Abstract

This report represents the work I have done in designing and implementing a 2D video game, specifically battleship.

This project is supervised by Dr Sven Schewe and Dr Andre Hernich of the University of Liverpool. The project began in September 2014 and has Ended in May 2015.

The game was chosen as my final year project as I'm interested in the games industry and I one day hope to work on games professionally.

The original description on what this project needed to be was given by my supervisor Dr Sven Schewe:

Battleship (http://en.wikipedia.org/wiki/Battleship_%28game%29) is a two player board game.

In your project, you are asked to implement the game as a two player game that can be played on distributed machines (different computers) and a solitaire version, where one player is replaced by a computer that plays with a heuristic, which you develop.

The report explains my approach in tackling this problem. I discuss the initial design of the system. Followed by the process I went through to implement and test the software.

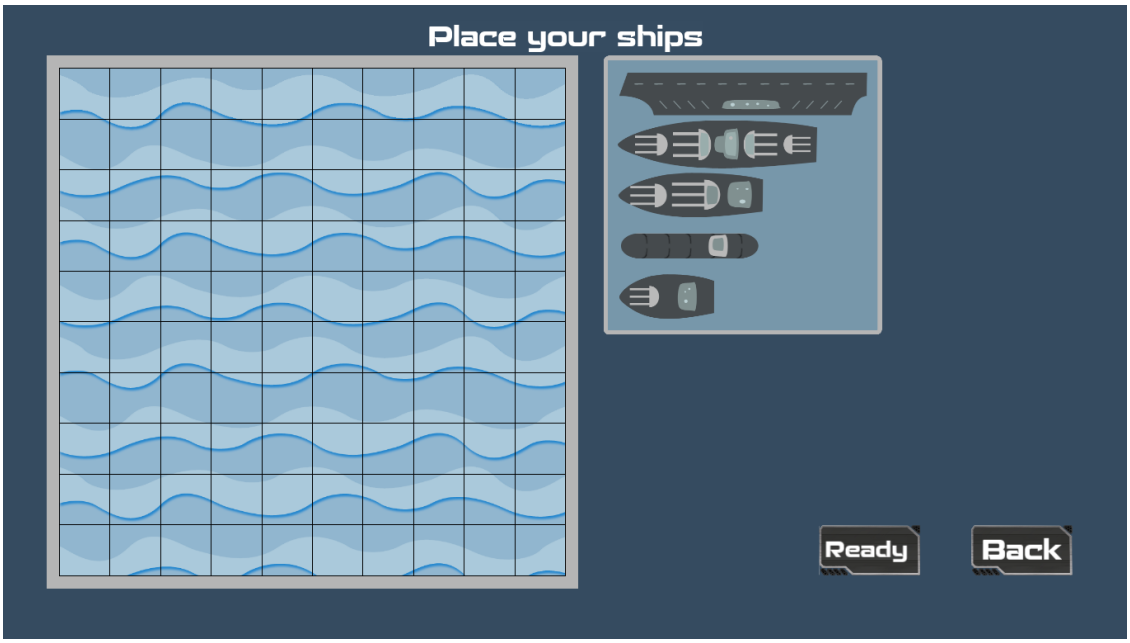
I also evaluate how successfully the project was overall, and I discuss the lessons I learned through working on this large project.

The game is completed and I would consider it to be in a beta stage of development where are the features are implemented and more testing and debugging on a large scale is needed before a full release can be considered. The game is in a fully operational and playable, it can be downloaded from my website at:

http://louis.bennette.co.uk/my-programs/battleships/

Battleships Video Game Report

by Louis Bennette - 200801270



1 Introduction

The aim of this project was to take the well known Battleship tabletop game made by the Milton Bradley co. and create a video game version of it. The original table top game was intended to be played by two individuals so the video game has a multiplayer game type so that users can play against other users. In this application of Battleship the games are played over a network, either the internet or a local area network. A networked multiplayer game is coordinated by a dedicated server that the clients connect to. The server is also distributed so that users can set up their own servers, public or private.

In addition to the multiplayer game type there is a singleplayer game type where the user can play against an AI (Artificial Intelligence)[1] heuristic which will act as an opponent.

As well as the game types the game comes with an options menu to toggle sound parameters as well as chose which server address and user-name is used to connect to a multiplayer

[1]The game industry refers to an NPC (Non Player Character) as an AI. The underlying mechanisms can use algorithms and/or a heuristic. The "AI" does not need to use true AI principles to be considered an AI in gaming terms.

session. The game uses the Hasbro official rule set [3] and the Hasbro ship size naming system. This is the most commonly used system worldwide. Other rule sets do exist however they are not covered in the project.

The secondary goal of this project was to demonstrate an understanding of how to program a game environment, the libraries used are 2D graphical rendering routine, a keyboard input routine as well as some asset loaders. The logic, structure and components of the game have been coded from scratch in java using original code.

The main challenges encountered were to create a capable AI that is not too easy to beat and not too hard either. The AI had to work for any arrangement of ships and was designed to destroy a ship as soon as the ship is discovered. As well as this the AI had to take into account the possibility of adjacent ships. The real difficulty for this was to test all the key endpoints of the AI's algorithm/heuristic. When building the multiplayer I had a few challenges. The multiplayer game type would connect to a separate BattleshipsServer running on a network, this software would coordinate and orchestrate a multiplayer game between connected clients. The challenge with the server was to ensure that it would work properly in it's multi threaded environment. As well as this, the server needed to reliably be capable of running for an indeterminate amount of time.

Another less technical challenge, but still an important one, was to make a user-friendly environment for the player to navigate.

The final version of the game respects the original design quite closely; none of the core features have been omitted, nonetheless, some additional unintended features have been added during the implementation (see 4.1) through expected feature creep. The game has a singleplayer where the user is matched against an AI opponent. The AI acts as another player and strikes a reasonable balance between between being too easy or too hard to defeat. It's worth briefly noting that the AI does not know more than the user does. The AI only has access to it's own ships positions and to a two-dimensional array of the attempts it has already made just like the user (4.1.1 *The AI opponent*).

The multiplayer connects to a custom dedicated server and a user is capable of playing a game with another user over a network and from any location. The server itself respects the original design almost exactly with a few feature additions made to allow for its distribution.

Both multiplayer and singleplayer modes conform to the official Hasbro rules and naming standards. All this has been done using custom code and the libraries stated in the design stage of development. During this process I have learnt to implement new code patterns and how to implement a simple 2D video game's engine and game logic.

I feel I have learnt much from this project and believe it to be a success.

2 Background

It is a software development exercise and challenge. This project exercised a variety of code patterns or workarounds as well as different code practices. I say workarounds because in some cases, time and deadlines were a factor to compete with and correct abstraction with inheritance structures can sometimes take a long time to produce. I hope that my work demonstrates an understanding of object oriented coding based hierarchies and abstraction. As I started the project I had a limited and minimal understanding of game engines and more importantly I did not know the best ways of using them.

If we take the jMonkey engine [4] for example, the engine encapsulates everything using an open-closed principle. Everything is already done for a user of jMonkey or other advanced engines of that sort, the problem is that it these engines become quite impenetrable as you need an inherent understanding of the engine along with it's lexicon and it's structure. I found myself rewriting code that already existed. Moreover, many engines these days have a large 3D focus and 2D is only ever used for Menus and HUDs (Heads Up Displays). I found myself needing to work around and cheat the engine into what I wanted it to do. This, I felt, would not produce very elegant code and would quickly become unmanageable as the code base grew.

So ultimately I decided against using a large engine. The game I had in mind did not require much, in fact all I needed was to draw rectangles and load textures, later on I worked on loading sounds. Another thing I needed was live user input, a feature that all game engines usually support. So the raw, lowest form of requirements I needed for my game were:

- Able to draw rectangles,
- Able to handle keyboard and mouse input,
- Able to draw textures,
- Able to output sound (originally optional),
- Able to send messages over a network (supported by java) and
- Able to draw fonts.

So for this project I decided to use LWJGL [1] (Light Weight Java Game Library) was chosen in the end because of it's simplicity. An alternative that could be used was SDL [6] (Simple DirectMedia Layer) which much like LWJGL offers access to all the required features above and uses openGL and openAL for it's graphics and sound. However, C++ would need to be used instead of java in SDL. As I am more proficient in java than C++ I felt it was safer to work using LWJGL. Also, java offers very simple inbuilt networking solutions through sockets whereas another library would need to be used if the project was written in C++ adding more testing and practice work. LWJGL offers very simple draw methods, audio playback methods and texture objects as well as a Display class to create a window in which the game runs.

3 Design

3.1 Client

The game's design is based around one simple structure. The game consists of views, components and managers. The views act as event listeners and control the update-and-draw of all the components that make it up. In order to interface properly with the user the game uses the manager classes in the views and the components. The managers are responsible for core functions like I/O or resource loading. The game will specifically consist of four scene views. Singleplayer, Multiplayer, Options and Menu. Components are the items that are graphically represented and manipulated by update calls these are for instance; Buttons, Ships or Grids.

The components are the interface the game has with the user. The components are the graphical representation on screen that the user interacts with. They also provide interactions to the user by, for example, allowing the user to drag and drop ships, click coordinates on a map or buttons. The components proposed in the original design are;

- Buttons,
- Ships,
- Grids,
- Labels and
- Text Areas

In addition to these components the design was built upon and the fowling components were added before and during the implementation process;

- A Toggle Button which toggles between two states when pressed and
- an Image component. This was overlooked in the design but is now key to the display of anything on screen.

There are a number of manager classes that are used for system-wide access to their functions. By system-wide I mean that any class in the client has access to their methods and fields (through getters and setters). The system-wide classes are:

- OpenGLManager: used to initialise openGL and to draw rectangles to screen
- TextureManager: used to return a texture given a string.
- FontManager: singleton class used to provide the game fonts to components
- ResolutionManager: singleton class used to provide margins and pre-calculated positions for game objects dependent on the aspect ratio.

- AudioManager: loads and controls audio files and their playback.
- Window: used to control the window and update the window for every frame.
- Input: Keeps track of all the keyboard and mouse inputs.
- Time: Keeps track of the time every frame lasts.
- GameGlobals: Keeps general information that the game can use as options

The AI will be used to play against the player in single player mode. The role of the AI is to provide a challenge to the user by using a heuristic. To tackle this problem I have devised a solution that would make the game still be possible to win but also allow the AI a fighting chance to beat the user. The AI will hold its own 2 grids, they will be used the same way as the user's, one for the AI's ships and one for the attempts on the opponent. Perhaps the AI will also require a sleep timer because the response of such an AI will be very fast and could make the user feel un-immersed.

The heuristic will be simplistic and straightforward, it consists of two states, search and kill. Meaning simply that the AI will search semi-randomly across the board and when it hits a ship it will attack until it has destroyed the ship. More specifically when an enemy ship has been hit the AI will add the points of the hit into a list. Regardless of the state the AI is in, every time it makes a successful hit, it will add that hit's coordinates to the list. When a ship has been sunk the respective points of that ship are removed from the list. This implies that any points in the list belong to a ship that has not been sunk. Only when the list is empty the AI can resume the search state. As well as updating the list, the AI adds every attempt it has made to the targetGrid array. This is to avoid the AI choosing the same point twice.

3.2 Server

During a multiplayer game the player will be playing against an opponent online via a server which is relaying messages to and from each client. The server here is designed to minimise the amount of process time it uses. To do this the server will just act as a relay, sending messages to and from the clients.

The multiplayer game will be sending the messages to the server, there will also be a message listener attached to the muliplayer game. This simply listens to any message that will be sent by the server. When a message is received it's placed in a list. The multiplayer game will check to see if there are any messages in the list. If there are it will send the message to a parsing function. Once parsed, the appropriate method calls are made based on the message command.

The server will be multi-threaded, this is because we need several instances of a game to run at once. We also need the server to keep adding new users and listeners for incoming

messages. The Server Listener will be listening to any new connections and adding them to a login thread which will handle their user-name and add them to the online users.

Online users is the list of currently connected users. The user class will hold data on the user's socket and their name. The user also has a listener associated with it. This listener is waiting for messages from the client that it is associated to. When a message is received the string will be added to a list. The game instance will check the lists and parse any message for both users in every loop. If there are 2 or more users in the Online Users list then the matchmaker will take the first two users in the list and add them to a game. When the game instance is started up it is expected to receive two users. At this point the message listeners for each user are started up and the game can begin.

The server will be run from the command line and was not originally intended to have any graphical interface. However during the implementation of the server I found an alternative application of the server that was more flexible and easy to use. (More on this in the Realisation section)

4 Realisation

The realisation of the project happens in two phases; Implementation and Testing. During the testing phase I had to go back to revisit the code to fix bugs and add user friendly features without violating any of the core features or adding new ones, this is the Beta phase of development.

After Beta testing is over, the game should move into a full release stage and would be ready for mass distribution.

4.1 Implementation

It was important during the implementation of the project to respect the order in which different components would be implemented. The program can be split into three categories;

- the managers,
- the structure and
- the components.

The managers act as an interface to the user and to hardware. The managers are responsible for reading and writing to files, loading assets into memory and drawing geometries to screen as well as reading input from the keyboard and mouse. The structure is crucial as it ties the components to the managers and wraps the mechanisms of the program.

The structure is the hierarchical system of classes as well as the "controller" classes that handle events and orchestrate the game.

The components are the elements that interact with the user by either being viewed and read like a label or are being clicked and moved like buttons or ships.

A typical implementation cycle would require the creation of parts of all three categories. For example, the very first thing that I implemented was clickable buttons on screen. I needed to create two managers for this; one to draw a geometry to screen, and one to take user mouse input. I also needed to start the game's structure so I created a GameLoop class and a MainMenu class which is a child of AbstractView. And finally, I needed to create the Button class which is a child of AbstractComponent, the button's event logic and drawing the button to screen.

This is a simplistic example of how I conducted myself during the implementation, I often started by making the managers followed by growing the structure of the project and finally by implementing the components I needed. Breaking the workload down like this made growing the project more simple. I could code in small sessions and have compilable source code whenever I resumed work.

During the implementation phase some features which were not discussed or planned in the design of the game became necessary to deliver a more interesting and user friendly game. I ended up adding the following features during the game implementation: An event system, a subview hierarchy, an animation system and a toggle button component.

4.1.1 Client structure

It became apparent that I would need an event system for the game. Originally, the plan would have been to use the java event system or "*action system*", by extending ActionEvent and implementing ActionListeners. It became obvious that for my more simplistic use of events, I could rewrite my own as a learning exercise, this came very early in the implementation phase. One of the first intractable things I created was a button which would trigger an event in my "EventResponder" class (ActionListener equivalent).

The event system:

The event system is made up of IEvent and of IEventResponder objects. An IEvent is an object that has the $onEvent()$ method and a $getName()$ method. An IEvent also has a IEventResponder for which the event is executed in. An IEventResponder has a $handleEvent(IEventevent)$ method, this handles the event in question and performs all execution of the event. When an event is triggered the IEvent objects's $onEvent()$ is called.

As we can see in Figure 1 the event responder's $handleEvent()$ method is called and the instance of the currently executing event is passed as a variable to the responder.

Game states:

At any time during a run of the game the game is in a scene state. This scene state corresponds to which scene should be loaded and run. There are four possible scene states and they are $MENU$, $SINGLEPLAYER$, $MULTIPLAYER$ and $OPTIONS$. These

```
@Override
public void onEvent() {
    if(!this.isKillEvent()){
        this.eventResponder.handleEvent(this);
    }
}
```

Figure 1: Event dispatch

are implemented as a an enum type and are globally accessible through variable storing the current scene state. As we can see in figure 2 the switch statement calls the globally accessible scene state. Based on the current Scene state we load the corresponding SceneView and we only load that view if it is not already loaded. A SceneView is an object that represents it's own individual view and event handling. For example, Singleplayer or Menu.

The active view is then updated and drawn to screen every frame. This call is the main update and draw call for every frame.

In addition to the global scene state there is a sub state called PlayerState which represents the current state of a running game. The PlayerState is only used in Multiplayer and Singleplayer. The PlayerStates are: *SHIP_PLACEMENT*, *ANIMATING*, *WAIT_FOR_OPONENT*, *USER_TURN*, *OPONENT_TURN*, *GAME_OVER* and *QUEUED* (only in Multiplayer). The PlayerStates allow the view to change what it's currently displaying on screen or which component updates need to be made or blocked.

Multiplayer parser

When a message strings are received from a server during a multiplayer game and need to be interpreted. The message's initial character indicates where the message comes from. A variable may also be appended to the message.

A message starting with a '1' character means that it should be considered by the Client as a message from the opponent and a message starting with a '2' is a message from the server.

During a multiplayer game when a user clicks a location to shoot at, the game sends a "1try=x,y" command where "x,y" are the coordinates of the point the user is attempting to shoot. "1" here means it's from a client. "try" means that the command is interpreted as an attempt from the opponent. Finally if the command has a variable, the variable is taken out of the string and stored to a usable type. Here into two integers which are used to determine if a ship is at those integer coordinates.

In code there is a message listener which is a separate thread which listens for any new messages from the server. When a message is received the string is immediately added to a new thread which will parse and handle the command. This is so that the message listener can resume to listen to messages from the server as soon as possible. The handling thread is actually a MessageRecievedEvent and the event responder is the MultiplayerGame object.

```
switch(SceneState.getGeneralState()){
   case MENU:
      if(currentSceneView == null || !(currentSceneView instanceof Menu)){
         TextureManager.flushTextures();
         currentSceneView = new Menu();
      }
      break;
   case SINGLEPLAYER:
      if(currentSceneView == null || !(currentSceneView instanceof
          SingleplayerGame)){
         TextureManager.flushTextures();
         currentSceneView = new SingleplayerGame();
      }
      break;
   case MULTIPLAYER:
      if(currentSceneView == null || !(currentSceneView instanceof
          MultiplayerGame)){
         TextureManager.flushTextures();
         currentSceneView = new MultiplayerGame();
      }
      break;
   case OPTIONS:
      if(currentSceneView == null || !(currentSceneView instanceof Options)){
         TextureManager.flushTextures();
         currentSceneView = new Options();
      }
      break;
}
currentSceneView.update();
currentSceneView.draw();
```

Figure 2: Game loop, state handling mechanism.

```
String variable = Parser.getMessageVariable(message);
message = Parser.truncateVariable(message);

switch (message) {
   case "op-name":
      this.opponentName = variable;
      this.bannerLabel.setText("Opponent found: " + this.opponentName);
      break;
```

Figure 3: Multiplayer message interpreter

The message is parsed to check if it's valid first. The check for validity is to see if the string starts with one of the expected characters. Then the command is used in a switch statement to execute the appropriate methods. (for an example of the Message interpretation see figure 3)

The AI opponent:

The AI as stated in the design has two states; Search and Kill. The AI opponent has the ability to have one loaded AIBehaviour the AI behaviour classes have a *getNextCell()* method which returns a point. There are two classes AISearchState and AIKillState, both extend the AIBehaviour and have the *getNextCell()* method.

The search state was implemented first. The search state was very simple to implement, to recall the search of the AI is to only choose every other cell as a search option. To do this the game generates two random numbers and makes sure that they are even. By only choosing even numbers the game can achieve the chequered search pattern needed. If a number is found to be odd the AI adds 1 to it and makes it even. Then it checks to see if that point has already been shot at. If it has then the process is repeated.

The kill state, as mentioned in the design, has three sub states IDENTIFY, HORIZONTAL and VERTICAL. These substates are, like all states in the game, implemented as an enum. The active state is stored in a variable. When a ship is found the AI swaps it's active search behaviour for a kill behaviour. This implementation of this kill behaviour is just as stated in the design. The AI uses a List object to store the Points that are hits. When a ship has been sunk the AI simply removes the points that correspond to that ship from the list. Once the list is empty the AI can then resume the search state. In addition to the list the AI holds the coordinates of the last attempt it made. This is very important to determine which direction the AI is shooting in and also to determine which point need to be removed from the list when a ship has been sunk.

Options view:

The Option view was implemented last, this was an optional feature proposed in the design stage. The implementation of the options was very straightforward and quick. One of the key features I wanted the options to have was a persistent memory of the player's

options. The options view uses the GameGlobals class game wide information is stored in the game global class such as; the user's online username, the server IP, the version of the game and the boolean variables to toggle the game music and the game sound effects.

When a variable is set in the GameGlobals it is also saved to a .txt file, when the game is loaded up the text files are checked to see if there is already a variable and when there is the game loads that variable.

The options view simply takes in the current settings of the global game variables and loads the values to the components, The user can then set their preferences for audio and change their user name or which server they wish to connect to. Once the user is done and confirms their changes, the values stored in the components are set in the GameGlobals class which as mentioned will in turn save the new variables to a .txt file.

4.1.2 Client managers

The Managers act as an interface to the user either for graphical display or user input, as well well as handling core game concepts.

The Game Logger:

The Logger was not an originally intended feature in the design of the program. This feature was needed however to help with debugging during the testing phase. When, the testers found a bug they could send the session log of their last play. The game logger is responsible for the logging of key moments during the game's runtime. It logs key milestones in the code, such as Attempts the AI makes against the player and visa versa, it will log the resolution of the screen and other useful information.

The logger also has three levels of security using the default java Logger's; Info, Warning and Severe (see Figure 4). The GameLogger allows the main game thread to log any message from any point in the game's code.

Time

Time is inspired by the Time class from the Oscar Veerhoek tutorial video [7]. The Time class offers a globally accessible method to retrieve the amount of time elapsed since the last frame. The variable delta is used as it holds the 'change' in time since the last frame. Delta is used for speed based calculations and is essential to be able to smoothly move objects across the screen. It's application in this game is move the missile across the screen. Because I have a value of time I can perform some calculations to induce accelerations and velocities on the missile component. The delta value is measured every frame by calling the $setDelta()$ method in the main loop (see Figure 5).

The Resolution manager:

This class takes the resolution of the current display that the game is running in and will generate some pixel based positions and sizes that are used to place and correctly scale components. Specifically the resolution class generates margins to the top, left, right and

```
public static void print(String message){
    getLogger().info(message);
}

public static void warning(String message){
    getLogger().warning(message);
}

public static void severe(String message){
    getLogger().severe(message);
}
```

Figure 4: GameLogger severity levels

```
/**
 * This is called for every loop of the game. For every frame in the game
    we
 * calculate the time since the previous frame. This should only be called
 * once every loop no more or no less.
 */
public static void setDelta(){
    Time.thisFrameStartTime = Time.getTime();
    Time.delta = (double)(Time.thisFrameStartTime -
        Time.lastFrameStartTime)/SECOND;
    Time.lastFrameStartTime = Time.thisFrameStartTime;
}
```

Figure 5: Time setDelta() the time elapsed since the last frame

bottom of the display area. The resolution manager also calculates the dimensions of the grid components cells. This is a default used in other components as well to ensure that components scale correctly for different screen sizes. The ResolutionManager is instantiated and initialised for every new view and the margins are recalculated. This was an intended feature so that in future I could flexibly create a resolution resize method and as soon as a new view is created all the new pixel locations are recalculated for the new display size. This feature however is not used.

