Welcome to this course where we'll be building a 4x strategy game in Godot using C#. Your instructor for this course is Cameron. The primary goal of this series is to create a turn-based strategy game based on a map of hexagonal tiles, a common format for strategy games, especially in the 4x genre. The skills and Godot features you'll learn in this project are applicable to a wide range of other projects, such as procedural generation and turn mechanics.

Course Overview

This core series will serve as a foundation for creating a complete 4x strategy game. You can use this base to develop a game that suits your vision. In this particular segment, we'll be covering UI and city generation. Alongside this, we'll make our map interactive, create a UI manager to handle the complex user interface, and define, create, and generate cities and factions.

Prerequisites

- Basic skills with Godot (GD script knowledge is sufficient)
- Basic knowledge of C#
- Completion of the previous course in this series on creating the hexagonal map

Topics Covered

1. System for selecting tiles on the map, including converting between local and global space and different coordinate systems
2. UI manager system to handle events and signals from the map and other gameplay entities
3. Differences between Godot engine built-in signals and raw C# events/signals (delegates)
4. Complex system for managing cities and factions, including spawning cities, placing them on the map, and generating random map colors for different factions

About Zenva

Zenva is an education platform with over a million students. We offer a wide range of courses for beginners and those looking to learn something new. Our courses are versatile, allowing you to learn through video tutorials, lesson summaries, or by following along with the included project files.

Getting Started

Without further ado, let's dive right into building this 4x strategy game in Godot.

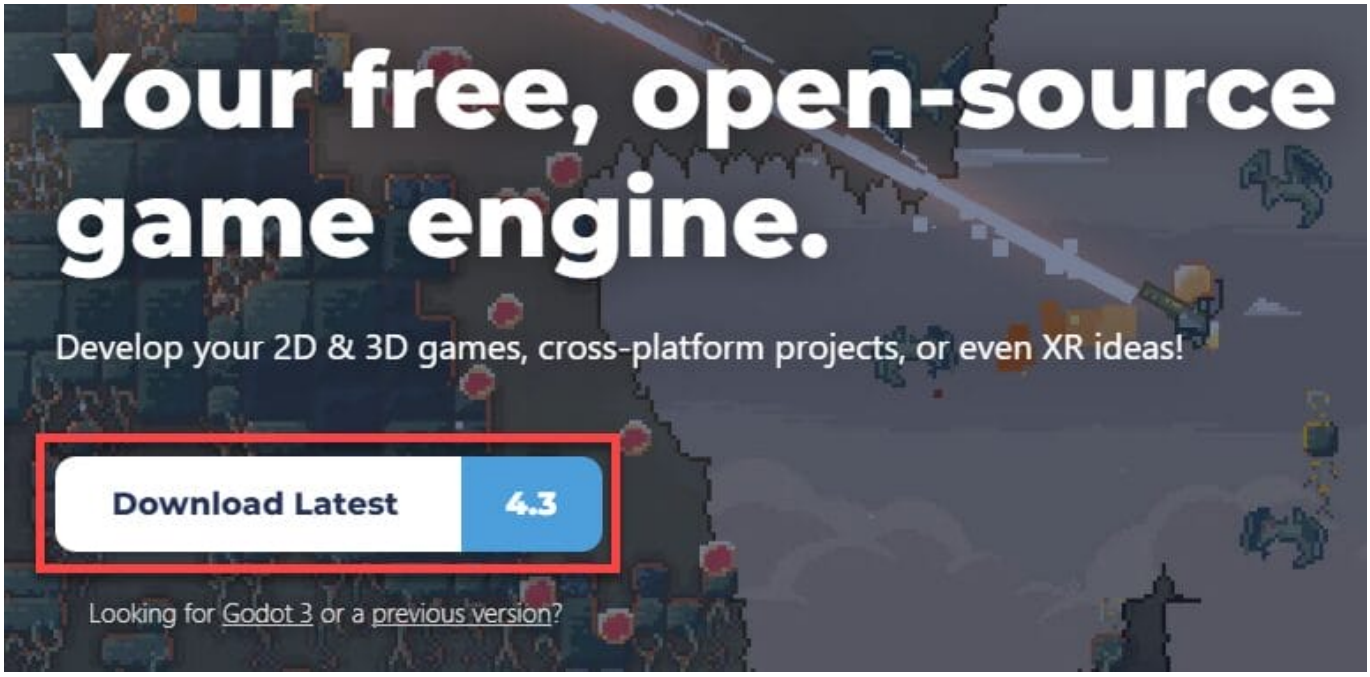
Which version of Godot should you use for this course?

Technology changes rapidly, so to make sure you have an optimal learning experience we recommend using **Godot 4.3** for this course – the latest stable release.

Please make sure not to use newer versions of the software than what we recommend, as the course material provided might not work as expected.

How to Install Version 4.3 .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.



Once on the download page, just click the option for **Godot 4.3 .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!

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:

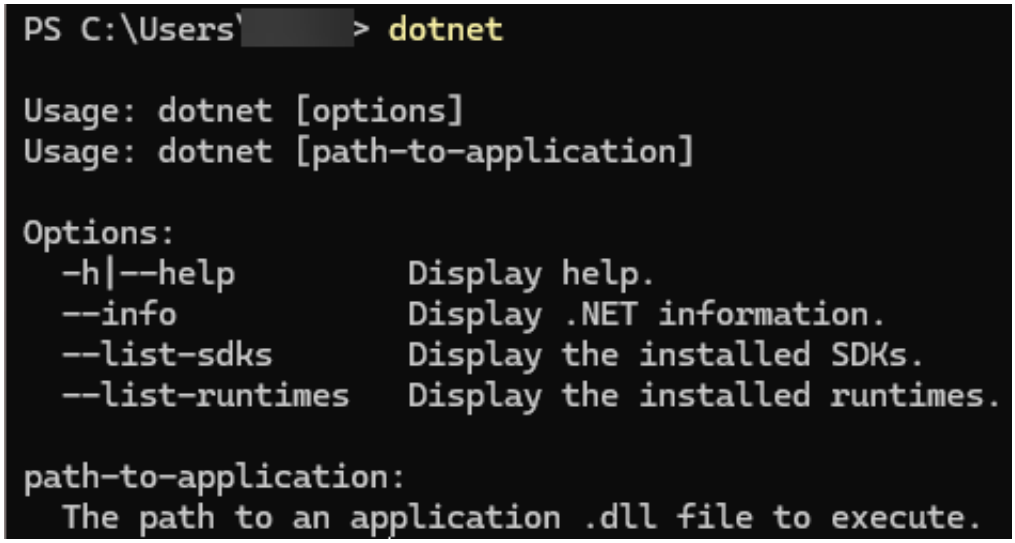
Additional .NET Setup

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, simply run the installer as you normally would for your device. 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.



Welcome back to our course on creating a hex-based strategy game in Godot Engine using C#. In this lesson, we will start adding interactivity to our map. Currently, our map only supports zooming and panning controls, but we will now implement a system to make the map clickable. This will allow us to select tiles and perform actions based on those selections.

Recap of Previous Progress

Before we dive into adding interactivity, let's recap what we have accomplished so far:

- We have created a terrain generation system that generates visually appealing ocean and continent terrain based on hexes.
- We have implemented basic zooming and panning controls for the map.

Goals for This Lesson

In this lesson, our primary goal is to make the map interactive. Specifically, we want to be able to click on a tile and print out information about the selected tile. This will serve as the foundation for more complex interactions in future lessons.

Steps to Implement Interactivity

To achieve our goal, we will follow these steps:

1. Add a helper function to the `Hex` class to override the `ToString` method, allowing us to print out the tile's coordinates and terrain type.
2. Implement the `unhandledInput` method in the `HexTileMap` class to detect mouse clicks and print out the selected tile's information.

Step 1: Override the `ToString` Method in the `Hex` Class

First, we need to add a helper function to the `Hex` class that will allow us to print out the tile's coordinates and terrain type. This is done by overriding the `ToString` method.

```
public override string ToString()
{
    return $"Coordinates: ({this.coordinates.X}, {this.coordinates.Y}). Terrain type:
 {this.terrainType}";
}
```

Step 2: Implement the `unhandledInput` Method in the `HexTileMap` Class

Next, we will implement the `unhandledInput` method in the `HexTileMap` class to detect mouse clicks and print out the selected tile's information. This method will only detect unhandled input, which means it will not interfere with other UI elements that might consume the input. For now, we just want to set up the framework to check if the event is a mouse button event.

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse)
    {
        // Rest of function
    }
}
```


Conclusion

In this lesson, we have implemented part of a basic system to make our map interactive. In the next lesson, we'll continue coding our unhandledInput method so we can print the coordinate information to the console.

Welcome back! In this lesson, we will continue setting up our basic map interaction using Godot's unhandled input function to handle mouse clicks. We have already overridden Godot's built-in unhandled input function and written an if statement to check whether the input event is a mouse click. Now, we will translate the mouse coordinates into map coordinates.

Translating Mouse Coordinates to Map Coordinates

Converting between screen space and world space can be daunting in game development. However, Godot makes this process straightforward. We need to convert the mouse coordinates, which are given in pixels relative to the 2D origin of the whole 2D world in Godot, into the actual hexes on our tile map.

Steps to Convert Mouse Coordinates

1. Pull the mouse coordinates using get_global_mouse_position().
2. Convert the global mouse position to local coordinates using to_local().
3. Convert the local coordinates to map coordinates using local_to_map().

Implementing the Conversion

Just to prove how uncomplicated the above is, we can combine everything into a single line:

```
Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));
```

Now, we can use these map coordinates to interact with our tile map. For example, we can print out the tile data at the clicked coordinates to verify that our conversion is working correctly.

```
GD.Print(mapData[mapCoords]);
```

Testing the Implementation

To test our implementation, we can run the game and click on different tiles. The console output should display the tile data at the clicked coordinates. This is a significant step forward in adding interactivity to our map because now we can know what tile our user is clicking on.

Challenge

This works pretty well, but there are still some refinements we need to make to ensure this is working correctly. One of the most important ones is limiting where we can actually click on the map because if we were to click outside of the boundaries of the map, we would cause errors and possibly crashes. As a challenge to you, try and figure out what checks we're going to need to make to wrap around this print statement we have so far to make sure that it's only going to fire if a click is made within the boundaries of the map. Try that out as a challenge on your own and we'll answer it in the subsequent video.

That's all for this lesson! In the next video, we will refine our implementation to handle edge cases and ensure our map interaction is robust.

In this lesson, we will finish our basic map interaction system by dealing with a few edge cases. As we tested in the last lesson, our basic system is working, but there are a few issues. The first issue is that if we click outside of the map in any situation, it's going to throw an error. This is because there are no hex tile map coordinates outside of the actual map. We will address this challenge by limiting our mouse click detection to be within the bounds of the map.

Limiting Mouse Click Detection

To limit our mouse click detection, we need to ensure that the click occurred within the boundaries of the map. We will do this by checking the map coordinates before doing anything with them.

```
if (mapCoords.X >= 0 && mapCoords.X < width && mapCoords.Y >= 0 && mapCoords.Y < heig
ht)
{
    // Proceed with the click action
}
```

This check ensures that the click is within the map's boundaries. If the click is outside the map, we will not proceed with any action.

Limiting to Left Mouse Button

Another improvement we can make is to limit our mouse input event system to only respond to the left mouse button. This can be done by querying the button mask of the input event (within the if statement we just made).

```
if (mouse.ButtonMask == MouseButtonMask.Left)
{
    // Proceed with the click action
}
```

By adding this check, we ensure that only the left mouse button triggers any action in this section.

Implementing Graphics for Clicking on the Map

Now that we have handled the edge cases, we can start to implement some actual graphics for clicking on the map. In the last course, we added a selection overlay layer that we will use to represent whether a tile has been clicked on. We will set a tile atlas for this layer and use it to highlight the clicked tile.

Then, in our code, we can set the cell in the overlay layer to highlight the clicked tile.

```
// Set the cell in the overlay layer
overlayLayer.SetCell(mapCoordinates, 0, new Vector2I(0, 1));
```

Testing the Implementation

Let's head back to the editor to make sure that this worked. Now, when we click on a tile, it gets highlighted. However, there are still some issues we need to address:

- We need to ensure that only one tile can be selected at a time.
- If we click outside of the map, all tiles should be deselected.

We will take care of these cases in the next video.

In this lesson, we will finish up our map tile selection system. Last time, we successfully implemented a selection graphic that appears on tiles when they are selected. However, we encountered a couple of issues: multiple tiles could be selected at once, and clicking outside the map did not deselect all tiles. To address these issues, we will keep track of the currently selected tile and ensure that only one tile can be selected at a time. Additionally, we will handle clicks outside the map to deselect any selected tiles.

Tracking the Currently Selected Tile

To keep track of the currently selected tile, we will introduce a class member variable called currentSelectedCell. This variable will store the coordinates of the currently selected tile. We will initialize it to a default value of (-1, -1), representing a non-selected cell that is not on our map.

```
Vector2I currentSelectedCell = new Vector2I(-1, -1);
```

Handling Tile Selection

We will modify the _UnhandledInput function to ensure that only one tile can be selected at a time. If the mouse click is on a different cell than the currently selected one, we will deselect the currently selected cell before selecting the new cell. Here is the updated code:

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse)
    {
        Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));

        if (mapCoords.X >= 0 && mapCoords.X < width && mapCoords.Y >= 0 && mapCoords.
Y < height)
        {
            if (mouse.ButtonMask == MouseButtonMask.Left)
            {
                GD.Print(mapData[mapCoords]);

                if (mapCoords != currentSelectedCell)
                {
                    overlayLayer.SetCell(currentSelectedCell, -1);
                }
                overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
                currentSelectedCell = mapCoords;
            }
        }
        else
        {
            // More code
        }
    }
}
```

Deselecting Tiles When Clicking Outside the Map

To handle clicks outside the map, we will add an else branch to the _UnhandledInput function. If the

click is outside the map, we will deselect the currently selected tile by setting its cell to -1.

```
else
{
    overlayLayer.SetCell(currentSelectedCell, -1);
}
```

Reducing Aliasing on Tile Graphics

Finally, we will address the issue of aliasing on tile graphics when zooming out. Aliasing can make the graphics difficult to look at. To alleviate this, we will lower the opacity of our tile borders to make them less aggressive. We will set the alpha value of the visibility of our canvas item to a lower value, such as 40.

