Loop n' Loops

A CS101 project developed by Sankalp Bhamare, utilizing the simplecpp library.

The project features many OOPS concepts and C++ functionality.

To get started read the instruction on how to compile.

Whats the game about ?

Well its basically a game where our task is to collect coins, the thing is that we have to catch them using a lasso. Basically we have to throw the lasso in a certain way that it catches the moving coin. But it's not that easy some levels have a timer , dangerous objects like bombs which should not be caught. It also has magnets which make coin collection very easy, as the coins just sweep close to lasso when in a certain range.

What are the controls?

Well the controls are quite simple and intuitive , there is a in game tutorial which teaches you the game controls, but still anyways:

> **Throw Controls**
>
> `W` : Increases lasso throw speed.
> `S` : Decreases lasso throw speed.
> `A` / `D` : Tilt the angle anticlockwise / clockwise respectively.
>
> **Lasso Controls**
> How to loop and yank the lasso
> `K` : Throws the lasso, basically releases it.
> `L` : Loops the lasso, meaning when pressed near an object it loops the lasso and catches the object.
> `M` : Yanks the lasso, basically pulls the lasso back to the starting point and resets it.
>
> A nice trick would be to place the left hand on `wasd` and the right hand on `klm` .

Note inorder to make the game little tough you can only **loop** the **lasso once**.

Features Implemented:

1. Make the coin go in a parabola
2. Make more than one coin appear at a time; at random times
3. Make bombs which should not be collected; score is subtracted if you catch a bomb
4. Impose a time limit for a game level
5. Some number of “lives” remaining for player;
6. Once in a while, a magnet is thrown; if caught, it has the ability to attract nearby coins; magnet expires after a delay
7. Enter name, maintain high score against name; optionally store this info in a file to restore it on next run
8. Make a help page with brief text explaining key controls

   Additional Features

   1. Extendable levels
   2. Multi thread input system: this custom system keeps a separate thread running to handle input Event this essentially prevents input buffer getting fulled. Essentially this allows game keys to held down and smooth control of the game.
   3. Simple minimal in game ui showing the basic state of the game.
   4. In game alert and confirmation screens give the game a proper control.
   5. A play menu.
   6. The early levels are like ingame tutorials that one can play around.
   7. `beginFrame()` and `endFrame()` optimization has been used in rendering this essentially makes canvas update much faster.

> **Suggestion**: Before glancing through the code first try playing the game, things might become more easy to comprehend after playing the game.

How to begin

Macos and windows users would require to compile and build there own executables.

For Ubuntu users a prebuilt binary is attached they can download and run it

> **Note** : You may need to give execution permission
> use `chmod +x execName` and then `./execName` , where execName is the path to the executable.

Structure

First let us get an insight on the structure of the project:

```
├── GameHandlers
├── GameObjects
├── UI
├── Manager
├── Misc
└── main.cpp
```

How to proceed further

1. **Misc** (Miscellaneous stuff)

   All the side import side features are hosted here.

   ```
   ├── GameConstants.h
   ├── Magnet
   └── Vector2D
   ```

- **GameConstants.h** : this file contains some of the most important constants of the game like `coin_size` , `window_x` , `step_time` .
- **Magnet**: this part contains code to handle how the magnet works
- **Vector2D** : this contains the Vector2D class which handles vector operation, like addition multiplication, and many more, essential the core physics is based on this class.

2. **GameObjects**

   Literally the heart and sole of the project all the important sub parts which move and are part of the game play are here , right from the `coin` to the `bomb` .

   ```
   ├── bomb
   ├── coin
   ├── lasso
   ├── MagnetGiver
   └── MO
   ```

   - **coin** : this class contains our favorite object the coin , which is essentially what we are after, you can start reading from this class.
   - **bomb** : this contains the bomb class which essential is a dangerous in game object that if caught by the lasso causes some serious problems
   - **lasso** : aka the player in this game this is the class around which we are supposed to play around it basically handles all the things right from throwing the lasso to catching the coin
   - **MagnetGiver**: this is responsible for spawning the magnets in game , which can be collected by the lasso.

   > **MovingObject**
   > The importance of this class can not be estimated it is the sole class responsible for handling physics and all. Right from moving things to getting them oriented on screen. You can bet that everything that moves is connected with the `MovingObject` class.

