Table of Contents

INTRODUCTION

The assignment task was to design and create a multiplayer game client application by modifying client code provided by the university along with the code for the server, which can be adjusted to satisfy the design needs of this project.

In particular to:

- design and create a multiplayer game client application by modifying C++ client application that connects to the provided Java server.
- implement core features (a bat for players 1 and 2, a ball, sound and players' scores) which would all combine to allow the game to run smoothly and with a high-level integrity.
- include a specific feature adjusting the players' bat speed temporarily after the implementation of the core game features. This feature should enable the bat to reach the specific position quick enough to be able to hit the ball.
- ensure that the communication between the server and client is secure enough that if any external party or hacker tried to intercept the message, they would be unable to read / understand the message. Consequently, creating an approach to handle encryption and decryption for the server and client respectively.

This report summarises information on the steps taken, and provides a critical review of the ultimate outcome.

IMPLEMENTATION

There are two applications that need to be built and developed in order to create the multiplayer game. Firstly, the Server which will initialize the game, constantly check for Client input and provide updates as necessary. Second, the Client Application will interpret the data to/from the Server and will render the game so that the user can play smoothly.

The steps taken to achieve this can best be explained by reference to each of the Server and the Client, respectively:

1. **Server (using Java)**
   There are three critical components that are required to ensure the Server is fulfilling its role:

   a. Player Movement - Player 1 only has access to keys 'W' (to go up) and 'S' (to go down). Player 2 has access to keys 'I' and 'K' to move up and down respectively.

      When the players are connected, each one cannot press the opponent's keys to try to interfere with their bat movement because part of the code has been designed to prevent this.

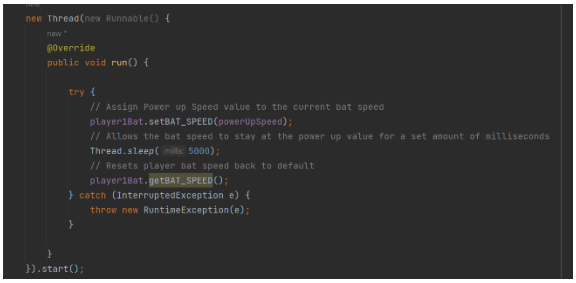
      This has been achieved by asking the Server to check if there is already a connection - if not, the Server will check the local session data of the Client and allocate an ID to the connection. In order to use the session data, the connection needs to be assigned to a variable with a connection data type.

      With this process in place the Server can easily distinguish the players.

b. Power-up Mechanic

Initially, the best way to handle the spawn and de-spawn of the Power-up ball was considered to be to create a new thread to set the bat speed and put the thread to sleep as a sort of delay.

In this way the bat could remain at the new bat speed. Once the delay ends it would revert back to the default bat speed.



After this initial concept, a better design approach was determined.

A *PowerUpComponent* class was created. This would ensure the Power-up ball does not pass beyond the boundary set, nor does the ball does not move too slowly.

In order for the bat speed to increase temporarily the Power-up ball needs to collide with the bat.

Once the Power-up ball collided with the bat, the ball disappeared on the Server but was not visually removed on the Client. To repair this, code was adjusted so that the Server would send a message to the Client to say the Power-up ball has been destroyed, this temporarily removes the image on the Client.

When a collision between the game objects happens, the *runOnce()* function starts and then when the Power-up is destroyed it will run into the background and wait a few seconds before spawning the Power-up ball back into the game.

Within the *runOnce()* function there are two events, one when the Power-up ball collides with the bat, the function *onStart()* commences – this increases the bat speed. Second, when the event ends, the function *onEnd()* is called which resets the speed to the default. In order to check which bat to increase the speed of, the Server has a handler *batPowerUpHandler* – managing the collision between the bat and Power-up.

The last feature of the Power-up Mechanic was to add to *PongFactory* class because the Power-up feature did not exist. It was edited firstly so that the Power-up ball spawns with physics components and secondly to generate the ball in random directions. Random direction (which is important for fair gameplay) was achieved using the function called *random()* This function enabled setting of the linear velocity to a random value which determines the direction of the ball.

c. Encryption – Security is an important feature of a multiplayer game but it was not prioritised as essential given the simplicity of this game design.

