ELEC-A7151 Object oriented programming with C++
Winter 2022

Group 5

Ashraf Syed	906586
Po Lim	648093

Introduction

This is a two-man project to create a tower defense (TD) game in C++ with the help of external libraries, such as, Simple and Fast Multimedia Library (SFML). Originally a four-man group was shrunk to two people due to personal reasons of the other members, but this was agreed on by all the parties. As the group was rather small, we did not have a project leader, the group members developed whenever they had time and took the initiative to build classes on their own. To prevent overlapping work, it was agreed on a fast pace on who would develop what part.

The goal of this project was to learn how to use external libraries, CMake, and using C++ in a real scenario while also having freedom to design on your own. In order to achieve this, the group had meetings at least once every week and towards the end of the project, we had a sprint. In this case, we had a workshop style meeting where all the members (two of us) sat on a Microsoft Teams call for the whole day to code. This allowed us to communicate and share the screen in order to debug and explain the blocks of codes to each other in order to extend it.

The resources provided by the university was our assistant, Mark Heidmets and a SFML development book by Haller J., et al (2013). SFML's own site also provided documentation and tutorials on how to use the library it provided. Other resources used were cppreference for C++ and Stackoverflow on how to use the code itself if the documentation was not enough.

Overview

The software a basic TD game. It can build towers on specific tiles, but the path tile, and the tile where a tower already exists are disallowed. Building towers requires money and if the user does not have enough money, the tower won't be built. The current implementation has five rounds, each with increasing difficulty. The user must first clear the round before clicking "start round" button, otherwise clicking it does nothing. The users are allowed to build towers mid round and after the round. If the monsters reach the end of the path the player will lose life and with enough life lost, the game. The current implementation of the software uses fixed time steps which are set to 1/60 seconds. This is used to calculate the correct fire rate for the towers as well as the movement speed of the monsters.

The current framework is lacking sell option and upgrade option for the towers. It also does not have range displayer for the towers. The software does not have sprites for the monsters or towers. The current implementation also does not log the end location for the path, but it does build the background and the path correctly from a text file. As long as the path doesn't cross itself, the pathfinding should work properly but game end won't. Path also has to go from left to right. Audio is also not included in the software. The software also does not support resize events, even though it can be logged. The size of the window should be 1200x800 for it to work properly as certain coordinates are not rescaled upon resize event.

Software structure

The software follows the current structure. Main creates the game class which then runs the game class itself. The run method is permanently looping through the events, the game logic and then finally it renders everything based on the game logic. The events are either flags, they either trigger something which is then updated through game logic and then the flag turns off or they can also be constructors for towers. The tower is built and then pushed into a vector. Rendering is also based off of this. Figure 1 shows the structure. The major classes are game, map, base_monster, and base_tower. The rest of the classes are supporting.

