HIGHWATER

Design Specification Document

v 1.0

GLASS PRISON GAMES

Prepared by

Colan Biemer,
Allison Frauenpreis,
Gabrielle Getz,
Jasmine Marcial,
Laura Mo,
Shreya Patel

Advisor

Jeff Salvage

Stakeholder

Frank Lee

Table of Contents

Document History

v 0.1	Created document template, added project overview	Gabrielle Getz	4/10/17
v 0.2	Added design goals, architecture overview and modules	Allison Frauenpreis, Gabrielle Getz, Jasmine Marcial Laura Mo	4/15/17
v 0.3	Added flocking and weather algorithms	Colan Biemer	4/17/17
v 0.4	Added Technologies Used section, Project Overview section, and Module overview sections.	Gabrielle Getz	4/17/17

Introduction

Purpose

This document specified architecture and software design decisions for the game, Highwater.

Scope

This describes the software architecture and software design decisions for the implementation of Highwater. The intended audience of this document is the developers, designers, and software testers of Highwater.

Design Goals

Creating a game poses a unique software development situation. The developers work closely with both technical and non-technical designers who may change requirements frequently, or request new functionality further into the development process. Additionally, there will only be one major release of the software, with potential bug fix patches. Taking this into consideration, the following principles guided the design of Highwater:

Maintainability

With a large team of developers it is important that developers are able to change one component of the project without needing to change many components that other developers may be working on. In addition, with requirements that were often in flux, it is important to be able to easily change functionality that had already been written without disturbing other related components. Therefore, all major components of the game such as the player, UI, and city generation are separated into decoupled modules. Furthermore, the back-end code that does not directly use objects in a Unity **scene** is contained in a layer separate from front-end code. This allows developers that did not program the backend to be able to make adjustments to code that uses this functionality in the front end.

Within Unity itself, **prefabs** were also used to define **GameObjects**. Updating a prefab will update all instances of that GameObject used in the game, allowing for repeated use of GameObjects both within and across scenes.

Optimized Performance

The game will need to have optimized performance as the game is updating and rendering 60 times per second. This means only performing costly operations when necessary and performing function that need to be run every frame as efficiently as possible to provide the best experience for users.

Designer Friendly

Allowing non-technical designers to make modifications to the behavior of the software without having to understand code was a necessity. This means avoiding hard coding variables wherever possible and instead exposing the values in the **Unity Inspector**, where designers can make changes without modifying any code. In addition, build processes were automated via scripts which designers could run through menus in Unity.

Furthermore, certain components in the project can be configured using a text file, allowing components to be easily expandable without modifying code.

Definitions

Block	The second-level subdivision of the city which define an area between roads where building may be placed.
City Bounds	A predefined volume in which the procedurally generated city and all object within the game are confined to.
District	The first-level subdivision of the city which have unique configuration for buildings and items generated there.
GameObject	Base class for entities in Unity scenes
Inspector	Unity window that shows the properties of whichever GameObject or script is currently selected in the open scene
Instance	A specific instantiation of an object
MonoBehaviour	Base class from which all Unity scripts derive
Prefab	A GameObject that is saved in Unity with set properties so that it may be duplicated in the scene as needed
Procedural Building	A type of building placed in the procedural city, generated at runtime from a configuration.
Scene	An instance of the game that can be compiled as part of a game build. Games must have at least one scene, but can have many (i.e. different Scenes for each level, start screen and game over screens, etc.)
Script	Code that is attached to game objects to trigger events and set their attributes
Template Building	A type of building placed in the procedural city created by a designer and placed as-is.
Unity	Cross-platform 3D game engine
YAML	(YAML Ain't Markup Language) A data serialization language made for human readability.

References

Highwater Game Design Document

Highwater Acceptance Test Plan

Highwater Generated Documentation: This document details all the modules and references the design document requirements listed in the requirements of the Game Design Document, linked above. Due to the length of the generated documentation and the large size of the codebase, we have linked the document and provided the PDF file (HighwaterGeneratedDocument.pdf) separate from this document.

System Overview

Technologies Used

The game engine Unity is used for lower-level functionality such as loading assets, rendering, and physics. Our source code is written in C#. We also leverage some third-party C# libraries for creating and parsing YAML files, and creating audio files through text-to-speech. Unity plugins are also used for managing sound (FMOD) and 3D model animations (DOTween).

Architecture

Highwater Generated Documentation: This document details all the modules and references the design document requirements listed in the requirements of the Game Design Document, linked above. Due to the length of the generated documentation and the large size of the codebase, we have linked the document and provided the PDF file (HighwaterGeneratedDocument.pdf) separate from this document.

Project Overview

The Highwater project is built around the Unity game engine and combines our source code along with third party libraries used with assets generated by the design team, including 2D textures, 3D meshes and animations, shaders written in GLSL, and audio files into one project.

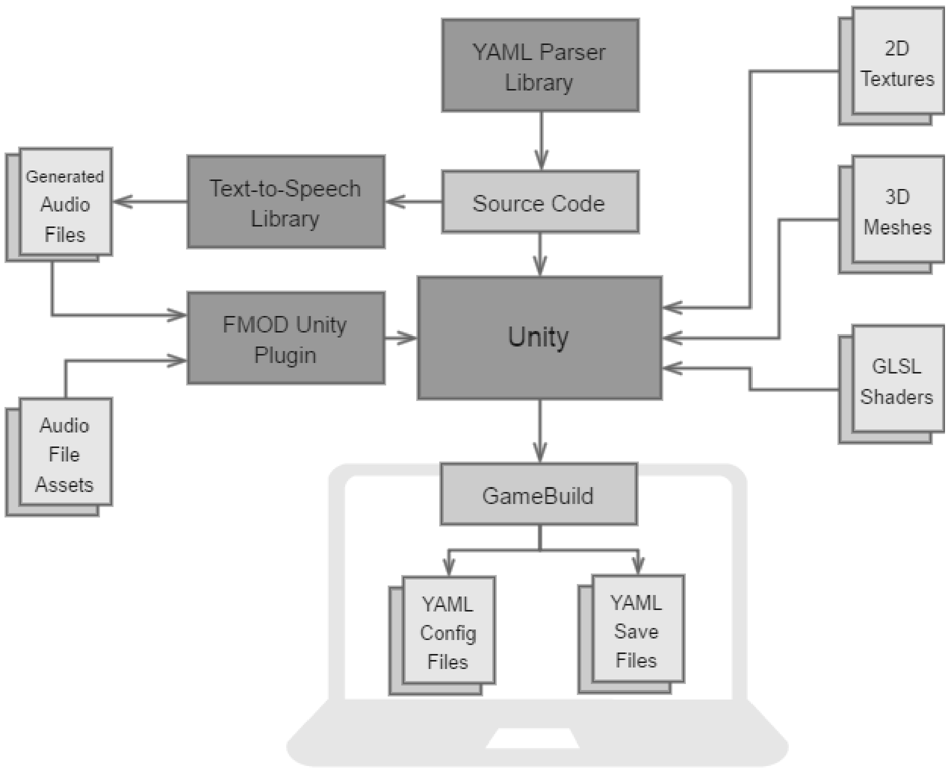


Figure 1. Architecture of Highwater project

Using Unity, the project can be built on both PC and Mac. When run locally on either of these systems, the game build employs Yaml files for configuration data, as well as to save data. This allows the game's configuration to be changed by both non-technical designers before the final release, as well as by the end users to modify and extend the game. Since Yaml is used to hold configuration data and libraries were chosen to parse these files, Yaml is also used to store and load saved game data.

Module Overview

The major code is broken down into modules, including the city, the weather, event controller, weather, player, and items & crafting. All separate modules are controlled and can be accessed through the master Game class.

