Hello and welcome to the first installment of this series where we're going to be creating a hexagonal tile-based 2D strategy game using Godot C#. My name is Cameron and I will be your instructor for this course.

Game Overview

The game we're going to be building up in the series is a turn-based strategy game with a map made up of hexagons. This is a common format for strategy games, especially in genres like 4X. What you learn in this project will not only be applicable to building your own hex-based strategy game or other kinds of strategy games, but core features of this project are also applicable to a wide range of other games and projects such as the turn system and procedural terrain generation with noise.

Course Scope

Strategy games are some of the most mechanically complex and feature-rich games out there. Therefore, in this course, we won't be able to implement every possible gameplay mechanic and feature that a game like this could have. This core series will serve mainly as a foundation for learning how to design the core features of these types of games, such as the map, basic units and interactions, and the turn system. It can also serve as a great jumping-off point for creating more complex projects in the future.

Learning Goals

The learning goals for this particular section of the course include:

- Implementing the map with procedural terrain generation
- Learning about Godot's tile map layer system as of Godot 4.3
- Learning how to use Godot's 2D camera to create a camera system
- Learning how to do some basic procedural terrain generation using Godot's Fast Noise Light noise generator

In addition, we're also going to use a lot of intermediate C# features such as C#'s language integrated query.

Prerequisites

The prerequisites for this course include basic skills with Godot and basic C# skills.

Specific Topics

The more specific topics we'll cover include:

1. The new tile map layer nodes as of Godot 4.3
2. Tile sets and texture atlases
3. Godot's Camera2D node
4. Getting mouse wheel input for scrolling on the map
5. Fast noise light, Godot's system for generating random noise
6. C# features such as language-integrated query or link

Before we get started, a word about Zenva. Zenva is an online learning academy with over a million students. We feature a wide range of courses for people who are just starting out or for people who want to learn something new. Our courses are versatile, allowing you to learn in really whatever way you want.

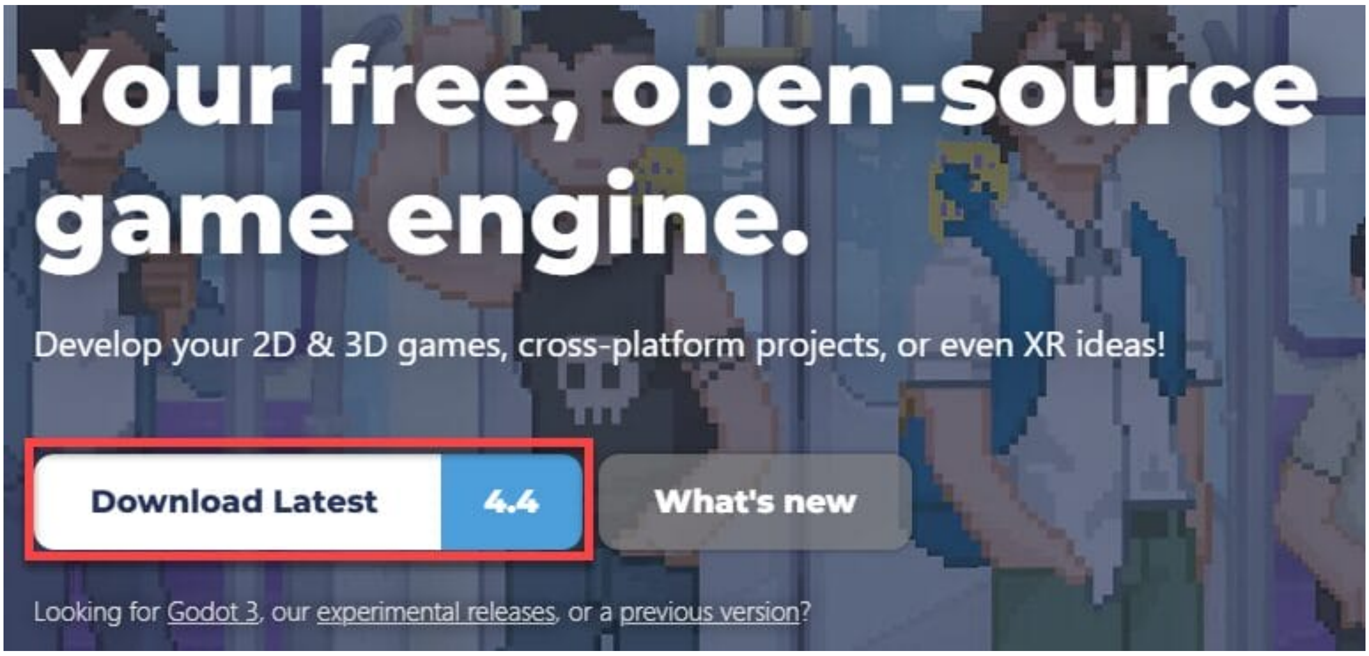
With that said, we're ready to begin our journey into creating a map for a hex-based strategy game. I look forward to seeing you in the next video.

Course Updated to Godot 4.4

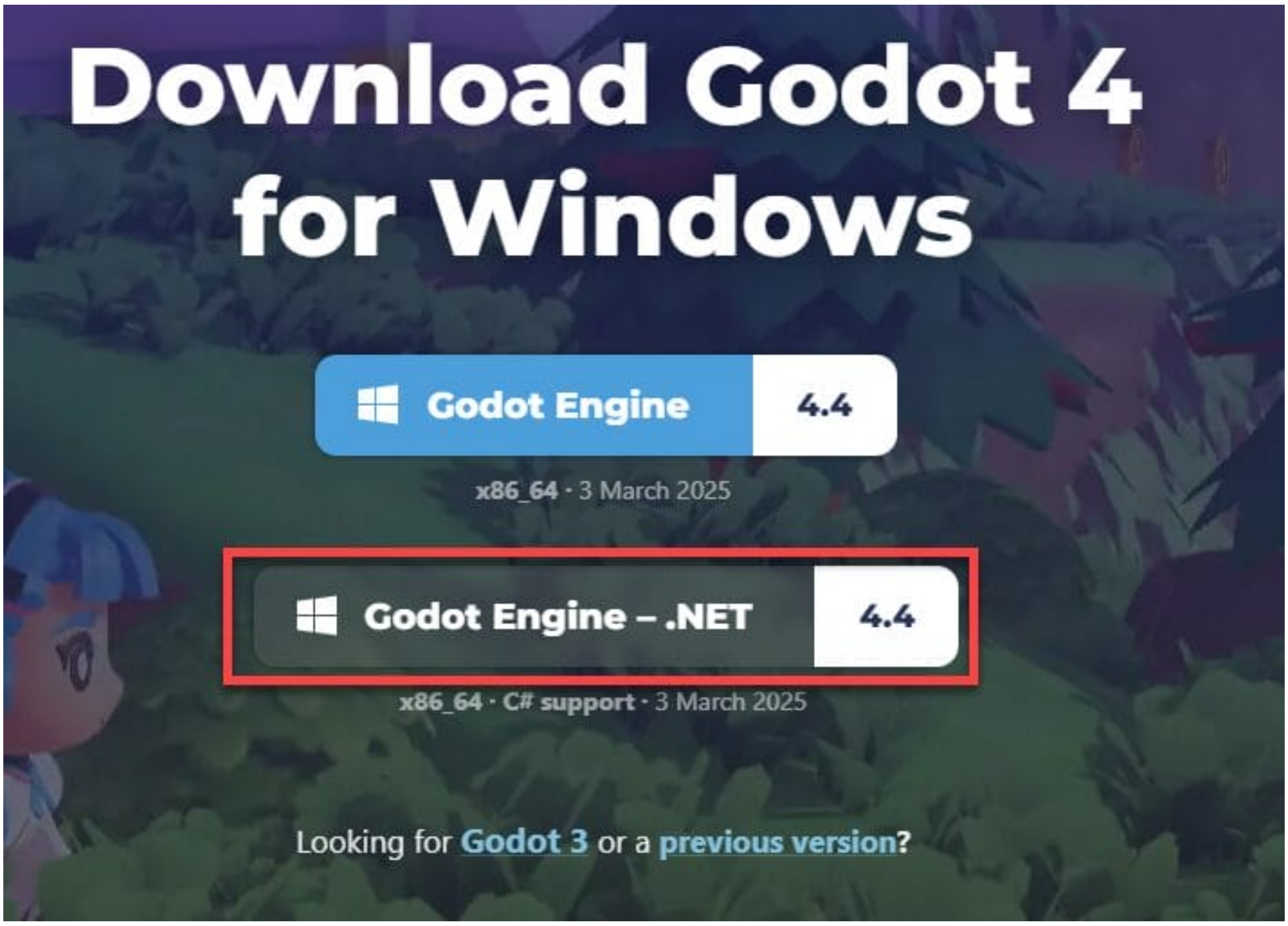
We've updated the project files to Godot version 4.4 for this course – the latest stable release.

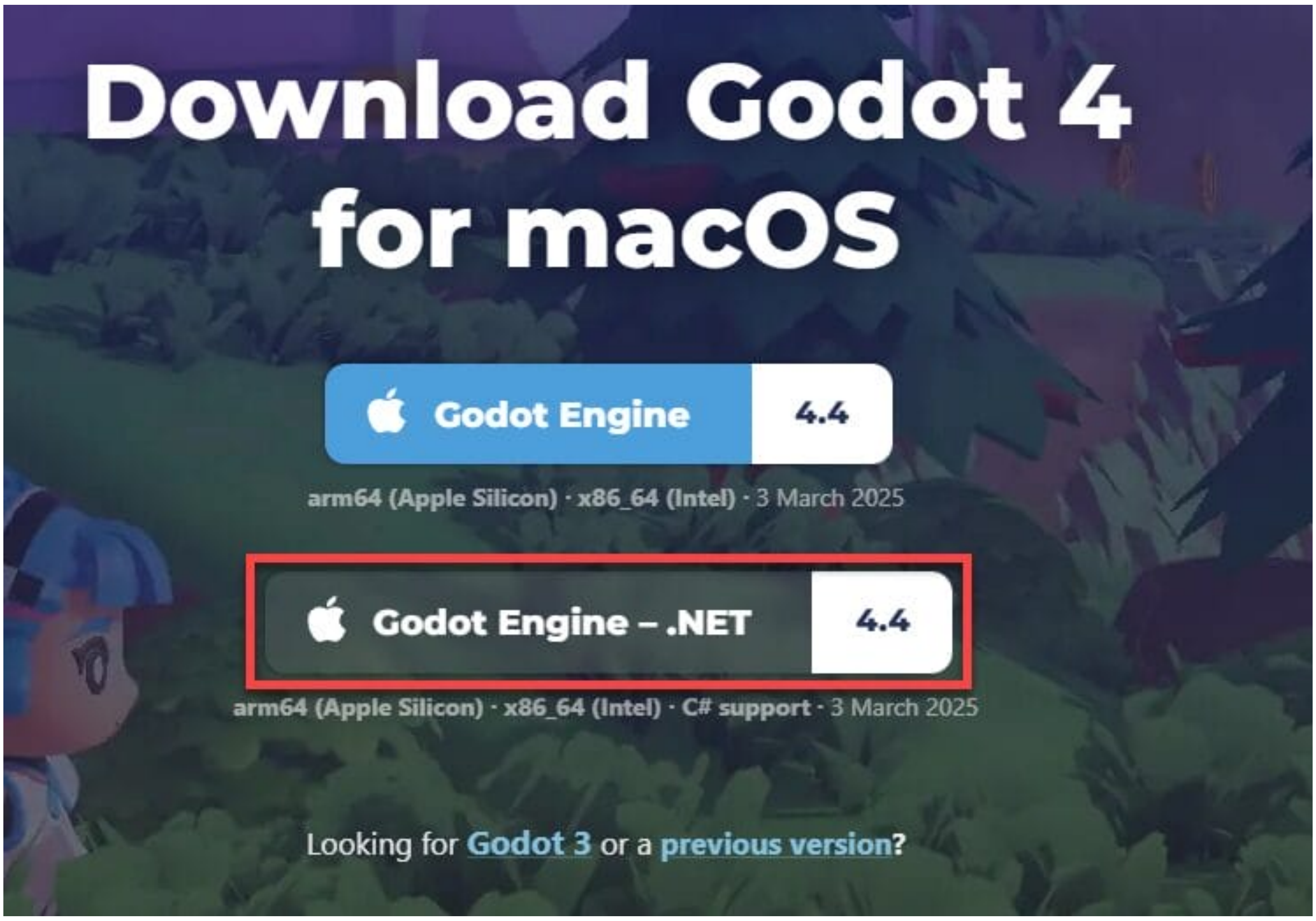
How to Install Version 4.4 .NET

You can download the most recent version of Godot by heading to the Godot website (https://godotengine.org/) and clicking the "Download Latest" button on the front page. This will automatically take you to the download page for your operating system.

