Product Documentation

Team Teal

GitHub Link: https://github.com/Filippo-Santiano/SECW2

December 13, 2024

This document contains the Installation Guide, User Manual & Maintenance Guide.

SECW2: Installation Guide

Chang, Chih-Ting Hulme, Jake Ogilvy, Calum Santiano, Filippo
Seadon, Christopher Singh, Anshu Sue-Ling, James

December 13, 2024

1 System Requirements

Note: The game is available for Windows only.

Minimum: The game is playable with adequate graphics quality using a system matching or exceeding the given specifications.

Recommended: The game runs with full graphics quality and desirable frame rate using a system matching or exceeding the given specifications.

Component	Minimum	Recommended
CPU	**Windows:** x86_32 CPU with SSE2 instructions, any x86_64 CPU, ARMv8 CPU **Example:** Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Snapdragon X Elite	**Windows:** x86_64 CPU with SSE4.2 instructions, with 4 physical cores or more, ARMv8 CPU **Example:** Intel Core i5-6600K, AMD Ryzen 5 1600, Snapdragon X Elite
GPU	**Forward+ rendering:** Integrated graphics with Vulkan 1.0 support **Example:** Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics (Kaveri)	**Forward+ rendering:** Dedicated graphics with Vulkan 1.2 support **Example:** NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)
RAM	2 GB	4 GB
Storage	150 MB (used for executable, project files, and cache)	150 MB (used for executable, project files, and cache)
Operating System	Windows 7/8/8.1 (supported on a best-effort basis)	Windows 10

Table 1: Desktop or Laptop PC Specifications - obtained from Godot docs (Godot, n.d.)

2 Installation

Download the 'Sustain-a-City.zip' file from the Releases tab in the GitHub repository.

Extract the .zip file using Windows Explorer or a tool such as 7-Zip or WinRAR.

Run the 'Sustain-a-City.exe' file.

References

Godot, n.d. System requirements [Online]. Available from: https://docs.godotengine.org/en/stable/about/system_requirements.html [Accessed 2024-12-13].

Sustain-a-city User Manual

December 13, 2024

Contents

List of Figures

1 Welcome to Sustain-a-city

In Sustain-a-city, you are tasked with creating humanity's last city, protected from extreme pollution by a large dome equipped with an air filter. To ensure your city's survival, you must make sustainable decisions that balance pollution levels, economic health, energy demands and population happiness.

You'll have access to various unique assets, each with its own impact on your city. To succeed in creating a sustainable city, you will have to use them wisely!

2 How to Access and Run Sustain-a-city

2.1 System Requirements

Note: The game is available for Windows only.

Minimum: The game is playable with adequate graphics quality using a system matching or exceeding the given specifications.

Recommended: The game runs with full graphics quality and desirable frame rate using a system matching or exceeding the given specifications.

Component	Minimum	Recommended
CPU	**Windows:** x86_32 CPU with SSE2 instructions, any x86_64 CPU, ARMv8 CPU **Example:** Intel Core 2 Duo E8200, AMD Athlon XE BE-2300, Snapdragon X Elite	**Windows:** x86_64 CPU with SSE4.2 instructions, with 4 physical cores or more, ARMv8 CPU **Example:** Intel Core i5-6600K, AMD Ryzen 5 1600, Snapdragon X Elite
GPU	**Forward+ rendering:** Integrated graphics with Vulkan 1.0 support **Example:** Intel HD Graphics 5500 (Broadwell), AMD Radeon R5 Graphics (Kaveri)	**Forward+ rendering:** Dedicated graphics with Vulkan 1.2 support **Example:** NVIDIA GeForce GTX 1050 (Pascal), AMD Radeon RX 460 (GCN 4.0)
RAM	2 GB	4 GB
Storage	150 MB (used for executable, project files, and cache)	150 MB (used for executable, project files, and cache)
Operating System	Windows 7/8/8.1 (supported on a best-effort basis)	Windows 10

Table 1: Desktop or Laptop PC Specifications - obtained from Godot docs (Godot, n.d.)

2.2 Installation Guide

Download the 'Sustain-a-City.zip' file from the Releases tab in the GitHub repository.

Extract the .zip file using Windows Explorer or a tool such as 7-Zip or WinRAR.

Run the 'Sustain-a-City.exe' file.

3 Quick Start Guide

Get started in 9 steps:

1. Select **"Start Game"** from the Main Menu.
2. Read the introductory information to understand your mission.
3. Begin building your city by opening the **building menu** in the bottom-right corner.
4. Select an asset you can afford (your starting money is £175). Since income is essential, a revenue-generating asset is a good first choice.
5. Place the selected asset on the map.
6. Keep building your city by repeating steps 3-5, using different assets to maintain a balance between pollution, economy, electricity and happiness.
7. Over time, some assets may require repairing. Click on any placed asset and, if you have enough money, press the repair button. You will also need to repair your protective dome's air filter, which keeps external pollution out.
8. As your city grows, keep an eye on your pollution, it cannot exceed the pollution limit (shown on the main screen) for more than 3 years. If this happens, you will lose the game.
9. When the game ends, you'll receive a score that reflects the sustainability of your city.

4 Controls

Action	**Windows**
Move pointer	Mouse/Trackpad motion
Select item/activate button/place asset	Left mouse button
Move camera	W,A,S,D
Reset camera	R
Zoom camera	Scroll wheel (up = zoom in, down = zoom out)
Pause Game	ESC

Table 2: Basic Controls

5 Main Menu

- **Start Game** - Start a new game.
- **Exit Game** - Closes Game.

6 Pause Menu

- **Resume Game** - Resumes the game.
- **Extras**

  - **Stats over time**: A graph showing how your city's stats have changed since its creation. This page also includes your current sustainability score.
  - **Data References**: A list of URLs referencing the data used in this game and additional resources that might be of interest if you want to learn more about sustainable practices.
- **Help** - Provides basic information about the game including:
  - The game's backstory
  - How to play the game
  - How to win/lose the game
- **Quit** - Returns to main menu

Figure 1: User interface

Numbers in the list below correspond to UI elements in Figure 1

1. **City Stats:** Current year, pollution / pollution limit, yearly income and money.
2. **Message feed:** Where city updates from your population are shown. These will inform you of the recent trends in your city and which areas you should focus on the most. Making decisions based on the information you receive from these messages is crucial for doing well in the game.
3. **Tooltips:** Displayed when you click on a placed tile. A tooltip holds the repair button and shows you live stats and a fun fact about the selected tile.
4. **Sustainability progress bars:** Show how well your city is performing. The values represent how sustainable your city is, and you should try to keep all bars green for as long as possible.
   a. **Environment**: Keep pollution to a minimum by prioritising clean technologies where possible.
   b. **Happiness**: Keep your citizens happy by placing enough happiness-generating tiles.
   c. **Economy**: Keep your yearly income as high as possible.

d. **Electricity**: Match your city's energy production to its demands so that you are generating a suitable amount of energy. Make sure not to waste energy by generating too much!

