LEWIS Arthur
H00341312

Game Project Assignment

introduction

my project was initially supposed to be a story-driven game but experiencing limitations with the BGI library I change the scope of my project to be more of proof of concept by developing a more advanced system and using complex techniques like multithreading and a complete and advanced movement system.
Even though it is a proof of concept the work produce took around 30 hours of making and is in my eyes complete enough.

How to Navigate?

use the **W A S D** keys to move and the **SPACEBAR** to change level

Summary of the headers files

- ENGINE_ini.h
  - initialize basics libraries
- ENGINE_Playerinput.h
  - Threaded Component
  - Binds
  - Movement Physics
    - smooth locomotion (acceleration and deceleration)
  - Collision Physics
    - double collision check
    - smooth collision
- ENGINE_Renderer.h
  - Threaded Component
  - Double Buffering
  - renderer loop
    - implementation of the double buffering
- ENGINE_Structure.h
  - hierarchy
  - Pawn
  - Wall
  - Leftover structure that are not in use
  - SetWall()
  - SetLevel()
  - ClearWall()
- ENGINE_Threadmanager.h
  - Thread Calling functions
    - CallPlayerInputThread()
    - CallRendererThread()

The Structures

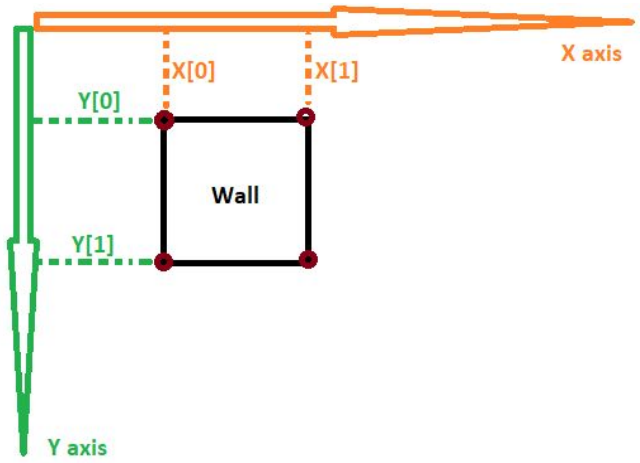
the Game use Structures as arrays that hold information that is inherited or used by different objects that have differents purpose.
the principle structures in this game are “Pawn” and “Wall”

they both contain two other structure:

- Location
  - cartesian coordinates of a location
  - cartesian coordinates of the hitbox corners location (represented as an array of index 0 or 1 representing the minimum for 0 and the maximum for 1 of each corner)
- LocationOffset
  - cartesian coordinates of an Offset (can be understood as the speed of the pawn or wall)

here is an example of what a hierarchy could look like for our Wall structure

- ➔ Wall
  - ◆ Location
    - x and y coordinate of the wall set by the user
    - x[0] = xmin(xmin being the lower x coordinate) , x[1] = xmax(xmax being the upper x coordinate)
    - y[0] =ymin(ymin being the lower y coordinate) , y[1] = ymax (ymax being the upper y coordinate)
  - ◆ LocationOffset
    - x = 0 and y = 0 (because our wall doesn’t move)



to be able to use multiple “Wall” or “Pawn” they had been set up as arrays to make their updating easier, however “Pawn” of index 0 refers to the player and is the only one ever used.

Walls are placed in the scene with the **SetWall()** function, this function sets up the location and the hitbox of the wall by taking the location of the wall, the wide and height of the wall in

argument. it also takes in argument the **WallID** which is used when all the wall need to be updated to be shown on the screen, it represents the index number used in the wall array structure.

A scene is created by the **SetLevel()** function which is setting all objects of the corresponding level at the right place.
as the name suggests it is setting a level up."it is notably use in the game loop inside main()"

Finally, we got the **ClearLevel()** function. this function put every wall far out of the screen in case a wall in the following level is not used.

Threadmanager

The thread manager is just an header file that initializes the two main thread ***(Of associated function PlayerInput() and RendererStart())*** to be called in the **main function.**
This game make use of multithreading to ensure a smooth experience by computing different tasks in parallel

Player (Pawn) Locomotion system

The locomotion system is at the center of this game, it contains smooth acceleration, deceleration, and smooth collision between a Wall.
To ensure a smooth experience I used multithreading to have part of the code run independently of the rest.

