AKADEMIA TARNOWSKA
Wydział Politechniczny

Kierunek: ***Informatyka***

2024/2025

Magdalena Kłósek

PRACA INŻYNIERSKA

Projekt i implementacja gry przygodowej

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

Spis treści **3**
1. Wstęp **5**
1.1 Cel projektu 5
1.2 Zakres prac 5
1.3 Grupa docelowa 5
1.4 Analiza rynku i konkurencji 6
2. Analiza biznesowa **7**
2.1 Wymagania funkcjonalne 7
2.2 Wymagania niefunkcjonalne 7
3. Opis wykorzystanych technologii **9**
3.1 Silnik Godot 9
3.2 GDScript 9
3.3 SAV 10
3.4 JSON 12
3.5 Zasoby Graficzne 13
3.6 Zasoby Dźwiękowe 14
4. Implementacja **15**
4.1 Architektura 15
4.2 Otrzymywanie obrażeń przez gracza 16
4.3 Wykrywanie gracza przez przeciwnika 18
4.4 Interakcja ze skrzynią 20
4.5 Mechanizm otwierania bramy do następnego poziomu. 22
4.6 Mechanizm płytki naciskowej 24
5. Interfejs **27**
5.1 Główne menu 27
5.2 Pasek zdrowia 28
5.3 Menu pauzy 28
5.4 Ekwipunek 29
5.5 Ekran śmierci 30
5.6 Ekran ukończenia gry 31
6. Elementy interaktywne **32**
6.1 Przedmioty 32
6.2 Mechanizm płytki naciskowej 32
6.3 Skrzynia 34
6.4 Dekoracje 35
6.5 Sekwencja runiczna 36
6.6 Portal do następnego poziomu 37
7. Testowanie **39**

7.1 Scenariusze testowe 39
8. Podsumowanie i wnioski.....46
Bibliografia 47
Spis rysunków.....48
Spis tabel 51

1. Wstęp

1.1 Cel projektu

Celem projektu jest implementacja gry łączącej elementy zagadkowe i zręcznościowe, osadzone w mrocznym świecie jaskiń. Gra oferuje wyzwania logiczne, które wymuszają na graczach kreatywne myślenie, a także elementy walki, uatrakcyjniające mechanikę rozgrywki. Kluczowym aspektem gry jest balansowanie pomiędzy eksploracją, walką z przeciwnikami oraz rozwiązywaniem różnych zagadek, gdzie każde zadanie będzie prowadzić do odblokowania kolejnych etapów.

1.2 Zakres prac

Zakres prac obejmuje między innymi przygotowanie trzech map o unikalnym wyglądzie,
animacje oraz oświetlenie poszczególnych elementów, udźwiękowienie gry, różne mechaniki gracza i jego przeciwników oraz ukryte przejścia. Gracz ma możliwość zapisu oraz ponownego wczytania osiągniętego etapu gry.

Każdy z trzech poziomów zawiera mechanizmy będące częścią zagadek takie jak: płytki naciskowe, zdefiniowaną liczbę przeciwników, których pokonanie jest kluczem do rozwiązania kolejnych zagadek, sekwencje run oraz skrzynie z przedmiotami, posiadającymi określone właściwości. Gra zawiera także elementy stałe, których status może zostać zapisany w trakcie rozgrywki. Należą do nich: stan aktywacji płytek naciskowych, skrzyń, portali do następnych poziomów, wykonanie poprawnej sekwencji run, położenie gracza na mapie oraz zawartość jego ekwipunku. Ponadto poszczególne poziomy gry zawierają określoną pulę przeciwników, z których każdy posiada sprecyzowaną ilość punktów zdrowia oraz obrażeń jakie może zadać, a także jeden z trzech typów zachowań wobec gracza.

1.3 Grupa docelowa

Gra jest skierowana do graczy, którzy lubią wyzwania związane z rozwiązywaniem łamigłówek i logicznym myśleniem, ale jednocześnie cenią sobie elementy zręcznościowe i chcą spróbować swoich sił w walce z różnymi przeciwnikami. Grupa ta obejmuje zarówno osoby szukające gier wymagających koncentracji i spostrzegawczości, jak i tych, którzy chcą

zanurzyć się w klimatycznym, immersyjnym świecie jaskiń i powoli odkrywać nowe poziomy, poprzez rozwiązywanie zagadek.

1.4 Analiza rynku i konkurencji

Rynek gier przygodowych 2D stale się rozwija, oferując różnorodne produkcje, które kładą nacisk na eksplorację, łamigłówki i narrację. Wiele popularnych gier z tego gatunku, takich jak „Hollow Knight" czy „Celeste", opiera się na precyzyjnej mechanice poruszania się i zręcznościowych wyzwaniach, podczas gdy inne, jak „Limbo" i „Inside", stawiają na bardziej minimalistyczny styl, łącząc atmosferę grozy z rozwiązywaniem łamigłówek.

Proponowana gra wyróżnia się połączeniem elementów logicznych z eksploracją i mrocznym klimatem jaskiń. Na tle konkurencji gra może zaoferować unikalną mieszankę mechanik – zagadki stanowią kluczowy element do postępu w grze, a dodatkowe przeszkody oraz interaktywne elementy otoczenia zwiększają poziom trudności, zmuszając gracza do zręcznościowego podejścia. Wyróżnikiem będzie także osadzenie gry w spójnej, immersyjnej atmosferze, co przyciągnie zarówno fanów gier logicznych, jak i platformowych.

Na rynku gier 2D istnieje zapotrzebowanie na produkcje, które potrafią w innowacyjny sposób łączyć rozwiązywanie zagadek z dynamiczną rozgrywką, a jednocześnie wciągają gracza w niepowtarzalny świat. Gra może stanowić odpowiedź na te potrzeby, oferując unikalne doświadczenia, które będą wyróżniać ją na tle konkurencyjnych tytułów.

2. Analiza biznesowa

W niniejszym rozdziale zostaną przybliżone wymagania funkcjonalne i niefunkcjonalne, które powinna spełniać aplikacja, aby zapewnić użytkownikowi satysfakcjonujące doświadczenie oraz sprawną i stabilną rozgrywkę.

2.1 Wymagania funkcjonalne

2.1.1 Rozgrywka

Gracz musi mieć możliwość sterowania postacią, co pozwala na eksplorację jaskini. Każdy poziom będzie zawierał różnorodne zagadki, których rozwiązanie jest konieczne do przejścia kolejnego poziomu. Dla uatrakcyjnienia rozgrywki w niektórych miejscach na danym poziomie pojawiać się będą przeszkody, które utrudnia graczowi eksplorację i zwiększą wyzwanie.

2.1.2 Interakcja z grą

Gra rozpoczyna się od ekranu startowego, gdzie gracz może rozpocząć nową grę lub kontynuować zapis z poprzedniej rozgrywki. Gra musi oferować opcję ręcznego zapisu stanu rozgrywki. Gracz powinien mieć dostęp do podstawowych informacji, takich jak zdrowie postaci i zawartość jego ekwipunku.

2.1.3 Świat gry

Każdy poziom powinien odróżniać się wizualnie od pozostałych unikalnymi elementami, które urozmaicą rozgrywkę. Gracz powinien mieć możliwość interakcji z obiektami w jaskini (np. niszczenie skał, przesuwanie przedmiotów itp).

2.2 Wymagania niefunkcjonalne

Wymagania niefunkcjonalne, zawarte w niniejszym podrozdziale, określają kryteria jakościowe, które aplikacja powinna spełniać, aby zagwarantować odpowiednie doświadczenia użytkownikowi oraz stabilność jego rozgrywki.

2.2.1 Wydajność

Gra powinna działać płynnie, nawet na urządzeniach o niskich parametrach technicznych oraz być zoptymalizowana pod kątem użycia pamięci i procesora. Rozgrywka powinna przebiegać bez zakłóceń, nawet przy bardziej skomplikowanych sekwencjach akcji i animacji, co jest istotne dla wrażeń płynących z eksploracji i rozwiązywania kolejnych zagadek.

2.2.2 Intuicyjne sterowanie

Sterowanie powinno być proste i intuicyjne, aby umożliwić graczowi szybkie przyswojenie mechanik gry. Kluczowe funkcje, takie jak ruch postaci, interakcja z przedmiotami oraz walka, powinny być dostępne za pomocą prostych i łatwo dostępnych kombinacji klawiszy lub przycisków, a ich układ powinien być logiczny i zgodny z powszechnymi standardami gier.

2.2.3 Estetyka gry

Gra powinna mieć estetykę 2D z klimatem mrocznych jaskiń, ale jednocześnie zachować klimat i czytelność otoczenia. Grafika powinna być nastrojowa, wykorzystująca ciemną kolorystykę i odpowiednie efekty świetlne, takie jak migoczące pochodnie czy odbicia światła.

2.2.4 Jakość dźwięku

Dźwięki w grze powinny być starannie dopasowane do środowiska jaskiń, wzbogacając doświadczenie gracza i potęgując wrażenie przebywania w tajemniczym, podziemnym świecie. Dźwięki interakcji, na przykład otwieranie skrzyń, przesuwanie kamieni czy odgłosy walki, muszą być wyraźne i zgodne z oczekiwaniami gracza, aby podkreślić dynamikę i akcję w grze. Odpowiednio dobrana ścieżka dźwiękowa powinna wprowadzać w nastrój tajemniczości i niepewności, jednocześnie nie dominując nad pozostałymi elementami gry.

3. Opis wykorzystanych technologii

Niniejszy rozdział przedstawia opisy technologii, które umożliwiają stworzenie wydajnej i funkcjonalnej aplikacji. Przybliżone zostały kluczowe cechy poszczególnych narzędzi oraz ich wykorzystanie w procesie tworzenia gry.