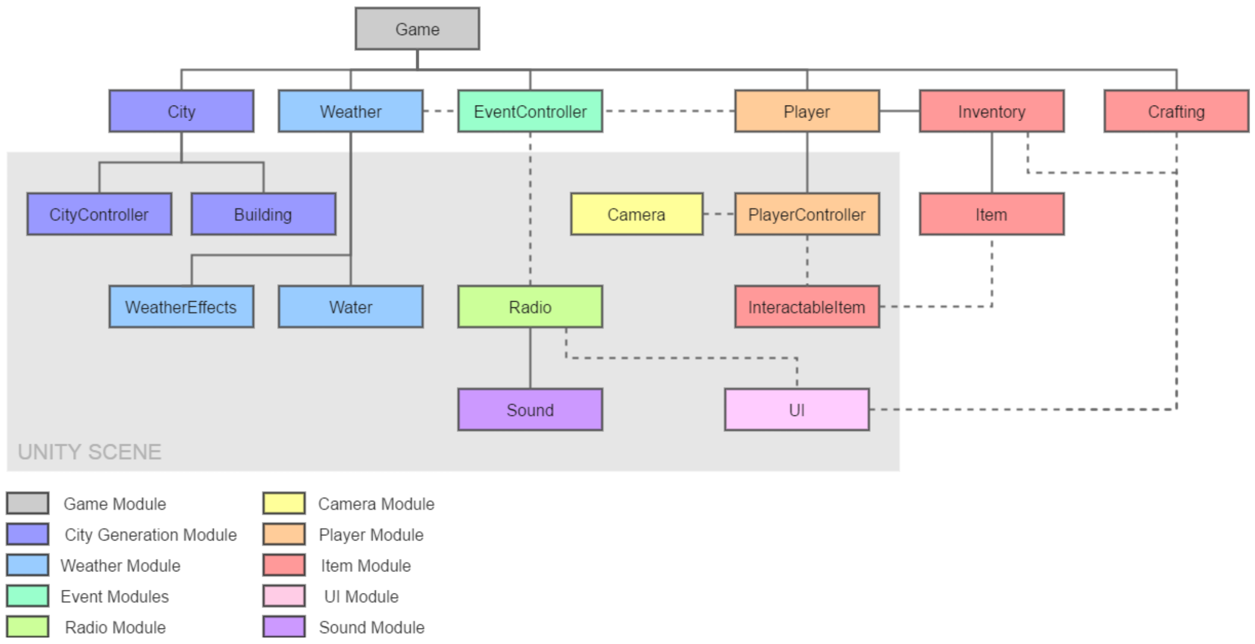


Figure 2. Overview of source code modules and their relationships

Furthermore, code within each module may be broken down into two groups. Back-end code can be considered standalone code that does not execute directly in a Unity **scene**. This would be data representations of objects like the city, the player, and her stats. Front-end code is attached to **GameObjects** and executes in the Unity scene. This involves controlling graphical objects, updating physics, or using ui or sound.

Game Module

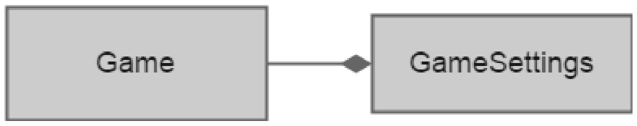


Figure 3. Game module class relationship diagram

Game is a singleton and can be accessed from any other class. This allows Game information and status to be used in any class easily. Game also contains an instance of GameSettings which holds the data for settings the user can configure in the menus, such as key bindings and music volume.

City Module

The City Module is comprised of three major elements, city representation, city generation and city management, all of which are controlled by the CityController.

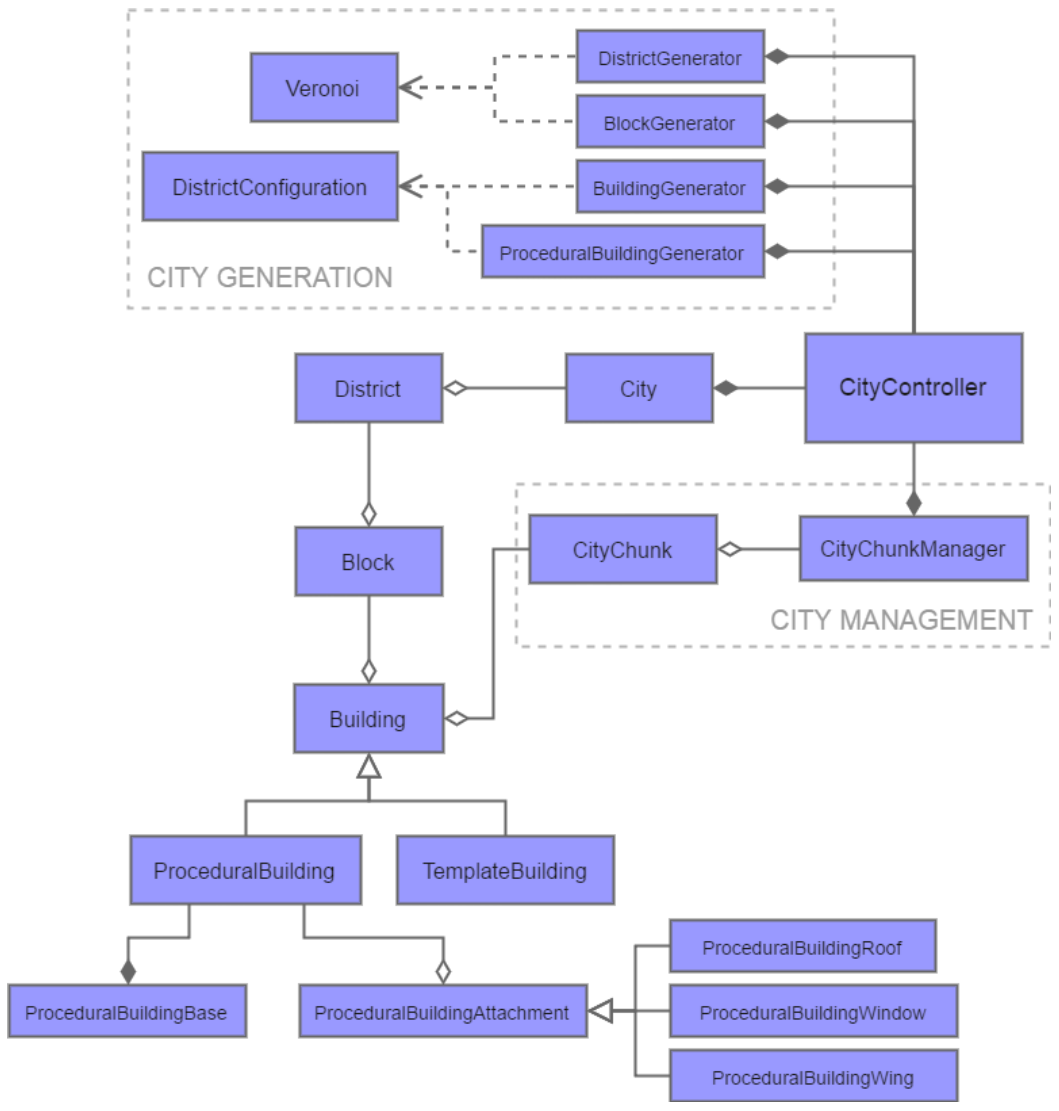


Figure 4. City module class relationship diagram

The city itself contains a list of districts, which contains a list of blocks contained in that district, which contains the buildings in that block. Buildings can be either a complete model (referred to as a `TemplateBuilding`), or a procedurally generated mesh (referred to as a `ProceduralBuilding`).

The city is generated based on a in-depth configuration that the designers specify in the Unity inspector. The `CityController` object has generators for each level of the city generation (`District`, `Block`, `Building`, and `ProceduralBuilding`) which contain variables the designers can manipulate to produce different results, as well a `Generate` method called by the `CityController` at the proper step in the generation process. The Voronoi utility class takes seed points as input and returns a Voronoi diagram.

