PHYS30762 Object-Oriented Programming in C++
Final Project

William Cooper

May 27, 2022

Abstract

This project consists of the development of the board game monopoly. The objectives of this project were to create a detailed structure of a board game with sufficient complexity to show advanced C++ object-oriented features. This project turned out well with the game functioning as expected and provides a monopoly-like experience. If time permitted a few more features could have been added such as adding 3 or 4 player options.

Introduction

Monopoly is a good candidate as it is complex with various unique properties and players which give the freedom to implement advanced C++ object-oriented programming such as inheritance. The method to create this game was to start with the fundamental 'game engine' that will update the game and present it to the user, then slowly build upon this with more classes and functions, with the game becoming more advanced in terms of functionality. This project also allowed the exploration and use of external libraries which was taken advantage of to create a visually pleasing, friendly user interface.

In this project I aimed to create a game that plays like monopoly. This includes a way to interact with the game, the ability to roll a dice and move positions on the board, the option to buy property and add houses to them as well as consequences on landing on owned properties or chest/chance cards.

Code Design and Implementation

For a game project a Graphical User Interface (GUI) felt appropriate to include for ease of use and so the game could be presented nicely. After research into various GUI's the Simple and Fast Multimedia Library (SFML) seemed to be the best option for this project [1]. Details on how to download this external library in Linux is included in the read me file. This library allows a window pop up with images, pictures, shapes to be displayed. Using this research, I found a way the user interacts with the game, the functionality of the game could then be built upon this to reach my other objectives. The rules are also in the read me file.

The board image used is an official design [2] and published by the Hasbro Gaming.

Classes have been split across header and cpp files for readability and structure. Pragma once is used in each header file so that the source file is only included once in a single compilation. A make file was made to run all the header cpp files and run the SFML application with a single command. The make file first compiles all the cpp files in the folder, then links the compiled file to the SFML libraries to get the final executable o files. Then executes the compiled program.

