TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

GRAPHICAL APPROACH BY C++ USING SDL2

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:

AVINASH ARYAL (PUL076BEI009)

DIWAS ADHIKARI (PUL076BEI014)

KAILASH PANTHA (PUL076BEI017)

Submitted to:

Department of Electronics and Computer Engineering, Pulchowk Campus

Institute of Engineering, Tribhuvan University

Lalitpur, Nepal

Abstract

Computer graphics helps us express procedures in a better pictorial way. Generally we were just limited to executing programs in a console window or a terminal so that, we decided to build programs graphically on GUI-interfaced generic window. Our sole purpose to create this project is to understand how computer graphics works and how we can manipulate graphical content for animation , transition…etc. We have strictly followed the object-oriented programming paradigm i.e. OOP structure which includes data abstraction, encapsulation, and data privacy.

Acknowledgment

Foremost, we would like to express our sincere gratitude to respected lecturer Er. Bikal Adhikari for the continuous support in our studies and projects for his patience, motivation, enthusiasm and immense knowledge. His guidance helped us in the build up of this project. We couldn't have imagined having a better advisor and mentor for our project. We cannot end this here without acknowledging the power of the internet that has a massive vault of useful and worthwhile resources that helped in every step of our project. At last, we would like to admire this opportunity where, we got to showcase our determination, commitment and teamwork .

TABLE OF CONTENTS

CHAPTER THREE : GAME DESIGN

3.1 Snake (Design)

- 3.1.1 Initialization
- 3.1.2 Movement
- 3.1.3 Collision Detection

3.2 PONG! (Design)

- 3.2.1 Initialization
- 3.2.2 Paddle Movemetnt
- 3.2.3 AI Implementation
- 3.2.4 Ball Movement And Collision Detection
- 3.2.5 Score Deduction And Winner Determination

CHAPTER FOUR : PROBLEM ANALYSIS

4.1 Algorithm

- 4.1.1 Snake (Algorithm)
- 4.1.2 PONG! (Algorithm)

4.2 Flowchart

- 4.2.1 Snake (Flowchart)
- 4.2.2 PONG! (Flowchart)

CHAPTER FIVE : CODING AND TESTING OF THE FRAMEWORK

5.1 Snake (Code)

- 5.1.1 Making ‘ Body ‘ Class
- 5.1.2 Making ‘ Food ‘ Class
- 5.1.3 Making ‘ Snake ‘ Class
- 5.1.4 Modification For ‘ Launcher ‘ Class
- 5.1.5 UML Diagram
- 5.1.6 Testing

CHAPTER ONE : INTRODUCTION

1.1 INTRODUCTION TO C++

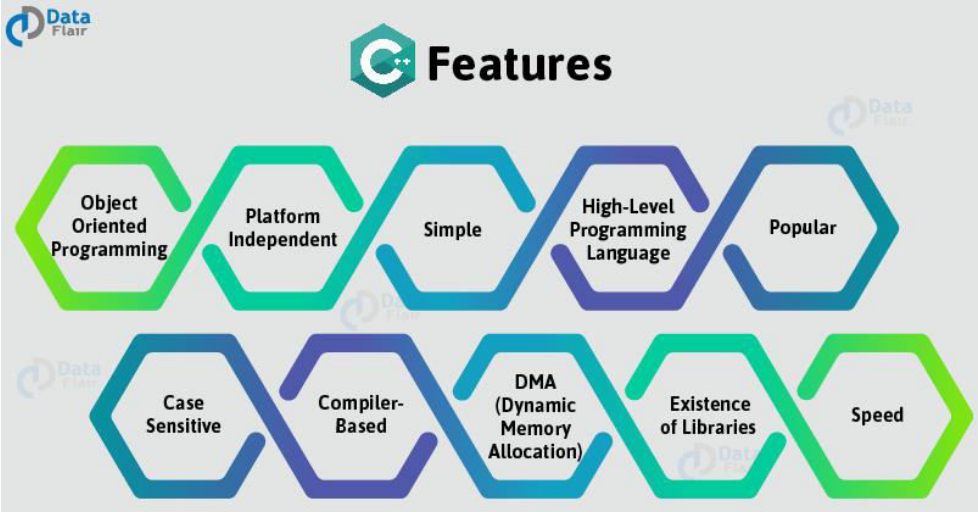
1.1.1 History

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, video games, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications (e.g. telephone switches or space probes).

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as *ISO/IEC 14882:2017* (informally known as C++17).The C++ programming language was initially standardized in 1998 as *ISO/IEC 14882:1998*, which was then amended by the C++03, C++11 and C++14 standards. The current C++17 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization. Since 2012, C++ is on a three-year release schedule, with C++20 the next planned standard (and then C++23).

1.1.2. Features of C++



1. **OOP (Object-Oriented Programming)**

C++ is an object-oriented language, unlike C which is a procedural language. This is one of the most important features of C++. It employs the use of objects while programming. These objects help you implement real-time problems based on data abstraction, data encapsulation, data hiding, and polymorphism. We have briefly discussed all the 5 main concepts of object-oriented programming.
The OOP concepts are:

- **Data Abstraction:** Data abstraction is an act of representing the important features of data without including the background details or the method applied to obtain it.

- **Data Encapsulation:** Data encapsulation is nothing but a process to implement data abstraction by wrapping up the data and functions into an exclusive block.

- **Inheritance:** The term inheritance refers to transferring the properties of the parent class to the child class. We can implement the basic idea of inheritance by creating more than one class, which we formally refer to as derived classes by linking them with what we call the base class. This concept reduces the redundancy of the program and makes it easy to transfer/copy the properties of one class to another.

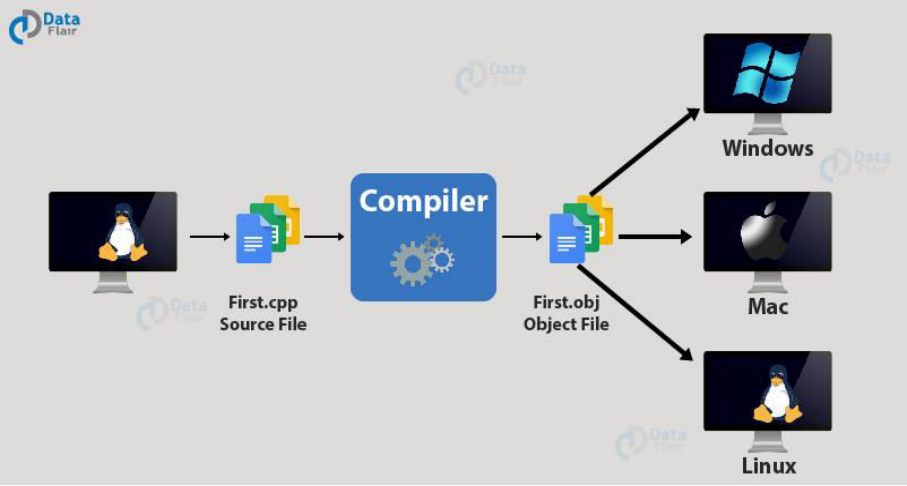
- **Data hiding:** Data hiding refers to protecting data from unauthorized access. It is basically responsible for securing the data. It is important to note that data encapsulation is different from data hiding as encapsulation mainly focuses on shifting the focus on important data than explaining its complex nature.

- **Polymorphism:** The word poly means 'many' and 'morphism' means 'forms'. Clearly, polymorphism refers to displaying that data in more than one form.

2. Platform or Machine Independent/ Portable

In simple terms, portability refers to using the same piece of code in varied environments.

Let us understand this C++ feature with the help of an example. Suppose you write a piece of code to find the name, age, and salary of an employee in Microsoft Windows and for some apparent reason you want to switch your operating system to LINUX. This code will work in a similar fashion as it did in Windows.



3. Simple

When we start off with a new language, we expect to understand in depth. The simple context of C++ gives an appeal to programmers, who are eager to learn a new programming language.

If you are already familiar with C, then you don't need to worry about facing any trouble while working in C++. The syntax of C++ is almost similar to that of C. After all C++ is referred to as "C with classes".

4. High-level programming language

It is important to note that C++ is a high-level programming language, unlike C which is a mid-level programming language. It makes it easier for the user to work in C++ as a high-level language as we can closely associate it with the human-comprehensible language, that is, English.

5. Popular

After learning C, it is the base language for many other popular programming languages which supports the feature of object-oriented programming. Bjarne Stroustrup found Simula-67, the first object-oriented language ever, lacking simulations and decided to develop C++.

6. Case sensitive

Just like C, it is pretty clear that the C++ programming language treats the uppercase and lowercase characters in a different manner. For instance, the meaning of the keyword **'cout'** changes if we write it as **'Cout'** or **"COUT"**. Other programming languages like HTML and MySQL are not case sensitive.

7. Compiler-Based

Unlike Java and Python that are interpreter-based, C++ is a compiler based language and hence it a relatively much faster than Python and Java.

8. DMA (Dynamic Memory Allocation)

Since C++ supports the use of pointers, it allows us to allocate memory dynamically. We may even use constructors and destructors while working with classes and objects in C++.

9. Existence of Libraries

The C++ programming language offers a library full of in-built functions that make things easy for the programmer. These functions can be accessed by including suitable header files.

10. Speed

As discussed earlier, C++ is compiler-based hence it is much faster than other programming languages like Python and Java that are interpreter-based.
Summary

Here, we shed light on the remarkable features of C++ by motivating the newbie programmers to study this language because of the features it offers, making it unique and ubiquitous. According to Tiobe-Index 2019, C++ holds the 4th position. Currently, many industries are using C++ and in futures, we can see many more real-time applications.

1.2.1 Introduction to SDL2.0:

SDL stands for “Simple Direct-Media Layer”. It is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. SDL officially supports Windows, Mac OS X, Linux, iOS, and android.

SDL is free and open-source software subject to the requirements of the z-lib License since version 2.0, and with prior versions subject to the GNU Lesser General Public License .Under the z-lib License, SDL 2.0 is freely available for static linking in closed-source projects, unlike SDL 1.2. SDL 2.0, released in 2013, was a major departure from previous versions, offering more opportunity for 3D hardware acceleration, but breaking backwards-compatibility.

SDL is extensively used in the industry in both large and small projects. Over 700 games, 180 applications, and 120 demos have been posted on the library website

A common misconception is that SDL is a game engine, but this is not true. However, the library is suited to building games directly, or is usable indirectly by engines built on top of it.

1.2.2. History:

Sam Lantinga created the library, first releasing it in early 1998, while working for *Loki Software*. He got the idea while porting a Windows application to *Macintosh*. Several other free libraries were developed to work alongside SDL, such as SMPEG and OpenAL. He also founded *Galaxy Gameworks* in 2008 to help commercially support SDL, although the company plans are currently on hold due to time constraints.

Sam Lantinga

Lantinga announced SDL 2.0 on 14 July 2012, at the same time announcing that he was joining Valve, the first version of which was announced the same day he joined the company. *Lantinga* announced the stable release of SDL 2.0.0 on 13 August 2013.

SDL 2.0 is a major update to the SDL 1.2 codebase with different, not backwards-compatible API. It replaces several parts of the 1.2 API with more general support for multiple input and output options. Some feature additions include multiple window support, hardware-accelerated 2D graphics, and better Unicode support.

Support for Mir and Wayland was added in SDL 2.0.2 and enabled by default in SDL 2.0.4.Version 2.0.4 also provided better support for Android.

1.2.3. Software Architecture:

SDL is a wrapper around the operating-system-specific functions that the game needs to access. The only purpose of SDL is to provide a common framework for accessing these functions for multiple operating systems (cross-platform).SDL provides support for 2D pixel operations, sound, file access, event handling, timing and threading. It is often used to complement OpenGL by setting up the graphical output and providing mouse and keyboard input, since OpenGL comprises only rendering.

A game using the Simple Direct-Media Layer will not automatically run on every operating system, further adaptations must be applied. These are reduced to the minimum, since SDL also contains a few abstraction APIs for frequent functions offered by an operating system. The syntax of SDL is function-based: all operations done in SDL are done by passing parameters to subroutines (functions).

1.2.4. Applications:

Over the years SDL was used for many commercial and non-commercial video game projects. For 120 games using SDL in 2013, and the SDL website itself listed around 700 games in 2012.Important commercial examples are *Angry Birds* and *Unreal Tournament*; ones from the open-source domainare: *OpenTTD*, *The Battle for Wesnoth* or, *Freeciv*.

The cross-platform game releases of the popular Humble Indie Bundles for Linux, Mac and Android are often SDL based.

SDL is also often used for later ports on new platforms with legacy code. For instance, the PC game Homeworld was ported to the Pandora handheld and Jagged Alliance 2 for Android via SDL. Also, several non-video game programs use SDL; examples are the emulators DOSBox and VisualBoyAdvance.

There were several books written for development with SDL (see further readings).

SDL is used in university courses teaching multimedia and computer science, for instance, in a workshop about game programming using libSDL at the University of Cadiz in 2010, or a Game Design discipline at UTFPR (Ponta Grossa campus) in 2015.

1.2.5. Extended libraries:

1. SDL_image

SDL_image is an image loading library that is used with the SDL library, and almost as portable. It allows a programmer to use multiple image formats without having to code all the loading and conversion algorithms themselves.

SDL is deliberately designed to provide the bare bones of creating a graphical program. As such, it doesn't provide methods for loading various image formats into your program. That's where a library like SDL_image comes in. Using SDL_image you can load in the popular image formats such as BMP, PNM (PPM/PGM/PBM), XPM, LBM, PCX, GIF, JPEG, PNG, TGA, and TIFF formats. These are loaded onto your SDL_Surface and painted onto the screen as normal. It supports alpha transparency for those file formats that store it (eg: .png).

```
SDL_Surface *IMG_Load(const char *file);    // or,
SDL_Surface *IMG_Load_RW(SDL_RWops *src, int freesrc);  // or,
SDL_Surface *IMG_LoadTyped_RW(SDL_RWops *src, int freesrc, char *type);
```

2. SDL_mixer

SDL_mixer is a sound mixing library that is used with the SDL library, and almost as portable. It allows a programmer to use multiple samples along with music without having to code a mixing algorithm themselves. It also simplifies the handling of loading and playing samples and music from all sorts of file formats.

```
Mix_Chunk * Mix_GetChunk(int)
```

3. SDL_ttf

SDL_ttf is a TrueType font rendering library that is used with the SDL library, and almost as portable. It depends on freetype2 to handle the TrueType font data. It allows a programmer to use multiple TrueType fonts without having to code a font rendering routine themselves. With the power of outline fonts and antialiasing, high quality text output can be obtained without much effort.

```
TTF_Font* TTF_OpenFont(const char* file, int ptsize);
SDL_Surface *TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg);
```

Game : SNAKE

Introduction

Snake is the common name for a video game concept where the player maneuvers a line which grows in length, with the line itself being a primary obstacle. The ease of implementing *Snake* has led to hundreds of versions.

History

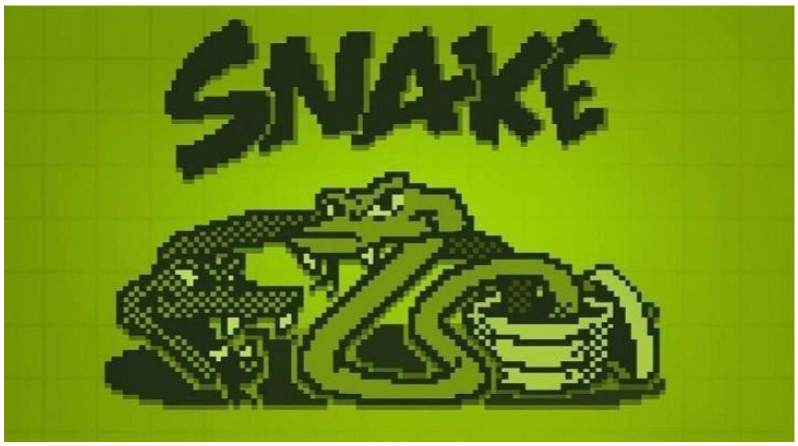
The snake design dates back to the arcade game Blockade developed and published by Gremlin in 1976.It was cloned as *Bigfoot Bonkers* the same year. In 1977, Atari released two *Blockade*-inspired titles: the arcade game *Dominos* and Atari VCS game *Surround*. *Surround* was one of the nine Atari VCS (later the Atari 2600) launch titles in the United States and was also sold by Sears under the name *Chase*. That same year, a similar game was launched for the Bally Astrocade as *Checkmate*.

The first known personal computer version, titled *Worm*, was programmed in 1978 by Peter Trefonas of the US on the TRS-80, and published by *CLOAD* magazine in the same year. This was followed shortly afterwards with versions from the same author for the Commodore PET and Apple II. An authorized version of *Hustle* was published by Milton Bradley for the TI-99/4A in 1980.In 1982's *Snake* for the BBC Micro, by Dave Bresnen, the snake is controlled using the left and right arrow keys relative to the direction it is heading in. The snake increases in speed as it gets longer, and there's only one life; one mistake means starting from the beginning.

Basic concept for how does the game works:

The snake moves in a very precise way. Based on what the user types, the snake will move in a given direction. Every time the snake moves, the head will go in the new direction, and every piece of the snake will move up, by occupying the space that was formerly occupied by the piece in front of it. To grow in size, the snake has to eat food. How can we show the snake eating? The simplest answer is that if the

head of the snake and the food are in the same place, we consider that the snake eats the food. This means that we have to know where the food is. When it's eaten, it disappears, the snake grows, and the food shows up somewhere else. The coordinates of the food are part of its variables and it has a function move, which will move it to a different place. To make the game interesting, we probably want this to be a random location, which means that we'll have to make sure that our program can generate random numbers. We also made sure that the food will keep track of its own color that would be another variable. But we also wanted to show a score, so we need a variable to keep track of that as well. In this case, we'll create a scoreboard class where we can increase the counter value and display it. Likewise, a new toxicity is introduced in the game which decreases the length of the snake the value of toxicity is high at the beginning of the program and gradually decreases as the game continues for the longer time which enables player to control the snake size.



Game : PONG !

Introduction

Pong! is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The points are earned when one fails to return the ball to the other.

