Project report

Group 7

Trimester 1, 2023 - 2024

1 Group member

Student ID	Full Name
22127196	Lê Ngọc Anh Khoa
22127264	Hoàng Tuý Minh
22127415	Nguyễn Đức Tín
22127464	Võ Thịnh Vượng

2 Overview

In this project, we will be recreating the classic game "Crossing road". The objective of the game is to move from one side of the road to the other while dodging moving obstacles. We are using C++ to build the game for Windows, utilizing the functions provided in the Windows.h header to indirectly run OS commands.

3 Project organization

a) **Sprites**
Contains all sprite files to draw.

- *Maps*: all maps displayed in main game.
- *Player*: main character sprite and move animations.
- *Obstacles*: obstacle sprites.
- *Char*: alphabet and numeric sprites
- *FX*: collision effects when player overlap with an dangerous obstacle.
- *UI*: game panels, settings and others effects of the game.

These are text files, following this format:

- The first line contains the *height* and *width* of the texture
- The next *height* lines contain *width* numbers, each representing a color (or lack thereof) for the pixel at that location

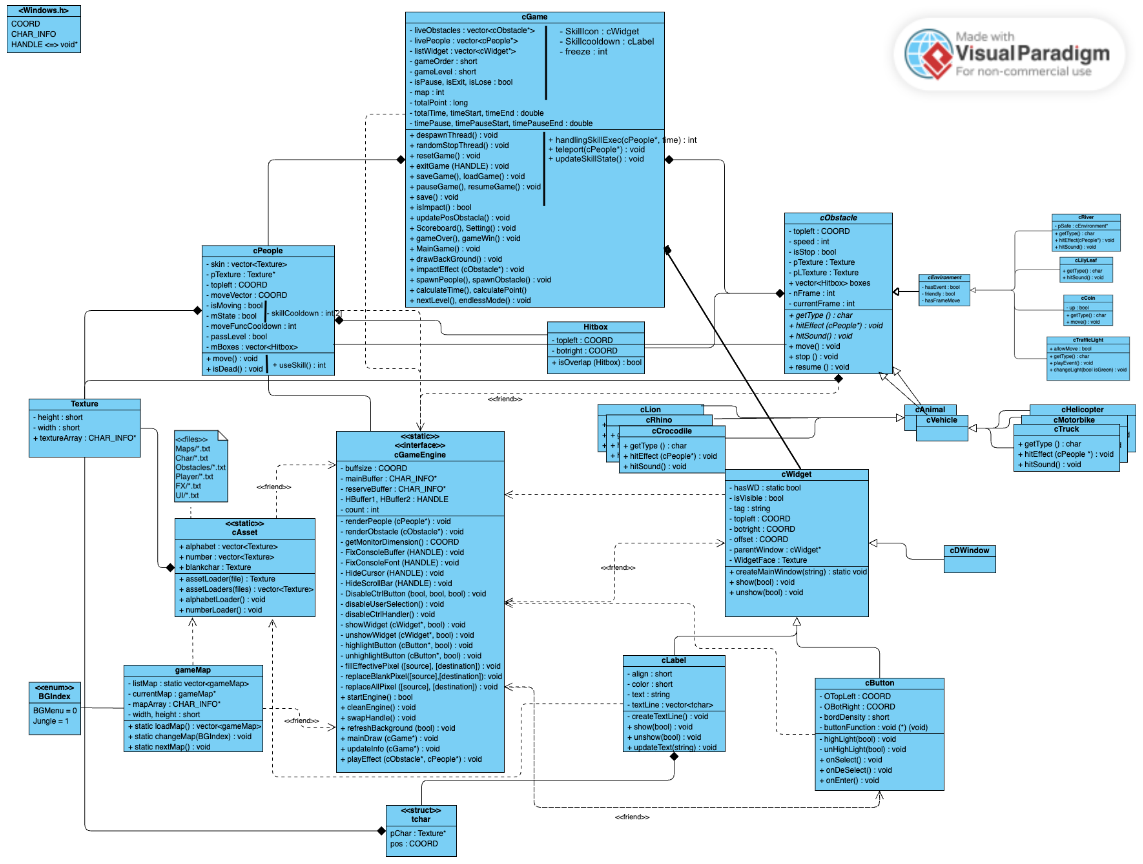
b) **Level**
Contains set up file for every level in game.