The Font manager:

The game requires two fonts. The font manager class is responsible for loading the fonts into memory and creating usable TrueTypeFont objects provided by slick 2D. A TrueTypeFont object has a $drawString()$; method which is how labels work in the game. This method uses openGL and fits well into the game's structure.

The openGL manager:

The entire update and rendering cycles of my game run a looping procedural path inside a single thread, I designed a globally accessible method call for drawing rectangles and latter drawing rectangles with textures (see Figure 6). There was no need for synchronisation, this being a mono threaded environment, as far as rendering was concerned. The geometries I used were simple enough for most GPUs to handle without the need of a more complex rendering procedure.

This meant that at any point the main game thread of execution I could call a draw to the GPU. When $glBegin()$; is called I am telling the openGL library to add the following geometry to it's display buffer. At the end of every game loop an update is called on the LWJGL Window object, this tells the Window to change it's active buffer to the one that I had just been adding geometries too, as well as clearing the old one to be used in the next frame.

Input:

The input class is not my original creation. It's taken from Benny Bobaganoosh's Input class ([2]) used in his 3D game engine in java tutorials. The input class requires an update for every loop which scans through every key on the keyboard and updates the list of keys currently held down in a list called $lastKeys$. The list will be used for the duration of the frame as a snapshot of which keys have been pressed (see Figure 7).

Any key that is currently pressed but not in the last frame's list means that it has just been pressed. A key that was pressed and is not in the current frame has just been released. This is the same for mouse and keyboard input.

The Audio manager

The AudioManager makes use of openAL (open Audio Library) supplied by LWJGL. The AudioManager loading system is based off the tutorials given on the LWJGL website [5] and expanded on slightly. As soon as the class is invoked, when the game loads up, all of the sound

```
public static void drawSquareWithTex(float posX, float posY, float width,
    float height, Texture tex){
     tex.bind(); // Tell openGL to draw this Texture in the following given
         area
     glEnable(GL_BLEND);
     glBegin(GL_QUADS);
         glTexCoord2f(0, 0);
         glVertex2f(posX, posY); // Upper Left
         glTexCoord2f(1, 0);
         glVertex2f(posX + width, posY); // Upper Right
         glTexCoord2f(1, 1);
         glVertex2f(posX + width, posY + height); // Bottom Right
         glTexCoord2f(0, 1);
         glVertex2f(posX, posY + height); // Bottom Left
     glEnd();
     glDisable(GL_BLEND);
 }
```

Figure 6: OpenGLManager drawSquareWithTex() called to draw a quad to screen at the given coordinates with the given texture and size

files are loaded together into memory. Every audio clip is stored as a SoundFile object which has a name and an ID. The name of the SoundFile is just the String of the .wav file name. All of the sound file's data is loaded into a buffer in the openAL library and when I want to play, pause or stop an audio clip I call the $AL10.alSourcePlay()$, $AL10.alSourcePause()$ or $AL10.alSourceStop()$ methods supplied by LWJGL and give the ID of the file I want to perform an action on.

The AudioManager acts as a simple layer between the game and LWJGL. All of the SoundFiles are globally accessible and are final. (see Figure 8)

As well as supplying a public access to the SoundFiles themselves the AudioManager supplies it's own $play()$, $pause()$ and $stop()$ methods which take in a SoundFile object as a parameter (see Figure 9). And finally, the user can choose in the Options menu if they would like to hear Music or Effects by toggling them on and off. In order to implement this the SoundFile object has a $Type$ parameter that is used by the AudioManager to know if the file is a $MUSIC$ clip or an $EFFECT$ clip. Based on the user's selection the file will or wont be played (see Figure 9 play() method).

The Texture manager:

The texture manager is mostly called inside the game constructors, all relevant textures are loaded for the particular scene and unused textures are naturally not loaded. Moreover, another feature I wanted to implement was, for want of a better word, "singleton" textures. meaning that I only need to load the texture for a button once, and any object that requires

```
    /**
     * Returns a boolean value if the tested key is currently held down live.
     * @param keyCode the key we want to check that is being held down.
     * @return true is the keyboard key is being held down false is not.
     */
    public static boolean getKey(int keyCode){
        return Keyboard.isKeyDown(keyCode);
    }

    /**
     * Checks if the requested button has started being pressed in the current
     * frame, we know because if the key was previously not pressed and now is
     * then it's just been pressed.
     * @param keyCode the key we want to check that has just been pressed.
     * @return true if the key has just stared being pressed in this frame.
     */
    public static boolean getKeyDown(int keyCode){
        return getKey(keyCode) && !lastKeys.contains(keyCode);
    }

    /**
     * Checks if the requested button has just been released in the current
     * frame, we know because if the key is currently now held down but
     * was held down in the previous frame, then it has been released.
     * @param keyCode the key we want to check that has just been released.
     * @return true if the key has just been released in this frame.
     */
    public static boolean getKeyUp(int keyCode){
        return !getKey(keyCode) && lastKeys.contains(keyCode);
    }
```

Figure 7: Input, the mechanism of how the game knows if a key has just been pressed or released

```
public static final SoundFile MENU_MUSIC = new SoundFile(0, "main-menu-music",
    SoundFile.Type.MUSIC);
```

Figure 8: A sound file stored in the AudioManager class

```
public static void play(SoundFile soundFile){
   if(GameGlobals.gameMusicOn && soundFile.getSoundType() ==
       SoundFile.Type.MUSIC){
      AL10.alSourcePlay(source.get(soundFile.getID()));
   }
   else if(GameGlobals.gameEffectSoundsOn && soundFile.getSoundType() ==
       SoundFile.Type.EFFECT){
      AL10.alSourcePlay(source.get(soundFile.getID()));
   }
}

public static void stop(SoundFile soudFile) {
   AL10.alSourceStop(source.get(soudFile.getID()));
}

public static void pause(SoundFile soudFile) {
   AL10.alSourcePause(source.get(soudFile.getID()));
}
```

Figure 9: The play, pause and stop methods used on SoundFiles in the AudioManager class

that texture will be passed a pointer to that texture rather than a new block of memory.

This is extremely common and good practice in texture loading for video games. The texture manager has a Map object that can contain Texture objects with String identifiers. (Texture, a class provided by 2D-slick) The manager identifies Texture objects based on the file name that the Texture was loaded from so it uses unique String identifiers, e.g. the texture that is loaded from the file "button.png" is identified by the string "button" without the extension.

When a texture with name "button" is requested, the manager will first check its list of Textures to see if the texture button is loaded. If it is, a pointer is returned rather than reloading the texture. If the texture is not in the list, i.e. not in memory. Then the manager will load the texture, update the loaded list, then return the pointer to the texture (see Figure 10).

The last feature of this TextureManager is that between large scene changes all the currently loaded textures are flushed and the new relevant ones are reloaded. This occurs during a large scene change from Menu to a Game for example this is so when the user exit a game back to main menu the game does not keep ship and grid textures in memory.

Regardless, it's worth noting that most computers have enough memory to maintain the textures of this game loaded throughout the entire execution of the game, a single load of all textures at the start could be done and at the cost of space the game can reduce load time, a concept that comes up throughout the development of games. Because of this, I believe that the optimal user experience is actually to do this one load and maintain very fast draw

```
    /**
    * Returns the texture with name given by the variable texName, returns
    * a pointer to the texture clones are not made.
    * @param texName the unique identifier of the texture e.g. button.png is
    * identified by the string button
    * @return Texture pointer used to draw the screen
    */
    public static Texture loadTexture(String texName){
        try {
            Texture tex;
            if(isLoaded(texName)){
                // If texture is already in memory then return the pointer to it
                tex = textureHolder.get(texName);
            }
            else{
                // Load the new texture.
                tex = TextureLoader.getTexture("PNG", new FileInputStream(new
                    File(path + texName + extention)));
                textureHolder.put(texName, tex);
                loadedTextures.add(texName);
            }
            return tex;

        } catch (IOException ex) {
            GameLogger.warning("TextureManager ERROR, texture " + texName +
                extention + " failed to load.");
            return TextureManager.getDefaultTexture();
        }
     }
```

Figure 10: TextureManager, the mechanism used to load textures into memory without duplication

speeds between scene changes.

However, this was an exercise to show and demonstrate an understanding of the principles in game development. The one load system scales very poorly, the ability to maintain only required textures in memory is needed for larger games, and this demonstrates from an engineering perspective a more robust and scalable way of loading textures.

In addition I will briefly make a note on the way openGL uses textures. The textures have to be very specific pixel sizes. 2^x *by* 2^y *to be exact, where it's possible that* $x = y$*. For example the grid's wave texture is 1024 by 1024 pixels.*

4.1.3 Client components

Components are the graphical elements on screen that the user is capable of interacting with to make changes occur in the game. As previously stated in the design section two new components have been added since the design proposal during the implementation of the project. Also, to reiterate, all the on screen rendering is done trough openGL supplied by LWJGL.

OpenGL has its own screen coordinate system separate to the underlying OS (Operating System). OpenGL considers the top left of the screen to be the origin. A positive X axis direction is towards the right on the screen much like in conventional graphs. However a positive direction in the Y axis is downwards towards the bottom of the screen. Also, a Z axis is available as a 3rd dimension but is not used in this game.

Therefore, the components are implemented as rectangles. They have an origin point in the top left to conform to the openGL standard and a width and height which expands to component to the left and downward respectively.

Every component has a texture. However, during the implementation I created the Image object component. This means that newer components use the Image object to render their graphic to screen and components that were implemented earlier use the components Texture object. This is has lead to a somewhat confusing discontinuity in code due to the unintended feature creep of the Image object. However, using the Image object is neater and simpler to understand in code.

Some components have access to a method to check if they have been clicked. Figure 11 shows an example of the isClicked method used by the Button object. I only consider a successful click if the component is pressed *and* released within its bounds.

The components proposed in the original design can be reordered form more primitive base components to the more complex "compounded" components which are made up the primitive components;

The two primitives:

- Images and
- Labels.

and the "compounded" components;

- Buttons,
- Toggle buttons,
- TextAreas,
- Ships and
- Grids.

```
/**
* A button is clicked when it has been pressed and released in the bounds
* of the button. This is used to execute the event of a button.
* @return true is the button has been clicked.
*/
public boolean isClicked(){
  // We check first to see if the button has been pressed.
  if(this.hasMouseHover()){
     if(Input.getMouseDown(Input.MOUSE_LEFT)){
        // Button is pressed
        this.buttonClicked = true;
     }
  }
  // Now we check if the button is released withing the button's bounds.
  if(Input.getMouseUp(Input.MOUSE_LEFT)){
     if(this.buttonClicked && this.hasMouseHover()){
        this.buttonClicked = false;
        // Only return true if the button is pressed and released in the
        // bounds of the button.
        return true;
     }
     else
     {
        this.buttonClicked = false;
     }
  }
  return false;
}
```

Figure 11: The isClicked() method. Here, extracted from the button class but reused in other classes.

An Image is a primitive component which was added around half way through the implementation as it was overlooked in the design. The advantages of using the image are a development only. It made the development process slightly easier than it would be using textures for every component. The Image object simply uses it's origin, width and height to display a loaded texture within those bounds.

A label, like all components, is a rectangle with an origin in the top left a width and a height. Labels hold a String as their text that is rendered on screen. As well as the text, a Label has a point size, a text colour, an alignment (format left, right or centre) and few other variables to determine whether the font should be resizeable and to determine if the label is a title.

One of the key features of these custom labels is that the Labels have an "autoresizable" option. This allows the game to calculate what size the font should be automatically based on the size of the space it needs to fit in. The automatic resize of the font only occurs if the text drawn at it's current font size is larger than the determined width and height of the label component. If it is larger, the game will iteratively downsize the font size until it finds a suitable font size that fits. The automatic resize feature would be a useful assist to help with translating the game into different languages.

An application of this feature is in the buttons, as we can see in figure 12 the font size of the "Singleplayer" and "Multiplayer" text on their respective buttons is smaller than the font size of the "Empty" text and the "Quit" text. This is because the string of characters for "Singleplayer" and "Multiplayer" is longer so the auto resize kicks in and resizes the fonts to fit into the space of the button.

Moreover in figure 12 we can also see the text format at work. The text has been set to centred and to auto resize by default for all Labels in the Button objects.

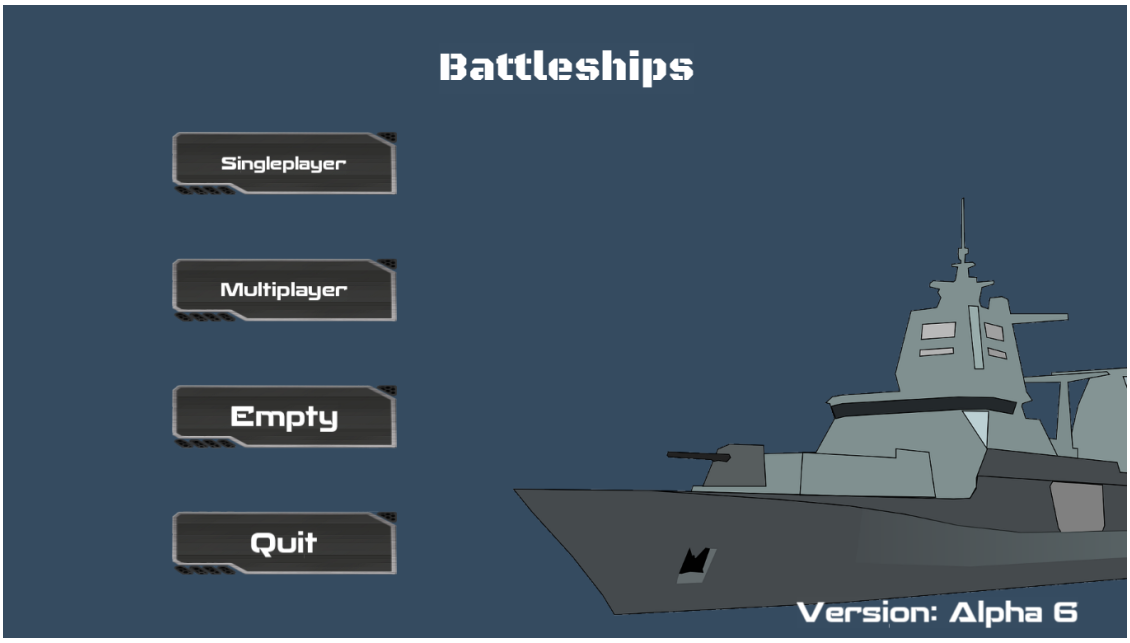


Figure 12: Menu screen shot from build version Alpha 6

The Buttons are typically used to make state changes in the game. The Buttons use the custom event system, each Button has an EventResponder and a unique name (see Section 4.1.1 *"The Event System"*). The Button is an IEvent and when the button is clicked the handleEvent method is called in the relevent EventResponder class. As an example; the Singleplayer button in the main menu has an EventResponder which is the Menu object and when the button is pressed the handleEvent() method is invoked in the Menu object.

Buttons hold an Image object and a Label object. The Label just holds the display text of the button.

The ToggleButton came very late in the development some time after Multiplayer was implemented. It's use is to flip between two states; ON or OFF. In it's implementation it is used to let the user selected whether they want the music or the sound effects to be muted or not. The toggle button extends the button class and works using the exact same method calls as a button does.

A Text Area (or TextField as referred to in the code) uses a Label to display its current input text and a background Image which changes to inform the user if it's being edited or not.

When the user clicks on the TextField object the object goes into a *isBeingEdited* mode where the background is changed and the TextField starts to listen to any keyboard presses by using the InputReader class (built on top of the Input class). This InputReader listens to all alpha numeric characters and if they are pressed creates a string of the keys that have been pressed. This string is simply appended to the current display string of the TextField.

The TextField also does some simplistic string manipulation. The user can backspace to delete the last character in the string and if the Shift keys are held down the input letter is capitalised.

The Ships are initialised in the player class which has access to the ships as well as the game grids. There is one player object for each player. The ships actually have two locations one location which is the current "live" or visible location on screen. When a user drags a ship around the live location is changed as this is the location that is used to draw the ship to screen. In addition to this location the ship has a "safe" location. This is used to snap the ship back to its "safe" when the ship is misplaced or is not put in a suitable location. When a position is deemed unsafe the live location is set to the safe location. When a ship is successfully repositioned the "safe" location is updated. In the code the "live" location is a Point object called "origin" and the "safe" location is a Point object called "safeCoords".

In addition the "safe" and "live" locations the ships have corresponding safe and live orientations (vertical or horizontal). The game uses the "live" orientation to draw the ship correctly and the "safe" orientation to reposition the ship, much like for the locations.

The ships also have a counter which is incremented every time the ship is hit. When the counter is larger or equal to the ships size the ship is considered "sunk". This is used to determine if the game is over.

There are two game grids for each player. The player class initialises and instantiates the a FleetGrid object and a TargetGrid object.

The FleetGrid is responsible for holding the locations of the ships that have been placed on it. The fleet grid has addShip and removeShip methods update the location of ships as the user decides on where to place them. The fleet grid places the ships in a 2D Array of type AbstractShip. The array is 10 by 10 just like the visual grid the user sees. In order to place a ship in a certain location in the game's logic and not just visually, I place the ship an amount of times corresponding to the ship's size. For example, if I are placing a Carrier (size 5 ship) I place 5 pointers to that ship in the corresponding 5 cells that ship occupies visually. This means that when a cell is targeted I can select the x and y coordinates for the selected cell in the player's target grid an check if there is a ship at that location in the opposing player's fleet grid.

The FleetGrid is also responsible for displaying where the opponent has made an attempt on the user's Fleet. It does this with another 2D Array of HitMarkers. A HitMarker is an enum where the value is either; NONE, HIT or MISS. Depending on which value is stored in the array I draw a decal of either a splash or an explosion on top of that cell.

The TargetGrid is responsible for displaying to the user which attempts they have made on their opponent. It does this by using the HitMarker enum just like the FleetGrid. As well as this it allows the user to select which cell they wish to attack. The users can click on the cell and a target reticule is rendered above the cell they wish to attack. This is done by working out which cell is clicked based on the pixel that was clicked. Because I know the dimensions of a cell and the grid and the location of the grid within the display the game

can calculate in which cell that pixel falls in. The selected cell is retained in memory until either a new cell is selected or the selection is confirmed. As previously mentioned, once the cell is confirmed then I use the user's seleced cell from the target cell to see if there is a ship in the opponents fleet grid at that location.

Subsequently, I then update the opponents fleet grid and the user's target grid based on if the result is a hit or a miss. We can see that the user is also updated with a label saying if their attempt was a hit, miss or even a sink (see figure 13).

Figure 13: Screen shot of the moment after the user confirms their selection

4.1.4 Server

The server was very closely implemented using the design. It's was important during the implementation of the design to respect the order in which the server was implemented. I first started with a small separate client capable of sending and reviving messages to and from a server. This was used to do all the testing later on as well before multiplayer was implemented in the game.

The server has a main thread that listens for any new connecting client by using the java socket methods. Once a new connection is discovered the new connection is passed into a separate Login thread so that the main connection thread can resume listening for new users. For each user that connects a new login thread is made, this is so when multiple users are connecting simultaneously they don't get queued up, once the login is done the login thread is terminated and eventually is cleaned up by the JVM (Java Virtual Machine). The Login

thread makes a "2get-username" command to the connecting client. The client responds with their user name before they are added to a list of online users and added to a queue. As the client are added to the list of OnlineUsers a new small MessageListener thread is made for each connected client to listen to messages from their specific socket.

There is a separate matchmaking thread which is implemented as a loop that checks the current OnlineUsers list which also has a sub list of queued users. Every loop the matchmaking thread will check to see if there are two or more queued users. If there are then 2 users are taken and placed in their own GameInstance which is also it's own separate thread.

The OnlineUsers list is only accessible through a getOnlineUsers method which is synchronised so that only one process can access it at any one time. This is the only set of data that needs synchronisation.

There is one game instance for every two currently playing users. Once the user's game is over the game instance is cleaned up by the JVM. The GameInstance class essentialy coordinated which state the connected clients are currently in (see Section 4.1.1 *Game States*). For example the very first thing the game instance does is tell each client to change to a *SHIP_PLACEMENT* state. As well as coordinate the state the clients are in the game instance relays the commands from one user to the other. This is to avoid direct peer to peer connections which are more unsafe and complex to implement using sockets. If a command incoming from the user starts with a '1' then it is ignored by the server and passed on to the other client who will interpret to command as mentioned in Section 4.1.1 *Multiplayer parser*.

In addition to coordinating the states the game is in, the game instance handles a lot of the disconnection situations. If a user is found to be disconnected then the game instance will inform the other player that their opponent has disconnected.

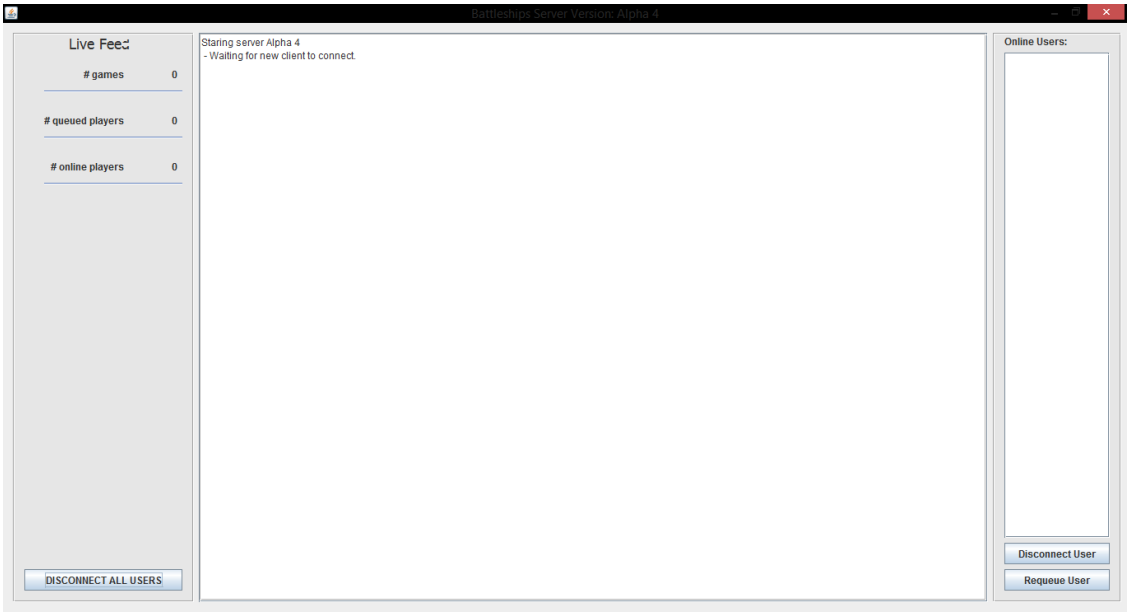


Figure 14: Screen shot of the server gui in version alpha 4

Finally, I realised during the development of the server that it could be distributed instead of having only one dedicated server managed by myself. All of the testing was performed on a local network and there was no need for any additional code to make it work on a remote server over the internet. Because of this I felt I could distribute the server separately.

