

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI, FOTONIKI I MIKROSYSTEMÓW

KIERUNEK: AUTOMATYKA I ROBOTYKA

PRACA DYPLOMOWA
INŻYNIERSKA

Tytuł pracy:

Wykorzystanie metod sztucznej inteligencji w
grze typu roguelike

AUTOR:

Adam Czarnowski

PROMOTOR:

Dr inż. Łukasz Jeleń

Spis treści

1. Wstęp	5
2. Cel i zakres pracy	6
3. Geneza gier typu Roguelike	7
3.1. Specyfikacja gatunku	7
3.2. Historia	7
4. Projekt	9
4.1. Elementy gry	9
4.2. Założenia funkcjonalne	10
4.3. Katalogi i moduły aplikacji	11
4.4. Architektura	13
5. Implementacja	15
5.1. Graficzny interfejs użytkownika	15
5.1.1. Tła i warstwy	15
5.1.2. Interakcja z programem	19
5.2. Model i logika gry	20
5.2.1. Zapis oraz wczytywanie gry	20
5.2.2. Pętle aplikacji	21
5.2.3. Generowanie mapy	23
5.2.4. Aktualizacja wyników	24
5.2.5. Mini gry	25
5.2.6. Algorytmy przeciwników	27
6. Podsumowanie	35
6.1. Wnioski	35
Literatura	36

Spis rysunków

3.1. <i>Rogue 1980</i>	7
3.2. <i>Hades, Gra roku 2020</i>	8
4.1. Przykład warstwy graficznej	9
4.2. Przykład lokacji niezawierającej warstwy i z warstwą naniezioną na grafikę tła	9
4.3. Losowo generowane lokacje mapy	10
4.4. Diagram przepływu pracy	13
4.5. Diagram zależności modułów	14
5.1. Graficzny interfejs użytkownika	15
5.2. Warstwa transparentna oraz nietransparentna	16
5.3. Plik z kolejnymi klatkami ataku przeciwnika	17
5.4. Widok pętli gameScreen() w aplikacji	21
5.5. Mini gra Spróbuj szczęścia	25
5.6. Mini gra Połącz cztery	27
5.7. Szkielet - algorytm walki	28
5.8. Minotaur - algorytm walki	29
5.9. Wiedźma - algorytm walki	30
5.10. Golem - algorytm walki	31
5.11. Zabójca - algorytm walki	32
5.12. Czarnoksiężnik - algorytm walki	33
5.13. Wojownik nocy - algorytm walki	34

Spis tabel

5.1. <i>Opis funkcji wyświetlania obrazu</i>	16
5.2. <i>Opis funkcji wyświetlania tekstu</i>	17
5.3. <i>Opis klasy SpriteSheet</i>	17
5.4. <i>Konstruktor klasy Button</i>	20

Rozdział 1

Wstęp

Gry komputerowe to nie tylko forma rozrywki czy dochód dla firm produkujących, ale również nieodłączna część kultury dzisiejszych czasów. Z łatwością można stwierdzić, że stanowią nowy gatunek sztuki, cieszący się popularnością na poziomie kina czy muzyki. Wraz z postępem technologicznym dystrybuowanych jest coraz więcej tytułów, co przekłada się na stale rosnące oczekiwania graczy, którym deweloperzy muszą sprostać. Skutkiem rosnących wymagań są nowe, lepsze, ciekawsze i efektywniejsze rozwiązania. Jednym z nich jest sztuczna inteligencja, której zadaniem jest tworzenie i kontrolowanie otoczenia postaci gracza oraz sterowanie fizjologią i psychologią postaci, wynikiem czego jest większe zaangażowanie przez gracza w grę. Stworzona przeze mnie aplikacją o nazwie *SoulCollector* jest grą z gatunku *Roguelike* zaprojektowana w języku Python w wersji 3.8 przy użyciu biblioteki Pygame. Zostały zaimplementowane algorytmy sztucznej inteligencji losowo generujące mapę rozgrywki, sterujące bohaterami niezależnymi oraz przeciwnikami, których głównym celem jest przeżycie możliwie najdłuższego czasu oraz doprowadzenie bohatera gracza do porażki.

Rozdział 2

Cel i zakres pracy

Celem pracy dyplomowej jest stworzenie wieloplatformowej, przygodowej gry komputerowej z gatunku roguelike z ciekawą szatą graficzną, w której przeciwnicy wykorzystują algorytmy sztucznej inteligencji. Zadaniem sztucznej inteligencji będzie przetrwanie jak największej ilości rund oraz doprowadzenie gracza do przegranej. Przeciwnicy muszą różnić się swoimi umiejętnościami oraz statystykami, by uzyskać wyższy poziom imersji. W poczynionych przeze mnie założeniach, gra SoulCollector miała być grą prostą i intuicyjną podczas przeprowadzanej rozgrywki. Aplikacja została zaimplementowana w języku Python.

Właściwa część mojej pracy zawiera trzy rozdziały, który każdy z nich porusza tematykę innej kategorii. W trzecim rozdziale opisuję charakterystykę gier z gatunku roguelike oraz przytaczam jej historię. Następny rozdział stanowi szczegółowy opis elementów rozgrywki i funkcjonalności dostępnych w grze. Rozdział czwarty zawiera również przedstawienie architektury programu wraz z opisem stworzonych katalogów i modułów. Najobszerniejszym rozdziałem, który traktuje o implementacji i realizacji praktycznej koncepcji jest rozdział piąty. We wspomnianym rozdziale przedstawiono proces wdrożenia graficznego interfejsu użytkownika oraz interakcji z programem, ale również model i logikę gry. Podrozdział model i logika gry, naświetla czytelnikowi algorytmy sztucznej inteligencji przeciwników i mini gier, ale również opisuje proces zapisu i wczytania gry, generowania mapy, aktualizacji wyników oraz pętli aplikacji pod względem programistycznym. Pracę zakończę krótkim podsumowaniem, w którym zawarte będą najważniejsze punkty poprzednich rozdziałów oraz wnioski odnoszące się do programowania sztucznej inteligencji w produkcjach gier wideo.

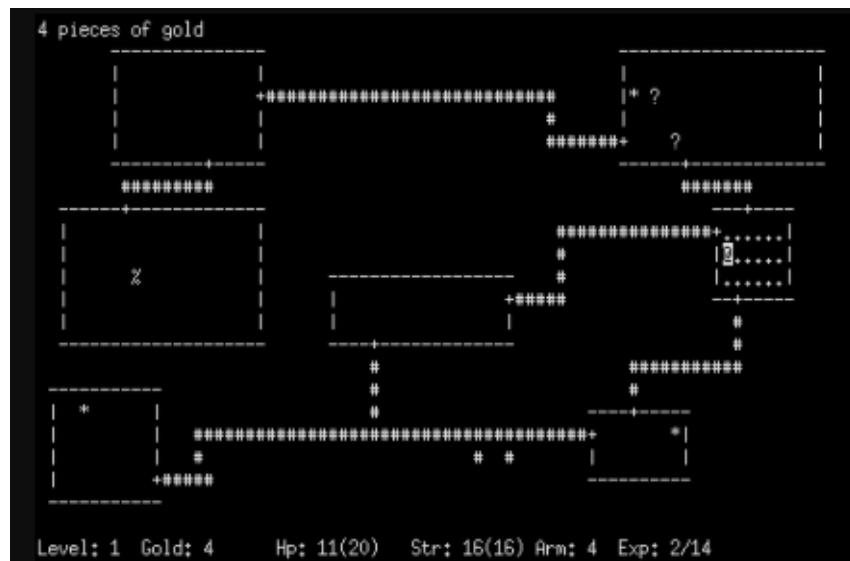
Rozdział 3

Geneza gier typu Roguelike

3.1. Specyfikacja gatunku

Roguelike jest to podgatunek gier RPG (*ang. role-playing game*), który charakteryzuje się dużą losowością oraz śmiercią permanentną. Najczęściej implementowanymi sposobami prowadzenia rozgrywki są gry turowe oraz gry oparte na siatce 2D, natomiast coraz częściej spotykane są tytuły z widokiem izometrycznym. Podczas każdej rozgrywki mapy oraz przeciwnicy są generowane na nowo według ściśle ustalonego schematu, dzięki czemu każdorazowo na gracza czeka nowe wyzwanie oraz niepewność co spotka przechodząc na kolejny poziom. Typ gry nie zezwala na zapis między etapami, co skutkuje częstą porażką i utratą zdobytych dóbr podczas rozgrywki. Dzięki systemowi śmierci permanentnej gracz musi przechodzić każdy z poziomów od początku, jednakże część progresu jest zachowywana, co pozwala na rozwój postaci, by z każdą próbą zwiększała się szansa na pokonanie wszystkich oponentów wraz z finałowym przeciwnikiem.

3.2. Historia

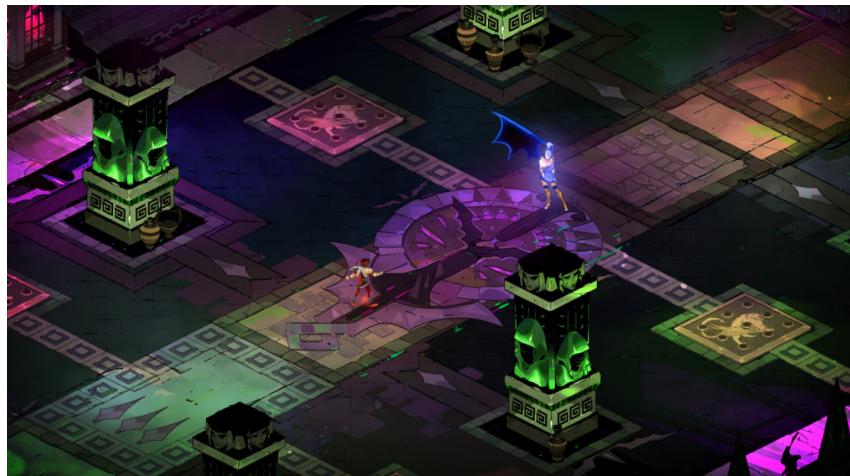


Rys. 3.1: *Rogue* 1980

Pierwszą stworzoną grą, dzięki której gatunek zawdzięcza swoją nazwę jest *Rogue* wydany w 1980 roku na systemy uniksowe. Mechanika gry została oparta na planszowej grze fantasy

Dungeons & Dragons, w której gracz porusza się między pokojami oraz korytarzami walcząc z potworami i zbierając skarby. *Rogue* cechuje się interfejsem tekstowym, gdzie wszystkie elementy rozgrywki włączając w to bohatera, lochy oraz przeciwników są znakami w standardzie ASCII.

Niefortunnie dla twórców, rodzaj rozgrywki i interfejs użytkownika nie cieszył się dużym uznaniem u schyłku XX wieku. Jednakże, skutkiem szybkiego postępu technologicznego na początku lat dwa tysiące był znaczny rozwój gatunku i jego popularności. W roku 2008 Derek Yu stworzył grę *Spelunky*, która uważana jest za główny wkład w rozwój niezależnych gier typu Roguelike. Po wielkim sukcesie Yu, inni deweloperzy dostrzegli potencjał gatunku i skupili się na jego rozwoju. Trzy lata później powstała gra *The Binding of Isaac*, która uzyskała rozmgłos na całym świecie. W roku 2014 twórcy poinformowali, że ich produkcja sprzedała się w ponad trzech milionach kopii, co było ogromnym osiągnięciem. Do dnia dzisiejszego gra jest rozwijana i zyskuje nowych graczy. Jednakże, gra *Hades* wyprodukowana w 2020 roku przez twórców *Supergiant Games* spowodowała, że gatunek z początku uznawany za mierny może odnosić wielkie sukcesy. Wspomniana gra zyskała tytuł gry roku 2020, co jest największym osiągnięciem dla deweloperów.



Rys. 3.2: *Hades*, Gra roku 2020

Rozdział 4

Projekt

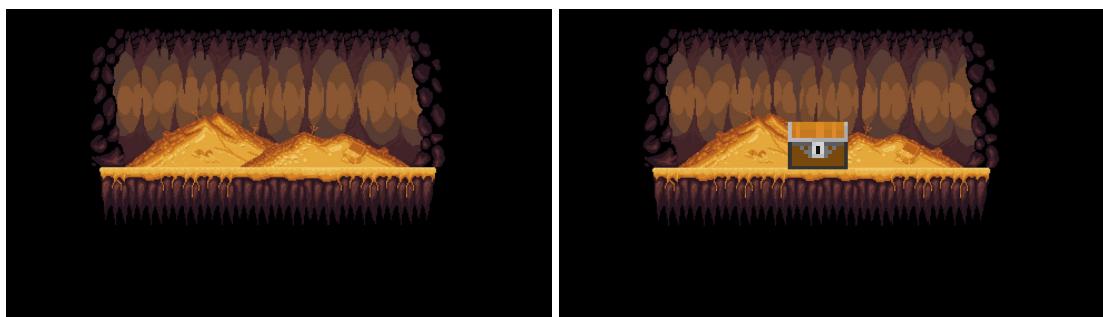
4.1. Elementy gry

Przed omówieniem dalszego opisu projektu gry oraz jej implementacji, ważnym jest przybliżenie pojęć związanych z grą *SoulCollector*, do których w dalszej części rozprawy będę się odnosił.

