Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki

Programowanie Komputerów

Sprawozdanie z projektu gry „Space Invaders”

Autor: Maria Wyganowska
Prowadzący: mgr Tomasz Kukuczka
Data: 23.06.2024

Temat projektu: gra typu Space Invaders

Analiza tematu

Opis projektu: Program polega na strzelaniu we wrogie statki pojazdem sterowanym przez gracza. Wrogowie strzelają z pewnym odstępem czasowym pociski, które lecą w dół w stronę statku użytkownika. Gdy zdrowie gracza spadnie do 0, gra się kończy i uzyskany wynik jest zapisywany do tabeli z najlepszymi wynikami. Celem gry jest pokonanie wszystkich wrogich statków w danym poziomie.
Szata graficzna została zrealizowana za pomocą biblioteki SFML.

Język programowania: C++

Specyfikacja zewnętrzna

Po uruchomieniu gry użytkownik z menu głównego ma kilka możliwości wyboru. Strzałkami góra i dół można zmienić aktualnie wybieraną opcję, a przyciskiem ENTER zatwierdzany jest wybór. Aktualnie wybierana opcja jest w kolorze fuksji.

Zrzut ekranu głównego menu

Po wciśnięciu „Start game” uruchamiana jest gra. Może się zdarzyć, że prędkość pocisków i statku gracza są za szybkie/za wolne- wartość prędkości pomiędzy moim komputerem a laptopem inaczej była realizowana pomiędzy tymi urządzeniami, co zauważyłam dopiero przy zaimportowaniu projektu na laptop. Należy wtedy w funkcjach player::moveLeft, player::moveRight itd. i player::fireBullet oraz w klasie invader zmienić wartości velocity tak, aby obiekty na ekranie poruszały się w racjonalny sposób.

Po uruchomieniu gry ustawiany jest poziom (generowane są typy wrogów i ustawiani są oni na ekranie) i rozpoczynana jest gra:

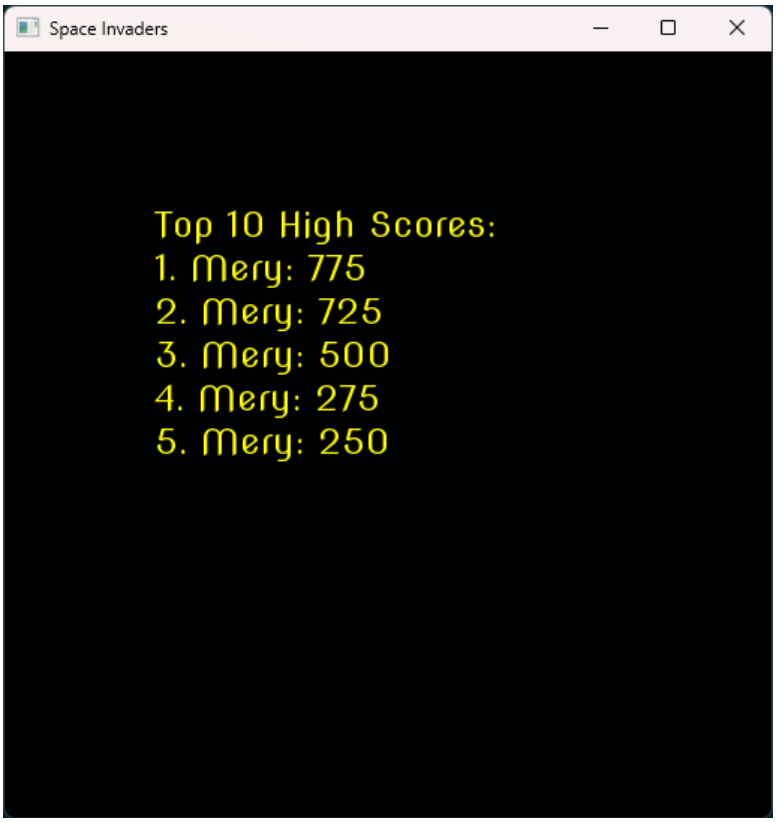
Zrzut ekranu z rozgrywki oraz przegranie gry

Sterowanie:

- strzałki – przemieszczanie statku gracza
- Z – strzelanie pociskami

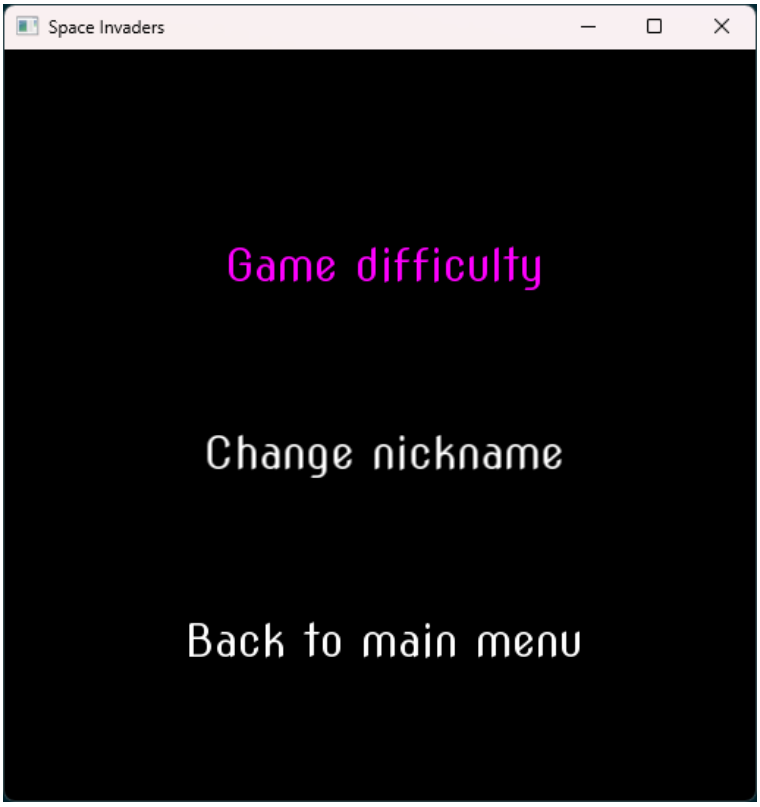
Gdy pasek zdrowia spadnie do zera, gra kończy się- uruchamia się ekran game over. Wynik gracza jest zapisywany (każde trafienie wroga to 25 punktów) wraz z jego pseudonimem (domyślnie jest to Mery). Po skończeniu gry gracz wraca do menu głównego.