The physic can be adjusted at any time by modifying the 5 basic physic constant being

```
const float FLOORGRIP = 0.7F;
const float SPEED_1 = 7;
const float MAX_SPEED = 25;
const float MIN_SPEED = 1;
const float GAMESPEED = 1;
```

The "Movement Loop" is independent of the refresh rate of the game and is represented by the constant **GAMESPEED** being by default equal to 1 however because it doesn't impact the game at this value I will choose to omit it.

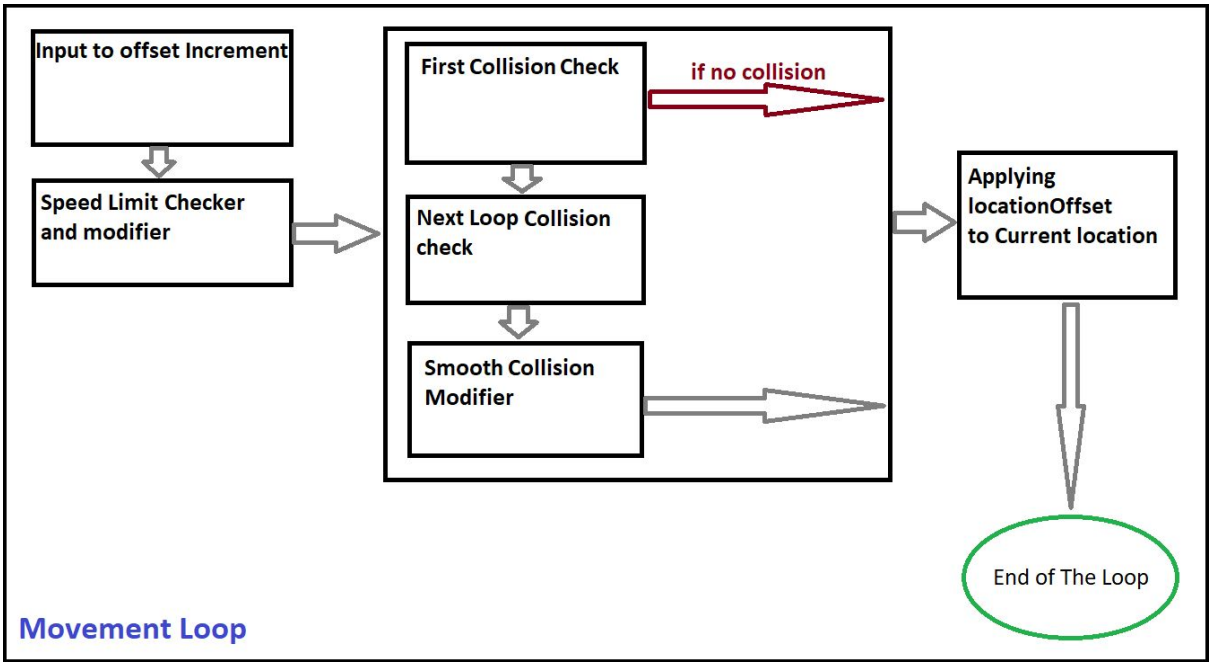
Earlier we talked about an Acceleration and a deceleration feature, our character can attain a max speed of 25 and a min speed of 1 being out of this range either lock the speed to 25 or to 0 (not moving) the floor grip is the constant that determines the deceleration speed, in the game, it looks like if the character is running on slippery ice or hard concrete.
SPEED_1 is the increment at which the speed increase per loop *(every 64 ms)*.
FLOORGRIP is a factor that determines the deceleration rate, it works as follows.

if FLOORGRIP is set to 0 the character will only move while a key is pressed and will be stopped as soon as it is release

if FLOORGRIP is set to 1 the character will quickly accelerate and then just derivative as if it was in space

FLOORGRIP can, of course, be set at any value in between (by default the game use 0.7)

The Physic in The game is run by one loop called the Movement Loop it works as follow



Input to offset Increment : Read key input using GetAsyncKeyState() function and increment the corresponding location offset with the SPEED_1 Value.

