Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka Techniczna (ITE)**
Specjalność: **IGM**

PRACA DYPLOMOWA
INŻYNIERSKA

Projekt i implementacja wieloosobowej gry on-line z użyciem silnika Godot

Józef Bossowski

Opiekun pracy
Dr inż. Tomasz Walkowiak

Słowa kluczowe: Godot, gra sieciowa

WROCŁAW 2022

STRESZCZENIE

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

ABSTRACT

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

SPIS TREŚCI

WSTĘP

CEL PRACY

Celem niniejszej pracy dyplomowej jest zaprojektowanie oraz zaprogramowanie sieciowej gry wieloosobowej.

Gra będzie umożliwiać połączenie od dwóch do sześciu osób w sieci lokalnej. Aplikacja zostanie przygotowana na komputery PC z systemem Windows 10. Gra będzie przygotowana z myślą o sterowaniu klawiaturą i myszą. Serwerem gry będzie komputer jednego z graczy, nazywanego również *hostem*. Zarówno mechaniki gry jak i system sieciowy zostaną zaimplementowane z wykorzystaniem silnika Godot. [12] W celu zapewnienia płynnej i komfortowej rozgrywki zostaną zastosowane techniki predykcji klienta. Oprogramowanie i interfejs użytkownika zostaną przygotowane w języku angielskim.

Początkowo przygotowane zostaną:

— *Koncepcja gry* - projekt mechanik gry, projekt wizualny, elementy *game designu*;
— *Projekt aplikacji* - projekt interfejsu użytkownika, plan przygotowania systemu sieciowego oraz działanie mechanik;
— *Prototyp aplikacji* - początkowa, okrojona wersja gry, przygotowana w celu zapoznania się z działaniem silnika oraz w celu określenia potencjalnych problemów z późniejszą implementacją.

Powyższe elementy będą przygotowywane równolegle, aby wiedza pozyskana w czasie rozwijania jednego mogła możliwie najlepiej wesprzeć pozostałe.

TECHNOLOGIE

W rozwoju projektu wykorzystane zostaną poniższe technologie:

— *Godot* - Silnik gier; implementacja mechanik gry, silnik fizyczny, komunikacja sieciowa, wysokopoziomowy interfejs sieciowy;
— *Git*, *GitHub* - System kontroli wersji;
— *Blender* - Modelowanie 3D, przygotowanie materiałów;
— *Trello* - Planowanie, harmonogram pracy;

1. ANALIZA TECHNOLOGII

1.1. SILNIK GIER

Silnik gier to oprogramowanie pozwalające tworzyć gry komputerowe w prosty i wydajny sposób. Silnik umożliwia tworzenie oprogramowania bez potrzeby przygotowywania wszystkich systemów od podstaw.

Do podstawowych zadań silnika gier należą [21]:

— Tworzenie i zarządzanie aplikacją - Zawarte są tu takie elementy jak okno aplikacji, komunikacja z interfejsami wejścia-wyjścia, zarządzanie wątkami itp.
— Generowanie i wyświetlanie grafiki - Programista może korzystać z abstrakcji poprzez korzystanie z zasobów, komponentów lub scen. Silnik zapewnia renderowanie grafiki dwuwymiarowej oraz rzutowanie scen trójwymiarowych. Zwykle w tym celu wyręcza programistę w komunikacji z bibliotekami niższego poziomu.
— Zarządzanie dźwiękiem - Silniki umożliwiają proste wprowadzenie dźwięków do gry bez potrzeby bezpośredniego interfejsowania z kartą graficzną lub abstrakcją systemu operacyjnego.
— Silnik fizyczny - Wiele silników gier umożliwia korzystanie z wbudowanych silników fizyki brył sztywnych w celu m.in. wykrywania kolizji.
— Dodatkowe moduły - np. nawigacja, sztuczna inteligencja, łączność sieciowa.
— Dostarczenie wizualnych narzędzi - Większość silników gier umożliwia wykonywanie podstawowych zadań bez edycji kodu, poprzez interakcję z interfejsem graficznym.
— Zapewnienie jednolitego środowiska programistycznego (ang. *Integrated Development Environment* - IDE) - Wiele silników pozwala na zarządzanie zasobami oraz ich edycją wewnątrz jednego, spójnego narzędzia. Nadal możliwe jest wykorzystanie zewnętrznych programów, przykładowo do przygotowania grafiki, jednak podstawowe potrzeby często są zapewnione przez środowisko silnika.

1.1.1. Popularne silniki gier

Na rynku dostępnych jest wiele silników gier. Można podzielić je na tworzone z myślą o konkretnej grze oraz silniki ogólnego przeznaczenia[11]. Te pierwsze najczęściej tworzone są przez zespoły jedynie do użytku wewnętrznego. Z tego względu mają często bardziej ograniczone możliwości, są jednak lepiej zoptymalizowane oraz, ze względu na zupełną kontrolę zespołu bądź firmy, mogą byc dostosowywane do konkretnych potrzeb.

Silniki ogólnego przeznaczenia tworzone są z myślą o wykorzystaniu w wielu projektach o różnych cechach i potrzebach. Poniżej opisane zostały najpopularniejsze silniki dostępne na rynku.

1.1.1.1. Unity

Unity jest silnikiem gier ogólnego przeznaczenia darmowym dla projektów zarabiających poniżej astu tysięcy dolarów rocznie[22][16]. Jest jednym z najpopularniejszych silników, wykorzystywanym w niemalże połowie projektów dostępnych na platformie itch.io[15]. Wykorzystuje język programowania C# i jest mocno związany z środowiskiem .NET. Ze względu na dużą popularność dostępnych jest wiele materiałów edukacyjnych, zasobów oraz rozszerzeń tworzonych przez społeczność fanów. Unity posiada szeroki zbiór narzędzi do tworzenia złożonych gier zarówno 2D jak i 3D. Jest prosty w obsłudze, jednak wizualnie mniej zaawansowany niż Unreal.

1.1.1.2. Unreal Engine

Silnik Unreal służy głównie do tworzenia gier 3D o wysokiej złożoności grafiki z naciskiem na realistyczność[10][5]. Słabiej niż Unity nadaje się do tworzenia mniejszych gier np. na platformy mobilne[6]. W związku z tym, szczególnie w ostatnich latach[9], częściej wykorzystywany jest w większych firmach niż małych niezależnych zespołach. Silnik jest darmowy, jednak dla projektów o zarobkach powyżej miliona dolarów należna jest opłata w wysokości 5% przychodów. Programowanie w tym silniku możliwe jest z wykorzystaniem języka C++ lub narzędzia programowania wizualnego „Blueprints".

1.1.1.3. GameMaker: Studio

GameMaker jest silnikiem przeznaczonym do tworzenia gier dwuwymiarowych. Pozwala na tworzenie gier z wykorzystaniem języka GML („GameMaker Language"). Ze względu na ograniczone możliwości jest o wiele mniej popularny niż silniki wymienione powyżej, jednak mimo to wielu twórców niezależnych z niego korzysta. Proste narzędzie programowania wizualnego oraz niskie wymagania sprzętowe powodują, że jest jednym z najpopularniejszych silników na platformie itch.io[15].

1.2. SILNIK GODOT

Silnik Godot[12] jest darmowym silnikiem gier służącym do rozwoju gier 2D oraz 3D. Jest rozwijany od roku 2007, z pierwszą pełną wersją dostępną od 2014 roku. W tym samym roku projekt został udostępniony otwartoźródłowo na platformie GitHub[1], z licencją MIT. Od tego czasu Godot jest nieustannie rozwijany, czego efektem jest wiele dostępnych wersji oprogramowania.

W momencie tworzenia projektu najnowsza dostępna wersja silnika to beta wersji 4.0. Dodaje ona szereg usprawnień względem wersji 3.x, w szczególności te dotyczące wydajności i renderowania grafiki 3D. Jest ona jednak dostępna we wczesnej fazie rozwoju, co może powodować problemy ze stabilnością oraz dostępnością rozszerzeń. Dla wersji 4.0 jest również niewiele dostępnych materiałów edukacyjnych. Z tych względów zdecydowano o wykorzystaniu najnowszej (w momencie rozpoczynania projektu) stabilnej wersji - 3.5[13].