3.1 Silnik Godot

Godot to wieloplatformowy silnik do tworzenia gier 2D i 3D, opracowany przez Juana Linietsky'ego i Ariela Manzura[1]. Jego publiczne wydanie miało miejsce w 2014 roku. Silnik wyróżnia się modularnością, prostotą oraz systemem scen i węzłów, który pozwala na budowanie aplikacji w sposób uporządkowany i skalowalny. Każdy element gry (tj. postacie, przedmioty, UI) jest reprezentowany przez węzeł, co ułatwia tworzenie złożonych struktur. Godot jest silnikiem *open-source* wydanym na licencji MIT. Regularnie rozwijany przez społeczność, oferuje wsparcie techniczne oraz dostęp do dokumentacji. Użytkownicy mogą tworzyć gry w wielu językach programowania, takich jak C#, C++,czy GDScript.[2][3]

System scen i węzłów pozwala na modularne budowanie aplikacji. Węzły łączą się w sceny, co umożliwia zarządzanie złożonością aplikacji. Dzięki tej hierarchii, zmiany w kodzie są łatwe do wdrożenia, a testowanie poszczególnych elementów jest proste. Silnik oferuje także fizykę, która umożliwia realistyczne interakcje między obiektami, oraz zaawansowany system animacji 2D, który poprawia dynamikę aplikacji.[2]

Godot ma również zintegrowany system UI, co pozwala na łatwe tworzenie responsywnych interfejsów użytkownika. Dzięki elastyczności UI, aplikacja dostosowuje się do różnych rozdzielczości, a stylizacja elementów zapewnia spójny wygląd.[2]

3.2 GDScript

GDScript to język programowania zaprojektowany specjalnie na potrzeby Godota. Jego składnia jest inspirowana Pythonem, co sprawia, że jest czytelny i łatwy do nauki, nawet dla początkujących programistów. W projekcie rola GDScript opiera się na tworzeniu logiki aplikacji, począwszy od mechanik gry, takich jak stany gracza i przeciwników, interakcje z obiektami oraz zarządzanie interfejsem użytkownika.

Jedną z największych zalet GDScript, jest jego bezpośrednia integracja z Silnikiem Godota, dzięki czemu umożliwia pisanie kodu, który współpracuje z węzłami i scenami. Każdy skrypt napisany w GDScript można przypisać do konkretnego węzła, co pozwala na wygodne rozdzielanie logiki gry na mniejsze, łatwe do zarządzania moduły. Dzięki temu struktura kodu pozostaje przejrzysta, a wszelkie zmiany czy poprawki są łatwe do wdrożenia.

Porównując GDScript do C# lub C++ (które również są wspierane przez Godot), można zauważyć, że GDScript, mimo swojej ograniczonej wydajności, zapewnia lepszą integrację z systemem węzłów oraz łatwiejsze debugowanie.

3.3 SAV

Pliki .sav są specjalistycznymi plikami wykorzystywanymi do przechowywania stanu gry, pozwalając graczowi na zapisanie postępu i powrót do gry w późniejszym czasie. W przeciwieństwie do formatów tekstowych, takich jak JSON, pliki .sav mogą być zapisane w formatach binarnych lub tekstowych, w zależności od gry i jej silnika. [4]

Kluczowe cechy SAV:

• **Złożoność danych**: Pliki .sav przechowują złożone dane o stanie gry, które mogą obejmować różne informacje, takie jak dane o postaci gracza (np. zdrowie, pozycja, przedmioty), informacje o świecie gry (np. aktualna scena, wykonane misje) oraz inne zmienne, które są niezbędne do kontynuowania gry. Dane są zazwyczaj przechowywane w zorganizowanej strukturze (np. w postaci słowników, tablic czy obiektów).

• **Format binarny**: Wiele gier korzysta z formatu binarnego do zapisu plików .sav, ponieważ jest on bardziej efektywny pod względem przechowywania danych. Pliki binarne są szybsze do ładowania i zapisania, a także zajmują mniej miejsca w porównaniu do plików tekstowych. Dodatkowo, pliki binarne są trudniejsze do edytowania przez użytkowników, co pomaga w ochronie przed oszustwami i manipulacjami.

• **Format tekstowy**: Niektóre gry wykorzystują formaty tekstowe, takie jak JSON lub XML, w plikach .sav, aby przechowywać dane o stanie gry. Pliki tekstowe są bardziej czytelne i łatwiejsze do debugowania, Format JSON, jak w przedstawionym przykładzie, jest często używany do zapisywania danych w formacie tekstowym.

• **Struktura danych**: W plikach .sav dane są przechowywane w formie klucz-wartość, podobnie jak w JSON, co ułatwia organizację i dostęp do informacji.

```
var current_save : Dictionary = {
    scene_path = "",
    player = {
        hp = 1,
        max_hp = 1,
        pos_x = 0,
        pos_y = 0
    },
    items = [],
    persistence = [],
}
```

Rys. 3.1 Słownik zawierający scenę, położenie oraz zdrowie gracza, przedmioty oraz elementy trwałe.

Przykładowo, na rysunku 3.1 zaprezentowanym powyżej, dane o grze, takie jak zdrowie gracza, jego pozycja czy przedmioty, są zapisane w formie słownika (Dictionary). Tego rodzaju struktura umożliwia łatwe zarządzanie danymi i ich szybkie wczytywanie.

• **Łatwość w edytowaniu**: W przypadku plików tekstowych .sav, takich jak JSON, dane mogą być łatwo edytowane za pomocą zwykłego edytora tekstu. Dzięki temu, programiści i gracze mogą modyfikować zapisane dane, co jest przydatne podczas debugowania, tworzenia modyfikacji do gier czy dostosowywania zapisów do własnych potrzeb. W plikach binarnych edytowanie danych jest bardziej skomplikowane i wymaga użycia specjalistycznych narzędzi.

• **Zgodność z językami programowania**: Pliki .sav są szeroko wspierane przez różne języki programowania, co pozwala na łatwą implementację mechanizmu zapisu i ładowania stanu gry w różnych środowiskach i platformach.

• **Dostosowanie do różnych typów gier**: Pliki .sav są elastyczne i mogą być wykorzystywane w różnych typach gier – od prostych gier 2D, po zaawansowane gry 3D z dynamicznie zmieniającym się światem. Dzięki złożonej strukturze danych, pliki .sav mogą przechowywać informacje o grze, które są specyficzne dla danej produkcji, takie jak systemy ekwipunku, misji, interakcji z NPC czy stany środowiska.

Pliki .sav to kluczowy element systemu zapisu i ładowania postępów w grach, zapewniający graczom możliwość kontynuowania rozgrywki z tego miejsca, w którym ją przerwali. Dzięki elastyczności formatów, takich jak JSON, oraz efektywności formatu binarnego, pliki .sav są powszechnie stosowane w przemyśle gier komputerowych.

3.4 JSON

JSON (JavaScript Object Notation) jest prostym, czytelnym formatem, który dobrze sprawdza się do przechowywania danych konfiguracyjnych oraz stanów gry, takich jak: położenie gracza, punkty zdrowia czy zawartość ekwipunku. Jest także łatwy do manipulacji zarówno w kodzie, jak i przy ręcznej edycji plików.

Kluczowe cechy JSON:

• **Prostota**: JSON jest prostym formatem tekstowym, który jest łatwy do zrozumienia zarówno dla ludzi, jak i komputerów. Składa się z podstawowych konstrukcji, takich jak pary klucz-wartość i tablice, co sprawia, że jest czytelny i zrozumiały, nawet dla osób, które nie mają doświadczenia w programowaniu.

• **Struktura klucz-wartość**: JSON przechowuje dane w postaci par klucz-wartość, gdzie klucz (string) jest unikalnym identyfikatorem, a wartość może być jednym z kilku typów danych (np. string, number, object, array, boolean, null). Taki format sprawia, że dane są łatwe do przechowywania i manipulacji.

• **Zgodność z językami programowania**: JSON jest szeroko wspierany przez prawie wszystkie języki programowania, takie jak JavaScript, Python, C#, Java, C++, PHP, Ruby i inne. Dzięki temu może być używany do wymiany danych między różnymi platformami i systemami, co czyni go idealnym do komunikacji między aplikacjami.

• **Obsługa różnych typów danych**: W JSON można przechowywać różne typy danych, co pozwala na modelowanie różnych struktur danych. Obsługiwane typy to:

- **String** (ciągi znaków) np. "name": "John"
- **Number** (liczby całkowite lub zmiennoprzecinkowe) np. "age": 30
- **Boolean** (wartości true lub false) np. "isActive": true
- **Object** (obiekty z parami klucz-wartość) np. "address": {"street": "Main St", "city": "Somewhere"}

- **Array** (tablice, listy wartości) np. "animals": ["cat", "dog"]
- **Null** (wartość pustą) np. "middleName": null

• **Hierarchiczna struktura**: JSON umożliwia tworzenie zagnieżdżonych struktur danych. Obiekty mogą zawierać inne obiekty, a tablice mogą zawierać obiekty, liczby, ciągi znaków lub inne tablice. Dzięki temu JSON jest bardzo elastyczny w modelowaniu bardziej złożonych danych.

• **Brak wbudowanej semantyki typów**: JSON nie posiada wbudowanego rozróżnienia na różne typy danych, takie jak liczby całkowite czy zmiennoprzecinkowe (wszystkie liczby są traktowane jako "*Number*"). To może być wadą w sytuacjach, gdzie precyzyjna semantyka danych jest istotna, ale w większości zastosowań JSON wystarcza.

• **Łatwość w edytowaniu**: JSON jest formatem tekstowym, co oznacza, że może być łatwo edytowany przez programistów i użytkowników za pomocą zwykłego edytora tekstu. Użytkownicy mogą łatwo dostosować dane w plikach JSON, co jest przydatne w sytuacjach, gdy dane muszą być ręcznie modyfikowane (np. w przypadku testowania, debugowania lub tworzenia modyfikacji do gier).

• **Dostosowanie do wielu przypadków użycia**: Dzięki swojej prostocie i elastyczności, JSON jest wykorzystywany w szerokim zakresie aplikacji: od prostych aplikacji mobilnych, przez gry komputerowe, aż po zaawansowane systemy przetwarzania danych w chmurze.

