Welcome to this course where we will be creating a 4x strategy game based on a map of hexagonal tiles using Godot and C#. Your instructor for this course is Cameron. In this installment, we will continue building our turn-based strategy game, focusing on creating a system of units, their mechanics, and implementing a main menu.

Course Overview

In this section, we will cover the following topics:

- Creating a system of units that can be spawned from cities and moved around the map.
- Implementing unit mechanics such as movement, blocking, combat, founding new cities, and capturing other cities.
- Designing a main menu to customize the game experience at the start.

Prerequisites

Before starting this course, you should have:

- Basic skills with the Godot Engine.
- Basic knowledge of C#.
- Completed the previous course in this series.

Specific Topics Covered

Here are some of the specific topics we will cover in detail:

1. Spawning units in cities.
2. Creating a UI to display unit information.
3. Moving units on the map.
4. Implementing specific unit mechanics such as combat between units, using units to found new cities, and using units to capture other cities.
5. Creating more map overlays.
6. Setting up the game with the help of the main menu.

About Zenva

Zenva is an online learning academy with over a million students. We offer a wide range of courses for both beginners and those looking to learn something new. Our courses are versatile, allowing you to learn in a way that suits you best.

Why Learn This?

The concepts and mechanics we will cover in this course are fundamental to strategy games and will provide a strong foundation for more complex projects in the future.

We look forward to seeing the amazing games you will create with the knowledge gained from this course!

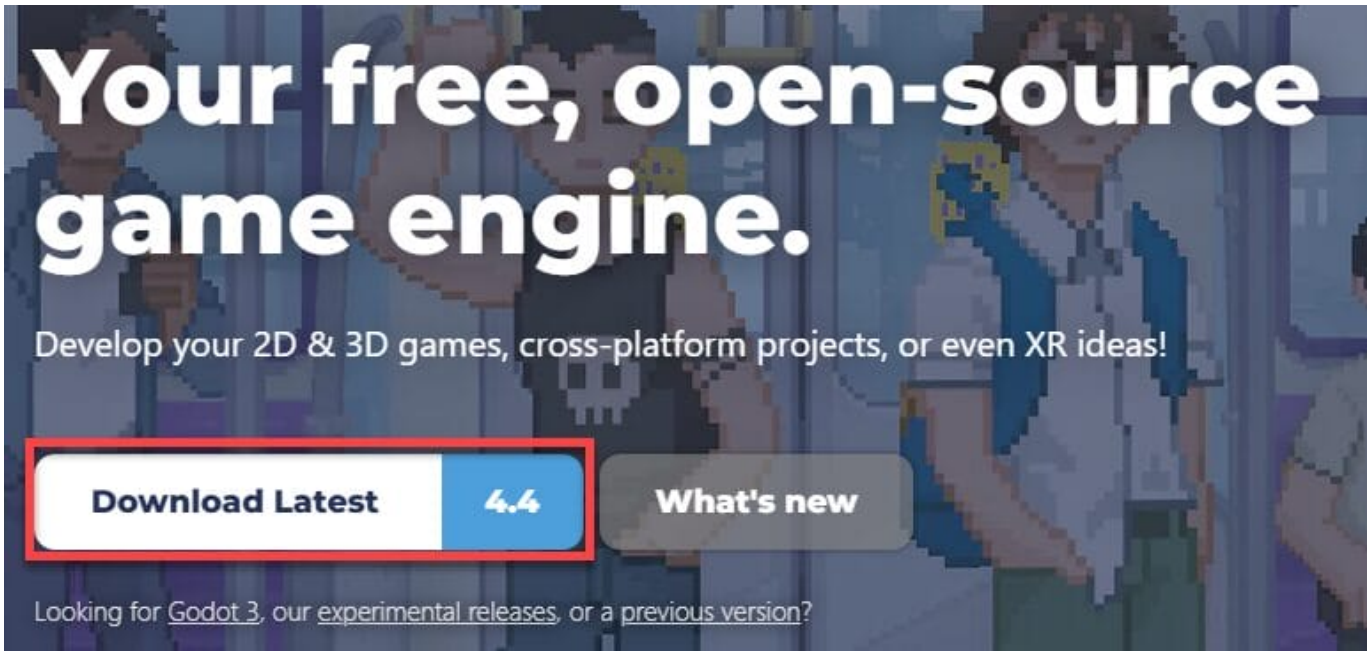
© 2024 Zenva Pty Ltd. All rights reserved.

Course Updated to Godot 4.4

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

How to Install Version 4.4 .NET

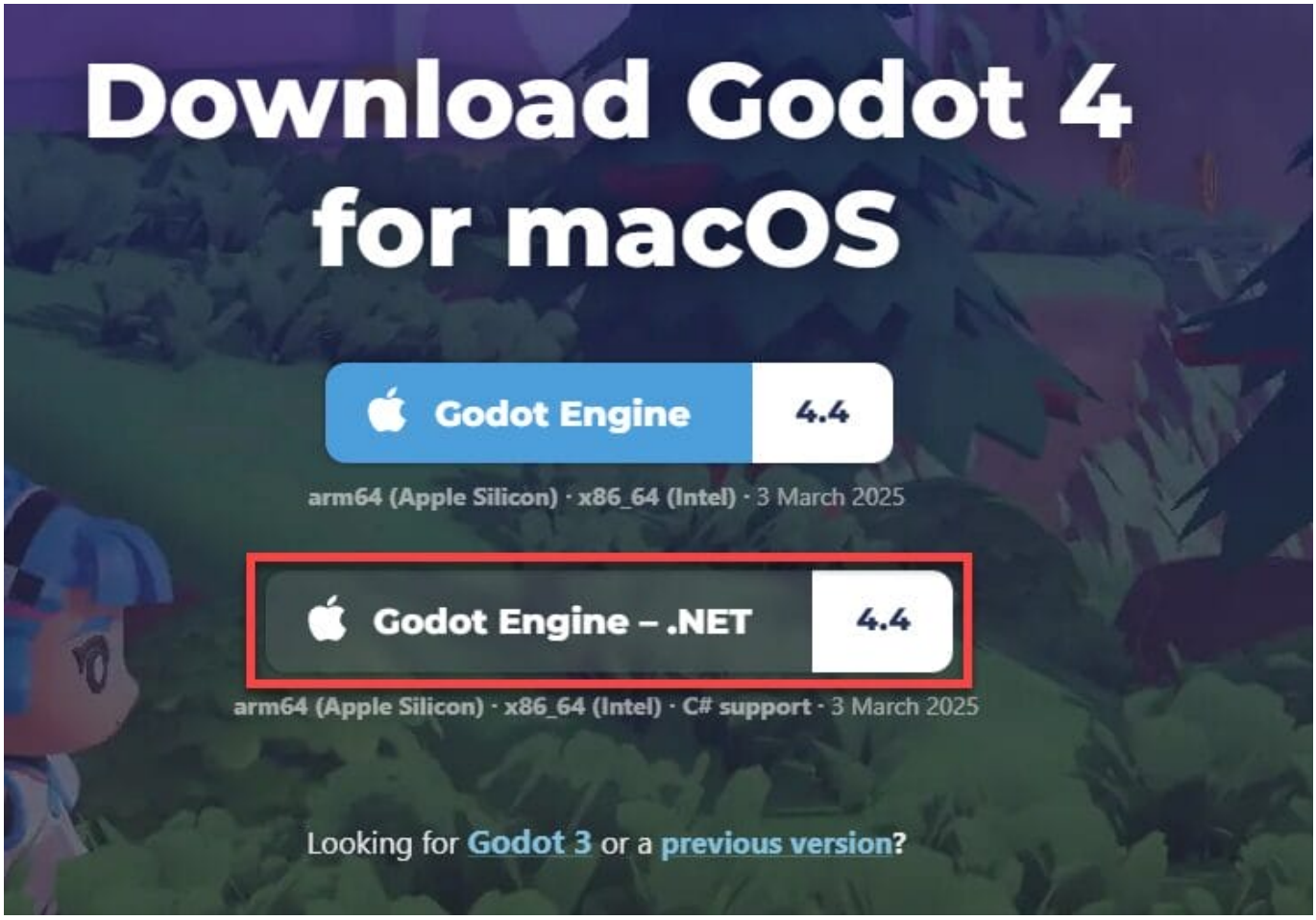
You can download the most recent version of Godot by heading to the Godot website (https://godotengine.org/) and clicking the "Download Latest" button on the front page. This will automatically take you to the download page for your operating system.



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

© 2025 Zenva Pty Ltd. All rights reserved.

© 2025 Zenva Pty Ltd. All rights reserved.

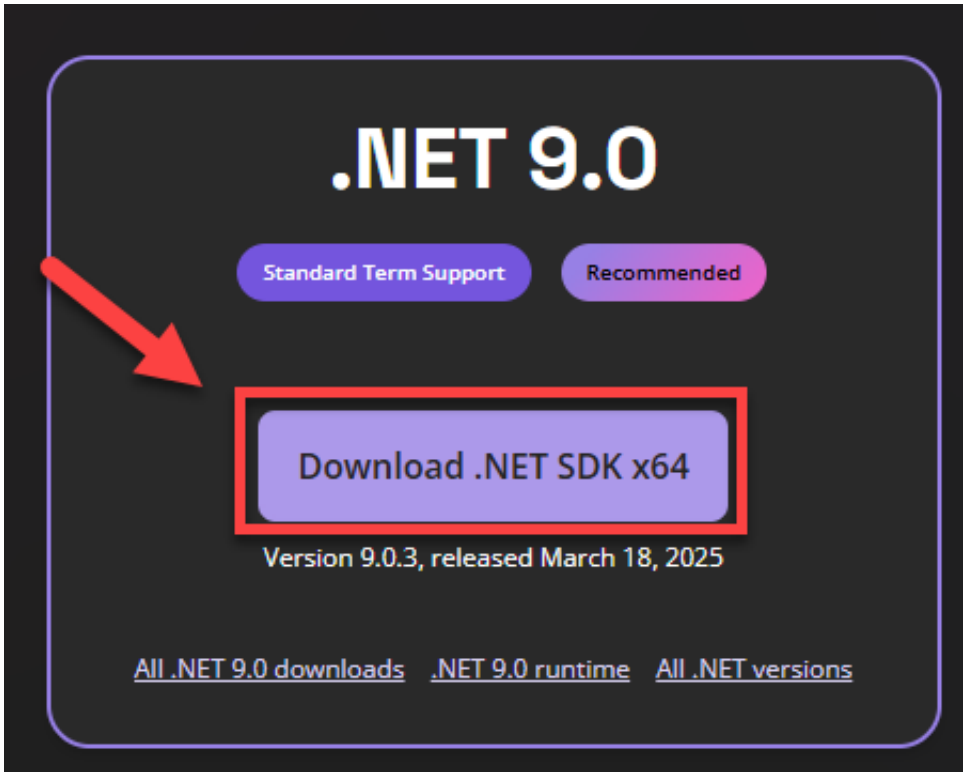


If Godot fails to automatically select the correct operating system for you, you can scroll down to the "Supported platforms" section on the download page to manually select the download version you would like to install:

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.

© 2025 Zenva Pty Ltd. All rights reserved.

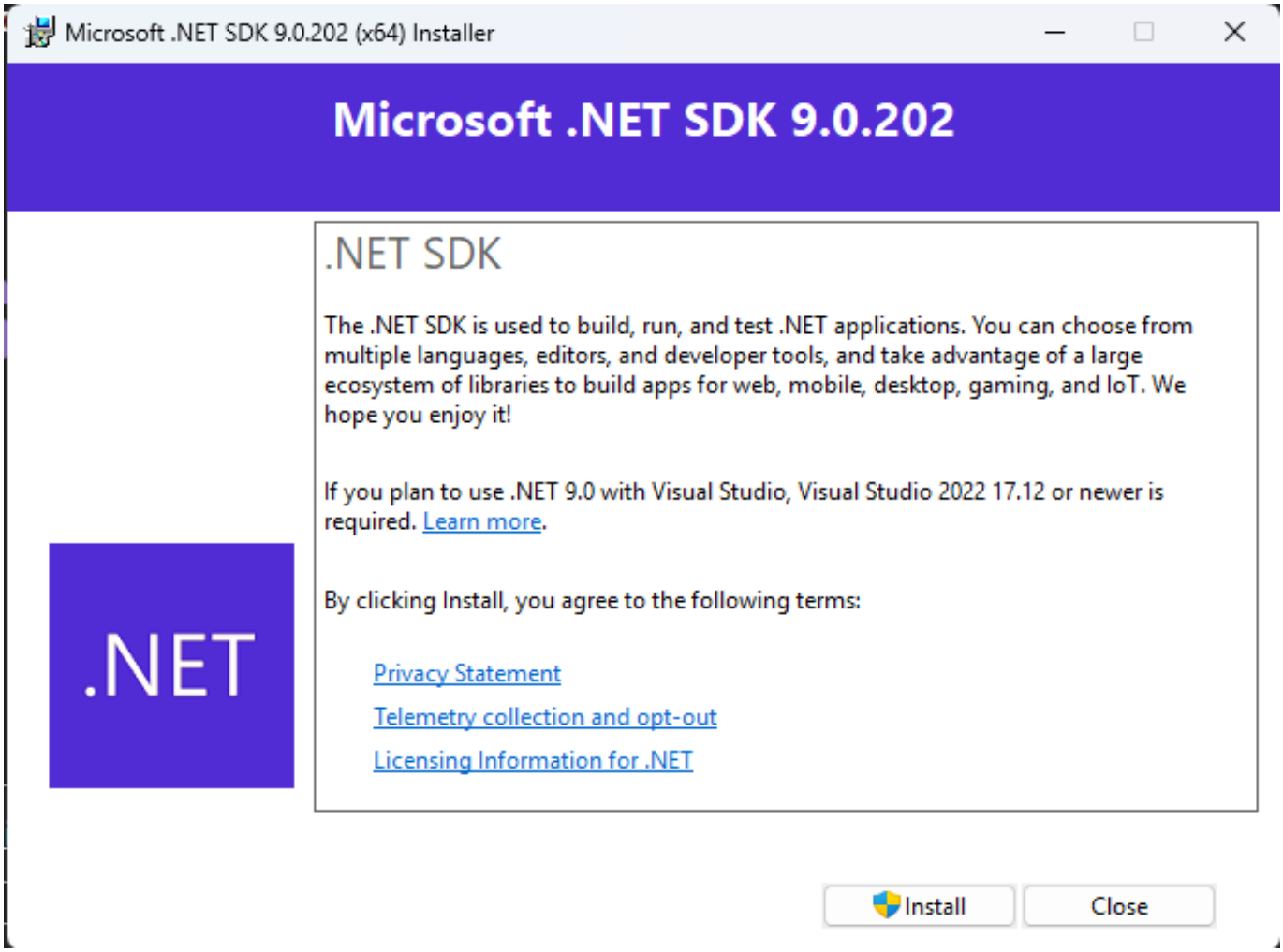


If the site fails to select the correct OS for you, you can also click the “All .NET downloads” link to access all available downloads for the SDK.

