AKADEMIA
TARNOWSKA
Wydział Nauk Technicznych

Kierunek: ***Informatyka***

2025/2026

Marcel Nosek

PRACA INŻYNIERSKA

Projekt i implementacja gry z elementami przetrwania i rozwoju postaci

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2026*

Spis treści

Spis treści 3
1. Wstęp 5
1.1 Cel pracy 5
1.2 Zakres pracy 5
2. Analiza Biznesowa 7
2.1 Analiza rynku 7
2.2 Wymagania funkcjonalne 8
2.3 Wymagania niefunkcjonalne 9
3. Opis wykorzystanych technologii 11
3.1 Godot 11
3.2 GDScript 12
3.3 Tilemap 12
3.4 Pixilart 13
4. Implementacja 15
4.1 Świat 15
4.2 Mechaniki gracza 19
4.3 System przeciwników 27
5. Interfejs 37
5.1 Menu główne 37
5.2 Elementy interfejsu podczas rozgrywki 38
5.3 Dodatkowe informacje 39
6. Testowanie 41
6.1 Test głównego menu 41
6.2 Testy mechanik gracza 42
6.3 Testy systemu poziomów 44
6.4 Testy wybranych umiejętności 47
6.5 Testy systemu punktów 48
7. Podsumowanie i wnioski 51
Bibliografia 53
Spis rysunków 55
Spis tabel 57
Spis listingów 59

1. Wstęp

Gry komputerowe są jedną z najpopularniejszych form rozrywki współczesnego społeczeństwa i dynamicznie reagują na zmieniające się gusta odbiorców. W branży gier często można zaobserwować powstawanie trendów, w których sukces określonego typu produkcji prowadzi do tworzenia wielu gier inspirowanych podobnymi mechanikami i założeniami rozgrywki. W ostatnich latach szczególną popularnością cieszą się gry 2D wykorzystujące mechaniki przetrwania i progresji postaci, które oferują dynamiczną i angażującą rozgrywkę. W odpowiedzi na obserwowane trendy w branży gier komputerowych, niniejsza praca inżynierska skupia się na projekcie oraz implementacji gry 2D, której celem jest przetrwanie w losowo generowanym świecie poprzez eliminację nadciągających grup przeciwników oraz rozwój sterowanej przez gracza postaci.

1.1 Cel pracy

Celem pracy inżynierskiej jest zaprojektowanie oraz zaimplementowanie dynamicznej gry 2D z elementami przetrwania i rozwoju postaci. Gra umożliwia graczowi rywalizację z kolejnymi falami przeciwników w losowo generowanym świecie oraz zdobywanie punktów doświadczenia prowadzących do awansu na kolejne poziomy. Dodatkowym celem pracy jest opracowanie oraz implementacja czytelnych i spójnych mechanik rozgrywki, w tym systemu progresji postaci oraz zachowanie jednolitej i spójnej oprawy stylistycznej gry, obejmującej zarówno warstwę wizualną, jak i logiczną.

1.2 Zakres pracy

Zakres pracy obejmuje zarówno część projektową, jak i implementacyjną gry. W ramach realizacji projektu wykonane zostaną następujące elementy:

- zaprojektowanie mechaniki rozgrywki, w tym zasad przetrwania, systemu punktacji oraz progresji postaci,
- implementacja sterowania i zachowania postaci gracza,
- stworzenie zróżnicowanych typów przeciwników wraz z ich logiką działania,
- opracowanie generatora losowego terenu zapewniającego zmienność świata gry,
- przygotowanie spójnej oprawy graficznej gry 2D,
- testowanie poprawności działania oraz stabilności aplikacji.

2. Analiza Biznesowa

W niniejszym rozdziale przedstawiono wymagania funkcjonalne i niefunkcjonalne projektu, które mają na celu zapewnienie prawidłowego działania gry oraz utworzenie satysfakcjonującego doświadczenia dla gracza. W części dotyczącej analizy biznesowej opisano gatunek projektowanej gry oraz przedstawiono obserwacje dotyczące odbioru i popularności podobnych produkcji wśród graczy.

2.1 Analiza rynku

Projektowana gra należy do gatunku gier akcji typu „survivor-like”, będących połączeniem gry „roguelike” oraz automatycznej walki. Produkcje tego typu charakteryzują się prostym sterowaniem, dynamicznym rozwojem postaci i rosnącym poziomem trudności, co zachęca graczy do wielokrotnego podejmowania rozgrywki. Wśród podobnych produkcji można wymienić m.in.:

- ***Vampire Survivors*** - prekursor gatunku, oferujący prostą, lecz niezwykle wciągającą rozgrywkę polegającą na przetrwaniu kolejnych fal przeciwników,

- ***Holocure*** - darmowa gra fanowska oparta na tej samej mechanice, lecz z rozbudowanymi elementami RPG (Role - Playing Game) i estetyką anime,

- ***Survivor.io*** - gra powstała na tych samych regułach jednak przystosowana pod urządzenia mobilne.

W ostatnich latach gry inspirowane tytułem *Vampire Survivors* zyskały dużą popularność dzięki swojej przystępności oraz satysfakcjonującemu systemowi progresji. Gracze cenią sobie szybki rozwój postaci i chętnie grają w gry tego typu, dlatego takie tytuły bez problemu utrzymują się na rynku gier.

Tworzona gra wyróżnia się motywem szachowym oraz wykorzystuje sprawdzone mechaniki znane z innych popularnych produkcji, co powinno zachęcić graczy do przetestowania lubianego przez nich gatunku w nowej odsłonie.

2.2 Wymagania funkcjonalne

Wymagania funkcjonalne określają, jakie mechaniki i elementy rozgrywki powinny zostać zaimplementowane w grze aby była ona w pełni grywalna.

- **Sterowanie** - system sterowania powinien umożliwiać graczowi swobodne poruszanie się postacią w różnych kierunkach, wyznaczenie kierunku ataku oraz interakcję z elementami interfejsu użytkownika. System sterowania musi być intuicyjny i responsywny, aby zapewnić płynność rozgrywki.

- **Świat gry -** środowisko rozgrywki powinno być generowane proceduralnie, tak aby gracz nie był ograniczony w kierunku, w którym może się poruszać. Podczas generowania terenu powinny być tworzone różnorodne przeszkody i obiekty, aby urozmaicić rozgrywkę.

- **Przeciwnicy -** w trakcie gry będą pojawiały się fale przeciwników nadciągających z różnych stron mapy. Przeciwnicy powinni dążyć w stronę gracza, móc zadawać mu obrażenia oraz potrafić omijać przeszkody.

- **Poziom i umiejętności -** po pokonaniu przeciwników gracz powinien mieć możliwość zbierania punktów doświadczenia w postaci wypadających z nich kulek. Po zdobyciu odpowiedniej ilości doświadczenia gracz awansuje na kolejny poziom, a na ekranie pojawi się okno, w którym można wybrać nowe umiejętności lub polepszyć statystyki postaci.

- **Kolizje -** elementy świata powinny posiadać odpowiednio zdefiniowane strefy kolizji, aby gracz, przeciwnicy oraz używane przez nich umiejętności mogły prawidłowo reagować na kontakt z obiektami otoczenia. Kolizję pomiędzy graczem a przeciwnikami zostaną częściowo ograniczone, aby zachować większą swobodę ruchu, przy jednoczesnym zachowaniu wykrywania trafień i obrażeń.

- **Punktacja -** w grze powinien znaleźć się licznik punktów, który pozwoli graczowi śledzić jego postęp. Punkty będą przyznawane za różne działania, takie jak eliminacja

przeciwników, zbieranie przedmiotów czy interakcja z elementami świata. System ten pozwala graczowi śledzić jego skuteczność i zachęca do osiągnięcia lepszego wyniku.

2.3 Wymagania niefunkcjonalne

Wymagania niefunkcjonalne określają jak zapewnić graczowi wygodne i przyjemne doświadczenie podczas grania, skupiając się na aspektach technicznych oraz estetycznych.

- **Wydajność i kompatybilność -** gra powinna obsługiwać jedynie niezbędne elementy fizyki i kolizji, aby zachować płynność rozgrywki. Nawet Przy dużej liczbie przeciwników na ekranie gra powinna utrzymać stabilność i działać bezproblemowo na komputerach wyposażonych w system Windows 11.

- **Responsywność -** gra powinna natychmiast reagować na działania gracza, takie jak wciskanie klawiszy na klawiaturze czy kliknięcia myszką. Efekty tych akcji powinny być od razu widoczne na ekranie.

- **Stabilność -** aplikacja nie powinna zawierać żadnych błędów uniemożliwiających lub przerywających rozgrywkę graczowi. Wszelkie błędy powinny być odpowiednio obsłużone, aby zapobiec frustrującym sytuacjom podczas rozgrywki.

- **Grafika -** oprawa graficzna powinna być spójna i czytelna. Wszystkie elementy interfejsu i otoczenia powinny być dobrze widoczne i łatwe do rozróżnienia, aby gracz nie miał problemu z śledzeniem aktualnego stanu gry, nawet przy dużej liczbie różnych elementów pojawiających się na ekranie.[3]

3. Opis wykorzystanych technologii

Rozdział ten przedstawia technologie oraz narzędzia wykorzystane przy tworzeniu gry. Wybór odpowiednich narzędzi jest kluczowy, aby realizacja projektu przebiegała sprawnie oraz aby osiągnąć zamierzone efekty wizualne i funkcjonalne.

