

**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, grafika 3D

---

WROCLAW 2022



## **STRESZCZENIE**

Celem niniejszej pracy dyplomowej było zaprojektowanie i stworzenie wieloosobowej gry sieciowej. Gra miała zostać stworzona z wykorzystaniem silnika Godot. Gra wykorzystuje grafikę 3D oraz widok trzecioosobowy. Aplikacja umożliwia połączenie od dwóch do sześciu graczy w sieci lokalnej LAN. Stworzona aplikacja została przetestowana jednostkowo i integracyjnie z wykorzystaniem narzędzia GUT oraz manualnie pod systemem operacyjnym Windows 10.

Przed rozpoczęciem implementacji przygotowana została koncepcja gry, plan mechanik oraz najważniejszych systemów. W następnej kolejności stworzony został prototyp. Przygotowując prototyp przeprowadzona została analiza trudności implementacji systemu sieciowego.

Zaprojektowane zostały interfejsy użytkownika oraz najważniejsze podsystemy. W następnej kolejności zostały one zaimplementowane zgodnie z koncepcją i projektami. Została również przeprowadzona dyskusja wyników, analiza wniosków. Opisano również możliwości dalszego rozwoju projektu.

## **ABSTRACT**

The goal of this thesis was to design and create a game featuring an on-line multiplayer using the Godot game engine. The game uses 3D graphics, providing player with third person view. Application enables two to six players to connect over local network. Unit and integration tests were created using the GUT toolkit. Manual tests were also ran under Windows 10.

Before implementing the software, a game design concept was formulated. It consists of plans for mechanics and systems to be featured in the game. A prototype was created in order to analyze the difficulty of implementing the network system.

Designs included graphical interfaces and the most important subsystems. Those were implemented according to concepts and designs. Finally, the results were analyzed and some ways of improving the project in the future were proposed.



# SPIIS TREŚCI

<b>Wstęp</b>	4
Cel pracy	4
Technologie	4
Zakres pracy	5
<b>1. Analiza technologii</b>	6
1.1. Silnik gier	6
1.1.1. Popularne silniki gier	6
1.2. Silnik Godot	7
1.2.1. Język GDScript	8
1.2.2. Drzewo gry	8
1.2.3. Zasoby	9
1.2.4. Renderowanie grafiki	9
1.2.5. Sygnały	10
1.2.6. Edytor	10
1.2.7. System sieciowy	10
1.3. GUT	11
1.4. Blender	12
<b>2. Analiza rynku</b>	13
2.1. Podobne rozwiązania	13
2.1.1. ROUNDS	13
2.1.2. Boomerang Fu	13
2.1.3. World of Tanks	13
2.1.4. Wii Play: Tanks!	14
<b>3. Koncepcja gry</b>	15
3.1. Mechaniki	15
3.1.1. Sterowanie gracza	15
3.1.2. Strzelanie	16
3.1.3. System zdrowia i życia	16
3.1.4. System rund	16
3.1.5. Atrybuty pasywne i aktywne	17
3.1.6. Karty ulepszeń	17
3.1.7. User experience	19
<b>4. Prototyp</b>	21

4.1.	Cele i zakres . . . . .	21
4.2.	Mechaniki . . . . .	21
4.2.1.	Przygotowanie projektu . . . . .	21
4.2.2.	Poruszanie . . . . .	22
4.2.3.	Celowanie . . . . .	23
4.2.4.	Strzelanie . . . . .	24
4.3.	System sieciowy . . . . .	24
4.4.	Wnioski . . . . .	26
<b>5.</b>	<b>Projekt aplikacji . . . . .</b>	<b>27</b>
5.1.	Interfejs użytkownika . . . . .	27
5.1.1.	Menu . . . . .	27
5.1.2.	Schemat sterowania . . . . .	31
5.2.	Projekt implementacji mechanik . . . . .	31
5.2.1.	Model danych postaci . . . . .	33
5.2.2.	Poruszanie . . . . .	34
5.2.3.	Strzelanie . . . . .	34
5.2.4.	Zdrowie . . . . .	35
5.2.5.	Rundy . . . . .	35
5.2.6.	Karty ulepszeń . . . . .	35
5.3.	System sieciowy . . . . .	36
5.3.1.	Poczekalnia . . . . .	36
5.3.2.	Przepływ danych w grze . . . . .	36
<b>6.</b>	<b>Implementacja . . . . .</b>	<b>38</b>
6.1.	Interfejs użytkownika . . . . .	38
6.2.	Postać gracza . . . . .	39
6.3.	Implementacja mechanik . . . . .	41
6.3.1.	Strzelanie . . . . .	41
6.3.2.	Komendy . . . . .	43
6.3.3.	Karty . . . . .	44
6.4.	Obiekty globalne . . . . .	44
6.5.	Testy . . . . .	45
<b>7.</b>	<b>Podsumowanie . . . . .</b>	<b>46</b>
7.1.	Wnioski . . . . .	46
7.2.	Znane problemy . . . . .	46
7.3.	Dalszy rozwój projektu . . . . .	47
7.3.1.	Warstwa techniczna . . . . .	47
7.3.2.	Warstwa kreatywna . . . . .	47
	<b>Bibliografia . . . . .</b>	<b>49</b>
	<b>Spis rysunków . . . . .</b>	<b>51</b>

**Spis tabel . . . . . 52**

# 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. [1] 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.



## **ZAKRES PRACY**

Rozdział 1. zawiera analizę najważniejszych technologii wykorzystywanych w projekcie. W rozdziale 2. przeanalizowano inne podobne rozwiązania aktualnie dostępne na rynku. Rozdział 3. opisuje planowane działanie gry i jej mechanik z perspektywy *game designu*. W rozdziale 4. opisana została implementacja prototypu przygotowana w celu zapoznania się z silnikiem i określenia trudności niektórych zadań. Rozdział 5. opisuje projekt implementacji najważniejszych części systemu. Rozdział 6. przedstawia istotne elementy i szczegóły implementacyjne przygotowanego rozwiązania. W rozdziale 7. przeprowadzona została dyskusja wyników oraz wniosków a także przedstawione zostały możliwości dalszego rozwoju projektu.

# 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 [2]:

- 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[3]. 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 pełną kontrolę zespołu bądź firmy, mogą być 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[4][5]. Jest jednym z najpopularniejszych silników, wykorzystywanym w niemalże połowie projektów dostępnych na platformie itch.io[6]. 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[7][8]. Słabiej niż Unity nadaje się do tworzenia mniejszych gier np. na platformy mobilne[9]. W związku z tym, szczególnie w ostatnich latach[10], 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[6].

### **1.2. SILNIK GODOT**

Silnik Godot[1] 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[11], 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[12].

### 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.

Podobnie jak Python, GDScript nie jest językiem o silnym typowaniu. Posiada jednak możliwość określenia typu zmiennej w celu uproszczenia rozwoju oprogramowania. Te typy są wykorzystywane tylko w fazie tworzenia gier. W czasie działania aplikacji zmienne mogą przechowywać dane dowolnego typu, i język nie chroni przed błędnymi przypisaniami. Ze względu na to ograniczenie należy stosować technikę znaną jako *duck typing* (ang. typowanie kaczkowe). Polega ona na sprawdzeniu istnienia konkretnej metody bądź pola przed ich użyciem w czasie egzekucji programu. Nazwa wywodzi się z założenia, że „jeżeli coś wygląda jak kaczka i kwacze jak kaczka, to jest to kaczka”.

### 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[13]. 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 „ulokować” 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 [14], 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. *Application 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`[15]. 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 być 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 połączanego 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*)[16] 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[17][18]. 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**

„ROUNDS” jest konkurencyjną grą dwuosobową wydaną w 2021 roku przez studio Landfall Games[19]. Umożliwia grę lokalnie, na jednym urządzeniu lub przez sieć korzystając z serwerów Steam. Rozgrywka prowadzona jest w rundach, między którymi gracz przegrywający otrzymuje karty wzmacniające jego możliwości. W grze zawartych jest ponad 60 kart modyfikujących różne atrybuty postaci. Rozgrywka odbywa się w płaszczyźnie dwuwymiarowej z widokiem „z boku”. Oznacza to, że zarówno na postaci graczy jak i na pociski oddziałuje grawitacja. Dodaje to znaczną głębię rozgrywki. Ponadto oprócz poruszania i strzału gracz ma do wykorzystania kolejną interakcję - blok. Blokując gracz przez krótką chwilę może odbić nadlatujące pociski. Dodatkowo wiele kart dodaje dodatkowe zdolności do bloku zwiększające jego możliwości bojowe.

#### **2.1.2. Boomerang Fu**

Wydane w 2020 roku przez studio Cranky Watermelon „Boomerang Foo” jest imprezową grą akcji[20]. Umożliwia grę wieloosobową dla 2 do 6 graczy na jednym urządzeniu. Gracze sterują postaciami, korzystając z bumerangów w celu wzajemnej eliminacji. Po każdej z rund gracze dostają punkty w zależności od liczby wyeliminowanych przeciwników. W czasie rozgrywki gracze mają możliwość otrzymania wzmocnień poprzez zebranie ich na planszy. Gracz może jednocześnie mieć maksimum trzy wzmocnienia. Zmieniają one zachowanie bumerangu lub postaci i nie modyfikują żadnych charakterystyk na stałe.

#### **2.1.3. World of Tanks**

Wydana przez studio Wargaming.net sieciowa gra „World of Tanks” jest sieciową strzelanką, w której gracze sterują czołgami[21]. Gra udostępnia ponad 600 różnych modeli czołgów z okresu 20. wieku. W związku ze skupieniem na historycznych modelach czołgów gra celuje w realistyczne oddanie wyglądu i mechaniki działania maszyn bojowych. Ponadto realizm osiągany jest również w samym świecie gry. Na wielu dostępnych poziomach istotne są nie tylko przeszkody fizyczne ale również ukształtowanie terenu ograniczające lub zwiększające możliwości gracza. Gra udostępnia też wiele różnorodnych trybów rozgrywki.

#### **2.1.4. Wii Play: Tanks!**

„Tanks!” jest jedną z minigier zawartych w zestawie „Wii Play” dostępnym na konsolę Nintendo Wii[22]. Jest to gra jednoosobowa w której gracz steruje czołgiem próbując wyeliminować sterowane przez sztuczną inteligencję czołgi przeciwników. Każdy z poziomów składa się z ustalonego zestawu wrogów. Gra prowadzona jest z trzeciej osoby ze stacjonarnym widokiem izometrycznym. Dostępny jest również lokalny wieloosobowy tryb częściowo-kooperacyjny. W tym trybie dwóch graczy rozgrywa poziomy jednoosobowe wspólnie, zdobywając punkty za każdego wyeliminowanego przeciwnika. Czołgi przeciwników należą do jednego z kilku rodzajów, każdy rodzaj posiada inne charakterystyki poruszania i strzałów.

### 3. KONCEPCJA GRY

[3] 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 szczegółowo opisane zostaną mechaniki gry oraz wymagania z dziedziny *game designu*.

#### 3.1. MECHANIKI

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*). [23] 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 wystrzelonemu pociskowi.

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 nadmierne 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 reprezentacji różnych cech postaci gracza każdy z nich ma swoje *atrybuty*. Dzieli 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*)[24].

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

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 kątowna	Ruch	Z taką najwyższą prędkością kątowną może obracać się postać gracza.
Maksymalna żywotność	Zdrowie	Tyle najwyższej 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.

#### **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 wywołane przez te polecenia zostają aktywowane w momencie gdy nastanie taka możliwość[25].

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 *wyduje się* być bardziej sprawiedliwa poprzez wybaczenie 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 są 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 "zgodnięcie" 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, Znacznie 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, AK-TYWNA: 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. Konstrukcyjna 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 niezależnego 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 kątowna 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 po-  
dąża 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 od-  
powiednich 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.

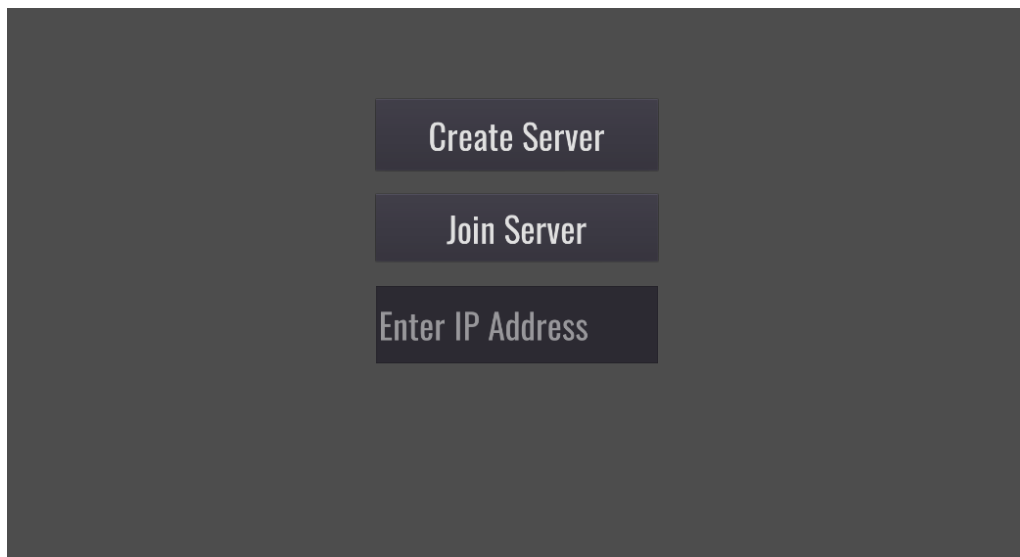
Do skryptu pocisku dodano stałe określające szybkość poruszania oraz limit czasu istnienia obiektu. W momencie utworzenia instancji pocisku licznik czasu istnienia zaczyna odliczać liczbę sekund równą limitowi. W przypadku zakończenia licznika obiekt pocisku jest niszczone. W procesie fizycznym pocisk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

### 4.3. SYSTEM SIECIOWY

Podstawowy system sieciowy został zaimplementowany z wykorzystaniem wideoporadnika [26]. 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.



Rys. 4.1: Menu startowe prototypu po wprowadzeniu systemu sieciowego

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).

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` 4.4 i `join_server` 4.5 służące do, odpowiednio, stworzenia serwera i dołączenia do serwera. W tym celu wy-

korzystana 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(DEAFULT_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, DEAFULT_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.

## 5. PROJEKT APLIKACJI

### 5.1. INTERFEJS UŻYTKOWNIKA

Istotną częścią aplikacji są interfejsy użytkownika. Możemy tu wyróżnić wyjściowe interfejsy graficzne oraz wejściowy interfejs sterowania - klawiaturę i mysz. Diagram na rysunku 5.1 prezentuje przejścia pomiędzy widokami prezentowanymi w sekcji 5.1.1.

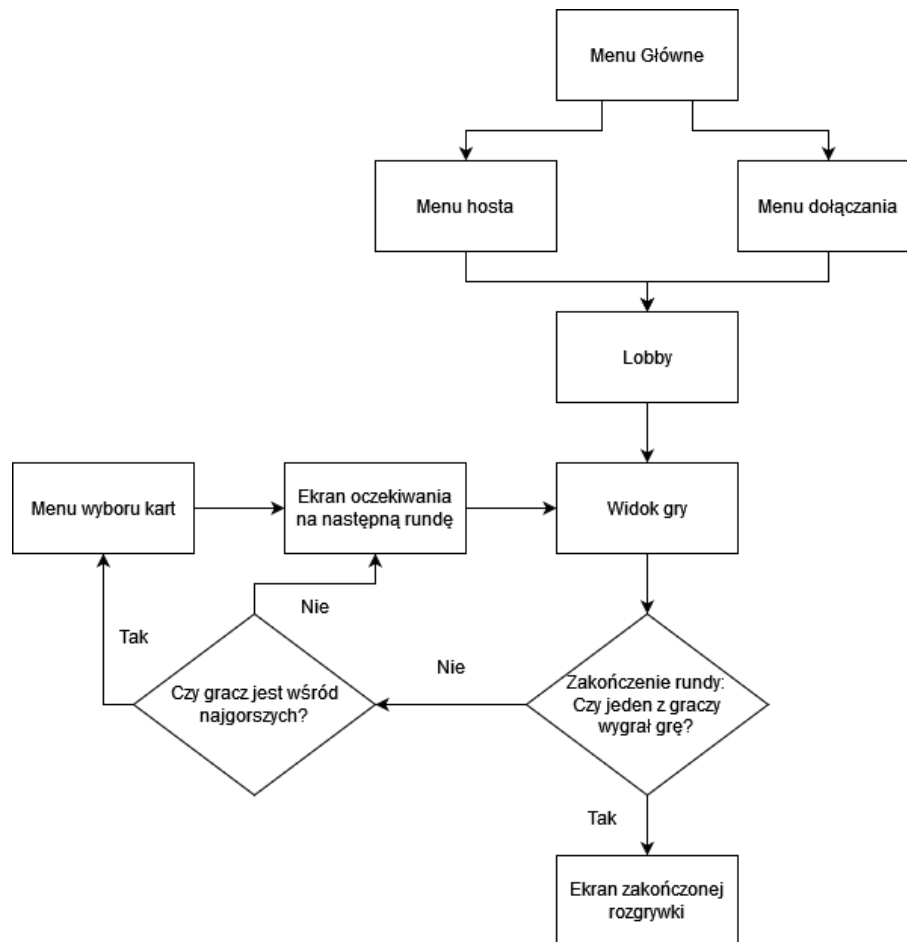
#### 5.1.1. Menu

Pierwszym ekranem widocznym dla gracza uruchamiającego grę będzie menu główne (rys. 5.2). Umożliwia ono przejście do menu hostowania gry, dołączania do gry lub wyłączenie aplikacji.

