Team 2 Documentation

Space Hunt the Game: <http://web.cecs.pdx.edu/~victor32/SpaceHunt/>

SPRINT 1: Contribution Summary

**US-1**

Spacecraft Movement

Victor worked on calculating the movement of the ship in the direction selected. The ship was able to travel any distance in a direction that was a 30 degree increment. A Ship class was made for it to contain the move function. If the destination was outside the map boundary the ship should encounter a wormhole. A wormhole function was made to randomize the destination of the ship, should it go out of bounds. Since Rene was working on MapObjects it made more sense for him to have the wormhole collision, so he placed it in MapObject.js. Willis created a wormhole object that would be encountered whenever a user traversed outside the map boundaries. The Ship class left a place to check if it ran out of energy or supplies after making the move. Since Cole was working on running out of energy and running out of supplies, that space was left blank for him to figure out to how to execute that.

**US-2**

Game Configuration for Development

To keep track of locations of celestial objects for testing purposes, Willis created an array data structure to hold the locations of objects at each coordinate point in the GameMap.js file. This array could be initialized to any size specified in the configuration menu. Willis wrote all code in Setup.js to create a modal window that allows for game configuration. In index.html, a button was created that called an initialization function in Setup.js when clicked. This function dynamically rendered text fields and radio buttons inside a modal window to select starting values for location, energy, supplies, credits, wormhole behavior, gameplay mode, and map size. When the “Save Changes” button was clicked, values from user input were stored in a gameData object in the main.js file before the modal window was removed. After this, affected objects in Ship.js and PopulateMap.js were set to update their state from gameData to reflect the user’s selections each time a new game was started.

**US-3**

Running out of Energy

Cole worked on checking for if the energy levels fell below zero. In order to check the energy levels, a globally defined variable that tracked the energy level was required. This variable was eventually refactored into the ship class object in Ship.js. For the operation of checking the levels, a single function named checkedLevel() was used. If the ships energy level variable was less than or equal to 0, the function returned a string to index.html stating, “No more Energy...Game Over!” This string would then be alerted to the page. This function function also checked if the game was in development or user mode. If in development mode, the function would not alert. This function is located in Ship.js.

**US-4**

Running out of Supplies

Cole worked on checking for if the supply levels fell equal to or below zero. In order to check the energy levels, a globally defined variable that tracked the supply level was required. This variable was ultimately refactored into the ship class object in Ship.js. For the operation of checking the levels, a single function named checkedLevel() was used. If the ships supply level variable was less than or equal to 0, the function returned a string to index.html stating, “No more Supplies...Game Over!” This string would then be alerted to the page. This function function also checked if the game was in development or user mode. If in development mode, the function would not alert. This function is located in Ship.js.

**US-5**

Collisions

To handle oldSpice’s collision with objects on the map Rene wrote Collision.js. This file contains a single function that fetches the object on the map that is at the same x, y coordinate as oldSpice. Using inheritance and dynamic dispatch, each object on the map contains a Collide method which executes logic consistent with that object (e.g. Collide for an Abandoned freighter adjusts supply and energy values for oldSpice). In addition, Rene wrote MapObjects.js which contains each object that populates the map (Celestial artifacts, planets, BadMax, etc). Each object inherits (using prototypal inheritance pattern) from a base class called MapObject which has the Collide method that gets overwritten by derived objects.

**US-6**

Sensors

Huanhua worked on this part as an equipment of the ship. He added a scan button to the control panel. When this button is clicked, it will call a function from the sensor to find out what celestial objects are within an amount of of CP according to the sensor level from the current ship position. Then, when there are any objects found near the ship, it outputs a message to tell the user what have found, or outputs there is no found to the message board. This story involves all code in sensor.js, adding scan function to the ship.js and adding a button to the game interface.

**US-7**

Celestial Map

Trevor created the basic DOM structure for the celestial map. This included creating a grid container element to hold the grid structure of the map, and individual grid blocks to populate it. The initial implementation of the map was static and would populate every time a new movement submission was submitted by the user. Although the basic structure saw continual use in Sprint 2, the static implementation of the map was refactored in favor of a more dynamic map approach created by Huanhua. Files affected by this user story were index.html, style.css, and ux.js.

**US-8**

Persistent State

Cole contributed code towards the persistentState.js file, which was deprecated after Sprint 1. This user story was incomplete and went back to the product backlog.

SPRINT 2: Contribution Summary

**US-2.1**

Revised Spacecraft Movement

Trevor changed the formatting for movement input selection per the requirements set in sprint 2. This included the removal of the direction sliding input and form submit button for a set of 4 individual submit buttons calling a different direction of 0, 90, 180, or 270 degrees when selected. This gave the user an effective means of going up, down, left, or right. Trevor then added styling fixes to render the new buttons in a way that graphically made sense to the user. Files affected by this user story were index.html, ux.js, and style.css.