1.2.1. Język GDScript

Godot umożliwia pisanie kodu z wykorzystaniem kilku języków. Przez twórców projektu wspierane są języki C# i C++, oraz interfejs programowania wizualnego. Z wykorzystaniem rozszerzeń przygotowanych przez społeczność możliwe jest dodanie wsparcia dla wielu innych języków. Domyślnym językiem dla Godota jest jednak ich własny język GDScript.

Składnia GDScripta jest oparta o składnię Pythona - bloki kodu oddzielone są wcięciami, a nie nawiasami. Te dwa języki dzielą również wiele podstawowych słów kluczowych.

1.2.2. Drzewo gry

Podstawową abstrakcją, z jaką budowane są gry w Godocie jest ich podział na drzewo węzłów (ang. *node*). Węzły mogą być grupowane w sceny (ang. *scenes*) w celu umożliwienia ponownego ich wykorzystywania. Sceny również reprezentowane są jako drzewo węzłów z których się składają.

Godot dostarcza wiele rodzajów węzłów, umożliwiających wprowadzenie do gry najważniejszych funkcjonalności.

`Node` Podstawowy, pusty węzeł. Najczęściej wykorzystywany jako kontener na inne węzły. Jest również bazą dla wspierających węzłów takich jak `Timer`, `Tween` czy `AnimationPlayer`.

`Spatial` Węzeł przestrzenny, będący bazą wszystkich węzłów wykorzystywanych w grach 3D, m.in. `Camera`, `PhysicsBody`, `CollisionShape`. Zawiera podstawowe informacje na temat położenia w przestrzeni.

`Control` Węzeł będący bazą węzłów interfejsu użytkownika takich jak `Button`, `Label` czy `ColorRect`.

`Node2D` Podstawowy węzeł dla elementów gier dwuwymiarowych. Zawiera w sobie przede wszystkim informacje o położeniu na płaszczyźnie.

W czasie rozgrywki silnik tworzy drzewo z korzeniem nazwanym `root`. Jako jego dziecko inicjowana jest scena główna, określona w ustawieniach projektu. Ponadto inicjowane są sceny statyczne, które również mogą być określone w ustawieniach projektu jako „Autoładowane”. Takie sceny pozwalają na globalny dostęp z innych miejsc drzewa, umożliwiają przechowywanie danych dostępnych między zmianami scen oraz pozwalają na

implementację wzorca projektowego singletonu[4]. Jednak, jak podkreślono w dokumentacji, samo autoładowanie sceny nie tworzy singletonu, ponieważ możliwe jest ponowne instancjonowanie scen autoładowanych.

Węzłem w drzewie sceny (a co za tym idzie, również w drzewie gry) może być również instancja innej sceny. Jest ona widoczna jako pojedynczy węzeł, mimo, że sama w sobie również jest drzewem. Tak inicjowane sceny również mogą mieć przypisane dzieci.

W czasie działania programu możliwe jest przełączenie sceny głównej programistycznej, poprzez wywołanie na drzewie metody `change_scene`, której argumentem jest ścieżka do pliku z nową sceną. Obiekt będący instancją poprzedniej sceny zostaje wtedy usunięty i zastąpiony instancją nowej sceny.

Drzewo udostępnia również szereg użytecznych metod. Przykładem takiej metody może być `notification` - metoda rozsyłająca powiadomienie do wszystkich obiektów aktualnie w drzewie. Jest ona przydatna przykładowo w momencie zamykania aplikacji - pozwala to zakończyć niezbędne procesy lub wyświetlić prośbę o potwierdzenie zamknięcia.

1.2.3. Zasoby

Zasoby (ang. *Resources*) są podstawowym sposobem na przechowywanie i dzielenie danych w silniku Godot. Zasoby dzielą się na zewnętrzne, będące plikami zapisanymi na dysku, oraz wbudowane, zapisane jako element sceny. W sytuacji gdy wiele węzłów lub zasobów korzysta z tego samego zasobu zmiany w nim będą widoczne dla każdego użytkownika. Jest to szczególnie istotne w przypadku materiałów. Aby zmienić materiał dla jednego modelu, nie wprowadzając zmian we wszystkich, należy „ulokalnić" go do wybranej sceny.

1.2.4. Renderowanie grafiki

Godot udostępnia dwa silniki graficzne: GLES2, oparty na OpenGL 2.1 oraz GLES3 oparty na OpenGL 3.3. Ze względu na nowocześniejszy silnik bazowy, GLES3 dostarcza funkcje niedostępne w GLES2, takie jak akceleracja GPU dla animacji cząsteczkowych. Ponadto zapewnia on lepszą wydajność. GLES2 jest kompatybilny z większą liczbą, szczególnie starszych, urządzeń.

Do rozwoju projektu zostanie zastosowany GLES3.

1.2.5. Sygnały

W Godocie wbudowana jest implementacja wzorca projektowego obserwatora [20], nazwana sygnałami. Węzły wysyłają sygnały, aby poinformować węzły nasłuchujące o zajściu jakiegoś wydarzenia. Wraz z sygnałem mogą zostać wysłane argumenty, których może użyć słuchacz

1.2.6. Edytor

Godot udostępnia kompleksowy edytor pozwalający na prostą edycję elementów projektu. Umożliwia edycję scen, grafiki, parametrów węzłów, ale również takich aspektów jak animacje i sygnały.

1.2.7. System sieciowy

W silniku Godot dostępnych jest wiele sposobów na wprowadzenie interakcji sieciowych do tworzonych gier. Możliwe jest tu korzystanie bezpośrednio z protokołów niskiego poziomu: TCP i UDP. Udostępnione są również interfejsy protokołów wyższego poziomu: SSL i HTTP. Najprostszym sposobem jest korzystanie z wysokopoziomowego API (ang. *Aplication Programming Interface*, Interfejs programowania aplikacji). Pozbawia ono twórcę dokładnej kontroli nad pakietami, wprowadzając jednak wiele abstrakcji upraszczających pracę.

Sieciowe API Godota oparte jest na zmodyfikowanym protokole UDP. Daje to możliwość komunikacji zawodnej dla szybkiej komunikacji lub komunikacji niezawodnej dla pewności otrzymania pakietów przez adresatów.

Podstawową klasą niezbędną do wprowadzenia funkcji sieciowych w Godocie jest `NetworkedMultiplayerENet`. Jest to implementacja interfejsu `PacketPeer` korzystająca z biblioteki ENet[14]. Instancja tej klasy zapisywana jest w drzewie gry jako `peer`. Zarządza ona komunikacją sieciową. Może zostać zainicjowana jako serwer lub klient. W roli serwera obiekt nasłuchuje komunikacji na podanym porcie oraz pod podanym interfejsem IP. Domyślnie używana jest "dzika karta" (ang. *wild card*) - serwer może nasłuchiwać na wszystkich dostępnych interfejsach IP. Możliwe jest również określenie maksymalnej liczby połączonych klientów w zakresie do 4095. W rzeczywistości jednak najpewniej możliwa do utrzymania będzie znacznie mniejsza liczba połączeń. W roli klienta, obiekt nawiązuje komunikację z serwerem nasłuchującym na podanym porcie pod podanym adresem.

Komunikacja na wyższym poziomie odbywa się pomiędzy odpowiadającymi sobie węzłami drzew gry u klientów i na serwerze. Tworząc węzły, które miałyby komunikować się między sobą należy nazwać je tak samo na wszystkich urządzeniach. Wszystkie odpowiadające sobie węzły muszą być pod kontrolą jednego z komputerów. Taka relacja nazwana jest *master-puppet* (ang. mistrz-marionetka). Mistrzem nazywany jest węzeł znajdujący się na kontrolującym go komputerze, podczas gdy marionetki to odpowiadające mu węzły na pozostałych komputerach biorących udział w komunikacji.