Speed limit checker and modifier : Check if either the x or y Pawn's LocationOffset component is in the 1 to 25 range(MIN_SPEED = 1;MAX_SPEED = 25;).
if it is under 1 the offset is set to 0 , if it is above 25 the offset is set to 25.

First Collision Check : Check if the pawn will be inside the hitbox of any of the walls if the offset is applied to the current location of the pawn.

Next loop collision check : Check if the pawn will be inside the hitbox of one of the walls if the base offset increment (SPEED_1 = 7) is applied to the current location of the pawn.
PS: this is one of the most important checks because without it the collision could be executed earlier than it should if the speed is too high,as such it used to result in a hard stop followed quickly by another motion toward the hitbox then another stop.

Smooth collision Modifier : Enable motion on the axis that is perpendicular to the edge of the hitbox that the pawn collides to.
It allows the pawn to slide again a wall without losing speed on the related axis.

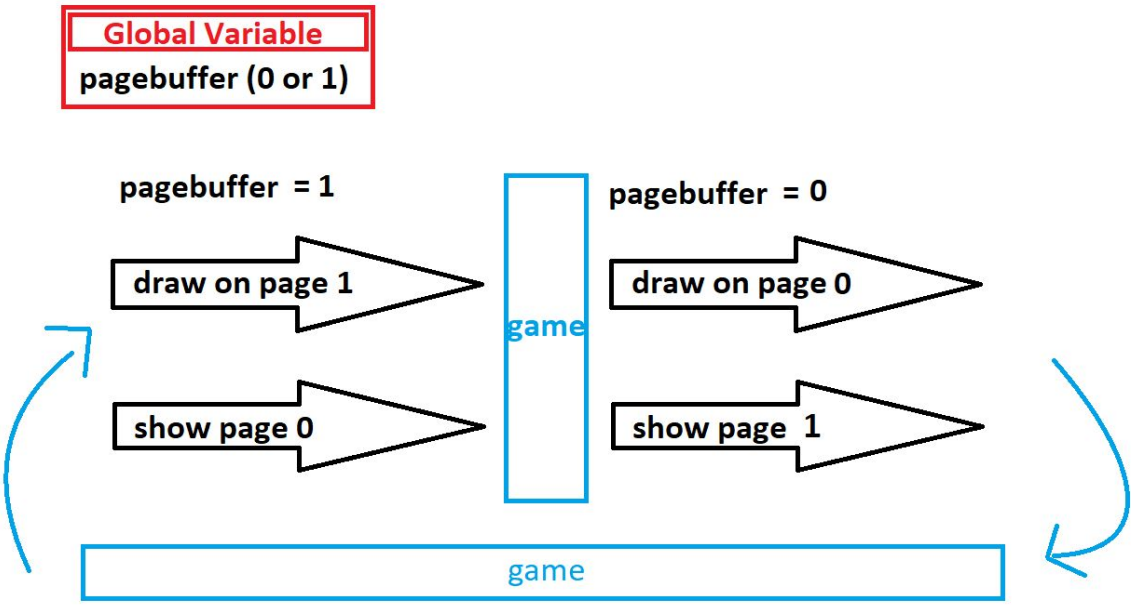
Applying LocationOffset to Current location : Compute the new current location by adding the offset resulting from the checks and the modifier to the current location.

Renderer
Like the physic the renderer use multithreading to run his code.
it is composed of two thread

- hRendererThread
    - is the main rendering loop it goes from all the objects of the scene like the player controlled pawn or the walls
    - it also initialize before the loop start and use hRendererDBThread

- hRendererDBThread
    - is use for double buffering and remove flickering at the cost of an increased latency

The double buffer is the most important part of the renderer because it make the display a lot less glitchy and make the experience much more enjoyable , here is how it work :



This simply make sure the page is fully drawn before it can be shown to the user by swapping between two pages (0 and 1) each render loop so that while one is rendering the other one is shown .

code that I didn't write:

- stdbool.h
- the sign() function inside ENGINE_Playerinput.h :
    - https://stackoverflow.com/questions/1903954/is-there-a-standard-sign-function-signum-sgn-in-c-c

Link to my game demo video :
https://www.youtube.com/watch?v=PKVXEs_Tx8s&feature=youtu.be