preview in the asset table to identify the facing direction in each frame.

Step C — Import and Animate the Bear Sprite Sheet

- Select the **AnimatedSprite2D** node in the Player scene.
- In the **Inspector**, locate the **Sprite Frames** property. Click the empty resource field and select **New SpriteFrames**.

- Click the newly created **SpriteFrames resource** to open the SpriteFrames animation editor in the bottom panel.
- In the animation list, rename the default animation to **walk_down**.
- Click **Add frames from a Sprite Sheet** and select **Assets/bear_spritesheet.png**.
- In the sprite sheet selection window, configure the grid:
- **Horizontal:** 4 columns
- **Vertical:** 4 rows
- Select the four frames in the **first row** (the bear facing downward). Add the selected frames to the **walk_down** animation.
- Repeat the process to create the remaining directional animations. Select the appropriate row for each direction:
- **walk_up:** Row 2 (bear facing away from the camera)
- **walk_left:** Row 3 (bear facing left)
- **walk_right:** Row 4 (bear facing right)
- Create an **idle** animation using one standing frame. A single frame is sufficient for a simple idle pose.
- Select each animation and use the Play button to preview it. Start with an animation speed of approximately **6–8 FPS**. Adjust the speed if the movement looks too fast or too slow.

Inspect the frames: Before continuing, preview each animation and check that the selected frames form a consistent walking cycle.

If a direction does not have a smooth sequence, use the clearest available frames or simplify that animation to a single pose.

The player should still move correctly even if an animation needs to be simplified.

Step D — Add PlayerMovement.gd

Attach a new GDScript named PlayerMovement.gd to the Player CharacterBody2D. The node name AnimatedSprite2D and animation names in the code must match your scene.

```
extends CharacterBody2D

# Allows the movement speed to be adjusted in the Inspector
@export var speed: float = 250.0
@onready var animated_sprite: AnimatedSprite2D =
$AnimatedSprite2D

func _physics_process(_delta: float) -> void:
    # Get movement input as a direction vector (left, right, up, down)
    var direction := Input.get_vector(
        "ui_left", "ui_right", "ui_up", "ui_down"
    )

    # Set movement speed and move the Player while handling collisions
    velocity = direction * speed
    move_and_slide()

    # Choose an animation based on the Player's movement direction
    if direction == Vector2.ZERO:
        animated_sprite.play("idle")
    elif abs(direction.x) > abs(direction.y):
        if direction.x > 0:
            animated_sprite.play("walk_right")
        else:
            animated_sprite.play("walk_left")
    elif direction.y > 0:
        animated_sprite.play("walk_down")
    else:
        animated_sprite.play("walk_up")
```

How the script works:

- **Input.get_vector()** reads the four built-in directional actions and produces a **direction vector**.
- **velocity** sets the **movement direction** and **speed**; **move_and_slide() moves the CharacterBody2D.**
- The script selects a directional animation based on the strongest axis of movement.
- When there is no input, the script plays idle.

If you use WASD: The example uses Godot's built-in ui_left, ui_right, ui_up, and ui_down actions. To use WASD, configure additional actions in Project Settings → Input Map and change the action names in the script.

Step E — Test the player

- Save the script and scene.
- **Add the Player scene to Game.tscn.**
    - Open **Game.tscn**. In the Scene dock, click the **Instantiate Child Scene** button. Navigate to the Scenes folder, select Player.tscn, and click **Open**. The Player scene will appear as a child of the Game root node in the Scene tree.
- Run the gameplay scene and move with the arrow keys.
- Confirm the bear moves and plays a walking animation while moving; confirm it returns to idle when stopped.
- Scale the player if necessary.
- Adjust the collision shape so it roughly matches the player's visible area.

4. Create Coins and Detect Collection

Step A — Build Coin.tscn

- Create a **new scene** with **Area2D as the root** and name it **Coin**.
- Add a **Sprite2D child** and assign **Assets/coin.png** to its **Texture property**.
- Add a **CollisionShape2D child** and choose a **CircleShape2D** or another suitable shape.

- Adjust the collision shape so it roughly matches the coin's visible area.
- Save the scene as **Scenes/Coin.tscn**.

