

THE UNIVERSITY OF WARWICK

WARWICK GAME DESIGN SOCIETY

A GODOT BASICS ARTICLE

Designing Checkers in Godot 4 for Beginners

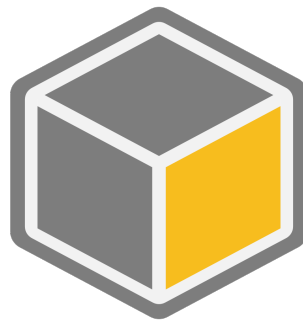
Authors

Thomas J. WALSH
Gustavo TEAGLE

Edited by

The Game Design
EXECUTIVE TEAM

October 7, 2026



1 Introduction

Welcome! Thank you for choosing to enhance your Godot skills with this tutorial! Today we'll take you through building Checkers in Godot, so you can play with your friends!

This tutorial mainly encompasses the following areas:

1. Nodes and the Scene Tree
2. Using Scenes & Instantiation
3. Scripts and coding in GDScript
4. Using Signals

If you'd like to exercise your skills in other areas of Godot, feel free to try out one of our other amazing tutorials!

This tutorial also starts out with basic Godot installation and familiarisation instructions, so if you're all good with that, feel free to skip to the section titled 'The Board'

Without any further ado, lets go!

2 Origins

The Engine The Godot Game Engine was released to the public on the 14th of January 2014, with the ambitious aim “to become the most amazing game engine the world has ever seen!”.⁽¹⁾ Godot is MIT-licensed; you receive permanent ownership, editing and redistribution rights when you download it.

The Game Checkers is, originally, not a video game (shocking, I'm sure). It is played on an 8x8 board with 2 players. Players move their

pieces diagonally forward one square at a time, and may capture opposing pieces by jumping over them; jumps may be chained if the position allows for it. If a piece gets to the opposite side of the board, it becomes a King and can move in any diagonal direction. A player loses when they have no more pieces on the board, or cannot legally move any of their pieces. Here's a [Hasbro](#) rulebook for Checkers if you're still confused: [Checkers.PDF](#)

Developing Checkers itself may seem simple, but it teaches us a lot about some of the core elements for many different board/card game style video games, such as [Hearthstone](#), or even tactical grid-based games like [Fire Emblem](#) and [Into The Breach](#)!

Installation There are two main ways to install Godot (with a litany of others including package managers, compilation from source etc.):

- *Steam* (recommended): convenient, fast and easy to update – what more is there to want?
- *Godot's Website*: useful when Steam's automatic updates cause problems in big projects.

3 Godot Orientation

Projects Godot opens to the project manager which lists your projects and provides project creation / import functionality. You may not have created any projects for the manager to list, so you need a new one to start developing. The project creation window [fig.1] is a form through which you tell Godot some general information about what it should expect from your game.

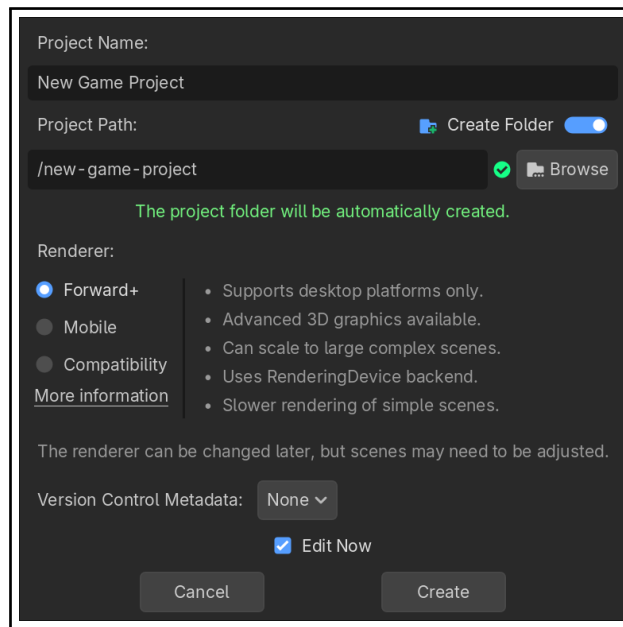


Figure 1: The project creation window

Name the project as you wish and place the project in an empty path (folder) of your choice. I have set the renderer option to ‘Forward+’ mode because my computer is reasonably powerful. Compatibility mode can provide a much better experience on some computers – see [here](#) for more detail. If you have experience in Git you can select it in the version control metadata drop-down; Godot will provide you with a preconfigured gitignore file. Make sure that ‘Edit Now’ is ticked and, when you’re ready, hit create.

The Editor Welcome to Godot's editor [fig.2]! Leave everything as it is for the time being. You will spend much of your time here during the course of making any game in Godot, so it's worth taking a moment to get familiar. The editor's centrepiece initialises to the 3D viewport. For now, switch over to the 2D viewport with the panel at the top of the screen.

Note for macOS. If you're using macOS, you may not see the [Scene, Project, Debug, Editor, Help] buttons at the very top left. They are hiding in the native Apple menu bar.

Everything should look fairly empty at the moment. Just a reminder: if at any point you get lost (e.g., this image is not what you're seeing), feel free to ask anyone for help. With that, we can start building Checkers!