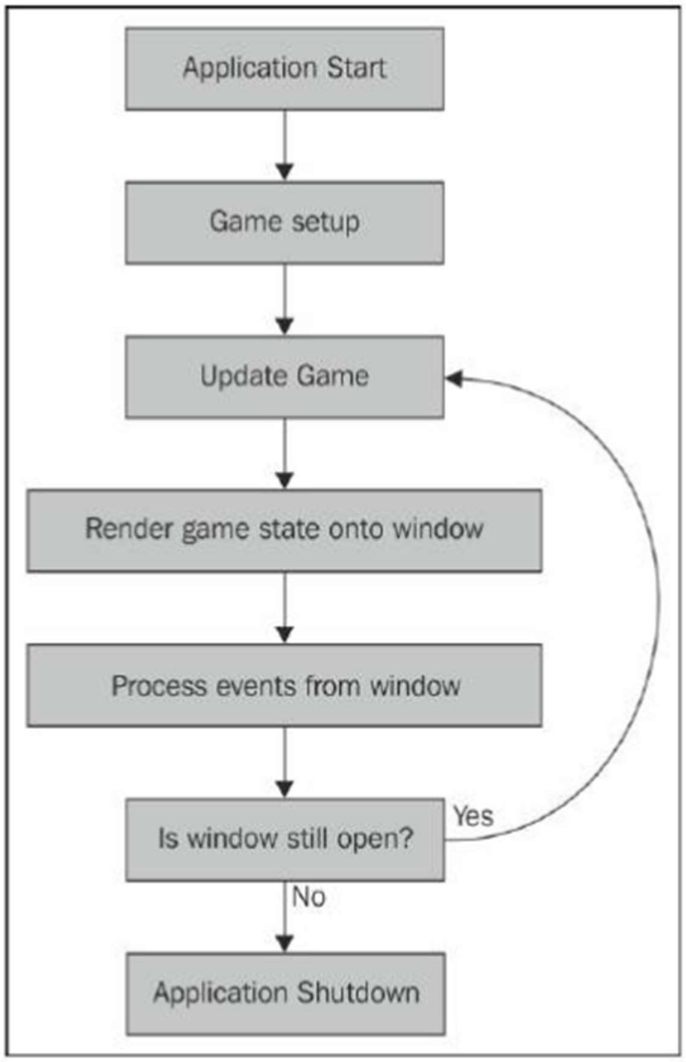


Figure 1, software diagram (Haller, J., et al. (2013), p. 40)

Game class

The game class itself holds three major methods and two major fields, the methods are for processing the events, updating the game logic, and rendering to screen. The major fields are standard vectors, capable of holding unique pointers of certain base classes, such as the tower base class and monster base class. Rest of the methods of the game class and or fields are mostly to support the major fields and classes to make the code cleaner. Boolean fields are mostly used for flagging in the events. The game class should be created once and once only, and the game should run inside the method of the game class.

The reason to make fields of the game class to hold unique pointers of the base class over the base class itself is to ensure that the vector will always hold an entity that is of the same size. There was a realistic chance that derived class was of different size compared to the base class. We also did not want to use raw pointers as that would've made us most likely use the keyword "new" and as such, also use the keyword "delete". Using standard library allows us to preserve RAII more easily.

Map class

The map class reads from a text file that must contain only characters 0.0 and 1.0, separated by a single whitespace and possibly newlines to denote the next line of tiles. This is then parsed by the class to construct disallowed tiles and the path which is then passed down to other classes, namely the tower and monster. The map building is initiated upon the construction of the game class, to ensure that it is only ran once and that the path exists.

Monster class

The monster class itself is used to hold all the information the monster needs. These are its health points (hp) and its movement speed. The monsters also have their own update and render function and other fields to support the game class itself. When the monster dies, it should award the player with some cash in order to build more towers. As the monster class is a base class where different monsters are derived from to create different types of monsters, it should have a virtual destructor and other virtual functions. The monsters know their own position, and they also fetch the coordinates of the path where they are supposed to move. The movement speed is controlled by the frame time and the distance between its own position and the coordinates it is trying to reach. Figure 2 shows the relationship between the base class and derived classes.

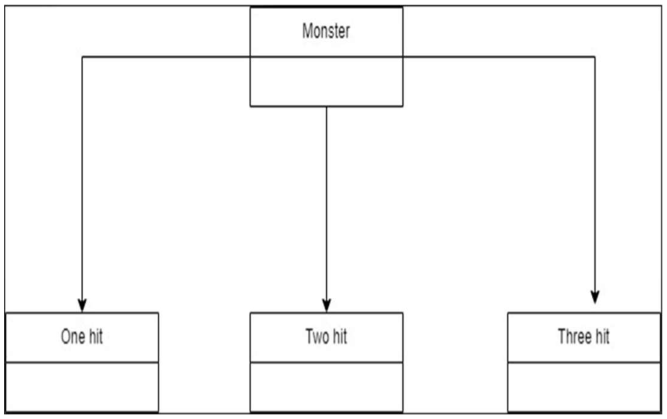


Figure 2, inheritance diagram

Tower class

The tower class is also a base class where other towers was supposed to be derived. This was not, however, necessary but to allow extension of the code it was kept as a base class in which other tower classes could be derived from. Different buttons build the same derived tower class, with different constructor parameters and with different costs. The tower knows its own position, and it is given the position of the monster. Then it checks whether the monster is in range and if it is capable of firing. If both of these return true, the tower will shoot and the capability to fire is set to false. After this the capability to shoot is set to false. Shooting happens with the help of bullet class.