Step B — Connect body_entered

- Attach a **script** to the **Coin root**.
- Select **Coin**, open the Node dock, find the **body_entered signal**, and **connect it to the coin script**.
- Use the generated signal method to handle collection.

```
extends Area2D # Allows this script to detect when another physics body enters the coin's area

# Tracks whether the coin has already been collected
var collected: bool = false

func _ready() -> void:

    # Starts the timer when the coin is created
    $LifetimeTimer.start()

func _on_body_entered(body: Node2D) -> void:
    # Stops the function if the coin has already been collected

     if collected:
         return

    # Checks whether the object that entered the area belongs to the player group.
    if body.is_in_group("player"):
        collected = true # Marks the coin as collected to prevent duplicate collection
        GameManager.collect_coin() # Calls the GameManager function to increase the player's score
        queue_free() # Removes the coin from the scene
```

```
# Checks whether the coin has not been collected before the timer runs out
func _on_lifetime_timer_timeout() -> void:
    if not collected:
        queue_free() # Removes the coin from the scene when its lifetime expires
```

- The **body_entered signal supplies the body that entered the area**.
- The **player group check prevents other physics bodies from collecting coins**.
- **queue_free() removes the coin from play**.

Step C — Add coin lifetime and spawning

- In **Coin.tscn**, add a **Timer child** named **LifetimeTimer**.
- Set **Wait Time** to **8.0 seconds**, **enable One Shot**, and leave **Autostart disabled**. The coin script starts this timer when the coin appears.
- Connect **LifetimeTimer's timeout signal** to **_on_lifetime_timer_timeout()** in the coin script. The coin should disappear when the timer expires if it has not been collected.
- In **Game.tscn**, add a **Node2D child named SpawnPoints**. Add at least **four Marker2D children** and **position them at different safe locations inside the gameplay area**. These markers define where coins may appear.

Do not instance coins manually. Game.gd will create Coin.tscn instances at randomly selected spawn points during the round.

Check your work: Confirm that the Coin scene has a working collision shape and LifetimeTimer, and that SpawnPoints contains at least four Marker2D nodes. You will implement and test coin spawning in Section 5.

5. Manage Score, Coin Spawning, and Round State

- The game runs for **60 seconds**. A **new coin appears every 3 seconds at a randomly selected spawn point**. Each coin **remains available for 8 seconds unless collected first.** The round **ends when the round timer reaches zero.**

Step A — Create GameManager.gd

- Create **Scripts/GameManager.gd** using Node as the base class.
- Add the following code:

```
extends Node # Allows this script to function as a general-purpose node.


# Signals notify other scripts when the score changes or the round is completed.
signal score_changed(new_score: int)
signal round_completed


# Stores the player's current score, final score, and saved high score.
var score: int = 0
var final_score: int = 0
var high_score: int = 0


# Tracks whether the current round has ended.
var round_finished: bool = false


# Defines the location where the high-score data will be saved.
# The user:// path points to Godot's application data directory.
const SAVE_PATH = "user://save_data.json"
```

```
func _ready() -> void:
    # Loads the previously saved high score when the GameManager starts.
    load_high_score()


func start_round() -> void:
    # Resets the scores and round status when a new round begins.
    score = 0
    final_score = 0
    round_finished = false

    # Sends a signal to notify other scripts that the score has changed.
    score_changed.emit(score)


func collect_coin() -> void:
    # Prevents the score from increasing after the round has ended.
    if round_finished:
        return

    # Increases the player's score by one for each collected coin.
    score += 1

    # Notifies other scripts so the score display can be updated.
    score_changed.emit(score)
```

```
func finish_round() -> void:
    # Prevents the round from being completed more than once.
    if round_finished:
        return

    # Marks the round as finished and stores the player's final score.
    round_finished = true
    final_score = score

    # Checks whether the player has earned a new high score.
    if final_score > high_score:
        high_score = final_score
        save_high_score() # Saves the new high score to the file.

    # Notifies other scripts that the round has ended.
    round_completed.emit()

func save_high_score() -> void:
    # Opens the save file in WRITE mode (creates or overwrites the file).
    var file = FileAccess.open(SAVE_PATH, FileAccess.WRITE)
```

```
    # Checks whether the file was successfully opened.
    if file == null:
        push_error("Could not open the save file.")
        return

    # Stores the high score in a Dictionary.
    var data = {"high_score": high_score}

    # Converts the Dictionary into a JSON string and writes it to the file.
    file.store_string(JSON.stringify(data))

func load_high_score() -> void:
    # Initializes the high score to zero before attempting to load saved data.
    high_score = 0

    # Checks whether a save file exists. If not, the high score remains zero.
    if not FileAccess.file_exists(SAVE_PATH):
        return

    # Opens the existing save file in READ mode.
    var file = FileAccess.open(SAVE_PATH, FileAccess.READ)
```

```
# Stops if the file could not be opened.
if file == null:
    return


# Reads the file contents and converts the JSON string into a Godot value.
var data = JSON.parse_string(file.get_as_text())


# Checks that the loaded data is a Dictionary before accessing its contents.
if data is Dictionary:
    # Retrieves the saved high score. Uses zero if the key is missing.
    var saved_score = data.get("high_score", 0)


    # Checks that the saved value is a non-negative floating-point number.
    if saved_score is float and saved_score >= 0:
        high_score = int(saved_score) # Converts saved value back to an integer.
```