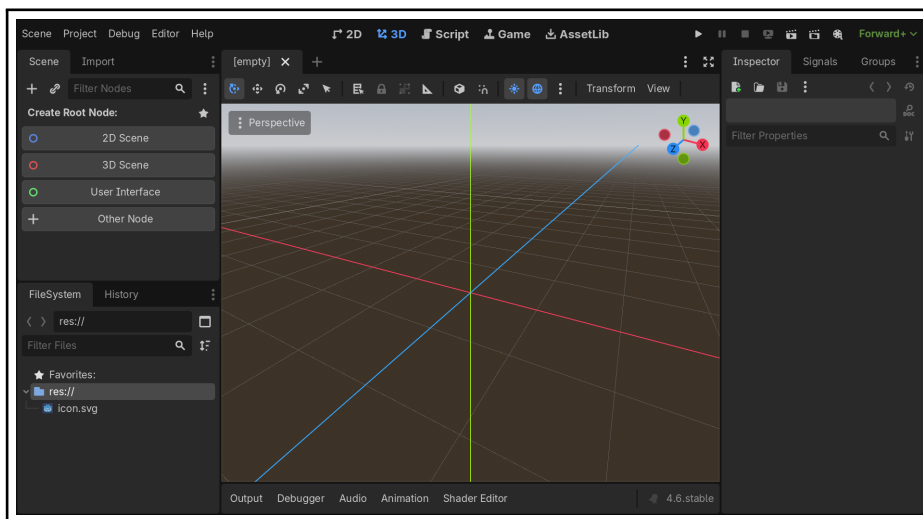


Figure 2: Godot's editor

4 The Board

Nodes, Scenes

In Godot, everything in a game is made up of Nodes. A node is an individual game object, which can be almost anything, from the player character, to a collectable coin, to an entire level. It's important to distinguish between nodes and resources: nodes are objects within Godot itself, whilst resources are used by Godot but are not a part of it. Some examples include:

Resources:

- A music file
- An image file containing a sprite
- An animation spritesheet

Nodes:

- A music player
- A image/sprite display
- An animation player

Nodes can have child nodes which add more attributes and functionality to it. We'll see examples of this very soon, where it should make more sense.

Scenes exist in the Godot editor to aid with development. A scene is merely a node and all of its child nodes. Scenes allow for the easy development of game objects, and also allow game objects to be instantiated during runtime in the form of a 'packed scene'. Again, we'll see examples of this later, so don't worry about memorizing all this information now!

The Board

OK, so let's actually create the board where our game of Checkers will take place. If you read the last section, you probably realise that the board needs to be a node. Let's take a look at the left sidebar[fig.3] - this is currently on the *scene view*, which shows you all the nodes in the current scene (remember - a scene is just a node and all of its subnodes!).

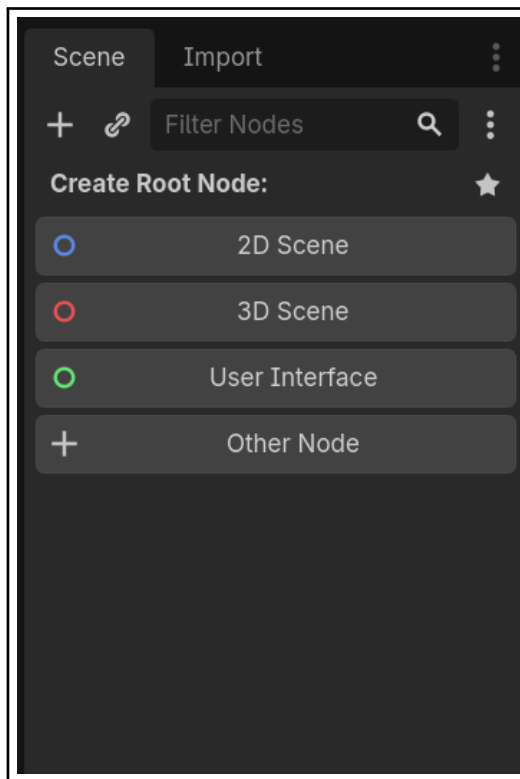


Figure 3: The scene panel

There are a variety of different nodes in Godot, all with their own unique functionality. Feel free to click ‘Other Node’ and have a look at all the different nodes and their descriptions. For us though, our ‘board’ node is just going to be an abstract space for us to play our game, so a standard Node2D will do just fine. You can select ‘Node2D’ from the list of nodes or simply click ‘2D Scene’ in the sidebar. You should now have a single node in your scene view, simply called Node2D. Let’s right click the node and rename it to something more appropriate, like ‘Board’.

Before we do anything further, let’s save the scene. Simply hit ctrl+s (cmd+s on mac) to save the scene. Notice how it brings up a pop-up to an empty folder you don’t recognise? This is the *project folder*, and is where Godot stores all the information needed to make the game, including scenes! Click [here](#) for more information on how projects are stored in Godot, if you’re

interested.

Finally, lets run our game. I know, I know, there's nothing actually *in* the game yet, but just to see how it works. Click the play icon in the top right corner of the editor (or hit f5), and watch as the game doesn't run. Godot should have warned you that you haven't registered a *main scene* for this project yet. A main scene is the scene that the game will start on when its opened. We want the board to be our main scene, so just click 'Select Current'. And now the game should run! If you're looking at a grey box right now, congratulations: you've been following the tutorial correctly thus far. Now, lets actually create the physical board, and not just some abstract space for us to play the game, shall we?

5 Tiles

No Board? A Checkers board is made up of 64 tiles arranged to make a square board. So before we can actually construct the board, we need to make the tiles first.

Go to the top of your screen and click the plus icon next to the tab that has the name of our current scene (i.e., board). This will create a new empty scene for us to work with. We will now create a tile node, but instead of making it just a generic Node2D, we want to make it something more specific. Click ‘Other Node’ and search for ‘Area2D’. Feel free to have a read of the in-built documentation to learn about what ‘Area2D’ does. Select that and rename it to something like ‘Tile’.

You should now notice a little yellow warning symbol next to our tile node in the scene view. Worry not! This is normal. If you hover over the warning icon, it will say what the issue is quite plainly:

CharacterBody2D Warning. This node has no shape, so it can’t collide or interact with other objects. Consider adding a CollisionShape2D or CollisionPolygon2D as a child to define its shape.

