Project Fog
User Interface and Network Messages

Author

Julian Brandrick – A00789229

Table of Contents

Overview

This document contains the technical design details relating to the pure functionality of the general user interface and the handling of network messages. These topics will be subsequently separated into chat functionality, network message protocol, HUD components and menu functionality. The Project Fog game design document will be heavily referenced in this document.

This aspect of game logic is very closely related to the work done by the Network and Multimedia teams. As it is, a few assumptions will be made as to how much of these details belong to game logic.

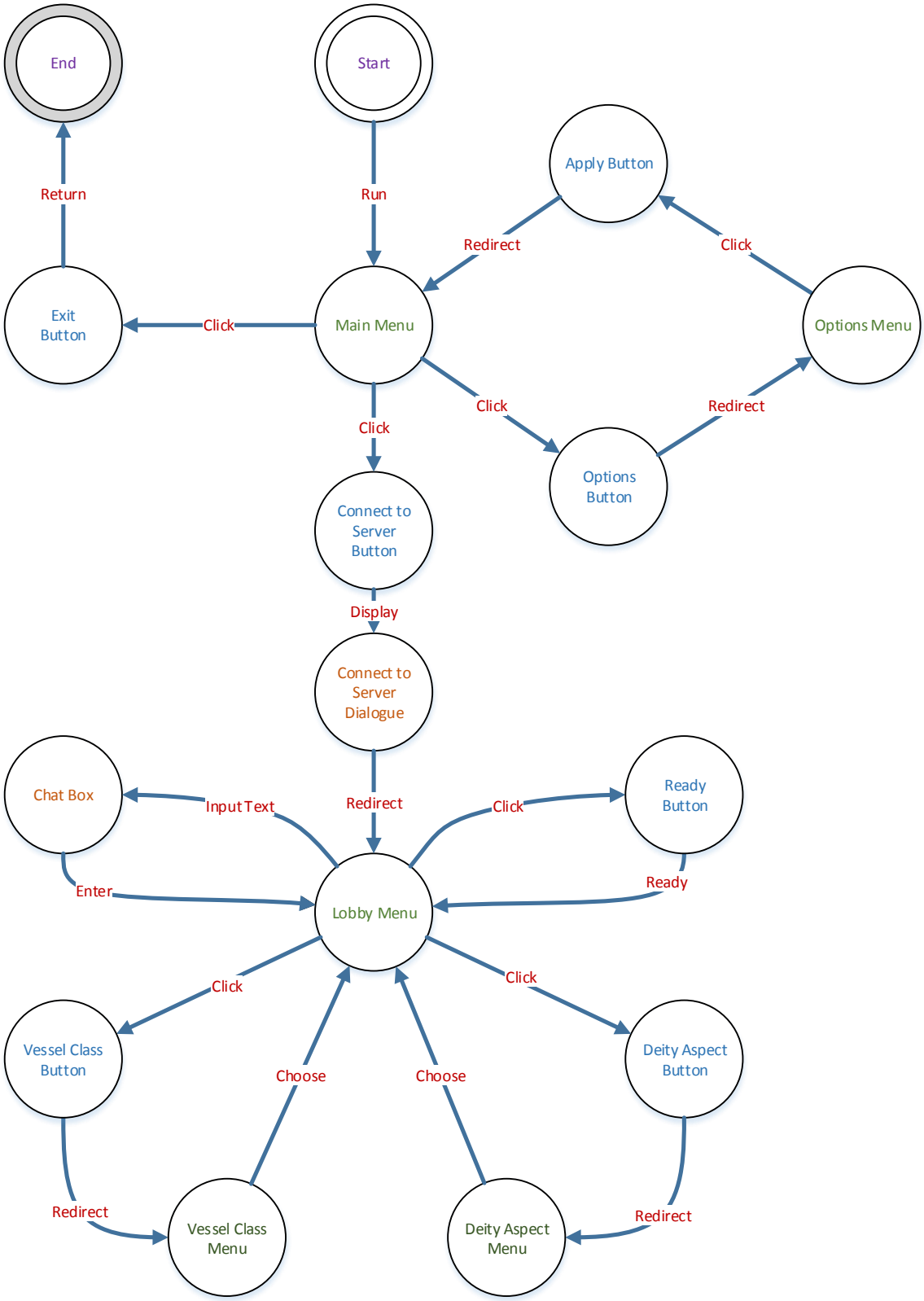
State Diagram

This is the state diagram for the flow of the menu. It will focus more on the functional elements rather than the visual.

