Kierunek: ***Informatyka***

2025/2026

Oskar Dąbrowski

PRACA INŻYNIERSKA

Projekt i implementacja gry komputerowej z dynamicznym systemem walki opartym o muzykę

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, 2026

Spis treści

Spis treści 2
1. Wstęp 5
1.1. Cel pracy 5
1.2. Zakres pracy 5
2. Analiza biznesowa 6
2.1. Wymagania funkcjonalne 6
2.2. Wymagania niefunkcjonalne 6
2.3. Analiza rynku i konkurencji 7
3. Stos technologiczny 8
3.1. Silnik gier Godot 8
3.2. GDScript 8
3.3. Plik .cfg 9
3.4. Plik BeatBeat 9
3.5. JSON 10
3.6. Zasoby wizualne i dźwiękowe 10
4. Implementacja projektu 12
4.1. Architektura 12
4.2. Obsługa plików BeatBeat 13
4.3. Generowanie mapy rytmicznej 15
4.4. Wprowadzenie rozgrywki przez sieć 16
4.5. System synchronizacji rozgrywki z muzyką 18
4.6. Systemy poruszania się i interakcji graczy 20
5. Interfejs użytkownika 22
5.1. Menu startowe 22
5.2. Wybór poziomu 23
5.3. Pokój gracza 25
5.4. Rozgrywka 26
5.5. Wyniki z rozgrywki 27
6. Testowanie 28
6.1. Działanie przycisków 28
6.2. Tworzenie pokoju gry i dołączanie do niego 29
6.3. Sprawdzanie późniejszego rozpoczęcia gry 29
6.4. Interakcja między graczami 30
6.5. Zliczanie punktów 30
6.6. Opuszczenie gry przez gracza 31
6.7. Włączenie kolejnej gry po rozegraniu poprzedniej 31
7. Podsumowanie i wnioski 32

Bibliografia **34**
Spis rysunków **35**
Spis tabel **36**
Spis listingów **37**

1. Wstęp

Współczesna branża gier komputerowych poszukuje ciekawych mechanik oraz oryginalnych pomysłów wyróżniających gry na tle innych. Jednym z często ostatnio występujących kierunków rozwoju jest wykorzystywanie dźwięku nie tylko jako tła do rozgrywki ale jako spójnego elementu rozgrywki. Gry rytmiczne zaczynały jako proste symulatory tańca, dzisiaj możemy zaobserwować produkcje takie jak *Hi-Fi Rush* czy *Metal:Hellsinger*, które wykorzystują muzykę jako kluczowy element rozgrywki.

1.1. Cel pracy

Celem pracy inżynierskiej jest zaprojektowanie i implementacja komputerowej gry rytmicznej typu PvP (gracz kontra gracz), w której system walki jest dostosowywany do analizy odtwarzanej muzyki. Gra ma umożliwiać rywalizację dwóch graczy poprzez wykonywanie akcji zsynchronizowanych z rytmem utworu, co wymaga od uczestników precyzyjnych reakcji i wyczucia rytmu.

1.2. Zakres pracy

Zakres pracy obejmuje pełny proces produkcji gry, od analiz teoretycznych po testy końcowe. Kluczowymi punktami są:

- Analiza literatury i istniejących rozwiązań - Analiza gier rytmicznych oraz metod synchronizacji dźwięku z obrazem w grze.
- Wybór technologii - Uzasadnienie wyboru narzędzi tj. silnik do gier lub narzędzi do obsługi audio lub testów aplikacji.
- Projektowanie systemów - Projektowanie mechaniki walki, przetwarzania dźwięku, synchronizacja z grą i innymi graczami.
- Implementacja - Pisanie skryptów odpowiedzialnych za logikę, integracja zasobów i przygotowanie warstwy wizualnej, ocena rytmiczności działań gracza.
- Testy - Przeprowadzenie testów i weryfikacja gry pod kątem grywalności i stabilności.
- Podsumowanie i wnioski - Ocena realizacji założonych celów oraz zaproponowanie dalszego rozwoju projektu.

2. Analiza biznesowa

Projekt zakłada wykonanie gry rytmicznej z elementami walki, w której użytkownik sam dobiera parametry pod własną rozgrywkę np. rodzaj muzyki, trudność utworu. Celem analizy jest wykazać wymagania funkcjonalne i niefunkcjonalne które pozwolą na płynną i satysfakcjonującą rozgrywkę.

2.1. Wymagania funkcjonalne

Rozgrywka

Gracz posiada kontrolę nad jedną z postaci na planszy gry, ma możliwość poruszania się i wykonywania niektórych akcji w rytm granego utworu np. unik, blok, atak. Poszczególne akcje mają swoje konsekwencje i wprowadzają złożoność do gry. Sam teren gry reaguje na rytm i może tworzyć przeszkody lub ułatwiać rozgrywkę niektórym graczom.

Przygotowanie gry

Przed rozpoczęciem rozgrywki gracz znajduje się w ekranie startowym aplikacji gdzie może dobierać ustawienia gry. Może on wybrać głośność granego utworu oraz wybrać i dostosować utwór do swoich upodobań.

Tworzenie mapy rytmicznej utworu

Poza samą grą w aplikacji możemy znaleźć prosty edytor do tworzenia własnych map rytmicznych wykorzystując własną muzykę i grafiki. Efektem końcowym edytora jest utworzenie pliku z mapą i wszystkimi jej zasobami.

2.2. Wymagania niefunkcjonalne

Wydajność

Gry rytmiczne oraz bijatyki muszą pracować w bardzo dobrze zsynchronizowanych systemach by zapewnić responsywność gry na interakcje graczy. Efekty poszczególnych interakcji zależą od reakcji rzędu kilku milisekund co sprawia, że wydajność gry jest najważniejszym elementem.

Sterowanie

Gracz powinien czuć się komfortowo kontrolując swoją postać. Dlatego rozstawienie klawiszy powinno być znane graczowi z innych gier. Sama ilość dostępnych klawiszy również powinna być ograniczona aby skupić się na rozgrywce.

Estetyka

Wizualna strona gry powinna być przejrzysta aby gracz szybko zrozumiał co dzieje się na ekranie. Rozgrywka dzieje się w środowisku dwuwymiarowym (2D) co ułatwia jej odbiór. Styl artystyczny także pozwala na łatwiejsze odnalezienie się w grze, ale nadal pozwala na dużą ekspresję wizualną otoczenia.

2.3. Analiza rynku i konkurencji

Gatunki pod które można przypisać grę są rzadko najpopularniejszymi typami gier. Jednymi z najpopularniejszych gier rytmicznych są gry *osu!* oraz *Beat Saber*. Zazwyczaj gry rytmiczne posiadają własne unikalne mechaniki co powoduje że są chętnie wybierane przez osoby szukające nowych wyzwań.

Drugim gatunkiem jakim są bijatyki (*Beat'em up*), które w przeciwieństwie do gier rytmicznych trzymają się standardów w tym gatunku. Seria gier *Super Smash Bros.* od firmy *Nintendo* albo gry *Tekken* wprowadzają nową zawartość poprzez nowe postacie bądź poziomy nie naruszając standardowych zasad.

Z uwagi na unikalne połączenie gatunków jakimi są gry rytmiczne i bijatyki gra powinna zostać wydana jako tytuł darmowy z dodatkowymi elementami płatniczymi. Dodatkową zawartością mogą być elementy kosmetyczne jak wygląd postaci gracza bądź dedykowane poziomy stworzone z pomocą znanych twórców muzyki. Darmowa gra pozwala na zdobycie większej liczby graczy jak i pozwala sprawdzić zapotrzebowanie tego typu gatunku gry na rynku.

3. Stos technologiczny

Rozdział skupia się na przedstawieniu narzędzi, technologii i zasobów użytych przy tworzeniu aplikacji. Głównymi punktami będą narzędzia zintegrowane z silnikiem gier.