c) **Sound**
All music and sound effects.

d) **Save**
Save files for games in progress and leaderboard.
The leaderboard file contains 3 numbers that are the top 3 scores on the machine.
File format for save files:

- The first line contains the time and date the save was created
- The second line contains basic game information in the following order: number of players, level, theme, phase, time elapsed in game, pause time elapsed.
- The third line contains the number of live players *pCount*
- ■ The next *pCount* lines contain the players' information in the following order: X-coordinate, Y-coordinate, the player's score, the number of coins collected.
- The next line contains the number of obstacles on screen *oCount*
- ■ The next *oCount* lines contain obstacle information in the following order: obstacle type, X-coordinate, Y-coordinate, obstacle speed, move direction.
- ■ The remaining lines contain coin information in the following order: object type (1 for coin), X-coordinate, Y-coordinate.

4 Class hierarchy



5 Class description

1. **Hitbox**
   Defines a horizontal rectangle in which 2 objects is considered to be "hitting" each other

   - *topleft*, *botright* : coordinates of the top-left and bottom-right corners of the hitbox, respectively
   - *isOverlap* : checks if 2 hitboxes are overlapping

```
function isOverlap(Hitbox $B$)
    if bottom $A$ is above top $B$ or top $A$ is below bottom $B$ then
        return false
    end if
    if left $A$ is right of right $B$ or right $A$ is left of left $B$ then
        return false
    else
        return true
    end if
end function
```

2. **Texture**
   A class that contains object sprite to be displayed in game.
   Class members:

   - *height*, *width* - short type data to store size of the sprite loaded

- *textureArray* - an array of CHAR_INFO type, $height \times width$ size . Each element of the array indicates the character and color to be printed at that specific location.

3. **gameMap**
   A class that contains information on the background to be displayed in game. Similar to **Texture**, but contains additional static members and methods to determine the current map being used.
   Class members:

   - *height*, *width* : measurements of the map image
   - *mapArray* : CHAR_INFO array with pixel information

   Static class members:

   - *listMap* : a list of **gameMap**, loaded at program start. Is a vector of vectors, each component vector contains background images for a map theme.
   - *currentTheme* : a number indicating the map theme, for example "jungle", "beach", ... Each theme has multiple variations.
   - *currentMap* : pointer to the current **gameMap** being used
   - *currentMapIndex* : a number indicating the current map image variation (called a 'frame') being used for the theme.
   - *numCurrentMapFrame* : the total number of frames available with the current theme
   - *mapLoopCooldown* : timer before the map frame changes

   Static methods:

   - *loadMap* : load a list of map frames from files. See section 3 for file format.
   - *changeMapTheme* : change the current map theme

     ```
     function CHANGEMAPTHEME(theme)
         $currentTheme \leftarrow theme$
         $currentMap \leftarrow$ first element in $listMap[theme]$
         $numCurrentMapFrame \leftarrow$ number of elements in $listMap[theme]$
         $currentMapIndex \leftarrow 0$                ▷ first element
     end function
     ```

   - *nextMapFrame* : cycle through the available map frame for the current map theme

     $$currentMapIndex \leftarrow (currentMapIndex + 1)\%numCurrentMapFrame$$

   - *mapChangeTick* : advance *mapLoopCooldown* timer, then call *nextMapFrame* (and reset timer) if the timer reaches 0

4. **cAsset**
   A pure static class that defines all functions to load sprites from files to objects in the game before starting game.
   Class members:

   - *alphabet* : **Texture**s of letter characters (A - Z)
   - *number* : **Texture**s of number characters (0 - 9)

- *special* : **Texture**s of other special characters
- *blankchar* : **Texture** of the blank (space) character
- $FxFrame$, $flashEffect$: **Texture**s of special effects

Class methods:

- *assetLoader* : loads a **Texture** from a file. See section 3 for file format.

  ```
  function ASSETLOADER($filename$)
      Open file with $filename$
      Create new Texture $loaded$
      Read into $loaded.height$ and $loaded.width$
      $count \leftarrow 0$
      while $count < height \times width$ do
          Read next number $num$
          if $num < 16$ then
              Set $loaded.textureArray_{count}$ to a filled character with color code $num * 16 + num$
          else
              Set $loaded.textureArray_{count}$ to a blank character with color code 0
          end if
      end while
      return $loaded$
  end function
  ```

