Spike: Task 8.P
Title: Game State Management

Author: Mitchell Wright, 100595153

Goals / deliverables:

Demonstrate the ability to collect and analyse software performance data, applying said knowledge to the comparison and evaluation of functions, IDE settings and compiler settings.

Items created during task:

- Code, see: \08 - Spike – Game State Management\zorkish\

Technologies, Tools, and Resources used:

- Visual Studio 2022
- SourceTree
- GitHub
- Lecture 2.2 – Game States & Stages

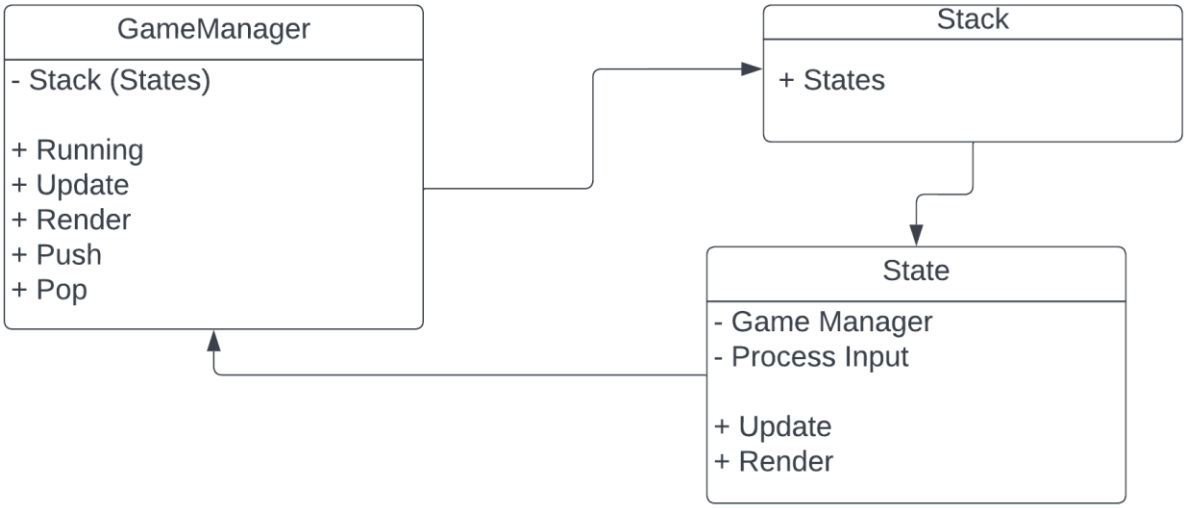
Tasks undertaken:

- Planning
- Developing the State Abstract Class
- Developing the Game Manager
- Developing the Basic States
- Developing the Gameplay
- Commit to Git

What we found out:

1. Planning:

The planning stage for this was pretty brief, as I'd already gotten most of the code figured out, due to the lecture notes providing a very strong starting point.



This was the basic plan I came up with in order to develop the game. Though please note that the Game Manager ended up just using current to access the update and render functions rather than having those functions itself.

2. Developing the State Abstract Class:

As per the plan above, most states will have the following functionality in common:
- User input.
- A pointer to the manager.
- Update.
- Render.

Therefore, having a parent class that contains all of these as virtual methods would be very useful.

```cpp
#pragma once
#include "GameManager.h"
#include <iostream>

using namespace std;

class GameManager;

class State
{
protected:
    GameManager* _manager = nullptr;
    char _command;

    virtual void processInput()
    {
        cin >> _command;
        cin.clear();
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    };

public:
    virtual void update() = 0;
    virtual void render() = 0;
};
```

Please note that there is a forward declaration of the Game Manager class as well. Without that, there were errors due to circular dependencies. Additionally, the user input is defined here as most state classes used a simple char as an input. Cin.ignore() is also used to clear the input stream, so that entering multiple characters doesn't result in multiple inputs being registered.

3. Developing the Game Manager:

I decided to go with the style of manager described in the lecture notes, using a stack in order to manage states. This means the manager is pretty simple, but still very usable and scalable. However, states could be improved in this sense as a result. It's also worth noting that due to a previous unit run by Marcus Lumpe, I'm making sure to use header files correctly and frequently. I still have nightmares about that class.

```
class GameManager
{
private:
    stack<State*> _states;
public:
    GameManager();
    ~GameManager();

    bool running();
    State* current();

    void push_state(State*);
    void pop_state();
};
```

The manager has some pretty basic functionality. It constructs itself with a MainMenu to start, it is able to check if it's running by asking the stack if it still has States in it and there's the push and pop interfaces. Additionally, there's a function that returns a pointer to the topmost element of the stack. Very simple stuff.

```
#include "GameManager.h"

GameManager::GameManager()
{
    push_state(new MainMenu(this));
}

GameManager::~GameManager()
{
    while (!_states.empty())
        pop_state();
}

bool GameManager::running()
{
    return !_states.empty();
}

State* GameManager::current()
{
    return _states.top();
}

void GameManager::push_state(State* state)
{
    _states.push(state);
}

void GameManager::pop_state()
{
    _states.pop();
}
```

4. Developing the Basic States:

All states, except Gameplay and Score are practically identical. The main changes are to the render function. So for this section, I'll quickly run through MainMenu as it's the most complicated of the lot.

```
MainMenu::MainMenu(GameManager* manager)
{
    _manager = manager;
}

MainMenu::~MainMenu()
{
    delete _manager;
    _manager = nullptr;
}
```

The constructor sets the manager so that pop and push are accessible when needed. The destructor does the exact opposite. Simple stuff there.

```
void MainMenu::update()
{
    processInput();

    switch (_command)
    {
    case '1':
        _manager->push_state(new LevelSelect(_manager));
        break;
    case '2':
        _manager->push_state(new HallOfFame(_manager));
        break;
    case '3':
        _manager->push_state(new Help(_manager));
        break;
    case '4':
        _manager->push_state(new About(_manager));
        break;
    case '5':
        _manager->push_state(new Quit(_manager));
        break;
    default:
        cout << "Invalid input." << endl;
    }
}
```

Update makes use of the process input function defined in the parent class, then uses a switch statement to determine the next state to push to the stack.

```
void MainMenu::render()
{
    cout
        << "============================" << endl
        << "Welcome to Zorkish Adventures" << endl
        << "============================" << endl
        << endl
        << "1. Select Adventure and Play" << endl
        << "2. Hall of Fame" << endl
        << "3. Help" << endl
        << "4. About" << endl
        << "5. Quit" << endl
        << "Select 1-5:" << endl
        << ">> ";
}
```

Render prints out a nice main menu, with the options presented to the player, and a sick lil chevron there, for the aesthetics. An improvement I'll likely make is having the title only appear in the MainMenu's construction.

5. Gameplay:

Realistically, the biggest difference between Gameplay and the other states, is the tweaks to input processing.

```
private:
    string _command;
```

That's the change. It's now a string.

```
void Gameplay::update()
{
    processInput();

    if (_command == "quit")
    {
        _manager->pop_state();
    }
    else if (_command == "hiscore")
    {
        _manager->pop_state();
        _manager->push_state(new Score(_manager));
    }
    else
        cout << "Invalid input." << endl;
}
```

This means that the update method needs to be changed as well, and since strings are being used now, switch is out the window. This is obviously temporary, since a better input processor will be needed once more complex commands are in use.

Score includes the same change to input type, though doesn't do anything with the input and instead just pops itself off so that game moves to the Hall of Fame.

```
void Score::update()
{
    processInput();

    _manager->pop_state();
    _manager->push_state(new HallOfFame(_manager));
}
```

6. Commit History: