Ernest Ramazani

IUPUI Summer 1 2023

CSCI 43700

Final Project: Space Shooter Game

This documentation is for a simple space shooter game written in C++. The code utilizes the SFML (Simple and Fast Multimedia Library) framework for multimedia handling.

The game features a player-controlled spaceship which can move left or right and shoot bullets. The player's goal is to shoot down enemies that move from the top to the bottom of the screen. The game also includes power-ups, game timer, and a scoring system. The game ends when the player's health reaches zero.

Main Components

1. Player: The player-controlled spaceship. The spaceship's properties are managed in `Player.h` and `Player.cpp`. The `Player` class includes methods to move the spaceship, draw it to the screen, handle collisions, and update the spaceship's position. The player can shoot bullets at a regular interval by pressing the space bar.

2. Bullet: The bullets fired by the player's spaceship. Properties and methods are defined in `Bullet.h` and `Bullet.cpp`. Bullets are created at the player's position and move upwards.

3. Enemy: The enemies that the player must shoot down. Enemies are spawned at the top of the screen and move downwards. If they reach the bottom of the screen, the player loses health. The enemy properties and behaviors are managed in `Enemy.h` and `Enemy.cpp`.

4. PowerUp: Power-ups are items that the player can collect for benefits. There are three types of power-ups: Bomb (clears all enemies), Double Score (doubles the player's score), and Extra Life (increases player's health). Power-ups are managed in `PowerUp.h` and `PowerUp.cpp`.

5. Sound: The game utilizes sound effects for shooting bullets, collecting power-ups, and background music, which are managed by the SFML Audio library.

Game Loop

The game loop runs continuously until the player's health reaches zero or the game window is closed. The loop is responsible for handling events (like keyboard input), updating game state (like player/enemy/bullet positions and collisions), and drawing the game elements to the screen.

At the start of each game loop iteration, the player's inputs are processed, allowing the player to move the spaceship and shoot bullets. Then, the game state is updated. This includes moving the bullets, spawning and moving the enemies, handling bullet-enemy collisions, and handling player-enemy

collisions. If a bullet hits an enemy, the bullet and enemy are removed and the score is increased. If an enemy reaches the bottom of the screen, the player loses health and the enemy is removed.

Power-ups are spawned every 5 seconds and move down the screen. If a power-up collides with the player, it is activated and provides its effect to the player.

After the game state is updated, the game elements are drawn to the screen and displayed.

When the player's health reaches zero, the game over screen is displayed. Pressing 'R' restarts the game.

My apologies for the oversight. Let's add those details in.

Game Mechanics

The game follows a typical space shooter mechanic:

- Player Control: The player can move their spaceship left or right at the bottom of the screen using the left and right arrow keys. They can fire bullets towards the enemies by pressing the ***spacebar.***

- Enemy Movement: Enemies appear from the top of the screen and move downwards towards the player's spaceship. Their movement pattern can vary – some move in a straight line, while others can move in more complex patterns.

- Scoring: The player earns points by shooting down enemies. The score increases depending on the type of enemy ship destroyed. Some enemies may require more than one hit to be destroyed.

- Lives: The player starts with a certain number of lives. They lose a life if an enemy ship reaches the bottom of the screen or if the player's spaceship collides with an enemy. The game ends when all lives are lost. The player loses 20 points if he collides with the enemy and 10 points if an enemy reaches the bottom.

- Power-Ups: Power-ups are spawned every 5 seconds and move down the screen. If a power-up collides with the player, it is activated and provides its effect to the player. There are 3 types of power ups: the first one add 50 points to the player’s health, the second one add 20 points to the score and the last one destroy all the enemies on the screen.

- Difficulty Progression: As the player's score increases, the game's difficulty also increases. This involve enemies moving faster, more enemies appearing on the screen, or enemies requiring more hits to be destroyed.

- Sound Effects and Music: Sound effects are played for various game events, like the player firing a bullet, or collecting a power up. Background music is also played during the game.