History

Pong! is a table tennis–themed arcade video game, featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. It was one of the earliest arcade video games; it was created by Allan Alcorn as a training exercise assigned to him by Atari co-founder Nolan Bushnell, but Bushnell and Atari co-founder Ted Dabney were surprised by the quality of Alcorn's work and decided to manufacture the game.

Pong was the first commercially successful video game, and it helped to establish the video game industry along with the Magnavox Odyssey. Soon after its release, several companies began producing games that closely mimicked its gameplay. Eventually, Atari's competitors released new types of video games that deviated from *Pong*'s original format to varying degrees, and this, in turn, led Atari to encourage its staff to move beyond *Pong* and produce more innovative games themselves.

Basic concept for how does the game works :

The ball moves in a very precise way. The boundary is set for the top and the bottom such that when the ball collides the wall then it returns back within the game field. The paddles are introduced into the game field at right and the left position respectively. The right paddle is controlled by computer where as the left paddle is controlled by player when the ball collides the paddle then it changes its direction and moves away from the paddle towards another paddle. If the paddle misses the ball and hits the boundary behind then the score decreases.

CHAPTER 2 :
INTRODUCTION TO GRAPHICS FRAMEWORK

As, our objective (previously on '*Objective'* section), was to create a graphical framework that is certainly able to render 2D-graphics on a window screen so that, we will be able to implement it for various graphical programs i.e. games, simulations, graphs, etc.

Since, we have generalized it as a framework, it is a skeleton of graphical rendering operations that is just able enough to start a window on its own which can be fused with textures, music and text on that rendering screen.

2.1. Approach to the framework:

Before making this project, we have been certain about the thing that it is not a graphical engine that can be manipulated by either CUI or, GUI interfaces. But, it will just be a main class that gets assisted by its helper classes. (*not to be confused with: sub-classes or, inherited classes*)

Basically, we are going to create a main class (*let's say: Launcher*) that would have graphic rendering functions which can be used to render graphics intertwined with other features, generated in other helper classes (*let's say: Texture, Music, etc.*) for respective purposes.

2.2. Requirements for the framework:

Though, we have decent knowledge of programming using C++ language, we still are unaware of the graphical abilities of the operating system. In Windows, the system uses *Win32* API for the purpose of graphical features and functions. But, it's extremely tedious to learn it since it is way more hardware-oriented. Hence, we could also have tried OpenGL as a graphics library but, we were not used to code in it and also, we also didn't require projections, 3-D rendering, shaders, etc. for rendering display. Hence, we picked up SDL2.0 as our graphics library to make this simple 2-D graphics framework. We have already given a brief explanation about this graphics API earlier in this report. (Check *'Chapter 1 – Introduction'*).

2.3. Why and How SDL2.0 is used? :

Simply, SDL2.0 is preferred as our library for graphical display because it's cross-platform, easy to use and completely free. It is way easier to create a graphical context in SDL2.0 than any other library (Except: *SFML*). Moreover, it is also capable of using OpenGL, Vulkan, etc. for 3-D rendering. (*though, we don't require it.)*

Our general plan in this project was to mask the functions of SDL2.0 in separate classes under separate namespaces so that, SDL2.0 will act as an underneath graphic driver under our framework. Our multiple classes in the framework can simply distribute the functionalities of SDL2.0 as separate entities referred by separate classes which could help us preserve the OOP architecture in programming. We haven't let go of the OOP techniques which includes encapsulation and data abstraction for the sake of our project. We have taken OOP and graphics rendering of SDL2.0 side by side, to generate a working framework to code graphics.

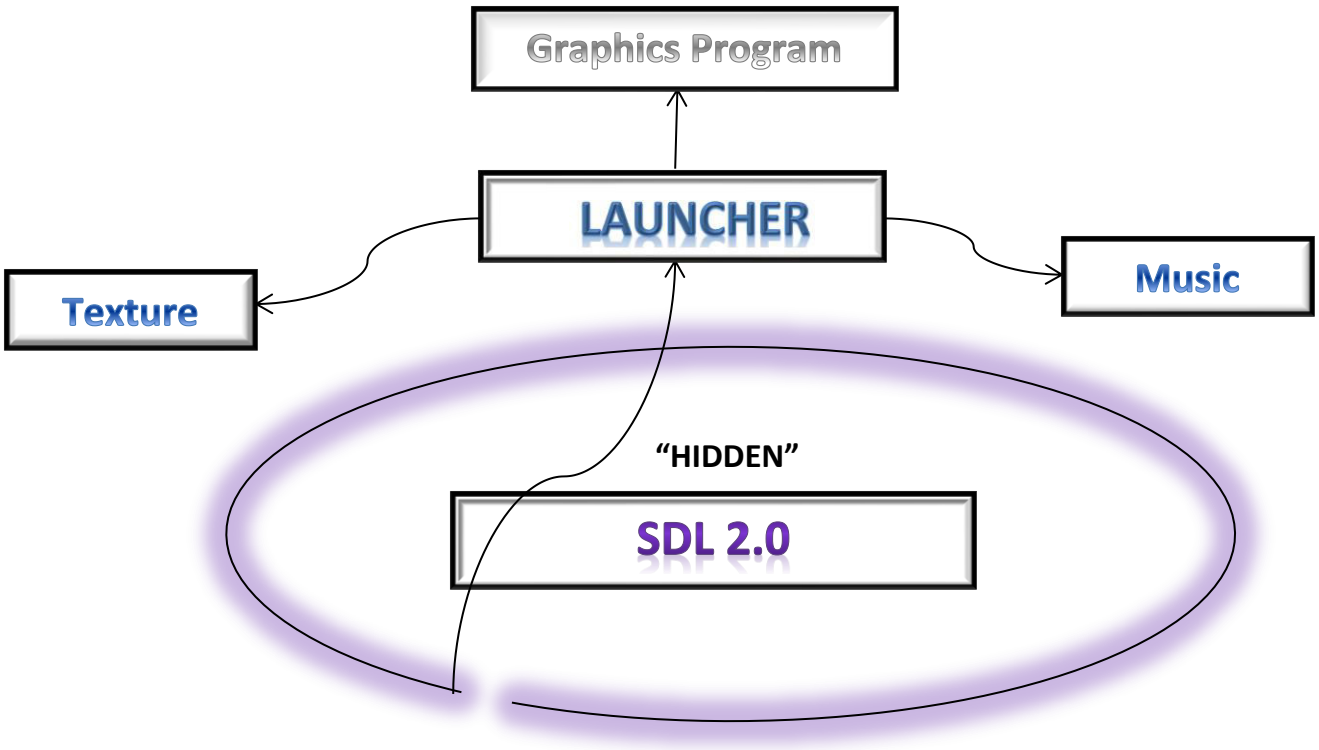


Fig: SDL2.0 adding graphics layer underneath the framework

2.4. Making Classes for the framework:

Since, we have already decided to make multiple classes; we made a main class (*not a parent class*) which will link all the other classes together. Thus, we figured out that we will create a *Launcher* class which would link all other classes together for graphical output. For other graphical necessities like texture, music, fonts, etc. we would create respective classes for them.

2.4.1. 'Launcher' Class:

This class would comprise of a basic window and a renderer that would be defined to make graphical context on it.

First of all, all header files for the graphics API i.e. SDL2.0 were included in the header file for the class: Launcher.hpp

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>
#include <SDL2/SDL_ttf.h>
```

All the member function and private variables were declared in the header file.

```
class Launcher
{
    public:

        void initScreen(const char *title, int width, int height, int flag) ;
        void handleEvents() ;
        void updateScreen() ;
        void renderScreen() ;
        void quit() ;
        bool isRunning() ;
        static SDL_Renderer *renderer ;
        static bool switchedOn ;
        static SDL_Event event ;
        void getDt(double t) ;
        static double deltaTime ;
        static int scr_width, scr_height ;

    private:

        SDL_Window *window ;

} ;
```

SDL_Window, SDL_Event and SDL_Renderer are API-defined variables of SDL2.0.

Then, for making a graphical launcher, we created a window and a renderer with a flag (switchedOn) which, if true will activate the rendering loop for the program. So, we create a function called initScreen() for this purpose. It requests for the width and height of screen with title for it and also asks for a flag that can toggle to fullscreen mode.

```
void Launcher::initScreen(const char *title, int width, int height, int flag)
{
    if(SDL_Init(SDL_INIT_EVERYTHING) != 0)
    {
        std::cout << "SDL 2.0 FAILED TO INITIALIZE ! " << std::endl ;
        scr_width = width ;
        scr_height = height ;
    }

    std::cout << "SDL 2.0 INITIALIZED SUCCESSFULLY ! " << std::endl ;
    window = SDL_CreateWindow(title,SDL_WINDOWPOS_CENTERED,SDL_WINDOWPOS_CENTERED,width,height,flag) ;
    renderer = SDL_CreateRenderer(window, -1, 0) ;
    switchedOn = true ;
    Launcher::scr_width = width ;
    Launcher::scr_height = height ;
}
```

Now, we created a function that could handle events for the window. It is used to handle quitting of the window with ‘X’ or, close button of the top-right window and can be used for minimizing or, maximizing the window. It can also help in key-mapping and mouse input.

```
void Launcher::handleEvents()
{
    SDL_PollEvent(&event) ;
    if(event.type == SDL_QUIT)
    {
        switchedOn = false ;
    }
    if(event.type == SDL_KEYDOWN)
    {
        switch(event.key.keysym.sym)
        {
            case SDLK_ESCAPE :
                switchedOn = false ;
                break ;

            default :
                break ;

        }
    }
}
```

Here, ESC can be pressed to quit the window easily.

We required a rendering instance that would iterate every frame to create transition or graphical behavior in the context of the screen. For that, we created a renderScreen() function to clear and present the canvas at which graphics is created along the layers. SDL_RenderClear() clears the rendering canvas and then it can be presented on the screen through renderer using SDL_RenderPresent().

```
void Launcher::renderScreen()
{
    SDL_RenderClear(renderer) ;

    SDL_SetRenderDrawColor(renderer, 0,0,0,255) ;
    SDL_RenderPresent(renderer) ;
}
```

Now, since we already have a basic rendering framework, we only need to free the memory allocated for our renderer and window. So, we made a quit() function to destroy them only if the launcher is closed or, if 'ESC' is pressed.

```
void Launcher::quit()
{
    std::cout << "SDL 2.0 TERMINATED ! " << std::endl ;
    SDL_DestroyRenderer(renderer) ;
    SDL_DestroyWindow(window) ;
    SDL_Quit() ;
}
```

We also needed a small function called isRunning() that would simply return the launcher flag that would help to identify whether the launcher must be closed or not.

```
bool Launcher::isRunning()
{
    return switchedOn ;
}
```

This class is pretty much capable enough to create its own blank window which can be cleared with different other colors by providing color scales to SDL_SetRenderDrawColor() function in the renderScreen() function. As, it only creates a window, we decided to create another class specific for loading images and clipping textures.

2.4.2. 'Texture' Class:

Though, *Launcher* class is capable enough of initializing SDL2.0 and creating its graphical window instance, we required a way to make our graphics engaged with textures and images to make it look good. *Texture* class is not only required for images but, its general purpose is to generate textures out of surfaces which can be text, image or even fonts.

This class would include 'Launcher.hpp' header file which would help it grab all the macros, constants and the include directives from the *Launcher* class. This would automatically include SDL2.0's external library i.e. SDL2_image which deals specifically with textures, surfaces and image loading/clipping.

First, a header file for *Texture* class was created in the way shown below:

```
#include "Launcher.hpp"

class Texture
{
    public :

        static SDL_Texture *loadTexture(const char *filename) ;

        static SDL_Texture *createFromSurface(SDL_Surface *s) ;

        static void free_surface(SDL_Surface *s) ;

} ;
```

In this class, it is clearly seen that all of the function are kept static. It's because we want to access every functionality in the main class i.e. *Launcher* so that, we only require to provide scope to these functions with its namespaces.

For example: if we wanted to create a texture in any graphics .cpp file that would include *Launcher* class, we can simply refer to *Texture* class by using the namespace to access member function.

SDL_Texture *texture = Texture::load_Texture("rsrc/images/1.png")

This will easily help to use other classes in the *Launcher* class and in fact, we used the macros of the *Launcher* for the namespaces making it way more linked with the main class.

Thus, in the source file of class, we defined the function loadTexture() that would take a file path as its argument to look up that path and load that image into a surface. It first creates a temporary surface that would be used to create texture which gets returned from this function.

```
SDL_Texture *Texture::loadTexture(const char *filename)
{
    SDL_Surface *surface = IMG_Load(filename) ;

    if(surface == nullptr)
    {
        std::cout << "\nTexture Loader Diagnosis -> " << std::endl ;
        std::cout << "Error loading the image : " << filename << std::endl << "{Troubleshoot : "
        << SDL_GetError() << "}\n" << std::endl ;
    }

    SDL_Texture *texture  = SDL_CreateTextureFromSurface(Launcher::renderer, surface) ;
    SDL_FreeSurface(surface) ;
    return texture ;
}
```

SDL_Texture is an API-defined variable to store texture-related data. IMG_Load() function loads the image and SDL_CreateTextureFromSurface() builds the texture of that image.

As we already addressed that textures are not only created out of images but also out of text or font. Hence, we also create a function that could simply intake a surface and convert it to a texture.

```
SDL_Texture *Texture::createFromSurface(SDL_Surface *s)
{
    SDL_Texture *texture  = SDL_CreateTextureFromSurface(Launcher::renderer, s) ;
    return texture ;
}
```

The memory allocated for the surface pointer needs to be freed so, we create a function free_surface() that could clean the pointer memory of the surface.

```
void Texture::free_surface(SDL_Surface *s)
{
    SDL_FreeSurface(s) ;
}
```

2.4.3. ‘Font’ Class:

Font class is a requirement since, if we want to add graphics on the screen, we may require to add text using various fonts. SDL2.0 allows use of TTF and RTF fonts. But, we only require TTF fonts which are simple basic fonts used in our general operating systems so that, we used another extended library for this class i.e. SDL2_ttf.

We have already stated that *Launcher* class needs to be included for almost every other class of this wrapper framework. Thus, first of all, we declared the class with the functions that we require which are of course, static in the header file of the *Font* class. (Why are static functions used – explained in 2.4.2 ‘*Texture*’)

```
class Font
{
    public :

        static TTF_Font *loadFont(const char *filename, int fontSize) ;

        static SDL_Surface *getSurface(TTF_Font *font, const char *text, SDL_Color color) ;

        static void close(TTF_Font *f) ;

} ;
```

In the source file, a function loadFont() is created which would take the file path as an argument and load that certain font while initializing TTF library.

```
TTF_Font *Font::loadFont(const char *filename, int fontSize)
{
    TTF_Init() ;
    TTF_Font *font  = TTF_OpenFont(filename, fontSize) ;

    if(font == nullptr)
    {
        std::cout << "\nError loading fonts : " << filename << std::endl ;
        std::cout << "{Troubleshoot : " << TTF_GetError() << "}" << std::endl ;
    }

    else
    {
        std::cout << "Font Renderer Activated ..." << std::endl ;
    }

    return font ;
}
```

This function would return a TTF_Font data type pointer used to allocate memory for font. First, TTF_Init() is used to initialize the TTF library. If initialized, it takes the file path and the font size to open and load the font of .ttf type. If file path is wrong or, if it doesn't exist then, an error pops up on the console window.

Now, the font can be used to generate text that would be assembled as a surface which can be used to build textures using *Texture* class. Thus, we created another function getSurface() that would intake a text with type of font to be used and the color of the texture. This would help us build a surface out of text that can make surfaces to clip textures out of it. Those textures can be copied on the primitives like rectangles to render it on the screen of graphical context.

```
SDL_Surface *Font::getSurface(TTF_Font *font, const char *text, SDL_Color color)
{
    SDL_Surface *tmp_surface = TTF_RenderText_Solid(font , text , color) ;
    return tmp_surface ;
}
```

SDL_Color is simply a structure of {r, g, b, a} values of the color that can be defined under SDL2.0. This just simply returns the surface that needs to be converted into textures using *Texture* class.

At last, now we required a function to close this library and also get rid of the memory taken by the fonts. Hence, we created a closeFont() function that would take the TTF_Font type as an argument to free the memory and also quit the TTF library.

```
void Font::close(TTF_Font *f)
{
    TTF_CloseFont(f) ;
    TTF_Quit() ;
}
```

This is pretty much enough for our *Font* class for using fonts to write texts and update textual data on the rendering display.

2.4.4. ‘Timer’ class:

Timer class may not look as if it is an important class in a graphical context, but we assure that it is the most important class among our helper classes. In simulations and games, frame rate plays a very important role in it where certain frames need to transition in a required time interval for a particular animation effect to occur. Hence, *Timer* class would help to make the software made by this wrapper framework run at the same frame rate in every system though, it has better hardware or, not. It helps to cap/hold our frame rate to a limit so that, it would produce similar performance in pretty much every type of hardware.

First of all, the class was declared with required member functions and the variables that we require for the class are kept private. ‘periodTicks’ will hold the amount of milliseconds for every frame, ‘deltaTicks’ will hold the ticks at which the frame rate is always differential or fluctuating and, ‘deltaTime’ will be the time that needs to be delayed for every frame to keep the frame rate constant for the graphical performance.

```
#define CAP_LIMIT_ENABLE 1
#define CAP_LIMIT_DISABLE 0

class Timer
{
    public :
        double setFraps(int fraps, Uint32 beforeCountFraps, int delayFlag) ;

        void displayFraps(double deltaTime) ;

    private :
        double periodTicks ;
        double deltaTime ;
        int deltaTicks ;

} ;
```

Thus, CAP_LIMIT_ENABLE and CAP_LIMIT_DISABLE are just the flags to either enable or disable the capping of frame rate. If kept disabled, it would just return the delta time for the frame which may be used in motion or movement graphics. If enabled, it would cap the frame rate as well as, return the delta time for every frame.

Thus, we created a function called setFraps() that would require the frame rate, the ticks or, milliseconds from which we need to get the delay time (*at the start of the frame for capping frame rate*) and the flag which toggles the capping of the frame rate. It would return a double value for delta time and also cap the frame rate, if toggled as 1 or, CAP_LIMIT_ENABLE.