© 2025 Zenva Pty Ltd. All rights reserved.

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

© 2025 Zenva Pty Ltd. All rights reserved.



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

```
PS C:\Users\      > dotnet

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

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

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

© 2025 Zenva Pty Ltd. All rights reserved.

Welcome back to our course on building a 4X turn-based strategy game using Godot and C#. In this lesson, we will focus on adding mechanics for units and founding cities. By the end of this lesson, you will have a basic foundation for units in your game, including the ability to spawn and interact with them on the map.

Recap of Previous Lesson

Before we dive into the new material, let's quickly recap where we left off last time. In the previous lesson, we successfully implemented the spawning of cities and civilizations on a randomly generated map. Each city had its own unique color and UI, and we even added a population mechanic that allowed cities to grow based on available resources.

Overview of This Lesson

In this lesson, we will:

- Create a base scene for units.
- Add inherited scenes for specific unit types (Warrior and Settler).
- Create scripts for the base unit class and the specific unit types.

Creating the Base Unit Scene

Let's start by creating a new scene for our units. This scene will serve as the base class for all units in the game.

1. In the file system, create a new scene named Unit. This will be a 2D scene with Node2D as the root.
2. Add a Sprite2D node as a child of the Unit node. This will hold the graphics for the unit.
3. Add a placeholder texture to the Sprite2D node. For example, you can use a settler icon from your textures folder.
4. Add an Area2D node as a child of the Sprite2D node. This will allow us to make the unit interactive via mouse input.
5. Add a CollisionShape2D node as a child of the Area2D node. Set the shape to a circle to match the unit's size.

Creating Inherited Scenes for Specific Unit Types

Next, we will create inherited scenes for the two types of units we will have in the game: Warrior and Settler.

1. Right-click on the Unit scene and select "New Inherited Scene". Name this new scene Warrior.
2. Adjust the sprite to reflect the Warrior unit type by selecting the appropriate texture.
3. Repeat the process to create a Settler scene. This scene already has the correct graphic, so no changes are needed.

Creating Scripts for the Units

Now, let's create scripts for our units. The base unit class will contain most of the logic, while the specific unit types will inherit from this base class.

1. Create a new script named Unit.cs for the base unit class. This script will be associated with the Unit scene.
2. Create inherited scripts for the Warrior and Settler scenes by extending the Unit.cs script.

© 2024 Zenva Pty Ltd. All rights reserved.

Conclusion

In this lesson, we laid the foundation for our unit mechanics by creating a base unit scene and scripts for different unit types. In the next lesson, we will extend our city graphics and logic to allow for spawning units on the map via the city UI. Stay tuned for that!

© 2024 Zenva Pty Ltd. All rights reserved.

In the previous lesson, we laid the foundations for our unit system by creating base classes for our units. In this lesson, we will extend our city UI to reflect what units are available for building and what units are currently queued for production. This will involve adding new labels and buttons to our city UI to facilitate unit production.

Extending the City UI

To start, we will add new labels to subdivide the city UI into sections for building and queuing units.

Adding Labels and Containers

First, we will add a child label for the build section and duplicate it for the queue section. We will then rename these labels appropriately.

- Add a child label for the build section.
- Duplicate the label for the queue section.
- Rename the labels to “Build” and “Queue” respectively.

Adding Scroll Containers and VBox Containers

Next, we will add scroll containers and VBox containers to hold our buttons and queue information.

- Add a scroll container under the build label.
- Add a VBox container inside the scroll container to hold the build buttons.
- Duplicate the scroll container and VBox container for the queue section.

Creating the Unit Build Button

To associate a particular button with a particular type of unit, we will create a new scene that extends from the Godot base button class. This new scene will be called UnitBuildButton.

Creating the UnitBuildButton Scene

We will create a new scene that inherits from the button class and attach a script to it. This script will define a signal that can pass unit data and a method to emit this signal when the button is pressed.

```
using Godot;
using System;

public partial class UnitBuildButton : Button
{
    [Signal]
    public delegate void OnPressedEventHandler(Unit u);

    public Unit u;

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

    public void SendUnitData()
    {
        EmitSignal(SignalName.OnPressed, u);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    }
}
```

Adding Unit Build Buttons to the City UI

Finally, we will add instances of the UnitBuildButton to our city UI. For simplicity, we will hard code two buttons, one for each type of unit (Settler and Warrior).

- Instantiate a child scene of UnitBuildButton inside the VBox container for the build buttons.
- Set the text of the first button to “Settler”.
- Instantiate another UnitBuildButton and set its text to “Warrior”.

Testing the City UI

To test our new city UI, we can open the game and check if the buttons are displayed correctly. The buttons should not do anything yet, as we will cover their functionality in the next few lessons.

By following these steps, we have successfully extended our city UI to include sections for building and queuing units. In the next lessons, we will make these buttons functional and implement unit production.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back! In the previous lesson, we created a nifty UI extension for our city UI, including a place for buttons to build units and a place to display a unit queue. Now, we need to implement the backend functionality to make this system work. In this lesson, we will add properties to our City class to keep track of units in the queue, populate the city UI with data from that queue, and connect signals from our buttons to our city.

Adding Properties to the City Class

First, let's head over to the City class in VS Code. We need to add a few more properties to keep track of all the data we need that are relevant to units with regards to the city.

We'll create a new section in the City properties for units. We need three things:

- A data structure that will serve as the unit build queue.
- A pointer to the current unit that's being built.
- An integer to track the progress of the current unit's build.

Let's start by creating the unit build queue. We'll create a new list of Unit objects and call it the unit build queue. Since we're creating a list here, we'll need to head down to the ready function to instantiate this alongside our other lists.

```
public List<Unit> unitBuildQueue;
```

In the _Ready function, we'll instantiate this list:

```
unitBuildQueue = new List<Unit>();
```

Next, we'll add a reference to a unit that we'll call currentUnitBeingBuilt:

```
public Unit currentUnitBeingBuilt;
```

Finally, we'll create an integer called unitBuildTracker that we will set to zero to begin with:

```
public int unitBuildTracker = 0;
```

Adding a Function to Add Units to the Build Queue

Now, we want some way to actually add a unit to the unit build queue. We'll create a new function called AddUnitToBuildQueue that takes in a unit instance. For now, all we'll do is add this unit to the build queue:

```
public void AddUnitToBuildQueue(Unit u)
{
    unitBuildQueue.Add(u);
}
```

© 2024 Zenva Pty Ltd. All rights reserved.

Populating the Unit Queue UI

Next, we'll switch gears and head over to the CityUI class. We'll add a function to display the contents of the unit queue. We'll call this function PopulateUnitQueueUI. It will take in a reference to a city.

In this function, we need to:

1. Get the actual UI node that we're going to be populating with data. This is a VBox container that we'll call queue, and we'll get that node of type VBoxContainer.
2. Make sure that there is no existing data in this queue. For every node that is currently in the queue, we'll get those children and remove them and free them.
3. Populate the queue with dynamically generated labels representing the units in the queue. We'll start the basic setup for this, but will need to add some properties to the Unit class first in order to set things up.

Here's the code for the PopulateUnitQueueUI function:

```
public void PopulateUnitQueueUI(City city)
{
    VBoxContainer queue = GetNode<VBoxContainer>("QueueContainer/VBoxContainer");

    foreach (Node n in queue.GetChildren())
    {
        queue.RemoveChild(n);
        n.QueueFree();
    }

    for (int i = 0; i < city.unitBuildQueue.Count; i++)
    {
        Unit u = city.unitBuildQueue[i];

        if (i == 0) // Unit is currently being built
        {
            queue.AddChild(new Label()
            {

            });
        }
    }
}
```

Adding Properties to the Unit Class

To have data to display in the unit queue UI, we need to add some properties to the Unit class that we haven't really touched yet. We need at least two properties:

- A string called unitName that defaults to "default".
- An integer called productionRequired.

Let's add these properties to the Unit class:

```
public string unitName = "DEFAULT";
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
public int productionRequired;
```

Now, we'll set these properties in the constructors of the specific unit classes, such as Warrior and Settler:

For the Warrior class:

```
public Warrior()
{
    unitName = "Warrior";
    productionRequired = 50;
}
```

For the Settler class:

```
public Settler()
{
    unitName = "Settler";
    productionRequired = 100;
}
```

In the next video, we will finish up this system, returning to the CityUI and actually get this all connected together.

© 2024 Zenva Pty Ltd. All rights reserved.

In the previous lesson, we worked on the CityUI class and created basic properties for the Unit class. We are now ready to create labels for units that are currently being built and those that are not yet in production. Additionally, we will connect the signals from the unit build buttons to the city to allow the city to receive the type of unit from these buttons.

Creating Labels for Units in Production

First, let's create a new label for unit that is currently being built. We will set the text of this label to display the unit's name, the amount of production that has gone into the unit so far, and the total production required. This will give players a clear view of the progress being made on each unit.

```
for (int i = 0; i < city.unitBuildQueue.Count; i++)
    {
      Unit u = city.unitBuildQueue[i];
      if (i == 0) // Unit is currently being built
      {
        queue.AddChild(new Label() {
          Text = $"{u.unitName} {city.unitBuildTracker}/{u.productionRequired}"
        });
      }
    }
```

Creating Labels for Units Not in Production

Next, we need to create labels for units that are not yet in production. For these units, we will hard code a zero for the production progress and display the total production required.

```
for (int i = 0; i < city.unitBuildQueue.Count; i++)
    {
      Unit u = city.unitBuildQueue[i];
      if (i == 0) // Unit is currently being built
      {
        queue.AddChild(new Label() {
          Text = $"{u.unitName} {city.unitBuildTracker}/{u.productionRequired}"
        });
      } else {
        queue.AddChild(new Label() {
          Text = $"{u.unitName} 0/{u.productionRequired}"
        });
      }
    }
```

Connecting Unit Build Signals

Now, we need to connect the signals from the unit build buttons to the city. This will allow the city to receive the type of unit from these buttons and add it to the build queue. We will create a function called ConnectUnitBuildSignals that takes in a particular city as a parameter.

From there, we will get a reference to our buttons and set which unit belongs to which button. In a more complex game, you'd want a more dynamic system to set this up. For our purposes, however, we will only have two units so can code this in manually.

© 2024 Zenva Pty Ltd. All rights reserved.

After this, we need to subscribe our functions to our pressed event.

```
public void ConnectUnitBuildSignals(City city)
{
    VBoxContainer buttons = GetNode("UnitBuildButtons/VBoxContainer");

    UnitBuildButton settlerButton = buttons.GetNode("SettlerButton");
    settlerButton.u = new Settler();

    UnitBuildButton warriorButton = buttons.GetNode("WarriorButton");
    warriorButton.u = new Warrior();

    settlerButton.OnPressed += city.AddUnitToBuildQueue;
    settlerButton.OnPressed += this.Refresh;
    warriorButton.OnPressed += city.AddUnitToBuildQueue;
    warriorButton.OnPressed += this.Refresh;
}
```

Note that we're also calling a Refresh function. Our current refresh function updates the CityUI data. However, it doesn't take any parameters – so we can't call it directly since we are sending unit data with our buttons. To fix this, we will create an overload function to call that will take the Unit as a parameter. It's only job will be to call the parameterless Refresh function. In this way, we avoid conflicts.

```
  public void Refresh(Unit u)
  {
    Refresh();
  }
}
```

Renaming Buttons in the Godot Editor

Before we can test our work, we need to rename the buttons in the Godot editor to match the names we used in the code. This will ensure that the signals are connected correctly.

- Rename the settler button to SettlerButton.
- Rename the warrior button to WarriorButton.

There is also one other thing we need to do, and that's call our signal connection function when the SetCityUI function is called:

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

Testing the Implementation

© 2024 Zenva Pty Ltd. All rights reserved.

Finally, we can test our implementation by heading over to a city and trying to add a unit to the build queue. We should see the queue populated with the units we add. Eventually, as we add more units, a scroll bar will be created due to the scroll box.

In the next few lessons, we will create mechanics for cities to go through the process of production on these units and eventually spawn them on the map. Stay tuned for that exciting content!

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back! In the previous lesson, we successfully created a unit queue in our city UI. This allows us to add units to the queue and display the build progress on the UI. However, as it stands, this is just a UI feature, and the cities do not actually build the units yet.

In this lesson, we're going to start programming the ability to produce the units and spawn them on the map.

Creating Helper Functions

To start, we'll work backwards by first creating a modular function that will actually spawn the unit on the map.

In the City class, we'll create a new function called SpawnUnit. This function will take in an instance of the Unit class.