- **Lokacja:** Termin ten odnosi się do dowolnego miejsca, które może odwiedzić gracz w trakcie gry. Każda lokacja zawiera grafiki stanowiące tło gry oraz podstawowe elementy sterowania.
- **Warstwa:** Jest to osobna grafika, którą nakłada się na grafikę tła lokacji. Jest to wizualna reprezentacja przedmiotów oraz postaci niezależnych dostępnych w świecie gry.



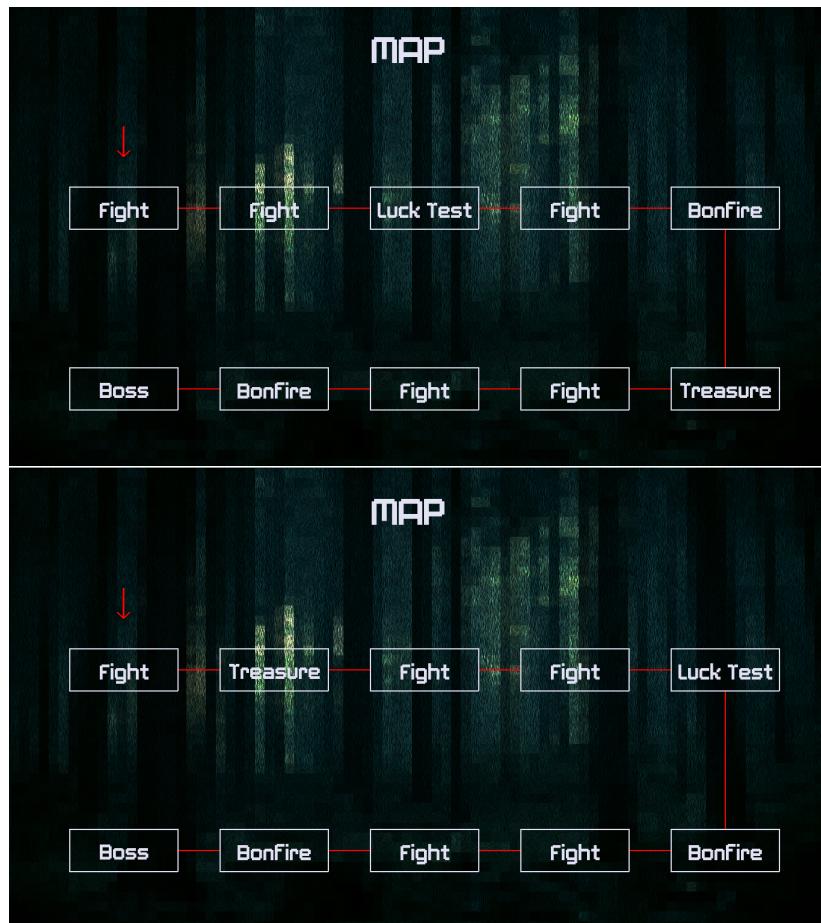
Rys. 4.1: Przykład warstwy graficznej



Rys. 4.2: Przykład lokacji niezawierającej warstwy i z warstwą naniesioną na grafikę tła

- **Przedmiot:** Określenie obejmujące logiczną strukturę danych, którą utożsamia się dokładnie z jednym obiektem w danej lokacji. Posiadać może wiele atrybutów takich jak np. wiadomość, miejsce występowania czy kształt. Dodatkowo, zebrane przez gracza przedmioty znacząco wpływają na rozwój postaci oraz postęp rozgrywki.
- **Mini gra:** Jest to pomniejsza forma logicznej zagadki lub przeszkody, mająca na celu urozmaicenie rozgrywki. Gracz w zależności od wyniku mini gry zyskuje lub traci bonusy dostępne dla postaci. Zawiera unikatowy interfejs w danej lokacji i posiada elementy interaktywnej komunikacji z graczem.

- **Ekwipunek:** Jest to zbiór zebranych przez bohatera przedmiotów w trakcie gry. Dzięki graficznej reprezentacji warstwy, użytkownik pozyskuje informację o statystykach przedmiotu.
- **Mapa:** Pokazuje kierunek zwiedzanych lokacji podczas wyprawy. Gracz poruszając się między etapami, nie jest w stanie przewidzieć co będzie się w nich znajdować. Mapa jest generowana losowo podczas każdej nowo rozpoczętej rozgrywki.



Rys. 4.3: Losowo generowane lokacje mapy

- **Animacja:** Jest to ruchomy element gry, w który gracz nie może ingerować, jedynie posiada możliwość jej wyzwalania. Animacje świata gry wyświetlane są w 60 klatkach na sekundę (*ang. FPS, frames per second*).
- **Złoto:** Podstawowa waluta wykorzystywana przez gracza do pozyskiwania dodatkowych jednorazowych bonusów w grze.
- **Dusza:** Druga obok złota waluta, wykorzystywana do ulepszania ulepszeń stałych postaci, tzn. bonusów, które nie resetują się po porażce gracza. Znacznie bardziej skomplikowana i trudna do zdobycia.
- **Wynik końcowy:** Jest to suma punktów pozyskanych przez gracza w trakcie pojedynczej rozgrywki. Obliczana jest na podstawie zdobytych osiągnięć, na podstawie trudności ich zdobycia. Z poziomu dostępnego dla gracza lokacji, możliwym jest sprawdzenie dziesięciu najlepszych wyników z wszystkich rozegranych gier.

4.2. Założenia funkcjonalne

Dla użytkownika lub testera korzystającego z aplikacji ważnym są wymagania funkcjonalne, to dzięki nim program osiąga pełne możliwości i określa zakres działań, które będą realizowane. Ich brak lub niespójność może znacznie wzbudzić niechęć do dalszego użytkowania produktu.

W grze SoulCollector zostało zaimplementowane wiele funkcji z gatunku roguelike, pozwalających graczowi na kontrolowanie postaci, walki i poruszania się po warstwie świata. Stworzona została lista najważniejszych wymagań funkcjonalnych w grze, którą przedstawiam poniżej.

- Dostępność podpowiedzi dla gracza podczas rozgrywki.
- Możliwość zmiany poziomu trudności przeciwników.
- Gracz dysponuje informacją o zdobytych przedmiotach.
- Gracz otrzymuje możliwość sprawdzenia podstawowych statystyk bohatera i przeciwnika.
- Nawigacja między lokacjami i elementami otoczenia jest przedstawiona w prosty i czytelny sposób.
- Gracz otrzymuje informacje o wymaganiach ulepszania postaci.
- W przypadku niespełnienia wymaganego warunku przez gracza, wyświetlane są odpowiednie komunikaty.
- Najlepsze dziesięć wyników z wszystkich rozegranych gier reprezentowana jest w postaci rankingu.
- Gra obsługuje funkcję autozapisu po każdej wykonanej czynności w głównej lokacji oraz po wygranej lub przegranej rozgrywce.
- Gracz po przegranej wraca do stanu początkowego postaci, wszystkie przedmioty oraz bonusy uzyskane podczas wyprawy zostają usunięte z ekwipunku.
- Bonusy kategorii stałej nie są resetowane przy porażce.

4.3. Katalogi i moduły aplikacji

W wyniku projektowania oraz tworzenia aplikacji zostały utworzone katalogi, których celem jest przechowywanie zasobów budujących grę, np. grafikę lokacji czy zapis wyników końcowych. Każda z dostępnych mechanik oraz funkcjonalności rozpisana jest w osobnych modułach, by uzyskać największą możliwą przejrzystość kodu. Poniżej przedstawiono opis katalogów i modułów.



Katalogi

- **fonts** - katalog zawiera plik czcionki z rozszerzeniem *.ttf*, który wykorzystywany jest do wyświetlania tekstów na ekranie gry.
- **img** - zawiera wszystkie elementy grafiki wykorzystywane w grze. Dzięki odpowiednim funkcjom, program wczytuje z katalogu wymaganą w zależności od lokacji grafikę.
- **save** - w tym katalogu przechowywane są pliki zapisu stanu gry. Pliki zapisu gry reprezentowane są przez pliki tekstowe, wczytywane do aplikacji w odpowiednich momentach rozgrywki.
- **sprites** - katalog zawierający wszystkie graficzne, poklatkowe elementy animacji przeciwników.



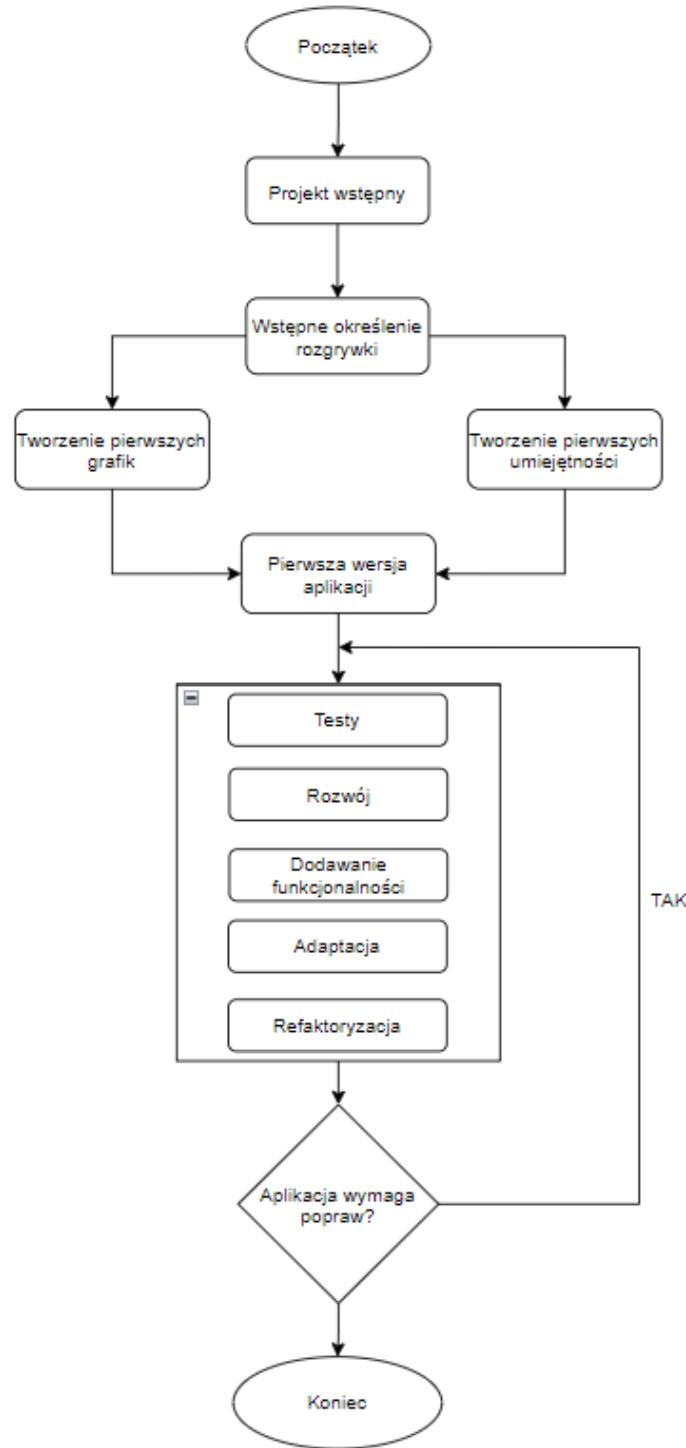
Moduły

- **main.py** - plik główny aplikacji, znajduje się tu jedna funkcja, odpowiedzialna za uruchomienie całego programu.
- **game.py** - moduł ten zawiera połączone funkcjonalności gry. Dzięki niemu wyświetlana jest cała aplikacja wraz z grafiką i elementami obsługi lokacji, mapy oraz walki.
- **controlUI.py** - moduł ten w całości przeznaczony jest do gromadzenia odpowiednich funkcji tekstowych wyświetlanych w odpowiednich momentach gry oraz klasa definiująca przyciski i ich akcje.

