

**CSED451 COMPUTER GRAPHICS**

**ASSIGNMENT 1**

2D DRAWING

SUBMITTED BY: ONG WEI HUA (49003139, ongweihua)  
 TAN WEI XUAN (49003140, tanweixuan)

TEAM NAME: SINGAPORE PEOPLE

DEPARTMENT: DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBMITTED TO: PROF LEE SEUNG YONG

SUBMISSION DATE: 15 MARCH 2019

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**2019**

# PROGAMMING ENVIRONMENT

The Programming Languages used are as follow:

* *FreeGlut 3.0.0-2*
* *GLEW 2.1.0*
* *GLM 0.9.9.1*
* *C++*
* *OpenGL & GLSL in Windows*

The Integrated Development used is

* *Microsoft Visual Studio 2007*

We also utilized the Source Control Platform, *GitHub*, and integrated it with Microsoft Visual Studio 2007 to allow us to be able to coordinate work between us and to allow us to easily track changes in our source code during our development process. The Repository can be accessed and view at the following link:

* *https://github.com/jermsinarocket/ComputerGraphics\_Assignment1*

# FUNCTIONALITY OF THE PROGRAM

The program that we implemented is a 2D Volleyball game that is based on the concept of the original “*Pikachu Beach Volleyball*” 2D side-scrolling game.

Our game consists of two characters that are situated on the left/right of the screen with a **net** separating them and determining their respective play zones. One of the characters is a **playable** one, where it can be controlled by the user, while the other one is an **AI** where it will be moved based on the logic and algorithm being implemented by us.

The “*Volleyball*” will be constantly moving and changing directions upon collision with the player/AI, net and game (window) boundaries.

The objective of the game is to reach a predetermined score by letting the ball touched the ground within the opposing side’s play zone.

# DESIGN AND IMPLEMENTATION

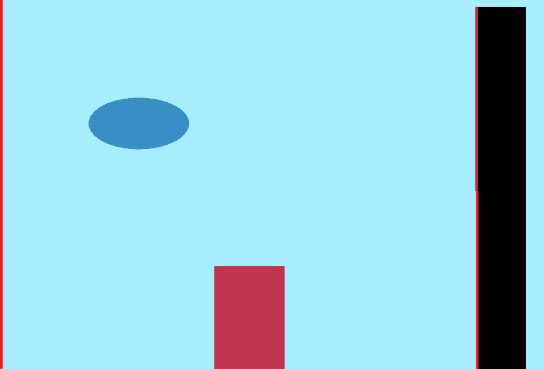
Our program’s source codes are being split into their respective header and implementation files, with the formal comprising of the class definitions and functions and the latter containing the implementation of that class. This helps us to improve the build time of our program and to enable us to link against code without having the source of the definitions.

Our game is initialized with a viewport of *1440* by *800* pixels and is projected and mapped to the clipping coordinates, , with the origin being fixed at the centre of the window. It is set to be in full screen mode to replicate an actual game and to prevent screen resizes from distorting the components within the game *(although it can be fixed by multiplying the projection matrix with the ratio of the resized width and height, the components within the game will be scaled to become either too large or too small)*.

The Net is centralised on the viewport with reference to the defined origin and it serves as a divider for the play zones of the playable character and AI respectively. It is stationary, and its position will not be changed throughout the entire execution of the game.

The playable character will be the one that the user will be able to control, and it will be centred initially at the left play zone with reference to the left game boundary and the left boundary of the net. It can be controlled/moved through the directional keys inputs on the keyboard (left/right), with the left directional key moving it left along the x-axis *(negating its speed and adding that to its [])* and the right directional key moving it right along the x-axis *(Adding its speed to its []),* with the position of the character between constrained within the boundaries *(Figure 1).*

*Left Game Boundary, x = -1*

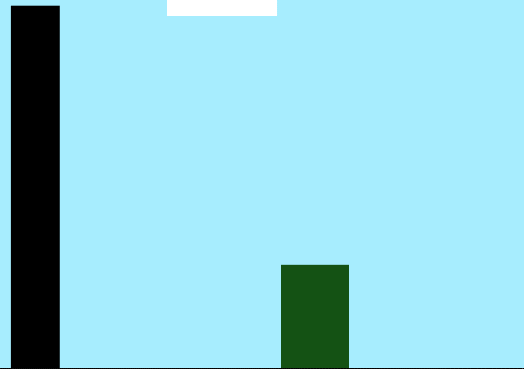


*Left Net Boundary*

Figure 1 Playable Character’s Play Zone Boundary

The AI will be centred initially at the right play zone with reference to the right game boundary and the right boundary of the net. Its movement will be automatic and continuous along the x-axis constrained within the boundaries and the AI’s position will be reversed upon collision with either the right boundary of the net or the right boundary of the game window. Its speed is set to be much lower than that of the playable character as it will be moving on every render, unlike the playable character.