- *assetLoaders* : load multiple **Texture**s from multiple files at once
- *alphabetLoader* : load the *alphabet* member
- *numberLoader* : load the *number* member
- *specialCharLoader* : load the *special* member
- *alphabetLoader* : load the *alphabet* member
- *settingsLoader* : set the volume of sound and music from the settings file
- *settingsSave* : write down the volume of sound and music to the settings file
- *getChar* : get the **Texture** of a character from loaded static members

5. **Sound**
   A pure static class that controls all music and effect sound in game. In this game, the Windows provided function *mciSendString* is used to play sound.
   Class members:

   - $SoundEffectList$: a list of sound effect file names
   - $TrackList$: a list of music track file names
   - *currentSound* : the file name of current sound being played
   - $BGSoundVolume$: the volume of the background music
   - $EffectSoundVolume$: the volume of sound effects

   Important class methods:

   - *startAudioEngine*, *cleanAudioEngine* : send commands to open sound files for use later / close sound files before ending the program

- ■ *playTrack* : play a music track. Can control whether the track loops through a bool parameter
- ■ *playSoundEffect* : play a sound effect

6. **cObstacle**
   An abstract class that acts as the base for obstacles in the game
   Class members:

   - *topleft* : current coordinates of the object
   - *speed* : how much the object will move each time
   - *isStop* : whether the object is currently stopped (this prevents the object from moving)
   - *pTexture*, *pLTexture* : contains information on how to display the object onto the console terminal

   Class methods:

   - ■ *move* : change the current position of the object according to its *speed*
   - ■ *stop*, *resume* : set *isStop* to true or false, respectively

   Abstract class methods:

   - ■ *getType* : returns the type of the object as a **char**. Each derived class has a unique return value
   - ■ *copy*, *construct* : copy object and set certain attributes. Used with *copyObject*, *constructObject*.

7. **Derived classes** of **cObstacle**:
   cLion, **cRhino**, **cCrocodile**, **cTruck**, **cHelicopter**, **cMotorbike**
   Each class has its own static member defining its sprite. There is also a static bootstrap member for each class, initialized at the start of the program (using the default constructor of each class). These objects help setup the static *objects* member in **cObstacle**.
   cEnvironment is also a derived class of cObstacle. It contains some attributes as *hasEvent*, *friendly*, *hasFrameMove* to know this environment object is safe or not and what will happen when this object overlapped with a player or an another object. This class is also the base class for **cRiver**, **cLilyleaf**, **cTrafficLight**, **cCoin**.

8. **CWidget**
   A base class that defines functions to control the background, button, label and bar displayed in console window.
   Class members:

   - Inherance attributes: *isVisible* (is displayed or not), *topLeft* - *botRight* (position of objects), *offset*, *WidgetFace* (thing to display), *parentWindow* (control the parent of this object)
   - Static members: *window* - the main window during all game sections

   Class methods

   - ■ *show*, *unshow* : control the display of this window

- ■ static $createMainWindow$ - initialize the $window$ member and create the main window for game.

Derived class:

- **cDWindow:** is the window that control the background of main window has no more attributes.
- **cButton:** is a button that displayed in window console that can be interacted by user, so it has methods onSelect(), onDeSelect(), onEnter() to control these functions. It also has members 0Topleft, 0Topright to control the postion and the size of this button in the main window.

9. **cGameEngine**
A pure static class to do all functions about Window Console and rendering / drawing Objects. Sets up **cGame**. This class uses double buffer technology to draw objects in window console.
Class member:

- $buffsize$: - a COORD data type to store width and height of buffer (console width and height)
- $hBuffer1, hBuffer2$ - HANDLE data type: handles for double buffers tech
- $mainBuffer, reservedBuffer$ - CHAR_INFO* data type: Buffers
- $count$ - int: a counter for double buffers tech

Some important class methods:

a) Pixel processing
These methods update the pixel information of the buffer array to be output to console.

