UNIVERSITY OF SCIENCE (HCMUS)
VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
(VNU-HCM)

SOLO PROJECT

TOPIC

Meow Invaders

Teacher:	Đinh Bá Tiến
	Lê Khánh Duy
Teacher Assistants:	Hồ Tuấn Thanh
	Nguyễn Lê Hoàng Dũng

HCM, 10/12/2023

Contents

Abstract

The project was inspired by the renowned game "Chicken Invaders," leading to the creation of "Meow Invaders" using the C++ programming language and the SFML (Simple and Fast Multimedia Library) framework. This report chronicles the comprehensive utilization and application of Object-Oriented Programming (OOP), Object-Oriented Design (OOD), SOLID principles, and various Design Patterns in Meow Invaders.

1 Introduction

The following report delineates the journey and achievements in the Solo Project pursued by Luong Nguyen Khoa, a student engaged in the 22APCS2 program at the University of Science. Meow Invaders embodies a thrilling escapade where players navigate a spaceship to safeguard Earth from impending adversaries. Comprising infinite levels, each tier comprises a maximum of three distinct phases: Enemy, Disaster, and Boss phases. Additionally, the game offers three power-ups, enhancing the player's capabilities for an enriched gaming experience.

2 Project Architecture

The figure 1 is my overall project architecture drawing in UML.

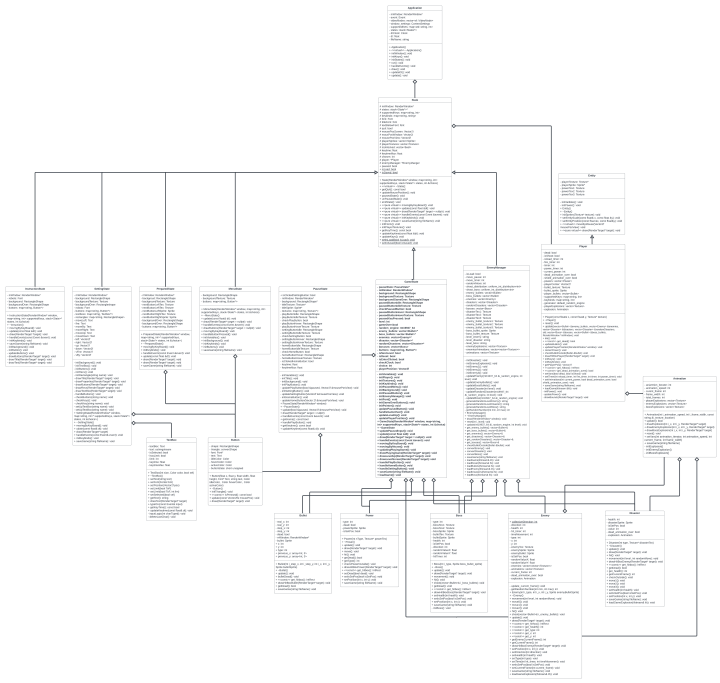

Figure 1: Class Diagram

The data storage section of my Meow Invaders project encompasses five primary directories:

1. **Assets:** This directory houses two subfolders:
   - **Images:** Stores various image assets used within the project.
   - **Fonts:** Contains font files utilized in the project's graphical elements.
2. **Build:** This directory materializes upon running the CMake file and contains the build-related output.

3. **External/SFML:** This folder serves as the repository for the SFML (Simple and Fast Multimedia Library) used within the project.
4. **Config:** Here, initialization files and the vital "save.txt" reside, responsible for serializing crucial project data.
5. **MEOW_INVADERS:** This pivotal folder holds the project's core codebase, structured into six distinct subfolders, complemented by the primary "main.cpp" file.
    - **Animations:** Manages the animation aspects for various entities within the project.
    - **Applications:** Governs the project's flow and overall execution.
    - **Entities:** Contains essential class representations for entities pivotal to the game, such as Player, Enemy, Disaster, Boss, Power, Bullet, EnemyManager,..
    - **Global Vars:** Houses universally accessible global variables utilized across multiple classes within the project.
    - **GUI:** Handles the creation and management of interactive elements like buttons and text boxes utilized throughout the project.
    - **States:** Holds and manages the different states of the project, encompassing various interfaces such as MenuStates, SettingState, GameState, InstructionState, ContinueState, among others.

3 Data Storage

In the **Config** folder, there are 4 files:

- gameState_keybinds: This file saves the buttons for the gameplay.
- save.txt: This file saves the gameState that will help the program load the game.
- supportedKeys.ini: This file saves the keys that SFML supported.
- window.ini: This file saves the window configuration.

4 Implementation Details

In this section, we will come to the implementation of the program in **MEOW_INVADERS** folder.

4.1 main.cpp

This is the core file that will run first in the program.

```
int main()
{
    try
    {
        Application application;
        application.run();
    }
    catch (std::exception& exception)
    {
        std::cout << "\nEXCEPTION: " << exception.what() << std::endl;
    }
}
```

4.2 Application

4.2.1 Class Application:

```
class Application
{
public:
    Application();
    virtual ~Application();

    // Initialization
    void initWindow();
    void initKeys();
    void initStates();

    // Functions
    void run();
    void handleEvents();
    void draw();

    //Update
    void updateDt();
    void update();

private:
    // Variables
    sf::RenderWindow *mWindow;
    sf::Event event;
    std::vector<sf::VideoMode> videoModes;
    sf::ContextSettings window_settings;

    std::map<std::string, int> supportedKeys;

    std::stack<State *> states;

    sf::Clock dtClock;
    float dt;

    std::string fileName;
};
```

This class will play a crucial role across all the states of the program, serving as a foundational state that makes the program run correctly.

Public field:

- Application(): This constructor will initialize all the variables.
- virtual ~ Application(): This is the destructor that will de-allocate all the variables that will make sure there are no memory leaks.
- void initWindow(): This function initializes the window variables.
- void initKeys(): This function initializes the key binds variables.
- void initStates(): This function initializes the states.
- void run(): This function is the core that will run from the main file.

- void handleEvents(): This function will control the handle events functions of all states.
- void draw(): This function will control the draw functions of all states.
- void updateDt(): This function will update the dt variable.
- void update(): This function will control the update functions of all states.

