Imię i nazwisko: Jakub Pachut Piotr Patla Mateusz Nosal Wiktor Nowak	**Kierunek:** ITE	**Rok studiów i grupa:** 1/zima grupa 3
Data zajęć: 12.01.2025	**Nazwa projektu:** Feline slayer	

Cel i opis projektu

Celem tego projektu było przećwiczenie i utrwalenie poznanych technik programistycznych. Podczas tworzenia tej gry nauczyliśmy się dużo nowych rzeczy m.in. podstaw programowania obiektowego w C++, pracy zespołowej oraz obsługi kontroli wersji w projekcie, w który zaangażowany jest więcej niż jedna osoba.

Nasza gra jest Top-Down wave survival shooterem w motywie pixel-art. Chodzimy postacią po dużej mapie i staramy się przetrwać coraz to większe fale przeciwników. Wraz z pokonywaniem przeciwników nasza postać leveluje i dzięki temu możemy wybrać jedno z trzech ulepszeń.
Ulepszenia do wyboru:

- Pełna regeneracja zdrowia
- Plus 10 punktów szybkości
- Plus 25 punktów do maksymalnego zdrowia

Gra posiada system zapisu i wczytania rozgrywki oraz minimalistyczne menu główne. Do gry dodaliśmy również Easter eggi w postaci cheat code z Gta San Andreas.

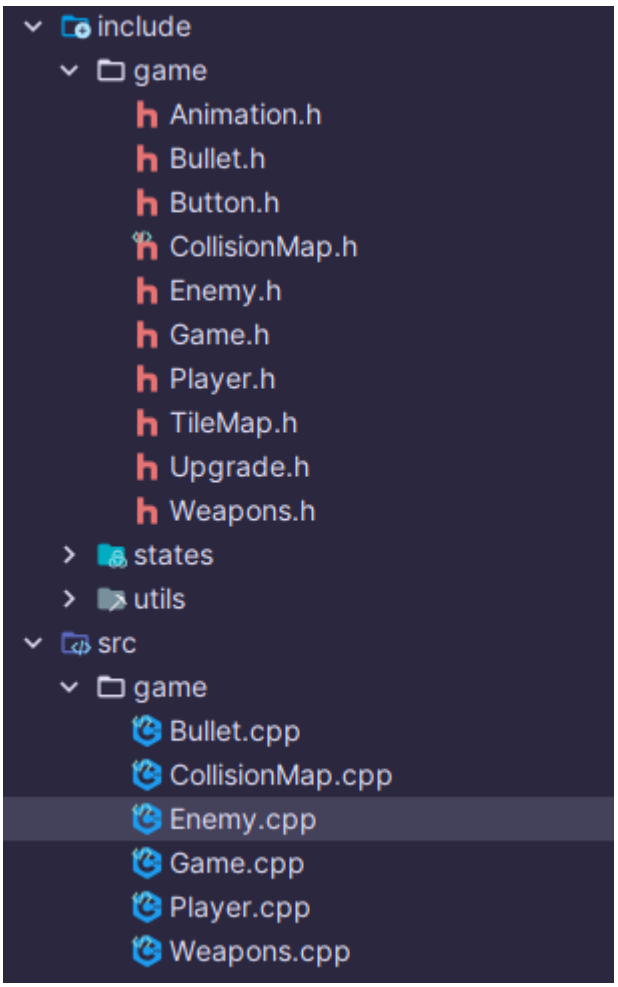
Zastosowane techniki programistyczne

Projekt został zrealizowany w paradygmacie programowania obiektowego. Logika gry została podzielona na klasy odpowiedzialne za konkretne elementy, takie jak Player, Enemy, Game, Bullet itp.

Zastosowaliśmy enkapsulację danych w niektórych klasach. Podzieliliśmy logikę gry na mniejsze moduły, które później wprowadzaliśmy do głównej klasy Game. Zastosowaliśmy również strukturę obiektu transferu danych (DTO), który przechowuje typ ulepszenia oraz jego wartość.