W czasie działania programu wykorzystywane są trzy mechanizmy komunikacji:

— `rpc` (ang. *Remote Procedure Call*, zdalne wywołanie procedury) - Pozwala wywołać metodę na odpowiadającym węźle innego komputera. Metody przekazywane są poprzez nazwę metody i zestaw argumentów np. `rpc("start_game")`. Aby móc wywołać

metodę zdalnie musi być ona zdefiniowana z jednym ze słów kluczowych określających metody zdalne.

— `rset` (ang. *Remote Set*, Ustaw Zdalnie) - Pozwala na zmianę wartości zmiennej na innych komputerach np. `rset("position", my_position)`. Podobnie jak metody - zmienne ustawiane w ten sposób muszą byc zdefiniowane z wykorzystaniem odpowiednich słów kluczowych.

— Wbudowane sygnały - Ustawiony w drzewie gry *network peer* emituje sygnały w przypadku zmiany statusu połączenia, np. gdy z serwerem połączy się nowy klient lub gdy połączenie z serwerem zostanie utracone.

Udostępnione są również zmodyfikowane wersje `rpc` i `rset`, korzystające z połączenia zawodnego (`rpc_unreliable`, `rset_unreliable`), pozwalające na przesyłanie danych do konkretnego komputera (`rpc_id`, `rset_id`) lub obu (`rpc_unreliable_id`, `rset_unreliable_id`). Ponadto metody oraz zmienne wykorzystywane w komunikacji muszą być tworzone z wykorzystaniem odpowiednich słów kluczowych:

`remote` Umożliwia wykorzystanie metody lub zmiennej do komunikacji z dowolnego połączonego urządzenia.

`puppet` Umożliwia wysyłanie poleceń jedynie z mistrza do marionetek.

`master` Umożliwia wysyłanie poleceń jedynie od marionetek do mistrza.

Ponadto dodając końcówkę `sync` do powyższych słów kluczowych możliwe jest wywołanie zapytania rpc lub rset również lokalnie, bez konieczności ręcznego wywoływania funkcji.

1.3. GUT

GUT (*Godot Unit Testing*)[8] jest zestawem narzędzi pozwalających na pisanie i uruchamianie testów jednostkowych w środowisku Godot. Zawiera narzędzia linii komend, rozszerzenie do edytora oraz niezbędne klasy i metody testujące.

Testy w GUT pisane są w języku GDScript. Należy umieszczać je w klasach rozszerzających `GutTest`. W ustawieniach dodatku określić można foldery oraz schematy nazewnictwa plików testowych. Nazwy funkcji testowych muszą rozpoczynać się od „`test_`”. Mogą one być również pogrupowane na podklasy.

Klasa rozszerzająca `GutTest` może korzystać z metod zarządzających testami - uruchamianych przed lub po każdym lub wszystkich testach (np. `before_each` lub `before_all`).

Do przeprowadzania testów udostępniony jest szeroki zestaw asercji, pozwalających sprawdzić zgodność otrzymanych wartości i zdarzeń z oczekiwaniami.

1.4. BLENDER

Blender jest darmowym, otwartoźródłowym programem do tworzenia grafiki 3D[7][17]. Pozwala tworzyć i edytować trójwymiarowe modele i sceny, teksturować je oraz tworzyć materiały i shadery. Ponadto umożliwia tworzenie animacji a nawet prostych gier. Został wykorzystany do stworzenia modeli w tym projekcie ze względu na jego popularność, dostępność materiałów edukacyjnych oraz licencję GNU GPL pozwalającą na darmowe korzystanie w każdym celu.

2. ANALIZA RYNKU

2.1. PODOBNE ROZWIĄZANIA

2.1.1. ROUNDS

2.1.2. Boomerang Foo

2.1.3. World of Tanks

2.1.4. Wii Play - Tanks

3. KONCEPCJA GRY

[11] W projektowanej grze gracze sterują czołgami, mogą do siebie strzelać, a w czasie rozgrywki będą dostawali ulepszenia, wzmacniające ich możliwości bojowe, zarówno pasywne i aktywne. W tym rozdziale szczwegółowo opisane zostaną mechaniki gry oraz wymagania z dziedziny *game designu*.

3.1. PROJEKT MECHANIK

Do zaimplementowania zaplanowane zostały następujące mechaniki:

— Sterowanie gracza,
— Strzelanie,
— System zdrowia/życia,
— System rund,
— Karty ulepszeń,
— Atrybuty pasywne i aktywne,
— *User experience*.

Zostaną one szczegółowo opisane w kolejnych sekcjach.

3.1.1. Sterowanie gracza

W przypadku większości gier, w których gracz steruje pewną postacią (awatarem), istnieją założenia gracza związane z tym jak takie sterowanie implementowane jest w innych produkcjach. W projektowanej grze awatarem gracza jest czołg, wykorzystany zostanie więc schemat sterowania zwany "sterowaniem czołgowym" (ang. *tank controls*). [2] Poza potencjalnie intuicyjną interpretacją sterowania przez graczy, taki schemat charakteryzuje się również przewidywalnym zachowaniem awatara w przypadku zmiany perspektywy kamery.

W tym schemacie sterowania awatar porusza się w kierunku określonym względem jego własnej orientacji, w przeciwieństwie do wielu współczesnych tytułów, w których postać porusza się w kierunku względnym do kamery. [przygotować schemat na rysunku]

Gra przygotowywana jest z myślą o sterowaniu klawiaturą i myszą, schemat sterowania zostanie więc zmapowany na odpowiednie przyciski i gesty związane z charakterystyką tych urządzeń. [tabela ze schematem sterowania]

3.1.2. Strzelanie

Podstawową akcją wykonywaną przez gracza będzie strzelanie do przeciwników.

Gracz będzie celował przy pomocy myszy, lufa czołgu będzie obracała się w kierunku rzutu myszy na jej płaszczyznę w przestrzeni 3D. Taka metoda sterowania da dużą dokładność celowania. Jest również stosunkowo prosta w implementacji i intuicyjna dla gracza.

Na pociski gracza nie będzie działać fizyka - będą poruszały się jedynie w płaszczyźnie równoległej do ziemi. Pociski będą wchodzić w kontakt z innymi graczami oraz z obiektami umieszczonymi w poziomie.

Gracz będzie mógł strzelić z wykorzystaniem lewego przycisku myszy. Każde wciśnięcie przycisku odpowiadać będzie jednemu wystrzelonwmu pociskowi, choć to zachowanie może ulec zmianie w czasie rozgrywki [odniesienie do rozdziału z kartami?].

Postać gracza będzie miała ograniczoną pojemność "magazynku". Przeładowanie będzie trwało określony czas. W przeciwieństwie do wielu gier z podobną mechaniką, przeładowanie nie będzie konsumować pocisków z większej puli - gracz ma możliwość nieograniczonych przeładowań. Rolą tej mechaniki jest wymuszenie na graczach myślenia taktycznego oraz zarządzania zasobami.

Ograniczona maksymalna szybkość ataku zostanie wprowadzona aby uniemożliwić nazbyt szybkie oddawanie strzałów. „Szybkość ataku” jest określona jako minimalny czas pomiędzy kolejnymi strzałami.

Atrybuty związane ze strzelaniem zostały zebrane i opisane w tabeli 3.1.

3.1.3. System zdrowia i życia

System zdrowia zostanie wprowadzony w celu umożliwienia graczom eliminacji przeciwników.

Każdą rundę gracz rozpoczyna z punktami życia równymi ich maksymalnej wartości. W czasie rozgrywki, otrzymując obrażenia, ta wartość będzie zmniejszana. Gdy wartość ta osiągnie 0, gracz zostaje wyeliminowany z aktualnej rundy.

W czasie rozgrywki żywotność postaci będzie się zamoistnie zwiększać, jeżeli wystarczająco długo nie otrzymał obrażeń. Nie może ona jednak przekroczyć maksymalnej wartości.

Atrybuty postaci związane ze zdrowiem zostały zebrane i opisane w tabeli 3.1.

3.1.4. System rund

Gra będzie podzielona na rundy. Każda z rund trwa do momentu, w którym tylko jeden gracz nie będzie wyeliminowany. Ten gracz dostaje jeden punkt zwycięstwa, a następnie gracze przechodzą do kolejnej rundy. Pomiędzy rundami następuje faza wyboru kart

(opisana w rozdziale 3.1.6). Gra trwa do rundy, po której jeden z graczy osiągnie docelową liczbę punktów zwycięstwa.