- **resources.py** - zawiera najważniejsze stałe - używane kolory w RGB, przypisane grafiki do zmiennych oraz wersje wykorzystywanych czcionek. Dodatkowo w module, znajdują się funkcje rysujące na ekranie czy funkcje obsługujące sytuowanie tekstu oraz grafik.
- **save.py** - dzięki zawartym w nim funkcjom obsługiwany jest zapis rozgrywki. Pozwala na czytanie danych plików tekstowych zawartych w katalogu *save* oraz ich nadpisywanie.
- **score.py** - w pliku znajdują się funkcjonalności zliczające wynik końcowy gracza, uaktualniający tablicę wyników oraz sortowanie wyników algorytmem szybkiego sortowania.
- **mapGenerator.py** - zawiera w sobie klasę algorytmu odpowiedzialną za losowe generowanie świata wyprawy oraz metody wykorzystywane do pozyskania pozycji bohatera.
- **turns.py** - moduł ten jest wykorzystywany podczas walki lub mini gry. Zawiera w sobie klasę obsługi systemu turowego. Podczas inicjalizacji losowo wybierana jest strona rozpoczętająca walkę.
- **player.py** - zawiera klasę postaci, w której skład wchodzą metody umożliwiające interakcję ze sterowanym bohaterem, jego umiejętności oraz statystyki go określające. Przykładowymi metodami są *expUp()*, która odpowiada za podniesienie statystyk po zdobytym poziomie bohatera oraz *heal()* używana podczas walki przez gracza do leczenia głównego bohatera.
- **enemy.py** - zawiera klasę przeciwników, w której zapisane są wszystkie statystyki oraz umiejętności. Najważniejszym elementem klasy są algorytmy sztucznej inteligencji, które są wykorzystywane podczas walki z użytkownikiem.
- **animations.py** - moduł ten opiera się na trzech klasach obsługujących wyświetlanie animacji. Klasa *SpriteSheet* przygotowywuje warstwy obrazów potrzebnych do wyświetlenia na ekranie użytkownika, natomiast klasa *PlayerAnimations* oraz *EnemyAnimations* odpowiedzialna jest za wyświetlanie animacji w odpowiedniej ilości klatek na sekundę w zależności od decyzji gracza.
- **equipment.py** - moduł zawierający funkcję wyświetlania przedmiotów oraz walut zdobytych podczas rozgrywki.
- **miniGame.py** - zawiera wszystkie algorytmy potrzebne do wyświetlenia oraz rozegrania mini gry opierającej się na popularnej grze *Connect 4*. Obsługuje również algorytm zachowania przeciwnika.
- **testLuckGame.py** - w tym module została zaimplementowana klasa innej mini gry. Również jak w module *miniGame.py* zawiera algorytmy wyświetlania oraz rozgrywania gry.
- **loot.py** - klasa *Loot* zawarta w module odpowiedzialna jest za generowanie statystyk oraz przedmiotów, które gracz zyskuje po każdej walce w trakcie wyprawy. W zależności od rodzaju przeciwnika i jego siły nagrody generowane są inaczej.
- **cactus.py** - w grze istnieje spersonifikowana postać kaktusa, która w zależności od poziomu dodaje bohaterowi gracza stałe, nienaruszalne bonusy. Klasa zawarta w module przechowuje informacje o wymaganiach, jakie należy spełnić podczas gry, ale również metody aktualizacji, zapisu poziomu postaci otoczenia oraz obsługa dodawania statystyk
- **cityBuff.py** - zawarta w module klasa odpowiada za losowe generowanie jednorazowych bonusów dostępnych do zakupienia przez gracza w odpowiedniej lokacji. Odczytuje co wybrał gracz oraz odpowiednio odejmuje pozyskaną walutę.
- **huntingHut.py** - moduł stworzony do obsługi funkcjonalności lokacji, w której użytkownik może podnieść swoją rangę, poświęcając dotychczas zebrane uznanie, by uzyskać stałe bonusy dla swojej postaci.
- **treasure.py** - moduł ten posiada klasę skarbu, w której zdefiniowane są wszystkie przedmioty dostępne do uzyskania podczas wyprawy. Przedmioty mają ścisłe określona hierarchię wartości, co przekłada się na nierówny procent szansy pozyskania. Obsługuje również automatyczne ulepszenie statystyk, jeżeli przedmioty się ze sobą łączą.

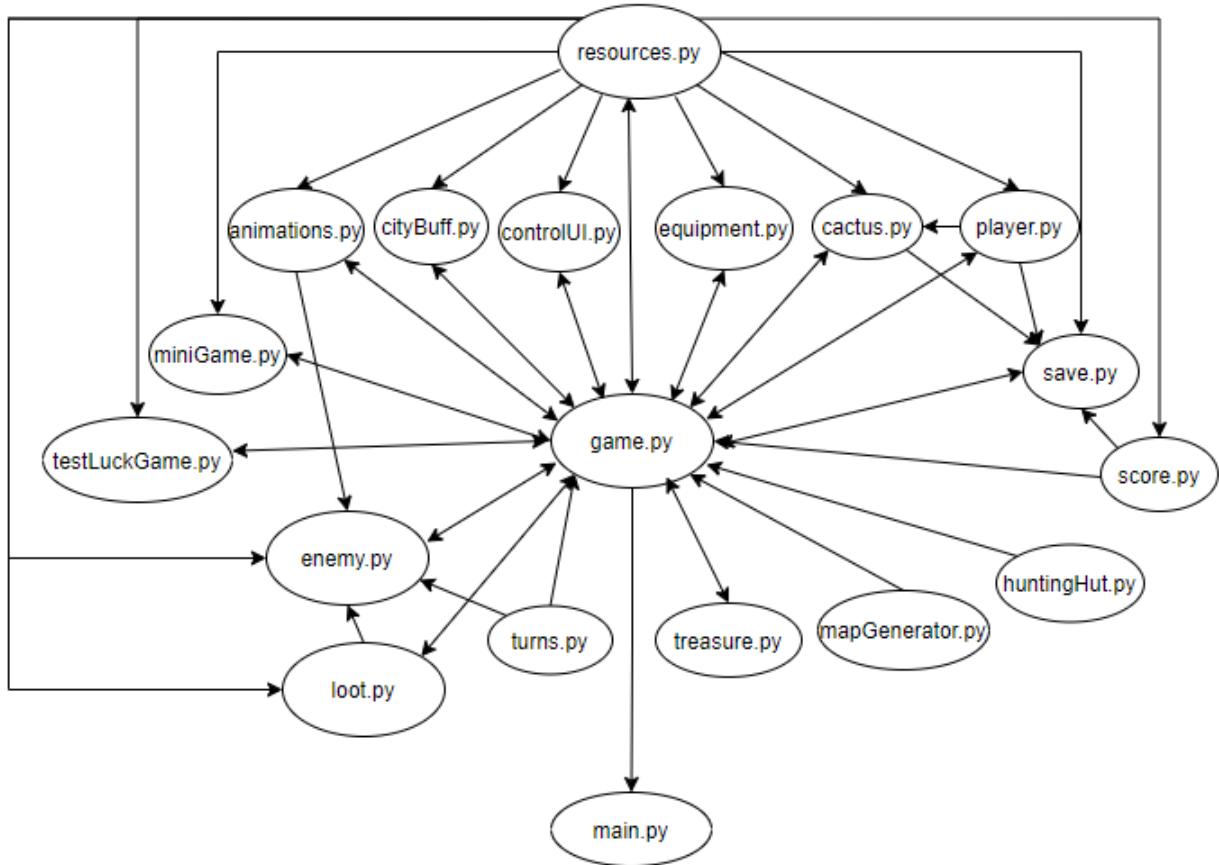
4.4. Architektura

Architektura aplikacji napisanej w języku Python polega na dokładnie zaplanowanej strukturze obiektów oraz odpowiednio przemyślanych interakcjach. Przed rozpoczęciem implementacji ważnym jest, aby zarysy końcowego systemu czy aplikacji były określone i dobrze przeanalizowane.



Rys. 4.4: Diagram przepływu pracy

Architektura aplikacji zaprojektowana została tak, by każda z klas stanowiła oddzielną jednostkę, odpowiedzialną za własną część logiki gry. Dzięki czemu, dowolny moduł może korzystać z usług innego modułu, poprzez globalne odwołanie do jego metod - ogranicza ilość zależności w kodzie oraz zwiększa jego czytelność.



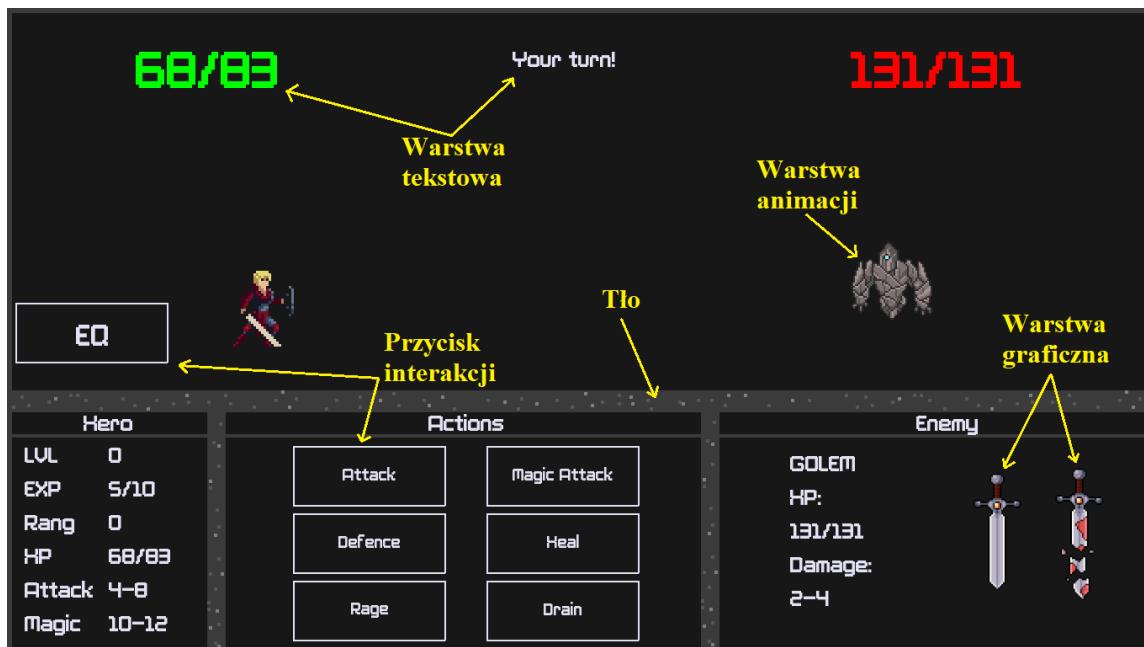
Rys. 4.5: Diagram zależności modułów

Rozdział 5

Implementacja

5.1. Graficzny interfejs użytkownika

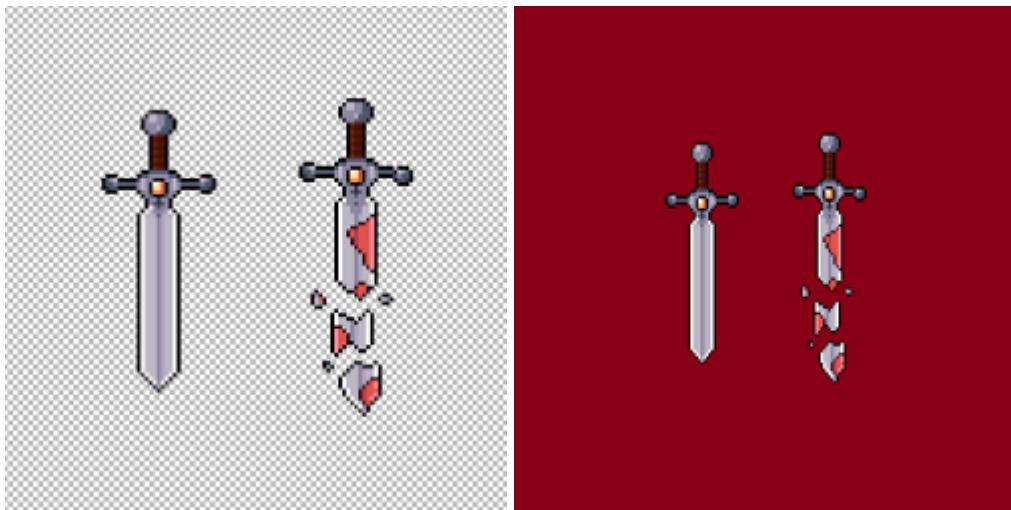
Ważnymi elementami dzisiejszych gier wideo jest grafika oraz łatwość poruszania się w aplikacji. Graficzny interfejs użytkownika (*ang. Graphical User Interface, GUI*) jest to sposób prezentacji danych i informacji przez komputer oraz interakcji z użytkownikiem. Wszystkie elementy interfejsu zaprogramowane w grze *SoulCollector* są wykonane w przejrzysty i uporządkowany sposób, aby zmaksymalizować przyjemność z rozgrywki. Z perspektywy programisty wszystkie pliki graficzne zawarte są w dwóch katalogach - *img* oraz *sprites*, aby utrzymać schludność oraz klarowność kodu. Interakcja użytkownika z programem opiera się wyłącznie na użyciu myszy.



Rys. 5.1: Graficzny interfejs użytkownika

5.1.1. Tła i warstwy

Każda odwiedzona przez gracza lokacja zawiera swoją unikatową grafikę tła oraz nałożone na nią warstwy. Wszystkie pliki graficzne są formatu *.png*, ponieważ są łatwo edytowalne oraz gwarantują bezstratną kompresję danych graficznych. Ważną zasadą tworzenia warstw graficznych jest usunięcie tła warstwy, by nie naruszać obrazu, na który warstwa jest rzutowana.