Private field:

- sf::RenderWindow *mWindow: is the variable that helps the program run the SFML library.
- sf::Event event: is the variable that helps the program run the SFML library.
- std::vector<sf::VideoMode> videoModes is the variable that helps the program run the SFML library.
- sf::ContextSettings window_settings: is the variable that helps the program run the SFML library.
- std::map<std::string, int> supportedKeys: This will store all the supported keys that help to play the game.
- std::stack<State *> states: This will store all the states that will push the target state.
- sf::Clock dtClock: This will help the many times when click the buttons which lie on each other.
- float dt: This will help the many times when click the buttons which lie on each other.
- std::string fileName: This is the path of file that will store all the variables and states.

4.3 Animations

```
class Animation
{
public:
    Animation(int i_animation_speed, int i_frame_width, int total_frame);

    bool update();

    void drawExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::
    Texture texture);
    void drawEnemyExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::
    Texture texture, int type);
    void drawBossExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::
    Texture texture);
    void drawDisasterExplosion(int i_x, int i_y, sf::RenderTarget *target,
    sf::Texture texture);

    void reset();

    // setter
    void setVars(int animation_iterator,
                 int animation_speed,
                 int current_frame,
                 int frame_width);

    int getCurrentFrame();

    // save
    void saveGame(std::string fileName);
```

```
25
26 private:
27     int animation_iterator;
28     int animation_speed;
29     int current_frame;
30     int frame_width;
31
32     int total_frames;
33
34     sf::Sprite sprite;
35 };
```

4.3.1 Class Animation:

This class will help improve the smoothness of Entity animation and create a standardized format so that it can be reused when needed, without having to rebuild it from scratch.

Public field:

- Animation(int i_animation_speed, int i_frame_width, int total_frame): This is the constructor which will initialize the variables of the class.
- bool update(): This is the function check whether the update function ends or not.
- void drawExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::Texture texture): This function will draw the player explosion's animation.
- void drawEnemyExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::Texture texture, int type): This function will draw the enemy's animation.
- void drawBossExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::Texture texture): This function will draw the boss's animation.
- void drawDisasterExplosion(int i_x, int i_y, sf::RenderTarget *target, sf::Texture texture): This function will draw the disaster's animation.
- void reset(): This function will reset all the variables of the class.
- void setVars(int animation_iterator, int animation_speed, int current_frame, int frame_width): This function is the setter which will set the target's variables.
- int getCurrentFrame(): This function gets the current frame of the picture.
- void saveGame(std::string fileName): This function will write all the variables and states at that time to the file "save.txt".

Private field:

- int animation_iterator: This variable is the pointer to check whether it meets the target or not.
- int animation_speed: This is the animation speed of the class.
- int current_frame: This is the current frame variable.
- int frame_width: This is the frame width of the picture.
- int total_frames: This is the total frame variable.
- sf::Sprite sprite: This will draw the sprite.

4.4 GlobalVars

```
constexpr int SCREEN_WIDTH = 1300;
constexpr int SCREEN_HEIGHT = 800;

constexpr int BASE_SIZE = 8;
constexpr int FAST_RELOAD_DURATION = 10;
constexpr int RELOAD_DURATION = 35;
constexpr int PLAYER_BULLET_SPEED = 5;
constexpr int FIRE_TIMER = 20;

// Enemy
constexpr int ENEMY_HIT_TIMER_DURATION = 20;
constexpr int ENEMY_MOVE_SPEED = 25;
constexpr int ENEMY_TYPES = 3;
constexpr int ENEMY_SHOOT_CHANCE = 3000;
constexpr int ENEMY_SHOOT_CHANCE_INCREASE = 100;
constexpr int ENEMY_SHOOT_CHANCE_MIN = 1024;
constexpr int ENEMY_MOVE_PAUSE_MIN = 3;
constexpr int ENEMY_MOVE_PAUSE_DECREASE = 1;
constexpr int ENEMY_MOVE_PAUSE_START_MIN = 47;
constexpr int ENEMY_MOVE_PAUSE_START = 63;
constexpr int ENEMY_BULLET_SPEED = 2;
constexpr int ENEMY_ANIMATION_SPEED = 22;

constexpr int BOSS_HEALTH = 15;

// Power
constexpr int POWER_DURATION = 500;

// Draw the outline hit box to debug
constexpr bool debug = 0;

constexpr unsigned int EXPLOSION_ANIMATION_SPEED = 22;
constexpr unsigned int EXPLOSION_ENEMY_ANIMATION_SPEED = 22;
constexpr unsigned int EXPLOSION_BOSS_ANIMATION_SPEED = 15;
```

These variables play a crucial role across the entirety of the program, serving as foundational elements that dictate behavior, states, or values consistently utilized across various modules, functions, or components.

4.5 Entities

4.5.1 Class Entity

```
class Entity
{
public:
    Entity();
    virtual ~Entity();

    // Initialization
    void initSprites(sf::Texture *texture);

    // Functions
    void setEntityScale(const float &x, const float &y);
    void setEntityPosition(const float &x, const float &y);
    virtual void move(const float &x, const float &y);
```

```
14      virtual void moveByMouse(sf::Vector2f mousePosView);
15      virtual void draw(sf::RenderTarget *target) = 0;
16
17  protected:
18      // Init Variables
19      void initVariables();
20      void initPower();
21
22      // Variables
23      sf::Texture *playerTexture;
24      sf::Sprite *playerSprite;
25
26      // Fast fire
27      sf::Texture powerTex1;
28      // 3 bullets
29      sf::Texture powerTex2;
30      // shield
31      sf::Texture powerTex3;
32  };
```

This is the base class of the Entities folder which will be the interface for all other classes.

Public field:

- Entity(): This is the constructor that will initialize all the variables.
- virtual ∼ Entity(): This is the destructor that will de-allocate all the variables.
- void initSprites(sf::Texture *texture): This will initialize the needed sprite.
- void setEntityScale(const float &x, const float &y): This will set the scale of the entity.
- void setEntityPosition(const float &x, const float &y): This will set the position of the entity.
- virtual void move(const float &x, const float &y): This is a pure function that will make the entity move by keyboard.
- virtual void moveByMouse(sf::Vector2f mousePosView): This is a pure function that will control the entity moved by the mouse.
- virtual void draw(sf::RenderTarget *target) = 0: This is a pure function that will draw the entity.