3.1 Godot

Godot Engine to otwartoźródłowy silnik do tworzenia gier 2D i 3D. Jest rozwijany na licencji MIT (*Massachusetts Institute of Technology*), co pozwala każdemu użytkownikowi na bezpłatne korzystanie z jego możliwości. Licencja ta jest niezwykle przyjazna dla twórców, ponieważ nie wymaga żadnych opłat licencyjnych, a jedynym wymogiem przy jej używaniu jest zachowanie informacji o prawach autorskich i wykorzystanej licencji w finalnym produkcie.

Silnik Godot jest prosty w obsłudze i oferuje szeroki zestaw funkcjonalności, które znacząco zwiększają efektywność pracy przy tworzeniu nowych gier i aplikacji. Jest przystosowany do wielu środowisk dzięki czemu można tworzyć gry zarówno na komputery jak i urządzenia mobilne czy przeglądarki internetowe. Dzięki wbudowanemu edytorowi scen, animatorowi, systemowi fizyki, narzędziom do tworzenia interfejsów użytkownika (UI) oraz obsłudze skryptów, twórca może bezproblemowo tworzyć i testować projekty w obrębie tylko jednego środowiska.

Godot umożliwia eksportowanie gier na wiele platform, w tym komputery osobiste (Windows, Linux, macOS), urządzenia mobilne (Android, iOS) oraz przeglądarki internetowe (HTML5/WebAssembly).

Silnik obsługuje kilka języków programowania, w tym GDScript, C# oraz C++, dzięki czemu użytkownik może wybrać technologię najlepiej dopasowaną do swoich potrzeb. [2][5]

3.2 GDScript

GDScript to imperatywno-obiektowy język programowania powstały specjalnie na potrzeby silnika Godot. Składnia języka była inspirowana innym popularnym językiem - Pythonem - dzięki czemu jest zwięzła, czytelna i łatwa do nauki.

Podobnie jak Python, GDScript kładzie nacisk na prostotę i przejrzystość kodu, co przekłada się na szybsze pisanie skryptów i ogranicza ilość błędów. Dodatkowo wbudowany edytor Godota pozwala na natychmiastową edycję i kompilację kodu co ułatwia testowanie aplikacji nawet przy wprowadzaniu najmniejszych zmian.

Język ten jest ściśle zintegrowany z architekturą silnika, co oznacza, że ma bezpośredni dostęp do wszystkich jego funkcji takich jak obsługa fizyki, animacji, kolizji, dźwięku, interfejsu użytkownika czy zarządzania scenami. Dzięki temu tworzenie logiki, jak przykładowo sterowanie postacią gry, jest wyjątkowo proste i wymaga niewielkiej ilości kodu. [1][6]

3.3 Tilemap

Tilemapa w silniku Godot umożliwia szybkie i uporządkowane tworzenie poziomów poprzez rozmieszczanie kafelków na siatce. Dzięki zdefiniowanym zestawom kafelków projektant może w prosty sposób budować rozległe obszary gry, zachowując spójność wizualną oraz logiczną struktury mapy. Rozwiązanie to znacznie przyspiesza proces tworzenia świata i minimalizuje potrzebę ręcznego ustawiania powtarzalnych elementów.

Każdy kafelek może posiadać własne właściwości, takie jak kolizje czy parametry nawigacyjne, co ułatwia tworzenie interakcji z otoczeniem. Silnik automatycznie uwzględnia kolizje postaci z obiektami świata, a system nawigacji może wykorzystywać dane zapisane w Tilemapie do generowania ścieżek dla przeciwników i innych obiektów. Pozwala to na tworzenie złożonych zachowań w środowisku gry bez konieczności dodawania osobnych komponentów kolizyjnych czy nawigacyjnych.

Dodatkowo Tilemapa jest wydajna pod względem renderowania oraz dobrze wspiera generowanie proceduralne. Dzięki możliwości dynamicznego modyfikowania kafelków, świat gry może być rozbudowywany na bieżąco, co jest szczególnie istotne w projektach wykorzystujących podział mapy na segmenty. Rozwiązanie to umożliwia tworzenie dużych, nieograniczonych przestrzeni przy zachowaniu wysokiej wydajności.

3.4 Pixilart

Pixilart to darmowa, przeglądarkowa aplikacja internetowa służąca do tworzenia grafik w stylu pixel art. Serwis ten oferuje prosty, ale rozbudowany edytor, który umożliwia tworzenie i edycję grafik pikselowych bez potrzeby instalowania dodatkowego oprogramowania. Dzięki temu jest dobrym wyborem do tworzenia prostych elementów graficznych do gier, takich jak postacie, tła, ikony, czy klatki animacji.

Pixilart udostępnia zestaw podstawowych narzędzi do rysowania (m.in. pędzle, gumkę, wypełnianie kolorem, warstwy oraz funkcję podglądu animacji). Pozwala również na korzystanie z systemu warstw, który jest powszechnie używany w wielu edytorach graficznych, aby zapewnić uporządkowany sposób pracy.

Kolejnymi zaletami Pixilart jest możliwość zapisu swojej pracy w formacie PNG (*Portable Network Graphics*) lub w natywnym formacie .pixil. A dzięki wbudowanej możliwości importu istniejących grafik w formacie PNG lub PIXIL, użytkownik może bez problemu kontynuować swoją pracę nad wcześniej zapisanymi projektami. Dodatkowo strona zapisuje ostatnie projekty w chmurze (dla zalogowanych użytkowników), co pozwala uniknąć utraty danych. [7]

4. Implementacja

Rozdział ten opisuje proces implementacji poszczególnych elementów gry. Przedstawiono sposób działania najważniejszych mechanik, takich jak generacja świata, zachowanie przeciwników, mechaniki gracza, przedmioty oraz system umiejętności.

4.1 Świat

Świat gry pełni kluczową rolę w rozgrywce, stanowiąc przestrzeń, po której porusza się gracz oraz w której generowane są wszystkie elementy otoczenia (rys. 4.1). W projekcie zastosowano dynamiczne budowanie świata, dzięki czemu jego struktura może rozwijać się w miarę postępu gracza, bez konieczności tworzenia pełnej mapy na początku gry. Rozwiązanie to pozwala na optymalizację wydajności oraz na tworzenie potencjalnie nieograniczonej przestrzeni rozgrywki. [4]

Rys. 4.1. Widok świata gry

4.1.1 Podział świata na segmenty

W celu umożliwienia płynnego działania gry oraz optymalnego zarządzania zasobami zastosowano podział świata na mniejsze segmenty przestrzeni o wymiarach 16×16 kafelków. Dzięki wykorzystaniu Tilemapy podział ten jest naturalny i pozwala na efektywne tworzenie oraz aktualizowanie świata w miarę przemieszczania się gracza.

Każdy segment jest generowany dopiero w momencie, gdy znajdzie się w określonym promieniu od aktualnej pozycji postaci. Pozwala to na dynamiczne, proceduralne rozszerzanie świata gry bez konieczności wcześniejszego tworzenia całej mapy. W efekcie gracz może poruszać się w dowolnym kierunku, a kolejne obszary są dodawane jedynie wtedy, gdy są rzeczywiście potrzebne.

Implementację mechanizmu odpowiedzialnego za wykrywanie aktualnego segmentu gracza oraz inicjalizację nowych fragmentów świata przedstawiono na listingu 4.1.

Listing 4.1. Kod odpowiedzialny za inicjowanie generacji nowych segmentów świata

```
func _process(_delta):
	var player_chunk = Vector2(
		int(player.position.x / chunk_size),
		int(player.position.y / chunk_size)
	)

	for x in range(player_chunk.x - 2, player_chunk.x + 3):
		for y in range(player_chunk.y - 2, player_chunk.y + 3):
			var coord = Vector2(x, y)
			if not loaded_chunks.has(coord):
				generate_chunk(coord)
```

4.1.2 Tworzenie segmentu

Każdy nowy segment po wygenerowaniu jest wypełniany podstawowym podłożem przedstawionym w formie wzoru przypominającego szachownicę. Następnie dany fragment przestrzeni jest analizowany pod kątem możliwości umieszczenia dodatkowych elementów otoczenia, takich jak przeszkody czy dekoracyjne obiekty (rys. 4.2 i 4.3). Aby zapobiec sytuacji w których elementy będą nachodziły na siebie, segment jest podzielony na cztery mniejsze strefy, z których każda ma określone prawdopodobieństwo wygenerowania obiektu.

Rys. 4.2. Figura konia

Rys. 4.3. Figura wieży

Zmiany te są nanoszone bezpośrednio na odpowiednie Tilemapy, co umożliwia natychmiastową aktualizację wyglądu oraz struktury świata gry. Proces generowania segmentu przedstawiono na listingu 4.2.

