****

**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**

**PULCHOWK CAMPUS**

**The Other Side**

**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:**

**Ankit Shahi (PUL076BEI006)**

**Ashmit Rajaure (PUL076BEI008)**

**Bibek Pariyar (PUL076BEI012)**

**Paarit Pokharel (PUL076BEI021)**

**Submitted to:**

**Department of Electronics and Computer Engineering, Pulchowk Campus**

**Institute of Engineering, Tribhuvan University**

**Lalitpur, Nepal**

**December, 2020**

# ABSTRACT

The main objective of this course project is to develop a 2-D game titled “**The Other Side**” in C++. Simple and Fast Multimedia Library (SFML) has been used in this project. In this game, the player has to help kangaroos cross the road and river to reach the other side, tackling various obstacles (i.e. car and river) on the way. The player gets 25 points for each kangaroo s/he helps and loses a live on colliding with car or drowning in river. The player will be given 5 lives and the more kangaroos s/he helps, the more s/he will score. This game has been developed using the concept of game engine. We have divided our game into four parts, namely Splash state, Main Menu state, Game state and Game over state. The flow of our program starts from Splash state and goes through Main Menu state, Game state and Game Over State respectively.

# ACKNOWLEGMENTS

First and foremost, we would like to thank our subject teacher **Mr. Bikal Adhikari** for making us familiar with the concepts of C++ programming language and OOP paradigm even in such difficult circumstances. Secondly, we would like to express our gratitude towards the Department of Electronics and Computer Engineering, Pulchowk Campus for providing us this golden opportunity which enabled us to apply our theoretical knowledge and skills we gained in OOP during lectures and lab classes into practical project. Similarly, we would like thank our family members who have indirectly helped us in completing this project with the love and support. Last but not the least, we would like to thank the content creators and youtubers, whose videos and ideas, have helped us in completing this project.

# Chapter One: Introduction

Our course project titled "The Other Side" is a 2-D game made in C++ using SFML library. Basically, you have to help kangaroos cross the road and river to reach the other side, tackling various obstacles in the way. You get 25 points for each kangaroo you help. You will be given 5 lives and the more kangaroos you help, the more you will score. Various features of OOP like class, objects, inheritance, polymorphism, etc. have been used and incorporated in this project to understand the essence of OOP.

## Background and problem statements

For technical students, the most important factor to measure their ability and skill is their practical performance rather than their theoretical knowledge. Considering this fact, an assignment has been given to the students of Electronics, Communication and Information Engineering to prepare a project using C++ programming language in order to develop their practical ability to develop programs and software using C++ programming language as per requirements. This project is a part of the subject Object Oriented Programming, first year and second part of BECIE course.

C++ has been used extensively for game development worldwide due to the features of OOP it offers and vast amount of libraries like SFML, SDL, etc. Gaming has become a multi-billion dollar industry and definitely it would have been unimaginable without the use of C++. C++ is behind the framework of many popular games and game engine we play in our computers and mobile devices. Likewise, game development requires lot of logic and test the abilities of a student like us. So, we have decided to develop a simple game in C++ using SFML.

## Objectives

* + To make students capable of applying the skills and knowledge gained during theory and lab classes in real life.
  + To develop a 2D game in C++ using SFML.
  + To understand the nitty-gritty of OOP through implementation.
  + To develop problem-solving skills and teamwork spirit in students.
  + To understand game design and game development cycle.
  + To learn to collaborate with teammates using online platform like Github.

## Limitation

* This game is developed for 32-bit system.

# Chapter Two: PROBLEM ANALYSIS

## Understanding the problem

A meeting was held among the group members for the preparation of ‘The other side’ game. The problems that had arisen were:

* How the layout of the game should be?
* How were we going to collaborate in the project?
* How should the score be calculated?
* How should the game run when the player was in the river?
* How should we get the graphics for the game?

## Input Requirements

The inputs required were for the movement of the player and to start/restart the game.

1. The inputs required were for the movement of the player and to start/restart the game.
2. Left arrow: To move the player in the left direction.
3. Right arrow: To move the player in the right direction.
4. Up arrow: To move the player forward.
5. Down arrow: To move the player backward.
6. Space: To restart the game after it is over.
7. Escape: To go back to the main menu after the game is over.

## Output Requirements

According to the input from the player, the player character in the game is updated and shown to the player immediately. Also, the movements of the vehicles in the road and logs in the river are shown to the player so he/she can dodge or jump over. Along with this, the score earned by the player and the lives left are shown in the top section.

## Processing Requirements

In this project, 4 people collaborated together for the completion of the game. The coding is done on 64-bit OS using Visual Studio 2019. Although it was coded in 64-bit, it is built for a 32-bit machine.

## Technical Feasibility

In the context of feasibility, the coding didn’t require much of economic expenditure, including electricity and power consumption and manpower due to simplicity of the game. The game runs in any machine with 32-bit/64-bit windows OS.

# Chapter Three: Review of Related Literatures

This game is based in middle-level language general purpose language i.e. C++ programming. In this project we use important parts of C++ programming which are class, objects, interface, inheritance, pointer, data file, functions, control statement, looping.

**C++ programming language:**

C++ is a programming language developed by Bjarne Stroustrup in 1979 at Bell Labs. C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. It is a superset of C, and that virtually any legal C program is a legal C++ program. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

**Class and Objects:**

C++ is an object-oriented programming language. Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake. Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members". A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

**Creating Class:**

class MyClass {       // The class  
  public:             // Access specifier  
    int myNum;        // Attribute (int variable)  
    string myString;  // Attribute (string variable)  
};

* The class keyword is used to create a class called MyClass.
* The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
* Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.
* At last, end the class definition with a semicolon ;.

**Creating object:**

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects. To create an object of MyClass, specify the class name, followed by the object name. To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

class MyClass {       // The class  
  public:             // Access specifier  
    int myNum;        // Attribute (int variable)  
    string myString;  // Attribute (string variable)  
};  
  
int main() {  
  MyClass **myObj**;  // Create an object of MyClass  
  
  // Access attributes and set values  
  **myObj.**myNum = 15;   
  **myObj.**myString = "Some text";  
  
  // Print attribute values  
  cout << myObj.myNum << "\n";  
  cout << myObj.myString;  
  return 0;  
}

**Interface:**

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows –

class Box {

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

**Inheritance:**

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

**Base and derived class:**

* An existing class that is "parent" of a new class is called a base class. New class that inherits properties of the base class is called a child class or derived class.
* Inheritance is a technique of code reuse. It also provides possibility to extend existing classes by creating derived classes.