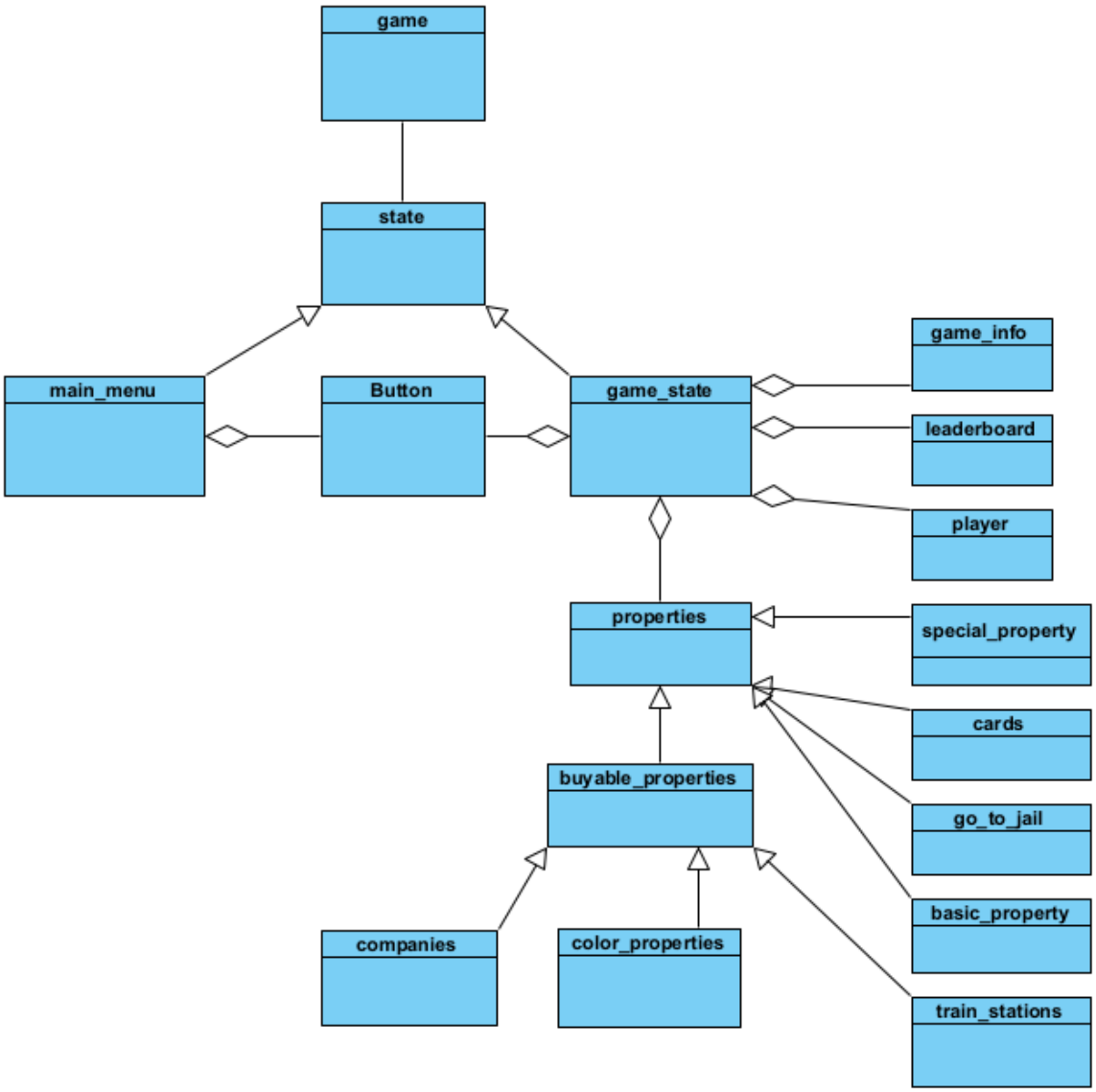

Figure 1. Class diagram for the monopoly game. Arrows line represent generalization, showing the inherited classes. Diamond end lines represent aggregation, a subset of association which is a 'has a' relationship. For example, the main menu class can exist without the class Button.

A game class was built to create a game engine, where it checks for updates in the game and what to render. A window is created to render and initialised using class functions, here the size, frame rate and more can be chosen. The size of the window was chosen so it would fit in a 1080p resolution screen. SFML allows to get an event that happened in the window, such as a mouse click or keyboard key press and the ability to add functionality like closing the window can be added.

An abstract state class was created, which main menu and game state classes were derived from as seen in figure 1. The state class has virtual functions to update, render and check if the state is wanting to be quit by polling events and using switch cases.

```
while(this->window->pollEvent(this->ev))//while we get events from window, we save into event variable we created
{
    //switch allows a variable to be tested equally against a list of values, value = case
    switch(this->ev.type)//each event has a type, ev comes will all data from that event
    {
        case sf::Event::Closed: //when close button on window pressed it closes
            this->window->close();     //it sends an event with type: close, we have to manually close it
            break;
```

Figure 2. The switch allows a variable, in this case an event, to be equally tested against a list of values by using a case.

A stack was created in the game class to hold the main menu state and game state classes. This was done so that when a user advances past the main menu the state is popped off the stack and then the game state is at the top of the stack. This stack was used in a while loop to call an update and render function for the state at the top of the stack, so that the game was updated and rendered every loop as shown in figure 3. This is how the game engine works.

```
this->states.push(std::make_unique<game_state>(this->window));
this->states.push(std::make_unique<menu_state>(this->window));
if(!this->states.empty()){
    this->states.top()->update();
    if(this->states.top()->get_quit()){
        this->states.top()->end_state();
        this->states.pop();
    }
}
```

Figure 3. Code of states stack and how each stack is updated and deleted.

This method used takes the complexity away from the main program, which was only used to create a game object and call the run function which updates and renders the game in the class. The structure and use of classes are mostly in the derived state classes.

Both states make use of the button class, this is the primary way the user interacts with the games interface. This design feature makes it easy for the user to know they are doing as it is intuitive. A button is created by using SFML to make a rectangle shape with specified position, width and height with the ability to align text on top, then on the event a mouse is clicked the game can check the mouses pixel position in the window and is checked if it was in the global boundaries of the shape, shown in figure 4. If it was then functionality can be added after the click, this acts as a button.

```
void Button::update(const sf::Vector2f mousePos)
{
    this->clicked=0;
    if(this->shape.getGlobalBounds().contains(mousePos))
    {
        //pressed
        if(sf::Mouse::isButtonPressed(sf::Mouse::Left))
        {
            this->clicked=1;
        }
    }
}
```

Figure 4. Code showing how a function in the Button class takes the mouse position as a parameter and checks if it is in bounds of the button shape.