3.1. Silnik gier Godot

Godot Engine to silnik do tworzenia gier 2D i 3D stworzony został przez Juana Linietsky'ego i Ariela Manzura. Silnik został wydany pod licencją MIT[1] w 2014 roku na platformie GitHub[2]. Silnik Godot jest rozwijany jako projekt *open-source* co pozwala społeczności na wprowadzanie aktualizacji czy naprawianie błędów. Godot posiada także własną dobrze rozbudowaną dokumentację[3].

Na tle innych silników do gier wyróżnia go system scen i węzłów, który pozwala na łatwe zarządzanie i tworzenie bardziej skomplikowanych komponentów. Silnik pozwala na wizualne budowanie poziomów i komponentów oraz posiada zintegrowany system UI co pozwala na łatwą implementację interfejsu użytkownika. Godot obsługuje również kilka języków programowania, takie jak C++, C# oraz GDScript.

3.2. GDScript

Język programowania GDScript został wprowadzony jako podstawowy język silnika Godot. Wzorowany wizualnie na podstawie języka Python[4] sprawia, że jest intuicyjny i łatwy do nauki. Integracja z silnikiem sprawia że mamy łatwy dostęp do elementów silnika które przedstawiane są wizualnie i możemy się do nich odnosić bez dodatkowej edycji w kodzie.

Pomimo podobieństwa do języka Python nie jest on od niego zależny. GDScript jest językiem dynamicznym co oznacza, że zmienne w kodzie przyjmują typ danych zależny od ich zawartości a nie odgórnie ustalany przy ich deklaracji. Powoduje to, że kod jest łatwiejszy w pisaniu i zrozumieniu. Niestety dużą wadą jest mniejsza wydajność kodu.

3.3. Plik .cfg

Jest to wbudowana w silnik klasa pomocnicza która pozwala na zapisywanie danych na dysku. Pliki te przypominają pliki konfiguracyjne INI które pełnią tą samą funkcje w innych aplikacjach.Zapisywane są w nich dane odpowiedzialne za ustawienia programów. Same dane są zapisywane w zwykłym pliku tekstowym jednak ich formatowanie pozwala na odczytanie kilku właściwości. Dane przedstawiane są za pomocą sekcji, kluczy i wartości.

Listing 3.1. przykładowa zawartość pliku .cfg

```
[sekcja]
liczba=42
tekst="Przykładowy tekst"
wektor=Vector3(1, 0, 2)
```

Dane grupowane są za pomocą sekcji dzięki czemu możemy łatwo znaleźć opcje, które nas interesują. Klucze opisują co dana wartość przedstawia. Wartości mogą przyjmować różne typy danych. W listingu 3.1 przedstawiono typ liczbowy, tekstowy oraz bardziej złożony wektor.

3.4. Plik BeatBeat

Jest to plik tworzony i czytany przez grę, który zawiera wszystkie informacje odnośnie poziomu. Jest on produktem końcowym edytora, który zawiera dane wprowadzone przez użytkownika i zapisane w jednym pliku binarnym z rozszerzeniem *.btbt*. Rozwiązanie to pozwala na łatwe dzielenie się z innymi użytkownikami gotowymi do gry mapami.

Co znajduje się w pliku BeatBeat ?:

- Plik audio w formacie .ogg lub .wav.
- Obraz w formacie .png.
- Dane tekstowe np. autor, nazwa utworu oraz mapa rytmiczna w formacie .json.
- Dane do odczytu pliku binarnego np.długość poszczególnych elementów w bitach.

Pliki BeatBeat są przechowywane w folderze AppData gdzie programy zapisują swoje dane. Gwarantuje to zabezpieczenie plików przed przypadkowym usunięciem. Dodatkowo pozwala nam to na szybką edycję naszego zbioru map jak i ich przesłanie innemu użytkownikowi.

3.5. JSON

JSON (*JavaScript Object Notation*) to format wymiany danych, który jest łatwy do czytania dla człowieka jak i maszyny[5]. Został on stworzony w oparciu o obiekty w języku JavaScript. Format ten jest elastyczny i łatwy do zaimplementowania w wielu językach programistycznych co czyni go bardzo popularnym i często używanym w wielu aplikacjach.

Format JSON składa się z kluczy i wartości. Klucze występują w formie tekstu natomiast wartości mogą przyjmować różne typy danych. Przykładowymi typami danych są:

- String (tekst)

```
{
 "imie": "Adam"
}
```

- Number (liczba)

```
{
 "wiek": 20
}
```

- Array (zbiór tekstów)

```
{
 "meble": ["szafa", "komoda", "stolik"]
}
```

3.6. Zasoby wizualne i dźwiękowe

W aplikacji zostały wykorzystane zasoby dozwolone do darmowego użytku ze stron internetowych np. *https://itch.io/*. Silnik Godot pozwala na łatwe importowanie tych zasobów co usprawnia pracę. Gotowe klatki animacji w połączeniu z łatwym w obsłudze systemem animacji pozwalają na szybkie tworzenie całych animowanych postaci lub środowisk.

Użytkownicy sami będą dostarczali utwory muzyczne do gry jednak nadal niezbędnym elementem jest strona wizualna. Mowa tutaj o np. budowie poziomu, wyglądzie postaci i tekstur. Dodanie tych detali pozwala na większe zaangażowanie w rozgrywkę.

Wykorzystanie darmowych zasobów pozwala na szybsze tworzenie projektu. Autorzy grafik bądź utworów w zamian mogą zostać wspomnieni w projekcie jako oryginalni autorzy prac. Pozwala to rozpromować ich pracę w zamian za łatwiejszą pracę nad projektem.

Tabela 3.1 Wykorzystane darmowe zasoby

Typ zasobu	Nazwa	Autor/źródło	Licencja
Czcionka	Pixel Game Font Family	suhadidesign [6]	Darmowa do użytku własnego.
Dźwięk	Countdown for race, game start	Pastew [7]	Creative commons 0
Utwory muzyczne	Różne utwory wykonawcy	The Fat Rat [8]	Darmowa do użytku własnego.
Tilemap	Legacy fantasy	Anokolisa [9]	Darmowa do użytku własnego.
Model postaci	Samurai 2D Pixel art	Mattz Art [10]	Darmowa do użytku własnego.

4. Implementacja projektu

W tym rozdziale zostanie przedstawiona implementacja i zastosowane rozwiązania na problemy jakie zostały napotkane w trakcie tworzenia aplikacji.

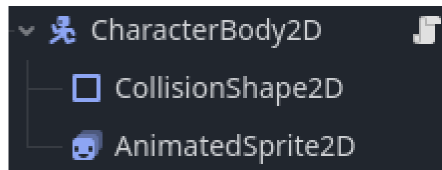
4.1. Architektura

Całość projektu została zbudowana wewnątrz Godot Engine i jednym z kluczowych elementów silnika jest system *Node'ów* (węzłów) i scen. Wszystkie elementy gry składają się z węzłów. Daje to możliwość wykonywania bardziej skomplikowanych modułów jak i prostszych pojedynczych elementów.

Węzły podzielone zostały w hierarchii drzewa gdzie każdy z nich dziedziczy po innym węźle. Najwyższym elementem w drzewie jest najprostszy węzeł *Node* od którego dziedziczą wszystkie inne elementy. Schodząc niżej w hierarchii znajdziemy bardziej skomplikowane węzły.

W silniku zaimplementowano wiele węzłów, które automatycznie tworzą już najczęściej potrzebne w grach koncepcje. Przykładowo *AnimatedSprite2D* jest elementem który pozwoli wprowadzić dwuwymiarowy obraz z animacjami. W samym węźle mamy zintegrowany system animacji pozwalający je tworzyć.