#include <iostream>

using namespace std;

class Account {

public:

float salary = 60000;

};

class Programmer: public Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

cout<<"Salary: "<<p1.salary<<endl;

cout<<"Bonus: "<<p1.bonus<<endl;

return 0;

}

**Output:**

Salary: 60000

Bonus: 5000

**Pointer:**

Pointer is a variable in C++ that holds the address of another variable. They have data type just like variables, for example an integer type pointer can hold the address of an integer variable and a character type pointer can hold the address of char variable.

**Syntax:**

data\_type \*pointer\_name;

**Example:**

#include <iostream>

using namespace std;

int main(){

//Pointer declaration

int \*p, var=101;

//Assignment

p = &var;

return 0;

}

**Data File:**

Most programs need to save data to disk files and read it back in. Working with disk files requires another set of classes: ifstream for input, fstream for both input and output, and ofstream for output. Objects of these classes can be associated with disk files, and we can use their member functions to read and write to the files.

**Opening and closing files:**

* A file must be opened before you can read from it or write to it.
* Either ofstream or fstream object may be used to open a file for writing. The ifstream object is used to open a file for reading purpose only.
* Following is the standard prototype for open() function, which is a member of fstream, ifstream, and ofstream objects.

*void open(const char \*filename, ios::openmode mode);*

* Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.
* In order to close an opened file, the corresponding ofstream/ifstream/fstream used to open the file should invoke the close() function.

**Example:**

#include<iostream>

#include<fstream>

#include<cstdlib>

using namespace std;

int main()

{

char ch;

ifstream in;

in.open("student.txt",ios::in);

if(in.fail()) //Error handling while opening a file

{

cout<<"Error opening file,exiting..."<<endl;

exit(1);

}

cout<<"Contents of the file:"<<endl;

while(in.get(ch))

{

cout<<ch;

}

in.close();

return 0;

}

**Functions:**

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

**Syntax:**

void myFunction() {  
  // code to be executed  
}

* myFunction() is the name of the function
* void means that the function does not have a return value.
* inside the function (the body), code that defines what the function should do is added

**Calling a function:**

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called. To call a function, the function's name is written followed by two parentheses () and a semicolon ;.

**Example:**

// Create a function  
void myFunction() {  
  cout << "I just got executed!";  
}  
  
int main() {  
  **myFunction();** // call the function  
  return 0;  
}

**Output:**

I just got executed!

**Control statements:**

A control statement is used in a programming language to control the flow of the program. They are nothing but a keyword or statements that are used in a program to transfer the flow of control to another statement based on the conditions. Based on the given condition, it evaluates the result and executes the corresponding statements. Control statements are the statement that controls the flow of the program in order to execute the piece of the code using various controls statement like if statement, if-else statement, break statement, continue statement, for loop, while loop, do while loop.

**if-else statement:**

if(*condition*) {

*//code if condition true*

} else {

*//code if condition false*

}

**switch statement:**

switch(*expression*){

case *constant1:*

*//code to be executed if expression is equal to the constant1*

break;

case constant2:

*//code to be executed if expression is equal to the constant2*

break;

default;

*//code to be executed if the expression is not equal to any constant*

}

**Looping:**

Looping is used when a certain block of code needs to run multiple times. There are different types of looping. They are for loop, while loop and do..while loop.

**for loop:**

for(*statement1, statement2, statement3*){

//code to run multiple times

}

**while loop:**

while(*condition*){

*//code run till the condition is false*

}

**do..while loop:**

do{

*//code to run*

} while (*condition);*

The code of do..while loop runs at least once.

# Chapter Four: ALGORITHM DEVELOPMENT, FLOWCHART AND UML DIAGRAMS

## Algorithm

## Algorithms for GameState

1. **FUNCTION : void Init()**

STEP 1 : Start

STEP 2 : If error occurs while loading sample sounds to soundBuffers, then display error message, else continue

STEP 3 : Load textures and fonts into respective maps ( \_textures and \_fonts )

STEP 4 : Setup GameState attributes [ lives, gamestate, score]

STEP 5 : Create object pointers of game elements [ vehicle, logs, kangaroo ]

STEP 6 : Set initial gameState as ready

STEP 7 : Stop

1. **FUNCTION : void HandleInput ()**

STEP 1 : Start

STEP 2 : Create an Event type object

STEP 3 : Poll event to the window

STEP 4 : If event is of Close type, close the window, else continue

STEP 5 : Stop

1. **FUNCTION : void Update ( float deltaTime )**

STEP 1 : Start

STEP 2 : Get kangaroo position

STEP 3 : If position of kangaroo crosses top screen, add score and move kangaroo to home position

STEP 4 : Move vehicle objects

STEP 5 : If clock time is greater than car\_spawn\_frequency, then spawn a car to a random lane

STEP 6 : Restart clock

STEP 7 : If clock2 time is greater than log\_spawn\_frequency ( 4.0f ), then spawn a log at random lane

STEP 8 : If clock5 time is in range ( 5, 6), set gameState to playing, now player can control kangaroo

STEP 9 : If gameState == Playing, then update kangaroo and restart clock3

STEP 10 : Pass control to DetectCollision() function

STEP 11 : ElseIf gameState == GameOver, change state to GameOverState

STEP 12 : Stop

1. **FUNCTION : void DetectCollision ( float deltaTime )**

STEP 1 : Start

STEP 2 : Load all vehicle object sprites to a vector VehicleSprites and detect DetectCollision

STEP 3 : Loop i = 0 until i less than total Sptires

STEP 4 : If collision takes, place, then

STEP 5 : Play kill sound

STEP 6 : Decrease a life

STEP 7 : If lives is less than zero, then set gameState to GameOver and send kangaroo to home postion