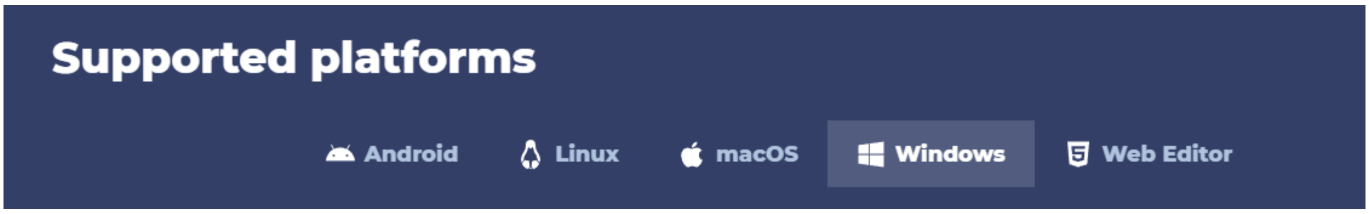

Your free, open-source game engine.
Develop your 2D & 3D games, cross-platform projects, or even XR ideas!
Download Latest
4.4
What's new
Looking for Godot 3, our experimental releases, or a previous version?

Once on the download page, just click the option for **Godot 4.4 .NET.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application launcher – no further installation steps are required for Godot itself!


Download Godot 4 for Windows
Godot Engine 4.4
x86_64 · 3 March 2025
Godot Engine – .NET 4.4
x86_64 · C# support · 3 March 2025
Looking for Godot 3 or a previous version?

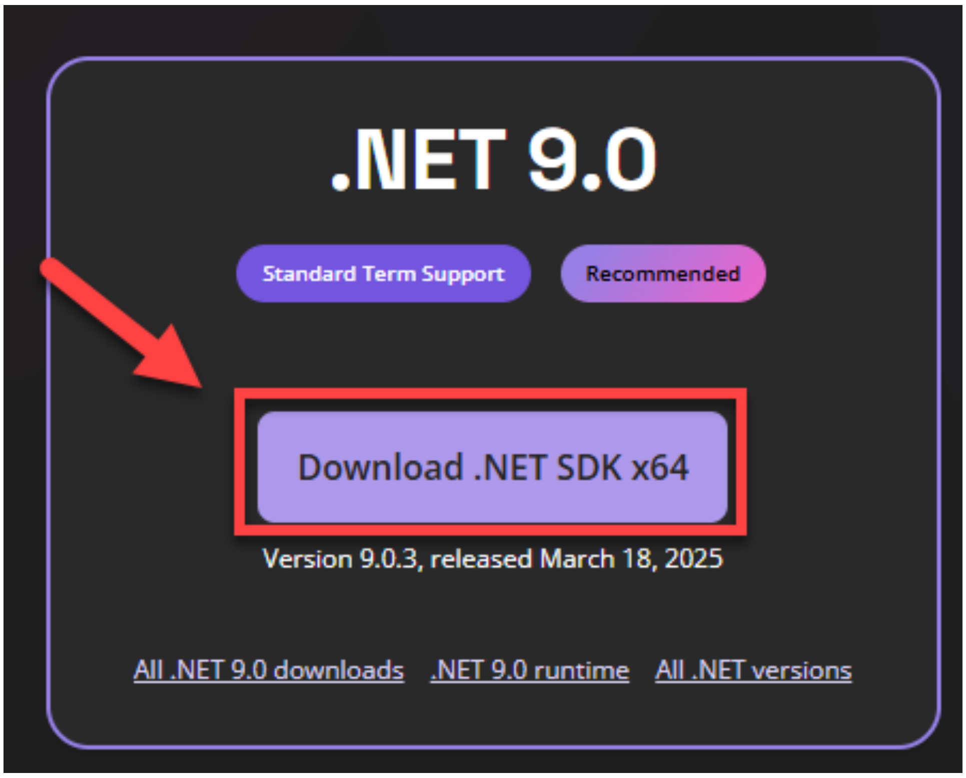

Download Godot 4 for macOS
Godot Engine 4.4
arm64 (Apple Silicon) · x86_64 (Intel) · 3 March 2025
Godot Engine – .NET 4.4
arm64 (Apple Silicon) · x86_64 (Intel) · C# support · 3 March 2025
Looking for Godot 3 or a previous version?

If Godot fails to automatically select the correct operating system for you, you can scroll down to the “Supported platforms” section on the download page to manually select the download version you would like to install:

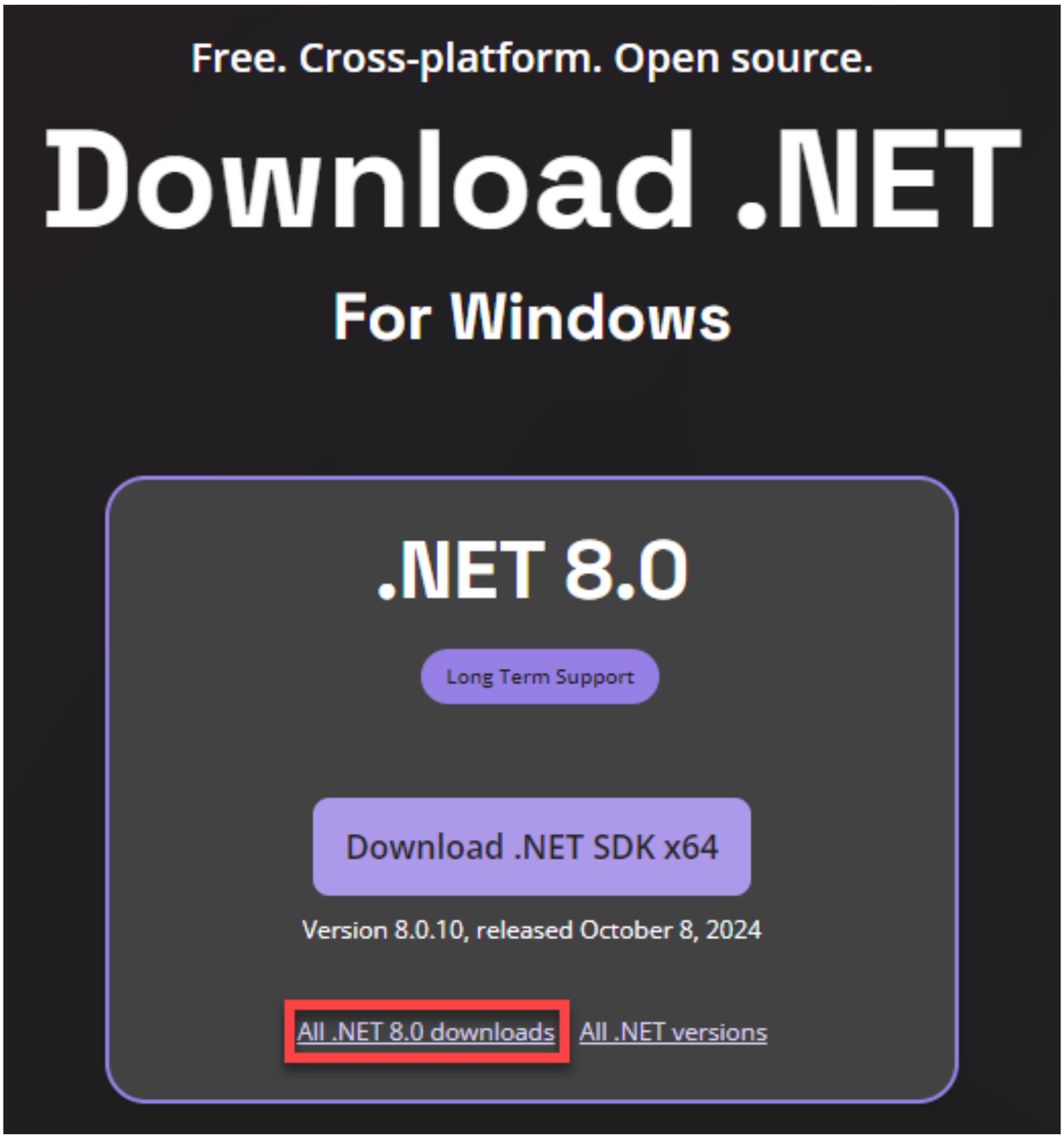


Additional .NET Setup

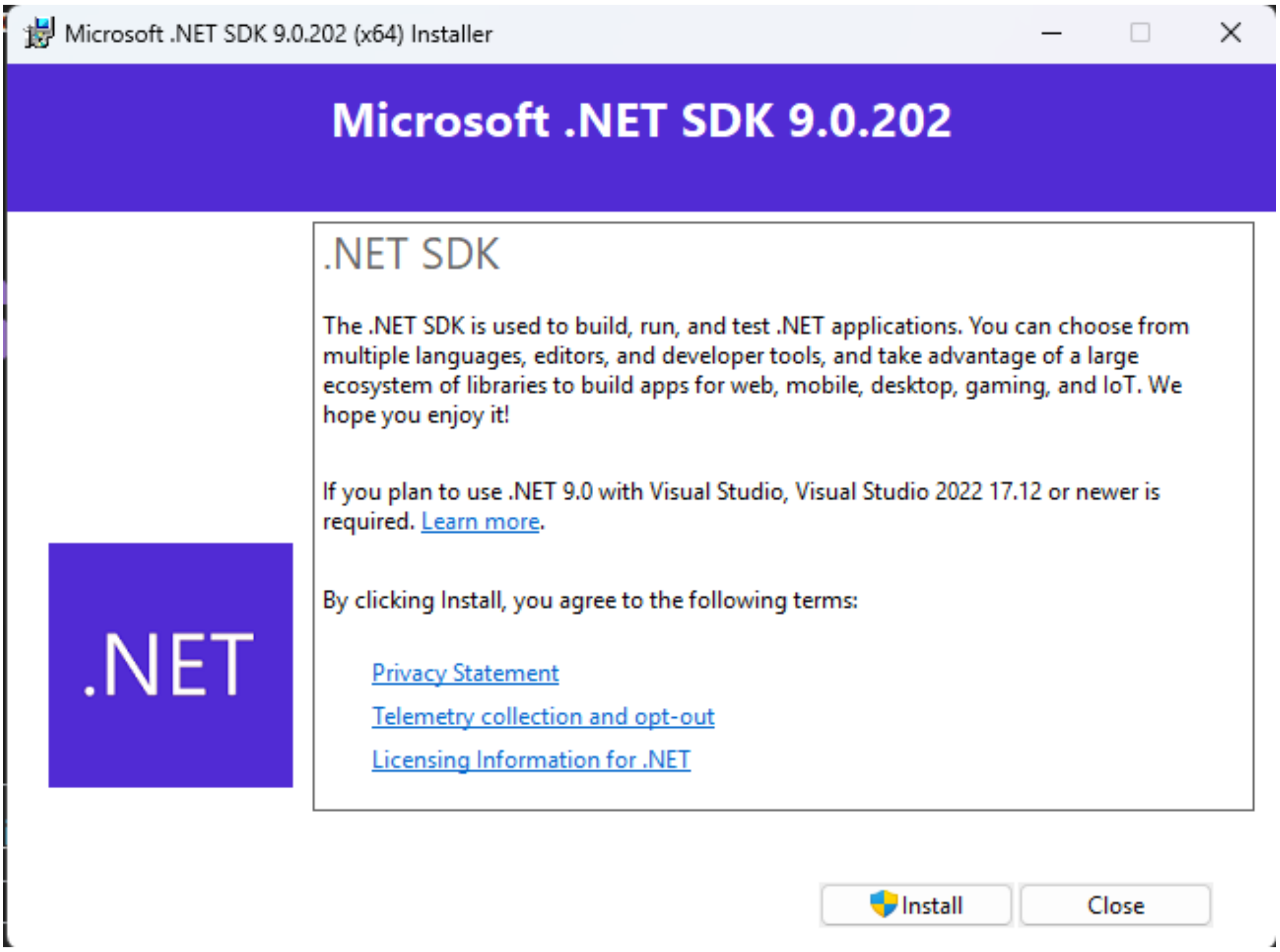
In order to use C# with Godot, you will also need to install the latest version of the **.NET SDK**. To do so, head to the .NET download page (https://dotnet.microsoft.com/en-us/download) and click the **Download .NET SDK** button. By default, your operating system should be automatically selected and download the correct version.



If the site fails to select the correct OS for you, you can also click the “All .NET downloads” link to access all available downloads for the SDK.



Once downloaded, double-click the file to start the installation process.



Microsoft .NET SDK 9.0.202 (x64) Installer

Microsoft .NET SDK 9.0.202

.NET SDK

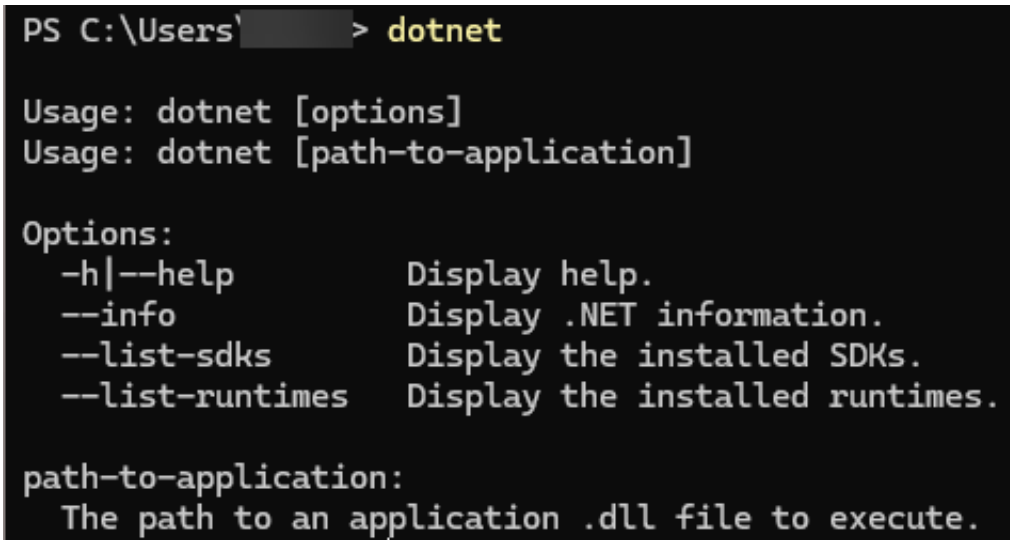
The .NET SDK is used to build, run, and test .NET applications. You can choose from multiple languages, editors, and developer tools, and take advantage of a large ecosystem of libraries to build apps for web, mobile, desktop, gaming, and IoT. We hope you enjoy it!

If you plan to use .NET 9.0 with Visual Studio, Visual Studio 2022 17.12 or newer is required. Learn more.

By clicking Install, you agree to the following terms:

Privacy Statement
Telemetry collection and opt-out
Licensing Information for .NET

You can then verify the installation by running dotnet in your Command Prompt/Terminal. If it is installed properly, you should see a list of options displayed.



```
PS C:\Users\[redacted]> dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help         Display help.
  --info            Display .NET information.
  --list-sdks       Display the installed SDKs.
  --list-runtimes   Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.
```

In this tutorial, we will guide you through the process of setting up C# in Godot, an open-source game engine. This guide assumes that you have a basic understanding of how to use the Godot engine, potentially with GDScript, but not necessarily with C#. If you have never used Godot with C#, there might be some setup required before you can get started.

Prerequisites

Before we dive into the setup process, let’s go over the prerequisites. You will need to have the following installed on your system:

- Godot .NET version
- .NET SDK
- Visual Studio Code (optional)

Godot .NET Version

An important thing to note about the Godot engine is that depending on the programming language you want to use, you’ll need to download a specific version of the engine. For C#, you’ll need to download the .NET version of Godot, regardless of the version you’re using. This is because C# is part of Microsoft’s .NET platform.

.NET SDK

Unlike GDScript, which is an integral part of the Godot engine, C# requires some external tools in the form of the .NET SDK. This includes the necessary compilers and libraries for running C# on your machine. The latest version of .NET that works with Godot as of 4.2 is .NET 7.0.

To check if you have a version of .NET installed on your system, you can use PowerShell on Windows. Open PowerShell and type .NET --version. If a version number is displayed, you have .NET installed. If the command does not work, you need to install the .NET SDK.

```
dotnet --version
```

Visual Studio Code

While it’s optional, we recommend using Visual Studio Code as your external editor for C#. This is because it offers excellent language support for C# and a range of helpful features that make it worth using outside of the Godot editor.

To link Visual Studio Code with the Godot editor, go to the Editor > Editor Settings > .NET > Editor > External Editor, and select Visual Studio Code from the dropdown.

Creating a C# Script in Godot

Now that everything is set up, you can create a C# script in Godot. Create a node and attach a new C# script to it. If everything is set up correctly, this should automatically open a new Visual Studio Code window with your script.

For testing purposes, in the _Ready() function, call the GD.Print function with a message. Save the script, run your project, and you should see your message printed in the console.

```
public override void _Ready()
```

```
{
    GD.Print("Hello, Godot C#");
}
```

Congratulations! You’ve successfully set up C# in Godot. In the next tutorials, we’ll dive into creating a game using C# in Godot. Stay tuned!

In this tutorial, we'll be starting the process of creating a hex-based strategy game in Godot C#. We'll begin by laying the foundation for our map. This tutorial is aimed at beginners, but it's important to note that we'll be using Godot version 4.3.

Setting up the Root Node

Every Godot project needs a root node, which serves as the entry point for the game. To create one:

1. Open a new project in Godot.
2. Click on "Other Node".
3. Select the generic node type and name it "Game".
4. Save it as a new scene (Ctrl + S), naming it "Game.TSCN".

With our root node set up, we can now move on to building our map.

Setting up Tile Maps

Tile maps in Godot 4.3 have undergone significant changes. Instead of a single tile map object, we now have a series of tile map layer nodes. While this might seem like extra work, it makes managing and composing different layers of tile maps much easier.

To begin with, we'll create a base node to represent our map as a whole. Add a new child to your game node, creating a Node2D and naming it "HexTileMap".

Creating Tile Map Layers

We'll be adding different layers to our tile map. For the basic map implementation, we'll need two layers:

- Base Layer: represents the colors of the terrain of the map.
- Hex Borders: displays a border on all of our hexagons.
- Selection Overlay: allows for things like clicking on our tile map and having it change color.

To create these layers, add a child node and search for "TileMapLayer". Avoid using the deprecated TileMap node. Name the new child nodes "BaseLayer", "HexBorders", and "SelectionOverlayLayer" respectively.

Configuring Tile Map Layers

Each tile map layer needs to be configured. Head over to the "BaseLayer" and take a look at the options in the inspector. You'll need to create a tile set for each layer, which contains all the data for the different tiles available for that layer. Change the tile shape from square to hexagon and change the tile size to 128 by 128 pixels.

Repeat this configuration for the other two tile map layers.

With that, we have our basic tile map layers set up. In the next part of this tutorial, we'll look at adding textures to them so we can start rendering our maps.

In the previous lesson, we set up our game root node, our hex tile map node 2D, and added three minimally configured tile map layers. These layers represent our map as a whole, and we switched them to hexagons and made them the right size. In this lesson, we will continue setting up the map by loading some textures into our tile sets. This will allow us to have some tiles to place on the map. Furthermore, we will create the skeleton of our terrain generation function, which will start to create the outline of our map.

Loading Textures into Tile Sets

The first thing that any tile map layer and tile set needs before we can use it is a tile atlas. A tile atlas is a PNG file or some other image texture file that contains the textures for the tiles that you want in the game. To add a tile atlas:

1. Select the base layer and head to the tile set section.
2. Underneath the tiles section, you'll see an empty space where you can add your tile set source.
3. Drag in the textures for your base terrain tiles from the course materials.
4. Create a textures folder in the resources section on the side of the dock for future organization.
5. Create a new Atlas and open up the textures folder.
6. Choose to automatically create tiles in the atlas.

At this point, our tile set has read in the PNG and parsed it as a tile atlas. If you hover over the base tiles, you can see that each one is now individually selected as a potential tile. Zooming in, you can see the outline of the hexagon that will be the actual size of the tile and the texture on top of that tile.

Setting Up Procedural Generation

While we could potentially paint these tiles onto our tile set manually, what we want eventually for our map is something that we can procedurally generate. This way, we can have different maps every time we play the game. To make that happen, we'll need to lay the foundation in code to populate our map with tiles programmatically.

First, we'll create our first script of the project on our HexTileMap node. We'll attach a new C# script to it. Inside of our new C# script for our HexTileMap node, we'll aim to populate our map with a single type of tile with a given width and height that we specify in the inspector. This will set some basic parameters for our map that we'll continue to develop to create some nice terrain generation over the rest of this course.

