TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

HEADBALL - A GAME OF HEADERS

A COURSE PROJECT SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE PRACTICAL COURSE ON OBJECT ORIENTED PROGRAMMING [CT 451]

Submitted by:
Prajwol Pradhan (PUL076BEI023)
Rujal Acharya (PUL076BEI029)
Sanjay K.C. (PUL076BEI038)
Asmin Silwal (PUL076BEI040)

Submitted to:
Department of Electronics and Computer Engineering
Pulchowk Campus, Institute of Engineering,
Tribhuvan University
Lalitpur, Nepal

December, 2020

Abstract

This project is about making a game called HeadBall in C++ programming language based on Object Oriented Programming (OOP) approach. Headball is a simple cross-platform, multiplayer game played for fun that exist only on digital platform. The sole purpose of creating this game is for learning game development in C++ with OOP approach and for partial fulfillment of the requirements for the practical course on Object Oriented Programming. For this purpose Simple and Fast Multimedia Library (SFML) is used for rendering windows and different objects in screen and Box2D is used as physics library to create realistic world in game. Both of these libraries are easily accessible in the web and can be used by reading the documentation. All the codes were written using Visual Studio Code, an open source code editor. Although the game is fully functioning, there are still rooms for improvement and new features can be added to make game better and entertaining. This game is recommended for players of any age group and the game with source code can be downloaded from github.

***Key Words**: Box2D, HeadBall, OOP, SFML*

Acknowledgements

We would like to thank every individual who helped and supported us in having this project completed, through their direct or indirect involvement. We would like to extend our deep gratitude to the Department of Electronics and Computer Engineering for the opportunity we were provided to apply our theoretical knowledge into practical form with this project. We would also like to thank our course instructor **Mr. Bikal Adhikari** for making us familiar with the theoretical aspect of C++ programming language and its Object Oriented approach, and also for providing us valuable guidelines in this project.

Besides, we are grateful to our colleagues of BEI 076 for their cooperation and feedback when we needed. It is not easy to start a project from scratch, especially when the person is a beginner. Hence, we cannot forget to acknowledge the people whose work we have referenced and taken invaluable lessons from in every step of the project, be it while learning some materials or while writing the code.

Lastly, we are indebted to our family members for their understanding and support that we have received not just during the course of our project but our studies as well.

Table of Contents

List of Figures

List of Tables

Abbreviations

- 2D: 2-Dimensional/2-Dimension
- API: Application Programming Interface
- BEI: Bachelors in Electronics, Communication and Information Engineering
- FTP: File Transfer Protocol
- GUI: Graphical User Interface
- HTTP: Hyper Text Transfer Protocol
- IOE: Institute of Engineering
- LAN: Local Area Network
- OOP: Object Oriented Programming
- OpenGL: Open Graphics Library
- OS: Operating System
- PPM: Pixels Per Meter
- SFML: Simple and Fast Multimedia Library
- TCP: Transmission Control Protocol
- UDP: User Datagram Protocol
- UML: Unified Modeling Language
- UNIX: Uniplexed Information Computing System
- VSCode: Visual Studio Code

Chapter 1

INTRODUCTION

This project is based on making a game named **HeadBall**, which is just like the famous sport football but played by heading the ball. **HeadBall** is a simple, open-source multiplayer game playable in the same machine where two players take control of two characters. They try to move the players in such a way that they can maneuver a ball resulting the ball to enter the opposition's goal post. They can simply collide with the ball resulting the ball to bounce off or kick the ball which makes the ball move with a greater force. After the time limit is exceeded, the player with the higher number of goals wins. In case the scores tie, the game is declared a draw. The players can also change the character controls as per their requirement and also make other adjustments flexibly in the game as needed. The game is made in C++ with an attempt to apply OOP concepts wherever applicable. Some open source libraries like SFML (for graphics rendering) and Box2D (as a physics engine) are also being used in the project. The engine behind the game was inspired by a YouTube tutorial series: *Flappy Bird SFML Tutorial Series*[1], where a Flappy Bird clone is made using SFML.

1 Background and problem statements

1.1 Background

Object Oriented Programming (OOP) is a modern programming paradigm where a complex problem is broken down into smaller and simpler terms known as classes and objects. Using this paradigm, complex applications can be created in a more systematic way by focusing only on a particular part of a problem at once. Similarly the use of other OOP features like inheritance, polymorphism and encapsulation makes the process feel more natural as the classes and objects in the program can be compared with real time objects. Considering the fact, the students of BE I076 were assigned a project to make

a real world application making the use of OOP concepts in C++, as a requirement for the completion of course of Object Oriented Programming (CT-451).

Game Development is a highly growing and fascinating field. A fully functional game can be made in any programming language using some programming logic and by making the use of graphics libraries, they can be made attractive and pleasing to the eyes. We also wanted to use the libraries available in C++ and make a fully functioning game out of it. As the process of game development using the existing libraries in C++ was highly based on the OOP paradigm, the game was decided to be made as the project for the completion of course.

1.2 Problem Statements

1. How can OOP paradigm be applied to create a real world application?
2. How can different open source libraries be utilized in order to enhance the project?
3. How does game development cycle work and how can a simple and fun game be made using OOP concepts of C++?
4. How can graphics rendering be done in C++ using simple libraries?
5. What are the ways of creating a realistic world in a game?
6. How do animation and sound effects work in game?
7. How can the project be organized and well documented?
8. How can an intuitive and easy to navigate Graphical User Interface (GUI) be implemented?

2 Objectives

1. To be familiar with basic OOP concepts and its implementation in a real world application.
2. To be familiar about game development and game design.
3. To learn about different open source libraries and also implement those libraries to enhance the project.
4. To make contribution to open source software project.
5. To 1earn about animation of different characters and ways it can be done.
6. To make a fully functional HeadBall game which is fun and entertaining to play and pretty to look at with a beautiful GUI.

7. To gain more experience on group cohesion and teamwork so that it can be applied in a real world scenario.
8. To get familiar with version control with git and collaboration tools like GitHub and VSCode Live Share.
9. To get familiar with industry-standard document typesetting tool such as LaTeX and documentation tool like Doxygen.

Chapter 2

PROBLEM ANALYSIS

1 Understanding the problem

In the initial days of the project, multiple meetings were held between all the group members to discuss about the topic to work on for the project. After multiple brainstorming sessions, it was decided that the game HeadBall will be made, which is a simple football type game where two characters play against each other in a small field. But this decision gave rise to some further confusions and questions. Some of them are as follows:

1. How is it possible to make a simple football type game in C++ using the OOP concepts?
2. What are the different libraries available in C++ for graphics rendering and game development? And which one of them is best suited for our need?
3. Are there libraries available in C++ for physics simulation for games and other projects?
4. How can the game be made fun to play and pretty to look at?
5. How can the GUI aspect be made beautiful and easy to use?

2 Input Requirements

In the game, there are two players that are to be controlled with a keyboard. They need to be able to move freely and also be able to kick a ball that is moving in the environment. Similarly, there need to be buttons where the player can click in order to change from one state/screen to the another (like from game state to paused state, i.e. pausing the game).

3 Output Requirements

The output of the project is a fully functional game where two players can play a simple football game. It should also have a nice menu screen, pause screen where the whole game play pauses, a timer to record the game time and hence declare the half time and full time after certain time passes and also a nice game over screen where the winner of the game is declared. An about screen and instructions screen is also needed for the information about the developers and the information about character control respectively.

4 Processing Requirements

As the project is developed entirely on 64-bit Linux systems, it is highly recommended to use 64-bit Linux machines or any UNIX based machines (like MacOS). But as the libraries used are cross platform in nature, the users can install the necessary libraries (SFML and Box2D) and build the game themselves without any change in the code in Windows or any other operating system. In such a case, the game works fine without any problems whatsoever.

5 Technical Feasibility

The project is made feasible by the use of two different open source libraries, SFML and Box2D. SFML is an open source framework that can be used to create 2D graphics. Similarly, Box2D is an open source physics engine that is used to simulate objects to give them real life behaviour. After getting some knowledge about the libraries and also some knowledge about the OOP concepts, the project is feasible to be made by a group of 4 students at a time span of about one and a half months.

Chapter 3

REVIEW OF RELATED LITERATURES

1 Basic OOP Concepts

1.1 Introduction

C++, as we all know is an extension to 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, multiparadigm, 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, rendering this language unworthy of being called completely Object Oriented.

1.2 Structure of Code

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

- **Standard Libraries Section**
- **Class Definition Section**
- **Functions Definition Section**
- **Main Function Section**

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

```cpp
#include <iostream>
#include <string>

//using namespace std;
class PrintText{
 public:
    PrintText(std::string);
}

PrintText::PrintText(std::string txt){
    std::cout << txt << std::endl;
}

int main() {
  PrintText text("Hello World!");
  return 0;
}
```

Standard Libraries Section

```cpp
#include <iostream>
#include <string>
//using namespace std;
```

- `#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 header file.
- The file `<iostream>` is merely for input-output streams. This header file contains code for console input and output operations. This file is a part of `std` namespace.
- The other header file `<string>` is used for string related functions and is also a part of `std` namespace.
- `namespace` is a prefix that is applied to all the names in a certain set. For example, `iostream` file is defined in a set what we call `std` and it defines two names used in this program - `cout` and `endl`.
- If `using namespace std;` is used, the compiler understands that the names like `cout` and `endl` of the `std` namespace are being used.

Class Definition Section

```
class PrintText{
 public:
    PrintText(string);
}
```

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 above program `PrintText` is the class and `text` is its object.

User-defined Function Definition Section

```
PrintText::PrintText(std::string txt){
    std::cout << txt << std::endl;
}
```

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 the function `PrintText` is a special member function called *constructor* of the class `PrintText`.

Main Function Section

```
    int main() {
        PrintText text("Hello World!");
        return 0;
    }
```

`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.

1.3 Features Of OOP

C++ is a **general-purpose** programming language that was developed as an enhancement of the C language to include object-oriented paradigm [2]. It is an imperative and a compiled language. Some of its main features are:

- **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.

  The following example shows a namespace declaration and method to access the member of the namespace from outside it.

```
namespace ContosoData{
class ObjectManager{
public:
    void DoSomething() {}
};
}
```

  The most convenient method to access its member is:

```
ContosoData::Objectmanager mgr;
mgr.DoSomething();
```

- **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 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 derived_class :: visibility-mode base_class
{
// body of the 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

Table 3.1: *Visibility modes*

- **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.

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

  - Compile time Polymorphism
  - Runtime Polymorphism

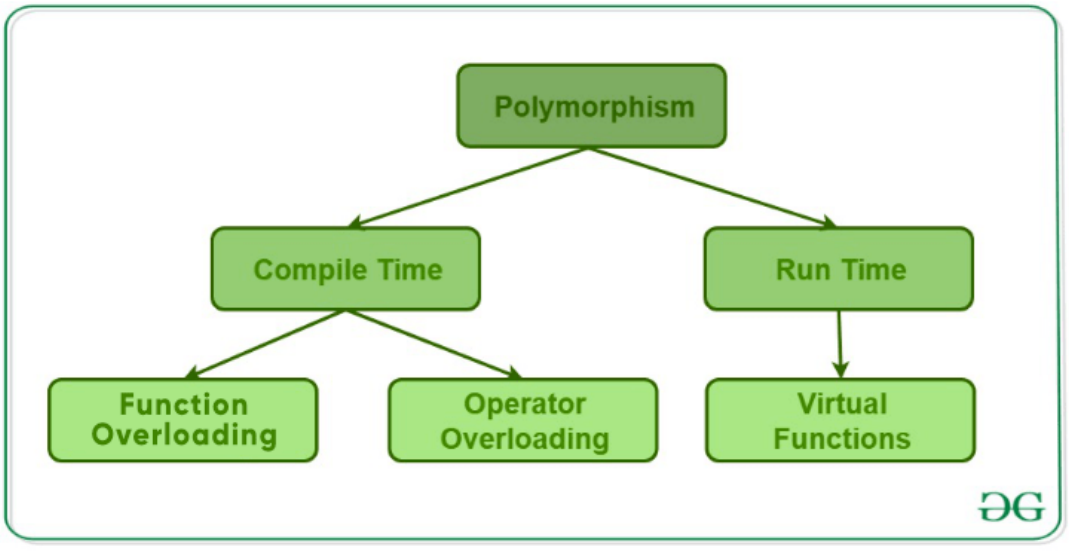

fig. 3.1: *Types of Polymorphism*

Source: GeeksForGeeks

1. **Compile time Polymorphism:** This type of polymorphism is achieved by function overloading or operator overloading.