```
double Timer::setFraps(int fraps, Uint32 beforeCountFraps, int delayFlag )
{
    periodTicks = (1000/fraps) ;
    deltaTicks = SDL_GetTicks() - beforeCountFraps ;

    switch(delayFlag)
        {
            case CAP_LIMIT_DISABLE :
                if(deltaTicks/1000 < periodTicks)
                {
                    deltaTime = periodTicks - deltaTicks/1000 ;
                }
                break ;

            case CAP_LIMIT_ENABLE :
                if(deltaTicks/1000 < periodTicks)
                {
                    deltaTime = periodTicks - deltaTicks/1000 ;
                }
                SDL_Delay(deltaTime) ;
                break ;

            default :
                break ;
        }
    return deltaTime ;
}
```

SDL_GetTicks() gives us the amount of milliseconds already passed since the initialization of SDL2.0 library. SDL_Delay() delays the amount of time in milliseconds which is given as an argument.

Now, we made a function simple enough to just display the frame rate i.e. FPS for the screen which is just 1000/(delta time) since, the delta time was in milliseconds.

```
void Timer::displayFraps(double deltaTime)
{
    std::cout << "FPS : " << 1000/deltaTime << std::endl ;
}
```

2.4.5. 'Music' class:

Music can add essence to graphics and simulations if used in a proper way. We may or, may not require music and tones but, we decided to also create a class *Music* for handling chunks of music and rhythms. This class could have been expanded but, we kept it simple and straight forward for simple audio handling.

For this class, we did the same as other classes by declaring *Music* class and its functions but, this class is completely dependent upon SDL2_mixer library whose sole purpose is to introduce audio handling features in SDL2.0. This library was already included in *Launcher* class so, it is not required to include it again.

```
class Music
{
    public :

        static Mix_Chunk *loadMusic(const char *filename) ;

        static void playMusic(Mix_Chunk *chunk, int volume) ;

} ;
```

Mix_Chunk is just an API-defined data type which can hold audio data for different types of audio files like .wav, .mp3, .ogg, etc. But, it can only hold small size of audio files which is enough for our framework. So, if large audio files are desired, Mix_Music must be used replacing Mix_Chunk.

First, we created a function which takes the file path of audio tones as an argument to initialize mixer library and also load the music into Mix_Chunk data type which can later be operated using an audio channel. That audio channel has already been defined using Mix_OpenAudio() in the loadMusic() function where music files will be loaded alongside the activation of the audio channel to operate for further processes. It operates the channel in the stereo mode (double channeled mode) at a decent audio frequency (22050) with the maximum size of 7060 MB possible for the audio file. Though, SDL_audio library already pre-exists in the normal library of SDL2.0, it has a very weaker performance compared to SDL_mixer library which has better functions and optimizations than the other library.

```
Mix_Chunk *Music::loadMusic(const char *filename)
{
    Mix_Init(MIX_INIT_MP3) ;
    Mix_OpenAudio(22050,MIX_DEFAULT_FORMAT ,2, 7600) ;
    Mix_Chunk *chunk = Mix_LoadWAV(filename) ;
    bool playing ;
    if(chunk == nullptr)
    {
        std::cout << "\nMusic Loader Diagnosis -> " << std::endl ;
        std::cout << "Error loading the audio : " << filename << std::endl << "{Troubleshoot : "
        << Mix_GetError() << "}\n" << std::endl ;
    }
    else
    {
        std::cout << "Audio Channel Initialized ..." << std::endl ;
    }

    return chunk ;
}
```

This can also help in error handling and trouble-shooting since, it throws error if exists, on the console window.

Then, we made a simple function playMusic() that would play the music in the default format on the channel initialized by loadMusic() function. We can also set the volume of the audio channel from this function. This function requires Mix_Chunk type argument so that, it can play audio as per the audio bit data provided to it. If *Launcher* closes, then SDL2_mixer library also gets quitted freeing the memory taken by the audio data.

```
void Music::playMusic(Mix_Chunk *chunk, int volume)
{
    if(!Launcher::switchedOn)
    {
        Mix_FreeChunk(chunk) ;
        Mix_Quit() ;
    }
    else
    {
        Mix_VolumeChunk(chunk, volume) ;
        Mix_PlayChannel(-1, chunk, 0) ;
    }
}
```

2.4.6. UML Diagram Representation:

Launcher is basically an object to produce a graphical context that will be assisted by the helper classes which, in case of our project are: *Texture, Font, Music* and *Timer.* It would grab the functionalities from each of those classes by addressing scopes for the particular class in the main class which is, of course *Launcher.* The thing not to be confused about this project is that, it is a wrapper class or a framework for the ease of coding 2-D graphics for the programmer but, not exactly the user. The software that gets build out of this class would be the final product for the user.

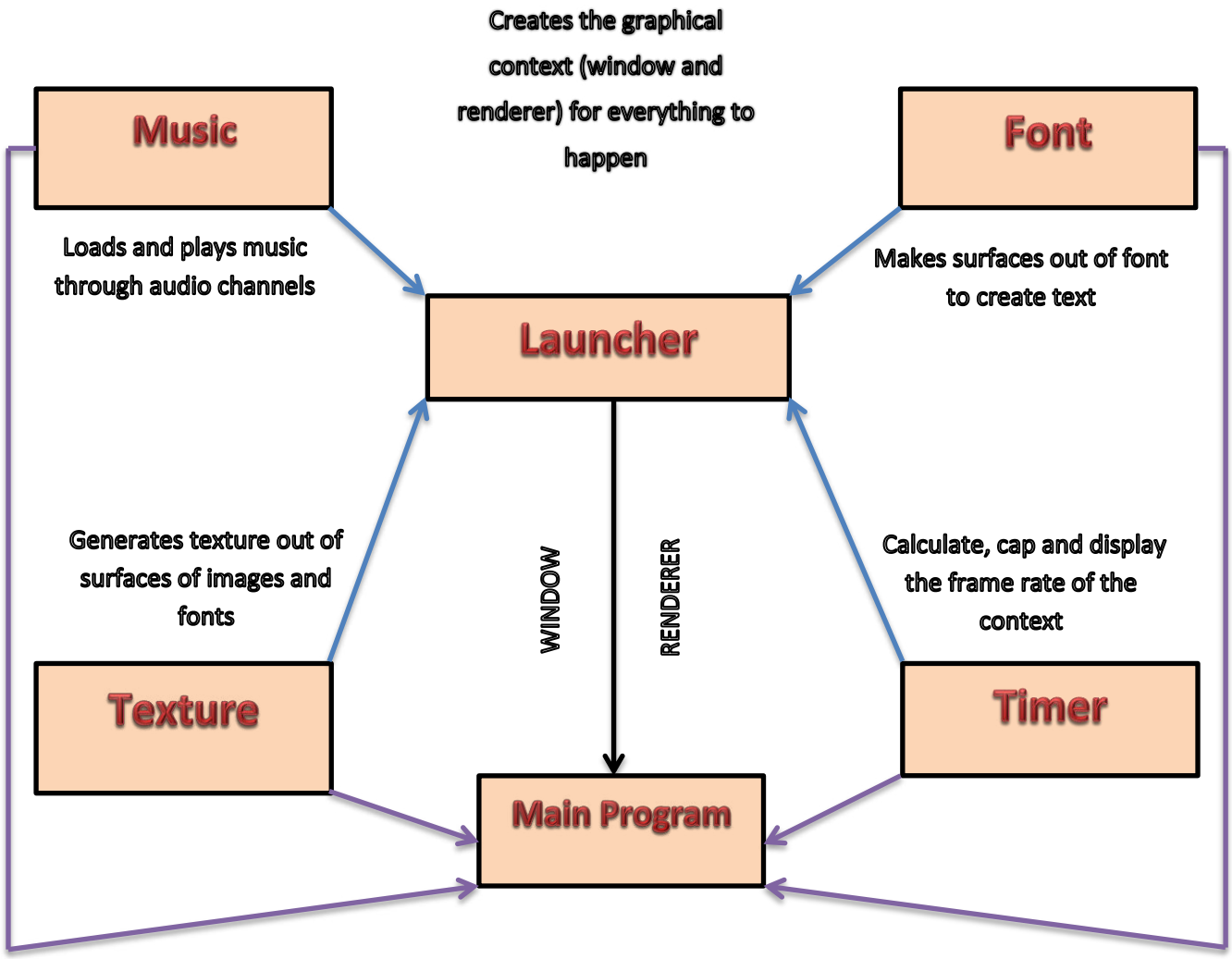


Fig: UML Diagram for the Framework

2.4.7. Making a sample program with the framework:

There is no better way to check if this program works than making a program out of it. So, we created a sample program which creates a simple window on the center of the screen with the dimension which we desire.

We created a source file to include *Launcher* class which is of course, the main class which holds all the other classes together. Then, in the main() function, we created *Launcher* and *Timer object* for their respective purposes in the program. We initialized the screen with required dimensions and title using initScreen() function keeping the fullscreen mode disabled. This automatically, initializes SDL2.0 as well as creates window and renderer for the program.

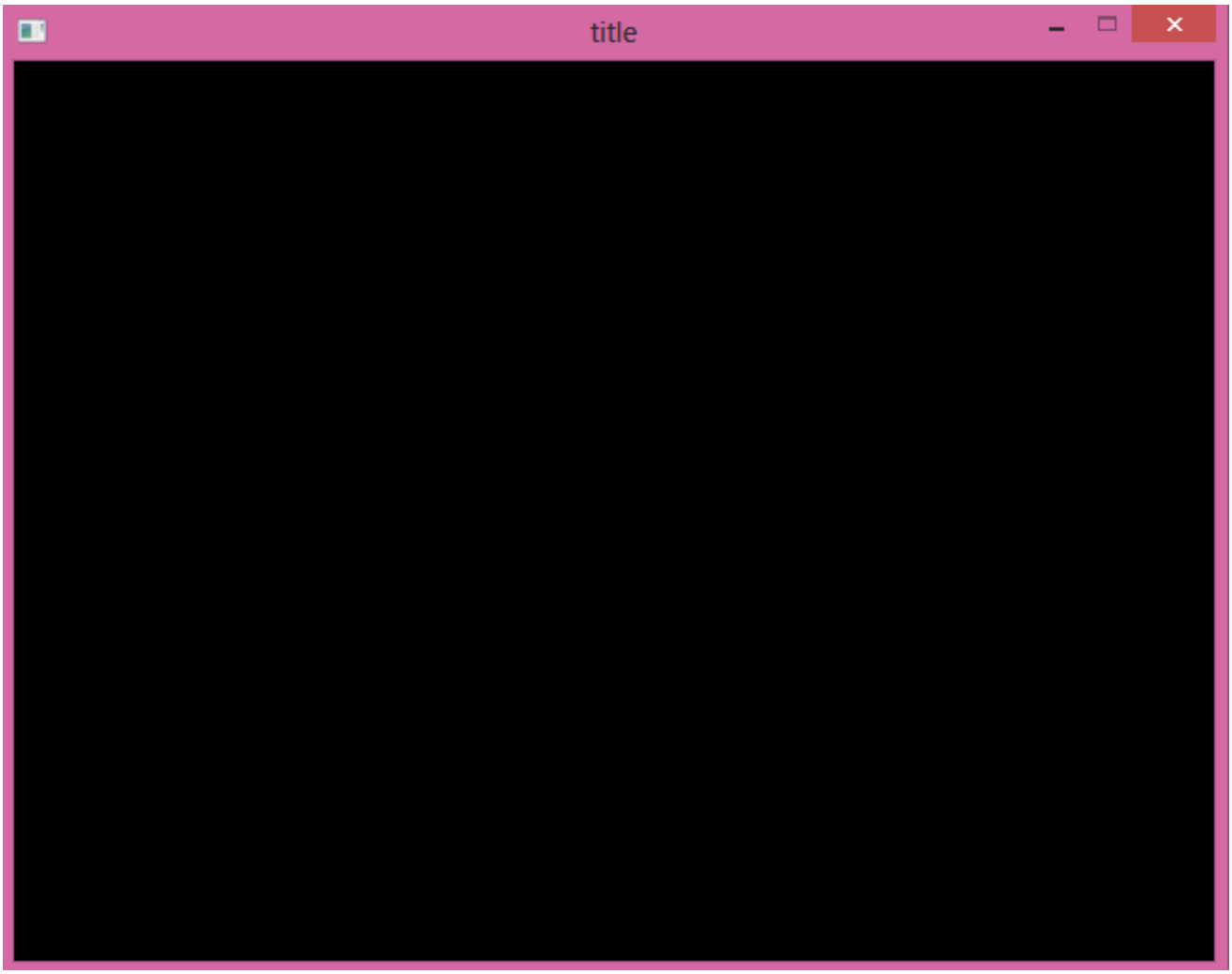
```
Launcher *launcher = nullptr ;
launcher = new Launcher() ;

Timer *timer = nullptr ;
timer = new Timer() ;

launcher->initScreen("title", 640 , 480, 0 ) ;
```

While initializing the screen, we already set a flag that allows generating frames continuously for the renderer till, the screen is quitted. Thus, within the frame, we handled our window events and even cleared the renderer.

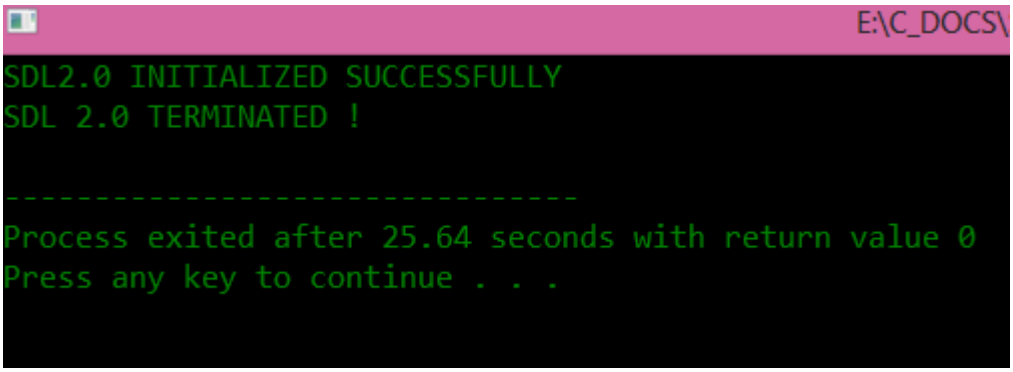
```
while(launcher->isRunning())
{
    Uint32 beforeTicks = SDL_GetTicks() ;

    launcher->handleEvents() ;
    launcher->updateScreen() ;
    launcher->renderScreen() ;

    double deltaTime = timer->setFraps(60, beforeTicks, CAP_LIMIT_ENABLE) ;
    timer->displayFraps(deltaTime) ;
}

launcher->quit() ;
```

Timer was just used to check if it's capping the frame rate to 60Hz (60 frames per second). If *Launcher* is quitted from events linked to I/O or display, *Launcher* will disable the running flag to quit the window and renderer.

We witnessed the window of 640*480 dimensions with title “title” as per the arguments provided in the code. Also, the screen was cleared with black color and presented on the screen.



Thus, the console window in this case would display dialogs declared in the *Launcher* class while executing the program. If ‘ESC’ key or quit button is pressed, it displays message referring to the termination of SDL2.0 library.



CHAPTER THREE: GAME DESIGN

3.1. SNAKE GAME DESIGN

3.1.1 Initialization:

First, the game is initialized by creating the field for the snake, along with the snake itself. Also, a food and a rune are generated at random places. Snake consists of a head and body. The head is a rectangle and the body is collection of certain number of segments (5 at the beginning) and each segment is a rectangle. In other words, the body is the array of rectangles. The *score* is set to zero and the *toxicity* is set to one hundred.

3.1.2 Movement:

We use *up, down, left* and *right* keys on the keyboard to move the snake up, down, left and right respectively. This is accomplished in following steps:

1. First, move the head rectangle in the direction of the pressed keys. For this, we have used SDL library to keep track of the pressed key and then we have altered the coordinates of the head rectangle according to the pressed key. In the following code snippet we have changed the value of Booleans *up, down, left, right* according to the pressed keys, so that we can use the Booleans later on for changing the coordinates of the head rectangle of snake.

```
if(Launcher::event.type == SDL_KEYDOWN)
{
    switch(Launcher::event.key.keysym.sym)
    {
        case SDLK_UP :
            left = right = false ;
            if(!down)
            {
                up = true ;
                down = false ;
            }
            else
            {
                down = true ;
                up = false ;
            }
            break ;

        case SDLK_DOWN :
            left = right = false ;
            if(!up)
            {
                down = true ;
                up = false ;
            }
            else
            {
                up = true ;
                down = false ;
            }
```

```
        case SDLK_LEFT :
            up = down = false ;
            if(!right)
            {
                left = true ;
                right = false ;
            }
            else
            {
                right = true ;
                left = false ;
            }
            break ;

        case SDLK_RIGHT :
            up = down = false ;
            if(!left)
            {
                right = true ;
                left = false ;
            }
            else
            {
                left = true ;
                right = false ;
            }
            break ;

        default :
            break ;

    }
```

2. To move the tail segments in the manner that they all seem to follow the tail segment in front of them, we have done following things.

- After moving the head, the previous coordinates of the head rectangle are given to first segment of the tail (The one closest to the head) but the previous coordinates of the first tail segment are stored as *prevx1* and *prevy1*.

Now we iterate from second to nth tail segment doing this process: For second tail segment, first its current coordinates are stored as *prevx2* and *prevy2* and its coordinates are now changed to *prevx1* and *prevy1* (previously stored first tail segment's coordinates).Now the values of *prevx1* and *prevy1* are made equal to *prevx2* and *prevy2*. Now ,the current coordinates of next (3rd) tail segment are stored as *prevx2* and *prevy2* and now it's coordinates are changed to *prevx1* and *prevy1*(previously stored coordinates of second tail segment).In this way tail segments follow the head whichever direction it goes. The following code snippet further clarifies:

```
prevX1 = tailRects[0].x ;
prevY1 = tailRects[0].y ;
tailRects[0].w = tailRects[0].h = PER_SEGMENT ;

for(int i=1; i<nth_tailPos; i++)
{
    if(right || left || up || down)
    {
        tailRects[0].x = headRect.x ;
        tailRects[0].y = headRect.y ;
        prevX2 = tailRects[i].x ;
        prevY2 = tailRects[i].y ;
        tailRects[i].x = prevX1 ;
        tailRects[i].y = prevY1 ;
        prevX1 = prevX2 ;
        prevY1 = prevY2 ;

        tailRects[i].w = tailRects[i].h = PER_SEGMENT ;
    }
```

3.1.3 Collision Detection:

After the movement we need to check for the collision of the head rectangle with different things.

3.1.3.1. Collision with wall:

Check if the coordinates of the head rectangle are matching with the coordinates of wall. If so, the snake dies and game over screen is shown.