Since there are many meshes in the city to render and calculate physics against, they cannot all be loaded at once. The city is managed by breaking the entire city down into a grid of chunks. Each chunk contains the list of buildings its boundaries encompass. The `CityChunkManager` each frame checks to see if the position of the player has changed. If it has, it updates by loading the chunks closest to the player and unloading the chunks far from the player.

Weather Module

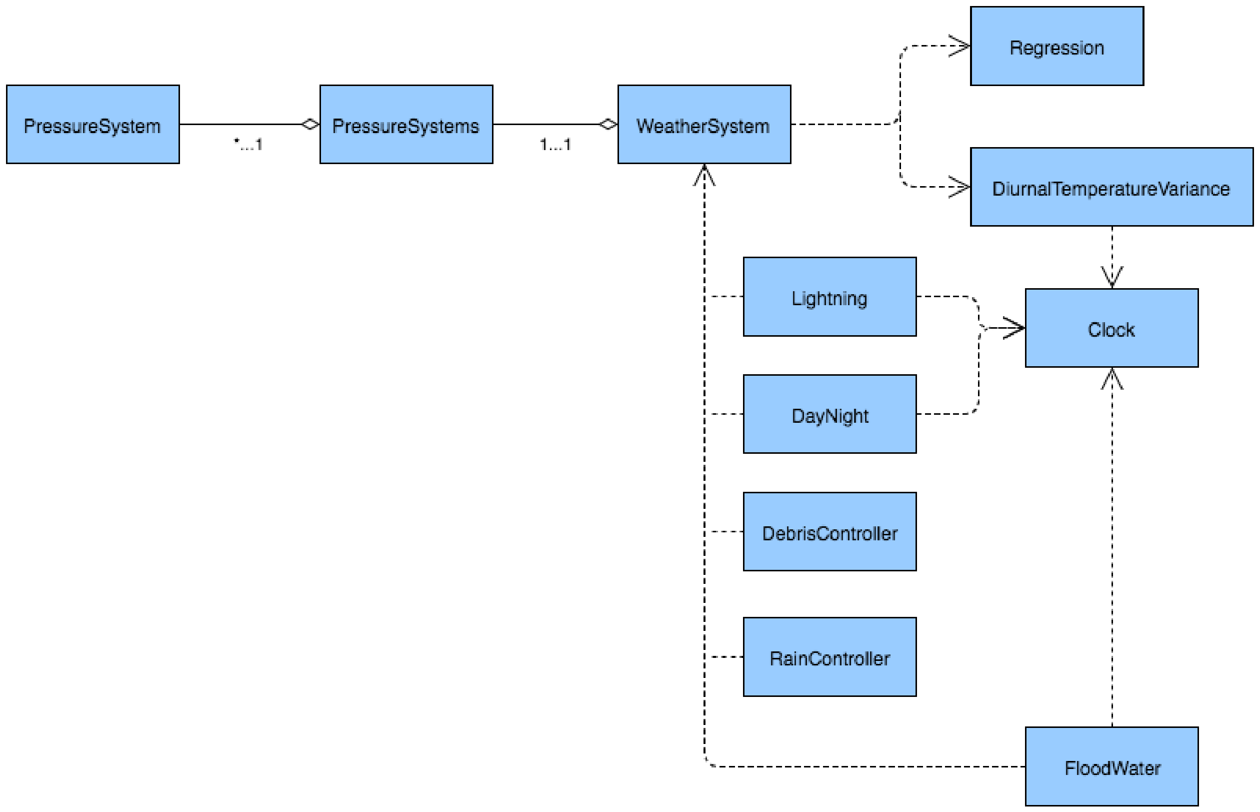


Figure 5. Weather module class relation diagram

The weather module uses the `PressureSystem` class as a data structure store the information associated with a pressure system. `PressureSystems` has a list of these and provides various functions such as finding the nearest pressure system to the `WeatherSystem` class. Additionally, the `WeatherSystem` class continuously updates the pressure systems else the pressure systems will remain stagnant. The weather system uses the `Regression` class to run a regression algorithm on coefficients and input. This is used in the calculation of the current weather for the player every frame. `DiurnalTemperatureVariance` is using the clock to determine what part of the day it is and affect the temperature based on this information. The clock is continuously updating and is responsible for handling diurnal temperatures along with many other functions out of scope for the weather. The `FloodWater` class performs a function similar to the the `DiurnalTemperatureVariance` in that it uses the theory behind it and applies it to tides and adjusts the height of the water. Additionally it uses the `WeatherSystem` to get the current level of precipitation to affect the height of the water over the course of the game.

Lightning uses the clock to vary how often a lightning strike will occur relative to the game. It also uses the weather system to define how often this will occur. `DayNight` rotates the sun and the moon around the world using the same theory behind diurnal temperature and tides. `DebrisController` and `RainController` control particle effects that show how bad or good the weather current is according the `WeatherSystem`.

Event Manager Module

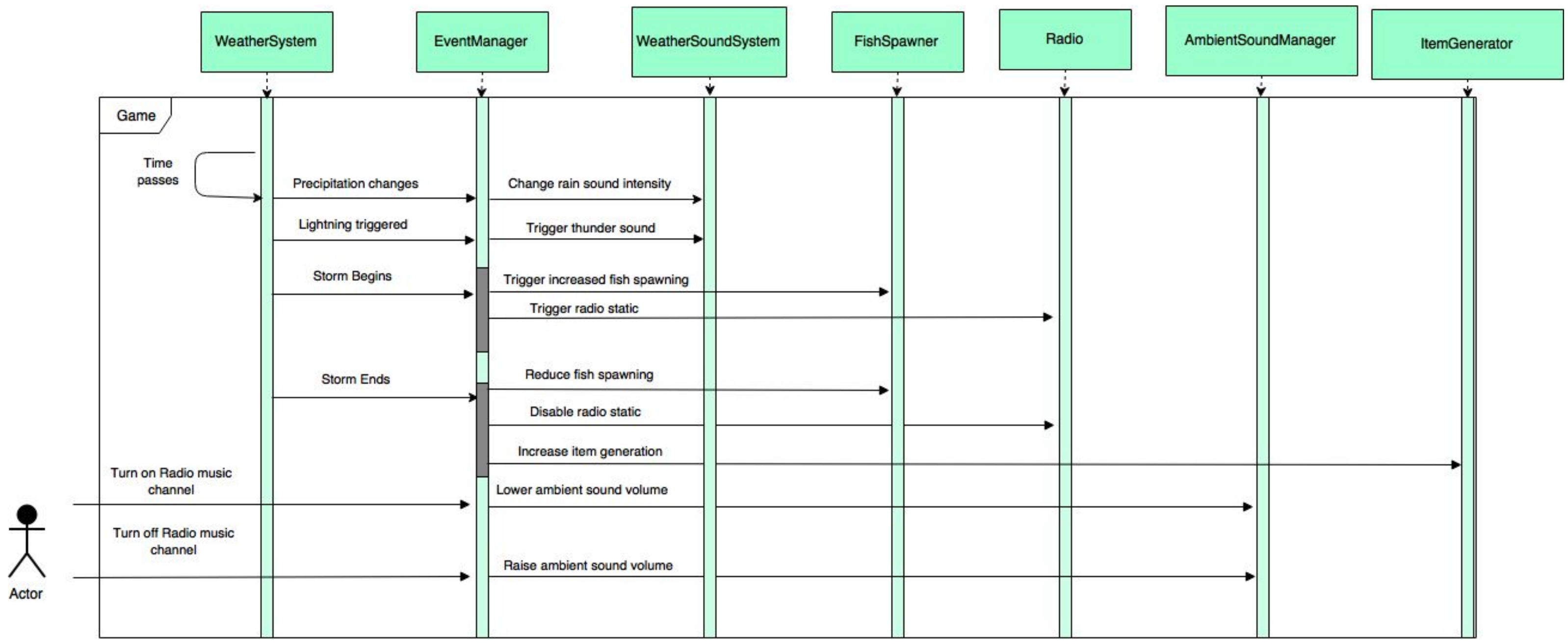


Figure 6. Event Manager Swim Lane Diagram

The `EventManager` module has one main component and acts as a liaison between several game modules, including the `WeatherSoundSystem`, `WeatherSystem`, `FishSpawner`, `Radio`, `AmbientSoundManager`, and `ItemGenerator`. This module maintains a series of events that can be triggered and subscribed to through an instance in the `Game` class, decreasing the dependency between different parts of the code and allowing for new interactions to be created without the need for major design changes in existing scripts.