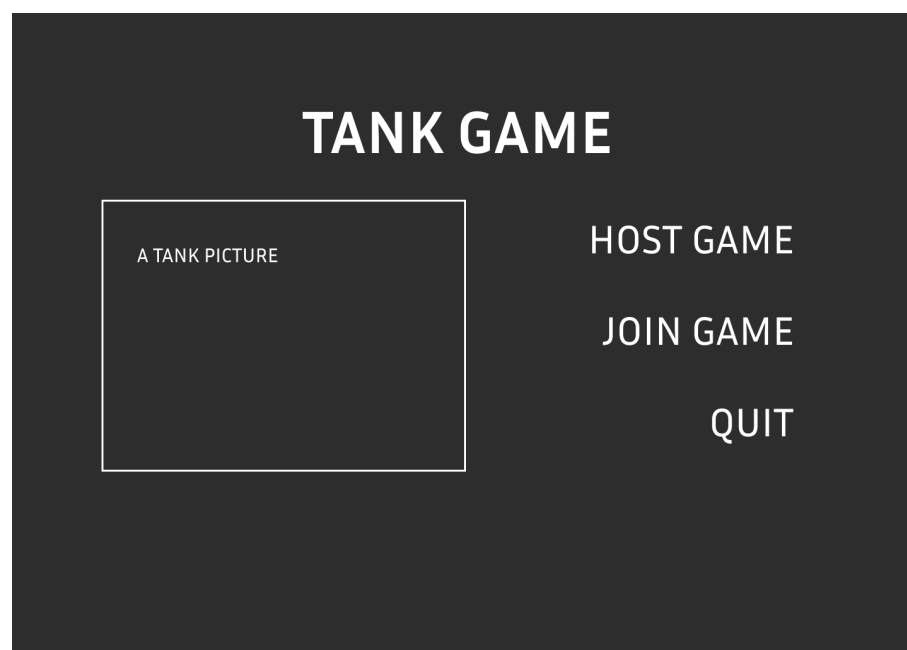
Menu hosta (rys. 5.3) pozwala graczowi wybrać podstawowe ustawienia gry - docelową liczbę punktów zwycięstwa, oraz własne ustawienia gracza: nazwę oraz kolor. Prezentuje również adres głównego interfejsu IP. Menu dołączania (rys. 5.4) pozwala podać adres IP serwera oraz dane gracza - nazwę oraz kolor.

Po stworzeniu lub dołączeniu do serwera prezentowany jest widok poczekalni (5.5). Widoczni są tu aktualnie połączeni z serwerem gracze. W tym widoku host ma możliwość rozpoczęcie gry. Gracze-klienci zamiast przycisku rozpoczynającego grę widzą jedynie komunikat o oczekiwaniu na rozpoczęcie gry przez hosta.

Po rozpoczęciu gry przez hosta wszyscy gracze przenoszeni są do widoku gry (rys 5.6). Na ekranie gracza główną część tego widoku zajmuje świat gry. W celu udostępnienia graczom niezbędnych informacji dodany został HUD[27] (ang. *Heads-Up Display*, wyświetlacz przezierny). Jest to metoda wyświetlania danych tak, aby użytkownik nie musiał przenosić wzroku w miejsce inne niż jego dotychczasowy widok. W tym przypadku HUD będzie wyświetlał w prawym dolnym rogu ekranu żywotność gracza, jego maksymalną pojemność magazynku oraz aktualnie dostępną liczbę pocisków. U góry ekranu wyświetlone będą paski żywotności pozostałych graczy wraz z ich nazwami.



Rys. 5.1: Diagram przejść między widokami.




Rys. 5.2: Projekt menu głównego.



The screenshot shows a dark-themed interface for hosting a game. At the top, the title 'HOST GAME' is displayed in large, bold, white capital letters. Below the title, the text 'Your IP: 192.168.0.1' is shown in a smaller white font. Further down, there is a label 'Victory Point Goal:' followed by the number '5' and a horizontal slider bar with a black knob positioned at the right end. Below this, the text 'Your Name:' is followed by a white rectangular input field. Underneath that, 'Your Colour:' is followed by a 'Colour picker' button with a light blue border and italicized text. At the bottom of the menu, there are two buttons: 'START GAME' in white text on a dark background, and 'BACK TO MENU' in white text on a slightly lighter dark background.

**HOST GAME**

Your IP: 192.168.0.1

Victory Point Goal: 5 

Your Name:

Your Colour:

START GAME

BACK TO MENU

Rys. 5.3: Projekt menu hosta.

The screenshot shows a dark-themed interface for joining a game. The title 'JOIN GAME' is at the top in large, bold, white capital letters. Below it, the text 'Server IP:' is followed by a white rectangular input field. Underneath, 'Your Name:' is followed by another white rectangular input field. Below that, 'Your Colour:' is followed by a 'Colour picker' button with a light blue border and italicized text. At the bottom, there are two buttons: 'START GAME' in white text on a dark background, and 'BACK TO MENU' in white text on a slightly lighter dark background.

**JOIN GAME**

Server IP:

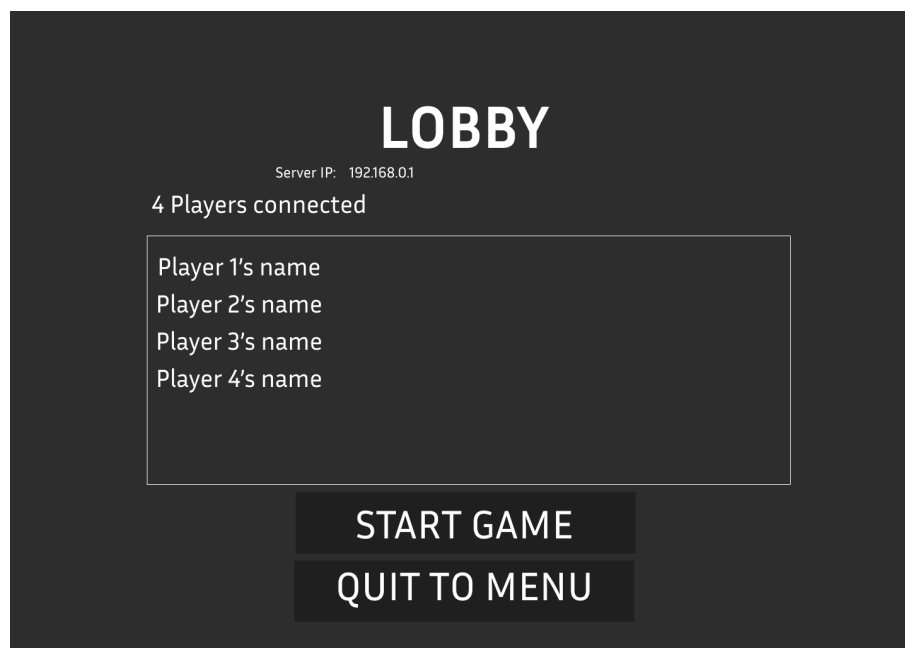
Your Name:

Your Colour:

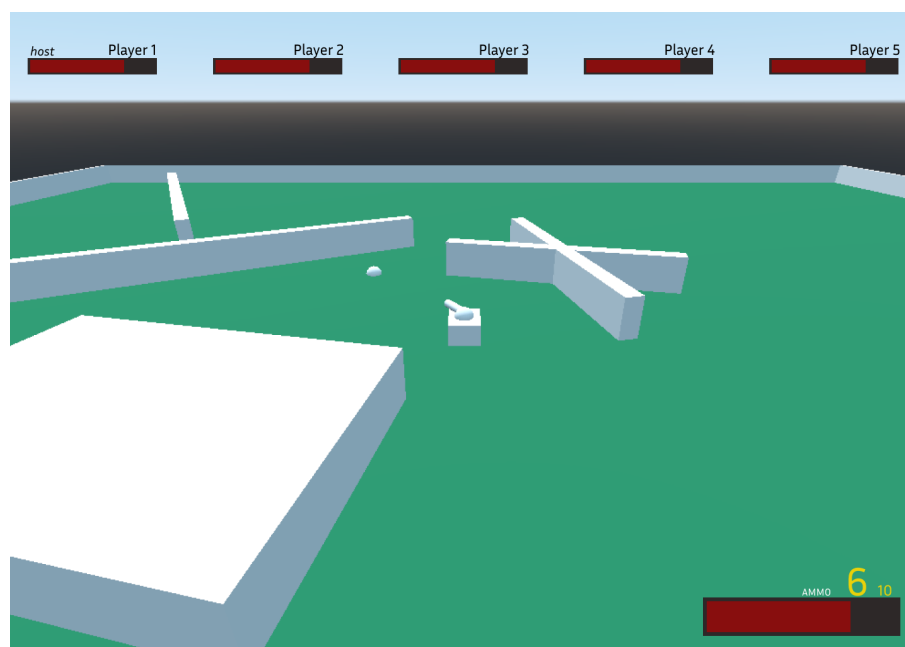
START GAME

BACK TO MENU

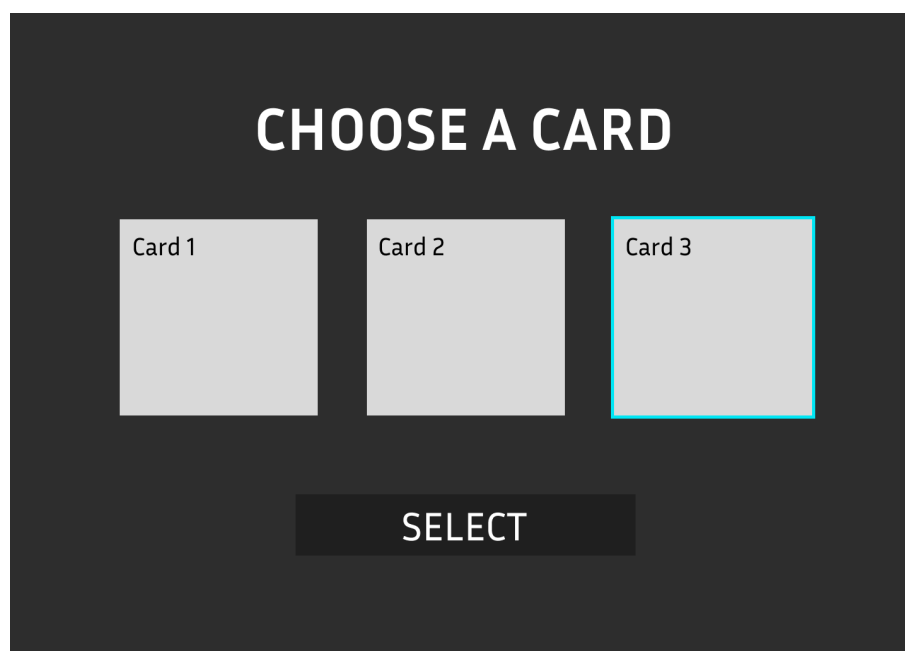
Rys. 5.4: Projekt menu dołączającego gracza.