Conclusion

With these changes, we have finished our basic map interaction and selection system. We can now select only one tile at a time and deselect tiles by clicking outside the map. Additionally, we have reduced the aliasing on tile graphics to improve their appearance when zooming out. In the next lesson, we will start creating a user interface (UI) and populating our tiles with resources, introducing our first real gameplay element.

Welcome back! In our previous lesson, we successfully implemented basic map interaction, allowing us to select specific tiles and get a readout in the console of the selected tile. Now, we are ready to start adding some first real gameplay elements to our tiles. Currently, each tile has a terrain and coordinate, but in anticipation of adding cities and city gameplay to our strategy game later in this course, we are going to populate our world with resources.

Adding Resource Attributes to Tiles

Each tile will have two basic resource values: a food value and a production value. Later in this course, we will tie these values to some gameplay elements, such as food tying into the rate of growth for a city, and so on. For now, we just want to make sure that our tiles have these attributes and that we can populate them in an interesting way using Perlin noise during our map generation process.

Modifying the Hex Class

Before we can generate resources for our tiles, we need to ensure that our Hex class, which represents particular tiles on the map, can accommodate these resources. For this simple skeletal strategy game, we are not going to have anything too complicated in the way of resources, and we will represent our food and production values with single integers.

Let's add two public integers to the Hex class, one for food and one for production:

```
public int food;
public int production;
```

This is all that we need to add to our Hex class to have these values available. Additionally, we might want to update the ToString method to include these new attributes:

```
public override string ToString()
{
    return $"Coordinates: ({this.coordinates.X}, {this.coordinates.Y}). Terrain type:
 {this.terrainType}). Food: {this.food}. Production: {this.production}";
}
```

Generating Resources

Each of our tiles now has a food and production value, but they are all set to a default of zero. We need to generate food and production values for these tiles during our map generation phase. We will base these values on the terrain type.

Let's create a new function called generateResources in our HexTileMap class. To achieve our desired results, a nested loop will iterate over each tile on the map, accessing the corresponding Hex object stored in the mapData collection (similar to how our map was generated).

For each Hex object, a switch statement will determine the terrain type and assign random values for food and production accordingly. For instance, tiles with a terrain type of PLAINS will receive between 2 and 6 units of food and 0 to 3 units of production. Similarly, FOREST tiles will receive between 1 and 4 units of food and 2 to 6 units of production. This random assignment will add a layer of realism and variability to the game world, providing a unique experience for players.

Of course, you can play around with the values as you see fit for your game and terrain!

```
public void GenerateResources()
{
    Random r = new Random();

    // populate tiles with food and production
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            Hex h = mapData[new Vector2I(x, y)];

            switch (h.terrainType)
            {
                case TerrainType.PLAINS:
                    h.food = r.Next(2, 6);
                    h.production = r.Next(0, 3);
                    break;
                case TerrainType.FOREST:
                    h.food = r.Next(1, 4);
                    h.production = r.Next(2, 6);
                    break;
                case TerrainType.DESERT:
                    h.food = r.Next(0, 2);
                    h.production = r.Next(0, 2);
                    break;
                case TerrainType.BEACH:
                    h.food = r.Next(0, 4);
                    h.production = r.Next(0, 2);
                    break;
            }
        }
    }
}
```

Finally, we need to call our generateResources function right after our terrain generation step in our _Ready function:

```
public override void _Ready()
{
    // ... existing code ...

    GenerateTerrain();
    GenerateResources();

    // ... existing code ...
}
```

Testing the Resource Generation

Now, let’s head over to the editor and see how this turned out. You should see that our forest now has some production, our plains have some food and production, and so on. We have randomly populated our map with resources.

Congratulations! You have successfully added resource attributes to your tiles and generated them based on the terrain type. This sets the foundation for more complex gameplay elements that we will introduce later in the course.

Welcome back to our series on creating a top-down two-dimensional strategy game using Godot. In the previous videos, we completed our system for interacting with the map. Now, we will start building one of the most crucial systems for our game: the UI system. While it may not be the most exciting part, the UI is vital for a strategy game as it serves as the primary means for players to interact with the game world. In the following videos, we will lay the foundation for our UI system by creating a UI manager and our first UI element for displaying information about terrain tiles.

Creating a Basic UI Scene for a Terrain Tile

Before we can display any information, we need to create a basic UI scene for a terrain tile. Let's go through the steps to create this:

1. Navigate to the **Resources** panel and create a new scene.

2. Since this is a UI scene, we will use a **Panel** as the root node type. Name this scene **TerrainTileUI**. Note that Godot uses a specific naming convention where the final 'i' in 'UI' is lowercase.

Setting Up the Terrain Tile UI

Now that we have our basic UI scene, let's set up the elements to display information about a terrain tile:

1. Expand the **TerrainTileUI** panel and change its size to 250×260 pixels.

2. Add a **TextureRect** node under the panel to serve as a placeholder for the terrain image. Set its width to 250 pixels and height to 160 pixels.

3. Add three **Label** nodes under the **TextureRect** to display the following information:

    - **Terrain**: The type of terrain.
    - **Food**: The food value of the terrain.
    - **Production**: The production value of the terrain.

4. Name the labels **TerrainLabel**, **FoodLabel**, and **ProductionLabel** respectively.

Importing Default Images

To get an idea of what the final UI will look like, we can import default images for the terrain:

1. Import the terrain images from the course materials into the **Textures** folder.

2. Set a default image for the **TextureRect** node. For now, we can use the 'plains' image and set it to fit the width of the node.

Remember, the image we set here is just a default and will be changed dynamically in the game to correspond to different terrain types.

Next Steps

With our basic UI scene set up, we are ready to move on to the next step. In the following video, we

will start creating our UI manager to begin displaying these UI elements on the screen. Stay tuned!

In the previous lesson, we created a basic terrain tile UI component. In this lesson, we will start integrating that UI component into a broader UI manager class. This class will handle not only the terrain UI but also UI for other gameplay elements like cities and units. Let's go through the steps to achieve this.

Adding a Script to the Terrain Tile UI

Before we move on to creating our UI manager, we need to add a script to our terrain tile UI. This script will be empty for now, but it is necessary for our UI manager to access specific functions and properties of the terrain tile.

```
using Godot;

public partial class TerrainTileUI : Panel
{
    // Empty class for now
}
```

The reason we need to do this is because in our UI manager class, we will want to be able to access specific functions and properties of the terrain tile itself. By creating this script, we ensure that our terrain tile is treated as a type in C#, which is necessary for our UI manager.

Creating the UI Manager

Now that we have added the script to our terrain tile UI, we are ready to create our UI manager. The first step is to add another canvas layer to our game. This canvas layer will be a separate 2D drawing surface for Godot to render 2D objects onto, placed above our map canvas.

1. Create a new child node on our game and name it CanvasLayer.
2. Underneath this canvas layer, create a new scene for our UI manager.
3. Name the root of this scene UIManager and set it as a generic 2D node.
4. Add this scene as a child of the canvas layer.

The important part here is the script for our UI manager. Let's create a new script for the UI manager:

```
using Godot;
using System;

public partial class UIManager : Node2D
{
    // Script content will be added here
}
```

Basic Theory of the UI Manager

The UI manager class is designed to handle various types of UI elements in our game. It will manage which UI panel should be on the screen at a given time and hook up necessary signals for interaction. Here are some key points about the UI manager:

- The UI manager will have packed scenes to represent different types of UI elements.
- It will manage which UI panel should be visible at any given time.
- It will handle signals to interact with the UI elements based on actions taking place on the map.

Loading Packed Scenes

The first important thing our UI manager class will have is packed scenes to represent all the different types of UI elements we want to be able to create. For now, we only have a terrain scene, so we will create a packed scene for that.

In the _Ready function, we will load the packed scene for the terrain tile UI:

```
PackedScene terrainUiScene;

public override void _Ready()
{
    terrainUiScene = ResourceLoader.Load("TerrainTileUI.tscn");
}
```

This will load the terrain tile UI scene so we are ready to use it to instantiate new terrain UI panels.

Conclusion

With these steps, we have set up the skeleton for our UI manager. In the next lesson, we will look at how to actually instantiate a terrain tile UI and hook it up via signals to actions taking place on our map. Stay tuned for that!

Welcome back! In this lesson, we will continue setting up our system for managing the UI in our strategy game. Last time, we started creating a skeleton of a UI manager class. Now, we will start linking all the components together, from the map to the terrain tile UI to the UI manager itself.

Linking Components

We will begin with the smallest component: the TerrainTileUI class. Last time, we created this class but didn't add any functionality. In this video, we will start filling it out with the necessary components for taking in data and updating the terrain tile UI with the appropriate hex data.

Getting References to UI Components

The first thing we need to do is get references to all the UI components inside our panel. This will allow us to update them programmatically when we receive new data. We have four basic components:

- A TextureRect that will have an image. We'll call this terrainImage.
- Three different labels: terrainLabel, foodLabel, and productionLabel.

We will create these as member variables and then get references to them inside the _Ready function:

```
TextureRect terrainImage;
Label terrainLabel, foodLabel, productionLabel;

public override void _Ready()
{
    terrainLabel = GetNode<Label>("TerrainLabel");
    foodLabel = GetNode<Label>("FoodLabel");
    productionLabel = GetNode<Label>("ProductionLabel");
    terrainImage = GetNode<TextureRect>("TerrainImage");
}
```

Referencing the Hex Object

Next, we need a reference to the actual hex object that will be the source of all our data. We will create a new hex object and explicitly set it to null to begin with, as we will populate it with hex data via a signal later on.

```
Hex h = null;
```

Creating the SetHex Function

To facilitate the process of transferring data from the map into the UI, we will create a function called SetHex that will take in a hex object from our map and populate our terrain tile with all of its attributes. For now, we will disregard the image and terrain, as we'll be dealing with that in a future lesson.

```
public void SetHex(Hex h)
{
    this.h = h;
    foodLabel.Text = $"Food: {h.food}";
```



```
    productionLabel.Text = $"Production: {h.production}";
}
```

Updating the UIManager Class

Going one step further up the chain, let's head over to our UIManager class and deal with actually setting this terrain tile data. The UIManager class will be the centralized hub where it will receive all the necessary data from the map and gameplay and distribute it accordingly to the necessary UI elements.

To that end, our UIManager class is going to need to hold references to all the potential UI elements that it can contain. For now, we will store a reference to the TerrainTileUI.

```
TerrainTileUI terrainUi = null;
```

Creating the SetTerrainUI Function

We will create a function for actually setting the terrain UI. This function will take in directly from the hex tile map the necessary hex data and propagate it to a brand new TerrainTileUI instance.

```
public void SetTerrainUI(Hex h)
{
    if (terrainUi is not null) terrainUi.QueueFree();

    terrainUi = terrainUiScene.Instantiate() as TerrainTileUI;
    AddChild(terrainUi);

    terrainUi.SetHex(h);
}
```

This should be all the logic we need to be able to receive a signal from HexTileMap, which we will deal with in the next video, finishing this chain of data propagation from our map to our UI.

Hello and welcome back. In this video, we'll be finishing up our data pipeline from our map to our UI manager to our terrain tile UI. This has been a lot of setup on the backend here to get this UI manager working. But once we have this system in place, it'll be a lot easier in the future to add other types of UI to it, such as UI for cities and units. So the last thing we have to do is to actually send signals from our HexTileMap class to our UI Manager, which will then propagate that data down into our TerrainTile UI.

Storing a Reference to the UI Manager

The first thing we'll need to do in our HexTileMap class is to store a reference to our UI Manager. We will create a new reference here, and we will assign this inside of our ready function.

```
public partial class HexTileMap : Node2D
{
    // UI
    UIManager uimanager;

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

        uimanager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");
    }
}
```

Setting Up UI Signals

Now that we have a reference to the UI manager, we can set up our UI signals. In Godot, you're probably familiar with Godot's built-in signal system, but in this particular instance, we're actually going to leverage the built-in C# signal system instead, or rather, what C# calls events and delegates. In Godot's C# implementation, this is actually what the Godot signal system is built on top of, but we're going to need to leverage the raw C# system here for the reason that our hex class is not actually a Godot type.

```
public partial class HexTileMap : Node2D
{
    // Signals
    public delegate void SendHexDataEventHandler(Hex h);
    public event SendHexDataEventHandler SendHexData;
}
```

Invoking the Event

Now that this is declared, all that's left to do is attach the signal and invoke it. We're going to attach the signal to our UI manager in our ready function here. We'll use C#'s very handy syntax for attaching the signal.

```
public override void _Ready()
{
    // Other initializations...
```



```
    uimanager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");

    // Attach the signal to the UI manager's SetTerrainUI function
    this.SendHexData += uimanager.SetTerrainUI;
}
```

Now we can head down to our unhandled input here and actually invoke this event when we click on a new terrain tile.

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse)
    {
        Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));

        if (mapCoords.X >= 0 && mapCoords.X < width && mapCoords.Y >= 0 && mapCoords.
Y < height)
        {
                Hex h = mapData[mapCoords];
                GD.Print(h)

                SendHexData?.Invoke(h);

                if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSel
ectedCell, -1);
                overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
                currentSelectedCell = mapCoords;
            }
        }
        else
        {
            overlayLayer.SetCell(currentSelectedCell, -1);
        }
    }
}
```

Ensuring Correct Order of Operations

One thing that we forgot to do here was we actually need to reverse the order of setting the hex and adding the terrain UI node as a child to the scene. The reason is that when we set the hex on a terrain UI, it's trying to reference the labels that won't actually exist until the terrain UI is added to the scene. Only then will Godot populate the children of that node with what is stored in the scene.