Ciekawym rozwiązaniem jest możliwość dodawania do node'ów skryptów. Pozwala to na łatwą edycję zachowania elementu bez potrzeby odwoływania się do elementu, w którym osadzony jest skrypt.



Rys. 4.1. Przykładowa budowa węzła grywalnej postaci.

Węzły mogą być przydzielane innym węzłom co pozwala na dziedziczenie niektórych ich funkcji na przykład położenia na scenie. Niektóre węzły potrzebują innych węzłów aby działały poprawnie. Przykładowo na rysunku 4.1 węzeł *CharacterBody2D* potrzebuje node'a odpowiedzialnego za kolizje postaci, tutaj jako przykład dodano *CollisionShape2D*.

Kolejnym elementem wartym zrozumienia jest system scen w silniku. Nazwa sugerowałaby, że jest to cały obszar gry dostępny w aktualnym oknie i jest to po części prawda. Sceną może być również pewien element zbudowany z wielu węzłów który został zapisany wcześniej w plikach gry. Sceny w godocie to gotowe do użycia zbiory komponentów jak i pliki przedstawiające całokształt aplikacji. Co powoduje, że sceny mogą występować w większych scenach.

4.2. Obsługa plików BeatBeat

Aplikacja obsługuje plik BeatBeat (btbt) oraz pozwala na jego wygenerowanie na podstawie działań w edytorze. Skrypt w aplikacji *BeatMapLoader.gd* pozwala na rozpoznanie i wprowadzenie danych z pliku do gry.

Listing 4.1. Fragment kodu dodający rozpoznawanie pliku btbt.

```
extends ResourceFormatLoader
#what type of extension read
func _get_recognized_extensions() -> PackedStringArray:
   return ["btbt"]
#what resource it will create
func _get_resource_type(_path: String) -> String:
   return "Beatmap"
```

Funkcje przedstawione na listingu 4.1 pozwalają nam dodać do aplikacji rozpoznawanie rozszerzenia pliku i w jaki sposób mają zostać zinterpretowane. Tutaj mają być zmienione w zasób Beatmap. Silnik samodzielnie potrafi nadawać typy zmiennym jednak jeżeli chcemy aby nadawał nasz własny typ trzeba go zdefiniować w kodzie.

Listing 4.2. Funkcja zmieniająca plik binarny w zasób w silniku.

```
func _load(path: String, _riginal_path: String, _use_sub_threads: bool, _cache_mode: int) -> Resource:
	var f = FileAccess.open(path, FileAccess.READ)
	if f == null:
		push_error("Cannot open file: " + path)
		return null

	var id = f.get_buffer(4).get_string_from_utf8()
	if id != "BTBT":
		push_error("Wrong format")
		return null

	var json_size = f.get_32()
	var json_data = JSON.parse_string(f.get_buffer(json_size).get_string_from_utf8())

	var audio_size = f.get_32()
	var audio_bytes = f.get_buffer(audio_size)

	var image_size = f.get_32()
	var image_bytes = f.get_buffer(image_size)

	var beatmap = Beatmap.new()
	beatmap.metadata = json_data
	beatmap.audio_data = audio_bytes
	beatmap.image_data = image_bytes

	return beatmap
```

Funkcja przedstawiona w listingu 4.2 otwiera plik binarny i zaczyna czytać plik. Pierwszym elementem jest identyfikator, który znajduje się na początku pliku i służy do identyfikacji pliku. Kolejnym elementem jest długość jaką zajmuje kolejny element pliku. Poza stałym elementem w postaci identyfikatora każda kolejna wartość musi być poprzedzona jego długością. Pozwala to na poprawne odczytanie danych z pliku. Wyciągane zostają kolejne elementy tj. dane w formacie json, plik audio, plik z obrazem. Zostaje wygenerowany nowy zasób beatmap i przypisane zostają do niego potrzebne dane. Na samym końcu zasób zostaje zwrócony jako wynik funkcji.

Listing 4.3. Przypisanie pliku do zasobu.

```
var beatmap = ResourceLoader.load(MapFolderCreate.folder+"/"+file_name)
```

Po wprowadzeniu skryptu w ustawieniach projektu jako unikalna instancja klasy mamy dostęp do pliku za pomocą jednej linii kodu. Listing 4.3 przedstawia przypisanie pliku za pomocą ścieżki do niego. Pomimo braku deklaracji typu zmienna beatmap przyjmuje odpowiedni typ zadeklarowany we wcześniejszym skrypcie z listingu 4.2.

4.3. Generowanie mapy rytmicznej

Ważnym elementem projektu było zbudowanie automatycznego systemu detekcji rytmu w utworze. Pozwoli to na szybkie generowanie map gotowych do gry. Gracze mogą od razu przystąpić do rozgrywki po wprowadzeniu podstawowych danych.

Została napisana funkcja która otrzymuje plik dźwiękowy do analizy i ma ona za zadanie zwrócić ilość uderzeń na minutę (*Beats Per Minute*). Sprawdzała ona energię w wybranym zakresie częstotliwości. Dla zobrazowania zakres przedstawiał pojedynczy instrument w utworze np. perkusję i na podstawie jego zmian w głośności ustalić można ilość uderzeń na minutę.

Pierwsze testy przeprowadzane były na utworze *Hiding In The Blue* autorstwa *TheFatRat* którego BPM wynosi 75. Kilka wyników sprawiało problemy, które pojawiły się w trakcie testowania rozwiązania.

Częsty wynik 150 BPM, który jest dwukrotnością ilości uderzeń w tym utworze, pokazał pewną zależność w muzyce. Instrumenty mogą przyspieszyć lub zwolnić swoje tempo w utworze dwukrotnie co nazywane jest jako *double time* lub *half time*. Powoduje to trudny do wykrycia błąd w analizie utworu.

Kolejnym problemem były wyniki bliskie aktualnej wartości. Wyniki zbliżone nie są dopuszczalne do wykorzystania, ponieważ dodatkowe uderzenia powodują brak synchronizacji tempa mapy z rytmem utworu.

Ostatnim problemem jest identyfikacja w jakim przedziale częstotliwości sprawdzana jest energia. Nie każdy utwór posiada ten sam instrument przewodni który nadaje rytm. W zależności od utworu może być trudniejsze bądź prostsze zidentyfikowanie instrumentu przewodniego.

Pomysł automatycznego generowania mapy musiał zostać zastąpiony innym rozwiązaniem. Zaimplementowany więc został edytor dostępny w aplikacji gdzie użytkownik będzie musiał sam podać liczbę uderzeń. Pozwoli to na wygenerowanie poprawnej mapy, która będzie gotowa do rozgrywki.

Listing 4.4. Fragment skryptu tworzenia pliku.

```
#JSON data
var json_data = JSON.stringify(data)
var json_bytes = json_data.to_utf8_buffer()
file.store_32(json_bytes.size())
file.store_buffer(json_bytes)

# Audio
var audio_bytes = FileAccess.get_file_as_bytes(audio_path)
file.store_32(audio_bytes.size())
file.store_buffer(audio_bytes)

# Image
var image_bytes = FileAccess.get_file_as_bytes(image_path)
file.store_32(image_bytes.size())
file.store_buffer(image_bytes)
```

Po podaniu podstawowych informacji w interfejsie zostanie stworzony plik w formacie *btbt*. Plik znajduje się w utworzonym przez grę folderze i można go wykorzystać do rozpoczęcia rozgrywki.

4.4. Wprowadzenie rozgrywki przez sieć

Gra przeznaczona jest dla maksymalnie czterech graczy co wymusza aby rozgrywka odbywała się na wielu komputerach. Ze względu na dużą dokładność w akcjach graczy gra wspiera połączenie tylko w sieciach lokalnych (ang. Local Area Network, LAN).