*Right Game Boundary, x = 1*



*Right Net Boundary*

Figure 2 AI’s Play Zone Boundaries

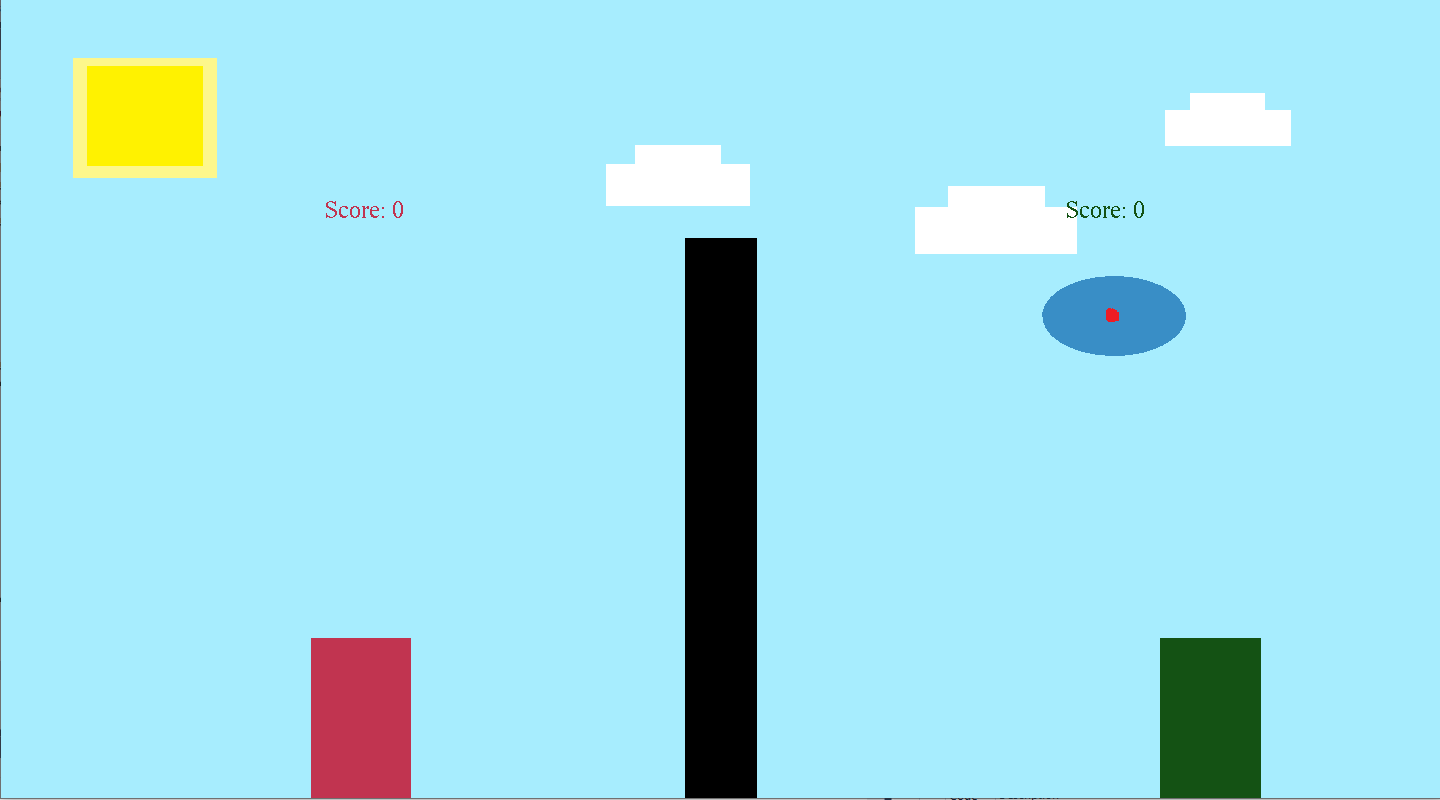
The ball will be centred initially between the top of the net and the top of the game window by referencing its position to the -coordinate of the net and the -coordinate of the game window. The ball will be constantly moving based on its *x-speed (determines its speed along the x-axis)* and *y-speed (determines its speed along the y-axis)* and it is constrained within the boundaries . We must include the radius of the ball in the calculation of the left, right, top and bottom boundaries of the game window as collision is calculated with respect to the centre of the ball *(x-coordinate).* Further constraints are being defined for the ball for when the ball collides with either side of the net or with the player/AI.

*Bottom Game Boundary, y = -1*

Figure 3 Ball Boundaries

*Radius of Ball*

*Top Game Boundary, x = -1*



*Left Game Boundary, x = -1*

*Right Game Boundary, x = 1*

*Centre of Ball, x,y = [0,0]*

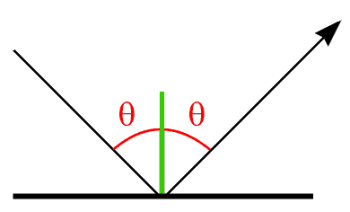
The angle in which the ball bounced on from the stipulated regions of collision must also be considered to make the game realistic. As this is a 2D game and the regions of collision are guaranteed to be either horizontal or vertical, we can simply bounce the ball off them by negating the *x-speed* of the ball when hitting a vertical wall, or the *y-speed* of the ball when hitting a horizontal wall *(Figure 4)*.

Figure 4 Incidence and Reflection

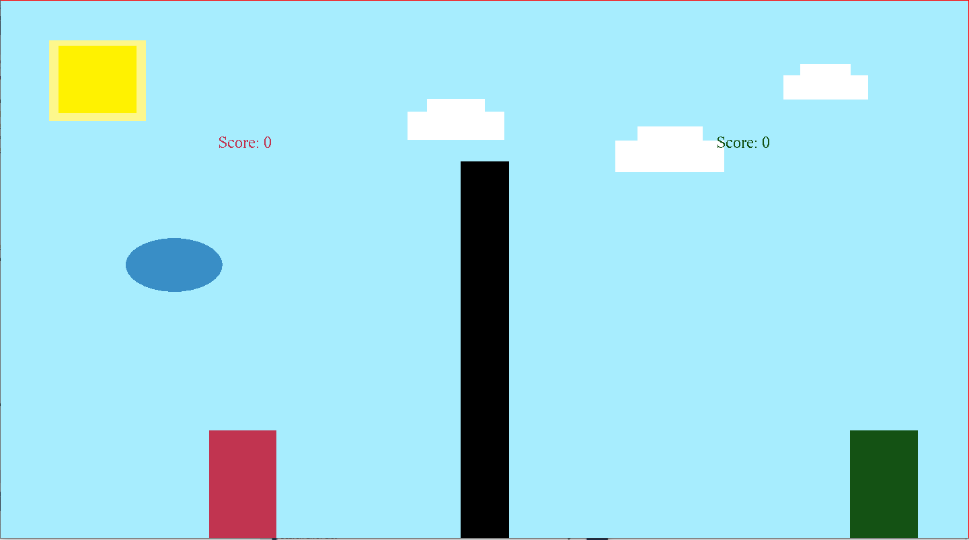
The objective of the game is to not let the ball falls to the bottom of your play zone and to reach a specified amount of points by letting the ball fall on the opposing play zone. To score a point, the ball must land on either bottom sides of the play zone and the point will be attributed to the player in the opposing play zone (E.g. Ball falls at AI’s play zone, Player Scores a point and vice versa) (Figure 5). The constraints to determine who scores a point are as follow:

Figure 5 Scoring Boundaries

*If Ball lands here, AI scores a point*

*If Ball lands here, Player scores a point*

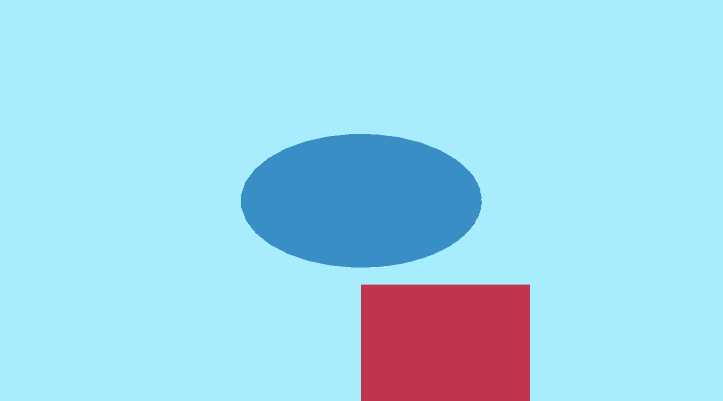
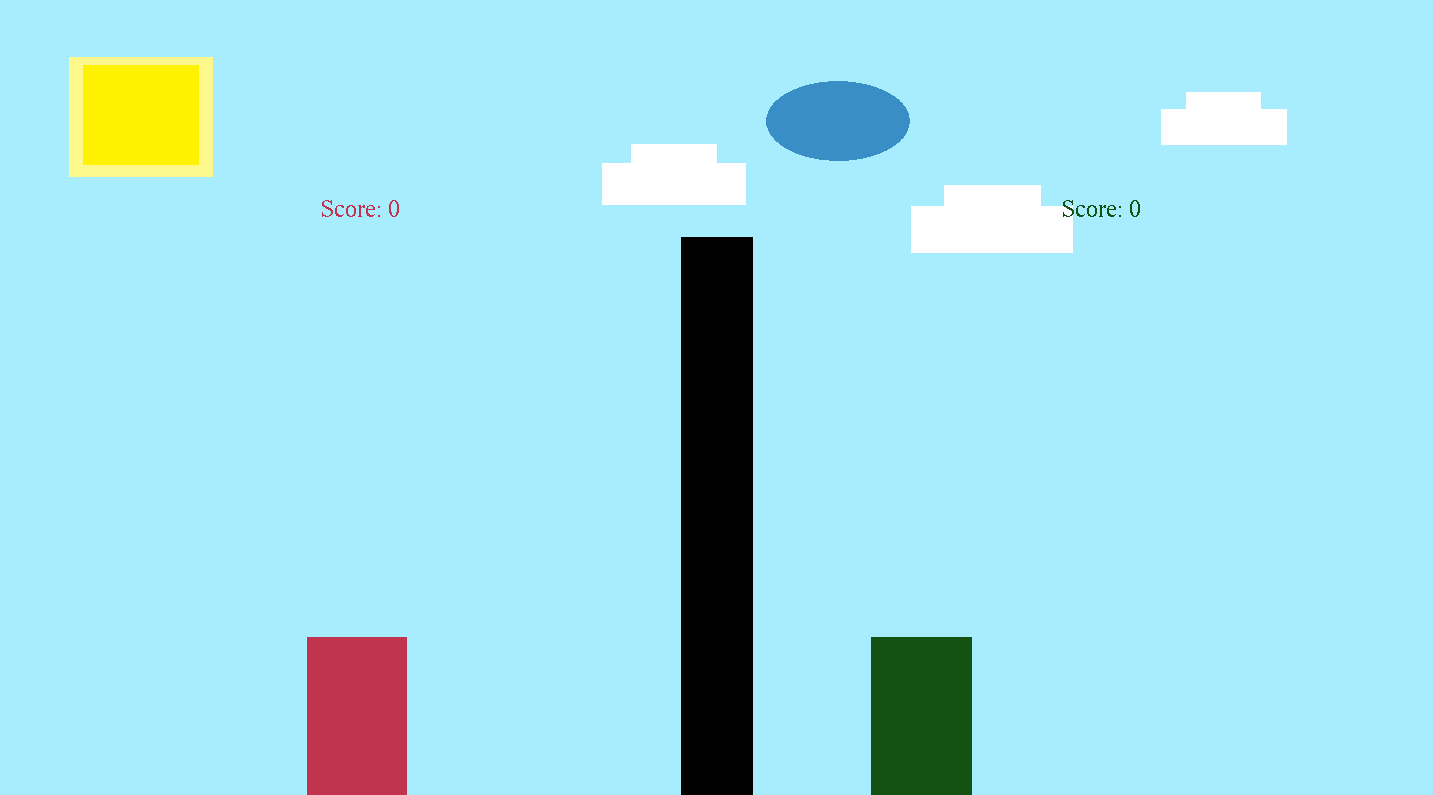
The game also comprises of two screen modes, a fixed screen *(un-zoomed)*  mode where the whole game space can be seen within the game window and there are no screen movements (Figure 6), and a partial *(zoomed)* screen mode (Figure 7) where the window is centred and moving with the ball *(Details will be explained in Section 5).* The two screen modes can be toggled on/off by pressing “z” on the keyboard.

