TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A PROJECT REPORT ON “CHESS”

Submitted by:

Aashirbad Bhandari(078BEL003)

Alish Thapa (078BEL010)

Arpit Pokhrel (078BEL020)

Bhumi Raj Sharma (078BEL025)

Submitted to:

Department of Electronics and Computer Engineering

ACKNOWLEDGEMENT

We would like to express our sincere gratitude towards our lecturer, Bibha Sthapit mam, for her constant guidance, inspiring lectures and precious encouragement.

We would also like to thank the Department of Electronics and Computer Engineering, Institute of Engineering, Pulchowk Campus for providing us the opportunity to develop a project that will enhance our knowledge in Object Oriented Programming with C++ and provide us a new experience of teamwork. We are grateful to our friends for the help and feedback they provided whenever we needed it. We would not have done this without the constant support and encouragement from our family, so we owe our deepest gratitude to our family.

Finally, we would like to express our earnest gratitude to all individuals who, through their direct or indirect involvement, granted their helping hands to accomplish our project timely and efficiently.

Authors:
Aashirbad Bhandari
Alish thapa
Arpit Pokhrel
Bhumi Raj Sharma

TABLE OF CONTENTS

INTRODUCTION

Chess is a widely popular game played in different forms all around the world. It is a two player turn based strategy game involving different pieces. The chess game has a really specific set of rules that should be followed while making a move.

We took it as a challenge to include all the rules of chess in our project. Our project **Chess** is a mock up of the chess game made by following the rules set by FIDE(the international chess federation). In our rendition of the chess game we have implemented the movement of the pieces, special movements like En-passant, promotion and castling. This is done by checking the position of the pieces in the board and proving desired output. The game can end in four conditions: checkmate, draw, dead position or if a player resigns the game awarding victory to the opponent.

The game was made using C++ with the goal of applying Object oriented programming concepts to create the game of chess. We used a free graphics library called SDL2(Simple DirectMedia Layer) to implement the graphics, SDL2_image to load and display sprites or images and SDL2_ttf for displaying fonts. We also used SDL_mixer to play audio.

OBJECTIVES

The main objectives of our project are as follows:

1. To create a project on Object Oriented Programming and understand its concepts better.
2. To learn the basics of game development.
3. To be familiarized with graphics programming and game development using SDL in C++ programming language.
4. To be able to compete with computers in our chess game.
5. To optimize the program in terms of time and space up to the greatest extent possible.
6. To build an attractive UI for the users to help them interact easily with our game.
7. To learn to work and communicate effectively in a team.
8. To be prepared to work on major projects in the coming year.

LITERATURE REVIEW

Introduction to C++

C++, as we all know, is an extension to the C language and was developed by Bjarne Stroustrup at Bell Labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features. C++ is a statically typed, free form, multi-paradigm, compiled general-purpose language.

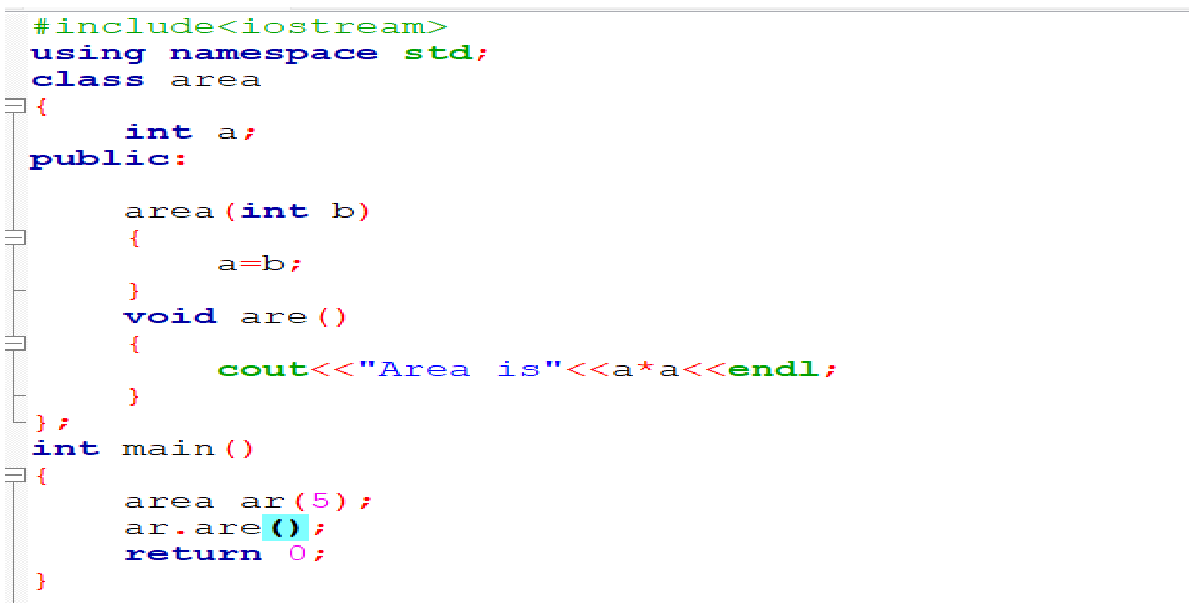
It is an Object Oriented Programming language but is not purely Object Oriented. Its features like friend and virtual violate some of the very important OOP concepts such as data hiding, so this language can't be called completely Object Oriented.

Structure of Code

A C++ program is structured in a specific and particular manner. In C++, a program is divided into the following four sections:

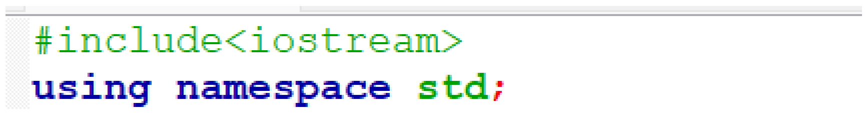
1. Standard Libraries Section
2. Class Definition Section
3. Functions Definition Section
4. Main Function Section

For example, let us look at the implementation of the simple program:



```
#include<iostream>
using namespace std;
class area
{
      int a;
public:

      area(int b)
      {
            a=b;
      }
      void are()
      {
            cout<<"Area is"<<a*a<<endl;
      }
};
int main()
{
      area ar(5);
      ar.are();
      return 0;
}
```

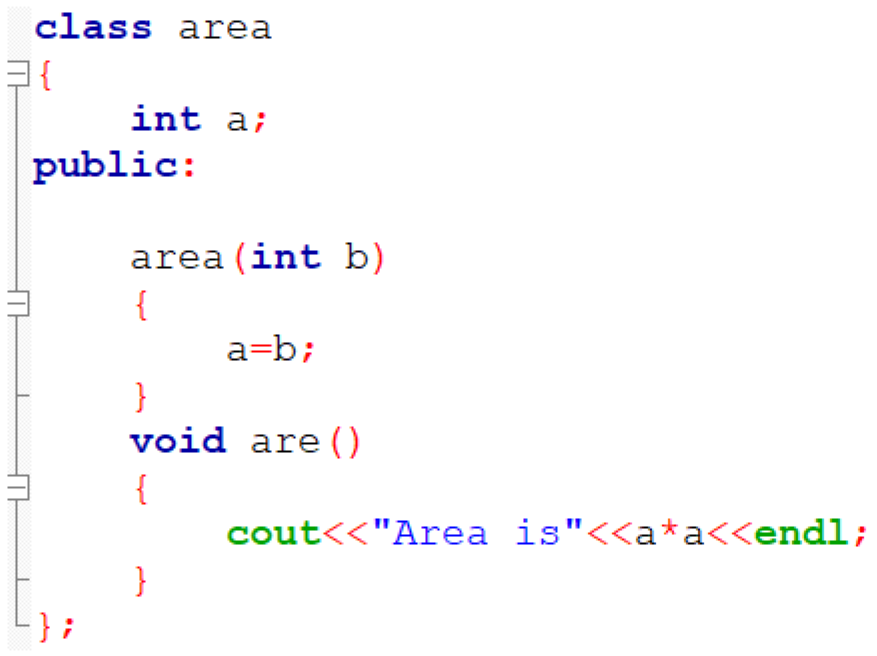
1. **Standard Libraries Section**



```
#include<iostream>
using namespace std;
```

1. **#include** is a specific preprocessor command that effectively copies and pastes the entire text of the file, specified between the angle brackets, into the source code. This file is also called a header file.
2. The file is merely for input output streams. This header file contains code for console input and output operations. This file is a part of the std namespace.
3. **namespace** is a prefix that is applied to all the names in a certain set. For example, **iostream** file is defined in a set which we call std and it defines two names used in this program cout and endl.
4. If **using namespace std;** is used, the compiler understands that the names like **cout** and **endl** of the std namespace are being used.

2. Class Definition Section



```
class area
{
    int a;
public:

    area(int b)
    {
        a=b;
    }
    void are()
    {
        cout<<"Area is"<<a*a<<endl;
    }
};
```

The classes are used to map real world entities into programming. The classes are key building blocks of any C++ program. A C++ program may include several class definitions. This is the section where we define all of our classes. In the above program **PrintText** is the class and text is its object.