```
public void SetTerrainUI(Hex h)
{
  if (terrainUi is not null) terrainUi.QueueFree();

  terrainUi = terrainUiScene.Instantiate() as TerrainTileUI;
  AddChild(terrainUi);
```



```
  terrainUi.SetHex(h);

}
```

With that we can take a look at our new UI system. We can see that our UI is appearing, and if we click on tiles that actually have food and production, you can see that it is updating accordingly. So in the next video, we'll be finishing this out by doing the necessary behind-the-scenes work to have a string for the terrain type displayed as well as loading in all of our terrain textures.

Hello! We are almost done with creating our terrain tile UI system. We've successfully gotten the UI to appear on the screen, as well as update food and production data. But what about the terrain types and the textures? In this lesson, we'll focus on creating dictionaries inside of our TerrainTileUI class to serve as mappings for which terrain type should be associated with which string to describe the terrain, and especially which terrain texture to display on the image.

Creating Static Members

To start, we'll create some static members inside our TerrainTileUI class. Static members are shared by all instances of the class and are not tied to a particular instantiation. This is perfect for things like enumerations and textures that every terrain tile UI instance will need access to.

Mapping Terrain Types to Strings

First, let's create a dictionary to map terrain types to strings. This will allow us to define a string that can be used to represent each terrain type in plain English.

```
public static Dictionary<TerrainType, string> terrainTypeStrings = new Dictionary<Ter
rainType, string>
{
    { TerrainType.PLAINS, "Plains" },
    { TerrainType.BEACH, "Beach" },
    { TerrainType.DESERT, "Desert" },
    { TerrainType.MOUNTAIN, "Mountain" },
    { TerrainType.ICE, "Ice" },
    { TerrainType.WATER, "Water" },
    { TerrainType.SHALLOW_WATER, "Shallow Water" },
    { TerrainType.FOREST, "Forest" },
};
```

Now, we can apply these mappings to our UI whenever we get new data:

```
terrainLabel.Text = "Terrain: " + terrainTypeStrings[hex.terrainType];
```

Loading Textures

Next, we need to load in textures for each terrain type. We'll create another dictionary to map terrain types to Texture2D objects. We'll also create a static function called LoadTerrainImages to load these textures one time at the beginning of the game.

```
public static Dictionary<TerrainType, Texture2D> terrainTypeImages = new Dictionary<T
errainType, Texture2D>();

public static void LoadTerrainImages()
{
    Texture2D plains = ResourceLoader.Load("res://textures/plains.jpg") as Texture2D;
    Texture2D beach = ResourceLoader.Load("res://textures/beach.jpg") as Texture2D;
    Texture2D desert = ResourceLoader.Load("res://textures/desert.jpg") as Texture2D;
    Texture2D mountain = ResourceLoader.Load("res://textures/mountain.jpg") as Textur
e2D;
    Texture2D ice = ResourceLoader.Load("res://textures/ice.jpg") as Texture2D;
```



```
    Texture2D ocean = ResourceLoader.Load("res://textures/ocean.jpg") as Texture2D;
    Texture2D shallow = ResourceLoader.Load("res://textures/shallow.jpg") as Texture2
D;
    Texture2D forest = ResourceLoader.Load("res://textures/forest.jpg") as Texture2D;

    terrainTypeImages = new Dictionary<TerrainType, Texture2D>
    {
        { TerrainType.PLAINS, plains },
        { TerrainType.BEACH, beach },
        { TerrainType.DESERT, desert },
        { TerrainType.MOUNTAIN, mountain },
        { TerrainType.ICE, ice },
        { TerrainType.WATER, ocean },
        { TerrainType.SHALLOW_WATER, shallow },
        { TerrainType.FOREST, forest },
    };
}
```

Now, we need to call this function somewhere in our program. We'll do that right when our game node is loaded in. We'll override the EnterTree function in our Game class:

```
public override void _EnterTree()
{
    TerrainTileUI.LoadTerrainImages();
}
```

Finally, we can assign these textures in our SetHex function:

```
terrainImage.Texture = terrainTypeImages[hex.terrainType];
```

With that, our system should be working. We have textures for all the different terrain types, and we can also see a string representation of the terrain type.

In the next video, we'll take care of one more thing we need to do for this UI system that will also be helpful with later UIs – what happens when we click off the map.

In this lesson, we will focus on enhancing our UI system by adding the ability to deselect the UI when clicking outside the map. This feature will make our UI more user-friendly and prepare us for future UI implementations. We will leverage Godot's built-in signal system to achieve this.

Deselecting the UI

Currently, when we click on a terrain tile, the UI is permanently fixed to the screen. To improve this, we will allow users to click outside the map to deselect the UI. This involves creating a function to hide or destroy the current UI popups and connecting this function to a signal.

Creating the hideAllPopups Function

First, we need to create a function called hideAllPopups in our UIManager class. This function will destroy the current terrain UI and ensure that we are not trying to call this on something that doesn't exist. For now, we can target our terrain UI and destroy it if it isn't null.

```
public void HideAllPopups()
{
    if (terrainUi is not null)
    {
        terrainUi.QueueFree();
        terrainUi = null;
    }
}
```

Creating a Signal in HexTileMap

Next, we will create a signal in our HexTileMap class that can be sent out whenever we click outside the boundaries of the map. We will use Godot's built-in signal system for this.

```
[Signal]
public delegate void ClickOffMapEventHandler();
```

Emitting the Signal

We will emit this signal in the _UnhandledInput method where we detect whether we have clicked off the map.

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse)
    {
        Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));

        if (mapCoords.X >= 0 && mapCoords.X < width && mapCoords.Y >= 0 && mapCoords.
Y < height)
        {

            if (mouse.ButtonMask == MouseButtonMask.Left)
            {
                Hex h = mapData[mapCoords];
```



```
                SendHexData?.Invoke(h);

                if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSel
ectedCell, -1);
                overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
                currentSelectedCell = mapCoords;
            }
        }
        else
        {
            overlayLayer.SetCell(currentSelectedCell, -1);
            EmitSignal(SignalName.ClickOffMap);
        }
    }
}
```

Connecting the Signal in the Editor

Finally, we will connect the signal to the hideAllPopups function in the editor. This involves building the project to register our signals and then connecting the signal to the UIManager node.

1. Build the project to register the signals.
2. Go to the HexTileMap node in the editor.
3. Connect the ClickOffMap signal to the UIManager node's hideAllPopups function.

Testing the Implementation

With these changes, we can now test our implementation. We should be able to click around the map and also click off the map to get rid of our UI. This completes our first fully functional piece of UI and sets us up for easier implementation of future UIs.

With this, we are ready to move on to subsequent videos to create our first real elements of gameplay, such as creating cities.

Welcome back to our Godot 4X series where we're building a hex-based strategy game in Godot Engine using C#. So far, we've made significant progress by enhancing our map, making it interactable, and creating a user-friendly UI system. Now, it's time to start adding some actual gameplay elements to this map. Our eventual goal is to have a two-tiered gameplay system where factions or civilizations can control multiple cities on the map. In the next several videos, we'll be implementing the base of that system, which is the city.

Creating a New Scene for the City

Before we write any code for the city, let's create a new scene to represent a city in our game. We'll create a new 2D scene and name it "City".

Adding Child Nodes

Our city display on the map is going to be pretty simple. It's going to be a sprite and a label. So, we'll add those child nodes here:

- **Sprite2D:** This will be a container for a simple graphic to represent a city on the map.
- **Label:** This will display the city's name.

Setting Up the Sprite2D

We'll use a pre-imported texture called "city.png" for our Sprite2D. Here's how to set it up:

1. Add a Sprite2D node as a child of the City node.
2. Drag the "city.png" texture into the texture slot of the Sprite2D node.
3. Ensure the offset is centered. You might need to check a box to make sure the city graphic is centered.

Setting Up the Label

Next, we'll set up the label to display the city's name:

1. Add a Label node as a child of the City node.
2. Set the default text to "City Name".
3. Change the label color to black to make it more readable on the map.
4. Increase the font size slightly for better visibility.

Attaching a Script to the City Node

Now that we have the graphical representation in place, we can attach a script to our city node. We'll call this script "City.cs". This script will contain the logic and data needed for our cities to function in the game.

Initializing the City Class

Our cities will eventually be able to do quite a bit, such as holding territory, growing population each turn, and spawning units. For now, we'll set up the basic structure of our city class. Here's what we'll include:

- A reference to the main map.
- A vector to hold the center coordinates of the city.
- A list of hexes to represent the city's territory.
- A list to keep track of possible tiles the city can expand into (border tile pool).
- A name for the city.

- References to the city scene's label and sprite.

City.cs Code Snippet

Here's the initial code for our City.cs script:

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

public partial class City : Node2D
{
    public HexTileMap map;
    public Vector2I centerCoordinates;

    public List<Hex> territory;
    public List<Hex> borderTilePool;

    public string name;

    Label label;
    Sprite2D sprite;
}
```

In the next video, we'll start to write some actual logic for initializing our city.

Welcome back to our course on implementing city mechanics for a strategy game. In the previous lessons, we laid the groundwork for our city mechanics by creating a basic city scene and a skeleton script for our city class. We also discussed the structure of our game, which includes cities and an overarching entity called a civilization. In this lesson, we will create the civilization class and start populating our map with cities.

Understanding the Game Structure

Our game will have two main entities:

- **Cities:** These are the main gameplay locus and focus. They control territory, have population, resources, and more.
- **Civilizations:** These are containers for cities. Each civilization controls multiple cities and has attributes like color and name.

Creating the Civilization Class

Since civilizations are abstract entities without any graphics or particular scene, we only need a script for this class. Let's create a new script called Civilization.cs.

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

public class Civilization
{
    public int id;
    public List<City> cities;
    public Color territoryColor;
    public string name;
    public bool playerCiv;

    public Civilization()
    {
        cities = new List<City>();
    }
}
```

In this script, we define the following member variables:

- id: An identifier for the civilization.
- cities: A list of cities that the civilization controls.
- territoryColor: The color that represents the civilization on the map.
- name: The name of the civilization.
- playerCiv: A boolean indicating whether this is the player's civilization.

We also define a constructor that initializes the list of cities.

Populating the Map with Cities

Now that we have our civilization class, we can start populating our map with cities. We will create a new function in our HexTileMap class called CreateCity.

First, let's import the packed scene for our city in the _Ready function of the HexTileMap class:

```
PackedScene cityScene;

public override void _Ready()
{
    cityScene = ResourceLoader.Load<PackedScene>("City.tscn");
    // Other initialization code...
}
```

Next, let's define the CreateCity function:

```
public void CreateCity(Civilization civ, Vector2I coords, string name)
{
    City city = cityScene.Instantiate() as City;
}
```

This function takes three parameters:

- civ: The civilization that the city belongs to.
- coords: The coordinates for the city center.
- name: The name of the city.

In this function, we instantiate a new city scene for now.

Conclusion

In this lesson, we created the civilization class and started populating our map with cities. In the next lesson, we will continue to build on these mechanics and add more functionality to our city class.

Welcome back to our course on creating a city in Godot! In this lesson, we will continue working on our createCity function. This function will involve jumping back and forth between different classes, so bear with the process. We will start by setting some initial properties for our city and then outline the tasks we need to complete to fully create a city.

Setting Initial Properties

Before we dive into the more complex parts of our createCity function, let's take care of some straightforward tasks. We have already instantiated a new city scene. Now, we need to set a few member variables for our city class.

Adding a Civilization Reference

First, we need to add a new member variable to the City class to reference the civilization that owns the city. We will call this variable civ.

```
public Civilization civ;
```

Setting the Map Reference

Next, we need to give our city a reference to the map. Every city will need to interact with the map quite a bit, so we will set the map variable of the city to the HexTileMap class that we are in.

```
city.map = this;
```

Adding the City to the Civilization

We also need to add the new city to the list of cities in the civilization class. This will register the city with the particular civilization that it should be a part of.

```
civ.cities.Add(city);
```

Setting the Civilization of the City

Finally, we need to set the civilization of our city to the civilization that we are working with. This ensures that both the city and the civilization have references to each other.

```
city.civ = civ;
```

Adding the City to the Scene Tree

Now that we have set the necessary references, we can add the city to the scene tree. This can technically be done anywhere after we have instantiated the city, but we will do it here to get it over with.

```
AddChild(city);
```


Outlining Tasks for Creating a City

With the initial properties set, let's outline the tasks we need to complete for our city. These tasks are fairly complicated, so we will write them out in comments for now.

- Set the color of the city's icon to reflect the color of the civilization.
- Set the city's name.
- Set the coordinates of the city, including both the map coordinates and the actual world coordinates.
- Add territory to the city. This includes setting default territory and handling edge cases to prevent glitchy overlap with other cities.

Setting the Coordinates of the City

Setting the coordinates of the city is fairly straightforward and can be done within our HexTileMap class. We will assume that the createCity function requires coordinates as an argument.

First, we will set the centerCoordinates variable of the city to the coordinates provided. This represents the city's location on the map grid.

```
city.centerCoordinates = coords;
```

Next, we will change the city's actual position. This is the Node2D's position attribute, which sets where the city graphic is positioned in the 2D space in Godot.

```
city.Position = baseLayer.MapToLocal(coords);
```

To convert the grid coordinates to pixel coordinates, we will use the MapToLocal function on the baseLayer of our HexTileMap class.

Conclusion

With these initial properties set and the tasks outlined, we are ready to tackle the more complex parts of creating a city in the next video. Stay tuned!

Welcome back to our course on creating a 4X game in Godot using C#. In this lesson, we will focus on implementing a function to add territory to our cities. This is an essential feature as it allows cities to have surrounding tiles from which to draw resources like food and production. Additionally, this will enable cities to expand their territories as the game progresses.

Understanding the Add Territory Function

The function we will implement is called AddTerritory. This function will be part of our City class. The primary goal of this function is to allow cities to claim multiple hexes as part of their territory.

Function Parameters

The AddTerritory function will take a list of hexes as a parameter. This design choice allows us to add multiple hexes to a city's territory in one go. Although this means we need to provide a list even if we want to add just one hex, it provides flexibility for future expansions.

Implementing the Add Territory Function

Let's start by writing the basic structure of the AddTerritory function in our City class.

```
public void AddTerritory(List<Hex> territoryToAdd)
{
    // Register each hex as owned by this city
    foreach (Hex h in territoryToAdd)
    {
        h.ownerCity = this;
    }

    territory.AddRange(territoryToAdd);
}
```

In this function, we use the AddRange method to add the list of hexes to the city's territory. We also loop through each hex in the list and set its ownerCity property to the current city. This ensures that each hex knows which city it belongs to, which is crucial for game logic and visual representation.