STEP 8 : Add 1 to i and Next i

STEP 9 : Load all logs object sprites to a vector VehicleSprites and detect DetectCollision

STEP 10 : Loop i = 0 until i less than total Sptires

STEP 11 : If collision takes, place, then

STEP 12 : Play kill sound

STEP 13 : Decrease a life

STEP 14 : If lives is less than zero, then set gameState to GameOver and send kangaroo to home postion

STEP 15 : Add 1 to i and Next i

STEP 16 : Set boolean variable inRiver as false, and triggers true if kangaroo is in the location of river

STEP 17 : Load all log sprites to a vector LogsSprite and detect collision

STEP 18 : Loop i = 0 until i less tha total Log sprites

STEP 19 : If kangaroo collides with logs, then attach kangaroo to log and restart clock4

STEP 20 : If inRiver is false, restart clock4

STEP 21 : Else if inRiver is true, then

STEP 22 : Play Splash sound

STEP 23 : Decrease a life

STEP 24 : If lives is less than zero, then set gameState to GameOver and send kangaroo to home postion

STEP 25 : Add 1 to i and Next i

STEP 26 : Update Remaining lives

STEP 27 : Stop

1. **FUNCTION : void Draw ( )**

STEP 1 : Start

STEP 2 : Clear window

STEP 3 : Draw Background

STEP 4 : Draw logs

STEP 5 : Draw kangaroo

STEP 6 : Draw Vehicles

STEP 7 : Draw Score

STEP 8 : Draw lives

STEP 9 : Display all drawn objects to window

STEP 10 : Stop

1. **FUNCTION : void addScore ( )**

STEP 1 : Start

STEP 2 : add 25 to score if player reaches top

STEP 3 : Set new value of score

STEP 4 : Set position of kangaroo to home, only changing y position

STEP 5 : Stop

## Algorithm for Kangaroo

1. **CONSTRUCTOR Kangaroo**

STEP 1 : Start

STEP 2 : Load jump sound to Buffer

STEP 3 : If file not found, display error

STEP 4 : Else Set jump sound from buffer

STEP 5 : Set Kangaroo Texture

STEP 6 : Set Starting Position for Kangaroo

STEP 7 : Stop

1. **FUNCTION : void getPosition ( )**

STEP 1 : Start

STEP 2 : Return y position of kangaroo sprite

STEP 3 : Stop

1. **FUNCTION : void setPosition ( float yPosition )**

STEP 1 : Start

STEP 2 : Get new Y position of kangaroo

STEP 3 : Get X position from sprite

STEP 4 : Set Kangaroo Sprite Position to new x and y position

STEP 5 : Stop

1. **FUNCTION : void update ( float deltaTime )**

STEP 1 : Start

STEP 2 : If left key pressed and kangaroo's x position is greater than zero, then

Play jump sound

move kangaroo sprite left by KANGAROO SPEED ( 45.0f)

STEP 3 : If right key pressed and kangaroo's x position is greater than zero, then

Play jump sound

move kangaroo sprite right by KANGAROO SPEED ( 45.0f)

STEP 4 : If up key pressed and kangaroo's x position is greater than zero, then

Play jump sound

move kangaroo sprite up by KANGAROO SPEED ( 45.0f)

STEP 5 : If down key pressed and kangaroo's x position is greater than zero, then

Play jump sound

move kangaroo sprite down by KANGAROO SPEED ( 45.0f)

STEP 6 : Stop

1. **FUNCTION : void Attach ( float Xvelocity )**

STEP 1 : Starting

STEP 2 : move kangaroo sprite with ( Xvelocity , 0)

STEP 3 : Stop

1. **FUNCTION : const sf::Sprite &GetSprite () const**

STEP 1 : Start

STEP 2 : Return kangaroo Sprite

STEP 3 : Stop

## Algorithm for Log

1. **FUNCTION : void spawnLogsLeft / Right ( float lane )**

STEP 1 : Start

STEP 2 : Call function LogsLaneSelect with Lane as parameter and save its value to SelectedLane

STEP 3 : Place log on the selected lane

STEP 4 : Push the logsprite to a vector of logsprites

STEP 5 : Stop

1. **FUNCTION : int LogsLaneSelect ( int Lane )**

STEP 1 : Start

STEP 2 : Switch cases for lane

STEP 3 : If Case 0, return 45

STEP 4 : If Case 1, return 90

STEP 5 : If Case 3, return 135

STEP 6 : If Case 4, return 180

STEP 7 : Else default, return 90

STEP 8 : Stop

1. **FUNCTION : void moveLogsLeft / right ( float deltaTime )**

STEP 1 : Start

STEP 2 : For i = 0 until i less than totalLogSprites

STEP 3 : If logsprite X coordinates crosses screen, delete that object

STEP 4 : Else continue moving log and Set XlogVelocity to movement

STEP 5 : Increase i by 1 and STEP 2

STEP 6 : Stop

1. **FUNCTION : int getXvelocity ( )**

STEP 1 : Start

STEP 2 : Return XlogVelocity

STEP 3 : Stop

1. **FUNCTION : void drawLogs ( )**

STEP 1 : Start

STEP 2 : For i = 0 until i less than totalLogSprites

STEP 3 : Draw logsprites

STEP 4 : Stop

1. **FUNCTION : const sf::Sprite &getLogsSprite () const**

STEP 1 : Start

STEP 2 : Return logsprites

STEP 3 : Stop

## Algorithm for Vehicle

1. **FUNCTION : void spawnVehicleLeft / Right ( float lane )**

STEP 1 : Start

STEP 2 : Call function LaneSelect with Lane as parameter and save its value to SelectedLane