I started to write a GUI once the server was implemented. I did not want to deviate from the design so none of the mechanisms I worked on could change. I decided that the GUI needed to be optional, show all of the console commands, show the number of online users and the number of queued users.

The GUI sits on top of the OnlineUsers data. Once every second the relevant data is read to update the view. Also I ended up adding a list of the connected users and the ability to individually disconnect and requeue a selected user. There is also a disconnect all button. Because of this, there is more functionality by using the GUI.

There is also a small cleanup thread using the same mechanism of the GUI. Every ten seconds the OnlineUsers list is ready by the cleanup and if a user has been inactive for more than three minutes that user is disconnected.

4.2 Testing

The game was tested in many different ways as the game covers many different areas that are separate and capable of causing different issues in the game.

The very first thing I implemented to help me with debugging the game was the GameLogger class. This is a normal Java logger wrapped us in some custom code. Any time, I want to use a println command I perform it through the GameLogger so that as well as printing

to the console it also prints to a log file. This proved to be very useful during the alpha testing phase where the game was deployed to testers.

The alpha stage was when the game was in a good enough playable state. The very first alpha stage featured the Main menu and a playable single player with a working AI, free from any obvious in-house bugs. This first release helped me find some bugs relating to the AI and some of the AI code was rewritten before Alpha 2.

Alpha 4 was the first well played release of multiplayer. Although multiplayer was running in alpha 3, I discouraged it as alpha 3 was prematurely released. During alpha 4 many bugs were founds and fixed for the next version. These bugs related to the server and to the client simultaneously in most cases.

Subsequent releases also helped to find the occasional bug which was very useful to help polish the game as the features had now all been implemented.

This whole distribution testing using volunteers to help find bugs and send me their game logs was very valuable. The process was managed in a spreadsheet and when a bug was found it was either immediately fixed and marked as fixed in the spreadsheet or it was added to the sheet as a to be fixed element along with a link to the relevant game log. This just helped me stay on top of the varied bugs and made sure that duplicate bug reports were amalgamated into the same problem.

Before the distribution phase of the game I performed a number of small tests on the server in an attempt to break it or slow it down. This was a "stress test" where I attempted to create a large volume of connections to the server to see how well the server could handle it. I performed the test using the small message client used to develop the server mentioned in section 4.1.4. The server was capable of handling volume in excess of 100+ connected clients however it was my personal machine that failed the stress test before the server did. It's worth noting that each client was sending a small message to the server every 2 seconds. There was minimal visual lag between when a message was sent a received and for the scope and purposes of the game I considered the test a success.

Other tests I performed on the server were sudden disconnection tests to see what type of disconnects the server can detect. I did this with a sudden Alt+F4 quit (in windows) which shuts down the current application, with a deliberate infinite loop to simulate a "program not responding" error and with the task manager "End Process" button. I wanted to test if the server could detect these form of disconnects. The server is capable of detecting Alt+F4 tests and "End Process" tests. However, the results of the program not responding tests were and still are inconclusive. In some cases the server can detect a disconnection and other times it will not work.

I did not perform many tests on the AI as it appeared to be working correctly and did not break the game. Sadly, creating unit tests for the AI would be overly complicated and

the tests themselves would be very large as it would need to initialise a large amount of data correctly in the right order just to make some assertions on the next point the AI would chose. The AI is non-deterministic as well so for every test there's a chance it will not do what I wanted it to do as it's next decision is not hard coded. In this respect I would have preferred to test the AI more thoroughly.

A test I did perform on the AI was to pit it against itself in a development build of the game. I played around 20 games of AI vs AI and they all ran to completion. The AI was more closely debugged in the distribution phase as players come up with more interesting ship placement than the AI's random ship placement.

5 Evaluation

The aims of the project were to create a game that was simple, intuitive to use and payable by all ages. The game had to have singleplayer and multiplayer game types.

In the original design I mentioned the following points which are still relevant to the game. They are:

- How well do users respond to navigating and playing the game?
- Is the game of high standard and enjoyable?
- Is the game easy to play but still engaging?
- How robust is the software for the client and the server?
- How responsive and fast are the messages dealt with?
- How easy is it for the user to connect to the server?

I've decided to evaluate my project based on these criteria. The user feedback I will access was received from the testers during the distribution phase. In general it was very positive, the testing age ranged from people aged 20 to 50.

Users have found the game very simple. One requirement that I asked of my users was that they knew how to play battleship. The use of the software appeared to be very straightforward as I was never asked to create tutorials and no individual has had trouble understanding what to do.

The finished game comprises graphics and sounds and runs smoothly. The experience of game-play allows the player control over what they want to do next such as to pause the game or return to the menu or simply shoot down the enemy ship. As well as this the game allows the user to mute sounds that some people could find irritating in an attempt to maximise the user's experience by empowering them.

The game is simple to install and run. When in game the user is prompted to chose if they would like to play either of the game types. When in game it is clear on what the user should be doing.

The game runs well on multiple platforms and all of the features proposed in the design have been implemented as well as some of the optional features.

In retrospect the process of making this game has thought me that the original design of some parts of the game are not efficient or very viable. I would have liked to do some things differently. This mean that the client software is not totally robust. A user with intent to break the game can still do so. Typically for games in a beta stage this is not an issue, the issue lies with the fix being to much of a rewrite as the problems with the AI for example are due to the entire structure of the game itself.

However the server is a robust piece of software it has been running for 3 weeks now in it's current Alpha 4 stage and has not created any problems and has not needed any restarts.

The server very rapidly deals with messages and a user can connect to the server without the need to do anything but press the multiplayer option and enter their user-name.

6 Learning Points

This experience has helped me learn how to implement a two dimensional video games. In some cases I've learnt through successful implementation of features such as my texture loader where I designed a scalable way of implementing a texture loading mechanism (see Section 4.1.2 *TextureManager*). In other cases I learnt through implementing poor code that does not perform very well, because of this I learnt what mistakes I made and why they are not good practice to implement for example the openGLManager. Though it works is not a good way of implementing the rendering of a 2D game.

I have also learnt that some of the features implemented have been poorly optimised and can be improved, mainly the graphical routines. Thankfully, the environment in which the development took place has allowed for programming mistakes and the occasional poorly made routine to be an opportunity enhance my understanding of how to create a 2d game. I have already undertaken a new game related project where I will be using SDL to create a 3D game in C++.

Future games will be made using SDL in C++ as Java has some downsides to creating games in a scalable way. There is little control over the memory and the half interpreted nature of Java means it's slow to run when compared with a language like C++.

Although multiplayer was running in alpha 3 I discouraged it as alpha 3 was prematurely released. This early release meant that I had not fixed all of the bugs discovered in version 2. I was attempting to add an unpolished bug filled software. The testers sent me the same

bug reports found in version 2 and keeping track of the origin of the bug became obfuscated by multiple versions. In future if I release a new patch the bugs found in the previous version will have to be addressed in-house first. Moreover, if I say I'm releasing a feature I will only release it if I feel it is finalised and well implemented.

There is one fundamental change in the client code I would need to re-plan, which would lead to a large rewrite of the project if I did this. In the code base there is no clear distinction between the game related logic code and the underlying "engine" code of the game. This is because the game engine was built along side the game itself. The game is hard wired into the engine. If I wanted to create a new game using the same managers and view structure I would not easily be able to use some of the engine code. Although, writing a new 2D game could be done realistically only I could do this as only I have a clear insight into the entirety of the code base. In future games I will make an effort to very distinctly separate the engine from the game. The game needs to call methods and not extend and inherit the methods. But more importantly the game's engine should be used like a library and the game handles the game specifics only.

7 Professional Issues

The design and purpose of the Application conforms to the British Computer Society Code of Conduct in the following respects:

- The rights, including Copyright, Patent and Intellectual Properties rights, of others have been investigated to the best of my ability to ensure the Application is not in breach.
- The Application makes no reference, implied or explicit, in relation to sex, ethnicity, religion or race.
- No claims are made for the purpose or usefulness of the Application.
- All work submitted as part of the Application is original and written solely by myself. I have not been aided directly or indirectly by another person in the production of this work.

8 Bibliography

References

[1] Lwjgl. `http://wiki.lwjgl.org/index.php?title=Main_Page`, 2013.

[2] Benny Bobaganoosh. 3d game engine input.java. `https://github.com/BennyQBD/3DGameEngine/blob/master/src/com/base/engine/core/Input.java`, 2014.

[3] Milton Bradley Co. Batteship for 2 players. http://www.hasbro.com/common/instruct/battleship.pdf, 1990.

[4] JMoneky3 Engine. Jmonkey engine explanation. http://jmonkeyengine.org/tour/introduction/, 2014.

[5] Jesse Maurais. Lwjgl openal tutorial. http://wiki.lwjgl.org/wiki/OpenAL_Tutorial_1_-_Single_Static_Source, 2012.

[6] SDL. Sdl wiki. http://wiki.libsdl.org/FrontPage, 2013.

[7] Oskar Veerhoek. Lwjgl timing - lwjgl tutorials. https://www.youtube.com/watch?v=0xsHwpzkNxQ/, 2011.

9 Appendices

The volume of code is too large to place in the appendix of this document. For direct links to the code please use the following urls.

The client code:

- http://louis.bennette.co.uk/downloads/battleships-client-source/

The server code:

- http://louis.bennette.co.uk/downloads/battleships-server-source/

The following classes of code are the classes from which the code segment figures have been taken from:

GameLoop.java

```
package battleships;

import battleships.engine.*;
import battleships.engine.events.TimeEventList;
import battleships.scene.SceneState;
import battleships.scene.view.IView;
import battleships.scene.view.credits.Credits;
import battleships.scene.view.menu.Menu;
import battleships.scene.view.multiplayer.MultiplayerGame;
import battleships.scene.view.options.Options;
import battleships.scene.view.singleplayer.SingleplayerGame;

/**
 *
 * @author Louis Bennette
 */
public class GameLoop {
    private static boolean closeRequested = false;
    private static boolean restartCurrentView = false;

    public static boolean isCloseRequested() {
        return closeRequested;
    }
    public static void setCloseRequested(boolean aCloseRequested) {
        closeRequested = aCloseRequested;
    }

    public static boolean isRestartCurrentView() {
        return restartCurrentView;
    }
    public static void setRestartCurrentView(boolean aRestartCurrentView) {
        restartCurrentView = aRestartCurrentView;
    }



    public GameLoop(){
        // Set game state.
        InputReader.initValidKeys();

        Time.initTime();

        SceneState.setGeneralState(SceneState.General.MENU);
    }
```

```
private IView currentSceneView;

/*
 * This statrs off the game loop. The game loop is made up of scene views.
 * There can only be one scene view active at any one time.
 * The sceneview is updated every loop.
 * One loop represents one frame in the game.
 */
public void start(){
    while(!Window.isCloseRequested()){
        OpenGLManager.clearBuffers();

        if(GameLoop.isRestartCurrentView()){
            restartCurrentView();
        }

        switch(SceneState.getGeneralState()){
            case MENU:
                if(currentSceneView == null || !(currentSceneView instanceof
                    Menu)){
                    TextureManager.flushTextures();
                    currentSceneView = new Menu();
                }
                break;
            case SINGLEPLAYER:
                if(currentSceneView == null || !(currentSceneView instanceof
                    SingleplayerGame)){
                    TextureManager.flushTextures();
                    currentSceneView = new SingleplayerGame();
                }
                break;
            case MULTIPLAYER:
                if(currentSceneView == null || !(currentSceneView instanceof
                    MultiplayerGame)){
                    TextureManager.flushTextures();
                    currentSceneView = new MultiplayerGame();
                }
                break;
            case OPTIONS:
                if(currentSceneView == null || !(currentSceneView instanceof
                    Options)){
                    TextureManager.flushTextures();
                    currentSceneView = new Options();
```

```
                }
                break;
            case CREDITS:
                if(currentSceneView == null || !(currentSceneView instanceof
                    Credits)){
                    TextureManager.flushTextures();
                    currentSceneView = new Credits();
                }
                break;
        }

        currentSceneView.update();
        currentSceneView.draw();

        Input.update();

        //All code conserning the game must be above the window update
        Window.update();
        Window.sync();

        // Handle any time based events.
        if(!TimeEventList.isEmpty()){
            TimeEventList.handleEvent();
            TimeEventList.cleanup();
        }

        Time.setDelta();

        // FPS TEST

        double timeSinceLastFrame = Time.getDelta();
        double fps = 60 / timeSinceLastFrame;
        //System.out.println(fps);


        if(GameGlobals.DEBUG_MODE){
            if(Input.getKey(Input.KEY_DEL)){
                GameLoop.setCloseRequested(true);
            }
        }

        if(GameLoop.isCloseRequested()){
            break;
        }
    }
```

```
    }

    private void restartCurrentView() {
        switch (SceneState.getGeneralState()) {
            case MENU:
                currentSceneView = new Menu();
                break;
            case SINGLEPLAYER:
                currentSceneView = new SingleplayerGame();
                break;
            case MULTIPLAYER:
                currentSceneView = new MultiplayerGame();
                break;
        }
        GameLoop.setRestartCurrentView(false);
    }
}
```

GameLogger.java

```
package battleships.engine;

import battleships.BattleshipsMain;
import java.io.IOException;
import java.util.Date;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;

/**
 *
 * @author Louis Bennette
 */
public class GameLogger {

    private static GameLogger loggingHandler;
    private FileHandler fileLocation;
    private Logger logger;
    private GameLogger(){
        try {
            fileLocation = new FileHandler("debug/lastSession.log");
            fileLocation.setFormatter(new SimpleFormatter());
            logger = Logger.getGlobal();
            logger.addHandler(fileLocation);
            logger.setUseParentHandlers(GameGlobals.DEBUG_MODE);

        } catch (IOException ex) {
            Logger.getLogger(GameLogger.class.getName()).log(Level.SEVERE, null,
                ex);
        } catch (SecurityException ex) {
            Logger.getLogger(GameLogger.class.getName()).log(Level.SEVERE, null,
                ex);
        }
    }

    private Logger getInstanceLogger(){
        return this.logger;
    }

    private static Logger getLogger(){
        if(loggingHandler == null){
            loggingHandler = new GameLogger();
        }
```

```
        return loggingHandler.getInstanceLogger();
    }

    public static void print(String message){
        getLogger().info(message);
    }

    public static void warning(String message){
        getLogger().warning(message);
    }

    public static void severe(String message){
        getLogger().severe(message);
    }
}
```

Input.java

```
package battleships.engine;

import java.util.ArrayList;
import org.lwjgl.input.Keyboard;
import org.lwjgl.input.Mouse;

/**
 *
 * @author Louis Bennette
 */
public class Input {
    public static final int KEY_LSHIFT = Keyboard.KEY_LSHIFT;
    public static final int KEY_RSHIFT = Keyboard.KEY_RSHIFT;
    public static final int KEY_R = Keyboard.KEY_R;
    public static final int KEY_M = Keyboard.KEY_M;
    public static final int KEY_P = Keyboard.KEY_P;
    public static final int KEY_DEL = Keyboard.KEY_DELETE;
    public static final int KEY_SPACE = Keyboard.KEY_SPACE;
    public static final int KEY_ESC = Keyboard.KEY_ESCAPE;
    public static final int KEY_BACK = Keyboard.KEY_BACK;
    public static final int KEY_ENTER = Keyboard.KEY_RETURN;


    public static final int MOUSE_LEFT = 0;
    public static final int MOUSE_RIGHT = 1;

    public static final int NUM_KEYCODES = 256;
    public static final int NUM_MOUSE_CODES = 5;

    private static ArrayList<Integer> lastKeys = new ArrayList<Integer>();
    private static ArrayList<Integer> lastMouseButtons = new ArrayList<Integer>();

    public static void update(){
        lastMouseButtons.clear();
        for(int i = 0; i < NUM_MOUSE_CODES; i++){
            if(getMouse(i)){
                lastMouseButtons.add(i);
            }
        }
        // This just stores all the keys that are currently pressed in this frame.
        lastKeys.clear();
        for(int i = 0; i < NUM_KEYCODES; i++){
            if(getKey(i)){
                lastKeys.add(i);
```

```
        }
    }
}

/**
 * Returns a boolean value if the tested key is currently held down live.
 * @param keyCode the key we want to check that is being held down.
 * @return true is the keyboard key is being held down false is not.
 */
public static boolean getKey(int keyCode){
    return Keyboard.isKeyDown(keyCode);
}

/**
 * Checks if the requested button has started being pressed in the current
 * frame, we know because if the key was previously not pressed and now is
 * then it's just been pressed.
 * @param keyCode the key we want to check that has just been pressed.
 * @return true if the key has just stared being pressed in this frame.
 */
public static boolean getKeyDown(int keyCode){
    return getKey(keyCode) && !lastKeys.contains(keyCode);
}

/**
 * Checks if the requested button has just been released in the current
 * frame, we know because if the key is currently now held down but
 * was held down in the previous frame, then it has been released.
 * @param keyCode the key we want to check that has just been released.
 * @return true if the key has just been released in this frame.
 */
public static boolean getKeyUp(int keyCode){
    return !getKey(keyCode) && lastKeys.contains(keyCode);
}


public static boolean getMouse(int keyCode){
    return Mouse.isButtonDown(keyCode);
}

public static boolean getMouseDown(int keyCode){
    return getMouse(keyCode) && !lastMouseButtons.contains(keyCode);
}

public static boolean getMouseUp(int keyCode){
```

```
        return !getMouse(keyCode) && lastMouseButtons.contains(keyCode);
    }

    public static boolean isInput(){
        return !lastKeys.isEmpty();
    }

    public static int[] getInput(){
        int[] tmp = new int[lastKeys.size()];
        int x = 0;
        for(int i: lastKeys){
            tmp[x++] = i;
        }
        return tmp;
    }
}
```

OpenGLManager.java

```
package battleships.engine;


import static org.lwjgl.opengl.GL11.*;
import org.newdawn.slick.Color;
import org.newdawn.slick.TrueTypeFont;
import org.newdawn.slick.opengl.Texture;
/**
 *
 * @author Louis Bennette
 */
public class OpenGLManager {
    public static void initOpenGL(){
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0, Window.getWidth(), Window.getHeight(), 0, 1, -1);
        glMatrixMode(GL_MODELVIEW);
        glEnable(GL_TEXTURE_2D);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        GameLogger.print("OpenGL initialised and ready.");
    }

    public static void clearBuffers(){
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
    public static void destroyOpenGl(){

    }

    public static void drawLine(Point startPoint, Point endPoint) {
    glBegin(GL_LINE_STRIP);
        glVertex2f((float)startPoint.getX(), (float)startPoint.getY());
        glVertex2f((float)endPoint.getX(), (float)endPoint.getY());
    glEnd();
}

    public static void drawSquare(float posX, float posY, float width, float
        height, Color color){
        glColor3f(color.r, color.g, color.b);
        glBegin(GL_QUADS);
            glVertex2f(posX, posY); // Upper Left;
            glVertex2f(posX + width, posY); // Upper Right;
            glVertex2f(posX + width, posY + height); // Bottom Right;
            glVertex2f(posX, posY + height); // Bottom Left;
```

```
        glEnd();
        glColor3f(0,0,0);
    }

    public static void drawSquareWithTex(float posX, float posY, float width,
        float height, Texture tex){
        tex.bind();
        glEnable(GL_BLEND);
        glBegin(GL_QUADS);
            glTexCoord2f(0, 0);
            glVertex2f(posX, posY); // Upper Left;
            glTexCoord2f(1, 0);
            glVertex2f(posX + width, posY); // Upper Right;
            glTexCoord2f(1, 1);
            glVertex2f(posX + width, posY + height); // Bottom Right;
            glTexCoord2f(0, 1);
            glVertex2f(posX, posY + height); // Bottom Left;
        glEnd();
        glDisable(GL_BLEND);
    }

    public static void drawSquareWithTexAtRightAngle(float posX, float posY,
        float width, float height, Texture tex){
        tex.bind();
        glEnable(GL_BLEND);
        glBegin(GL_QUADS);
            glTexCoord2f(0, 1);
            glVertex2f(posX, posY); // Upper Left;
            glTexCoord2f(0, 0);
            glVertex2f(posX + width, posY); // Upper Right;
            glTexCoord2f(1, 0);
            glVertex2f(posX + width, posY + height); // Bottom Right;
            glTexCoord2f(1, 1);
            glVertex2f(posX, posY + height); // Bottom Left;
        glEnd();
        glDisable(GL_BLEND);
    }

    // Development method. I use colours to tell different components apart.
    public static void setColor(int r, int g, int b) {
        glColor3f(r,g,b);
    }
}
```

TextureManager.java

```
package battleships.engine;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.newdawn.slick.opengl.Texture;
import org.newdawn.slick.opengl.TextureLoader;

/**
 *
 * @author Louis Bennette
 */
public class TextureManager {
    private static ArrayList<String> loadedTextures = new ArrayList<String>();
    private static Map<String, Texture> textureHolder = new Hashtable();
    private static final String path = "assets/textures/";
    private static final String extention = ".png";

    /**
     * Returns the texture with name given by the variable texName, returns
     * a pointer to the texture clones are not made.
     * @param texName the unique identifier of the texture e.g. button.png is
     * identified by the string  button
     * @return Texture pointer used to draw the screen
     */
    public static Texture loadTexture(String texName){
        try {
            Texture tex;
            if(isLoaded(texName)){
                // If texture is already in memory then return the pointer to it
                tex = textureHolder.get(texName);
            }
            else{
                // Load the new texture.
                tex = TextureLoader.getTexture("PNG", new FileInputStream(new
                    File(path + texName + extention)));
                textureHolder.put(texName, tex);
                loadedTextures.add(texName);
            }
```

```
            return tex;

        } catch (IOException ex) {
            GameLogger.warning("TextureManager ERROR, texture " + texName +
                extention + " failed to load.");
            return TextureManager.getDefaultTexture();
        }
    }

    /**
     * This is the default texture that is returned if a texture is not found.
     * If this texture is not found either the game will log a severe error and
     * crash.
     */
    private static Texture defaultTexture;
    /**
     * If the original texture is not found this method act as a fail safe so
     * that the game does not crash immediately.
     * @return the pointer to the default texture.
     */
    private static Texture getDefaultTexture(){
        if(TextureManager.defaultTexture == null){
            try {
                TextureManager.defaultTexture = TextureLoader.getTexture("PNG",
                    new FileInputStream(new File("assets/textures/default.png")));
                GameLogger.warning("Loaded default texture.");
            } catch (IOException ex) {
                Logger.getLogger(TextureManager.class.getName()).log(Level.SEVERE,
                    null, ex);
                GameLogger.severe("Could not find/load texture or default
                    texture.\nI is possible that the \"" + path + "\" path is
                    incorrect.\n"+ex.toString());
                ex.printStackTrace();
                System.exit(1);
            }
        }
        return TextureManager.defaultTexture;
    }

    /**
     * flushTextures is called to purge the current textures in memory, The
     * pointers are removed and the JVM will eventually remove the textures from
     * memory.
     * Note this does not affect the default texture.
     */
```

```
    public static void flushTextures(){
        loadedTextures = new ArrayList<String>();
        textureHolder = new Hashtable();
    }

    /**
     * Debug method used to return the names of the currently loaded textures.
     * @return the names of the currently loaded textures as a string.
     */
    public static String getListOfLoadedTextures(){
        String list = "";
        for(String item : loadedTextures){
            list = list + item + "\n";
        }
        return list;
    }

    /**
     * Checks if the texture is currently loaded in memory.
     * @param texName the name of the texture we are looking for
     * @return true if the texture is in memory and false if it is not.
     */
    private static boolean isLoaded(String texName){
        for(String key : loadedTextures){
            if(key.equals(texName)){
                return true;
            }
        }
        return false;
    }
}
```