```
public void SpawnUnit(Unit u)
{

}
```

Mapping Unit Types to Scenes

However, we immediately face a conundrum. In Godot, we need a reference to the particular scene or the packed scene class for whatever scene we want to spawn. Here, we don't have such a reference when we just have a unit reference. To solve this, we'll head back over to the Unit class and create a global static dictionary that will map unit types to their particular packed scene that we can use to instantiate them.

In the Unit class, we'll create a new property called unitSceneResources. This is going to be a static property, shared between all members of the Unit class, and will be a dictionary of types to packed scenes.

```
public static Dictionary<Type, PackedScene> unitSceneResources;
```

We'll also create a static function called LoadUnitScenes. This function will load in all of the packed scenes for all the different unit types so we have access to them whenever we want to spawn a new unit.

```
public static void LoadUnitScenes()
{
    unitSceneResources = new Dictionary<Type, PackedScene>
    {
        { typeof(Settler), ResourceLoader.Load<PackedScene>("res://Settler.tscn") },
        { typeof(Warrior), ResourceLoader.Load<PackedScene>("res://Warrior.tscn") }
    };
}
```

Loading Unit Scenes in the Game Class

Next, we're going to need to go to the Game class and load our Dictionary in so it's ready to use.

© 2024 Zenva Pty Ltd. All rights reserved.

Inside of our _EnterTree function in the Game class, we'll call Unit.LoadUnitScenes. This will make sure these will be ready to go when the game starts.

```
public override void _EnterTree()
{
    Unit.LoadUnitScenes();

    TerrainTileUI.LoadTerrainImages();
}
```

Setting the Owner and Color of the Unit

Returning to the City class, we can finally spawn a unit. By using our Dictionary, we can now find the matching type of unit to properly instantiate it. We'll set the position of the unit on the map as we spawn it. Ideally, we want this to spawn at the city center.

```
public void SpawnUnit(Unit u)
{
  Unit unitToSpawn = (Unit) Unit.unitSceneResources[u.GetType()].Instantiate();
  unitToSpawn.Position = map.MapToLocal(this.centerCoordinates);
}
```

For us to spawn at the city center, though, we need to know which city built the unit. As such, we need to add some properties and methods to the Unit class to set the owner of the unit to a particular civilization. This will also let us set the color of the unit to the correct color for the civilization that spawned it.

In the Unit class, we'll create a function called SetCiv that takes in a reference to a particular civilization. We also want to make sure that the unit has a property that lets us know what the owner civilization of this unit is.

```
public Civilization civ;

public void SetCiv(Civilization civ)
{
    this.civ = civ;
    GetNode<Sprite2D>("Sprite2D").Modulate = civ.territoryColor;
}
```

Now, we want to make sure that civilizations have some idea of what units are in their possession. So, we'll head over to the Civilization class and make sure that civilizations can keep track of their own list of units when we spawn them.

In the Civilization class, we'll create a new list of units. In the constructor, we will construct this new list of units.

```
public List<Unit> units;

public Civilization()
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
{
    cities = new List<City>();
    units = new List<Unit>();
}
```

Back in our Unit class, we can add this unit to that civilization's list of units when we call our **SetCiv function**.

```
this.civ.units.Add(this);
```

Now we've registered this unit with a particular faction, set the territory color, and have everything hooked up. Back in the City class, we can call SetCiv on this city's civilization in our spawn function.

```
unitToSpawn.SetCiv(this.civ);
```

That registers the unit with a particular faction. In the next video, we will finish this whole system up.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will continue the process of spawning units from a city's unit build queue. Last time, we started creating a function to spawn a unit from a city. We have already registered the unit with a particular civilization. Now, we need to ensure that a unit knows its hex coordinates when it spawns. We will also implement a function to process the unit build queue every turn.

Adding Location Awareness to Units

To make sure that a unit knows where it is spawning, we need to add location awareness to our unit class. We will start by adding a vector of integer coordinates to the unit class.

```
public Vector2I coords = new Vector2I();
```

Next, we will set the unit's coordinates to the coordinates of the city center when the unit is spawned.

```
public void SpawnUnit(Unit u)
{
    Unit unitToSpawn = (Unit) Unit.unitSceneResources[u.GetType()].Instantiate();
    unitToSpawn.Position = map.MapToLocal(this.centerCoordinates);
    unitToSpawn.SetCiv(this.civ);
    unitToSpawn.coords = this.centerCoordinates;

    map.AddChild(unitToSpawn);
}
```

Processing the Unit Build Queue

Now that we have added location awareness to our units, we need to implement a function to process the unit build queue every turn. This function will check if there are any units in the build queue and take appropriate actions based on some conditions.

```
public void ProcessUnitBuildQueue()
{
    if (unitBuildQueue.Count > 0)
    {
        if (currentUnitBeingBuilt == null)
        {
            currentUnitBeingBuilt = unitBuildQueue[0];
        }

        unitBuildTracker += totalProduction;

        if (unitBuildTracker >= currentUnitBeingBuilt.productionRequired)
        {
            SpawnUnit(currentUnitBeingBuilt);

            unitBuildQueue.RemoveAt(0);
            currentUnitBeingBuilt = null;

            unitBuildTracker = 0;
        }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    }
}
```

This function first checks if there are any units in the build queue. If there are, it checks if the current unit being built is null. If it is, it sets the current unit being built to the first unit in the build queue. Then, it adds the total production of the city to the unit build tracker.

After this, it checks if the unit build tracker is greater than or equal to the production required for the current unit being built. If it is, it spawns the unit, removes it from the build queue, sets the current unit being built to null, and resets the unit build tracker to zero.

Integrating the ProcessUnitBuildQueue Function

Finally, we need to make sure that every single turn that the city processes, we do some work on the build queue with the ProcessUnitBuildQueue function. We will add this function to the ProcessTurn method in the City class.

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

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

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

    ProcessUnitBuildQueue();
}
```

It is important to note that the ProcessUnitBuildQueue function should not be inside the if block that processes the population growth tracker. This ensures that the processing of units is not connected to whether the population growth tracker is above the growth threshold.

Testing the Implementation

Now we are ready to test our implementation in the game. We can head over to a city and build a warrior. We should see that every turn, the unit production goes up a little bit until finally, the unit is done and spawns on the screen. We can also test spawning a settler to ensure everything is working correctly.

In the next few videos, we will start figuring out how we can click on the units, get some information about them, and eventually start moving and interacting with them.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back! In the previous lesson, we completed our system for spawning units from a city. We now have a system of buttons and a queue in our city UI, and our cities apply their resources to building a unit each turn. At the end of this process, we end up with a new unit on our city.

However, we currently have no way to interact with these units. In the next few lessons, we will create such a system, starting with creating a UI for units and laying some foundations for having the units interactable by mouse click.

Creating the Unit UI

To begin, we need to create a new scene for our unit UI. This UI will inherit from a panel since that's the base of our other UI systems. Let's head to our Godot editor and create a new scene.

1. Create a new scene and name it UnitUI.
2. Set the root node of the scene to a Panel.

Now, let's set the layout of the panel to give it some size:

- Set the width to 250.
- Set the length to 400.

Next, we will add a TextureRect to display the graphics for whatever units we are working with:

- Add a new TextureRect node.
- Change the expand mode to Ignore Size.
- Set the layout to have the size be about 250 by 160.
- Drag in one of the unit UI textures (e.g., settler image) as a default. This will be programmatically changed based on what unit is actually being displayed in the UI later on.

Now, let's add some labels to display relevant information about the unit:

- Add a label and name it unitTypeLabel to display the type of unit being played.
- Add another label and name it healthLabel to display the unit's health.
- Add another label and name it movesLabel to display the number of move points the unit has.
- Add a final label and name it actionsTitle to serve as a heading for a section that we will call actions. Set the font to be a little bigger for this label.

Underneath the actionsTitle label, add a VBoxContainer that will hold some actions that each unit can take. This will be a placeholder for now.

Finally, let's add a script for our unit UI:

- Attach a new script to the UnitUI panel and name it UnitUI.cs.

In the script, we will grab references to all of the parts of the unit UI system that we just created and also include a property to reference the Unit we've interacted with:

```
using Godot;
using System;

public partial class UnitUI : Panel
{
    TextureRect unitImage;
    Label unitType, moves, hp;
```

© 2025 Zenva Pty Ltd. All rights reserved.

```
    Unit u;

    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {
        unitImage = GetNode<TextureRect>("TextureRect");
        unitType = GetNode<Label>("UnitTypeLabel");
        hp = GetNode<Label>("HealthLabel");
        moves = GetNode<Label>("MovesLabel");
    }

    public void SetUnit(Unit u)
    {
        this.u = u;

        Refresh();
    }

    public void Refresh()
    {
    }
}
```

In the _Ready function, we grab references to all of these labels and the TextureRect. We also create a function called SetUnit that will go through the process of setting all of this up when we get a particular unit for this UI. We also create a Refresh function that will do most of the work here so that we can refresh this when we move the unit or end the turn or something like that.

In the next lesson, we will do some of that work in the unit class to get the necessary values set up that we will be able to display on the UI.

© 2025 Zenva Pty Ltd. All rights reserved.

In this lesson, we will continue setting up the unit UI system that we started creating in the previous video. In the last video, we laid the foundation by creating the actual unit UI graphics and setting it up in the script.

Now, we need to ensure that our unit keeps track of the necessary values that we want units to keep track of. Specifically, we will focus on the health of the unit and the number of moves that the unit has remaining.

Adding Properties to the Unit Class

To track a unit's vital statistics, we'll create new properties for HP and move points. While it may seem like a single value is enough to represent a unit's HP, we actually need two: one for the current HP and another for the maximum HP, which determines the unit's full health when it spawns. Similarly, we'll use two values to represent move points: one for the maximum available and another for the current amount. This allows us to accurately manage a unit's status and abilities.

Let's add these properties to the Unit class:

```
public int maxHp;
public int hp;

public int maxMovePoints;
public int movePoints;
```

Setting Values for Unit Types

Next, let's assign specific values to our unit types. For a Warrior unit, we'll set the maximum HP to 3, with a starting HP of 3 when they spawn. Their move points will be 1, with a maximum of 1. In contrast, a Settler unit will have a maximum HP of 1, making them vulnerable to destruction since they're not designed for combat. However, Settlers will have a movement advantage, with 2 move points and a maximum of 2, allowing them to move faster than Warriors.

Let's set these values in the Warrior and Settler classes:

```
public partial class Warrior : Unit
{
    public Warrior()
    {
        unitName = "Warrior";
        productionRequired = 50;

        maxHp = 3;
        hp = 3;

        movePoints = 1;
        maxMovePoints = 1;
    }
}

public partial class Settler : Unit
{
    public Settler()
    {
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
        unitName = "Settler";
        productionRequired = 100;

        maxHp = 1;
        hp = 1;

        movePoints = 2;
        maxMovePoints = 2;
    }
}
```

Updating the Unit UI

Back in the unit UI, we can update the refresh function to display the unit's information. We'll set the unit type to the unit's name, which we already have access to. For the move points and HP, we'll use formatted strings to display the current values alongside their maximum values. This will give us a clear representation of the unit's status, showing the number of move points and HP remaining out of their total capacity.

Let's update the Refresh method in the UnitUI class:

```
public void Refresh()
{
    unitType.Text = $"Unit Type: {u.unitName}";
    moves.Text = $"Moves: {u.movePoints}/{u.maxMovePoints}";
    hp.Text = $"HP: {u.hp}/{u.maxHp}";
}
```

Assigning Unit Graphics to the UI

The final step is to assign a specific unit graphic to the UI. To achieve this, we'll revisit the unit class and draw inspiration from our previous work with packed scenes. We'll create a new dictionary, **uiImages**, that maps unit types to their corresponding graphics. To populate this dictionary, we'll define a new static function called loadTextures. This function will initialize the UI_images dictionary and define its contents, linking each unit type to its respective UI graphic.

Let's add the dictionary and the method to load textures in the Unit class:

```
public static Dictionary<Type, Texture2D> uiImages;

public static void LoadTextures()
{
    uiImages = new Dictionary<Type, Texture2D>
    {
        { typeof(Settler), (Texture2D) ResourceLoader.Load("res://textures/settler_im
age.png") },
        { typeof(Warrior), (Texture2D) ResourceLoader.Load("res://textures/warrior_im
age.jpg") }
    };
}
```

© 2024 Zenva Pty Ltd. All rights reserved.

Let's update the _EnterTree method in the Game class as we did before so this is loaded in when the game starts:

```
public override void _EnterTree()
{
    Unit.LoadUnitScenes();
    Unit.LoadTextures();

    TerrainTileUI.LoadTerrainImages();
}
```

Now that we have the dictionary set up, we can return to the unit UI and use it to assign the correct graphic to each unit. We'll set the unit image texture by accessing the dictionary and using the unit's type as the key, which we can obtain using the get_type method. This will allow us to display the corresponding graphic for each unit in the UI.

```
public void Refresh()
{
  unitImage.Texture = Unit.uiImages[u.GetType()];
  unitType.Text = $"Unit Type: {u.unitName}";
  moves.Text = $"Moves: {u.movePoints}/{u.maxMovePoints}";
  hp.Text = $"HP: {u.hp}/{u.maxHp}";
}
```

To complete the UI system setup, we have one final step remaining, which we'll cover in the next video: integrating the unit UI system with the UI manager. We'll tackle this task in the next installment, tying everything together to finalize our UI setup.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back to our course on creating a unit UI display system for our strategy game. In this lesson, we will focus on setting up the unit UI scene inside the UI manager class. This setup is crucial because the UI manager acts as a central coordinating entity between the actual game data (our model) and the UI displays. This architecture helps us avoid a messy web of signals going back and forth between individual UI components.