Updating the Hex Class

To support the ownership feature, we need to add a member variable to our Hex class. This variable will store a reference to the city that owns the hex.

```
public class Hex
{
    public readonly Vector2I coordinates;
    public TerrainType terrainType;
    public int food;
    public int production;
    public City ownerCity; // New member variable

    public Hex(Vector2I coords)
    {
        this.coordinates = coords;
        ownerCity = null; // Initialize ownerCity to null
    }
```



```
}
```

Adding Territory in the Create City Function

Now that we have the AddTerritory function, we need to use it in our CreateCity function to add the initial territory to a city. This includes the city center hex and the surrounding hexes.

```
public void CreateCity(Civilization civ, Vector2I coords, string name)
  {
    City city = cityScene.Instantiate() as City;
    city.map = this;

    civ.cities.Add(city);
    city.civ = civ;

    AddChild(city);

    // Set the color of the city's icon

    // Set the city's name

    // Set the coordinates of the city
    city.centerCoordinates = coords;
    city.Position = baseLayer.MapToLocal(coords);

    // Adding territory to the city
    city.AddTerritory(new List<Hex>{mapData[coords]});

    // Add the surrounding territory
  }
```

Finding Surrounding Hexes

To find the surrounding hexes, we will create a helper function called GetSurroundingHexes. This function will take coordinates as input and return a list of surrounding hexes.

```
public List<Hex> GetSurroundingHexes(Vector2I coords)
{
    List<Hex> result = new List<Hex>();

    foreach (Vector2I coord in baseLayer.GetSurroundingCells(coords))
    {
        // More code
    }

    return result;
}
```

In this function, we use the GetSurroundingCells method from the TileMapLayer node to get the surrounding cells. We then check if each cell is within the bounds of the map and add the

corresponding hex to the result list (which we'll cover more in the next lesson).

Conclusion

In this lesson, we implemented a basic version of the AddTerritory function in our City class. We also updated the Hex class to include an owner city and modified the CreateCity function to add the initial territory to a city. Additionally, we created a helper function to find the surrounding hexes.

This implementation provides a foundation for adding territory to cities. In future lessons, we will refine this function and integrate it with other systems to ensure that cities play nicely with each other on the map.

Welcome back! In this lesson, we continue our journey of adding a territory to our cities when they are spawned into the map. So far, we have created a basic `AddTerritory` function for our city and started implementing it in our `CreateCity` function within the `HexTileMap` class. Now, we need to ensure that the surrounding territory is added correctly to give the city some substance.

Adding Surrounding Territory

To achieve this, we need to write a helper function called `GetSurroundingHexes`. This function will take a coordinate on our hex grid and return all the surrounding cells. However, we need to ensure that the cells we add are valid and within the boundaries of the map to avoid crashes or errors.

Checking Boundaries

To check if a hex is within the bounds of the map, we will write another helper function called `HexInBounds`. This function will take some coordinates and check if they are within the map's width and height. If the coordinates are outside the map, the function will return `false`; otherwise, it will return `true`.

```
public bool HexInBounds(Vector2I coords)
{
  if (coords.X < 0 || coords.X >= width ||
    coords.Y < 0 || coords.Y >= height)
    return false;

  return true;
}
```

Implementing GetSurroundingHexes

Now, we can confidently use the `HexInBounds` function in our `GetSurroundingHexes` function to ensure that only valid hexes are added to the results.

```
public List<Hex> GetSurroundingHexes(Vector2I coords)
{
    List<Hex> result = new List<Hex>();

    foreach (Vector2I coord in baseLayer.GetSurroundingCells(coords))
    {
        if (HexInBounds(coord))
            result.Add(mapData[coord]);
    }

    return result;
}
```

Adding Surrounding Territory to the City

With the `GetSurroundingHexes` function ready, we can now add the surrounding territory to the city in the CreateCity function. We will create a new list of hexes called `surroundingHexes` and call `GetSurroundingHexes` on the city center's coordinates.

```
List<Hex> surroundingHexes = GetSurroundingHexes(city.centerCoordinates);
```


Checking for Ownership

Before adding the surrounding hexes to the city's territory, we need to ensure that these hexes are not already owned by another city. We will iterate through the `surroundingHexes` list and check if the `ownerCity` of each hex is `null`. If it is, we will add the hex to the city's territory.

```
foreach (Hex hex in surroundingHexes)
{
    if (hex.ownerCity == null)
    {
        city.AddTerritory(new List<Hex> { hex });
    }
}
```

Setting City Name and Icon Color

Finally, we need to set the city's name and icon color. We will add two new functions to the `City` class: `SetCityName` and `SetIconColor`.

```
public void SetCityName(string newName)
{
    name = newName;
    label.Text = newName;
}

public void SetIconColor(Color c)
{
    sprite.Modulate = c;
}
```

In the `CreateCity` function, we will call these new functions to set the city's name and icon color based on the civilization's color.

```
city.SetCityName(name);
city.SetIconColor(civ.territoryColor);
```

Conclusion

With these steps, we have successfully added the city center coordinate and the surrounding territory to the city. We have also ensured that the territory is not already owned by another city. Additionally, we have set the city's name and icon color based on the civilization's color. In the next lesson, we will circle back to some more peripheral but important tasks for setting up these cities.

Welcome back! We are nearing the completion of our `createCity` function, which is a crucial part of spawning cities on our map. This function involves several behind-the-scenes tasks to ensure that cities are properly registered and displayed on the map. In this lesson, we will focus on adding some essential data to our `HexTileMap` class and updating our `createCity` function to register this data.

Adding Gameplay Data to HexTileMap

Currently, our `HexTileMap` class does not have a way to keep track of all the cities present on the map. This information is vital for various gameplay logic reasons. Therefore, we need to add some new data to our `HexTileMap` class.

- Create a dictionary to store coordinates as keys and cities as values.
- Create a list to keep track of all civilizations present on the map.

Let's add these data structures to the `HexTileMap` class:

```
public partial class HexTileMap : Node2D
{
    // GAMEPLAY DATA
    public Dictionary<Vector2I, City> cities;
    public List<Civilization> civs;

    // Other existing code...
}
```

Next, we need to instantiate these lists in the `_Ready` method of the `HexTileMap` class:

```
public override void _Ready()
{
    // Existing code...

    // GAMEPLAY DATA
    cities = new Dictionary<Vector2I, City>();
    civs = new List<Civilization>();

    // Other existing code...
}
```

Updating the createCity Function

Now that we have added the necessary data structures to our `HexTileMap` class, we need to update our `createCity` function to register this data. Specifically, we need to add the newly created city to the `cities` dictionary with the city center's coordinates as the key.

Let's update the `createCity` function accordingly:

```
public void CreateCity(Civilization civ, Vector2I coords, string name)
{
    City city = cityScene.Instantiate() as City;
    city.map = this;
    civ.cities.Add(city);
```



```
    city.civ = civ;

    AddChild(city);

    // Set the color of the city's icon
    city.SetIconColor(civ.territoryColor);

    // Set the city's name
    city.SetCityName(name);

    // Set the coordinates of the city
    city.centerCoordinates = coords;
    city.Position = baseLayer.MapToLocal(coords);

    // Adding territory to the city
    city.AddTerritory(new List<Hex>{mapData[coords]});

    // Add the surrounding territory
    List<Hex> surrounding = GetSurroundingHexes(coords);

    foreach (Hex h in surrounding)
    {
        if (h.ownerCity == null)
            city.AddTerritory(new List<Hex>{h});
    }

    UpdateCivTerritoryMap(civ);

    // Register the city in the cities dictionary
    cities[coords] = city;
}
```

Registering City Center Data

We also need to ensure that our hexes know whether they are a city center or not. This information will be useful for gameplay logic later on. Let's add a boolean variable `isCityCenter` to the `Hex` class and set it to `false` by default:

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

    public TerrainType terrainType;

    public int food;
    public int production;

    public City ownerCity;
    public bool isCityCenter = false;

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



```
        ownerCity = null;
    }

    // Other existing code...
}
```

Now, when we create a city, we need to set the `isCityCenter` property of the corresponding hex to `true`:

```
public void CreateCity(Civilization civ, Vector2I coords, string name)
{
    City city = cityScene.Instantiate() as City;
    city.map = this;
    civ.cities.Add(city);
    city.civ = civ;

    AddChild(city);

    // Set the color of the city's icon
    city.SetIconColor(civ.territoryColor);

    // Set the city's name
    city.SetCityName(name);

    // Set the coordinates of the city
    city.centerCoordinates = coords;
    city.Position = baseLayer.MapToLocal(coords);
    mapData[coords].isCityCenter = true;

    // Adding territory to the city
    city.AddTerritory(new List<Hex>{mapData[coords]});

    // Add the surrounding territory
    List<Hex> surrounding = GetSurroundingHexes(coords);

    foreach (Hex h in surrounding)
    {
        if (h.ownerCity == null)
            city.AddTerritory(new List<Hex>{h});
    }

    UpdateCivTerritoryMap(civ);

    // Register the city in the cities dictionary
    cities[coords] = city;
}
```

Painting the City's Territory

Finally, we need to create a function that will paint the city's territory on the map. This function will be called `UpdateCivTerritoryMap` and will be defined at the level of the civilization rather than a

particular city. This function will loop through all the cities for a particular civilization and then loop through the hexes that each city owns to set the color of the cells on the map.

```
public void UpdateCivTerritoryMap(Civilization civ)
  {
    foreach (City c in civ.cities)
    {
      foreach (Hex h in c.territory)
      {

      }
    }
  }
```

However, setting up the graphical part of this function will require creating a new map layer in the Godot editor, which we will cover in the subsequent video.

That’s it for this lesson! We have made significant progress in our `createCity` function and are now ready to move on to the next steps.

In this lesson, we will set up a new TileMap layer in the Godot editor to handle the coloring of tiles for different cities. This involves creating a new TileMap layer, setting its properties, and preparing it for use in our code. Let's go through the steps one by one.

Creating a New TileMap Layer

First, we need to create a new TileMap layer in the Godot editor. Follow these steps:

1. Open the Godot editor and navigate to your scene.
2. Add a child node to your scene by clicking the "+" button in the Scene tab.
3. Select "TileMapLayer" from the list of available nodes.
4. Name the new layer "CivColorsLayer" and place it between the "BaseLayer" and "HexBordersLayer" in the hierarchy.

Setting the TileMap Layer Properties

Next, we need to set some properties for our new TileMap layer:

- Assign a TileSet to the layer. In this case, we will use the standard terrain tileset.
- Set the modulate property of the layer to have an alpha value of 60. This will make the tiles slightly transparent, allowing us to see the terrain underneath.

Preparing the TileSet

We will use a specific tile from the terrain tileset that can be easily modulated with different colors. In this case, we will use the ice tile:

```
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) },
    { TerrainType.CIV_COLOR_BASE, new Vector2I(0, 3) },
};
```

Updating the Code

Now that we have set up our TileMap layer in the editor, we need to update our code to load and use this layer. First, we need to add the layer to our list of layers and load its resource:

```
baseLayer = GetNode<TileMapLayer>("BaseLayer");
borderLayer = GetNode<TileMapLayer>("HexBordersLayer");
overlayLayer = GetNode<TileMapLayer>("SelectionOverlayLayer");
civColorsLayer = GetNode<TileMapLayer>("CivColorsLayer");
```

Next, we need to update our `UpdateCivTerritoryMap` function to set the cell on the

`CivColorsLayer` to the appropriate tile:

```
public void UpdateCivTerritoryMap(Civilization civ)
{
    foreach (City c in civ.cities)
    {
        foreach (Hex h in c.territory)
        {
            civColorsLayer.SetCell(h.coordinates, 0, terrainTextures[TerrainType.CIV_
COLOR_BASE], civ.territoryColorAltTileId);
        }
    }
}
```

Using Alternative Tiles

Godot's TileMap layers have a system called alternative tiles, which allows us to create variants of existing base tiles. We can use this system to create alternative tiles for each civilization's color. To do this, we need to add a new variable to our `Civilization` class to store the ID of the alternative tile:

```
public class Civilization
{
    public int id;
    public List<City> cities;
    public Color territoryColor;
    public int territoryColorAltTileId;
    public string name;
    public bool playerCiv;

    public Civilization()
    {
        cities = new List<City>();
    }

    // ... other methods ...
}
```

Finally, we need to update our `CreateCity` function to call the `UpdateCivTerritoryMap` function after creating a new city:

```
public void CreateCity(Civilization civ, Vector2I coords, string name)
{
    City city = cityScene.Instantiate() as City;
    city.map = this;
    civ.cities.Add(city);
    city.civ = civ;

    AddChild(city);

    city.SetCityName(name);
```



```
    city.SetIconColor(civ.territoryColor);
    city.centerCoordinates = coords;
    city.Position = baseLayer.MapToLocal(coords);
    mapData[coords].isCityCenter = true;

    city.AddTerritory(new List<Hex>{mapData[coords]});

    List<Hex> surrounding = GetSurroundingHexes(coords);
    foreach (Hex h in surrounding)
    {
        if (h.ownerCity == null)
            city.AddTerritory(new List<Hex>{h});
    }

    UpdateCivTerritoryMap(civ);

    cities[coords] = city;
}
```

With these steps, we have successfully set up a new TileMap layer to handle the coloring of tiles for different cities. In the next lesson, we will move on to instantiating some civilizations, creating those alternative tiles, and actually tying this whole thing together by spawning a bunch of cities on the map, figuring out where to place them, and all of that good stuff. Thank you for watching, and I'll see you there!

Welcome back! In our previous lesson, we successfully created a function to generate cities. Now, we need to determine where these cities will be placed on the map. This task involves ensuring that cities are only spawned on valid locations, such as plains tiles, to avoid placing them in impassable areas like mountains or bodies of water.

Objective

Our goal in this lesson is to create a function that generates valid starting locations for civilizations. This function will return a list of coordinates, each representing a valid city starting location.

Steps to Generate Starting Locations

1. Create a new function called generateCivStartingLocations that returns a list of Vector2 integers.
2. Initialize a list to store the valid plains tiles.
3. Iterate through the map to collect all plains tiles.
4. Use a random number generator to select valid starting locations from the list of plains tiles.
5. Implement a helper function to validate the selected locations.

