NEPTUNE OCEAN EXPLORER

NEPTUNE Canada
Transforming Ocean Science

University of Victoria

SENG 499

Team:

Michael Pattie

Tom Gibson

Colton Phillips

NEPTUNE Canada sponsor: Dr. Maia Hoeberechts

ECE Supervisor: Dr. Kin Fun Li

Table of Contents

Table of Figures

1. Introduction

NEPTUNE: Ocean Explorer is a horizontal prototype developed for NEPTUNE Canada as an outreach program to develop interest in the NEPTUNE ocean observatory. Prior to the development of the NEPTUNE observatory, scientists relied on infrequent ship cruises to conduct their research. NEPTUNE Canada offers a unique approach to ocean science: an underwater ocean observatory connecting deep sea instruments to the Internet. People everywhere can view the seafloor while ocean scientists run deep-water experiments from labs and universities anywhere around the world.

In NEPTUNE: Ocean Explorer, players experience the challenge of building an underwater ocean observatory connecting deep sea instruments to the Internet. Players build their network one node at a time, with each node acting as a platform to connect instruments to. Each instrument has a price, a rate of data production and specific terrain on which it produces data. The data produced by the instruments generates funding for the player. Instruments display real cabled observatory data.

The goals set for the NEPTUNE: Ocean Explorer game are educating the player on cabled observatory technology, increasing awareness of oceanographic data and engaging the public in modern ocean exploration. The prototype has been realized with functioning and extensible systems; however, it is missing much of the science content that would engage players. The purpose of this report is to give a broad overview of the game architecture and to discuss problems and potential solutions to problems.

1.1 Implementation Platform

Because it targets Flash, the game is playable on any browser and may be ported to mobile devices. The game uses the open source game engine FlashPunk for most of its core functionality. The game is built in mind for educational deployment as it is easily navigable by any school computer. The target audience is younger children (8+) with sufficient reading skills. The game runs reliably on desktop and laptop machines. The game can be published to Facebook, web portals such as Kano Games, or even Steam. To our knowledge there are currently no mobile FlashPunk games that are published, making that specific venture shaky, but potentially viable.

2. Game Structure

The game structure implements the Model View Controller pattern. The model consists of the persistent and temporary game data, stored in the PlayerData and GameState objects in Game, which include the terrain at the hex level and the instruments in the observatory. The player experiences the model through the Views, each of which implements NeptuneWorld and is used for a different section of the game. To deal with subsections within a View, many Views are broken down into a stack of Displays. Displays and Views extend the World Class in Flashpunk which is used to hold Entities, objects in the game with locations and images associated with them.

The Controllers are different depending on the view. Complex views like the HexView and MapView have several controllers associated with them and use a different one depending on the player's current activity. In this case, the correct controller is created by a ControllerFactory depending on the GameState. For simpler views, no controllers are used and the mouse clicks are managed by the Entities on screen.

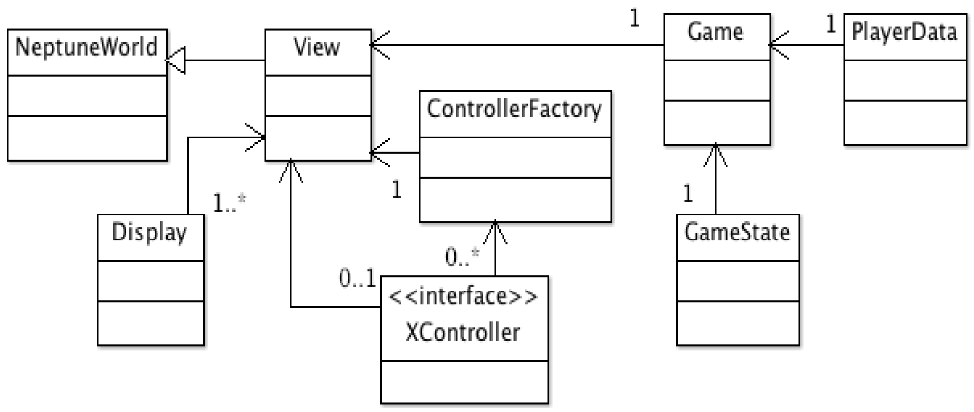

Fig 2.1: Game Structure

2.1 Views and Transitions

At any given time, three states are relevant to the player and dictate what options are available to them. These states are the current View, the current controller for the view, and the GameState in Game. The GameState is primarily used for passing important information during the transitions between the worlds, so the player actions and game flow are better understood by looking at the worlds as shown below. The controllers will be discussed later as each View is explained independently.

The player experiences the game and interacts with it through the Views and Controllers, so the active View, Controller, and current GameState. These can be seen as a set of states that the user transitions between throughout the game and are illustrated as a finite state machine below.

