**Game Report**

Game Back Ground:

The game is based on the 2013 Southeast Asian Haze, where the haze period was caused by large-scale burning in many parts of Sumatra and Borneo. The haze was notable for causing record high levels of pollution in Singapore where the 3-hour Pollution Standard Index (PSI) readings reached a record high of 401.

In this game, you play as a store owner, stocking up on gas masks at the start of each day, and trying to make a profit on the sales!

Gameplay:

At the beginning, the player starts with a sum of money and 3 stores. The player chooses the amount of gas masks to purchase, before assigning them to whichever stores at chosen prices. The day will start after confirmation of stock placement, counting down the day and simulating a situation where the haze is drifting into Singapore.

At the start of each day, the player gets to replenish their stock, and reassign prices and amounts to the store of their choice. The player also gets to move their stores on the map to how they see fit, depending on the sometimes in-accurate weather forecast or the dynamic haze movements.

The game ends after 7 in- game days, where the PSI readings will reach up to even 500 PSI. Success or failure will be determined if the player has earned a profit or loss.

Features:

* Fully randomized haze generation and gas type diffuse system.
* A forecast system that can read patterns and predict the weather.
* Buyers with their own unique preferences.
* Saving and loading of game.
* A singleton sound class.
* A system of money where buyers will buy from the stalls and you get to earn money.
* A singleton time class.
* The world is on a 2.5D grid.
* Tile Class
* Game States
* 2.5D camera with zoom in and out.
* Clicking Class

By Roland:

Haze:

The haze class consists of three parts, the generation of the haze PSI readings, the haze rendering, and the haze diffuse. The generation of the PSI readings was done with a sine graph, with Perlin Noise added in. The rendering of the haze was done in such a way that each tile would contain four tilechilds. All the tilechilds would be interlinked, and they would then render the haze according to its own haze reading, using a HSL to RGB converter for the haze colour. The alpha values of the haze would also be changed to how thick the haze is. Lastly, the diffusing of the haze throughout the tilechilds was done by linking all the childs, then using an application of cellular automata with a diffuse rule. The program runs by taking a value out of the generated sine reading and giving it a factor, than seed it onto the world where it would instantly diffuse itself and render.

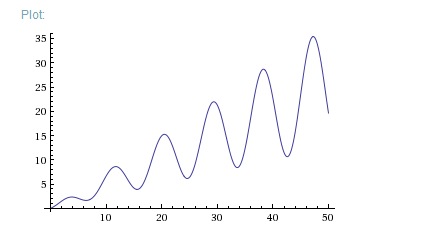


Figure Haze data generation pattern ½ x + ¼ x Sin[ 0.7 \* x]

Buyer:

The buyer class consists of two parts, the finite state AI itself, and a personality factor. The personality factor is randomized to create three different parameters; distance preference, haze value preference, and money preference. These preferences are just a ratio, and would ultimately be used to determine which shop the player would buy a haze mask from. The finite state AI consists of three states, **Idle walking, Walking to buy, Cannot Buy walking**. **Idle walking** just lets the AI walk to where it is targeted to go, and despawn. If haze conditions are right, it will decide to buy a mask and walk towards a shop, while changing state to **walking to buy**. **Walking to buy** lets the AI walk to the appointed shop. If the shop has masks to sell, it will buy a mask. Else, it would go to the next shop it can buy from. If there are no shops with masks left, they will change to **Cannot Buy Walking,** where they will just walk to their original target and despawn.

The formula used to determine whether the buyer will buy is n = 2 ^ (-Price \* price.preference – Distance \* distance.preference + haze \* haze.preference) \* 0.05.

Figure AI Buyer Finite State Machine Chart

Highscore Class:

Reads a file name called Highscore.txt and gets the high score list, if it has one. After that, it contains an insert function, where the player can insert their username with their high score and show it out.

Knowledge Applied:

Knowledge gained from all the modules we studied for the past 4 semesters has been applied into the making of this game.

By YingTzu:

Stalls and Money Class:

The money system that the buyers can buy from the stalls and earn money.

If the player buys too many masks, it might cause a loss in profits at the end of the day.