Listing 4.5. Funkcje broadcast

```
func start_broadcasting():
    isBroadcasting=true
    udp_broadcast = PacketPeerUDP.new()
    udp_broadcast.set_broadcast_enabled(true)
    var timer = get_tree().create_timer(1.0)
    timer.timeout.connect(_on_broadcast_timer)

func _on_broadcast_timer():
    if multiplayer.is_server() and isBroadcasting:
        var data = room_name + "|" + str(players.size()) + "|" + Options.PlayerNickname
        udp_broadcast.set_dest_address(broadcast_address, portUDP)
        udp_broadcast.put_packet(data.to_utf8_buffer())
        get_tree().create_timer(1.0).timeout.connect(_on_broadcast_timer)

func start_listening():
    udp_listener = PacketPeerUDP.new()
    udp_listener.bind(portUDP)
```

Na listingu 4.5 przedstawione są funkcje odpowiadające za wysyłanie oraz odbieranie pakietów UDP (ang. User Datagram Protocol). Wywołanie funkcji *start_broadcasting* spowoduje, że aplikacja będzie wysyłać dane odnośnie rozgrywki tj. nazwa utworu, ilość graczy w pokoju oraz nazwę hosta.

Funkcja *start_listening* otwiera wyznaczony port pod zmienną *portUDP* aby nasłuchiwać przychodzących danych od innych użytkowników w sieci. Pozwala to nam na odnalezienie hosta w sieci oraz dołączenie do niego.

Listing 4.6. Funkcje połączenia ENet

```
func host_game():
    players[1]={"name": Options.PlayerNickname,"ready":true,"download":true}
    peer.create_server(port, maxplayers)
    multiplayer.multiplayer_peer = peer
    start_broadcasting()

func join_game(ip):
    peer.create_client(ip, port)
    multiplayer.multiplayer_peer = peer
    players[multiplayer.get_unique_id()]={"name": Options.PlayerNickname,"ready":false,"download":false}
```

Listing 4.6 przedstawia funkcje które pozwalają na zbudowanie oraz dołączenie do serwera. Serwer ten działa na bibliotece ENet, która rozwija działanie standardowego protokołu UDP.

ENet pozwala na większą kontrolę nad danymi. Możemy zagwarantować przesył danych kiedy tego potrzebujemy. Nie jesteśmy ograniczeni tylko jednym strumieniem danych przez co możemy przesyłać więcej niż jeden pakiet informacji na raz.

4.5. System synchronizacji rozgrywki z muzyką

Dzięki odpowiednim efektom dźwiękowym i wizualnym gracze wykonują akcje w trakcie rozgrywki. Aby to umożliwić każdy z graczy musi widzieć ten sam efekt na ekranie.

Listing 4.7. Funkcja skryptu metronom

```
func _process(_delta: float) -> void:
    var position=Store.trackTime
    if position >=activeBeatStart:
        lastBeat+=1
        beat_track.emit(lastBeat)
        beatOpen.emit()
        actual_beat_position=nextBeatPosition
        nextBeatPosition+=beatDuration
        activeBeatStart=nextBeatPosition-Store.margin
    if position>=actual_beat_position:
        Options.beat_hit.emit(true)
        actual_beat_position=nextBeatPosition
    if position >=activeBeatStop:
        Options.beat_hit.emit(false)
        beat_track.emit(-1)
        beatClose.emit()
        activeBeatStop=nextBeatPosition+Store.margin
```

Funkcja *_process* z listingu 4.7 wykonuje się w każdym odświeżeniu ekranu gracza i na bieżąco aktualizuje stany zmiennych które decydują o poprawności wykonanej akcji. Najważniejszą zmienną jest aktualny czas utworu ze zmiennej *Store.trackTime,* która zawiera aktualny czas utworu.

Listing 4.8. Obliczanie aktualnego czasu utworu

```
func _process(_delta: float) -> void:
    trackTime=mainTrack.get_playback_position()
    trackTime+=AudioServer.get_time_since_last_mix()
    trackTime-=AudioServer.get_output_latency()
    if(count_track):
        SendTrackTime.emit(trackTime)
```

Funkcja *get_playback_position* pozwala nam na odczytanie w jakim momencie utworu się znajdujemy jednak aktualizacja tej zmiennej odbywa się rzadziej niż raz na odświeżenie ekranu. Aby sprecyzować wynik używamy innych zmiennych związanych z czasem utworu.

Wartość którą dodajemy do czasu utworu reprezentuje jak dawno silnik audio wysłał dane do karty dźwiękowej. Wartość odjęta przedstawia przewidywane opóźnienie wynikające z przesyłu danych dźwiękowych z gry do naszego głośnika.

4.6. Systemy poruszania się i interakcji graczy

Gracze w trakcie rozgrywki mogą wpływać na akcje oraz pozycje swojej postaci jak i innych graczy. Wszystkie akcje są synchronizowane za pomocą wcześniej wspomnianej biblioteki ENet oraz specjalnych węzłów *MultiplayerSpawner* oraz *MultiplayerSynchronizer*

Pierwszy węzeł pozwala nam połączyć funkcje ENet z kodem gry. Najprostszym przykładem będzie wielokrotne tworzenie postaci gracza na scenie w zależności od ilości graczy na serwerze. Drugi węzeł natomiast przypięty jest do obiektów tworzonych przez *MultiplayerSpawner*. Informuje on silnik co ma być odtwarzane na ekranach innych klientów. Jeżeli gracz zmieni położenie na mapie to również ta zmiana wykona się na ekranach innych użytkowników automatycznie.

Listing 4.9. Skrypt poruszania się gracza

```
var direction := Input.get_axis("ui_left", "ui_right")
  if direction !=0:
    velocity.x = move_toward(velocity.x,direction * SPEED,ACCELERATION * delta)
    if !is_attacking:
      animation.play("walk",1.0)
      animation.flip_h = direction < 0
    if(animation.flip_h):
      pivot.scale.x=-1
    else:
      pivot.scale.x=1
  else:
    velocity.x = move_toward(velocity.x, 0, FRICTION * delta)
    if !is_attacking and !is_blocking and velocity.x==0:
      animation.play("idle",1.0)
```

Podstawowe akcje gracza czyli ruch w lewo bądź w prawo oraz skok są możliwe do wykonania zawsze pomimo rytmu utworu. Postać gracza porusza się po ekranie poprzez manipulacje wartością *velocity,* która wpływa na jego symulowaną fizykę.

Listing 4.10. Skrypt akcji gracza do rytmu

```
if Input.is_action_just_pressed("attack") and can_act :
      if get_blocked<=0:
         Options.player_action.emit(name,0)
      can_act=false
   if Input.is_action_just_pressed("dodge") and can_act:
      Options.player_action.emit(name,1)
      can_act=false
   if Input.is_action_just_pressed("block") and can_act:
      Options.player_action.emit(name,2)
      can_act=false

   if free_action_window:
      if selected_action != -1:
         if selected_action == 0:
            perform_attack()
         if selected_action == 1:
            perform_dodge()
         if selected_action == 2:
            perform_block()
         selected_action=-1
```

Na listingu 4.10 zostały zaprezentowane dwie części skryptu. Pierwsza część pozwala graczowi zadeklarować chęć wykonania ataku. Jeżeli zostanie wciśnięty przycisk odpowiadający za akcję zostanie ona zadeklarowana i nie będzie jej można zmienić przez zmienną *can_act*.

Deklaracja zostaje wysłana do komponentu, który ocenia poprawność tej akcji. Sprawdza czy aktualnie okno czasowe na wykonanie akcji jest otwarte. Jeśli tak to odsyła do tego elementu pozwolenie na tą akcje.