Setting Up the Unit UI Scene

To begin, we need to create a new packed scene reference for our unit UI scene. This is because we will need to instantiate new instances of the unit UI every time a unit is clicked. We also need a reference to the unit UI itself.

```
// Create a new packed scene reference for the unit UI scene
PackedScene unitUiScene;

// Reference to the unit UI
UnitUI unitUI;
```

Initializing the Unit UI in the Ready Function

Next, we need to initialize the unit UI scene in the ready function. We will copy and paste some of the existing code for other UI components to ensure consistency.

```
public override void _Ready()
{
    // Load the unit UI scene
    unitUiScene = ResourceLoader.Load<PackedScene>("UnitUI.tscn");

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

Hiding All Popups

We need to add the unit UI to the hide all popups function. This function ensures that when we want to hide all UI popups, the unit UI is also destroyed and its reference is set to null.

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

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

© 2024 Zenva Pty Ltd. All rights reserved.

Refreshing the Unit UI

We need to add the unit UI to the refresh UI function. This function ensures that if the unit UI is not null, we call the refresh method on it.

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

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

Setting the Unit UI

Finally, we need a function to set the unit UI, passing in a unit. This function will hide all current popups, destroy the current unit UI, create a new unit UI, add it as a child, set the unit, and call the refresh method.

```
public void SetUnitUI(Unit u)
{
  HideAllPopups();

  unitUi = unitUiScene.Instantiate() as UnitUI;
  AddChild(unitUi);
  unitUi.SetUnit(u);
}
```

Summary

In this lesson, we have set up the unit UI scene inside the UI manager class. We created a new packed scene reference for the unit UI scene, initialized it in the ready function, added it to the hide all popups function, added it to the refresh UI function, and created a function to set the unit UI. This setup ensures that our unit UI is properly integrated into our game's UI system.

In the following lessons, we will start creating a system of interaction with our units, beginning with figuring out how to register a mouse click on a particular unit.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back to our series! In the previous lesson, we completed our shiny new unit UI. However, we still need a way to interact with our units by clicking on them. In this lesson, we will create a system to select units on the map by clicking on them.

Objectives

- Modulate the unit's color when clicked.
- Register the click event with the unit class.
- Override the unhandled input function to detect mouse clicks.

Define Selection Functions & Properties

First, we need to define two new functions inside the unit class: setSelected and setDeselected. These functions will handle the visual feedback and logical registration of the unit's selection state. We can add these to the Unit class.

```
public void SetSelected()
{

}

public void SetDeselected()
{

}
```

We need a Boolean variable to keep track of whether a unit is currently selected or not. Add this variable to the unit class properties:

```
public bool selected = false;
```

With our property defined, let's set up our SetSelected function first. The first thing we want to do is set our boolean property to true. Next, we will grab the sprite and modulate its color slightly so we can tell it is the selected unit.

```
public void SetSelected()
{
    selected = true;
    Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");
    Color c = new Color(sprite.Modulate);
    c.V = c.V - 0.25f;
    sprite.Modulate = c;
}
```

Next, we'll add functionality for deselection. For now, we just want to reset our selected boolean to false and modulate the sprites color back to its previous.

```
public void SetDeselected()
{
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  selected = false;

  GetNode<Sprite2D>("Sprite2D").Modulate = civ.territoryColor;
}
```

Adjust Collider

In order for us to detect collisions, we need to get a reference to our collider. To do this, we'll need a property to reference it. We can fill the variable in our _Ready function.

```
public Area2D collider;

public override void _Ready()
{
  collider = GetNode<Area2D>("Sprite2D/Area2D");
}
```

Override the Unhandled Input Function

Next, we need to override the _UnhandledInput function in the unit class to start polling for mouse events. This function will detect left mouse button clicks and check if the click overlaps with the unit's collider.

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mouse && mouse.ButtonMask == MouseButtonMask.
Left)
    {
        var spaceState = GetWorld2D().DirectSpaceState;
        var point = new PhysicsPointQueryParameters2D();
        point.CollideWithAreas = true;
        point.Position = GetGlobalMousePosition();
        var result = spaceState.IntersectPoint(point);
        if (result.Count > 0 && (Area2D)result[0]["collider"] == collider)
        {
            setSelected();
            GetViewport().SetInputAsHandled();
        }
        else
        {
            setDeselected();
        }
    }
}
```

Let's break it down:

1. We get a reference to the DirectSpaceState of the 2D world, which allows us to perform physics queries.
2. We create a PhysicsPointQueryParameters2D object to define the query. We set

© 2024 Zenva Pty Ltd. All rights reserved.

CollideWithAreas to true to include area colliders in the query.
3. We set the Position of the query to the current mouse position using GetGlobalMousePosition().
4. We call IntersectPoint on the DirectSpaceState to perform the query, passing in the point object. This returns a list of colliders that intersect with the mouse position.
5. If the list is not empty and the first collider is the same as the collider variable, we call SetSelected() to select the object. We also mark the input as handled using GetViewport().SetInputAsHandled() to prevent other parts of the game from responding to the same input event.
6. If the list is empty or the first collider is not the same as the collider variable, we call SetDeselected() to deselect the object.
With this code, when the player clicks on an object with the left mouse button, it will be selected, and when they click elsewhere, it will be deselected.

Conclusion

In this lesson, we implemented a system to select units on the map by clicking on them. We defined selection functions, added a Boolean variable for the selection state, overrode the unhandled input function, and registered the collider. This allows us to interact with units by clicking on them, providing visual feedback and logical registration of the selection state.

In the next lesson, we will continue to build upon this functionality by adding more interactive features to our units.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will continue developing the system for selecting units on the map. We'll address a few issues that arose in the previous implementation and enhance the functionality to ensure a smoother user experience.

Issues to Address

- The UI for the selected unit is not hooked up yet.
- If a tile is selected and then a unit is selected, the tile does not get unselected, leading to a visually confusing state.

Solution Overview

To resolve these issues, we will:

1. Create a signal in the Unit class to represent when a unit is clicked.
2. Hook up this signal to the UI manager to display the unit's information in the UI.
3. Ensure that selecting a unit deselects any previously selected tile.

Setting up Warriors and Settlers

Before we fix our major problems, there is one extra thing we need to hook everything up: tell our Warriors and Settlers classes to use the Ready function from the Unit. We can do so by simply adding the following to both classes:

```
public override void _Ready()
{
  base._Ready();
}
```

This makes sure our logic is properly called within our subclasses as we fix our problems in the next steps.

Implementing the Signal in the Unit Class

Let's tackle our major issues now. First, we need to create a signal in the Unit class that will be emitted whenever a unit is clicked. This signal will allow other parts of the game to react to the unit being selected.

```
[Signal]
public delegate void UnitClickedEventHandler(Unit u);
```

Next, we need to hook up this signal to the UI manager so that the unit's information is displayed in the UI when the unit is clicked.

```
public override void _Ready()
{
  collider = GetNode<Area2D>("Sprite2D/Area2D");

  UIManager manager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");
  this.UnitClicked += manager.SetUnitUI;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
}
```

Ensuring Tile Deselection

To ensure that selecting a unit deselects any previously selected tile, we need to modify the HexTileMap class to include a method for deselecting the current cell.

```
public void DeselectCurrentCell(Unit u = null)
{
  overlayLayer.SetCell(currentSelectedCell, -1);
}
```

Back in the Unit class, we'll make use of this. We'll first get a reference to our HexTileMap. Then, in the Ready function, we'll subscribe our new function so that when we click our unit, it deselect the tile.

```
public HexTileMap map;

  public override void _Ready()
  {
    collider = GetNode<Area2D>("Sprite2D/Area2D");

    UIManager manager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");
    this.UnitClicked += manager.SetUnitUI;

    map = GetNode<HexTileMap>("/root/Game/HexTileMap");
    this.UnitClicked += map.DeselectCurrentCell;
  }
```

In our _UnhandledInput function, we now just need to emit the signal when the unit is selected.

```
if (result.Count > 0 && (Area2D) result[0]["collider"] == collider)
{
  EmitSignal(SignalName.UnitClicked, this);
  SetSelected();
  GetViewport().SetInputAsHandled();
} else {
  SetDeselected();
}
```

After implementing these changes, we can test the system by building a unit and clicking on it. We should see the unit's UI displaying and the tile deselection logic working correctly.

Fixing the Unit Darkening Issue

One final issue to address is that clicking on a unit multiple times causes it to darken excessively. We can fix this by adding a check to ensure that the unit only darkens if it is not already selected.

© 2024 Zenva Pty Ltd. All rights reserved.

```
public void SetSelected()
{
    if (!selected)
    {
        selected = true;

        Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");
        Color c = new Color(sprite.Modulate);
        c.V = c.V - 0.25f;
        sprite.Modulate = c;
    }
}
```

With these changes, we have successfully created a system for clicking on and selecting units on the map. In the next few videos, we will move forward with creating a system to actually move our units around the map.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back! In the previous lesson, we completed the system for clicking on our units. Now, we're going to put this into action by creating a system to allow our units to move on the map. The goal is to select a unit and right-click somewhere on the map within the range of acceptable moves, causing the unit to move there.

Overview of the Movement System

This movement system will be complex with many interacting parts. Here's a breakdown of what we'll cover:

- Calculating valid adjacent movement hexes for a unit.
- Defining impassable terrain types.
- Updating the unit's valid movement hexes in various states (ready, selected, deselected).

Calculating Valid Adjacent Movement Hexes

To simplify the complexity, we'll limit the movement to one tile at a time before having to click again. We'll create a function called **CalculateValidAdjacentMovementHexes** that calculates from any given position what tiles a unit can move into.

Here's how we can implement this function in our Unit class:

```
public List<Hex> CalculateValidAdjacentMovementHexes()
{
    List<Hex> hexes = new List<Hex>();

    hexes.AddRange(map.GetSurroundingHexes(this.coords));

    return hexes;
}
```

We first make a list of hexes that we'll be able to move to. Then, we'll get the surrounding hexes based on the unit's position. Then, we return the hexes list.

Defining Impassable Terrain Types

We need to define a set of impassable terrain types that the units will adhere to. This will ensure that units cannot move into water, shallow water, ice, or mountains.

Here's how we can define this set:

```
public HashSet<TerrainType> impassible = new HashSet<TerrainType>
{
    TerrainType.WATER,
    TerrainType.SHALLOW_WATER,
    TerrainType.ICE,
    TerrainType.MOUNTAIN
};
```

Back in our CalculateValidAdjacentMovementHexes function, we can now populate our list by making sure to only add tiles that are passable:

© 2024 Zenva Pty Ltd. All rights reserved.

```
public List<Hex> CalculateValidAdjacentMovementHexes()
{
  List<Hex> hexes = new List<Hex>();
  hexes.AddRange(map.GetSurroundingHexes(this.coords));
  hexes = hexes.Where(h => !impassible.Contains(h.terrainType)).ToList();
  return hexes;
}
```

Updating Valid Movement Hexes

We need to update the unit's valid movement hexes in various states:

- When the unit is ready (spawns on the map).
- When the unit is selected.
- When the unit is deselected.

When the Unit is Ready

We'll create a property in the unit class called validMovementHexes that represents the valid movement hexes at any given time. Then, in the _Ready function, we'll set this property to the return value of the calculateValidAdjacentMovementHexes function.

```
public List<Hex> validMovementHexes = new List<Hex>();

public override void _Ready()
{
    validMovementHexes = CalculateValidAdjacentMovementHexes();
}
```

When the Unit is Selected

We'll recalculate the valid movement hexes whenever a unit is selected to ensure no changes have taken place since the last time the unit moved.

```
public void SetSelected()
{
    if (!selected)
    {
        selected = true;

        Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");
        Color c = new Color(sprite.Modulate);
        c.V = c.V - 0.25f;
        sprite.Modulate = c;

        validMovementHexes = CalculateValidAdjacentMovementHexes();
    }
}
```

When the Unit is Deselected

© 2024 Zenva Pty Ltd. All rights reserved.

We'll clear out the valid movement hexes whenever a unit is deselected.

```
public void SetDeselected()
{
    selected = false;

    validMovementHexes.Clear();

    GetNode<Sprite2D>("Sprite2D").Modulate = civ.territoryColor;
}
```

That's it for this lesson! In the next lesson, we'll continue building on this movement system to handle the actual movement of the units on the map.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back! In this lesson, we will continue building the movement system for units in our game. We will tackle the core of the movement problem by registering the units' actual positions on the map and creating the functions to carry out the movement itself.

Registering Unit Positions

To keep track of all units' positions on the map, we will create a new property on the Unit class. This property will be a static dictionary that will be shared among all instances of the Unit class or its subclasses.

```
public static Dictionary<Hex, List<Unit>> unitLocations = new Dictionary<Hex, List<Un
it>>();
```

The rationale for having a list of units rather than a single unit is to allow for the possibility of stacking multiple units on a particular tile in the future. For now, we will only handle one unit at a time in a tile.

We will instantiate this dictionary in the _Ready function and register every new unit in its starting position with this dictionary. To facilitate this, we need to create a helper function in the HexTileMap class to get the hex data for a particular coordinate.

```
public Hex GetHex(Vector2I coords)
{
    return mapData[coords];
}
```

Now, we can populate our list in our _Ready function. Let's also handle the case where the dictionary does not contain the key for the current hex – in which case we'll add it to a new list.

```
if (unitLocations.ContainsKey(map.GetHex(this.coords)))
{
  unitLocations[map.GetHex(this.coords)].Add(this);
} else {
  unitLocations[map.GetHex(this.coords)] = new List<Unit>{this};
}
```

Creating Movement Functions

Now we can set up our actual movement function. We will create two functions here. One will be a wrapper function called Move, and the other will be the main movement function called MoveToHex.