3.5 Zasoby Graficzne

W projekcie kluczowym aspektem są elementy graficzne częściowo wykonane ręcznie na potrzeby realizacji pracy oraz pochodzące z zasobów dostępnych na licencjach pozwalających na ich bezpłatne wykorzystanie. Główne źródła zasobów to platformy takie jak: *craftpix.net, itch.io* oraz *opengameart.org*, które oferują szeroki wybór sprite'ów, tła oraz obiektów środowiskowych. Silnik Godota oferuje odpowiednie dopasowanie zasobów graficznych do stylu gry za pomocą narzędzi, takich jak: skalowanie, animacje oraz efekty post-processingu. Dzięki systemowi węzłów, możliwe będzie łatwe zarządzanie grafikami w tym ich integracja z interfejsem użytkownika i animacjami.

3.6 Zasoby Dźwiękowe

Do projektu gry w systemie jaskiń kluczowym elementem do wprowadzenia gracza w klimat świata są dźwięki otoczenia i muzyka. Strony takie jak *freesound.org* i *itch.io* oferują szeroki wybór plików dźwiękowych na licencjach Creative Commons. W ramach projektu dodane zostaną różnorodne efekty dźwiękowe, w tym dźwięki ataku, interakcji oraz muzyki w tle. Godot oferuje rozbudowane narzędzia do zarządzania dźwiękiem za pomocą AudioStreamPlayer i AudioStreamPlayer2D, co pozwoli na precyzyjne ustawienie dźwięków w przestrzeni 2D i synchronizację z wydarzeniami w grze.

4. Implementacja

Rozdział został poświęcony architekturze projektu oraz wybranym rozwiązaniom implementacyjnym poszczególnych elementów gry. Przybliżone zostały mechaniki takie jak: otrzymywanie obrażeń przez gracza, system wykrywania gracza przez przeciwnika, interakcji ze skrzynią, interakcję z bramą do następnego poziomu oraz mechanizm płytki naciskowej.

4.1 Architektura

W ramach implementacji gry w Godot Engine wykorzystano system *Nodów*, czyli węzłów, które są centralnym elementem tej platformy. Dzięki modularnej strukturze węzły umożliwiają intuicyjne i efektywne budowanie logiki gry, a także implementację zaawansowanych mechanik w sposób przejrzysty i łatwy do modyfikacji. Każdy węzeł działa jako niezależny komponent, który można łączyć w hierarchię drzewa dodając kolejne elementy. Przykładowo *Node2D*, który reprezentuje pozycję w przestrzeni dwuwymiarowej, może mieć węzeł podrzędny - *Sprite*, będący obrazkiem bądź siatką zawierającą klatki animacji oraz *CollisionShape2D*, będący kształtem obszaru kolizji dla danego obiektu. Kluczowe elementy gry, takie jak mechanika walki, interakcja z otoczeniem, wykorzystanie przedmiotów czy zagadki, zostaną zaimplementowane w ramach odpowiednio zorganizowanych węzłów oraz powiązanych z nimi skryptów napisanych w GDScript.

Każdy poziom gry został zaprojektowany jako niezależna scena, co pozwala na precyzyjne rozmieszczenie elementów takich jak: przeciwnicy, przeszkody czy zagadki. Edytor Godot umożliwia łatwą konfigurację i testowanie poziomów bez konieczności przeładowywania całej aplikacji, co znacząco ułatwia proces tworzenia gry.

Gra została podzielona na odrębne warstwy funkcjonalności, co pozwala na czytelną organizację projektu:

- **Global**: skrypty odpowiedzialne za wczytywanie poziomów, odtwarzanie muzyki w tle, dodanie/usunięcie gracza z danej sceny, aktualizację punktów zdrowia i położenia.
- **Audio**: zawiera foldery z podziałem na ścieżki dźwiękowe dla poszczególnych elementów np.: przeciwników, gracza czy otoczenia.

- **General Nodes**: zawiera elementy takie jak: *Hitbox* i *HurtBox*, *EnemyCounter* czy *PersistentDataHandler*, które mogą być wykorzystywane dla różnego rodzaju obiektów na mapie.
- **GUI** : zawiera elementy odpowiedzialne za interfejs użytkownika.
- **Interactables** : elementy, z którymi gracz może wejść w interakcję m.in. skrzynie, bramy, portale, płytki naciskowe.
- **Items**: zawiera elementy związane z przedmiotami dostępnymi w grze, takie jak: rodzaje przedmiotów, ich tekstury czy efekty jakie dają po użyciu.
- **Props**: elementy sceny, które gracz może niszczyć, np.: skały, kamienie czy rośliny.
- **Actors**: zawiera osobne sceny dla każdego przeciwnika oraz gracza wraz z siatką animacji.
- **Scripts**: przechowuje skrypty gracza i przeciwników z podziałem na osobne stany.
- **Levels**: elementy związane z poziomami. Zawiera wszystkie dostępne mapy, tekstury elementów użytych do ich stworzenia, scenę przejścia do następnego poziomu wraz ze skryptem ją obsługującym oraz miejsce pojawienia się gracza na nowym poziomie – *Spawnpoint*.

Zastosowanie systemu węzłów w Godot Engine pozwala na modularne podejście do budowy gry, co znacząco ułatwia implementację oraz późniejsze modyfikacje. Każdy element gry działa niezależnie, jednocześnie mogąc się komunikować z innymi za pomocą sygnałów, co zapewnia spójność całego projektu oraz możliwości rozbudowy.

4.2 Otrzymywanie obrażeń przez gracza

Klasa *HurtBox* reprezentuje obszar zadający obrażenia i zawiera zmienną *damage*, która określa ilość zadawanych obrażeń (domyślnie 1). Gdy obiekt *HurtBox* pojawia się w scenie, sygnał *area_entered* jest połączony z funkcją *AreaEntered*. Funkcja *AreaEntered* jest wywoływana za każdym razem, gdy inny obiekt (*Area2D*) wejdzie w obszar *HurtBox*. Jeśli obiekt ten jest typu *HitBox* (czyli obszarem postaci mogącej przyjmować obrażenia), *AreaEntered* wywołuje funkcję *TakeDamage()* obiektu *HitBox*, przekazując do niej siebie jako argument.

```
class_name HurtBox extends Area2D

@export var damage : int = 1

func _ready() -> void:
    area_entered.connect( AreaEntered )
    pass

func AreaEntered( a : Area2D ) -> void:
    if a is HitBox:
        a.TakeDamage( self )
    pass
```

Rys. 4.1 Obszar otrzymywania obrażeń

```
func TakeDamage( hurt_box : HurtBox ) -> void:
    if invulnerable == true:
        return
    UpdateHP( -hurt_box.damage )
    if hp > 0:
        PlayerDamaged.emit( hurt_box )
    else:
        PlayerDamaged.emit( hurt_box )
        UpdateHP( 99 )
    pass

func UpdateHP( delta : int ) -> void:
    hp = clampi( hp + delta, 0, max_hp )
    PlayerHud.update_hp( hp, max_hp )
    pass

func MakeInvulnerable( _duration : float = 1.0 ) -> void:
    invulnerable = true
    hit_box.monitoring = false

    await get_tree().create_timer( _duration ).timeout

    invulnerable = false
    hit_box.monitoring = true
    pass
```

Rys. 4.2 Fragment kodu odpowiedzialny za otrzymywanie obrażeń, aktualizację zdrowia i statusu niewrażliwości

We fragmencie kodu klasy *Player*, widocznym na rysunku 4.2, funkcja *TakeDamage()* obsługuje mechanizm zadawania obrażeń graczowi, sprawdzając najpierw, czy gracz nie jest odporny na obrażenia (*invulnerable*). Jeśli gracz nie ma odporności, funkcja *UpdateHP()* zmniejsza punkty życia o wartość otrzymanych obrażeń (*damage*) przekazaną z klasy *HurtBox* widocznej na rysunku 4.1. Jeśli po otrzymaniu obrażeń wartość *hp* gracza pozostaje większa od zera, emitowany jest sygnał *PlayerDamaged*, który może wyzwalać efekty takie jak np.: animacja otrzymywania obrażeń. Jeśli *hp* wynosi zero lub mniej, *PlayerDamaged* również jest emitowany, a zdrowie gracza zostaje zresetowane do 99, co symuluje odrodzenie.

4.3 Wykrywanie gracza przez przeciwnika

Klasa *VisionArea* odpowiada za wykrywanie gracza przez przeciwnika i dostosowanie kierunku widzenia w zależności od kierunku, w którym zwrócony jest przeciwnik.

```
signal player_entered()
signal player_exited()


func _ready() -> void:
    body_entered.connect( _on_body_enter )
    body_exited.connect( _on_body_exit )

    var p = get_parent()
    if p is Enemy:
        p.direction_changed.connect( _on_direction_changed )
```

Rys. 4.3 Fragment kodu przedstawiający sygnały wejścia oraz wyjścia z obszaru widzenia oraz sygnał zmiany kierunku.

```
func _on_direction_changed( new_direction : Vector2 ) -> void:
    match new_direction:
        Vector2.DOWN:
            rotation_degrees = 0
        Vector2.UP:
            rotation_degrees = 180
        Vector2.LEFT:
            rotation_degrees = 90
        Vector2.RIGHT:
            rotation_degrees = -90
        _:
            rotation_degrees = 0
    pass
```

Rys. 4.4 Funkcja ustawiająca kierunek

```
func _on_body_enter( _b : Node2D ) -> void:
    if _b is Player:
        player_entered.emit()
    pass

func _on_body_exit( _b : Node2D) -> void:
    if _b is Player:
        player_exited.emit()
    pass
```

Rys. 4.5 Funkcje sprawdzające czy obiekt jest graczem i emitujące sygnały wyjścia i wejścia w obszar interakcji

Na początku *VisionArea* definiuje dwa sygnały: *player_entered*, który emituje informację, gdy gracz wchodzi w obszar widzenia, oraz *player_exited*, który sygnalizuje opuszczenie tego obszaru przez gracza. W metodzie *_ready()*, widocznej na rysunku 4.3, sygnały *body_entered* i *body_exited* (emitowane przy wejściu lub wyjściu dowolnego obiektu z obszaru *VisionArea*) są połączone odpowiednio z funkcjami *_on_body_enter* oraz *_on_body_exit*.