Our tile doesn’t have a shape, and we can give it a shape via a child node. Right click the node and select ‘Add Child Node’. Then, search for ‘CollisionShape2D’, and select it. We’ve now given our tile a shape! ...or at least, a node that *can* be a shape. You’ll notice the warning has moved to our new child node, and has changed:

Node Configuration Warning. A shape must be provided for CollisionShape2D to function. Please create a shape resource for it!

Again, simple enough, we just need to give the node a shape to be. If you look at the right sidebar, you'll see a bunch of information. This right sidebar is the *Node Inspector*, and shows information about the currently selected node. With our CollisionShape2D node selected, there should be a *shape* attribute which is currently empty. Click the text that says 'empty' (NOT the folder icon), and a list of shapes should appear. Tiles are square, so we're gonna go with RectangleShape2D. Now all the warnings are gone! Yippee! Let's save our new scene.

Scene Tree I will now take this moment to explain the core concept that makes up every project ever made in Godot: the Scene Tree.

Notice how the new CollisionShape2D node we just added became connected to our Tile node by a little line? This is a visual representation of the scene tree. The scene tree is how Godot organises all the nodes within a project. This creates a form of hierarchical structuring of the nodes in a game.

Every Godot project starts with an invisible 'root node' that makes up the top of the tree. Then, the main scene is loaded on as a child of the root node. Any child nodes of the main scene are then loaded onto the scene tree as well. This process repeats recursively until every (currently active) node in the game is loaded into the scene tree. Take a look at the example scene tree in [fig.4].

Here, we have the 'Level 1' node as the root node (below the true invisible root node), and it has many child nodes. A node having a child node means that the child node exists within the parent node's context, e.g. the 'Camera' node exists (and only

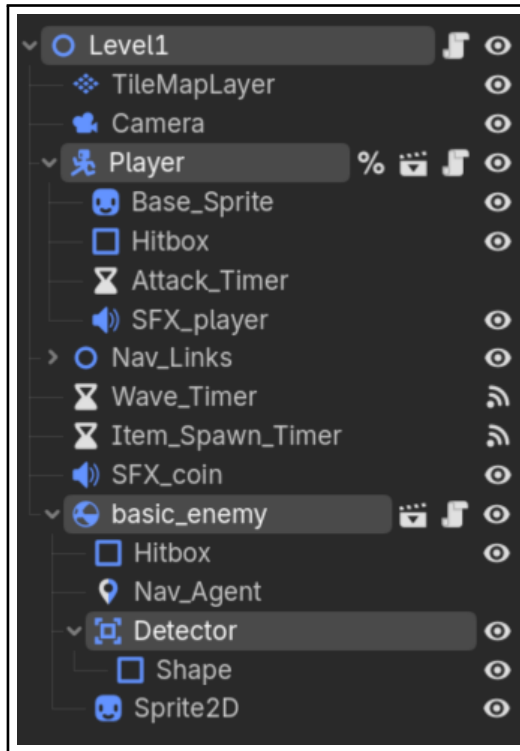


Figure 4: An example scene tree

exists) within ‘Level 1’. If we changed to a different root node, say ‘Level 2’, that did not have its own camera node, then there would be no camera for that level, so we wouldn’t be able to see it properly.

Child nodes can also add functionality to their parent node. For example, ‘Player’ has many child nodes that give it some functionality, like a sprite or a hitbox. Since these are children of the ‘Player’ node, they exist within the ‘Player’ node’s context, i.e., when the player moves, they will move to, without being explicitly told to in any code.

On top of that, having this tree-like approach influences the way nodes interact with one another. A parent node has direct access to all of its children (and its children’s children), but a child node does not have direct access to all of its parents. This means the root node, ‘Level 1’, can directly access and change all nodes

you can see in the tree if it wanted to, whereas ‘Attack Timer’ only has control over itself, since it has no children. You may notice an issue with this, if a node wishes to communicate with another node higher in the tree hierarchy than it. This is what *signals* are for, and we will cover those in more detail later.

Before moving on, take a moment to make sure you fully understand the scene tree. A proper understanding of the basics of how projects are structured makes it a lot easier to develop games in Godot!. Ask yourself these questions to test your knowledge:

- How many child nodes does ‘Player’ have?
- Who is the direct parent of ‘Item-Spawn-Timer’?
- Does ‘Basic enemy’ have access to ‘Shape’?
- Does ‘TileMapLayer’ have access to ‘SFX-Coin’?

To learn more about the scene tree, check out the [Godot Docs](#).

Tiles Lets go back to our tile. So far, it has a shape, but no actual sprite, so we won’t be able to see it in-game. We can fix this by giving it a sprite. Add a child node to our ‘Tile’ node and select Sprite2D. Now the tile has a node to display a sprite, but there is no actual sprite resource to be displayed. Alongside this tutorial, there should have been an ‘Assets’ folder provided to you (please ask an exec member if you can’t find it!). Take that folder and move it into your project folder for this project. You can open your project folder in your file explorer by right-clicking the ‘res://’ folder in the bottom-left ‘FileSystem’ tab [fig.5], and selecting ‘Open in File Explorer’.