Let's start by defining the generateCivStartingLocations function. This function will take one parameter, which is the number of locations we want to generate.

```
public List<Vector2I> generateCivStartingLocations(int numLocations)
{
    List<Vector2I> locations = new List<Vector2I>();
    List<Vector2I> plainsTiles = new List<Vector2I>();

    // Collect all plains tiles
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            if (mapData[new Vector2I(x, y)].terrainType == TerrainType.PLAINS)
            {
                plainsTiles.Add(new Vector2I(x, y));
            }
        }
    }

    Random r = new Random();
    for (int i = 0; i < numLocations; i++)
    {
        Vector2I coord = new Vector2I();
        bool valid = false;
        int counter = 0;

        while (!valid && counter < 10000)
        {
            coord = plainsTiles[r.Next(plainsTiles.Count)];

    }

    return locations;
}
```


Helper Function: isValidLocation

To determine if a location is valid, we need to create a helper function called isValidLocation. This function will check if the location is too close to existing locations and if it is within the valid range of the map. For now we'll just make the skeleton, and write the function in full next lesson.

```
private bool isValidLocation(Vector2I coord, List<Vector2I> locations)
{

}
```

Summary

In this lesson, we created a function to generate valid starting locations for civilizations. We collected all plains tiles, used a random number generator to select valid locations, and started a helper function to validate these locations. In the next lesson, we will finish this helper function and complete the logic for generating starting locations.

Stay tuned for the next lesson!

Welcome back! In the previous video, we worked on generating starting locations for our civilizations. To do this, we needed to create a helper function to determine whether a given tile or coordinate is a valid location. Let's dive into the logic required to make this determination.

Determining Valid Locations

There are a couple of things we need to ensure when determining if a location is valid:

- The city should not spawn too close to the edge of the map, as this could lead to issues with adding territory.
- The city should not spawn too close to other cities, to prevent territories from overlapping.

Boundary Checking

First, let's handle the logic for ensuring that the coordinate is not too close to the edge of the map. This involves some boundary checking with the x and y coordinates. We want to ensure that the city is at least three tiles away from the edge of the map.

```
if (coord.X < 3 || coord.X > width - 3 ||
  coord.Y < 3 || coord.Y > height - 3)
{
  return false;
}
```

If the coordinate does not pass this test, we return false because it is too close to the edge of the map and should be rejected.

Checking Proximity to Other Cities

Next, we need to ensure that the location is not too close to an already generated location. We will loop through the locations in our locations parameter and test that these are far enough away from each other.

```
foreach (Vector2I l in locations)
{
  if (Math.Abs(coord.X - l.X) < 20 || Math.Abs(coord.Y - l.Y) < 20)
    return false;
}
```

If the location is within 20 tiles of any previously generated location, we return false because it is too close and should be rejected.

Valid Location

If the location passes all of these tests, we return true, indicating that it is a good, valid location.

```
return true;
```

Integrating the Helper Function

Now, let's return to the GenerateCivStartingLocations function and integrate our helper function. We will use our while loop to generate a valid location.

```
while(!valid && counter < 10000)
{
    coord = plainsTiles[r.Next(plainsTiles.Count)];
    valid = IsValidLocation(coord, locations);
    counter++;
}
```

After the while loop is broken, we know that a given location has passed the test. Here's what we need to do next:

1. Remove the coordinate from the list of plain tiles to prevent later ones from selecting this coordinate as well.
2. Remove a buffer zone around this coordinate from the list of plain tiles for performance reasons.

Removing the Buffer Zone

We will use the GetSurroundingHexes helper function to create a buffer zone around the coordinate. This will ensure that not only the surrounding hexes of the coordinate get removed, but also the surrounding hexes of the surrounding hexes, and so on.

```
plainsTiles.Remove(coord);
foreach (Hex h in GetSurroundingHexes(coord))
{
  foreach (Hex j in GetSurroundingHexes(h.coordinates))
  {
    foreach (Hex k in GetSurroundingHexes(j.coordinates))
    {
      plainsTiles.Remove(h.coordinates);
      plainsTiles.Remove(j.coordinates);
      plainsTiles.Remove(k.coordinates);
    }
  }
}
```

Finally, we can add this tile that has passed all these tests and trials of our spawning logic to our final locations list.

```
locations.Add(coord);
```

With this, we have created a self-contained function that will allow us to generate starting locations for our civilizations. In the next video, we can finally start to put all of this together by generating some civilizations and actually spawning them on the map, seeing the fruits of our labor over the last several videos in spawning cities and civilizations.

Hello and welcome back. We have completed almost all the functions needed to generate civilizations and cities on the map at the start of the game. So far, we have created a function to create a new city on the map and another function to select spawning locations for cities on the map. Let's take a look at the main function in our HexTileMap class to see the flow we are starting to create.

Recap of Functions Created

- Function to create a new city on the map.
- Function to select spawning locations for cities on the map.

Generating Starting Locations

In our main function for the HexTileMap class, we instantiate some containers for our civilizations and cities for the whole map game. We then generate some starting locations using the function we have already created.

```
// CIVILIZATION AND CITIES GEN
civs = new List<Civilization>();
cities = new Dictionary<Vector2I, City>();

// Generate starting locations
List<Vector2I> starts = GenerateCivStartingLocations(NUM_AI_CIVS + 1);
```

The question is, how many of these starting locations do we want to generate? This is an opportunity to parameterize our game a little bit. We can create a new exported integer in the gameplay settings called numberOfAICivilizations. This will allow us to customize how many AI civilizations are on the map when we start the game later on. Remember, since we also have a player civilization, we need to add one to this number in the snippet above to generate enough locations.

```
[Export]
public int NUM_AI_CIVS = 6;
```

Now, if everything went well with generating starting locations, we should have a nice list of starts. We have two more tasks remaining here: generating the player civilization and generating the AI civilizations. These two processes are fairly similar but require two separate functions due to a few key differences in the process of creating a player civilization and creating an AI civilization.

Generating AI Civilizations

Since the player civilization is the exception here, we will write the function to generate the AI civilizations first. We can create a function called GenerateAICivs that takes in a list of Vector2I coordinates representing all the starting locations that we generated.

```
public void GenerateAICivs(List<Vector2I> civStarts)
{
}
```

Creating the AI Civilization Function

In this function, we need to iterate through all the starting locations and deal with each of them individually. For each starting location, we will:

1. Create a new civilization object.
2. Initialize fields such as the ID and whether it is a player civilization.
3. Assign the civilization a random color (which we'll cover separately)
4. Create the starting city for the civilization. To create the starting city for the civilization, we will use the CreateCity function that we have already created. We will pass in the current civilization, the starting coordinates, and a name for the city.
5. Register the new civilization with the map.

```
public void GenerateAICivs(List<Vector2I> civStarts)
{
  for (int i = 0; i < civStarts.Count; i++)
  {
    Civilization currentCiv = new Civilization
    {
      id = i + 1,
      playerCiv = false
    };

    // Assign a random color

    // Create starting city
    CreateCity(currentCiv, civStarts[i], "City " + civStarts[i].X);
    civs.Add(currentCiv);

  }
}
```

Assigning a Random Color

Assigning a random color to the AI civilization requires a new function in the Civilization class. We will create a function called SetRandomColor that generates a random color for the civilization.

```
public void SetRandomColor()
{
    Random r = new Random();
    territoryColor = new Color(r.Next(255)/255.0f, r.Next(255)/255.0f, r.Next(255)/25
5.0f);
}
```

Back in our HexTileMap class, we can call this function to assign our civilization a random color.

```
currentCiv.SetRandomColor();
```

Next Steps

The next task is to ensure that the actual tiles in our tile set reflect this civilization's random color. This is going to be a little bit more involved and will be taken care of in the next video.

In the previous lesson, we began writing a function to generate AI factions for our game. We left off with the task of creating a new tile to reflect a particular faction's randomly assigned color. In this lesson, we will complete this task by working with base tiles and alternative tiles in Godot.

Understanding Base Tiles and Alternative Tiles

In Godot, tile sets have the concept of base tiles and alternative tiles. Alternative tiles are variants of a base tile. For our game, we will create an alternative tile for each civilization that reflects the color we generated for that civilization.

Setting Up the Tile Atlas Source

To work with these tiles programmatically, we need a reference to the tile atlas. Godot provides a type of object called TileSetAtlasSource for this purpose. We will declare this in our member variables and assign it properly in our _Ready function.

```
TileSetAtlasSource terrainAtlas;
```

In the _Ready function, we will assign the tile atlas source:

```
terrainAtlas = civColorsLayer.TileSet.GetSource(0) as TileSetAtlasSource;
```

Creating Alternative Tiles

Now that we have access to the tile set source, we can start to create alternative tiles. This is a two-step process:

1. Create a new alternative tile.
2. Set the alternative tile's modulate property to reflect the color assigned to the civilization.

First, we create a new alternative tile. The syntax for this is a bit unusual. When we create a new alternative tile, we don't get back a tile object. Instead, we get an ID number that we need to keep track of. This ID number allows us to indirectly modify the tile using the GetTileData function.

```
int id = terrainAtlas.CreateAlternativeTile(terrainTextures[TerrainType.CIV_COLOR_BAS
E]);
```

Next, we set the modulate property of the alternative tile to the color of the civilization:

```
terrainAtlas.GetTileData(terrainTextures[TerrainType.CIV_COLOR_BASE], id).Modulate =
civ.territoryColor;
```

Finally, we need to assign the ID of the alternative tile to the civilization's territoryColorAltTileId member:

```
civ.territoryColorAltTileId = id;
```


Fixing a Typo in the Get Surrounding Hexes Function

Before we can run our code and see the cities populated on the map, we need to fix a subtle typo in the GetSurroundingHexes function. Instead of adding the same tile over and over, we need to add the correct tile.

Make sure to replace coords with coord in the for each loop:

```
public List<Hex> GetSurroundingHexes(Vector2I coords)
{
  List<Hex> result = new List<Hex>();
  foreach (Vector2I coord in baseLayer.GetSurroundingCells(coords))
  {
    if (HexInBounds(coord))
      result.Add(mapData[coord]);
  }
  return result;
}
```

Running the Code

With these changes in place, we can now run our code and see our cities populated on the map with random colors, names, and other attributes. This marks the completion of the behind-the-scenes work required to create and color the cities.

In subsequent courses, we will add interactivity and gameplay elements to these city objects, such as spawning units, growing populations, and more. For now, congratulations on making it this far! You have laid the groundwork for a complex and engaging game.

In the previous lesson, we successfully tested our code to spawn AI civilizations on the map. Now, we will focus on spawning the player's civilization. This process will be similar to spawning AI civilizations but will require some logical separation. Let's dive into the steps to achieve this.

Creating the Player Civilization

To create the player civilization, we need to define a new function that will handle the logic for generating the player's civilization. This function will run before we generate the AI civilizations.

Steps to Create the Player Civilization

1. Define a new function called CreatePlayerCiv that returns a Civilization object.
2. Pass a single Vector2I of integers to this function, representing the starting location for the player civilization.
3. Set the ID of the player civilization to 0 to distinguish it from AI civilizations.
4. Set the playerCiv boolean property of the civilization object to true.
5. Assign a specific color to the player civilization, which can be changed later in the editor.
6. Create an alternative tile ID for the player's territory color and set its modulate property to the player's color.
7. Add the player civilization to the map's master list of civilizations.
8. Call the CreateCity function to create the player's city at the specified starting location.

Implementing the CreatePlayerCiv Function

Let's implement the CreatePlayerCiv function in the HexTileMap class:

```
public Civilization CreatePlayerCiv(Vector2I start)
{
    Civilization playerCiv = new Civilization();
    playerCiv.id = 0;
    playerCiv.playerCiv = true;
    playerCiv.territoryColor = new Color(PLAYER_COLOR);

    int id = terrainAtlas.CreateAlternativeTile(terrainTextures[TerrainType.CIV_COLOR
_BASE]);
    terrainAtlas.GetTileData(terrainTextures[TerrainType.CIV_COLOR_BASE], id).Modulat
e = playerCiv.territoryColor;

    playerCiv.territoryColorAltTileId = id;

    civs.Add(playerCiv);

    CreateCity(playerCiv, start, "Player City");

    return playerCiv;
}
```

Updating the Main Generation Pipeline

Now, we need to update the main generation pipeline to create the player civilization before generating the AI civilizations. We will call the CreatePlayerCiv function and pass the first generated starting location to it. After using this location, we will remove it from the list before passing the remaining locations to the function that generates the AI civilizations.


```
// Generate player civilization
Civilization playerCiv = CreatePlayerCiv(starts[0]);
starts.RemoveAt(0);

// Generate AI civilizations
GenerateAICivs(starts);
```

Testing the Player Civilization

After implementing the above steps, we should be able to see the player civilization generated alongside the AI civilizations on the map. Initially, the player civilization will have a default color (white in this case). We can change this color in the editor to any other color, such as blue, to verify that the player civilization is correctly generated with the specified color.

With this, we have completed the logic for generating cities, including both AI and player civilizations. In the subsequent lessons, we will start building up the gameplay logic for these cities, beginning with creating a unique UI for cities.

Stay tuned for the next lesson!

In the previous lesson, we created the graphics for the city UI that will appear when we click on city centers on the map. In this lesson, we will start laying the foundation in code for making that happen. We will ensure that our cities contain the relevant information that we need to display and create a script for the city UI scene to populate it with those values from a given city.

Updating the City Class

First, we need to make sure that our cities contain the relevant information that we want to display. We will add new member variables to the City class to keep track of the population, food, and production values.

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

public partial class City : Node2D
{
    // Population
    public int population = 1;

    // Resources
    public int totalFood;
    public int totalProduction;

    // Other member variables...
}
```

We will also add a new function to calculate the total food and production values for the city's territory.

```
public void CalculateTerritoryResourceTotals()
{
    totalFood = 0;
    totalProduction = 0;
    foreach (Hex h in territory)
    {
        totalFood += h.food;
        totalProduction += h.production;
    }
}
```

We will call this function whenever we add a new hex to a city's territory to recalculate its totals.

```
public void AddTerritory(List<Hex> territoryToAdd)
{
    // Other code...

    CalculateTerritoryResourceTotals();
}
```


Creating the CityUI Script

Next, we will create a new script for the city UI scene to populate it with the values from a given city. We will add references to the labels in the city UI scene and a reference to the associated city.