Listing 4.2. Kod odpowiedzialny za generacje nowego segmentu świata

```
func generate_chunk(coord: Vector2):
	var start_pos = coord * chunk_size_in_tiles
	for x in range(chunk_size_in_tiles):
		for y in range(chunk_size_in_tiles):
			if (x + y) % 2 == 0:
				tilemap_floor.set_cell(start_pos
								 + Vector2(x, y),
								 source_id, light)
			else:
				tilemap_floor.set_cell(start_pos
								 + Vector2(x, y),
								 source_id, dark)

	for chunk_fragment in range(chunk_fragments):
		var chunk_fragment_start_pos = start_pos + Vector2(
					(chunk_fragment % 2)
					* chunk_fragment_size,
					int(chunk_fragment/2)
					* chunk_fragment_size)
		var obs = obstacles_cords[randi_range(
					0, obstacles_cords.size() - 1)]
		var size_in_tiles = get_tile_size_in_tiles(
					tilemap_obstacles, source_id, obs)

		var obs_x = randi_range(1, chunk_size_in_tiles
					/sqrt(chunk_fragments)
					- size_in_tiles.x - 1)
		var obs_y = randi_range(1, chunk_size_in_tiles
					/sqrt(chunk_fragments)
					- size_in_tiles.y - 1)
		var start_tile = chunk_fragment_start_pos + Vector2(
					obs_x, obs_y)

		tilemap_obstacles.set_cell(start_tile, source_id, obs)

	loaded_chunks[coord] = true
```

Funkcja *generate_chunk()* widoczna na listingu 4.2 generuje nowy segment świata na podstawie współrzędnej *coord*. Najpierw obliczana jest pozycja początkowa segmentu, a następnie podwójna pętla wypełnia go kafelkami podłoża. Wzór szachownicy powstaje dzięki sprawdzaniu parzystości sumy współrzędnych (*(x + y) % 2*). Do umieszczania kafelków wykorzystywana jest metoda *set_cell()*, która wstawia wybrany kafelek do Tilemapy na podanej pozycji (tu można dopisać szczegóły o atlasie i identyfikatorach kafelków).

Po utworzeniu podłoża segment zostaje podzielony na mniejsze fragmenty, w których losowane i rozmieszczane są przeszkody. Rozmiar przeszkody określa funkcja *get_tile_size_in_tiles()*. Następnie wybierana jest losowa pozycja mieszcząca się w obrębie fragmentu i ustawiana w Tilemapie przeszkód za pomocą *set_cell()*.

Na końcu segment zostaje oznaczony jako załadowany poprzez wpisanie go do słownika *loaded_chunks*.

4.2 Mechaniki gracza

W tym podrozdziale opisano podstawowe funkcje gracza, takie jak poruszanie się, atakowanie, otrzymywanie obrażeń oraz zdobywanie doświadczenia. Omówiono również system kolizji, zbierania przedmiotów i działania umiejętności.

4.2.1 Poruszanie się gracza

Podstawowym elementem rozgrywki jest swobodne poruszanie się postaci po mapie. Gracz kontroluje ruch przy użyciu klawiszy kierunkowych, a ruch odbywa się z wykorzystaniem funkcji fizycznych silnika Godot. Kod odpowiedzialny za poruszanie przedstawiono na listingu 4.3.

Listing 4.3. Implementacja ruchu gracza.

```
func _physics_process(delta: float):
	movement_direction = Vector2(Input.get_axis("ui_left", "ui_right"),
	                             Input.get_axis("ui_up", "ui_down")).normalized()

	if movement_direction.length() > 0:
		move(delta)
		player_animation.play()
	else:
		player_animation.stop()
		player_animation.frame = 0

func move(_delta: float):
	if movement_direction != Vector2.ZERO:
		velocity = movement_direction * SPEED
	else:
		velocity = Vector2.ZERO

	move_and_slide()
```

W kodzie przedstawionym na listingu 4.3 wykorzystano funkcję *_physics_process*, która jest wywoływana co klatkę. Wartość *movement_direction* określa kierunek ruchu na podstawie aktualnie wciśniętych klawiszy, a po jej normalizacji wektor ma zawsze długość równą 1, co zapewnia jednakową prędkość ruchu we wszystkich kierunkach.

Funkcja *move(_delta)* odpowiada za nadanie postaci wektora prędkości na podstawie wyznaczonego kierunku ruchu oraz stałej *SPEED*, która określa szybkość poruszania się gracza. Gdy gracz nie wykonuje żadnego ruchu, prędkość zostaje wyzerowana, co zatrzymuje postać w miejscu.

Z kolei funkcja *move_and_slide()* zapewnia płynne poruszanie się obiektu po powierzchniach z uwzględnieniem fizyki oraz kolizji, dzięki czemu postać nie przenika przez inne elementy świata gry.

4.2.2 Atak i zadawanie obrażeń

Mechanika ataku opiera się na odtwarzaniu animacji oraz wykrywaniu obiektów znajdujących się w zasięgu postaci. Kod przedstawiono na listingu 4.4.

Listing 4.4. Implementacja ataku gracza.

```
func _physics_process(_delta: float):
	move_area()

func move_area():
	var mouse_pos = get_global_mouse_position()
	look_at(mouse_pos)

func _on_attack_timer_timeout():
	attack_animation.play()
	attack_timer.wait_time = 2.0 / attack_speed
	for body in get_overlapping_bodies():
		body.get_hit(attack_damage)
		body.get_pushed(stun_strength, stun_duration)
```

Atak gracza jest wykonywany w sposób cykliczny - w równych odstępach czasu, które określa zmienna *attack_timer.* Po każdym upływie wyznaczonego czasu wywoływana jest funkcja *_on_attack_timer_timeout()*

W momencie uruchomienia ataku odtwarzana jest animacja (*attack_animation.play()*), a następnie następuje sprawdzenie wszystkich obiektów znajdujących się w zasięgu gracza. Funkcja *get_overlapping_bodies()* zwraca listę obiektów kolidujących z obszarem ataku. Dla każdego z nich wywoływane są dwie funkcje:

- *get_hit(attack_damage)* - odpowiedzialna za zadanie obrażeń, oraz
- *get_pushed(stun_strength, stun_duration)* - nadająca efekt odrzutu i chwilowego ogłuszenia przeciwnika.

4.2.3 Przyjmowanie obrażeń

Kiedy przeciwnik atakuje gracza, wywoływana jest funkcja odpowiedzialna za zmniejszanie punktów życia Jej implementację przedstawiono na listingu 4.5.

Listing 4.5. Funkcja odpowiedzialna za otrzymywanie obrażeń.

```
func get_hit(damage):
	HP -= damage
	emit_signal("player_hp_change",HP)
	if HP <= 0:
		emit_signal("player_die")
```

Funkcja *get_hit()* odejmuje wartość obrażeń od punktów życia gracza, a po ich wyczerpaniu wysyła sygnał o śmierci postaci.

4.2.4 Podnoszenie przedmiotów

Gracz posiada niewidoczny obszar kolizji, który służy do wykrywania i podnoszenia znajdujących się w pobliżu przedmiotów. Implementację przedstawiono na listingu 4.6.

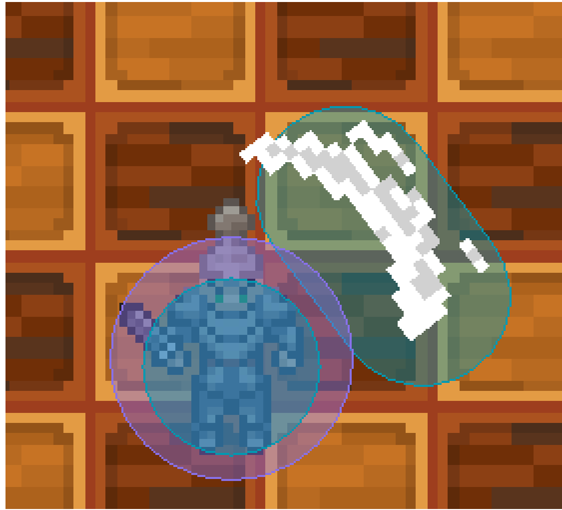
Listing 4.6. Skrypt odpowiedzialny za podnoszenie przedmiotów.

```
extends Area2D

@export_category("Collect collider params")
@export var pickup_range: float = 10

signal get_exp(exp:int)
signal get_ultimate_pawn()

func _on_area_entered(area: Area2D):
	if area.is_in_group("Exp"):
		emit_signal("get_exp",area.exp_value)
		area.queue_free()
	if area.is_in_group("PawnUltimate"):
		emit_signal("get_ultimate_pawn")
		area.queue_free()
```

Po wejściu w obszar przedmiotu skrypt rozpoznaje jego typ na podstawie przypisanej grupy (*Exp* lub *PawnUltimate*) i emituje odpowiedni sygnał.

4.2.5 Kolizja

Gracz posiada kilka obiektów kolizyjnych (rys. 4.4), z których każdy pełni określoną funkcję w zależności od rodzaju interakcji. Podstawowy obiekt odpowiada za wykrywanie kontaktu z elementami otoczenia, co zapobiega przenikaniu postaci przez przeszkody i umożliwia prawidłowe poruszanie się po mapie.