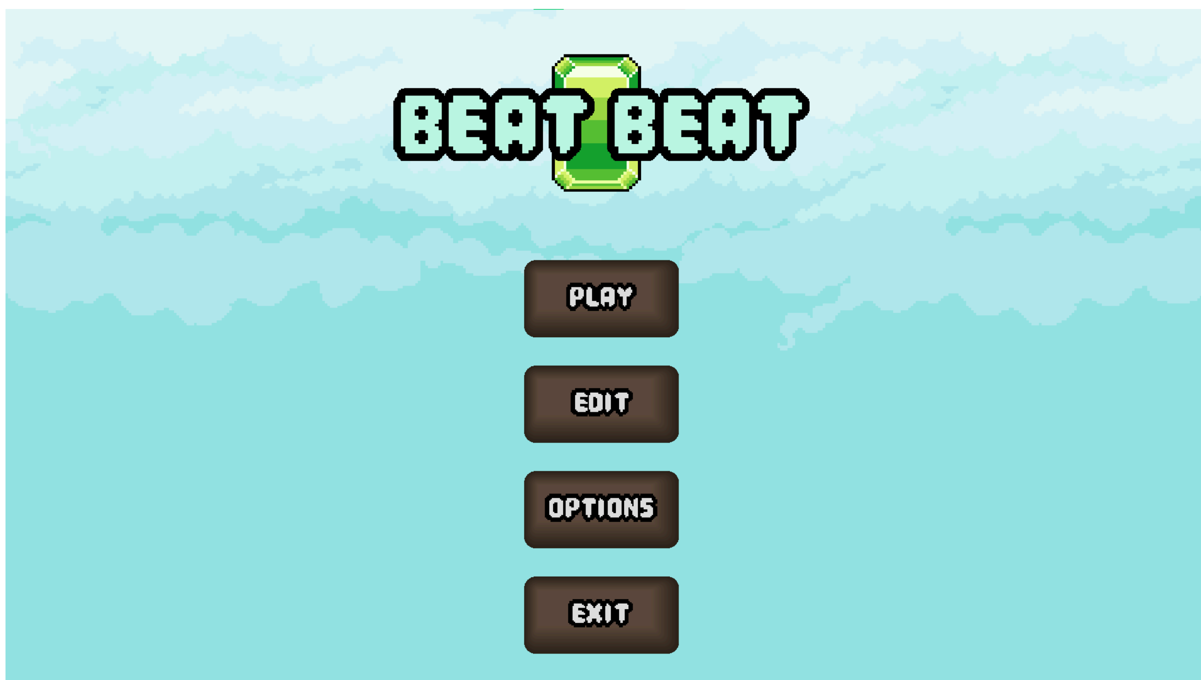
Druga część kodu wywołuje zadeklarowaną czynność, ale dzieje się to dopiero gdy następuje dokładny moment uderzenia rytmu zamiast w momencie wciśnięcia klawisza. Dzieje się tak aby każdy gracz wykonał swoją akcję w tym samym czasie.

Gracze tymi akcjami wpływają na swoje postacie. Trafienie innego gracza atakiem, odepchnie go i zada obrażenia, które zwiększą odrzut przy kolejnym ataku. Unik pozwala na szybsze poruszanie się oraz na uniknięcie obrażeń. Blok również pozwala na uniknięcie obrażeń jednak jeżeli inny gracz uderzy blokującego to atakujący nie będzie mógł wykonać dalszego ataku przez najbliższe dwa uderzenia rytmu

5. Interfejs użytkownika

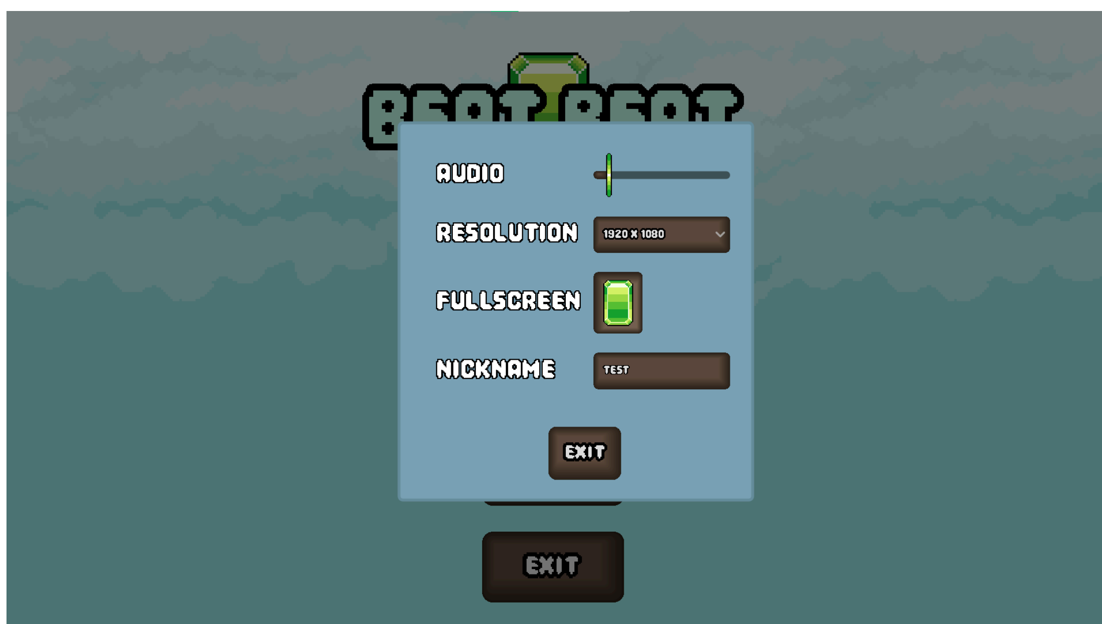
Rozdział ten zostanie poświęcony przedstawieniu oprawy graficznej aplikacji. Opisane zostanie poruszanie się po poszczególnych oknach oraz co użytkownik może w nich zrobić.

5.1. Menu startowe



Rys. 5.1. Menu początkowe gry.

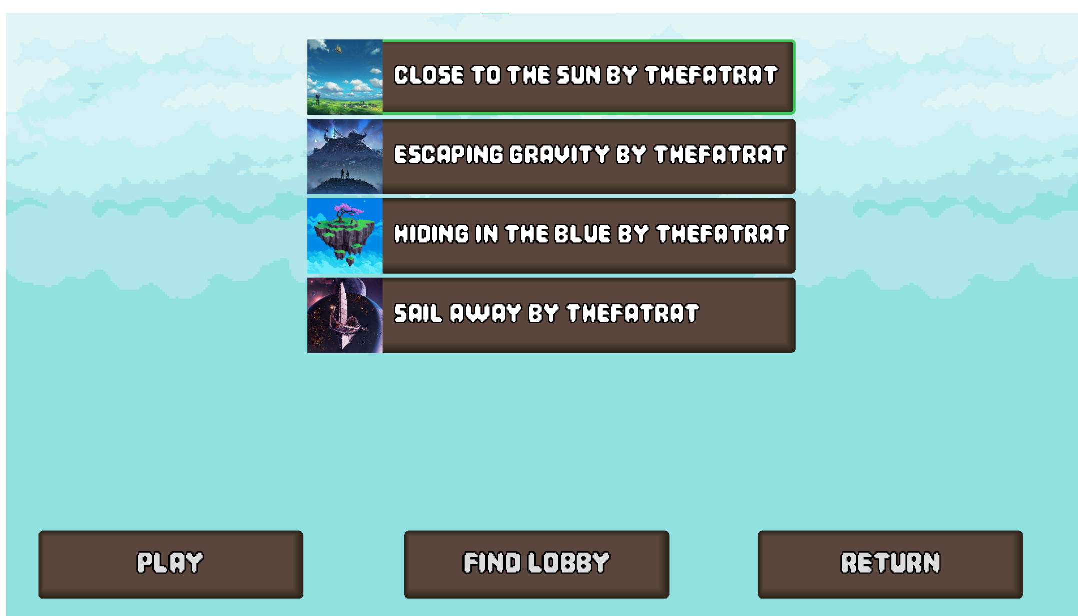
Użytkownik zostanie powitany ekranem startowym przedstawiającym tytuł gry oraz kilkoma przyciskami którymi może poruszać się po aplikacji. Interfejs gry został opisany w języku angielskim, aby aplikacja była zrozumiała dla większej grupy odbiorców. Dostępne przyciski prowadzą do różnych sekcji aplikacji odpowiadających za różne jej aspekty. Wybranie przycisku *Play* lub *Edit* przeniesie gracza do menu wyboru poziomu.