- **Important:** JSON uses a **general** number type and does **not distinguish between integers and floating-point numbers.** When Godot's JSON.parse_string() reads numerical values from a JSON file, it **converts them into floating-point numbers (float), even if they were originally stored as integers.** Therefore, **check that the loaded value is a non-negative float, then use int() to convert it back to an integer before assigning it to the high_score variable.**
- This **Autoload** keeps **the current score**, **final score**, and **high score** accessible across scene changes.

Step B — Register the Autoload

- Open Project Settings and find the **Globals/Autoload** section.
- Choose **Scripts/GameManager.gd**, register it with the name **GameManager**, and enable it.

Step C — Set up Game.tscn, timers, and the score label

- In **Game.tscn**, make sure the scene tree contains **Game**, **Background**, **Player**, **SpawnPoints**. **SpawnPoints** should contain the **Marker2D nodes** you created in Section 4.
- Add a **Label node** named **ScoreLabel** as a **child of Game**. Position it near the bottom of the screen and set its **initial text to Score: 0**.
- Add another **Label** named **TimeLabel**, position it near the bottom of the screen, and set its **initial text to Time: 60**. The Game.gd script will update both labels during gameplay.
- Add **two Timer nodes** as **children of Game: SpawnTimer** and **RoundTimer.**
- Set **SpawnTimer** to a **Wait Time of 3.0 seconds** and leave **One Shot disabled** so it repeats.
- Set **RoundTimer** to a **Wait Time of 60.0 seconds** and **enable One Shot** so it runs only once.
- Connect **SpawnTimer's timeout signal** to **_on_spawn_timer_timeout()** on the Game root.
- Connect **RoundTimer's timeout signal** to **_on_round_timer_timeout()** on the Game root. The script starts both timers when gameplay begins and spawns the first coin immediately; **SpawnTimer creates additional coins every three seconds.**
- The script **chooses a random Marker2D under SpawnPoints each time it creates a coin**. Add more markers or reposition them to change where coins can appear. Keep all markers inside the playable area and away from locations the player cannot reach.

Node names matter: The script uses the exact paths $ScoreLabel, $TimeLabel, $SpawnPoints, $SpawnTimer, and $RoundTimer.

These nodes **must be direct children of Game.**

If you rename any of them or place them elsewhere in the scene tree, update the corresponding paths in Game.gd.

Step D — Add the Game.gd Script

- Attach a new GDScript named **Game.gd** to the **Game root node**. This script updates the score and countdown labels, spawns coins, and ends the round when the timer runs out. Add the following code:

extends Node2D # Makes this script a 2D node that manages the main gameplay scene.

Preloads the Coin scene so new coin instances can be created during gameplay.

const COIN_SCENE = preload("res://Scenes/Coin.tscn")

Gets references to existing nodes in the Game scene when this node is ready.

@onready var score_label: Label = $ScoreLabel # Displays the player's current score.

@onready var time_label: Label = $TimeLabel # Displays the remaining time.

@onready var spawn_points: Node2D = $SpawnPoints # Contains the possible coin spawn locations.

@onready var spawn_timer: Timer = $SpawnTimer # Controls how often new coins appear.

@onready var round_timer: Timer = $RoundTimer # Tracks the total time remaining in the round.