Na rysunku 4.3 widać, że jeśli rodzicem *VisionArea* jest obiekt typu *Enemy*, klasa ta nasłuchuje także sygnału *direction_changed* emitowanego przez przeciwnika. Wykrycie zmiany kierunku przeciwnika uruchamia funkcję *on_direction_changed*, widoczną na rysunku 4.4, która dostosowuje rotację *VisionArea*, tak aby była skierowana w nowym kierunku, w jakim przeciwnik patrzy.

Funkcja *_on_body_enter*, widoczna na rysunku 4.5, sprawdza, czy obiekt wchodzący do obszaru widzenia jest typu *Player*; jeśli tak, emituje sygnał *player_entered*, co sygnalizuje obecność gracza w zasięgu widzenia przeciwnika. Analogicznie, gdy gracz opuszcza obszar, *on_body_exit* emituje sygnał player_exited, co oznacza, że przeciwnik stracił gracza z pola widzenia.

Przez te mechanizmy, *VisionArea* umożliwia przeciwnikowi dynamiczne wykrywanie obecności gracza i odpowiednie reagowanie na jego obecność w polu widzenia oraz śledzi zmiany kierunku patrzenia przeciwnika, co pozwala na dokładniejsze śledzenie gracza.

4.4 Interakcja ze skrzynią

Klasa *TreasureChest* reprezentuje skrzynię, z którą gracz może wchodzić w interakcję, aby otrzymać przedmiot. Klasa zawiera zmienne *item_data*, przechowującą dane o przedmiocie znajdującym się w skrzyni oraz *quantity* przechowującą jego ilość.

```
func _ready() -> void:
    _update_texture()
    _update_label()
    if Engine.is_editor_hint():
        return
    interact_area.area_entered.connect( _on_area_enter )
    interact_area.area_exited.connect( _on_area_exit )
    persistent_data_is_open.data_loaded.connect( set_chest_state )
    set_chest_state()
    pass
```

Rys. 4.6 Ustawienie tekstury, etykiety i statusu otwarcia skrzyni

```
func set_chest_state() -> void:
    is_open = persistent_data_is_open.value
    if is_open:
        animation_player.play("opened")
    else:
        animation_player.play("closed")
```

Rys. 4.7 Funkcja ustawiająca animację skrzyni.

W fragmencie kodu, widocznym na rysunku 4.6, funkcja najpierw aktualizuje teksturę i etykietę skrzyni na podstawie przypisanego do niej przedmiotu i jego ilości. Następnie sprawdzany jest sygnał, czy skrzynia działa w grze, co umożliwia reagowanie na wejście gracza w obszar interakcji. Na końcu funkcja łączy sygnał *data_loaded* z funkcją *set_chest_state*, widoczną na rysunku 4.7, która ustawia teksturę skrzyni w zależności czy została już otworzona przez gracza.

Kiedy gracz wchodzi w interakcję ze skrzynią, wywoływana jest funkcja *player_interact()*. Jeśli skrzynia jest zamknięta, *is_open* jest ustawione na *true*,

a *persistent_data_is_open.set_value()* zapisuje jej stan. Odtwarzana jest animacja otwierania skrzyni, a gracz otrzymuje przedmioty.

```
func _on_area_enter( a : Area2D ) -> void:
    PlayerManager.interact_pressed.connect( player_interact )
    pass

func _on_area_exit( a : Area2D ) -> void:
    PlayerManager.interact_pressed.disconnect( player_interact )
    pass
```

Rys. 4.8 Funkcje emitujące sygnały o interakcji.

Funkcja *on_area_enter()*, zaprezentowana na rysunku 4.8, jest wywoływana, gdy gracz wejdzie w obszar interakcji ze skrzynią. Sygnał *interact_pressed* z *PlayerManager* jest połączony z *player_interact*, co umożliwia graczowi otwarcie skrzyni. Gdy gracz opuści obszar interakcji (funkcja *_on_area_exit()*), połączenie z *player_interact* jest usuwane.

Dzięki tej logice, klasa *TreasureChest* pozwala graczowi na otwieranie skrzyni, odbieranie przedmiotów oraz zapamiętywanie stanu otwarcia, co umożliwia zachowanie danych w przypadku zapisania gry i ponownego wczytania poziomu.

4.5 Mechanizm otwierania bramy do następnego poziomu.

Klasa *NextLevelGate* implementuje mechanikę bramy, która otwiera przejście na kolejny poziom, jeśli gracz posiada odpowiedni klucz (*key_item*) w ekwipunku. Brama może być otwierana za pomocą interakcji, a jej stan (*is_open*) jest zapisywany w obiekcie *PersistentDataHandler*, dzięki czemu brama pozostaje otwarta, przy wczytaniu zapisanego postępu w grze.

```
func _ready() -> void:
    interact_area.area_entered.connect( _on_area_enter )
    interact_area.area_exited.connect( _on_area_exit )
    is_open_data.data_loaded.connect( set_state )
    set_state()
```

Rys 4.9 Funkcje łączące z sygnałem wyjścia i wejścia w obszar interakcji

```
func open_gate() -> void:
    if key_item == null:
        return
    var gate_unlocked = PlayerManager.INVENTORY_DATA.use_item( key_item )

    if gate_unlocked:
        animation_player.play( "open_gate" )
        audio.stream = open_audio
        is_open_data.set_value()
    else:
        audio.stream = locked_audio

    audio.play()
    pass
```

Rys 4.10 Funkcja odpowiadająca za otwarcie bramy za pomocą klucza

```
func close_gate() -> void:
    animation_player.play( "close_gate" )

func set_state() -> void:
    is_open = is_open_data.value
    if is_open:
        animation_player.play( "opened" )
    else:
        animation_player.play( "closed" )
```

Rys. 4.11 Funkcja zamykająca bramę oraz funkcja ustawiająca stan otwarcia

Funkcja *_ready()*, widoczna na rysunku 4.9, podłącza sygnały *area_entered* i *area_exited* obszaru interakcji (*interact_area*) odpowiednio do funkcji *on_area_enter* oraz *on_area_exit*, aby reagować na obecność gracza w pobliżu bramy. Łączy również sygnał *data_loaded* obiektu *is_open_data* z funkcją *set_state*, która ustawia stan bramy (otwarta/zamknięta) w oparciu o zapisane dane.

Funkcja *open_gate()*, widoczna na rysunku 4.10, jest wywoływana, gdy gracz próbuje otworzyć bramę klikając przycisk interakcji:

- Jeśli brama wymaga klucza (*key_item*) i gracz użyje go z powodzeniem (sprawdza to *PlayerManager.INVENTORY_DATA.use_item(key_item)*), brama zostaje otwarta. Odtwarzana jest animacja otwierania (*open_gate*), ustawiany jest dźwięk *open_audio*, a stan otwarcia jest zapisywany w *PersistentDataHandler* za pomocą *is_open_data.set_value()*.
- Jeśli gracz nie ma odpowiedniego klucza, odtwarzany jest dźwięk blokady (*locked_audio*).

Funkcja *close_door()*, widoczna na rysunku 4.11, służy do zamykania bramy. Odtwarza animację zamykania (*close_gate*). Nie modyfikuje stanu zapisanych danych, dlatego brama pozostaje logicznie zamknięta, dopóki gracz jej ponownie nie otworzy.

Funkcja *set_state()* ustawia stan otwarcia bramy, na podstawie wartości zapisanej w *is_open_data*. Jeśli *is_open* jest *true*, odtwarzana jest animacja *opened*; w przeciwnym wypadku, animacja *closed*.

```
func _on_area_enter( _a : Area2D ) -> void:
    PlayerManager.interact_pressed.connect( open_gate )


func _on_area_exit( _a : Area2D ) -> void:
    PlayerManager.interact_pressed.disconnect( open_gate )
```

Rys. 4.12 Sygnały wyjścia i wejścia w obszar interakcji

Funkcja *_on_area_enter()*, widoczna na rysunku 4.12, łączy sygnał *interact_pressed* z funkcją *open_gate()*, co umożliwia graczowi otwarcie bramy, jeśli ten znajduje się w obszarze interakcji. Funkcja *_on_area_exit()* odłącza ten sygnał, gdy gracz opuści obszar.

4.6 Mechanizm płytki naciskowej

Klasa *PressurePlate* implementuje mechanikę naciskowej płyty, która wykrywa obecność obiektów (np. gracza lub innych ciał fizycznych) w jej obszarze. Płyta emituje sygnały *activated* i *deactivated*, informujące o aktywacji lub dezaktywacji.

```
func _ready() -> void:
    area_2d.body_entered.connect( _on_body_entered )
    area_2d.body_exited.connect( _on_body_exited )
    off_rect = sprite.region_rect
```

Rys 4.13 Inicjalizacja sygnałów

Rysunek 4.13 przedstawia funckję *_ready()*, która podłącza sygnały *body_entered* i *body_exited* obszaru detekcji (*area_2d*) odpowiednio do *on_body_entered* i *on_body_exited*. Wartość *off_rect*, która reprezentuje domyślną pozycję regionu tekstury na *spricie*, jest zapisana na podstawie aktualnej wartości *sprite.region_rect*.

```
func _on_body_entered( b : Node2D ) -> void:
    bodies += 1
    check_is_activated()
    pass

func _on_body_exited( b : Node2D ) -> void:
    bodies -= 1
    check_is_activated()
    pass
```

Rys. 4.14 Funkcje odpowiedzialne za rejestrowanie wejścia i wyjścia z obszaru interakcji

Funkcja *on_body_entered()*, widoczna na rysunku 4.14, jest wywoływana, gdy obiekt (b) wejdzie w obszar działania płyty naciskowej:

- Zmienna *bodies* (licznik obiektów na płycie) zostaje zwiększona o 1.

- Wywoływana jest funkcja *check_is_activated()*, która sprawdza, czy płyta powinna zostać aktywowana.