STEP 3 : Place log on the selected lane

STEP 4 : Push the Vehicleprite to a vector of Vehiclesprites

STEP 5 : Stop

1. **FUNCTION : int VehicleLaneSelect ( int Lane )**

STEP 1 : Start

STEP 2 : Switch cases for lane

STEP 3 : If Case 0, return 455

STEP 4 : If Case 3, return 410

STEP 5 : If Case 1, return 365

STEP 6 : If Case 4, return 320

STEP 7 : If Case 2, return 275

STEP 8 : Else default, return 455

STEP 9 : Stop

1. **FUNCTION : void moveVehicleLeft / right ( float deltaTime )**

STEP 1 : Start

STEP 2 : For i = 0 until i less than totalVehiclesprites

STEP 3 : If Vehiclesprite X coordinates crosses screen, delete that object

STEP 4 : Increase i by 1 and STEP 2

STEP 5 : Stop

1. **FUNCTION : void drawVehicle ( )**

STEP 1 : Start

STEP 2 : For i = 0 until i less than totalVehiclesprites

STEP 3 : Draw Vehicleprites

STEP 4 : Stop

1. **FUNCTION : const sf::Sprite &getVehicleSprite () const**

STEP 1 : Start

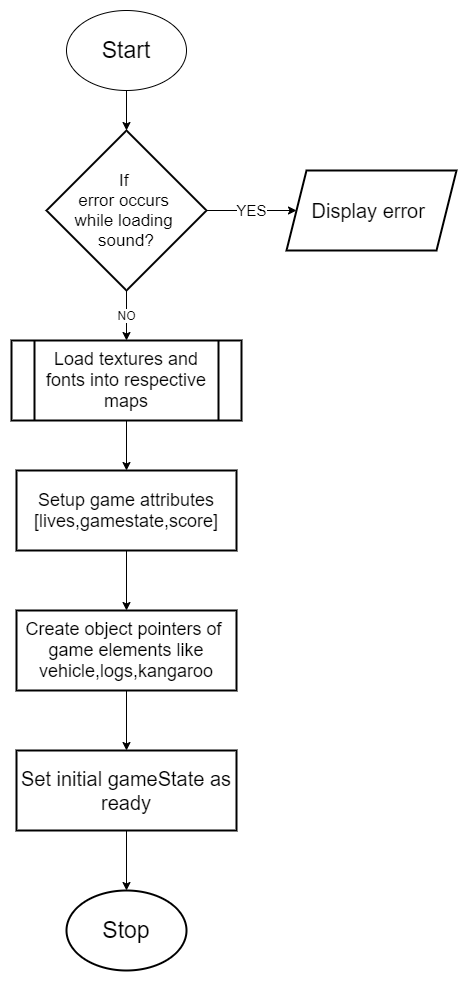
STEP 2 : Return Vehicleprites

STEP 3 : Stop

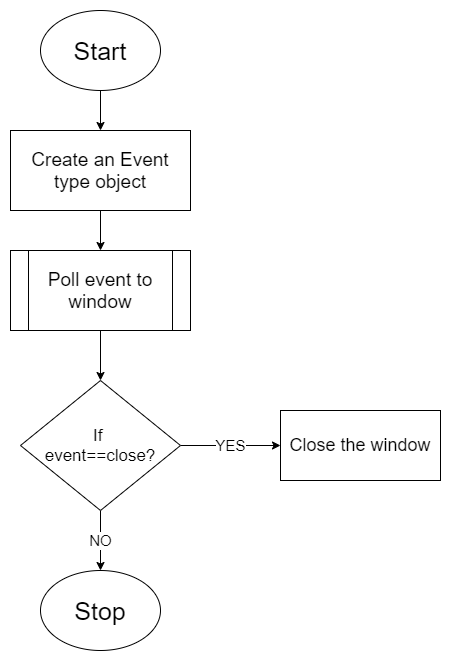
## Flowchart

## Flowchart for GameState

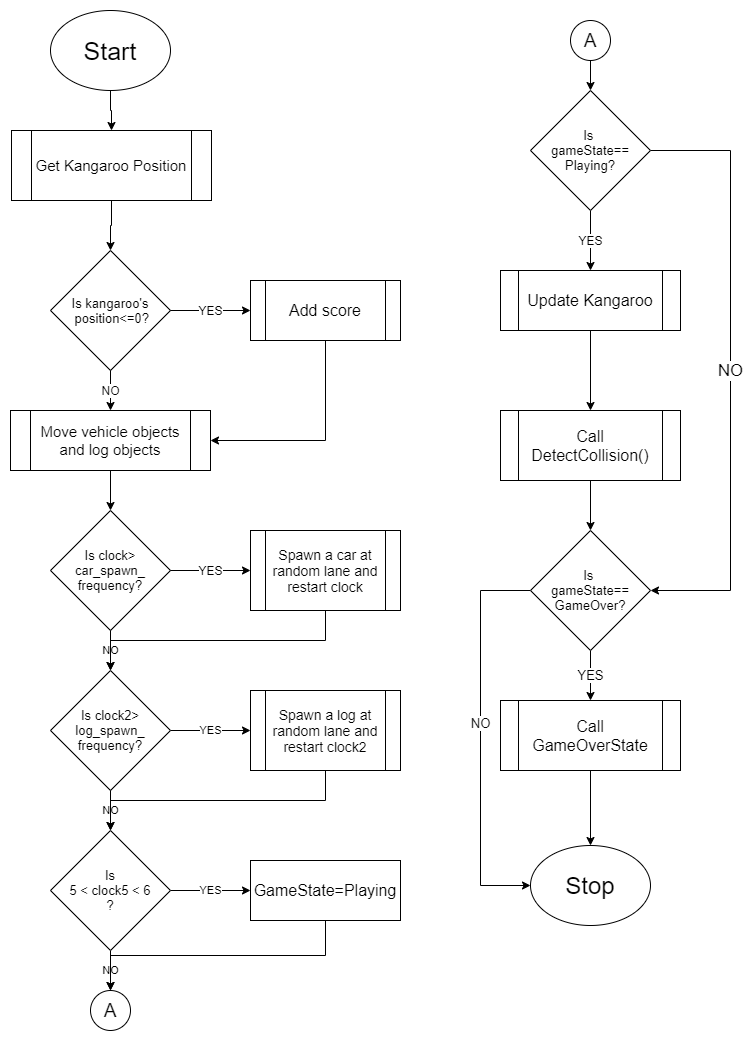