– **Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

```
#include <iostream>
class FunOver {
    public:
    // function with 1 int parameter
    void func(int x) {
        std::cout << "value of x is " << x << endl;
    }

    // function with same name but 1 double
    ↪  parameter
    void func(double x) {
        std::cout << "value of x is " << x << endl;
    }

    // function with same name and 2 int parameters
    void func(int x, int y) {
        std::cout << "value of x and y is " << x <<
        ↪  ", " << y << endl;
    }
};

int main() {
    FunOver obj1;
    // Which function is called will depend on the
    //parameters passed
    obj1.func(7); // The first 'func' is called
    obj1.func(9.132); // The second 'func' is called
    obj1.func(85,64); // The third 'func' is called
    return 0;
}
Output:
```

In the above example, a single function named func acts differently in three different situations which is the property of polymorphism.

– **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

```
#include<iostream>
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {
        real = r;   imag = i;
    }  // This is automatically called when '+' is
     ↪  used between two Complex objects

    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() {
        std::cout << real << " + i" << imag << endl;
    }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to
                          //"operator+"
    c3.print();
    return 0;
}
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating

point) but here the operator is made to perform addition of two imaginary or complex numbers.

2. **Runtime Overload:** This type of polymorphism is achieved by Function Overriding.

   – **Function Overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
#include <iostream>
class base {
public:
    virtual void print () {
        std::cout << "print base class" << std::endl;
        ↪ }
    void show () {
        std::cout << "show base class" << std::endl;
        ↪ }
};

class derived:public base {
public:
    void print () //print () is already virtual
    ↪ function in derived class, we could also
    ↪ declared as virtual void print () explicitly
    { std::cout << "print derived class" <<
    ↪ std::endl; }

    void show ()
    { std::cout << "show derived class" << std::endl;
    ↪ }
};

//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
```

```
    //virtual function, binded at runtime (Runtime
    ↪ polymorphism)
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is print derived class as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is show base class as pointer is of base type).

- **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 keyword 'class'.

2 Smart Pointer

2.1 Introduction

Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced). They are the classes wrapped around a normal pointer where the normal

pointer operators like * and -> are overloaded. Using smart pointers we can make pointers work in a way that we do not need to explicitly call delete. It is a feature in C++ which is influenced by Simula's approach of memory management[3]. There are different types of smart pointers that are used for different purposes. Following are the different types of smart pointers used in this project:

- Unique Pointer (`unique_ptr`)
- Shared Pointer (`shared_ptr`)

2.2 Types

2.2.1 Unique Pointer (`unique_ptr`)

Unique pointer is a smart pointer that was introduced in C++11 and is defined in the header `<memory>`. Unique pointer makes sure that only one copy of the pointer is possible at a given state. Unique pointer makes sure that only one pointer can point to the object at a time. It prevents the copy of the contained pointer as copy constructor and assignment operators are explicitly removed[4]. But `std::move` can be used to transfer the ownership from one pointer to another. After the moving of the pointer is done, the previous pointer cannot make reference to the object. The following program is an example of the use of `unique_ptr`[5]:

```
std::unique_ptr<int> p1(new int(5));

std::unique_ptr<int> p2 = p1;  // Compile error.

std::unique_ptr<int> p3 = std::move(p1);
// Transfers ownership. p3 now owns the memory and p1 is set to
↪  nullptr.

p3.reset();  // Deletes the memory.
p1.reset();  // Does nothing.
```

2.2.2 Shared Pointer (`shared_ptr`)

Shared pointer is also a smart pointer that was introduced in C++11 and is defined in the header `<memory>`. The shared pointer is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function. This is particularly useful in the context of OOP, to store a pointer as a member variable and return it to access the referenced value outside the scope of the class. While `shared_ptr` is used,

it also maintains a reference counter to count the number of pointers that refers to the same object. C++11 also introduces `std::make_shared( )` method to safely conduct the dynamic allocation the memory to the shared pointer. The following program is an example of the use of `shared_ptr`[5]:

```
std::shared_ptr<int> p0 = std::make_shared<int> (5);
// Valid, allocates 1 integer and initialize it with value 5.

std::shared_ptr<int[]> p1(new int[5]);
// Valid, allocates 5 integers.

std::shared_ptr<int[]> p2 = p1;  // Both now own the memory.

p1.reset();  // Memory still exists, due to p2.
p2.reset();  // Frees the memory, since no one else owns the
↪  memory
```

3 SFML

3.1 Introduction

SFML stands for Simple and Fast Multimedia Library. This is a framework written in C++ and is based on OpenGL for its graphical rendering part. SFML is a library, which speeds up and eases the process of developing applications that rely on extensive use of media content, such as video, text, still images, audio, and animation for interactivity. It provides an easy to use application programming interface (API), compiles and runs out of the box on Windows, Linux, and Mac OS X, and is supported by multiple languages, such as C, .NET, C++, Java, Ruby, Python etc. It is also open source, so one can always go and look at the source code at `https://github.com/SFML/SFML`.

SFML is composed of five modules, which are independent of one another and can be included when needed:

- **System:** This is the main module, and is required by all the others. It provides features like clocks, threads, and two or three dimensions with all their logics (mathematics operations).
- **Window:** This module provides a means of creating and managing a window, gathering user input and events, as well as using SFML alongside OpenGL.
- **Graphics:** The graphics module deals with two-dimensional graphical rendering

part. This is used to draw shapes and load images(textures) into sprites and draw them on the screen.

- **Audio:** This module handles any operation related to playing music, sounds, audio streams, or recording audio.
- **Network:** This module simplifies the use of sockets like TCP and UDP for internet-based transmission of data packets. It also eases the implementation of protocols like HTTP and FTP.

3.2 Structure of SFML-based Code

A simple example for SFML:

```
#include <SFML/Graphics.hpp>
int main()
{
sf::RenderWindow window(sf::VideoMode(400, 400),"Simple Example");
window.setFramerateLimit(60);
//create a circle
sf::CircleShape circle(150);
circle.setFillColor(sf::Color::Blue);
circle.setPosition(10, 20);
//game loop
while (window.isOpen())
{
//manage the events
sf::Event event;
while(window.pollEvent(event))
{
    if ((event.type == sf::Event::Closed) || (event.type ==
    ↪  sf::Event::KeyPressed && event.key.code
    ↪  ==sf::Keyboard::Escape))
        window.close(); //close the window
}
window.clear(); //clear the window, default color   is black
window.draw(circle); //draw the circle
window.display(); //display the result on screen
}
return 0;
```

```
}
```

It can be better understood with a simple data-flow diagram of SFML .

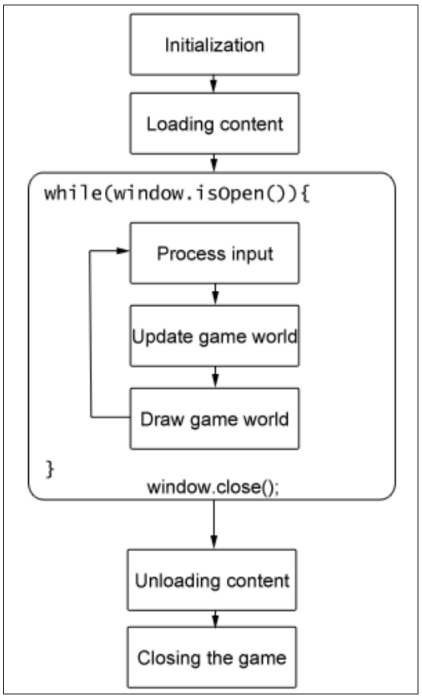

fig. 3.2: *Basic structure of code in SFML*

Basic SFML Drawing

```
sf::RectangleShape rectangle(sf::Vector2f(128.0f,128.0f));
rectangle.setFillColor(sf::Color::Red);
rectangle.setPosition(320,240);
```

sf::RectangleShape is a derived class of sf::Shape that inherits from sf::Drawable, which is an abstract base class that all entities must inherit from and implement its virtual methods in order to be able to be drawn on screen. It also inherits from sf::Transformable, which provides all the necessary functionality in order to move, scale, and rotate an entity. This relationship allows our rectangle to be transformed, as well as rendered to the screen. In its constructor a new data type: sf::Vector2f is introduced. It's essentially just a struct of two floats, x and y,that represent a point in a two-dimensional universe, not to be confused with the std::vector, which is a data container.

The rectangle constructor takes a single argument of sf::Vector2f which represents the size of the rectangle in pixels and is optional. On the second line, the fill color of the rectangle by providing one of SFML's predefined . Lastly, the position of our shape

by calling the setPosition method and passing its position in pixels alongside the x and y axis, which in this case is the centre of our window. There is only one more thing missing until we can draw the rectangle:

```
window.draw(rectangle);
```

Drawing Image in SFML

To draw images in SFML, One needs to be familar with two class sf::Texture and sf::Sprite. A texture is simply a graphical image that is located in computer's storage. Any image can be turned into texture by loading it.

```
sf::Texture texture;
texture.loadFromFile("filename.png");
```

Now, the image has been loaded. It is time to turn it into sprite. A sprite, much like the sf::Shape derivatives we have worked with so far, is a sf::Drawable object, which in this case represents a sf::Texture and also supports a list of transformations, both physical and graphical. Think of it as a simple rectangle with a texture applied to it.[6] [7]

Since the image has already been loaded as a texture. Its very easy to create sprite:

```
sf::Sprite sprite(texture);
window.draw(sprite);
```

4 Box2D

4.1 Introduction

Box2D is one of the most popular 2D physics engines. It is light, robust, efficient and highly portable. It has been used by several applications in several platforms.

A physics engine simulates the physics of objects to give them believable real-life movement. Although it can be used for other applications, it is primarily used as a library for games, and games make up the majority of softwares using Box2D. The engine is written and maintained by Erin Catto and is easily accessible at: `https://github.com/erincatto/box2d` and one can visit `https://box2d.org/documentation` for documentation of different features of the engine.

4.2 Core Concepts of Box2D

1. **World:** World manages the physics simulation. It knows everything about the overall coordinate space and also stores lists of every element in the world.
2. **Body:** Body serves as the primary element in the Box2D world. It has a location and also has a velocity.
3. **Shape:** Shape keeps track of all the necessary collision geometry attached to a body.
4. **Fixture:** Fixture attaches a shape to a body and sets properties such as density, friction, and restitution.
5. **Joint:** Joint acts as a connection between two bodies (or between one body and the world itself).

4.3 Creating a world and bodies

To create world, first a gravitational force for the world is created.

```
b2Vec2 gravity(0.0f, -10.0f);
```

Now a world is created:

```
b2World world(gravity);
```

Bodies are made by first setting up a definition, and then using this to create the body object itself.

```
b2BodyDef myBodyDef;
myBodyDef.type = b2_dynamicBody; //this will be a dynamic body
myBodyDef.position.Set(0, 20); //set the starting position
myBodyDef.angle = 0; //set the starting angle
```

Now, let us use this definition to create the body instance:

```
b2Body* dynamicBody = world->CreateBody(&myBodyDef);
```

Here, the world variable is a b2World object. The world object is like the boss of everything in Box2D, it handles the creation and deletion of physics objects. A body is basically invisible, so if a program like this is built and run, there is nothing to see yet.

To give a body its size, shape, and other tangible characteristics, we add fixtures to it. Also, the default behaviour is that adding fixtures will affect the mass of the body too. A body can have many fixtures attached to it, each fixture added will affect the total mass of the body. For now, let us add one simple fixture to this body, a square, and look a bit further at fixtures in the next topic.

