Sprawozdanie z projektu semestralnego

Przedmiot: Programowanie komputerów 4

Imię i nazwisko: Kamil Szymański

Grupa: 1

Sekcja: 2

Semestr: 4

Kierunek: Informatyka

Osoba prowadząca: dr inż. Anna Gorawska

1. Temat projektu

Tematem projektu wykonanego przeze mnie podczas 4 semestru zajęć z programowania komputerów, jest gra Metal Forest.

2. Analiza tematu

Metal Forest to gra z perspektywy „top view” (gra z widokiem na gracza i przeciwników z góry). Dużą inspiracją podczas tworzenia jej, była gra The Binding Of Isaac, stworzona przez Edmunda McMillena w 2011 roku. Została przeze mnie napisana w języku C++, z wykorzystaniem biblioteki SFML (simple and fast multimedia library). Uznałem to za najlepszy wybór, ze względu na moją obszerną wiedzę z programowania w tym języku, oraz prostotę implementacji nieskomplikowanych gier 2D, zapewnioną przez tę bibliotekę. Cała aplikacja jest uruchamiana w oknie i posiada rozbudowany interfejs graficzny. Implementacja poszczególnych składowych została zrealizowana w metodyce obiektowej, z wykorzystaniem klas. Ich podział ułatwił osiągnięcie większego porządku w kodzie, oraz podzielenie go na moduły, dzięki którym dodawanie nowych mechanik, przeciwników czy obiektów z którymi gracz wchodzi w interakcję, jest dużo łatwiejsze i nie wymaga rewolucyjnych zmian w implementacji całej aplikacji.

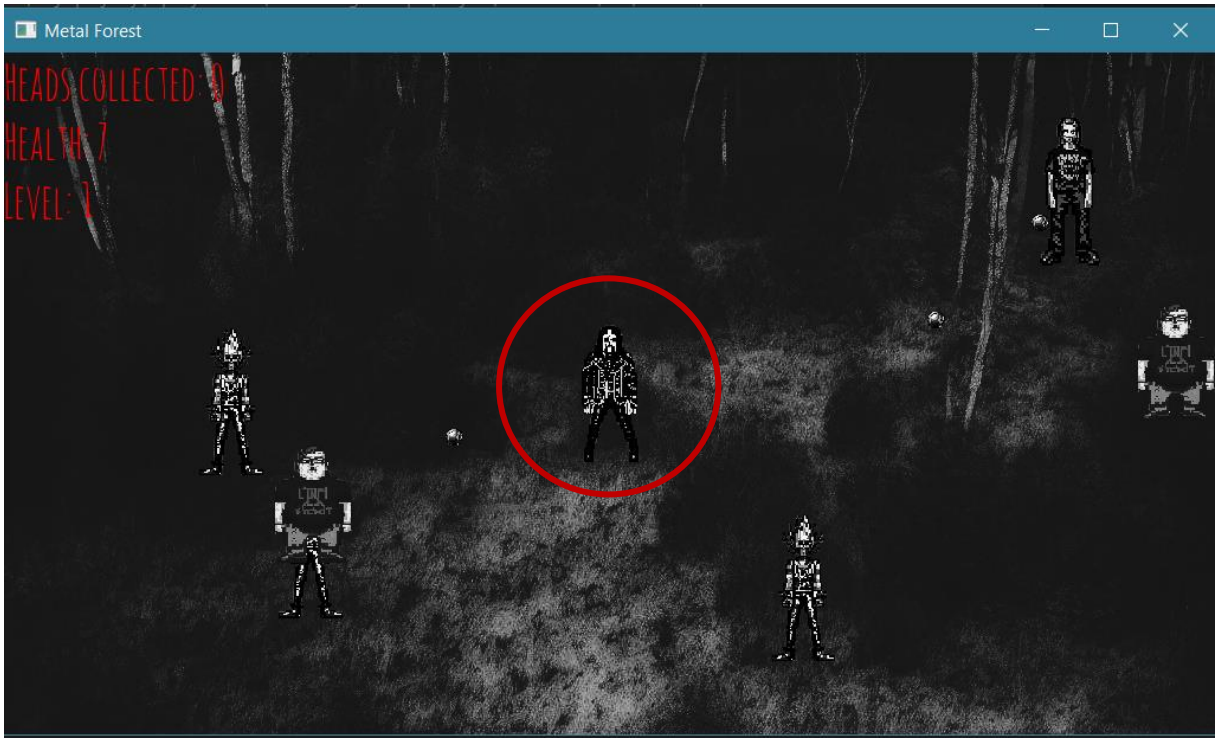
3. Specyfikacja zewnętrzna

Uruchamiając aplikację, zaczynamy od menu głównego:

Zdjęcie nr 1. Menu główne.

Mamy do wyboru opcje rozpoczęcia gry, lub jej zakończenia. Po naciśnięciu drugiej opcji, aplikacja kończy swoje działanie.