Taki system rozgrywki prowadzi do różnej liczby rund dla różnej liczby graczy oraz dla różnej docelowej liczby punktów.

Minimalna liczba rund będzie miała miejsce w rozgrywce, w której w każdej rundzie wygrywa ten sam gracz. Maksymalna liczba rund odbędzie się w rozgrywce, w której każdy gracz wygrywa liczbę rund o jeden mniejszą niż docelowa, a następnie jeden z graczy wygrywa rundę, tym samym wygrywając grę.

W związku ze zwiększeniem liczby rund znacznie zwiększy się czas rozgrywki. Z tego względu dla większych liczb graczy sugerowane będą mniejsze cele punktowe, jednak nie zostaną one ograniczone domyślnie; decyzja na temat liczby docelowych punktów zwycięstwa zostanie pozostawiona hostowi, jednak jedynie w zakresie od 1 do 10.

3.1.5. Atrybuty pasywne i aktywne

W celu rezprezentacji różnych cech postaci gracza każdy z nich ma swoje *atrybuty*. Dzielą się one na pasywne i aktywne.

Atrybuty pasywne reprezentowane są jedynie jako wartości liczbowe, które wpływają na zwykłe akcje wykonywane przez gracza. Każda z postaci zapisuje atrybuty pasywne dotyczące jej samej oraz używanych przez nią pocisków. Każdy z graczy ma stały zbiór atrybutów pasywnych.

Atrybuty aktywne to wydarzenia wywoływane w momencie zaistnienia innego wydarzenia. W projektowanych mechanikach jedynym wydarzeniem inicjującym jest trafienie pociskiem innego gracza lub przeszkody, jednak możliwe jest wprowadzenie kolejnych w czasie rozwoju gry.

3.1.6. Karty ulepszeń

W czasie rozgrywki gracze będą otrzymywali ulepszenia wzmacniające ich możliwości bojowe. Te ulepszenia gracze wybierają w fazie wyboru kart, pomiędzy rundami.

Po zakończeniu rundy, każdy z graczy o najmniejszej liczbie punktów zwycięstwa otrzyma zestaw losowych kart, spośród których będzie musiał wybrać jedną. Wybrana karta zostaje zapisana dla tego gracza i od tej pory będzie działała na jego postać. Po wybraniu karty przez każdego z graczy następuje przejście do kolejnej rundy.

Ulepszenia nadawane są jedynie postaciom o najmniejszej liczbie punktów zwycięstwa w celu wyrównania szans pomiędzy graczami o różnym poziomie umiejętności. Taka technika nazywana jest *metodą gumki recepturki* (ang. *rubberbanding*)[3].

Każda z kart ulepszeń modyfikuje co najmniej jeden atrybut postaci - zwiększa lub zmniejsza wartość pasywną lub dodaje kolejny atrybut aktywny. Karta poprawi przynajmniej jedną wartość atrybutu, jednak możliwe jest, że pogorszy inne atrybuty aby zbalansować jej działanie. Wprowadza to również nieoczywistą decyzję do podjęcia dla gracza

Tabela 3.1. Opis atrybutów postaci i pocisku

Nazwa atrybutu	Kategoria	Opis
Maksymalna prędkość	Ruch	Z taką najwyższą prędkością może poruszać się postać gracza.
Maksymalna prędkość kątowa	Ruch	Z taką najwyższą prędkością kątową może obracać się postać gracza.
Maksymalna żywotność	Zdrowie	Tyle najwyżej punktów zdrowia może mieć postać.
Opóźnienie leczenia	Zdrowie	Tyle czasu należy odczekać od ostatnich otrzymanych obrażeń przed rozpoczęciem samoleczenia.
Okres leczenia	Zdrowie	Tyle czasu upływa między kolejnymi wydarzeniami leczenia.
Wartość leczenia	Zdrowie	O taką wartość zwiększone zostanie aktualne zdrowie postaci w każdym wydarzeniu leczenia.
Pojemność magazynku	Strzelanie	Tyle pocisków może wystrzelić gracz przed przeładowaniem.
Czas przeładowania	Strzelanie	Tyle czasu upływa pomiędzy początkiem a końcem przeładowania.
Szybkość ataku	Strzelanie	Tyle czasu należy odczekać pomiędzy kolejnymi strzałami.
Obrażenia	Atrybuty pocisku	Wartość o jaką zmniejszy się życie trafionej pociskiem postaci.
Szybkość pocisku	Atrybuty pocisku	Szybkość z jaką przemieszcza się pocisk.
Okres istnienia pocisku	Atrybuty pocisku	Czas, przez jaki istnieje pocisk. Po upływie tego czasu od utworzenia, pocisk jest usuwany.
Rozmiar	Atrybuty pocisku	Określa wymiary pocisku. Związany z obrażeniami. Skalowany jest zarówno model jak i figura kolizji.
Właściciel	Atrybuty pocisku	Gracz który wystrzelił pocisk.
Efekty przy trafieniu	Atrybuty pocisku	Wydarzenia wywoływane po trafieniu pociskiem w postać lub przeszkodę.

(przykładowo: *Czy warto wybrać kartę poprawiającą dwukrotnie zadawane obrażenia, ale obniżającą o połowę własną żywotność?*).

3.1.6.1. Początkowy zbiór kart

W celu zróżnicowania rozgrywek przygotowane zostanie 20 kart z możliwością dodania kolejnych w przyszłości. Planowane do zaimplementowania karty przedstawione zostały w tabeli 3.2.

3.1.7. User experience

W celu zapewnienia satysfakcjonującej rozgrywki zostaną zastosowane techniki poprawy doświadczenia graczy (*User Experience* - UX).

3.1.7.1. Buforowanie akcji

Buforowanie akcji odnosi się do techniki, w której polecenia wydawane przez gracza są zapisywane w przypadku gdy nie jest możliwe wykonanie ich natychmiast. Akcje wywoływane przez te polecenia zostają aktywowane w momencie gdy nastanie taka możliwość. [18]

W przygotowywanej grze ta technika zostanie wykorzystana dla akcji strzelania. Gdy postać gracza będzie w stanie przeładowywania lub oczekiwania na kolejny strzał a gracz wprowadzi polecenie strzału zostanie ono zapisane na kilka klatek i wykonane, jeżeli w tym czasie ta akcja zostanie odblokowana.

Zastosowanie tej techniki pozwoli uniknąć sytuacji, w której gracz wprowadzi polecenie tuż przed zmianą stanu. Bez buforowania akcji to polecenie zostałoby zignorowane, co może prowadzić do poczucia niesprawiedliwego potraktowania przez grę. Wprowadzenie buforowania sprawia, że gra *wydaje się* być bardziej sprawiedliwa poprzez wybaczanie drobnych błędów gracza.

3.1.7.2. Predykcje klienckie

W grze sieciowej nie sposób uniknąć opóźnień związanych z połączeniem, nawet gdy komputery graczy znajdują się w tej samej sieci. Przesyłając pozycję graczy przez sieć opóźnienie i utracone pakiety sprawiają, że postaci innych graczy wyglądają jakby "przeskakiwały". W związku z tym należy wprowadzić usprawnienia sprawiające, że te "przeskoki" nie sa widoczne.

Taki efekt można osiągnąć na kilka sposobów. Po pierwsze, wprowadzona zostanie interpolacja pozycji i rotacji postaci. Nie będą one zmieniały pozycji nagle, lecz będzie ona płynnie, liniowo zmieniana w czasie. Ponadto zastosowane zostaną predykcje klienckie - poza pozycją i rotacją gracza przesyłana będzie również jego prędkość. W przypadku utraty wielu pakietów postać będzie nadal poruszała się z tą samą prędkością. Jest to sposób na "zgadnięcie" przyszłych, nieznanych pozycji gracza.