W menu głównym można również zobaczyć 10 najlepszych wyników w zakładce „high scores” (aby z niego wyjść, należy wcisnąć przycisk ESC):

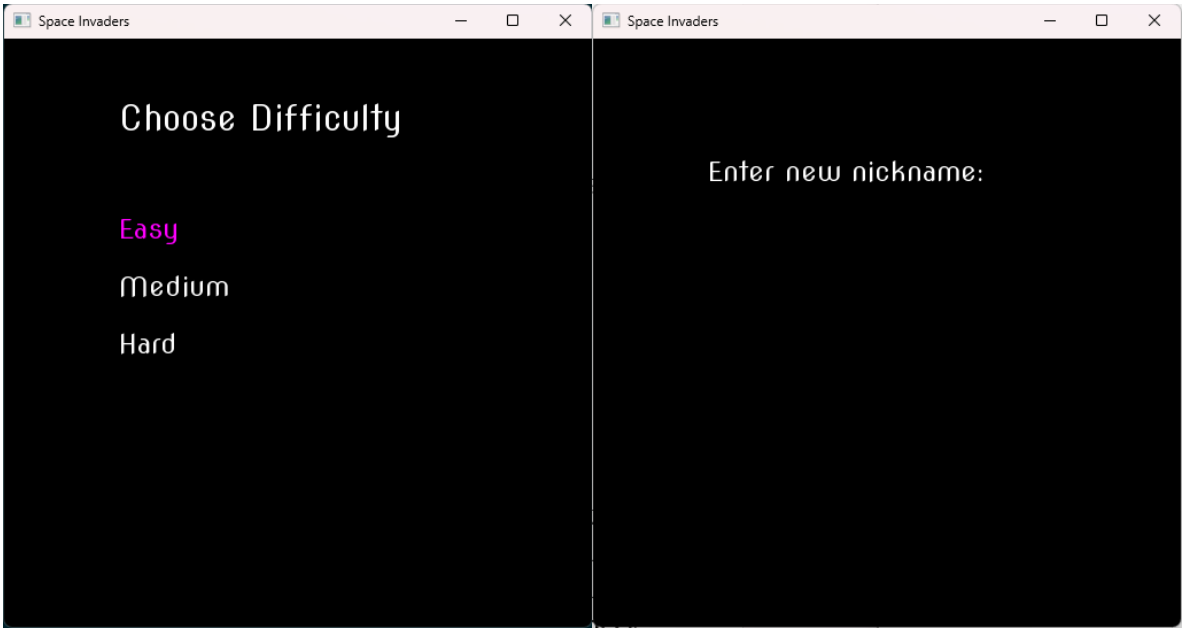


Zrzut ekranu z najlepszymi wynikami

W menu opcji można zmienić poziom trudności rozgrywki (co zmienia prędkość, z jaką poruszają się pociski wrogów) oraz pseudonim gracza:



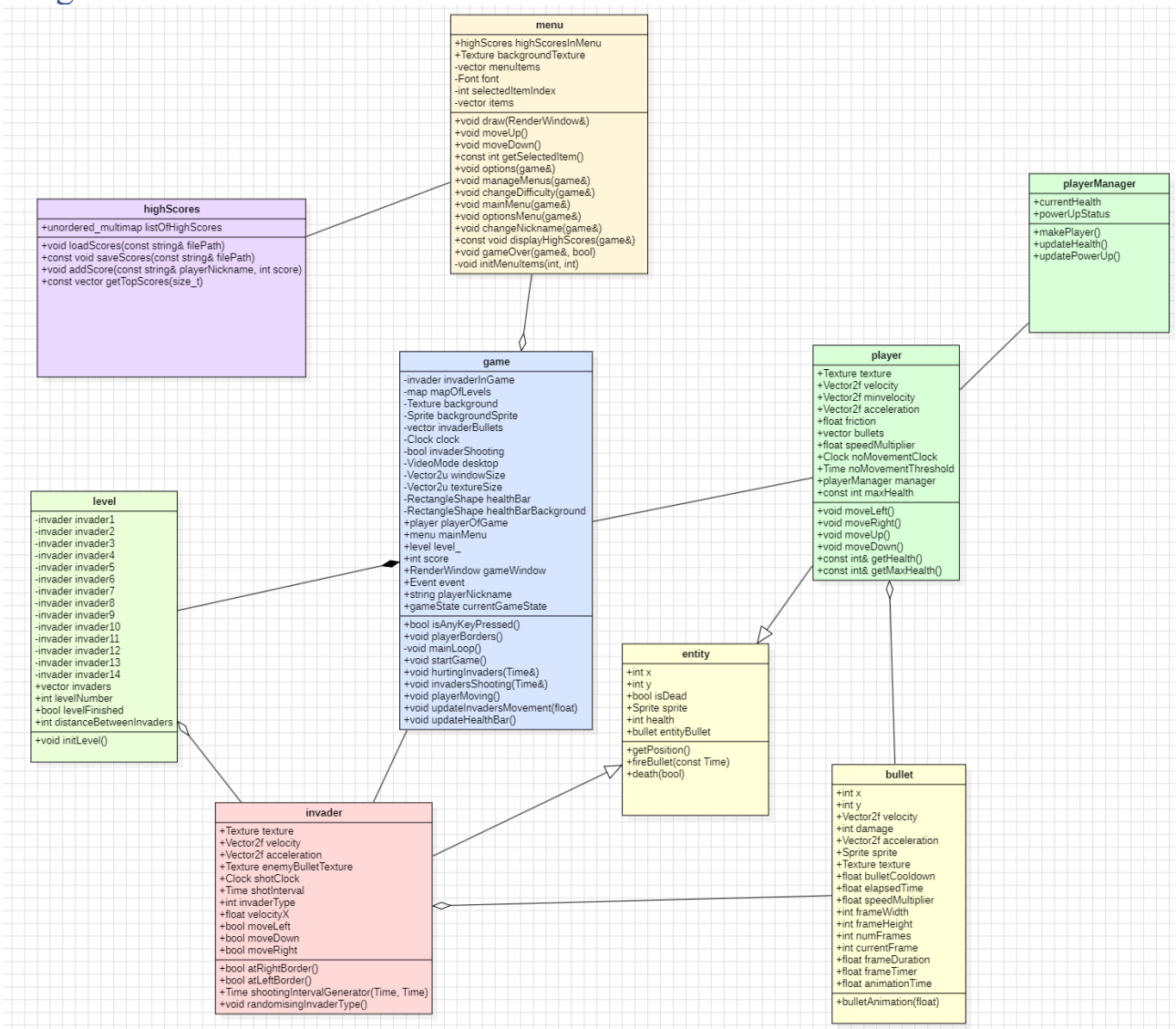
Zrzut ekranu z opcjami



Zrzuty ekranu zmiany poziomu trudności oraz zmiany pseudonimu gracza

Specyfikacja wewnętrzna

Diagram klas



Klasa player

```
class player :public entity {
public:
	sf::Texture texture;
	sf::Vector2f velocity{ 0.0f, 0.0f };
	sf::Vector2f minvelocity{ 0.0f, 0.0f };
	sf::Vector2f acceleration{ 0.001f, 0.001f };
	float friction;
	std::vector<bullet> bullets;
	float speedMultiplier;
	sf::Clock noMovementClock;
	sf::Time noMovementThreshold;
	void moveLeft();
	void moveRight();
	void moveUp();
	void moveDown();
	void fireBullet(const sf::Time&) override;
	playerManager manager;
	const int maxHealth = 150.f;
	const int& getHealth() const;
	const int& getMaxHealth() const;
```

```
};
```
Dziedziczy ona po klasie entity i reprezentuje obiekt gracza widziany na ekranie.
Istotne metody:
```
void player::fireBullet(const sf::Time& time) {
	entityBullet.elapsedTime += time.asSeconds();
	if (sf::Keyboard::isKeyPressed(sf::Keyboard::Z) && entityBullet.elapsedTime >= 
entityBullet.bulletCooldown && !isDead) {
		entityBullet.sprite.setPosition(sprite.getPosition().x + 
sprite.getGlobalBounds().width / 2 - entityBullet.sprite.getGlobalBounds().width / 2, 
sprite.getPosition().y);
		entityBullet.velocity = sf::Vector2f(0.f, -0.5f);
		entityBullet.currentFrame = 0;
		entityBullet.animationTime = 0.f;
		entityBullet.sprite.setTextureRect(sf::IntRect(0, 0, entityBullet.frameWidth, 
entityBullet.frameHeight));
		bullets.push_back(entityBullet);
		entityBullet.elapsedTime = 0.f;
	}
	for (auto it = bullets.begin(); it != bullets.end();) {
		it->sprite.move(it->velocity);
		it->bulletAnimation(time.asSeconds());
		if (it->sprite.getPosition().y < 0) {
			it = bullets.erase(it);
		}
		else {
			++it;
		}
	}
}
```
Metoda fireBullet jest nadpisywaną metodą klasy entity. Metoda ta sprawdza, czy wciśnięty został klawisz Z- jeśli tak, to ustawia nowy pocisk na środku sprite'a gracza, nadaje mu odpowiednią prędkość i dodaje do wektora pocisków widzianych na ekranie. Uruchamia także animację pocisku.

```
void player::moveUp() {
	if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && !isDead)
	{
		velocity.y -= 0.8f;
		noMovementClock.restart();

	}
	return;
}
```
Jedna z metod moveX(), która obsługuje poruszanie się gracza po ekranie. Wszystkie te metody zostały „wrzucone" do jednej metody game::playerMoving.

Klasa invader
```
class invader :public entity {
public:
	sf::Texture texture;
	sf::Vector2f velocity{ 0.02f, 0.02f };
	sf::Vector2f acceleration{ 0.001f, 0.001f };
	float friction;
	bool atRightBorder();
	bool atLeftBorder();
	sf::Texture enemyBulletTexture;
	sf::Clock shotClock;
	sf::Time shotInterval;
	sf::Time shootingIntervalGenerator(sf::Time, sf::Time);
	int invaderType;
	void fireBullet(const sf::Time&) override;
	float velocityX;
	bool moveLeft;
	bool moveDown;
	bool moveRight;
```

```
        void randomisingInvaderType();
};
```

Klasa invader reprezentuje jednego wroga widzianego na ekranie. Dziedziczy po klasie entity.
Istotne metody:

```
sf::Time invader::shootingIntervalGenerator(sf::Time minInterval, sf::Time maxInterval) {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> distr(minInterval.asSeconds(), maxInterval.asSeconds());
        return sf::seconds(distr(gen));
}
```

Metoda ta zajmuje się generowaniem losowego interwału dla strzelania pocisków przez wrogów. Została zaimplementowana po to, żeby wrogowie nie strzelali falami w tym samym momencie.

```
void invader::randomisingInvaderType() {
    std::cout << "Randomising invader type for invader " << this << std::endl;
        std::cout << invaderType << std::endl;
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> distr(1, 3);
        invaderType = distr(gen);
        std::cout << invaderType << std::endl;
        switch (invaderType) {
    case 1: {
        if (!texture.loadFromFile("invader1.png")) {
            std::cout << ":(" << std::endl;
        }
        else {
            sprite.setTexture(texture);
            std::cout << ":)" << std::endl;
        }
        health = 40;
        break;
    }
    case 2: {
        if (!texture.loadFromFile("invader2.png")) {
            std::cout << "Nie udało się załadować tekstury invader2.png" << std::endl;
        }
        else {
            sprite.setTexture(texture);
            std::cout << "Invader typ 2 ustawiony" << std::endl;
        }
        health = 100;
        break;
    }
    case 3: {
        if (!texture.loadFromFile("invader3.png")) {
            std::cout << "Nie udało się załadować tekstury invader3.png" << std::endl;
        }
        else {
            sprite.setTexture(texture);
            std::cout << "Invader typ 3 ustawiony" << std::endl;
        }
        health = 20;
        break;
    }
        }
        return;
}
```

W celu urozmaicenia rozgrywki zaimplementowałam metodę randomisingInvaderType(), którą można przy ustawianiu poziomu wylosować typ danego wroga, co zmienia jego wygląd i ilość zdrowia.

Klasa menu

```
class menu {
public:
    menu() = default;
    menu(float, float);
    void draw(sf::RenderWindow&);
    void moveUp();
    void moveDown();
    int getSelectedItem() const;
    void options(game&);
    void manageMenus(game&);
    highScores highScoresInMenu;
    void changeDifficulty(game&);
    void mainMenu(game&);
    void optionsMenu(game&);
    void changeNickname(game&);
    sf::Texture backgroundTexture;
    void displayHighScores(game& SpaceInvaders) const;
    void gameOver(game&, bool);
private:
    std::vector<sf::Text> menuItems;
    sf::Font font;
    int selectedItemIndex;
    std::vector<std::string> items;
    void initMenuItems(int, int);
};
```

Klasa menu obsługuje działanie różnych menu w interfejsie gry.
Istotne metody:

```
void menu::displayHighScores(game& SpaceInvaders) const {
    highScores& mutableHighScoresInMenu = const_cast<highScores&>(highScoresInMenu);
    mutableHighScoresInMenu.loadScores("scores/high_scores.txt");
    sf::Font font;
    if (!font.loadFromFile("NovaSlim-Regular.ttf")) {
        std::cout << "Failed to load font file." << std::endl;
        return;
    }
    std::vector<std::pair<std::string, int>> topScores =
mutableHighScoresInMenu.getTopScores(10);

    sf::Text scoresText;
    scoresText.setFont(font);
    scoresText.setCharacterSize(24);
    scoresText.setFillColor(sf::Color::Yellow);
    scoresText.setPosition(100, 100);

    std::string scoresDisplay = "Top 10 High Scores:\n";
    for (size_t i = 0; i < topScores.size(); ++i) {
        scoresDisplay += std::to_string(i + 1) + ". " + topScores[i].first + ": " +
std::to_string(topScores[i].second) + "\n";
    }
    scoresText.setString(scoresDisplay);

    bool exitHighScores = false;
    while (SpaceInvaders.gameWindow.isOpen() && !exitHighScores) {
        sf::Event event;
        while (SpaceInvaders.gameWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                SpaceInvaders.gameWindow.close();
            }
            else if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Escape) {
                    SpaceInvaders.currentGameState = gameState::Options;
                    exitHighScores = true;
                }
            }
        }
        SpaceInvaders.gameWindow.clear();
        SpaceInvaders.gameWindow.draw(scoresText);
        SpaceInvaders.gameWindow.display();
```

```
    }
    SpaceInvaders.currentGameState = gameState::MainMenu;
}
```