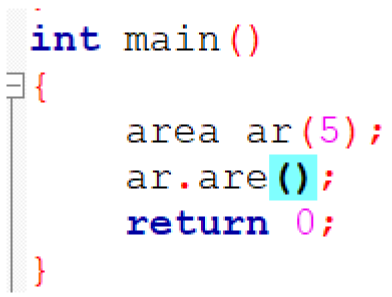
3. Function Definition Section

C++ allows the programmer to define their own function. A user defined function groups code to perform a specific task and that group of code is given a name (identifier). When the function is invoked from any part of the program, it executes the codes defined in the body of the function.

In this program class area is used to find the area of the default variable given to the class in the form of a parameterized constructor.

4. Main Function Section

main() function is the function called when any C++ program is run. The execution of all C++ programs begins with the **main()** function and also ends with it, regardless of where the function is actually located within the code.The object of class is created in the main function and it is used to invoke the member function of the class.



```
int main()
{
    area ar(5);
    ar.are();
    return 0;
}
```

Features of C++

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object oriented paradigm. It is an imperative and a compiled language. Some of its main features are:

1. **Namespace:**

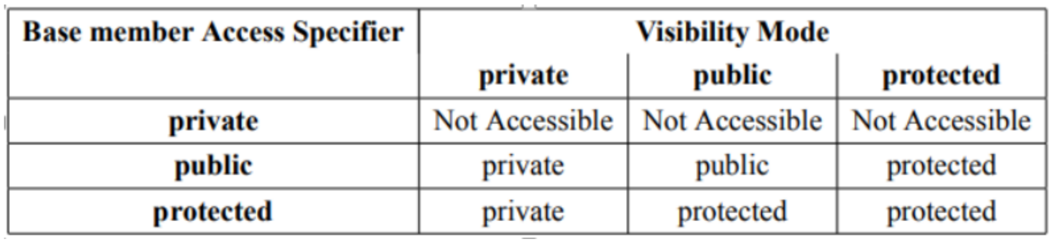
   A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when the code base includes multiple libraries.

2. **Inheritance:**

   Inheritance is a process in which one object acquires some (or all) of the properties and behaviors of its parent object automatically. In such a way, one can reuse, extend or modify the attributes and behaviors which are defined in other classes, from existing classes. The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

   The syntax for derived class is:
   class class_name::visibility_mode base_class{//body of derived class};

   There are three visibility modes in which a derived class inherits from its base class. They are private, public and protected. Following table clarifies the concept:



Base member Access Specifier	Visibility Mode		
	private	**public**	**protected**
private	Not Accessible	Not Accessible	Not Accessible
public	private	public	protected
protected	private	protected	protected

3. Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered as one of the important features of Object Oriented Programming. There are two types of polymorphism:

A. Compile time polymorphism

It refers to early or static binding.Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters).

It is also called early binding because the calls are already bound to the proper type of functions during the compilation of the program depending upon type and sequence of parameters.

E.g. Function overloading, operator overloading

B. Runtime polymorphism

It refers to Late or dynamic binding.A function is said to exhibit dynamic polymorphism if it exists in various forms,and the resolution of different function calls are made dynamically during execution time. This feature makes the program more flexible as a function can be called, depending on the context.

E.g. Function overriding using Virtual Functions.

4. **Templates**

   A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

   C++ adds two new keywords to support templates: **'template**' and **'typename'**. The second keyword can always be replaced by the keyword **'class'**.

5. **Constructors and Destructors**

   In C++, constructors and destructors are special member functions of a class that are defined by the user. A constructor is used to initialize the object of the class while a destructor is called by the compiler when the object is to be destroyed. A constructor is called when the memory is allocated to an object, while a destructor is called when memory is needed to be deallocated. Both constructors and destroyers are implicitly called by the compiler.

6. **Encapsulation**

   The wrapping up of data and functions into a single unit(called class) is known as encapsulation. Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation. It also hides the internal implementation details of a class from external code.

7. **Method Overriding**

   Method overriding in OOP is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided ny one of its superclasses or parent classes.In addition to providing data-driven algorithm-determined parameters across virtual network interfaces, it also allows for a specific type of polymorphism.

8. **Abstraction**

   Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost,and functions to operate on these attributes.

Simple DirectMedia Layer(SDL)

Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games including Valve's award winning catalog and many Humble Bundle games.

While not all of these will be used in our game-programming adventures, some of them are invaluable and make SDL an even better framework to use to develop games. We will be taking advantage of the new hardware-accelerated 2D graphics to make sure our games have excellent performance.

SDL2 libraries also contain extensions to keep SDL as light as possible. Some of the libraries are SDL_image, SDL_net, SDL_mixer, SDL_ttf, SDL_rtf, etc. SDL_image is used to load different images, SDL_net is used for cross platform networking, SDL_mixer is an audio mixer library that supports WAV, MP3, MIDI and OGG. SDL_ttf is used to write using fonts in the program and SDL_rtf is Rich Text Format library.

METHODOLOGY

The following methods were used for the completion of the project:

Planning and gathering information

We researched online and gathered various information about chess and different libraries required for the project. The planning for the work division among the three of us was done.

System Model and Design

A basic model of the game was proposed documenting the information about the project was carried out before the coding and implementation during the proposal submission. New testable features were added regularly and validated.

Software Development

This project is based on C++ programming language and Object Oriented Programming concept using SDL 2 (Simple DirectMedia Layer) library for graphics. Additional SDL2 Libraries (SDL2 ttf, SDL2 mixer, SDL2 image) are used for text, sound and image in the program. As a compiler, we will be using gcc (using mingw) for windows and GCC for unix systems. For linux, we have used Cmake for build automation and debugging. And for windows systems, scripts have been written.

Testing and Implementation

The program went through testing to measure usability, functionality and performance. We used git for version control and GitHub for sharing of code among our team members. Adhering to the style of OOP, different classes were made with suitable access specifiers and data hiding was given a priority by making the members private as much as possible. The concept of code reusability, data abstraction were implemented in the project. We have created an object for each of the elements of our game and then worked on their interaction with each other.

OBJECT ORIENTED APPROACH

Since the primary objective of the project was to learn object oriented programming concepts by practically implementing them, we have tried our best to implement the features of object oriented programming in our project.

Classes

We have created around 10 classes that specify various data and their operations. Through some classes, we have tried to abstract real world chess entities. We have created different files for each class to achieve better readability and easy maintenance of the large program. We have separated header files for class declarations while the member function definitions of a class are stored in separate class implementation files.

1. **Piece:** It is a parent class for all the pieces. Different pieces like Rook, Bishop, Queen, King, Knight, Pawn are derived from the piece class. Each piece has information about its position and color. Each piece also has a virtual function to generate all moves(even moves that are not possible due to checks). The piece class has a function to generate legal moves which does so by going through each move and checking if it causes checks and other conditions.
2. **Board:** Stores information about the 8x8 board, and other information required for the game and also provides functions to perform moves, check the game state, and various other actions.
3. **Error:** It inherits from ***std::exception*** class and stores information about various errors we can encounter during the execution of the program.
4. **Game:** This class manages the overall game state,including the board position,players turn,move history and possibly additional game related information.
5. **Sound:** Manages the audio of the program.
6. **Texture:** Creating an interface to interact with SDL Texture. It also has a static data member ***FONTS*** that stores various font sizes loaded in the program for reusability and speed improvements.

7. **Test:** It is used to perform automated testing of the various components and functionalities of the program. Automated testing helps ensure that the code behaves as expected, catches bugs early, and maintains the correctness of the application. It is used for testing FEN parsing. FEN stands for Forsyth-Edwards Notation and is used for describing a particular board position in a chess game. The purpose of FEN is to provide all the necessary information to restart a game from a particular position.
8. **Player:** It is used to represent the human or chess engine participating in the game. It is the parent class for ***Human*** and ***Stockfish*** classes**.** This class stores information about the player, and has a function to get the move chosen by the player. Currently, we use this function to get moves from the chess engine and handle the human player's input from the UI. We use Stockfish chess engine as our engine of choice but it is possible to use any chess engine that supports UCI protocol with minimal changes to the code.
9. **Gamescene:** This class is the parent class for ***chessGame*** and ***gameMenu*** class. It encapsulates the common functionality between the chess game and chess menu such as background, objects, setup, update and rendering method. It also stores information about the currently displayed UI and handles events needed to update it.

Abstraction

Abstraction is a feature of object oriented programming that hides the internal details of how an object does its work and only provides the interface to use the service. We can manage complexity through abstraction.

In OOP, classes are used for creating user-defined data for abstraction. When data and its operation are presented together, it is called ADT (Abstract Data Type). Hence, a class is an implementation of abstract data type. So, in OOP, classes are used in creating Abstract Data Type.

For instance, we have created a class ***GameScene*** and made it available in the program. Now we can implement the class in creating objects and its manipulation without knowing its implementation.

We created a ***Texture*** class which abstracts away the task of creating textures and rendering textures. We implemented it such that it can load entire strings as textures using the method *loadString*, load images *loadImage* and load characters using *loadChar* which provided us an interface for interacting with textures. This made it so that we could render textures and text without thinking about SDL2's functions and memory management.

Encapsulation And Data Hiding

Combining data and function together into a single unit is called encapsulation. We can say encapsulation is a protective box that prevents the data from being accessed by other code that is defined outside the box. We can easily achieve abstraction by making use of encapsulation.

We can achieve encapsulation through classes. In classes, each data or function is kept under an access specifier (private, protected or public).