5. **Building menu:** Contains all the assets you can place and their prices. To open and close this, press the bottom-right button.
6. **Repair Air Filter Button:** Repairs the air filter in the protective dome that is protecting your city from external pollution.

7 Building Your City

There are four main asset groups that you have access to when building your city:

1. **Revenue-generating assets**
2. **Energy-generating assets**
3. **Happiness-generating assets**
4. **Pollution-reducing assets**

An asset's group determines its primary impact on your city but assets will also affect other metrics. For example, a revenue-generating asset will consume energy and increase pollution to generate income. The extent to which an asset affects other metrics is unique for each asset (See Table 3).

1. **Revenue-generating assets**
    a. **Office** - Generates the most income but requires the most energy, produces the most pollution, and reduces citizens' happiness the most out of the revenue-generating group.
    b. **Wheat Farm** - Produces significantly less income than offices but requires less energy, generates less pollution, and has a neutral effect on citizens' happiness.
    c. **Dairy Farm** - Produces the least income and pollution out of this group, requires the least energy, and increases happiness.
2. **Energy-generating assets**
    a. **Nuclear Power Plant** - Generates the most electricity in the game and produces significantly less pollution than coal power plants. However, nuclear power plants make citizens unhappy and are expensive to construct and maintain.
    b. **Wind Farm** - Produces less energy than the other power plants but produces significantly less pollution and is not strongly disliked by citizens.
    c. **Coal Power Plant** - Produces a significant amount of energy at a much lower price. However, coal power plants produce huge quantities of pollution and make citizens extremely unhappy.
3. **Happiness-generating assets**
    a. **Stadium** - Makes citizens happy and provides income at the cost of high construction costs and energy demands.
    b. **Leisure Centre** - Does not please the population as much or generate as much income as a stadium, but requires less energy and is cheaper to build and maintain.
    c. **Park** - Provides the smallest increase in citizen happiness but requires the least energy and is the cheapest asset to construct.
4. **Pollution-reducing assets**

a. **Rubber Forest** - This group's second-highest pollution-reducing asset. It produces more income than forests and orange forests but less than the others.

b. **Cocoa Forest** - Removes more pollution than the rubber forest but produces more income.

c. **Forest** - Removes the most pollution and increases happiness the most, but has no income.

d. **Orange Forest** - A cheaper alternative that removes less pollution and produces less income.

e. **Palm Forest** - Removes the least pollution but produces the second-highest income.

8 Playing the Game

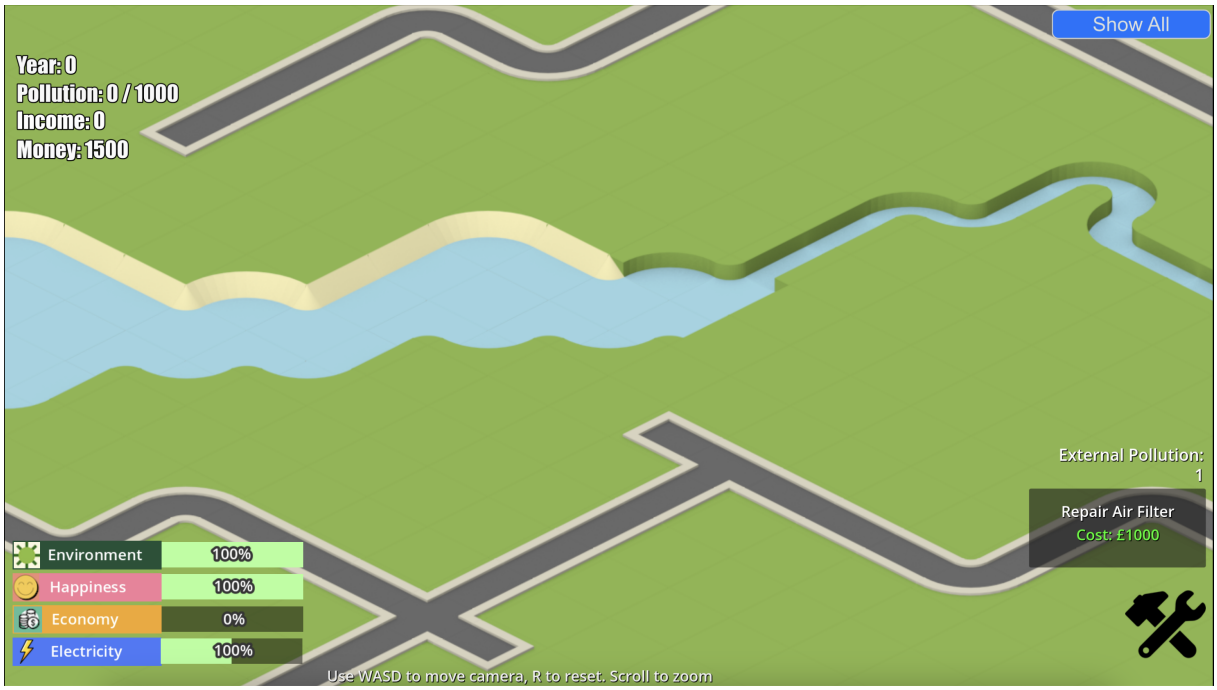

Figure 2: New game

When you start a new game, you'll be presented with an empty map (Figure 2). To help you get started, you have £175 in your account—enough to purchase a few of the more affordable assets. Since your city will need income, selecting a revenue-generating asset is a good place to begin. As you place assets, they will begin to affect your city's pollution levels, economy, energy demands, and population happiness.

To achieve a high sustainability score, you must keep your city's pollution below the limit and all four metrics within their respective green zones, as indicated by the health bars. This requires strategic city planning, constant monitoring of the city's health, and use of all the information available to you. The most influential sources of information are the message feed from your population, the progress bars, and your stats over time, which can be accessed in the pause menu.