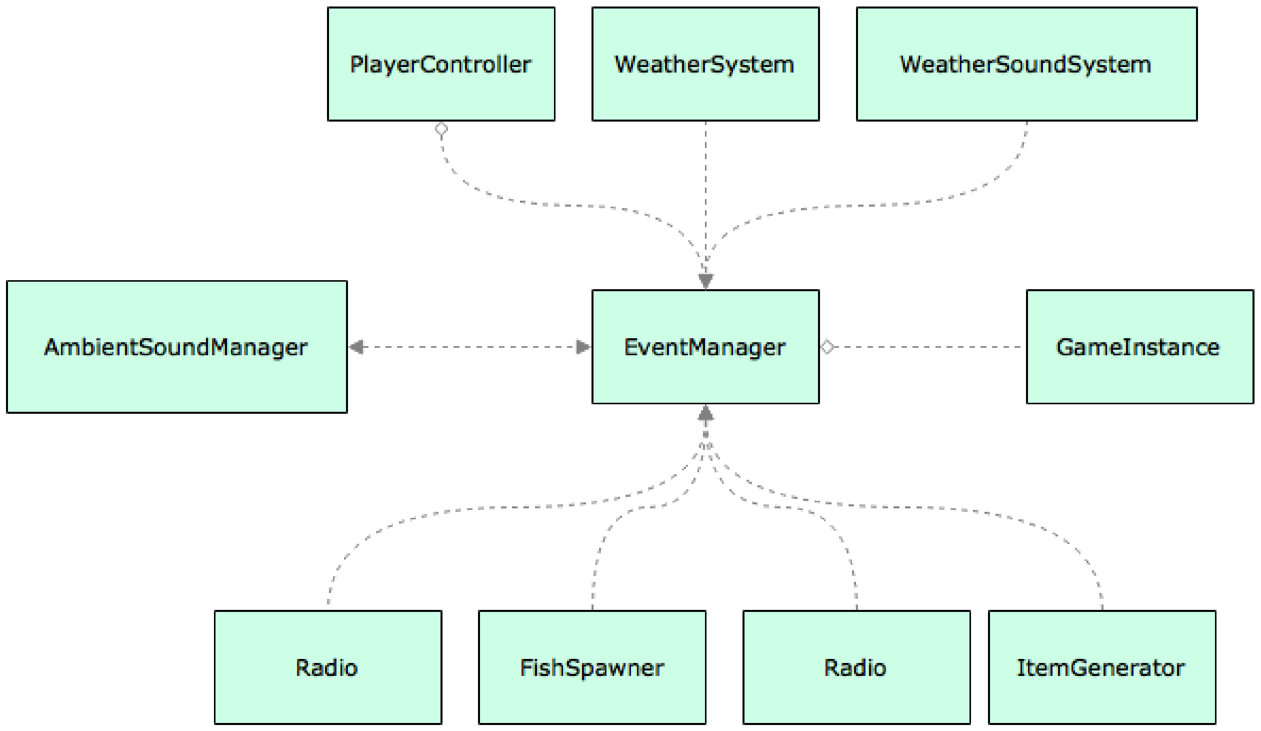


Figure 7. Event Manager Class Relationship Diagram

Radio Module

The radio module has one main component. The `Radio` class controls the functionality of the radio, such as: changing the station and producing sound based on the selected station. To produce sound, two third party tools called FMOD and RT Voice are utilized. FMOD is a sound system, while RT Voice is a real-time text-to-speech asset.

Player Module

The player module has two major components, the backend code representing the player data like stats, and the front-end representation, which controls the player GameObject, like her movement, animations, and getting keyboard and mouse input from the user.

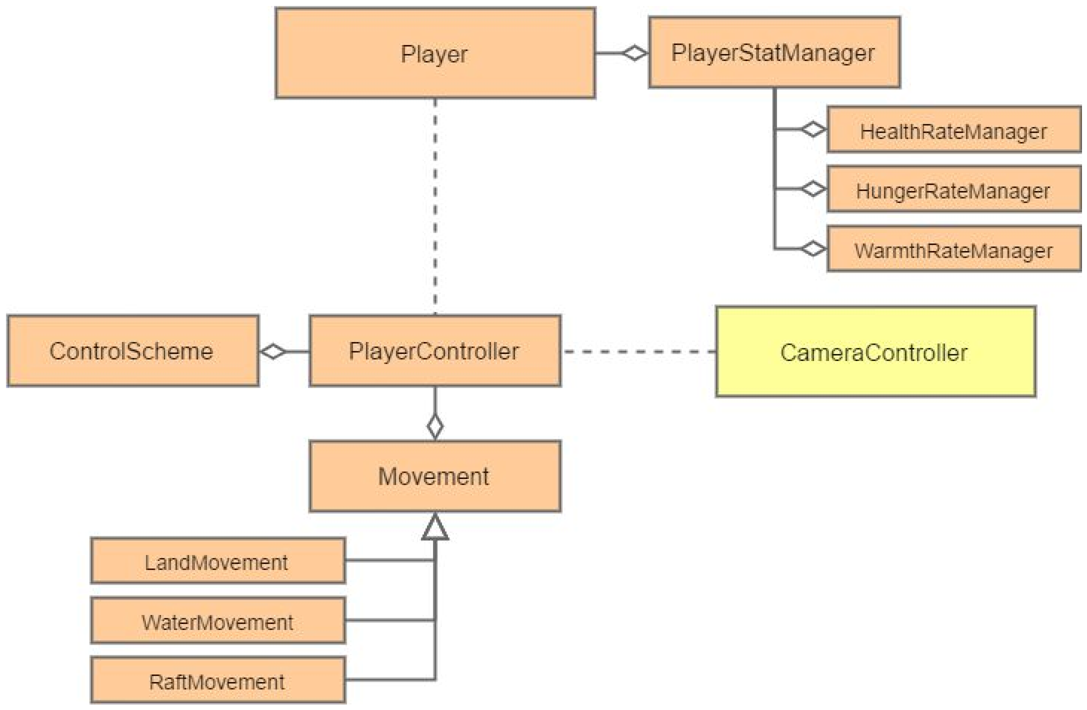


Figure 8. Player module and Camera module class relationship diagram

The `Player` class holds data about the current status of the player like death or any current status conditions. This includes the player stats, health, hunger, and warmth. The `PlayerStatManager` updates these stats according to rules specified in the `HealthRateManager`, `HungerRateManager`, and `WarmthRateManager`.

The `PlayerController` class is the class which controls the player GameObject in the scene. It checks for player input, checks against the current control scheme configured in the `ControlScheme` class, and performs the proper functions. This includes movement, which takes advantage of the `Bridge` pattern. Here the player will perform Move(), which is defined differently depending on what surface the player is on, either through `LandMovement`, `WaterMovement`, or `RaftMovement` which all inherit from the parent Movement class.

Camera Module

The camera module consists of one class which controls the Camera GameObject. It uses information about the player in the scene, such as the player position and user input to update the camera position and rotation.

Crafting and Inventory Module

The crafting and inventory module consists of all classes related to `Item` definition, the `Inventory` class that stores those items, and the factories that creates items. The `Inventory` contains stacks of items, allowing for multiple instances of the same item to occupy one slot.

The `ItemFactory` is used to create backend data of an item used by the `Inventory`. The `WorldItemFactory` is used to create an `Interactable` gameobject in the world that represents an Item.