Figure 6 Un-zoomed mode

Figure 7 Zoomed Mode

# BRIEF EXPLANATION OF THE PROGRAM

## Running the Program

There are two ways to run our program, through Microsoft Visual Studio or directly running the executable file:

1. To run the program through Microsoft Visual Studio, launch the ***Assignment\_1.sln*** file located in the ***Assignment\_1* *folder***. After Microsoft Visual Studio has been launched and the solution has been loaded, **build the Solution** *(Ctrl + Shift + B)* and then run the program by pressing **F5**
2. You can launch the program directly by running the executable file ***Assigmnent\_1.exe*** located within the ***bin folder*** *(Assignment\_1\bin)*

## Playing the Game w/ Examples

Upon running the program, you will be greeted with a screen containing a start menu with the instructions for playing the game. Click on the **“Start”** button to start playing the game. (Figure 8).



*Click to Start Game*

Figure 8 Start Screen

Use Directional Keys (Left/Right) to move your Player left/right (Figure 9).

*Directional Keys to move Player left/right*

Figure 9 Game Screen (Controlling Player)

Prevent the Ball from falling to the bottom of the player’s play zone and moving the player to the falling ball’s position (Figure 10).

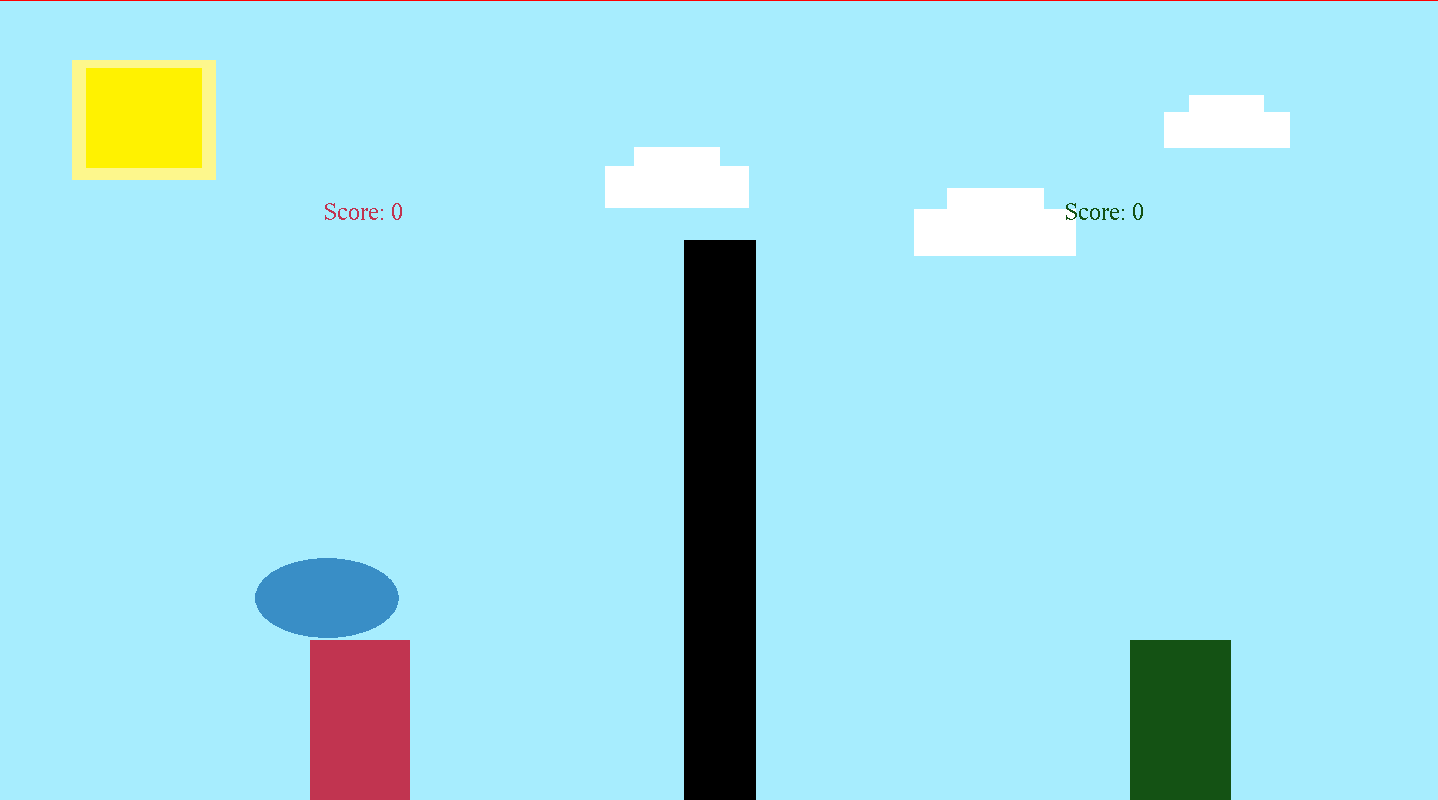


Figure 10 Player Catching the Ball

Score a point (Figure 12) if the opposing side fails to catch the ball (Figure 11).