Files

The main file for the game is `main.cpp`, and the other classes are stored in their respective header (`.h`) and source (`.cpp`) files. The other files include:

- Player.h and Player.cpp: Contains the player class with methods to move the player, update its position, draw it on the screen, handle collisions, and manage the player's health.

- Bullet.h and Bullet.cpp: Contains the Bullet class, which handles the creation, movement, drawing, and collision detection of bullets.

- Enemy.h and Enemy.cpp: Contains the Enemy class, which handles the creation, movement, drawing, collision detection, and removal of enemies.

- PowerUp.h and PowerUp.cpp : Contains the PowerUp class, which handles the creation, movement, drawing, activation, and collision detection of power-ups. The power-up effects are also implemented in this class.

Classes

- Player: The `Player` class is responsible for managing the player's spaceship. This includes moving the spaceship, firing bullets, and handling collisions with enemies.

- Bullet: The `Bullet` class handles bullets that the player fires. It's responsible for moving the bullets, detecting collisions with enemies, and removing bullets that have left the screen or hit an enemy.

- Enemy: The `Enemy` class handles the enemy spaceships. This includes moving the enemies, detecting collisions with the player or bullets, and removing enemies that have been shot or reached the bottom of the screen.

- PowerUp: The `PowerUp` class is responsible for managing power-ups. This includes moving the power-ups, detecting collisions with the player, and activating the power-up effect.

Prerequisites

To compile and run this game, you will need a C++ compiler and the SFML library. The SFML library provides the functionalities for windowing, graphics, and audio. It's highly recommended to have a basic understanding of C++ and familiarity with object-oriented programming. Also, understanding of game development concepts like game loops, collision detection, and basic game physics can be beneficial.