Rys. 5.5: Projekt menu poczekalni.



Rys. 5.6: Projekt interfejsu HUD.



Rys. 5.7: Projekt menu wyboru kart.

Między rundami graczom wyświetlane są dwa ekrany. W przypadku gdy gracz należy do najgorszych dotychczas, przenoszony jest do ekranu wyboru kart (rys. 5.7). Prezentowane są mu tutaj trzy losowe karty - ich nazwy i opisy. Może wybrać jedną z nich a następnie potwierdzić wybór przyciskiem „Select”. Po wybraniu karty lub jeśli gracz nie był jednym z najgorszych prezentowany jest widok oczekiwania na pozostałych graczy (rys. 5.8). Widoczna jest na nim aktualna punktacja wszystkich graczy. Gdy wszyscy gracze wybiorą już karty oczekiwana jest jeszcze chwila, aby ostatni z graczy mógł zapoznać się z punktacją, po czym wszyscy gracze przenoszeni są ponownie do widoku rozgrywki.

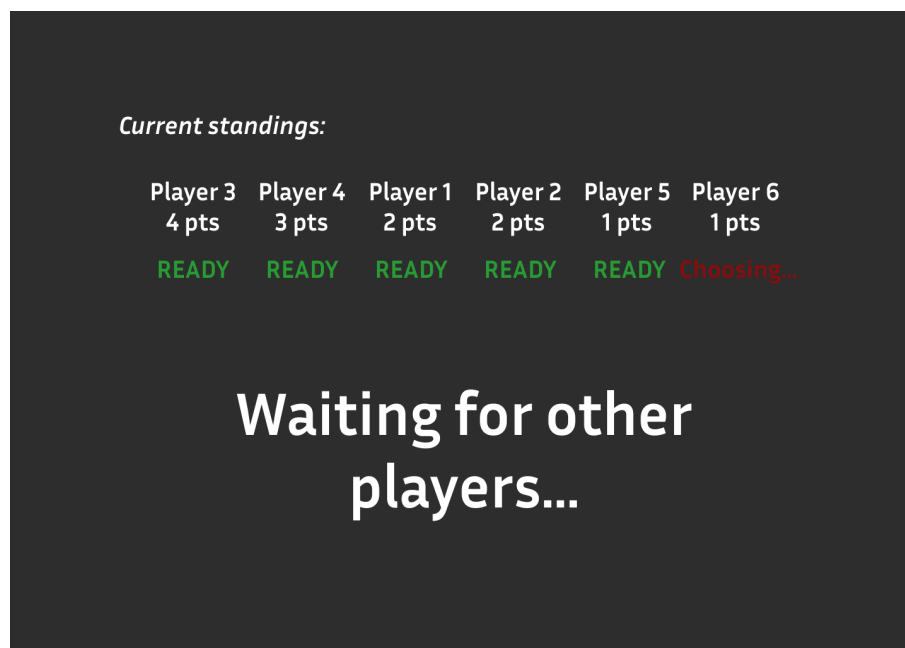
W przypadku, gdy na koniec rundy jeden z graczy ma docelową liczbę punktów wszyscy gracze przenoszeni są do widoku końca rozgrywki (rys. 5.9), gdzie można zapoznać się z ostateczną punktacją.

#### 5.1.2. Schemat sterowania

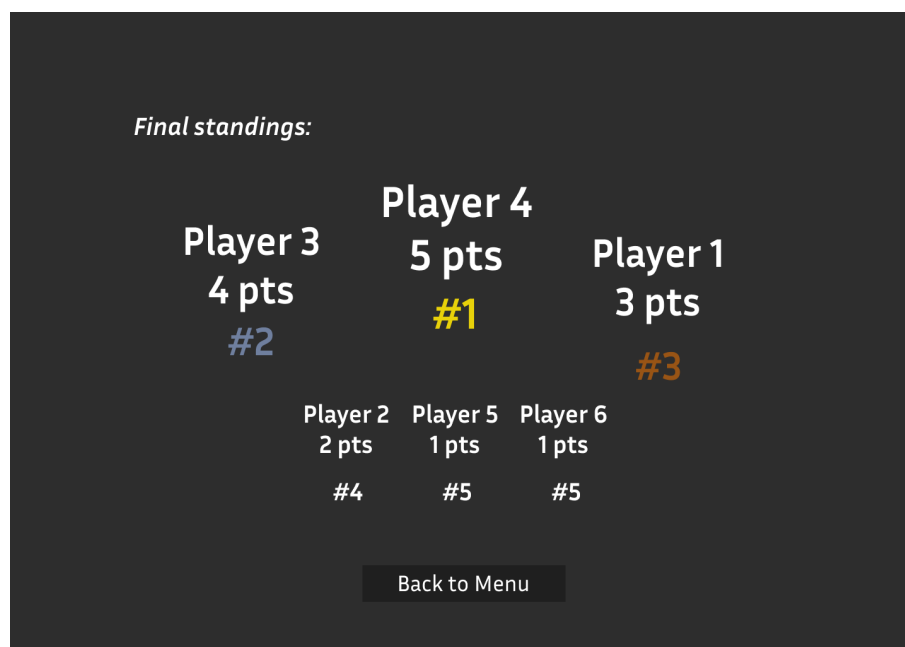
W ustawieniach projektu zostały wprowadzone akcje związane z wejściami gracza zgodne z tabelą 5.1 i założeniami z sekcji 3.1.1.

## 5.2. PROJEKT IMPLEMENTACJI MECHANIK

Opisane w sekcji 3.1 mechaniki zostaną zaimplementowane zgodnie z poniższymi projektami.



Rys. 5.8: Projekt menu oczekiwania między turami.



Rys. 5.9: Projekt ekranu po zakończeniu rozgrywki.

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.

### 5.2.1. Model danych postaci

Atrybuty opisane w tabeli 3.1 muszą być przechowywane jako pola obiektu postaci gracza. Ponadto ten obiekt przechowywać będzie dane związane z rozgrywką: nazwę gracza, jego wybrany kolor i aktualne punkty zwycięstwa. Do atrybutów postaci dodana została liczba odbić w celu ułatwienia implementacji kart.

Postać gracza przechowuje też własną prędkość jako wektor trójwymiarowy oraz prędkość kątową jako liczbę zmiennoprzecinkową. Te wartości wyrażone są jako zmiana na sekundę, w związku z tym będą musiały zostać w ten sposób wykorzystane podczas poruszania.

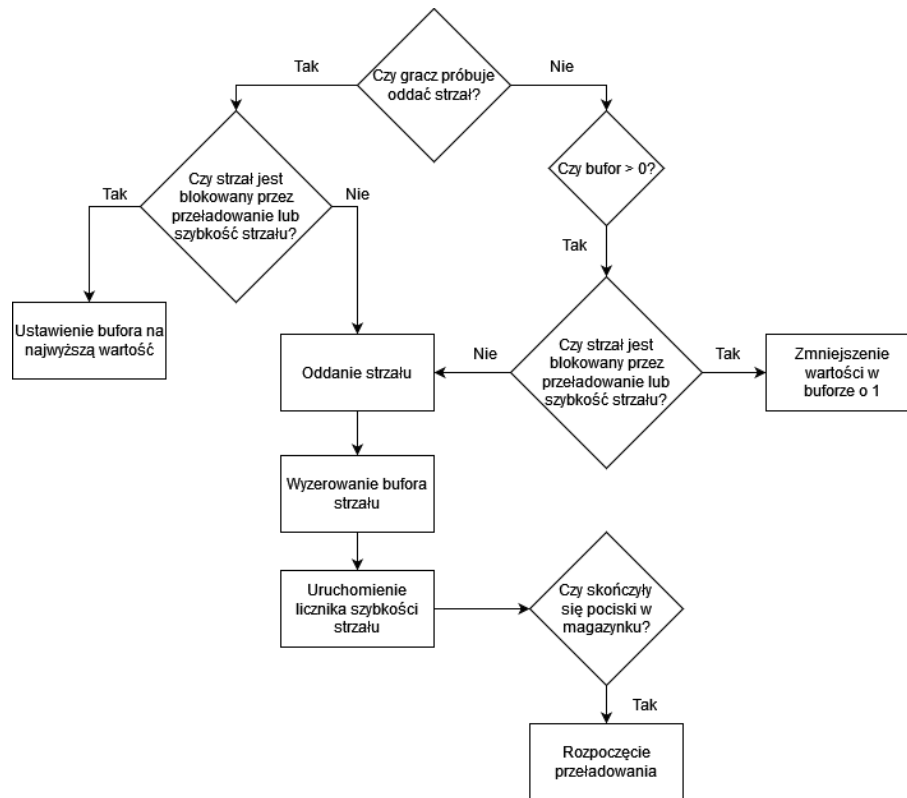
Wiele z atrybutów postaci reprezentuje wartości maksymalne. Analogicznie do nich wprowadzone zostaną wartości aktualne: aktualna liczba kul w magazynku i aktualne zdrowie. W celu wprowadzenia możliwości tymczasowej zmiany prędkości dla jednej z kart, do postaci dodana zostanie również aktualna maksymalna szybkość.