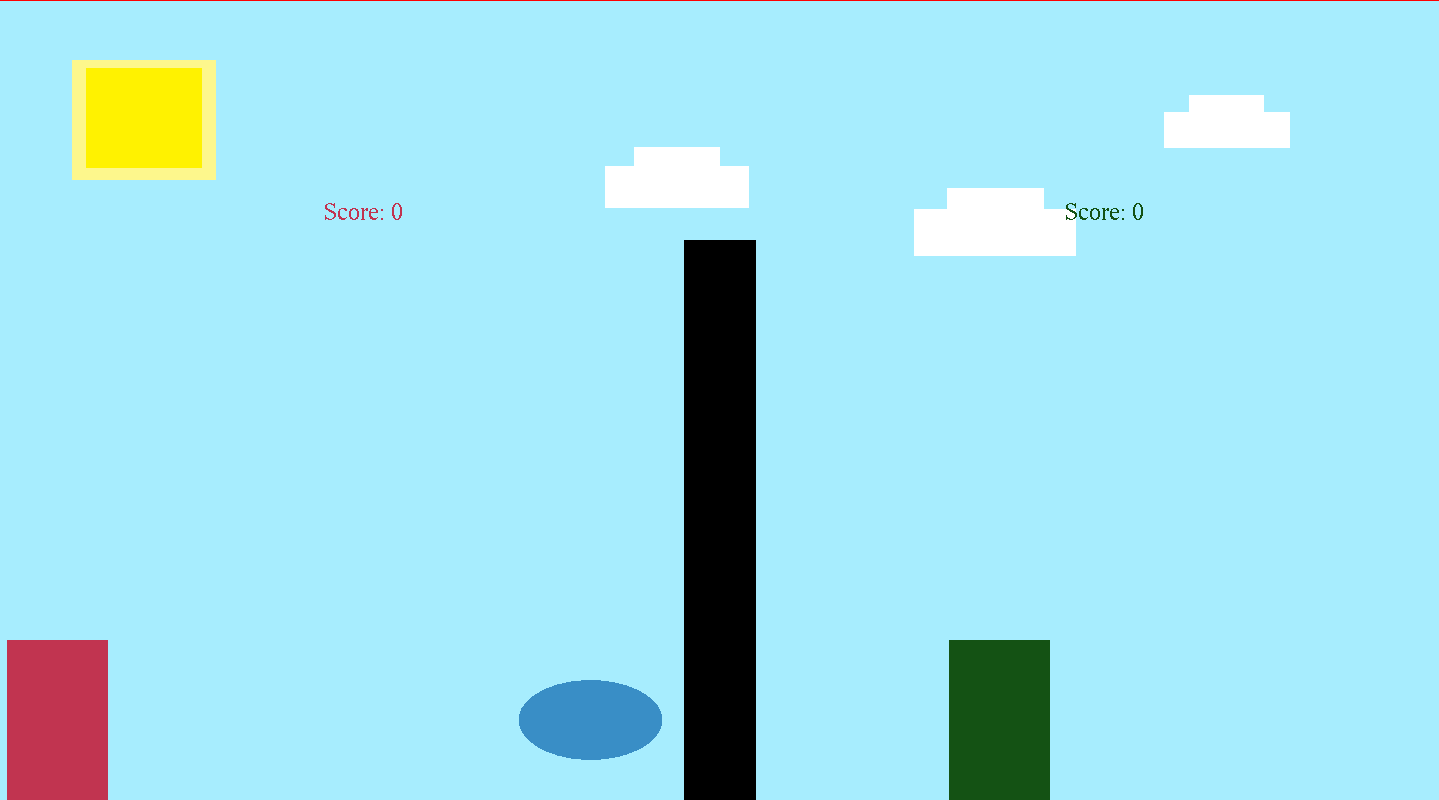
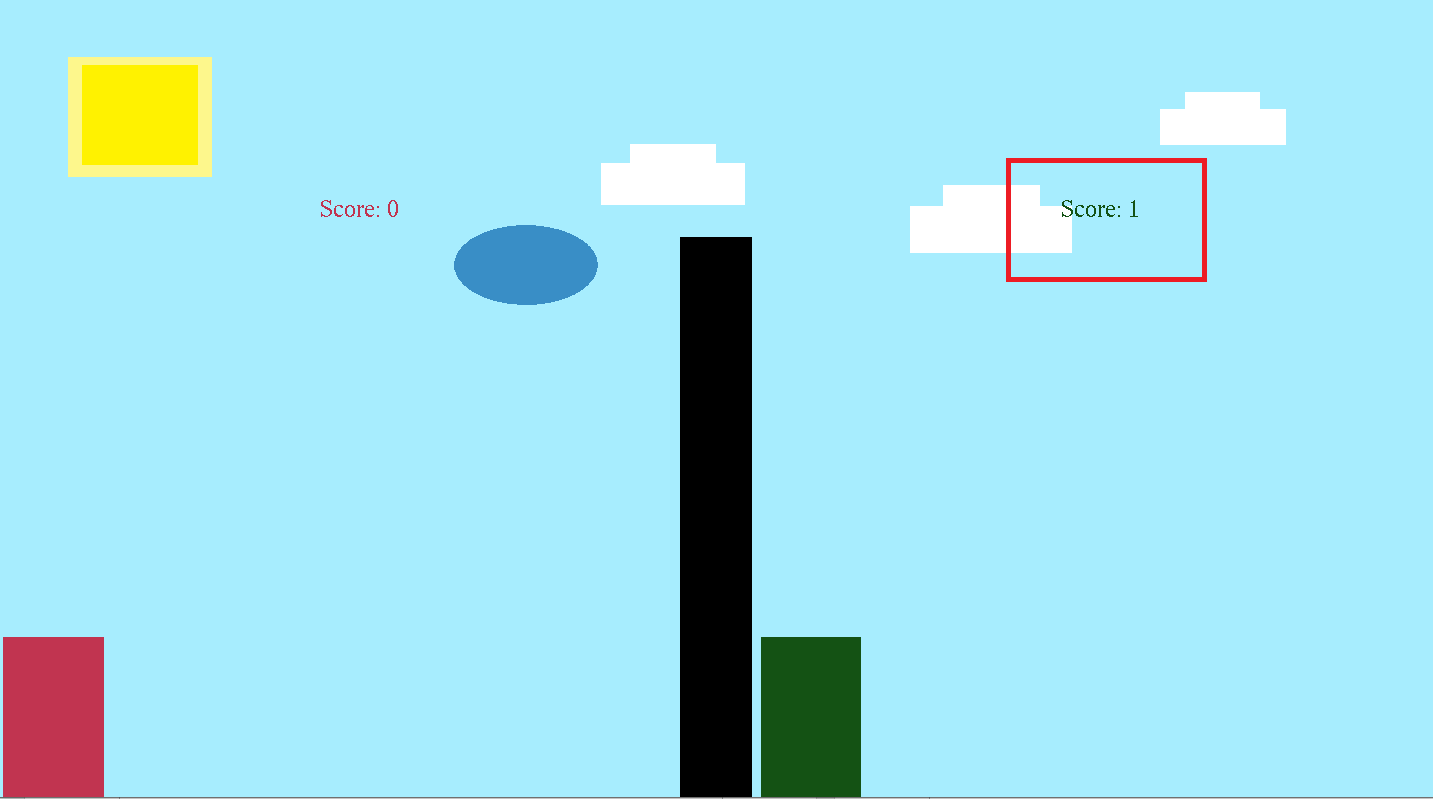


Figure 12 Opposing side score increases

Figure 11 Player failing to catch the ball

Reach the specified number of points to win the game (Figure 13).