Tabela 3.2. Początkowy zbiór kart

lp.	Nazwa angielska	Opis
1	TANK	Zwiększona żywotność, Zmniejszona szybkość
2	Speedy Gonzales	Zwiększona szybkość, Zmniejszona żywotność
3	Glass Canon	Zwiększone obrażenia, Zancznie zmniejszona żywotność
4	Cockroach	Zmniejszona żywotność, zwiększona wartość leczenia
5	Sniper	Zwiększone obrażenia, zwiększona szybkość kuli, zmniejszona szybkostrzelność, zmniejszona pojemność magazynka
6	Rubber Bullets	Dodatkowe odbicia kuli, Wydłużony czas przeładowania
7	FULL AUTO	Zwiększona szybkostrzelność, zmniejszone obrażenia, znacznie zwiększona pojemność magazynku, zwiększony czas przeładowania (jeśli się uda: Automatyczne strzelanie, nie trzeba puszczać przycisku strzału, aby wykonać kolejny strzał)
8	Granade Launcher	Zmniejszone obrażenia pocisku, zmniejszona prędkość pocisku, AKTYWNA: Przy trafieniu lub zakończeniu życia pocisk eksploduje, zadając obrażenia w promieniu
9	Flame	Pociski podpalają trafionego gracza, zadając mu obrażenia przez kilka sekund, Zwiększone obrażenia, wydłużony czas przeładowania
10	Life Steal	Trafienie przeciwnika leczy właściciela o część zadanych obrażeń
11	NUKE	Zmniejszenie pojemności magazynku, Znaczne zwiększenie obrażeń, Eksplozja
12	Quick Reload	Znaczne zmniejszenie czasu przeładowania
13	BIG BOY	Znaczne zwiększenie żywotności, Znaczne zmniejszenie wartości leczenia (albo zwiększenie czasu leczenia)
14	Fragmentation	Po trafieniu przeszkody w miejscu trafienia tworzone są mniejsze, słabsze pociski skierowane w losowych kierunkach (odłamki), Zwiększony czas przeładowywania
15	Freezing Bullet	Zwiększony czas przeładowania, Po trafieniu postaci zostaje ona zamrożona - jej szybkość zostaje znacznie zmniejszona
16	Cowboy	Znacznie zwiększona szybkostrzelność, Zwiększone obrażenia, Znacznie zwiększony czas przeładowania
17	CHAOS	Znacznie więcej odbić kuli, Zmniejszone obrażenia
18	Knockback	Trafiona postać zostaje odepchnięta w kierunku, w którym leciała kula
19	Directed Bounce	Dodatkowe jedno odbicie, Kule odbijają się w kierunku najbliższego widocznego gracza, jeżeli żaden nie jest widoczny odbijają się normalnie
20	Tactical Advantage	Trafienie przeciwnika przeładowuje magazynek właściciela, Znaczne zwiększenie czasu przeładowania

4. PROTOTYP

4.1. CELE I ZAKRES

Jako początkowy etap przygotowania projektu stworzono prototyp gry. Został on przygotowany w kilku celach.

1. Zapoznanie z możliwościami i ograniczeniami silnika.
2. Określenie trudności przygotowania warstwy sieciowej.
3. Zapoznanie z językiem skryptowym GDScript.

W ramach prototypu zaimplementowano jedynie bardzo podstawowe mechaniki: poruszanie, celowanie i strzelanie. System sieciowy również zaimplementowano w uproszczonej formie, nie implementując również wszystkich mechanik sieciowo - jedynie poruszanie i celowanie.

Na tym etapie nie przygotowano żadnych grafik, jako modele i poziom zostały wykorzystane jedynie proste figury geometryczne generowane w silniku.

4.2. MECHANIKI

W ramach prototypu wprowadzone zostały implementacje poruszania/sterowania, celowania i strzelania.

4.2.1. Przygotowanie projektu

Tworzenie projektu rozpoczęto od przygotowania świata gry oraz postaci gracza, w celu umożliwienia testowania oprogramowania.

Świat gry został zbudowany z przeskalowanych prostopadłościanów działających jako podłoże, ściany ograniczające poziom oraz przeszkody w poziomie. Wykorzystano również figury CSG (*Constructive Solid Geometry* - ang. Konstrulcyjna Geometria Bryłowa) w celu stworzenia bardziej skomplikowanej przeszkody. Bryły tego rodzaju można łączyć w spójne figury z pomocą takich operacji jak suma, różnica czy część wspólna.

Awatar gracza został złożony z dwóch części - ciała (ang. *body*) i głowy (ang. *head*). Są to jedynie nazwy dla prostego rozróżnienia kadłuba od wieżyczki wraz z lufą. Taki podział został wprowadzony w celu prostszego i niezależego sterowania transformacją oraz rotacją tych elementów. Całość została przygotowana z wykorzystaniem trzech brył - prostopadłościanu dla ciała, kuli dla wieżyczki oraz walca dla lufy.

4.2.2. Poruszanie

Został wprowadzony schemat sterowania zgodny z ustaleniami sekcji 3.1.1 oraz tabeli 5.1.

Postać gracza poruszana jest jedynie w przypadku jeżeli odpowiedni przycisk jest przytrzymywany. W tym celu w procesie fizycznym wykonywane jest sprawdzenie wprowadzanych przez gracza poleceń.

Proces fizyczny to metoda w skryptach Godota, wywoływana co ustalony czas, niezależny od wyświetlanych klatek. Pozwala to na ujednolicenie działania krytycznych części kodu w przypadku gdy scena i klatka może być generowana dłużej niż zwykle.

Ponieważ ruch w osi przód-tył i obrót w prawo-lewo są różnymi zachowaniami wejścia z nimi związane zostały zapisane do oddzielnych zmiennych. Ponadto, ponieważ jednoczesny obrót lub ruch w obie strony się wykluczają, wartość tych dwóch poleceń zostaje od siebie odjęta. 4.1

Listing 4.1. Kod pobierający polecenia gracza

```
var forward_input = int(Input.is_action_pressed("Forward"))
  - int(Input.is_action_pressed("Back"))
var turn_input = int(Input.is_action_pressed("Left"))
  - int(Input.is_action_pressed("Right"))
```

`Input` jest obiektem tworzonym przez silnik, singletonem, który zarządza interfejsami poleceń gracza takimi jak klawiatura i mysz. Zamiast metody `is_action_pressed`, która sprawdza aktualny stan akcji, możliwe jest również wykorzystanie metody `get_axis`, która pozwala skrócić powyższy zapis zachowując ten sam rezultat. Odpowiednie akcje zostały stworzone i zmapowane do odpowiednich wejść z tabeli 5.1, korzystając z mapowania wejścia w ustawieniach projektu Godota.

W skrypcie gracza zdefiniowane zostały atrybuty niezbędne do implementacji poruszania - prędkość maksymalna i aktualna poruszania oraz prędkość kątowa maksymalna i aktualna obrotu. Aktualna prędkość poruszania jest wartością wektorową, pozostałe zaś są wartościami skalarnymi. Zmienne przechowujące wartości maksymalne są zdefiniowane wykorzystaniem słowa kluczowego `export` co pozwala na edytowanie ich w oknie silnika Godot.

Jako oś przód-tył wykorzystano oś z modelu gracza, gdzie przód jest zwrócony zgodnie z dodatnią częścią osi. W celu poruszenia całego obiektu gracza wywołana zostaje funkcja `move_and_slide`, która służy do przemieszczania obiektów z uwzględnieniem kolizji. Do obrócenia modelu wykorzystano metodę `rotate_y`, ponieważ w silniku Godot oś "y" jest osią pionową.

Została podjęta decyzja o wykorzystaniu kamery trzecioosobowej, tj. takiej, która podąża za graczem i widzi również jego model, w odróżnieniu od kamery pierwszoosobowej, która widzi świat z perspektywy gracza. Wybrano wbudowaną w silnik Godot kamerę

interpolowaną (`InterpolatedCamera`). Jest to kamera, która porusza się płynnie tak, aby jej położenie pokrywało się z jej celem. Cel kamery jest również obiektem o odpowiedniej pozycji i rotacji.

Cel oraz kamera zostały dodane do sceny świata, a nie gracza, aby na etapie dodawania systemu sieciowego (4.3) móc uniknąć problemu wielu niewykorzystanych obiektów.