Appendix

Code

```
//main.cpp
#include <sstream>
#include<cstdlib>
#include <iostream>
#include<SFML/Audio.hpp>

using std::stringstream;

#include "Player.h"
#include "Enemy.h"
#include "Bullet.h"
#include "PowerUp.h"

int main() {
    sf::SoundBuffer bulletSoundBuffer;
    sf::SoundBuffer powerupSoundBuffer;
    sf::Music backgroundMusic;
    sf::Clock bulletClock;
    sf::Clock gameClock;
    sf::Clock powerupClock;
    sf::Clock enemySpawnClock;

    bool restart = false;
    bool bombCollected = false;

    int score = 0;
```

```
float currentEnemySpeedFactor = 1.0f;
int enemyCountFactor = 1;
const float TIME_TO_INCREASE_SPEED = 10.f;
const float SPEED_INCREMENT = 1.5f;

sf::RenderWindow window(sf::VideoMode(800, 600), "Final Project");

sf::Sprite backgroundSprite;



sf::Texture playerTexture, bulletTexture, enemyTexture,
   bombTexture, doubleScoreTexture, extraLifeTexture, backgroundTexture;

if (!playerTexture.loadFromFile("playerTexture.png") ||
   !bulletTexture.loadFromFile("bulletTexture.png") ||
   !enemyTexture.loadFromFile("enemyTexture.png") ||
   !bombTexture.loadFromFile("bombTexture.png") ||
   !doubleScoreTexture.loadFromFile("doubleScoreTexture.png") ||
   !extraLifeTexture.loadFromFile("extraLifeTexture.png") ||
   !backgroundTexture.loadFromFile("backgroundTexture.png")) {
   // handle error
}
float scaleX = static_cast<float>(window.getSize().x) / backgroundTexture.getSize().x;
float scaleY = static_cast<float>(window.getSize().y) / backgroundTexture.getSize().y;
backgroundSprite.setScale(scaleX, scaleY);

backgroundSprite.setTexture(backgroundTexture);
```

```
Player player(sf::Vector2f(400, 450), playerTexture);

if (!backgroundMusic.openFromFile("backgroundMusic.ogg") ||
    !bulletSoundBuffer.loadFromFile("bulletSound.ogg") ||
    !powerupSoundBuffer.loadFromFile("powerupSound.ogg")) {

}
backgroundMusic.setLoop(true);
backgroundMusic.play();

sf::Sound bulletSound;
bulletSound.setBuffer(bulletSoundBuffer);
bulletSound.setVolume(100);
sf::Sound powerupSound;
powerupSound.setBuffer(powerupSoundBuffer);

std::vector<Bullet> bullets;
std::vector<Enemy> enemies;
std::vector<PowerUp*> powerups;



for (int i = 0; i < 10; ++i) {
    bullets.emplace_back(sf::Vector2f(400, 200), sf::Vector2f(0, -1), bulletTexture);
}
for (int i = 0; i < 5; ++i) {
    enemies.emplace_back(sf::Vector2f(i * 150, 50), enemyTexture);
```

```
}



sf::Font font;
if (!font.loadFromFile("C:\\Windows\\Fonts\\arial.ttf")) {

}

sf::Text scoreText;
sf::Text gameOverText;

scoreText.setFont(font);
scoreText.setCharacterSize(24);
scoreText.setFillColor(sf::Color::White);
scoreText.setPosition(400, 0);

gameOverText.setFont(font);
gameOverText.setCharacterSize(32);
gameOverText.setFillColor(sf::Color::Red);
gameOverText.setString("Game Over \n Press R to Restart the Game");
gameOverText.setPosition(window.getSize().x / 2 - gameOverText.getLocalBounds().width / 2,
   window.getSize().y / 2 - gameOverText.getLocalBounds().height / 2);


sf::Text healthText;
healthText.setFont(font);
healthText.setCharacterSize(24);
healthText.setFillColor(sf::Color::White);
```

```
while (window.isOpen() && !restart) {
    sf::Event event;

    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close();
        }
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
        if (bulletClock.getElapsedTime().asSeconds() > 0.2f) {
            bullets.emplace_back(player.getPosition(), sf::Vector2f(0, -1), bulletTexture);
            bulletClock.restart();
            bulletSound.play();
        }
    }


    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
        // Move player left
        player.move(sf::Vector2f(-0.1, 0));
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
        // Move player right
        player.move(sf::Vector2f(0.1, 0));
    }

    if (gameClock.getElapsedTime().asSeconds() > TIME_TO_INCREASE_SPEED) {
```

```
    currentEnemySpeedFactor *= SPEED_INCREMENT; // Double the speed factor
    for (Enemy& enemy : enemies) {
        enemy.increaseSpeed(currentEnemySpeedFactor); // Pass the speed factor to enemies
    }
    gameClock.restart(); // Reset the clock
}


//Update game state
player.update(window);


for (Bullet& bullet : bullets) {
    bullet.update();
}

for (Enemy& enemy : enemies) {
    enemy.update(window);
}



  // Collision checks
for (auto bullet_it = bullets.begin(); bullet_it != bullets.end(); ) {
    Bullet& bullet = *bullet_it;
    bullet.update();

    bool bullet_collided_with_enemy = false;
    for (auto enemy_it = enemies.begin(); enemy_it != enemies.end(); ) {
```

```
            Enemy& enemy = *enemy_it;

            if (bullet.getBounds().intersects(enemy.getBounds())) {
                bullet_collided_with_enemy = true;
                enemy_it = enemies.erase(enemy_it);
                enemies.emplace_back(sf::Vector2f(rand() % 800, 0), enemyTexture,
currentEnemySpeedFactor);
                score += 1;
            }

            else {
                ++enemy_it;
            }
        }

        // If bullet has gone off the screen, or it hit an enemy, remove it
        if (bullet.getPosition().y < 0 || bullet_collided_with_enemy) {
            bullet_it = bullets.erase(bullet_it);