Cała projekt buildowaliśmy dzięki CMake, poniżej nasza konfiguracja:

```
cmake_minimum_required(VERSION 3.28)
project(Game_PI LANGUAGES CXX)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
include(FetchContent)
FetchContent_Declare(
        SFML
        GIT_REPOSITORY https://github.com/SFML/SFML.git
        GIT_TAG 3.0.2
        GIT_SHALLOW ON
        EXCLUDE_FROM_ALL
)
FetchContent_Declare(
   nlohmann_json
   GIT_REPOSITORY https://github.com/nlohmann/json.git
   GIT_TAG v3.12.0
)
FetchContent_MakeAvailable(nlohmann_json)
set(SFML_BUILD_EXAMPLES OFF CACHE BOOL "" FORCE)
set(BUILD_SHARED_LIBS OFF CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(SFML)
file(GLOB_RECURSE SOURCES CONFIGURE_DEPENDS "src/*.cpp")
add_executable(Game_PI ${SOURCES}
        src/game/CollisionMap.cpp
        include/game/CollisionMap.h)
target_include_directories(Game_PI PRIVATE include)
target_link_libraries(Game_PI PRIVATE sfml-graphics sfml-window
sfml-system)
target_link_libraries(Game_PI PRIVATE
nlohmann_json::nlohmann_json)
```



Wykorzystane biblioteki

Cała gra została napisana w języku **C++** wraz z biblioteką **SFML**, która służyła do wyświetlania grafiki, wczytywania okna, obsługi klawiatury i myszy oraz wczytywania tekstur i tworzenia spritów.

Dodatkowo korzystaliśmy z biblioteki **nlohmann_json,** dzięki której mogliśmy łatwo pracować z plikami json (między innymi do rysowania mapy).

Wybrane funkcjonalności

Jakub Pachut

Przy projekcie byłem odpowiedzialny za mechaniki przeciwników oraz system levelowania postaci.

Przykładowe funkcjonalności:

Poruszanie się przeciwników w stronę gracza jest to podstawowa i najważniejsza mechanika przeciwnika

Kod:

```
void Enemy::updatePosition(float dt, sf::Vector2f playerPosition) {
   sf::Vector2f playerVectorDirection = playerPosition -
this->position;

   float length = playerVectorDirection.length();
   if (length != 0.f) playerVectorDirection /= length;

   float radians = std::atan2(playerVectorDirection.y,
playerVectorDirection.x);
   float degrees = radians * 180.f / PI;

   this->velocity.x = playerVectorDirection.x * this->speed;
   this->velocity.y = playerVectorDirection.y * this->speed;
   this->position.x += velocity.x * dt;
   this->position.y += velocity.y * dt;

   if (degrees >= -45 && degrees <= 45)
       direction = EnemyDirection::Right;
   else if (degrees >= 45 && degrees < 135)
       direction = EnemyDirection::Down;
   else if (degrees < -45 && degrees >= -135)
       direction = EnemyDirection::Up;
   else
       direction = EnemyDirection::Left;

   this->sprite.setPosition(this->position);
   sf::FloatRect bounds = this->sprite.getLocalBounds();
   this->sprite.setOrigin(sf::Vector2f(bounds.size.x / 2.f,
bounds.size.y / 2.f));
   this->sprite.setScale(sf::Vector2f(this->scale, this->scale));
}
```

Wyjaśnienie:
W metodzie przyjmujemy argumenty dt (delta time) i playerPosition. Na bazie pozycji gracza liczymy wektor od przeciwnika do gracza. Normalizujemy go, żeby przeciwnik się nie teleportował, tylko był skierowany w stronę gracza. Następnie obliczamy kąt, pod jakim jest ustawiony przeciwnik i dzięki temu możemy stwierdzi, w którym kierunku patrzy i zmienić teksturę. Na koniec ustawiamy nową pozycję.