The Move function will handle some initial checks before calling the main movement function. In this case, we just want to check if we have a unit selected and they *can* move. Then, we check that the hex they're moving to is valid. If so, we can move the unit and emit our unit clicked signal.

```
public void Move(Hex h)
{
    if (selected && movePoints > 0)
    {
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
        if (validMovementHexes.Contains(h))
        {
            MoveToHex(h);
            EmitSignal(SignalName.UnitClicked, this);
        }
    }
}
```

The MoveToHex function will handle the actual movement logic. We will start by checking if the target hex is unoccupied. We will then remove the unit from our Unit Location list, get the coordinates of our hex on the map, and change the coords parameter tracking our unit location to those coordinates as well

```
public void MoveToHex(Hex h)
{
    if (!unitLocations.ContainsKey(h) || (unitLocations.ContainsKey(h) && unitLocatio
ns[h].Count == 0))
    {
        unitLocations[map.GetHex(this.coords)].Remove(this);

        Position = map.MapToLocal(h.coordinates);
        coords = h.coordinates;
}
```

In this lesson, we have laid the groundwork for unit movement by registering unit positions and creating the basic movement functions. In the following video, we will continue to refine and expand these functions to handle more complex scenarios.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back to our course on developing a movement system for units in a turn-based strategy game using Godot. In this lesson, we will continue working on our MoveToHex function, which is part of the movement system we are developing for the units we've spawned on the map. So far, we have updated the unit's position, moved the unit in 2D space, and removed the unit from its current position or its previous position. Now, we need to update the unit to its new position.

Updating the Unit Locations Dictionary

To update the unit to its new position, we need to check if the unit locations dictionary contains a key for this hex. If it does not, this means this hex has never been visited before. In this case, we need to create a new list as the value for this hex and add this unit as a single element. Otherwise, if the hex has been visited before, we simply add this unit to the already existing list.

```
if (!unitLocations.ContainsKey(h))
{
    unitLocations[h] = new List<Unit> { this };
}
else
{
    unitLocations[h].Add(this);
}
```

Recalculating Valid Movement Hexes

Next, we need to recalculate our valid movement hexes now that we've moved to this new location. We also need to subtract our move points as we just moved by one tile. This is all we need in this section of this if block. Later on, we will add combat in another section, but that will happen in subsequent videos.

```
validMovementHexes = CalculateValidAdjacentMovementHexes();
movePoints -= 1;
```

Handling Move Points Replenishment

One thing we need to do before we can create signals and make this whole system work is manage our move points. We haven't done much with these yet, because we hadn't implemented movement. We are subtracting move points now, but we don't have a way yet to replenish these move points. The way we want it to work is pretty simple: every time there's a new turn, we want the move points of a unit to be replenished.

To achieve this, we will create a ProcessTurn function for units that will reset the move points back to the maximum move points. Then, to make sure that this process turn actually occurs in the ready function, we will add a line to hook up the signal for the UI manager where we are processing that turn.

```
public void ProcessTurn()
{
    movePoints = maxMovePoints;
}

// In the ready function
manager.EndTurn += this.ProcessTurn;
```

© 2024 Zenva Pty Ltd. All rights reserved.

Setting Up Right-Click Signal for Unit Movement

All that we have left to do to finish this movement system is to have some way for the unit to be able to detect when the map has been clicked on to issue a move signal. We will use right-clicking for unit movement to distinguish it from selecting tiles. The first thing we'll need to do here is to go back to our HexTileMap class and set up a signal for right-clicking on the map.

This is going to be a default C# signal because our Hex class does not inherit from the Godot node hierarchy as a performance consideration. We've done a few of these before already, such as the sendHexDataEventHandler.

```
public delegate void RightClickOnMapEventHandler(Hex h);
public event RightClickOnMapEventHandler RightClickOnMap;
```

Emitting the Right-Click Signal

Next, we need to determine where we want to actually emit this signal. We already have a place in the unhandledInput function of the HexTileMap class where we check if the button mask is the left mouse button. We can create another possibility here that will be if the mouse button mask is the right mouse button. If we get a right click, we will just emit this signal.

```
if (mouse.ButtonMask == MouseButtonMask.Left) {
    // Existing code
}
if (mouse.ButtonMask == MouseButtonMask.Right) {
    RightClickOnMap?.Invoke(h);
}
```

Hooking Up the Right-Click Signal to the Unit's Move Function

Finally, we just need to hook that up to the unit's move function. Back in the unit's ready function, we will say map.RightClickOnMap += Move;.

```
public override void _Ready()
{
    map.RightClickOnMap += Move;
}
```

Testing the Movement System

To test the movement system, follow these steps:

1. Find a city.
2. Queue up a unit.
3. Right-click on the map to move the unit.
4. Check if the unit moves correctly and if the move points are subtracted.
5. End the turn to see if the move points replenish.
6. Test movement across different terrain types to ensure proper terrain restrictions.
7. Test if units successfully block each other's movement.

© 2024 Zenva Pty Ltd. All rights reserved.

If you encounter any issues, such as units moving across impassable terrain, make sure to check the CalculateValidAdjacentMovementHexes function for any errors. Ensure that the result of the query is correctly assigned to the hexes list.

With that, we have successfully created a movement system. In the next few videos, we will be creating systems for unit combat, mechanics for settlers founding cities, capturing cities, and giving the power of spawning and moving units to the AI cities.

© 2024 Zenva Pty Ltd. All rights reserved.

In the previous lesson, we completed the system that allows units to move around the map. Now, we will start implementing mechanics for these units to interact with the world. The first mechanic we will create is the ability for settler units to found cities. When a settler unit is selected, we want to have an option in the unit UI to found a city. When this action is performed, a new city will be created that will be part of the civilization to which the settler belongs.

Steps to Implement the City Founding Mechanic

To achieve this, we need to perform two main tasks:

1. Create a new function on the settler class to actually create a city.
2. Create an action in the unit UI to give us a button with which to activate the create city function.

Creating the Found City Function

First, we will write the FoundCity function inside the Settler class. This function will handle the logic for founding a city.

```
public partial class Settler : Unit
{
    public void FoundCity()
    {
        if (map.GetHex(this.coords).ownerCity is null && !City.invalidTiles.ContainsK
ey(map.GetHex(this.coords)))
        {
            bool valid = true;
            foreach (Hex h in map.GetSurroundingHexes(this.coords))
            {
                valid = h.ownerCity is null && !City.invalidTiles.ContainsKey(h);
            }

            if (valid)
            {
                map.CreateCity(this.civ, this.coords, $"Settled City {coords.X}");
                this.DestroyUnit();
            }
        }
    }
}
```

In this function, we perform several checks to ensure that the settler can found a city in the current location:

- The tile the settler is on should not be owned by any city.
- The tile should not be in the list of invalid tiles.
- The surrounding tiles should also be valid (not owned by any city and not in the list of invalid tiles).

If all these conditions are met, we call the CreateCity function to create a new city and then destroy the settler unit.

Destroying the Unit

© 2024 Zenva Pty Ltd. All rights reserved.

To destroy the unit, we need to create a new function in the Unit base class. This function will handle the destruction and cleanup of unit resources.

```
public partial class Unit : Node2D
{
    public void DestroyUnit()
    {
        map.RightClickOnMap -= Move;

        if (selected)
        {
            SelectedUnitDestroyed?.Invoke();
        }

        this.civ.units.Remove(this);
        unitLocations[map.GetHex(this.coords)].Remove(this);

        this.QueueFree();
    }
}
```

In this function, we perform the following steps:

- Disconnect the movement signal to prevent the unit from receiving movement signals after it is destroyed.
- If the unit is selected, we invoke the SelectedUnitDestroyed signal to notify the UI manager to hide the unit UI.
- Remove the unit from the civilization's unit registry and the master list of unit locations.
- Finally, we call QueueFree to remove the unit from the scene tree.

We do, of course, need to create our SelectedUnitDestroyed signal. To do so, we first set up the delegate in the Unit class as we've done in other classes:

```
public delegate void SelectedUnitDestroyedEventHandler();
public event SelectedUnitDestroyedEventHandler SelectedUnitDestroyed;
```

Then, in our Ready function, we will subscribe our UI Manager's Hide Popups function to this signal.

```
public override void _Ready()
{
  this.SelectedUnitDestroyed += manager.HideAllPopups;
  }
```

With this, our logic for founding a city is set up. However, we have no way yet to actually call this function. We will be working on that in the next lesson!

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will continue building the mechanics for our settler units in the game. Specifically, we will focus on adding a user interface (UI) element that allows players to activate the settler unit's ability to found a city. This involves adding a button to the UI that appears only for settler units and hooking up this button to the appropriate function in the settler class.

Adding the Found City Button

To add the "Found City" button to the UI, we need to modify the UnitUI class. This class is responsible for displaying information about the selected unit. We will add logic to check if the selected unit is a settler and, if so, add a button that allows the player to found a city.

Modifying the UnitUI Class

Let's start by modifying the UnitUI class to include the logic for adding the "Found City" button. Specifically, we want to adjust our **SetUnit function**.

```
public void SetUnit(Unit u)
{
        this.u = u;

        if (this.u.GetType() == typeof(Settler))
        {
            VBoxContainer actionsContainer = GetNode<VBoxContainer>("VBoxContainer");
            Button foundCityButton = new Button();
            foundCityButton.Text = "Found City";
            actionsContainer.AddChild(foundCityButton);

            Settler settler = this.u as Settler;
            foundCityButton.Pressed += settler.FoundCity;
        }

        Refresh();
    }
}
```

Explanation of the Code

- **Checking the Unit Type:** The code checks if the selected unit is a settler by comparing its type to typeof(Settler).
- **Adding the Button:** If the unit is a settler, a new button is created and added to the VBoxContainer in the UI.
- **Connecting the Button to the Function:** The button's Pressed signal is connected to the FoundCity method of the settler unit.

Testing the Settler Mechanic

Now that we have added the "Found City" button to the UI, we can test the settler mechanic in the game. Follow these steps to test the functionality:

1. Select a settler unit in the game.
2. Verify that the "Found City" button appears in the UI.
3. Move the settler unit to a location where you want to found a city.

© 2024 Zenva Pty Ltd. All rights reserved.

4. Click the “Found City” button to found a new city.

If everything is set up correctly, you should see a new city appear on the map, and the settler unit will be removed from the game.

Conclusion

In this lesson, we added a “Found City” button to the UI for settler units, allowing players to found new cities. This is an important step in building the game mechanics and providing players with the ability to expand their civilizations. In the next lesson, we will take things to another level by giving AI civilizations and cities the ability to spawn and move units, which will really start to bring the world to life.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will focus on enabling other civilizations to spawn and move units on the map. This functionality will be implemented in the Civilization class, specifically within the ProcessTurn function. This process will apply to all civilizations, regardless of whether they are player-controlled or not.

Adding Units to the Build Queue

First, we need to ensure that each city in a civilization has a chance to add units to its build queue. We will use random number generation to determine the probability of spawning units each turn. We want to wrap this in a check to make sure it's not the player's civilization, however. This should only apply to the AI cities. Here's how you can do it:

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

    if (!playerCiv)
    {
        Random r = new Random();

        foreach (City c in cities)
        {
            int rand = r.Next(30);

            if (rand > 27)
            {
                c.AddUnitToBuildQueue(new Warrior());
            }

            if (rand > 28)
            {
                c.AddUnitToBuildQueue(new Settler());
            }
        }
}
```

Creating a Helper Function for Random Movement

Next, we need to create a helper function in the Unit class that will allow the AI to make random moves with their units. This function will calculate the valid adjacent movement hexes and randomly pick one.

```
public void RandomMove()
{
    Random r = new Random();
    validMovementHexes = CalculateValidAdjacentMovementHexes();
    Hex h = validMovementHexes.ElementAt(r.Next(validMovementHexes.Count));

    MoveToHex(h);
}
```

© 2024 Zenva Pty Ltd. All rights reserved.

With our helper function setup, back in the **ProcessTurn** function we want to simply loop through each unit and have them randomly move.

```
foreach (Unit u in units)
{
    u.RandomMove();
}
```

Handling Settler City Founding

Next, let's add the ability for settlers to randomly found a city within this loop

However, there is a catch: when a settler unit founds a city, it will be destroyed. Ssince this action occurs within a foreach loop, we need to handle the destruction carefully to avoid modifying the collection while it is being iterated through; modifying a collection during a loop is against C# coding principles.

We will create a separate list to keep track of settlers that need to found cities and process this list after the main loop.

```
  List<Settler> citiesToFound = new List<Settler>();
  foreach (Unit u in units)
  {
    u.RandomMove();
    if (u is Settler && r.Next(10) > 8)
    {
      Settler s = u as Settler;
      citiesToFound.Add(s);
    }
  }
  for (int i = 0; i < citiesToFound.Count; i++)
  {
    citiesToFound[i].FoundCity();
  }
}
```

Capping the Number of Units

To prevent performance issues, we need to cap the number of units a civilization can have. We will introduce a new property called maxUnits in the Civilization class and set it to a default value of 3. This value will be updated based on the number of cities the civilization has.

```
public int maxUnits = 3;
```

However, this will apply to the entire civilization, not each city. Thus, in our ProcessTurn function, we want to multiple the maxUnits by the number of cities it has.

```
maxUnits = this.cities.Count * 3;
```

© 2024 Zenva Pty Ltd. All rights reserved.

We will then enforce this cap in the City class by checking the maximum number of units before adding a unit to the build queue.

```
public void AddUnitToBuildQueue(Unit u)
{
    if (this.civ.maxUnits > this.civ.units.Count)
        unitBuildQueue.Add(u);
}
```

Testing the Implementation

Finally, we can test our work by running the game and observing the civilizations on the map. They should now be spawning units and founding cities. This will create a dynamic and living process on the map, with civilizations expanding and interacting with each other.

In the next lesson, we will start working on combat.

© 2024 Zenva Pty Ltd. All rights reserved.

In the last video, we successfully implemented the ability for non-player civilizations to spawn units and found cities, adding a significant layer of dynamism to the game. The next step is to introduce a combat system that allows units from different factions to engage in battles. This will be a crucial aspect of the game, as it will enable interactions between civilizations and add a strategic element to the gameplay.

Plan for Combat System

Our plan is to create a simple combat system where any unit from an opposing faction can initiate combat with another unit. Additionally, we want warrior units to be able to capture cities of opposing factions. Since we don't have a complex diplomacy system, all factions will be constantly at war with one another.

Adding Attack Value to Units

Currently, our units have a health (HP) value, but they lack an attack value. We will add an attack value to each unit type to determine the damage they can inflict during combat. First, in the Unit class itself, let's add the property:

```
public int attackVal;
```

In the Warrior class, we'll set the attack value to 2.