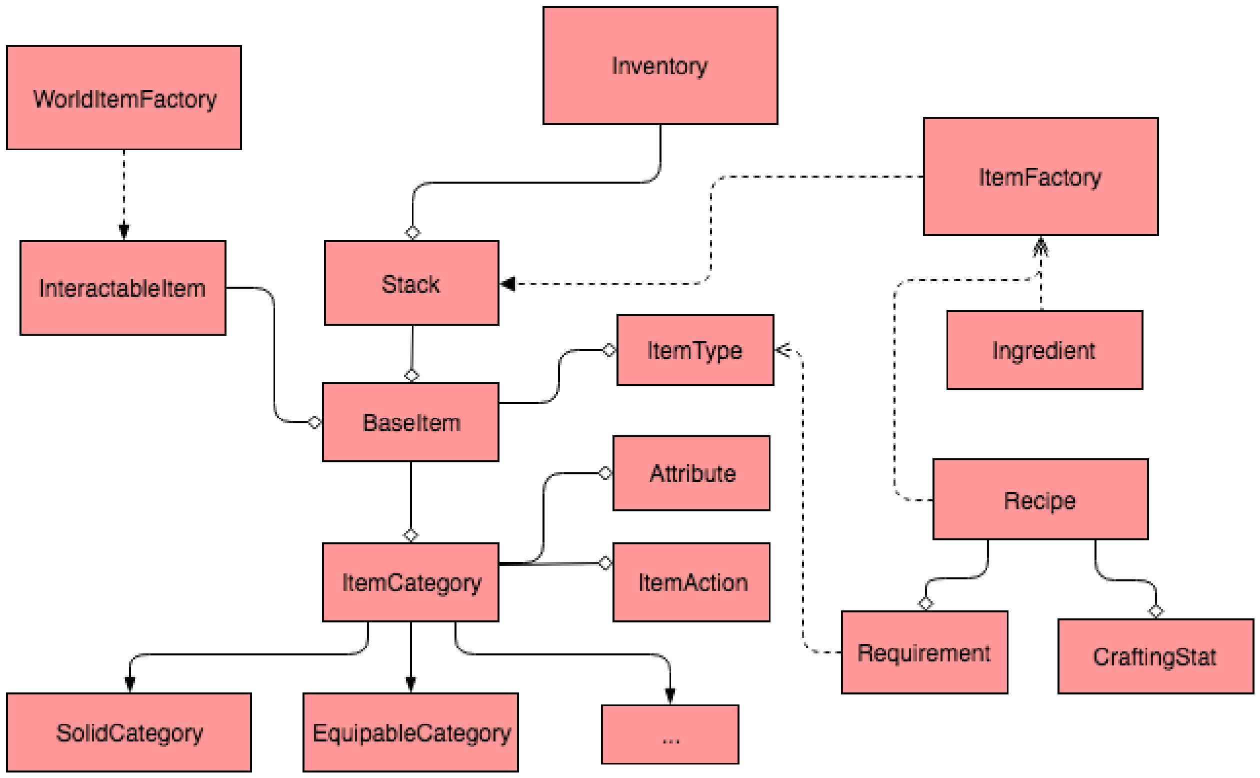


Figure 9. Item Crafting and Inventory Module class relation diagram

Items are defined by `Item Categories` which are suites of `Actions` and `Attributes` that describe how an item behaves. `ItemTypes` describe what the item is. This is implemented using a decorator pattern. Various category classes can be attached to the `BaseItem` class to create unique combinations that drive item behavior. Inventories are made up of stacks, which consist of an item and the amount of the item that is in the inventory.

Crafting occurs when multiple items are combined to make a new item. Recipes define how items can be combined. Each recipe contains a list of Requirements--which state an item type required, and how many of that item type is required--and the `CraftingStats` that will be checked to determine the quality of the crafted item. During the crafting process, the items selected to be used for a recipe is saved as Ingredients. The `ItemFactory` takes these `Ingredients` and the `Recipe` specified and creates a new item. The new item is added to the `Inventory`, and items used up as an `Ingredient` is removed.

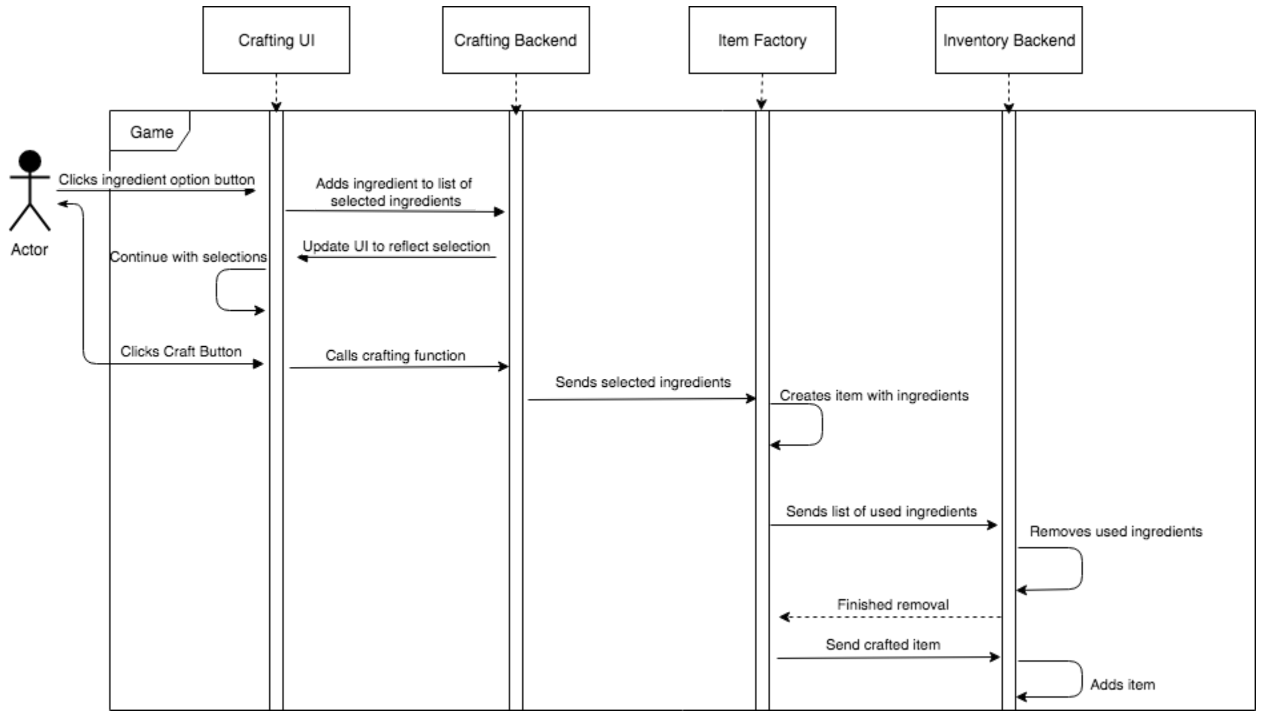


Figure 10. Swim lane diagram describing Crafting code flow

UI Module

The UI module consists of the classes that connect to backend data, interpret it, and display the information on screen for the user to interact with. This information updates when the backend data is changed. The UI includes the `Player Stats`, buttons to open up menus, the Inventory UI Panel, the Crafting UI Panel, the Radio Panel, and the Notifications Panel.