Rys. 4.4. Widoczne pola kolizji gracza

Dodatkowy obiekt kolizyjny służy do wykrywania obecności przeciwników, dzięki czemu mogą oni reagować na pozycję gracza oraz zadawać mu obrażenia. Osobna strefa kolizji została również wykorzystana do podnoszenia przedmiotów - po wejściu w jej zasięg wywoływane są odpowiednie funkcje umożliwiające zebranie obiektu.

Ostatni z elementów kolizyjnych odpowiada za wykrywanie trafień podczas ataku, co pozwala określić, które jednostki znalazły się w jego zasięgu. Wszystkie te obiekty przypisane są do odpowiednich warstw kolizji i posiadają zdefiniowane maski kolizji, precyzyjnie określające, z jakimi elementami świata gry mogą wchodzić w interakcję.

4.2.6 Umiejętności

Gracz może rozwijać swoje statystyki i zwiększać efektywność ataków poprzez nabywanie umiejętności (rys. 4.5). Przykładowe funkcje odpowiadające za wzmocnienie ataku przedstawiono na listingu 4.7.

Rys. 4.5. Widok wyboru nowych umiejętności

Listing 4.7. Przykładowe funkcje wzmacniające atak gracza.

```
func attack_up(value:float):
	attack_zone.attack_damage += value

func attack_speed_up(value:float):
	attack_zone.attack_speed += value
```

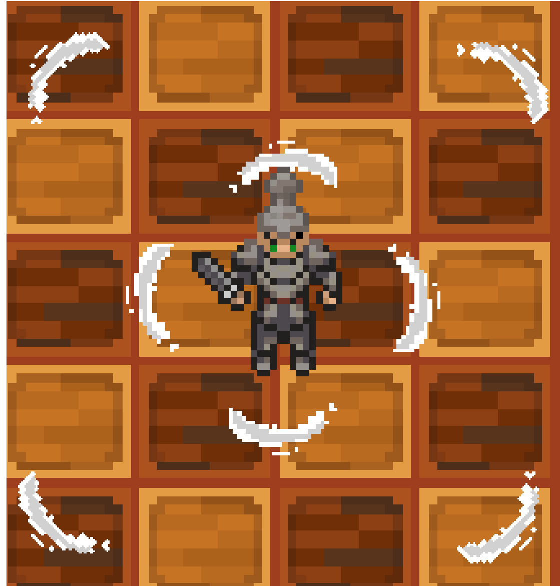
Każda umiejętność modyfikuje określone parametry gracza, takie jak obrażenia czy szybkość ataku.

Gracz dysponuje również umiejętnością specjalną, która uruchamia serię fal pocisków rozchodzących się w różnych kierunkach. Mechanizm ten przedstawiono na listingu 4.8.

Listing 4.8. Implementacja umiejętności specjalnej.

```
func ultimate() -> void:
	var dirs_straight = [Vector2.RIGHT, Vector2.LEFT,
	                     Vector2.UP, Vector2.DOWN]

	var dirs_diagonal = [
		Vector2(1, 1),
		Vector2(-1, 1),
		Vector2(1, -1),
		Vector2(-1, -1)
	]

	spawn_waves(dirs_straight, dirs_diagonal)

func spawn_projectile(dir: Vector2):
	var p = ultimate_projectile_prefab.instantiate()

get_tree().get_current_scene().get_node("PowersContainer").add_child(p)
	p.position = self.global_position
	p.init(dir)

func spawn_waves(dirs_straight: Array, dirs_diagonal: Array):
	for i in range(ultimate_waves):
		if i % 2 == 0:
			# Fala prosta (pion/poziom)
			for j in range(4):
				spawn_projectile(dirs_straight[j %
				             dirs_straight.size()])
		else:
			# Fala ukośna
			for j in range(4):
				spawn_projectile(dirs_diagonal[j %
				             dirs_diagonal.size()])



		$PowersContainer/UltimateWaveTimer.start()
		await $PowersContainer/UltimateWaveTimer.timeout
```

Umiejętność specjalna (rys. 4.6) stanowi potężny atak obszarowy aktywowany po naciśnięciu odpowiedniego klawisza. Aby jej użyć, gracz musi wcześniej zebrać cztery pionki - przedmioty symbolizujące energię potrzebną do uruchomienia zdolności.



Rys. 4.6. Działanie umiejętności specjalnej

Po aktywacji umiejętności wywoływana jest funkcja *ultimate()*, która przygotowuje dwa zestawy kierunków - pionowe i poziome (*dirs_straight*) oraz ukośne (*dirs_diagonal*). Następnie uruchamiana jest funkcja *spawn_waves*(), odpowiedzialna za generowanie kolejnych fal pocisków.

Każda fala składa się z serii pocisków tworzonych za pomocą funkcji *spawn_projectile()*, która tworzy nowy obiekt, ustawia jego pozycję w miejscu gracza oraz nadaje mu kierunek ruchu. Fale są generowane naprzemiennie - raz w kierunkach prostych, a raz po przekątnych, co tworzy efekt dynamicznego, rozchodzącego się ataku.

Pomiędzy kolejnymi falami zastosowano krótki odstęp czasowy przy użyciu timera *UltimateWaveTimer*, co sprawia, że atak przebiega płynnie i rytmicznie, zamiast być natychmiastowy. Umiejętność ta pozwala graczowi na zadanie dużych obrażeń wielu przeciwnikom jednocześnie, stanowiąc ważny element strategiczny rozgrywki.

4.3 System przeciwników

System przeciwników w grze pełni kluczową rolę w budowaniu dynamiki rozgrywki. Odpowiada on za sterowanie zachowaniem wrogów, ich poruszanie się, interakcje z graczem oraz generowanie kolejnych fal o rosnącym poziomie trudności. W niniejszym podrozdziale przedstawiono szczegółowy opis mechanizmów sterujących przeciwnikami.

4.3.1 Poruszanie się i omijanie przeszkód

Przeciwnicy w grze nieustannie dążą w kierunku gracza (rys. 4.7), co tworzy charakterystyczną dla gatunku presję i wymusza ciągły ruch. W celu realizacji tego zachowania każdy przeciwnik wykorzystuje promień kolizji (rys. 4.8), pozwalający mu wykrywać przeszkody znajdujące się na drodze do celu.

Rys. 4.7. Przeciwnicy goniący gracza

Rys. 4.8. Przeciwnik z widocznymi polami i wektorami kolizji

W przypadku wykrycia kolizji przeciwnik podejmuje próbę obejścia napotkanego obiektu poprzez czasową zmianę orientacji promienia oraz kąta poruszania się. Ruch przeciwników oraz logika omijania przeszkód jest przedstawiona na listingu 4.9

Listing 4.9. Funkcja odpowiedzialna za ruch przeciwnika i wykrywanie kolizji.

```
func move_toward_player(delta: float):
	var dir_to_player = (player.global_position -
	                     global_position).normalized()
	var angle_to_player = dir_to_player.angle()

	if enemy_collision_ray.is_colliding():
		is_colliiding = true
		$RaysBox/RayFocusTimer.start()
		rotation_speed = 15.0
		var rotate_dir = int(enemy_focus_ray.rotation_degrees) % 90
		if rotate_dir < 0:
			rotate_dir += 90
		if rotate_dir < 45:
			enemy_collision_ray.rotation += rotation_speed * delta
		else:
			enemy_collision_ray.rotation += -rotation_speed * delta

	else:
		rotation_speed = 3.0
		enemy_collision_ray.rotation = lerp_angle(
			enemy_collision_ray.rotation,
			angle_to_player,
			rotation_speed * delta
		)

	if !is_colliiding:
		enemy_focus_ray.look_at(player.position)
	var forward = enemy_collision_ray.global_transform.x.normalized()
	position += forward * speed * delta
```

W przedstawionym fragmencie kluczową rolę pełni promień kolizji (*enemy_collision_ray*), którego zadaniem jest wykrywanie przeszkód znajdujących się

przed przeciwnikiem. Przeciwnik porusza się zawsze w kierunku wyznaczanym przez osobny promień – *focus_ray*.

Gdy promień kolizji wykryje przeszkodę aktywuje się mechanizm który polega na stopniowym obrocie promienia *focus_ray* w lewo lub w prawo do momentu, gdy promień kolizji przestanie zgłaszać kolizję, co tworzy prosty, ale skuteczny system unikania obiektów.

W sytuacji, gdy nie wykryto kolizji, promień ruchu zostaje płynnie dopasowany do kąta prowadzącego w stronę gracza dzięki interpolacji funkcją *lerp_angle*, co zapewnia naturalne, niegwałtowne obroty przeciwnika.

4.3.2 Atakowanie gracza oraz reakcje na obrażenia

Przeciwnicy są w stanie wchodzić w interakcję z graczem poprzez zadawanie obrażeń po wejściu w określony zasięg ataku. Dodatkowo reagują na ciosy i umiejętności gracza, tracąc punkty zdrowia i mogąc doznać efektów kontroli tłumu, takich jak odrzucenie czy ogłuszenie.