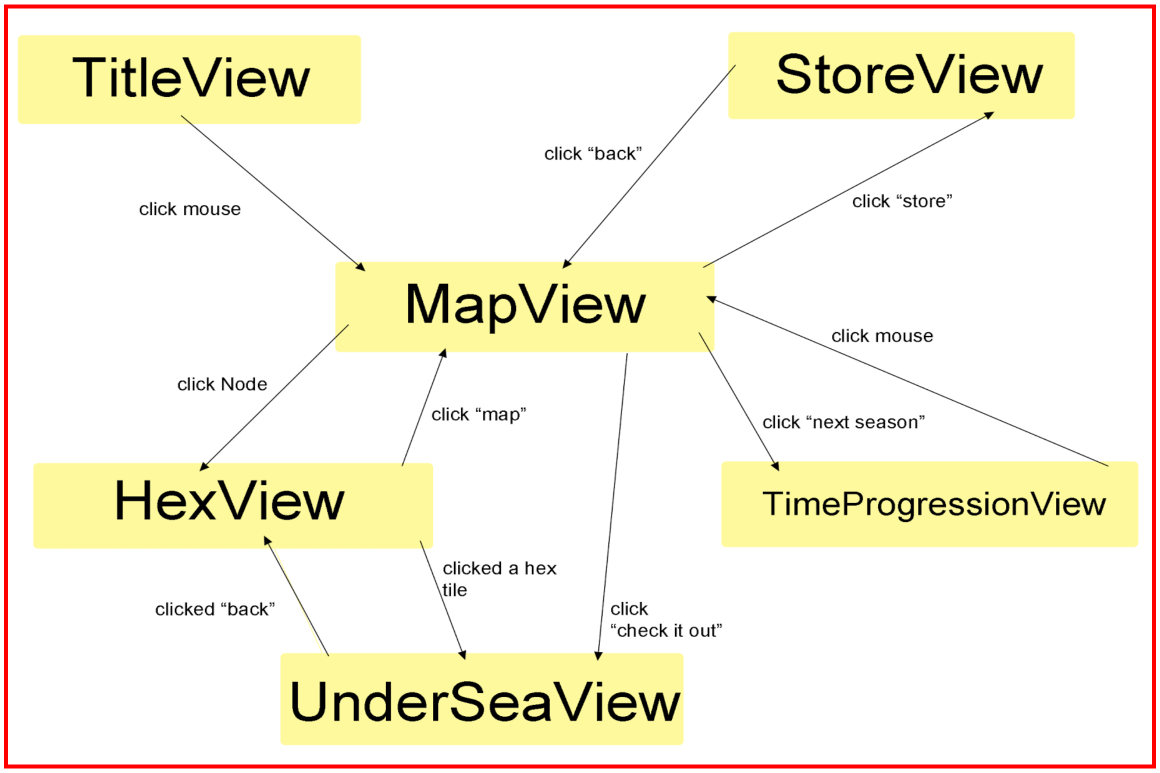

Fig 2.2: Game View Transitions

The first thing to note is that each of these files is referred to as a view. Every time a view changes, the state of the game has taken a transition on the diagram above. Each view is created on the transition to it, taking the persistent information across views as a Game object in its parameter. Views can have their control modified by displays which go on top of them, but these are still in the same View so when the display is closed, the player is exactly where they were. Displays will be discussed later in the Displays and Controllers section.

The player enters the game through the TitleView and upon clicking the screen, transitions to the MapView. The MapView can be seen as the centre of the user experience. Season transitions take place here, and all other views lead back, directly or indirectly, to MapView.

From the MapView the player can navigate to the StoreView by clicking the "store" button. In the store, they are able to buy new ObservatoryComponents. From the StoreView, they can return to the MapView by clicking the “back” button.

There are two transitions from the MapView to the HexView. The first is simply clicking on a Node to view the HexView. When the player clicks a Node, the player transitions to the HexView with the camera centred on the selected Node. The player is then free to navigate the HexView and can transition to the UnderseaView by clicking on a hex. The second transition occurs when placing an instrument (not a Node). When placing an instrument, the player is prompted to select a Node to connect to. Once they click a Node, they transition to the HexView but the Controller in use will be an InstrumentPlacer rather than a TileViewer, meaning that they will be unable to go to the UnderseaView until they have finished placing the instrument. Once again the UnderseaView has a “back” button that returns the user to the HexView, and the HexView has a “map” button that returns them to the MapView.

The only other notable transition is the interesting data event. If an instrument is recording interesting data, a window pops up at the start of the season that allows the player to jump to the instrument recording the data by clicking “check it out”. This takes the player directly to the UnderseaView leaving them to get back to the MapView through the normal process.

2.2 Game Data

2.2.1 Dynamic Game Data

Two structures hold the dynamic information in the game: the state (GameState), which holds temporary information, like an instrument being placed or a hex tile being viewed, and the data (PlayerData), which holds persistent data like terrain for known hexes, the player’s inventory, and the list of nodes and instruments placed on the map. When deciding whether an object should be saved in the state or the data, the key question is whether the user should be able to end the process they are engaged in, go to another screen, and do something else, and then come back to find the object exactly as it was, ready for them to pick up where they left off. If so, then the object should be in the data. However, anything that is meaningless if the action associated with it ends should be stored in the state. When saving and loading are implemented, only the data needs to be saved and reconstructed for the player to resume their experience. Whether a piece of data should be lost when the user ends a session, or preserved, is a another litmus test for where the variable should be placed.

For example, the instruments in the inventory and the instruments in the observatory are stored in PlayerData while the InstrumentBeingPlaced is stored in the GameState during the placement process. However an instrument is not removed from the inventory until it is placed on the HexView, so if the game were to crash during the placing process, the player would restore the game and find the instrument still in their inventory, but no instrument being placed. They can pick up where they left off by selecting the instrument again for placement and restarting the process.