Exporting Map Width and Height

To achieve this, we'll need to export a couple of properties that will determine the width and height of our map. We'll create two properties, one for the width and one for the height, and preface them with the export directive. This will ensure that these properties show up in the Godot inspector so that we can set them from the editor. Here is an example of how to do this:

```
[Export]
public int width = 100;

[Export]
public int height = 60;
```

Generating the Terrain

Next, we’ll start creating the skeleton of our function that’s going to handle all of the generation of the terrain on our map. We’ll create a new void function called GenerateTerrain.

```
public void GenerateTerrain()
{

}
```

To make this work, we’ll need access to a few resources. In our hex tile map’s ready function, we’ll get a hold of references to all of our tile map layers. We’ll need these references to be able to populate them with the tiles we want. To do this:

1. Declare three tile map layer variables: baseLayer, borderLayer, and overlayLayer.
2. Inside of the ready function, call getNode for each of the declared variables to get a reference to it.

```
// Map data
TileMapLayer baseLayer, borderLayer, overlayLayer;

public override void _Ready()
{
  baseLayer = GetNode<TileMapLayer>("BaseLayer");
  borderLayer = GetNode<TileMapLayer>("HexBordersLayer");
  overlayLayer = GetNode<TileMapLayer>("SelectionOverlayLayer");
}
```

Now that we have references to these nodes, we’re ready to do some basic terrain generation in our function. In the next video, we will iterate through each of the ready that we specify here.

In this lesson, we will continue setting up our basic terrain generation for a hexagonal tile map in Godot. We aim to generate a map of the desired proportions specified in the Godot inspector, using a single type of tile on the base layer. This will lay a solid foundation for more advanced terrain generation in the upcoming lessons.

Review of Previous Steps

So far, we have completed the following steps in our hex tile map C# code:

- Exported two important variables for our map to the Godot inspector: the width and the height of the map.
- Obtained references to our three tile map layers: the base layer, the border layer, and the overlay layer.

For now, we will only work with the base layer and test out a simple terrain generation setup. This will help us understand the process before we move on to more complex terrain generation tasks.

Generating the Map

When generating the map, we need to assign a value to each possible tile on the map. This means we need to visit every single spot on the map at least once during the generation process. The easiest way to achieve this for a square map is to use nested for loops. One loop will iterate through the X direction, and the other will iterate through the Y direction. Here's how you can set it up:

```
// Outer loop for X direction
for (int x = 0; x < width; x++)
{
  // Inner loop for Y direction
  for (int y = 0; y < height; y++)
  {
    // Code for terrain generation goes here
  }
}
```

Setting the Cells

Next, we need to set the cell on our tile map layer for each of the XY coordinate pairs that our loop visits. We will use the `setCell` function on our tile map layer object for this. This function, as per the Godot documentation, sets the tile for a cell at any given coordinate pair.

The `setCell` function takes in several arguments:

- The first argument is a vector of integers representing the x, y coordinates on the tile map where we're setting this cell.
- The next parameter is the source ID, which indicates what tile atlas on our tile set we're drawing from. For us, this is usually going to be zero because we only have one tile set atlas, and the default first ID is zero.
- The last parameter we'll be specifying here is the atlas coordinate. This represents where in the tile atlas we want to draw this particular texture from. It tells which texture in our atlas we want this tile to be set to.

Here's how you can use the `setCell` function:

```
// Set the cell for each XY coordinate pair
baseLayer.SetCell(new Vector2I(x, y), 0, new Vector2I(0, 0));
```

With this, we have everything we need to generate a bit of terrain and see how this works on our map.

Running the Map

After writing the code, it’s time to test our map in the Godot inspector. However, since C# is a compiled language, we need to build the project first before Godot can register the changes we’ve made to our C# code. Click on the ‘Build’ button to build the .NET project.

Once the project is built, you can set the width and height values for the map. Let’s set a modest size of 10×10 for testing purposes. Now, run the scene and you should see a nice 10×10 grid of tiles.

In this lesson, we’ve taken the first steps towards generating a hexagonal map for a strategy game. In the next lesson, we will add borders to our hexes for better visualization and delve deeper into procedural terrain generation.

In the previous lesson, we generated the first part of our map using hexagonal tiles. Before we delve into more detailed terrain generation or camera handling, we'll take a detour to add a small but aesthetically significant feature to our game. We'll add a tile map layer to contain thin black borders for all our tiles. This will visually separate different tiles, creating a cleaner and nicer look for our game.

Setting Up the Tile Map Layer

In a previous lesson, we set up the necessary tile map layer for this. The ordering of these tile maps is important. In Godot 4.3, individual nodes replaced what was in previous versions an attribute on a single tile map node. If you've used tile maps before, you'll know that you could order these layers in the inspector so that one layer is on top of another. However, in this new way of doing things in Godot 4.3 and going forward, the layer of the tile map is determined by how they're ordered in the node tree. So, the ordering in the node tree matters here.

Adding a Texture for Our Border

Before we can programmatically assign our tile borders to all of our tiles, we need an actual texture for our border. Just like we did with our base layer, we'll drag in a texture atlas to our tile set source for this particular tile map layer. In the course materials, you'll find a file called hex overlays that has two textures, one for the hex tile border and one for the hex tile overlay. We'll drag that into our textures folder here in the file system. Then we can drag that into our hex borders layer tile set. Note here that when this generated this tile atlas it did generate one extra tile. If you want to get rid of that, all you have to do is grab the eraser and just get rid of it. Then click here on these three dots and select remove tiles in fully transparent texture regions.

Assigning the Border Texture

Now that we have the tile set texture atlas ready, we can go into our code and assign this border texture at atlas coordinates 0, 0 to all of our tiles. We're going to assign these in our generate terrain function. Every tile, regardless of what terrain type it ends up being, is going to need a border. So, inside of our nested loop here, for iterating through the width and height of our map, for every one of these XY coordinates, we're going to assign a tile border texture to our border layer.

You may be wondering, what is the point of separating these two out? Why not just have a texture that has the border included? One possible advantage to this sort of architecture is that if in the future you want to replace the border graphic with something else, separating these two allows that to happen very easily. If they were not separate, then that would require you to redo whole sections of textures for your game.

Finally, to set this, we'll add a line of set cell code in our generate terrain function. We're going to take our border layer and do the same thing we did above with our base layer. We're going to call setCell. We're going to set the cell of a new vector 2, containing the XY coordinates that we're currently at, iterating through the map. We know that the ID of this atlas is also going to be 0, since it was the first one and the only one we made for this tilemap layer. Then finally, we know that the atlas coordinates of the border texture that we want are going to be 0, 0.

```
borderLayer.SetCell(new Vector2I(x, y), 0, new Vector2I(0, 0));
```

After saving your code, build and run it. You should now see that your tiles have cleaner and nicer looking borders.

In the previous tutorial, we developed a hex grid tile map filled with a single terrain texture and a border layer for a clean look. However, the map is static and we cannot pan around or interact with it. This becomes a problem when we generate a map larger than the screen display. Hence, before we move on to terrain generation, we need to implement some camera controls to allow us to pan around the map. This is a crucial aspect of any top-down strategy game.

Creating a Camera Node

The first step in implementing camera controls is creating a camera node. Godot provides us with a built-in Camera2D node that can handle a lot of the work for us. To create a camera node:

1. Underneath your game node, add a child and find the Camera2D.
2. Create a new Camera2D and rename it to ‘camera’.

Upon creating the camera, you’ll notice a pink outline on the screen representing the field of view of the camera at its current position at the origin. Unlike most things in Godot, where the origin is at the top left, the camera’s origin is at the center. This is because it is more intuitive to think of cameras as originating from the center of the screen rather than from the top left.

Setting Up the Input Map

Before we create our camera script to control it, we need to set up some things in the input map. The input map is where we can define different actions for user input and bind them to keys. This is much cleaner and flexible than having to pull directly for whatever key or mouse movement is being used.

For our map controls, we need six basic controls: moving the map left, right, up, and down, and zooming the map in and out. We will bind these to keys on the keyboard and also do some zooming with the mouse wheel. More complex map movement or camera movement, such as dragging or moving the mouse cursor to the edge of the screen, is beyond the scope of this tutorial. We will stick to simple keybinds for now.

To add these actions:

1. Go to Project Settings.
2. Select the Input Map tab.
3. Add new actions for moving the map left, right, up, and down, and for zooming in and out.
4. Also add two extra actions for using the mouse to zoom in and out.

```
Map Left: A key
Map Right: D key
Map Up: W key
Map Down: S key
Zoom In (Keyboard): =/+ key
Zoom Out (Keyboard): - key
Zoom In (Mouse): Mouse Wheel Up
Zoom Out (Mouse): Mouse Wheel Down
```

With the camera node and input map set up, we are ready to create our camera script to control it, which we will cover in the next tutorial.

In this lesson, we will start implementing our camera script in Godot. This script will allow us to control the camera's movement and zoom on a 2D map. We will begin by exporting a couple of properties that we need to adjust the performance of our camera, namely the speed of the camera (which we will call velocity) and the speed of zooming the camera (which we will call zoom speed).

Creating the Camera Script

To start, we need to attach a new script to our camera. To do this, select the camera in the Godot editor and hit "Attach Script". Make sure that it's a C# script and name it "camera.cs".

Now, let's head over to VS Code and start coding our camera controls. Inside our new camera class, we will define the two export variables mentioned above:

```
[Export]
int velocity = 15;
[Export]
float zoom_speed = 0.05f;
```

The purpose of the export keyword here is to make sure that these variables appear in the Godot inspector. This allows us to edit them directly from the editor. The default velocity is set to 15 and the zoom speed is set to 0.05. We use a float and a small number for the zoom speed because large numbers tend to make the zoom unreasonably fast. Thus, 0.05 is a calibrated default that provides a smoother zooming experience.

Implementing Camera Panning

With our velocity in place, we're ready to implement the first functionality for our camera – panning. In this lesson, we will implement the panning to the right function, leaving the implementation of left, up, and down to you as an exercise.

It's important to note that we're not going to use the default process function to handle these events. Instead, we're going to use a function called physics process. Physics process is not set by default in these scripts, so we'll need to change our function to physics process:

```
public override void _PhysicsProcess(double delta)
```

Physics process is useful because it ensures that it's called at a stable tick rate. This is important for preventing anomalies and errors with the physics engine, which requires a stable rate at which the function is called. For the camera, this is crucial to avoid any potential stuttering or hiccups, ensuring a smooth camera movement.

Implementing the Right Panning Function

To implement moving the camera to the right, we will first poll our input events to check if the right action has been pressed. If it has, we want to change the position of the camera. Since the camera inherits from node 2D, it has a position attribute. Since we're moving on the x-axis by moving right, we will adjust the position's x value:

```
if (Input.IsActionPressed("map_right"))
```

```
{
  this.Position += new Vector2(velocity, 0);
}
```

Now, if you go back to the Godot editor and run your project, you should be able to move the camera to the right. This is a good starting point for our camera controls. The implementation of the right panning function also gives you a good foundation to implement the other directions for the camera.

In the next lesson, we'll go over how to implement zooming and setting boundaries for our camera. This will give our camera more functionality and make our 2D map more interactive and user-friendly.

In the previous lesson, we started implementing our camera controls in Godot. We filled in the math for panning the camera right and left it as a challenge to implement the up, down, and zoom controls. In this lesson, we'll continue where we left off and complete the implementation of our camera controls.