```
using Godot;
using System;

public partial class CityUI : Panel
{
    Label cityName, population, food, production;

    // City data
    City city;

    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {
        cityName = GetNode<Label>("CityName");
        population = GetNode<Label>("Population");
        food = GetNode<Label>("Food");
        production = GetNode<Label>("Production");
    }
}
```

We will also add a function to update the contents of the city UI with the values from a given city.

```
public void SetCityUI(City city)
{
    this.city = city;

    cityName.Text = this.city.name;
    population.Text = "Population: " + this.city.population;
    food.Text = "Food: " + this.city.totalFood;
    production.Text = "Production: " + this.city.totalProduction;
}
```

In the next lesson, we will finish out this pipeline by adding the relevant code to the UI manager and the hex tile map to actually get a signal from the map to the UI manager, which will manage the creation and deletion of these city UI panel instances.

Welcome back! In this lesson, we will continue setting up the data flow pipeline for displaying information about a city center on our city UI. We have already created the relevant data and updated it inside our city class. Additionally, we have created the basic script for the city UI itself, where we take a given city reference and populate the UI with data from that reference. Now, we will finish this pipeline by linking these elements to the UI manager and then via a signal with the map.

Updating the UI Manager

The next step in the pipeline is the UI manager. We already have some experience with this from our terrain tile UI. The city UI will work in a similar way. We need to ensure that we have a packed scene for the city UI, as we will need to create an arbitrary number of instances of this scene within our UI manager.

Steps to Update the UI Manager

1. Ensure you have a packed scene for the city UI.
2. Load the packed scene for the city UI in the UI manager.
3. Create a new function to set the city UI.
4. Update the `HideAllPopups` function to handle the city UI.

Loading the Packed Scene

First, we need to load the packed scene for the city UI. This is similar to how we loaded the terrain tile UI scene.

```
PackedScene cityUiScene;

public override void _Ready()
{
    cityUiScene = ResourceLoader.Load<PackedScene>("CityUI.tscn");
}
```

Creating the SetCityUI Function

Next, we create a new function called `SetCityUI` that will take a city reference and update the UI accordingly.

```
public void SetCityUI(City c)
{
    HideAllPopups();

    cityUi = cityUiScene.Instantiate() as CityUI;
    AddChild(cityUi);

    cityUi.SetCityUI(c);
}
```

Updating the HideAllPopups Function

We need to update the `HideAllPopups` function to handle the city UI. This function will ensure that the city UI is hidden when necessary.


```
public void HideAllPopups()
{
    if (terrainUi is not null)
    {
        terrainUi.QueueFree();
        terrainUi = null;
    }

    if (cityUi is not null)
    {
        cityUi.QueueFree();
        cityUi = null;
    }
}
```

Connecting the Signal

The last piece of the puzzle is to connect the signal from the map to the UI manager. We need to define a signal in the `HexTileMap` class and emit it when a city center is clicked.

Defining the Signal

We will define a new signal called `SendCityUIInfo` in the `HexTileMap` class.

```
[Signal]
public delegate void SendCityUIInfoEventHandler(City c);
```

Emitting the Signal

We will emit this signal in the `_UnhandledInput` function when a city center is clicked.

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse)
    {
        Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));

        if (mapCoords.X >= 0 && mapCoords.X = 0 && mapCoords.Y < height)
        {
            Hex h = mapData[mapCoords];
            if (mouse.ButtonMask == MouseButtonMask.Left)
            {
                if (cities.ContainsKey(mapCoords))
                {
                    EmitSignal(SignalName.SendCityUIInfo, cities[mapCoords]);
                }
                else
                {
                    SendHexData?.Invoke(h);
                }

                if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSel
```



```
ectedCell, -1);
                overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
                currentSelectedCell = mapCoords;
            }
        }
        else
        {
            overlayLayer.SetCell(currentSelectedCell, -1);
            EmitSignal(SignalName.ClickOffMap);
        }
    }
}
```

Connecting the Signal in the Editor

Finally, we need to connect the signal in the editor. In the `HexTileMap` node, connect the `SendCityUIInfo` signal to the `SetCityUI` function in the UI manager.

Testing the Pipeline

Now, we can test the pipeline by clicking on a city center and observing the city UI. If everything is set up correctly, the city UI should display the relevant information about the city.

Fixing Overlap Issues

You might notice an issue where clicking on a terrain tile directly after clicking on a city center does not hide the city UI. To fix this, update the `SetTerrainUI` function in the UI manager to call `HideAllPopups` instead of directly hiding the terrain UI.

```
public void SetTerrainUI(Hex h)
{
    HideAllPopups();

    terrainUi = terrainUiScene.Instantiate() as TerrainTileUI;
    AddChild(terrainUi);

    terrainUi.SetHex(h);
}
```

With these steps, our city UI is all hooked up and ready to go. You should now see the city UI displaying information about the city when you click on a city center.

Welcome back! In the previous lessons, we set up a pipeline for getting data from our city centers to the UI. In this lesson, we will embark on a new segment of the project: creating the turn system. Before we dive into implementing this system, let's address a small issue with the city UI highlighting. We'll then move on to creating the turn system, which will be the heart of our gameplay.

Fixing the City UI Highlighting

You may have noticed that when you click on a city, the map tile highlighting doesn't change as expected. To fix this, we need to adjust the code in the HexTileMap script. Specifically, we'll move the highlighting logic out of a nested if statement so that it happens regardless of whether the clicked tile is a city or not.

```
// Move this block out of the nested if statement
if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSelectedCell, -1);
overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
currentSelectedCell = mapCoords;
```

After making this adjustment, the city center highlighting should work as intended.

Creating the Turn System

The turn system will be the beating heart of our gameplay. Everything in our game will happen on a turn-by-turn basis, including population growth, unit movement, spawning units, and queuing units. Currently, our game is static, so creating a turn system is the first step towards making it dynamic and interactive.

Setting Up the General UI

To implement the turn system, we need to create a new UI element that will always be present on the screen. This UI will act like a nav bar, displaying the current turn number and providing a button to end the turn.

1. Create a new scene called GeneralUI that inherits from a Panel.
2. Set the panel's size to 650 pixels by 50 pixels and position it to the right of the existing UIs.
3. Add a Label to the panel to display the turn number. Set its text to "Turn 0" and adjust the font size to 24.
4. Add a Button to the panel to end the turn. Set its text to "End Turn" and position it in the middle of the UI.

General UI Script

Next, we'll set up a GeneralUI script to manage our interface. We first need to keep track of the turn number we're on and our label. In our ready function, we'll make sure to initialize the values in the interfact.

Then, in a new function we can call when the turn is incremented, we'll update the turns value and update the label.

```
// GeneralUI.cs
using Godot;
using System;
```



```
public partial class GeneralUI : Panel
{
    int turns = 0;
    Label turnLabel;

    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {
        turnLabel = GetNode("TurnLabel");
        turnLabel.Text = "Turn: " + turns;
    }

    public void IncrementTurnCounter()
    {
        turns += 1;
        turnLabel.Text = "Turn: " + turns;
    }
}
```

Next Steps

In the next lesson, we will hook up the GeneralUI system to the UIManager class to make the buttons and other UI elements functional. This will bring us one step closer to having a fully interactive game.

That's it for this lesson! See you in the next one.

Welcome back! In this lesson, we will continue setting up our turn system for the strategy game. Previously, we created the front-end graphics and a small script to control the turn counter and store the number of turns as a variable. Today, we will hook this up to our UI manager to enable the turn system to send signals and interface with the rest of the game.

Setting Up the General UI

Unlike the city UI and terrain UI, which are ephemeral instances that appear and disappear when we click on the map, the general UI will be a persistent navbar UI that sits at the top of the screen for the entire game. To achieve this, we will instantiate a child scene called GeneralUI inside our UIManager class. This will ensure that the general UI is always present and we don't need to spawn it in code.

Creating a Reference to the General UI

First, we need to create a reference to the GeneralUI in our UIManager class. Since we have already instantiated it, we don't need to load a packed scene. We just need to get ahold of the node. Remember that UI names are made lowercase by default by Godot, so make sure the casing is correct.

```
// Create a reference to the GeneralUI
GeneralUI generalUI = GetNode<GeneralUI>("GeneralUI");
```

Defining the End Turn Signal

Next, we need to define a signal that the UIManager will send out to tell the rest of the game that the end turn button has been clicked. This signal will be called EndTurn.

```
[Signal]
public delegate void EndTurnEventHandler();
```

Getting a Reference to the End Turn Button

To propagate the signal from the end turn button up the chain, we need to get a reference to the button inside our GeneralUI in our UIManager and attach a function to that signal.

```
// Get a reference to the end turn button
Button endTurnButton = generalUI.GetNode<Button>("EndTurnButton");
```

Creating the Intermediary Function

We need to create an intermediary function that will take the button's pressed signal and use it to send out the EndTurn signal. This function will also increment the turn counter.

```
public void SignalEndTurn()
{
    EmitSignal(SignalName.EndTurn);
```



```
    generalUI.IncrementTurnCounter();
}
```

Connecting the Button Signal to the Function

Finally, we need to connect the button's pressed signal to the SignalEndTurn function.

```
endTurnButton.Pressed += SignalEndTurn;
```

Handling the End Turn Signal in the HexTileMap

Now that the UIManager is set up to send out the EndTurn signal, we need to handle this signal in our HexTileMap class. We will create a new function called ProcessTurn that will represent what should happen when we end a turn. For now, this function will just print out "Turn ended".

```
public void ProcessTurn()
{
    GD.Print("Turn ended");
}
```

Connecting the UIManager's EndTurn Signal to the HexTileMap

In our HexTileMap class, we already have a section where we get the UIManager and attach some signals. We will add the EndTurn signal to this section and connect it to the ProcessTurn function.

```
// In the _Ready function of HexTileMap
UIManager uiManager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");
uiManager.EndTurn += ProcessTurn;
```

Testing the Turn System

With everything set up, we can now test our turn system in the Godot editor. When we click the end turn button, we should see the turn counter incrementing and the hex tile map catching the signal and printing "Turn ended".

Congratulations! You have successfully set up the turn system for your strategy game. In the subsequent lessons, we will implement city population growth mechanics that will take place on a turn-by-turn basis.

In the last section of the course, we implemented our new City UI system and TURN system. Now that we have a working system for keeping track of and sending signals out about the status of turns in our game, we're ready to add some actual gameplay mechanics that are attached to this turn system. For the remainder of the course, we're going to focus on creating a population growth mechanic for cities.

Overview of Population Growth Mechanic

Before we dive into the code, let's go over what we want this mechanic to do. Cities have food and production values, and they also have a population that starts at one. In this first gameplay mechanic, we want to tie our food to our population. After a certain number of turns accruing a certain amount of food from a city's territory, the population will grow, and the city's territory will expand.

Conceptually, this is a simple idea, but implementing it involves several tricky details. For instance, we don't want two cities to try and expand into the same tile at the same time. We need to ensure that cities play nicely with each other on the map.

Setting Up the City Class

Let's start by setting up our city class. We'll be using a list of hexes called borderTilePool that we created earlier in the course. This list will help us keep track of potential tiles for expanding into and improve performance while ensuring that cities play nicely with each other on the map.

First, we'll instantiate this borderTilePool list as a new list of hexes:

```
public List<Hex> borderTilePool;

public override void _Ready()
{
    borderTilePool = new List<Hex>();
}
```

Tracking Population Growth

Next, we need to keep track of some additional variables relevant to population growth. We'll add two new variables:

- populationGrowthThreshold: This keeps track of the next value of food needed to hit for the population to grow.
- populationGrowthTracker: This represents how close we are to passing that threshold.

We'll also create a static integer called POPULATION_THRESHOLD_INCREASE that represents the rate of change of the population growth threshold. This will make it harder for cities to grow as they get larger, preventing runaway exponential growth.

```
public static int POPULATION_THRESHOLD_INCREASE = 15;

public int populationGrowthThreshold;
public int populationGrowthTracker;
```

Processing Turns

Now, let's create a ProcessTurn function. This function will handle the logic for growing the population. Every turn, we'll increment the populationGrowthTracker by the amount of food the city has. If the populationGrowthTracker exceeds the populationGrowthThreshold, we'll increment the population, reset the tracker, and increase the threshold.

```
public void ProcessTurn()
{
    populationGrowthTracker += totalFood;
    if (populationGrowthTracker > populationGrowthThreshold) // Grow population
    {
        population++;
        populationGrowthTracker = 0;
        populationGrowthThreshold += POPULATION_THRESHOLD_INCREASE;
    }
}
```

Handling Invalid Tiles

To ensure that cities don't overlap and cause glitches, we'll create a static dictionary called invalidTiles. This dictionary will be shared between all cities and will help them keep tabs on each other to prevent expanding into territory that has already been claimed.

```
public static Dictionary<Hex, City> invalidTiles = new Dictionary<Hex, City>();
```

Conclusion

In this lesson, we've laid the foundation for our population growth mechanic. We've set up the necessary variables and functions in our city class to track and grow the population based on food. In the next few videos, we'll create consequences for this population growth, such as expanding the city's territory on the map. We'll also ensure that cities interact well with each other by implementing additional logic in our other functions like AddTerritory.

In the previous lesson, we set up the basic properties for our city class and initialized the underlying data for our population growth system. In this lesson, we will focus on expanding the city's territory both in data and on the map as a result of population growth. This involves creating a border tile pool to manage potential tiles for expansion.

Understanding the Border Tile Pool

The border tile pool is a crucial component for managing the expansion of a city's territory. It helps distinguish between tiles that the city currently owns and those that are potential candidates for expansion. For example, if a city owns a set of tiles, the border tile pool will keep track of the surrounding tiles that the city can expand into when its population grows.

Why Use a Border Tile Pool?

There are several reasons why using a border tile pool is beneficial:

- **Performance:** As cities grow larger, calculating potential expansion tiles dynamically every turn becomes computationally expensive. By caching these calculations, we improve performance.
- **Avoiding Conflicts:** Cities need to avoid expanding into each other's territory. A border tile pool helps manage this by keeping track of potential expansion areas for each city.