Time.java

```
package battleships.engine;

/**
 * @author Louis Bennette
 */
public class Time {
    private static double delta;
    private static long thisFrameStartTime;
    private static long lastFrameStartTime;

    public static final long SECOND = 1000000000L;

    /**
     * Gets the current system time in nanoseconds as accurately as java
     * nanoTime() provides.
     * @return the time as a long
     */
    public static long getTime(){
        return System.nanoTime();
    }

    /**
     * Every frame is one loop of the game loop. For every loop we update the
     * time since the last loop. Delta represents the time between frames of the
     * game.
     * @return The time since the previous frame
     */
    public static double getDelta(){
        return delta;
    }

    /**
     * This is called for every loop of the game. For every frame in the game we
     * calculate the time since the previous frame. This should only be called
     * once every loop no more or no less.
     */
    public static void setDelta(){
        Time.thisFrameStartTime = Time.getTime();
        Time.delta = (double)(Time.thisFrameStartTime -
            Time.lastFrameStartTime)/SECOND;
        Time.lastFrameStartTime = Time.thisFrameStartTime;
    }

    /**
```

```
     * Must init the time just before the game loop starts or objects with
     * motion could, maybe, perhaps ping out of existance.
     */
    public static void initTime()
    {
        GameLogger.print("Started game time.");
        Time.lastFrameStartTime = Time.getTime();
    }
}
```

AudioManager.java

```
package battleships.engine.audio;


import battleships.engine.GameGlobals;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.nio.FloatBuffer;
import java.nio.IntBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.lwjgl.BufferUtils;
import org.lwjgl.LWJGLException;
import org.lwjgl.openal.AL;
import org.lwjgl.openal.AL10;
import org.lwjgl.util.WaveData;

/**
 *
 * @author Louis Bennette
 */
public class AudioManager {
    public AudioManager(){}

    public static final int NUM_BUFFERS = 20;
    public static final int NUM_SOURCES = 20;

    public static final SoundFile MENU_MUSIC = new SoundFile(0,
        "main-menu-music", SoundFile.Type.MUSIC);
    public static final SoundFile CLICK = new SoundFile(1, "click",
        SoundFile.Type.EFFECT);
    public static final SoundFile MISSILE_FLY = new SoundFile(2, "fly",
        SoundFile.Type.EFFECT);
    public static final SoundFile E_HIT = new SoundFile(3, "e-hit",
        SoundFile.Type.EFFECT);
    public static final SoundFile E_SINK = new SoundFile(4, "e-sink",
        SoundFile.Type.EFFECT);
    public static final SoundFile E_MISS = new SoundFile(5, "e-miss",
        SoundFile.Type.EFFECT);
    public static final SoundFile GAME_OVER_WIN = new SoundFile(6, "win",
        SoundFile.Type.EFFECT);
    public static final SoundFile GAME_OVER_LOOSE = new SoundFile(7, "loose",
        SoundFile.Type.EFFECT);
    public static final SoundFile PLACE_SHIP = new SoundFile(8, "place-ship",
```

```
        SoundFile.Type.EFFECT);
    public static final SoundFile GAME_MUSIC = new SoundFile(9, "game-music",
        SoundFile.Type.MUSIC);
    public static final SoundFile CREDITS = new SoundFile(10, "credits-music",
        SoundFile.Type.MUSIC);
    public static final SoundFile P_HIT = new SoundFile(11, "p-hit",
        SoundFile.Type.EFFECT);
    public static final SoundFile P_SINK = new SoundFile(12, "p-sink",
        SoundFile.Type.EFFECT);
    public static final SoundFile P_MISS = new SoundFile(13, "p-miss",
        SoundFile.Type.EFFECT);
    public static final SoundFile SECRET = new SoundFile(14, "secret",
        SoundFile.Type.EFFECT);
    public static final SoundFile EXPLOSION = new SoundFile(15, "explosion",
        SoundFile.Type.EFFECT);
    public static final SoundFile AMBIENCE = new SoundFile(16, "ambience",
        SoundFile.Type.MUSIC);

    private static final String path = "assets/sounds/";
    private static final String extention = ".wav";

    private static IntBuffer buffer = BufferUtils.createIntBuffer(NUM_BUFFERS);
    private static IntBuffer source = BufferUtils.createIntBuffer(NUM_SOURCES);
    private static FloatBuffer sourcePos =
        BufferUtils.createFloatBuffer(3*NUM_BUFFERS).put(new float[]{0.0f, 0.0f,
        0.0f});
    private static FloatBuffer sourceVel =
        BufferUtils.createFloatBuffer(3*NUM_BUFFERS).put(new float[]{0.0f, 0.0f,
        0.0f});
    private static FloatBuffer listenerPos =
        BufferUtils.createFloatBuffer(3).put(new float[]{0.0f, 0.0f, 0.0f});
    private static FloatBuffer listenerVel =
        BufferUtils.createFloatBuffer(3).put(new float[]{0.0f, 0.0f, 0.0f});
    private static FloatBuffer listenerOri =
        BufferUtils.createFloatBuffer(6).put(new float[]{0.0f, 0.0f, -1.0f, 0.0f,
        1.0f, 0.0f});

    public static void initOpenAL(){
        try {
            AL.create();
        } catch (LWJGLException ex) {
            Logger.getLogger(AudioManager.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }
```

```
public static void destroyOpenAL(){
    AL10.alDeleteSources(source);
    AL10.alDeleteBuffers(buffer);
    AL.destroy();
}

public static void createSounds(){
    try {
        if (loadSoundFiles() == AL10.AL_FALSE) {
            System.err.println("Error loading sound data.");
            System.exit(1);
        }
    } catch (FileNotFoundException ex) {
    }

}

private static int loadSoundFiles() throws FileNotFoundException {
    AL10.alGenBuffers(buffer);
    if(AL10.alGetError() != AL10.AL_NO_ERROR){
        System.err.println("Buffer Error");
        return AL10.AL_FALSE;
    }
    WaveData waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + MENU_MUSIC.getFileName() + extention)));
    AL10.alBufferData(buffer.get(MENU_MUSIC.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + CLICK.getFileName() + extention)));
    AL10.alBufferData(buffer.get(CLICK.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + MISSILE_FLY.getFileName() + extention)));
    AL10.alBufferData(buffer.get(MISSILE_FLY.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + E_HIT.getFileName() + extention)));
    AL10.alBufferData(buffer.get(E_HIT.getID()), waveFile.format,
```

```
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + E_SINK.getFileName() + extention)));
    AL10.alBufferData(buffer.get(E_SINK.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + E_MISS.getFileName() + extention)));
    AL10.alBufferData(buffer.get(E_MISS.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + GAME_OVER_WIN.getFileName() + extention)));
    AL10.alBufferData(buffer.get(GAME_OVER_WIN.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + GAME_OVER_LOOSE.getFileName() + extention)));
    AL10.alBufferData(buffer.get(GAME_OVER_LOOSE.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + PLACE_SHIP.getFileName() + extention)));
    AL10.alBufferData(buffer.get(PLACE_SHIP.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + GAME_MUSIC.getFileName() + extention)));
    AL10.alBufferData(buffer.get(GAME_MUSIC.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();

    waveFile = WaveData.create(new BufferedInputStream(new
        FileInputStream(path + CREDITS.getFileName() + extention)));
    AL10.alBufferData(buffer.get(CREDITS.getID()), waveFile.format,
        waveFile.data, waveFile.samplerate);
    waveFile.dispose();
```

```
        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + P_HIT.getFileName() + extention)));
        AL10.alBufferData(buffer.get(P_HIT.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + P_SINK.getFileName() + extention)));
        AL10.alBufferData(buffer.get(P_SINK.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + P_MISS.getFileName() + extention)));
        AL10.alBufferData(buffer.get(P_MISS.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + SECRET.getFileName() + extention)));
        AL10.alBufferData(buffer.get(SECRET.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + EXPLOSION.getFileName() + extention)));
        AL10.alBufferData(buffer.get(EXPLOSION.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        waveFile = WaveData.create(new BufferedInputStream(new
            FileInputStream(path + AMBIENCE.getFileName() + extention)));
        AL10.alBufferData(buffer.get(AMBIENCE.getID()), waveFile.format,
            waveFile.data, waveFile.samplerate);
        waveFile.dispose();

        AL10.alGenSources(source);
        if (AL10.alGetError() != AL10.AL_NO_ERROR){
            System.err.println("Sources Error");
            return AL10.AL_FALSE;
        }

        AL10.alSourcei(source.get(MENU_MUSIC.getID()), AL10.AL_BUFFER,
            buffer.get(MENU_MUSIC.getID()));
        AL10.alSourcei(source.get(CLICK.getID()),      AL10.AL_BUFFER,
```

```
            buffer.get(CLICK.getID()));
        AL10.alSourcei(source.get(MISSILE_FLY.getID()), AL10.AL_BUFFER,
            buffer.get(MISSILE_FLY.getID()));
        AL10.alSourcei(source.get(E_HIT.getID()),      AL10.AL_BUFFER,
            buffer.get(E_HIT.getID()));
        AL10.alSourcei(source.get(E_SINK.getID()),     AL10.AL_BUFFER,
            buffer.get(E_SINK.getID()));
        AL10.alSourcei(source.get(E_MISS.getID()),     AL10.AL_BUFFER,
            buffer.get(E_MISS.getID()));
        AL10.alSourcei(source.get(GAME_OVER_WIN.getID()), AL10.AL_BUFFER,
            buffer.get(GAME_OVER_WIN.getID()));
        AL10.alSourcei(source.get(GAME_OVER_LOOSE.getID()), AL10.AL_BUFFER,
            buffer.get(GAME_OVER_LOOSE.getID()));
        AL10.alSourcei(source.get(PLACE_SHIP.getID()), AL10.AL_BUFFER,
            buffer.get(PLACE_SHIP.getID()));
        AL10.alSourcei(source.get(GAME_MUSIC.getID()), AL10.AL_BUFFER,
            buffer.get(GAME_MUSIC.getID()));
        AL10.alSourcei(source.get(CREDITS.getID()),    AL10.AL_BUFFER,
            buffer.get(CREDITS.getID()));
        AL10.alSourcei(source.get(P_HIT.getID()),      AL10.AL_BUFFER,
            buffer.get(P_HIT.getID()));
        AL10.alSourcei(source.get(P_SINK.getID()),     AL10.AL_BUFFER,
            buffer.get(P_SINK.getID()));
        AL10.alSourcei(source.get(P_MISS.getID()),     AL10.AL_BUFFER,
            buffer.get(P_MISS.getID()));
        AL10.alSourcei(source.get(SECRET.getID()),     AL10.AL_BUFFER,
            buffer.get(SECRET.getID()));
        AL10.alSourcei(source.get(EXPLOSION.getID()),  AL10.AL_BUFFER,
            buffer.get(EXPLOSION.getID()));
        AL10.alSourcei(source.get(AMBIENCE.getID()),   AL10.AL_BUFFER,
            buffer.get(AMBIENCE.getID()));


        if (AL10.alGetError() == AL10.AL_NO_ERROR){
            return AL10.AL_TRUE;
        }
        return AL10.AL_FALSE;
    }

    public static void play(SoundFile soundFile){
        if(GameGlobals.gameMusicOn && soundFile.getSoundType() ==
            SoundFile.Type.MUSIC){
            AL10.alSourcePlay(source.get(soundFile.getID()));
        }
        else if(GameGlobals.gameEffectSoundsOn && soundFile.getSoundType() ==
```

```
            SoundFile.Type.EFFECT){
            AL10.alSourcePlay(source.get(soundFile.getID()));
        }
    }

    public static void stop(SoundFile soudFile) {
        AL10.alSourceStop(source.get(soudFile.getID()));
    }

    public static void pause(SoundFile soudFile) {
        AL10.alSourcePause(source.get(soudFile.getID()));
    }
}
```

IEvent.java

```
package battleships.engine.events;

/**
 *
 * @author Louis Bennette
 */
public interface IEvent {
    public String getName();
    public void onEvent();
}
```

AbstractButton.java

```
package battleships.scene.components.button;

import battleships.engine.Input;
import battleships.engine.OpenGLManager;
import battleships.engine.Point;
import battleships.engine.TextureManager;
import battleships.engine.events.IEvent;
import battleships.engine.events.IEventResponder;
import battleships.scene.components.AbstractComponent;
import battleships.scene.components.label.Label;
import org.newdawn.slick.opengl.Texture;

/**
 *
 * @author Louis Bennette
 */
public abstract class AbstractButton extends AbstractComponent implements IEvent
    {
    private boolean buttonClicked;
    /**
     * We only load one static texture for all buttons.
     */
    private Texture hoverTex;
    /**
     * We only load one static texture for all buttons.
     */
    private Texture clickedTex;

    private Label textLabel;

    protected IEventResponder eventResponder;

    protected AbstractButton(double x, double y, double width, double height,
        String name){
        this.origin = new Point(x, y);
        this.width = width;
        this.height = height;
        this.name = name;
        this.buttonClicked = false;

        this.textLabel = new Label((x+10), y, (width-20), height, (name+"Label"));
        this.textLabel.setText(name);
        this.textLabel.textAlignCentered();
```

```
        this.setTexture(TextureManager.loadTexture("generic-button"));
        hoverTex = TextureManager.loadTexture("button-hover");
        clickedTex = TextureManager.loadTexture("button-clicked");
    }


    @Override
    public void draw() {
        //System.out.println("Drawing a button");
        if (this.isVisible()) {
            Texture drawTex = this.getTexture();
            if (this.hasMouseHover() && !this.buttonClicked) {
                drawTex = this.getHoverTex();
            } else if (this.buttonClicked) {
                drawTex = this.getClickedTex();
            }
            OpenGLManager.drawSquareWithTex((float) this.getX(), (float)
                this.getY(), (float) this.getWidth(), (float) this.getHeight(),
                drawTex);
            //System.out.println("!!!!!!!!!!!" + this.getX() + "," + this.getY());

            this.getButtonLabel().draw();
        }
    }

    @Override
    public void update() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    /**
     * A button is clicked when it has been pressed and released in the bounds
     * of the button. This is used to execute the event of a button.
     * @return true is the button has been clicked.
     */
    public boolean isClicked(){
        // We check first to see if the button has been pressed.
        if(this.hasMouseHover()){
            if(Input.getMouseDown(Input.MOUSE_LEFT)){
                // Button is pressed
                this.buttonClicked = true;
            }
        }
        // Now we check if the button is released withing the button's bounds.
        if(Input.getMouseUp(Input.MOUSE_LEFT)){
```

```
            if(this.buttonClicked && this.hasMouseHover()){
                this.buttonClicked = false;
                // Only return true if the button is pressed and released in the
                // bounds of the button.
                return true;
            }
            else
            {
                this.buttonClicked = false;
            }
        }
        return false;
    }

    /**
     * Set the name of the component.
     * @param name the name as a string.
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Sets the text of the label on the button.
     * @param text The text that is drawn to screen of the button.
     */
    public void setText(String text){
        this.getButtonLabel().setText(text);
    }

    /**
     * Retrieve the text of the button. The text is held by the label, this is a
     * nested call to the button's label.
     * @return the text of the button.
     */
    public String getText(){
        return this.getButtonLabel().getText();
    }

    /**
     * This is the Label object held by the button this hold all the text
     * information of the button
     * @return a Label object of the button to be manipulated if needed.
     */
    public Label getButtonLabel() {
```

```
        return textLabel;
    }

    /**
     * @return the hoverTex
     */
    public Texture getHoverTex() {
        return hoverTex;
    }
    /**
     * @param aHoverTex the hoverTex to set
     */
    public void setHoverTex(Texture aHoverTex) {
        hoverTex = aHoverTex;
    }

    /**
     * @return the clickedTex
     */
    public Texture getClickedTex() {
        return clickedTex;
    }
    /**
     * @param aClickedTex the clickedTex to set
     */
    public void setClickedTex(Texture aClickedTex) {
        clickedTex = aClickedTex;
    }
}
```