```
attackVal = 2;
```

In the Settler class, we'll set the attack value to 0.

```
attackVal = 0;
```

Implementing the Combat Function

To implement combat, we will create a new function called CalculateCombat in the Unit class. This function will take two units as parameters: the attacker and the defender. The defender's HP will be reduced by the attacker's attack value, and the attacker's HP will be reduced by half of the defender's attack value. If either unit's HP drops to zero or below, that unit will be destroyed.

```
public void CalculateCombat(Unit attacker, Unit defender)
{
    defender.hp -= attacker.attackVal;
    attacker.hp -= defender.attackVal / 2;

    if (defender.hp <= 0)
    {
        defender.DestroyUnit();
    }

    if (attacker.hp <= 0)
    {
        attacker.DestroyUnit();
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    }
}
```

Initiating Combat

We will initiate combat anytime a unit from an opposing civilization tries to move into a hex occupied by another unit from an opposing civilization. This will be done in the MoveToHex function in the Unit class. Since we already check for empty hexes, if that condition fails, we can assume we found a unit to attack. We can therefore put our logic in an else statement.

Our logic here is pretty simple. We get the unit we're going to attack, make sure it doesn't belong to the same civ, and then call our CalculateCombat function.

```
public void MoveToHex(Hex h)
{
    if (!unitLocations.ContainsKey(h) || (unitLocations.ContainsKey(h) && unitLocatio
ns[h].Count == 0))
    {
        // Existing code
    }
    else
    {
        // Combat
        Unit opp = unitLocations[h][0];

        if (opp.civ != this.civ)
        {
            CalculateCombat(this, opp);
        }
    }
}
```

Testing the Combat System

To test the combat system, we can open the game and observe the interactions between civilizations. We should see units engaging in combat and potentially capturing cities. This will add a new layer of strategy and excitement to the game.

In the next video, we will take the next logical step by allowing warrior units to capture city centers and add cities to their own civilization, adding real dynamism to what happens on the map.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will introduce a mechanic to facilitate the capturing of cities by warrior units on the map. This might sound complicated, but we have already laid a solid foundation with the functions and classes we have created so far. This mechanic will involve two main parts:

Part 1: Change Ownership Function in the City Class

First, we need to create a function in the City class that will change the ownership of a city from one civilization to another. This function will handle all the logistical registry changes that come with changing ownership.

Steps to Implement the Change Ownership Function

1. Create a new function called ChangeOwnership in the City class.
2. This function will take in a Civilization object as a parameter, which we will call newOwner.
3. Keep track of the current civilization that owns the city before overwriting it.
4. Remove the city from the current owner's registry.
5. Add the city to the new owner's registry.
6. Set the city's civilization property to the new owner.
7. Set the icon color to the territory color of the new owner.
8. Update the territory map of both the new owner and the old owner to reflect the changes.

Code Implementation

```
public void ChangeOwnership(Civilization newOwner)
{
    Civilization oldOwner = this.civ;

    this.civ.cities.Remove(this);
    newOwner.cities.Add(this);

    this.civ = newOwner;

    SetIconColor(newOwner.territoryColor);

    map.UpdateCivTerritoryMap(newOwner);
    map.UpdateCivTerritoryMap(oldOwner);
}
```

Part 2: Capturing Cities in the Unit Class

Next, we will introduce some logic in the Unit class to handle the capturing of cities when a warrior unit occupies a city center.

Steps to Implement the Capture Logic

1. In the MoveToHex function, add a check to see if the tile is a city center and if the civilization owning the city is not the same as the civilization owning the unit.
2. If the unit is a warrior, change the ownership of the city to the civilization that owns the unit.

Code Implementation

```
public void MoveToHex(Hex h)
{
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    if (!unitLocations.ContainsKey(h) || (unitLocations.ContainsKey(h) && unitLocatio
ns[h].Count == 0))
    {
        // Existing code

        if (h.isCityCenter && h.ownerCity.civ != this.civ && this is Warrior)
        {
            h.ownerCity.ChangeOwnership(this.civ);
        }
    }
    else
    {
        // Combat
        Unit opp = unitLocations[h][0];

        if (opp.civ != this.civ)
        {
            CalculateCombat(this, opp);
        }
    }
}
```

Testing the Capture Mechanic

To test the capture mechanic, you can start a game and queue a warrior unit. Once the warrior is ready, you can move it to a city center owned by another civilization. If the capture mechanic is working correctly, the city should change ownership to the player's civilization.

In the next lesson, we will take a look at creating a map overlay for highlighting cities to give the game a bit better polish.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will create a system to highlight city-owned tiles on the map. The goal is to make it so that when a city is selected, all tiles other than the city's territory are covered by a transparent, dark overlay. This will give the appearance that only the city's territory is highlighted.

Creating a New Tile Map Layer

To achieve this, we need to create a new tile map layer called "Highlight Layer." This layer will be placed between the "Civilization Colors Layer" and the "Hex Borders Layer."

Setting Up the Tile Set

Next, we need to set up the tile set for the highlight layer. Follow these steps:

1. Create a new tile set.
2. Drag the "hex 128.png" texture from the "textures" folder into the tile set.
3. Ensure the tile set has the correct characteristics: hexagons of size 128×128.

Initializing the Highlight Layer

Now, we will create a script for the highlight layer to initialize it. The script will include basic variables for keeping track of the map's height and width, a list of hexes representing the currently highlighted hexes, and a reference to the city object.

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

public partial class HighlightLayer : TileMapLayer
{
    int w;
    int h;

    List<Hex> currentlyHighlighted = new();

    City current;
}
```

Setting Up the Highlight Layer

Let's first create a function to set up the highlight layer. The SetupHighlightLayer method initializes the highlight layer by setting the width and height of the map and filling the layer with the default shaded tile. The layer is initially set to be invisible.

```
public void SetupHighlightLayer(int w, int h)
{
  this.w = w;
  this.h = h;
  for (int x = 0; x < w; x++)
  {
    for (int y = 0; y < h; y++)
    {
      SetCell(new Vector2I(x, y), 0, new Vector2I(0, 3));
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    }
  }
  Visible = false;
}
```

Highlighting a City's Territory

Next, we want to make a function to call so we highlight the territory of a specific city. This function first resets the highlight layer, then adds the city's territory and border tiles to the highlighted list, and sets the corresponding cells to be highlighted. Finally, it makes the layer visible.

```
public void SetHighlightLayerForCity(City c)
{
  ResetHighlightLayer();
  current = c;
  foreach (Hex h in c.territory)
  {
    currentlyHighlighted.Add(h);
    SetCell(h.coordinates, -1);
  }
  foreach (Hex h in c.borderTilePool)
  {
    currentlyHighlighted.Add(h);
    SetCell(h.coordinates, -1);
  }
  Visible = true;
}
```

Resetting the Highlight Layer

Per the above, we also need a helper function that actually resets the highlight layer. All this function does is clear the highlighted list and toggle our properties back to the default state.

```
public void ResetHighlightLayer()
{
  foreach (Hex h in currentlyHighlighted)
  {
    SetCell(h.coordinates, 0, new Vector2I(0, 3));
  }
  current = null;
  Visible = false;
}
```

Challenge: Implementing the Refresh Highlight Layer Function

As a challenge, you are tasked with implementing a function to refresh the highlight layer. This function should recalculate the layer for the currently selected city. If no city is selected, it should set the layer to be invisible. This will ensure that the highlight layer reflects any changes in the city's territory, such as growth in population or changes in the turn.

© 2024 Zenva Pty Ltd. All rights reserved.

Good luck with your implementation! In the next video, we will review the solution to this challenge.

© 2024 Zenva Pty Ltd. All rights reserved.

In this lesson, we will first go over the solution to the challenge posed at the end of the last video. Then, we will finish up our highlight layer system. We need some code to refresh the highlight layer because if a city is selected when a turn is ending, we need to refresh the layer without totally resetting it. Let's dive into the details.

Refreshing the Highlight Layer

To refresh the highlight layer, we need to check if a city is currently selected. If it is, we will repeat the logic to highlight the city's territory and border tiles. We also make sure the layer is visible. If no city is selected, but we hide the layer.

```
public void RefreshLayer()
{
  if (current != null)
  {
    foreach (Hex h in current.territory)
    {
      currentlyHighlighted.Add(h);
      SetCell(h.coordinates, -1);
    }
    foreach (Hex h in current.borderTilePool)
    {
      currentlyHighlighted.Add(h);
      SetCell(h.coordinates, -1);
    }
    Visible = true;
  } else {
    Visible = false;
  }
}
```

Integrating the Highlight Layer with the Hex Tile Map

Next, we will integrate the highlight layer with the rest of our map. First, in the HexTileMap class, we'll add a property for the layer:

```
HighlightLayer highlightLayer;
```

In the **Ready()** function, we will then fill the property by getting our node:

```
highlightLayer = GetNode<TileMapLayer>("HighlightLayer") as HighlightLayer;
```

Handling City Selection and Highlighting

Now, we will handle the logic for selecting a city and highlighting its territory. We need to set the highlight layer for the selected city and reset it if a different city is selected or if the player clicks off the city.

First, in the **Ready** function again, we want to set up the highlight layer so it's ready upon being loaded:

© 2024 Zenva Pty Ltd. All rights reserved.

```
highlightLayer.SetupHighlightLayer(width, height);
```

In the **UnhandledInput** function, we want to set the highlight layer when we left-click and a city is found. If we left click on anything other then a city, we want to call our Reset function. Note we need to call this in both else blocks to ensure proper functionality.

```
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 < he
ight)
  {
    Hex h = mapData[mapCoords];
    if (mouse.ButtonMask == MouseButtonMask.Left)
    {
      if (cities.ContainsKey(mapCoords))
      {
        EmitSignal(SignalName.SendCityUIInfo, cities[mapCoords]);
        highlightLayer.SetHighlightLayerForCity(cities[mapCoords]);
      } else {
        highlightLayer.ResetHighlightLayer();
        SendHexData?.Invoke(h);
      }
      if (mapCoords != currentSelectedCell) overlayLayer.SetCell(currentSelectedCell,
 -1);
      overlayLayer.SetCell(mapCoords, 0, new Vector2I(0, 1));
      currentSelectedCell = mapCoords;
    }
    if (mouse.ButtonMask == MouseButtonMask.Right)
    {
      RightClickOnMap?.Invoke(h);
    }
  } else {
    highlightLayer.ResetHighlightLayer();
    overlayLayer.SetCell(currentSelectedCell, -1);
    EmitSignal(SignalName.ClickOffMap);
  }

}
```

Updating the Highlight Layer During Turn Processing

Finally, we need to update the highlight layer during the turn processing. We will call the refresh function we wrote earlier in the **ProcessTurn** function to ensure the highlight layer is updated correctly.

```
// Update the highlight layer during turn processing
highlightLayer.RefreshLayer();
```

© 2024 Zenva Pty Ltd. All rights reserved.

Setting the Modulate Property in Godot Editor

One final thing we need to do is set the modulate property of the highlight layer in the Godot editor. This will make the tile slightly transparent and add a darker color to it. Here are the steps:

1. Set the alpha value of the modulate property to 175.
2. Set the color value of the modulate property to RGB 28, 28, 28.

By following these steps and using the provided code snippets, you will be able to implement the highlight layer system in your game. This feature adds depth to the visuals of the game and improves the user experience by highlighting selected cities.

In the next couple of videos, we will finish up our work on this game by implementing a main menu that allows the player to set settings before starting the game, such as the number of civilizations and the height and width of the map.

© 2024 Zenva Pty Ltd. All rights reserved.

In this tutorial, we will guide you through the process of creating a main menu for your game using Godot. This tutorial will primarily focus on designing the front end of the main menu and later demonstrate how to integrate it with the rest of the game, allowing for various customization options before starting the game.

Creating a New Scene for the Main Menu

The first step is to create a new scene that will contain our main menu. Follow these steps:

1. Create a new scene that inherits from a 2D scene.
2. Name this scene “Main Menu”.

Designing the Background

Next, we will create a basic background using a Polygon2D node and add a background image.

1. Add a Polygon2D node to the scene.
2. Set the size to 4 to create a square shape.
3. Set the size of the Polygon2D to match the Godot default window size (1152×648).
4. Change the color of the Polygon2D to a pleasant blue.
5. Add a Sprite2D node for the background image.
6. Drag the “menu_bg.png” texture from the textures folder into the Sprite2D’s texture property.
7. Adjust the scale of the Sprite2D to fit the window appropriately.
8. Set the opacity of the Sprite2D using the modulate property to make the background color more visible.

Adding a Title Label

We will now add a label to serve as the title of the main menu.

1. Add a Label node to the scene.
2. Set the text of the label to your game’s title (e.g., “Hex-based Strategy Game”).
3. Increase the font size to 48.
4. Change the color of the text to a bright gold.
5. Enable the shadow setting and set the offset to 2 in the x direction for a 3D effect.
6. Position the label in the middle of the screen.

Setting Up Menu Buttons

To organize the menu buttons, we will use VBoxContainer and HBoxContainer nodes.

1. Add a VBoxContainer node to the scene.
2. Add three HBoxContainer nodes inside the VBoxContainer.
3. In the first HBoxContainer, add two Button nodes: one for “Start Game” and one for “Quit”.

Adding Customization Options

We will add three different settings for customization:

- Map width and height
- Number of AI civilizations
- Player civilization color

Map Width and Height

© 2024 Zenva Pty Ltd. All rights reserved.