Rys. 5.2: Warstwa transparentna oraz nietransparentna

Dzięki bibliotece Pygame, w łatwy sposób można wyświetlać grafiki i warstwy oraz edytować ich rozmiar. W celu ograniczenia powtarzalności kodu, zaimplementowana została funkcja widoczna na rysunku 4.3.

Listing 5.1: Funkcja imageSetting

```
def imageSetting(display, pos, size, img):
    tempImg = img
    tempImg = pygame.transform.scale(tempImg, size)
    display.blit(tempImg, pos)
```

Tab. 5.1: Opis funkcji wyświetlania obrazu

imageSetting()	
display	Argument funkcji określający ekran wyświetlania
pos	Argument w postaci nieedytowalnej krotki (<i>.ang tuple</i>), zawierający pozycję wyświetlania w pikselach
size	Argument pozwalający na skalowanie obrazu, zmienna typu float
img	Argument zawierający przypisany obraz w module resources.py

Warstwę tekstową definiuje i lokuje na ekranie funkcja pokazana na rysunku 4.4. Wykorzystuje wcześniej zaimplementowaną czcionkę w rozszerzeniu *.ttf*. Jednakże, aby tekst mógł być wyświetlony, należy wyrenderować czcionkę jako obraz, co uzyskałem wykorzystując moduł *pygame.font*. Dodatkowo, każda wyświetlana czcionka poddawana jest antialiasingowi, czyli wygładzaniu krawędzi.

Listing 5.2: Funkcja textSetting

```
def textSetting(display, pos, font, colour, text):
    img = font.render(text, True, colour)
    display.blit(img, pos)
```

Tab. 5.2: Opis funkcji wyświetlania tekstu

textSetting()	
display	Argument funkcji określający ekran wyświetlania
pos	Argument w postaci nieedytowalnej krotki (<i>.ang tuple</i>), zawierający pozycję wyświetlania w pikselach
colour	Argument pozwalający na określenie koloru tekstu, należy podać jako krotkę wartości RGB
text	Argument zawierający tekst, który ma zostać wyświetlony

Warstwą animacji jest sekwencyjne wyświetlanie po sobie warstw, które przy odpowiedniej ilości klatek na sekundę tworzą iluzję ruchu. Plik zawierający grafikę animacji różni się od pliku wykorzystywanego jako zwykła pojedyncza warstwa - zawierają każdą klatkę, które są umieszczone w transparentnej przestrzeni.



Rys. 5.3: Plik z kolejnymi klatkami ataku przeciwnika

Jednakże wyświetlenie obrazu animacji nie wystarczy, by uzyskać poklatkowy ruch. Pierwszym krokiem było zaprogramowanie klasy *SpriteSheet*, która odpowiedzialna jest za pobieranie z pliku pojedynczej warstwy i rzutowanie jej na powierzchnię o odpowiednich rozmiarach. Kod klasy z opisem przedstawiono w listingu 4.3 oraz w tabeli 4.3.

Listing 5.3: Klasa SpriteSheet

```
class SpriteSheet:
    def __init__(self, image):
        self.sheet = image

    def getImage(self, frame, width, height, scale, colour):
        image = pygame.Surface((width, height)).convert_alpha()
        image.blit(self.sheet, (0, 0), ((frame * width), 0, width, height))
        image = pygame.transform.scale(image, (width * scale, height * scale))
        image.set_colorkey(colour)

    return image
```

Tab. 5.3: Opis klasy SpriteSheet

SpriteSheet	
sheet	Zmienna przechowująca warstwę
frame	Zmienna odnosząca się do pojedynczej klatki warstwy
width	Szerokość klatki warstwy
height	Wysokość klatki warstwy
scale	Zmienna pozwalająca na skalowanie rozmiaru klatki
colour	Wartość RGB tła, na które będzie rzutowana klatka warstwy

Następnie stworzone zostały klasy *PlayerAnimations* i *EnemyAnimations* obsługujące zmianę oraz wyświetlanie klatek. Do każdego z plików graficznych, przypisany zostaje stan, na podstawie którego wyświetlane są odpowiednie animacje, w zależności od wykonywanych przez gracza lub przeciwnika akcji. W dalszej części wywodu posłużę się fragmentem klasy *PlayerAnimations*, aby przybliżyć działanie klas.

Listing 5.4: Klasa PlayerAnimations

```
class PlayerAnimations:
    def __init__(self):
        self.state = 0
        ...
        self.heroAttackAnimation = res.attackPlayerAnimation
        self.attackAnimationList = []
        self.attackAnimationSteps = 22
    ...
    self.attackSpriteSheet = SpriteSheet(self.heroAttackAnimation)
    ...
    self.lastUpdate = pygame.time.get_ticks()
    self.animationCooldown = 100
    ...
    self.attackFrame = 0
    ...
    for x in range(self.attackAnimationSteps):
        self.attackAnimationList.append(
            (self.attackSpriteSheet.getImage(x, 144, 64, 3, (0, 0, 0)))
    ...
def updateAttackAnimation(self):
    currentTime = pygame.time.get_ticks()
    if currentTime - self.lastUpdate >= self.animationCooldown:
        self.attackFrame += 1
        self.lastUpdate = currentTime
        if self.attackFrame >= len(self.attackAnimationList):
            self.attackFrame = 0
            self.state = 0
            game.fightTurn.endTurn()

def displayAttackAnimation(self):
    game.screen.blit(self.attackAnimationList[self.attackFrame], (200, 270))
```

- **state** - Zmienna odpowiedzialna za przechowywanie aktualnego stanu animacji, np. dla animacji ataku wartość zmiennej przyjmuje 1.
- **heroAttackAnimation** - Przypisanie do zmiennej pliku graficznego, zawiera wszystkie kolejne klatki.
- **attackAnimationList** - Jest to tablica, która przechowuje wszystkie pojedyncze klatki animacji
- **attackAnimationSteps** - Zmienna określająca całkowitą ilość klatek dla danej animacji
- **attackSpriteSheet** - Zmienna wykorzystująca klasę *SpriteSheet*, do edycji klatek
- **lastUpdate** - Do zmiennej przypisana jest wartość ostatniej wartości odświeżania aplikacji
- **animationCooldown** - Zmienna określa szybkość wyświetlania następujących klatek po sobie
- **attackFrame** - Określa numerycznie aktualnie wyświetlana klatkę

Przy użyciu pętli for następuje uzupełnienie tablicy każdą pojedynczą klatką, wygenerowaną dzięki metodzie *getImage* klasy pomocniczej *SpriteSheet*. Następnie zdefiniowane są dwie metody obsługujące aktualizację animacji oraz wyświetlanie animacji w ekranie rozgrywki. Metoda *updateAttackAnimation* aktualnia obecnie wyświetlaną klatkę, jeżeli różnica czasów odświeżania aplikacji jest większa lub równa szybkości wyświetlania klatki. W przypadku, gdy obecnie wyświetlana klatka jest ostatnią dostępną klatką, zmiennej *attackFrame* przypisywana jest wartość 0, aby przy kolejnym wyświetaniu animacji rozpocząć od pierwszej dostępnej klatki. Dodatkowo zmieniany jest stan animacji do stanu początkowego oraz następuje zakończenie tury. Metoda *displayAttackAnimation* wyświetla w aplikacji animację w zależności od obecnego stanu zmiennej *attackFrame*.

5.1.2. Interakcja z programem

Ważnym elementem każdej gry jest interakcja z programem, to dzięki niej użytkownik może eksplorować zawartość produktu. W zależności od studia oraz podejścia deweloperów, sterowanie przyjmuje różne postaci, np. w grze *Cuphead* użytkownik posługuje się wyłącznie klawiaturą. W zaimplementowanej przez mnie grze wideo gracz komunikuje się z aplikacją wyłącznie za pomocą myszy, co znacznie zwiększa tempo rozgrywki oraz jej przejrzystość. Jednakże, aby to osiągnąć została zaimplementowana klasa *Button*, która pozwala na wykonanie każdej dostępnej akcji w grze.

Listing 5.5: Klasa Button

```
class Button:
    width = 180
    height = 70

    def __init__(self, displayScreen, x, y, colour,
                 clickedColour, whichFont, isFramed, text):
        self.display = displayScreen
        self.x = x
        self.y = y
        self.colour = colour
        self.clickedColour = clickedColour
        self.whichFont = whichFont
        self.isFramed = isFramed
        self.text = text

    def drawButton(self):
        global clicked
        action = False

        pos = pygame.mouse.get_pos()

        buttonRect = Rect(self.x, self.y, self.width, self.height)

        if self.isFramed:
            pygame.draw.line(self.display, self.colour,
                             (self.x, self.y), (self.x + self.width, self.y), 2)
            pygame.draw.line(self.display, self.colour,
                             (self.x, self.y), (self.x, self.y + self.height), 2)
            pygame.draw.line(self.display, self.colour,
                             (self.x, self.y + self.height),
                             (self.x + self.width, self.y + self.height), 2)
            pygame.draw.line(self.display, self.colour,
                             (self.x + self.width, self.y),
                             (self.x + self.width, self.y + self.height), 2)

        if buttonRect.collidepoint(pos):
            if pygame.mouse.get_pressed()[0] == 1:
                clicked = True
                if self.isFramed:
                    pygame.draw.line(self.display, self.clickedColour,
                                     (self.x, self.y),
                                     (self.x + self.width, self.y), 2)
                    pygame.draw.line(self.display, self.clickedColour,
                                     (self.x, self.y),
                                     (self.x, self.y + self.height), 2)
                    pygame.draw.line(self.display, self.clickedColour,
                                     (self.x, self.y + self.height),
                                     (self.x + self.width, self.y + self.height), 2)
                    pygame.draw.line(self.display, self.clickedColour,
                                     (self.x + self.width, self.y),
                                     (self.x + self.width, self.y + self.height), 2)
                elif pygame.mouse.get_pressed()[0] == 0 and clicked == True:
                    clicked = False
                    action = True

        textImg = self.whichFont.render(self.text, True, self.colour)
        textLen = textImg.get_width()
        self.display.blit(textImg,
                          (self.x + int(self.width / 2) - int(textLen / 2), self.y + 25))
        return action
```

Tab. 5.4: Konstruktor klasy Button

<u>__init__</u> klasy Button	
displayScreen	Zmienna decydująca, na którym ekranie wyświetlony zostanie przycisk
x, y	Zmienne pozycyjne, które określają położenie przycisku na ekranie
colour	Zmienna przyjmuje wartości RGB, od której zależy kolor obramowania przycisku
clickedColour	Zmienna przyjmuje wartości RGB, podczas kliknięcia zmienia kolor przycisku
whichFont	Zmienna przechowująca czcionkę wyświetlaną w przycisku
isFramed	Zmienna typu bool, określająca czy przycisk wyświetla się razem z obramowaniem
text	Zmienna określająca wyświetlany tekst na przycisku

Biblioteka PyGame pozwala na pozyskanie oraz przypisanie do zmiennej pozycji kurSORA w aplikacji, za pomocą metody `get_pos()` w postaci krotki, przechowującej wartość horyzontalną i wertykalną pikseli. Jeżeli obydwie wartości pozycji kurSORa znajdują się w polu przycisku i zostanie naciśnięty lewy przycisk myszy, wykona się pożądana akcja. Przykład wykorzystania klasy z przypisaną akcją przedstawiono w listingu 4.6.

Listing 5.6: Wykorzystanie klasy Button

```
startButton = cui.Button(screen, 50, 350, res.menuTextColour,
                         res.menuTextColourClicked, res.pixelFont,
                         True, 'Play')

...
if startButton.drawButton():
    gameScreen()
```

5.2. Model i logika gry

5.2.1. Zapis oraz wczytywanie gry

Nieodłącznym elementem każdej gry video jest funkcja zapisu oraz wczytania stanu rozgrywki. W dzisiejszych czasach mało kto potrafi przejść grę w całości bez jej wyłączenia, więc możliwość zapisywania wydaje się nieoceniona. Aby dokonać poprawnego zapisu należy przemyśleć, jakie elementy gry powinny być zachowywane w pamięci. W grze SoulCollector wykonanie funkcji zapisu oraz wczytania odbywa się automatycznie tzn. gdy gracz zmienia lokację lub wyłącza grę, a zapisywane elementy gry można wypunktować następująco:

- Statystyki bohatera
- Poziom ulepszenia lokacji miasta
- Najlepsze dziesięć wyników gry

Pliki zapisu znajdują się w katalogu `save`, do którego program ma stały dostęp. Do wykonania funkcji zapisu oraz wczytania stanu rozgrywki wykorzystałem bibliotekę `json`, która pozwala na

odczyt danych z plików tekstowych oraz ich edycję. Poniżej w listingu 4.7 i 4.8, jako przykład przedstawiam funkcję zapisującą i wczytującą statystyki bohatera z pliku.