Connects GameManager signals to functions in this script.

func _ready() -> void:

    # These functions run automatically when the corresponding signals are emitted.

    GameManager.score_changed.connect(_on_score_changed)

    GameManager.round_completed.connect(_on_round_completed)

    # Resets the score and round status to begin a new round.

    GameManager.start_round()

```
    # Updates the score and timer displays when the scene starts.
    _on_score_changed(GameManager.score)
    _update_time_label()

    # Creates the first coin and starts both timers.
    _spawn_coin()
    spawn_timer.start()
    round_timer.start()

func _process(_delta: float) -> void:
    # Updates the countdown display continuously while the game is running.
    _update_time_label()

func _update_time_label() -> void:
    # Displays the remaining time as a whole number, rounding up.
    time_label.text = "Time: " + str(ceili(round_timer.time_left))

func _spawn_coin() -> void:
    # Prevents additional coins from spawning after the round ends.
    if GameManager.round_finished:
        return
```

```
    # Gets a list of all child nodes under SpawnPoints.
    var points = spawn_points.get_children()

    # Checks whether any spawn points have been added to the scene.
    if points.is_empty():
        push_warning("Add Marker2D nodes under SpawnPoints.")
        return

    # Creates a new instance of the preloaded Coin scene.
    var coin = COIN_SCENE.instantiate()
    add_child(coin) # Adds the coin to the Game scene.

    # Randomly selects one of the available spawn points.
    var chosen_point = points.pick_random()

    # Positions the new coin at the selected spawn point.
    coin.global_position = chosen_point.global_position

func _on_spawn_timer_timeout() -> void:
    # Creates another coin whenever the spawn timer runs out.
    _spawn_coin()
```

```
func _on_round_timer_timeout() -> void:
    # Ends the round when the countdown timer reaches zero.
    GameManager.finish_round()

func _on_score_changed(new_score: int) -> void:
    # Updates the score label whenever the player's score changes.
    score_label.text = "Score: " + str(new_score)

func _on_round_completed() -> void:
    # Stops both timers when the round is finished.
    spawn_timer.stop()
    round_timer.stop()

    # Changes the current scene to the GameOver screen.
    get_tree().change_scene_to_file("res://Scenes/GameOver.tscn")
```

Step E — Create the GameOver Screen

The **GameOver screen appears when the 60-second countdown ends**.

It displays the player's final score and provides options to play another round or return to the main menu.

1. Build GameOver.tscn

- Create a **new scene** with a **Control node as the root**. Rename it **GameOver**.
- Save the scene as **Scenes/GameOver.tscn**.
- Add a **Label child** named **GameOverLabel**. Set its text to **Round Complete!**.

- Add another **Label child** named **FinalScoreLabel**. This will display the player's final score.
- Add a **Button child** named **PlayAgainButton**. Set its text to **Play Again**.
- Add another **Button child** named **MainMenuButton**. Set its text to **Main Menu**.
- Arrange the labels and buttons so they are centered and easy to read.

2. Add the GameOver.gd Script

- Attach a new GDScript named **GameOver.gd** to the **GameOver root node**. This script displays the final score and handles the two buttons.

```
extends Control # Makes this script a user interface (UI) control node.

@onready var final_score_label: Label = $FinalScoreLabel

func _ready() -> void:
    # Displays the score earned during the completed round.
    final_score_label.text = "Final Score: " + str(GameManager.final_score)

func _on_play_again_button_pressed() -> void:
    # Starts a new round by returning to the gameplay scene.
    get_tree().change_scene_to_file("res://Scenes/Game.tscn")

func _on_main_menu_button_pressed() -> void:
    # Returns the player to the main menu.
    get_tree().change_scene_to_file("res://Scenes/MainMenu.tscn")
```

3. Connect the Buttons

- Connect each button's pressed signal to the GameOver root node:
    - Select **PlayAgainButton**, open the **Node dock**, and connect its **pressed signal** to **_on_play_again_button_pressed()**.
    - Select **MainMenuButton** and connect its **pressed signal** to **_on_main_menu_button_pressed()**.
- Make sure the generated function names match the names in the script.

Check your work: The GameOver screen should display the final score from the completed round.

Clicking Play Again should return to Game.tscn and begin a new round with the score reset to zero.

Clicking Main Menu should return to MainMenu.tscn.