In the first HBoxContainer, add the following nodes:

1. Add a Label node with the text “Map Width”.
2. Add a SpinBox node for the map width.
3. Duplicate the Label and SpinBox nodes for the map height.
4. Set the range for the map width SpinBox (min: 44, max: 106).
5. Set the range for the map height SpinBox (min: 26, max: 66).

Number of AI Civilizations

In the second HBoxContainer, add the following nodes:

1. Add a Label node with the text “Number of AI”.
2. Add a SpinBox node for the number of AI civilizations.
3. Set the range for the SpinBox (min: 0, max: 6).
4. Enable container sizing and expand the SpinBox to fill the horizontal section.

Player Civilization Color

In the third HBoxContainer, add the following nodes:

1. Add a Label node with the text “Player Color”.
2. Add a ColorPickerButton node for selecting the player’s civilization color.

With these steps, you have created a main menu with customization options. In the next tutorial, we will make final tweaks to the menu and integrate it with the game’s logic.

© 2024 Zenva Pty Ltd. All rights reserved.

Welcome back to our course on creating a main menu for our game. In this lesson, we will finalize the adjustments to our main menu by adding some aesthetic improvements and implementing the logic to start and quit the game.

Adjusting the Main VBox Container

First, we will adjust the separation value in the Theme Override section under Constants to make the buttons a little further apart from each other. This will enhance the aesthetic appeal of our menu.

- Navigate to the Theme Override section under Constants.
- Set the separation value to 25.

Creating the Main Menu Script

Next, we will create a script for the main menu to handle the logic for starting and quitting the game. We will create a new class called MainMenu.cs.

In this class, the first thing we'll create is the quit function. This simply quits out of our game with a single line of code.

```
public void Quit()
{
  GetTree().Quit();
}
```

Next, we'll create a Start function. In this function, we need to do a few things:

- First, we need to get references to our game scene, so we can load it, and our map so we can set the properties.
- Next, we want to take the size values from our spinboxes and set these as the map height and map width. This will allow us to actually pick our map size.
- Likewise, we want to take the spinbox value for our number of AI civs and set that property too. The same applies to our player color.
- Finally, we remove our main menu and add our game to the scene.

```
public void Start()
{
  Game g = ResourceLoader.Load<PackedScene>("Game.tscn").Instantiate() as Game;
  HexTileMap map = g.GetNode<HexTileMap>("HexTileMap");

  map.height = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer/SpinBox2").Va
lue;
  map.width = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer/SpinBox").Valu
e;

  map.NUM_AI_CIVS = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer2/SpinBox
").Value;

  map.PLAYER_COLOR = this.GetNode<ColorPickerButton>("VBoxContainer/HBoxContainer3/Co
lorPickerButton").Color;

  GetNode("/root/MainMenu").QueueFree();
  GetTree().Root.AddChild(g);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
}
```

Connecting Signals in the Editor

Now, we need to connect the signals in the editor to the corresponding methods in our script.

1. Select the Quit button in the editor.
2. Go to the Node tab and find the pressed signal.
3. Connect the pressed signal to the Quit method in the MainMenu script.
4. Repeat the process for the Start button, connecting the pressed signal to the Start method.

Setting the Main Menu as the Main Scene

Finally, we need to set the main menu as the main scene in the project settings.

1. Go to Project Settings.
2. Change the Main Scene from game.tscn to main_menu.tscn.

Testing the Main Menu

With all the adjustments and logic in place, we can now test our main menu. Start the game and you should see the main menu with options to change the map size, number of AI civilizations, and player color. Set the desired options and click the Start button to begin the game with the specified parameters.

Congratulations! You have successfully parameterized the game and created a functional main menu. This concludes our lesson on adding content to the 4X strategy game. You have done a lot of good work and created a very nice base for this type of game.

© 2024 Zenva Pty Ltd. All rights reserved.

Congratulations on reaching this point in the course! Let’s take a moment to review what you’ve accomplished thus far. Our learning goals for this course were comprehensive and included the following:

Learning Goals

- Understanding and implementing a unit system
- Creating unit mechanics
- Developing a main menu for the game

Topics Covered

1. Spawning units from cities
2. Creating a unit queue system and its internals
3. Developing a unit UI system, including signal integration for newly spawned units
4. Implementing unit movement on the map, with logic for different terrain types and unit blocking
5. Specific unit mechanics such as combat, settlement, and capturing cities
6. Creating a new map overlay for viewing specific city territories
7. Setting up the game with a main menu to customize the experience each time

By completing these topics, you have gained a solid foundation for creating strategy games in Godot using C#. This knowledge will serve as a valuable asset as you continue to develop your skills in game development.

About Zenva

Zenva is an online learning academy with over a million students. We offer a wide range of courses suitable for both beginners and those looking to expand their knowledge. Our courses are designed to be versatile, allowing you to learn in the way that best suits you. You can choose to watch video tutorials, read lesson summaries, or follow along with the instructor.

Thank you for participating in this course. Your dedication and effort are commendable!

© 2024 Zenva Pty Ltd. All rights reserved.

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 code controls the movement and zooming of a camera in a 2D game. It allows the camera to move within set boundaries and zoom in and out using keyboard inputs or mouse wheel scrolling. The camera's movement speed and zoom speed can be adjusted using the `velocity` and `zoom_speed` 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)
  {
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
   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 code defines a City class that represents a city in a game, with properties such as its location, population, resources, and units being built. The city can grow its population, expand its territory, and build units over time. The class also includes methods for managing the city’s resources, units,

© 2024 Zenva Pty Ltd. All rights reserved.

and ownership.

```
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;

 // Units
 public List<Unit> unitBuildQueue;
 public Unit currentUnitBeingBuilt;
 public int unitBuildTracker = 0;

 // 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>();
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 borderTilePool = new List<Hex>();
 unitBuildQueue = new List<Unit>();
}

public void ProcessTurn()
{
 CleanUpBorderPool();

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

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

 }

 ProcessUnitBuildQueue();
}

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)
{
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 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;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  totalProduction += h.production;
 }
}

// Unit functions

public void AddUnitToBuildQueue(Unit u)
{
 if (this.civ.maxUnits > this.civ.units.Count)
  unitBuildQueue.Add(u);
}

public void SpawnUnit(Unit u)
{
 Unit unitToSpawn = (Unit) Unit.unitSceneResources[u.GetType()].Instantiate();
 unitToSpawn.Position = map.MapToLocal(this.centerCoordinates);
 unitToSpawn.SetCiv(this.civ);
 unitToSpawn.coords = this.centerCoordinates;

 map.AddChild(unitToSpawn);
}

public void ProcessUnitBuildQueue()
{
 if (unitBuildQueue.Count > 0)
 {
  if (currentUnitBeingBuilt == null)
  {
   currentUnitBeingBuilt = unitBuildQueue[0];
  }

  unitBuildTracker += totalProduction;

  if (unitBuildTracker >= currentUnitBeingBuilt.productionRequired)
  {
   SpawnUnit(currentUnitBeingBuilt);

   unitBuildQueue.RemoveAt(0);
   currentUnitBeingBuilt = null;

   unitBuildTracker = 0;
  }
 }
}

public void ChangeOwnership(Civilization newOwner)
{
 Civilization oldOwner = this.civ;

 this.civ.cities.Remove(this);
 newOwner.cities.Add(this);

 this.civ = newOwner;

 SetIconColor(newOwner.territoryColor);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  map.UpdateCivTerritoryMap(newOwner);
  map.UpdateCivTerritoryMap(oldOwner);
 }

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

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

CityUI.cs

Found in folder: /

This code defines a CityUI class that manages the user interface for a city in a game. It updates the city's name, population, food, and production labels, and populates a unit build queue UI. The class also connects signals to handle unit building and refreshing the UI when changes occur.

```
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();

  ConnectUnitBuildSignals(this.city);
 }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
public void ConnectUnitBuildSignals(City city)
{
 VBoxContainer buttons = GetNode<VBoxContainer>("UnitBuildButtons/VBoxContainer");

 UnitBuildButton settlerButton = buttons.GetNode<UnitBuildButton>("SettlerButton");
 settlerButton.u = new Settler();

 UnitBuildButton warriorButton = buttons.GetNode<UnitBuildButton>("WarriorButton");
 warriorButton.u = new Warrior();

 settlerButton.OnPressed += city.AddUnitToBuildQueue;
 settlerButton.OnPressed += this.Refresh;
 warriorButton.OnPressed += city.AddUnitToBuildQueue;
 warriorButton.OnPressed += this.Refresh;
}

public void PopulateUnitQueueUI(City city)
{
 VBoxContainer queue = GetNode<VBoxContainer>("QueueContainer/VBoxContainer");

 foreach (Node n in queue.GetChildren())
 {
  queue.RemoveChild(n);
  n.QueueFree();
 }

 for (int i = 0; i < city.unitBuildQueue.Count; i++)
 {
  Unit u = city.unitBuildQueue[i];

  if (i == 0) // Unit is currently being built
  {
   queue.AddChild(new Label() {
    Text = $"{u.unitName} {city.unitBuildTracker}/{u.productionRequired}"
   });
  } else {
   queue.AddChild(new Label() {
    Text = $"{u.unitName} 0/{u.productionRequired}"
   });
  }

 }
}

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;

 PopulateUnitQueueUI(this.city);

}
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 public void Refresh(Unit u)
 {
  Refresh();
 }
}
```

Civilization.cs

Found in folder: /

This code defines a Civilization class that represents a civilization in a game. The class has properties for the civilization's ID, cities, units, territory color, and name, as well as methods to set a random territory color and process the civilization's turn. During the turn, the civilization's cities process their turns, and if the civilization is not controlled by the player, it may build new units and found new cities.

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

public class Civilization
{

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

 public int maxUnits = 3;

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

 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();
  }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  if (!playerCiv)
  {
   Random r = new Random();

   foreach (City c in cities)
   {
    int rand = r.Next(30);

    if (rand > 27)
    {
     c.AddUnitToBuildQueue(new Warrior());
    }

    if (rand > 28)
    {
     c.AddUnitToBuildQueue(new Settler());
    }
   }

   List<Settler> citiesToFound = new List<Settler>();
   for (int i = 0; i < units.Count; i++)
   {
    Unit u = units[i];
    u.RandomMove();

    if (u is Settler && r.Next(10) > 8)
    {
     Settler s = u as Settler;
     citiesToFound.Add(s);
    }
   }

   for (int i = 0; i < citiesToFound.Count; i++)
   {
    citiesToFound[i].FoundCity();
   }
  }

  maxUnits = this.cities.Count * 3;
 }

}
```

Game.cs

Found in folder: /

This code defines a class called Game that inherits from Node. When the Game node enters the scene tree, it loads unit scenes and textures, and also loads terrain images for the TerrainTileUI. The noise variable is exposed to the editor, allowing for configuration of noise settings.

```
using Godot;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
using System;

public partial class Game : Node
{

 [Export]
 FastNoiseLite noise;

    public override void _EnterTree()
    {
        Unit.LoadUnitScenes();
        Unit.LoadTextures();

        TerrainTileUI.LoadTerrainImages();
    }
}
```

GeneralUI.cs

Found in folder: /

This code defines a UI component called GeneralUI that displays the current turn number. When the component is initialized, it sets the turn number to 0 and updates the label text accordingly. The IncrementTurnCounter method increments the turn number and updates the label text to reflect the new turn number.

```
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

© 2024 Zenva Pty Ltd. All rights reserved.

Found in folder: /

This code defines a hexagonal tile map system for a game, including terrain generation, city creation, and user interaction. It uses the Godot game engine and C# programming language. The script initializes a map with different terrain types, generates resources, and allows for city creation and management. It also handles user input, such as clicking on tiles to select them or display city information.

```
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;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
HighlightLayer highlightLayer;

// 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);


public delegate void RightClickOnMapEventHandler(Hex h);
public event RightClickOnMapEventHandler RightClickOnMap;


// 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");
 highlightLayer = GetNode<TileMapLayer>("HighlightLayer") as HighlightLayer;

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

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

 // Initialize map data
 mapData = new Dictionary<Vector2I, Hex>();
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  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;

  highlightLayer.SetupHighlightLayer(width, height);

 }

 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)
    {
```

© 2024 Zenva Pty Ltd. All rights reserved.

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

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

    }

    if (mouse.ButtonMask == MouseButtonMask.Right)
    {
     RightClickOnMap?.Invoke(h);
    }

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

  }
    }

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

  highlightLayer.RefreshLayer();
 }

 public void DeselectCurrentCell(Unit u = null)
 {
  overlayLayer.SetCell(currentSelectedCell, -1);
 }

 public Hex GetHex(Vector2I coords)
 {
  return mapData[coords];
 }


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

© 2024 Zenva Pty Ltd. All rights reserved.

