AKADEMIA TARNOWSKA

Wydział Nauk Technicznych

Kierunek: ***Informatyka***

2025/2026

Mateusz Lasko

PRACA INŻYNIERSKA

Projekt i implementacja gry typu Tower Defense

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2026*

Spis treści

Spis treści....1
1. Wstęp....3
- 1.1. Motywacja....3
- 1.2. Cel pracy....4
- 1.3. Zakres pracy....4

2. Analiza Biznesowa....6
- 2.1 Analiza biznesowa i ogólny opis projektu....6
- 2.2 Wymagania funkcjonalne....6
- 2.3 Wymagania niefunkcjonalne....7
- 2.4 Analiza rynkowa....8

3. Wykorzystane technologie....9
- 3.1 Silnik Godot....9
- 3.2 Język GDScript....10
- 3.3 Środowisko pracy....10
- 3.4 Grafika i zasoby....11
- 3.5 JSON....11

4. Implementacja....12
- 4.1 Implementacja systemu przeciwników....12
- 4.2 System wież i ulepszeń....15
- 4.3 System fal przeciwników....19
- 4.4 System zapisu i odczytu danych....21
- 4.5 Implementacja pocisków i systemu obrażeń....23
- 4.6 Implementacja map i systemu budowy wież....24
- 4.7 Tryby gry: Kampania i Endless....26

5. Interfejsy użytkownika....28
- 5.1 Główne założenia projektowe UI....28
- 5.2 Ekran główny gry....28
- 5.3 Ekran Kampanii....29
- 5.4 Tryb Endless Mode....30
- 5.5.1 Poziom Kampanii....30
- 5.5.2 Menu pauzy i sterowanie rozgrywką....32

6. Testy....35
- 6.1 Scenariusze testowe....35

7. Charakterystyka jednostek w grze....44
- 7.1 Typy wież....44
- 7.1.1 Strzelec (Sniper)....44
- 7.1.2 AOE (obszarowy)....46
- 7.1.3 MG (szybkostrzelny)....47
- 7.2 Typy przeciwników....48

7.2.1 Normalny......48
7.2.2 Szybki...... 49
7.2.3 Opancerzony...... 50
7.3 Skalowanie trudności w trybie endless......50
7.4 Zestawienie statystyk...... 52
8. Podsumowanie i wnioski...... 54
8.1. Osiągnięte cele pracy......54
8.2. Spełnienie założeń projektowych...... 54
8.3. Napotkane trudności...... 55
8.4. Wnioski......55
8.5. Kierunki dalszego rozwoju...... 55
9. Bibliografia...... 56
10. Spis rysunków......57
11. Spis tabel......58
12. Spis listingów...... 59

1. Wstęp

Przedmiotem niniejszej pracy inżynierskiej jest gra komputerowa typu Tower Defense, stworzona przy wykorzystaniu silnika Godot w wersji 4.x. W kolejnych rozdziałach przedstawione zostaną technologie wykorzystane przy jej tworzeniu, analiza wymagań funkcjonalnych i niefunkcjonalnych, szczegółowy projekt systemu, wybrane elementy implementacyjne oraz uzyskane efekty działania aplikacji.

1.1. Motywacja

Gry komputerowe od wielu lat stanowią jedną z najpopularniejszych form rozrywki, łącząc w sobie elementy artystyczne, techniczne i logiczne. Szczególne miejsce wśród nich zajmują gry strategiczne typu Tower Defense, które cieszą się niezmienną popularnością od ponad dekady. Mechanika tego gatunku opiera się na obronie wyznaczonego punktu przed falami przeciwników poruszających się po ustalonej ścieżce, przy wykorzystaniu wież obronnych różniących się parametrami takimi jak zasięg, obrażenia, szybkość ataku czy koszt budowy.

Popularność gier Tower Defense wynika z ich unikalnego połączenia prostoty zasad z głęboką strategią. Gracz musi nie tylko zrozumieć podstawowe mechaniki, ale także opracować optymalne rozmieszczenie wież, zarządzać ograniczonymi zasobami oraz reagować na zmieniające się warunki rozgrywki. Klasyczne tytuły takie jak „Plants vs Zombies", „Bloons Tower Defense" czy „Kingdom Rush" pokazały, że gatunek ten ma ogromny potencjał komercyjny i może przyciągać miliony graczy na różnych platformach – od komputerów osobistych po urządzenia mobilne.

Motywacją do powstania niniejszego projektu była osobista pasja autora do gier komputerowych, która narodziła się w dzieciństwie podczas gry w „Plants vs Zombies". Z czasem zainteresowanie ewoluowało w kierunku zrozumienia technicznych aspektów tworzenia gier – jak powstają systemy przeciwników, logika fal, mechaniki wież czy interfejs użytkownika. Studia informatyczne dostarczyły niezbędnej wiedzy z programowania obiektowego, algorytmiki i projektowania systemów, co naturalnie skierowało autora ku realizacji własnej gry w ramach pracy dyplomowej.

Współczesny rozwój otwartych silników gier, takich jak Godot, sprawił, że tworzenie profesjonalnych aplikacji 2D stało się dostępne nie tylko dla dużych studiów deweloperskich, ale także dla indywidualnych twórców i studentów. Projekt ten stanowi więc idealne połączenie osobistych zainteresowań z praktycznym zastosowaniem wiedzy inżynierskiej

zdobytej podczas siedmiu semestrów studiów na kierunku Informatyka w Akademii Tarnowskiej.

1.2. Cel pracy

Celem niniejszej pracy inżynierskiej było zaprojektowanie oraz zaimplementowanie kompletnej gry typu Tower Defense, działającej w środowisku Godot 4.x z wykorzystaniem języka skryptowego GDScript. Stworzona aplikacja ma pełnić funkcję w pełni grywalnej rozrywki, promującej logiczne myślenie, planowanie przestrzenne oraz zarządzanie zasobami, przy jednoczesnym zachowaniu wysokiej jakości technicznej i intuicyjnego interfejsu użytkownika.

Na rynku istnieje wiele gier Tower Defense, jednak większość z nich została stworzona przez profesjonalne zespoły deweloperskie. Niewiele produkcji tego typu powstaje w ramach projektów studenckich, co stanowiło dodatkowe wyzwanie. Głównym celem aplikacji było stworzenie gry oferującej dwa tryby rozgrywki – kampanię z wieloma poziomami oraz tryb endless – z pełnym systemem wież, przeciwników, fal i zarządzania postępami gracza.

Jednym z kluczowych założeń projektu było zapewnienie modularnej architektury kodu, umożliwiającej łatwą rozbudowę gry o nowe typy wież, przeciwników, poziomy czy mechaniki bez konieczności przebudowy istniejących komponentów. Szczegółowe wymagania dotyczące funkcji systemu, zarówno funkcjonalne jak i niefunkcjonalne, zostały opisane w rozdziale analizy biznesowej.

1.3. Zakres pracy

W ramach projektu inżynierskiego zaimplementowano kompletną grę komputerową składającą się z następujących głównych modułów:

- Silnik gry i logika podstawowa – implementacja w Godot 4.x z wykorzystaniem GDScript.
- System przeciwników – ruch po ścieżce, system życia/pancerza, różnorodne typy wrogów.
- System wież – budowa, ulepszanie, automatyczny atak, różne typy jednostek obronnych.

- System fal – generowanie fal w trybie kampanii i endless z rosnącą trudnością.
- Interfejs użytkownika – menu główne, ekrany trybów gry, HUD z informacjami o stanie.
- System zapisu/odczytu – przechowywanie postępów trybu endless.
- Grafika i zasoby – pixel-art 2D przygotowany przy użyciu narzędzi graficznych (GIMP) oraz wspomagany generatywnie przez AI, z dodatkiem darmowych assetów.

Gra łączy się wewnętrznie poprzez system węzłów (nodes) i sygnałów (signals) silnika Godot, tworząc spójną całość. Najważniejsze funkcjonalności zostały przetestowane pod kątem poprawności działania mechanik budowy wież, systemu obrażeń, generowania fal oraz wydajności przy dużej liczbie obiektów na ekranie.

Wszystkie wymienione komponenty zostały szczegółowo opisane w dalszych rozdziałach pracy, wraz z fragmentami kodu źródłowego, diagramami architektury oraz wynikami testów funkcjonalnych i wydajnościowych. Zakres projektu obejmuje pełny cykl wytwarzania oprogramowania - od analizy wymagań, przez projektowanie i implementację, aż po testowanie i dokumentację.

Dalsza część pracy została zorganizowana następująco: rozdział 2 zawiera analizę biznesową i rynkową, rozdział 3 opisuje wykorzystane technologie, rozdziały 4-6 szczegółowo omawiają projekt i implementację kolejnych modułów gry, rozdział 7 prezentuje wyniki testów, a 8 rozdział zawiera podsumowanie i wnioski dotyczące możliwości dalszego rozwoju aplikacji.

2. Analiza Biznesowa

2.1 Analiza biznesowa i ogólny opis projektu

Celem projektu jest projekt i implementacja gry komputerowej typu Tower Defense, w której gracz musi bronić się przed kolejnymi falami przeciwników, rozmieszczając wieże obronne w różnych miejscach na mapie. Pomysł na grę powstał z chęci połączenia zainteresowań autora związanych z grami komputerowymi i programowaniem. Projekt ma też pozwolić na praktyczne wykorzystanie wiedzy zdobytej podczas studiów, głównie z zakresu programowania obiektowego, projektowania interfejsów i algorytmiki.

Gra ma przede wszystkim pełnić funkcję rozrywkową, ale też w pewnym stopniu edukacyjną - rozwijać logiczne myślenie, planowanie oraz zarządzanie zasobami. Rozgrywka będzie polegać na tym, że gracz broni swojej bazy lub punktu kontrolnego przed falami wrogów. Z każdą kolejną falą poziom trudności będzie się zwiększał, a przeciwnicy będą coraz silniejsi. Gracz będzie miał możliwość stawiania i ulepszania wież, które automatycznie atakują wrogów, zanim ci dotrą do bazy.

Gra będzie miała dwa główne tryby: kampanię, w której gracz przechodzi kolejne poziomy, oraz tryb endless, w którym fale przeciwników pojawiają się bez końca, a celem jest przetrwanie jak najdłużej. W ten sposób projekt łączy klasyczny model gry z elementem rywalizacji i bicia rekordów.

2.2 Wymagania funkcjonalne

Podstawowym wymaganiem funkcjonalnym jest to, żeby gra pozwalała na rozgrywkę w trybie kampanii. Kampania będzie składała się z kilku map (poziomów), które różnią się wyglądem i rozmieszczeniem ścieżek, po których poruszają się wrogowie. Każdy poziom będzie miał swój własny zestaw fal przeciwników, zróżnicowanych pod względem liczebności, szybkości i wytrzymałości.

Kolejnym elementem jest tryb endless, w którym przeciwnicy pojawiają się bez końca, a gra kończy się dopiero wtedy, gdy przeciwnicy zniszczą bazę gracza. Tryb ten ma na celu zapewnienie dłuższej rozgrywki i sprawdzenie umiejętności gracza w warunkach ciągłego wzrostu trudności.

Gracz będzie miał do dyspozycji kilka rodzajów wież, które różnią się między sobą zasięgiem, szybkością ataku i zadawanymi obrażeniami. Dodatkowo będzie można te wieże ulepszać w trakcie gry, wydając walutę zdobytą za pokonanych przeciwników. Ulepszenia

zwiększą ich skuteczność i pozwolą na lepsze dopasowanie strategii obrony do rodzaju nadchodzących wrogów.

Wrogowie w grze będą podzieleni na różne typy - normalnych, szybkich, opancerzonych. Taka różnorodność ma sprawić, że gracz będzie musiał zmieniać strategię i nie będzie mógł korzystać z jednej taktyki przez całą grę.