Gdy zdecydujemy się na pierwszą opcję, gra się rozpoczyna:



Zdjęcie nr 2. Rozgrywka

Mechanika rozgrywki:

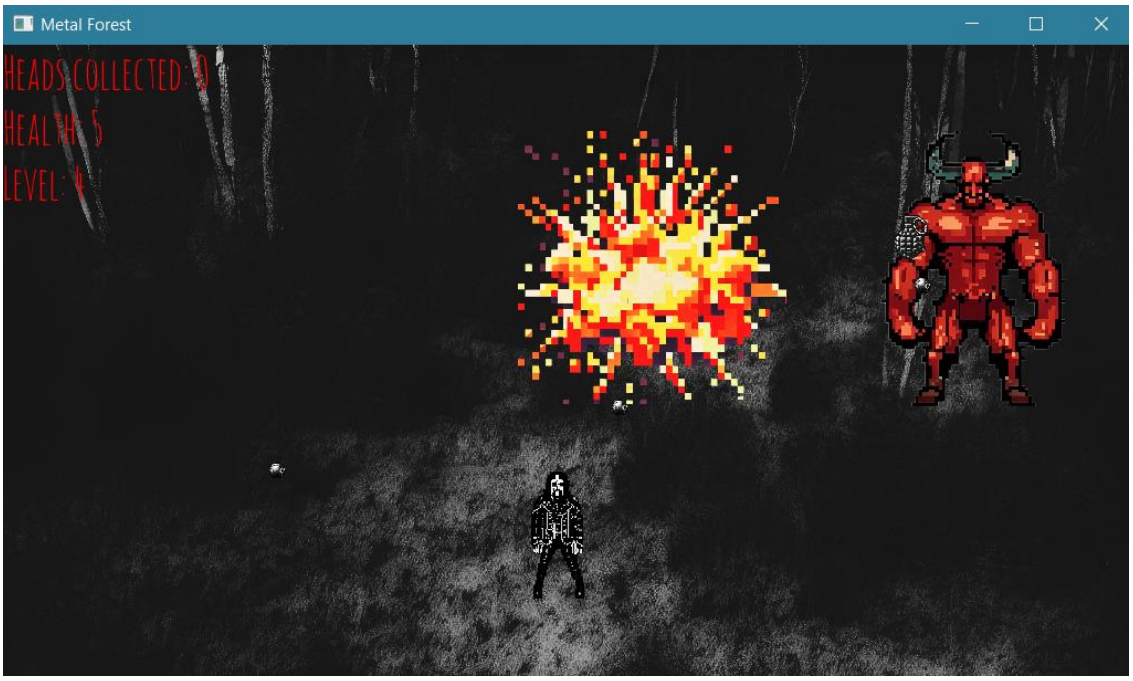
Poruszamy się za pomocą naszej postaci (zaznaczona za pomocą czerwonego okręgu). Poruszamy się za pomocą klawiszy: W, A, S, D. Aby zaatakować (rzucić gitarą), musimy nacisnąć klawisz spacji. Po trafieniu w przeciwnika, ilość jego punktów życia jest obniżana o 1, a jego sylwetka zmienia się na kilka sekund na czerwono (szaro w przypadku bossa)

Przeciwnicy atakują nas za pomocą pocisków, z częstotliwością odpowiadającą swojemu typowi. Gra posiada 4 takie typy: NuMetal, GlamMetal, DeathMetal (zdjęcie nr 2) oraz boss (zdjęcie nr 3). Każdy przeciwnik posiada swoją własną częstotliwość wykonywania strzałów (jedno trafienie pociskiem odbiera nam 1 punkt życia), oraz ilość punktów życia (jedno trafienie gitarą liczy się jako -1 pkt).

Gra podzielona jest na z góry przygotowane poziomy. W każdym poziomie pojawia się określona ilość przeciwników. Po zabiciu wszystkich, przechodzimy do następnego. W poziomach będących wielokrotnością liczb 1,2,3, może pojawić się każdy rodzaj przeciwników, oprócz bossa. On pojawia się w poziomach, których liczba jest wielokrotnością liczby 4. Z każdym poziomem wzrasta trudność gry, ze względu na zwiększanie ilości pojawiających się przeciwników. Ta zasada dotyczy również poziomów z bossem, których ilość zwiększa się zawsze o 1.

Po zabiciu przeciwnika, na mapie pojawia się obiekt (głowa, zdjęcie nr 4), który przyznaje nam jedno życie i jeden punkt. Aby go zebrać, musimy po prostu przesunąć się swoją postacią na niego.