Tile Attributes Table

Group	Name	Yearly Pollution	Income	Electricity Req.	Electricity Gen.	+Happiness	-Happiness	Cost
Revenue-Generating Assets	Office	25	350	2.5	0	0	8	380
	Wheat Farm	8	60	0.2	0	0	1	100
	Dairy Farm	2	40	0.2	0	0	2	50
Energy-Generating Assets	Nuclear Power Plant	20	-10	5	25	0	50	4000
	Wind Farm	0	-5	1	7	10	2	2500
	Coal Power Plant	60	-50	5	10	0	75	1000
Happiness-Generating Assets	Stadium	50	500	12	0	200	5	2500
	Leisure Centre	3	30	1	0	25	5	400
	Park	-5	-2	0	0	5	0	40
Pollution-Reducing Assets	Rubber Forest	-25	6	0	0	5	0	120
	Cocoa Forest	-22	8	0	0	5	0	110
	Forest	-30	0	0	0	7	0	100
	Orange Forest	-20	5	0	0	5	0	90
	Palm Forest	-15	7	0	0	2	0	80

Table 3: Sustain-A-City Tile Attributes

References

Godot, n.d. System requirements [Online]. Available from: https://docs.godotengine.org/en/stable/about/system_requirements.html [Accessed 2024-12-13].

Maintenance Guide

Chang, Chih-Ting Hulme, Jake Ogilvy, Calum Santiano, Filippo
Seadon, Christopher Singh, Anshu Sue-Ling, James

December 13, 2024

Contents

List of Figures

Listings

1 Introduction

This document seeks to provide the necessary information for new developers to maintain, modify and expand on the serious game *Sustain-A-City*. This game is built using the game engine Godot v4.3 which can be downloaded from this link: Download Godot (Windows). Some pre-existing knowledge of Godot Engine is assumed throughout this maintenance guide.

The purpose of this game is to educate the target audience of young teens about how to develop businesses sustainably, and to understand the carbon footprint that these businesses have. The game has 4 different variables that the player has to balance: environment, economy, electricity and happiness. The player manages these variables by building and repairing offices, power plants, trees etc. Each of these variables contributes to the final score, there is no end to the game, rather the game gets progressively more difficult as time progresses until the player eventually loses. The primary lose condition is that the player produces too much pollution and is over the threshold for 3 consecutive years. The secondary lose condition is that the player's income and money equal zero and are therefore not able to progress any further. The player is guided in-game by the Oracle, which provides relevant commentary on your game as your statistics change.

To run and develop this game, you must first have Godot v4.3 installed with a computer that meets system requirements listed here: Godot System Requirements. It is also recommended that developers use a version control system such as Git.

2 Project Structure

2.1 Getting Started

Git Large File Storage

To begin development, you will need to install Git Large File Storage (LFS) which replaces files with text pointers inside Git. This was used for the assets in the game.

Cloning The Project

Before making any modifications to the game, you will need to ensure that you have the latest versions of the project files downloaded on your machine.

To obtain the latest version, clone the 'main' branch from the GitHub repository here.

File Structure

Once cloned, you will be able to access the game's main scripts, assets and Godot project files required to open the game within the Godot editor and start making modifications.

Here is a brief overview of the project structure:

[Project Folder]

/scenes – Contains all .tcsn files containing scenes within the game: for instance, 'MainMenu.tscn' can be loaded in order to edit the contents of the initial menu (title) screen.

/scripts – Contains all of the scripts (.gd files) used to define variables and functions within the game's nodes. This is where the majority of the game-specific code is located. All scripts are written in GDScript.

/assets – Contains the game's assets. As it stands, this contains only .png image files, which are used both for any icons shown within the game and for each tile that makes up the game's map.

project.godot – The main project file. This is the file that should be imported into the Godot 'Project Manager' to load and edit the contents of the game.

Importing the Game into Godot With Godot installed, upon loading you should be presented with the Project Manager screen. Select 'Import' and navigate to the cloned project folder. Choose 'Select Current Folder', and if all is well you should be shown a 'Valid project found at path' message. Then choose 'Import & Edit' to import the project into the Project Manager and load it for editing.

2.2 Key Scripts and Scenes

A project in Godot is made up of individual 'scenes' that contain different elements of the game. This section is an overview of these scenes, explaining their functions and how they can be accessed and edited.

2.2.1 Main (main.tscn)

This scene contains the main gameplay loop, including the base map, camera, and UI elements. Running the scene loads the playable, main game screen, shown in Figure 1.

Figure 1: Main scene

Nodes The Main scene contains nodes for each of the main game functions, some of which are scenes in themselves. These nodes include:

Tiles Where the main 'TileMapLayer' nodes are located, separated into two layers, 'Layer0' and 'Layer1'. Layer0 contains the ground tiles, while Layer1 contains anything placed above ground, including tiles placed by the player, like buildings, trees and power plants. Layer 0 is fixed in-game, while layer 1 is able to be interacted with by the player. This node has the 'tiles.gd' script attached, which is used for defining all tile data as well as for functions like tile placement and showing the ToolTipBox when the player clicks on a tile.

PlayerController A basic 'Node2D' node that contains a script which defines how the player can interact with the game (placing tiles). This also contains another 'TileMapLayer' node, named

'HoverTiles' which is used to show the tile that the player has selected, when in **Build Mode**, before they have chosen to place it.

WorldBounds An 'Area2D' node that defines the area in which the camera is allowed to move.

ToolTipBox An imported scene, with a basic 'Control' (UI) node as its parent, which displays a tooltip box when a player clicks on a tile outside of **Build Mode.**

MainCamera The game's camera. A Camera2D node with a script that defines the camera's controls as well as functions like zooming in and out.

UI A 'CanvasLayer' node containing all of the game's UI elements. Putting UI nodes within the CanvasLayer ensures that they stay in the same position on the screen at all times, regardless of the camera position.

YearsTimer A 'Timer' node used for keeping track of the years within the game. The attached script also contains some calculations for the changing of each tile's attributes which happens every year.

2.2.2 Building Menu (building_menu.tscn)

The Building Menu scene contains nodes that formulate the building menu, which allows the player to choose which tile they would like to build.