Listing 4.10. Funkcja przeciwnika odpowiedzialna za otrzymywanie obrażeń.

```
func get_hit(damage):
	hp -= damage
	if hp <= 0:
		enemy_die()
		print_stack()
		call_deferred("queue_free")
```

Funkcja przedstawiona na listingu 4.10 odpowiada za obsługę zadawania obrażeń przeciwnikowi. Każde trafienie zmniejsza wartość punktów zdrowia (*hp*) o wartość przekazanego parametru damage. W momencie osiągnięcia lub przekroczenia wartości zero przeciwnik zostaje uznany za pokonanego.

W zależności od rodzaju ataku gracza przeciwnik może zostać odrzucony na pewną odległość oraz chwilowo unieruchomiony. W zależności od źródła otrzymanych obrażeń efekt kontroli tłumu może być silniejszy lub mieć większy czas odrzucenia.

Listing 4.11. Implementacja efektu odrzucenia i ogłuszenia przeciwnika.

```
func get_pushed(strength, push_time = 0.1, stun_time = 0):
	var push_direction = (position - player.global_position)
	push_direction = push_direction.normalized()
	knockback_velocity = push_direction * strength
	knockback_timer = push_time
	stun_timer = stun_time
	is_knocked_back = true
	is_stunned = true
```

Listing 4.11 przedstawia funkcję odpowiedzialną za zastosowanie efektów kontroli tłumu na przeciwniku. Wyznaczany jest wektor *push_direction*, który wskazuje kierunek przeciwny do pozycji gracza, a następnie mnożony przez parametr *strength*, co określa siłę odrzucenia.

Zmienna *knockback_timer* określa czas trwania przesunięcia, a *stun_timer* opcjonalnie dodaje efekt ogłuszenia, który chwilowo uniemożliwia przeciwnikowi podejmowanie jakichkolwiek działań. Ustawienie flag *is_knocked_back* oraz *is_stunned* aktywuje odpowiednie sekcje logiki w głównej pętli przeciwnika. Dzięki temu gracz ma możliwość stosowania zróżnicowanych ataków, które nie tylko zadają obrażenia, ale również kontrolują pozycję i zachowanie wrogów na polu walki.

Listing 4.12. Wyzwalanie ataku przeciwnika po wejściu w jego zasięg.

```
func _on_body_entered(body):
	if target == null:
		target = body
		target.get_hit(10)
		$Timer.start()
```

W kodzie przedstawionym na listingu 4.12 przeciwnik wykorzystuje obszar kolizji jako strefę ataku. Gdy obiekt (np. gracz) wejdzie do strefy, funkcja *_on_body_entered* zostaje wywołana automatycznie. Jeśli przeciwnik nie ma aktualnie wyznaczonego celu, przypisuje go do zmiennej target, po czym natychmiast zadaje mu obrażenia poprzez wywołanie *get_hit*(). Następnie uruchamiany jest timer, który

stanowi mechanizm ograniczający częstość ataków, dzięki czemu przeciwnik nie może zadawać obrażeń w każdej klatce i musi odczekać określony czas przed kolejnym uderzeniem.

4.3.3 Generator przeciwników

Wrogowie nie pojawiają się jednocześnie – gra wykorzystuje generator fal, który odpowiada za stopniowe zwiększanie trudności rozgrywki. Każda fala posiada określoną wartość punktową która jest „wydawana” na tworzenie przeciwników w zależności od ich kosztu.

Przy starcie poziomu generator pobiera wszystkie typy przeciwników i wybiera te, które mogą pojawić się od pierwszej fali.

Listing 4.13. Inicjalizacja systemu fal i przygotowanie pierwszej fali przeciwników.

```
func _ready():
	wave_value = wave_starting_value
	wave_count = 1
	for i in enemy_scenes.size():
		var enemy = enemy_scenes[i].instantiate()
		enemies_dict[i] = {
			"value": enemy.value,
			"enemy": enemy,
			"wave_min": enemy.wave_min,
			"wave_max": enemy.wave_max
		}
		if enemies_dict[i]["wave_min"] <= 1:
			enemies_in_use[i] = enemies_dict[i]
```

W listingu 4.13 przedstawiono proces przygotowania generatora przeciwników. Na początku ustalana jest początkowa wartość fali (*wave_value*) oraz początkowa numeracja. Następnie wszystkie dostępne typy przeciwników są wczytywane

i zapisywane w strukturze *enemies_dict*, zawierającej informacje o ich wartości punktowej oraz zakresach fal, w których mogą występować.

Przeciwnicy, którzy mogą pojawić się już od pierwszej fali (*wartość wave_min ≤ 1*), są zapisywani w strukturze *enemies_in_use*, która stanowi aktualną pulę możliwych przeciwników dla bieżącej fali.

Timer systematycznie wywołuje kolejne instancje przeciwników, aż do wyczerpania wartości fali. Po zakończeniu generowania odczekuje określony czas, zwiększa numer fali i aktualizuje listę dostępnych przeciwników. Jego działanie zostało przedstawione na listingu 4.14.

Listing 4.14. Obsługa generowania kolejnych przeciwników oraz zmiany fal

```
func _on_timer_timeout():
	if wave_value > 0:
		var enemy_value = spawn_enemy()
		wave_value -= enemy_value
		$Timer.start(0.2)
	else:
		wave_count += 1
		$Timer.start(30)
		wave_value = wave_starting_value * pow(wave_scale,
		                                       wave_count)
		enemies_in_use.clear()
		for i in enemy_scenes.size():
			if (enemies_dict[i]["wave_min"] <= wave_count
			and enemies_dict[i]["wave_max"] >= wave_count):
				enemies_in_use[i] = enemies_dict[i]
```

W listingu 4.14 generator fal kontroluje proces pojawiania się przeciwników. Jeśli bieżąca fala nie została jeszcze „wykorzystana” (czyli jej wartość wave_value jest większa od zera), przeciwnik jest tworzony poprzez funkcję spawn_enemy(), a jego

koszt odejmowany od pozostałej wartości fali. Timer ustawiony na 0.2 sekundy zapewnia równomierne, stopniowe pojawianie się wrogów.

Po wyczerpaniu wartości fali generator rozpoczyna przygotowania do kolejnej fali: zwiększa jej numer, ponownie oblicza wartość punktową oraz aktualizuje listę dostępnych przeciwników zgodnie z parametrami wave_min oraz wave_max. Dzięki temu mocniejsi przeciwnicy zaczynają pojawiać się dopiero na odpowiednich etapach rozgrywki.

Przeciwnik jest losowany z puli dostępnej dla danej fali, a następnie umieszczany w jednym z losowych punktów strefy respawnu – obszaru znajdującego się poza widocznym zakresem kamery, co zapobiega zauważeniu momentu pojawienia się wroga przez gracza.

Listing 4.15. Losowe tworzenie instancji przeciwników w wyznaczonym obszarze.

```
func spawn_enemy():
	if enemies_in_use.size() == 0:
		return 0

	var keys = enemies_in_use.keys()
	var random_key = keys[rng.randi_range(0, keys.size() - 1)]

	var enemy_scene = enemy_scenes[random_key]
	var enemy = enemy_scene.instantiate()

	enemy.global_position = get_random_point_in_zone()
	spawn_node.add_child(enemy)
```

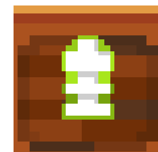
Listing 4.15 prezentuje funkcję odpowiedzialną za losowe tworzenie przeciwników. Najpierw sprawdzana jest wielkość puli dostępnych wrogów - jeśli jest pusta, funkcja nic nie tworzy. Następnie losowany jest klucz odpowiadający jednemu z dostępnych typów przeciwników, po czym jego model jest instancjonowany.

Nowo utworzony przeciwnik otrzymuje losową pozycję w specjalnie wyznaczonej strefie, która znajduje się poza obszarem widocznym dla gracza. Zapobiega to zauważeniu procesu pojawiania się wrogów i zwiększa immersję rozgrywki. Ostatecznie przeciwnik zostaje dodany do odpowiedniego węzła (*spawn_node*) w scenie gry.

4.3.4 System łupów i punktów

W grze zaimplementowano system łupów, który aktywuje się w momencie wyeliminowania przeciwnika. Po śmierci wroga generowany jest jeden z dwóch typów przedmiotów: kula doświadczenia (rys. 4.9) lub specjalny pionek (rys. 4.10). Kula doświadczenia pozwala zwiększać poziom gracza, natomiast pionek służy do ładowania umiejętności specjalnej.

Rys. 4.9. Kula doświadczenia



Rys. 4.10. Specjalny pionek

Listing 4.16. Generowanie łupu po śmierci przeciwnika

```
func enemy_die():
	var kill_reward = null
	if randf() < 0.1:
		kill_reward = pawn_ultimate_scene.instantiate()
	else:
		kill_reward = exp_scene.instantiate()
	if kill_reward:
		kill_reward.position = self.position
		get_tree().get_current_scene().get_node(
			"CollectablesContainer").add_child(kill_reward)
```

W zaprezentowanej na listingu 4.16 funkcji *enemy_die()* losowany jest rodzaj łupu, który pojawi się na pozycji pokonanego przeciwnika. Mechanizm opiera się na

funkcji *randf()*, zwracającej liczbę z przedziału [0, 1). Jeśli wylosowana wartość jest mniejsza niż 0.1, generowany jest pionek umiejętności specjalnej, co oznacza że prawdopodobieństwo jego zdobycia wynosi 10%. W przeciwnym razie gracz otrzymuje kulę doświadczenia.