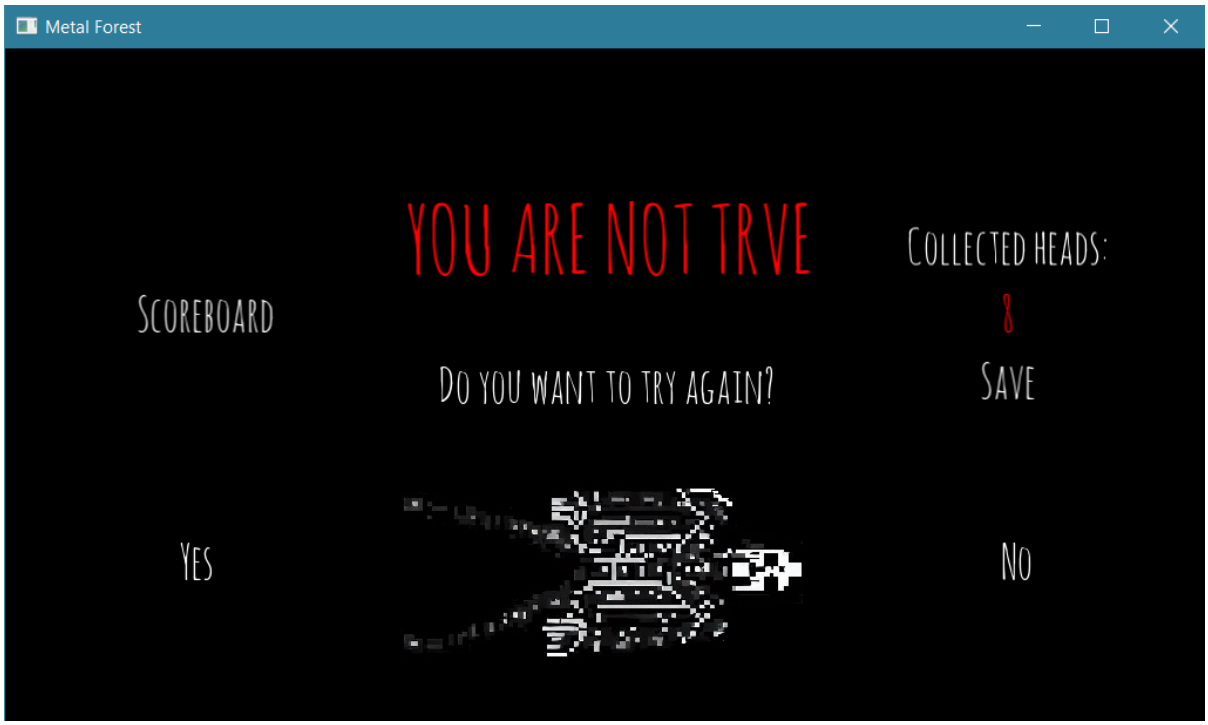
Gra kończy się po śmierci naszej postaci.



Zdjęcie nr 3. Walka z bossem.