Bullet class

The bullet class was originally created to give the game the capability to do projectiles. When the monster is in range of the tower and it is capable of shooting, the tower will then automatically create a bullet class and then append it into a vector field of the game class. This field will then be looped which allows the game to update the location of the bullet. When collision between the monster and the bullet is detected, the monster hp is reduced by one and the bullet disappears. The priority is the monster that is closest to the end.

Button class

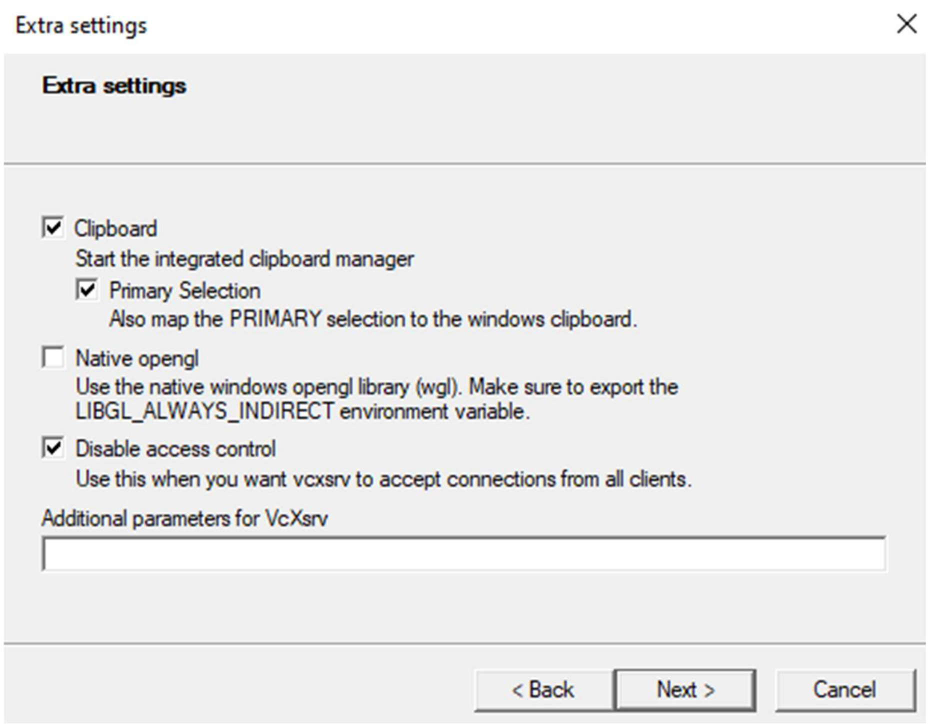
The button class was created in order to differentiate the buttons that are clicked. SFML's own implementation of sf::IntRect has a method called "contains" that allows the user to check whether certain coordinates are inside the IntRect. This is used to detect whether the button is clicked and being a class, it allows us to create fields, such as "button type" in order to know exactly what button was clicked. The button building is initiated upon game construction and is rendered via sf::RectangleShape as the IntRect itself is invisible. This means that a simple rectangular image is projected on top of the button, to visualize it.

Instructions & compilation

The building requires SFML 2.5.1 library, the CMakeList.txt file is provided for the user. The compilation happens in C++17 standards with g++ compiler. The users are expected to have SFML properly downloaded with all the dependencies. After "cmake ." and "make" are called, the users should be able to execute the program by writing "./MyGame" in the terminal. The source code is provided in gitlab. This software is not tested with MSVC and in Windows. The software is built and compiled in a virtual environment of WSL2. The users should also be using X11 server and have the following settings ticked in the X11 launcher as seen in figure 3. Users should also run these export settings in the terminal or append them to the end of vim ~/.bashrc.