1. **Public**: It contains data and functions that the external users of the class may know about.
2. **Private**: This section can only be accessed by code that is a member of a class.
3. **Protected**: This section is visible to class members, derived class members, friend functions of class, friend classes and friend classes of derived classes.

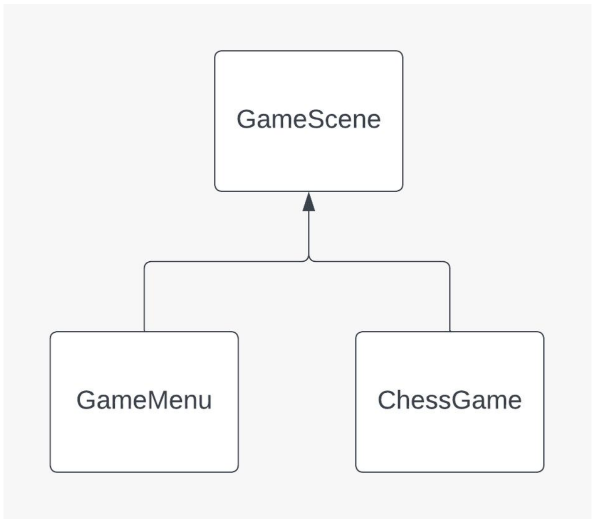
The insulation of data from direct access by the program is called data hiding. Data hiding by making them private or protected makes it safe from accidental alteration. Understanding this, we have tried to make our data private as much as possible.

Inheritance

The main purpose of using inheritance is for:

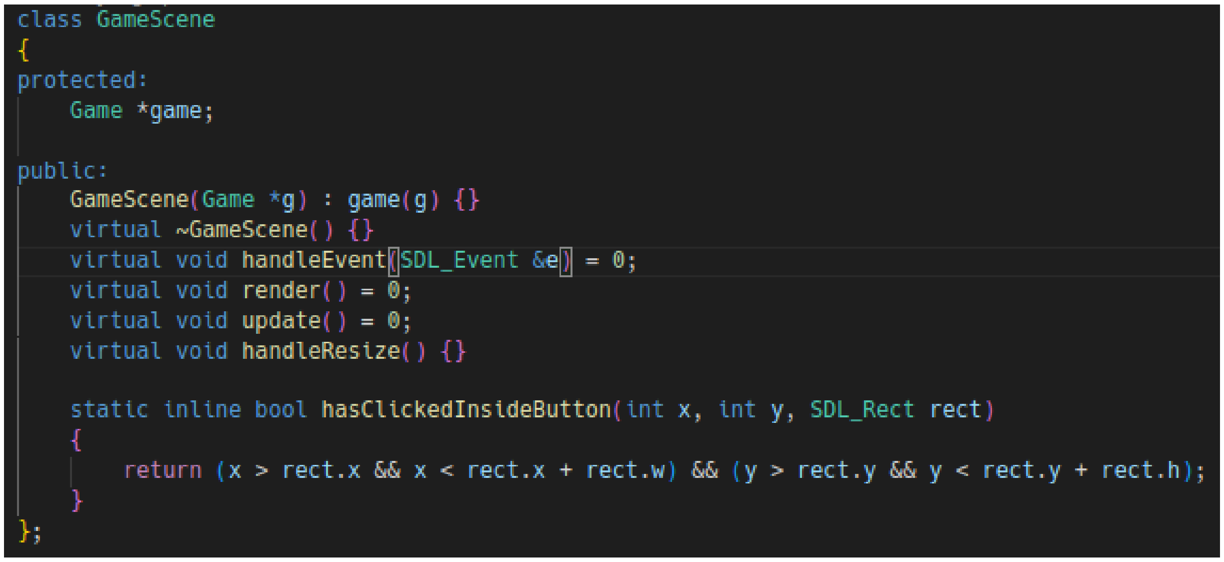
1. The reuse and extension of existing code.
2. The elimination of redundant code.

The use of Hierarchical Inheritance :



Base class: **GameScene**
Derived classes: **Chessgame** and **Gamemenu**

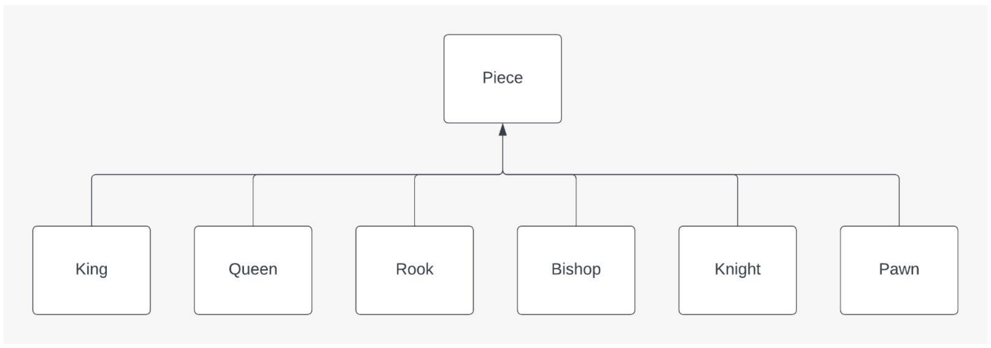
The **GameScene** class has pure virtual functions, which makes it an abstract base class.



```
class GameScene
{
protected:
    Game *game;

public:
    GameScene(Game *g) : game(g) {}
    virtual ~GameScene() {}
    virtual void handleEvent(SDL_Event &e) = 0;
    virtual void render() = 0;
    virtual void update() = 0;
    virtual void handleResize() {}

    static inline bool hasClickedInsideButton(int x, int y, SDL_Rect rect)
    {
        return (x > rect.x && x < rect.x + rect.w) && (y > rect.y && y < rect.y + rect.h);
    }
};
```

We have derived classes ***ChessGame*** and ***ChessMenu*** from the base class GameScene. The derived classes have their own different function definitions. We do this inheritance as we need to store both ***ChessGame*** and ***ChessMenu*** in a stack. We cannot store two objects of different types in a container and because they share similar functionalities. We make them derived classes as every derived class is a sub-type of base class. In this way, we can store them in a stack and render the game menu and game board as per need.

The use of hierarchical inheritance:



Base class: Piece
Derived classes: Rook, Bishop, Queen, King, Knight and Pawn. Each of the derived classes generates a set of moves and stores the moves in a *std::vector*. These moves are known as "Pseudo Legal" moves as the moves generated do not take checks into account. After each derived class generates all the moves, the Base class is used to filter out all the illegal moves, giving us an *std::vector* of legal moves.

Polymorphism

Polymorphism is another important feature of OOP. It allows different objects to respond to the same operation in different ways. The different ways of using the same function or operator depending on what they are operating on is called polymorphism.

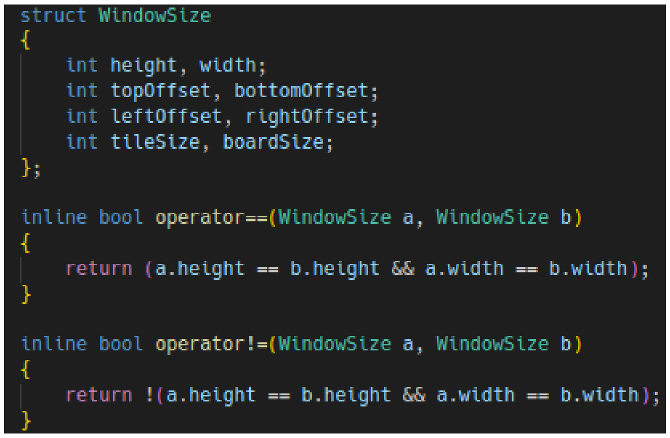
In C++, polymorphism is mainly divided into two types:

1. Compile time polymorphism

        - Function Overloading
        - Operator Overloading
2. Runtime polymorphism
        - Virtual Functions

Operator Overloading:

In our project, we have overloaded equal to, not equal to and stream operators to check if coordinates are equal and to convert our coordinate system to "chess coordinates".



```
struct WindowSize
{
    int height, width;
    int topOffset, bottomOffset;
    int leftOffset, rightOffset;
    int tileSize, boardSize;
};

inline bool operator==(WindowSize a, WindowSize b)
{
    return (a.height == b.height && a.width == b.width);
}

inline bool operator!=(WindowSize a, WindowSize b)
{
    return !(a.height == b.height && a.width == b.width);
}
```

Fig: Relational Operator Overloading

Pure Virtual Function:

Pure virtual functions is a virtual function that only has a declaration but doesn't have a definition. Since they have no definition, these functions cannot be called an object consisting of pure virtual functions cannot be created. Its usefulness comes from the fact that any class that derives from a base class consisting of a pure virtual function must implement the function for the derived class.

```
class Parent {
public:
        virtual void doSomething() = 0;
};
```

Since the function “doSomething()” is a pure virtual function, it makes the class an Abstract Class and an object of the class “Parent” cannot be created. But the function can be overridden in a Child class as:

```
class Child: public Parent {
public:
        void dosomething()
        { // Does something }
};
```

Now we can create the class ”Child” and call the method ”doSomething()” Pure virtual functions were used extensively in our project and the example can be seen in the next section.