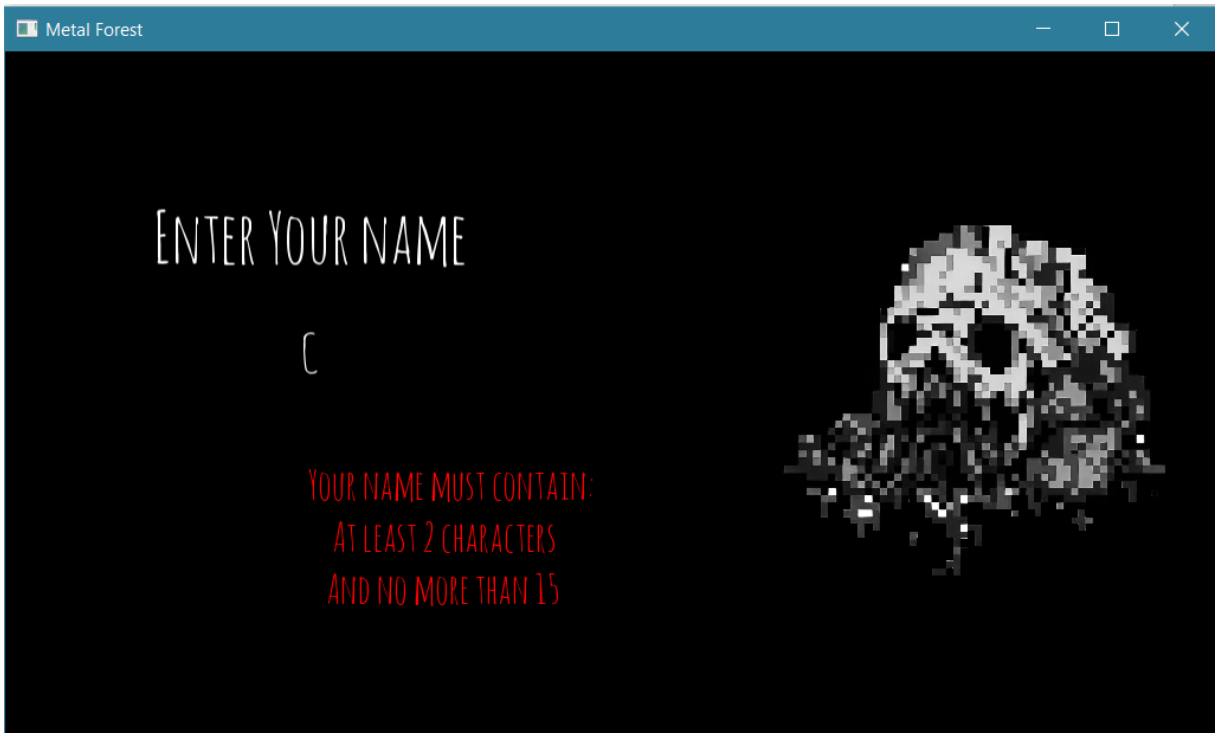
Zdjęcie nr 4. Głowa przeciwnika

Po śmierci naszej postaci, trafiamy do menu końcowego (zdjęcie nr 5)

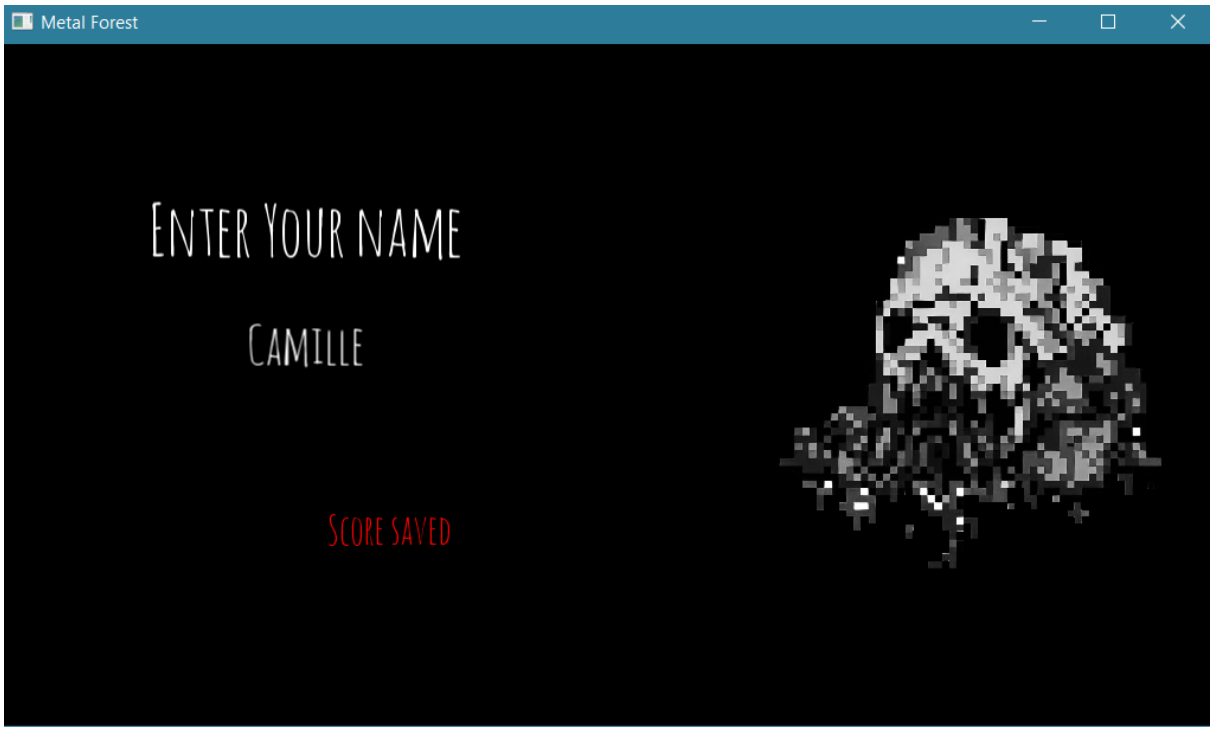


Zdjęcie nr 5. Menu końcowe.

Mamy do wyboru opcje zapisania swojego wyniku (zdjęcie nr 6 i 7), podejrzenia obecnej tabeli najlepszych wyników (zdjęcie nr 8), zrestartowania gry, lub jej zakończenia.