Button objects were created using a unique pointer for dynamic memory allocation. A unique pointer cannot be copied but can be moved. This works for a button because there is only need for one owner of the underlying pointer. The make_unique is used over the new operator as it is exception safe and with a unique pointer there is no need for delete operator as it is done automatically, figure 3.

The game state class contains the bulk of the code and has a substantial use of other classes to create the actual game and functionality of monopoly. The dice roll was created by obtaining a random number from hardware then a Mersenne Twister pseudo-random generator of a 32-bit number whilst defining the range of a single dice.

A player class was created to hold the information of each player in the game, such as money and position on the board. The class also renders the players piece on the board which is a coloured circle. A vector of pointers to the player class was initialised in the game glass to hold each player in the game. An integer iterator was used to cycle through this vector, to simulate different players turns.

Polymorphism is used to create more consistent code, instead of creating new functions for the same thing in each derived class. The iterator is used as an index in the player vector to access the player functions on their turn. This iterator is increased when the end turn button is pressed. For two players the iterator cannot go above one, so a safety measure is introduced to cover this.

When creating a vector of base class pointers, shared pointer is used as the raw pointer needs to be assigned to multiple owners and it will not go out of scope until all owners have. This is ideal especially when players/properties must be updated and rendered. Make_shared operator is used in this case instead of new for the same reasons as make_unique stated above and we are using a shared pointer.

To display the score, or each player budget a leaderboard class was created, this is updated by a function that takes the players number and their money count.

A game information class was created to keep the user up to date in the game, such as what their dice roll was. A string vector represented the number of lines used to display the game information and was used to push back each new update in the game. The first index of the vector, or line, always contained which players turn it was. The other lines were updated from line 2 downwards, if something happened it would shift the line 2 to line 3 and so on, then add the new update to line 2. This created an almost news feed type of box to display new game updates.

An abstract base class called properties was created to start the building blocks for every other property, it has few variables and is mainly used to create the name. It also contains a virtual landed on function which is fundamental to update the game when a player lands on a property. As seen in figure 1 a buyable property class is inherited from this. This class adds further functionality such as allocating owners to the object when bought and gets the rent if landed on. When a buyable property is bought a marker is put on the property to show who owns it. Further inheritance is used to create further detailed buyable properties, such as train stations and colour properties. A text file is read in to parametrize the price, name and rent of each buyable property.

Each buyable property also has a transparent button over the top of it, this is so when the option for houses is true for a group of properties, by clicking on the property it puts a house on it and increases the rent.

Other properties are inherited from the base properties such as card properties. These cards read in a file with the commands and money/position to move to. The decision was made to make chance just advance, and chest cards are just reward/pay for simplicity.

When calling functions from an inherited class a dynamic cast must be used to handle this polymorphism. It casts the pointer down the inherited class structure.

```
dynamic_cast<color_properties *>(properties_vec[i].get())->house_bought()==true
```

Figure 5. Example of a dynamic cast used to get check if a house has been bought on a property.

Results

The main menu is as expected, it is quite basic, but it has all the required functionality for a menu. By pressing 'play game' it moves onto the next state. However, the user must select the number of players first as the game needs to know how many players to create and render. This functionality allows 3 and 4 player options to be added in the future. This is the state inherited from the state class and has an association with the button class. Figure 6.