W skrypcie świata, w procesie fizycznym cel kamery jest przenoszony w pozycję o odpowiednich koordynatach. kamera będzie automatycznie przemieszczała się tak, aby śledzić ten punkt, jednak jej rotacja również ustawiana programistycznie, poprzez wykorzystanie metody `look_at`.

4.2.3. Celowanie

Celowanie zostało zaimplementowane zgodnie z sekcją 3.1.2. W tym celu należy wykonać następujące kroki:

1. Pobrać pozycję myszy w oknie gry;
2. Pobrać pozycję kamery w świecie gry;
3. Wyznaczyć promień przechodzący od kamery w kierunku wskazywanym przez mysz;
4. Zbadać przecięcia powyższego promienia z obiektami świata gry;
5. Wycelować w kierunku wyznaczonego punktu przecięcia.

Kod funkcji realizującej punkty 1-4 został zamieszczony w listingu 4.2.

Listing 4.2. Funkcja rzutująca mysz na świat gry

```
func mousePositionToWorldPosition():
  var space_state = get_world().direct_space_state
  var mouse_pos = get_viewport().get_mouse_position()

  var camera = get_tree().root.get_camera()
  if camera == null:
    return null

  var ray_origin = camera.global_translation
  var ray_direction = camera.project_position(mouse_pos, 300)

  var ray_array = space_state.intersect_ray(ray_origin, ray_direction)

  if ray_array.has("position"):
    return ray_array["position"]
  return null
```

Jeżeli zostanie znaleziony punkt przecięcia promienia z obiektami świata, cała "głowa" modelu gracza jest obracana w jego kierunku. W prototypie punkt ten jest również wizualizowany kulą, która się w nim pojawia. Pomogło to w zniwelowaniu błędów wynikających z różnicy między przestrzenią lokalną modelu gracza a globalną.

4.2.4. Strzelanie

Stworzony został obiekt pocisku. Jego modelem jak i kształtem kolizji jest figura kapsuły.

Jako część postaci gracza dodany został węzeł typu `Spatial` reprezentujący punkt, w którym tworzone będą pociski. Został on umieszczony na końcu lufy i przypisany jako dziecko tego obiektu. W ten sposób punkt ten będzie przemieszczał się tak samo jak jego rodzic. Do skryptu gracza została również dodana zmienna przechowująca referencję do sceny pocisku.

Do skryptu gracza dodano także funkcję strzału, oraz kod przyjmujący polecenie strzału z myszy. Podobnie jak akcje poruszania, akcja strzału została zmapowana w ustawieniach projektu. Z perspektywy gracza strzał polega jedynie na stworzeniu instancji sceny pocisku, umieszczeniu jej w punkcie strzału na końcu lufy oraz przypisaniu jej do świata gry. Pociski nie mogą być zapisywane jako dzieci gracza, ponieważ wtedy przemieszczałyby się razem z nim.

Do skryptu pocisku dodano stałe określające szybkość poruszania oraz limit czasu istnienia obiektu. W momencie utworzenia instancji pocisku licznik czasu istnienia zaczyna odliczać liczbę sekund równą limitowi. W przypadku zakończenia licznika obiekt pocisku jest niszczony. W procesie fizycznym pocisk przemieszczany jest z odpowiednią szybkością w jego lokalnym kierunku $+z$. Po wykryciu zderzenia pocisk jest niszczony.

4.3. SYSTEM SIECIOWY

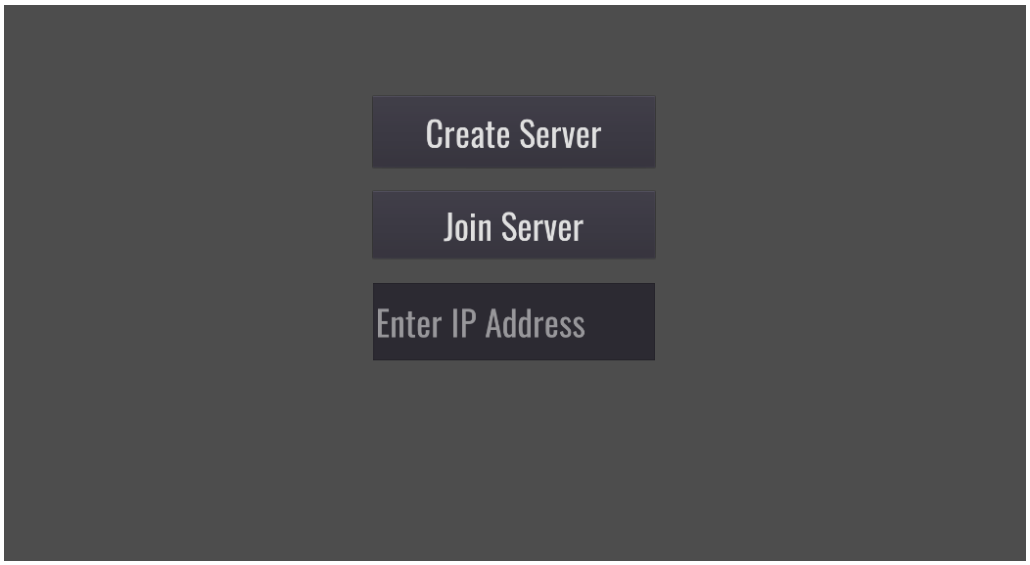
Podstawowy system sieciowy został zaimplementowany z wykorzystaniem wideoporadnika [19]. Ostateczna implementacja została jednak dostosowana do przygotowywanego rozwiązania, ponieważ projekt z poradnika jest tworzony z wykorzystaniem grafiki 2D.

Do projektu dodano pusty węzeł statyczny mający przechowywać graczy w drzewie. Ponadto dodano menu startowe (Rys. 4.1). Stworzono również dwa skrypty globalne oraz zmodyfikowano skrypt gracza.

Na interfejsie menu widoczne są dwa przyciski. Przycisk “Create Server” powoduje rozpoczęcie gry jako host. Gra hostowana jest wtedy na adresie IP maszyny użytkownika. Przycisk “Join Server” powoduje rozpoczęcie gry na serwerze, którego adres IP wpisany jest w pole tekstowe. W prototypie nie zostały zastosowane żadne techniki obrony przed wpisaniem niepoprawnego adresu IP ani próby połączenia z nieistniejącym serwerem.

Skrypt globalny `Network` jest odpowiedzialny za połączenia sieciowe. Korzysta on z wysokopoziomowego interfejsu sieciowego Godota.

Podczas uruchomienia gry ten skrypt pobiera adres IP maszyny. Następnie łączy się z sygnałami związanymi ze zmianami sieciowego stanu gry (Listing 4.3).



Rys. 4.1. Menu startowe prototypu po wprowadzeniu systemu sieciowego

Listing 4.3. Podłączanie do najważniejszych sygnałów sieciowych

```
get_tree().connect("connected_to_server", self, "_connected_to_server")
get_tree().connect("server_disconnected", self, "_server_disconnected")
get_tree().connect("network_peer_connected", self, "_player_connected")
get_tree().connect("network_peer_disconnected",self,"_player_disconnected")
```

Poniżej zostały opisane sygnały oraz przypisane do nich funkcje:

— `connected_to_server` zostaje emitowany gdy gra połączy się z serwerem. Funkcja tworzy instancję własnej postaci gracza.
— `server_disconnected` zostaje emitowany gdy gra zostanie rozłączona z serwerem; W prototypie funkcja jedynie loguje wydarzenie.
— `network_peer_connected` zostaje emitowany gdy inny gracz zostaje podłączony do gry. W sytuacji gdy gracz pierwszy raz podłącza się do serwera ten sygnał emitowany jest dla wszystkich graczy już na serwerze. Funkcja tworzy instancję postaci nowo podłączonego gracza.
— `network_peer_disconnected` zostaje emitowany gdy inny gracz odłącza się od serwera. Funkcja usuwa postać rozłączonego gracza.

Ponadto w skrypcie `Network` zdefiniowane są funkcje `create_server`**??** i `join_server`**??** służące do, odpowiednio, stworzenia serwera i dołączenia do serwera. W tym celu wykorzystana została klasa `NetworkedMultiplayerENet`, implementująca warstwę sieciową korzystającą z połączenia UDP.