Metoda ta pobiera, filtruje i wyświetla 10 najwyższych wyników.

```
void menu::changeDifficulty(game& SpaceInvaders) {
    std::vector<std::string> difficulties = { "Easy", "Medium", "Hard" };
    while (SpaceInvaders.gameWindow.isOpen()) {
        sf::Event event;
        while (SpaceInvaders.gameWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                SpaceInvaders.gameWindow.close();
            }
            else if (event.type == sf::Event::KeyReleased) {
                if (event.key.code == sf::Keyboard::Up) {
                    selectedItemIndex = (selectedItemIndex - 1 + difficulties.size()) %
difficulties.size();
                }
                else if (event.key.code == sf::Keyboard::Down) {
                    selectedItemIndex = (selectedItemIndex + 1) % difficulties.size();
                }
                else if (event.key.code == sf::Keyboard::Return) {
                    switch (selectedItemIndex) {
                    case 0:
                        for (auto el : SpaceInvaders.level_.invaders) {
                            el.entityBullet.velocity = sf::Vector2f(0.f, 0.025f);
                        }
                        break;
                    case 1:
                        for (auto el : SpaceInvaders.level_.invaders) {
                            el.entityBullet.velocity = sf::Vector2f(0.f, 0.125f);
                        }
                        break;
                    case 2:
                        for (auto el : SpaceInvaders.level_.invaders) {
                            el.entityBullet.velocity = sf::Vector2f(0.f, 0.325f);
                        }
                        break;
                    }
                    return;
                }
            }
        }
        SpaceInvaders.gameWindow.clear();
        sf::Font font;
        if (!font.loadFromFile("NovaSlim-Regular.ttf")) {
        }
        sf::Text title("Choose Difficulty", font, 30);
        title.setPosition(100, 50);
        SpaceInvaders.gameWindow.draw(title);
        for (size_t i = 0; i < difficulties.size(); ++i) {
            sf::Text text(difficulties[i], font, 24);
            text.setPosition(100, 150 + i * 50);
            if (i == selectedItemIndex) {
                text.setFillColor(sf::Color::Magenta);
            }
            SpaceInvaders.gameWindow.draw(text);
        }

        SpaceInvaders.gameWindow.display();
    }
    selectedItemIndex = 0;
}
```

Metoda ta ustawia wybrany przez użytkownika poziom trudności i zmienia prędkość pocisków wszystkich wrogów w wektorze wrogów na danym poziomie.

```
void menu::changeNickname(game& SpaceInvaders) {
    std::string newNickname;
    bool enteringNickname = true;
    sf::Text nicknameText;
```

```
    sf::Font font;

    if (!font.loadFromFile("NovaSlim-Regular.ttf")) {
        std::cout << ":(" << std::endl;
    }
    nicknameText.setFont(font);
    nicknameText.setCharacterSize(24);
    nicknameText.setFillColor(sf::Color::White);
    nicknameText.setPosition(100, 100);
    std::regex nicknameRegex("^[a-zA-Z0-9]+$");
    bool exitChangeNickname = false;
    std::string displayedText = "Enter new nickname: ";
    while (SpaceInvaders.gameWindow.isOpen() && enteringNickname) {

        sf::Event event;
        while (SpaceInvaders.gameWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                SpaceInvaders.gameWindow.close();
            }
            else if (event.type == sf::Event::TextEntered) {
                if (event.text.unicode < 128) {
                    if (event.text.unicode == '\b') {
                        if (!newNickname.empty()) {
                            newNickname.pop_back();
                        }
                    }
                    else if (event.text.unicode == '\r') {
                        if (std::regex_match(newNickname, nicknameRegex)) {
                            SpaceInvaders.playerNickname = newNickname;
                            std::cout << SpaceInvaders.playerNickname << std::endl;
                            enteringNickname = false;
                            while (event.type != sf::Event::KeyReleased || event.key.code !=
sf::Keyboard::Return) {
                                SpaceInvaders.gameWindow.pollEvent(event);
                            }
                            SpaceInvaders.currentGameState = gameState::Options;
                            return;
                        }
                        else {
                            newNickname.clear();
                            displayedText = "Invalid nickname! \nEnter new nickname: ";

                        }

                    }
                    else {
                        newNickname += static_cast<char>(event.text.unicode);
                    }

                }
            }
            else if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Escape) {
                    enteringNickname = false;
                    SpaceInvaders.currentGameState = gameState::Options;
                    exitChangeNickname = true;
                }
            }
        }

       nicknameText.setString(displayedText+newNickname);
        SpaceInvaders.gameWindow.clear();
        SpaceInvaders.gameWindow.draw(nicknameText);
        SpaceInvaders.gameWindow.display();
    }
    return;
}
```

Metoda ta pobiera od użytkownika nową nazwę użytkownika (gdzie za pomocą regexa jest sprawdzane, czy składa się tylko z liter i cyfr) i następnie ustawia ją jako nowy „playerNickname” w danej rozgrywce.

```
void menu::gameOver(game& SpaceInvaders, bool playerWon) {
```

```
    sf::Font font;
    if (!font.loadFromFile("NovaSlim-Regular.ttf")) {
        std::cout << "Failed to load font file." << std::endl;
        return;
    }

    sf::Text gameOverText;
    gameOverText.setFont(font);
    gameOverText.setCharacterSize(36);
    gameOverText.setFillColor(sf::Color::White);
    gameOverText.setPosition(100, 100);

    sf::Color backgroundColor;
    std::string gameOverMessage;

    if (playerWon) {
        backgroundColor = sf::Color::Green;
        gameOverMessage = "Congratulations! You Won!\nPress ESC to exit\nto main menu.";
    }
    else {
        backgroundColor = sf::Color::Red;
        gameOverMessage = "Game Over! You Lost.\nPress ESC to exit\nto main menu.";
    }

    sf::RectangleShape backgroundRect(sf::Vector2f(512, 512));
    backgroundRect.setFillColor(backgroundColor);

    gameOverText.setString(gameOverMessage);
    gameOverText.setPosition(512 / 2.0f - gameOverText.getGlobalBounds().width / 2.0f, 256);

    while (SpaceInvaders.gameWindow.isOpen()) {
        sf::Event event;
        while (SpaceInvaders.gameWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                SpaceInvaders.gameWindow.close();
            }
            else if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Escape) {

SpaceInvaders.mainMenu.highScoresInMenu.addScore(SpaceInvaders.playerNickname,
SpaceInvaders.score);

SpaceInvaders.mainMenu.highScoresInMenu.saveScores(".\\scores\\high_scores.txt");
                    SpaceInvaders.currentGameState = gameState::MainMenu;
                    SpaceInvaders.playerOfGame.isDead = false;
                    SpaceInvaders.playerOfGame.health =
SpaceInvaders.playerOfGame.getMaxHealth();
                    SpaceInvaders.playerOfGame.sprite.setPosition(256 -
SpaceInvaders.playerOfGame.sprite.getGlobalBounds().width, 512);
                    SpaceInvaders.playerOfGame.bullets.clear();
                    SpaceInvaders.level_.invaders.clear();
                    SpaceInvaders.level_.levelNumber = 1;
                    SpaceInvaders.level_.initLevel();
                    SpaceInvaders.score = 0;
                    SpaceInvaders.startGame();
                    return;
                }
            }
        }

        SpaceInvaders.gameWindow.clear();
        SpaceInvaders.gameWindow.draw(backgroundRect);
        SpaceInvaders.gameWindow.draw(gameOverText);
        SpaceInvaders.gameWindow.display();
    }
}
```