Interfejs użytkownika ma być prosty i przejrzysty. Na ekranie będą widoczne podstawowe informacje, takie jak liczba fal, poziom życia bazy, ilość waluty i dostępne wieże. Menu główne pozwoli na rozpoczęcie gry, wybór kampanii lub trybu endless, a także zakończenie rozgrywki.

Gra będzie zapamiętywać postępy gracza, takie jak najwyższy wynik w trybie endless. Dane będą zapisywane lokalnie, aby można było później kontynuować grę od wybranego momentu.

2.3 Wymagania niefunkcjonalne

Wymagania niefunkcjonalne dotyczą tego, jak gra ma działać i jakie powinna mieć cechy techniczne. Przede wszystkim gra powinna działać płynnie na komputerach z systemem Windows, nawet na sprzęcie o średnich parametrach. Projekt ma być zoptymalizowany tak, żeby nie powodował dużego obciążenia procesora i karty graficznej. Interfejs graficzny ma być czytelny i przyjemny dla oka. Nie chodzi o bardzo rozbudowaną grafikę, tylko o spójną i estetyczną oprawę, w której łatwo znaleźć potrzebne informacje. Kolory i układ elementów mają być dobrane tak, żeby nie rozpraszały gracza i pozwalały mu skupić się na samej rozgrywce. Kod gry będzie pisany w sposób modularny, co oznacza, że poszczególne elementy (np. wieże, wrogowie, interfejs, logika rozgrywki) będą oddzielone od siebie. Dzięki temu w przyszłości będzie można łatwo dodawać nowe funkcje, takie jak kolejne poziomy, typy wież czy przeciwników, bez potrzeby modyfikowania całej aplikacji. Aplikacja powinna być odporna na błędy użytkownika - nie może się zawieszać ani kończyć działania w przypadku błędnych danych. Na przykład, jeśli gracz spróbuje ustawić wieżę w niedozwolonym miejscu, gra powinna po prostu wyświetlić komunikat, że to niemożliwe, a nie przerywać rozgrywki. Gra ma być intuicyjna i łatwa do opanowania nawet dla osób, które nie grały wcześniej w produkcje typu Tower Defense. Menu, przyciski i komunikaty muszą być zrozumiałe, a sterowanie proste - najlepiej przy użyciu samej myszy.

2.4 Analiza rynkowa

Gry typu Tower Defense są obecne na rynku od wielu lat i cieszą się niezmienną popularnością zarówno wśród młodszych, jak i starszych graczy. Ich głównym atutem jest połączenie prostych zasad z koniecznością logicznego myślenia i planowania strategii. Gatunek ten rozwijał się wraz z postępem technologicznym – od prostych produkcji przeglądarkowych, po rozbudowane tytuły mobilne i komputerowe. Do najbardziej znanych przedstawicieli tego typu gier można zaliczyć:

- **Plants vs Zombies** – jedna z najpopularniejszych gier w historii gatunku, która łączy elementy strategii z humorystyczną oprawą graficzną.
- **Bloons Tower Defense (BTD)** – seria gier, w której gracze rozmieszczają różne wieże, aby powstrzymać fale balonów. Znana z ogromnej liczby ulepszeń i poziomów trudności.
- **Kingdom Rush** – przykład dopracowanej gry z bogatą oprawą graficzną i systemem progresji postaci, dostępnej na PC i urządzenia mobilne.

Na rynku można zauważyć, że większość popularnych gier Tower Defense koncentruje się na prostocie rozgrywki, intuicyjnym interfejsie i dużej różnorodności przeciwników. Wiele z nich stosuje system rozwoju lub ulepszania wież, który motywuje gracza do dalszej gry.

W ostatnich latach widoczny jest również trend łączenia gatunku Tower Defense z innymi mechanikami, takimi jak elementy *RPG(role-playing game)*, symulacja ekonomii lub tryby kooperacyjne. To pokazuje, że mimo długiej historii gatunku, nadal istnieje duży potencjał do tworzenia nowych i oryginalnych tytułów.

Projekt realizowany w ramach tej pracy inżynierskiej ma na celu stworzenie własnej wersji gry Tower Defense, która łączy klasyczne zasady z nowoczesnym podejściem do rozgrywki. Gra będzie wyróżniać się prostą, przejrzystą oprawą graficzną oraz trybem endless, pozwalającym na długotrwałą zabawę i bicie własnych rekordów. Dzięki temu projekt wpisuje się w aktualne trendy, a jednocześnie zachowuje charakter edukacyjny i poznawczy, pozwalając autorowi wykorzystać zdobytą wiedzę techniczną.

3. Wykorzystane technologie

3.1 Silnik Godot

Silnik Godot Engine został wybrany jako podstawowa technologia do realizacji projektu ze względu na swoją prostotę, otwartość oraz rozbudowane możliwości w zakresie tworzenia gier 2D. Jest to środowisko typu open-source, rozwijane przez społeczność, które nie wymaga żadnych opłat licencyjnych ani prowizji od przychodu, co wyróżnia je na tle konkurencji, takiej jak Unity czy Unreal Engine [1].

Godot udostępnia użytkownikowi intuicyjny interfejs graficzny, w którym każdy element gry jest reprezentowany przez tzw. węzły (nodes). Każdy węzeł może posiadać własne właściwości, metody i sygnały, co pozwala na modularne budowanie logiki gry. Z kolei zbiory węzłów mogą tworzyć **sceny**, które następnie można osadzać w innych scenach, tworząc złożone struktury hierarchiczne. Dzięki temu twórca może z łatwością modyfikować pojedyncze elementy gry, bez ryzyka uszkodzenia reszty projektu [2].

Silnik zapewnia obsługę zarówno gier 2D, jak i 3D, jednak w przypadku tej pracy zdecydowano się skupić na środowisku 2D, które jest w pełni wystarczające dla realizacji założeń gry typu Tower Defense. Moduł 2D w Godocie pozwala na dokładne odwzorowanie fizyki, kolizji, animacji oraz efektów wizualnych. Dodatkowo silnik oferuje wbudowane narzędzia do testowania i profilowania wydajności, co pozwala monitorować zużycie zasobów i optymalizować kod w czasie rzeczywistym.

Warto wspomnieć, że w wersji Godot 4.x, wykorzystanej w projekcie, wprowadzono nowy system renderowania o nazwie *Forward+*, który zapewnia lepszą jakość grafiki i większą płynność działania nawet przy dużej liczbie obiektów na ekranie. Ponadto nowa wersja silnika umożliwia eksport projektów na różne platformy, w tym Windows, Linux, macOS oraz Android. Dzięki temu gotowa gra może być łatwo udostępniona na wielu urządzeniach bez konieczności wprowadzania istotnych zmian w kodzie.

Kolejnym atutem Godota jest wbudowany edytor wizualny, który pozwala na tworzenie poziomów w trybie graficznym. Dzięki temu autor może rozmieszczać elementy mapy, punkty startowe wrogów czy pozycje wież obronnych w sposób intuicyjny, bez konieczności wpisywania współrzędnych ręcznie. Silnik oferuje też możliwość podglądu rozgrywki „na żywo” (ang. *live debugging*), co znacząco przyspiesza proces testowania i wprowadzania poprawek.

Pod względem architektury projektu, Godot wspiera wzorzec Model-View-Controller (MVC), co sprzyja utrzymaniu przejrzystości kodu i rozdzieleniu logiki od warstwy prezentacji. Dzięki temu rozwijanie gry, np. dodawanie nowych typów wrogów lub ulepszeń wież, jest procesem prostym i bezpiecznym.

3.2 Język GDScript

Logika gry została zaimplementowana w języku GDScript, który został zaprojektowany specjalnie na potrzeby silnika Godot. Jest to język wysokiego poziomu, składniowo bardzo zbliżony do Pythona, co ułatwia naukę i zrozumienie jego zasad. Według dokumentacji [3], GDScript jest zoptymalizowany do komunikacji z silnikiem, dzięki czemu działa szybciej niż ogólne języki skryptowe wykorzystywane w grach (np. Lua czy JavaScript).

Język ten pozwala na definiowanie klas, dziedziczenie oraz korzystanie z systemu sygnałów, które umożliwiają obiektom reagowanie na określone zdarzenia w grze (np. zniszczenie wroga, kliknięcie wieży, rozpoczęcie nowej fali). Dzięki temu struktura kodu jest czytelna, a jego rozbudowa nie wymaga dużych modyfikacji.

W ramach projektu język GDScript zostanie wykorzystany m.in. do:

- sterowania ruchem wrogów po ścieżce,
- obsługi kolizji i zadawania obrażeń,
- zarządzania falami przeciwników,
- obsługi logiki interfejsu (punkty, życie, zasoby),
- oraz zapisu postępów gracza.

W porównaniu z Pythonem, GDScript ma kilka kluczowych różnic — m.in. jest kompilowany do kodu bajtowego wewnątrz silnika, a nie interpretowany, co pozwala na większą wydajność. Dodatkowo Godot zapewnia pełną integrację z edytorem kodu, w tym kolorowanie składni, podpowiedzi, oraz debugowanie w czasie rzeczywistym.

3.3 Środowisko pracy

Do tworzenia projektu wykorzystany zostanie wbudowany edytor Godot, który integruje w sobie edytor scen, animacji, kodu oraz interfejsu użytkownika. Wszystkie zmiany są natychmiast widoczne w podglądzie, co przyspiesza proces iteracji i testowania.

Projekt będzie rozwijany w systemie operacyjnym Windows 11, jednak Godot jako aplikacja wieloplatformowa umożliwia bezproblemową pracę również w systemach Linux oraz

macOS. Nie wymaga instalacji dodatkowych bibliotek, co znacząco upraszcza przenoszenie projektu między komputerami.

3.4 Grafika i zasoby

Grafika i zasoby – pixel-art 2D przygotowany przy użyciu narzędzi graficznych (GIMP) oraz wspomagany generatywnie przez AI, z dodatkiem darmowych assetów.kiem darmowych assetów

Grafika w grze została utrzymana w prostym, kolorowym stylu 2D. Elementy wizualne, takie jak wieże, przeciwnicy, pociski oraz tła, zostały wygenerowane przy użyciu narzędzi sztucznej inteligencji np.canva, a następnie poddane obróbce w programach graficznych GIMP oraz Krita. Dodatkowo wykorzystano darmowe assety z otwartych źródeł, takich jak OpenGameArt.org, z zachowaniem zasad licencji Creative Commons [4]. Takie podejście pozwoliło uzyskać spójną i estetyczną oprawę wizualną, typową dla klasycznych gier Tower Defense.

3.5 JSON

W grze zaimplementowano system zapisu oparty na formacie JSON, który przechowuje najlepszy wynik osiągnięty w trybie endless. Plik savegame.json jest przechowywany lokalnie i zawiera pojedynczą wartość - rekord gracza (endless_best). System przewidziano z możliwością rozbudowy o zapis postępów w kampanii, stanu złota czy pozycji wież, jednak w obecnej implementacji skupiono się na kluczowym elemencie motywującym do dalszej gry - biciu własnych rekordów w trybie endless.

4. Implementacja

Implementacja gry typu Tower Defense została przeprowadzona na podstawie wcześniej przygotowanych założeń projektowych oraz wymagań funkcjonalnych. Całość oparto na silniku Godot 4.x oraz języku skryptowym GDScript, który pozwala w stosunkowo prosty sposób tworzyć logikę gier 2D. Proces wdrażania wykonywano etapowo, zaczynając od przygotowania podstawowych elementów rozgrywki, a następnie ich dalszej rozbudowy i integracji. W implementacji zwracano uwagę na modularność kodu, tak aby kolejne elementy można było dodawać bez konieczności przerabiania istniejącej logiki.