Nodes

The important nodes here are of type 'Button', and these are used in two different ways:

BuildButton Opens and closes the build menu. This node does not have a script attached, and the function here is controlled by its 'button pressed' signal which is connected to the main building_menu.gd script.

(Tile)Item Button for the individual tile that a user that purchase and place on the map. These buttons have a 'BuildingMenuItem.gd' script with an ID variable, allowing you to adjust which tile is purchased by that button through the **Inspector**, as well as functionality that automatically sets the cost shown in the menu to the correct value.

2.2.3 Twitter Feed (twitter_feed.tscn)

The Twitter Feed scene contains nodes that formulate the Oracle menu, which informs the user about the state of the statistics (pollution, happiness etc.) at the current point in time within the game.

2.2.4 ToolTip Box (tool_tip_box.tscn)

See Tool Tip.

2.2.5 Progress Bars Ui (ProgressBarsUi.tscn)

The Progress Bars scene contains UI nodes that create the progress bars, shown on the bottom left of the main screen. These show the player details about the current total pollution, happiness, economy and electricity values each expressed as a percentage.

This scene is constructed mainly of Labels and other UI elements, with a script 'ProgressBarsUI.gd' that calculates and sets the displayed percentages.

2.2.6 Pause Menu (pause_menu.tscn)

The game's pause menu, accessed by pressing 'Esc', is defined here. Whilst open, the rest of the game is paused entirely.

Nodes

The pause menu scene is again comprised mostly of Labels and Buttons. The most notable nodes here are:

PanelContainer Contains the main Labels and Buttons that comprise the pause menu.

ExtrasMenu A simple 'Node2D' node that contains the nested Extras menu, that can be accessed via the Extras button within the pause menu.

2.2.7 End Screen (end_screen.tscn)

The End Screen scene displays the players final score, and plots each variable, money, pollution happiness and electricity, over time. This scene is displayed when the lose conditions are met.

2.2.8 global.gd

In addition to these key scenes, much of the data handling happens in a global script, 'global.gd'. This script runs as an AutoLoad and can be accessed by any node within the project, and as such is useful for storing global variables and recurring functions.

Some of these variables and functions include:

Variables

Money (int) The amount of money the player has to spend on buildings, repairs, etc.

Pollution (int) The overall level of pollution in the world's environment.

Electricity (int) The amount of electricity currently being generated by the player

currentYear (float) The current year within the game. This value is expressed as a floating point variable to allow us to track exactly where we are within the year and measure, for example, when something like half a year has passed.

ExternalPollution (int) The amount of pollution coming in from outside the playable area. This rises every year regardless of the player's actions, but can be reduced by repairing the Air Filter.

Constants

AIR_FILTER_REPAIR_COST (int) The cost, in the same unit as the player's money, of repairing the air filter in the game.

AIR_FILTER_REPAIR_MULTIPLIER (int) A floating point value that determines how much External Pollution should be reduced upon repairing the air filter.

Functions

update_tile_attributes() Used to apply the multipliers to each attribute of a tile, in order to slowly reduce income or increase pollution. This currently runs each year, called by YearsTimer, iterating through each of the placed tile's attributes and updating them.

updateExternalPollution() Used to update the **External Pollution** each year by a (float) multiplier **totalExternalPollutionMultiplier** declared within the function.

updateData(x,y) Called when a tile is placed for the first time, this updates the global pollution and income values with the initial values set within 'Initial_Tile_Attributes'.

collect_yearly_data() Appends all of the current year's values (pollution, electricity etc.) to an array for use in graphing the player's progress. Called each year in YearsTimer.

calculate_final_score() Calculates and returns a score, assessing the player's level of success, using the values stored using the previous function. This is shown on the **Current Stats** and ending screens.

repairAirFilter() Multiplies the current **External Pollution** by the AIR_FILTER_REPAIR_COST multiplier.

chargeMoney(amount) Used to check whether the player has a given amount of money and, if so, removes that amount. Otherwise this displays a 'not enough money!' animation. This is used whenever the player needs to make a purchase.

3 Asset Management

3.1 New asset

To load a new tile into the game:

1. Design/Download isometric tile
2. Copy asset into Assets folder within the project
3. Reimport assets within Godot
4. Select the tileset.tres file to open the tile set
5. Select the newly imported tile
6. Adjust the dimensions of the tile to match the in-game dimensions of 128x64
7. Adjust the xyz axis to fit with the map

More information on importing static assets into Godot can be found here: Import Process - Godot Documentation

3.2 Base Map

To adjust the base map:

1. Select Layer 0 from the node Tiles
2. Select the tile you would like to add to the base layer
3. Using the Godot tools, "paint" the new tiles onto the map

Figure 2 displays an example of adding in Grass tiles onto the base map.