```
{
    if(headRect.x < 0)
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.x = 0 ;
    }
    if(headRect.x > (SCR_W-headRect.w))
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.x = (SCR_W-headRect.w );
    }
    if(headRect.y < 0)
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.y = 0 ;
    }
    if(headRect.y > (SCR_H-headRect.h))
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.y = (SCR_H-headRect.h );
    }

}
```

As we can see in the above code snippet, if the snake collides with any wall, first it prevented from going out of window by keeping the coordinates constant. Moreover, we change the Boolean variable *alive* to *false* so that the snake dies.

3.1.3.2. Collision with food:

Check if the head rectangle is intersecting with the food rectangle. If so, increase the number of tail rectangles by 1. The score is increased by 1 and the toxicity is decreased by 1. Also, food is generated at another random coordinate.

```
if(SDL_HasIntersection(&h, &f))
{
    snakeBody->increaseTail() ;
    food->generateFood(FOOD_PIXEL*(rand()%(SCR_W/FOOD_PIXEL)),FOOD_PIXEL*(rand()%(SCR_H/FOOD_PIXEL))) ;
    Music::playMusic(get, 20) ;
}
```

As seen above, we detect the collision using in built function of SDL2 library called *SDL_HasIntersection()* and the head rectangle and food rectangle are passed as argument and we have used other functions defined in *body* and *food* class for accomplishing this task.

3.1.3.3. Collision with rune:

Check if the head rectangle is intersecting with the rune rectangle. If so, decrease the length by number of segments equal to toxicity. If new length of snake is less than 1, the snake dies and the game over screen is shown. As shown below, the tail is decreased according to the toxicity value.The toxicity value depends upon the score. The more the score, the less the toxicity (as shown in first code snippet). Again, *SDL_HasIntersection()* is used and the head and rune rectangles are passed as argument to detect the collision.

```
if(snakeBody->getScore()<25) snakeBody->updateToxicity(100) ;
if(snakeBody->getScore()>25 && snakeBody->getScore()<50) snakeBody->updateToxicity(50) ;
if(snakeBody->getScore()>50 && snakeBody->getScore()<75) snakeBody->updateToxicity(5) ;
if(snakeBody->getScore()>75 && snakeBody->getScore()<100) snakeBody->updateToxicity(4) ;
if(snakeBody->getScore()>100 && snakeBody->getScore()<150) snakeBody->updateToxicity(3) ;
if(snakeBody->getScore()>150) snakeBody->updateToxicity(2) ;
if(snakeBody->getScore()>25) Body::retardation = 2 ;
if(snakeBody->getScore()>50) Body::retardation = 1 ;
if(snakeBody->getScore()>100) Body::retardation = 0 ;
```

3.1.3.4. Collision with tail:

Check if the head rectangle is intersecting with any of the tail rectangle. If so, the snake dies and the game over screen is shown.

```
if(headRect.x == tailRects[i].x && headRect.y == tailRects[i].y)
{
    alive = false ;
    nth_tailPos = 0 ;
}
```

As shown above, the coordinates of head rectangle is compared with the coordinates of each of the tail rectangles by iterating ‘i’ from 1 to total number of tails. If any of the coordinates match, we just set the value of Boolean *alive* to *false* so that the snake dies.

```
if(SDL_HasIntersection(&h, &r))
{
    food->generateRune(-100,-100) ;

    if(snakeBody->getScore()<25) snakeBody->reduceTail(100) ;
    if(snakeBody->getScore()>25 && snakeBody->getScore()<50) snakeBody->reduceTail(50) ;
    if(snakeBody->getScore()>50 && snakeBody->getScore()<75) snakeBody->reduceTail(5) ;
    if(snakeBody->getScore()>75 && snakeBody->getScore()<100) snakeBody->reduceTail(4) ;
    if(snakeBody->getScore()>100 && snakeBody->getScore()<150) snakeBody->reduceTail(3) ;
    if(snakeBody->getScore()>150) snakeBody->reduceTail(2) ;

    Music::playMusic(get, 20) ;
}
```

3.2 PONG GAME DESIGN:

3.2.1 Initialization:

The game “PONG ! “ is initialized with a *field* for playing the game. Now , one *ball* is generated and two *paddles* are also generated to hit the ball. One of the paddle can be moved by the player and another paddle is to be moved by the CPU. Player paddle is displayed at the right of the screen whereas the CPU paddle is displayed at the left side of the screen. Both of the paddles are rectangles and also the ball is a small rectangle. Since circle was not available in SDL2 library, a small rectangle with equal sides was used. Also, two scoreboards are displayed which display how many lives the player and the CPU have remaining. They both have 3 lives in the beginning.

3.2.2. Paddle Movement:

The player paddle can be moved only vertically, keeping the x-coordinate constant. Player should move the paddle so that it hits the ball. To move the paddle, we have used SDL Events to keep track of the button pressed. The y-coordinate of the paddle is altered according to the button pressed by the player to move the paddle. If UP key is pressed the paddle moves upwards whereas the paddle moves downwards if DOWN key is pressed. But , it is prevented from going out of the border by using the second code snippet shown below.

```
if(Launcher::event.type == SDL_KEYDOWN)
{
    switch(Launcher::event.key.keysym.sym)
    {
        case SDLK_UP :
            paddleRect.y -= MOVE ;
            break ;

        case SDLK_DOWN :
            paddleRect.y += MOVE ;
            break ;

        default:
            break ;
    }
}
```

Where MOVE is the number of pixels to me moved in a single frame. (velocity in pixels/frame)

```
if(paddleRect.y<0+BORDER)
{
    paddleRect.y = 0+BORDER ;
}
if(paddleRect.y + paddleRect.h + BORDER> SCR_H)
{
    paddleRect.y = SCR_H-paddleRect.h - BORDER ;
}
```

Where, BORDER is the width of the border in pixels.

3.2.3. AI Implementation:

A simple AI was made to control the CPU paddle . This AI makes the paddle move towards the y-coordinate of the ball so that it can receive the ball and bounce it back to the player.

```
if(b.y > paddleRect.y+(paddleRect.h/2))
{
    paddleRect.y += MOVE ;
}
if(b.y < paddleRect.y+(paddleRect.h/2))
{
    paddleRect.y -= MOVE ;
}
```

Where ‘b’ is ball rectangle and MOVE is the velocity of paddle in pixels/frame.

3.2.4. Ball Movement and Collision Detection:

The ball bounces inside the game field when it strikes something (paddle or wall) in different manners which is explained below

3.2.4.1. Collision with walls :

When ball collides with the horizontal walls it just simply bounces back. For this we have made its velocity negative as soon as it hits the wall, which gives the bouncing effect. Also, if the ball collides with vertical left wall the computer score is deducted and if it collides with right vertical wall , the player score is deducted (since they were not able to receive the ball).

The wall collision is further explained by this code snippet.

```
{
    ballRect.x += velX ;
    ballRect.y += velY ;

    if(ballRect.x+BALL_W>=SCR_W)
    {
        velX =- velX ;
        deductCOM = true ;
    }
    if(ballRect.y+BALL_H>=SCR_H)
    {
        velY =- velY ;
    }
    if(ballRect.x <= 0)
    {
        velX =- velX ;
        deduct = true ;
    }
    if(ballRect.y <= 0)
    {
        velY =- velY ;
    }

}
```

3.2.4.2.Collision with paddles:

If the ball collides with the paddle , the ball needs to bounce back in an angle depeding upon the point where the ball hit the paddle. To accomplish this , first the angle with which the ball hit the paddle is determined according to the distance of the hitting point from the centre of the paddle, which is shown in the following code snippet.

```
if(SDL_HasIntersection(&paddleRect, &a))
{
    double relativeSite = (paddleRect.y+(paddleRect.h/2))-(a.y+(a.h/2)) ;
    double normalizeSite = relativeSite/(paddleRect.h) ;
    double angle = normalizeSite*(3*PI/2) ;
    return angle ;
}
```

Now, the ball is bounced back according to this angle using some trigonometry. Which is shown below:

```
if(checkCollision(l, b))
{
    double angleLeft = leftPaddle->getHitAngle(b) ;
    ball->updateVelocity(BALL_S*cos(angleLeft), BALL_S*sin(angleLeft)) ;
    leftPaddle->controlPaddle() ;
    Music::playMusic(bounce,20) ;
}

if(checkCollision(r, b))
{
    double angleRight = rightPaddle->getHitAngle(b) ;
    ball->updateVelocity(-BALL_S*cos(angleRight), -BALL_S*sin(angleRight)) ;
    leftPaddle->controlPaddle() ;
    Music::playMusic(bounce,20) ;
}
```

3.2.5 Score Deduction And Winner Declaration:

The score should be deducted when the ball hits the wall on player's side or CPU's side accordingly. For this, the flags *deduct and deductCOM* are altered as soon as the ball hits right and left wall respectively (Explained in **3.2.4.1**). Now using these flags, the score is deducted. Also the paddles are moved to center for new round and ball position is also reset.

```
if(ball->shouldDeductScore())
{
    leftPaddle->deductScore() ;
    leftPaddle->updatePosition(32 , SCR_H/2-PAD_H/2) ;
    rightPaddle->updatePosition(SCR_W-32-PAD_W/2 , SCR_H/2-PAD_H/2 ) ;
    ball->updateBallPosition(SCR_W/2-BALL_W/2, SCR_H/2-BALL_H/2) ;
    ball->updateVelocity(BALL_S/2, 0) ;
    ball->avoidPlay() ;
}
if(ball->AIfail())
{
    rightPaddle->deductScore() ;
    leftPaddle->updatePosition(32 , SCR_H/2-PAD_H/2) ;
    rightPaddle->updatePosition(SCR_W-32-PAD_W/2 , SCR_H/2-PAD_H/2 ) ;
    ball->updateBallPosition(SCR_W/2-BALL_W/2, SCR_H/2-BALL_H/2) ;
    ball->updateVelocity(BALL_S/2, 0) ;
    ball->avoidCOM() ;
}
```

o return the va

0. If player sco

er is announce

```
if(leftPaddle->getScore() == 0) gameOver = true ;
if(rightPaddle->getScore() == 0) win = true ;
```

CHAPTER FOUR: PROBLEM ANALYSIS

4.1. ALGORITHM:

The algorithms for Snake and Pong game are given below.

4.1.1. Snake Algorithm:

1. Display the gamefield and the snake body in the screen , set the *score* to zero and set the *toxicity* to one hundred. Display the values of *score* and *toxicity* in *Scoreboard* and *Toxicboard* respectively.
2. Move the snake head according the pressed key. i.e Move up,down,right or left if UP,DOWN,RIGHT or LEFT keys are pressed , respectively.
3. Make the tail segments follow the head (for first tail segment) or the tail segment in front of itself (for all other tail segments) by giving them the coordinates of the rectangle which they need to follow.
4. Check if the snake eats the food .If yes, increase the score .
5. Update the *toxicity* according to the *score* (the more the *score* the less the *toxicity* , explained in *3.1.3.3*).
6. Check if the snake eats the rune . If yes , deduct the number of tails equal to the *toxicity* value . If the new length of the snake is less than 0 , give *true* value to *gameover boolean*.
7. Check if the snake head collides with the wall. If yes , give the value *true* to *gameover boolean*.
8. Check the collision of snake head with each of the tail rectangles. If there is a collision , give *true* value to *gameover boolean*.
9. Check the value of *gameover boolean*. If *true* , display the gameover screen .

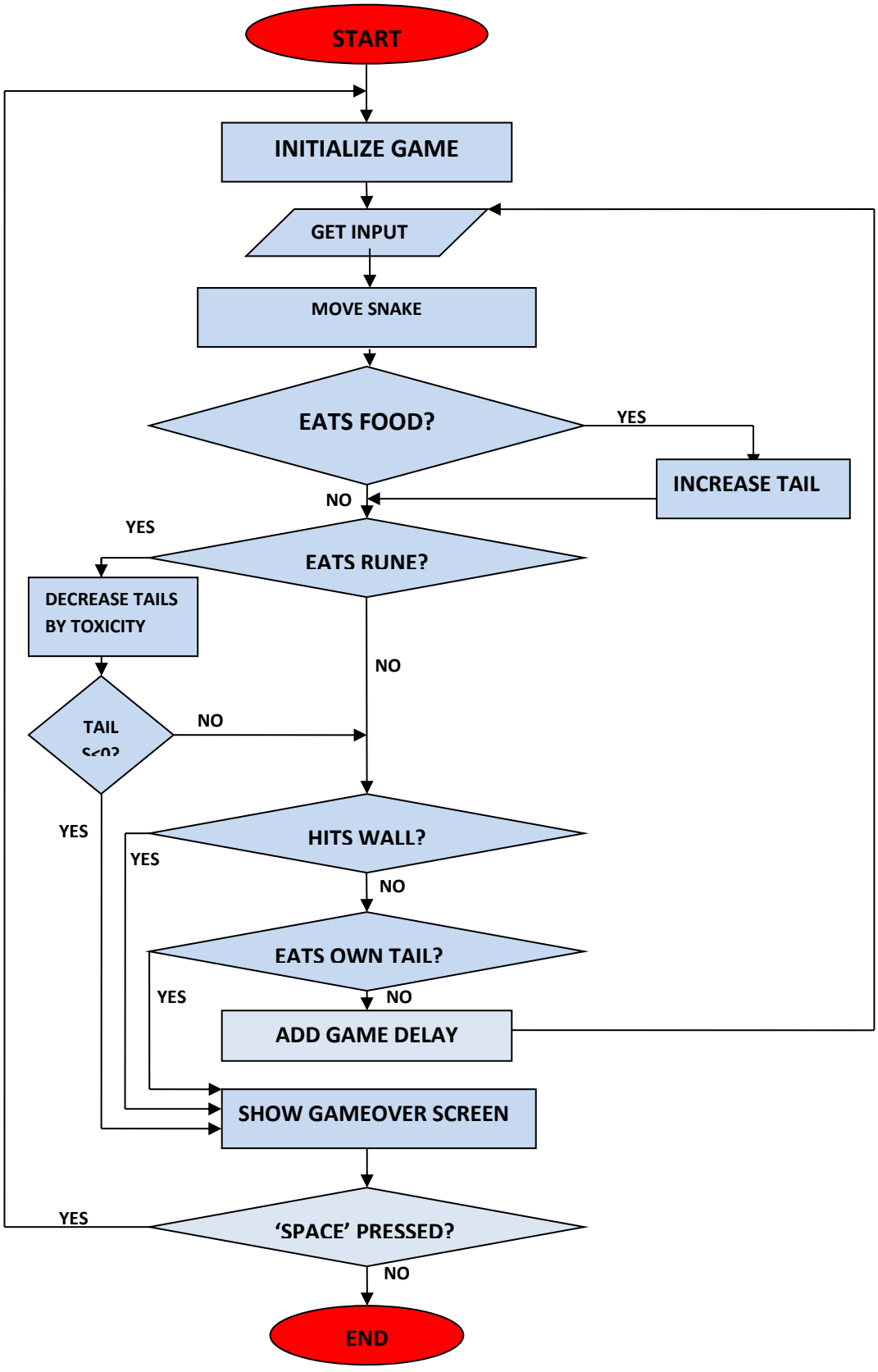
10. Check if the user presses SPACE key. If yes , repeat the steps from step-1. If no , quit the game.

4.1.2 Pong! Algorithm:

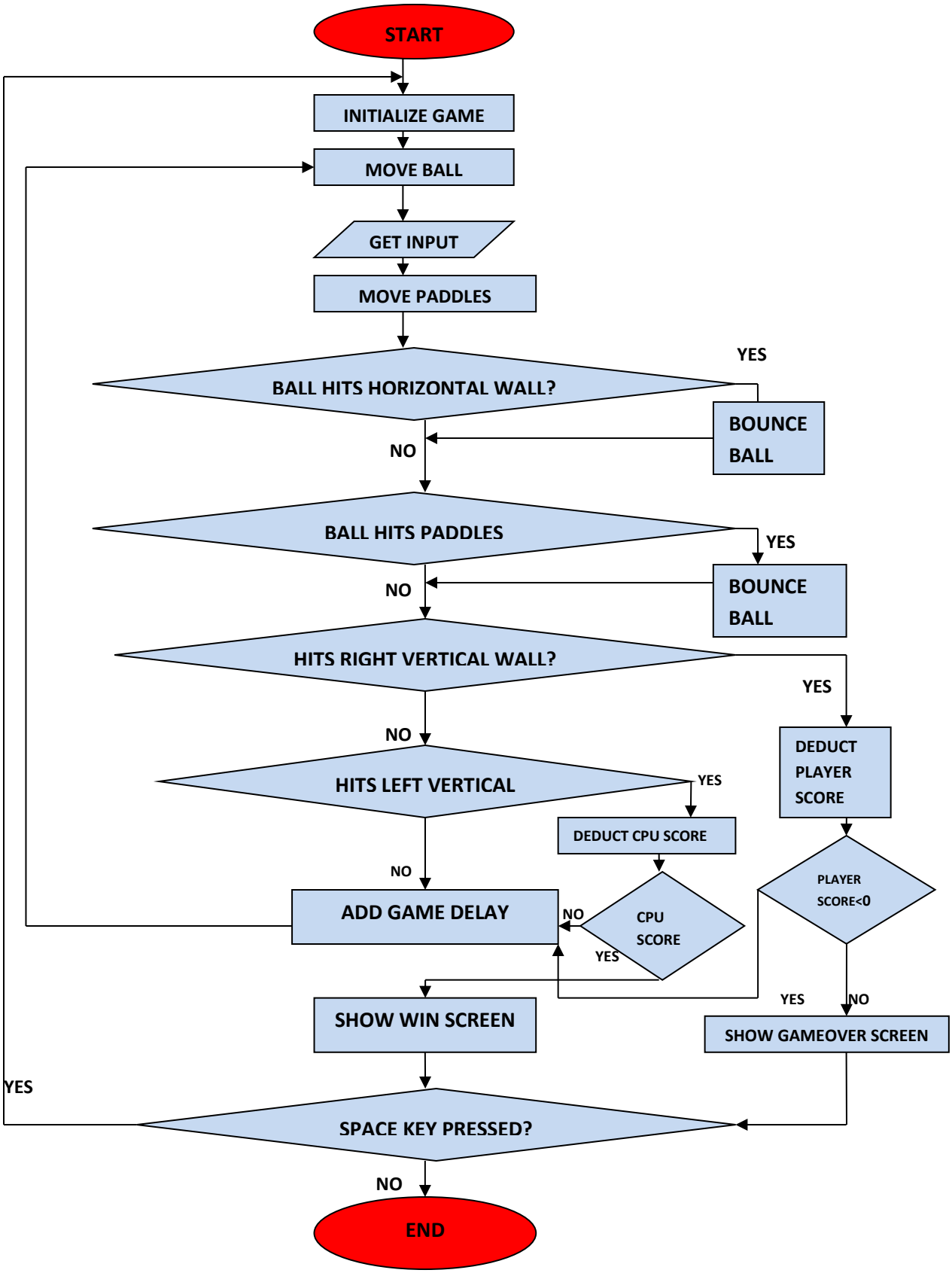
1. Display the game field, the paddles, the ball and the scoreboards. Set both the *score* values to three.
2. Move the ball, move the player paddle up or down according to the pressed keys. Move the CPU paddle towards the *y-position* of the ball.
3. Check if the ball collides with horizontal walls. If yes, make the direction of *y-velocity* opposite.
4. Check if the ball collides with the paddles. If yes, bounce the ball back at an angle depending upon the distance of point of contact of ball and paddle from the centre of the paddle(explained in 3.2.4.2).
5. Check if the ball collides with the vertical walls. If the ball collides with the right vertical wall , deduct the *player score* and if the ball collides with the left wall, deduct the *CPU score* by 1 and reset the positions of paddles and the ball.
6. Check if either of the scores(*CPU score* and *Player score*) is *zero*.If *CPU score* is *zero* , give *true* value to *win Boolean*.If *Player score* is *zero* , give *true* value to *gameover Boolean*.
7. Check the values of *gameover* and *win* . If both are *false* , continue . If *gameover* is *true* , display the *gameover message*. If *win* is *true* , display the *win message* . After this, give *false* value to both these *Booleans*.
8. If the *gameover* or *win message* is displayed , check if the user presses *SPACE* key. If yes , restart the game from *step-1*. Otherwise, quit the game.

4.2 Flowchart:

4.2.1 Snake Flowchart:



4.2.2 PONG! Flowchart:



CHAPTER 5

CODING AND TESTING OF FRAMEWORK FOR GAMES

Since, we have already developed our framework, now it's time to implement it to actually make 2-D graphics. There is no better way to do it rather than by making classic 2-D games out of it. Thus, we will test the potential of our framework by making two classic games: '*Snake'* and '*Pong!'.* We already analyzed and elaborated the approach and technique to execute these two games before in 'Chapter 3: *Problem Analysis*' so, this only showcases the code implementation of these programs.

5.1. SNAKE:

In this classic 2-D Game, we move a snake that is slithering in a void arena where constantly food gets generated which can be consumed to increase its length until it's dead. The snake will only lose its life if it bites its own body or, if it collides with the boundary of the arena i.e. of course, the whole screen.

5.1.1. Making 'Body' Class:

First, we created a *Body* class at which we would make a snake's body out of array of rectangles. We linked our framework class i.e. *Launcher* to the header file so that we already create a graphical context for the program. All the required macros, variables and functions were declared in the header.

We needed a way to actually create and draw a snake's body on the screen so that, we can move it using keyboard keys easily. Then, during collision with any food type object i.e. another rectangle, it can detect it and increase its tail length. We introduced runes in the game which have variable toxicity that can either kill the snake or aid the snake if its length is too long and if it's less toxic. We have already discussed about the basics of how this game works in every explanatory way in 'Chapter 3: *Problem Analysis*'. Thus, the source code in C++ to create this sort of class is shown here:

(Note: *The functions and variables have pretty much the same name for what it is used for !*)

Body.hpp