Metoda ta obsługuje wykrywanie końca gry i wyświetlanie odpowiedniego komunikatu w oknie. Jeśli gracz nie żyje, to od razu pojawia się ekran końca gry z wiadomością, że przegrał. Jeśli gracz przeżył i ukończył odpowiednią liczbę poziomów, to wiadomość i wygląd komunikatu z końcem gry jest inna (wygrana). Dodatkowo wynik uzyskany w trakcie danej gry jest zapisywany.

```
void menu::manageMenus(game& SpaceInvaders) {
    while (SpaceInvaders.gameWindow.isOpen()) {
        SpaceInvaders.gameWindow.clear();
        switch (SpaceInvaders.currentGameState) {
        case gameState::MainMenu: {
            mainMenu(SpaceInvaders);
            break;
        }
        case gameState::Options: {
            items.clear();
            menuItems.clear();
            optionsMenu(SpaceInvaders);
            break;
        }
        case gameState::HighScores:
            displayHighScores(SpaceInvaders);
            break;
        case gameState::Playing:
            SpaceInvaders.mainLoop();
            break;
        case gameState::Exit:
            SpaceInvaders.gameWindow.close();
            break;
        default:
            break;
        }
        SpaceInvaders.gameWindow.display();
    }
}
```

Metoda ta zarządza aktualnie wyświetlanymi rzeczami w oknie. Używa ona zadeklarowanego w klasie game obiektu typu gameState, która jest zadeklarowana w module gameState.ixx:

```
export module gameState;
export enum class gameState {
    MainMenu,
    OptionsMenu,
    Playing,
    HighScores,
    Exit,
    ChangeNickname,
    Options,
    GameOver
};
```

Klasa game

```
class game {
    invader invaderInGame;
    std::map<int, level> mapOfLevels;
    sf::Texture background;
    sf::Sprite backgroundSprite;
    std::vector<bullet> invaderBullets;
    sf::Clock clock;
    bool invaderShooting;
    sf::VideoMode desktop;
    sf::Vector2u windowSize;
    sf::Vector2u textureSize;
    std::future<void> invaderMovementFuture;
    sf::RectangleShape healthBar;
    sf::RectangleShape healthBarBackground;
    bool playerWon;
public:
    player playerOfGame;
    menu mainMenu;
    menu pauseMenu;
    level level_;
    int score;
```

```
    sf::RenderWindow gameWindow;
    sf::Event event;
    std::string playerNickname;
    bool isAnyKeyPressed();
    void playerBorders();
    void mainLoop();
    void startGame();
    void hurtingInvaders(sf::Time&);
    void invadersShooting(sf::Time&);
    void playerMoving();
    void updateInvadersMovement(float);
    gameState currentGameState;
    void updateHealthBar();
    }
```

Istotne metody:

```
void game::updateHealthBar() {
    float healthPercent = static_cast<float>
(playerOfGame.getHealth())/playerOfGame.getMaxHealth();
    playerOfGame.manager.playerHealthBar.setSize(sf::Vector2f(300.f * healthPercent,
playerOfGame.manager.playerHealthBar.getSize().y));
}
```

Metoda ta zarządza paskiem zdrowia widzianego w lewym górnym kącie okna w trakcie trwania rozgrywki. Gdy gracz traci zdrowie, jeden z prostokątów (wierzchni) zostaje zmodyfikowany tak, aby pokazywał aktualny poziom zdrowia gracza.

```
void game::playerBorders() {
    if (playerOfGame.sprite.getPosition().x < 0)
    {
        playerOfGame.sprite.setPosition(0, playerOfGame.sprite.getPosition().y);
        playerOfGame.velocity = { 0.f, 0.f };
        playerOfGame.acceleration = { 0.f, 0.f };
    }
    else if (playerOfGame.sprite.getPosition().x > backgroundSprite.getGlobalBounds().width -
playerOfGame.sprite.getGlobalBounds().width)
    {
        playerOfGame.sprite.setPosition(backgroundSprite.getGlobalBounds().width -
playerOfGame.sprite.getGlobalBounds().width, playerOfGame.sprite.getPosition().y);
        playerOfGame.velocity = { 0.f, 0.f };
        playerOfGame.acceleration = { 0.f, 0.f };
    }
    if (playerOfGame.sprite.getPosition().y < 0)
    {
        playerOfGame.sprite.setPosition(playerOfGame.sprite.getPosition().x, 0);
        playerOfGame.velocity = { 0.f, 0.f };
        playerOfGame.acceleration = { 0.f, 0.f };
    }
    else if (playerOfGame.sprite.getPosition().y > backgroundSprite.getGlobalBounds().height -
playerOfGame.sprite.getGlobalBounds().height)
    {
        playerOfGame.sprite.setPosition(playerOfGame.sprite.getPosition().x,
backgroundSprite.getGlobalBounds().height - playerOfGame.sprite.getGlobalBounds().height);
        playerOfGame.velocity = { 0.f, 0.f };
        playerOfGame.acceleration = { 0.f, 0.f };
    }
    return;
}
```

Metoda sprawdzająca, czy gracz nie znajduje się na brzegu tła (będącego tego samego rozmiaru, co okno). Jeśli tak jest, prędkość gracza zostaje wyzerowana.

```
void game::hurtingInvaders(sf::Time& time) {
    playerOfGame.fireBullet(time);
    for (auto it = playerOfGame.bullets.begin(); it != playerOfGame.bullets.end();) {
        bool hitInvader = false;
        for (auto& el : level_.invaders) {
```

```
            if (it->sprite.getGlobalBounds().intersects(el.sprite.getGlobalBounds()) &&
!el.isDead) {
                el.health -= 20;
                hitInvader = true;
                it = playerOfGame.bullets.erase(it);
                score += 25;
                break;
            }
        }
        if (!hitInvader) {
            ++it;
        }
    }
}
```