Jako bufor strzału dodane zostanie pole określające liczbę klatek, przez jakie sprawdzana będzie możliwość oddania strzału. Po nieudanej próbie oddania strzału zmienna zostanie ustawiona na maksymalną wartość - 3. W każdej klatce procesu fizycznego, w której nie zostanie oddany strzał, wartość bufora będzie zmniejszana o 1. Jeżeli wartość bufora była większa od 0 i oddanie strzału okaże się możliwe, strzał zostanie oddany a bufor - wyzerowany.

Ze strzałami związane są również zmienne warunkujące możliwość ich oddawania. Strzał nie jest możliwy w trzech sytuacjach: gdy postać przeładowuje broń, gdy postać właśnie oddała strzał oraz gdy gra jest zatrzymana (między rundami). Dla każdej z tych sytuacji dodane zostało oddzielne pole, określające możliwość oddania strzału.

Wartości atrybutów określające czas mają wpływ na czas oczekiwania konkretnych liczników czasu. Te liczniki zostaną dodane jako węzły do drzewa postaci oraz powiązane za pomocą sygnałów z konkretnymi metodami.

Po zmianie zdrowia oraz liczby pocisków (po strzale lub przeładowaniu) postać emituje sygnały niezbędne do aktualizowania interfejsów własnych i innych graczy.



Rys. 5.10: Schemat procesu oddawania strzału.

### 5.2.2. Poruszanie

Wejścia gracza związane z poruszaniem będą pobierane i interpretowane w taki sam sposób jak w przygotowanym prototypie.

### 5.2.3. Strzelanie

Mechanika strzelania zostanie rozszerzona o buforowanie, przeładowanie oraz szybkość strzału względem prototypu. Proces strzału został przedstawiony na schemacie blokowym na rysunku 5.10.

#### 5.2.3.1. Wydarzenia po trafieniu

Wydarzenia po trafieniu będą przechowywane jako obiekty implementujące wzorzec projektowy Komendy[14]. Taka implementacja pozwoli na potencjalne rozszerzenie wydarzeń o dodatkowe wartości lub pamięć.

Stworzone zostaną dwa rodzaje komend - podstawowe i opóźnione. Gracz będzie przechowywał listę takich obiektów i przekazywał ją tworzonemu pociskowi. Po trafieniu pocisk będzie uruchamiał wszystkie komendy podstawowe, następnie ranił trafionego gracza, wykonywał odbicia, a dopiero ostatecznie, uruchamiał komendy opóźnione. Taki podział jest niezbędny dla implementacji niektórych kart.

#### 5.2.4. Zdrowie

Postać gracza posiada określoną maksymalną żywotność oraz w każdym momencie - określoną aktualną żywotność. Po trafieniu przez pocisk zadaje on określone obrażenia, które odejmowane są od aktualnej żywotności. Trafienie rozpoczyna odliczanie licznikiem przed leczeniem. Po zakończeniu tego odliczania rozpoczyna się proces samoleczenia. Uruchamiany zostaje powtarzający się licznik leczenia. Po każdorazowym jego zakończeniu postać zostaje uleczona o konkretną wartość zależną od atrybutów postaci.

#### 5.2.5. Rundy

Po rozpoczęciu rozgrywki przez hosta rozpoczyna się pierwsza runda. Każda z rund rozpoczyna się rozesłaniem odpowiedniego sygnału lokalnie oraz zdalnie przez hosta. Każdy z graczy ustawia wtedy odpowiednie wartości początkowe.

Po ustawieniu tych wartości wszyscy gracze ustawiani są w odpowiednich, predefiniowanych punktach na mapie. Pozycje na mapie są ustalone jako część drzewa poziomego, ale przypisywane są graczom w losowej kolejności, dzięki czemu pozycja graczy nie będzie identyczna w każdej rundzie.

W czasie rozgrywki host zbiera i zapisuje dane na temat wyeliminowanych graczy. Gdy zostaną wyeliminowani wszyscy poza jednym, runda zostaje zakończona a host rozsyła odpowiedni sygnał lokalnie i zdalnie.

Wraz ze zdalnym poleceniem zakończenia rundy rozsyłana jest lista najgorszych graczy oraz flaga określająca, czy gra została zakończona. Każdy z klientów oraz serwer analizuje otrzymane dane. Jeżeli flaga wskazuje na zakończenie gry, wyświetlany jest ekran końcowy. W przeciwnym wypadku, jeżeli gracz jest jednym z najgorszych wyświetlany jest ekran wyboru kart. W przeciwnym wypadku graczowi wyświetlany jest ekran oczekiwania.

Miedzy rundami każdy z graczy wybierających karty wysyła do serwera informację o zakończeniu wyboru. Gdy każdy z graczy wybierających wyśle już taki sygnał, rozgrywka jest kontynuowana w kolejnej rundzie.

#### 5.2.6. Karty ulepszeń

Karty ulepszeń, zgodnie z koncepcją mogą zmieniać atrybuty statyczne i dodawać komendy do pocisku. Karta będzie składała się z następujących elementów:

- **Nazwa** Krótka nazwa karty, reprezentująca tematycznie jej działanie.
- **Opis** Opis działania karty. Nie jest to szczegółowy opis a jedynie wskazanie graczowi jakie *najważniejsze* cechy zmienia karta.
- **Zmiany atrybutów** Słownik reprezentujący zmiany atrybutów postaci. Składa się z par klucz - wartość. Kluczami są nazwy zmienianych atrybutów. Przypisaną wartością jest lista dwuelementowa w której pierwszym elementem jest sposób, w jaki zmieniana jest wartość a drugim - wartość o jaką zmieniany jest atrybut.

— **Komendy po trafieniu** Lista komend dodawanych do listy komend pocisków gracza.

Zmiany atrybutów opisywane są dwoma możliwymi sposobami: mnożeniem lub dodawaniem. Oznacza to, że wartość zmiany atrybutu może zostać zastosowana jako, odpowiednio, pomnożenie aktualnej wartości przez wartość zmiany lub dodanie zmiany do aktualnej wartości.

### 5.3. SYSTEM SIECIOWY

Implementowany system sieciowy wymaga znacznej rozbudowy względem przygotowanego w ramach prototypu. Głównymi aspektami tego wprowadzanego systemu będzie synchronizacja zmian scen, w tym scena poczekalni, synchronizacja danych graczy oraz synchronizacja wydarzeń takich jak poruszanie czy trafiać pocisków.

#### 5.3.1. Poczekalnia

Po rozpoczęciu gry przez hosta gra uruchamiany jest poczekalni. Na tym etapie do gry przyjmowani są kolejni gracze.

Gdy do serwera podłączy się kolejny gracz sygnał o tym rozsyłany jest do wszystkich pozostałych graczy. Gracz podłączający się również otrzymuje sygnały o podłączeniu wszystkich graczy połączonych dotychczas z serwerem.

Gdy gracz dostanie informacje o połączeniu z nowym graczem przesyła temu graczowi swoje dane - nazwę oraz wybrany kolor. Te informacje są zapisywane na każdym z komputerów i prezentowane na ekranie poczekalni w liście.

Gdy wszyscy oczekiwani gracze połączą się już z serwerem host podejmuje decyzję o rozpoczęciu gry poprzez wciśnięcie odpowiedniego przycisku. Od tego momentu nie jest możliwe dołączenie kolejnych graczy. Rozpoczęcie gry nie jest możliwe, gdy żaden inny gracz nie jest połączony z serwerem.

#### 5.3.2. Przepływ danych w grze

W czasie rozgrywki niezbędna jest synchronizacja danych pomiędzy aplikacjami. Wyodróżnione zostały tu dwa rodzaje wydarzeń: interpretowane przez mistrza węzła oraz interpretowane przez serwer.

Wydarzenia interpretowane przez mistrza to takie, w których prawdziwe dane na ich temat posiada mistrz danego węzła. Rozsyła on takie dane do marionetek aby mogły one zaktualizować swoje dane. Do takich wydarzeń należą, w przypadku postaci gracza: poruszanie, strzał, synchronizacja nazwy i koloru oraz zmiana zdrowia. Z perspektywy pocisków do takich wydarzeń zaliczamy ruch i niszczenie, na przykład pod wpływem zderzenia.

Wydarzenia interpretowane przez serwer to te kluczowe dla prowadzenia rozgrywki, w związku z czym uznaje się, że to serwer jest „sędzią” w ich przypadku. Ze względu



na opóźnienia w komunikacji przez sieć takie wydarzenia mogą zachodzić w różnych momentach na różnych komputerach. W takich sytuacjach jedynie wydarzenia z serwera są synchronizowane. Takimi wydarzeniami są zadawanie obrażeń, eliminowanie graczy i wybór karty.

## 6. IMPLEMENTACJA

### 6.1. INTERFEJS UŻYTKOWNIKA

Z wykorzystaniem wbudowanych w Godot węzłów zbudowane zostały interfejsy zaprojektowane w sekcji 5.1. Problematyczne okazało się dostosowanie układu tak, aby przy zmianie rozmiaru okna elementy nie nachodziły na siebie. Dodatkowo, przy skalowaniu okna nie ma możliwości dynamicznego dostosowania wielkości czcionki do wielkości kontrolki w której się znajduje. Było to szczególnym problemem podczas testowania oprogramowania, kiedy często należało uruchamiać kilka instancji na jednym ekranie. W domyślnym użytkowaniu gry taki problem nie powinien występować, ponieważ gra będzie rozgrywana na pełnym ekranie.

W celu ustawienia kontrolki w odpowiednich miejscach i w odpowiednich relacjach wykorzystane zostały węzły kontenerowe: `VBoxContainer`, `HBoxContainer` i `GridContainer`.