At the end of every day, if the player does not manage to sell all the masks, the masks will not be refunded.

Button Class:

The main control of the whole game play is to use the mouse to click the buttons and the transition from states to states. The player can see the shop they have selected when they click on the shop texture.

States:

Menu:

Before the menu appears, the Singapore map will emerge before the menu appears. The background animation shows a haze simulation.

Tutorial State:

Newspaper animation and brief tutorial explanation.

Buy Mask State:

This state allows the player to buy the masks to sell by clicking the 3 buttons. The cost is stated at the top of the button.

Start of the Day:

This state allows the player to decide the number of masks and price to sell for every shop by clicking the buttons.

End of the Day:

Shows the player how much each shop has earned and total money at the end of the day, and how many masks were sold or left. However, the masks left will not be returned to the player.

Bar class:

Shows the amount of time passed in each day to the player.

Money Animation Class:

Animation of money flying up when a customer buys a mask.

Textures:

Most of the textures in the game (Background, buttons and shops).

By Wei Qi:

Forecast Class

This class uses a few algorithms.

First, it stores preloaded PSI values from the haze class.

Second, it checks the current day and choose an appropriate range of PSI that the current day would be.

As the days goes on, it checks the values and patterns of the previous days.  
If the values are going up, there’s a high chance to go up and a low chance to go down, and vice versa.

If there’s an up and down (or vice versa) pattern from the previous 2 days, it will consider whether if it will rise or fall using previous values.

Then using an appropriate range of PSI, it use a random number generator. The range with the best possibility will have a higher chance than the other numbers in the range while the numbers outside the range is best not considered since a forecast is not 100% correct.

It gets the result 3 times to get an average and return the average range it generated.

SaveLoad Class, SaveState and LoadState

The States reads information from .txt files and store them temporary in a SaveLoad class.

It will load the values inside the save file you choose and the game will use it.

When the player saves their progress, it will overwrite and show the new information.

Models

The models of the stalls and people were done using simple shapes using maya. As well as UV mapping the stall while someone else did it for the people.

Buying during Gameplay

Using a simple function, it will add more masks to the current shop selected and deducts money.

It shows the current shop’s information like the price of the mask and the remaining amount of masks. Also shows the current money the player have.

By Samuel:

GameStateManager and InGameStateManager

The game state manager consists of a vector list storing all the game state instances and allows changing of the states. The in-game state manager stores all the in-game states in a vector list and allows changing of the in-game states to and fro while in the PlayState.

Camera

The game uses a 3D third-person camera that rotates around the point it is looking at and is restricted to the grid. The player is able to move the camera with WASD, zoom in and out by scrolling on the mouse wheel and rotating the camera by holding the right mouse button while moving the cursor. Changing the tile size and tile numbers of the grid changes the movement restriction of the camera, such that the player can never move and look away from the grid.

Mouse picking and Stall selection (placement)

The game uses colour picking to check if the player clicks on the stalls and the tiles. Basically, whenever the player clicks the left mouse button down in the GamePlayState, the game will change its rendering into its colour picking scheme, without changing the SwapBuffer, for a frame. In colour picking mode, only the grid and the stalls are rendered in its unique colour scheme whereas all other game objects are not rendered at all. Each tile in the grid has its own unique colour scheme and the stall’s colours corresponds to the tile it is on. While the game is rendered in its colour picking mode, glReadPixel is called on the cursor position and it gets the colour it picks on. Using a simple colour check, the game gets the tile it clicks on, black if it did not click on anything. The game does not call SwapBuffer for that frame so as to not change the render buffer, making the colour picking mode not visible to the player. If the player clicks on a stall, the stall will be selected and be able to be repositioned. If a stall is selected and the player clicks on a place-able tile, the stall will be repositioned to the selected tile.

EndGameState

The EndGameState is a state that implements the WinLose class for the winlose condition and the HighScore class for the list of highscores and displays them in a visual state for the player. If the player wins and obtains a new highscore, he is able to enter his username and store the highscore together with the username into the list of highscores.