Listing 4.4. Funkcja inicjująca serwer gry.

```
func create_server() -> void:
  server = NetworkedMultiplayerENet.new()
  server.create_server(DEAFAULT_PORT, MAX_CLIENTS)
```

```
  get_tree().set_network_peer(server)
```

Listing 4.5. Funkcja łącząca do serwera gry.

```
func join_server() -> void:
  client = NetworkedMultiplayerENet.new()
  client.create_client(ip_address, DEAFAULT_PORT)
  get_tree().set_network_peer(client)
```

Skrypt `Global` składa się z metod ułatwiających instancjonowanie graczy. Są one wykorzystywane podczas rozpoczynania gry.

Do sceny gracza dodano węzły `Timer` - odliczający okres synchronizacji z serwerem i `Tween` - pozwalający płynnie przekształcać właściwości węzła. Do skryptu gracza dodano zmienne oznaczone słowem kluczowym typu `puppet` - są to wartości, które będą synchronizowane przez sieć. Takie zmienne stworzono dla pozycji i rotacji gracza, rotacji wieżyczki oraz prędkości gracza.

`Timer` ustawiony został na okres 0.3 sekundy. Co taki okres wysyła on sygnał, który wywołuje funkcję synchronizującą wartości zmiennych puppet po sieci. W momencie ustawienia w ten sposób pozycji jest ona interpolowana z wykorzystaniem węzła `Tween`. Jest to implementacja założeń z sekcji 3.1.7.2. Postać gracza wysyła wtedy wartość swoich rzeczywistych zmiennych. Awatar gracza w grze pozostałych graczy w procesie fizycznym zmienia również rotację ciała i wieżyczki. Jeżeli `Tween` nie jest aktywny, postać jest przemieszczana z prędkością otrzymaną z sieci. Jest to implementacja predykcji klienckich, również z sekcji 3.1.7.2.

4.4. WNIOSKI

Zgodnie z założeniami, przygotowanie prototypu pozwoliło zapoznać się z zawiłościami silnika Godot oraz zlokalizować przyszłe trudności.

Implementacja podstawowych mechanik nie będzie stanowić większych trudności. Zostaną one zaimplementowane w sposób zbliżony do tego z prototypu. Połączenie sieciowe będzie stanowiło największe wyzwanie. Aby jego implementacja przebiegła sprawnie niezbędny będzie dokładny projekt modelu danych oraz ich przepływu przez sieć. Jednak zastosowanie wysokopoziomowej warstwy sieciowej silnika Godot pozwoli znacznie uprościć cały system sieciowy.

Tabela 5.1. Schemat sterowania

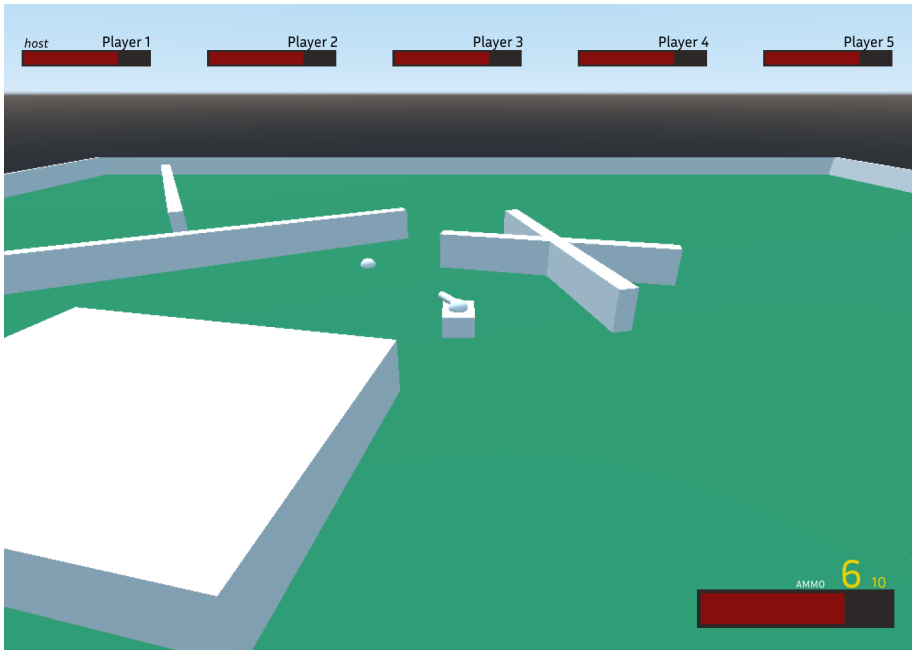
Nazwa akcji	Przypisany przycisk	Opis czynności
Forward	W	Poruszanie do przodu.
Back	S	Poruszanie do tyłu.
Left	A	Obrót postaci w lewo, przeciwnie do ruchu wskazówek zegara.
Right	D	Obrót postaci w prawo, zgodnie z ruchem wskazówek zegara.
MainAction	Lewy przycisk myszy	Akcja podstawowa - strzał.

Rys. 5.1. Projekt menu głównego.

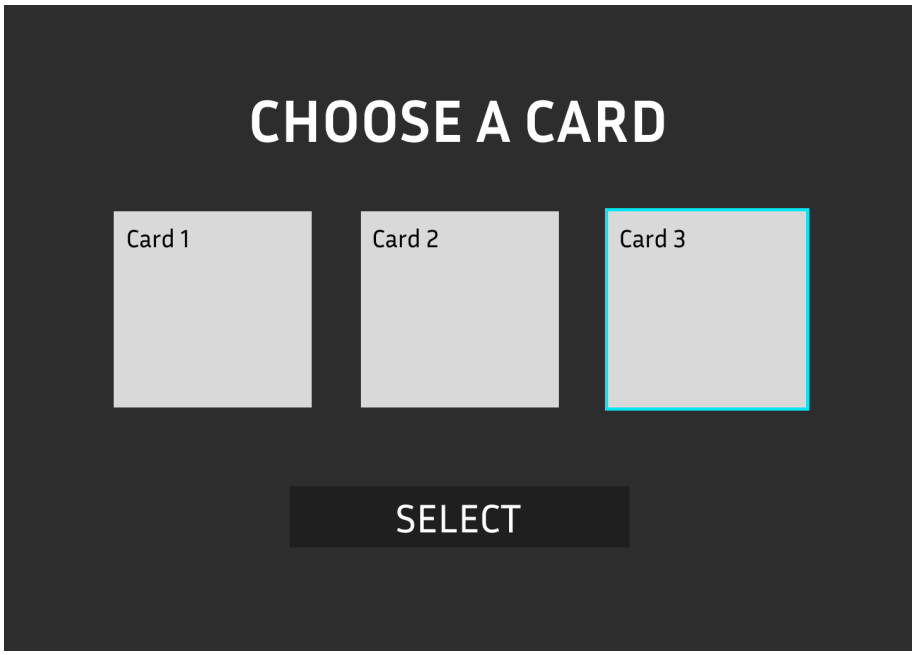
Rys. 5.2. Projekt menu hosta.

Rys. 5.3. Projekt menu dołączającego gracza.