Innym problemem napotkanym podczas implementacji okazało się przełączanie scen. Problem był zauważalny między rundami, gdzie menu wyboru kart nie było wyświetlane mimo, że powinno było. Problemem okazał się być fakt, iż przełączenie sceny nie dzieje się od razu. Jest ono odsunięte w czasie na klatkę po wykonaniu całej funkcji. Dodatkowo zmiana sceny nie powoduje zakończenia funkcji. Doprowadziło to do sytuacji, w której scena była zmieniana wielokrotnie w jednym wykonaniu. Po odnalezieniu powodu takiego zachowania w dokumentacji rozwiązaniem było opuszczanie funkcji słowem kluczowym `return` po zamianie sceny.

HUD gracza podzielono na podelementy. Oddzielnie przygotowano scenę informacji własnych gracza, oddzielnie zaś scenę informacji innych graczy. Sceny te połączono z danymi na temat żywotności i liczby pocisków poszczególnych graczy przy pomocy sygnałów. Sceny z informacjami innych graczy tworzone i dodawane są do gry w momencie rozesłania sygnału rozpoczęcia rozgrywki. Pełna scena HUD dodana została do sceny gracza. Sprawia to, że jest ona stale dostępna w oknie gry, jednak w przypadku uruchomienia sceny menu, HUD jest przez nią zasłaniany.

W menu kart, same karty również zostały przygotowane jako oddzielne sceny. Pozwala to na łatwe generowanie widoku z różnymi kartami, wyglądającymi tak samo. Jako tło karty ustawiony został węzeł `TextureRect`. Umożliwia to w przyszłości przygotowanie różnorodnych grafik dla różnych kart. W aktualnej implementacji. Podjęte zostały również

próby umieszczenia na karcie informacji na temat modyfikowanych atrybutów, jednak potencjalnie duża możliwa ich liczba doprowadziła do decyzji o ukryciu ich przed graczem.

## 6.2. POSTAĆ GRACZA

Model postaci został przygotowany w programie Blender (rys. 6.1a). Ze względu na prostsze wykonanie, sam model wykonano w stylu *low poly* (ang. mało wielokątów). Przygotowane zostały również materiały: pokrywający łufę i gąsienice oraz dwa pokrywające karoserię i wieżyczkę. W grze materiały kolorowe zostają zmodyfikowane tak, aby ich kolorem bazowym był ten wybrany przez gracza (rys. 6.1b). W celu wprowadzenia takiej modyfikacji należało zduplikować odpowiednie materiały (listing 6.1). Pominięcie tego kroku powodowało, że wszystkie modele w grze zmieniały kolor na ten sam. Dzieje się tak, ponieważ Godot domyślnie wykorzystuje zasoby jako współdzielone aby oszczędzać pamięć.

Listing 6.1: Kod zmieniający kolor materiału postaci gracza.

```
func set_player_colour(new_colour: Color):
    player_colour = new_colour

    var body_mesh: Mesh = \%Model/Body/Body.mesh
    var tower_mesh: Mesh = \%Model/Head/Tower.mesh
    var barrel_mesh: Mesh = \%Model/Head/Barrel.mesh

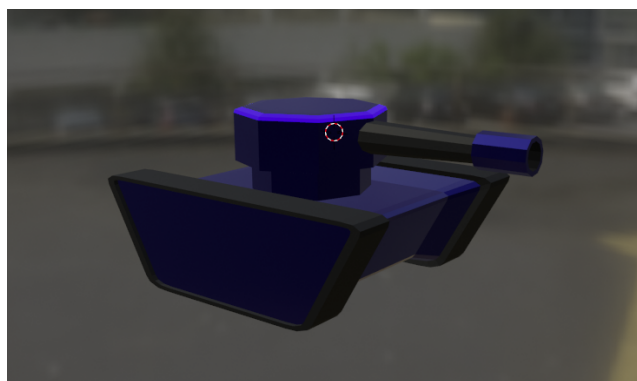
    var body_material: SpatialMaterial
        = preload("res://Resources/Models/BodyMaterial.material")
        .duplicate()
    var light_ring_material: SpatialMaterial
        = preload("res://Resources/Models/BodyLight.material")
        .duplicate()

    body_material.albedo_color = new_colour
    light_ring_material.albedo_color = new_colour
    light_ring_material.emission = new_colour

    body_mesh.surface_set_material(0, body_material)
    tower_mesh.surface_set_material(0, body_material)
    tower_mesh.surface_set_material(1, light_ring_material)
    barrel_mesh.surface_set_material(0, body_material)

    if is_network_master():
        rset("puppet_player_colour", player_colour)
```

Scena gracza jest rozszerzeniem węzła `KinematicBody`. Udostępnia on wiele metod związanych z ruchem i kolizjami, na przykład główną metodę wykorzystywaną do



(a) Model w programie Blender.



(b) Model o programowo zmienionych barwach.

Rys. 6.1: Model czołgu.

poruszania postaci: `move_and_slide`. Pozwala ona przemieszczać obiekty interpretując odpowiednio ich kolizje z innymi - przesuwając ciało wzdłuż płaszczyzny kolizji.

Drzewo sceny gracza zawiera również inne istotne węzły (rys. 6.2).

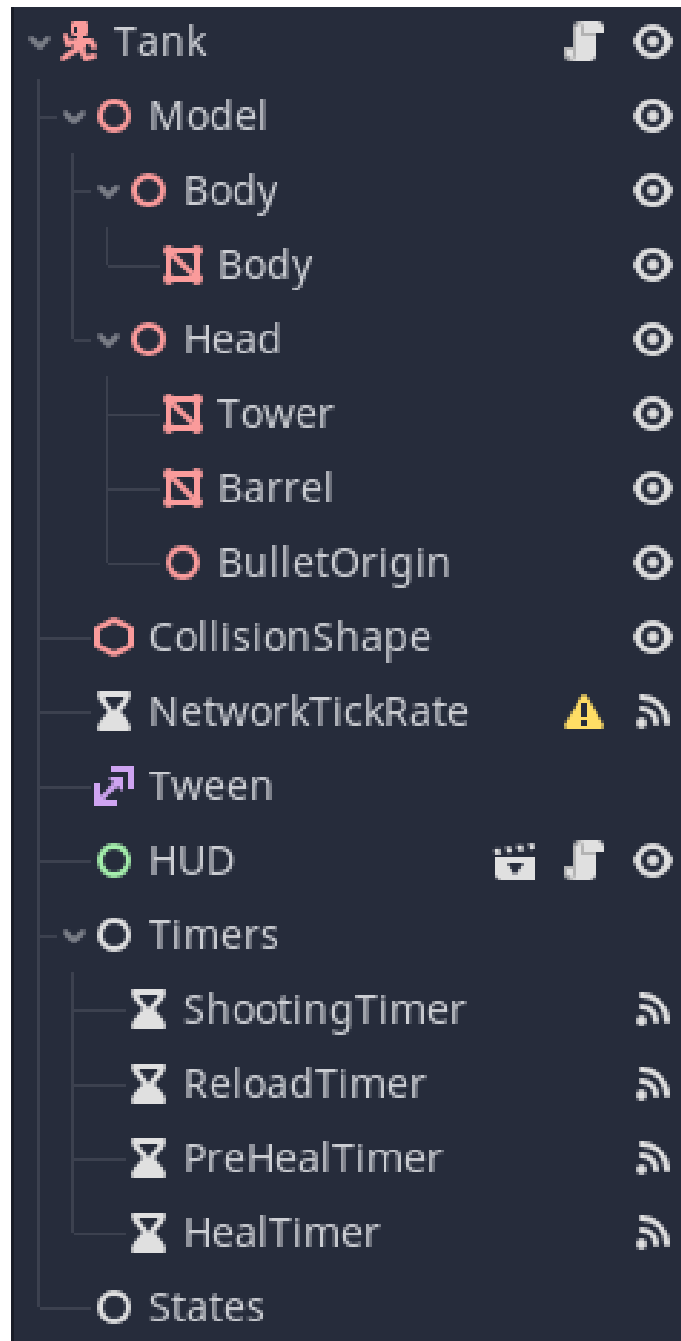
- `Model` - Model postaci importowany z Blendera. Podzielony jest na trzy elementy wizualne: `Body`, reprezentujący główny kadłub, `Tower`, reprezentujący wieżyczkę i `Barrel`, reprezentujący lufę. Dwa ostatnie elementy są zgrupowane, ponieważ poruszają i obracają się wspólnie. Wraz z nimi zgrupowany jest `BulletOrigin` reprezentujący punkt w którym pociski są tworzone. Początkowo był on dzieckiem lufy, jednak wtedy aplikowane były do niego przekształcenia takie jak skalowanie i obroty, przeniesione z Blendera. Wpływały one na tworzone pociski, dlatego też został odłączony od lufy.
- `CollisionShape` Kształt określający gdzie obiekty mogą wchodzić z czołgiem w kolizję.
- `NetworkTickRate` Licznik czasu, na którego zakończenie wysyłane są dane na temat pozycji gracza. Odmierza 0.03 sekundy, co oznacza nieco ponad 33 razy na sekundę. Ze względu na działanie silnika zegar nie może uruchomić sygnału zakończenia częściej niż raz na klatkę, w związku z tym przy spowolnieniu aplikacji komunikacja internetowa również będzie odbywać się wolniej.
- `HUD` Opisany w sekcji 5.1.1 element interfejsu użytkownika.
- `Timers` Zbiór liczników czasu potrzebnych w strzelaniu i leczeniu. `ShootingTimer` blokuje zbyt częste strzały. `ReloadTimer` odmierza czas przeładowania. `PreHealTimer` odmierza czas między otrzymaniem obrażeń a leczeniem. `HealTimer` określa jak często będzie leczenie po jego rozpoczęciu.
- `States` Węzeł zbierający w sobie wszelkie stany nadane graczowi jako efekt trafienia zmodyfikowanymi pociskami.