```
b2PolygonShape boxShape;
boxShape.SetAsBox(1,1);
b2FixtureDef boxFixtureDef;
boxFixtureDef.shape = &boxShape;
boxFixtureDef.density = 1;
dynamicBody->CreateFixture(&boxFixtureDef)
```

The body definition must also specify the "type" of body we want to make. There are three possibilities:

- **Dynamic:** This is what we will use most often—a "fully simulated" body. A dynamic body moves around the world, collides with other bodies, and responds to the forces in its environment.
- **Static:** A static body is one that cannot move (as if it had an infinite mass). We'll use static bodies for fixed platforms and boundaries.
- **Kinematic:** A kinematic body can be moved manually by setting its velocity directly. If you have a user-controlled object in your world, you can use a kinematic body. Note that kinematic bodies collide only with dynamic bodies and not with other static or kinematic ones.

Chapter 4

ALGORITHM DEVELOPMENT AND FLOWCHART

1 Algorithm

1.1 Algorithm for State Machine

```
if isAdding ==true:
    if stack is not empty and isReplacing == true:
        pop the top state at stack
    else if stack is not empty:
        pause the top state at stack

     push new state to stack
     init top of the stack

     isAdding = false


if isRemoving==true and stack is not empty:
    pop the top state of stack

    if stack is not empty:
        resume the top state at stack

    isRemoving = false
```

1.2 Algorithm for Splash Screen State

```
load sprites
x = 0
loop{
    x++
    display sprite no x
    if x == 12 go to main menu
}
```

1.3 Algorithm for Main Menu State

```
load sprites
display the sprites
play theme music
loop{
    if play button is clicked or enter is pressed{
        stop theme music
        player1 score = player 2 score = 0
        time = 0
        go to game state
    }
    if about button is clicked:
        go to about screen
    if instructions button is clicked:
        go to instructions screen
    if exit button is clicked:
        exit the game
}
```

1.4 Algorithm for Game State

```
load the sprites
set the position of players and ball
display the sprites for the environment and players
resume time
isSecondHalf = false
loop{
    get input from players for movement
```

```
    if player moves left or right:
        animate the player
        change its position

    if player jumps:
        apply linear velocity  upwards
    if player presses kick button {
        check if ball collides the player
        if player collides the ball {
            play kick sfx
            apply force to the ball at the direction respective to
             ↪  its relative position with the player
        }
    }

    display the score and time

    process the world in Box2D by one step
    process the position of players and ball

    if position of ball lies in the bounds of goal post:
        play goal sfx
        score of respective player += 1
        reposition the player and ball

    if game time >= 45 and isSecondHalf == false:
        pause time
        isSecondHalf = true
        go to half time state

    if game time >= 90:
        go to game over state

    display the ball and players at its respective position
}
```

1.5 Algorithm for Game Over State

```
load sprites
display the sprites

if player 1 score > player 2 score:
    display "Player 1 Wins"
else if player 2 score > player 1 score:
    display "Player 2 Wins"
else:
    display "Match Tied"

loop{
    if replay button is clicked:
        restart timer
        player 1 score = player 2 score = 0
        go to game state