Protected field:

- void initVariables(): This will initialize the variables.
- void initPower(): This will initialize the power of the player.
- sf::Texture *playerTexture: This is the texture for the player.
- sf::Sprite *playerSprite: This is the sprite for the player.
-
- sf::Texture powerTex1: This is the texture for power1.
- sf::Texture powerTex2: This is the texture for power2.
- sf::Texture powerTex3: This is the texture for power3.

4.5.2 Class Player

This is the derived class that inherit from the Entity class.

```
class Player : public Entity
{
public:
    Player(const float &x, const float &y, sf::Texture *texture);
    virtual ~Player();

    // Functions
    void reset();
    void update(std::vector<Bullet> &enemy_bullets,
                std::vector<Enemy> &enemies,
                std::vector<Disaster> &disasters,
                std::vector<Disaster> &randomDisasters,
                std::vector<Boss> &bosses,
                std::vector<Bullet> &boss_bullets,
                sf::RenderWindow *mWindow);
    void draw(sf::RenderTarget *target);
    void die();

    bool get_dead() const;

    void updateBullets();
    void updatePlayerPosition(sf::RenderWindow *mWindow);
    void restartPower();

    void checkBulletOutside(Bullet &bullet);
    void drawHitBoxPlayer(sf::RenderTarget *target);

    void initKeys();
    void initKeybinds();

    sf::Vector2f getPlayerPos();

    sf::IntRect get_hitbox() const;
    bool get_dead_animation_over() const;

    // setter
    void setVars(bool dead, bool isShoot);
    void setTimer(int reload_timer, int fire_timer, int timer, int
   power_timer);
    void setAnimation(int current_power, bool dead_animation_over, bool
   shield_animation_over);

    // save
    void saveGame(std::string fileName);

    // load
    void loadGame(std::ifstream &ifs);

private:
    void initPower();
    void initExplosion();

    void updatePower();

    void drawBounds(sf::RenderTarget *target);

    // Variables
```

```
56      bool dead;
57      bool isShoot = false;
58
59      int reload_timer;
60      int fire_timer;
61      int timer;
62      int power_timer;
63
64      int current_power;
65      bool dead_animation_over;
66      bool shield_animation_over;
67
68      std::vector<Power> powers;
69
70      sf::Vector2f playerCenter;
71
72      sf::Texture bullet_texture;
73      sf::Sprite bullet_sprite;
74      std::vector<Bullet> player_bullets;
75
76      std::map<std::string, int> supportedKeys;
77      std::map<std::string, int> keybinds;
78
79      std::default_random_engine generator;
80
81      std::vector<sf::Texture> playerExplosions; // explosions for player
82
83      Animation explosion;
84  };
```

Public field:

- Player(const float &x, const float &y, sf::Texture *texture): This is the constructor that will initialize all the variables.
- virtual ∼ Player(): This is a destructor that will de-allocate all the variables.
- void reset(): This is the function that will reset all the variables.
- void update(): This function will update all the time.
- void draw(sf::RenderTarget *target): This function will draw the player sprite.
- void die(): This function will make the player die.
- bool get_dead() const: This will get the dead variable.
- void updateBullets(): This will update the Bullet class.
- void updatePlayerPosition(sf::RenderWindow *mWindow): This will update the player position.
- void restartPower(): This will restart the Power class.
- void checkBulletOutside(Bullet &bullet): This function will check whether the Bullet is outside or not.
- void drawHitBoxPlayer(sf::RenderTarget *target): This will draw the hitbox of Player.
- void initKeys(): This will initialize the keys.

- void initKeybinds(): This will initialize the keys.
- sf::Vector2f getPlayerPos(): This will get the Player position.
- sf::IntRect get_hitbox() const: This will get the hitbox variable.
- bool get_dead_animation_over() const: This will get the "dead animation over" variable.
- void setVars(bool dead, bool isShoot): This will set the variables.
- void setTimer(int reload_timer, int fire_timer, int timer, int power_timer): This will set the timer.
- void setAnimation(int current_power, bool dead_animation_over, bool shield_animation_over): This will set the animation variable.
- void saveGame(std::string fileName): This will write all the variables to save.txt.
- void loadGame(std::ifstream &ifs): This will read the variables from save.txt.

Private field:

- void initPower(): This will initialize the Power class.
- void initExplosion(): This will initialize the explosion variable.
- void updatePower(): This will update Power class.
- void drawBounds(sf::RenderTarget *target): THis will draw the bounds of Power.