- ■ $fillEffectivePixel$: copy pixel information from a source to a destination array, except for blank characters. Only the pixels that make up the source sprite will be overwritten in the buffer.
Knowing the size of the destination buffer $dest_x$, $dest_y$, the size of the source sprite src_x, src_y, and the location in the destination array $position_x$, $position_y$ to insert the sprite into, the index to replace is calculated as follows:

$$\begin{aligned} replace_x &= (position_x + currentIndex \,\%\, src_x) \,\%\, dest_x \\ replace_y &= position_y + \left\lfloor \frac{currentIndex}{src_x} \right\rfloor \\ replaceIndex &= replace_y \times dest_x + replace_x \end{aligned}$$

$currentIndex$ is the index of the current element in the source sprite to copy, and $replaceIndex$ is the index of the element in the buffer to be overwritten
- ■ $replaceBlankPixel$: copy from a source to destination array only where the destination contains a blank character.
- ■ $replaceAllPixel$: copy all pixels from a source to a destination array.
- ■ $paintBucket$: change the color of all pixels in an area in a destination array

b) Widget display

- *showWidget* : display a **cWidget** on screen

```
function showWidget($widget$)
    Copy $widget$ texture into $reservedBuffer$
    if $widget$ is child of cWidget $widgetParent$ then
        replaceBlankPixel($reservedBuffer$, $widgetParent$ texture)
                                  ▷ reserved buffer filled with background
    end if
    Print $reservedBuffer$ to console at widget location
end function
```

- *unshowWidget* : hides a widget, removing it from screen

```
function hideWidget($widget$)
Ensure: $widget$ has parent $widgetParent$
    replaceAllPixel($reservedBuffer$, $widgetParent$ texture)
    Print $reservedBuffer$ to console
end function
```

- *HighLightButton* : display a highlight border around a button widget

```
function HighLightButton($button$)
    Fill $reservedBuffer$ with highlight color, buffer size is size of button including highlight border
    Print $reservedBuffer$ to console        ▷ this overwrites the button
    showWidget($button$)                     ▷ redraw button
end function
```

- *UnHighLightButton* : remove the highlight border around a button widget
  Instead of filling the screen with a color light its highlight counterpart, this function fills the screen with the parent (background) widget.

c) Console draw

- *refreshBackGround* : fill *mainBuffer* with the background image. Can trigger a console draw using a bool parameter.
- *fillScreen* : draw the contents of *mainBuffer* to console

d) Update object information

- *renderPeople*, *renderObstacle* : Call *fillEffectivePixel* to push objects' textures into *mainBuffer*

e) Special effects

- *playEffect*, *playFlashEffect* : draw special effects

f) Initialization and clean up

- *startEngine* - bool: call setting up functions for window console and intialize for double buffers tech engine.
- *cleanEngine* - void: cleanup. Delete memory allocated to buffers and close handles.

10. **cPeople**
    Defines the character that players will be controlling in the game.
    This class is similar to **cObstacle**
    Class members:

- *skin* : a list of possible sprites for the player, stored as **Texture**s. Loaded when object is created by the *assetLoaders* method of **cAsset**.
- *pMotionFrame* : a pointer to the sprite (element in *skin*) currently being used. Changes accordingly when the player acts.
- *topleft* : current coordinates of the player
- *mState* : indicates whether the player is dead or alive
- *passLevel* : indicates whether the player has reached the goal
- *moveCooldown* : a timer that prevents repeated and instantaneous vertical movements. Will only allow the player to move when it reaches 0.

Class methods:

■ *move* : moves the player.

```
function MOVE
    if $moveCooldown > 0$ then
        $moveCooldown \leftarrow moveCooldown - 1$
        Exit function
    end if
    Read keyboard input for any directional keys being pressed
    if player is not at and moving towards screen edge then
        Set player to new position by adding or subtracting a set amount
from the $x$ and/or $y$ coordinates correspoding to the direction.
                                  ▷ see setCoordinates below for move amount
        procedure SETCOORDINATES
            $topleft_x \leftarrow topleft_x \pm 6$, and/or
            $topleft_y \leftarrow topleft_y \pm$ [sprite height]
        end procedure
        Change $pMotionFrame$ to point to the sprite facing the moving
direction
    end if
    if move was vertical then
        $moveCooldown \leftarrow 8$
    end if
    if player has reached the goal then
        $passLevel \leftarrow$ true
    end if
end function
```