Funkcja *on_body_exited()* jest wywoływana, gdy obiekt opuści obszar płyty:

- Zmienna *bodies* zostaje zmniejszona o 1.
- Ponownie wywoływana jest funkcja *check_is_activated()*, aby sprawdzić, czy płyta powinna zostać dezaktywowana.

```
func check_is_activated() -> void:
    if bodies > 0 and is_active == false:
        is_active = true
        sprite.region_rect.position.x = off_rect.position.x + 32
        play_audio( audio_activate )
        activated.emit()
    elif  bodies <= 0 and is_active == true:
        is_active = false
        sprite.region_rect.position.x = off_rect.position.x
        play_audio( audio_deactivate )
        deactivated.emit()
```

Rys. 4.15 Sprawdzenie stanu aktywacji płytki naciskowej

Rysunek 4.15 przedstawia funkcję *check_is_activated()*, która kontroluje stan płyty na podstawie liczby obiektów (*bodies*) i jej aktualnego stanu (*is_active*):

- **Aktywacja płyty**: Jeśli liczba obiektów (*bodies*) jest większa od 0 i płyta nie jest aktywna (*is_active == false*):
    - Flaga *is_active* jest ustawiana na *true*.
    - Pozycja regionu tekstury na *spricie* jest przesuwana w poziomie o 32 jednostki (symuluje wizualną zmianę stanu płyty).
    - Wywoływana jest funkcja *play_audio()* z dźwiękiem aktywacji (*audio_activate*).
    - Emitowany jest sygnał *activated*.

- **Dezaktywacja płyty**: Jeśli liczba obiektów (*bodies*) wynosi 0 i płyta jest aktywna (*is_active == true*):
    - Flaga *is_active* jest ustawiana na *false*.
    - Pozycja regionu tekstury wraca do domyślnej (off_rect).
    - Wywoływana jest funkcja *play_audio()* z dźwiękiem dezaktywacji (*audio_deactivate*).
    - Emitowany jest sygnał *deactivated*.

```
func play_audio( _stream : AudioStream ) -> void:
    audio.stream = _stream
    audio.play()
```

Rys 4.16 Odtwarzanie dźwięku

Funkcja *play_audio()* widoczna na rysunku 4.16, przypisuje strumień dźwiękowy do *audio.stream*, a następnie uruchamia odtwarzanie dźwięku za pomocą *audio.play()*.

Klasa *PressurePlate* realizuje logikę płyty naciskowej, która reaguje na obecność obiektów w swoim obszarze działania. Zlicza obiekty za pomocą zmiennej *bodies* i na tej podstawie aktywuje lub dezaktywuje się, modyfikując swój wizualny stan (*sprite.region_rect*), odtwarzając odpowiednie dźwięki (*audio_activate*, *audio_deactivate*) oraz emitując sygnały *activated* i *deactivated*. Dzięki tym mechanizmom, płytka naciskowa może być łatwo wykorzystana jako element interaktywny do uruchamiania innych mechanizmów.

5. Interfejs

Gra tworzona na silniku Godot Engine, oferuje graczom przejrzysty i intuicyjny interfejs wspierający eksplorację świata, zarządzanie zasobami oraz rozgrywkę. Interfejs został zaprojektowany w sposób minimalistyczny, aby nie odciągać uwagi od kluczowych aspektów gry. W poniższych sekcjach omówiono każdy aspekt widocznych interfejsów.

5.1 Główne menu

Ekran początkowy jest jednym z głównym interfejsów w grach. Umożliwia rozpoczęcie rozgrywki od nowa lub wczytanie zapisanego stanu gry.

Rys 5.1 Główne menu

Na rysunku 6.1 przedstawiono widok głównego menu gry z logiem oraz dwoma przyciskami. Pierwszy przycisk odpowiada za uruchomienie nowej rozgrywki, drugi pozwala na kontynuowanie postępu z wcześniej zapisanego stanu gry. Przy pierwszym uruchomieniu na nowym urządzeniu, gdy żaden wcześniej zapisany stan gry nie jest dostępny, przycisk "*continue*" nie pojawi się na ekranie.

5.2 Pasek zdrowia

Pasek zdrowia wyświetlany podczas eksploracji poziomu odpowiada za dostarczanie graczowi kluczowych informacji o punktach życia postaci.

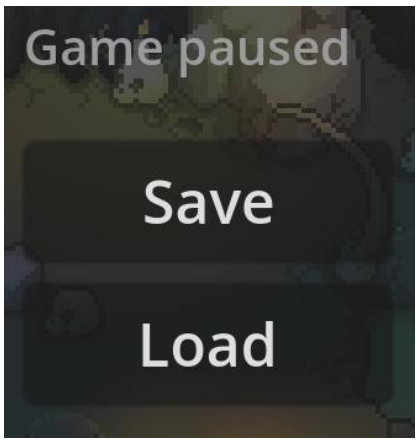
Rys. 5.2 Pasek pełnego zdrowia gracza

Rys. 5.3 Pasek zdrowia gracza po otrzymaniu obrażeń

Na rysunku 5.1 informacja o obecnym stanie zdrowia gracza widoczna jest w prawym górnym rogu. Każda pojedyncza ikona serca reprezentuje 2 punkty zdrowia. W momencie otrzymania obrażeń, widocznym na rysunku 5.2, ikona aktualizuje się tym samym odzwierciedlając nowy poziom zdrowia.

5.3 Menu pauzy

Menu pauzy jest istotnym elementem interfejsu, umożliwiającym graczowi zatrzymanie rozgrywki i dostęp do opcji takich jak zapis gry oraz wczytanie zapisanego stanu.



Rys 5.4 Menu pauzy

Menu pauzy jest wyświetlane jako centralny ekran dialogowy, który przykrywa aktualną scenę gry. Składa się z dwóch przycisków *Save* (zapis gry) oraz *Load* (wczytanie gry).

Zapis gry: Po wybraniu opcji "*Save*", obecny stan gry jest zapisywany do pliku w systemie. Stan ten obejmuje:

- Dokładną lokalizację gracza na mapie.
- Bieżący stan zdrowia gracza.
- Wszystkie przedmioty w ekwipunku.
- Status otwarcia i aktywowania poszczególnych mechanizmów i skrzyń na mapie.

Wczytanie gry: Opcja "*Load*" umożliwia przywrócenie wcześniej zapisanego postępu w rozgrywce.

5.4 Ekwipunek

W trakcie gry gracz zbiera określone przedmioty, które przechowywane są w jego ekwipunku. Przedmioty te mogą zostać użyte do regeneracji zdrowia lub otwarcia bram i portali prowadzących do dalszych obszarów jaskini.

Rys 5.5 Ekwipunek wraz z przedmiotami i ich opisami

Ekran ekwipunku, widoczny na rysunku 5.4, jest podzbiorem menu pauzy, wyświetlanym z prawej strony ekranu. Jest on podzielony na 12 miejsc (*slotów*). Każdy *slot* może zawierać dany rodzaj przedmiotu wraz z liczbą reprezentującą ich ilość lub być pusty. Obecnie aktywny *slot* zaznaczony jest ramką, co pozwala na precyzyjne poruszanie się po ekwipunku. Po zebraniu przedmiotu jest on automatycznie przypisywany do najbliższego wolnego *slotu*. W przypadku, gdy ekwipunek jest pełny wyświetli się komunikat informujący o braku wolnego miejsca. W momencie, gdy gracz zaznaczy ramką dany przedmiot, pod ekwipunkiem wyświetli się opis, zawierający informacje na temat przedmiotu lub sposobu jego użycia.

5.5 Ekran śmierci

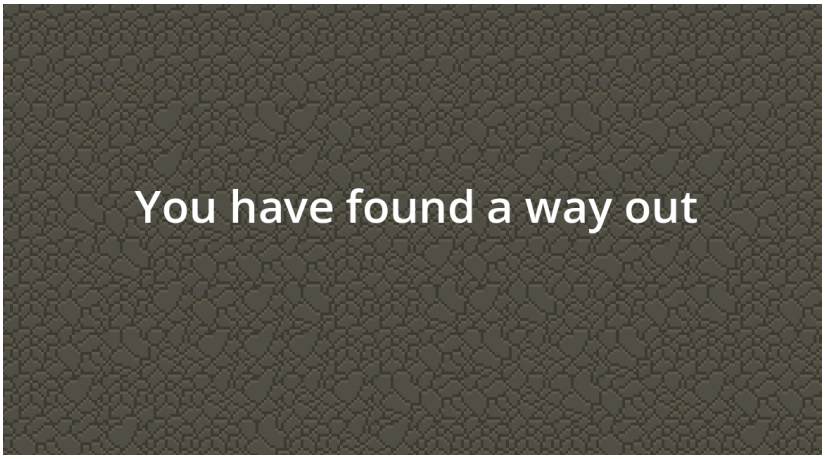
Ekran wyświetlany po śmierci gracza informuje o zakończeniu rozgrywki i oferuje możliwość wznowienia gry od momentu ostatniego jej zapisu lub powrót do głównego menu.

Rys 5.6 Ekran śmierci

Na rysunku 5.6 widoczny jest ekran wyświetlany po śmierci gracza. Kliknięcie przycisku "*continue*", pozwala na kontynuowanie gry od momentu ostatniego zapisu, natomiast przycisk "*main menu*", przenosi gracza do ekranu początkowego, gdzie może on zdecydować o rozpoczęciu gry od nowa lub kontynuacji rozgrywki z wcześniejszego zapisu.

5.6 Ekran ukończenia gry

Ekran ukończenia gry wyświetlany jest w chwili, gdy gracz odnajdzie wyjście z systemu jaskiń. Zawiera on krótką animację, a po jej wykonaniu wyświetla się logo oraz przycisk pozwalający na powrót do głównego menu.



Rys 5.7 Fragment animacji po ukończeniu gry

Rys 5.8 Ekran ukończenia gry

Rysunek 5.7 przedstawia część animacji, jaka wyświetlana jest po odnalezieniu wyjścia. Zawiera ona informację o ukończeniu gry, pełne logo oraz informację o autorze. Na rysunku 5.8 widoczny jest ekran końcowy z przyciskiem umożliwiającym powrót do menu głównego, gdzie gracz może rozpocząć grę od początku lub wczytać zapisany stan rozgrywki.