Legend

State Name	State Description
Start	When the application is first initialized.
Main Menu	The main navigational page
Options Button	Redirects from the Main Menu to the Options Menu
Options Menu	A page that contains all of the adjustable technical settings for the game
Apply Button	Applies the settings the user has specified
Connect to Server Button	Displays the Connect to Server Dialogue
Connect to Server Dialogue	Allows the user to enter a user name and the IP address of a server
Lobby Menu	A page that allows the different users to chat, select their class or aspects and join a game
Chat Box	A text input box that allows the user to send text messages to other users connected to the same server
Ready Button	Sets the user's ready status to true and changes their ready symbol from red to green
Vessel Button	Redirects the user to the Vessel Menu
Vessel Menu	Allows the user to choose a Vessel class and ability
Deity Aspect Button	Redirects the user to the Deity Aspect Menu
Deity Aspect Menu	Allows the user to pick Deity aspects
Exit Button	Signals the application to start freeing its resources
End	When all of the applications resources are freed

Diagram


End
Start
Return
Run
Apply Button
Redirect
Click
Exit Button
Click
Main Menu
Options Menu
Click
Redirect
Click
Options Button
Connect to Server Button
Display
Connect to Server Dialogue
Chat Box
Input Text
Redirect
Click
Ready Button
Enter
Ready
Lobby Menu
Click
Click
Vessel Class Button
Deity Aspect Button
Choose
Choose
Redirect
Redirect
Vessel Class Menu
Deity Aspect Menu

Menu Functionality

The functionality of the menu can essentially be broken down into what kinds of buttons there are and what are their responsibilities. There are three types of buttons classes, all inheriting from a generic Button class.

Button Class

This is a generic button with no functionality on its own. It holds the buttons ID and name as well as the abstract ButtonClicked function which will run whenever the button is clicked.

Pseudocode

```
1.  class Button
2.  {
3.      variables:
4.          button_name;
5.          button_ID;
6.      functions:
7.          ButtonClicked ();
8.  }
```

RedirectButton Class

This is the simplest type of button. When clicked, a new page is displayed. It inherits all of the Button class's members and the abstract function ButtonClicked. It has a single native variable called page_link which holds the descriptor for the new page.

Pseudocode

```
1.  class RedirectButton extends Button
2.  {
3.      variables:
4.          page_link
5.      functions:
6.          ButtonClicked ()
7.          {
8.              Go to page_link's page
9.          }
10. }
```

UtilityButton class

This button type is the least solid. It handles all option setting changes for the specific client. This is currently the least solid since there are no concrete option settings. Despite this I can assume that the options menu will have several settings that can all be changed independently and will be saved with an 'Apply' button. It makes the most sense for this button to be an Information Expert and simply hold all of the settings data, at least for now.

Pseudocode

```
1.  class UtilityButton extends Button
2.  {
3.      variables:
4.          /*
5.           * Various settings
6.           *  resolution
7.           *  sound
8.           *  etc
9.           */
10.     functions:
11.         ButtonClicked ()
12.         {
13.             Get value from each input control on the page
14.             Store them in variables
15.         }
16. }
```

ActionButton class

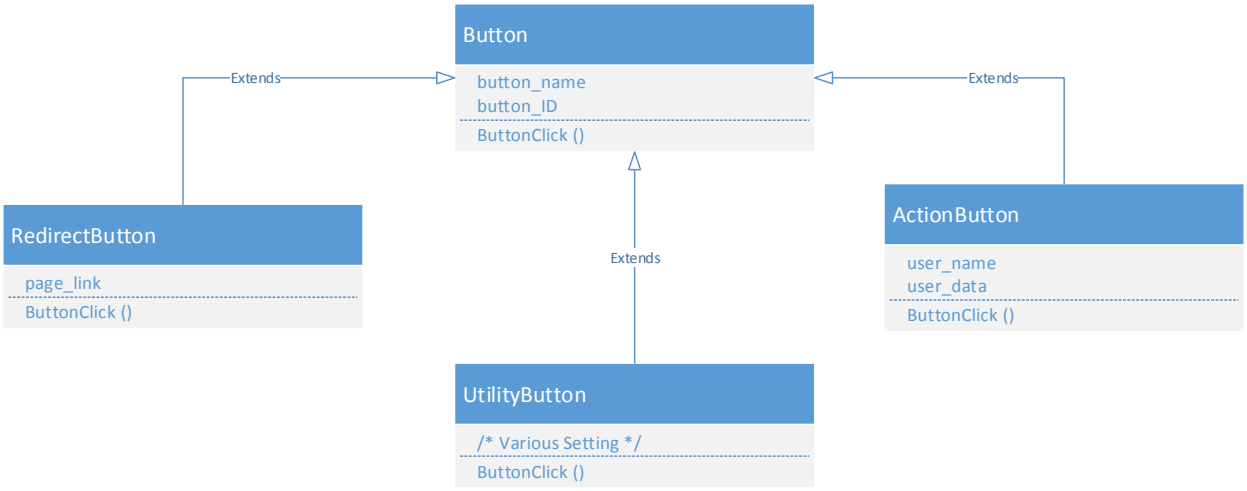
This button type is the most complex, it is any button that is connected to data that must be sent to the server. It has two native variables, the first of which is user_name. This is used to identify the client the network message is coming from. The second is user_data, which holds the data related to the button click. When 'Connect to Server' is clicked it sends the IP address, when a class is selected it sends the selected class.