Metoda ta sprawdza, czy którykolwiek z pocisków gracza trafił w jednego z wrogów na danym poziomie. Gdy pocisk przecina się ze sprite'm wroga, zostaje mu odjęta pewna liczba zdrowia, a pocisk wystrzelony przez gracza zostaje usunięty. Dodatkowo zostają tutaj przyznane punkty za trafienie wroga.

```
void game::invadersShooting(sf::Time& time) {
    if (level_.invaders.empty()) {
        return;
    }
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> invaderIndex(0, level_.invaders.size() - 1);
    int selectedInvaderIndex = invaderIndex(gen);
    bool hitPlayer = false;
    if (!level_.invaders[selectedInvaderIndex].isDead &&
level_.invaders[selectedInvaderIndex].shotClock.getElapsedTime() >=
level_.invaders[selectedInvaderIndex].shotInterval) {
        level_.invaders[selectedInvaderIndex].shotClock.restart();
        std::uniform_int_distribution<> shootChance(0, 1);
        if (shootChance(gen) == 0) {
            bullet newBullet = level_.invaders[selectedInvaderIndex].entityBullet;
            newBullet.sprite.setPosition(
                level_.invaders[selectedInvaderIndex].sprite.getPosition().x +
level_.invaders[selectedInvaderIndex].sprite.getGlobalBounds().width / 2 -
newBullet.sprite.getGlobalBounds().width / 2,
                level_.invaders[selectedInvaderIndex].sprite.getPosition().y +
level_.invaders[selectedInvaderIndex].sprite.getGlobalBounds().height
            );

            invaderBullets.push_back(newBullet);
        }
    }
    auto visibleBullets = invaderBullets | std::views::filter([](const bullet& b) {
        return b.sprite.getPosition().y <= 512;
        });
    for (auto& b : visibleBullets) {
        b.sprite.move(b.velocity);
        b.bulletAnimation(time.asSeconds());
    }

    for (auto it = invaderBullets.begin(); it != invaderBullets.end();) {
        it->sprite.move(it->velocity);
        it->bulletAnimation(time.asSeconds());

        if (it->sprite.getGlobalBounds().intersects(playerOfGame.sprite.getGlobalBounds())) {
            playerOfGame.health -= 20;
            it = invaderBullets.erase(it);
        }
        else {
            ++it;
        }
    }
    level_.invaders[selectedInvaderIndex].shotInterval =
level_.invaders[selectedInvaderIndex].shootingIntervalGenerator(sf::seconds(1),
sf::seconds(10));
}
```

Metoda zajmująca się strzelaniem przez wrogów pocisków. Przyznawana jest tutaj losowo możliwość wystrzelenia pocisku przez danego wroga, a następnie generowany jest pocisk z odpowiednim położeniem i prędkością oraz uruchamiana jest animacja pocisku. Za pomocą widoku visibleBullets wszystkie pociski są filtrowane pod kątem wylatywania poza ekran – takie pociski usuwane są z wektora pocisków wrogów. W metodzie tej sprawdzane jest także, czy gracz został trafiony przez pocisk wroga i odejmowane są w takiej sytuacji punkty zdrowia.

```
void game::updateInvadersMovement(float deltaTime) {

    bool changeDirection = false;

    for (auto& invader : level_.invaders) {
        if (invader.moveLeft) {
            invader.sprite.move(-invader.velocityX * deltaTime, 0.f);
            if (invader.sprite.getPosition().x < 0) {
                changeDirection = true;
            }
        }
        else {
            invader.sprite.move(invader.velocityX * deltaTime, 0.f);
            if (invader.sprite.getPosition().x + invader.sprite.getGlobalBounds().width >
backgroundSprite.getGlobalBounds().width) {
                changeDirection = true;
            }
        }
    }

    if (changeDirection) {
        for (auto& invader : level_.invaders) {
            invader.moveLeft = !invader.moveLeft;

            if (invader.moveLeft) {
                invader.sprite.move(0.1f, invader.sprite.getGlobalBounds().height);
            }
            else if(!invader.isDead){
                invader.sprite.move(0.1f, invader.sprite.getGlobalBounds().height);
            }
            if (invader.sprite.getPosition().y + invader.sprite.getGlobalBounds().height >=
backgroundSprite.getGlobalBounds().height) {
                playerWon = false;
                mainMenu.gameOver(*this, playerWon);
            }
        }
    }
}
```

Metoda ta porusza sprite’ami wrogów po ekranie (prawo-lewo) oraz przenosi ich o pewną odległość w dół, gdy są na krawędzi ekranu, zaraz przed zmianą kierunku lotu. Dodatkowo, jeśli wrogowie znajdą się na dole ekranu, gra kończy się przegraną.

```
void game::mainLoop() {
    while (gameWindow.isOpen()) {
        while (gameWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                gameWindow.close();
            }
        }
        sf::Time time = clock.restart();
        invadersShooting(time);
        playerOfGame.moveLeft();
        playerOfGame.moveRight();
        playerOfGame.moveUp();
        playerOfGame.moveDown();
        updateHealthBar();
```

```
        playerBorders();
        hurtingInvaders(time);
        updateInvadersMovement(time.asSeconds());
        playerOfGame.sprite.move(playerOfGame.velocity * playerOfGame.speedMultiplier);
        gameWindow.draw(backgroundSprite);

        if (!level_.levelFinished && std::all_of(level_.invaders.begin(), level_.invaders.end(),
[](const invader& inv) { return inv.isDead; })) {
            level_.levelFinished = true;
            if (level_.levelNumber == 2 && level_.levelFinished && !playerOfGame.isDead) {
                playerWon = true;
                mainMenu.gameOver(*this, playerWon);
            }
            level_.levelNumber++;
            std::cout << level_.levelNumber << std::endl;
            level_.initLevel();
        }
        for (auto& el : level_.invaders) {
            if (el.health < 1) {
                el.isDead = true;
                playerWon = false;
            }
            else {
                gameWindow.draw(el.sprite);
            }

        }
        auto visibleBullets = invaderBullets | std::views::filter([](const bullet& b) {
            return b.sprite.getPosition().y <= 512;
            });
        for (const auto& bullet : playerOfGame.bullets) {
            gameWindow.draw(bullet.sprite);

        }
        for (const auto& bullet : visibleBullets) {
            gameWindow.draw(bullet.sprite);
        }
        if (playerOfGame.health <= 0 && !playerOfGame.isDead) {
            playerOfGame.isDead = true;
            mainMenu.gameOver(*this, playerWon);
        }

        gameWindow.draw(playerOfGame.sprite);
        gameWindow.draw(playerOfGame.manager.playerHealthBarBack);
        gameWindow.draw(playerOfGame.manager.playerHealthBar);
        gameWindow.display();
    }

}
```

Metoda głównej pętli gry, gdzie „wrzucone” są wszystkie metody potrzebne do działania gry wraz z wyświetlaniem wszystkiego w oknie. Oprócz tego sprawdzane jest tutaj ukończenie poziomu (i przełączenie na kolejny) oraz czy gracz ukończył pomyślnie grę. Są także odbierane graczowi punkty zdrowia w przypadku bycia trafionym przez pocisk wroga.