Abstract Base Class:

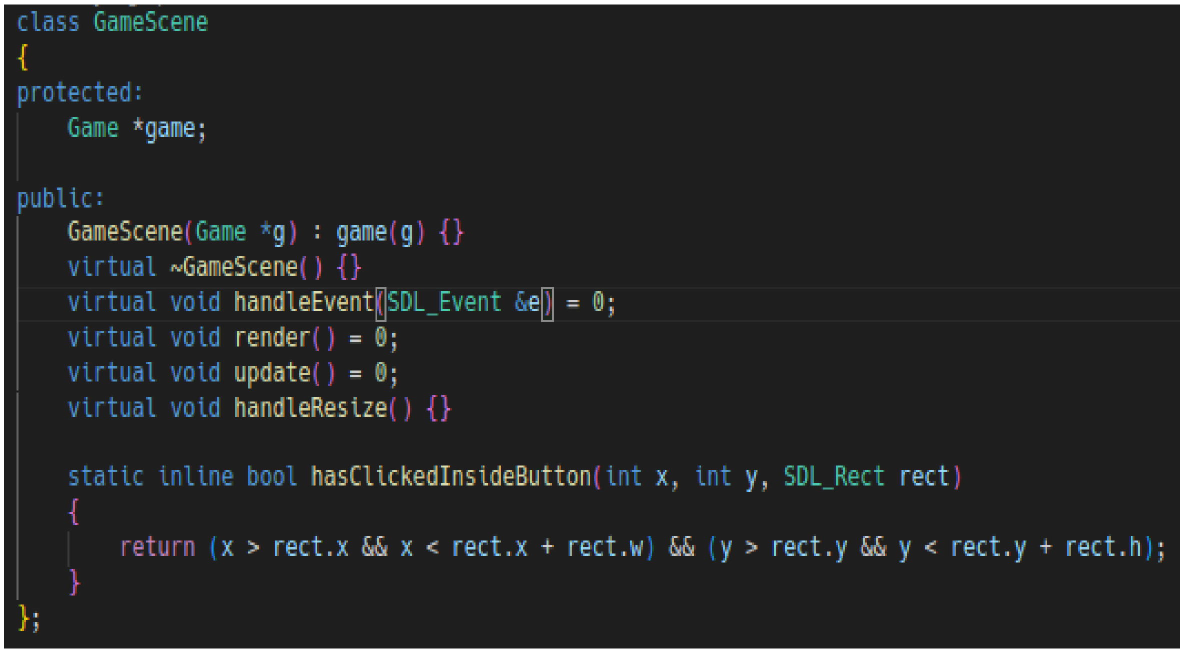
A class that has a pure virtual function is an Abstract Base Class. These classes cannot be used to instantiate an object but serve the following function.

- Deter from creation of the base class.
- Act as a base class from any derived class and allow for easy polymorphism.

Example of abstract base class:

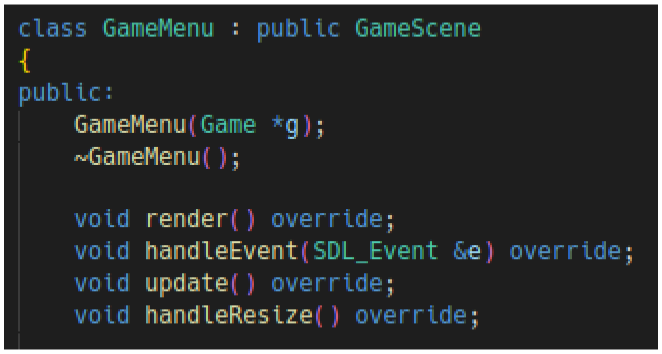
```
class Parent
{
 public:
virtual void doSomething() = 0;
 };
```

In our project we used this concept to create the concept of abstract classes and used to create two classes ***ChessGame*** and ***GameMenu***. These act as Abstract base classes for other derived classes as standalone they do not have much meaning or functionality. We used the concept of virtual classes extensively which can be seen in the code below:

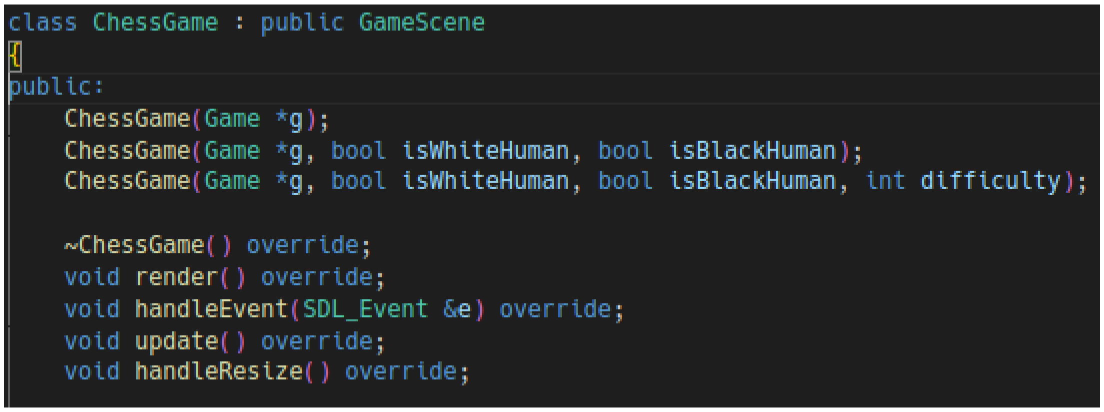


```cpp
class GameScene
{
protected:
    Game *game;

public:
    GameScene(Game *g) : game(g) {}
    virtual ~GameScene() {}
    virtual void handleEvent(SDL_Event &e) = 0;
    virtual void render() = 0;
    virtual void update() = 0;
    virtual void handleResize() {}

    static inline bool hasClickedInsideButton(int x, int y, SDL_Rect rect)
    {
        return (x > rect.x && x < rect.x + rect.w) && (y > rect.y && y < rect.y + rect.h);
    }
};
```

Fig: Parent class



```cpp
class GameMenu : public GameScene
{
public:
    GameMenu(Game *g);
    ~GameMenu();

    void render() override;
    void handleEvent(SDL_Event &e) override;
    void update() override;
    void handleResize() override;
```

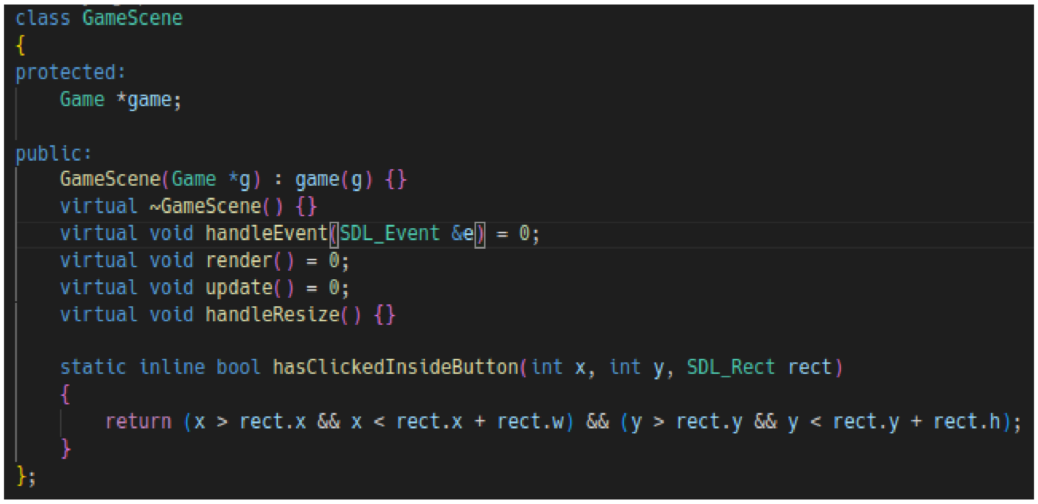


```cpp
class ChessGame : public GameScene
{
public:
    ChessGame(Game *g);
    ChessGame(Game *g, bool isWhiteHuman, bool isBlackHuman);
    ChessGame(Game *g, bool isWhiteHuman, bool isBlackHuman, int difficulty);

    ~ChessGame() override;
    void render() override;
    void handleEvent(SDL_Event &e) override;
    void update() override;
    void handleResize() override;
```

Fig: Child classes

Virtual Destructors:

Since a derived class may have many heap allocations, it is important for it to have its own destructor. For this we make the destructor of the base class virtual and override it in our derived class. We used Virtual Destructors as such as:



```cpp
class GameScene
{
protected:
    Game *game;

public:
    GameScene(Game *g) : game(g) {}
    virtual ~GameScene() {}
    virtual void handleEvent(SDL_Event &e) = 0;
    virtual void render() = 0;
    virtual void update() = 0;
    virtual void handleResize() {}

    static inline bool hasClickedInsideButton(int x, int y, SDL_Rect rect)
    {
        return (x > rect.x && x < rect.x + rect.w) && (y > rect.y && y < rect.y + rect.h);
    }
};
```

Fig: Virtual Destructor

IMPLEMENTATION

The code for this project can be viewed in the Github repository of this project: https://github.com/aashirbadb/chess/