    if exit button is clicked:
        exit the game
}
```

2 Flowchart

2.1 Flowchart for State Machine State

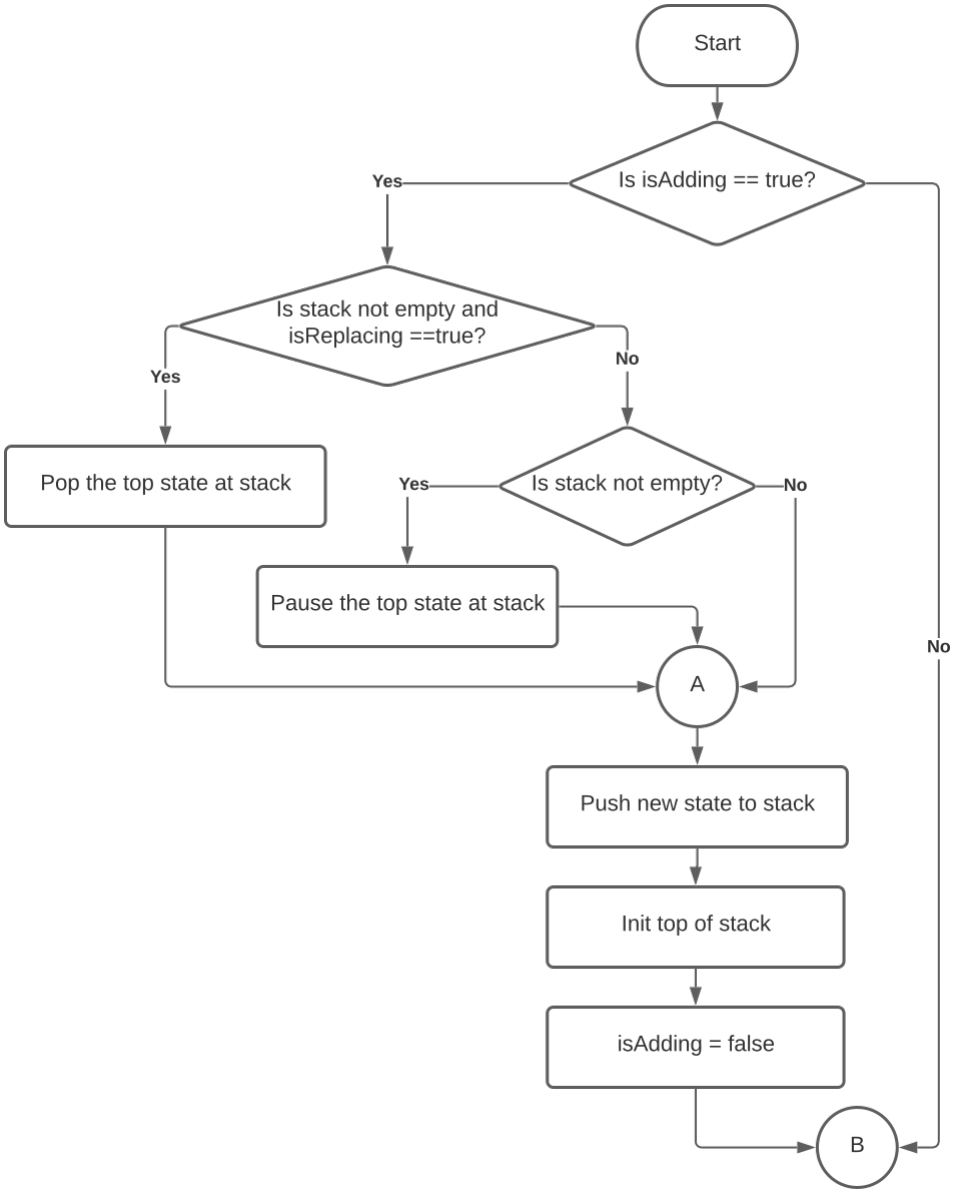

fig. 4.1: *Flowchart for State Machine A*

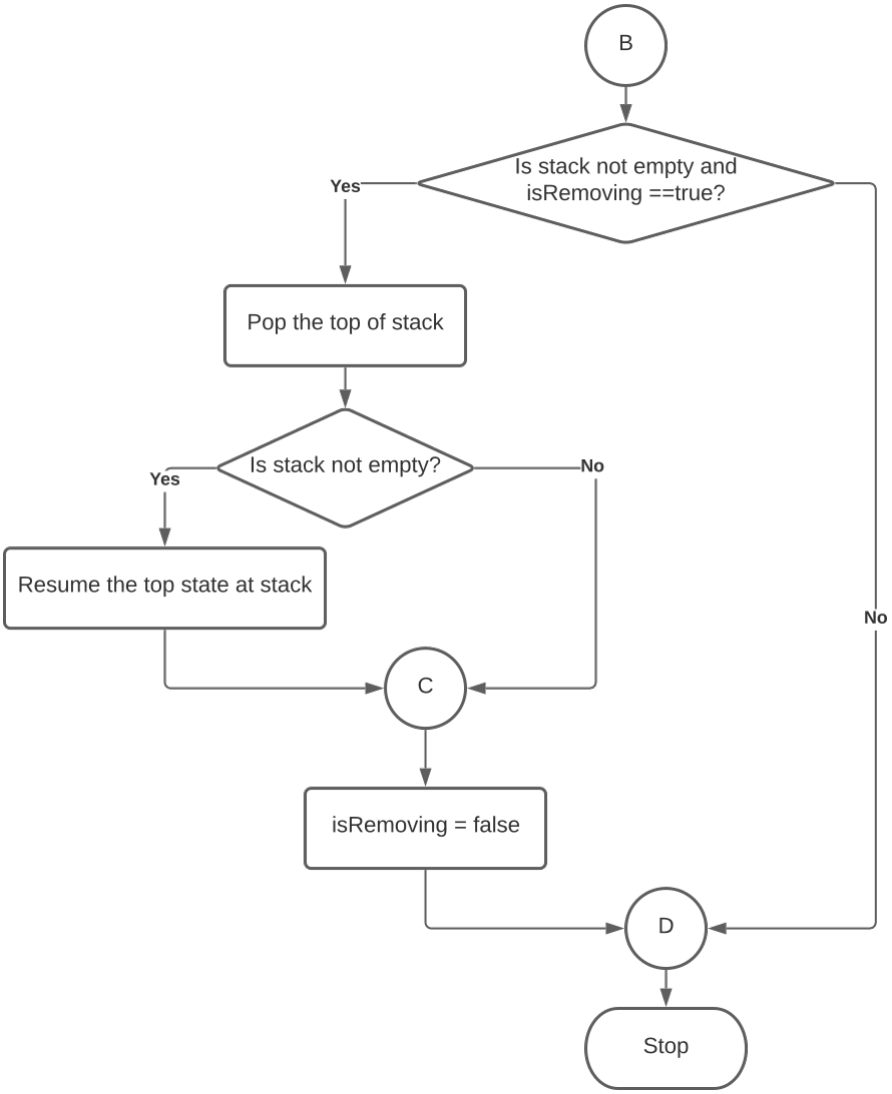

fig. 4.2: *Flowchart for State Machine B*

2.2 Flowchart for Splash Screen

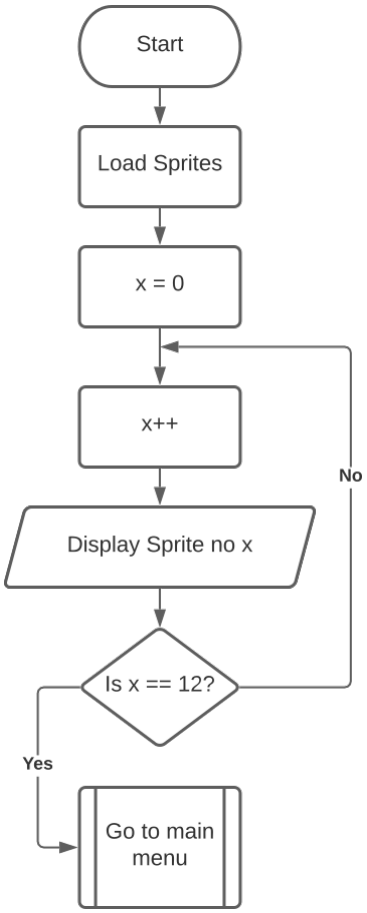

fig. 4.3: *Flowchart for Splash Screen State*

2.3 Flowchart for Main Menu State

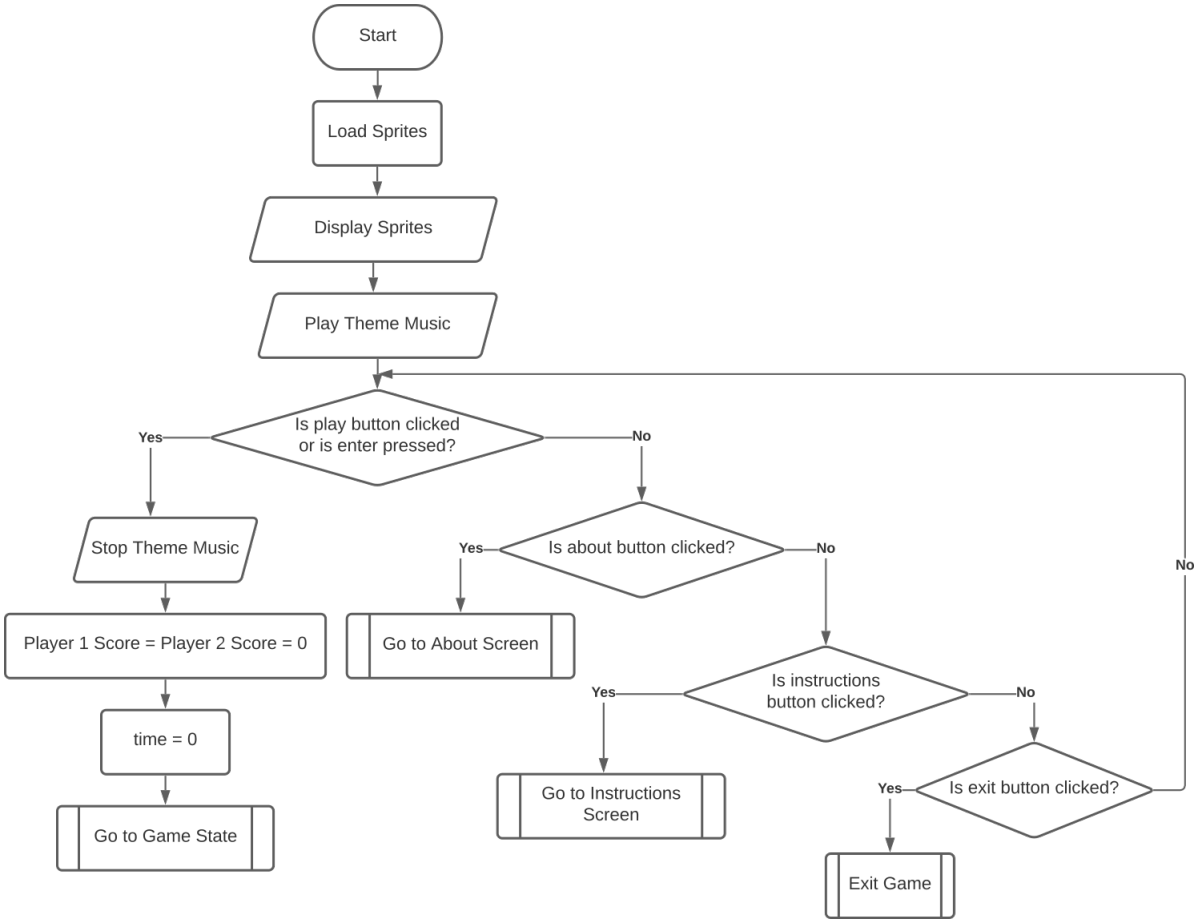

fig. 4.4: *Flowchart for Main Menu State*

2.4 Flowchart for Game State

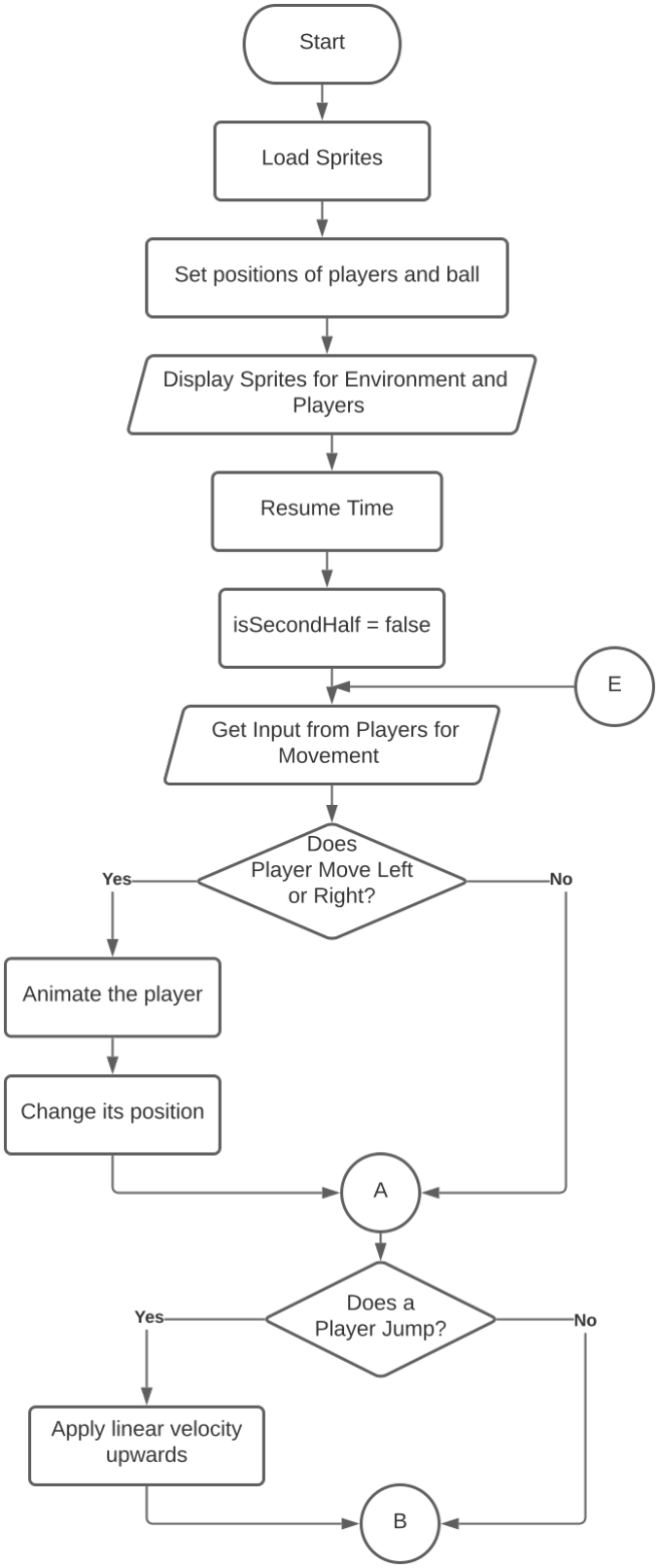

fig. 4.5: *Flowchart for Game State A*

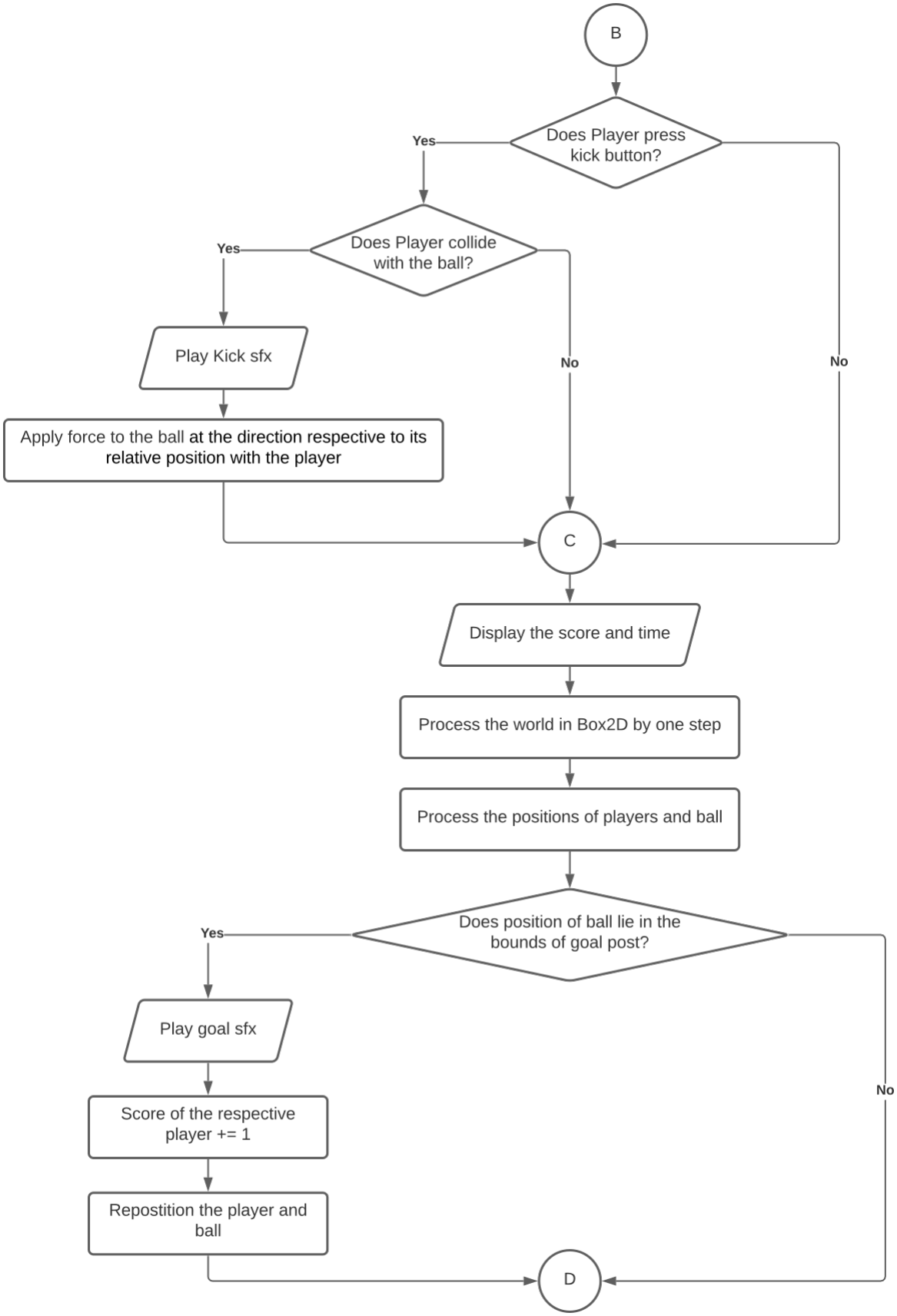

fig. 4.6: *Flowchart for Game State B*

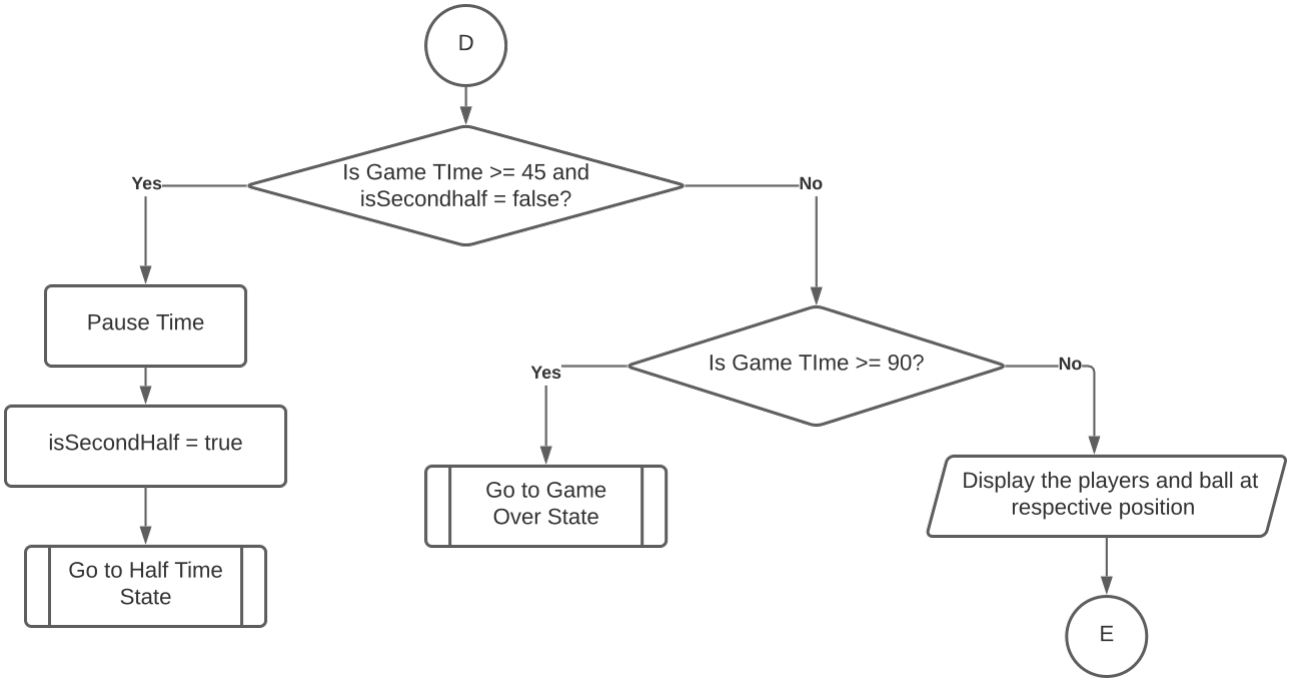

fig. 4.7: *Flowchart for Game State C*

2.5 Flowchart for Game Over State

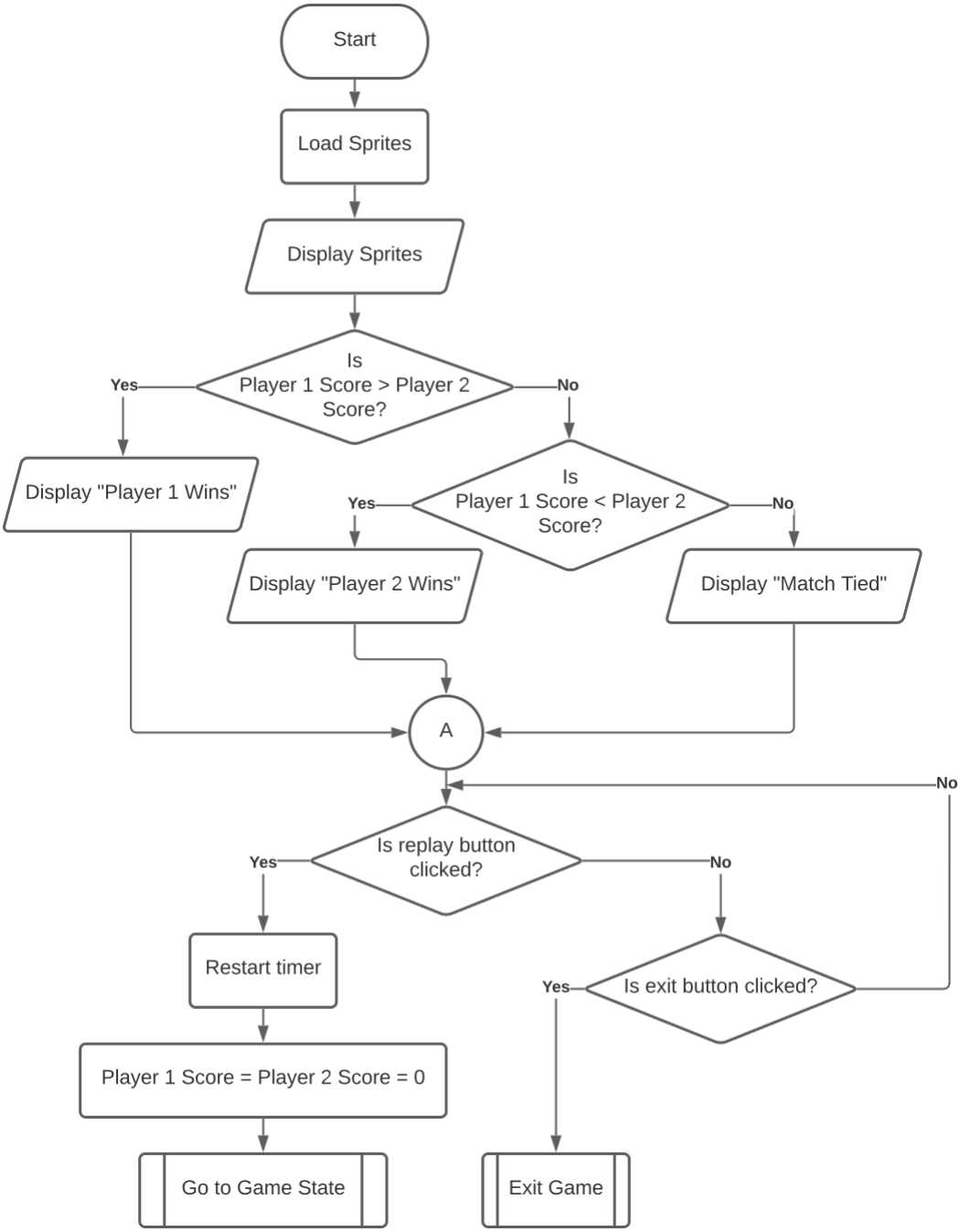

fig. 4.8: *Flowchart of Game Over State*

3 UML Diagrams

3.1 Collaboration diagram for GameState class

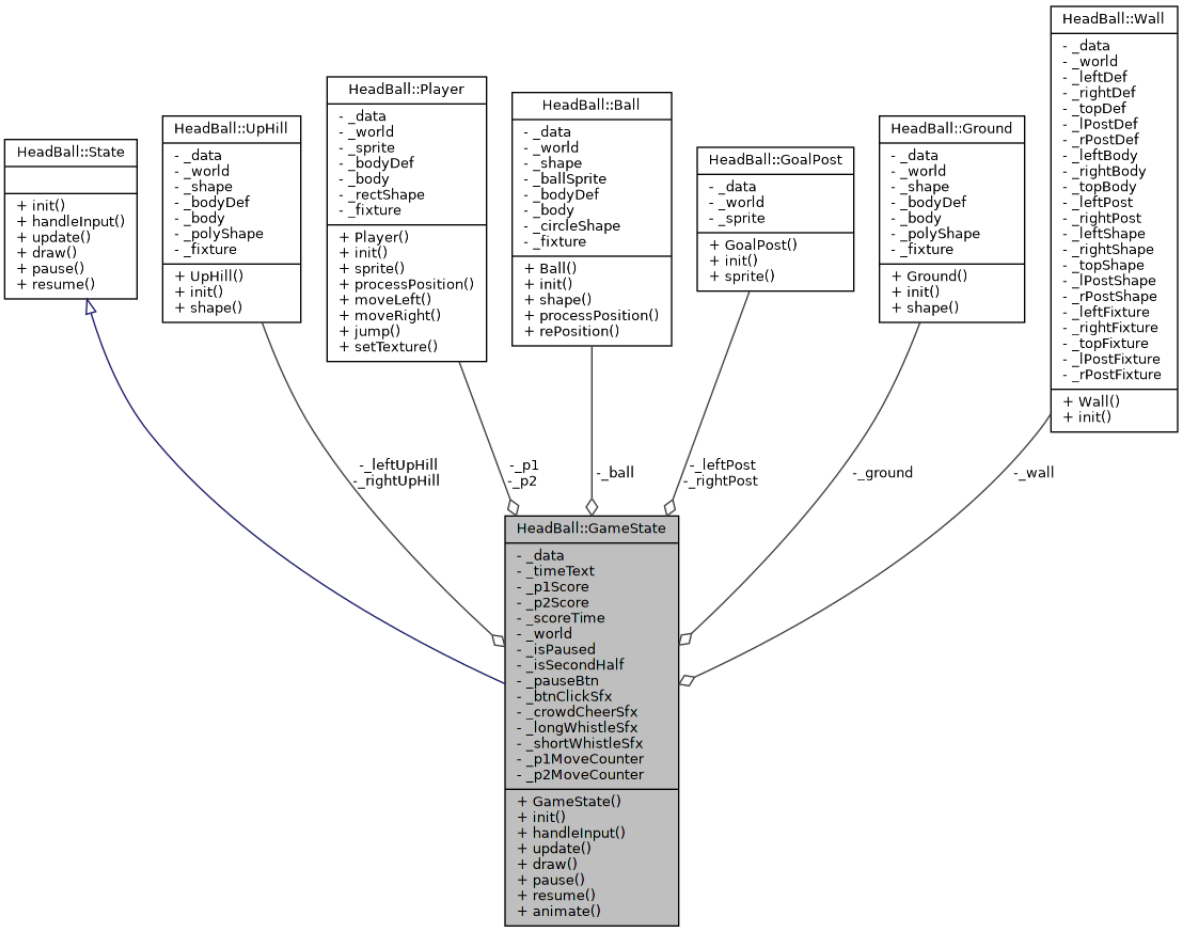

fig. 4.9: *Collaboration diagram for GameState class*

3.2 State Class Hierarchy Diagram

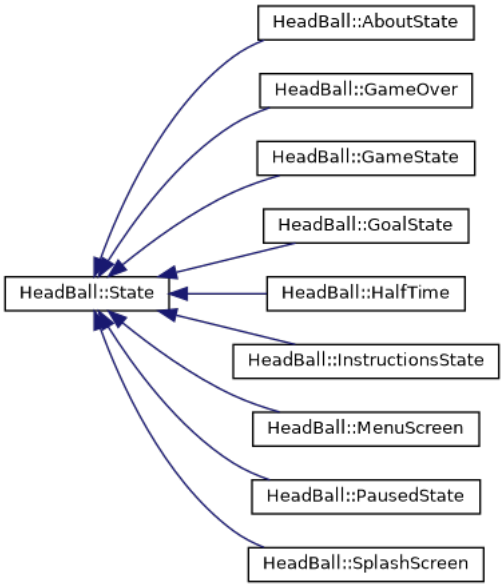

fig. 4.10: *State Class Hierarchy Diagram*

3.3 Inheritance diagram for State Class

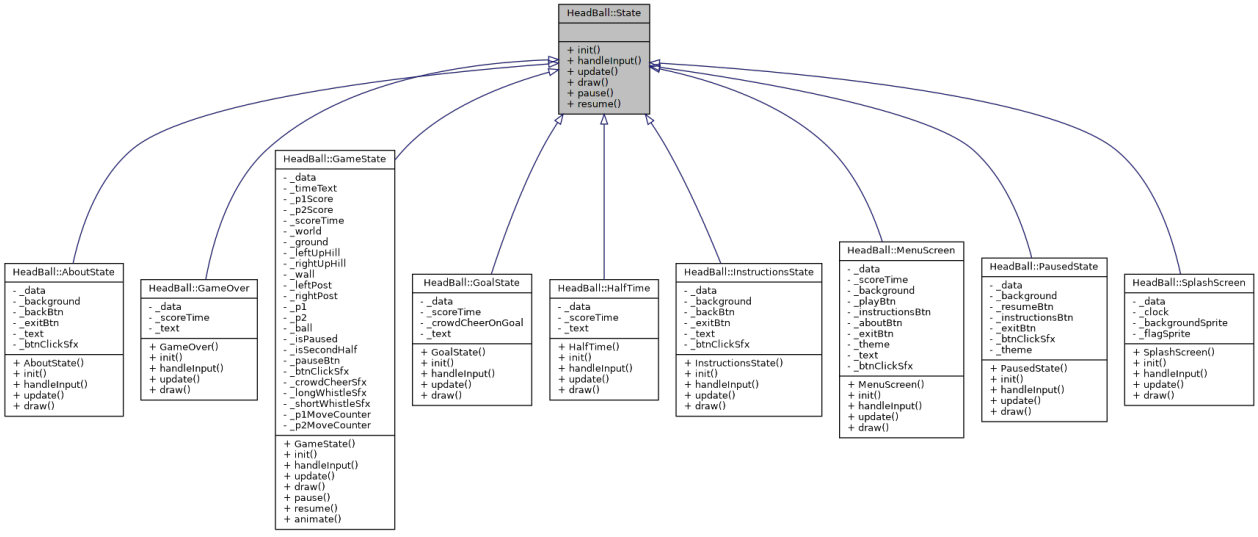

fig. 4.11: *Inheritance diagram for State Class*

Chapter 5

IMPLEMENTATION AND CODING

1 Implementation

The game is built with Object Oriented approach and SFML framework provides the underlying foundation. Box2D library is used for physics simulation with SFML. State machine or simply `State` class is the backbone of the game on top of which all the game elements are built. `State` in simple words represents the various stages of the game. All the states such as `Splash Screen State`, `Menu State`, `Game State`, etc. are derived from the `State` class.

There is `Game` class as well that handles all the types of data that need to be used across different states. It also initiates input handling, data updating and state changing mechanisms among others.

The namespace `HeadBall` wraps all the structures, classes and their members.

1.1 States