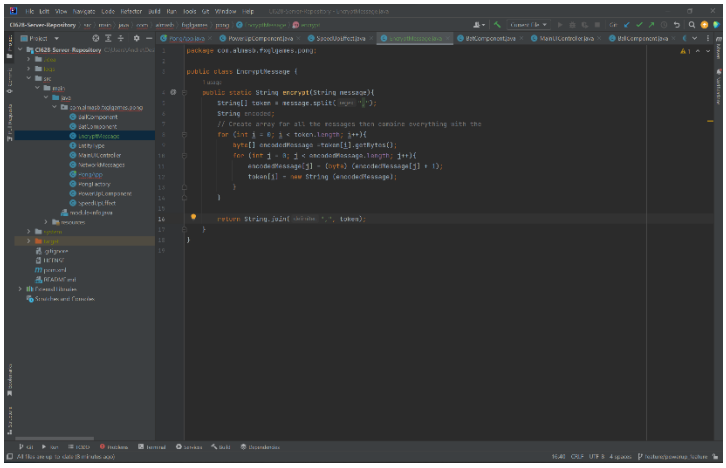
During the design process encryption was developed but there were challenges where messages clashed with each other. As it takes longer to encode and decode the message instead of simply sending the message, the feature was not ultimately implemented. With more time, a solution to the issue of timing between messages and a more secure encryption method would have been included in the final game.

The Server had an Encryption class which takes the message, splitting it when there is a ‘,’ and storing each section in a string array. Each string within the array contains a specific portion of the game data.

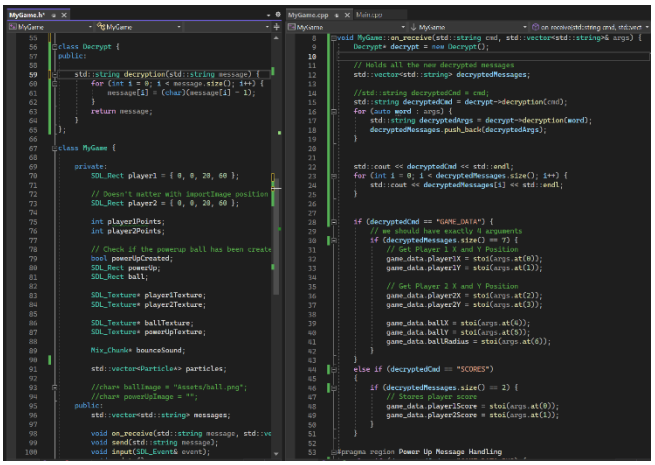
The string array iterates over each message and converts the message into bytes and adds 1 to the encoded byte and reassigns it to another string array.

Within the Client there is a Decryption class which takes the encoded message received and subtracts the encoded byte by one and converts each string into a decoded message.

Below are the screen grabs of the Encryption/Decryption code.



Encryption



Decryption

2. Client Application (using C++)

To interpret the Server and present the game to the user as a playable application, a number of critical steps were required to achieve this, and these are summarised as follows:

a. Multiple Variables – In the design, the Server sends messages with ‘,’ between each value, and the Client should interpret and store into Variables.

For each message, there is a game data string which is interpreted on the Client. The game data string determines what type of data is being received e.g., *GAME_DATA, GAME_DATA_PU, SCORES*. Because it is labelled, each data string is specific as it becomes easily identifiable when handling the messages.

In this implementation there was insufficient time and more specific game data strings would have been better. For example, it would have been preferrable if each aspect of the game data had had their own unique data string, e.g., *GAME_DATA_PLAYER* holding values specific to the player (such as size and position).

One further beneficial improvement could have been that instead of having *if statements* to check if the message sent matched the message received, there could have been a *switch case* which would compare the *enumerable* values to the strings received.

b. Creating the Ball – the ball created on the Client was drawn manually using SDL2 as can be seen in the screen grab.

This was achieved by drawing points around the edge of a circle in increments of (π / 180) and then by drawing lines between the center of the ball and the edge of the circle to create a filled-ball effect.

```
/// <summary>
/// TODO - Input parameter radius so that it can be adjusted when server adjusts radius
/// but for now radius is set to 5
/// There is better approach
/// </summary>
/// <param name="renderer"></param>
/// <param name="centerX">Center of Circle x coordinate</param>
/// <param name="centerY">Center of Circle y coordinate</param>
void MyGame::createCircle(SDL_Renderer* renderer, int centerX, int centerY, int radius) {
    for (float i = 0.0f; i < 2*M_PI; i+= (M_PI/180))
    {
        // find a way to allow radius to be changed by server
        // Moved origin point to be the radius coordinates as the collision point would have been on the top left
        int x = (int)((centerX + radius) + radius * cos(i));
        int y = (int)((centerY + radius) + radius * sin(i));
        SDL_RenderDrawPoint(renderer, x, y);
        // Fills the circle but flickers
        // TODO find better way to Fill circle
        SDL_RenderDrawLine(renderer, x, y, (centerX + radius), (centerY + radius));
    }
}
```

However, drawing the ball on screen is time consuming and often causes glitches, consequently detracting from the performance of the game.

c. Using Images – the Client uses *IMG_INIT()* before loading images and in particular to initialize the correct file type of the image.