6. Elementy interaktywne

Na mapie znajdują się elementy, z którymi gracz może wchodzić w interakcję. Każdy z elementów posiada własną, unikalną mechanikę oraz animacje i dźwięk.

6.1 Przedmioty

Gra oferuje możliwość zbierania różnych przedmiotów wraz z postępem w rozgrywce. Każdy przedmiot posiada unikalne właściwości takie jak np.: możliwość uleczenia gracza o określoną ilość punktów życia lub otwarcie przejścia prowadzącego do dalszej części mapy.

Rys. 6.1 Klucz, który gracz może podnieść

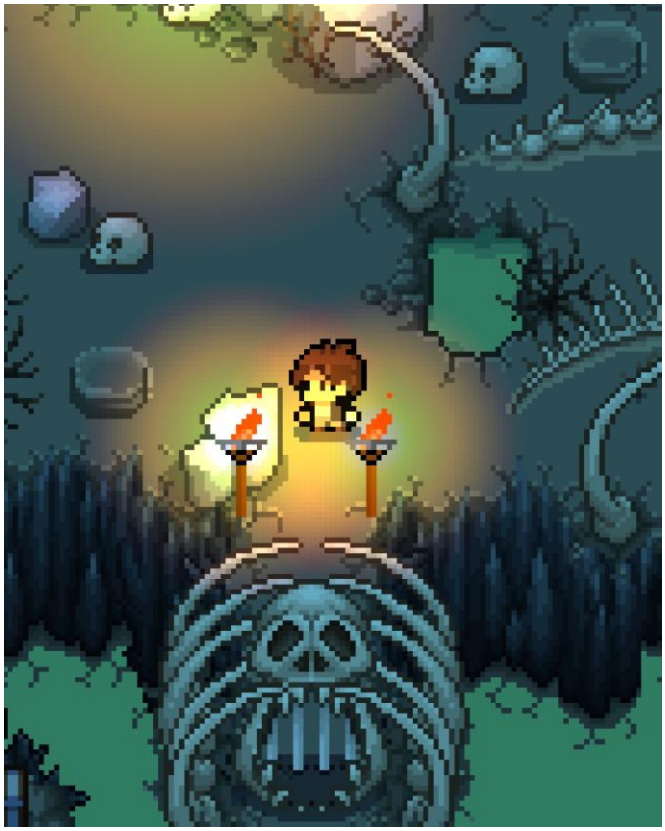
Rysunek 6.1 przedstawia gracza stojącego obok klucza. Przedmiot ten może zostać podniesiony i tym samym zostanie on dodany do jego ekwipunku. Gracz może go użyć do otwarcia odpowiedniego obiektu na mapie. Przedmioty mogą zostać pozyskane z różnych źródeł. W chwili, gdy gracz wykona określoną czynność np. pokona przeciwnika, rozwiąże zagadkę lub otworzy skrzynię, będzie możliwy do zdobycia przedmiot w postaci jednego z trzech rodzai kluczy lub dwóch rodzai mikstur zdrowia.

6.2 Mechanizm płytki naciskowej

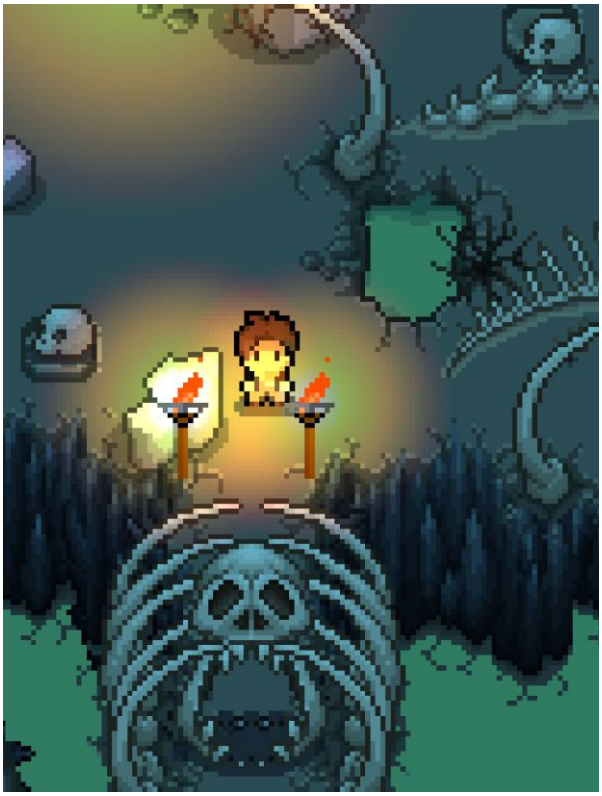
W grze występują zagadki związane z jednoczesną aktywacją określonej ilości płytek naciskowych w celu otwarcia bramy, umożliwiając tym dalszą eksplorację jaskini. Elementami, które umożliwiają jednoczesne wciśnięcie wszystkich płytek są czaszki, które gracz może dowolnie przesuwać. Wraz z postępem rozgrywki ilość płytek naciskowych zmienia się. Niektóre płytki i czaszki mogą być ukryte za elementami dekoracyjnymi.

Rys 6.2 Czaszka

Na rysunku 6.2 widoczna jest czaszka, którą gracz może przesuwać w dowolnym kierunku, wykorzystując ją do aktywowania płytki naciskowej i otwarcia bramy.



Rys. 6.3 Płytki naciskowe wraz z bramą.



Rys. 6.4 Aktywowanie mechanizmu otwierającego bramę.

Na rysunkach 6.3 oraz 6.4 widoczne jest wykonanie mechanizmu otwierającego bramę. Po przesunięciu obu czaszek na płytki naciskowe brama otwiera się, odblokowując tym samym przejście do dalszej części danego poziomu jaskini. W przypadku, gdy gracz omyłkowo przesunie czaszkę w miejsce, w którym się ona zablokuje, uniemożliwiając jej przesuwanie, po wczytaniu gry czaszka wróci do położenia początkowego. Status jej przesunięcia na płytkę naciskową wraz ze statusem otwarcia bramy może zostać zapisany przez gracza, tym samym po wczytaniu gry czaszka automatycznie znajdzie się na płytce, a brama będzie otwarta.

6.3 Skrzynia

W grze występują trzy typy skrzyń, z których każda ma inny kolor. Skrzynie w kolorze brązowym zawierają małe mikstury zdrowia. Są one najbardziej powszechnym rodzajem skrzyń na mapie. Kolejnym typem są skrzynie w kolorze złotym. Mogą one zawierać zarówno małe i duże mikstury zdrowia, a także jeden z trzech rodzajów kluczy - brązowy. Ostatni typ skrzyń ma kolor fioletowy i jest najrzadziej występującym w grze. Może zawierać jedynie złoty lub srebrny klucz, potrzebny do aktywacji portalu, umożliwiającego przejście na następny poziom jaskini.

Rys 6.5 Zamknięta skrzynia **Rys 6.6** Interakcja gracza ze skrzynią **Rys 6.7** Otwarta skrzynia

Na rysunkach 6.5, 6.6 oraz 6.7 przedstawiono kolejno przebieg interakcji gracza ze skrzynią. Po wejściu w obszar interakcji gracz może nacisnąć przycisk (domyślnie F), który powoduje jej otwarcie, tym samym dodając przedmiot jaki się w niej znajdował do ekwipunku gracza. Po zapisaniu i ponownym wczytaniu stanu gry, skrzynia pozostaje otwarta, a gracz traci możliwość interakcji.

6.4 Dekoracje

Gra oferuje możliwość niszczenia niektórych elementów dekoracyjnych umieszczonych na mapie. Należą do nich między innymi rośliny, stosy kości, małe skały i kamienie. Każdy z elementów posiada określoną ilość uderzeń, którą gracz musi wykonać, aby go zniszczyć. Za niektórymi z nich mogą znajdować się skrzynie, elementy związane z zagadkami lub cenne przedmioty.

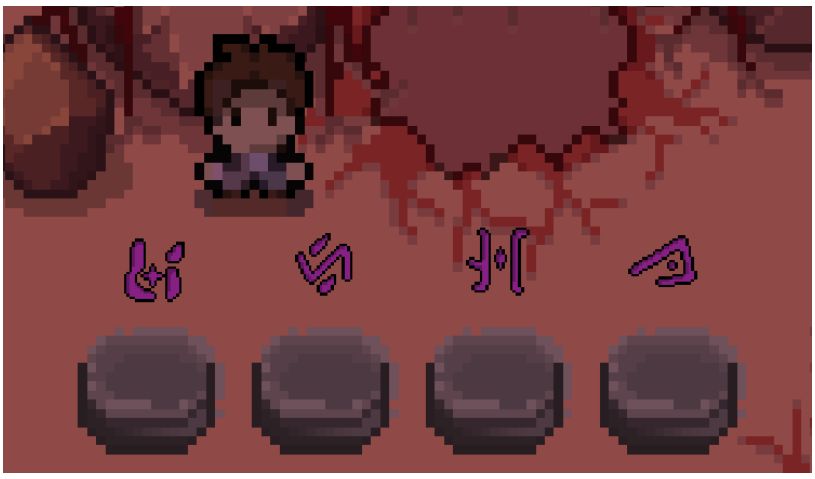
Rys 6.8 Kamień blokujący dostęp do zagadki

Rys 6.9 Zniszczenie kamienia atakiem

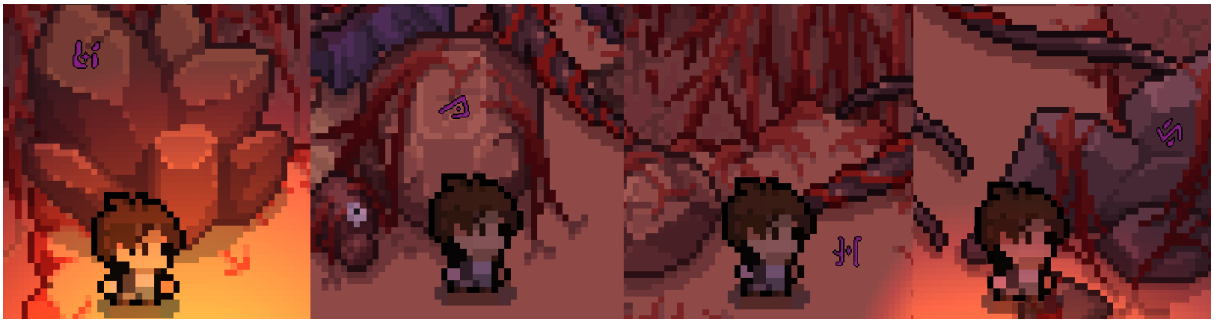
Rysunki 6.8 oraz 6.9 przedstawiają jeden z dostępnych elementów dekoracyjnych, które mogą zostać zniszczone. Elementy posiadają kształt kolizji utrudniając graczowi przejście przez ten obiekt. Użycie ataku powoduje redukcję ilości uderzeń jakie element może przyjąć. W momencie osiągnięcia maksymalnej ilości obrażeń, dekoracja zostaje zniszczona, a obszar kolizji przestaje blokować przejście.