Implementing the Remaining Camera Controls

To implement the remaining camera controls, we can start by copying the if block we created for panning right and pasting it three times. These will serve as the structure for our other directions. We'll fill in these blocks with the appropriate code for each direction.

For panning left, we only need to make the velocity vector negative. This ensures that we move in the opposite direction on the x-axis:

```
if (Input.IsActionPressed("map_left")) {
    this.Position += new Vector2(-velocity, 0);
}
```

For panning up and down, we apply our velocity to the y-axis. Keep in mind that the up direction is negative on the y-axis in 2D graphics:

```
if (Input.IsActionPressed("map_up")) {
    this.Position += new Vector2(0, -velocity);
}

if (Input.IsActionPressed("map_down")) {
    this.Position += new Vector2(0, velocity);
}
```

With these additions, our basic implementation of panning for the camera should be working. You can test this in the Godot Inspector.

Smooth Camera Movement

To make our camera movement smoother, we can enable position smoothing. This feature allows the camera to slow down to a stop in a smooth fashion rather than abruptly stopping when the player releases the key. You can enable this in the Inspector and set the speed of the stopping to about 10 pixels per second. Running the game again, you should notice that the camera movement is a lot more fluid and responsive.

Implementing Zoom Controls

Next, we need to implement zooming for our camera. We'll start by handling zooming with the keyboard. We've already assigned "map_zoom_in" and "map_zoom_out" as actions in our input map, so we'll need to pull "is_action_pressed" for these actions:

```
if (Input.IsActionPressed("map_zoom_in")) {
    // Zoom in code
}
```

```
if (Input.IsActionPressed("map_zoom_out")) {
    // Zoom out code
}
```

Similar to panning, we'll add our zoom speed value to the current value of our zoom. Camera2D nodes have a property called zoom that we can use directly:

```
if (Input.IsActionPressed("map_zoom_in")) {
        this.Zoom += new Vector2(zoom_speed, zoom_speed);
}

if (Input.IsActionPressed("map_zoom_out")) {
        this.Zoom -= new Vector2(zoom_speed, zoom_speed);
}
```

Now, you should be able to zoom in and out using the keyboard. However, you may notice that you can zoom in too close or zoom out too far. To limit the zoom level, we'll add boundary checks on our zooming:

```
if (Input.IsActionPressed("map_zoom_in")) {
    if (this.Zoom < new Vector2(3f, 3f))
        this.Zoom += new Vector2(zoom_speed, zoom_speed);
}

if (Input.IsActionPressed("map_zoom_out")) {
    if (this.Zoom > new Vector2(0.1f, 0.1f))
        this.Zoom -= new Vector2(zoom_speed, zoom_speed);
}
```

With these boundaries, our zooming should now be limited and prevent any errors. You can test this in the Godot Inspector.

To sum up, we have now implemented basic camera controls for our strategy game, including panning and zooming. In the next lesson, we will add mouse wheel scroll zoom and set boundaries for panning on the map.

In this lesson, we are going to finalize our implementation of the camera in our Godot game. We will add in a zoom functionality using the mouse scroll wheel, and set boundaries for our camera panning to ensure we can't pan away from the map completely.

Mouse Scroll Wheel Zooming

After implementing the zoom controls in the previous lesson, you might recall that we defined mappings for the mouse scroll wheel in the input map. We want the user to be able to zoom with the scroll wheel as an alternative to the keyboard.

However, the way Godot handles mouse wheel events is different from how it handles keyboard events. Therefore, we can't use the same 'input map action pressed' method for the mouse wheel. Instead, we need to use the 'isActionJustReleased' method.

To implement mouse wheel zooming, we're going to create a system that tracks whether the mouse wheel is currently scrolling and call the zoom controls accordingly.

Creating Mouse Wheel Zooming System

The first part of this system involves creating two boolean variables in our camera class, which we'll call 'mouseStates'. One variable will track whether the mouse wheel is scrolling up, and the other will track whether it's scrolling down. We'll initially set both variables to false:

```
bool mouseWheelScrollingUp = false;
bool mouseWheelScrollingDown = false;
```

Next, we need to use the 'isActionJustReleased' function to check the state of our mouse wheel input and update our variables accordingly. After that, we'll add a check for these variables inside our already existing zoom code to finalize our mouse wheel zoom.

Checking Mouse Wheel Zooming

We need to check whether our mouse is zooming in or out. We'll need four checks in total: one to check if the mouse wheel zoom in has started, one to check if it has stopped, and two similar checks for zooming out.

Here's how to check if the mouse wheel zoom in has started:

```
if (Input.IsActionJustReleased("mouseZoomIn"))
    mouseWheelScrollingUp = true;
```

And here's how to check if it has stopped:

```
if (!Input.IsActionJustReleased("mouseZoomIn"))
    mouseWheelScrollingUp = false;
```

We'll repeat this pattern for zooming out:

```
if (Input.IsActionJustReleased("mouseZoomOut"))
    mouseWheelScrollingDown = true;

if (!Input.IsActionJustReleased("mouseZoomOut"))
    mouseWheelScrollingDown = false;
```

Now, we have all the data we need for our mouse wheel scrolling actions and states. We can check for these inside our existing zoom controls by amending the statements to include an 'OR' operator:

```
if (Input.IsActionPressed("map_zoom_in") || mouseWheelScrollingUp)
    // Zoom in code here

if (Input.IsActionPressed("map_zoom_out") || mouseWheelScrollingDown)
    // Zoom out code here
```

With these changes, our game now supports zooming in and out using the mouse scroll wheel, providing a smoother user interface for controlling our map.

Setting Camera Panning Boundaries

The final task for this lesson is to set boundaries for our camera panning. Currently, we can pan indefinitely and end up losing sight of our map. We want to restrict the camera to the boundaries of our map.

We'll cover how to implement these boundaries in the next lesson.

Welcome back! In this final video on setting up our camera, we'll be taking care of setting boundaries on the panning of our map. To do this, we're going to need to head back to our Camera class and also make a slight addition to our HexTileMap class.

Understanding the Problem

The fundamental problem we're going to need to solve here is we're going to need the camera to somehow know where the boundaries of the map are. Remember, since our map abstraction is just a set of tile map layers, we don't really have any precise way to do this, especially not in the coordinate system that the camera is going to require. Our camera is going to operate on 2D world coordinates, so we're going to need a way to translate between the coordinate system of our TileMap and the coordinate system of the actual Godot 2D world.

Designing the Solution

First of all, we're going to need to keep track of four different values for the boundaries of the map, one for each direction. So we're going to add some new class variables to the Camera class. We will call these map boundaries, and these are going to be floating point numbers. We'll just have one called leftBound, rightBound, topBound, and bottomBound.

```
public partial class Camera : Camera2D
{
  // Map boundaries
  float leftBound, rightBound, topBound, bottomBound;

  // Rest of the code...
}
```

In our ready function, we're going to need to actually assign these to some values, calculate these based on our map. So the first thing we're going to need to do is we're going to need to get a reference to our map so that we can have access to the data that we'll need to create these boundaries. This will create a reference to our map. There's a type HexTileMap.

```
public partial class Camera : Camera2D
{
  // Map boundaries
  float leftBound, rightBound, topBound, bottomBound;

  // Map reference
  HexTileMap map;

  public override void _Ready()
  {
    map = GetNode<HexTileMap>("../HexTileMap");

    // Rest of the code...
  }
}
```

Calculating Boundaries

Now we're ready to actually calculate our boundaries. To do this, we're going to use a function called toGlobal. This function transforms a coordinate provided in a local coordinate space. So let's say coordinates that are relative only to a particular hex tile map layer, and this translates them to the global 2D coordinates of the Godot 2D world that the camera is going to need to orient itself.

However, we're going to need to actually figure out a way to get the local coordinates out of our TileMap. And this is where we're going to need to write a little utility function inside of our TileMap.

```
public class HexTileMap : Node2D
{
  // Rest of the code...

  public Vector2 MapToLocal(Vector2I coords)
  {
    return baseLayer.MapToLocal(coords);
  }
}
```

Now, we can use this function in our Camera class to set our boundaries.

```
public partial class Camera : Camera2D
{
  // Map boundaries
  float leftBound, rightBound, topBound, bottomBound;

  // Map reference
  HexTileMap map;

  public override void _Ready()
  {
    map = GetNode<HexTileMap>("../HexTileMap");

    leftBound = ToGlobal(map.MapToLocal(new Vector2I(0, 0))).X + 100;
    rightBound = ToGlobal(map.MapToLocal(new Vector2I(map.width, 0))).X - 100;
    topBound = ToGlobal(map.MapToLocal(new Vector2I(0, 0))).Y + 50;
    bottomBound = ToGlobal(map.MapToLocal(new Vector2I(0, map.height))).Y - 50;
  }
}
```

Applying Boundaries to the Panning Code

Now we have calculated boundaries of our map, and we need to actually apply them to our panning code. So what we have so far is panning code that only checks whether our map right, for instance, key is pressed, and if it is, then it will update the position accordingly. We also need to add to this check a check for whether the position of the camera is actually within bounds.

```
public partial class Camera : Camera2D
```

```
{
  // Rest of the code...

  public override void _PhysicsProcess(double delta)
  {
    // Map controls
    if (Input.IsActionPressed("map_right") && this.Position.X     {
      this.Position += new Vector2(velocity, 0);
    }

    if (Input.IsActionPressed("map_left") && this.Position.X > leftBound)
    {
      this.Position += new Vector2(-velocity, 0);
    }

    if (Input.IsActionPressed("map_up") && this.Position.Y > topBound)
    {
      this.Position += new Vector2(0, -velocity);
    }

    if (Input.IsActionPressed("map_down") && this.Position.Y     {
      this.Position += new Vector2(0, velocity);
    }

    // Rest of the code...
  }
}
```

With that, we have effectively established some boundaries for our camera. Now we have all the tools we need to really dive into terrain generation starting in the next video.

In this lesson, we're going to focus on setting up the basic utilities and data structures necessary for terrain generation in a hex-based strategy game using Godot with C#. We'll create an enumeration (enum) for different types of terrains and a new class to represent hexes on the map. Let's dive in.

Creating the Terrain Type Enumeration

First, we'll create an enum to represent all the different types of terrain in our game. Enumerations in C# are a way to assign constant values to labels, making your code more readable and easier to work with. For our game, we'll have different types of terrain such as plains, forests, deserts, mountains, water, beach, and ice. Here's how we can create the enum:

```
public enum TerrainType { PLAINS, WATER, DESERT, MOUNTAIN, ICE, SHALLOW_WATER, FOREST
, BEACH }
```

In this enum, each terrain type is associated with an integer, but we use the labels for clarity in our code.

Creating the Hex Class

Next, we'll create a new class called Hex. This class will represent a hex on the map in terms of gameplay. It will store data like the hex's coordinates and terrain type, and make it easier for us to pass hexes around between functions. Here's how we can define the Hex class:

```
public class Hex
{
    public readonly Vector2I coordinates;
    public TerrainType terrainType;

    public Hex(Vector2I coords)
    {
        this.coordinates = coords;
    }
}
```

In this class, we declare a read-only Vector2I to store the hex's coordinates. We make it read-only because a hex's coordinates should never change once it's created on the map. We also declare a TerrainType to store the type of terrain of the hex.