```
#define MAX 200
#define PER_SEGMENT 10

#define FONT_SIZE 20

class Body
{
    public :

        Body() ;
        ~Body() ;

        void createSnakeBody(int x, int y, int w, int h) ;
        void displaySnakeBody(Uint8 r, Uint8 g, Uint8 b) ;
        void createScoreBoard(int x, int y, int w, int h) ;
        void displayScoreBoard(Uint8 r=255, Uint8 g=255, Uint8 b=255) ;
        void createToxicBoard(int x, int y, int w, int h) ;
        void displayToxicBoard(Uint8 r=255, Uint8 g=255, Uint8 b=255) ;
        void controlSnakeBody() ;
        SDL_Rect getHead() ;
        void updatePositionAndTail() ;
        void stayWithInTheScreen() ;
        void surpassScreen() ;
        void reduceTail(int n) ;
        bool isSnakeAlive() ;
        void kill() ;
        void showMyScore(int x, int y, int w, int h) ;
        void increaseTail() ;
        int getScore() ;
        void clean() ;
        void reset(int x, int y, int w, int h) ;
        int getTime() ;
        void displayTimer(int x, int y, int w, int h) ;
        void updateToxicity(int n) ;
        static int retardation ;
```

```
    private :

        SDL_Rect headRect, tailRects[MAX] ;
        int nth_tailPos=5 ;
        bool alive ;
        int prevX1, prevY1, prevX2, prevY2 ;
        bool up, down, left, right, stop ;
        int score ;

        SDL_Surface *sc , *timer, *toxic, *final, *over ;
        SDL_Texture *scoreTexture, *timerTexture , *gameOverTexture, *finalTexture, *toxicTexture;
        TTF_Font *font = Font::loadFont("rsrc/AGENCYR.ttf", FONT_SIZE) ;
        SDL_Rect scoreBoard, timerRect, gameOverRect, finalRect, toxicBoard ;
        std::string scoreText, timerText, scoreFinalText, gameOverText , toxicText;
        int secondsF=0 , seconds=0 ;
        int toxicity ;
} ;
```

Body.cpp

```cpp
#include "Body.hpp"

Body::Body()
{}
Body::~Body()
{}

void Body::clean()
{
    Font::close(font) ;
    SDL_DestroyTexture(scoreTexture) ;
    SDL_DestroyTexture(timerTexture) ;
    SDL_DestroyTexture(gameOverTexture) ;
    SDL_DestroyTexture(finalTexture) ;
    SDL_DestroyTexture(toxicTexture) ;
}

void Body::reduceTail(int n)
{
    nth_tailPos -= n ;
}

void Body::updateToxicity(int n)
{
    toxicity = n ;
}

void Body::kill()
{
    alive = false ;
}

void Body::createToxicBoard(int x, int y, int w, int h)
{
    toxicBoard = {x, y, w, h} ;
}
```

```cpp
void Body::displayToxicBoard(Uint8 r, Uint8 g, Uint8 b)
{
    SDL_Color a = {r,g,b} ;
    toxicText = "TOXICITY : " + std::to_string(toxicity) ;
    toxic = Font::getSurface(font, toxicText.c_str(), a) ;
    toxicTexture = Texture::createFromSurface(toxic) ;
    Texture::free_surface(toxic) ;
    SDL_RenderCopy(Launcher::renderer, toxicTexture, NULL, &toxicBoard) ;
}

void Body::createSnakeBody(int x, int y, int w, int h)
{
    headRect = {x,y,w,h} ;
    alive = true ;
    score = 0 ;
    secondsF = 0 ;
    seconds = 0 ;
    for(int i=0; i<nth_tailPos; i++)
    {
        tailRects[i] = {headRect.x-(PER_SEGMENT*(i+1)), headRect.y, PER_SEGMENT, PER_SEGMENT} ;
    }
}

void Body::reset(int x, int y, int w, int h)
{
    alive = true ;
    score = 0 ;
    secondsF = 0 ;
    seconds = 0 ;
    nth_tailPos = 5 ;
    Body::retardation = 3 ;
    headRect = {x,y,w,h} ;
}
```

```cpp
SDL_Rect Body::getHead()
{
    return headRect ;
}

void Body::displaySnakeBody(Uint8 r, Uint8 g, Uint8 b)
{
    for(int i=0; i<nth_tailPos; i++)
    {
        SDL_SetRenderDrawColor(Launcher::renderer, 255,255,0,255) ;
        SDL_RenderFillRect(Launcher::renderer, &headRect) ;
        SDL_SetRenderDrawColor(Launcher::renderer, r, g, b, 255) ;
        SDL_RenderFillRect(Launcher::renderer, &tailRects[i]) ;
    }
}

void Body::createScoreBoard(int x, int y, int w, int h)
{
    scoreBoard = {x, y, w, h} ;
}

void Body::displayScoreBoard(Uint8 r, Uint8 g, Uint8 b)
{
    SDL_Color a = {r,g,b} ;
    scoreText = "COINS : " + std::to_string(score) ;
    sc = Font::getSurface(font, scoreText.c_str(), a) ;
    scoreTexture = Texture::createFromSurface(sc) ;
    Texture::free_surface(sc) ;
    SDL_RenderCopy(Launcher::renderer, scoreTexture, NULL, &scoreBoard) ;
}

int Body::getScore()
{
    return score+(seconds/10) ;
}
```

```cpp
void Body::displayTimer(int x, int y, int w, int h)
{
    timerRect = {x, y, w, h} ;
    if(alive)
    {
        secondsF += 1 ;
        if(secondsF > 59-(Body::retardation*10))
        {
            int temp = secondsF/(60-(Body::retardation*10)) ;
            seconds += temp ;
            secondsF = secondsF%(60-(Body::retardation*10)) ;
        }
    }
    timerText  = "TIME : " + std::to_string(seconds) + " s ";
    SDL_Color color = {255,255,255} ;
    timer = Font::getSurface(font, timerText.c_str(), color) ;
    timerTexture = Texture::createFromSurface(timer) ;
    Texture::free_surface(timer) ;
    SDL_RenderCopy(Launcher::renderer, timerTexture, NULL, &timerRect) ;
}
```

```cpp
bool Body::isSnakeAlive()
{
    return alive ;
}

void Body::surpassScreen()
{
    if(headRect.x < 0)
    {
        headRect.x = (SCR_W-headRect.w ) ;
    }
    if(headRect.x > (SCR_W-headRect.w))
    {
        headRect.x = 0 ;
    }
    if(headRect.y < 0)
    {
        headRect.y = (SCR_H-headRect.h ) ;
    }
    if(headRect.y > (SCR_H-headRect.h))
    {
        headRect.y = 0 ;
    }
}

int Body::getTime()
{
    return seconds ;
}
```

```cpp
void Body::stayWithInTheScreen()
{
    if(headRect.x < 0)
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.x = 0 ;
    }
    if(headRect.x > (SCR_W-headRect.w))
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.x = (SCR_W-headRect.w );
    }
    if(headRect.y < 0)
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.y = 0 ;
    }
    if(headRect.y > (SCR_H-headRect.h))
    {
        alive = false ;
        nth_tailPos = 0 ;
        headRect.y = (SCR_H-headRect.h );
    }

}

void Body::increaseTail()
{
    nth_tailPos += 1 ;
    score += 1 ;
}
```

```cpp
void Body::showMyScore(int x, int y, int w, int h)
{
    gameOverRect = {x,y-h,w,h} ;
    finalRect = {x-50,y,w*3,h} ;
    gameOverText  = "    GAME OVER " ;
    scoreFinalText = "YOUR SCORE : COINS + (TIME/10) ~ " + std::to_string(score+(seconds/10)) ;
    SDL_Color a = {255,0,0} ;
    SDL_Color b = {0,0,255} ;
    over = Font::getSurface(font, gameOverText.c_str(), a) ;
    final = Font::getSurface(font, scoreFinalText.c_str(), b) ;
    gameOverTexture = Texture::createFromSurface(over) ;
    finalTexture = Texture::createFromSurface(final) ;
    Texture::free_surface(over) ;
    Texture::free_surface(final) ;
    SDL_RenderCopy(Launcher::renderer, gameOverTexture, NULL, &gameOverRect) ;
    SDL_RenderCopy(Launcher::renderer, finalTexture, NULL, &finalRect) ;
}

void Body::updatePositionAndTail()
{
    if(up && !down)
    {
        headRect.y -= (PER_SEGMENT) ;
    }
    if(down && !up)
    {
        headRect.y += (PER_SEGMENT) ;
    }
    if(left && !right)
    {
        headRect.x -= (PER_SEGMENT) ;
    }
    if(right && !left)
    {
        headRect.x += (PER_SEGMENT);
    }
```

```cpp
    //Main Algorithm

    prevX1 = tailRects[0].x ;
    prevY1 = tailRects[0].y ;
    tailRects[0].w = tailRects[0].h = PER_SEGMENT ;

    for(int i=1; i<nth_tailPos; i++)
    {
        if(right || left || up || down)
        {
            tailRects[0].x = headRect.x ;
            tailRects[0].y = headRect.y ;
            prevX2 = tailRects[i].x ;
            prevY2 = tailRects[i].y ;
            tailRects[i].x = prevX1 ;
            tailRects[i].y = prevY1 ;
            prevX1 = prevX2 ;
            prevY1 = prevY2 ;

            tailRects[i].w = tailRects[i].h = PER_SEGMENT ;
        }

        if(headRect.x == tailRects[i].x && headRect.y == tailRects[i].y)
        {
            alive = false ;
            nth_tailPos = 0 ;
        }
    }

    if(nth_tailPos <= 1)
    {
        alive = false ;
    }
}
```

```
void Body::controlSnakeBody()
{

    if(Launcher::event.type == SDL_KEYDOWN)
    {
        switch(Launcher::event.key.keysym.sym)
        {
            case SDLK_UP :
                left = right = false ;
                if(!down)
                {
                    up = true ;
                    down = false ;
                }
                else
                {
                    down = true ;
                    up = false ;
                }
                break ;

            case SDLK_DOWN :
                left = right = false ;
                if(!up)
                {
                    down = true ;
                    up = false ;
                }
                else
                {
                    up = true ;
                    down = false ;
                }

                break ;
```

```
            case SDLK_LEFT :
                up = down = false ;
                if(!right)
                {
                    left = true ;
                    right = false ;
                }
                else
                {
                    right = true ;
                    left = false ;
                }
                break ;

            case SDLK_RIGHT :
                up = down = false ;
                if(!left)
                {
                    right = true ;
                    left = false ;
                }
                else
                {
                    left = true ;
                    right = false ;
                }
                break ;

            default :
                break ;

        }
    }
}
```

5.1.2. Making 'Food' Class:

Food class is just prepared for an easy objective which is to generate a new food every time a snake eats one. But, in this game prepared from our framework, we switched food with coins that flicker due to animation sprites used in the class. We have also introduced a new consumable in this game called rune which if eaten, will decrease the length of snake as per the toxicity rate given in lower-left corner of the screen. Basically, in this class, we rendered animation sheets on top of the rectangles that disappear and randomly generate which if found intersecting with the array of rectangles of snake would eliminate rectangle of last index i.e. tail of the snake. This class uses vectors of pair of sprite co-ordinates which shifts the position of the source image to clip the rendering texture so that, it seems like transition in time to create animation effects for coin and runes in this game.

Food.hpp

```
#define FOOD_PIXEL 20

class Food
{
    public :

        Food() ;
        ~Food() ;
        void generateFood(int x, int y) ;
        SDL_Rect getFood() ;
        void displayFood() ;
        void generateRune(int x, int y) ;
        SDL_Rect getRune() ;
        void displayRune() ;
        void clean() ;

    private :

        SDL_Rect foodRect, sourceRect, runeRect, sRect ;
        SDL_Texture *coinTexture = Texture::loadTexture("rsrc/Coin.png") ;
        std::vector<std::pair<int,int>> coinSprites = {{0,0},{10,0},{20,0},{30,0}} ;
        SDL_Texture *runeTexture = Texture::loadTexture("rsrc/Rune.png") ;
        std::vector<std::pair<int,int>> runeSprites = {{0,0},{16,0},{32,0},{48,0}} ;
} ;
```

Food.cpp

```cpp
#include "Food.hpp"

Food::Food()
{

}

Food::~Food()
{

}

void Food::clean()
{
    SDL_DestroyTexture(coinTexture) ;
    SDL_DestroyTexture(runeTexture) ;
}

void Food::generateFood(int x, int y)
{
    foodRect={x,y,FOOD_PIXEL,FOOD_PIXEL} ;
}

void Food::generateRune(int x, int y)
{
    runeRect={x,y,FOOD_PIXEL,FOOD_PIXEL} ;
}
```

```cpp
SDL_Rect Food::getFood()
{
    return foodRect ;
}

SDL_Rect Food::getRune()
{
    return runeRect ;
}

void Food::displayRune()
{
    int timeR = (SDL_GetTicks()/100%4) ;
    sRect.x = runeSprites[timeR].first ;
    sRect.y = runeSprites[timeR].second ;
    sRect.w = sRect.h = 16 ;
    SDL_RenderCopy(Launcher::renderer, runeTexture, &sRect, &runeRect) ;
}

void Food::displayFood()
{
    int timeF = (SDL_GetTicks()/100%4) ;
    sourceRect.x = coinSprites[timeF].first ;
    sourceRect.y = coinSprites[timeF].second ;
    sourceRect.w = sourceRect.h = 10 ;
    SDL_RenderCopy(Launcher::renderer, coinTexture, &sourceRect, &foodRect) ;
}
```

5.1.3. Making ‘Snake’ Class:

This class is the main class for the game whose object produces the game content for the *Launcher* and holds together the other two classes. In this class, we simply made simple functions that got assembled by the features of *Body* which produces snake body which can be controlled through keyboard and *Food* which generates coins and runes for the snake to fetch and increase its score. All the UIs and textures are implemented through the renderer after being passed to the *Launcher* class as, it is the class which created the window and the renderer. This class also needs to be inter-linked with the *Launcher* class so that, this class would be an object that creates the instance of the game we require to run.

Snake.hpp

```
#include "Body.hpp"
#include "Food.hpp"

class Snake
{
    public :