Rys. 5.2. Otwarte okno opcji.

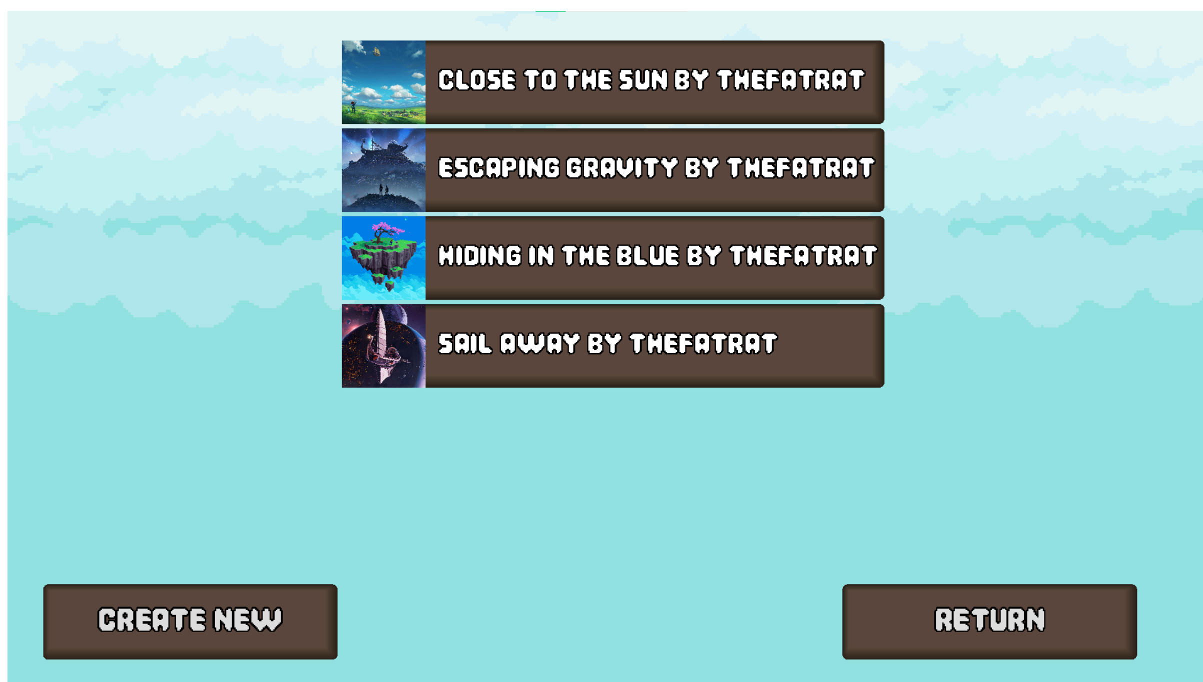
Na rysunku 5.2 przedstawione jest okno z opcjami ogólnymi, które powinny być zmieniane rzadko w aplikacji. Można tu wyróżnić głośność dźwięku lub nazwę gracza, która zostanie do niego przypisana w trakcie rozgrywki. Pozwala to na dostosowanie gry pod każdego użytkownika do jego potrzeb przed przystąpieniem do gry.

5.2. Wybór poziomu



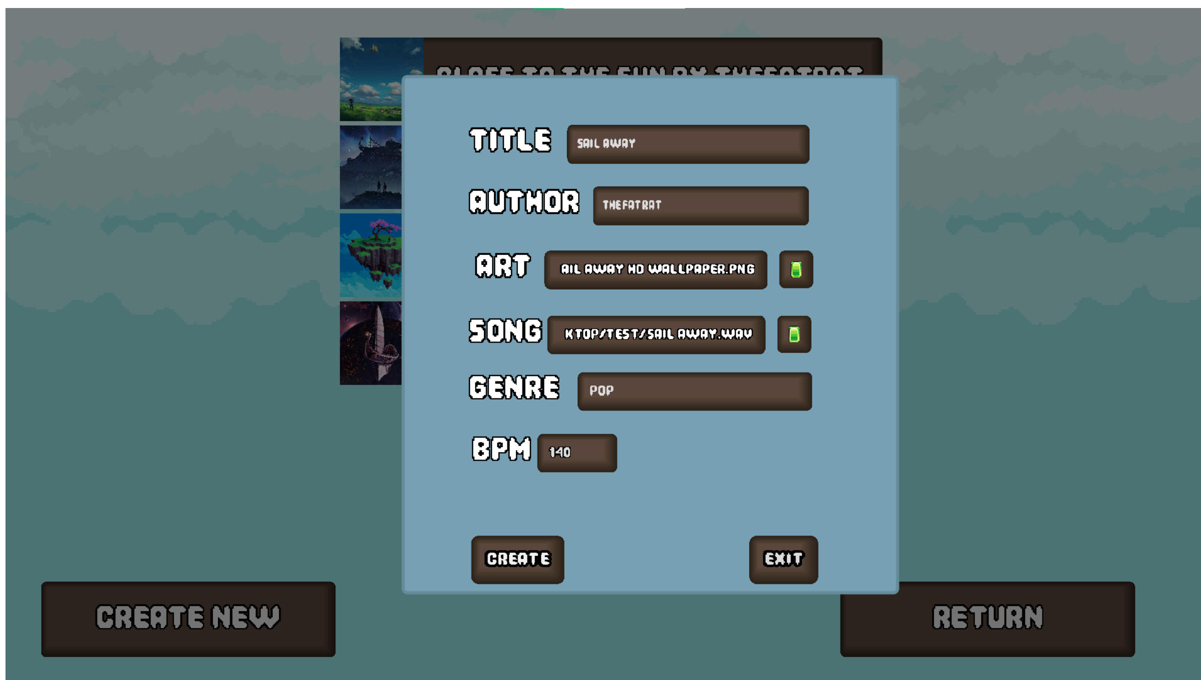
Rys. 5.3. Menu wyboru poziomu dla przycisku Play.

Menu przedstawia posiadane mapy, które można wybrać aby rozpocząć rozgrywkę. Zmienić mapę możemy klikając interesujący nas utwór. Po wybraniu utworu i kliknięciu przycisku *Play* przejdziemy do ekranu z pokojem, gdzie będą spotykać się gracze. Przycisk *Join* pozwala na wyświetlenie dostępnych pokoi stworzonych w tej sieci.



Rys. 5.4. Menu wyboru poziomu dla przycisku Edit.

Ekran edycji map jest tym samym co poprzedni. Jedyną różnicą jest inny przycisk *Create new*. Naciśnięcie przycisku przeniesie gracza do edytora gdzie może stworzyć mapę. Ekran początkowo miał służyć do implementacji edytora mapy jednak nie zostało to zaimplementowane.



Rys. 5.5. Ekran tworzenia nowej mapy.

Przycisk *Create new* otwiera okno potrzebne do stworzenia mapy gdzie wprowadza pliki jak i dane tekstowe. Po wprowadzeniu wszystkich danych mapa jest gotowa do rozpoczęcia rozgrywki.

5.3. Pokój gracza

Rys. 5.6. Pokój gracza.