Setting Up the Hex Tile Map Class

Now, we'll head back to our HexTileMap class and create a data structure to store hexes and represent our map. We'll use a dictionary that maps integer vectors (representing hex tile map coordinates) to actual Hex objects:

```
Dictionary<Vector2I, Hex> mapData;
```

We’ll initialize this dictionary in the ready function of the HexTileMap class:

```
public override void _Ready()
{
    mapData = new Dictionary<Vector2I, Hex>();
}
```

With these utilities and data structures in place, we’re now ready to proceed with terrain generation using Perlin noise. In the next lesson, we’ll look at how to map terrain types to their texture in the texture atlas for easy access in our code, and then we’ll delve into using Perlin noise for terrain generation.

In this lesson, we'll continue setting up our boilerplate code for terrain generation in Godot. We'll specifically focus on managing the mapping of our terrain types to their corresponding textures in our texture atlas.

Terrain Type Enumeration and Texture Atlas

Each of the terrain types we created in our enumeration has a corresponding texture in our texture atlas. To avoid hardcoding the atlas coordinates throughout our codebase, we can define them in one place and map these coordinates to a readable enumeration label. This way, instead of typing in the Atlas coordinates every time we want to render a tile, we just need to remember the type of tile we want. This will be managed in one place, making our code cleaner and more efficient.

Creating a Terrain Textures Dictionary

To achieve this, we will create a dictionary in our HexTileMap class, which we'll call 'terrainTextures'. This dictionary will map each terrain type to an integer vector representing the tile atlas texture coordinates.

```
Dictionary<TerrainType, Vector2I> terrainTextures;
```

We'll initialize this dictionary with some values inside of our 'ready' function. In C#, we can declare the values we want inside a dictionary when we declare or initialize it. Instead of using parentheses to create an empty dictionary, we can directly add some values using brackets.

Filling in the Dictionary

With our dictionary set up, we can now fill it with key-value pairs. The keys will be our terrain type enumeration values, and the values will be the coordinates in our texture atlas. We'll start with the 'Plains' type, mapping it to an integer vector (0, 0). This is because the light green hex we want to represent our plains terrain is on the top left of the atlas.

```
{ TerrainType.PLAINS, new Vector2I(0, 0) },
```

We'll continue this process for each of our terrain types. For instance, for 'Desert', we'll map it to the vector (0,1), as that's its location in the texture atlas. We'll continue this process for all our terrain types, mapping each to its corresponding texture atlas coordinates.

```
{ TerrainType.WATER, new Vector2I(1, 0)}
{ TerrainType.DESERT, new Vector2I(0, 1)},
{ TerrainType.MOUNTAIN, new Vector2I(1, 1)},
{ TerrainType.SHALLOW_WATER, new Vector2I(1, 2)},
{ TerrainType.BEACH, new Vector2I(0, 2)},
{ TerrainType.FOREST, new Vector2I(1, 3)},
{ TerrainType.ICE, new Vector2I(0, 3)},
```

While this process is tedious, it is necessary for mapping all of the individual terrain types to their necessary textures. It will also make our code cleaner and easier to manage in the long run.

With this, we have successfully set up our terrain textures dictionary, mapping each of our terrain types to their corresponding texture atlas coordinates. This will make it easier for us to render tiles in our terrain without having to manually input and remember all the texture atlas coordinates.

In this lesson, we are going to delve into the mechanics of generating procedural terrain for a hex-based strategy game in Godot using C#. Our tool of choice for this task is Godot's built-in noise generator, FastNoiseLite. This noise generator can create various types of random noise, which we will use to create diverse and interesting terrains for our game.

Understanding FastNoiseLite

FastNoiseLite is an instance that can be exported in a script attached to a node. For instance, to view it in the inspector, you could add it to a game node script. However, for our purposes, we will be generating and utilizing our FastNoiseLite instances entirely within our code, so we won't actually see it in the inspector.

The FastNoiseLite instance exposes several parameters that allow us to control the type of noise generated. Let's go through some of the main ones:

- **Noise Type:** This parameter determines the algorithm or set of algorithms FastNoiseLite uses to generate the random points. The different types of noise available include Perlin, Cellular, Value, and Simplex, each generating distinctly different patterns.
- **Seed:** The seed is a value that allows us to reproduce exact configurations of noise. Changing the seed value generates a completely new pattern, but the same seed value will always produce the same pattern. In our terrain generation, we will set this to a random value.
- **Frequency:** This parameter affects the density of the noise pattern. Higher frequency values result in denser, more "zoomed-out" looking patterns, while lower values result in less dense, more "zoomed-in" patterns.
- **Fractal Options:** These options give fine-grained control over the noise's appearance. They include different fractal types that can significantly alter the noise's look, the number of octaves that can make the noise appear blurrier, and lacunarity that affects the roughness of the fractal edges.

It's recommended to play around with these parameters to get an intuitive sense of how they affect the generated noise.

Generating Terrain with FastNoiseLite

Now that we understand FastNoiseLite, we can use it to generate our terrain. We will be doing this in our HexTileMap script. Here's an example of how we might set up our FastNoiseLite instance for generating terrain:

```
FastNoiseLite noise = new FastNoiseLite();

noise.Seed = seed; // A random seed value
noise.Frequency = 0.008f; // Adjust to suit your desired pattern density
noise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm; // Adjust to suit your desired
 noise pattern
noise.FractalOctaves = 4; // Adjust to suit your desired blurriness
noise.FractalLacunarity = 2.25f; // Adjust to suit your desired roughness
```

With this setup, we can now generate a 2D array of noise values that we can use to determine the terrain type for each tile in our hex map. We can generate different types of terrain by using different FastNoiseLite configurations and applying them to different ranges of tiles.

In the next lesson, we will start implementing this in our code to generate our game’s terrain.

In this lesson, we will continue to work with FastNoiseLite to generate terrain for our map. Previously, we have experimented with some parameters of FastNoiseLite and gained an understanding of how it works. Now, we will use these objects to generate values and apply them as terrain to our map. We will be using Visual Studio Code and working within our HexTileMap class, inside our generateTerrain function.

Setting Up Data Structures to Hold Noise Values

Firstly, we need to set up some data structures to hold the noise values that we generate. We will generate four different runs of FastNoiseLite noise. The first run will be for the basic structure of our continents and oceans on our map, and the other three will be for populating the continents with interesting features like forests, deserts, and mountains. All of these will be stored in two-dimensional arrays of floating-point numbers, as noise generates as floats.

```
float[,] NoiseMap = new float[mapWidth, mapHeight];
float[,] ForestMap = new float[mapWidth, mapHeight];
float[,] DesertMap = new float[mapWidth, mapHeight];
float[,] MountainsMap = new float[mapWidth, mapHeight];
```

Here, we declare a new 2D array of floats for each type of terrain feature we want to generate. Each array is instantiated with the same dimensions as our map, as each position in the array will represent a tile on our map.

Creating FastNoiseLite Objects

With our data structures in place, we can now go ahead and create our FastNoiseLite objects. We will first create a random number generator, which we will use several times throughout our terrain generation. In C#, we can generate random numbers with a Random object.

```
Random random = new Random();
int seed = random.Next(100000);
```

We use the random number generator to create a single seed value. This seed value is randomly generated each time we generate terrain. However, it ensures that all of our FastNoiseLite objects are running with the same seed. This allows our terrain generation to be reproduced, even though we are using different FastNoiseLite instances for each type of terrain.

Configuring FastNoiseLite Objects

Now that we have our seed value, we can start generating our base terrain, which will consist of water, beach, and plains. This will establish the general shape of the terrain on our map with continents and oceans. We will try this first, and then later we will come back and generate the other kinds of terrain.

The basic procedure for setting up a generation is as follows:

1. Create a new instance of FastNoiseLite.
2. Configure the instance with the desired parameters.
3. Generate the noise.

Let's start by creating a new instance of FastNoiseLite:

```
FastNoiseLite noise = new FastNoiseLite();
```

Next, we configure the instance with the desired parameters. We set the seed to our seed value. For the other parameters, we will use some pre-configured values that have been tested to create interesting but still playable continents and oceans.

```
noise.Seed = seed;
noise.Frequency = 0.008f;
noise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
noise.FractalOctaves = 4;
noise.FractalLacunarity = 2.25f;
```

Finally, we will need one more variable, noiseMax, which we will set to zero for now. This variable will keep track of the maximum value of the noise in a given generation. We will need this to accurately establish different ratios or sections of the noise range that we want to assign to different terrain values.

```
float noiseMax = 0f;
```

In the next lesson, we will actually generate the noise and see what it looks like on our map.

In this lesson, we'll continue generating the base terrain for our map. We'll configure a FastNoiseLite and iterate through the width and height of our map to generate noise values for each of the tiles. We'll then use these noise values to assign terrain values to different sub-ranges of the noise, creating a varied landscape for our game.

Creating a Separate Loop for Noise Generation

To generate our noise values, we'll create a separate for loop to iterate through the dimensions of the map. This is because we need to have the full set of noise generated for us to be able to accurately calculate the ratios and ranges of the noise that we'll need to assign terrain values to different sub ranges of the noise.

An important note here is that we're not doing this in the main loop because we need all the noise values set up and calculated before we can apply them in the main loop that we've already written.

Here's how we can write this separate loop for generating noise values:

```
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {
    // Noise generation code goes here
  }
}
```

Storing Noise Values

Inside the loop, we'll start using the data structures that we created earlier to store noise values. We'll take the absolute value of the noise at a given spot in our map and store it in our noise map. We're taking the absolute value to avoid dealing with negative numbers, which doesn't affect the generation but makes it easier for us to handle.

To get the noise value at a particular spot, we'll call a function on our FastNoiseLite called `getNoise2D`. This function takes an x and y coordinate and returns the noise value at that particular pixel.

Here's how we can do this:

```
noiseMap[x, y] = Math.Abs(noise.GetNoise2D(x, y));
```

Tracking the Maximum Noise Value

As we're generating the noise, we'll need to keep track of what the maximum value of our noise is. This is because we'll need this maximum value later when we start calculating our ratios for the terrain types.

We can do this by checking if the current noise value is greater than the maximum value we've seen so far and if it is, we update our maximum value to this new value.

```
if (noiseMap[x, y] > noiseMax) {
  noiseMax = noiseMap[x ,y];
}
```

Defining Terrain Ratios

Now that we have our noise values, we’ll need to map these values to different terrain types. For this, we’ll define a list of ranges for each terrain type. Each range will have a minimum and maximum value, and a terrain type associated with it.

Here’s how we can define this list:

```
List<(float Min, float Max, TerrainType Type)> terrainGenValues = new List<(float Min
, float Max, TerrainType Type)>{
  (0, noiseMax/10 * 2.5f, TerrainType.WATER),
  (noiseMax/10 * 2.5f, noiseMax/10 * 4, TerrainType.SHALLOW_WATER),
  (noiseMax/10 * 4, noiseMax/10 * 4.5f, TerrainType.BEACH),
  (noiseMax/10 * 4.5f, noiseMax + 0.05f, TerrainType.PLAINS)
};
```

With this, we’ve defined the ranges of noise values that each terrain type should have and can now proceed to generate the terrain and render it on the map.

In this lesson, we are going to generate the base oceans and continents for our terrain. We'll use the utility functions we created earlier for generating noise and calculating terrain ratios. Let's begin by jumping back into the main loop where we set our cells.