As aforementioned, states are the backbone of the game. The whole game is constructed upon the concept of states. When an event is taking place in the game, there is a state making that event happen. When that event is to be suspended for some time, another state reference is pushed onto the stack and when the operation of the event is to be resumed, the corresponding state reference is brought to the top of the stack. The class handling all the stack related operations is the `StateMachine` class.

The `State` class hierarchy showing only the derived class names is given in fig. 5.1.

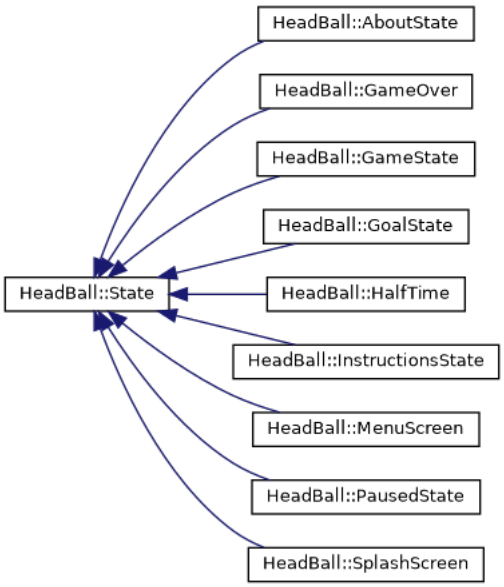

fig. 5.1: *State class hierarchy with derived class names*

Brief descriptions of each of the derived classes of the base class `State` are given below with nestings as necessary, and are arranged in the order their occurrence from the time the game is run:

- SplashScreen State: For the screen to show immediately before the menu screen is shown.
- MenuScreen State: To show menu screen having various menu options.
- Game State: The major state where the gameplay happens.
  - Goal State: To show the screen when a player scores.
  - Halftime State: To show the screen when it is halftime.
  - Paused State: To show the screen when the game is paused.
- GameOver State: To show the screen when it is fulltime or the end of the game.
- Instructions State: For showing the instructions page.
- About State: For showing the about page.

1.2 Resource Managers

Like there are different states to manage various stages of the game, there are a few manager classes that are used for the purpose of efficiently managing the resources and elements. These include:

- AssetManager class: Class for managing assets.
- InputManager class: Class for handling the user inputs through mouse and keyboard.
- TimeManager class: Class for handling time during games.

2 Coding

Here are the explanations of some of the important parts of the project:

2.1 State Management

State management is done by the `StateMachine` class. First of all a `unique_ptr` is defined which points to the different game states. This is made possible by inheriting all the game state classes to a common abstract class `State`. The inheritance diagram for this class is given in fig. 4.11.

The `StateMachine` class contains a stack where the different game states are kept. In every game loop, the state that is at the top of the stack gets resumed. At every game loop, the `processStateChanges` method of the `StateManager` class is called, which manages the transition from one state to another.