MultiplayerGame.java

```
package battleships.scene.view.multiplayer;

import battleships.engine.*;
import battleships.engine.audio.AudioManager;
import battleships.engine.events.IEvent;
import battleships.engine.events.MessageRecievedEvent;
import battleships.engine.events.TimeEvent;
import battleships.engine.events.TimeEventList;
import battleships.scene.SceneState;
import battleships.scene.components.button.Button;
import battleships.scene.components.grid.AbstractGrid;
import battleships.scene.components.ship.AbstractShip;
import battleships.scene.view.AbstractGame;
import battleships.scene.view.player.UserPlayer;
import battleships.scene.view.popupview.GameMenu;
import battleships.scene.view.popupview.MessagePopup;
import battleships.scene.view.popupview.UsernamePopup;
import java.io.IOException;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Deahgib
 */
public class MultiplayerGame extends AbstractGame{
    private MessagePopup messagePopup;

    private Socket socket;
    private MessageListener messageListener;
    private Thread messageListenerThread;
    private UserPlayer userPlayer;
    private String opponentName;
    private boolean madeAShot;

    public MultiplayerGame(){
        super();

        this.userPlayer = new UserPlayer();

        this.bannerLabel.setText("Multiplayer");
```

```
        Button backButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight()*2, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight(), dimentions.getGridCellHeight()*2,
            dimentions.getGridCellHeight(), "Back", this);
        Button readyButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight()*5, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight(), dimentions.getGridCellHeight()*2,
            dimentions.getGridCellHeight(), "Ready", this);
        readyButton.getButtonLabel().setFontSize(28f);

        this.gameButtons.add(backButton);
        this.gameButtons.add(readyButton);

        Button notreadyButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight() * 2, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight(), dimentions.getGridCellHeight() * 2,
            dimentions.getGridCellHeight(), "Notready", this);
        notreadyButton.setVisible(false);
        notreadyButton.setText("Not Ready");
        this.gameButtons.add(notreadyButton);

        Button fireButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight() * 4, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight() * 3, dimentions.getGridCellHeight() *
            4, dimentions.getGridCellHeight() * 3, "Fire", this);
        fireButton.setVisible(false);
        this.gameButtons.add(fireButton);


        GameLogger.print("Singleplayer game object successfully created.");

        AudioManager.play(AudioManager.GAME_MUSIC);
        //AudioManager.play(AudioManager.AMBIENCE);
        //TimeEventList.add(new TimeEvent(this, 30 * Time.SECOND,
            "ambienceLooper"));
        UsernamePopup child = new UsernamePopup(this, true);
        this.addChildView(child);
        this.setActiveSubview(child);

         messagePopup = new MessagePopup(this, true, "", "");
         this.addChildView(this.messagePopup);

         this.opponentName = "Opponent";
         this.madeAShot = false;
    }
```

```
@Override
public void update() {
    if (this.shouldUpdate()) {
        this.gameButtons.handleClickEvent();
        switch (SceneState.getPlayerState()) {
            case QUEUED:
                break;
            case SHIP_PLACEMENT:
                this.userPlayer.updateShipPlacement();
                if(this.userPlayer.isMovingShip()){
                    this.bannerLabel.setText("'Right Click' or 'R' to rotate");
                }
                else{
                    this.bannerLabel.setText("Place your ships");
                }
                break;

            case WAIT_FOR_OPONENT:
                break;

            case USER_TURN:
                //System.out.println("My turn");
                if (!this.madeAShot) {
                    this.userPlayer.upadteSelectedTarget();
                    if (Input.getKeyUp(Input.KEY_SPACE)) {
                        TimeEventList.add(new TimeEvent(this, "Fire"));
                    }
                }
                break;

            case OPONENT_TURN:

                break;

            case GAME_OVER:

                break;
        }

        if (Input.getKeyUp(Input.KEY_ESC)) {
            if(this.getActiveSubview() == null){
                GameMenu child = new GameMenu(this, false);
                this.addChildView(child);
                this.setActiveSubview(child);
```

```
            }
        }
    }

    if(SceneState.getPlayerState() == SceneState.Player.ANIMATING) {
        if (this.getAnimation() != null) {
            if (this.getAnimation().isAnimationStarted()) {
                if (this.getAnimation().isAnimationEnded()) {
                    this.endAnimation();
                    //System.out.println("Clear animation");
                } else {
                    this.getAnimation().updateAnimation();
                }
            } else {
                this.getAnimation().start();
            }
        }
    }

    this.updateSubviews();
}

@Override
public void draw(){
    this.background.draw();

    SceneState.Player state = SceneState.getPlayerState();
    if (this.getAnimation() != null) {
        if (this.getAnimation().isAnimationStarted()) {
            state = this.getAnimation().getPreviousState();
        }
    }

    switch (state) {
        case GAME_OVER:
            break;
        case QUEUED:
        case WAIT_FOR_OPONENT:
        case SHIP_PLACEMENT:
            this.userPlayer.getShipHolder().draw();
        case OPONENT_TURN: // On opponent turn show the user's fleet.
            this.userPlayer.getFleetGrid().draw();
            for (AbstractShip ship : this.userPlayer.getShips()) {
                if(ship.isShipBeingMoved() &&
                    this.userPlayer.getFleetGrid().hasMouseHover()){
```

```
                    Point p =
                        ship.getShipHoverPositionForGrid(this.userPlayer.getFleetGrid());
                    if(this.userPlayer.getFleetGrid().isPositionValidFor(ship,
                        (int)p.getX(), (int)p.getY())){
                        this.userPlayer.getFleetGrid().drawShipHoverBox(ship,
                            (int)p.getX(), (int)p.getY());
                    }
                }
                ship.draw();
            }
            this.userPlayer.getFleetGrid().drawDecals();
            break;

        case USER_TURN: // On users turn show the user's attempts/target grid.
            this.userPlayer.getTargetGrid().draw();
            break;
    }

    if (this.getAnimation() != null) {
        if (this.getAnimation().isAnimationStarted()) {
            this.getAnimation().drawAnimation();
        }
    }

    this.bannerLabel.draw();
    this.shortMessageLabel.draw();
    this.gameButtons.draw();

    this.drawSubviews();
}

@Override
public void handleEvent(IEvent event) {
    GameLogger.print("Singleplayer event \"" + event.getName() + "\" was
        dispached.");
    switch (event.getName()) {
        case "Back":
            SceneState.setGeneralState(SceneState.General.MENU);
            this.disconnectFromServer();
            break;
        case "Ready":
            if (this.userPlayer.getFleetGrid().isGridFilled()) {
                MessageIOHandler.sendServerCommand("user-ready", this.socket);
            }
            break;
```

```
            case "Notready":
                if(this.userPlayer.isReady()){
                    MessageIOHandler.sendServerCommand("user-not-ready",
                        this.socket);
                    this.userPlayer.setReady(false);
                }
                break;
            case "Fire":
                if (SceneState.getPlayerState() == SceneState.Player.USER_TURN &&
                    !this.madeAShot) {
                    if (this.userPlayer.getTargetGrid().getSelected() != null) {
                        int selx = (int)
                            this.userPlayer.getTargetGrid().getSelected().getX();
                        int sely = (int)
                            this.userPlayer.getTargetGrid().getSelected().getY();
                        MessageIOHandler.sendPlayerCommand(("try=" + selx + "," +
                            sely), this.socket);
                        this.madeAShot = true;
                    }
                }
                break;
            case "opponentToUser":
                break;
            case "depause":
                this.getAnimation().end();
                break;
            case "messageRecieved":
                MessageRecievedEvent mre = (MessageRecievedEvent)event;
                this.handleMessageRecieved(mre.getMessage());
                break;
        }
    }

    public void setSocket(Socket s){
        this.socket = s;
    }

    public Socket getSocket(){
        return this.socket;
    }

    public void setMessageListener(MessageListener ml, Thread mlThread){
        this.messageListener = ml;
        this.messageListenerThread = mlThread;
    }
```

```
public void disconnectFromServer() {
    MessageIOHandler.sendServerCommand("dco", this.socket);
    try {
        this.socket.close();
    } catch (IOException ex) {
    }
}

private void handleMessageRecieved(String message){
    int type = Parser.getMessageType(message);
    message = Parser.getMessage(message);

    if (type == Parser.PLAYERtoPLAYER || type == Parser.SERVERtoPLAYER) {
        if (Parser.doesMessageHaveVariable(message)) {
            String variable = Parser.getMessageVariable(message);
            message = Parser.truncateVariable(message);

            switch (message) {
                case "op-name":
                    this.opponentName = variable;
                    this.bannerLabel.setText("Opponent found: " +
                        this.opponentName);
                    break;
                case "state-turn":
                    if (variable.equals("0")) {
                        SceneState.setPlayerState(SceneState.Player.OPONENT_TURN);
                        this.bannerLabel.setText(this.opponentName+"'s turn");
                    } else {
                        SceneState.setPlayerState(SceneState.Player.USER_TURN);
                        this.bannerLabel.setText("Your turn");
                    }
                    this.madeAShot = false;
                    this.shortMessageLabel.setVisible(true);
                    this.shortMessageLabel.setText("");
                    this.gameButtons.removeButtonWithName("Notready");
                    this.gameButtons.removeButtonWithName("Ready");
                    this.gameButtons.removeButtonWithName("Back");
                    this.gameButtons.getButtonWithName("Fire").setVisible(true);
                    break;
                case "try":
                    if (SceneState.getPlayerState() ==
                        SceneState.Player.OPONENT_TURN) {
                        String[] s = variable.split(",");
                        int x = Integer.parseInt(s[0]);
```

```
                    int y = Integer.parseInt(s[1]);

                    if (this.userPlayer.getFleetGrid().isHit(x, y)) {
                        this.userPlayer.getFleetGrid().registerAttempt(x,
                            y, AbstractGrid.HitMarker.HIT);
                        if (this.userPlayer.getFleetGrid().isSink(x, y)) {
                            GameLogger.print("Online opponent sunk player's
                                " +
                                this.userPlayer.getFleetGrid().getShipAt(x,
                                y).getName());
                            this.shortMessageLabel.setText(this.opponentName+"
                                sunk your " +
                                this.userPlayer.getFleetGrid().getShipAt(x,
                                y).getName());
                            AudioManager.stop(AudioManager.E_HIT);
                            AudioManager.play(AudioManager.E_SINK);
                            if (!this.hasPlayerLost(userPlayer)) {
                                MessageIOHandler.sendPlayerCommand(("sink="
                                    +
                                    this.userPlayer.getFleetGrid().getShipAt(x,
                                    y).getName()), this.socket);
                            } else {
                                MessageIOHandler.sendServerCommand("user-lost",
                                    this.socket);
                            }
                        } else {
                            GameLogger.print("Online Opponent HIT player at
                                x=" + x + ", y=" + y);
                            // Update the view's model.
                            AudioManager.play(AudioManager.E_HIT);
                            MessageIOHandler.sendPlayerCommand("hit",
                                this.socket);
                        }
                    } else {
                        MessageIOHandler.sendPlayerCommand("miss",
                            this.socket);
                        this.userPlayer.getFleetGrid().registerAttempt(x,
                            y, AbstractGrid.HitMarker.MISS);
                        AudioManager.play(AudioManager.E_MISS);
                    }
                }
                break;
            case "sink":
                if (SceneState.getPlayerState() ==
                    SceneState.Player.USER_TURN && this.madeAShot) {
```

```
                    if (this.userPlayer.getTargetGrid().getSelected() !=
                        null) {
                        int x = (int)
                            this.userPlayer.getTargetGrid().getSelected().getX();
                        int y = (int)
                            this.userPlayer.getTargetGrid().getSelected().getY();
                        this.shortMessageLabel.setText("You sunk their " +
                            variable);
                        this.userPlayer.getTargetGrid().registerAttempt(x,
                            y, AbstractGrid.HitMarker.HIT);
                        AudioManager.play(AudioManager.P_SINK);
                        this.userPlayer.getTargetGrid().resetSelected();
                        MessageIOHandler.sendServerCommand("next-turn",
                            this.socket);
                    }
                }
                break;
            case "game-over":
                this.shortMessageLabel.setText("");
                GameLogger.print("Adding view");
                this.messagePopup.setTitle("GAME OVER");
                if (variable.equals("0")) {
                    if (this.getActiveSubview() == null) {
                        this.messagePopup.setMessage("You lose.");
                        AudioManager.stop(AudioManager.E_SINK);
                        AudioManager.play(AudioManager.GAME_OVER_LOOSE);
                    }
                } else {
                    if (this.getActiveSubview() == null) {
                        this.messagePopup.setMessage("You defeated " +
                            this.opponentName + "!");
                        AudioManager.stop(AudioManager.P_SINK);
                        AudioManager.play(AudioManager.GAME_OVER_WIN);
                    }
                }
                this.setActiveSubview(this.messagePopup);
                break;
        }
    } else {
        switch (message) {
            case "in-queue":
                this.bannerLabel.setText("Queued for a game.");
                SceneState.setPlayerState(SceneState.Player.QUEUED);
                break;
            case "hit":
```

```
                    if (SceneState.getPlayerState() ==
                        SceneState.Player.USER_TURN && this.madeAShot) {
                        if (this.userPlayer.getTargetGrid().getSelected() !=
                            null) {
                            int x = (int)
                                this.userPlayer.getTargetGrid().getSelected().getX();
                            int y = (int)
                                this.userPlayer.getTargetGrid().getSelected().getY();
                            GameLogger.print("Player HIT AI at x=" + x + ", y="
                                + y);
                            this.shortMessageLabel.setText("Hit!");
                            this.userPlayer.getTargetGrid().registerAttempt(x,
                                y, AbstractGrid.HitMarker.HIT);
                            AudioManager.play(AudioManager.P_HIT);
                            this.userPlayer.getTargetGrid().resetSelected();
                            MessageIOHandler.sendServerCommand("next-turn",
                                this.socket);
                        }
                    }
                    break;
                case "miss":
                    if (SceneState.getPlayerState() ==
                        SceneState.Player.USER_TURN && this.madeAShot) {
                        if (this.userPlayer.getTargetGrid().getSelected() !=
                            null) {
                            int x = (int)
                                this.userPlayer.getTargetGrid().getSelected().getX();
                            int y = (int)
                                this.userPlayer.getTargetGrid().getSelected().getY();
                            this.shortMessageLabel.setText("Miss");
                            this.userPlayer.getTargetGrid().registerAttempt(x,
                                y, AbstractGrid.HitMarker.MISS);
                            AudioManager.play(AudioManager.P_MISS);
                            this.userPlayer.getTargetGrid().resetSelected();
                            MessageIOHandler.sendServerCommand("next-turn",
                                this.socket);
                        }
                    }
                    break;
                case "state-ship-placement":
                    if (SceneState.getPlayerState() !=
                        SceneState.Player.SHIP_PLACEMENT) {
                        this.gameButtons.getButtonWithName("Notready").setVisible(false);
                        this.gameButtons.getButtonWithName("Ready").setVisible(true);
                        this.gameButtons.getButtonWithName("Back").setVisible(true);
```

```
                        SceneState.setPlayerState(SceneState.Player.SHIP_PLACEMENT);
                        this.userPlayer.setReady(false);
                    }
                    break;
                case "state-wait-for-opponent":
                    SceneState.setPlayerState(SceneState.Player.WAIT_FOR_OPONENT);
                    this.gameButtons.getButtonWithName("Notready").setVisible(true);
                    this.gameButtons.getButtonWithName("Ready").setVisible(false);
                    this.gameButtons.getButtonWithName("Back").setVisible(false);
                    this.userPlayer.setReady(true);
                    this.bannerLabel.setText("Waiting for opponent");
                    break;
                case "usr-dco":
                    messageListenerThread.interrupt();
                    messageListenerThread = null;
                    SceneState.setGeneralState(SceneState.General.MENU);
                    this.disconnectFromServer();
                    break;

                case "op-dco":
                    this.messagePopup.setTitle("Opponent Disconnected");
                    this.messagePopup.setMessage("Your opponent has
                        disconnected from the game.");
                    this.setActiveSubview(this.messagePopup);
                    break;

                case "fatal":
                    this.messagePopup.setTitle("Fatal Server Error");
                    this.messagePopup.setMessage("There has been an error on
                        the server, please contact the server admin.");
                    this.setActiveSubview(this.messagePopup);
                    break;
            }
        }
    } else if (type == Parser.CHAT) {
        // Do nothing chat not implemented yet.
    } else {
        GameLogger.severe("Suspected tampering message " + type + message + "
            recieved from server.");
    }
  }

}
```

SingleplayerGame.java

```
package battleships.scene.view.singleplayer;

import battleships.engine.*;
import battleships.engine.animation.MissileAnimation;
import battleships.engine.animation.PauseAnimation;
import battleships.engine.audio.AudioManager;
import battleships.engine.events.IEvent;
import battleships.engine.events.TimeEvent;
import battleships.engine.events.TimeEventList;
import battleships.scene.SceneState;
import battleships.scene.components.button.Button;
import battleships.scene.components.grid.AbstractGrid;
import battleships.scene.components.ship.AbstractShip;
import battleships.scene.view.AbstractGame;
import battleships.scene.view.player.ComputerAIPlayer;
import battleships.scene.view.player.UserPlayer;
import battleships.scene.view.popupview.GameMenu;
import battleships.scene.view.popupview.GameOver;

/**
 *
 * @author Louis Bennette
 */
public class SingleplayerGame extends AbstractGame{
    private UserPlayer userPlayer;
    private ComputerAIPlayer enemyPlayer;

    public SingleplayerGame(){
        super();

        this.userPlayer = new UserPlayer();
        this.enemyPlayer = new ComputerAIPlayer();

        // Debug
        this.enemyPlayer.initAIShips();

        Button backButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight()*2, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight(), dimentions.getGridCellHeight()*2,
            dimentions.getGridCellHeight(), "Back", this);
        Button readyButton = new Button(dimentions.getRightMargin() -
            dimentions.getGridCellHeight()*5, dimentions.getBottomMargin() -
            dimentions.getGridCellHeight(), dimentions.getGridCellHeight()*2,
            dimentions.getGridCellHeight(), "Ready", this);
```

```
        readyButton.getButtonLabel().setFontSize(28f);

        this.gameButtons.add(backButton);
        this.gameButtons.add(readyButton);

        GameLogger.print("Singleplayer game object successfully created.");

        AudioManager.play(AudioManager.GAME_MUSIC);
        //AudioManager.play(AudioManager.AMBIENCE);
        //TimeEventList.add(new TimeEvent(this, 30 * Time.SECOND,
            "ambienceLooper"));
    }

    /**
     * This is the main update call used in singleplayer. This update loop will
     * call subsequent update loops belonging to components of singleplayer.
     * This method is called once a frame, it will handle the update of
     * component locations, user interactions with components and update called
     * to child views if active as well as animations.
     * It knows what calls to make based on the state the game is currently in.
     */
    @Override
    public void update() {
        if (this.shouldUpdate()) {
            this.gameButtons.handleClickEvent(); // Everyframe handle button
                clicks
            switch (SceneState.getPlayerState()) {
                case SHIP_PLACEMENT: // Allow the player in this state to move
                    ships around and place them.
                    this.userPlayer.updateShipPlacement();
                    if(this.userPlayer.isMovingShip()){
                        this.bannerLabel.setText("'Right Click' or 'R' to rotate");
                    }
                    else{
                        this.bannerLabel.setText("Place your ships");
                    }
                    break;

                case WAIT_FOR_OPONENT:
                    // In this state the user's ships are locked
                    // The AI's ships are allways ready, so we start the game.
                    if (this.userPlayer.isReady() && this.enemyPlayer.isReady()) {
                        this.shortMessageLabel.setVisible(true);
                        // Determine who goes first.
                        java.util.Random n = new java.util.Random();
```

```
                int i = n.nextInt(2);
                if (i == 0) {
                    SceneState.setPlayerState(SceneState.Player.USER_TURN);
                    this.bannerLabel.setText("Your turn");
                } else {
                    SceneState.setPlayerState(SceneState.Player.OPONENT_TURN);
                    this.bannerLabel.setText("Opponent's turn");
                }
                this.gameButtons.removeButtonWithName("Notready");
                this.gameButtons.add(new Button(dimentions.getRightMargin()
                    - dimentions.getGridCellHeight() * 4,
                    dimentions.getBottomMargin() -
                    dimentions.getGridCellHeight() * 3,
                    dimentions.getGridCellHeight() * 4,
                    dimentions.getGridCellHeight() * 3, "Fire", this));
            } else {
                this.enemyPlayer.initAIShips();
            }
            break;

        case USER_TURN:
            // This gives the user control where they would like to
            // fire at their opponent.
            this.userPlayer.upadteSelectedTarget();
            if(Input.getKeyUp(Input.KEY_SPACE)){
                TimeEventList.add(new TimeEvent(this, "Fire"));
            }
            break;

        case OPONENT_TURN:
            // This is when the AI kicks in and decides what it's next
            // target is.
            this.enemyPlayer.findNextCellToAttack();
            if (this.enemyPlayer.getTargetGrid().getSelected() != null) {

                this.setAnimation(new MissileAnimation(this,
                        SceneState.Player.USER_TURN,
                        SceneState.getPlayerState(),
                        this.enemyPlayer.getTargetGrid().getSelected(),
                        true, (Time.SECOND * 1)));
                SceneState.setPlayerState(SceneState.Player.ANIMATING);
                this.getAnimation().start();
            }
            break;
```

```
            case GAME_OVER:
                //System.out.println("Game Over");
                break;
        }

        // ============================================================
        //                  GAME OVER DETECTED
        // ============================================================
        if (this.isGameOver(this.userPlayer, this.enemyPlayer)) {
            TimeEventList.add(new TimeEvent(this, "gameOver"));
        }

        // Debug Game Over popup test
        if (GameGlobals.DEBUG_MODE) {
            if (Input.getKeyUp(Input.KEY_P)) {
                if (this.getActiveSubview() == null) {
                    GameOver child = new GameOver(this, true, "You cheated!");
                    this.addChildView(child);
                    this.setActiveSubview(child);
                }
            }
        }

        // Open the Menu when "Esc" is pressed.
        if (Input.getKeyUp(Input.KEY_ESC)) {
            if(this.getActiveSubview() == null){
                GameMenu child = new GameMenu(this, true);
                this.addChildView(child);
                this.setActiveSubview(child);
            }
        }
    }

    if(SceneState.getPlayerState() == SceneState.Player.ANIMATING) {
        if (this.getAnimation() != null) { // If animation exisists.
            // If the animation is not started we start it, or if it has
            // been started, we update it.
            if (this.getAnimation().isAnimationStarted()) {
                // When the animation set's itself to ended we detect it
                // here and end it properly.
                if (this.getAnimation().isAnimationEnded()) {
                    this.endAnimation();
                } else {
                    this.getAnimation().updateAnimation();
                }
```

```
            } else {
                this.getAnimation().start();
            }
        }
    }

    this.updateSubviews();
}

@Override
public void draw(){
    this.background.draw();

    SceneState.Player state = SceneState.getPlayerState();
    if (this.getAnimation() != null) {
        if (this.getAnimation().isAnimationStarted()) {
            state = this.getAnimation().getPreviousState();
        }
    }

    switch (state) {
        case GAME_OVER:
            break;
        case WAIT_FOR_OPONENT:
        case SHIP_PLACEMENT:
            this.userPlayer.getShipHolder().draw();
        case OPONENT_TURN: // On opponent turn show the user's fleet.
            this.userPlayer.getFleetGrid().draw();
            for (AbstractShip ship : this.userPlayer.getShips()) {
                if(ship.isShipBeingMoved() &&
                    this.userPlayer.getFleetGrid().hasMouseHover()){
                    Point p =
                        ship.getShipHoverPositionForGrid(this.userPlayer.getFleetGrid());
                    if(this.userPlayer.getFleetGrid().isPositionValidFor(ship,
                        (int)p.getX(), (int)p.getY())){
                        this.userPlayer.getFleetGrid().drawShipHoverBox(ship,
                            (int)p.getX(), (int)p.getY());
                    }
                }
                ship.draw();
            }
            this.userPlayer.getFleetGrid().drawDecals();
            break;

        case USER_TURN: // On users turn show the user's attempts/target grid.
```

```
            this.userPlayer.getTargetGrid().draw();
            break;
    };

    if (this.getAnimation() != null) {
        if (this.getAnimation().isAnimationStarted()) {
            this.getAnimation().drawAnimation();
        }
    }

    this.bannerLabel.draw();
    this.shortMessageLabel.draw();
    this.gameButtons.draw();

    this.drawSubviews();
}

/**
 * This is the event handling method, If any singleplayer component launches
 * an event it is handled here.
 * @param event
 */
@Override
public void handleEvent(IEvent event) {
    GameLogger.print("Singleplayer event \"" + event.getName() + "\" was
        dispached.");

    switch (event.getName()) {
        case "Back":
            SceneState.setGeneralState(SceneState.General.MENU);
            break;
        case "Ready":
            if (this.userPlayer.getFleetGrid().isGridFilled()) {
                SceneState.setPlayerState(SceneState.Player.WAIT_FOR_OPONENT);
                this.gameButtons.removeButtonWithName("Ready");
                this.gameButtons.removeButtonWithName("Back");
                this.gameButtons.add(new Button(dimentions.getRightMargin() -
                    dimentions.getGridCellHeight() * 2,
                    dimentions.getBottomMargin() -
                    dimentions.getGridCellHeight(),
                    dimentions.getGridCellHeight() * 2,
                    dimentions.getGridCellHeight(), "Notready", this));
                this.userPlayer.setReady(true);
                this.bannerLabel.setText("Waiting for opponent");
            }
```

```
            break;
        case "Notready":
            this.gameButtons.removeButtonWithName("Notready");
            this.gameButtons.add(new Button(dimentions.getRightMargin() -
                dimentions.getGridCellHeight() * 2,
                dimentions.getBottomMargin() - dimentions.getGridCellHeight(),
                dimentions.getGridCellHeight() * 2,
                dimentions.getGridCellHeight(), "Back", this));
            this.gameButtons.add(new Button(dimentions.getRightMargin() -
                dimentions.getGridCellHeight() * 5,
                dimentions.getBottomMargin() - dimentions.getGridCellHeight(),
                dimentions.getGridCellHeight() * 2,
                dimentions.getGridCellHeight(), "Ready", this));
            SceneState.setPlayerState(SceneState.Player.SHIP_PLACEMENT);
            this.userPlayer.setReady(false);
            break;
        case "Fire":
            if (this.userPlayer.getTargetGrid().getSelected() != null) {
                int x = (int)
                    this.userPlayer.getTargetGrid().getSelected().getX();
                int y = (int)
                    this.userPlayer.getTargetGrid().getSelected().getY();

                if (this.enemyPlayer.getFleetGrid().isHit(x, y)) {
                    GameLogger.print("Player HIT AI at x=" + x + ", y=" + y);
                    this.shortMessageLabel.setText("Hit!");
                    this.userPlayer.getTargetGrid().registerAttempt(x, y,
                        AbstractGrid.HitMarker.HIT);
                    AudioManager.play(AudioManager.P_HIT);

                    if (this.enemyPlayer.getFleetGrid().isSink(x, y)) {
                        this.shortMessageLabel.setText("You sunk my " +
                            this.enemyPlayer.getFleetGrid().getShipAt(x,
                            y).getName());
                        GameLogger.print("Player sunk AI's " +
                            this.enemyPlayer.getFleetGrid().getShipAt(x,
                            y).getName());
                        AudioManager.stop(AudioManager.P_HIT);
                        AudioManager.play(AudioManager.P_SINK);
                    }
                } else {
                    this.shortMessageLabel.setText("Miss");
                    this.userPlayer.getTargetGrid().registerAttempt(x, y,
                        AbstractGrid.HitMarker.MISS);
                    AudioManager.play(AudioManager.P_MISS);
```

```
                }

                this.setAnimation(new PauseAnimation(this,
                        SceneState.Player.OPONENT_TURN,
                        SceneState.getPlayerState(), (Time.SECOND * 2)));
                SceneState.setPlayerState(SceneState.Player.ANIMATING);
                this.getAnimation().start();

                this.userPlayer.getTargetGrid().resetSelected();
            }
            break;
        case "missileAnim":
            int x = (int)
                this.enemyPlayer.getTargetGrid().getSelected().getX();
            int y = (int)
                this.enemyPlayer.getTargetGrid().getSelected().getY();
            if (this.userPlayer.getFleetGrid().isHit(x, y)) {
                GameLogger.print("AI HIT player at x=" + x + ", y=" + y);

                // Tell the AI that it hit!
                this.enemyPlayer.hitAShip(this.enemyPlayer.getTargetGrid().getSelected());

                // Update the view's model.
                this.enemyPlayer.getTargetGrid().registerAttempt(x, y,
                    AbstractGrid.HitMarker.HIT);
                this.userPlayer.getFleetGrid().registerAttempt(x, y,
                    AbstractGrid.HitMarker.HIT);
                AudioManager.play(AudioManager.E_HIT);
                if(this.userPlayer.getFleetGrid().isSink(x, y)){
                    this.enemyPlayer.setSunkAShip(this.userPlayer.getFleetGrid().getShipAt(x,
                        y).getSize());
                    this.shortMessageLabel.setText("Ai sunk your " +
                        this.userPlayer.getFleetGrid().getShipAt(x,
                        y).getName());
                    AudioManager.stop(AudioManager.E_HIT);
                    AudioManager.play(AudioManager.E_SINK);
                    GameLogger.print("AI sunk player's " +
                        this.userPlayer.getFleetGrid().getShipAt(x,
                        y).getName());
                }
            } else {
                this.enemyPlayer.getTargetGrid().registerAttempt(x, y,
                    AbstractGrid.HitMarker.MISS);
                this.userPlayer.getFleetGrid().registerAttempt(x, y,
                    AbstractGrid.HitMarker.MISS);
```

```
                AudioManager.play(AudioManager.E_MISS);
            }
            this.enemyPlayer.getTargetGrid().resetSelected();
            break;
        case "gameOver":
            SceneState.setPlayerState(SceneState.Player.GAME_OVER);
            this.shortMessageLabel.setVisible(true);
            if (this.hasPlayerLost(this.userPlayer)) {
                this.shortMessageLabel.setText("");
                if (this.getActiveSubview() == null) {
                    GameLogger.print("Adding view");
                    GameOver child = new GameOver(this, true, "You lose.");
                    this.addChildView(child);
                    this.setActiveSubview(child);
                    AudioManager.stop(AudioManager.E_SINK);
                    AudioManager.play(AudioManager.GAME_OVER_LOOSE);
                }
            } else if (this.hasPlayerLost(this.enemyPlayer)) {
                if (this.getActiveSubview() == null) {
                    this.shortMessageLabel.setText("");
                    GameOver child = new GameOver(this, true, "You win!");
                    this.addChildView(child);
                    this.setActiveSubview(child);
                    AudioManager.stop(AudioManager.P_SINK);
                    AudioManager.play(AudioManager.GAME_OVER_WIN);
                }
            } else {
                // This should never happen.
                this.shortMessageLabel.setText("RulesDrawError");
            }
            break;
        case "depause":
            this.getAnimation().end();
            if(this.getAnimation().getResumeState()==SceneState.Player.OPONENT_TURN){
                this.bannerLabel.setText("Opponent's turn");
                this.shortMessageLabel.setText("");
            }
            else
                if(this.getAnimation().getResumeState()==SceneState.Player.USER_TURN){
                this.bannerLabel.setText("Your turn");
                this.shortMessageLabel.setVisible(true);
            }

            break;
```

```
            case "ambienceLooper":
                AudioManager.play(AudioManager.AMBIENCE);
                TimeEventList.add(new TimeEvent(this, 30 * Time.SECOND,
                    "ambienceLooper"));
                break;
        }
    }
}
```

Appended to this document is the Design document which is referenced multiple times in the report above.