4.5.3 Class EnemyManager

```
class EnemyManager
{
public:
  EnemyManager();
  virtual ~EnemyManager();

  void draw(sf::RenderWindow *window);
  void reset(int i_level);
  void update(std::mt19937_64 &i_random_engine, int level);

  std::vector<Bullet> &get_enemy_bullets();
  std::vector<Bullet> &get_boss_bullets();
  std::vector<Enemy> &get_enemies();
  std::vector<Disaster> &get_disasters();
  std::vector<Disaster> &get_randomDisasters();
  std::vector<Boss> &get_bosses();

  void checkBulletOutside(Bullet &bullet);

  void convertEnemy();
  void convertDisaster();
  void convertBoss();

  // Save
  void saveGame(std::string fileName);

  // Load
  void loadGame(std::ifstream &ifs);
  void loadReset(std::ifstream &ifs);
```

```
30    void loadBullets(std::ifstream &ifs);
31    void loadEnemy(std::ifstream &ifs);
32    void loadDisasters(std::ifstream &ifs);
33    void loadBoss(std::ifstream &ifs);
34
35  private:
36    // Init
37    void initDisaster();
38    void initEnemyExplosion();
39    void initDisasterExplosion();
40    void initBossExplosion();
41    void initEnemy1();
42    void initEnemy2();
43    void initEnemy3();
44
45    void updateEnemy(std::mt19937_64 &i_random_engine, int level);
46    void updateEnemyBullets();
47    void updateBossBullets();
48    void updateDisaster(int level);
49    void updateRandomDisaster(std::mt19937_64 &i_random_engine, int level);
50    void updateBoss(std::mt19937_64 &i_random_engine);
51
52    std::string generateRandomLevelEnemy();
53    std::string generateRandomLevelDisaster();
54    std::string generateRandomLevelBoss();
55
56    int getRandomNumber(int min, int max);
57
58    bool isLoad;
59
60    int move_pause;
61    int move_timer;
62    int randomMove;
63
64    // To use the randomness from the <random> library, we need to define the
       distribution.
65    std::uniform_int_distribution<int> shoot_distribution;
66    std::uniform_int_distribution<int> shoot_boss;
67
68    std::vector<Bullet> enemy_bullets;
69    std::vector<Bullet> boss_bullets;
70    std::vector<Enemy> enemies;
71    std::vector<Disaster> disasters;
72    std::vector<Disaster> randomDisasters;
73    std::vector<Boss> bosses;
74
75    sf::Texture disasterTex1;
76    sf::Texture disasterTex2;
77    sf::Texture disasterTex3;
78
79    sf::Sprite enemy_bullet_sprite1;
80    sf::Sprite enemy_bullet_sprite2;
81    sf::Sprite enemy_bullet_sprite3;
82
83    sf::Texture enemy_bullet_texture1;
84    sf::Texture enemy_bullet_texture2;
85    sf::Texture enemy_bullet_texture3;
86
87    sf::Sprite boss_bullet_sprite;
88    sf::Texture boss_bullet_texture;
```

```
89
90    std::string level_enemy = "";
91    std::string level_disaster = "";
92    std::string level_boss = "";
93
94    std::vector<sf::Texture> enemyExplosions; // explosion for enemy
95    std::vector<sf::Texture> disasterExplosions; // explosion for disaster
96    std::vector<sf::Texture> bossExplosions; // explosion for boss
97
98    std::vector<std::vector<sf::Texture>> enemiesAnimations;
99    std::vector<sf::Texture> animations;
100 };
```

Public field:

- EnemyManager(): This is the constructor that will initialize the variables.
- virtual ∼ EnemyManager(): This is the destructor that will de-allocate the variables.
- void draw(sf::RenderWindow *window): This is the draw function that will draw the target entity.
- void reset(int i_level): This will reset all the variables and update to the next level.
- void update(std::mt19937_64 &i_random_engine, int level): This will update the class.
- std::vector<Bullet> &get_enemy_bullets(): This will get the enemy bullets variable.
- std::vector<Bullet> &get_boss_bullets(): This will get the boss bullets variable.
- std::vector<Enemy> &get_enemies(): This will get the enemies variable.
- std::vector<Disaster> &get_disasters(): This will get the disaster variable.
- std::vector<Disaster> &get_randomDisasters(): This will get the random Disaster.
- std::vector<Boss> &get_bosses(): This will get the bosses.
- void checkBulletOutside(Bullet &bullet): This will check whether the bullet is outside or not.
- void convertEnemy(): This will convert the character to the enemies.
- void convertDisaster(): This will convert the character to the disaster.
- void convertBoss(): This will convert the character to boss class.
- void saveGame(std::string fileName): This will write all the variables to the save.txt.
- void loadGame(std::ifstream &ifs): This will read all the variables from the save.txt.
- void loadReset(std::ifstream &ifs): This reads the reset variable.
- void loadBullets(std::ifstream ifs): This reads the bullet variable.
- void loadEnemy(std::ifstream &ifs): This read the enemy variable.
- void loadDisasters(std::ifstream &ifs): This read the disaster variable.
- void loadBoss(std::ifstream &ifs): This read the boss variable.

Private field:

- void initDisaster(): This initializes the disaster.
- void initEnemyExplosion(): This initializes the enemy explosion.
- void initDisasterExplosion(): This initializes the disaster explosion.
- void initBossExplosion(): This initializes the boss explosion.
- void initEnemy1(): This initializes the enemy.
- void updateEnemy(std::mt19937_64 &i_random_engine, int level): This updates enemy.
- void updateEnemyBullets();
- void updateBossBullets();
- void updateDisaster(int level);
- void updateRandomDisaster(std::mt19937_64 &i_random_engine, int level);
- void updateBoss(std::mt19937_64 &i_random_engine);

4.5.4 Class Boss

```
class Boss {
public:
    Boss(int i_type, sf::Sprite boss_bullet_sprite);
    ~Boss();

    void update();
    void draw(sf::RenderTarget* target, sf::Texture texture);

    void movement();
    void hit();
    void shoot(std::vector<Bullet>& i_boss_bullets);

    bool getDead();
    int get_health();
    int getCurrentFrame();

    sf::IntRect get_hitbox() const;
    void drawHitBoxBoss(sf::RenderTarget* target);

    // setter
    void setHealth(int health);
    void setIsSetPos(bool isSetPos);
    void setPosition(int x, int y);

    // save
    void saveGame(std::string fileName);
    // load game
    void loadGameExplosion(std::ifstream& ifs);
private:
    // Init
    void initBoss();

    int type;
    int bossTimer;

    sf::Texture bossTex1;
```

```
    sf::Texture bossTex2;

    sf::Sprite bossSprite;

    sf::Texture bulletTex;
    sf::Sprite bulletSprite;

    int health;

    bool isSetPos;

    int direction;
    float randomValueX = 2.0f;
    float randomValueY = 3.0f;
    Animation explosion;
    bool dead_animation_over;
};
```

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init function initializes the variables.