Loading an image to the game SDL2 uses a function called *IMG_LOAD()*. The function requires a parameter of a filename to load that image.

To ensure the image is loaded correctly, Error Handling is implemented. This can be achieved by using an *if statement* to check if the file has been loaded correctly and is the correct file type. If it is not, then it will exit the application.

d. Using Sound – sound needs to be initialised using a function called *Mix_OpenAudio()*.

There are several parameters this function requires: frequency to play-back audio in Hz (which is a default frequency of 44100 Hz); audio format; number of channels; and audio buffer size.

In addition, *Mix_LoadWave()* must be loaded with a file name extension of '.wav' sound, which returns *Mix_Chunk* thereby allowing the sound to be played when necessary.

The Server sends predetermined messages to the Client when one object collides with another. Using these messages, *Mix_PlayChannel()* will execute and play the sound using the return value of *Mix_LoadWave().*

e. Using Fonts - to imprint text onto the screen, the application utilizes *TTF_fonts*. As for initialising an image, fonts should be initialised using *TTF_INIT()*.

Having selected the font that matches the game style, the font should be loaded using *TTF_OpenFont().* This requires specific parameters of filename and size of font.

The stored player score received from the Server, the Client converts the integer player score into a string and creates a surface by rendering the text with the colour white.

To avoid being unable to use a string and display the text, used *TTF_RenderText_Solid()* which requires a parameter of a data type of *char*, using *c_str()*.

Using the texture created from the surface, *SDL_QueryTexture()* should be executed with the texture as part of the parameter to retrieve the settings, this information includes the width and height of the texture.

In-order to display the text on the screen it must be copied to the current render session.

f. Particle Effect – the design required a particle effect and therefore a class called *Particles* was created, which contains variables for both ball positions and the timer. In order to access the Particles class, an object was created. This object was iterated using a for loop for a defined number of additional particles.

   For each particle the position of the game object was stored in a Particles vector. Using the vector created, for each particle, the position was randomly set with a maximum and minimum value over a defined period. Each particle of the vector should be rendered by copying the texture to the current session.

   Blending the particles together was required and the *SDL_SetTextureBlendMode()* function was used which further required the *SDL_BLENDMODE_ADD* parameter to achieve the blending.

   Transparency and trail effect was required, but the visual quality needed to be strong. Therefore, a new variable was created in the for loop called *alphaRatio*. This variable was set early in the *for loop* and iterates across each particle. Each particle having an alpha ratio set between 0-1 in order to start to build the trail effect.

   To achieve the desired visual presentation a change in the alpha value was needed using opacity to achieve the effect. The alpha ratio for each particle was multiplied by the maximum 255 to set a new alpha value by using *SDL_SetTextureAlphaMod()* and achieve high quality visual movement.

g. Load Assets - each asset (c~f above) has a specific variable which can be called upon within the render function. Each of these assets are loaded by a higher-level function *loadAssets()* This aspect also improves the flexibility and structure of the code.

h. Order of Rendering – the sequence of the rendering is critically important to achieving a high-quality visual display. It was important to review this aspect carefully.

i. Destroying Assets - once the game concludes, all the assets need to destroyed. Only fonts and images need to be cleaned up, done by the function *TTF_Quit()* to destroy fonts and *IMG_Quit()* to destroy images. This task is an important part of memory management.

j. Disconnecting – once the player(s) wishes to end the game, they will need to press the “esc” key. This action sends a *DISCONNECT* message to the Server and the Server sends back a message *DISCONNECTING* and terminates the connection. Once the client receives the *DISCONNECTING* message, the Client will exit the program and exist the game.

CRITICAL REVIEW

An important aspect of the assignment is to analyse what has been developed, highlighting three of the positive aspects, as well as three areas that could be improved.

Positive Aspects

1. *Powerup Mechanic*
   There are three important features that provide players with a temporary opportunity for a competitive advantage over the other player.

   Firstly, by controlling the speed:

   i). increasing the bat speed to reach the ball quicker and/or
   ii). by hitting the ball at an increased speed using the momentum of the bat.

Second, the ability to respawn the ball in a random direction - this provides each player with a fair chance of hitting the ball.

Third, the Power-up ball may collide with the main ball which then changes the ultimate trajectory and disturbs the game flow for both players.

2. *Implementation of Classes and Organisation of Code*
   The way Comments have been used to describe specific parts of the Code very much supports readability, and clearly identifies the ultimate purpose. Having implemented Data Structures, this provides greater flexibility in task structuring, which in-turn will benefit future program development.