2.2.1.1 PlayerData

The PlayerData class is responsible for holding all the persistent data for the game. This includes single values like the player’s current amount of money, or the seasonal count for the number of nodes or instruments placed. More importantly, it holds the lists of instruments, hexes and map features that drive the game. They are as follows:

● instrumentsInventory:	Holds all purchased but unplaced instruments
● storeList:	Holds all instruments available in the store
● nodeList:	Holds all the nodes currently on the map
● hexData:	Holds the list of existing hexes in the HexView
● unresolvedTerraianForces:	Holds all the terrain forces
● terrainFeatures:	Holds all the features on the map

ObservatoryComponent is a blanket class used to include any object that can be placed on the seafloor. Instrument, Node, and Connectable all extend this class. An ObservatoryComponent has a ComponentData object which it uses to answer any queries for static information about the component, like its name, image, or visibility. This way, each ComponentData defines a single kind of instrument, while each ObservatoryComponent represents an instance of some kind of instrument, including information about its location and what other components it is connected to.

The complete observatory is represented in the NodeList. Each Node is the root of a tree of instruments and Connectables, ObservatoryComponents that can have further components connected to them.

InstrumentsInventory and storeList hold ComponentData objects. They use this instead of ObservatoryComponents because the instruments have not been instantiated yet.

The HexData class is used to hold all the information for a given tile, namely its terrain, any ObservatoryComponents that are placed in the hex, and a boolean for whether the hex has been discovered or revealed to the player. When the HexView is created, it creates HexTiles to match the entries in HexData as they appear on screen.

UnresolvedTerrainForces, as will be discussed later, is part of a system that should be removed in the next iteration of the game. The idea is that features on the map exert an influence on the choice of tiles for a given hex called a TerrainForce. Forces are created when a node is placed and are applied when nodes are discovered, but until then, they are stored here.

2.2.1.2 GameState

The game state is responsible for holding and sometimes updating the temporary values that are independent of the views. Most of its functions are for the instrument placing process in which the state saves the instrument being placed, and the connectable it is being added to. It is also responsible for managing the music selection and the event popups on the MapView. It also had some use in the terrain force propagation process, but the system is not functioning properly and its replacement is recommended in section 3, so the details of the GameState's role in it will not be discussed here.

2.2.2 Static Game Data

The static data is kept separately in two files. GameConstants hold all static numerical data. This includes the map sizes, the player's initial income and the hex to map pixel ratio. GameTables is more complicated. The file is so named because it is intended to stand in for a database.

2.3 The Builder Pattern or Fluent Constructor

Because many of the classes used have a large number of parameters, the game uses the Builder pattern to simplify the construction process. For a class with a large number of paramaters, there is a companion builder class with member variables that correspond to the parameters of the first class, and methods to set each of them individually. It also has a build or finish method that plugs in the parameters and returns the finished object. The trick is that the setter methods all return the builder class so they can be chained together as seen here:

```
var subject:Subject = (new Builder().setA(1).setB(2).setC(3).build());
```

The default values for each paramater are set in the builders constructor. Virtually all uses of this pattern correspond to an object that should be created from attributes in a database in GameTables.

2.4 Displays and Controllers

In order to provide subsections in the UI, Views have the capacity to contain several overlapping displays. For instance the MapView has a control panel that displays the player’s money and the current year and season, the inventory display that provides the components available for the users to place, and both are drawn over the map display that shows the map and the nodes placed on it. On top of this additional displays can be added for events or player prompts without changing the View.

Controllers allow different states to have different interactions with the View. Again, using the MapView as an example, there is a controller for placing a node, a controller for placing an instrument, a controller for viewing an event display, and a default controller for viewing the map. Meanwhile the controller used to view the map allows the user to click any of the buttons in the MapView as well as the nodes on the map, and transition to another view accordingly.

2.4.1 Common Displays

The data display gives the user information about their current status. This displays the number of the year and the season for determining how much time they have until a given event. It also shows the user how much money they have, and is currently the only place where that number is displayed. It appears on the Map and Hex Views.

The instruction display appears when the view is first created and when triggered by a player action, such as pressing an incorrect button or triggering a new phase of instrument placement. It only stays for a few seconds before it goes of screen and is removed. Ii appears throughout the game.

2.5 Events

2.5.1 The EventDisplay Queue

One of the structures in GameState is the EvendDisplay queue, implemented as a vector. From anywhere in the program, a display can be added to this queue. When the player enters the MapView, the first entry is popped of the queue and displayed to the player until they close it, at which point the next entry is popped and so on until the queue is empty.

Currently, only the events use this feature during season transitions so load up the messages for each new season.

2.5.2 Seasonal Events

Seasonal events are used to give the player specific goals and walk them through the game. Each has three displays associated with it: the initial announcement display which is fed into the constructor as a String, the achievement display which is shown when the user has completed the event successfully, and the failure display which is shown when the time on an event expires without the success conditions being met. The success and failure displays are handled by their respective get methods which create the displays needed.

To check the success of an event, the wasAchieved method runs a custom algorithm to return a boolean indicating success or failure at the current time. The other sometimes used component of the GameEvent is the wasAnnounced method that allows the event to change PlayerData or other parts of the game information at the moment it is announced. This is currently used by the SpringBloom event to add the reef to the list of MapFeatures, while the BuyFirstInstrument event uses it to call the finishStoreList method in player data to add the instruments after the first season.

Most events are created in the SpecialEvents class adds pre-defined events to PlayerData. However, this happens after the start of the first spring, so any event that needs to be in the system at the start of the game is created and added in main instead. Because of issues like this, events are discussed again later in section 3.