Następnie wybrany obiekt tworzony jest na scenie i dodawany do kontenera *CollectablesContainer*, który odpowiada za przechowywanie wszystkich aktywnych przedmiotów do zebrania.

Oprócz łupów każdy zebrany przedmiot przyznaje graczowi punkty. Wynik końcowy po zakończonej rozgrywce jest porównywany z dotychczasowym najlepszym rezultatem (rekordem), który przechowywany jest w pamięci lokalnej urządzenia. Do zapisu i odczytu rekordu wykorzystano wbudowany system plików Godota. System zapisu najlepszego wyniku został przedstawiony na listingu 4.17.

Listing 4.17. Obsługa zapisu i odczytu najlepszego wyniku

```
var highscore: int = 0

func _ready():
	load_highscore()

func save_highscore():
	var file = FileAccess.open("user://highscore.save",
	                           FileAccess.WRITE)
	file.store_var(highscore)
	file.close()

func load_highscore():
	if FileAccess.file_exists("user://highscore.save"):
		var file = FileAccess.open("user://highscore.save",
		                           FileAccess.READ)
		highscore = file.get_var()
		file.close()
	else:
		highscore = 0
```

Po uruchomieniu sceny funkcja *_ready()* wywołuje *load_highscore()*, która sprawdza, czy plik z rekordem istnieje. Jeśli tak – wynik zostaje odczytany, a jeśli nie – rekord inicjalizowany jest wartością 0.

Funkcja *save_highscore()* zapisuje aktualny najwyższy wynik do pliku highscore.save. Mechanizm ten pozwala zachować postępy gracza między kolejnymi uruchomieniami gry i stanowi prosty, lecz skuteczny system trwałej pamięci.

5. Interfejs

Wygląd gry jest jednym z kluczowych aspektów wpływających na komfort i satysfakcję z rozgrywki. To, co gracz widzi na ekranie, bezpośrednio oddziałuje na jego odbiór gry oraz podejmowane decyzje. Dlatego projekt interfejsu graficznego powinien być spójny pod względem stylistycznym, intuicyjny w obsłudze, a jednocześnie przejrzysty i pozbawiony zbędnych elementów. W niniejszym rozdziale przedstawiono najważniejsze komponenty interfejsu, które zostały zaimplementowane w grze.

5.1 Menu główne

Menu główne to pierwszy element, z którym gracz ma kontakt przy uruchomieniu gry. Jego zadaniem jest nie tylko umożliwić rozpoczęcie rozgrywki, ale również zbudować pierwsze wrażenie i zaprezentować ogólną stylistykę projektu. W zaprojektowanym menu głównym (rys. 5.1) umieszczono kilka podstawowych elementów:

- **Przycisk „Start"** – rozpoczyna grę i przenosi gracza bezpośrednio do rozgrywki.
- **Przycisk „Quit"** – pozwala zamknąć aplikację.
- **Informacja o najlepszym wyniku** – prezentuje najwyższy wynik osiągnięty przez gracza, co stanowi element motywujący oraz pozwala łatwo śledzić postępy.

Układ menu został zaprojektowany tak, aby wszystkie elementy były dobrze widoczne i jasno sugerowały swoje zastosowanie.

Rys. 5.1. Wygląd głównego menu

5.2 Elementy interfejsu podczas rozgrywki

Podczas gry gracz musi mieć stały dostęp do najważniejszych informacji dotyczących stanu postaci. Dlatego w trakcie rozgrywki wyświetlane są kluczowe elementy interfejsu:

- **Pasek zdrowia** (rys. 5.2) – informuje o aktualnym stanie życia postaci.

Rys. 5.2. Pasek Zdrowia gracza

- **Pasek doświadczenia** (rys. 5.3) – pokazuje postęp do kolejnego poziomu.

Rys. 5.3. Pasek doświadczenia gracza

- **Pasek umiejętności specjalnej** (rys. 5.4) – wskazuje, kiedy umiejętność jest gotowa do użycia.

Rys. 5.4. Pasek umiejętności specjalnej gracza

Dodatkowo, po zdobyciu nowego poziomu doświadczenia pojawia się okno wyboru ulepszenia. Gracz może wybrać jedną z trzech wylosowanych umiejętności. Każda jest przedstawiona za pomocą ramki zawierającej nazwę i opis danej umiejętności (rys. 5.5).

Opisy są krótkie i treściwe aby gracz nie potrzebował zbyt wiele czasu na zrozumienie działania umiejętności.

Rys. 5.5. Ramka opisu umiejętności

5.3 Dodatkowe informacje

Nadmierna ilość informacji wyświetlanych na ekranie może rozpraszać gracza i utrudniać skupienie się na akcji. Dlatego podstawowy interfejs został ograniczony do niezbędnego minimum. Jednocześnie istnieją dane, które mogą być przydatne, lecz nie są potrzebne przez cały czas trwania rozgrywki. Aby je udostępnić w wygodny sposób, wprowadzono system wyświetlania informacji dodatkowych, aktywowany po naciśnięciu klawisza *Tab* (rys. 5.6). Takie rozwiązanie pozwala utrzymać przejrzystość interfejsu, jednocześnie dając graczowi dostęp do pełniejszych danych w momencie, gdy ich potrzebuje.

Rys. 5.6 Informacje dodatkowe (aktualny czas i wynik)

6. Testowanie

Testowanie jest istotnym etapem procesu wytwarzania oprogramowania, którego celem jest weryfikacja poprawności działania systemu oraz jego zgodności z założeniami projektowymi. Pozwala ono na wykrycie błędów, ocenę stabilności aplikacji oraz sprawdzenie spełnienia wymagań funkcjonalnych i niefunkcjonalnych. Niniejszy rozdział przedstawia proces testowania opracowanej gry, obejmujący sprawdzenie kluczowych mechanik rozgrywki i interfejsu użytkownika. Przeprowadzone testy umożliwiły identyfikację i eliminację wykrytych błędów oraz potwierdzenie poprawności działania elementów gry.

6.1 Test głównego menu

Test T01 (tabela 6.1) miał na celu potwierdzenie poprawności interakcji z elementami menu głównego. W jego wyniku sprawdzano, czy przyciski prawidłowo reagują na kursor myszy oraz czy wywołują odpowiednie akcje, takie jak przejście do właściwej sceny gry lub zakończenie działania aplikacji.

Tabela 6.1 Test działania menu głównego

ID	T01
Tytuł	Testowanie przycisków głównego menu
Warunki początkowe	Uruchomiona gra
Kroki testowe	1. Widząc menu kliknij: - Przycisk *„Start”* - Przycisk *„Quit”*
Oczekiwany rezultat	Gra odpowiednio reaguje na kliknięte przyciski: - Przycisk *„Start”* - rozpoczyna się rozgrywka - Przycisk *„Quit”* - aplikacja się zamyka

Rys. 6.1. Kliknięty przycisk w menu głównym

6.2 Testy mechanik gracza

Celem testów w niniejszym podrozdziale była weryfikacja poprawności działania podstawowych mechanik sterowania postacią gracza. Sprawdzono zachowanie systemu ruchu oraz mechanizmu ataku w odpowiedzi na działania użytkownika, a także ich zgodność z założeniami projektowymi gry.

Test T02 (tabela 6.2) ruchu gracza miał na celu sprawdzenie, czy postać poprawnie reaguje na sygnały wejściowe użytkownika oraz porusza się w odpowiednich kierunkach. Weryfikacji poddano płynność ruchu, brak niepożądanych kolizji oraz poprawność aktualizacji pozycji gracza w świecie gry.

Tabela 6.2 Test poruszania się gracza

ID	T02
Tytuł	Testowanie ruchu gracza
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Spróbuj poruszać się z użyciem przycisków “AWSD” 2. Spróbuj przejść przez dowolną przeszkodę 3. Spróbuj przejść przez postać przeciwnika
Oczekiwany rezultat	Postać gracza porusza się płynnie w odpowiednim kierunku w zależności od klikniętych przycisków. W przypadku zderzenia z przeszkodą jest blokowana, a w przypadku zderzenia z przeciwnikiem przechodzi przez niego swobodnie.

Rys. 6.2. Postać gracza koliduje z przeszkodą

Test ataku gracza T03 (tabela 6.3) polegał na sprawdzeniu poprawności działania mechanizmu atakowania przeciwników. Oceniono, czy atak jest wyzwalany zgodnie z założeniami, czy działa w odpowiednim zasięgu oraz czy poprawnie oddziałuje na obiekty przeciwników.

Tabela 6.3 Test ataku gracza

ID	T03
Tytuł	Testowanie ataku gracza
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Spróbuj wybrać kierunek ataku umieszczając kursor myszy po wybranej stronie gracza. 2. Zbliż się do przeciwników, wyceluj atak w ich stronę i poczekaj aż atak ich dosięgnie. 3. Powtórz czynność 2 dopóki przeciwnik nie zniknie (zginie).
Oczekiwany rezultat	Obszar ataku płynnie zmienia pozycje w zależności od umieszczenia kursora. Przeciwnicy trafieni przez atak są odpychani, a po odpowiedniej ilości ataków znikają (umierają).