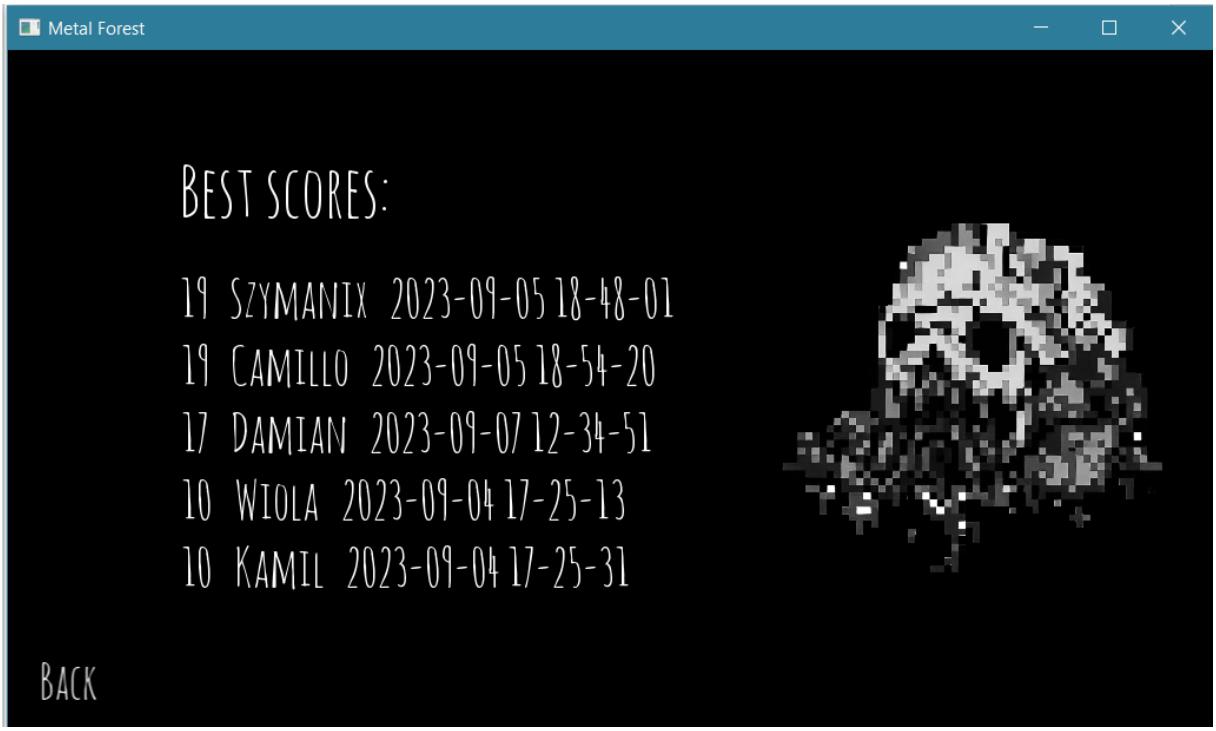


Zdjęcie nr 6. Ekran zapisywania wyników, po błędnym wprowadzeniu nazwy.



Zdjęcie nr 7. Ekran zapisywania wyników, po poprawnym wprowadzeniu nazwy.

Aby zapisać swój wynik, musimy wprowadzić nazwę użytkownika, która zawiera co najmniej 2 znaki i nie zawiera ich więcej niż 15. Jeśli nie zostaną spełnione te warunki, gra zakomunikuje nam konieczność wprowadzenia prawidłowej wartości.

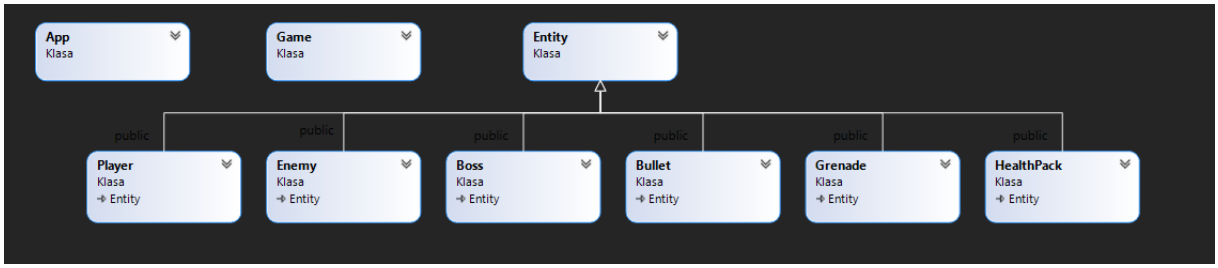


Zdjęcie nr 8. Ekran najlepszych wyników.

4. Specyfikacja wewnętrzna.

Specyfikacja wewnętrzna (klasy [znaczenie obiektu, powiązania z innymi klasami, istotne
Założenie aplikacji jest takie jak przewiduje to prosty system tworzenia gier w SFML. Tworzone jest okno, a następnie obiekty, potem są one updatowane, a na końcu okno jest czyszczone i wszystkie zaktualizowane obiekty są renderowane ponownie. W aplikacji utworzone są poszczególne poziomy, które odpowiadają za ilość przeciwników która jest tworzona. Po zabiciu wszystkich przeciwników, program przechodzi do kolejnego poziomu, tworząc daną ich ilość. Gdy poziom życia gracza spadnie do 0, wszystkie obiekty są usuwane, a gra przechodzi do fazy menu końcowego.