Once you’ve copied the folder in, it should appear within the Godot editor in the ‘FileSystem’ tab [fig.5], alongside our two scenes that we’ve already made, and one ‘icon.svg’ (which is a little freebie that Godot gives you whenever you start a new

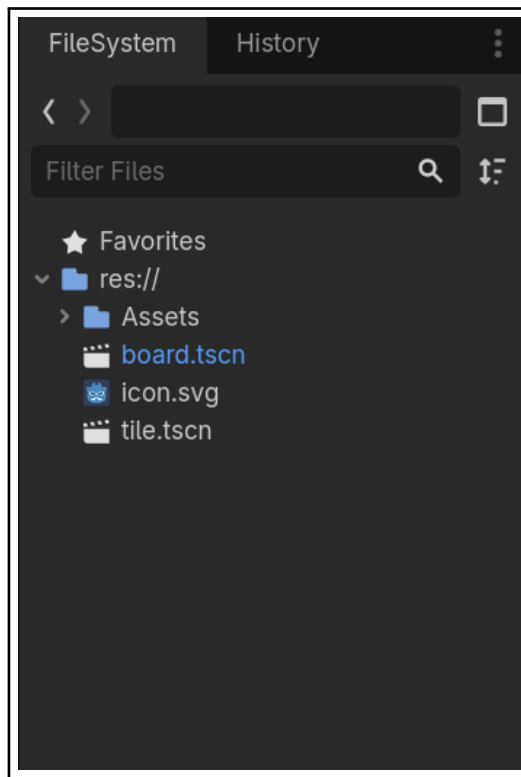


Figure 5: The FileSystem tab

project). With the Sprite2D node selected, drag the ‘tile.png’ image into the ‘Texture’ property on the Inspector tab. You should now be looking at a tile on your screen! You might also notice that it looks a bit fuzzy at the edges. This is because we’re working with pixel art, but haven’t told Godot that we are yet, so it’s trying to make the image look all smooth-like, which we don’t want. You can fix this by going to the top left of your screen and going to Project -> Project Settings -> Rendering -> Textures and change ‘Default Texture Filter’ from ‘Linear’ to ‘Nearest’.

A quirk of the scene tree that we discussed previously is that nodes get "drawn onto" the screen in the order they are in the scene tree. Try rearranging the Sprite2D and CollisionShape2D nodes by right-clicking them and selecting ‘Move Up’ or ‘Move Down’. You should see that when the Sprite is first in the tree,

you can see the collision shape drawn on top of the sprite in the editor window. Resize the collision shape to fit the sprite by clicking on it in the editor window and dragging its corner points out.

Now we have a tile that we can look at! With that said, lets finally get onto making the board, shall we? (Don't forget to save your progress!)

6 Scriptwriting

Scripts Scripts are files of code that are attached to nodes that allow us to give the nodes extra functionality. Godot supports two programming languages: its own custom language called GDScript, and C#. Ultimately, which one you use is down to personal preference. However, I personally do recommend using GDScript. Even though you'll have to learn a new language, since GDScript is bespoke to Godot, it gives a lot more freedom with controlling the nodes and lower-level operations of Godot. I'll be using GDScript in this tutorial, but again, which one you use is up to you.

Lets add a script to our board node in order to generate it when we start the game. Go back the the board scene and select the board node. You should notice a little medieval scroll looking icon appear at the top of the scene view tab. That's the *attach script* button, so click that. It will give you an option for how to name/save the script, we're not particularly worried about clean folders here so just click OK. The script should now open up as seen in [fig.6]

As you can see, the script comes with two functions already filled out: `ready()` and `process()`.

- The `ready()` function gets called whenever the node the script is attached to first enters the scene tree. So for us, it will run as soon as the game starts.
- The `process()` function runs once every frame the game is running. I.e., if the game is running at 30FPS (frames per second), the `process()` function will run 30 times a second.

Click [here](#) for more information on the syntax and semantics of GDScript. Feel free to refer to the docs as much as you want

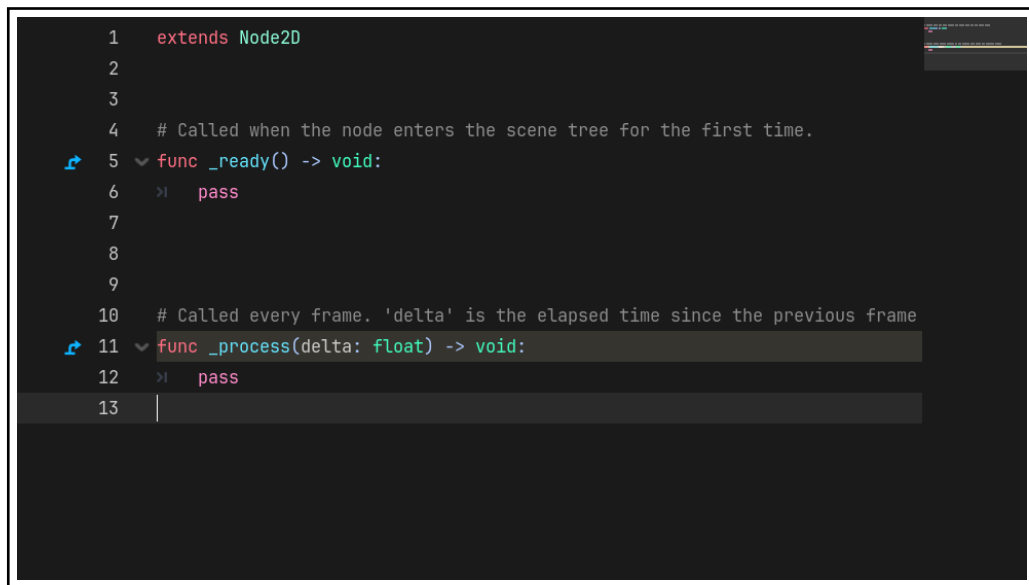


Figure 6: A blank script

during this tutorial to ensure you understand what's going on. Feel free to ask an exec if you need additional help!