Klasa bullet

```
class bullet {
public:
        int x;
        int y;
        sf::Vector2f velocity{ 0.0f, 0.0f };
        sf::Vector2f acceleration{ 0.01f, 0.01f };
        int damage;
        sf::Sprite sprite;
        sf::Texture texture;
        float bulletCooldown = 0.2f;
        float elapsedTime = 0.f;
        float speedMultiplier = 0.1f;
```

```
        int frameWidth = 32;
        int frameHeight = 32;
        int numFrames = 4;
        int currentFrame = 0;
        float frameDuration = 0.1f;
        float frameTimer = 0.0f;
        float animationTime;
        void bulletAnimation(float);

};
```

Klasa reprezentująca pocisk, używana przez klasę entity.
Istotna metoda:

```
void bullet::bulletAnimation(float deltaTime) {
        animationTime += deltaTime;
        if (animationTime >= frameDuration) {
                animationTime = 0.f;
                currentFrame = (currentFrame + 1) % numFrames;
                sf::IntRect newFrame(currentFrame * frameWidth, 0, frameWidth, frameHeight);
                sprite.setTextureRect(newFrame);
        }
}
```

Metoda ta uruchamia animację danego pocisku. Tekstura pocisku to 4-klatkowa animacja, która jest dzielona na pojedyncze klatki, zarządzane przez tę metodę.

Klasa highScores

```
class highScores {
public:
    std::unordered_multimap<std::string, int> listOfHighScores;
    void loadScores(const std::string& filePath);
    void saveScores(const std::string& filePath) const;
    void addScore(const std::string& playerNickname, int score);
    std::vector<std::pair<std::string, int>> getTopScores(size_t count) const;
};
```

Klasa zarządzająca najwyższymi wynikami przechowywanymi w kontenerze unoredered_multimap.
Istotne metody:

```
void highScores::saveScores(const std::string& filePath) const {
    std::filesystem::path savePath = std::filesystem::path(filePath).parent_path();
    if (!std::filesystem::exists(savePath)) {
        std::filesystem::create_directory(savePath);
    }

    std::ofstream file(filePath);
    if (!file.is_open()) {
        std::cout << "Failed to open high scores file for saving: " << filePath << std::endl;
        return;
    }

    for (const auto& entry : listOfHighScores) {
        file << entry.first << " " << entry.second << std::endl;
    }

    file.close();
    std::cout << "Saved high scores to file: " << filePath << std::endl;
}
```

Metoda ta zapisuje wyniki do pliku tekstowego (pseudonim gracza wraz z uzyskanym wynikiem). Za pomocą biblioteki filesystem tworzony jest folder przechowywujący ten plik tekstowy.

```
std::vector<std::pair<std::string, int>> highScores::getTopScores(size_t count) const {
```

```
    std::vector<std::pair<std::string, int>> scores(listOfHighScores.begin(),
listOfHighScores.end());

    std::sort(scores.begin(), scores.end(), [](const auto& a, const auto& b) {
        return b.second < a.second;
        });

    if (scores.size() > count) {
        scores.resize(count);
    }

    return scores;
}
```

Metoda ta filtruje wyniki tak, żeby uzyskać określoną przez zmienną count ilość najwyższych wyników.

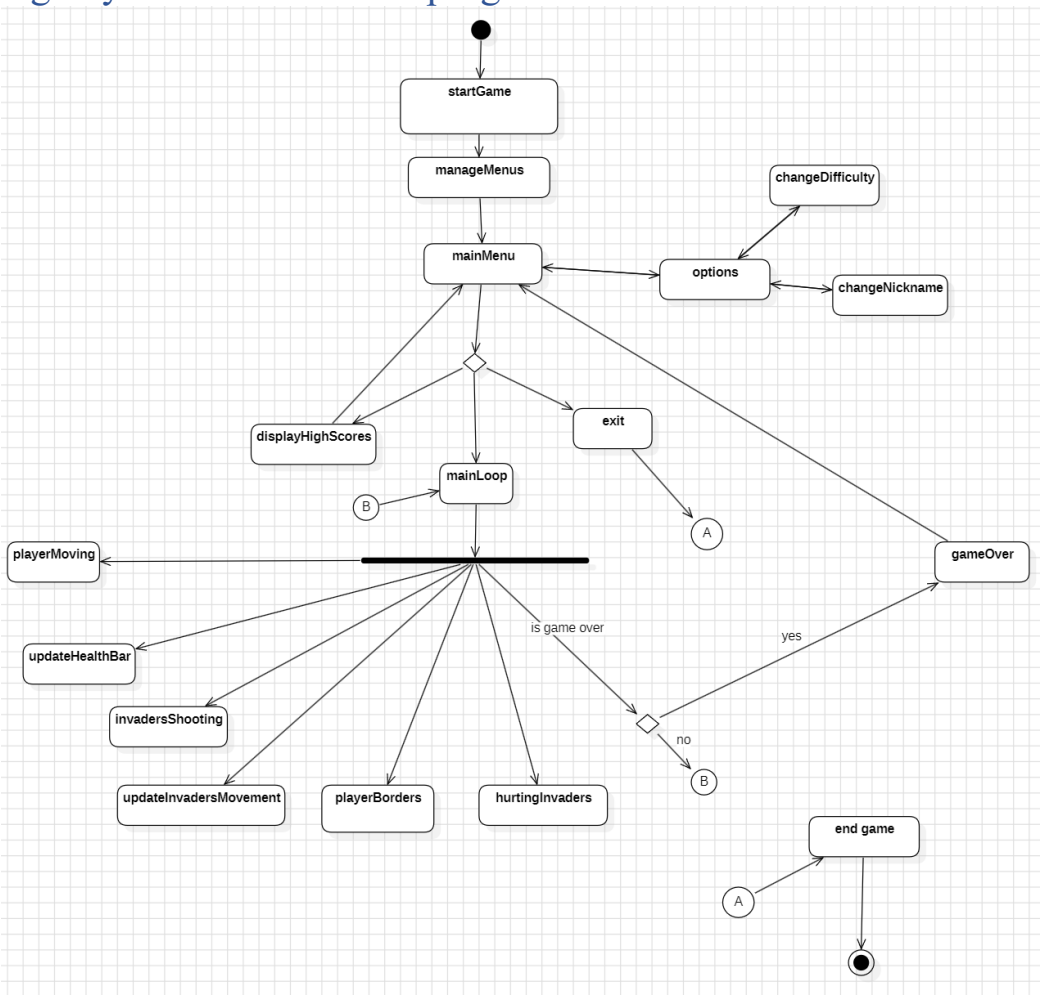
Wykorzystane techniki obiektowe

- polimorfizm – utworzyłam ogólną klasę entity, która ma metodę wirtualną fireBullet, która jest wykorzystywana i nadpisywana przez klasy player i invader.
- dziedziczenie – klasa entity jest dziedziczona przez klasy player i invader
- enkapsulacja – metody i zmienne w klasach są odpowiednio deklarowane jako prywatne lub publiczne, np. w klasie menu.