**US-2.2**

Game Configuration for Marketing

Willis decided to add a second modal window to the configuration menu defined in Setup.js to accommodate the additional setup parameters for planets and celestial objects. New objects consisting of sets of coordinates were added to the gameData object in main.js to store results from user input, allowing the user to add zero or more celestial objects at fixed locations on the map. In PopulateMap.js, Willis wrote checks to determine whether positions of planets and celestial objects should be random or chosen from the setup menu and instantiate new objects to populate the game map accordingly. Finally, Willis added bounds checking for all input fields in Setup.js to ensure that values entered were formatted properly.

**US-2.3**

Player Dying

Rene implemented notifying the player of how they lost the game/died. For events that trigger death (asteroid, badmax, energy <= 0, and supply <=0) a call to the GameOver method of the GameMode instance (named ctrecipe) is made with a corresponding “reason” argument. The reason argument is a string that is specific to the death event. The GameMode object is defined in GameMode.js and is intended to define win and loss conditions and include helper variables for developer mode. In addition, GameMode contains a “tickObjects” list with corresponding “tick” method that allows a list of functions to be called whenever we want to update the game state (for spacehunt a tick event is defined as an oldSpice movement event).

**US-2.4**

Celestial Gazetteer

Huanhua worked on this part as a guiding feature of the game. This is a scrollable list containing object information (name, coordinate) that the ship identifies using its sensor. When the ship does a scan, any objects identified will be added to this list for later reference. This story involves code in the main.js, designing of the Gaze list in index.html and style.css.

**US-2.5**

Nameable Persistent State

Victor worked on the nameable persistent state. After many hours of failure, with just storing the player’s name, Victor went to Warren Harrison for help. Spending a few hours, Warren discovered that the browsers had changed how the functions that are called when a browser/tab is closed or refreshed, he informed the team. Huanhua made a save button to trigger the persistent state for any game that clicked that button during their game play. Victor continued working on the persistent state incorporating the save button as the trigger. The persistent state used the global gameData object Huanhua made to instantiate the ship and map size. GameData was updated with all the objects that a map could possibly have, and their locations. All the map objects were saved to the persistent state as soon as they were created and place on the map. Since they would never move, we wouldn’t have to worry about updating their coordinates. This way, the gazetteer also repolutated in the same order from a continued game as when the player initially started the game. The ship’s location and status were saved when a player clicked the save button. After Rene released BadMax with it respawning after and encounter the map had to be saved again when the save button was encountered. Victor was finally able to get the persistent state to save upon closing the tab/browser by using both beforeunload and unload on the same function, no prompt, or alert worked but it would correctly update all the data on the persistent state for the appropriate player.

**US-2.6**

Encountering an Abandoned Freighter

Rene implemented the abandoned freighter encounter. Inside of the AbFreighter object’s Collide method in MapObjects.js, a simple loot table is created whereby based on a random number, a player is rewarded 10% of max rewards 75% of the time, 50% of max rewards 20% of the time, and 100% of max rewards 5% of the time. The freighter is removed from the map after the encounter to prevent multiple accesses of the same freighter.

**US-2.7**

Docking at a Space Station

Cole implemented the random possibility to possibly win more credits when docking at a space station. This feature is implemented in two functions in MapObjects.js in the space station section. This feature requires a development mode option to guarantee the game of chance. This radio button was added to Setup.js and a global boolean variable was used to capture this information. If in development mode, the function would call another PlayGameOfChance() to start the game of chance. If accepted, an alerted notified the user of the results from the game of chance. The function then updated the global ship credits variable in Ship.js and updated the levels on the UI. The winnings are randomly generated with a random number generator. When not in development mode, another random generator was used to give a 1 in 3 chance of being asked to play. If the user was lucky, the PlayGameOfChance() function was called.

**US-2.8**

Encounter a Meteor Storm

Victor made a new map object from the MapObject.js file and added a collision function to execute if the ship ran into a meteor shower. The collision function set the boolean variable isDamaged to true for the ship.

**US-2.9**

Collision with an Asteroid

Huanhua worked on this feature by putting a function to the GameMap.js to calculate if there is an Asteroid in the way of the ship, and then let the ship stop by the Asteroid if the ship is trying to bypass the Asteroid, so that reflects a collision with an Asteroid. This story involved code in GameMap.js and Ship.js.

**US-2.10**

Being Boarded by Bad Max

Rene implemented the BadMax encounter. Inside the BadMax object’s Collide method in MapObjects.js, a a random number is generated that determines which function to call: Escape(), Steal(), or DestroyShip(). The player escapes 50% of the time, BadMax steals all credits and ½ supply 30% of the time, and destroys oldSpice 20% of the time. If the encounter results in Escape or Steal, the encountered BadMax is removed from the map, and a new instance is created randomly on the map. If the DestroyShip method is called, the game ends.