Podczas implementacji szczególny nacisk położono na czytelność struktury, podział projektu na sceny oraz unikanie nadmiernego powiązania elementów. Dzięki temu możliwe było niezależne tworzenie wież, przeciwników, fal oraz interfejsu użytkownika. W dalszych etapach dodawano kolejne systemy, takie jak zapis postępów, tryby gry oraz mechanizmy obsługi zdarzeń. Rozdział opisuje szczegółowo wszystkie główne moduły oraz sposób ich działania, a także przedstawia przebieg implementacji całego systemu gry.

4.1 Implementacja systemu przeciwników

System przeciwników w grze został oparty na wspólnej scenie Enemy, której rootem jest węzeł CharacterBody2D. Takie rozwiązanie umożliwia wykorzystanie wbudowanej fizyki 2D silnika Godot do obsługi ruchu oraz kolizji, a jednocześnie pozwala łatwo tworzyć kolejne warianty przeciwników na bazie jednego skryptu.

W przedstawionym fragmencie kodu zdefiniowano zestaw parametrów opisujących statystyki jednostki: maksymalną liczbę punktów życia *(max_hp)*, prędkość poruszania się *(speed)*, nagrodę za jej zniszczenie (*gold_value*) oraz współczynnik pancerza *(armor)*. Zmienna hp przechowuje aktualny stan zdrowia, inicjalizowany w metodzie *_ready()* wartością *max_hp*.

Ruch przeciwnika po mapie realizowany jest za pomocą tablicy punktów kontrolnych *path_points*, przekazywanej z poziomu sceny gry za pomocą funkcji *set_path()*. Metoda ta ustawia listę kolejnych współrzędnych, po których jednostka ma się poruszać, oraz ustawia indeks bieżącego punktu na wartość początkową. Dodatkowo po wywołaniu *set_path()* pozycja przeciwnika jest ustawiana na pierwszy punkt ścieżki, dzięki czemu cała logika ruchu może zakładać, że jednostka zawsze startuje z poprawnego miejsca.

W funkcji *_physics_process()* zaimplementowano właściwy algorytm podążania po ścieżce. Dla aktualnego indeksu *current_index* pobierany jest docelowy punkt *target_pos*, a następnie obliczany jest znormalizowany wektor kierunku od obecnej pozycji przeciwnika do tego punktu. Wektor ten jest mnożony przez prędkość speed i przypisywany do zmiennej velocity, co pozwala wykorzystać metodę *move_and_slide()* do przesunięcia jednostki w przestrzeni. Dodatkowo sprite przeciwnika jest obracany w kierunku ruchu, co zwiększa realizm wizualny. W momencie, gdy odległość od celu spadnie poniżej ustalonego progu (4 piksele), indeks ścieżki jest zwiększany o jeden. Jeżeli po inkrementacji *current_index* wyjdzie poza zakres tablicy, oznacza to, że przeciwnik dotarł do końca trasy – w takim przypadku emitowany jest sygnał *reached_goal*, a jednostka usuwa się z drzewa sceny. Pozwala to scenie poziomu zareagować na przejście przeciwnika, odejmując punkty życia bazie gracza.

Logika obrażeń oraz śmierci przeciwnika została skupiona w metodzie *apply_damage()*. Funkcja przyjmuje wartość nominalnych obrażeń oraz flagę pierce określającą, czy atak przebija pancerz. W zależności od flagi obliczany jest końcowy efekt – dla ataków przebijających pancerz obrażenia są przekazywane w pełnej wartości, w przeciwnym razie uwzględniany jest współczynnik armor zgodnie ze wzorem amount * (1.0 - armor). W ten sposób dla przeciwników o wysokim pancerzu realne obrażenia będą odpowiednio niższe, co pozwala budować zróżnicowane typy jednostek bez modyfikacji kodu pocisku. Po odjęciu obrażeń od hp następuje sprawdzenie, czy punkty życia spadły do zera lub poniżej. W takim przypadku przeciwnik emituje sygnał died z przekazaną wartością nagrody *gold_value*, a następnie usuwa się z gry poprzez *queue_free()*.

Scena poziomu korzysta z tych sygnałów do aktualizacji stanu gry. Przy tworzeniu nowych jednostek *WaveManager* przypisuje im wspólną ścieżkę ruchu przez wywołanie *set_path(path*) oraz podłącza sygnały died i *reached_goal* do odpowiednich metod obsługi. W reakcji na sygnał died zwiększana jest ilość złota gracza, a interfejs użytkownika odświeża wyświetlaną wartość. Z kolei *reached_goa*l powoduje zmniejszenie liczby punktów życia bazy oraz sprawdzenie warunku przegranej. Takie rozdzielenie odpowiedzialności sprawia, że przeciwnik samodzielnie zarządza swoim ruchem i życiem, natomiast logika nagród i kar pozostaje po stronie nadrzędnego menedżera poziomu.

Listing 4.1. Fragment skryptu Enemy.gd realizujący ruch po ścieżce i obrażenia

```
extends CharacterBody2D

signal died(gold_value: int)
signal reached_goal

@export var max_hp: int = 100
@export var speed: float = 80.0
@export var gold_value: int = 10
@export var armor: float = 0.0
@export var enemy_type: int = 0

var hp: int
var path_points: Array[Vector2] = []
var current_index: int = 0

func _ready() -> void:
    hp = max_hp
    _setup_visuals()

func _setup_visuals() -> void:
    match enemy_type:
        0:
            $Sprite2D.texture = load("res://enemy_normal.png")
        1:
            $Sprite2D.texture = load("res://enemy_fast.png")
        2:
            $Sprite2D.texture = load("res://enemy_armored.png")

func set_path(points: Array[Vector2]) -> void:
    path_points = points
    current_index = 0
    if path_points.size() > 0:
        global_position = path_points[0]

func _physics_process(_delta: float) -> void:
    if path_points.size() == 0 or current_index >=
path_points.size():
        return
    var target_pos := path_points[current_index]
    var dir := (target_pos - global_position).normalized()
    velocity = dir * speed
```

```
    move_and_slide()
    if global_position.distance_to(target_pos) < 4.0:
        current_index += 1
        if current_index >= path_points.size():
            emit_signal("reached_goal")
            queue_free()

func apply_damage(amount: int, pierce: bool = false) -> void:
    var final_damage: int
    if pierce:
        final_damage = amount
    else:
        final_damage = int(amount * (1.0 - armor))
    hp -= final_damage
    if hp <= 0:
        emit_signal("died", gold_value)
        queue_free()
```

4.2 System wież i ulepszeń

Wieże w grze zostały zaimplementowane jako osobne sceny, których rootem jest węzeł Node2D. Każda scena zawiera element graficzny reprezentujący wieżę oraz skrypt odpowiedzialny za logikę ataku.

Podstawowe parametry wieży, takie jak zasięg działania, obrażenia, szybkość ataku i koszt budowy, zostały zdefiniowane jako zmienne eksportowane *(base_range, base_damage, base_fire_rate, base_cost)*. Dzięki temu można je wygodnie dostosowywać dla różnych typów wież bez modyfikowania kodu.

Każda wieża posiada parametr level, który określa aktualny poziom ulepszenia. W metodzie *_ready()* wywoływana jest funkcja *_recalculate_stats()*, przeliczająca bieżące statystyki: *current_range, current_damage, current_fire_rate* oraz koszt kolejnego ulepszenia *(next_upgrade_cost)*. Zasięg i obrażenia rosną liniowo wraz z poziomem, a szybkość ataku zwiększa się poprzez skrócenie interwału między strzałami.

Wieże mają system wyboru celów. Gracz może zmieniać tryb celowania, aby lepiej dostosować strategię:

- Najbliższy – atakuje przeciwnika najbliżej wieży.

- Pierwszy – atakuje przeciwnika najbliżej końca ścieżki.
- Najsilniejszy – atakuje przeciwnika z największą liczbą punktów życia

Wieże typu AOE (Area of Effect) zadają obrażenia wszystkim przeciwnikom w zasięgu jednocześnie i wywołują efekt wizualny w postaci pomarańczowego błysku. Atakowanie przeciwników realizowane jest w funkcji *_process()*. Wieża śledzi czas od ostatniego strzału w zmiennej time_since_shot. Po przekroczeniu progu 1.0 / *current_fire_rate* wywoływana jest metoda *_find_enemy_in_range()*. Funkcja wybiera cel zgodnie z trybem celowania, a następnie wywołuje *_shoot(enemy)* dla zwykłych wież lub *_shoot_aoe()* dla wież obszarowych.

Funkcja *can_upgrade(current_gold)* sprawdza, czy gracz ma wystarczającą ilość złota na ulepszenie, porównując bieżące zasoby z next_upgrade_cost. Jeżeli to możliwe, metoda *apply_upgrade()* zwiększa poziom level i ponownie uruchamia *_recalculate_stats()*, aktualizując wszystkie parametry bojowe zgodnie z modelem skalowania.

Listing 4.2. Fragment skryptu Tower.gd - wyszukiwanie celów i strzelanie

```
func _process(delta: float) -> void:
	current_target = _find_enemy_in_range()
	time_since_shot += delta
	if time_since_shot >= 1.0 / current_fire_rate:
		if is_aoe:
			_shoot_aoe()
			time_since_shot = 0.0
		elif current_target != null and 
is_instance_valid(current_target):
			_shoot(current_target)
			time_since_shot = 0.0

func _find_enemy_in_range() -> Node2D:
	var enemies_in_range = []
	for enemy in get_tree().get_nodes_in_group("enemies"):
		if not enemy is Node2D:
			continue
		if not is_instance_valid(enemy):
			continue
		var d := 
global_position.distance_to(enemy.global_position)
		if d <= current_range:
```

```
                enemies_in_range.append(enemy)

        if enemies_in_range.is_empty():
            return null

        match target_mode:
            0:
                var closest: Node2D = null
                var closest_dist := current_range + 1.0
                for enemy in enemies_in_range:
                    var d :=
global_position.distance_to(enemy.global_position)
                    if d < closest_dist:
                        closest = enemy
                        closest_dist = d
                return closest
            1:
                var first: Node2D = null
                var min_dist := 999999.0
                for enemy in enemies_in_range:
                    if "path_points" in enemy and "current_index" in
enemy:
                        var dist = 0.0
                        var idx = enemy.current_index
                        if idx < enemy.path_points.size():
                            dist +=
enemy.global_position.distance_to(enemy.path_points[idx])
                            for i in range(idx,
enemy.path_points.size() - 1):
                                dist +=
enemy.path_points[i].distance_to(enemy.path_points[i + 1])
                        if dist < min_dist:
                            first = enemy
                            min_dist = dist
                return first if first != null else enemies_in_range[0]
            2:
                var strongest: Node2D = null
                var max_hp := -1
                for enemy in enemies_in_range:
                    if "hp" in enemy and enemy.hp > max_hp:
                        strongest = enemy
                        max_hp = enemy.hp
```

```
            return strongest if strongest != null else
enemies_in_range[0]
        _:
            return enemies_in_range[0]

func _shoot(enemy: Node2D) -> void:
    var b := BulletScene.instantiate()
    b.global_position = global_position
    b.damage = current_damage
    b.target = enemy
    b.armor_pierce = armor_pierce
    get_tree().current_scene.add_child(b)

func _shoot_aoe() -> void:
    var hit_any = false
    for enemy in get_tree().get_nodes_in_group("enemies"):
        if not enemy is Node2D:
            continue
        var d :=
global_position.distance_to(enemy.global_position)
        if d <= current_range:
            if enemy.has_method("apply_damage"):
                enemy.apply_damage(current_damage, false)
                hit_any = true

    if hit_any:
        var flash = ColorRect.new()
        flash.color = Color(1.0, 0.5, 0.0, 0.3)
        flash.size = Vector2(current_range * 2, current_range * 2)
        flash.position = Vector2(-current_range, -current_range)
        add_child(flash)
        await get_tree().create_timer(0.1).timeout
        flash.queue_free()
```

Listing 4.3. Fragment skryptu Tower.gd - system ulepszeń

```
func can_upgrade(current_gold: int) -> bool:
    if level >= 3:
        return false
    return current_gold >= next_upgrade_cost

func apply_upgrade() -> void:
    if level >= 3:
```

```
        return
    level += 1
    _recalculate_stats()

func _recalculate_stats() -> void:
    current_range = base_range * (1.0 + 0.2 * (level - 1))
    current_damage = int(base_damage * (1.0 + 0.3 * (level - 1)))
    current_fire_rate = base_fire_rate * (1.0 + 0.15 * (level -
1))
    next_upgrade_cost = base_cost + upgrade_cost_step * (level -
1)
```

4.3 System fal przeciwników

System fal przeciwników został wydzielony do osobnego skryptu *WaveManager.gd*, który pełni rolę generatora fal oraz licznika aktywnych jednostek.