        Snake() ;
        ~Snake() ;

        void createGame() ;

        void displayGame() ;

        void updateGame() ;

        void quitGame() ;

        bool shouldGamePause() ;

    private :

        Mix_Chunk *get = Music::loadMusic("rsrc/goodypop.wav") ;
        bool stop ;
} ;
```

Snake.cpp

```cpp
#include "Snake.hpp"

Body *snakeBody = nullptr ;
Food *food = nullptr ;
int Body::retardation = 3 ;

Snake::Snake()
{}
Snake::~Snake()
{}

bool Snake::shouldGamePause()
{
    return stop ;
}

void Snake::createGame()
{
    snakeBody = new Body() ;
    food = new Food() ;
    snakeBody->createSnakeBody(SCR_W/2-PER_SEGMENT/2, SCR_H/2-PER_SEGMENT/2, PER_SEGMENT, PER_SEGMENT) ;
    snakeBody->createScoreBoard(0,0,100,20) ;
    snakeBody->createToxicBoard(0,SCR_H-20,120,20) ;
    srand(time(0)) ;
    food->generateFood(FOOD_PIXEL*(rand()%(SCR_W/FOOD_PIXEL)),FOOD_PIXEL*(rand()%(SCR_H/FOOD_PIXEL))) ;
    food->generateRune(FOOD_PIXEL*(rand()%(SCR_W/FOOD_PIXEL)),FOOD_PIXEL*(rand()%(SCR_H/FOOD_PIXEL))) ;
    stop = false ;
}
```

```cpp
void Snake::displayGame()
{
    snakeBody->updatePositionAndTail() ;
    snakeBody->displayScoreBoard() ;
    snakeBody->displayToxicBoard() ;
    snakeBody->displayTimer(0,20,120,20) ;
    snakeBody->displaySnakeBody(0,255,0) ;
    food->displayRune() ;
    food->displayFood() ;
    if( snakeBody->getTime() != 0 && (snakeBody->getTime())%15 == 0)
    {
        food->generateRune(FOOD_PIXEL*(rand()%(SCR_W/FOOD_PIXEL)),FOOD_PIXEL*(rand()%(SCR_H/FOOD_PIXEL))) ;
    }
    if(!snakeBody->isSnakeAlive())
    {
        stop = true ;
        snakeBody->showMyScore(SCR_W/2-200,SCR_H/2-30,200,30) ;
        std::cout << "PRESS 'SPACE' TO RESTART" << std::endl ;
        if(Launcher::event.type == SDL_KEYDOWN && Launcher::event.key.keysym.sym == SDLK_SPACE)
        {
            std::cout << "NEW GAME IN 3 SECONDS " << std::endl ;
            std::cout << "LAST SCORE : " << snakeBody->getScore() << std::endl ;
            SDL_Delay(3000) ; // 3 seconds delay : 3,2,1 - Go !
            snakeBody->reset(SCR_W/2-PER_SEGMENT/2, SCR_H/2-PER_SEGMENT/2, PER_SEGMENT, PER_SEGMENT) ;
            stop = false ;
        }}
}
```

```
void Snake::updateGame()
{
    snakeBody->controlSnakeBody() ;
    SDL_Rect h = snakeBody->getHead() ;
    SDL_Rect f = food->getFood() ;
    SDL_Rect r = food->getRune() ;
    if(SDL_HasIntersection(&h, &f))
    {
        snakeBody->increaseTail() ;
        food->generateFood(FOOD_PIXEL*(rand()%(SCR_W/FOOD_PIXEL)),FOOD_PIXEL*(rand()%(SCR_H/FOOD_PIXEL))) ;
        Music::playMusic(get, 20) ;
    }
    if(snakeBody->getScore()<25) snakeBody->updateToxicity(100) ;
    if(snakeBody->getScore()>25 && snakeBody->getScore()<50) snakeBody->updateToxicity(50) ;
    if(snakeBody->getScore()>50 && snakeBody->getScore()<75) snakeBody->updateToxicity(5) ;
    if(snakeBody->getScore()>75 && snakeBody->getScore()<100) snakeBody->updateToxicity(4) ;
    if(snakeBody->getScore()>100 && snakeBody->getScore()<150) snakeBody->updateToxicity(3) ;
    if(snakeBody->getScore()>150) snakeBody->updateToxicity(2) ;
    if(snakeBody->getScore()>25) Body::retardation = 2 ;
    if(snakeBody->getScore()>50) Body::retardation = 1 ;
    if(snakeBody->getScore()>100) Body::retardation = 0 ;
    if(SDL_HasIntersection(&h, &r)){
        food->generateRune(-100,-100) ;
        if(snakeBody->getScore()<25) snakeBody->reduceTail(100) ;
        if(snakeBody->getScore()>25 && snakeBody->getScore()<50) snakeBody->reduceTail(50) ;
        if(snakeBody->getScore()>50 && snakeBody->getScore()<75) snakeBody->reduceTail(5) ;
        if(snakeBody->getScore()>75 && snakeBody->getScore()<100) snakeBody->reduceTail(4) ;
        if(snakeBody->getScore()>100 && snakeBody->getScore()<150) snakeBody->reduceTail(3) ;
        if(snakeBody->getScore()>150) snakeBody->reduceTail(2) ;
        Music::playMusic(get, 20) ;
    }
    snakeBody->stayWithInTheScreen() ;
}

void Snake::quitGame()
{
    snakeBody->kill() ;
    snakeBody->clean() ;
    food->clean() ;
    Mix_FreeChunk(get) ;
}
```

5.1.4. Modification for '*Launcher*' Class:

In *Launcher* class, we need to start the instance of the game by defining an object of *Snake* class to link it with the renderer and the window. We just introduced all the functionalities of the game to the *Launcher* class so that, the game would be fetched by the main function of the project file.

```
Snake *snake = nullptr ;
snake = new  Snake() ;
snake->createGame() ;
while(launcher->isRunning)
{
        snake->displayGame() ;
        snake->updateGame() ;
}
        snake->quitGame() ;
```

5.1.5. UML Diagram (SNAKE):

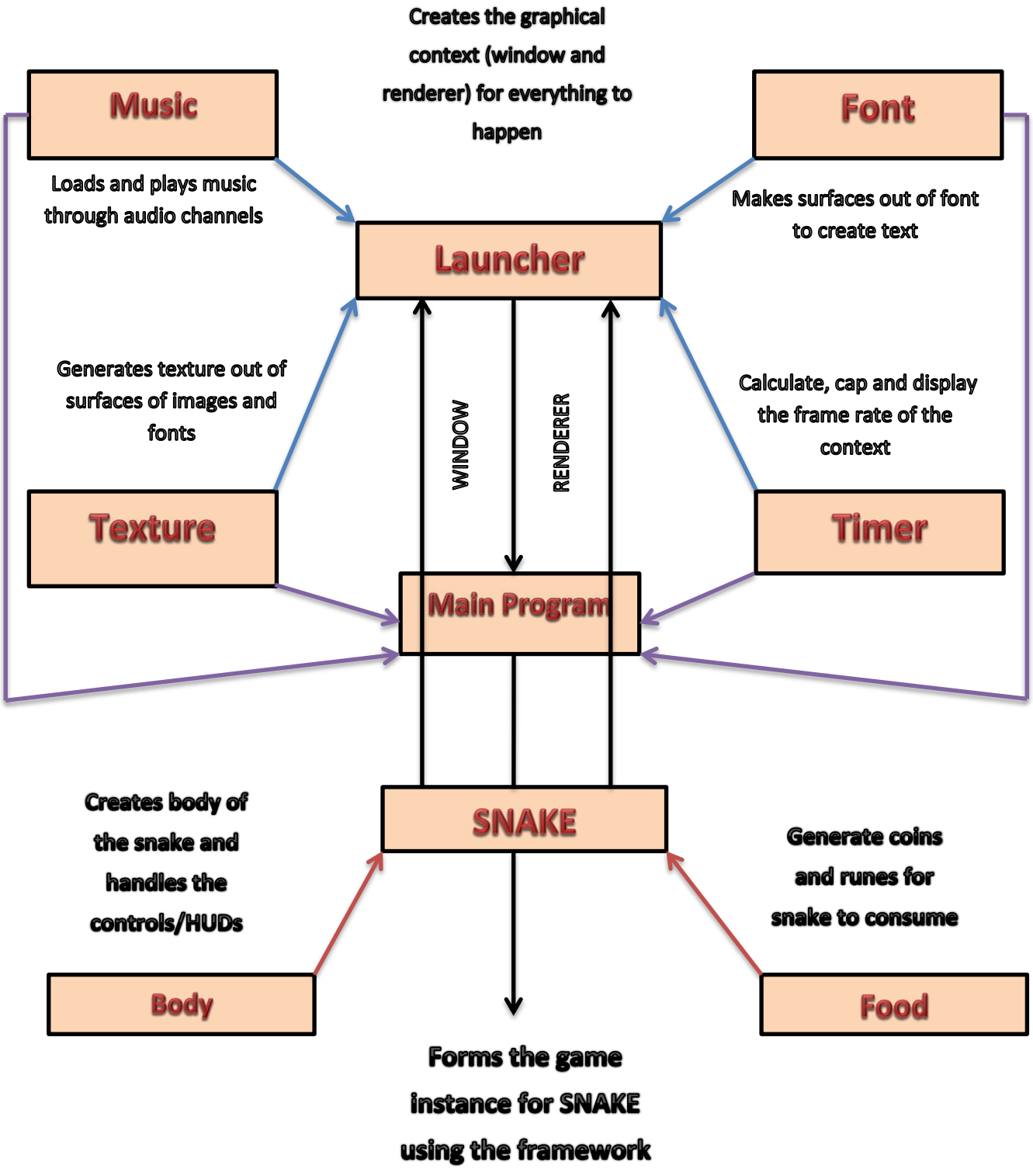
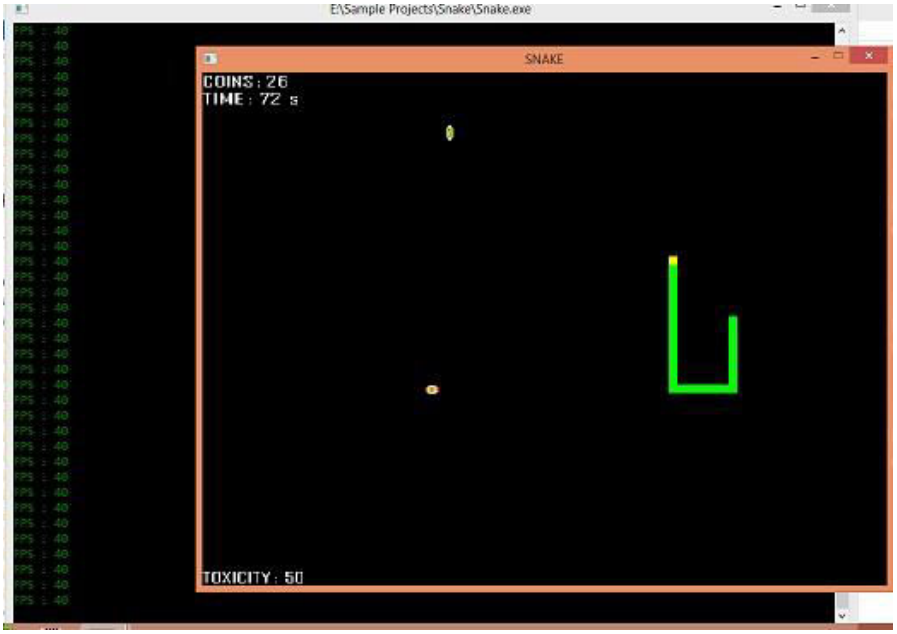


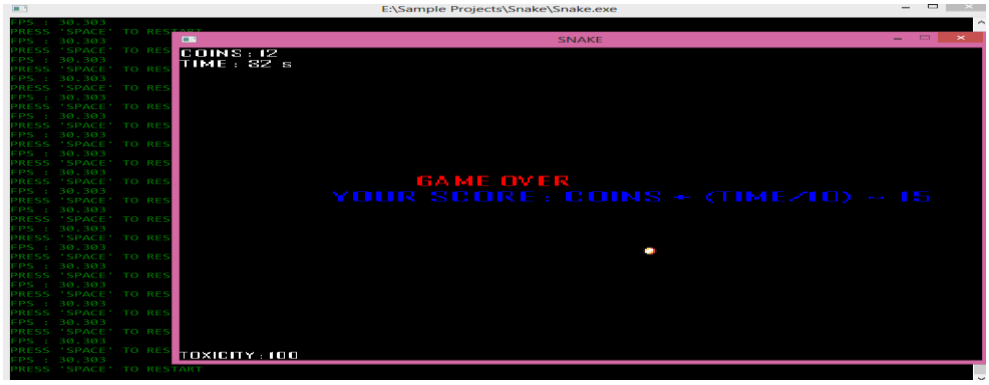
Fig: UML Diagram of the framework along with the game

5.1.6. Testing the game:

This game was capped to 30-60 FPS as per the progression of the score in the game. The HUDs and the counters/timers in the window were projecting the status of game as expected. The audio channel and the font renderer worked perfectly fine. The movements through the keyboard keys were visible and excellently working. The game seemed very fluid and no major glitches were found at all.



Also, the collision detection was working very fine and, there was no problem in the generation of coins and runes. The snake's movement algorithm previously discussed worked fine and the termination as well as restart of the game was perfect. The game over screen is shown below and we can also see the frame rate of the screen in the console window.



5.2. PONG! :

Pong! is a table-tennis themed arcade 2-D game made by Atari, Inc. during the 70's. This game has its own legacy in computer graphics and gaming. This game consists of two paddles: one controlled by the player and another by the CPU. The objective is to volley the ball continuously from either side. If paddle fails to be in contact with the ball, the player loses a point. The game starts with three points for each of the side where, the side which has nil points would lose the game.

5.2.1. Making 'Paddle' Class:

This game would definitely require a paddle by which we would volley the ball. We created a class that creates rectangle at the center of left side of the screen. That rectangle can be moved through keyboard control since, it was assigned with keys. This class would create a HUD board that would show the points remaining for the paddle which gets updated when the paddle fails to connect to the ball. Its purpose is just to create a movable paddle rendered on the screen which is kept within the bounds of the screen.

Paddle.hpp

```
#define PAD_W 7
#define PAD_H SCR_H/4
#define SCORE_W 25
#define SCORE_H 20
#define MOVE 15
#define BORDER 10

class Paddle
{
    public :
        Paddle() ;
        ~Paddle() ;
        void createPaddle(int x, int y, int w, int h) ;
        void createScoreBoard(int x, int y, int w, int h) ;
        void displayScoreBoard(Uint8 r, Uint8 g, Uint8 b) ;
        void displayPaddle(Uint8 r, Uint8 g, Uint8 b) ;
        void controlPaddle() ;
        void updatePaddle() ;
        int getScore() ;
        void deductScore() ;
        void updatePosition(int x, int y) ;
        void stayWithInTheScreen() ;
        void resetScore() ;
        SDL_Rect getPaddle() ;
        void incrementScore() ;
        double getHitAngle(SDL_Rect a) ;
        void launchAI(SDL_Rect b) ;
    private :
        std::string scoreText ;
        int scoreCount ;
        SDL_Rect paddleRect, scoreRect ;
        SDL_Texture *scoreBoard ;
        TTF_Font *font = Font::loadFont("rsrc/AGENCYR.ttf", 15) ;
} ;
```

Paddle.cpp

```cpp
#include "Paddle.hpp"
Paddle::Paddle()
{}
Paddle::~Paddle()
{}
SDL_Rect Paddle::getPaddle()
{   return paddleRect ;}

void Paddle::controlPaddle()
{
    if(Launcher::event.type == SDL_KEYDOWN)
    {
        switch(Launcher::event.key.keysym.sym)
        {
            case SDLK_UP :
                paddleRect.y -= MOVE ;
                break ;

            case SDLK_DOWN :
                paddleRect.y += MOVE ;
                break ;

            default:
                break ;
        }
    }
}

void Paddle::createPaddle(int x, int y, int w, int h)
{
    paddleRect = {x,y,w,h} ;
    scoreCount = 3 ;
}
```

```cpp
void Paddle::deductScore()
{
    scoreCount -=1 ;
}

void Paddle::updatePosition(int x, int y)
{
    paddleRect.x = x ;
    paddleRect.y = y ;
}

void Paddle::incrementScore()
{
    scoreCount += 1 ;
}

void Paddle::createScoreBoard(int x, int y, int w, int h)
{
   scoreRect = {x,y,w,h} ;
}

void Paddle::displayScoreBoard(Uint8 r, Uint8 g, Uint8 b)
{
    SDL_Color a = {r,g,b} ;
    scoreText = std::to_string(scoreCount) ;
    scoreBoard = Font::getText(font, scoreText.c_str(), a) ;
    SDL_RenderCopy(Launcher::renderer, scoreBoard, NULL, &scoreRect) ;
}

void Paddle::displayPaddle(Uint8 r, Uint8 g, Uint8 b)
{
    SDL_SetRenderDrawColor(Launcher::renderer, r,g,b,255) ;
    SDL_RenderFillRect(Launcher::renderer, &paddleRect) ;
}
```

```
void Paddle::stayWithInTheScreen()
{
    if(paddleRect.y<0+BORDER)
    {
        paddleRect.y = 0+BORDER ;
    }
    if(paddleRect.y + paddleRect.h + BORDER> SCR_H)
    {
        paddleRect.y = SCR_H-paddleRect.h - BORDER ;
    }
}

double Paddle::getHitAngle(SDL_Rect a)
{
    if(SDL_HasIntersection(&paddleRect, &a))
    {
        double relativeSite = (paddleRect.y+(paddleRect.h/2))-(a.y+(a.h/2)) ;
        double normalizeSite = relativeSite/(paddleRect.h) ;
        double angle = normalizeSite*(3*PI/2) ;
        return angle ;
    }
}

void Paddle::launchAI(SDL_Rect b)
{
    if(b.y > paddleRect.y+(paddleRect.h/2))
    {
        paddleRect.y += MOVE ;
    }
    if(b.y < paddleRect.y+(paddleRect.h/2))
    {
        paddleRect.y -= MOVE ;
    }
}
```