```
  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++)
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  {
   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);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
    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);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  // 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)
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
{
 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
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  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);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
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]);
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
   }

   // 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);
 }

}
```

HighlightLayer.cs

Found in folder: /

This code defines a HighlightLayer class that manages a layer of highlighted tiles on a tile map. It provides methods to set up the layer, highlight tiles for a specific city, reset the layer, and refresh the layer's visibility. The class keeps track of the currently highlighted tiles and the city being highlighted.

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

public partial class HighlightLayer : TileMapLayer
{

 int w;
 int h;

 List<Hex> currentlyHighlighted = new();

 City current;

 public void SetupHighlightLayer(int w, int h)
 {
  this.w = w;
  this.h = h;

  for (int x = 0; x < w; x++)
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 {
  for (int y = 0; y < h; y++)
  {
   SetCell(new Vector2I(x, y), 0, new Vector2I(0, 3));
  }
 }
 Visible = false;
}

public void SetHighlightLayerForCity(City c)
{
 ResetHighlightLayer();

 current = c;

 foreach (Hex h in c.territory)
 {
  currentlyHighlighted.Add(h);
  SetCell(h.coordinates, -1);
 }

 foreach (Hex h in c.borderTilePool)
 {
  currentlyHighlighted.Add(h);
  SetCell(h.coordinates, -1);
 }

 Visible = true;
}

public void ResetHighlightLayer()
{
 foreach (Hex h in currentlyHighlighted)
 {
  SetCell(h.coordinates, 0, new Vector2I(0, 3));
 }
 current = null;
 Visible = false;
}

public void RefreshLayer()
{
 if (current != null)
 {
  foreach (Hex h in current.territory)
  {
   currentlyHighlighted.Add(h);
   SetCell(h.coordinates, -1);
  }

  foreach (Hex h in current.borderTilePool)
  {
   currentlyHighlighted.Add(h);
   SetCell(h.coordinates, -1);
  }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
   Visible = true;
  } else {
   Visible = false;
  }

 }

}
```

MainMenu.cs

Found in folder: /

This code defines a MainMenu class that extends Node2D, handling the game's main menu functionality. The Start method loads a new game scene, configures its map settings based on user input from spin boxes and a color picker, and then transitions to the new game scene. The Quit method simply exits the game when called.

```
using Godot;
using System;

public partial class MainMenu : Node2D
{

 public void Start()
 {
  Game g = ResourceLoader.Load<PackedScene>("Game.tscn").Instantiate() as Game;
  HexTileMap map = g.GetNode<HexTileMap>("HexTileMap");

  map.height = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer/SpinBox2").Va
lue;
  map.width = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer/SpinBox").Valu
e;

  map.NUM_AI_CIVS = (int) this.GetNode<SpinBox>("VBoxContainer/HBoxContainer2/SpinBox
").Value;

  map.PLAYER_COLOR = this.GetNode<ColorPickerButton>("VBoxContainer/HBoxContainer3/Co
lorPickerButton").Color;

  GetNode("/root/MainMenu").QueueFree();
  GetTree().Root.AddChild(g);
 }


 public void Quit()
 {
  GetTree().Quit();
 }

}
```

© 2024 Zenva Pty Ltd. All rights reserved.

Settler.cs

Found in folder: /

This code defines a Settler unit with specific properties, such as name, production cost, health, and movement points. The Settler has a special ability to found a city on a valid hex tile, checking that the tile and its surroundings are not already owned by another city. If the tile is valid, the Settler creates a new city and destroys itself.

```
using Godot;
using System;

public partial class Settler : Unit
{

 public Settler()
 {
  unitName = "Settler";
  productionRequired = 100;

  maxHp = 1;
  hp = 1;
  attackVal = 0;

  movePoints = 2;
  maxMovePoints = 2;
 }

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

 public void FoundCity()
 {
  if (map.GetHex(this.coords).ownerCity is null && !City.invalidTiles.ContainsKey(map
.GetHex(this.coords)))
  {
   bool valid = true;
   foreach (Hex h in map.GetSurroundingHexes(this.coords))
   {
    valid = h.ownerCity is null && !City.invalidTiles.ContainsKey(h);
   }

   if (valid)
   {
    map.CreateCity(this.civ, this.coords, $"Settled City {coords.X}");
    this.DestroyUnit();
   }
  }
 }


 // Called every frame. 'delta' is the elapsed time since the previous frame.
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 public override void _Process(double delta)
 {
 }
}
```

TerrainTileUI.cs

Found in folder: /

This code defines a UI component for displaying information about a hex tile in a game, including its terrain type, food production, and production values. It loads images and strings for each terrain type and uses them to populate the UI components when a hex tile is set. The UI is updated dynamically based on the data of the assigned hex tile.

```
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},
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
   { 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 code defines a UIManager class that handles the user interface for a game. It loads and instantiates UI scenes for terrain, cities, and units, and provides methods to show and hide these UI elements. The class also emits a signal to end the current turn and updates the UI accordingly.

```
using Godot;
using System;

public partial class UIManager : Node2D
{

 PackedScene terrainUiScene;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
PackedScene cityUiScene;
PackedScene unitUiScene;

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

[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");
 unitUiScene = ResourceLoader.Load<PackedScene>("UnitUI.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;
 }

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

© 2024 Zenva Pty Ltd. All rights reserved.

```
 }

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

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

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

  cityUi.SetCityUI(c);
 }

 public void SetUnitUI(Unit u)
 {
  HideAllPopups();

  unitUi = unitUiScene.Instantiate() as UnitUI;
  AddChild(unitUi);

  unitUi.SetUnit(u);

 }

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

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

  terrainUi.SetHex(h);


 }
}
```

Unit.cs

Found in folder: /

This code defines a `Unit` class that represents a game unit in a turn-based strategy game. It handles unit movement, combat, and selection, and includes methods for loading unit scenes and textures. The unit's properties, such as its name, production requirements, and movement points, are also defined.

© 2024 Zenva Pty Ltd. All rights reserved.

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

public partial class Unit : Node2D
{

 [Signal]
 public delegate void UnitClickedEventHandler(Unit u);

 public delegate void SelectedUnitDestroyedEventHandler();
 public event SelectedUnitDestroyedEventHandler SelectedUnitDestroyed;

 public static Dictionary<Type, PackedScene> unitSceneResources;
 public static Dictionary<Type, Texture2D> uiImages;

 public static void LoadUnitScenes()
 {
  unitSceneResources = new Dictionary<Type, PackedScene> {
   { typeof(Settler), ResourceLoader.Load<PackedScene>("res://Settler.tscn")},
   { typeof(Warrior), ResourceLoader.Load<PackedScene>("res://Warrior.tscn")}
  };
 }

 public static void LoadTextures()
 {
  uiImages = new Dictionary<Type, Texture2D>
  {
   { typeof(Settler), (Texture2D) ResourceLoader.Load("res://textures/settler_image.p
ng")},
   { typeof(Warrior), (Texture2D) ResourceLoader.Load("res://textures/warrior_image.j
pg")}
  };
 }

 public string unitName = "DEFAULT";

 public int productionRequired;

 public Civilization civ;

 public int maxHp;
 public int hp;

 public int maxMovePoints;
 public int movePoints;

 public int attackVal;


 public Vector2I coords = new Vector2I();

 public bool selected = false;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 public Area2D collider;

 public HexTileMap map;


 public HashSet<TerrainType> impassible = new HashSet<TerrainType>
 {
  TerrainType.WATER,
  TerrainType.SHALLOW_WATER,
  TerrainType.ICE,
  TerrainType.MOUNTAIN
 };

 List<Hex> validMovementHexes = new List<Hex>();

 public static Dictionary<Hex, List<Unit>> unitLocations = new Dictionary<Hex, List<U
nit>>();

 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  collider = GetNode<Area2D>("Sprite2D/Area2D");

  UIManager manager = GetNode<UIManager>("/root/Game/CanvasLayer/UiManager");
  this.UnitClicked += manager.SetUnitUI;

  manager.EndTurn += this.ProcessTurn;
  this.SelectedUnitDestroyed += manager.HideAllPopups;

  map = GetNode<HexTileMap>("/root/Game/HexTileMap");
  this.UnitClicked += map.DeselectCurrentCell;


  map.RightClickOnMap += Move;

  validMovementHexes = CalculateValidAdjacentMovementHexes();

  if (unitLocations.ContainsKey(map.GetHex(this.coords)))
  {
   unitLocations[map.GetHex(this.coords)].Add(this);
  } else {
   unitLocations[map.GetHex(this.coords)] = new List<Unit>{this};
  }

 }

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

 public void ProcessTurn()
 {
  movePoints = maxMovePoints;
 }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 public void CalculateCombat(Unit attacker, Unit defender)
 {
  defender.hp -= attacker.attackVal;
  attacker.hp -= defender.attackVal/2;

  if (defender.hp <= 0)
  {
   defender.DestroyUnit();
  }

  if (attacker.hp <= 0)
  {
   attacker.DestroyUnit();
  }
 }


 public void MoveToHex(Hex h)
 {
  if (!unitLocations.ContainsKey(h) || (unitLocations.ContainsKey(h) && unitLocations
[h].Count == 0))
  {
   unitLocations[map.GetHex(this.coords)].Remove(this);

   Position = map.MapToLocal(h.coordinates);
   coords = h.coordinates;

   if (!unitLocations.ContainsKey(h))
   {
    Unit.unitLocations[h] = new List<Unit>{this};
   } else {
    unitLocations[h].Add(this);
   }

   validMovementHexes = CalculateValidAdjacentMovementHexes();
   movePoints -= 1;

   if (h.isCityCenter && h.ownerCity.civ != this.civ && this is Warrior)
   {
    h.ownerCity.ChangeOwnership(this.civ);
   }

  } else {
   // Combat
   Unit opp = unitLocations[h][0];

   if (opp.civ != this.civ)
   {
    CalculateCombat(this, opp);
   }

  }
 }
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
public void Move(Hex h)
{
 if (selected && movePoints > 0)
 {
  if (validMovementHexes.Contains(h))
  {
   MoveToHex(h);
   EmitSignal(SignalName.UnitClicked, this);
  }
 }
}


public void SetCiv(Civilization civ)
{
 this.civ = civ;
 GetNode<Sprite2D>("Sprite2D").Modulate = civ.territoryColor;
 this.civ.units.Add(this);
}

public void SetSelected()
{
 if (!selected)
 {
  selected = true;

  Sprite2D sprite = GetNode<Sprite2D>("Sprite2D");
  Color c = new Color(sprite.Modulate);
  c.V = c.V - 0.25f;
  sprite.Modulate = c;

  validMovementHexes = CalculateValidAdjacentMovementHexes();
 }
}

public void SetDeselected()
{
 selected = false;

 validMovementHexes.Clear();

 GetNode<Sprite2D>("Sprite2D").Modulate = civ.territoryColor;
}

public List<Hex> CalculateValidAdjacentMovementHexes()
{
 List<Hex> hexes = new List<Hex>();

 hexes.AddRange(map.GetSurroundingHexes(this.coords));
 hexes = hexes.Where(h => !impassible.Contains(h.terrainType)).ToList();

 return hexes;
}

public void DestroyUnit()
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 {
  map.RightClickOnMap -= Move;

  if (selected)
  {
   SelectedUnitDestroyed?.Invoke();
  }

  this.civ.units.Remove(this);
  unitLocations[map.GetHex(this.coords)].Remove(this);

  this.QueueFree();
 }

    public override void _UnhandledInput(InputEvent @event)
    {
        if (@event is InputEventMouseButton mouse && mouse.ButtonMask == MouseButtonM
ask.Left)
  {
   var spaceState = GetWorld2D().DirectSpaceState;
   var point = new PhysicsPointQueryParameters2D();
   point.CollideWithAreas = true;
   point.Position = GetGlobalMousePosition();
   var result = spaceState.IntersectPoint(point);
   if (result.Count > 0 && (Area2D) result[0]["collider"] == collider)
   {
    EmitSignal(SignalName.UnitClicked, this);
    SetSelected();
    GetViewport().SetInputAsHandled();
   } else {
    SetDeselected();
   }
  }
    }

 public void RandomMove()
 {
  Random r = new Random();
  validMovementHexes = CalculateValidAdjacentMovementHexes();
  Hex h = validMovementHexes.ElementAt(r.Next(validMovementHexes.Count));

  MoveToHex(h);
 }

}
```

UnitBuildButton.cs

Found in folder: /

This code defines a custom button class called UnitBuildButton that inherits from Godot's Button class. When the button is pressed, it emits a signal called OnPressed and passes a Unit object associated with the button. The Unit object is stored in the button's `u` property.

© 2024 Zenva Pty Ltd. All rights reserved.

```
using Godot;
using System;

public partial class UnitBuildButton : Button
{

 [Signal]
 public delegate void OnPressedEventHandler(Unit u);

 public Unit u;

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

 public void SendUnitData()
 {
  EmitSignal(SignalName.OnPressed, u);
 }

}
```

UnitUI.cs

Found in folder: /

This code defines a UI component for displaying information about a unit in a game. It retrieves references to UI elements, such as images and labels, and updates them to reflect the unit's properties, like its type, moves, and health. The code also dynamically adds a "Found City" button for settler units, allowing the player to found a city.

```
using Godot;
using System;

public partial class UnitUI : Panel
{

 TextureRect unitImage;
 Label unitType, moves, hp;


 Unit u;


 // Called when the node enters the scene tree for the first time.
 public override void _Ready()
 {
  unitImage = GetNode<TextureRect>("TextureRect");
  unitType = GetNode<Label>("UnitTypeLabel");
  hp = GetNode<Label>("HealthLabel");
  moves = GetNode<Label>("MovesLabel");
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
 }

 public void SetUnit(Unit u)
 {
  this.u = u;

  if (this.u.GetType() == typeof(Settler))
  {
   VBoxContainer actionsContainer = GetNode<VBoxContainer>("VBoxContainer");
   Button foundCityButton = new Button();
   foundCityButton.Text = "Found City";
   actionsContainer.AddChild(foundCityButton);

   Settler settler = this.u as Settler;
   foundCityButton.Pressed += settler.FoundCity;
  }

  Refresh();
 }

 public void Refresh()
 {
  unitImage.Texture = Unit.uiImages[u.GetType()];
  unitType.Text = $"Unit Type: {u.unitName}";
  moves.Text = $"Moves: {u.movePoints}/{u.maxMovePoints}";
  hp.Text = $"HP: {u.hp}/{u.maxHp}";
 }
}
```

Warrior.cs

Found in folder: /

This code defines a Warrior class that inherits from the Unit class. The Warrior class sets its own properties, such as name, production cost, health points, attack value, and movement points, in its constructor.

```
using Godot;
using System;

public partial class Warrior : Unit
{

 public Warrior()
 {
  unitName = "Warrior";
  productionRequired = 50;

  maxHp = 3;
  hp = 3;
  attackVal = 2;

  movePoints = 1;
```

© 2024 Zenva Pty Ltd. All rights reserved.

```
  maxMovePoints = 1;
 }


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

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

© 2024 Zenva Pty Ltd. All rights reserved.