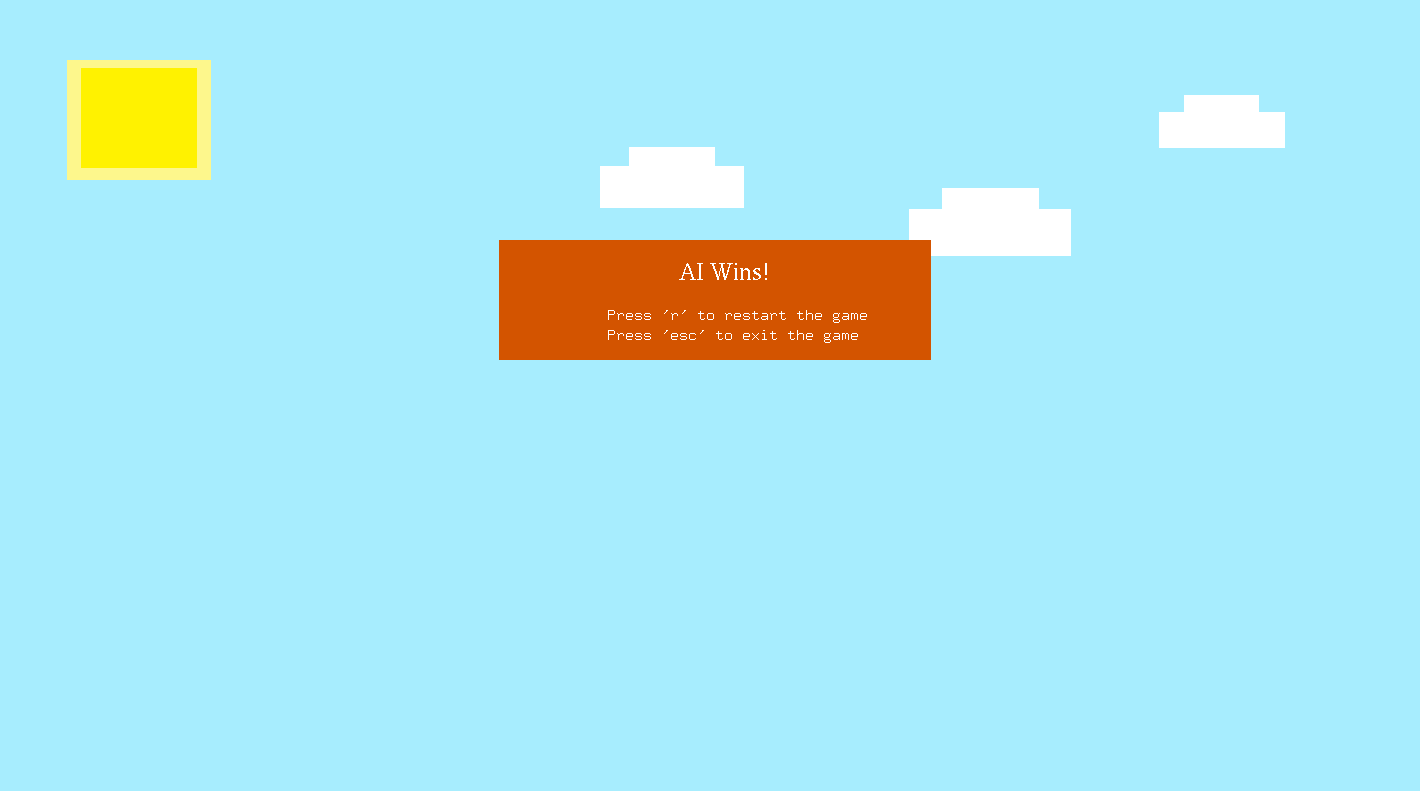


Figure 13 Winning Screen

*\*You can toggle between the Fixed Screen and Partial Screen Modes by pressing “z”(Refer to Section 3 and Figures 6,7).*

## Additional Features

We have implemented several additional features to our program and they are as follow:

1. **Game Sounds**

* We have implemented Game Sounds at different portions of the game
  + 1st Portion: Running the Game will play **the Game Theme Song**.
  + 2nd Portion: Clicking the “Start” Button will trigger the **Start Sound**.
  + 3rd Portion: When the ball collides with points within its defined collision region, a **Bouncing Sound** will be played.
  + 4th Portion: When the Player/AI scores a point, a **Cheering Sound** will be played.
  + 5th Portion: When the Player/AI wins the game, the **Ending Theme Song** will be played

1. **Restarting the Game**

* We have implemented a function where by the user will be able to restart the game by pressing the ***“r”*** key. This will reset the game entirely to its initial state *(Revert back to the Start Screen)*

1. **Exiting the Game**

* User will be able to exit the game straight by pressing the ***“esc***” key.

1. **Switching between Fixed/Partial Screen Modes**

* In addition to showing a partial screen mode, we implemented a function whereby users can toggle between the fixed and partial screen modes by pressing the “z” key.

# PROBLEMS FACED AND SOLUTIONS

There are several problems that we have encountered throughout the processing of completing this assignment

Firstly, as we are new to coding with OpenGL, we are not familiar with the syntax of OpenGL, how to properly utilized the OpenGL library and how to implement a program with OpenGL. Despite the both of us having the programming background knowledge, the languages, C++ *(Only being taught the basics in our Home University)* and OpenGL programming, are a relatively new concept to us. We faced quite a number of difficulties in grasping the concept of OpenGL programming and we had to overcome these difficulties by learning through reading materials related to OpenGL and C++, trial and error, and watching online tutorials.