Modifying the Add Territory Function

To implement the border tile pool, we need to modify the AddTerritory function. This function will not only add the territory to the city but also populate the border tile pool with valid neighboring tiles.

```
public void AddTerritory(List<Hex> territoryToAdd)
{
    foreach (Hex h in territoryToAdd)
    {
        h.ownerCity = this;

        // Add new border hexes to the border tile pool
        AddValidNeighborsToBorderPool(h);
    }

    territory.AddRange(territoryToAdd);
    CalculateTerritoryResourceTotals();
}
```

Adding Valid Neighbors to the Border Pool

We need a function to add valid neighboring tiles to the border pool. This function will check the surrounding tiles of a given hex and add them to the border pool if they meet certain criteria.

```
public void AddValidNeighborsToBorderPool(Hex h)
{
    List<Hex> neighbors = map.GetSurroundingHexes(h.coordinates);

    foreach (Hex n in neighbors)
    {
```



```
        if (IsValidNeighborTile(n)) borderTilePool.Add(n);

        invalidTiles[n] = this;
    }
}
```

Determining Valid Neighbor Tiles

To determine if a tile is a valid neighbor, we need to check several conditions. These include the terrain type and whether the tile is already owned by another city.

```
public bool IsValidNeighborTile(Hex n)
{
    if (n.terrainType == TerrainType.WATER ||
        n.terrainType == TerrainType.ICE ||
        n.terrainType == TerrainType.SHALLOW_WATER ||
        n.terrainType == TerrainType.MOUNTAIN)
    {
        return false;
    }

    if (n.ownerCity != null && n.ownerCity.civ != null)
    {
        return false;
    }

    if (invalidTiles.ContainsKey(n) && invalidTiles[n] != this)
    {
        return false;
    }

    return true;
}
```

With these modifications, we have completed the first step towards implementing the population growth mechanic. In the next lesson, we will continue to build on this foundation.

In this lesson, we will continue designing and implementing the population growth system for our cities. We will focus on using the border tile pool system to randomly grow our territory whenever we want to increase the population. To achieve this, we will create a new function that takes a random tile from our border pool and adds it to the city's territory.

Adding a Random New Tile

To add a random new tile to our city's territory, we will create a function called AddRandomNewTile. This function will check if there are any tiles in the border tile pool. If there are, it will generate a random index to select a tile from the pool and add it to the city's territory. Here is the step-by-step process:

1. Check if the border tile pool is not empty.
2. Generate a random index using the count of elements in the border tile pool.
3. Use the generated index to access a random tile in the border tile pool.
4. Add the selected tile to the city's territory.
5. Remove the selected tile from the border tile pool.

Here is the code snippet for the AddRandomNewTile function:

```
public void AddRandomNewTile()
{
    if (borderTilePool.Count > 0)
    {
        Random r = new Random();
        int index = r.Next(borderTilePool.Count);
        this.AddTerritory(new List<Hex>{borderTilePool[index]});
        borderTilePool.RemoveAt(index);
    }
}
```

Updating the Territory Map

After adding a new tile to the city's territory, we need to update the territory map to reflect the changes. We will call the UpdateCivTerritoryMap function from the HexTileMap class, passing in the civilization. This function ensures that the new territory is reflected in the colored tiles on the map.

```
map.UpdateCivTerritoryMap(civ);
```

Cleaning Up the Border Pool

To ensure that the border tile pool does not contain any invalid tiles, we will create a helper function called CleanUpBorderPool. This function will iterate through the border tile pool and remove any tiles that are present in the invalid tiles dictionary and are not claimed by the current city.

Here is the code snippet for the CleanUpBorderPool function:

```
public void CleanUpBorderPool()
{
    List<Hex> toRemove = new List<Hex>();
    foreach (Hex b in borderTilePool)
    {
```



```
        if (invalidTiles.ContainsKey(b) && invalidTiles[b] != this)
        {
            toRemove.Add(b);
        }
    }

    foreach (Hex b in toRemove)
    {
        borderTilePool.Remove(b);
    }
}
```

Processing the Turn

Now that we have implemented the functions to add a random new tile and clean up the border pool, we can hook these up inside the ProcessTurn function in the City class. We will call the CleanUpBorderPool function at the beginning of the ProcessTurn function to ensure that the border tile pool is clean before processing the turn.

Here is the updated ProcessTurn function:

```
public void ProcessTurn()
{
    CleanUpBorderPool();

    populationGrowthTracker += totalFood;
    if (populationGrowthTracker > populationGrowthThreshold) // Grow population
    {
        population++;
        populationGrowthTracker = 0;
        populationGrowthThreshold += POPULATION_THRESHOLD_INCREASE;

        // Grow territory
        AddRandomNewTile();
        map.UpdateCivTerritoryMap(civ);
    }
}
```

Processing Civilization Turns

Next, we need to process the turns for each civilization. We will loop through every civilization in the list of saves in the HexTileMap class and call the ProcessTurn function for each civilization. If the ProcessTurn function is not yet implemented in the Civilization class, we will implement it later.

Here is the code snippet for processing civilization turns:

```
public void ProcessTurn()
{
    foreach (Civilization c in civs)
    {
        c.ProcessTurn();
    }
```



```
}
```

Refreshing the City UI

Finally, we need to ensure that the city UI reflects the changes in the city's population and resources. We will create a Refresh function in the CityUI class that updates the UI components with the current data from the city. We will also create a RefreshUI function in the UIManager class that calls the Refresh function on the city UI.

Here is the code snippet for the Refresh function in the CityUI class:

```
public void Refresh()
{
    cityName.Text = this.city.name;
    population.Text = "Population: " + this.city.population;
    food.Text = "Food: " + this.city.totalFood;
    production.Text = "Production: " + this.city.totalProduction;
}
```

And here is the code snippet for the RefreshUI function in the UIManager class:

```
public void RefreshUI()
{
    if (cityUi is not null)
        cityUi.Refresh();
}
```

With these changes, our cities will now grow their territory based on population growth, and the UI will reflect these changes accurately. Congratulations on making it this far in the course!

Congratulations on reaching the end of this course! We've covered a significant amount of material, so let's take a moment to review what you've learned.

Course Learning Goals

- Implementing map interaction
- Implementing a UI system with a UI manager class
- Adding resources to tiles
- Spawning cities and factions

Key Concepts and Achievements

1. **Map Interaction:** You've learned how to interact with map layers, send signals, and convert between local and global coordinate spaces.
2. **UI Manager System:** You've created a UI manager system to handle gameplay events and signals, displaying them on the UI.
3. **Godot Signals and C# Events:** You've explored the differences between built-in Godot signals and C# events and signals.
4. **Implementing Cities and Factions:** You've done extensive work on spawning cities, placing them on the map, assigning territory and colors, and setting up for future gameplay.

About Zenva

Zenva is an educational platform with over a million students, offering a wide range of courses for both beginners and those looking to learn something new. Our courses are versatile and allow you to learn in various ways, including video tutorials, lesson summaries, and following along with the provided infrastructure.

Thank you for joining us on this learning journey. Your instructor, Cameron, appreciates your dedication and effort.

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 folder: /

This script controls the movement and zooming of a camera in a 2D scene. It allows the camera to move horizontally and vertically within set boundaries and zoom in and out using keyboard or mouse wheel input. The camera's movement and zooming speed can be adjusted using exported variables.

```
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;

 }
}
```

City.cs

Found in folder: /

This script manages the behavior of a city in a game, including its population growth, territory expansion, and resource calculation. It also updates the city’s label and sprite accordingly. The city’s growth and expansion are based on its current population, food, and production resources.


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

public partial class City : Node2D
{

 public static Dictionary<Hex, City> invalidTiles = new Dictionary<Hex, City>();

 public HexTileMap map;
 public Vector2I centerCoordinates;

 public List<Hex> territory;
 public List<Hex> borderTilePool;

 public Civilization civ;


 // Gameplay constant
 public static int POPULATION_THRESHOLD_INCREASE = 15;

 // City name
 public string name;


 // Population
 public int population = 1;
 public int populationGrowthThreshold;
 public int populationGrowthTracker;


 // Resources
 public int totalFood;
 public int totalProduction;


 // Scene nodes
 Label label;
 Sprite2D sprite;

 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  label = GetNode<Label>("Label");
  sprite = GetNode<Sprite2D>("Sprite2D");

  label.Text = name;

  territory = new List<Hex>();
  borderTilePool = new List<Hex>();
 }

 public void ProcessTurn()
 {
  CleanUpBorderPool();
```



```
 populationGrowthTracker += totalFood;
 if (populationGrowthTracker > populationGrowthThreshold) // Grow population
 {
  population++;
  populationGrowthTracker = 0;
  populationGrowthThreshold += POPULATION_THRESHOLD_INCREASE;

  // Grow territory
  AddRandomNewTile();
  map.UpdateCivTerritoryMap(civ);
 }
}

public void CleanUpBorderPool()
{
 List<Hex> toRemove = new List<Hex>();
 foreach (Hex b in borderTilePool)
 {
  if (invalidTiles.ContainsKey(b) && invalidTiles[b] != this)
  {
   toRemove.Add(b);
  }
 }

 foreach (Hex b in toRemove)
 {
  borderTilePool.Remove(b);
 }
}

public void AddRandomNewTile()
{
 if (borderTilePool.Count > 0)
 {
  Random r = new Random();
  int index = r.Next(borderTilePool.Count);
  this.AddTerritory(new List<Hex>{borderTilePool[index]});
  borderTilePool.RemoveAt(index);
 }
}

public void AddTerritory(List<Hex> territoryToAdd)
{

 foreach (Hex h in territoryToAdd)
 {
  h.ownerCity = this;

  // Add new border hexes to the border tile pool
  AddValidNeighborsToBorderPool(h);
 }

 territory.AddRange(territoryToAdd);
 CalculateTerritoryResourceTotals();
```



```
}

public void AddValidNeighborsToBorderPool(Hex h)
{
 List<Hex> neighbors = map.GetSurroundingHexes(h.coordinates);

 foreach (Hex n in neighbors)
 {
  if (IsValidNeighborTile(n)) borderTilePool.Add(n);

  invalidTiles[n] = this;
 }
}

public bool IsValidNeighborTile(Hex n)
{
 if (n.terrainType == TerrainType.WATER ||
  n.terrainType == TerrainType.ICE ||
  n.terrainType == TerrainType.SHALLOW_WATER ||
  n.terrainType == TerrainType.MOUNTAIN)
 {
  return false;
 }

 if (n.ownerCity != null && n.ownerCity.civ != null)
 {
  return false;
 }

 if (invalidTiles.ContainsKey(n) && invalidTiles[n] != this)
 {
  return false;
 }

 return true;
}

public void CalculateTerritoryResourceTotals()
{
 totalFood = 0;
 totalProduction = 0;
 foreach (Hex h in territory)
 {
  totalFood += h.food;
  totalProduction += h.production;
 }
}

public void SetCityName(string newName)
{
 name = newName;
 label.Text = newName;
}
```



```
 public void SetIconColor(Color c)
 {
  sprite.Modulate = c;
 }
}
```

CityUI.cs

Found in folder: /

This script controls a user interface panel for a city, displaying its name, population, food, and production values. It retrieves references to the UI labels in the `_Ready` method and updates their text in the `Refresh` method. The `SetCityUI` method sets the city data and triggers a refresh of the UI labels.

```
using Godot;
using System;

public partial class CityUI : Panel
{

 Label cityName, population, food, production;

 // City data
 City city;

 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  cityName = GetNode<Label>("CityName");
  population = GetNode<Label>("Population");
  food = GetNode<Label>("Food");
  production = GetNode<Label>("Production");
 }

 public void SetCityUI(City city)
 {
  this.city = city;

  Refresh();
 }

 public void Refresh()
 {
  cityName.Text = this.city.name;
  population.Text = "Population: " + this.city.population;
  food.Text = "Food: " + this.city.totalFood;
  production.Text = "Production: " + this.city.totalProduction;
 }
}
```


Civilization.cs

Found in folder: /

This code defines a Civilization class that represents a civilization in the game. It has properties such as an ID, a list of cities, a territory color, a name, and methods to set a random territory color and process a turn. The ProcessTurn method iterates over the civilization's cities and calls each city's ProcessTurn method.

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

public class Civilization
{

 public int id;
 public List<City> cities;
 public Color territoryColor;
 public int territoryColorAltTileId;
 public string name;
 public bool playerCiv;

 public Civilization()
 {
  cities = new List<City>();
 }

 public void SetRandomColor()
 {
  Random r = new Random();
  territoryColor = new Color(r.Next(255)/255.0f, r.Next(255)/255.0f, r.Next(255)/255.
0f);
 }

 public void ProcessTurn()
 {
  foreach (City c in cities)
  {
   c.ProcessTurn();
  }
 }

}
```

Game.cs

Found in folder: /

This code initializes a Godot game scene by overriding the `_EnterTree` method. When the scene is loaded, it calls the `LoadTerrainImages` method of the `TerrainTileUI` class. The scene also has a FastNoiseLite object that can be exported and edited in the Godot editor.


```
using Godot;
using System;

public partial class Game : Node
{

 [Export]
 FastNoiseLite noise;

    public override void _EnterTree()
    {
        TerrainTileUI.LoadTerrainImages();
    }
}
```

GeneralUI.cs

Found in folder: /

This script manages a turn counter for a game. When the scene is loaded, it initializes a label to display the current turn number. The `IncrementTurnCounter` function updates the turn number and refreshes the label to show the new turn count.

```
using Godot;
using System;

public partial class GeneralUI : Panel
{

 int turns = 0;
 Label turnLabel;

 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  turnLabel = GetNode<Label>("TurnLabel");
  turnLabel.Text = "Turn: " + turns;
 }

 public void IncrementTurnCounter()
 {
  turns += 1;
  turnLabel.Text = "Turn: " + turns;
 }

}
```

HexTileMap.cs

Found in folder: /

This code defines a class `HexTileMap` that represents a map of hexagonal tiles, each with its own terrain type, resources, and ownership. The class generates a map with random terrain, resources, and city locations, and handles user input to select and interact with tiles. It also manages the game state, including civilizations, cities, and territory ownership.