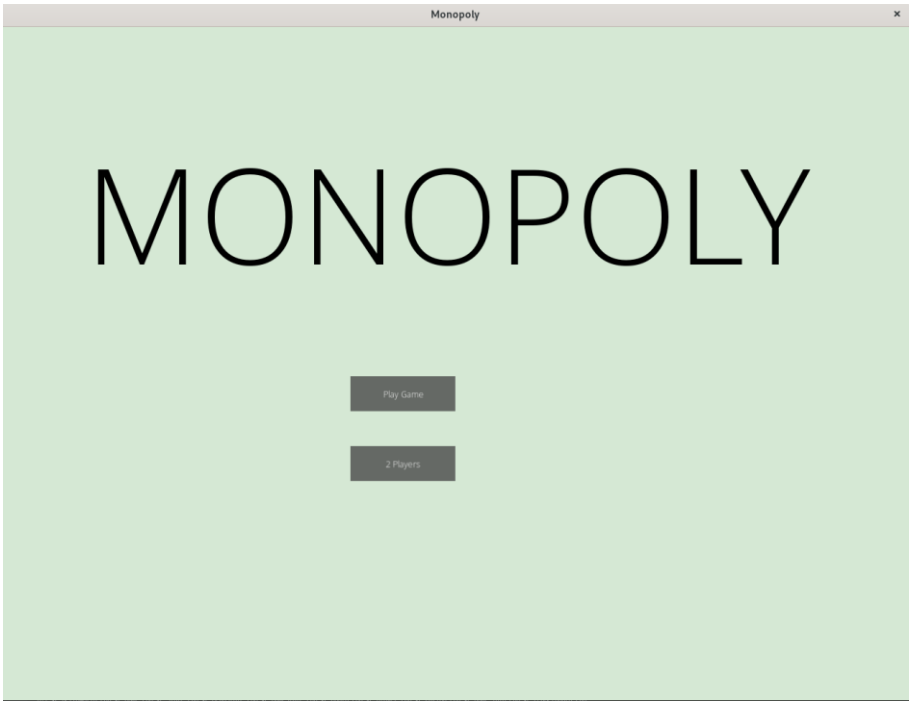

Figure 6. Main menu state.

The design out the game state is as wanted, figure 7. The board is rendered using an image from the official monopoly game [2]. The right-hand side consists of an information and interactive part of the game, where the text and buttons are presented. This game state is inherited from the abstract state class and is associated with 5 more classes which are button, properties, player, game info and leaderboard.

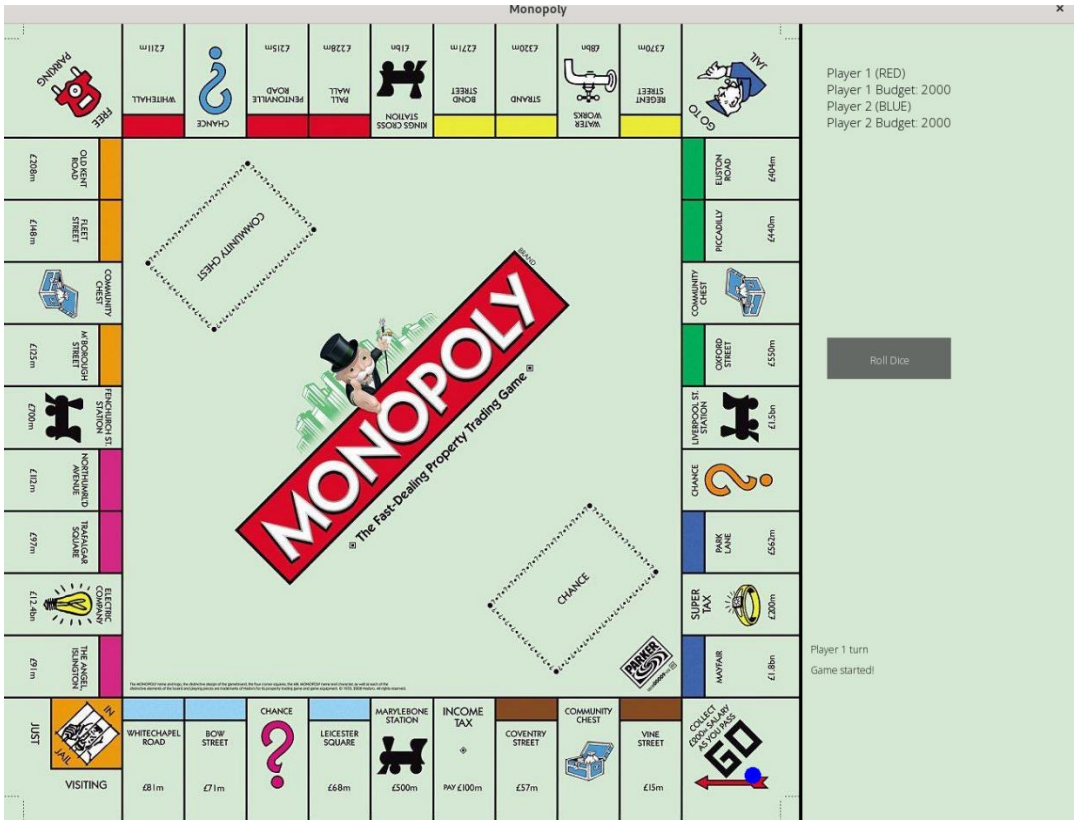

Figure 7. Game state, the board and the user interface can be seen. The score board is top right and game information is displayed bottom right.