6.5 Sekwencja runiczna

Podczas rozgrywki gracz spotyka na swojej drodze zagadki, które musi rozwiązać, aby móc dostać się do dalszych rejonów mapy. Wraz z postępem, poszczególne elementy związane z zagadkami stają się trudniejsze, aby urozmaicić rozgrywkę i zapewnić graczowi coraz większe wyzwanie.



Rys 6.10 Zagadka z runami



Rys 6.11 Sekwencja run zgodnie z kolejnością eksploracji

Na rysunku 6.10 widoczne są przyciski z odpowiadającymi im runami na ziemi. Podczas poruszania się po jaskini, gracz może zaobserwować runy na kamieniach, ziemi oraz innych elementach otoczenia, co przedstawia rysunek 6.11. Kluczem do rozwiązania zagadki i otworzenia przejścia jest naciśnięcie przycisków we właściwej kolejności, zgodnej z kierunkiem poruszania się po jaskini. Po otwarciu bramy, gracz może zapisać jej status. Po ponownym wczytaniu gry stan jej aktywacji pozostanie taki, jaki był w chwili zapisu rozgrywki.

6.6 Portal do następnego poziomu

Celem przejścia na kolejny poziom jaskini, jest znalezienie portalu oraz pasującego do niego klucza. Każda z bram posiada własny, unikalny rodzaj klucza, który służy do jej otwarcia.

Rys 6.12 Nieaktywny portal

Rys **6.13** Aktywowany portal

Rys 6.14 Klucz do bramy

Rys 6.15 Klucz wykorzystany do otwarcia

Na rysunkach 6.12 i 6.14 znajdują się kolejno portal do następnego poziomu oraz ekwipunek zawierający klucz, który go aktywuje. Po wejściu w obszar interakcji i naciśnięciu przycisku (domyślnie F), portal zostaje aktywowany, a klucz znika z ekwipunku gracza, co zostało pokazane na rysunkach 6.13 oraz 6.15. Stan aktywowania danego portalu może zostać zapisany przez gracza.

7. Testowanie

Poniższy rozdział poświęcony został testom, które mają na celu weryfikację stanu gry, interakcji gracza z otoczeniem oraz poprawności animacji. Testy obejmują między innymi sprawdzenie przejść między stanami gracza, testowanie mechanik gry takich jak: zagadki, interakcje z obiektami na mapie oraz przejścia między animacjami. Celem poniższych testów jest sprawdzenie, czy wszystkie aspekty rozgrywki działają zgodnie z zamierzeniami.

7.1 Scenariusze testowe

Testy manualne, to proces weryfikacji oprogramowania wykonywany przez testera w sposób ręczny, bez użycia narzędzi automatyzujących. Tester odgrywa rolę użytkownika końcowego, przechodząc przez interfejs aplikacji i wykonując zaplanowane przypadki testowe w celu zidentyfikowania błędów. Przypadki testowe (ang. *Test Case*) to szczegółowy opis testu, który określa dokładne kroki, dane wejściowe, oczekiwane wyniki i warunki wstępne, wymagane do weryfikacji określonej funkcjonalności aplikacji. Poniższy rozdział zawiera wybrane przypadki scenariuszy testowych.

Tabela 7.1 Przypadek testowy animacji bezczynności

ID	**TC001**
Tytuł	Stan bezczynności.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Sprawdź animacje gracza dla każdego z kierunków obracając postać przyciskami WSAD.
Oczekiwany rezultat	Animacja wykonuje się dla każdego z kierunków.

Rys 7.1 Animacja bezczynności dla kierunków prawo oraz dół

Tabela 7.2 Przypadek testowy poruszania postacią

ID	**TC002**
Tytuł	Poruszanie postacią.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Wciśnij jeden z klawiszy WASD, aby poruszyć postacią. 2. Wykonaj krok 1 dla każdego kierunku.
Oczekiwany rezultat	Postać porusza się w wyznaczonym kierunku wykonując animację chodzenia.

Rys 7.2 Animacja poruszania się dla kierunku w prawo

Tabela 7.3 Przypadek testowy przejścia między stanami

ID	**TC003**
Tytuł	Przejście między stanami.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Porusz postacią wciskając jeden z klawiszy WASD. 2. Zatrzymaj się zwalniając klawisz. 3. Wykonaj atak klikając lewy przycisk myszy.
Oczekiwany rezultat	Postać płynnie przechodzi między stanami, wykonując animacje zgodnie z otrzymanym wejściem – *Input.*

Rys 7.3 Płynne przejście między animacjami bezczynności, chodzenia i ataku

Tabela 7.4 Przypadek testowy ataku gracza

ID	**TC004**
Tytuł	Atak gracza.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź przeciwnika. 2. Użyj lewej przycisku myszy, będąc blisko przeciwnika.
Oczekiwany rezultat	Wykonuje się animacja ataku gracza, przeciwnik podświetla się na czerwono oraz wykonuje się animacja ogłuszenia.

Rys 7.4 Animacja ataku gracza i ogłuszenia przeciwnika

Tabela 7.5 Przypadek testowy otrzymywania obrażeń

ID	**TC005**
Tytuł	Otrzymywanie obrażeń przez gracza.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź przeciwnika. 2. Podejdź jak najbliżej przeciwnika.
Oczekiwany rezultat	Gracz zostaje zaatakowany, punkty zdrowia zostają zredukowane o ilość otrzymanych obrażeń, wykonuje się animacja ogłuszenia gracza.

Rys 7.5 Animacja otrzymania obrażeń oraz redukcja punktów zdrowia

Tabela 7.6 Przypadek testowy śmierci gracza

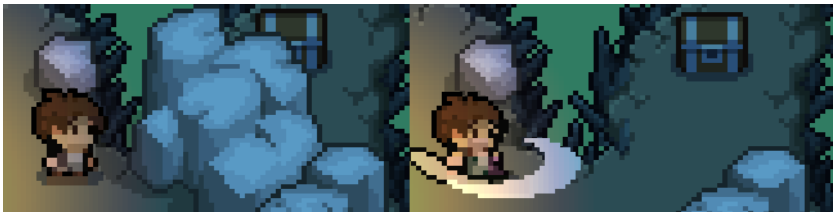
ID	**TC006**
Tytuł	Śmierć gracza.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź przeciwnika. 2. Podejdź jak najbliżej przeciwnika. 3. Zaczekaj, aż punkty zdrowia zostaną zredukowane do 0.
Oczekiwany rezultat	Po zniknięciu ostatniego punktu zdrowia wyświetla się animacja śmierci gracza, po czym widoczny jest ekran śmierci z możliwością wczytania zapisanego stanu gry, lub powrotu do głównego menu.

Rys 7.6 Animacja śmierci gracza

Rys 7.7 Ekran przegranej po śmierci gracza

Tabela 7.7 Przypadek testowy interakcji z otoczeniem

ID	**TC007**
Tytuł	Interakcja z otoczeniem.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Podejdź do obiektu interaktywnego. 2. Spróbuj przejść przez obiekt. 3. Wykonaj atak.
Oczekiwany rezultat	Obiekt blokuje przejście do momentu wykonania ataku i zniszczenia go.



Rys 7.8 Zniszczenie skały blokującej przejście atakiem

Tabela 7.8 Przypadek testowy otwierania skrzyni

ID	**TC008**
Tytuł	Otwarcie skrzyni.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź skrzynię. 2. Podejdź do skrzyni. 3. Naciśnij klawisz F, aby wejść w interakcję.
Oczekiwany rezultat	Skrzynia wykonuje animację otwarcia ukazując przedmiot, jaki się w niej znajdował.

Rys 7.9 Otwarcie skrzyni

Tabela 7.9 Przypadek testowy dodania przedmiotu do ekwipunku

ID	**TC009**
Tytuł	Dodanie przedmiotu do ekwipunku.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź przedmiot na mapie. 2. Podejdź jak najbliżej przedmiotu. 3. Naciśnij klawisz Escape, aby otworzyć ekwipunek.
Oczekiwany rezultat	Przedmiot znika z mapy i pojawia się w ekwipunku gracza.