11. **cGame**
Controls main functions of a game.
Class members:

a) Game objects

- *liveObstacles* : a list of obstacles currently present
- *livePeople* : a list of players in the game
- *environmentObject* : a list of other objects in the game that players are not required to avoid
- *coins* : a list of coins currently present

b) Game information

- *gameOrder* : the number of players in the game
- *gameLevel* : the current level
- *currentTheme* : the current level theme. Determines the background.
- *isPause*, *isExit*, *isLoad*, *isLose* : flags for current game status. Indicates whether the game is paused, about to exit, is loading a save file or the player has lost, respectively.
- *totalPoint* : the player's score
- *totalTime* : amount of time elapsed

c) Skill-related

- *SkillIcon* : a list of widgets that display skill icons
- *Skillcooldown* : a list of labels that display the cooldown time for each skill
- *defaultSkillCooldown* : the list of default cooldown time for special skills
- *freezeTime* : the active duration of the *Freeze* skill

d) Window management

- *listWidget* : a list of **cWidget** objects to render on screen (main window). Contains buttons and smaller windows.
- *listLabel* : a list of **cLabel** objects to render on screen. Each label is a block of text.
- *window*, *mainMenu* : static window objects to initialize when the program starts

e) Threads Each thread loops infinitely until a flag signaling the game's end is set.

- *collisionThreadHandle* : thread to check for collisions
- *drawThreadHandle* : thread to draw to console
- *randomStopThreadHandle* : thread to create an event to stop some obstacles at random intervals

Important class methods:

a) Initialization

- ■ *InitGame* : Initialize static objects to prepare for the game to start
- ■ *onGameReady* : Create the main menu screen and start the main game loop

b) Game start

- ■ *spawnObstacle* : read a level setup file and create obstacle objects at the specified locations
- ■ *spawnEnvironment* : create environment objects based on the chosen theme
- ■ *spawnCoin* : create some coins randomly on the map
- ■ *spawnPeople* : create players at their starting location

c) Game logic

- ■ *isImpact* : checks for impact with obstacles

```
function isImpact
    for each live player $P$ do
```

```
            for each obstacle $Obj$ do
                if $P.hitbox$ overlaps with $Obj.hitbox$ then
                    Set $P$ state to dead
                    $isLose \leftarrow$ true
                    Set up variables to display effects with $P$ and $Obj$
                end if
            end for
        end for
    end function
```

- ■ *randomStopThread* : randomly stops a lane

```
    procedure RANDOMSTOP
        for each lane $L$ in the map do
            Reduce remaining stop duration $duration$
                                        ▷ this also increases stop chance
            if $duration \leq 0$ then
                Resume objects in lane $L$
                Get random number $rand$ from $-10000 \rightarrow -5000$
                if $duration < rand$ then
                    Change traffic light in lane $L$ to red
                    Randomly set stop duration from $2500 \rightarrow 7500$(ms)
                else
                    Change traffic light in lane $L$ to green
                end if
            else
                Stop objects in lane $L$
            end if
        end for
    end procedure
```

d) Save and load

- *save*, *load* : save the game or load a save file. See section 3 for file format.

e) Scoring

- ■ *calculateTime* : calculate the time elapsed from when the level started to when this function is called
- ■ *calculatePoint* : calculate the score earned for completing the level
The score for the level is calculated with the formula:

$$score = 100 + bonus_{time} + bonus_{coin}$$

In which:

$$bonus_{time} = \begin{cases} 120 \text{ if } time \leq 5 \\ 70 \text{ if } 5 < time \leq 10 \\ 35 \text{ if } 10 < time \leq 15 \\ 15 \text{ if } 15 < time \leq 20 \\ 5 \text{ if } 20 < time \leq 25 \\ 0 \text{ if } time > 25 \end{cases}$$