Pseudocode

The UserMessage class is explained in the Network Message Protocol section.

```
1.  class ActionButton extends Button
2.  {
3.      variables:
4.          user_name
5.          user_data
6.      functions:
7.          ButtonClicked ()
8.          {
9.              Create UserMessage with user_name and user_data
10.             Call onUpdate network function with Message structure
11.         }
12. }
```

Diagram



Chat Functionality

Part of the chat functionality has been explained above, pressing the Enter key to send the text will be handled by an ActionButton. When Enter is pressed it will take the name of the current client's user and the text from the textbox and send them in a Message structure to the network. It has two variables:

- ActionButton
  - Explained above.
- TextBox
  - A class used to represent an area on the screen that the user can enter text into. It has a buffer variable which holds characters and a KeyboardListener function which will take input from the keyboard and store it in the buffer.

Pseudocode

```
1.  class TextBox
2.  {
3.      variables:
4.          buffer
5.
6.      functions:
7.          KeyboardListener ()
8.          {
9.              Pull input from keyboard
10.             Store input in buffer
11.         }
12. }
13. class ChatBox
14. {
15.     variables:
16.         ActionButton
17.         TextBox
18.
19.     functions:
20.         MouseListener ()
21.         {
22.             if click on TextBox is registered
23.                 Activate TextBox's KeyboardListener function
24.                 Get buffer from TextBox
25.
26.             if click on ActionButton is registered
27.                 if chat dialogue is not displayed
28.                     display chat dialogue
29.                 if buffer is empty
30.                     Ignore action
31.                 else
32.                     Call ActionButton's ButtonClicked function with buffer
33.         }
```

Diagram

HUD Components

This part of the User Interface is about the functionality behind the users Heads-Up Display. With the previous sections it's easy to see where the Game Logic responsibility starts and ends, but here it is a bit vaguer. For now I will assume that Game Logic's focus will be on having the HUD data available and knowing when to change it. There will be an InitHUD function that will set all of the initial values for the HUD at the start of a round. There is also an UpdateHUD function that waits for an event from the client that will signal it to update a HUD element.

Pseudocode

```
1.  Function InitHUD ()
2.  {
3.      Set initial values for all HUD elements
4.  }
5.
6.  Function UpdateHUD ()
7.  {
8.      Wait for character event
9.      Update HUD element based on character event
10. }
```

Network Message Protocol

This section concerns the messages sent from a client, to the server and back to the other clients. Any time an event occurs with a user that all other user must know about (ex: vessel moved, enemy lost health, deity chats), any time an entity must be updated (ex: minion dies, boss moves, pot gets destroyed) and any when the server wants to verify the connection of a client.

For these purposes there will be three Message classes all inheriting from a generic parent.

Message class

This is a generic Message which will never be used for functional communication between the server and the client, but only as a base for the three other Message types. It only holds a message_type variable which will be used to differentiate the three different message types for the server.

Pseudocode

```
1.  class Message
2.  {
3.      variables:
4.          message_type
5.  }
```

EntityMessage class

This class is used for all messages concerning entities in the game environment and will be the most heavily used. It has four native variables:

- entity_event
  - Describes the event of that prompted this message to be sent. It can either be Create, Delete or Update.
- entity_ID
  - The unique descriptor of the entity being updated
- entity_value
  - The value of the entity being changed.
    - If the event is an Update, this value will be one of the entities variables.
    - If the event is a Create, this value will be one of the Entity child classes.
    - If this event is a Delete, this value will be NULL or something of similar meaning.
- entity_update
  - This variable will currently only be used for Update events. It lets the server know how much the specified value must be changed
    - If the event is not an Update, this variable will be zero. This seems like a waste of a variable, but it is justified since the vast majority of Entity messages will be due to Update events.

Pseudocode

```
1.  class EntityMessage extends Message
2.  {
3.      variables:
4.          entity_event
5.          entity_ID
6.          entity_value
7.          entity_update
8.  }
```

UserMessage class

This class concerns all user-based events that must be communicated to all other users, but do not take place in the game environment. Whenever a user chats or when a user picks a vessel class, all other users must know about this event. This class has three native variables:

- user_event
  - The event, triggered by the user, which caused this message to be sent. As there currently aren't that many user-based events that could be sent, it is easy enough to have them each defined as a separate instance.
- user_name
  - A unique descriptor for the user who triggered the event.
    - If the event was a Chat, the user's name will appear before the text in the chat window and their Chat privileges will be checked.
    - If the event is a Class Selection, the user's class will update.
- user_data
  - The data associated with the specific event.
    - If the event is a Chat, the data will contain the text message.
    - If the event is a Class Selection, the data will contain the name of the class type.

Pseudocode

```
1.  class UserMessage extends Message
2.  {
3.      variables:
4.          user_event
5.          user_name
6.          user_data
7.  }
```

ConnectionMessage class

The responsibility of this class is to update the server with the connection status of the client. It may be used nearly as much as the EntityMessage class depending on how often the Network team feels the server should be notified. It holds a single native variable:

- connection_stable
    - A binary type used to tell the server the status of the connection between it and the client
        - If connection_stable is true, the connection is still intact.
        - If connection_stable is false, the connection has encountered an error.

Pseudocode

```
1.  class ConnectionMessage extends Message
2.  {
3.      variables:
4.          connection_stable
5.  }
```

Diagram