2.5.3 Instrument Events

The instrument events are managed on a seasonal cycle by the InstrumentEventCoordinator and the InstrumentEventDispatcher. Each new season, the eventDispatcher checks to see if an instrument event will occur this season using the probability provided in GameConstants. If an event is happening, the instrument's startInterestingEvent method is called which replaces the boring data sample with one of the interesting samples for the instrument's dataType. Currently only one interesting event can happen at a time.

2.6 The Map View

2.6.1 Operation

The MapView shows the player where their nodes are on the map and what is currently in their inventory. Players can select an instrument in their inventory from the inventory display on their right. This can be either a node to be placed on the MapView, or an instrument to be connected to the observatory on the HexView which the player will reach by clicking a node while placing an instrument. The player can see their money, the number of data units being produced, and the current year and season.

2.6.2 Construction

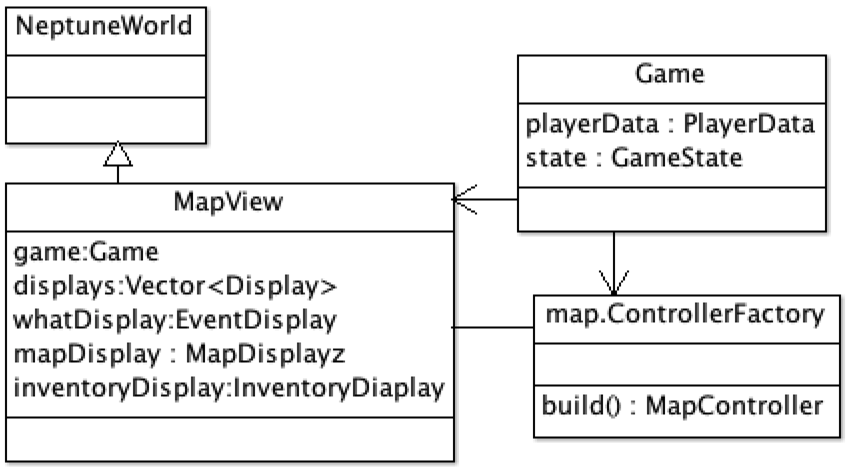

Figure 2.3: MapView Construction

On construction, the MapView stores the game into its member variable, then uses the game data when instantiating a ControllerFactory to produce the default start controller for the MapView, the HexViewer. Next it pushes onto he Display stack inherited from NeptuneWorld all of the displays it will immediately need. These are, in order, the MapDisplay, the InventoryDisplay, the DataDisplay and the InstructionDisplay. If there are events pending they will be detected when the engine calls MapView's ShowNextEventDisplay method.

This construction process and the accompanying UML are an accurate representation of how most views are constructed.

2.6.3 Displays

The map display is used to display the map and the MapEntities that sit on it. At the moment, the only MapEntities are the nodes and the reef image that indicates its location to the player.

The InventoryDisplay pulls the list of instruments in the player's possession but not yet placed and provides them in groups of four for the user to select and place on the network as desired. This display also holds the "next season" and "store" buttons.

Event displays appear as needed and are retrieved from game.playerData until all of them have been read and dealt with by the player. The interesting data display appears last because it will allow the user to go to the undersea display.

2.6.4 Controllers

There are three controllers for the map view: the HexViewer and the instrument and node placers. The HexViewer is so named because it is used to go to the HexView through a node, or to view the hexes. If the player is not otherwise engaged in an action like placing an observatory component, this controller is used.

The other two are created when the player chooses a component from the inventory to place on the network. Selecting a node will switch to a NodePlacer, while selecting anything else will switch to an InstrumentPlacer. The key difference is that the NodePlacer adds the node to PlayerData as soon as the player clicks on the map, never leaving the MapView.

The InstrumentPlacer requires the user to click a Node. This triggers the transition to the HexView where the player can finish placing the instrument.

2.6.5 Map Features

Map features are a way of storing information about the hex grid terrain in the map view. They have a LocationStructure that defines where they are on the map and the area they affect. They also store the terrain (at the moment a terrain force) that they apply to the hex grid when the tiles in their range are discovered.

The reef terrain uses a LocationPoint, which has an x and y coordinate. The isInRange operation takes another coordinate pair and calculates the direct linear distance between them. If the distance is less than the range, the coordinates are in range. A map feature uses this test to determine whether to apply a terrain force.

2.7 The Hex View

2.7.1 Operation

Selecting a node from the MapView brings the player to the HexView. The HexView is where the players connect instruments to nodes on the seafloor. This view also provides a clear feeling of what the sea floor looks like.

The sea floor is represented as a grid of hexagonal tiles. Each of these tiles has its own terrain and may contain nodes or instruments connected to the network,

2.7.2 Construction

The HexView construction starts with the view assigning the lastViewedHex in the GameState. This specifies the point on the tile map at which the player entered the view. This ensures that the camera's position is recalled correctly when returning to the HexView from a view at a lower level. The HexView then adds its displays: a HexDisplay, a HexControlPanel, a DataDisplay, and an InstructionDisplay. These are discussed in greater detail in Section 2.6.3.

The HexDisplay goes on to instantiate objects related to the grid of tiles, namely a HexFactory and a HexGrid. These are discussed in greater detail in Section 2.6.3.1. Additionally, a CableAdder is created. This object adds all of the cables connecting instruments to the network.

2.7.3 Displays

The primary display in the HexView is the HexDisplay. This display shows the hex tiles and makes use of different classes related to the hexagon space. The other displays can be considered supplementary.

2.7.3.1 HexDisplay

The HexDisplay shows the ocean floor and the structures constructed on it. This display contains the HexTile entities, as well as observatory components and the cables connecting them.