1. **Flowchart for init()**

****

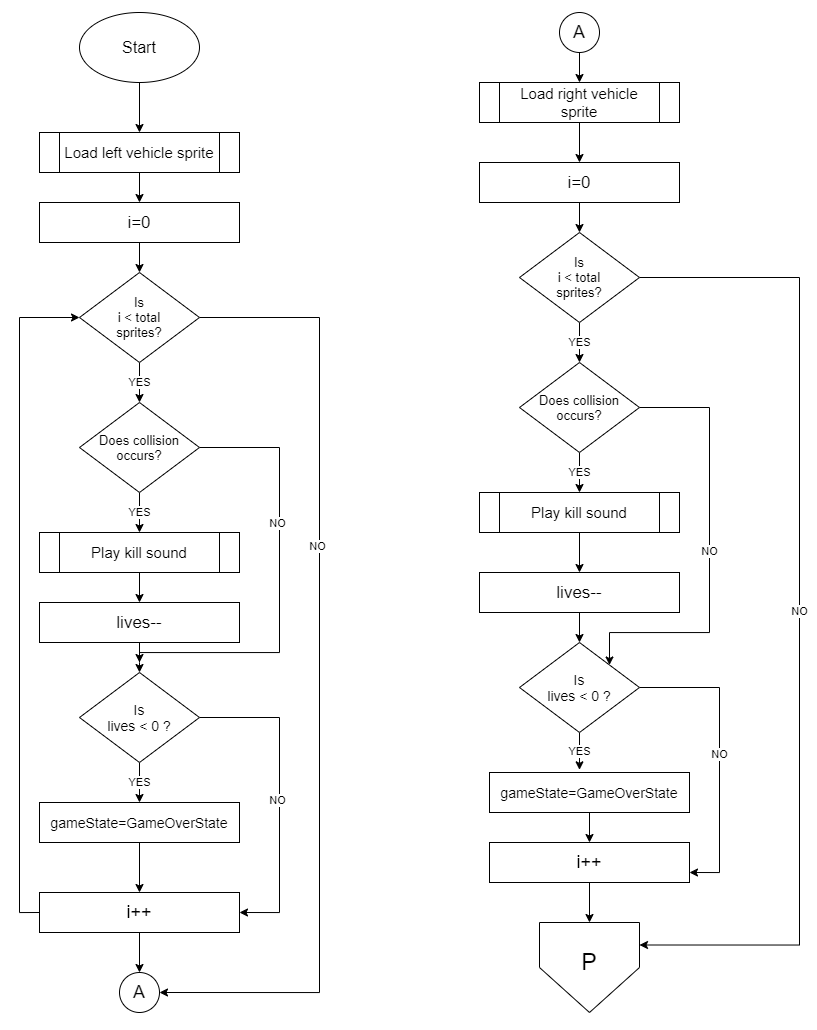
1. **Flowchart for HandleInput()**

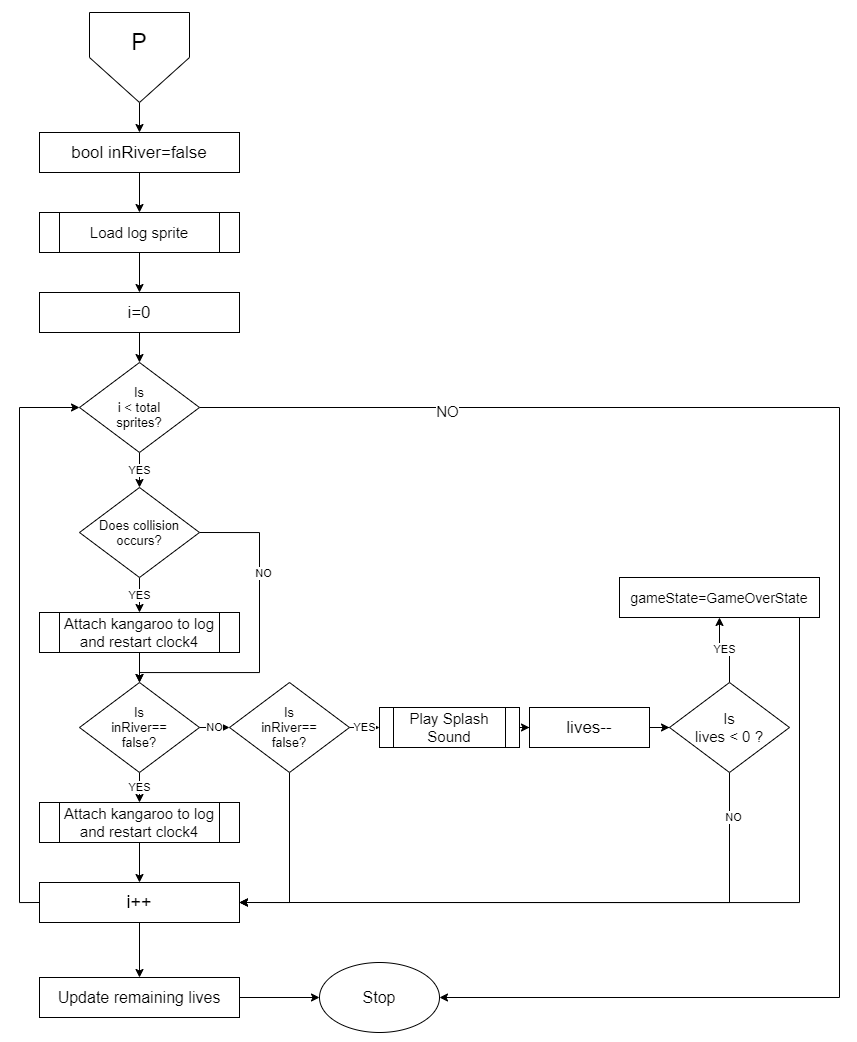
****

1. **Flowchart for Update ( float deltaTime )**

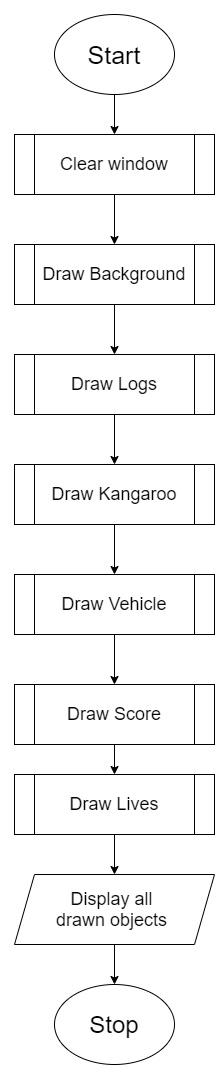
****

1. **Flowchart for DetectCollision ( float deltaTime )**

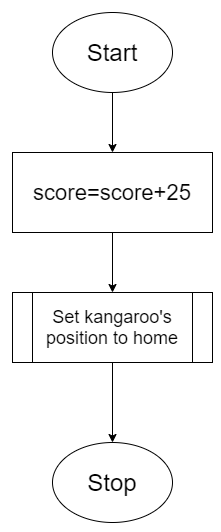
****



1. Flowchart for Draw()



1. Flowchart for addScore()



## UML diagrams

## Inheritance for state

## 

## UML diagram for state classes

## 

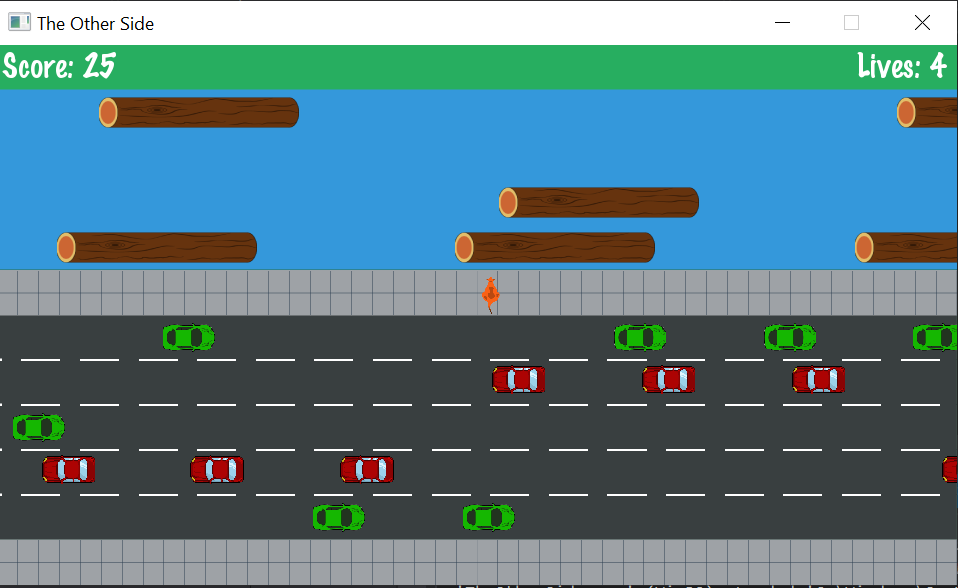
# Chapter Six: RESULTS AND DISCUSSION

## Results

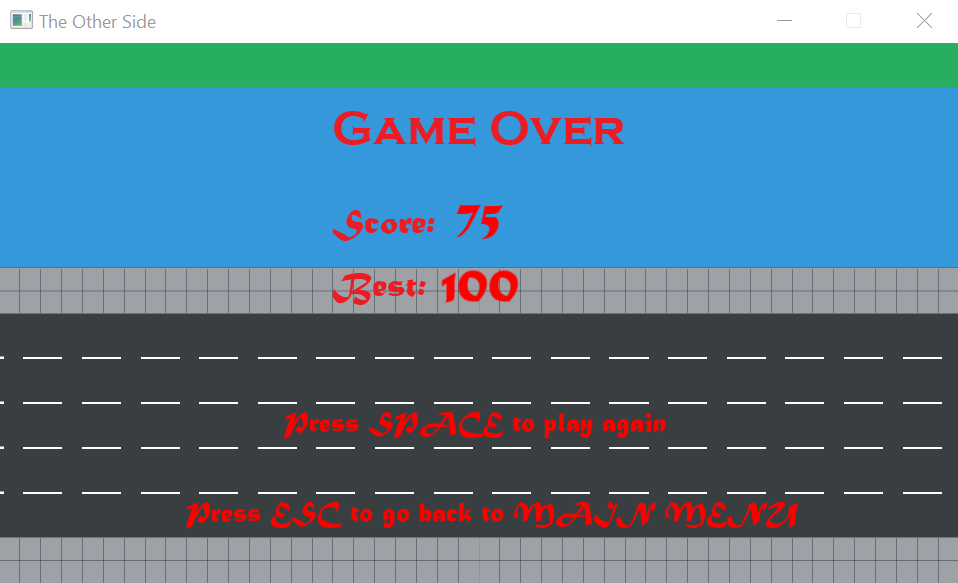
The Other Side is simple, fun and compelling game to play with. As soon as, the game is executed, you get the main menu. In order to play, the Play button must be clicked.



Now you can enjoy the game and get as many kangaroos you can to the other side until you run out of lives.



After you run out of lives, you get the game over message with your score and high score and have two option, either to play again or return back to main menu.



## Discussion

We all have tried our best to develop this game and complete it in given time frame and circumstances. We faced a bit trouble with the drowning logic of kangaroo while crossing river which was ultimately resolved through team efforts and lots of trial and error. But still the game could be made a bit more interesting by increasing the spawning speed of cars with time and varying the length of raft. Similarly, this game has been developed from top-view perspective but it could have been developed in front-view perspective to make the game more realistic. Likewise, we have used only two types of car texture but more variety of cars could have been added to make the game look more interesting.

# Chapter Five: IMPLEMENTATION AND CODING

**5.1 Implementation**

The objectives of building The Other Side has already been discussed and planned as a problem in the above chapters. But the implementation of the plan and the idea of how the final game looks like can be obtained from the below pictures.



Figure 5 a Main menu

5 a shows the main menu of the game where the player is asked to select an option either to play the game or quit.

Let ‘Play’ be selected.