Gra zrealizowana jest obiektowo. Zostało utworzone 10 klas (szczegółowy diagram ze wszystkimi atrybutami oraz metodami znajduje się w projekcie):



Zdjęcie nr 9. Diagram klas

Znaczenie poszczególnych klas w projekcie:

1. App

Odpowiada za zarządzanie logiką aplikacji, czyli tworzenie okna, inicjowanie menu głównego i menu końcowego, uruchamianie w pętli gry (tworzenie obiektu Game i zarządzanie jego kluczowymi metodami, takimi jak update i render), oraz zarządzanie muzyką. Odpowiada też za cały user interface, poza interfejsem samej rozgrywki. Dodatkowo odpowiada za zapisywanie wyników do pliku oraz przeszukiwanie ich, w celu utworzenia ekranu wyników.

2. Game

Odpowiada za zarządzanie logiką samej gry, tzn inicjowanie rozgrywki, tworzenie nowych instancji Entity, updatowanie i usuwanie ich, rozporządzanie interakcjami między poszczególnymi obiektami Entity, oraz renderowanie ich w oknie. Dodatkowo odpowiada za naliczanie punktów rozgrywki.

3. Entity

Jest to klasa czysto wirtualna, po której dziedziczą klasy Player, Enemy, Boss, Bullet, Grenade oraz HealthPack. Każda z tych klas pochodnych posiada własną implementację metod: init, update, attack, move, oraz setPosition. Dodatkowo klasa posiada atrybuty i metody, które odpowidają za zarządzanie zdrowiem, teksturą danej klasy pochodnej (jak i chwilową jej zmianą na teksturę z czerwonym obrysem po dostaniu obrażenia), typy ruchów które są przez nie wykorzystywane, oraz oznaczenie które instancje należy usunąć w fazie update klasy Game.

4. Player

Klasa dziedzicząca wirtualnie po Entity. Na początku gry tworzona jest tylko jedna instancja tej klasy. Jej ruch jest sterowany za pomocą metody, która sprawdza które klawisze są naciskane (W,A,S, lub D), i przekłada to na funkcjonalność move, z odpowiednimi współczynnikami na osiach X oraz Y. Atak jest

aktualizowany na podobnej zasadzie, tylko ze sprawdzeniem klawisza spacji. Tworzony wtedy jest nowy obiekt klasy Bullet (typ 2), w miejscu pozycji instancji Player, oraz porusza się już niezależnie od niej.

5. Enemy

Odpowiada za wszystkich przeciwników, którzy nie są bossem. Typ przeciwnika jest ustalany na podstawie typu otrzymanego w konstruktorze parametrycznym z pozycji Game. Wtedy ustala się również poziom zdrowia, tekstura i częstotliwość ataku. Ich ruch jest taki sam, niezależnie od typu. Porusza się w mechanice „randomMove" a potem „moveSquare". Wybiera on losowe miejsce w obrębie okna, ustala kierunek, oraz przesuwa się tam powoli zmniejszając dystans na osi X oraz Y. Jeśli dystans na jednej z osi wynosi 0, obiekt przesuwa się po linii w lewo, prawo, górę lub dół. Jeśli dystans na obu osiach nie jest równy 0, przesuwa się po skosie. Obiekt strzela pociskami, tzn gdy timer odpowiadający za fireRate spadnie do 0 (minie odpowiednia ilość cykli update), nowa instancja klasy Bullet jest tworzona i ustawiana w miejscu swojego właściciela. Potem jest już od niego niezależna. Gdy dojdzie do wyznaczonego miejsca, losowane jest kolejne do którego się przesuwa. Po każdym uderzeniu gitarą, obiekt traci jedno życie z puli. Gdy wartość puli spadnie do 0, obiekt jest oznaczony jako do usunięcia, oraz jest usuwany w fazie update klasy Game.

6. Boss

Odpowiada za zarządzanie mechaniką bossa. Jest ona bardzo podobna do Enemy, lecz posiada dodatkowe funkcje takie jak wyrzucanie granatów. Instancje klasy Grenade są tworzone i usuwane w podobny sposób co bullet. Mechanika obrażeń jest taka sama jak w przypadku Enemy.