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

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

public class Hex
{
 public readonly Vector2I coordinates;

 public TerrainType terrainType;

 public int food;
 public int production;

 public City ownerCity;
 public bool isCityCenter = false;

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

    public override string ToString()
    {
        return $"Coordinates: ({this.coordinates.X}, {this.coordinates.Y}. Terrain ty
pe: {this.terrainType}). Food: {this.food}. Production: {this.production}";
    }

}

public partial class HexTileMap : Node2D
{

 PackedScene cityScene;

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

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

 // Tile atlas
 TileSetAtlasSource terrainAtlas;
```



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

// UI
UIManager uimanager;


// GAMEPLAY DATA
public Dictionary<Vector2I, City> cities;
public List<Civilization> civs;

[Export]
public int NUM_AI_CIVS = 6;

[Export]
public Color PLAYER_COLOR = new Color(255, 255, 255);

// Signals
[Signal]
public delegate void ClickOffMapEventHandler();

public delegate void SendHexDataEventHandler(Hex h);
public event SendHexDataEventHandler SendHexData;

[Signal]
public delegate void SendCityUIInfoEventHandler(City c);


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

 cityScene = ResourceLoader.Load<PackedScene>("City.tscn");

 baseLayer = GetNode<TileMapLayer>("BaseLayer");
 borderLayer = GetNode<TileMapLayer>("HexBordersLayer");
 overlayLayer = GetNode<TileMapLayer>("SelectionOverlayLayer");
 civColorsLayer = GetNode<TileMapLayer>("CivColorsLayer");

 uimanager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");

 this.terrainAtlas = civColorsLayer.TileSet.GetSource(0) as TileSetAtlasSource;

 // 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)},
   { TerrainType.CIV_COLOR_BASE, new Vector2I(0, 3)},
  };

  GenerateTerrain();

  GenerateResources();


  // CIVILIZATION AND CITIES GEN
  civs = new List<Civilization>();
  cities = new Dictionary<Vector2I, City>();

  // Generate starting locations
  List<Vector2I> starts = GenerateCivStartingLocations(NUM_AI_CIVS + 1);

  // Generate player civilization
  Civilization playerCiv = CreatePlayerCiv(starts[0]);
  starts.RemoveAt(0);

  // Generate AI civilizations
  GenerateAICivs(starts);


  // UI signals
  this.SendHexData += uimanager.SetTerrainUI;
  uimanager.EndTurn += ProcessTurn;

 }

 Vector2I currentSelectedCell = new Vector2I(-1, -1);

    public override void _UnhandledInput(InputEvent @event)
    {
        if (@event is InputEventMouseButton mouse) {
   Vector2I mapCoords = baseLayer.LocalToMap(ToLocal(GetGlobalMousePosition()));

   if (mapCoords.X >= 0 && mapCoords.X < width && mapCoords.Y >= 0 && mapCoords.Y < h
eight)
   {
    Hex h = mapData[mapCoords];
    if (mouse.ButtonMask == MouseButtonMask.Left)
    {

     if (cities.ContainsKey(mapCoords))
     {
      EmitSignal(SignalName.SendCityUIInfo, cities[mapCoords]);
     } else {
      SendHexData?.Invoke(h);
     }

     if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSelectedCell,
-1);
     overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
     currentSelectedCell = mapCoords;
```



```
    }

   } else {
    overlayLayer.SetCell(currentSelectedCell, -1);
    EmitSignal(SignalName.ClickOffMap);
   }

  }
    }

 public void ProcessTurn()
 {
  foreach (Civilization c in civs)
  {
   c.ProcessTurn();
  }
 }


 public Civilization CreatePlayerCiv(Vector2I start)
 {
  Civilization playerCiv = new Civilization();
  playerCiv.id = 0;
  playerCiv.playerCiv = true;
  playerCiv.territoryColor = new Color(PLAYER_COLOR);

  int id = terrainAtlas.CreateAlternativeTile(terrainTextures[TerrainType.CIV_COLOR_B
ASE]);
  terrainAtlas.GetTileData(terrainTextures[TerrainType.CIV_COLOR_BASE], id).Modulate
= playerCiv.territoryColor;

  playerCiv.territoryColorAltTileId = id;

  civs.Add(playerCiv);

  CreateCity(playerCiv, start, "Player City");

  return playerCiv;
 }


 public void GenerateResources()
 {
  Random r = new Random();

  // populate tiles with food and production
  for (int x = 0; x < width; x++)
  {
   for (int y = 0; y < height; y++)
   {
    Hex h = mapData[new Vector2I(x, y)];

    switch (h.terrainType)
    {
```



```
     case TerrainType.PLAINS:
      h.food = r.Next(2, 6);
      h.production = r.Next(0, 3);
      break;
     case TerrainType.FOREST:
      h.food = r.Next(1, 4);
      h.production = r.Next(2, 6);
      break;
     case TerrainType.DESERT:
      h.food = r.Next(0, 2);
      h.production = r.Next(0, 2);
      break;
     case TerrainType.BEACH:
      h.food = r.Next(0, 4);
      h.production = r.Next(0, 2);
      break;
    }
   }
  }
 }

 public void GenerateAICivs(List<Vector2I> civStarts)
 {
  for (int i = 0; i < civStarts.Count; i++)
  {
   Civilization currentCiv = new Civilization
   {
    id = i + 1,
    playerCiv = false
   };

   // Assign a random color
   currentCiv.SetRandomColor();

   // Create alt tiles
   int id = terrainAtlas.CreateAlternativeTile(terrainTextures[TerrainType.CIV_COLOR_
BASE]);
   terrainAtlas.GetTileData(terrainTextures[TerrainType.CIV_COLOR_BASE], id).Modulate
 = currentCiv.territoryColor;

   currentCiv.territoryColorAltTileId = id;

   // Create starting city
   CreateCity(currentCiv, civStarts[i], "City " + civStarts[i].X);

   civs.Add(currentCiv);

  }
 }

 public List<Vector2I> GenerateCivStartingLocations(int numLocations)
 {
  List<Vector2I> locations = new List<Vector2I>();

  List<Vector2I> plainsTiles = new List<Vector2I>();
```



```
 for (int x = 0; x < width; x++)
 {
  for (int y = 0; y < height; y++)
  {
   if (mapData[new Vector2I(x, y)].terrainType == TerrainType.PLAINS)
   {
    plainsTiles.Add(new Vector2I(x, y));
   }
  }
 }

 Random r = new Random();
 for (int i = 0; i < numLocations; i++)
 {
  Vector2I coord = new Vector2I();

  bool valid = false;
  int counter = 0;

  while(!valid && counter < 10000)
  {
   coord = plainsTiles[r.Next(plainsTiles.Count)];
   valid = IsValidLocation(coord, locations);
   counter++;
  }

  plainsTiles.Remove(coord);
  foreach (Hex h in GetSurroundingHexes(coord))
  {
   foreach (Hex j in GetSurroundingHexes(h.coordinates))
   {
    foreach (Hex k in GetSurroundingHexes(j.coordinates))
    {
     plainsTiles.Remove(h.coordinates);
     plainsTiles.Remove(j.coordinates);
     plainsTiles.Remove(k.coordinates);
    }
   }
  }

  locations.Add(coord);

 }


 return locations;

}

private bool IsValidLocation(Vector2I coord, List<Vector2I> locations)
{
 if (coord.X < 3 || coord.X > width - 3 ||
  coord.Y < 3 || coord.Y > height - 3)
 {
```



```
  return false;
 }

 foreach (Vector2I l in locations)
 {
  if (Math.Abs(coord.X - l.X) < 20 || Math.Abs(coord.Y - l.Y) < 20)
   return false;
 }

 return true;
}


public void CreateCity(Civilization civ, Vector2I coords, string name)
{
 City city = cityScene.Instantiate() as City;
 city.map = this;
 civ.cities.Add(city);
 city.civ = civ;

 AddChild(city);

 // Set the color of the city's icon
 city.SetIconColor(civ.territoryColor);

 // Set the city's name
 city.SetCityName(name);

 // Set the coordinates of the city
 city.centerCoordinates = coords;
 city.Position = baseLayer.MapToLocal(coords);
 mapData[coords].isCityCenter = true;

 // Adding territory to the city
 city.AddTerritory(new List<Hex>{mapData[coords]});

 // Add the surrounding territory
 List<Hex> surrounding = GetSurroundingHexes(coords);

 foreach (Hex h in surrounding)
 {
  if (h.ownerCity == null)
   city.AddTerritory(new List<Hex>{h});
 }

 UpdateCivTerritoryMap(civ);

 cities[coords] = city;

}

public void UpdateCivTerritoryMap(Civilization civ)
{
 foreach (City c in civ.cities)
 {
```



```
   foreach (Hex h in c.territory)
   {
    civColorsLayer.SetCell(h.coordinates, 0, terrainTextures[TerrainType.CIV_COLOR_BA
SE], civ.territoryColorAltTileId);
   }
  }
 }


 public List<Hex> GetSurroundingHexes(Vector2I coords)
 {
  List<Hex> result = new List<Hex>();

  foreach (Vector2I coord in baseLayer.GetSurroundingCells(coords))
  {
   if (HexInBounds(coord))
    result.Add(mapData[coord]);
  }

  return result;

 }

 public bool HexInBounds(Vector2I coords)
 {
  if (coords.X < 0 || coords.X >= width ||
   coords.Y < 0 || coords.Y >= height)
   return false;

  return true;
 }


    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 M
in, 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 + 0.05f
);
  // Desert gen values
  Vector2 desertGenValues = new Vector2(desertNoiseMax/10 * 6, desertNoiseMax + 0.05f
);
  // Mountain gen values
  Vector2 mountainGenValues = new Vector2(mountainNoiseMax/10 * 5.5f, mountainNoiseMa
x + 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);
 }

}
```

TerrainTileUI.cs

Found in folder: /

This script manages the UI for a terrain tile, displaying its type, food, and production values. It loads

images for each terrain type and stores them in a dictionary for easy access. The `SetHex` method updates the UI components with data from a given `Hex` object.

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

public partial class TerrainTileUI : Panel
{

 public static Dictionary<TerrainType, string> terrainTypeStrings = new Dictionary<Te
rrainType, string>
 {
  { TerrainType.PLAINS, "Plains" },
  { TerrainType.BEACH, "Beach"},
  { TerrainType.DESERT, "Desert"},
  { TerrainType.MOUNTAIN, "Mountain"},
  { TerrainType.ICE, "Ice"},
  { TerrainType.WATER, "Water"},
  { TerrainType.SHALLOW_WATER, "Shallow Water"},
  { TerrainType.FOREST, "Forest"},
 };

 public static Dictionary<TerrainType, Texture2D> terrainTypeImages = new();

 public static void LoadTerrainImages()
 {
  Texture2D plains = ResourceLoader.Load("res://textures/plains.jpg") as Texture2D;
  Texture2D beach = ResourceLoader.Load("res://textures/beach.jpg") as Texture2D;
  Texture2D desert = ResourceLoader.Load("res://textures/desert.jpg") as Texture2D;
  Texture2D mountain = ResourceLoader.Load("res://textures/mountain.jpg") as Texture2
D;
  Texture2D ice = ResourceLoader.Load("res://textures/ice.jpg") as Texture2D;
  Texture2D ocean = ResourceLoader.Load("res://textures/ocean.jpg") as Texture2D;
  Texture2D shallow = ResourceLoader.Load("res://textures/shallow.jpg") as Texture2D;
  Texture2D forest = ResourceLoader.Load("res://textures/forest.jpg") as Texture2D;

  terrainTypeImages = new Dictionary<TerrainType, Texture2D>
  {
   { TerrainType.PLAINS, plains},
   { TerrainType.BEACH, beach},
   { TerrainType.DESERT, desert},
   { TerrainType.MOUNTAIN, mountain},
   { TerrainType.ICE, ice},
   { TerrainType.WATER, ocean},
   { TerrainType.SHALLOW_WATER, shallow},
   { TerrainType.FOREST, forest}
  };

 }


 // Data hex
 Hex h = null;
```



```
 // UI Components
 TextureRect terrainImage;
 Label terrainLabel, foodLabel, productionLabel;


 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  terrainLabel = GetNode<Label>("TerrainLabel");
  foodLabel = GetNode<Label>("FoodLabel");
  productionLabel = GetNode<Label>("ProductionLabel");
  terrainImage = GetNode<TextureRect>("TerrainImage");
 }

 public void SetHex(Hex h)
 {
  this.h = h;

  terrainImage.Texture = terrainTypeImages[h.terrainType];
  foodLabel.Text = $"Food: {h.food}";
  productionLabel.Text = $"Production: {h.production}";
  terrainLabel.Text = $"Terrain: {terrainTypeStrings[h.terrainType]}";
 }
}
```

UIManager.cs

Found in folder: /

This script manages the user interface for a game, specifically handling the display and interaction of terrain and city UI elements. It loads UI scenes, sets up event handlers, and provides methods to show, hide, and refresh UI elements. It also emits a signal when the end turn button is pressed.

```
using Godot;
using System;

public partial class UIManager : Node2D
{

 PackedScene terrainUiScene;
 PackedScene cityUiScene;

 TerrainTileUI terrainUi = null;
 CityUI cityUi = null;
 GeneralUI generalUi;

 [Signal]
 public delegate void EndTurnEventHandler();


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



```
{
 terrainUiScene = ResourceLoader.Load<PackedScene>("TerrainTileUI.tscn");
 cityUiScene = ResourceLoader.Load<PackedScene>("CityUI.tscn");

 generalUi = GetNode<Panel>("GeneralUi") as GeneralUI;

 // End turn button
 Button endTurnButton = generalUi.GetNode<Button>("EndTurnButton");
 endTurnButton.Pressed += SignalEndTurn;
}

public void SignalEndTurn()
{
 EmitSignal(SignalName.EndTurn);
 generalUi.IncrementTurnCounter();
 RefreshUI();
}


public void HideAllPopups()
{
 if (terrainUi is not null)
 {
  terrainUi.QueueFree();
  terrainUi = null;
 }

 if (cityUi is not null)
 {
  cityUi.QueueFree();
  cityUi = null;
 }

}

public void RefreshUI()
{
 if (cityUi is not null)
  cityUi.Refresh();
}

public void SetCityUI(City c)
{
 HideAllPopups();

 cityUi = cityUiScene.Instantiate() as CityUI;
 AddChild(cityUi);

 cityUi.SetCityUI(c);
}

public void SetTerrainUI(Hex h)
{
 // if (terrainUi is not null) terrainUi.QueueFree();
 HideAllPopups();
```



```
  terrainUi = terrainUiScene.Instantiate() as TerrainTileUI;
  AddChild(terrainUi);

  terrainUi.SetHex(h);


 }
}
```