($time$ is the time it took to clear the level)

$$bonus_{coin} = [\text{number of coins collected}] \times 30$$

f) Clean-up

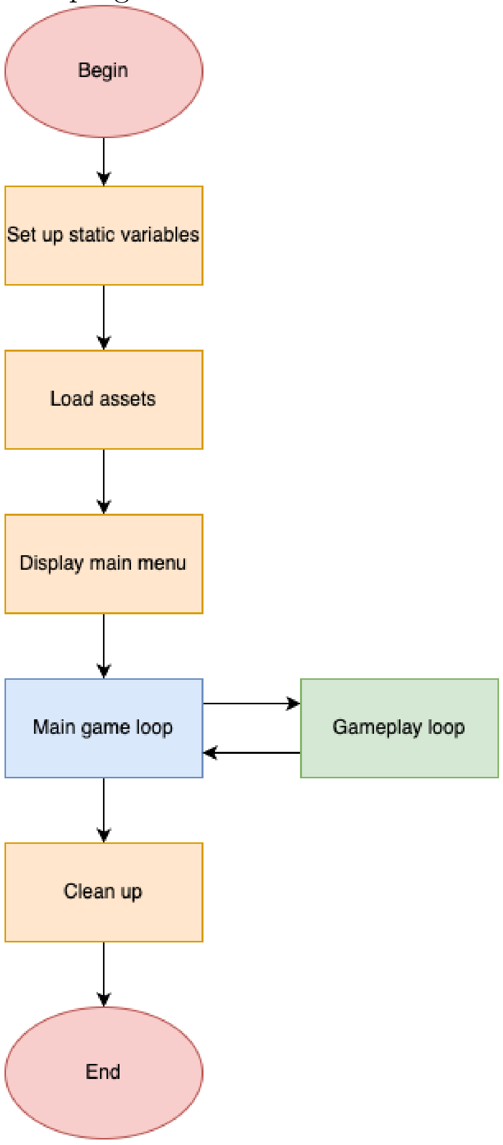
- ■ *clearObjects* : clear *liveObstacle*, *livePeople*, *environmentObject* and *coins*, deallocating memory occupied by those objects
- ■ *clearUI* : clear *listWidget* and *listLabel*, deallocating memory occupied by those widgets
- ■ *clearSkillUI* : clear *SkillIcon* and *Skillcooldown*, deallocating memory occupied by those widgets

12. **Miscellaneous classes**

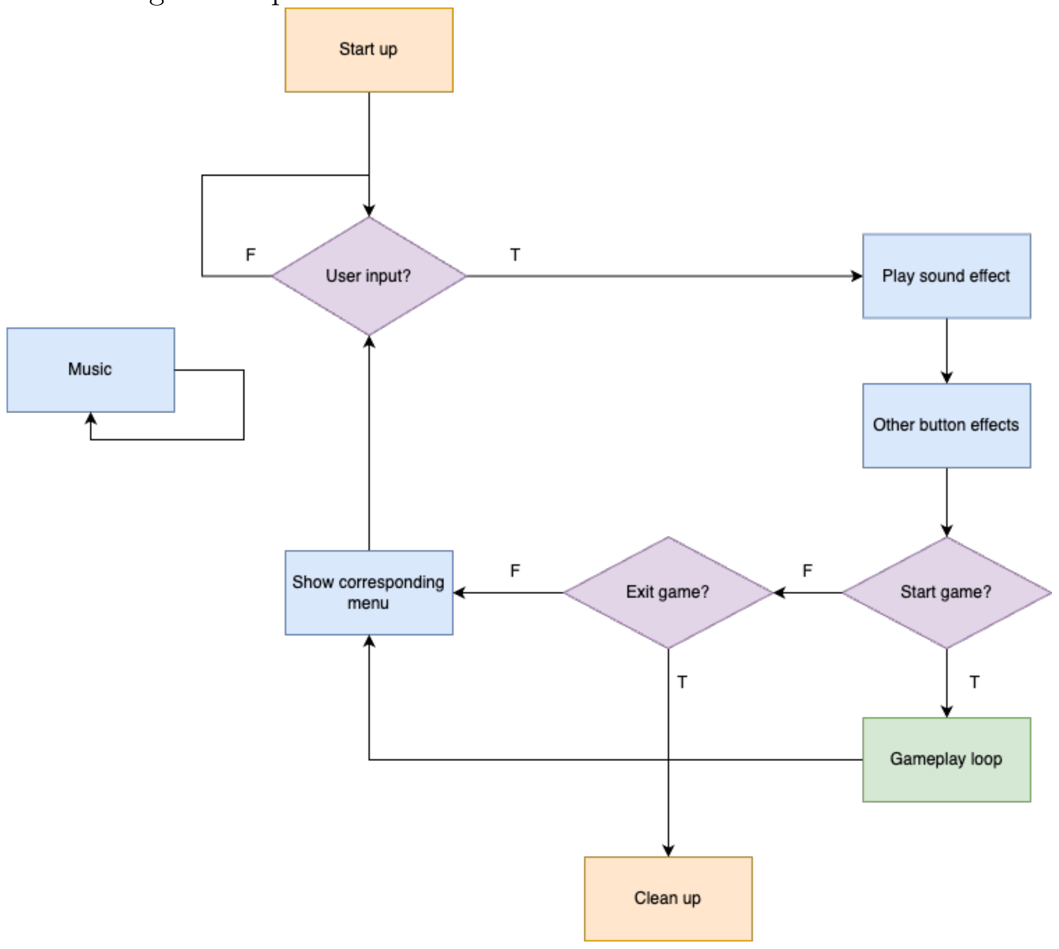
    - **Time** : stores time in day, month, year, hour, minute, second. Has method to convert a Unix timestamp to standard format.