System levelowania postaci jest kolejną ważną funkcjonalnością w grze wymagał on pracy z kilkoma klasami. Za każdy nowy level możemy wybrać jedno z trzech ulepszeń. Levelujemy poprzez walkę z przeciwnikami. Za każdym razem kiedy dostajemy nowy level następny wymaga większej ilości XP.

```
bool Player::isLvlUp() const {
    return this->currentXp >= this->nextLvlCap;
}
```

Tutaj sprawdzamy czy przeciwnik żyje jeśli nie to dodajemy XP graczowi usuwamy przeciwnika z tablicy i sprawdzamy czy gracz leveluje:

```
for (int i = 0 ; i < this->enemies.size(); i++) {
    if (!enemies[i]->is_alive()) {
        player.currentXp += enemies[i]->xp;
        enemies.erase(enemies.begin() + i);
        this->currentEnemies--;
        if (player.isLvlUp()) {
```

```
            player.lvlUp();
            this->isLvlUp = {true};
            this->maxEnemies += 20;
            upgradeState.rollUpgrades();
        } else {
            this->isLvlUp = {false};
        }
    }
}
```

Struktura Upgrade która zawiera typ ulepszenia oraz jego wartość:

```
#ifndef GAME_PI_UPGRADE_H
#define GAME_PI_UPGRADE_H

enum class UpgradeType {
    Speed,
    MaxHp,
    Heal,
};

struct Upgrade {
    UpgradeType type;
    float value;
};

#endif //GAME_PI_UPGRADE_H
```

Tutaj system losowania ulepszeń oraz wyswietlenie odpowiedniego napisu zaleznie od typu ulepszenia (pomijam rysowanie ramek itp)

```
void UpgradeState::updateTexts() {
        auto setupText = [](sf::Text& txt, const Upgrade& upg) {
            txt.setString(upgradeToString(upg));
            sf::FloatRect b = txt.getGlobalBounds();
            txt.setOrigin(sf::Vector2f(b.size.x / 2.f, b.size.y /
2.f));
            txt.setScale(sf::Vector2f(0.4f,0.4f));
        };

        setupText(upgradeText1, upgrades[0]);
        setupText(upgradeText2, upgrades[1]);
        setupText(upgradeText3, upgrades[2]);
}

Upgrade randomUpgrade() {
```

```
    int type = rand() % 3;

    switch (type) {
        case 0: return { UpgradeType::Speed, 10.f };
        case 0: return { UpgradeType::MaxHp, 25.f };
        case 0: return { UpgradeType::Heal, 0.f };
    }
    return { UpgradeType::Speed, 5.f };
}

void UpgradeState::rollUpgrades() {
    for (int i = 0; i < 3; ++i)
        upgrades[i] = randomUpgrade();

    updateTexts();
}
```

I w klasie Player zastosowanie ulepszenia poprzez switch upgrade.type:

```
void Player::applyUpgrade(const Upgrade& upgrade) {
    switch (upgrade.type) {
        case UpgradeType::Speed:
            this->speed += upgrade.value;
            break;
        case UpgradeType::MaxHp:
            this->maxHp += static_cast<int>(upgrade.value);
            this->hp += static_cast<int>(upgrade.value);
            break;
        case UpgradeType::Heal:
            this->hp = this->maxHp;
            break;
    }
}
```

W pętli gry sprawdzane jest czy występuje event lvlUp:

```
if (isLvlUp) {
    upgradeState.update(dt, player.position);

    UpgradeChoice choice = upgradeState.getSelected();
    if (choice != UpgradeChoice::None) {
        Upgrade upgrade = upgradeState.getUpgrade(choice);
        player.applyUpgrade(upgrade);

        isLvlUp = false;
        upgradeState.resetSelection();
    }
}
```

W poolEvents sprawdzane jest czy ulepszenie zostało wybrane i które:

```
else if (const auto* mouse =
event->getIf<sf::Event::MouseButtonPressed>()) {
    if (mouse->button == sf::Mouse::Button::Left && isLvlUp) {
        sf::Vector2f mousePos =

window->mapPixelToCoords(sf::Mouse::getPosition(*window));

        upgradeState.handleClick(mousePos);
    }
}
```