7. Bullet

Odpowiada za zarządzanie mechaniką pocisków, zarówno pocisków przeciwników, jak i gitary gracza. W przypadku pocisków, po utworzeniu jest umiejscawiany na pozycji właściciela, oraz porusza się w tzw mechanice „moveDiagonallyToPlayer". Do funkcji przekazywany jest vector zawierający 2 wartości float, odpowiadające ostaniej pozycji gracza. Gdy ustalony jest już cel, pocisk porusza się po skosie w jego kierunku. Kierunek ten, jest obliczany według zasady normalizacji wektora ruchu, tzn poprzez wyliczenie długości wektora wg wzoru sqrt(x*x+y*y), a następnie podzielenie składowych x oraz y przez tę długość. Umożliwia to ruch tego obiektu pod dowolnym kątem, z taką samą prędkością. Jeśli poruszając się, napotka gracza, jest usuwany, a gracz traci jedno życie. Jeśli nie, posusza się w tym samym kierunku, aż napotka granice okna, po czym jest usuwany. W przypadku gitary gracza, mechanika ruchu jest zaimplementowana w trochę inny sposób. Zczytywany jest kierunek ruchu gracza (lewo, prawo, góra, dół lub na ukos) i gitara przejmuje ten kierunek. Porusza się również aż do napotkania przeciwnika (po styczności usuwa mu życie), lub do granic okna, po czym jest usuwana.

8. Grenade

Odpowiada za zarządzanie mechaniką granatów należących do bossa. Kolejne instancje są tworzone w ten sam sposób co pociski, tzn z odpowiednią częstotliwością, w miejscu bossa. Granat nie porusza się, lecz ma timer który odlicza jego wybuch. Po eksplocji, zadaje bardzo duże obrażenia graczowi, gdy znajdzie się w jego polu rażenia. Po wybuchu jest usuwany.

9. HealthPack

Odpowiada za zarządzanie mechaniką tzw głów. Ich główny cel to dodawanie graczowi punktów życia i punktów zwycięstwa. Jest tworzony po usunięciu danego przeciwnika, w miejscu jego śmierci. Gdy gracz w niego wejdzie, przyznawane są mu odpowiednie wartości, a sam obiekt jest usuwany.

W projekcie zostały użyte takie struktury danych jak vector (np. przechowywanie instancji Enemy) czy tuple (np przechowywanie wyniku razem z nazwą oraz datą). Ważnym elementem całego projektu są inteligentne wskaźniki, a konkretniej unique_ptr. Jest on używany podczas tworzenia wszystkich instancji klas pochodnych Entity (poza Player). Umożliwia on łatwe kontrolowanie momentów w których są one usuwane oraz zapobieganie wyciekom pamięci.

W projekcie zostały zastosowane poniższe tematy laboratoryjne:

1. Wielowątkowość

```
App MetalForest;
std::thread musicThread([&MetalForest]() {MetalForest.musicMethod();});
while (MetalForest.getGamePhase() != 4)
{
    switch (MetalForest.getGamePhase())
    {
    case 1: MetalForest.mainMenuMethod(); break;
    case 2: MetalForest.gameMethod(); break;
    case 3: MetalForest.endGameMethod(); break;
    default: std::cout << "GAMEPHASE ERROR" << std::endl; break;
    }
}
musicThread.join();
return 0;
```

Zdjęcie nr 10. Implementacja wielowątkowości

Temat ten jest wykorzystany przy wywoływaniu funkcji odpowiadającej za muzykę. Jest ona zarządzana w osobnym wątku. Zaimplementowanie tego w ten sposób, było konieczne by móc zapętlać poszczególne utwory, bez wpływu na samą rozgrywkę.

2. Regex

```
std::regex correctName ( "\\w{2,15}");
while (!nameReady and appWindowPtr->isOpen())
```

Zdjęcie nr 11. Implementacja regex 1

```
if (std::regex_match(enteredText, correctName))
{
    result.setString("\n    Score saved");
    displayResult = true;
    uploadScoreToFile(enteredText);
    end = true;
}
```

Zdjęcie nr 12. Implementacja regex 2

Temat został wykorzystany do sprawdzania poprawności wprowadzanej nazwy gracza. Poprawna nazwa to taka, która zawiera przynajmniej 2 dowolne znaki, lecz nie więcej niż 15.

3. Ranges

```
std::sort(scoreVector.begin(), scoreVector.end());
auto reversed = std::ranges::views::reverse(scoreVector);
std::vector <std::tuple<int, std::string, std::string>> toCut;
std::ranges::for_each(reversed, [&toCut](auto score) {toCut.push_back(score); });
auto cut = std::ranges::views::take(toCut, 5);
std::string finalScores;
std::ranges::for_each(cut, [&finalScores](auto score)
    {
        finalScores+= std::to_string(std::get<0>(score)) + "    " + std::get<1>(score
    });
```

Zdjęcie nr 13. Implementacja ranges

Biblioteka ranges została wykorzystana do obrócenia wektora z wynikami (tuple), a następnie wybrania tylko 5 pierwszych wartości z niego. Na koniec wszystkie składowe tuple są konwertowane do stringa i wyświetlane na ekranie wyników.