6 Program flow

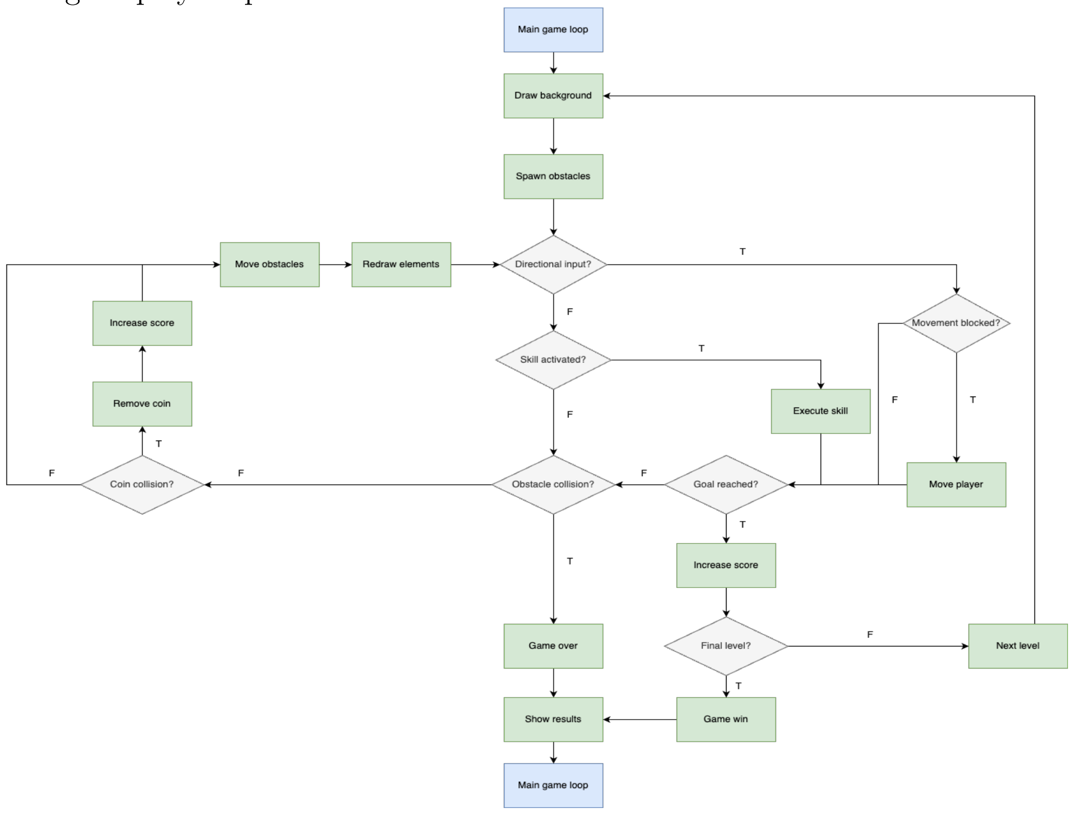
The main game uses two thread. One is sub-thread which is used to draw and update background, objects, side windows. The other is main-thread is used to check collisions, check if the player has completed the level and get keyboard inputs to interact with user. The folling diagrams illustrate the program flow
The program flowchart:



The main game loop:



The gameplay loop:



7 Demo video

A demo of the project can be found here:
https://youtu.be/PdXGWGMGf4E?feature=shared