Pomocnicza klasa od wyświetlania odpowiedniego napisu:

```
inline std::string upgradeToString(const Upgrade& u) {
    switch (u.type) {
        case UpgradeType::Speed:
            return "Predkosc + " +
std::to_string(static_cast<int>(u.value));
        case UpgradeType::MaxHp:
            return "Max HP + " +
std::to_string(static_cast<int>(u.value));
        case UpgradeType::Heal:
            return "Ulecz do max HP";
    }
    return "";
}
```

Wiktor Nowak

Moim zadaniem było stworzenie klasy gracza oraz utworzenie podstawowych funkcjonalności takich jak obrót postaci, podstawowe poruszanie się i pasek zdrowia.

```
class Player {
    private:
    sf::RectangleShape hitBox;
    sf::Vector2f lastPosition; // przechowuje poprzednią pozycję gracza

    std::vector<Weapon> arsenal;
    int weapon_index = 0;
    std::shared_ptr<sf::Texture> guns_texture=
std::make_shared<sf::Texture>("../../assets/images/all_guns.png");;

    public:
    void applyUpgrade(const Upgrade& upgrade);
    sf::Texture walkTxt = sf::Texture("../../assets/images/walk.png");
    sf::Texture idleTxt = sf::Texture("../../assets/images/Idle.png");
    sf::Texture plateTxt =
sf::Texture("../../assets/images/border.png");
    sf::Sprite sprite = sf::Sprite(idleTxt);
    sf::Vector2f position {};
    sf::RectangleShape hpBar {};
    sf::RectangleShape bBar;
    sf::Sprite plate = sf::Sprite(plateTxt);
```

```
    int ad {};
    int armor {};
    int lvl {};
    float speed {};
    int nextLvlCap {};
    int currentXp {};
    WeaponType weapon {WeaponType::Gun1};
    //stats
    int maxHp=100;
    int hp=maxHp;
```

W celu zanimowania postaci utworzyłem klasę Animation, która obsługuje tzw. Sprite sheety, czyli zbiory tekstur. W konstruktorze można ustawić parametry takie jak, ilość klatek, czy też odstęp czasowy pomiędzy wyświetlaniem kolejnych klatek animacji.

```
    class Animation {
public:
   int frameCount;
   int frame=0;
   int posX;
   int posY;
   int frameW;
   int frameH;
   float speed;
   sf::Sprite *sprite;
   sf::Clock time;
   bool pause=false;
```

Odpowiadałem również za stworzenie menu. W tym celu utworzyłem klasę Button, która reprezentuje przycisk.

```
class Button {
public:
   sf::RectangleShape rect;
   std::string label;
   sf::Font font =
sf::Font("../../assets/fonts/Cristone.ttf");
   sf::Text text = sf::Text(font);
   sf::Vector2f size;
   sf::Vector2f position;
   bool isClicked=false;
```

Dodałem możliwość zapisu stanu gry, wczytania go, opcje wyłączenia dźwięku i muzyki oraz sprawdzenie, czy punkty zdrowia gracza są równe zero i wyświetlenie odpowiedniego komunikatu jeżeli zachodzi taki warunek.

```
if (room==0) {
   ambient.stop();
   if (menuMusic.getStatus()!=sf::Music::Status::Playing &&
musicOn) menuMusic.play();
   this->window->setView(view);

titleSprite.setPosition(window->getView().getCenter()+sf::Vect
or2f{0,-60});

   loadB.update(*this->window);
   if (loadB.isClicked) {
       if (soundOn) clickSnd.play();
       loadSave();
   }

   newB.update(*this->window);
   if (newB.isClicked) {
       if (soundOn) clickSnd.play();
       isStopped=false;
       isGameOver=false;
       sf::Vector2f mapSize = ground.getSize();
       sf::Vector2f centerPos = { mapSize.x / 2.f, mapSize.y /
2.f };
       this->player.position = centerPos;
       this->player.sprite.setPosition(centerPos);
       player.currentXp=0;
       player.lvl=1;
       player.maxHp=100;
       player.hp=player.maxHp;
       player.ad=20;
       player.armor = {10};
       player.nextLvlCap = {30};
       player.speed = {70};
       player.switch_weapon(0);
       maxEnemies=10;
       currentEnemies=0;

       enemies.clear();
       bullets.clear();
```

```
        room=1;
    }

    settingsB.update(*this->window);
    if (settingsB.isClicked) {
        if (soundOn) clickSnd.play();
        Sleep(100);
        room=2;
    }

    exitB.update(*this->window);
    if (exitB.isClicked) {
        if (soundOn) clickSnd.play();
        this->window->close();
    }
}
```

```
gameOver();
menuB.update(*this->window);
if (menuB.isClicked) {
    if (soundOn) clickSnd.play();
    Sleep(2000);
    room=0;
}

loadB.update(*this->window);
if (loadB.isClicked) {
    if (soundOn) clickSnd.play();
    loadSave();
}
```