```
int Paddle::getScore()
{   return scoreCount ;}

void Paddle::resetScore()
{   scoreCount = 3 ;}
```

This class is also responsible to detect at which angle the ball is incident on the paddle so that, it can be deflected to the opposite direction with similar angle. It also keeps account of the score and resets its score for every new game to be played. A simple AI mechanism was implemented where the CPU paddle will just track the ball's movement and go towards it to keep the play continuing from the computer's side.

5.2.2. Making 'Ball' Class:

This class was just created to make a rectangle that could be volleyed by the player as the ball. For the ball, we implemented the collision detection with the bounds of screen where if it collides, it bounces back with a little more velocity than before. The class also had functions to check hit angle of the ball with the paddle so that, the ball would follows the same angle while, bouncing off the paddle. This class also works for checking the status of collision detection of the ball, paddle or the wall as per which, the score will be affected.

Ball.hpp

```
#include "Launcher.hpp"
#define BALL_W 10
#define BALL_H 10
#define BALL_S 9
class Ball
{
    public :
            Ball() ;
            ~Ball() ;

            void createBall(int x, int y, int w, int h) ;
            void drawBall(Uint8 r, Uint8 g, Uint8 b) ;
            void updateBall() ;
            void updateVelocity(int vx, int vy) ;
            SDL_Rect getBall() ;
            void updateBallPosition(int x, int y) ;
            bool shouldDeductScore() ;
            void avoidPlay() ;
            void avoidCOM() ;
            bool AIfail() ;


    private :

        SDL_Rect ballRect ;
        float velX, velY ;
        bool deduct, deductCOM ;
} ;
```

(Here, *velX* and *velY* are the two-dimensional velocities of the ball which can be updated through class.)

Ball.cpp

```cpp
#include "Ball.hpp"

Ball::Ball()
{}
Ball::~Ball()
{}

void Ball::createBall(int x, int y, int w, int h)
{
    ballRect = {x,y,w,h} ;
    velX = BALL_S/2 ;
    velY = 0 ;
}

void Ball::drawBall(Uint8 r, Uint8 g, Uint8 b)
{
    SDL_SetRenderDrawColor(Launcher::renderer, r, g, b, 255) ;
    SDL_RenderFillRect(Launcher::renderer, &ballRect) ;
}

void Ball::updateVelocity(int vx, int vy)
{
    velX = vx ;
    velY = vy ;
}

void Ball::updateBallPosition(int x, int y)
{
    ballRect.x = x ;
    ballRect.y = y ;
    velX = BALL_S/2 ;
    velY = 0 ;
}
```

```cpp
void Ball::updateBall()
{
    ballRect.x += velX ;
    ballRect.y += velY ;
    if(ballRect.x+BALL_W>=SCR_W)
    {
        velX =- velX ;
        deductCOM = true ;
    }
    if(ballRect.y+BALL_H>=SCR_H)  velY =- velY ;
    if(ballRect.x <= 0)
    {
        velX =- velX ;
        deduct = true ;
    }
    if(ballRect.y <= 0) velY =- velY ;
}

bool Ball::shouldDeductScore()
{   return deduct ; }

bool Ball::AIfail()
{   return deductCOM ;  }

void Ball::avoidCOM()
{   deductCOM = false ; }

void Ball::avoidPlay()
{   deduct = false ;    }

SDL_Rect Ball::getBall()
{   return ballRect ;   }
```

5.2.3. Making 'Pong' Class:

This class is the main class for the game whose object produces the game content for the *Launcher* and holds together the other two classes. In this class, we simply made simple functions that got assembled by the features of *Paddle* which creates paddles for both CPU and the player which can be controlled through keyboard and *Ball* which generates ball for the paddle to volley and defeat the CPU. All the HUDs and textures are implemented through the renderer after being passed to the *Launcher* class as, it is the class which created the window and the renderer. This class also needs to be included within the *Launcher* class so that, this class would be an object that creates the instance of the game we require to run.

Pong.hpp

```
class Pong
{
    public :

        Pong() ;
        ~Pong() ;

        void createPong() ;
        void displayPong() ;
        void updatePong() ;
        bool isGameOver() ;
        bool didPlayerWin() ;
        void restart() ;

    private :

        bool gameOver, win ;
        SDL_Surface *surface ;
        SDL_Rect rect ;
        std::string text ;
        SDL_Texture *texture ;
        SDL_Color red = {255,0,0,255} ;
        SDL_Color green = {0,255,0,255} ;
        SDL_Texture *bg = Texture::getImage("rsrc/bg.png") ;
        TTF_Font *font = Font::loadFont("rsrc/AGENCYR.ttf", 15) ;
        Mix_Chunk *bounce = Music::loadMusic("rsrc/bounce.wav") ;

} ;
```

Pong.cpp

```cpp
#include "Pong.hpp"

Paddle *leftPaddle = nullptr ;
Paddle *rightPaddle = nullptr ;
Ball *ball = nullptr ;
Pong::Pong()
{
    leftPaddle = new Paddle() ;
    rightPaddle = new Paddle() ;
    ball = new Ball() ;
    gameOver = false ;
    win = false ;
}
Pong::~Pong()
{
    SDL_DestroyTexture(texture) ;
}
void Pong::restart()
{
    gameOver = false ;
    win = false ;
    rightPaddle->resetScore() ;
    leftPaddle->resetScore() ;
}
void Pong::createPong()
{
    //32 is just space between paddle and left side of screen
    leftPaddle->createPaddle(32 , SCR_H/2-PAD_H/2, PAD_W, PAD_H ) ;
    //100 is just space between paddle and left side of screen
    leftPaddle->createScoreBoard(200, 50, SCORE_W , SCORE_H) ;
    rightPaddle->createPaddle(SCR_W-32-PAD_W/2 , SCR_H/2-PAD_H/2  , PAD_W , PAD_H ) ;
    rightPaddle->createScoreBoard(SCR_W-200-SCORE_W/2, 50, SCORE_W , SCORE_H) ;
    ball->createBall(SCR_W/2-BALL_W/2, SCR_H/2-BALL_H/2, BALL_W, BALL_H) ;
    leftPaddle->activateTurn() ;
}
```

```cpp
void Pong::displayPong()
{
    SDL_RenderCopy(Launcher::renderer,bg,NULL,NULL) ;

    if(gameOver)
    {
        text = "GAME OVER" ;
        rect = {SCR_W/2-400/2,SCR_H/2-50,400,50} ;
        surface = Font::getSurface(font, text.c_str(), red) ;
        texture = Texture::createTextureFromSurface(surface) ;
        SDL_RenderCopy(Launcher::renderer, texture, NULL, &rect) ;
        Texture::free_surface(surface) ;
        std::cout << "PRESS 'SPACE' TO RESTART !" << std::endl ;
    }
    if(win)
    {
        text = "YOU WON !" ;
        rect = {SCR_W/2-400/2,SCR_H/2-50,400,50} ;
        surface = Font::getSurface(font, text.c_str(), green) ;
        texture = Texture::createTextureFromSurface(surface) ;
        SDL_RenderCopy(Launcher::renderer, texture, NULL, &rect) ;
        Texture::free_surface(surface) ;
        std::cout << "PRESS 'SPACE' TO RESTART !" << std::endl ;
    }

    leftPaddle->displayPaddle(255,0,0) ;
    leftPaddle->displayScoreBoard(255,255,255) ;
    rightPaddle->displayPaddle(0,0,255) ;
    rightPaddle->displayScoreBoard(255,255,255) ;
    ball->drawBall(255,255,255) ;
}
```

```cpp
bool Pong::isGameOver()
{
    return gameOver ;
}

bool Pong::didPlayerWin()
{
    return win ;
}

void Pong::updatePong()
{
    if(ball->shouldDeductScore())
    {
        leftPaddle->deductScore() ;
        leftPaddle->updatePosition(32 , SCR_H/2-PAD_H/2) ;
        rightPaddle->updatePosition(SCR_W-32-PAD_W/2 , SCR_H/2-PAD_H/2 ) ;
        ball->updateBallPosition(SCR_W/2-BALL_W/2, SCR_H/2-BALL_H/2) ;
        ball->updateVelocity(BALL_S/2, 0) ;
        ball->avoidPlay() ;
    }
    if(ball->AIfail())
    {
        rightPaddle->deductScore() ;
        leftPaddle->updatePosition(32 , SCR_H/2-PAD_H/2) ;
        rightPaddle->updatePosition(SCR_W-32-PAD_W/2 , SCR_H/2-PAD_H/2 ) ;
        ball->updateBallPosition(SCR_W/2-BALL_W/2, SCR_H/2-BALL_H/2) ;
        ball->updateVelocity(BALL_S/2, 0) ;
        ball->avoidCOM() ;
    }

    if(leftPaddle->getScore() == 0) gameOver = true ;
    if(rightPaddle->getScore() == 0) win = true ;

    leftPaddle->stayWithInTheScreen() ;
    rightPaddle->stayWithInTheScreen() ;
    leftPaddle->controlPaddle() ;
    ball->updateBall() ;

    SDL_Rect l = leftPaddle->getPaddle() ;
    SDL_Rect r = rightPaddle->getPaddle() ;
    SDL_Rect b = ball->getBall() ;

    rightPaddle->launchAI(b) ;

    if(checkCollision(l, b))
    {
        double angleLeft = leftPaddle->getHitAngle(b) ;
        ball->updateVelocity(BALL_S*cos(angleLeft), BALL_S*sin(angleLeft)) ;
        leftPaddle->controlPaddle() ;
        Music::playMusic(bounce,20) ;
    }

    if(checkCollision(r, b))
    {
        double angleRight = rightPaddle->getHitAngle(b) ;
        ball->updateVelocity(-BALL_S*cos(angleRight), -BALL_S*sin(angleRight)) ;
        leftPaddle->controlPaddle() ;
        Music::playMusic(bounce,20) ;
    }

}
```

5.2.4. Modification for '*Launcher*' Class:

In *Launcher* class, we need to start the instance of the game by defining an object of *Pong* class to link it with the renderer and the window. We just introduced all the functionalities of the game to the *Launcher* class so that, the game would be fetched by the main function of the project file.

```
Pong *pong = nullptr ;
pong = new  Pong() ;
pong->createPong() ;
while(launcher->isRunning)
{
        pong->displayPong() ;
        pong->updatePong() ;