To load the board in, write the script from [Listing.1] into the ready() function.

```
1 func _ready() -> void:
2     const TILE = preload("res://tile.tscn")
3     for y in range(-4, 4):
4         for x in range(-4, 4):
5             var newTile = TILE.instantiate()
6             var tileShape =
7                 newTile.get_node("CollisionShape2D").shape
8             newTile.position.x = position.x +
9                 (x*tileShape.size.x) + tileShape.size.x/2
10            newTile.position.y = position.y +
11                (y*tileShape.size.y) + tileShape.size.y/2
12            add_child(newTile)
```

Listing 1: The code for loading in the board

We put this code in the ready() function so it runs only once when we start the game.

We see what we referred to earlier as a PackedScene in Line 2.

A packed scene is just the blueprint or description of a scene that Godot uses to instantiate a version of that scene later in the code. When we *preload()* the packed scene "res://tile.tscn" into our constant *TILE*, it gives Godot the blueprint to use for when it creates actual tiles.

By the way, a constant is a variable that can't be changed after it's been set.

After preloading our tile scene, we open 2 for loops. These make sure we're making the right amount of tiles for the board. Each one ranges between -4 and 4, so 8 in total - the last value in a for loop isn't counted - so it will be -4, -3, -2, -1, 0, 1, 2, 3. We have one loop of 8 inside another loop of 8, meaning the code inside both loops will run $8 \times 8 = 64$ times, corresponding to our 64 tiles.

Within the actual loops, we *instantiate* an object of our packed tile scene. Instantiating a scene means making it an actual object in the game, so Godot is using the blueprint we gave it earlier to make a tile node in the actual game. We store this instantiated node in a variable called *newTile*.

Then we need to position our tile properly. First, we need to get the shape of the tile - so we get the child node of our tile scene that we made earlier, the *CollisionShape2D*. Then, we set the x and y positions of our new tile to make a grid shape. We set each coordinate to be offset from the board's central position (*position.x* and *position.y*). We offset it by the x and y ranges of each for loop, to put it in the correct grid pattern ($x * \text{tileShape.size.x}$ and $y * \text{tileShape.size.y}$). Then finally we offset it by half the size of the tile itself, to make sure its centred ($\text{tileShape.size.x}/2$ and $\text{tileShape.size.y}/2$).

Finally, and this is the MOST IMPORTANT step, make sure

you use `add_child()` to add the new node to the game. It won't be in the game if you don't do this!

I appreciate that is probably very confusing for someone who's never coded before. If you don't fully understand, feel free to ask an exec who will be happy to explain it to you!

The Board 2 Now if you run the game, you should see our board! ...which is cut off in the top left corner of the window. This is because we haven't given the game a proper camera yet, so it is using the default viewport of the scene to create the window, which we don't really want.

To fix this, add a `Camera2D` node as a child of the board node. The camera node automatically starts centred in the scene, so when we run the game now, we can see our board! Except, it's a little bit small. We want our players to not have to be squinting at the screen when they play our game, so we can zoom the camera into the board to make it easier to see.

With the `Camera2D` node selected, navigate over to the inspector tab and look for the *Zoom* property. You should see there is a value for both the x and y zoom values of the camera. You can click the chain icon between them to link/unlink them. Try out some zoom values! See which you think works best.

And just like that, we've finally created our board! But what's a board without pieces to play on it?

7 Pieces

Pieces Time for a new scene! Click the ‘add new scene’ button at the top of the editor, and select another Area2D node as the root node. Rename this node ‘Piece’ or something similar. Once again give the node a CollisionShape2D, but this time make it a circle instead of a square. You can also give the node a Sprite2D by using the ‘piece.png’ resource in the Assets folder. Align your CollisionShape2D to more or less cover the whole sprite.

Next, with the piece node selected, go over to the inspector and find the *Ordering* option under *Canvas Item*. Expand it and change the *Z Index* from 0 to 1. This ensures that the pieces will always be visible on top of the tiles, since the pieces now have a higher Z index.

And just like that, we’ve got a piece! Go ahead and save that scene.

Placement Now this is where it gets a little more tricky. Let’s think about this theoretically for a second.

When a piece is on the board, it sits on a specific tile. Pieces are never halfway between tiles or anything like that, and pieces that are captured are removed from the game. This means we’ll only ever have one piece per tile. So, it would make sense to have each tile track if it has a piece on it or not, and if so which specific piece is on it.

To do this, we’re going to do some more coding. Go back to the tile scene and attach a script to the tile node (you’ll notice no functions appeared. This is because we’re using a different node type, so a different *template* for the script was used. For more information click [here](#)). We’re going to be adding a variable to

the script. Write the line from [Listing.2] into the script.

```
1 var occupant:Area2D = null
```

Listing 2: Making a variable

This creates a way for the tile to track which piece is on it. Right now we set it to null, which is a special value meaning that there is no object currently assigned to the variable (in our context, the tile has no occupant).

When we open the game, we want each piece to be in its correct starting square. Since all games of Checkers start in the same configuration, we can do this during the ready() function as well. Add the code in [Listing.3] to your ready() function in the board script.

```
1 func _ready() -> void:
2     const TILE = preload("res://tile.tscn")
3     const PIECE = preload("res://piece.tscn")
4     for y in range(-4, 4):
5         for x in range(-4, 4):
6             var newTile = TILE.instantiate()
7             var tileShape =
8                 newTile.get_node("CollisionShape2D").shape
9             newTile.position.x = position.x +
10                (x*tileShape.size.x) + tileShape.size.x/2
11             newTile.position.y = position.y +
12                (y*tileShape.size.y) + tileShape.size.y/2
13             add_child(newTile)
14
15             if ((x + y)%2 == 0) and (y != -1) and (y != 0):
16                 var newPiece = PIECE.instantiate()
17                 add_child(newPiece)
18                 newTile.occupant = newPiece
19                 newPiece.position = newTile.position
```

Listing 3: Adding pieces to the board

First, we preload the piece scene into our PIECE constant on line 3. Then, after instantiating each tile, we have a new block

of code that instantiates a piece if its on the correct tile. Checkers pieces are only ever on one 'colour' of tile (on a traditional board). That actually works out well for us since it means that on every tile with an even sum of coordinates, there should be a piece. That's what the if statement on line 15 checks for. It also ensures that no pieces spawn in the middle two rows ($y \neq -1$ and $y \neq 0$).