## 6.3. IMPLEMENTACJA MECHANIK

### 6.3.1. Strzelanie

Do procesu strzelania dodane zostały odpowiednie elementy zgodnie z diagramem 5.10. Implementacja opiera się na licznikach czasu, które po zakończeniu odliczania zmieniają wartości odpowiednich zmiennych. Kod przedstawiający decyzje o strzale z uwzględnieniem buforowania został przedstawiony w listingu 6.2.

Po dodaniu możliwości zmiany szybkości pocisku kartami dotychczasowa implementacja przemieszczania okazała się niewystarczająca. Pocisk w implementacji prototypu był przenoszony na pozycję będącą przemieszczeniem o odpowiednio przemnożony wektor prędkości. W przypadku znacznego przyspieszenia pocisku jest to niewystarczająca implementacja. Prowadzi ona do „przenikania” pocisków przez ściany. Kolizje między kulkami nie są wykrywane, a pocisk może zostać przeniesiony bez jej wykrycia. W tym



Rys. 6.2: Drzewo sceny postaci gracza.

celu węzeł główny pocisku zmieniono z `Area` na `KinematicBody`. Umożliwiło to wykorzystanie funkcji `move_and_collide`. Przemieszcza ona obiekt o podany wektor, wykrywa jednak kolizje na drodze. Zwrócona przez funkcję kolizja może zostać w prosty sposób zinterpretowana pod kątem kolidującego obiektu, ruchu obu obiektów ale również punktu kolizji czy wektora normalnego płaszczyzny kolidującej w tym punkcie. Te dane okazały się przydatne w implementacji kolejnych kart znacznie upraszczając tworzone zachowania.

Listing 6.2: Implementacja decyzji o dokonaniu strzału.

```
if Input.is_action_just_pressed("MainAction") and can_shoot:
    if has_just_shot or is_reloading:
        shoot_buffer = SHOOT_BUFFER_FRAMES
    else:
        shoot()
elif shoot_buffer > 0: #shot buffering
    if is_reloading or has_just_shot:
        shoot_buffer -= 1
    else:
        shoot()
```

Do pocisku została dodana liczba pozostałych odbić. Po trafieniu przeszkody, jeżeli liczba odbić jest większa od zera zostaje ona zdekrementowana a następnie dochodzi do odbicia. Implementacja odbicia wykorzystuje wektor normalny płaszczyzny kolidującej. Jest on przekazywany do metody `Vector3.reflect`, która odbija wektor od zadanej wektorem normalnym powierzchni. W ten sposób zostaje obrócona prędkość pocisku. Następnie pocisk obracany jest w kierunku wskazywanym przez prędkość metodą `look_at`.

### 6.3.2. Komendy

Wydarzenia po trafieniu zostały zaimplementowane jako komendy. W GDScript funkcje nie są obiektami jak w wielu innych językach, nie można również wyznaczyć wskaźnika na funkcję. W celu przechowywania funkcji jako zmiennej w obiekcie wykorzystano klasę `FuncRef`. Pozwala ona na stworzenie referencji na funkcję w *konkretnym obiekcie*, która następnie może zostać wywołana. W związku z tym, aby takie funkcje móc wywoływać niezbędne jest zapewnienie istnienia obiektu, na którym będą wywoływane. W tym celu zostało stworzone repozytorium komend. W klasie `CommandsRepository` zdefiniowane są wszystkie niezbędne funkcje oraz tworzone są wszystkie dostępne w grze komendy. Klasa ta zostaje autoładowana na początku działania aplikacji, jej instancja jest więc dostępna globalnie.

Określony został zestaw argumentów niezbędnych do stworzenia komend. Są to: pocisk wywołujący komendę, obiekt uczestniczący w kolizji oraz punkt kolizji. Te argumenty będą przekazywane przy każdym wywołaniu funkcji z komend. Możliwe jest dodanie

kolejnych lub zmiana aktualnych argumentów poprzez stworzenie klasy dziedziczącej po `BaseCommand`, jednak wymagałoby to zmiany kodu obsługującego wywoływanie komend.

### 6.3.3. Karty

Została stworzona klasa bazowa dla kart, definiująca ich cztery atrybuty: nazwę, opis, zmiany atrybutów i listę komend. Te atrybuty zostały dokładniej opisane w sekcji 5.2.6. Została również dodana metoda aplikująca wszystkie zmiany do gracza, przedstawiona w listingu 6.3

Listing 6.3: Metoda aplikująca zmiany atrybutów gracza.

```
func attach_to_player(player: KinematicBody):
    for key in attribute_changes.keys():
        if player.get(key) != null:
            match attribute_changes[key][0]:
                "*":
                    player[key] *= attribute_changes[key][1]
                "+":
                    player[key] += attribute_changes[key][1]
    if player.get("bullet_on_hit") != null:
        player["bullet_on_hit"].append_array(on_hit)
    if player.get("cards") != null:
        player["cards"].append(self)
```

Dla kart, podobnie jak dla komend, stworzone zostało autoładowane repozytorium `CardsRepository`. Tworzone są w nim obiekty każdej z kart. Udostępnia ono również metody do losowania kart, w szczególności do wylosowania trzech kart w celu wyboru między rundami.

## 6.4. OBIEKTY GLOBALNE

W ustawieniach projektu skonfigurowane zostały obiekty autoładowane odpowiedzialne za zarządzanie różnymi aspektami rozgrywki. W poprzednich sekcjach opisane zostały repozytoria komend i kart.

Kolejnym obiektem globalnym jest `GameState` - odpowiedzialny za zarządzanie stanem gry. Zapisane są w nim najważniejsze dane użytkowników potrzebne do prezentacji w menu. Zapisywane są tam również postępy w rozgrywce - docelowa liczba punktów zwycięstwa, aktualna liczba punktów każdego z graczy, lista wyeliminowanych w aktualnej rundzie graczy oraz zwycięzca ostatniej rundy. Udostępnione są tu również metody pozwalające zarządzać postępem rund - rozpoczynanie i zakańczanie rund, znajdowanie najsłabszych graczy i zwycięzcy oraz zbieranie danych na temat wybieranych przez graczy kart.

Skrypt globalny `Network` odpowiada za zarządzanie połączeniami sieciowymi. Udostępnia metody pozwalające tworzyć serwer oraz do niego dołączać, odłączać się od sieci.



```
*** Run Summary ***
All tests passed

Totals
Scripts:          3
Passing tests     13
Failing tests     0
Risky tests       0
Pending:          0
Asserts:          40 of 40 passed

Warnings/Errors:
* 1 Warnings.

13 passed 0 failed. Tests finished in 0.4s
```

Rys. 6.3: Wyniki uruchomienia testów automatycznych.

Umożliwia również nazywanie obiektów sieciowych w sposób synchronizowany pomiędzy klientami. Skrypt ten jest również połączony z najważniejszymi sygnałami sieciowymi emitowanymi przez drzewo gry.

Obiekt `Global` odpowiedzialny jest za zarządzanie najważniejszymi obiektami. Udo-  
stępnia też wiele metod upraszczających tworzenie nowych węzłów.

Węzeł globalny `PersistentNodes` nie ma przypisanego skryptu. Jako jego dzieci zapisywane są tworzone w trakcie gry obiekty dynamiczne, których częste tworzenie może znacznie obciążać komputer. Tutaj zapisywane są obiekty postaci graczy i pociski.

## 6.5. TESTY

Z wykorzystaniem narzędzia GUT stworzone zostały testy skryptów globalnych. Skryp-  
ty `Global`, `Network` i `GameState` zostały przetestowane integracyjnie i jednostkowo. Su-  
marycznie zostało napisanych 13 testów składających się z 40 asercji (rys. 6.3).

Przeprowadzone zostały również testy manualne zarówno wielu instancji aplikacji na  
jednym komputerze jak i połączenia po sieci lokalnej LAN. W wyniku testów odkryto  
wiele błędów, większość z których udało się naprawić. Przykładem takiego błędu był błąd,  
który sprawiał, że połączenia serwera oraz dane gry nie były usuwane po zakończeniu  
gry. Z tego powodu kolejne uruchomienie gry powodowało wskazanie, że większa liczba  
graczy jest podłączona do serwera niż w rzeczywistości. Naprawienie tego błędu wymagało  
pełnego czyszczenia stanu gry po każdym zakończeniu rozgrywki.

## **7. PODSUMOWANIE**

### **7.1. WNIOSKI**

Aplikacja została w pełni zaimplementowana zgodnie z wymaganiami i koncepcją. Korzystanie z silnika Godot znacznie ułatwiło pracę nad wieloma aspektami gry. Wprowadzenie systemu sieciowego było o wiele prostszym zadaniem niż wyglądało. Mimo to w przypadku dalszego rozwoju warto byłoby dokładniej zaplanować działanie tego systemu. Szczególnie w przypadku znacznego rozrostu funkcjonalności gry niezbędne może okazać się zbudowanie kodu sieciowego od nowa tak, aby podział odpowiedzialności był oczywisty. W przygotowanej implementacji zdarza się, że podobne zachowanie aplikacji wywoływane jest z wielu powodów.

Rozpoczęcie pracy od przygotowania koncepcji i prototypu pomogło głębiej zrozumieć działanie silnika Godot oraz jego interfejsów sieciowych. Błędem było jednak budowanie dalszych rozwiązań bezpośrednio na implementacji prototypu. Lepszym rozwiązaniem byłoby zbudowanie ostatecznej implementacji niezależnie, korzystając jedynie ze zdobytej wiedzy.