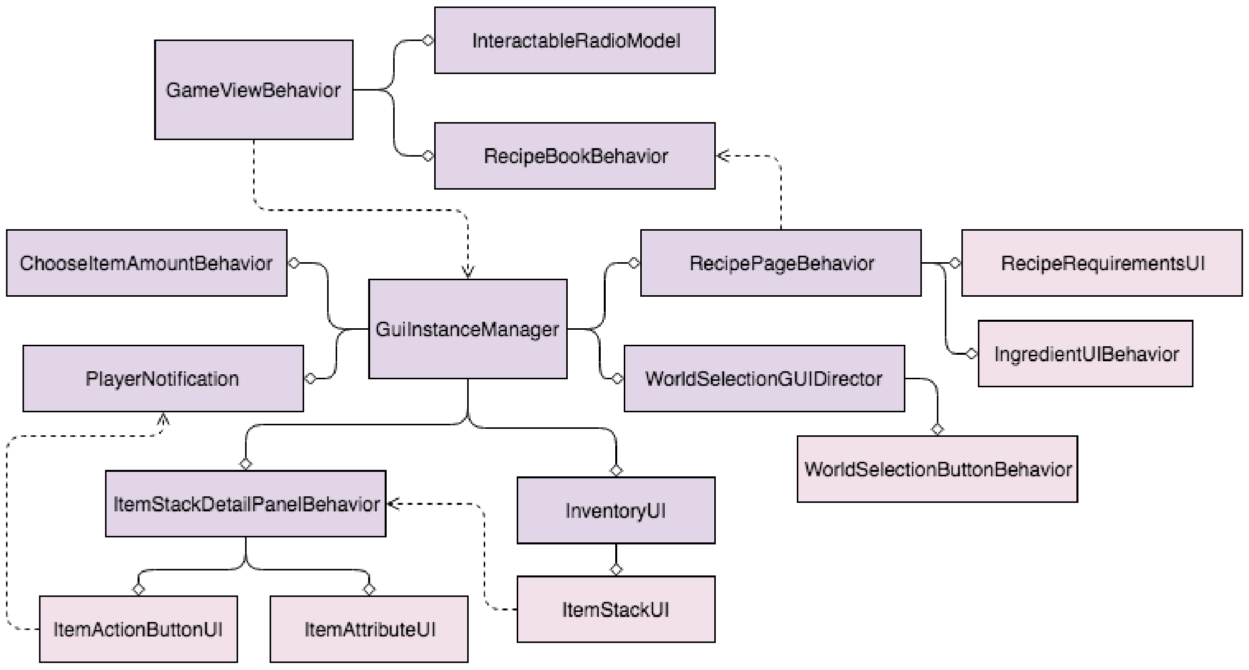


Figure 11. UI Module class relation diagram

The UI Controllers are singleton objects stored within the `GuiInstanceManager` class. The `GameViewBehavior` class references the `GuiInstanceManager` in order to control which UI elements should appear on screen.

`InventoryUI` controls the panel that contains a grid of all item stacks that the user has on hand. Within that panel is a collection of `ItemStackUI` classes that controls what happens when the user hover over an item stack or clicks on the item. It also updates how many items there are in the stack and the sprite image and name of the item when the information is changed.

When a grid cell is clicked, the `ItemStackUI` class will call the `ItemStackDetailPanelBehavior` class through the `GuiInstanceManager`. The `ItemStackDetailPanelBehavior` class handles the behavior of the subpanel that details information about an item. On the subpanel, an item's description, attribute values, and possible actions are shown. The `ItemAttributeUI` drives an UI element which displays an item's attribute's name and value. Each attribute ui element has this class attached. The `ItemActionButtonUI` stores an action that can be done to an item, and fires it when the button it is attached to is pressed. Once an action button is pressed, the `ChooseItemAmountBehavior` class is called through the Instance Manager. The `ChooseItemAmountBehavior` class handles selecting how many items in the stack will be affected by the action.

The other major UI Panel is crafting panel which is controlled at the highest level by the `RecipeBookBehavior` class. This class handles displaying recipes and updating the list to ensure that craftable recipes are placed at the top of the list. Once a recipe in the list is clicked, control shifts to `RecipePageBehavior`. The `RecipePageBehavior` class handles displaying information about a recipe. Through the `RecipeRequirementsUI` class, the `RecipePageBehavior` class displays what requirements there are for the recipe. If the user choose to begin crafting the recipe, the `RecipePageBehavior` class changes the UI to display panels containing buttons that utilize the `IngredientUIBehavior` class to select the items to be used in the crafting process.

The `InteractableRadioModel` links to the backend `Radio` code. It handles interactions with the 3D radio that appears when the Radio button is clicked. It handles radio animation when the player clicks on the radio's buttons or turns its knobs. It also handles how the radio appears and fades from the screen.

When interacting with items inside the world, at times it may require the player to select an action to do to the item or an item from the inventory that should be used as part of that action. In this case, the `WorldSelectionGUIDirector` class is used. It populates a panel with context specific actions or item choices.

All notifications and warnings to the player use the `PlayerNotification` class, which controls the notifications panel and the text that should appear on it.

YAML File Structure

YAML files are used to define configuration data as well as to save game data. YAML (YAML Ain't Markup Language) is a data serialization language made for human readability. Yaml files use whitespace to denote structure and all begin with three dashes to indicate the start of the file. Each file uses a specific structure as detailed below.

Item List YAML

This Yaml file contains the complete list of items that can be found during gameplay. For each item added, the following structure must be used:

```
---

       - itemName:
         baseItem:
          itemName:
          rarity:                                                 Can be uncommon, common, or rare
          inventorySprite:                                        The path to the sprite used for the
                                                                  inventory UI
                                                                  The file should be in Assets/

          worldModel:                                             The path to the model used for the item
                                                                  when placed in the world
                                                                  The file should be in Assets/Resources
          types:
```

```
  - “type”
 flavorText:                                   The description of the item
 modifyingActionNames:                         The actions that modify the item
  - “action”
 actionModifiedModels:                         The models that are used after a
  - “modifiedItem”                             modifying action is used
                                               Note: They must be in the same order as
                                               their corresponding action above

 actionModifiedSprites:                        The sprites that are used after a
  - “modifiedSprite”                           modifying action is used
                                               Note: They must be in the same order as
                                               their corresponding action above

itemCategories:                                The categories the item belongs to
  - !!category                                 Each category name should start with !!
    Attribute: value
```

Inventory YAML

This Yaml file stores the contents of an inventory. It’s accessed during the start of the game to load the current state. Any changes to the inventory is recorded in the file when saving. This file is very similar in structure to the Item List Yaml. Items in this file, however, are stackable, so there are several stacks that are identifiable by their unique `stackId`. The `stackId` initiates at 0 and increments by 1 for each additional stack.

```
---
- inventoryName:
 items:
  - stackId:                                   Initiates at 0
    itemAmount:                                Number of items in stack
    item:
     itemName:
     rarity:
     inventorySprite:
     worldModel:
     types:
      - “type”
     flavorText:
     modifyingActionNames:
      - “action”
     actionModifiedModels:
      - "modifiedModel"
     actionModifiedSprites:
      - "modifiedSprite"
```

```
itemCategories:
  - !!category
    attribute: value
```

Recipe YAML

This yaml file contains all the recipes that the player may use during gameplay. This differs from the `UnlockedRecipes.yml` file which contains only the recipes that are currently available to the player. However, both of these files utilize the same yaml structure. There is a tiered attribute that is used to identify if certain stats of the required items should dictate whether the created item is poor, standard, or good.