If all the checks pass, we instantiate the piece and add it to the game. Then we ensure the piece is set as the occupier of the tile it's on, and position it to be on top of the tile.

Now if you run your game, you should see a game of Checkers ready to be played!

Colours At the moment, it's obvious which pieces belong to which player due to their positioning. But during the messy mid-game, it would be very difficult to tell who owns which pieces if they all look the same. So we can modify the pieces to make it more obvious.

Modify the code in the `ready()` function as in [Listing.4].

```
1 func _ready() -> void:
2     const TILE = preload("res://tile.tscn")
3     const PIECE = preload("res://piece.tscn")
4     for y in range(-4, 4):
5         for x in range(-4, 4):
6             ...
7
8         if ((x + y)%2 == 0) and (y != -1) and (y != 0):
9             var newPiece = PIECE.instantiate()
10            add_child(newPiece)
11            newTile.occupant = newPiece
12            newPiece.position = newTile.position
13            if (y < 0):
14                newPiece.get_node("Sprite2D").modulate =
15                    Color(0.2, 0.1, 0.6, 1.0)
```

Listing 4: Changing piece colour

All this does is it changes the colour modulation of all the pieces above the centre of the board to be more blue. Now if you run the game, you'll see do different types of pieces!

Now that we have the pieces on the board, its time to move them around!

8 Moving Pieces

Selection When our player wants to select a piece, they will click on the piece. But this also means they will click on the tile the piece is on. Similarly, when a player wants to move said piece, they will click on the empty tile they want the piece to go to. This means if we just implement code for detecting clicks on tiles, it saves us half the effort.

First, we need to be able to detect when the mouse is over a tile. For that, we'll need signals.

Signals In Godot, Signals act as a way for a node to react to events. These events can range from objects colliding with the node, to the node entering or exiting the scene tree. You can even write your own signals to react to custom events, like an enemies health dropping to 0. Signals are definitely my favourite part of Godot!

Firstly, add the variable seen in [Listing.5] to your board script.

```
1 var hover_tile:Area2D = null
```

Listing 5: New variable

Select your tile node and go to the top of the right side-bar, where it says 'Inspector'. You should notice a 'Signals' tab to the right of it. Click on that, and it will turn the right side-bar from the node inspector to a list of the node's signals. The signal we want to use is `mouse_entered()` (this is why we made the tile an `Area2D` node, by the way). Navigate back to your board script and add a `connect()` function after you finish instantiating a new tile, as seen in [Listing.6].

```

1 func _ready() -> void:
2     ...
3         add_child(newTile)
4         newTile.connect("mouse_entered",
5             on_mouse_entered,
6             CONNECT_APPEND_SOURCE_OBJECT)
7     ...

```

Listing 6: Connecting the Signal

What this does is set the board node to be the node that is listening for the `mouse_entered()` signal that will be emitted by the tiles. Once it hears this signal, it will run the function `on_mouse_entered()`, which you can now add into your script, as in [Listing.7]. The `CONNECT_APPEND_SOURCE_OBJECT` parameter simply adds the node emitting the signal to the parameters of signal (parameters are data that passed alongside the signal), so that we know which tile it is that is emitting the signal.

```

1 func on_mouse_entered(tile: Area2D) -> void:
2     hover_tile = tile

```

Listing 7: The receiving function

Great, now we know which tile the mouse is currently hovering over. All we need to do now is detect when a mouse click occurs, and we can select the tile the mouse is hovering over. To do that, we need to actually tell Godot that a mouse click is an input in our game. Go to the top-left of your screen and navigate to Project -> Project Settings -> Input Map. You should be met with a window like the one in [fig.7]

Where it says ‘Add New Action’, type in ‘select’ and hit enter. That should add a ‘select’ action to the list of possible actions in your game. Press the plus button on the ‘select’ line to bring up the *Event Configuration* window. Godot is now listening for