        }
        else {
            ++bullet_it;
        }
    }


    // Similarly, update enemy loop
    for (auto it = enemies.begin(); it != enemies.end(); ) {
        Enemy& enemy = *it;
```

```
        // Update enemy and handle collision with player
        enemy.update(window);
        if (enemy.getBounds().intersects(player.getBounds())) {
            player.setHealth(player.getHealth() - 20);
            it = enemies.erase(it);
            for (int i = 0; i < enemyCountFactor; i++) {
                enemies.emplace_back(sf::Vector2f(rand() % 800, 0), enemyTexture,
currentEnemySpeedFactor);
            }
        }

        else if (enemy.getPosition().y >= window.getSize().y) {
            if (!enemy.getHasImpacted()) {
                player.setHealth(player.getHealth() - 10);
                enemy.setHasImpacted(true);
                std::cout << "Enemy reached bottom, player health: " << player.getHealth() << std::endl;
            }
            enemy.setPosition(sf::Vector2f(rand() % 800, 0));  // Reset enemy position
            it = enemies.erase(it);
            enemies.emplace_back(sf::Vector2f(rand() % 800, 0), enemyTexture);
        }



        else {
            ++it;
        }
    }
```

```
// Update powerups and generate new ones every 5 seconds
    if (powerupClock.getElapsedTime().asSeconds() >= 5.f) { // Check if 5 seconds have passed
        int powerupType = rand() % 3;
        sf::Vector2f position(rand() % 800, 0); // Random x position at the top of the screen
        if (powerupType == 0) {
            PowerUp* bomb = new Bomb(position, bombTexture);
            bomb->setDirection(sf::Vector2f(0, 1));
            bomb->setSpeed(0.05f);
            powerups.push_back(bomb);
        }
        else if (powerupType == 1) {
            PowerUp* doubleScore = new DoubleScore(position, doubleScoreTexture);
            doubleScore->setDirection(sf::Vector2f(0, 1));
            doubleScore->setSpeed(0.05f);
            powerups.push_back(doubleScore);
        }
        else {
            PowerUp* extraLife = new ExtraLife(position, extraLifeTexture);
            extraLife->setDirection(sf::Vector2f(0, 1));
            extraLife->setSpeed(0.05f);
            powerups.push_back(extraLife);
        }
        powerupClock.restart(); // Reset the clock
    }


   // Updating and drawing power-ups
    for (auto powerup_it = powerups.begin(); powerup_it != powerups.end(); ) {
```

```
        PowerUp* powerup = *powerup_it;
        powerup->update(); // Update power-up position




        if (powerup->getBounds().intersects(player.getBounds())) {
            powerup->activate(enemies, player, score);
            powerupSound.play(); // Play the powerup sound
            delete powerup;
            powerup_it = powerups.erase(powerup_it);
        }
        else if (powerup->getPosition().y > window.getSize().y) { // If powerup goes beyond screen, remove it
            delete powerup;
            powerup_it = powerups.erase(powerup_it);
        }
        else {
            ++powerup_it;
        }
    }


    // After Bomb Collection
    for (auto powerup_it = powerups.begin(); powerup_it != powerups.end(); ) {
        PowerUp* powerup = *powerup_it;
        powerup->update();


        if (powerup->getBounds().intersects(player.getBounds())) {
            powerup->activate(enemies, player, score);
```

```
        // Check if the powerup was a Bomb
        Bomb* bomb = dynamic_cast<Bomb*>(powerup);
        if (bomb != nullptr) {
            bombCollected = true;
            enemySpawnClock.restart();
        }

        delete powerup;
        powerup_it = powerups.erase(powerup_it);
    }
    else {
        ++powerup_it;
    }
}

// Before spawning new enemies, check if a bomb has been collected recently
if (bombCollected && enemySpawnClock.getElapsedTime().asSeconds() < 2.f) {
    continue;
}
// Draw everything


window.clear();
window.draw(backgroundSprite);
std::stringstream scoreSS;
scoreSS << "Score: " << score;
scoreText.setString(scoreSS.str());
```

```
std::stringstream healthSS;
healthSS << "Health: " << player.getHealth();
healthText.setString(healthSS.str());

// Draw score text
window.draw(scoreText);

// Draw health text
window.draw(healthText);



player.draw(window);

for (Bullet& bullet : bullets) {
    bullet.draw(window);
}

for (Enemy& enemy : enemies) {
    enemy.draw(window);
}



for (PowerUp* powerup : powerups) {
    powerup->draw(window);
}

window.display();
// Game Over Condition
if (player.getHealth() <= 0) {
```

```
        break; // Game Over, exit the game loop
    }
}
// New game over loop :
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) {
            window.close();
        }

        // Restart condition ( pressing the "R" key)
        if (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::R) {
            restart = true;
            break;
        }
    }

    if (restart || !window.isOpen()) {
        break;
    }

    window.clear();
    window.draw(backgroundSprite);
    window.draw(gameOverText);
    window.display();
}

if (restart) {
```

```
        main(); // Recursive call to main function, starting the game again

    }



    return 0;

}
```