Block Diagram

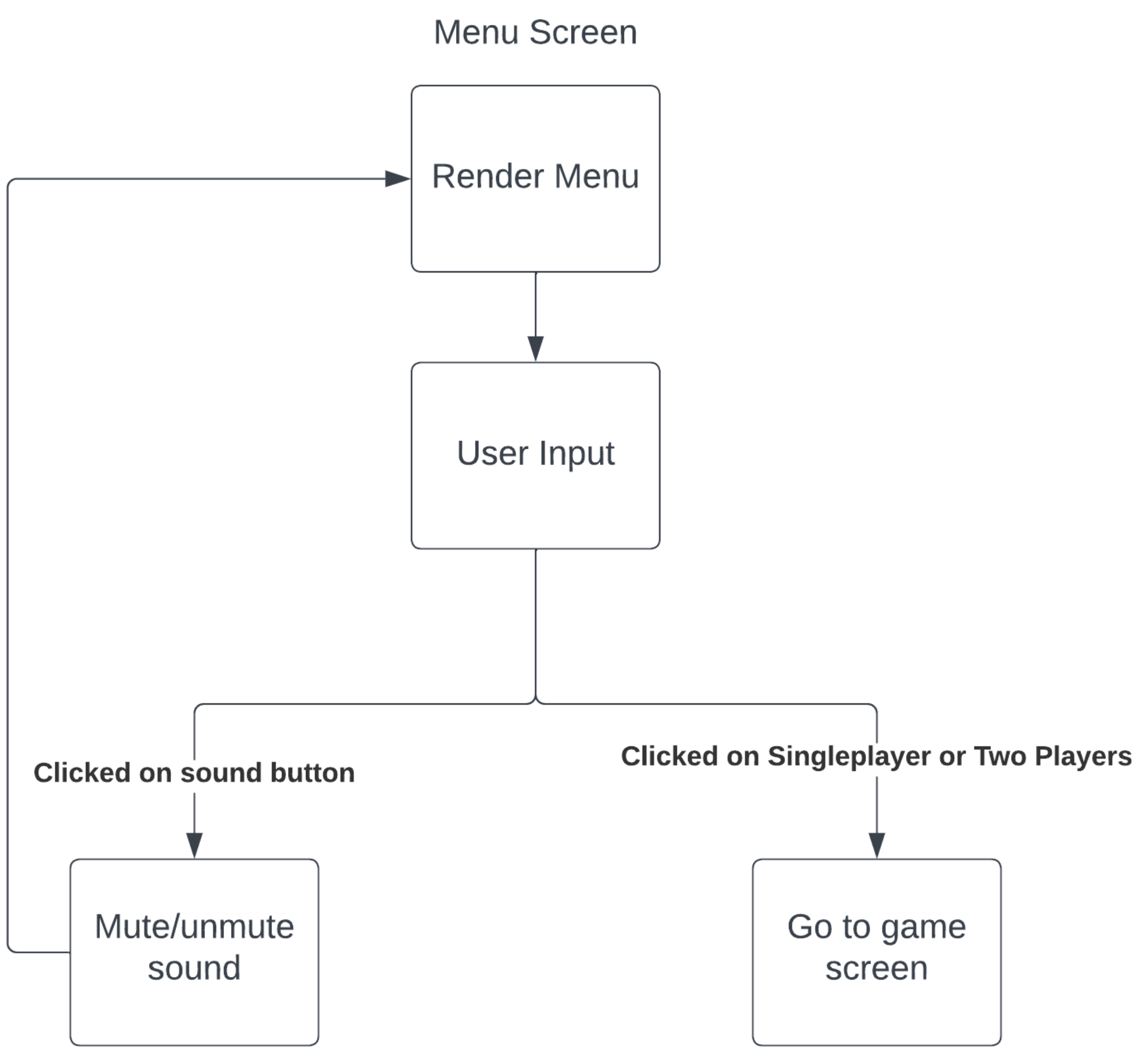


Fig: Block diagram of main menu

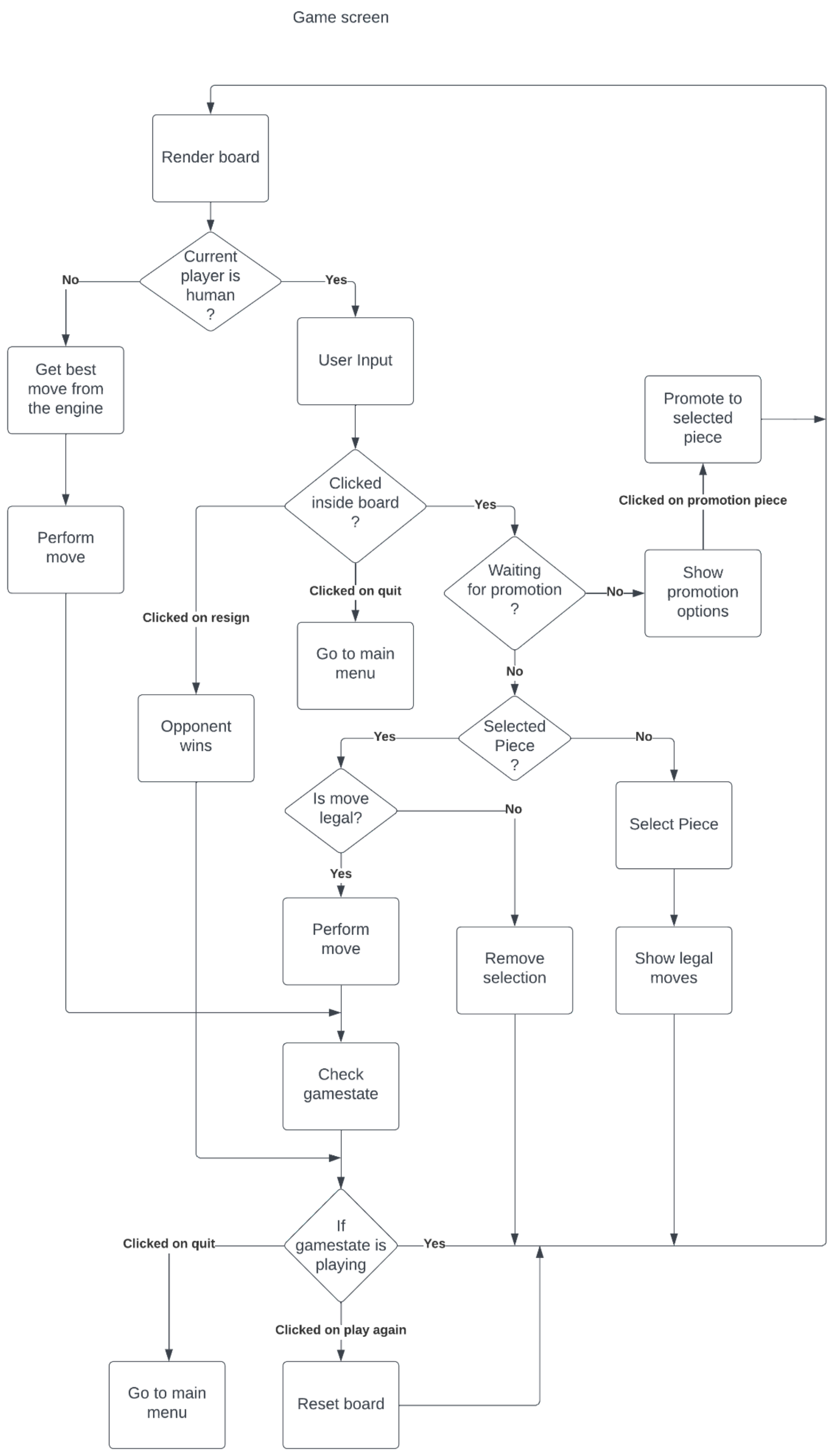
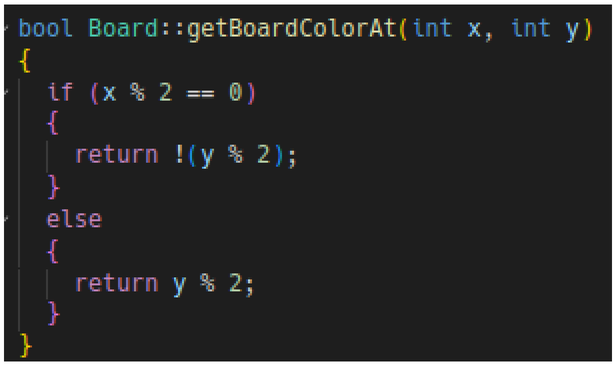


Fig: Block diagram of game screen

The Board

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)	(7, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)	(7, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	(6, 2)	(7, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	(6, 3)	(7, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)	(6, 4)	(7, 4)
(0, 5)	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)	(6, 5)	(7, 5)
(0, 6)	(1, 6)	(2, 6)	(3, 6)	(4, 6)	(5, 6)	(6, 6)	(7, 6)
(0, 7)	(1, 7)	(2, 7)	(3, 7)	(4, 7)	(5, 7)	(6, 7)	(7, 7)

An 8x8 game board was created. The top left corner of the board was made the origin. The following code has been used for the implementation of the board color.



```
bool Board::getBoardColorAt(int x, int y)
{
    if (x % 2 == 0)
    {
        return !(y % 2);
    }
    else
    {
        return y % 2;
    }
}
```

Here, true represents white tile and false represents black tile.