```
void StateManager::processStateChanges {
    if (_isRemoving == true && !_states.empty() ) {
        this->_states.pop( );

        if (!_states.empty ( )) {
            this->_states.top( )->resume( );
        }

        this->_isRemoving = false;
    }

    if (_isAdding == true) {
        if (_isReplacing == true && !_states.empty()) {
            this->_states.pop( );
        } else if (!_states.empty()) {
            this->_states.top( )->pause( );
        }
```

```
            this->_states.push(std::move(this->_newState));
            this->_states.top( )->init( );

            this->_isAdding = false;
        }
    }
```

Here, the bool attributes `_isAdding`, `_isRemoving` and `_isReplacing` indicate whether a new state is being added, removed or replaced from the stack. These values are changed during the course of the game loop according to the input given by the user.

- When a state is being removed, first of all it is checked if the stack is empty. If the stack is not empty, it pops the state that is at the top of the stack. Again it checks whether a state is empty. If it is not empty, it resumes the state that is at the top of the stack.
- When a state is being added, first of all it is checked whether it is being replaced or not. If it is being replaced and the stack is not empty, the top state at the stack is popped. And if it is not being popped and the stack is not empty, the top state in the stack is paused. Now, the new state is being pushed in the stack and initiated.

2.2 Input Management

Input management is done by `InputManager` class.In this class, inputs given by the user is checked and a boolean value is returned which then is processed through `Game` class to provide required state output.

```
bool InputManager::isSpriteClicked (sf::Sprite sprite,
↪  sf::Mouse::Button button, sf::RenderWindow& window) {
   if (sf::Mouse::isButtonPressed(button)) {
       sf::IntRect spriteRect (sprite.getPosition().x -
       ↪  sprite.getGlobalBounds().width / 2,
       ↪  sprite.getPosition().y -
       ↪  sprite.getGlobalBounds().height / 2,
       ↪  sprite.getGlobalBounds().width,
       ↪  sprite.getGlobalBounds().height);
       if (spriteRect.contains(sf::Mouse::getPosition(window))) {
           return true;
       }
```

```
    }
    return false;
}

sf::Vector2i InputManager::getMousePosition (sf::RenderWindow&
↪  window) {
    return sf::Mouse::getPosition(window);
}

bool InputManager::isMoving (std::string position, std::string
↪  player) {
    bool isp2 = false;
    if (player == "p2") {
        isp2 = true;
    }

    if (!isp2 && position == "left") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P1_LEFT)) {
            return true;
        }
    }

    else if (!isp2 && position == "right") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P1_RIGHT)) {
            return true;
        }
    }

    else if (isp2 && position == "left") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P2_LEFT)) {
            return true;
        }
    }
```

```
    else if (isp2 && position == "right") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P2_RIGHT)) {
            return true;
        }
    }
    return false;
}

bool InputManager::isDoing (std::string action, std::string
↪  player) {
    bool isp2 = false;
    if (player == "p2") {
        isp2 = true;
    }

    if (!isp2 && action == "jump") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P1_JUMP)) {
            return true;
        }
    }

    else if (!isp2 && action == "kick") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P1_KICK)) {
            return true;
        }
    }

    else if (isp2 && action == "jump") {
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P2_JUMP)) {
            return true;
        }
    }

    else if (isp2 && action == "kick") {
```

```
        if (sf::Keyboard::isKeyPressed
        ↪  (sf::Keyboard::Key::P2_KICK)) {
            return true;
        }
    }
    return false;
}
```

There are four member functions of this class for four different input methods.They are:

- `isSpriteClicked()` is used to check if the player has clicked the sprites or not and returns boolean value as per the input.In this function, firstly it checks whether the mouse button is pressed or not, if the button is pressed then it checks whether the sprite provided in the parameter contains the cursor and if it does then the `true` value is returned. If any of the condition is not fullfilled `false` value is returned.
- `getMousePosition()` is used to return the vector position of the mouse cursor in the rendering window.
- `isMoving()` checks if any of the player(P1,P2) is pressing the defined key to move the player and returns the boolean value.
- `isDoing()` checks if the player is pressing the defined key to jump or kick the ball and returns the boolean value.

2.3 Movement of Player and Kicking Ball

2.3.1 Movement of Player

```
void Player::moveLeft ( ) {
    b2Vec2 currentVelocity = this->_body->GetLinearVelocity ( );
    this->_body->SetLinearVelocity (b2Vec2(- MOVEMENT_VELOCITY ,
    ↪  currentVelocity.y));
}

void Player::moveRight(){
    b2Vec2 currentVelocity = this->_body->GetLinearVelocity ( );
    this->_body->SetLinearVelocity(b2Vec2(MOVEMENT_VELOCITY,
    ↪  currentVelocity.y));
}
```

```
void Player::jump ( ) {
    b2Vec2 currentVelocity = this->_body->GetLinearVelocity ( );
    this->_body->SetLinearVelocity (b2Vec2 (currentVelocity.x, -
     ↪  MOVEMENT_VELOCITY));
}
```

There are primarily three types of player movement i.e moving left, moving right and jumping. These movement is performed by calling above functions after input is provided by the user.

Here `b2Vec2` is used to define vector in box2d world. It takes two float values as parameters. e.g A new vector can be defined as `b2Vec2 newVector(3.2f, 0.65f);`

`GetLinearVelocity()` is a function that returns the current linear velocity of the center of mass of the body in box2d world. Syntax for this function is `const b2Vec2 & b2Body::GetLinearVelocity( ) const`.

Similarly `SetLinearVelocity()` sets the linear velocity to center of mass of the body in box2d world. Syntax for this function is `void b2Body::SetLinearVelocity(const b2Vec2 &v)`

The variable `MOVEMENT_VELOCITY` is defined in Definition.hpp header file and its value is 5.0f.

2.3.2 Kicking Ball

```
void GameState::kick (std::string player) {
    if (player == "p1") {
        if (this->_p1.sprite( ).getGlobalBounds( ).intersects
         ↪  (this->_ball.shape( ).getGlobalBounds( ))) {
        this->_playerKickSfx.play();
        this->_ball.body( )->ApplyForceToCenter(b2Vec2
         ↪  (Converter::pixelsToMeters (this->_ball.shape(
         ↪  ).getPosition( ).x - this->_p1.sprite( ).getPosition(
         ↪  ).x) * KICK_FORCE_SCALE, Converter::pixelsToMeters
         ↪  (this->_ball.shape( ).getPosition( ).y -
         ↪  (this->_p1.sprite( ).getPosition( ).y +
         ↪  this->_p1.sprite( ).getGlobalBounds( ).height / 2)) *
         ↪  KICK_FORCE_SCALE), true);
```

```
        }
    }

    if (player == "p2") {
        if (this->_p2.sprite( ).getGlobalBounds( ).intersects
        ↪  (this->_ball.shape( ).getGlobalBounds( ))) {
        this->_playerKickSfx.play();
        this->_ball.body( )->ApplyForceToCenter(b2Vec2
        ↪  (Converter::pixelsToMeters (this->_ball.shape(
        ↪  ).getPosition( ).x - this->_p2.sprite( ).getPosition(
        ↪  ).x) * KICK_FORCE_SCALE, Converter::pixelsToMeters
        ↪  (this->_ball.shape( ).getPosition( ).y -
        ↪  (this->_p2.sprite( ).getPosition( ).y +
        ↪  this->_p2.sprite( ).getGlobalBounds( ).height / 2)) *
        ↪  KICK_FORCE_SCALE), true);
        }
    }
}
```

Kicking of the ball is done by multiple steps. Firstly, it is tested whether the ball and player have touched each other or not. It is detected by the line `if(this->_p1.sprite( ).getGlobalBounds( ).intersects(this->_ball.shape( ).getGlobalBounds( )))`. In this line, the function `getGlobalBounds()` returns the coordinates of boundary of the sprite that calls it.

The function `ApplyForceToCenter()` applies the force to the center of mass of the body and wakes the body in box2d world. This process of applying force is considered as kicking the ball. The syntax for this function is `void b2Body::ApplyForceToCenter (const b2Vec2& force, bool wake)`.

The x and y components of the force to be applied is given as follows:

- The x component is a multiple of the difference between the x components of the positions of the ball and the player
- The y component is a multiple of the difference between :
    1. The y component of the position of the ball
    2. The y position of the bottom(leg) of the player i.e the sum of the y component of the position of the player and half of the height of the player.

2.4 Asset Management

Assets managed by this class include Textures, Fonts and Sound.

These assets are stored using `std::map` objects.

An example code snippet of asset management is given below. The code snippet shows the process of loading a texture in `map`, getting reference to the loaded texture and checking whether the requested texture is present in the `std::map`. The management of fonts and sound clips follow the same approach.

In `AssetManager` class of `AssetManager.hpp`

```
// public members:
// for textures:
void loadTexture(std::string name, std::string fileName);
sf::Texture &getTexture(std::string name);
bool isTexturePresent(std::string textureName);

//...