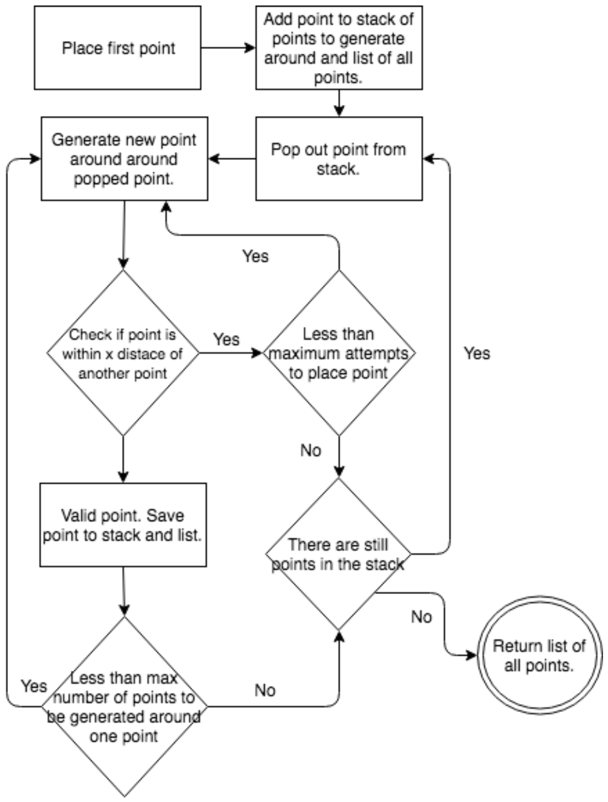
```
---

        - recipeName:
         tiered:                          Either true or false
         toolRequirements:                The tools that are needed to
                                          complete the recipe

         resourceRequirements:
          - itemType: type
            amountRequired:
         statsToCheck:                    The stats that are used to help
          - statName:                     identify the what tier the
                                          created item will be

            statAffectingItems:           The items that affect the
             - "itemType"                 created item based on the
                                          identified stat

            qualityThreshold:             The threshold that defines what
                                          tier the created item will be
             - lowerThreshold
             - upperThreshold
```

Algorithms

Highwater takes advantage of several known algorithms as part of several of the systems, including generating the city, items, and weather.

Poisson Disc Sampling

Poisson Disc Sampling is used to generate several positions procedurally that seem random and also naturally distributed. Unlike pure random generation which may appear clumped or irregular when observed, Poisson Disc Sampling produces a spread of points that are more evenly spread and more natural in appearance to human eye. This method is used throughout the project to select location including to generate the city locations, and well as to distribute items throughout the city.



Place first point
Add point to stack of points to generate around and list of all points.
Pop out point from stack.
Generate new point around around popped point.
Yes
Check if point is within x distace of another point
Yes
Less than maximum attempts to place point
Yes
No
Valid point. Save point to stack and list.
There are still points in the stack
No
Return list of all points.
Yes
Less than max number of points to be generated around one point
No

Figure 12. Poisson distribution algorithm flow

Points are generated by taking a generated point and attempting to generate a new point around the existing point. The new point is generated a minimum distance away from the existing point. The smaller the minimum distance, the tighter objects will be packed. Then, the space around that new point is checked to ensure that no points in its vicinity is within the minimum distance specified. If there are none, then the point is valid and placed. Otherwise, another point is generated. In the case of item generation, Poisson Disc Sampling also takes into account the size of the objects it is attempting to place and the minimum distance desired between placement points.

A grid is used to keep track of existing points to allow fast lookup of neighboring points. Placed points are given a coordinate in the grid and stored. When a new point is checking to see if any neighboring points are within minimum distance, all the points in the grid cells surrounding the new point's grid coordinate are checked.

Voronoi Diagrams

Voronoi diagrams partitions an area based on the location of seed points into cells. Partitions between cells are created between points so that both points are equidistant to the partition line.

The seed points are defined beforehand as passed as input. This creates natural looking cells of varying shapes and sizes, and can be controlled by changing the seed points passed to the algorithm.

City Generation

City generation make use of Voronoi diagrams and the Poisson Disc Sampling described above.

Numbered seeds were used as input to generate the city. One of the most important aspects of the city is that it generate differently for each seed, but providing the same seed would result in the same city each time. To enforce this, a random number generator is seeded with the provided seed at the beginning of the process, and all generation variations are based on output from the random number generator.

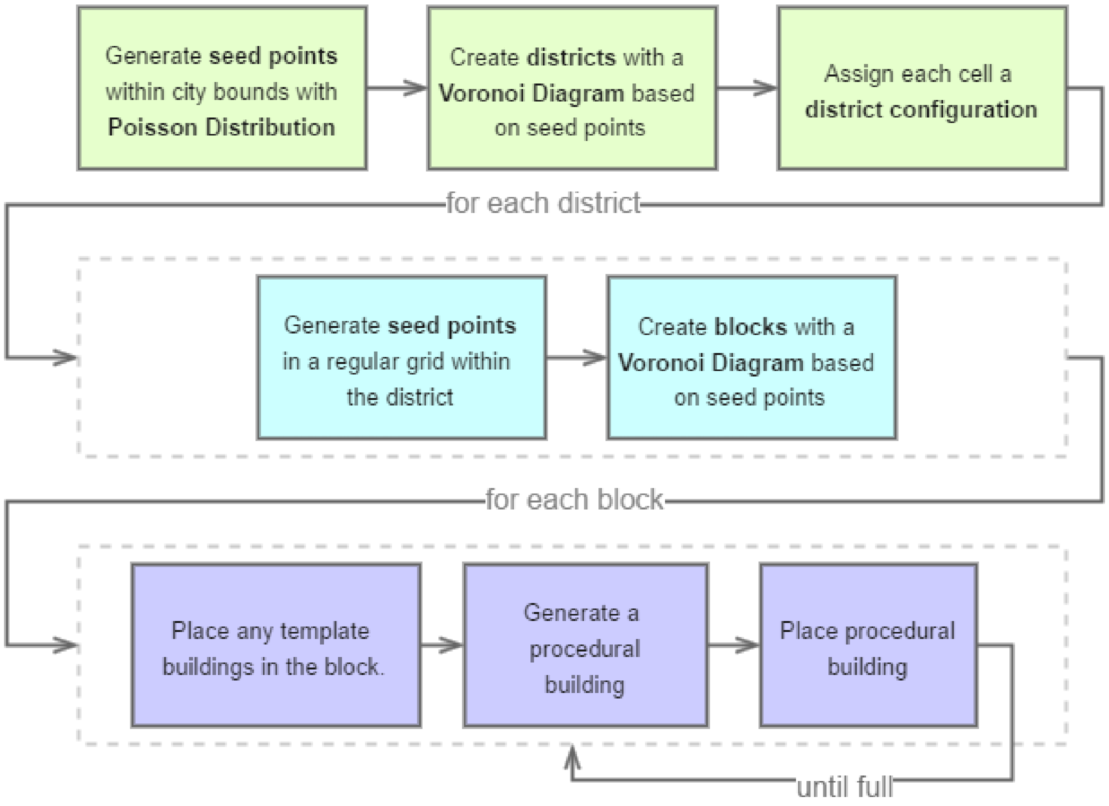

Generate **seed points** within city bounds with **Poisson Distribution**
Create **districts** with a **Voronoi Diagram** based on seed points
Assign each cell a **district configuration**
for each district
Generate **seed points** in a regular grid within the district
Create **blocks** with a **Voronoi Diagram** based on seed points
for each block
Place any template buildings in the block.
Generate a procedural building
Place procedural building
until full

Figure 13. Process for generating the city

First, seed points are generated within a predefined volume call the **city bounds** using Poisson Disc Sampling. These points are then used as input to a Voronoi diagram. The cells of this diagram are city **districts**, and each is assigned a unique district configuration which includes a name and configuration information for how buildings are generated in this district as well as which items may be generated there.

Next, within each district, seed points are generated in a regular grid pattern. Some jitter may be applied to the grid point positions in order to give the grids slight natural variations. These seed points are then again used as input for a Voronoi diagram. The cells of these diagrams define **blocks**, or areas where buildings may be generated, and the edges of the diagram are used to define road to border each block.

The city is populated with two types of buildings, template buildings and procedural buildings. **Template buildings** are duplicates of gameObjects designers create. These are simply placed as is into a location in the city. **Procedural buildings** are used to fill in the city and are the majority of the building seen. They are generated by selected one of several building base meshes and combining this mesh with one of several roof meshes. Then for each side of the building, there is a designer-defined chance of generated a building attachment, which could be a wing of the building, a shelter object, or a window washer platform object.

Inside each block, template buildings are placed first based on Poisson Disc Sampling described above, as they tend to be larger and more important than the procedural buildings which act as filler. The remaining space of the block is filled by generating procedural buildings. A building is generated and packed into the block until no more buildings have room to be placed.