List of the textures

gameTitle.tga

map.tga

start.tga

loadGame.tga

help.tga

quit.tga

background.tga

box.tga

box2.tg

next.tga

50.tga

100.tga

200.tga

go.tga

4.tga

5.tga

6.tga

back.tga

reset.tga

shop.tga

shop2.tga

shop3.tga

shopSelected.tga

shop2Selected.tga

shop3Selected.tga

pause.tga

skybox\_near.tga

skybox\_far.tga

skybox\_left.tga

skybox\_right.tga

skybox\_top.tga

skybox\_bottom.tga

Gress\_Texture.tga

x.tga

background.tga

save.tga

win.tga

lose.tga

money.tga

back.tga

volDown.tga

volUp.tga

description.tga

MouseControl.tga

yes.tga

no.tga

tutorialStateNewspaper.tga

endgameStatehighscore.tga

endgameStatehighscore\_bg.tga

tutorialStateFirstBuyingMaskScreen.tga

tutorialStateSecondSetupScreen.tga

tutorialStateThirdGameplay.tga

tutorialStateLastEndDay.tga

buyerTex.tga

By Amos:

Sound Class:

The sound class is a singleton class that allows the control of sound files, such as the setting of file names, and the situational playing of sounds.

Sounds Used:

- Sims bgm

- Clicking sfx

- Cash Register sfx

- Stock sfx.

Win Lose Class:

This is a class that when called, will take in the value of money of the player at the end of the 7 days and compare it to the initial amount. Thus, if it is more than the initial amount, you win! And your score is placed in the highscores.

Help state:

The help state shows the player the background story of the game, and also allowing the player to change volume levels.

Grid Class:

This class is used to draw a grid on the screen to separate parts of the game map. It also returns the location of each separate tile on the grid, showing its exact location.

Entity Class:

This Class sets various items as entities and sets them with certain properties. This is used to find out details such as location and angle it is facing.

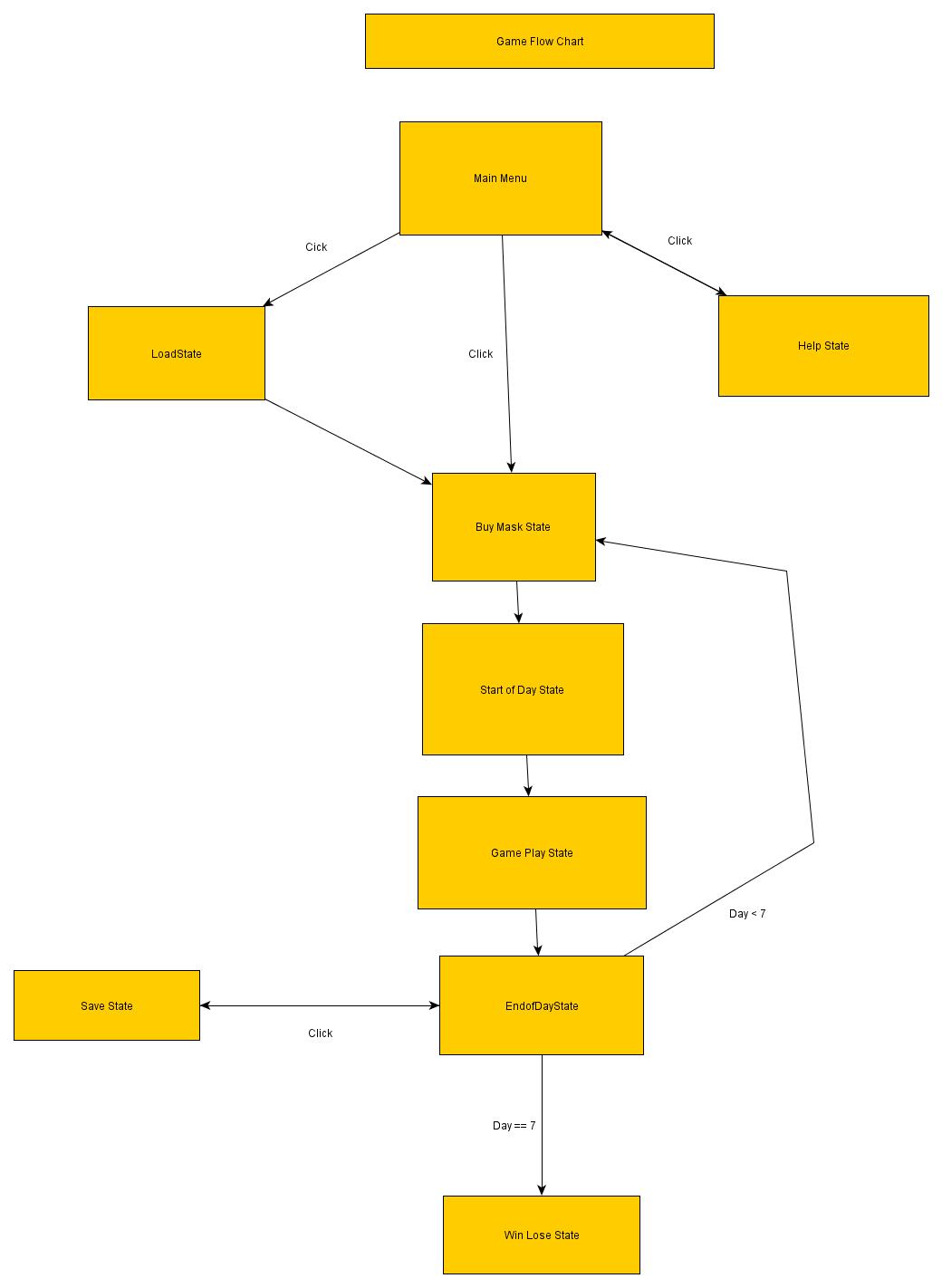
Problems & Challenges Faced:

- Did not know about singleton classes and how they are used or implemented.

(Solved by researching and asking those with more knowledge.)

- Limited Knowledge in programming. (Solved by researching and asking those with more knowledge.)

­­*Game Outline  
Flow Chart of Game States*



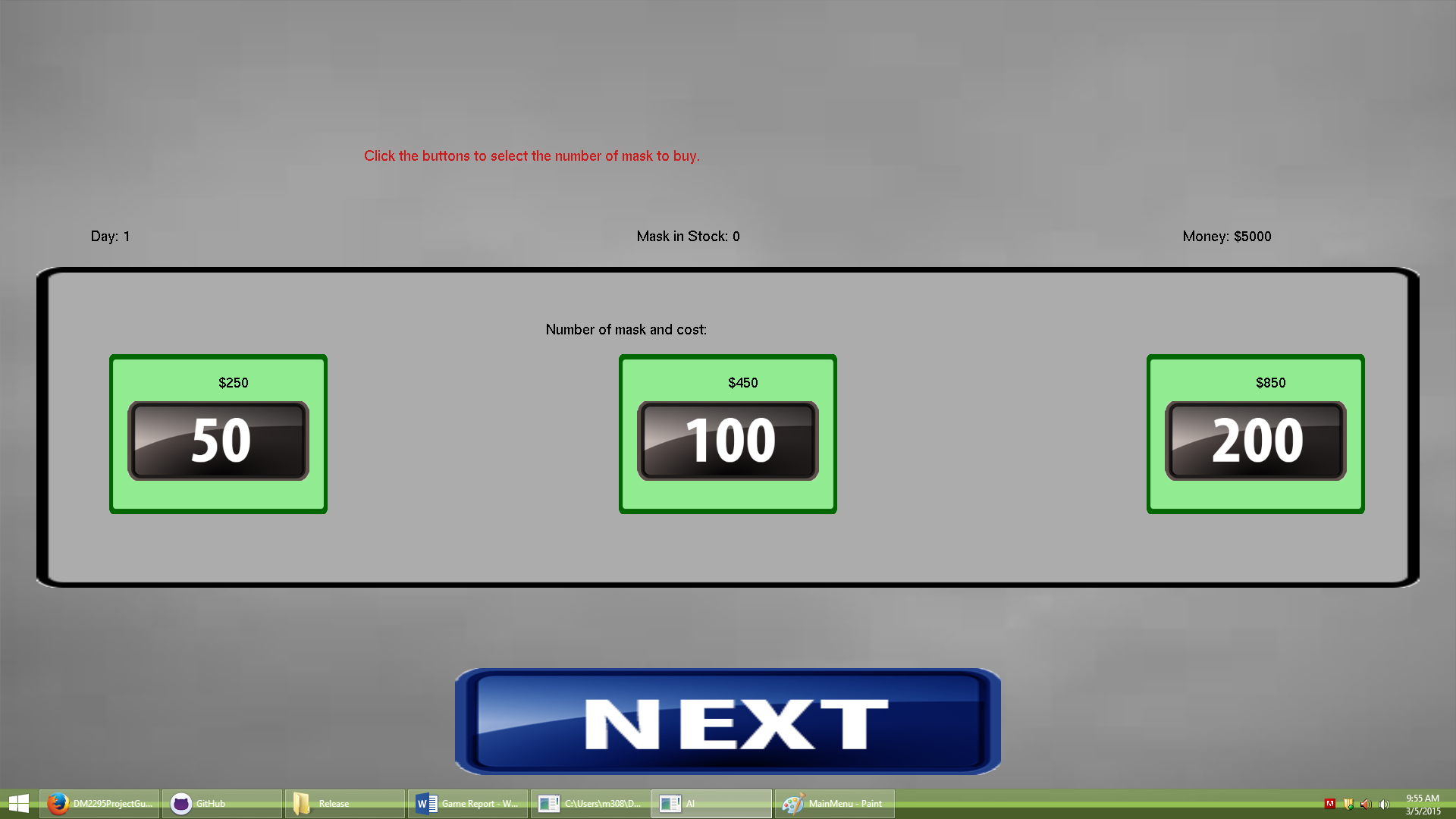




Figure Start of Day State

Figure Start of Day State



Figure 4 Game Play State



Figure End of Day State



Figure Help State

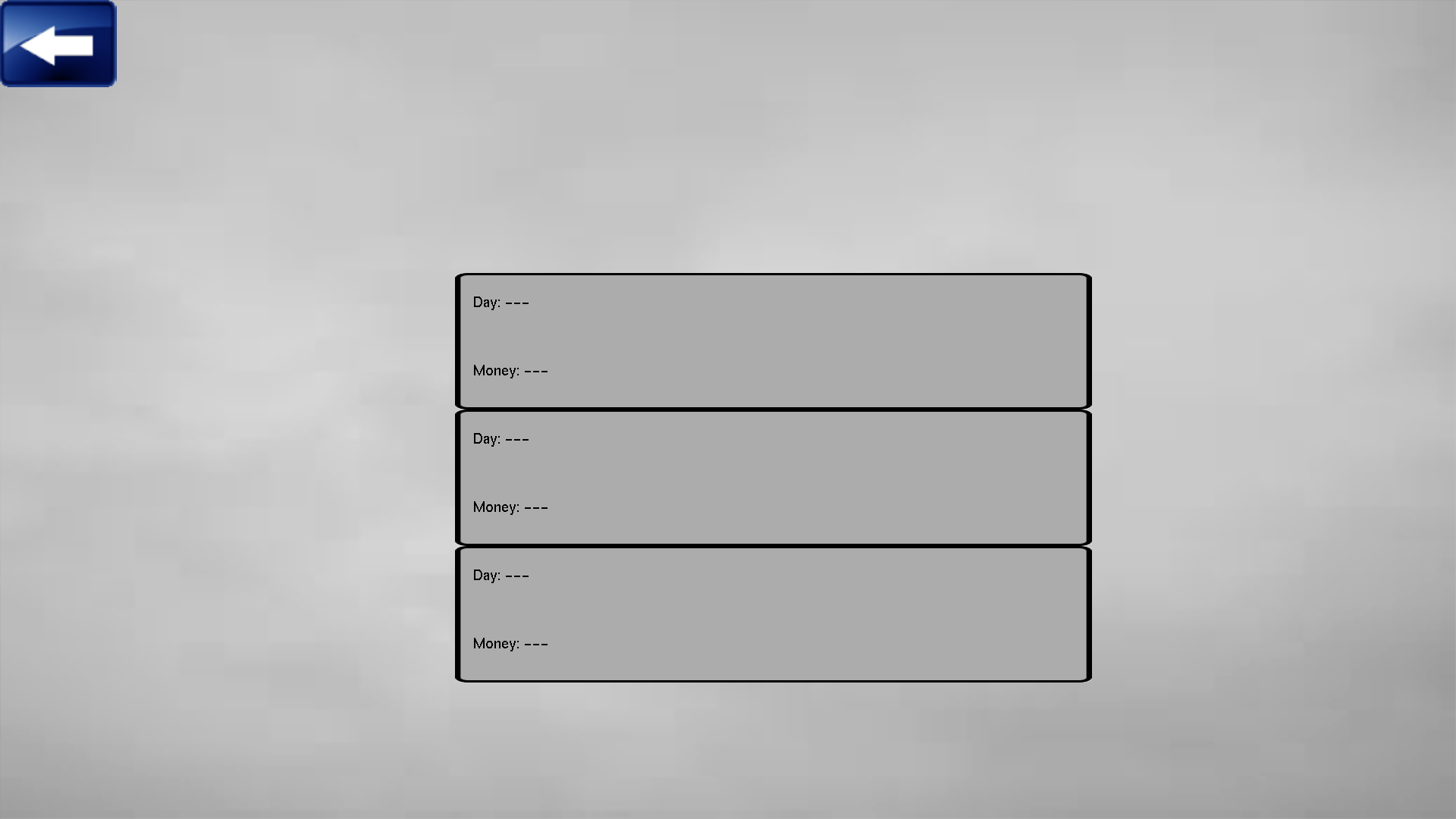


Figure Save State

Figure Main Menu

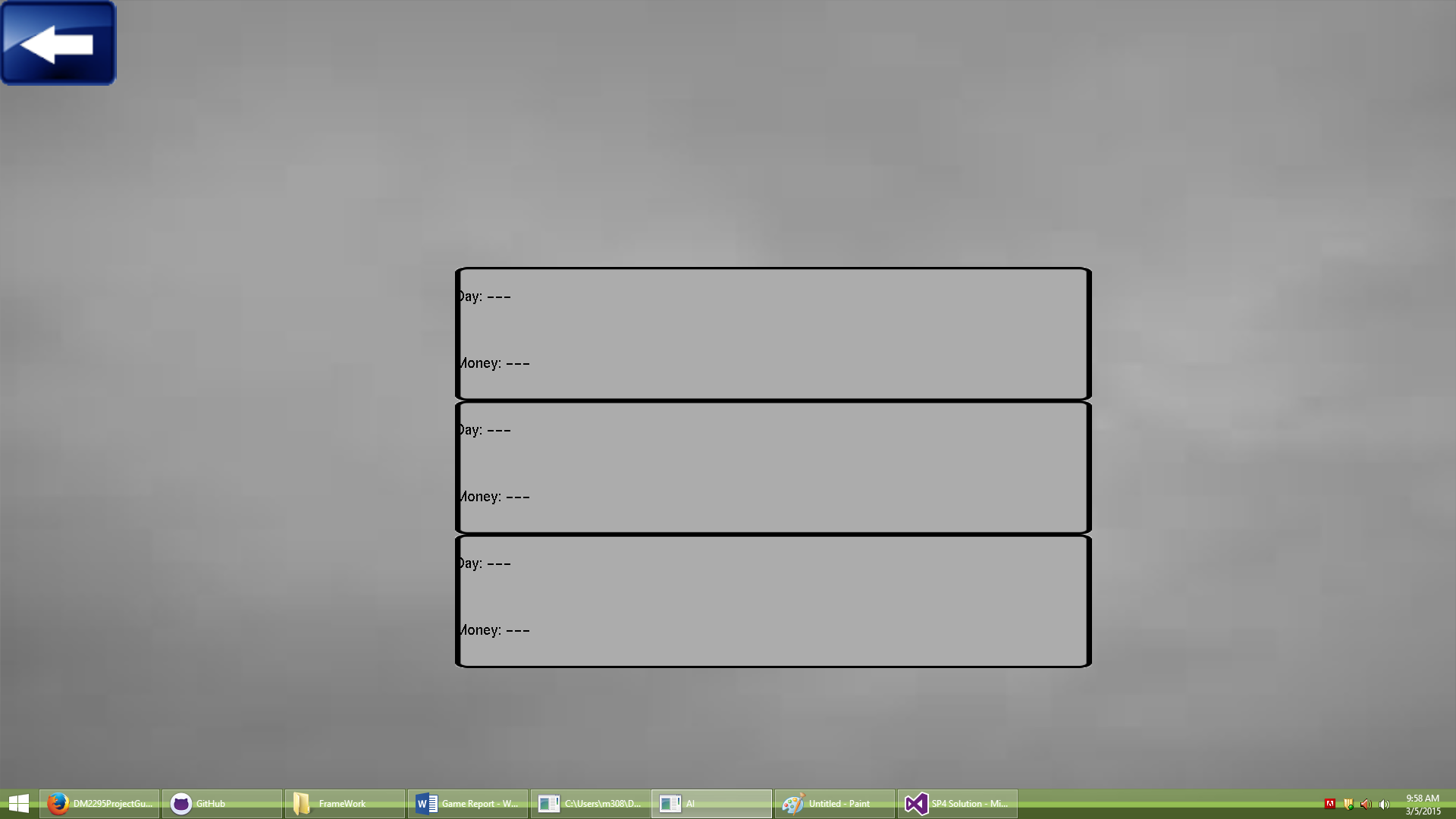


Figure Load State

***Problems Encountered/ Challenges***

Problems encountered by Team:

We had an original framework that could only support one stack of states. However, we needed another stack of states in order to be separate the game play logic from the rest of the game.

Problems encountered by Roland:

1. Very little experience with data generation in a patterned manner. Had to go internet to get the right formula in order to create the pattern for the haze data generation.