// private members:
std::map<std::string, sf::Texture> _textures;
//...
// similar for fonts and sound clips
```

In `AssetManager.cpp`

```
// Textures
void AssetManager::loadTexture(std::string name, std::string
↪  fileName) {
    sf::Texture texture;

    // loadFromFile() method below loads texture and returns
    ↪  true if successfully loaded, false otherwise
    if(texture.loadFromFile(fileName)) {
        this->_textures[name] = texture;
    }
}

// return the value(texture) at the given key in the map
```

```
    sf::Texture &AssetManager::getTexture(std::string name) {
        return this->_textures.at(name);
    }

    // if count value == 0(false), return false, else true
    bool AssetManager::isTexturePresent(std::string textureName){
       if (!_textures.count(textureName)){
           return false;
       }
       return true;
   }

   //...
```

2.5 Time Management

Time management is done by the `TimeManager` class.In this class various functions are used to start the clock, check time, reset time, process time and display it in the window.This class has following attributes;

- sf::Clock _clock
- sf::Time T
- sf::Time tempTime
- std::stringstream ss
- int t
- int vt
- int s
- int m

```
TimeManager::TimeManager ( ) {
    this->t = 0;
    this->vt = 0;
    this->s = 0;
    this->m = 0;
}
```

```
void TimeManager::resetTimer ( ) {
    this->_clock.restart ( );
}

void TimeManager::processTime ( ) {
    this->T = this->_clock.getElapsedTime() + this->tempTime;
    this->t = this->T.asSeconds();
    this->vt = this->t * (90  / GAME_TIME); //Increase game time
    ↪  by the factor of (90/GAME_TIME)
    this->m = this->vt / 60;    //to convert vt to minutes which
    ↪  is in seconds
    this->s = this->vt - this->m * 60;
}

int TimeManager::getTime ( ) {
    return this->m;
}

std::string TimeManager::displayTimer () {
    this->ss.str (std::string ( ));

    if (m < 10) {
        this->ss << 0 << this->m;
    } else {
        this->ss << this->m;
    }

    if (s < 10) {
        this->ss << ":" << 0 << this->s;
    } else {
        this->ss << ":" << this->s;
    }

    return this->ss.str ( );
}

void TimeManager::pause ( ) {
    this->tempTime = this->T;
```

```
}

void TimeManager::resume ( ) {
    this->resetTimer ( );
}

void TimeManager::setTime (sf::Time timer) {
    this->T = timer;
}

void TimeManager::zero ( ) {
    this->_clock.restart( );
    this-> T = sf::Time ( );
    this->tempTime = this->T;
}
```

- `TimeManager()` is the constructor of this class which sets the initial value of some attributes to zero when a object of TimeManager class is initialized.
- `resetTime()` is used to restart the clock.
- `processTime()` function is used to get the time elapsed in seconds and convert it to game relevant time.
- `getTime()` returns the game time in minutes.
- `displayTimer()` is used to display the timer in the rendering window. `displayTimer()` function uses `stringstream` object to display time in understandable format. (Minute:Second)
- `pause()` function is used to store time in tempTime attribute when the game is paused.
- `resume()` function calls `resetTimer()` function so the clock restarts.
- `setTime()` function sets the value of `this->T` to the given time value.
- `zero()` function restarts the clock and sets the value of attributes: T and tempTime to zero.

2.6 Goal Detection

For the detection of goal, the position of ball is compared with the global bounds of the goal post. If the ball lies at the bounds of the goal post, then it is considered goal and the score of the respective player is increased. The following snippet of code shows the portion where goal is detected.

```
sf::IntRect leftPostRect (this->_leftPost.sprite( ).getPosition(
↪  ).x - this->_leftPost.sprite( ).getGlobalBounds( ).width / 2,
↪  this->_leftPost.sprite( ).getPosition( ).y -
↪  this->_leftPost.sprite( ).getGlobalBounds( ).height / 2,
↪  this->_leftPost.sprite( ).getGlobalBounds( ).width,
↪  this->_leftPost.sprite( ).getGlobalBounds( ).height);
if (leftPostRect.contains(this->_ball.shape( ).getPosition( ).x,
↪  this->_ball.shape( ).getPosition( ).y)) {
    this->_scoreTime->p2Score ++;
    this->_scoreTime->time.pause ( );
    this->_data->machine.addState (StateRef (new GoalState
    ↪  (this->_data, this->_scoreTime, this->_isSecondHalf)) );
}
```

First of all, an `sf::IntRect` object is defined which represents the bounds of the goal post. Then this `IntRect` object checks whether it includes the position of the ball using the `contains( )` method. If the `IntRect` contains the position of the ball, then it is clear that the ball has entered the post. Then, the score of the respective player is increased. Similarly, the time is paused and `GoalState` state is initiated using the `StateMachine`.

2.7 Animation

Animation in a simple sense is just changing the texture of a sprite after some frames. The same approach is taken for animations in the project. The following snippet is a simple example of animation in the splash screen:

```
this->_animationCounter ++;

if (this->_animationCounter == this->_spriteCounter *
↪  FRAMES_PER_ANIMATION * 2) {
    std::stringstream ss;
```

```
    ss << "Splash Anim " << this->_spriteCounter;

    this->_logoSprite.setTexture (this->_data->assets.getTexture
    ↪  (ss.str ( )));
    this->_logoSprite.setOrigin(this->_logoSprite.getGlobalBounds(
    ↪  ).width ,this->_logoSprite.getGlobalBounds( ).height / 2);
    this->_logoSprite.setPosition(WINDOW_WIDTH / 2, WINDOW_HEIGHT
    ↪  / 2);

    this->_spriteCounter ++;
}

if (this->_spriteCounter == 12) {
    this->_data->machine.addState (StateRef (new MenuScreen
    ↪  (_data)));
}
```

The given snippet of code loops every frame. In every frame, the `_animationCounter` increases by one. When the `_animationCounter` is equal to a certain multiple of `_spriteCounter`, then the respective animation texture is applied to the sprite. For this `std::stringstream` is used to concatenate the `_spriteCounter` to the string which defines the texture in the `map`. Then, the texture is set to the sprite using the `setTexture( )` method. Also, the `_spriteCounter` is incremented by one. If the `_spriteCounter` is equal to 12 (the number of sprites in the animation), it marks the end of animation and the state is replaced with `MenuScreen` in the stack.

2.8 Conversion of Coordinate Systems

Box2D and SFML have different coordinate systems. Box2D uses the metric system. It uses meters for the position and radian for angle. While SFML uses pixels coordinates for position and degree for angle.

Converting radians to degrees or vice versa is not difficult, but for the conversion between pixels to meters and vice versa, we need to specify pixels per meter (PPM).

To maintain consistency in the code, a `Converter` namespace is used:

```
namespace Converter {
    const float PPM = 30.0f;
```

```
    const float PI = 3.1415;

    template <class T>
    constexpr T pixelsToMeters (const T &x) { return x / PPM; }

    template <class T>
    constexpr T metersToPixels (const T &x) { return x * PPM; }

    template <class T>
    constexpr T degToRad (const T &x) { return PI * x / 180.0; };