Byłem również odpowiedzialny za wybranie dźwięków i muzyki oraz napisanie kodu, który sprawi, że będą one odgrywane w trakcie gry.

Piotr Patla

```
bool loadFromJsonLayer(const std::filesystem::path& jsonPath,
                       const std::string& layerName,
                       const std::filesystem::path& tilesetPath)
    {
        std::ifstream file(jsonPath);
```

```
        if (!file.is_open())
            return false;

        nlohmann::json j;
        file >> j;

        const unsigned tileSize = j.at("tileSize").get<unsigned>();
        const unsigned width    = j.at("mapWidth").get<unsigned>();
        const unsigned height   = j.at("mapHeight").get<unsigned>();

        std::vector<int> tiles(width * height, -1);
```

Ten fragment odpowiada za wstępną konfigurację warstwy graficznej mapy na podstawie danych z pliku JSON. Po otwarciu pliku i sprasowaniu jego zawartości, kod pobiera kluczowe parametry: rozmiar kafelka oraz szerokość i wysokość mapy. Wartości te są niezbędne do przygotowania głównego kontenera danych, czyli wektora tiles. Rozmiar tego wektora jest definiowany jako iloczyn szerokości i wysokości mapy, co odpowiada całkowitej liczbie pól na planszy. Na koniec wektor zostaje wypełniony wartościami -1, co funkcjonuje jako domyślna inicjalizacja oznaczająca puste pole.

```
bool loadCollisionLayer(const std::filesystem::path& jsonPath, const
std::string& layerName) {
        std::ifstream file(jsonPath);
        if (!file.is_open())
            return false;
        nlohmann::json j;
        file >> j;

        unsigned tileSize = j.at("tileSize").get<unsigned>();
        unsigned width    = j.at("mapWidth").get<unsigned>();
        unsigned height   = j.at("mapHeight").get<unsigned>();

        float offsetX = -(static_cast<float>(width * tileSize) / 2.f);
        float offsetY = -(static_cast<float>(height * tileSize) / 2.f);
```

Fragment powyżej odpowiedzialny jest za rozpoczynanie działania od wczytania pliku pobrania wymiarów mapy, a jej cel jest ściśle związany z fizyką. Zamiast alokować pamięć na strukturę siatki, kod koncentruje się na obliczeniach potrzebnych do ustalenia pozycji mapy w grze. Wykorzystując rzutowanie na typ float, wyliczane są wartości przesunięcia offsetX . Obliczenie to polega na wyznaczeniu połowy rozmiaru mapy w pikselach i zanegowaniu wartości. Dzięki

temu warstwa kolizji zostanie wycentrowana względem początku układu współrzędnych, co ułatwia późniejsze pozycjonowanie obiektów.

```
using json = nlohmann::json;

bool CollisionMap::loadFromSpriteFusion(
    const std::filesystem::path& jsonPath,
    const std::set<std::string>& collidableLayers,
    const std::set<std::string>& overrideLayers)
{
    colliders.clear();

    std::ifstream file(jsonPath);
    if (!file.is_open())
        return false;

    nlohmann::json j;
    file >> j;

    const unsigned tileSize = j.at("tileSize").get<unsigned>();
    const unsigned width    = j.at("mapWidth").get<unsigned>();
    const unsigned height   = j.at("mapHeight").get<unsigned>();
```