Creating a New Hex

The first thing we need to do when generating terrain for a new tile is to create a new hex. This is done by instantiating a new hex object in the same loop where we set our cells. We will pass the coordinates of our current position in the map (represented by a Vector2) to the hex.

```
Hex hex = new Hex(new Vector2I(x, y));
```

Getting Noise Value

Next, we need to calculate the noise value for the current position in the map. This value will be used to determine the type of terrain for our hex.

```
float noiseValue = noiseMap[x, y];
```

Setting the Terrain Type

We have a list of ranges that we calculated earlier, which will be used to determine the type of terrain based on the noise value. We can do this using a feature in C# called Language Integrated Query (LINQ). This allows us to query our data in a more readable and concise way.

We will use the `First` method from LINQ, which returns the first element in a sequence that satisfies a specified condition. In our case, we want to find the first range where the noise value is greater than or equal to the minimum value of the range and less than the maximum value of the range. The type of this range will be our terrain type.

```
hex.terrainType = terrainGenValues.First(range => noiseValue >= range.Min
                                    && noiseValue < range.Max).Type;
```

Adding Hex to Map Data

Once we have set the terrain type for our hex, we can add it to our map data at the current coordinates.

```
mapData[new Vector2I(x, y)] = hex;
```

Setting the Cell in Tile Map

Finally, we need to set the cell in our tile map to reflect the terrain type of our hex. We can do this by using the `TerrainTextures` dictionary that we created earlier. This dictionary maps terrain types to texture coordinates in our tile atlas. We can use the terrain type of our hex to get the appropriate texture coordinate.

```
baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[hex.terrainType]);
```

With all these steps in place, we should now be able to see some terrain generation in our map. Each time we run the project, a new random terrain should be generated.

Next Steps

In the next lesson, we will add more features to our terrain such as forests, deserts, and mountains. We will also add ice caps to the map as a final touch. Stay tuned!

In the previous lesson, we generated the base terrains for our game, which included oceans and continents. Now, we are going to extend this by generating three other types of terrain: forest, desert, and mountain. This process will involve a lot of repetition and copying of the previous configurations, with slight modifications to ensure each terrain type has its unique feel and look.

The first step is to generate some forest on top of our plains. For this, we'll need a new FastNoiseLite configuration. We'll use a different type of noise this time – cellular noise, which gives a more splotchy forest-like look to the terrain.

Generating Forest Terrain

```
FastNoiseLite forestNoise = new FastNoiseLite();

forestNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.Cellular;
forestNoise.Seed = seed;
forestNoise.Frequency = 0.04f;
forestNoise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
forestNoise.FractalLacunarity = 2f;

float forestNoiseMax = 0f;
```

Next, we'll generate the desert terrain. For the desert, we want large swaths of smooth land, so we'll use simplex smooth noise.

Generating Desert Terrain

```
FastNoiseLite desertNoise = new FastNoiseLite();

desertNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.SimplexSmooth;
desertNoise.Seed = seed;
desertNoise.Frequency = 0.015f;
desertNoise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
desertNoise.FractalLacunarity = 2f;

float desertNoiseMax = 0f;
```

Now that we have our configurations set up for forest and desert, we need to actually generate these noise values. This process will look identical to how we generated base terrain.

```
// Forest
forestMap[x, y] = Math.Abs(forestNoise.GetNoise2D(x, y));
if (forestMap[x, y] > forestNoiseMax) forestNoiseMax = forestMap[x ,y];

// Desert
desertMap[x, y] = Math.Abs(desertNoise.GetNoise2D(x, y));
if (desertMap[x, y] > desertNoiseMax) desertNoiseMax = desertMap[x ,y];
```

We’re now ready to establish some sub-ranges for this. We’ll create a Vector2 to hold the range that we want our forest and desert to generate in.

```
// Forest gen values
Vector2 forestGenValues = new Vector2(forestNoiseMax/10 * 7, forestNoiseMax + 0.05f);
// Desert gen values
Vector2 desertGenValues = new Vector2(desertNoiseMax/10 * 6, desertNoiseMax + 0.05f);
```

Finally, we’re ready to actually generate these terrains. We’re going to make sure that desert is not generating in the middle of the ocean. We need to ensure that it’s only going to generate on top of plains tiles to make sure that it is squarely within our continents.

```
// Desert
if (desertMap[x, y] >= desertGenValues[0] &&
  desertMap[x, y] <= desertGenValues[1] &&
  h.terrainType == TerrainType.PLAINS)
{
  h.terrainType = TerrainType.DESERT;
}
```

As a challenge, try to fill in the generation code for the forest and configure the last basic terrain type – mountains.

Generating Mountain Terrain

Generating mountain terrain will be very similar to the previous steps, but you’re encouraged to experiment with FastNoiseLite and find some fractals that you think would be good for generating mountains. There’s no right answer here – it’s all about creating the terrain you want to generate.

In this lesson, we will continue with our terrain generation in Godot. We will finalize the creation of the forest tiles and then introduce the concept of generating mountain tiles. We will also discuss how to use noise values to create a more realistic-looking terrain.

Finishing the Forest Generation

First, let’s complete the rendering of the forest section. The process is similar to how we created the desert tiles. We will use the desert as a base and modify its values to generate a forest. Here is the initial setup:

```
// Forest
if (forestMap[x, y] >= forestGenValues[0] &&
  forestMap[x, y] <= forestGenValues[1] &&
  h.terrainType == TerrainType.PLAINS)
{
  h.terrainType = TerrainType.FOREST;
}
```

We are checking to see if the noise value for the forest in a particular tile is within the range we specified for generating a forest. If the tile’s base type is planes, meaning it is on our continent, we will turn this tile into a forest.

Generating Mountain Tiles

Mountains are generated differently from deserts and forests. We want our mountains to resemble long, thin strips of tiles, rather than large patches like with deserts and forests. To achieve this, we will use a different type of noise called ‘ridged simplex noise’. This noise type creates long, snaking strips that can be transformed into something resembling mountain ranges.

The generation pipeline for mountains is similar to that of deserts and forests. We will copy one of these blocks and adjust the values accordingly. We will use mountain noise here and change the configurations:

```
// Mountain
FastNoiseLite mountainNoise = new FastNoiseLite();
mountainNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.Simplex;
mountainNoise.Seed = seed;
mountainNoise.Frequency = 0.02f;
mountainNoise.FractalType = FastNoiseLite.FractalTypeEnum.Ridged;
mountainNoise.FractalLacunarity = 2f;
float mountainNoiseMax = 0f;
```

Now, we need to generate the noise. This process is similar to what we did with the Forest and Desert. We’ll use Mountain Map with our Mountain Noise and our Mountain Noise Max. If we had even more types of terrain to generate here, it would be beneficial to turn this into a generic function to avoid repetition.

```
// Mountain
mountainMap[x, y] = Math.Abs(mountainNoise.GetNoise2D(x, y));
if (mountainMap[x, y] > mountainNoiseMax) mountainNoiseMax = mountainMap[x ,y];
```

Finally, we create some mountain generation values...

```
// Mountain gen values
Vector2 mountainGenValues = new Vector2(mountainNoiseMax/10 * 5.5f, mountainNoiseMax
+ 0.05f);
```

...and generate the mountains, just like we did with the forest and desert tiles:

```
// Mountain
if (mountainMap[x, y] >= mountainGenValues[0] &&
  mountainMap[x, y] <= mountainGenValues[1] &&
  h.terrainType == TerrainType.PLAINS)
{
  h.terrainType = TerrainType.MOUNTAIN;
}
```

With all these steps done, we should now see all of our desert, forest, and mountain generation in Godot. Each run will generate different terrain due to the random nature of the noise, but the mountains should appear in long, thin strips, unlike the patches for forests and deserts.

Conclusion

We have now successfully generated our terrain types for our base terrain map. In the next lesson, we will cover the final piece of terrain generation, which is creating ice terrain on the north and south poles of our map. Stay tuned!

In this lesson, we will continue with our terrain generation by adding ice terrain at the northern and southern extremes of our map. We will use a new generation strategy to create our ice caps, different from the fast noise light code we've been using till now.

Algorithm for Ice Cap Generation

Our basic algorithm for generating ice caps on our map is as follows:

1. We will traverse the map along the x-axis.
2. For each x position, we will place ice tiles a random number of tiles deep from the top and bottom extremities of the map.

This will result in the entire bottom and top of the map being covered in ice, with some random jagged edges of the ice caps reaching up from the bottom and top of the map.

Code Implementation

First, we need to decide the maximum layer of ice we want. After some experimentation, we've settled on a value of five for this maximum ice value. However, you're encouraged to experiment with this value to see how it affects your terrain.

Next, we'll create a for loop to iterate through all the x values for our map. We will iterate across the entire width and have one check for the north pole and one check for the south pole. For each x position, we want to generate ice tiles, a random number from one to the maximum ice value, and place the ice tiles accordingly.

Here's how the code looks:

```
int maxIce = 5;
for (int x = 0; x < width; x++)
{
    // North pole
    for (int y = 0; y < r.Next(maxIce) + 1; y++)
    {
        Hex h = mapData[new Vector2I(x, y)];
        h.terrainType = TerrainType.ICE;
        baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[h.terrainType]);
    }

    // South pole
    for (int y = height - 1; y > height - 1 - r.Next(maxIce) - 1; y--)
    {
        Hex h = mapData[new Vector2I(x, y)];
        h.terrainType = TerrainType.ICE;
        baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[h.terrainType]);
    }
}
```

We're grabbing the hex that we need to turn into ice and changing its terrain type to ice. Then we set the cell at this coordinate to that ice texture. This is done after all our previous generation is complete, so we are overwriting some terrain with ice to put the ice caps on top as a finishing touch.

After running this code, you should see ice sections generating at the top and bottom of your map. With this, we’ve finished our base terrain generation. In future lessons, we will add gameplay features such as cities and units and interactions between them. For now, we have a beautifully generated terrain on a hex grid, which can be applicable to a wide range of possible game genres and projects. Congratulations on making it this far!

Congratulations on reaching this point in our course on Godot C# and procedural terrain generation. We have covered a lot of ground, and this article serves as a summary of the key points we have learned.

Course Objectives

Our main objectives in this course were:

1. Understanding how to utilize Godot’s tile map layer system.
2. Creating a basic 2D camera suitable for strategy games.
3. Using Godot’s fast noise light system for procedural terrain generation.
4. Learning intermediate C# features.

Specific Learnings

More specifically, we delved into the following topics:

- The tile map layer nodes introduced in Godot 4.3.
- The use of tilesets and texture atlases with these nodes.
- The application of Camera2D for map panning and zooming.
- Polling mousewheel input for camera zooming.
- Setting up, configuring, and getting data from Godot’s Fast Noise Light random noise generator.
- Intermediate C# features such as language-integrated query.

About Zenva

Zenva is an online learning academy with over a million students. We offer a diverse range of courses for beginners and those looking to learn something new. Our courses are versatile, allowing you to learn in a way that suits you. You can choose to watch the video tutorials, read the lesson summaries, or follow along with the instructor using the included project files.