Użyte techniki przerabiane na zajęciach

1) Użycie modułów zamiast #include – stworzyłam moduł gameState.ixx, gdzie zadeklarowana klasa gameState jest importowana w plikach menu.cpp oraz game.cpp. Klasa gameState jest używana w klasie menu do zarządzania wyświetlanymi menu i przełączaniem między nimi.
2) Wyrażenia regularne <regex> - za pomocą wyrażenia regularnego ^[a-zA-Z0-9]+$ w metodzie menu::changeNickname() sprawdzana jest poprawność pseudonimu ustawianego przez użytkownika (czy zawiera tylko litery i cyfry).
3) System plików <filesystem> - za pomocą systemu plików tworzony jest folder na najlepsze wyniki (jeśli już taki nie istnieje) w metodzie highScores::saveScores(const std::string&).
4) Biblioteka <ranges> - za pomocą widoku filtrowane są pociski wrogów tak, aby w wektorze pocisków były tylko te, które są widoczne w oknie.

Ogólny schemat działania programu



Testowanie i uruchamianie

Podczas wykonywania projektu pojawiły się błędy wymagające bardziej szczegółowych testów z pomocą m.in. napisania komunikatów do konsoli w odpowiednich miejscach w kodzie, np.:

1) Będąc w menu opcji, jeśli wybrało się zmianę pseudonimu i wcisnęło ENTER, na chwilę migały opcje z menu opcji, po czym okno z powrotem pokazywało zmianę pseudonimu. Po dodaniu komunikatów w odpowiednich miejscach kodu i przeanalizowaniu działania menu z różnymi przyciskami na klawiaturze okazało się, że najpierw wciśnięty ENTER zatwierdzał zmianę pseudonimu i wracał do menu opcji, a następnie puszczony klawisz zatwierdzał wybranie elementu, na którym był indeks menu (czyli zmiana pseudonimu). Zostało to rozwiązane dodatkowymi warunkami i zmienną exitChangeNickname.
2) Na wczesnym etapie tworzenia gry statek poruszany przez gracza miał pewien współczynnik tarcia, dzięki czemu statek hamował po puszczeniu klawisza strzałki w danym kierunku. Przez przypadek odkryłam jednak pewien błąd związany z moją implementacją: hamowanie trwało

w nieskończoność, przez co wartość wektora prędkości statku się cały czas malała/rosła. Objawiało się to tym, że po zatrzymaniu dużo trudniej można było ruszyć w tym samym kierunku, a dalsze testy z komunikatami potwierdziły, że była to wina pętli z implementowaniem hamowania. Udało się to rozwiązać wprowadzając zegar noMovementClock, który sprawiał, że po jakimś czasie nieruszania się, pętla z hamowaniem przestawała być wykonywana. Ostatecznie hamowanie zostało usunięte ze względu na dużo trudniejsze poruszanie się statkiem.

3) Jednym z błędów menu było zamrażanie się całego programu, gdy wracało się z menu opcji lub wysokich wyników do głównego menu. Po licznych testach osobno dla tych dwóch menu, okazało się, ze problem polegał na braku zarządzania nad stanami gry, co w pętli sprawiało, że gra nie dostawała informacji na temat tego, co wyświetlać, ostatecznie nic nie wyświetlając i zamrażając się. Dzięki klasie gameState i metodzie manageMenus, problem ten został rozwiązany i można teraz swobodnie poruszać się pomiędzy różnymi menu.

Uwagi i wnioski

Różnice między wstępnymi założeniami a ostateczną implementacją

Nie wszystkie elementy z opisu i interfejsu wstępnego zostały zaimplementowane w ostatecznej wersji projektu. Nie rozbiłam różnych menu (options, gameOverScreen itd.) na osobne klasy, ponieważ uznałam, że rzeczy te nie są wystarczająco dużymi konceptami na to. Power-up jako koncept został odrzucony po obejrzeniu rozgrywki z klasycznej wersji gry, gdzie takowych nie było. Klasa levelMaker nie została zaimplementowana, ponieważ jej przeznaczenie zostało zawarte w metodach klasy level. Klasa healthBar nie została zaimplementowana, ponieważ ostatecznie healthBar został włączony w klasę game, gdzie są zmienne i metody zajmujące się paskiem zdrowia. Poza tym, ze względu na brak power-upów, klasa playerManager ma zdecydowanie mniej metod.
Osadzenie gry w uniwersum „Diuny” okazało się błędem, ponieważ robienie własnych tekstur do gry było zbyt wielkim wyzwaniem. Postanowiłam uogólnić grę i dodać do niej grafiki pobrane ze strony itch.io.

Nierozwiązane błędy

Pomimo licznych testowań, niektóre błędy zostały zauważone przeze mnie zbyt późno. Jeśli zmienimy w menu opcji i poziom trudności, i pseudonim gracza, po uruchomieniu rozgrywki nie są widoczni wrogowie (mimo, że powinni istnieć według komunikatów w konsoli). Dodatkowo, pomimo ustawiania go jak pozostałe poziomy, poziom 3 nie działał prawidłowo (wrogowie lecieli w lewą stronę i nieskończenie długo przesuwali się w dół okna natychmiast po zetknięciu się z lewą krawędzią okna). Ostatni z nierozwiązanych błędów jest związany z końcem gry – pomimo spełniania warunków zadeklarowanych w metodzie

klasy game, gdy gracz przechodzi wszystkie poziomy i żyje, nie pojawia się komunikat o wygranej grze. Rozwiązanie tych błędów będzie częścią dalszego rozwoju projektu.

Dalsze możliwości rozwoju projektu i wnioski

Projekt ten był bardzo rozwijający pod względem obsługiwania biblioteki graficznej SFML oraz ogólnego konceptu tworzenia gier. Mogłam także zobaczyć, jak w praktyce można implementować techniki poznane na zajęciach laboratoryjnych w języku C++.
Dalszy rozwój projektu, już na własną rękę, polegałby na stworzeniu nieskończonego trybu gry, gdzie poziomy generowane byłyby na bieżąco, a nie statycznie ustawiane w metodzie. Oprócz tego dodałabym ściankę, którą wrogowie mogą podniszczać, a gracz może używać do ukrywania się. Poza tym cała gra byłaby doszlifowana pod kątem wcześniej wspomnianych błędów i zoptymalizowana tak, aby działała w taki sam sposób na każdym urządzeniu. Dodatkowo, jedną z drobniejszych zmian, byłoby dodanie muzyki i dźwięków, które w trakcie robienia projektu wydawały się mało istotne, ale teraz uważam, że znacznie polepszyłyby odbiór gry. Rozpatrując sam kod, oczyściłabym go z zapomnianych i nieużywanych zmiennych i metod, a niektóre metody przeniosłabym do klas, których bardziej dotyczą.