Powyższy kod służy do inicjalizacji kolizji mapy na podstawie danych wyeksportowanych zSprite Fusion. Na początku wywołuje colliders.clear(), co usuwa wszelkie wcześniej przechowywane dane o kolizjach, zapewniając nowy stan przed wczytaniem nowego poziomu. Następnie następuje otwarcie pliku i odczytanie zawartości do obiektu JSON, z obsługą błędu w przypadku niepowodzenia otwarcia pliku. W końcowej widocznej części kodu pobierane są kluczowe dane mapy: rozmiar kafelka (tileSize) oraz szerokość i wysokość planszy, które są niezbędne do prawidłowego interpretowania współrzędnych fizycznych w dalszej części algorytmu.

```
void Game::updateBullets(float dt) {
   static float shoot_timer = 0.f;
   shoot_timer += dt;

   Weapon& current_weapon = this->player.get_current_weapon();

   if (sf::Keyboard::isKeyPressed(sf::Keyboard::Scancode::Space) && shoot_timer >
current_weapon.fire_rate) {
       shoot_timer = 0.f;

       // Mouse position
       sf::Vector2f mouse_position =
this->window->mapPixelToCoords(sf::Mouse::getPosition(*this->window));
       this->bullets.push_back(std::make_unique<Bullet>(
           this->player.position,
           mouse_position,
           this->bullet_texture
           ));
   }

   for (int i = 0; i < this->bullets.size(); i++) {
       this->bullets[i]->update(dt);

       if (this->bullets[i]->isDead()) {
           this->bullets.erase(this->bullets.begin() + i);
           i--;
           continue;
       }

       auto bullet_bounds = this->bullets[i]->getGlobalBounds();

       for (int j = 0; j < this->enemies.size(); j++) {
           if (bullet_bounds.findIntersection(this->enemies[j]->getBounds())) {
               this->enemies[j]->getAttack(current_weapon.damage);

               this->bullets.erase(this->bullets.begin() + i);
               i--;
               break;
           }
       }
   }
```

Ta funkcja zarządza logiką strzelania, sprawdzając licznik czasu i tworząc nowe pociski lecące w stronę kursora myszy. Odpowiada również za aktualizację istniejących pocisków oraz wykrywanie ich kolizji z przeciwnikami.

```
enum class WeaponType {
   Gun1,
   Gun2,
   Gun3,
```

```
    Gun4,
    Gun5,
    Gun6,
    Gun7,
    Gun8,
    Grenade
};

struct Weapon {
    std::string name;
    float fire_rate;
    int damage;
    float bullet_speed;
    WeaponType type;

    std::shared_ptr<sf::Texture> texture;
    sf::Sprite icon;

    Weapon(std::string n, float rate, int dmg, float speed,
WeaponType t,
        std::shared_ptr<sf::Texture> tex, sf::IntRect rect):
    name(n),
    fire_rate(rate),
    damage(dmg),
    bullet_speed(speed),
    type(t),
    texture(tex),
    icon(*tex) {
        this->icon.setTextureRect(rect);
        this->icon.setScale({1, 1});
    }

};
```

Tutaj definiuje strukturę, która łączy statystyki broni z grafiką (wycina fragment grafiki ze zdjęcia ze wszystkimi broniami)

```
this->guns_texture = std::make_shared<sf::Texture>();
this->guns_texture->loadFromFile("assets/images/all_guns.png");

this->arsenal.push_back(Weapon("Gun1", 0.5f, 40, 100.f, WeaponType::Gun1,
this->guns_texture, sf::IntRect({0, 10}, {32, 16})));
this->arsenal.push_back(Weapon("Gun2", 0.4f, 50, 600.f, WeaponType::Gun2,
this->guns_texture, sf::IntRect({30, 10}, {32, 16})));
this->arsenal.push_back(Weapon("Gun3", 1.5f, 110, 300.f, WeaponType::Gun3,
this->guns_texture, sf::IntRect({58, 10}, {32, 16})));
this->arsenal.push_back(Weapon("Gun4", 0.3f, 50, 600.f, WeaponType::Gun4,
this->guns_texture, sf::IntRect({86, 10}, {32, 16})));
```

```
this->arsenal.push_back(Weapon("Gun5", 1.3f, 100, 400.f, WeaponType::Gun5,
this->guns_texture, sf::IntRect({118, 12}, {32, 16})));
this->arsenal.push_back(Weapon("Gun6", 0.8f, 100, 250.f, WeaponType::Gun6,
this->guns_texture, sf::IntRect({0, 28}, {32, 16})));
```