Where the HexData collection in the PlayerData represents the persistent model of the hexagon grid, HexTile instances act as the view for this data. These HexTile instances are responsible for drawing the terrain of HexData and creating the view objects for observatory components which have been placed there.

On request, the HexFactory creates a HexTile instance at a given point on the map. To do so, it must supply the HexData associated with that location. If no HexData instance exists for that location, a new, undiscovered HexData instance is created, added to the PlayerData, and used in constructing the HexTile.

The HexTiles shown in the HexDisplay are maintained in a HexGrid. This data structure stores the active tiles and provides accessors to areas of the grid such as all of the tiles which are currently on screen. The HexGrid creates HexTiles lazily such that tiles are only created when they are requested. This avoids creating all of the tiles in the map at once. Instead, it’s possible to create only the tiles which the HexDisplay must show the player as they explore the sea floor.

2.7.3.2 Additional Displays

The HexControlPanel display contains a button for returning to the map and shows the legend for the hex terrain types. This serves as a guide to the player as to the nature of the ocean floor presented to them.

InstructionDisplay is used in the same manner as it is in the MapView. Through this display, the player is provided with advice about how to play the game.

2.7.4 Controllers

Player input in the HexView is interpreted by implementors of the HexController interface. These controllers respond to the player selecting tiles.

If the player is not in the process of connecting a new instrument to the network, the controller for the HexView defaults to a TileViewer. When the player selects a tile, this controller causes the game to transition to an undersea view focussed on the selected tile, thereby allowing them to view the network at a lower level. If the player is connecting a new instrument, a more interesting controller is selected for the HexView.

Connecting a new instrument to the existing network is a two stage process. First, the player must select the node or junction box to which the instrument will be connected. This establishes the start of the connection. After this, the player must select a tile with free space on which the instrument is to be placed.

If the game state indicates that the player has selected an instrument to place, a ConnectionStarter will be created as the controller for the HexView. When the player selects a tile, this controller checks whether they have selected a node or junction box from which a connection could be started. If this is the case, the ConnectionStarter sets that object as the start of the connection and replaces itself with an InstrumentPlacer as the HexView's controller.

Where the ConnectionStarter begins the instrument placement action, the InstrumentPlacer completes it. When the player selects a tile, this controller checks whether it has enough space for the instrument being placed. If it does, this controller connects the instrument to the array and sets the HexView's controller as a new TileViewer.

2.7.5 Terrain Generation

Hexes have two states once they are initially created. At the start of the game, none of the hexes exist. When the player views the HexView, each tile they see is created, but not necessarily discovered. A discovered tile has its terrain permanently set. An undiscovered tile has no terrain and simply appears as an unknown tile in the HexView. When a node is placed, the tiles around it are discovered, and their terrain values are set appropriately given the coordinates of the node.

When a node is placed, the area around it is explored, generating HexData objects to be added to PlayerData. The program starts at the node, then discovers all the hexes adjacent to it by tracing a circle around the start hex. At the end of that circle, it increases its radius and travels in another circle around the currently discovered hexes, discovering each hex as it goes. This process is repeated until there are three layers of discovered hexes around the node.

When a hex is discovered, its terrain is set based on the terrain forces that are applied to it. Currently, the only possible force is a reef force, which will make the tile a reef tile. All other discovered tiles are mud by default. The terrain forces are propagated to each tile through a process similar to discovery, using the same circling technique, but this is not described in detail because it does not function correctly, and is among the list of things that must be replaced in the section 3 of this report.

2.7.6 Connections and Wiring

To be added to the structure of active instruments, an instrument must be connected to a node. The Node class extends Connectable, and all checks for whether an instrument can be hooked up to an object reference (or should reference) Connectable rather than Node. This is done to incorporate junction boxes in future designs as will be discussed in section 3. The reason node is in a further extension of the class is that it is placed on the map instead of the hex grid and its placement adds a new entry to the node list instead of connecting to an existing node.

The wire drawing occurs in two ways. The InstrumentPlacer in the hex grid creates a dynamic line that tracks the mouse, then adds a static wire to the grid upon placement. All other wires are drawn in the hex display. After all the instruments have been drawn, the a WireAdder is created which goes throughout the nodes and for each of them retrieves the wireX and wireY for each sub entity if an instrument connected to a node. The nodes coordinates are retrieved by querying the space converter with the node's stored mapX and mapY. Then a line is added between the coordinates at the centre of the node tile, and each of the instrument coordinates

2.7.7 Visibility of Instruments on the Hex Grid

For demonstration purposes, all instruments are currently set as being seen on the hex grid. However, the intended purpose of this variable is to make nodes, junction boxes, instrument platforms, and stand-alone instruments appear in the HexView, while instruments on platforms will not. In addition, nodes are also clickable by the instrument placer and junction boxes will be as well, but they are the only components that should be.

2.8 The Undersea View

2.8.1 Operation

The UnderseaView allows the player to investigate the instruments which they have placed in the HexView. An UnderseaView is specific to a unique hex tile. ObservatoryComponents which are placed on the UnderseaView can be clicked to open new displays which will give additional information about the instrument. Players will typically navigate to the UnderseaView to learn more about the instruments they are placing or check out an interesting data sample.

2.8.2 Displays

The UnderseaView is a relatively simple component of the game consisting of three displays. The UnderseaDisplay is always active and created immediately with the UnderseaView. The UnderseaDisplay positions any undersea ObservatoryComponents that are on the selected hex on the sea floor as buttons. Each button uses the clickedComponent event to create a NodeDisplay or a InstrumentDisplay depending on the ObservatoryComponent in question.