Player.cpp

```
#include "Player.h"

Player::Player(const sf::Vector2f& position, const sf::Texture& texture) {
	this->position = position;
	velocity = sf::Vector2f(0, 0);
	health = 1000;

	sprite.setTexture(texture);
	sprite.setPosition(this->position);
	sprite.setScale(0.12f, 0.12f);


}

Player::~Player() {

}

void Player::move(const sf::Vector2f& offset) {
	velocity = offset;
	position += velocity;
	sprite.setPosition(position);
}

void Player::draw(sf::RenderWindow& window) {
	window.draw(sprite);
}

void Player::handleCollision() {
}

void Player::update(sf::RenderWindow& window) {
	position += velocity;
	// Handle border collision
	if (position.x < 0) {
		position.x = 0;
	}
	else if (position.x > window.getSize().x - sprite.getGlobalBounds().width) {
		position.x = window.getSize().x - sprite.getGlobalBounds().width;
	}


	sprite.setPosition(position);
	velocity = sf::Vector2f(0, 0);
```

```
}

sf::FloatRect Player::getBounds() const {
	return sprite.getGlobalBounds();
}

int Player::getHealth() {
	return health;
}

void Player::setHealth(int Health) {
	this->health = Health;
}

sf::Vector2f Player::getPosition() const {
	return position;
}

void Player::fire(std::vector<Bullet>& bullets, const sf::Texture& bulletTexture) {
	sf::Vector2f bulletVelocity(0.0f, -1.0f);
	bullets.emplace_back(getPosition(), bulletVelocity, bulletTexture);
}
```