These commands must be run:

```
export DISPLAY=$(awk '/nameserver / {print $2; exit}' /etc/resolv.conf 2>/dev/null):0
export LIBGL_ALWAYS_INDIRECT=0
```


Extra settings

Extra settings

- [✓] Clipboard
  Start the integrated clipboard manager
  - [✓] Primary Selection
    Also map the PRIMARY selection to the windows clipboard.
- [ ] Native opengl
  Use the native windows opengl library (wgl). Make sure to export the LIBGL_ALWAYS_INDIRECT environment variable.
- [✓] Disable access control
  Use this when you want vcxsrv to accept connections from all clients.

Additional parameters for VcXsrv

Figure 3, X11 server settings

X11 server must be running before "./MyGame" is executed, otherwise the window won't appear. X11 server can be, for example, a VcXsrv found from this link. Users should make sure that only one instance of X11 server is alive. If the make complains about some files missing, for example "vorbis", then the users can get them by writing "sudo apt-get libvorbis-dev". These also need to be the developer branch, hence the suffix -dev.

How to play

After the game is running, the users can click on the "Start round" button in order to spawn the enemies. By default, the round starts at round 1. If the user wishes to build towers, they should first click on the grass (green tiles) and then click on any of the three towers to build. The building will succeed as long as the user has enough money to build one, there is no tower already on the tile, and the tile is not a path tile. Different towers have different fire rates and range. For example, the rapid-fire tower has very low range but, as the name implies, high rate of fire. If the monsters reach the end of the path, the player life will be reduced by an amount dictated by the type of monster that reached the end. Once hp reaches 0, the player is greeted with a game over screen.

Once all the monsters are dead, the player can click on start round again to start the next round. Building towers can be done at any point of the game. Killing monsters grants the player money in order to build more towers.

Testing

The testing was conducted when the classes were built. For example, for the tower classes, it was mostly enough that something got drawn into the correct tile, and after construction it would print out its fields properly or the size of the vector was checked. On the pathfinding it was enough to have the monster created and it should then follow a certain path with the correct speed. We did not deploy the implementation until the class was tested in a way that it worked.

The idea is to have procedural development style. First the window, then something on the window, then some dynamic elements, such as the buttons, monsters, or towers in the window. This would allow us to check that classes would both render and update correctly. Automated unit tests were dropped due to time constraints, we expected it to be faster to do continuous integration over halting the development for making unit tests for each class that we did not know whether they would be used or needed.

Work log and meeting notes

The first two weeks were spent on getting to know the team and researching the SFML itself. The idea was to get a window ready and the window to draw something (could be anything) during the first two weeks. We would check in with each other every week to see what was done and what was left on the to-do list. After the first week, one of the group members wrote the basic framework of the game which was mostly followed. The team also noticed that one of the group members was dropped on the very first week, while the second group member was dropped in December.

The first weeks were also spent on debugging the SFML and making X11 work. One group member did the background while the other modified the whole code into object-oriented style. These were then merged into one. The next weeks were spent on doing basic window functions, such as, buttons and events while merging the code. Some vectors were also changed to hold unique pointers over the base class itself.

The last weeks were full on sprint weeks, the group did roughly 6-10 hours of coding per day on the last week. The order of the process was to get the window ready, then window buttons ready, then monsters and towers and finally the interface between the monsters and the towers in order to shoot and receive damage.

Work responsibilities

As the group size was rather small, there was no direct responsibility zones, both group members had to continuously integrate their own code into the master branch. Ashref was the one doing pathfinding, background, monsters, code commenting, and the bullet class while Po did the buttons, UI, game class main loops, the documentation, and the towers. Ultimately both team members had to make modifications in each other's codes so the line between who did what becomes rather blurry.

Weekly workload

Week 1&2

- Research and get to know each other
- Successfully making cmake work with sfml
- X11 server debugging
- Total: 15h

Week 3&4

- Background
- Window
- OOP style
- Total: 8h

Week 5&6

- Sprint week
- Derived classes
- Allowing towers to shoot monsters
- Bug fixes
- Total: 50h

Source

Haller, J., Hansson, H., V., Moreira, A., (2013), *SFML Game Development, 1st edition*, Packt Publishing, [Online], Available at: https://ebookcentral-proquest-com.libproxy.aalto.fi/lib/aalto-ebooks/detail.action?pq-origsite=primo&docID=1214972, (Accessed 11.12.2022)