Listing 5.7: Funkcja zapisu

```
def makeSave():
    game.hero.updateStats()
    with open('save/save.txt', 'w') as heroSave:
        json.dump(pl.heroStats, heroSave)
```

Ważnym elementem podczas nadpisywania pliku tekstowego jest użycie flagi 'w', aby w otwartym pliku móc tylko nadpisywać umieszczone wcześniej dane. Za pomocą metody *dump*, *json* umożliwia wprowadzenie zmiany do wybranego pliku. Jeżeli plik o danej nazwie nie istnieje, zostanie wygenerowany w odpowiednim katalogu.

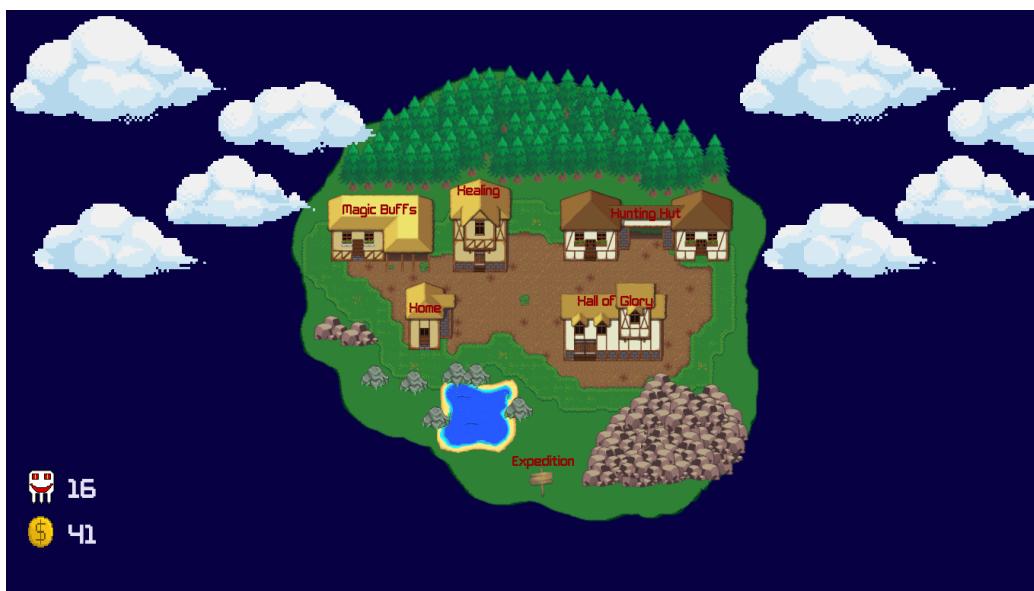
Listing 5.8: Funkcja wczytania

```
def readSave():
    try:
        with open('save/save.txt') as heroSave:
            pl.heroStats = json.load(heroSave)
    except:
        print('No such a file or directory yet.')
```

W przypadku pobierania danych z pliku tekstowego, program nadpisuje zmienną w programie przy pomocy metody *load*. Jednakże, gdy nie istnieje żaden plik o podanej nazwie, program wyświetla w konsoli komunikat, lecz nie zatrzymuje działania gry.

5.2.2. Pętle aplikacji

Zaprogramowana aplikacja, reprezentowana jest przez wiele pętli, które obsługują wyświetlane lokacje oraz ich funkcjonalności. Dzięki wykorzystanym pętlom *while* możliwym jest zmiana stanu gry oraz widoku rozgrywki. Główną pętlą programu jest *startScreen*, od której rozpoczyna się rozgrywkę. Umieszczone na ekranie przyciski i ich wyzwalanie pozwala na przejście do pętli właściwej rozgrywki lub opcji. Jako przykładem, posłużę się lokacją miasta, która umożliwia graczowi rozwój postaci czy sprawdzenie najlepszych wyników.



Rys. 5.4: Widok pętli *gameScreen()* w aplikacji

Listing 5.9: Przedstawienie widoku miasta w postaci kodu

```

def gameScreen():
    ...
    while True:
        screen.blit(res.bgCity, (0, 0))
        res.balance(screen)

        if expeditionButton.drawButton():
            spawnMap()
        if homeButton.drawButton():
            homeScreen()
        if magicButton.drawButton():
            cityBuffs()
        if healingButton.drawButton():
            healingCentre()
        if gloryButton.drawButton():
            glory()
        if huntingHutButton.drawButton():
            huntHut()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                save.makeSave()
                pygame.quit()
                sys.exit()

        pygame.display.update()
        gameClock.tick(60)

```

W przybliżonym listingu 4.9, pętla while odpowiedzialna jest za ciągłe wyświetlanie tła lokacji oraz dostępnych interakcji, dzięki którym gracz może swobodnie zmieniać stan gry. Dla przykładu, po naciśnięciu przycisku wyruszenia na ekspedycję, automatycznie otwierana jest pętla mapy, umożliwiająca dalsze eksplorowanie (Rys. 3.3). Każda z zaimplementowanych pętli znacząco różni się od siebie objętością oraz funkcjonalnościami - pętla walki zawiera obsługę interakcji gracza, algorytmy obsługujące zachowanie przeciwników czy animacje walki, natomiast pętla umożliwiająca sprawdzenie najlepszych wyników, jedynie wczytuje i wyświetla punkty zdobyte podczas najlepiej rozegranych gier.

Listing 5.10: Pętla obsługująca wyświetlanie wyników

```

def glory():
    run = True
    ...
    while run:
        screen.fill((0, 0, 0))
        screen.blit(res.paper, (0, 0))
        res.textSetting(screen, (480, 50), res.pixelTitleFont,
                        res.menuTextColour, 'HALL OF GLORY')
        score.gloryHallTable(screen)

        if exitButton.drawButton():
            run = False
        ...

```

5.2.3. Generowanie mapy

Charakterystycznym elementem rozgrywki gier typu roguelike jest losowo generowana mapa świata. Jednakże, losowość kreowania mapy jest odgórnie ustalona, by zachować odpowiednią logikę rozgrywki. W przypadku gry SoulsCollector, schemat generowania jest ściśle określony. Oto kilka zasad, które definiują mapę podczas rozgrywki:

- Podczas rozgrywki dostępne są dla gracza dwie lokacje ogniska, dzięki którym gracz uzupełnia część zdrowia bohatera
- Jedno z ognisk musi znajdować się przed finałową walką z przeciwnikiem ostatecznym
- Lokacja finałowego przeciwnika musi znajdować się na końcu ścieżki
- Gracz podczas wyprawy może znaleźć tylko jedną lokację ze skarbem
- Gracz podczas wyprawy może rozegrać tylko jedną mini grę
- Mini gry na mapie wybierane są losowo
- Mapa zawsze zawiera pięć walk ze zwykłymi przeciwnikami

Aby przybliżyć proces tworzenia mapy od strony programistycznej, opiszę działanie generatora na podstawie poniższego listingu 4.11. Pierwszym krokiem generowania jest dodawanie do tablicy ośmiu lokacji walki, następnie losowa lokacja między drugą, a siódmą zamieniana jest na lokację ogniska. Za pomocą pętli while, losowo wybrana lokacja walki zamieniana jest na lokację mini gry oraz na lokację, w której gracz pozyskuje przedmioty. Poprzez sprawdzenie ilości lokacji mini gry oraz skarbu, decyduje się o maksymalnej ilości obu lokacji. Ostatcznym krokiem jest uzupełnienie ostatnich dwóch lokacji - przedostatnią jest lokacja ogniska oraz ostatnią finałowy przeciwnik.

Listing 5.11: Generator mapy

```
class Generator(object):
    def __init__(self):
        self.luckTest = True
        self.treasure = True
        self.stages = []

    for i in range(0, 8):
        self.stages.append('Fight')

    x = random.randint(1, 6)
    if self.stages[x] == 'Fight':
        self.stages[x] = 'Bonfire'

    while self.luckTest:
        x = random.randint(0, 7)
        if self.stages[x] == 'Fight':
            self.stages[x] = 'Luck Test'
        else:
            pass

        if self.stages.count('Luck Test') == 1:
            self.luckTest = False

    while self.treasure:
        x = random.randint(0, 7)
        if self.stages[x] == 'Fight':
            self.stages[x] = 'Treasure'
```

```

        else:
            pass

        if self.stages.count('Treasure') == 1:
            self.treasure = False

        self.stages.append('Bonfire')
        self.stages.append('Boss')
    
```

5.2.4. Aktualizacja wyników

Wyniki pozyskane przez gracza w każdej rozgrywce są zapisywane w tablicy o rozmiarze ściśle określonym. Pierwszemu indeksowi tablicy przypisuje się największy wynik, natomiast ostatni najmniejszy. Dzięki czemu, podczas wyświetlania dziesięciu najlepszych wyników są one zapisane od góry do dołu na ekranie aplikacji. Aby pozyskany wynik był wyświetlany, musi być większy niż ostatni element tablicy wyników. W przypadku, gdy uzyskany wynik jest większy, plasuje się w odpowiednim indeksie, a wynik najmniejszy usuwany jest z listy.

Algorytm quicksort

Do sortowania uzyskanych wyników użyto algorytmu sortowania *quicksort*. Jest to algorytm sortowania działający na zasadzie "dziel i zwyciężaj". Zasada ta polega na dzieleniu rekurencyjnie problemu na dwa lub więcej mniejszych podproblemów, aż do momentu uzyskania prostych fragmentów, które prowadzą do bezpośredniego rozwiązania. W przypadku rozwiązań podproblemów, łączy się je w całość, by uzyskać rozwiązanie całego zadania.

Listing 5.12: Algorytm quicksort

```

def partition(arr, low, high):
    i = low - 1
    pivot = arr[high]

    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
    
```

Funkcja *partition* wybiera ostatni element jako osią (*ang. pivot*) i umieszcza go w odpowiednim miejscu. Następnie wszystkie mniejsze od osi elementy, umieszczone są po lewej stronie od osi, natomiast większe po prawej. Funkcja *quickSort* z początku sprawdza czy tablica elementów jest większa od 1, w przeciwnym wypadku zwraca tablicę jednoelementową. Jeżeli tablica nie jest jednoelementowa, po sprawdzeniu warunku czy indeks początkowy jest mniejszy od indeksu końcowego, wykorzystuje funkcję partycji i przypisuje do zmiennej indeks partycjonowania. Ostatnim etapem jest osobne sortowanie elementów przed i po partycjonowaniu.

Listing 5.13: Aktualizacja wyników

```

def updateScoreList(score):
    if len(allScores) >= 10:
        if score > allScores[-1]:
            allScores[-1] = score
        else:
            pass
    else:
        allScores.append(score)

    n = len(allScores)
    quickSort(allScores, 0, n - 1)
    allScores.reverse()

```

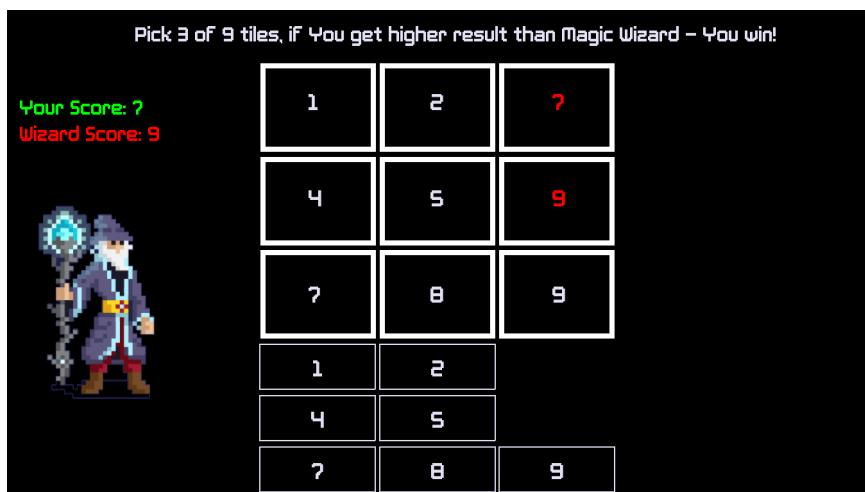
Zaimplementowana funkcja *updateScoreList* przedstawiona w listingu 4.13, odpowiedzialna jest za zachowanie hierarchii wyświetlanych wyników. W przypadku, gdy tablica wyników składa się z dziesięciu elementów i zostaje dodany nowy wynik, większy niż ostatni element tablicy, następuje zamiana wartości dla ostatniego indeksu. Następnie tablica wyników jest sortowana wyżej opisonym algorytmem i odwracana, by indeks początkowy zawierał najwyższy uzyskany wynik.

5.2.5. Mini gry

Wiele firm deweloperskich wyposaża swoje duże produkcje o mniejsze formy rozgrywki, aby zapewnić użytkownikom dodatkowe urozmaicenie. Najczęściej, mini gry przyjmują budowę logicznej zagadki, przeskody lub szczęśliwego trafu, gwarantując dodatkowe bonusy za pomyślne ich ukończenie. W wykonywanym przeze mnie projekcie, dostępne są do użytku dwie mini gry, dzięki którym gracz może pozyskać dodatkową walutę.