4.5.5 Class Bulllet

```
class Bullet
{
public:
    Bullet(int i_step_x, int i_step_y, int i_x, int i_y, sf::Sprite
   bulletSprite);
    virtual ~Bullet();

    // Functions
    void update();
    void bulletDead();
    sf::IntRect get_hitbox() const;
    void drawHitBoxBullet(sf::RenderTarget *target);
    bool getDead();

    void saveGame(std::string fileName);

    int x;
    int y;

    int type;

```

```
    std::array<int, 3> previous_x;
    std::array<int, 3> previous_y;

private:
    int real_x;
    int real_y;
    int step_x;
    int step_y;

    bool dead;

    sf::RenderWindow *mWindow;
    sf::Sprite bullet;
};
```

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

4.5.6 Class Disaster

```
class Disaster
{
public:
    Disaster(int type, sf::Texture* disasterTex);
    ~Disaster();

    void update();
    void draw(sf::RenderTarget* target, sf::Texture texture);
    void hit();
    void movement(int level, int randomMove);
    void drawHitBoxEnemy(sf::RenderTarget* target);
    sf::IntRect get_hitbox() const;

    bool getDead();
    int get_health();
    int getCurrentFrame();
    void checkOutside();

    void move1();
    void move2();
    void move3();

    // setter
    void setHealth(int health);
    void setIsSetPos(bool isSetPos);
    void setPosition(int x, int y);

    // save game
```

```
    void saveGame(std::string fileName);
    // load game
    void loadGameExplosion(std::ifstream& ifs);
private:
    int health;
    sf::Sprite disasterSprite;
    bool isSetPos;
    int value;
    int type;
    Animation explosion;
    bool dead_animation_over;
};
```

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field: This store the texture and sprite for drawing disaster.

4.5.7 Class Enemy

```
class Enemy
{
public:
    Enemy(int i_type, int i_x, int i_y, sf::Sprite enemyBulletSprite);
    virtual ~Enemy();

    // Movements
    void movement(int level, int randomMove);
    void move0();
    void move1();
    void move2();

    // Functions
    void hit();
    void shoot(std::vector<Bullet> &i_enemy_bullets);
    void update();
    void draw(sf::RenderTarget *target, sf::Texture texture, sf::Texture
    enemyTex);

    // Access
    sf::IntRect get_hitbox() const;
    int get_health() const;
    int get_hit_timer() const;
    int get_type() const;
    int get_x() const;
    int get_y() const;
    bool get_dead() const;
    int getCurrentFrame();
```

```
28      int getEnemyCurrentFrame();
29      void drawHitBoxEnemy(sf::RenderTarget *target);
30
31      static int collectiveDirection;
32
33      // setter
34      void setPosition(int x, int y);
35      void setDirection(int direction);
36      void setHealth(int health);
37      void setType(int type);
38      void setTime(int hit_timer, int timeMovement);
39      void setIsSetPos(bool isSetPos);
40      void setCurrentFrame(int current_frame);
41
42      // save
43      void saveGame(std::string fileName);
44      // load game
45      void loadGameExplosion(std::ifstream &ifs);
46
47  private:
48      // change Frame
49      void update_current_frame();
50      int getRandomNumber(int min, int max);
51
52      int direction;
53      int health;
54
55      int hit_timer;
56      int timeMovement;
57      bool dead_animation_over;
58
59      // type of enemy
60      int type;
61
62      int x;
63      int y;
64
65      sf::Sprite enemySprite;
66      sf::Sprite enemyBullet;
67
68      bool isSetPos;
69      float randomValueX = 1.0f;
70      float randomValueY = 2.0f;
71
72      int current_frame;
73      Animation explosion;
74  };
```

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.

- The load function reads the variables from save.txt.

Private field:

- void update_current_frame(): This update the current frame of the enemy.
- int getRandomNumber(int min, int max): This will return the random number.

4.5.8 Class Power

```
class Power
{
public:
    Power(int nType, sf::Texture *powerTex);
    ~Power();

    void update();
    void draw(sf::RenderTarget *target);
    void move();
    void hit();

    bool getDead();
    int getType();

    void checkPowerOutside();
    void drawHitBoxPower(sf::RenderTarget *target);
    sf::IntRect get_hitbox() const;

    // setter
    void setDead(bool dead);
    void setIsSetPos(bool isSetPos);
    void setPosition(int x, int y);

    // save
    void saveGame(std::string fileName);

private:
    // Power Up
    // 1 - Fast fire
    // 2 - 3 bullets
    // 3 - shield
    int type;
    bool dead;
    sf::Sprite powerSprite;
    bool isSetPos;
};
```

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

4.6 States

This folder holds all the states in the Program such as GameState, PreparedState, MenuState,...

4.6.1 Class State

```
class State
{
public:
    State(sf::RenderWindow *window, std::map<std::string, int> *
    supportedKeys, std::stack<State *> *states, int &choice);
    virtual ~State();

    // Functions
    const bool getQuit() const;
    void updateMousePosition();
    void pausedState();
    void unPausedState();
    void endState();

    // Pure Functions
    virtual void movingByKeyBoard() = 0;
    virtual void update(const float &dt) = 0;
    virtual void draw(sf::RenderTarget *target = nullptr) = 0;
    virtual void handleEvents(const sf::Event &event) = 0;
    virtual void initKeybinds() = 0;
    virtual void saveGame(std::string fileName) = 0;

    // Initialization
    void initFonts();
    void initPlayerTextures();

    const bool getKeytime();
    void updateKeytime(const float &dt);

    // Update keybinds
    void updateKeys();

    bool paused;

    static void setIsLoad(bool isLoad);
    static void setIsSaved(bool isSaved);

    static bool isLoad;
    static bool isSaved;

protected:
    sf::RenderWindow *mWindow;
    std::stack<State *> *states;
    std::map<std::string, int> *supportedKeys;
    std::map<std::string, int> keybinds;
    std::map<std::string, std::string> keys;

    sf::Font font;
    sf::Font titleFont;
    sf::Font textBelowFont;

    bool quit;

    // Mouse Positions
```

```
    sf::Vector2i mousePosScreen;
    sf::Vector2i mousePosWindow;
    sf::Vector2f mousePosView;

    // Resources
    std::vector<sf::Sprite> playerSprites;
    std::vector<sf::Texture> playerTextures;

    Player *player;
    EnemyManager *enemyManager;
    int chosen;

    float keytime;
    float keytimeMax;
};
```

Public State:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.
- The pure function create the interface for all the derived class.

Protected field:

The variables to used for all the derived class.