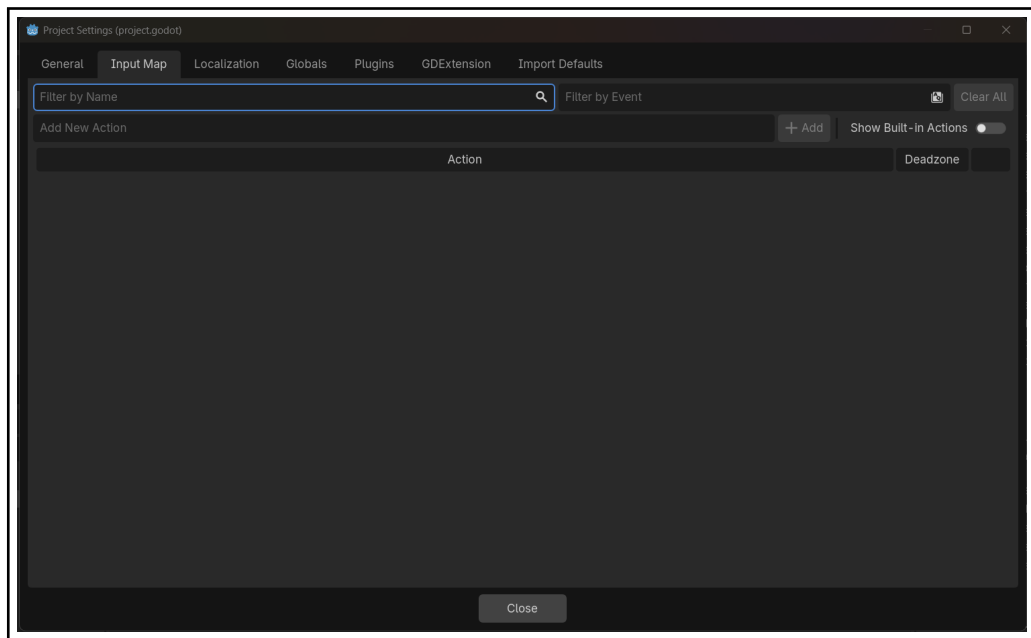


Figure 7: The Input Map window

any input from you to map to the ‘select’ option. Click the text that says ‘Listening for Input’ and it should be replaced with ‘Left Mouse Button’. Hit OK, and voila! We have a way for our player to interact with our game. Exit out of the Project Settings and go back to the board script.

Movement It’s time to finally use the board’s `process()` function. Add the code from [Listing.8] and [Listing.9] to your board script.

```
1 var current_piece:Area2D = null
```

Listing 8: A new variable

```
1 func _process(delta: float) -> void:
2     if Input.is_action_just_pressed("select"):
3         if current_piece == null:
4             if hover_tile.occupant != null:
5                 current_piece = hover_tile.occupant
6                 hover_tile.occupant = null
7         else:
8             if hover_tile.occupant == null:
9                 hover_tile.occupant = current_piece
```

```
10         current_piece.position = hover_tile.position
11         current_piece = null
```

Listing 9: Piece moving code

The first section of the code is responsible for selecting (or "picking up") a piece from a tile when no other piece is currently selected. Then, when we have a piece selected, the second section of code first checks that the tile the mouse has clicked on is empty, then places the piece on that tile if it is.

Feel free to run the game and have a go at moving the pieces around! As you can see, you're able to place pieces wherever you want on the board currently, which isn't how the rules of Checkers work. Now it's time to code in the rules of the board!

9 Movement Logic

Diagonals Pieces should only be able to move diagonally. We can implement this by giving the board a coordinate system and checking if the target tile is exactly one diagonal away from the source tile. Start by heading to the tile script and adding the variable shown in [Listing.10].

```
1 var coords:Vector2i = Vector2i(-1, -1)
```

Listing 10: Coordinate variable

Then, in your board script, add a line to the ready() function as seen in [Listing.11]

```
1 func _ready() -> void:
2     ...
3         add_child(newTile)
4         newTile.connect("mouse_entered",
5             on_mouse_entered,
6             CONNECT_APPEND_SOURCE_OBJECT)
7         newTile.coords = Vector2i(x+4, y+4)
8     ...
```

Listing 11: Adding Coordinates to the Board

This gives each tile a pair of coordinates corresponding to its position on the board.

Now, whenever we move a piece, we just need to check that the difference in the absolute values of the coordinates from the source tile to the destination tile is exactly (1, 1), which corresponds to a diagonal movement. Modify the process() function as seen in [Listing.12] and [Listing.13].

```
1 var source:Area2D = null
```

Listing 12: Source Tile variable

```

1 func _process(delta: float) -> void:
2     if Input.is_action_just_pressed("select"):
3         if current_piece == null:
4             if hover_tile.occupant != null:
5                 current_piece = hover_tile.occupant
6                 hover_tile.occupant = null
7                 source = hover_tile
8         else:
9             if (hover_tile.occupant == null) and
10                (abs(source.coords-hover_tile.coords)
11                 == Vector2i(1, 1)):
12                 hover_tile.occupant = current_piece
13                 current_piece.position = hover_tile.position
14                 current_piece = null
15                 source = null

```

Listing 13: Diagonal movement code

Now test out moving diagonally!

You may have noticed it's a bit tricky to tell which piece you have currently selected. We can easily fix this by once again modulating the colour of our currently selected piece, like in [Listing.14].

```

1 func _process(delta: float) -> void:
2     if Input.is_action_just_pressed("select"):
3         if current_piece == null:
4             if hover_tile.occupant != null:
5                 current_piece = hover_tile.occupant
6                 hover_tile.occupant = null
7                 source = hover_tile
8                 current_piece.modulate
9                     = Color(0.1, 0.1, 0.1, 1.0)
10        else:
11            if (hover_tile.occupant == null) and
12               (abs(source.coords-hover_tile.coords)
13                == Vector2i(1, 1)):
14                hover_tile.occupant = current_piece
15                current_piece.position = hover_tile.position
16                current_piece.modulate = Color(1, 1, 1, 1)
17                current_piece = null
18                source = null

```

Listing 14: Highlighting selected piece

Now it's a lot easier to tell.

Ownership As well as only being able to move diagonally, pieces can also only move forwards, but 'forwards' is a relative term depending on which way you're looking at the board. So, we need a way to tell which pieces should be moving in which direction, and to do that we need to code in piece ownership.

First, we want to make a new script for our pieces. Go to the piece scene and attach a new script to the piece node. Then, add the line in [Listing.15] to that script.

```
1 var player_owner:bool = true
```

Listing 15: New Piece script

Then, go back to the board script and modify the ready() function as seen in [Listing.16].

```
1 func _ready() -> void:
2     const TILE = preload("res://tile.tscn")
3     const PIECE = preload("res://piece.tscn")
4     for y in range(-4, 4):
5         for x in range(-4, 4):
6             ...
7
8         if ((x + y) % 2 == 0) and (y != -1) and (y != 0):
9             var newPiece = PIECE.instantiate()
10            add_child(newPiece)
11            newTile.occupant = newPiece
12            newPiece.position = newTile.position
13            if (y < 0):
14                newPiece.get_node("Sprite2D").modulate
15                    = Color(0.4, 0.2, 0.7, 1.0)
16                newPiece.player_owner = false
```

Listing 16: Setting ownership

We'll use this to represent player 1 with true, and player 2 with false. player 1 will be moving up the board, whilst player 2 will be moving down it.