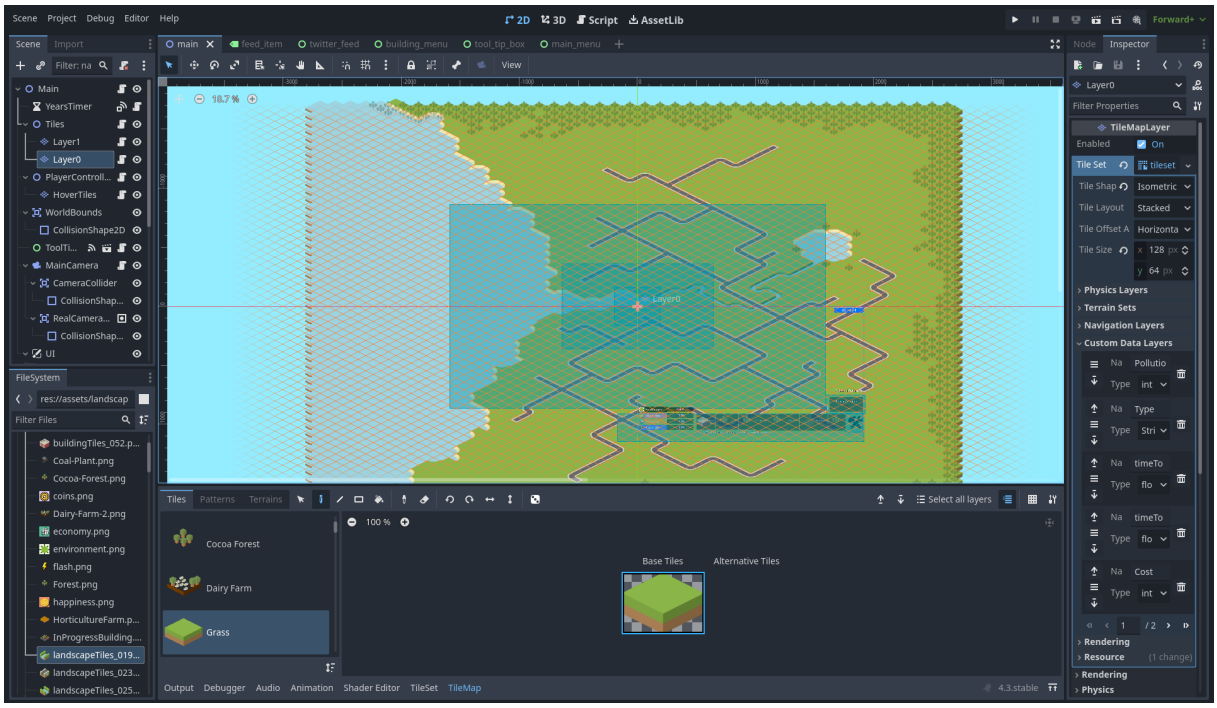

Figure 2: Add tile to layer 0.

3.3 Custom values

Each asset has a unique set of initial values which are defined in the tiles.gd script. The dictionary Initial_Tile_Attributes contains the 7 different unique attributes that each tile can currently hold. Listing 1 shows an example of the dynamic variables for the Office building.

```
const Initial_Tile_Attributes = {
  1: {  # Office
    "name" : "Office",
    "yearly_pollution": 50,
    "income": 350,
    "electricityRequired": 2.5,
    "electricityGenerated": 0,
    "positiveHappiness": 0,
    "negativeHappiness": 8
  },
    ...
}
```

Listing 1: Example set of initial attributes

To set the values for a new building, a new dictionary needs to be created with the game ID value as the identifier within the Initial_Tile_Attributes dictionary as in Listing 1. From here, the new values for each attribute can be defined and modified. To modify any of the current attributes of tiles already in the game, you must first locate that tile by the ID value or name of the tile, and then adjust the relevant value from that tile's specific dictionary.

Each asset also has fixed variables which are assigned as custom variables to the asset directly. To adjust these fixed variables:

1. Select the tileset.tres file to open the tile set
2. Select the relevant tile from the tileset
3. Adjust the custom data for that tile

An example of the custom data values for the Office building are shown in Figure 3.

Fixed Custom Variables

- **Pollution:** This is the pollution cost in building that tile
- **Time To Build:** This is how many years it takes to build
- **Cost:** This is the in-game cost of the tile
- **Can Sell:** This indicates whether the player should be able to sell the tile in the game

Figure 3: Modify Custom Data

3.4 Multipliers

Each variable for each asset has a unique multiplier which is defined in the tiles.gd script. The dictionary Tile_Multipliers contains the 7 different unique multipliers for each attributes for each tile. Listing 2 shows an example of the dynamic variables for the Office building.

```
const Tile_Multipliers = {
  1: { #Office
      "yearly_pollution": 0.03, # +3% pollution
      "income": -0.05,
      "electricityRequired": 0.05,
      "electricityGenerated": 0,
      "positiveHappiness": 0,
      "negativeHappiness": 0.05
  },
     ...
}
```

Listing 2: Example set of multipliers

To set the multipliers for a new building, a new dictionary needs to be created with the game ID value as the label within the Tile_Multipliers dictionary as in Listing 2. From here, the new multipliers for each attribute can be defined and modified. To modify any of the current multipliers of tiles already in the game, you must first locate that tile by the ID value or name of the tile, and then adjust the relevant value from that tile's specific dictionary.

4 Adjusting In-Game Functions

4.1 Oracle

The Oracle in Sustain-A-City provides the player with relevant feedback, warnings and encouragement. This is to help guide the player towards a better score and more sustainable city. The Oracle is implemented as a twitter feed that appears at the top right of the screen, and displays a message when a certain game condition is met. This message is a random selection out of a dictionary of messages that are relevant to that condition being met.

The Oracle is implemented within the twitter_feed.tscn scene. The conditions.gd script contains the boolean functions for each different event that the Oracle will provide comment on. An example of the conditions for the 3 different pollution criteria is shown in Listing 3.

```
### Functions for Pollution ###
func poll_pos_to_neg() -> bool:
  return Global.Pollution < 0

func poll_half_thresh() -> bool:
  return Global.Pollution * 2 > Global.PollutionThreshold

func poll_greater_thresh() -> bool:
  return Global.Pollution > Global.PollutionThreshold
```

Listing 3: Example of conditions for Oracle

The other script in the scene is the twitter_feed.gd script, which contains the rest of the implementation for the Oracle. Within the _ready() function the conditions.gd script is linked to twitter_feed.gd script with a new dictionary called conditions within the as shown in Listing 4.

```
func _ready():
  check = Conditions.new()
  add_child(check)
  # Add the different conditions
  conditions = {
    "Pollution": {
      "pos_to_neg": check.poll_pos_to_neg,
      "half_thresh": check.poll_half_thresh,
      "poll_greater_thresh": check.poll_greater_thresh,
    },
    ...
```

Listing 4: Example of linking to the conditions for Oracle

This dictionary is then used to indicate what message should be displayed from the dictionary messages which is also contained within the _ready() function, as shown in Listing 5.

```
  # Messages to show after conditions match
  messages = {
    "Pollution": {
      "pos_to_neg": [
        "Pollution has dropped below zero! Great job!",
        "The city celebrates achieving a carbon-neutral status!"
      ],
      "half_thresh": [
        "Pollution is halfway to the threshold! Be cautious!",
        "Environmental concerns rise as pollution nears dangerous levels."
      ],
      "poll_greater_thresh": [
        "Pollution is greater than threshold please look out for the city.",
        "The city is dying! Please reduce the pollution."
      ]
    },
    ...
```

Listing 5: Example of messages for different conditions for Oracle

To add a new condition, you must first define a new boolean function within the conditions.gd script, then link it to the twitter_feed.gd script in the conditions dictionary and finally link that and define the relevant messages within the messages dictionary.

The system keeps a record of what message has been displayed for each condition, such that the next time that condition is triggered, it will ignore that message and choose another at random. In

the example shown in Listing 5 there are only two messages per condition, but the developer can additional messages by appending the array for a specific condition.