Pieces

We have six types of pieces in total: King, Queen, Bishop, Knight, Rook and Pawn. We have 1 King, 1 Queen, 2 Bishops, 2 Knights, 2 Rooks and 8 pawns in black as well as white. For pieces, we have created an abstract base class called ***Pieces***, and six derived classes: ***King***, ***Queen***, ***Bishop***, ***Rook***, ***Knight***, ***Pawn***. Hierarchical inheritance is being used. The pseudo legal moves have been defined separately in all derived classes while the legal moves for all are checked in the base class.

FEN Notation

FEN notation is a way of representing a chess board. The starting position can be represented in FEN as:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq – 0 1

Field 1:

The first field contains letters and numbers that represent all pieces and their locations in the board. It begins from the top row of the chess board and the slash represents a new row. The letters represent pieces as follows: r: rook, n: knight, b: bishop, q: queen, k: king, p: pawn. Lowercase letters denote black pieces and uppercase letters denote white ones. The numbers represent the amount of black spaces. For eg. If the number is 8 means that it has 8 blank spaces i.e. 8/8/8/8 : means four empty rows.

Field 2:

It tells whose turn it currently is. Here w is for white and b is for black.

Field 3:

This field represents castling availability, where k, q is for black and K, Q is for white). If there is a k, king side castling is available. If there is a q, queen side castling is available.

Field 4:

This field is the square that can be moved onto for en passant. If there is no square, it means that the last move is not a pawn moving two squares, then there is just a dash.

The last two fields are related to how many moves have been made.

<u>Interactivity</u>

Here, we check for mouse presses and check where the mouse is clicked on and move the chess pieces accordingly making sure that the players are not able to move the opponent piece.

Pseudo legal moves:
A pseudo-legal move is any move that can be made on the board based on how the pieces move. Pseudo legal moves ignore checks and checkmates. Pseudo-legal moves are:

- **King:** Moves one square in any direction.
- **Queen:** Combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.
- **Bishop:** Can move any number of squares diagonally, but cannot leap over other pieces.
- **Rook:** Can move any number of squares along a rank or file, but cannot leap over other pieces.
- **Knight:** moves in an *L* shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the only piece that can leap over other pieces.
- **Pawn:** Can move forward to the unoccupied square immediately in front of it on the same file, or on its first move it can advance two squares along the same file, provided both squares are unoccupied. A pawn can capture an opponent’s piece on a square diagonally in front of it by moving to that square. A pawn has two special moves: the en passant capture and promotion.

<u>Game States and Special Moves</u>

Check:
When a king is under immediate attack, it is said to be in check. A move in response to a check is legal only if it results in a position where the king is no longer in check. This can involve capturing the checking piece; interposing a piece

between the checking piece and the king (which is possible only if the attacking piece is a queen, rook, or bishop and there is a square between it and the king); or moving the king to a square where it is not under attack. Castling is not a permissible response to a check.

Checkmate:
The aim of the game is to checkmate the opponent; this occurs when the opponent's king is in check, and there is no legal way to get it out of check. It is never legal for a player to make a move that puts or leaves the player's own king in check.

Pawn Promotion:
The new piece replaces the pawn on its square on the same move. The choice of the new piece is not limited to pieces previously captured, thus promotion can result in a player owning, for example, two or more queens despite starting the game with one.

En passant:
It is a special pawn capture that can only occur immediately after a pawn makes a move of two squares from its starting square, and it could have been captured by an enemy pawn had it advanced only one square. The opponent captures the just moved pawn "as it passes" through the first square. The result is the same as if the pawn had advanced only one square and the enemy pawn had captured it normally. The en passant capture must be made on the very next turn or the right to do so is lost.

Castling:
Castling is a special move in chess that uses both a rook and the king. In castling, the king is moved two squares toward the rook, and the rook moves past the king to the square right next to where the king has moved. Castling takes one move, and is the only way for a player to move two of his own pieces on the same move. Castling can be done on either side of the board. Castling can either be done on the king-side (also known as castling short) or on the queen-side (also known as castling long).

Rules of Castling:

- Neither the king nor the rook being used to castle have moved in the game.
- The king is not in check, and is not moving into check or through check.
- There are no pieces between the king and the rook.

Once a piece is clicked, the possible moves and captures are highlighted. Similarly, the last position and the current position are also highlighted.

Legal moves:
For legal moves, we go through every pseudo-legal move and check if the king is being checked. If it isn't, then we add it to the legal moves.

End of the Game

Win: The king is in check and the player has no legal move.

Resignation: A player may resign, conceding the game to the opponent. Most tournament players consider it good etiquette to resign in a hopeless position.

Stalemate: If the player to move has no legal move, but is not in check, the position is a stalemate, and the game is drawn.

Draw: The *Fifty-move rule* in chess states that a player can claim a draw if no capture has been made and no pawn has been moved in the last fifty moves(for this purpose a "move" consists of a player completing a turn followed by the opponent completing a turn).The purpose of this rule is to prevent a player with no chance of winning from obstinately continuing to play indefinitely or seeking to win by tiring the opponent. We automatically draw the game without the player needing to claim a draw.

RESULTS

We were able to achieve the objects of the project and a chess game was designed with all the rules and the necessary features. The development of the game is fulfilled but still some additional features can be added in the future.

From this project, we understood the concepts of Object Oriented Programming better, learnt the basics of game development and graphics programming using SDL2 library. We got better acquainted with version control with git and collaboration with GitHub. Finally, we learnt about how to work and communicate effectively in a team which has prepared us to work on major projects in the coming years.

The following screenshots from different states throughout the game illustrate the final result of the project:

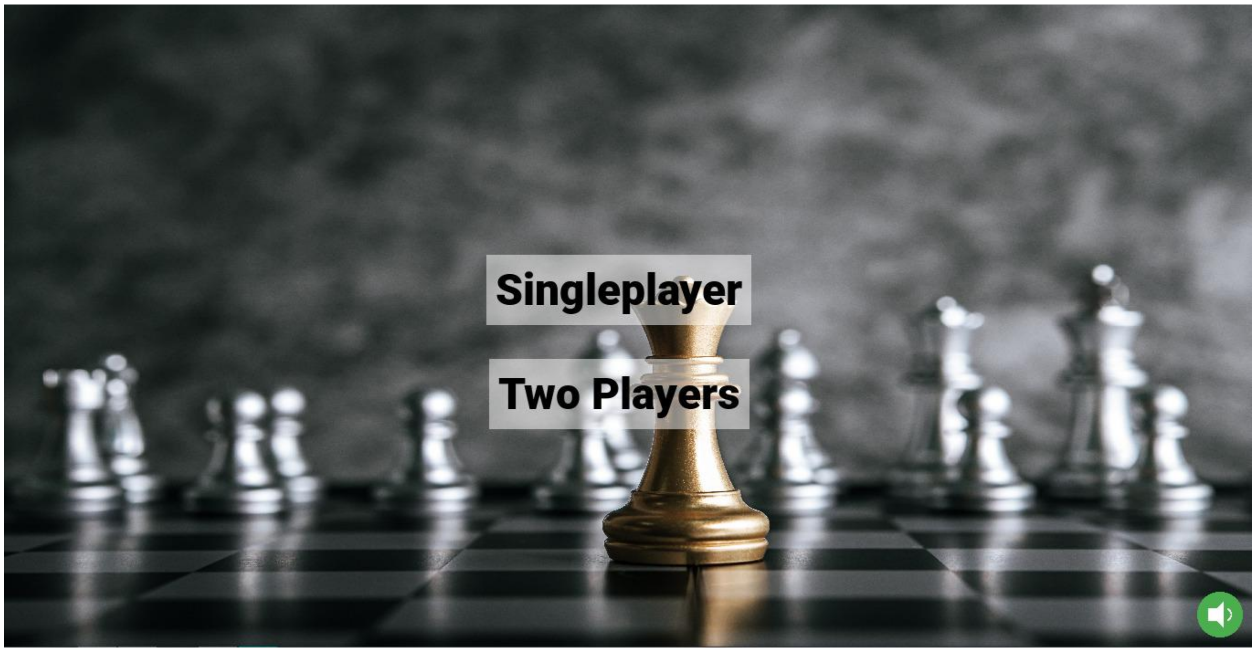


Fig: Main menu

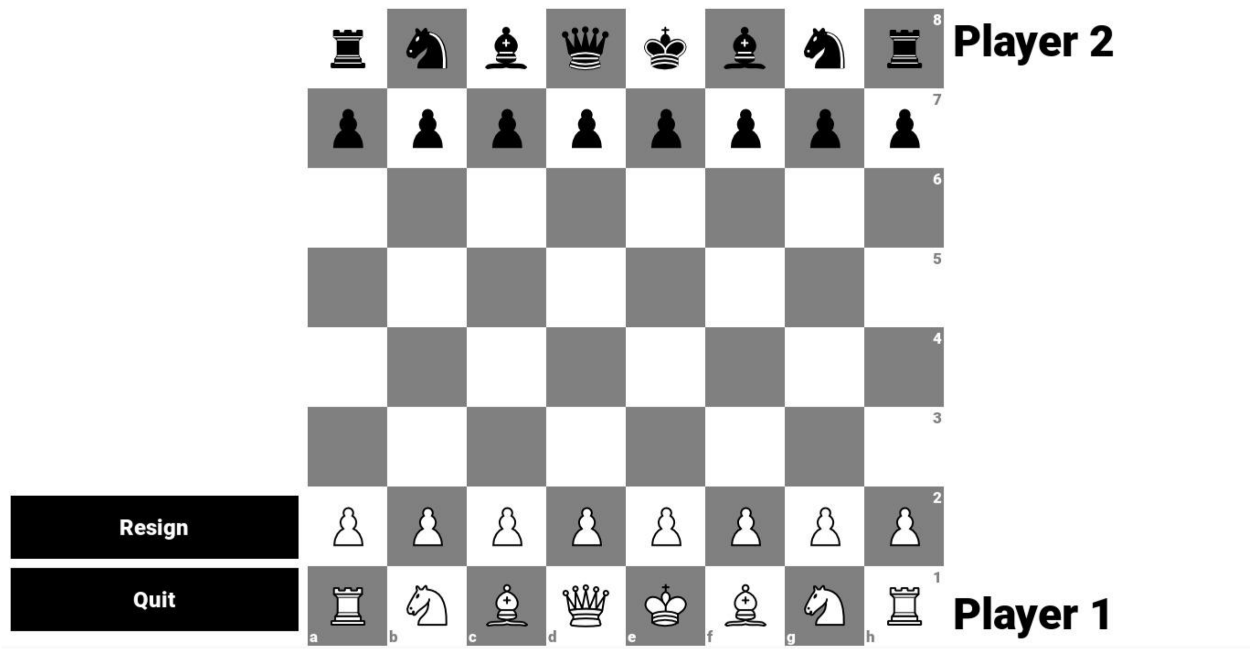


Fig: Starting position

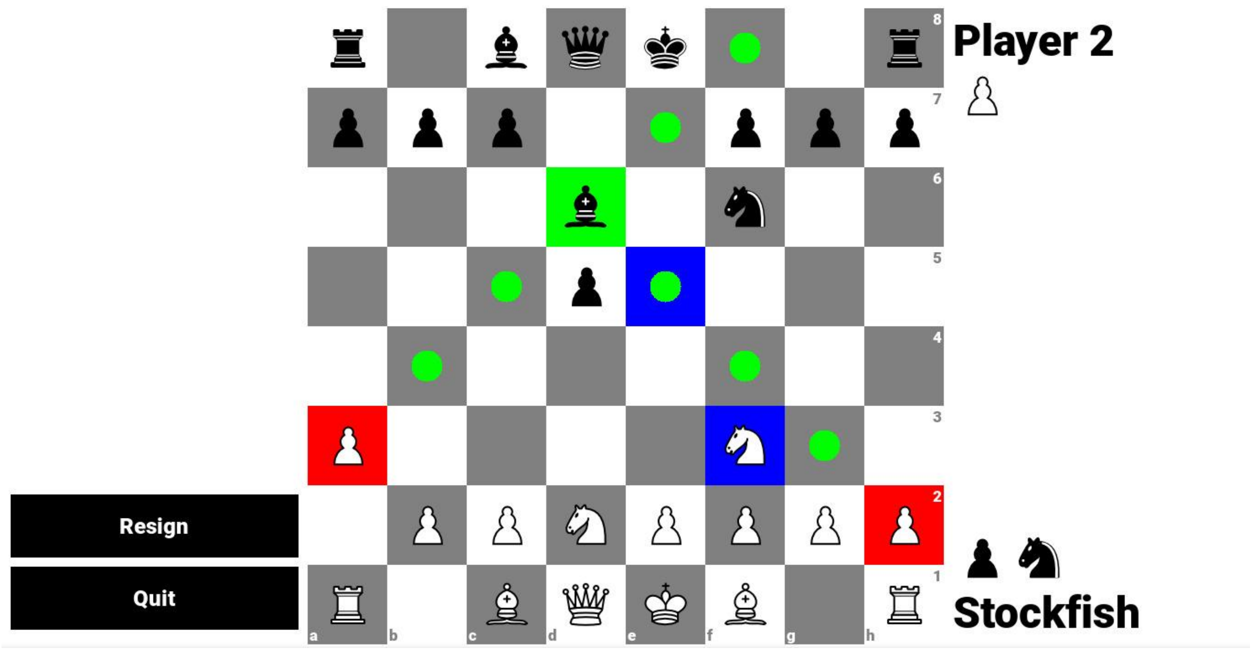


Fig: Random position in a game

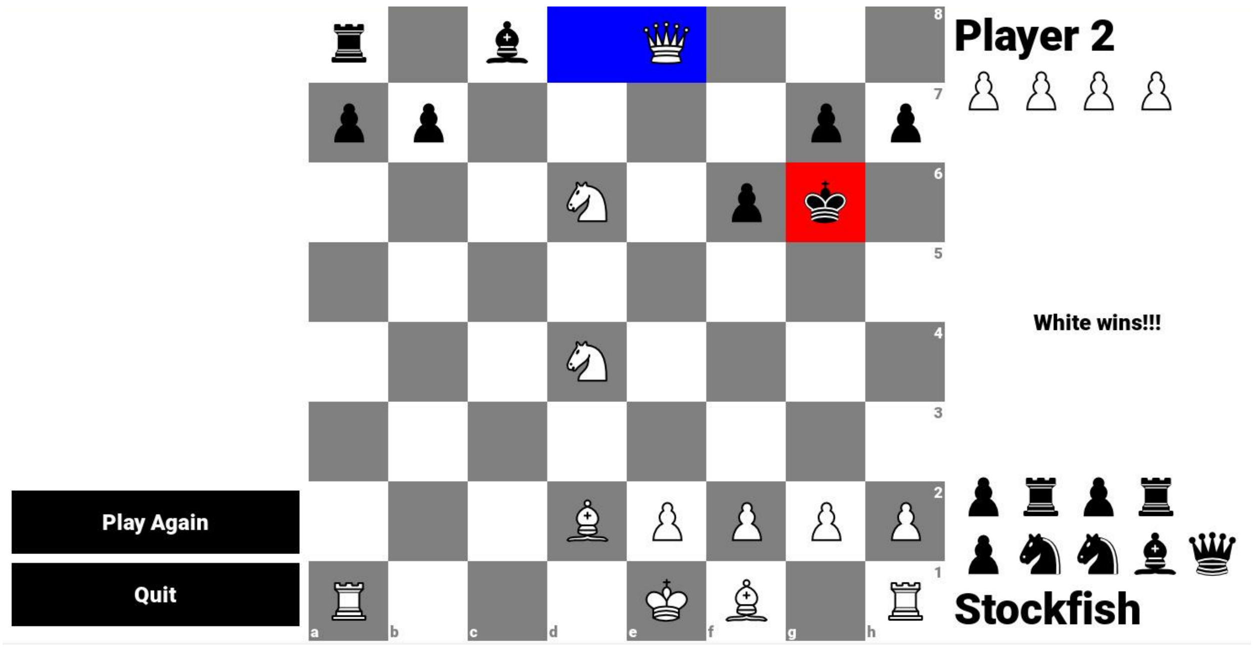


Fig: Checkmate

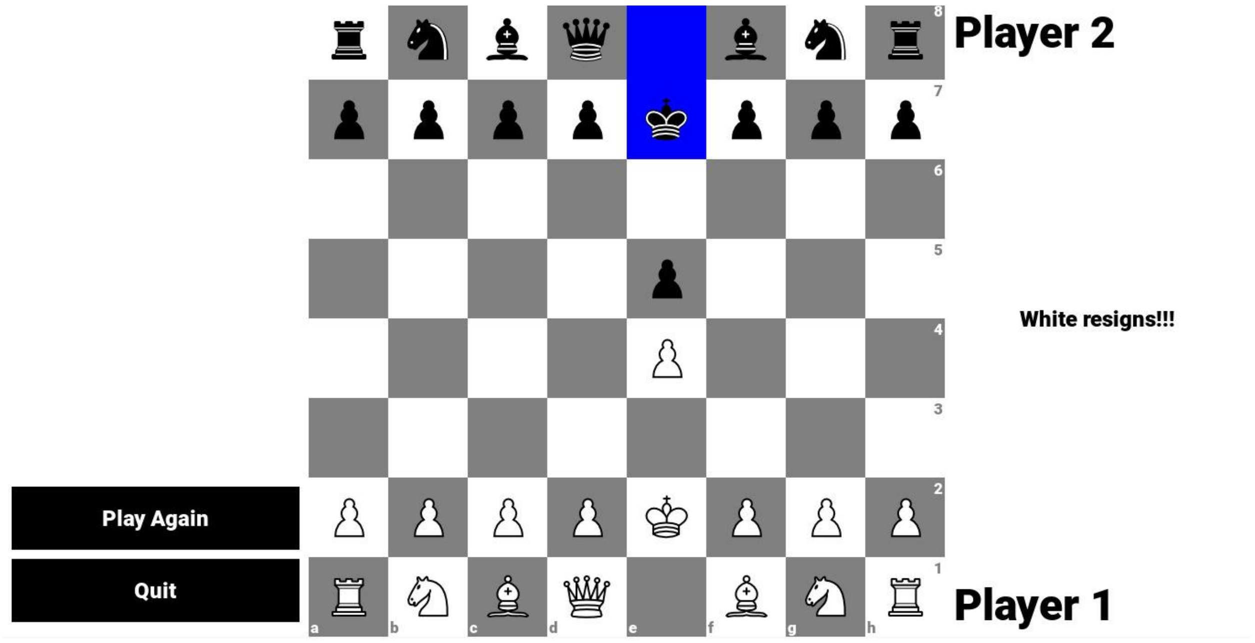


Fig: Resign

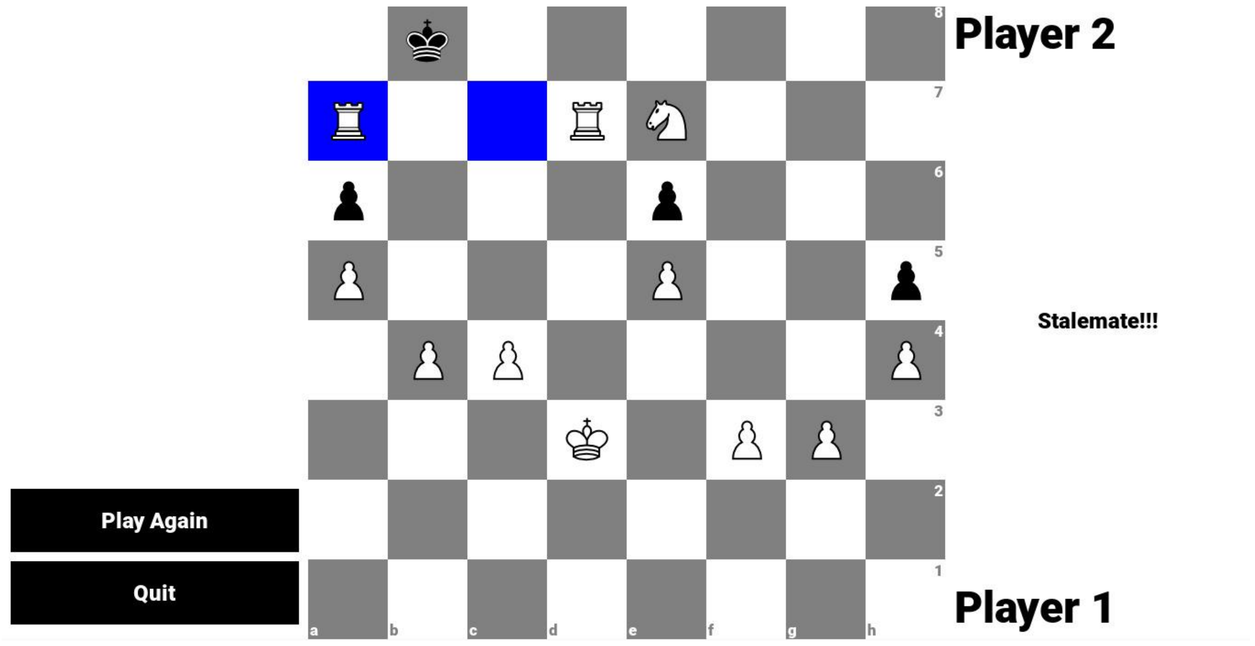


Fig: Stalemate

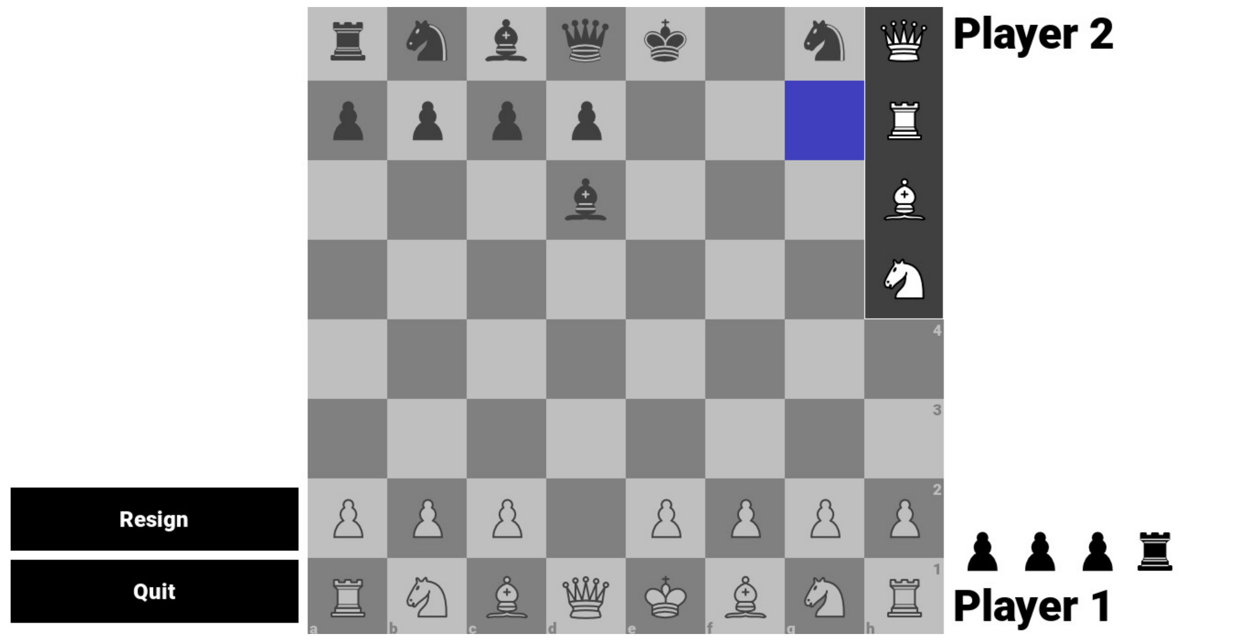


Fig: Promotion piece selection

PROBLEMS FACED AND SOLUTIONS

We faced a number of problems while creating this game.

Scenes management

Our program consists of multiple "scenes", each with different initialization, rendering, updating and event handling codes. So, we decided to use *std::stack* to store all the scenes in the game. The top element of the stack is the current scene.

Fixing memory leaks and crashes

Since, C++ requires us to manage our memory ourselves, unintended memory leaks were bound to occur at some point because of our inexperience. These memory leaks included forgetting to delete SDL textures, Mix chunks, and other objects that were allocated on the heap.

We fixed leaks due to forgetting to deallocate SDL textures by creating a wrapper class ***Textures*** which handles allocating and deleting SDL textures. Similarly, we created a ***Sound*** class to handle sounds used in the game.

We experienced random crashes with little to no indication of what had caused it. So, we created an ***Error*** class which consisted of a number of common issues we had encountered previously and used *throw* blocks in places where such errors could occur. This helped in speeding the process of debugging significantly as we had a general idea of what caused the error. We also used a debugger to set breakpoints in the code to go through the program step-by-step to help in finding bugs due to use of uninitialized variables.

Speed and optimization

In the later half of our project, we realized how crucial speed and optimizations are. But it was too late to overhaul our entire structure and try a different approach. The problem with our method was that all the processes were very brute force in nature and we were copying too many things.

1. We checked for checks in a brute force manner, that is, we ran a check for each move by the opponent to see if they could attack the king.

2. Filtering legal moves from pseudo illegal moves was also quite inefficient as it would look for checks even when there was no possibility of a given one, and a lot of the checks were redundant.
3. Filtering legal moves consists of creating a copy of the board, performing the move and checking for checks. Creating a copy of the board is slow.