The appropriate buttons show only when the user can appropriately use them, such as the buy button is only presented and clickable if they land on a buy-able property.

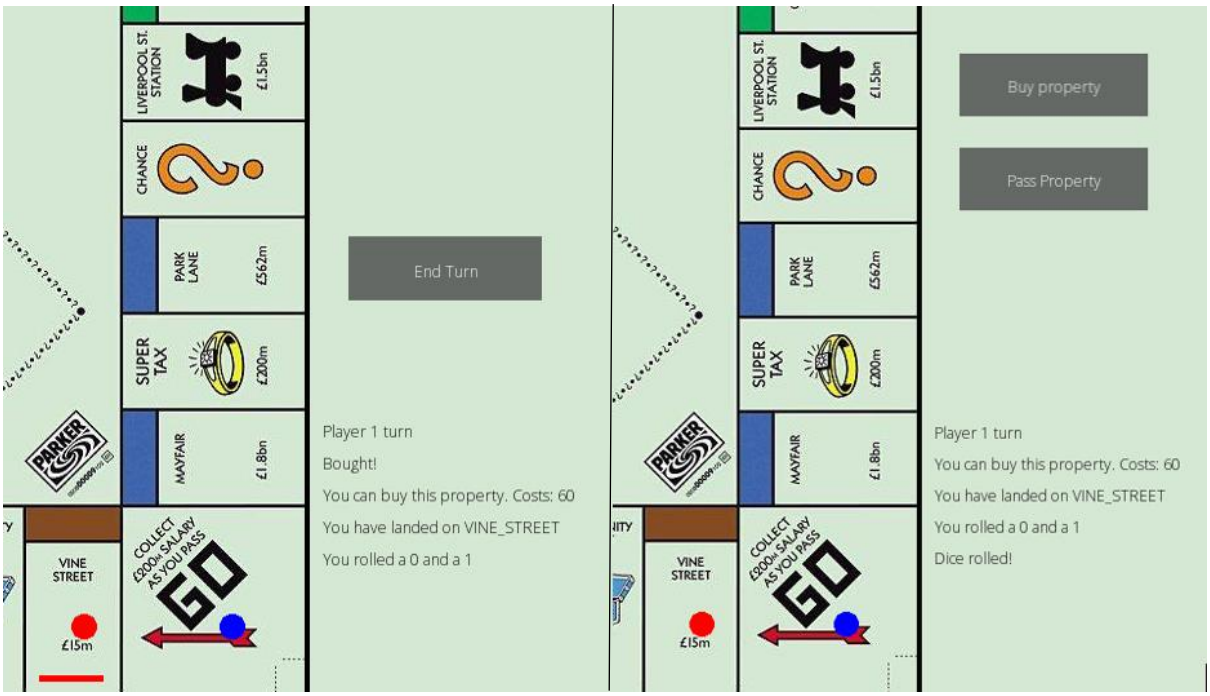

Figure 8. Example of a player landing on a property and getting the option to buy or pass (right) and the game after they bought the property (left). The game marks the property with the players colour to keep track of every player property. The end turn button is also displayed here after the user has made their choice.