Na początku zdefiniowano listę słowników waves, opisujących konfigurację poszczególnych fal: liczbę przeciwników *(count)*, odstęp czasu między ich pojawieniem się *(delay)*, mnożniki punktów życia i prędkości *(enemy_hp_mult, enemy_speed_mult)*, dodatkową nagrodę za ukończenie fali *(reward)* oraz tablicę typów przeciwników *(types)*.
W grze zdefiniowano 6 predefiniowanych fal o rosnącym poziomie trudności, od prostych po zaawansowane fale z mieszanką typów wrogów.

Metoda *start_next_wave()* uruchamia kolejną falę. W trybie kampanii korzysta z danych zapisanych w tablicy waves, natomiast w trybie endless generuje nową falę dynamicznie przy użyciu funkcji *_generate_endless_wave()*. Funkcja ta zwiększa stopniowo liczbę przeciwników oraz mnożniki punktów życia i prędkości, podnosząc poziom trudności.
Funkcja *_spawn_single_enemy()* instancjuje pojedynczego przeciwnika na podstawie sceny *EnemyScene*, ustawia mu ścieżkę ruchu przez *set_path(path)* oraz modyfikuje parametry max_hp i speed zgodnie z mnożnikami zapisanymi w słowniku fali. Podłącza sygnały died i *reached_goal* do funkcji, które aktualizują licznik alive_enemies. Gdy liczba utworzonych przeciwników osiąga count, timer zostaje zatrzymany.

Sygnały związane z życiem przeciwników wykorzystywane są do detekcji końca fali. W lambdach podłączonych do died i *reached_goal* zmniejszany jest licznik alive_enemies. Gdy liczba żywych przeciwników spada do zera, emitowany jest sygnał *wave_finished*, informujący nadrzędny menedżer poziomu o zakończeniu fali i możliwości przygotowania kolejnej.

Listing 4.4. Fragment skryptu WaveManager.gd - definicja fal i ich tworzenie

```
var waves := [
    {"count": 5, "delay": 1.2, "enemy_hp_mult": 1.0,
"enemy_speed_mult": 0.8, "reward": 2, "types": [0, 0, 0, 0, 0]},
    {"count": 7, "delay": 1.0, "enemy_hp_mult": 1.1,
"enemy_speed_mult": 0.85, "reward": 3, "types": [0, 0, 1, 0, 0, 1,
0]},
    {"count": 9, "delay": 0.9, "enemy_hp_mult": 1.2,
"enemy_speed_mult": 0.9, "reward": 3, "types": [0, 1, 0, 2, 0, 0,
1, 0, 2]},
    {"count": 11, "delay": 0.8, "enemy_hp_mult": 1.4,
"enemy_speed_mult": 0.95, "reward": 4, "types": [2, 0, 1, 0, 2, 0,
1, 0, 2, 0, 1]},
    {"count": 13, "delay": 0.7, "enemy_hp_mult": 1.6,
"enemy_speed_mult": 1.0, "reward": 5, "types": [2, 1, 0, 2, 1, 0,
2, 1, 2, 0, 1, 2, 0]},
    {"count": 16, "delay": 0.6, "enemy_hp_mult": 2.0,
"enemy_speed_mult": 1.1, "reward": 6, "types": [2, 2, 1, 0, 2, 1,
2, 0, 2, 1, 0, 2, 2, 1, 0, 2]},
]

var endless_mode: bool = false
var current_wave_index: int = 0
var alive_enemies: int = 0
var path: Array[Vector2] = []

func set_path(points: Array[Vector2]) -> void:
    path = points

func start_next_wave() -> void:
    if not endless_mode and current_wave_index >= waves.size():
        emit_signal("wave_finished")
        return
    var wave_data: Dictionary
    if endless_mode:
        wave_data = _generate_endless_wave()
    else:
        wave_data = waves[current_wave_index]
    current_wave_index += 1
    _spawn_wave(wave_data)

func _generate_endless_wave() -> Dictionary:
```

```
    return {
        "count": 5 + current_wave_index * 2,
        "delay": max(0.4, 0.9 - current_wave_index * 0.03),
        "enemy_hp_mult": 1.0 + 0.08 * current_wave_index,
        "enemy_speed_mult": 1.0 + 0.03 * current_wave_index,
        "reward": 3 + current_wave_index,
        "types": []
    }

func _spawn_single_enemy(wave_data: Dictionary, wave_timer: Timer)
-> void:
    var e := EnemyScene.instantiate()
    e.set_path(path)
    e.max_hp = int(e.max_hp * wave_data["enemy_hp_mult"])
    e.speed = e.speed * wave_data["enemy_speed_mult"]
    e.died.connect(func(value: int) -> void:
        alive_enemies -= 1
        emit_signal("enemy_died", value + wave_data["reward"])
        if alive_enemies <= 0:
            emit_signal("wave_finished"))
    e.reached_goal.connect(func() -> void:
        alive_enemies -= 1
        emit_signal("enemy_reached_goal")
        if alive_enemies <= 0:
            emit_signal("wave_finished"))
    get_tree().current_scene.add_child(e)
    alive_enemies += 1
```

4.4 System zapisu i odczytu danych

System zapisu gry bazuje na lokalnym pliku w formacie JSON, przechowywanym w katalogu użytkownika silnika Godot.

Za obsługę zapisu odpowiada skrypt *SaveManager.gd*. Zdefiniowano w nim stałą *SAVE_PATH*, wskazującą lokalizację pliku, oraz metody *save_game()* i *load_game()*. Funkcja *save_game()* przyjmuje słownik z danymi gry, serializuje go do formatu JSON za pomocą *JSON.stringify()* i zapisuje w pliku przez FileAccess. Funkcja *load_game()* sprawdza, czy plik istnieje, odczytuje jego zawartość i próbuje zdeserializować tekst do słownika. W przypadku błędu zwracany jest pusty słownik.

System zapisu wykorzystywany jest głównie do przechowywania najlepszego wyniku w trybie endless. Po zakończeniu gry sprawdzane jest, czy aktualny wynik *(current_wave)* jest wyższy od zapisanego rekordu *(endless_best)*. Jeżeli tak, nowy wynik zostaje zapisany w pliku.

Struktura danych pozwala też na przechowywanie numeru ostatnio ukończonego poziomu kampanii, stanu złota oraz listy wież, co umożliwia przyszłą rozbudowę systemu zapisu.

Listing 4.5. Fragment skryptu SaveManager.gd – zapis i odczyt danych

```
const SAVE_PATH = "user://savegame.json"

func save_game(data: Dictionary) -> void:
    var file = FileAccess.open(SAVE_PATH, FileAccess.WRITE)
    if file:
        file.store_string(JSON.stringify(data))
        file.close()

func load_game() -> Dictionary:
    if not FileAccess.file_exists(SAVE_PATH):
        return {}
    var file = FileAccess.open(SAVE_PATH, FileAccess.READ)
    if not file:
        return {}
    var text = file.get_as_text()
    file.close()
    var result = JSON.parse_string(text)
    if result is Dictionary:
        return result
    return {}

func build_save_data(campaign_level: int, endless_best: int, gold:
int, towers: Array) -> Dictionary:
    return {
        "campaign_level": campaign_level,
        "endless_best": endless_best,
        "gold": gold,
        "towers": towers
    }
```

4.5 Implementacja pocisków i systemu obrażeń

System pocisków w grze oparty jest na osobnej scenie *Bullet*, której głównym węzłem jest Area2D.
Pocisk posiada dwa podstawowe parametry: speed i damage. Zmienna target przechowuje referencję do aktualnie wybranego przeciwnika. W funkcji *_process()* sprawdzane jest, czy cel nadal istnieje w drzewie scen. Jeśli przeciwnik został zniszczony, pocisk wywołuje *queue_free()* i znika z planszy, co zapobiega pozostawaniu „osieroconych" pocisków.
Jeżeli cel jest aktywny, pocisk oblicza aktualny wektor kierunku jako znormalizowaną różnicę między pozycją celu a własną pozycją. Pozycja pocisku jest przesuwana o *direction * speed * delta*, co daje efekt pocisku podążającego za przeciwnikiem, podobnie do mechaniki samonaprowadzających pocisków.

Moment zadawania obrażeń również sprawdzany jest w *_process()*. Gdy pocisk znajduje się w odległości mniejszej niż 20 pikseli od celu, wywoływana jest metoda *apply_damage()* na obiekcie przeciwnika z wartością damage i flagą *armor_pierce*, a następnie pocisk usuwa się.

Po stronie przeciwnika logika obrażeń znajduje się w metodzie *apply_damage()*. Klasa *Enemy* przechowuje aktualne punkty życia w zmiennej hp oraz parametr armor reprezentujący redukcję obrażeń. Funkcja oblicza efektywne obrażenia jako iloczyn nominalnej wartości i (1.0 - armor) i odejmuje je od hp. Jeśli hp spadnie do zera lub poniżej, przeciwnik wywołuje *queue_free()*.

Strzelanie pociskami inicjuje wieża. W klasie *Tower* przechowywana jest referencja do sceny pocisku *(BulletScene)*, wartość obrażeń *(damage)* oraz częstotliwość strzałów *(fire_rate)*. Funkcja *_process()* zlicza czas od ostatniego wystrzału i po przekroczeniu progu wywołuje *_shoot(enemy)*. Metoda *_shoot()* tworzy nową instancję pocisku, nadaje jej pozycję wieży, wartość obrażeń oraz referencję do przeciwnika jako target. Dodanie pocisku do sceny powoduje, że jego ruch jest obsługiwany przez skrypt *Bullet.gd*.
Taka architektura pozwala łatwo rozbudowywać system pocisków. Można tworzyć nowe sceny pocisków z odmiennym zachowaniem w *_process()* (np. wybuch obszarowy, spowolnienie czy podpalenie), pozostawiając niezmienioną integrację z wieżami i przeciwnikami. Dzięki temu nowe typy ataków można dodawać bez ingerencji w kod logiki poziomów, co ułatwia rozwój projektu i balans rozgrywki.

Listing 4.6. Fragment skryptu Bullet.gd – ruch pocisku i obsługa kolizji

```
extends Area2D

var damage: int = 20
var target: Node2D = null
var speed: float = 300.0
var armor_pierce: bool = false

func _ready() -> void:
    $Sprite2D.modulate = Color(1.0, 0.9, 0.0)

func _process(delta: float) -> void:
    if not is_instance_valid(target):
        queue_free()
        return
    var dir := (target.global_position -
global_position).normalized()
    global_position += dir * speed * delta
    if global_position.distance_to(target.global_position) < 20.0:
        if target.has_method("apply_damage"):
            target.apply_damage(damage, armor_pierce)
        queue_free()
```

4.6 Implementacja map i systemu budowy wież

Mapa poziomu definiuje przestrzeń, na której gracz może rozstawiać wieże oraz po której poruszają się przeciwnicy. W implementacji zastosowano podział na obszar budowy oraz ścieżkę przeciwników. Ścieżka jest reprezentowana poprzez tablicę punktów kontrolnych *(path_points)* oraz prostokąty kolizji *(path_rects)* wykorzystywane do sprawdzania, czy dane miejsce jest dozwolone do budowy (listing 4.7).