Bullet.cpp

```
#include "Bullet.h"


Bullet::Bullet(const sf::Vector2f& position, const sf::Vector2f& velocity, const
sf::Texture& texture) {
    this->position = position;
    this->velocity = sf::Vector2f(0,-0.9f);

    sprite.setTexture(texture);
    sprite.setPosition(this->position);
    sprite.setScale(0.2f, 0.2f);
}

Bullet::~Bullet() {

}

void Bullet::move() {
    // Bullets typically move in a straight line, so we don't need any complex logic
here
    position += velocity;
    sprite.setPosition(position);
}

void Bullet::draw(sf::RenderWindow& window) {
    window.draw(sprite);
}

void Bullet::handleCollision() {
    // implementation depends on what it collides with (enemy, player, etc.)
}

sf::FloatRect Bullet::getBounds() const {
    return sprite.getGlobalBounds();
```

```
}
sf::Vector2f Bullet::getPosition() const {
    return sprite.getPosition();
}

void Bullet::update() {
    move();
    sprite.setPosition(position);
}

bool Bullet::operator==(const Bullet& other) const {
    // Compare relevant fields for equality
    return position == other.position && velocity == other.velocity;
    // Add more fields to compare if needed
}
```

Enemy.cpp

```
#include<iostream>
#include "Enemy.h"

Enemy::Enemy(const sf::Vector2f& position, const sf::Texture& texture) {
    this->position = position;
    velocity = sf::Vector2f(0, 0.01f); // initial velocity can be updated based on
AI logic
    health = 100;

    sprite.setTexture(texture);
    sprite.setPosition(this->position);
    sprite.setScale(0.1f, 0.1f);
    hasImpacted = false;
}

Enemy::Enemy(sf::Vector2f position, sf::Texture& texture, float velocityFactor)
    : position(position), hasImpacted(false) {
    sprite.setTexture(texture);
    velocity = sf::Vector2f(0, 0.01f * velocityFactor);
    health = 100;
    sprite.setPosition(this->position);
    sprite.setScale(0.15f, 0.15f);
}


Enemy::~Enemy() {

}
void Enemy::move() {

    position += velocity;
    sprite.setPosition(position);
}

void Enemy::draw(sf::RenderWindow& window) {
    window.draw(sprite);
}

void Enemy::handleCollision() {

}
```

```
void Enemy::update(sf::RenderWindow& window) {
    move();
    sprite.setPosition(position);



}

sf::FloatRect Enemy::getBounds() const {
    return sprite.getGlobalBounds();
}

void Enemy::setHealth(int newHealth) {
    health = newHealth;
}

int Enemy::getHealth() const {
    return health;
}

sf::Vector2f Enemy::getPosition() const {
    return position;
}



bool Enemy::getHasImpacted() const {
    return hasImpacted;
}

void Enemy::setHasImpacted(bool impacted) {
    hasImpacted = impacted;
}

void Enemy::setPosition(const sf::Vector2f& newPosition) {
    position = newPosition;
    sprite.setPosition(position);
}

void Enemy::setVelocity(const sf::Vector2f& newVelocity) {
    velocity = newVelocity;
}
void Enemy::increaseSpeed(float factor) {
    sf::Vector2f currentVelocity = velocity;
    setVelocity(currentVelocity * factor);
}
```

PowerUp.cpp

```
#include "PowerUp.h"

PowerUp::PowerUp(const sf::Vector2f& position, const sf::Texture& texture, float
speed)
    : position(position), speed(speed) { //
    sprite.setTexture(texture);
    sprite.setPosition(this->position);
    sprite.setScale(0.1f, 0.1f); // Adjust as necessary
}
```

```
void PowerUp::draw(sf::RenderWindow& window) {
    window.draw(sprite);
}

sf::FloatRect PowerUp::getBounds() const {
    return sprite.getGlobalBounds();
}


// Implementations for Bomb, DoubleScore, and ExtraLife
Bomb::Bomb(const sf::Vector2f& position, const sf::Texture& texture)
    : PowerUp(position, texture, speed) {}

void Bomb::activate(std::vector<Enemy>& enemies, Player& player, int& score) {
    // Bomb resets all enemies

    for (Enemy& enemy : enemies) {


        enemy.setPosition(sf::Vector2f(rand() % 800, 0));
    }
}

DoubleScore::DoubleScore(const sf::Vector2f& position, const sf::Texture& texture)
    : PowerUp(position, texture, speed) {}

void DoubleScore::activate(std::vector<Enemy>& enemies, Player& player, int& score)
{
    score += 20; // Double the current score
}

ExtraLife::ExtraLife(const sf::Vector2f& position, const sf::Texture& texture)
    : PowerUp(position, texture, speed) {}

void ExtraLife::activate(std::vector<Enemy>& enemies, Player& player, int& score) {
    player.setHealth(player.getHealth() + 50); // Adds 50 to player's health
}

void PowerUp::setDirection(const sf::Vector2f& new_direction) {
    direction = new_direction;
}

void PowerUp::setSpeed(float new_speed) {
    speed = new_speed;
}

sf::Vector2f PowerUp::getPosition() const {
    return position;
}

// update function
void PowerUp::update() {
    position += speed * direction;
    sprite.setPosition(position);  // Update sprite position
}
```