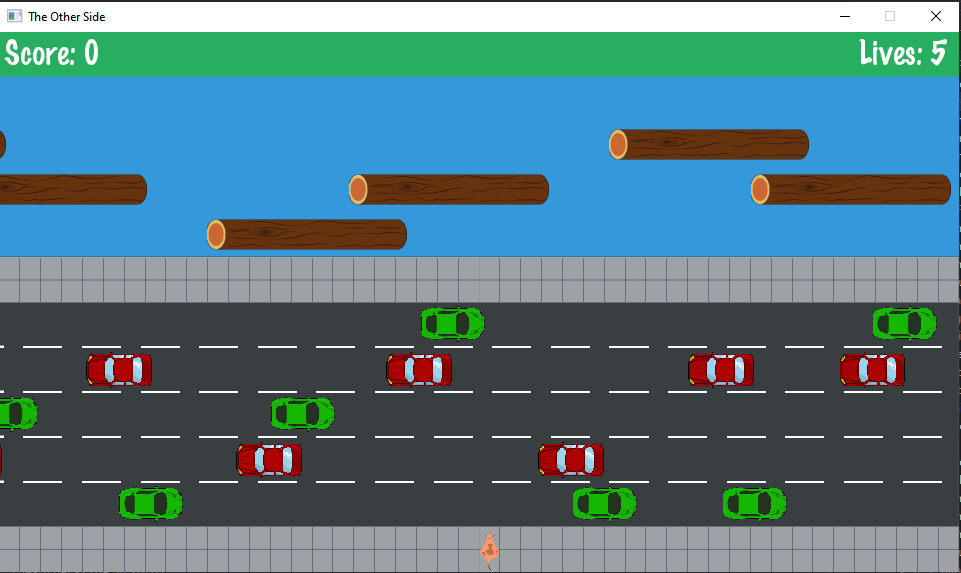


Figure 5 b Game screen

5 b shows the game screen. Here we can see the kangaroo (player) at the bottom of the screen. There are 5 lanes on the road and the cars are moving in the left and right direction alternatively. There is a river which has 4 lanes and logs are floating on it. There is a partition between the road and the river. The player has to reach the top most row to win a score.

We can also see the score and lives on the top corners. As we can see, the player has 5 lives in total. After 5 lives, the game will be over.



Figure 5 c Game over

After the player is hit by a car or drowns in the river 5 times, the game will be over. As we can see in the figure 5 c, the player can press space key to restart the game or esc to go back to the main menu.

**5.2 Coding of the game:**

In this game, C++ programming language is used. Since C++ is an Object Oriented Programming (OOP) language, we met our objective by writing different classes and interface. The classes of the project are as follows:

* AssetManager
* Collision
* Game
* GameOverState
* GameState
* InputManager
* Kangaroo
* Logs
* State
* MainMenuState
* SplashState
* StateMachine
* Vehicle

**AssetManager:**

AssetManager class stores all the textures and fonts in a map. Other classes can get their required texture and font using this class’s member functions.

Functions:

* *void LoadTexture ( std::string name, std::string filename )* : This function loads the texture from file and stores in a map object.
* *sf::Texture &GetTexture ( std::string name ) :* This function returns the texture using the name which is the key in the map.
* *void LoadFont ( std::string name, std::string filename ):* This function loads the font from file and stores in a map object.
* *sf::Font &GetFont ( std::string name ) :* This function returns the font using the name which is the key in the map.

Variables:

* std::map<std::string, sf::Texture> \_textures : This stores the textures in a key-value pair.
* std::map<std::string, sf::Font> \_fonts: This stores the fonts in a key-value pair.

**Collision:**

Collision class is responsible for detecting the collision between the player and vehicle or log.

Functions:

* bool CheckSpriteCollision ( sf::Sprite sprite1, sf::Sprite sprite2) : This function returns true if the two sprites passed to the function are colliding with each other.

**Game:**

Game class creates a window and controls the main loop of the game.

Functions:

* *Game(int width, int height, std::string title)*: This is the constructor of the Game class. It creates a window of the height and width with title passed to it.
* *void Run():* This function contains the main loop of the program. The loop stops only when the window is closed.

Variables:

* *GameDataRef \_data = std::make\_shared<GameData>( ):*  This variable stores the objects of StateMachine, Window, AssetManager, InputManager.
* *sf::Clock \_clock:* This is used to get the elapsed time of the game.
* *const float deltaTime = 1.0f / GAME\_FRAME\_RATE :* It stores the time the game takes to render a frame.

**GameOverState:**

GameOverState class is a derived class of State. This class handles when the player has lost all 5 lives. It receives the best score from the file and compares with the score of the player.

Functions:

* *GameOverState(GameDataRef data, int score) :*  This is the constructor of the class. The score earned by the player is passed to it.
* *void Init():* This function opens the best score file and compares the best score with the current score. If the current score is more than the best, the data in the file is overwritten. Also, the texts on the screen are initialized with fonts, position etc.
* *void HandleInput():* It handles the input in the gameover screen. If space is pressed, the game restarts. If ESC is pressed, the player is taken to the main menu.
* *void Draw(float deltaTime);* This function draws the texts, background onto the screen.

Variables:

* GameDataRef \_data; : This variable stores the objects of StateMachine, Window, AssetManager, InputManager.
* sf::Sprite \_background; : This stores the sprite for the background.
* sf::Sprite \_gameOverTitle; : This stores the sprite for the Game Over title.
* sf::Sprite \_gameOverBody; : This stores the sprite for the Game Over body.
* sf::Text \_playAgain; : This stores the sprite for the play again text.
* sf::Text \_backToMainMenu; : This stores the sprite for the back to main text.
* sf::Text \_scoreText; : This stores the sprite for the score text.
* sf::Text \_bestText; : This stores the sprite for the bests core text.
* int \_score; : This stores the score.
* int \_best; : This stores the best score from the file.

**GameState:**

This is the main class that controls everything in the playable part of the game. This class is also a derived class of the State interface.

Functions:

* *GameState(GameDataRef); :* This is the constructor of the GameState class.
* *void Init(); :* This functions initializes the state. It loads the audio, textures, score text, sets lives of the player and creates vehicles, logs and kangaroo object.
* *void HandleInput();* : This functions closes the window when the cross button is pressed.
* *void Update(float deltaTime); :* This function controls the spawning of the logs and vehicles.
* *void Draw(float deltaTime); :* This functions draws all the items to the screen.
* *void DetectCollision(float deltaTime); :* This functions checks collision between the kangaroo and vehicles/logs.
* *void addScore(); :*  This functions adds the score when the player reaches the end.