4.6.2 Class SettingState

```
class SettingState : public State
{
public:
    SettingState(sf::RenderWindow *window, std::map<std::string, int> *
   supportedKeys, std::stack<State *> *states, int &choice);
    ~SettingState();

    void movingByKeyBoard() {}
    void update(const float &dt);
    void draw(sf::RenderTarget *target);
    void handleEvents(const sf::Event &event);
    void initKeybinds();

    void saveGame(std::string fileName) {}

private:
    // Init
    void initBackground();
    void initTextBox();
    void initButtons();
```

```
20      void initRecs();
21      void initRectangle(std::string name);
22
23      // Draw
24      void drawTitle(sf::RenderTarget *target);
25      void drawProperties(sf::RenderTarget *target);
26      void drawButtons(sf::RenderTarget *target);
27      void drawRecs(sf::RenderTarget *target);
28      void drawTextBox(sf::RenderTarget *target);
29
30      void handleButtons();
31      void checkButtons(std::string name);
32      void checkKeys();
33      void checkKey(std::string name);
34      // bool invalidKeys();
35
36      void setUpTextBox(std::string name);
37
38      sf::RenderWindow *mWindow;
39
40      // Background
41      sf::RectangleShape background;
42      sf::Texture backgroundTexture;
43      sf::RectangleShape backgroundOver;
44
45      sf::Font roboto;
46
47      std::map<std::string, Button *> buttons;
48      std::map<std::string, TextBox *> textboxs;
49      std::map<std::string, sf::RectangleShape *> rectangles;
50
51      sf::Text moveLeft;
52      sf::Text fire;
53      sf::Text moveBy;
54      sf::Text moveRight;
55      sf::Text moveUp;
56      sf::Text moveDown;
57
58      sf::Vector2f left, right, up, down, vFire, vBy;
59  };
```

This is the SettingState that used for Setting Menu.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.

- The draw functions draw all the sprites.
- The logic functions to check the other variables.

4.6.3 Class PreparedState

```
class PreparedState : public State
{
public:
    PreparedState(sf::RenderWindow *window, std::map<std::string, int> *
    supportedKeys, std::stack<State *> *states, int &choice);
    ~PreparedState();

    // Functions
    void movingByKeyBoard() {}
    void initKeybinds() {}

    void handleEvents(const sf::Event &event);
    void update(const float &dt);
    void draw(sf::RenderTarget *target);

    void saveGame(std::string fileName) {}

private:
    // Initialization
    void initBackground();
    void initNextButtons();
    void initButtons();

    // Next Buttons
    void updateNextButtons();
    void drawNextButtons(sf::RenderTarget *target);

    void drawButtons(sf::RenderTarget *target);
    void handleButtons();

    // Draw Spaceship
    void drawSpaceship(sf::RenderTarget *target);

    // Draw preparation
    void drawTitle(sf::RenderTarget *target);

    sf::RenderWindow *mWindow;

    // Background
    sf::RectangleShape background;
    sf::Texture backgroundTexture;

    // left next button
    sf::Texture nextButtonLeftTex;
    sf::Sprite nextButtonLeftSprite;

    // right right button
    sf::Texture nextButtonRightTex;
    sf::Sprite nextButtonRightSprite;

    sf::RectangleShape backgroundOver;
    std::map<std::string, Button*> buttons;
};
```

This is the PreparedState that is used for choosing the player before playing game.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.
- The draw functions draw all the sprites.
- The logic functions to check the other variables.

4.6.4 Class PauseState

```
class PauseState
{
public:
    PauseState(sf::RenderWindow *window);
    virtual ~PauseState();

    // Functions
    void update(bool &paused, sf::Vector2f &mousePosView);
    void draw(sf::RenderTarget *target = nullptr);
    void handleEvents(const sf::Event &event);
    const bool getHome();
    void handlePlayButton();

    bool isClickedSettingButton;

    const bool getKeytime();
    void updateKeytime(const float &dt);

private:
    // Initialization
    void initVariables();
    void initTitle();
    void initBackground();

    // Play button
    void initPlayButton();
    void updatePlayButton(bool &paused, sf::Vector2f &mousePosView);

    // Setting button
    void initSettingButton();
    void updateSettingButton(sf::Vector2f &mousePosView);

    // Home button
    void initHomeButton();
```

```
35      void updateHomeButton(sf::Vector2f &mousePosView);
36
37      sf::RenderWindow *mWindow;
38      sf::RectangleShape *background;
39      sf::Texture *titleTexture;
40      sf::Sprite *titleSprite;
41
42      std::map<std::string, sf::Texture *> textures;
43
44      // Play button idle
45      sf::RectangleShape playButtonIdle;
46      sf::Texture playButtonIdleTexture;
47
48      // Play button hover
49      bool checkPlayButton;
50      sf::RectangleShape playButtonHover;
51      sf::Texture playButtonHoverTexture;
52
53      // Setting button idle
54      sf::RectangleShape settingButtonIdle;
55      sf::Texture settingButtonIdleTexture;
56
57      // Setting button hover
58      bool checkSettingButton;
59      sf::RectangleShape settingButtonHover;
60      sf::Texture settingButtonHoverTexture;
61
62      // Home button idle
63      sf::RectangleShape homeButtonIdle;
64      sf::Texture homeButtonIdleTexture;
65
66      // Home button hover
67      bool checkHomeButton;
68      sf::RectangleShape homeButtonHover;
69      sf::Texture homeButtonHoverTexture;
70
71      bool isClickedHomeButton;
72      float keytime;
73      float keytimeMax;
74  };
```

This is the PauseState that used for pause the game.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.

- The draw functions draw all the sprites.
- The logic functions to check the other variables.

4.6.5 Class MenuState

```
class MenuState : public State
{
public:
    MenuState(sf::RenderWindow *window, std::map<std::string, int> *
    supportedKeys, std::stack<State *> *states, int &choice);
    virtual ~MenuState();

    // Functions
    void update(const float &dt);
    void draw(sf::RenderTarget *target = nullptr);
    void handleEvents(const sf::Event &event);
    void drawButtons(sf::RenderTarget *target = nullptr);
    void movingByKeyBoard();
    void handleButtonPressed();

    // Initialization
    void initVariables();
    void initBackground();
    void initKeybinds();
    void initButtons();

    void saveGame(std::string fileName) {}

protected:
private:
    sf::RectangleShape background;
    sf::Texture backgroundTexture;
    std::map<std::string, Button *> buttons;
};
```

This is the MenuState which is used for create a menu for the program.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.
- The draw functions draw all the sprites.
- The logic functions to check the other variables.