Spróbuj szczęścia

Pierwszą zaimplementowaną mini grą jest *Spróbuj szczęścia*, czyli gra opierająca się na naprzemiennej odkrywaniu pól, które posiadają losowo generowane wartości z przedziału od 0 do 9. Każdy z graczy wybiera trzy pola, z których wyniki się sumują. W zależności od uzyskanego wyniku przyznawana jest nagroda, natomiast wynik komputera również ma znaczenie, ponieważ wartość nagrody rośnie wraz z przewagą punktową nad przeciwnikiem. Jednakże, w przypadku przegranej gracz musi utracić ustaloną wartość zasobu.



Rys. 5.5: Mini gra Spróbuj szczęścia

Połącz cztery

Mini gra *Połącz cztery* została zaimplementowana na podstawie gry planszowej *Connect 4*. Jest to gra turowa, w której gracze naprzemiennie wrzucają żetony do pionowo ustawionej planszy o wymiarach 6x7. Celem gry jest połączenie czterech żetonów tego samego koloru w rzędzie, kolumnie lub diagonalnie. W przeciwieństwie do mini gry Spróbuj szczęścia, Połącz cztery jest bardziej złożoną grą, do której obsługi został zaimplementowany algorytm min-max, który polega na minimalizowaniu maksymalnych strat.

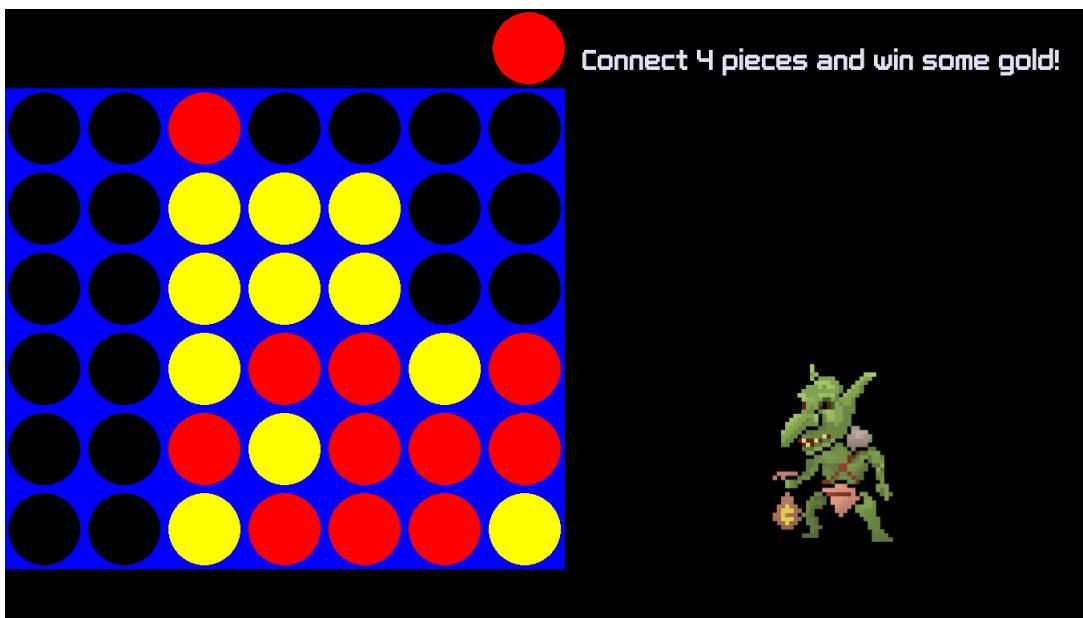
Listing 5.14: Algorytm min-max

```
def minimax(board, depth, alpha, beta, maximizingPlayer):
    validLocations = getValidLocations(board)
    isFinal = isFinalNode(board)
    if depth == 0 or isFinal:
        if isFinal:
            if winMove(board, comPiece):
                return None, 1000000000000000
            elif winMove(board, playerPiece):
                return None, -1000000000000000
            else: # Game is over, no more valid moves
                return None, 0
        else: # Depth is zero
            return None, scorePos(board, comPiece)
    if maximizingPlayer:
        value = -math.inf
        column = random.choice(validLocations)
        for col in validLocations:
            row = nextRow(board, col)
            boardCopy = board.copy()
            pieceDrop(boardCopy, row, col, comPiece)
            newScore = minimax(boardCopy, depth - 1, alpha,
                               beta, False)[1]
            if newScore > value:
                value = newScore
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else: # Minimizing player
        value = math.inf
        column = random.choice(validLocations)
        for col in validLocations:
            row = nextRow(board, col)
            boardCopy = board.copy()
            pieceDrop(boardCopy, row, col, playerPiece)
            newScore = minimax(boardCopy, depth - 1, alpha,
                               beta, True)[1]
            if newScore < value:
                value = newScore
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value
```

Zaimplementowany algorytm min-max w listingu 4.14, zwraca wartość heurystyczną dla węzłów końcowych i węzłów z maksymalną głębokością wyszukiwania. Wartość heurystyczna jest

wynikiem mierzącym przychylność węzła dla gracza maksymalizującego, stąd węzły z korzystniejszą wartością, np. wygrana są dla gracza korzystniejsze niż dla gracza minimalizującego. Wartością heurystyczną dla końcowych węzłów jest wygrana, przegrana lub remis dla gracza maksymalizującego. Dodatkowo, zastosowany został algorytm alfa-beta, który jest algorymem przeszukującym. Jego celem jest zmniejszenie liczby węzłów ocenianych przez algorytm min-max w drzewie wyszukiwania. Ocena ruchu zostaje zaprzestana, gdy zostaje znaleziona możliwość gorsza niż wykonana w poprzednim badanym ruchu, co wpływa na ostateczną decyzję podczas rozgrywki.



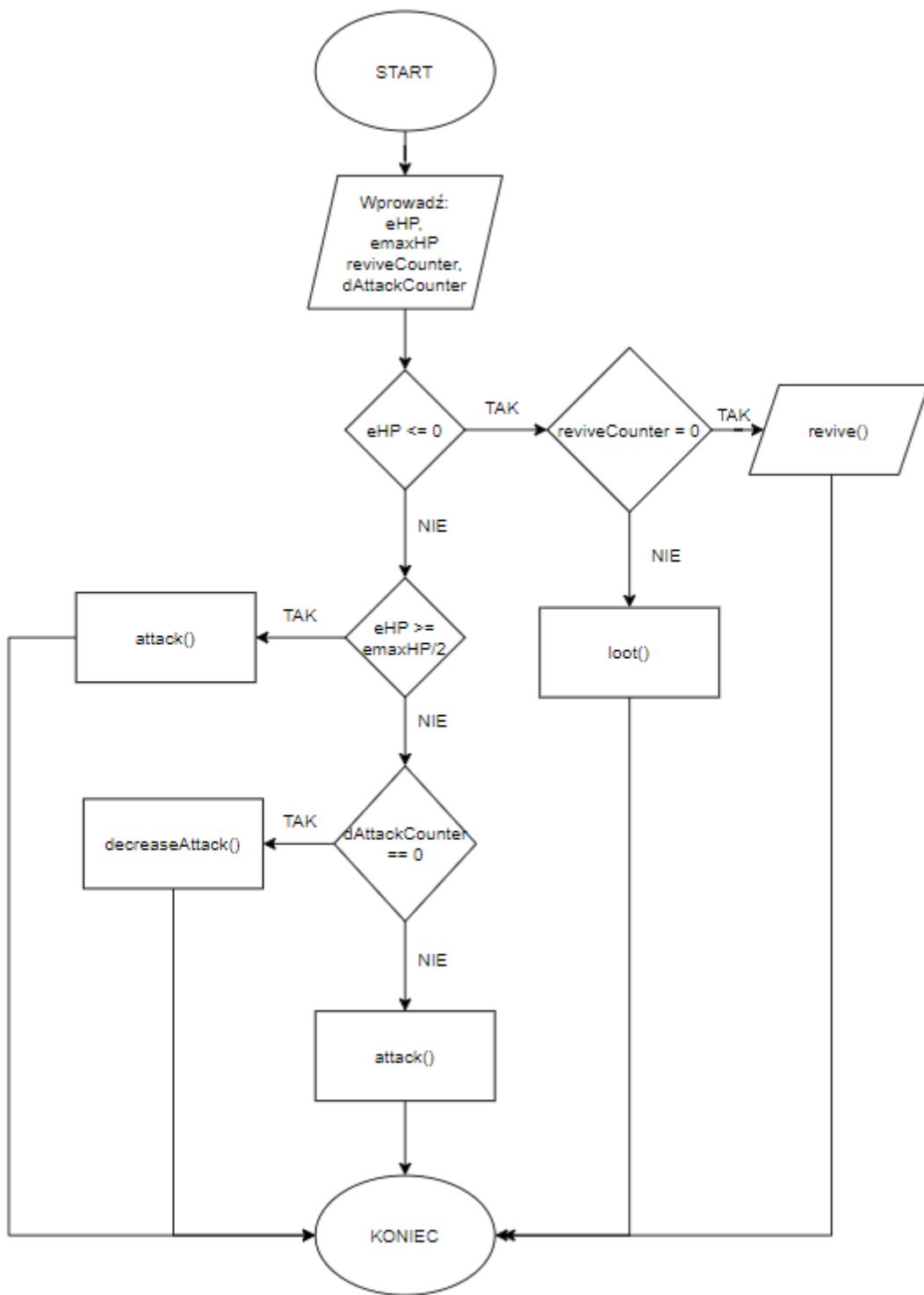
Rys. 5.6: Mini gra Połącz cztery

5.2.6. Algorytmy przeciwników

Podczas korzystania z aplikacji SoulCollector, gracz mierzy się z dwoma rodzajami przeciwników - podstawowymi oraz finałowymi. Podstawowi przeciwnicy charakteryzują się niższymi wartościami statystyk i prostszymi umiejętnościami, co czyni je łatwiejszymi do pokonania, natomiast przeciwnicy finałowi są ulepszoną wersją zwykłych oponentów. Jednakże, gra nie podaje graczu informacji o posiadanych przez przeciwnika umiejętnościach, jak w wielu grach z gatunku roguelike, zabieg ten wynosi grę na wyższy poziom trudności. Jedynymi danymi wyświetlonymi na ekranie rozgrywki jest poziom życia oraz przedział zadawanych obrażeń, co skutkuje niepewnością w skrajnych sytuacjach - gracz musi podejmować trudne decyzje, które mogą doprowadzić go do klęski. Poniżej przedstawiono wszystkich dostępnych przeciwników i ich umiejętności.

Szkielet

Jednym z podstawowych przeciwników jest Szkielet, który jako jedyny charakteryzuje się umiejętnością wskrzeszania oraz zmniejszania obrażeń zadawanych przez bohatera o 25%. Wspomniane umiejętności wykorzystywane są przez przeciwnika tylko raz na całą walkę - Szkielet nie może się wskrzeszać nieskończoną ilość razy, a obniżenie wartości ataku bohatera trwa od momentu rozpoczęcia do końca walki. Jak każdy przeciwnik, po swojej porażce gwarantuje bohaterowi nagrody w postaci doświadczenia, złota oraz dusz, potrzebnych do dalszego rozwoju postaci. Poniżej został przedstawiony schemat blokowy algorytmu walki na rysunku 4.7.