Now, to enforce the correct directional movement, modify the `process()` function as seen in [Listing.17].

```
1 func _process(delta: float) -> void:
2     if Input.is_action_just_pressed("select"):
3         if current_piece == null:
4             ...
5         else:
6             if (hover_tile.occupant == null) and
7                 (abs(source.coords-hover_tile.coords)
8                 == Vector2i(1, 1)):
9                 var direction = source.coords.y
10                    - hover_tile.coords.y
11                 if (current_piece.player_owner and
12                     direction > 0) or
13                     ((not current_piece.player_owner)
14                     and direction < 0):
15                     hover_tile.occupant = current_piece
16                     current_piece.position =
17                         hover_tile.position
18                     current_piece.modulate =
19                         Color(1, 1, 1, 1)
20                     current_piece = null
21                     source = null
```

Listing 17: Forcing correct direction

After checking if a move is legal, the code now gets the direction of the move by subtracting the source and destination tile's y coordinates. If the direction is the correct direction that the player is supposed to be going, it allows the move to happen.

Run the game and try for yourself! We're getting very close to having a completed Checkers game now! Next up, capturing.

10 Capturing Pieces

Lock in OK, this is probably where the code gets the most complicated in the whole tutorial. But you've been training for this. You've got this.

Firstly, we're going to need a way to store and retrieve each tile on the board. To do that, we first create a new array ([Listing.19]).

```
1 var tiles_array = []
```

Listing 18: Tile array

Then, we make a function to get a tile from the array using its specific coordinates ([Listing.??]).

```
1 func get_tile(coords:Vector2i) -> Area2D:  
2     for tile in tiles_array:  
3         if tile.coords == coords:  
4             return tile  
5     return null
```

Listing 19: Retrieve tiles

Finally, we edit the ready() function to make sure to add each new tile to the tile array when we instantiate them ([Listing.20])

```
1 func _ready() -> void:  
2     const TILE = preload("res://tile.tscn")  
3     const PIECE = preload("res://piece.tscn")  
4     for y in range(-4, 4):  
5         for x in range(-4, 4):  
6             ...  
7             newTile.coords = Vector2i(x+4, y+4)  
8             tiles_array.append(newTile)
```

Listing 20: Adding tiles to the array

OK. Those were the easy bits. Now the hard bit. We're going to rearrange the `process()` function and add some new code. Modify the function as shown in [Listing.21].

```
1 func _process(delta: float) -> void:
2     if Input.is_action_just_pressed("select"):
3         if current_piece == null:
4             ...
5         else:
6             if (hover_tile.occupant == null):
7                 var direction = source.coords.y
8                     - hover_tile.coords.y
9                 if (current_piece.player_owner and
10                     direction > 0) or
11                     ((not current_piece.player_owner)
12                     and direction < 0):
13                     var diff = abs(source.coords -
14                         hover_tile.coords)
15                     [Continued in the next Listing]
```

Listing 21: Code for Capturing

```
1 if (diff == Vector2i(2, 2)):
2     var capture_coords = source.coords +
3         Vector2i((hover_tile.coords - source.coords)/2)
4     var capture_tile = get_tile(capture_coords)
5     if ((capture_tile.occupant != null) and
6         (current_piece.player_owner !=
7         capture_tile.occupant.player_owner)):
8         capture_tile.occupant.queue_free()
9         capture_tile.occupant = null
10        move()
11 elif (diff == Vector2i(1, 1)):
12     move()
```

Listing 22: Code for Capturing part 2

Let's break down what's going on here.

First, we do all the checks we were doing previously:

- We check that we're moving to an empty tile
- We get the direction of movement and check its correct

- We check we're the right distance away

Except for that last point, there's now an extra step. We're allowing both one diagonal and 2 diagonal spaces away, for capturing purposes.

If we are attempting to move 2 spaces away now, we perform additional checks to verify we can capture. We first do some vector math to get the coordinates of the tile that we are about to jump over. We then check that said tile has a piece on it, and that said piece is of the opposite type. If all of these checks pass, we call `queue_free()` on the piece, which is Godot's way of deleting a node from the scene tree.

We then call this `move()` function, which just has all of the code from regular movement we wrote previously. Make sure to add it into your script as seen in [Listing.23].

```
1 func move() -> void:
2     hover_tile.occupant = current_piece
3     current_piece.position = hover_tile.position
4     current_piece.modulate = Color(1, 1, 1, 1)
5     current_piece = null
6     source = null
```

Listing 23: New movement function

And that's capturing! Go on, have a go at taking some pieces!

11 The End?

Or is it? And that's everything for this tutorial! Of course, the game of Checkers is not nearly complete, but it is now in a state where two people could theoretically play and one of them could win.

If you are enjoying yourself, please do continue developing! Here

are ideas for what to add next:

1. Make it so you can deselect pieces after selecting them, if you change your mind.
2. Implement chain-capturing
3. Implement a King piece
4. Implement a win check and win screen
5. Implement a main menu
6. Implement a proper turn tracker, to alternate which pieces can move
7. Implement your own rules!

Use the Godot docs and help from the exec to catapult your way to Checkers greatness!

Thanks for reading this tutorial!

References

- [1] Juan Linietsky. First public release!, January 2014. URL: <https://godotengine.org/article/first-public-release/>.