4.6.6 Class InstructionState

```
class InstructionState : public State
{
public:
    InstructionState(sf::RenderWindow *window, std::map<std::string, int> *
    supportedKeys, std::stack<State *> *states, int &choice);
    ~InstructionState();

    void movingByKeyBoard() {}
    void update(const float &dt);
    void draw(sf::RenderTarget *target);
    void handleEvents(const sf::Event &event);
    void initKeybinds() {}
    void saveGame(std::string fileName) {}

private:
    // init
    void initBackground();
    void initButton();

    void updateButtons();

    void drawButtons(sf::RenderTarget *target);
    void drawTitle(sf::RenderTarget *target);
    void drawText(sf::RenderTarget *target);

    sf::RenderWindow *mWindow;

    sf::Font roboto;

    sf::RectangleShape background;
    sf::Texture backgroundTexture;
    sf::RectangleShape *backgroundOver;

    std::map<std::string, Button *> buttons;
};
```

This is the InstructionState which is used for writing instruction for the player.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.
- The draw functions draw all the sprites.
- The logic functions to check the other variables.

4.6.7 Class GameState

```
class GameState : public State
{
public:
    GameState(sf::RenderWindow *window, std::map<std::string, int> *
    supportedKeys, std::stack<State *> *states, int &choice);
    virtual ~GameState();

    // Functions
    void updatePausedInput();
    void update(const float &dt);
    void draw(sf::RenderTarget *target = nullptr);
    void handleEvents(const sf::Event &event);

    void movingByKeyBoard();
    void movingByMouse();

    // Playing game
    void updatingPlayingGame();
    void drawPlayingGame(sf::RenderTarget *target);
    void drawLevelUp(sf::RenderTarget *target);
    void drawLevelScreen(sf::RenderTarget *target);

    void handlePlayButton();
    void handleHomeButton();
    void handleSettingButton();

    void saveGame(std::string fileName);

    // Load game
    void loadGame();

private:
    // Initialization
    void initVariables();
    void initPlayer();
    void initTextures();
    void initKeybinds();
    void initPausedMenu();
    void initBackground();
    void initPausedButton();
    void initEnemyManager();
    void initFont();
    void initGameOverButtons();
    void initPointer();

    // Functions
    void updatePausedButton();
    void handleGameOver();
    void updateLevelUp();
    void updatePlayerPosition();

    PauseState *pauseState;
    sf::RenderWindow *mWindow;

    sf::RectangleShape background;
    sf::Texture backgroundTexture;

    // Game over background
    sf::RectangleShape backgroundGameOver;
```

```
59
60      // Paused button idle
61      sf::RectangleShape pausedButtonIdle;
62      sf::Texture pausedButtonIdleTexture;
63
64      // Paused button hover
65      bool checkPausedButton;
66      sf::RectangleShape pausedButtonHover;
67      sf::Texture pausedButtonHoverTexture;
68
69      bool pauseKeyPressed;
70
71      int level;
72      bool gameOver;
73
74      std::mt19937_64 random_engine;
75
76      std::vector<Bullet> *enemy_bullets;
77      std::vector<Bullet> *boss_bullets;
78      std::vector<Enemy> *enemies;
79      std::vector<Disaster> *disasters;
80      std::vector<Disaster> *randomDisasters;
81      std::vector<Boss> *bosses;
82
83      std::map<std::string, Button *> buttons;
84
85      bool isNextLevel;
86      bool isReset;
87      bool isEnterClicked;
88      bool checkClock;
89      int choice;
90
91      sf::Vector2f playerPosition;
92  };
```

This is the GameState class that is the core class in the application.

Public field:

- The constructor allocates the variables.
- The destructor de-allocate the variables.
- The setter function will set the target's variables.
- The getter function will get the target's variables.
- The save function writes the variables to save.txt.
- The load function reads the variables from save.txt.

Private field:

- The init functions initialize the variables.
- The draw functions draw all the sprites.
- The logic functions to check the other variables.

5 Technical Problems and Solutions

Problem:
The biggest difficulty I faced when completing this project was serializing data to a save.txt file. I'm quite puzzled about how to save it and then read it back, especially with such a large amount of data.

Solution:
Our approach is to break it down into smaller parts, applying a "divide and conquer" strategy. Essentially, I'll save the data in a predetermined order, and for each class, I'll create a saveGame function so it can save the game at that point, significantly reducing complexity. Applying a similar mindset for loadGame will help in handling the loading process.

6 Features Demonstration

This is the Video Link of the project. MEOW INVADERS

Features
This is the Menu of the project.

Figure 2: MenuState

This is the InstructionState of the project.

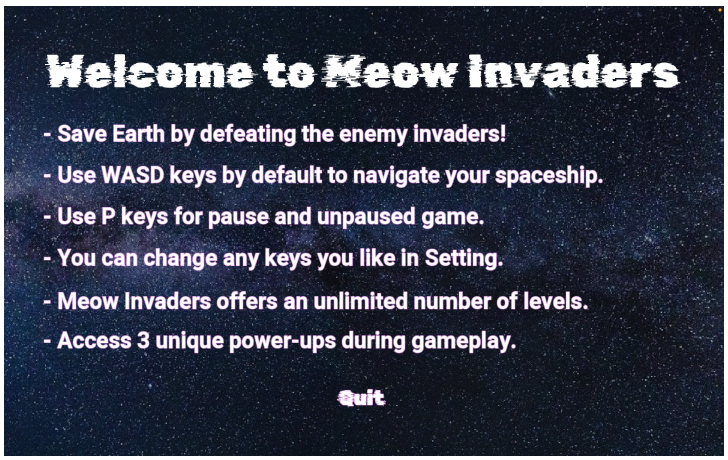

Figure 3: InstructionState

This is the SettingState of the project. It can changed the keys in the GameState.