Panel wyboru wież w interfejsie użytkownika łączy się bezpośrednio z logiką budowy. Naciśnięcie przycisku odpowiedzialnego za wieżę ustawia zmienną *placing_tower* na *true* oraz *selected_tower_type* na odpowiednią wartość. Oznacza to, że następne kliknięcie myszy na planszy zostanie zinterpretowane jako próba postawienia wieży. Obsługa budowania odbywa się wewnątrz funkcji *_input()*, która reaguje na kliknięcia lewym przyciskiem myszy.

Przed postawieniem wieży sprawdzanych jest kilka warunków: czy gracza stać na zakup (odpowiedni koszt w zależności od typu wieży), czy kliknięcie znajduje się w dozwolonym obszarze budowy (odpowiedni zakres współrzędnych Y), czy nie koliduje z istniejącymi wieżami (odległość mniejsza niż 64 piksele) oraz czy nie znajduje się na ścieżce przeciwników. Dopiero po spełnieniu wszystkich warunków wieża jest tworzona i dodawana do kontenera *TowerContainer*, a odpowiednia ilość złota jest odejmowana od zasobów gracza.

Listing 4.7. Fragment skryptu poziomu – system budowy wież

```
func _try_place_tower(pos: Vector2) -> void:
    var cost = tower1_cost
    if selected_tower_type == 2:
        cost = tower2_cost
    elif selected_tower_type == 3:
        cost = tower3_cost
    if gold < cost or not _can_place_on_position(pos):
        placing_tower = false
        return

    var tower = TowerScene.instantiate()
    tower.global_position = pos
    if selected_tower_type == 2:
        tower.base_damage = 30
        tower.base_fire_rate = 0.5
        tower.base_range = 120.0
        tower.is_aoe = true
        tower.get_node("Sprite2D").texture =
load("res://tower_aoe.png")
    elif selected_tower_type == 3:
        tower.base_damage = 8
        tower.base_fire_rate = 3.0
        tower.base_range = 130.0
        tower.get_node("Sprite2D").texture =
load("res://tower_mg.png")
    else:
        tower.armor_pierce = true
        tower.get_node("Sprite2D").texture =
load("res://tower_sniper.png")
    $TowerContainer.add_child(tower)
```

```
    gold -= cost
    placing_tower = false
    _update_hud()
```

4.7 Tryby gry: Kampania i Endless

Gra została zaprojektowana tak, aby zawierała dwa główne tryby rozgrywki: tryb kampanii oraz tryb endless. Każdy z tych trybów wymagał osobnej implementacji logiki fal przeciwników, warunków zwycięstwa oraz sposobu naliczania punktów.

Tryb kampanii opiera się na zestawie ręcznie przygotowanych poziomów (level1.tscn, level2.tscn, level3.tscn), gdzie każda mapa posiada określoną liczbę fal przeciwników. Po ukończeniu wszystkich fal gracz wygrywa poziom i może przejść do kolejnego. Każdy poziom został zaprojektowany ze stopniowo rosnącym poziomem trudności — level1 zawiera 4 fale, level2 5 fal, a level3 6 fal z dwiema ścieżkami jednocześnie. Implementacja poszczególnych poziomów znajduje się w plikach level1.gd, level2.gd oraz level3.gd

Tryb endless *(endless_level.gd)* został zaprojektowany w odmienny sposób. W tym trybie fale generowane są w sposób nieskończony, a każda kolejna fala jest trudniejsza od poprzedniej. Zwiększenie trudności odbywa się poprzez stopniowe podnoszenie punktów życia, prędkości oraz liczby przeciwników (funkcja *_generate_endless_wave()* w *WaveManager.gd*). Po piątej fali aktywowana zostaje druga ścieżka *(both_paths_active = true)*, co dodatkowo zwiększa poziom trudności. Gracz zdobywa punkty za każdą pokonaną falę *(score = current_wave * 100)*, a po przegranej najlepszy wynik zapisywany jest jako rekord *(endless_best)* w pliku savegame.json.

Listing 4.8. Fragment skryptu endless_level.gd – logika trybu endless

```
var gold: int = 150
var base_hp: int = 30
var current_wave: int = 0
var score: int = 0
var both_paths_active: bool = false

func _ready() -> void:
    wave_manager = $WaveManager
```

```
    wave_manager.endless_mode = true
    wave_manager.wave_finished.connect(_on_wave_finished)
    wave_manager.enemy_died.connect(_on_enemy_died)
    wave_manager.enemy_reached_goal.connect(_on_enemy_reached_goal)

func _on_start_fala() -> void:
    wave_manager.start_next_wave()
    if both_paths_active:
        wave_manager_bottom.start_next_wave()
    $HUD/BtnStartFala.disabled = true

func _on_wave_finished() -> void:
    _waves_finished_this_round += 1
    var expected = 2 if both_paths_active else 1
    if _waves_finished_this_round < expected:
        return
    _waves_finished_this_round = 0
    current_wave += 1
    score = current_wave * 100
    if current_wave >= 5 and not both_paths_active:
        both_paths_active = true
        _show_message("Uwaga! Druga brama otwarta!")
    $HUD/BtnStartFala.disabled = false
    _update_hud()

func _game_over() -> void:
    var data = save_manager.load_game()
    var best = data.get("endless_best", 0)
    if current_wave > best:
        var new_data =
save_manager.build_save_data(data.get("campaign_level", 0),
current_wave, gold, [])
        save_manager.save_game(new_data)
```

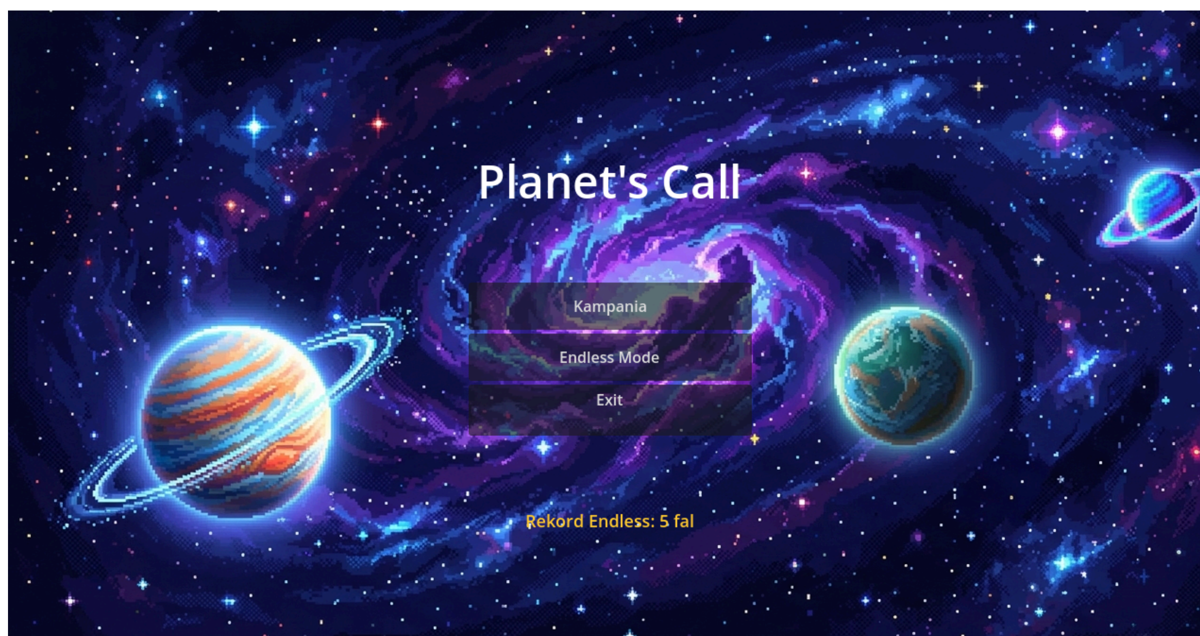
5. Interfejsy użytkownika

5.1 Główne założenia projektowe UI

Interfejs użytkownika w grze Tower Defense pełni kluczową rolę, ponieważ umożliwia graczowi szybki dostęp do wszystkich funkcji gry, kontrolowanie przebiegu rozgrywki i monitorowanie stanu zasobów. Podczas projektowania UI kierowano się zasadą czytelności, minimalizmu oraz intuicyjnej obsługi. Wszystkie elementy zostały rozmieszczone w taki sposób, aby były łatwo dostępne, a jednocześnie nie zasłaniały pola gry. Interfejs został zaprojektowany tak, aby był w pełni skalowalny i poprawnie działał na różnych rozdzielczościach. Dzięki wykorzystaniu węzłów Control, kontenerów oraz kotwic elementy UI automatycznie dostosowują swoją pozycję i rozmiar. W kolejnych podrozdziałach opisano szczegółowo zaprojektowane ekrany oraz ich funkcjonalność.

5.2 Ekran główny gry

Po uruchomieniu gry użytkownik zostaje przeniesiony do ekranu głównego (Rysunek 1.). Jest to centralny hub, z którego gracz wybiera tryb rozgrywki oraz ma dostęp do ustawień. Interfejs został zaprojektowany tak, by był prosty i czytelny. Na środku górnej części ekranu znajduje się logo gry Planet Call. Pod logiem umieszczono przyciski pełniące rolę nawigacyjną. Pierwszym przyciskiem jest Kampania, który przenosi gracza do listy poziomów w trybie głównym. Poniżej znajduje się Endless Mode, czyli tryb przetrwania. Ostatnim przyciskiem jest Exit zamykający aplikację.

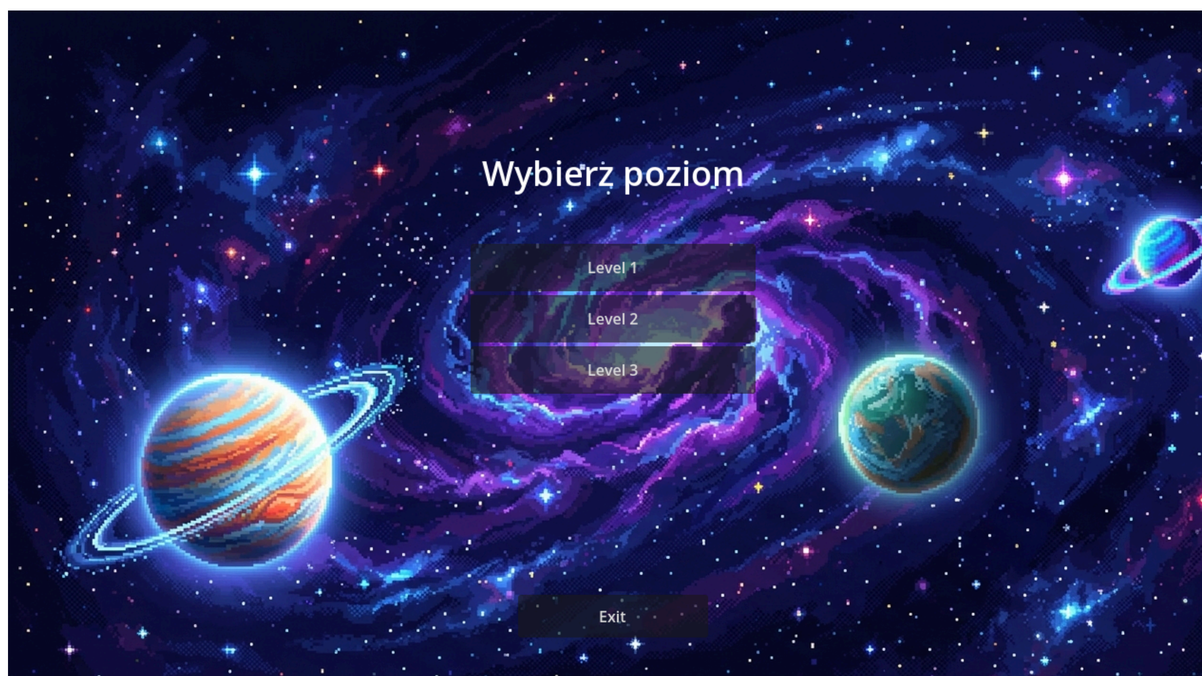


Rys. 5.1. Ekran główny.