Rys 7.10 Dodanie podniesionego klucza do ekwipunku

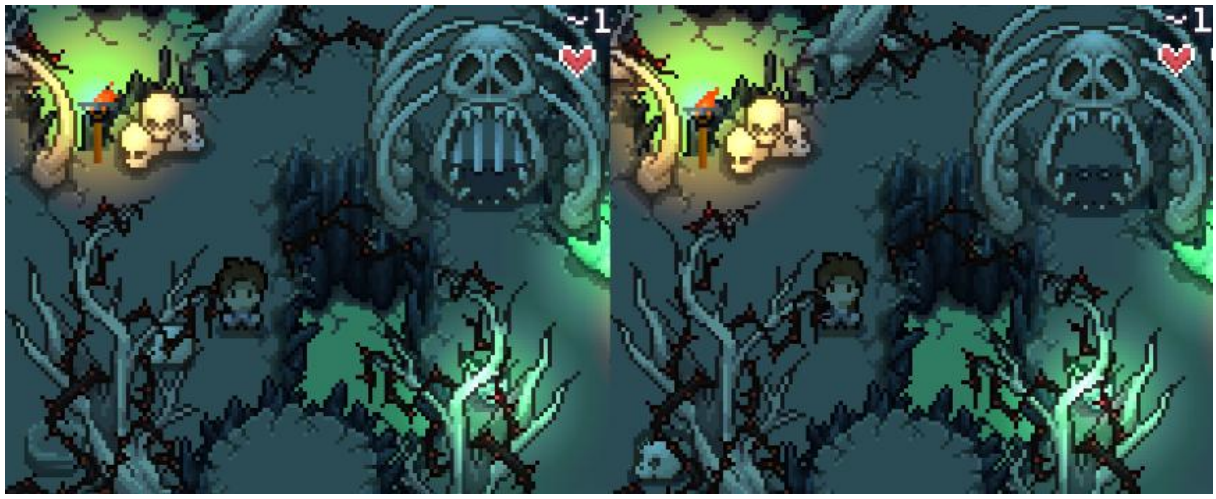
Tabela 7.10 Przypadek testowy rozwiązanie zagadki z przeciwnikiem

ID	**TC010**
Tytuł	Rozwiązanie zagadki z przeciwnikami – świetlikami.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź 6 świetlików na mapie. 2. Pokonaj wszystkie świetliki.
Oczekiwany rezultat	Na ziemi pojawia się klucz.

Rys 7.11 Po pokonaniu świetlików pojawia się klucz

Tabela 7.11 Przypadek testowy rozwiązanie zagadki z mechanizmem

ID	**TC011**
Tytuł	Rozwiązanie zagadki z płytką naciskową.
Warunki początkowe	Gra jest uruchomiona, gracz znajduje się na jednym z poziomów.
Kroki testowe	1. Znajdź bramę. 2. Podejdź do bramy i spróbuj przez nią przejść. 3. Znajdź czaszkę. 4. Znajdź płytkę naciskową. 5. Używając klawiszy WASD przesuń czaszkę na płytkę naciskową.
Oczekiwany rezultat	Brama pozostaje zamknięta blokując przejście. Po przesunięciu czaszki na płytkę naciskową brama otwiera się.



Rys 7.12 Otwarcie bramy za pomocą płytki naciskowej

8. Podsumowanie i wnioski

Celem niniejszej pracy, było zaprojektowanie oraz implementacja gry przygodowej 2D, która łączy w sobie elementy zagadkowe i zręcznościowe, osadzone w klimatycznym, mrocznym świecie jaskiń. Proces projektowy uwzględniał zarówno aspekty wizualne, jak i techniczne, a głównym założeniem było stworzenie interesującego środowiska gry, które zachęci gracza do kreatywnego myślenia, eksploracji i podejmowania wyzwań.

Wszystkie wymagania określone w rozdziałach 2.1 oraz 2.2 zostały pomyślnie zrealizowane, co potwierdza pełną zgodność końcowego produktu z założeniami projektowymi. Interfejs użytkownika został zaprojektowany w sposób intuicyjny i spójny z klimatem gry. Mechaniki rozgrywki, takie jak system zagadek, eksploracja czy elementy zręcznościowe, zostały starannie zaimplementowane i przetestowane. Grafiki oraz efekty dźwiękowe, wykorzystane w projekcie, bazują na zasobach dostępnych na licencji Open-Source, co pozwoliło na zachowanie wysokiej jakości wizualnej i akustycznej, przy minimalnych kosztach. Część grafik wykonana została własnoręcznie, aby dopasować elementy do szaty graficznej występującej w grze.

Logika gry została opracowana przy użyciu skryptów, napisanych w języku GDScript. Jest to język programowania dedykowany silnikowi Godot, bazujący na składni Pythona. Dzięki zastosowaniu GDScript, możliwa była niemal bezproblemowa integracja mechanik i logiki rozgrywki z elementami wizualnymi. Wykorzystanie silnika Godot oraz języka skryptowego dostosowanego do jego możliwości, pozwoliło na optymalizację procesu tworzenia gry, jednocześnie umożliwiając realizację złożonych funkcjonalności w sposób przejrzysty i efektywny.

Gra posiada potencjał do dalszego rozwoju i usprawnień. Podział każdego elementu gry na osobne komponenty, pozwala na dodanie nowych mechanik, bez konieczności ingerencji w istniejące już skrypty. Gra może zostać rozbudowana o dodatkowe funkcje, takie jak: system wytwarzania przedmiotów, nowe rodzaje broni, dzienniczek z listą zadań do wykonania oraz komnaty z silnymi przeciwnikami – bossami, a także mechaniki walki, pozwalające na strzelanie z łuku i posługiwanie się magią.

Bibliografia

[1] Wywiad z twórcą
https://80.lv/articles/godot2-interview/

[2] Chris Bradfield, Packt Publishing, Godot Engine Game Development Projects, 2018

[3] Juan Linietsky, Ariel Manzur and the Godot community, 2016
https://docs.huihoo.com/godotengine/godot-docs/godot.pdf

[4] Forum UnrealEngine
https://forums.unrealengine.com/t/difference-between-using-sav-files-and-using-a-custom-save-file-with-fmemorywriter-reader/477345

[5] File Extensions Wiki
https://file-extensions.fandom.com/wiki/Sav

[6] Forum ttlg
https://www.ttlg.com/forums/showthread.php?t=152326&s=856fef27592d3a0bade288bffaad6270

[7] Dokumentacja JSON
https://www.json.org/json-en.html

Spis rysunków

Rys. 3.1 Słownik zawierający scenę, położenie oraz zdrowie gracza, przedmioty oraz elementy trwałe.

Rys. 4.1 Obszar otrzymywania obrażeń

Rys. 4.2 Fragment kodu odpowiedzialny za otrzymywanie obrażeń, aktualizację zdrowia i statusu niewrażliwości

Rys. 4.3 Fragment kodu przedstawiający sygnały wejścia oraz wyjścia z obszaru widzenia oraz sygnał zmiany kierunku.

Rys. 4.4 Funkcja ustawiająca kierunek

Rys. 4.5 Funkcje sprawdzające czy obiekt jest graczem i emitujące sygnały wyjścia i wejścia w obszar interakcji

Rys. 4.6 Ustawienie tekstury, etykiety i statusu otwarcia skrzyni

Rys. 4.7 Funkcja ustawiająca animację skrzyni.

Rys. 4.8 Funkcje emitujące sygnały o interakcji.

Rys 4.9 Funkcje łączące z sygnałem wyjścia i wejścia w obszar interakcji

Rys 4.10 Funkcja odpowiadająca za otwarcie bramy za pomocą klucza

Rys. 4.11 Funkcja zamykająca bramę oraz funkcja ustawiająca stan otwarcia

Rys. 4.12 Sygnały wyjścia i wejścia w obszar interakcji

Rys 4.13 Inicjalizacja sygnałów

Rys. 4.14 Funkcje odpowiedzialne za rejestrowanie wejścia i wyjścia z obszaru interakcji

Rys. 4.15 Sprawdzenie stanu aktywacji płytki naciskowej

Rys 4.16 Odtwarzanie dźwięku

Rys 5.1 Główne menu

Rys. 5.2 Pasek pełnego zdrowia gracza

Rys. 5.3 Pasek zdrowia gracza po otrzymaniu obrażeń

Rys 5.4 Menu pauzy

Rys 5.5 Ekwipunek wraz z przedmiotami i ich opisami

Rys 5.6 Ekran śmierci

Rys 5.7 Fragment animacji po ukończeniu gry

Rys 5.8 Ekran ukończenia gry

Rys. 6.1 Klucz, który gracz może podnieść

Rys 6.2 Czaszka

Rys. 6.3 Płytka naciskowa wraz z bramą.

Rys. 6.4 Aktywowanie mechanizmu otwierającego bramę.

Rys 6.5 Zamknięta skrzynia

Rys 6.6 Interakcja gracza ze skrzynią

Rys 6.7 Otwarta skrzynia

Rys 6.8 Kamień blokujący dostęp do zagadki

Rys 6.9 Zniszczenie kamienia atakiem

Rys 6.10 Zagadka z runami

Rys 6.11 Sekwencja run zgodnie z kolejnością eksploracji

Rys 6.12 Nieaktywny portal

Rys 6.13 Aktywowany portal

Rys 6.14 Klucz do bramy

Rys 6.15 Klucz wykorzystany do otwarcia

Rys 7.1 Animacja bezczynności dla kierunków dół oraz prawo

Rys 7.2 Animacja poruszania się dla kierunku w prawo

Rys 7.3 Płynne przejście między animacjami chodzenia, ataku i bezczynności

Rys 7.4 Animacja ataku gracza i ogłuszenia przeciwnika

Rys 7.5 Animacja otrzymania obrażeń oraz redukcja punktów zdrowia

Rys 7.6 Animacja śmierci gracza

Rys 7.7 Ekran przegranej po śmierci gracza

Rys 7.8 Zniszczenie krzaka blokującego przejście atakiem

Rys 7.9 Otwarcie skrzyni

Rys 7.10 Dodanie podniesionych kluczy do ekwipunku

Rys 7.12 Po przesunięciu czaszki na płytkę naciskową brama otworzyła się

Spis tabel

Tabela 6.1 Przypadek testowy animacja bezczynności

Tabela 6.2 Przypadek testowy poruszania postacią

Tabela 6.3 Przypadek testowy przejścia między stanami

Tabela 6.4 Przypadek testowy ataku gracza

Tabela 6.5 Przypadek testowy otrzymywania obrażeń

Tabela 6.6 Przypadek testowy śmierci gracza

Tabela 6.7 Przypadek testowy interakcji z otoczeniem

Tabela 6.8 Przypadek testowy otwierania skrzyni

Tabela 6.9 Przypadek testowy dodania przedmiotu do ekwipunku

Tabela 6.10 Przypadek testowy rozwiązanie zagadki z przeciwnikiem

Tabela 6.11 Przypadek testowy rozwiązanie zagadki z mechanizmem