Rys. 6.3. Atak gracza uderza przeciwnika

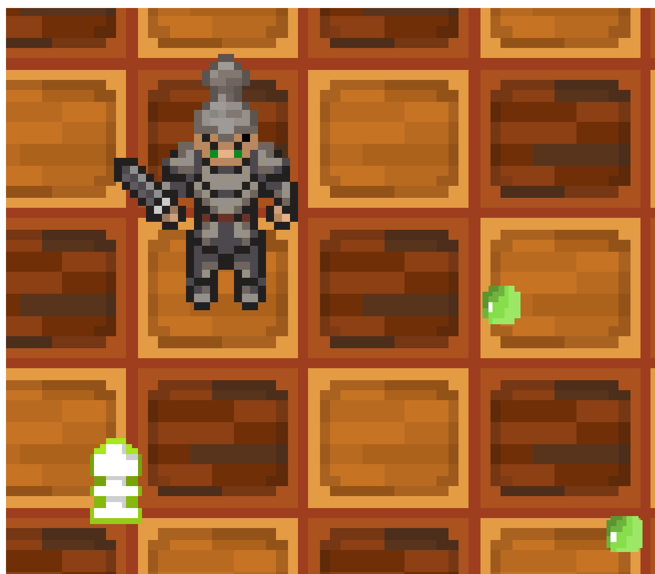
6.3 Testy systemu poziomów

Celem testów w tym podrozdziale była weryfikacja poprawności działania systemu rozwoju postaci. Sprawdzono mechanizmy związane z wypadaniem łupów z przeciwników, ich podnoszeniem przez gracza oraz proces zdobywania kolejnych poziomów doświadczenia. Dodatkowo przetestowano poprawność wyświetlania ekranu wyboru umiejętności oraz wpływ wybranej umiejętności na dalszą rozgrywkę.

Celem testu T04 (tabela 6.4) było sprawdzenie, czy po pokonaniu przeciwnika poprawnie generowane są nagrody zgodnie z założeniami systemu rozwoju postaci. Weryfikacji poddano poprawność pojawiania się łupów oraz ich dostępność dla gracza.

Tabela 6.4 Test wypadania nagród z przeciwników

ID	T04
Tytuł	Testowanie wypadania nagród z przeciwników
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Pokonaj przeciwnika. 2. Sprawdź czy została po nim nagroda w postaci kulki doświadczenia lub pionka specjalnego. 3. Powtórz czynność ok. 30 razy.
Oczekiwany rezultat	Po pokonaniu przeciwników za każdym razem powinien pojawić się jeden z możliwych łupów.

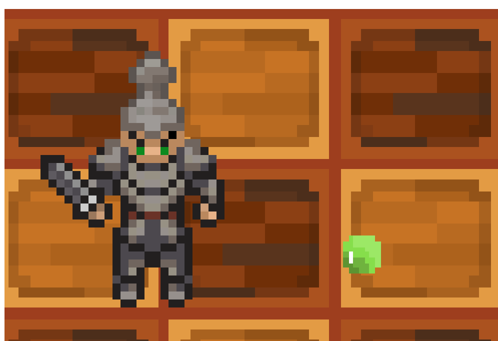


Rys. 6.4. Pozostawione łupy po pokonanych przeciwnikach

Test T05 (tabela 6.5) miał na celu potwierdzenie, że gracz może poprawnie podnosić dostępne łupy oraz że są one prawidłowo przypisywane do systemu rozwoju postaci. Sprawdzono również reakcję gry na interakcję gracza z obiektami łupów.

Tabela 6.5 Test podnoszenia łupów

ID	T05
Tytuł	Testowanie podnoszenia łupów
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Pokonaj przeciwnika 2. Podejdź wystarczają blisko łupu aby automatycznie go podnieść. 3. Podnieś przynajmniej raz każdy rodzaj łupu: - kulka doświadczenia - specjalny pionek
Oczekiwany rezultat	Gracz jest w stanie zbierać przedmioty które pozostawili przeciwnicy. Dodatkowo podniesie konkretnego przedmiotu daje pożądane rezultaty: - kulka doświadczenia - zmienia wartość doświadczenia gracza widoczną na pasku doświadczenia, - specjalny pionek - zmienia stan paska umiejętności specjalnej.



Rys. 6.5. Łup przed podniesieniem

Rys. 6.6. Łup podniesiony

Celem testu T06 (tabela 6.6) było zweryfikowanie poprawności działania systemu zdobywania poziomów doświadczenia oraz mechanizmu wyboru ulepszeń. Sprawdzono, czy po osiągnięciu odpowiedniego poziomu wyświetlany jest ekran wyboru umiejętności oraz czy wybrane ulepszenie jest prawidłowo stosowane w rozgrywce.

Tabela 6.6 Test systemu poziomów i wyboru ulepszeń

ID	T06
Tytuł	Testowanie zdobycia poziomu i wyboru umiejętności
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Pokonuj przeciwników i zbieraj kulki doświadczenia które z nich wypadną, do momentu zapełnienia paska poziomu. 2. Po pojawieniu się interfejsu wyboru umiejętności kliknij w jedną z wylosowanych umiejętności.
Oczekiwany rezultat	Po zapełnieniu paska doświadczenia wyskakuje UI wyboru umiejętności a reszta gry zatrzymuje się. Zawsze pojawiają się 3 umiejętności do wyboru. Po dokonaniu wyboru umiejętności gra się wznawia, a pasek doświadczenia jest znowu możliwy do zapełnienia.

Rys. 6.7. Częściowo wypełniony pasek doświadczenia

6.4 Testy wybranych umiejętności

Niniejszy podrozdział poświęcony jest testom wybranych umiejętności dostępnych w grze. Weryfikacji poddano poprawność ich działania, wpływ na rozgrywkę oraz zgodność z opisanym przeznaczeniem mechanicznym.

Przeprowadzono testy T07 (tabela 6.7) i T08 (tabela 6.8) dwóch zaimplementowanych umiejętności: *„Jumping Horse”* oraz *„Rook's Wall”*. Sprawdzono m.in. sposób aktywacji umiejętności, ich oddziaływanie na przeciwników oraz poprawność działania w różnych sytuacjach podczas rozgrywki. Wybrane umiejętności stanowią przykładowe przypadki testowe, prezentujące sposób weryfikacji mechanik tego typu. Przeprowadzenie testów wszystkich dostępnych umiejętności nie było zasadne ze względu na ich dużą liczbę oraz powtarzalny charakter mechanizmów działania.

Tabela 6.7 Test umiejętności „*Jumping Horse*”

ID	T07
Tytuł	Testowanie umiejętności: „*Jumping Horse*”
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Zdobywaj poziom dopóki w wyborze nie pojawi się umiejętności „*Jumping Horse*” 2. Wybierz umiejętność „*Jumping Horse*”. 3. Poczekaj kilka sekund żeby zauważyć działanie umiejętności
Oczekiwany rezultat	Po wybraniu umiejętności widać systematycznie pojawiające się pionki koni skaczące po szachownicy i uderzające przeciwników przy zderzeniu. Po krótkiej chwili znikają.

Rys. 6.8. Działanie umiejętności „*Jumping Horse*”

Tabela 6.8 Test umiejętności „*Rooks Wall*”

ID	T08
Tytuł	Testowanie umiejętności: „*Rooks Wall*”
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Zdobywaj poziomu dopóki w wyborze nie pojawi się umiejętności „*Rooks Wall*” 2. Wybierz umiejętność „*Rooks Wall*” 3. Poczekaj kilka sekund żeby zauważyć działanie umiejętności
Oczekiwany rezultat	Po wybraniu umiejętności widać systematycznie pojawiające się ściany złożone z wież. Ściana pojawia się z przeciwnej strony niż atakuje gracz i przesuwa się zadając obrażenia przeciwnikom na których wpadnie. Po krótkiej chwili znika.

Rys. 6.9. Działanie umiejętności „*Rooks Wall*”

6.5 Testy systemu punktów

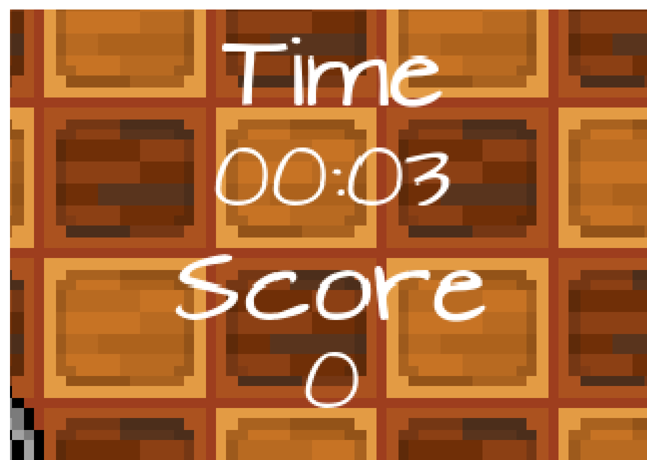
Celem testów w niniejszym podrozdziale była weryfikacja poprawności działania systemu punktacji w grze. Sprawdzono mechanizm przyznawania punktów za pokonywanie przeciwników oraz podnoszenie łupów, a także poprawność ich sumowania w trakcie rozgrywki.