3. **Manager**

   Well everything we talked about till now needs some way to spawn into the game , this is handled by the manager class.

   ```
   ├── BombManager.cpp
   ├── BombManager.h
   ├── CoinManager.cpp
   └── CoinManager.h
   ```

   - **BombManager** : it is responsible for the bombs.
   - **CoinManager** it is responsible for the coins.

   > Essentially both perform similar tasks, like spawning coin, random spawning. Updation / Event Loop handling of multiple coins / bombs. also there interaction with lasso is interfaced through them only.

4. **UI**

   Nothing much to explain here. this just contains a lot of simplecpp commands to create simple and elegant UIs for the game.

   ```
   ├── Coin
   ├── Heart
   ├── Key
   ├── PrelimRenderer
   ├── TimeLeft
   └── Util
   ```

   Well basicsally the component names say it all Coin : shows coins collected, Hearts: shows hearts left...

   > **PrelimRenderer**
   >
   > This is the main / biggest UI class it performs the job of integrating / merging all the small widgets together. It also handles the updation of the individual

> components. The component is the one which pushes `GameState` to the canvas.

5. **GameHandlers**

   This is one of the most important / essential part of the game, it essentially pulls all the strings and wires of the game, explain this part is the most crucial.

```
├── Engine
│   ├── GameEngine.cpp
│   └── GameEngine.h
├── HighScore
│   ├── HighScoreData.cpp
│   ├── HighScoreData.h
│   ├── ScoreBoard.cpp
│   ├── ScoreBoard.h
│   ├── ScoreSubmit.cpp
│   └── ScoreSubmit.h
├── LevelClass
│   ├── GameLevel.cpp
│   └── GameLevel.h
├── LevelManager
│   ├── LevelManager.cpp
│   └── LevelManager.h
├── Levels
│   ├── Level1
│   ├── Level2
│   ├── Level3
│   ├── Level4
│   ├── Level5
│   ├── Level6
│   ├── Level7
│   ├── Level8
│   └── Level9
└── State
    ├── GameState.cpp
    └── GameState.h
```

   Overview:

   - **Engine** : contains the main game loop / calls up all the render updates, process all the user input.
   - **HighScore** : Responsible for maintaining highScore in game stores it to an `score.dat` file. Allowing to maintain an leaderboard
   - **LevelClass** : This project uses some real OOPS things , this `GameLevel` class allows to implement game levels in the game, the idea is if we want to create a new level we can just inherit this class and start modifying the game settings and scrippting , basically the game can be extended without even touching the engine. Custom functionality write from object spawning to level scoring / achievements can be implemented in inherited classes.
   - **LevelManager** : well multiple levels do require some class to manage their transition startup and running, this exactly happens here, things like which level to open, current level, finding a level all happen here.
   - **Levels** : this where the game levels are kept , this all have the `GameLevel` base class, all these just tweak the current game.

- **State** : All state of the game is handled here , state like the player health coin collected , time left etc. This is essential the part that contains info and allows it to be easily shared between multiple components , espicially the UI and engine, this basically ensures that the engine doesn't need to update the ui rather it just updates the state.

Compile

To compile the source code use the command specified below.

```
s++ *.cpp */*.cpp */*/*.cpp */*/*/*.cpp -I. -pthread
```

```
 -I. : is to  for the inclusion of header files in the current directory
-pthread : is required for the multithreaded part used by the code.
```

> This command may not be that nice as it involes using '*' in listing all cpp files.

In the project there is also a runLinux script attached which invokes the above without using wildcards.

> For windows the script similar to the above can be used but it may require linking of other libraries like x11 x11r8 etc..

CMake

This project has cmake support , but you would need to open cmake file and make few changes:

1. You would need to include the simplecpp header file directory , this cannot be auto configured as it is system dependent.
2. X11 libraries must be properly linked check the path specified in the cmake file and your system files to make sure it matches.

3. Macos should have similar compilation process to ubuntu.

Incase of issue contact: bhamare.sankalp@gmail.com -Sankalp Bhamre.

> Project has been tested on Ubuntu 20.02 and 18.02. Macos and Windows OS users are adviced to use Method 1 for compilation of code as it is more easy.

Binaries

Binary built on ubuntu 18.02 is available here.

binary_LoopNLoopsUbuntu18.02

> Compatible with 18.02 and Up.