Interfejs został zaprojektowany z myślą o nowych graczach, dlatego wszystkie elementy są rozmieszczone symetrycznie i zgodnie z typowymi nawykami użytkowników. Centralne umieszczenie logo i menu zapewnia przejrzystość. Zadbano o czytelność tekstu poprzez zastosowanie kontrastowej czcionki oraz odpowiednich marginesów wokół przycisków. W przyszłości planowane jest dodanie animowanego tła, np. powoli poruszającej się planety, co zwiększy atrakcyjność wizualną menu. Interfejs główny będzie rozwijany o system zapisów, profile gracza oraz osiągnięcia.

5.3 Ekran Kampanii

Po wybraniu Kampanii gracz przechodzi do ekranu z mapą świata gry (Rysunek 2). Jest to statyczny widok planety z rozmieszczonymi punktami reprezentującymi poziomy. Każdy poziom ma własną ikonę. W przyszłości przewidziano również możliwość odblokowywania dodatkowych poziomów.



Rys. 5.2. Mapa kampanii.

5.4 Tryb Endless Mode

Po wybraniu Trybu Endless Mode gracz zostaje przeniesiony na specjalną mapę z otwartym układem ścieżek (Rysunek 3). Celem gry jest przetrwanie jak najdłużej podczas kolejnych fal przeciwników.

Fale wrogów w tym trybie generowane są dynamicznie w zależności od numeru aktualnej fali (current_wave_index):

- Liczba wrogów rośnie liniowo wraz z kolejnymi falami.
- Opóźnienie między pojawianiem się wrogów maleje w miarę postępu gry, co zwiększa tempo pojawiania się wrogów.
- Statystyki wrogów (punkty życia i prędkość) są mnożone przez współczynniki zależne od numeru fali, dzięki czemu wrogowie stają się trudniejsi w kolejnych rundach.
- Nagroda za pokonanie wroga jest stała w obrębie jednej fali, ale rośnie wraz z numerem fali – każdy wróg z danej fali daje tyle samo złota.

Rys. 5.3. Tryb Endless.

5.5.1 Poziom Kampanii

Centralnym punktem interfejsu jest widok pojedynczego poziomu (Rysunek 4.), czyli ekran, na którym toczy się zasadnicza rozgrywka. Większość przestrzeni zajmuje plansza z

zaznaczoną ścieżką, po której przemieszczają się wrogowie od punktu wejścia aż do bazy gracza. Ścieżka jest wyraźnie odróżniona od reszty tła, co pozwala intuicyjnie śledzić kierunek kolejnych fal przeciwników i planować rozmieszczenie wież jeszcze przed pojawieniem się pierwszych jednostek.

Na dole, w lewym rogu ekranu, umieszczono pasek wyboru wież. Każda wieża jest reprezentowana przez prostokątny przycisk z krótką nazwą, na przykład „Strzelec", „AOE" lub „MG". Rozwiązanie oparte na tekście zamiast samych ikon ma na celu maksymalną czytelność – gracz już na etapie nauki gry nie musi zapamiętywać symboli, lecz może od razu kojarzyć funkcję wieży z jej nazwą. Po kliknięciu jednego z przycisków interfejs przechodzi w tryb budowania: kursor myszy wskazuje miejsce potencjalnego postawienia wieży, a kolejne kliknięcie na planszy powoduje umieszczenie wybranej konstrukcji, pod warunkiem że dane pole jest dozwolone do budowy.

Taki sposób interakcji jest intuicyjny i zbliżony do rozwiązań znanych z innych gier strategicznych. Gracz czytelnie przełącza się pomiędzy dostępnymi rodzajami wież, a jednocześnie zachowuje pełną kontrolę nad docelowym miejscem ich ustawienia. W przyszłości pasek można łatwo rozbudować o dodatkowe funkcje – na przykład podpowiedzi wyświetlane po najechaniu kursorem, prezentujące zasięg ataku, koszt budowy czy opis specjalnych umiejętności wybranej wieży.

W lewym górnym rogu ekranu znajduje się panel zasobów. Kluczową informacją jest tutaj wartość złota, prezentowana w postaci prostego napisu: „Złoto: X". Złoto pełni rolę głównej waluty gry – jest zdobywane za pokonywanie kolejnych przeciwników i wykorzystywane do budowy nowych wież lub ewentualnych ulepszeń. Umieszczenie tej informacji w górnym rogu pozwala graczowi jednym spojrzeniem ocenić swoje możliwości ekonomiczne, bez konieczności odrywania wzroku od pola gry na dłużej.

Panel zasobów jest stale widoczny podczas całej rozgrywki, niezależnie od liczby przeciwników na ekranie i intensywności akcji. Dzięki temu nawet w dynamicznych sytuacjach gracz może podjąć szybką decyzję, czy lepiej zainwestować w nową wieżę, czy odczekać na zebranie większej ilości środków. Takie rozwiązanie sprzyja budowaniu nawyku ciągłego kontrolowania sytuacji ekonomicznej, co jest istotne w grach typu tower defense, gdzie zbyt pochopne wydanie zasobów może utrudnić obronę w późniejszych falach.

Środkowa część ekranu, poza planszą i pociskami, pozostaje możliwie wolna od elementów interfejsu. Jedynymi dynamicznymi obiektami w tej strefie są jednostki wroga, wieże oraz ich pociski. Minimalna liczba dodatkowych ikon i paneli sprawia, że gracz może

szybko ocenić sytuację taktyczną – rozróżnić typy wrogów po kolorach, zauważyć luki w obronie oraz przewidzieć, w których miejscach warto zainwestować w dodatkowe wieże.

Rys. 5.4. Poziom 1 trybu kampanii.

5.5.2 Menu pauzy i sterowanie rozgrywką

Drugim istotnym elementem interfejsu poziomu jest menu pauzy, odpowiedzialne za zatrzymywanie gry oraz podstawowe akcje związane z zarządzaniem bieżącym poziomem. Do menu pauzy prowadzi przycisk „Menu" umieszczony w lewym górnym rogu ekranu. Takie położenie jest czytelne i zgodne z przyzwyczajeniami użytkowników – w wielu aplikacjach oraz grach kluczowe przyciski systemowe znajdują się właśnie w tym miejscu.
Po kliknięciu przycisku „Menu" gra zostaje wstrzymana, a na środku ekranu pojawia się półprzezroczysty panel. Panel zawiera trzy przyciski: „Wróć do gry", „Restart" oraz „Menu Główne". Taki zestaw opcji obejmuje wszystkie najczęściej wykorzystywane akcje związane z zarządzaniem aktualną sesją gry.

Przycisk „Resume" służy do kontynuowania rozgrywki. Jego działanie jest natychmiastowe – po kliknięciu panel znika, gra zostaje odwieszona, a wszystkie jednostki i pociski wracają do swoich standardowych zachowań. Rozwiązanie to pozwala zatrzymać grę na krótki moment, na przykład w celu przeanalizowania sytuacji na planszy, bez ryzyka

utraty postępów. Jest to szczególnie przydatne na wyższych poziomach trudności, gdy liczba przeciwników i wież rośnie, a podejmowanie decyzji „w biegu” staje się coraz trudniejsze. Przycisk „Restart” umożliwia szybkie rozpoczęcie bieżącego poziomu od nowa. Po jego użyciu gra resetuje stan planszy: wszystkie wieże i wrogowie są usuwani, a licznik zasobów oraz fala przeciwników wracają do wartości początkowych. Funkcja ta jest skierowana przede wszystkim do graczy, którzy chcą przetestować inną strategię obrony lub popełnili błąd na wczesnym etapie misji i wolą rozpocząć ją jeszcze raz zamiast kontynuować w niekorzystnej sytuacji.

Trzeci przycisk, „Main Menu”, odpowiada za zakończenie bieżącej rozgrywki i powrót do ekranu głównego gry. Po jego wybraniu poziom zostaje przerwany, a gra przenosi użytkownika z powrotem do centralnego huba, w którym można wybrać inną misję, zmienić tryb rozgrywki lub zakończyć działanie aplikacji. Umieszczenie tej opcji w menu pauzy sprawia, że gracz nie musi szukać sposobu wyjścia z poziomu w dodatkowych podmenu – wszystkie kluczowe akcje są dostępne w jednym, prostym oknie.

Ważnym założeniem projektu było to, aby menu pauzy było łatwo dostępne, ale jednocześnie nie przeszkadzało w standardowej rozgrywce. Dlatego przycisk „Menu” ma niewielkie rozmiary i znajduje się w rogu ekranu, natomiast sam panel pojawia się tylko w momencie aktywnego wstrzymania gry. Podczas normalnej gry cała centralna przestrzeń pozostaje więc przeznaczona wyłącznie na obserwację pola bitwy. Takie podejście pozwala utrzymać rytm rozgrywki, a jednocześnie daje graczowi pełną kontrolę nad czasem – może on zatrzymać poziom w dowolnej chwili, przeanalizować sytuację, zdecydować o kontynuowaniu lub całkowitym wyjściu z misji.

Rys. 5.5. Menu Pauzy.

6. Testy

Niniejszy rozdział poświęcony został procesowi weryfikacji i walidacji aplikacji gry Tower Defense. Testy przeprowadzono w celu potwierdzenia poprawności implementacji wymagań funkcjonalnych oraz zapewnienia stabilności działania na docelowej platformie. Obejmują one zarówno weryfikację kluczowych mechanik rozgrywki (system budowy wież, walka z przeciwnikami, zarządzanie zasobami), jak i aspekty techniczne takie jak wydajność, obsługa błędów oraz integralność systemu zapisu i odczytu danych.

Testowanie realizowano etapami, począwszy od testów jednostkowych poszczególnych modułów (Enemy.gd, Tower.gd, WaveManager.gd), a następnie przejść do testów integracyjnych całych systemów gry. Każdy przypadek testowy (Test Case) zawiera szczegółowy opis warunków początkowych, listy kroków do wykonania oraz spodziewane rezultaty, co umożliwia zarówno powtarzalność testów, jak i łatwą identyfikację ewentualnych defektów.

6.1 Scenariusze testowe

W pierwszym kroku przetestowano poprawne uruchomienie poziomu kampanii oraz inicjalizację stanu gry. Celem testu jest sprawdzenie, czy po wybraniu poziomu gra wczytuje właściwą mapę oraz wyświetla komplet kluczowych informacji startowych dla gracza (życia bazy, początkowe zasoby, panel wież, licznik fali). Dzięki temu można zweryfikować, że rozgrywka zaczyna się w spójnym i przewidywalnym stanie.

Tabela 6.1 Przypadek testowy uruchomienia poziomu kampanii

ID	TC001
Tytuł	Uruchomienie poziomu kampanii i wyświetlenie stanu początkowego gry.
Warunki początkowe	Gra uruchomiona, menu główne widoczne, tryb kampanii dostępny. Pierwszy poziom nie został wcześniej ukończony.
Kroki testowe	1. Z menu głównego wybierz opcję „Kampania". 2. Z listy poziomów wybierz pierwszy dostępny poziom. 3. Obserwuj ekran gry.

Oczekiwany rezultat	Ładuje się mapa pierwszego poziomu, wyświetlana jest liczba żyć bazy (np. 20), aktualny stan zasobów (złoto: 100), lista dostępnych wież w panelu po lewej stronie oraz licznik fali (Fala 1 z N). Animacja ścieżki i oprawy graficznej jest widoczna bez zamieszania.