W ramach testów T09 (tabela 6.9) oraz T10 (tabela 6.10) zweryfikowano również poprawność aktualizacji i wyświetlania aktualnego wyniku w dodatkowym menu dostępnym pod klawiszem „Tab”. Dodatkowo sprawdzono działanie systemu najlepszego wyniku

(highscore), który jest zapisywany oraz aktualizowany po zakończeniu każdej rozgrywki i widoczny w menu głównym gry.

Tabela 6.9 Test systemu punktów

ID	T09
Tytuł	Testowanie systemu punktów
Warunki początkowe	Uruchomiona gra, Rozgrywka jest rozpoczęta
Kroki testowe	1. Klikając przycisk „Tab” wyświetl interfejs informacji dodatkowych zawierający punktacje. 2. Pokonaj kilku przeciwników i podnieś z nich łupy. 3. Kliknij ponownie przycisk „Tab”, aby zamknąc interfejs.
Oczekiwany rezultat	Po użyciu przycisku „Tab” interfejs informacji dodatkowych odpowiednio pojawia się i znika. Po podniesieniu łupów widać natychmiastową zmianę wyniku punktów.



Rys. 6.10. Wynik gracza w trakcie gry

Tabela 6.10 Test zdobywania najwyższego wyniku

ID	T10
Tytuł	Testowanie zdobycia nowego najlepszego wyniku
Warunki początkowe	Uruchomiona gra
Kroki testowe	1. Sprawdź ostatni największy wynik w menu głównym. 2. Rozpocznij rozgrywkę i uzyskaj większy wynik niż ostatnio największy. 3. Zakończ rozgrywkę i sprawdź czy wyświetlany jest nowy najnowszy wynik. 4. Wyłącz grę i włącz ją ponownie 5. Sprawdź czy ostatni najwyższy wynik jest odpowiednio wyświetlany.
Oczekiwany rezultat	Najlepszy wynik jest odpowiednio wyświetlany i aktualizowany w głównym menu, zarówno po rozgrywce jak i przy uruchomieniu gry

Rys. 6.11. Widok najwyższego wyniku

7. Podsumowanie i wnioski

Celem niniejszej pracy inżynierskiej było zaprojektowanie oraz zaimplementowanie gry 2D z elementami przetrwania i rozwoju postaci, inspirowanej aktualnymi trendami w branży gier komputerowych. W ramach projektu opracowano kompletną aplikację umożliwiającą dynamiczną rozgrywkę w losowo generowanym świecie, w której zadaniem gracza jest przetrwanie jak najdłużej poprzez eliminację nadciągających grup przeciwników.

W trakcie prac projektowych zaprojektowano i wdrożono kluczowe mechaniki gry, takie jak sterowanie postacią gracza, system przeciwników, generator terenu oraz system progresji oparty na zdobywaniu punktów doświadczenia i ulepszeń. Szczególną uwagę poświęcono zachowaniu spójności mechanik rozgrywki oraz jednolitej oprawy stylistycznej, co pozytywnie wpłynęło na czytelność gry i komfort użytkownika.

Zrealizowany projekt spełnia założone cele funkcjonalne i potwierdza możliwość stworzenia gry 2D oferującej angażującą i powtarzalną rozgrywkę. Implementacja losowo generowanego świata oraz systemu progresji postaci znacząco zwiększa różnorodność rozgrywki i motywuje gracza do dalszej gry.

Podczas realizacji pracy napotkano wyzwania związane z balansowaniem poziomu trudności, optymalizacją działania gry oraz integracją poszczególnych systemów. W celu weryfikacji poprawności działania zastosowano testy funkcjonalne oraz testy rozgrywki, które umożliwiły wykrycie błędów oraz dostosowanie parametrów mechanik gry.

Przedstawiony projekt może stanowić solidną podstawę do dalszego rozwoju poprzez dodanie nowych przeciwników, umiejętności, elementów terenu, przedmiotów czy wprowadzenie bardziej złożonych zmian jak klasy postaci o zróżnicowanych stylach rozgrywki.

Podsumowując, realizacja pracy inżynierskiej umożliwiła praktyczne wykorzystanie zdobytej wiedzy oraz realizację założonych celów projektowych. Osiągnięte rezultaty potwierdzają trafność wyboru tematu pracy oraz skuteczność zastosowanych rozwiązań technicznych.

Bibliografia

[1] Chris B. - *Godot Engine Game Development Projects*, Birmingham, Wielka Brytania - Packt Publishing, 2018.

[2] Alan T. - *Game Development with Godot 4: A Complete Introduction*, Boca Raton, USA - CRC Press, 2025.

[3] Janusz S. - *Podstawy animacji. Projekty 2D*, Gliwice, Polska - Helion, 2014.

[4] Noor S., Julian T., Mark J. N. - *Procedural Content Generation in Games* - Cham, Szwajcaria - Springer 2016.

[5] Dokumentacja Godot 4.4, Listopad 2025
https://docs.godotengine.org/en/4.4/index.html

[6] Dokumentacja GDScript, Listopad 2025
https://docs.godotengine.org/en/4.4/tutorials/scripting/gdscript/index.html

[7] Pixilart
https://www.pixilart.com

Spis rysunków

Rys. 4.1. Widok świata gry

Rys. 4.2. Figura konia

Rys. 4.3. Figura wieży

Rys. 4.4. Widoczne pola kolizji gracza

Rys. 4.5. Widok wyboru nowych umiejętności

Rys. 4.6. Działanie umiejętności specjalnej

Rys. 4.7. Przeciwnicy goniący gracza

Rys. 4.8. Przeciwnik z widocznymi polami i wektorami kolizji

Rys. 4.9. Kula doświadczenia

Rys. 4.10. Specjalny pionek

Rys. 5.1. Wygląd głównego menu

Rys. 5.2. Pasek Zdrowia gracza

Rys. 5.3. Pasek doświadczenia gracza

Rys. 5.4. Pasek umiejętności specjalnej gracza

Rys. 5.5. Ramka opisu umiejętności

Rys. 5.6. Informacje dodatkowe (aktualny czas i wynik)

Rys. 6.1. Kliknięty przycisk w menu głównym

Rys. 6.2. Postać gracza koliduje z przeszkodą

Rys. 6.3. Atak gracza uderza przeciwnika

Rys. 6.4. Wypadnięte łupy z pokonanych przeciwników

Rys. 6.5. Łup przed podniesieniem

Rys. 6.6. Łup podniesiony

Rys. 6.7. Częściowo wypełniony pasek doświadczenia

Rys. 6.8. Działanie umiejętności „*Jumping Horse*”

Rys. 6.9. Działanie umiejętności „*Rooks Wall*”

Rys. 6.10. Wynik gracza w trakcie gry

Rys. 6.11. Widok najwyższego wyniku

Spis tabel

Tabela 6.1 Test działania menu głównego

Tabela 6.2 Test poruszania się gracza

Tabela 6.3 Test ataku gracza

Tabela 6.4 Test wypadania nagród z przeciwników

Tabela 6.5 Test podnoszenia łupów

Tabela 6.6 Test systemu poziomów i wyboru ulepszeń

Tabela 6.7 Test umiejętności „*Jumping Horse*"

Tabela 6.8 Test umiejętności „*Rooks Wall*"

Tabela 6.9 Test systemu punktów

Tabela 6.10 Test zdobywania najwyższego wyniku

Spis listingów

Listing 4.1. Kod odpowiedzialny za inicjowanie generacji nowych segmentów świata

Listing 4.2. Kod odpowiedzialny za generacje nowego segmentu świata

Listing 4.3. Implementacja ruchu gracza.

Listing 4.4. Implementacja ataku gracza.

Listing 4.5. Funkcja odpowiedzialna za otrzymywanie obrażeń.

Listing 4.6. Skrypt odpowiedzialny za podnoszenie przedmiotów.

Listing 4.7. Przykładowe funkcje wzmacniające atak gracza.

Listing 4.8. Implementacja umiejętności specjalnej.

Listing 4.9. Funkcja odpowiedzialna za ruch przeciwnika i wykrywanie kolizji.

Listing 4.10. Funkcja przeciwnika odpowiedzialna za otrzymywanie obrażeń.

Listing 4.11. Implementacja efektu odrzucenia i ogłuszenia przeciwnika.

Listing 4.12. Wyzwalanie ataku przeciwnika po wejściu w jego zasięg.

Listing 4.13. Inicjalizacja systemu fal i przygotowanie pierwszej fali przeciwników.

Listing 4.14. Obsługa generowania kolejnych przeciwników oraz zmiany fal

Listing 4.15. Losowe tworzenie instancji przeciwników w wyznaczonym obszarze.

Listing 4.16. Generowanie łupu po śmierci przeciwnika

Listing 4.17. Obsługa zapisu i odczytu najlepszego wyniku