Battleships Video Game Design

by Louis Bennette - 200801270

1 Summary

Battleships is a two-player table-top board game. Both players have a hidden hand consisting of two grids, the first being the placement of their battleships and the other being a grid of the attempts they have made to attack their opponent. The game requires that the players take turns at making guesses as to where the opponent's ships are. If the guess is a hit, a miss or a ship sinking, the opponent must announce it. More specifically for this project I will be using the official Hasbro rule set [4]. I will be transferring all the concepts and rules over into a playable interactive video game.

1.1 Aims and Objectives

I intend to reproduce the Battleships board game as a video game which can be played offline and online.

1. Singleplayer is the offline mode where the player will play against an AI opponent.
2. Multiplayer is the online mode where player will be paired with another player opponent by connecting to a game hosting server.

The game will have images rendered as graphics to represent all of the graphical user interface elements.

1.2 Changes to the original specification

The original specification was a board view of what would be featured in the game. There are few new decisions that need to be added as well as some decisions that have changed.

The confirmed language that will be used is Java as it has very good server socket control [5] which will be very useful for by the client server model that will be implemented. Java is multi-platform which will make the game more accessible to a wider range of users. The game is expected to run on individual's home and personal computers. There a fragmentation issue due to the inevitability of different screen sizes. This is a problem because the game will run full screen and utilise all of the screen. As a simple solution I have chosen to only deal with 16:9 and 4:3 aspect ratios. If a user does not have a screen with the required aspect ratio the game will be windowed and will run in scaled 16:9 mode.

There are also a few features that I will implement into the game if I have enough time. All of the core features have not been changed of had anything added to them. The following are optional features only:

- I will make the program available on apple OSX computers. The libraries that are being used make this feature feasible as I only need to package the correct operating system native files.

  This therefore changes the deliverables at the end of the project by adding an OSX version of the client.

- I will implement multi-language options into the game, I will do so using dictionaries in both the client and the server. The client dictionaries will hold language specific information for components such as button names, button text, label text, ship names. These are all that singe-player mode requires. Multi-player uses all of these but will use the connection to the server to retrieve multi-player specific text.

- I will implement an in-game chat room for multi-player games so the user can talk with their opponent.

The game will run without an internet connection. The user will be able to fully play the game and use the single-player and options features. However, if the user is not connected to the internet and attempts to use the multi-player, the scene will load but there will be a message stating that the user needs to be connected to the internet. An internet connection is required for the multi-player game.

The libraries I had chosen in the specification will still be used. LWJGL [1] will be my graphics to screen manipulation using openGL. Slick2D [7] will be used for sounds and texture loading and binding.

1.3 Research that has been conducted

For one week after my specification was handed in I was conducting test programs.

The first of these programs was testing a small client-server based system to understand how the server socket programming in java works. For the socket programming practice I used Jones's BSD Sockets programming from a multi-language perspective [5] The system was running on two machines on a local area network and was successfully making a connection, sending and receiving data and closing the connection safely. This first project taught me the risks involved with server connections and also a better from of practice to use for the listeners. The listeners will have their own thread and will call back to the respective class with the message that is required. More on this in the Multiplayer, Server Login and Matchmaking section 2.4 on page 10. The risks here are that catching brute loss of connections is difficult and can quickly cause a chain effect that crashes the client and/or the server.

The second test I conducted was a simple texture and text renderer. My program would load up textures and rectangles with textures bound to them. This test also had a small bit of

text with a personalised font that printed to screen the current mouse's x and y coordinates. This was a crucial test as these were all the basic requirements that my program needed. Based on this test I can now create buttons with text and have them be click-able. I can also draw a ship to screen and have drag and drop features using these simple principles. I learnt that I should use an update-and-draw system where at every iteration of the game loop I update the elements I need to update and draw the elements I need to draw to screen. This means every loop represents one frame in the game. More on this in the Scene Views and Components section 2.2 on page 4.

2 Design

2.1 Overview

The game will run on a simple update-and-draw loop principle. The GameLoop class will be the main controller for the game. In this class there will be a loop designed to loop indefinitely until the game close is requested. The game loop creates and controls the appropriate scene views and the scene views them selves create and control their respective Components. The game loop calls the *draw*() and *update*() functions. The SceneViews call the *draw*() and *update*() on their Components. Finally the Components draw to screen and update their location and state. The game loop makes use of the GameState class which holds different enumerations for different states. Specifically the game loop makes use of the General state to know what scene view it needs to call update and draw on. These simple building blocks are the basic structure of the game. The implementation will be using object oriented design patterns such as singleton classes and factory classes [6].

There are a few factory classes that are used for system wide access to their functions. By system wide I mean that any class in the client has access to their methods and fields (through getters and setters). The system wide classes are:

- OpenGLManager: used to initialise openGL and to draw rectangles to screen
- TextureManager: used to return a texture given a string.
- FontManager: singleton class used to provide the game fonts to components
- ResolutionManager: singleton class used to provide margins and pre-calculated positions for game objects dependent on the aspect ratio
- Window: used to control the window and update the window for every frame.
- Input: Keeps track of all the keyboard and mouse inputs.
- Time: Keeps tract of the time every frame lasts.

For my Input and Time I will be taken most of the code from Benny Bogoganoosh's Input.java [2] for the Input class and his Time.java [3] for the Time class. This is because these classes do exactly what I needed them to, as the project is a large body of work using these simple classes will aid me greatly.

2.2 Scene Views and Components

The scene views are used to hold components. The game will specifically consist of four scene views. Singleplayer, Multiplayer, Options and Menu. Components are the items that are graphically represented and manipulated by update calls these are for instance; Buttons, Ships or Grids.

Scene Views:

The scene views are responsible for initialising their components. When the game loop updates and draws the view, the view must in turn update and draw all of the relevant components. The views make use of the GameState class to know information about the Player state or the General state of the system. The Singleplayer and Multiplayer classes inherit from the AbstractGame class, this provides the user the ability to get game options such as return to menu (Quit) or to change the settings (Options).

• *Menu*: The menu is the first view that the user will see as they load the game. The menu will be the view from which the user can navigate to other views. Menu will hold a button registry and an *onClick(Button b)* function to handle click events from the buttons belonging to menu. These buttons' particular funtionality is to allow the user to navigate to other scene views. There will be a single player button, Multiplayer button, Options button and an Exit button.

• *Singleplayer*: This view will inherit from the abstract game class. The view will hold the players as one user player and as one AI computer opponent. The view will also have a button registry with the same functionality as in the Menu. The player will hold it's own components such as ships and grids so the player classes also need the update-and-draw functions. The view will start by letting the player place their ships anyway they like,. When the user is ready the view will randomly choose a player to start first, then the game will start. When the game is won by someone the scene view will inform the players who has won and provide buttons as option for what to do next.

• *Multiplayer*: This view is very similar to the Singleplayer view. This view has a button registry as well, it also has a user player. The opponent is an online player. The user will initially see a text field where they will be prompted to enter a username. This will be sent to the server to establish a connection. Then the user will be randomly allocated an opponent for them to battle with. The rest of the game from that point will be the same as the single player game.

• *Options*: This view is where the user will choose configurations for the game such as mute options, or resolution options. It's worth noting that much of the options view's

functionality is of lower priority and is therefore treated as optional feature sets.

Components:

The components are the interface the game has with the user. The components provide a graphical representation on screen for the user. They also provide interactions to the user by, for example, allowing the user to drag and drop ships or click coordinates on a map or buttons, etc..

• *Buttons*: This component simply provides the user the ability to click the button and have that button correspond to a particular slice of code that the game will execute. The buttons require an IClickable as a parameter. An IClickable is typically a scene view. It holds the ability to deal with button events. When button in a clickable view is pressed the the *onClick(Button b)* function of that view is called and based on the parameter *b*'s name the correct bit of code is executed.

• *Ships*: The ships are components that belong to the player class. The ships are initialised when the player class is initialised. The ships can be dragged and dropped onto the grid during the placement phase and can be rotated. The ships are actually made up of five classes, one for each ship, Carrier, Battleship, Cruiser, Submarine and Destroyer. This is because in one game we will always have two instances of each ship. The ships are also constant in their sizes.

• *Grids*: The grids are a representation of the battle field but the can also act as a container for the location of the ships. The grids are used to determine what cells have been hit and where those hits missed or succeeded.

• *Labels*: This is a container for printing text out onto an area of the screen. Labels can not be updated they are used by the game as a type of status bar. A label can be marked as a title so that when it is drawn, the font used will be the title font.

• *Text Areas*: This is container that allows the user to type in a string of text which allows editing and keyboard commands. The Text area will have an action assigned to it. For instance, the chat text area in a multiplayer game will perform a broadcast action with the string written by the user and empty the current text field's contents. The text area will have a isBeingEdited field. This field is flag that allows the update method to append direct input from the user to the text area.

2.3 AI Opponent

The AI will be used to play against the player in single player mode. The role of the AI is to provide a challenge to the user by using a heuristic. To tackle this problem I have devised a solution that would make the game still be possible to win but also allow the AI a fighting chance to beat the user.

The AI will hold its own 2 grids, they will be used the same way as the user's, one for the AI's ships and one for the attempts on the opponent. Perhaps the AI will also require a sleep timer because the response of such an AI will be very fast and could make the user feel un-immersed.

The heuristic will be simplistic and straight forward, it consists of two states, search and kill. Meaning simply that the AI will search semi-randomly across the board and when it hits a ship it will attack until it has destroyed the ship.

More specifically when an enemy ship has been hit we will add the points of the hit into a list. Regardless of the state the AI is in, every time it makes a successful hit, it will add that hit's coordinates to the list. When a ship has been sunk the respective points of that ship are removed from the list. This implies that any points in the list belong to a ship that has not been sunk. Only when the list is empty the AI can resume the search state. As well as updating the list we add every attempt the AI has made to the targetGrid array. This is to avoid the AI choosing the same point twice.

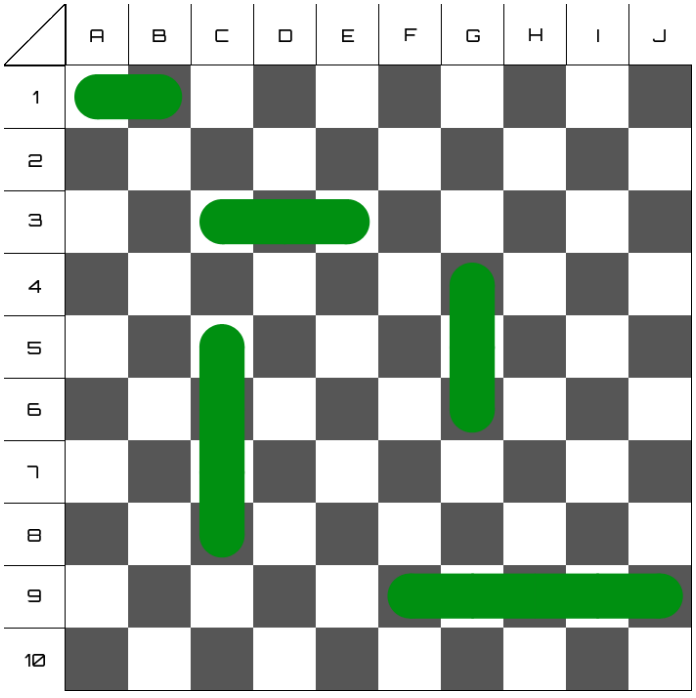


Figure 1: Every ship has to exist on a black and a white cell of the grid regardless of placement and orientation.

Search State:

The semi-random behaviour was a behaviour I came up with over the summer, the idea was reinforced when I read an article expressing the same behaviour. That is, if we assume the game grid is like a chess board with checkered black and white tiles (see figure 1). The AI only needs to search in the black squares to find all of the ships. This is because all the ships are 2 or more squares in length meaning they have to appear in both black and white regions in any placement. Using this will shorten the maximum time it takes to explore the whole board by half. (Providing we stay in the search state) Another thing that the search state will do is avoid redundant cells. In figure 2 we can see that If the above behaviour chose the ? cell it would be a waste of a turn as this cell could never possibly have a ship in it. The algorithm would ignore this cell and make an attempt on another cell.

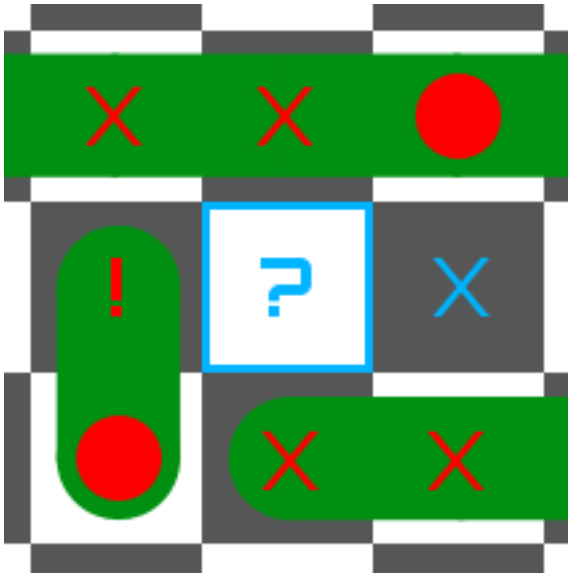


Figure 2: • = Origin, × = Hit, ! = Sinking hit, × = Miss. Example of a redundant cell where the cell '?' is the cell in question.

Kill State:

The kill state is an original creation based on tackling the possibility that two ships are adjacent and making sure the AI can account for all the identified hits before switching back to search mode. There are 3 sub states: Identify, Horizontal Attack, Vertical Attack. It does so by holding and analysing a list of all the hits it has made. When an enemy ship has been hit we will add the points of the hit into a list. The first point of that list will be the "origin" of the kill state. Next turn the ai will randomly chose a point adjacent to the origin to fire.

Identify is the initial internal step of the kill state. This step requires the AI to 'identify' the orientation of the ship it is hunting. The heuristic here is to assume that if we have a point in the list then there will be at least one adjacent point to it. The AI therefore starts by randomly firing at any of the adjacent points in the grid. If there is a miss we stay in

the identify state. If there is a successful hit in the cell above and below the origin we enter vertical mode. If the hit is to the left or right cell we enter horizontal mode.

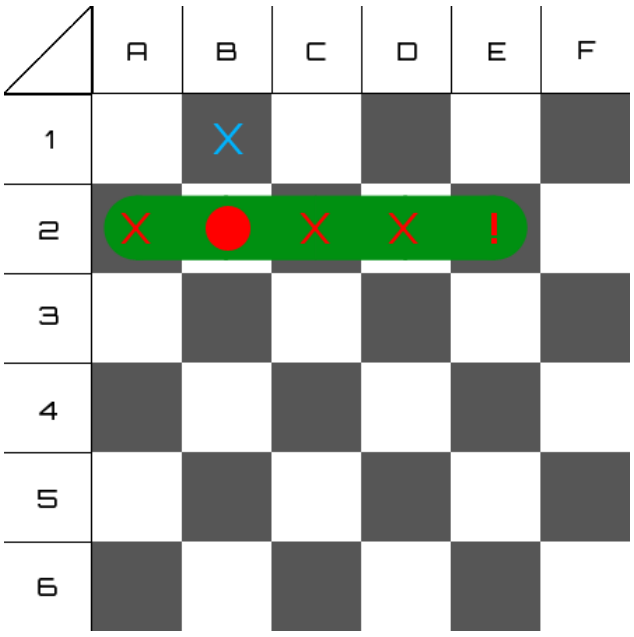


Figure 3: • = Origin, × = Hit, ! = Sinking hit, × = Miss.

In figure 3 we see that the AI started in it's identify state with origin $B2$, the AI made an attempt at $B1$ but it missed and remained in the identify state only to attack $A2$ and then move to the horizontal mode.

Horizontal and vertical mode are essentially the same mode except for their respective orientations. For now we will discuss these modes by using horizontal mode in the examples. It's simply worth noting that vertical mode will work the same way.

If we look at figure 3 The AI successfully entered horizontal mode when it hit $A2$. On the next turn the AI will attempt to continue in the **direction of it's last successful hit** but if an attempt has already been made in that cell or if the cell is out of bounds then we remain in horizontal mode but change direction. This is what is happening in figure 3 after the hit in $A2$ the AI changes direction and hits $C2$ and so on.

$list_4$	$E2$
$list_3$	$D2$
$list_2$	$C2$
$list_1$	$A2$
$list_0$	$B2$

Figure 4: Representation of the list that would be generated by the action made in figure 3

In figure 4 we can see that the items are added to list in order that they were hit and not in their correct order on the grid, this makes removing a ship a little bit more tricky. However due to the steps we take there is an inevitable situation that will always arise. That is, that the sinking blow will always be on the extremity of the ship.

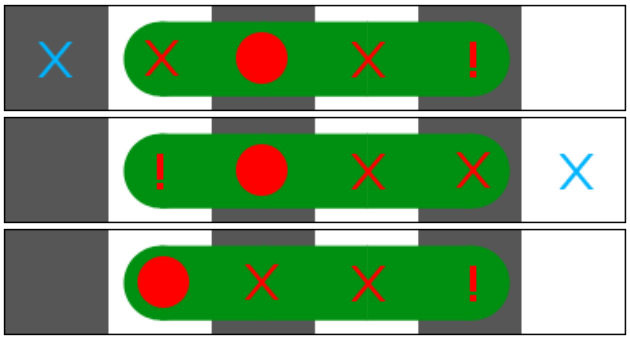


Figure 5: • = Origin, × = Hit, ! = Sinking hit, × = Miss. This is the iteration of the kill algorithm over the three possible states for a ship of size 4.

Figure 5 is a representation of the possible outcomes the algorithm would have on a ship of size 4. Although in figure 5 we only show for a ship of size 4 this rule applies to a ship of any length, providing the length is a positive integer. The situations in figure 5 are unique. Although they can be mirrored, in essence these are the only situations that can occur.

We can prove that these four situations are the only unique situations that can arise for a ship of size 4. To do this I will prove this to be true for a ship of size n. My definition here of a unique situation is; an outcome where none of the outputted situations are mirrors of each other. Firstly for a ship of size n where n is even and $n > 1$: We take

$$uniqueStarts = n \div 2$$

where $uniqueStarts$ represents the number of possible cells that the origin can start without those cells being mirror images of each other. Then we take

$$possibleFollowActions = (uniqueStarts - 1) \times 2 + 1$$

where $possibleFollowActions$ represents the number of possible cells that will be hit after the origin has been hit and those cells are cells that belong to the ship. This is important because if the next cell after the origin is a miss then we are back in the identify mode. But here we ignore all the searching and we only want information about the ship being hit. $(uniqueStarts - 1)$ here represents all the cells that are not on the extremities of the ship. The cells on the extremities only have one possible successful hit and that's moving towards the centre of the ship, this is why we add one.

Secondly for a ship of size n where n is odd and $n > 1$: We take

$$uniqueStarts = (n - 1) \div 2 + 1$$

we can assume that the central point of the ship is unique as it has no mirror images. So we always ensure that the central point is a possible unique start point for the algorithm.

Once the AI has hit it's 'secondary' cells, it will iterate the algorithm for every turn and move in the same direction then finish. Or it will miss, turn around, move in the other direction then finish. Regardless of encountering other ships the AI once it misses will go to the origin change directions and reach the end of a ship.

In other words the *possibleFollowActions* represents the number of unique situations that can arise from the searching and identify states once in the kill states. For any ship of size n if we iterate the algorithm for any start point on the ship, the sinking hit or last hit will always be on an extremity of the ship.

Because we know that a sinking hit ship will always be on an extremity when a sink hit is announced. The AI will also receive the length of the ship. We also know the direction in which we were travelling along the ship just before sinking it. We can work back from that point, the AI will take the coordinates of the sinking hit then using the reverse direction, the orientation and the size of the ship the AI will make a small new list with all the points corresponding to the ship that has just been sunk.The AI will quickly check to see if all the points in the new list exist in the hit list. If they do the AI will remove all those points from the hit list as all the hits are now accounted for. If the origin was one of the points that was removed then we make the first point on the list the new origin and return to the identify mode. If the hit list is empty then we simply return to the search state. Finally, it's worth noting that if the identify mode can't attack an adjacent square then we change the origin to the next point on the hit list and so on. If everything is done well there will always be a point on the hit list that is eligible to become the new origin.

2.4 Multiplayer, Server Login and Matchmaking

During a multiplayer game the player will be playing against an opponent online via a server which is relaying messages to and from each client. The server here will be designed to minimise the amount of memory it uses. To do this the server will just act as a relay, sending messages to and from the clients.

The multiplayer game will be sending the messages to the server, there will also be a message listener attached to the muliplayer game. This simply listens to any message that will be sent by the server. When a message is received it's placed in a list. The multiplayer game will check to see if there are any messages in the list. If there are it will send the message to a parsing function. Once parsed the appropriate output will execute the following correct piece of code.

The client will need to distinguish between the commands. I will be using a code based system. The code is inserted internally and can not be done manually by the user.