4. Filesystem

```
void App::uploadScoreToFile(std::string name)
{
    std::filesystem::path scoresPath = std::filesystem::current_path() / "Scores";
    if (!std::filesystem::exists(scoresPath))
        std::filesystem::create_directory(scoresPath);
    std::filesystem::path nameDirPath = scoresPath / name;
    if (!std::filesystem::exists(nameDirPath))
        std::filesystem::create_directory(nameDirPath);
    std::time_t currentTime;
    std::tm localTimeInfo;
    char timeStr[100];
    if (std::time(&currentTime) != -1 && localtime_s(&localTimeInfo, &currentTime) == 0)
    {
        std::strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H-%M-%S", &localTimeInfo);
        std::string time = timeStr;
        std::string txt(".txt");
        std::filesystem::path newScore = nameDirPath / (time + txt);
        std::ofstream scoreStream(newScore);
        if (scoreStream.is_open())
        {
            scoreStream << "Score:\n" << score;
        }
        scoreStream.close();
    }
```

Zdjęcie nr 14. Implementacja filesystem 1

```
std::filesystem::directory_entry current_directory(std::filesystem::current_path() / "Scores");
for (const std::filesystem::directory_entry& entry : std::filesystem::directory_iterator(current_directory))
{
    auto name = entry.path().filename();
    for (const std::filesystem::directory_entry& entry2 : std::filesystem::directory_iterator(entry))
    {
        auto date = entry2.path().filename();
        std::ifstream filestream(entry2.path().string());
        if (filestream.is_open())
        {
            std::string score;
            while (std::getline(filestream, score))
            {
                if (score == "Score:")
                {
                    if (std::getline(filestream, score))
                    {
                        std::string txt(".txt");
                        std::string dateCut = date.string();
                        auto found= dateCut.find(txt);
                        if (found != std::string::npos)
                            dateCut = dateCut.erase(found, txt.length());
                        std::tuple <int, std::string, std::string> s = std::make_tuple(std::stoi(score), name.string(), dateCut);
                        scoreVector.push_back(s);
```

Zdjęcie nr 15. Implementacja filesystem 2

Temat ten został wykorzystany zarządzania katalogami przechowującymi wyniki. Podczas zapisywania wyników, tworzony jest nowy katalog z nazwą gracza (jeśli nie istnieje), a potem zapisywany jest w nim wynik, z tytułem daty osiągnięcia go. Podczas tworzenia i wyświetlania tabeli wyników, zbierane są wszystkie wyniki osiągnięte przez wszystkich graczy, po czym dany wynik z imieniem oraz datą wpisywany jest w tuple i umieszczany w wektorze.

5. Testowanie i uruchamianie

Program został przetestowany pod kątem błędów, wyjątków oraz wycieków pamięci.

Program nie generuje żadnych z powyższych problemów (jedyny zostanie opisany poniżej). Wszystkie instancje klas tworzone za pomocą new są usuwane dzięki inteligentym wskaźnikom, a konkretniej unique_ptr, dzięki czemu nie pozostawiają po sobie żadnych wycieków. Jedynym problemem, jest funkcja przedstawiona na zdjęciu 17. Występuje wyciek pamięci pod koniec działania programu, a konkretniej w funkcji odpowiadającej za zapis wyniku do pliku. Wynika on z zastosowanej funkcjonalności std::time. Niestety nie udało się go usunąć, lecz nie wpływa on na rozgrywkę, ani odpowiednie działanie programu.

Source File	Value	Hit Count	Leak Type	Module Name	Module P...	Size	Proce...	Times...	Seque...	Thread Id
App.cpp, line 393 (C:\Users\Kamil\Deskto...	0x00000148d94bf4...	55	Heap memo...	Metal Forest.exe	C:\Users\...	5048	22692	07.09....	206758	15892
App.cpp, line 393 (C:\Users\Kamil\Deskto...	0x00000148c8d70d...	1	Heap memo...	Metal Forest.exe	C:\Users\...	448	22692	07.09....	206717	15892
exe_common.inl, line 167 (D:\a_work\1\s\...	0x00000148c8d41e...	1	Heap memo...	Metal Forest.exe	C:\Users\...	143	22692	07.09....	141947	15892

Zdjęcie nr 16. Fragment wyników z aplikacji Deleaker

```
if (std::time(&currentTime) != -1 && localtime_s(&localTimeInfo, &currentTime) == 0)
{
    std::strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H-%M-%S", &localTimeInfo);
    std::string time = timeStr;
    std::string txt(".txt");
    std::filesystem::path newScore = nameDirPath / (time + txt);
    std::ofstream scoreStream(newScore);
    if (scoreStream.is_open())
    {
        scoreStream << "Score:\n" << score;
    }
    scoreStream.close();
}
```

Zdjęcie nr 17. Fragment kodu generujący wyciek pamięci

6. Wnioski

Stworzenie tego projektu było bardzo rozwijające. Największą trudnością było zaimplementowanie odpowiednich mechanik obiektowych, oraz mechanik ruchu, takich jak np. normalizacja wektora ruchu. Przedmiot okazał się bardzo pomocny w poszerzeniu mojej wiedzy w tych dziedzinach.