        if (pong->isGameOver() || pong->didPlayerWin() )
           pong->restart() ;
}
launcher->quit() ;
```

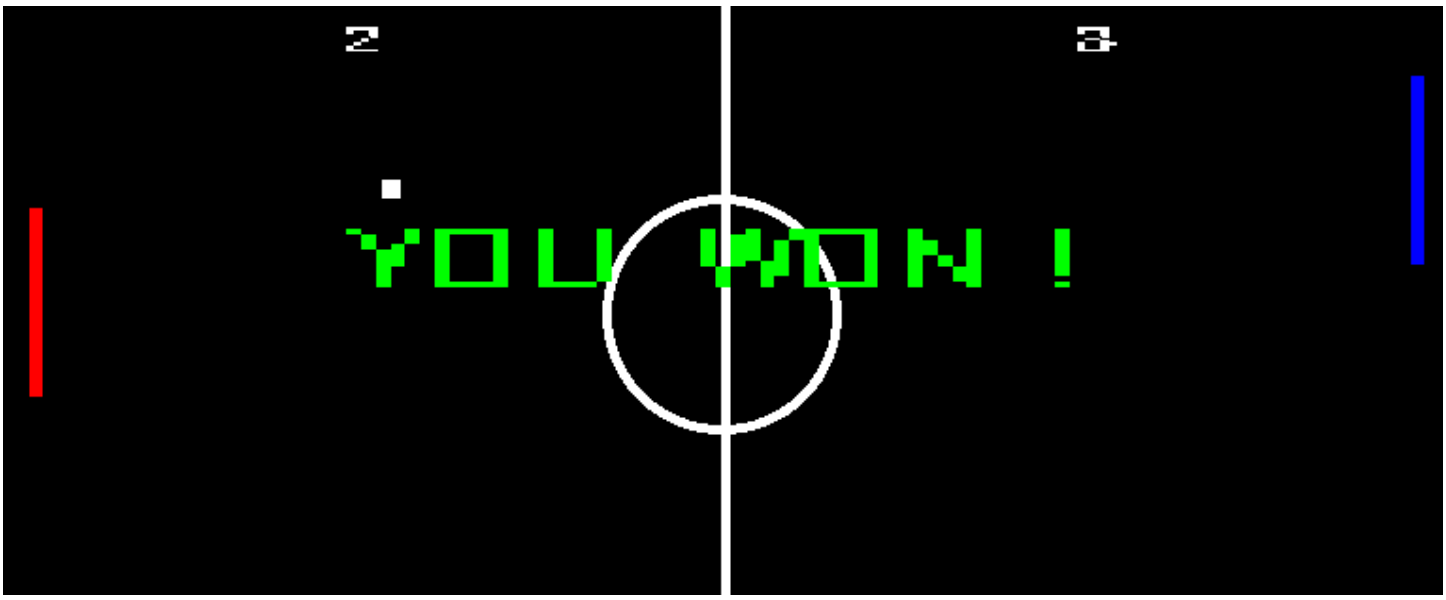


Fig: Instance of a PONG! Game

5.2.5. UML Diagram (PONG!):

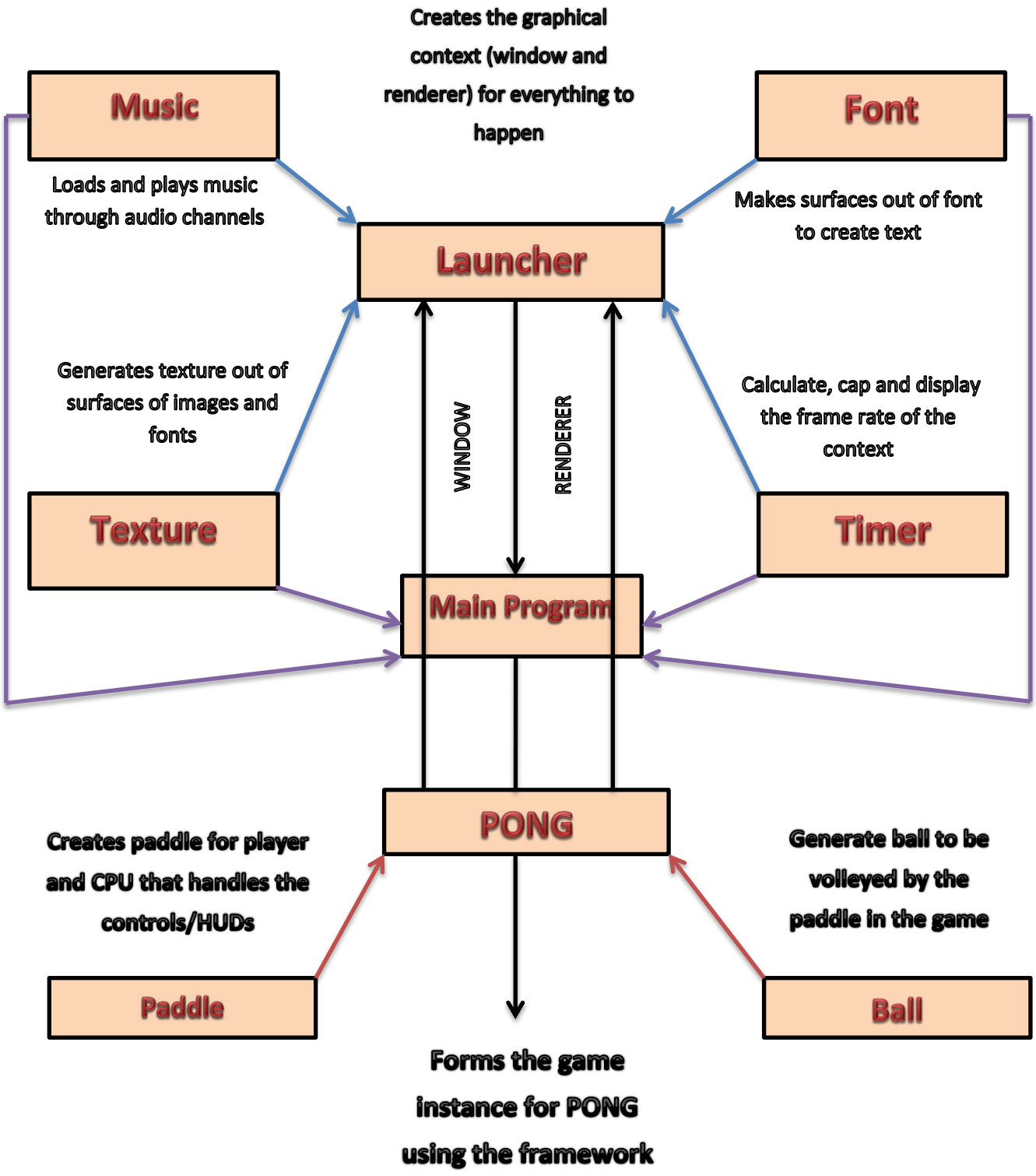
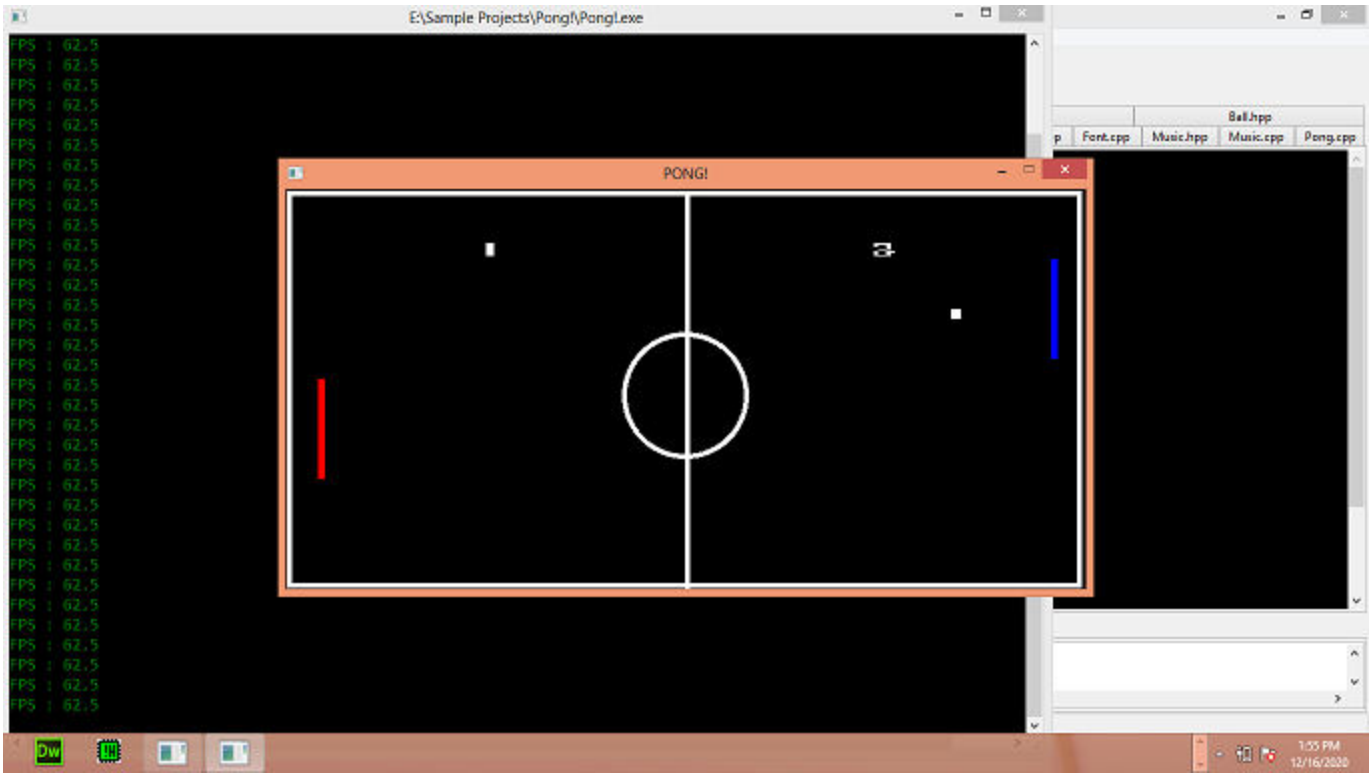


Fig: UML Diagram of the framework along with the game

5.2.6. Testing the game:

This game was capped to 60 FPS throughout the game. The score board for the points count in the window was updating during failure of the paddle to make contact with the ball. The audio channel and the font renderer worked perfectly fine. The movements through the keyboard keys were visible and excellently working. The game seemed very fluid and no major glitches were found at all.



Also, the game over screen also worked fine if player has got no points to spare.

CHAPTER 6:

CAPABILITIES AND LIMATIONS OF FRAMEWORK

6.1. Capabilities of the wrapper framework:

We have tried and tested our framework already to make two 2-D retro games which we thoroughly discussed in this documentation. But, just for the sake of testing this wrapper network, we linked it to other graphical presentations.

6.1.1. Conway's Game of Life:

Inspired by the algorithm of late John H. Conway, we made the popular cellular automata from this framework. It is just a cellular simulation where a void grid exists which can be clicked to turn a particular cell alive. Thus, this game is just the iterations of cells trying to replicate and survive. There was mouse drag and click controls for it and iterations can be altered using keys. Here is an instance of the game made out of this framework:

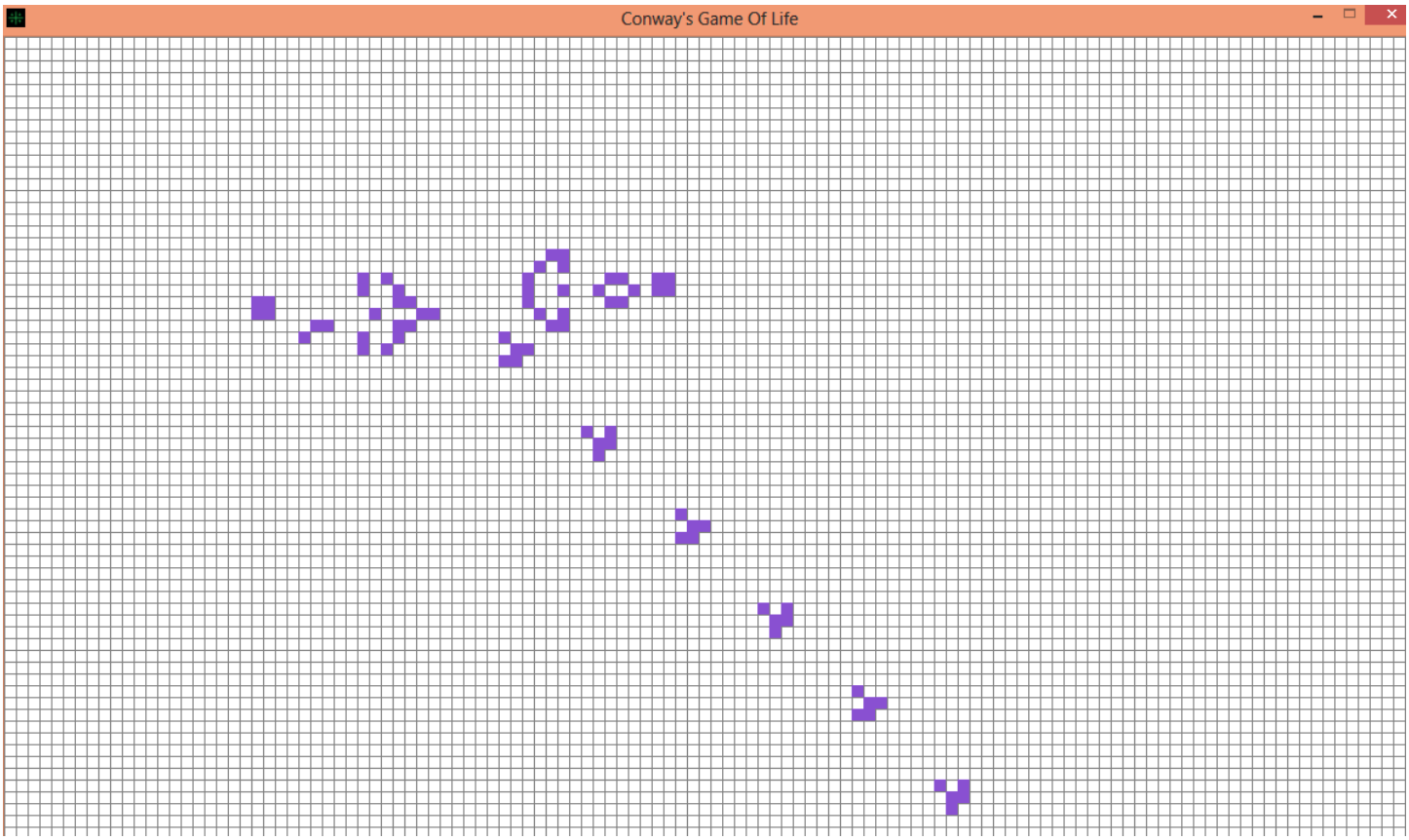


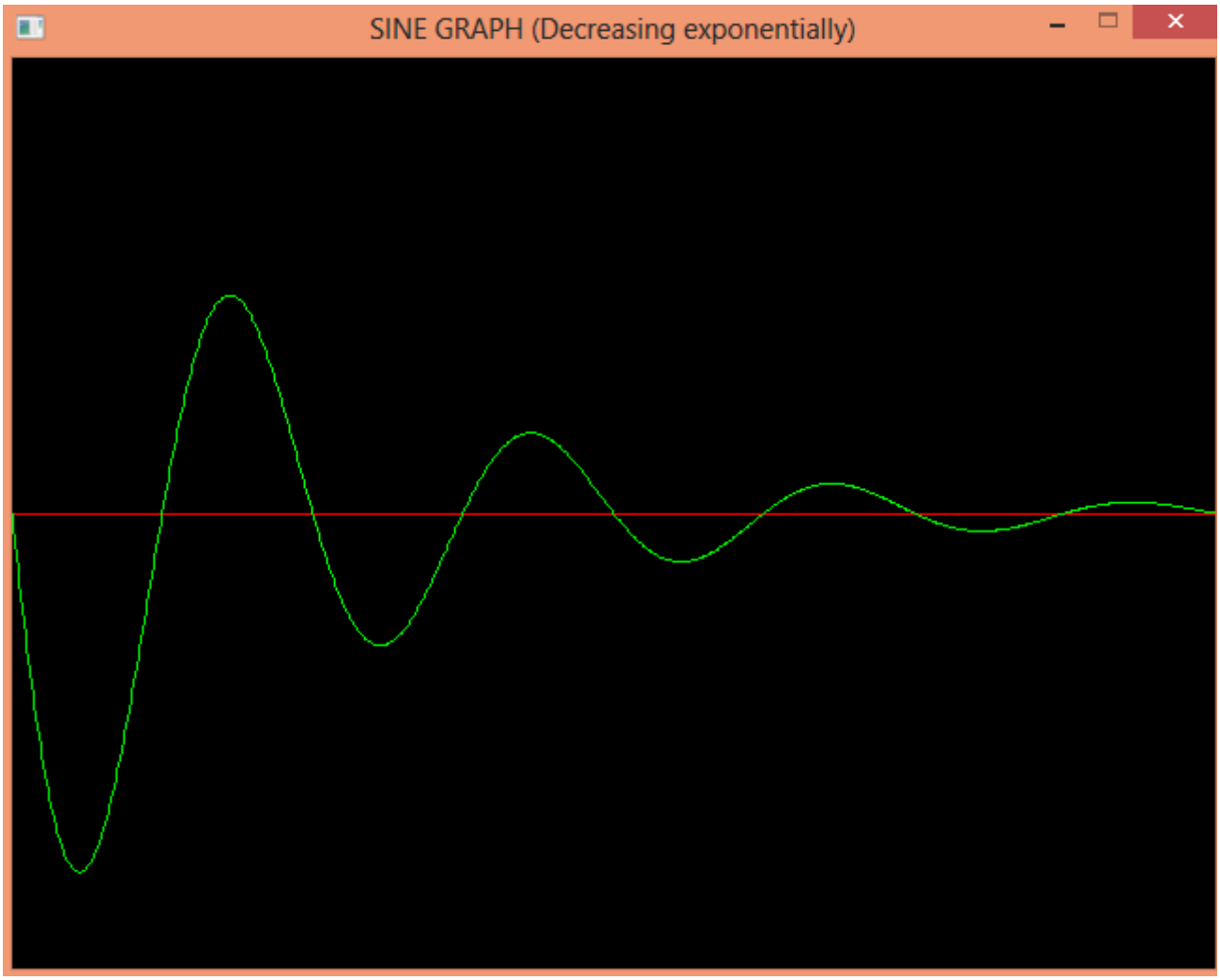
Fig: Glider Gun in Game of Life

6.1.2. Sine Graph Generator:

We also applied this framework just to create a graph generator that produces sine waves that are normal, exponentially increasing or, exponentially decreasing with the input of amplitude and wave cycles.

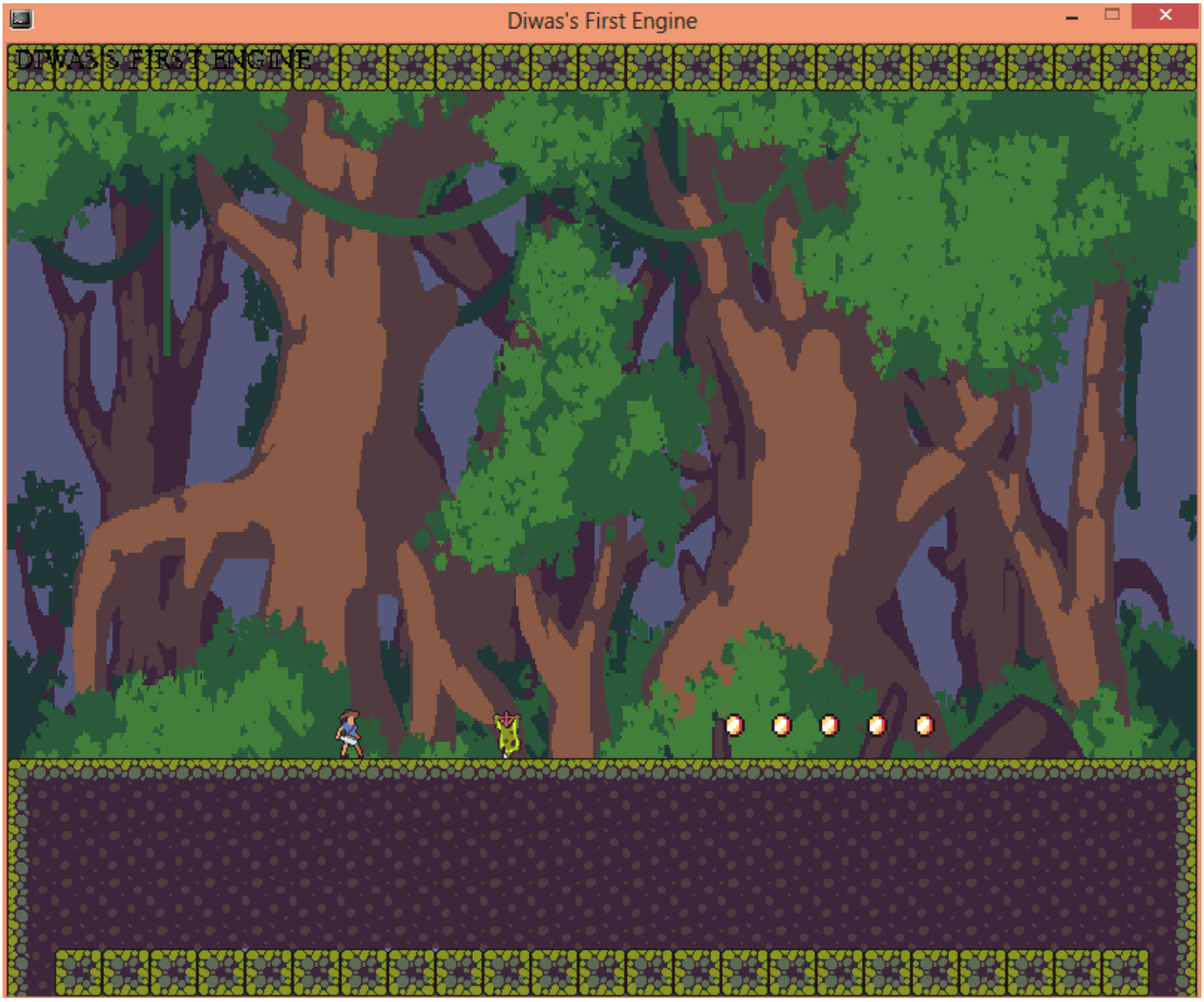
```
                                SINE GRAPH CALCULATOR                                [1 unit Amplitude = 480]
                        *Keep the amplitude strictly less than or, equal to 1 units !*
                        *[CAUTION : Keeping amplitude (>1) gets graph larger than display]*
Select the type of SINE GRAPH :
1.Normal
2.Exponentially Increasing
3.Exponentially Decreasing

3
Give the wave number (i.e. number of wavelengths or, say ~ number of cycles) :
-> 4
Give the value of amplitude(A) :
-> 1
```

It just shows sine graphs but, it helped us to test the rendering pixel quality of SDL2.0 which of course, is the engine of our framework.



6.1.3. Side Scroller Game:

We also tested our framework if, it is good enough for 2-D side scroller games which were most popular during the 90's. With some modifications on the classes, we made simple side scroller on the framework.



Our class was very capable enough to build these types of 2-D graphics. We would tell that SDL2.0 is easily capable of showcasing these texture graphics but, we eventually tested it with limited and masked functions of SDL2.0 inside classes of our framework.

6.1.4. Tetris:

Tetris is simply a block builder game in which a vault exists which getting filled by shapes that are made out of four squares i.e. tetrominos. We also tested this framework to test the game that is the most popular arcade game which still has world tournaments organized for it. If the vault is completely filled, the game is over but, we can defy that by constantly pushing blocks to make lines which reduce the level of the vault.

6.2. Limitations of the framework:

Before even starting the journey of this framework, we knew that it was not capable to do a lot of things. SDL2.0 has variety of functions and properties but, our framework is limited to few functions so that, to test the capabilities, we used functions out of scope of framework in external classes. Some few limitations are enlisted here.

- This framework is working under SDL2.0 but, only few functions are libraries of SDL2.0 are used here.
- This is simply basic so that, to create something extra-ordinary, you need to modify it or, it is not capable enough for it.
- This is not a game engine. Only programmers can get the gist of this wrapper framework as it eases the process of building programs since, we don't have to build screen and rendering context every time. The users will not get any benefit from it because they only need to scare about the program made out of it.
- This cannot handle 3D-graphics. We can make a similar framework for 3-D graphics using Open-GL later but, this (*even SDL2.0 alone*) simply cannot do 3-D graphics at all.
- This does not support any primitive rather than rectangle and has no access to alpha-blending, roto-zooming and image filtering which SDL2.0 is capable of.

DISCUSSION

The coding of the snake was not that too difficult but, we had some glitches due to poor memory management we easily resolved those errors by continuously clearing the memories of data structures so that it will not continuously load data on memory and take over a lot of system memory. In this way both the games ran very smoothly on different platforms since we used our timer class of the frame work to cap the frame rate. Multiple file inclusions created a lots of problems but we were able to debug the undefined references caused due to multiple classes using same types of header files. It was avoided using pragma directives and include header guard.

Since, we used an external library for our frame work we had some runtime errors and segmentation faults due to errors in linking external libraries to the compiler. Later we were able to solve this problem taking the reference through the internet. But, at the end of the day, we had a lot of fun doing this interactive project.

CONCLUSION

In this way, we successfully designed our 2D graphics framework. All the functions in this framework were working completely fine and we were able to develop applications using this framework. All the applications worked perfectly as expected. We learned how to manipulate two-dimensional graphics and render it on a normal window context.

We kept graphical content hand in hand with object oriented programming paradigm so that, we can use multiple classes together to form objects that can solve real world problems. Hence, we accomplished all the objectives set in the beginning of this project.

REFERENCES

Wikipedia: https://en.wikipedia.org

SDL2.0 Official Website: https://www.libsdl.org

Writing Games in SDL by Thomas Lively : https://www.youtube.com/watch?v=yFLa3ln16w0&t=1128s