Rys. 6.1. Menu poziomu 1.

Kolejny scenariusz weryfikuje podstawową funkcjonalność budowania wież na dozwolonych polach. Test sprawdza, czy po wybraniu odpowiedniego typu wieży i kliknięciu w kafelek budowy gra poprawnie tworzy obiekt wieży, aktualizuje zasoby gracza oraz odświeża interfejs. Jest to kluczowe, ponieważ mechanika stawiania wież stanowi podstawę całej rozgrywki.

Tabela 6.2 Przypadek testowy budowy wieży na dozwolonym polu

ID	**TC002**
Tytuł	Budowa wieży na kafelku oznaczonym jako pole budowy.
Warunki początkowe	Poziom kampanii jest uruchomiony, gracz posiada 100 złota (wystarczająca ilość dla wieży podstawowej).

Kroki testowe	1. Z panelu wież wybierz wieżę typu „Strzelec" (koszt: 50 złota). 2. Kliknij na kafelek oznaczony jako dozwolone pole budowy. 3. Obserwuj zmianę stanu zasobów.
Oczekiwany rezultat	Wieża pojawia się na wybranym polu, liczba złota zmniejsza się z 100 do 50, interfejs aktualizuje wyświetlaną wartość zasobów w prawym górnym rogu ekranu. Wieża jest aktywna i gotowa do ataku.

Rys. 6.2. Aktywna wieża.

W tym przypadku testowany jest mechanizm walidacji miejsca budowy. Scenariusz sprawdza, czy gra poprawnie blokuje możliwość stawiania wież na ścieżce przeciwników i w innych niedozwolonych lokalizacjach. Istotne jest, aby w takiej sytuacji zasoby gracza nie były modyfikowane, a aplikacja przekazywała czytelny komunikat o błędzie, co zapobiega frustracji i potencjalnym błędom logiki.

Tabela 6.3 Przypadek testowy budowy wieży w niedozwolonym miejscu

ID	TC003
Tytuł	Próba budowy wieży na ścieżce przeciwników.

Warunki początkowe	Poziom kampanii jest uruchomiony, gracz posiada wystarczającą ilość złota. Na mapie widoczna jest wyraźnie zaznaczona ścieżka poruszania się wrogów.
Kroki testowe	1. Z panelu wież wybierz dowolną wieżę. 2. Spróbuj postawić ją na kafelku należącym do ścieżki przeciwników. 3. Obserwuj odpowiedź aplikacji.
Oczekiwany rezultat	Wieża nie zostaje zbudowana, zasoby gracza (złoto) nie zmieniają się, pojawia się komunikat ostrzegawczy kursor zmienia kolor na czerwony, sygnalizując niedozwolone miejsce. Gra pozostaje stabilna bez błędów.

Rys. 6.3. Próba budowy wieży w niedozwolonym miejscu.

Ten scenariusz koncentruje się na kluczowej mechanice automatycznego atakowania wrogów przez wieże. Celem testu jest potwierdzenie, że wieża prawidłowo wykrywa przeciwników w swoim zasięgu, obraca się w ich kierunku, generuje pociski oraz że obrażenia są prawidłowo naliczane. Dzięki temu można ocenić, czy podstawowa pętla rozgrywki – wejście wroga w zasięg i jego eliminacja – działa zgodnie z założeniami projektowymi.

Tabela 6.4 Przypadek testowy automatycznego ataku wieży

ID	**TC004**
Tytuł	Automatyczne strzelanie wieży do przeciwnika w zasięgu.
Warunki początkowe	Na mapie stoi wieża typu „Strzelec" na poziomie 1. Pierwsza fala przeciwników jest aktywna, do mapy wszedł co najmniej jeden przeciwnik.
Kroki testowe	1. Poczekaj, aż przeciwnik wejdzie w zasięg wieży (wizualnie powinien być w promieniu koła zasięgu). 2. Obserwuj wieżę przez kilka sekund. 3. Sprawdź, czy pociski są wytwarzane i trafiają w cel.
Oczekiwany rezultat	Wieża automatycznie emituje pociski w kierunku przeciwnika, pociski są wizualnie widoczne (animacja lotu), liczba HP przeciwnika maleje z każdym trafieniem, wieża zmienia cel na następnego przeciwnika po zniszczeniu pierwszego.

Rys. 6.4. Automatyczne strzelanie wieży do przeciwnika w jej zasięgu.

W tym przypadku testowana jest funkcja rozwoju wież poprzez ulepszenia. Scenariusz sprawdza, czy po wykonaniu akcji ulepszenia zmieniają się statystyki bojowe wieży, poziom jest poprawnie prezentowany w interfejsie oraz czy koszt ulepszenia zostaje odjęty od zasobów gracza. Poprawne działanie tej mechaniki ma duże znaczenie dla balansu gry i poczucia progresu.

Tabela 6.5 Przypadek testowy ulepszania wieży

ID	**TC005**
Tytuł	Ulepszenie istniejącej wieży i weryfikacja wzrostu statystyk.
Warunki początkowe	Na mapie stoi wieża typu „Strzelec" na poziomie 1. Gracz posiada 100 złota, koszt ulepszenia wieży z poziomu 1 na 2 wynosi 75 złota.
Kroki testowe	1. Zaznacz wieżę poprzez kliknięcie LPM. 2. W wyskakującym panelu informatycznym znajduje się przycisk „Ulepsz" (lub podobny). 3. Naciśnij przycisk „Ulepsz". 4. Obserwuj zmianę parametrów wieży i stanu zasobów.

Oczekiwany rezultat	Poziom wieży zmienia się z 1 na 2, wskaźnik poziomu aktualizuje się na ekranie (np. „Lvl: 2"), parametry wieży zwiększają się. Zasięg: +20% względem bazowego Obrażenia: +30% względem bazowego Szybkostrzelność: +15% względem bazowego

Rys. 6.5. Ulepszenie istniejącej wieży.

Scenariusz TC006 (Tabela 6.6) bada zachowanie systemu w sytuacji braku wystarczających zasobów na ulepszenie. Celem testu jest upewnienie się, że gra nie pozwala na wykonanie operacji, jeśli gracz nie spełnia warunków kosztu, oraz że interfejs jasno komunikuje przyczynę braku możliwości ulepszenia. Jest to ważne zarówno dla poprawności logiki, jak i dla czytelności komunikacji z użytkownikiem.

Tabela 6.6 Przypadek testowy braku złota na ulepszenie

ID	**TC006**
Tytuł	Próba ulepszenia wieży przy niewystarczającym złocie
Warunki początkowe	Na mapie stoi wieża na poziomie 1. Gracz posiada tylko 50 złota, a koszt ulepszenia wynosi 75 złota.

Kroki testowe	1. Zaznacz wieżę. 2. Spróbuj nacisnąć przycisk ulepszenia .
Oczekiwany rezultat	Przycisk „Ulepsz" pozostaje nieaktywny (wyszarzony/disabled), pojawia się komunikat tekstowy: „Za mało złota", poziom wieży nie zmienia się, zasoby gracza pozostają bez zmian (10 złota). Gra nie przerwała działania.

Rys. 6.6. Próba ulepszenia wieży przy niewystarczającym złocie.

Ostatni scenariusz w tym zestawie dotyczy obsługi zakończenia fali przeciwników. Test weryfikuje, czy po zniszczeniu wszystkich wrogów licznik fali zwiększa się a gra

reaguje zgodnie z założeniami, umożliwia przejście do kolejnej fali. Funkcjonalność ta zamyka podstawową pętlę gry i wpływa na poczucie postępu w kampanii.

Tabela 6.7 Przypadek testowy zakończenia fali przeciwników

ID	**TC007**
Tytuł	Zakończenie fali po zniszczeniu wszystkich przeciwników.
Warunki początkowe	Poziom kampanii jest uruchomiony, aktywna jest pierwsza fala . Na mapie stoi co najmniej 1–2 wieże w strategie sposób rozmieszczone.
Kroki testowe	1. Pozwól wieżom automatycznie zniszczyć wszystkich przeciwników z bieżącej fali.
Oczekiwany rezultat	Licznik fali zmienia się z 0/4 na 1/4

Rys. 6.7. Zakończenie fali po zniszczeniu wszystkich przeciwników.

7. Charakterystyka jednostek w grze

W grze występują trzy rodzaje wież oraz trzy typy przeciwników. Każda jednostka posiada unikalne właściwości, które wpływają na strategię rozgrywki. Poniżej przedstawiono szczegółową charakterystykę wszystkich jednostek wraz z ich statystykami bazowymi oraz sposobem skalowania w trakcie rozgrywki.

7.1 Typy wież

Wieże są podstawowymi jednostkami obronnymi w grze. Gracz może je stawiać w wyznaczonych miejscach na mapie, a następnie ulepszać w miarę gromadzenia zasobów. Każda wieża posiada trzy parametry bojowe: zasięg, obrażenia oraz szybkostrzelność. Po ulepszeniu wszystkie parametry zwiększają się według stałego wzoru: zasięg +20%, obrażenia +30%, szybkostrzelność +15% względem wartości bazowej dla każdego poziomu.

7.1.1 Strzelec (Sniper)

Jest to podstawowa wieża o zbalansowanych parametrach. Jej główną zaletą jest zdolność do przebijania pancerza przeciwników (armor_pierce = true), co czyni ją szczególnie skuteczną przeciwko opancerzonym wrogom. Strzelec posiada największy zasięg spośród wszystkich wież.

Rys. 7.1. Wieża Strzelec.

Statystyki bazowe:

- Zasięg: 150
- Obrażenia: 20
- Szybkostrzelność: 1.0 strzałów na sekundę
- Koszt budowy: 50 złota
- Koszt ulepszenia z poziomu 1 na 2: 80 złota
- Koszt ulepszenia z poziomu 2 na 3: 110 złota

7.1.2 AOE (obszarowy)

Wieża AOE zadaje obrażenia obszarowe, atakując jednocześnie wszystkich przeciwników znajdujących się w jej zasięgu. Charakteryzuje się niską szybkostrzelnością, ale wysokimi obrażeniami. Jej statystyki różnią się w zależności od poziomu kampanii – w poziom 3 otrzymuje wzmocnioną wersję.

Rys. 7.2. Wieża AOE.

Statystyki bazowe (level1–level2):

- Zasięg: 120
- Obrażenia: 30

  Szybkostrzelność: 0.5 strzałów na sekundę (jeden strzał co 2 sekundy)
- Koszt budowy: 75 złota

Statystyki bazowe (level3):

- Zasięg: 120
- Obrażenia: 50
- Szybkostrzelność: 0.4 strzałów na sekundę
- Koszt budowy: 75 złota

7.1.3 MG (szybkostrzelny)

Wieża MG cechuje się bardzo wysoką szybkostrzelnością przy niskich obrażeniach pojedynczego pocisku. Doskonale sprawdza się przeciwko szybkim i słabo opancerzonym przeciwnikom, gdzie kluczowa jest częstotliwość trafień.

Rys. 7.3. Wieża MG.