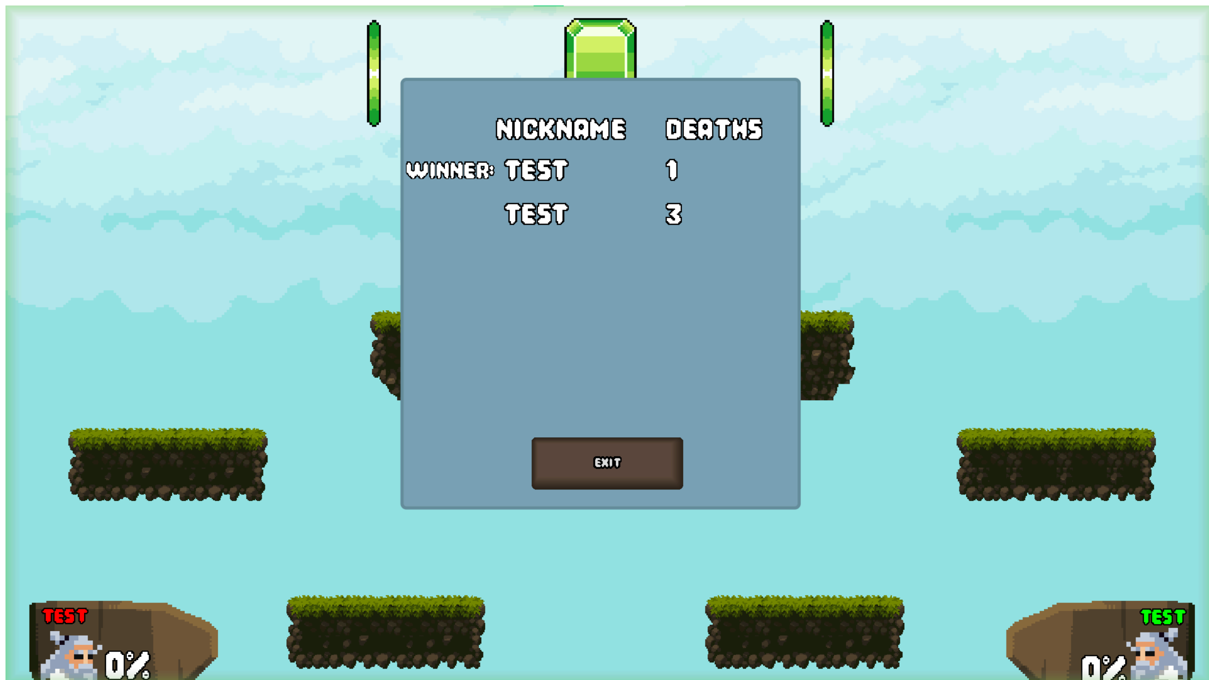
W pokoju gracza mamy wyświetlaną aktualną listę graczy, którzy dołączyli do tego pokoju. Wyświetlane są ich pseudonimy oraz stan połączenia. Na dole ekranu mamy przyciski pozwalające nam opuścić lub rozpocząć rozgrywkę. Osoba która dołączyła do pokoju zamiast przycisku *Start* będzie widziała przycisk *Ready.* Rozgrywka może zostać rozpoczęta jeżeli wszyscy gracze w pokoju będą gotowi.

5.4. Rozgrywka

Rys. 5.7. Wycinek z trwającej rozgrywki.

Rozgrywka odbywa się na planszy z platformami, na których poruszają się postacie. Wychodząc poza obszar mapy postać ginie i rozpoczyna rozgrywkę od nowa na środku mapy. Na górze ekranu widnieje zielony kryształ ze zbliżającymi się do jego wnętrza pionowymi kryształami. Wizualizuje on rytm na mapie i pozwala na określenie kiedy gracz może wykonywać akcje. Poza samym paskiem podświetlają się też narożniki ekranu. Pomaga to w wyczuciu rytmu bez potrzeby skupiania się na górnej części ekranu. W rogach ekranu mamy informacje o graczach. Możemy z nich dowiedzieć się jak łatwo będzie ich odepchnąć oraz jaki mają pseudonim w grze.

5.5. Wyniki z rozgrywki



Rys. 5.8. Tabela wyników

Kiedy utwór się kończy zostaje wyświetlona tabela z graczami i liczbą, która mówi ile razy gracz wypadł z mapy. Gracz z najmniejszym wynikiem wygrywa. Naciśnięcie klawisza exit przeniesie nas z powrotem do menu głównego.

6. Testowanie

Jednym z ważniejszych aspektów tworzenia gier jest testowanie poszczególnych elementów w celu odnalezienia problemów z systemami które mogłyby utrudniać rozgrywkę. Ważne jest też odnalezienie błędów krytycznych, które powodowałyby, że nie ma możliwości korzystania z aplikacji.

W tabelach od 6.1 do 6.7 zaprezentowano kilka przeprowadzonych testów aplikacji w celu znalezienia potencjalnych błędów z zaimplementowanymi systemami. Przedstawione zostaną w tabelach i oznaczone identyfikatorem testu (ID).

6.1. Działanie przycisków

Tabela 6.1 Scenariusz testowy działania przycisków w menu

ID	1
Tytuł	Działanie przycisków.
Warunki początkowe	Uruchomiona gra, użytkownik znajduje się w głównym menu.
Kroki testowe	1.Poruszaj się po różnych oknach gry wciskając przyciski
Oczekiwany rezultat	Gracz wciskając przycisk zostaje przeniesionych do odpowiednich scen.
Wyniki	Odpowiednie okna zostają otwarte, przyciski poprawnie blokują się przed niepoprawnym wciśnięciem.

6.2. Tworzenie pokoju gry i dołączanie do niego

Tabela 6.2 Scenariusz testowy dołączenia do pokoju gry

ID	2
Tytuł	Tworzenie pokoju gry i dołączanie do niego
Warunki początkowe	Uruchomiona gra przez 2-4 graczy, użytkownicy znajdują się w wyborze mapy
Kroki testowe	1.Jeden gracz tworzy pokój 2.Reszta graczy dołącza do pierwszego gracza 3.Gracze dołączający sprawdzają działanie pobrania mapy oraz przycisku gotowości do gry.
Oczekiwany rezultat	Wszyscy gracze mogą pobrać mapę od hosta i oznaczyć się jako gotowi do gry.
Wyniki	Gracze poprawnie pobierają mapę oraz można rozpocząć rozgrywkę. System nie działa jeśli testujemy z większą ilością niż 2 graczy na tym samym urządzeniu (blokada portów).

6.3. Sprawdzanie późniejszego rozpoczęcia gry

Tabela 6.3 Scenariusz testowy opóźnionego startu gry

ID	3
Tytuł	Sprawdzanie późniejszego rozpoczęcia gry.
Warunki początkowe	Uruchomiona gra przez 2-4 graczy, jeden z graczy posiada mniej wydajny komputer.
Kroki testowe	1.Gracze rozpoczynają rozgrywkę.
Oczekiwany rezultat	Gra zostanie rozpoczęta dopiero gdy wszyscy gracze będą gotowi do rozgrywki.
Wyniki	Gra rozpoczyna się po około sekundzie u wszystkich graczy gdy wolniejszy gracz się załadował.

6.4. Interakcja między graczami

Tabela 6.4 Scenariusz testowy interakcji między graczami

ID	4
Tytuł	Interakcja między graczami.
Warunki początkowe	Rozgrywka w trakcie
Kroki testowe	1.Gracz wykonuje atak w kierunku innego gracza. 2.Inny gracz wykonuje unik,blok lub nic nie robi. 3.Gracze powtarzają poprzednie kroki
Oczekiwany rezultat	Atakowany gracz zostaje odepchnięty przy ataku kiedy nic nie robi, gracz nie otrzymuje obrażeń przy uniku, gracz blokując atak nakłada na atakującego blokadę na ataki.
Wyniki	Przy uniku gracz nie otrzymuje obrażeń, zablokowany gracz otrzymuje blokadę, gdy gracz otrzymuje obrażenia w rzadkich przypadkach atak zostaje zadany dwukrotnie.