The system also keeps a record of what condition has been triggered such that it will only display a message from that condition again, if a different condition from that same metric has been triggered. For the Pollution metric, as shown in Listings 3, 4 & 5, if the Oracle has already indicated to the player that they are halfway to the pollution threshold, it will only display a half_thresh message again once they have either triggered the poll_greater_thresh or poss_to_neg condition. This avoids overloading the player with messages that they have already been told about.

A timer node, MessageDelay, is included in this scene to restrict how fast messages are displayed to the player. When a new condition is triggered and a message is to be displayed, it is added to a queue. After a period of time defined by the MessageDelay node, the messages which are shown to the player are updated from the queue. If there are more than 5 messages already being displayed, then the oldest message will be removed and a new message from the queue will be added at the top, otherwise a new message is just added.

A separate function to display messages about whether the player is over the threshold and only has a certain number of years to live is defined in the age_over() function. This allows it to have top priority in the queue and be displayed to the player when it is triggered rather than have to wait for the queue to clear out.

A button ShowAllButton is also included in this scene, such that the player can toggle between seeing 2 messages or all the messsages (which is limited to 5).

4.2 Build Function

Adding a New Tile to the Building Menu

See Asset Management for details on how to add a new tile to the tileset and set any applicable initial values such as cost, time to build and properties like pollution and electricity consumption.

To enable a player to buy a tile from the building menu, it must first be added as an item in the menu via the Building Menu scene. These have been implemented using Button nodes.

Each item needs to be attached to the 'BuildingMenuItem.gd' script, which then gives them an ID and allows connection to the main BuildingMenu node, as shown in Figure 4.

The easiest way to add a new item is to select one of the established button items, right click, and select Duplicate to create an identical copy. You can then adjust the displayed text, icon and the ID of the tile to be purchased as shown in Figure. Occasionally the text labels get slightly misplaced when copying in this way, so ensure that the position of the 'Cost' label within the new button's box looks correct.

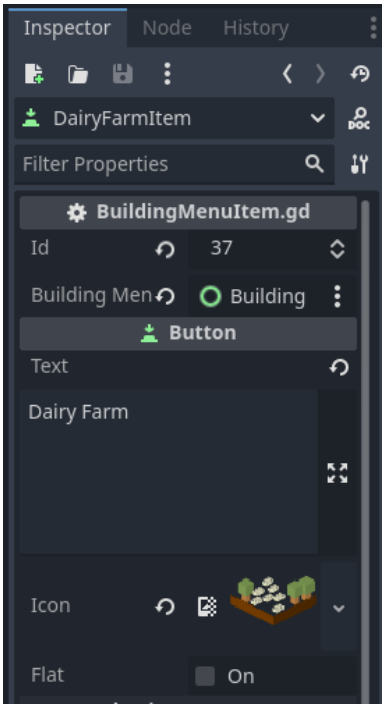

Figure 4: The Inspector showing the properties of an item in the build menu, including its ID and connection to the 'BuildingMenu' node.

4.3 Tool Tip

Adding New Stats

The tooltip shows the player the effect a tile is having on the game world. Currently, it shows its effects on the economy, environment, electricity and happiness, but this could be adapted to reflect different variables or to express these values in a different way.

The implementation of the tooltip panel can be found within the ToolTipBox scene, and its associated tool_tip_box.gd script. The properties shown are Labels, organised within VBox and HBox Containers, and their displayed text can be altered using the 'set_text' function. This function uses the 'find_child("name")' godot function to find a label with a given name. This means, for example, you could set the 'Environment' label to a different value by calling

set_text([new value],"Environment").

Currently, this 'set_text' function is called within the 'tiles.gd' script when the popup is shown.

As such, in the event that a new variable needs to be added, you would add a new label and use that label's name in the set_text function. It is worth noting that due to the nature of this approach, each label added must have a unique name.

Adding/Modifying Fun Facts

The tooltip panel also shows 'Fun Facts' about the selected tile.

The fun facts are stored within the 'tiles.gd' script, in a function called 'generate_fun_fact(asset name)'. Each tile can have multiple facts defined, and in this case, one of these is picked at random and returned. Note the asset_name here refers to the name of a tile as established in Initial_Tile_Attributes.

This function is called when the tooltip box is shown, in the update_box(tile,type) function in 'tiles.gd'.

Updating the fun fact label itself uses the same method as in [Adding New Stats], taking advantage of the set_text function (as in: set_text(funFact,"Fun Fact")).

Repair Button

The **Repair Button** is also shown inside the tooltip panel and allows the player to repair the selected building, resetting its attributes back to their initial values as defined in the **Initial_Tile_Attributes** dictionary.

This functionality is implemented within the 'tiles.gd' script. Its basic logic is as follows:

When the repair button is pressed, the 'get_repair_data()' function is called. This accesses the selected tile's current and initial attributes, calculates the difference, and displays this in the tooltip panel using the 'set_repair' function. The repair button now becomes a 'confirm' button, and the cost of the repair (defined by variable **RepairCost**) is displayed in its box. The **RepairCost** is set to half of the original cost of the tile.

Pressing the 'confirm' button then enacts the repair, calling the 'repair_tile' function. This sets the current, to-be repaired tile back to its initial attributes as shown in Figure 5.

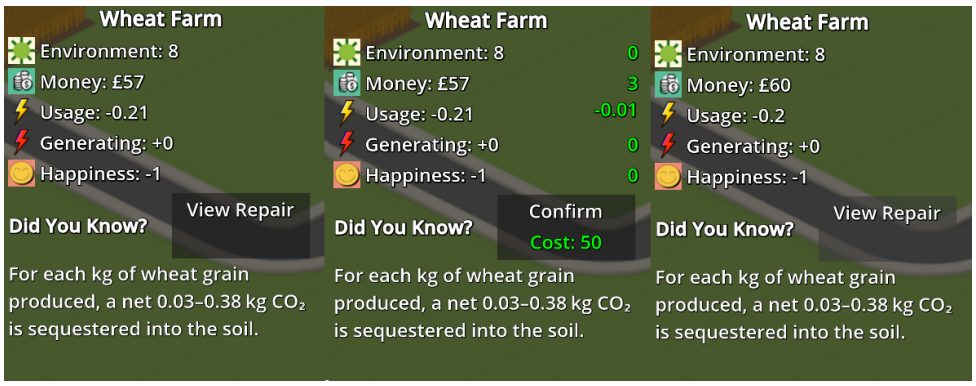

Figure 5: Stages of repairing a Wheat Farm tile

4.4 Pause Menu

The pause menu consists of one main menu containing buttons that can resume, exit the game, or open further sub-menus. The pause menu is implemented within the PauseMenu scene, with further menu scenes HelpMenu, ExtrasMenu, DataReferences and CurrentStats as its children.