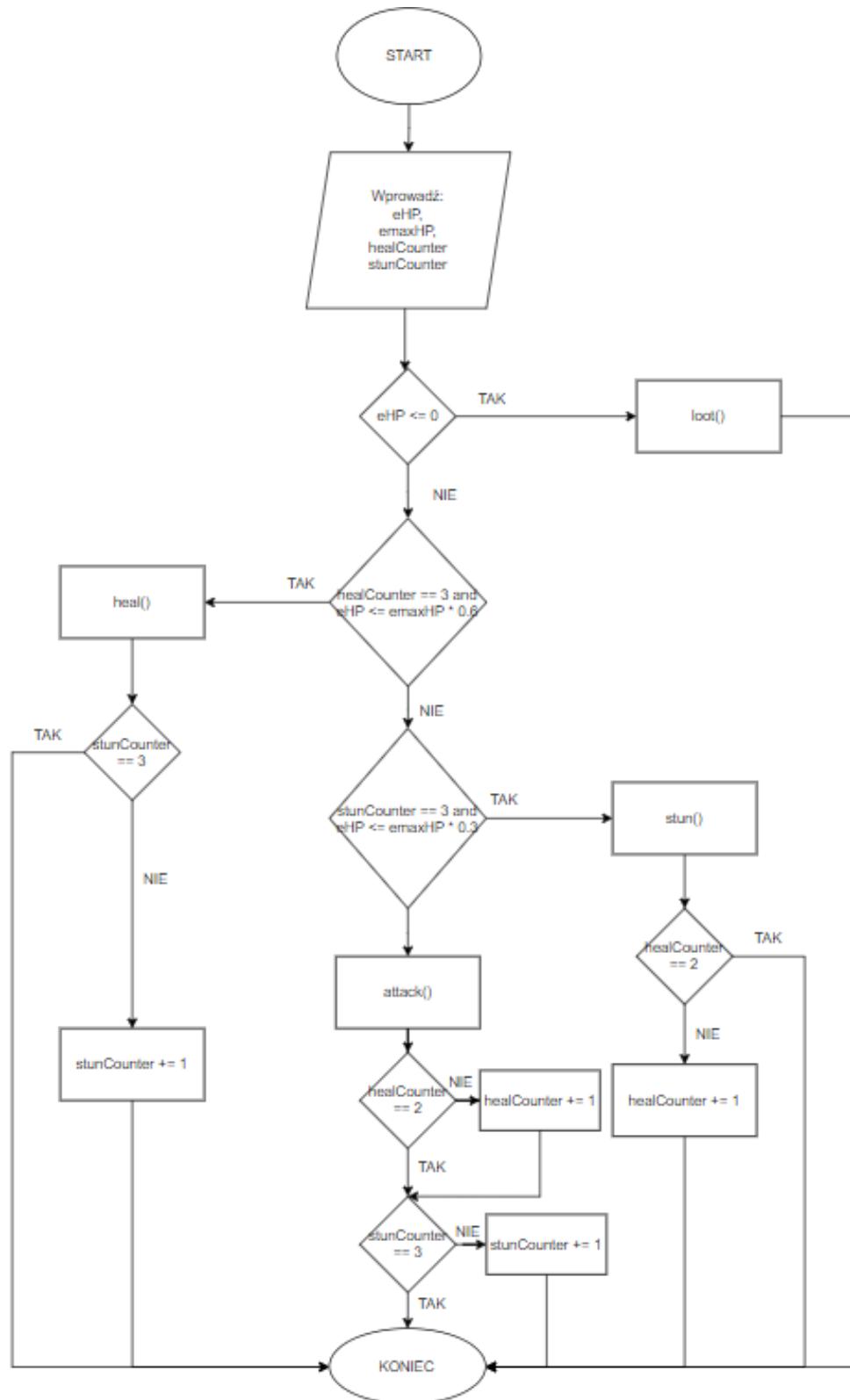


Rys. 5.7: Szkielet - algorytm walki

Minotaur

Kolejnym przeciwnikiem jest Minotaur, który jest bardziej wytrzymałą jednostką o większej sile ataku. Dysponuje dwoma umiejętnościami - leczeniem, które zwraca część straconych punktów życia oraz ogłuszeniem, niepozwalające graczowi przez kilka tur wykonać ruchu. Jednakże, zostały wprowadzone ograniczenia turowe, niepozwalające przeciwnikowi na ciągłe używanie umiejętności specjalnych. Minotaur jest przykładem przeciwnika wielofazowego, co oznacza,

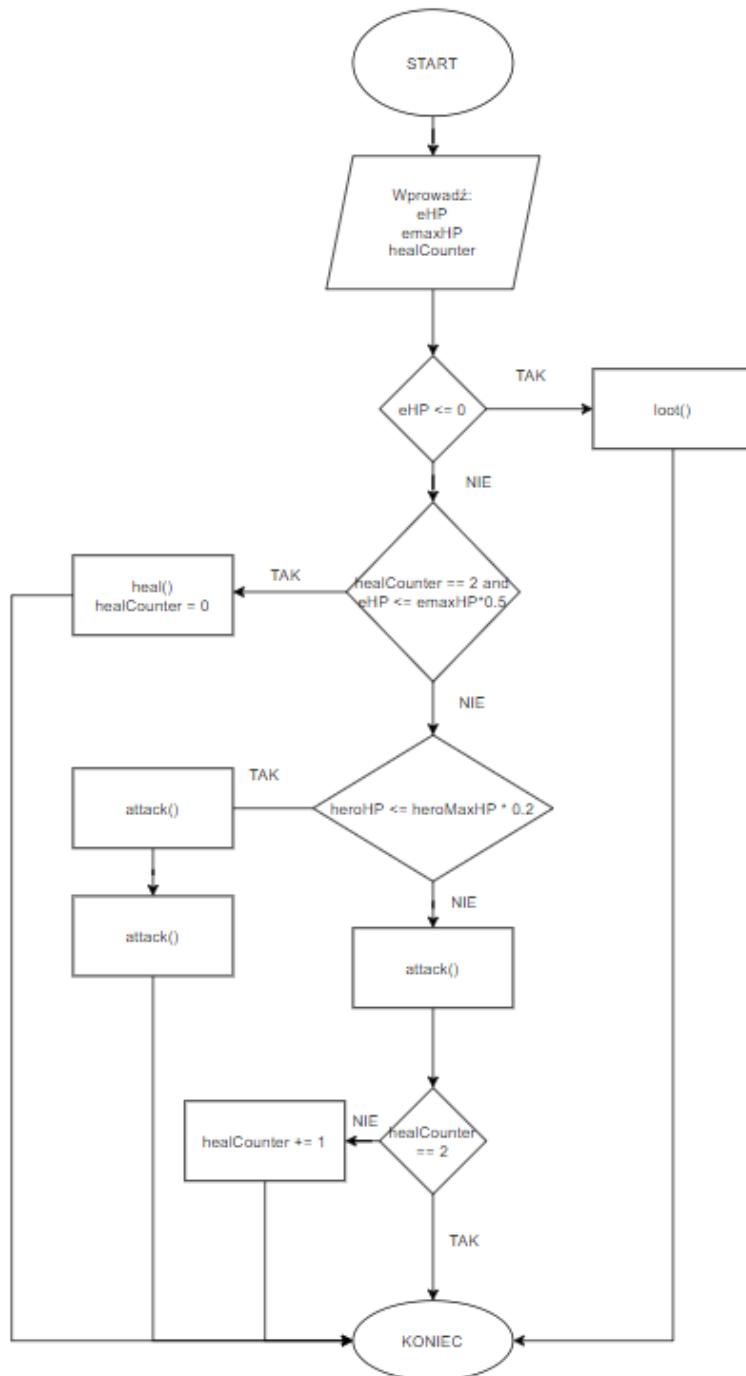
że umiejętności dostępne są do użytku w momencie spełnienia warunku. Warunkiem koniecznym do zastosowania umiejętności leczenia jest wartość obecnego stanu zdrowia mniejsza niż 60% maksymalnych punktów życia, natomiast dla umiejętności ogłuszenia jest to 30% maksymalnych punktów życia. Poniżej, na rysunku 4.8 przedstawiono schemat blokowy algorytmu walki.



Rys. 5.8: Minotaur - algorytm walki

Wiedźma

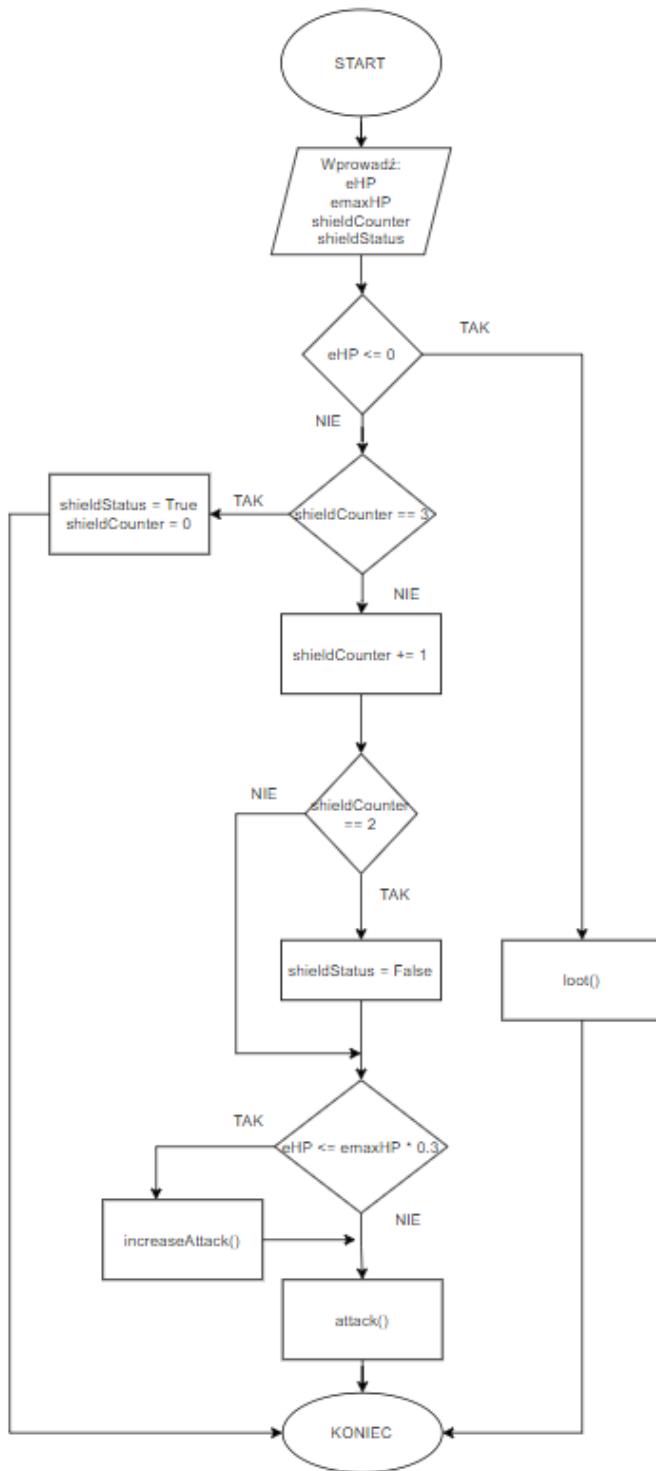
Trzecim z podstawowych przeciwników jest Wiedźma, która charakteryzuje się małą ilością punktów życia, natomiast sporymi wartościami ataku. Posiada tylko jedną umiejętność, którą jest leczenie - w porównaniu do jednostki Minotaura, może używać jej z większą częstotliwością, jednakże warunkiem użycia jest poziom punktów życia poniżej 50% maksymalnych punktów. Dodatkowo, wiedźma wykonuje podwójny atak, gdy punkty życia bohatera kontrolowanego przez gracza spadną poniżej 20%, co zazwyczaj skutkuje dla gracza porażką. Aby przybliżyć przebieg walki wiedźmy, poniżej na rysunku 4.9 przedstawiono schemat blokowy działania algorytmu.



Rys. 5.9: Wiedźma - algorytm walki

Golem

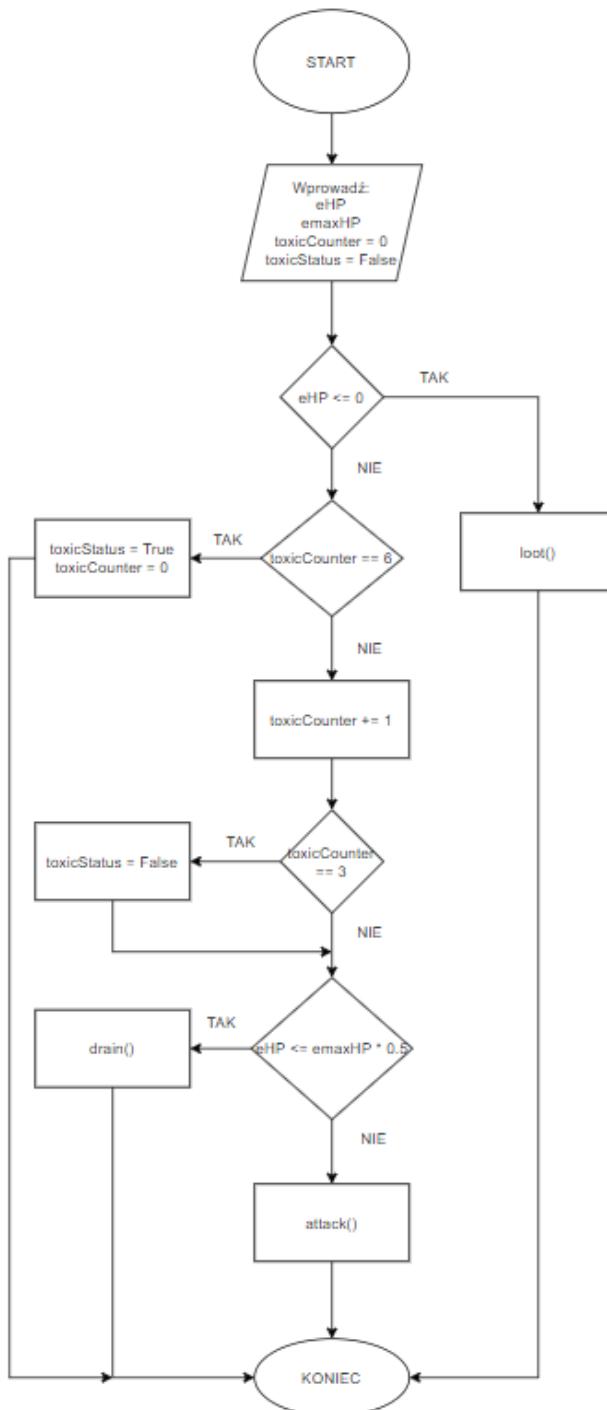
Najbardziej defensywnym przeciwnikiem w grze SoulCollector jest Golem, jest przeciwnieństwem opisanej wyżej wiedźmy. Posiada wysoki poziom początkowych punktów życia, jednakże niski przedział zadawanych obrażeń. Główną umiejętnością przeciwnika jest tarcza, która absorbuje wszystkie otrzymywane obrażenia przez następne dwie tury. W przypadku, gdy poziom punktów życia przeciwnika spadnie poniżej 40%, zyskuje on bonus zwiększający statystyki obrażeń, co czyni go trudnym do pokonania przy niskich wartościach zdrowia. Poniżej przedstawiono algorytm walki na rysunku 4.10.