We could have used a stack to store moves and performing moves and moves could just have been a push and pop operation which would eliminate the need to copy the board every time we have to check if a move is legal.

LIMITATIONS AND FUTURE ENHANCEMENTS

Even though a lot of work was put onto it, to make it as good as possible, there still are some limitations to the game. Some of these limitation include:

1. No take back of moves.
2. No online multiplayer, which would make it really fun to play with other friends.
3. Users cannot perform any action in single player mode as we have to wait for stockfish to give the move. We have a timeout to prevent stockfish from taking too much time to generate a move but even with that, stockfish sometimes takes a long time to generate a move.
4. The single player mode is fixed at a certain difficulty level(stockfish level 10) which is difficult to beat for beginners.

The possible future enhancements are as follows.

1. A multiplayer mode, in which two people from any place can play with each other.
2. An overhaul in the way we store the state of the game with the help of stack to track moves. This would allow for move take backs and will make it so that we don't need to copy the state of the board again and again to make changes.
3. Option for users to set stockfish level in single player mode.
4. Use bitboards to represent the board.

<u>CONCLUSION</u>

In this way, we completed our project. The project was a great learning experience for our group. We learned about the library SDL2, features of c++ and most importantly the object oriented paradigm. We learned how powerful the object oriented paradigm is, with the help of key concepts like inheritance, polymorphism, encapsulation and abstraction. We were able to create interfaces that made the coding experience much cleaner.

We also learned a great deal about collaboration and teamwork from the project. We learned to use various industry tools such as Github for version control of our project and Lucidcharts for making flowcharts.

The project taught us a great deal about the development cycle, including planning, analysis, development, testing and debugging. We learned the power of Object Orientated Programming paradigm in making efficient software.

REFERENCES

- https://www.chessprogramming.org
- https://backscattering.de/chess/uci/
- https://disservin.github.io/stockfish-docs/pages/Commands.html
- https://lazyfoo.net/tutorials/SDL
- https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt
- https://github.com/Disservin/fast-process-communication