2. The rendering of the haze was also done differently from what was planned, because what was planned was not suitable. The plan was to render the haze directly onto the tiles in the grid, where everything was located. This resulted in the haze being very pixelated, or in the other extreme, my teammates had problem trying to link their things to the tile grids. There were two ways to fix this, create an entirely new invisible grid above the original one, or to use the tiles as a quad tree and go multiple levels in and create the grid. The difference between the two is that if it was done in a tree style, it would be able to link the tiles to the tilechilds(what we call the children that spawn from the tiles); thus we decided to do it in this way. Lack of foresight and experience caused this problem.
3. It was a huge challenge creating the personality class for the buyer, in that I had no idea how to do it. In the end, I took the model of a pie chart and randomized three portions. This way, a personality points would always add up to the same value.
4. Determining whether a buyer will buy from a certain stall was also a huge challenge, in that I had to create my own formula to determine whether the buyer will buy a mask from a stall. In the end, I asked the Mr.Sim for advice and was able to create an exponential graph with a threshold point in order to determine whether the buyer will buy. The formula used is n = 2 ^ (-Price \* price.preference – Distance \* distance.preference + haze \* haze.preference) \* 0.05.

Problems encountered by: Samuel

1. Having problems creating a separate game state manager and implementing it into a game state. Changing of states requires the current game state to not be deleted in case the player returns to the previous game state.
2. A huge problem with solving the colour picking on mouse click as swap buffer should not be called when rendering the colour scheme. In the end I created to booleans to check if the mouse is clicked and use them to check if the SwapBuffer needs to be enabled or disabled.

Problems encountered by Wei Qi

1. The idea of the forecast was confusing at first and required some time to formulate and break the idea down into smaller pieces. Also it needed to grab values from another class.

2. Sometimes the Forecast class will give unexpected results which is possible to get but I chose to put a range so the result won’t be too unexpected.

3. Sometimes some things gets changed and needed to find them.

Problems encountered by Ying Tzu:

1. Time bar did not run correctly when the game runs in other computer. With help of Roland and Samuel using delta time to make sure the value run correctly in other computer.
2. UI placement did not clearly show to the player what each buttons does.

With help of Amos, rearrange all the UI placement, explanation for some buttons will be show when the mouse move over to the button.