Step F — Test the Gameplay

- Run **Game.tscn** and check that the **score begins at zero**, the countdown begins at **60 seconds**, and the **first coin appears immediately**.
- **Additional coins should appear every 3 seconds** at randomly selected spawn points.
- **Collected coins should disappear and increase the score by 1.**
- **Uncollected coins should disappear after 8 seconds.**
- When the **countdown reaches zero**, the **results screen should appear**.

6. Create the Title Screen

Step A — Build MainMenu.tscn

- Create a **new scene with Control** as the root node. Rename the root **MainMenu**.
- Save the scene as **Scenes/MainMenu.tscn**.
- Add a **TextureRect child** named **Background** to the **MainMenu node**.
- Assign the desired **title screen background image** to its **Texture property**.
- Use the 2D viewport to resize and position the image so it fills the game window.
- Make sure Background is behind the labels and buttons in the scene tree.

Step B — Add the Menu UI

- Add three Button nodes as children of the MainMenu node.
- Rename and configure them as follows:

Node	Type	Displayed Text
StartButton	Button	Start Game
InstructionsButton	Button	Instructions
ScoresButton	Button	Scores

- Use the 2D viewport to resize and position the buttons in the open area below the bear.
- Arrange the buttons vertically, leaving enough space between them so they are easy to select.

Step C — Create MainMenu.gd

Attach a script to the MainMenu root and add:

```
extends Control

func _on_start_button_pressed() -> void:
    get_tree().change_scene_to_file("res://Scenes/Game.tscn")

func _on_instructions_button_pressed() -> void:

get_tree().change_scene_to_file("res://Scenes/Instructions.tscn")

func _on_scores_button_pressed() -> void:
    get_tree().change_scene_to_file("res://Scenes/Scores.tscn")
```

Step D — Connect the Buttons

Connect each button's pressed signal to the MainMenu root.

Button	Signal	Method
StartButton	pressed	_on_start_button_pressed
InstructionsButton	pressed	_on_instructions_button_pressed
ScoresButton	pressed	_on_scores_button_pressed

To connect each signal:

- Select the button in the Scene tree.
- Open the Node dock.
- Find the pressed signal.
- Double-click it and connect it to the MainMenu root.
- Use the corresponding method name from the table.

Step E — Set the Main Scene

- Open **Project → Project Settings**.
- Select **Application → Run**.
- Find **Main Scene** and **select Scenes/MainMenu.tscn**.
- Close Project Settings.
- Press Play Project to test the game. The **Main Menu should appear first**.
- Select **Start Game** to open **Game.tscn**.

Note: Keep GameManager registered under Project → Project Settings → Globals → Autoload. The Autoload and Main Scene settings serve different purposes.

Step F — Create the Instructions and Scores Screens

The Main Menu buttons are already connected to **Instructions.tscn** and **Scores.tscn**.

Create these scenes so that all three menu buttons lead to working screens.

1. Create Instructions.tscn

- Create a new scene with a **Control root node** named **Instructions**.
- Save it as **Scenes/Instructions.tscn**.
- Add a **Label** containing the game instructions:

Use the arrow keys or configured WASD controls to move the bear.
Collect coins to earn points.
Coins disappear after 8 seconds, so collect them quickly!
Earn as many points as possible before the 60-second timer ends.

- Add a Button named **BackButton** and set its **displayed text** to **Back**.
- Attach a script named **Instructions.gd** to the root node. Add this function:

```
extends Control

func _on_back_button_pressed() -> void:
    get_tree().change_scene_to_file("res://Scenes/MainMenu.tscn")
```

- Connect the **BackButton node**'s **pressed signal** to **_on_back_button_pressed()**.

2. Create Scores.tscn

- Create a **new scene** with a **Control root node** named **Scores**.
- Save it as **Scenes/Scores.tscn**.
- Add a **Label** named **FinalScoreLabel** to display the most recently completed round's score.
- Add a **Label** named **HighScoreLabel** to display the highest score saved by the game.
- Add a **Button** named **BackButton** and set its displayed text to **Back**.
- Attach a script named **Scores.gd** to the root node. Add this script:

```
extends Control

@onready var final_score_label: Label = $FinalScoreLabel
@onready var high_score_label: Label = $HighScoreLabel

func _ready() -> void:
	final_score_label.text = "Last Score: " + str(GameManager.final_score)
	high_score_label.text = "High Score: " + str(GameManager.high_score)

func _on_back_button_pressed() -> void:
	get_tree().change_scene_to_file("res://Scenes/MainMenu.tscn")
```

- Connect the **BackButton node**'s **pressed signal** to **_on_back_button_pressed()**.