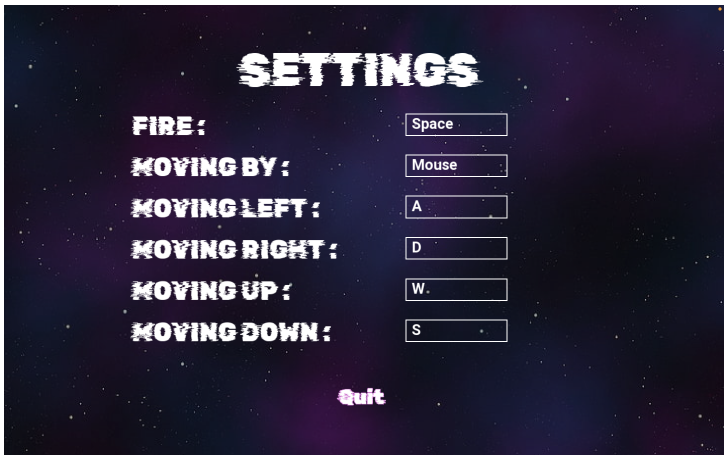

Figure 4: SettingState

This is the PreparedState of the project that will make the choice for the player and there are five spaceships for the player to choose.

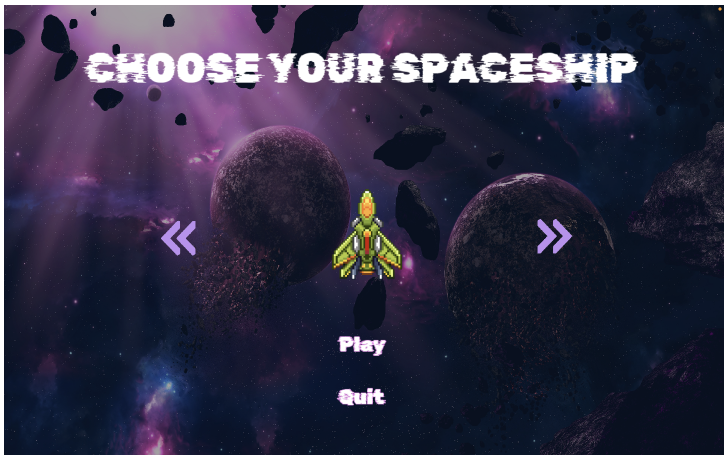

Figure 5: PreparedState

This is the GameState of the project. There are 3 enemies and 2 bosses. Each of the enemies will have their specific shooting. And there are also 5 different bullets.

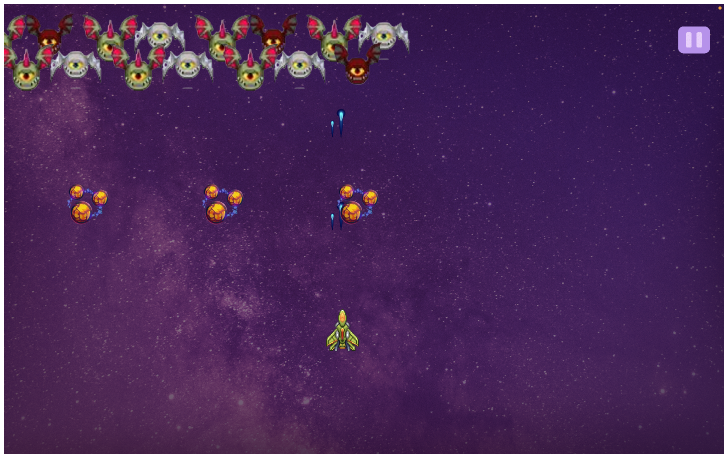

Figure 6: GameState

This is the Powerup in the project. There are 3 power up:

1. The red one will make the spaceship shoot 3 times.
2. The blue one will protect the spaceship.
3. The green one will make the spaceship shoot faster.

Figure 7: GameState

This is the random disaster in the project.

Figure 8: GameState

This is the boss. There are 2 bosses.

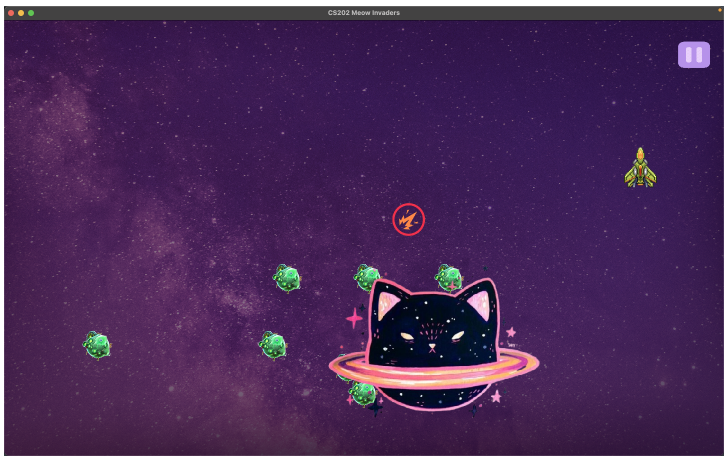

Figure 9: GameState

This is the UI when you lose.

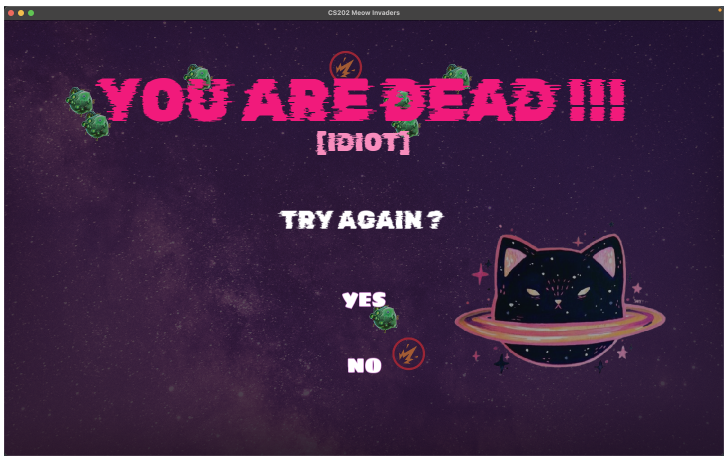

Figure 10: GameState

7 Conclusion

Through this project, I've gained a wealth of knowledge in applying the principles of OOP and designing effectively. Despite certain shortcomings and areas where improvements could be made, overall, I've put in my best effort, so I'm truly delighted to have completed this project.

8 References

[1] Making SPACE INVADERS in C++,Kofybrek