The buttons utilise signal connections to perform actions and open and close menus. For example, clicking on the 'Help' button sends a signal to the Pause Menu (pause_menu.gd) script, via the HelpMenu scene, which hides the main pause menu and shows the new pause menu. The same (but reversed) applies to the 'Back' button once the sub-menu is open.

The pause_menu script also defines what button opens the pause menu in-game ('Esc' by default). Additionally, on opening the menu, the script pauses the game by manipulating the variable get_tree.paused.

More information on Godot's pause functionality can be found here: Pausing games and process mode - Godot Documentation

Listing 6 shows an example of this implementation, taken from the pause_menu.gd script.

```
1 func _on_help_pressed() -> void:
2   panel_container.visible = false
3   help_menu.show_menu()  # Show the Extras menu
4
5 func _on_exit_submenu() -> void:
6   # Called when exiting any submenu
7   extras_menu.hide_menu()  # Hide Extras menu
8   help_menu.hide_menu()  # Hide Extras menu
9   panel_container.visible = true  # Show the main pause menu
```

Listing 6: Example of menu implementation

4.5 Air Filter

The 'Air Filter' is a system within the game that allows the player to reduce the amount of pollution coming in externally. This pollution is stored in global.gd, with the ExternalPollution variable, and updates every year by the constant AIR_FILTER_MULTIPLIER.

The game features a 'Repair Air Filter' button, above the Building Menu, that allows the player to reduce the ExternalPollution by the AIR_FILTER_REPAIR_MULTIPLIER, which costs them an amount of money as defined by the AIR_FILTER_REPAIR_COST.

The air filter repair button is a simple Button node that takes advantage of the button_pressed signal to perform its function. The button is a child of the Main scene and is named AirFilterButton.

4.6 Time

Time is tracked using YearsTimer, a node of type Timer that is set to emit a signal every second.

YearsTimer stores the current year in a (float) variable Years, as well as a floating point value yearsPerMinute. This determines the speed at which time flows in the game; every second, the yearsPerMinute value is divided by 60 and then added to the Years value. Once the Years value has increased by a whole number, the Global currentYear value is updated.

The yearsPerMinute variable is shown in the **Inspector**, so quick adjustments of the speed of time flow can be made simply by changing this number outside of the script as shown in Figure 6.

Figure 6: YearsTimer's settings in the **Inspector**, including adjustable yearsPerMinute

5 Score and Stats

5.1 Calculations

When a tile is placed, provided the player has enough money, the attributes, multipliers, time it was placed and the asset ID is appended to the dictionary tile_data, which is located in the global.gd script.

Each year, within the years_timer.gd script, the relevant multiplier for each variable for each tile is applied by calling the function update_tile_attributes from global.gd, updating all the values within the dictionary. For example, tiles like the Office would need maintenance work done and so it's ability to produce income would be reduced, or for a forest, would increase the amount of pollution it removes from the environment. The following variables are then updated with the following calculations each year.

Happiness is calculated by dividing the total positive happiness values by the total negative happiness values to create a percentage. If total positive happiness is greater than the total negative, then the final happiness score it just set at 100%. In the ProgressBarsUI.gd script, this value is then displayed as a bar showing the percentage.

Environment is calculated in the ProgressBarsUI.gd script by subtracting pollution percentage from 100. The pollution percentage is the current level of pollution, divided by the pollution threshold, multiplied by 100. The overall pollution score, as defined in the global.gd script, is a running total of the fixed cost of pollution in building each tile and the pollution produced yearly from each tile. Pollution threshold is calculated in the years_timer.gd script, as 1000 multiplied by the happiness of the population for that year. If the population is unhappy they will tolerate less pollution and it will be easier to lose the game.

Electricity is split into attributes electricity generated and electricity required. These individual attributes are re-calculated each year by summing all the tiles together. The final electricity value is calculated in the ProgressBarsUI.gd script by dividing the total electricity generated by the total electricity required. This is then shown in the bar which is centred around 100%.

Economy is calculated by the current total revenue divided by total possible revenue generation from the tiles placed (i.e. their initial values). The current total revenue, or income as is shown in the game, is calculated in the years_timer.gd script, by the sum of all current income values from the different tiles, which, if electricity is less than 100%, is then multiplied by electricity. For example, if electricity is at 70% and the sum of all income attributes for that year was 100, the actual income would only be 70. The final calculation to determine the economy score is performed and displayed in the ProgressBarsUI.gd script.

Player Score is calculated within the global.gd script in the calculate_final_score function. Each year, the total values calculated for each attribute from the tile_data dictionary are appended into the yearly_data dictionary. The calculate_final_score function then goes through that dictionary,

applies weights and sums the total score. Given happiness is a decimal, these values are multiplied by 10. Positive pollution scores are a negative and so these values are multiplied by -0.1. Money (which is all the income received minus all the money spent) has no weight.

5.2 Win/Lose

The win objective of the game is to achieve the highest score possible. The lose conditions of the game are triggered when the player has a pollution score above the pollution threshold for more than 3 consecutive years or when the player does not have any money nor the means to produce any more.

Both lose conditions are contained within the years_timer.gd script in the yearPassed function. Each year that passes, the function checks the global variables of Money, Income, Pollution and Pollution Threshold to see if these conditions are true. If Money == 0 and Income < 25, or if Pollution > Pollution Threshold and the Years_Over counter > 2 then the end scene you_lose.tscn is played. If Pollution > Pollution Threshold but the Years_Over counter < 2, then the Years_Over counter increases by 1, and if Pollution < Pollution Threshold then the Years_Over counter is reset back to 0.

5.3 Graphs

The end screen scene, end_screen.tscn, is attached to the script end_screen.gd. In this, the logic to create the graph to display the historical data of the city is executed. This is done as follows:

- In global.gd, the collect_yearly_data() function appends the four health metric values (money, pollution, electricity and happiness) to individual arrays : money_data[], pollution_data[], electricity_data[], happiness_data[],
- In years_timer.gd, the function year_passed() is executed when the year increases by 1. When this happens, the collect_yearly_data() function is called, and thus the health metric values for that year are appended into the appropriate arrays
- In end_screen.gd, the arrays which store the yearly data for for the four health metrics are accessed from the global.gd script. The function store_normalised_data() iterates through the four arrays, calculates the minimum and maximum value for each, and uses this to normalise the data (i.e., each value will be between 0 and 1) and appends these to the four arrays: normalised_money[], normalised_pollution[], normalised_electricity[] and normalised_happiness[]
- In end_screen.gd, the _draw() function calculates where each value for a health metric will be on the screen relative to the axes of the graph. These are stored in the arrays: points_money[], points_pollution[], points_electricity[] and points_happiness[]. Then these points are plotted with the built-in Godot function, draw_line(), which draws a line between the position of each point. The x and y axis are also drawn in this function. The line chart is shown by Figure 7.

How to change visuals of the graph:

- **Colour of lines:** In the built-in function, draw_line(), the colour of the line is set with RGB values. Simply changing the RGB value to the desired colour for the metric will change its colour on the graph. For example, to make a line red, it would be a case of inputting Color(1, 0, 0) into draw_line()
- **Thickness of lines:** Following setting the colour, there is a number which sets the

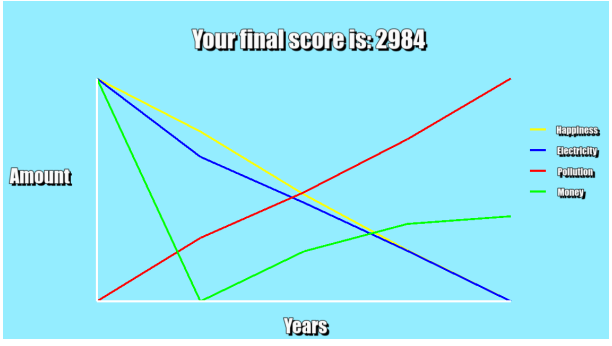

Figure 7: Line chart

thickness of the line. The thickness upon downloading the game is set at 6. This can be changed by typing in the desired thickness.

How to add data to the graph:
If a new asset was introduced to the game which had a health metric other than the four current metrics, the steps to add this into the graph would be:

- Create a new array in global.gd for the metric and populate it in the collect_yearly_data() function
- In end_screen.gd, calculate the location of the points and populate a new array using the same method as with the other health metrics. Then, add a new draw_line() command for the new health metric and set the colour to be different to the existing metrics.
- Ensure a legend item is added for the new metric added to the graph.

6 UI

6.1 Camera

The **MainCamera** node contains the basic camera implementation, which allows xy movement of the camera around the world, within a set area, and zooming the camera in and out.

Movement The camera moves when the player presses one of the directional buttons (set by default to W, A, S and D). These keys set a movement direction, stored in Vector2 Inputs, and then this direction is multiplied by the constant MOVE_SPEED and is added to the camera's position whenever the buttons are pressed.

The camera's MOVE_SPEED can be adjusted at the top of the 'main_camera.gd' script and by default is set to 20.

Bounds

The camera's position is limited by a boundary set by WorldBounds, a node of type Area2D that contains a rectangular **CollisionShape2D**.

This is implemented through the Area2D CameraCollider, which is a child of the main camera node. When the position is manipulated in the camera script, as explained in Movement, the script first checks whether the camera collider is still within the WorldBounds area. If it isn't, rather than continuing to move, the camera is instead reset to its previous position (the xy position it was at on the last frame, set by Vector2 prevPos each frame the camera is moved).

Zoom

The camera's zoom property scrolls between three steps, defined by the (float) ZOOM_STEPS constant. These control how far the camera is zoomed in, with the default steps being 1.0 (largest field of view), 1.5, and 2.0 (smallest field of view).

The allInputs() function scrolls between these values using button inputs, moving up a step if the player scrolls up (and if not already on the max step) and down one on scroll down (if not already on the minimum).

6.2 Start and End Screen

Main Menu & Welcome Screens

The game opens with a start screen that shows the game's title and allows them to start the game.

Choosing 'Start Game' shows a series of 'Welcome' screens, explaining the basic plot and objectives.

All of these screens are put together using basic Labels and Buttons, with connections between them similar to how the Pause Menu is implemented.

For more information about title screens and buttons in Godot, see Design a title screen - Godot Documentation

6.3 Controls

The controls of the game are defined within various nodes' scripts, depending on the properties being controlled.

These nodes and scripts are:

PlayerController Contains a 'build_inputs()' function, called by the _physics_process() native method that **places a tile** when the **left mouse button** is pressed (when in build mode).

Tiles Contains an Input function '_input(event)' that calls the show_popup() function, to show the Tooltip panel, when the **left mouse button** is pressed.

MainCamera Contains a getInputs() function that defines camera movement as the W,S,A and D keys (up, down, left and right), scrolling up or down to zoom in and out, and R to reset its position back to the starting point (reset).

All of these inputs can be adjusted by changing the string input referenced in

'Input.is_action_pressed(string)'

to any input defined in the project's **Input Map**. More information on the **Input Map** in Godot can be found here: Input Examples: InputMap - Godot Documentation.

7 Building the Game

To create an executable build of the game, choose the 'Project' drop-down in the Godot Engine toolbar and select 'Export...'. You will be presented with the export menu, in which you can select the platforms you would like to build for. Currently, the project is designed to be built for Windows only, as the game was not designed for mobile or web platforms. Linux builds do compile correctly, but running these has not been tested. Building a signed executable for Mac OS, to ensure the system allows the game to run, requires some extra steps and as such is untested.

When you have added the 'Windows Desktop' preset to the list, ensure that you have set a valid export path (the default is /build in the project directory), you have ticked the 'Runnable' box, and you have set an application name and icon.

At this point, you can select 'Export All...' and export a Debug or Release build of the game.

More information about exporting projects in Godot can be found here: Exporting projects - Godot Documentation

8 Glossary

CollisionShape2D A node used for defining an area that can detect collisions with other objects.

Input Map Found in Project Settings, used to define what specific input events (key presses, mouse clicks etc.) correspond to actions defined in the game's code.

Inspector A panel in the rightmost section of the Godot GUI, used for modifying properties of the selected node. Variables can be 'exported' here for quick adjustments.