- '0' will be a string message to be displayed to the player's text area. Here the '0' will be followed by any message, for example "0Hello World!" The '0' is removed from the string and the rest is printed.

- '1' will be a client command. The command will be a 3 letter string which will refer to a particular piece of code to be executed. Anything after the 3 letters will be used as parameters if needed. For example "1try1,5" will be taken in by the opponent as a try (fire) command where "1,5" are the coordinates. The opponent will respond with either miss, hit or sink. So "1ansm", ans (answer) command with m (miss) as a variable.

- '2' will be server commands for example "2dco" to disconnect the user. Or "2req" to re queue the user for another game. These commands are not relayed to the users they are internally managed by the server.

When the user starts a multiplayer game, the game will in turn initialise an user-player and a online-player. The online-player will be a representation of the opponent's state but will not hold any data about the grid. The user will be expected to position their ships and say when they are ready. When the user is ready their personal state is updated. The game will also send a message to the server which will relay the message to the other connected client. The other client will then update the state of it's online-player. This structure is how all the messages are transferred to each other.

The online user will hold very little information about the opponent. The online user will also hold simple information about the opponent however it will not hold information about the opponent's ships or their position on the grid. It will however hold data about the opponent's state and about what ships are still on the field. This is for simple game checking. If we send a try coordinate command and our answer is "1snk1" then we know that the ship '1', which is a Battleship, has been sunk. The game therefore updates it's instance of online-user. When there are no more ships in the online-user instance, we wont need to ask the opponent via the server. In this way both connected clients will be updated simultaneously.

The multiplayer game will also need to have a small inbuilt parser. The parser will take the first character in the string to know what type of message it is. Then depending on the type we will take the next part of the message and perform an action or more checks. This system is so the user will just need to click a location on the grid and the hit commands will be automated without explicit communication between the users themselves.

The Server:

The server will be multi-threaded, this is because we need several instances of a game to run at once. We also need the server to keep adding new users and listeners for incoming messages. The Server Listener will be listening to any new connections and adding them to a login thread which will handle their username and add them to the online users.

Online users is the list of currently connected users. The user class will hold data on the user's socket and their name. The user also has a listener associated with it. This listener is waiting for messages from the client that it is associated to. When a message is recieved the string will be added to a list. The game instance will check the lists and parse any message for both users in every loop. If there are 2 or more users in the Online Users list then we

take the first two of the list and add them to a game. When the game instance is started up it is expected to receive two users. At this point the message listeners for each user are started up and the game can begin.

It's also worth noting that the server will be run from the command line and therefore will not have any graphical interface. The server will be logging all important activity occurring in the system this is to help debug in case of server crashes.

2.5 Client GUI

The following are simplistic views. The views are not finished images that the user will see. They are spacial representations of where components will be. The colours represent different components: Blue = Grid, Green = TextArea, Red = Button, Yellow = Ship, Purple = Label.

Figure 6: Menu: The black border represents the space where a graphic could be used for the title screen. The title of the game will run along the top on the label.

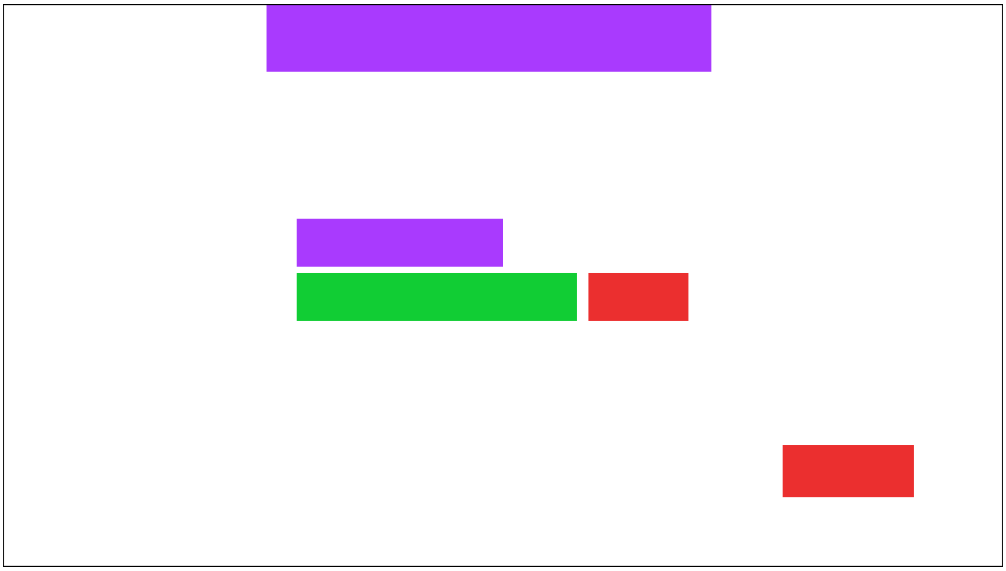


Figure 7: Multiplayer: This is what the user will see when they are asked to enter their username. The back button is in the bottom right.

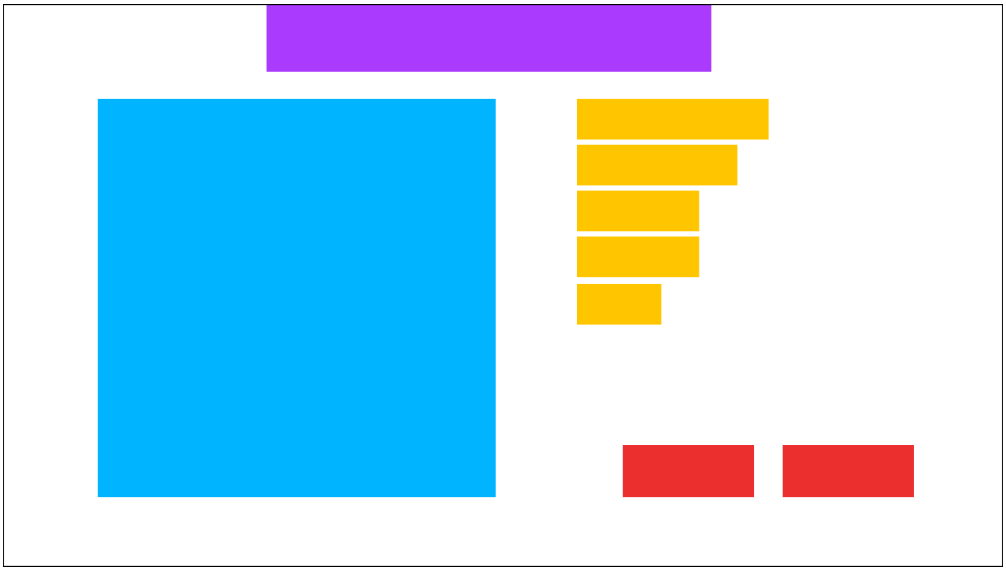


Figure 8: Multiplayer and Singleplayer: This is what players will see when they are expected to position their ships onto the grid. The buttons are the back button and the ready button.

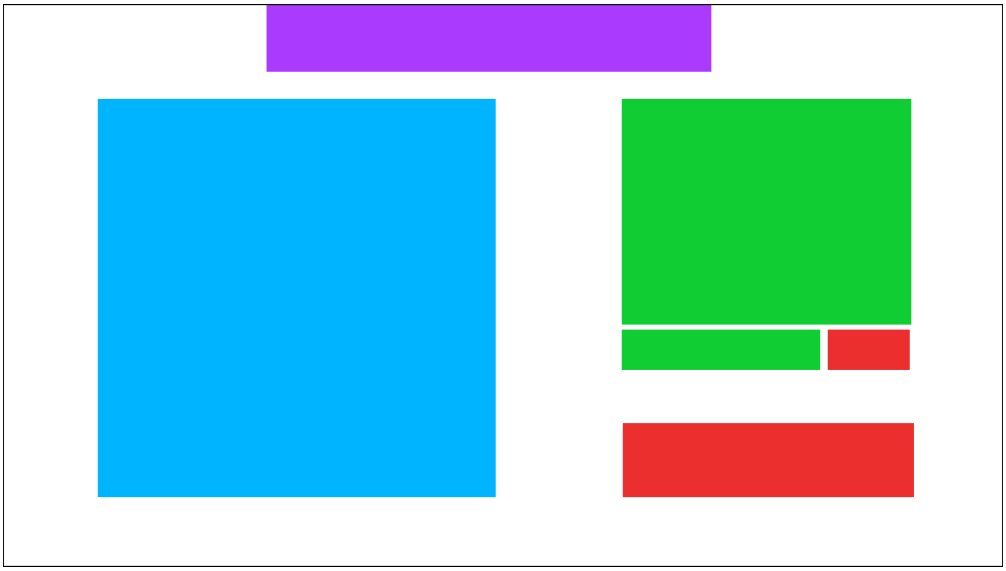
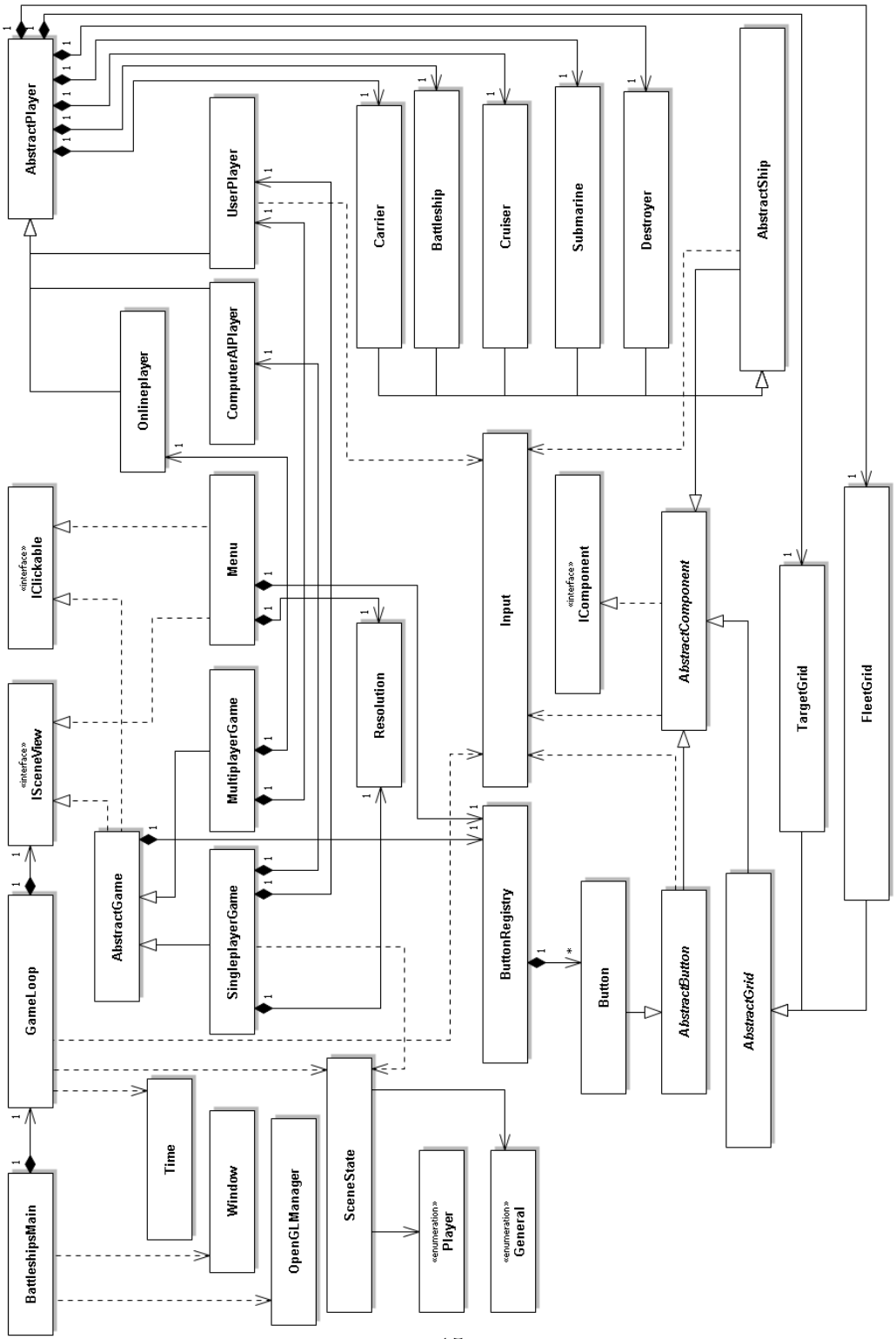


Figure 9: Multiplayer and Singleplayer:

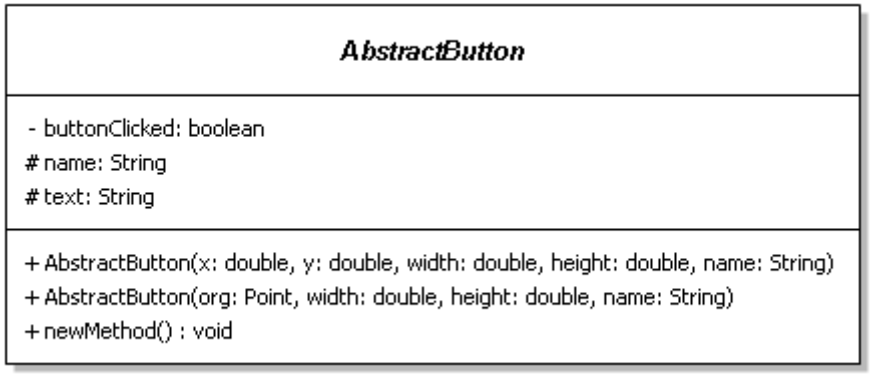
2.6 Client Structure

Figure 10: Battleships client class diagram.

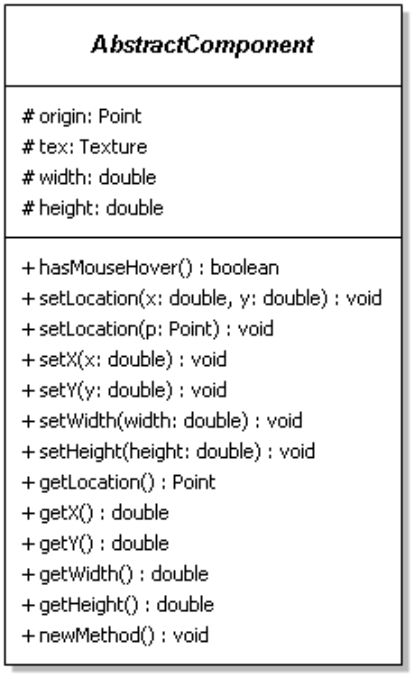


The following is the detailed version of the classes used in the class diagram represented above in figure 10