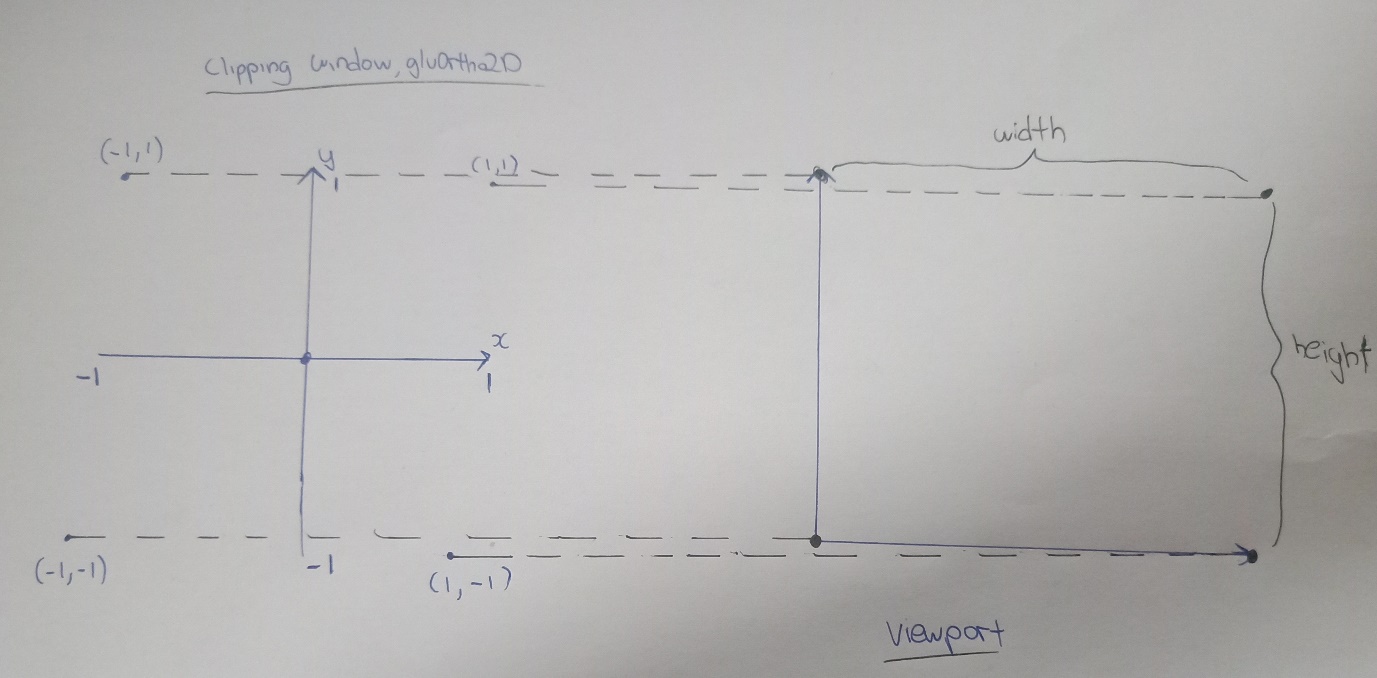
Secondly, the concept of Viewport and Clipping Area was very confusing for us and we faced several difficulties in understanding them. We got to understand the concept of Viewport and Clipping Area by drawing out how the clipping area is being mapped to the view port (Figure 14).

Figure 14 Mapping Clipping Area to Viewport

Thirdly, we encountered an issue whereby when we want to transform a specific object, all the other objects that we have previously drawn were being transformed as well. We did a bit of research online and realized that as OpenGL is based on a state machine and we can use the two commands, glPushMatrix() and glPopMatrix() to apply transformations to a specific object without affecting other drawn objects. glPushMatrix() allows us to store the current state that we are in and glPopMatrix() allow us to return to our previous state and thus any translations done within these two commands will only affect a specific object.

Next, we faced an issue regarding collision detection. Initially we define that the ball will collide with any of the defined regions of collision if the *<x,y>* coordinates of the ball is equivalent to the *<x,y>* coordinates of the area of collision. The problem we encounter was that the ball always went past the <x,y> coordinates before collision is being detected (Figure 15).

Radius of Ball

Ball’s <x,y>

Figure 15 Initial Collision Area

Ball’s <x,y> + Radius of Ball

Figure 16 Intended Collision Area

After analyzing the <x,y> coordinates at which collision is being detected, we realized the fact that the ball’s <x,y> is actually located at the center. Thus, if we want collision to be detected when the edges of the ball comes into contact with a region of collision, we have to factor in the **radius** of the ball into our equation for calculating collision (Figure 16).

Lastly, the most **major problem** that we encountered was displaying a partial screen mode where the window is centered at and moving with the ball, with the ball being situated at the center of the screen. We actually got an idea of how to resolve this after grasping the concept of the viewport and clipping area together with the concepts taught in class on the topic of 2D Viewing. We also went to read more about the Modelview matrix and the projection matrix before we embark on solving this problem.

We realized we can actually reduce the clipping area that is being mapped to the viewport to simulate a zoom effect. Next, we had to tackle the issue of centering the ball in the zoomed window. We accomplished this by creating the clipping area based off the current <x,y> coordinates of the ball (Figure 17).