3. Test the Menu Screens

Run the project using F5 (Play Project) and check that:

- The **Main Menu appears first** with the background image and three buttons.
- Clicking **Start Game** opens **Game.tscn**.
- Clicking **Instructions** opens **Instructions.tscn**.
- Clicking **Scores** opens **Scores.tscn**.

- Clicking **Back** on either screen **returns to the Main Menu**.

Step G — Test High Score Persistence

The **GameManager** script already includes the functions needed to save and load the high score.

An Autoload keeps the GameManager available while the player moves between scenes, but ordinary variables are reset when the application closes. Saving the high score to a file allows it to remain available when the game is launched again.

Before continuing, make sure your GameManager.gd script includes:

- **const SAVE_PATH = "user://save_data.json"** to define where the save file is stored.
- A call to **load_high_score()** inside **_ready()** so the **saved high score is loaded when the game starts.**
- A call to **save_high_score()** inside **finish_round()** when the **player earns a new high score.**

Test your work:

- Run the game and complete a round with a new high score.
- Return to the Main Menu and open the Scores screen. Confirm that the high score is displayed.
- Close the running game completely and launch it again.
- Open the Scores screen and confirm that the high score is still displayed.
- Play another round with a lower score. Confirm that the saved high score does not decrease.

Important: The high score is saved **between sessions.**

The last-round score shown on the Scores screen is **held in memory** and is **not saved between application launches** unless you add separate save functionality for it.

7. Test and Troubleshoot

- Before exporting, test the complete game.
- Use the debugger and Output panel to identify errors.
- Fix the cause of a problem rather than making unrelated code changes.
- Run the game from the configured Main Scene and test the following features:

Feature	What to check
Main Menu	Appears first when the project runs.
Player movement	The bear moves in the configured directions.
Coin collection	Coins disappear and increase the score.
Coin spawning	A new coin appears every 3 seconds at a random spawn point.
Coin expiration	Uncollected coins disappear after 8 seconds.
Countdown	The round ends after 60 seconds.
GameOver screen	Displays the final score.
Play Again	Starts a new round with the score reset.
Main Menu	Returns to the title screen.
High score	Saves and loads correctly.
Instructions and Scores	Both screens open and their Back buttons work.

8. Export to the Web and Publish on itch.io

Once your game is working, export it as a Web build so players can run it in a browser.

Step A — Prepare the Web Export

- Open the **Project > Export panel** in Godot.
- Click **Add...** and select **Web** to create a **Web export preset.**
- If Godot prompts you to install export templates, follow the instructions to install them.
- Select the **Web preset** and review its settings. The default settings should be sufficient for this activity.
- Choose an **export location in a separate folder**, such as **CollectTheCoins_GodotLab**.
- Click **Export Project** and save the exported game.

Step B — Package the Web Build

- Open the folder containing your exported files.
- Select all the files inside the export folder, including index.html and the supporting game files.
- Compress the selected files into a ZIP archive.
- **Important: The ZIP must contain index.html at its root. Do not place the files inside an additional folder within the ZIP.**

For example, the ZIP should look like this:

```
CollectTheCoins_GodotLab.zip
├─── index.html
├─── index.js
├─── index.wasm
├─── index.pck
└─── Other exported files...
```

Your exported filenames may differ. Include all files generated by Godot, not just the examples shown above.

Step C — Upload to itch.io

- Visit **itch.io** and sign in to your account.
- Select **Create new project**.
- Enter your game's title and a short description explaining its objective and controls.
- Under Kind of project, select **HTML**.
- Upload your Web build **ZIP file**.
- Enable **This file will be played in the browser** for the uploaded file.
- Configure the game's viewport dimensions to match your project settings.
- Add a cover image or screenshots if available.
- Save the project and open its page to test the published game.

Step D — Submit to the CISC 3667 Godot Lab Game Jam

- Open the **CISC 3667 F26 Godot Lab Submission Game Jam**.
- Sign in to your itch.io account.
- Select the option to submit your game to the Jam.
- Choose the itch.io project page you just created.
- Review your submission and confirm that it has been submitted successfully.

Important: Publishing your game on itch.io does not automatically submit it to the class Game Jam. You must complete both steps.