The InstrumentDisplay displays the name, description, and data description of the instrument, as well as any sample video or images to go along with it. The NodeDisplay, though unimplemented, would serve a similar purpose.

2.9 The StoreView

2.9.1 Operation

The player uses the StoreView to purchase items in the game such as Instruments and Nodes. The player can easily navigate through all store items. Selecting a store item gives the user additional information, and the ability to confirm purchase. Purchased items are automatically added to the inventory in the MapView.

2.9.2 Displays

Upon construction of the StoreView, two important displays are initialized immediately: the PurchaseButtonsDisplay and NavButtonsDisplay. PurchaseButtonsDisplay is responsible for pagination of the ObservatoryComponents (via the ButtonPaginator) that are currently available for purchase in the store. NavButtonsDisplay provides ‘left’ and ‘right’ buttons to navigate through paginated store items, as well as a ‘back’ button to return to the MapView. NavButtonsDisplay is given a reference to the PurchaseButtonsDisplay to allow this functionality.

PurchaseButtonsDisplay uses the ButtonPaginator to create a display filled with PurchaseComponentButtons. When clicked, PurchaseComponentButtons either open a PrePurchaseDisplay or InsufficientFundsDisplay, depending on whether or not the player has enough money to purchase the ObservatoryComponent.

The PrepurchaseDisplay displays the name, description, cost, and data type of the ObservatoryComponent, as well as a ‘buy’ button that allows the player to confirm the purchase. InsufficientFundsDisplay simply states that the user cannot afford the item.

2.10 Season Transition View

2.10.1 Operation

When players are satisfied with the current state of their ocean network (no desire to purchase or place any ObservatoryComponents) the player selects “next season” to continue the game progression. Transitioning from season to season will generate money for successful data collected during the prior season. Upon entering the new season the player may be prompted with special events.

2.10.2 Construction

When the user clicks the “next season” button, they move from the MapView to the TimeProgressionView which manages the transition between seasons. It performs several calculations and operations behind the scenes, but the user just sees some stats relating to their performance and data collection and will return to the MapView by clicking the mouse anywhere.

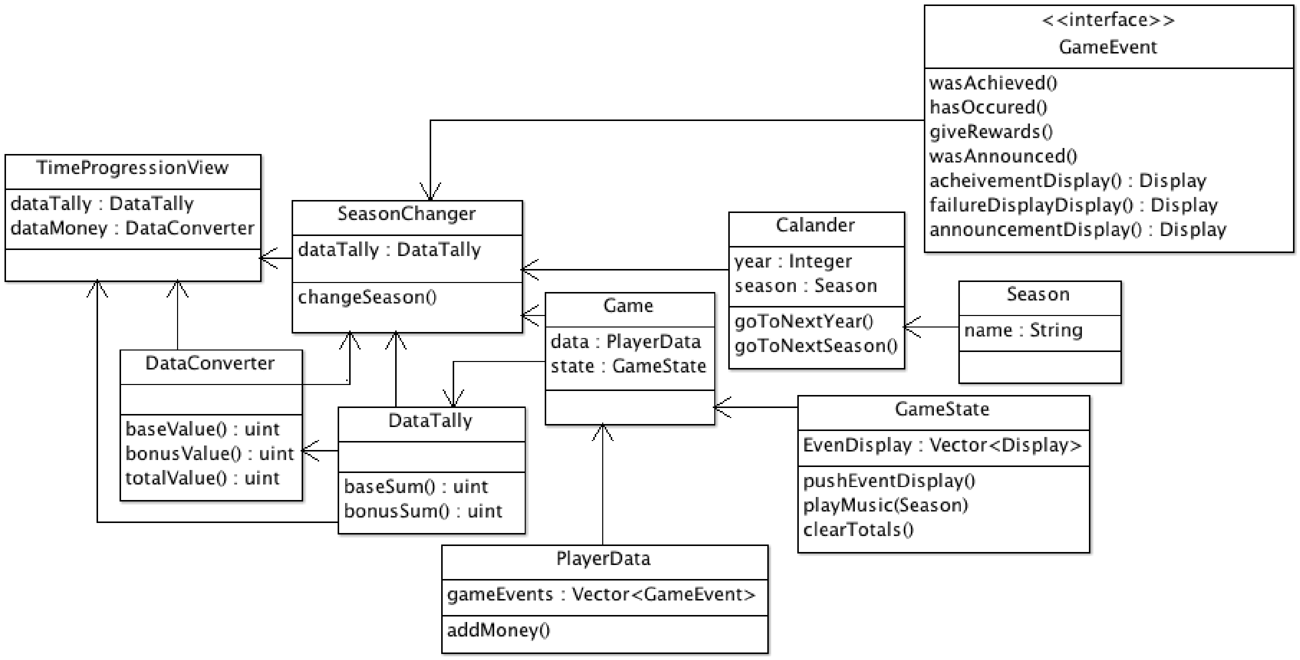

Figure 2.4: Season Transition Construction

Behind the scenes, the first step is to create the SeasonChanger which ends any special events on the instruments before calling the calendar to get the new year and season values. The endSeason method called on the instruments also grabs a random sample of boring data for the instrument to display, except for the camera which will always display boring reef data if it is on a reef tile. Next it checks for any world events that may be have the new season as their deadline. The pop-ups from these events are put on the EventDisplay queue so that they will be displayed in the new season. After that it calls the EventCoordinator to check if any interesting data should be created and shuffle the DataSamples for each instrument. All that remains is to calculate the money gained this transition using the DataTally object which counts up all the data units being produced as well as the number of active instruments. In the SeasonChanger, the DataTally counts all the dataUnits so they can be sent to the DataConverter which will calculate the money earned from the data. This total is added to the players stockpile in PlayerData and the SeasonChanger is done.