powerUp.h

```
#ifndef POWERUP_H
#define POWERUP_H

#include <SFML/Graphics.hpp>
#include "Player.h"
#include "Enemy.h"

class PowerUp {
protected:
    sf::Vector2f position;
    sf::Sprite sprite;
    float speed;
    sf::Vector2f direction;

public:
    PowerUp(const sf::Vector2f& position, const sf::Texture& texture, float speed);
    virtual void activate(std::vector<Enemy>& enemies, Player& player, int& score) =
0;  // Pure virtual function
    void draw(sf::RenderWindow& window);
    void update();
    void setDirection(const sf::Vector2f& new_direction);
    void setSpeed(float new_speed);
    sf::Vector2f getPosition() const;
    sf::FloatRect getBounds() const;
};


class Bomb : public PowerUp {
public:
    Bomb(const sf::Vector2f& position, const sf::Texture& texture);
    void activate(std::vector<Enemy>& enemies, Player& player, int& score) override;
};

class DoubleScore : public PowerUp {
public:
    DoubleScore(const sf::Vector2f& position, const sf::Texture& texture);
    void activate(std::vector<Enemy>& enemies, Player& player, int& score) override;
};

class ExtraLife : public PowerUp {
public:
    ExtraLife(const sf::Vector2f& position, const sf::Texture& texture);
    void activate(std::vector<Enemy>& enemies, Player& player, int& score) override;
};

#endif // POWERUP_H
```

Enemy.h

```
#ifndef ENEMY_H
#define ENEMY_H
#include <SFML/Graphics.hpp>

class Enemy {
private:
    sf::Vector2f position;
    sf::Vector2f velocity;
```

```
    int health;
    sf::Sprite sprite;
    bool hasImpacted;


public:

    Enemy(const sf::Vector2f& position, const sf::Texture& texture);
    ~Enemy();


    Enemy(sf::Vector2f position, sf::Texture& texture, float velocityFactor);

    void move();
    void draw(sf::RenderWindow& window);
    void handleCollision();
    void update(sf::RenderWindow& window);
    sf::FloatRect getBounds() const;
    void setHealth(int newHealth);
    int getHealth() const;
    sf::Vector2f getPosition() const;
    bool getHasImpacted() const;
    void setHasImpacted(bool impacted);
    void setPosition(const sf::Vector2f& newPosition);
    void setVelocity(const sf::Vector2f& newVelocity);
    void increaseSpeed(float factor);


};

Player.h
#ifndef PLAYER_H
#define PLAYER_H
#include<vector>
#include "Bullet.h"
#include "SFML/Graphics.hpp"

class Player {
private:
      sf::Vector2f position;
      sf::Vector2f velocity;
      int health;
      sf::Sprite sprite;
public:

      Player(const sf::Vector2f& position, const sf::Texture& texture);
      ~Player();
      void move(const sf::Vector2f& offset);
      void draw(sf::RenderWindow& window);
      void handleCollision();
      void update(sf::RenderWindow& window);
      sf::FloatRect getBounds() const;
      sf::Vector2f getPosition() const;
      int getHealth();
      void setHealth(int Health);
      void fire(std::vector<Bullet>& bullets, const sf::Texture& bulletTexture);
      int lives = 3;
};
```

```
#endif // !PLAYER_H


Bullet.h
#ifndef BULLET_H
#define BULLET_H
#include <SFML/Graphics.hpp>

class Bullet {
private:
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Sprite sprite;

public:
    Bullet(const sf::Vector2f& position, const sf::Vector2f& velocity, const
sf::Texture& texture);
    ~Bullet();
    void move();
    void draw(sf::RenderWindow& window);
    void handleCollision();
    sf::FloatRect getBounds() const;
    void update();
    sf::Vector2f getPosition() const;
    bool operator==(const Bullet& other) const;

};


#endif // !BULLET_H
```