Flocking

Flocking is the process of merging several steering behaviors, specifically cohesion, separation, alignment, and wander. To combine each behavior a weight is applied to each result acceleration vector and the resulting vectors are added together. This acceleration is applied to the GameObject of the fish to create realistic flocking behaviors. Additionally, this has many variables associated with it such as the weight for each of the behaviors. Instead of creating one file that every fishes uses, each agent at the start will create a configuration with randomized values between pre-decided values. This is to give the flocking a more authentic look as each fish will move different presented with the same situation.

Cohesion, separation, and alignment will be affected by the surrounding fish. A physics overlapsphere will be used, built into unity, to find the fish in the desired radius. This radius is set in a configuration file. Testing showed that this was more efficient than looping through all the fish available. Each algorithm will then use the found fish to affect their behavior. This effect will be listed below in each steering behaviors description.

Cohesion is a behavior that brings surrounding agents together. To accomplish this a physics overlapsphere is used to find the nearby fish and then the average position of each fish can be found. A direction vector can be calculated by subtracting the current fish's position from the average position of each fish. The resulting vector is normalized and returned. If no fish are found then an empty vector with zero values is returned.

Separation will cause agents to disperse so they don't get too close to each other. This will once again find the nearby fishes and iterate through each. In each iteration a vector will be calculated to find the direction vector facing towards the fish defining its behavior by subtracting the fish's position by the fish that is currently being iterated on. If the magnitude of the resulting vector is not 0 then the result of the normalized vector divided by the regular vectors length will be added to a resultant vector. This makes it so the contribution of each fish will vary based on the distance. The resulting vector will be normalized and returned. If, however, no fish are found at the start a zero vector will be returned.

Alignment forces agents to face the same direction. This will once again use a physics overlapsphere to find the nearby fishes. Each of the nearby fishes velocities will be added up and normalized. This normalized vector is returned, unless no fish are found and then a zero vector is returned.

Wander will cause the agent to wander around the environment. This uses a configured jitter, a float, which will affect how much the agent will wander. This is done by calculating a vector which uses a random binomial multiplied by the jitter for each coordinate in the world space. A random binomial is a random number, between zero and one, subtracted from another random number. The result is a number that is between -1 and 1 with a higher likelihood towards 0. This resulting vector is then normalized and multiplied by a configured wander radius to give the configured magnitude for the behavior. The vector is then transformed into world space as it currently represents a point. The world space coordinates are subtracted from the agent's position to give a resulting direction vector. This vector is normalized and returned.

Weather

Weather is built around pressure systems that exist in the world. These pressure systems will move around the world using attraction and detraction forces. These forces, similar to magnets, are decided upon based on whether the pressure system is a high or low pressure system. Two low or high pressure systems will detract, push, and one of each will attract, pull. Each of these pressure systems will have a pressure associated with it. Low pressure systems are associated with stormy weather and high pressure systems are associated with clear skies and warmer weather. The pressure systems are gamified by adding an additional force that pushes low pressure systems towards the center of the city.

To calculate the weather the player is currently facing the closest pressure system is found and the pressure at the center is used as the basis for calculating the remaining weather variables: pressure, temperature, wind velocity, relative humidity, relative dew point, and precipitation.

Pressure is calculated by measuring the distance the player is from the center of the pressure system and adjusting the pressure systems pressure based on the distance for the player. If the player is in a low pressure system then the pressure is increased, else, the player is in a high pressure system, the pressure will decrease. This is done by using a multiplier of 1 for low pressure systems and -1 for high pressure systems.

Temperature is calculated from two different results. The first is based on the time of day, where when the sun is at the highest it will add the most heat. When the moon is at the highest point, it will subtract the most heat from the temperature. This is a diurnal temperature system which is an approximation of the real world's temperature throughout the day. The second part is calculated based on the pressure that was found above using the ideal gas law:

$$temperature = \frac{pressure}{Air\ Density}$$

The air density uses a configured constant based on the average density of air in the real world. This once again is an approximation as this will only work in an ideal world. The resulting temperature is converted from kelvin to fahrenheit.

Wind velocity required very complex formulas and fluid dynamics that were out of scope for this project. Additionally, the required information for these formulas do not exist in the game world. Instead a quadratic regression was opted for to approximate the way wind behaves. To find the ideal coefficients a separate machine learning library can be used to find the coefficients from real world data. In this case SkLearn, a python machine learning library, and Wunderground.com can be used to approximate the functions. In addition, extra data will be needed to handle the extreme points of high and low pressure. The coefficients can be given to a regression function which will then calculate the result based on the inputs. For wind speed, every previously calculated variable is used as each relates to wind: pressure and temperature. Figure X shows the resulting wind speed calculation. Additionally it displays why quadratic regression was opted for over linear regression and it provides a better approximation of the data with the extremes.

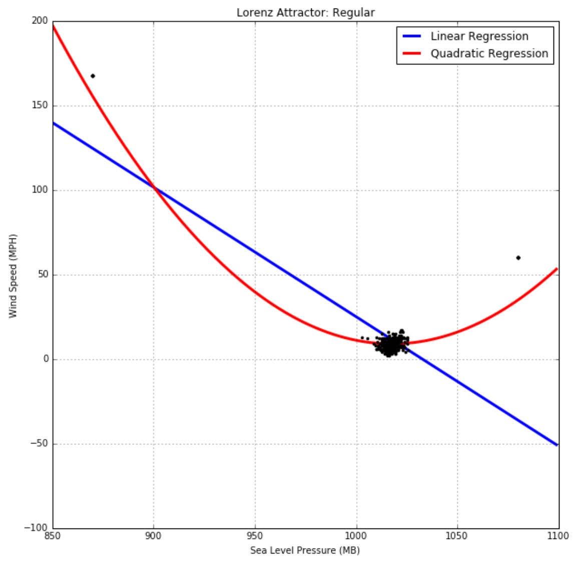


Figure 14. Wind speed approximation based on real world data.

With the wind speed calculated the direction still needs to be calculated. For this common knowledge for weather systems is used to inform the way the wind is moving. Low pressure systems will have winds rotating around them counter-clockwise and inward. High pressure systems will have winds rotating around them clockwise and outward. The inward and outward directions indicate that each direction will change slightly to not be perpendicular to the resulting direction vector based on being counter or counter-clockwise. The wind direction vector is normalized and multiplied by the correct wind speed to give the correct magnitude.

Relative humidity isn't affected by the wind, but will require intense calculations or data that hasn't been found in the world up until this point in execution. Regression is once again used, this time with pressure and temperature with a different set of coefficients.

Relative dew point can be calculate using the August-Roche-Magnus approximation ("RH stands for relative humidity and "T" stands for temperature):

$$dew\ point\ = \frac{243.04 *(ln(\frac{RH}{100}) + (\frac{17.624*T}{243.04 + T})}{(17.625-ln(\frac{RH}{100})-\frac{17.625*T}{243.04+T}}$$

Precipitation is calculated with regression on pressure, temperature, relative humidity, wind speed, and the relative dew point that we have found above. A different set of coefficients is again used for this regression task.

As can be seen each calculation, besides pressure, requires variables from behavior. This results in a pipeline that runs one after the other. Also, due to the coefficients being pre-calculated this weather prediction requires very little time as no calculation requires extensive work.

Detailed Class Descriptions

Unity Background

Many Highwater classes inherit from the Unity builtin class **MonoBehaviour**. A class inheriting from MonoBehaviour allows the code to be connected to and control properties of a **GameObject**, Unity's representation of an object in a scene. Public variables as well as serialized private variables are exposed in Unity's **Inspector** where designers may change the value of those variables. MonoBehaviour also allows the user to implement built in methods like "Start" and "Update" which execute at specific times in the game loop.

See Unity's documentation for more information:
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Class Descriptions

Auto-generated Class Documentation

Traceability Matrix

The link to the comprehensive traceability matrix can be found here: https://docs.google.com/document/d/1oIqAubYWg2lhRLsJNgC_NQDcFtHDIlai5-3FAAZtmsE/edit?usp=sharing