Rys. 5.10: *Golem - algorytm walki*

Zabójca

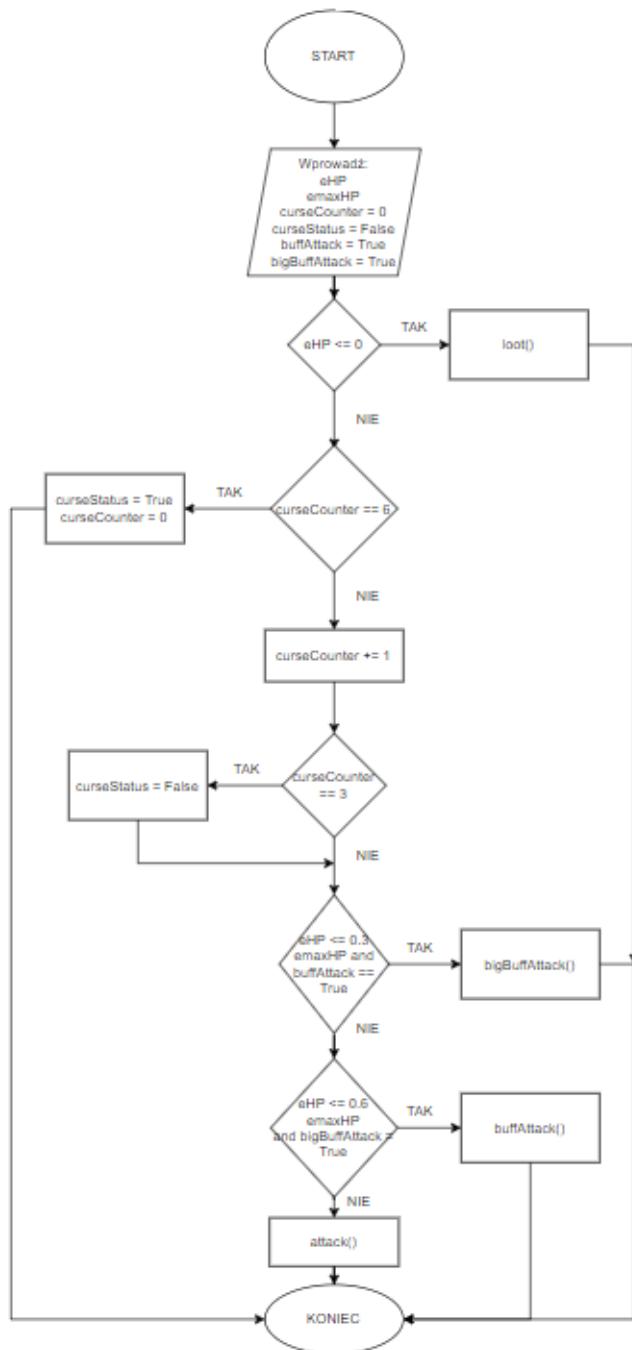
Ostatnim podstawowym przeciwnikiem jest Zabójca, który posiada średnią ilość punktów życia oraz średni przedział wartości ataku. Jednostka zabójcy charakteryzuje się dwiema umiejętnościami, pierwszą z nich jest zatrucie, które obniża poziom życia bohatera o zadaną wartość przez kolejne trzy tury - są to obrażenia, których nie można zmniejszyć lub zabsorbować. Drugą umiejętnością specjalną jest drenaż życia, który polega na zadaniu bohaterowi obrażeń i uleczeniu o tą samą wartość punktów życia przeciwnika. Warunkiem użycia drenażu życia jest poziom punktów zdrowia przeciwnika poniżej 50% maksymalnych punktów żywotności. Na rysunku 4.11 przedstawiono algorytm walki zabójcy.



Rys. 5.11: Zabójca - algorytm walki

Czarnoksiężnik

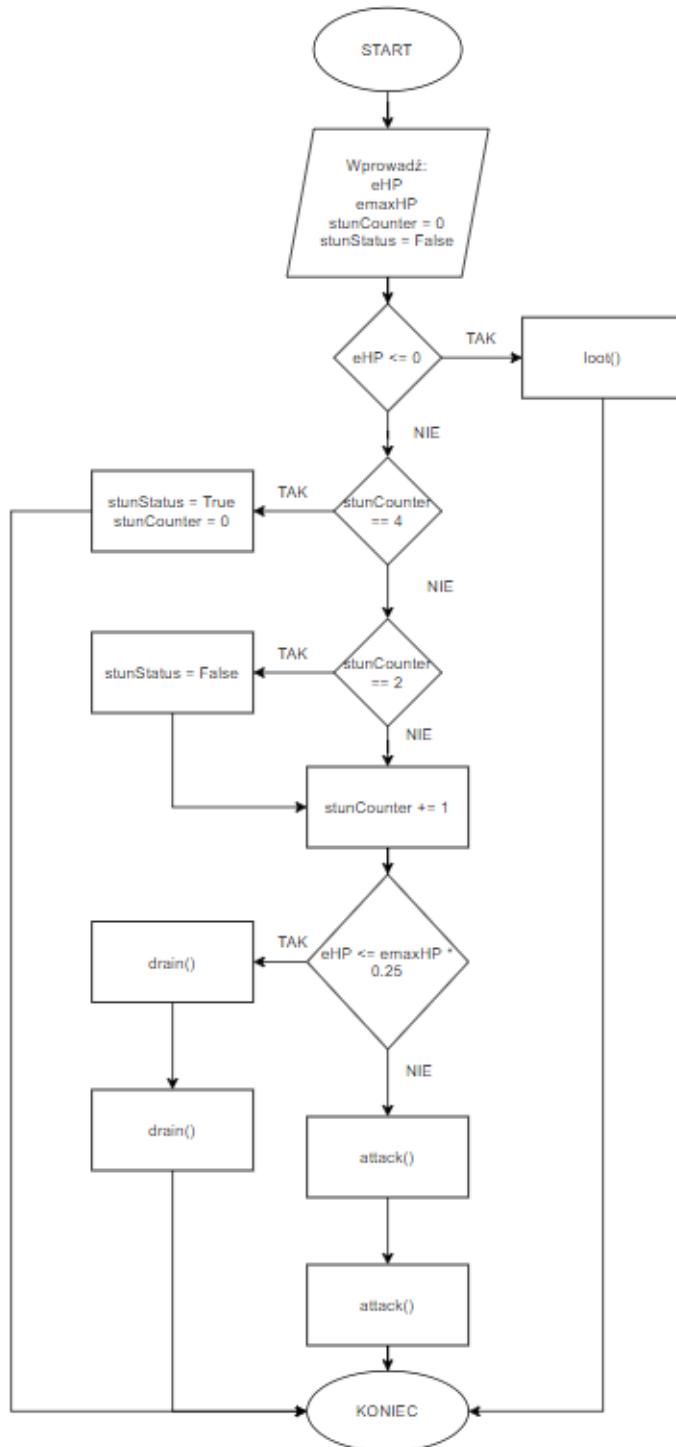
Czarnoksiążnik jest jednym z dwóch finałowych przeciwników w grze SoulsCollector, który posiada duże ilości życia oraz średni przedział wartości ataku. Jednostka posiada trzy umiejętności specjalne, pierwszą z nich jest klątwa, która tak jak w przypadku zabójcy, obniża poziom życia bohatera o zadaną wartość przez kolejne trzy tury. W sytuacji, gdy punkty żywotności przeciwnika będą mniejsze niż 60%, aktywuje się wzmacnienie pierwszego stopnia, minimalnie zwiększające zadawane obrażenia. Natomiast, gdy poziom zdrowia przeciwnika będzie mniejszy niż 30%, aktywuje się wzmacnienie drugiego stopnia, gwarantujące przeciwnikowi spory bonus do zadawanych obrażeń. Obydwa wzmacnienia łączą się ze sobą, co znacznie zwiększa poziom trudności walki. Schemat blokowy algorytmu walki Czarnoksiążnika przedstawiono na rysunku 4.12.



Rys. 5.12: Czarnoksiążnik - algorytm walki

Wojownik nocy

Ostatnim finałowym przeciwnikiem jest Wojownik nocy, posiada średnie ilości życia oraz wyższy przedział wartości ataku. Podobnie jak Minotaur oraz Zabójca, wykorzystuje umiejętność ogłuszenia, jednakże ogłusza bohatera na mniejszą ilość tur oraz umiejętność drenażu życia. W odróżnieniu od pozostałych przeciwników, Wojownik nocy zawsze atakuje podwójnie. W przypadku, gdy poziom punktów życia jest niższy niż 25% maksymalnych punktów, zastępuje zwykły atak drenażem życia. Poniżej przedstawiono schemat blokowy algorytmu walki Wojownika nocy na rysunku 4.13.



Rys. 5.13: Wojownik nocy - algorytm walki

Rozdział 6

Podsumowanie

6.1. Wnioski

Praca nad implementacją gry wideo rozpoczęła się we wrześniu 2021 roku i planowana była na okres dwóch miesięcy. Mimo wyznaczonych terminów, gra była ciągle rozwijana i udoskonalana, nawet podczas pisania pracy dyplomowej. Powodem takiego stanu rzeczy jest nadzwyczaj uzależniający proces tworzenia gier, który obecnie jest znacznie atrakcyjniejszy niż samo użytkowanie aplikacji. Dodatkowo, chcąc ciągłego rozwoju programu nie pozwalała mi na podjęcie decyzji o ukończeniu tytułu. Oddając się retrospekcji stwierdzam, że należało się trzymać przyjętych na początku projektu celów, ponieważ wielokrotnie pod naciskiem własnej ambicji zmieniałem koncepcję gry oraz sposób jej rozgrywki.

Ważnym elementem implementacji kodu aplikacji była kontrola wersji. Przed każdym rozpoczęciem pracy z nowymi funkcjonalnościami, zapisywałem stan kodu w serwisie GitHub¹, co było zbawienne w momencie, gdy zaimplementowany kod okazał się być bezużytecznym lub uszkadzał działający program. Oprócz kontroli wersji, personalnie ważnym było, abym mógł pracować z tworzonym kodem na wielu urządzeniach, ponieważ niejednokrotnie sytuacja tego wymagała.

W niniejszej pracy zależało mi na przybliżeniu charakterystyki gatunku roguelike oraz przedstawieniu mojej koncepcji programowania gier. Dokonałem opisu projektu, w którym przedstawiłem intencje i główne założenia dotyczące gry SoulCollector. W kolejnych rozdziałach przybliżyłem metody projektowania animacji, interakcji z programem oraz logikę gry. Uważam, że udało mi się ukazać czytelnikowi tematykę sztucznej inteligencji w grach komputerowych oraz przedstawić przyjemną w użytkowaniu bibliotekę PyGame.

Aktualnie gra jest niemal skończona pod względem programistycznym, więc jej zawartość nie będzie ulegała wielu zmianom. Jednakże, niektóre elementy gry tj. fabuła oraz dźwięki, wymagają dodania i poprawek, by gra w przyszłości była wydana na rynek gier komputerowych. Platformą docelową w założeniach projektu był komputer osobisty, natomiast w przyszłości planuję przenieść grę na system Android i zmienić jej interfejs, by odpowiadał wymaganiom użytkowników tego systemu. Obecnie jest to pomysł płynący z przyszłości, jednakże biorąc pod uwagę wyraźne zainteresowanie rynkiem gier mobilnych, system Android może być ważnym punktem przyszłych działań.

¹GitHub - serwis internetowy przeznaczony dla projektów wykorzystujących system kontroli Git, oferuje hosting programów open source oraz prywatnych repozytoriów.

Literatura

- [1] **Ian Millington, John Funge** *Artificial Intelligence for Games Second Edition*
6 Sierpień 2009
- [2] **Eike F. Anderson** *Playing Smart - Artificial Intelligence in Computer Games*
1 Styczeń 2003
- [3] **Georgios N. Yannakakis, Julian Togelius** *Artificial Intelligence and Games*
26 Styczeń 2018
- [4] **David L. Craddock** *Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Changed the Course of Video Games*
5 Sierpień 2015
- [5] **Raphael Holzer** *Pygame tutorial Documentation*
17 Czerwiec 2021
- [6] **Jeremy Parish** <https://www.usgamer.net/articles/the-gateway-guide-to-roguelikes>
6 Kwiecień 2015
- [7] **Dorian Lazar** <https://towardsdatascience.com/understanding-the-minimax-algorithm-726582e4f2c6>
15 Wrzesień 2020