    template <class T>
    constexpr T radToDeg (const T &x) { return 180.0 * x / PI; }
}
```

Note:
A detailed description of all of the classes and their members can be found at: `https://rujalacharya.github.io/HeadBall`

Chapter 6

RESULTS AND DISCUSSION

1 Results

The game has been developed with user-friendliness and simple to play approach in consideration. The following screenshots from different states throughout the game illustrate the final result:

fig. 6.1: *Menu of the game*

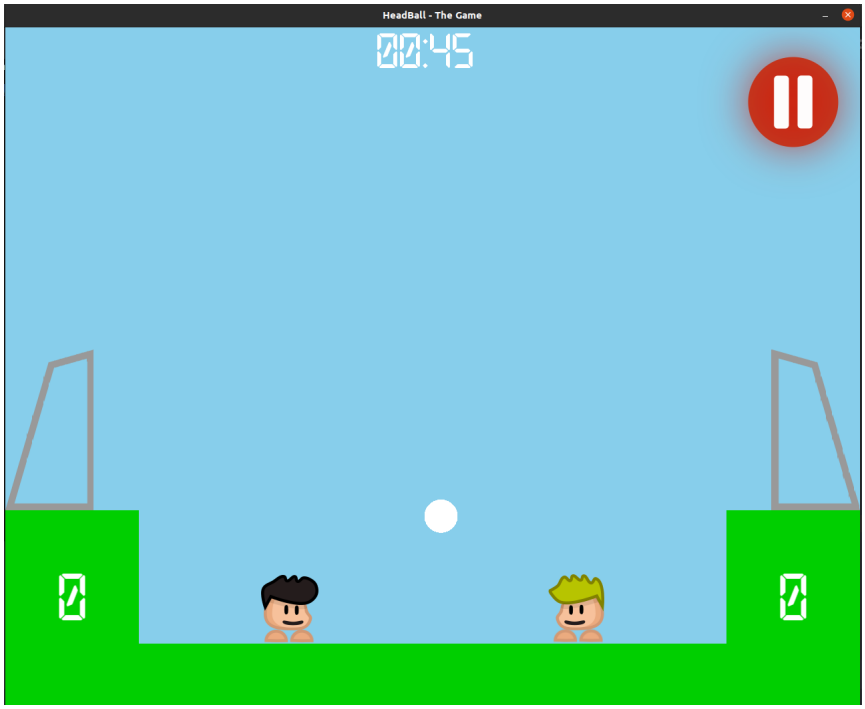

fig. 6.2: *Game Play*

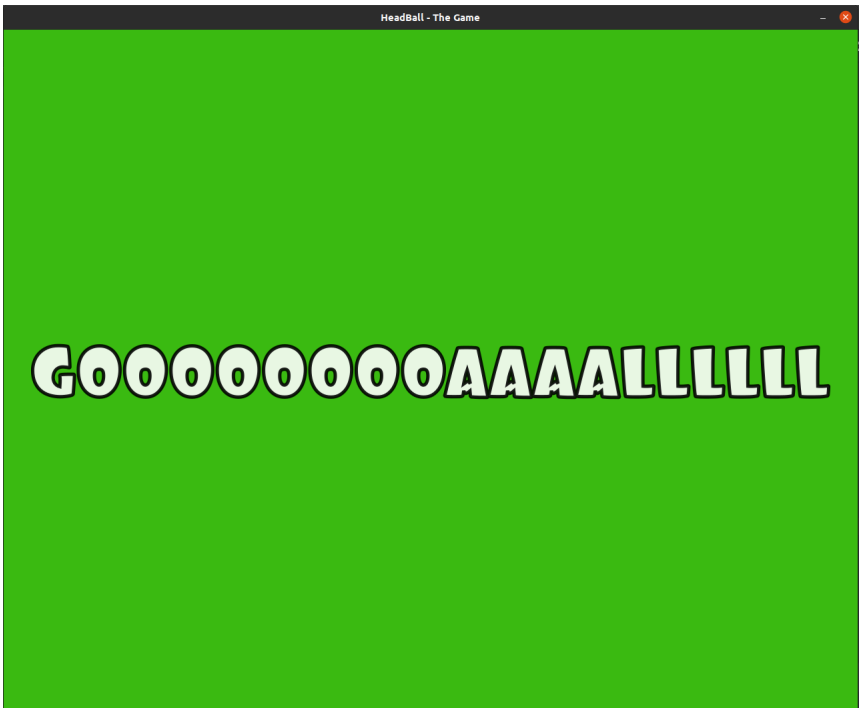

fig. 6.3: *Goal*

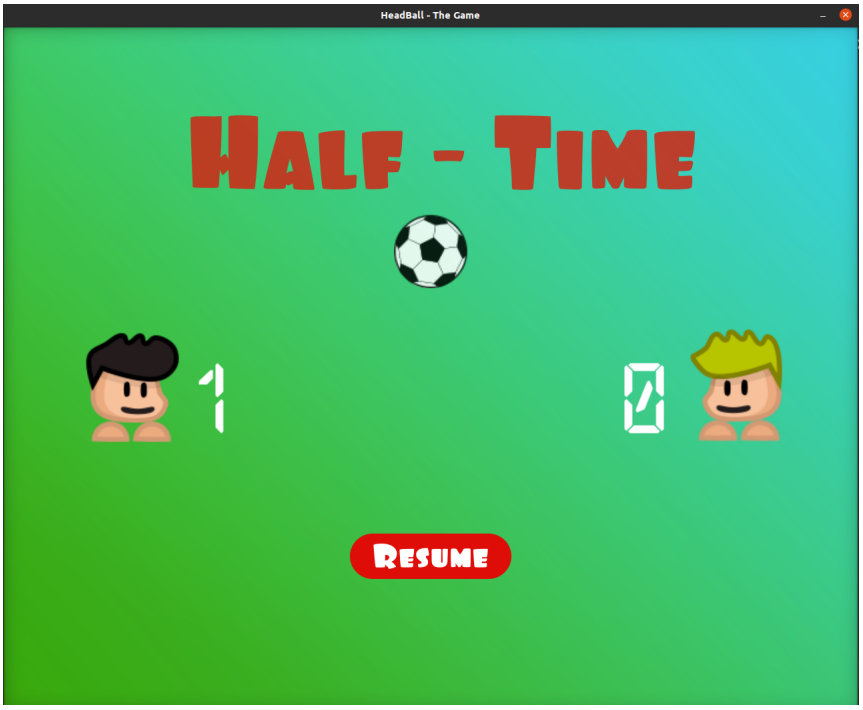

fig. 6.4: Half Time

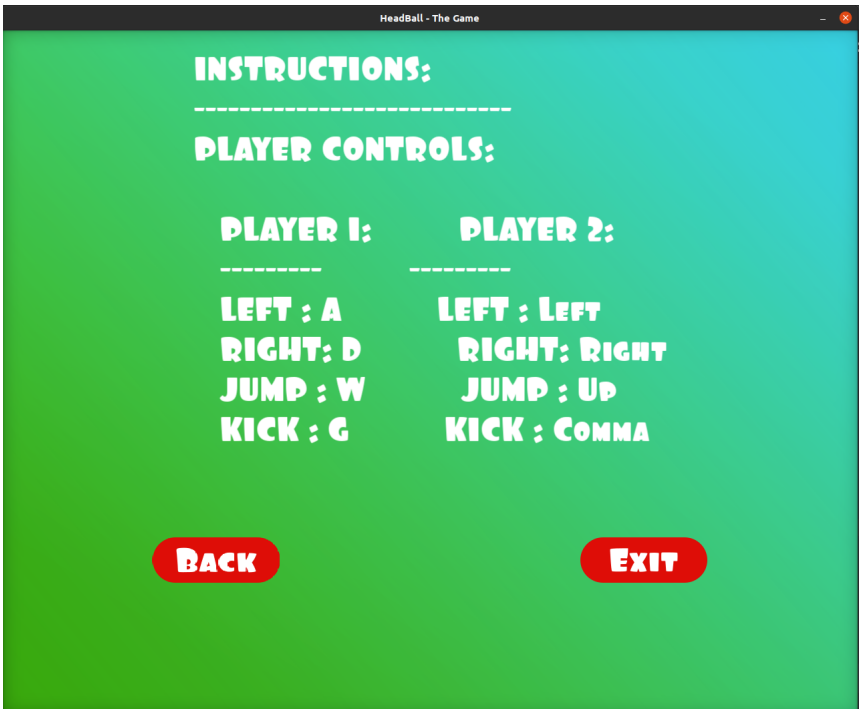

fig. 6.5: Instructions

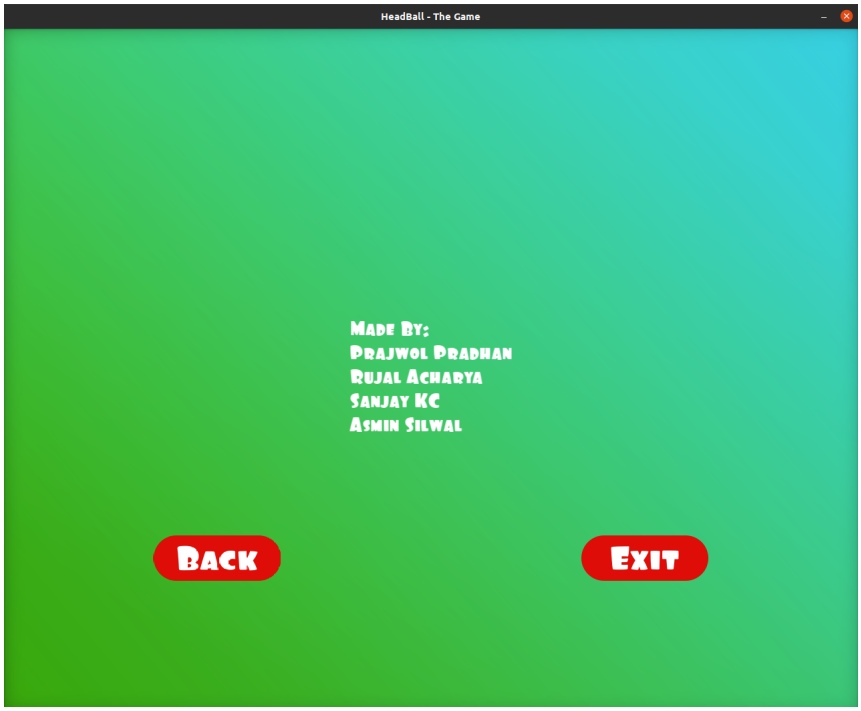

fig. 6.6: About the game

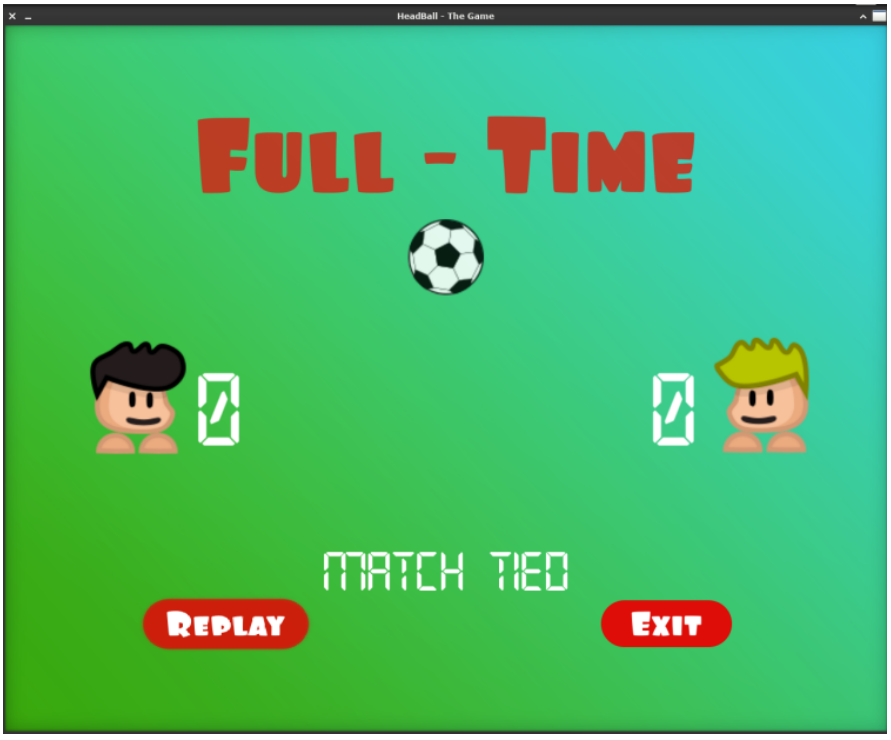

fig. 6.7: Game Over

2 Limitations

Even though there has been put enough effort to make the game as good as possible, there still are some limitations in the game, some of those limitations are:

1. Both players should play on the same computer and use the same keyboard which probably can kill the fun of the game.
2. There is no online multiplayer mode yet so the players cannot play over LAN or Internet.
3. There is no Player vs Computer mode where a player can compete against the computer.
4. Currently the settings need to be configured via code only. So a common user might find it less appealing.
5. The game lacks very impressive graphics.

3 Further Improvements

There are many aspects of the game that can be improved. Some of them are:

1. It can be made an online game by adding networking capability so that two players can play from different computers over LAN or Internet.
2. Voice chat and Text chat functionalities can be added in online version to make it more exciting.
3. Also, Player vs Computer mode can be added so that a single player can play against the computer.
4. The graphical aspect can be further improved to make the game look more beautiful.

Chapter 7

CONCLUSIONS

In this way the project is completed. This project was a great learning experience for our group members as we successfully implemented theoretical knowledge into practice and succeeded in achieving the goal. It taught us different aspects of game development and more importantly, it taught us to write C++ code in OOP paradigm following certain coding guidelines. This project has filled us with more enthusiasm for more projects of game development and may even guide us to professional game development as a career.

Additionally, we learned about team-work through collaborative problem-solving. We also got the opportunity to learn more about the industry-standard tools and technologies such as LaTeX for typesetting documents, GitHub for version control and hosting of our project, Doxygen for documentation generation, and so on.

This project taught us different aspects of software and game development cycle which includes the stages like debugging, building and testing of the programs. In conclusion, C++ programming language with OOP paradigm is a very powerful tool that can be used in software and game development fields.

Bibliography

[1] S. Systems, “Flappy bird sfml tutorial series - youtube,” 09 2017.

[2] GeeksforGeeks, “Features of c++,” 09 2020.

[3] “Smart pointers - cppreference.com.”

[4] “Smart pointer,” 10 2020.

[5] GeeksforGeeks, “Introduction of smart pointers in c++ and it’s types,” 06 2014.

[6] M. Barbier, *SFML Blueprints*. Packt Publishing Ltd, 2015.

[7] R. Pupius, *SFML Game Development By Example*. Packt Publishing Ltd, 2015.

Appendix A (Relevant Links)

- The GitHub repository of the project:
  `https://github.com/RujalAcharya/HeadBall`
- Online Documentation of the project:
  `https://rujalacharya.github.io/HeadBall`
- SFML:
  `https://www.sfml-dev.org`
- Box2D:
  `https://box2d.org`