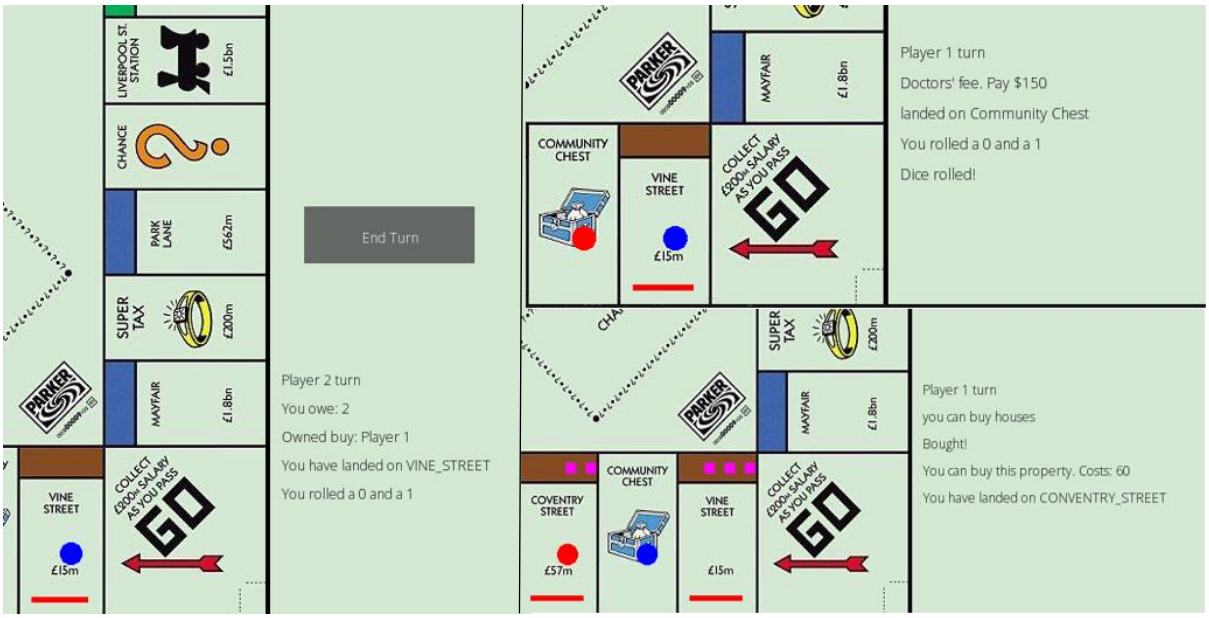

Figure 9. The left figure shows the outcome of a player landing on a property that is already owned. Bottom right displays the game telling the user that owns all of one colour property that they can buy houses, and they have as seen by the pink squares. Top right is a player landing on the community chest, the consequence is shown in the information box.

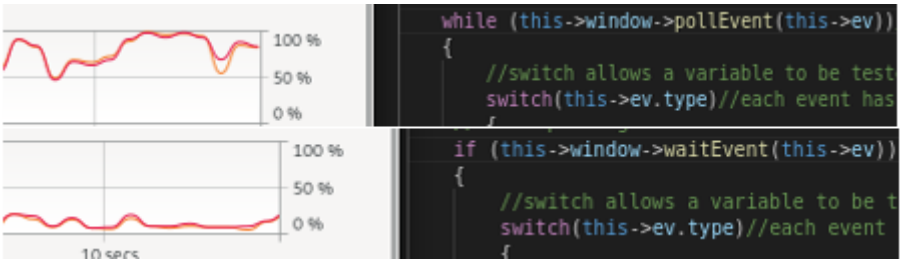

Figure 10. Since the game waits for input it is forever looping, by using waitEvent it fixes this. However, this caused not responding errors, this is a known bug.

The game functionality can be tested by accessing the game state cpp file and manipulating what the dice rolls, I will comment out what can be changed. For example, in these figures the dice rolled a 0 and 1 every time to test out ownership and houses.

Along with the functionality shown in figures 6-8; the tax property correctly deducts money from each player, property values including train stations rent increases when multiple of the same properties are owned, chance cards move the position of player to the correct property the cars says and they are presented with the offer to buy it or pay the owner, company properties rent is the value of the dice roll and increases by a factor of ten when both are owned, go to jail successfully sends player to jail and cannot get out until a double is rolled. The game ends when a player budget is below 0, the correct player is displayed as the winner and the game prompts the user to press escape to exit the game.

When the game ends the remaining player cannot press the end turn button, but it might have been nice to create a game over state to show the winner. The price of the properties on the board are not representative of the prices in the actual game, this is due to the image having incorrect prices. All colour properties have the same house cost, this could have been fixed by having it read through a text file like all other properties parameters.

When pressing a button in game, if the mouse button is held down, they it will register as multiple clicks. This would be investigated if time permitted.

Conclusion

Overall, I think the project turned out very well, the game plays like monopoly without out a couple of features but it doesn't take away from the gameplay. The game successfully takes advantage of advanced C++ features such as polymorphism, smart pointers and a graphical user interface.

If time permitted all monopoly features could be added, the game has a good structure, allowing features and further complexity to be added with relative ease. The option for 3 or 4 players is half embedded into the game code but could not be finished in time. Mortgages and trading are also not included; however, these features allow the game to be played a bit quicker as there is no 'second chance'.

References

[1] SFML, https://www.sfml-dev.org/

[2] monopoly board design, https://wallpaperaccess.com/monopoly-game