We hope you enjoyed this course and look forward to seeing you in the next one.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Camera.cs

Found in the Project root folder

This Godot script extends `Camera2D` to control the camera's movement and zoom on a 2D hex tile map. It ensures the camera stays within defined boundaries and responds to input actions for panning (left, right, up, and down) and zooming (in and out), utilizing mouse wheel events for additional zoom control.

```
using Godot;
using System;

public partial class Camera : Camera2D
{

  [Export]
  int velocity = 15;
  [Export]
  float zoom_speed = 0.05f;

  // Mouse states
  bool mouseWheelScrollingUp = false;
  bool mouseWheelScrollingDown = false;

  // Map boundaries
  float leftBound, rightBound, topBound, bottomBound;

  // Map reference
  HexTileMap map;

  // Called when the node enters the scene tree for the first time.
  public override void _Ready()
  {
    map = GetNode<HexTileMap>("../HexTileMap");

    leftBound = ToGlobal(map.MapToLocal(new Vector2I(0, 0))).X + 100;
    rightBound = ToGlobal(map.MapToLocal(new Vector2I(map.width, 0))).X - 100;
    topBound = ToGlobal(map.MapToLocal(new Vector2I(0, 0))).Y + 50;
    bottomBound = ToGlobal(map.MapToLocal(new Vector2I(0, map.height))).Y - 50;

  }

  // Called every frame. 'delta' is the elapsed time since the previous frame.
  public override void _PhysicsProcess(double delta)
  {
    // Map controls
    if (Input.IsActionPressed("map_right") && this.Position.X < rightBound)
    {
      this.Position += new Vector2(velocity, 0);
    }

    if (Input.IsActionPressed("map_left") && this.Position.X > leftBound)
    {
      this.Position += new Vector2(-velocity, 0);
    }

    if (Input.IsActionPressed("map_up") && this.Position.Y > topBound)
    {
      this.Position += new Vector2(0, -velocity);
    }

    if (Input.IsActionPressed("map_down") && this.Position.Y < bottomBound)
    {
      this.Position += new Vector2(0, velocity);
    }

    // Zoom controls
    if (Input.IsActionPressed("map_zoom_in") || mouseWheelScrollingUp)
    {
      if (this.Zoom < new Vector2(3f, 3f))
        this.Zoom += new Vector2(zoom_speed, zoom_speed);
    }

    if (Input.IsActionPressed("map_zoom_out") || mouseWheelScrollingDown)
    {
      if (this.Zoom > new Vector2(0.1f, 0.1f))
        this.Zoom -= new Vector2(zoom_speed, zoom_speed);
    }

    if (Input.IsActionJustReleased("mouse_zoom_in"))
      mouseWheelScrollingUp = true;

    if (!Input.IsActionJustReleased("mouse_zoom_in"))
      mouseWheelScrollingUp = false;

    if (Input.IsActionJustReleased("mouse_zoom_out"))
      mouseWheelScrollingDown = true;

    if (!Input.IsActionJustReleased("mouse_zoom_out"))
      mouseWheelScrollingDown = false;

  }
}
```

Game.cs

Found in the Project root folder

This code outlines the basic structure of a `Game` class in Godot, inheriting from `Node`, and includes a FastNoiseLite instance for procedural noise generation, which is exposed to the editor.

The `_Ready` method prepares the node when it enters the scene, and the `_Process` method updates every frame.

```
using Godot;
using System;

public partial class Game : Node
{

  [Export]
  FastNoiseLite noise;

  // Called when the node enters the scene tree for the first time.
  public override void _Ready()
  {
  }

  // Called every frame. 'delta' is the elapsed time since the previous frame.
  public override void _Process(double delta)
  {
  }
}
```

HexTileMap.cs

Found in the Project root folder

This code defines a hexagonal tile map in Godot, with various terrain types generated using noise functions for a procedural terrain. It initializes and sets tile data for different terrain types such as plains, water, forests, and mountains, including the borders and overlays for each tile.

```
using Godot;
using System;
using System.Collections.Generic;
using System.Linq;

public enum TerrainType { PLAINS, WATER, DESERT, MOUNTAIN, ICE, SHALLOW_WATER, FOREST
, BEACH }

public class Hex
{
  public readonly Vector2I coordinates;

  public TerrainType terrainType;

  public Hex(Vector2I coords)
  {
    this.coordinates = coords;
  }

}
```

```
public partial class HexTileMap : Node2D
{

  [Export]
  public int width = 100;
  [Export]
  public int height = 60;

  // Map data
  TileMapLayer baseLayer, borderLayer, overlayLayer;

  Dictionary<Vector2I, Hex> mapData;
  Dictionary<TerrainType, Vector2I> terrainTextures;

  // Called when the node enters the scene tree for the first time.
  public override void _Ready()
  {
    baseLayer = GetNode<TileMapLayer>("BaseLayer");
    borderLayer = GetNode<TileMapLayer>("HexBordersLayer");
    overlayLayer = GetNode<TileMapLayer>("SelectionOverlayLayer");

    // Initialize map data
    mapData = new Dictionary<Vector2I, Hex>();
    terrainTextures = new Dictionary<TerrainType, Vector2I>
    {
      { TerrainType.PLAINS, new Vector2I(0, 0) },
      { TerrainType.WATER, new Vector2I(1, 0) },
      { TerrainType.DESERT, new Vector2I(0, 1)},
      { TerrainType.MOUNTAIN, new Vector2I(1, 1)},
      { TerrainType.SHALLOW_WATER, new Vector2I(1, 2)},
      { TerrainType.BEACH, new Vector2I(0, 2)},
      { TerrainType.FOREST, new Vector2I(1, 3)},
      { TerrainType.ICE, new Vector2I(0, 3)},
    };

    GenerateTerrain();

  }

  public void GenerateTerrain()
  {
    float[,] noiseMap = new float[width, height];
    float[,] forestMap = new float[width, height];
    float[,] desertMap = new float[width, height];
    float[,] mountainMap = new float[width, height];

    Random r = new Random();
    int seed = r.Next(100000);

    // BASE TERRAIN (Water, Beach, Plains)
    FastNoiseLite noise = new FastNoiseLite();

    noise.Seed = seed;
    noise.Frequency = 0.008f;
    noise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
    noise.FractalOctaves = 4;
    noise.FractalLacunarity = 2.25f;

    float noiseMax = 0f;


    // Forest
    FastNoiseLite forestNoise = new FastNoiseLite();

    forestNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.Cellular;
    forestNoise.Seed = seed;
    forestNoise.Frequency = 0.04f;
    forestNoise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
    forestNoise.FractalLacunarity = 2f;

    float forestNoiseMax = 0f;


    // Desert
    FastNoiseLite desertNoise = new FastNoiseLite();

    desertNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.SimplexSmooth;
    desertNoise.Seed = seed;
    desertNoise.Frequency = 0.015f;
    desertNoise.FractalType = FastNoiseLite.FractalTypeEnum.Fbm;
    desertNoise.FractalLacunarity = 2f;

    float desertNoiseMax = 0f;

    // Mountain
    FastNoiseLite mountainNoise = new FastNoiseLite();

    mountainNoise.NoiseType = FastNoiseLite.NoiseTypeEnum.Simplex;
    mountainNoise.Seed = seed;
    mountainNoise.Frequency = 0.02f;
    mountainNoise.FractalType = FastNoiseLite.FractalTypeEnum.Ridged;
    mountainNoise.FractalLacunarity = 2f;

    float mountainNoiseMax = 0f;



    // Generating noise values
    for (int x = 0; x < width; x++)
    {
      for (int y = 0; y < height; y++)
      {
        // Base terrain
        noiseMap[x, y] = Math.Abs(noise.GetNoise2D(x, y));
        if (noiseMap[x, y] > noiseMax) noiseMax = noiseMap[x ,y];

        // Desert
        desertMap[x, y] = Math.Abs(desertNoise.GetNoise2D(x, y));
        if (desertMap[x, y] > desertNoiseMax) desertNoiseMax = desertMap[x ,y];
```

```
        // Forest
        forestMap[x, y] = Math.Abs(forestNoise.GetNoise2D(x, y));
        if (forestMap[x, y] > forestNoiseMax) forestNoiseMax = forestMap[x ,y];

        // Mountain
        mountainMap[x, y] = Math.Abs(mountainNoise.GetNoise2D(x, y));
        if (mountainMap[x, y] > mountainNoiseMax) mountainNoiseMax = mountainMap[x ,y
];

      }
    }


    List<(float Min, float Max, TerrainType Type)> terrainGenValues = new List<(float
 Min, float Max, TerrainType Type)>
    {
      (0, noiseMax/10 * 2.5f, TerrainType.WATER),
      (noiseMax/10 * 2.5f, noiseMax/10 * 4, TerrainType.SHALLOW_WATER),
      (noiseMax/10 * 4, noiseMax/10 * 4.5f, TerrainType.BEACH),
      (noiseMax/10 * 4.5f, noiseMax + 0.05f, TerrainType.PLAINS)
    };

    // Forest gen values
    Vector2 forestGenValues = new Vector2(forestNoiseMax/10 * 7, forestNoiseMax + 0.0
5f);
    // Desert gen values
    Vector2 desertGenValues = new Vector2(desertNoiseMax/10 * 6, desertNoiseMax + 0.0
5f);
    // Mountain gen values
    Vector2 mountainGenValues = new Vector2(mountainNoiseMax/10 * 5.5f, mountainNoise
Max + 0.05f);


    for (int x = 0; x < width; x++)
    {
      for (int y = 0; y < height; y++)
      {
        Hex h = new Hex(new Vector2I(x, y));
        float noiseValue = noiseMap[x, y];

        h.terrainType = terrainGenValues.First(range => noiseValue >= range.Min
                                    && noiseValue < range.Max).Type;
        mapData[new Vector2I(x, y)] = h;

        // Desert
        if (desertMap[x, y] >= desertGenValues[0] &&
          desertMap[x, y] <= desertGenValues[1] &&
          h.terrainType == TerrainType.PLAINS)
        {
          h.terrainType = TerrainType.DESERT;
        }

        // Forest
        if (forestMap[x, y] >= forestGenValues[0] &&
          forestMap[x, y] <= forestGenValues[1] &&
          h.terrainType == TerrainType.PLAINS)
        {
          h.terrainType = TerrainType.FOREST;
        }

        // Mountain
        if (mountainMap[x, y] >= mountainGenValues[0] &&
          mountainMap[x, y] <= mountainGenValues[1] &&
          h.terrainType == TerrainType.PLAINS)
        {
          h.terrainType = TerrainType.MOUNTAIN;
        }


        baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[h.terrainType]);

        // Set tile borders
        borderLayer.SetCell(new Vector2I(x, y), 0, new Vector2I(0, 0));
      }
    }

    // Ice cap gen
    int maxIce = 5;
    for (int x = 0; x < width; x++)
    {
      // North pole
      for (int y = 0; y < r.Next(maxIce) + 1; y++)
      {
        Hex h = mapData[new Vector2I(x, y)];
        h.terrainType = TerrainType.ICE;
        baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[h.terrainType]);
      }

      // South pole
      for (int y = height - 1; y > height - 1 - r.Next(maxIce) - 1; y--)
      {
        Hex h = mapData[new Vector2I(x, y)];
        h.terrainType = TerrainType.ICE;
        baseLayer.SetCell(new Vector2I(x, y), 0, terrainTextures[h.terrainType]);
      }


    }



  }

  public Vector2 MapToLocal(Vector2I coords)
  {
    return baseLayer.MapToLocal(coords);
  }

}
```