Statystyki bazowe:

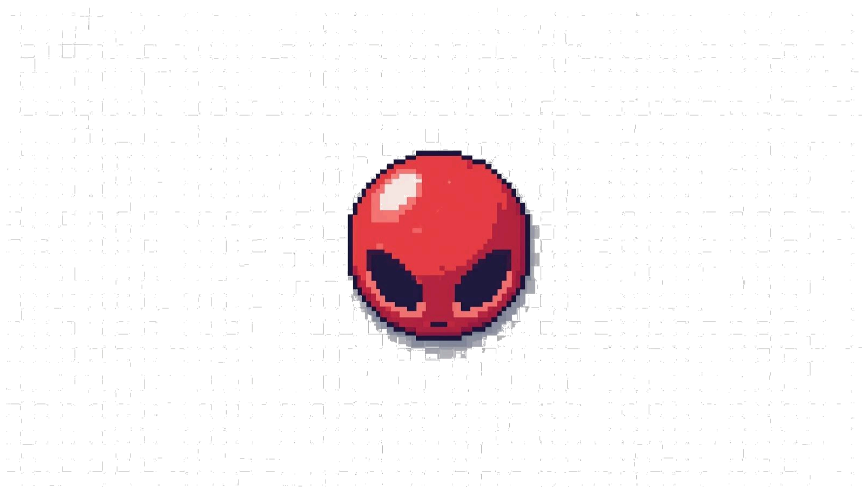
- Zasięg: 130
- Obrażenia: 8
- Szybkostrzelność: 3.0 strzałów na sekundę
- Koszt budowy: 40 złota
- Koszt ulepszenia z poziomu 1 na 2: 70 złota
- Koszt ulepszenia z poziomu 2 na 3: 100 złota

7.2 Typy przeciwników

Przeciwnicy w grze są podzieleni na trzy typy, różniące się parametrami bojowymi, wyglądem oraz zachowaniem. Każdy typ został zaprojektowany tak, aby wymusić na graczu stosowanie zróżnicowanych strategii obrony.

7.2.1 Normalny

Podstawowy typ przeciwnika, najczęściej występujący w grze. Charakteryzuje się zbalansowanymi parametrami i brakiem specjalnych właściwości. Jego kolor to czerwony.



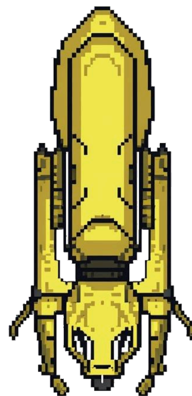
Rys. 7.4. Wróg normalny.

Statystyki bazowe:

- Punkty życia: 80
- Prędkość: 60
- Wartość po pokonaniu: 5 złota
- Pancerz: 0%

7.2.2 Szybki

Przeciwnik o wysokiej prędkości poruszania się, ale niskiej liczbie punktów życia. Stanowi zagrożenie szczególnie na długich, prostych odcinkach ścieżki, gdzie może szybko dotrzeć do bazy. Jego kolor to żółty.



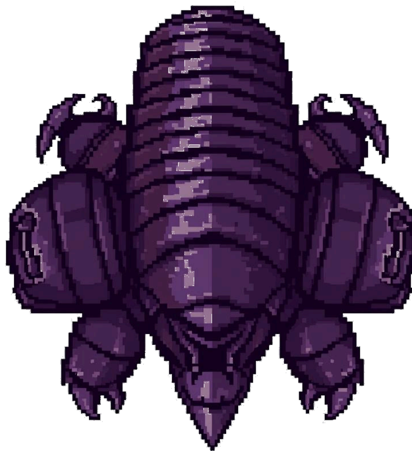
Rys. 7.5. Wróg szybki.

Statystyki bazowe:

- Punkty życia: 30
- Prędkość: 110
- Wartość po pokonaniu: 7 złota
- Pancerz: 0%

7.2.3 Opancerzony

Najwolniejszy, ale najbardziej wytrzymały typ przeciwnika. Posiada wysoki pancerz, który redukuje otrzymywane obrażenia o 35%. Jest odporny na ataki wież nieposiadających cechy armor_pierce. Jego kolor to fioletowy.



Rys. 7.6. Wróg opancerzony.

Statystyki bazowe:

- Punkty życia: 180
- Prędkość: 40
- Wartość po pokonaniu: 12 złota
- Pancerz: 35%

7.3 Skalowanie trudności w trybie endless

W trybie endless parametry przeciwników są dynamicznie skalowane wraz z numerem fali. Zastosowano następujące wzory:

- Liczba przeciwników w fali: 5 + (current_wave_index * 2)
- Mnożnik punktów życia: 1.0 + (0.08 * current_wave_index)
- Mnożnik prędkości: 1.0 + (0.03 * current_wave_index)

- Nagroda za falę: 3 + current_wave_index

Dodatkowo po piątej fali aktywowana zostaje druga ścieżka, co zmusza gracza do rozbudowy obrony na dwóch frontach jednocześnie. Punkty przyznawane są według wzoru: score = current_wave * 100.

7.4 Zestawienie statystyk

Tabela 7.1. Porównanie statystyk wież na poziomach 1–3

Wieża	Poziom	Zasięg	Obrażenia	Szybkostrzelność
Strzelec	1	150	20	1.00/s
Strzelec	2	180	26	1.15/s
Strzelec	3	210	32	1.30/s
AOE (kampania poziom 1–2)	1	120	30	0.50/s
AOE (kampania poziom 1–2)	2	144	39	0.58/s
AOE (kampania poziom 1–2)	3	168	48	0.65/s
AOE (kampania poziom 3)	1	120	50	0.40/s
AOE (kampania poziom 3)	2	144	65	0.46/s
AOE (kampania poziom 3)	3	168	80	0.52/s

MG	1	130	8	3.00/s
MG	2	156	10	3.45/s
MG	3	182	12	3.90/s

Tabela 7.2. Porównanie typów przeciwników

Typ	HP	Prędkość	Wartość	Pancerz	Kolor
Normalny	80	60	5	0%	Czerwony
Szybki	30	110	7	0%	Żółty
Opancerzony	180	40	12	35%	Fioletowy

8. Podsumowanie i wnioski

8.1. Osiągnięte cele pracy

Niniejsza praca inżynierska miała na celu zaprojektowanie i zaimplementowanie gry typu Tower Defense w silniku Godot 4.x z wykorzystaniem języka GDScript. Wszystkie założone cele zostały w pełni zrealizowane - powstała kompletna aplikacja oferująca tryb kampanii oraz tryb endless, wraz z pełnym systemem wież, przeciwników, fal oraz zarządzania zasobami.

Gra obejmuje kluczowe mechaniki, takie jak ruch przeciwników po ścieżce, automatyczny atak wież, generowanie fal o rosnącej trudności, budowę i ulepszanie jednostek obronnych oraz intuicyjny interfejs użytkownika z mechanizmem zapisu najlepszych wyników. Zastosowana modularna architektura kodu umożliwia łatwą rozbudowę gry o nowe elementy.

8.2. Spełnienie założeń projektowych

Zaimplementowano wszystkie wymagania funkcjonalne i niefunkcjonalne. Gra działa płynnie na sprzęcie o przeciętnej wydajności, a system zapisu poprawnie przechowuje najlepszy wynik w trybie endless. Jednocześnie struktura danych została zaprojektowana w sposób umożliwiający przyszłe rozszerzenie o zapis postępów kampanii.

Zastosowana grafika w stylu pixel-art 2D, częściowo wspomagana narzędziami AI, zapewnia spójny i estetyczny wygląd aplikacji. Silnik Godot okazał się trafnym wyborem - intuicyjny edytor, wydajny język GDScript oraz system sygnałów znacząco przyspieszyły proces tworzenia gry. Projekt stanowi podsumowanie umiejętności zdobytych w trakcie studiów.

8.3. Napotkane trudności

Największe wyzwania dotyczyły optymalizacji wydajności przy dużej liczbie obiektów jednocześnie obecnych na ekranie oraz implementacji systemu zapisu w formacie JSON.

8.4. Wnioski

Realizacja pracy zakończyła się sukcesem, czego efektem jest w pełni funkcjonalna gra typu Tower Defense. Projekt potwierdził, że silnik Godot jest skutecznym narzędziem do tworzenia gier 2D, a także umożliwił praktyczne zastosowanie wiedzy z zakresu programowania obiektowego, projektowania interfejsów użytkownika oraz optymalizacji aplikacji czasu rzeczywistego.

Najcenniejszym doświadczeniem było przejście przez pełny cykl wytwarzania oprogramowania - od analizy wymagań, przez implementację, aż po testowanie i przygotowanie dokumentacji.

8.5. Kierunki dalszego rozwoju

W przyszłości gra może zostać rozszerzona o następujące elementy:

- nowe typy wież, np. spowalniające przeciwników lub zadające obrażenia w czasie,
- rozszerzenie systemu zapisu o pełny postęp w kampanii,
- proceduralnie generowane mapy zwiększające regrywalność,
- system osiągnięć (achievementów) motywujący do dalszej gry,
- port na platformy mobilne (Android/iOS),
- tryb kooperacyjny lub sieciowy,
- edytor poziomów z możliwością udostępniania map przez Steam Workshop.

9. Bibliografia

[1] Godot Engine – Official Website, https://godotengine.org/

[2] Godot Documentation – Architecture Overview, https://docs.godotengine.org/

[3] Godot Documentation – GDScript Basics,:
https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/

[4] OpenGameArt.org – Free Assets for Games, https://opengameart.org/

[5] Godot Documentation – Nodes and Scenes,
https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html

[6] Godot Documentation – Signals,
https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

[7] Godot Documentation – Physics Introduction,
https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html

[8] Godot Documentation – CharacterBody2D,
https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html

[9] Godot Documentation – TileMap and Grid-based Maps,
https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html

[10] Godot Documentation – Path2D and PathFollow2D,
https://docs.godotengine.org/en/stable/classes/class_path2d.html
https://docs.godotengine.org/en/stable/classes/class_pathfollow2d.html

[11] Godot Documentation – Performance and Optimization,
https://docs.godotengine.org/en/stable/tutorials/performance/index.html

[12] Godot Documentation – PackedScene,
https://docs.godotengine.org/en/stable/classes/class_packedscene.html

10. Spis rysunków

Rys. 5.1. Ekran główny gry
Rys. 5.2. Mapa kampanii
Rys. 5.3. Tryb Endless
Rys. 5.4. Poziom 1 trybu kampanii
Rys. 5.5. Menu pauzy
Rys. 6.1. Menu poziomu 1
Rys. 6.2. Aktywna wieża
Rys. 6.3. Budowanie wieży w niedozwolonym miejscu
Rys. 6.4. Automatyczne strzelanie wieży do przeciwnika w zasięgu
Rys. 6.5. Ulepszenie istniejącej wieży
Rys. 6.6. Próba ulepszenia wieży przy niewystarczającym złocie
Rys. 6.7. Zakończenie fali po zniszczeniu wszystkich przeciwników
Rys. 7.1. Wieża Strzelec
Rys. 7.2. Wieża AOE
Rys. 7.3. Wieża MG
Rys. 7.4. Wróg normalny
Rys. 7.5. Wróg szybki
Rys. 7.6. Wróg opancerzony

11. Spis tabel

Tabela 6.1. Przypadek testowy TC001 - uruchomienie poziomu

Tabela 6.2. Przypadek testowy TC002 - budowa wieży

Tabela 6.3. Przypadek testowy TC003 - budowa w niedozwolonym miejscu

Tabela 6.4. Przypadek testowy TC004 - automatyczny atak

Tabela 6.5. Przypadek testowy TC005 - ulepszanie wieży

Tabela 6.6. Przypadek testowy TC006 - brak złota

Tabela 6.7. Przypadek testowy TC007 - zakończenie fali

Tabela 7.1. Porównanie statystyk wież na poziomach 1-3

Tabela 7.2. Porównanie typów przeciwników

12. Spis listingów

Listing 4.1. Fragment skryptu Enemy.gd - ruch i obrażenia.

Listing 4.2. Fragment skryptu Tower.gd - wyszukiwanie celów.

Listing 4.3. Fragment skryptu Tower.gd - system ulepszeń.

Listing 4.4. Fragment skryptu WaveManager.gd - definicja fal.

Listing 4.5. Fragment skryptu SaveManager.gd - zapis i odczyt.

Listing 4.6. Fragment skryptu Bullet.gd - ruch pocisku.

Listing 4.7. Fragment skryptu poziomu - system budowy wież.

Listing 4.8. Fragment skryptu endless_level.gd - tryb endless.