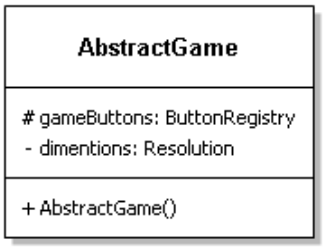
AbstractButton



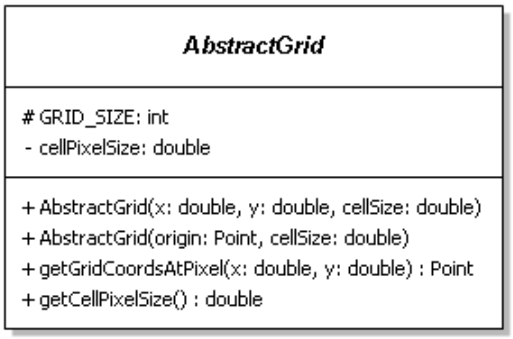
AbstractComponent



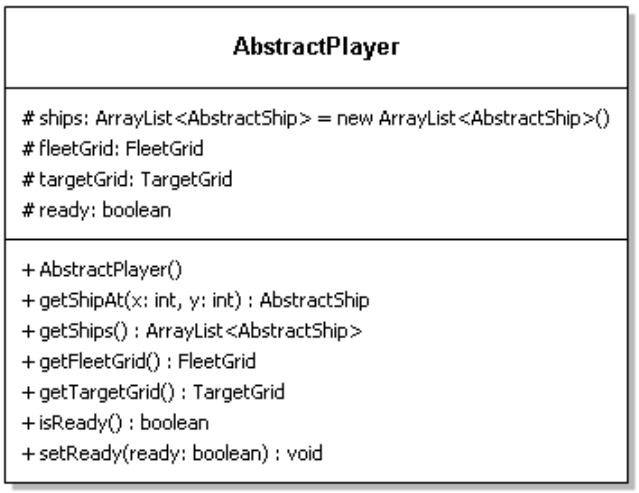
AbstractGame



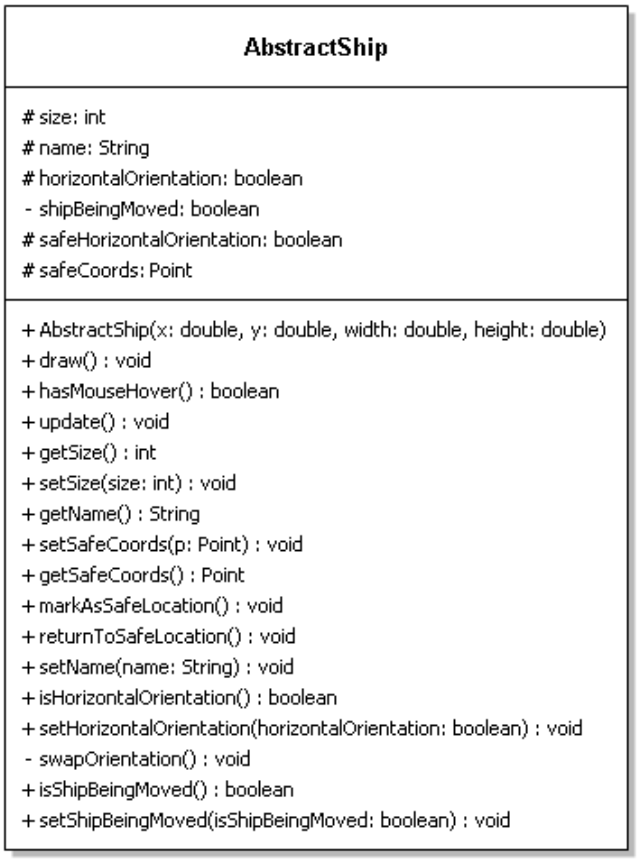
AbstractGrid



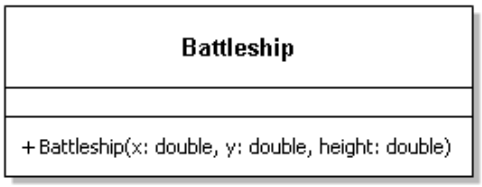
AbstractPlayer



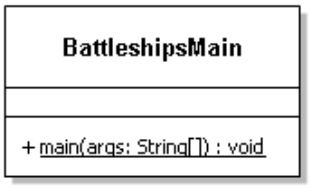
AbstractShip



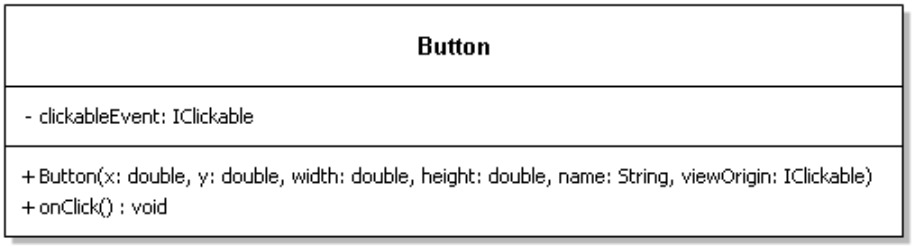
Battleship



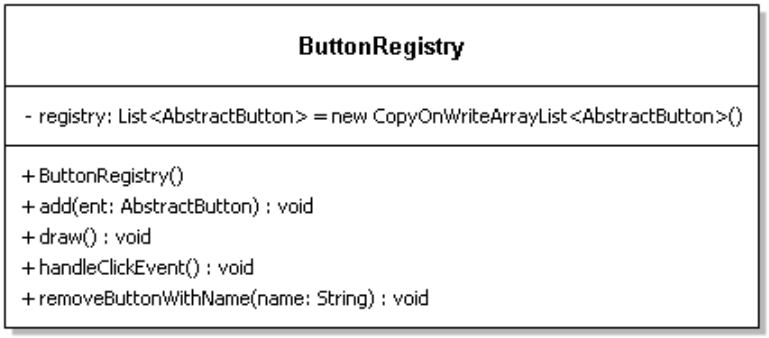
BattleshipsMain



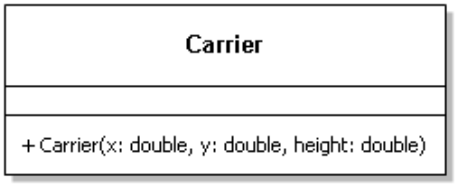
Button



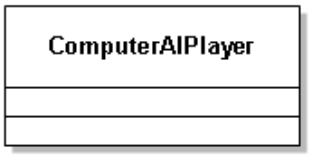
ButtonRegistry



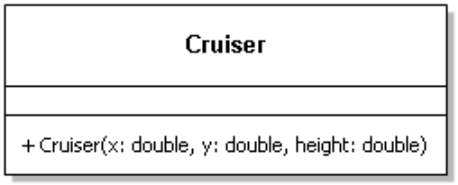
Carrier



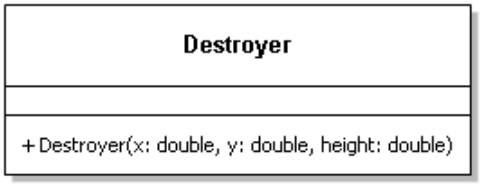
ComputerAIPlayer



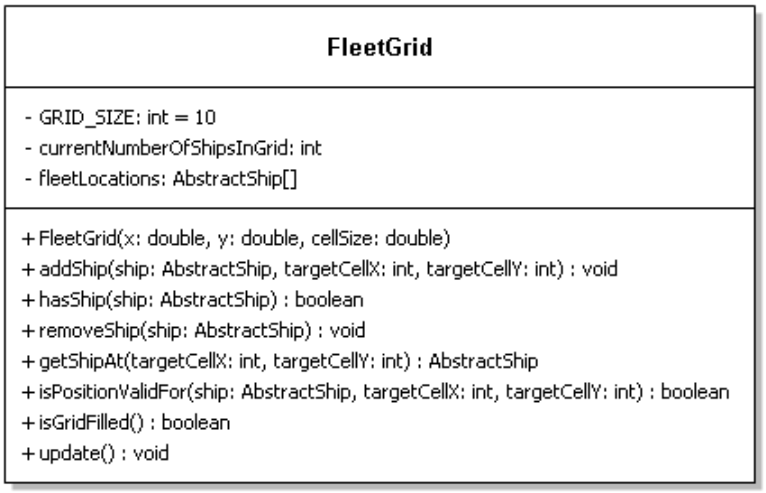
Cruiser



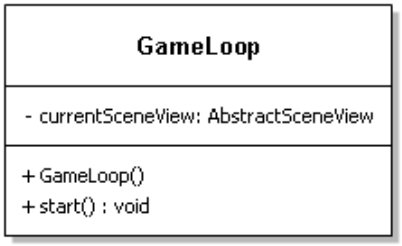
Destroyer



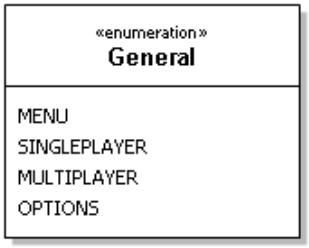
FleetGrid



GameLoop

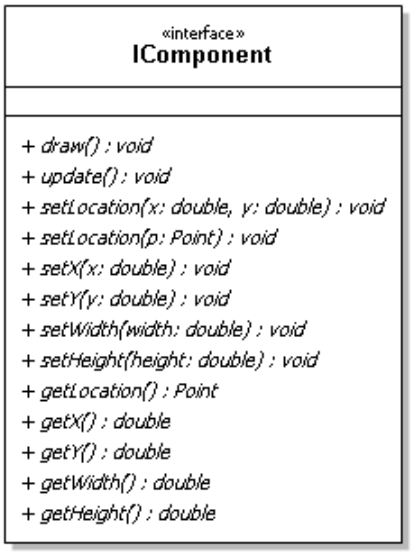


General

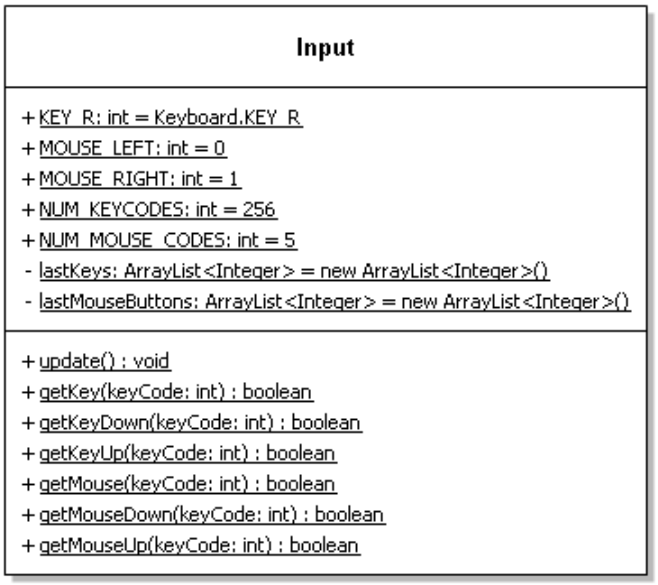


IClickable

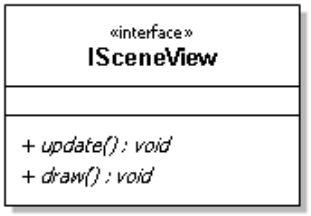
IComponent



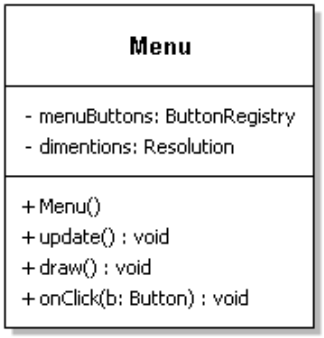
Input



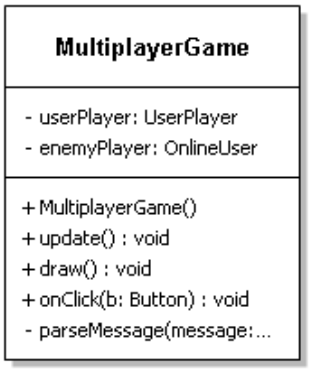
ISceneView



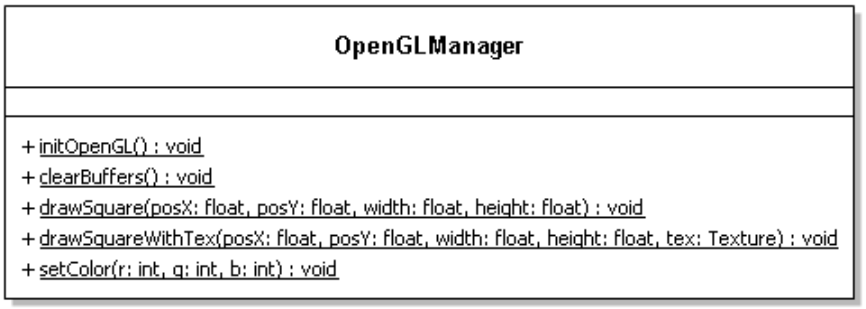
Menu



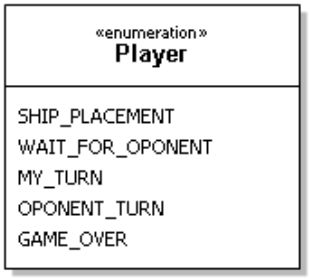
MultiplayerGame



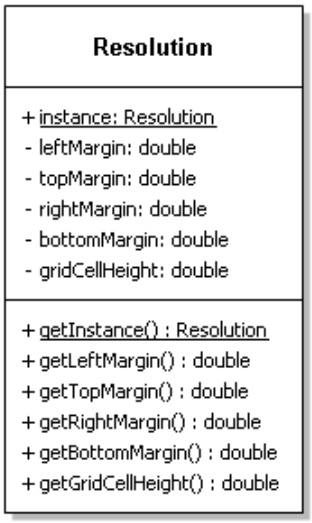
OpenGLManager



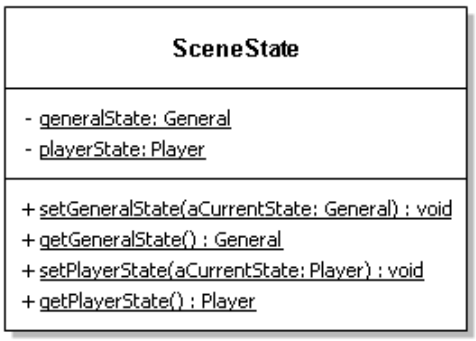
Player



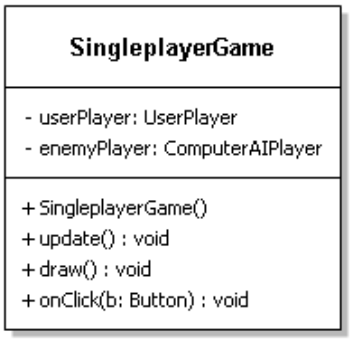
Resolution



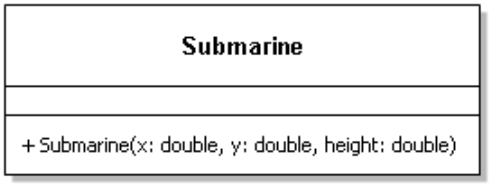
SceneState



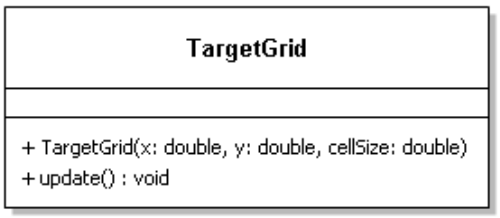
SingleplayerGame



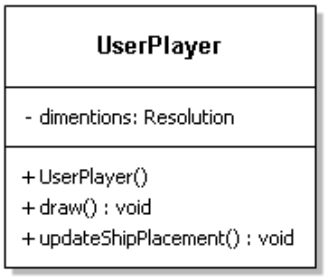
Submarine



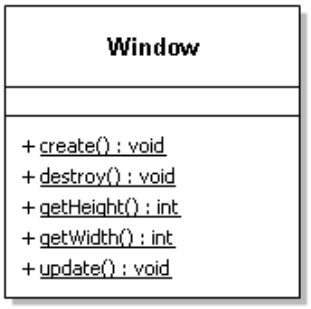
TargetGrid



UserPlayer

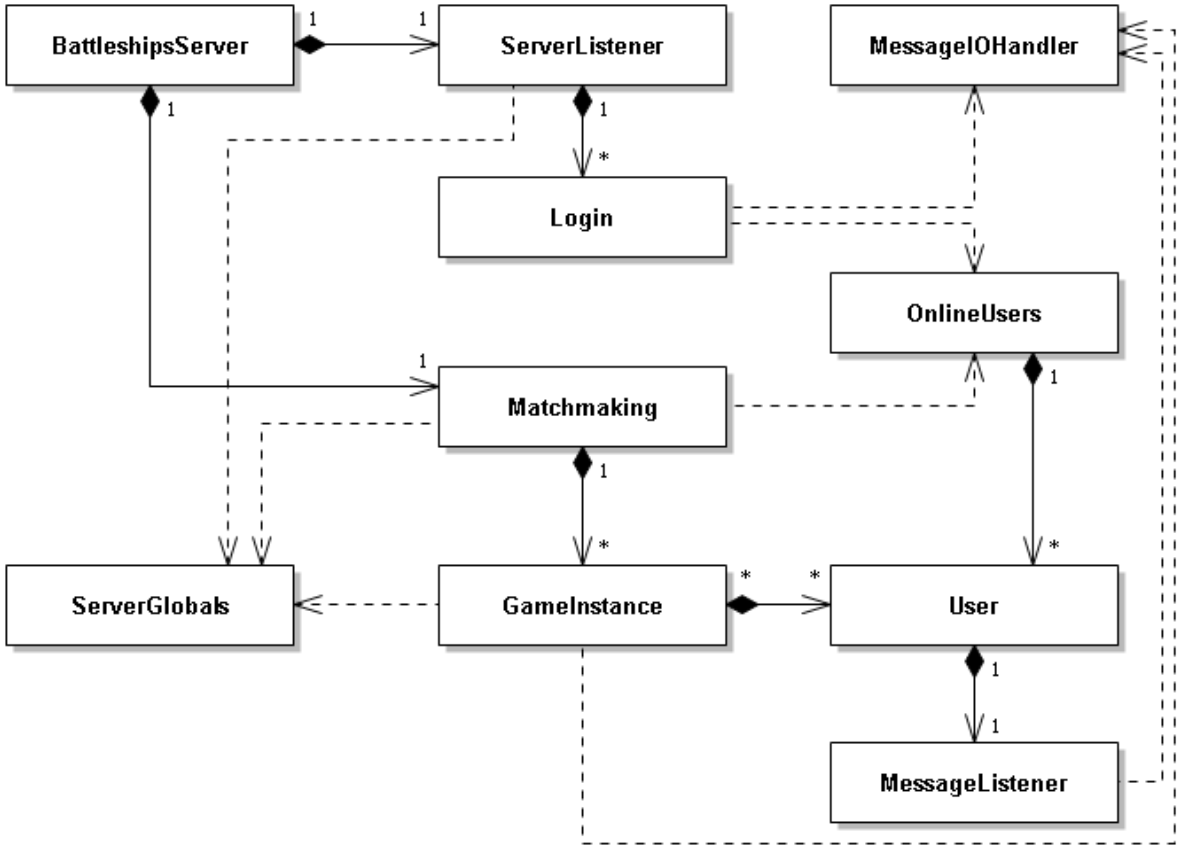


Window



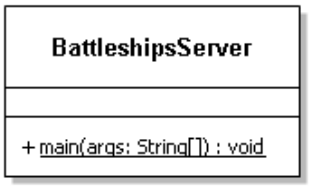
2.7 Server Structure

Figure 11: Battleships server class diagram.

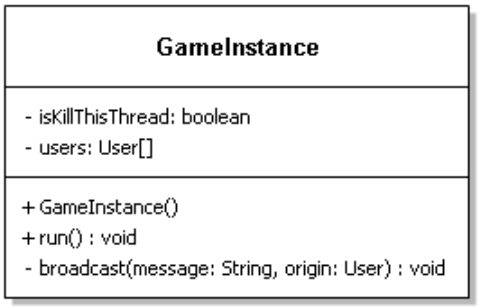


The following is the detailed version of the classes used in the class diagram represented above in figure 11. Note that the class diagram does not tell what methods are synchronized or not. The public static methods in MessageIOHandler and in OnlineUsers are actually public static synchronized methods.

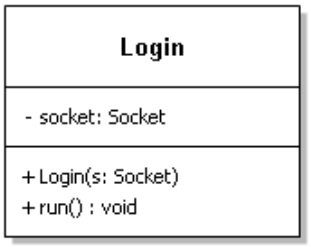
BattleshipsServer



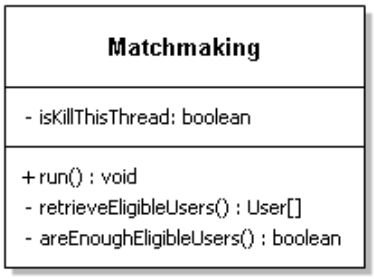
GameInstance



Login

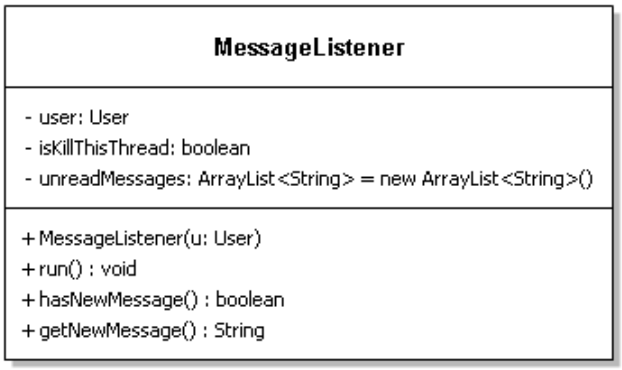


Matchmaking

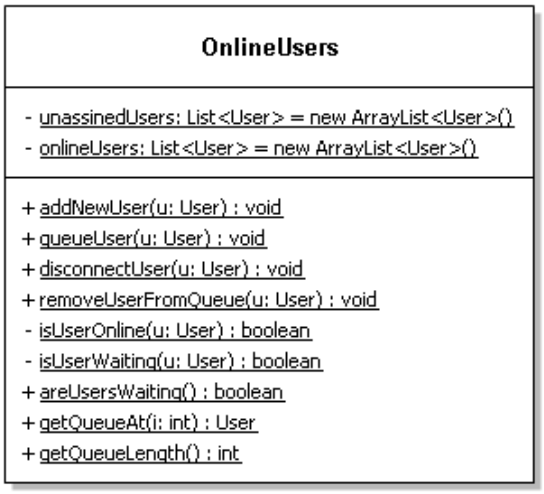


MessageIOHandler

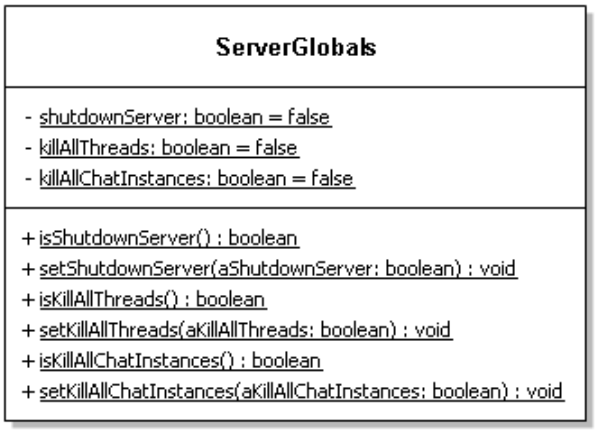
MessageListener



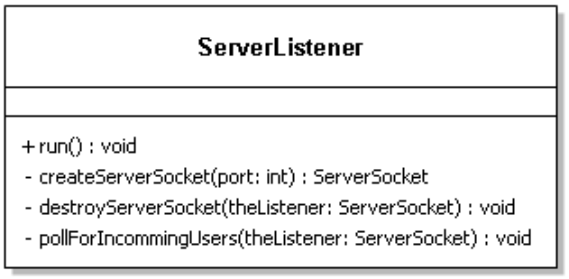
OnlineUsers



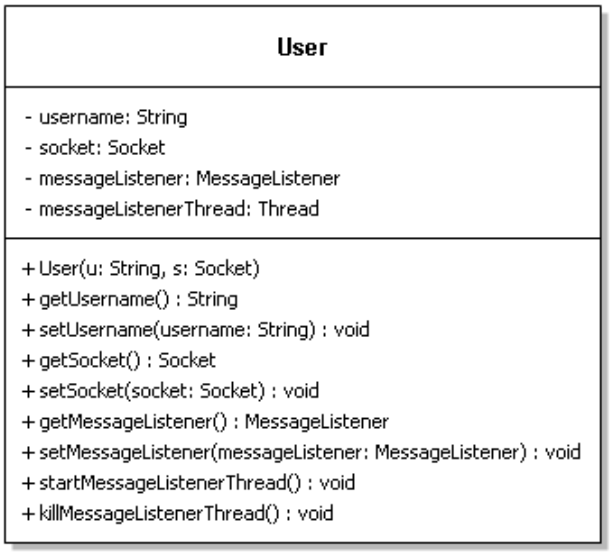
ServerGlobals



ServerListener



User



3 Review

The gantt chart specified durring the specification phase has been altered. So far, in a general perspective I have achieved what I was expecting to this far along the project. There have been a few things that have been done differently during the design phase. I decided that my project design did not need as many diagrams to express how it would work. I also spent more time in my research phase as I ended up coding a substantial amount during this testing. Some of the code I tested will be reused in the implementation itself especially the work done on the server socket tests.

The following summarises what has been delivered and what is expected to be delivered in the future.

Delivered:

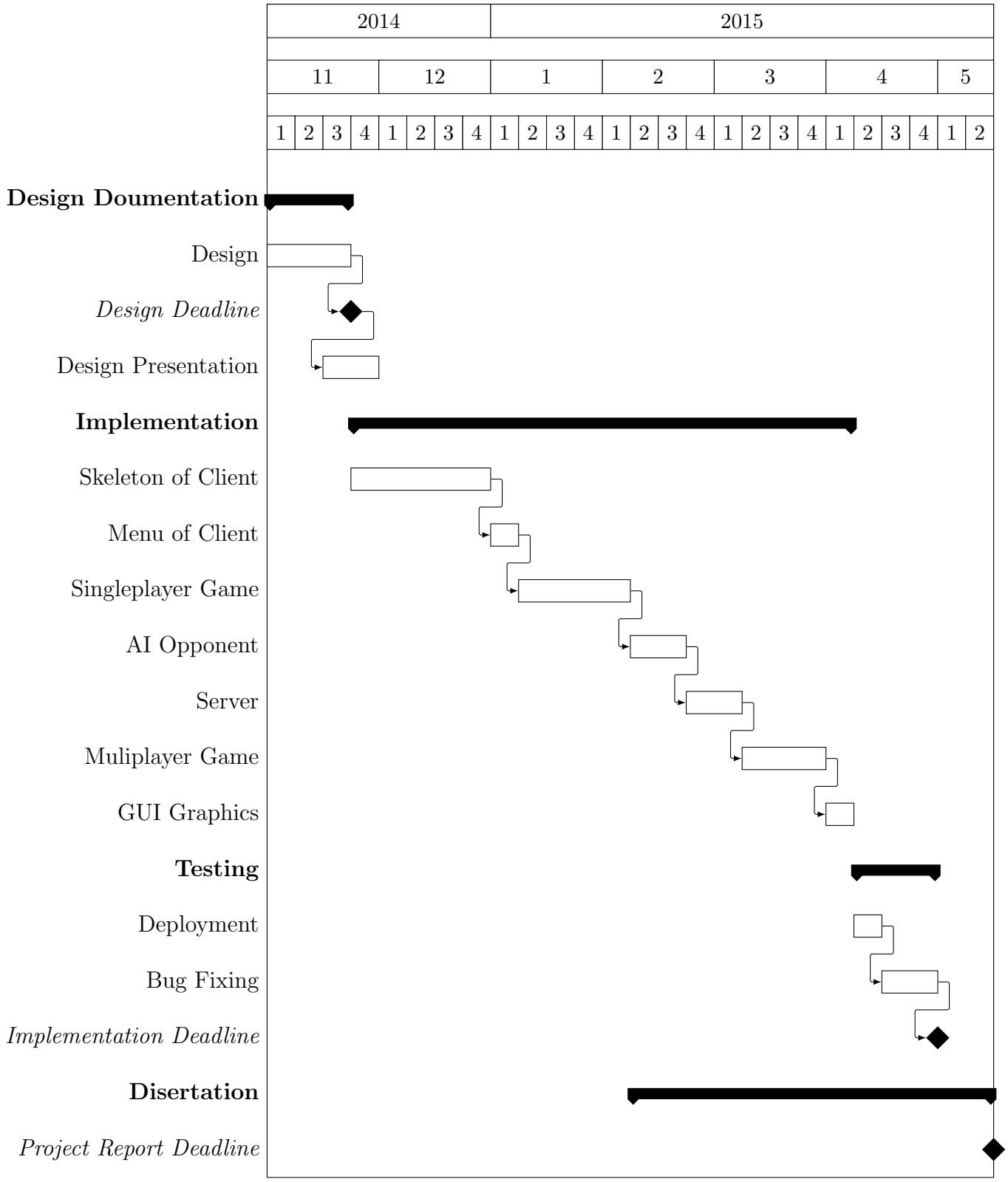
- Specification document
- Design document
- Design presentation slides

To be delivered:

- Interim progress report
- A working game client
- A working game server
- Implementation presentation slides
- Project Report document

In order to evaluate the software I must re-iterate what the aims of the project are and what will be delivered at the end of the project. I plan to have a client to project the game to the user via a clear and easy to use interface. The other part of my system is a server which will be expected to be running and serving the clients for several days on end. So the evaluation of the project will be; how well do users respond to navigating and playing the game? How clear are the images? Is the game of high standard and enjoyable? Is the game easy to play but still engaging? For the server my evaluation will be; how robust is the software for the client and the server? How responsive and fast are the messages dealt with? How easy is it for the user to connect to the server? The actual evaluation of the game will be done when the implementation of the software is completed and has been deployed and tested.

The following is a revised version of the gantt chart presented in the specification document. This gantt chart is in less detail as the project itself is agile and will require reworking.


2014
2015
11
12
1
2
3
4
5
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
Design Doumentation
Design
Design Deadline
Design Presentation
Implementation
Skeleton of Client
Menu of Client
Singleplayer Game
AI Opponent
Server
Muliplayer Game
GUI Graphics
Testing
Deployment
Bug Fixing
Implementation Deadline
Disertation
Project Report Deadline

References

[1] Lwjgl, 2013.

[2] Benny Bobaganoosh. 3d game engine input.java, 2014.

[3] Benny Bobaganoosh. 3d game engine time.java, 2014.

[4] Milton Bradley Co. Batteship for 2 players, 1990.

[5] M. Tim Jones. *BSD Sockets programming from a multi-language perspective [electronic book] / M. Tim Jones.* Online access with subscription: Ebrary. Hingham, Mass. : Charles River Media, c2004., 2004.

[6] Partha Kuchana. *Software architecture design patterns in Java [electronic book] / Partha Kuchana.* Online access with purchase: Taylor & Francis. Boca Raton, Fla. : Auerbach Publications, 2004., 2004.

[7] New Dawn Software. Slick2d java documentation, 1999.