Język GDScript wykorzystany w projekcie jest prostym, czytelnym i bardzo łatwym w obsłudze narzędziem. Autor projektu odczuł jednak, że brak silnego typowania oraz nietypowa implementacja obiektowego paradygmatu programowania opóźniały pracę i wprowadzały błędy, których można by uniknąć w innych językach. Ponadto korzystanie z innych języków programowania wiązałoby się z możliwościami korzystania z ich, o wiele bogatszej dokumentacji i dokładniejszej oraz materiałów edukacyjnych dostępnych w Internecie.

### **7.2. ZNANE PROBLEMY**

Istnieje znany problem którego dotychczas nie udało się rozwiązać. W czasie rozgrywki co jakiś czas pojawiają się błędy kamery, obraca się ona i przesuwają w chaotyczny sposób. Możliwe, że jest to związane z aktualizacjami pozycji spowodowanymi opóźnieniem połączenia internetowego, ciężko jest jednak sprawdzić tego typu błędy.

### 7.3. DALSZY ROZWÓJ PROJEKTU

Istnieje wiele możliwości dalszego rozwoju projektu. Podzielić je można na dwie główne grupy. Rozwój „techniczny” jest związany z poprawą jakości technologicznej części projektu oraz dodaniem dodatkowych funkcjonalności. Rozwój „kreatywny” wiąże się z poszerzeniem sfery *game designu* oraz audiowizualnej.

#### 7.3.1. Warstwa techniczna

Wśród możliwości technicznych większość ma związek z usprawnieniem kodu sieciowego. Jednym z możliwych usprawnień byłoby stworzenie dedykowanej aplikacji serwerowej, tj. takiej bez warstwy wizualnej, uruchamianej na przykład jako polecenie terminala. Taka aplikacja nie mogłaby być związana z żadną postacią gracza. Stworzenie takiej wersji aplikacji jest możliwe z wykorzystaniem Godota. Może to być oddzielna aplikacja lub inaczej uruchamiana modyfikacja przygotowanego projektu.

Alternatywą mogłoby być wprowadzenie obsługi zewnętrznej usługi serwerów. Przykładem gotowego rozwiązania sieciowego którego obsługę możnaby wprowadzić są serwery Steam. Istnieje rozszerzenie Godota pozwalające na wprowadzenie ich obsługi. Wymagałoby to jednak najpewniej zupełnej przebudowy kodu sieciowego.

Kolejnym aspektem technicznym możliwym do dodania jest wprowadzenie obsługi kontrolerów oraz własne mapowanie wejść przez gracza. Oba z tych rozwiązań są wspierane przez silnik, istnieje więc możliwość prostego wprowadzenia takich rozszerzeń.

Aktualnie aplikacja obsługiwana jest jedynie w języku angielskim. Wartym rozważenia rozszerzeniem mogłoby być wprowadzenie tłumaczeń na inne języki. Wprowadzanie tłumaczeń jest wspierane przez Godot.

W grze rozwinięta została w pewnym stopniu warstwa wizualna, wiele mogłoby jednak dodać również wprowadzenie obsługi dźwięków takich jak odgłosy czołgów graczy, strzałów czy eksplozji. Ponadto, warstwa wizualna mogłaby zostać rozbudowana o animacje. Wprowadzenie cieni i realistycznych odbić w obiektach metalicznych zostało wypróbowane, jednak wygląd takich rozwiązań był niezadowolający.

Istotnym elementem po dodaniu powyższych rozszerzeń jest wprowadzenie menu ustawień, w którym gracz mógłby zmieniać aspekty działania aplikacji. Ponadto do gry powinno zostać dodane menu pauzy, w którym gracz mógłby zobaczyć listę aktualnie posiadanych kart.

#### 7.3.2. Warstwa kreatywna

Ciekawym rozszerzeniem i urozmaicheniem gry jest dodanie dodatkowych trybów rozgrywki. W innych trybach gracze dostawaliby punkty za osiągnięcie innych celów, mogliby również łączyć się w zespoły i współpracować.

Oczywistym dodatkiem do gry jest również poszerzenie bazy dostępnych kart. Korzystając z aktualnie dostępnych systemów możliwe jest stworzenie wielu ciekawych kombinacji. Dodatkowym aspektem może być również dodanie kolejnej akcji, na przykład uruchamianej drugim przyciskiem myszy. Byłaby ona uruchamiana w punkcie, w którym znajduje się gracz. Taka mechanika również korzystałaby z komend.

Innym sposobem na urozmaicenie gry jest rozwinięcie warstwy wizualnej. Jedną z możliwości jest tu wprowadzenie dodatkowych poziomów. Każda runda mogłaby wtedy rozgrywać się na losowym poziomie o różnych charakterystykach. Dodanie do poziomów zagrożeń środowiskowych również mogłoby rozszerzyć możliwości tworzenia kolejnych poziomów.

Kolejnym artystycznym aspektem możliwym do rozwinięcia jest wprowadzenie grafik do menu, ekranów i jako ikonę gry. Ponadto wprowadzenie kolejnych możliwości personalizacji modeli również może urozmaicić rozgrywkę.

## BIBLIOGRAFIA

- [1] Juan Linietsky, Ariel Manzur. Strona główna projektu godot. <https://godotengine.org/>. Dostęp 23.11.2022.
- [2] 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.
- [3] Jason Gregory. *Game Engine Architecture*. A K Peters, Ltd., 2009.
- [4] Unity Technologies. Strona główna unity. <https://unity.com/>. Dostęp 11.12.2022.
- [5] Michelle Menard, Bryan Wagstaff. *Game Development with Unity*. Cengage Learning PTR, 2015.
- [6] 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.
- [7] Epic Games. Strona główna unreal engine. <https://www.unrealengine.com/en-US>. Dostęp 11.12.2022.
- [8] Andrew Sanders. *An introduction to Unreal engine 4*. AK Peters/CRC Press, 2016.
- [9] Antonín Šmíd. Comparison of unity and unreal engine. <https://core.ac.uk/download/pdf/84832291.pdf>, 2017. Dostęp 11.12.2022.
- [10] 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. Dostęp 11.12.2022.
- [11] Projekt godot na platformie github. <https://github.com/godotengine>. Dostęp 23.11.2022.
- [12] Juan Linietsky, Ariel Manzur and the Godot community. Godot docs. <https://docs.godotengine.org/en/3.5/>. Dostęp 12.11.2022.
- [13] Alexander Shvets. Design patterns: Singleton. <https://refactoring.guru/design-patterns/singleton>. Dostęp 23.11.2022.
- [14] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.
- [15] Lee Salzman. Dokumentacja biblioteki ENet. <http://enet.bespin.org/usergroup0.html>. Dostęp 11.12.2022.
- [16] Butch Wesley. Wiki projektu gut. <https://github.com/bitwes/Gut/wiki>. Dostęp 11.12.2022.
- [17] Blender Foundation. Strona główna projektu blender. <https://www.blender.org/>. Dostęp 11.12.2022.
- [18] Tony Mullen. *Mastering blender*. John Wiley & Sons, 2011.
- [19] Landfall Games. ROUNDS. <https://landfall.se/rounds>, 2021. Dostęp 11.12.2022.

- [20] Cranky Watermelon Pty Ltd. Boomerang Fu. <https://www.boomerangfu.com/>, 2020. Dostęp 11.12.2022.
- [21] Wargaming.net. World of Tanks. <https://worldoftanks.eu/>, 2009. Dostęp 11.12.2022.
- [22] Wii Play: Tanks! <https://nintendo.fandom.com/wiki/Tanks!>
- [23] 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.
- [24] Adam Kramarzewski, Ennio De Nucci. *Practical Game Design*. Packt, 2018.
- [25] Nathan Schuetz. Game feel: Input buffering. <https://barbariangrunge.com/game-feel-input-buffering/>, 2022. Dostęp 12.11.2022.
- [26] PlugWorld. Godot networked multiplayer shooter tutorial. <https://www.youtube.com/playlist?list=PL6bQeQE-ybqDmGuN7Nz4ZbTAqyCMyEHQa>, 2021. Dostęp 23.11.2022.
- [27] Erik Fagerholt, Magnus Lorentzon. Beyond the HUD. Master's thesis, Chalmers University of Technology,(Göteborg, Sweden), 2009.

## SPIS RYSUNKÓW

4.1	Menu startowe prototypu po wprowadzeniu systemu sieciowego . . . . .	25
5.1	Diagram przejść między widokami. . . . .	28
5.2	Projekt menu głównego. . . . .	28
5.3	Projekt menu hosta. . . . .	29
5.4	Projekt menu dołączającego gracza. . . . .	29
5.5	Projekt menu poczekalni. . . . .	30
5.6	Projekt interfejsu HUD. . . . .	30
5.7	Projekt menu wyboru kart. . . . .	31
5.8	Projekt menu oczekiwania między turami. . . . .	32
5.9	Projekt ekranu po zakończeniu rozgrywki. . . . .	32
5.10	Schemat procesu oddawania strzału. . . . .	34
6.1	Model czołgu. . . . .	40
6.2	Drzewo sceny postaci gracza. . . . .	42
6.3	Wyniki uruchomienia testów automatycznych. . . . .	45

## SPIS TABEL

3.1	Opis atrybutów postaci i pocisku . . . . .	18
3.2	Początkowy zbiór kart . . . . .	20
5.1	Schemat sterowania . . . . .	33