W tym miejscu wczytuję zdjęcie (all_guns.png) i przypisuję statystyki broni oraz współrzędne ich miejsca na zdjęciu

```
this->window->draw(this->borderSprite);
sf::Sprite weaponIcon = this->player.get_current_weapon().icon;
sf::FloatRect border_boudns = this->borderSprite.getGlobalBounds();
sf::FloatRect weapon_bounds = weaponIcon.getGlobalBounds();

float x = border_boudns.position.x + (border_boudns.size.x / 2.f) -
(weapon_bounds.size.x / 2.f);
float y = border_boudns.position.y + (border_boudns.size.y / 2.f) -
(weapon_bounds.size.y / 2.f);


weaponIcon.setPosition({x, y});
this->window->draw(weaponIcon);
```

Rysuję ramkę oraz ikonę aktualnie wybranej broni oraz ustawiam broń na środku tej ramki.

```
Bullet::Bullet(sf::Vector2f start_position, sf::Vector2f target_position, const
sf::Texture& texture) : sprite(texture) {
   // this->shape.setRadius(5.f);
   // this->shape.setFillColor(sf::Color::White);
   //
   // this->shape.setOrigin({5.f, 5.f});
   // this->shape.setPosition(start_position);

   this->sprite.setTexture(texture);
   sf::Vector2u size = texture.getSize();
   this->sprite.setOrigin({static_cast<float>(size.x / 2.f), static_cast<float>(size.y /
2.f)});
   this->sprite.setPosition(start_position);
   this->sprite.setScale({0.5, 0.5});



   this->life_time = 0.f;
   this->max_life_time = 3.0f;
   float speed = 200.f;

   sf::Vector2f direction = target_position - start_position;

   float length = std::sqrt(direction.x * direction.x + direction.y * direction.y);

   if (length != 0) {
       direction /= length;
       float rotation = std::atan(direction.y / direction.x) * 180.f / 3.14159f;
       this->sprite.setRotation(sf::degrees(rotation));
   }

   this->velocity = direction * speed;
}
```

```cpp
void Bullet::update(float dt) {
    this->sprite.move(this->velocity * dt);
    this->life_time += dt;
}

void Bullet::render(sf::RenderTarget& target) {
    target.draw(this->sprite);
}

sf::FloatRect Bullet::getGlobalBounds() const {
    return this->sprite.getGlobalBounds();
}

bool Bullet::isDead() const {
    return this->life_time >= this->max_life_time;
}
```

Klasa bullet oblicza wektorowy kierunek i kąt obrotu tak, żeby pocisk leciał w kierunku myszy. Odpowiada również za przesuwanie pocisku w każdej klatce gry (funkcja update) oraz sprawdzanie czy jego czas życia dobiegł końca, aby go usunąć i zwolnić pamięć.

Link do repozytorium na GitHubie

https://github.com/Szafter12/PI-game

W repozytorium znajduje się plik README, w którym jest m.in. jak zbuildować i uruchomić grę lokalnie

Pozostałe linki

Assety/tekstury/sprity pochodzą ze strony https://itch.io

Linki do spirtesheetów:

https://cupnooble.itch.io/sprout-lands-asset-pack

https://sscary.itch.io/the-adventurer-female

Dźwięki pochodzą z gry Pokemon Red

Ambient z poziomu pochodzi z gry Lego Dimensions

Tytuł w menu został wykonany na stronie:
https://pl.textstudio.com/logo/tekstowe-kolorowe-litery-edytowalne-online-635

Muzyka do menu została wykonana na stronie:

https://www.beepbox.co