3. *Handling Assets / Storing Values*
   Having created a specific area within the Code to manage assets, this has improved the overall flow. Every aspect of each asset has a specific function which is reusable, and the function returns a value which is stored into a unique variable. This variable is then able to be used to render the asset.

Areas for improvement

1. *Allowing more than 2 players to play the game simultaneously*
   As a single 2-player game, the session performs well. However, when any additional player tries to access the same session, the game will crash because the Server can only handle 2 players in one game, at any one time.

   A solution to this problem would be to improve “error handling” – for example by creating an automatic new session for every 2 players that connect to the Server to play the game.

2. *Improving the communication between the server and client with encryption*
   Even though encryption and decryption were implemented on the Server and Client respectively but it needs further improvement. For example, using a more secure encryption technique to send data to / from the Server and Client. Another example to improve on is the timing for which messages are being sent. Sometimes the message takes a long time to encrypt which causes clashes with timings of sending messages which may send multiple messages at the same time leading to unreadability.

3. *Particle effect of the Powerup and the main ball*
   Even though the positioning of the particles provides a trail, visually, the look of the effect is poor and not as had been anticipated. Different methods to improve the particle effect can be researched later.

   The problem is that *SDL_BLENDMODE_ADD* for *SDL_SetTextureBlendMode()* seems to combine all the RGB colours of the particles together and provides brighter particles.

CONCLUSION

There have been several important learnings from this project, one of the most important has been to deeply understand the architectural structure between a Server and a Client to support the design of a multiplayer game and enhanced the knowledge of how a Server and Client communicate.

There are four other particular learnings to highlight:

a. Transmission Control Protocol (‘TCP’) is more stable and reliable to send game data between the Server and the Client which reduces the likelihood of failure of messages. This is the main reason why TCP is arguably the best alternative for this project as opposed to User Datagram Protocol which is faster but less reliable.

b. Using SDL2 Application Programming Interface to create the Client, identified options to the game development process instead of only using game engines.

c. Encryption is clearly an important part of the Server/Client architecture and researching different methods provided a better understanding to the process of encryption and its importance to security.

d. creating code that is functioning to its purpose, but also able to be adapted for future and possibly longer-term development and not closed.

In summary, this assignment and the steps taken to develop it, have built a solid foundation in understanding the importance of code-design planning. Further that these learnings can also be applied to many different applications.

ESTIMATED GRADE

The grades for the three key elements of the project have been estimated as follows. If more time had been available, further enhancement and development of features would have been possible, and this has influenced these grading estimates.

Game Concept	B
Implementation	A-
Report	A
Average	A-

REFERENCES

- *Lazy Foo' productions* (2014) *Lazy Foo' Productions - Alpha Blending*. Available at: https://lazyfoo.net/tutorials/SDL/13_alpha_blending/index.php (Accessed: January 1, 2023)

- *Clip royalty free library transparent sphere shiny - 2d blue ball PNG, PNG download , Transparent png image - pngitem PNGitem.com*. Available at: https://www.pngitem.com/middle/hwwmJmo_clip-royalty-free-library-transparent-sphere-shiny-2d/ (Accessed: January 1, 2023) (Image License: Personal Use Only)

- Marko2311 *Messenger 3d vector, messenger pink icon 3D ball, Messenger Icons, Pinkicons, 3D icons PNG image for free download*, *Pngtree*. Available at: https://pngtree.com/freepng/messenger-pink-icon-3d-ball_5528609.html (Accessed: January 1, 2023)

- Imaginelabs (2018) *Breakout (brick breaker) tile set - free*, *OpenGameArt.org*. OpenGameArt.org. Available at: https://opengameart.org/content/breakout-brick-breaker-tile-set-free (Accessed: January 1, 2023) (License: CC0)

- *Download Free Game Sound effects Mixkit*. Available at: https://mixkit.co/free-sound-effects/game/ (Accessed: January 1, 2023) (License: Mixkit Sound Effects Free License)

- *Pong: Coderdojo york Andy Callaghan*. Available at: https://yorkdojo.github.io/worksheets/scratch/pong/ (Accessed: January 1, 2023) (License: Creative Commons Attribution-NonCommercial 4.0 International License.)

- Larabie, R. *Stentiga font · 1001 fonts*, *1001 Fonts*. Available at: https://www.1001fonts.com/stentiga-font.html (Accessed: January 1, 2023) (License: custom typodermic-eula-03-2020.pdf within zip-file stentiga.zip .1001Fonts general font usage terms)

- *Build software better, together GitHub*. Available at: https://github.com/logos (Accessed: January 2, 2023).

- *YouTube brand resources and guidelines - how YouTube works YouTube*. YouTube. Available at: https://www.youtube.com/howyoutubeworks/resources/brand-resources/#logos-icons-and-colors (Accessed: January 2, 2023).