Back in the TimeTransitionView, another DataTally and DataConverter are instantiated, this time to display the number of instruments producing data, the number of data units gathered, and the money earned. If the EventCoordinator assigned interesting data to one of the instruments, the bonus value will be greater than zero and an extra message will be displayed in the TimeProgressionView.

3. Growing the Game

3.1 Recommended Iterative Changes and Additions

3.1.1 UI Changes

Buttons should be altered to provide a number of new functions. Firstly, there needs to be a way to prompt the user to push a button, such as the button flashing or displaying a moving pattern. This needs to be on a switch so it can be turned on or off for each button as needed. Buttons also need to respond more to the user. They should have a reaction when the user mouses over them, like some form of highlighting, to indicate that they are clickable. They should also have a pressing animation when the user clicks them to provide more tangible feedback.

The current tutorials have too much text and have the instructions for the user at the end of the paragraph instead of standing on its own. The text should be shortened to only provide the point and the InstructionDisplay should be used instead to give the users a series of scripted prompts at the start of the game. These should walk them through buying their first node, placing it, changing seasons, buying their first instrument, placing it, and viewing its data, and changing seasons again.

To make the user less overwhelmed by the interface, the game should be able to start with all the buttons removed and only add them as they are needed. The store nicely demonstrates this although a bug allows the user to circumvent it by going directly to the next season. At the start the only button should be the “store” button. In the store, the only button should be the node, then the “back” button will appear after they have bought the node. When they return to the map, they will find their node available for selection, (and flashing for good measure) and only after they have placed the node will the next season button become available.

3.1.2 Events

The current event system is scripted to create a tutorial that shows of the functionality of the prototype. However, for long term use in the game, a more varied approach would be better. Ideally, a clear template for what an event needs could be created following the example of ComponentData for instruments and nodes. The details for several events should be put into a database as described later in 3.2.2, allowing for future events to have their attributes added to the db instead of the codebase. The system needs to be modified to have multiple events happen in a given season and have effects that hit many tiles at once. For instance, a small earthquake would have an epicenter hex, and would apply earthquake data samples to any instrument in a near enough hex that would be capable of detecting the tremor.

Another detail that needs to be better thought out is the sequence of events during season transitions. Currently, at the end of a season, the interesting events are dispatched and then the data units and money for the season are calculated and applied. This means that the user is paid for the data detected in a given season at the start of the season rather than the end. This is not technically an invalid way of doing things, but it clashes with the other information counting the number of nodes and instruments placed over the last season. Ideally, an interesting data event would first appear to the player in the popup on the MapView, then be mentioned and and applied to the data unit count at the end of that season.

3.1.3 Splines and Map Features

Classes to provide splines are already in the system though they were never fully implemented or tested. These classes are specifically made to create Bezier splines which are defined recursively. To find a point that is a percentage of the way from the start of the spline to the end, the program makes a new list of control points each of which is the same percentage of the distance between each pair of consecutive control points. This means that the new list of control points is shorter by control point. The process is repeated with the new list of control points recursively until the list has only one control point. This is the point the correct percentage of the distance along the full spline.

The purpose of the bezier splines in the code is twofold. They can be used on the undersea, hex, and map views to create curved wires which add more variety to the games visuals. They can also serve as the base for a location structure, although this will require developing an algorithm to determine the distance to the spline from an arbitrary point. Such a location structure would allow a more accurate tracing of the geological features of the map view or trace a curving migration route to use for placing whale or fish populations.

3.1.4 Text Displays

Because of the limited time available getting ready for the demo, the in-game text in many places was formatted by hardcoding newlines into the text as needed and testing to see of they properly fit in the frame. A text formatter was considered, but unlike monospaced text, the default text for flashpunk does not consistently use the same amount of horizontal space for letter (t’s and i’s are thinner). Most of the letters do though so a text formatter can and should be developed which can be given a width and will add newlines to the text so that no line exceeds the width.

In some places, the on-screen room will be insufficient for a very detailed description and in the interest of allowing as much information as possible, the instrument and data descriptions in the undersea and store views should be modified to allow scrolling over the text.

With these changes, the science team will be able to write the descriptions of the data and instruments and the game can handle the formatting automatically.

3.1.5 Undersea View Completion

The UnderseaView was not intended to be the very bottom of the view chain. Instead, one or more of the ObservatoryComponents available will be InstrumentPlatforms which will appear on the HexView and in the UnderseaView like all instruments do right now. However, rather than bringing up an InstrumentDisplay, clicking on the platform will open PlatformView which will work more or less the same way as the UnderseaView, but will show the instruments on the platform instead of the instruments on the tile. Instruments on the platform will not be visible on the HexView. Depending on the implementation, players may be able to place on a platform without connecting to a Connectable and instead will use a connection that is part of the platform's class.

In both the UnderseaView and the future PlatformView, there should be a far more complex image of the sea floor and the instruments. Special sprites should be made for the view and the wires should be drawn in this view as well, though they do not have to correspond directly with their appearance in the HexView. The instrument sprites, like the buttons elsewhere in the UI, should respond to the user mousing over them with some sort of highlighting. Lastly the background image should be able to change depending on the terrain of the tile and the tile around it.