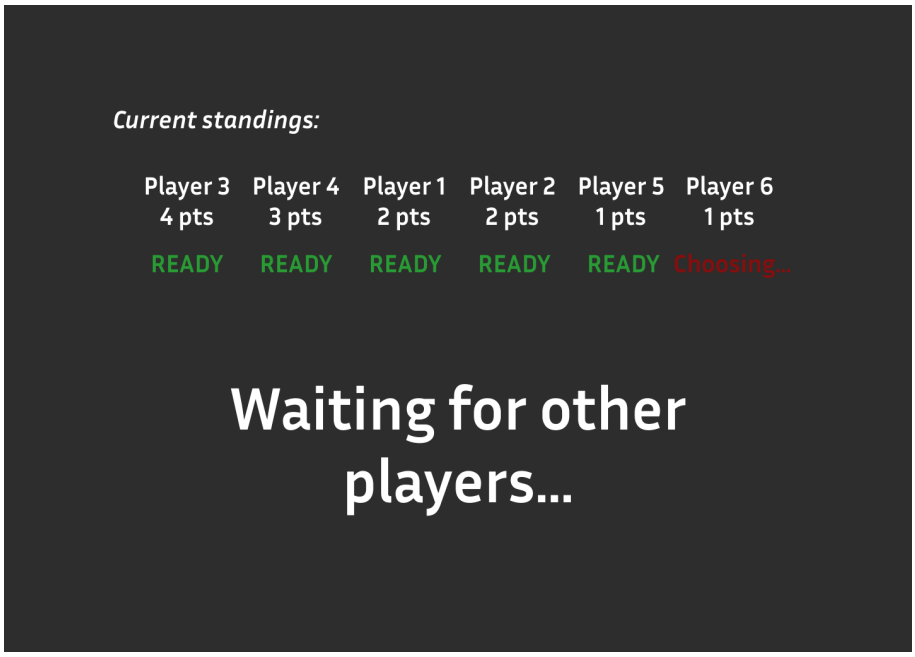
Rys. 5.4. Projekt menu poczekalni.



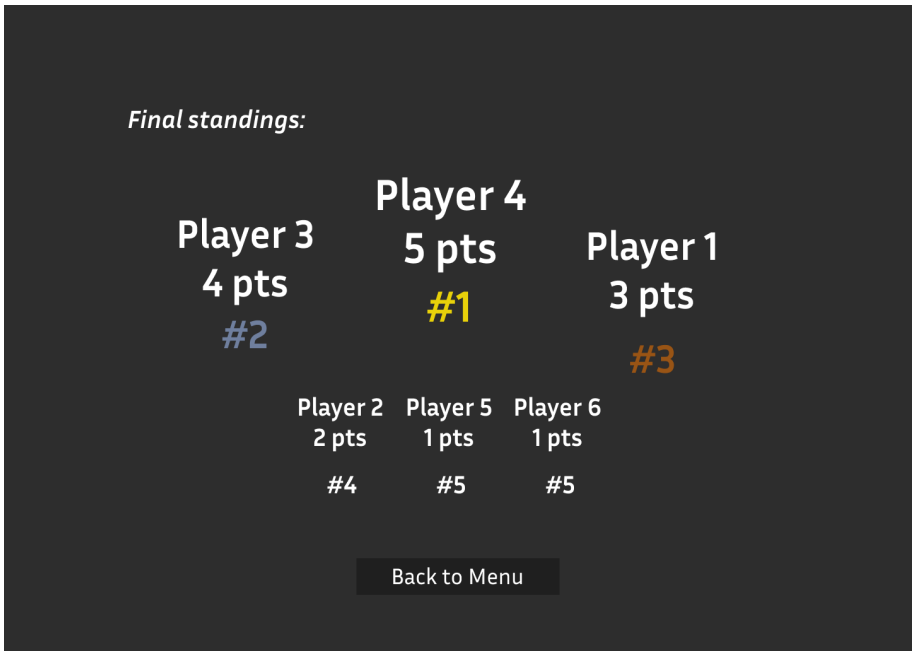
Rys. 5.5. Projekt interfejsu HUD.



Rys. 5.6. Projekt menu wyboru kart.



Rys. 5.7. Projekt menu oczekiwania między turami.



Rys. 5.8. Projekt ekranu po zakończeniu rozgrywki.

5. PROJEKT APLIKACJI

5.1. INTERFEJS UŻYTKOWNIKA

5.1.1. Menu

5.1.2. HUD

5.1.3. Schemat sterowania

5.2. IMPLEMENTACJA MECHANIK

5.2.1. Model danych postaci

5.2.2. Poruszanie

5.2.3. Strzelanie

5.2.3.1. Wydarzenia po trafieniu

5.2.4. Zdrowie

5.2.5. Rundy

5.2.6. Karty ulepszeń

5.3. SYSTEM SIECIOWY

5.3.1. Poczekalnia

5.3.2. Proces hostowania gry

5.3.3. Proces dołączania do gry

5.3.4. Przepływ danych w grze

6. IMPLEMENTACJA

6.1. USTAWIENIA PROJEKTU

6.1.1. Mapowanie wejść

6.1.2. Ustawienia okna

6.1.3. Logowanie

6.2. INTERFEJS UŻYTKOWNIKA

6.3. IMPLEMENTACJA MECHANIK

6.4. SYSTEM SIECIOWY

6.5. TESTY

6.6. WYNIKI IMPLEMENTACJI

7. PODSUMOWANIE

7.1. DALSZY ROZWÓJ PROJEKTU

7.2. WNIOSKI

BIBLIOGRAFIA

[1] *Projekt godot na platformie github*, https://github.com/godotengine. Dostęp 23.11.2022.

[2] Adam Dodd, *[horror declassified] an examination of tank controls*, https://bloody-disgusting.com/news/3224958/horror-declassified-an-examination-of-tank-controls/. 2013. Dostęp 12.11.2022.

[3] Adam Kramarzewski, Ennio De Nucci, *Practical Game Design* (Packt, 2018).

[4] Alexander Shvets, *Design patterns: Singleton*, https://refactoring.guru/design-patterns/singleton. Dostęp 23.11.2022.

[5] Andrew Sanders, *An introduction to Unreal engine 4* (AK Peters/CRC Press, 2016).

[6] Antonín Šmíd, *Comparison of Unity and Unreal Engine*, Praca magisterska, Czech Technical University in Prague. 2017. Dostęp 11.12.2022.

[7] Blender Foundation, *Strona główna projektu blender*, https://www.blender.org/. Dostęp 11.12.2022.

[8] Butch Wesley, *Wiki projektu gut*, https://github.com/bitwes/Gut/wiki. Dostęp 11.12.2022.

[9] CD Projekt RED, *Studio cd projekt red będzie tworzyć gry na silniku unreal engine 5 w ramach strategicznej współpracy z epic games*, https://www.cdprojekt.com/pl/media/aktualnosci /nowa-wiedzminska-saga-zapowiedziana-studio-cd-projekt-red-bedzie-tworzyc-gry-na-silniku -unreal-engine-5-w-ramach-strategicznej-wspolpracy-z-epic-games/. 2022.

[10] Epic Games, *Strona główna unreal engine*, https://www.unrealengine.com/en-US. Dostęp 11.12.2022.

[11] Jason Gregory, *Game Engine Architecture* (A K Peters, Ltd., 2009).

[12] Juan Linietsky, Ariel Manzur, *Strona główna projektu godot*, https://godotengine.org/. Dostęp 23.11.2022.

[13] Juan Linietsky, Ariel Manzur and the Godot community, *Godot docs*, https://docs.godotengine.org/en/3.5/. Dostęp 12.11.2022.

[14] Lee Salzman, *Dokumentacja biblioteki enet*, http://enet.bespin.org/usergroup0.html. Dostęp 11.12.2022.

[15] Marcus Toftedahl, *Which are the most commonly used game engines?*, https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-. 2019. Dostęp 11.12.2022.

[16] Michelle Menard, Bryan Wagstaff, *Game Development with Unity* (Cengage Learning PTR, 2015).

[17] Mullen, T., *Mastering blender* (John Wiley & Sons, 2011).

[18] Nathan Schuetz, *Game feel: Input buffering*, https://barbariangrunge.com/game-feel-input-buffering/. 2022. Dostęp 12.11.2022.

[19] PlugWorld, *Godot networked multiplayer shooter tutorial*,

https://www.youtube.com/playlist?list=PL6bQeQE-ybqDmGuN7Nz4ZbTAqyCMyEHQa. 2021. Dostęp 23.11.2022.

[20] Robert Nystrom, *Game Programming Patterns* (Genever Benning, 2014).

[21] Szymon Datko, *Projektowanie i programowanie gier. wykład nr 4 - silniki gier komputerowych i ciekawostki techniczne*, https://datko.pl/PiPG/wyk4.pdf. 2021. Dostęp 19.11.2022.

[22] Unity Technologies, *Strona główna unity*, https://unity.com/. Dostęp 11.12.2022.

SPIS RYSUNKÓW

SPIS TABEL