**InputManager:**

The InputManager class handles the input from the player. It handles the input using the following functions:

Functions:

* *bool IsTextHovered ( sf::Text object, sf::RenderWindow &window); :* It returns true when a text is hovered by the mouse cursor.
* *bool IsTextClicked(sf::Text object, sf::Mouse::Button button, sf::RenderWindow &window); :* It returns true when a text is clicked.
* *bool IsHovered ( sf::Sprite object, sf::RenderWindow &window); :* It returns true when a sprite is hovered by the mouse cursor.
* *bool IsSpriteClicked(sf::Sprite object, sf::Mouse::Button button, sf::RenderWindow &window); :* It returns true when a sprite is clicked.
* *sf::Vector2i GetMousePosition( sf::RenderWindow &window); :*  It returns the position of mouse cursor in the window.

**Kangaroo:**

Kangaroo class controls the player in the game. This class gets the input from the player and moves the player accordingly. It also manages the audio when the kangaroo moves.

Functions:

* *void update(float deltaTime); :* This function checks the input from the player and moves the kangaroo in the game.
* *float getPosition(); :* This function returns the current position of the kangaroo in the game.
* *void setPosition(float yPosition); :* It sets the position of the kangaroo in the game.
* *void drawKangaroo(); :* It draws the sprite of kangaroo in the screen.
* *void Attach(float Xvelocity); :*  It attaches the kangaroo to the log and moves the kangaroo with it.

**Log:**

Log class controls the logs in the river. This class spawns the logs randomly in the river and moves them in a certain direction according to their lane position.

Functions:

* *void spawnLogsLeft(float Lane); :*  This functions spawns the logs in the left side.
* *void spawnLogsRight(float Lane); :*  This functions spawns the logs in the right side.
* *void moveLogsLeft(float deltaTime); :* This function moves the logs in the left direction.
* *void moveLogsRight(float deltaTime); :* This function moves the logs in the right direction.
* *int getXvelocity(); :*  It returns the velocity of the logs.
* *void drawLogs(); :*  It draws the logs in the screen.

**State:**

State is an abstract class. This is an interface from which other state classes are derived. The virtual functions are as follows:

* virtual void Init( ) = 0;
* virtual void HandleInput( ) = 0;
* virtual void Update( float deltaTime ) = 0;
* virtual void Draw( float deltaTime ) = 0;

**MainMenuState:**

MainMenuState is the state that comes after SplashState in the program. This is a display state where user can choose between two options, play and quit. Play option takes user to the main GameState, whereas the quit button closes the program.

Functions:

* *void Init():* This is to load textures, sound and fonts to set properties
* of attributes of this class.
* *void HandleInput()* : This handles any events occuring on the screen such as minimize, resize and close. In this case, it also handles clicks for the buttons displayed on the main menu. If play is clicked, it creates a new state called GameState.
* *void Update():* This method updates buttons of they are hovered. It also changes the buttons color if mouse is hovered above these buttons.
* *void Draw():* This method draws all the sprites, and buttons on the window.

**SplashState:**

SplashState is a state that inherits from the abstract class State. It only appears on the screen for provived duration. It displays an image for certain time and it ends.

Functions:

* *void Init():* This is to load image for the splashState and set its texture.
* *void HandleInput() :* This handles any events occuring on the screen such as minimize, resize and close.
* *void Update():* This method processes all the changes that occurs on the screen. In splashState, there is no changes so nothing to update here.
* *void Draw ():* This method draws the image to the window.

**StateMachine:**

StateMachine is a class that creates a state of game such as splashState, MainMenuState, GameState and GameOverState. Basically, it is a class that create new pages and only one page can be displayed at a time.

Functions:

* *void AddState ( StateRef newState, bool isReplacing = true):*This method is used to add a new state to the program. It takes name of new state as a parameter, and it replaces the previous state, so isReplacing is set to true. New state is added to top of the stack.
* *void RemoveState ( ) :* This method is used to set isRemoving boolean as true when called.
* *void ProcessStateChanges ():* This method monitors all the activities of states. Actual adding and removing of states takes place here. StateRef GetActiveState(): This method returns the active state i.e. state present on the top of stack to perform operations on it.

**Vehicle:**

Vehicle class is the class that is responsible for all the car sprites drawn onto the screen. There are total 5 lanes on the background image and vehicle spawning is done out of which red car spawns on 3 lanes, namely 1st , 3rd and 4th lane from bottomgreen car spawns on the remaining two. On the basis of direction of spawning of vehicle, there are two object pointers created in the gameState, one to spawn vehicle from left and other from right.

Functions:

* *Vehicle Constructor*: This constructor sets its \_data attribute with the shared data of type GameDataRef
* *void spawnVehicleLeft ( float lane ):* spawnVehicleLeft spawn vehicle starting from the left of the screen . It takes one argument 'lane' which is a random number generated in the gameState and selects lane accordingly.
* *void spawnVehicleRight ( float lane):* spawnVehicleRight spawns vehicle starting from the right side of the screen. It also selects lane based on the random number passed to it from gameState.
* *void moveVehicleLeft ( float deltaTime ):* This method moves vehicle to the left and gives visualization of moving vehicle.
* *void moveVehicleLeft ( float deltaTime ):* It is pretty much the same as moveVehicleLeft. The only difference is it moves vehicle to right direction.
* *const std::vector< sf::Sprite > &getVehicleSprite( ) const:* This method is necessary for collision detection. When this method is called, it returns vehicle sprite which is used to check collision with the player object.

# Chapter Seven: CONCLUSIONS

After the completion of coding of the game, the player can have fun and enjoyable experience. The game is pretty simple and intuitive that the players will not have difficult time understanding how to play the game and how to score in the game. The game also utilizes the data file in the computer system in order to store the best score for future reference. The features of the game are:

* It is light and can run in any windows computer.
* It is pretty intuitive. The players will not have difficult time understanding it.
* It stores the best score and compares with the player’s score to make the game more competitive.

By building this game, we learnt more about the C++ programming language and about the object oriented programming system.