6.5. Zliczanie punktów

Tabela 6.5 Scenariusz testowy liczenia punktów

ID	5
Tytuł	Zliczanie punktów.
Warunki początkowe	Uruchomiona gra przez 2-4 graczy, dwóch graczy posiada ten sam pseudonim, gra w trakcie.
Kroki testowe	1.Gracze rozpoczynają rozgrywkę. 2.Gracze liczą własne śmierci podczas rozgrywki.
Oczekiwany rezultat	Punkty zostają odpowiednio naliczane.
Wyniki	Punkty zgadzają się z liczbą śmierci graczy.

6.6.Opuszczenie gry przez gracza

Tabela 6.6 Scenariusz testowy opuszczenia gry

ID	6
Tytuł	Opuszczenie gry przez gracza.
Warunki początkowe	Uruchomiona gra przez 2-4 graczy, gra w trakcie.
Kroki testowe	1.Gracze rozpoczynają rozgrywkę. 2.Jeden z graczy opuszcza grę.
Oczekiwany rezultat	Postać gracza zostaje usunięta, gra toczy się dalej.
Wyniki	Jeśli graczem jest klient gra toczy się dalej, gracz który opuścił mecz widnieje jako <*null*> na tabeli wyników. Jeśli grę opuścił serwer gra zostaje całkowicie przerwana.

6.7. Włączenie kolejnej gry po rozegraniu poprzedniej

Tabela 6.7 Scenariusz testowy ponownej rozgrywki

ID	7
Tytuł	Włączenie kolejnej gry po rozegraniu poprzedniej
Warunki początkowe	Uruchomiona gra przez 2-4 graczy
Kroki testowe	1.Gracze rozpoczynają rozgrywkę. 2.Gracze kończą rozgrywkę wracając do głównego menu. 3.Gracze powtarzają poprzednie kroki.
Oczekiwany rezultat	Kolejne gry będą tworzone bez żadnych błędów lub danych z poprzednich gier.
Wyniki	Wpisy graczy w pokoju gry lub na tabeli wyników wyświetlają poprawne dane.

7. Podsumowanie i wnioski

W ramach pracy inżynierskiej napisana została gra, która pozwala na podstawie wybranej przez gracza muzyki wykonywać akcje do rytmu utworu. Gracze mogą rywalizować między sobą oraz tworzyć mapy i dzielić się nimi z innymi graczami.

Aplikacja spełnia wszystkie wymagania projektowe pomimo zmian w implementacji niektórych rozwiązań. Gracze mogą bezproblemowo rozpocząć grę w lokalnej sieci jak i pobrać od innego gracza mapę potrzebną na rozgrywkę. Menu aplikacji zostało zaprojektowane z myślą o prostocie użytkowania. Elementy takie jak tekstury mapy czy darmowa muzyka od twórców pozwoliła na łatwiejszą implementację projektu.

Gra została napisana przy użyciu silnika Godot oraz jego języka skryptowego *GDscript*. Pozwoliło to na łatwą implementację różnych modułów jak i prostsze testowanie aplikacji dzięki wbudowanym systemom i funkcją. Ciągle rozwijający się silnik Godot znacząco przyczynił się do rozbudowy gry.

System plików binarnych w grze pozwolił na łatwą obsługę i tworzenie map. Dzięki takiemu podejściu można w prosty sposób dalej rozbudowywać samo tworzenie plików nawet pod inną gre. Trzeba jednak zwrócić uwagę na bezpieczeństwo przy pracy z takimi plikami. Gra wymaga specyficznego pliku aby mapa została pobrana na urządzenie innego gracza jednak nadal powoduje to możliwość pobrania niechcianego pliku jeśli ktoś przygotuje go przedwcześnie. Dzięki temu że gra ogranicza się tylko do sieci lokalnej pozwala na mniejsze ryzyko ataku.

Dzięki systemowi węzłów i łatwej organizacji projektu gra może być dalej rozbudowana o kolejne usprawnienia. Do gry można wprowadzić większą ingerencję muzyki na samo otoczenie na przykład poprzez poruszanie częściami mapy. Wprowadzenie pełnoprawnej rozgrywki przez sieć byłaby dużym usprawnieniem dla wszystkich graczy. Pozwoliłoby to na dalszą rywalizację nie tylko z lokalnymi graczami. Ważnym elementem rozwoju aplikacji jest też dalsze testowanie nowych funkcji i naprawa napotkanych błędów.

Podsumowując, aplikacja spełnia wszystkie wymagania zawarte w założeniach projektowych. Implementacja projektu pozwoliła na poszerzenie wiedzy w zakresie tworzenia oraz testowania gier. Poznane narzędzia oraz metody programowania pomogą w dalszym tworzeniu aplikacji. Dzięki nabytemu doświadczeniu z pracą w silniku Godot kolejne projekty będą konstruowane szybciej oraz bardziej profesjonalnie.

Bibliografia

[1] Licencja silnika [Online]
https://docs.godotengine.org/en/stable/about/complying_with_licenses.html
[2] Strona silnika na platformie GitHub [Online]
https://github.com/godotengine/godot
[3] Dokumentacja Godot [Online]
https://docs.godotengine.org
[4] O języku Python [Online]
https://www.python.org/about/
[5] Strona formatu JSON [Online]
https://www.json.org
[6] Pixel Game Font Family [Online]
https://www.1001fonts.com/pixel-game-font.html
[7] Countdown for race, game start [Online]
https://freesound.org/people/Pastew/sounds/813525/
[8] Strona z muzyką artysty The Fat Rat [Online]
https://www.thefatrat.com/
[9] Legacy fantasy tilemap [Online]
https://anokolisa.itch.io/sidescroller-pixelart-sprites-asset-pack-forest-16x16
[10] Samurai 2D Pixel art [Online]
https://xzany.itch.io/samurai-2d-pixel-art

Spis rysunków

Rys. 4.1 Przykładowa budowa węzła grywalnej postaci

Rys. 5.1 Menu początkowe gry

Rys. 5.2 Otwarte okno opcji.

Rys. 5.3 Menu wyboru poziomu dla przycisku Play.

Rys. 5.4 Menu wyboru poziomu dla przycisku Edit

Rys. 5.5 Ekran tworzenia nowej mapy

Rys. 5.6 Pokój gracza

Rys. 5.7 Wycinek z trwającej rozgrywki

Rys. 5.8 Tabela wyników

Spis tabel

Tabela 3.1 Wykorzystane darmowe zasoby

Tabela 6.1 Scenariusz testowy działania przycisków w menu

Tabela 6.2 Scenariusz testowy dołączenia do pokoju gry

Tabela 6.3 Scenariusz testowy opóźnionego startu gry

Tabela 6.4 Scenariusz testowy interakcji między graczami

Tabela 6.5 Scenariusz testowy liczenia punktów

Tabela 6.6 Scenariusz testowy opuszczenia gry

Tabela 6.7 Scenariusz testowy ponownej rozgrywki

Spis listingów

Listing 3.1. przykładowa zawartość pliku .cfg.

Listing 4.1. Fragment kodu dodający rozpoznawanie pliku btbt.

Listing 4.2. Funkcja zmieniająca plik binarny w zasób w silniku.

Listing 4.3. Przypisanie pliku do zasobu.

Listing 4.4. Część skryptu tworzenia pliku.

Listing 4.5. Funkcje broadcast

Listing 4.6. Funkcje połączenia ENet

Listing 4.7. Funkcja skryptu metronom

Listing 4.8. Obliczanie aktualnego czasu utworu

Listing 4.9. Skrypt poruszania się gracza

Listing 4.10. Skrypt akcji gracza do rytmu