3.1.6 Instrument Difficulties

The instruments as currently implemented are perfect machines that operate underwater indefinitely without ever having technical problems. Obviously this is not very realistic. Adding difficulties for the user to encounter and deal with would add to the game's variety and make it much more interesting. To implement degradation or technical problems, the various ObservatoryComponent classes could be modified to have a condition variable that slowly deteriorates until the instrument ceases operation. More specific technical failures could use a random chance of various parts of an instrument breaking. The potential failures and consequences for the data collected could be added to ComponentData along with the cost of repair. Because these operations were not built into the instruments for the demo version, future developers have a lot of freedom in deciding how to manage them.

One other interesting problem proposed was interference between instruments, for instance a light based sensor would pick up the lights for a camera in the same area. The simplest way to implement this is to check the other instruments in a tile, but depending on the size of a tile, this can make interference only happen within a very short range. A more advanced implementation could repurpose the hex traversing algorithm used for terrain discovery to seek out any instruments that might conflict with a given instrument. Again, the management of this system is up to future developers.

3.1.7 Efficient Instrument Placement

At the moment , there is no cost associated with an instrument beyond purchasing it. However, many of NEPTUNE's expenses come in the placement and cable laying processes. To simulate this, one idea was to display the cost of laying the cable along with the cursor during placement and vary the amount depending on the distance between the connection point and the new component. A vast change in the price could occur when the distance becomes far enough to merit the switch from serial to fiber optic connections.

To expand placement the capabilities of the player, a JunctionBox class should be added that extends Connectable. This would allow players to branch out from a node to nearby areas of interest, rather than needing to place another Node next to it. It also opens up interesting possibilities for players to control when to switch between serial and optical connections as JunctionBoxes could be chained to reach greater distances.

Nodes should also be modified so that they go through a similar process for their placement, although a root point will be needed either in the form of a connection point at Port Alberni, or an initial node.

For the demo, a hex could hold one node or two instruments, but there is no need for the instrument limit to be that low in future versions. The more important population limits should be added to the Connectable classes so that only so many instruments can be linked to a given node or junction box.

Another idea was to illustrate the problems in planning an efficient trip by having a boat move between the nodes and remain at a given node as long as the user continues adding or repairing instruments at that site. If the user performs an operation at another site the boat moves and adds a substantial cost for the distance it needs to travel. The functions of the boat could be expanded, but they would seriously alter the interaction with the game and are beyond the scope of this report.

3.2 Recommended Replacements

3.2.1 Terrain Implementation

The current terrain implementation is not very good. The idea was not terribly well conceived in the first place, although it did open up some interesting gameplay possibilities. It complicates the terrain problem more than it solves it and needs to be replaced with a simpler method. The recommended change is to have the mapping function translate the hex coordinates back to the corresponding map coordinates, then compare that to the feature list the same way a node does when placed. There will need to be new algorithms devised to create varied terrain but the simplest rule that could form the backbone deciding the terrain would be to give priority to the feature with the shortest range. This would give more power to the features which have the smallest region of influence will get the most out of the range they have.

The current system is also limited in that it only allows a tile to reflect one terrain feature. The new implementation could have tiles with multiple attributes, for instance a tile near the mid-ocean ridge might feature a rocky slope, a methane-hydrate formation, and a colony of deep sea life. All of these should be able to be reflected at the same time. Alternatively, since the hexes are meant to represent only a small area, they could continue to only represent one land feature, but have enough hexes given the scale that the terrain can be represented purely in single hexes.

3.2.2 Database Integration

GameTables has always been a stand in for a full database hookup. It should be replaced with the infrastructure to query a database that is distributed with the game in mobile versions, and queriable on the web for the web version. As the game grows, it should be possible to add instruments, terrain, map features, and data samples, all through the database alone, with no changes to the game. At the moment, a number of these pieces have escaped to other parts of the codebase and should be refactored to use the database instead. Objects that should be moved to the database are as follows:

- ComponentData
- DataSamples (boring and interesting)
- Terrain types (Currently in hex.terrain)
- Events

3.2.3 Flash Video

One of the difficulties encountered towards the end of the project was the inability for FlashDevelop to compile the Flash video assets due to lack of memory. This problem was never solved, and the demo got by by only embedding one video sample as proof of concept. A system will need to be devised by which the flash videos can be embedded individually without all having to be on the heap at the same time. If this is not feasible or desirable, the flash videos must be streamed from a url via an Internet connection.

3.2.4 Art Style

The game has a pixel aesthetic using an online tool to rasterize the graphics using select palettes. This procedure may be done dynamically in Flash to allow players to customize the look and feel of their game, from 8-bit retro graphics to high resolution sea photos.

This feature would provide more value to in-game currency by selling new rendering palettes. A seasoned pixel artist is required to fully realize the aesthetic presented in the demo.

4. Closing Remarks

The NEPTUNE demo builds on the popular game engine FlashPunk with lightweight, custom frameworks making the codebase stable and relatively clean. The project has met all of it's listed goals from the initial project report except for the feature which states the player will encounter challenges such as malfunctioning instruments or weather delays, and Scientists are present in the game but only as a nebulous crowd instead of as individuals as was imagined for in the early designs. All the stated milestones have been met, though here is still much work to be done to create a fully realized videogame. A brief refactoring period is necessary before any development push is made.