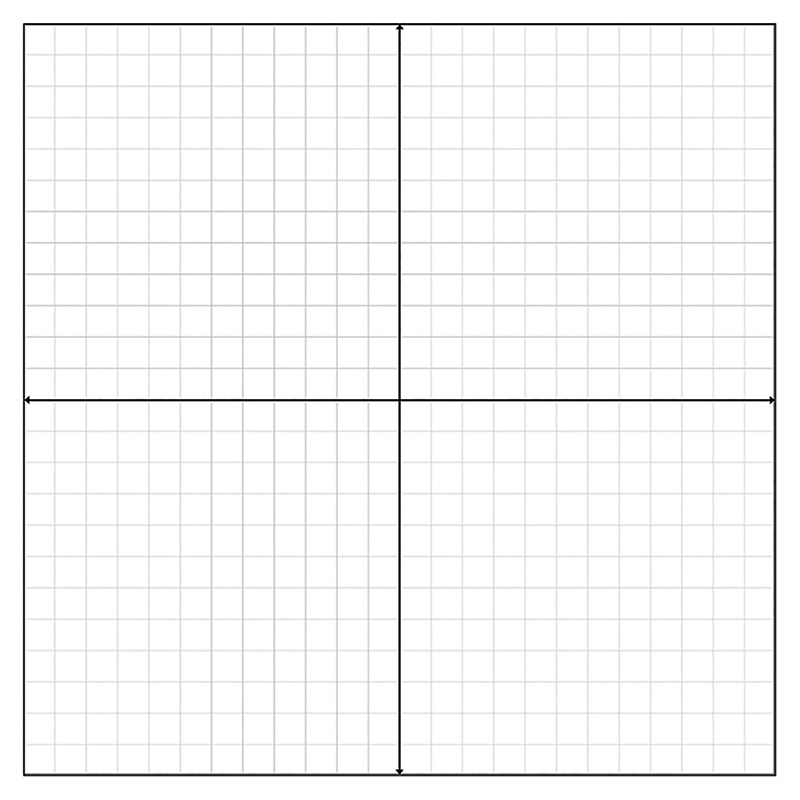


Figure 17 Clipping Area (Zoomed)

ZOOM\_FACTOR

Y - ZOOM\_FACTOR

X + ZOOM\_FACTOR

(x + ZOOM\_FACTION, y - ZOOM\_FACTOR)

(x – ZOOM\_FACTION, y - ZOOM\_FACTOR)

(x + ZOOM\_FACTION, y + ZOOM\_FACTOR)

(x – ZOOM\_FACTION, y + ZOOM\_FACTOR)

Y + ZOOM\_FACTOR

X – ZOOM\_FACTOR

(x,y)

The next problem we faced was that if the ball was constantly moving, how will we be able to move the screen according to the ball’s movement. We solved this by translating the Modelview Matrix based of the *x-speed* and *y-speed* of the ball for every movement of the ball. For example, if the ball’s *x-speed* and *y-speed* are **-0.2** and **-0.1** respectively *(Ball is moving towards the left and downwards),* we will want to translate the Modelview Matrix by the **negation** of the ball’s *x-speed* and *y-speed* (*Translate Modelview Matrix right by 0.2 and downwards by 0.1).* We need to translate the negation because if we simply translate by the ball’s x-speed and y-speed without any negation, the view that we see will simply be a static one.

*\*These theories were being formulized by as and finalized after hours of brainstorming and countless trails and errors.*

Also, since we have created the partial screen mode, we wanted to implement a feature whereby the user will be able to toggle back to the original full screen mode and vice versa. In order to achieve this, we stored all the translations we have made during the partial screen mode into a variable and for switching back to the fixed screen mode, we simply translate the current Modelview Matrix by the negation of the accumulated value of translation variable that we defined. We then switched the Clipping Area back to the one that we have used initially for the fixed screen mode.

# IMPROVEMENT TO PROGRAM

There are still numerous improvements that can be made to our program.

Firstly, the “Texts” that are currently displayed to the user are Bitmap characters and the limitations to these characters is that their font sizes are fixed. Improvements can be made if we used render the texts using the FreeType library and utilizing fonts, glyphs and shaders.

Next, all the objects being displayed on the screen are actually very simple objects without any forms of design. In the future, we could apply texture rendering using a sprite sheet/texture map and mapping it to the Player/AI/Ball/Net to order to beautify the game.

Currently, we utilized the *PlaySound* function in C++ to play our game sounds and the limitations to the function is that we are unable to play multiple sounds concurrently (*Playing a Second Sound will Stop the First Sound)*. We would like to improve our program in the future by utilizing other Sound Effect Engines such as FMOD or DirectX to play sounds concurrently in order to make the gameplay more interactive.

Currently, all the defined regions of collisions are either horizontal/vertical planes, and thus the incidence and reflection angle of the ball is always the same. The velocity of the ball is always fixed, Adding more regions such as a jagged, curved region of collision of the screen and calculating the ball’s reflection angle more dynamically together with changing the velocity of the ball upon collision will help to improve the game in terms of realisticity.

Lastly, our game only supports movement of the characters in the left/right direction and the user have to “wait” for the ball to “come” to them in order to “hit” the ball. We would like to implement jump functions to the characters. With the jump functions, the gameplay will become much more dynamic and thus gameplay will be significantly improved.

# CONCLUSION

In summary, through this assignment, we have gained a better understanding about the concepts of OpenGL such as the differences between a Modelview Matrix and Project Matrix and how the Clipping Area is being mapped to the Viewport. We have also learnt how to use OpenGL with C++ to develop a simple 2D game. Also, as we did not have a lot of C++ background, this assignment helped us to brush up on our C++ knowledge and skills. The most important aspect is that we were able to apply the concepts that we were taught during class to our assignment and gained a even better understanding them. All in all, this assignment has served as an important foundation and stepping stone for us to move even further into our study of Computer Graphics.

# References

1. Codersource.net. (2019). *Developing a simple 2D game with OpenGL (Ball with Bat) – tutorial 9 | CoderSource.net*. [online] Available at: http://www.codersource.net/2011/02/06/a-simple-2d-game-with-opengl-ball-with-bat/ [Accessed 3 Mar. 2019].
2. Collision, D., Calsbeek, J. and Gouveia, D. (2019). *Determine Resulting Angle of Wall Collision*. [online] Game Development Stack Exchange. Available at: https://gamedev.stackexchange.com/questions/23672/determine-resulting-angle-of-wall-collision [Accessed 1 Mar. 2019].
3. [3] Lazyfoo.net. (2019). *Lazy Foo' Productions - OpenGL Tutorials*. [online] Available at: http://lazyfoo.net/tutorials/OpenGL/index.php [Accessed 3 Mar. 2019].